



# InterSystems SQL リファレンス

Version 2024.1  
2024-06-03

## InterSystems SQL リファレンス

InterSystems IRIS Data Platform Version 2024.1 2024-06-03

Copyright © 2024 InterSystems Corporation

All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble®, InterSystems HealthShare®, InterSystems IRIS®, および TrakCare は、InterSystems Corporation の登録商標です。HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, および InterSystems TotalView™ For Asset Management は、InterSystems Corporation の商標です。TrakCare は、オーストラリアおよび EU における登録商標です。

ここで使われている他の全てのブランドまたは製品名は、各社および各組織の商標または登録商標です。

このドキュメントは、インターシステムズ社(住所: One Memorial Drive, Cambridge, MA 02142)あるいはその子会社が所有する企業秘密および秘密情報を含んでおり、インターシステムズ社の製品を稼働および維持するためにのみ提供される。この発行物のいかなる部分も他の目的のために使用してはならない。また、インターシステムズ社の書面による事前の同意がない限り、本発行物を、いかなる形式、いかなる手段で、その全てまたは一部を、再発行、複製、開示、送付、検索可能なシステムへの保存、あるいは人またはコンピュータ言語への翻訳はしてはならない。

かかるプログラムと関連ドキュメントについて書かれているインターシステムズ社の標準ライセンス契約に記載されている範囲を除き、ここに記載された本ドキュメントとソフトウェアプログラムの複製、使用、廃棄は禁じられている。インターシステムズ社は、ソフトウェアライセンス契約に記載されている事項以外にかかるソフトウェアプログラムに関する説明と保証をするものではない。さらに、かかるソフトウェアに関する、あるいはかかるソフトウェアの使用から起こるいかなる損失、損害に対するインターシステムズ社の責任は、ソフトウェアライセンス契約にある事項に制限される。

前述は、そのコンピュータソフトウェアの使用およびそれによって起こるインターシステムズ社の責任の範囲、制限に関する一般的な概略である。完全な参照情報は、インターシステムズ社の標準ライセンス契約に記載され、そのコピーは要望によって入手することができる。

インターシステムズ社は、本ドキュメントにある誤りに対する責任を放棄する。また、インターシステムズ社は、独自の裁量にて事前通知なしに、本ドキュメントに記載された製品および実行に対する代替と変更を行う権利を有する。

インターシステムズ社の製品に関するサポートやご質問は、以下にお問い合わせください:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: [support@InterSystems.com](mailto:support@InterSystems.com)

# 目次

記号および構文規則 .....	1
InterSystems SQL で使用する記号 .....	2
構文規則 .....	7
SQL コマンド .....	9
ALTER FOREIGN SERVER (SQL) .....	10
ALTER FOREIGN TABLE (SQL) .....	12
ALTER ML CONFIGURATION (SQL) .....	14
ALTER MODEL (SQL) .....	16
ALTER TABLE (SQL) .....	18
ALTER USER (SQL) .....	28
ALTER VIEW (SQL) .....	30
BUILD INDEX (SQL) .....	34
CALL (SQL) .....	36
CANCEL QUERY (SQL) .....	41
CASE (SQL) .....	43
%CHECKPRIV (SQL) .....	46
CLOSE (SQL) .....	50
COMMIT (SQL) .....	52
CREATE AGGREGATE (SQL) .....	54
CREATE DATABASE (SQL) .....	58
CREATE FOREIGN SERVER (SQL) .....	60
CREATE FOREIGN TABLE (SQL) .....	62
CREATE FUNCTION (SQL) .....	68
CREATE INDEX (SQL) .....	75
CREATE METHOD (SQL) .....	84
CREATE ML CONFIGURATION (SQL) .....	90
CREATE MODEL (SQL) .....	93
CREATE PROCEDURE (SQL) .....	97
CREATE QUERY (SQL) .....	106
CREATE ROLE (SQL) .....	112
CREATE SCHEMA (SQL) .....	114
CREATE TABLE (SQL) .....	115
CREATE TABLE AS SELECT (SQL) .....	148
CREATE TRIGGER (SQL) .....	152
CREATE USER (SQL) .....	163
CREATE VIEW (SQL) .....	165
DECLARE (SQL) .....	173
DELETE (SQL) .....	176
DROP AGGREGATE (SQL) .....	186
DROP DATABASE (SQL) .....	187
DROP FOREIGN SERVER (SQL) .....	189
DROP FOREIGN TABLE (SQL) .....	190
DROP FUNCTION (SQL) .....	192
DROP INDEX (SQL) .....	194
DROP METHOD (SQL) .....	198
DROP ML CONFIGURATION (SQL) .....	200
DROP MODEL (SQL) .....	201

DROP PROCEDURE (SQL)	202
DROP QUERY (SQL)	204
DROP ROLE (SQL)	206
DROP SCHEMA (SQL)	208
DROP TABLE (SQL)	209
DROP TRIGGER (SQL)	213
DROP USER (SQL)	216
DROP VIEW (SQL)	218
EXPLAIN (SQL)	220
FETCH (SQL)	223
FREEZE PLANS (SQL)	227
GRANT (SQL)	229
INSERT (SQL)	238
INSERT OR UPDATE (SQL)	258
%INTRANSACTION (SQL)	266
JOIN (SQL)	267
LOAD DATA (SQL)	282
LOCK (SQL)	298
OPEN (SQL)	301
PURGE CACHED QUERIES (SQL)	302
REVOKE (SQL)	304
ROLLBACK (SQL)	310
SAVEPOINT (SQL)	313
SELECT (SQL)	316
SET ML CONFIGURATION (SQL)	340
SET OPTION (SQL)	341
SET TRANSACTION (SQL)	346
START TRANSACTION (SQL)	351
TRAIN MODEL (SQL)	357
TRUNCATE TABLE (SQL)	360
TUNE TABLE (SQL)	365
UNFREEZE PLANS (SQL)	368
UNLOCK (SQL)	370
UPDATE (SQL)	372
USE DATABASE (SQL)	387
VALIDATE MODEL (SQL)	389
SQL 節	393
DISTINCT (SQL)	394
FROM (SQL)	400
GROUP BY (SQL)	406
HAVING (SQL)	411
INTO (SQL)	419
ORDER BY (SQL)	424
TOP (SQL)	432
UNION (SQL)	437
VALUES (SQL)	443
WHERE (SQL)	446
WHERE CURRENT OF (SQL)	455
SQL の述語条件	457
述語の概要	458

ALL (SQL) .....	465
ANY (SQL) .....	467
BETWEEN (SQL) .....	468
EXISTS (SQL) .....	471
%FIND (SQL) .....	473
FOR SOME (SQL) .....	475
FOR SOME %ELEMENT (SQL) .....	478
IN (SQL) .....	482
%INLIST (SQL) .....	485
%INSET (SQL) .....	488
IS JSON (SQL) .....	490
IS NULL (SQL) .....	492
LIKE (SQL) .....	493
%MATCHES (SQL) .....	497
%PATTERN (SQL) .....	500
SOME (SQL) .....	503
%STARTSWITH (SQL) .....	504
SQL 集約関数 .....	511
集約関数の概要 .....	512
AVG (SQL) .....	517
COUNT (SQL) .....	521
%DLIST (SQL) .....	528
JSON_ARRAYAGG (SQL) .....	532
LIST (SQL) .....	536
MAX (SQL) .....	540
MIN (SQL) .....	543
STDDEV、STDDEV_SAMP、STDDEV_POP (SQL) .....	546
SUM (SQL) .....	548
VARIANCE、VAR_SAMP、VAR_POP (SQL) .....	551
XMLAGG (SQL) .....	554
SQL ウィンドウ関数 .....	559
ウィンドウ関数の概要 .....	560
AVG (SQL) .....	566
COUNT (SQL) .....	567
CUME_DIST() (SQL) .....	568
DENSE_RANK() (SQL) .....	569
FIRST_VALUE (SQL) .....	570
LAG (SQL) .....	571
LAST_VALUE (SQL) .....	572
LEAD (SQL) .....	573
MAX (SQL) .....	574
MIN (SQL) .....	575
NTH_VALUE (SQL) .....	576
NTILE (SQL) .....	577
PERCENT_RANK() (SQL) .....	578
RANK() (SQL) .....	579
ROW_NUMBER() (SQL) .....	580
SUM (SQL) .....	581
SQL 関数 .....	583

ABS (SQL) .....	584
ACOS (SQL) .....	586
ASCII (SQL) .....	587
ASIN (SQL) .....	588
ATAN (SQL) .....	589
ATAN2 (SQL) .....	590
CAST (SQL) .....	591
CEILING (SQL) .....	602
CHAR (SQL) .....	604
CHARACTER_LENGTH (SQL) .....	605
CHARINDEX (SQL) .....	607
CHAR_LENGTH (SQL) .....	610
COALESCE (SQL) .....	612
CONCAT (SQL) .....	615
CONVERT (SQL) .....	618
COS (SQL) .....	626
COT (SQL) .....	627
CURDATE (SQL) .....	628
CURRENT_DATE (SQL) .....	630
CURRENT_TIME (SQL) .....	632
CURRENT_TIMESTAMP (SQL) .....	634
CURTIME (SQL) .....	638
DATABASE .....	640
DATALength (SQL) .....	641
DATE (SQL) .....	642
DATEADD (SQL) .....	645
DATEDIFF (SQL) .....	651
DATENAME (SQL) .....	658
DATEPART (SQL) .....	662
DATE_TRUNC (SQL) .....	667
DAY (SQL) .....	671
DAYNAME (SQL) .....	672
DAYOFMONTH (SQL) .....	674
DAYOFWEEK (SQL) .....	677
DAYOFYEAR (SQL) .....	681
DECODE (SQL) .....	683
DEGREES (SQL) .....	686
%EXACT (SQL) .....	687
EXP (SQL) .....	689
%EXTERNAL (SQL) .....	691
\$EXTRACT (SQL) .....	693
\$FIND (SQL) .....	697
FLOOR (SQL) .....	700
GETDATE (SQL) .....	702
GETUTCDATE (SQL) .....	705
GREATEST (SQL) .....	708
hour (SQL) .....	710
IFNULL (SQL) .....	712
INSTR (SQL) .....	716
%INTERNAL (SQL) .....	718
ISNULL (SQL) .....	720

ISNUMERIC (SQL) .....	723
JSON_ARRAY (SQL) .....	725
JSON_OBJECT (SQL) .....	728
\$JUSTIFY (SQL) .....	731
LAST_DAY (SQL) .....	734
LAST_IDENTITY (SQL) .....	736
LCASE (SQL) .....	738
LEAST (SQL) .....	739
LEFT (SQL) .....	741
LEN (SQL) .....	742
LENGTH (SQL) .....	743
\$LENGTH (SQL) .....	746
\$LIST (SQL) .....	749
\$LISTBUILD (SQL) .....	753
\$LISTDATA (SQL) .....	756
\$LISTFIND (SQL) .....	758
\$LISTFROMSTRING (SQL) .....	760
\$LISTGET (SQL) .....	761
\$LISTLENGTH (SQL) .....	764
\$LISTSAME (SQL) .....	766
\$LISTTOSTRING (SQL) .....	768
LOG (SQL) .....	770
LOG10 (SQL) .....	771
LOWER (SQL) .....	772
LPAD (SQL) .....	773
LTRIM (SQL) .....	775
%MINUS (SQL) .....	776
MINUTE (SQL) .....	778
MOD (SQL) .....	780
MONTH (SQL) .....	782
MONTHNAME (SQL) .....	784
NOW (SQL) .....	786
NULLIF (SQL) .....	788
NVL (SQL) .....	790
%OBJECT (SQL) .....	793
%ODBCIN (SQL) .....	794
%ODBCOUT (SQL) .....	795
%OID (SQL) .....	796
PI (SQL) .....	797
\$PIECE (SQL) .....	798
%PLUS (SQL) .....	802
POSITION (SQL) .....	804
POWER (SQL) .....	806
PREDICT (SQL) .....	808
PROBABILITY (SQL) .....	810
QUARTER (SQL) .....	812
RADIANS (SQL) .....	814
REPEAT (SQL) .....	815
REPLACE (SQL) .....	816
REPLICATE (SQL) .....	818
REVERSE (SQL) .....	819

RIGHT (SQL) .....	821
ROUND (SQL) .....	822
RPAD (SQL) .....	825
RTRIM (SQL) .....	827
SEARCH_INDEX (SQL) .....	828
SECOND (SQL) .....	830
SIGN (SQL) .....	833
SIN (SQL) .....	835
SPACE (SQL) .....	836
%SQLSTRING (SQL) .....	837
%SQLUPPER (SQL) .....	840
SQRT (SQL) .....	843
SQUARE (SQL) .....	844
STR (SQL) .....	845
STRING (SQL) .....	846
STUFF (SQL) .....	848
SUBSTR (SQL) .....	850
SUBSTRING (SQL) .....	852
SYSDATE (SQL) .....	855
%SYSTEM_SQL.DefaultSchema() .....	856
TAN (SQL) .....	857
TIMESTAMPADD (SQL) .....	858
TIMESTAMPDIFF (SQL) .....	861
TO_CHAR (SQL) .....	864
TO_DATE (SQL) .....	873
TO_NUMBER (SQL) .....	880
TO_POSIXTIME (SQL) .....	883
TO_TIMESTAMP (SQL) .....	889
\$TRANSLATE (SQL) .....	896
TRIM (SQL) .....	898
TRUNCATE (SQL) .....	901
%TRUNCATE (SQL) .....	904
\$TSQL_NEWID (SQL) .....	906
UCASE (SQL) .....	907
UNIX_TIMESTAMP (SQL) .....	908
UPPER (SQL) .....	911
USER (SQL) .....	913
VECTOR_COSINE (SQL) .....	914
VECTOR_DOT_PRODUCT (SQL) .....	916
WEEK (SQL) .....	918
XMLCONCAT (SQL) .....	921
XMLELEMENT (SQL) .....	922
XMLFOREST (SQL) .....	926
YEAR (SQL) .....	929
SQL 単項演算子 .....	931
- (負の数) .....	932
+ (正の数) .....	933
SQL リファレンス資料 .....	935
Data Types (SQL) .....	936
日付/時刻文 (SQL) .....	964



既定のユーザ名とパスワード (SQL) .....	967
SQLCODEのエラー・コード .....	968
フィールド制約 .....	969
予約語 (SQL) .....	970
特殊変数 .....	972
文字列操作 (SQL) .....	974

# テーブル一覧

テーブル B-1:	95
テーブル C-1: SQL 等値比較述語	414
テーブル C-2: SQL 等値比較述語	450
テーブル C-3: SQL 部分文字列述語	451
テーブル D-1: LIKE ワイルドカード文字	493
テーブル G-1: \$HOROLOGY の日付と時刻の形式	646
テーブル G-2: 日付形式	647
テーブル G-3: 時刻形式	647
テーブル G-4: \$HOROLOGY の日付と時刻の形式	652
テーブル G-5: 日付形式	653
テーブル G-6: 時刻形式	654
テーブル G-7:	667
テーブル G-8: \$HOROLOGY の日付と時刻の形式	668
テーブル G-9: 日付形式	668
テーブル G-10: 時刻形式	669
テーブル G-11: 日付形式	866
テーブル G-12: 時刻形式	867
テーブル G-13: 数の形式	867

# 記号および構文規則

## InterSystems SQL で使用する記号

InterSystems SQL で演算子として使用される文字のテーブル。

### 記号のテーブル

以下は、InterSystems IRIS® データ・プラットフォーム上の InterSystems SQL で使用されるリテラル記号です。(このリストには、言語の一部ではない[形式規約](#)を示す記号は含まれていません)。別途、“[ObjectScript で使用する記号](#)”のテーブルも用意されています。

各記号の名前の後には、ASCII 10 進数コード値が続きます。

記号	名前と使用法
[空白]、または [タブ]	空白 (タブ (9) またはスペース (32)) : キーワード、識別子、および変数の間の 1 つ以上の空白文字。
!	感嘆符 (33) : 条件式の中の述語間の <a href="#">OR 論理演算子</a> 。WHERE 節や HAVING 節などで使用します。  SQL シェルでは、ObjectScript コマンドラインの発行に <a href="#">! command</a> が使用されます。
!=	感嘆符/等号 : <a href="#">比較条件</a> に等しくない。
"	引用符 (34) : <a href="#">区切り文字付き識別子名</a> を囲みます。  <a href="#">ダイナミック SQL</a> では、クラス・メソッド引数のリテラル値を囲むために使用されます (%Prepare() メソッドの文字列引数としての SQL コードや %Execute() メソッドの文字列引数としての入力パラメータなど)。  <a href="#">%PATTERN</a> では、パターン文字列内でリテラル値を囲むために使用されます。例 : '3L1"L".L' (3 つの小文字の後に 1 つの大文字の “L” が続き、その後に任意数の小文字が続くことを示します)。  <a href="#">XMLELEMENT</a> では、タグ名の文字列リテラルを囲むために使用されます。
""	2 つの引用符 : それ自体では、無効な <a href="#">区切り識別子</a> 。 <a href="#">区切り識別子</a> 内では、リテラル引用符のエスケープ・シーケンスです。例 : "a" "good" "id"。
#	シャープ記号 (35) : 有効な <a href="#">識別子名</a> の文字 (最初の文字ではありません)。  前後にスペースを持つモジュール <a href="#">算術演算子</a> です。  埋め込み SQL では、ObjectScript の <a href="#">マクロ・プリプロセッサ指示文</a> の接頭語です。例 : #include。  SQL シェルでは、SQL シェルの履歴バッファから文を呼び出すのに <a href="#"># command</a> が使用されます。
\$	ドル記号 (36) : 有効な <a href="#">識別子名</a> の文字 (最初の文字ではありません)。  InterSystems IRIS 拡張の SQL 関数の最初の文字です。
\$\$	二重ドル記号 : ObjectScript ユーザ定義関数 (外部関数とも呼ばれます) を呼び出すのに使用します。詳細は、SELECT のリファレンス・ページで selectItem 引数の <a href="#">関数およびメソッド呼び出しによる選択</a> を参照してください。

記号	名前と使用法
%	<p>パーセント記号 (37) : 識別子名の最初の有効な文字 (最初の文字のみ)。</p> <p>文字列照合関数 (%SQLUPPER)、集約関数 (%DLIST)、述語条件 (%STARTSWITH) を含む、SQL 標準に対する一部の InterSystems SQL 拡張の最初の文字。</p> <p>SELECT 内のキーワード %ID、%TABLENAME、および %CLASSNAME の最初の文字。</p> <p>一部の特権キーワード (%CREATE_TABLE、%ALTER) および一部のロール名 (%All) の最初の文字。</p> <p>一部の埋め込み SQL システム変数 (%ROWCOUNT、%msg) の最初の文字。</p> <p>データ型の最大長インジケータ。(例) CHAR(%24)</p> <p>LIKE 条件述語の複数文字ワイルドカード。</p>
%%	<p>二重パーセント記号 : 擬似フィールド参照変数キーワードの接頭語。(例) %%CLASSNAME、%%CLASSNAMEQ、%%ID、%%TABLENAME。ObjectScript の計算フィールド・コードおよびトリガ・コードで使用されます。</p>
&	<p>アンパサンド (38) : WHERE 節およびその他の条件式の AND 論理演算子。</p> <p>\$BITLOGIC ビット文字列 And 演算子。</p> <p>埋め込み SQL 呼び出し接頭語。(例) &amp;sql(SQL commands)</p>
'	<p>一重引用符 (39) : 文字列リテラルを囲みます。</p>
''	<p>二重の一重引用符 : 空の文字列リテラルを囲みます。</p> <p>文字列値内ではリテラル一重引用符のエスケープ・シーケンスです。(例) 'can't'</p>
( )	<p>括弧 (40,41) : コンマで区切られたリストを囲みます。SQL 関数の引数を囲みます。プロシージャ、メソッド、またはクエリでのパラメータ・リストを囲みます。多くの場合、引数またはパラメータが提供されない場合でも、括弧を指定する必要があります。</p> <p>SELECT DISTINCT BY 節で、一意の値を選択するために使用される項目または項目のコンマ区切りのリストを囲みます。</p> <p>SELECT 文で、FROM 節内のサブクエリを囲みます。UNION で使用される事前定義済みクエリの名前を囲みます。</p> <p>ホスト変数配列添え字を囲みます。例えば INTO :var(1),:var(2) です。</p> <p>embedded SQL コードを囲みます。(例) &amp;sql( code )</p> <p>算術演算で優先順位を指定するために使用されます。(例) 3+(3*5)=18。述語をグループ化するために使用されます。(例) WHERE NOT (Age&lt;20 AND Age&gt;12)。</p>
(( ))	<p>二重の括弧 : クエリ・キャッシュでリテラル置換を抑制します。例えば、SELECT TOP ((4)) Name FROM Sample.Person WHERE Name %STARTSWITH (('A')) のようになります。NULL 以外の異常値の WHERE 節による選択を最適化します。</p>

記号	名前と使用法
*	<p>アスタリスク (42) : 以下の場合で、“すべて”を示すワイルドカード。<a href="#">SELECT</a> では、すべての列を検索します。(例) <a href="#">SELECT * FROM</a> テーブル。<a href="#">COUNT</a> で、すべての行 (NULL と重複を含む) をカウントします。<a href="#">GRANT</a> および <a href="#">REVOKE</a> で、すべての特権、すべてのテーブル、または現在定義されているすべてのユーザ。</p> <p><a href="#">%MATCHES</a> パターン文字列で、複数文字ワイルドカード。</p> <p>乗算算術演算子。</p>
*/	アスタリスク・スラッシュ : <a href="#">複数行コメント</a> の末端文字。コメントは、/* で開始します。
+	プラス記号 (43) : 加算算術演算子。単項正符号演算子
,	<p>コンマ (44) : リストの区切り記号。例えば、複数のフィールド名を区切る場合に使用します。</p> <p><a href="#">データ・サイズ定義</a>内にあります。(例) <a href="#">NUMERIC (precision,scale)</a></p>
-	<p>ハイフン (マイナス記号) (45) : 減算算術演算子。単項負符号演算子</p> <p>SQLCODE <a href="#">エラー・コード</a>接頭語です。(例) -304</p> <p><a href="#">日付区切り文字</a>です。</p> <p><a href="#">%MATCHES</a> パターン文字列で、各括弧内で指定されるレンジ・インジケータ。例えば、[a-m] のように指定します。</p>
--	二重ハイフン : 単一の行の <a href="#">コメント</a> 文字。
->	ハイフン、より大きい (矢印) : <a href="#">暗黙結合</a> 矢印構文。
.	<p>ピリオド (46) : 修飾されている<a href="#">テーブル名</a>の schema.tablename や<a href="#">列名</a>の tablealias.fieldname など、マルチパート名の各部分を区切るために使用されます。</p> <p>アメリカの数値形式では、<a href="#">数値リテラル</a>の小数点です。</p> <p>ロシア、ウクライナ、およびチェコのロケールでは、<a href="#">日付区切り文字</a>です。DD.MM.YYYY のように使用されます。</p> <p><a href="#">参照渡し</a>を指定する変数または配列名の接頭語。(例) .name</p> <p><a href="#">%PATTERN</a> パターン文字列の複数文字ワイルドカード。</p>
/	<p>スラッシュ (47) : 除算算術演算子。</p> <p><a href="#">日付区切り文字</a>です。</p>
/*	スラッシュ・アスタリスク : <a href="#">複数行コメント</a> の開始文字。コメントは、*/ で終了します。
:	<p>コロン (58) : <a href="#">ホスト変数</a>の指示接頭語。(例) :var</p> <p>時、分、秒の<a href="#">時間区切り文字</a>です。CAST 関数と CONVERT 関数では、オプションの <a href="#">1000 分の 1 秒の区切り文字</a>です。</p> <p><a href="#">トリガ・コード</a>では、ObjectScript ラベル行を示す接頭語です。</p> <p>CREATE PROCEDURE ObjectScript コード本文では、<a href="#">マクロ・プリプロセッサ指示文</a>の接頭語です。例 : :#include。</p>
::	二重コロン : <a href="#">トリガ・コード</a> では、この二重接頭語は、その行の最初にある識別子 (::name) がラベル行ではなくホスト変数であることを示します。

記号	名前と使用法
;	セミコロン (59): プロシージャ、メソッド、クエリ、およびトリガ・コード内の SQL 文末区切り文字。 ImportDDL() で、または TSQL 言語を使用して SQL コードを指定している場所であればどこでも、オプションの文末区切り文字として受け入れられます。それ以外の場合、InterSystems SQL は SQL 文の最後にセミコロンを使用せず、許容しません。
<	より小さい (60): 比較条件より小さい。
<=	以下: 比較条件以下。
<>	より小さい/より大きい: 比較条件に等しくない。
=	等号 (61): 比較条件に等しい。 WHERE 節では、内部結合です。
>	より大きい (62): 比較条件より大きい。
>=	以上: 比較条件以上。
?	疑問符 (63): ダイナミック SQL 内で Execute メソッドによって入力される入力パラメータ変数。 %MATCHES パターン文字列で、単一文字ワイルドカード。 SQL シェルでは、? command により SQL シェル・コマンドのヘルプ・テキストが表示されます。
@	アット記号 (64): 有効な識別子名の文字 (最初の文字ではありません)。
E, e	文字 “E” (69, 101): 指数文字。 任意の表示可能文字を指定する %PATTERN のコード。
[	角括弧 (開始) (91): Contains 述語。WHERE 節や HAVING 節などで使用します。
[ ]	開および閉角括弧: %MATCHES パターン文字列で、マッチング文字列のリストまたは範囲を囲みます。例: [abc]、[a-m]。
¥	バックスラッシュ (92): 整数除算算術演算子。 %MATCHES パターン文字列で、エスケープ文字。
]	角括弧 (終了) (93): Follows 述語。WHERE 節や HAVING 節などで使用します。
^	キャレット (94): %MATCHES パターン文字列で、NOT の文字。例: [^abc]。
_	アンダースコア (95): 有効な識別子名の最初 (またはそれ以降) の文字。特定のユーザ名 (パスワードではなく) の有効な最初の文字です。 埋め込みシリアル・クラス of データを表すために列名で使用します。SELECT Home_State では、Home はシリアル・クラスを参照するフィールドで、State はそのシリアル・クラスで定義されているプロパティです。 LIKE 条件の述語の単一文字ワイルドカード。
{ }	中括弧 (123,125): ODBC スカラ関数を囲みます。(例) {fn name(...) }。時刻/日付文の関数を囲みます。(例) {d 'string'}、{t 'string'}、{ts 'string'} プロシージャ、メソッド、クエリ、およびトリガ・コード内で ObjectScript コードを囲みます。

記号	名前と使用法
	<p>二重の垂直バー (124) : <a href="#">連結演算子</a>。</p> <p>複合 ID インジケータ。InterSystems IRIS で、生成された複合オブジェクト ID (連結 ID) 内にある複数のプロパティ間の区切り文字として使用されます。これは、複数のプロパティに対して定義された <a href="#">IDKey インデックス</a> (prop1   prop2)、または <a href="#">親子リレーションシップの ID</a> (parent   child) になります。IDKEY フィールドのデータには使用できません。</p>



# 構文規則

InterSystems SQL リファレンスで使用する規則を指定します。

## 概要

以下は、このリファレンスで使用する形式規約です。これらの形式文字は、使用法を説明するもので、SQL プログラムのコーディングの際に指定するものではありません。SQL コーディングで使用する記号のテーブルは、“[SQL 記号](#)”のテーブルを参照してください。

記号	意味
[ nnnn ]	角括弧で囲まれた引数はオプションです。何も指定しないか、1 つ指定します。
{ nnnn }	中括弧で囲まれた引数はオプションで、複数回繰り返して使用できます。何も指定しないか、1 つあるいは複数指定します。  中括弧は、{fn FUNCTION(arg)} の形式の ODBC スカラ関数などでリテラル文字としても使用されます。
mmmm   nnnn	垂直バーは OR を意味します。1 つまたは複数指定します。
...	省略記号は、完全な SQL 文の未指定部分です。var1,var2,... のような繰り返しの指定にも使用できます。
::=	これは以下と同等であることを意味します。

引数が “item-list” として示される場合、引数は特定の文字で区切った 1 つまたは複数の特定の item で構成できます。item-list からの相互参照は、item のページ自体を指します。

引数が “item-commalist” として示される場合、引数はコンマで区切った 1 つまたは複数の特定の item で構成できます。item-commalist からの相互参照は、item のページ自体を指します。

項目が [( identifier )] のように、角括弧 [] 付きの丸括弧内に示される場合、ペア構成 (単位) の丸括弧 () はオプションです。



# SQL コマンド

## ALTER FOREIGN SERVER (SQL)

外部サーバ定義を変更します。

### 構文

#### 接続の変更

```
ALTER [ FOREIGN ] SERVER server-name
    ALTER CONNECTION jdbc-connection
```

```
ALTER [ FOREIGN ] SERVER server-name
    ALTER HOST file-path
```

#### 区切り識別子の変更

```
ALTER [ FOREIGN ] SERVER server-name ALTER id-option
```

#### 接続と区切り識別子の変更

```
ALTER [ FOREIGN ] SERVER server-name
    MODIFY [ CONNECTION jdbc-connection | HOST file-path ],
    id-option
```

### 引数

引数	説明
<i>server-name</i>	変更する外部サーバ定義の名前。有効な識別子を指定します。
CONNECTION <i>cxn-name</i>	InterSystems IRIS を外部データ・ソースに接続する新しい JDBC 接続の名前。有効な識別子を指定します。定義済みの JDBC 接続の名前とする必要があります。それぞれを区切って記述します。
HOST <i>file-path</i>	InterSystems IRIS に投影するファイルへのアクセスで使用する新しいファイル・パス。
<i>id-option</i>	DELIMITEDIDS または NODELIMITEDIDS を入力します。外部データ・ソースが区切り識別子を受け入れるかどうかに基づいて動作を設定します。

### 説明

ALTER FOREIGN SERVER コマンドを使用すると、外部サーバをどのように外部データ・ソースに接続するかを変更できます。このコマンドの ALTER バリエントで 1 つのパラメータを変更でき、MODIFY バリエントで複数のパラメータを変更できます。特に、外部サーバを外部ソースに接続するときに使用するファイル・パス、JDBC 接続、または区切り識別子のオプションを変更できます。

CONNECTION プロパティまたは HOST プロパティを使用して外部サーバの接続パラメータを変更する前に、その外部サーバに定義した外部テーブルへのアクセスが、この変更の影響を受けないことを確認する必要があります。例えば、HOST のファイル・パスを変更しても、定義済みのテーブルに引き続きアクセスできるようにする場合は、外部テーブルに関連付けた .csv ファイルをすべて新しいファイル・パスへ移動します。適切に変更しない限り、これらのテーブルのデータにはアクセスできなくなります。ALTER FOREIGN SERVER を使用して、外部テーブルを定義していない外部サーバでこれらのパラメータを変更するのであれば、このような懸念はありません。

### 例

以下の例では、外部サーバのファイル・パスを変更して、別のディレクトリからデータを読み取ります。

```
ALTER FOREIGN SERVER Sample.Test ALTER HOST '/second/filepath'
```

以下の例では、外部サーバの JDBC 接続を変更して、別のデータベース・ソースからデータを読み取ります。この外部データ・ソースが区切り識別子を許可することを指定しています。

```
ALTER FOREIGN SERVER Sample.Test MODIFY CONNECTION 'anotherConnection', DELIMITEDIDS
```

## 関連項目

- ・ [CREATE FOREIGN SERVER](#)
- ・ [DROP FOREIGN SERVER](#)
- ・ [CREATE FOREIGN TABLE](#)

## ALTER FOREIGN TABLE (SQL)

外部テーブル定義を変更します。

### 構文

#### 列名の変更

```
ALTER FOREIGN TABLE table-name ALTER [ COLUMN ] old-name
    RENAME new-name
ALTER FOREIGN TABLE table-name MODIFY old-name
    RENAME new-name, old-name2 RENAME new-name2, ...
ALTER FOREIGN TABLE table-name ALTER [ COLUMN ] old-name
    RENAME new-name VALUES ( external-name )
ALTER FOREIGN TABLE table-name MODIFY old-name
    RENAME new-name, old-name2 RENAME new-name2, ...
    VALUES ( newexternal-name, newexternal-name2, ... )
```

#### データ型の変更

```
ALTER FOREIGN TABLE table-name ALTER col-name datatype
ALTER FOREIGN TABLE table-name MODIFY col-name datatype
    {, col-name datatype ...}
```

### 引数

引数	説明
table-name	変更する外部テーブルの名前。有効な識別子を指定します。このコマンドを発行する前に、この引数値が、外部サーバに存在する外部テーブルの名前であることを確認します。
old-name	InterSystems IRIS で変更する現在の列名。有効な識別子を指定します。このコマンドを発行する前に、この引数値が、外部テーブルに存在する列の名前に対応していることを確認します。
new-name	InterSystems IRIS での新しい列名。有効な識別子を指定します。
external-name	RENAME 節で対応する列にデータを投影する、外部データ・ソースの列の新しい名前。
col-name	新しいデータ型に変換される列の名前。有効な識別子を指定します。このコマンドを発行する前に、この引数値が、外部テーブルに存在する列の名前に対応していることを確認します。
datatype	列の新しいデータ型。有効な <a href="#">SQL データ型</a> であることが必要です。

### 説明

ALTER FOREIGN TABLE コマンドは、外部テーブル定義を変更します。目的のテーブルに適用できる変更のタイプとして次の 2 つがあります。

- ・ 1 つの列の名前または列のリストにある列の名前を変更します。
- ・ 列のデータ型または列のリストにある列のデータ型を変更します。

#### 列名の変更

ALTER FOREIGN TABLE コマンドを使用して、外部テーブルにある 1 つの列の名前または外部テーブルの列のリストにある列の名前を変更できます。

以下の 2 つのバリエーションがあります。

- ALTER FOREIGN TABLE table-name ALTER [ COLUMN ] old-name RENAME new-name : 外部テーブルにある 1 つの列の名前を old-name から new-name に変更します。
- ALTER FOREIGN TABLE table-name MODIFY old-name RENAME new-name : 外部テーブルにある 1 つ以上の列の名前を、それぞれの old-name からそれに対応する new-name に変更します。
- ALTER FOREIGN TABLE table-name ALTER [ COLUMN ] old-name RENAME new-name VALUES ( external-name ) : 外部テーブルにある 1 つの列の名前を old-name から new-name に変更します。このバリエーションでは、指定した列にデータを投影する、外部データ・ソースの列の名前も変更されます。
- ALTER FOREIGN TABLE table-name MODIFY old-name RENAME new-name, old-name2 RENAME new-name2 VALUES ( external-name, external-name2 ) : 外部テーブルにある一連の列の名前を、old-name からそれに対応する new-name に変更します。このバリエーションでは、指定した複数列にデータを投影する、外部データ・ソースの複数列の名前も変更されます。

**注釈** ALTER FOREIGN TABLE コマンドで 1 つまたは複数の列の名前を変更することはお勧めしません。外部テーブルは他のソースにあるデータの投影にすぎません。したがって、外部データ・ソースの内容を大幅に変更する場合は、このコマンドを使用するのではなく、外部テーブルを削除し、データベースまたは .csv ファイルを編集したうえで外部テーブルを再作成します。

## 列のデータ型の変更

ALTER FOREIGN TABLE コマンドを使用して、外部テーブルにある 1 つの列のデータ型または外部テーブルの列のリストにある列のデータ型を変換できます。新しいデータ型は、InterSystems SQL の有効なデータ型とする必要があります。

この変更によってストリーム・データ型がストリームではないデータ型になる場合、またはストリームではないデータ型がストリーム・データ型になる場合、その列のデータ型は変更できません。そのようにしようとすると SQLCODE -374 エラーが返されます。

以下の 2 つのバリエーションがあります。

- ALTER FOREIGN TABLE table-name ALTER col-name datatype : 1 つの列のデータ型を変更します。
- ALTER FOREIGN TABLE table-name MODIFY col-name datatype {, col-name datatype ...} : 1 つ以上の列のデータ型を変更します。列ごとに異なるデータ型を指定できます。

## 例

以下の例では、外部テーブル Sample.Person の LastName 列の名前を変更します。このコマンドの ALTER 形式と MODIFY 形式の両方を使用しています。

```
ALTER FOREIGN TABLE Sample.Person ALTER COLUMN LastName RENAME Surname
ALTER FOREIGN TABLE Sample.Person MODIFY LastName RENAME FamilyName, FirstName RENAME GivenName
```

以下の例では、外部テーブル Sample.Account の Amount 列のデータ型を変更します。このコマンドの ALTER 形式と MODIFY 形式の両方を使用しています。

```
ALTER FOREIGN TABLE Sample.Person ALTER Amount INTEGER
ALTER FOREIGN TABLE Sample.Person MODIFY Amount INTEGER
```

## 関連項目

- [CREATE FOREIGN TABLE](#)
- [DROP FOREIGN TABLE](#)

## ALTER ML CONFIGURATION (SQL)

ML 構成を変更します。

### 構文

```
ALTER ML CONFIGURATION ml-configuration-name
  [ PROVIDER provider-name ] [ %DESCRIPTION description ]
  [ USING json-object-string ] [ provider-connection-settings ]
```

### 引数

<i>ml-configuration-name</i>	変更する ML 構成の名前。
PROVIDER <i>provider-name</i>	機械学習プロバイダの名前を指定する文字列。値は次のとおりです。 <ul style="list-style-type: none"> <li>AutoML</li> <li>H2O</li> <li>DataRobot</li> <li>PMML</li> </ul>
%DESCRIPTION <i>description</i>	オプション - 文字列。ML 構成の説明テキスト。この後の <a href="#">説明</a> を参照してください。
USING <i>json-object-string</i>	オプション - 1 つ以上のキーと値のペアを指定する JSON 文字列。この後の <a href="#">説明</a> を参照してください。
<i>provider-connection-settings</i>	接続に必要な追加の設定。機械学習プロバイダによって異なります。この後の <a href="#">説明</a> を参照してください。

### 説明

ALTER ML CONFIGURATION 文は、ML 構成定義内の 1 つまたは複数のパラメータを変更します。以下を変更できます。

- ・ [プロバイダ](#)
- ・ [説明](#)
- ・ [USING 節](#)
- ・ [プロバイダの接続設定](#)

### ML 構成の説明

%DESCRIPTION は一重引用符で囲まれたテキスト文字列を受け入れます。これを使用して、構成を文書化するための説明を入力できます。このテキストの長さに制限はなく、空白スペースを含むすべての文字を使用できます。

### USING

構成の既定の USING 節を指定できます。この節は、1 つ以上のキーと値のペアで構成される JSON 文字列を受け入れます。TRAIN MODEL を実行すると、既定では構成の USING 節が使用されます。

```
ALTER ML CONFIGURATION MyConfiguration USING {"seed": 3}
```

指定するパラメータが、選択するプロバイダによって認識されることを確認する必要があります。これを行わないと、トレーニング時にエラーが発生することがあります。



## プロバイダの接続設定

構成で指定されたプロバイダによっては、正常な接続を確立するために、いくつかの追加フィールドに入力しなければならないことがあります。

### DataRobot

DataRobot に正常に接続するには、以下の値を指定する必要があります。

- ・ URL [ = ] *url-string* - *url-string* は DataRobot エンドポイントの URL です。
- ・ APITOKEN [ = ] *token-string* - *token-string* は DataRobot AutoML サーバにアクセスするためのクライアント API トークンです。

DataRobot の ML 構成の変更は、次のようなクエリで実行できます。

```
ALTER ML CONFIGURATION datarobot-configuration URL url-string APITOKEN token-string
```

*url-string* と *token-string* に適切な値を指定します。

### 必要なセキュリティ特権

ALTER ML CONFIGURATION を呼び出すには、%ALTER\_ML\_CONFIGURATION 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%ALTER\_ML\_CONFIGURATION 特権を割り当てるには、[GRANT](#) コマンドを使用します。

### 例

以下の SQL クエリは TestH2O という既存の構成を編集して、トレーニングするすべてのモデルに使用する USING 節を追加します。

```
ALTER ML CONFIGURATION TestH2O USING {"seed": 2}
```

### 関連項目

- ・ [CREATE ML CONFIGURATION](#)、[DROP ML CONFIGURATION](#)

## ALTER MODEL (SQL)

モデルを変更します。

### 構文

```
ALTER MODEL model-name PURGE [ ALL ] [ integer DAYS ]
```

```
ALTER MODEL model-name DEFAULT [ TRAINED MODEL ] trained-model-name
```

### 引数

model-name	変更する機械学習モデルの名前。
DEFAULT trained-model-name	トレーニングされた機械学習モデル。
integer DAYS	整数。

### 説明

ALTER MODEL 文は機械学習モデルを変更します。1 つの ALTER MODEL 文で実行できる処理は 1 種類のみです。

- ・ PURGE は、指定されたスコープに基づいて、関連するモデルのすべてのトレーニング実行と検証実行を削除します。
  - － スコープが指定されていない場合、既定のトレーニング済みモデルに関連するものを除き、すべてのレコードが削除されます。
  - － integer DAYS が指定されている場合、integer 日以上経過したすべてのレコードが削除されます。
  - － ALL が指定されている場合、いつ発生したかに関係なく、すべてのレコードが削除されます。
- ・ DEFAULT (または DEFAULT TRAINED MODEL) は、既定のトレーニング済みモデルを指定されたモデルに設定します。これは、同じモデル定義を使用して複数の TRAIN MODEL 文を作成し、各トレーニング済みモデルを別の名前で作成し、既定の名前が指すモデルを切り替える場合に便利です。存在しないモデルを指定すると、エラーが発生します。

### 必要なセキュリティ特権

ALTER MODEL を呼び出すには、%MANAGE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%MANAGE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

### 例

以下のクエリは、PURGE 節を使用して、SpamFilter モデルのすべてのトレーニング実行データと検証実行データを削除します。

```
ALTER MODEL SpamFilter PURGE ALL
```

以下のクエリは、DEFAULT 節を使用して、既定のトレーニング済みモデルの SpamFilter を SpamFilter3 に変更します。

```
ALTER MODEL SpamFilter DEFAULT SpamFilter3
```

## 関連項目

- ・ [CREATE MODEL、DROP MODEL](#)

## ALTER TABLE (SQL)

テーブルを変更します。

### 構文

ALTER TABLE *table* *alter-action*

where *alter-action* is one of the following:

```
ADD [(add-action {,add-action} [ ])] |
DROP [COLUMN] drop-column-action {,drop-column-action} |
DROP drop-action |
DELETE drop-action |
ALTER [COLUMN] field alter-column-action |
MODIFY modification-spec {,modification-spec}
RENAME table
```

*add-action* ::=

```
[CONSTRAINT identifier]
[(FOREIGN KEY (field-commalist)
  REFERENCES table (field-commalist)
  [ON DELETE ref-action] [ON UPDATE ref-action]
  [NOCHECK] [ ])]
|
[(UNIQUE (field-commalist) [ ])]
|
[(PRIMARY KEY (field-commalist) [ ])]
|
DEFAULT [(default-spec [ ])] FOR field
|
[COLUMN] [(field datatype [sqlcollation]
  [%DESCRIPTION string]
  [DEFAULT [(default-spec [ ])] ]
  [ON UPDATE update-spec ]
  [UNIQUE] [NOT NULL]
  [REFERENCES table (field-commalist)
  [ON DELETE ref-action] [ON UPDATE ref-action]
  [NOCHECK] ]
  [ ])]
```

*drop-column-action* ::=

```
[COLUMN] field [RESTRICT | CASCADE] [%DELDATA | %NODELDATA]
```

*drop-action* ::=

```
FOREIGN KEY identifier |
PRIMARY KEY |
CONSTRAINT identifier |
```

*alter-column-action* ::=

```
RENAME newfieldname |
datatype |
[SET] DEFAULT [(default-spec [ ])] | DROP DEFAULT |
NULL | NOT NULL |
COLLATE sqlcollation
```

*modification-spec* ::=

```
oldfieldname RENAME newfieldname |
field [datatype]
  [DEFAULT [(default-spec [ ])] ]
  [CONSTRAINT identifier] [NULL] [NOT NULL]
```

*sqlcollation* ::=

```
{ %EXACT | %MINUS | %MVR | %PLUS | %SPACE |
  %SQLSTRING [(maxlen)] | %SQLUPPER [(maxlen)] |
  %TRUNCATE[(maxlen)] }
```

## 引数

引数	説明
table	変更する <b>テーブル</b> の名前。テーブル名は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、 <b>既定のスキーマ名</b> が使用されます。スキーマ検索パスの値は使用されません。
identifier	制約に割り当てられた一意の名前。有効な <b>識別子</b> である必要があります。
field	変更 (追加、修正、削除) する列の名前。有効な <b>識別子</b> である必要があります。
field-commalist	列、またはコンマ区切りの列のリストの名前。field-commalist は、列が 1 つしか指定されていない場合でも括弧で囲む必要があります。“ <b>識別子</b> ”を参照してください。
datatype	InterSystems SQL の有効なデータ型。“ <b>Data Types (SQL)</b> ”を参照してください。
default-spec	このフィールドがユーザ指定のデータ値でオーバーライドされない場合に、このフィールドに自動的に入力される既定のデータ値。使用できる値は、リテラル値、以下のキーワード・オプションのいずれか (NULL、USER、CURRENT_USER、SESSION_USER、SYSTEM_USER、CURRENT_DATE、CURRENT_TIME、および CURRENT_TIMESTAMP)、または OBJECTSCRIPT 式です。既定値として <b>長さゼロの SQL 文字列</b> は使用しないでください。詳細は、“ <b>CREATE TABLE</b> ”を参照してください。
update-spec	“CREATE TABLE”の“ <b>ON UPDATE</b> ”を参照してください。
COLLATE sqlcollation	オプション - SQL 照合タイプ (%EXACT、%MINUS、%PLUS、%SPACE、%SQLSTRING、%SQLUPPER、%TRUNCATE、または%MVR) のいずれかを指定します。既定は、 <b>ネームスペースの既定の照合</b> です (変更していない場合は %SQLUPPER です)。%SQLSTRING、%SQLUPPER、および %TRUNCATE は、オプションの最大長のトランケーション引数である、括弧で囲んだ整数を指定できます。これらの照合パラメータ・キーワードの先頭のパーセント記号 (%) はオプションです。COLLATE キーワードはオプションです。詳細は、“ <b>テーブルのフィールド/プロパティ定義の照合</b> ”を参照してください。

## 概要

ALTER TABLE 文は、要素の追加や削除または既存の要素の修正を行い、テーブル定義を変更します。1 つの ALTER TABLE 文で実行できる処理は 1 種類のみです。

- ・ RENAME では、**テーブルの名前を変更**したり、ALTER COLUMN または MODIFY の構文を使用してテーブルの既存の列の名前を変更したりできます。
- ・ ADD は、複数の列および/または制約をテーブルに追加できます。ADD キーワードは一度指定し、その後にコンマ区切りリストを指定します。コンマ区切りリストを使用して、**複数の新しい列をテーブルに追加**したり、**制約のリストを既存の列に追加**したり、既存の列に新しい列と制約の両方を追加したりすることができます。
- ・ DROP COLUMN は、**テーブルから複数の列を削除**できます。DROP キーワードは一度指定し、その後にそれぞれオプションのカスケードおよび/またはデータ削除オプションを伴う列のコンマ区切りリストを指定します。
- ・ ALTER COLUMN では、**単一の列の定義を変更**できます。複数の列を変更することはできません。
- ・ MODIFY では、**単一の列またはコンマ区切りの列のリストの定義を変更**できます。ALTER COLUMN で指定されたオプションの一部はサポートされません。
- ・ DROP では、**単一のフィールドまたは複数のフィールドのグループから制約を削除**できます。DROP は単一の制約でのみ機能します。

ALTER TABLE DROP キーワードと ALTER TABLE DELETE キーワードは同義語です。

指定のテーブルが現在のネームスペースに存在するかどうかを確認するには、`$SYSTEM.SQL.Schema.TableExists()` メソッドを使用します。

## 特権とロック

ALTER TABLE コマンドは特権を必要とする操作です。ALTER TABLE を実行するには、ユーザは `%ALTER_TABLE` [管理特権](#) を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ ” 'name' ” に設定されます。

ユーザは、指定されたテーブルに対する `%ALTER` 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対する `%ALTER` 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する `%ALTER` 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ ” 'name' ” 'Schema.TableName' ” `%ALTER` ” に設定されます。

現在のユーザが `%ALTER` 特権を持っているかどうかを確認するには、[%CHECKPRIV](#) コマンドを呼び出します。指定のユーザが `%ALTER` 特権を持っているかどうかを確認するには、`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを呼び出します。

必要な管理特権を割り当てるには、`%ALTER_TABLE` 特権で GRANT コマンドを使用します。これには適切な付与特権が必要です。`%ALTER` オブジェクト特権を割り当てるには、以下を使用できます。

- GRANT コマンドと `%ALTER` 特権。これには適切な付与特権が必要です。
- ロールまたはユーザを編集するためページの管理ポータル [の \[SQL Tables\] タブ](#) にある、目的のテーブルの [\[ALTER\] チェック・ボックス](#)。これには適切な付与特権が必要です。

埋め込み SQL では、以下のように `$SYSTEM.Security.Login()` メソッドを使用して適切な特権を持ったユーザとしてログインできます。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

`$SYSTEM.Security.Login` メソッドを呼び出すには、`%Service_Login:Use` 特権が必要です。詳細は、[“%SYSTEM.Security”](#) を参照してください。

- ALTER TABLE は、テーブル・クラスの定義に [\[DdlAllowed\]](#) が含まれている場合を除き、[永続クラスから投影されたテーブル](#) では使用できません。使用すると、操作は SQLCODE -300 エラーで失敗し、%msg が “DDL schema.tablename ” に設定されます。
- ALTER TABLE は、[導入済みの永続クラス](#) から投影されたテーブルでは使用できません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ ” classname ” DDL ” に設定されます。

ALTER TABLE は、table に対してテーブルレベルのロックを取得します。これにより、他のプロセスはこのテーブルのデータを変更できなくなります。このロックは ALTER TABLE 操作が終了すると自動的に解除されます。ALTER TABLE は対応するクラス定義をロックする際に、現在のプロセスに対して [SQL ロック・タイムアウト設定](#) を使用します。

テーブルを変更する場合、そのテーブルを別のプロセスによって EXCLUSIVE MODE または SHARE MODE でロックしないでください。ロックされているテーブルを変更しようとする、SQLCODE -110 エラーになり、%msg は “ 'Sample.MyTest' ” になります。

## RENAME Table

以下の構文を使用して、既存のテーブルの名前を変更できます。

```
ALTER TABLE schema.TableName RENAME NewTableName
```

この操作では、既存のスキーマ内にある既存のテーブルの名前を変更します。変更できるのはテーブル名だけです。テーブル・スキーマは変更できません。NewTableName でスキーマ名を指定すると、SQLCODE -1 エラーになります。古いテーブルと新しいテーブルの両方に同じテーブル名を指定すると、SQLCODE -201 エラーが生成されます。

テーブルの名前を変更すると、SQL テーブル名が変更されます。対応する永続クラス名は変更されません。

テーブルの名前を変更しても、トリガ内にある古いテーブル名への参照は変更されません。

ビューで既存のテーブル名を参照している場合、そのテーブルの名前を変更しようとする失敗します。その理由は、テーブルの名前の変更を試行するのはビューのリコンパイルを引き起こすアトミックな操作であるため、これによって SQLCODE -30 エラー “ schema.oldname ” が生成されます。

## ADD COLUMN の制限

ADD COLUMN では、単一の列、またはコンマ区切りの列のリストを追加できます。

ALTER TABLEtablename ADD COLUMN 文を使用してテーブルにフィールドを追加することを考えます。

- その名前の列が既に存在している場合、この文は失敗し SQLCODE -306 エラーが発生します。
- この文で、列に NOT NULL 制約を設定し、さらに既定値を設定しない場合、そのテーブルに既にデータがあると、この文はエラーになります。これは、DDL 文が完了した後では NOT NULL 制約は既存の行すべてを満たさないためです。その結果、[エラー・コード](#) SQLCODE -304 が生成されますが、これはデータが存在するテーブルに、既定値のない NOT NULL フィールドを追加しようとするためです。
- この文で、列に NOT NULL 制約を設定し、さらに既定値を設定した場合、テーブルにある既存の行は更新され、追加したフィールドの列には設定した既定値が割り当てられます。これには、CURRENT\_TIMESTAMP などの既定値が含まれます。
- この文で、列に NOT NULL 制約を設定せず、既定値を設定した場合は、既存のどの行の列でもデータは更新されません。これらの行に対する列の値は NULL です。

この既定の NOT NULL 制約の動作を変更するには、“[SET OPTION](#)” コマンドの “COMPILEMODE=NOCHECK” を参照してください。

“ID” という名前の通常のデータ・フィールドを指定し、[RowID](#) フィールドが既に “ID” (既定) という名前である場合、ADD COLUMN 操作は成功します。ALTER TABLE は ID データ列を追加し、RowId 列を “ID1” に名前変更して名前の重複を回避します。

## 整数カウンタの追加

ALTER TABLE tablename ADD COLUMN 文を使用してテーブルに整数カウンタ・フィールドを追加しようすると、以下のようになります。

- テーブルに [IDENTITY](#) フィールドが存在しない場合、テーブルに IDENTITY フィールドを追加できます。テーブルに IDENTITY フィールドが既に存在する場合、ALTER TABLE 操作は SQLCODE -400 エラーで失敗し、%msg は “ #5281: identity : 'Sample.MyTest::MyIdent2' ” になります。ADD COLUMN を使用してこのフィールドを定義する場合、対応する [RowID](#) 整数値を使用して InterSystems IRIS はこのフィールドで既存データ行を入力します。

CREATE TABLE で[ビットマップ・エクステント・インデックス](#)を定義して、後でテーブルに IDENTITY フィールドを追加した場合、IDENTITY フィールドが MINVAL が 1 以上の %BigInt、%Integer、%SmallInt、または %TinyInt のデータ型ではなく、テーブルにデータがないときは、システムは自動的にビットマップ・エクステント・インデックスを削除します。

- テーブルに 1 つ以上の [Serial \(%Library.Counter\)](#) フィールドを追加できます。ADD COLUMN を使用してこのフィールドを定義する場合、このフィールドの既存のデータ行は NULL です。UPDATE を使用して、このフィールドの NULL である既存のデータ行に値を指定することができます。UPDATE を使用して非 NULL 値を変更することはできません。



- ・ テーブルに [ROWVERSION](#) フィールドが存在しない場合、ROWVERSION フィールドを追加できます。テーブルに ROWVERSION フィールドが既に存在する場合、ALTER TABLE 操作は SQLCODE -400 エラーで失敗し、%msg は " #5320: 'Sample.MyTest' %Library.RowVersion 1 : MyVer,MyVer2" になります。ADD COLUMN を使用してこのフィールドを定義する場合、このフィールドの既存のデータ行は NULL です。NULL である ROWVERSION 値を更新することはできません。

## ALTER COLUMN の制限

ALTER COLUMN では、単一の列の定義を変更できます。

- ・ 構文 ALTER TABLE tablename ALTER COLUMN oldname RENAME newname を使用して列の名前を変更します。列の名前を変更すると、SQL フィールド名が変更されます。対応する永続クラスのプロパティ名は変更されません。ALTER COLUMN oldname RENAME newname は、トリガー・コードと ComputeCode の oldfield 名の参照を置き換えます。
- ・ 列の特性（データ型、既定値、NULL/NOT NULL、および照合タイプ）を変更します。

テーブルにデータが含まれている場合、データが含まれている列の[データ型](#)を変更することでストリーム・データが非ストリーム・データになる場合、または非ストリーム・データがストリーム・データになる場合、そのデータ型変更は実行できません。これを実行しようとする、SQLCODE -374 エラーが返されます。既存のデータがない場合は、このデータ型の変更が可能です。

ALTER COLUMN を使用して、[フィールドの既定値](#)を追加、変更、または削除できます。

テーブルにデータが含まれていて、列に NULL 値が含まれている場合は、その列に NOT NULL を指定することはできません。これは SQLCODE -305 エラーになります。

データを含む列の[照合タイプ](#)を変更する場合は、その列の[すべてのインデックスを再構築](#)する必要があります。

## MODIFY column の制限

MODIFY では、単一の列またはコンマ区切りの列のリストの定義を変更できます。

- ・ 構文 ALTER TABLE tablename MODIFY oldname RENAME newname を使用して列の名前を変更します。列の名前を変更すると、SQL フィールド名が変更されます。対応する永続クラスのプロパティ名は変更されません。MODIFY oldname RENAME newname は、トリガー・コードと ComputeCode の oldfield 名の参照を置き換えます。
- ・ 列の特性（データ型、既定値、および他の特性）を変更します。

テーブルにデータが含まれている場合、以下のように、データが含まれている列の[データ型](#)を互換性のないデータ型に変更することはできません。

- ・ 既存のデータ値と競合する場合、より低い（より包括的ではない）[データ型の優先順位](#)を持つデータ型。これを実行しようとする、SQLCODE -104 エラーになり、どのフィールドとどのデータ値がエラーの原因となっているかを示す %msg が表示されます。
- ・ 既存のデータ値と競合する場合、より小さい MAXLEN、または MAXVAL/MINVAL を持つデータ型。これを実行しようとする、SQLCODE -104 エラーになり、どのフィールドとどのデータ値がエラーの原因となっているかを示す %msg が表示されます。
- ・ ストリーム・データ型から非ストリーム・データ型、または非ストリーム・データ型からストリーム・データ型へのデータ型の変更。これを実行しようとする、SQLCODE -374 エラーが返されます。既存のデータがない場合は、このデータ型の変更が可能です。

MODIFY を使用して、[フィールドの既定値](#)を追加または変更できます。MODIFY を使用して、フィールドの既定値を削除することはできません。

テーブルにデータが含まれていて、列に NULL 値が含まれている場合は、その列に NOT NULL を指定することはできません。これは SQLCODE -305 エラーになります。構文形式 ALTER TABLE mytable MODIFY field1 NOT NULL



および ALTER TABLE mytable MODIFY field1 CONSTRAINT nevernull NOT NULL は同じ操作を実行します。オプションの CONSTRAINT identifier 節は、互換性を保つための空命令です。InterSystems IRIS ではこのフィールド制約名を保持または使用しません。このフィールド制約名を指定してこのフィールド制約を削除しようとすると、SQLCODE -315 エラーになります。

## DROP COLUMN の制限

DROP COLUMN は、コンマ区切りリストとして指定された複数の列定義を削除します。リストされる各列名の後には、その RESTRICT または CASCADE (指定しない場合、既定は RESTRICT)、および %DELDDATA または %NODELDDATE (指定しない場合、既定は %NODELDDATA) オプションが続く必要があります。

既定では、列定義を削除しても、その列に格納されているデータはデータマップから削除されません。列定義とそのデータの両方を削除するには、%DELDDATA オプションを指定します。

列の定義を削除しても、対応する列レベルの特権は削除されません。例えば、その列でデータを挿入、更新、または削除するためにユーザに与えられた特権です。これは、以下のような影響があります。

- ・ 列が削除され、同じ名前を持つ別の列が追加されると、ユーザとロールは新しい列でも古い列で持っていたのと同じ特権を持ちます。
- ・ 列が削除されても、その列のオブジェクト特権を削除することはできません。

この理由により、通常は列定義を削除する前に、[REVOKE](#) コマンドを使用して、列レベルの特権を削除することをお勧めします。

**RESTRICT キーワード (またはキーワードなし) :** 列がインデックスで使用されている場合や、外部キー制約または他の一意制約で定義されている場合、その列を削除することはできません。その列を削除しようとすると、SQLCODE -322 エラーが発生して失敗します。既定は RESTRICT です。“[DROP INDEX](#)” を参照してください。

**CASCADE キーワード :** 列がインデックスで使用されている場合は、インデックスが削除されます。複数のインデックスが存在する場合があります。列が外部キーで使用されている場合は、外部キーが削除されます。複数の外部キーが存在する場合があります。

列が COMPUTECODE 節内または COMPUTEONCHANGE 節内で使用されている場合は、その列を削除することはできません。これを実行しようとすると、SQLCODE -400 エラーが返されます。

## ADD CONSTRAINT の制限

制約をフィールドのコンマ区切りリストに追加できます。例えば、UNIQUE (FName,SurName) 制約を追加できます。この制約は、2 つのフィールド FName と Surname の値を組み合わせる UNIQUE 制約を設定します。同様に、フィールドのコンマ区切りリストに対して [主キー制約](#) や [外部キー制約](#) を追加することもできます。

制約は名前付き制約にも名前のない制約にもできます。名前のない制約の場合、制約名はテーブル名を使用して生成されます。例えば、MYTABLE\_Unique1 や MYTABLE\_PKEY1 のようになります。

以下の例では、名前のない制約を 2 つ作成して、一意制約と主キー制約の両方をフィールドのコンマ区切りリストに追加します。

### SQL

```
ALTER TABLE SQLUser.MyStudents
  ADD UNIQUE (FName,SurName), PRIMARY KEY (FName,Surname)
```

- ・ 制約で使用するフィールドが存在している必要があります。存在しないフィールドを指定すると、SQLCODE -31 エラーが生成されます。
- ・ 制約で RowId フィールドを使用することはできません。RowId (ID) フィールドを指定すると、SQLCODE -31 エラーが生成されます。
- ・ 制約でストリーム・フィールドを使用することはできません。ストリーム・フィールドを指定すると、SQLCODE -400 “エラーインデックス属性に誤り” が生成されます。

- ・ 制約は 1 つのフィールドに一度だけ適用できます。1 つのフィールドに同じ制約を 2 回指定すると、SQLCODE -400 エラー “インデックス名が重複しています” が生成されます。

オプションの CONSTRAINT identifier キーワード節を使用して、名前付き制約を作成できます。名前付き制約は、有効な識別子である必要があります。制約名では、大文字と小文字は区別されません。これにより、将来的に使用できるように、制約の名前が指定されます。以下に例を示します。

## SQL

```
ALTER TABLE SQLUser.MyStudents
ADD CONSTRAINT UnqFullName UNIQUE (FName,SurName)
```

複数の制約をコンマ区切りリストとして指定できます。制約名は最初の制約に適用され、その他の制約には既定の名前が適用されます。

制約名は、テーブルで一意である必要があります。1 つのフィールドに同じ制約名を 2 回指定すると、SQLCODE -400 エラー “インデックス名が重複しています” が生成されます。

## ADD PRIMARY KEY の制限

主キーの値は必須かつ一意です。したがって、主キー制約を既存のフィールドにまたはフィールドの組み合わせに追加すると、これらの各フィールドは必須フィールドになります。既存のフィールドのリストに主キー制約を追加する場合、これらのフィールドの値の組み合わせが一意である必要があります。NULL 値の入力が認められているフィールドには、主キー制約を追加できません。フィールド (またはフィールドのリスト) に一意でない値が含まれる場合、そのフィールド (またはフィールドのリスト) に主キー制約を追加することはできません。

既存のフィールドに主キー制約を追加する場合、フィールドが自動的に IDKey インデックスとして定義されることもあります。これはデータが存在するかどうか、および構成設定が以下のいずれかの方法で設定されているかどうかによります。

- ・ SQL SET OPTION PKEY\_IS\_IDKEY 文
- ・ システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DDLPrimaryKeyNotIDKey。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、「DDL ID」と表示されます。既定値は 1 です。
- ・ 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[DDL を使用して作成されたテーブルの ID キーとして主キーを定義する] の現在の設定を表示します。
  - このチェック・ボックスにチェックが付いていない場合 (既定)、主キーはクラス定義の IDKey インデックスになりません。IDKEY ではない主キーを使用したレコードへのアクセスはかなり非効率的です。しかし、このタイプの主キー値は変更することができます。
  - このチェック・ボックスにチェックが付いている場合、主キー制約が DDL で指定され、フィールドにデータがないときには、主キー・インデックスが IDKey インデックスとしても定義されます。フィールドにデータがある場合、IDKey インデックスは定義されません。主キーを IDKey インデックスとして定義すると、データ・アクセスはより効率的ですが、一度設定された主キー値は変更できません。

CREATE TABLE でビットマップ・エクステン・インデックスを定義している場合に、後で ALTER TABLE を使用して IDKey でもある主キーを追加すると、システムは自動的にビットマップ・エクステン・インデックスを削除します。

## 主キーが既に存在する場合に主キーを作成する

主キーは 1 つしか定義できません。既定では、InterSystems IRIS は、主キーが既に存在する場合に主キーを定義しようとしたり、同じ主キーを 2 回定義しようとしたりすると、それを拒否して SQLCODE -307 エラーを発行します。主キーの第 2 定義が最初の定義と同じ場合も SQLCODE -307 エラーを発行します。現在の構成を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[DDL Create Primary Key] 設定が表示されます。既定値は 0 (いいえ) で、これが推奨される構成設定です。このオプションが 1 (はい) に設定されてい

ると、ALTER TABLE ADD PRIMARY KEY により、InterSystems IRIS は主キー・インデックスをクラス定義から削除し、指定の主キー・フィールドを使用したインデックスを再生成します。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション（および他の同様の作成、変更、および削除のオプション）をシステム全体で設定できます。

ただし、既存の主キーが存在する状態で主キーを作成できるようにこのオプションを設定していても、テーブルにデータがある場合は、IDKey インデックスを兼ねる主キー・インデックスは再作成できません。これを実行しようとすると、SQLCODE -307 エラーが生成されます。

## ADD FOREIGN KEY の制限

外部キーの詳細は、CREATE TABLE コマンドの“外部キーの定義”と外部キーの“参照アクション節”、および“外部キーの使用法”を参照してください。

既定では、同じ名前の 2 つの外部キーを持つことはできません。これを実行しようとすると、SQLCODE -311 エラーが生成されます。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[

DDL ADD 1 設定が表示されます。既定値は 0（いいえ）で、これが推奨設定です。1（はい）の場合、同じ名前が存在しても、DDL を使用して外部キーを追加できます。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション（および他の同様の作成、変更、および削除のオプション）をシステム全体で設定できます。

テーブル定義では、同じ field-commalist フィールドを参照して相反する参照アクションを実行する、名前が異なる 2 つの外部キーを指定することはできません。同一のフィールドに対して相反する参照アクションを実行する 2 つの外部キーを定義した場合（例：ON DELETE CASCADE と ON DELETE SET NULL）、InterSystems SQL は ANSI 標準に従い、エラーを生成しません。この代わりに InterSystems SQL では、DELETE 処理または UPDATE 処理でこのような相反する外部キー定義に遭遇したときにエラーを生成します。

存在しない外部キー・フィールドを ADD FOREIGN KEY で指定すると、SQLCODE -31 エラーが発生します。

存在しない親キー・テーブルを ADD FOREIGN KEY で参照すると、SQLCODE -310 エラーが発生します。既存の親キー・テーブルに存在しないフィールドを ADD FOREIGN KEY で参照すると、SQLCODE -316 エラーが発生します。親キー・フィールドを指定していない場合、既定の ID フィールドになります。

ADD FOREIGN KEY の発行前、ユーザは、参照されるテーブルまたは参照されるテーブルの列の [REFERENCES 特権](#)を持っている必要があります。REFERENCES 特権は、ダイナミック SQL を介して、または SQL ドライバ接続により ALTER TABLE を実行する場合に必要になります。

一意でない値を設定できるフィールド（またはフィールドの組み合わせ）を ADD FOREIGN KEY で参照すると、SQLCODE -314 エラーが発生して、%msg に追加の詳細が表示されます。

NO ACTION は、[シャード・テーブル](#)でサポートされている唯一の参照アクションです。

テーブルにデータがある場合は、ADD FOREIGN KEY は制約されます。この既定の制約の動作を変更するには、“[SET OPTION](#)” コマンドの“COMPILEMODE=NOCHECK オプション”を参照してください。

単一のフィールドのために ADD FOREIGN KEY 制約を定義し、外部キーが参照先テーブルの idkey を参照する場合は、InterSystems IRIS によって外部キー内のプロパティが参照プロパティに変換されます。この変換は、以下の制限に従います。

- ・ テーブルにデータを含めることはできません。
- ・ 外部キーに関するプロパティを永続クラスのプロパティにできません（つまり、既に参照プロパティにできません）。
- ・ 外部キー・フィールドおよび参照先 idkey フィールドのデータ型とデータ型パラメータは同じである必要があります。
- ・ 外部キー・フィールドを [IDENTITY フィールド](#)にすることはできません。

## DROP CONSTRAINT の制限

既定では、外部キー制約によって参照されている一意キー制約または主キー制約は削除できません。これを実行しようとすると、SQLCODE -317 エラーが返されます。この既定の外部キー制約の動作を変更するには、“SET OPTION” コマンドの “COMPILEMODE=NOCHECK オプション” を参照してください。

主キー制約の削除による影響は、上記のように [主キーも ID キーである] 設定の内容によって異なります。

- PrimaryKey インデックスが IDKey インデックスを兼ねていない場合、主キー制約を削除すると PrimaryKey インデックスの定義が削除されます。
- PrimaryKey インデックスが IDKey インデックスを兼ねていて、テーブルにデータがない場合、主キー制約を削除するとインデックスの定義全体が削除されます。
- PrimaryKey インデックスが IDKey インデックスを兼ねていて、テーブルにデータがある場合、主キー制約を削除すると、IDKey インデックスの定義から PRIMARYKEY 修飾子のみが削除されます。

## 存在しない制約の削除

既定では、制約を持たないフィールドのフィールド制約を削除しようとした場合、InterSystems IRIS は削除を拒否し、SQLCODE -315 エラーを発行します。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[ DDL DROP ] 設定が表示されます。既定値は 0 (いいえ) で、これが推奨設定です。このオプションが 1 (はい) に設定されていると、ALTER TABLE DROP CONSTRAINT により、InterSystems IRIS は処理を実行せず、エラー・メッセージを発行しません。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション（および他の同様の作成、変更、および削除のオプション）をシステム全体で設定できます。

## 例

以下の例では埋め込み SQL プログラムを使用して、テーブルを作成し、2 行を生成してからテーブルの定義を変更します。

この動作をはっきり示すために、最初の 2 つの埋め込み SQL プログラムは示されている順序で実行してください（埋め込み SQL では参照されるテーブルが既に存在していなければ INSERT 文をコンパイルすることができないため、ここでは 2 つの埋め込み SQL プログラムを使用する必要があります）。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(DROP TABLE SQLUser.MyStudents)
IF SQLCODE=0 { WRITE !,"Deleted table" }
ELSE { WRITE "DROP TABLE error SQLCODE=",SQLCODE }
&sql(CREATE TABLE SQLUser.MyStudents (
    FirstName VARCHAR(35) NOT NULL,
    LastName VARCHAR(35) NOT NULL
))
IF SQLCODE=0 { WRITE !,"Created table" }
ELSE { WRITE "CREATE TABLE error SQLCODE=",SQLCODE }
```

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
NEW SQLCODE,%msg
&sql(INSERT INTO SQLUser.MyStudents (FirstName, LastName)
VALUES ('David','Vanderbilt'))
IF SQLCODE=0 { WRITE !,"Inserted data in table" }
ELSE { WRITE !,"SQLCODE=",SQLCODE," : ",%msg }
&sql(INSERT INTO SQLUser.MyStudents (FirstName, LastName)
VALUES ('Mary','Smith'))
IF SQLCODE=0 { WRITE !,"Inserted data in table" }
ELSE { WRITE !,"SQLCODE=",SQLCODE," : ",%msg }
```

以下の例は、ALTER TABLE を使用して ColorPreference 列を追加します。列定義で既定値を指定するため、システムではテーブルの既存の 2 つの行に対して ColorPreference に値 'Blue' を入力します。

#### ObjectScript

```
NEW SQLCODE,%msg
&sql(ALTER TABLE SQLUser.MyStudents
  ADD COLUMN ColorPreference VARCHAR(16) NOT NULL DEFAULT 'Blue')
IF SQLCODE=0 {
  WRITE !,"Added a column",! }
ELSEIF SQLCODE=-306 {
  WRITE !,"SQLCODE=",SQLCODE," : ",%msg }
ELSE { WRITE "SQLCODE error=",SQLCODE }
```

以下の例は、ALTER TABLE を使用して、2 つの計算列 FLName および LFName を追加します。既存の行のこれらの列に値はありません。その後に挿入される行については、これらの列それぞれに対して値が計算されます。

#### ObjectScript

```
NEW SQLCODE,%msg
&sql(ALTER TABLE SQLUser.MyStudents
  ADD COLUMN FLName VARCHAR(71) COMPUTECODE { SET {FLName}={FirstName}_ " _{LastName}}
    COMPUTEONCHANGE (FirstName,LastName),
  COLUMN LFName VARCHAR(71) COMPUTECODE { SET {LFName}={LastName}_ " ," _{FirstName}}
    COMPUTEONCHANGE (FirstName,LastName) )
IF SQLCODE=0 {
  WRITE !,"Added two computed columns",! }
ELSE { WRITE "SQLCODE error=",SQLCODE }
```

## 関連項目

- ・ [CREATE TABLE、DROP TABLE](#)
- ・ [JOIN](#)
- ・ [SELECT](#)
- ・ [INSERT、UPDATE、INSERT OR UPDATE、DELETE](#)
- ・ [テーブルの定義](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)



## ALTER USER (SQL)

ユーザのパスワードを変更します。

### 構文

```
ALTER USER user-name IDENTIFY BY password
ALTER USER user-name IDENTIFIED BY password
ALTER USER user-name [ WITH ] PASSWORD password
```

### 説明

ALTER USER コマンドを使用すると、ユーザのパスワードを変更できます。自分のパスワードはいつでも変更できます。別のユーザのパスワードを変更するには、以下のいずれかを持つユーザとしてログインする必要があります。

- ・ USE 権限がある [%Admin\\_Secure](#) 管理リソース
- ・ USE 権限がある [%Admin\\_UserEdit](#) 管理リソース
- ・ システムに対する全面的なセキュリティ権限

IDENTIFY BY、IDENTIFIED By、および WITH PASSWORD キーワードは同義語です。

user-name は既存のユーザでなければなりません。存在しないユーザを指定すると、SQLCODE -400 エラーが生成され、%msg は " #838: badname " のようになります。\$SYSTEM.SQL.Security.UserExists() メソッドを呼び出すことによって、ユーザが存在するかどうかを判別できます。

[区切り識別子](#)として指定された user-name には、SQL 予約語を使用することも、コンマ (,), ピリオド (.), キャレット (^), および 2 文字の矢印シーケンス (->) を含めることもできます。この名前の先頭の文字には、アスタリスク (\*) を除く任意の有効な文字を使用できます。

password には文字列リテラル、数値、または識別子を指定できます。文字列リテラルは引用符で囲む必要があり、空白スペースを含む文字の任意の組み合わせを使用できます。数値または識別子は引用符で囲む必要はありません。数値は 0 ～ 9 の文字のみで構成する必要があります。[識別子](#)は、文字 (大文字/小文字) あるいは % (パーセント記号) で始める必要があります。その後ろに、任意の文字、任意の数字、\_ (アンダースコア)、& (アンパサンド)、\$ (ドル記号)、@ (アットマーク) を自由に組み合わせて使用できます。

ALTER USER は、新しいパスワードが既存のものと同じでもエラー・コードを発行しません。この場合、SQLCODE = 0 (正常終了) として設定されます。

\$SYSTEM.Security.ChangePassword() メソッドを使用して、ユーザ・パスワードを変更することもできます。

```
$SYSTEM.Security.ChangePassword(args)
```

### 特権

ALTER USER コマンドは特権を必要とする操作です。埋め込み SQL で ALTER USER を使用するには、USE 権限がある [%Admin\\_Secure](#) 管理リソース、USE 権限がある [%Admin\\_UserEdit](#) 管理リソース、またはシステムに対する全面的なセキュリティ特権を持つユーザとしてログインしておく必要があります。特権がない場合は、SQLCODE -99 エラー (特権違反) が返されます。\$SYSTEM.Security.Login() メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

#### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、"**%SYSTEM.Security**" を参照してください。

## 引数

### user-name

変更するパスワードを持っている既存のユーザの名前。ユーザ名は、大文字と小文字が区別されません。

### password

ユーザの新しいパスワード。パスワードは少なくとも 3 文字必要で、32 文字を超えることはできません。パスワードは大文字と小文字が区別されます。パスワードには Unicode 文字を使用することができます。

## 例

以下の埋め込み SQL の例は、Bill というユーザのパスワードを “temp\_pw” から “pw4AUser” に変更します。

### ObjectScript

```

Main
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(CREATE USER Bill IDENTIFY BY temp_pw)
IF SQLCODE=0 { WRITE !,"Created user" }
ELSE { WRITE "CREATE USER error SQLCODE=",SQLCODE,! }
&sql(ALTER USER BILL IDENTIFY BY pw4AUser)
IF SQLCODE=0 { WRITE !,"Altered user password" }
ELSE { WRITE "ALTER USER error SQLCODE=",SQLCODE,! }
Cleanup
SET toggle=$RANDOM(2)
IF toggle=0 {
&sql(DROP USER Bill)
IF SQLCODE=0 { WRITE !,"Dropped user" }
ELSE { WRITE "DROP USER error SQLCODE=",SQLCODE }
}
ELSE {
WRITE !,"No drop this time"
QUIT
}

```

## 関連項目

- SQL 文 : [CREATE USER](#)、[DROP USER](#)、[GRANT](#)、[REVOKE](#)
- SQL のユーザ、ロール、および特権
- ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数
- [SQLCODE エラー・メッセージ](#)

## ALTER VIEW (SQL)

ビューを変更します。

### 構文

```
ALTER VIEW viewName AS query
ALTER VIEW viewName (column, column2, ...) AS query

ALTER VIEW viewName ... AS query WITH READ ONLY
ALTER VIEW viewName ... AS query WITH
  [LOCAL | CASCADED] CHECK OPTION
```

### 概要

ALTER VIEW コマンドは、[CREATE VIEW](#) コマンドを使用して作成されたビューまたは永続クラスから投影されたビューを変更します。変更したビューで既存のビューが置き換えられるので、ビューにある特定の列を変更することはできません。

view は、SELECT クエリや、そのようなクエリの UNION の結果セットに基づく仮想テーブルです。ビューの詳細は、["ビューの定義と使用"](#) を参照してください。

- ALTER VIEW [viewName](#) AS [query](#) は、ビューの既存の列を、SELECT クエリで返される列に置き換えます。ビュー列の名前は、クエリの結果セットに返された列名から派生するので次のようになります。
  - クエリの対象とするテーブルまたはビューの列名またはそのエイリアス
  - [テーブル値関数](#)として定義したクラス・クエリの列名

以下の文は、過去 12 か月以内に雇用された従業員のみが含まれるように NewEmployees ビューを変更します。ビューの列名 Name、Office、StartDate はソース・テーブルの列名と一致します。

#### SQL

```
ALTER VIEW NewEmployees AS
  SELECT Name,Office,StartDate
  FROM Sample.Employees
  WHERE DATEDIFF('month',StartDate,CURRENT_DATE) <= 12
```

- ALTER VIEW viewName ([column](#), column2, ...) AS query は、ビューに含める列の名前を指定します。指定する列名は SELECT クエリで返されるテーブル列の数および順序と一致させる必要があります。これらのビュー列名は SELECT 文クエリで列名のエイリアスとして指定することもできます。以下の ALTER VIEW 文は同等です。

#### SQL

```
ALTER VIEW MyView (MyViewCol1,MyViewCol2,MyViewCol3) AS
  SELECT TableCol1, TableCol2, TableCol3 FROM MyTable
```

#### SQL

```
ALTER VIEW MyView AS SELECT TableCol1 AS ViewCol1,
  TableCol2 AS ViewCol2,
  TableCol3 AS ViewCol3
  FROM MyTable
```

列を指定すると、ビューで指定されている既存の列はすべて置き換えられます。

例：[ビューの作成と変更](#)

- ALTER VIEW viewName ... AS query WITH READ ONLY は、このビューの基になっているテーブルに対して、このビューからは挿入、更新、削除の各操作を実行できないように指定します。既定では、指定された任意の WITH CHECK OPTION 制約を条件として、ビュー経由でこれらの操作が許可されます。



### 例：読み取り専用ビューの設定

- ALTER VIEW viewName ... AS query WITH [LOCAL | CASCADED] CHECK OPTION は、更新される行またはこのビューに挿入される行がビューの **WHERE** 制約を満たしていることを確認します。行がこれらの制約を満たしていない場合、行は更新または挿入されません。以下のチェック・オプションを指定できます。
  - WITH LOCAL CHECK OPTION – INSERT または UPDATE 文で指定されたビューの WHERE 節のみをチェックします。
  - WITH CASCADED CHECK OPTION または WITH CHECK OPTION – INSERT または UPDATE 文で指定されたビューの WHERE 節とそのビューの基になっているすべての基本ビューをチェックします。このオプションは、これらの基本ビューのすべての WITH LOCAL CHECK OPTION 節を上書きします。更新可能なすべてのビューにこのオプションを推奨します。

これらのオプションの詳細は、“[WITH CHECK オプション](#)” を参照してください。

### 例：ビューを介して行われたテーブルの変更の検証

## 引数

### viewName

変更するビュー名。[テーブル名](#)と同じ命名規則で変更します。ビュー名は修飾 (schema.viewname)、未修飾 (viewname) のどちらでもかまいません。ビュー名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。

指定のビューが現在のネームスペースに存在するかどうかを確認するには、`$SYSTEM.SQL.Schema.ViewExists()` メソッドを使用します。

永続クラスからビューを投影する場合は、そのビューに `Classtype="view"` キーワードと `DDLAllowed` キーワードを指定している場合にのみ、ALTER VIEW を実行できます。クラス・クエリから投影するビューは変更できません。

### query

ビューの基準となるクエリの結果セット。クエリは、**SELECT** 文、または 2 つ以上の **SELECT** 文の **UNION** として指定できます。UNION コマンドの使用例については、“[結合された SELECT クエリを使用したビューの変更](#)” を参照してください。

ビュー・クエリには、[ホスト変数](#)や **INTO** キーワードを格納できません。query でホスト変数を参照しようとする、システムにより、SQLCODE -148 エラーが生成されます。

### column

変更されたビューに含まれる列の名前。列名が複数の場合はコンマ区切りリストで指定します。列名は、[viewName](#) 引数の後または [query](#) 引数で指定できます。

## 例

### ビューの作成と変更

この例では、ビューを作成し、変更する方法を示します。ビューに対してクエリを実行する方法、およびビューを削除する方法も示します。

米国マサチューセッツ州に住む人の名前を含むビューを作成します。この例では、`Sample.Person` テーブルが既に存在し、`Home_State` 列が含まれていることを前提としています。

### SQL

```
CREATE VIEW MassFolks (vFullName) AS
  SELECT Name FROM Sample.Person WHERE Home_State='MA'
```

次に、通常のテーブルと同じように、ビューに対してクエリを実行できます。

## SQL

```
SELECT * FROM MassFolks
```

ビューを変更して新しい列を含めます。ビューを変更すると、列リストが新しい列リストに置き換わりませんが、既存の列リストは保存されません。このため、この変更されたビューには `vMassAbbrev` 列と `vCity` 列のみが含まれ、`vFullName` 列は含まれません。

## SQL

```
ALTER VIEW MassFolks (vMassAbbrev,vCity) AS
    SELECT Home_State,Home_City FROM Sample.Person WHERE Home_State='MA'
```

ビューを削除します。通常のテーブルと同じような方法でビューを削除できます。

## SQL

```
DROP VIEW MassFolks
```

## 結合された SELECT クエリを使用したビューの変更

ビューを変更して、2 つの SELECT クエリの結果を結合したものを含めます。結果を結合するには、UNION コマンドを使用します。

## SQL

```
ALTER VIEW MyView (vname,vstate) AS
    SELECT t1.name,t1.home_state
        FROM Sample.Person AS t1
    UNION
    SELECT t2.name,t2.office_state
        FROM Sample.Employee AS t2
```

## 読み取り専用ビューの設定

ビューを変更し、このビューを介して基本テーブルが変更されないようにします。

## SQL

```
ALTER VIEW YoungPeople AS
    SELECT Name,DOB
        FROM Sample.Person
    WHERE DATEDIFF(year,DOB,CURRENT_DATE) <= 18
WITH READ ONLY
```

このビューを介して行を更新すると、WITH READ ONLY によって更新が阻止されます。

## SQL

```
UPDATE YoungPeople (DOB)
VALUES (02/17/2022)
WHERE Name='Page,Laura O.'
```

## ビューを介して行われたテーブルの変更の検証

GPA 基準を満たさない学生が挿入されないように、この特待生のビューを変更します。この例では、`Sample.Student` テーブルが既に存在することを前提としています。

## SQL

```
ALTER VIEW HonorsStudent AS
  SELECT Name, GPA
  FROM Sample.Student
  WHERE GPA > 3.0
WITH CHECK OPTION
```

GPA が非常に低い学生をこのビューに挿入しようとする、VIEW CHECK OPTION によって挿入が阻止されます。

## SQL

```
INSERT INTO HonorsStudent (Name, GPA)
VALUES ('Waal, Edgar P.', 2.9)
```

## セキュリティおよび特権

ALTER VIEW コマンドは特権を必要とする操作です。ALTER VIEW を実行するには、ユーザは `%ALTER_VIEW` [管理特権](#) を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ ” 'name' ” に設定されます。

ユーザは、指定されたビューに対する `%ALTER` 特権を持っている必要があります。ユーザがビューの所有者（作成者）である場合、ユーザにはそのビューに対する `%ALTER` 特権が自動的に付与されます。そうでない場合は、ユーザにビューに対する `%ALTER` 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ ” 'name' ” 'Schema.ViewName' ” に設定されます。

適切な付与特権を保持している場合は、[GRANT](#) コマンドを使用して `%ALTER_VIEW` 特権と `%ALTER` 特権を割り当てることができます。

現在のユーザが `%ALTER` 特権を持っているかどうかを確認するには、[%CHECKPRIV](#) コマンドを呼び出します。指定のユーザが `%ALTER` 特権を持っているかどうかを確認するには、`$SYSTEM.Security.CheckPrivilege()` メソッドを呼び出します。

埋め込み SQL では、以下のように `$SYSTEM.Security.Login()` メソッドを使用して適切な特権を持ったユーザとしてログインできます。

```
DO $SYSTEM.Security.Login("myUserName", "myPassword")
&sql(...)
```

`$SYSTEM.Security.Login` メソッドを呼び出すには、[%Service\\_Login:Use](#) 特権が必要です。詳細は、["%SYSTEM.Security"](#) を参照してください。

ALTER VIEW は、[導入済みの永続クラス](#)から投影されたテーブルに基づくビューでは使用できません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ ” classname DDL ” に設定されます。

## 関連項目

- ・ [CREATE VIEW](#)、[DROP VIEW](#)、[GRANT](#)
- ・ [ビューの定義](#)
- ・ [SQLCODE エラー・メッセージ](#)

## BUILD INDEX (SQL)

1 つ以上のインデックスにデータを入力します。

### 構文

```
BUILD INDEX [%NOLOCK] [%NOJOURN] FOR TABLE table-name
  [INDEX index-name [, index-name]]

BUILD INDEX [%NOLOCK] [%NOJOURN] FOR SCHEMA schema-name

BUILD INDEX [%NOLOCK] [%NOJOURN] FOR ALL
```

### 説明

BUILD INDEX には、定義されているすべてのインデックスを構築/再構築するための 3 つの構文形式が用意されています。

- ・ テーブル：BUILD INDEX FOR TABLE *table-name*。オプションの INDEX 節を使用すると、指定したインデックスのみを構築/再構築できます。
- ・ スキーマ内のすべてのテーブル：BUILD INDEX FOR SCHEMA *schema-name*
- ・ 現在のネームスペース内のすべてのテーブル：BUILD INDEX FOR ALL

インデックスを構築する理由には、次のようなものがあります。

- ・ CREATE INDEX を使用して、既にデータを含むテーブルに 1 つ以上のインデックスを追加した場合。
- ・ INSERT、UPDATE、または DELETE の各操作でインデックスに書き込むことによるパフォーマンス・オーバーヘッドを受け入れるのではなく、%NOINDEX オプションを使用して各操作をテーブルに対して実行した場合。

どちらの場合も、BUILD INDEX を使用してこれらのインデックスにデータを入力します。

BUILD INDEX は、変更されたテーブルの数を、影響を受けた行数として返します。

DEFER BUILD オプションと共に CREATE INDEX を使用してインデックスを作成した場合、インデックスは手動で構築する必要があります。BUILD INDEX はインデックスのデータを構築しますが、クエリでそのインデックスを選択可能、または使用可能にするわけではないことに注意してください。インデックスを選択可能にするには、SetMapSelectability() メソッドを使用します。マップが選択可能かどうかを管理ポータルで表示するには、[\[システムエクスプローラ\]→\[SQL\]→\[カタログの詳細\]](#) に移動し、[\[マップ\]/\[インデックス\]](#) ボタンを選択します。

ObjectScript を利用して定義したクラスは、スーパークラスから構築する必要のあるインデックスを継承することがあります。このような“継承された”インデックスを構築するには、そのインデックスを定義しているスーパークラスに対して BUILD INDEX を呼び出す必要があります。そのインデックスを使用するサブクラスに対して呼び出すではありません。

テーブルで [%Storage.SQL](#) を使用する場合、そのクラスで明示的に定義したインデックスは構築されません。

### 特権

BUILD INDEX コマンドは特権を必要とする操作です。BUILD INDEX を実行するには、ユーザは [%BUILD\\_INDEX](#) [管理特権](#)を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “[%BUILD\\_INDEX](#) [管理特権](#)” に設定されます。適切な付与特権を持っていれば、[GRANT](#) コマンドを使用して、ユーザまたはロールに [%BUILD\\_INDEX](#) 特権を割り当てることができます。管理特権はネームスペース固有のものです。詳細は、“[特権](#)” を参照してください。

ユーザは、指定されたテーブルに対する SELECT 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対する SELECT 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する SELECT 特権を付与する必要があります。

- ・ 指定したテーブルに対する SELECT 特権がない場合に BUILD INDEX FOR TABLE を発行すると、SQLCODE -30 エラーが発生し、%msg が " 'name' " に設定されます。
- ・ BUILD INDEX FOR SCHEMA を発行すると、ユーザが SELECT 特権を持つテーブルについてのみインデックスが構築されます。ユーザがスキーマ内のどのテーブルに対する SELECT 特権も持っていない場合、コマンドはエラーなしで完了し、影響を受ける列は 0 となります。

[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが SELECT 特権を持っているかどうかを確認できます。[GRANT](#) コマンドを使用して、指定したテーブルに SELECT 特権を割り当てることができます。詳細は、“[特権](#)”を参照してください。

## ロックとジャーナリング

既定では、BUILD INDEX 文はインデックスを構築する前に各テーブルのエクステント・ロックを取得します。これにより、他のプロセスはこのテーブルのデータを変更できなくなります。このロックは、BUILD INDEX 操作が終了すると自動的に解除されます。[%NOLOCK](#) を指定して、テーブルのロックを防ぐことができます。

既定では、BUILD INDEX 文は現在のプロセスのジャーナル設定を使用します。[%NOJOURN](#) を指定して、ジャーナリングを防ぐことができます。

[%NOLOCK](#) または [%NOJOURN](#) を使用するには、対応する SQL 管理特権が必要です。この特権は、[GRANT](#) コマンドを使用して設定できます。

## エラー・コード

- ・ 指定した table-name が存在しない場合、InterSystems IRIS は SQLCODE -30 エラーを発行し、%msg が " 'sample.tname' " に設定されます。このエラー・メッセージが返されるのは、テーブルではなくビューを指定した場合、または SELECT 特権を持たないテーブルを指定した場合です。
- ・ 指定した index-name が存在しない場合、InterSystems IRIS は SQLCODE -400 エラーを発行し、%msg が " #5066: 'sample.tname::badindex' " に設定されます。
- ・ 指定した schema-name が存在しない場合、InterSystems IRIS は SQLCODE -473 エラーを発行し、%msg が "Schema 'sample' not found" に設定されます。

## 引数

### FOR TABLE table-name

既存のテーブルの名前。table-name は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。

### INDEX index-name

インデックス名、またはコンマで区切った複数のインデックス名 (オプション)。指定した場合、指定したインデックスのみが構築されます。指定しない場合、テーブルに定義されているすべてのインデックスが構築されます。

### FOR SCHEMA schema-name

既存のスキーマの名前。このコマンドは、指定されたスキーマ内のすべてのテーブルのすべてのインデックスを構築します。

## 関連項目

- ・ [CREATE INDEX](#)
- ・ [インデックスの定義と構築](#)
- ・ [SQLCODE エラー・メッセージ](#)

## CALL (SQL)

ストアド・プロシージャを実行します。

### 構文

```
CALL procname(arg_list) [USING contextvar]  
  
retval=CALL procname(arg_list) [USING contextvar]
```

### 概要

CALL 文は、SQL ストアド・プロシージャとして公開されたクエリを呼び出します。procname は、現在のネームスペースにある既存のストアド・プロシージャであることが必要です。procname が見つからないと、InterSystems IRIS は SQLCODE -428 エラーを生成します。procname は、SqlProc=True のストアド・プロシージャであることが必要です。["SqlProc \(メソッド・キーワード\)"](#) を参照してください。

ストアド・プロシージャの詳細は、["CREATE PROCEDURE"](#) コマンドを参照してください。

### 引数

#### procname

既存のストアド・プロシージャの名前。引数を指定しない場合でも、procname の後には括弧を付加する必要があります。プロシージャ名は以下の形式のいずれかで指定することができます。

- ・ 未修飾：既定のスキーマ名が使用されます。例えば、MedianAgeProc() です。
- ・ 修飾：スキーマ名を指定します。例えば、Patient.MedianAgeProc() です。
- ・ 複数レベル：対応するクラス・パッケージ・メンバに一致させるために 1 つ以上のスキーマ・レベルで修飾します。この場合、procname に含めることができるピリオド文字は 1 つだけです。対応するクラス・メソッド名の他のピリオドは下線に置き換えられます。ピリオドは、最下位レベルのクラス・パッケージ・メンバの前に指定します。例えば、%SYSTEM.SQL\_GetROWID() や %SYS\_PTools.StatsSQL\_Export() のように指定します。

未修飾の procname と一致するプロシージャをスキーマ内で検索する際、[既定のスキーマ名](#)、または (指定されている場合は) [スキーマ検索パス](#) のスキーマ名が使用されます。スキーマ検索パスとシステム全体のスキーマ既定値のいずれを使用しても指定されたプロシージャが見つからない場合は、SQLCODE -428 エラーが生成されます。\$SYSTEM.SQL.Schema.Default() メソッドを使用して、現在のシステム全体の既定のスキーマ名を確認できます。初期のシステム全体の既定のスキーマ名は、クラス・パッケージ名 **User** に対応する SQLUser です。

procname が現在のネームスペースに存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.ProcedureExists() メソッドを使用します。procname では、大文字と小文字は区別されません。

引数をまったく指定しない場合でも、procname の後ろに引数の括弧を付加する必要があります。これを実行しないと、SQLCODE -1 エラーが返されます。

#### arg\_list

値をストアド・プロシージャに渡すために使用される引数のリストです。arg\_list は括弧で囲み、リストの引数はコンマで区切ります。引数を指定しない場合も括弧は必須です。

arg\_list 引数は省略可能です。このコンマ区切りリストは実引数リストと呼ばれ、このリストで指定される引数の数と順序は、プロシージャ定義内の仮引数リストのものと一致している必要があります。実引数値の指定が、ストアド・プロシージャで定義された仮引数より少なくてもかまいません。ストアド・プロシージャで定義された仮引数よりも多くの実引数値を指定した場合、システムは SQLCODE -370 エラーを生成します。このエラー・メッセージには、ストアド・プロシージャ名、指定した引数の数、およびそのストアド・プロシージャで定義している引数の数が示されます。



末尾の引数は省略できます。省略された末尾の引数は未定義となり、既定値が使用されます。引数リスト内で未定義の引数を指定するには、プレースホルダとしてコンマを指定します。例えば、(arg1,,arg3) と指定した場合は 3 つの引数が渡され、そのうち 2 番目の引数は未定義です。一般に、未定義の引数の値は、ストアド・プロシージャの定義時に指定された既定値となります。既定値が定義されていない場合、未定義の引数の値は NULL を取ります。詳細は、“[NULL および空文字列](#)” を参照してください。

ストアド・プロシージャで定義されたデータ型と一致しない引数値を指定した場合は、既定値が指定されている場合でも、その引数は NULL を取ります。例えば、ストアド・プロシージャで引数が IN numarg INT DEFAULT 99 と定義されているとします。CALL で数値引数が指定されている場合は、その arg 値が使用されます。CALL でその引数が省略されている場合は、定義された既定値が使用されます。ただし、CALL で非数値引数が指定されている場合は、定義された既定値ではなく NULL が使用されます。

arg\_list 引数にユーザ定義関数 (値を返すメソッド・ストアド・プロシージャ) を指定できます。

## USING contextvar

オプションの引数です。contextvar は、プロシージャの呼び出しで生成されるプロシージャ・コンテキスト・オブジェクトを受け取る記述子領域変数を指定します。省略した場合の既定値は %sqlcontext です。

## retval

プロシージャの戻り値を受け取るために指定する変数 (オプション)。格納できるのは単一の値です。結果セットは格納できません。ローカル変数、ホスト変数、または疑問符 (?) 引数として指定できます。

## 埋め込み SQL からの呼び出し

ObjectScript の埋め込み SQL では、CALL 文の発行、または DO コマンドによる基本ルーチンや基本メソッドの呼び出しが可能です。

埋め込み SQL では、CALL に渡す引数値をリテラルとして指定することも、:name 形式の[ホスト変数](#)や疑問符 (?) の[入力パラメータ](#)を自由に組み合わせて指定することもできます。次に例を示します。

### ObjectScript

```
SET a=7,b="A",c=99
&sql(CALL MyProc(:a,:b,:c))
```

### ObjectScript

```
&sql(CALL MyProc(?, :b, ?))
```

既定では、埋め込み SQL 内の最初の CALL 文の呼び出しにより、%sqlcontext 変数が作成されます。後続の反復は、この既存の %sqlcontext 変数を使用します。つまり、複数の反復が累積して、%sqlcontext が <STORE> エラーになる可能性があります。CALL 文が繰り返して反復される場合は、USING 節で %sqlcontext 変数を明示的に指定する必要があります。USING 節内でプロシージャ・コンテキストが指定されると、InterSystems IRIS は、呼び出されるたびにプロシージャ・コンテキストで [NEW](#) を発行します。

出力の arg に使用されるホスト変数には、単一値、配列の参照、1 つの oref.property の参照、または多次元 oref.property の参照を指定できます。

次のように、ホスト変数または疑問符 (?) を使用して、CALL 文から値を返すことができます。

### ObjectScript

```
&sql(:rttnval=CALL MyProc())
```

### ObjectScript

```
&sql(?=CALL MyProc())
```

CALL の返り値は、単一値にする必要があります。埋め込み SQL の CALL 文から結果セットを返すことはできません。値を返さないプロシージャに `retval=CALL` 構文を使用しようとすると SQLCODE -371 エラーが生成されます。

詳細は、“[埋め込み SQL の使用法](#)”を参照してください。

## ダイナミック SQL からの呼び出し

次のダイナミック SQL の例では、ストアード・プロシージャ `Sample.PersonSets` を呼び出し、このプロシージャから `Sample.Person` テーブルに対して 2 つのクエリを実行します。このストアード・プロシージャの引数では、これら 2 つのクエリで使用する WHERE 節の値を指定します。最初の引数では、最初のクエリで Name 値が `arg1` (この場合、文字 “M”) で始まるすべてのレコードを返すように指定しています。2 番目の引数では、2 番目のクエリで `Home_State = arg2` (この場合、“MA”) を満たすすべてのレコードを返すように指定しています。

### ObjectScript

```
SET mycall = "CALL Sample.PersonSets(?, 'MA')"
```

```
SET tStatement = ##class(%SQL.Statement).%New()
```

```
SET qStatus = tStatement.%Prepare(mycall)
```

```
IF qStatus=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
```

```
SET rset = tStatement.%Execute("M")
```

```
IF rset.%SQLCODE '= 0 {WRITE "SQL error=", rset.%SQLCODE QUIT}
```

```
DO rset.%Display()
```

次のダイナミック SQL の例でも、ストアード・プロシージャ `Sample.PersonSets` を呼び出し、それぞれのクエリに対して個別に結果セットを返します。`%Next()` メソッドは、最初のクエリ結果セットで繰り返し処理を行います。`%MoreResults()` メソッドは、2 番目のクエリの結果セットにアクセスします。クエリが 3 つ以上ある場合、`%MoreResults()` は各結果セットに順番にアクセスします。

### ObjectScript

```
#include %occStatus
```

```
set mycall = "CALL Sample.PersonSets(?, 'MA')"
```

```
set tStatement = ##class(%SQL.Statement).%New()
```

```
set qStatus = tStatement.%Prepare(mycall)
```

```
if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}
```

```
set rset = tStatement.%Execute("M")
```

```
if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}
```

```
FirstResultSet
```

```
while rset.%Next()
```

```
{
```

```
    write "Name: ", rset.%Get("Name")
```

```
    if rset.%Get("Spouse") {write " Spouse: ", rset.%Get("Spouse"), !}
```

```
    else {write " unmarried", !}
```

```
}
```

```
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}
```

```
write !, "1st row count=", rset.%ROWCOUNT, !!
```

```
SecondResultSet
```

```
while rset.%MoreResults()
```

```
{
```

```
    do rset.%CurrentResult.%Display()
```

```
}
```

`%Next()` を呼び出す前に、CALL の実行によって設定された `%SQLCODE` 値を確認することが重要です。`%Next()` メソッドを呼び出すと、その前の CALL の `%SQLCODE` 値を上書きする形で、`%SQLCODE` が設定されます。`%Next()` が結果セット・データを受け取らない場合、このメソッドは `%SQLCODE=100` と設定します。このメソッドは、空の結果セット (行が選択されていない) と、CALL 処理のエラーに起因する存在しない結果セットを区別しません。

`%SQL.Statement` の詳細、およびストアード・プロシージャの仮パラメータやその他のメタデータのリストを表示する方法については、“[ダイナミック SQL の使用法](#)”を参照してください。また、`%Display()` メソッドの詳細と例は、“[完全な結果セットの返送](#)”を参照してください。`%Next()` メソッドおよび `%Get()` メソッドの詳細および例については、“[結果セットからの特定値の返送](#)”を参照してください。



## ObjectScript からの呼び出し

埋め込み SQL から直接ストアード・プロシージャを呼び出すのではなく、ストアード・プロシージャを持つクラス・メソッドを ObjectScript から呼び出すことによって、そのストアード・プロシージャを実行することもできます。この場合、パラメータの管理が必要になります。また、クエリに基づいたストアード・プロシージャでは、個別のメソッドを呼び出し、フェッチ・ループを管理する必要もあります。

例えば、引数を持たない UpdateAllAvgScores というストアード・プロシージャとして公開されたメソッドを呼び出すためのコードは、以下のようになります。

### ObjectScript

```
NEW phnd
SET phnd=##class(%SQLProcContext).%New()
DO ##class(students).UpdateAllAvgScores(phnd)
IF phnd.%SQLCODE {QUIT phnd.%SQLCODE}
USE 0
WRITE !,phnd.%ROWCOUNT," Rows Affected"
```

CALL 文でプロシージャの引数を指定する際、そのプロシージャで %Library.SQLProcContext パラメータが明示的に定義されている場合は、%Library.SQLProcContext パラメータを指定しないでください。このパラメータは自動的に処理されます。

以下の例では、ストアード・プロシージャは2つの引数を取ります。また、明確に定義されたプロシージャ・コンテキストを使用します。

### ObjectScript

```
NEW phnd
SET phnd=##class(%SQLProcContext).%New()
SET rtn=##class(Sample.ResultSets).PersonSets("D","NY")
IF phnd.%SQLCODE {QUIT phnd.%SQLCODE}
DO %sqlcontext.%Display()
WRITE !,"All Done"
```

クエリとして実装されているストアード・プロシージャを呼び出すためには、以下の3つのメソッドすべてを呼び出す必要があります。

### ObjectScript

```
NEW qhnd
DO ##class(students).GetAvgScoreExecute(.qhnd,x1)
NEW avgrow,AtEnd
SET avgrow=$lb("")
SET AtEnd=0
DO ##class(students).GetAvgScoreFetch(.qhnd,.avgrow,.AtEnd)
SET x5=$lg(avgrow,1)
DO ##class(students).GetAvgScoreClose(qhnd)
```

クエリ・ベースのストアード・プロシージャを多くのストアード・プロシージャ内に入れ子にする場合、ラップ・メソッドを記述してすべて非表示にすると便利です。

## ODBC または JDBC からの呼び出し

InterSystems IRIS は、ODBC 2.x 標準および JDBC 1.0 標準で定義されている CALL 構文を全面的にサポートしています。JDBC では、**CallableStatement** クラスのメソッドから CALL を実行できます。ODBC では、API を利用します。CALL の構文と意味は、JDBC と ODBC でまったく同じです。さらに、処理方法も同様です。どちらのドライバも文のテキストを解析し、CALL 文の場合は SQL エンジンを経由せずにサーバ側の専用メソッドを直接呼び出します。

PERSON クラスに SP1 と呼ばれるストアード・プロシージャが存在する場合、ODBC や JDBC クライアント (Microsoft Query など) からこのプロシージャを以下のように呼び出すことができます。

```
retcode = SQLExecDirect(hstmt, "{?=call PERSON_SP1(?,?)}", SQL_NTS);
```

InterSystems IRIS では、ストアド・プロシージャを呼び出す文の構造は、ODBC 標準に準拠しています。この標準の情報に関しては、関連するドキュメントを参照してください。

ODBC で使用する場合にのみ、InterSystems IRIS では呼び出しに柔軟な構文を適用できるので、CALL の前後の { } 括弧やパラメータの前後の括弧は必要ありません(これは、適切なプログラミング形式です。上記の例はこれらを使用しています)。

同様に、ODBC で使用する場合にのみ、InterSystems IRIS では、既定のパラメータを使用するために変更された構文を適用できます。したがって、CALL SP は CALL SP() とは異なる機能を持ちます。2 番目のフォームは、既定のパラメータを使用しないことを意味します – これは CALL SP ( , , ) や SP ( , ? , )、または他の構文などと同様です。つまり、CALL では、括弧を使用したフォームと括弧を使用しないフォームは異なります。

## 関連項目

- ・ SQL 文 : [CREATE PROCEDURE](#)、[CREATE QUERY](#)、[CREATE METHOD](#)
- ・ ObjectScript : [DO コマンド](#)
- ・ [ストアド・プロシージャの定義と使用](#)
- ・ [SQLCODE エラー・メッセージ](#)

# CANCEL QUERY (SQL)

現在システム上で実行されているクエリをキャンセルします。

## 構文

```
CANCEL QUERY pid [ IDENTIFIED BY sql-id ]
[ TIMEOUT timeout ]
```

## 説明

クエリが消費するシステム・リソースが多すぎる場合、その実行をキャンセルすることができます。CANCEL QUERY コマンドは、クエリの実行をキャンセルします。クエリは、そのクエリが実行されているプロセスの ID、およびオプションでそのクエリの SQL Statement ID を指定することによりキャンセルされます。キャンセルされたクエリは引き続き準備が整った状態であるため、初めて実行中のクエリをキャンセルしても、クエリ・キャッシュは生成されます。

文インデックスに格納されている SQL Statement ID は、最初にその文が実行されたときに割り当てられ、変更されることはありません。この ID は、INFORMATION\_SCHEMA.STATEMENTS にクエリを実行するか、自分のインスタンスで現在実行中の文であれば、INFORMATION\_SCHEMA.CURRENT\_STATEMENTS にクエリを実行することによっても確認できます。

クエリは、`$SYSTEM.SQL.CancelQuery()` メソッドを使用してキャンセルすることもできます。

指定されたプロセス ID がクエリを実行していない場合、CANCEL QUERY コマンドは SQLCODE -400 で失敗します。

## 特権

CANCEL QUERY または、別のユーザが発行した `$SYSTEM.SQL.CancelQuery()` を使用してクエリをキャンセルするユーザには、`%CANCEL_QUERY` 特権が必要です。

## 引数

### pid

SQL クエリが実行されているプロセスを特定するプロセス ID。[\\$JOB](#) を使用して、プロセス ID を特定します。

sql-id 引数が指定されない場合、システムは、そのプロセス内で実行される最初のクエリをキャンセルします。特定のクエリをキャンセルするには、sql-id 引数を指定する必要があります。

### sql-id

SQL 文インデックス内に格納される SQL クエリの ID を指定する引数 (オプション)。この引数が省略された場合、システムは、指定されたプロセス内で実行される最初のクエリをキャンセルします。特定のクエリをキャンセルするには、sql-id 引数を指定する必要があります。

### timeout

指定したクエリをキャンセルする前に待機する秒数を指定する引数 (オプション)。省略した場合、クエリは既定で直ちにキャンセルされます。

## 例

以下の例は、プロセス 8044 で実行中のクエリをキャンセルします。

### SQL

```
CANCEL QUERY 8044
```

以下の例は、SQL Statement ID が 68 の、プロセス 12889 で実行中のクエリをキャンセルします。

### SQL

```
CANCEL QUERY 12889 IDENTIFIED BY 68
```

以下の例は、SQL Statement ID が 104 の、プロセス 10455 で実行中のクエリを、30 秒待機した後にキャンセルします。

### SQL

```
CANCEL QUERY 10455 IDENTIFIED BY 104 TIMEOUT 30
```

## 関連項目

- ・ ["CALL \(SQL\)"](#) および ["\\$SYSTEM.SQL.CancelQuery\(\)"](#)
- ・ ["INFORMATION\\_SCHEMA.STATEMENTS"](#) および ["INFORMATION\\_SCHEMA.CURRENT\\_STATEMENTS"](#)
- ・ [\\$JOB](#)

# CASE (SQL)

何らかの条件に従って、特定の値の組み合わせを選択します。

## 構文

```
CASE WHEN search_condition THEN value_expression
[ WHEN search_condition THEN value_expression ... ]
[ ELSE value_expression ]
END

CASE value_expression WHEN value_expression THEN value_expression
[ WHEN value_expression THEN value_expression ... ]
[ ELSE value_expression ]
END
```

## 引数

引数	説明
search_condition	SQL ブーリアン式。
value_expression	SQL 式 (リテラル値やフィールド名など)。

## 概要

CASE 式を使用すると、一連の値に対する比較テストを実行し、最初に値が一致したときに返り値を得ることができます。

CASE 式には単純と検索の 2 つの形式があります。

単純 CASE 式は、(WHEN 節で指定した) 一連の値式をテストし、与えられた値式と等しいかどうかを調べます。

## SQL

```
SELECT
CASE Field1
  WHEN 1 THEN 'ONE'
  WHEN 2 THEN 'TWO'
  ELSE NULL
END
FROM MyTable
```

最初に一致する式に対応する値が CASE 式の値として返されます。

数値の value\_expression 値には、さまざまな複数のデータ型を使用できます。返されるデータ型は、考えられるすべての結果値との互換性が最も高く、[データ型の優先順位](#)が最も高いデータ型です。value\_expression 値が数値の場合、CASE は可能なすべての結果値の中から、最大の長さ、有効桁数、および小数桁数を返します。NULL の結果値はデータ型の優先順位が最下位になりますが、すべての結果値が NULL の場合、返されるデータ型は VARCHAR です。

検索 CASE 式は、(WHEN 節で指定した) 一連の検索条件をテストし、評価が True になる最初の WHEN 条件を検索し、対応する値を返します。

## SQL

```
SELECT
CASE
  WHEN Field1 = 1 THEN 'ONE'
  WHEN Field1 = 2 THEN 'TWO'
  ELSE NULL
END
FROM MyTable
```

どちらの CASE 式の形式でも、WHEN 節の条件が True でない場合に返す値を ELSE 節で指定できます。ELSE 節を省略した際に、どの WHEN 節の条件も True にならない場合、CASE は NULL を返します。

NULL についての CASE 比較には、IS NULL または IS NOT NULL のいずれかのキーワード句を使用する必要があります。NULL はデータ値ではありません（値が存在しないことを示します）。そのため、NULL の等式テストまたは算術テストは常に False を返します。NULL と任意のデータ値を比較する CASE 式は常に False を返します。例えば、NULL < 1 と NULL > 1 は両方とも False を返します。NULL どうしを比較する CASE 式も常に False を返します。

CASE 式の最後には END トークンを記述します。

## 例

以下のクエリは単純 CASE 式の例で、ここでは指定したフィールドの値を指定した値に置き換えます。END キーワードの後に列のエイリアス RetireAge を記述しています。また、オプションの AS キーワードは省略しています。

### SQL

```
SELECT Name,
CASE Age
  WHEN 65 THEN 'Retire this year'
  WHEN 64 THEN 'Retire next year'
  ELSE 'Past retirement age ' || Age
END RetireAge
FROM Sample.Person
WHERE Age > 63
ORDER BY Age
```

以下のクエリは、別の単純 CASE 式の例です。このクエリでは、特定の Home\_State 値を持つ行に “Northern NE” または “Southern NE” のラベルを付け、この列のその他の Home\_State 値を NULL に設定します。さらに、As 節を使用してこの列に “NewEnglanders” のラベルを付け、Name と元の Home\_State 値を表示します。この結果得られる行は、まず NewEnglanders 列の値の降順で並べられ、その中で Home\_State、Name の順でアルファベット順に並べられます。

### SQL

```
SELECT Name,
CASE Home_State
  WHEN 'VT' THEN 'Northern NE'
  WHEN 'NH' THEN 'Northern NE'
  WHEN 'ME' THEN 'Northern NE'
  WHEN 'MA' THEN 'Southern NE'
  WHEN 'CT' THEN 'Southern NE'
  WHEN 'RI' THEN 'Southern NE'
  ELSE NULL
END AS NewEnglanders, Home_State
FROM Sample.Person
ORDER BY NewEnglanders DESC, Home_State, Name
```

以下のクエリは、検索 CASE 式の例です。ここでは、論理演算子（不等号 >、論理積 &、論理和 !）を使用して、WHEN 節ごとにブーリアン文を指定します。True をテストする最初の WHEN 節によって、THEN キーワードに続く値式が設定されます。この例では、Age フィールドと Home\_State フィールドの値を使用して、Yankees の値を、Old Yankees、Yankees（New England の 6 州の住民）、および野球チーム New York Yankees のファンと思われる人のいずれかに特定します。

### SQL

```
SELECT Name,
CASE
  WHEN Age > 55 & Home_State = 'VT'
    ! Home_State='ME' ! Home_State='NH'
    ! Home_State='MA' ! Home_State='CT'
    ! Home_State='RI'
  THEN 'Old Yankee'
  WHEN Home_State = 'VT'
    ! Home_State='ME' ! Home_State='NH'
    ! Home_State='MA' ! Home_State='CT'
    ! Home_State='RI'
  THEN 'Yankee'
  WHEN Home_State='NY' THEN 'Yankees Fan'
  ELSE Home_State
END AS Yankees
FROM Sample.Person
```

以下の例では、どのような値と NULL を比較しても必ず False が返されることを示しています。

## SQL

```
SELECT TOP 5 Name,
CASE NULL
  WHEN NULL THEN 'Null = Null'
  WHEN 0 THEN 'Null = 0'
  WHEN '' THEN 'Null = empty string'
  WHEN CHAR(0) THEN 'Null = CHAR(0)'
  ELSE 'Null Arithmetic Invalid'
END
FROM Sample.Person
```

以下の例は、NULL を持つフィールドで CASE を使用する方法を示しています。

## SQL

```
SELECT TOP 20 Name,
CASE
  WHEN FavoriteColors IS NULL THEN 'No Colors'
  ELSE $LISTTOSTRING(FavoriteColors,':')
END
FROM Sample.Person
```

以下の例のように、CASE はクエリでの使用に限定されません。

## SQL

```
INSERT INTO SQLUser.MyStudents (Name, PxTs) VALUES (
CASE ?
  WHEN 'a' THEN 'Alice'
  WHEN 'b' THEN 'Barney'
  ELSE 'Unknown' END,
CURRENT_TIMESTAMP)
```

## 関連項目

- SQL 関数: [DECODE](#)、[GREATEST](#)、[LEAST](#)、[NULLIF](#)、[COALESCE](#)
- ObjectScript 関数: [\\$CASE](#)

## %CHECKPRIV (SQL)

指定の特権がユーザにあるかどうかを確認します。

### 構文

```
%CHECKPRIV [GRANT OPTION FOR | ADMIN OPTION FOR] syspriv [,syspriv]
%CHECKPRIV [GRANT OPTION FOR] objpriv
    ON object
%CHECKPRIV column-privilege (column-list)
    ON table
```

### 概要

%CHECKPRIV には以下の 2 つの使用方法があります。

- ・ 現在のユーザに指定されたシステム特権があるかどうか、またはコンマ区切りリストで指定されたすべてのシステム特権があるかどうかを確認する。
- ・ 現在のユーザに、指定のオブジェクトに対する指定のタイプのユーザ特権があるかどうかを確認する。これらのオブジェクトには、テーブルまたはビューに対するテーブルレベルの特権、指定された列に対する列レベルの特権、およびストアド・プロシージャに対する特権を指定できます。

ユーザが指定された特権を保持している場合、%CHECKPRIV は SQLCODE=0 を設定します。ユーザが指定された特権を保持していない場合、%CHECKPRIV は SQLCODE=100 を設定します。

%CHECKPRIV を使用すると、特権を所有しているかどうかを確認できます。それによって特権を強要するわけではありません。

- ・ [埋め込み SQL](#) は、SQL 特権を強要しません。%CHECKPRIV は、主に埋め込み SQL のために使用されます。“[埋め込み SQL と特権](#)”を参照してください。
- ・ [ダイナミック SQL](#) は、実行時に特権を強要します。例えば、%CREATE\_TABLE システム特権がない場合、%CHECKPRIV %CREATE\_TABLE は SQLCODE=100 を設定し、この特権がないことを示します。ダイナミック SQL はこの特権を強要します。CREATE TABLE 操作は SQLCODE -99 エラーで失敗します。

実行時、ダイナミック SQL と ODBC/JDBC は特権を強要し、適切なエラーを生成します。管理ポータルの [\[クエリ実行\]](#) SQL インタフェースと SQL シェルはどちらもダイナミック SQL として実行します。

%CHECKPRIV は、その結果を判断するために SQLCODE 100 値 (SQLCODE 状態値であり、SQLCODE エラー値ではない) にアクセスする必要があるため、エラー状態とエラーのない状態とを区別することしかできない JDBC および他のクライアントから直接 %CHECKPRIV を使用することはできません。

%CHECKPRIV はすぐに作成および実行され、通常一度しか実行されないため、InterSystems IRIS では、%CHECKPRIV にはクエリ・キャッシュは作成されません。

### CheckPrivilege() メソッド

\$SYSTEM.SQL.Security.CheckPrivilege() メソッドは、テーブル、ビュー、またはストアド・プロシージャに対するユーザの特権のチェックのためのより優れた機能を提供します。

- ・ CheckPrivilege() は、指定されたユーザの特権をチェックします。%CHECKPRIV は、現在のユーザの特権のみをチェックします。
- ・ CheckPrivilege() は、複数の特権をチェックできます。%CHECKPRIV の呼び出しごとに 1 つの objpriv 特権のみをチェックできます。
- ・ CheckPrivilege() では、別のネームスペースで定義されたテーブル、ビュー、またはプロシージャに対する特権をチェックできます。%CHECKPRIV は、現在のネームスペース内のオブジェクトの特権のみをチェックします。



## 埋め込み SQL と特権

埋め込み SQL については、特権が自動的に確認されることや適用されることはありません。したがって、埋め込み SQL プログラムでは(ほとんどの場合)、更新作業のように特権を必要とする操作を実行する前に、%CHECKPRIV を呼び出しておく必要があります。

### ObjectScript

```
SET name="Fred",age=25
SET SQLCODE=""
&sql(%CHECKPRIV UPDATE ON Sample.Person)
IF SQLCODE=100 {
    WRITE !,"No UPDATE privilege"
    QUIT }
ELSEIF SQLCODE < 0 {
    WRITE !,"Unexpected SQL error: ",SQLCODE," ",%msg
    QUIT }
ELSE {
    WRITE !,"Proceeding with UPDATE" }
&sql(UPDATE Sample.Person SET Name=:name,Age=:age WHERE Address='123 Bedrock')
IF SQLCODE=0 { WRITE !,"UPDATE successful" }
ELSE { WRITE "UPDATE error SQLCODE=",SQLCODE }
```

## 引数

### GRANT OPTION FOR

このキーワードは、現在のユーザが指定の特権について WITH GRANT OPTION 特権を持っているかどうか確認することを指定します(オプション)。このオプションを設定した %CHECKPRIV では、ユーザが指定の特権を持っているかどうかを確認されません。

### ADMIN OPTION FOR

このキーワードは、現在のユーザが、指定の特権を他のユーザやロールに付与できるかどうか確認することを指定します(オプション)。このオプションを設定した %CHECKPRIV では、ユーザが指定の特権を持っているかどうかを確認されません。

### syspriv

1 つのシステム特権です。または、複数のシステム特権をコンマで区切って列挙した一覧です。使用可能な syspriv オプションには、16 個のオブジェクト定義特権と 4 つのデータ変更特権が含まれています。

それらのオブジェクト定義特権は、%CREATE\_FUNCTION、%DROP\_FUNCTION、%CREATE\_METHOD、%DROP\_METHOD、%CREATE\_PROCEDURE、%DROP\_PROCEDURE、%CREATE\_QUERY、%DROP\_QUERY、%CREATE\_TABLE、%ALTER\_TABLE、%DROP\_TABLE、%CREATE\_VIEW、%ALTER\_VIEW、%DROP\_VIEW、%CREATE\_TRIGGER、%DROP\_TRIGGER です。%DB\_OBJECT\_DEFINITION を指定すれば、この 16 個のオブジェクト定義特権すべてについて確認できます。

データ変更特権は、INSERT、UPDATE、および DELETE 操作のための %NOCHECK、%NOINDEX、%NOLOCK、%NOTRIGGER 特権です。

### objpriv

指定の object に関係するオブジェクト特権です。指定できるオプションは、%ALTER、DELETE、SELECT、INSERT、UPDATE、EXECUTE、および REFERENCES です。

### object

objpriv の確認対象とするオブジェクトの名前です。

### column-privilege

1 つ以上のリストされた列に関連付けられている列レベルの特権。指定できるオプションは、SELECT、INSERT、UPDATE、および REFERENCES です。

## column-list

特権の割り当てをチェックする 1 つ以上の列名のリスト。このリストは括弧で囲み、リスト内の項目はコンマで区切ります。column-privilege 名と最初の括弧の間には、スペースを入れることも省略することもできます。

## table

column-list 列を含む**テーブル**またはビューの名前。テーブル名またはビュー名は修飾 (schema.tablename)、未修飾 (tablename) のどちらでもかまいません。未修飾名は**既定のスキーマ名**を取ります。**スキーマ検索パス**は無視されます。

## 例

以下の埋め込み SQL の例では、特定のテーブルに対する特定のオブジェクト特権が現在のユーザにあるかどうかを確認します。

### ObjectScript

```
&sql(%CHECKPRIV UPDATE ON Sample.Person)
IF SQLCODE=0 {WRITE "Have update privilege"}
ELSEIF SQLCODE=100 {WRITE "Do not have update privilege" QUIT}
ELSE {WRITE "Unexpected %CHECKPRIV error: ",SQLCODE," ",%msg QUIT}
```

以下の埋め込み SQL の例では、3 種類のテーブル操作のシステム特権が現在のユーザにあるかどうかを確認します。特権がある場合、ユーザは以下のようにテーブルを作成します。

### ObjectScript

```
&sql(%CHECKPRIV %CREATE_TABLE,%ALTER_TABLE,%DROP_TABLE)
IF SQLCODE=0 {WRITE "Have table privileges",!}
ELSEIF SQLCODE=100 {WRITE "Do not have one or more table privileges" QUIT}
ELSE {WRITE "Unexpected %CHECKPRIV error: ",SQLCODE," ",%msg QUIT}
&sql(CREATE TABLE Sample.MyTable (Name VARCHAR(40),Age INTEGER))
WRITE "Created table"
```

以下の埋め込み SQL の例では、16 個すべてのオブジェクト定義特権が現在のユーザにあるかどうかを確認します。SQLCODE 値は、16 個のオブジェクトすべてに対する特権があれば 0、1 つでも特権のないオブジェクトがあれば 100 に設定されます。

### ObjectScript

```
&sql(%CHECKPRIV %DB_OBJECT_DEFINITION)
IF SQLCODE=0 {WRITE "Have all system privileges"}
ELSEIF SQLCODE=100 {WRITE "Do not have one or more system privileges"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE," ",%msg}
```

以下の埋め込み SQL の例では、現在のユーザが、他のユーザやロールに %CREATE\_TABLE 特権を付与できるかどうかを確認します。

### ObjectScript

```
&sql(%CHECKPRIV ADMIN OPTION FOR %CREATE_TABLE)
IF SQLCODE=0 {WRITE "Have admin option on privilege"}
ELSEIF SQLCODE=100 {WRITE "Do not have admin option on privilege"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE," ",%msg}
```

以下の埋め込み SQL の例では、指定した列レベルの特権が現在のユーザにあるかどうかを確認します。特権の名前に続いて、列の名前 (またはコンマで区切られた列のリスト) を括弧で囲んで指定します。

### ObjectScript

```
&sql(%CHECKPRIV UPDATE(Name,Age) ON Sample.Person)
IF SQLCODE=0 {WRITE "Have privilege on all specified columns"}
ELSEIF SQLCODE=100 {WRITE "Do not have privilege on one or more specified columns"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE," ",%msg}
```

## 関連項目

- ・ SQL 文 : [GRANT](#)、[REVOKE](#)
- ・ [SQL のユーザ、ロール、および特権](#)
- ・ ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数

## CLOSE (SQL)

カーソルをクローズします。

### 構文

```
CLOSE cursor-name
```

### 概要

CLOSE 文は、オープンしているカーソルをクローズします。これは現在の結果セットをリリースし、現在カーソルがある行によって設定されたカーソル・ロックを解放します。ただし、CLOSE はカーソルを削除しません。再度オープンするときのためにデータ構造をアクセス可能なままにしますが、カーソルを再度オープンするまで、フェッチや位置付け更新はできません。この動作は以下のコマンド・シーケンスによって示されます。

- ・ DECLARE c1、OPEN c1、FETCH c1、CLOSE c1 は標準シーケンスです。
- ・ DECLARE c1、OPEN c1、CLOSE c1、OPEN c1 は、宣言されたカーソル c1 を再度オープンします。
- ・ DECLARE c1、OPEN c1、CLOSE c1、DECLARE c1、OPEN c1 は、最初の DECLARE で指定されたカーソルを再度オープンし、2 番目の DECLARE は無視します。
- ・ DECLARE c1、OPEN c1、FETCH c1、CLOSE c1、OPEN c1、FETCH c1 は、両方のフェッチ操作を行って、同じレコードを取得します。

CLOSE は、オープンされたカーソルで発行する必要があります。宣言されただけの（オープンされていない）カーソルや、既にクローズされたカーソルで CLOSE を発行すると、SQLCODE -102 エラーが発生します。存在しないカーソル（定義されたカーソルと大文字/小文字が違うカーソルなど）で CLOSE を発行すると、SQLCODE -52 エラーが発生します。

*cursor-name* はネームスペース固有ではありません。現在のネームスペースを変更しても、宣言されたカーソルの使用には影響はありません。ネームスペースで考慮する必要がある唯一の点は、FETCH は、クエリ対象のテーブルを含むネームスペースで行う必要があるということです。

SQL 文では、CLOSE は埋め込み SQL でのみサポートされることに注意してください。同様の操作は、ODBC でも ODBC API を使用してサポートされます。

### 引数

#### *cursor-name*

クローズするカーソルの名前。カーソル名は DECLARE 文で指定されています。カーソル名は、大文字と小文字を区別します。

### 例

以下は、EmpCursor という名前のカーソルをオープンしてクローズする埋め込み SQL の例です。

## ObjectScript

```

SET name="LastName,FirstName",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
      SELECT Name, Home_State
      INTO :name,:state FROM Sample.Employee
      WHERE Home_State %STARTSWITH 'A')
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
      QUIT:SQLCODE
      WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"After FETCH SQLCODE: ",SQLCODE
WRITE !,"After FETCH row count: ",%ROWCOUNT
&sql(CLOSE EmpCursor)
IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
WRITE !,"After CLOSE SQLCODE: ",SQLCODE
WRITE !,"After CLOSE row count: ",%ROWCOUNT
WRITE !,"AFTER: Name=",name," State=",state

```

カーソルをクローズした後も、ホスト変数には最後にフェッチしたデータ値が設定されたままになっており、%ROWCOUNT には取得した行数が設定されたまま残っている点に注意してください。ただし、フェッチ終了時の SQLCODE の値 (SQLCODE=100) は、CLOSE の SQLCODE の値 (SQLCODE=0) で上書きされます。

以下の埋め込み SQL の例は、複数ネームスペース間でカーソルが持続することを示しています。このカーソルは %SYS で宣言され、USER でオープンおよびフェッチされ、SAMPLES でクローズされます。OPEN は、クエリされるテーブルを含むネームスペースで実行する必要があること、および FETCH は出力ホスト変数にアクセスできる必要があり、出力ホスト変数はネームスペース固有であることに注意してください。

```

&sql(USE DATABASE %SYS)
WRITE $ZNSPACE,!
&sql(DECLARE NSCursor CURSOR FOR SELECT Name INTO :name FROM Sample.Employee)
&sql(USE DATABASE "USER")
WRITE $ZNSPACE,!
&sql(OPEN NSCursor)
IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW SQLCODE,%ROWCOUNT,%ROWID
FOR { &sql(FETCH NSCursor)
      QUIT:SQLCODE
      WRITE "Name=",name,! }
&sql(USE DATABASE SAMPLES)
WRITE $ZNSPACE,!
&sql(CLOSE NSCursor)
IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}

```

## 関連項目

- [DECLARE、FETCH、OPEN](#)
- [SQL カーソル](#)

## COMMIT (SQL)

トランザクション中に実行された処理をコミットします。

### 構文

COMMIT [WORK]

### 概要

COMMIT 文は、現在のトランザクション中に完了したすべての処理をコミットし、トランザクション・レベル・カウンタをリセットして、設定されたすべてのロックを解除します。これによりトランザクションが完了します。コミットされた処理はロールバックできません。

COMMIT 文と COMMIT WORK 文は同等です。両者には互換性があります。

トランザクションは、トランザクションの開始を表す START TRANSACTION 文を含む操作として定義されます。COMMIT を実行すると、トランザクション・レベル・カウンタ (\$TLEVEL) が、そのトランザクションを初期化した START TRANSACTION 文の直前の状態にリストアされます (InterSystems SQL は入れ子になったトランザクションをサポートしないため、トランザクション内で START TRANSACTION 文を追加発行しても、トランザクションの初期化ポイントには影響しません)。

単一の COMMIT によりトランザクション内のすべてのセーブポイントがコミットされます。

START TRANSACTION 文は新しいトランザクションを明示的に開始するものです。ただし、START TRANSACTION を使用するかどうかは任意です。トランザクション処理がアクティブなとき、最初のデータベース操作に続いて COMMIT が実行されると、新しいトランザクションが暗黙的に開始されます。トランザクション処理が反映されない場合や、自動コミットによって反映される場合、COMMIT 文は意味を持ちません。進行中のトランザクションがなければ、COMMIT は正常に完了しますが (SQLCODE 0)、処理は何も実行されません。

クエリでの COMMIT の結果は現在の分離レベルによって異なります。これらのトランザクション・パラメータは、SET TRANSACTION または START TRANSACTION コマンドのどちらでも設定できます。

トランザクションの動作が正常完了できなかった場合は、SQLCODE -400 が発行されます。

### ObjectScript と SQL のトランザクション

ObjectScript と SQL のトランザクション・コマンドは完全に互換性があり、置き換え可能ですが、以下の例外があります。

ObjectScript TSTART と SQL START TRANSACTION はどちらも、トランザクションが進行中でない場合にトランザクションを開始します。ただし、START TRANSACTION では、入れ子になったトランザクションはサポートされません。そのため、入れ子になったトランザクションが必要な場合 (または必要になる可能性がある場合) には、トランザクションを TSTART で始めることをお勧めします。SQL 標準との互換性が必要な場合は、START TRANSACTION を使用してください。

ObjectScript トランザクション処理は、入れ子になったトランザクションを限定的にサポートします。SQL トランザクション処理はトランザクション内のセーブポイントをサポートします。

トランザクションに SQL 更新文が含まれる場合、SQL の START TRANSACTION 文でトランザクションが開始され、COMMIT 文でコミットされます。TSTART/TCOMMIT を入れ子にして使用するメソッドは、トランザクションを開始するものでなければ、トランザクションに組み込むことができます。メソッドとストアード・プロシージャは、通常、設計でトランザクションの主要なコントローラにならない限り、SQL トランザクション制御文を使用しません。ストアード・プロシージャは、独自のトランザクション制御モデルの ODBC/JDBC から呼び出されるため、通常 SQL トランザクション制御文を使用しません。

### 例

以下の埋め込み SQL の例は、トランザクション内で確立された SAVEPOINTS の数に関係なく、COMMIT がトランザクション・レベル・カウンタ (\$TLEVEL) を START TRANSACTION の直前のレベルにリストアする方法を示しています。このプログラム内の 2 番目の START TRANSACTION は、\$TLEVEL に影響しない空命令です。

## ObjectScript

```

&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT b)
WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION) /* Performs no operation */
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT c)
WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL

```

以下の埋め込み SQL の例では、COMMIT 文がトランザクション全体をコミットします。余分な COMMIT 文は何も影響を与えず、エラーも発生しません。

## ObjectScript

```

&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT) /* Performs no operation */
WRITE !,"Commit again, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT) /* Performs no operation */
WRITE !,"Commit again, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL

```

## 関連項目

- SQL コマンド : [ROLLBACK SAVEPOINT SET TRANSACTION START TRANSACTION \\$TLEVEL](#)
- [トランザクション処理](#)
- ObjectScript コマンド: [TCOMMIT](#)



# CREATE AGGREGATE (SQL)

ユーザ定義の集約関数を作成します。

## 構文

```
CREATE [OR REPLACE] AGGREGATE name(parameter_list) [ RETURNS datatype ]
[ INITIALIZE WITH function-name ]
[ ITERATE WITH function-name ]
[ MERGE WITH function-name ]
[ FINALIZE WITH function-name ]
```

## 説明

CREATE AGGREGATE コマンドは、ユーザ定義の集約関数 (UDAF) を作成します。呼び出されると、このユーザ定義集約関数は、行の値を反復処理し、1 つ以上のユーザ定義関数を呼び出して集約値を計算します。CREATE AGGREGATE を使用すると、標準の InterSystems IRIS SQL [集約関数](#)では提供されない集約演算を提供できます。

CREATE AGGREGATE を呼び出して既に存在する UDAF を作成すると、SQL は SQLCODE -428 エラーを返します。生成される %msg は " `SQLUser.MyUDAF` " のようになります。オプションの OR REPLACE キーワード節 (CREATE OR REPLACE AGGREGATE) を指定する場合、既存の UDAF の名前を指定してもエラーは生成されません。既存の UDAF は指定された定義により更新されます。

ユーザ定義の集約関数を削除するには、[DROP AGGREGATE](#) コマンドを使用します。

## 特権

CREATE AGGREGATE コマンドは特権を必要とする操作です。CREATE AGGREGATE を使用する前に、UDAF と参照されるすべてのユーザ定義関数に対する EXECUTE 特権が必要です。特権がない場合は、SQLCODE -99 エラー (特権違反) が返されます。

## 集約関数名

UDAF name は有効な[識別子](#)である必要があります。集約関数名では、大文字と小文字は区別されません。

UDAF name は修飾 (schema.aggname)、未修飾 (aggname) のどちらでもかまいません。未修飾の name では[既定のスキーマ名](#)が使用されます。

UDAF name は、既存のストアド・プロシージャの名前と同じにはできません。ストアド・プロシージャ名が重複する UDAF の作成を試みると、SQLCODE -428 エラーが生じ、%msg は " `SQLUser.MyFunction` " のようになります。

## INITIALIZE WITH 節

オプションの INITIALIZE WITH 節は、指定したユーザ定義関数またはクラス・メソッドを呼び出し、初期状態オブジェクトを構成します。状態オブジェクトの値を使用して、中間集約値または最後の計算を実行するために必要なその他の変数を渡します。この節が指定されていない場合、ITERATE WITH 節で指定された関数に初期状態オブジェクトとして NULL オブジェクトが渡されます。

指定されたユーザ定義の function-name は、CREATE AGGREGATE が呼び出されたときに存在している必要があります。存在しない場合、SQLCODE -428 エラーが生成され、%msg により UDAF 関数、節、および存在しない関数名が指定されます。

以下に、初期状態オブジェクトを定義するユーザ定義関数を示します。

## SQL

```
CREATE FUNCTION MyAggregateInit() returns varchar language ObjectScript { RETURN "^" }
```



## ITERATE WITH 節

ITERATE WITH 節は、集約される行ごとに一度、指定したユーザ定義関数またはクラス・メソッドを呼び出します。これは、入力パラメータとして中間結果と現在の行の列値を表す状態オブジェクトを取り、その状態オブジェクトに対する操作（集約値を累積します）を実行します。すべての行が処理されると、新しい状態値が返されます。

指定されたユーザ定義の function-name は、CREATE AGGREGATE が呼び出されたときに存在している必要があります。存在しない場合、SQLCODE -428 エラーが生成され、%msg により UDAF 関数、節、および存在しない関数名が指定されます。

## MERGE WITH 節

オプションの MERGE WITH 節を指定すると、ユーザ定義の集約関数の並列処理を有効にできます。指定しない場合は、UDAF を呼び出すクエリがシングル・スレッド処理を使用します。詳細は、“[並列処理](#)”を参照してください。

## FINALIZE WITH 節

オプションの FINALIZE WITH 節は、処理の最後で一度、指定したユーザ定義関数またはクラス・メソッドを呼び出し、ITERATE WITH 節関数への最後の呼び出しから返された状態値に基づいて最後の計算を実行します。呼び出すクエリで GROUP BY 節を指定すると、このユーザ定義関数は GROUP BY でグループ化された値ごとに一度呼び出されます。

指定されたユーザ定義の function-name は、CREATE AGGREGATE が呼び出されたときに存在している必要があります。存在しない場合、SQLCODE -428 エラーが生成され、%msg により UDAF 関数、節、および存在しない関数名が指定されます。

## 引数

### name

作成するユーザ定義の集約関数の名前。name は有効な識別子である必要があります。name は修飾 (schema.aggname)、未修飾 (aggname) のどちらでもかまいません。未修飾の name では既定のスキーマ名が使用されます。集約関数名では、大文字と小文字は区別されません。name の後には 1 つ以上のパラメータが含まれる括弧が続く必要があります。

### parameter\_list

値を集約関数に渡すために使用するパラメータのリストです。このパラメータ・リストは括弧で囲まれます。1 つのパラメータを指定することも、パラメータをコンマで区切ったリストを指定することもできます。リスト内の各パラメータは、パラメータ名とデータ型で構成されます。例：(param1 INTEGER,param2 NUMERIC)

## RETURNS datatype

集約関数の値を返すデータ型を指定する引数（オプション）。省略した場合、データ型は既定で parameter\_list の最初のパラメータのデータ型になります。

### function-name

CREATE FUNCTION コマンドを使用して作成した既存のユーザ定義関数の名前、または SQL プロシージャとして値を返し、投影されるクラス・メソッドの名前です。ユーザ定義関数は、ストアド・プロシージャ・クラスのメソッドとして格納されます。例えば、ユーザ定義関数 MyFunction は、既定のスキーマ名：SQLUser.MyFunction を取ります。これは、クラス・メソッド MyFunction() を含むクラス User.funcMyFunction に対応しています。

## ユーザ定義集約関数の呼び出し

ユーザ定義集約関数は、標準の集約関数と同じ使用規定に従います。

UDAF は、SELECT リストで呼び出されます。リストされた select-item としてか、サブクエリの select-item 内で使用できます。これは、列のエイリアスを指定できます。列のエイリアスが指定されない場合は、既定で Aggregate\_n となります。以下に例を示します。

## SQL

```
SELECT Home_State,AVG(Age) AS AvgAge,MAX(Age) AS MaxAge,SecondHighest(Age) AS SecondMaxAge
FROM Sample.Person GROUP BY Home_State
```

UDAF は **ORDER BY** 節内で直接使用することはできません。これを実行しようとする、SQLCODE -73 エラーが生成されます。ただし、対応する列エイリアスや select-item のシーケンス番号を指定することにより、ORDER BY 節でユーザ定義集約関数を使用できます。

UDAF は、**HAVING** 節で直接使用できます。ただし、HAVING 節は、明示的にユーザ定義集約関数を指定する必要があります。対応する select-item 列エイリアスや select-item シーケンス番号を使用して UDAF を指定することはできません。

集約関数は、以下のような箇所で直接使用することはできません。

- ・ WHERE 節。これを実行しようとする、SQLCODE -19 エラーが生成されます。
- ・ GROUP BY 節。これを実行しようとする、SQLCODE -19 エラーが生成されます。
- ・ TOP 節。これを実行しようとする、SQLCODE -1 エラーが生成されます。
- ・ JOIN。ON 節で集約を指定しようとする、SQLCODE -19 エラーが生成されます。USING 節で集約を指定しようとする、SQLCODE -1 エラーが生成されます。

標準の集約関数とは異なり、ユーザ定義の集約関数は、DISTINCT、%FOREACH、または %AFTERHAVING 節を指定できません。

## 並列処理

オプションの MERGE WITH 節が指定されると、その MERGE WITH 関数は 2 つ以上の並列サブクエリの ITERATE WITH 関数から提供される状態オブジェクトをマージし、集約された状態を表す単一のマージ値を返します。MERGE WITH 関数は、並列処理の回数だけ自動的に呼び出されます。これらのマージの結果は、FINALIZE WITH 節に提供されます。

MERGE WITH 関数を宣言すると、状態オブジェクトは暗黙的なシリアル化をサポートすると見なされます (ObjectScript に %SerialObject インタフェースを実装するなど)。

MERGE WITH 関数が提供されない場合、%PARALLEL またはシャーディングが指定されていないと、ユーザ定義集約関数は並列スレッドにより処理されません。これは単一のスレッドとして処理されます。

## ユーザ定義集約関数のリスト

**INFORMATION.SCHEMA.USERDEFINEDAGGREGATES** 永続クラスは、現在のネームスペース内のすべてのユーザ定義集約関数に関する情報を表示します。これは、その節で指定されるユーザ定義関数の名前を含む、多くのプロパティを提供します。

以下の例では、現在のネームスペース内のユーザ定義のすべての集約関数のスキーマ名、ユーザ定義集約名、ITERATE 関数名、および返されるデータ型を返します。

## SQL

```
SELECT AGGREGATE_SCHEMA,AGGREGATE_NAME,ITERATE_FUNCTION,RETURN_TYPE
FROM INFORMATION_SCHEMA.USER_DEFINED_AGGREGATES
```

RETURNS 節が指定されていなければ、RETURN\_TYPE の値は NULL になります。

## 例

以下の例では、大きな値 (5 以上) はすべて加算し、小さい値 (5 未満) の場合はすべて 5 を減算することにより、ユーザ定義の集約関数を作成します。すべての値はデータ型 NUMERIC(4,1) です。最初に、状態変数 (tot) と入力変数 (num) を指定して、反復関数を作成します。

## SQL

```
CREATE FUNCTION Sample.AddSub(tot NUMERIC(4,1),IN num NUMERIC(4,1)) RETURNS NUMERIC(4,1)
LANGUAGE OBJECTSCRIPT {IF num>=5 {SET tot=tot+num} ELSE {SET tot=tot-5} QUIT tot}
```

次に、ユーザ定義の集約関数を定義します。

```
CREATE AGGREGATE Sample.SumAddSub(arg NUMERIC(4,1))
  ITERATE WITH Sample.AddSub
```

以下のように、Score フィールドでユーザ定義集約関数を呼び出します。

```
SELECT TestSubject,Score,SUM(Score) AS ScoreSum,Sample.SumAddSub(Score) AS ScoreAddHighSubtractLow
```

負の値を回避するには、FINALIZE WITH 関数を追加します。

```
CREATE FUNCTION Sample.NoNeg(tot NUMERIC(4,1)) RETURNS NUMERIC(4,1)
LANGUAGE OBJECTSCRIPT {IF num>0 {QUIT tot} ELSE {SET tot=0 QUIT tot}}
```

```
CREATE OR REPLACE AGGREGATE Sample.SumAddSub(arg NUMERIC(4,1))
  ITERATE WITH Sample.AddSub
  FINALIZE WITH Sample.NoNeg
```

## 関連項目

- ・ [CREATE FUNCTION コマンド](#)
- ・ [DROP AGGREGATE コマンド](#)
- ・ [集約関数の概要](#)
- ・ [SQLCODE エラー・メッセージ](#)

# CREATE DATABASE (SQL)

データベース (ネームスペース) を作成します。

## 構文

```
CREATE DATABASE dbname [ON DIRECTORY pathname]
  [WITH [ENCRYPTED_DB] [GLOBAL_JOURNAL_STATE [=] {YES | NO}]] ]
```

## 概要

CREATE DATABASE コマンドは、1 つの [ネームスペース](#)、およびそれに関連付けられた 2 つの [データベース](#) を作成します。これにより、SQL 内でネームスペースを作成できるようになります。

指定した dbname は、作成されるネームスペース、および対応するデータベース・ファイルを含むディレクトリの名前となります。ネームスペース名は、大文字と小文字を区別しません。dbname は、[SQL 識別子](#) の名前付け規約に従います。また、次のような制限事項もあります。

- アンダースコア ( ) 文字を dbname の先頭文字として使用することは認められていません (ただし、名前の文字列内の他の位置では使用可能です)。@、#、および \$ 文字は、dbname 内での使用は認められていません。これらの無効な文字を dbname 内に含めようとすると、SQLCODE -343 エラーが生成されます。
- ハイフン (-) 文字を dbname 内で使用することは認められていません (ハイフンは有効な SQL 識別子文字ではありません)。ただし、他の方法で作成されたネームスペース名には、ハイフン文字が含まれている場合があります。
- dbname の文字数は 63 文字以内である必要があります。これより長い dbname の場合は、適切な %msg と共に SQLCODE -400 致命的エラーが生成されます。

指定された dbname ネームスペースが既に存在する場合、InterSystems IRIS によって SQLCODE -341 エラーが発行されます。

WITH オプションである ENCRYPTED\_DB および/または GLOBAL\_JOURNAL\_STATE のどちらも指定しないか、どちらか一方、または両方を指定することができます。両方を指定する場合は、それらを WITH ENCRYPTED\_DB GLOBAL\_JOURNAL\_STATE=NO のようにスペースで区切ります。

既定では、CREATE DATABASE は、mgr ディレクトリ内に 2 つのデータベースを作成します。このディレクトリには dbname の名前のサブディレクトリがあり、その下には C (コード) および D (データ) という 2 つのサブディレクトリがあります。これらの各サブディレクトリには、IRIS.DAT ファイル、iris.lck ファイル、および空のストリーム・フォルダが含まれています。例えば、Windows システムでは、CREATE DATABASE Barney によってネームスペース BARNEY および以下のデータベース・ファイルが作成されます。

```
c:\InterSystems\IRIS\mgr\Barney\C containing IRIS.DAT, iris.lck, stream folder
c:\InterSystems\IRIS\mgr\Barney\D containing IRIS.DAT, iris.lck, stream folder
```

C (コード) ディレクトリは、ネームスペースのルーチン・データベースのために使用されます。D (データ) ディレクトリは、ネームスペースのグローバル・データベースのために使用されます。mgr ディレクトリの位置に戻るには、%SYSTEM.Util.ManagerDirectory() メソッドを使用します。

オプションの ON DIRECTORY pathname 節では、データベース・ファイルのために、ネームスペースと同じ名前のディレクトリではなく異なる場所を指定できます。例えば以下ようになります。

## SQL

```
CREATE DATABASE Flintstone ON DIRECTORY 'c:\InterSystems\IRIS\mgr\Fred'
```

指定された pathname が既に存在する場合は、InterSystems IRIS によって SQLCODE -341 エラーが発行されます。

CREATE DATABASE コマンドの実行には特権が必要です。CREATE DATABASE を使用する前に、%Admin\_Manage リソースを有するユーザとしてログインする必要があります。特権がない場合は、SQLCODE -99 エラー (特権違反) が返されます。

\$SYSTEM.Security.Login() メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

#### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“%SYSTEM.Security”を参照してください。

管理ポータルからネームスペースを作成することもできます。[\[システム管理\]](#)、[\[構成\]](#)、[\[システム構成\]](#)、[\[ネームスペース\]](#)の順に選択して、既存のネームスペースをリストします。既存ネームスペースのこのテーブルの一番上で、[\[新規ネームスペースの作成\]](#)をクリックできます。

1 つの InterSystems IRIS インスタンスに設定できるネームスペース数の最大数は 2048 です。

## 引数

### dbname

作成するデータベース (ネームスペース) の名前。

### pathname

データベースのルート・パス名の位置を示す引数 (オプション)。引用符付きの文字列として指定します。C および D ディレクトリは、このルート・パスのサブディレクトリとして作成されます。既定では、データベースは mgr ディレクトリ内に作成されます。

### WITH ENCRYPTED\_DB

データベースを暗号化するかどうかを指定する引数 (オプション)。既定では、暗号化されません。

### WITH GLOBAL\_JOURNAL\_STATE

データベースをジャーナルするかどうかを指定する引数 (オプション)。YES の場合は、データベースをジャーナルするよう指定します (推奨)。NO の場合は、データベースをジャーナルしないよう指定します。等号 (=) は任意で使用できます。既定では、ジャーナルされます。

## 関連項目

- [DROP DATABASE コマンド](#)
- [USE DATABASE コマンド](#)

## CREATE FOREIGN SERVER (SQL)

外部サーバを作成します。

### 構文

```
CREATE [ FOREIGN ] SERVER server-name [ TYPE server-type ]
    FOREIGN DATA WRAPPER CSV HOST host-name

CREATE [ FOREIGN ] SERVER server-name [ TYPE server-type ]
    FOREIGN DATA WRAPPER JDBC CONNECTION connection-name id-options
```

### 引数

引数	説明
server-name	作成する外部サーバ定義の名前。テーブル名と同様の追加の名前付け制約に従う、有効な識別子です。外部サーバ名は修飾名です。
TYPE server-type	外部サーバのタイプ。外部サーバには、'DB' または 'FILE' の 2 つのタイプのいずれかを指定できます。区切り文字は必須です。
FOREIGN DATA WRAPPER [ CSV   JDBC ]	外部データへのアクセスに使用するプロトコルを示します。外部データ・ラップには、CSV または JDBC の 2 つのオプションのいずれかを指定できます。
HOST host-name	データを格納するソース。この名前は、ファイル・システムにおけるフォルダです。host-name は、一重引用符で区切る必要があります。
CONNECTION connection-name	InterSystems IRIS を外部システムに接続する JDBC 接続の名前。区切る必要があります。JDBC 接続の確立の詳細は、“JDBC 経由での SQL ゲートウェイへの接続”を参照してください。
id-options	オプション - DELIMITEDIDS または NODELIMITEDIDS。外部データ・ソースが区切り識別子を受け入れるかどうかを指定します。

### 説明

CREATE FOREIGN SERVER コマンドは、InterSystems SQL が外部サーバと呼ばれる外部データ・ソースへのアクセスに使用できるリモートの場所を定義します。このコマンドは、外部データ・ソースから外部テーブル（ネイティブのテーブルと共に照会できる）にデータを投影するために使用できるメタデータを格納します。さらに、外部サーバが外部ソースのデータにアクセスするために使用するプロトコルを決定する、外部データ・ラップを定義します。

InterSystems SQL では現在、それぞれ .csv ファイルまたはデータベースから外部データを取得する、'FILE' と 'DB' という 2 つのタイプの外部サーバをサポートしています（オプションで TYPE キーワードを使用して指定）。タイプ 'FILE' の外部サーバはファイル・システム内のファイルにアクセスするのにに対し、タイプ 'DB' の外部サーバは事前定義された JDBC 接続を使用して外部データベースにアクセスします。外部サーバのタイプは、外部データ・ラップによって暗黙的に設定されます。

#### .csv ファイルの外部サーバの作成

.csv ファイルに格納されているデータを読み取って外部テーブルを作成する外部サーバを定義する場合は、CSV 外部データ・ラップを使用します。この方法で定義された外部サーバは、少なくとも、InterSystems SQL に投影できる .csv ファイルを格納するローカル・ファイル・パスを定義する必要があります。このファイル・パスは、HOST キーワードを使用して指定します。

以下の例では、.csv ファイルにアクセスする外部サーバを作成します。

```
CREATE FOREIGN SERVER Sample.DumpDir FOREIGN DATA WRAPPER CSV HOST '/data/dumps'
```

## JDBC 接続での外部サーバの作成

外部データベースに格納されているデータを読み取って外部テーブルを作成する外部サーバを定義する場合は、JDBC 外部データ・ラップアを使用します。この方法で定義された外部サーバは、InterSystems SQL のインスタンスと外部データ・ソースを接続する JDBC 接続を指定する必要があります。この接続の名前は、CONNECTION キーワードを使用して指定します。

以下の例では、JDBC 接続の外部サーバを作成します。

```
CREATE FOREIGN SERVER Sample.Postgres FOREIGN DATA WRAPPER JDBC CONNECTION 'PostgresSQLConnection'
```

## 区切り識別子の使用

外部データ・ソースに接続する際は、外部サーバが[区切り識別子](#)を受け入れるかどうかを指定することが必要になる場合があります。既定では、プロジェクションの作成時に InterSystems SQL は区切り識別子を外部データベース管理システムに送信できますが、すべてのデータベース管理システムで区切り識別子が許可されるわけではありません。区切り識別子を受け入れない外部データベース管理システムを使用している場合は、CREATE FOREIGN SERVER コマンドの末尾に NODELIMITEDIDS を指定する必要があります。既定の設定では、区切り識別子が許可されます。

## 関連項目

- ・ [DROP FOREIGN SERVER](#)
- ・ [ALTER FOREIGN SERVER](#)
- ・ [CREATE FOREIGN TABLE](#)



# CREATE FOREIGN TABLE (SQL)

外部テーブルを作成します。

## 構文

### ファイルから外部テーブルを作成

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
( column type, column2 type2, ... )
SERVER server-name FILE file-name
[ USING json-options ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
( column type, column2 type2, ... )
SERVER server-name FILE file-name
COLUMNS ( col-name, col-name2, ... )
[ USING json-options ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
( column type, column2 type2, ... )
SERVER server-name FILE file-name
COLUMNS ( col-name, col-name2, ... )
VALUES ( header, header2, ... )
[ USING json-options ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
( column type, column2 type2, ... )
SERVER server-name FILE file-name
VALUES ( header, header2, ... )
[ USING json-options ]
```

### データベースから外部テーブルを作成

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
[ ( column type, column2 type2, ... ) ]
SERVER server-name

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
[ ( column type, column2 type2, ... ) ]
SERVER server-name TABLE external-table
[ VALUES ( header, header2, ... ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
[ ( column type, column2 type2, ... ) ]
SERVER server-name QUERY query

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
[ ( column type, column2 type2, ... ) ]
SERVER server-name VALUES ( header, header2, ... )
```

## 概要

CREATE FOREIGN TABLE コマンドは、指定の構造に外部テーブル定義を作成します。CREATE FOREIGN TABLE は、InterSystems IRIS ネイティブのデータと共に、問い合わせることができる外部データ・ソースのデータの投影を作成します。

IF NOT EXISTS オプションを指定せずに、既存の外部テーブルと同じ名前で作成しようとすると、システムは SQLCODE -201 エラーを返します。このエラーは IF NOT EXISTS オプションを指定すれば抑制できますが、その場合、この外部テーブルは再作成されません。

名前が name であるスキーマが既に存在する場合に発生するエラーを抑制します。スキーマは再作成されません。

.csv ファイルから外部テーブルを作成する場合は、[LOAD DATA](#) の場合とまったく同じように、JSON オブジェクトまたは JSON オブジェクトを記述した文字列を USING 節に使用することで投影オプションを指定できます。



## ファイルから外部テーブルを作成

外部ファイルのデータを InterSystems IRIS のインスタンスに投影する外部テーブルを作成できます。このような場合、テーブルの作成先とする外部サーバでは、その外部データ・ラップとして CSV を使用している必要があります。指定したファイルが存在するかどうかによって、動作にわずかな違いがあります。

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] **table-name** ( **column type**, column2 type2, ...) SERVER **server-name** FILE **file-name** [ USING **json-options** ] は、指定した名前のファイルに格納されているデータを投影した外部テーブルを作成します。

- ファイルにヘッダがない場合、新しい外部テーブルの列には、ファイルの先頭 n 個の列にあるデータが置かれます。n は、プライマリリストの長さです。InterSystems SQL では、プライマリリストの列名を参照することで、この外部テーブルを問い合わせることができます。

```
CREATE FOREIGN TABLE (
    firstName VARCHAR(15),
    lastName VARCHAR(15),
    DOB DATE
) Sample.Person SERVER Sample.HospitalDir FILE 'person.csv'
```

- ファイルにヘッダがある場合、プライマリリストの各列名は、ファイルにある各列のヘッダ名に対応している必要があります。ファイルの中でプライマリリストの列名に対応している列名のみが、投影したテーブルに置かれます。

```
CREATE FOREIGN TABLE (
    firstName VARCHAR(15),
    lastName VARCHAR(15),
    DOB DATE
) Sample.Person SERVER Sample.HospitalDir FILE 'person.csv' USING { "from": { "file": { "header": true } } }
```

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] **table-name** ( **column type**, column2 type2, ...) SERVER **server-name** FILE **file-name** COLUMNS ( **col-name type**, col-name2 type2, ... ) [ USING **json-options** ] は、指定のファイルに格納されているデータを投影した外部テーブルを作成します。このファイルでは、COLUMNS 節で指定した順序で列が並んでいる必要があります。プライマリリストにある名前は、外部テーブルの列名を指定し、それぞれはファイルの各列と相互に位置的な関連性があります。COLUMNS 節に記述した名前は、プライマリリストにある名前と同一で、それぞれが節とリストの両方に存在する必要があります。COLUMNS 節を使用して、ファイルから取得する列を並べ替えることができます。COLUMNS 節に記述する列の順序が、プライマリリストでの列の順序と一致している必要はありません。外部テーブルでの列の順序は、COLUMNS 節に記述した列名の位置で決まります。

ファイルにヘッダがあっても無視されるので、このコマンドの動作は変わりません。その場合は、USING 節で **from.file.header** JSON オプションに **true** を指定する必要があります。

```
CREATE FOREIGN TABLE Sample.Person (
    FileColumnOne VARCHAR(10),
    FileColumnTwo VARCHAR(20)
) SERVER Sample.HospitalDir FILE person.csv COLUMNS (FileColumnTwo VARCHAR(20), FileColumnOne VARCHAR(10))
```

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] **table-name** ( **column type**, column2 type2, ...) SERVER **server-name** FILE **file-name** COLUMNS ( **col-name type**, col-name2 type2, ... ) VALUES ( **header**, header2, ... ) [ USING **json-options** ] は、指定のファイルに格納されたデータを投影した外部テーブルを作成します。このファイルでは、VALUES 節で指定した順序で列が並んでいる必要があります。外部テーブルでは、**.csv** ファイルの特定の列が欠落することがあります。プライマリリストでは、外部テーブルに置かれる列の名前とタイプを定義します。COLUMNS 節には、ファイルにある列の名前とタイプを記述します。ここに記述した列の数が、プライマリリストの列数より多くなってもかまいませんが、列名は同一にする必要があります。VALUES 節では、COLUMNS 節に記述した列名を並べ替えますが、列数はプライマリリストの列数と同じになります。

VALUES 節を使用して、COLUMNS 節でファイルから指定した列のうち、特定の列を外部テーブルから除外できます。VALUES 節に名前を記述した順序は、プライマリリストでの列名の順序にマッピングされます。InterSystems SQL では、プライマリリストの列名を参照することで、この外部テーブルを問い合わせることができます。

ファイルにヘッダがあっても無視されるので、このコマンドの動作は変わりません。その場合は、USING 節で `from.file.header` JSON オプションに `true` を指定する必要があります。

以下の例では、FieldOne 列が COLUMNS 節の 2 番目の要素を投影し、FieldTwo 列が COLUMNS 節の 1 番目の要素を投影します。また、FieldThree 列が COLUMNS 節の 4 番目の要素を投影します。

```
CREATE FOREIGN TABLE Sample.Person (
    FieldOne VARCHAR(10),
    FieldTwo VARCHAR(20),
    FieldThree INTEGER
) SERVER Sample.HospitalDB FILE person.csv COLUMNS (FirstName VARCHAR(10), LastName(20), DOB DATE,
Age INTEGER) VALUES (LastName, FirstName, Age)
```

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] `table-name` ( `column type`, `column2 type2`, ... ) SERVER `server-name` FILE `file-name` VALUES ( `header`, `header2`, ... ) [ USING `json-options` ] は、指定のファイルに格納されたデータの一部を投影する外部テーブルを作成します。VALUES 節に記述する列名は、.csv ファイルでの列名に対応している必要があります。この列名が、プライマリ列リストでの名前と異なってもかまいません。外部テーブルでの列の順序は、プライマリ列リストでの列の順序で決まります。これらの列には、VALUES 節でその列と位置的に関連する要素のデータが置かれます。

ファイルにヘッダがない場合、VALUES 節は無視され、意味がないものになります。

```
CREATE FOREIGN TABLE Sample.Person (
    FirstName VARCHAR(10),
    LastName VARCHAR(20)
) SERVER Sample.HospitalDB FILE person.csv VALUES (FirstNameInFile, LastNameInFile) USING { "from":
{ "file": { "header": 1 } } }
```

## データベースから外部テーブルを作成

外部データベースのデータを InterSystems IRIS のインスタンスに投影する外部テーブルを作成できます。このような場合、テーブルの作成先とする外部サーバでは、その外部データ・ラップとして JDBC を使用している必要があります。

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] `table-name` ( `column type`, `column2 type2`, ... ) SERVER `server-name` [ TABLE `external-table` ] は、指定のデータベースに存在するテーブルのデータを投影する外部テーブルを作成します。作成されたテーブルには、外部データベースにあるテーブルと同じ列が置かれます。TABLE 節を省略すると、InterSystems IRIS は、`external-table` ではなく `table-name` を使用して、外部サーバ上にあるテーブルにアクセスしようとします。

```
CREATE FOREIGN TABLE Sample.Person (
    FirstName VARCHAR(10),
    LastName VARCHAR(20)
) SERVER Sample.ExternalDB TABLE 'hospital.people'
```

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] `table-name` ( `column type`, `column2 type2`, ... ) SERVER `server-name` QUERY `query` は、外部データベースに存在するテーブルに対して実行したクエリ `query` で返されたデータを投影する外部テーブルを作成します。外部データベースに対して実行する前のクエリが InterSystems SQL で検証されることはありません。

```
CREATE FOREIGN TABLE Sample.Team (
    FirstName VARCHAR(10),
    LastName VARCHAR(20)
) SERVER Sample.ExternalDB QUERY 'SELECT FirstName,LastName FROM Hospital.Patients'
```

- CREATE FOREIGN TABLE [ IF NOT EXISTS ] `table-name` ( `column type`, `column2 type2`, ... ) SERVER `server-name` [ TABLE `external-table` ] VALUES ( `header`, `header2`, ... ) は、指定したテーブルに格納されているデータを投影した外部テーブルを、外部データ・ソースでの列名とは異なる列名で作成します。TABLE 節を省略すると、InterSystems IRIS は、`external-table` ではなく `table-name` を使用して、外部サーバ上にあるテーブルにアクセスしようとします。

VALUES 節のヘッダ名で外部データ・ソースの列名が特定されますが、これは列リストで指定した名前と異なっていることがあります。したがって、VALUES 節には、列リストの列と同じ数の列を指定する必要があります。

```
CREATE FOREIGN TABLE Sample.Team (
    TeamID BIGINT,
    Name VARCHAR(100)
) SERVER Sample.ExternalDB TABLE 'hospital.teams' VALUES (team_id, name)
```

## 引数

### table-name

CREATE FOREIGN TABLE コマンドでは、作成する外部テーブルの名前を、有効な識別子でこの引数に指定します。テーブル名は修飾、未修飾のどちらでもかまいません。

- 未修飾の外部テーブル名には、構文 `tablename` を使用します。スキーマ名とピリオド (.) 文字は省略します。未修飾のテーブルには既定のスキーマ名が使用されます。システム規模で既定の初期のスキーマ名は、既定のクラス・パッケージ名である `User` に対応する `SQLUser` です。スキーマ検索パスの値は無視されます。

JDBC 接続を使用して外部テーブルを作成していて、TABLE 節を省略している場合は、その未修飾テーブル名を外部データ・ソースに対して利用して投影が作成されます。ただし、このテーブルには、既定のスキーマで修飾した名前でも InterSystems SQL からアクセスできます。

JDBC 接続を使用して未修飾の外部テーブル名を指定し、TABLE 節を指定していない場合、

システム全体の既定のスキーマ名は構成可能です。

現在のシステム全体の既定のスキーマ名を確認するには、`$SYSTEM.SQL.Schema.Default()` メソッドを使用します。

- 修飾した外部テーブル名には構文 `schema.tablename` を使用します。既存のスキーマ名を指定することも、新規スキーマ名を指定することもできます。既存のスキーマ名を指定すると、そのスキーマに外部テーブルが配置されます。新規スキーマ名を指定すると、その名前のスキーマとそれに関連するクラス・パッケージが作成され、外部テーブルがそのスキーマに配置されます。

### column

CREATE FOREIGN TABLE コマンドでは、作成するテーブルの列を定義する 1 つの列名または列名のコンマ区切りのリストをプライマリ列リストに指定します。列名は任意の順序で指定でき、スペースを使用して、列名に関連付けたデータ型から分離します。規則によれば、通常は各列の定義を別々の行にインデントして記述します。この規則は必須ではありませんが、従うことをお勧めします。

プライマリ列リストは括弧で囲んで記述します。

### type

`column` で指定した列名の InterSystems SQL データ型クラス。データ型を指定すると、列で利用できるデータ値がそのデータ型に適した値に制限されます。InterSystems SQL は、大半の標準的な [SQL データ型](#) をサポートしています。

外部データ・ソースのデータには、プロジェクトの過程で指定の型が強制されます。日付形式が無効な場合など、フィールドに型を強制できない場合は実行時エラーが発生します。

### server-name

CREATE FOREIGN TABLE コマンドでは、外部データ・ソースにアクセスする外部サーバ構成を、この引数で指定します。

修飾サーバ名または未修飾サーバ名のどちらで指定してもかまいません。未修飾の外部サーバ名を指定すると、既定のスキーマである `SQLUser` の中でその外部サーバが検索されます。修飾した外部サーバ名を指定すると、指定したスキーマの中でその外部サーバが検索されます。

指定されたスキーマに目的の外部サーバが見つからないと `SQLCODE -360` エラーが発生します。

## file-name

CREATE FOREIGN TABLE コマンドでは、InterSystems IRIS に投影するデータを収めた **.csv** ファイルの場所をこの引数で指定します。この場所は、完全なファイル・パスを引用符で囲んで定義します。このコマンドで指定した外部サーバで、その外部データ・ラッパに CSV オプションが使用されている場合にのみ、この引数を使用します。

- ・ 外部テーブルに投影する行を、このファイルの行ごとに 1 行ずつ指定します。既定の行区切り文字は新規行文字 (“**\n**”) です。空白行は無視されます。
- ・ 行内の各データ値を列区切り文字で区切ります。既定の列区切り文字はコンマです。プレースホルダ列区切り文字で示される未指定のデータを含め、すべてのデータ・フィールドを列区切り文字を使用して記述する必要があります。別の列区切り文字を定義するには、USING json-options 節で columnseparator オプションを指定します。
- ・ 既定では、エスケープ文字は定義されていません。データ値にリテラルとして列区切り文字を記述するには、そのデータ値を引用符で囲みます。引用符で囲んだデータ値に引用符を記述するには引用符文字を二重にします。エスケープ文字を定義するには、USING json-options 節で escapechar オプションを指定します。
- ・ 既定では、外部テーブルに記述されたフィールドの順序でデータ値を指定します。COLUMNS 節と VALUES 節を使用すると、別の順序でデータを指定できます。
- ・ **.csv** ファイルのすべてのデータは、テーブルのデータ条件と一致するか検証されます。この条件として、レコードにあるデータ・フィールドの数や、各フィールドのデータ型とデータ長などがあります。検証できないレコードがファイルにあるとエラー・メッセージが表示されます。**.csv** ファイルにある日時/時刻文は ODBC 形式とする必要があります。他の形式にすると、エラーが発生することや正しくないクエリ結果が得られることがあります。

## col-name

CREATE FOREIGN TABLE コマンドでは、この引数を COLUMNS 節に使用します。ファイルにヘッダがない場合は、COLUMNS 節でファイルの各列名を指定します。ファイルにヘッダがあれば、多くの場合、COLUMNS 節を無視してかまいません。

## header

CREATE FOREIGN TABLE コマンドでは、この引数を VALUES 節に使用します。さまざまなシナリオで VALUES 節を使用できます。

## external-table

JDBC 接続で外部データ・ソースに接続する CREATE FOREIGN TABLE コマンドでは、InterSystems IRIS に投影する外部テーブルの名前をこの引数で指定します。列リストを省略すると、この方法で作成する外部テーブルにはデータ・ソースの列定義がコピーされます。この定義として、列名やサポートされているデータ型などがあります。

## query

JDBC 接続で外部データ・ソースに接続する CREATE FOREIGN TABLE コマンドでは、外部テーブルの列定義と列データを取得するために外部データ・ソースのテーブルに対して実行するクエリを、この引数で指定します。これは、外部データ・ソースに対して実行する SELECT クエリです。

この方法で作成する外部テーブルには、外部データ・ソースから列定義がコピーされます。この定義として、列名やサポートされているデータ型などがあります。外部データ・ソースの複数のテーブルをこのクエリで結合して指定すると、それら複数のテーブルの列定義を外部テーブルにコピーできます。

## json-options

JSON オブジェクトとして読み込むためのオプションや JSON オブジェクトを記述した文字列を、USING 節にこの引数で指定します。この使用法は、LOAD DATA コマンドでこの引数に相当する引数の使用法と同じです。その構文とオプションの詳しい概要は、[LOAD DATA のドキュメント](#)を参照してください。CREATE FOREIGN TABLE コマンドでは from.file ツリーでのオプションのみをサポートしている点に注意が必要です。

## 関連項目

- ・ [CREATE FOREIGN SERVER](#)
- ・ [ALTER FOREIGN TABLE](#)
- ・ [DROP FOREIGN TABLE](#)
- ・ [DROP FOREIGN SERVER](#)
- ・ [LOAD DATA](#)

# CREATE FUNCTION (SQL)

クラス内でメソッドとして関数を生成します。

## 構文

```
CREATE FUNCTION name(parameter_list) [characteristics]
[ LANGUAGE SQL ]
BEGIN code_body ;
END

CREATE FUNCTION name(parameter_list) [characteristics]
LANGUAGE OBJECTSCRIPT
{ code_body }

CREATE FUNCTION name(parameter_list) [characteristics]
LANGUAGE { JAVA | PYTHON | DOTNET }
EXTERNAL NAME external-stored-procedure
```

## 説明

CREATE FUNCTION 文は、クラスのメソッドとして関数を生成します。このクラス・メソッドは、SQL ストアド・プロシージャとして投影されます。[CREATE PROCEDURE](#) 文は、SQL ストアド・プロシージャとして投影されるメソッドの生成にも使用できます。メソッドで値を返すときには CREATE FUNCTION を使用する必要がありますが、値を返さないメソッドの生成にも使用できます。

オプションのキーワード OR REPLACE を使用すると、既存の関数を変更または置換できます。CREATE OR REPLACE FUNCTION には、DROP FUNCTION を呼び出して古いバージョンの関数を削除し、続いて CREATE TRIGGER を呼び出す操作と同じ効果があります。

関数の生成には、[GRANT](#) コマンドで指定された %CREATE\_FUNCTION 管理者特権が必要です。

クラス定義が[導入済みのクラス](#)の場合、クラスで関数を作成することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ `classname` DDL ” に設定されます。

SQL 文内からの SQL 関数の呼び出しの詳細は、“InterSystems SQL の使用法” の “データベースの問い合わせ” の章にある “[ユーザ定義関数](#)” を参照してください。さまざまな状況での SQL ストアド・プロシージャの呼び出しについては、“[CALL](#)” 文を参照してください。

## 引数

### name

ストアド・プロシージャ・クラスに作成する関数の名前。パラメータを指定しない場合でも、name は有効な識別子である必要があり、後には括弧を付加する必要があります。この名前は、未修飾 (StoreName) にして[既定のスキーマ名](#)を使用しても、スキーマ名を指定して修飾 (Patient.StoreName) してもかまいません。`$SYSTEM.SQL.Schema.Default()` メソッドを使用して、現在のシステム全体の既定のスキーマ名を確認できます。初期のシステム全体の既定のスキーマ名は、クラス・パッケージ名 **User** に対応する `SQLUser` です。

FOR 特性 (後述) は、name で指定されたクラス名をオーバーライドすることに注意してください。この名前の関数が既に存在する場合、この操作は失敗して SQLCODE -361 エラーが発行されます。

生成されるクラスの名前は、スキーマ名に対応するパッケージ名の後にドット、“func”、指定の name が順に続いたものになります。例えば、未修飾の関数名 `RandomLetter` で初期の既定のスキーマ `SQLUser` が使用される場合、生成されるクラス名は `User.funcRandomLetter` となります。詳細は、“InterSystems SQL の使用法” の “ストアド・プロシージャの定義と使用” の章の “[SQL からクラス名への変換](#)” を参照してください。

InterSystems SQL では、大文字/小文字の区別が異なるだけの重複関数名を指定することは許可されていません。既存の関数名と大文字/小文字区別が異なるだけの関数名を指定した場合には、SQLCODE -400 エラーが生成されます。



## parameter-list

値を関数に渡すために使用するパラメータのリストです (オプション)。パラメータのリストは括弧で囲み (これは、パラメータが指定されていない場合でも必須です)、リストのパラメータ宣言はコンマで区切ります。リスト内の各パラメータ宣言は、(先頭から順番に) 以下の要素で構成します。

- ・ パラメータ・モードが IN (入力値)、OUT (出力値)、または INOUT (変更値) のいずれであるかを指定するオプションのキーワード。省略した場合、既定のパラメータ・モードは IN です。
- ・ パラメータ名。パラメータ名では、大文字と小文字が区別されます。
- ・ パラメータの [データ型](#)。
- ・ オプション：パラメータの既定値。DEFAULT キーワードの後ろに既定値を付けて指定できます。DEFAULT キーワードはオプションです。既定値が指定されていない場合、既定値は NULL であると見なされます。

以下の例では、2 つの入力パラメータを指定しており、どちらにも既定値があります。オプションの DEFAULT キーワードは、1 つ目のパラメータでは指定されていますが、2 つ目のパラメータでは省略されています。

## SQL

```
CREATE FUNCTION RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END
```

ユーザ定義関数は、[ユーザ定義の集約関数](#)の節に指定します。ユーザ定義の集約関数で使用する関数を定義する場合は、出力値を集約して渡すために使用する状態パラメータを定義します。

関数は、データの行の値に依存するパラメータ (%ID フィールドなど) を 1 つ以上取る場合、相関関数になります。相関関数は行単位で評価され、相関関数ではない関数 (つまり、パラメータを取らないか、すべての行にわたって一貫性を維持する引数を取る関数) は 1 回だけ評価されます。

## characteristics

関数の特性を指定する 1 つ以上のキーワードを含む引数 (オプション)。複数の特性は空白 (スペースまたは改行) で区切ります。特性は任意の順序で指定できます。利用できるキーワードは以下のとおりです。

FOR className	関数を生成するクラス名を指定します。そのクラスが存在しない場合は新規作成します。関数名を認証することによりクラス名を指定することもできます。FOR 節で指定されたクラス名の方が、関数名の認証により指定されたクラス名よりも優先されます。
FINAL	サブクラスによって関数がオーバーライドされないように指定します。既定では、関数は FINAL ではありません。FINAL キーワードはサブクラスによって継承されます。
PRIVATE	関数がそれ自身のクラス、またはサブクラスの他の関数によってのみ起動できることを指定します。既定では関数はパブリックで、制限なしに起動できます。この制限はサブクラスによって継承されます。
PROCEDURE	関数が SQL ストアド・プロシージャとして投影されることを指定します。ストアド・プロシージャはサブクラスに継承されます。CREATE FUNCTION は常に SQL ストアド・プロシージャに投影するため、このキーワードはオプションです。このキーワードは、PROC と略することができます。
RETURNS datatype	関数の呼び出しで返される値のデータ型を指定します。RETURNS が省略されると、関数は値を返すことができません。この指定内容はサブクラスによって継承され、サブクラスによって変更できます。この datatype には、MINVAL、MAXVAL、SCALE などのタイプのパラメータを指定できます。例えば RETURNS DECIMAL (19,4) のように指定します。値が返されるとき、datatype の長さは無視されます。例えば、RETURNS VARCHAR (32) は、この関数の呼び出しによって返されるあらゆる長さの文字列を受け取ることができます。
SELECTMODE mode	LANGUAGE が SQL (既定値) の場合にのみ使用されます。これを指定した場合、InterSystems IRIS は、 <code>#SQLCOMPILE SELECT=mode</code> 文を対応するクラス・メソッドに追加することで、指定された SELECTMODE で、メソッドで定義された SQL 文を生成します。mode に指定できる値は、LOGICAL、ODBC、RUNTIME、および DISPLAY です。既定は LOGICAL です。

SELECTMODE 節は、SELECT クエリ操作、および INSERT と UPDATE 操作で使います。これは、コンパイル時の選択モードを指定します。SELECTMODE に指定した値は、`#sqlcompile select=mode` のように InterSystems ObjectScript のクラス・メソッド・コードの先頭に追加されます。詳細は、“ObjectScript の使用法” の “ObjectScript マクロとマクロ・プリプロセッサ” の章にある “[#sqlcompile select](#)” を参照してください。

- SELECT クエリでは、SELECTMODE はデータを返すモードを指定します。mode 値が LOGICAL の場合は、論理 (内部保存) 値が返されます。例えば、日付は \$HOROLOGY 形式で返されます。mode 値が ODBC の場合、論理と ODBC 間の変換が適用され、ODBC 形式値が返されます。mode 値が DISPLAY の場合、論理と表示間の変換が適用され、表示形式値が返されます。mode 値が RUNTIME の場合、実行時に表示モードを (LOGICAL、ODBC、または DISPLAY に) 設定できます。
- [INSERT](#) または [UPDATE](#) 操作では、SELECTMODE RUNTIME オプションで、表示形式 (DISPLAY または ODBC) から論理格納形式への入力データ値の自動変換がサポートされています。このコンパイルされた表示データから論理データへの変換コードは、SQL コード実行時の選択モード設定が LOGICAL (すべての InterSystems SQL 実行インタフェースの既定値) の場合のみ適用されます。

“InterSystems SQL の使用法” の “[ダイナミック SQL の使用法](#)” の章に説明されているように、SQL コードが実行されると、`%SQL.Statement` クラスの `%SelectMode` プロパティが、実行時の選択モードを指定します。SelectMode オプションの詳細は、“InterSystems SQL の使用法” の “InterSystems IRIS SQL の基礎” の章にある “[データ表示オプション](#)” を参照してください。

## LANGUAGE

プロシージャ・コード言語を指定するキーワード節です (オプション)。利用可能なオプションは以下のとおりです。



- ・ LANGUAGE OBJECTSCRIPT (ObjectScript の場合) または LANGUAGE SQL。プロシージャ・コードは `code_body` で指定します。
- ・ LANGUAGE JAVA、LANGUAGE PYTHON、または LANGUAGE DOTNET (これらのいずれかの言語で外部ストアド・プロシージャを呼び出す SQL プロシージャの場合)。外部ストアド・プロシージャの構文は、以下のようになります。

```
LANGUAGE langname EXTERNAL NAME external-routine-name
```

`langname` は JAVA、PYTHON、または DOTNET で、`external-routine-name` は指定した言語の外部ルーチン名を含む引用符付き文字列です。SQL プロシージャは既存のルーチンと呼び出します。これらの言語のコードを CREATE FUNCTION 文内に記述することはできません。これらの言語のストアド・プロシージャ・ライブラリは IRIS の外部に保存されているため、IRIS 内でパッケージ、インポート、コンパイルする必要はありません。以下に、値を返す既存の JAVA 外部ストアド・プロシージャを呼び出す CREATE FUNCTION の例を示します。

```
CREATE FUNCTION getPrice (item_name VARCHAR)
RETURNS INTEGER
LANGUAGE JAVA
EXTERNAL NAME 'Orders.getPrice'
```

LANGUAGE 節が省略される場合は、SQL が既定です。

### code\_body

生成されるメソッドのプログラム・コード。このコードは SQL または ObjectScript で指定します。SQL プログラム・コードの開始には BEGIN キーワードを使用し、終了には END キーワードを使用します。`code_body` 内の完結した各 SQL 文は、セミコロン (;) で終わります。ObjectScript プログラム・コードは中括弧で囲みます。コード行は、インデントする必要があります。使用する言語は LANGUAGE 節と一致させる必要があります。ただし、ObjectScript のコードには埋め込み SQL を記述できます。

InterSystems IRIS は、メソッドの実際のコードを生成するために提供されたコードを使用します。指定するコードが SQL の場合、メソッドの生成時にコード行が追加されます。追加のコードでは、SQL が ObjectScript “ラッパ” に埋め込まれ、(必要に応じて) プロシージャ・コンテキスト・ハンドラが作成され、返り値が処理されます。以下はこの InterSystems IRIS から生成されたラッパ・コードの例です。

### ObjectScript

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql( SELECT col FROM tbl )
QUIT $GET(title)
```

指定するコードが OBJECTSCRIPT である場合、ObjectScript コードは中括弧で囲む必要があります。すべてのコード行を列 1 からインデントする必要がありますが、ラベルとマクロ・プリプロセッサ指示文はその必要はありません。ラベルやマクロ指示文の前には、列 1 でコロン (:) を付加する必要があります。

ObjectScript コードの場合、“ラッパ” (変数を NEW で処理して、QUIT を使用して終了し、(オプションで) 完了時に値を返す) を明示的に定義する必要があります。

ストアド・プロシージャが呼び出されると、クラス `%Library.SQLProcContext` のオブジェクトが `%sqlcontext` 変数でインスタンス化されます。このプロシージャ・コンテキスト・ハンドラによって、プロシージャとその呼び出し元 (ODBC サーバなど) 間でプロシージャ・コンテキストの受け渡しが行われます。

`%sqlcontext` は、エラー・オブジェクト、SQLCODE エラー・ステータス、SQL 行カウント、およびエラー・メッセージを含む、いくつかのプロパティで構成されます。以下の例は、これらプロパティの設定に使用される値を示しています。

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=msg
```

SQLCODE と %ROWCOUNT の値は、SQL 文の実行によって自動的に設定されます。`%sqlcontext` オブジェクトは、実行される前に毎回リセットされます。

または、%SYSTEM.Error オブジェクトをインスタンス化し、%sqlcontext.Error として設定することによって、エラー・コンテキストを設定することができます。

指定した関数が既に存在する場合は、SQLCODE -361 エラーが生成されます。このエラーを回避するには、オプションの OR REPLACE キーワードを使用するか、最初に DROP FUNCTION で古い関数を削除します。

## ユーザ定義関数の実行

以下のように SELECT 文内で関数を実行できます。

### SQL

```
SELECT StudentName,StudentAge,SQLUser.HalfAge() AS HalfTheAge
FROM SQLUser.MyStudents
```

この関数が存在しない場合は、SQLCODE -359 エラーが生成されます。

この関数の実行結果がエラーになる場合は、SQLCODE -149 エラーが生成されます。エラーのタイプは %msg で示されます。

## 例

以下の例では、ランダムな大文字を生成するプロシージャとして格納される RandomLetter() 関数 (メソッド) を作成します。その後、この関数を SELECT 文内で呼び出すことができます。RandomLetter() 関数を削除するため、DROP FUNCTION が指定されています。以下の例は相関関数ではない関数の例です。したがって、SELECT 文の結果セットには、ランダムに選択された同じ文字で始まる名前と、その文字で始まる名前の数が記述されています。結果セットの例を示します。

### SQL

```
CREATE FUNCTION RandomLetter()
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
SET x=$RANDOM(90)
IF x<65 {GOTO Top}
ELSE {QUIT $CHAR(x)}
}
```

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter()
```

```
Abbott, Amelia P.
Adams, John J.
Alton, Lionel N.
Amblin, Stephen O.
Amory, Jennifer E.
Andrews, Olivia G.
Arias, Rowan K.
Avery, Marvin N.
```

```
DROP FUNCTION RandomLetter
```

以下の例では、大文字をランダムに生成するプロシージャとして格納される RandomLetter() 関数 (メソッド) を、%ID の変化する値に依存する相関関数として作成します。引数自体は RandomLetter() の本文に使用されません。SELECT

文の結果セットには、ランダムに選択された異なる文字で始まる名前が含まれ、その長さには可変個数の要素が含まれます。結果セットの例を示します。

```
CREATE FUNCTION RandomLetter(IN id INTEGER)
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
  SET x=$RANDOM(90)
  IF x<65 {GOTO Top}
  ELSE {QUIT $CHAR(x)}
}
```

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter(%ID)
```

```
Alton,Lionel N.
Cooper,Peter H.
Hertz,Lana C.
Jones,Alyssa D.
```

以下の例は、ObjectScript コードを呼び出す関数を作成します。ObjectScript コードは埋め込み SQL を含んでいます。

### ObjectScript

```
&sql(CREATE FUNCTION TraineeName(
  SSN VARCHAR(11),
  OUT Name VARCHAR(50) )
PROCEDURE
RETURNS VARCHAR(30)
FOR SQLUser.MyStudents
LANGUAGE OBJECTSCRIPT
{
  NEW SQLCODE,%ROWCOUNT
  SET Name=""
  &sql(SELECT Name INTO :Name FROM Sample.Employee
    WHERE SSN = :SSN)
  IF $GET(%sqlcontext)!='' {
    SET %sqlcontext.%SQLCODE=SQLCODE
    SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
  QUIT Name
})
IF SQLCODE=0 { WRITE !,"Created a function" QUIT}
ELSE { WRITE !,"CREATE FUNCTION error: ",SQLCODE," ",%msg,!
  &sql(DROP FUNCTION TraineeName FROM SQLUser.MyStudents) }
IF SQLCODE=0 { WRITE !,"Dropped a function" QUIT}
ELSE { WRITE !,"Drop error: ",SQLCODE }
```

`%sqlcontext` オブジェクトを使用します。対応する SQL 変数を使用して、その `%SQLCODE` プロパティおよび `%ROWCOUNT` プロパティを設定します。関数の `LANGUAGE OBJECTSCRIPT` キーワードに続く ObjectScript コードを中括弧で囲むことに注意してください。ObjectScript コードには、`&sql` でマークされ、角括弧で囲まれた埋め込み SQL コードがあります。

## セキュリティおよび特権

`CREATE FUNCTION` コマンドは、ユーザに `%Development:USE` 権限が必要な特権操作です。このような権限は管理ポータルを介して付与できます。これらの特権なしで `CREATE FUNCTION` コマンドを実行すると、`SQLCODE -99` エラーが発生し、コマンドは失敗します。

適切な特権を持たないユーザでも、以下の 2 つの状況のいずれかではこのコマンドを実行できます。

- ・ 埋め込み SQL を介してコマンドを実行する場合。この場合は特権が確認されません。
- ・ 特権を確認しないことをユーザが明示的に指定する場合。例えば、`checkPriv` 引数を 0 に設定して `%Prepare()` を呼び出すか、`%SQL.Statement` に対して `%ExecDirectNoPriv()` を呼び出します。

## 関連項目

- ・ [DROP FUNCTION コマンド](#)
- ・ [CREATE AGGREGATE コマンド](#)
- ・ “InterSystems SQL の使用法” の “[ストアド・プロシージャの定義と使用](#)” の章

# CREATE INDEX (SQL)

テーブルにインデックスを定義します。

## 構文

```
CREATE index-type INDEX index-name
  ON [TABLE] table-name (field-name, ...)
  [AS index-class-name [ (parameter-name = parameter_value, ... ) ] ]
  [WITH DATA (datafield-name, ...)]
  [ [ IMMEDIATE | DEFER ] [BUILD] ]
```

## 引数

### index-type

作成するインデックスのタイプを指定する引数 (オプション)。インデックス・タイプのオプションは以下のとおりです。

- **UNIQUE** : インデックス内のすべてのフィールドで、テーブル内に同じ値を持つ行が存在しないようにする制約。このキーワードはビットマップ・インデックスまたはビットスライス・インデックスに指定することはできません。  
UNIQUE キーワードの後に (またはそれに置き換えて) CLUSTERED または NONCLUSTERED キーワードを指定することができます。これらのキーワードは空命令であり、他のベンダとの互換性を保持するために指定します。
- **BITMAP** : **ビットマップ・インデックス**の生成の指定。ビットマップ・インデックスは、個別値が少ないフィールドで高速なクエリを可能にします。
- **BITMAPEXTENT** : **ビットマップ・エクステント・インデックス**の生成の指定。テーブルに対して作成できるビットマップ・エクステント・インデックスは、最大で 1 つです。field-name は BITMAPEXTENT と共には指定されません。
- **BITSLICE** : **ビットスライス・インデックス**の生成の指定。ビットスライス・インデックスは、合計や値域条件など、特定の式について非常に高速な評価を可能にします。これは特殊なインデックス・タイプであり、非常に特定の問題を解決する場合に限り使用してください。
- **COLUMNAR** : **列指向インデックス**の生成の指定。列指向インデックスを使用すると、基盤となるデータが複数の行にわたって格納されている列に対してきわめて高速なクエリを実行できます。特にフィルタリングと集約の処理を伴うクエリで効果的です。列指向インデックスは、2022.2 の試験的機能です。

### index-name

定義するインデックス。識別子の名前です。

### table-name

インデックスを定義する既存の**テーブル**名。ビューのインデックスは作成できません。table-name は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、**既定のスキーマ名**が使用されます。

### field-name

インデックスの基準となる、1 つまたは複数の**フィールド**の名前。フィールド名は括弧で囲む必要があります。複数のフィールド名はコンマで区切ります。

各フィールド名の後に ASC または DESC キーワードを指定することができます。これらのキーワードは空命令であり、他のベンダとの互換性を保持するために指定します。

### AS index-class-name

インデックスを定義するクラスを指定する引数 (オプション)。オプションで、パラメータ名と関連する値の 1 つまたは複数のコンマ区切りペアを括弧で囲んで後に付けることができます。

## WITH DATA (datafield-name)

インデックスの [Data](#) のプロパティとして定義されている 1 つまたは複数のフィールドの名前を指定する引数 (オプション)。フィールド名は括弧で囲む必要があります。複数のフィールド名はコンマで区切ります。BITMAP または BITSlice インデックスを指定する場合、WITH DATA 節は指定できません。

## IMMEDIATE BUILD

作成したインデックスを直ちに構築することを指定する引数 (オプション)。インデックスは既定で直ちに構築されるので、この節は省略できます。BUILD キーワードはオプションです。

## DEFER BUILD

作成時にインデックスの構築を無効にすることを指定する引数 (オプション)。このオプションによって、インデックスが選択不可とされるので、そのインデックスをクエリで使用できなくなります。インデックスを後で使用するには、[BUILD INDEX](#) を使用して構築した後、SetMapSelectability() メソッドを使用して選択可能にする必要があります。マップが選択可能かどうかを管理ポータルで表示できます。管理ポータルで [\[システムエクスプローラ\]](#) > [\[SQL\]](#) > [\[カタログの詳細\]](#) に移動し、[\[マップ\]](#)/[\[インデックス\]](#) ボタンを選択します。BUILD キーワードはオプションです。

詳細は、以下の追加の互換性構文を参照してください。

## 概要

CREATE INDEX 節は、指定したテーブルの指定したフィールドでソートされたインデックスを作成します。InterSystems IRIS がインデックスを使用することによって、クエリを操作するパフォーマンスが向上します。InterSystems IRIS は、INSERT、UPDATE、および DELETE を操作する際、インデックスのメンテナンスを自動的に行います。そのため、これらのデータを変更する操作の場合は、インデックスのメンテナンスによってパフォーマンスが逆に低下する場合があります。

インデックスを作成するには、“[インデックスの定義と構築](#)” の説明に従い、CREATE INDEX コマンドを使用するか、クラス定義にインデックス定義を追加します。インデックスを削除するには、[DROP INDEX](#) コマンドを使用します。

インデックスを作成できるプロパティと作成できないプロパティについての詳細は、“[インデックスを付けることができるプロパティ](#)” を参照してください。

CREATE INDEX を使用すると、以下のいずれかのタイプのインデックスを作成できます。

- ・ 通常のインデックス (Type=index) : CREATE INDEX (一意でない値の場合) または CREATE UNIQUE INDEX (一意の値の場合) を指定します。
- ・ ビットマップ・インデックス (Type=bitmap) : CREATE BITMAP INDEX を指定します。
- ・ ビットスライス・インデックス (Type=bitslice) : CREATE BITSlice INDEX を指定します。
- ・ 列指向インデックス (Type=columnar) : CREATE COLUMNAR INDEX を指定します。

%Dictionary.IndexDefinition クラスを使用してもインデックスを定義できます。

CREATE INDEX を使用して、インデックスを[シャード・テーブル](#)に追加できます。

クラス・レベルでのインデックスに関する詳細は、%Library.FunctionalIndex の説明を参照してください。

## 特権とロック

CREATE INDEX コマンドは特権を必要とする操作です。CREATE INDEX を実行するには、ユーザは %ALTER\_TABLE 管理特権を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg 'name' %ALTER\_TABLE が表示されます。適切な付与特権を持っていれば、[GRANT](#) コマンドを使用して、ユーザまたはロールに %ALTER\_TABLE 特権を割り当てることができます。管理特権はネームスペース固有のものです。詳細は、“[特権](#)” を参照してください。

ユーザは、指定されたテーブルに対する %ALTER 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対する %ALTER 特権が自動的に付与されます。そうでない場合は、ユー



ずにテーブルに対する %ALTER 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg 'name' 'Schema.TableName' %ALTER が表示されます。[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが %ALTER 特権を持っているかどうかを確認できます。[GRANT](#) コマンドを使用して、指定したテーブルに %ALTER 特権を割り当てることができます。詳細は、“InterSystems SQL の使用法” の “[特権](#)” を参照してください。

- CREATE INDEX は、テーブル・クラスの定義に [DdlAllowed] が含まれている場合を除き、[永続クラスから投影されたテーブル](#)では使用できません。使用すると、操作は SQLCODE -300 エラーで失敗し、%msg が “DDL schema.tablename” に設定されます。
- CREATE INDEX は、[導入済みの永続クラス](#)から投影されたテーブルでは使用できません。この操作は SQLCODE -400 エラーで失敗し、%msg が “classname DDL” に設定されます。

CREATE INDEX 文は table-name に対してテーブル・レベルのロックを取得します。これにより、他のプロセスはこのテーブルのデータを変更できなくなります。このロックは、CREATE INDEX 操作が終了すると自動的に解除されます。CREATE INDEX では、インデックス・データの入力を含め、インデックス作成操作が完了するまで、対応するクラス定義はロックされたままになります。

インデックスを作成する場合、そのテーブルを別のプロセスによって EXCLUSIVE MODE または SHARE MODE でロックしないでください。ロックされているテーブルで CREATE INDEX 操作を実行しようとすると、SQLCODE -110 エラーになり、%msg は “'Sample.MyTest'” になります。

## 互換性のみにサポートされるオプション

InterSystems SQL は、構文解析の目的にのみ以下の CREATE INDEX オプションをサポートし、既存の SQL コードから InterSystems SQL への変換を支援します。これらのオプションは、実際には機能しません。

CLUSTERED | NONCLUSTERED owner.catalog. ASC | DESC

以下は、これらの空命令キーワードの配置を示す例です。

```
CREATE UNIQUE CLUSTERED INDEX index-name ON TABLE owner.catalog.schema.table (field1 ASC, field2 DESC)
```

## インデックス名

インデックス名は、指定されたテーブル内では一意の必要があります。インデックス名は、[識別子規約](#)に従い、以下の制限を受けます。既定のインデックス名は、簡単な識別子です。インデックス名は区切り文字付き識別子とすることができます。インデックス名は、128 文字を超えることはできません。インデックス名では、大文字と小文字が区別されません。

InterSystems IRIS は、“SqlName” と呼ばれる) 入力された名前を使用して、クラスおよびグローバル内で対応するインデックス・プロパティ名を生成します。このインデックス・プロパティ名には英数字 (文字と数字) のみを使用でき、最大長は 96 文字です。InterSystems IRIS は、インデックス・プロパティ名を生成するために、最初に指定した SqlName から句読点を削除し、次に 96 文字以内の一意の識別子を生成します。

- インデックス名は、フィールド、テーブル、ビューと同じ名前にすることができますが、このような名前の重複はお勧めできません。
- インデックス・プロパティ名 (句読点の削除後) は一意である必要があります。重複する SQL インデックス名を指定すると、SQLCODE -324 エラーが生成されます。既存の SQL インデックス名とは句読点のみ異なる SQL インデックス名を指定すると、一意のインデックス・プロパティ名を作成するために、最後の文字が大文字 (“A” で始まる) に置き換えられます。したがって、句読点のみ異なる SQL インデックス名を作成することは可能です (ただし推奨はしません)。
- インデックス・プロパティ名は文字で始まる必要があります。したがって、インデックス名の先頭の字または最初の句読点の後ろの字は、文字にする必要があります。有効な文字は、[\\$ZNAME](#) テストに合格する文字です。SQL インデックス名の最初の文字が句読点 (%) または () で、2 番目の文字が数字である場合、InterSystems IRIS は、削除されたインデックス・プロパティ名の最初の文字として小文字の “n” を追加します。

- ・ インデックス名は 31 文字よりも大幅に長くすることができますが、最初の 31 文字の英数字と異なるインデックス名にすると作業が簡単になります。

管理ポータル の SQL インタフェース [\[カタログの詳細\]](#) には、各インデックスの SQL インデックス名 ([SQL マップ名]) および対応するインデックス・プロパティ名 ([インデックス名]) が表示されます。

既存インデックスと同じ名前のインデックスを作成する場合に何が起きるかは、以下に説明しています。

## 既存インデックス

既定で InterSystems IRIS は、テーブルに既存インデックスと同じ名前のインデックスを作成することを拒否し、SQLCODE -324 エラーを発行します。現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。これにより、`[ DDL CREATE INDEX ]` 設定が表示されます。既定値は 0 で、この設定を推奨します。このオプションを 1 に設定した場合、InterSystems IRIS はクラス定義から既存インデックスを削除して、CREATE INDEX を実行して再作成します。CREATE INDEX で指定されたテーブルから、指定されたインデックスを削除します。このオプションにより、UNIQUE 制約インデックスの削除/再作成が可能になります (DROP INDEX コマンドでは実行できません)。主キー・インデックスの削除/再作成は、["ALTER TABLE"](#) コマンドを参照してください。

管理ポータル、[\[システム管理\]](#)、[\[構成\]](#)、[\[SQL とオブジェクトの設定\]](#)、[\[SQL\]](#) から [\[冗長な DDL ステートメントを無視\]](#) チェック・ボックスにチェックを付けることにより、このオプション (および他の同様の作成、変更、および削除のオプション) をシステム全体で設定できます。

ただし、このオプションが既存インデックスの再作成を許可するように設定されていても、テーブルにデータが格納されている場合は、[主キー IDKEY](#) インデックスの再作成はできません。これを実行しようとすると、SQLCODE -324 エラーが生成されます。

## テーブル名

既存のテーブルの名前を指定する必要があります。

- ・ `table-name` が存在しない場合、CREATE INDEX は SQLCODE -30 エラーで失敗し、`%msg` が `"SQLUSER.MYTABLE"` に設定されます。
- ・ `table-name` がビューの場合、CREATE INDEX は SQLCODE -30 エラーで失敗し、`%msg` が `[ SQLUSER.MYVIEW CREATE INDEX 'My_Index' ]` に設定されます。

インデックスを作成するとテーブルの定義が変更されます。ユーザにテーブル定義を変更する特権がない場合、CREATE INDEX は SQLCODE -300 エラーで失敗し、`%msg` が `"DDL schema.tablename"` に設定されます。

## フィールド名

インデックスを作成するフィールド名は 1 つ以上指定する必要があります。フィールド名は 1 つだけ指定するか、コンマで区切った複数のフィールド名のリストを括弧で囲んで指定します。重複するフィールド名を使用でき、インデックス定義に保持できます。複数のフィールドを指定した場合、Group By State の後で各州内の Group By City を実行するような、GROUP BY 操作のパフォーマンスが向上する場合があります。通常は、重複するデータが大量にあるフィールド (または複数のフィールド) にインデックスを作成しないでください。例えば、人のデータベースの Name フィールドでインデックスを作成するのは、ほとんどの人名が一意であるため、問題ありません。しかし、State フィールドでインデックスを作成するのは、ほとんどの場合、重複するデータが大量に出現するため、適切ではありません。指定するフィールドは、テーブルまたはテーブルの永続クラスのスーパークラスで定義される必要があります (もちろん、すべてのクラスをコンパイルする必要があります)。存在しないフィールドを指定すると、SQLCODE -31 エラーが生成されます。

通常のデータ・フィールドに加えて、CREATE INDEX を使用して以下のインデックスを作成できます。

- ・ [SERIAL フィールド](#) (%Counter フィールド)
- ・ [IDENTITY フィールド](#)



## ・ コレクションの ELEMENTS 値または KEYS 値

インデックスをストリーム値フィールドに対して作成することはできません。

IDKEY フィールド (プロパティ) のいずれかが [SQL 計算] の場合には、**IDKEY フィールドが複数ある**インデックスを作成することはできません。この制限は、IDKEY フィールドが 1 つしかないインデックスには適用されません。1 つのインデックスに複数の IDKEY フィールドがある場合は、各フィールドの区切り文字列として “||” (二重の垂直バー) が使用されています。したがって、IDKEY フィールドのデータにはこの文字列を使用できません。

### 埋め込みオブジェクト (%SerialObject) 内のフィールド

埋め込みオブジェクト内のフィールドにインデックスを作成するには、その埋め込みオブジェクトを参照するテーブル (%Persistent クラス) でインデックスを作成します。CREATE INDEX の field-name で、次の例のように、テーブル (%Persistent オブジェクト) 内の参照フィールドの名前を埋め込みオブジェクト (%SerialObject) 内のフィールド名にアンダーバーで結合して指定します。

### SQL

```
CREATE INDEX StateIdx ON TABLE Sample.Person (Home_State)
```

ここでは、Home は、State フィールドを含む埋め込みオブジェクト Sample.Address を参照する Sample.Person 内のフィールドです。

永続クラスの参照プロパティに関連付けられている埋め込みオブジェクト・レコードのみにインデックスが作成されます。%SerialObject プロパティにインデックスを直接作成することはできません。

埋め込みオブジェクト (シリアル・オブジェクトとも呼ばれる) の定義の詳細は、“[埋め込みオブジェクト \(%SerialObject\)](#)” を参照してください。埋め込みオブジェクトで定義されているプロパティ (フィールド) にインデックスを作成する方法の詳細は、“[埋め込みオブジェクト \(%SerialObject\) のプロパティのインデックス作成](#)” を参照してください。

## インデックス・クラス名

このオプションの構文を使用すると、ユーザが SQL を使用して機能インデックスのクラスとパラメータを指定できます。

以下に SQL の例を示します。

```
CREATE INDEX HistIdx ON TABLE Sample.Person (MedicalHistory) AS %iFind.Index.Basic (LANGUAGE='en', LOWER=1)
```

詳細は、“[SQL Search のためのソースのインデックス作成](#)” を参照してください。

## WITH DATA 節

この節を指定すると、インデックスの読み込みのみでクエリを解決できるようになるので、ディスク I/O の量が減り、パフォーマンスが向上します。

field-name で[文字照合](#)を使用する場合は、field-name と WITH DATA の datafield-name に同じフィールドを指定する必要があります。これにより、マスタ・マップに移動せずに、照合されていない値を取得できます。field-name 内の値が文字照合を使用しない場合は、WITH DATA の datafield-name でこのフィールドを指定しても利点がありません。

WITH DATA の datafield-name にはインデックスが作成されていないフィールドを指定できます。これにより、多くのクエリをマスタ・マップに移動せずにインデックスから実行できます。トレードオフは、維持する必要があるインデックスの数です。また、インデックスにデータを追加すると非常に大きくなるので、データを必要としない操作が遅くなります。

テーブルの永続クラスに対し、スーパークラスで定義されるフィールドを WITH DATA datafield-name で指定できます。

## UNIQUE キーワード

UNIQUE キーワードを使用して、インデックス内の各レコードが一意的な値を持つよう指定できます。具体的には、インデックス内に (つまり、インデックスがあるテーブル内に) 同じ照合された値を持つレコードは存在しないよう指定できます。既定では、大半のインデックスは、(大文字小文字に関係なく検索するように) 大文字の[文字照合](#)を使用します。この場合、

“Smith”と“SMITH”は等しい値であり、それぞれ一意的ではありません。CREATE INDEX では、既定以外のインデックス文字列照合を指定できません。[クラス定義でインデックスを定義する](#)ことで、個々のインデックスに対して異なる文字列照合を指定できます。

[ネームスペースの既定の照合](#)を変更することで、既定でフィールド/プロパティの大文字と小文字が区別されるようにすることができます。このオプションを変更するには、ネームスペースですべてのクラスをリコンパイルし、すべてのインデックスを再構築する必要があります。管理ポータルで[\[クラス\]](#)オプションを選択します。次にストアド・クエリのネームスペースを選択し、[\[コンパイル\]](#)オプションを使用して、対応するクラスをリコンパイルします。その後すべてのインデックスを再構築します。これで大文字と小文字が区別されるようになります。

**注意**           他のユーザがテーブルのデータにアクセスしている間はインデックスを再構築しないでください。再構築すると、クエリ結果が不正確になる可能性があります。

## BITMAP キーワード

BITMAP キーワードを使用する場合、このインデックスは、ビットマップ・インデックスであることを指定できます。ビットマップ・インデックスは、ビット位置が行 ID を表す 1 つ以上のビット文字列と、その行内のフィールドに固有の値が存在する (1) または存在しない (0) ことを表す各ビット値 (または組み合わせられる field-name フィールドの値) で構成されます。InterSystems SQL は、データの挿入、更新、削除を行う場合にこれらの位置ビットを (圧縮されたビット文字列として) 維持します。ビットマップ・インデックスと通常のインデックスのどちらを使用しても、INSERT、UPDATE、または DELETE 操作のパフォーマンスには大きな違いはありません。ビットマップ・インデックスは、クエリ操作の多数のタイプで非常に効率的です。ビットマップ・インデックスは、以下の特性を持ちます。

- ・ ビットマップ・インデックスを定義できるテーブル (クラス) は、システムによって割り当てられた正の整数の [RowID](#) 値を使用しているものか、カスタム ID 値の定義に[主キー IDKEY](#)を使用しているもの (IDKEY が [%Integer](#) タイプで MINVAL が 0 より大きい単一のプロパティ、または [%Numeric](#) タイプで SCALE が 0 と等しく MINVAL が 0 より大きい単一のプロパティに基づいている場合) に限られます。

`$SYSTEM.SQL.Util.SetOption()` メソッド SET

`status=$SYSTEM.SQL.Util.SetOption("BitmapFriendlyCheck",1,.oldval)` を使用すると、この制限をコンパイル時にチェックし、定義されたビットマップ・インデックスが [%Storage.SQL](#) クラス内で許可されるかどうかを判断するようにシステム全体用の構成パラメータを設定できます。このチェックは、[%Storage.SQL](#) を使用するクラスにのみ適用されます。既定値は 0 です。`$SYSTEM.SQL.Util.GetOption("BitmapFriendlyCheck")` を使用すると、このオプションの現在の構成を判断できます。

ビットマップ・インデックスは、既定の ([%Storage.Persistent](#)) 構造を使用しているテーブルに対してのみ定義できます。子テーブルなど、複合キーを持つテーブルでは、ビットマップ・インデックスを使用できません。テーブルの作成に (クラス定義の使用と反対に) DDL を使用する場合、これらの要件に合致し、ビットマップ・インデックスを有効に利用できます。

- ・ ビットマップ・インデックスは、フィールドの個別値の数が限られ、比較的少ない場合にのみ使用できます。例えば、性別、国籍、タイムゾーンなどのフィールドに適しています。ビットマップは、UNIQUE 制約を持つフィールドでは使用しないでください。フィールドに 10,000 を超える個別値がある場合や、複数のインデックス・フィールドに 10,000 を超える個別値がある場合は、ビットマップを使用しないでください。
- ・ ビットマップ・インデックスは、[WHERE](#) 節内で論理 AND および OR 演算子と組み合わせると非常に効果的です。一般に 2 つ以上のフィールドを組み合わせるクエリを実行する場合は、これらのフィールドにビットマップ・インデックスを定義すると効果的です。

詳細は、“[ビットマップ・インデックス](#)”を参照してください。

詳細は、“[ビットマップ・インデックス](#)”を参照してください。

## BITMAPEXTENT キーワード

ビットマップ・エクステント・インデックスは、テーブル自体のビットマップ・インデックスです。InterSystems SQL はこのインデックスを使用して、テーブル内のレコード (行) 数を返す [COUNT\(\\*\)](#) のパフォーマンスを改善します。テーブルには最

大 1 つのビットマップ・エクステント・インデックスを指定できます。複数のビットマップ・エクステント・インデックスを作成しようとすると、SQLCODE -400 エラーが発生し、メッセージ %msg ERROR #5445 :  
DDLBEIndex が表示されます。

CREATE TABLE を使用して定義されたすべてのテーブルでは、自動的にビットマップ・エクステント・インデックスが定義されます。この自動的に生成されたインデックスには、インデックス名 DDLBEIndex および SQL マップ名 %DDLBEIndex が割り当てられます。クラスとして定義されたテーブルには、\$ClassName のインデックス名と SQL マップ名で定義されたビットマップ・エクステント・インデックスを指定できます。

CREATE BITMAPEXTENT INDEX を使用して、テーブルにビットマップ・エクステント・インデックスを追加したり、自動的に生成されたビットマップ・エクステント・インデックスの名前を変更したりできます。指定する index-name は、テーブルの table-name と対応するクラス名にする必要があります。これはインデックスの SQL マップ名になります。field-name または WITH DATA 節は指定できません。

以下の例では、ビットマップ・エクステント・インデックスを、インデックス名 DDLBEIndex と SQL マップ名 Patient を指定して作成します。Sample.Patient に既に %DDLBEIndex ビットマップ・エクステント・インデックスがある場合、この例ではそのインデックスを次の SQL マップ名 Patient の名前に変更します。

## SQL

```
CREATE BITMAPEXTENT INDEX Patient ON TABLE Sample.Patient
```

詳細は、“[ビットマップ・エクステント・インデックス](#)” を参照してください。

## BITSlice キーワード

BITSlice キーワードを使用する場合、このインデックスは、ビットスライス・インデックスであることを指定できます。ビットスライス・インデックスは、計算で使用される数値データでのみ使用されます。ビットスライス・インデックスは、各数値データをバイナリのビット文字列として表します。ビットスライス・インデックスは、ビットマップ・インデックスのようにブーリアン・フラグを使用して数値データのインデックスを作成するのではなく、各レコードに個別のビット文字列である各数値のビット文字列を作成します。これは高速の集約計算のためにのみ使用される非常に特殊なタイプのインデックスです。例えば、以下はビットスライス・インデックスを使用する例です。

## SQL

```
SELECT SUM(Salary) FROM Sample.Employee
```

文字列データ・フィールドについてビットスライス・インデックスを作成できますが、ビットスライス・インデックスは、これらのデータ値をキャノニック形式の数値として表します。つまり、“abc” のような数値以外の文字列には 0 としてインデックスが作成されます。このタイプのビットスライス・インデックスは、文字列フィールドの値があるレコードをすばやくカウントし、NULL のレコードをカウントしないような場合に使用できます。

ビットスライス・インデックスは、SQL クエリ・オブティマイザによって使用されないため、WHERE 節では使用しないでください。

INSERT、UPDATE、または DELETE 操作を使用してビットスライス・インデックスを生成および維持すると、ビットマップ・インデックスや通常のインデックスを使用するよりも速度が大幅に遅くなります。複数のビットスライス・インデックスを使用したり、頻繁に更新されるフィールドでビットスライス・インデックスを使用すると、パフォーマンス・コストが増える可能性があります。

ビットスライス・インデックスは、正の整数値を持つ、システムが割り当てた行 ID を持つレコードでのみ使用できます。ビットスライス・インデックスは、1 つの field-name でのみ使用できます。WITH DATA 節は指定できません。

詳細は、“[ビットスライス・インデックス](#)” を参照してください。

## COLUMNAR キーワード

COLUMNAR キーワードを使用して、このインデックスが列指向インデックスになることを指定できます。目的の列を頻繁にクエリするものの、それが属するテーブルが行ストレージ構造を基盤としている場合に列指向インデックスを使用しま

す。既定では、テーブルのそれぞれの行が \$LIST として別々のグローバル添え字で格納されます。詳細は、“[列指向インデックス](#)” および “[SQL テーブルのストレージ・レイアウトの選択](#)” を参照してください。

## インデックスの再構築

CREATE INDEX 文を使用してインデックスの生成を行うと自動的にインデックスが構築されます。しかし、インデックスを明示的に再構築することが必要になる場合もあります。

**注意**      他のユーザがテーブルのデータにアクセスしている場合、インデックスを再構築するには追加手順が必要となります。これを行わないと、クエリ結果が不正確になる場合があります。詳細は、“[Building Indexes on an Active System](#)” を参照してください。

インデックスは以下のように構築または再構築できます。

- ・ [BUILD INDEX](#) SQL コマンドを使用する。
- ・ [管理ポータル](#)を使用して、指定したクラス (テーブル) のインデックスをすべて再構築します。
- ・ %BuildIndices() メソッドを使用する。

非アクティブ・テーブルのすべてのインデックスを再構築するには、以下を実行します。

### ObjectScript

```
SET status = ##class(myschema.mytable).%BuildIndices()
```

既定では、このコマンドにより、インデックスが再構築前に削除されます。この既定の削除をオーバーライドし、%PurgeIndices() メソッドを使用して、指定したインデックスを明示的に削除できます。特定範囲の ID の値で %BuildIndices() を呼び出す場合、既定では InterSystems IRIS はインデックスを削除しません。

以下のようにして、指定したインデックスを削除/再構築することもできます。

### ObjectScript

```
SET status = ##class(myschema.mytable).%BuildIndices($ListBuild("NameIDX","SpouseIDX"))
```

前述のように、壊れたインデックスの削除または再構築、あるいはインデックスの大文字/小文字区別の設定変更が可能です。[ビットマップ・インデックスを再圧縮する](#)には、削除/再構築ではなく %SYS.Maint.Bitmap メソッドを使用します。

詳細は、“[インデックスの構築](#)” を参照してください。

## 例

以下の例では、Fred という名前のテーブルを作成してから、Fred テーブルの **Lastword** フィールドと **Firstword** フィールドに対して (入力された名前 “Fred\_Index” から句読点を削除して) “FredIndex” というインデックスを作成します。

### SQL

```
CREATE TABLE Fred (
  TESTNUM      INT NOT NULL,
  FIRSTWORD    CHAR (30) NOT NULL,
  LASTWORD     CHAR (30) NOT NULL,
  CONSTRAINT FredPK PRIMARY KEY (TESTNUM))

CREATE INDEX Fred_Index
ON TABLE Fred (LASTWORD,FIRSTWORD)
```

以下の例は、Staff テーブルの **City** フィールドに対して、“CityIndex” という名前のインデックスを作成します。

### SQL

```
CREATE INDEX CityIndex ON Staff (City)
```

以下の例は、Staff テーブルの **EmpName** フィールドに対して、EmpIndex という名前のインデックスを作成します。フィールド内に同一の値を持つ行を避けるために、UNIQUE 制約を使用します。

## SQL

```
CREATE UNIQUE INDEX EmpIndex ON TABLE Staff (EmpName)
```

以下の例は、Purchase テーブルの **SKU** フィールドに対して、“SKUIndex” という名前のビットマップ・インデックスを作成します。BITMAP キーワードは、コードがビットマップ・インデックスであることを示します。

## SQL

```
CREATE BITMAP INDEX SKUIndex ON TABLE Purchases (SKU)
```

## 関連項目

- ・ [BUILD INDEX コマンド](#)
- ・ [DROP INDEX コマンド](#)
- ・ [SEARCH\\_INDEX 関数](#)
- ・ [テーブルの定義](#)
- ・ [インデックスの定義と構築](#)
- ・ [クエリ処理でのインデックスの使用](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)
- ・ [%Library.FunctionalIndex](#)

## CREATE METHOD (SQL)

クラスにメソッドを生成します。

### 構文

```
CREATE [STATIC] METHOD name (parameter_list)
  [ characteristics ]
  [ LANGUAGE SQL ]
  BEGIN code_body ;
END

CREATE [STATIC] METHOD name (parameter_list)
  [ characteristics ]
  LANGUAGE OBJECTSCRIPT
  { code_body }
```

### 説明

CREATE METHOD 文は、クラス・メソッドを生成します。このクラス・メソッドは、ストアド・プロシージャにすることも、しないことも可能です。メソッドを SQL ストアド・プロシージャとして公開されているクラスに生成するには、PROCEDURE キーワードを指定する必要があります。既定では、CREATE METHOD はストアド・プロシージャでもあるメソッドを生成しませんが、[CREATE PROCEDURE](#) 文はそのようなメソッドを常に生成します。

オプションの STATIC キーワードは、作成されたメソッドが静的 (クラス) メソッドであり、インスタンス・メソッドでないことを明確化するために用意されています。このキーワードには、実際の機能はありません。

メソッドの生成には、[GRANT](#) コマンドで指定された %CREATE\_METHOD 管理者特権が必要です。定義された所有者を持つ既存のクラスのメソッドを作成しようとする場合、クラスの所有者としてログインする必要があります。そうでない場合、操作は SQLCODE -99 エラーで失敗します。

クラス定義が[導入済みのクラス](#)の場合、クラスでメソッドを作成することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ `classname` DDL ” に設定されます。

以下の 2 つの例は、両方とも同じクラス・メソッドの作成を示しています。最初の例は CREATE METHOD を使用し、2 番目はクラス User.Letters のクラス・メソッドを定義します。

### SQL

```
CREATE METHOD RandCaseLetter(IN caps CHAR)
  RETURNS INTEGER
  PROCEDURE
  LANGUAGE OBJECTSCRIPT
  {
:Top
  IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
    ELSE {GOTO Top}}
  ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
    ELSE {GOTO Top}}
  ELSE {QUIT "case must be 'U' or 'L'"}
}

Class User.Letters Extends %Persistent [ DdlAllowed ]
{
  ClassMethod RandCaseLetter(caps) As %String [ SqlName = RandomLetter, SqlProc ]
  {
    Top
    IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
    ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
    ELSE {QUIT "case must be 'U' or 'L'"}
  }
}
```

SQL 文内からのメソッドの呼び出しの詳細は、“[ユーザ定義関数](#)”を参照してください。さまざまな状況での SQL ストアド・プロシージャの呼び出しについては、“[CALL](#)” 文を参照してください。



## 引数

### name

生成するメソッドの名前。この名前は、未修飾 (StoreName) にしてシステム全体の既定のスキーマ名を使用しても、スキーマ名を指定して修飾 (Patient.StoreName) してもかまいません。\$SYSTEM.SQL.Schema.Default() メソッドを使用して、現在のシステム全体の既定のスキーマ名を確認できます。初期のシステム全体の既定のスキーマ名は、クラス・パッケージ名 **User** に対応する **SQLUser** です。

FOR 特性 (後述) は、name で指定されたクラス名をオーバーライドすることに注意してください。この名前のメソッドが既に存在する場合、この操作は失敗して SQLCODE -361 エラーが発行されます。このエラーを回避するには、オプションのキーワード OR REPLACE を使用して既存のメソッドを変更または置換します。CREATE OR REPLACE METHOD には、DROP METHOD を呼び出して古いバージョンのメソッドを削除し、続いて CREATE METHOD を呼び出す操作と同じ効果があります。

生成されるクラスの名前は、スキーマ名に対応するパッケージ名の後にドット、“meth”、指定の name が順に続いたものになります。例えば、未修飾のメソッド名 RandomLetter で初期の既定のスキーマ SQLUser が使用される場合、生成されるクラス名は User.methRandomLetter となります。詳細は “SQL からクラス名への変換” を参照してください。

InterSystems SQL では、大文字/小文字の区別が異なるだけの重複メソッド名を指定することは許可されていません。既存メソッド名と大文字/小文字区別が異なるだけのメソッド名を指定した場合には、SQLCODE -400 エラーが生成されます。

### parameter-list

値をメソッドに渡すために使用されるパラメータのリストです。パラメータのリストは括弧で囲み、リストのパラメータ宣言はコンマで区切ります。パラメータを指定しない場合でも括弧は必須です。リスト内の各パラメータ宣言は、(先頭から順番に) 以下の要素で構成します。

- ・ パラメータ・モードが IN (入力値)、OUT (出力値)、または INOUT (変更値) のいずれであるかを指定するオプションのキーワード。省略した場合、既定のパラメータ・モードは IN です。
- ・ パラメータ名。パラメータ名では、大文字と小文字が区別されます。
- ・ パラメータのデータ型。
- ・ オプション：パラメータの既定値。DEFAULT キーワードの後ろに既定値を付けて指定できます。DEFAULT キーワードはオプションです。既定値が指定されていない場合、既定値は NULL であると見なされます。

メソッドからの出力値は、論理形式から表示/ODBC 形式に自動的に変換されます。

メソッドへの入力値は、既定では、表示/ODBC 形式から論理形式には変換されません。ただし、入力の表示形式から論理形式への変換は、\$SYSTEM.SQL.Util.SetOption(“SQLFunctionArgConversion”) メソッドを使用してシステム全体について構成できます。\$SYSTEM.SQL.Util.GetOption(“SQLFunctionArgConversion”) を使用すると、このオプションの現在の構成を判断できます。

以下の例では、2 つの入力パラメータを指定しており、どちらにも既定値があります。オプションの DEFAULT キーワードは、1 つ目のパラメータでは指定されていますが、2 つ目のパラメータでは省略されています。

### SQL

```
CREATE METHOD RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END
```

### characteristics

利用できるキーワードは以下のとおりです。

FOR className	メソッドを生成するクラス名を指定します。そのクラスが存在しない場合は新規作成します。メソッド名を認証することによりクラス名を指定することもできます。FOR 節で指定されたクラス名の方が、メソッド名の認証により指定されたクラス名よりも優先されます。
FINAL	サブクラスによってメソッドがオーバーライドされないように指定します。既定では、メソッドは Final ではありません。FINAL キーワードはサブクラスによって継承されます。
PRIVATE	メソッドがそれ自身のクラス、またはサブクラスの他のメソッドによってのみ起動できることを指定します。既定ではメソッドはパブリックで、制限なしに起動できます。この制限はサブクラスによって継承されます。
PROCEDURE	メソッドが SQL ストアド・プロシージャであることを指定します。ストアド・プロシージャはサブクラスに継承されます (このキーワードは、PROC と略することができます)。
RESULT SETS DYNAMIC RESULT SETS [n]	作成されるメソッドに <a href="#">ReturnResultsets</a> キーワードが含まれることを指定します。この characteristics 句のすべての形式は同義語です。
RETURNS datatype	このメソッドの呼び出しで返される値のデータ型を指定します。RETURNS が省略されると、メソッドは値を返すことができません。この指定内容はサブクラスによって継承され、サブクラスによって変更できます。この datatype には、MINVAL、MAXVAL、SCALE などのタイプのパラメータを指定できます。例えば RETURNS DECIMAL(19,4) のように指定します。値が返されるとき、datatype の長さは無視されます。例えば、RETURNS VARCHAR(32) は、このメソッドの呼び出しによって返されるあらゆる長さの文字列を受け取ることができます。
SELECTMODE mode	LANGUAGE が SQL (既定値) の場合にのみ使用されます。これを指定した場合、InterSystems IRIS は、 <a href="#">#SQLCOMPILE SELECT=mode</a> 文を対応するクラス・メソッドに追加することで、指定された SELECTMODE で、メソッドで定義された SQL 文を生成します。mode に指定できる値は、LOGICAL、ODBC、RUNTIME、および DISPLAY です。既定は LOGICAL です。

メソッドに対して有効でないクエリ・キーワード (CONTAINSID や RESULTS など) を指定する場合、システムは SQLCODE -47 エラーを発行します。重複するクエリ・キーワード (例: FINAL FINAL) を指定した場合は、SQLCODE -44 エラーが生成されます。

SELECTMODE 節は、SELECT クエリ操作、および INSERT と UPDATE 操作で使います。これは、コンパイル時の選択モードを指定します。SELECTMODE に指定した値は、[#sqlcompile select=mode](#) のように InterSystems ObjectScript のクラス・メソッド・コードの先頭に追加されます。詳細は、["#sqlcompile select"](#) を参照してください。

- SELECT クエリでは、SELECTMODE はデータを返すモードを指定します。mode 値が LOGICAL の場合は、論理 (内部保存) 値が返されます。例えば、日付は \$HOROLOG 形式で返されます。mode 値が ODBC の場合、論理と ODBC 間の変換が適用され、ODBC 形式値が返されます。mode 値が DISPLAY の場合、論理と表示間の変換が適用され、表示形式値が返されます。mode 値が RUNTIME の場合、実行時に表示モードを (LOGICAL、ODBC、または DISPLAY に) 設定できます。
- INSERT または UPDATE 操作では、SELECTMODE RUNTIME オプションで、表示形式 (DISPLAY または ODBC) から論理格納形式への入力データ値の自動変換がサポートされています。このコンパイルされた表示データから論理データへの変換コードは、SQL コード実行時の選択モード設定が LOGICAL (すべての InterSystems SQL 実行インタフェースの既定値) の場合のみ適用されます。



“[ダイナミック SQL の使用法](#)” に説明されているように、SQL コードが実行されると、**%SQL.Statement** クラスの **%SelectMode** プロパティが、実行時の選択モードを指定します。SelectMode オプションの詳細は、“[データ表示オプション](#)” を参照してください。

## LANGUAGE

code\_body に使用している言語を指定するキーワード節。使用可能な節は、LANGUAGE OBJECTSCRIPT (ObjectScript の場合) か、LANGUAGE SQL です。LANGUAGE 節が省略される場合は、SQL が既定です。

### code\_body

生成されるメソッドのプログラム・コード。このコードは SQL または ObjectScript で指定します。使用する言語は LANGUAGE 節と一致させる必要があります。ただし、ObjectScript のコードには埋め込み SQL を記述できます。

InterSystems IRIS は、メソッドの実際のコードを生成するために提供されたコードを使用します。

指定するコードが SQL の場合、メソッドの生成時にコード行が追加されます。追加のコードでは、SQL が ObjectScript “ラップ” に埋め込まれ、(必要に応じて) プロシージャ・コンテキスト・ハンドラが作成され、戻り値が処理されます。以下はこの InterSystems IRIS から生成されたラップ・コードの例です。

### ObjectScript

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql( SELECT col FROM tbl )
QUIT $GET(title)
```

指定するコードが OBJECTSCRIPT である場合、ObjectScript コードは中括弧で囲む必要があります。すべてのコード行を列 1 からインデントする必要がありますが、ラベルとマクロ・プリプロセッサ指示文はその必要はありません。ラベルやマクロ指示文の前には、列 1 でコロン (:) を付加する必要があります。

ObjectScript コードの場合、“ラップ” (変数を NEW で処理して、QUIT を使用して終了し、(オプションで) 完了時に値を返す) を明示的に定義する必要があります。

PROCEDURE キーワードを指定することで、メソッドをストアード・プロシージャとして公開することができます。ストアード・プロシージャが呼び出されると、クラス **%Library.SQLProcContext** のオブジェクトが **%sqlcontext** 変数でインスタンス化されます。このプロシージャ・コンテキスト・ハンドラによって、プロシージャとその呼び出し元 (ODBC サーバなど) 間でプロシージャ・コンテキストの受け渡しが行われます。

**%sqlcontext** は、エラー・オブジェクト、SQLCODE エラー・ステータス、SQL 行カウント、およびエラー・メッセージを含む、いくつかのプロパティで構成されます。以下の例は、これらプロパティの設定に使用される値を示しています。

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=msg
```

SQLCODE と %ROWCOUNT の値は、SQL 文の実行によって自動的に設定されます。**%sqlcontext** オブジェクトは、実行される前に毎回リセットされます。

または、**%SYSTEM.Error** オブジェクトをインスタンス化し、**%sqlcontext.Error** として設定することによって、エラー・コンテキストを設定することができます。

## 例

以下の 2 つの例は、両方とも同じクラス・メソッドの作成を示しています。最初の例は CREATE METHOD を使用し、2 番目はクラス User.Letters のクラス・メソッドを定義します。

## SQL

```

CREATE METHOD RandCaseLetter(IN caps CHAR)
  RETURNS INTEGER
  PROCEDURE
LANGUAGE OBJECTSCRIPT
{
  :Top
  IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
    ELSE {GOTO Top}}
  ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
    ELSE {GOTO Top}}
  ELSE {QUIT "case must be 'U' or 'L'"}
}

Class User.Letters Extends %Persistent [ DdlAllowed ]
{
  ClassMethod RandCaseLetter(caps) As %String [ SqlName = RandomLetter, SqlProc ]
  {
    Top
    IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
    ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
    ELSE {QUIT "case must be 'U' or 'L'"}
  }
}

```

以下の例では、2 つの入力パラメータを指定しており、どちらにも既定値があります。オプションの DEFAULT キーワードは、1 つ目のパラメータでは指定されていますが、2 つ目のパラメータでは省略されています。

## SQL

```

CREATE METHOD RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END

```

以下の例は、SQL コードで CREATE METHOD を使用して、Sample.Employee クラスに UpdateSalary メソッドを生成しています。

以下の例は、SQL コードで CREATE METHOD を使用して、Sample.Employee クラスに UpdateSalary メソッドを生成しています。

## SQL

```

CREATE METHOD UpdateSalary ( IN SSN VARCHAR(11), IN Salary INTEGER )
FOR Sample.Employee
BEGIN
  UPDATE Sample.Employee SET Salary = :Salary WHERE SSN = :SSN;
END

```

以下の例では、ランダムな大文字を生成するプロシージャとして格納される RandomLetter() メソッドを作成します。その後、このメソッドを関数として SELECT 文内で呼び出すことができます。RandomLetter() メソッドを削除するため、DROP METHOD が指定されています。

## SQL

```

CREATE METHOD RandomLetter()
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
  :Top
  SET x=$RANDOM(91)
  IF x<65 {GOTO Top}
  ELSE {QUIT $CHAR(x)}
}

```

## SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter()
```

## SQL

```
DROP METHOD RandomLetter
```

以下の埋め込み SQL の例は、ObjectScript コードで CREATE METHOD を使用して、SQLUser.MyStudents クラスに TraineeTitle メソッドを生成し、Title 値を返します。

## ObjectScript

```
&sql(CREATE METHOD TraineeTitle(
  IN SSN VARCHAR(11),
  INOUT Title VARCHAR(50) )
  RETURNS VARCHAR(30)
  FOR SQLUser.MyStudents
  LANGUAGE OBJECTSCRIPT
  {
    NEW SQLCODE,%ROWCOUNT
    &sql(SELECT Title INTO :Title FROM Sample.Employee
      WHERE SSN = :SSN)
    IF $GET(%sqlcontext)=' ' {
      SET %sqlcontext.%SQLCODE=SQLCODE
      SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
    QUIT
  })
IF SQLCODE=0 { WRITE !,"Created a method" QUIT}
ELSEIF SQLCODE=-361 { WRITE !,"Method already exists SQLCODE: ",SQLCODE
  &sql(DROP METHOD TraineeTitle FROM SQLUser.MyStudents)
  IF SQLCODE=0 { WRITE !,"Dropped a method" QUIT}}
ELSE { WRITE !,"SQL error: ",SQLCODE }
```

%sqlcontext オブジェクトを使用します。対応する SQL 変数を使用して、その %SQLCODE プロパティおよび %ROWCOUNT プロパティを設定します。メソッドの LANGUAGE OBJECTSCRIPT キーワードに続く ObjectScript コードを中括弧で囲むことに注意してください。ObjectScript コードには、&sql でマークされ、角括弧で囲まれた埋め込み SQL コードがあります。

## セキュリティおよび特権

CREATE METHOD コマンドは、ユーザに %Development:USE 権限が必要な特権操作です。このような権限は管理ポータルを介して付与できます。これらの特権なしで CREATE METHOD コマンドを実行すると、SQLCODE -99 エラーが発生し、コマンドは失敗します。

適切な特権を持たないユーザでも、以下の 2 つの状況のいずれかではこのコマンドを実行できます。

- ・ 埋め込み SQL を介してコマンドを実行する場合。この場合は特権が確認されません。
- ・ 特権を確認しないことをユーザが明示的に指定する場合。例えば、checkPriv 引数を 0 に設定して %Prepare() を呼び出すか、%SQL.Statement に対して %ExecDirectNoPriv() を呼び出します。

## 関連項目

- ・ [CALL](#)
- ・ [CREATE PROCEDURE](#)
- ・ [DROP METHOD](#)
- ・ [ストアド・プロシージャの定義と使用](#)

## CREATE ML CONFIGURATION (SQL)

ML 構成を作成します。

### 構文

```
CREATE [ OR REPLACE ] ML CONFIGURATION ml-configuration-name PROVIDER provider-name
[ %DESCRIPTION description ] [ USING json-object-string ]
[ provider-connection-settings ]
```

### 引数

<i>ml-configuration-name</i>	作成する ML 構成の名前。テーブル名と同様の追加の名前付け制約に従う、有効な識別子です。ML 構成名は未修飾です (mlconfig-name)。ML 構成名が未修飾の場合は、既定のスキーマ名が使用されます。
PROVIDER <i>provider-name</i>	機械学習プロバイダの名前を指定する文字列。値は次のとおりです。 <ul style="list-style-type: none"> <li>AutoML</li> <li>H2O</li> <li>DataRobot</li> <li>PMML</li> </ul>
%DESCRIPTION <i>description</i>	オプション – 文字列。ML 構成の説明テキスト。この後の説明を参照してください。
USING <i>json-object-string</i>	オプション – 1 つ以上のキーと値のペアを指定する JSON 文字列。この後の説明を参照してください。
<i>provider-connection-settings</i>	接続に必要な追加の設定。機械学習プロバイダによって異なります。この後の説明を参照してください。

### 説明

CREATE ML CONFIGURATION コマンドは、モデルをトレーニングするための ML 構成を作成します。以下の 1 つ以上のプロパティを指定できます。

- ・ プロバイダ (必須)
- ・ 説明
- ・ [USING 節](#)
- ・ [プロバイダの接続設定](#)

### ML 構成の説明

%DESCRIPTION は一重引用符で囲まれたテキスト文字列を受け入れます。これを使用して、構成を文書化するための説明を入力できます。このテキストの長さに制限はなく、空白スペースを含むすべての文字を使用できます。

### USING

構成の既定の USING 節を指定できます。この節は、1 つ以上のキーと値のペアで構成される JSON 文字列を受け入れます。TRAIN MODEL を実行すると、既定では構成の USING 節が使用されます。

指定するパラメータが、選択するプロバイダによって認識されることを確認する必要があります。これを行わないと、トレーニング時にエラーが発生することがあります。

H2O をプロバイダとして指定した例

```
CREATE ML CONFIGURATION h2o_config PROVIDER H2O USING {"seed":100, "nfolds":4}
```

## プロバイダの接続設定

構成で指定されたプロバイダによっては、正常な接続を確立するために、いくつかの追加フィールドに入力しなければならないことがあります。

### DataRobot

DataRobot に正常に接続するには、以下の値を指定する必要があります。

- ・ URL [ = ] *url-string* - *url-string* は DataRobot エンドポイントの URL です。
- ・ APITOKEN [ = ] *token-string* - *token-string* は DataRobot AutoML サーバにアクセスするためのクライアント API トークンです。

DataRobot の完全な ML 構成を、次のようなクエリで作成できます。

```
CREATE ML CONFIGURATION datarobot-configuration PROVIDER DataRobot1 URL url-string APITOKEN token-string
```

*url-string* と以下に対して適切な値を指定します：*token-string*

## 必要なセキュリティ特権

CREATE ML CONFIGURATION を呼び出すには、%CREATE\_ML\_CONFIGURATION 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%CREATE\_ML\_CONFIGURATION 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## 構成の名前付け規約

構成名は、[識別子](#)の規則に従い、以下のような制約を受けます。既定の構成名は、簡単な識別子です。構成名は 256 文字を超えることはできません。また、構成名では大文字と小文字が区別されません。

InterSystems IRIS® は構成名を使用して、対応するクラス名を生成します。クラス名には英数字 (文字および数字) のみを使用し、最初の 96 文字は一意である必要があります。このクラス名を生成するために、InterSystems IRIS は最初に構成名から句読点を削除し、次に最初の 96 文字が一意である識別子を生成します。その際、クラス名の一意性を維持するために、必要に応じて最後の文字を (0 で始まる) 整数に置き換えます。InterSystems IRIS は有効な構成名から一意のクラス名を生成しますが、構成の名前を付ける際には、この名前の生成に伴う以下の制約について考慮する必要があります。

- ・ 構成名には、最低でも 1 文字を含める必要があります。ビュー名の先頭の文字または最初の句読点に続く文字は、数字以外の文字にする必要があります。
- ・ InterSystems IRIS は 16 ビット (ワイド) 文字の構成名をサポートします。\$ZNAME テストに合格した文字は、有効な文字です。
- ・ 構成名の最初の文字が句読点文字の場合、2 番目の文字に数字を指定することはできません。これにより、SQLCODE -400 エラーが発生し、生成される %msg の値は "エラー #5053: クラス名 'schema.name' は正しくありません" になります (句読点文字なし)。例えば、指定した構成名が %7A の場合、生成される %msg は "エラー #5053: クラス名 'User.7A' は正しくありません" になります。
- ・ 生成されたクラス名には句読点が含まれないため、句読点のみ既存の構成名と異なる構成名を作成することは可能ですが、お勧めできません。この場合、InterSystems IRIS は、一意のクラス名を作成するために名前最後の文字を (0 で始まる) 整数に置き換えます。
- ・ 構成名は 96 文字よりも大幅に長くすることができますが、最初の 96 の英数字が異なるように構成名を作成すると処理ははるかに容易になります。

構成名は未修飾にする必要があります。構成名が未修飾の場合は (viewname)、[システム全体の既定のスキーマ名](#)が使用されます。

ML 構成で同じ名前を使用するように再定義する場合は、OR REPLACE オプションを指定して、既存の ML 構成を異なる動作に置き換えることができます。

## 例

```
CREATE ML CONFIGURATION autoML_config PROVIDER AutoML %DESCRIPTION 'my AutoML configuration!'
```

## 関連項目

- ・ [ALTER ML CONFIGURATION](#)、[DROP ML CONFIGURATION](#)

# CREATE MODEL (SQL)

モデル定義を作成します。

## 構文

### モデルの分類または回帰

```
CREATE MODEL [ IF NOT EXISTS ] model-name
  PREDICTING ( label-column )
  FROM model-source
  [ USING json-object ]
```

```
CREATE MODEL [ IF NOT EXISTS ] model-name
  PREDICTING ( label-column )
  WITH feature-column-clause
  [ USING json-object ]
```

```
CREATE MODEL [ IF NOT EXISTS ] model-name
  PREDICTING ( label-column )
  WITH feature-column-clause
  FROM model-source
  [ USING json-object ]
```

### 時系列モデル

```
CREATE [ TIME ] SERIES MODEL [ IF NOT EXISTS ] model-name
  PREDICTING ( label-column1, label-column2, ...)
  BY ( timestep )
  FROM model-source
  [ USING json-object ]
```

## 引数

このコマンド概要は、CREATE MODEL の有効な形式を示します。CREATE MODEL コマンドには FROM 節または WITH 節 (または両方) が必要です。

model-name	作成するモデル定義の名前。テーブル名と同様の追加の名前付け制約に従う、有効な <a href="#">識別子</a> です。モデル名は未修飾です (modelname)。モデル名が未修飾の場合は、既定のスキーマ名が使用されます。
PREDICTING ( label-column )	予測される列名 (ラベル列)。標準の <a href="#">識別子</a> です。この後の <a href="#">説明</a> を参照してください。
WITH feature-column-clause	列名とそのデータ型として、または列名とデータ型のコンマ区切りリストとしての、モデル (特徴列) への入力。各列名は、標準の <a href="#">識別子</a> です。
FROM model-source	モデルの構築元にするテーブルまたはビュー。 <a href="#">テーブル</a> 、 <a href="#">ビュー</a> 、または <a href="#">結合</a> の結果です。
USING json-object-string	オプション - 1 つ以上のキーと値のペアを指定する JSON 文字列。詳細は、この後の <a href="#">説明</a> を参照してください。
BY ( timestep )	時系列モデルが構築される時間ベースのデータを含む列。

## 説明

CREATE MODEL コマンドは、指定された構造のモデル定義を作成します。少なくとも以下が含まれます。

- ・ モデル名
- ・ ラベル列 (時系列モデルの場合は複数の列)



## ・ 特徴列

回帰モデルと分類モデルは、ほとんど同じ方法で作成され、考慮事項も同じです。ただし、時系列モデルでは別の考慮事項が必要になるため、若干異なる構文を採用しています。これらのモデルのタイプ間の相違点は、以下の該当する節で列挙しています。

## 予測

入力列 (特徴列) を基にして、モデルが予測する出力列 (ラベル列) を指定する必要があります。例えば、スパム・メールである電子メールを識別する SpamFilter モデルを設計する場合、IsSpam というラベル列を使用できます。これは、指定された電子メールがスパムかどうかを指定するブーリアン値です。この列のデータ型を指定することもできます。指定しない場合、IntegratedML は以下の型を推測します。

```
CREATE MODEL SpamFilter PREDICTING (IsSpam) FROM EmailData
CREATE MODEL SpamFilter PREDICTING (IsSpam binary) FROM EmailData
```

時系列モデルの作成時には、複数の列の値を予測したいことがよくあります。これを行うには、コンマ区切りのリストで予測する列の名前を指定します。この列のデータ型を指定することもできます。指定しない場合、IntegratedML は以下の型を推測します。モデルがテーブル内のすべての列の値を予測するように指定するには、アスタリスク (\*) を使用します。

```
CREATE TIME SERIES MODEL WeatherForecast PREDICTING (Temp, Precipitation, Humidity, UVIndex) BY (Date)
FROM WeatherData
CREATE TIME SERIES MODEL WeatherForecast PREDICTING (*) BY (DATE) FROM WeatherData
```

## WITH と FROM

モデルのスキーマ特性を指定するには、分類または回帰のモデル定義に WITH または FROM (あるいはその両方) が含まれている必要があります。時系列モデルには FROM 節を含める必要があり、WITH を含めることはできません。

### WITH

WITH を使用して、モデル定義に含める入力列 (特徴) を指定できます。文で FROM 節を使用している場合でも、各列のデータ型を指定する必要があることに注意してください。

```
CREATE MODEL SpamFilter PREDICTING (IsSpam) WITH (email_length int, subject_title varchar)
CREATE MODEL SpamFilter PREDICTING (IsSpam) WITH (email_length int, subject_title varchar) FROM EmailData
```

### FROM

FROM を使用すると、各列を個別に識別しなくても、指定されたテーブルまたはビューのすべての列を使用できます。

```
CREATE MODEL SpamFilter PREDICTING (IsSpam) FROM EmailData
```

この節は一般的であり、任意のサブクエリ式を指定できます。IntegratedML は各列のデータ型を推測します。FROM を使用して、このモデル定義を使用する将来の TRAIN MODEL 文の既定のデータ・セットを指定します。FROM と WITH を一緒に使用して、既定のデータ・セットを指定し、特徴列に明示的に名前を付けることができます。

WITH 節がない場合、IntegratedML は各列のデータ型を推測して、FROM 節の結果を以下のクエリであるかのように暗黙的に使用します。

```
SELECT * FROM model-source
```

## USING

モデル定義の既定の USING 節を指定できます。この節は、1 つ以上のキーと値のペアで構成される JSON 文字列を受け入れます。TRAIN MODEL を実行すると、既定ではモデル定義の USING 節が使用されます。ML 構成の USING 節で指定したすべてのパラメータは、モデル定義の USING 節の同じパラメータを上書きします。

指定するパラメータが、選択するプロバイダによって認識されることを確認する必要があります。これを行わないと、トレーニング時にエラーが発生することがあります。



## 時系列パラメータ

時系列モデルは、USING 節の 3 つのオプション・パラメータもサポートしています。これらのパラメータでは大文字と小文字が区別されません。これらは、以下のとおりです。

- forward では、予測する将来の時間ステップの数を正の整数で指定します。元のデータセットの最新の時刻または日付の後に、予測された行が表示されます。これと backward 設定の両方を同時に指定することもできます。
- backward では、予測する過去の時間ステップの数を正の整数で指定します。元のデータセットの最も早い時刻または日付の前に、予測された行が表示されます。これと forward 設定の両方を同時に指定することもできます。AutoML プロバイダではこのパラメータは無視されます。
- frequency では、予測する時間ステップのサイズと単位の両方を正の整数として指定し、その後に時間の単位を表す 1 文字を付加します。この値を指定しない場合、データ内で最も一般的な時間ステップが指定されます。DataRobot プロバイダではこのパラメータは無視されます。

時間の単位の 1 文字での省略形を以下のテーブルに示します。

テーブル B-1:

省略形	時間の単位
y	year
m	month
w	week
d	day
h	hour
t	minute
s	second

## 必要なセキュリティ特権

CREATE MODEL を呼び出すには、%MANAGE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%MANAGE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## モデルの名前付け規約

モデル名は、[識別子](#)の規則に従い、以下のような制約を受けます。既定のモデル名は、簡単な識別子です。モデル名は 256 文字を超えることはできません。また、モデル名では大文字と小文字が区別されません。

InterSystems IRIS はモデル名を使用して、対応するクラス名を生成します。クラス名には英数字 (文字および数字) のみを使用し、最初の 96 文字は一意である必要があります。このクラス名を生成するために、InterSystems IRIS は最初にモデル名から句読点を削除し、次に最初の 96 文字が一意である識別子を生成します。その際、クラス名の一意性を維持するために、必要に応じて最後の文字を (0 で始まる) 整数に置き換えます。InterSystems IRIS は有効なモデル名から一意のクラス名を生成しますが、モデルの名前を付ける際には、この名前の生成に伴う以下の制約について考慮する必要があります。

- モデル名には、最低でも 1 文字を含める必要があります。ビュー名の先頭の文字または最初の句読点に続く文字は、数字以外の文字にする必要があります。
- InterSystems IRIS は 16 ビット (ワイド) 文字のモデル名をサポートします。\$ZNAME テストに合格した文字は、有効な文字です。
- モデル名の最初の文字が句読点文字の場合、2 番目の文字に数字を指定することはできません。これにより、SQLCODE -400 エラーが発生し、生成される %msg の値は "エラー #5053: クラス名 'schema.name' は正しくありま

せん”になります (句読点文字なし)。例えば、指定したモデル名が %7A の場合、生成される %msg は “エラー #5053: クラス名 'User.7A' は正しくありません” になります。

- ・ 生成されたクラス名には句読点が含まれないため、句読点のみ既存のモデル名と異なるモデル名を作成することは可能ですが、お勧めできません。この場合、InterSystems IRIS は、一意のクラス名を作成するために名前の最後の文字を (0 で始まる) 整数に置き換えます。
- ・ モデル名は 96 文字よりも大幅に長くすることができますが、最初の 96 の英数字が異なるようにモデル名を作成すると処理がはるかに容易になります。

モデル名は未修飾にする必要があります。モデル名が未修飾の場合は (viewname)、[システム全体の既定のスキーマ名](#)が使用されます。

## 例

### SQL

```
CREATE MODEL PatientReadmit PREDICTING (IsReadmitted) FROM patient_table USING {"seed": 3}
CREATE MODEL PatientReadmit PREDICTING (IsReadmitted) WITH (age, gender, encounter_type, admit_reason,
starttime, endtime, prior_visits, diagnosis, comorbidities)
CREATE TIME SERIES MODEL BusinessGrowth PREDICTING (*) BY (date) FROM BusinessData USING {"Forward":5}
```

## 関連項目

- ・ [ALTER MODEL](#)、[DROP MODEL](#)、[TRAIN MODEL](#)

# CREATE PROCEDURE (SQL)

SQL ストアド・プロシージャとして公開されるメソッドまたはクエリを生成します。

## 構文

```
CREATE PROCEDURE procname(parameter_list) [ characteristics ]
[ LANGUAGE SQL ]
BEGIN code_body ;
END
```

```
CREATE PROCEDURE procname(parameter_list) [ characteristics ]
LANGUAGE OBJECTSCRIPT
{ code_body }
```

```
CREATE PROCEDURE procname(parameter_list) [ characteristics ]
LANGUAGE { JAVA | PYTHON | DOTNET }
EXTERNAL NAME external-stored-procedure
```

## 説明

CREATE PROCEDURE 文は既定で SQL ストアド・プロシージャとして公開されるメソッドまたはクエリを生成します。ストアド・プロシージャは、現在のネームスペース内のすべてのプロセスによって呼び出すことができます。ストアド・プロシージャはサブクラスに継承されます。

- LANGUAGE SQL の場合、ストアド・プロシージャとして公開するクエリを生成するために code\_body に SELECT 文を含める必要があります。コードに SELECT 文が含まれていない場合、CREATE PROCEDURE でメソッドが生成されます。
- LANGUAGE OBJECTSCRIPT の場合、ストアド・プロシージャとして公開するクエリを生成するために、code\_body で Execute() メソッドおよび Fetch() メソッドを呼び出す必要があります。また、Close()、FetchRows()、GetInfo() の各メソッドを呼び出すことも可能です。コードで Execute() および Fetch() を呼び出していない場合は、CREATE PROCEDURE でメソッドが生成されます。

ストアド・プロシージャとして公開されないメソッドを生成するには、[CREATE METHOD](#) または [CREATE FUNCTION](#) を使用します。ストアド・プロシージャとして公開されないクエリを生成するには、[CREATE QUERY](#) を使用します。これらの文は、PROCEDURE 特性キーワードを指定することで、ストアド・プロシージャとして公開されるメソッドまたはクエリの生成にも使用できます。

プロシージャの生成には、[GRANT](#) コマンドで指定された %CREATE\_PROCEDURE 管理者特権が必要です。定義された所有者を持つ既存のクラスのプロシージャを作成しようとする場合、クラスの所有者としてログインする必要があります。そうでない場合、操作は SQLCODE -99 エラーで失敗します。

クラス定義が[導入済みのクラス](#)の場合、クラスでプロシージャを作成することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ `classname` DDL ” に設定されます。

ストアド・プロシージャは、[CALL](#) 文を使用して実行されます。

SQL 文内からのメソッドの呼び出しの詳細は、“[ユーザ定義関数](#)” を参照してください。

## 引数

### procname

ストアド・プロシージャとして生成するメソッドまたはクエリの名前です。パラメータを指定しない場合でも、procname の後には括弧を付加する必要があります。プロシージャ名は以下の形式のいずれかで指定することができます。

- 未修飾：[既定のスキーマ名](#)が使用されます。例えば、MedianAgeProc() です。
- 修飾：スキーマ名を指定します。例えば、Patient.MedianAgeProc() です。

- ・ 複数レベル：対応するクラス・パッケージ・メンバに一致させるために 1 つ以上のスキーマ・レベルで修飾します。この場合、procname に含めることができるピリオド文字は 1 つだけです。対応するクラス・メソッド名の他のピリオドは下線に置き換えられます。ピリオドは、最下位レベルのクラス・パッケージ・メンバの前に指定します。例えば、%SYSTEM.SQL\_GetROWID() や %SYS\_PTools.StatsSQL\_Export() のように指定します。

未修飾：procname では **既定のスキーマ名** が使用されます。\$SYSTEM.SQL.Schema.Default() メソッドを使用して、現在のシステム全体の既定のスキーマ名を確認できます。初期のシステム全体の既定のスキーマ名は、クラス・パッケージ名 **User** に対応する SQLUser です。

FOR 特性(後述)は、procname で指定されたクラス名をオーバーライドすることに注意してください。この名前のプロシージャが既に存在する場合、この操作は失敗して SQLCODE -361 エラーが発行されます。

InterSystems SQL は SQL procname を使用して、対応するクラス名を生成します。この名前は、スキーマ名に対応するパッケージ名と、その後に順に続くドット、“proc”、および指定のプロシージャ名で構成されます。例えば、未修飾のプロシージャ名 RandomLetter() で既定のスキーマ SQLUser が使用される場合、生成されるクラス名は User.procRandomLetter() となります。詳細は [“SQL からクラス名への変換”](#) を参照してください。

InterSystems SQL では、大文字/小文字が異なるだけの procname を指定することは許可されていません。既存のプロシージャ名と大文字/小文字が異なるだけの procname を指定した場合は、SQLCODE -400 エラーが生成されます。

指定した procname が現在のネームスペース内に既に存在する場合は、SQLCODE -361 エラーが生成されます。指定の procname が現在のネームスペースに既に存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.ProcedureExists() メソッドを使用します。

オプションのキーワード OR REPLACE を指定して、エラーが発生することなく、既存のプロシージャを変更または置換します。CREATE OR REPLACE PROCEDURE には、DROP PROCEDURE を呼び出して古いバージョンのプロシージャを削除し、続いて CREATE PROCEDURE を呼び出す操作と同じ効果があります。

**注釈** InterSystems SQL のプロシージャ名と InterSystems TSQL のプロシージャ名は、同じ名前セットを共有しています。したがって、同じネームスペース内の TSQL プロシージャと同じ名前を持つ SQL プロシージャを作成することはできません。これを実行しようとすると、SQLCODE -400 エラーが返されます。

## parameter\_list

値をメソッドまたはクエリに渡すために使用するパラメータのリストです。パラメータのリストは括弧で囲み、リストのパラメータ宣言はコンマで区切ります。パラメータを指定しない場合も括弧は必須です。

リスト内の各パラメータ宣言は、(先頭から順番に) 以下の要素で構成します。

- ・ パラメータ・モードが IN (入力値)、OUT (出力値)、または INOUT (変更値) のいずれであるかを指定するオプションのキーワード。省略した場合、既定のパラメータ・モードは IN です。
- ・ パラメータ名。パラメータ名では、大文字と小文字が区別されます。
- ・ パラメータの [データ型](#)。
- ・ オプション：パラメータの既定値。DEFAULT キーワードの後ろに既定値を付けて指定できます。DEFAULT キーワードはオプションです。既定値が指定されていない場合、既定値は NULL であると見なされます。

以下の例では、入力パラメータが 2 つある (どちらにも既定値がある) ストアド・プロシージャを作成します。一方の入力パラメータではオプションの DEFAULT キーワードを指定し、もう一方の入力パラメータではこのキーワードを省略します。

## SQL

```
CREATE PROCEDURE AgeQuerySP(IN topnum INT DEFAULT 10,IN minage INT 20)
BEGIN
  SELECT TOP :topnum Name,Age FROM Sample.Person
  WHERE Age > :minage ;
END
```

以下の例は、前述の例と機能的に同じです。オプションの DEFAULT キーワードは省略されます。

## SQL

```
CREATE PROCEDURE AgeQuerySP(IN topnum INT 10,IN minage INT 20)
BEGIN
  SELECT TOP :topnum Name,Age FROM Sample.Person
  WHERE Age > :minage ;
END
```

CALL AgeQuerySP(6,65)、CALL AgeQuerySP(6)、CALL AgeQuerySP(,65)、CALL AgeQuerySP() はすべて、このプロシージャで有効な CALL 文です。

以下の例では、3 つのパラメータを持つストアド・プロシージャとして公開されるメソッドを作成します。

## SQL

```
CREATE PROCEDURE UpdatePaySP
  (IN Salary INTEGER DEFAULT 0,
   IN Name VARCHAR(50),
   INOUT PayBracket VARCHAR(50) DEFAULT 'NULL')
BEGIN
  UPDATE Sample.Person SET Salary = :Salary
  WHERE Name=:Name ;
END
```

ストアド・プロシージャはパラメータの自動形式変換を実行しません。例えば、ODBC 形式や表示形式の入力パラメータは、その形式を保持します。アプリケーションに適した形式で入出力値を処理して必要な変換を実行するのは、プロシージャを呼び出すコードと、そのプロシージャ・コード自体の責任になります。

メソッドまたはクエリがストアド・プロシージャとして公開されるため、プロシージャ・コンテキスト・ハンドラによって、プロシージャとその呼び出し元間でプロシージャ・コンテキストの相互受け渡しが行われます。ストアド・プロシージャが呼び出されると、クラス `%Library.SQLProcContext` のオブジェクトが `%sqlcontext` 変数でインスタンス化されます。これを使用して、プロシージャとその呼び出し元 (ODBC サーバなど) 間でプロシージャ・コンテキストの受け渡しが行われます。

`%sqlcontext` は、エラー・オブジェクト、SQLCODE エラー・ステータス、SQL 行カウント、およびエラー・メッセージを含む、いくつかのプロパティで構成されます。以下の例は、これらプロパティの設定に使用される値を示しています。

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

SQLCODE と %ROWCOUNT の値は、SQL 文の実行によって自動的に設定されます。`%sqlcontext` オブジェクトは、実行される前に毎回リセットされます。

または、`%SYSTEM.Error` オブジェクトをインスタンス化し、`%sqlcontext.Error` として設定することによって、エラー・コンテキストを設定することができます。

## characteristics

メソッドの生成にはクエリの生成とは異なる `characteristics` を使用します。

指定した `characteristics` の値が無効な場合は、SQLCODE -47 エラーが生成されます。`characteristics` を重複して指定すると、SQLCODE -44 エラーが発生します。

有効なメソッド `characteristics` キーワードは、以下のとおりです。

メソッド・キーワード	意味
FOR className	<p>メソッドを生成するクラス名を指定します。そのクラスが存在しない場合は、新規作成します。メソッド名を修飾することにより、クラス名を指定することもできます。FOR 節で指定されたクラス名の方が、メソッド名の修飾により指定されたクラス名よりも優先されます。</p> <p>FOR my.class 構文を使用してクラス名を指定する場合、InterSystems IRIS は Sqlname=procname を使用してクラス・メソッドを定義します。そのため、メソッドを (my.class_procname()) ではなく my.procname() として呼び出す必要があります。</p>
FINAL	サブクラスがメソッドをオーバーライドできないように指定します。既定では、メソッドは Final ではありません。FINAL キーワードはサブクラスによって継承されます。
PRIVATE	メソッドがそれ自身のクラス、またはサブクラスの他のメソッドによってのみ起動できることを指定します。既定ではメソッドはパブリックで、制限なしに起動できます。この制限はサブクラスによって継承されます。
RESULT SETS DYNAMIC RESULT SETS [n]	作成されるメソッドに <a href="#">ReturnResultsets</a> キーワードが含まれることを指定します。この characteristics 句のすべての形式は同義語です。
RETURNS datatype	メソッドの呼び出しで返される値のデータ型を指定します。RETURNS が省略されると、メソッドは値を返すことができません。この指定内容はサブクラスによって継承され、サブクラスによって変更できます。この datatype には、MINVAL、MAXVAL、SCALE などのタイプのパラメータを指定できます。例えば RETURNS DECIMAL(19,4) のように指定します。値が返されるとき、datatype の長さは無視されます。例えば、RETURNS VARCHAR(32) は、このメソッドの呼び出しによって返されるあらゆる長さの文字列を受け取ることができます。
SELECTMODE mode	LANGUAGE が SQL (既定値) の場合にのみ使用されます。これを指定した場合、InterSystems IRIS は、 <a href="#">#SQLCOMPILE SELECT=mode</a> 文に対応するクラス・メソッドに追加することで、指定された SELECTMODE で、メソッドで定義された SQL 文を生成します。mode に指定できる値は、LOGICAL、ODBC、RUNTIME、および DISPLAY です。既定は LOGICAL です。

有効なクエリ characteristics キーワードは、以下のとおりです。

クエリ・キーワード	概要
CONTAINID integer	フィールドが存在する場合は、どのフィールドが ID を返すかを指定します。CONTAINID を ID を返す列の番号に設定するか、または ID を返す列が存在しない場合は 0 を設定します。InterSystems IRIS では、指定されたフィールドが実際に ID を含んでいるかどうかの検証が行われなため、ユーザの入力の誤りによってデータの不一致が起こる可能性があります。
FOR className	メソッドを生成するクラス名を指定します。そのクラスが存在しない場合は、新規作成します。メソッド名を修飾することにより、クラス名を指定することもできます。FOR 節で指定されたクラス名の方が、メソッド名の修飾により指定されたクラス名よりも優先されます。
FINAL	サブクラスがメソッドをオーバーライドできないように指定します。既定では、メソッドは Final ではありません。FINAL キーワードはサブクラスによって継承されます。



クエリ・キーワード	概要
RESULTS (result_set)	<p>データ・フィールドをクエリで返された順序で指定します。RESULTS 節を指定する場合、クエリによって返されるすべてのフィールドを括弧で囲まれたコンマ区切りリストとしてリストする必要があります。クエリによって返されるよりも少ないか多いフィールドを指定した場合は、カーディナリティの不一致エラー SQLCODE -76 が生成されます。</p> <p>各フィールドについて、列名 (列ヘッダとして使用されます) とデータ型を指定します。</p> <p>LANGUAGE SQL の場合は、RESULTS 節を省略できます。RESULTS 節を省略した場合、ROWSPEC がクラス・コンパイル時に自動生成されます。</p>
SELECTMODE mode	<p>クエリをコンパイルするために使用するモードを指定します。使用可能な値は、LOGICAL、ODBC、RUNTIME、DISPLAY です。既定は RUNTIME です。</p>

SELECTMODE 節は、SELECT クエリ操作、および INSERT と UPDATE 操作で使します。これは、コンパイル時の選択モードを指定します。SELECTMODE に指定した値は、`#sqlcompile select=mode` のように InterSystems ObjectScript のクラス・メソッド・コードの先頭に追加されます。詳細は、“[#sqlcompile select](#)” を参照してください。

- SELECT クエリでは、SELECTMODE はデータを返すモードを指定します。mode 値が LOGICAL の場合は、論理 (内部保存) 値が返されます。例えば、日付は \$HOROLOG 形式で返されます。mode 値が ODBC の場合、論理から ODBC への変換が適用され、ODBC 形式値が返されます。mode 値が DISPLAY の場合、論理から表示への変換が適用され、表示形式値が返されます。mode 値が RUNTIME の場合、実行時に表示モードを (LOGICAL、ODBC、または DISPLAY に) 設定できます。
- [INSERT](#) または [UPDATE](#) 操作では、SELECTMODE RUNTIME オプションで、表示形式 (DISPLAY または ODBC) から論理格納形式への入力データ値の自動変換がサポートされています。このコンパイルされた表示データから論理データへの変換コードは、SQL コード実行時の選択モード設定が LOGICAL (すべての InterSystems SQL 実行インタフェースの既定値) の場合のみ適用されます。

“[ダイナミック SQL の使用法](#)” に説明されているように、SQL コードが実行されると、`%SQL.Statement` クラスの `%SelectMode` プロパティが、実行時の選択モードを指定します。SelectMode オプションの詳細は、“[データ表示オプション](#)” を参照してください。

RESULTS 節は、クエリの結果を指定します。RESULTS 節の SQL データ型パラメータが、クエリの ROWSPEC の対応する InterSystems IRIS データ型パラメータに変換されます。例えば、RESULTS 節の RESULTS ( Code VARCHAR(15) ) では、ROWSPEC = “Code:%Library.String(MAXLEN=15)” という ROWSPEC 仕様が生成されます。

## LANGUAGE

プロシージャ・コード言語を指定するキーワード節です。利用可能なオプションは以下のとおりです。

- LANGUAGE OBJECTSCRIPT (ObjectScript の場合) または LANGUAGE SQL。プロシージャ・コードは code\_body で指定します。
- LANGUAGE JAVA、LANGUAGE PYTHON、または LANGUAGE DOTNET (これらのいずれかの言語で外部ストアド・プロシージャを呼び出す SQL プロシージャの場合)。外部ストアド・プロシージャの構文は、以下のようになります。

```
LANGUAGE langname EXTERNAL NAME external-routine-name
```

langname は JAVA、PYTHON、または DOTNET で、external-routine-name は指定した言語の外部ルーチン名を含む引用符付き文字列です。SQL プロシージャは既存のルーチンを呼び出します。これらの言語のコードを CREATE PROCEDURE 文内に記述することはできません。これらの言語のストアド・プロシージャ・ライブラリは IRIS の外部に

保存されているため、IRIS 内でパッケージ、インポート、コンパイルする必要はありません。以下に、既存の JAVA 外部ストアード・プロシージャを呼び出す CREATE PROCEDURE の例を示します。

```
CREATE PROCEDURE updatePrice (item_name VARCHAR, new_price INTEGER)
LANGUAGE JAVA
EXTERNAL NAME 'Orders.updatePrice'
```

LANGUAGE 節を省略する場合は、SQL が既定です。

### code\_body

生成されるメソッドまたはクエリのプログラム・コード。このコードは SQL または ObjectScript で指定します。使用する言語は LANGUAGE 節と一致させる必要があります。ただし、ObjectScript のコードには埋め込み SQL を記述できます。InterSystems IRIS は、メソッドまたはクエリの実際のコードを生成するために提供されたコードを使用します。

- SQL プログラム・コードの開始には BEGIN キーワードを使用し、その後に SQL コード自体を続けます。完結した各 SQL 文の末尾では、セミコロン (;) を指定します。1 つのクエリに含めることができるのは、1 つの SQL 文、つまり 1 つの SELECT 文のみです。データの挿入、更新または削除を行うプロシージャを作成することもできます。SQL プログラム・コードの終了には、END キーワードを使用します。

入力パラメータは、SQL 文内で :name の形式で [ホスト変数](#) として指定されます (SQL コード内で疑問符 (?) を使用して入力パラメータを指定しないでください。プロシージャは正常に構築されますが、呼び出されたときに、これらのパラメータを渡すことができないか、既定値が使用されます)。

- ObjectScript プログラム・コードは、{ code } のように中括弧で囲みます。コード行はインデントする必要があります。ラベルまたは #include プリプロセッサ・コマンドを指定する場合、そのラベルまたはコマンドは、次の例に示すようにコロンで始まり、1 列目に配置される必要があります。

### SQL

```
CREATE PROCEDURE SP123()
LANGUAGE OBJECTSCRIPT
{
:Top
#include %occConstant
WRITE "Hello World"
IF 0=$RANDOM(2) { GOTO Top }
ELSE {QUIT $$$OK }
}
```

システムによって、%occInclude が自動的に組み込まれます。プログラム・コードに InterSystems IRIS マクロ・プリプロセッサ文 (# コマンド、## 関数、または \$\$\$ マクロ参照) が含まれている場合、これらの文の処理と展開はプロシージャのメソッド定義の一部であり、これらの文はこのメソッドのコンパイル時に処理および展開されます。プリプロセッサ・コマンドの詳細は、"[システム・プリプロセッサ・コマンド・リファレンス](#)" を参照してください。

SQL を ObjectScript “ラップ” に埋め込んだプロシージャを生成するとき、InterSystems IRIS によって、プロシージャ・コンテキスト・ハンドラを提供し、戻り値を処理するコード行が追加されます。以下はこの InterSystems IRIS から生成されたラップ・コードの例です。

### ObjectScript

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql(
-- code_body
)
QUIT $GET(title)
```

指定するコードが OBJECTSCRIPT の場合、“ラップ” (変数を NEW で処理して、QUIT val を使用して完了時に値を返す) を明示的に定義する必要があります。



## 例

以下の例は、SQL code\_body の使用例と、ObjectScript code\_body の使用例で構成されています。

### SQL コードの使用例

以下の例は、ストアド・プロシージャとして公開される、PersonStateSP という名前のシンプルなクエリを作成します。このクエリでは、パラメータは宣言されず、characteristics および LANGUAGE に既定値が使用されます。

#### ObjectScript

```
CREATE PROCEDURE PersonStateSP() BEGIN
    SELECT Name,Home_State FROM Sample.Person ;
END
```

管理ポータルで、[クラス] オプションを選択し、SAMPLES ネームスペースを選択します。上の例で作成したストアド・プロシージャの **User.procPersonStateSP.cls** を見つけます。上のプログラム例を再実行する前に、ここでこのプロシージャを削除できます。もちろん、DROP PROCEDURE を使用してプロシージャを削除できます。

#### ObjectScript

```
DROP PROCEDURE SAMPLES.PersonStateSP)
```

以下の例は、データを更新するプロシージャを作成します。この例は、CREATE PROCEDURE を使用して、Sample.Employee クラスに UpdateSalary メソッドを生成します。

### SQL

```
CREATE PROCEDURE UpdateSalary ( IN SSN VARCHAR(11), IN Salary INTEGER )
FOR Sample.Employee
BEGIN
    UPDATE Sample.Employee SET Salary = :Salary WHERE SSN = :SSN;
END
```

### ObjectScript コードの使用例

以下の例では、ランダムな大文字を生成する RandomLetterSP() ストアド・プロシージャ・メソッドを作成します。その後、このメソッドを関数として SELECT 文内で呼び出すことができます。RandomLetterSP() メソッドを削除するため、DROP PROCEDURE が指定されています。

### SQL

```
CREATE PROCEDURE RandomLetterSP()
RETURNS INTEGER
LANGUAGE OBJECTSCRIPT
{
    :Top
    SET x=$RANDOM(90)
    IF x<65 {GOTO Top}
    ELSE {QUIT $CHAR(x)}
}
```

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetterSP()
```

### SQL

```
DROP PROCEDURE RandomLetterSP
```

以下の CREATE PROCEDURE の例は、Execute()、Fetch()、Close() の各メソッドに対する ObjectScript による呼び出しを使用します。このようなプロシージャには、FetchRows() メソッド呼び出しおよび GetInfo() メソッド呼び出しを含めることも可能です。

## SQL

```
CREATE PROCEDURE GetTitle()
  FOR Sample.Employee
  RESULTS (ID %Integer)
  CONTAINID 1
  LANGUAGE OBJECTSCRIPT
  Execute(INOUT qHandle %Binary)
  { QUIT 1 }
  Fetch(INOUT qHandle %Binary, INOUT Row %List, INOUT AtEnd %Integer)
  { QUIT 1 }
  Close(INOUT qHandle %Binary)
  { QUIT 1 }
```

以下の CREATE PROCEDURE の例は、**%SQL.Statement** 結果セット・クラスに対する ObjectScript による呼び出しを使用します。

## SQL

```
CREATE PROCEDURE Sample_Employee.GetTitle(
  INOUT Title VARCHAR(50) )
  RETURNS VARCHAR(30)
  FOR Sample.Employee
  LANGUAGE OBJECTSCRIPT
  {
    SET myquery="SELECT TOP 10 Name,Title FROM Sample.Employee"
    SET tStatement = ##class(%SQL.Statement).%New()
    SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
    SET rset = tStatement.%Execute()
    DO rset.%Display()
    WRITE !,"End of data"
  }
```

ObjectScript コード・ブロックがデータをローカル変数 (例えば、Row) に取得する場合は、コード・ブロックを SET Row=" 行で終了して、データの終了 (end-of-data) 条件を示す必要があります。

以下の例は、埋め込み SQL を含む ObjectScript コードで CREATE PROCEDURE を使用します。このプロシージャは、Sample.Employee クラスに GetTitle メソッドを生成し、パラメータとして Title 値を渡します。

## SQL

```
CREATE PROCEDURE Sample_Employee.GetTitle(
  IN SSN VARCHAR(11),
  INOUT Title VARCHAR(50) )
  RETURNS VARCHAR(30)
  FOR Sample.Employee
  LANGUAGE OBJECTSCRIPT
  {
    NEW SQLCODE,%ROWCOUNT
    &sql(SELECT Title INTO :Title FROM Sample.Employee
      WHERE SSN = :SSN)
    IF $GET(%sqlcontext)'= " " {
      SET %sqlcontext.%SQLCODE=SQLCODE
      SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
    QUIT
  }
```

**%sqlcontext** オブジェクトを使用します。対応する SQL 変数を使用して、その %SQLCODE プロパティおよび %ROWCOUNT プロパティを設定します。プロシージャの LANGUAGE OBJECTSCRIPT キーワードに続く ObjectScript コードを中括弧で囲むことに注意してください。ObjectScript コードには、&sql でマークされ、角括弧で囲まれた埋め込み SQL コードがあります。

## セキュリティおよび特権

CREATE PROCEDURE コマンドは、ユーザに %Development:USE 権限が必要な特権操作です。このような権限は管理ポータルを介して付与できます。これらの特権なしで CREATE PROCEDURE コマンドを実行すると、SQLCODE -99 エラーが発生し、コマンドは失敗します。

適切な特権を持たないユーザでも、以下の 2 つの状況のいずれかではこのコマンドを実行できます。

- ・ 埋め込み SQL を介してコマンドを実行する場合。この場合は特権が確認されません。
- ・ 特権を確認しないことをユーザが明示的に指定する場合。例えば、checkPriv 引数を 0 に設定して %Prepare() を呼び出すか、**%SQL.Statement** に対して %ExecDirectNoPriv() を呼び出します。

## 関連項目

- ・ [SELECT](#)
- ・ [CALL](#)
- ・ [DROP PROCEDURE](#)
- ・ [CREATE METHOD、CREATE FUNCTION](#)
- ・ [GRANT](#)
- ・ [ストアド・プロシージャの定義と使用](#)
- ・ [データベースの問い合わせ](#)

## CREATE QUERY (SQL)

クエリを作成します。

### 構文

```
CREATE [OR REPLACE] QUERY queryname(parameter_list)
  [ characteristics ]
  [ LANGUAGE SQL ]
  BEGIN code_body ;
END

CREATE QUERY queryname(parameter_list) [characteristics]
  LANGUAGE OBJECTSCRIPT
  { code_body }
```

### 説明

CREATE QUERY 文は、クラスにクエリを生成します。既定では、MySelect と命名したクエリが、**User.queryMySelect** または **SQLUser.queryMySelect** として保存されます。

CREATE QUERY で作成されるクエリは、ストアド・プロシージャとして公開される場合とそうでない場合があります。ストアド・プロシージャとして公開するクエリを生成するには、characteristics の 1 つとして PROCEDURE キーワードを指定する必要があります。[CREATE PROCEDURE](#) 文を使用して、ストアド・プロシージャとして公開されるクエリを生成することも可能です。

クエリの生成には、[GRANT](#) コマンドで指定された %CREATE\_QUERY 管理者特権が必要です。定義された所有者を持つ既存のクラスのクエリを作成しようとする場合、クラスの所有者としてログインする必要があります。そうでない場合、操作は SQLCODE -99 エラーで失敗します。

クラス定義が[導入済みのクラス](#)の場合、クラスでクエリを作成することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “                    classname                    DDL                    ” に設定されます。

### 引数

#### queryname

ストアド・プロシージャ・クラスに作成するクエリの名前。パラメータを指定しない場合でも、queryname は有効な識別子である必要があり、後には括弧を付加する必要があります。このプロシージャ名は、未修飾 (StoreName) にして[既定のスキーマ名](#)を使用しても、スキーマ名を指定して修飾 (Patient.StoreName) してもかまいません。\$SYSTEM.SQL.Schema.Default() メソッドを使用して、現在のシステム全体の既定のスキーマ名を確認できます。初期のシステム全体の既定のスキーマ名は、クラス・パッケージ名 **User** に対応する **SQLUser** です。

FOR 特性 (後述) は、queryname で指定されたクラス名をオーバーライドすることに注意してください。この名前のメソッドが既に存在する場合、この操作は失敗して SQLCODE -361 エラーが発行されます。

生成されるクラスの名前は、スキーマ名に対応するパッケージ名の後にドット、“query”、指定の queryname が順に続いたものになります。例えば、未修飾のクエリ名 RandomLetter で初期の既定のスキーマ SQLUser が使用される場合、生成されるクラス名は User.queryRandomLetter となります。詳細は [“SQL からクラス名への変換”](#) を参照してください。

InterSystems SQL では、大文字/小文字が異なるだけの queryname を指定することは許可されていません。既存のクエリ名と大文字/小文字が異なるだけの queryname を指定した場合は、SQLCODE -400 エラーが生成されます。

指定した queryname が現在のネームスペース内に既に存在する場合は、SQLCODE -361 エラーが生成されます。

オプションのキーワード OR REPLACE を指定して、エラーが発生することなく、既存のクエリを変更または置換します。CREATE OR REPLACE QUERY には、DROP QUERY を呼び出して古いバージョンのクエリを削除し、続いて CREATE QUERY を呼び出す操作と同じ効果があります。

## parameter-list

値をクエリに渡すために使用するパラメータのパラメータ宣言リストです。パラメータのリストは括弧で囲み、リストのパラメータ宣言はコンマで区切ります。パラメータを指定しない場合も括弧は必須です。

リスト内の各パラメータ宣言は、(先頭から順番に) 以下の要素で構成します。

- ・ パラメータ・モードが IN (入力値)、OUT (出力値)、または INOUT (変更値) のいずれであるかを指定するオプションのキーワード。省略した場合、既定のパラメータ・モードは IN です。
- ・ パラメータ名。パラメータ名では、大文字と小文字が区別されます。
- ・ パラメータのデータ型。
- ・ オプション：パラメータの既定値。DEFAULT キーワードの後ろに既定値を付けて指定できます。DEFAULT キーワードはオプションです。既定値が指定されていない場合、既定値は NULL であると見なされます。

以下の例では、入力パラメータが 2 つあるストアード・プロシージャとして公開されるクエリを作成します。これらの入力パラメータのどちらにも既定値があります。topnum 入力パラメータではオプションの DEFAULT キーワードを指定し、minage 入力パラメータではこのキーワードを省略します。

## SQL

```
CREATE QUERY AgeQuery(IN topnum INT DEFAULT 10,IN minage INT 20)
PROCEDURE
BEGIN
SELECT TOP :topnum Name,Age FROM Sample.Person
WHERE Age > :minage ;
END
```

CALL AgeQuery(6,65)、CALL AgeQuery(6)、CALL AgeQuery(,65)、CALL AgeQuery() はすべて、このクエリで有効な CALL 文です。

## characteristics

クエリの特性を指定する 1 つ以上のキーワードを示す引数 (オプション)。特性は任意の順序で指定できます。有効な characteristics キーワードは、以下のとおりです。

characteristics キーワード	概要
CONTAINID integer	フィールドが存在する場合は、どのフィールドが ID を返すかを指定します。CONTAINID を ID を返す列の番号に設定するか、または ID を返す列が存在しない場合は 0 を設定します。InterSystems IRIS では、指定されたフィールドが実際に ID を含んでいるかどうかの検証が行われないため、ユーザの入力の誤りによってデータの不一致が起こる可能性があります。
FOR className	メソッドを生成するクラス名を指定します。そのクラスが存在しない場合は新規作成します。メソッド名を認証することによりクラス名を指定することもできます。FOR 節で指定されたクラス名の方が、メソッド名の修飾により指定されたクラス名よりも優先されます。
FINAL	サブクラスがメソッドをオーバーライドできないように指定します。既定では、メソッドは Final ではありません。FINAL キーワードはサブクラスによって継承されます。
PROCEDURE	クエリが SQL ストアド・プロシージャであることを指定します。ストアード・プロシージャはサブクラスに継承されます(このキーワードは、PROC と略することができます)。

characteristics キーワード	概要
RESULTS (result_set)	<p>データ・フィールドをクエリで返された順序で指定します。RESULTS 節を指定する場合、クエリによって返されるすべてのフィールドを括弧で囲まれたコンマ区切りリストとしてリストする必要があります。クエリによって返されるよりも少ないか多いフィールドを指定した場合は、カーディナリティの不一致エラー SQLCODE -76 が生成されます。</p> <p>各フィールドについて、列名 (列ヘッダとして使用されます) とデータ型を指定します。</p> <p>LANGUAGE SQL の場合は、RESULTS 節を省略できます。RESULTS 節を省略した場合、ROWSPEC がクラス・コンパイル時に自動生成されます。</p>
SELECTMODE mode	<p>クエリをコンパイルするために使用するモードを指定します。可能な値は、LOGICAL、ODBC、RUNTIME、DISPLAY です。既定は RUNTIME です。</p>

クエリに対して有効でないメソッド・キーワード (PRIVATE や RETURNS など) を指定する場合、システムは SQLCODE -47 エラーを発行します。characteristics を重複して指定すると、SQLCODE -44 エラーが発生します。

SELECTMODE 節は、データを返すモードを指定します。mode 値が LOGICAL の場合は、論理 (内部保存) 値が返されます。例えば、日付は \$HOROLOG 形式で返されます。mode 値が ODBC の場合、論理から ODBC への変換が適用され、ODBC 形式値が返されます。mode 値が DISPLAY の場合、論理から表示への変換が適用され、表示形式値が返されます。mode 値が RUNTIME の場合、**%SQL.Statement** クラスの %SelectMode プロパティを設定することによって、実行時にモードを LOGICAL、ODBC、DISPLAY のいずれかに設定できます。これについては、“[ダイナミック SQL の使用法](#)” で説明しています。RUNTIME モードの既定は LOGICAL です。SelectMode オプションの詳細は、“[データ表示オプション](#)” を参照してください。SELECTMODE に指定した値は、#SQLCompile SELECT=mode のように InterSystems ObjectScript クラス・メソッド・コードの最初に追加されます。詳細は、“[#sqlcompile select](#)” を参照してください。

RESULTS 節は、クエリの結果を指定します。RESULTS 節の SQL データ型パラメータが、クエリの ROWSPEC の対応する InterSystems IRIS データ型パラメータに変換されます。例えば、RESULTS 節の RESULTS ( Code VARCHAR(15) ) では、ROWSPEC = "Code:%Library.String(MAXLEN=15)" という ROWSPEC 仕様が生成されます。

## LANGUAGE

code\_body に使用している言語を指定するキーワード節 (オプション)。使用可能な節は、LANGUAGE OBJECTSCRIPT か、LANGUAGE SQL です。LANGUAGE 節を省略する場合は、SQL が既定です。

LANGUAGE が SQL の場合、**%Library.SQLQuery** 型のクラス・クエリが生成されます。LANGUAGE が OBJECTSCRIPT の場合、**%Library.Query** 型のクラス・クエリが生成されます。

### code\_body

生成されるクエリのプログラム・コード。このコードは SQL または ObjectScript で指定します。使用する言語は LANGUAGE 節と一致させる必要があります。ただし、ObjectScript のコードには埋め込み SQL を記述できます。

指定したコードが SQL の場合、1 つの SELECT 文で構成する必要があります。SQL クエリのプログラム・コードは BEGIN キーワードで開始し、その後に実際のプログラム・コード (1 つの SELECT 文) を続ける必要があります。プログラム・コードの末尾にはセミコロン (;) を付け、END キーワードで終了します。

指定したコードが OBJECTSCRIPT の場合、InterSystems IRIS が提供する **%Library.Query** クラスの Execute() クラス・メソッドと Fetch() クラス・メソッドの呼び出しを含める必要があります。また、Close()、FetchRows()、GetInfo() の各メソッド呼び出しを含めることも可能です。ObjectScript コードは中括弧で囲みます。Execute() または Fetch() が見つからない場合は、コンパイル時に SQLCODE -46 エラーが生成されます。

ObjectScript コード・ブロックがデータをローカル変数 (例えば、Row) に取得する場合は、コード・ブロックを SET Row=" " 行で終了して、データの終了 (end-of-data) 条件を示す必要があります。



クエリをストアード・プロシージャとして公開すると (characteristics に PROCEDURE キーワードを指定)、プロシージャ・コンテキスト・ハンドラの使用によって、プロシージャとその呼び出し元間でプロシージャ・コンテキストの相互受け渡しが行われます。

ストアード・プロシージャが呼び出されると、クラス `%Library.SQLProcContext` のオブジェクトが `%sqlcontext` 変数でインスタンス化されます。これを使用して、プロシージャとその呼び出し元 (ODBC サーバなど) 間でプロシージャ・コンテキストの受け渡しが行われます。

`%sqlcontext` は、エラー・オブジェクト、SQLCODE エラー・ステータス、SQL 行カウント、およびエラー・メッセージを含む、いくつかのプロパティで構成されます。以下の例は、これらプロパティの設定に使用される値を示しています。

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=msg
```

SQLCODE と %ROWCOUNT の値は、SQL 文の実行によって自動的に設定されます。`%sqlcontext` オブジェクトは、実行される前に毎回リセットされます。

または、`%SYSTEM.Error` オブジェクトをインスタンス化し、`%sqlcontext.Error` として設定することによって、エラー・コンテキストを設定することができます。

InterSystems IRIS は、クエリの実際のコードを生成するために提供されたコードを使用します。

## 例

以下の例は、DocTestPersonState という名前のクエリを作成します。このクエリでは、パラメータは宣言されず、SELECTMODE characteristic が設定されます。また、LANGUAGE に既定 (SQL) が使用されます。

## SQL

```
CREATE QUERY DocTestPersonState() SELECTMODE RUNTIME
BEGIN
SELECT Name,Home_State FROM Sample.Person ;
END
```

管理ポータルで、[クラス] オプションを選択し、SAMPLES ネームスペースを選択します。上の例で作成したクエリの `User.queryDocTestPersonState.cls` を見つけます。上のプログラム例を再実行する前に、ここでこのクエリを削除できます。もちろん、DROP QUERY を使用してクエリを削除することも可能です。

以下の埋め込み SQL の例は、SQLCODE とその説明のリストを取得する DocTestSQLCODEList という名前のメソッド・ベースのクエリを作成します。このクエリでは、RESULTS 結果セット特性を設定し、LANGUAGE を ObjectScript に設定して、Execute()、Fetch()、Close() の各メソッドを呼び出します。

## ObjectScript

```
&sql(CREATE QUERY DocTestSQLCODEList()
RESULTS (SQLCODE SMALLINT,Description VARCHAR(100))
PROCEDURE
LANGUAGE OBJECTSCRIPT
Execute(INOUT QHandle BINARY(255))
{
SET QHandle=1,%i(QHandle)="
QUIT ##lit($$OK)
}
Fetch(INOUT QHandle BINARY(255), INOUT Row %List, INOUT AtEnd INT)
{
SET AtEnd=0,Row="
SET %i(QHandle)=$o(^%qCacheSQL("SQLCODE",%i(QHandle)))
IF %i(QHandle)=" {SET AtEnd=1 QUIT ##lit($$OK) }
SET Row=$lb(%i(QHandle),^%qCacheSQL("SQLCODE",%i(QHandle),1,1))
QUIT ##lit($$OK)
}
Close(INOUT QHandle BINARY(255))
{
KILL %i(QHandle)
QUIT ##lit($$OK)
}
}
```



```

IF SQLCODE=0 { WRITE !,"Created a query" }
ELSEIF SQLCODE=-361 { WRITE !,"Query exists: ",%msg }
ELSE { WRITE !,"CREATE QUERY error: ",SQLCODE }

```

管理ポータルで、[クラス] オプションを選択し、SAMPLES ネームスペースを選択します。上の例で作成したクエリの **User.queryDocTestSQLCODEList.cls** を見つけます。上のプログラム例を再実行する前に、ここでこのクエリを削除できます。もちろん、DROP QUERY を使用してクエリを削除することも可能です。

以下のダイナミック SQL の例では、DocTest という名前のクエリを作成して、**%SQL.Statement** クラスの **%PrepareClassQuery()** メソッドを使用してこのクエリを実行しています。

### ObjectScript

```

/* Creating the Query */
set myquery=4
set myquery(1)="CREATE QUERY DocTest() SELECTMODE RUNTIME "
set myquery(2)="BEGIN "
set myquery(3)="SELECT TOP 5 Name,Home_State FROM Sample.Person ; "
set myquery(4)="END"
set tStatement = ##class(%SQL.Statement).%New()

set qStatus = tStatement.%Prepare(.myquery)
if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}

set rset = tStatement.%Execute()
if (rset.%SQLCODE '= 0) {write "%Unable to call query", !, "SQLCODE ", rset.%SQLCODE, ": ",
rset.%Message quit}

/* Calling the Query */
write !,"Calling a class query",!
set cqStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
if $$$ISERR(cqStatus) {write "%PrepareClassQuery failed:" do $SYSTEM.Status.DisplayError(cqStatus)
quit}

set rset = tStatement.%Execute()
if (rset.%SQLCODE '= 0) {write "Unable to call class query", !, "SQLCODE ", rset.%SQLCODE, ": ",
rset.%Message quit}

write "Query data",!,!
while rset.%Next()
{
    do rset.%Print()
}
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
write !,"End of data"

/* Deleting the Query */
&sql(DROP QUERY DocTest)
if SQLCODE = 0 {write !,"Deleted the query"}

```

詳細は、“[ダイナミック SQL の使用](#)”を参照してください。

## セキュリティおよび特権

CREATE QUERY コマンドは、ユーザに %Development:USE 権限が必要な特権操作です。このような権限は管理ポータルを介して付与できます。これらの特権なしで CREATE QUERY コマンドを実行すると、SQLCODE -99 エラーが発生し、コマンドは失敗します。

適切な特権を持たないユーザでも、以下の 2 つの状況のいずれかではこのコマンドを実行できます。

- ・ 埋め込み SQL を介してコマンドを実行する場合。この場合は特権が確認されません。
- ・ 特権を確認しないことをユーザが明示的に指定する場合。例えば、checkPriv 引数を 0 に設定して %Prepare() を呼び出すか、**%SQL.Statement** に対して %ExecDirectNoPriv() を呼び出します。

## 関連項目

- ・ [SELECT](#)
- ・ [CALL](#)

- ・ DROP QUERY
- ・ CREATE PROCEDURE
- ・ データベースの問い合わせ
- ・ ストアド・プロシージャの定義と使用

## CREATE ROLE (SQL)

ロールを生成します。

### 構文

```
CREATE ROLE role-name
```

### 説明

CREATE ROLE コマンドはロールを生成します。ロールは複数のユーザに割り当てることができる名前付きの特権セットです。1 つのロールを複数のユーザに割り当てても、また複数のロールを 1 人のユーザに割り当ててもできます。ロールはシステム全体で利用できるもので、特定のネームスペースに限定されません。

role-name は 64 文字までの有効な識別子です。role-name は、[識別子](#)の名前付け規約に従っている必要があります。ロール名には Unicode 文字を使用することができます。ロール名は、大文字と小文字が区別されません。[\[区切り識別子をサポート\]](#) 構成オプションにチェックが付いている場合 (既定)、role-name を、引用符で囲まれた[区切り識別子](#)にすることができます。区切り識別子の場合、role-name に SQL 予約語を使用できます。ピリオド (.)、キャレット (^)、および 2 文字の矢印シーケンス (->) を含めることもできます。コンマ (,) またはコロン (:) 文字を含めることはできません。この名前の先頭の文字には、アスタリスク (\*) を除く任意の有効な文字を使用できます。

最初に作成されたときは、ロールは名前が付いているだけで特権はありません。ロールに特権を付与するには、[GRANT](#) コマンドを使用します。GRANT コマンドを使用すると、1 つのロールに 1 つ以上のロールを割り当ててもできます。この方法によってロールの階層を作成することができます。

CREATE ROLE を実行して既に存在するロールを作成しようとすると、SQL は SQLCODE -118 エラーを返します。`$$SYSTEM.Security.RoleExists()` メソッドを呼び出すことによって、ロールが既に存在するかどうかを判別できます。

#### ObjectScript

```
WRITE $$SYSTEM.Security.RoleExists("%All"),!  
WRITE $$SYSTEM.Security.RoleExists("Madmen")
```

このメソッドは、指定したロールが存在する場合 1 を返し、存在しない場合は 0 を返します。ロール名は、大文字と小文字が区別されません。

ロールの削除には、DROP ROLE コマンドを使用します。

### 特権

CREATE ROLE コマンドの実行には特権が必要です。埋め込み SQL で CREATE ROLE を使用する前に、以下のいずれかを持つユーザとしてログインする必要があります。

- USE 権限がある [%Admin\\_Secure](#) 管理リソース
- USE 権限がある [%Admin\\_RoleEdit](#) 管理リソース
- システムに対する全面的なセキュリティ権限

これらの特権がない場合に CREATE ROLE コマンドを実行すると、SQLCODE -99 エラー (特権違反) が返されます。`$$SYSTEM.Security.Login()` メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

#### ObjectScript

```
DO $$SYSTEM.Security.Login(username,password)  
&sql( )
```

`$$SYSTEM.Security.Login()` メソッドを呼び出すには、[%Service\\_Login:Use](#) 特権が必要です。詳細は、“インターシステムズ・クラス・リファレンス” の “[%SYSTEM.Security](#)” を参照してください。

## 引数

### role-name

生成するロールの名前。[識別子](#)です。ロール名は、大文字と小文字が区別されません。

## 例

以下の例は、BkUser という名前のロールを作成します。最初の例のユーザ "FRED" にはロール作成特権がありません。次の例のユーザ "\_SYSTEM" にはロール作成特権があります。

### ObjectScript

```
DO $SYSTEM.Security.Login("FRED","Fred'sPassword")
&sql(CREATE ROLE BkUser)
IF SQLCODE=-99 {
    WRITE !,"You don't have CREATE ROLE privileges" }
ELSEIF SQLCODE=-118 {
    WRITE !,"The role already exists" }
ELSE {
    WRITE !,"Created a role. Error code is: ",SQLCODE }
```

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
Main
&sql(CREATE ROLE BkUser)
IF SQLCODE=-99 {
    WRITE !,"You don't have CREATE ROLE privileges" }
ELSEIF SQLCODE=-118 {
    WRITE !,"The role already exists" }
ELSE {
    WRITE !,"Created a role. Error code is: ",SQLCODE }
Cleanup
SET toggle=$RANDOM(2)
IF toggle=0 {
    &sql(DROP ROLE BkUser)
    WRITE !,"DROP USER error code: ",SQLCODE
}
ELSE {
    WRITE !,"No drop this time"
    QUIT
}
```

(\$RANDOM トグルが記述されているので、このプログラム例を繰り返し実行できます。)

## 関連項目

- SQL 文 : [DROP ROLE](#)、[CREATE USER](#)、[DROP USER](#)、[GRANT](#)、[REVOKE](#)、[%CHECKPRIV](#)
- SQL のユーザ、ロール、および特権
- SQLCODE エラー・メッセージ
- ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数

## CREATE SCHEMA (SQL)

スキーマを作成します。

### 構文

```
CREATE SCHEMA [ IF NOT EXISTS ] name
```

### 引数

引数	説明
<i>name</i>	作成するスキーマの名前。 <a href="#">識別子</a> の名前です。
IF NOT EXISTS	オプション - <i>name</i> を持つスキーマが既に存在する場合に発生するエラーを抑制します。スキーマは再作成されません。

### 説明

対応する[パッケージ定義](#)と共にスキーマ定義を作成します。スキーマの所有者は、このコマンドを発行するユーザとして定義されます。この方法で作成されたスキーマは、スキーマ内にテーブルが作成されるまで INFORMATION\_SCHEMA.SCHEMATA には表示されません。

IF NOT EXISTS が指定されていて、スキーマが既に存在する場合、このコマンドは何も実行しません。IF NOT EXISTS が指定されていないが、同じ名前のスキーマが既に存在する場合、SQLCODE -476 が返されます。

### 関連項目

- ・ DROP SCHEMA
- ・ [SQLCODE エラー・メッセージ](#)

# CREATE TABLE (SQL)

テーブル定義を作成します。

## 構文

### 基本的なテーブルの作成

```
CREATE TABLE [IF NOT EXISTS] table (column type, column2 type2, ...)
CREATE TABLE [IF NOT EXISTS] table AS SELECT query ...
```

### 列の制約

```
CREATE TABLE table (column type NOT NULL, ...)

CREATE TABLE table (column type UNIQUE, ...)
CREATE TABLE table (UNIQUE (column, column2, ...), ...)
CREATE TABLE table (... , CONSTRAINT uniqueName UNIQUE (column, column2, ...))

CREATE TABLE table (column type PRIMARY KEY, ...)
CREATE TABLE table (... , PRIMARY KEY (column, column2, ...))
CREATE TABLE table (... , CONSTRAINT pKeyName PRIMARY KEY (column, column2, ...))
CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (column) REFERENCES refTable (refColumn))
CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (column, column2, ...) REFERENCES refTable (refColumn, refColumn2, ...))
CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES refTable())
CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... ON UPDATE refAction))
CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... ON DELETE refAction))
CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... NOCHECK))
```

### 特殊な列と列のプロパティ

```
CREATE TABLE table (column type DEFAULT defaultSpec, ...)
CREATE TABLE table (column type COMPUTECODE [OBJECTSCRIPT | PYTHON ] {code}, ...)
CREATE TABLE table (column type COMPUTECODE ... {code} COMPUTEONCHANGE (column, column2, ...), ...)
CREATE TABLE table (column type COMPUTECODE ... {code} CALCULATED, ...)
CREATE TABLE table (column type COMPUTECODE ... {code} TRANSIENT, ...)
CREATE TABLE table (column type ON UPDATE updateSpec, ...)
CREATE TABLE table (column type IDENTITY, ...)
```

### テーブルのオプション

```
CREATE TABLE table ... SHARD
CREATE TABLE table ... SHARD KEY (shardKeyColumn, shardKeyColumn2, ...)
CREATE TABLE table ... SHARD KEY (coshardKeyColumn) COSHARD WITH (coshardTable)
CREATE GLOBAL TEMPORARY TABLE table ...
CREATE TABLE table ... WITH %CLASSPARAMETER pName = pValue, %CLASSPARAMETER pName2 = pValue2, ...
```

## 概要

CREATE TABLE コマンドは、指定された構造のテーブル定義を作成します。CREATE TABLE は、SQL テーブルと対応する InterSystems IRIS® クラスの両方を作成します。詳細は、“[作成したテーブルのクラス定義](#)”を参照してください。

注釈 これらの構文には、互換性目的でのみ解析されオペレーションを伴わないキーワードは記載されていません。このようなキーワードの詳細は、“[互換性のみにサポートされるオプション](#)”を参照してください。

### 基本的なテーブルの作成

列の定義とそのデータ型を指定することで、テーブルを作成できます。または、CREATE TABLE AS SELECT クエリを使用して、列の定義とデータを既存のテーブルからコピーできます。

- CREATE TABLE [IF NOT EXISTS] table (column type, column2 type2, ...) は、それぞれデータ型が指定された、1 つ以上の列を含むテーブルを作成します。

この文は、2 つの列を含むテーブルを作成します。最初の列は、最大 30 文字の文字列値を受け入れます。2 つ目の列は有効な日付を受け入れます。

## SQL

```
CREATE TABLE Sample.Person (
    Name VARCHAR(30),
    DateOfBirth TIMESTAMP)
```

### 例: テーブルの作成および生成

- CREATE TABLE [IF NOT EXISTS] table AS SELECT query は、指定された SELECT クエリに基づいて、既存のテーブルから列の定義および列のデータを新しいテーブルにコピーします。SELECT クエリでは、テーブルまたはビューの任意の組み合わせを指定できます。関連の節を指定することで、STORAGETYPE、%CLASSPARAMETER、または シャード・テーブルを指定することもできます。

以下の文は、Sample.People テーブルのデータの一部から、新しいテーブル Sample.YoungPeople を列指向ストレージ・タイプで作成します。

```
CREATE TABLE Sample.YoungPeople
AS SELECT Name, Age
FROM Sample.People
WHERE Age < 21
WITH STORAGETYPE = COLUMNAR
```

テーブルを作成するとき、IF NOT EXISTS 条件を指定できます。これにより、table が既に存在する場合のエラーを抑制できます。詳細は、[既存のテーブルのチェック](#)方法に関する以下のセクションを参照してください。

## 列の制約

列の制約によって、列に許可される値、列の既定値、および列の値が一意である必要があるかどうかを規定します。また、列の主キーと外部キーの制約も定義できます。列ごとに複数の列制約を、任意の順序で指定できます。列制約を区切るにはスペースを使用します。

### NOT NULL 制約

- CREATE TABLE table (column type NOT NULL, ...) では、指定された列のすべてのレコードに値が定義されている必要があります。つまり、NULL 値にすることはできません。

この文は、いずれの列も null にすることができないテーブルを作成します。

## SQL

```
CREATE TABLE Sample.Person (
    Name VARCHAR(30) NOT NULL,
    DateOfBirth TIMESTAMP NOT NULL)
```

空の文字列 (') は null 値と見なされません。列が NOT NULL 制約で定義されているとしても、文字列を受け入れる列に空文字列を挿入することができます。

NULL データ制約キーワード (NOT なし) は、この列が null 値を受け入れることを明示的に指定します。これは列に対する既定の定義です。

### DEFAULT 制約

- CREATE TABLE table (column type DEFAULT defaultSpec, ...) は、この列のデータ値が INSERT で指定されていない場合に、INSERT の処理で自動的にこの列に指定する既定のデータ値を指定します。DEFAULT 値と NOT NULL 制約の両方が指定されている列に INSERT 処理で NULL 値の挿入を指定すると、その列には DEFAULT 値が使用されます。この列で NOT NULL 制約が定義されていない場合は、DEFAULT 値ではなく NULL 値が使用されます。

この文は、MembershipStatus 列および MembershipTerm 列に既定値を設定します。



## SQL

```
CREATE TABLE Sample.Member (
    MemberId INT NOT NULL,
    MembershipStatus CHAR(13) NOT NULL DEFAULT 'M',
    MembershipTerm INT NOT NULL DEFAULT 2)
```

## UNIQUE 制約

UNIQUE 制約が指定されている場合は、一意の値のみを列に指定できます。どの列に UNIQUE 制約が設定されているかを確認するには、“[テーブルのカatalogの詳細](#)”を参照してください。

- CREATE TABLE table (column type UNIQUE, ...) は、指定された列が一意の値のみを受け入れるよう制約を加えます。この列には同じ値を持つレコードは存在しないことになります。

この文は、UserName 列に対して UNIQUE 制約を設定します。

## SQL

```
CREATE TABLE Sample.People (
    UserName VARCHAR(30) UNIQUE NOT NULL,
    FirstName VARCHAR(30),
    LastName VARCHAR(30))
```

SQL では、[空文字列](#) (') はデータ値と見なされるため、UNIQUE データ制約が適用されている場合は、複数のレコードでこの列を空文字列値にすることはできません。[NULL](#) はデータ値と見なされないため、UNIQUE データ制約は複数の NULL に対して適用されません。列に対して NULL を制限するには、NOT NULL キーワード制約を使用します。

**注釈** [シャード・テーブル](#)では、UNIQUE 制約によって挿入および更新のパフォーマンス・コストが大幅に増大します。挿入または更新のパフォーマンスが重要である場合、この制約を使用しないか、テーブルのシャード・キーを含めてください。シャード・テーブルには、UNIQUE 制約に関する追加の制限事項があることに注意してください。

クエリ・パフォーマンスの詳細は、“[一意制約の評価](#)” および “[シャード・クラスタにおけるクエリ](#)”を参照してください。

- CREATE TABLE table (UNIQUE (column, column2, ...), ...) では、列の指定されたグループのすべての値を 1 つに連結した場合にその値が一意の値であることが求められます。個々の列が一意である必要はありません。定義する列のコンマ区切りリストで、任意の場所にこの制約を指定できます。

この文では、作成されたテーブル内の FirstName と LastName のレコードが一意であることが必要ですが、FirstName と LastName のレコードにそれぞれ重複が含まれていてもかまいません。

## SQL

```
CREATE TABLE Sample.People (
    FirstName VARCHAR(30),
    LastName VARCHAR(30),
    UNIQUE (FirstName,LastName))
```

- CREATE TABLE table (... , CONSTRAINT [uniqueName](#) UNIQUE (column,column2, ...)) は、UNIQUE 制約の名前を指定します。テーブル定義から UNIQUE 制約を削除する場合は、[ALTER TABLE](#) コマンドでこの制約の名前が必要になります。

この文は、前の文と機能的に同等であり、制約に FirstLast という名前を付けます。

## SQL

```
CREATE TABLE Sample.People (
    FirstName VARCHAR(30),
    LastName VARCHAR(30),
    CONSTRAINT FirstLast UNIQUE (FirstName,LastName))
```

## PRIMARY KEY 制約

PRIMARY KEY 制約は、1 つの列または列の組み合わせを主キーとして指定し、列が一意であり、null でないように制約を加えます。主キーの定義はオプションです。テーブルを定義する際、InterSystems IRIS は生成される列 RowID 列 (既定名 "ID") を自動的に作成します。これは一意の行識別子として機能します。主キーの詳細は、"[主キーの定義](#)" を参照してください。

- CREATE TABLE table (column type PRIMARY KEY, ...) は、テーブル内の 1 つの列を主キーとして指定し、その列の値が一意で、null ではないように制約します。

この文は、EmpNum 列を主キーとして指定するテーブルを作成します。

### SQL

```
CREATE TABLE Sample.Employee (
    EmpNum INT PRIMARY KEY,
    NameLast CHAR (30) NOT NULL,
    NameFirst CHAR (30) NOT NULL,
    StartDate TIMESTAMP,
    Salary MONEY)
```

管理ポータル の [\[カタログの詳細\]](#) セクションでは、生成される主キーの名前は tablePKeyN の形式を取ります。ここで、table はテーブルの名前で、N は制約カウント整数です。

- CREATE TABLE table (... PRIMARY KEY (column, column2, ...)) は、1 つ以上の列を主キーとして指定します。列のコンマ区切りのリストにおいて、任意の場所で PRIMARY KEY 節を指定できます。この節で単一の列を指定することは、前の構文を使用して特定の列でこの節を指定することと機能的に同等です。この節でコンマ区切りリストの列を指定する場合、各列は NULL ではないと定義されますが、列の値の組み合わせが一意の値である限り、重複する値を含めることができます。

この文は、FirstName 列と LastName 列の組み合わせを主キーとして指定します。

### SQL

```
CREATE TABLE Sample.People (
    FirstName VARCHAR(30),
    LastName VARCHAR(30),
    PRIMARY KEY (FirstName, LastName))
```

- CREATE TABLE table (... CONSTRAINT pKeyName PRIMARY KEY (column, column2, ...)) では、主キーの名前を明示的に付けることができます。主キーの名前は、管理ポータル の [\[カタログの詳細\]](#) セクションから表示できます。

この文は、最初の PRIMARY KEY 構文と機能的に同等で、主キーに EmployeePK という名前を付けます。

### SQL

```
CREATE TABLE Sample.Employee (
    EmpNum INT,
    NameLast CHAR (30) NOT NULL,
    NameFirst CHAR (30) NOT NULL,
    StartDate TIMESTAMP,
    Salary MONEY,
    CONSTRAINT EmployeePK PRIMARY KEY (EmpNum))
```

## 外部キー制約

FOREIGN KEY 制約は、1 つの列または列の組み合わせを、別のテーブルへの参照として指定します。外部キー列に格納された値は、その他のテーブル内のレコードを一意に識別します。テーブルごとに複数の外部キーを指定できます。各外部キー参照は、参照されるテーブル内に常に存在し、一意として定義される必要があります。参照される列は重複値または NULL を持つことはできません。外部キーの詳細は、"[外部キーの定義](#)" を参照してください。

- CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (column) REFERENCES refTable (refColumn)) は、refTable 参照テーブルの refColumn 列を参照する外部キーとして作成されたテーブルから、列を指定します。外部キー列および参照される列は異なる名前でもかまいませんが、データ型と列の制約は同じでなければなりません。fKeyName は外部キーの名前を指定し、必須の要素です。

この文は、CustomersFK という名前の外部キーを定義する Orders テーブルを作成します。この外部キーでは、CustomerNum 列の値は、Customers テーブルの CustID 列で指定された ID になります。

## SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID))
```

- CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (column, column2, ...) REFERENCES refTable (refColumn, refColumn2, ...)) は、列の組み合わせを、参照される列の外部キーとして指定します。外部キー列および参照される列は、列の数およびリストでの順序が一致している必要があります。

この文は、Orders の CustomerNum 列と SalesPersonNum 列の組み合わせを外部キーとして指定します。これらの列の値は、Customers テーブルの対応する CustID 列と SalespID 列を参照します。

## SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    SalesPersonNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum,SalesPersonNum) REFERENCES Customers
(CustID,SalespID))
```

- CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES refTable)) は、参照列の名前を省略します。列または列の組み合わせの外部キーは、既定で参照テーブルの主キーになります (定義されている場合)。そうでない場合は、参照テーブルの IDENTITY 列 (定義されている場合)、あるいは参照テーブルの RowID 列になります。

この文は、CustomerNum 列が Customers テーブルの主キーを参照する外部キーを設定します。この場合は、このテーブルに主キーが定義済みであることを前提とします。

## SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    SalesPersonNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers)
```

- CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... ON UPDATE refAction)) は、参照テーブルに対する UPDATE ルールを定義します。参照テーブルの行の主キーの値を変更しようとする場合は、ON UPDATE 節で、そのテーブルの行に対して実行されるアクションが定義されます。有効な参照アクションの値は、NO ACTION (既定)、SET DEFAULT、SET NULL、および CASCADE です。この節は ON DELETE 節と組み合わせて指定できます。

この文は、参照列 CustID の更新時に、外部キー列 CustomerNum が同じ更新を受け取るテーブルを作成します。

## SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID)
    ON UPDATE CASCADE)
```

- ・ CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... ON DELETE refAction)) は、参照テーブルに対する DELETE ルールを定義します。参照テーブルから行を削除しようとする場合は、ON DELETE 節で、そのテーブル内の行に対して実行されるアクションが定義されます。有効な参照アクションの値は、NO ACTION (既定)、SET DEFAULT、SET NULL、および CASCADE です。この節は ON UPDATE 節と組み合わせて指定できます。

この文は、外部キー列に参照列の値の更新をカスケードするテーブルを作成しますが、参照列の値が削除された場合、対応する外部キーの値は NULL に設定されます。

## SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID)
    ON UPDATE CASCADE
    ON DELETE SET NULL)
```

- ・ CREATE TABLE table (... , CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... NOCHECK)) は、外部キーの参照整合性のチェックを無効にします。つまり、INSERT または UPDATE 処理により、参照テーブル内の行に対応しない値が外部キー列に指定される可能性があります。

NOCHECK キーワードも、外部キーに対する ON DELETE または ON UPDATE 参照アクションが指定されている場合に、これらのアクションの実行を妨げます。SQL クエリ・プロセッサでは、外部キーを使用してテーブルでの結合を最適化できます。ただし、外部キーが NOCHECK として定義されている場合、SQL クエリ・プロセッサはそれを定義済みと見なしません。NOCHECK 外部キーは、引き続きデータベース・ドライバ・カタログ・クエリに対して外部キーとして報告されます。詳細は、“[外部キーの使用法](#)”を参照してください。

## 特殊な列と列のプロパティ

## 計算列

これらの構文は、ユーザ指定ではなく、INSERT または UPDATE 時に計算される列を定義する方法を示しています。これらの列の詳細は、“[INSERT または UPDATE 時の計算フィールドの値](#)”を参照してください。

- ・ CREATE TABLE table (column type COMPUTECODE [OBJECTSCRIPT | PYTHON] {code}, ...) は、指定された ObjectScript コードまたは Python コードを使用して、INSERT 時に値が計算および格納される列を定義します。OBJECTSCRIPT キーワードも PYTHON キーワードも指定しないと、コードは既定で ObjectScript になります。計算済みの列の値は、UPDATE コマンドやトリガ・コード操作など、それ以降のテーブルの更新によって変更されることはありません。

以下の文は、行が挿入されると DOB 列で指定された日付に基づいて Age 列を計算するテーブルを作成します。

## SQL/ObjectScript

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB DATE,
    Age VARCHAR(12) COMPUTECODE {
        set bdate = $zdate({DOB}, 8)
        set today = $zdate($horolog,8)
        set {Age} = $select(bdate = ":", 1:(today - bdate) \ 10000)},
    Grade INT)
```

## SQL/Python

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB DATE,
    Age VARCHAR(12) COMPUTECODE PYTHON {
        import datetime as d
        iris_date_offset = d.date(1840,12,31).toordinal()
        bdate = d.date.fromordinal(cols.getfield('DOB') + iris_date_offset).strftime("%Y%m%d")
        today = d.date.today().strftime("%Y%m%d")
        return str((int(today) - int(bdate)) // 10000) if bdate else ""},
    Grade INT)
```

- CREATE TABLE table (column type COMPUTECODE ... {code} COMPUTEONCHANGE (column, column2, ...), ...) は、COMPUTEONCHANGE 節で指定されたテーブル列のいずれかが以降のテーブル更新で変更されると、計算済みの列の値を再計算します。再計算された値は、以前に保存された値を置き換えます。COMPUTEONCHANGE で指定された列がテーブル仕様に含まれていない場合は、InterSystems SQL で SQLCODE -31 エラーが生成されます。

以下の文は、DOB 列が更新されると Age 列を再計算します。Birthday 列には、列が前回変更された時点のタイムスタンプが含まれます。

## SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(20) NOT NULL,
    DOB TIMESTAMP,
    Birthday VARCHAR(40) COMPUTECODE {
        set {Birthday} = $zdate({DOB})
        _" changed: "_$zdatetime($ztimestamp) }
    COMPUTEONCHANGE (DOB))
```

COMPUTEONCHANGE は、列定義に対応するクラス・プロパティのために [SqlComputeOnChange](#) キーワードを %%UPDATE 値と共に定義します。このプロパティ値は、最初に INSERT 処理の一部として計算され、UPDATE 処理の間に再計算されます。対応する永続クラスの定義の詳細は、“[永続クラスの作成によるテーブルの定義](#)”を参照してください。

- CREATE TABLE table (column type COMPUTECODE ... {code} CALCULATED, ...) は、列の値がデータベースに格納されておらず、列が照会されるたびに生成されるように指定します。計算列によりデータ・ストレージのサイズが縮小しますが、クエリのパフォーマンスが低下する可能性があります。

この列は、列定義に対応するクラス・プロパティの [Calculated](#) ブーリアン・キーワードを定義します。CALCULATED のプロパティには、そのプロパティが [SQLComputed](#) でない場合はインデックスを付けることはできません。

この文は DaysToBirthday 列の値を計算します。この値は現在の日付に応じて変化します。{\*} コードは、計算される列を指定する構文で、この場合は DaysToBirthday になります。

## SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(20) NOT NULL,
    DOB TIMESTAMP,
    DaysToBirthday INT COMPUTECODE {
        set {*} = $zdate({DOB},14) - $zdate($horolog,14) } CALCULATED)
```

- CREATE TABLE table (column type COMPUTECODE ... {code} TRANSIENT, ...) は CALCULATED に似ていますが、列がデータベースに保存されないことも指定します。

この列は、列定義に対応するクラス・プロパティの [Transient](#) ブーリアン・キーワードを定義します。TRANSIENT のプロパティにインデックスを付けることはできません。

CALCULATED および TRANSIENT キーワードは相互に排他的で、同様の動作を提供します。TRANSIENT ではプロパティが保存されません。CALCULATED ではプロパティに対してインスタンス・メモリが割り当てられません。そのため、CALCULATED が指定されるとTRANSIENT が暗黙的に設定されます。

- CREATE TABLE table (column type ON UPDATE updateSpec, ...) は、updateSpec によって指定される値に基づいて、行がテーブル内で更新される場合は常に再計算される列を定義します。列に COMPUTECODE データ制約も存在する場合、ON UPDATE 節を指定することはできません。

この文は、LastUpdated 列を含むテーブルを作成します。この列の値は、対応する行が更新される場合に常に現在の時刻に更新されます。テーブルに格納されたタイムスタンプの値は、2 桁の精度を持ちます。

```
CREATE TABLE MyStudents (
    Name VARCHAR(20) NOT NULL,
    DOB TIMESTAMP,
    LastUpdated TIMESTAMP DEFAULT CURRENT_TIMESTAMP(2) ON UPDATE CURRENT_TIMESTAMP(2))
```

- CREATE TABLE table (column type IDENTITY, ...) は、システムで生成された整数値を持つ RowID 列を、指定された名前の列で置き換えます。

RowID 列と同様に、この列は一意のシステム生成された整数値を持つ単一の列の IDKEY インデックスとして動作します。ここで、各値は対応するテーブル行の一意のレコード ID として機能します。IDENTITY 列を定義することにより、[IDKEY としての主キー](#)の定義が回避されます。テーブルごとに 1 つの IDENTITY 列のみを定義できます。type は整数データ型である必要があります。type を省略すると、データ型は BIGINT として定義されます。IDENTITY 値はユーザ指定にすることができず、UPDATE 文で変更することもできません。

この値は IdNum 列を IDKEY として設定します。この列は SELECT \* などの選択クエリの一部として返されます。

## SQL

```
CREATE TABLE Employee (
    EmpNum INT NOT NULL,
    IdNum IDENTITY NOT NULL,
    Name CHAR(30) NOT NULL,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum))
```

IDENTITY 列の操作の詳細は、"[IDENTITY キーワードを使用した名前付き RowId 列の作成](#)" を参照してください。

## カウンタ列

InterSystems SQL は、3 つのタイプのシステム生成の整数カウンタ列を提供します。これらの列は相互排他的ではなく、同じテーブル内で同時に指定できます。3 つの列すべてのデータ型が、%Library.BigInt データ型クラスにマッピングしています。



カウンタ・タイプ	カウンタの範囲	自動インクリメントの条件	カウントが発生するユーザ指定値	ユーザ指定値	重複値	このタイプの列	カウンタ・リセット条件	シャード・テーブルのサポート
<b>AUTO_INCREMENT</b>	テーブル単位	INSERT	NULL または 0	可能。 システム・カウンタには影響しない。	可能	テーブルごとに 1 つ	TRUNCATE TABLE	あり
<b>SERIAL</b>	シリアル・カウンタ列単位	INSERT	NULL または 0	可能。 システム・カウンタをインクリメントする場合がある。	可能	テーブルごとに複数	TRUNCATE TABLE	なし
<b>ROWVERSION</b>	ネームスペース全体	INSERT または UPDATE	なし	不可	不可	テーブルごとに 1 つ	リセットなし	なし

これらのカウンタ列の詳細は、“[RowVersion](#)、[AutoIncrement](#)、および [Serial カウンタ・フィールド](#)” を参照してください。

- CREATE TABLE table (column type AUTO\_INCREMENT, ...) は、テーブルに挿入が行われるたびにインクリメントするカウンタ列を作成します。テーブルごとに 1 つの AUTO\_INCREMENT カウンタ列のみを指定できます。明示的な整数データ型の後、AUTO\_INCREMENT キーワードを設定する必要があります。例：

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    AutoInc BIGINT AUTO_INCREMENT)
```

または、`%Library.AutoIncrement` データ型を使用して AUTO\_INCREMENT 列を定義できます。したがって、以下は有効な列定義構文です：`MyAutoInc %AutoIncrement, MyAutoInc %AutoIncrement AUTO_INCREMENT`

- CREATE TABLE table (column SERIAL, ...) は、テーブルに対して INSERT 処理が行われるたびにインクリメントするカウンタ列を作成します。SERIAL カウンタ列として複数の列を指定できます。明示的なデータ型の後に SERIAL キーワードを指定します。例：

## SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    Counter SERIAL)
```

- CREATE TABLE table (column ROWVERSION, ...) は、ネームスペース内のすべてのテーブルで INSERT または UPDATE 処理が行われるたびにインクリメントするカウンタ列を作成します。明示的なデータ型の後に ROWVERSION キーワードを指定します。例：



## SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    RowVer ROWVERSION)
```

## %DESCRIPTION キーワード

- CREATE TABLE table (... , %DESCRIPTION [description](#)) は、作成されるテーブルの説明を指定します。説明のテキスト文字列を引用符で囲みます。例：

## SQL

```
CREATE TABLE Employee (
    %Description 'Employees at XYZ Inc.',
    EmpNum INT PRIMARY KEY,
    NameLast VARCHAR(30) NOT NULL,
    NameFirst VARCHAR(30) NOT NULL,
    StartDate TIMESTAMP)
```

- CREATE TABLE table (column type %DESCRIPTION description, ...) は、列の説明を指定します。列ごとに 1 つの説明を指定できます。説明のテキスト文字列を引用符で囲みます。例：

## SQL

```
CREATE TABLE Employee (
    EmpNum INT PRIMARY KEY,
    NameLast VARCHAR(30) NOT NULL,
    NameFirst VARCHAR(30) NOT NULL,
    StartDate TIMESTAMP %Description 'Format: MM/DD/YYYY')
```

## %PUBLICROWID キーワード

- CREATE TABLE table (%PUBLICROWID, ...) は、一意の、システム生成された整数値を持つ [RowID](#) 列をパブリックにします。例：

## SQL

```
CREATE TABLE Employee (
    %PUBLICROWID,
    EmpNum INT PRIMARY KEY,
    NameLast VARCHAR(30) NOT NULL,
    NameFirst VARCHAR(30) NOT NULL,
    StartDate TIMESTAMP)
```

この列は “ID” と名付けられ、列 1 に割り当てられます。テーブルに対応するクラスは “Not SqlRowIdPrivate” で定義されます。ALTER TABLE を使用して %PUBLICROWID を指定することはできません。

RowID がパブリックの場合、以下のようになります。

- RowID 値は [SELECT \\*](#) によって表示されます。
- RowID は、[外部キー参照](#)として使用できます。
- 主キーが定義されていない場合、RowID は[制約名](#)が RowIDField\_As\_PKey の [制約](#)として扱われます。
- コピーされる列名を指定しなければ、テーブルを使用して[データを重複テーブルにコピーする](#)ことはできません。

RowID 列の詳細は、“[RowID は非表示か](#)” を参照してください。

## 照合プロパティ

- CREATE TABLE table (column type COLLATE [sqlCollation](#), ...) は、指定された列で値を並べて比較するために使用される照合タイプを設定します。有効な照合の値は、%EXACT、%MINUS、%PLUS、%SPACE、%SQL-STRING、%SQLUPPER、%TRUNCATE、および %MVR です。

この文は、UserName 列を大文字と小文字を区別する文字列としてソートし、列内の NULL および数値を文字列の文字として処理します。

## SQL

```
CREATE TABLE Sample.People (
    UserName VARCHAR(30) COLLATE %SQLSTRING,
    FirstName VARCHAR(30),
    LastName VARCHAR(30))
```

## テーブルのオプション

### シャード・テーブル

これらの構文はシャード・テーブルを定義するオプションを提供します。このテーブルでは、その列の 1 つをシャード・キーとして使用することで、テーブル行は自動的にデータ・ノード間で水平方向に分割されます。シャード・テーブルにより、特に多数の行がテーブルに格納されている場合に、テーブルに対するクエリのパフォーマンスが向上します。大きなテーブルを結合するクエリで高いパフォーマンスを得るには、各テーブルをコシャードします。コシャードでは、各テーブルからシャード・キー値が同じ行を 1 つのデータ・ノードにまとめることでテーブルの行を分割します。シャードの詳細は、"[シャードディングによるデータ量に応じた水平方向の拡張](#)" を参照してください。

注釈 すべての CREATE TABLE 構文でシャード・テーブルがサポートされるわけではありません。詳細は、"[シャード・テーブルの制限事項](#)" を参照してください。

- CREATE TABLE table ... SHARD は、シャード・テーブルを定義し、RowID 列をシャード・キーとして使用します。これは、システム割り当てシャード・キー (SASK) として知られています。テーブルに [IDENTITY 列](#) が定義されていて、明示的なシャード・キーがない場合、InterSystems SQL では代わりに IDENTITY 列を SASK として使用します。SASK を使用することは、テーブルをシャードディングするための最も簡単で効果的な方法です。列定義の後に SHARD キーワードを指定します。

この文は、既定の RowID 列をシャード・キーとして使用するシャード・テーブルを作成します。

```
CREATE TABLE Vehicle (
    Make VARCHAR(30) NOT NULL,
    Model VARCHAR(20) NOT NULL,
    Year INT NOT NULL,
    Vin CHAR(17) NOT NULL)
SHARD
```

- CREATE TABLE table ... SHARD KEY ([shardKeyColumn](#), shardKeyColumn2, ...) は、1 つの列、または列のコンマ区切りのリストをシャード・キーとして使用するよう指定します。これは、ユーザ定義のシャード・キー (UDSK) として知られています。

この文は、2 つの列で構成されたシャード・キーを持つテーブルを作成します。

```
CREATE TABLE Car (
    Owner VARCHAR(30) NOT NULL,
    Plate VARCHAR(10) NOT NULL,
    State CHAR(2) NOT NULL)
SHARD KEY (Plate, State)
```

この構文を使用して、定義している 2 つ以上のテーブルをコシャードすることもできます。コシャードしたテーブルの UDSK 列の結合は、UDSK ではない列の結合よりもはるかに効率的に動作します。したがって、頻繁に使用する結合列のセットには UDSK を定義することをお勧めします。

これらの文は、列 `Vin` および `VehicleNumber` にシャード・キーを定義した 2 つのテーブルを作成します。

```
CREATE TABLE Vehicle (
    Make VARCHAR(30) NOT NULL,
    Model VARCHAR(20) NOT NULL,
    Year INT NOT NULL,
    Vin CHAR(17) NOT NULL)
SHARD KEY (Vin)

CREATE TABLE Citation (
    CitationID VARCHAR(8) NOT NULL,
    Date TIMESTAMP NOT NULL,
    LicenseNumber VARCHAR(12) NOT NULL,
    Plate VARCHAR(10) NOT NULL,
    VehicleNumber CHAR(17) NOT NULL)
SHARD KEY (VehicleNumber)
```

このようなテーブルでは、UDSK 列を結合するときのパフォーマンスが向上します。例えば、以下のように車両に関連付けられた交通違反通知を返すクエリが考えられます。

## SQL

```
SELECT * FROM Citation, Vehicle WHERE Citation.VehicleNumber = Vehicle.Vin
```

シャード・キーの選択の詳細は、“[シャード・キーの選択](#)”を参照してください。

- CREATE TABLE table ... SHARD KEY ([coshardKeyColumn](#)) COSHARD WITH ([coshardTable](#)) はテーブルを作成し、その整数値列の 1 つをシャード・キー `coshardKeyColumn` として設定します。結合のこのシャード・キーを、別のシャード済みテーブル `coshardTable` で使用できます。このテーブルには、SHARD 構文を使用してシステム割り当てシャード・キー (SASK) を定義する必要があります。必要に応じて、`coshardKeyColumn` を `coshardTable` の SASK 列への外部参照として定義することもできます。外部キー参照を使用して参照整合性を適用できます。同時に、シャード・キーを利用することにより、他のテーブルの SASK 列とのマッチングによるそのテーブルとの結合で優れたパフォーマンスが得られます。

以下の文は、`CustomerID` 列をシャード・キーとして定義したテーブルを作成します。このキーは、既存のシャード済みテーブルである `Customer` のシャード・キーを使用したコシャード結合に使用します。

```
CREATE TABLE Order (
    Date TIMESTAMP NOT NULL,
    Amount DECIMAL(10,2) NOT NULL,
    CustomerID CUSTOMER NOT NULL)
SHARD KEY (CustomerID) COSHARD WITH Customer
```

## 一時テーブル

- CREATE GLOBAL TEMPORARY TABLE table ... は、テーブルをグローバル一時テーブルとして作成します。ここで、テーブル定義はすべてのプロセスで利用可能ですが、テーブル・データ (ストリーム・データを含む) およびインデックスは、テーブルを作成したプロセスの間のみ維持されます。このデータはプロセス・プライベート・グローバルに格納され、プロセスの終了時に削除されます。

この文は、以下のように一時テーブルを作成します。

## SQL

```
CREATE GLOBAL TEMPORARY TABLE TempEmp (
    EmpNum INT NOT NULL,
    NameLast CHAR(30) NOT NULL,
    NameFirst CHAR(30) NOT NULL,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum))
```

一時テーブルを作成するプロセスに関係なく、一時テーブルの所有者は自動的に `_PUBLIC` に設定されています。つまり、キャッシュされた一時テーブル定義にはすべてのユーザがアクセスできます。例えば、あるストア・プロシージャによって一時テーブルが作成された場合、そのストア・プロシージャを呼び出す許可が与えられているユーザであれば、誰でもこの一時テーブルにアクセスできます。これは一時テーブル定義にのみ適用されます。一時テ

ブル・データは呼び出しに固有のものです。したがって、一時テーブル・データにアクセスできるのは、現在のユーザ・プロセスのみです。

グローバル一時テーブルのテーブル定義は、ベース・テーブルと同じです。グローバル一時テーブルには一意の名前を付ける必要があります。既存のベース・テーブルと同じ名前を付けようとすると、SQLCODE -201 エラーが返されます。このテーブルは、明示的に削除 (DROP TABLE を使用) されるまで残り続けます。ALTER TABLE を使用してテーブル定義を変更できます。

グローバル一時テーブルは、DDL 文によってのみ定義できます。

InterSystems IRIS の標準的なテーブルと同様、ClassType=persistent ですが、このクラスには Final キーワードが含まれているため、このクラスにサブクラスを作成することはできません。

## テーブル・ストレージ

- CREATE TABLE table ... WITH STORAGETYPE = [ROW | COLUMNAR] は、基盤となるデータをテーブルに格納するために使用するレイアウトを指定します。
  - ROW を指定すると、データが行単位で格納されます。行ストレージを使用すると効率的なトランザクションが実現します。例えば、オンライン・トランザクション処理 (OLTP) のワークフローで頻繁に行を更新または挿入する場合です。WITH STORAGETYPE 節を省略すると、作成したテーブルは既定で行ストレージになります。
  - COLUMNAR を指定すると、データが列単位で格納されます。列指向ストレージを使用すると効率的なクエリが実現します。例えば、オンライン分析処理 (OLAP) のワークフローで特定列のデータをフィルタ処理する場合や集計する場合です。列指向ストレージは、2022.2 での試験的機能です。InterSystems IRIS の以前のバージョンでは、すべてのテーブル・データが行単位で格納されます。

注釈 照合タイプを明示的に指定しない限り、パフォーマンス上の理由から、列指向ストレージのレイアウトでは既定で EXACT 照合が使用されます。行ストレージのレイアウトでは、ネームスペースの既定である SQLUPPER 照合が使用されます。

ストレージ・レイアウト選択の詳細は、“[SQL テーブルのストレージ・レイアウトの選択](#)”を参照してください。

以下の文は、列指向ストレージ・レイアウトによるテーブルを作成します。

```
CREATE TABLE Sample.TransactionHistory (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))
WITH STORAGETYPE = COLUMNAR
```

以下の文は、CREATE TABLE AS SELECT 節を使用することで、列指向ストレージによるテーブルを作成します。

```
CREATE TABLE Sample.TransactionHistory AS
  SELECT AccountNumber, TransactionDate, Description, Amount, Type
  FROM Sample.BankTransaction
WITH STORAGETYPE = COLUMNAR
```

Tip ヒン CREATE TABLE AS SELECT を使用すれば、それぞれ別々のストレージ・タイプによる複数のテーブルを作成し、そのパフォーマンスを比較して実験できます。

- CREATE TABLE table (column type ... WITH STORAGETYPE = [ROW | COLUMNAR], ...) は、列単位で行ストレージまたは列指向ストレージを指定します。他のすべての列には、既定の行ストレージが設定されるか、テーブル定義の最後に WITH STORAGETYPE 節で指定したストレージ・タイプが設定されます。

以下の文は、Amount 列のみを列指向ストレージ・レイアウトとしたテーブルを作成します。この文では、テーブル定義の最後に WITH STORAGETYPE 節がないので、残りの列は既定の行ストレージ・レイアウトになります。

```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2) WITH STORAGETYPE = COLUMNAR,
  Type VARCHAR(10))
```

## クラス・パラメータ

- CREATE TABLE table ... WITH %CLASSPARAMETER pName = pValue, %CLASSPARAMETER pName2 = pValue2, ... は、作成されるテーブルの主要な側面を定義する、1 つ以上の名前と値の組み合わせである %CLASSPARAMETER を指定します。各パラメータ名 pName は、指定された値である pValue に設定されます。

この文では、USEEXTENTSET クラス・パラメータは、`^EPgS.D8T6.1` のような生成されるグローバル名の使用を無効にします。このようなグローバルは、データに対する IDKEY インデックスとして使用されます。DEFAULTGLOBAL クラス・パラメータは、`^GL.EMPLOYEE` を **インデックスの明示的なグローバル名** として指定します。

```
CREATE TABLE Employees (
  EmpNum INT NOT NULL,
  NameLast CHAR(30) NOT NULL,
  NameFirst CHAR(30) NOT NULL,
  CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum)
)
WITH %CLASSPARAMETER USEEXTENTSET = 0,
    %CLASSPARAMETER DEFAULTGLOBAL = '^GL.EMPLOYEE'
```

DEFAULTGLOBAL を使用して、**拡張グローバル参照**を指定できます。これは、完全参照 (%CLASSPARAMETER DEFAULTGLOBAL = '^|"USER"|GL.EMPLOYEE') にすることも、ネームスペース部分のみ (%CLASSPARAMETER DEFAULTGLOBAL = '^|"USER"|') にすることもできます。

## 引数

### table

CREATE TABLE コマンドで、作成するテーブルの名前を有効な **識別子**として指定します。テーブル名は修飾、未修飾のどちらでもかまいません。

- 未修飾のテーブル名には、tablename という構文を使用します。schema (およびピリオド(.) 文字) は省略します。テーブル名が未修飾の場合は、**既定のスキーマ名**が使用されます。初期のシステム全体の既定のスキーマ名は、既定のクラス・パッケージ名 User に対応する SQLUser です。スキーマ検索パスの値は無視されます。

**システム全体の既定のスキーマ名は構成可能です。**

現在のシステム全体の既定のスキーマ名を確認するには、`$SYSTEM.SQL.Schema.Default()` メソッドを使用します。

- テーブル名を修飾する場合は、schema.tablename という構文を使用します。既存のスキーマ名を指定することも、新規スキーマ名を指定することもできます。既存のスキーマ名を指定すると、テーブルはスキーマ内に配置されます。新規スキーマ名を指定すると、その名前のスキーマ (および関連するクラス・パッケージ) が作成され、テーブルがそのスキーマ内に配置されます。

### テーブル名とスキーマ名の名前付け規約

テーブル名とスキーマ名は、**SQL 識別子**の名前付け規約と、非英数文字の使用、一意性、および最大長に関する追加の制約に従います。% 文字で始まる名前は、システムによる使用のために予約されています。既定では、スキーマ名とテーブル名は単純な識別子であり、大文字と小文字が区別されません。



InterSystems IRIS はテーブル名を使用して、対応するクラス名を生成します。クラス名には英数字 (文字および数字) のみを使用し、最初の 96 文字は一意である必要があります。クラス名を生成するために、InterSystems IRIS はまず記号 (非英数字) 文字をテーブル名から削除し、一意性と最大長の制約を課して一意のクラス名を生成します。

InterSystems IRIS はスキーマ名を使用して、対応するクラス・パッケージ名を生成します。パッケージ名を生成するために、InterSystems IRIS はまずスキーマ名内の記号 (非英数字) 文字に対して削除または特殊処理のいずれかを実行します。次に InterSystems IRIS は、一意性と最大長の制約を課して一意のパッケージ名を生成します。スキーマ名では大文字と小文字が区別されませんが、対応するクラス・パッケージ名では大文字と小文字が区別されます。指定したスキーマ名が、既存のクラス・パッケージ名と大文字/小文字のみが異なり、パッケージ定義が空の (クラス定義が含まれていない) 場合、InterSystems IRIS では、クラス・パッケージ名の大文字/小文字を変更して 2 つの名前が照合されます。

スキーマとテーブルに同じ名前を使用することはできますが、同じスキーマ内でテーブルとビューに同じ名前を使用することはできません。スキーマ名とテーブル名からパッケージ名とクラス名を生成する方法の詳細は、[“テーブル名とスキーマ名”](#) を参照してください。

### テーブル名の文字の制限

InterSystems IRIS は 16 ビット (ワイド) 文字のテーブル名と列名をサポートします。大部分のロケールで、テーブル名にはアクセント記号付き文字を使用でき、生成されるクラス名にもアクセント記号が含まれます。

**注釈** 日本のロケールでは、アクセント記号付きの文字を識別子でサポートしていません。日本語の識別子には、日本語の文字に加え、ラテン文字の A ~ Z と a ~ z (65 ~ 90 および 97 ~ 122)、アンダースコア文字 (95)、とギリシャ文字の大文字 (913 ~ 929 および 931 ~ 937) を使用できます。nl s. Language テストでは、= ではなく [ (包括関係演算子) を使用しています。オペレーティング・システム・プラットフォームごとに日本語ロケールが異なるためです。

### 既存のテーブルのチェック

テーブルが現在のネームスペースに既に存在するかどうかを確認するには、`$SYSTEM.SQL.Schema.TableExists("schema.tname")` を使用します。

既定では、既存のテーブルと同じ名前のインテーブルを作成しようとすると、InterSystems IRIS はテーブルの作成試行を拒否し、SQLCODE -201 エラーを発行します。現在のシステム全体の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。これにより、[ `DDL CREATE TABLE` `CREATE VIEW` ] 設定が表示されます。既定値は 0 で、これが推奨設定です。このオプションを 1 に設定すると、InterSystems IRIS はこのテーブルに対応するクラス定義を削除し、クラスを再作成します。これは、`DROP TABLE` を実行して既存のテーブルを削除し、`CREATE TABLE` を実行する動作に似ています。この場合、`$SYSTEM.SQL.CurrentSettings()` で [ `DDL DROP TABLE` ] の値を 1 (既定) に設定することを強くお勧めします。詳細は、[“DROP TABLE”](#) を参照してください。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション (および他の同様の作成、変更、および削除のオプション) をシステム全体で設定できます。これらの設定の詳細は、[SQL 構成パラメータに関するページ](#) を参照してください。

上記の設定よりも、述語 `IF NOT EXISTS` の動作が優先されます。これらの設定によって実質的にテーブルが上書きされ、SQLCODE 0 が返されます。`IF NOT EXISTS` を指定すると、このコマンドでは何も実行されず、メッセージと共に SQLCODE 1 が返されます。

### column

`CREATE TABLE` コマンドで、作成中のテーブルの列を定義するために使用する 1 つの列名または列名のコンマ区切りのリストを指定します。列名は任意の順序で指定でき、スペースを使用して列名を関連付けられたデータ型から分離します。例: `CREATE TABLE myTable (column1 INT, column2 VARCHAR(10))`。規則によって、各列の定義は通常、別の行にインデントされて表示されます。これは必須ではありませんがお勧めします。列名は、一意の主キーおよび外部キーの制約を定義するためにも使用できます。

列名のリストを括弧で囲みます。

1 つの列を定義する代わりに、複数の列 (プロパティ) を定義する既存の埋め込みシリアル・オブジェクトを列定義で参照することもできます。列名に続けて、シリアル・オブジェクトのパッケージ名とクラス名を指定します。例えば、Office Sample.Address のように指定します。データ型やデータ制約は指定しないでください。ただし、%DESCRIPTION を指定することはできません。CREATE TABLE を使用して、埋め込みシリアル・オブジェクトを作成することはできません。

### 列名の名前付け規約

列名は、識別子の規約に従い、テーブル名と同様の名前付け制約を受けます。列名の先頭文字に % は使用しないでください。列 %z または %Z で始まる列名は許容されます。列名は 128 文字を超えることはできません。既定の列名は、簡単な識別子です。大文字と小文字は区別されません。同一テーブル内の別の列と大文字/小文字区別のみが異なる列名を作成しようとする、SQLCODE -306 エラーが生成されます。

InterSystems IRIS は列名を使用して、対応するクラス・プロパティ名を生成します。プロパティ名には英数字 (文字および数字) のみを使用し、その長さは最大 96 文字です。InterSystems IRIS は、このプロパティ名を生成するために、最初に列名から句読点を削除し、次に 96 文字以内の一意の識別子を生成します。InterSystems IRIS は、一意のプロパティ名を作成するために、必要に応じて列名の最後の文字を、0 で始まる整数に置き換えます。

この例は、InterSystems IRIS が句読点のみ異なる列名を処理する方法を示します。これらの列に対応するクラス・プロパティには、PatNum、PatNu0、および PatNu1 という名前が付けられています。

### SQL

```
CREATE TABLE MyPatients (
  _PatNum VARCHAR(16),
  %Pat@Num INTEGER,
  Pat_Num VARCHAR(30),
  CONSTRAINT Patient_PK PRIMARY KEY (_PatNum))
```

CREATE TABLE で指定された列名は、クラス・プロパティで SqlFieldName キーワード値として表示されます。

動的な SELECT 操作では、一般的な大文字/小文字による変形を促すためにプロパティ名のエイリアスが生成されることがあります。例えば、列名 Home\_Street の場合、プロパティ名のエイリアスとして home\_street、HOME\_STREET、および HomeStreet が割り当てられる可能性があります。その名前が別のフィールド名、または別のフィールド名に割り当てられているエイリアスと競合する場合、エイリアスは割り当てられません。

### type

column によって指定される列名のデータ型クラス。指定されたデータ型により、列で使用するデータ値がそのデータ型に適した値に制限されます。InterSystems SQL は、多くの標準 SQL データ型をサポートしています。

InterSystems SQL データ型 (例えば、VARCHAR(24) や CHARACTER VARYING(24)) またはデータ型のマッピング先のクラス (例えば、%Library.String(MAXLEN=24) または %String(MAXLEN=24)) のどちらかを指定できます。

追加のデータ定義パラメータを定義する場合に、データ型クラスを指定できます。例えば、許可されるデータ値の列挙リスト、許可されるデータ値のパターン・マッチング、数値の最大値と最小値、最大長 (MAXLEN) を超えるデータ値の自動切り捨てなどです。

**注釈** データ型クラス・パラメータの既定値は、InterSystems SQL データ型の既定値とは異なる場合があります。例えば、VARCHAR() および CHARACTER VARYING() は既定で MAXLEN=1 ですが、対応するデータ型クラス %Library.String は既定で MAXLEN=50 です。

InterSystems IRIS は SQL.SystemDataTypes マッピング・テーブルおよび SQL.UserDataTypes マッピング・テーブルによって、これらの標準 SQL データ型を InterSystems IRIS データ型にマップします。

現在のデータ型のマッピングを表示および変更するには、管理ポータルに進み、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[システム DDL マッピング] の順に選択します。データ型マッピングをさらに作成するには、管理ポータルに進み、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[ユーザー DDL マッピング] の順に選択します。



InterSystems IRIS に対応するデータ型がない SQL のデータ型を指定する場合、その SQL データ型の名前が、対応するクラス・プロパティのデータ型として使用されます。DDL の実行 (SQLExecute) 前に、このユーザ定義の InterSystems IRIS データ型を作成する必要があります。

また、データ型のマッピングは、1 つのパラメータ値を上書きできます。例えば、VARCHAR(100) を、与えられた %String(MAXLEN=100) 標準マッピングにマップしたくないとします。この場合、テーブルに VARCHAR(100) の DDL データ型を追加して、これを上書きし、対応する InterSystems IRIS のタイプを指定します。以下はその例です。

```
VARCHAR(100) maps to MyString100(MAXLEN=100)
```

## データ・サイズ

データ型の後に続けて、許容データ・サイズを括弧で囲んで指定できます。データ型の名前とデータ・サイズの括弧の間には空白を挿入してもしなくてもかまいません。

文字列の場合、データ・サイズは最大の文字数を表します。例えば以下のようになります。

```
ProductName VARCHAR (64)
```

小数が許可される数値のデータ・サイズは、(p,s) のように整数のペアとして表します。1 つ目の整数 p は、データ型の精度です。この数字は数値精度、つまり数値の桁数とは異なります。これは、基盤となる InterSystems IRIS データ型クラスは精度を持たないためです。代わりに、これらのクラスはこの数字を MAXVAL 値および MINVAL 値を計算するために使用します。2 つ目の整数 s は、小数部の最大桁数を指定するスケールです。例：

```
UnitPrice NUMERIC(6,2) /* maximum value 9999.99 */
```

精度と位取りの仕組みの詳細は、“[データ型](#)” を参照してください。

## query

CREATE TABLE AS SELECT 構文を使用して作成されたテーブルに列定義と列データを提供する [SELECT](#) クエリ。このクエリは、テーブル、ビュー、または複数の結合テーブルを指定できます。ただし、通常の SELECT クエリのような [パラメータ](#) は指定できません。

CREATE TABLE AS SELECT クエリのデータ定義は以下のとおりです。

- ・ CREATE TABLE AS SELECT は、query テーブルから列定義をコピーします。コピーした列の名前を変更するには、query で [列エイリアス](#) を指定します。  
query で結合テーブルを指定する場合、CREATE TABLE AS SELECT は複数のテーブルから列定義をコピーできます。
- ・ CREATE TABLE AS SELECT は、常に RowID を非公開として定義します。
  - － ソース・テーブルに非公開の RowID がある場合、CREATE TABLE AS SELECT はソース・テーブルの RowID をコピーしませんが、作成したテーブルに新しい RowID 列を作成します。コピーされた行には、新しい一連の RowID 値が割り当てられます。
  - － ソース・テーブルに公開された (非表示ではない) RowID がある場合、またはクエリで明示的に非公開の RowID を選択している場合、CREATE TABLE AS SELECT はこのテーブルに新しい RowID 列を作成します。ソース・テーブルの RowID は、非公開でも、一意でも、必須でもない通常の BigInt 列として新しいテーブルにコピーされます。ソース・テーブルの RowID を “ID” と名付けた場合、新しいテーブルの RowID の名前は “ID1” となります。
- ・ ソース・テーブルに [IDENTITY 列](#) がある場合、CREATE TABLE AS SELECT はこの列とその現在のデータを、ゼロでない正の整数の、一意でも必須でもない通常の BIGINT 列としてコピーします。
- ・ CREATE TABLE AS SELECT は、IDKEY インデックスを定義します。これはコピーされた列定義に関連付けられたインデックスをコピーしません。

- ・ CREATE TABLE AS SELECT は、列の制約をコピーしません。これはコピーされた列定義に関連付けられた NULL/NOT NULL、UNIQUE、主キー、または外部キーの制約をコピーしません。
- ・ CREATE TABLE AS SELECT は、既定の制約またはコピーされた列定義に関連付けられた値をコピーしません。
- ・ CREATE TABLE AS SELECT は、コピーされた列定義に関連付けられた COMPUTECODE データ制約をコピーしません。
- ・ CREATE TABLE AS SELECT は、コピーされたテーブルまたは列定義に関連付けられた %DESCRIPTION 文字列をコピーしません。

## defaultSpec

列の既定値は、リテラル値またはキーワード・オプションとして DEFAULT 節で指定できます。リテラル既定値として指定する文字列は一重引用符で囲む必要があります。数値既定値には一重引用符は必要ありません。例えば以下のようになります。

## SQL

```
CREATE TABLE membertest (
  MemberId INT NOT NULL,
  Membership_status CHAR(13) DEFAULT 'M',
  Membership_term INT DEFAULT 2)
```

DEFAULT 値はテーブル作成時に検証されません。DEFAULT 値を定義すると、その値ではデータ型、データ長、およびデータ制約の制限を無視できます。ただし、INSERT を使用してテーブルにデータを提供する際には、DEFAULT 値は制限されます。DEFAULT 値はデータ型とデータ長の制限は受けませんが、データ制約の制限は受けます。例えば、`Ordernum INT UNIQUE DEFAULT 'No Number'` で定義された列の場合、最初は INT データ型の制限を無視して既定値を取ることができます。しかし、2 回目は列の UNIQUE データ制約に違反するため既定値を取ることができません。

DEFAULT が指定されていないければ、暗黙の既定は NULL になります。列に NOT NULL データ制約があれば、この列には明示的に、または DEFAULT で値を指定する必要があります。NOT NULL の既定値として [長さゼロの SQL 文字列](#) (空文字列) は使用しないでください。NULL 値と空文字列の詳細は、["NULL"](#) を参照してください。

DEFAULT データ制約は、NULL、USER、CURRENT\_USER、SESSION\_USER、SYSTEM\_USER、CURRENT\_DATE、CURRENT\_TIME、CURRENT\_TIMESTAMP、SYSDATE、OBJECTSCRIPT などのキーワード・オプションを受け入れます。

USER、CURRENT\_USER、および SESSION\_USER の DEFAULT キーワードは、列値を ObjectScript の `$USERNAME` 特殊変数に設定します。

[CURRENT\\_DATE](#)、[CURRENT\\_TIME](#)、[CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[GETUTCDATE](#)、および [SYSDATE](#) SQL 関数も、DEFAULT 値として使用できます。これらについては、それぞれのリファレンス・ページで説明しています。DEFAULT 値として使用する場合、CURRENT\_TIME またはタイムスタンプ関数は、有効桁数の値を付けても付けなくても指定できます。有効桁数を指定しない場合、InterSystems SQL では SQL 構成設定 [GETDATE(), CURRENT\_TIME, CURRENT\_TIMESTAMP のデフォルトの時刻精度] の有効桁数が使用されます。既定値は 0 です。DEFAULT 関数では、CREATE TABLE 文の実行時ではなく、文が準備/コンパイルされるときに有効になっている時刻精度設定が使用されます。

[CURRENT\\_TIMESTAMP](#) は、データ型 `%Library.PosixTime` または `%Library.TimeStamp` の列の既定値として指定できます。現在の日付と時刻は、列のデータ型によって指定された形式で格納されます。[CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[GETUTCDATE](#)、および [SYSDATE](#) は、`%Library.TimeStamp` 列 (データ型 `TIMESTAMP` または `DATETIME`) の既定値として指定できます。InterSystems IRIS は、データ型に合った形式にデータ値を変換します。

## SQL

```
CREATE TABLE mytest (
  TestId INT NOT NULL,
  CREATE_TIMESTAMP DATE DEFAULT CURRENT_TIMESTAMP(2),
  WORK_START TIMESTAMP DEFAULT SYSDATE)
```

データ型 DATE には [TO\\_DATE](#) 関数を DEFAULT データ制約として使用できます。データ型 TIMESTAMP には [TO\\_TIMESTAMP](#) 関数を DEFAULT データ制約として使用できます。

DATE、TIMESTAMP、または TIMESTAMP2 フィールドでは、defaultSpec は ODBC 日付形式で記述できます。InterSystems IRIS は、指定された列タイプへの変換を処理します。

また、OBJECTSCRIPT literal キーワード句を使用すると、以下の例に示すように、ObjectScript コードを含む引用符付き文字列を指定することで既定値を生成できます。

## SQL

```
CREATE TABLE mytest (
    TestId INT NOT NULL,
    CREATE_DATE DATE DEFAULT OBJECTSCRIPT '+$HOROLOG' NOT NULL,
    LOGNUM NUMBER(12,0) DEFAULT OBJECTSCRIPT '$INCREMENT(^LogNumber)')
```

詳細は、“[ObjectScript リファレンス](#)”を参照してください。

## uniqueName

有効な識別子として指定された、CONSTRAINT UNIQUE 節でリストされた制約名です。[区切り識別子](#)として指定されている場合、制約名に “.”、“””、“,”、および “->” 文字を含めることができます。制約名はその制約を一意に識別し対応するインデックス名を生成する際にも使用されます。この制約名は、[ALTER TABLE](#) コマンドを使用してテーブル定義から制約を削除する際に必要になります。CONSTRAINT UNIQUE にリストされている列を ALTER TABLE で削除することはできないことに注意してください。これを実行しようとすると、SQLCODE -322 エラーが生成されます。

CONSTRAINT UNIQUE 節には以下の構文があります。

```
CONSTRAINT uniqueName UNIQUE (column1,column2)
```

この制約は列値 column1 および column2 の組み合わせが必ず一意でなければならないことを示します。これらの個々の列の値自体は一意でなくてもかまいません。この制約に対して、1 つ以上の列を指定できます。

この制約で指定する列はすべて、列定義で定義する必要があります。列定義で定義されていない列を指定すると、SQLCODE -86 エラーが生成されます。指定する列には NOT NULL を定義する必要があります。指定する列にはいずれも [UNIQUE](#) を定義しないでください。そのように定義すると、この制約を指定しても意味がありません。

列は任意の順序で指定できます。この列の順序によって、対応するインデックス定義の列の順序が決まります。重複する列名は許可されます。複数列での UNIQUE 制約に単一の列名を指定することができますが、これはその列に UNIQUE データ制約を指定するのと機能的には同じです。将来的に使用できるように、個々の単一系列の制約が 1 つの制約名を指定します。

複数列での一意制約の複数の文を 1 つのテーブル定義で指定することができます。制約文は列定義の任意の場所で指定できます。慣例により、通常は定義される列のリストの最後に配置します。

UNIQUE 制約と共に定義されているテーブルの列をリストする方法は、“[\[カタログの詳細\] の \[制約\] オプション](#)”を参照してください。

## pKeyName

有効な識別子として指定された、PRIMARY KEY 制約節で定義された主キー名です。[区切り識別子](#)として指定されている場合、制約名に “.”、“””、“,”、および “->” 文字を含めることができます。[ALTER TABLE](#) ではこのオプションの制約名を使用して、定義されている制約を識別します。

テーブルの主キーの定義の詳細は、“[主キーの定義](#)”を参照してください。

## fKeyName

有効な識別子として指定された、FOREIGN KEY 制約節で定義された外部キー名です。[区切り識別子](#)として指定されている場合、制約名に “.”、“””、“,”、および “->” 文字を含めることができます。[ALTER TABLE](#) ではこのオプションの制約名を使用して、定義されている制約を識別します。

テーブルの外部キーの定義の詳細は、“[外部キーの定義](#)”を参照してください。

## refTable

有効な識別子として指定された、FOREIGN KEY 節で参照するテーブル名です。テーブル名は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。

## refColumn

外部キー制約で指定した参照テーブルで定義されている列名またはコンマで区切られた既存の列名のリスト。参照される列を括弧で囲みます。refColumn を省略すると、“[外部キーの定義](#)”で説明しているように、.CREATE TABLE は既定の参照列を割り当てます。

参照列として明示的な RowID を指定するには、refColumn を %ID として指定してください。例えば、FOREIGN KEY (CustomerNum) REFERENCES Customers (%ID) のようになります。この値は、参照テーブルに主キーまたは外部キーが指定されていない限り、省略された列名と同義です。テーブルのクラス定義に [SqlRowIdName](#) が含まれる場合、この値を明示的な RowID として指定できます。

## refAction

テーブルに外部キーが含まれる場合は、1つのテーブルでの変更が他のテーブルにも影響します。外部キーを定義するときにデータの整合性を維持するには、外部キーの元となっているレコードを変更したときに外部キー値に与える影響も定義します。CREATE TABLE で、ON DELETE refAction および ON UPDATE refAction 節は、[refColumn](#) で指定された外部キー列が変更されたときに実行されるアクションを指定します。

- ON DELETE 節は、参照テーブルに対する DELETE ルールを定義します。参照テーブルの行を削除しようとした場合に、ON DELETE 節は参照テーブル内の行に対して実行されるアクションを定義します。
- ON UPDATE 節は、参照テーブルに対する UPDATE ルールを定義します。参照テーブルの行の主キーの値を変更 (更新) しようとした場合に、ON UPDATE 節は参照テーブル内の行に対して実行されるアクションを定義します。

InterSystems SQL は以下の外部キー参照アクションをサポートします。

参照アクション	説明
NO ACTION (既定)	外部キー列内のいずれかの行が、削除または更新対象の行を参照している場合、削除または更新は失敗します。この制約は、外部キーが自身を参照している場合、適用されません。
SET DEFAULT	削除または更新対象の行を参照している外部キー列を、既定値に設定します。外部キー列が既定値を持たない場合は、NULL に設定されます。行は参照されるテーブル内に存在しなければなりません。また、参照されるテーブルには、既定値に対するエントリが含まれます。
SET NULL	削除または更新対象の行を参照している外部キー列を、NULL に設定します。外部キー列は、NULL 値を許可します。
CASCADE	ON DELETE – 削除対象の行を参照する外部キー列の行も削除します。 ON UPDATE – 更新対象の行を参照する外部キー列の行も更新します。

同じ列の組み合わせを参照して相反する参照アクションを実行する、名前が異なる2つの外部キーを定義しないでください。そのような2つの外部キーを定義した場合(例: ON DELETE CASCADEとON DELETE SET NULL)、InterSystems SQLはANSI標準に従い、エラーを生成しません。この代わりにInterSystems SQLでは、DELETE処理またはUPDATE処理でこのような相反する外部キー定義に遭遇したときにエラーを生成します。詳細は、“[外部キーの使用法](#)”を参照してください。

## code

列の既定値を計算するためにCOMPUTECODEデータ制約で使用するコード行です。コードを中括弧で囲んで指定します。空白および改行は中括弧の前後で使用できます。

コードのプログラミング言語は、COMPUTECODE節で設定した値に応じて異なります。

- COMPUTECODE または COMPUTECODE OBJECTSCRIPT - code を ObjectScript コードで指定します。コードでは、中括弧で囲んで SQL 列名を参照できます(例: {DOB})。ObjectScript コードでは[埋め込み SQL](#)を使用できます。投影されたクラスでは、COMPUTECODE に列名 [SqlComputeCode](#) とその列の値の計算を指定します。
- COMPUTECODE PYTHON - code を Python コードで指定します。コードでは、cols.getField メソッドを使用して SQL 列名を参照できます(例: cols.getField('DOB'))。投影されたクラスでは、COMPUTECODE に *PropertyComputation* クラス・メソッドを指定します。このメソッドに、列の値を計算するコードを収めます。*Property* は計算する列の名前です。投影されたクラスでは、このクラス・メソッドが *SqlComputeCode* プロパティ・キーワードの代わりに使用されます。

COMPUTECODE または *SqlComputeCode* プロパティ・キーワードに計算済みフィールドの名前を指定する場合は、SQL フィールド名を指定する必要があります。生成済みテーブルの対応するプロパティ名ではありません。

COMPUTECODE によって提供される既定データ値は、論理(内部保存)モードである必要があります。計算コード内の埋め込み SQL は、自動的にコンパイルされ、論理モードで実行されます。

以下の例では、Birthday COMPUTECODE 列を定義します。この定義は、ObjectScript コードを使用して DOB 列値から既定値を計算します。

## SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    Birthday VARCHAR(12) COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB},9),",",")},
    Grade INT)
```

COMPUTECODE には、擬似フィールド参照変数 %%CLASSNAME、%%CLASSNAMEQ、%%OPERATION、%%TABLE-NAME、%%ID を指定できます。これらの変数は、クラスのコンパイル時に特定の値に変換されます。変数の大文字と小文字は区別されません。

- ObjectScript の計算コードでは、擬似フィールド参照変数を中括弧で囲んで呼び出します。例: {%%CLASSNAME}
- Python の計算コードでは、cols.getField メソッドを使用して擬似フィールド参照変数を呼び出します。例: cols.getField(%%CLASSNAME)

COMPUTECODE 値は既定値です。列に値を指定しない場合にのみ返されます。COMPUTECODE 値はデータ型の制限を受けません。COMPUTECODE 値は UNIQUE データ制約などのデータ制約の制限を受けません。DEFAULT および COMPUTECODE の両方を指定した場合は、必ず DEFAULT が取られます。

COMPUTECODE ではオプションで COMPUTEONCHANGE、CALCULATED、または TRANSIENT キーワードを取ることができます。

ObjectScript の COMPUTECODE コードにエラーがある場合は、そのコードが初めて実行されるときまで、SQL ではこのエラーは検出されません。そのため、挿入時に値が初めて計算される場合は、INSERT 操作は SQLCODE -415 エラーで失敗します。更新時に値が初めて計算される場合は、UPDATE 操作は SQLCODE -415 エラーで失敗します。クエリ時に値が初めて計算される場合は、SELECT 操作は SQLCODE -350 エラーで失敗します。



COMPUTECODE 保存値にインデックスを付けることができます。アプリケーション開発者は、特に計算列のインデックスを定義する (またはしようとしている) 場合、そのデータ型に基づいて、計算列の保存値が検証され、正規化されている (キャノン形式の数値) ことを確認する必要があります。

## updateSpec

ON UPDATE 節を使用してテーブルを作成し、列を指定すると、その列はテーブルで行が更新されるたびに計算されます。この機能が最もよく使用されるのは、前回行が更新されたときのタイムスタンプ値を含むテーブルで列を定義する場合です。

利用可能な updateSpec オプションは以下のとおりです。

```
CURRENT_DATE | CURRENT_TIME[(precision)] | CURRENT_TIMESTAMP[(precision)] | GETDATE([prec]) |
GETUTCDATE([prec]) | SYSDATE | USER | CURRENT_USER | SESSION_USER | SYSTEM_USER | NULL | <literal>
| -<number>
```

以下の例は、行の挿入時、および行の更新のたびに、RowTS 列を現在のタイムスタンプ値に設定します。

```
CREATE TABLE mytest (
  Name VARCHAR(48),
  RowTS TIMESTAMP DEFAULT Current_Timestamp(6) ON UPDATE Current_Timestamp(6) )
```

この例で、DEFAULT キーワードは、INSERT 時に RowTS 列に明示的な値が指定されていない場合、この RowTS に現在のタイムスタンプを設定します。UPDATE が RowTS 列に明示的な値を指定した場合、ON UPDATE キーワードは指定された値を検証しますが、これを無視し、RowTS を現在のタイムスタンプで更新します。指定された値が検証に失敗した場合、SQLCODE -105 エラーが生成されます。

以下の例では、HasBeenUpdated 列をブーリアン値に設定します。

```
CREATE TABLE mytest (
  Name VARCHAR(48),
  HasBeenUpdated TINYINT DEFAULT 0 ON UPDATE 1 )
```

以下の例では、WhoLastUpdated 列を現在のユーザ名に設定します。

```
CREATE TABLE mytest (
  Name VARCHAR(48),
  WhoLastUpdated VARCHAR(48) DEFAULT CURRENT_USER ON UPDATE CURRENT_USER )
```

列に COMPUTECODE データ制約も存在する場合、ON UPDATE 節を指定することはできません。それらを参照すると、コンパイルまたは準備時に SQLCODE -1 エラーが返されます。

## description

InterSystems SQL で提供されている %DESCRIPTION キーワードは、テーブルまたは列の説明を挿入するために使用します。%DESCRIPTION の後には、一重引用符で囲んだテキスト文字列の description を続けます。このテキストの長さに制限はなく、空白スペースを含むすべての文字を使用できます(説明文字列内の一重引用符は、二重引用符で代用します。例えば、'Joe's Table' のようになります)。1 つのテーブルには 1 つの %DESCRIPTION を使用できます。テーブルの各列には、データ型の後にそれぞれ独自の %DESCRIPTION を指定できます。テーブル全体の %DESCRIPTION を複数指定すると、SQLCODE -82 エラーが発行されます。列の %DESCRIPTION を複数指定すると、最後に指定された %DESCRIPTION のみが保持されます。ALTER TABLE を使用して既存の説明を変更することはできません。

対応する永続クラス定義では、説明は、対応するクラス (テーブル) 構文またはプロパティ (列) 構文の 1 つ前の行に、3 つのフラッシュが付付けられて表示されます。例えば、/// Joe's Table のようになります。対応する永続クラスの "クラス・リファレンス" では、テーブルの説明は、上部のクラス名と SQL テーブル名の直後に表示されます。列の説明は、対応するプロパティ構文の直後に表示されます。

%DESCRIPTION テキストを表示するには、INFORMATION.SCHEMA.TABLES または INFORMATION.SCHEMA.COLUMNS の DESCRIPTION プロパティを使用します。例:

## SQL

```
SELECT COLUMN_NAME,DESCRIPTION FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='MyTable'
```

### sqlCollation

列の値をソートするために使用される照合タイプ。SQL 照合タイプ (%EXACT、%MINUS、%PLUS、%SPACE、%SQLSTRING、%SQLUPPER、%TRUNCATE、または %MVR) のいずれかとして指定します。照合のキーワードでは、大文字と小文字は区別しません。プログラミングを明確にする目的で、照合パラメータの前にオプション・キーワードの COLLATE を指定することを推奨します。ただし、このキーワードは必須ではありません。これらのさまざまな照合パラメータ・キーワードの先頭のパーセント記号 (%) もオプションです。

既定は、[ネームスペースの既定の照合](#)です (変更していない場合は %SQLUPPER です)。%SQLSTRING、%SQLUPPER、および %TRUNCATE は、オプションの最大長のトランケーション引数である、括弧で囲んだ整数を指定できます。照合の詳細は、["テーブルのフィールド/プロパティ定義の照合"](#) を参照してください。

%EXACT 照合は、ANSI (または Unicode) 文字の照合順に従います。大文字/小文字を区別する文字列照合を行い、先頭および末尾の空白およびタブ文字を認識します。

%SQLUPPER 照合は、照合目的ですべての文字を大文字に変換します。大文字と小文字を区別しない照合の詳細は、["%SQLUPPER"](#) 関数を参照してください。

%SPACE および %SQLUPPER 照合はデータに空白を追加します。これにより、NULL と数値の文字列照合が強制されます。

%SQLSTRING、%SQLUPPER、および %STRING 照合では、括弧で囲む必要のある maxlen パラメータをオプションで使うことができます。maxlen は切り捨てを命令する整数で、照合の実行時に対象とする最大文字数を指定します。このパラメータは、サイズの大きなデータ値を持つ列にインデックスを作成するときに便利です。

%PLUS および %MINUS 照合は、NULL をゼロ (0) 値として処理します。

InterSystems SQL には、これらの照合タイプのほとんどに対応する関数があります。詳細は、[%EXACT](#)、[%SQLSTRING](#)、[%SQLUPPER](#)、[%TRUNCATE](#) の各関数を参照してください。

ObjectScript では、データ照合変換のために %SYSTEM.Util クラスの Collation() メソッドが提供されています。

**注釈**      ネームスペースの既定の照合を %SQLUPPER (大文字/小文字の区別なし) から %SQLSTRING (大文字/小文字の区別あり) などの他の照合タイプに変更するには、以下のコマンドを使用します。

#### ObjectScript

```
WRITE $$SetEnvironment^%apiOBJ("collation", "%Library.String", "SQLSTRING")
```

このコマンドを発行した後に、インデックスを削除し、すべてのクラスを再コンパイルしてから、再度インデックスを構築する必要があります。他のユーザがテーブルのデータにアクセスしている間はインデックスを再構築しないでください。再構築すると、クエリ結果が不正確になる可能性があります。

### shardKeyColumn

シャード・キーとして使用される 1 つの列または列のコマ区切りのリストです。shardKeyColumn は、SHARD KEY 節内のテーブル列リストの閉じ括弧の直後、かつ WITH 節 (指定される場合) の前に指定します。シャード・キー定義をテーブル列リスト内の要素として指定することは、下位互換性保持のためにサポートされていますが、両方の場所でシャード・キーを定義すると、SQLCODE -327 エラーが生成されます。

RowID 列をシャード・キーとして定義することはできません。ただし、作成されたテーブルに [IDENTITY 列](#) または [IDKEY](#) が含まれている場合は、これらの列のどちらかをシャード・キーとして定義できます。

シャード・キーの選択の詳細は、["シャード・キーの選択"](#) を参照してください。



## coshardKeyColumn

coshardTable で定義したテーブルのシャード・キーとのコシャード結合で使用するシャード・キー列の名前です。coshardKeyColumn は、COSHARD WITH 構文 (SHARD KEY (*coshardKeyColumn*) COSHARD WITH *coshardTable*) で指定します。

## coshardTable

作成中のテーブルがコシャードする既存のテーブルの名前。COSHARD WITH 節で指定されたテーブルは、システムによって割り当てられたシャード・キーを持つシャード・テーブルである必要があります。

このテーブルを指定すると、シャード・テーブルの [ShardKey インデックス](#)に CoshardWith インデックス・キーワードが設定されます。この CoshardWith インデックス・キーワードは、テーブルを投影するクラスと等しくなります。

クエリで指定したどのシャード・テーブルがコシャードされるか確認するには、Cosharding コメント・オプションを表示します。

## pName = pValue

pName という名前のクラス・パラメータを pValue の値に設定する、%CLASSPARAMETER の名前と値のペア。コンマ区切りの名前と値のペアを使用して、複数の %CLASSPARAMETER 節を指定できます。例: WITH %CLASSPARAMETER DEFAULTGLOBAL = '^GL.EMPLOYEE', %CLASSPARAMETER MANAGEDEXTENT 0 等号または少なくとも 1 つのスペースを使用して、名前と値を区切ります。クラス・パラメータ値はリテラル文字列および数字で、定数値として定義する必要があります。

現在使用されているクラス・パラメータには、[ALLOWIDENTITYINSERT](#)、[DATALOCATIONGLOBAL](#)、[DEFAULTGLOBAL](#)、[DSINTERVAL](#)、[DSTIME](#)、[EXTENTQUERYSPEC](#)、[EXTENTSIZE](#)、[GUIDENABLED](#)、[MANAGEDEXTENT](#)、[READONLY](#)、[ROWLEVELSECURITY](#)、[SQLPREVENTFULLSCAN](#)、[USEEXTENTSET](#)、[VERSIONCLIENTNAME](#)、[VERSIONPROPERTY](#) などがあります。これらのクラス・パラメータの説明は、“[%Library.Persistent](#)” クラスを参照してください。

USEEXTENTSET および DEFAULTGLOBAL クラス・パラメータを使用して、テーブル・データ・ストレージとインデックス・データ・ストレージの[グローバル名前付け方式](#)を定義できます。

IDENTIFIEDBY クラス・パラメータは非推奨です。IDENTIFIEDBY リレーションシップは、InterSystems IRIS でサポートされるように、適切な親子リレーションシップに変換する必要があります。

シャード・テーブルを定義する CREATE TABLE で、クラス・パラメータ DEFAULTGLOBAL、DSINTERVAL、DSTIME、または VERSIONPROPERTY を定義することはできません。

必要に応じて追加のクラス・パラメータを指定できます。詳細は、“[クラス・パラメータ](#)” を参照してください。

## 例

### テーブルの作成および生成

CREATE TABLE を使用して、複数の列を持つテーブル Employee を作成します。

- EmpNum 列 (会社の従業員の ID 番号を含む) は、NULL ではない整数です。また、テーブルの主キーとして宣言され、テーブルに行が挿入されるたびに自動的にインクリメントされます。
- 従業員の姓と名は、最大 30 文字で NULL にはできない文字列の列に格納されます。

- ・ 残りの列には、従業員の入社日や有給休暇、病欠日を格納します。これらの列には `TIMESTAMP` および `INT` データ型を使用します。

```
CREATE TABLE Employee (
  EmpNum INT NOT NULL AUTO_INCREMENT,
  NameLast CHAR(30) NOT NULL,
  NameFirst CHAR(30) NOT NULL,
  StartDate TIMESTAMP,
  AccruedVacation INT,
  AccruedSickLeave INT,
  CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum))
```

テーブル・スキーマを変更するには、`ALTER TABLE` を使用します。例えば、この文はテーブルの名前を `Employee` から `Employees` に変更します。

```
ALTER TABLE Employee RENAME Employees
```

テーブルに行を挿入するには、`INSERT` を使用します。例えば、この文はテーブルに必要な列のみを持つ行を挿入します。EmpNum 列も必要ですが、これは自動インクリメントされるため、指定する必要はありません。

## SQL

```
INSERT INTO Employees (NameLast, NameFirst) VALUES ('Zubik','Jules')
```

挿入された行を更新するには、`UPDATE` を使用します。例えば、挿入された行で、この文はデータが欠落している列の 1 つに値を設定します。

## SQL

```
UPDATE Employees SET AccruedVacation = 15 WHERE Employees.EmpNum = 1
```

行を削除するには、`DELETE` を使用します。例えば、この文は挿入された行を削除します。

## SQL

```
DELETE FROM Employess WHERE EmpNum = 1
```

テーブル全体を削除するには、`DROP TABLE` を使用します。DROP TABLE の使用には注意が必要です。`%NODELDDATA` キーワードを指定しない限り、このコマンドはテーブルとすべての関連付けられたデータの両方を削除します。

## SQL

```
DROP TABLE Employess
```

## セキュリティおよび特権

CREATE TABLE コマンドは、`%CREATE_TABLE` の管理権限を必要とする、特権の必要な操作です。特権なしで CREATE TABLE コマンドを実行すると、SQLCODE -99 エラーが返されます。`%CREATE_TABLE` 特権をユーザまたはロールに割り当てるには、`GRANT` コマンドを使用します。この場合、適切な付与特権を保持していることを前提とします。CREATE TABLE AS SELECT 構文を使用する場合、`query` で指定されるテーブル上で SELECT 特権が必要です。管理特権はネームスペース固有のものです。詳細は、“特権”を参照してください。

既定では、CREATE TABLE のセキュリティ特権が適用されます。システム全体でこの特権の要件を構成するには、`$SYSTEM.SQL.Util.SetOption()` メソッドを使用します。例: `SET status=$SYSTEM.SQL.Util.SetOption("SQLSecurity",0,.oldval)` 現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` メソッドを呼び出します。これにより、[SQL ] の設定が表示されます。既定値は 1 (有効) です。SQL セキュリティが有効な場合 (推奨)、ユーザは特権を保持しているテーブルやビューのみでアクションを実行できます。このメソッドを 0 に設定すると、この設定の変更後に開始された新しいプロセスすべてで SQL セキュリティは無効になります。つまり、特権ベースのテーブルやビューのセキュリティは抑制されていることを意味します。

ユーザを指定しなくてもテーブルの作成が可能になります。この場合、ダイナミック SQL はユーザとして “\_SYSTEM” を、埋め込み SQL はユーザとして “” (空文字列) を割り当てます。ユーザは特権がなくてもテーブルやビューに対してアクションを実行することができます。

埋め込み SQL は、SQL 特権を使用しません。埋め込み SQL では、以下のように \$SYSTEM.Security.Login() メソッドを使用して適切な特権を持ったユーザとしてログインできます。\$SYSTEM.Security.Login() メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。例：

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
NEW SQLCODE, %msg
&sql(CREATE TABLE MyTable (col1 INT, col2 INT))
IF SQLCODE=0 {WRITE !, "Table created"}
ELSE {WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
```

詳細は、“%SYSTEM.Security” を参照してください。

実行コードを必要とする計算済み列で CREATE TABLE を使用する場合は、%CREATE\_TABLE 特権のほかに %Development:USE 特権が必要です。ただし、このコマンドを埋め込み SQL で使用する場合は除きます。

また、%SQL.Statement クラスのコマンドを作成し、checkPriv 引数を 0 に設定した %Prepare() メソッドまたは %ExecDirectNoPriv() メソッドを使用して、特権の確認を回避することもできます。

## 詳細

### 作成されたテーブルのクラス定義

CREATE TABLE を使用して SQL テーブルを作成すると、InterSystems IRIS® は、このテーブル定義に対応する [永続クラス](#) と、列定義に対応するプロパティを自動的に作成します。

CREATE TABLE は、対応するクラスを [DdlAllowed](#) として定義します。対応するクラス定義で明示的な [StorageStrategy](#) を指定しません。既定のストレージ [%Storage.Persistent](#) を使用します。既定では、CREATE TABLE は対応するクラス定義で [Final](#) クラス・キーワードを指定します。これは、サブクラスを持ってないことを示しています (既定値は 1 です。\$SYSTEM.SQL.Util.SetOption() メソッドの SET status=\$SYSTEM.SQL.Util.SetOption("DDLFinal", 0, .oldval) を使用して、このシステム全体の既定値を変更できます。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() method) を呼び出します。

### 主キーの定義

主キーの定義はオプションです。テーブルを定義すると、生成列 [RowID 列](#) (既定名 “ID”) が自動的に作成されます。これは一意の行識別子として機能します。各レコードがテーブルに追加されると、InterSystems IRIS はそのレコードの RowID 列に、変更できない一意の正の整数を割り当てます。オプションで、一意の行識別子としても機能する主キーを定義できます。主キーにより、ユーザはアプリケーションにとって意味のある行識別子を定義できます。例えば、主キーは、従業員 ID 列、社会保障番号、患者レコード ID 列、在庫ストック番号などになります。PRIMARY KEY 節を使用して、列または列のグループを主レコード識別子として明示的に定義することができます。

主キーには、重複する値や NULL 値は設定できません ([主キー・インデックス](#)・プロパティは自動的に必須として定義されるわけではありませんが、主キー列には NULL 値を保存できないので、事実上、必須です)。主キーの [照合タイプ](#) は、列そのものの定義で指定されます。

主キーとして定義されているテーブルの列をリストする方法は、“[\[カタログの詳細\] の \[制約\] オプション](#)” を参照してください。

詳細は、“[主キー](#)” を参照してください。

## IDKEY の主キー

既定では、主キーは一意の IDKEY インデックスではありません。通常は、主キー値の更新や、主キーの[照合タイプ](#)の設定などを行えるため、既定を推奨します。ただし、主キーを IDKEY インデックスとして定義の方が好ましい場合もあります。その場合は、その後主キーを使用するにあたって IDKEY が制限されることを考慮する必要があります。

既存の列に主キー制約を追加する場合、列が自動的に [IDKEY インデックス](#) として定義されることもあります。これはデータが存在するかどうか、および構成設定が以下のいずれかの方法で設定されているかどうかによります。

- ・ SQL [SET OPTION](#) PKEY\_IS\_IDKEY 文
- ・ システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DDLPrimaryKeyNotIDKey。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、「DDL ID」と表示されます。既定値は 1 です。
- ・ 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[DDL を使用して作成されたテーブルの ID キーとして主キーを定義する] の現在の設定を表示します。
  - － このチェック・ボックスにチェックが付いていない場合 (既定)、主キーはクラス定義の [IDKEY インデックス](#) になりません。IDKEY ではない主キーを使用したレコードへのアクセスはかなり非効率的です。しかし、主キーの値は変更することができます。
  - － このチェック・ボックスにチェックが付いている場合、DDL で主キー制約を指定していると、主キーが自動的にクラス定義の [IDKEY インデックス](#) になります。このオプションにチェックを付けると、データ・アクセスはより効率的ですが、一度設定された主キー値は変更できません。

ただし、[IDENTITY 列](#)がテーブルで定義済みであると、これらの構成設定のいずれかを使用して主キーを IDKEY として定義していても、主キーを IDKEY として定義することはできません。

InterSystems IRIS では、IDKEY インデックスに属するプロパティ (列) を [SqlComputed](#) にすることができます。例えば、親参照列などです。このプロパティは、トリガによって計算される列とする必要があります。SqlComputed として定義した IDKEY プロパティは、新規オブジェクトを初めて保存するときまたは INSERT 操作のときのみ計算されます。UPDATE 計算は、IDKEY インデックスに属する列を更新できないため、サポートされていません。

## 主キーなし

多くの場合、明示的に主キーを定義する必要があります。ただし、主キーが指定されない場合、InterSystems IRIS では、以下のルールに基づいて、別の列を ODBC/JDBC プロジェクションの主キーとして使用します。

1. 単一の列上に [IDKEY インデックス](#)が存在する場合、IDKEY 列を SQLPrimaryKey 列として報告する。
2. そうでない場合、クラスが [SqlRowIdPrivate=0](#) (既定) で定義されていれば、RowID 列を SQLPrimaryKey 列として報告する。
3. そうでない場合、IDKEY インデックスが存在すると、IDKEY 列を SQLPrimaryKey 列として報告する。
4. そうでない場合、SQLPrimaryKey を報告しない。

## 複数の主キー

主キーは 1 つしか定義できません。既定では、InterSystems IRIS は、主キーが既に存在する場合に主キーを定義しようとしたり、同じ主キーを 2 回定義しようとしたりすると、それを拒否して SQLCODE -307 エラーを発行します。主キーの第 2 定義が最初の定義と同じ場合も SQLCODE -307 エラーを発行します。現在の構成を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[DDL Create Primary Key] 設定が表示されます。既定値は 0 (いいえ) で、これが推奨される構成設定です。このオプションが 1 (はい) に設定されている場合、InterSystems IRIS は既存の主キー制約を削除し、最後に指定された主キーをテーブルの主キーとして設定します。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション（および他の同様の作成、変更、および削除のオプション）をシステム全体で設定できます。

例えば、以下に CREATE TABLE 文があります。

## SQL

```
CREATE TABLE MyTable (f1 VARCHAR(16),
CONSTRAINT MyTablePK PRIMARY KEY (f1))
```

この文は、主キーを作成します（存在しない場合）。次に、ALTER TABLE 文があります。

## SQL

```
ALTER TABLE MyTable ADD CONSTRAINT MyTablePK PRIMARY KEY (f1)
```

上記の例は、SQLCODE -307 エラーを生成します。

## 外部キーの定義

外部キーは他のテーブルを参照する列です。外部キー列に保存された値は、他のテーブル内のレコードを一意に識別します。この参照の最も単純な形式を以下の例に示します。この例では、外部キーが Customers テーブルの主キー列 CustID を暗黙的に参照しています。

## SQL

```
CREATE TABLE Orders (
  OrderID INT UNIQUE NOT NULL,
  OrderItem VARCHAR,
  OrderQuantity INT,
  CustomerNum INT,
  CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
  CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID))
```

通常、外部キーは他のテーブルの主キー列を参照します。ただし、外部キーは RowID (ID) または **IDENTITY 列** を参照できます。外部キー参照は参照されるテーブルに常に存在し、一意として定義される必要があります。参照される列は重複値または NULL を持つことはできません。

外部キー定義では以下の内容を指定できます。

- ・ 1 つの列名: FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID)。外部キー列 (CustomerNum) および参照される列 (CustID) は異なる名前でも（同じ名前でも）かまいませんが、データ型と列の制約は同じでなければなりません。
- ・ コンマで区切られた列名のリスト: FOREIGN KEY (CustomerNum,SalespersonNum) REFERENCES Customers (CustID,SalespID)。外部キー列および参照される列は、列の数およびリストでの順序が一致している必要があります。
- ・ 省略された列名: FOREIGN KEY (CustomerNum) REFERENCES Customers。
- ・ 明示的な RowID 列: FOREIGN KEY (CustomerNum) REFERENCES Customers (%ID)。省略された列名と同義です。テーブルのクラス定義に **SqlRowIdName** が含まれる場合、この値を明示的な RowID として指定できます。

外部キーを定義して、参照される列名を省略した場合、外部キーの既定は次のようになります。

1. 指定されたテーブルで定義されている主キー列。
2. 指定されたテーブルに主キーが定義されていない場合、外部キーの既定は指定されたテーブルで定義されている IDENTITY 列になります。



3. 指定されたテーブルに主キーも IDENTITY 列も定義されていない場合、外部キーの既定は RowID になります。これが発生するのは、指定されたテーブルで RowID をパブリックとして定義した場合のみです。指定されたテーブル定義でこれを明示的に行うには、%PUBLICROWID キーワードを指定するか、対応するクラス定義で `SqlRowIdPrivate=0` (既定) を指定します。指定されたテーブルで RowID をパブリックとして定義していない場合、InterSystems IRIS は SQLCODE -315 エラーを発行します。RowID で外部キーを定義する際は、参照される列名を省略する必要があります。ID を参照される列名として明示的に指定しようとすると、SQLCODE -316 エラーが発生します。

上記のいずれの既定も適用されない場合は、SQLCODE -315 エラーが発行されます。

外部キー列および外部キーに対して生成された制約名として定義されているテーブルの列をリストする方法は、“[\[カタログの詳細\] の \[制約\] オプション](#)” を参照してください。

次の例に示すように、クラス定義では、親テーブルの IDKEY プロパティに基づく列を含む外部キーを指定できます。

```
ForeignKey Claim(CheckWriterPost.Hmo,Id,Claim) References SQLUser.Claim.Claim(DBMSKeyIndex);
```

子の外部キーに定義された親列は親クラスの **IDKEY インデックス**の一部である必要があるため、このタイプの外部キーでサポートされている[参照アクション](#)は NO ACTION のみです。

- 存在しないテーブルを外部キーで参照すると、SQLCODE -310 エラーが発行され、%msg に補足情報が示されます。
- 存在しない列を外部キーで参照すると、SQLCODE -316 エラーが発行され、%msg に補足情報が示されます。
- 一意ではない列を外部キーで参照すると、SQLCODE -314 エラーが発行され、%msg に補足情報が示されます。

外部キー列が 1 つの列を参照する場合は、2 つの列で、データ型と列のデータ制約が同じである必要があります。

親子リレーションシップでは、子の順序は定義されていません。アプリケーション・コードは特定の順序に依存しないでください。

読み取り専用としてマウントされたデータベースのクラスを参照する外部キー制約を定義できます。FOREIGN KEY を定義するには、ユーザは、参照されるテーブルまたは参照されるテーブルの列の **REFERENCES 特権**を持っている必要があります。REFERENCES 特権は、ダイナミック SQL または xDBC を介して CREATE TABLE を実行する場合に必要なになります。

### シャード・テーブルと外部キー

キー・テーブルがシャード化され、外部キー・テーブルがシャード化されていない場合、キー・テーブルがシャード化されておらず、外部キー・テーブルがシャード化されている場合、キー・テーブルと外部キー・テーブルの両方がシャード化されている場合など、シャード・テーブルとシャード化されていないテーブルのあらゆる組み合わせについて、外部キーがサポートされています。参照されるテーブルのキーはシャード・キーでも別のキーでもかまいません。外部キーは 1 つの列でも、複数の列でもかまいません。

NO ACTION は、シャード・テーブルでサポートされている唯一の[参照アクション](#)です。

詳細は、“[シャード・クラスタにおけるクエリ](#)” を参照してください。

### 暗黙的な外部キー

すべての外部キーは明示的に定義することをお勧めします。明示的な外部キーが定義されている場合、InterSystems IRIS はこの制約を報告し、暗黙的な外部キー制約は定義されません。

ただし、暗黙的な外部キーを ODBC/JDBC および管理ポータルに投影することもできます。これらの暗黙的な外部キーは、NO ACTION の UPDATE および DELETE [参照アクション](#)として報告されます。参照アクションが適用されていないため、この暗黙的な参照外部キーは、真の外部キーではありません。この参照に対して報告されるこの外部キーの名前は、“IMPLICIT\_FKEY\_REFERENCE\_”\_columnname となります。この参照を外部キーとして報告する動作は、サードパーティ製ツールとの相互運用性のために提供されています。

## ビットマップ・エクステント・インデックス

CREATE TABLE を使用してテーブルを作成する場合、既定では、InterSystems IRIS は自動的に対応するクラスの **ビットマップ・エクステント・インデックス** を定義します。ビットマップ・エクステント・インデックスの SQL マップ名は、`%%DDLBEIndex` です。

```
Index DDLBEIndex [ Extent, SqlName = "%%DDLBEIndex", Type = bitmap ];
```

このビットマップ・エクステント・インデックスは、以下の状況では作成されません。

- ・ テーブルはグローバル一時テーブル (CREATE TABLE GLOBAL TEMPORARY TABLE ...) として定義されます。
- ・ テーブルが明示的な **IDKEY インデックス** を定義している。
- ・ テーブルに、MINVAL=1 が指定されていない、定義済みの **IDENTITY 列** がある。
- ・ \$SYSTEM.SQL.Util.SetOption() メソッドの DDLDefineBitmapExtent オプションが、既定値をシステム全体でオーバーライドするように 0 に設定されている。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() メソッドを呼び出します。これにより、[DDL CREATE TABLE ] 設定が表示されます。

ビットマップ・インデックスを作成した後に、**CREATE BITMAPEXTENT INDEX** コマンドを、ビットマップ・エクステント・インデックスが自動的に定義されたテーブルに対して実行すると、それ以前に定義されているビットマップ・エクステント・インデックスは、CREATE BITMAPEXTENT INDEX 文により指定された名前に変更されます。

既存のビットマップ・エクステント・インデックスを自動的に削除する DDL 操作については、“**ALTER TABLE**” を参照してください。

詳細は、“**ビットマップ・エクステント・インデックス**” を参照してください。

## IDENTITY キーワードを使用した名前付き RowID 列の作成

InterSystems SQL では、テーブルごとに RowID 列が自動的に作成されます。この列には、一意のレコード ID となるシステム生成の整数が格納されます。オプションの IDENTITY キーワードを使用すると、RowID レコード ID 列と同じプロパティを持つ名前付きの列を定義できます。IDENTITY 列は、システムが生成する一意の整数値を値として持つ単一の IDKEY インデックスとして動作します。

IDENTITY 列を定義することにより、**IDKEY としての主キー** の定義が回避されます。

システムが生成するあらゆる ID 列と同様に、IDENTITY 列は以下の特性を持ちます。

- ・ IDENTITY 列として定義できるのは、テーブルごとに 1 つの列のみです。複数の IDENTITY 列を定義しようとすると、SQLCODE -308 エラーが発生します。
- ・ IDENTITY 列のデータ型は整数データ型であることが必要です。データ型を指定しないと、データ型は自動的に BIGINT として定義されます。INTEGER や SMALLINT など、任意の整数データ型を指定できます。RowID のデータ型と一致するように BIGINT を使用することをお勧めします。NOT NULL や UNIQUE のような列制約は受け入れられませんが無視されます。
- ・ データ値はシステムが生成します。値は一意で、0 以外の正の整数です。
- ・ 既定では、IDENTITY 列のデータ値をユーザが指定することはできません。既定では、INSERT 文は IDENTITY 列の値を指定しません。指定しようとしてもできません。これを実行しようとすると、SQLCODE -111 エラーが生成されます。IDENTITY 列の値を指定できるかどうかを確認するには、\$SYSTEM.SQL.Util.GetOption("IdentityInsert") メソッドを呼び出します。既定値は 0 です。現在のプロセスに対してこの設定を変更するには、\$SYSTEM.SQL.Util.SetOption() メソッドを SET status=\$SYSTEM.SQL.Util.SetOption("IdentityInsert", 1, .oldval) のように呼び出します。テーブル定義で %CLASSPARAMETER ALLOWIDENTITYINSERT=1 を指定することもできます。ALLOWIDENTITYINSERT=1 を指定すると、SetOption("IdentityInsert") を使用して適用された設定がオーバーライドされます。詳細は、“**INSERT**” 文を参照してください。



- ・ IDENTITY 列のデータ値を UPDATE 文で変更することはできません。これを実行しようとする、SQLCODE -107 エラーが生成されます。
- ・ システムが自動的に、IDENTITY 列上の主キーを ODBC および JDBC に投影します。CREATE TABLE 文または ALTER TABLE 文によって、IDENTITY 列または IDENTITY 列を含む列セット上に主キー制約または一意の制約を定義すると、その制約定義は無視されます。したがって、対応する主キーまたは一意のインデックス定義が作成されることはありません。
- ・ SELECT \* 文はテーブルの IDENTITY 列を返します。

INSERT、UPDATE、または DELETE 操作に続いて [LAST\\_IDENTITY](#) 関数を使用すると、直前に変更したレコードの IDENTITY 列の値を返すことができます。IDENTITY 列が定義されていない場合、LAST\_IDENTITY は直前に変更したレコードの RowID 値を返します。

これらの SQL 文は、IDENTITY 列を持つテーブルを作成し、そのテーブルに行を挿入し、作成されたテーブルに対して IDENTITY 列の値を生成します。

## SQL

```
CREATE TABLE Employee (
    EmpNum INT NOT NULL,
    MyID IDENTITY NOT NULL,
    Name VARCHAR(30) NOT NULL,
    CONSTRAINT EmployeePK PRIMARY KEY (EmpNum))
```

## SQL

```
INSERT INTO Employee (EmpNum,Name)
SELECT ID,Name FROM SQLUser.Person WHERE Age >= '25'
```

この場合、主キー EmpNum は別のテーブルの ID 列から取得されます。EmpNum の値は一意の整数になりますが、WHERE 節のため、この列の数字が連続しない場合もあります。IDENTITY 列の MyID は、ユーザから可視の一意の連続した整数を、各レコードに割り当てます。

## シャード・テーブルの制限事項

シャード・テーブルを定義する際は、以下の制限があることに留意してください。

- ・ シャード・テーブルはシャード環境でのみ使用できます。シャード化されていないテーブルは、シャード環境またはシャード化されていない環境で使用できます。すべてのテーブルがシャーディングに適した候補とは限りません。シャード環境で最適なパフォーマンスを得るには、一般に、シャード・テーブル（一般的に非常に大きなテーブル）とシャード化されていないテーブルの組み合わせを使用します。詳細は、“[シャーディングの効果の評価](#)” および “[シャーディングに関する既存のテーブルの評価](#)” を参照してください。
- ・ CREATE TABLE または [永続クラス定義](#) のいずれかを使用して、シャード・テーブルとしてテーブルを定義する必要があります。ALTER TABLE を使用して、シャード・キーを既存のテーブルに追加することはできません。
- ・ シャード・キーが一意キーのサブセットでない限り、シャード・テーブル列の一意制約によって、挿入/更新のパフォーマンスに重大な悪影響が生じることがあります。詳細は、“[シャーディングによるデータ量に応じた InterSystems IRIS の水平方向の拡張](#)” にある “[一意制約の評価](#)” を参照してください。
- ・ アトミック性を必要とする複雑なトランザクションに含まれるテーブルをシャード化することはお勧めしません。
- ・ シャード・テーブルに ROWVERSION データ型または SERIAL (%Library.Counter) データ型の列を含めることはできません。
- ・ シャード・テーブルでは、VERSIONPROPERTY クラス・パラメータを指定することはできません。
- ・ シャード・キーを指定するには、現在のネームスペースがシャーディング用に構成されている必要があります。現在のネームスペースがシャーディング用に構成されていない場合、シャード・キーを指定した CREATE TABLE は SQLCODE -400 エラーで失敗します。シャーディング用のネームスペースの構成の詳細は、“[シャード・マスタ・データ・サーバの構成](#)” を参照してください。

- ・ [シャード・テーブル](#)でサポートされている唯一の[参照アクション](#)は、NO ACTION です。他の参照アクションはすべて、SQLCODE -400 エラーになります。
- ・ シャード・キー列では、%EXACT、%SQLSTRING、または %SQLUPPER 照合のみを使用でき、切り捨ては行われません。詳細は、["シャード・クラスタにおけるクエリ"](#)を参照してください。

シャードの詳細は、["ターゲット・シャード・テーブルの作成"](#)を参照してください。

## レガシー・オプション

### %EXTENTSIZE キーワードと %NUMROWS キーワード

%EXTENTSIZE キーワードと %NUMROWS キーワードは、作成中のテーブルに想定される行数を格納するためのオプションを提供します。InterSystems SQL クエリ・オブティマイザは、この値を使用してクエリ・プランのコストを見積もります。テーブルでこれらの値のいずれかを定義できますが、両方はできません。例：

#### SQL

```
CREATE TABLE Sample.DaysInAYear (
    %EXTENTSIZE 366,
    MonthName VARCHAR(24),
    Day INTEGER)
```

2021 年 2 月以降、テーブルに対して初めてクエリを実行するときに、InterSystems IRIS はテーブル・サイズなどの統計を自動的に収集します。SQL クエリ・オブティマイザはこれらの生成された統計を使用してクエリ・プランを提案するため、%EXTENTSIZE キーワードと %NUMROWS キーワードが不要になります。テーブル統計を使用したテーブルの最適化の詳細は、["クエリ・オブティマイザで使用するテーブル統計"](#)を参照してください。

### %FILE キーワード

%FILE キーワードは、テーブルを示すファイル名を指定するオプションを提供します。例：

#### SQL

```
CREATE TABLE Employee (
    %FILE 'C:\SQL\employee_table_desc.txt',
    EmpNum INT PRIMARY KEY,
    NameLast VARCHAR(30) NOT NULL,
    NameFirst VARCHAR(30) NOT NULL,
    StartDate TIMESTAMP %Description 'MM/DD/YY')
```

このキーワードは推奨されません。代わりに、[%DESCRIPTION](#) キーワードを使用してテーブルについて説明します。

### 列リストの括弧内のシャード・キーと %CLASSPARAMETER

古い CREATE TABLE コードには、シャード・キー定義と %CLASSPARAMETER 節が、テーブル要素を含む括弧内にコメントで区切られた要素として含まれる場合があります。例：CREATE TABLE myTable(Name VARCHAR(50), DOB DATE, %CLASSPARAMETER USEEXTENTSET = 1)。これらの節を閉じ括弧の後に指定する構文を推奨します。例：CREATE TABLE myTable(Name VARCHAR(50), DOB TIMESTAMP) WITH %CLASSPARAMETER USEEXTENTSET = 1。これらの節を重複して指定すると、SQLCODE -327 エラーが生成されます。

### 互換性のみにサポートされるオプション

InterSystems SQL は、構文解析の目的にのみ以下の CREATE TABLE オプションをサポートし、既存の SQL コードから InterSystems SQL への変換を支援します。これらのオプションは、実際には機能しません。

```
{ON | IN} dbspace-name LOCK MODE [ROW | PAGE] [CLUSTERED | NONCLUSTERED] WITH FILLFACTOR = literal
MATCH [FULL | PARTIAL] CHARACTER SET identifier COLLATE identifier /* But COLLATE keyword is still
used*/ NOT FOR REPLICATION
```

## 関連項目

- ・ [ALTER TABLE](#)、[DROP TABLE](#)

- ・ [SELECT、JOIN](#)
- ・ [INSERT、UPDATE、INSERT OR UPDATE](#)
- ・ [GRANT](#)
- ・ [テーブルの定義](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

## CREATE TABLE AS SELECT (SQL)

列定義と列データを既存のテーブルから新しいテーブルへコピーします。

### 構文

```
CREATE TABLE table-name AS query [shard-key] [WITH table-option]
```

### 引数

table-name	作成するテーブルの名前。有効な識別子を指定します。テーブル名は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、既定のスキーマ名が使用されます。
query	新しいテーブルに列定義と列データを提供する SELECT クエリ。このクエリは、テーブル、ビュー、または複数の結合テーブルを指定できます。ただし、通常の SELECT 文のような？パラメータは指定できません。
shard-key	オプション - シャード・キー定義。SHARD キーワード単独で構成するか、後にシャード・キー定義構文が続きます。
WITH table-option	オプション - 1 つ以上のテーブル・オプション (例えば、%CLASSPARAMETER キーワードと、それに続く名前および関連するリテラル) のコンマ区切りリスト。

### 説明

CREATE TABLE AS SELECT コマンドは、SELECT クエリで指定したように、既存のテーブルの列定義と列データをコピーすることにより、新しいテーブルを作成します。SELECT クエリでは、テーブルまたはビューの任意の組み合わせを指定できます。

注釈 CREATE TABLE AS SELECT では、既存のテーブル定義からコピーします。新しいテーブル定義を指定するには、CREATE TABLE コマンドを使用します。

テーブルのコピー操作は、QueryToTable() メソッド呼び出しを使用して呼び出すこともできます。

```
DO $SYSTEM.SQL.Schema.QueryToTable(query,table-name,0)
```

### データ定義のコピー

- CREATE TABLE AS SELECT は、query テーブルから列定義をコピーします。コピーした列の名前を変更するには、query で列エイリアスを指定します。  
query で結合テーブルを指定する場合、CREATE TABLE AS SELECT は複数のテーブルから列定義をコピーできます。
- CREATE TABLE AS SELECT は、常に RowID を非公開として定義します。
  - ソース・テーブルに非公開の RowID がある場合、CREATE TABLE AS SELECT はソース・テーブルの RowID をコピーしませんが、作成したテーブルに新しい RowID 列を作成します。コピーされた行には、新しい一連の RowID 値が割り当てられます。
  - ソース・テーブルに公開された (非表示ではない) RowID がある場合、またはクエリで明示的に非公開の RowID を選択している場合、CREATE TABLE AS SELECT はこのテーブルに新しい RowID 列を作成します。ソース・テーブルの RowID は、非公開でも、一意でも、必須でもない通常の BigInt フィールドとして新しいテーブルにコピーされます。ソース・テーブルの RowID を "ID" と名付けた場合、新しいテーブルの RowID の名前は "ID1" となります。

- ・ ソース・テーブルに **IDENTITY フィールド**がある場合、CREATE TABLE AS SELECT はこのフィールドとその現在のデータを、ゼロでない正の整数の、一意でも必須でもない通常の BIGINT フィールドとしてコピーします。
- ・ CREATE TABLE AS SELECT は、IDKEY インデックスを定義します。これはコピーされた列定義に関連付けられたインデックスをコピーしません。
- ・ CREATE TABLE AS SELECT は、列の制約をコピーしません。これはコピーされた列定義に関連付けられた NULL/NOT NULL、UNIQUE、主キー、または外部キーの制約をコピーしません。
- ・ CREATE TABLE AS SELECT は、既定の制約またはコピーされた列定義に関連付けられた値をコピーしません。
- ・ CREATE TABLE AS SELECT は、コピーされた列定義に関連付けられた COMPUTECODE データ制約をコピーしません。
- ・ CREATE TABLE AS SELECT は、コピーされたテーブルまたは列定義に関連付けられた %DESCRIPTION 文字列をコピーしません。

## 特権

CREATE TABLE AS SELECT コマンドは特権を必要とする操作です。CREATE TABLE AS SELECT を実行するには、ユーザは %CREATE\_TABLE **管理特権**を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %CREATE\_TABLE " に設定されます。適切な付与特権を持っていれば、**GRANT** コマンドを使用して、ユーザまたはロールに %CREATE\_TABLE 特権を割り当てることができます。管理特権はネームスペース固有のものです。詳細は、“**特権**”を参照してください。

ユーザは、query で指定されたテーブルに対する SELECT 特権を持っている必要があります。

## テーブル名

テーブル名は修飾、未修飾のどちらでもかまいません。

- ・ 未修飾のテーブル名には、tablename という構文を使用します。schema (およびピリオド(.) 文字) は省略します。テーブル名が未修飾の場合は、**既定のスキーマ名**が使用されます。初期のシステム全体の既定のスキーマ名は、既定のクラス・パッケージ名 User に対応する SQLUser です。スキーマ検索パスの値は無視されます。

**既定のスキーマ名は構成可能です。**

現在のシステム全体の既定のスキーマ名を確認するには、\$SYSTEM.SQL.Schema.Default() メソッドを使用します。

- ・ テーブル名を修飾する場合は、schema.tablename という構文を使用します。既存のスキーマ名を指定することも、新規スキーマ名を指定することもできます。既存のスキーマ名を指定すると、テーブルはスキーマ内に配置されます。新規スキーマ名を指定すると、その名前のスキーマ (および関連するクラス・パッケージ) が作成され、テーブルがそのスキーマ内に配置されます。

テーブル名とスキーマ名は、**SQL 識別子**の名前付け規約と、非英数文字の使用、一意性、および最大長に関する追加の制約に従います。% 文字で始まる名前は、システムによる使用のために予約されています。既定では、スキーマ名とテーブル名は単純な識別子であり、大文字と小文字が区別されません。

InterSystems IRIS はテーブル名を使用して、対応するクラス名を生成します。InterSystems IRIS はスキーマ名を使用して、対応するクラス・パッケージ名を生成します。クラス名には英数字 (文字および数字) のみを使用し、最初の 96 文字は一意である必要があります。クラス名を生成するために、InterSystems IRIS はまず記号 (非英数字) 文字をテーブル名から削除し、一意性と最大長の制約を課して一意のクラス名を生成します。パッケージ名を生成するために、InterSystems IRIS はスキーマ名内の記号 (非英数字) 文字に対して削除または特殊処理のいずれかを実行します。次に InterSystems IRIS は、一意性と最大長の制約を課して一意のパッケージ名を生成します。スキーマ名とテーブル名からパッケージ名とクラス名を生成する方法の詳細は、“**テーブル名とスキーマ名**”を参照してください。

スキーマとテーブルに同じ名前を使用することはできません。同じスキーマ内でテーブルとビューに同じ名前を使用することはできません。

スキーマ名では大文字と小文字が区別されず、対応するクラス・パッケージ名では大文字と小文字が区別されます。指定したスキーマ名が、既存のクラス・パッケージ名と大文字/小文字のみが異なり、パッケージ定義が空の(クラス定義が含まれていない) 場合、InterSystems IRIS では、クラス・パッケージ名の大文字/小文字を変更して 2 つの名前が照合されます。スキーマ名の詳細は、“[テーブル名とスキーマ名](#)”を参照してください。

InterSystems IRIS は 16 ビット(ワイド) 文字のテーブル名と列名をサポートします。大部分のロケールで、テーブル名にはアクセント記号付き文字を使用でき、生成されるクラス名にもアクセント記号が含まれます。以下の例は、SQL テーブル名の検証テストを実行します。

### ObjectScript

```
TableNameValidation
SET tname="MyTestTableName"
SET x=$SYSTEM.SQL.IsValidRegularIdentifier(tname)
IF x=0 {IF $LENGTH(tname)>200
    {WRITE "Tablename is too long" QUIT}
ELSEIF $SYSTEM.SQL.IsReservedWord(tname)
    {WRITE "Tablename is reserved word" QUIT}
ELSE {
    WRITE "Tablename contains invalid characters",!
    SET nls=##class(%SYS.NLS.Locale).%New()
    IF nls.Language [ "Japanese" {
        WRITE "Japanese locale cannot use accented letters"
        QUIT }
    QUIT }
}
ELSE { WRITE tname," is a valid table name"}
```

注釈 日本ロケールでは、アクセント記号付きの文字を識別子でサポートしていません。日本語の識別子には、日本語の文字に加え、ラテン文字の A ～ Z と a ～ z (65 ～ 90 および 97 ～ 122)、アンダースコア文字 (95)、とギリシャ文字の大文字 (913 ～ 929 および 931 ～ 937) を使用できます。nls.Language テストでは、= ではなく [ (包括関係演算子) を使用しています。オペレーティング・システム・プラットフォームごとに日本語ロケールが異なるためです。

### 既存のテーブル

テーブルが現在のネームスペースに既に存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.TableExists("schema.tname") を使用します。

既定では、既存のテーブルと同じ名前インテーブルを作成しようとすると、InterSystems IRIS はテーブルの作成試行を拒否し、SQLCODE -201 エラーを発行します。現在のシステム全体の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[ DDL CREATE TABLE CREATE VIEW ] 設定が表示されます。既定値は 0 で、この設定を推奨します。このオプションを 1 に設定すると、InterSystems IRIS はこのテーブルに対応するクラス定義を削除し、クラスを再作成します。つまり、DROP TABLE を実行して既存のテーブルを削除し、CREATE TABLE を実行する動作と同じ結果を生じます。この場合、\$SYSTEM.SQL.CurrentSettings() で [ DDL DROP TABLE ] の値を 1 (既定) に設定することを強くお勧めします。詳細は、“[DROP TABLE](#)”を参照してください。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション (および他の同様の作成、変更、および削除のオプション) をシステム全体で設定できます。

### WITH テーブル・オプション

オプションの WITH 節は、SELECT 節の後に指定できます。WITH 節には、%CLASSPARAMETER 節のコンマ区切りリストを含めることができます。

%CLASSPARAMETER キーワードを使用すると、クラス・パラメータを CREATE TABLE AS SELECT コマンドの一部として定義できます。クラス・パラメータは必ず定数値として定義します。%CLASSPARAMETER キーワードの後には、そのクラス・パラメータに割り当てるクラス・パラメータ名、オプションの等号、およびリテラル値 (文字列または数値) を指定します。



複数の %CLASSPARAMETER キーワード節を指定できます。1 つの節につき 1 つのクラス・パラメータを定義します。複数の CLASSPARAMETER 句はコンマで区切ります。

例えば、既定で CREATE TABLE AS SELECT はグローバル名が生成されている作成済みのテーブルの IDKEY インデックス (^EPgS.D8T6.1 など)を作成します。追加のインデックスは、一意の整数の接尾語を付けた同じグローバル名を使用します。以下の例は、[IDKEY インデックスの明示的なグローバル名](#)と将来的に追加されるインデックスの指定方法を示します。

```
CREATE TABLE Sample.YoungPeople
AS SELECT Name, Age
FROM Sample.People
WHERE Age < 21
WITH %CLASSPARAMETER DEFAULTGLOBAL = '^GL.UNDERTWENTYONE'
```

詳細は、CREATE TABLE のリファレンス・ページの [“WITH 節と %CLASSPARAMETER キーワード”](#) を参照してください。

## 関連項目

- ・ [CREATE TABLE、ALTER TABLE、DROP TABLE](#)
- ・ [SELECT、JOIN](#)
- ・ [GRANT](#)
- ・ [テーブルの定義](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)



## CREATE TRIGGER (SQL)

トリガを作成します。

### 構文

```
CREATE [OR REPLACE] TRIGGER trigname {BEFORE | AFTER} event [,event]
  [ORDER integer] ON table
  [REFERENCING {OLD | NEW} [ROW] [AS] alias] action
```

### 引数

引数	説明
<i>trigname</i>	作成するトリガの名前。識別子です。トリガ名は修飾、未修飾のどちらでもかまいません。修飾する場合、そのスキーマ名はテーブルのスキーマ名と同じである必要があります。
BEFORE <i>event</i> AFTER <i>event</i>	<i>event</i> でトリガを実行するタイミング (BEFORE または AFTER)。  トリガ・イベント、またはトリガ・イベントのコンマ区切りリスト。指定できる <i>event</i> リスト・オプションは、INSERT、DELETE、および UPDATE です。  指定できる UPDATE OF <i>event</i> . は 1 つです。UPDATE OF 節の後には、列名またはコンマで区切られた列名のリストが続きます。UPDATE OF 節は、LANGUAGE が SQL の場合にのみ指定できます。UPDATE OF 節は、コンマ区切りの <i>event</i> リストでは指定できません。
ORDER <i>integer</i>	オプション — 同じタイミングとイベントのトリガがテーブルに複数存在する場合に、それらのトリガを実行する順番。順番の指定を省略したトリガには、順番 0 が割り当てられます。
ON <i>table</i>	トリガが作成されるテーブル。テーブル名は修飾、未修飾のどちらでもかまいません。修飾する場合は、トリガがテーブルと同じスキーマにある必要があります。
REFERENCING OLD ROW AS <i>alias</i> REFERENCING NEW ROW AS <i>alias</i>	オプション — REFERENCING 節は、LANGUAGE が SQL の場合にのみ使用できます。REFERENCING 節を使用して、列の参照に使用できるエイリアスを指定できます。 REFERENCING OLD ROW を指定すると、UPDATE または DELETE トリガの実行中に、列の元の値を参照できます。 REFERENCING NEW ROW を指定すると、INSERT または UPDATE トリガの実行中に、列の新しい値を参照できます。 ROW AS キーワードは省略可能です。UPDATE では、REFERENCING OLD <i>oldalias</i> NEW <i>newalias</i> のように、OLD と NEW の両方を同じ REFERENCING 節で指定できます。

引数	説明
action	トリガのプログラム・コード。action 引数には、 <b>FOR EACH 節</b> 、トリガされるアクションの実行を制御する述語条件を含む <b>WHEN 節</b> 、および LANGUAGE SQL または LANGUAGE OBJECTSCRIPT のいずれかを指定する <b>LANGUAGE 節</b> をこの順序で並べた、各種オプション・キーワード節を含めることができます。LANGUAGE 節を省略する場合は、SQL が既定です。これらの節の後に、トリガが実行されたときに実行するアクションを指定する <b>SQL トリガ・コード</b> または <b>ObjectScript トリガ・コード</b> を 1 行以上記述します。

## 概要

CREATE TRIGGER コマンドは、トリガ、つまり、指定のテーブルのデータが変更されたときに実行されるコードのブロックを定義します。指定のテーブルへの新しい行の挿入など、具体的なトリガ・イベントが発生すると、トリガが実行（“起動”または“プル”）されます。トリガによって、ユーザが指定したトリガ・コードが実行されます。ユーザは、トリガ・イベントの前または後にそのコードを実行するようにトリガを指定できます。トリガは指定のテーブルに固有です。

- トリガは、指定した event、つまり INSERT、DELETE、または UPDATE のいずれかの操作で起動できます。events のコンマ区切りのリストを指定すると、指定したいいずれかのイベントが指定のテーブルで発生したときにトリガを実行できます。
- トリガは、event によって複数回起動されることも、1 回だけ起動されることもあります。行レベルのトリガは変更した行ごとに 1 回起動されます。文レベルのトリガは event に対して 1 回起動されます。このトリガ・タイプは **FOR EACH 節**を使用して指定します。行レベルのトリガが既定のトリガ・タイプです。
- 一般的には、トリガ・コードを起動すると、ログ操作の実行やメッセージの表示など、別のテーブルまたはファイルに対する操作が実行されます。トリガを起動することでトリガ・レコードのデータを変更することはできません。例えば、レコード 7 の更新によってトリガが起動される場合、そのトリガのコード・ブロックでレコード 7 を更新または削除することはできません。トリガを呼び出したテーブルと同じテーブルをトリガで変更することはできますが、**再帰トリガ**の無限ループを防ぐため、トリガ event とトリガ・コードの処理は異なっている必要があります。

オプションのキーワード OR REPLACE によって、既存のトリガを変更または置換できます。CREATE OR REPLACE TRIGGER には、DROP TRIGGER を呼び出して古いバージョンのトリガを削除し、続いて CREATE TRIGGER を呼び出す操作と同じ効果があります。**DROP TABLE** コマンドを実行すると、そのテーブルに関連付けられているトリガはすべて削除されます。

## 特権とロック

CREATE TRIGGER コマンドの実行には特権が必要です。CREATE TRIGGER を実行するには、ユーザは %CREATE\_TRIGGER 管理特権を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %CREATE\_TRIGGER " に設定されます。

ユーザは、指定されたテーブルに対する %ALTER 特権を持っている必要があります。ユーザがテーブルの所有者（作成者）である場合、ユーザにはそのテーブルに対する %ALTER 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する %ALTER 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' 'Schema.TableName' %ALTER " に設定されます。

適切な付与特権を持っている場合は、**GRANT** コマンドを使用して %CREATE\_TRIGGER 特権および %ALTER 特権を割り当てることができます。

埋め込み SQL では、以下のように \$SYSTEM.Security.Login() メソッドを使用して適切な特権を持ったユーザとしてログインできます。

## ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“インターシステムズ・クラス・リファレンス”の“%SYSTEM.Security”を参照してください。

- CREATE TRIGGER は、テーブル・クラスの定義に [DdlAllowed] が含まれている場合を除き、[永続クラスから投影されたテーブル](#)では使用できません。使用すると、操作は SQLCODE -300 エラーで失敗し、%msg が “DDL schema.tablename” に設定されます。
- CREATE TRIGGER は、[導入済みの永続クラス](#)から投影されたテーブルでは使用できません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ classname DDL ” に設定されます。

CREATE TRIGGER 文は table に対してテーブル・レベルのロックを取得します。これにより、他のプロセスはこのテーブルのデータを変更できなくなります。このロックは CREATE TRIGGER 処理が終了すると自動的に解除されます。

トリガを作成する場合、そのテーブルを別のプロセスによって EXCLUSIVE MODE または SHARE MODE でロックしないでください。ロックされているテーブルで CREATE TRIGGER 操作を実行しようとすると、SQLCODE -110 エラーになり、%msg が “ 'Sample.MyTest' ” に設定されます。

## その他のトリガ定義方法

SQL トリガをクラス・オブジェクトとして定義できます。“[トリガ定義](#)”を参照してください。以下はオブジェクト・トリガの例です。

## Class Member

```
Trigger SQLJournal [ CodeMode = objectgenerator, Event = INSERT/UPDATE, ForEach = ROW/OBJECT, Time = AFTER ]
{ /* ObjectScript trigger code
   that updates a journal file
   after a row is inserted or updated. */
}
```

## 引数

## trigname

トリガ名にはテーブル名と同様の[識別子](#)要件があります。しかし一意性の要件は異なります。トリガ名はスキーマ内のすべてのテーブルに対して一意である必要があります。したがって、スキーマ内の異なるテーブルを参照しているトリガを同じ名前にすることはできません。この一意性の要件に違反すると、DROP TRIGGER エラーが発生する可能性があります。

トリガとそのトリガが関連するテーブルは同じスキーマ内に存在している必要があります。同じスキーマ内では、トリガとテーブルに同じ名前を使用できません。トリガの名前付け規約に違反すると、CREATE TRIGGER の実行時に SQLCODE -400 エラーが発生します。

トリガ名は、修飾、未修飾のどちらでもかまいません。修飾されたトリガ名は次のようになります。

```
schema.trigger
```

トリガ名が未修飾の場合、トリガのスキーマ名は、既定で、指定したテーブルのスキーマと同じスキーマになります。テーブル名が未修飾の場合、テーブルのスキーマ名は、既定で、指定したトリガのスキーマと同じスキーマになります。両方が未修飾の場合は、[既定のスキーマ名](#)が使用されます。スキーマ検索パスは使用されません。両方を修飾する場合は、トリガのスキーマ名がテーブルのスキーマ名と同じである必要があります。スキーマ名が一致しないと SQLCODE -366 エラーになります。これはトリガ名とテーブル名の両方が修飾されていて、異なるスキーマ名を指定しているときのみ起こります。

トリガ名は、識別子の規則に従い、以下のような制約を受けます。既定のトリガ名は、簡単な識別子です。トリガ名は 128 文字を超えることはできません。トリガ名は、大文字と小文字が区別されません。

InterSystems IRIS では、trigname を使用して InterSystems IRIS クラスの対応するトリガ名を生成します。対応するクラス・トリガ名には英数字 (文字と数字) のみを使用でき、最大長は 96 文字です。この識別子名を生成するために、InterSystems IRIS は最初にトリガ名から句読点を削除し、次に 96 文字 (未満) の一意の識別子を生成します。その際、トリガ名の一意性を維持するために、必要に応じて最後の文字を数字に置き換えます。トリガの名前を付ける際には、この名前生成に伴う以下の制約について考慮する必要があります。

- ・ トリガ名には、最低でも 1 文字を含める必要があります。トリガ名の先頭の文字または最初の句読点に続く文字は、数字以外の文字にする必要があります。
- ・ InterSystems IRIS は 16 ビット (ワイド) 文字のトリガ名をサポートします。`$ZNAME` テストに合格した文字は、有効な文字です。
- ・ InterSystems IRIS クラスに対して生成される名前には句読点が含まれないため、句読点のみが異なるトリガ名の作成は可能であっても、お勧めできません。
- ・ トリガ名は 96 文字よりも大幅に長くすることができますが、最初の 96 の英数文字が異なるようにトリガ名を作成すると処理はるかに容易になります。

既存のトリガの名前を使用して CREATE TRIGGER を発行すると、SQLCODE -365 “トリガ名がユニークではありません” エラーが発行されます。オプションの OR REPLACE キーワードを使用するか、先に DROP TRIGGER で古いトリガを削除します。

1 つのスキーマ内の異なるテーブルを参照している 2 つのトリガが同じ名前である場合、DROP TRIGGER を実行すると、SQLCODE -365 “トリガ名がユニークではありません” エラーが、メッセージ “トリガ 'MyTrigName' が 2 つのクラスで見つかりました” と共に表示されることがあります。

## event

トリガ起動のタイミングは、BEFORE キーワードまたは AFTER キーワードで指定します。これらのキーワードで、トリガ操作をトリガの event の発生前に実行するか、発生後に実行するかを指定します。BEFORE トリガが実行される時点は、指定した event の実行前で、その event の検証が終了した後です。例えば、BEFORE DELETE トリガを実行できるのは、DELETE 文が指定の行に対して有効で、DELETE の実行 (外部キー参照の整合性チェックなど) に必要な特権がプロセスに与えられている場合のみです。指定した event をプロセスで実行できない場合は、event に対してエラー・コードが発行され、BEFORE トリガは実行できません。

BEFORE キーワードまたは AFTER キーワードの後に、トリガ・イベントの名前、またはトリガ・イベントのコンマ区切りリストを続けます。INSERT として指定したトリガは、指定のテーブルに行を挿入したときに実行されます。DELETE として指定したトリガは、指定のテーブルから行を削除したときに実行されます。UPDATE として指定したトリガは、指定のテーブルの行を更新したときに実行されます。1 つのトリガ・イベントを指定することも、INSERT、UPDATE、または DELETE の各トリガ・イベントを任意の順序で記述したコンマ区切りリストを指定することもできます。

UPDATE OF として指定したトリガは、指定のテーブルにある行で 1 つ以上の指定した列を更新したときにのみ実行されます。列名はコンマ区切りのリストで指定します。列名は任意の順序で指定できます。UPDATE OF トリガには以下の制限があります。

- ・ UPDATE OF は、トリガ・コードの言語が SQL (既定) の場合にのみ有効です。トリガ・コードの言語が OBJECTSCRIPT の場合、SQLCODE -50 エラーが発行されます。
- ・ UPDATE OF を他のトリガ・イベントと組み合わせることはできません。トリガ・イベントのコンマ区切りリストで UPDATE OF を指定すると、SQLCODE -1 エラーが発行されます。
- ・ UPDATE OF で存在しないフィールドを指定することはできません。指定すると、SQLCODE -400 エラーが発行されます。
- ・ UPDATE OF で重複するフィールド名を指定することはできません。指定すると、SQLCODE -58 エラーが発行されます。

以下に、event タイプの例を示します。

## SQL

```
CREATE TRIGGER TrigBI BEFORE INSERT ON Sample.Person
    INSERT INTO TLog (Text) VALUES ('before insert')
```

## SQL

```
CREATE TRIGGER TrigAU AFTER UPDATE ON Sample.Person
    INSERT INTO TLog (Text) VALUES ('after update')
```

## SQL

```
CREATE TRIGGER TrigBUOF BEFORE UPDATE OF Home_Street,Home_City,Home_State ON Sample.Person
    INSERT INTO TLog (Text) VALUES ('before address update')
```

```
CREATE TRIGGER TrigAD AFTER UPDATE,DELETE ON Sample.Person
    INSERT INTO TLog (Text) VALUES ('after update or delete')
```

## ORDER

ORDER 節は、同じタイミングとイベントのトリガが 1 つのテーブルに複数存在する場合に、それらのトリガを実行する順番を指定します。例えば、2 つの AFTER DELETE トリガのような場合です。ORDER の整数値が最小のトリガが最初に実行され、その後は、次に高い整数値を持つトリガが順次実行されます。ORDER 節が指定されない場合は、ORDER 値に 0 (ゼロ) が割り当てられてトリガが生成されます。したがって、ORDER 節を持たないトリガは、必ず ORDER 節を持つトリガの前に実行されます。

複数のトリガに同じ ORDER 値を割り当てることができます。また、ORDER が 0 (明示的または暗黙的) のトリガを複数作成することもできます。タイミング、イベント、および順序が同じ複数のトリガは、任意の順序でまとめて実行されます。

トリガはタイミング、順序、イベントの順に実行されます。BEFORE INSERT トリガと BEFORE INSERT,UPDATE トリガがある場合は、ORDER 値が最も小さいトリガが最初に実行されます。BEFORE INSERT トリガと BEFORE INSERT,UPDATE トリガがあり、ORDER 値が同じ場合は、INSERT が INSERT,UPDATE よりも前に実行されます。タイミングと順序が同じ場合、シングル・イベント・トリガは常にマルチ・イベント・トリガの前に実行されるためです。2 つ (またはそれ以上) のトリガでタイミング、順序、イベントの値が同じ場合は、命令の実行はランダムになります。

以下の例は、ORDER 値がどのように機能するかを示します。以下のどの CREATE TRIGGER 文でも、同じイベントで実行するトリガが作成されます。

## SQL

```
CREATE TRIGGER TrigA BEFORE DELETE ON doctable
    INSERT INTO TLog (Text) VALUES ('doc deleted')
    /* Assigned ORDER=0 */
```

## SQL

```
CREATE TRIGGER TrigB BEFORE DELETE ORDER 4 ON doctable
    INSERT INTO TReport (Text) VALUES ('doc deleted')
    /* Specified as ORDER=4 */
```

## SQL

```
CREATE TRIGGER TrigC BEFORE DELETE ORDER 2 ON doctable
    INSERT INTO Ttemps (Text) VALUES ('doc deleted')
    /* Specified as ORDER=2 */
```

## SQL

```
CREATE TRIGGER TrigD BEFORE DELETE ON doctable
    INSERT INTO Tflags (Text) VALUES ('doc deleted')
    /* Also assigned ORDER=0 */
```



これらのトリガは、(TrigA または TrigD)、TrigC、TrigB の順に実行されます。TrigA と TrigD は同じ ORDER 値となるので、任意の順序で実行されます。

## REFERENCING

REFERENCING 節では、行の元の値、新しい値、またはその両方に対してエイリアスを指定できます。元の値とは、UPDATE または DELETE トリガのトリガ動作が実行される前の行の値のことです。新しい値とは、UPDATE または INSERT トリガのトリガ動作が実行された後の行の値のことです。UPDATE トリガの場合は、以下のようにして、元の行値と新しい値の両方にエイリアスを指定できます。

```
REFERENCING OLD ROW AS oldalias NEW ROW AS newalias
```

キーワードの ROW と AS は省略可能です。したがって、同じ節を以下のように指定することもできます。

```
REFERENCING OLD oldalias NEW newalias
```

INSERT 前の元の値や DELETE 後の新しい値を参照することには、意味がありません。それらを参照すると、コンパイル時に SQLCODE -48 が返されます。

REFERENCING 節は action プログラム・コードが SQL の場合にのみ、使用できます。LANGUAGE OBJECTSCRIPT 節と共に REFERENCING 節を指定すると、SQLCODE -49 エラーになります。

以下は、INSERT と共に REFERENCING を使用する例です。

## SQL

```
CREATE TRIGGER TrigA AFTER INSERT ON doctable
  REFERENCING NEW ROW AS new_row
BEGIN
  INSERT INTO Log_Table VALUES ('INSERT into doctable');
  INSERT INTO New_Log_Table VALUES ('INSERT into doctable',new_row.ID);
END
```

## action

トリガ動作は、以下の要素から構成されます。

- ・ オプションの FOR EACH 節。使用できる値は、FOR EACH ROW、FOR EACH ROW\_AND\_OBJECT、および FOR EACH STATEMENT です。既定は FOR EACH ROW です。
  - FOR EACH ROW – このトリガは、トリガ文の影響を受ける各行によって起動されます。TSQL では、行レベル・トリガはサポートされていないことに注意してください。
  - FOR EACH ROW\_AND\_OBJECT – このトリガは、トリガ文の影響を受ける各行、またはオブジェクト・アクセスによる変更によって起動されます。TSQL では、行レベル・トリガはサポートされていないことに注意してください。

このオプションは、統一トリガを定義します。SQL またはオブジェクト・アクセスにより発生するデータ変更によって起動されるトリガであるため、そのように呼ばれます (対照的に、他のトリガを使用すると、オブジェクト・アクセスによる変更の発生時に同じ論理を使用するには、%OnDelete() などのコールバックを実装する必要があります)。
- FOR EACH STATEMENT – このトリガは、文全体で 1 回起動されます。文レベルのトリガは、ObjectScript トリガと [TSQL トリガ](#) の両方でサポートされています。

対応するトリガ・クラス・オプションについては、“[FOREACH](#)” を参照してください。

**INFORMATION.SCHEMA.TRIGGERS** の **ACTIONORIENTATION** プロパティを使用して、[各トリガの FOR EACH 値をリスト表示](#)できます。

- ・ オプションの WHEN 節。WHEN 節は、WHEN キーワードとその後の括弧で囲まれた (単純または複雑な) 述語条件から成ります。述語条件が True に評価されると、トリガが実行されます。WHEN 節は、LANGUAGE が SQL の場

合にのみ使用できます。WHEN 節は、oldalias 値または newalias 値を参照できます。述語条件式の詳細、および使用可能な述語のリストは、本ドキュメントの“[述語の概要](#)”のページを参照してください。

- ・ オプションの LANGUAGE 節。LANGUAGE SQL または LANGUAGE OBJECTSCRIPT を指定できます。既定は LANGUAGE SQL です。
- ・ トリガが実行されたときに実行するユーザ記述のコードです。

## SQL トリガ・コード

LANGUAGE SQL の場合、既定でトリガ文は SQL プロシージャ・ブロックです。SQL プロシージャ・ブロックは、1 つの SQL プロシージャ文とその後のセミコロン、またはキーワード BEGIN で始まりキーワード END で終わる 1 つ以上の SQL プロシージャ文と各文の後のセミコロンから成ります。

トリガ動作はアトミックで、完全に適用されるかまったく適用されないかのどちらかです。これに COMMIT 文または ROLLBACK 文を含めることはできません。キーワード BEGIN ATOMIC は、キーワード BEGIN と同義です。

LANGUAGE SQL の場合、CREATE TRIGGER 文はオプションで REFERENCING 節、WHEN 節、および/または UPDATE OF 節を含むことができます。UPDATE OF 節は、トリガに指定した 1 つ以上の列で UPDATE が実行されたときにのみ、トリガを実行することを指定します。LANGUAGE OBJECTSCRIPT の CREATE TRIGGER 文では、これらの節を含むことはできません。

SQL トリガ・コードは埋め込み SQL として実行されます。これは、InterSystems IRIS で SQL トリガ・コードが ObjectScript に変換されることを意味します。したがって、SQL トリガ・コードに対応するクラス定義を表示すると、トリガ定義では Language=objectscript が表示されることになります。

SQL トリガ・コードの実行時に、システムは自動的に、トリガ・コード内で使用されているすべての変数をリセット (NEW) します。各 SQL 文の実行後に、InterSystems IRIS は SQLCODE をチェックします。エラーが発生すると、InterSystems IRIS は %ok 変数に 0 をセットし、トリガ・コードの処理および関連する INSERT、UPDATE、または DELETE を中止してロールバックします。

## ObjectScript トリガ・コード

LANGUAGE OBJECTSCRIPT の場合、CREATE TRIGGER 文は REFERENCING 節、WHEN 節、または UPDATE OF 節を含むことはできません。これらの SQL 限定である節を LANGUAGE OBJECTSCRIPT と共に指定すると、それぞれコンパイル時に SQLCODE エラーの -49、-57、または -50 が発生します。

LANGUAGE OBJECTSCRIPT の場合、トリガ文は、中括弧で囲まれた 1 つ以上の ObjectScript 文からなるブロックです。

トリガのコードはプロシージャとして生成されないため、トリガ内のすべてのローカル変数はパブリック変数となります。つまり、トリガ内のすべての変数は NEW 文で明示的に宣言される必要があります。これにより、トリガを呼び出すコード内の変数との競合を避けることができます。

トリガ・コードに[マクロ・プリプロセッサ文](#) (# コマンド、## 関数、または \$\$\$ マクロ参照) が含まれる場合、これらの文は CREATE TRIGGER DDL コード本体より前にコンパイルされます。

ObjectScript トリガ・コードには[埋め込み SQL](#)を含めることができます。

%ok 変数を 0 に設定して、トリガ・コードからエラーを発行できます。これにより、トリガの実行を中止してロール・バックする実行時エラーが生成されます。また、適切な SQLCODE エラー (例えば、SQLCODE -131 “After Insert トリガが失敗しました”) が生成され、トリガ・コード・エラーの原因を説明する文字列として %msg 変数のユーザ指定値が返されます。%ok を数値以外の値に設定すると、%ok=0 に設定されることに注意してください。

トリガ・コードは一度だけ生成されます。これは、マルチ・イベント・トリガの場合でも同様です。

## フィールド参照および擬似フィールド参照

ObjectScript で記述するトリガ・コードには、{fieldname} で指定するフィールド参照を含めることができます。ここで、fieldname は現在のテーブルの既存フィールドを指定します。中括弧内に空白スペースは許可されません。



fieldname の後に \*N (新)、\*O (旧)、または \*C (比較) を付けて、挿入、更新、または削除されたフィールドのデータ値の処理方法を以下のとおりに指定できます。

- ・ {fieldname\*N}
  - UPDATE の場合は、指定された変更が行われた後の新しいフィールド値を返します。
  - INSERT の場合は、挿入した値を返します。
  - DELETE の場合は、削除前のフィールド値を返します。
- ・ {fieldname\*O}
  - UPDATE の場合は、指定された変更が行われる前の元のフィールド値を返します。
  - INSERT の場合は、NULL を返します。
  - DELETE の場合は、削除前のフィールド値を返します。
- ・ {fieldname\*C}
  - UPDATE の場合は、新しい値が古い値と異なる場合に 1 (TRUE) を返します。そうでない場合は 0 (FALSE) を返します。
  - INSERT の場合は、挿入した値が NULL でない場合に 1 (TRUE) を返します。そうでない場合は 0 (FALSE) を返します。
  - DELETE の場合は、削除される値が NULL でない場合に 1 (TRUE) を返します。そうでない場合は 0 (FALSE) を返します。

UPDATE、INSERT、または DELETE では、{fieldname} は {fieldname\*N} と同じ値を返します。

例えば、以下のトリガは、Sample.Employee に新しく挿入された行の Name フィールドの値を返します (SQL シェルから INSERT を実行して、この結果を表示できます)。

```
CREATE TRIGGER InsertNameTrig AFTER INSERT ON Sample.Employee
LANGUAGE OBJECTSCRIPT
{WRITE "The employee ",{Name*N}," was ",{%OPERATION},"ed on ",{%TABLENAME},!}
```

フィールド値を設定する文の中では改行ができません。詳細は、“クラス定義リファレンス”の“[SqlComputeCode](#)”プロパティ・キーワードを参照してください。

GetAllColumns() メソッドを使用すると、テーブルのために定義されているフィールド名をリストできます。詳細は、“[列の名前と番号](#)”を参照してください。

また、ObjectScript で記述するトリガ・コードには、擬似フィールド参照変数の {%CLASSNAME}、{%CLASSNAMEQ}、{%OPERATION}、{%TABLENAME}、および {%ID} を含めることもできます。擬似フィールドは、クラスのコンパイル時に特定の値に変換されます。これらの擬似フィールド・キーワードは、すべて大文字と小文字が区別されません。

- ・ {%CLASSNAME} と {%CLASSNAMEQ} は、ともに SQL テーブル定義を投影するクラスの名前に変換されます。{%CLASSNAME} は引用符なし文字列を返し、{%CLASSNAMEQ} は引用符付き文字列を返します。
- ・ {%OPERATION} は、トリガを呼び出した操作に応じて、文字列リテラル (INSERT、UPDATE、または DELETE のいずれか) に変換されます。
- ・ {%TABLENAME} は [テーブルの完全修飾名](#) に変換されます。
- ・ {%ID} は [RowID 名](#) に変換されます。この参照は、RowID フィールドの名前がわからないときに役立ちます。

## ストリーム・プロパティの参照

{StreamField}、{StreamField\*O}、または {StreamField\*N} など、[ストリーム・フィールド/プロパティ](#)がトリガ定義で参照される場合は、{StreamField} 参照の値がストリームの OID (オブジェクト ID) 値となります。

BEFORE INSERT または BEFORE UPDATE トリガについては、新しい値が INSERT/UPDATE/ObjectSave によって指定される場合、{StreamField\*N} 値が一時的ストリーム・オブジェクトの OID、または新しいリテラル・ストリーム値のどちらかとなります。BEFORE UPDATE トリガについては、新しい値がストリーム・フィールド/プロパティに対して指定されていない場合、{StreamField\*O} および {StreamField\*N} は両方とも現在のフィールド/プロパティ・ストリーム・オブジェクトの OID になります。

## SQLComputed プロパティの参照

一時 SqlComputed フィールド/プロパティ (“Calculated” または明示的な “Transient” のどちらか) がトリガ定義で参照される場合、そのトリガは Get()/Set() メソッドのオーバーライドを認識しません。プロパティの Get() または Set() メソッドをオーバーライドするのではなく、[SQLCOMPUTED/SQLCOMPUTONCHANGE](#) を使用してください。

Get()/Set() メソッドのオーバーライドを使用すると、{property\*O} 値の決定に SQL が使用されて、オーバーライドされた Get()/Set() メソッドが使用されないという誤った結果がもたらされる可能性があります。プロパティがディスク上に格納されないため、{property\*O} では SqlComputeCode が使用されて古い値が “再作成” されます。ただし、{property\*N} は、オーバーライドされた Get()/Set() メソッドを使用してプロパティの値にアクセスします。その結果、プロパティが実際に変更されなかった場合でも、{property\*O} と {property\*N} は異なる場合があります (したがって、{property\*C}=1)。

## ラベル

トリガ・コードには、[行ラベル](#) (タグ) が含まれる場合があります。トリガ・コードでラベルを指定するには、ラベル行の先頭にコロンを付けて、その行が最初の列で始まることを示します。InterSystems IRIS はコロンを削除して、残りの行をラベルとして処理します。ただし、トリガ・コードはプロシージャ・ブロックの範囲外で生成されるため、すべてのラベルはクラス定義に対して一意である必要があります。クラスのルーチンにコンパイルされるその他すべてのコードには、同じラベルを定義することはできません。これには、その他のトリガ、プロシージャ・ブロックを使用しないメソッド、[SqlComputeCode](#) などが含まれます。

**注釈** ラベル用のコロン接頭語の使用は、[ホスト変数参照](#)用のコロン接頭語の使用よりも優先されます。この競合を回避するために、埋め込み SQL トリガ・コード行の先頭にホスト変数参照を記述しないことをお勧めします。トリガ・コード行の先頭にホスト変数参照を記述する必要がある場合は、二重のコロン接頭語を使用することで、それがホスト変数でありラベルでないことを明示してください。

## メソッドの呼び出し

クラス・メソッドは開いているオブジェクトの有無に依存しないため、クラス・メソッドをトリガ・コード内から使用できます。クラス・メソッドを呼び出すには、`##class(classname).Method()` 構文を使用する必要があります。`..Method()` 構文では、現在開いているオブジェクトが必要なので、この構文は使用できません。

クラス・メソッドの引数として現在の行のフィールドの値を渡すことができますが、クラス・メソッド自体はフィールド構文を使用できません。

## 既存のトリガのリスト

**INFORMATION.SCHEMA.TRIGGERS** クラスを使用して、現在定義されているトリガをリストできます。このクラスは各トリガについて、トリガ名、関連するスキーマとテーブル名、およびトリガ作成タイムスタンプをリストします。各トリガについて、EVENT\_MANIPULATION プロパティ (INSERT、UPDATE、DELETE、INSERT/UPDATE、INSERT/UPDATE/DELETE) と ACTION\_TIMING プロパティ (BEFORE、AFTER) をリストします。また、生成された SQL トリガ・コードである ACTION\_STATEMENT もリストします。

## トリガの実行時エラー

トリガおよびそのトリガが起動するイベントは、単一行ベースでアトミック処理として実行されます。つまり、以下のように処理されます。

- 失敗した BEFORE トリガはロールバックされ、関連する INSERT、UPDATE、または DELETE 操作は実行されません。さらに行のすべてのロックは解放されます。

- ・ 失敗した AFTER トリガはロールバックされ、関連する INSERT、UPDATE、または DELETE 操作がロールバックされます。さらに行のすべてのロックは解放されます。
- ・ 失敗した INSERT、UPDATE、または DELETE 操作はロールバックされ、関連する BEFORE トリガがロールバックされます。さらに行のすべてのロックは解放されます。
- ・ 失敗した INSERT、UPDATE、または DELETE 操作はロールバックされ、関連する AFTER トリガは実行されません。さらに行のすべてのロックは解放されます。

整合性が維持されるのは、現在の行の操作のみであることに注意してください。アプリケーション・プログラムは、トランザクション処理文を使用して、複数行での操作も含めてデータの整合性の問題に対処する必要があります。

トリガはアトミック処理であるため、コミットやロールバックのようなトランザクション文は、トリガ・コード内ではコーディングできません。

INSERT、UPDATE、または DELETE 操作が複数のトリガを実行する場合、1 つのトリガが失敗すると残りすべてのトリガは実行されないままになります。

- ・ SQLCODE -415 : トリガ・コードにエラーがある場合 (例えば、存在しないテーブルや未定義の変数への参照)、トリガ・コードの実行は実行時に失敗し、InterSystems IRIS は SQLCODE -415 エラー “SQL ファイラ内で致命的なエラーが発生しました” を発行します。
- ・ SQLCODE -130 ~ -135 : トリガ処理が失敗すると、InterSystems IRIS は失敗したトリガのタイプにより SQLCODE エラー・コードの -130 から -135 のうちの 1 つを実行時に発行します。トリガ・コード内で %ok 変数を 0 に設定することで、トリガを失敗させることができます。これにより、適切な SQLCODE エラー (例えば、SQLCODE -131 “After Insert トリガが失敗しました”) が発行され、トリガ・コード・エラーの原因を説明する文字列として %msg 変数のユーザ指定値が返されます。

## 例

以下の例では、ObjectScript DELETE トリガとの CREATE TRIGGER の使用を実際に示します。ここでは、レコードが含まれるデータ・テーブル (TestDummy) が存在することを想定しています。これは、ログ・テーブル (TestDummyLog) と DELETE トリガを作成します。DELETE トリガは、データ・テーブルで削除が実行されたときにログ・テーブルへの書き込みを行います。このトリガは、データ・テーブルの名前、削除された行の RowId、現在の日付、および実行された操作のタイプ (%oper 特殊変数) (この場合は “DELETE”) を挿入します。

### SQL

```
CREATE TABLE TestDummyLog
  (TableName VARCHAR(40),
   IDVal INTEGER,
   LogDate DATE,
   Operation VARCHAR(40))
```

### ObjectScript

```
&sql(CREATE TRIGGER TrigTestDummy AFTER DELETE ON TestDummy
  LANGUAGE OBJECTSCRIPT {
    NEW id
    SET id = {ID}
    &sql(INSERT INTO TestDummyLog (TableName, IDVal, LogDate, Operation)
      VALUES ('TestDummy', :id, $HOROLOG, :%oper))
  }
)
WRITE !, "SQL trigger code is: ", SQLCODE
```

以下の例では、SQL INSERT トリガとの CREATE TRIGGER の使用を実際に示します。1 つ目のプログラムでは、テーブル、そのテーブルの INSERT トリガ、およびそのトリガの使用を記録するログ・テーブルを作成します。2 つ目のプログラムでは、テーブルに対して INSERT を発行し、トリガを呼び出してログ・テーブルにエントリを記録します。ログ・エントリの表示後、プログラムは両方のテーブルを削除し、このプログラムを繰り返し実行できるようにします。

## SQL

```
CREATE TABLE TestDummy (  
    testnum      INT NOT NULL,  
    firstword    CHAR (30) NOT NULL,  
    lastword     CHAR (30) NOT NULL,  
    CONSTRAINT TestDummyPK PRIMARY KEY (testnum))  
CREATE TABLE TestDummyLog (  
    entry CHAR (60) NOT NULL)  
)  
CREATE TRIGGER TrigTestDummy AFTER INSERT ON TestDummy  
BEGIN  
    INSERT INTO TestDummyLog (entry) VALUES  
    (CURRENT_TIMESTAMP||' INSERT to TestDummy');  
END
```

## SQL

```
INSERT INTO TestDummy (testnum,firstword,lastword) VALUES  
(46639,'hello','goodbye')  
SELECT entry FROM TestDummyLog  
DROP TABLE TestDummy  
DROP TABLE TestDummyLog
```

以下に、括弧内の述語条件が満たされた場合にのみ action が実行されるように指定する WHEN 節の例を示します。

## SQL

```
CREATE TRIGGER Trigger_2 AFTER INSERT ON Table_1  
WHEN (f1 %STARTSWITH 'A')  
BEGIN  
    INSERT INTO Log_Table VALUES (new_row.Category);  
END
```

以下の例では、Sample.Employee での行の挿入、更新、または削除後に Name フィールドの古い値と新しい値を返すトリガを定義します (SQL シェルからこのトリガ・イベント操作を実行して、この結果を表示できます)。

```
CREATE TRIGGER EmployNameTrig AFTER INSERT,UPDATE,DELETE ON Sample.Employee  
LANGUAGE OBJECTSCRIPT  
{WRITE "Employee old name:",{Name*O}," new name:",{Name*N}," ",{%%OPERATION}," on ",{%%TABLENAME},!}
```

## 関連項目

- ・ [DROP TRIGGER](#)
- ・ [GRANT](#)
- ・ [トリガの使用法](#)
- ・ [SQLCODE エラー・メッセージ](#)

# CREATE USER (SQL)

ユーザ・アカウントを作成します。

## 構文

```
CREATE USER user-name IDENTIFY BY password
CREATE USER user-name IDENTIFIED BY password
CREATE USER user-name [ WITH ] PASSWORD password
```

## 説明

CREATE USER コマンドは、指定したパスワードを持つユーザ・アカウントを作成します。

user-name は 160 文字までの有効な識別子です。user-name は、[識別子](#)の名前付け規約に従っている必要があります。user-name には Unicode 文字を使用することができます。ユーザ名は、大文字と小文字が区別されません。

[区切り識別子](#)として指定された user-name には、SQL 予約語を使用することも、コンマ (,)、ピリオド (.), キャレット (^)、および 2 文字の矢印シーケンス (->) を含めることもできます。この名前の先頭の文字には、アスタリスク (\*) を除く任意の有効な文字を使用できます。

IDENTIFY BY、IDENTIFIED By、および WITH PASSWORD キーワードは同義語です。

password には数値リテラル、識別子、または引用符付きの文字列を指定できます。数値リテラルまたは識別子は引用符で囲む必要はありません。引用符付きの文字列は通常、パスワードに空白を含めるために使用します。引用符付きのパスワードには、文字の任意の組み合わせを使用できますが、引用符自体は使用できません。数値リテラルは 0 ～ 9 の文字のみで構成する必要があります。[識別子](#)は、文字 (大文字/小文字) あるいは % (パーセント記号) で始める必要があります。その後ろに、任意の文字、任意の数字、\_ (アンダースコア)、& (アンパサンド)、\$ (ドル記号)、@ (アットマーク) を自由に組み合わせて使用できます。

パスワードは大文字と小文字が区別されます。パスワードは 3 文字以上 33 文字未満にする必要があります。指定したパスワードが長すぎるか、または短かすぎると、SQLCODE -400 エラーが生成されます。生成される %msg の値は “エラー #845 : パスワードが長さまたはパターンの要件と一致しません” になります。

ホスト変数を使用して user-name または password の値を指定することはできません。

ユーザを作成しても、ロールの作成やユーザへのロールの付与は行われません。ただし、ユーザはデータベースにログインする許可が与えられ、ネームスペースでの SQL 特権を 1 つでも持っていれば、%SQL/Service サービスの USE 許可が与えられます。ユーザに特権やロールを割り当てるには、GRANT コマンドを使用します。ロールの作成には、CREATE ROLE コマンドを使用します。

CREATE USER を実行して既に存在するユーザを作成しようとすると、SQL は SQLCODE -118 エラーを返します。生成される %msg の値は “ユーザが付けた 'name' は既に存在しています” になります。\$SYSTEM.SQL.Security.UserExists() メソッドを呼び出すことによって、ユーザが既に存在するかどうかを判別できます。

## ObjectScript

```
WRITE $SYSTEM.SQL.Security.UserExists("Admin"),!
WRITE $SYSTEM.SQL.Security.UserExists("BertieWooster")
```

このメソッドは、指定したユーザが存在する場合 1 を返し、存在しない場合は 0 を返します。ユーザ名は、大文字と小文字が区別されません。

## 特権

CREATE USER コマンドの実行には特権が必要です。埋め込み SQL で CREATE USER を使用する前に、以下のいずれかを持つユーザとしてログインする必要があります。

- ・ USE 権限がある [%Admin\\_Secure](#) 管理リソース

- ・ USE 権限がある [%Admin\\_UserEdit](#) 管理リソース
- ・ システムに対する全面的なセキュリティ特権

これらの特権がない場合に CREATE USER コマンドを実行すると、SQLCODE -99 エラー（特権違反）が返されます。  
\$SYSTEM.Security.Login() メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( /* SQL code here */ )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“インターシステムズ・クラス・リファレンス”の“%SYSTEM.Security”を参照してください。

## 引数

### user-name

作成するユーザ名。名前は、最大 128 文字の識別子です。Unicode 文字を使用できます。user-name では大文字と小文字が区別されません。

### password

このユーザのパスワード。password は少なくとも 3 文字必要で、32 文字を超えることはできません。パスワードは大文字と小文字が区別されます。パスワードには Unicode 文字を使用することができます。

## 例

以下の埋め込み SQL の例は、“Carl4SHK”というパスワードを持つ“BillTest”という新しいユーザを作成します（\$RANDOM トグルが記述されているので、このプログラム例を繰り返し実行できます）。

### ObjectScript

```
Main
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
SET x=$SYSTEM.SQL.Security.UserExists("BillTest")
IF x=0 {&sql(CREATE USER BillTest IDENTIFY BY Carl4SHK)
      IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                      QUIT}
      }
WRITE "User BillTest exists",!
Cleanup
SET toggle=$RANDOM(2)
IF toggle=0 {
  &sql(DROP USER BillTest)
  IF SQLCODE '= 0 {WRITE "DROP USER error: ",SQLCODE,!}
}
ELSE {WRITE !,"No drop this time",!}
WRITE "User BillTest exists? ",$SYSTEM.SQL.Security.UserExists("BillTest"),!
QUIT
```

## 関連項目

- ・ SQL 文：[ALTER USER](#)、[DROP USER](#)、[GRANT](#)、[REVOKE](#)、[CREATE ROLE](#)
- ・ [SQL のユーザ、ロール、および特権](#)
- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript：[\\$ROLES](#) および [\\$USERNAME](#) 特殊変数



# CREATE VIEW (SQL)

ビューを作成します。

## 構文

```
CREATE [OR REPLACE] VIEW view-name [(column-commalist)]
AS select-statement
[ WITH READ ONLY | WITH [level] CHECK OPTION ]
```

## 概要

CREATE VIEW コマンドは、[ビュー](#)の内容を定義します。ビューを定義する SELECT 文は、複数のテーブルや他のビューを参照できます。

## 特権

CREATE VIEW コマンドの実行には特権が必要です。CREATE VIEW を実行するには、ユーザは `%CREATE_VIEW` [管理特権](#)を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %CREATE\_VIEW " に設定されます。適切な付与特権を持っている場合は、[GRANT](#) コマンドを使用して `%CREATE_VIEW` 特権を割り当てることができます。

作成するビューの SELECT 節内から参照するオブジェクトを選択するには、適切な特権が必要です。

- ・ ダイナミック SQL またはデータベース・ドライバを使用してビューを作成する場合、ビューが参照する基のテーブル (またはビュー) から選択するすべての列に対する SELECT 特権が必要です。指定したテーブル (またはビュー) に対する SELECT 特権を持っていない場合は、CREATE VIEW コマンドを実行できません。

ただし、定義したビューを投影するクラスをコンパイルするときには、ビューが参照する基のテーブル (またはビュー) から選択する列について、これらの SELECT 特権は適用されません。例えば、(これらの SELECT 特権を持つ) 特権ルーチンを使用してビューを作成すると、後でビュー・クラスをコンパイルできます。これは、ビューが参照するテーブルに対する SELECT 特権を持っているかどうかに関係なく、そのビューの所有者であるためです。

- ・ ビューに対する SELECT 特権の WITH GRANT OPTION を取得するには、ビューが参照するすべてのテーブル (またはビュー) について WITH GRANT OPTION を持っていなければなりません。
- ・ ビューについて INSERT、UPDATE、DELETE、または REFERENCES 特権を取得するには、ビューが参照するすべてのテーブル (またはビュー) に対して同じ特権を持っている必要があります。これらの特権の WITH GRANT OPTION を取得するには、基になるテーブルに対して WITH GRANT OPTION 特権を持っている必要があります。
- ・ ビューが WITH READ ONLY に指定されていると、基本テーブルに対する特権を持っていてもそのビューに対しての INSERT、UPDATE、または DELETE 特権は認められません。後でビューが読み取り/書き込み可能に再定義されると、ビューに投影されているクラスがリコンパイルされるときにこれらの特権が追加されます。

[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザがこれらのテーブルレベルの特権を持っているかどうかを確認できます。`$SYSTEM.Security.CheckPrivilege()` メソッドを呼び出すことにより、指定のユーザがこれらのテーブルレベルの特権を持っているかどうかを確認できます。特権の割り当てについては、["GRANT"](#) コマンドを参照してください。

ビューがコンパイルされると、ビューの作成者 (所有者) に `%ALTER` 特権の WITH GRANT OPTION が与えられます。

埋め込み SQL では、以下のように `$SYSTEM.Security.Login()` メソッドを使用して適切な特権を持ったユーザとしてログインできます。

## ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```



\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“%SYSTEM.Security”を参照してください。

%CREATE\_VIEW 特権は **GRANT** コマンドで割り当てます。このときユーザまたはロールにこの特権を割り当てる必要があります。既定では、CREATE VIEW のセキュリティ特権が適用されます。この特権要件は、\$SYSTEM.SQL.Util.SetOption() メソッドの SET status=\$SYSTEM.SQL.Util.SetOption("SQLSecurity",0,.oldval) を使用してシステム全体で構成できます。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() メソッドを呼び出します。これにより、[SQL ] 設定が表示されます。

既定値は 1 (有効) です。SQL セキュリティが有効な場合、ユーザは特権が付与されているテーブルやビューのみでアクションを実行できます。この設定を推奨します。

このメソッドが 0 に設定された場合、この設定の変更後に開始された新しいプロセスすべてで SQL セキュリティは無効になります。つまり、特権ベースのテーブルやビューのセキュリティは抑制されていることを意味します。ユーザを指定しなくてもテーブルの作成が可能になります。この場合、ダイナミック SQL はユーザとして “\_SYSTEM” を、埋め込み SQL はユーザとして “” (空文字列) を割り当てます。ユーザは特権がなくてもテーブルやビューに対してアクションを実行することができます。

## ビューの名前付け規約

ビュー名の名前付け規約はテーブル名と同じで、同じ名前セットを共有します。そのため、同じスキーマ内でテーブルとビューに同じ名前を使用することはできません。これを実行しようとすると、SQLCODE -201 エラーが返されます。テーブルが現在のネームスペースに既に存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.TableExists("schema.tname") メソッドを使用します。同じ名前を持つテーブル定義およびビュー定義を投影するクラスでも、SQLCODE -201 エラーが生成されます。

ビュー名は、**識別子**の規則に従い、以下のような制約を受けます。既定のビュー名は、簡単な識別子です。ビュー名は 128 文字を超えることはできません。ビュー名は、大文字と小文字が区別されません。

InterSystems IRIS はビュー名を使用して、対応するクラス名を生成します。クラス名には英数字 (文字および数字) のみを使用し、最初の 96 文字は一意である必要があります。このクラス名を生成するために、InterSystems IRIS は最初にビュー名から句読点を削除し、次に最初の 96 文字が一意である識別子を生成します。その際、クラス名の一意性を維持するために、必要に応じて最後の文字を (0 で始まる) 整数に置き換えます。InterSystems IRIS は有効なビュー名から一意のクラス名を生成しますが、ビューの名前を付ける際には、この名前の生成に伴う以下の制約について考慮する必要があります。

- ・ ビュー名には、最低でも 1 文字を含める必要があります。ビュー名の先頭の文字または最初の句読点に続く文字は、数字以外の文字にする必要があります。
- ・ InterSystems IRIS は 16 ビット (ワイド) 文字のビュー名をサポートします。**\$ZNAME** テストに合格した文字は、有効な文字です。
- ・ ビュー名の最初の文字が句読点文字の場合、2 番目の文字に数字を指定することはできません。これにより、SQLCODE -400 エラーが発生し、生成される %msg の値は “エラー #5053 : クラス名 'schema.name' が無効です” になります (句読点文字なし)。例えば、指定したビュー名が %7A の場合、生成される %msg は “エラー #5053 : クラス名 'User.7A' が無効です” になります。
- ・ 生成されたクラス名には句読点が含まれないため、句読点のみ既存のビュー名またはテーブル名と異なるビュー名を作成することは可能ですが、お勧めできません。この場合、InterSystems IRIS は、一意のクラス名を作成するために名前の最後の文字を (0 で始まる) 整数に置き換えます。
- ・ ビュー名は 96 文字よりも大幅に長くすることができますが、最初の 96 の英数字が異なるようにビュー名を作成すると処理はるかに容易になります。

ビュー名は修飾、未修飾のどちらでもかまいません。

ビュー名を修飾する場合は (schema.viewname)、既存のスキーマを指定することも、新規スキーマを指定することもできます。新規スキーマを指定する場合、システムがそのスキーマを作成します。

ビュー名が未修飾の場合は (viewname)、既定のスキーマ名が使用されます。

## 既存のビュー

指定のビューが現在のネームスペースに既に存在するかどうかを確認するには、`$SYSTEM.SQL.Schema.ViewExists("schema.vname")` メソッドを使用します。

既存のビューと同じ名前のビューを作成しようとしたときの反応は、オプションの `OR REPLACE` キーワードおよび構成設定により異なります。

### OR REPLACE を指定する場合

`CREATE OR REPLACE VIEW` を指定すると、`SELECT` 節で指定されたビュー定義、および指定された任意の `WITH READ ONLY` または `WITH CHECK OPTION` によって既存のビューが置き換えられます。これは該当する `ALTER VIEW` 文を実行することと同じです。元のビューに付与されていた特権はすべて残ります。`CREATE OR REPLACE VIEW` 文を実行するユーザにビューの所有権が移転されます。

このキーワード句の機能はすべて `ALTER VIEW` で使用できます。このキーワード句は Oracle SQL コードとの互換性のために用意されたものです。

### OR REPLACE を指定しない場合

既定では、`CREATE VIEW` を指定すると、既存のビューと同じ名前のビューを作成しようとしても拒否され、`SQLCODE -201` エラーが発行されます。現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。これにより、`[ DDL CREATE TABLE CREATE VIEW ]` 設定が表示されます。既定値は 0 (いいえ) で、これが推奨設定です。このオプションを 1 (はい) に設定すると、InterSystems IRIS はこのビューに対応するクラス定義を削除し、クラスを再作成します。これは、`DROP VIEW` を実行してから `CREATE VIEW` を実行するのと同じです。この設定を変更すると、`CREATE VIEW` と `CREATE TABLE` の両方に影響することに注意してください。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション（および他の同様の作成、変更、および削除のオプション）をシステム全体で設定できます。

## 列名

ビューには括弧で囲まれた列名のリスト `column-commalist` をオプションで含めることができます。このような列名を指定すると、そのビューを使用する際には、この名前を使用して列のデータへのアクセスや列のデータの表示が行われます。

`column-commalist` を省略する場合は、以下の条件が適用されます。

- このビューを使用するときには、`SELECT` のソース・テーブルの列名を使用してデータにアクセスして表示します。
- `SELECT` のソース・テーブルのいずれかの列名に列エイリアスがある場合、その列エイリアスが、このビューを使用するときにデータにアクセスして表示するために使用される名前になります。
- `SELECT` のソース・テーブルの列名にテーブル・エイリアスがある場合、そのテーブル・エイリアスは、このビューを使用するときにデータにアクセスして表示するために使用される名前では使われません。

列名のリストを省略する場合は、括弧も省略する必要があります。

`column-commalist` を指定する場合は、以下の条件が適用されます。

- 単一のフィールドを指定する場合でも、列名のリストでは括弧を指定する必要があります。複数の列名はコンマで区切る必要があります。`column-commalist` 内では空白およびコメントを使用できます。
- 列名の数は `SELECT` 文で指定した列の数と一致する必要があります。ビューとクエリとで列数が一致していない場合は、コンパイル時に `SQLCODE -142` エラーが発生します。
- 列名は有効な識別子である必要があります。列名には `SELECT` の列名と異なる名前、同じ名前、あるいはこの両方を組み合わせて指定できます。指定するビューの列名の順序と `SELECT` の列名の順序は一致させます。ビューの列に対して無関係の `SELECT` 列の名前を割り当てることのできるため、ビューの列名を割り当てる際には注意が必要です。

- ・ 列名は一意にしてください。重複する列名を指定すると、SQLCODE -97 エラーが発生します。列名は、句読点を削除することで、[対応するクラス・プロパティ名に変換](#)されます。句読点のみが異なる列名は許可されますが、お勧めしません。

以下の例は、ビューとクエリで列が一致するときの CREATE VIEW を示しています。

## SQL

```
CREATE VIEW MyView (ViewCol1, ViewCol2, ViewCol3) AS
  SELECT TableCol1, TableCol2, TableCol3
  FROM MyTable
```

また、クエリで AS キーワードを使ってビュー列をクエリ列 (またはビュー列のペア) として関連付けることもできます。以下はその例です。

## SQL

```
CREATE VIEW MyView AS
  SELECT TableCol1 AS ViewCol1,
         TableCol2 AS ViewCol2,
         TableCol3 AS ViewCol3
  FROM MyTable
```

## SELECT 列とビューの列

- ・ 複数の SELECT 列からのデータは、1 つのビューの列に連結できます。以下に例を示します。

## SQL

```
CREATE VIEW MyView (fullname) AS SELECT firstname||' '||lastname FROM MyTable
```

- ・ 複数のビューの列が同じ SELECT 列を参照できます。以下に例を示します。

## SQL

```
CREATE VIEW MyView (lname,surname) AS SELECT lastname,lastname FROM MyTable
```

## SELECT 節の考慮事項

ビューは、特定のテーブルの行や列の単純なサブセットである必要はありません。ビューは、任意の複雑度を持つ SELECT 節を使用し、テーブルやビューを任意に組み合わせて指定することで作成できます。ただし、ビュー定義の SELECT 節には、いくつかの制約があります。

- ・ [ORDER BY](#) 節は [TOP](#) 節と組み合わせた場合にのみ含めることができます。ビュー内のすべての行を含める場合は、TOP ALL 節を使用します。TOP 節は ORDER BY 節を指定しなくても含めることができます。ただし、TOP 節を使用せずに ORDER BY 節を含めると、SQLCODE -143 エラーが生成されます。ORDER BY 節を使用したクエリを持つビュー・クラスの SQL ビューを投影すると、ビュー・プロジェクションではその ORDER BY 節が無視されます。
- ・ [ホスト変数](#)を含むことはできません。SELECT 節でホスト変数を参照しようすると、システムにより、SQLCODE -148 エラーが生成されます。
- ・ [INTO](#) キーワードを含むことはできません。INTO 節と共に SELECT を指定するビューを作成することはできますが、このビューを実行すると、SQLCODE -25 エラーで失敗します。

CREATE VIEW には、2 つのテーブルの UNION 結合から列を選択する [UNION](#) 文を含めることができます。以下の例のように、UNION を指定することができます。

## SQL

```
CREATE VIEW MyView (vname,vstate) AS
  SELECT t1.name,t1.home_state
    FROM Sample.Person AS t1
  UNION
  SELECT t2.name,t2.office_state
    FROM Sample.Employee AS t2
```

上記の例のように、ビュー名が未修飾の場合は、ビューによって参照されるテーブルが Sample スキーマに属していたとしても、[既定のスキーマ名](#) (初期のスキーマ既定値 SQLUser.MyView など) が既定で使用されることに注意してください。ビューに関連するテーブルと共に確実に格納するために、ビュー名は常に修飾付きにすることをお勧めします。

### View ID : %vid

ビューを使用してデータにアクセスする際に、InterSystems IRIS では、そのビューによって返される行のそれぞれに、連続した整数のビュー ID (%vid) が割り当てられます。テーブル行の ID 番号と同様、これらのビューの行 ID 番号は、システムによって割り当てられる一意の非ゼロ、非 NULL 値で、変更することはできません。この %vid は、通常は非表示です。テーブル行 ID とは異なり、アスタリスク構文の使用時には表示されません。表示されるのは、SELECT で明示的に指定された場合のみです。%vid を使用すると、ビューにアクセスする SELECT で返される行の数をさらに制限することができます。%vid の使用の詳細は、["ビューの定義と使用"](#) を参照してください。

## 引数

### view-name

作成する[ビューの名前](#)。[テーブル名](#)と同様の追加の名前付け制約に従う、有効な[識別子](#)です。ビュー名は修飾 (schema.viewname)、未修飾 (viewname) のどちらでもかまいません。ビュー名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。同じスキーマ内では、テーブルとビューに同じ名前を使用することはできません。

### column-commalist

オプションの引数です。ビューを構成する列名。1 つ以上の有効な[識別子](#)です。指定する場合、このリストは括弧で囲み、リスト内の項目はコンマで区切ります。

### AS select-statement

ビューを定義する [SELECT](#) 文。

### WITH READ ONLY

このビューの基になっているテーブルに対して、このビューからは挿入、更新、削除の各操作を実行できないように指定する引数 (オプション)。既定では、後述の制限を条件として、ビュー経由でこれらの操作が許可されます。

### WITH level CHECK OPTION

このビューの基になっているテーブルに対して、挿入、更新、削除の各操作をこのビューからどのように実行するかを指定する引数 (オプション)。level には、キーワード LOCAL または CASCADED を指定できます。level を指定しない場合、WITH CHECK OPTION は既定で CASCADED になります。

## ビュー経由の更新

ビューを使用して、ビューの基であるテーブルの更新が可能です。ビュー経由で、新しい行の [INSERT](#) による挿入、行データの [UPDATE](#) による更新、および、行の [DELETE](#) による削除が可能です。CREATE VIEW 文にこの機能を指定していれば、INSERT 文、UPDATE 文、DELETE 文をビューに発行できます。ビューを使用した更新を許可するには、ビューの定義時に WITH CHECK OPTION (既定) を指定します。

注釈 ビューがシャード・テーブルに基づく場合、WITH CHECK OPTION でビューを介して INSERT、UPDATE、または DELETE を実行することはできません。これを試みると、SQLCODE -35 が生じ、%msg が “  
 (sample.myview) with check option INSERT/UPDATE/DELETE  
 ” に設定されます。

ビュー経由の更新を禁止するには、WITH READ ONLY を指定します。WITH READ ONLY で作成されるビューで INSERT、UPDATE、または DELETE を実行すると、SQLCODE = -35 エラーが生成されます。

ビュー経由の更新を実行するには、GRANT コマンドによって、テーブルやビューの更新に対応する特権を指定する必要があります。

ビュー経由で更新するには、以下の制限に従います。

- ・ ビューは、ビューとして投影されたクラス・クエリではありません。
- ・ ビューのクラスはクラス・パラメータ READONLY=1 を含むことはできません。
- ・ ビューの SELECT 文は DISTINCT、TOP、GROUP BY、または HAVING 節を含むことはできず、UNION の一部にはできません。
- ・ ビューの SELECT 文はサブクエリを含むことはできません。
- ・ ビューの SELECT 文は、列参照である値式だけをリストします。
- ・ ビューの SELECT 文には 1 つのテーブル参照のみを含めることができます。select-list や WHERE 節に FROM 節、JOIN 構文、または 矢印構文を含めることはできません。テーブル参照は、更新可能なテーブルまたは更新可能なビューのいずれかを指定する必要があります。

WITH CHECK OPTION 節を指定すると、挿入または更新操作の際に、その結果行がビュー定義の WHERE 節に対して検証されます。これで、挿入または変更した行が、確実に導出されたビュー・テーブルの一部になります。利用可能なチェック・オプションは以下に示す 2 つです。

- ・ WITH LOCAL CHECK OPTION では、INSERT または UPDATE 文で指定されたビューの WHERE 節のみがチェックされます。
- ・ WITH CASCADED CHECK OPTION では、INSERT または UPDATE 文で指定されたビューおよびすべての基本ビューで指定された WHERE 節がチェックされます。これは、この基本ビューのあらゆる WITH LOCAL CHECK OPTION 節よりも優先されます。更新可能なすべてのビューについて WITH CASCADED CHECK OPTION を使用することをお勧めします。

WITH CHECK OPTION を指定すると、チェック・オプションは既定で CASCADED に設定されます。キーワード CASCADE は CASCADED の同義語です。

INSERT オペレーションで WITH CHECK OPTION 検証 (上記の説明にある) に失敗すると、InterSystems IRIS は SQLCODE -136 エラーを発行します。

UPDATE オペレーションで WITH CHECK OPTION 検証 (上記の説明にある) に失敗すると、InterSystems IRIS は SQLCODE -137 エラーを発行します。

## 例

以下の例は、PhoneBook テーブルから “CityPhoneBook” という名前のビューを作成します。

### SQL

```
CREATE VIEW CityPhoneBook AS
  SELECT Name FROM PhoneBook WHERE City='Boston'
```

以下の例は、Guides テーブルから “GuideHistory” という名前のビューを作成します。すべてのタイトル (Title 列から) と、その人が退職したか否かをリストにします。



## SQL

```
CREATE VIEW GuideHistory AS
  SELECT Guides, Title, Retired, Date_Retired
  FROM Guides
```

以下の例は、テーブル MyTest を作成し、次に MyTest から 1 つのフィールドを選択するビュー MyTestView を作成します。

## SQL

```
CREATE TABLE Sample.MyTest (
  TestNum      INT NOT NULL,
  FirstWord     CHAR (30) NOT NULL,
  LastWord      CHAR (30) NOT NULL,
  CONSTRAINT MyTestPK PRIMARY KEY (TestNum))

CREATE VIEW Sample.MyTestView AS
  SELECT FirstWord FROM Sample.MyTest
  WITH CASCADED CHECK OPTION
```

以下の例は、MyTest から 2 つのフィールドを選択するビュー MyTestView を作成します。このビューの SELECT クエリには TOP 節と ORDER BY 節が含まれます。

## SQL

```
CREATE TABLE Sample.MyTest (
  TestNum      INT NOT NULL,
  FirstWord     CHAR (30) NOT NULL,
  LastWord      CHAR (30) NOT NULL,
  CONSTRAINT MyTestPK PRIMARY KEY (TestNum))

CREATE VIEW Sample.MyTestView AS
  SELECT TOP ALL FirstWord,LastWord FROM Sample.MyTest
  ORDER BY LastWord)
```

以下の例は、3 つのテーブル (Proj、Staff、Works) から “StaffWorksDesign” という名前のビューを作成します。Name 列、Cost 列、Project 列はデータを提供します。

## SQL

```
CREATE VIEW StaffWorksDesign (Name,Cost,Project)
  AS SELECT EmpName,Hours*2*Grade,PName
  FROM Proj,Staff,Works
  WHERE Staff.EmpNum=Works.EmpNum
  AND Works.PNum=Proj.PNum AND PType='Design'
```

以下の例は、UNION を使用して b.table2 と a.table1 から選択することにより “v\_3” という名前のビューを作成します。

## SQL

```
CREATE VIEW v_3(fvarchar)
  AS SELECT DISTINCT *
  FROM
    (SELECT fVARCHAR2 FROM b.table2
    UNION ALL
    SELECT fVARCHAR1 FROM a.table1)
```

## 関連項目

- ・ [ALTER VIEW](#)
- ・ [DROP VIEW](#)
- ・ [CREATE TABLE](#)
- ・ [GRANT](#)
- ・ [SELECT](#)



- ・ [ビューの定義と使用](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

# DECLARE (SQL)

カーソルを宣言します。

## 構文

```
DECLARE cursor-name CURSOR FOR query
```

## 概要

DECLARE 文は、[カーソル・ベース埋め込み SQL](#) で使用するカーソルを宣言します。カーソルを宣言した後、[OPEN](#) 文を発行してカーソルを開いてから、一連の [FETCH](#) 文を発行して個別のレコードを取得します。このカーソルは、これらの FETCH 文によって取得するレコードの選択に使用する SELECT クエリを定義します。[CLOSE](#) 文を発行してカーソルを閉じます (ただし削除はしません)。

SQL 文として、DECLARE は埋め込み SQL からのみサポートされます。ダイナミック SQL の場合は、代わりに単純な SELECT 文 (INTO 節のないもの) を使用するか、ダイナミック SQL と埋め込み SQL の組み合わせを使用します。同様の操作は、ODBC でも ODBC API を使用してサポートされます。

DECLARE は前方向のみ (スクロールできない) カーソルを宣言します。フェッチ操作は、クエリの結果セットの最初のレコードから始まり、結果セットのレコードを順次処理します。FETCH はレコードを 1 回しかフェッチできません。次の FETCH では、結果セットの次のレコードが順次フェッチされます。

DECLARE は宣言であり実行文ではないため、SQLCODE 変数を設定したり削除したりすることはできません。

## カーソル名

カーソル名は、大文字と小文字を区別します。

カーソル名は、ルーチンおよびその対応するクラス内で一意である必要があります。カーソル名の長さに制限はなく、最初の 29 文字は一意である必要があります。カーソル名は、大文字と小文字を区別します。指定したカーソルが既に宣言されている場合、コンパイル・エラーは発行されません。SQL の実行時には、そのカーソルのうち最後に宣言されたインスタンスが使用されます。

カーソル名はネームスペース固有ではありません。あるネームスペースでカーソルを DECLARE でき、別のネームスペースからこのカーソルを OPEN、FETCH、または CLOSE できます。埋め込み SQL のコンパイルは OPEN コマンドの実行時に行われます。SQL テーブルとローカル変数はネームスペース固有であるため、OPEN 操作は、クエリで指定されたテーブルが配置されているネームスペースと同じネームスペースで呼び出す必要があります (またはそのネームスペース内のテーブルにアクセスできる必要があります)。

カーソル名の最初には文字を指定する必要があります。カーソル名の 2 文字目以降には、文字または数字のいずれかを指定する必要があります。SQL [識別子](#) とは異なり、カーソル名に句読点文字を使用することはできません。

カーソル名に SQL 予約語を指定するには、区切り文字 (二重引用符) を使用できます。区切られたカーソル名は、SQL 区切り識別子ではありません。区切られたカーソル名でも大文字と小文字が区別され、句読点文字を含むことはできません。多くの場合、SQL 予約語はカーソル名として使用するべきではありません。

## カーソルを使用した更新

[WHERE CURRENT OF](#) 節を指定して UPDATE 文または DELETE 文を使用することによって、宣言したカーソルを使用してレコードの更新と削除を実行できます。InterSystems SQL では、対象のテーブルおよび列に対する適切な特権がある場合には、UPDATE 操作または DELETE 操作に常にカーソルを使用できます。オブジェクト特権の割り当ての詳細は、[GRANT](#) 文を参照してください。

DECLARE 文では、クエリの次に FOR UPDATE キーワード節または FOR READ ONLY キーワード節を指定できます。これらの節はオプションで、操作は何も実行されません。これらは、クエリを発行するプロセスが必要な更新および削除のオブジェクト特権を持っているまたは持っていないことをコードに記述する方法として提供されています。

## 引数

### cursor-name

カーソル名。文字で開始し、文字および数字のみを使用する必要があります (カーソル名は SQL 識別子の規約に従いません)。カーソル名は、大文字と小文字を区別します。以下のように追加の名前付け制約に従います。

### query

カーソルの結果セットを定義する標準 **SELECT** 文。この SELECT には %NOFPLAN キーワードを含めることで、InterSystems IRIS がこのクエリの凍結プラン (ある場合) を無視するように指定できます。この SELECT で指定する ORDER BY 節には、TOP 節を記述しても記述しなくてもかまいません。この SELECT は、FROM 節にテーブル値関数を指定できます。

## 例

以下の埋め込み SQL の例では、DECLARE を使用して、2 つの出力ホスト変数を指定するクエリのカーソルを定義しています。その後、カーソルはオープンされて、繰り返しフェッチされ、クローズされます。

### ObjectScript

```
SET name="John Doe",state="###"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'A'
    FOR READ ONLY)
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg QUIT}
WRITE !,"AFTER: Name=",name," State=",state
```

以下の埋め込み SQL の例では、DECLARE を使用して、INTO 節の出力ホスト変数と WHERE 節の入力ホスト変数の両方を指定するクエリのカーソルを定義しています。その後、カーソルはオープンされて、繰り返しフェッチされ、クローズされます。

### ObjectScript

```
NEW SQLCODE,%ROWCOUNT,%ROWID
SET EmpZipLow="10000"
SET EmpZipHigh="19999"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name,Home_Zip
    INTO :name,:zip
    FROM Sample.Employee WHERE Home_Zip BETWEEN :EmpZipLow AND :EmpZipHigh)
&sql(OPEN EmpCursor)
IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg QUIT}
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,name," ",zip }
&sql(CLOSE EmpCursor)
IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg QUIT}
```

以下の埋め込み SQL の例では、[テーブル値関数](#)を query の FROM 節として使用しています。

## ObjectScript

```
SET $NAMESPACE="Samples"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name INTO :name FROM Sample.SP_Sample_By_Name('A')
    FOR READ ONLY)
&sql(OPEN EmpCursor)
IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE "Name=",name,! }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

## 関連項目

- ・ [CLOSE コマンド](#)
- ・ [FETCH コマンド](#)
- ・ [OPEN コマンド](#)
- ・ [WHERE CURRENT OF 節](#)
- ・ [SQL カーソル](#)

# DELETE (SQL)

テーブルの行を削除します。

## 構文

```
DELETE [%keyword] [FROM] table-ref [[AS] t-alias]
  [FROM [optimize-option] select-table [[AS] t-alias]
  {,select-table2 [[AS] t-alias}} ]
  [WHERE condition-expression]

DELETE [%keyword] [FROM] table-ref [[AS] t-alias]
  [WHERE CURRENT OF cursor]
```

## 引数

引数	説明
%keyword	オプション – %NOCHECK, %NOFLAN, %NOINDEX, %NOJURN, %NOLOCK, %NOTRIGGER, %PROFILE, %PROFILEALL のキーワード・オプションのうちの 1 つ、またはこれらのキーワードの空白で区切られたリスト。
FROM table-ref	行を削除するテーブル。これは、FROM 節ではありません。1 つのテーブル参照が続く FROM キーワードです(FROM キーワードはオプションで、table-ref は必須です)。  テーブル名 (またはビュー名) は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。未修飾の名前は、スキーマ検索パス (指定されている場合) または既定のスキーマ名を使用して、そのスキーマと照合されます。  テーブル参照の代わりに、テーブル行を削除できるビューを指定することも、括弧で囲んだサブクエリを指定することもできます。SELECT 文の FROM 節とは異なり、ここでは optimize-option キーワードは指定できません。この引数でテーブル値関数や JOIN 構文を指定することはできません。

引数	説明
FROM 節	<p>オプション – table-ref の後に指定する FROM 節。この <b>FROM</b> を使用して、削除する行を選択するために使用される 1 つ以上の select-table テーブルを指定できます。</p> <p>複数のテーブルは、コンマ区切りのリストとして指定するか、ANSI 結合キーワードで関連付けることができます。テーブルあるいはビューのあらゆる組み合わせを指定できます。ここで 2 つの select-table の間にコンマを指定する場合、InterSystems IRIS は複数のテーブルに <b>CROSS JOIN</b> を実行して、JOIN 処理の結果テーブルからデータを取得します。ここで 2 つの select-table の間に ANSI 結合キーワードを指定する場合、InterSystems IRIS は指定された結合処理を実行します。詳細は、このドキュメントの “<b>JOIN</b>” のページを参照してください。</p> <p>オプションで、クエリ実行を最適化するために、1 つ以上の optimize-option キーワードを指定できます。使用可能なオプションは、%ALLINDEX、%FIRSTTABLE tablename、%FULL、%INORDER、%IGNOREINDICES、%NOFLATTEN、%NOMERGE、%NOSVSO、%NOTOPOPT、%NOUNIONROPT、%PARALLEL、および %STARTTABLE です。詳細は、“<b>FROM</b>” 節を参照してください。</p>
AS t-alias	<p>オプション – <b>テーブルまたはビューの名前のエイリアス</b>。エイリアスは有効な識別子である必要があります。AS キーワードはオプションです。</p>
WHERE condition-expression	<p>オプション – 削除する行を限定する 1 つまたは複数のブーリアン述語を指定します。WHERE 節または WHERE CURRENT OF 節を指定することができます (両方は不可)。WHERE 節 (または WHERE CURRENT OF 節) が指定されていない場合、DELETE はテーブルからすべての行を削除します。詳細は、“<b>WHERE</b>” を参照してください。</p>
WHERE CURRENT OF cursor	<p>オプション: 埋め込み SQL のみ – DELETE 処理がカーソルの現在の位置でレコードを削除することを指定します。WHERE CURRENT OF 節または WHERE 節を指定することができます (両方は不可)。WHERE CURRENT OF 節 (または WHERE 節) が指定されていない場合、DELETE はテーブルからすべての行を削除します。詳細は、“<b>WHERE CURRENT OF</b>” を参照してください。</p>

## 概要

DELETE コマンドは、指定条件に適合する行をテーブルから削除します。テーブルからの行の削除は、直接、またはビュー経由が可能です。あるいは、サブクエリを使用して選択した行を削除できます。ビュー経由で削除する場合は、**CREATE VIEW** で説明されているように、必要条件や制限事項に従います。

また、DELETE 操作により、%ROWCOUNT ローカル変数に削除された行数が設定され、%ROWID ローカル変数に最後に削除された行の RowID 値が設定されます。行が削除されない場合、%ROWCOUNT=0 および %ROWID が定義されないか、前の値に設定されたままとなります。

table-ref は指定が必須です。table-ref の前の FROM キーワードは省略可能です。テーブルからすべての行を削除するには、以下のように指定します。

## SQL

```
DELETE FROM tablename
```

または



## SQL

DELETE tablename

これにより、テーブルからすべての行データが削除されますが、RowID、IDENTITY、ストリーム・フィールドの OID 値、および SERIAL (%Library.Counter) フィールドの各カウンタはリセットされません。TRUNCATE TABLE コマンドが、テーブルからすべての行データを削除し、これらのカウンタをリセットします。既定では、DELETE FROM tablename が削除トリガをプルします。削除トリガをプルしない場合は DELETE %NOTRIGGER FROM tablename と指定します。TRUNCATE TABLE は削除トリガをプルしません。

DELETE を使用する場合、一般的には condition-expression に基づいて削除する行を指定します。既定では、DELETE 処理はテーブルのすべての行をチェックし、condition-expression を満たすすべての行を削除します。condition-expression を満たす行がない場合、DELETE は正常に終了して SQLCODE=100 (データがこれ以上ありません) を設定します。

WHERE 節または WHERE CURRENT OF 節を指定することができます (両方は不可)。WHERE CURRENT OF 節が使用される場合、DELETE 処理は現在のカーソル位置のレコードを削除します。WHERE CURRENT OF を使用する DELETE の例は、後述の“[埋め込み SQL とダイナミック SQL の例](#)”を参照してください。指定位置での実行の詳細は、“[WHERE CURRENT OF](#)”を参照してください。

既定では、DELETE は、全か無かのイベントです。指定されたすべての行が削除されるか、削除されないかのいずれかです。InterSystems IRIS は、DELETE の成功または失敗を示す SQLCODE ステータス変数を設定します。

テーブルから行を削除するには、以下の条件を満たしている必要があります。

- ・ テーブルは、現在の (または指定された) ネームスペースに存在している必要があります。指定されたテーブルが見つからない場合、InterSystems IRIS は SQLCODE -30 エラーを発行します。
- ・ ユーザは、指定されたテーブルに対する DELETE 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対する DELETE 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する DELETE 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ name ” に設定されます。[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが DELETE 特権を持っているかどうかを確認できます。[GRANT](#) コマンドを使用して、指定したテーブルに DELETE 特権を割り当てることができます。詳細は、“[特権](#)”を参照してください。
- ・ 他のプロセスはテーブルを IN EXCLUSIVE MODE でロックできません。ロックされているテーブルから行を削除しようとすると、SQLCODE -110 エラーになり、%msg は “RowID = '10' Sample.Person DELETE ” になります。SQLCODE -110 エラーは、DELETE 文が削除される最初のレコードを検出したときにのみ発行され、タイムアウト期間内にロックできないことに注意してください。
- ・ DELETE コマンドの WHERE 節で、存在しないフィールドを指定すると、SQLCODE -29 が発行されます。指定したテーブルに定義されているフィールド名をすべてリストするには、“[列の名前と番号](#)”を参照してください。フィールドは存在するが DELETE コマンドの WHERE 節の条件を満たすフィールド値がない場合は、影響を受ける行はないため、SQLCODE 100 (データの末尾) が発行されます。
- ・ テーブルを READONLY として定義することはできません。読み取り専用テーブルを参照する DELETE をコンパイルすると、SQLCODE -115 エラーが発生します。このエラーは実行時にのみ発生するのではなく、コンパイル時にも発生するようになったことに注意してください。“[永続クラスのその他のオプション](#)”の READONLY オブジェクトの説明を参照してください。
- ・ ビュー経由で削除する場合、ビューを WITH READ ONLY として定義することはできません。これを実行しようとすると、SQLCODE -35 エラーが返されます。ビューがシャード・テーブルに基づく場合、WITH CHECK OPTION で定義されたビューを介して DELETE を実行することはできません。これを試みると、SQLCODE -35 が生じ、%msg が “ (sample.myview) with check option INSERT/UPDATE/DELETE ” に設定されます。詳細は、“[CREATE VIEW](#)” コマンドを参照してください。同様に、サブクエリ経由で削除する場合は、そのサブクエリが更新可能である必要があります。例えば、サブクエリ DELETE FROM (SELECT COUNT(\*) FROM Sample.Person) AS x を実行すると、SQLCODE -35 エラーが発生します。

- 削除する行が存在している必要があります。通常、存在しない行を削除しようとすると、指定された行が見つからないため、SQLCODE 100 (これ以上データがない) が返されます。ただしまれに、%NOLOCK が指定された DELETE で削除する行が見つかったが別のプロセスによりその行が即座に削除された場合には、SQLCODE -106 が返されます。このエラーの %msg は、テーブル名と RowID をリストします。
- 削除対象に指定されたすべての行が削除可能である必要があります。既定では、1 行以上の行が削除不可能である場合、DELETE 操作は失敗し、行は削除されません。削除する行が別の同時プロセスによってロックされている場合、DELETE は SQLCODE -110 エラーを発行します。%NOCHECK が指定されていない状態で、指定された行のいずれかの削除操作が外部キーの参照整合性に違反する場合、DELETE は SQLCODE -124 エラーを発行します。この既定の動作は以下のように変更できます。
- システム提供の特定 %SYS ネームスペース機能が削除されないように保護されています。例えば、DELETE FROM Security.Users は、\_SYSTEM、\_PUBLIC、または UnknownUser の削除には使用できません。これを実行しようとすると、SQLCODE -134 エラーが返されます。

## FROM 構文

DELETE コマンドには、テーブルを指定する 2 つの FROM キーワードを含めることができます。この 2 つの FROM の使用法は根本的に異なります。

- table-ref の前の FROM で、1 つ以上の行を削除するテーブル (またはビュー) を指定します。これは FROM キーワードであり、FROM 節ではありません。テーブルは 1 つのみ指定できます。結合構文または optimize-option キーワードを指定することはできません。FROM キーワード自体はオプションで、table-ref は必須です。
- table-ref の後の FROM はオプションの FROM 節で、削除する行を特定するのに使用できます。1 つ以上のテーブルを指定できます。結合構文や optimize-option キーワードを含め、SELECT 文に使用できるすべての FROM 節の構文がサポートされます。この FROM 節は、(常にではありませんが) 通常は WHERE 節と共に使用します。

したがって、以下のいずれも有効な構文形式になります。

```
DELETE FROM table WHERE ... DELETE table WHERE ... DELETE
FROM table FROM table2 WHERE ... DELETE table FROM table2 WHERE ...
```

この構文では、Transact-SQL と互換性のある方法で複雑な選択条件がサポートされます。

以下の例は、2 つの FROM キーワードを使用する方法を示しています。ここでは、Retirees テーブル内に同じ EmpId がある場合、それらのレコードが Employees テーブルから削除されます。

## SQL

```
DELETE FROM Employees AS Emp
FROM Retirees AS Rt
WHERE Emp.EmpId = Rt.EmpId
```

2 つの FROM キーワードで同じテーブルを参照する場合には、参照するテーブルが文字どおり同じであることも、テーブルの 2 つのインスタンスの結合であることもあります。これは、テーブルのエイリアスの使用方法によって異なります。

- 両方のテーブル参照にエイリアスがない場合には、両方とも同じテーブルを参照します。

```
DELETE FROM table1 FROM table1,table2 /* join of 2 tables */
```

- 両方のテーブル参照に同じエイリアスがある場合には、両方とも同じテーブルを参照します。

```
DELETE FROM table1 AS x FROM table1 AS x,table2 /* join of 2 tables */
```

- 両方のテーブル参照にエイリアスがあり、それぞれのエイリアスが異なる場合には、InterSystems IRIS は 2 つのテーブルのインスタンスの結合を実行します。

```
DELETE FROM table1 AS x FROM table1 AS y,table2 /* join of 3 tables */
```

- 最初のテーブル参照にエイリアスがあり、2 番目にはない場合には、InterSystems IRIS は 2 つのテーブルのインスタンスの結合を実行します。

```
DELETE FROM table1 AS x FROM table1,table2 /* join of 3 tables */
```

- 最初のテーブル参照にエイリアスがなく、2 番目にはエイリアスのあるテーブルへの単一の参照がある場合には、両方とも同じテーブルを参照し、このテーブルには指定されたエイリアスがあります。

```
DELETE FROM table1 FROM table1 AS x,table2 /* join of 2 tables */
```

- 最初のテーブル参照にエイリアスがなく、2 番目にはテーブルへの複数の参照がある場合には、それぞれのエイリアスされたインスタンスは個別のテーブルと見なされ、それらのテーブルの結合が実行されます。

```
DELETE FROM table1 FROM table1,table1 AS x,table2 /* join of 3 tables */
DELETE FROM table1 FROM table1 AS x,table1 AS y,table2 /* join of 4 tables */
```

## %Keyword オプション

%keyword 引数を指定すると、以下のように処理を制限します。

- %NOCHECK** – 削除される行を参照する外部キーの参照整合性チェックを抑制します。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する **%NOCHECK 管理特権** を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %NOCHECK " に設定されます。
- %NOFPLAN** – この操作の凍結されたプラン (ある場合) を無視して、新しいクエリ・プランを生成します。凍結されたプランは保持されますが、使用されません。詳細は、“[凍結プランの構成](#)” を参照してください。
- %NOINDEX** – 削除される行のために全インデックスでのインデックス・エントリの削除を抑制します。テーブル・インデックス内に孤立した値が残るため、これは十分に注意して使用する必要があります。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する **%NOINDEX 管理特権** を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %NOINDEX " に設定されます。
- %NOJOURN** – 削除操作の間にジャーナリングを抑制し、トランザクションを無効化します。プルされたトリガを含め、行での変更はどれもジャーナリングされません。**%NOJOURN** が指定された文の後で ROLLBACK を実行した場合、その文で行われた変更はロールバックされません。この制限を適用するには、現在のネームスペースで **%NOJOURN 管理特権** が必要です。この管理特権がない場合は SQLCODE -99 エラーが発生し、%msg が " 'name' %NOJOURN " に設定されます。
- %NOLOCK** – 削除される行の行のロックを抑制します。単独のユーザ/処理がデータベースを更新する際にのみ使用します。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する **%NOLOCK 管理特権** を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %NOLOCK " に設定されます。
- %NOTRIGGER** – DELETE 処理中にベース・テーブル・トリガの実行を抑制して、トリガが引き出されないようにします。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する **%NOTRIGGER 管理特権** を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %NOTRIGGER " に設定されます。
- %PROFILE** または **%PROFILE\_ALL** – これらのキーワード指示文のいずれかが指定された場合、SQLStats 収集コードが生成されます。これは、PTools を ON にして生成されるものと同じコードです。違いは、SQLStats 収集コードはこの特定の文に対してのみ生成されるという点です。コンパイルされるルーチン/クラス内のその他すべての SQL 文は、PTools が OFF であるかのようにコードを生成します。これにより、ユーザは、調査されない SQL 文の関係のない統計を収集することなく、アプリケーション内の特定の問題の SQL 文をプロファイリング/調査できます。詳細は、“[SQL 実行時統計](#)” を参照してください。

%PROFILE はメイン・クエリ・モジュールに対して SQLStats を収集します。%PROFILE\_ALL はメイン・クエリ・モジュールとそのすべてのサブクエリ・モジュールに対して SQLStats を収集します。

複数の %keyword 引数を順不同で指定できます。複数の引数は、空白で区切られます。

親レコードの削除時に %keyword 引数を指定すると、対応する子レコードの削除時に同じ %keyword 引数が適用されます。

## 参照整合性

%NOCHECK を指定しない場合、InterSystems IRIS では、システム全体の構成設定を使用して外部キーの参照整合性チェックを実行するかどうかが決まります。既定では、外部キーの参照整合性チェックを実行します。“[外部キーの参照整合性チェック](#)”の説明に従って、この既定値をシステム全体で設定できます。現在のシステム全体の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。

DELETE 操作時には、すべての外部キー参照について、参照されるテーブルの該当する行に対する共有ロックが取得されます。この行は、トランザクションの終了までロックされています。これにより、参照される行は、DELETE のロールバックがあってもそれより前に変更されることがなくなります。

一連の外部キー参照が CASCADE として定義されている場合は、DELETE 操作により、循環参照が発生する可能性があります。InterSystems IRIS では、CASCADE 参照動作のある DELETE による循環参照ループの反復を防止します。元のテーブルに戻ると、InterSystems IRIS は、カスケード・シーケンスを終了します。

CASCADE、SET NULL、または SET DEFAULT で定義された外部キー・フィールドに対して、%NOLOCK を指定して DELETE 操作を実行した場合は、対応する参照アクションが %NOLOCK によって実行されて外部キー・テーブルが変更されます。

## アトミック性

既定では、DELETE、UPDATE、INSERT、および TRUNCATE TABLE はアトミック処理として実行されます。DELETE は、正常に完了するか、すべての操作がロールバックされるかのいずれかです。指定した行のいずれかを削除できない場合、指定した行は 1 行も削除できずにデータベースは DELETE を発行する前の状態に戻ります。

現在のプロセスに対するこの既定は、[SET TRANSACTION %COMMITMODE](#) を呼び出すことによって SQL 内で変更できます。現在のプロセスに対するこの既定は、SetOption() メソッドを SET

status=\$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval) のように呼び出すことによって ObjectScript 内で変更できます。以下の intval 整数オプションを使用できます。

- ・ 1 または IMPLICIT (自動コミットがオン) – 上記のように、これが既定の動作です。DELETE ごとに個別のトランザクションが構成されます。
- ・ 2 または EXPLICIT (自動コミットがオフ) – 進行中のトランザクションがない場合は、DELETE コマンドによってトランザクションは自動的に開始されます。ただし、COMMIT または ROLLBACK で明示的にトランザクションを終了する必要があります。EXPLICIT モードでは、トランザクションあたりのデータベース操作の数は、ユーザ定義です。
- ・ 0 または NONE (自動トランザクションなし) – DELETE を呼び出してもトランザクションは開始されません。DELETE 操作の失敗により、指定された行の一部が削除されたり削除されなかったりすることで、データベースが整合性のない状態になる可能性があります。このモードでトランザクションのサポートを提供するには、START TRANSACTION を使用してトランザクションを開始し、COMMIT または ROLLBACK を使用してトランザクションを終了する必要があります。

[シャード・テーブル](#)は常に、自動トランザクションなしのモードに設定されます。つまり、シャード・テーブルに対する挿入、更新、および削除はすべて、トランザクションの範囲外で実行されます。

現在のプロセスのアトミック性設定を確認するには、以下の ObjectScript の例のように、GetOption("AutoCommit") メソッドを使用します。



## ObjectScript

```

SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }

```

## トランザクションでのロック

%NOLOCK を指定しない場合、INSERT、UPDATE、および DELETE 操作時に自動的にレコードに標準のロックがかかります。影響を受ける各レコード (行) は、現在のトランザクションが継続している間はロックされます。

既定のロックしきい値は、テーブルごとに 1000 ロックです。つまり、トランザクションの間にテーブルから 1000 件を超えるレコードを削除すると、ロックのしきい値に到達し、InterSystems IRIS は自動的にロック・レベルをレコード・ロックからテーブル・ロックに上げます。これによってトランザクション時に、ロック・テーブルをオーバーフローすることなく、大規模な削除を実行できます。

InterSystems IRIS は、以下の 2 つのロック・エスカレーション策のどちらかを適用します。

- ・ “E” タイプのロック・エスカレーション: InterSystems IRIS は、次のことに該当する場合にこのタイプのロック・エスカレーションを使用します。(1) クラスで [%Storage.Persistent](#) が使用されている (これは、管理ポータル の SQL スキーマ表示で [\[カタログの詳細\]](#) から判断できます)。(2) クラスで、IDKey インデックスが指定されていないか、単一プロパティの IDKey インデックスが指定されている。“E” タイプのロック・エスカレーションについては、“[LOCK](#)” コマンドで説明されています。
- ・ 従来の SQL ロック・エスカレーション: クラスで “E” タイプのロック・エスカレーションが使用されない理由は、マルチプロパティの IDKey インデックスの存在にあると考えられます。この場合は、%Save ごとにロック・カウンタがインクリメントされます。つまり、トランザクション内の単一オブジェクトを 1001 回保存を行うと、InterSystems IRIS はロックのエスカレーションを試みます。

どちらのロック・エスカレーション策の場合も、\$SYSTEM.SQL.Util.GetOption(“LockThreshold”) メソッドを使用して、現在のシステム全体用ロックしきい値を決定できます。既定は 1000 です。このシステム全体のロックしきい値は、以下の方法を使用して設定できます。

- ・ \$SYSTEM.SQL.Util.SetOption(“LockThreshold”) メソッドを使用します。
- ・ 管理ポータルを使用して、[\[システム管理\]](#)、[\[構成\]](#)、[\[SQL およびオブジェクトの設定\]](#)、[\[SQL\]](#) の順に選択します。[\[ロック・エスカレーションしきい値\]](#) の現在の設定を表示して編集します。既定は 1000 ロックです。この設定を変更すると、変更後に開始される新しいプロセスは、新しい設定になります。

ロックしきい値を変更するには、%Admin Manage Resource の USE 許可が必要です。InterSystems IRIS は、ロックしきい値の変更を現在のプロセスすべてに即座に適用します。

結果として、自動ロック・エスカレーションでは、デッドロックの状況が起こる可能性があります。つまり、テーブル・ロックへのエスカレーションを試みたときに、テーブル内のレコード・ロックを保持する別プロセスとの競合が起こる可能性があります。これを避けるための方策としては、次のいくつかが考えられます。(1) ロック・エスカレーションがトランザクション内で起こる可能性が低くなるように、ロック・エスカレーションのしきい値を上げる。(2) ロック・エスカレーションが即座に起こるように、ロック・エスカレーションのしきい値を大幅に下げる。これにより、別プロセスが同一テーブル内のレコードをロックする機会が少なくなります。(3) トランザクションが継続している間はテーブル・ロックを適用し、レコード・ロックは実行しない。これは、LOCK TABLE、UNLOCK TABLE (テーブル・ロックがトランザクションの終了まで持続するよう、IMMEDIATE キーワードはなし) の順に指定することで、トランザクション開始時に実行できます。その後、%NOLOCK オプションを使用して削除を実行します。

自動ロック・エスカレーションは、ロック・テーブルのオーバーフローを防ぐことを目的としています。ただし、大量の削除などを実行したために <LOCKTABLEFULL> エラーが発生した場合は、DELETE によって SQLCODE -110 エラーが発行されます。

トランザクションでのロックの詳細は、“[トランザクション処理](#)” を参照してください。

## 例

以下の例は、両方とも、TempEmployees テーブルからすべての行を削除します。FROM キーワードはオプションです。

### SQL

```
DELETE FROM TempEmployees
```

### SQL

```
DELETE TempEmployees
```

以下の例は、Employees テーブルから、従業員番号 234 番を削除します。

### SQL

```
DELETE
  FROM Employees
 WHERE EmpId = 234
```

以下の例は、CurStatus 列が “Retired” に設定されているすべての行を、ActiveEmployees テーブルから削除します。

### SQL

```
DELETE FROM ActiveEmployees
 WHERE CurStatus = 'Retired'
```

以下の例は、サブクエリを使用して行を削除します。

### SQL

```
DELETE FROM (SELECT Name, Age FROM Sample.Person WHERE Age > 65)
```

## テーブルの削除例

以下は、新たに作成したテーブルから行を削除し、その後、テーブル自体を削除する例です。

この例の最初のコマンドで、3 つの列を持つ SQLUser.WordPairs というテーブルを作成します。

### SQL

```
CREATE TABLE SQLUser.WordPairs (
  Lang      CHAR(2) NOT NULL,
  Firstword CHAR(30),
  Lastword  CHAR(30))
```

続くいくつかのコマンドで、このテーブルに 6 つのレコードを挿入します。



## SQL

```
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
('En','hello','goodbye')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
('Fr','bonjour','au revoir')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
('It','pronto','ciao')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
('Fr','oui','non')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
('En','howdy','see ya')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
('Es','hola','adios')
```

以下のコマンドでは、[カーソル・ベースの埋め込み SQL](#) を使用して、すべての英語のレコードを削除します。

## ObjectScript

```
#sqlcompile path=Sample
NEW %ROWCOUNT,%ROWID
&sql(DECLARE WPCursor CURSOR FOR
    SELECT Lang FROM WordPairs
    WHERE Lang='En')
&sql(OPEN WPCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH WPCursor)
    QUIT:SQLCODE
    &sql(DELETE FROM WordPairs
        WHERE CURRENT OF WPCursor)
    IF SQLCODE=0 {
        WRITE !,"Delete succeeded"
        WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
    ELSE {
        WRITE !,"Delete failed, SQLCODE=",SQLCODE }
    }
&sql(CLOSE WPCursor)
```

さらに、以下のコマンドで、すべてのフランス語のレコードを削除します。

## SQL

```
DELETE FROM WordPairs WHERE Lang='Fr'
```

最後の 2 つのコマンドで、テーブルに残りのレコードを表示し、そのテーブルを削除します。

## SQL

```
SELECT %ID,* FROM SQLUser.WordPairs
DROP TABLE SQLUser.WordPairs
```

## 関連項目

- ・ [FROM](#)
- ・ [TRUNCATE TABLE](#)
- ・ [INSERT UPDATE](#)
- ・ [CREATE VIEW](#)
- ・ [WHERE](#)
- ・ [WHERE CURRENT OF](#)
- ・ [データベースの変更](#)
- ・ [テーブルの定義](#)
- ・ [ビューの定義](#)

- ・ [トランザクション処理](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP AGGREGATE (SQL)

---

ユーザ定義の集約関数を削除します。

### 構文

```
DROP AGGREGATE [IF EXISTS] name
```

### 説明

DROP AGGREGATE コマンドは、ユーザ定義の集約関数 (UDAF) を削除します。ユーザ定義の集約関数を作成するには、[CREATE AGGREGATE](#) コマンドを使用します。

存在しない UDAF の削除を試みると、SQL は SQLCODE -428 エラーを発行します。生成されるメッセージは ”  
Sample.SecondHighest ” のようになります。

UDAF を削除すると、その UDAF を参照するすべてのクエリ・キャッシュが自動的に削除されます。

### 引数

#### **name**

削除するユーザ定義の集約関数の名前。name は修飾 (schema.aggname)、未修飾 (aggname) のどちらでもかまいません。未修飾の name では[既定のスキーマ名](#)が使用されます。

### 関連項目

- ・ [CREATE AGGREGATE](#) コマンド
- ・ [集約関数の概要](#)
- ・ [SQLCODE エラー・メッセージ](#)

# DROP DATABASE (SQL)

データベース (ネームスペース) を削除します。

## 構文

```
DROP DATABASE [IF EXISTS] dbname [RETAIN_FILES]
```

## 説明

DROP DATABASE コマンドは、1 つのネームスペース、およびそれに関連するデータベースを削除します。

指定した dbname は、ネームスペース、および対応するデータベース・ファイルを含むディレクトリの名前となります。dbname は識別子として指定します。ネームスペース名は、大文字と小文字を区別しません。指定した dbname ネームスペースが存在しない場合、InterSystems IRIS は SQLCODE -340 エラーを発行します。

DROP DATABASE コマンドは特権を必要とする操作です。DROP DATABASE を使用する前に、%Admin\_Manage リソースを有するユーザとしてログインする必要があります。ユーザには、ルーチンおよびグローバルなデータベース定義のために、そのリソースの READ 許可も必要となります。特権がない場合は、SQLCODE -99 エラー (特権違反) が返されます。

\$SYSTEM.Security.Login() メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“%SYSTEM.Security” を参照してください。

DROP DATABASE は、特権に関係なく、システム・ネームスペースの削除には使用できません。これを実行しようとする、SQLCODE -342 エラーが返されます。

DROP DATABASE は、現在使用しているか接続しているネームスペースの削除には使用できません。これを実行しよう、SQLCODE -344 エラーが返されます。

管理ポータルを使用してネームスペースを削除することもできます。**[システム管理]**、**[構成]**、**[システム構成]**、**[ネームスペース]** の順に選択して、既存のネームスペースをリストします。削除するネームスペースのための **[削除]** ボタンをクリックします。

## RETAIN\_FILES

このオプションを指定した場合は、物理ファイル構造を保持して、データベースおよびそれに関連するネームスペースを削除します。この操作を実行した後に、dbname を使用するために以下のことを試みると、次のような結果となります。

- RETAIN\_FILES を指定しないで DROP DATABASE を使用する場合は、この物理ファイル構造を削除できません。代わりに、SQLCODE -340 エラー (データベースが見つかりません) が返されます。
- RETAIN\_FILES を指定して DROP DATABASE を使用する場合も、SQLCODE -340 エラー (データベースが見つかりません) が返されます。
- CREATE DATABASE を使用する場合は、新規データベースを同じ名前で作成することができません。代わりに、SQLCODE -341 エラー (データベースのためのデータベース・ファイルを作成できません) が返されます。
- このネームスペースを使用しよう、<NAMESPACE> エラーが返されます。

## サーバ Init コードおよび切断コード

\$SYSTEM.SQL.Util.SetOption("ServerInitCode",value) および \$SYSTEM.SQL.Util.SetOption("ServerDisconnectCode",value) メソッドを使用して、サーバ Init コードとサーバ切断コードをネームスペースに割り当てることができます。対応する \$SYSTEM.SQL.Util.GetOption() メソッド・オプションを使用して、現在の値を確認できます。

DROP DATABASE または他のインタフェースを使用してネームスペースを削除すると、これらのサーバ Init コードとサーバ切断コードの値は削除されます。したがって、ネームスペースを削除してから再作成する場合は、これらの値を再度指定する必要があります。

## 引数

### IF EXISTS

指定すると、存在しないデータベースに対してコマンドが実行される場合にエラーを抑制する引数 (オプション)。

### dbname

削除するデータベース (ネームスペース) の名前。

### RETAIN\_FILES

指定した場合、物理データベース・ファイル (IRIS.DAT ファイル) は削除されなくなる引数 (オプション)。既定では、.DAT ファイルはネームスペースおよび他のデータベース・エンティティと共に削除されます。

## 例

以下の例では、ネームスペースとそれに関連するデータベース (この場合は、'c:¥InterSystems¥IRIS¥mgr¥DocTestDB') を削除します。物理データベース・ファイルは保持されます。

### SQL

```
CREATE DATABASE DocTestDB ON DIRECTORY 'c:\InterSystems\IRIS142\mgr\DocTestDB'
```

### SQL

```
DROP DATABASE DocTestDB RETAIN_FILES
```

## 関連項目

- ・ [CREATE DATABASE コマンド](#)
- ・ [USE DATABASE コマンド](#)

# DROP FOREIGN SERVER (SQL)

外部サーバを削除します。

## 構文

```
DROP [ FOREIGN ] SERVER server-name [ RESTRICT | CASCADE ]
```

## 引数

引数	説明
<i>server-name</i>	削除する外部サーバの名前。この名前は有効な識別子である必要があります。コマンドを正常に実行するには、この名前の外部サーバが存在する必要があります。
RESTRICT	オプション - 何も定義されていない場合は、外部サーバを削除することを指定します。このオプションは既定の動作を提供します。
CASCADE	オプション - 外部サーバ内で定義されたオブジェクト（テーブルを含む）はすべて、外部サーバと共に削除されることを指定します。

## 説明

DROP FOREIGN SERVER コマンドは、外部テーブルをホストするよう構成された外部サーバを削除します。

既定では、このコマンドは外部テーブルが定義されていない外部サーバのみを削除します。RESTRICT キーワードを指定することで、この動作を明示的に適用できます。RESTRICT キーワードが暗黙的または明示的に指定されている場合、1 つ以上のテーブルが定義されている外部サーバを削除しようとすると、SQLCODE -321 エラーが生成されます。

CASCADE が指定されている場合、DROP FOREIGN SERVER は外部サーバと外部サーバで定義されたすべてのテーブルを正常に削除します。

外部テーブルを削除するには、以下の条件を満たす必要があります。

- 外部サーバが、現在のネームスペースに存在している必要があります。存在しない外部サーバを削除しようとすると、SQLCODE -30 エラーが生成されます。
- 外部サーバを削除するには、必要な特権を持っている必要があります。特権を持たずに外部サーバを削除しようとすると、SQLCODE -99 エラーが生成されます。

## 例

以下の例では、テーブルが定義されていない外部サーバを削除します。

```
DROP FOREIGN SERVER EmptyServer RESTRICT
```

以下の例では、テーブルが定義されている外部サーバを削除します。外部サーバを削除する過程で、関連付けられているテーブルも削除されます。

```
DROP FOREIGN SERVER FullServer CASCADE
```

## 関連項目

- [CREATE FOREIGN SERVER](#)
- [ALTER FOREIGN SERVER](#)
- [DROP FOREIGN TABLE](#)



## DROP FOREIGN TABLE (SQL)

外部テーブルを削除します。

### 構文

```
DROP FOREIGN TABLE [ IF EXISTS ] table-name [ RESTRICT | CASCADE ]
```

### 引数

引数	説明
IF EXISTS	オプション - table-name を持つ外部テーブルが存在しない場合に発生するエラーを抑制します。
table-name	削除する外部テーブルの名前。この名前は有効な識別子である必要があります。コマンドを正常に実行するには、この名前の外部テーブルが存在する必要があります。
RESTRICT	オプション - ビューなどの SQL オブジェクトが外部テーブルで定義されている場合、その外部テーブルを削除すべきではないことを指定します。このオプションは既定の動作を提供します。
CASCADE	オプション - 外部テーブルで定義されたオブジェクト (ビューなど) はすべて、外部テーブルと共に削除されることを指定します。

### 説明

DROP FOREIGN TABLE コマンドは、外部サーバから外部テーブルを削除します。

既定では、このコマンドはビューが関連付けられていない外部テーブルのみを削除します。RESTRICT キーワードを指定することで、この動作を明示的に適用できます。RESTRICT キーワードが暗黙的または明示的に指定されている場合、関連付けられているビューと共に外部テーブルを削除しようとすると、SQLCODE -321 エラーが生成されます。

CASCADE キーワードが指定されている場合、DROP FOREIGN TABLE はテーブルとテーブルに関連付けられているすべてのビューを正常に削除します。

外部テーブルを削除するには、以下の条件を満たす必要があります。

- 外部テーブルが、現在のネームスペース内の外部サーバに存在している必要があります。存在しない外部テーブルを削除しようとすると、SQLCODE -30 エラーが生成されます。このエラーは、IF EXISTS キーワードを指定することで抑制できます。
- 外部テーブルを削除するには、必要な特権を持っている必要があります。特権を持たずにテーブルを削除しようとすると、SQLCODE -99 エラーが生成されます。

### 例

以下の例では、オブジェクトが定義されていない外部テーブルを削除します。

```
DROP FOREIGN TABLE Example.MyTable RESTRICT
```

以下の例では、外部テーブルと外部テーブルに関連付けられているビューを削除します。

```
DROP FOREIGN TABLE Example.MyTable CASCADE
```

### 関連項目

- [CREATE FOREIGN TABLE](#)

- [ALTER FOREIGN SERVER](#)
- [DROP FOREIGN SERVER](#)

## DROP FUNCTION (SQL)

関数を削除します。

## 構文

DROP FUNCTION [IF EXISTS] *name* [ FROM *className* ]

## 概要

DROP FUNCTION コマンドは関数を削除します。関数を削除すると、InterSystems IRIS は、権限が与えられている全ユーザおよびロールを無効にして、データベースから削除します。

関数の削除には、**GRANT** コマンドで指定された **%DROP\_FUNCTION** 管理者特権が必要です。この特権がないと、**SQLCODE -99 エラー (特権違反)** が生成されます。

関数は、その関数定義を含むクラス定義が**導入済みのクラス**の場合、削除することはできません。この操作はSQLCODE -400 エラーで失敗し、%msg が “                    classname                    DDL                    ” に設定されます。

name と FROM className の次のような組み合わせがサポートされています。FROM 節では、SQL 名ではなく、クラス・パッケージ名と関数名を指定することに注意してください。これらの例では、システム全体の既定のスキーマ名は、User クラス・パッケージに対応する SQLUser です。

- ・ `DROP FUNCTION BonusCalc FROM funcBonusCalc` : 関数 `SQLUser.BonusCalc()` を削除します。
- ・ `DROP FUNCTION BonusCalc FROM User.funcBonusCalc` : 関数 `SQLUser.BonusCalc()` を削除します。
- ・ `DROP FUNCTION Test.BonusCalc FROM funcBonusCalc` : 関数 `SQLUser.BonusCalc()` を削除します。
- ・ `DROP FUNCTION BonusCalc FROM Employees.funcBonusCalc` : 関数 `Employees.BonusCalc()` を削除します。
- ・ `DROP FUNCTION Test.BonusCalc FROM Employees.funcBonusCalc` : 関数 `Employees.BonusCalc()` を削除します。

指定された関数が存在しない場合、DROP FUNCTION は SQLCODE -362 エラーを発行します。指定されたクラスが存在しない場合、DROP FUNCTION は SQLCODE -360 エラーを発行します。指定された関数が複数の関数を参照する可能性がある場合、DROP FUNCTION は SQLCODE -361 エラーを生成します。このあいまいさを解消するには、className を指定する必要があります。

## 引数

IF EXISTS

存在しない関数に対してコマンドが実行される場合にエラーを抑制する引数 (オプション)。

## name

削除される関数の名前。[識別子](#)の名前です。関数のパラメータの括弧は指定しないでください。name は修飾 (schema.name)、未修飾 (name) のどちらでもかまいません。FROM className 節が指定されていない限り、関数名が未修飾の場合は[システム全体の既定のスキーマ名](#)が使用されます。

## FROM className

指定があれば、FROM className 節は与えられたクラスから関数を削除します。関数の SQL 名 (BonusCalc) ではなく、className (funcBonusCalc) を指定する必要があります。FROM 節が指定されていない場合、InterSystems IRIS はそのスキーマのすべてのクラスで該当する関数を検索し、削除します。しかし、この名前での関数が見つからない場合、また

は複数の同名の関数が見つかった場合、エラー・コードが返されます。関数の削除の結果、クラスが空になる場合、DROP FUNCTION はそのクラスも削除します。

## 例

以下の例は、クラス User.Employee からの myfunc の削除を試行します。(クラス User.Employee の作成例は、“CREATE TABLE” を参照してください)。

## SQL

```
DROP FUNCTION myfunc FROM User.Employee
```

## 関連項目

- ・ [CREATE FUNCTION](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP INDEX (SQL)

インデックスを削除します。

### 構文

```
DROP INDEX [IF EXISTS] [%NOJOURN] index-name [ON [TABLE] table-name]
```

```
DROP INDEX [IF EXISTS] table-name.index-name
```

### 概要

DROP INDEX 文はテーブル定義からインデックスを削除します。DROP INDEX を使用して、標準のインデックス、ビットマップ・インデックス、またはビットスライス・インデックスを削除できます。また、DROP INDEX を使用すると、対応する Unique インデックスを削除することにより、一意の制約または主キー制約を削除できます。ビットマップ・エクステント・インデックスやマスタ・マップ (データ/マスタ) IDKEY インデックスの削除に DROP INDEX を使用することはできません。

インデックスを削除する理由には、次のようなものがあります。

- ・ テーブルに対して頻繁に INSERT、UPDATE、DELETE を実行する必要がある場合。このような各操作でインデックスに書き込みが行われ、パフォーマンスのオーバーヘッドが増加する場合は、各操作に %NOINDEX オプションを使用することができます。または、状況によっては、インデックスを削除し、データベースへの変更を一括して行った後、インデックスを再作成および生成する方法もあります。
- ・ クエリ操作に使用されないフィールド、またはフィールドの組み合わせにインデックスが存在する場合。この場合、インデックスをメンテナンスするパフォーマンスのオーバーヘッドがあまり大きくない場合があります。
- ・ 現在、重複するデータが大量に含まれているフィールドまたはフィールドの組み合わせにインデックスが存在する場合。この場合、クエリのパフォーマンスはあまり改善されない場合があります。

IDKEY インデックスは、テーブルにデータがあるときは削除できません。これを実行しようとする、SQLCODE -325 エラーが生成されます。

### 特権とロック

DROP INDEX コマンドは特権を必要とする操作です。DROP INDEX を実行するには、ユーザは %ALTER\_TABLE 管理特権を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %ALTER\_TABLE " に設定されます。適切な付与特権を持っていれば、GRANT コマンドを使用して、ユーザまたはロールに %ALTER\_TABLE 特権を割り当てることができます。管理特権はネームスペース固有のものです。詳細は、“[特権](#)”を参照してください。

ユーザは、指定されたテーブルに対する %ALTER 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対する %ALTER 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する %ALTER 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' 'Schema.TableName' %ALTER " に設定されます。%CHECKPRIV コマンドを呼び出すことにより、現在のユーザが %ALTER 特権を持っているかどうかを確認できます。GRANT コマンドを使用して、指定したテーブルに %ALTER 特権を割り当てることができます。詳細は、“[特権](#)”を参照してください。

- ・ DROP INDEX は、テーブル・クラスの定義に [DdlAllowed] が含まれている場合を除き、[永続クラスから投影されたテーブル](#)では使用できません。使用すると、操作は SQLCODE -300 エラーで失敗し、%msg が “DDL schema.tablename ” に設定されます。
- ・ DROP INDEX は、[導入済みの永続クラス](#)から投影されたテーブルでは使用できません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ classname DDL ” に設定されます。

DROP INDEX 文は、table-name に対してテーブル・レベルのロックを取得します。これにより、他のプロセスはこのテーブルのデータを変更できなくなります。このロックは、DROP INDEX 操作が終了すると自動的に解除されます。

## インデックス名

インデックスを作成するために index-name を指定すると、システムは対応するクラス・インデックス名を句読点を削除して生成し、指定した index-name をインデックスの SqlName 値としてクラスに保持します (SQL マップ名)。index-name を DROP INDEX に指定する際、[SQL マップ名] としてテーブルの [管理ポータル](#) の [SQL の \[カタログの詳細\]](#) にリストされる、句読点を含む名前を指定します。例えば、[インデックス名] (MYTABLEUNIQUE2) ではなく、Unique 制約の生成された [SQL マップ名] (MYTABLE\_UNIQUE2) を指定します。この index-name では、大文字と小文字は区別されません。

## テーブル名

インデックスに関連するテーブルは、以下の DROP INDEX 構文形式を使用して指定できます。

- index-name ON TABLE 構文：テーブル名の指定はオプションです。省略した場合、InterSystems IRIS はネームスペース内のすべてのクラスから、対応するインデックスを検索します。
- table-name.index-name 構文：テーブル名の指定は必須です。

どちらの構文でも、テーブル名は未修飾 (table) または修飾 (schema.table) のいずれでもかまいません。スキーマ名を省略すると、[既定のスキーマ名](#)が使用されます。

DROP INDEX でテーブル名が指定されていない場合、InterSystems IRIS は index-name と一致するインデックス SqlName のすべてのインデックス、またはインデックスに対して SqlName が指定されていない場合はインデックスの index-name と一致するインデックス名のすべてのインデックスを検索します。InterSystems IRIS がどのクラスでも一致するインデックスを検出できなかった場合、そのようなインデックスが存在しないことを示す SQLCODE -333 エラーが発生します。一致するインデックスを InterSystems IRIS が複数検出した場合、削除するインデックスを DROP INDEX が決定できないため、SQLCODE -334 エラー (“インデックス名が曖昧です。複数のテーブルでインデックスが見つかりました。”) が発生します。InterSystems IRIS のインデックス名は、ネームスペースごとに一意ではありません。

## 存在しないインデックス

既定では、存在しないインデックスを削除しようとすると、DROP INDEX から SQLCODE -333 エラーが発行されます。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[DDL DROP] 設定が表示されます。既定値は 0 (“いいえ”) です。これが推奨設定です。1 (“はい”) に設定した場合、存在しないインデックスに対する DROP INDEX は処理を実行せず、エラー・メッセージも発行しません。詳細は、[“SQL およびオブジェクトの設定ページ”](#) を参照してください。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [\[冗長な DDL ステートメントを無視\]](#) チェック・ボックスにチェックを付けることにより、このオプション (および他の同様の作成、変更、および削除のオプション) をシステム全体で設定できます。

述語 IF EXISTS の動作は、[管理ポータルでの設定および DDL 文を制御する構成パラメータ・ファイル \(CPF\)](#) での設定よりも優先されます。これらの設定によって SQLCODE 0 が返され、通知なしでエラーが抑制されます。IF EXISTS を指定していると、このコマンドからはメッセージと共に SQLCODE 1 が返されます。

## ジャーナリング

%NOJOURN キーワードを指定すると、DROP INDEX によって [ジャーナリング](#) が抑制され、このオペレーション中はトランザクションが無効になります。%NOJOURN を指定するには、%NOJOURN SQL 管理特権が必要です ([GRANT](#) コマンドで設定できます)。

## テーブル名

オプションの table-name を指定する場合は、既存のテーブルに対応している必要があります。

- 指定した table-name が存在しない場合、InterSystems IRIS は SQLCODE -30 エラーを発行し、%msg が “SQLUser.tname ” に設定されます。



- 指定した table-name は存在するけれども index-name という名前のインデックスがない場合、InterSystems IRIS は SQLCODE -333 エラーを発行し、%msg が " SQLUSER.TNAME DROP INDEX MyIndex " に設定されます。
- 指定した table-name がビューの場合、InterSystems IRIS は SQLCODE -333 エラーを発行し、%msg が " SQLUSER.VNAME DROP INDEX EmpSalaryIndex " に設定されます。

## 引数

### IF EXISTS

存在しないインデックスに対してコマンドが実行される場合にエラーを抑制する引数 (オプション)。詳細は、[存在しないインデックスに関する以下のセクション](#)を参照してください。

#### index-name

削除するインデックスの名前。index-name は、アンダースコアやその他の句読点を含む、SQL バージョンの名前です。これは、テーブルの[管理ポータル](#)の [SQL の \[カタログの詳細\]](#) に [SQL マップ名] としてリストされます。

### ON table-name、ON TABLE table-name

インデックスが関連付けられているテーブルの名前を指定する引数 (オプション)。table-name は前述のいずれかの構文で指定できます。最初の構文は ON 節を使用しています。TABLE キーワードはオプションです。2 番目の構文は修飾付きの名前構文 (schema-name.table-name.index-name) を使用しています。table-name は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。table-name 全体を省略した場合、InterSystems IRIS は、以下に説明するように index-name に最初に一致したインデックスを削除します。

## 例

最初の例では、Employee という名前のテーブルを作成しています。これは、このセクションのすべての例で使用されます。

以下の例は、"EmpSalaryIndex" という名前のインデックスを作成し、後でそれを削除します。この例の DROP INDEX では、インデックスに関連付けられているテーブルが指定されておらず、このネームスペース内で "EmpSalaryIndex" が一意のインデックス名であると想定されています。

#### SQL

```
CREATE TABLE Employee (
    EMPNUM      INT NOT NULL,
    NAMELAST    CHAR(30) NOT NULL,
    NAMEFIRST   CHAR(30) NOT NULL,
    STARTDATE   TIMESTAMP,
    SALARY      MONEY,
    ACCRUEDVACATION INT,
    ACCRUEDSICKLEAVE INT,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM))
CREATE INDEX EmpSalaryIndex
ON TABLE Employee
(NameLast,Salary)
DROP INDEX EmpSalaryIndex
```

以下の例では、ON TABLE 節を使用して、削除するインデックスに関連付けられているテーブルを指定しています。

#### SQL

```
CREATE INDEX EmpVacaIndex
ON TABLE Employee
(NameLast,AccruedVacation)
DROP INDEX EmpVacaIndex ON TABLE Employee
```

以下の例では、修飾付きの名前構文を使用して、削除するインデックスに関連付けられているテーブルを指定しています。

## SQL

```
CREATE INDEX EmpSickIndex  
  ON TABLE Employee  
    (NameLast,AccruedSickLeave)  
DROP INDEX Employee.EmpSickIndex
```

以下のコマンドは、存在しないインデックスを削除しようとしています。そのため、SQLCODE -333 エラーが生成されます。

## SQL

```
DROP INDEX PeopleIndex ON TABLE Employee
```

## 関連項目

- ・ [CREATE INDEX](#)
- ・ [インデックスの定義と構築](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP METHOD (SQL)

メソッドを削除します。

### 構文

```
DROP METHOD [IF EXISTS] name [ FROM className ]
```

### 概要

DROP METHOD コマンドは、メソッドを削除します。メソッドを削除すると、InterSystems IRIS は、そのメソッドが許可されているすべてのユーザとロールからそのメソッドを削除して、データベースから削除します。

メソッドを削除するためには、[GRANT](#) コマンドで指定される %DROP\_METHOD 管理者特権が必要です。定義された所有者を持つクラスのメソッドを削除しようとする場合、クラスの所有者としてログインする必要があります。この特権がないと、SQLCODE -99 エラー（特権違反）が生成されます。

メソッドは、そのメソッド定義を含むクラス定義が[導入済みのクラス](#)の場合、削除することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ `classname` DDL ” に設定されます。

name と FROM className の次のような組み合わせがサポートされています。FROM 節では、SQL 名ではなく、クラス・パッケージ名とメソッド名を指定することに注意してください。これらの例では、[システム全体の既定のスキーマ名](#)は、User クラス・パッケージに対応する SQLUser です。

- `DROP METHOD BonusCalc FROM methBonusCalc` : メソッド SQLUser.BonusCalc() を削除します。
- `DROP METHOD BonusCalc FROM User.methBonusCalc` : メソッド SQLUser.BonusCalc() を削除します。
- `DROP METHOD Test.BonusCalc FROM methBonusCalc` : メソッド SQLUser.BonusCalc() を削除します。
- `DROP METHOD BonusCalc FROM Employees.methBonusCalc` : メソッド Employees.BonusCalc() を削除します。
- `DROP METHOD Test.BonusCalc FROM Employees.methBonusCalc` : メソッド Employees.BonusCalc() を削除します。

指定されたメソッドが存在しない場合、DROP METHOD は SQLCODE -362 エラーを発行します。指定された className が存在しない場合、DROP METHOD は SQLCODE -360 エラーを発行します。指定されたメソッドが複数のメソッドを参照する可能性がある場合、DROP METHOD は SQLCODE -361 エラーを生成します。このあいまいさを解消するには、className を指定する必要があります。

メソッドが PROCEDURE 特性キーワードを使用して定義されている場合は、`$SYSTEM.SQL.Schema.ProcedureExists()` メソッドを呼び出すことで、そのメソッドが現在のネームスペース内に存在しているかどうかを確認できます。PROCEDURE キーワードを使用して定義されたメソッドは、DROP METHOD または DROP PROCEDURE によって削除できます。

また、クラス定義からメソッドを削除してクラスをリコンパイルするか、クラス全体を削除することによって、メソッドを削除することもできます。

### 引数

#### IF EXISTS

存在しないメソッドに対してコマンドが実行される場合にエラーを抑制する引数（オプション）。

## name

削除するメソッドの名前。[識別子](#)の名前です。メソッドのパラメータの括弧は指定しないでください。name は修飾 (schema.name)、未修飾 (name) のどちらでもかまいません。FROM className 節が指定されていない限り、メソッド名が未修飾の場合は[既定のスキーマ名](#)が使用されます。

## FROM className

指定があれば、FROM className 節は与えられたクラスからメソッドを削除します。メソッドの SQL 名 (BonusCalc) ではなく、className (methBonusCalc) を指定する必要があります。この節が指定されていない場合、InterSystems IRIS はそのスキーマのすべてのクラスで該当するメソッドを検索し、削除します。しかし、この名前でのメソッドが見つからない場合、または複数の同名のメソッドが見つかった場合、エラー・コードが返されます。メソッドの削除の結果、クラスが空になる場合、DROP METHOD はそのクラスも削除します。

## 例

以下の例は、クラス User.Employee からの mymeth の削除を試行します。(クラス User.Employee の作成例は、“CREATE TABLE” を参照してください)。

## SQL

```
DROP METHOD mymeth FROM User.Employee
```

## 関連項目

- ・ [CREATE METHOD](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP ML CONFIGURATION (SQL)

ML 構成を削除します。

### 構文

```
DROP ML CONFIGURATION ml-configuration-name
```

### 引数

<i>ml-configuration-name</i>	削除する ML 構成の名前。
------------------------------	----------------

### 説明

DROP ML CONFIGURATION コマンドは、ML 構成とそれに対応するクラス定義を削除します。

### 条件

- ML 構成は、現在のネームスペースに存在している必要があります。存在しない ML 構成を削除しようとすると、SQLCODE -30 エラーが生成されます。
- システムの既定の ML 構成を削除することはできません。これを行おうとすると、SQLCODE -189 エラーが返されます。

### 必要なセキュリティ特権

DROP ML CONFIGURATION を呼び出すには、%DROP\_ML\_CONFIGURATION 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%DROP\_ML\_CONFIGURATION 特権を割り当てるには、[GRANT](#) コマンドを使用します。

### 関連項目

- [ALTER ML CONFIGURATION](#)、[CREATE ML CONFIGURATION](#)

# DROP MODEL (SQL)

モデルを削除します。

## 構文

```
DROP MODEL model-name
```

## 引数

<code>model-name</code>	削除するモデルの名前。
-------------------------	-------------

## 説明

DROP MODEL コマンドは、モデルとそれに対応するクラス定義を削除します。モデルに関連するトレーニング実行と検証実行も削除します。

## 存在しないモデルの削除

モデルは、現在のネームスペースに存在している必要があります。存在しないモデルを削除しようとすると、SQLCODE -30 エラーが生成されます。

## 必要なセキュリティ特権

DROP MODEL を呼び出すには、%MANAGE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%MANAGE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## 関連項目

- ・ [ALTER MODEL](#)、[CREATE MODEL](#)

# DROP PROCEDURE (SQL)

プロシージャを削除します。

## 構文

```
DROP PROCEDURE [IF EXISTS] procname [ FROM className ]
DROP PROC procname [ FROM className ]
```

## 概要

DROP PROCEDURE コマンドは、現在のネームスペース内のプロシージャを削除します。プロシージャを削除すると、InterSystems IRIS は、そのプロシージャ権限が与えられている全ユーザおよびロールでそれを無効にし、データベースから削除します。

プロシージャの削除には、[GRANT](#) コマンドで指定された %DROP\_PROCEDURE 管理者特権が必要です。定義された所有者を持つ既存のクラスのプロシージャを削除しようとする場合、クラスの所有者としてログインする必要があります。この特権がないと、SQLCODE -99 エラー（特権違反）が生成されます。

プロシージャは、そのプロシージャ定義を含むクラス定義が[導入済みのクラス](#)の場合、削除することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ `classname` DDL ” に設定されます。

procname では、大文字と小文字は区別されません。procname は、パラメータの括弧なしで指定する必要があります。パラメータの括弧を指定した場合は、SQLCODE -25 エラーが発生します。

procname と FROM className の次のような組み合わせがサポートされています。FROM 節では、SQL 名ではなく、クラス・パッケージ名とプロシージャ名を指定することに注意してください。これらの例では、[システム全体の既定のスキーマ名](#)は、User クラス・パッケージに対応する SQLUser です。

- ・ DROP PROCEDURE BonusCalc FROM procBonusCalc : プロシージャ SQLUser.BonusCalc() を削除します。
- ・ DROP PROCEDURE BonusCalc FROM User.procBonusCalc : プロシージャ SQLUser.BonusCalc() を削除します。
- ・ DROP PROCEDURE Test.BonusCalc FROM procBonusCalc : プロシージャ SQLUser.BonusCalc() を削除します。
- ・ DROP PROCEDURE BonusCalc FROM Employees.procBonusCalc : プロシージャ Employees.BonusCalc() を削除します。
- ・ DROP PROCEDURE Test.BonusCalc FROM Employees.procBonusCalc : プロシージャ Employees.BonusCalc() を削除します。

指定されたプロシージャが存在しない場合、DROP PROCEDURE は SQLCODE -362 エラーを発行します。指定されたクラスが存在しない場合、DROP PROCEDURE は SQLCODE -360 エラーを発行します。指定されたプロシージャが複数のプロシージャを参照する可能性がある場合、DROP PROCEDURE は SQLCODE -361 エラーを生成します。このあいまいさを解消するには、className を指定する必要があります。

指定の procname が現在のネームスペースに存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.ProcedureExists() メソッドを使用します。このメソッドは、PROCEDURE キーワードを使用して定義されたプロシージャとメソッドの両方を認識します。PROCEDURE キーワードを使用して定義されたメソッドは、DROP PROCEDURE を使用して削除できます。

ObjectScript クラス・クエリ・プロシージャであるプロシージャに対して DROP PROCEDURE を実行した場合は、そのプロシージャに関連するメソッド (myprocExecute()、myprocGetInfo()、myprocFetch()、myprocFetchRows()、myprocClose() など) も削除されます。



また、クラス定義からストアド・プロシージャを削除してクラスをリコンパイルするか、クラス全体を削除することによって、プロシージャを削除することもできます。

## 引数

### procname

削除するプロシージャの名前。[識別子](#)の名前です。プロシージャのパラメータの括弧は指定しないでください。name は修飾 (schema.name)、未修飾 (name) のどちらでもかまいません。FROM className 節が指定されていない限り、プロシージャ名が未修飾の場合は[既定のスキーマ名](#)が使用されます。

### FROM className

指定があれば、FROM className 節は与えられたクラスからプロシージャを削除します。この節が指定されていない場合、InterSystems IRIS はそのスキーマのすべてのクラスで該当するプロシージャを検索し、削除します。しかし、この名前でのプロシージャが見つからない場合、または複数の同名のプロシージャが見つかった場合、エラー・コードが返されます。プロシージャの削除の結果、クラスが空になる場合、DROP PROCEDURE はそのクラスも削除します。

## 例

以下の例は、クラス User.Employee からの myprocSP の削除を試行します。(クラス User.Employee の作成例は、“CREATE TABLE” を参照してください)。

### SQL

```
DROP PROCEDURE myprocSP FROM User.Employee
```

## 関連項目

- ・ [CREATE PROCEDURE](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP QUERY (SQL)

クエリを削除します。

### 構文

```
DROP QUERY [IF EXISTS] name [ FROM className ]
```

### 概要

DROP QUERY コマンドはクエリを削除します。クエリを削除すると、InterSystems IRIS は、そのクエリ権限が与えられている全ユーザおよびロールでそれを無効にし、データベースから削除します。

クエリの削除には、[GRANT](#) コマンドで指定された %CREATE\_QUERY 管理者特権が必要です。定義された所有者を持つクラスのクエリを削除しようとする場合、クラスの所有者としてログインする必要があります。この特権がないと、SQLCODE -99 エラー（特権違反）が生成されます。

クエリは、そのクエリ定義を含むクラス定義が[導入済みのクラス](#)の場合、削除することはできません。この操作は SQLCODE -400 エラーで失敗し、%msg が “ `classname` DDL ” に設定されます。

name と FROM className の次のような組み合わせがサポートされています。FROM 節では、SQL 名ではなく、クラス・パッケージ名とクエリ名を指定することに注意してください。これらの例では、[システム全体の既定のスキーマ名](#)は、User クラス・パッケージに対応する SQLUser です。

- `DROP QUERY BonusCalc FROM queryBonusCalc` : クエリ SQLUser.BonusCalc() を削除します。
- `DROP QUERY BonusCalc FROM User.queryBonusCalc` : クエリ SQLUser.BonusCalc() を削除します。
- `DROP QUERY Test.BonusCalc FROM queryBonusCalc` : クエリ SQLUser.BonusCalc() を削除します。
- `DROP QUERY BonusCalc FROM Employees.queryBonusCalc` : クエリ Employees.BonusCalc() を削除します。
- `DROP QUERY Test.BonusCalc FROM Employees.queryBonusCalc` : クエリ Employees.BonusCalc() を削除します。

指定されたクエリが存在しない場合、DROP QUERY は SQLCODE -362 エラーを発行します。指定されたクラスが存在しない場合、DROP QUERY は SQLCODE -360 エラーを発行します。指定されたクエリが複数のクエリを参照する可能性がある場合、DROP QUERY は SQLCODE -361 エラーを生成します。このあいまいさを解消するには、className を指定する必要があります。

また、クラス定義からクエリ（ストアドプロシージャとして投影）を削除してクラスをリコンパイルするか、クラス全体を削除することによって、クエリを削除することもできます。

### 引数

#### IF EXISTS

存在しないクエリに対してコマンドが実行される場合にエラーを抑制する引数（オプション）。

#### name

削除するクエリの名前。[識別子](#)の名前です。クエリのパラメータの括弧は指定しないでください。name は修飾（schema.name）、未修飾（name）のどちらでもかまいません。FROM className 節が指定されていない限り、クエリ名が未修飾の場合は[システム全体の既定のスキーマ名](#)が使用されます。

## FROM className

指定があれば、FROM className 節は与えられたクラスからクエリを削除します。この節が指定されていない場合、InterSystems IRIS はそのスキーマのすべてのクラスで該当するクエリを検索し、削除します。しかし、この名前でのクエリが見つからない場合、または同名のクエリが複数見つかった場合、エラー・コードが返されます。クエリの削除の結果、クラスが空になる場合、DROP QUERY はそのクラスも削除します。

## 例

以下の例は、クラス User.Employee からの myq の削除を試行します。(クラス User.Employee の作成例は、“CREATE TABLE” を参照してください)。

## SQL

```
DROP QUERY myq FROM User.Employee
```

## 関連項目

- ・ [CREATE QUERY](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP ROLE (SQL)

ロールを削除します。

### 構文

```
DROP ROLE [IF EXISTS] role-name
```

### 概要

DROP ROLE 文は、ロールを削除します。ロールを削除すると、InterSystems IRIS は、そのロール権限が与えられている全ユーザおよびロールでそれを無効にし、データベースから削除します。

\$SYSTEM.SQL.Security.RoleExists() メソッドを呼び出すことによって、ロールが存在するかどうかを判別できます。存在しない（または既に削除されている）ロールを削除しようとすると、DROP ROLE は SQLCODE -118 エラーを発行します。

### 特権

DROP ROLE コマンドは特権を必要とする操作です。埋め込み SQL 内で DROP ROLE を使用する前に、以下の要件を 1 つ以上満たす必要があります。

- ・ ロールの所有者である。
- ・ 以下のいずれかでログインしている。
  - USE 権限がある [%Admin\\_Secure](#) 管理リソース
  - USE 権限がある [%Admin\\_RoleEdit](#) 管理リソース
  - システムに対する全面的なセキュリティ特権
- ・ ロール WITH ADMIN OPTION を付与されている。

特権がない場合は、SQLCODE -99 エラー（特権違反）が返されます。

\$SYSTEM.Security.Login() メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

#### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“%SYSTEM.Security” を参照してください。

### 引数

#### IF EXISTS

存在しないロールに対してコマンドが実行される場合にエラーを抑制する引数（オプション）。

#### role-name

削除するロールの名前。[識別子](#)の名前です。ロール名は、大文字と小文字が区別されません。

### 例

以下の例は、BkUser という名前のロールを作成し、その後それを削除しています。

## SQL

```
CREATE ROLE BkName  
DROP ROLE BkName
```

## 関連項目

- ・ SQL 文 : [CREATE ROLE](#)、[CREATE USER](#)、[DROP USER](#)、[GRANT](#)、[REVOKE](#)、[%CHECKPRIV](#)
- ・ [SQL のユーザ、ロール、および特権](#)
- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数

## DROP SCHEMA (SQL)

スキーマ定義を削除します。

### 構文

```
DROP SCHEMA [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

### 引数

引数	説明
name	削除するスキーマの名前。 <a href="#">識別子</a> の名前です。
IF EXISTS	オプション - name を持つスキーマが存在しない場合に発生するエラーを抑制します。
CASCADE	オプション - テーブル、ビュー、ストアド・プロシージャとして投影されるクエリとメソッド、およびユーザ定義の集約など、スキーマを持つすべてのオブジェクトを削除することを指定します。
RESTRICT	オプション - スキーマ内に何も定義されていない場合のみ、スキーマを削除することを指定します。このオプションが想定されるのは、CASCADE が指定されていない場合です。

### 説明

このコマンドは、スキーマ定義を削除します。このコマンドを発行するユーザは、操作を実行するためには、スキーマを所有しているか、%SQLSchemaAdmin [リソース](#)を持っている必要があります。

CASCADE が指定されている場合、スキーマ内のテーブル、ビュー、ストアド・プロシージャとして投影されるクエリとメソッド、およびユーザ定義の集約はすべて削除されます。

既定で RESTRICT オプションが指定されていますが、手動で指定することもできます。指定されている場合、スキーマはスキーマ内に何も定義されていない場合のみ削除されます。CASCADE なしで DROP SCHEMA が指定されていて、スキーマが空でない場合は、SQLCODE -475 が返されます。

操作の実行中、DROP SCHEMA は暗黙的な %NOJOURN を提供してジャーナリングを抑制し、トランザクションを無効にします。また、CASCADE が指定されている場合には、暗黙的な [%DELDATA](#) を提供し、削除するテーブルに関連付けられているデータを削除します。

存在しないスキーマで DROP SCHEMA を実行すると、SQLCODE -473 が返されます。

### 関連項目

- ・ CREATE SCHEMA
- ・ “InterSystems IRIS エラー・リファレンス” にリストされた [SQLCODE エラー・メッセージ](#)

## DROP TABLE (SQL)

テーブルおよび (オプションで) データを削除します。

## 構文

DROP TABLE *table* [RESTRICT | CASCADE] [%DELDATA | %NODELDATA]

## 概要

DROP TABLE コマンドは、テーブルとそれに対応する永続クラス定義を削除します。テーブルがそのスキーマの最後の項目である場合、テーブルを削除すると、スキーマとそれに対応する永続クラス・パッケージも削除されます。

既定では、DROP TABLE は、テーブル定義と**テーブルのデータ** (ある場合) の両方を削除します。%NODELDATA キーワードを使用すると、テーブル定義を削除し、テーブルのデータを削除しないことを指定できます。

DROP TABLE は、テーブルに関連付けられているすべてのインデックスとトリガを削除します。

テーブルを削除するには、以下の条件を満たす必要があります。

- ・ テーブルは、現在のネームスペースに存在している必要があります。存在しないテーブルを削除しようとする、SQLCODE -30 エラーが生成されます。
- ・ テーブル定義は変更可能でなくてはなりません。テーブルを投影するクラスが [DdlAllowed] なしで定義されている場合は、テーブルを削除しようすると SQLCODE -300 エラーが生成されます。
- ・ テーブルが別の同時プロセスでロックされていてはなりません。テーブルがロックされている場合は、DROP TABLE はロックから解放されるまで無制限に待機します。ロックの競合の可能性がある場合は、テーブルに対して IN EXCLUSIVE MODE で **LOCK** を実行してから DROP TABLE を発行することが重要です。
- ・ テーブルに関連付けられているビューがあつてはなりません。また、DROP TABLE に CASCADE キーワードを指定する必要があります。CASCADE を指定せずにテーブルと関連付けられたビューを削除しようすると、SQLCODE -321 エラーが生成されます。
- ・ テーブルを削除するには、必要な特権を持っている必要があります。特権を持たずにテーブルを削除しようすると、SQLCODE -99 エラーが生成されます。
- ・ 対応するクラスが **導入済みのクラス** として定義されていても、テーブルを削除できます。
- ・ テーブルを投影する永続クラスに 派生クラス (**サブクラス**) がある場合、テーブルは削除できません。スーパークラスを削除しようとする (サブクラスが孤立したままになる) と、SQLCODE -300 エラーが生成され、“  
'MySuperClass'  
DDL  
” というメッセージが返されます。

\$SYSTEM.SQL.Schema.DropTable() メソッドを使用して、現在のネームスペース内の 1 つのテーブルを削除できます。SQL テーブル名を指定します。DROP TABLE とは異なり、このメソッドは、[DdlAllowed] なしで定義されたテーブルを削除できます。2 番目の引数は、テーブル・データを削除するかどうかを指定します。既定では、データは削除されません。

## ObjectScript

```
DO $$SYSTEM.SQL.Schema.DropTable("Sample.MyTable",1,.SQLCODE,.%msg)
IF SQLCODE != 0 {WRITE "SQLCODE ",SQLCODE," error: ",%msg}
```

\$SYSTEM.OBJ.Delete() メソッドを使用して、現在のネームスペース内の 1 つ以上のテーブルを削除できます。テーブルを投影する永続クラス名 (SQL テーブル名ではなく) を指定する必要があります。ワイルドカードを使用して複数のクラス名を指定できます。2 番目の引数は、テーブル・データを削除するかどうかを指定します。既定では、データは削除されません。



## 特権

DROP TABLE コマンドは特権を必要とする操作です。DROP TABLE を実行するには、ユーザは %DROP\_TABLE 管理特権を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ %DROP\_TABLE ” に設定されます。適切な付与特権を持っている場合は、GRANT コマンドを使用して %DROP\_TABLE 特権を割り当てることができます。

DROP TABLE 操作でテーブルとテーブル・データの両方を削除する場合でも、ユーザは指定のテーブルに対する DELETE オブジェクト特権を持っている必要はありません。

埋め込み SQL では、以下のように \$SYSTEM.Security.Login() メソッドを使用して適切な特権を持ったユーザとしてログインできます。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、%Service\_Login:Use 特権が必要です。詳細は、“インターシステムズ・クラス・リファレンス” の “%SYSTEM.Security” を参照してください。

DROP TABLE は、テーブル・クラスの定義に [DdlAllowed] が含まれている場合を除き、永続クラスを定義して作成したテーブルでは使用できません。使用すると、操作は SQLCODE -300 エラーで失敗し、%msg が “DDL schema.tablename ” に設定されます。

## 既存のオブジェクト特権

テーブルを削除しても、そのテーブルに対するオブジェクト特権は削除されません。例えば、そのテーブルでデータを挿入、更新、または削除するためにユーザに与えられた特権です。これには、以下のような 2 つの影響があります。

- ・ テーブルが削除され、同じ名前を持つ別のテーブルが作成されると、ユーザとロールは新しいテーブルでも古いテーブルで持っていたのと同じ特権を持ちます。
- ・ テーブルを削除しても、そのテーブルのオブジェクト特権を削除することはできません。

この理由により、通常はテーブルを削除する前に、REVOKE コマンドを使用して、テーブルからオブジェクト特権を削除することをお勧めします。

## データを含むテーブル

既定では、DROP TABLE はテーブル定義とテーブルのデータを削除します。このテーブル・データの削除はアトミック処理です。DROP TABLE が削除できないデータ（参照制約のある行など）を検出した場合、既に実行されたデータの削除が自動的にロールバックされ、テーブル・データは何も削除されません。

\$SYSTEM.SQL.Util.SetOption() メソッドの DDLDropTabDelData オプションを使用して、テーブル・データの削除に関するシステム全体の既定値を設定できます。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[DDL DROP TABLE ] の設定が表示されます。

既定は 1 (“はい”) です。この設定を推奨します。テーブル定義の削除時に、DROP TABLE でテーブルのデータを削除しない場合は、このオプションを 0 (“いいえ”) に設定します。

データの削除に関する設定は、テーブルごとにオーバーライドできます。テーブルの削除時に、DROP TABLE に %NODELDDATA オプションを指定すると、テーブル・データが自動的に削除されるのを防止できます。システム全体の既定値がテーブル・データを削除しないように設定されている場合、DROP TABLE に %DELDDATA オプションを指定すると、テーブルごとにデータを削除できます。

ほとんどの場合、DROP TABLE は、非常に効率的な kill extent 操作を使用して、テーブルのデータを自動的に削除します。テーブルにそのテーブルを参照する外部キーがある、テーブルを投影するクラスが永続クラスのサブクラスである、クラスが既定のストレージを使用しない、ForEach = “row/object” トリガがある、既定でないストリーム・フィールドのグロー

[バル位置](#)を参照するストリーム・フィールドがあるなどの場合、kill extent は使用できません。いずれかが当てはまる場合、DROP TABLE はあまり効率的ではないレコード削除操作を使用して、テーブルのデータを削除します。

TRUNCATE TABLE コマンドを使用すると、テーブル定義を削除せずに、テーブルのデータを削除できます。

## 適用されるロック

DROP TABLE 文は、table に対してテーブル・レベルの排他ロックを取得します。これにより、他のプロセスはテーブル削除の実行中にこのテーブルの定義やデータを変更できなくなります。テーブルの定義およびデータの削除に対してはこのテーブル・レベル・ロックで十分なので、DROP TABLE によってテーブル・データの各行のロックが取得されることはありません。このロックは、DROP TABLE 操作が終了すると自動的に解除されます。

## 外部キー制約

既定では、削除対象のテーブルを参照する別テーブルに外部キー制約が定義されている場合、そのテーブルは削除できません。参照されているテーブルを削除する前に、それを参照しているすべての外部キー制約を削除する必要があります。これらの外部キー制約を削除せずに DROP TABLE 操作を実行すると、SQLCODE -320 エラーが返されます。

外部キー制約での既定の動作は、RESTRICT キーワード・オプションと同じです。この制約では、CASCADE キーワード・オプションはサポートされません。

この既定の外部キー制約の動作を変更するには、“[SET OPTION](#)” コマンドの “COMPILEMODE=NOCHECK” オプションを参照してください。

## 関連付けられたクエリ

テーブルを削除すると、自動的にすべての関連クエリ・キャッシュが削除され、%SYS.PTools.StatsSQL によって生成されたクエリ情報が削除されます。テーブルを削除すると、関連する全クエリのすべての [SQL 実行時統計 \(SQL Stats\)](#) 情報が自動的に削除されます。

## 存在しないテーブル

指定のテーブルが現在のネームスペースに存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.TableExists() メソッドを使用します。

既定では、存在しないテーブルを削除しようとする、DROP TABLE は SQLCODE -30 エラーを発行します。これが推奨設定です。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[

DDL DROP ] 設定が表示されます。既定値は 0 (“いいえ”) です。このオプションを 1 (“はい”)

に設定した場合、存在しないテーブルに対する DROP TABLE は処理を実行せず、エラー・メッセージも発行しません。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション（および他の同様の作成、変更、および削除のオプション）をシステム全体で設定できます。

## 引数

### table

削除する[テーブル](#)の名前。テーブル名は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。スキーマ検索パスの値は使用されません。

## RESTRICT、CASCADE

オプションの引数です。RESTRICT を指定すると、従属ビューや整合性制約のないテーブルのみが削除されます。キーワードが指定されていない場合は、RESTRICT が既定になります。CASCADE により、従属ビューや整合性制約のあるテーブルを削除でき、テーブル削除処理の一環として、参照しているビューまたは整合性制約もすべて削除されます。[外部キー制約](#)では、CASCADE キーワード・オプションはサポートされません。

## %DELDATA、%NODELDATA

これらのキーワードは、テーブルの削除時に、テーブルに関連付けられているデータを削除するかどうかを指定します (オプション)。既定では、テーブル・データを削除します。

## 例

以下の例では、SQLUser.MyEmployees という名前のテーブルを作成し、後でそれを削除しています。この例では、テーブルの削除時に、このテーブルに関連付けられているデータを削除しないことを指定しています。

## SQL

```
CREATE TABLE SQLUser.MyEmployees (  
  NAMELAST      CHAR (30) NOT NULL,  
  NAMEFIRST     CHAR (30) NOT NULL,  
  STARTDATE     TIMESTAMP,  
  SALARY        MONEY)  
  
DROP TABLE SQLUser.MyEmployees %NODELDATA
```

## 関連項目

- ・ [ALTER TABLE、CREATE TABLE、TRUNCATE TABLE](#)
- ・ [テーブルの定義](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

# DROP TRIGGER (SQL)

トリガを削除します。

## 構文

```
DROP TRIGGER [ IF EXISTS] name [ FROM table ]
```

## 概要

DROP TRIGGER コマンドはトリガを削除します。既存のトリガを変更する場合は、DROP TRIGGER を呼び出してトリガの古いバージョンを削除してから、CREATE TRIGGER を呼び出す必要があります。

注釈 DROP TABLE を実行すると、そのテーブルに関連付けられているトリガはすべて削除されます。

## 特権とロック

DROP TRIGGER コマンドは特権を必要とする操作です。DROP TRIGGER を実行するには、ユーザは %DROP\_TRIGGER 管理特権を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %DROP\_TRIGGER " に設定されます。

ユーザは、指定されたテーブルに対する %ALTER 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対する %ALTER 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する %ALTER 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' 'Schema.TableName' %ALTER " に設定されます。

適切な付与特権を持っている場合は、GRANT コマンドを使用して %DROP\_TRIGGER 特権および %ALTER 特権を割り当てることができます。

埋め込み SQL では、以下のように \$SYSTEM.Security.Login() メソッドを使用して適切な特権を持ったユーザとしてログインできます。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、%Service\_Login:Use 特権が必要です。詳細は、"%SYSTEM.Security" を参照してください。

- ・ DROP TRIGGER は、テーブル・クラスの定義に [DdlAllowed] が含まれている場合を除き、永続クラスから投影されたテーブルでは使用できません。使用すると、操作は SQLCODE -300 エラーで失敗し、%msg が "DDL schema.tablename" に設定されます。
- ・ DROP TRIGGER は、導入済みの永続クラスから投影されたテーブルでは使用できません。この操作は SQLCODE -400 エラーで失敗し、%msg が " classname DDL " に設定されます。

DROP TRIGGER 文は、table に対してテーブル・レベルのロックを取得します。これにより、他のプロセスはこのテーブルのデータを変更できなくなります。このロックは、DROP TRIGGER 操作が終了すると自動的に解除されます。

## FROM 節

トリガとそのテーブルは同じスキーマに存在している必要があります。トリガ名が未修飾の場合、トリガのスキーマ名は既定で、FROM 節で指定したように、テーブルのスキーマと同じスキーマになります。トリガ名が未修飾で、FROM 節がないか、テーブル名も未修飾の場合、トリガのスキーマは既定で既定のスキーマ名になります。スキーマ検索パスは使用されません。両方の名前を修飾する場合は、トリガのスキーマ名がテーブルのスキーマ名と同じである必要があります。

スキーマ名が一致しないと SQLCODE -366 エラーになります。これはトリガ名とテーブル名の両方が修飾されていて、異なるスキーマ名を指定しているときにのみ起こります。

InterSystems SQL では、トリガ名はスキーマ内の特定のテーブルに関して一意である必要があります。したがって、1 つのスキーマ内に同じ名前のトリガが複数存在する場合があります。オプションの FROM 節は、削除するトリガを決めるために以下のように使用されます。

- FROM 節が指定されておらず、指定された名前と一致する一意のトリガがスキーマ内で見つかった場合、InterSystems IRIS はそのトリガを削除します。
- FROM 節が指定され、指定された名前と FROM 節のテーブル名の両方に一致する一意のトリガがスキーマ内で見つかった場合、InterSystems IRIS はそのトリガを削除します。
- FROM 節が指定されておらず、指定された名前と一致する複数のトリガが見つかった場合、InterSystems IRIS は SQLCODE -365 エラーを発行します。
- FROM 節で指定されたテーブルに関して指定された名前のトリガが見つからないか、FROM 節が指定されていない場合にスキーマ内でどのテーブルに関しても指定された名前のトリガが見つからない場合、InterSystems IRIS は SQLCODE -363 エラーを発行します。

## 引数

### IF EXISTS

存在しないトリガに対してコマンドが実行される場合にエラーを抑制する引数 (オプション)。

#### name

削除するトリガの名前。トリガ名は修飾、未修飾のどちらでもかまいません。修飾する場合、そのスキーマ名はテーブルのスキーマ名と同じである必要があります。

### FROM テーブル

トリガを削除するテーブルを指定する引数 (オプション)。FROM 節を指定すると、そのテーブルでのみ指定のトリガが検索されます。FROM 節を指定しない場合は、name で指定したスキーマ全体で指定のトリガが検索されます。

## 例

以下の例は、システム全体の既定のスキーマ内で任意のテーブルに関連する Trigger\_1 というトリガを削除します (初期の既定のスキーマは SQLUser です)。

#### SQL

```
DROP TRIGGER Trigger_1
```

以下の例は、A スキーマ内で任意のテーブルに関連する Trigger\_2 というトリガを削除します。

#### SQL

```
DROP TRIGGER A.Trigger_2
```

以下の例は、システム全体の既定のスキーマ内で Patient テーブルに関連する Trigger\_3 というトリガを削除します。Trigger\_3 という名前のトリガが見つかって、Patient と関連していなければ、InterSystems IRIS は SQLCODE -363 エラーを発行します。

#### SQL

```
DROP TRIGGER Trigger_3 FROM Patient
```

以下の例はすべて、Test スキーマ内で Patient テーブルに関連する Trigger\_4 というトリガを削除します。

## SQL

```
DROP TRIGGER Test.Trigger_4 FROM Patient
```

## SQL

```
DROP TRIGGER Trigger_4 FROM Test.Patient
```

## SQL

```
DROP TRIGGER Test.Trigger_4 FROM Test.Patient
```

## 関連項目

- ・ [CREATE TRIGGER](#)
- ・ [GRANT](#)
- ・ [トリガの使用法](#)
- ・ [SQLCODE エラー・メッセージ](#)

## DROP USER (SQL)

ユーザ・アカウントを削除します。

### 構文

```
DROP USER [IF EXISTS] user-name
```

### 概要

DROP USER コマンドは、ユーザ・アカウントを削除します。このユーザ・アカウントは、CREATE USER を使用して作成され、*user-name* を指定されています。指定された *user-name* が既存のユーザ・アカウントに対応しない場合、InterSystems IRIS は SQLCODE -118 エラーを発行します。`$SYSTEM.Security.UserExists()` メソッドを呼び出すことによって、ユーザが存在するかどうかを判別できます。

ユーザ名は、大文字と小文字が区別されません。

管理ポータルを使用してユーザを削除することもできます。[\[システム管理\]](#)、[\[セキュリティ\]](#)、[\[ユーザ\]](#) の順に選択して、既存のユーザをリストします。ユーザ・アカウントのこのテーブルで、削除するユーザ・アカウントの [\[削除\]](#) をクリックします。

### 特権

DROP USER コマンドは特権を必要とする操作です。埋め込み SQL で DROP USER を使用する前に、以下のいずれかを持つユーザとしてログインする必要があります。

- ・ USE 権限がある [%Admin\\_Secure](#) 管理リソース
- ・ USE 権限がある [%Admin\\_UserEdit](#) 管理リソース
- ・ システムに対する全面的なセキュリティ特権

これらの特権がない場合に DROP USER コマンドを実行すると、SQLCODE -99 エラー (特権違反) が返されます。

`$SYSTEM.Security.Login()` メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

#### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

`$SYSTEM.Security.Login` メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、["%SYSTEM.Security"](#) を参照してください。

### 引数

#### *user-name*

存在しないユーザに対してコマンドが実行される場合にエラーを抑制する引数 (オプション)。

### 例

以下の文を発行して、PSMITH を削除できます。

#### SQL

```
DROP USER psmith
```



## 関連項目

- ・ SQL 文 : [CREATE USER](#)、[ALTER USER](#)、[GRANT](#)、[REVOKE](#)、[%CHECKPRIV](#)
- ・ [SQL のユーザ、ロール、および特権](#)
- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数

## DROP VIEW (SQL)

ビューを削除します。

### 構文

```
DROP VIEW [IF EXISTS] view-name [CASCADE | RESTRICT]
```

### 概要

DROP VIEW コマンドはビューを削除しますが、基となるテーブルやデータは削除しません。

DROP VIEW 操作は、DropView() メソッド呼び出しを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Schema.DropView(viewname,SQLCODE,%msg)
```

### 特権

DROP VIEW コマンドは特権を必要とする操作です。DROP VIEW を使用する前に、%DROP\_VIEW 管理者特権または指定されたビューに対する DELETE オブジェクト特権を持っている必要があります。特権がない場合は、SQLCODE -99 エラー (特権違反) が返されます。%CHECKPRIV コマンドを呼び出すことにより、現在のユーザが DELETE 特権を持っているかどうかを確認できます。\$SYSTEM.SQL.Security.CheckPrivilege() メソッドを呼び出すことにより、指定のユーザが DELETE 特権を持っているかどうかを確認できます。適切な付与特権を持っている場合は、GRANT コマンドを使用して %DROP\_VIEW 特権を割り当てることができます。

埋め込み SQL では、以下のように \$SYSTEM.Security.Login() メソッドを使用して適切な特権を持ったユーザとしてログインできます。

#### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、%Service\_Login:Use 特権が必要です。詳細は、“%SYSTEM.Security”を参照してください。

導入済みの永続クラスから投影されたテーブルに基づくビューを削除できます。

### 存在しないビュー

指定のビューが現在のネームスペースに存在するかどうかを確認するには、\$SYSTEM.SQL.Schema.ViewExists() メソッドを使用します。

既定では、存在しないビューを削除しようとする、DROP VIEW は SQLCODE -30 エラーを発行します。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、DDL DROP 1 設定が表示されます。既定値は 0 (“いいえ”) です。この設定を推奨します。1 (“はい”) に設定した場合、存在しないビューおよびテーブルに対して DROP VIEW または DROP TABLE を発行すると、処理は実行されず、エラー・メッセージも発行されません。

管理ポータル、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] から [冗長な DDL ステートメントを無視] チェック・ボックスにチェックを付けることにより、このオプション (および他の同様の作成、変更、および削除のオプション) をシステム全体で設定できます。

述語 IF EXISTS の動作は、管理ポータルでの設定および DDL 文を制御する構成パラメータ・ファイル (CPF) での設定よりも優先されます。これらの設定によって SQLCODE 0 が返され、通知なしでエラーが抑制されます。IF EXISTS を指定していると、このコマンドからはメッセージと共に SQLCODE 1 が返されます。

## 他のビューから参照されるビュー

クエリで他のビューから参照されているビューを削除しようとすると、DROP VIEW は既定で SQLCODE -321 エラーを発行します。これは RESTRICT キーワードの動作です。

CASCADE キーワードを指定すると、クエリで他のビューによって参照されているビューの削除は成功します。さらに、DROP VIEW はそれらの参照しているビューも削除します。InterSystems IRIS で (SQLCODE -300 エラーなどにより) ビューのカスケード削除が一部でも実行できない場合は、いずれのビューも削除されません。

## 関連付けられたクエリ

ビューを削除すると、自動的にすべての関連クエリ・キャッシュが削除され、%SYS.PTools.StatsSQL によって生成されたクエリ情報が削除されます。ビューを削除すると、関連する全クエリに関するすべての [SQL 実行時統計 \(SQL Stats\)](#) 情報が自動的に削除されます。

## 引数

### IF EXISTS

存在しないビューに対してコマンドが実行される場合にエラーを抑制する引数 (オプション)。詳細は、[存在しないテーブルに関する以下のセクション](#)を参照してください。

### view-name

削除するビューの名前。ビュー名は修飾 (schema.viewname)、未修飾 (viewname) のどちらでもかまいません。ビュー名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。

### CASCADE、RESTRICT

オプションの引数です。view-name を参照する他のすべてのビューを削除する場合は CASCADE キーワードを指定します。RESTRICT は、view-name を参照するビューが他に存在するときに SQLCODE -321 エラーを発行する場合に指定します。既定値は RESTRICT です。

## 例

以下の例は、“CityAddressBook” という名前のビューを作成し、その後そのビューを削除します。このビューは RESTRICT キーワード (既定) を使用して指定されるので、ビューが他のビューから参照される場合、SQLCODE -321 エラーが発行されます。

### SQL

```
CREATE VIEW CityAddressBook AS
  SELECT Name, Home_Street FROM Sample.Person
  WHERE Home_City='Boston'
DROP VIEW CityAddressBook RESTRICT)
```

## 関連項目

- [ALTER VIEW、CREATE VIEW、GRANT](#)
- [ビューの定義と使用](#)
- [SQL およびオブジェクトの設定ページ](#)
- [SQLCODE エラー・メッセージ](#)

## EXPLAIN (SQL)

指定されたクエリのクエリ・プランを返します。

### 構文

```
EXPLAIN [ALT | ALL] [STAT | STATS] [INTO :host-variable] query
```

### 説明

EXPLAIN コマンドは、指定されたクエリのクエリ・プランを XML タグが付けられたテキスト文字列として返します。このクエリ・プランは、Plan という名前の単一のフィールドで構成される結果セットとして返されます。

query には SELECT、DELETE、UPDATE のいずれかのクエリを指定する必要があります。INSERT query を指定すると、SQLCODE -474 が返されます。他のキーワードを指定して EXPLAIN を使用すると、SQLCODE -51 が返されます。[\[プラン表示\]](#) を使用して、他のクエリ (SELECT 節を使用した INSERT クエリなど) のクエリ・プランを表示できます。EXPLAIN コマンドによる query 参照が実行されると、すべてのエラーが処理およびスローされます。

ALT キーワードと STAT キーワードは任意の順序で指定できます。INTO キーワードは、これらのキーワードの後に指定する必要があります。オプションの ALT キーワードは、代替クエリ・プランを生成します。代替クエリ・プランはすべて、同じ XML タグが付いたテキスト文字列で返されます。正規化された query テキスト (<sql> としてタグ付け) は、各クエリ・プランの前にリストされます。オプションの STAT キーワードは、クエリ・プランの各モジュールに対して[実行時パフォーマンス統計](#)を生成します。STAT キーワードは、SELECT クエリでのみサポートされています。クエリ・プランを格納した XML タグ テキスト文字列に実行時統計も記述されます。各モジュールについて以下の統計が収集されます。

- ・ <ModuleName> : モジュール名。
- ・ <TimeSpent> : モジュールの総実行時間 (秒単位)。
- ・ <GlobalRefs> : グローバル参照の数。
- ・ <LinesOfCode> : コード実行行数。
- ・ <DiskWait> : ディスク待機時間 (秒単位)。
- ・ <RowCount> : 結果セット内の行数。
- ・ <ModuleCount> : このモジュールが実行された回数。
- ・ <Counter> : このプログラムが実行された回数。

これらの統計は、クエリ・プランのテキスト内で、XML タグが付けられたテキスト文字列として返されます。関連付けられているクエリ・プランの前に、クエリ・プラン内のすべてのモジュールのパフォーマンス統計が返されます。埋め込み SQL では、実行時パフォーマンス統計を生成したり返したりすることはできません。STAT キーワードは無視され、エラーは発行されません。

EXPLAIN コマンドを発行するには、%SYSTEM.QUERY\_PLAN プロシージャを実行する[特権](#)が必要です。

EXPLAIN コマンドは、\$SYSTEM.SQL.Explain() メソッドを呼び出して、その結果セットを XML タグが付けられたテキスト文字列が含まれる単一のフィールドの形式にすることによって、[プラン表示](#)の結果を返します。EXPLAIN ALT コマンドは、修飾子 all=1 を付けて \$SYSTEM.SQL.Explain() メソッドを呼び出して、その結果セットを XML タグが付けられたテキスト文字列が含まれる単一のフィールドの形式にすることによって、[\[別のプランを表示\]](#)の結果を返します。

**注釈** このコマンドは、埋め込み SQL、ダイナミック SQL、SQL シェル、管理ポータル、JDBC、および ODBC インタフェースでの使用が全面的にサポートされます。

## 結果セット XML の構造

以下に、EXPLAIN ALT STAT query の、XML タグが付けられたテキスト文字列の構造を示します。改行、インデント、およびコメントの注は説明のために付けられています。

```
<plans> /* tag included even if there is only one plan */
  <plan> /* the first query plan */
    <sql> /* the normalized SELECT statement text */ </sql>
    <cost value="1147000"/>
    /* if STAT, include the following <stats> tags */
    <stats> <ModuleName>MAIN</ModuleName> /* XML-tagged list of stats (above) for MAIN module */ </stats>
    <stats> <ModuleName>FIRST</ModuleName> /* XML-tagged list of stats (above) for FIRST module */ </stats>
    <stats> /* additional modules */ </stats>
    /* text of query plan */
  </plan>
  <plan> /* if ALT, same info for first alternate plan */
    ...
  </plan>
</plans>
```

## Explain() メソッド

\$SYSTEM.SQL.Explain() メソッドを使用して、ObjectScript から同じクエリ・プラン情報を返すことができます。以下に例を示します。

```
SET myquery=2
SET myquery(1)="SELECT Name, Age FROM Sample.Person WHERE Name %STARTSWITH 'Q' "
SET myquery(2)="ORDER BY Age"
SET status=$SYSTEM.SQL.Explain(.myquery, {"all":0}, .plan)
IF status=1 {WRITE "Explain() failed:" DO $System.Status.DisplayError(status) QUIT}
ZWRITE plan
```

## 引数

### ALT

代替クエリ・プランを返す引数（オプション）。既定では、1 つのクエリ・プランを返します。

### STAT

（ダイナミック SQL のみ）：クエリ・プランの実行時パフォーマンス統計を返す引数（オプション）。既定では、実行時統計なしでクエリ・プランを返します。この構文は、埋め込み SQL では無視されます。

### INTO :host-variable

（埋め込み SQL のみ）：クエリ・プランが配置される出力**ホスト変数**（オプション）。この構文は、ダイナミック SQL では無視されます。

### query

SELECT、UPDATE、または DELETE クエリ。

## 例

以下の例では、クエリ・プランが XML 文字列で返されます。最初に SQL クエリ文字列、次にクエリ・プランが返されます。

```
EXPLAIN SELECT Name, DOB FROM Sample.Person WHERE Name [ 'Q'
```

以下の例では、クエリ・プランとパフォーマンス統計が XML 文字列で返されます。最初に SQL クエリ文字列、次にパフォーマンス統計（モジュール別）、その次にクエリ・プランが返されます。

```
EXPLAIN STAT SELECT Name, DOB FROM Sample.Person WHERE Name [ 'Q'
```

以下の例では、代替クエリ・プランが XML 文字列で返されます。最初に SQL クエリ文字列、次に各クエリ・プランが返されます。

```
EXPLAIN ALT SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'
```

以下の例では、より複雑なクエリ・プランが返されます。クエリ・プランの前と内部の両方にパフォーマンス統計が表示されます。

```
EXPLAIN STAT SELECT p.Name AS Person, e.Name AS Employee
FROM Sample.Person AS p, Sample.Employee AS e
WHERE p.Name %STARTSWITH 'Q' GROUP BY e.Name ORDER BY p.Name
```

以下の埋め込み SQL の例は、クエリ・プランを XML 文字列として返します。最初に SQL クエリ文字列、次にクエリ・プランが返されます。

```
#sqlcompile select=Runtime
&sql(EXPLAIN INTO :qplan SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'])
WRITE qplan
```

以下の埋め込み SQL の例は、代替クエリ・プランを XML 文字列として返します。最初に SQL クエリ文字列、次に 1 つ目のクエリ・プラン、その次に SQL クエリ文字列、続いて 2 つ目のクエリ・プランというように返されます。

```
#sqlcompile select=Runtime
&sql(EXPLAIN ALT INTO :qplans SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'])
WRITE qplans
```

以下の埋め込み SQL の例は、クエリ・プランを返します。STAT キーワードは無視されます。

```
#sqlcompile select=Runtime
&sql(EXPLAIN INTO :qplan SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'])
WRITE qplan
```

## 関連項目

- ・ [SELECT](#)
- ・ [JOIN](#)
- ・ [プランの表示](#)
- ・ [SQL 実行時統計情報](#)
- ・ [データベースの問い合わせ](#)

# FETCH (SQL)

カーソルからデータを取り出します。

## 構文

```
FETCH cursor-name [ INTO host-variable-list ]
```

## 概要

埋め込み SQL アプリケーション内で、FETCH 文はカーソルからデータを取り出します。操作の必須順序は、DECLARE、OPEN、FETCH、CLOSE です。オープンしていないカーソルに FETCH を実行すると、SQLCODE -102 エラーが返されます。

SQL 文として埋め込み SQL からだけサポートされています。同様の操作は、ODBC でも ODBC API を使用してサポートされます。詳細は、“埋め込み SQL の使用法”を参照してください。

INTO 節は、DECLARE 文の節、FETCH 文の節、またはその両方として指定できます。INTO 節は、ローカルホスト変数内に、フェッチの列からデータを配置します。リスト内の各ホスト変数は左から右に、カーソル結果セット内で対応する列に関連しています。各変数のデータ型は、結果セット列に対応するデータ型の暗黙の変換に一致するかサポートされる必要があります。変数の数は、カーソル選択リスト内の列数と一致しなければなりません。

FETCH 操作は、カーソルがデータの最後に達したときに完了します。このとき、SQLCODE=100 (これ以上データがない) に設定されます。また、%ROWCOUNT 変数もフェッチした行数に設定されます。

**注釈** INTO 節のホスト変数から返される値は、SQLCODE=0 の場合のみ信頼できます。SQLCODE=100 (これ以上データがない) の場合は、ホスト変数の値は使用しないでください。

cursor-name はネームスペース固有ではありません。現在のネームスペースを変更しても、宣言されたカーソルの使用には影響はありません。ネームスペースで考慮する必要がある唯一の点は、FETCH は、クエリ対象のテーブルを含むネームスペースで行う必要があるということです。

## %ROWID

FETCH が更新可能なカーソルの行を取得すると、%ROWID がフェッチされた行の RowID 値に設定されます。更新可能なカーソルとは、最上位の FROM 節にテーブル名か更新可能なビュー名のいずれかの要素が 1 つだけ含まれるカーソルのことです。

取得される各行の %ROWID の設定には、以下の条件が適用されます。

- DECLARE cursorname CURSOR 文と OPEN cursorname 文では %ROWID は初期化されないため、%ROWID 値は以前の値から変更されません。FETCH が初めて正常に実行されると、%ROWID が設定されます。行を取得する後続の各 FETCH は、%ROWID を現在の RowID にリセットします。FETCH は、更新可能なカーソルの行を取得すると、%ROWID を設定します。カーソルが更新可能でない場合、%ROWID は変更されません。クエリ選択条件に一致する行がない場合、FETCH は %ROWID 値を変更しません。CLOSE 時、または FETCH が SQLCODE 100 (データがない、またはこれ以上データがない) を発行すると、%ROWID には取得された最後の行の RowID が含まれます。
- カーソル・ベースの SELECT で DISTINCT キーワードまたは GROUP BY 節を使用すると、%ROWID は設定されません。%ROWID 値に以前の値がある場合、%ROWID 値はその値から変更されません。
- 集約演算のみを実行するカーソル・ベースの SELECT では、%ROWID は設定されません。%ROWID 値に以前の値がある場合、%ROWID 値はその値から変更されません。

カーソルが宣言されていない埋め込み SQL SELECT では、%ROWID は設定されません。%ROWID 値は単純な SELECT 文の完了時に未変更のままとなります。



## UPDATE または DELETE の FETCH

FETCH を使用して、更新または削除する行を取得できます。UPDATE または DELETE では、WHERE CURRENT OF 節を指定する必要があります。DECLARE では FOR UPDATE 節を使用する必要があります。以下の例は、選択された行をすべて削除するカーソル・ベースの削除を示しています。

### ObjectScript

```
SET $NAMESPACE="Samples"
&sql(DECLARE MyCursor CURSOR FOR SELECT %ID,Status
      FROM Sample.Quality WHERE Status='Bad' FOR UPDATE)
&sql(OPEN MyCursor)
      IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR {&sql(FETCH MyCursor)  QUIT:SQLCODE'=0
      &sql(DELETE FROM Sample.Quality WHERE CURRENT OF MyCursor) }
WRITE !,"Number of rows updated=",%ROWCOUNT
&sql(CLOSE MyCursor)
      IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

## 引数

### cursor-name

現在オープンしているカーソルの名前。カーソル名は **DECLARE** コマンドで指定されています。カーソル名は、大文字と小文字を区別します。

### INTO host-variable-list

フェッチの列からローカル変数にデータをコピーする変数 (オプション)。host-variable-list には、1 つの**ホスト変数**またはコンマで区切られた複数のホスト変数のリストを指定します。この変数がカーソルに関連付けられたデータの格納先になります。INTO 節は、オプションです。指定されていない場合は、FETCH 文はカーソルのみを配置します。

## 例

以下の埋め込み SQL の例は、EmpCursor という名前のカーソルからデータを取り出す引数なしの FOR ループによって呼び出された FETCH を示しています。INTO 節は DECLARE 文に指定されています。

### ObjectScript

```
&sql(DECLARE EmpCursor CURSOR FOR
      SELECT Name, Home_State
      INTO :name,:state FROM Sample.Employee
      WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
      IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
      QUIT:SQLCODE'=0
      WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
      WRITE " Name=",name," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
      IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

以下の埋め込み SQL の例は、EmpCursor という名前のカーソルからデータを取り出す引数なしの FOR ループによって呼び出された FETCH を示しています。INTO 節は、FETCH 文の一部として指定されています。

## ObjectScript

```

&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name,Home_State FROM Sample.Employee
    WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
FOR { &sql(FETCH EmpCursor INTO :name,:state)
    QUIT:SQLCODE'=0
    WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
    WRITE " Name=",name," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}

```

以下の埋め込み SQL の例は、WHILE ループを使用して呼び出された FETCH を示しています。

## ObjectScript

```

&sql(DECLARE C1 CURSOR FOR
    SELECT Name,Home_State INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'M')
&sql(OPEN C1)
    IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
    WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
    WRITE " Name=",name," State=",state,!
    &sql(FETCH C1) }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE C1)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}

```

以下の埋め込み SQL の例は、集約関数値を取得する FETCH を示しています。%ROWID は設定されていません。

## ObjectScript

```

&sql(DECLARE PersonCursor CURSOR FOR
    SELECT COUNT(*),AVG(Age) FROM Sample.Person )
&sql(OPEN PersonCursor)
    IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT
FOR { &sql(FETCH PersonCursor INTO :cnt,:avg)
    QUIT:SQLCODE'=0
    WRITE %ROWCOUNT," Num People=",cnt," Average Age=",avg,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE PersonCursor)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}

```

以下の埋め込み SQL の例は、DISTINCT 値を取得する FETCH を示しています。%ROWID は設定されていません。

## ObjectScript

```

&sql(DECLARE EmpCursor CURSOR FOR
    SELECT DISTINCT Home_State FROM Sample.Employee
    WHERE Home_State %STARTSWITH 'M'
    ORDER BY Home_State )
&sql(OPEN EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT
FOR { &sql(FETCH EmpCursor INTO :state)
    QUIT:SQLCODE'=0
    WRITE %ROWCOUNT," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}

```

以下の埋め込み SQL の例は、複数ネームスペース間でカーソルが持続することを示しています。このカーソルは %SYS で宣言され、USER でオープンおよびフェッチされ、SAMPLES でクローズされます。OPEN は、クエリされるテーブルを

含むネームスペースで実行する必要があること、および FETCH は出力ホスト変数にアクセスできる必要があり、出力ホスト変数はネームスペース固有であることに注意してください。

```
&sql(USE DATABASE %SYS)
WRITE $ZNSPACE,!
&sql(DECLARE NSCursor CURSOR FOR SELECT Name INTO :name FROM Sample.Employee)
&sql(USE DATABASE "USER")
WRITE $ZNSPACE,!
&sql(OPEN NSCursor)
  IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
  NEW SQLCODE,%ROWCOUNT,%ROWID
FOR { &sql(FETCH NSCursor)
  QUIT:SQLCODE
  WRITE "Name=",name,! }
&sql(USE DATABASE SAMPLES)
WRITE $ZNSPACE,!
&sql(CLOSE NSCursor)
  IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

## 関連項目

- ・ [CLOSE、DECLARE、OPEN](#)
- ・ [SQL カーソル](#)
- ・ [SQLCODE エラー・メッセージ](#)

# FREEZE PLANS (SQL)

1 つ以上のクエリ・プランを凍結します。

## 構文

```
FREEZE PLANS BY ID statement-hash
FREEZE PLANS BY TABLE table-name
FREEZE PLANS BY SCHEMA schema-name
FREEZE PLANS
```

## 説明

FREEZE PLANS コマンドは、クエリ・プランを凍結します。凍結したクエリ・プランを凍結解除するには、[UNFREEZE PLANS](#) コマンドを使用します。

FREEZE PLANS は、プランの状態が `NOT_FROZEN` のクエリ・プランを凍結できます。プランの状態が `FROZEN` / `DELETED` のクエリ・プランは凍結できません。

FREEZE PLANS には、クエリ・プランを凍結するための 4 つの構文形式が用意されています。

- ・ 指定されたクエリ・プラン：FREEZE PLANS BY ID *statement-hash*。*statement-hash* 値は、二重引用符で区切る必要があります。
- ・ テーブルのすべてのクエリ・プラン：FREEZE PLANS BY TABLE *table-name*。テーブル名またはビュー名を指定できます。クエリ・プランが複数のテーブルまたはビューを参照する場合、これらのテーブルまたはビューのいずれかを指定すると、クエリ・プランは凍結されます。
- ・ スキーマ内のすべてのテーブルのすべてのクエリ・プラン：FREEZE PLANS BY SCHEMA *schema-name*。
- ・ 現在のネームスペースのすべてのテーブルのすべてのクエリ・プラン：FREEZE PLANS。

このコマンドは、1 つ以上のクエリ・プランが凍結される場合は SQLCODE 0 を発行し、クエリ・プランが凍結されない場合は SQLCODE 100 を発行します。影響を受けた行 (%ROWCOUNT) は、凍結されたクエリ・プランの数を示します。

## その他のインタフェース

次の `$SYSTEM.SQL.Statement` メソッドを使用して 1 つのクエリ・プランまたは複数のクエリ・プランを凍結できます：1 つのプランの場合は `FreezeStatement()`、関係のすべてのプラン（クエリ・プランで参照されるテーブルまたはビュー）の場合は `FreezeRelation()`、スキーマのすべてのプランの場合は `FreezeSchema()`、現在のネームスペースのすべてのプランの場合は `FreezeAll()`。対応する `Unfreeze` メソッドがあります。

管理ポータルを使用して、クエリ・プランを凍結できます。詳細は、“[凍結プランのインタフェース](#)”を参照してください。

## 引数

### statement-hash

引用符で囲まれた、クエリ・プランの SQL 文の定義の内部ハッシュ表現。場合によっては、同じ SQL 文のように見えても、文のハッシュ・エントリが異なることがあります。SQL 文の異なるコード生成を必要とする設定/オプションの相違によって、異なる文ハッシュが生成されます。これは、異なる内部最適化をサポートする異なるクライアント・バージョンや異なるプラットフォームで発生することがあります。“[SQL 文の詳細](#)”を参照してください。

### table-name

既存のテーブルまたはビューの名前。*table-name* は修飾 (`schema.table`)、未修飾 (`table`) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。

## schema-name

既存のスキーマの名前。このコマンドは、指定されたスキーマ内のすべてのテーブルのすべてのクエリ・プランを凍結します。

## セキュリティおよび特権

FREEZE PLANS コマンドは、ユーザに %Development:USE 権限が必要な特権操作です。このような権限は管理ポータルを介して付与できます。この特権なしで FREEZE PLANS コマンドを実行すると、SQLCODE -99 エラーが発生し、コマンドは失敗します。これには、以下の 2 つの例外があります。

- ・ 埋め込み SQL を介してコマンドを実行する場合。この場合は特権が確認されません。
- ・ 特権を確認しないことをユーザが明示的に指定する場合。例えば、checkPriv 引数を 0 に設定して %Prepare() を呼び出すか、**%SQL.Statement** に対して %ExecDirectNoPriv() を呼び出します。

## 関連項目

- ・ [UNFREEZE PLANS コマンド](#)
- ・ [凍結プラン](#)
- ・ [SQL 文と SQL 統計の分析](#)

# GRANT (SQL)

ユーザまたはロールに特権を与えます。

## 構文

```
GRANT admin-privilege TO grantee [WITH ADMIN OPTION]
GRANT role TO grantee [WITH ADMIN OPTION]

GRANT role TO grantee [WITH ADMIN OPTION]

GRANT object-privilege ON object-list
  TO grantee [WITH GRANT OPTION]

GRANT SELECT ON CUBE[S] object-list
  TO grantee [WITH GRANT OPTION]
GRANT column-privilege (column-list) ON table
  TO grantee [WITH GRANT OPTION]
```

## 概要

GRANT コマンドは、指定したテーブル、ビュー、列、またはその他のエンティティ上で指定タスクを実行する特権を、指定したユーザやロールに与えます。以下の基本操作を実行できます。

- ・ ユーザに特権を与える
- ・ ロールに特権を与える
- ・ ユーザにロールを与える
- ・ ロールにロールを与え、ロールの階層を作成する

ユーザに特権を与える場合は、そのユーザが即座にその特権を行使できます。ロールに特権を与える場合は、そのロールを与えられたユーザが即座にその特権を行使できます。特権を削除すると、そのユーザは即座にその特権を失います。特定の特権は、特定のユーザに一度だけ与えればよいので効率的です。複数のユーザが同じ特権を 1 人のユーザに複数回与えることができますが、1 回の REVOKE でその特権は削除されます。

特権はネームスペースごとに付与されます。

SQL 特権は、ODBC、JDBC、およびダイナミック SQL ([%SQL.Statement](#)) を介してのみ適用されます。

GRANT はすぐに作成および実行され、通常一度しか実行されないため、InterSystems IRIS では、ODBC、JDBC、またはダイナミック SQL での GRANT にはクエリ・キャッシュは作成されません。\* の展開は GRANT コマンドの実行時に行われます。

## GRANT admin-privilege

SQL 管理者 (admin) 特権は、ユーザまたはロールに適用します。特定のオブジェクトに結び付けられていない特権 (つまり、そのユーザやロールの一般特権) は、管理者特権と見なされます。このような特権は、現在のネームスペースに基づいて、ネームスペースごとに付与されます。

%DB\_OBJECT\_DEFINITION 特権は、16 個すべてのデータ定義特権を付与しま

す。%BUILD\_INDEX、%NOCHECK、%NOINDEX、%NOLOCK、および %NOTRIGGER 特権は付与されません。それらは、明示的に付与する必要があります。

%BUILD\_INDEX 特権は [BUILD INDEX](#) コマンドの使用権限を付与します。%NOCHECK、%NOINDEX、%NOLOCK、および %NOTRIGGER 特権により、[INSERT](#)、[UPDATE](#)、[INSERT OR UPDATE](#)、または [DELETE](#) 文の restriction 節でこれらのオプションを使用できるようになります。%NOINDEX キーワードを述語条件の前で使用しても効果はありません。[TRUNCATE TABLE](#) は、%NOTRIGGER 動作によってテーブルからのすべての行の削除を実行するため、TRUNCATE TABLE の実行には %NOTRIGGER 特権が必要となります。INSERT、UPDATE、INSERT OR UPDATE、または DELETE 文を作成する際にその restriction を使用するには、適切な %NOCHECK、%NOINDEX、%NOLOCK、または %NOTRIGGER 特権が必要となります。

指定された管理者特権が (例えばスペルの誤りなどの理由で) 有効な特権名でない場合でも、InterSystems IRIS は正常に完了し、SQLCODE 100 (データの末尾に到達) を発行します。InterSystems IRIS は、指定されたユーザ (またはロール) が存在するかどうかは確認しません。指定された管理者特権は有効だが、指定されたユーザ (または、ロール) が存在しない場合、InterSystems IRIS は SQLCODE -118 エラーを発行します。

## GRANT role

この形式の GRANT は、指定されたロールにユーザを割り当てます。また、別のロールにロールを割り当てることもできます。割り当て先として指定されたロールが存在しない場合、InterSystems IRIS は SQLCODE 100 (データの最後に達した) を発行します。ロールへの割り当て対象として指定されたユーザ (またはロール) が存在しない場合、InterSystems IRIS は SQLCODE -118 エラーを発行します。SuperUser でない場合に、自身が所有していないロールを付与しようとし、さらにそのロールに対する ADMIN OPTION も保有していない場合、InterSystems IRIS は SQLCODE -112 エラーを発行します。

ロールの作成には、[CREATE ROLE](#) 文を使用します。ロール名が [区切り文字付き識別子](#) の場合には、ロールへの割り当て時にロール名を引用符で囲む必要があります。

ロールは、SQL の GRANT コマンドおよび REVOKE コマンド、または InterSystems IRIS システム・セキュリティを使用して付与または削除できます。

- ・ 管理ポータルに進み、[\[システム管理\]](#)、[\[セキュリティ\]](#)、[\[ユーザ\]](#) の順に選択し、現在のユーザを表示します。目的のユーザの名前を選択して、そのユーザの編集オプションを表示し、次に [\[ロール\]](#) タブを選択して、ユーザの 1 つ以上のロールへの割り当て (または割り当て解除) を行います。
- ・ 管理ポータルに進み、[\[システム管理\]](#)、[\[セキュリティ\]](#)、[\[ロール\]](#) の順に選択し、現在のロールを表示します。目的のロールの名前を選択して、そのロールの編集オプションを表示し、次に [\[割り当て先\]](#) タブを選択して、ロールの 1 つ以上のロールへの割り当て (または割り当て解除) を行います。ただし、ObjectScript の [\\$ROLES](#) 特殊変数を使用しても、ロールに対して与えられたロールは表示されません。

## GRANT object-privilege

オブジェクト特権は、ユーザまたはロールに特定のオブジェクトに対する特権を与えます。object-list に関する object-privilege を grantee に付与します。object-list では、現在のネームスペース内の 1 つまたは複数のテーブル、ビュー、ストアド・プロシージャ、またはキューブを指定できます。コンマ区切りのリストを使用することで、1 つの GRANT 文で、複数のオブジェクトに関する複数のオブジェクト特権を複数のユーザおよび/またはロールに付与できます。

以下に、使用できる object-privilege 値を示します。

- ・ [%ALTER](#) および [DELETE](#) の特権は、テーブルやビューの定義へのアクセス権を与えます。
- ・ [SELECT](#)、[INSERT](#)、[UPDATE](#)、[DELETE](#)、および [REFERENCES](#) の各特権は、テーブル・データへのアクセス権を与えます。
- ・ [EXECUTE](#) 特権は、ストアド・プロシージャへのアクセス権を与えます。この特権は、ストアド・プロシージャを実行したり、クエリで [ユーザ定義 SQL 関数](#) を呼び出す場合に必須です。例えば、`SELECT Field1, MyFunc() FROM SQLUser.MyTable` では、SQLUser.MyTable に対しては [SELECT](#) 特権、SQLUser.MyFunc プロシージャに対しては [EXECUTE](#) 特権が必要になります。
- ・ [ALL PRIVILEGES](#) 特権は、テーブルとビューのすべての特権を与えますが、[EXECUTE](#) 特権は与えません。

object-list の値にアスタリスク (\*) ワイルドカードを使用すると、現在のネームスペースのすべてのオブジェクトに対する object-privilege を付与できます。例えば、`GRANT SELECT ON * TO Deborah` は、このユーザに、すべてのテーブルおよびビューに対する [SELECT](#) 特権を与えます。`GRANT EXECUTE ON * TO Deborah` は、このユーザに、非公開でないすべてのストアド・プロシージャに対する [EXECUTE](#) 特権を与えます。



SCHEMA schema-name を object-list 値として使用すると、現在のネームスペース内の指定されたスキーマのすべてのテーブル、ビュー、およびストアド・プロシージャへの object-privilege を付与できます。以下の例により、Sample スキーマ内のすべてのオブジェクトに対する SELECT 特権が、このユーザに付与されます。

```
GRANT SELECT ON SCHEMA Sample TO Deborah
```

これには、このスキーマ内で将来定義されるすべてのオブジェクトが含まれます。複数のスキーマをコンマ区切りのリストとして指定できます。例えば以下の例では、Sample スキーマと Cinema スキーマの両方にあるすべてのオブジェクトに対して SELECT 特権が付与されます。

```
GRANT SELECT ON SCHEMA Sample,Cinema TO Deborah
```

キューブはスキーマ名で修飾していない SQL 識別子です。キューブの object-list を指定するには、CUBE (または CUBES) キーワードを指定する必要があります。キューブに付与できる特権は SELECT のみです。

以下の例では、特定のテーブルについて、特定のユーザーに SELECT と UPDATE の特権を付与しています。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
CreateUser
SET x=$SYSTEM.SQL.Security.UserExists("DeborahTest")
IF x=0 {&sql(CREATE USER DeborahTest IDENTIFY BY birdpw)
    IF SQLCODE != 0 {WRITE "CREATE USER error: ",SQLCODE,!
        QUIT}
    }
ELSE {WRITE "User DeborahTest exists, not changing privileges",!
    QUIT }
GrantPrivsToUser
&sql(GRANT SELECT,UPDATE ON SQLUSER.T1 TO DeborahTest)
WRITE !,"GRANT error code: ",SQLCODE
DropUser
&sql(DROP USER DeborahTest)
IF SQLCODE != 0 {WRITE "DROP USER error: ",SQLCODE,!}
```

特権は、既存のテーブル、ビュー、またはストアド・プロシージャにのみ明示的に付与できます。指定されたオブジェクトが存在しない場合、InterSystems IRIS は SQLCODE -30 エラーを発行します。ただし、スキーマには特権を付与できます。この場合、そのスキーマ内のすべての既存オブジェクトと、そのスキーマ内に将来作成されるすべてのオブジェクト (特権を付与する時点で存在していないオブジェクト) の両方に対する特権を与えます。

テーブルの所有者が PUBLIC の場合、ユーザには、テーブルにアクセスするためのオブジェクト特権を付与する必要はありません。

指定されたユーザが存在しない場合、InterSystems IRIS は SQLCODE -118 エラーを発行します。指定されたオブジェクト特権が既に付与されている場合、InterSystems IRIS は SQLCODE 100 (データの最後に達した) を発行します。

オブジェクト特権は、以下のいずれかによって付与または削除できます。

- ・ GRANT コマンドおよび REVOKE コマンド。
- ・ \$SYSTEM.SQL.Security.GrantPrivilege() メソッドおよび \$SYSTEM.SQL.Security.RevokePrivilege() メソッド。これらのメソッドは、%Status の値を返し、SQLCODE 変数を設定します。すべてのメソッドや関数と同じように、必ず最初に返り値をテストしてください。
  - %Status=1 で SQLCODE=0 の場合：特権は付与または削除されました。
  - %Status=1 で SQLCODE=100 の場合：特権は既に付与または削除されているため、付与または削除されませんでした。
  - %Status が 1 ではなく、SQLCODE が設定されておらず未定義の可能性がある場合：メソッドのエラーのため、特権は付与または削除されませんでした。%Status にはエラーのタイプを示す SQLCODE が含まれます。ObjPriv：特権が無効な場合、SQLCODE -60。ObjList：指定されたオブジェクト・タイプの ObjList オブジェクトが存在しません：SQLCODE -30、-187、-428、または -473。Type：SQLCODE -400、TABLE、VIEW、CUBES、

SCHEMA、または STORED PROCEDURES のオブジェクト・タイプが予期されていました。User : SQLCODE -118、未知またはユニークではないユーザまたはロール。

- InterSystems IRIS システム・セキュリティの使用。管理ポータルに進み、[システム管理]、[セキュリティ]、[ユーザ] (または [システム管理]、[セキュリティ]、[ロール]) の順に選択し、目的のユーザまたはロールの名前を選択してから、[SQLテーブル] タブまたは [SQLビュー] タブを選択します。ドロップダウン・リストから目的の [ネームスペース] を選択します。次に、[テーブルの追加] ボタンまたは [ビューの追加] ボタンを選択します。表示されたウィンドウでスキーマを選択し、1 つまたは複数のテーブルを選択して、特権を割り当てます。

`%CHECKPRIV` コマンドを呼び出すことにより、現在のユーザが指定されたオブジェクト特権を持っているかどうかを確認できます。以下の例のように、`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを呼び出すことにより、指定のユーザが指定されたテーブルレベルのオブジェクト特権を持っているかどうかを確認できます。

### ObjectScript

```
WRITE "SELECT privilege? ", $SYSTEM.SQL.Security.CheckPrivilege("DeborahTest", "1,SQLUSER.TestT1", "s"), !
WRITE "UPDATE privilege? ", $SYSTEM.SQL.Security.CheckPrivilege("DeborahTest", "1,SQLUSER.TestT1", "u"), !
WRITE "DELETE privilege? ", $SYSTEM.SQL.Security.CheckPrivilege("DeborahTest", "1,SQLUSER.TestT1", "d"), !
```

### オブジェクト所有者の特権

テーブル、ビュー、またはプロシージャの所有者は常に、その SQL オブジェクトに対するすべての SQL 特権を暗黙的に持っています。オブジェクトの所有者は、そのオブジェクトがマップされているすべてのネームスペース内でそのオブジェクトに対する特権を持っています。

### GRANT column-privilege

列特権は、指定したテーブルまたはビューの指定した列のリストに対して指定されている特権を、ユーザまたはロールに与えます。これにより、テーブルの一部の列へのアクセスを許可し、同じテーブル内の他の列へのアクセスを許可しないようにできます。また、テーブルやビュー全体に対して特権を定義する GRANT object-privilege オプションよりも、より詳細なアクセス制御を指定できます。grantee に特権を与える際は、テーブルに対するテーブルレベルの特権または列レベルの特権のいずれかを与える必要があります。両方を与える必要はありません。SELECT、INSERT、UPDATE、および REFERENCES の各特権は、個々の列内のデータに対するアクセス権を与えるために使用できます。

テーブルに対する SELECT、INSERT、UPDATE、または REFERENCES object-privilege を WITH GRANT OPTION 付きで付与されているユーザは、その他のユーザに、そのテーブルの列に対する同じタイプの column-privilege を与えることができます。

単一の列、またはコンマ区切りの列のリストを指定できます。column-list は括弧で囲む必要があります。列名は任意の順序で指定できます。重複していてもかまいません。対象となる列特権を、既にその特権を持っている列に与えても何の影響もありません。

以下の例は、2 つの列に対して UPDATE 特権を与えています。

### SQL

```
GRANT UPDATE(Name,FavoriteColors) ON Sample.Person TO Deborah
```

列特権はテーブルまたはビューに与えることができます。ユーザ・リスト、ロール・リスト、\*、および PUBLIC を含む、あらゆるタイプの grantee に対して列特権を与えることができます。ただし、特権、フィールド名、またはテーブル名にアスタリスク (\*) ワイルドカードを使用することはできません。

ユーザがテーブルに新しいレコードを挿入すると、データは、その列特権を与えられているフィールドにのみ挿入されます。その他すべてのデータ列は、定義された列の既定値か、既定値が定義されていない場合は NULL のいずれかに設定されます。RowID 列や Identity 列に列レベルの INSERT または UPDATE 特権を与えることはできません。INSERT 時、InterSystems SQL は RowID および (必要に応じて) Identity 列の値を自動的に入力します。

列レベルの特権は、SQL の GRANT コマンドおよび REVOKE コマンド、または InterSystems IRIS システム・セキュリティを使用して付与または削除できます。管理ポータルに進み、[システム管理]、[セキュリティ]、[ユーザ] (または [システム管理]、[セキュリティ]、[ロール]) の順に選択し、目的のユーザまたはロールの名前を選択してから、[SQLテーブル] タブまたは [SQLビュー] タブを選択します。ドロップダウン・リストから目的の [ネームスペース] を選択します。次に、[列の追加] ボタンを選択します。表示されたウィンドウでスキーマを選択し、テーブルを選択して 1 つまたは複数の列を選択し、特権を割り当てます。

## 複数の特権の付与

単一の GRANT 文を使用して、以下の特権の組み合わせを指定できます。

- ・ 1 つ以上のロール。
- ・ 1 つ以上のテーブルレベルの特権と、1 つ以上の列レベルの特権。複数のテーブルレベルの特権と列レベルの特権を指定するには、特権を column-list の直前に置いて、列レベルの特権を与える必要があります。そうでない場合は、テーブルレベルの特権が与えられます。
- ・ 1 つ以上の admin-privilege。admin-privilege とロール名またはオブジェクト特権を同一の GRANT 文で指定することはできません。これを実行しようとする、SQLCODE -1 エラーが返されます。

以下の例は、Deborah にテーブルレベルの SELECT 特権と UPDATE 特権、および列レベルの INSERT 特権を与えています。

### SQL

```
GRANT SELECT,UPDATE,INSERT(Name,FavoriteColors) ON Sample.Person TO Deborah
```

以下の例は、Deborah に列レベルの SELECT、INSERT、および UPDATE 特権を与えています。

### SQL

```
GRANT SELECT(Name,FavoriteColors),INSERT(Name,FavoriteColors),UPDATE(FavoriteColors) ON Sample.Person TO Deborah
```

## WITH GRANT OPTION 節

オブジェクトの所有者は自動的に、このオブジェクトにかかわるすべての特権を維持します。GRANT 文の TO 節は、アクセス権が与えられるユーザまたはロールを指定します。grantee を指定する TO オプションを使用した後は、WITH GRANT OPTION キーワードを任意で指定して、grantee が他のユーザに同じ特権を与えることもできます。WITH GRANT OPTION キーワード節は、オブジェクト特権または列特権と共に使用できます。CASCADE 付きの REVOKE コマンドを使用すると、付与されているこの階層式の一連の特権を元に戻すことができます。

例えば、以下のコマンドを使用して、EMPLOYEES テーブルの %ALTER、SELECT、または INSERT 特権を Chris というユーザに与えることができます。

### SQL

```
GRANT %ALTER, SELECT, INSERT
    ON EMPLOYEES
    TO Chris
```

Chris に他のユーザに対するこれらの特権を与えることができるようにするために、GRANT コマンドは WITH GRANT OPTION 句を含みます。

### SQL

```
GRANT %ALTER, SELECT, INSERT
    ON EMPLOYEES
    TO Chris WITH GRANT OPTION
```

%SQLCatalogPriv.SQLUsers() メソッド呼び出しを使用した GRANT 文の結果を見つけることができます。

スキーマ WITH GRANT OPTION に特権を付与すると、特権受領者 (1 人または複数人) は、同じスキーマ特権を他のユーザに付与できるようになります。ただし、特権受領者は、そのユーザがその特定のオブジェクトの WITH GRANT OPTION に関する特権を明示的に与えられていない限り、そのスキーマ内の指定されたオブジェクトに対する特権を付与することはできません。詳細は、以下の例を参照してください。

- ・ UserA および UserB には、最初は特権はありません。
- ・ UserA に、スキーマ Sample WITH GRANT OPTION に対する SELECT 特権を付与します。
- ・ UserA は、スキーマ Sample に対する SELECT 特権を UserB に付与できます。
- ・ UserA は、テーブル Sample.Person に対する SELECT 特権を UserB に付与することはできません。

## WITH ADMIN OPTION 節

WITH ADMIN OPTION 節が grantee に特権を与えると、grantee は自分が与えられたのと同じ特権を他のユーザに与える権利を持つことになります。システム特権を与えるには、そのユーザが WITH ADMIN OPTION でシステム特権を与えられている必要があります。

ユーザがロールを与えることができるのは、自分に WITH ADMIN OPTION でそのロールが与えられているか、%Admin\_Secure:"U" リソースを持っている場合です。

WITH ADMIN OPTION での特権の付与は、同じ特権のこのオプションなしでの前回の付与に置き換わります。したがって、ユーザに WITH ADMIN OPTION なしで特権を与え、次に同じ特権を WITH ADMIN OPTION 付きで同じユーザに与えると、そのユーザは WITH ADMIN OPTION 権限を保有します。一方、WITH ADMIN OPTION なしでの付与は、同じ特権のこのオプション付きでの前回の付与に置き換わりません。特権から WITH ADMIN OPTION 権限を削除するには、その特権を削除して、この節なしで同じ特権を再度与える必要があります。

## 特権のエクスポート

特権は、\$SYSTEM.SQL.Schema.ExportDDL() メソッドを使用してエクスポートできます。このメソッドでテーブルを指定すると、InterSystems IRIS では、テーブルレベルのすべての特権とそのテーブルに与えられた列レベルのすべての特権がエクスポートされます。詳細は、“インターシステムズ・クラス・リファレンス”を参照してください。

## InterSystems IRIS セキュリティ

埋め込み SQL 内で GRANT を使用する前に、適切な特権を持つユーザとしてログインする必要があります。特権がない場合は、SQLCODE -99 エラー (特権違反) が返されます。\$SYSTEM.Security.Login() メソッドを使用して、以下のようユーザに適切な特権を割り当ててください。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、%Service\_Login:Use 特権が必要です。詳細は、“インターシステムズ・クラス・リファレンス”の“%SYSTEM.Security”を参照してください。

## 特権の強制

SQL 特権は、ODBC、JDBC、およびダイナミック SQL (%SQL.Statement) を介してのみ適用されます。

システム全体への特権の強制は、\$SYSTEM.SQL.Util.SetOption("SQLSecurity") メソッド呼び出しの設定によって決まります。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[SQL ON:] の設定が表示されます。

既定値は 1 (はい) です。ユーザは特権が付与されているテーブルやビューのみでアクションを実行できます。この設定を推奨します。このオプションが 0 (いいえ) に設定された場合、この設定の変更後に開始された新しいプロセスすべてで SQL セキュリティは無効になります。つまり、特権ベースのテーブルやビューのセキュリティは抑制されていることを意味します。ユーザを指定しなくてもテーブルの作成が可能になります。この場合、管理ポータルはユーザとして "\_SYSTEM"

を、埋め込み SQL はユーザとして "" (空文字列) を割り当てます。ユーザは特権がなくてもテーブルやビューに対してアクションを実行することができます。

## 引数

### grantee

コンマで区切られた、1 つ以上のユーザまたはロールのリスト。有効な値は、ユーザのリスト、ロールのリスト、"\*"、または \_PUBLIC です。アスタリスク(\*)は、現在定義されていて %All ロールを持たないすべてのユーザを指定します。\_PUBLIC キーワードにより、現在定義されているすべてのユーザと未定義のユーザが指定されます。

### admin-privilege

許可されている管理者レベル特権または管理者レベル特権のコンマで区切られたリスト。以下のうち 1 つ以上をさまざまな順序で使用してリストを構成できます：

%CREATE\_METHOD、%DROP\_METHOD、%CREATE\_FUNCTION、%DROP\_FUNCTION、%CREATE\_PROCEDURE、%DROP\_PROCEDURE、%CREATE\_QUERY、%DROP\_QUERY、%CREATE\_TABLE、%ALTER\_TABLE、%DROP\_TABLE、%CREATE\_VIEW、%ALTER\_VIEW、%DROP\_VIEW、%CREATE\_TRIGGER、%DROP\_TRIGGER

%DB\_OBJECT\_DEFINITION：上記の 16 種類の特権をすべて与えます。

INSERT、UPDATE、および DELETE 操作のための %NOCHECK、%NOINDEX、%NOLOCK、%NOTRIGGER 特権。

[BUILD INDEX](#) コマンドに特権を与える %BUILD\_INDEX。

### role (ロール)

特権が与えられているロール、またはコンマで区切られたロールのリスト

### object-privilege

付与対象の基本レベル特権、または基本レベル特権のコンマで区切られたリスト。%ALTER、DELETE、SELECT、INSERT、UPDATE、EXECUTE、および REFERENCES のうち 1 つ以上を使用してリストを構成できます。"ALL [PRIVILEGES]" または "\*" のどちらかを引数の値として使用して、テーブルとビューのすべての特権を与えることもできます。キューブに付与できる特権は SELECT のみです。

### object-list

object-privilege (複数の場合もあります) の付与先とする 1 つ以上の[テーブル](#)、[ビュー](#)、ストアド・プロシージャ、またはキューブのコンマ区切りリスト。SCHEMA キーワードを使用して、指定されたスキーマ内の全オブジェクトへの object-privilege の付与を指定できます。"\*" を使用すると、現在のネームスペース内で非表示になっていないすべてのストアド・プロシージャ、またはすべてのテーブルに対する object-privilege の付与を指定することができます。object-list がキューブの場合、CUBE (または CUBES) キーワードが必要です。また、付与できる特権は SELECT のみです。

### column-privilege

1 つ以上のリストされた列に付与される基本レベル特権。指定できるオプションは、SELECT、INSERT、UPDATE、および REFERENCES です。

### column-list

コンマで区切り、括弧で囲んだ 1 つ以上の列名のリスト。

### table

column-list 列を含む [テーブル](#) またはビューの名前。



## 例

以下の例は、ユーザとロールを作成してからそのロールをユーザに割り当てます。ユーザまたはロールが既に存在する場合は、SQLCODE -118 エラーが発行されます。特権またはロールが既に割り当てられている場合、エラーは発行されません (SQLCODE = 0)。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
CreateUser
SET x=$SYSTEM.SQL.Security.UserExists("MarthaTest")
IF x=0 {&sql(CREATE USER MarthaTest IDENTIFY BY birdpw)
      IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                      QUIT}
      }
ELSE {WRITE "User MarthaTest exists, not changing its roles",!
      QUIT }
CreateRoleAndGrant
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE error code: ",SQLCODE
&sql(GRANT %CREATE_TABLE TO workerbee)
WRITE !,"GRANT privilege error code: ",SQLCODE
&sql(GRANT workerbee TO MarthaTest)
WRITE !,"GRANT role error code: ",SQLCODE
```

以下の例は、複数の特権の割り当てを示しています。ここでは、1 人のユーザと 2 つのロールを作成します。単一の GRANT 文は、これらのロールと admin-privilege のリストをユーザに割り当てます。ユーザまたはロールが既に存在する場合は、SQLCODE -118 エラーが発行されます。特権またはロールが既に割り当てられている場合、エラーは発行されません (SQLCODE = 0)。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
CreateUser
SET x=$SYSTEM.SQL.Security.UserExists("NoahTest")
IF x=0 {&sql(CREATE USER NoahTest IDENTIFY BY birdpw)
      IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                      QUIT}
      }
ELSE {WRITE "User NoahTest exists, not changing its roles",!
      QUIT }
Create2RolesAndGrant
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE 1 error code: ",SQLCODE
&sql(CREATE ROLE drone)
WRITE !,"CREATE ROLE 2 error code: ",SQLCODE
&sql(GRANT workerbee,drone,%CREATE_TABLE,%DROP_TABLE TO NoahTest)
WRITE !,"GRANT roles & privileges error code: ",SQLCODE
```

以下の例は、現在定義されていて %All ロールを持たないすべてのユーザに、現在のネームスペース内のすべてのテーブルに対する 7 つすべての基本的な特権を与えています。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(GRANT * ON * TO *)
```

## 関連項目

- [%CHECKPRIV REVOKE](#)
- [SELECT INSERT DELETE UPDATE](#)
- [CREATE USER CREATE ROLE](#)
- [SQL のユーザ、ロール、および特権](#)
- [CREATE FUNCTION CREATE METHOD CREATE PROCEDURE CREATE QUERY](#)
- [CREATE TABLE CREATE VIEW CREATE TRIGGER](#)

- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数



# INSERT (SQL)

テーブルに新しい行を追加します。

## 構文

### 単一行の挿入

```
INSERT INTO table (column, column2, ...) VALUES (value, value2, ...)
INSERT INTO table SET column = value, column2 = value2, ...
INSERT INTO table DEFAULT VALUES
INSERT INTO table VALUES (value, value2, ...)
INSERT INTO table VALUES :array()
```

### 複数行の挿入

```
INSERT INTO table query
INSERT INTO table (column, column2, ...) query
```

### 挿入オプション

```
INSERT table ...
INSERT %keyword [INTO] table ...
```

## 説明

INSERT コマンドは、単一の行をテーブルに挿入するか、または SELECT クエリの結果を使用してテーブルに複数の行を挿入します。このコマンドは、指定されたすべての列のデータを挿入し、指定されていない列の値を既定の NULL または定義済みの既定値にします。%ROWCOUNT 変数は挿入された行数に設定されます。

挿入される行が既に存在している場合 (UNIQUE チェックが失敗する場合など)、INSERT はエラーを生成します。このような場合に既存の行を更新するには、[INSERT OR UPDATE](#) を使用します。

### 単一行の挿入

- INSERT INTO [table](#) ([column](#), column2, ...) VALUES ([value](#), value2, ...) 指定したテーブルの列に、値の行を挿入します。VALUES 節の値は、列のリスト内の列の名前と位置的に対応する必要があります。

既定では、INSERT は、全か無かのイベントです。1 行すべてが挿入されるか、まったく挿入されないかのいずれかです。InterSystems IRIS® は、SQLCODE ステータス変数を返し、INSERT の成功もしくは失敗を示します。テーブルに行を挿入するには、その挿入が table、column、および value 引数で記述されたすべての要件を満たす必要があります。

以下の文は、新しい行を Sample.Records テーブルに挿入し、StatusDate 列の値を '05/12/22' に、Status 列の値を 'Purged' に設定します。

### SQL

```
INSERT INTO Sample.Records (StatusDate,Status) VALUES ('05/12/22','Purged')
```

#### 例：指定した値を使用して、行をテーブルに挿入する

- INSERT INTO table VALUES (value, value2, ...) は、テーブルの値の行を列番号順に挿入します。データ値は、定義されている列リストと位置的に対応する必要があります。指定可能なすべてのテーブル列に値を指定する必要があります。定義済みの既定値を使用することはできませんが、空の文字列を値として指定することはできます。RowID 列は指定可能ではないため、RowID 値は VALUES リストに含めないでください。

以下の文は、4 つの値を持つ行を順に Sample.Address テーブルに挿入します。この文では、テーブルに Street、City、State、ZipCode など、ちょうど 4 つの列が含まれ、それらのデータが挿入される値に対応していると想定しています。

## SQL

```
INSERT INTO Sample.Address VALUES ('22 Main St.','Anytown','PA','65342')
```

例：指定した値を使用して、行をテーブルに挿入する

- INSERT INTO table SET column = value, column2 = value2, ... は、特定の列の値を明示的に設定することにより、値の行を挿入します。

以下の文は、INSERT INTO table (column, column2, ...) VALUES (value, value2, ...) 構文と同じ操作を実行します。

## SQL

```
INSERT INTO Sample.Records SET StatusDate='05/12/22',Status='Purged'
```

例：指定した値を使用して、行をテーブルに挿入する

- INSERT INTO table DEFAULT VALUES は、既定の列値のみを含む行をテーブルに挿入します。
  - 定義済みの既定値を持つ列は、その値に設定されます。
  - 定義された既定値がない列は、NULL に設定されます。

以下の文は、既定の列値を持つ行を Sample.Person テーブルに挿入します。

## SQL

```
INSERT INTO Sample.Person DEFAULT VALUES
```

NOT NULL 制約を指定して定義され、DEFAULT が定義されていない列の場合、この操作は SQLCODE -108 で失敗します。

UNIQUE 制約を指定して定義されている列は、この文を使用して挿入できます。列が DEFAULT 値なしで UNIQUE 制約を指定して定義されている場合は、DEFAULT VALUES クラスを繰り返すことで、この UNIQUE 列が NULL に設定されている行が複数挿入されます。列が UNIQUE 制約と DEFAULT 値を指定して定義されている場合、この文は 1 回しか使用できません。2 回目の呼び出しは、SQLCODE -119 エラーで失敗します。

DEFAULT VALUES は、RowID 列、IDENTITY 列、SERIAL (%Counter) 列、ROWVERSION 列などのカウンタ列に、システムで生成された整数値を含む行を挿入します。

- INSERT INTO table VALUES :array() は、ホスト変数として指定された配列の値を、テーブルの列に挿入します。この構文は、埋め込み SQL でのみ使用できます。この配列の値は、暗黙的に、列番号順に行の列に対応している必要があります。指定可能な各列に値を指定する必要があります。列順を使用する INSERT では、定義された列の既定値を取ることはできません。

以下の文は、実行時に配列に値を移入し、挿入する列の指定を実行時まで遅らせることができます。その他すべての挿入では、INSERT を準備する際に、挿入する列を指定する必要があります。リンク・テーブルでこの構文を使用することはできません。これを実行しようすると、SQLCODE -155 エラーが返されます。

このクラス・メソッドは、埋め込み SQL を使用して、配列を Sample.FullName テーブルに挿入します。myarray(1) は、RowID 列用に予約されているため、指定されません。

## Class Member

```
ClassMethod EmbeddedSQLInsertHostVarArray()
{
    set myarray(2)="Juanita"
    set myarray(3)="Pybus"
    &sql(INSERT INTO Sample.FullName VALUES :myarray())
    if SQLCODE '= 0 {
        write !, "Insert failed, SQLCODE= ", SQLCODE, ! ,%msg
        quit
    }
    write !,"Insert succeeded" quit
}
```

ホスト変数と配列については、“[添え字付き配列としてのホスト変数](#)”を参照してください。

例：

- [埋め込み SQL を使用してデータを挿入する](#)
- [ストリーム・データをテーブルに挿入する](#)

## 複数行の挿入

- ・ INSERT INTO table [query](#) は、[SELECT](#) クエリの結果セットから取得したデータの行を挿入します。結果セット内の列は、テーブル内の列と一致する必要があります。INSERT を SELECT と共に使用することで、別のテーブルから抽出した既存のデータをテーブルに移入できます。

以下の文は、ソース・テーブル Sample.SrcTable と同じ値をテーブル Sample.DupTable に移入します。この 2 つのテーブルの列数、列名、および列順は同じである必要があります。

### SQL

```
INSERT INTO Sample.DupTable SELECT * FROM Sample.SrcTable
```

例：[SELECT クエリを使用して別のテーブルからデータを挿入する](#)

- ・ INSERT INTO table (column, column2, ...) [query](#) は、クエリ結果セットのデータの行を、指定された列に挿入します。INSERT 文は、行内の指定されていない列値を NULL または既定値に設定します。

以下の文は、Sample.Person の Name および DOB 列のクエリ結果セットのデータを、Sample.Kids テーブルの一致する列に挿入します。

### SQL

```
INSERT INTO Sample.Kids (Name,DOB) SELECT Name,DOB FROM Sample.Person WHERE Age <= 18
```

例：[SELECT クエリを使用して別のテーブルからデータを挿入する](#)

## 挿入オプション

- ・ INSERT table ... は、INTO キーワードを省略します。
- ・ INSERT [%keyword](#) [INTO] table ... は、1 つまたは複数の [%keyword](#) オプションを空白で区切って設定します。有効なオプションは、%NOCHECK、%NOFPLAN、%NOINDEX、%NOJOURN、%NOLOCK、%NOTRIGGER、%PROFILE、および %PROFILE\_ALL です。

## 引数

### table

挿入を実行する**テーブル**または**ビュー**の名前。テーブル名またはビュー名は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。未修飾の名前は、**スキーマ検索パス** (指定されている場合)、または**既定のスキーマ名**を使用して、そのスキーマと照合されます。

table 引数の代わりにサブクエリを使用して INSERT を実行することもできます。以下に例を示します。

### SQL

```
INSERT INTO (SELECT column1 AS c1 FROM MyTable) (c1) VALUES ('test')
```

行をテーブルに挿入するには、適切な**テーブルレベルの特権**が必要です。

### column

列名、または**列名**のコンマ区切りのリスト。後者の場合は、値のリストに対応した順序で指定します。省略した場合、値のリストは列番号順にすべての列に適用されます。

行をテーブルに挿入するには、適切な**列レベルの特権**が必要です。

### value

column 内の対応する列のデータ値を指定する、VALUES 節で指定されるスカラ式またはコンマ区切りのスカラ式のリスト。指定した値の数が列数より少ない場合、SQLCODE -62 エラーが生成されます。指定した値の数が列数より多い場合、SQLCODE -116 エラーが生成されます。

テーブルに行を挿入するには、value で指定される列値が、以下の要件を満たしている必要があります。

- ・ 各列の値が**データ型**の妥当性検証に合格する必要があります。不適切な列値を列のデータ型に挿入しようとする、SQLCODE -104 エラーが返されます。この要件は挿入されるデータ値にのみ適用されます。**DEFAULT** 値を取る列は、データ型の検証またはデータ・サイズの検証に合格する必要はありません。適切であるかどうかは、挿入されるデータ値のデータ型ではなく、列のデータ型によって決まります。例えば、日付列に文字列値を挿入しようすると、現在のモードでその文字列が日付の検証に合格しない限り失敗します。ただし、文字列の列に日付値を挿入しようとした場合は成功します。INSERT は、日付をリテラル文字列としてテーブルに挿入します。データを挿入先のデータ型に変換するには、**CONVERT** 関数を使用します。
- ・ 各データ値は、その列の MAXLEN、MAXVAL、および MINVAL の範囲内でなければなりません。例えば、VARCHAR(24) として定義された列に 24 文字より長い文字列を挿入しようしたり、TINYINT として定義された列に 127 より大きい数値を挿入しようすると、SQLCODE -104 エラーが発生します。
- ・ ODBC または JDBC を介して無効な DOUBLE 数値を指定すると、SQLCODE -104 エラーが発生します。
- ・ 挿入されるデータ値は表示モードから論理モードへの変換に合格する必要があります。InterSystems SQL では、データを logical モード形式で格納します。一部のデータ型では、論理形式が表示形式と異なることがあります。例えば、**日付**データは日数を示す整数値として格納され、**時刻**データは午前 0 時 00 分からの秒数として格納され、%List データはエンコードされた文字列として格納されます。文字列や数値など、その他のデータ型の場合、変換は必要ありません。論理格納値に変換できない形式で値を挿入しようすると、エラーが発生します。モード変換の詳細は、“**データ表示オプション**” を参照してください。
- ・ 各データ値は、挿入先の列のデータ制約の妥当性検証に合格する必要があります。
  - **NOT NULL として定義された列**には、データ値が与えられる必要があります。**DEFAULT** 値のない列の場合、データ値を指定しないと、SQLCODE -108 エラーが発生します。
  - データ値は、列または列のグループで定義された **UNIQUE データ制約**に従う必要があります。UNIQUE 列 (主キー列など)、または UNIQUE 列 グループに重複値を挿入しようすると、SQLCODE -119 エラーが発生

します。このエラーは、値を指定せず、その列の **DEFAULT** の 2 回目の使用により重複値が指定される場合にも発生します。

- **VALUELIST パラメータ**で永続クラス・プロパティとして定義された列は、VALUELIST でリストされた値のみを受け入れることができ、それ以外の場合は値なし (NULL) となります。VALUELIST の値では、大文字と小文字が区別されます。VALUELIST の値と一致しないデータ値を指定すると、SQLCODE -104 の列のエラーが発生します。
- ・ 数値は**キャノニック形式**で挿入されますが、先頭と末尾の 0 や、先頭の複数の符号を付けて指定できます。ただし、SQL では、2 つの連続したマイナス記号は、1 行コメント文字として解析されます。したがって、先頭に 2 つの連続したマイナス符号を付けて数値を指定しようとする、SQLCODE -12 エラーになります。
- ・ 既定では、システム生成値が格納される **RowID**、IDKey、IDENTITY などの列を、値の挿入対象に指定することはできません。既定では、これらの列のいずれかに非 NULL 値を挿入すると、SQLCODE -111 エラーが返されます。これらの列のいずれかに NULL を挿入すると、InterSystems IRIS はその NULL をシステム生成値でオーバーライドします。エラーは生成されません。
  - テーブルで **ROWVERSION** 列を定義すると、行が挿入された場合、その列にはシステムで生成されたカウンタ値が自動的に割り当てられます。ROWVERSION 列に値を挿入すると、SQLCODE -138 エラーが返されます。
  - ユーザが指定した値を受け取る IDENTITY 列を作成することができます。SetOption("IdentityInsert") メソッドを設定すると、IDENTITY 列の既定の制約をオーバーライドし、IDENTITY 列に一意的な整数値を挿入できるようになります。この制約の現在の設定を返すには、GetOption("IdentityInsert") メソッドを呼び出します。IDENTITY 列に値を挿入すると IDENTITY カウンタが変更され、後続のシステム生成値が、このユーザ指定の値からインクリメントを開始するようになります。IDENTITY 列に NULL を挿入すると、SQLCODE -108 エラーが発生します。
  - IDKey データには次の制限があります。1 つのインデックスに複数の IDKey 列がある場合は、列の区切り文字列として "||" (二重の垂直バー) が使用されています。したがって、IDKey 列にこの文字列を使用したデータ値を挿入することはできません。
- ・ INSERT コマンドで **%NOCHECK キーワード**が指定されているか、外部キーが NOCHECK キーワードで定義されていない限り、挿入される値は、**外部キーの参照整合性**に違反してはなりません。外部キーの参照整合性に違反する挿入が試みられると、SQLCODE -121 エラーが発生します。テーブルの外部キー制約および外部キー制約の名前のリストの詳細は、[\[カタログの詳細\]](#)の[\[制約\]](#)を参照してください。
- ・ データ値をサブクエリにすることはできません。サブクエリを列値に指定しようとする、SQLCODE -144 エラーが返されます。

## 非表示の文字値

非表示文字を含む値を挿入するには、**CHAR** 関数と**連結演算子**を使用します。例えば以下の文では、文字 "A"、改行文字、および文字 "B" で構成される文字列が挿入されます。

## SQL

```
INSERT INTO MyTable (Text) VALUES ('A' || CHAR(10) || 'B')
```

関数の結果を連結するには、ObjectScript で使用される **\_** 連結演算子ではなく、**||** 連結演算子を使用する必要があります。

## 特殊変数値

value を以下の特殊変数のいずれかとして指定できます。

- ・ **%TABLENAME** または **%CLASSNAME** **疑似列変数**キーワード。%TABLENAME は、現在のテーブル名を返します。%CLASSNAME は、現在のテーブルに対応するクラスの名前を返します。

- 以下の 1 つ以上の ObjectScript 特殊変数 (これらの省略形式を含む) : \$HOROLOG、\$JOB、\$NAMESPACE、\$TLEVEL、\$USERNAME、\$ZHOROLOG、\$ZJOB、\$ZNSPACE、\$ZPI、\$ZTIMESTAMP、\$ZTIMEZONE、\$ZVERSION。

## リスト値

InterSystems IRIS は、リスト構造のデータ型、%List、データ型クラス %Library.List をサポートしています。この圧縮バイナリ形式は、InterSystems SQL で対応するネイティブなデータ型にマップしません。代わりに、データ型 VARBINARY に対応しており、その MAXLEN の既定値は 32749 です。このため、[ダイナミック SQL](#) は、型 %List のプロパティ値を設定するときに、INSERT も UPDATE も使用できません。詳細は、“[データ型](#)”を参照してください。

## IDENTITY とカウンタ値

InterSystems SQL により、[IDENTITY](#) 列などのシステムで生成される列、または [RowVersion](#)、[AutoIncrement](#)、および [Serial カウンタ](#)列など、各 INSERT または UPDATE 操作を自動的にインクリメントする列を定義できます。

これらの列のいずれかに値を挿入すると、列タイプによっては、INSERT 操作が失敗する可能性があります。

列タイプ	INSERT 可能か
IDENTITY	既定では不可。  IDENTITY を構成して挿入された値を受け入れるには、テーブルを定義するときに、%CLASSPARAMETER ALLOWIDENTITYINSERT=1 の値を設定します。詳細は、“ <a href="#">IDENTITY キーワードを使用した名前付き RowId 列の作成</a> ”を参照してください。
ROWVERSION	ROWVERSION 列には、ユーザ定義の値、計算値、または既定値を挿入することはできません。
SERIAL	はい。
AUTO_INCREMENT	正の整数値を指定すると、INSERT は、列に値を挿入し、既定のカウンタ値をオーバーライドします。  値なし、0 (ゼロ)、または数値以外の値を指定すると、INSERT は指定された値を無視し、この列の値を 1 だけインクリメントして、その値を列に挿入します。

## 計算値

以下の条件で、[COMPUTECODE](#) が定義された列に値を挿入できます。



定義された列	値の振る舞い
関連する計算キーワードのない COMPUTE CODE	値が計算され、INSERT に格納されます。UPDATE では値は変更されません。
COMPUTE ON CHANGE を使用する COMPUTE CODE	値が計算され、INSERT に格納されます。 値が再計算され、UPDATE に格納されます。
DEFAULT と COMPUTE ON CHANGE を使用する COMPUTE CODE	既定値が INSERT に格納されます。値が計算され、UPDATE に格納されます。
CALCULATED または TRANSIENT を使用する COMPUTE CODE	有効な値を計算列に挿入すると、InterSystems IRIS はその行を挿入し、ROWCOUNT をインクリメントします。ただし、この値は格納されていないため、挿入されません。この列を照会すると、InterSystems SQL は値を再計算し、その値を返します。  このタイプの列が外部キー制約の一部である場合、参照整合性チェックを実行するために、この列の値が挿入時に計算されます。この計算値は格納されません。

計算コードにプログラミング・エラーがある場合（例えば、ゼロでの除算）、INSERT 操作は SQLCODE -415 エラーで失敗します。

詳細は、“[INSERT または UPDATE 時の計算フィールドの値](#)”を参照してください。

## query

その結果セットが、[column](#) で指定された対応する列にデータ値を指定する [SELECT クエリ](#)。

SELECT クエリは 1 つ以上のテーブルから列データを抽出し、INSERT コマンドはこの列データを含むテーブル内に、対応する新しい行を作成します。挿入されるデータがテーブル列に収まることのできる限り、対応する列は、さまざまな列名および列長を持つことができます。対応する列がデータ型および長さ検証チェックに合格しない場合、InterSystems SQL は SQLCODE -104 エラーを生成します。

挿入される行の数を制限するには、[SELECT](#) 文内で [TOP](#) 節を指定します。クエリがこれらの上位行のうちどれを選択するかを決定するには、SELECT 文で [ORDER BY](#) 節を使用します。

特定の列の一意の値のみを挿入するには、クエリで [GROUP BY](#) 節を指定します。既定では、GROUP BY により、値がグループ化のために大文字に変換されます。挿入された値の大文字/小文字を維持するには、クエリで [%EXACT](#) 照合を指定します。以下に例を示します。

## SQL

```
INSERT INTO Sample.UniquePeople (Name, Age)
  SELECT Name, Age FROM Sample.Person
  WHERE Name IS NOT NULL GROUP BY %EXACT Name
```

SELECT で INSERT を使用すると、[%ROWCOUNT](#) 変数は挿入した行の数に設定されます (0 または正の整数)。

## array

[ホスト変数](#)として指定する値の動的なローカル配列。この値は、埋め込み SQL にのみ適用されます。

配列の最下位の添え字は指定しないでください。:myupdates()、:myupdates(5,)、および :myupdates(1,1,) はすべて、有効な指定になります。



## %keyword

INSERT 処理を構成するキーワード・オプション。キーワード・オプションは順不同で指定できます。複数のキーワード・オプションは、スペースで区切ります。

以下の表で、指定できるキーワード・オプションについて説明します。

キーワード・オプション	説明
-------------	----

キーワード・オプション	説明
%NOCHECK	<p>一意の値のチェックおよび外部キーの参照整合性チェックを無効にします。%NOCHECK は、列のデータ型、最大長、列のデータ制約の検証も無効にします。ビューを介して INSERT を実行すると、WITH CHECK OPTION 検証は実行されません。</p> <p>注釈    %NOCHECKを使用した挿入は、無効データとなります。一括挿入や一括更新を高速化するなどのためにこのオプションを有効にする場合は、信頼できるソースからのデータであることを確認してください。</p> <p>このオプションでは、対応する %NOCHECK <a href="#">管理特権</a>を設定する必要があります。この特権を設定しない場合、挿入時に SQLCODE -99 エラーが発生します。</p> <p>%NOCHECK を指定する際に一意でないデータ値の挿入を防止するには、INSERT の前に <a href="#">EXISTS</a> チェックを実行します。</p> <p>外部キーの参照整合性チェックを無効にするには、代わりに \$SYSTEM.SQL.SetFilerRefIntegrity() メソッドを使用します。または、NOCHECK キーワードを使用してテーブルで外部キーを定義することで、外部キーの参照整合性チェックが実行されないようにすることができます。外部キーの参照整合性の詳細は、“<a href="#">外部キーの参照整合性チェック</a>”を参照してください。</p>
%NOFPLAN	<p>この操作の凍結プランを無視し、新しいクエリ・プランを生成します。凍結プランは保持されますが、使用されません。詳細は、“<a href="#">凍結プラン</a>”を参照してください。</p>
%NOINDEX	<p>INSERT 処理の際にインデックス・マップの設定を無効にします。このオプションでは、対応する %NOINDEX <a href="#">管理特権</a>を設定する必要があります。この特権を設定しない場合、挿入時に SQLCODE -99 エラーが発生します。</p> <p>挿入時にインデックス付けされなかった行を含むテーブルのインデックスを構築するには、<a href="#">BUILD INDEX</a> を使用します。</p>
%NOJOURN	<p>挿入操作の間、ジャーナリングを抑制し、トランザクションを無効化します。プルされたトリガを含め、行での変更はどれもジャーナリングされません。%NOJOURN が指定された文の後に ROLLBACK を実行した場合、その文で行われた変更はロールバックされません。このオプションでは、対応する %NOJOURN <a href="#">管理特権</a>を設定する必要があります。この特権を設定しない場合、挿入時に SQLCODE -99 エラーが発生します。</p>
%NOLOCK	<p>INSERT の実行時に行のロックを無効にします。このオプションは、単独のユーザまたは処理がデータベースを更新する際にのみ設定します。このオプションでは、対応する %NOLOCK <a href="#">管理特権</a>を設定する必要があります。この特権を設定しない場合、挿入時に SQLCODE -99 エラーが発生します。</p>
%NOTRIGGER	<p>INSERT の処理時にベース・テーブル<a href="#">挿入トリガ</a>はかけません。このオプションでは、対応する %NOTRIGGER <a href="#">管理特権</a>を設定する必要があります。この特権を設定しない場合、挿入時に SQLCODE -99 エラーが発生します。</p>

キーワード・オプション	説明
%PROFILE	INSERT 文のパフォーマンス分析統計 (SQLStats) を生成します。
%PROFILE_ALL	<ul style="list-style-type: none"> <li>・ %PROFILE はメイン・クエリ・モジュールに対して SQLStats を収集します。</li> <li>・ %PROFILE_ALL はメイン・クエリ・モジュールとそのすべてのサブクエリ・モジュールに対して SQLStats を収集します。</li> </ul> <p>生成される文は、<a href="#">SQL パフォーマンス分析ツールキット</a>を有効にして生成した文と同じです。このキーワード・オプションにより、調査の必要のない他のコンパイルされた文については統計を無効にしたまま、個々の文をプロファイリングおよび調査することができます。これらの統計の詳細は、“<a href="#">SQL 実行時統計情報</a>”を参照してください。</p>

## 例

### 指定した値を使用して、行をテーブルに挿入する

この例では、新しい値の行をテーブルに挿入するさまざまな方法を示します。

会社データを含むテーブルを作成します。このテーブルには、会社名（一意である必要があります）と本社がある国という、2 つの必須列があります。2 番目の列、Revenue（収益）は必須ではなく、既定値 0 を持ちます。

#### SQL

```
CREATE TABLE Sample.Company (
    Name VARCHAR(20) UNIQUE NOT NULL,
    Revenue INTEGER DEFAULT 0,
    Country VARCHAR(10) NOT NULL)
```

テーブルにデータ行を挿入します。列の値は、テーブルの列順と同じ順序で指定する必要があります。

#### SQL

```
INSERT INTO Sample.Company VALUES ('CompanyA',10000,'BEL')
```

別のデータ行を、今度は挿入する列名を指定して挿入します。この文は、Revenue 列を省略しているため、InterSystems SQL はこの列の値を既定値 0 に設定します。

#### SQL

```
INSERT INTO Sample.Company (Name,Country) VALUES ('CompanyB','CAN')
```

3 行目のデータを、今度は column=value 構文を使用して挿入します。column=value ペアは、テーブル列の順序である必要はありません。

#### SQL

```
INSERT INTO Sample.Company Set Name = 'CompanyC', Country = 'ECU', Revenue = 25000
```

挿入されたデータを Revenue 順に表示します。

#### SQL

```
SELECT * FROM Sample.Company ORDER BY Revenue DESC
```

会社名	収益	国
CompanyC	25000	ECU
CompanyA	10000	BEL
CompanyB	0	CAN

完了したら、テーブルを削除します。

## SQL

```
DROP TABLE Sample.Company
```

## ストリーム・データをテーブルに挿入する

以下の例は、埋め込み SQL を使用して、ストリーム・フィールドに挿入できるさまざまなデータ値タイプを表示します。任意のテーブルで、文字列リテラル、または文字列リテラルが含まれるホスト変数を挿入できます。以下に例を示します。

### ObjectScript

```
set literal="Technique 1"
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:literal))
```

シャード化されていないテーブルのストリーム・オブジェクトに、オブジェクト参照 (OREF) を挿入することもできます。InterSystems IRIS は、このオブジェクトを開いて、そのコンテンツを新しいストリーム・フィールドにコピーします。例えば以下のようになります。

### ObjectScript

```
set oref=##class(%Stream.GlobalCharacter).%New()
do oref.Write("Technique non-shard 1")

//do the insert; use an actual OREF
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:oref))
```

あるいは、ストリーム・オブジェクトに OREF の文字列バージョンを挿入することもできます。

### ObjectScript

```
set oref=##class(%Stream.GlobalCharacter).%New()
do oref.Write("Technique non-shard 2")

//next line converts OREF to a string OREF
set string=oref_" "

//do the insert
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:string))
```

シャード・テーブルの場合は、`^IRIS.Stream.Shard` グローバルに保存されている一時ストリーム・オブジェクトを使用して、オブジェクト ID (OID) を挿入できます。

### ObjectScript

```
set clob=##class(%Stream.GlobalCharacter).%New("Shard")
do clob.Write("Technique Sharded Table 1")
set sc=clob.%Save() // Handle $$$ISERR(sc)
set ClobOid=clob.%Oid()

&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:ClobOid))
```

適切に定義されていないストリーム値を挿入すると、SQLCODE -412 エラーが返されます。

詳細および例は、“[ストリーム・データ・フィールドへのデータの挿入](#)”を参照してください。

## 埋め込み SQL を使用してデータを挿入する

この埋め込み SQL の例では、ホスト変数配列を使用して 3 つの列値を持つ行を挿入しています。配列要素は列順に番号付けされます。指定される配列値は、2 番目の要素（この場合、company(2)）で始まる必要があります。最初の配列要素は、RowID 列に対応します。これは自動的に提供され、定義できません。

### ObjectScript

```
SET company(2)="Company1"
SET company(3)=15000
SET company(4)="JPN"
&sql(INSERT INTO Sample.Company VALUES :company())
```

この埋め込み SQL の例では、実行時に INSERT に値の配列を渡すために、最後の添え字が未定義の動的なローカル配列を使用します。

### ObjectScript

```
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Employee VALUES :emp('profile',))
WRITE !,"SQL Error code: ",SQLCODE," Row Count: ",%ROWCOUNT
```

前の文により、挿入された "Employee" 行の各列は次のように設定されます。ここで "col" は Sample.Employee テーブルの列番号です。

```
emp("profile",col)
```

## ダイナミック SQL を使用してデータを挿入する

このクラス・メソッドでは、ダイナミック SQL を使用し、メソッドに渡された引数に基づいてテーブルに値を挿入します。

### Class Member

```
ClassMethod DynamicSQLInsert(name As %String, revenue As %Integer, country As %String)
{
    set sqltext = "INSERT INTO Sample.Company (Name,Revenue,Country) VALUES (?,?,?)"

    set tStatement = ##class(%SQL.Statement).%New(0,"Sample")
    set qStatus = tStatement.%Prepare(sqltext)
    if qStatus'=1 {write "%Prepare failed:" DO $System.Status.DisplayError(qStatus) quit}
    set rtn = tStatement.%Execute(name,revenue,country)
    if rtn.%SQLCODE=0 {
        write !,"Insert succeeded"
        write !,"Row count=",rtn.%ROWCOUNT
        write !,"Row ID=",rtn.%ROWID }
    elseif rtn.%SQLCODE=-119 {
        write !,"Duplicate record not written",!,rtn.%Message quit }
    else { write !,"Insert failed, SQLCODE=",rtn.%SQLCODE }
}
```

## SELECT クエリを使用して別のテーブルからデータを挿入する

この例では、INSERT を SELECT と共に使用して、別のテーブルから抽出したデータをテーブルに移入する方法を示します。この例では、Name、DOB、および Age 列を含む Sample.Person テーブルを既に定義していることを前提としています。このようなテーブルは、GitHub (<https://github.com/interSystems/Samples-Data>) からダウンロードできます。ダウンロードの手順は、「[インターシステムズ製品で使用するサンプルのダウンロード](#)」を参照してください。

MyStudents というテーブルを作成します。このテーブルには、名前と生年月日の列（どちらも指定します）、および年齢の列（生年月日の列から計算されます）が含まれます。

## SQL

```
CREATE TABLE MyStudents (
  StudentName VARCHAR(32),
  StudentDOB DATE,
  StudentAge INTEGER COMPUTECODE {set {StudentAge} =
    $piece(($piece($horolog,"",1)-{StudentDOB})/365,".",1)}
    CALCULATED)
```

Sample.Person テーブルからの学生データを MyStudents テーブルに挿入します。21 歳以下の人を選択する SELECT クエリを使用します。以下の 2 つのクエリのいずれかを使用できます。2 つのテーブルの列順は一致しており、格納されるのは 2 つの列のみであるため、列名は省略できます。

## SQL

```
INSERT INTO MyStudents (StudentName,StudentDOB)
  SELECT Name,DOB
  FROM Sample.Person WHERE Age <= 21
```

## SQL

```
INSERT INTO MyStudents
  SELECT Name,DOB
  FROM Sample.Person WHERE Age <= 21
```

年齢順に結果を表示します。

## SQL

```
SELECT * FROM MyStudents ORDER BY StudentAge
```

完了したら、テーブルを削除します。

## SQL

```
DROP TABLE MyStudents
```

INSERT を SELECT と共に使用して、既存のテーブルから重複テーブルを作成することもできます。この操作を使用して、既存のデータを再定義されたテーブルにコピーでき、元のテーブルでは有効にならなかったであろう将来の列データ値がこのテーブルで受け入れられます。詳細は、["重複テーブルにデータをコピーする"](#) を参照してください。

## 互換性

INSERT を使用して、Microsoft Access を使用する InterSystems IRIS テーブルにデータを追加するには、テーブルの [RowID 列](#) をプライベートとしてマークするか、1 つ以上の追加の列に一意のインデックスを定義します。

## セキュリティおよび特権

### テーブルレベルの特権

テーブルに 1 つ以上のデータ行を挿入するには、ユーザ（または指定したユーザ）に、そのテーブルに対するテーブルレベルの特権か列レベルの特権が必要です。

- ・ テーブルにデータを挿入する際は、そのテーブルに対する INSERT 特権が必要です。
- ・ SELECT クエリを使用して別のテーブルからデータを挿入する場合、ユーザはそのテーブルに対する SELECT 特権を持っている必要があります。

テーブルの所有者（作成者）にはそのテーブルに対するすべての特権が自動的に付与されます。所有者でない場合は、そのテーブルに対する特権が付与される必要があります。これを実行しないと、SQLCODE -99 エラーが返されます。



適切な特権があるかどうかを確認するには、[%CHECKPRIV](#) を使用します。ユーザにテーブルの特権を割り当てるには、[GRANT](#) を使用します。詳細は、“[特権](#)”を参照してください。

シャード・テーブルに挿入するには、ターゲット・テーブルに対する INSERT 特権が必要です。これらの特権がない場合は、SQLCODE -253 エラーが返されます。

テーブルレベルの特権は、テーブルの全列に対して列レベルの特権を持っていることと同等ですが、まったく同じではありません。

## 列レベルの特権

テーブルレベルの INSERT 特権がない場合に、指定された値を列に挿入するには、その列に対する列レベルの INSERT 特権が必要です。INSERT コマンドで指定した値は、INSERT 特権を持つ列のみに挿入されます。

指定した列に対して列レベルの INSERT 特権がない場合、InterSystems SQL は、列の既定値 (定義されている場合) または NULL (既定値が定義されていない場合) を挿入します。既定値のない、NOT NULL と定義された列に対してユーザが INSERT 特権を持たない場合、InterSystems IRIS は準備時に SQLCODE -99 (特権違反) エラーを発行します。

INSERT コマンドで SELECT サブクエリの WHERE 節内に列を指定する場合、以下の特権が必要です。

- ・ データ挿入列ではない場合、それらの列に対する SELECT 特権。
- ・ これらの列が結果セットに含まれる場合には、これらの列に対する SELECT 特権と UPDATE 特権の両方。

プロパティが [ReadOnly](#) として定義されている場合、対応するテーブル列も ReadOnly として定義されます。ReadOnly 列への値の割り当ては、[InitialExpression](#) または [SqlComputed](#) を使用した場合のみ可能です。列レベルの ReadOnly (SELECT または REFERENCES) 特権を持っている列に値を挿入しようとすると、SQLCODE -138 エラーが発生します。

適切な特権があるかどうかを確認するには、[%CHECKPRIV](#) を使用します。ユーザに列レベルの特権を割り当てるには、[GRANT](#) を使用します。詳細は、“[特権](#)”を参照してください。

## 行レベル・セキュリティ

InterSystems IRIS の行レベル・セキュリティにより、行セキュリティの定義で行への次のアクセスが許可されていない場合でも、INSERT を使用して行を追加できるようになります。INSERT により行への今後の SELECT アクセスが妨げられないようにするには、WITH CHECK OPTION を持つビューで INSERT を実行します。詳細は、“[CREATE VIEW](#)”を参照してください。

## Fast Insert

JDBC または ODBC を使用してテーブルに行を挿入すると、既定で InterSystems IRIS は自動的に非常に効率的な Fast Insert 操作を実行します。Fast Insert は、サーバからクライアントに挿入されるデータの正規化とフォーマットを実行します。これでサーバは、サーバ上で操作を行うことなく、テーブルのデータの行全体をグローバルに直接設定できます。これにより、これらのタスクをサーバからクライアントにオフロードし、INSERT パフォーマンスを大幅に向上させることができます。クライアントがデータのフォーマットのタスクを担っているため、クライアント環境で予期できない使用量の増加が生じる場合があります。これが問題になる場合は、FeatureOption プロパティを使用して、Fast Insert を無効にできます。

Fast Insert はサーバとクライアントの両方でサポートされる必要があります。Fast Insert をクライアントで有効または無効にするには、以下のように、クラス・インスタンスの定義で FeatureOption プロパティを使用します。

```
Properties p = new Properties();
p.setProperty("FeatureOption","3"); // 2 is fast Insert, 1 is fast Select, 3 is both
```

Fast Insert がアクティブの場合、クエリ・キャッシュを使用して実行される INSERT は、Fast Insert を使用して実行されません。クエリ・キャッシュを生成したこの最初の INSERT は Fast Insert を使用して実行されません。これにより、最初の挿入と、後続のクエリ・キャッシュを使用して実行される Fast Insert のパフォーマンスを比較できます。Fast Insert がサポートされていない (以下のような理由により) 場合、通常の INSERT が実行されます。

Fast Insert はテーブル上で実行される必要があります。更新可能なビューでは実行できません。Fast Insert は、テーブルが以下の特性のいずれかを有する場合、実行されません。

- ・ テーブルが埋め込み (入れ子) ストレージ構造 (%SerialObject) を使用している。
- ・ テーブルがリンク・テーブルである。
- ・ テーブルが子テーブルである。
- ・ テーブルに、明示的に定義されたマルチフィールド IDKEY インデックスがある。
- ・ テーブルに SERIAL (%Counter)、AUTO INCREMENT、または %RowVersion フィールドがある。
- ・ テーブルに VALUELIST パラメータが定義されたプロパティ (フィールド) がある。
- ・ テーブルに INSERT トリガが定義されている。
- ・ テーブルがフィールド値の LogicalToStorage 変換を実行する。
- ・ テーブルがシャード・マスタ・テーブルである。

Fast Insert は、INSERT 文が以下の特性のいずれかを有する場合、実行できません。

- ・ ストリーム・フィールド (データ型 %Stream.GlobalCharacter または %Stream.GlobalBinary)、コレクション・フィールド (リストまたは配列)、または ReadOnly フィールドを指定する。このようなタイプのフィールドは、テーブルに存在することはできますが、INSERT で指定することはできません。
- ・ リテラル置換を抑制する二重括弧で囲まれたリテラル値を指定する。例えば (('A')) などです。
- ・ 日付値を省略する {ts} タイムスタンプ値を指定する。
- ・ DEFAULT VALUES 節を含む。

データベース・ドライバにより生成される SQL 文の監査イベントでは、Fast Insert インタフェースを使用する INSERT 文には SQL fastINSERT Statement の記述があります。Fast Insert インタフェースが使用されている場合、監査イベントにパラメータ・データは含まれませんが、メッセージ " fastInsert " が含まれます。

## ODBC データ型の処理

ODBC を使用した Fast Insert を使用する場合、整数または \_int64 データ型を使用して、フィールド TIMESTAMP または POSIX タイプのフィールドを設定できます。指定された数値は \$HOROLOG 値として扱われます。

倍精度値に変換される文字列値が検証され、その文字列が数値を表すことが保証されます。この検証では、"INF"、"infinity"、および "NaN" の値 (+ または - 記号が先頭に付く場合もあります) も常に受け入れます。

数値から整数に変換される値は、四捨五入ではなく、切り捨てられます。

## トランザクションの考慮事項

### トランザクションのアトミック性設定

既定では、INSERT、UPDATE、DELETE、および TRUNCATE TABLE はアトミック処理として実行されます。INSERT は、正常に完了するか、すべての操作がロールバックされるかのいずれかです。指定した行のいずれかを挿入できない場合、指定した行は 1 行も挿入できずにデータベースは INSERT を発行する前の状態に戻ります。

現在のプロセスに対するこの既定は、SET TRANSACTION %COMMITMODE を呼び出すことによって SQL 内で変更できます。現在のプロセスに対するこの既定は、次の構文を使用して SetOption() メソッドを呼び出すことによって ObjectScript 内で変更できます。

```
SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)
```

以下の intval 整数オプションを使用できます。

- ・ 1 または IMPLICIT (自動コミットがオン – 既定) – INSERT ごとに個別のトランザクションが構成されます。
- ・ 2 または EXPLICIT (自動コミットがオフ) – 進行中のトランザクションがない場合は、INSERT コマンドによってトランザクションは自動的に開始されます。ただし、COMMIT または ROLLBACK で明示的にトランザクションを終了する必要があります。EXPLICIT モードでは、トランザクションあたりのデータベース操作の数は、ユーザ定義です。
- ・ 0 または NONE (自動トランザクションなし) – INSERT を呼び出してもトランザクションは開始されません。INSERT 操作の失敗により、行の一部が挿入されたり挿入されなかったりすることで、データベースが整合性のない状態になる可能性があります。このモードでトランザクションのサポートを提供するには、START TRANSACTION を使用してトランザクションを開始し、COMMIT または ROLLBACK を使用してトランザクションを終了する必要があります。

シャード・テーブルは常に、自動トランザクションなしのモードに設定されます。つまり、シャード・テーブルに対する挿入、更新、および削除はすべて、トランザクションの範囲外で実行されます。

現在のプロセスのアトミック性設定を確認するには、以下の ObjectScript の例のように、GetOption("AutoCommit") メソッドを使用します。

### ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }
```

## トランザクションのロックしきい値の変更

**%NOLOCK** キーワードを指定しない場合、INSERT、UPDATE、および DELETE 操作時に自動的にレコードに標準のロックがかかります。影響を受ける各レコード (行) は、現在のトランザクションが継続している間はロックされます。

既定のロックしきい値は、テーブルごとに 1000 ロックです。トランザクションの間にテーブルから 1000 件を超えるレコードを挿入すると、ロックのしきい値に到達し、InterSystems IRIS は自動的にロック・レベルをレコード・ロックからテーブル・ロックに上げます。これによってトランザクション時に、ロック・テーブルをオーバーフローすることなく、大規模な挿入を実行できます。

InterSystems IRIS は、以下のロック・エスカレーション策のいずれかを適用します。

- ・ “E” タイプのロック・エスカレーション – InterSystems IRIS は、以下が当てはまる場合にこのロック・エスカレーションを使用します。
  1. テーブルのクラスで **%Storage.Persistent** を使用している。これは、管理ポータル の SQL スキーマ表示の [\[カタログの詳細\]](#) から確認できます。
  2. クラスで、IDKey インデックスが指定されていないか、単一プロパティの IDKey インデックスが指定されている。

“E” タイプのロック・エスカレーションの詳細は、“**LOCK**” を参照してください。

- ・ 従来の SQL ロック・エスカレーション – このロック・エスカレーションは、クラスにマルチプロパティの IDKey インデックスがある場合に発生する可能性があります。この場合は、%Save ごとにロック・カウンタがインクリメントされます。つまり、トランザクション内の単一オブジェクトを 1001 回保存すると、InterSystems IRIS はロックのエスカレーションを試みます。

どちらのロック・エスカレーション策の場合も、\$SYSTEM.SQL.Util.GetOption("LockThreshold") メソッドを使用して、現在のシステム全体用ロックしきい値を決定できます。既定値は 1000 です。以下のオプションのいずれかを使用してシステム全体のロックしきい値を構成できます。

- ・ \$SYSTEM.SQL.Util.SetOption("LockThreshold") メソッドを呼び出します。

- ・ 管理ポータルで、[システム管理]、[構成]、[SQL とオブジェクトの設定]、[SQL] の順に選択します。[ロック・エスカレーションしきい値] の現在の設定を表示して編集します。既定は 1000 ロックです。この設定を変更すると、変更後に開始される新しいプロセスは、新しい設定になります。

ロックしきい値を変更するには、%Admin リソース管理に対する USE 許可を持っている必要があります。InterSystems IRIS は、ロックしきい値の変更を現在のプロセスすべてに即座に適用します。

結果として、自動ロック・エスカレーションでは、デッドロックの状況が起こる可能性があります。つまり、テーブル・ロックへのエスカレーションを試みたときに、テーブル内のレコード・ロックを保持する別プロセスとの競合が起こる可能性があります。これを回避するには、いくつかの方法が考えられます。

1. ロック・エスカレーションがトランザクション内で起こる可能性が低くなるように、ロック・エスカレーションのしきい値を上げる。
2. ロック・エスカレーションが即座に起こるように、ロック・エスカレーションのしきい値を大幅に下げる。これにより、別プロセスが同一テーブル内のレコードをロックする機会が少なくなります。
3. トランザクションが継続している間はテーブル・ロックを適用し、レコード・ロックは実行しない。これは、LOCK TABLE、UNLOCK TABLE (テーブル・ロックがトランザクションの終了まで持続するよう、IMMEDIATE キーワードはなし) の順に指定することで、トランザクション開始時に実行できます。その後、%NOLOCK オプションを使用して挿入を実行します。

自動ロック・エスカレーションは、ロック・テーブルのオーバーフローを防ぐことを目的としています。ただし、大量の挿入などを実行したために <LOCKTABLEFULL> エラーが発生した場合は、INSERT によって SQLCODE -110 エラーが発行されます。

トランザクションでのロックの詳細は、“[トランザクション処理](#)” を参照してください。

## 子テーブルの挿入

子テーブルへの INSERT 操作時には、親テーブル内の対応する行に共有ロックがかかります。この行は、子テーブルの挿入中はロックされます。その後、ロックは解除されます (トランザクションの終了までロック状態が継続することはありません)。これにより、参照される親の行がこの挿入操作の間に変更されることがなくなります。

## 詳細

### 重複テーブルにデータをコピーする

列の順序が一致し、互換性のあるデータ型であれば、SELECT \* と組み合わせて INSERT を使用し、あるテーブルから重複テーブルへデータをコピーできます。この操作を使用して、既存のデータを再定義されたテーブルにコピーし、元のテーブルでは有効にならなかったであろう将来の列データ値がこのテーブルで受け入れられます。以下にサンプル構文を示します。

#### SQL

```
INSERT INTO Sample.DupTable SELECT * FROM Sample.SrcTable
```

列名が一致する必要はありませんが、コピー元のテーブルとコピー先のテーブルのデータは以下の要件を満たす必要があります。

- ・ コピー元テーブルの値のデータ型は、コピー先テーブルの列のデータ型と互換性がある必要があります。例えば、INTEGER 値は VARCHAR に変換できるため、INTEGER 列の整数データを VARCHAR 列に挿入することは可能です。データ値に互換性がない場合、INSERT は SQLCODE -104 エラーで失敗します。挿入するデータを挿入先のデータ型に明示的に変換するには、[CONVERT](#) 関数を使用します。
- ・ コピー元テーブルの値のデータ型の長さは、コピー先テーブルの列の長さと適合する必要があります。定義されている列データの長さが相互に一致する必要はありません。実際のデータと一致する必要があるだけです。例えば、SrcTable は FullName VARCHAR (60) 列を持ち、DupTable は対応する PersonName VARCHAR (40) 列を持

つとします。40 文字を超える既存の FullName がない場合、INSERT は成功します。いずれかの FullName が 40 文字を超える場合、INSERT は SQLCODE -104 エラーで失敗します。

- ・ 2 つのテーブルの列順には互換性がある必要があり、互換性がない場合、INSERT コマンドは SQLCODE -64 エラーで失敗します。DDL CREATE TABLE は、定義された順序で列をリストします。テーブルを定義する永続クラスはアルファベット順に列をリストします。
- ・ テーブルには互換性のある列数が必要ですが、コピー先テーブルにはコピーされた列以外の列を含めることができます。例えば、SrcTable は、列 FullName VARCHAR(60) および Age INTEGER を持つことができ、DupTable は PersonName VARCHAR(60)、Years INTEGER、および ShoeSize INTEGER を持つことができます。
- ・ コピー元テーブルとコピー先テーブルのいずれかでパブリック RowID が定義されている場合、データのコピーは以下の表に示すように制限されます。

コピー元テーブル	コピー先テーブル	コピー操作の動作
プライベート RowID	プライベート RowID	INSERT SELECT を SELECT * と共に使用して、データを重複テーブルにコピーできます。
パブリック RowID	パブリック RowID	INSERT SELECT を使用して、データを重複テーブルにコピーすることはできません。SQLCODE -111 エラーが生成されます。
プライベート RowID	パブリック RowID	INSERT SELECT を使用して、データを重複テーブルにコピーすることはできません。SQLCODE -111 エラーが生成されます。
パブリック RowID	プライベート RowID	<p>INSERT SELECT を SELECT * と共に使用して、データを重複テーブルにコピーすることはできません。一方の選択リストに RowID が存在していて選択リストが非互換になるため、SQLCODE -64 エラーが生成されます。INSERT SELECT をすべての列名のリスト (RowID は含まない) と共に使用して、重複テーブルにデータをコピーできます。ただし、コピー元に外部キーのパブリック RowID がある場合、外部キー・リレーションシップはコピー先テーブルでは保持されません。コピー先は新しいシステム生成 RowID を受け取ります。</p> <p>コピー元テーブルに外部キーのパブリック RowID があり、コピー先テーブルに同じ外部キー・リレーションシップを設定したい場合は、CREATE TABLE で %CLASSPARAMETER ALLOWIDENTITYINSERT=1 を使用してコピー先テーブルを定義する必要があります。テーブルが ALLOWIDENTITYINSERT=1 として定義されている場合、この設定を SetOption("IdentityInsert") メソッドで変更することはできません。</p>

DDL CREATE TABLE は既定で RowID をプライベートとして定義します。テーブルを定義する永続クラスは、既定で RowID をパブリックとして定義します。これをプライベートにするには、永続クラスを定義する際に、SqlRowIdPrivate クラス・キーワードを指定する必要があります。ただし、外部キーで参照できるのはパブリック RowID を持つテーブルだけです。

コピー元またはコピー先のテーブルの永続クラスで Final キーワードが定義されている場合、このキーワードは重複テーブルへのデータのコピーに影響を与えません。



## %SerialObject プロパティにデータを挿入する

[%SerialObject](#) にデータを挿入する際は、埋め込み [%SerialObject](#) を参照するテーブル (永続クラス) に挿入する必要があります。[%SerialObject](#) に直接挿入することはできません。例えば、永続クラスにプロパティ `PAddress` があり、これはプロパティ `Street`、`City`、`Country` がこの順番に含まれるシリアル・オブジェクトを参照するものとします。このプロパティの値を挿入する方法は以下のとおりです。

- 参照するフィールドを使用して、複数の [%SerialObject](#) プロパティの値を [%List](#) 構造として挿入する。以下に例を示します。

### SQL

```
INSERT INTO MyTable SET PAddress=$LISTBUILD('123 Main St.','Newtown','USA')
```

### SQL

```
INSERT INTO MyTable (PAddress) VALUES ($LISTBUILD('123 Main St.','Newtown','USA'))
```

[%List](#) には、シリアル・オブジェクト (またはプレースホルダとしてのコンマ) のプロパティの値が、このシリアル・オブジェクトで指定されている順序で含まれる必要があります。

このタイプの挿入では、[%SerialObject](#) プロパティ値の検証は実行されない場合があります。したがって、[%List](#) 構造を使用して [%SerialObject](#) プロパティを挿入したら、`$SYSTEM.SQL.Schema.ValidateTable()` メソッドを使用して[テーブル・データの検証](#)を実行してください。

- アンダースコア構文を使用して、個々の [%SerialObject](#) プロパティの値を任意の順序で挿入する。以下に例を示します。

### SQL

```
INSERT INTO MyTable SET PAddress_City='Newtown',PAddress_Street='123 Main St.',PAddress_Country='USA'
```

指定されていないシリアル・オブジェクト・プロパティは、既定の `NULL` になります。

このタイプの挿入では、[%SerialObject](#) プロパティ値の検証が実行されます。

## 関連項目

- [INSERT OR UPDATE](#)
- [UPDATE](#)
- [DELETE](#)
- [CREATE TABLE](#)
- [JOIN](#)
- [SELECT](#)
- [VALUES](#)
- [データベースの変更](#)
- [テーブルの定義](#)
- [ビューの定義](#)
- [トランザクション処理](#)
- [SQL およびオブジェクトの設定ページ](#)
- [SQLCODE エラー・メッセージ](#)

# INSERT OR UPDATE (SQL)

テーブルで新規の行を追加するか、既存の行を更新します。

## 構文

### 単一行の挿入または更新

```
INSERT OR UPDATE table (column, column2, ...) VALUES (value, value2, ...)
INSERT OR UPDATE table VALUES (value, value2, ...)
INSERT OR UPDATE table SET column = value, column2 = value2, ...
INSERT OR UPDATE table DEFAULT VALUES
INSERT OR UPDATE table VALUES :array()
```

### 複数行の挿入または更新

```
INSERT OR UPDATE table query
INSERT OR UPDATE table (column, column2, ...) query
```

### 挿入または更新のオプション

```
INSERT OR UPDATE INTO table ...
INSERT OR UPDATE %keyword [INTO] table ...
```

## 説明

INSERT OR UPDATE コマンドは、INSERT コマンドを拡張したもので、以下のような相違点があります。

- ・ 挿入される行が存在しない場合、INSERT OR UPDATE は [INSERT](#) 操作を実行します。
- ・ 挿入する行が既に存在する場合、INSERT OR UPDATE は [UPDATE](#) 操作を実行し、行を指定された列値で更新します。指定されたデータ値が既存のデータと同一である場合でも、更新は行われます。

既存の行とは、挿入される値が一意制約を含む列に既に存在する行です。詳細は、[“一意性チェック”](#) を参照してください。

INSERT OR UPDATE は、同じ構文を使用し、一般に INSERT 文と同じ機能および制限があります。このページでは、INSERT OR UPDATE の特別な考慮事項について説明します。特に明記されていない限り、詳細は [“INSERT”](#) を参照してください。

### 単一行の挿入または更新

- ・ INSERT OR UPDATE [table](#) ([column](#), column2, ...) VALUES ([value](#), value2, ...) 指定したテーブルの列に、値の行を挿入するか、この値の行を更新します。VALUES 節の値は、列のリスト内の列の名前と位置的に対応する必要があります。単一行の挿入または更新では、%ROWCOUNT 変数が 1 に設定され、%ROWID 変数が挿入される行または更新される行に設定されます。

この文は、まず新しいデータの行を Sample.Records テーブルに挿入しようとします。RecordID 列が [UNIQUE 制約](#) を適用し、挿入される RecordID が既に存在する場合、INSERT OR UPDATE は代わりにその既存の行を更新します。

### SQL

```
INSERT OR UPDATE Sample.Records (RecordID,StatusDate,Status) VALUES (105,'05/12/22','Purged')
```

例： [テーブルに行を挿入するか、これを更新する](#)

- ・ INSERT OR UPDATE table VALUES (value, value2, ...) は、テーブルの値の行を列番号順に挿入するか、これを更新します。データ値は、定義されている列リストと位置的に対応する必要があります。指定可能なすべてのテーブ



ル列に値を指定する必要があります。定義済みの既定値を使用することはできませんが、空の文字列を値として指定することはできます。RowID 列は指定可能ではないため、RowID 値は VALUES リストに含めないでください。

以下の文は、まず、4 つの値を持つ行を順に `Sample.Address` テーブルに挿入しようとします。この列の組み合わせに一意制約があり、このキーの値が既にテーブルで定義されている場合、INSERT OR UPDATE は代わりに既存の行を更新します。

## SQL

```
INSERT OR UPDATE Sample.Address VALUES ('22 Main St.', 'Anytown', 'PA', '65342')
```

- INSERT OR UPDATE table SET column = value, column2 = value2, ... は、特定の列の値を明示的に設定することにより、値の行を挿入または更新します。

以下の文は、INSERT OR UPDATE table (column, column2, ...) VALUES (value, value2, ...) 構文と同じ操作を実行します。

## SQL

```
INSERT OR UPDATE Sample.Records SET RecordID=105, StatusDate='05/12/22', Status='Purged'
```

- INSERT OR UPDATE table DEFAULT VALUES は、既定の列値のみを含む行を挿入または更新します。
  - 定義済みの既定値を持つ列は、その値に設定されます。
  - 定義された既定値がない列は、NULL に設定されます。

以下の文は、既定の列値を持つ行を `Sample.Person` テーブルに挿入します。

## SQL

```
INSERT OR UPDATE Sample.Person DEFAULT VALUES
```

- INSERT OR UPDATE table VALUES :array() は、ホスト変数として指定された配列の値を、テーブルの列に挿入するか、これを更新します。この構文は、[埋め込み SQL](#) でのみ使用できます。この配列の値は、暗黙的に、列番号順に行の列に対応している必要があります。指定可能な各列に値を指定する必要があります。列順を使用する INSERT OR UPDATE では、定義された列の既定値を取ることはできません。

このクラス・メソッドは、埋め込み SQL を使用して、配列を `Sample.FullName` テーブルに挿入するか、これを更新します。`myarray(1)` は、[RowID 列](#)用に予約されているため、指定されません。

## Class Member

```
ClassMethod EmbeddedSQLInsertOrUpdateHostVarArray()
{
    set myarray(2)="Juanita"
    set myarray(3)="Pybus"
    &sql(INSERT OR UPDATE Sample.FullName VALUES :myarray())
    if SQLCODE '= 0 {
        write !, "Insert or update failed, SQLCODE= ", SQLCODE, !, %msg
        quit
    }
    write !, "Insert or update succeeded" quit
}
```

ホスト変数と配列については、“[添え字付き配列としてのホスト変数](#)”を参照してください。

## 複数行の挿入または更新

- INSERT OR UPDATE table [query](#) は、SELECT クエリの結果セットから取得したデータの行を挿入または更新します。結果セット内の列は、テーブル内の列と一致する必要があります。INSERT OR UPDATE を SELECT と共に使用することで、別のテーブルから抽出した既存のデータをテーブルに移入できます。

この文は、Sample.Customer テーブルの Name 行を Sample.Person テーブルに挿入するか、Sample.Person の既存の行に対応する Sample.Customer の値で更新します。

## SQL

```
INSERT OR UPDATE Sample.Person SELECT Name FROM Sample.Customer
```

- INSERT OR UPDATE table (column, column2, ...) query は、クエリ結果セットのデータ行を、指定された列に挿入するか、この行を更新します。

以下の文は、Sample.Person の Name および DOB 列のクエリ結果セットのデータを、Sample.Kids テーブルの一致する列に挿入するか、このデータを更新します。

## SQL

```
INSERT OR UPDATE Sample.Kids (Name,DOB) SELECT Name,DOB FROM Sample.Person WHERE Age <= 18
```

## 挿入または更新のオプション

- INSERT OR UPDATE INTO table ... は、オプションの INTO キーワードを指定します。
- INSERT OR UPDATE %keyword [INTO] table ... は、1 つまたは複数の %keyword オプションを空白で区切って設定します。有効なオプションは、%NOCHECK、%NOFPLAN、%NOINDEX、%NOJOURN、%NOLOCK、%NOTRIGGER、%PROFILE、および %PROFILE\_ALL です。

注釈     %NOCHECK キーワードを使用すると一意の値のチェックが無効になるため、INSERT OR UPDATE %NOCHECK は常に挿入操作になり、INSERT と等価になります。

## 引数

### table

挿入を実行するテーブルまたはビューの名前。この引数はサブクエリでもかまいません。

### column

列名、または列名のコンマ区切りのリスト。後者の場合は、値のリストに対応した順序で指定します。省略した場合、値のリストは列番号順にすべての列に適用されます。

IDKEY 値は挿入はできますが、更新はできません。これらの制約の詳細は、“IDKEY 列値”を参照してください。

### value

column 内の対応する列のデータ値を指定する、VALUES 節で指定されるスカラ式またはコンマ区切りのスカラ式のリスト。指定した値の数が列数より少ない場合、SQLCODE -62 エラーが生成されます。指定した値の数が列数より多い場合、SQLCODE -116 エラーが生成されます。

INSERT OR UPDATE には INSERT と同じ値制約があります。詳細は、“INSERT” コマンドの “value” 引数を参照してください。

### array

ホスト変数として指定する値の動的なローカル配列。この値は、埋め込み SQL にのみ適用されます。

配列の最下位の添え字は指定しないでください。:myupdates()、:myupdates(5,)、および :myupdates(1,1,) はすべて、有効な指定になります。

## query

その結果セットが、column で指定された対応する列にデータ値を指定する **SELECT** クエリ。

SELECT クエリは 1 つ以上のテーブルから列データを抽出し、INSERT OR UPDATE コマンドはこの列データを含むテーブル内に、対応する新しい行を作成します。挿入されるデータがテーブル列に収まることのできる限り、対応する列は、さまざまな列名および列長を持つことができます。対応する列がデータ型および長さ検証チェックに合格しない場合、InterSystems SQL は SQLCODE -104 エラーを生成します。

SELECT と共に INSERT OR UPDATE を使用すると、**%ROWCOUNT** 変数は挿入または更新した行の数に設定されます (0 または正の整数)。

## %keyword

INSERT OR UPDATE 処理を構成するキーワード・オプション。キーワード・オプションは順不同で指定できます。複数のキーワード・オプションは、スペースで区切ります。

以下のキーワードを指定できます。

- ・ **%NOCHECK** — 一意の値のチェックおよび外部キーの参照整合性チェックを無効にします。INSERT OR UPDATE の更新操作を無効にします。
- ・ **%NOFPLAN** — この操作の凍結プランを無視し、新しいクエリ・プランを生成します。
- ・ **%NOINDEX** — INSERT OR UPDATE 処理の際にインデックス・マップの設定を無効にします。
- ・ **%NOJOURN** — 挿入操作の間、ジャーナリングを抑制し、トランザクションをオフにします。
- ・ **%NOLOCK** — INSERT OR UPDATE の実行時に行のロックを無効にします。
- ・ **%NOTRIGGER** — INSERT OR UPDATE の処理時にベース・テーブル**挿入トリガ**はかけません。
- ・ **%PROFILE**、**%PROFILE\_ALL** — INSERT OR UPDATE 文のパフォーマンス分析統計 (SQLStats) を生成します。
  - **%PROFILE** はメイン・クエリ・モジュールに対して SQLStats を収集します。
  - **%PROFILE\_ALL** はメイン・クエリ・モジュールとそのすべてのサブクエリ・モジュールに対して SQLStats を収集します。

これらのキーワードの詳細は、“INSERT” コマンドの “**keyword**” 引数を参照してください。

## 例

### テーブルに行を挿入するか、これを更新する

この例では、新しいテーブル (SQLUser.CaveDwellers) を作成し、INSERT を使用してテーブルにデータを移入し、さらに INSERT OR UPDATE を使用して行を追加し、既存の行を更新します。

主キーとして指定された列 Num を持つテーブルを作成します。この制約により、列値は強制的に NULL ではなく一意となります。

## SQL

```
CREATE TABLE SQLUser.CaveDwellers (
  Num INTEGER PRIMARY KEY,
  CaveCluster CHAR(80) NOT NULL,
  Troglodyte CHAR(50) NOT NULL)
```

INSERT OR UPDATE 文を使用してテーブルに 3 つの行を挿入し、SELECT \* を使用してそのテーブル・データを表示します。これらの行は以前は存在していなかったため、INSERT OR UPDATE は、これらすべてに対して挿入操作を実行します。

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (1,'Bedrock','Flintstone,Fred')
```

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (2,'Bedrock','Flintstone,Wilma')
```

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES  
(3,'Bedrock','Flintstone,Pebbles')
```

## SQL

```
SELECT * FROM SQLUser.CaveDwellers
```

Num	CaveCluster	Troglodyte
1	Bedrock	Flintstone,Fred
2	Bedrock	Flintstone,Wilma
3	Bedrock	Flintstone,Pebbles

追加の 4 つのデータ行を挿入または更新します。

- 最初の 3 つの文では、主キー列 Num に挿入される値はまだテーブルに存在していないため、INSERT OR UPDATE は挿入操作を実行します。
- 最後の文では、Num 列値として 3 が既にテーブルに存在するため、INSERT OR UPDATE は更新操作を実行します。INSERT OR UPDATE は Troglodyte 列をその行の新しい値で更新します。

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (4,'Bedrock','Rubble,Barney')
```

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (5,'Bedrock','Rubble,Betty')
```

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (6,'Bedrock','Rubble,Bamm-Bamm')
```

## SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES  
(3,'Bedrock','Flintstone-Rubble,Pebbles')
```

## SQL

```
SELECT * FROM SQLUser.CaveDwellers
```

Num	CaveCluster	Troglodyte
1	Bedrock	Flintstone,Fred
2	Bedrock	Flintstone,Wilma
3	Bedrock	Flintstone-Rubble,Pebbles
4	Bedrock	Rubble,Barney
5	Bedrock	Rubble,Betty
6	Bedrock	Rubble,Bamm-Bamm

完了したら、テーブルを削除します。

## SQL

```
DROP TABLE SQLUser.CaveDwellers
```

## セキュリティおよび特権

INSERT OR UPDATE には INSERT 特権と UPDATE 特権の両方が必要です。これらの特権は、表レベルの特権または列レベルの特権のいずれかとして持つ必要があります。テーブルレベルの特権

- ・ ユーザは、実際に実行する操作に関係なく、指定されたテーブルに対して INSERT 特権と UPDATE 特権の両方を持っている必要があります。
- ・ SELECT クエリを使用して別のテーブルからデータを挿入または更新する場合、ユーザはそのテーブルに対する SELECT 特権を持っている必要があります。

テーブルの所有者(作成者)にはそのテーブルに対するすべての特権が自動的に付与されます。そうでない場合、ユーザにはテーブルに対する特権が付与される必要があります。これを実行しないと、SQLCODE -99 エラーが返されます。適切な特権があるかどうかを確認するには、[%CHECKPRIV](#) コマンドを使用します。テーブル特権を割り当てるには、[GRANT](#) コマンドを使用します。詳細は、“[特権](#)”を参照してください。

## 詳細

### 一意性チェック

INSERT OR UPDATE は、UNIQUE 列値を既存のデータ値に突き合わせることによって、行があるかどうかを判別します。UNIQUE 制約の違反が生じた場合、INSERT OR UPDATE は更新操作を実行します。UNIQUE 列値は、INSERT OR UPDATE で明示的に指定することも、列の既定値または計算された値の結果とすることもできます。

サブクラスであるテーブルに対して INSERT OR UPDATE が発行され、スーパー・クラスは既に UNIQUE 制約を満たしている場合、このコマンドは SQLCODE -119 で失敗します。ただし、スーパー・クラスであるテーブルに対して INSERT OR UPDATE が発行され、サブクラスは既に UNIQUE 制約を満たしている場合、更新は成功し、サブクラスとスーパー・クラスの両方に表示されているフィールドは更新されますが、サブクラスのみにあるフィールドは更新されません。

シャード・テーブルに対して INSERT OR UPDATE が実行されると、[シャード・キー](#) が UNIQUE KEY 制約と同じか、そのサブセットである場合、INSERT OR UPDATE は更新操作を実行します。

別の一意の値(シャード・キーではない)が見つかったために INSERT OR UPDATE が更新を試みると、一意制約違反により、SQLCODE -120 エラーでこのコマンドが失敗します。

### カウンタ列

INSERT OR UPDATE を実行する場合、InterSystems IRIS は最初に、操作は挿入であると想定します。したがって、[SERIAL \(%Library.Counter\)](#) 列に整数を指定するために使用される内部カウンタを 1 ずつインクリメントします。挿入では、これらのインクリメントされたカウンタ値を使用して、これらの列に整数値を割り当てます。ただし、InterSystems IRIS

は、更新操作を行う必要があると判断した場合には、INSERT OR UPDATE は既に内部カウンタをインクリメントしていますが、これらのインクリメントされた整数値をカウンタ列に割り当てません。このため、次の操作が挿入の場合、これらの列の整数シーケンスにギャップが生じます。詳細は、以下の例を参照してください。

1. 内部カウンタ値は 4 です。INSERT OR UPDATE は、内部カウンタをインクリメントし、次に行 5 を挿入します (内部カウンタ=5、SERIAL 列値=5)。
2. INSERT OR UPDATE は、内部カウンタをインクリメントし、次に既存の行に対して更新を実行する必要があると判断します (内部カウンタ=6、列カウンタの変更なし)。
3. INSERT OR UPDATE は、内部カウンタをインクリメントし、次に行を挿入します (内部カウンタ=7、SERIAL 列値=7)。

## IDENTITY 列と RowID 列

RowID 値の割り当てに対する INSERT OR UPDATE の影響は、IDENTITY 列があるかどうかによって異なります。

- ・ IDENTITY 列がテーブルに定義されていない場合、挿入操作によって、連続する次の整数値が ID (RowID) 列に自動的に割り当てられます。更新操作は後続の挿入に影響しません。したがって、INSERT OR UPDATE は INSERT と同じ挿入操作を実行します。
- ・ IDENTITY 列がテーブルに定義されている場合、INSERT OR UPDATE 操作が挿入になるか更新になるかを判断する前に、IDENTITY 列に整数を提供するために使用される内部カウンタが InterSystems IRIS で 1 つインクリメントされます。挿入操作では、インクリメントされたそのカウンタ値が IDENTITY 列に割り当てられます。ただし、INSERT OR UPDATE 操作が更新である必要があると InterSystems IRIS が判断した場合には、内部カウンタが既にインクリメントされていても、インクリメントされたそれらの整数値は割り当てられません。このため、次の INSERT OR UPDATE 操作が挿入である場合は、IDENTITY 列の整数シーケンスにギャップが生じます。RowID 列の値は IDENTITY 列の値から取得されるため、ID (RowID) の整数値の割り当てにギャップが生じます。

## IDKEY 列値

INSERT OR UPDATE を使用している場合、IDKEY 列値は更新できず、挿入のみ可能です。テーブルに IDKEY インデックスと、別の UNIQUE 制約がある場合、INSERT OR UPDATE はこれらの列を組み合わせ、挿入または更新のどちらを実行するかを決定します。一方のキーの制約が失敗すると、INSERT OR UPDATE は挿入ではなく更新が強制的に実行されます。ただし、指定された IDKEY 列の値が既存の IDKEY 列の値と一致しない場合、この更新により IDKEY の列の変更が試行されるので、更新は失敗し、SQLCODE -107 エラーが生成されます。

列 A、B、C、および D があるテーブルについて考えます。主キーが IDKEY である環境に主キー (A,B) があり、UNIQUE 制約が列 (C,D) に適用されています。

### SQL

```
SET OPTION PKEY_IS_IDKEY = TRUE
```

### SQL

```
CREATE TABLE ABCD (
  A INTEGER,
  B INTEGER,
  C INTEGER,
  D INTEGER,
  CONSTRAINT AB PRIMARY KEY (A,B),
  CONSTRAINT CD UNIQUE (C,D))
```

テーブルには次の 2 つのデータ行もあります。

### SQL

```
INSERT INTO ABCD SET A=1, B=1, C=2, D=2
```

## SQL

```
INSERT INTO ABCD SET A=1, B=2, C=3, D=4
```

以下の値の挿入を試みるとします。

## SQL

```
INSERT OR UPDATE ABCD (A,B,C,D) VALUES (2,2,3,4)
```

UNIQUE (C,D) 制約は失敗するため、この文は挿入を実行できません。代わりに、行 2 の更新を試みます。行 2 の IDKEY は (1,2) であるため、INSERT OR UPDATE 文は、列 A の値を 1 から 2 に変更しようとし、IDKEY の値を変更することはできないため、更新は SQLCODE -107 エラーで失敗します。

環境を既定の設定（主キーは IDKEY ではない）にリセットします。

## SQL

```
SET OPTION PKEY_IS_IDKEY = FALSE
```

## 関連項目

- ・ [CREATE TABLE](#)
- ・ [INSERT](#)
- ・ [UPDATE](#)
- ・ [データベースの変更](#)
- ・ [テーブルの定義](#)
- ・ [ビューの定義](#)
- ・ [トランザクション処理](#)
- ・ [SQLCODE エラー・メッセージ](#)



## %INTRANSACTION (SQL)

トランザクション状態を示します。

### 構文

```
%INTRANSACTION
%INTRANS
```

### 説明

%INTRANSACTION 文は、トランザクション状態を示す SQLCODE を設定します。

- ・ トランザクションが現在進行中である場合は SQLCODE=0
- ・ トランザクションが進行中でない場合は SQLCODE=100

%INTRANSACTION は、トランザクションが進行中である場合に SQLCODE=0 を返します。このトランザクションは、START TRANSACTION または SAVEPOINT によって開始された SQL トランザクションです。[TSTART](#) によって開始された ObjectScript トランザクションの場合もあります。

トランザクションの入れ子は、%INTRANSACTION に影響しません。SET TRANSACTION は、%INTRANSACTION に影響しません。

[\\$TLEVEL](#) を使用してトランザクション状態を判定することもできます。%INTRANSACTION は、トランザクションが進行中かどうかだけを示します。[\\$TLEVEL](#) は、トランザクションが進行中かどうかだけでなく、現在のトランザクション・レベル数も示します。

### 例

以下の埋め込み SQL の例は、%INTRANSACTION が SQLCODE をどのように設定するかを示しています。

#### ObjectScript

```
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "Before %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "SetTran %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(START TRANSACTION)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "StartTran %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(SAVEPOINT a)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "Savepoint %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(ROLLBACK TO SAVEPOINT a)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "Rollback to Savepoint %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(COMMIT)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "After Commit %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL
```

### 関連項目

- ・ [COMMIT ROLLBACK SAVEPOINT SET TRANSACTION START TRANSACTION \\$TLEVEL](#)
- ・ [トランザクション処理](#)

# JOIN (SQL)

2 つのテーブルのデータを基にテーブルを作成する SELECT 従属節です。

## 構文

### INNER JOIN

```
SELECT ... FROM table1 INNER JOIN table2 ON condition
SELECT ... FROM table1 INNER JOIN table2 USING (column, column2, ...)
SELECT ... FROM table1 JOIN table2 ...

SELECT ... FROM table1 NATURAL INNER JOIN table2
SELECT ... FROM table1 NATURAL JOIN table2
```

### LEFT OUTER JOIN

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON condition
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING (column, column2, ...)
SELECT ... FROM table1 LEFT JOIN table2 ...

SELECT ... FROM table1 NATURAL LEFT OUTER JOIN table2
SELECT ... FROM table1 NATURAL LEFT JOIN table2
```

### RIGHT OUTER JOIN

```
SELECT ... FROM table1 RIGHT OUTER JOIN table2 ON condition
SELECT ... FROM table1 RIGHT OUTER JOIN table2 USING (column, column2, ...)
SELECT ... FROM table1 RIGHT JOIN table2 ...

SELECT ... FROM table1 NATURAL RIGHT OUTER JOIN table2
SELECT ... FROM table1 NATURAL RIGHT JOIN table2
```

### FULL OUTER JOIN

```
SELECT ... FROM table1 FULL OUTER JOIN table2 ON condition
SELECT ... FROM table1 FULL JOIN table2 ON condition
```

### CROSS JOIN

```
SELECT ... FROM table1 CROSS JOIN table2
```

## 概要

JOIN 操作は、2 つのテーブルの一致する行を 1 つのテーブルに結合します。2 つのテーブルにまたがる行は、指定された 1 つ以上の列に同じ値がある場合、一致と見なされます。結合されたテーブルで返される行をさらに制限するには、追加の制限を指定します。

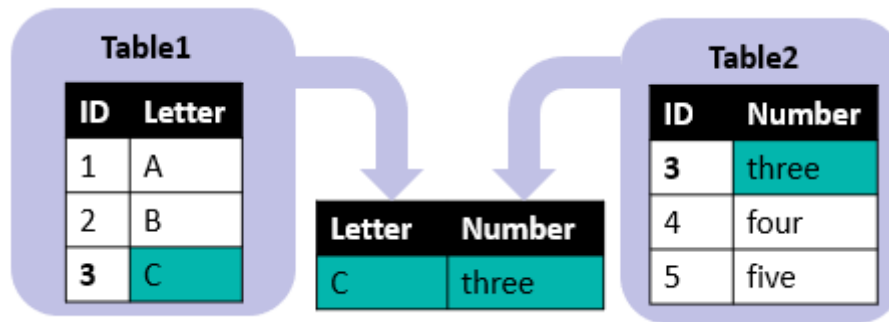
結合を使用して、テーブル間で関連するデータをリンクするレポートとクエリを生成します。SELECT クエリで、[FROM](#) 節の一部として JOIN 操作を指定します。クエリ内では、複数の内部結合および外部結合を任意の順序で指定できます。

### INNER JOIN

INNER JOIN は、最初のテーブルと 2 番目のテーブルの一致する行を返します。以下に例を示します。

#### SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
INNER JOIN Table2
ON Table1.ID = Table2.ID
```



- SELECT ... FROM `table1` INNER JOIN `table2` ON `condition` は、ON 節で指定された条件式に一致する `table1` と `table2` の行を返します。ON 節は、結合式内の任意の場所に指定できます。

このクエリは、`Sample.Employee` テーブル (E というエイリアス) と `Sample.Company` テーブル (C というエイリアス) のデータを結合し、従業員の名前とその会社を返します。両方のテーブルの `CompanyID` 列の値が一致している行についてのみ、`E.Name` と `C.Name` の値を返します。

### SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
INNER JOIN Sample.Company AS C
ON E.CompanyID = C.CompanyID
```

このクエリでは、さらに 20 歳を超える従業員の列のみを返すことで、データを制限しています。

### SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
INNER JOIN Sample.Company AS C
ON E.CompanyID = C.CompanyID AND E.Age > 20
```

### 例：内部結合と外部結合を使用したテーブル・データの結合

- SELECT ... FROM `table1` INNER JOIN `table2` USING (`column`, `column2`, ...) は、指定された列の値に一致する、`table1` と `table2` の行を返します。USING 節で指定される列は、両方のテーブルに存在する必要があります。この構文を使用することで、リンクされている列が両方のテーブルで同じ名前を持っている場合に、ON 構文より簡潔に等値条件が表されます。マルチ結合クエリでは、最初の結合に対してのみ USING 節を指定できます。

このクエリは、両方の列に結合できる `CompanyID` 列があるため、前の構文と同じ結合を実行します。

### SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
INNER JOIN Sample.Company AS C
USING (CompanyID)
```

### 例：2 つのテーブル間で同一の名前を持つ列での結合

- SELECT ... FROM `table1` JOIN `table2` ... は、前の INNER JOIN 構文と等価です。
- SELECT ... FROM `table1` NATURAL INNER JOIN `table2` は、2 つのテーブル間で同一の名前を持つすべての列に対して INNER JOIN を実行します。マルチ結合クエリでは、1 つの NATURAL 結合のみを指定でき、これが最初の結合である必要があります。

`CompanyID` が両方のテーブルに出現する唯一の列とすると、このクエリは、前の構文と同じ操作を実行します。これらのテーブルに同じ名前の列が複数含まれる場合、クエリは、マッチング結果を返す前に、これらの列も結合します。

## SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
NATURAL INNER JOIN Sample.Company AS C
```

例：2つのテーブル間で同一の名前を持つ列での結合

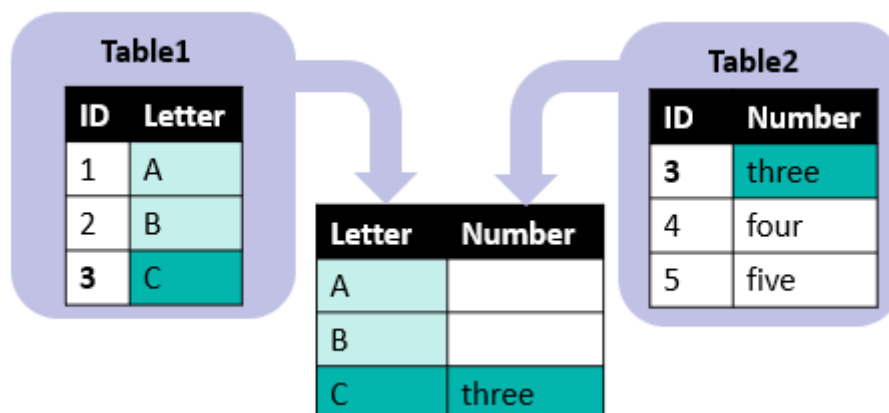
- SELECT ... FROM table1 NATURAL JOIN table2 は、NATURAL INNER JOIN 構文と等価です。

## LEFT OUTER JOIN

LEFT OUTER JOIN は、最初のテーブルのすべての行と、最初のテーブルの行と一致する 2 番目のテーブルの行を返します。結合されたテーブルでは、2 番目のテーブルからの列の一致しない行には、NULL 値が移入されます。以下に例を示します。

## SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
LEFT OUTER JOIN Table2
ON Table1.ID = Table2.ID
```



- SELECT ... FROM table1 LEFT OUTER JOIN table2 ON condition は、table1 のすべての行を返し、これらを、ON 節で指定された条件式を満たす table2 の行と結合します。

このクエリは、Sample.Employee テーブル (E というエイリアス) と Sample.Company テーブル (C というエイリアス) のデータを結合し、従業員の名前とその会社を返します。これは、すべての E.Name 値を返しますが、C.Name 値については、両方の CompanyID 列の値が一致している行についてのみ返します。一致しない行については、C.Name 値は NULL に設定されます。

## SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
LEFT OUTER JOIN Sample.Company AS C
ON E.CompanyID = C.CompanyID
```

注釈 あるいは、明示的な LEFT OUTER JOIN 構文を使用するのではなく、SELECT 文内の矢印構文 (→) で指定されるより簡潔な暗黙結合を使用することもできます。例えば、以下のクエリは前のクエリと同等です。

## SQL

```
SELECT Name, CompanyID->Name
FROM Sample.Employee
```

この構文は、Sample.Employee の CompanyID 列が、結合される Name 列を含む Sample.Company テーブルの行の ID を参照することを前提としています。暗黙結合の操作の詳細は、“[暗黙結合](#)”を参照してください。

例：

- [内部結合と外部結合を使用したテーブル・データの結合](#)
- [結合データに対する追加の制限の設定](#)

- ・ SELECT ... FROM table1 LEFT OUTER JOIN table2 USING (column, column2, ...) は、table1 のすべての行と、指定された列の値に一致する、table2 の行を返します。列は両方のテーブルに存在する必要があります。

このクエリは、両方の列に結合できる CompanyID 列があるため、前の構文と同じ結合を実行します。

## SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
LEFT OUTER JOIN Sample.Company AS C
USING (CompanyID)
```

例：[2 つのテーブル間で同一の名前を持つ列での結合](#)

- ・ SELECT ... FROM table1 LEFT JOIN table2 ... は、LEFT OUTER JOIN 構文と等価です。
- ・ SELECT ... FROM table1 NATURAL LEFT OUTER JOIN table2 は、2 つのテーブル間で同一の名前を持つすべての列に対して LEFT OUTER JOIN を実行します。式に複数の結合が含まれる場合は、最初に NATURAL 結合を指定します。NATURAL 結合では、同じ名前を持つ列はマージされません。

CompanyID が両方のテーブルに出現する唯一の列とすると、このクエリは、前の構文と同じ操作を実行します。テーブルに同一の名前を持つ列が複数含まれる場合、クエリは列ごとに追加の結合を実行します。

## SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
NATURAL LEFT OUTER JOIN Sample.Company AS C
```

例：[2 つのテーブル間で同一の名前を持つ列での結合](#)

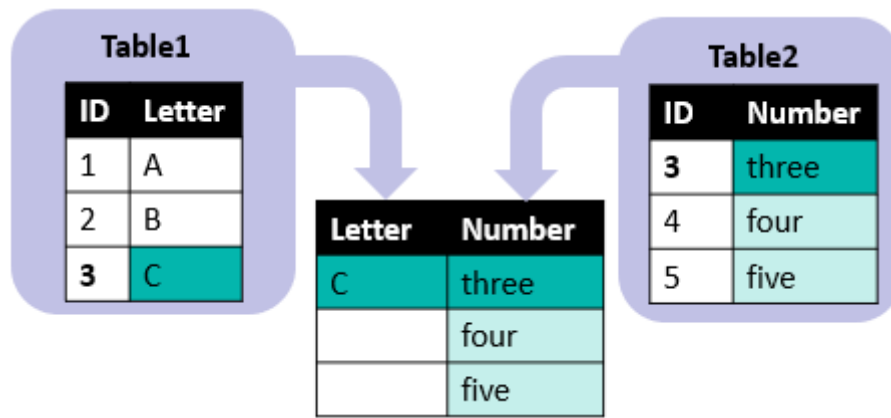
- ・ SELECT ... FROM table1 NATURAL LEFT JOIN table2 は、NATURAL LEFT OUTER JOIN 構文と等価です。

## RIGHT OUTER JOIN

RIGHT OUTER JOIN は、2 番目のテーブルのすべての行と、2 番目のテーブルの行と一致する最初のテーブルの行を返します。結合されたテーブルでは、最初のテーブルからの列の一致しない行には、NULL 値が移入されます。以下に例を示します。

## SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
RIGHT OUTER JOIN Table2
ON Table1.ID = Table2.ID
```



- SELECT ... FROM table1 RIGHT OUTER JOIN table2 ON condition は、table2 のすべての行を返し、これらを、ON 節で指定された条件式を満たす table1 の行と結合します。

このクエリは、Sample.Employee テーブル (E というエイリアス) と Sample.Company テーブル (C というエイリアス) のデータを結合し、従業員の名前とその会社を返します。これは、すべての C.Name 値を返しますが、E.Name 値については、両方の CompanyID 列の値が一致している行についてのみ返します。一致しない行については、E.Name 値は NULL に設定されます。

#### SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
LEFT OUTER JOIN Sample.Company AS C
ON E.CompanyID = C.CompanyID
```

例：

- 内部結合と外部結合を使用したテーブル・データの結合
- 結合データに対する追加の制限の設定

- SELECT ... FROM table1 RIGHT OUTER JOIN table2 USING (column, column2, ...) は、table2 のすべての行と、指定された列の値に一致する、table1 の行を返します。列は両方のテーブルに存在する必要があります。

このクエリは、両方の列に結合できる CompanyID 列があるため、前の構文と同じ結合を実行します。

#### SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
RIGHT OUTER JOIN Sample.Company AS C
USING (CompanyID)
```

例：2 つのテーブル間で同一の名前を持つ列での結合

- SELECT ... FROM table1 RIGHT JOIN table2 ... は、RIGHT OUTER JOIN 構文と等価です。
- SELECT ... FROM table1 NATURAL RIGHT OUTER JOIN table2 は、2 つのテーブル間で同一の名前を持つすべての列に対して RIGHT OUTER JOIN を実行します。式に複数の結合が含まれる場合は、最初に NATURAL 結合を指定します。NATURAL 結合では、同じ名前を持つ列はマージされません。

CompanyID が両方のテーブルに出現する唯一の列とすると、このクエリは、前の構文と同じ操作を実行します。テーブルに同一の名前を持つ列が複数含まれる場合、クエリは列ごとに 1 つの結合を実行します。

## SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
NATURAL RIGHT OUTER JOIN Sample.Company AS C
```

例：2 つのテーブル間で同一の名前を持つ列での結合

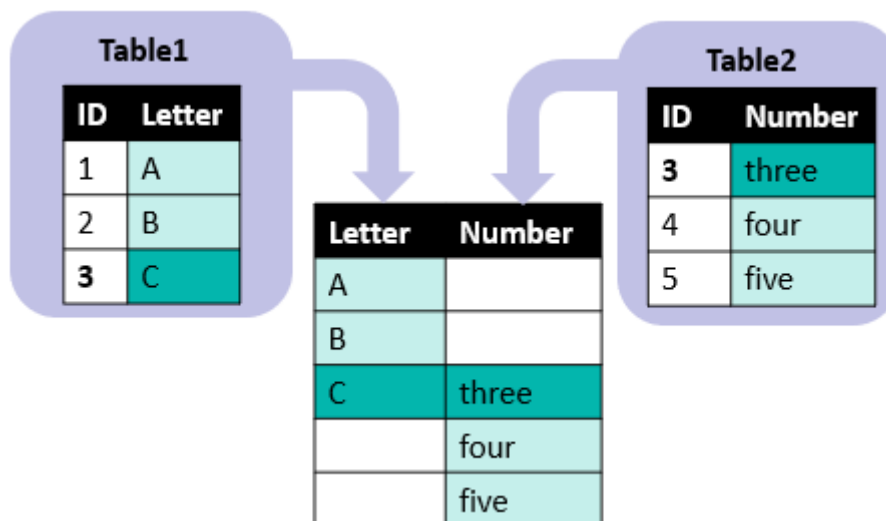
- SELECT ... FROM table1 NATURAL RIGHT JOIN table2 は、NATURAL RIGHT OUTER JOIN 構文と等価です。

## FULL OUTER JOIN

FULL OUTER JOIN は、両方のテーブルのすべての行を結合します。結合されたテーブルでは、いずれかのテーブルからの列の一致しない行には、NULL 値が移入されます。以下に例を示します。

## SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
FULL OUTER JOIN Table2
ON Table1.ID = Table2.ID
```



- SELECT ... FROM table1 FULL OUTER JOIN table2 ON condition は、指定した condition に一致する table1 および table2 のすべての行を返します。

このクエリは、人の名前とその勤務先の会社名を返します。Sample.Person と Sample.Company のデータが結合されます。それぞれの行について、PersonID で指定された人に Company 列か Person 列のいずれかが欠落している場合、その列値は NULL になります。

## SQL

```
SELECT P.Name, E.Company
FROM Sample.Person AS P
INNER JOIN Sample.Employee AS E
ON P.PersonID = E.PersonID
```

例：内部結合と外部結合を使用したテーブル・データの結合

- SELECT ... FROM table1 FULL JOIN table2 ON condition は、FULL OUTER JOIN 構文と等価です。

FULL OUTER JOIN では、USING および NATURAL 構文をサポートしていません。

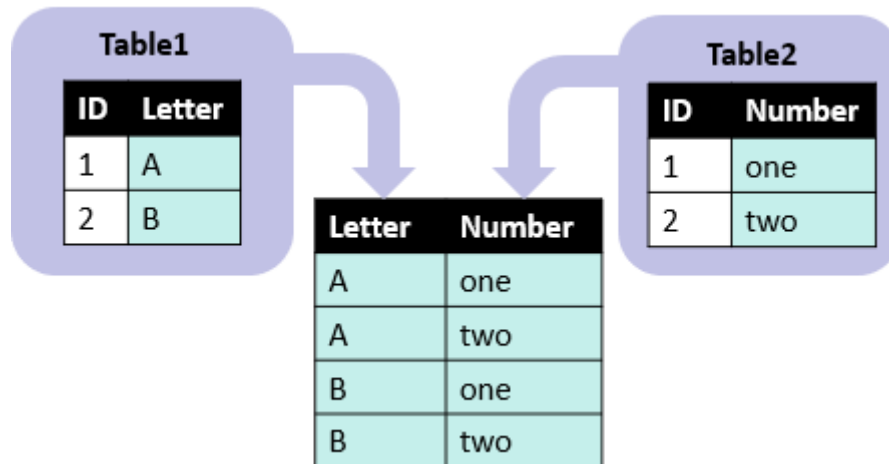


## CROSS JOIN

CROSS JOIN は最初のテーブルのすべての行を 2 番目のテーブルのすべての行と交差させます。以下に例を示します。

### SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
CROSS JOIN Table2
```



- SELECT ... FROM `table1` CROSS JOIN `table2` は、`table1` のすべての行と `table2` のすべての行を交差させ、多くのデータの重複を含む、大規模で論理的に包括的なテーブルを生成します。通常、この結合はコンマで区切られたテーブルのリストを FROM 節内で指定し、その後 WHERE 節を使用して制限条件を指定することによって実行されます。

このクエリは、`Sample.LettersAtoZ` および `Sample.Numbers1to10` 内の行の各組み合わせの行を返します。

```
SELECT * FROM Sample.LettersAtoZ CROSS JOIN Sample.Numbers1to10
```

以下のクエリは前のクエリと同等です。

### SQL

```
SELECT * FROM Sample.LettersAtoZ, Sample.Numbers1to10
```

ローカル・テーブルと、ODBC または JDBC ゲートウェイ接続を介してリンクされた外部テーブル (FROM `Sample.Person` `Mylink.Person` など) が関与する交差結合を実行しようとする、SQLCODE -161 エラーが発生します。この交差結合を実行するには、リンクされるテーブルをサブクエリとして指定する必要があります (例: FROM `Sample.Person`, (SELECT \* FROM `Mylink.Person`))。

JOIN キーワードの明示的な使用は、コンマ構文を使用した交差結合の指定より優先されます。したがって、InterSystems IRIS® では `t1,t2 JOIN t3` が `t1,(t2 JOIN t3)` と解釈されます。

## 引数

### `table1`, `table2`

結合されるテーブルの名前。FROM キーワードの後に、最初のテーブル `table1` を指定します。JOIN キーワードの後に、2 番目のテーブル `table2` を指定します。

- ON 節のある JOIN では、JOIN のどちらのオペランドにも、テーブル、ビュー、またはサブクエリを指定できます。

- ・ NATURAL 結合または USING 結合では、結合のいずれかのオペランドに、単純なベース・テーブル参照のみ (ビューやサブクエリではない) を指定できます。

table1 と table2 は、どちらも**テーブル・エイリアス**をサポートしています。

## condition

結合される行を制限するために ON 節で指定される、1 つ以上の条件式の述語。JOIN では、InterSystems SQL でサポートされている**述語**のほとんどがサポートされます。ただし、FOR SOME %ELEMENT コレクション述語を使用して結合操作を制限することはできません。

AND、OR、および NOT 論理演算子を使用して、複数の条件式を関連付けることができます。AND は OR よりも優先します。条件式を入れ子にしたり、グループ化するには、括弧を使用します。以下に例を示します。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
ON Patient.DocID = Doctor.DocID AND
   NOT (Doctor.State = 'NH' OR Doctor.State = 'MA')
```

condition には以下の制限事項があります。

- ・ condition は、ANSI キーワードの JOIN 操作で明示的に指定されたテーブルのみを参照できます。FROM 節で指定されたテーブルを参照すると、SQLCODE -23 エラーが発生します。
- ・ condition は、JOIN のオペランドに含まれる列のみを参照できます。複数の結合における構文優先順位が原因となって、ON 節が失敗する可能性があります。例えば、以下のクエリは、t1 と t3 が結合のオペランドでないため失敗します。t1 は、t2 JOIN t3 の結果セットと結合します。

## SQL

```
SELECT * FROM t1,t2 JOIN t3 ON t1.p1=t3.p3
```

構文を以下のように変更すると、このクエリの実行は成功します。

## SQL

```
SELECT * FROM t1 CROSS JOIN t2 JOIN t3 ON t1.p1=t3.p3
```

## SQL

```
SELECT * FROM t2,t1 JOIN t3 ON t1.p1=t3.p3
```

- ・ OUTER JOIN 節では、テーブルに影響を与えるすべての条件で NULL 値を渡す可能性がある比較を使用しており、テーブル自体が外部結合のターゲットである場合、SQLCODE -94 エラーが発生する可能性があります。例えば、以下の LEFT OUTER JOIN クエリは無効です。

## SQL

```
SELECT * FROM Table1
LEFT OUTER JOIN Table2 ON Table1.k = Table2.k
LEFT OUTER JOIN Table3 ON COALESCE(Table1.k,Table2.k) = Table3.k
```

FULL OUTER JOIN または RIGHT OUTER JOIN を使用した同様の例にも、この制限があります。

## column

両方のテーブルで名前が同じ列を結合するために USING 節で指定される、列名、または列名のコンマ区切りリスト。列リストは括弧で囲みます。明示的な列名のみ使用できます。自動生成された RowID 列を参照する %ID 行を指定することはできません。重複する列名は無視されます。同じ名前の列はマージされません。

## 例

### 内部結合と外部結合を使用したテーブル・データの結合

この例では、2 つのサンプル・テーブルを作成し、異なる INNER JOIN, LEFT OUTER JOIN、および RIGHT OUTER JOIN 構文を使用してそのデータを 1 つのテーブルに結合し、異なる結合結果を比較します。

#### テーブルの作成

この例では以下の 2 つのテーブルを使用します。

- Sample.HighestPeaks – 世界で最も高い山の標高 (フィート単位)。
- Sample.Himalayas – ヒマラヤの山の名前。

この例では指定されていませんが、両方のテーブルの MountainID 列と PeakID 列は、より大規模な山のデータベースへの外部キー参照であるとしてします。したがって、両方のテーブルで同じ ID 列値の行は同じ山を参照します。

Sample.HighestPeaks テーブルを作成し、3 つのデータ行を挿入します。テーブルを表示します。

#### SQL

```
CREATE TABLE Sample.HighestPeaks (
  PeakID INTEGER UNIQUE NOT NULL,
  Elevation INTEGER NOT NULL)
```

#### SQL

```
INSERT INTO Sample.HighestPeaks VALUES (1, 29032)
```

#### SQL

```
INSERT INTO Sample.HighestPeaks VALUES (2, 28251)
```

#### SQL

```
INSERT INTO Sample.HighestPeaks VALUES (3, 28169)
```

#### SQL

```
SELECT * FROM Sample.HighestPeaks
```

PeakID	Elevation
1	29032
2	28251
3	28169

Sample.Himalayas テーブルを作成し、3 つのデータ行を挿入します。MountainID の 2 は、意図的に省略されています。ID 2 の山はヒマラヤにはないものとします。テーブルを表示します。

## SQL

```
CREATE TABLE Sample.Himalayas (  
    MountainID INTEGER UNIQUE NOT NULL,  
    Name VARCHAR(30) UNIQUE NOT NULL)
```

## SQL

```
INSERT INTO Sample.Himalayas VALUES (1, 'Everest')
```

## SQL

```
INSERT INTO Sample.Himalayas VALUES (3, 'Kangchenjunga')
```

## SQL

```
INSERT INTO Sample.Himalayas VALUES (4, 'Lhotse')
```

## SQL

```
SELECT * FROM Sample.Himalayas
```

MountainID	Name
1	Everest
3	Kangchenjunga
4	Lhotse

## INNER JOIN の実行

INNER JOIN を使用して 2 つのテーブルの山の名前と標高のデータを結合します。これらは、MountainID 列と PeakID 列で結合します。結合されたテーブルには、両方のテーブルに ID が存在する ID 1 と ID 3 の山のデータのみが含まれます。

## SQL

```
SELECT H.Name, P.Elevation  
FROM Sample.Himalayas AS H  
INNER JOIN Sample.HighestPeaks as P  
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
Kangchenjunga	28169

## LEFT OUTER JOIN の実行

LEFT OUTER JOIN を使用して山の名前と標高のデータを結合します。これらは、MountainID 列と PeakID 列で結合します。結合されたテーブルには、最初のテーブル (Sample.Himalayas) のすべての行が含まれますが、2 番目のテーブル (Sample.HighestPeaks) については、PeakID 値が 1 と 3 の行のみ (これらは最初のテーブルの MountainID 列にも存在しているため) が含まれます。欠落している Lhotse 山の標高は、NULL 値を取ります。

## SQL

```
SELECT H.Name, P.Elevation  
FROM Sample.Himalayas AS H  
LEFT OUTER JOIN Sample.HighestPeaks as P  
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
Kangchenjunga	28169
Lhotse	

### RIGHT OUTER JOIN の実行

RIGHT OUTER JOIN を使用して山の名前と標高のデータを結合します。これらは、MountainID 列と PeakID 列で結合します。結合されたテーブルには、2 番目のテーブル (Sample.HighestPeaks) のすべての行が含まれますが、最初のテーブル (Sample.Himalayas) については、MountainID 値が 1 と 3 の行のみ (これらは 2 番目のテーブルの PeakID 列にも存在しているため) が含まれます。欠落している 28,251 フィートの標高の山の名前は、NULL 値を取ります。

### SQL

```
SELECT H.Name, P.Elevation
FROM Sample.Himalayas AS H
RIGHT OUTER JOIN Sample.HighestPeaks as P
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
	28251
Kangchenjunga	28169

### FULL OUTER JOIN の実行

FULL OUTER JOIN を使用して山の名前と標高のデータを結合します。これらは、MountainID 列と PeakID 列で結合します。結合されたテーブルには、両方のテーブルのすべての行が含まれます。欠落している山の名前と標高は NULL 値を取ります。

### SQL

```
SELECT H.Name, P.Elevation
FROM Sample.Himalayas AS H
FULL OUTER JOIN Sample.HighestPeaks as P
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
Kangchenjunga	28169
Lhotse	
	28251

### テーブルの削除

完了したら、サンプル・テーブルを削除します。

### SQL

```
DROP TABLE Sample.Himalayas
```

## SQL

```
DROP TABLE Sample.HighestPeaks
```

## 2つのテーブル間で同一の名前を持つ列での結合

次の例では、2つのテーブル間で同一の名前を持つ列を結合する際に使用できるさまざまな構文を示します。

以下の2つのテーブルについて考えます。

- ・ Patient – 患者に関する情報 (患者のかかりつけ医の ID コード (DocID) など) が含まれます。
- ・ Doctor – 医師に関する情報 (医師の ID コード (DocID) など) が含まれます。

以下の INNER JOIN は患者名と医師名を返します。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
ON Patient.DocID = Doctor.DocID
```

結合する列は、両方のテーブルで同じ名前 (DocID) を持つため、ON 節を USING 節に置き換えることができます。以下の構文では、括弧で囲んで列のみを指定し、テーブル名は省略します。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
USING (DocID)
```

USING 節を LEFT OUTER JOIN または RIGHT OUTER JOIN で指定することもできますが、FULL OUTER JOIN はサポートされていません。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
RIGHT OUTER JOIN Doctor
USING (DocID)
```

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
USING (DocID)
```

DocID が2つのテーブル間で同じ名前を持つ唯一の列である場合は、さらに単純化して NATURAL JOIN 構文を使用できます。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
NATURAL INNER JOIN Doctor
```

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
NATURAL LEFT OUTER JOIN Doctor
```

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
NATURAL RIGHT OUTER JOIN Doctor
```

2つのテーブルに他に同一の列が含まれている場合、NATURAL JOIN は、結合操作でこれらの列もリンクさせます。結合する列をより詳細に指定するには、USING 節または ON 節を使用します。FULL OUTER JOIN では、NATURAL JOIN をサポートしていません。

## 結合データに対する追加の制限の設定

この例では、追加の制限の設定により、さまざまな結合で返されるデータがどのように影響を受けるかを示します。

以下の 2 つのテーブルについて考えます。

- ・ Patient – 患者に関する情報 (患者のかかりつけ医の ID コード (DocID) など) が含まれます。
- ・ Doctor – 医師に関する情報 (医師の ID コード (DocID) など) が含まれます。

以下の INNER JOIN は 45 歳を超える医師の患者名とその医師名を返します。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
ON Patient.DocID = Doctor.DocID AND Doctors.Age > 45
```

同じクエリの LEFT OUTER JOIN を実行しても、結合されるテーブルの一致しない行の NULL 値は排除されません。例えば、この LEFT OUTER JOIN は、依然として Doctor.DName 列に NULL 値を返します。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
ON Patient.DocID = Doctor.DocID AND Doctors.Age > 45
```

NULL 値を排除するには、年齢条件を WHERE 節に移動し、結合操作後にここで処理します。ただし、これによってクエリは効果的に INNER JOIN に変換されます。例えば、以下のクエリはこの例の最初のクエリと同等です。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
ON Patient.DocID = Doctor.DocID
WHERE Doctors.Age > 45
```

IS NULL 節を追加すると、元の LEFT OUTER JOIN の動作は保持されますが、元の LEFT OUTER JOIN クエリより冗長になります。

## SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
ON Patient.DocID = Doctor.DocID
WHERE Doctors.Age > 45 AND Doctors.Age IS NULL
```

この動作は、RIGHT OUTER JOIN 操作でも同様です。FULL OUTER JOIN では、一致に関係なく、両方のテーブルのすべての行が返されるため、条件を指定しても返される行には影響しません。



## パフォーマンス

### クエリ・オプティマイザの結合に対する影響

結合操作のパフォーマンスを最大にするため、SQL オプティマイザは指定された順序でテーブルを結合しない場合があります。代わりに、オプティマイザは、[テーブルのチューニング](#)など、テーブルで収集される統計に基づいてテーブルの結合順序を決定します。

ほとんどの場合、SQL オプティマイザの最適化方法で最適な結果が得られます。ただし、特定のクエリに対して、既定の最適化方法をオーバーライドするため、FROM キーワードの直後に[クエリ最適化オプション](#)を指定することができます。

- ・ %INORDER、%FIRSTTABLE、および %STARTTABLE – 複数の結合が含まれる複雑なクエリでは、これらのオプションで明示的に結合するテーブルの順序を設定します。これらのキーワードは、CROSS JOIN や RIGHT OUTER JOIN では使用できません。使用しようとすると SQLCODE -34 エラーが発生します。
- ・ %NOFLATTEN – このオプションは、特定のサブクエリを明示的な結合に変換するサブクエリの平坦化を無効にします。サブクエリ为数が少ない場合、サブクエリの平坦化により結合パフォーマンスが大幅に向上することがあります。ただし、サブクエリが増加すると、サブクエリの平坦化のパフォーマンスは低下し始め、このキーワードを使用して無効にする必要が生じる場合があります。

### ON 節のインデックス作成

結合の ON 節で参照される列にインデックスを指定すると、クエリのパフォーマンスが大幅に向上します。ON 節では、いくつかの結合条件のみを満たす既存のインデックスを使用できます。複数の列に対して条件を指定する ON 節は、添え字としてこれらの列のサブセットのみを含むインデックスを使用して、結合を部分的に満たすことができます。InterSystems IRIS は、直接テーブルの残りの列に対して結合条件をテストします。

ON 節で参照されているフィールドの[照合タイプ](#)は、対応するインデックス内でのそのフィールドの照合タイプと一致する必要があります。照合タイプが一致していない場合は、インデックスは使用されない可能性があります。ただし、結合条件が %EXACT 照合を持つ列に存在しており、照合された列値のインデックスのみ使用可能な場合、InterSystems IRIS はそのインデックスを使用して、正確な値であるかどうか確認する行を制限できます。照合タイプのマッチングの詳細は、["インデックス照合"](#)を参照してください。

ON 節の条件のインデックスを無効にするには、この先頭に %NOINDEX キーワードを付けます。インデックスとパフォーマンスの詳細は、["クエリ処理でのインデックスの使用"](#) および ["インデックスの最適化オプション"](#)を参照してください。

## 代替案

InterSystems IRIS には、外部結合の表示に 2 種類の形式があります。

1. (推奨) ANSI 標準構文の LEFT OUTER JOIN と RIGHT OUTER JOIN。以下の例で示しているように、SQL 標準構文は WHERE 節ではなく、SELECT 文の FROM 節で外部結合を入力します。

#### SQL

```
SELECT table1.columnA, table2.columnB
FROM table1
LEFT OUTER JOIN table2
ON (table1.columnX = table2.columnY)
```

2. エスケープ構文 {oj joinExpression} を使用する、ODBC 仕様外部結合拡張構文 (joinExpression は任意の ANSI 標準結合構文)。

ON 節のある結合では ANSI 結合キーワード構文のみ使用できます。

## 関連項目

- ・ [SELECT](#)、[FROM](#)、[ORDER BY](#)、[WHERE](#)

- ・ ALTER TABLE、CREATE TABLE、DROP TABLE
- ・ INSERT、UPDATE
- ・ テーブルの定義
- ・ データベースの問い合わせ
- ・ SQLCODE エラー・メッセージ

## LOAD DATA (SQL)

データをテーブルにロードします。

### 構文

#### ファイルからのロード

```
LOAD DATA FROM FILE filePath INTO table
LOAD DATA FROM FILE filePath INTO table (column, column2, ...)
LOAD DATA FROM FILE filePath COLUMNS (header type, header2 type2, ...)
  INTO table ...
LOAD DATA FROM FILE filePath COLUMNS (header type, header2 type2, ...)
  INTO table ... VALUES (header, header2, ...)
LOAD DATA FROM FILE filePath INTO table ... USING jsonOptions
```

#### JDBC ソースからのロード

```
LOAD DATA FROM JDBC CONNECTION jdbcConnection
  TABLE jdbcTable INTO table
LOAD DATA FROM JDBC CONNECTION jdbcConnection
  TABLE jdbcTable INTO table (column, column2, ...)
LOAD DATA FROM JDBC CONNECTION jdbcConnection
  TABLE jdbcTable INTO table ... VALUES (header, header2 ...)
LOAD DATA FROM JDBC URL path TABLE jdbcTable ...
```

#### 一括ロード・オプション

```
LOAD BULK DATA FROM ...
LOAD %NOJOURN DATA FROM ...
LOAD BULK %NOJOURN DATA FROM ...
LOAD [ load-option] DATA FROM ...
```

### 概要

LOAD DATA コマンドは、ソースからのデータを以前に定義した InterSystems IRIS® SQL テーブルにロードします。ソースはデータ・ファイルまたは JDBC を使用してアクセスされるテーブルのどちらでもかまいません。十分に検証されたデータで迅速にテーブルを生成するには、このコマンドを使用します。

データをロードするテーブルが空の場合、LOAD DATA はこのテーブルにソース・データ行を移入します。テーブルに既にデータが含まれている場合、LOAD DATA は、既存の行は上書きせずに、テーブルにソース・データ行を挿入します。

データをロードする際、%ROWCOUNT 変数は、正常にロードされた行数を示します。入力データ内の行にエラーが含まれる場合、LOAD DATA はこの行のロードをスキップして、次の行のロードに進みます。SQLCODE はこれをエラーとして報告しませんが、%SQL\_Diag.Result ログにはどれだけのレコードがロードに失敗したかが示されます。詳細は、“[ロードされたデータの診断ログの表示](#)”を参照してください。

**注釈** LOAD DATA コマンドは、基礎となる Java ベースのエンジンを使用します。これには、Java 仮想マシン (JVM) がサーバ上にインストールされていることが必要です。既に JVM のセットアップがあり、PATH 環境変数でアクセス可能な場合、最初に LOAD DATA を使用すると、InterSystems IRIS は自動的にその JVM を使用して外部言語サーバを起動します。外部言語サーバをカスタマイズして、特定の JVM を使用するか、リモート・サーバを使用するには、“[外部サーバ接続の管理](#)”を参照してください。

#### ファイルからのロード

##### ヘッダのないファイルからのロード

ソース・ファイルの最初の行にヘッダ行が含まれていない場合は、これらの構文を使用します。そうしないと、LOAD DATA はヘッダ行をテーブルにロードします。

- ・ `LOAD DATA FROM FILE filePath INTO table` `filePath` で指定されたファイルからのソース・データをターゲットの SQL テーブルにロードします。既定では、`LOAD DATA` は、位置でデータ・ソースからの列をターゲット・テーブルにマッチングします。`LOAD DATA` は SQL の列の順序 (`SELECT *` の列の順序) を使用します。
  - データ・ソースに入力テーブルより多くの列がある場合、余分な列は無視され、テーブルにロードされません。
  - データ・ソースの列が入力テーブルより少ない場合、どのデータもテーブルにロードされません。

`LOAD DATA` コマンドは、ロードされるデータのデータ型が、ターゲット・テーブルの列のデータ型と一致することを期待しています。

この文は、`countries` CSV ソース・ファイルからのすべての列を、ターゲットとなる `Sample.Countries` テーブルの対応する位置にある列にロードします。

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries
```

- ・ `LOAD DATA FROM FILE filePath ... INTO table (column, column2, ...)` 位置的に指定したターゲット・テーブルの列のソース・データのみをロードします。データ・ソースの列が入力テーブルより少ない場合、挿入行でこれらの列が空白になります。

この文は、`countries` CSV ファイルの最初の 3 つの列を、`Sample.Countries` テーブルの `Name`、`Continent`、および `Region` の列にロードします。テーブルでこれらの列が異なる順序で格納されている場合でも、あるいは、ここで示した 3 つの列の間に列が存在する場合でも、`LOAD DATA` は `Name`、`Continent`、`Region` にのみデータをロードします。

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries (Name,Continent,Region)
```

- ・ `LOAD DATA FROM FILE filePath COLUMNS (header type, header2 type2, ...) INTO table` ターゲット・テーブルと列の順序が異なるソース・ファイルのデータをロードできます。`COLUMNS` 節は、ソース・ファイル内の列のヘッダ名とデータ型を提供します。ヘッダ名は、ターゲット・テーブル内の列の名前と一致する必要があり、データ型は、これらのテーブルの列のデータ型と一貫している必要があります。
  - `INTO table` 節がターゲットの列を指定している場合、`COLUMNS` 節で指定されている列は `INTO table` 節にも出現する必要があります。ただし、順序は任意です。
  - `INTO table` 節がターゲットの列を指定しない場合、`LOAD DATA` はソースの列を位置に基づいてテーブルにロードします。`COLUMNS` 節は、ターゲット・テーブルに出現するすべての列を指定する必要があります。

この文は、`countries` CSV ファイルの 3 つの列を `Sample.Countries` テーブルの対応する列にロードします。`Sample.Countries` テーブルの列の順序がソース・ファイルと異なる場合 (例えば、`Name`、`Continent`、`SurfaceArea` の順ではなく、`Name`、`SurfaceArea`、`Continent` の順の場合)、テーブルの列の順序は変更されません。

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
COLUMNS (
    Name VARCHAR(50),
    Continent VARCHAR(30),
    SurfaceArea Integer)
INTO Sample.Countries (Name,Continent,SurfaceArea)
```

- ・ `LOAD DATA FROM FILE filePath COLUMNS (header type, header2 type2, ...) INTO table ... VALUES (header, header2, ...)` では、さらに、ソース・ファイルの列のサブセットをターゲットの列にロードできます。これらの列名は、ターゲット・テーブルの列名と一致する必要はありません。

`VALUES` 節は、`COLUMNS` 節のヘッダで指定されたとおりにソース列を指定し、ターゲット・テーブルにロードします。

- INTO table 節でターゲットの列を指定する場合、LOAD DATA は、ソース列をそれらが指定されている順序でターゲットの列にロードします。VALUES 内のソースの列ヘッダの数は、INTO table 節内の列の数と一致する必要があります。
- INTO table 節がターゲットの列を指定しない場合、LOAD DATA はソースの列を位置に基づいてテーブルにロードします。VALUES 内のソース・ヘッダ数は、テーブル内の列数と一致している必要があります。

この文は、countries CSV ファイルの 3 つの列を Sample.Countries テーブルの対応する列にロードします。COLUMNS 節には、追加の列 src\_continent のヘッダ名が含まれます。これはテーブルにはロードされません。この列名は無視されますが、LOAD DATA がそのデータを後続の列 (src\_region および src\_surface\_area) からテーブルにロードできるように、含められる必要があります。

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
COLUMNS (
    src_name VARCHAR(50),
    src_continent VARCHAR(30),
    src_region VARCHAR(30),
    src_surface_area INTEGER)
INTO Sample.Countries (Name, SurfaceArea, Region)
VALUES (src_name, src_surface_area, src_region)
```

COLUMNS 節を使用せずに VALUES 節を指定する場合、VALUES 節は無視されます。

### ヘッダ付きのファイルからのロードとオプションの指定

この構文は、ソース・ファイルの最初の行にヘッダ行が含まれる場合に使用します。この構文を使用すると、ヘッダ行をスキップするオプションを指定できます。他には、既定の列の変更、ヘッダ以外の追加行のスキップ、既定のエスケープ文字の変更のオプションが含まれます。

- LOAD DATA FROM FILE filePath INTO table ... USING jsonOptions は、JSON オブジェクトまたは JSON オブジェクトを含む文字列を使用することにより、ロード・オプションを指定します。

この文は JSON オブジェクトを使用して、ファイルにヘッダ行を含めることを指定し、LOAD DATA がテーブル内にこの行を含めないようにします。この文で、countries CSV ファイルのヘッダ名は Sample.Countries テーブル列のヘッダ名と一致していると見なされます。

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries
USING {"from":{"file":{"header":true}}}
```

**注釈** ヘッダ・テキストがフィールド・データ・タイプに対して検証しない場合 ("Total" という名前のヘッダを持つ整数フィールドなど)、LOAD DATA はそのヘッダ行を省略できます。ただし、この検証拒否のメソッドは信頼性が低く、お勧めしません。代わりに USING 節を使用してヘッダを省略します。

この文は、countries CSV ファイルの 3 つの列のデータを Sample.Countries テーブルの対応する 3 つの列にロードします。この文で、countries CSV ファイルのヘッダ名は Sample.Countries テーブル列のヘッダ名と一致しません。VALUES 節はこのファイルから取得される列ヘッダ名を指定します。これらの列のデータは、INTO table 節の対応する位置にあるテーブル列にロードされます。

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries (Name, Region, SurfaceArea)
VALUES (country_name, region_name, surface_area)
USING {"from":{"file":{"header":true}}}
```

### JDBC ソースからのロード

- LOAD DATA FROM JDBC CONNECTION connection TABLE jdbcTable INTO table 外部 JDBC データ・ソースのデータをターゲット・テーブルにロードします。データ・ソース jdbcTable は、定義した SQL ゲートウェイ接続

connection を使用して接続する JDBC 準拠の SQL テーブルです。詳細は、“JDBC 経由での SQL ゲートウェイへの接続”を参照してください。

この文は、JDBC ソース・テーブル countries のすべての列を Sample.Countries テーブルの対応する列にロードします。

```
LOAD DATA FROM JDBC CONNECTION MyJDBCConnection
TABLE countries
INTO Sample.Countries
```

- LOAD DATA FROM JDBC CONNECTION connection TABLE jdbcTable (column, column2, ...) は、JDBC ソース・データを、位置的に指定されたターゲットのテーブル列にのみロードします。JDBC ソースの列が入力テーブルより少ない場合、挿入行でこれらの列が空白になります。

この文は、JDBC の countries テーブルの最初の 3 つの列を、Countries テーブルの Name、Continent、および Region の列にロードします。テーブルでこれらの列が異なる順序で格納されている場合でも、あるいは、ここで示した 3 つの列の間に列が存在する場合でも、LOAD DATA は Name、Continent、Region にのみデータをロードします。

```
LOAD DATA FROM JDBC CONNECTION MyConnection
TABLE countries
INTO Sample.Countries (Name,Continent,Region)
```

- LOAD DATA FROM JDBC CONNECTION connection TABLE jdbcTable ... INTO table ... VALUES (header,header2 ...) は、JDBC ソースのデータを VALUES 節で指定されたヘッダ名を持つ列のみにロードします。この構文を使用すると、JDBC ソース・テーブル内の任意の場所にある列データを、ターゲット・テーブルの任意の場所にある列に配置できます。

INTO table 節内の列数は、VALUES 節内のヘッダ数と一致する必要があります。INTO table 節が列を指定しない場合、VALUES 節内のヘッダ数はテーブルのヘッダ数と一致する必要があります。この場合、ソース・データは位置に基づいてテーブルにロードされます。

この文は JDBC countries テーブルの name、surface\_area、および region 列のデータを Sample.Countries テーブルの対応する列にロードします。

```
LOAD DATA FROM JDBC CONNECTION MyConnection
TABLE countries
INTO Sample.Countries (Name,SurfaceArea,Region)
VALUES (name,surface_area,region)
```

VALUES 節が SQL の列名を位置でマッチングする方法は、INSERT コマンドの構文と似ています。

- LOAD DATA FROM JDBC URL path TABLE jdbcTable INTO table 外部 JDBC データ・ソースのデータをターゲット・テーブルにロードします。path で定義されるデータ・ソースは、データ・ソースの接続 URL にあります。詳細は、“JDBC 経由での SQL ゲートウェイへの接続”を参照してください。

この文は、JDBC ソース・テーブル countries のすべての列を Sample.Countries テーブルの対応する列にロードします。

```
LOAD DATA FROM JDBC URL jdbc:oracle:thin:@//oraserver:1521/SID
TABLE countries
INTO Sample.Countries
```

## 一括ロード・オプション

これらのオプションは、データがテーブルに挿入される際に実行される一般的なチェックや操作を無効にします。これらのオプションを無効にすることにより、大量の行を含むデータのロードを大幅に高速化できます。

**注意** これらの一括ロード・オプションにより、無効なデータがロードされる場合があります。これらのオプションを使用する前に、データが有効で、信頼できるソースからのものであることを確認してください。



- LOAD BULK DATA FROM ... は、以下の INSERT [%keyword](#) オプションを指定してデータをロードします。
  - [%NOCHECK](#) – 一意の値のチェック、外部キーの参照整合性チェック、NOT NULL 制約 (フィールド・チェックが必要)、および列のデータ型、最大列長、列のデータ制約の検証を無効にします。
  - [%NOINDEX](#) – INSERT 処理の際にインデックス・マップの設定を無効にします。LOAD BULK DATA 操作時に、ターゲット・テーブルに対する SQL 文の実行が不完全となる、または正しくない結果を返す可能性があります。
  - [%NOLOCK](#) – INSERT の実行時に行のロックを無効にします。

BULK キーワードを使用するには、[%NOCHECK](#)、[%NOINDEX](#)、および [%NOLOCK](#) 管理特権 ([GRANT](#) コマンドを使用して設定可能) が必要です。

この文は、ファイルからバルク・データをロードします。

```
LOAD BULK DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries
```

- LOAD [%NOJOURN](#) DATA FROM ... は、[%NOJOURN](#) INSERT [%keyword](#) オプションを指定してデータをロードします。このオプションは、挿入操作の間、ジャーナリングを抑制し、トランザクションを無効化します。[%NOJOURN](#) オプションを使用するには、[%NOJOURN](#) SQL 管理特権 ([GRANT](#) コマンドを使用して設定可能) が必要です。

この形式の LOAD DATA コマンドは、ターゲット・テーブルに対するテーブルレベルのロックを取得しますが、各行は [%NOLOCK](#) により挿入されます。テーブル・レベル・ロックは、コマンドが完了すると解放されます。

この文は、JDBC 接続を介してテーブルからデータをロードし、ジャーナリングを無効にします。

```
LOAD %NOJOURN DATA FROM JDBC CONNECTION MyJDBCConnection
TABLE countries
INTO Sample.Countries
```

- LOAD [%NOJOURN](#) BULK DATA FROM ... は、前に指定した構文から INSERT [%keyword](#) オプションを指定してデータをロードします。[%NOJOURN](#) と BULK は、任意の順序で指定できます。
- LOAD [ load-option ] DATA FROM ... は、load-option ヒントを使用してデータをロードします。これは、[%NOCHECK](#)、[%NOINDEX](#)、[%NOLOCK](#)、[%NOJOURN](#) の任意の組み合わせとなります。これにより、必要に応じて一括ロードの特定の最適化を採用できます。BULK を [%NOCHECK](#)、[%NOINDEX](#)、または [%NOLOCK](#) と共に指定することはできません。

## 引数

### filePath

ロードするデータを含むテキスト・ファイルのサーバ側の位置。引用符で囲まれた完全なファイル・パスで指定します。

- ファイル内の各行は、テーブルにロードされる個別の行を指定します。空白行は無視されます。
- 行内のデータ値は列区切り文字で区切られます。コンマが既定の列区切り文字です。プレースホルダ列区切り文字で示される未指定のデータを含め、すべてのデータ・フィールドが列区切り文字で示される必要があります。別の列区切り文字を定義するには、[USING jsonOptions](#) 節で [columnseparator](#) オプションを指定します。
- 既定では、エスケープ文字は定義されていません。データ値内にリテラルとして列区切り文字を含めるには、データ値を疑問符で囲みます。引用符付きのデータ値内に引用符を含めるには、引用符文字を二重にします (``)。エスケープ文字を定義するには、[USING jsonOptions](#) 節で [escapechar](#) オプションを指定します。
- 既定では、データ値はテーブル (またはビュー) 内のフィールドの順に指定されます。[COLUMNS](#) 節を使用すると、データを別の順序で指定できます。ビューを使用してデータ・レコードをテーブルにロードするには、ビュー内で定義されたフィールドの値のみを指定します。



- データ・ファイル・レコード内のすべてのデータは、テーブルのデータ条件に対して検証されます。この条件には、レコード内のデータ・フィールドの数、各フィールドのデータ型とデータ長などがあります。検証に失敗したデータ・ファイル・レコードは無視されます（ロードされません）。エラー・メッセージは発行されません。引き続き次のレコードのデータがロードされます。

注釈 日付またはタイムスタンプは、確実に検証するため、ODBC のタイムスタンプ形式 ('yyyy-mm-dd hh:mm:ss') で書き込む必要があります。

## table

データのロード先のテーブル。テーブル名は修飾 (schema.tablename)、未修飾 (tablename) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。ビューを指定すると、そのビューからアクセスされるテーブルにデータをロードできます。

## column

ファイル・データのロード先のテーブル列。ファイル内の列の順序で指定します。この列名のリストにより、選択するテーブルの列を指定し、データ・ファイル項目の順序を table 内の列に合わせることができます。テーブル内で定義されている未指定の列は、既定値を取ります。この節が省略された場合、table 内のすべてのユーザ定義フィールドが、データ・ファイルに表示されます。

## header

データ・ソースからロードする列の特定に使用されるヘッダ値のコンマ区切りリスト。

- ヘッダ行を含まないファイル・ソースのデータをロードする際は、COLUMNS (*header type*, *header2 type2*, ...) 節内にヘッダを指定することにより列を指定します。
  - VALUES 節を含める場合、これらのヘッダ名を VALUES (*header*, *header2*) に指定して、テーブルにロードする列を選択します。
  - VALUES 節を含めない場合は、これらのヘッダ名はターゲット・テーブルの列名と一致させる必要があります。
- ヘッダ行を含むファイル・ソースのデータをロードする場合は、VALUES (*header*, *header2*) 節にヘッダを指定し、データのロード元のソース・ファイルのヘッダを特定します。これらのヘッダ名は、ソース・ファイルに存在している必要があります。
- JDBC ソースのデータをロードする場合は、VALUES (*header*, *header2*) 節にヘッダを指定し、データのロード元の JDBC ソース・テーブルの列を特定します。これらのヘッダ名は、JDBC ソース・テーブルに存在している必要があります。

## type

COLUMNS (*header type*, *header2 type2*, ...) 節に指定されたヘッダのデータ型。各列のデータ型は、テーブルのデータ型と互換性がある必要があります。テーブルのデータ長 (COLUMNS のデータ長ではない) を使用してデータを検証します。

## jsonOptions

USING 節で JSON (JavaScript Object Notation) オブジェクト、または JSON オブジェクトを含む文字列として指定されるロード・オプション。以下の各構文は同等です。

```
USING {"from":{"file":{"header":true}}}
```

```
USING '{"from":{"file":{"header":true}}}'
```

これらの JSON オブジェクトを使用して、SQL キーワードでは設定できないロード・オプションを設定します。["JSON 値"](#)の説明のように、入れ子になった key:value ペア構文を使用してこれらのオブジェクトを指定します。

このオブジェクトの主な用途は、FROM FILE 構文を補足する、ロードされるデータのオプションの設定ですが、LOAD DATA の実行中に、並列処理し、エラーを許可するオプションもあります。この例は、複数のオプションが指定されたサンプルの JSON オブジェクトを示しています。ここに示されている空白はオプションで、読みやすくするためだけに使用されています。

```
USING
{
  "from": {
    "file": {
      "header": true,
      "skip": 2
      "charset": "UTF-8"
      "escapechar": "\\\"
      "columnseparator": "\t"
    }
  }
}
```

以下のテーブルは、指定できるオプションを示しています。未指定のオプションは既定値を使用します。

オプション	説明	例
-------	----	---

オプション	説明	例
from.file.header	<p>true (1) を設定すると、ソース・ファイルの最初の行がヘッダ行であることを示します。この場合、このヘッダ内の列名を指定することができ、COLUMNS 節が指定されていない場合は、これらを VALUES で使用できます。詳細は、“<a href="#">ヘッダ付きのファイルからのロードとオプションの指定</a>”を参照してください。</p> <p>既定値 : false (0)</p>	<code>{"from":{"file":{"header":true}}}</code>
from.file.skip	<p>ファイルの最初でスキップする行数を指定します。header が true に設定されている場合、skip はヘッダに加えてスキップする行の数を示します。</p> <p>既定値 : 0</p>	<code>{"from":{"file":{"skip":2}}}</code>
from.file.charset	<p>入力データの解析に使用する文字セットを指定します。</p> <p>既定値 : LOAD DATA は、ホストのオペレーティング・システムの文字セットを使用します。</p>	<code>{"from":{"file":{"charset":"UTF-8"}}}</code>
from.file.escapechar	<p>列の値内で使用される列区切り文字など、リテラル値に使用するエスケープ文字を指定します。</p> <p>既定値 : なし</p>	<code>{"from":{"file":{"escapechar":"\\"}}}</code>
from.file.columnseparator	<p>列区切り文字を指定します。</p> <p>既定値 : ", "</p>	<code>{"from":{"file":{"columnseparator":","}}}</code>

オプション	説明	例
<code>into.jdbc.threads</code>	<p>JDBC ライターを並列化するスレッドの数を指定します。このオプションは、JDBC ソースからデータをロードしない場合でも使用できます。各スレッドは、INSERT コマンドを実行する単一のサーバ・プロセスにデータを提供します。テーブルで定義されたとおりの順序でデータをテーブルにロードすることが重要な場合は、<code>"threads":1</code> を指定する必要があります。一般的に、複数の LOAD DATA コマンドを並列で実行する場合は、小さい値を指定する必要があります。</p> <p>既定値 : <code>\$System.Util.NumberOfCPUs() - 2</code></p>	<code>{"into":{"jdbc":{"threads":4}}}</code>
<code>from.jdbc.fetchsize</code>	<p>テーブルにロードするために一度に取得する行数を指定します。多くの場合、大量のデータをロードするときに、<code>fetchsize</code> に大きな値を設定すると、パフォーマンスが向上します。<code>fetchsize</code> が 0 の場合は、JDBC ドライバが取得サイズを選択します。ほとんどの JDBC ドライバは 10 という値を選択します。この設定を有効にするには、サーバも <code>fetchsize</code> パラメータをサポートする必要があります。サポートしていない場合、これは無視されます。</p> <p>既定値 : 0</p>	<code>{"from":{"jdbc":{"fetchsize":256}}}</code>
<code>maxerrors</code>	<p>LOAD DATA コマンド中に生じる可能性のあるエラーの最大数。この数を超えると、操作全体が失敗と判断され、トランザクションが閉じられ、すべての変更がロールバックされます。</p> <p>既定値 : 0</p>	<code>{"maxerrors":5}</code>

## jdbcConnection

JDBC ソースからのデータのロードに使用される、定義された SQL ゲートウェイ接続の名前。JDBC 接続の確立の詳細は、“JDBC 経由での SQL ゲートウェイへの接続”を参照してください。

## jdbcTable

JDBC 接続を介してアクセスされる外部 SQL データ・ソース・テーブル。JDBC 接続の確立の詳細は、“JDBC 経由での SQL ゲートウェイへの接続”を参照してください。

## jdbcTable

JDBC ソースからのデータのロードに使用される、SQL ゲートウェイ接続の URL。JDBC 接続の確立の詳細は、“JDBC 経由での SQL ゲートウェイへの接続”を参照してください。

## load-option

1 つ以上の INSERT [%keyword](#) オプション。1 つの空白文字で区切られます。これらのオプションは、LOAD DATA コマンドの特定の動作を指定します。これらのオプションの詳細は、“[一括ロード・オプション](#)”を参照してください。

## セキュリティおよび特権

LOAD DATA は、ユーザがロード先のテーブルを変更し、サーバ上の JVM にアクセスするために、INSERT 特権を必要とする特権操作です。

### INSERT 特権

テーブルで LOAD DATA を実行するには、ユーザがそのテーブルに対してテーブルレベルまたは列レベルの特権を持っている必要があります。特に、ユーザは、テーブルに対する INSERT 特権を持っている必要があります。テーブルの所有者（作成者）にはそのテーブルに対するすべての特権が自動的に付与されます。所有者でない場合は、そのテーブルに対する特権が付与される必要があります。適切な特権がない場合、InterSystems IRIS では SQLCODE -99 エラーが発生します。

テーブルレベルの特権は、テーブルの全列に対して列レベルの特権を持っていることと同等ですが、まったく同じではありません。テーブルの列のサブセットに対する列レベルの特権のみがある場合、これらの列にデータをロードすることのみが可能です。許可のない列にデータをロードしようとすると、InterSystems IRIS では SQLCODE -99 エラーが発生します。

適切な特権があるかどうかを確認するには、[%CHECKPRIV](#) を使用します。ユーザにテーブルの特権を割り当てるには、[GRANT](#) を使用します。詳細は、“[特権](#)”を参照してください。

### ゲートウェイ特権

LOAD DATA を実行するには、ユーザがサーバ上の JVM へのアクセス権を持っている必要があります。InterSystems IRIS の[外部言語サーバ](#)へのアクセスと同様、そのような接続には特権が必要です。JVM に適切にアクセスするには、ユーザは `%Gateway_Object:USE` 特権を持っている必要があります。

## トランザクションの考慮事項

### アトミック性

LOAD DATA はアトミック処理です。他のアトミック処理と同様、LOAD DATA コマンドは、既定でトランザクションを使用することにより、成功しなかった場合に完全にロールバックされます。このコマンドが完了できなかった場合、データは挿入されず、データベースは LOAD DATA を発行する前の状態に戻ります。他のアトミック処理とは異なり、このルールには、注目すべき例外があります。これらの例外は以下のとおりです。

- LOAD BULK DATA と LOAD [%NOJOURN DATA](#) は、トランザクションを開始しません。
- LOAD DATA は、USING 節内で [jsonoption](#) を使用できるようにするため、アトミック処理の中でも独特です。maxerrors JSON を使用して、LOAD DATA コマンド中の挿入エラーに対する上限を指定できます。この上限に達すると、トランザクションは失敗し、データベースは LOAD DATA を発行する前の状態に戻ります。この制限に達しない場合は、トランザクションが成功し、正常にロードされたデータがデータベースに表示されます。ただし、ロードに失敗したデータはデータベースに表示されません。

現在のプロセスに対するこの既定は、[SET TRANSACTION %COMMITMODE](#) を呼び出すことによって SQL 内で変更できます。現在のプロセスに対するこの既定は、次の構文を使用して SetOption() メソッドを呼び出すことによって ObjectScript 内で変更できます。

```
SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)
```

以下の `intval` 整数オプションを使用できます。

- ・ 1 または `IMPLICIT` (自動コミットがオン – 既定) – `LOAD DATA` の呼び出しで独自のトランザクションを開始および終了します。
- ・ 2 または `EXPLICIT` (自動コミットがオフ) – 進行中のトランザクションがない場合は、`LOAD DATA` コマンドによってトランザクションは自動的に開始されます。ただし、`COMMIT` または `ROLLBACK` で明示的にトランザクションを終了する必要があります。EXPLICIT モードでは、トランザクションあたりのデータベース操作の数は、ユーザ定義です。
- ・ 0 または `NONE` (自動トランザクションなし) – `LOAD DATA` を呼び出してもトランザクションは開始されません。`LOAD DATA` 操作の失敗により、行の一部が挿入されたり挿入されなかったりすることで、データベースが整合性のない状態になる可能性があります。このモードでトランザクションのサポートを提供するには、`START TRANSACTION` を使用してトランザクションを開始し、`COMMIT` または `ROLLBACK` を使用してトランザクションを終了する必要があります。

**シャード・テーブル**は常に、自動トランザクションなしのモードに設定されます。つまり、シャード・テーブルに対する挿入、更新、および削除はすべて、トランザクションの範囲外で実行されます。

現在のプロセスのアトミック性設定を確認するには、以下の `ObjectScript` の例のように、`GetOption("AutoCommit")` メソッドを使用します。

### ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }
```

## 例

### CSV ファイルから SQL テーブルおよびビューへのデータのロード

この例は、CSV (comma-separated value) ファイルに格納されているデータを既存のテーブルおよびビューにロードする方法を示しています。

データのロード先のテーブルを作成します。このテーブルには、メンバシップ・データを指定する 3 つのフィールド、メンバ ID、月単位でのメンバシップ期間の長さ、およびメンバが住んでいる米国の州 (2 文字の州の省略形を使用) が含まれます。

### SQL

```
CREATE TABLE Sample.Members (
    MemberId INT PRIMARY KEY,
    MemberTerm INT DEFAULT 12,
    MemberState CHAR(2))
```

これらのデータ・レコードをテキスト・ファイルにコピーします。このファイルをローカル・マシンに保存し、`members.csv` という名前を付けます。このファイルは、メンバシップ ID とメンバの州の値を指定します。2 番目の行は、値が配置される位置の前に挿入されたプレースホルダのコンマによって示されているように、値が欠落しています。

```
6138830,MA
1720936,
4293608,NH
```



LOAD DATA を使用して Sample.Members テーブルにデータをロードします。ここに示されているパスを、ファイルを保存したパスに置き換えます。

```
LOAD DATA FROM FILE 'C://temp/members.csv' INTO Sample.Members (MemberId,MemberState)
```

データを確認します。MemberId 列と MemberState 列が生成されています。ソース・ファイルには MemberTerm 列のデータは含まれていなかったため、これらの列の値は既定の 12 となっています。欠落している行の値は、NULL 値としてロードされます。

## SQL

```
SELECT * FROM Sample.Members
```

MemberId	MemberTerm	MemberState
6138830	12	MA
1720936	12	
4293608	12	NH

LOAD DATA の全体としての SQLCODE 結果は 0 (成功) であるため、LOAD DATA はデータが欠落していても SQLCODE エラーを報告しません。以下の場合、LOAD DATA 操作は成功と見なされます。

- ・ LOAD DATA がソースにアクセスできる。
- ・ ターゲット・テーブルが存在する。
- ・ LOAD DATA 操作が有効である。例えば、操作により正しい数の列が指定され、それらの列名がターゲット・テーブルに存在する場合などです。

個々の行の SQLCODE エラーを、LOAD DATA 操作に関するその他の情報と共に表示するには、%SQL\_Diag.Result テーブルと %SQL\_Diag.Message テーブルを使用します。詳細は、“[ロードされたデータの診断ログの表示](#)”を参照してください。

最新の LOAD DATA 操作のメッセージを表示します。州の省略形が欠落している行は、SQLCODE エラー -104 を報告しています。表示されている結果は読みやすいように切り詰められています。

## SQL

```
SELECT actor,message,severity,sqlcode
FROM %SQL_Diag.Message
WHERE diagResult =
  (SELECT TOP 1 resultId
   FROM %SQL_Diag.Result
   ORDER BY resultId DESC)
```

アクター	メッセージ	重大度	SQLCODE
server	{"resultid":"1","bufferrowcount":500, ...}	info	0
FileReader	Reader Complete: Total Input file read time: 23 ms,	completed	0
JdbcWriter	[SQLCODE: <-104>:<Field validation failed in INSERT>] [%msg: ...(Varchar Value: 'state...'Length: 5) > maxlen: (2)>]	error	-104
JdbcWriter	Writer Complete: Total write time: 72 ms,	completed	0

テーブルからビューを作成する場合は、ビューを使用してデータをテーブルにロードすることもできます。Sample.Members テーブルのメンバシップ ID と状態の列のみを表示するビューを作成します。

## SQL

```
CREATE VIEW Sample.VMem (Mid,State) AS SELECT MemberId,MemberState FROM Sample.Members
```

これらの追加データ・レコードをテキスト・ファイルにコピーします。このファイルをローカル・マシンに保存し、members2.csv という名前を付けます。

```
6785674,VT
4564563,RI
4346756,ME
```

LOAD DATA を使用して、作成したビューを使用することにより、この新しい CSV データをテーブルにロードします。

```
LOAD DATA FROM FILE 'C://temp/members2.csv' INTO Sample.VMem(Mid,State)
```

ビューによって返されたデータを確認します。ここには、ロードされた両方の CSV ファイルのデータが含まれています。

## SQL

```
SELECT * FROM Sample.VMem
```

Mid	State
6138830	MA
1720936	
4293608	NH
6785674	VT
4564563	RI
4346756	ME

ベース・テーブルのデータを確認します。ここには、両方の CSV ファイルの結合された列データが含まれています。2 度目にロードされた CSV ファイルでも、MemberTerm 列の値には既定値の 12 が適用されています。

## SQL

```
SELECT * FROM Sample.Members
```

MemberId	MemberTerm	MemberState
6138830	12	MA
1720936	12	
4293608	12	NH
6785674	12	VT
4564563	12	RI
4346756	12	ME

ビューとテーブルを削除します。

## SQL

```
DROP VIEW Sample.VMem
```

## SQL

```
DROP TABLE Sample.Members
```

## トラブルシューティング

## ロードされたデータの診断ログの表示

LOAD DATA を呼び出すたびに、SQL Diagnostic Logsのエントリ (管理ポータルの [システムオペレーション]→[システムログ]→[SQL Diagnostic Logs] で表示できます) と %SQL\_Diag.Result テーブル内の新しい行の両方が生成されます。このテーブルには、操作に関する診断情報が含まれます。これらの行は、SELECT クエリを使用して表示できます。例えば、以下のクエリは、最近の 5 つの LOAD DATA 呼び出しを返します。

## SQL

```
SELECT TOP 5 * FROM %SQL_Diag.Result ORDER BY createTime DESC
```

返されるテーブルには、以下の列が含まれます。

- ・ ID – ログ・エントリの整数の ID。この値は、テーブルの主キーです。
- ・ resultId – ID と同じ。
- ・ createTime – LOAD DATA 操作が行われ、ログ・エントリ行が作成された時点のタイムスタンプ。タイムスタンプは、ローカル時刻ではなく、UTC (協定世界時) です。
- ・ namespace – LOAD DATA 操作が行われたネームスペース。
- ・ processId – LOAD DATA 操作を実行したプロセスの整数の ID。
- ・ user – LOAD DATA 操作を実行したユーザ。
- ・ sqlCode – LOAD DATA 操作全体の SQLCODE。
- ・ inputRecordCount – 正常にロードされたレコード数。
- ・ errorCount – 発生したエラーの数。LOAD DATA コマンドの原因となるエラーや、個々のデータ行のロードまたは書き込みの失敗が含まれます。
- ・ maxErrorCount – LOAD DATA が許容できる行挿入エラーの最大数。これを超えると操作は失敗となります。
- ・ status – LOAD DATA 操作のステータス。LOAD DATA の実行中、ステータスは "In Progress" に設定されます。LOAD DATA 操作が完了すると、ステータスは "Complete" に更新されます。LOAD DATA の実行でエラーが発生すると、ステータスは "Failed" に更新されます。
- ・ statement – この %SQL\_Diag.Result レコードを生成するために実行された SQL 文のテキスト。

%SQL\_Diag.Message は、%SQL\_Diag.Result テーブルにログ記録された各 LOAD DATA 操作の詳細なメッセージ・データを提供します。%SQL\_Diag.Message の diagResult 列は、%SQL\_Diag.Result テーブルの resultId 列への外部キー参照であり、個々の LOAD DATA 操作のメッセージへのアクセスを可能にします。

例えば、このクエリは、resultId 29 を持つ LOAD DATA 操作に関連するすべてのエラー・メッセージを返します。このデータを使用して、テーブルのどの行がロードに失敗したのかを診断できます。

## SQL

```
SELECT *  
FROM %SQL_Diag.Message  
WHERE severity = 'error'  
AND diagResult = 29
```

返されるテーブルには、以下の列が含まれます。

- ・ **ID** – メッセージの整数の ID。この値は、テーブルの主キーです。
- ・ **actor** – 、`FileReader`、`JdbcWriter` など、メッセージを報告したエンティティ。
- ・ **diagResult** – **%SQL\_Diag.Result** テーブルに記録された LOAD DATA ログ・エントリの行 ID。
- ・ **message** – メッセージ・データ。エラーの場合、この列には **SQLCODE** 値と **%msg** テキストが含まれます。
- ・ **messageTime** – ローカル時刻ではなく、UTC (協定世界時) でのメッセージのタイムスタンプ。
- ・ **severity** – メッセージの重大度レベル。重大度は、対応する表示を持つ論理整数です。有効な値は、`"completed"`、`"info"`、`"warning"`、`"error"`、`"abort"` です。
- ・ **sqlcode** – メッセージの **SQLCODE**。`"completed"` または `"info"` の重大度のメッセージは、**SQLCODE** 0 を報告します。`"warning"` または `"error"` の重大度のメッセージは、そのメッセージに関連する **SQLCODE** を報告します。`"abort"` の重大度のメッセージは、**SQLCODE** -400 を報告します。

## 関連項目

- ・ [INSERT](#)
- ・ [CREATE TABLE](#)
- ・ [SQL データのインポートとエクスポート](#)
- ・ [SQLCODE エラー・メッセージ](#)

## LOCK (SQL)

テーブルをロックします。

### 構文

```
LOCK [TABLE] tablename IN EXCLUSIVE MODE [WAIT seconds]
LOCK [TABLE] tablename IN SHARE MODE [WAIT seconds]
```

### 説明

LOCK と LOCK TABLE は同義語です。

LOCK コマンドは、SQL テーブルを明示的にロックします。このテーブルは、ユーザが適切な特権を持つ既存のテーブルである必要があります。tablename が実在しないテーブルである場合、コンパイル・エラーが発生して LOCK が失敗します。tablename が一時テーブルである場合、コマンドは正常に完了しますが、処理は何も実行されません。tablename がビューである場合、SQLCODE -400 エラーが発生してコマンドが失敗します。

UNLOCK コマンドは LOCK 操作を取り消します。明示的な LOCK コマンドは、同じモードで UNLOCK を明示的に発行するまで、またはプロセスが終了するまで有効です。

LOCK を使用してテーブルを複数回ロックすることができますが、明示的にロックした回数と同じ回数だけ明示的に UNLOCK を適用する必要があります。UNLOCK のたびに、対応する LOCK と同じモードを指定する必要があります。

### 特権

LOCK コマンドは特権を必要とする操作です。LOCK IN SHARE MODE を使用する前に、指定されたテーブルに対する SELECT 特権をプロセスに付与しておく必要があります。LOCK IN EXCLUSIVE MODE を使用する前に、指定されたテーブルに対する INSERT、UPDATE、および DELETE 特権をプロセスに付与しておく必要があります。IN EXCLUSIVE MODE では、テーブルの 1 つ以上のフィールドに対する INSERT、または UPDATE 特権が必要です。特権がない場合は、SQLCODE -99 エラー（特権違反）が返されます。[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが必要な特権を持っているかどうかを確認できます。`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを呼び出すことにより、指定のユーザが必要な特権を持っているかどうかを確認できます。特権の割り当てについては、“[GRANT](#)” コマンドを参照してください。

これらの特権はロック取得に必要なもので、ロックの性質を定義するものではありません。IN EXCLUSIVE MODE ロックは、ロック所有者に対応する特権があるかどうかにかかわらず、他のプロセスから INSERT、UPDATE、および DELETE の操作が実行できないようにします。

### ロック・モード

LOCK では、SHARE と EXCLUSIVE の 2 つのモードがサポートされます。これらのロック・モードに相互依存性はありません。SHARE ロックと EXCLUSIVE ロックの両方を同一のテーブルに適用できます。EXCLUSIVE モードのロックは、EXCLUSIVE モードの UNLOCK でのみアンロックできます。SHARE モードのロックは、SHARE モードの UNLOCK でのみアンロックできます。

- ・ LOCK mytable IN SHARE MODE では、他のプロセスから mytable に対する EXCLUSIVE ロックの発行、DROP TABLE などの DDL 操作の呼び出しを実行できません。
- ・ LOCK mytable IN EXCLUSIVE MODE では、他のプロセスから mytable に対する EXCLUSIVE ロックや SHARE ロックの発行や、挿入、更新、削除操作の実行、DROP TABLE などの DDL 操作の呼び出しを実行できません。

LOCK では、テーブルに対する読み取りアクセスが許可されます。他のプロセスからの、READ UNCOMMITTED モード（既定は SELECT モード）にあるテーブルの SELECT の実行は、どの LOCK モードでも阻止されません。

### ロックの競合

- ・ 別のユーザが IN EXCLUSIVE MODE でテーブルを既にロックしている場合、どのロック・モードも適用できません。

- 別のユーザが IN SHARE MODE でテーブルを既にロックしている場合、IN SHARE MODE でテーブルにさらにロックを適用することはできますが、IN EXCLUSIVE MODE のロックは適用できません。

このような LOCK 競合がある場合、SQLCODE -110 エラーが生成され、%msg " 'Sample.Person' " が生成されます。

## ロック・タイムアウト

LOCK は、タイムアウトが発生するまで、指定された SQL テーブル・ロックの獲得を試行します。タイムアウトが発生すると、LOCK は SQLCODE -110 エラーを生成します。

- WAIT seconds を指定している場合、その秒数が経過すると SQL テーブル・ロック・タイムアウトが発生します。
- 指定していない場合、SQL テーブル・ロック・タイムアウトは、現在のプロセスの SQL タイムアウトが経過すると発生します。現在のプロセスのロック・タイムアウトは、\$SYSTEM.SQL.Util.SetOption() メソッドの ProcessLockTimeout オプションを使用して設定できます。SQL コマンド [SET OPTION](#) に LOCK\_TIMEOUT オプションを指定して、現在のプロセスにロック・タイムアウトを設定することもできます。(SET OPTION は SQL シェルからは使用できません。) 現在のプロセスの SQL ロック・タイムアウトは、既定で、システム全体に適用される SQL ロック・タイムアウトになります。
- 指定していない場合、SQL テーブル・ロック・タイムアウトは、システム全体の SQL タイムアウトが経過すると発生します。このシステム全体の既定は、10 秒です。システム全体のロック・タイムアウトは、以下の 2 つの方法で設定できます。
  - \$SYSTEM.SQL.Util.SetOption() メソッドの LockTimeout オプションを使用します。これは新規のプロセスに適用される、システム全体のロック・タイムアウトの既定を即時に変更します。さらに、現在のプロセスに適用される ProcessLockTimeout をこの新しいシステム全体の値に再設定します。システム全体のロック・タイムアウトを設定しても、現在実行中の他のプロセスの ProcessLockTimeout 設定には影響しません。
  - 管理ポータルを使用して、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[ロックタイムアウト(秒)]の現在の設定を表示して編集します。これは、構成変更を保存した後に開始する新規のプロセスに適用される、システム全体のロック・タイムアウトの既定を変更します。現在実行中のプロセスには影響しません。

現在のシステム全体のロック・タイムアウト値を返すには、\$SYSTEM.SQL.Util.GetOption("LockTimeout") メソッドを呼び出します。

現在のプロセスのロック・タイムアウト値を返すには、\$SYSTEM.SQL.Util.GetOption("ProcessLockTimeout") メソッドを呼び出します。

## トランザクション処理

LOCK 操作は、トランザクションの構成部分ではありません。LOCK が発行されたトランザクションをロール・バックしてもアンロックされません。UNLOCK は、現在のトランザクションの最後に、または即時に発生するように定義できます。

## その他のロック操作

ALTER TABLE や DELETE TABLE などの DDL 操作の多くは、排他的なテーブル・ロックを取得します。

INSERT、UPDATE、および DELETE コマンドもロックを実行します。既定では、現在のトランザクションの実行中にレコード・レベルでロックが適用されます。これらのコマンドのいずれかが十分な数の (既定の設定では 1000 個) レコードをロックした場合、このロックは自動的にテーブル・ロックに昇格します。LOCK コマンドを使用すると、テーブル・レベルのロックを明示的に設定できるため、データ・リソースのロック制御能力が向上します。キーワード %NOLOCK を指定すると、INSERT、UPDATE、および DELETE で LOCK をオーバーライドできます。

LOCK\_TIMEOUT オプションが指定された InterSystems SQL [SET OPTION](#) は、INSERT、UPDATE、DELETE、または SELECT 操作の現在のプロセスに対してタイムアウトを設定します。

InterSystems SQL は、\$SYSTEM.SQL.Util.SetOption() メソッドの CachedQueryLockTimeout オプションをサポートしています。

## 引数

### tablename

ロックする**テーブル**の名前。tablename は既存のテーブルでなければなりません。tablename は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。[スキーマ検索パス](#)は無視されます。

### IN EXCLUSIVE MODE/IN SHARE MODE

キーワード句 IN EXCLUSIVE MODE は、通常の InterSystems IRIS ロックを作成します。キーワード句 IN SHARE MODE は、共有 InterSystems IRIS ロックを作成します。

### WAIT seconds

ロック取得の試行期間を指定する秒数 (オプション)。この秒数を経過すると**タイムアウト**になります。省略した場合、既定のタイムアウトが適用されます。

## 例

以下の例は、テーブルを作成してロックします。

### SQL

```
CREATE TABLE mytest (  
  ID NUMBER(12,0) NOT NULL,  
  CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),  
  WORK_START DATE DEFAULT SYSDATE)
```

### SQL

```
LOCK mytest IN EXCLUSIVE MODE WAIT 4
```

管理ポータルから実行される SQL プログラムにより、プログラムの実行後に直ちに終了するプロセスが生成されます。このように、ロックはほとんど即座に解除されます。このため、ロックの競合を監視するには、まず、同じネームスペースで SQL シェルを実行しているターミナルから LOCK mytest IN EXCLUSIVE MODE コマンドを発行します。次に上のロック・プログラムを実行します。ターミナル SQL シェルから UNLOCK mytest IN EXCLUSIVE MODE を発行します。次に上のロック・プログラムを再実行します。

## 関連項目

- ・ [UNLOCK](#)
- ・ [INSERT](#)、[UPDATE](#)、[DELETE](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)



# OPEN (SQL)

カーソルをオープンします。

## 構文

```
OPEN cursor-name
```

## 概要

OPEN 文は、カーソルの [DECLARE](#) 文で指定されているパラメータに従って[カーソル](#)をオープンします。オープンすると、カーソルを取得できます。オープンしたカーソルは、クローズする必要があります。

- ・ 宣言されていないカーソルをオープンしようとすると、SQLCODE -52 エラーが返されます。
- ・ 既にオープンしているカーソルをオープンしようとすると、SQLCODE -101 エラーが返されます。
- ・ オープンしていないカーソルをフェッチまたはクローズしようとすると、SQLCODE -102 エラーが返されます。

OPEN を正常に実行できると、結果セットが空の場合でも SQLCODE = 0 が設定されます。

SQL 文として、埋め込み SQL からのみサポートされます。同様の操作は、ODBC でも ODBC API を使用してサポートされます。

## 引数

### cursor-name

既に宣言されている、カーソルの名前。カーソル名は DECLARE 文で指定されています。カーソル名は、大文字と小文字を区別します。

## 例

以下は、EmpCursor という名前のカーソルをオープンしてクローズする埋め込み SQL の例です。

### ObjectScript

```
SET name="LastName,FirstName",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'A')
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
IF SQLCODE != 0 { WRITE "Open error: ",SQLCODE
    QUIT }
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
WRITE !,"AFTER: Name=",name," State=",state
```

## 関連項目

- ・ [CLOSE](#)、[DECLARE](#)、[FETCH](#)
- ・ [SQL カーソル](#)
- ・ [SQLCODE エラー・メッセージ](#)

# PURGE CACHED QUERIES (SQL)

1 つ以上のクエリ・キャッシュを削除します。

## 構文

```
PURGE [CACHED] QUERIES  
PURGE [CACHED] QUERIES BY AGE n  
PURGE [CACHED] QUERIES BY TABLE table-name  
PURGE [CACHED] QUERIES BY NAME class-name [, class-name]
```

## 説明

PURGE CACHED QUERIES コマンドは、指定されたスコープ内の定義済みのクエリ・キャッシュを削除します。

- ・ PURGE CACHED QUERIES は、現在のネームスペース内のすべてのクエリ・キャッシュを削除します。
- ・ PURGE CACHED QUERIES BY AGE *n* は、現在のネームスペース内のクエリ・キャッシュの中で、過去 *n* 日以内に使用（準備）されていないものをすべて削除します。*n* の値に 0 を指定すると、現在のネームスペースのすべてのクエリ・キャッシュを削除します。
- ・ PURGE CACHED QUERIES BY TABLE *table-name* は、指定されたテーブルを参照するクエリ・キャッシュをすべて削除します。クエリが複数のテーブルを参照する場合、これらのテーブルごとに 1 つのクエリ・キャッシュが生成されて表示されます。これらのテーブルのいずれかに対して PURGE CACHED QUERIES BY TABLE を発行すると、このクエリ・キャッシュがこれらのすべてのテーブルから削除されます。
- ・ PURGE [CACHED] QUERIES BY NAME *class-name* は、クエリ・キャッシュ・クラス名によって指定されたクエリ・キャッシュを削除します。複数のクエリ・キャッシュを、コンマ区切りのリストで指定することができます。リストされたクエリ・キャッシュは異なるテーブルを参照できますが、すべて現在のネームスペース内にある必要があります。クエリ・キャッシュ名では大文字と小文字が区別されます。

CACHED キーワードはオプションです。

指定された *class-name* が存在しない場合、または指定された大文字/小文字が正しくない場合、そのクラス名はスキップされ、コマンドはリスト内の次のクエリ・キャッシュを削除します。無効なクラス名に対しては何の処理も実行されず、エラーは生成されません。指定されたテーブルに関連付けられたクエリ・キャッシュがない場合、またはテーブルが存在しない場合、何の処理も実行されず、エラーは生成されません。

## 引数

*n*

クエリ・キャッシュが最後に使用されてからの日数（整数）。引用符付きの文字列として指定します。

*table-name*

クエリ・キャッシュがある既存の**テーブル**の名前。*table-name* は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、**既定のスキーマ名**が使用されます。

*class-name*

クエリ・キャッシュ・クラス名またはクエリ・キャッシュ・クラス名のコンマ区切りのリスト。クエリ・キャッシュ・クラス名では大文字と小文字が区別されます。

## 例

以下の例では、名前で指定したクエリ・キャッシュを削除します。

```
PURGE CACHED QUERIES BY NAME %sqlcq.USER.cls2
```

以下の例では、過去 2 日間に使用されていないクエリ・キャッシュをすべて削除します。

```
PURGE CACHED QUERIES BY AGE "2"
```

## 関連項目

- ・ `%SYSTEM.SQL.PurgeCQClass()` は、クエリ・キャッシュ・クラスの名前が指定されると、クエリ・キャッシュを削除します。個々のクエリ・キャッシュを削除するには、この関数を使用する必要があります。

## REVOKE (SQL)

ユーザまたはロールから特権を削除します。

### 構文

#### 特権の削除

```
REVOKE admin-privilege FROM grantee
```

#### ロールの削除

```
REVOKE role FROM grantee
```

#### オブジェクト特権の削除

```
REVOKE [GRANT OPTION FOR] object-privilege ON object-list  
FROM grantee [CASCADE | RESTRICT] [AS grantor]  
REVOKE [GRANT OPTION FOR] SELECT ON CUBE[S] object-list  
FROM grantee
```

#### テーブルレベル/列レベルの特権の削除

```
REVOKE column-privilege (column-list) ON table  
FROM grantee [CASCADE | RESTRICT]
```

### 概要

REVOKE 文は、指定したテーブル、ビュー、列、またはその他のエンティティ上で指定タスクを実行する特権を、ユーザまたはロールから削除します。REVOKE では、ユーザからロールの割り当てを削除することもできます。REVOKE は [GRANT](#) コマンドの逆の動作を行います。特権全般の詳細は、“GRANT” コマンドを参照してください。

特権を削除するには、以下のいずれかである必要があります。

- ・ 特権を付与したユーザである。
- ・ 特権を [CASCADE 操作](#)によって削除する。
- ・ 別のユーザによって付与された特権を削除する。これを実行できるのは、[%Admin\\_Secure](#) 管理リソースに対する USE 許可を持っているか、システムに対する完全なセキュリティ特権を持っているユーザとしてログインしている場合です。

指定したユーザ、ユーザのリスト、またはすべてのユーザ (\* 構文の使用) からロールまたは特権を削除できます。

REVOKE はすぐに作成および実行され、通常一度しか実行されないため、InterSystems IRIS では、ODBC、JDBC、またはダイナミック SQL での REVOKE にはクエリ・キャッシュは作成されません。

REVOKE は、実際の削除を (例えば、指定した特権が付与されていない、または削除済みであるなどの理由で) 実行できなくても、正常に完了します。ただし、REVOKE の操作中にエラーが発生した場合、SQLCODE は負の数に設定されます。

#### ロールの削除

ロールは、SQL の GRANT コマンドおよび REVOKE コマンド、または `^SECURITY` InterSystems IRIS システム・セキュリティを使用して付与または削除できます。REVOKE を使用すると、ユーザからロールを削除することや、別のロールからロールを削除することができます。InterSystems IRIS システム・セキュリティを使用して、別のロールにロールを付与することや、別のロールからロールを削除することはできません。`$ROLES` 特殊変数は、ロールに付与されたロールは表示しません。

REVOKE では、削除するロールを 1 つまたはコンマ区切りのリストで指定できます。REVOKE では、指定したユーザ (またはロール)、ユーザ (またはロール) のリスト、またはすべてのユーザ (\* 構文を使用) から 1 つまたは複数のロールを削除できます。

GRANT コマンドでは、存在しないロールをユーザに付与できます。REVOKE を使用すると、存在しないロールを既存のユーザから削除できます。ただし、ロール名は、ロールの付与に使用した文字と同じ大文字/小文字の区別で指定する必要があります。

存在していないユーザまたはロールから既存のロールを削除しようとすると、InterSystems IRIS は SQLCODE -118 エラーを発行します。SuperUser でない場合に、自身が所有していないロールを削除しようとし、さらにそのロールに対する ADMIN OPTION も保有していない場合、InterSystems IRIS は SQLCODE -112 エラーを発行します。

## オブジェクト特権の削除

オブジェクト特権は、ユーザまたはロールに特定のオブジェクトに対する特権を与えます。object-list に関する object-privilege を grantee から削除します。object-list では、現在のネームスペース内の 1 つまたは複数のテーブル、ビュー、ストアド・プロシージャ、またはキューブを指定できます。コンマ区切りのリストを使用することで、1 つの REVOKE 文で、複数のオブジェクトに関する複数のオブジェクト特権を複数のユーザおよび/またはロールから削除できます。

object-list の値にアスタリスク (\*) ワイルドカードを使用すると、現在のネームスペース内のすべてのオブジェクトから object-privilege を削除できます。例えば、REVOKE SELECT ON \* FROM Deborah は、このユーザから、すべてのテーブルおよびビューに対する SELECT 特権を削除します。REVOKE EXECUTE ON \* FROM Deborah は、このユーザから、非公開でないすべてのストアド・プロシージャに対する EXECUTE 特権を削除します。

SCHEMA schema-name を object-list 値として使用すると、現在のネームスペース内の指定されたスキーマのすべてのテーブル、ビュー、およびストアド・プロシージャに対する object-privilege を削除できます。例えば、REVOKE SELECT ON SCHEMA Sample FROM Deborah は、このユーザから、Sample スキーマ内のすべてのオブジェクトに対する SELECT 特権を削除します。複数のスキーマをコンマ区切りのリストとして指定できます。例えば、REVOKE SELECT ON SCHEMA Sample,Cinema FROM Deborah は、Sample スキーマと Cinema スキーマの両方にあるすべてのオブジェクトに対する SELECT 特権を削除します。

ユーザまたはロールからオブジェクト特権を削除できます。ロールから特権を削除すると、ロールを通してしかその特権を持たないユーザからは、その特権がなくなります。特権がなくなったユーザは、そのオブジェクト特権を必要とする既存クエリ・キャッシュを実行できなくなります。

REVOKE がオブジェクト特権を呼び出すと、正常に完了し、SQLCODE は 0 に設定されます。REVOKE は、実際の削除を (例えば、指定したオブジェクト特権が付与されていない、または削除済みであるなどの理由で) 実行しない場合でも、正常に完了し、SQLCODE は 100 (これ以上データがない) に設定されます。REVOKE の操作中にエラーが発生した場合、SQLCODE は負の数に設定されます。

キューブはスキーマ名で修飾していない SQL 識別子です。キューブの object-list を指定するには、CUBE (または CUBES) キーワードを指定する必要があります。キューブは SELECT 特権しか持てないため、キューブからは SELECT 特権しか削除できません。

オブジェクト特権は、以下のいずれかによって削除できます。

- REVOKE コマンド。
- \$SYSTEM.SQL.Security.RevokePrivilege() メソッド。
- InterSystems IRIS システム・セキュリティの使用。管理ポータルに進み、[システム管理]、[セキュリティ]、[ユーザ] (または [システム管理]、[セキュリティ]、[ロール]) の順に選択し、目的のユーザまたはロールについて [編集] を選択した後、[SQLテーブル] タブまたは [SQLビュー] タブを選択します。ドロップダウン・リストから目的の [ネームスペース] を選択します。目的のテーブルまで下方にスクロールしてから、[取り消す] をクリックして特権を削除します。

%CHECKPRIV コマンドを呼び出すことにより、現在のユーザが指定されたオブジェクト特権を持っているかどうかを確認できます。\$SYSTEM.SQL.Security.CheckPrivilege() メソッドを呼び出すことにより、指定のユーザが指定されたテーブルレベルのオブジェクト特権を持っているかどうかを確認できます。

## オブジェクト所有者特権の削除

オブジェクトの所有者から SQL オブジェクトに対する特権を削除しても、所有者は引き続き暗黙的にそのオブジェクトに対する特権を持ちます。オブジェクトの所有者からオブジェクトに対するすべての特権を完全に削除するには、そのオブジェクトを別の所有者または所有者なしに変更する必要があります。

## テーブルレベルおよび列レベルの特権の削除

REVOKE を使用すると、テーブルレベルの特権または列レベルの特権の付与を無効にできます。テーブルレベルの特権では、テーブル内のすべての列へのアクセスが提供されます。列レベルの特権では、テーブル内の指定された列へのアクセスが提供されます。テーブル内のすべての列に対する列レベルの特権を付与することは、テーブルレベルの特権を付与することと機能的に同等です。ただし、まったく同じというわけではありません。列レベルの REVOKE は、列レベルで付与されている特権のみを削除できます。テーブルに対してテーブルレベルの特権を与えることはできず、この特権を、1 つ以上の列に対する列レベルで削除することもできません。この場合、REVOKE 文は付与された特権に影響を与えません。

## CASCADE または RESTRICT

InterSystems IRIS では、REVOKE object-privilege の動作を指定する、オプションの CASCADE および RESTRICT キーワードをサポートしています。いずれのキーワードも指定されていない場合、既定は RESTRICT です。

CASCADE または RESTRICT を使用すると、ユーザから object-privilege または column-privilege を削除する場合に、WITH GRANT OPTION を介して、その特権を受け取った他のユーザからもその特権を削除するかどうかを指定できます。CASCADE は、そのように関連付けられたすべての特権を削除します。RESTRICT (既定) は、関連付けられた特権が検出された場合、REVOKE を失敗させます。代わりに、SQLCODE -126 エラー “RESTRICT の REVOKE に失敗しました” を設定します。

これらのキーワードの使用法を以下の例に示します。

### SQL

```
--UserA
GRANT Select ON MyTable TO UserB WITH GRANT OPTION
```

### SQL

```
--UserB
GRANT Select ON MyTable TO UserC
```

### SQL

```
--UserA
REVOKE Select ON MyTable FROM UserB
-- This REVOKE fails with SQLCODE -126
```

### SQL

```
--UserA
REVOKE Select ON MyTable FROM UserB CASCADE
-- This REVOKE succeeds
-- It revokes this privilege from UserB and UserC
```

CASCADE および RESTRICT は、MyTable を参照する UserB によって作成されたビューには影響を与えないことに注意してください。

## クエリ・キャッシュへの影響

特権またはルールを削除すると、システムのすべてのクエリ・キャッシュが更新され、その特権の変更が反映されます。しかし、データベース・サーバへの ECP 接続の停止などにより、アクセスできないネームスペースがあると、REVOKE は正常に終了しますが、そのネームスペースのクエリ・キャッシュの操作は実行されません。これは、REVOKE が、接続できないネームスペースのクエリ・キャッシュを更新できず、クエリ・キャッシュ・レベルで特権を削除することができないためです。エラーは発行されません。



接続できなかったデータベース・サーバが復旧すると、そのネームスペースにあるクエリ・キャッシュの特権が正しくなくなっている可能性があります。ネームスペースがアクセス不能なときにロールまたは特権が削除された可能性がある場合は、そのネームスペースにあるクエリ・キャッシュを削除することをお勧めします。

## InterSystems IRIS セキュリティ

REVOKE コマンドは特権を必要とする操作です。埋め込み SQL で REVOKE を使用するには、その特権の付与者であるか、あるいは `%Admin_Secure` 管理リソースに対する USE 許可、またはシステムに対する完全なセキュリティ特権を持っているユーザとしてログインしている必要があります。特権がない場合は、SQLCODE -99 エラー（特権違反）が返されます。

`$SYSTEM.Security.Login()` メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

`$SYSTEM.Security.Login` メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“インターシステムズ・クラス・リファレンス”の“**%SYSTEM.Security**”を参照してください。

## 引数

### admin-privilege

許可されている管理者レベル特権、または管理者レベル特権のコンマで区切られたリスト。使用可能な syspriv オプションには、16 個のオブジェクト定義特権と 4 つのデータ変更特権が含まれています。

それらのオブジェクト定義特権は、`%CREATE_FUNCTION`、`%DROP_FUNCTION`、`%CREATE_METHOD`、`%DROP_METHOD`、`%CREATE_PROCEDURE`、`%DROP_PROCEDURE`、`%CREATE_QUERY`、`%DROP_QUERY`、`%CREATE_TABLE`、`%ALTER_TABLE`、`%DROP_TABLE`、`%CREATE_VIEW`、`%ALTER_VIEW`、`%DROP_VIEW`、`%CREATE_TRIGGER`、`%DROP_TRIGGER` です。別の方法としては、`%DB_OBJECT_DEFINITION` を指定すると、この 16 個すべてのオブジェクト定義特権を削除できます。

データ変更特権は、`INSERT`、`UPDATE`、および `DELETE` 操作のための `%NOCHECK`、`%NOINDEX`、`%NOLOCK`、`%NOTRIGGER` 特権です。

### grantee

SQL システム特権、SQL オブジェクト特権、またはロールを持つ 1 人以上のユーザのリスト。有効な値は、コンマで区切られたユーザまたはロールのリスト、または `"*"` です。アスタリスク (\*) は、現在定義されているユーザのうち、付与者であるユーザをすべて指定します。AS grantor 節を使用するには、`%All` ロールまたは `%Admin_Secure` リソースを持っている必要があります。

### AS grantor

この節では、元の付与者の名前を指定することによって、別のユーザによって付与された特権を削除できます。有効な grantor の値は、ユーザ名、コンマで区切られたユーザ名のリスト、または `"*"` です。アスタリスク (\*) は、現在定義されているユーザのうち、付与者であるユーザをすべて指定します。AS grantor 節を使用するには、`%All` ロールまたは `%Admin_Secure` リソースを持っている必要があります。

### role

ロールまたはコンマで区切られたロールのリストで、ユーザにより特権が無効とされるもの。

### object-privilege

許可されている基本レベル特権、または基本レベル特権のコンマで区切られたリスト。`%ALTER`、`DELETE`、`SELECT`、`INSERT`、`UPDATE`、`EXECUTE`、および `REFERENCES` のうち 1 つ以上を使用してリストを構成できます。すべての権限



を削除する場合は、引数の値として "ALL [PRIVILEGES]" または "\*" のいずれかを使用します。キューブに付与できる特権は SELECT のみなので、キューブから削除できる特権も SELECT のみです。

### object-list

削除する object-privilege (複数の場合もあります) を持つ 1 つ以上のテーブル、ビュー、ストアド・プロシージャ、またはキューブのコンマ区切りリスト。SCHEMA キーワードを使用すると、指定されたスキーマ内の全オブジェクトからの object-privilege の削除を指定できます。 "\*" を使用すると、現在のネームスペース内の全オブジェクトからの object-privilege の削除を指定できます。

### column-privilege

column-list にリストされた 1 つ以上の列から削除される基本レベル特権。指定できるオプションは、SELECT、INSERT、UPDATE、および REFERENCES です。

### column-list

コンマで区切り、括弧で囲んだ 1 つ以上の列名のリスト。

### table

column-list 列を含む テーブル またはビューの名前。

## 例

以下の埋め込み SQL の例は、2 人のユーザと 1 つのロールを作成し、そのロールをユーザに割り当てます。その後、アスタリスク (\*) 構文を使用してすべてのユーザからそのロールを削除します。ユーザまたはロールが既に存在する場合は、CREATE 文は SQLCODE -118 エラーを発行します。ユーザが存在しない場合、GRANT または REVOKE 文は SQLCODE -118 エラーを発行します。ユーザが存在するが、ロールがない場合、GRANT または REVOKE 文は SQLCODE 100 を発行します。ユーザおよびロールが存在する場合、GRANT または REVOKE 文は SQLCODE 0 を発行します。これは、ロールの付与または削除が既に行われている場合、または付与されたことのないロールを削除しようとしている場合でも同様です。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(CREATE USER User1 IDENTIFY BY fredpw)
&sql(CREATE USER User2 IDENTIFY BY barneypw)
WRITE !,"CREATE USER error code: ",SQLCODE
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE error code: ",SQLCODE
&sql(GRANT workerbee TO User1,User2)
WRITE !,"GRANT role error code: ",SQLCODE
&sql(REVOKE workerbee FROM *)
WRITE !,"REVOKE role error code: ",SQLCODE
```

以下の例では、あるユーザ (Joe) が特権を与え、別のユーザ (John) が AS grantor 節を使用してその特権を削除しています。

### SQL

```
/* User Joe */
GRANT SELECT ON Sample.Person TO Michael
```

### SQL

```
/* User John */
REVOKE SELECT ON Sample.Person FROM Michael AS Joe
```

John は、%All ロールまたは %Admin\_Secure リソースを持っている必要があります。

## 関連項目

- ・ SQL 文 : [CREATE USER](#)、[DROP USER](#)、[CREATE ROLE](#)、[DROP ROLE](#)、[GRANT](#)、[%CHECKPRIV](#)
- ・ [SQL のユーザ、ロール、および特権](#)
- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript : [\\$ROLES](#) および [\\$USERNAME](#) 特殊変数

# ROLLBACK (SQL)

トランザクションをロール・バックします。

## 構文

```
ROLLBACK [WORK]
```

```
ROLLBACK TO SAVEPOINT pointname
```

## 概要

ROLLBACK 文は、[トランザクション](#) をロール・バックし、実行したけれどもコミットされていない作業を元に戻し、[\\$TLEVEL](#) トランザクション・レベル・カウンタをディクリメントしてロックを解除します。ROLLBACK を使用すると、以前の一貫性のある状態にデータベースをリストアすることができます。

- ROLLBACK は現在のトランザクションの間に実行されたすべての作業をロール・バックし、[\\$TLEVEL](#) トランザクション・レベル・カウンタをゼロにリセットして、すべてのロックを解除します。これにより、データベースはトランザクションが開始される前の状態にリストアされます。ROLLBACK 文と ROLLBACK WORK 文は同等です。両者には互換性があります。
- ROLLBACK TO SAVEPOINT pointname は、指定されたセーブポイント以降に実行されたすべての作業をロール・バックし、元に戻したセーブポイントの数だけ [\\$TLEVEL](#) トランザクション・レベル・カウンタをディクリメントします。すべてのセーブポイントがロール・バックまたはコミットされ、トランザクション・レベル・カウンタがゼロにリセットされると、トランザクションは完了します。指定されたセーブポイントが存在しない場合、または既にロール・バックされている場合、ROLLBACK は [SQLCODE -375](#) エラーを発行し、現在のトランザクション全体をロール・バックします。

ROLLBACK TO SAVEPOINT では、pointname を指定する必要があります。これを実行しないと、[SQLCODE -301](#) エラーが返されます。

セーブポイントの設定方法の詳細は、["SAVEPOINT"](#) を参照してください。

トランザクションの動作が正常完了できなかった場合は、[SQLCODE -400](#) エラーが発行されます。

## ロールバック対象外

以下のものは ROLLBACK 操作の影響を受けません。

- ロール・バックでは既定クラスの IDKey カウンタをディクリメントしません。IDKey は [\\$INCREMENT](#) (または [\\$SEQUENCE](#)) によって自動的に生成されます。そこでは SQL トランザクションとは別にカウントを保持しています。
- ロール・バックでは、クエリ・キャッシュの作成、変更、および削除は元に戻りません。これらの処理は、トランザクションの一部として処理されません。
- トランザクション中に発生する DDL 操作や[テーブル・チューニング](#)操作は、一時ルーチンを作成したり実行する場合があります。この一時ルーチンは、クエリ・キャッシュと同様に処理されます。つまり、一時ルーチンの作成、コンパイル、および削除はトランザクションの一部として処理されません。一時ルーチンの実行は、トランザクションの一部であると見なされます。

ロールバックされるまたはロールバックされない非 SQL 項目については、ObjectScript の ["TROLLBACK"](#) コマンドを参照してください。

## ロールバックのロギング

ロールバックが発生したことを示すメッセージ、およびロールバック処理中に発生したエラーは、MGR ディレクトリの [messages.log](#) ファイルにログ記録されます。管理ポータルの [\[システム処理\]](#)→[\[システムログ\]](#)→[\[メッセージログ\]](#) オプションを使用して、[messages.log](#) を表示できます。

## 一時停止されたトランザクション

%SYSTEM.Process クラスの TransactionsSuspended() メソッドを使用して、プロセスの現在のすべてのトランザクションを一時停止および再開することができます。トランザクションを一時停止すると、変更のジャーナリングが一時停止されます。したがって、現在のトランザクション中にトランザクションの一時停止が発生した場合、トランザクションが一時停止されているときに行われた変更は、ROLLBACK によってロールバックできません。ただし、ROLLBACK は、トランザクションの一時停止が有効になる前または後に発生した、現在のトランザクション中に行われたすべての変更をロールバックします。

詳細は、“[トランザクション処理での ObjectScript の使用法](#)”を参照してください。

## ObjectScript : トランザクション・コマンド

ObjectScript と SQL のトランザクション・コマンドは完全に互換性があり、置き換え可能ですが、以下の例外があります。

ObjectScript TSTART と SQL START TRANSACTION はどちらも、トランザクションが進行中でない場合にトランザクションを開始します。ただし、START TRANSACTION では、入れ子になったトランザクションはサポートされません。そのため、入れ子になったトランザクションが必要な場合 (または必要になる可能性がある場合) には、トランザクションを TSTART で始めることをお勧めします。SQL 標準との互換性が必要な場合は、START TRANSACTION を使用してください。

ObjectScript トランザクション処理は、入れ子になったトランザクションを限定的にサポートします。SQL トランザクション処理はトランザクション内のセーブポイントをサポートします。

### 例

以下の埋め込み SQL の例は、ROLLBACK がトランザクション・レベル・カウンタ (\$TLEVEL) を START TRANSACTION の直前のレベルである 0 にリストアする方法を示しています。

#### ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT b)
WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT c)
WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(ROLLBACK)
WRITE !,"Rollback transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

以下の埋め込み SQL の例は、ROLLBACK TO SAVEPOINT name が、トランザクション・レベル (\$TLEVEL) を指定された SAVEPOINT の直前のレベルにリストアする方法を示しています。

#### ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level at a=", $TLEVEL
&sql(SAVEPOINT b)
WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
WRITE !,"Transaction level at b=", $TLEVEL
&sql(ROLLBACK TO SAVEPOINT b)
WRITE !,"Rollback to b, SQLCODE=",SQLCODE
WRITE !,"Rollback transaction level=", $TLEVEL
```

```
&sql(SAVEPOINT c)
WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
WRITE !,"Transaction level at c=", $TLEVEL
&sql(SAVEPOINT d)
WRITE !,"Set Savepoint d, SQLCODE=",SQLCODE
WRITE !,"Transaction level at d=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

## 引数

### pointname

既存のセーブポイントの名前。[識別子](#)として指定します。

## 関連項目

- ・ SQL コマンド: [COMMIT](#)、[SAVEPOINT](#)、[SET TRANSACTION](#)、[START TRANSACTION](#)、[\\$TLEVEL](#)
- ・ [トランザクション処理](#)
- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript : [TROLLBACK](#)
- ・ ObjectScript : [トランザクション処理](#)

# SAVEPOINT (SQL)

トランザクション内のポイントを指定します。

## 構文

SAVEPOINT *pointname*

## 概要

SAVEPOINT 文は、トランザクション内のポイントを指定します。セーブポイントを設定することにより、セーブポイントにトランザクションをロールバックできます。ロールバックは実行されたすべての処理を取り消し、その期間に取得されたロックをすべて解放します。実行期間の長いトランザクション、または内部制御構造を持つトランザクションでは、トランザクションの間に送信されたすべての処理を取り消すのではなく、トランザクションの一部をロールバックできる方が便利なが場合が多くあります。

セーブポイントを設定すると、\$TLEVEL トランザクション・レベル・カウンタがインクリメントされます。セーブポイントまでロール・バックすると、\$TLEVEL トランザクション・レベル・カウンタが、そのセーブポイントの直前の値にデクリメントされます。トランザクション内には最大 255 のセーブポイントを設定できます。このセーブポイントの数を超えると、SQLCODE -400 致命的エラーである SQL 実行中の <TRANSACTION LEVEL> 例外のキャッチが発生します。ターミナル・プロンプトには、現在のトランザクション・レベルがプレフィックスの接頭語の TLn: として表示されます。n は、現在の \$TLEVEL のカウントを表す 1 ～ 255 の整数です。

各セーブポイントは、一意の識別子であるセーブポイント名に関連付けられています。また、セーブポイント名では大文字と小文字が区別されません。セーブポイント名は、区切り文字付き識別子とすることができます。

- pointname を指定していない SAVEPOINT、または有効な識別子ではないか、SQL 予約語である pointname を指定した SAVEPOINT を指定すると、ランタイム SQLCODE -301 エラーが発行されます。
- “SYS” で始まる pointname を指定した SAVEPOINT を指定すると、ランタイム SQLCODE -302 エラーが発行されます。これらのセーブポイント名は予約されます。

セーブポイント名は、大文字小文字が区別されないの、resetpt、ResetPt、および "RESETPT" は同じ pointname です。この重複は、SAVEPOINT ではなく ROLLBACK TO SAVEPOINT の間に検出されます。重複する pointname を使用して SAVEPOINT 文を指定すると、InterSystems IRIS は、pointname が一意である場合と同様にトランザクション・レベル・カウンタをインクリメントさせます。ただし、最新の pointname がセーブポイント名のテーブル内のこれまでの重複する値をすべて上書きします。そのため、ROLLBACK TO SAVEPOINT pointname を指定すると、InterSystems IRIS はその pointname を持つ最後に設定された SAVEPOINT にロールバックし、トランザクション・レベル・カウンタを適切にデクリメントします。ただし、再度同じ名前でも ROLLBACK TO SAVEPOINT pointname を指定すると、SQLCODE -375 エラーが生成され、%msg が " 'name' ROLLBACK " となり、完全トランザクションがロールバックされ、\$TLEVEL カウントが 0 に戻ります。

## セーブポイントの使用法

SAVEPOINT 文は、埋め込み SQL、動的 SQL、ODBC、および JDBC でサポートされます。JDBC は、`connection.setSavepoint(pointname)` はセーブポイントを設定し、`connection.rollback(pointname)` は指定されたセーブポイントにロールバックします。

セーブポイントを設定した場合：

- ROLLBACK TO SAVEPOINT pointname は、指定したセーブポイント以降に実行された処理をロールバックし、そのセーブポイントとすべての中間セーブポイントを削除して、削除したセーブポイント数分だけ \$TLEVEL トランザクション・レベル・カウンタをデクリメントします。pointname が存在しないか既にロール・バックされている場合、このコマンドはトランザクション全体をロール・バックし、\$TLEVEL を 0 にリセットして、すべてのロックを解除します。

- ROLLBACK は、現在のトランザクションの間に実行されたすべての作業をロール・バックし、START TRANSACTION 以降処理された作業をロール・バックします。\$TLEVEL トランザクション・レベル・カウンタを 0 にリセットし、すべてのロックを解除します。一般的な ROLLBACK は、セーブポイントが無視します。
- COMMIT は、現在のトランザクションの間に実行されたすべての作業をコミットします。\$TLEVEL トランザクション・レベル・カウンタを 0 にリセットし、すべてのロックを解除します。COMMIT はセーブポイントが無視することに注意してください。

トランザクション内で 2 番目の START TRANSACTION を発行しても、セーブポイントまたは \$TLEVEL トランザクション・レベルのカウンタには影響を与えません。

トランザクションの動作が正常完了できなかった場合は、SQLCODE -400 エラーが発行されます。

## 引数

### pointname

セーブポイントの名前。識別子として指定します。

## 例

次の埋め込み SQL の例では、2 つのセーブポイントを持つトランザクションを作成しています。

### ObjectScript

```
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(START TRANSACTION)
&sql(DELETE FROM Sample.Person WHERE Name=NULL)
IF SQLCODE=100 { WRITE !,"No null name records to delete" }
ELSEIF SQLCODE'=0 {&sql(ROLLBACK)}
ELSE {WRITE !,%ROWCOUNT," null name records deleted"}
&sql(SAVEPOINT svpt_age1)
&sql(DELETE FROM Sample.Person WHERE Age=NULL)
IF SQLCODE=100 { WRITE !,"No null age records to delete" }
ELSEIF SQLCODE'=0 {&sql(ROLLBACK TO SAVEPOINT svpt_age1)}
ELSE {WRITE !,%ROWCOUNT," null age records deleted"}
&sql(SAVEPOINT svpt_age2)
&sql(DELETE FROM Sample.Person WHERE Age>65)
IF SQLCODE=0 { &sql(COMMIT)}
ELSEIF SQLCODE=100 { &sql(COMMIT)}
ELSE {
    &sql(ROLLBACK TO SAVEPOINT svpt_age2)
    WRITE !,"retirement age deletes failed"
}
&sql(COMMIT)
&sql(COMMIT)
```

## ObjectScript と SQL のトランザクション

TSTART と TCOMMIT を使用した ObjectScript トランザクション処理は、SQL 文 START TRANSACTION、SAVEPOINT、および COMMIT を使用した SQL トランザクション処理とは異なるもので、互換性也没有ありません。ObjectScript および InterSystems SQL のいずれも、入れ子構造のトランザクションのサポートに制限があります。ObjectScript トランザクション処理は、SQL ロック制御変数 (特に SQL ロック・エスカレーション変数) と相互にやり取りしません。アプリケーションは、これら 2 種類のトランザクション処理を混同しないよう注意する必要があります。

トランザクションに SQL 更新文が含まれる場合、SQL の START TRANSACTION 文でトランザクションが開始され、COMMIT 文でコミットされます。トランザクションを開始するものでない限り、TSTART/TCOMMIT を入れ子にして使用するメソッドをトランザクションに組み込むことができます。メソッドとストアド・プロシージャは、通常、設計でトランザクションの主要なコントローラにならない限り、SQL トランザクション制御文を使用しません。

## 関連項目

- SQL コマンド : [COMMIT ROLLBACK SET TRANSACTION START TRANSACTION \\$TLEVEL](#)
- [トランザクション処理](#)



- ・ [SQLCODE エラー・メッセージ](#)
- ・ ObjectScript コマンド: [TCOMMIT](#)

# SELECT (SQL)

データベース内のテーブルから行を検索します。

## 構文

### 基本的な選択

```
SELECT * FROM table
SELECT selectItem FROM table
SELECT selectItem, selectItem2, ... FROM table
SELECT ... FROM table, table2, ...
```

### 述語条件

```
SELECT ... FROM ... WHERE condition
SELECT ... FROM ... [WHERE condition] GROUP BY column
SELECT ... FROM ... [WHERE condition][GROUP BY column]
    HAVING condition
SELECT ... FROM ... [WHERE condition][GROUP BY column]
    [HAVING condition] ORDER BY itemOrder [ASC | DESC]
```

### エイリアス

```
SELECT selectItem AS columnAlias FROM ...
SELECT selectItem AS columnAlias, selectItem2 AS columnAlias2, ... FROM ...
SELECT ... FROM table AS tableAlias ...
SELECT ... FROM table AS tableAlias, table2 AS tableAlias2, ...
```

### 選択条件

```
SELECT DISTINCT ... FROM ...
SELECT DISTINCT BY (distinctItem) ... FROM ...
SELECT DISTINCT BY (distinctItem, distinctItem2, ...) ... FROM ...
SELECT TOP numRows ... FROM ...
SELECT DISTINCT TOP ... FROM ...
SELECT TOP ALL ... FROM ...
SELECT ALL ... FROM ...
```

### 埋め込み SQL のホスト変数

```
SELECT selectItem INTO :var FROM ...
SELECT selectItem, selectItem2, ... INTO :var, :var2, ... FROM ...
SELECT * INTO :var() FROM ...
```

### キーワード・オプション

```
SELECT %keyword ... FROM ...
```

### サブクエリとクエリ・キャッシュ

```
(SELECT ... FROM ...)
```

### サンプルの選択

```
SELECT ... FROM table WHERE %ID %FIND %SQL.SAMPLE( tablename, percent, seed )
```

### 時系列機械学習モデルからの選択

```
SELECT WITH PREDICTIONS( model-name ) ... FROM ...
```

## 説明

SELECT 文は、InterSystems IRIS® データベースのデータを取得するクエリを実行します。最も単純な形式では、1 つのテーブルの 1 つまたは複数の列からデータを取得します。SELECT [selectItem](#) 節は、選択する列を指定します。FROM [table](#) 節は、選択元のテーブルを指定します。オプションの WHERE 節は、どの行の列の値を返すかを指定する [condition](#) 要素を 1 つまたは複数指定します。

より複雑なクエリでは、SELECT 文は、列データ、集約データ、計算された列データを取得したり、結合を使用して複数のテーブルからデータを取得したりできます。ビューを使用してデータを取得することもできます。

SELECT 文は、以下のコンテキストで使用できます。

- ・ 独立した InterSystems SQL クエリ。
- ・ 括弧で囲まれた SELECT 文に値を提供するサブクエリ。
- ・ [UNION](#) のサブセット。[UNION](#) 文で、1 つまたは複数の SELECT 文を単独のクエリに結合できます。
- ・ ビューで使用可能なデータを定義する [CREATE VIEW](#) の一部。
- ・ SELECT 文と組み合わせた INSERT 文の一部。INSERT 文は、SELECT を使用して、別のテーブルからデータを選択して、複数行のデータ値をテーブルに挿入できます。詳細は、“[複数行の挿入](#)”を参照してください。
- ・ [ダイナミック SQL](#) クエリ、[埋め込み SQL](#) クエリ、または[クラス・クエリ](#)として準備された独立したクエリ。ダイナミック SQL を使用すると、クエリ内で指定した列の数、クエリ内で指定した列の名前 (またはエイリアス)、クエリ内で指定した列のデータ型など、SELECT クエリに関するメタデータを返すこともできます。
- ・ 埋め込み SQL と共に使用する [DECLARE CURSOR](#) の一部。

SELECT は、クエリの結果を結果セットで返します。また、[SQLCODE](#) ステータス変数も設定し、クエリの成功または失敗を示します。詳細は、“[SELECT のステータスと戻り値](#)”を参照してください。

SELECT 節は、構文に示す順序で指定する必要があります。SELECT 節を不適切な順序で指定すると、SQLCODE - 25 エラーが生成されます。SELECT 構文の順序は SELECT 節のセマンティック処理順序と同じではありません。詳細は、“[SELECT 節の実行順序](#)”を参照してください。

## 基本的な選択

- ・ SELECT \* FROM [table](#) は、すべての項目を 1 つのテーブルから選択します。通常、これらの項目は、テーブル内の列です。

以下のクエリは、`Sample.Person` テーブルからすべての列を選択します。

### SQL

```
SELECT * FROM Sample.Person
```

この形式の詳細は、“[すべての列の選択 \(アスタリスク構文\)](#)”を参照してください。

- ・ SELECT [selectItem](#) FROM [table](#) は、1 つの項目を 1 つのテーブルから選択します。

以下のクエリは、`Sample.Person` テーブルから `Name` 列を選択します。

### SQL

```
SELECT Name FROM Sample.Person
```

- ・ SELECT [selectItem](#), [selectItem2](#), ... FROM [table](#) は、[selectItem](#) の値のコンマ区切りリストを使用して、1 つのテーブルから複数の項目を選択します。

以下のクエリは、`Sample.Person` テーブルから `Name` 列および `Age` 列を選択します。

## SQL

```
SELECT Name, Age FROM Sample.Person
```

- SELECT ... FROM table, table2, ... は、table の名前のコンマ区切りリストを使用して、複数のテーブルから項目を選択します。フィールドの選択元のテーブルを指定するためにエイリアスを使用することなく、複数のテーブルに存在する列名を選択しようとする、SQLCODE -27 が発生します。

以下のクエリは、Sample.Person テーブルと Sample.Company テーブルの両方から SSN 列および Mission 列を選択します。SSN ごとに、すべての Mission が返されます。

## SQL

```
SELECT SSN, Mission FROM Sample.Person, Sample.Company
```

SELECT 文で複数のテーブルを関連付け、それらのテーブルの共通部分からデータを選択するには、JOIN 式を使用します。

注釈 関数からデータを返す文など、テーブル・データを参照しない文の場合、FROM 節はオプションです。この節の詳細は、“FROM” を参照してください。

## 述語条件

- SELECT ... FROM ... WHERE condition は、condition (論理演算子で結合した一連の述語) が真となるテーブル行を返します。例えば、以下の文は、40 歳以上のマサチューセッツ州の居住者のみを選択します。

## SQL

```
SELECT Name, Age, Home_State FROM Sample.Person WHERE Age > 40 AND Home_State = 'MA'
```

また、condition 引数は、集約関数に提供する値を該当する行からの値に制限します。WHERE 節の述語は、集約関数を直接受け入れません。これらの値は、他の節から WHERE 節に渡す必要があります。詳細は、“WHERE” を参照してください。

例：述語条件を使用したデータのサブセットの選択

- SELECT ... FROM ... [WHERE condition] GROUP BY column は、クエリ結果セットをグループに編成し、指定したテーブル列から取得した個別値ごとに 1 行を返します。

column 引数には、列名のコンマ区切りリスト、または列名に評価されるスカラ式を指定できます。GROUP BY 節は多くの場合、集約関数と組み合わせて使用されます。

以下のクエリは、Home\_State 列で見つかった個別の州ごとに 1 行を返すと共に、COUNT(Home\_State) の選択結果を計算して各州の数を返します。

## SQL

```
SELECT Home_State, COUNT(Home_State)
FROM Sample.Person
GROUP BY Home_State
```

これらの節の詳細は、“GROUP BY (SQL)” を参照してください。

例：述語条件を使用したデータのサブセットの選択

- SELECT ... FROM ... [WHERE condition][GROUP BY column] HAVING condition は、HAVING condition が真であるテーブル行を返します。WHERE 節と異なり、HAVING 節はグループを操作するもので、多くの場合 GROUP BY 節と組み合わせて使用されます。

HAVING condition は、返される行を指定しますが、既定では、集約関数に提供される値をこれらの行の値に制限しません。この既定値を上書きするには、`%AFTERHAVING` キーワードを使用します。

WHERE 節と異なり、HAVING 節には[集約関数](#)を指定できます。

以下のクエリは、Age がデータベース内の全員の平均年齢を上回る行を返します。

## SQL

```
SELECT Name,AVG(Age %AFTERHAVING)
FROM Sample.Person
HAVING (Age > AVG(Age))
```

この節の詳細は、“[HAVING \(SQL\)](#)” を参照してください。

例： [述語条件を使用したデータのサブセットの選択](#)

- SELECT ... FROM ... [WHERE condition][GROUP BY column][HAVING condition] ORDER BY [itemOrder](#) [ASC | DESC] は、返される行を表示する順序を指定します。itemOrder は、SELECT [selectItem](#) 節で指定した選択項目として指定するか、該当する項目のコンマ区切りリストとして指定します。

各項目には、その項目の返り値の順序を指定する ASC (昇順) キーワードまたは DESC (降順) キーワードをオプションで指定できます。既定は昇順です。

以下のクエリは、データベースのすべての行の選択された列を年齢の昇順で返します。

## SQL

```
SELECT Home_State, Name, Age
FROM Sample.Person
ORDER BY Age
```

SELECT コマンドは、ORDER BY 節をクエリの結果に適用します。この節は多くの場合、TOP 節と組み合わせて使用されます。ORDER BY 節をサブクエリまたは CREATE VIEW クエリで使用する場合、TOP 節は必須です。

“[ウィンドウ関数の概要](#)” で説明しているように、ORDER BY 節にはウィンドウ関数を含めることができます。

この節の詳細は、“[ORDER BY \(SQL\)](#)” を参照してください。

例： [述語条件を使用したデータのサブセットの選択](#)

## エイリアス

- SELECT [selectItem](#) AS [columnAlias](#) FROM ... は、列または他の selectItem の値の名前にエイリアスを設定します。columnAlias の値は、結果セットの列見出しとして表示されます。エイリアスを使用すると、返されるデータを理解しやすくなります。エイリアスを指定しない場合、結果セットでは、選択項目で指定されている列名が使用されます。

以下のクエリは、Home\_State の結果を US\_State\_Abbrev という名前の列に返します。

## SQL

```
Select Home_State AS US_State_Abbrev FROM Sample.Person
```

例： [結果セットの列の大文字/小文字の変更](#)

- SELECT selectItem AS columnAlias, selectItem2 AS columnAlias2, ... FROM ... は、複数の選択項目のエイリアスを設定します。

以下のクエリは、Name および Home\_State の結果をそれぞれ PersonName 列および State 列に返します。

## SQL

```
Select Name AS PersonName,Home_State AS State FROM Sample.Person
```

例：マルチテーブル・クエリにおける列名の区別

- SELECT ... FROM `table` AS `tableAlias` ... は、テーブル名のエイリアスを設定します。テーブル・エイリアスは、`selectItem` 値のテーブルの接頭語として使用できます。
- SELECT ... FROM `table` AS `tableAlias`, `table2` AS `tableAlias2`, ... は、複数のテーブルのエイリアスを設定します。テーブル・エイリアスを使用して、選択した列が属するテーブルを示すことができます。以下に例を示します。

#### SQL

```
SELECT P.Name, E.Name FROM Sample.Person AS P, Sample.Employee AS E
```

例：マルチテーブル・クエリにおける列名の区別

## 選択条件

### DISTINCT 節

- SELECT DISTINCT ... FROM ... は、`selectItem` 値の一意の組み合わせごとに 1 行のみを返します。この節を使用して、重複する列値を結果セットから除外できます。1 つまたは複数の選択項目を指定できます。

以下のクエリは、`Home_State` 値と `Age` 値の一意の組み合わせごとに 1 行を返します。

#### SQL

```
SELECT DISTINCT Home_State, Age FROM Sample.Person
```

- SELECT DISTINCT BY (`distinctItem`) ... FROM ... は、`distinctItem` の一意の値ごとに 1 行を返します。`distinctItem` は、`selectItem` の項目である必要があります。`distinctItem` は括弧で囲みます。DISTINCT BY を使用すると、`selectItem` で指定した項目以外の項目に基づいて個別値を返すことができます。クエリ `SELECT DISTINCT BY (item) item FROM Sample.Person` は、`SELECT DISTINCT item FROM Sample.Person` と同じです。

以下のクエリは、一意の `Age` 値ごとに `Name` 値と `Age` 値が含まれる 1 行を返します。

#### SQL

```
SELECT DISTINCT BY (Age) Name, Age FROM Sample.Person
```

- SELECT DISTINCT BY (`distinctItem`, `distinctItem2`, ...) ... FROM ... は、コンマ区切りリストを使用して、`distinctItem` 値の一意の組み合わせごとに 1 行を返します。

以下のクエリは、`Home_State` 値と `Age` 値の一意の組み合わせごとに、`Name` 値と `Age` 値が含まれる 1 行を返します。

#### SQL

```
SELECT DISTINCT BY (Home_State, Age) Name, Age FROM Sample.Person
```

### TOP 節

- SELECT TOP `numRows` ... FROM ... は、返された `table` の "上位" から、`numRows` で指定した行数分を返します。既定の "上位" の行は、予測不能な場合があります。返される上位の行をさらに制御するには、**DISTINCT** 節を含めて一意の値のみを返したり、**ORDER BY** 節を含めて特定の行に基づいて値を並べることができます。SELECT コマンドは、TOP 行を選択する前にこれらの節を適用します。

以下のクエリは、上位 10 個の `Name` 値をアルファベット順で返します。

#### SQL

```
SELECT TOP 10 Name FROM Sample.Person ORDER BY Name ASC
```

詳細は、“[TOP](#)”を参照してください。

- SELECT DISTINCT TOP ... FROM ... は、“上位”の一意の行を指定された数だけ返します。
- SELECT TOP ALL ... FROM ... は、すべての行を選択します。この構文は、サブクエリまたは CREATE VIEW 文で使った場合にのみ意味を持ちます。この構文は、そのような状況で ORDER BY 節の使用をサポートするために使用します。サブクエリ、または CREATE VIEW 文で使用するクエリでは、ORDER BY 節を TOP 節と組み合わせる必要がありますが、この構文により、その要件を満たすことができます。TOP ALL 操作は、返される行数を制限するものではありません。

## ALL 節

- SELECT ALL ... FROM ... は、SELECT 条件を満たすすべての行を返します。これは、InterSystems SQL の既定です。ALL キーワードは何の操作も実行しません。これは、SQL の互換性のためだけに用意されたものです。

## 埋め込み SQL のホスト変数

- SELECT [selectItem](#) INTO :var FROM ... は、1 つのテーブルから 1 つの列を選択して[ホスト変数](#) var に保存します。ホスト変数は、[埋め込み SQL](#) のクエリでのみ指定できます。
- SELECT selectItem, selectItem2, ... INTO :var, :var2, ... FROM ... は、複数の列を選択して、対応するホスト変数に保存します。列数は、ホスト変数の数と一致しなければなりません。ホスト変数は述語節でも使用できます。

このクラス・スニペットは、2 つのフィールドを選択し、後でフェッチできるようにそれらをホスト変数に格納するカーソルを宣言します。カーソルは、クラスの入力引数を個別のホスト変数として格納することによって、フェッチ結果をソートし、フィルタ処理します。

### Class Member

```
ClassMethod AgeThreshold(ageThreshold As %Integer, orderBy As %String = "") As %Status
{
    write "People who are age " _ageThreshold_ " and up:"

    &sql(declare CC cursor for
        SELECT Name, Age
        INTO :name, :age
        FROM Demo.Person
        WHERE (Age >= :ageThreshold)
        ORDER BY :orderBy)

    // ...
}
```

例：埋め込み SQL およびダイナミック SQL を使用した ObjectScript プログラム内からのデータの選択

- SELECT \* INTO :var() FROM ... は 1 つのテーブルからすべての列を選択して添え字付き変数 var に保存します。クラス定義でプライベートとして指定された列は含まれません。ホスト変数にアクセスするには、構文 var(colIndex) を使用します。ここで colIndex は、この列の SqlColumnNumber によって特定される列の順序インデックスです。詳細は、“[列番号を添え字とするホスト変数](#)”を参照してください。

## キーワード・オプション

- SELECT %keyword ... FROM ... は、1 つまたは複数の %keyword オプションを空白で区切って設定します。有効なオプションは、%NOFPLAN、%NOLOCK、%NORUNTIME、%PROFILE、および %PROFILE\_ALL です。%keyword 引数を使用するには、現在のネームスペースに対応する管理特権が必要となります。詳細は、“[GRANT](#)”を参照してください。

## サブクエリとクエリ・キャッシュ

- (SELECT ... FROM ...) は、追加された括弧のペアごとに個別のクエリ・キャッシュを生成します。サブクエリを指定する際にも、クエリを括弧で囲む必要があります。



SELECT 文では、[selectItem](#) リスト、FROM 節、または EXISTS 述語か IN 述語が含まれる WHERE 節にサブクエリを指定できます。また、UPDATE 文または DELETE 文にもサブクエリを指定できます。サブクエリは括弧で囲む必要があります。

独立した SELECT クエリ、UNION サブセットの SELECT クエリ、CREATE VIEW SELECT クエリ、および DECLARE CURSOR SELECT クエリの場合、1 つまたは複数の括弧の組はオプションです。SELECT クエリを括弧で囲むと、サブクエリの構文ルールに従うことになります。特に、ORDER BY 節は TOP 節と組み合わせる必要があります。

括弧は INSERT ... SELECT 文では許可されません。

## サンプルの選択

- SELECT ... FROM table WHERE %ID %FIND %SQL.SAMPLE( tablename, percent, seed ) は、テーブルからランダムなサンプルを選択します。percent 引数は、各データ・ブロックのサンプリングする割合を指定します。データ・ブロックに格納されている行数は可変であるため、テーブルの 10% をサンプリングしても、テーブル内のすべての行の 10% が返されるとは限りません。table と tablename は、サンプリング元のテーブルの修飾名である必要がありますが、table は識別子、tablename は文字列である必要があります。

また、table はビューを参照できず、[ビットマップ・エクステント・インデックス](#)が必要で、ブロックサンプリングが有効である必要があります。RowID フィールドは、既定の動作と同様に、宣言されるか、暗黙的に正の整数である必要があります。

seed 引数はオプションです。これが指定されていない場合は、既定で空の文字列となります。

## 時系列機械学習モデルからの選択

- SELECT WITH PREDICTIONS( model-name ) ... FROM ... は、時系列モデルから指定の列を選択し、[モデルの作成](#)時に指定した予測時間枠に応じて、予測される行を結果の先頭または末尾に付加します。WHERE 節を使用して、時系列の作成元の列をフィルタ処理することにより、結果を予測された行に制限できます。

IntegratedML の詳細は、[“IntegratedML の概要”](#) を参照してください。

## 引数

### selectItem

selectItem はすべての SELECT 文に必須の引数です。この引数で、テーブルから選択する 1 つの項目、または複数の項目のコンマ区切りリストを指定します。各 selectItem は以下のいずれかとして指定できます。

- FROM 節で指定したテーブルの列名。[“テーブル列の選択”](#) を参照してください。
- テーブル列またはテーブル全体を参照するサブクエリ。[“サブクエリの選択”](#) を参照してください。
- FROM 節で指定していないテーブルの列名。暗黙結合 (矢印構文とも呼ばれる) を使用して指定します。[“暗黙結合による選択”](#) を参照してください。
- テーブル内のすべての列。アスタリスク(\*) 構文を使用して指定します。[“すべての列の選択”](#) を参照してください。
- 埋め込みシリアル・オブジェクト内のプロパティ。アンダースコア ( ) 構文を使用して指定します。[“埋め込みシリアル・オブジェクトの選択”](#) を参照してください。

テーブル列から選択したデータを変更する関数、または選択内容から新しいデータを計算する関数として、selectItem を指定することもできます。以下の項目を指定できます。

- 選択した列データを集約し、1 つの値を返す SQL 関数。[“集約関数による選択”](#) を参照してください。
- 各行の集約、ランキングなどの関数を、その行に固有の“ウィンドウ・フレーム”に基づいて計算するウィンドウ関数。[“ウィンドウ関数による選択”](#) を参照してください。

- ・ 選択したテーブル・データを操作する SQL 関数、ObjectScript クラス・メソッド呼び出し、または ObjectScript 関数呼び出し。“[関数およびメソッド呼び出しによる選択](#)”を参照してください。

最後に、selectItem を使用して、返されたすべての行に対して同じ値が含まれる列を生成できます。選択した列の各行に同じテキストなどのデータを挿入することもできます。これらの選択はテーブル・データを操作するものではありません。“[非テーブル・データの選択](#)”を参照してください。

## テーブル列の選択

一般に、selectItem は、FROM 節で指定された [table](#) 内の列を参照します。複数の列を指定するには、コンマ区切りのリストを使用します。以下に例を示します。

### SQL

```
SELECT Name, Age FROM Sample.Person
```

列名では、大文字と小文字は区別されません。ただし、結果セットの列に関連付けられているラベルでは、selectItem に指定された大文字/小文字は使用されません。代わりに、テーブル定義で指定された、対応する [SqlFieldName](#) ObjectScript プロパティの大文字/小文字が使用されます。大文字/小文字の解決の詳細は、“[結果セットの列の大文字/小文字の変更](#)”の例を参照してください。

指定したテーブルに定義されている列名をすべてリストするには、“[列の名前と番号](#)”を参照してください。

[RowID](#) (レコード ID) を表示するには、[%ID 疑似フィールド変数](#)エイリアスを使用できます。これは割り当てられている名前に関係なく、RowID を表示します。既定では、RowID の名前は ID ですが、テーブルに既に ID という名前の列が含まれている場合、名前が変更されることがあります。既定では、RowID は非表示の列です。

ストリーム列に対して SELECT クエリを実行すると、開いているストリーム・オブジェクトの OREF (オブジェクト参照) が返されます。以下に例を示します。

### SQL

```
SELECT Name, Picture FROM Sample.Employee WHERE Picture IS NOT NULL
```

FROM 節に複数のテーブルまたはビューを指定する場合、テーブル名や列名をピリオドで区切って、selectItem の一部としてテーブル名を含める必要があります。以下に例を示します。

### SQL

```
SELECT Sample.Person.Name, Sample.Employee.Company
FROM Sample.Person, Sample.Employee
```

テーブル・エイリアスを指定した場合は、代わりにそのエイリアスを selectItem で指定します。以下に例を示します。

### SQL

```
SELECT p.Name, e.Company
FROM Sample.Person AS p, Sample.Employee AS e
```

テーブル名に既にエイリアスが割り当てられている場合、selectItem の一部として完全なテーブル名を指定すると、SQLCODE -23 エラーが発生します。

## サブクエリによる選択

selectItem をサブクエリとして指定すると、指定したテーブルから 1 つの列が返されます。この列には、1 つのテーブル列の値、または 1 つの列として返された複数のテーブル列の値を含めることができます。連結 (SELECT Home\_City||Home\_State) を使用するか、コンテナ列 (SELECT Home) を指定することによって、1 つの列で複数の列を返すことができます。サブクエリでは、[暗黙結合 \(矢印構文\)](#)を使用できます。サブクエリでは、サブクエリ内で引用されたテーブルに 1 つのデータ列しかない場合でも、アスタリスク構文は使用できません。

サブクエリを使用して、GROUP BY 節の影響を受けない集約関数を指定できます。以下の例では、GROUP BY 節によって、年齢を 10 年ごとにグループ化します (例えば、25 歳から 34 歳)。AVG(Age) selectItem は、GROUP BY 節で定義される各グループの平均年齢となります。全グループ内のすべてのレコードの平均年齢を取得するために、サブクエリを使用します。

## SQL

```
SELECT Age AS Decade,
       COUNT(Age) AS PeopleInDecade,
       AVG(Age) AS AvgAgeForDecade,
       (SELECT AVG(Age) FROM Sample.Person) AS AvgAgeAllDecades
FROM Sample.Person
GROUP BY ROUND(Age,-1)
ORDER BY Age
```

### 暗黙結合による選択 (矢印構文)

FROM 節のテーブル以外のテーブルから列にアクセスするには、矢印構文 (->) を使用して、selectItem を暗黙結合として指定します。以下の例では、Sample.Employee テーブルに Company 列が含まれていて、この列に、Sample.Company テーブル内の対応する会社名の RowID が含まれています。矢印構文では、そのテーブルから会社名を取得します。

## SQL

```
SELECT Name,Company->Name AS CompanyName
FROM Sample.Employee
```

この場合、参照されるテーブルに対する SELECT 特権 (参照される列と、参照されるテーブルの RowID 列の両方に対する、テーブルレベルの SELECT 特権または列レベルの SELECT 特権) が必要です。矢印構文の詳細は、“[暗黙結合 \(矢印構文\)](#)” を参照してください。

### すべての列の選択 (アスタリスク構文)

テーブル内のすべての列を選択するには、アスタリスク (\*) 構文を使用します。以下に例を示します。

## SQL

```
SELECT TOP 5 * FROM Sample.Person
```

項目は、[列番号順](#)に返されます。アスタリスク構文による選択には、シリアル・オブジェクト内に入れ子になったシリアル・オブジェクトのプロパティなど、[埋め込みシリアル・オブジェクト](#)のプロパティが含まれます。シリアル・オブジェクトを参照する列は選択されません。例えば、埋め込みシリアル・オブジェクトの Home\_City プロパティは選択されますが、(City プロパティを含む) Sample.Address 埋め込みシリアル・クラスにアクセスするために使用される Home 参照列は選択されません。

アスタリスク構文では、非表示の列は選択されません。既定では、[RowID](#) は非表示です (SELECT \* では表示されません)。ただし、%PUBLICROWID を指定してテーブルが定義されている場合は、SELECT \* は RowID 列および非表示ではないすべての列を返します。既定では、この列の名前は ID ですが、ID という名前のユーザ定義列が既に存在する場合、名前が変更されることがあります。

アスタリスクと複数の [table](#) が指定されているクエリは、結合されたすべてのテーブルのすべての列を選択します。例えば、以下のクエリは、Sample.Company と Sample.Employee の両方の上位 5 行の列をすべて選択します。

## SQL

```
SELECT TOP 5 * FROM Sample.Company,Sample.Employee
```

アスタリスク構文は、修飾、未修飾のどちらでもかまいません。selectItem が、テーブル名 (またはテーブル名エイリアス) およびピリオド (.) を接頭語としてアスタリスクの前に付けて使用することによって修飾される場合、selectItem は、指定されたテーブル内のすべての列を選択します。修飾されたアスタリスク構文を、他のテーブルを対象とした他の選択項目

と組み合わせることができます。以下の例では、selectItem は、テーブルからすべての列を選択する未修飾のアスタリスク構文で構成されます。重複する列名（この例では Name）および列でない selectItem 要素（この例では {fn NOW}）も指定できます。

## SQL

```
SELECT TOP 5 {fn NOW} AS QueryDate,
            Name AS Client,
            *
FROM Sample.Person
```

以下の例では、selectItem は、1 つのテーブルからすべての列を選択する修飾されたアスタリスク構文と、別のテーブルの列名のリストで構成されます。

## SQL

```
SELECT TOP 5 E.Name AS EmpName,
            C.*,
            E.Home_State AS EmpState
FROM Sample.Employee AS E, Sample.Company AS C
```

**注釈** SELECT \* は、アプリケーションの開発およびデバッグ中に非常に便利な InterSystems SQL を完全にサポートしている部分です。ただし、プロダクション・アプリケーションでは、選択された列を明示的にリストするプログラミング方法をお勧めします。列を明示的にリストすることで、アプリケーションを明確かつ簡単に理解でき、維持管理が容易になるほか、列を名前で簡単に検索できるようになります。

## 埋め込みシリアル・オブジェクトの選択（アンダースコア構文）

**埋め込みシリアル・オブジェクト・プロパティ**（埋め込みシリアル・クラス・データ）を選択するには、アンダースコア構文を使用して selectItem を指定します。アンダースコア構文は、オブジェクト・プロパティの名前、アンダースコア、および埋め込みオブジェクト内のプロパティで構成されます。例えば、Home\_City や Home\_State です（例えばインデックス・テーブルなど他のコンテキストでは、これらはドット構文 Home.City を使用して記述されます）。

以下の例を考えてみます。

## SQL

```
SELECT Home_City, Home_Phone_AreaCode FROM Sample.Person
```

Home\_City という列名の場合、Sample.Person テーブルに参照列 Home が含まれます。この列は、プロパティ City を定義する埋め込みシリアル・オブジェクトを参照します。Home\_Phone\_AreaCode、という列名の場合、プロパティ AreaCode を定義する入れ子になった埋め込みシリアル・オブジェクトを参照する埋め込みシリアル・オブジェクト・プロパティ Phone を参照する参照列 Home がテーブルに含まれています。Home や Home\_Phone のような参照列を選択した場合、シリアル・オブジェクト内のプロパティすべての値を **%List データ型**形式で受け取ります。

アンダースコア構文を使用するのではなく、SELECT を使用して、参照列（Home など）を直接クエリできます。返されるデータはリスト形式のため、\$LISTTOSTRING 関数または \$LISTGET 関数を使用してデータを表示できます。以下に例を示します。

## SQL

```
SELECT $LISTTOSTRING(Home, '^') AS HomeAddress FROM Sample.Person
```

## 集約関数による選択

selectItem には、1 つまたは複数の SQL **集約関数**を含めることができます。集約関数は、常に単独の値を返します。以下の表は、指定可能な集約関数のタイプを示しています。

集約関数のタイプ	例
単独の列名。クエリによって選択された行の NULL でないすべての値に対して、集約を計算します。	<code>SELECT AVG(Age) FROM Sample.Person</code>
集約を計算するスカラー式。	<code>SELECT SUM(Age) / COUNT(*) FROM Sample.Person</code>
アスタリスク構文(*)。テーブルの行数を計算するために、COUNT 関数と共に使用されます。	<code>SELECT COUNT(*) FROM Sample.Person</code>
SELECT ... DISTINCT 関数。余剰値を消去することで、集約を計算します。	<code>SELECT COUNT(DISTINCT Home_State) FROM Sample.Person</code>
1 つの SELECT 文における列名と集約関数の組み合わせ (InterSystems SQL では使用できますが、ANSI SQL では使用できません)。	<code>SELECT Name, COUNT(DISTINCT Home_State) FROM Sample.Person</code>
%FOREACH を使用する集約関数。単独または複数の列の個別値ごとに集約を計算します。	<code>SELECT DISTINCT Home_State, AVG(Age %FOREACH(Home_State)) FROM Sample.Person</code>
%AFTERHAVING を使用する集約関数。これによって、HAVING 節で指定されるサブ母集団に対して集約が計算されます。	<code>SELECT Name, AVG(Age %AFTERHAVING) FROM Sample.Person HAVING (Age &gt; AVG(Age))</code>
この例では、クエリは、Age がデータベース内の全員の平均年齢を上回っているレコードを返します。	

## ウィンドウ関数による選択

selectItem を[ウィンドウ関数](#)として指定すると、行ごとに集約、ランキングなどの関数を、その行に固有の "ウィンドウ・フレーム" に基づいて計算できます。以下の構文がサポートされます。

```
windowFunction() OVER (
    PARTITION BY partColumn
    ORDER BY orderColumn)
```

- ・ windowFunction : 以下のウィンドウ関数がサポートされます。AVG()、ROW\_NUMBER()、RANK()、PERCENT\_RANK()、FIRST\_VALUE(column)、LAST\_VALUE(column)、NTH\_VALUE(column, n)、LAG(column, offset)、LEAD(column, offset)、MAX(column)、MIN(column)、および SUM(column)。
- ・ OVER : OVER キーワードとそれに続く括弧は必須です。この括弧内の節はオプションです。
- ・ PARTITION BY partColumn : 指定した partColumn により行を分割するオプションの節。partColumn 引数には、単一の列、またはコンマ区切りの列のリストを指定できます。指定する場合、ORDER BY の前に PARTITION BY を指定する必要があります。
- ・ ORDER BY orderColumn : 指定した orderColumn で行を並べ替えるオプションの節。orderColumn には、単一の列、またはコンマ区切りの列のリストを指定できます。

ウィンドウ関数で指定した列は、[テーブル・エイリアス接頭語](#)を取ることができます。

ウィンドウ関数には列エイリアスを指定できます。既定では、列に window\_n というラベルが付けられます。

詳細は、["ウィンドウ関数の概要"](#)を参照してください。



## 関数およびメソッド呼び出しによる選択

selectItem では、関数やメソッドによる演算を使用することで、選択した列の値に追加の処理を適用できます。以下の演算タイプを指定できます。

- ・ 算術演算。例えば、以下の選択では、Age 列から平均年齢を減算して新しい列を生成します。

### SQL

```
SELECT Name, Age, Age-AVG(Age) FROM Sample.Person
```

selectItem 算術演算に除算が含まれており、除数が 0 または NULL になる列値がある場合、ゼロでの除算を避けるためにテストの順序に頼ることはできません。その代わりに、CASE 文を使用してリスクを抑制します。

- ・ SQL 関数。例えば、以下のクエリでは、Name 列の各値の長さを示す列が生成されます。

### SQL

```
SELECT Name, $LENGTH(Name) FROM Sample.Person
```

以下のクエリは、Name 列の大文字/小文字を大文字に変換して新しい列で返します。

### SQL

```
SELECT Name, UCASE(Name) FROM Sample.Person
```

- ・ XMLELEMENT、XMLFOREST、または XMLCONCAT 関数。これらは、指定された列名から取得したデータ値を XML タグまたは HTML タグで囲みます。詳細は、“[XMLELEMENT](#)” を参照してください。
- ・ **照合関数**。selectItem 列のソートや表示を指定できます。照合関数は括弧なし (SELECT %SQLUPPER Name) でも括弧付き (SELECT %SQLUPPER(Name)) でも指定できます。照合関数で切り捨てを指定する場合には括弧が必要です (SELECT %SQLUPPER(Name, 10))。
- ・ **プロシージャ**として格納されているユーザ定義のクラス・メソッド。クラス・メソッドは、未修飾メソッド名 (RandLetter()) などまたは修飾メソッド名 (Sample.RandLetter()) などのいずれかにすることができます。以下のクラスでは、Cube() クラス・メソッドが入力整数のキューブを返します。

### Class Definition

```
Class Sample.Person Extends %Persistent [DdlAllowed]
{
  /// Find the Cube of a number
  ClassMethod Cube(val As %Integer) As %Integer [SqlProc]
  {
    RETURN val * val * val
  }
}
```

以下のクエリは、Age 列に対して Cube クラス・メソッドを呼び出し、3 乗した年齢を返します。

### SQL

```
SELECT Age, Person_Cube(Age) FROM Sample.Person
```

InterSystems IRIS は、メソッドの戻り値を 形式から /ODBC 形式に変換します。既定では、メソッドへの入力はいずれの形式から 形式に変換されません。ただし、\$SYSTEM.SQL.Util.SetOption("SQLFunctionArgConversion") メソッドを使用して、システム全体で表示形式から論理形式への入力の変換を構成できます。このオプションの現在の構成を確認するには、\$SYSTEM.SQL.Util.GetOption("SQLFunctionArgConversion") を使用します。

指定されたメソッドが現在のネームスペース内に存在していない場合、SQLCODE -359 エラーが生成されます。指定したメソッドがあいまいな（複数のメソッドを参照する可能性がある）場合、SQLCODE -358 エラーが生成されます。クラス・メソッドの作成の詳細は、“[CREATE METHOD](#)”を参照してください。

- データベースの列を操作するユーザ指定の ObjectScript 関数呼び出し（外部関数）。以下に例を示します。

### SQL

```
SELECT $$REFORMAT(Name) FROM MyTable
```

SQL 文でそのような関数を呼び出すには、[SQL ] オプションをシステム全体で構成する必要があります。詳細は、“[関数：内部および外部](#)”を参照してください。既定では、外部関数は無効になっており、ユーザ指定の関数を呼び出そうとすると、SQLCODE -372 エラーが生成されます。

ユーザ指定の関数を使用して % ルーチンを呼び出そうとすると、SQLCODE -373 エラーが生成されます。

### 非テーブル・データの選択

selectItem 引数は、FROM 節内にあるテーブルを参照することなく、すべてのレコードに対して同じ値を返すことができます。テーブル・データを参照する selectItem 要素がない場合、FROM 節はオプションです。FROM 節を含める場合には、指定したテーブルが存在している必要があります。詳細は、“[FROM](#)”を参照してください。

この形式の選択の一般的な使用法は以下のとおりです。

- 算術演算。

### SQL

```
SELECT Name, Age, 9 - 6 FROM Sample.Person
```

- 文字列リテラル、または文字列リテラルを操作する関数。

### SQL

```
SELECT UCASE('fred') FROM Sample.Person
```

- 結果を読みやすくするために追加する文字列リテラル。

### SQL

```
SELECT TOP 10 Name, 'was born on', %EXTERNAL(DOB)
FROM Sample.Person
```

数値リテラルの指定方法によってデータ型が決まります。例えば、文字列 '123' のデータ型は VARCHAR になり、数値 123 のデータ型は INTEGER または NUMERIC になります。

- [%TABLENAME](#) または [%CLASSNAME](#) 疑似フィールド変数キーワード。%TABLENAME は、現在のテーブル名を返します。%CLASSNAME は、現在のテーブルに対応するクラスの名前を返します。クエリが複数のテーブルを参照する場合には、キーワードの接頭語としてテーブルのエイリアスを指定できます。例えば、SUB\_ACCESSIBLE\_FILE のように指定します。
- 以下のいずれかの ObjectScript 特殊変数（またはそれらの省略形）：[\\$HOROLOGY](#)、[\\$JOB](#)、[\\$NAMESPACE](#)、[\\$TLEVEL](#)、[\\$USERNAME](#)、[\\$ZHOROLOGY](#)、[\\$ZJOB](#)、[\\$ZNSPACE](#)、[\\$ZPI](#)、[\\$ZTIMESTAMP](#)、[\\$ZTIMEZONE](#)、[\\$ZVERSION](#)。

### table

データの取得元の 1 つ以上の[テーブル](#)、[ビュー](#)、テーブル値関数、または[サブクエリ](#)。これらの table タイプの任意の組み合わせを、コンマ区切りのリストとして、または [JOIN](#) 構文を使用して指定できます。

- 単一の table 名を指定する場合、指定されたデータはそのテーブルまたはビューから取得されます。



- ・ 複数の table 名を指定する場合、InterSystems SQL はテーブルに対して結合操作を実行し、そのデータを指定されたデータの取得元の結果テーブルにマージします。

FROM 節には有効なテーブル参照が必要です。これは、SELECT でそのテーブルが参照されていない場合でも当てはまります。

- ・ 現在のネームスペースにテーブルまたはビューが存在するかどうかを確認するには、`$SYSTEM.SQL.Schema.TableExists("schema.tname")` メソッドまたは `$SYSTEM.SQL.Schema.ViewExists("schema.vname")` メソッドを使用します。
- ・ テーブルまたはビューに対する [SELECT 特権](#)を持っているかどうかを確認するには、`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを使用します。

table は、修飾 (schema.tablename) でも未修飾 (tablename) でもかまいません。未修飾の table には、[既定のスキーマ名](#)または[スキーマ検索パス](#)のスキーマ名が指定されます。

オプションで、エイリアス [tableAlias](#) をそれぞれの table に割り当てることができます。

オプションで、クエリ実行を最適化するために、1 つ以上の optimize-option キーワードを指定できます。使用可能なオプションは以下のとおりで

す。[%ALLINDEX](#)、[%FIRSTTABLE](#)、[%FULL](#)、[%INORDER](#)、[%IGNOREINDEX](#)、[%NOFLATTEN](#)、[%NOMERGE](#)、[%NOREDUCE](#)、[%NOSMSO](#)、[%NOOPT](#)、[%NOUNIONOPT](#)、[%PARALLEL](#)、および [%STARTTABLE](#)。これらのオプションの詳細は、["FROM \(SQL\)"](#) を参照してください。

## condition

取得するデータの行を指定するために [WHERE](#) 節および [HAVING](#) 節で使用する論理テスト (述語)。SELECT 文では、WHERE condition および HAVING condition は、condition が真に評価される行を返します。

論理述語条件を組み合わせるには、AND および OR の論理演算子を使用します。条件を反転させるには、NOT 単項論理演算子を使用します。

以下の表は、述語条件の例を示しています。

述語	説明	例
等値比較	=、<、>などの比較演算子を使用して行を返します。	SELECT Name, Age FROM Sample.Person WHERE Age < 21
BETWEEN	特定の値の範囲にある行を返します。	SELECT Name, Age FROM Sample.Person WHERE Age BETWEEN 18 AND 21
IN および %INLIST	リストの項目に一致する行を返します。	SELECT Name, Home_State FROM Sample.Person WHERE Home_State IN ( 'ME', 'NH', 'VT' )
部分文字列比較	部分文字列に一致する行を返します。	SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'S'
NULL	未定義の値の検出に基づいて行を返します。	SELECT Name, Age FROM Sample.Person WHERE Age IS NOT NULL
EXISTS	テーブル内の 1 つ以上の行の存在に基づいて行を返します。多くの場合、サブクエリと共に使用されます。	SELECT Name FROM Sample.Person WHERE EXISTS (SELECT * FROM Employee WHERE Employee.Number = Person.Number)
FOR SOME	特定の列値の条件テストに基づいて行を返します。多くの場合、あるテーブル内の特定の値が別のテーブルにもあるかどうかをテストする場合に使用されます。	SELECT Name, COUNT(Name) FROM Sample.Person WHERE FOR SOME (Sample.Employee) (Sample.Employee.Name = Sample.Person.Name)
FOR SOME %ELEMENT	特定のリスト要素の値に一致する行を返します。	SELECT Name, FavoriteColors FROM Sample.Person WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
LIKE、%MATCHES、および %PATTERN	特定のパターンに当てはまる行を照合します。	SELECT Name FROM Sample.Person WHERE Name LIKE '%Mac%'

これらの論理述語の詳細は、“[WHERE \(SQL\)](#)”を参照してください。

condition に集約関数を含めることはできません。集約関数によって返される値を使用して選択条件を指定するには、[HAVING 節](#)を使用します。

WHERE 節では、condition は、= (内部結合) シンボル結合演算子を使用して、2 つのテーブルの間の明示的な結合を指定できます。詳細は、“[JOIN](#)”を参照してください。

WHERE 節は、矢印構文 (->) 演算子を使用して、ベース・テーブルと別のテーブルの列との間の明示的な結合を指定できます。詳細は、“[暗黙結合](#)”を参照してください。

## column

取得したデータの編成方法を指定する列のコンマ区切りリスト。有効な column 値は、以下のとおりです。

- ・ 列名 (GROUP BY City)
- ・ %ID (すべての行を返します)
- ・ 列名を指定するスカラ関数 (GROUP BY ROUND(Age, -1))
- ・ 列名を指定する照合関数 (GROUP BY %EXACT(City))

詳細は、“GROUP BY” を参照してください。

## itemOrder

[selectItem](#)、または行が表示される順序を指定する項目のコンマ区切りリスト。各項目に、オプションで ASC (昇順) または DESC (降順) のキーワードを指定できます。既定は昇順です。ORDER BY 節は、クエリの結果を操作します。ORDER BY 節をサブクエリで使用する場合 (UNION 文で使用する場合など) は、TOP 節と組み合わせる必要があります。ORDER BY 節が指定されていない場合、返されるレコードの順序は予測できません。ORDER BY 節にウィンドウ関数を含めることができます。詳細は、“ORDER BY” を参照してください。

## columnAlias

SELECT クエリでは、[selectItem](#) 内の各列にエイリアスを指定できます。列のエイリアスは、結果セット内の列見出しとして表示されます。列のエイリアスを指定しない場合、選択項目の名前が結果セットの列名として使用されます。AS キーワードは、selectItem を columnAlias から分離します。このキーワードはオプションですが、読みやすくするために推奨されています。したがって、以下の構文は同等で有効です。

```
SELECT Name AS PersonName, DOB AS BirthDate FROM Sample.Person
SELECT Name PersonName, DOB BirthDate FROM Sample.Person
```

InterSystems SQL では、列エイリアスは指定された大文字/小文字で表示されますが、エイリアスを ORDER BY 節内で参照する場合、大文字/小文字は区別されません。columnAlias 名は、[区切り識別子](#)を含め、有効な[識別子](#)である必要があります。区切り識別子を使用すると、列エイリアスに空白やその他の句読点文字を含めたり、列エイリアスを SQL 予約名にしたりできます (例えば、SELECT Name AS "Customer Name" や SELECT Home\_State AS "From")。

SQL は、列エイリアスのための一意性チェックは実行しません。列と列エイリアスの名前を同じにすることや、2 つの列エイリアスを同じにすることができます (ただし、推奨されていません)。このように、列エイリアスが一意でないと、ORDER BY 節によって参照されたときに SQLCODE -24 “ソートカラムがあいまいです” エラーが発生する場合があります。列エイリアスは、すべての SQL 識別子と同様に、大文字と小文字を区別しません。

他の SELECT 節での列エイリアスの使用は、[クエリのセマンティック処理順序](#)によって制御されます。列を参照するには、ORDER BY 節でその列エイリアスを使用します。

以下の場所で列エイリアスを参照することはできません。

- ・ 選択リスト内の別の selectItem
- ・ DISTINCT BY 節
- ・ WHERE 節
- ・ GROUP BY 節
- ・ HAVING 節
- ・ JOIN 操作の ON 節または USING 節

ただし、“[データベースの問い合わせ](#)” で説明されているように、サブクエリを使用すると、こういった他の SELECT 節で列エイリアスを使用できるようになります。

列エイリアスの設定以外に、集約関数、式、または他の計算列にもエイリアスを設定できます。計算列には、自動的に列名が割り当てられます。エイリアスを指定しない場合、InterSystems SQL では、Expression\_1 や Aggregate\_3 などの一意の列名が指定されます。整数の接尾語は、SELECT 文で指定されている selectItem の位置 (つまり、selectItem 列番号) を参照します。これらの値は、そのタイプの列の数ではありません。

以下に、自動的に割り当てられる列名を示します (n は整数です)。これらの名前は、包括性が低いものから高い順にリストされています。例えば、プラスまたはマイナス記号を数値に加えると、数値は HostVar から Expression に昇格します。HostVar と Literal を連結すると、値は Expression に昇格します。サブクエリで Literal、HostVar、Aggregate、または Expression を指定すると、値は SubQuery に昇格します。

- Literal\_n : %TABLENAME や NULL 指定子のような疑似フィールド変数。%ID は Literal\_n ではないことに注意してください。これには実際の RowID 列の列名が付けられます。
- HostVar\_n : ホスト変数。これは、リテラル ('text', 123, 空の文字列 ('')) など、入力変数 (:myvar)、またはリテラルで置き換えられる ? 入力パラメータになります。数字への記号の追加、文字列の連結、算術演算など、リテラルに対する式評価によって Expression\_n になります。? パラメータに指定されたリテラル値は、式評価なしで、変更されないまま返されます。例えば、5+7 を指定すると、文字列 '5+7' が HostVar\_n として返されます。
- Aggregate\_n : 集約関数。例えば、AVG(Age) や COUNT(\*) です。1 番外側の操作が集約関数の場合は、その集約に式が含まれている場合であっても、列の名前は Aggregate\_n になります。例えば、COUNT(Name)+COUNT(Spouse) は Expression\_n ですが、MAX(COUNT(Name)+COUNT(Spouse)) は Aggregate\_n、-AVG(Age) は Expression\_n ですが、AVG(-Age) は Aggregate\_n です。以下の例では、AVG 関数によって作成された集約列に、列エイリアス AvgAge が付けられます。既定の名前は Aggregate\_3 です (SELECT リストで位置 3 にある集約列)。

## SQL

```
SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
```

- Expression\_n : リテラルまたは列に対する selectItem リストまたは Aggregate\_n、HostVar\_n、Literal\_n、または Subquery\_n selectItem で何らかの操作を行うと、列名が Expression\_n に変更されます。これには、数値に対する単項演算 (-Age)、算術演算 (Age+5)、連結 ('USA:' || Home\_State)、データ型 CAST 操作、SQL 照合関数 (%SQLUPPER(Name) %SQLUPPER Name)、SQL スカラ関数 (\$LENGTH(Name))、ユーザ定義のクラス・メソッド、CASE 式、および特殊変数 (CURRENT\_DATE や \$ZPI など) が含まれます。
- Window\_n : ウィンドウ関数の結果。OVER キーワードの閉じ括弧の後に列エイリアスを指定します。
- Subquery\_n : 1 つの selectItem を指定するサブクエリの結果。selectItem は、列、集約関数、式、またはリテラルにできます。サブクエリ内ではなく、サブクエリの後に列エイリアスを指定します。以下に例を示します。

## SQL

```
SELECT Name AS PersonName,
       (SELECT Name FROM Sample.Employee) AS EmpName,
       Age AS YearsOld
FROM Sample.Person
```

## tableAlias

SELECT 文では、テーブル名またはビュー名 (table) に、有効な識別子としてオプションのエイリアスを指定できます。これには区切り識別子も含まれます。AS キーワードは、table を tableAlias から分離します。このキーワードはオプションですが、読みやすくするために推奨されています。したがって、以下の構文は同等で有効です。

```
SELECT P.Name FROM Sample.Person AS P
SELECT P.Name FROM Sample.Person P
```

tableAlias は、クエリ内のテーブル・エイリアス間で一意である必要があります。すべての識別子などの tableAlias では、大文字/小文字は区別されません。大文字/小文字だけが異なる 2 つの tableAlias 名を指定すると、SQLCODE -20 “名前の重複” エラーが発生します。

テーブル・エイリアスは、列が属するテーブルを示すために、列名の接頭語 (ピリオド付き) として使用されます。以下に例を示します。

## SQL

```
SELECT P.Name, E.Name
FROM Sample.Person AS P, Sample.Employee AS E
```

クエリで同じ列名が存在する複数のテーブルを指定する場合は、テーブル参照の接頭語を使用する必要があります。テーブル参照の接頭語としては、tableAlias (1 つ前の例を参照) を使用することや、以下の同義の例で示すように完全修飾テーブル名を使用することができます。

## SQL

```
SELECT Sample.Person.Name, Sample.Employee.Name
FROM Sample.Person, Sample.Employee
```

テーブル名に tableAlias を割り当ててから、selectItem の一部として完全なテーブル名を指定すると、SQLCODE -23 エラーが発生します。テーブル・エイリアスは、クエリのシナリオによって必須の場合とオプションの場合があります。

シナリオ	テーブル・エイリアス
クエリが 1 つのテーブルのみを参照する。	オプション
クエリが複数のテーブルを参照し、参照される列名が各テーブルに固有である。	オプション (ただし、指定することを推奨)
クエリが複数のテーブルを参照し、参照される列名はテーブルが異なっても同じである。	必須 t-alias (または完全修飾テーブル名) 接頭語の指定に失敗すると、SQLCODE -27 “該当テーブル内でフィールド '%1' があいまいです” エラーが発生します。

サブクエリを指定する際に、オプションで tableAlias を使用することもできます。以下に例を示します。

## SQL

```
SELECT Name, (SELECT Name FROM Sample.Vendor)
FROM Sample.Person
```

tableAlias は、クエリ実行のためだけに列を一意に識別します。クエリ結果セットの表示のために列を一意に識別するには、列エイリアス ([columnAlias](#)) を使用する必要もあります。以下のクエリでは、テーブル・エイリアス (Per および Emp) と列エイリアス (PName および Ename) を組み合わせて使用しています。

## SQL

```
SELECT Per.Name AS PName, Emp.Name AS EName
FROM Sample.Person AS Per, Sample.Employee AS Emp
WHERE Per.Name %STARTSWITH 'G'
```

名前が競合することなく、列、列エイリアス、およびテーブル・エイリアスのいずれかまたはすべてに同じ名前を使用できます。

tableAlias 接頭語は、参照先のテーブルを区別するために使用します。以下に例を示します。

## SQL

```
SELECT P.%ID As PersonID,
       AVG(P.Age) AS AvgAge,
       Z.%TABLENAME||'=' AS Tablename,
       Z.*
FROM Sample.Person AS P, Sample.USZipCode AS Z
WHERE P.Home_City = Z.City
GROUP BY P.Home_City
ORDER BY Z.City
```

## distinctItem

結果セットで除外したい重複行が含まれる [selectItem](#) 列のコンマ区切りリスト。distinctItem 引数は、任意の有効な selectItem 値を受け入れます。すべての項目を選択するアスタリスク (\*) キーワードは受け入れられません。列名のエイリアスも受け入れられません。

どちらのタイプの DISTINCT 節でも、一意性をテストする項目を複数指定できます。複数の項目をリストすると、すべての項目の組み合わせに個別な行がすべて取得されます。DISTINCT では、NULL を一意の値とは見なしません。詳細は、“[DISTINCT \(SQL\)](#)” を参照してください。

## numRows

TOP 節 (例えば、TOP numRows) と組み合わせて使用した場合、返される行数。クエリに ORDER BY 節が含まれていない場合、返される “上位” 行は予測できません。クエリに ORDER BY 節が含まれている場合、上位行は指定された順序に基づきます。クエリで TOP の前に DISTINCT キーワードが含まれている場合、クエリは numRows で指定した数だけ一意の値を返します。numRows は、正の整数、または疑問符 (?) を使用した [ダイナミック SQL](#) 入力パラメータ (正の整数に解決される) として指定します。TOP キーワードが指定されていない場合、既定で SELECT 条件を満たすすべての行が表示されます。

## var

[selectItem](#) の値を配置する 1 つまたは複数の [ホスト変数](#)。複数のホスト変数は、コンマ区切りリストまたは単一のホスト変数配列として指定します。詳細は、“[INTO](#)” を参照してください。

ODBC、JDBC、またはダイナミック SQL を介して処理される SELECT クエリで INTO 節を指定すると、SQLCODE -422 エラーが発生します。

## %keyword

空白で区切られた、1 つまたは複数の %keyword 引数。これらのキーワードは、以下のように処理に影響を与えます。

- ・ **%NOFPLAN** — この操作の凍結されたプラン (ある場合) を無視して、新しいクエリ・プランを生成します。凍結プランは保持されますが、使用されません。詳細は、“[凍結プラン](#)” を参照してください。
- ・ **%NOLOCK** — InterSystems IRIS はテーブルのいずれに対してもロックを実行しません。このキーワードを指定すると、クエリは現在のトランザクションの分離モードに関係なく [READ UNCOMMITTED モード](#) でデータを取得します。詳細は、“[トランザクション処理](#)” を参照してください。
- ・ **%NORUNTIME** — 実行時プラン選択 (RTPC) による最適化を使用しません。
- ・ **%PROFILE** または **%PROFILE\_ALL** — SQLStats 収集コードを生成します。これは、PTools を ON にして生成されるものと同じコードです。違いは、SQLStats 収集コードはこの特定の文に対してのみ生成されるという点です。コンパイルされるルーチンまたはクラス内のその他すべての SQL 文は、PTools が OFF であるかのようにコードを生成します。これにより、調査対象ではない SQL 文の無関係な統計を収集することなく、アプリケーション内で問題のある特定の SQL 文をプロファイリングおよび調査できます。詳細は、“[SQL パフォーマンス分析ツールキット](#)” を参照してください。

%PROFILE はメイン・クエリ・モジュールの SQLStats を収集します。%PROFILE\_ALL はメイン・クエリ・モジュールとそのすべてのサブクエリ・モジュールの SQLStats を収集します。



## 例

### 述語条件を使用したデータのサブセットの選択

述語条件のさまざまな組み合わせを使用して、テーブルからデータのサブセットを選択します。以下の例で示している節は、正しい順序で指定する必要があります。4 つの例すべてで、Sample.Person テーブルから 3 つの列 (Name、Home\_State、および Age) を選択して、他の 2 つの列 (AvgAge および AvgMiddleAge) を計算します。

#### HAVING および ORDER BY

以下のクエリは、Sample.Person 内のすべてのレコードに対して AvgAge 列を計算します。HAVING 節は、Sample.Person 内のすべてのレコードから 40 歳以上の人の平均年齢を計算して、AvgMiddleAge 計算列を設定します。したがって、各行の AvgAge および AvgMiddleAge の値が同じになります。ORDER BY 節は、Home\_State 列の値によって、行の表示をアルファベット順に並べます。

#### SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
HAVING Age > 40
ORDER BY Home_State
```

#### WHERE、HAVING、および ORDER BY

以下のクエリでは、WHERE 節は、指定された北東の 7 つの州に選択を限定します。このクエリは、これらの州のレコードに対して AvgAge 列を計算します。HAVING 節は、指定された Home\_State 列内のレコードで 40 歳以上の人の平均年齢を計算して、AvgMiddleAge 計算列を設定します。したがって、各行の AvgAge および AvgMiddleAge の値が同じになります。ORDER BY 節は、Home\_State 列の値によって、行の表示をアルファベット順に並べます。

#### SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT','NY')
HAVING Age > 40
ORDER BY Home_State
```

#### GROUP BY、HAVING、および ORDER BY

以下の例では、GROUP BY 節により、クエリは Home\_State グループごとに AvgAge 列を計算します。また、GROUP BY 節は、各 Home\_State で最初に見つかったレコードに、出力表示を制限します。HAVING 節は、各 Home\_State グループで 40 歳以上の人の平均年齢を計算して、AvgMiddleAge 計算列を設定します。ORDER BY 節は、Home\_State 列の値によって、行の表示をアルファベット順に並べます。

#### SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
GROUP BY Home_State
HAVING Age > 40
ORDER BY Home_State
```

#### WHERE、GROUP BY、HAVING、および ORDER BY

以下のクエリでは、WHERE 節は、指定された北東の 7 つの州に選択を限定します。GROUP BY 節により、クエリはこれら 7 つの Home\_State グループごとに AvgAge 列を個別に計算します。また、GROUP BY 節は、指定した各 Home\_State で最初に見つかったレコードに、出力表示を制限します。HAVING 節は、7 つの各 Home\_State グループで 40 歳以上の人の平均年齢を計算して、AvgMiddleAge 計算列を設定します。ORDER BY 節は、Home\_State 列の値によって、行の表示をアルファベット順に並べます。



## SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT','NY')
GROUP BY Home_State
HAVING Age > 40
ORDER BY Home_State
```

## 埋め込み SQL およびダイナミック SQL を使用した ObjectScript プログラム内からのデータの選択

埋め込み SQL およびダイナミック SQL を使用して、ObjectScript プログラム内から SELECT クエリを発行できます。

以下の埋め込み SQL プログラムは、あるレコードからデータの値を取得して、それを [INTO](#) 節で指定される出力[ホスト変数](#)に代入します。

## ObjectScript

```
NEW SQLCODE,%ROWCOUNT
&sql(SELECT Home_State,Name,Age
      INTO :a, :b, :c
      FROM Sample.Person)
IF SQLCODE=0 {
  WRITE !,"  Name=",b
  WRITE !,"  Age=",c
  WRITE !," Home Home_State=",a
  WRITE !,"Row count is: ",%ROWCOUNT }
ELSE {
  WRITE !,"SELECT failed, SQLCODE=",SQLCODE }
```

このプログラムで取得される行は多くても 1 行であるため、%ROWCOUNT 変数は、0 または 1 に設定されます。複数の行を取得するには、カーソルを宣言し、[FETCH](#) コマンドを使用する必要があります。詳細は、“[埋め込み SQL の使用法](#)”を参照してください。

以下のダイナミック SQL の例では、まず必要なテーブルが存在するかどうかをテストし、そのテーブルに対する現在のユーザの SELECT 特権をチェックします。次にクエリを実行して結果セットを返します。続いて WHILE ループを使用して、結果セットの最初の 10 レコードに対して %Next メソッドを繰り返し呼び出します。これは、SELECT 文で指定されたように列位置を指定する %GetData メソッドを使用して 3 つの列値を表示します。

## ObjectScript

```
SET tname="Sample.Person"
IF $SYSTEM.SQL.Schema.TableExists(tname)
  & $SYSTEM.SQL.Security.CheckPrivilege($USERNAME,"l","_tname","s")
  {GOTO SpecifyQuery}
ELSE {WRITE "Table unavailable"  QUIT}
SpecifyQuery
SET myquery = 3
SET myquery(1) = "SELECT Home_State,Name,SSN,Age"
SET myquery(2) = "FROM "_tname
SET myquery(3) = "ORDER BY Name"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 {
  SET x=0
  WHILE x < 10 {
    SET x=x+1
    SET status=rset.%Next()
    WRITE rset.%GetData(2)," " /* Name column */
    WRITE rset.%GetData(1)," " /* Home_State column */
    WRITE rset.%GetData(4),! /* Age column */
  }
  WRITE !,"End of Data"
  WRITE !,"SQLCODE=",rset.%SQLCODE," Row Count=",rset.%ROWCOUNT
}
ELSE {
  WRITE !,"SELECT failed, SQLCODE=",rset.%SQLCODE }
```

詳細は、“[ダイナミック SQL の使用法](#)”を参照してください。

## 結果セットの列の大文字/小文字の変更

`selectItem` に指定する列名では、大文字と小文字は区別されません。ただし、列エイリアスを指定する場合以外は、結果セットの列名は、列プロパティに関連付けられた `SqlFieldName` の大文字/小文字に従います。`SqlFieldName` の大文字/小文字は、`selectItem` リストの指定ではなく、テーブル定義で指定された列名と一致します。そのため、`SELECT name FROM Sample.Person` では、列ラベルが `Name` として返されます。列エイリアスを使用すると、表示する大文字/小文字を指定できます。例えば、以下のクエリは、結果セットの `Name` 列を `NAME` (すべて大文字) として表示します。

### SQL

```
SELECT name AS NAME
FROM Sample.Person
```

大文字/小文字の解決には時間がかかります。SELECT のパフォーマンスを最大化するには、テーブル定義の指定に従って、列名と同じ大文字/小文字を指定します。ただし多くの場合、テーブル定義で列の大文字/小文字を指定するのは、不便で誤りが発生しやすいという問題があります。その代わりに、列エイリアスを使用して大文字/小文字の問題を回避することができます。列エイリアスへのすべての参照の大文字/小文字を一致させる必要があることに注意してください。

以下のダイナミック SQL の例では、大文字/小文字の解決が必要です (`SqlFieldNames` は “Latitude” と “Longitude” )。

### ObjectScript

```
set query = "SELECT latitude,longitude FROM Sample.USZipCode"
set statement = ##class(%SQL.Statement).%New()

set status = statement.%Prepare(query)
if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

set rset = statement.%Execute()
if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}

while rset.%Next()
{
    write rset.latitude," ",rset.longitude,!
}
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}
```

以下のダイナミック SQL の例は大文字/小文字の解決が不要なため、より高速に実行されます。

### ObjectScript

```
set query = "SELECT latitude AS northsouth,longitude AS eastwest FROM Sample.USZipCode"
set statement = ##class(%SQL.Statement).%New()

set status = statement.%Prepare(query)
if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

set rset = statement.%Execute()
if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}

while rset.%Next()
{
    write rset.northsouth," ",rset.eastwest,!
}
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}
```

## マルチテーブル・クエリにおける列名の区別

SELECT クエリから返される結果セットには、テーブル・エイリアスの接頭語 `tableAlias` は含まれません。したがって、このクエリは `Name` という名前の列を 2 つ返します。

## SQL

```
SELECT p.Name,e.Name
FROM Sample.Person AS p LEFT JOIN Sample.Employee AS e ON p.Name=e.Name
```

このようなクエリで列を区別するには、列エイリアスを指定します。例えば、以下のように変更したクエリでは、これら 2 つの列を `PersonName` および `EmployeeName` として返します。

## SQL

```
SELECT p.Name AS PersonName,e.Name AS EmployeeName
FROM Sample.Person AS p LEFT JOIN Sample.Employee AS e ON p.Name=e.Name
```

## セキュリティおよび特権

1 つ以上のテーブルで SELECT クエリを実行するには、以下のうち、1 つ以上を持っている必要があります。

- ・ 指定したすべての `selectItem` 列に対する列レベルの SELECT 特権
- ・ 指定した `table` テーブルまたはビューに対するテーブルレベルの SELECT 特権
- ・ そのテーブルのスキーマに対する SELECT 特権

テーブルのエイリアス (`t.Name` や `"MyAlias".Name` など) を使用して指定された `selectItem` 列には、列レベルの SELECT 特権のみが必要で、テーブルレベルの SELECT 特権は不要です。

SELECT \* を使用する場合、列レベルの特権は、GRANT 文内で指定されたすべてのテーブル列に適用されます。テーブルレベルの特権は、特権の割り当て後に追加された列を含め、すべてのテーブル列に適用されます。

必要な特権を持っていないと SQLCODE -99 エラー (特権違反) になります。現在のユーザが SELECT 特権を持っているかどうかを確認するには、[%CHECKPRIV](#) コマンドを呼び出します。`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを呼び出すことにより、指定のユーザがテーブルレベルの SELECT 特権を持っているかどうかを確認できます。特権の割り当ての詳細は、["GRANT"](#) を参照してください。

**注釈** テーブルに対してテーブルレベルの SELECT 特権を保有していても、そのテーブルが実際に存在することが十分に証明されるわけではありません。指定されたユーザが `%All` ロールを保有している場合、指定されたテーブルまたはビューが存在していなくても `CheckPrivilege()` は 1 を返します。

FROM 節のない SELECT クエリでは、SELECT 特権は一切必要ありません。FROM 節のある SELECT クエリでは、そのクエリによってアクセスされる列データがない場合でも、SELECT 特権が必要です。

## 詳細

### SELECT のステータスと返り値

SELECT 操作を実行すると、InterSystems IRIS により、その操作の成功または失敗を示す [SQLCODE](#) ステータス変数が設定されます。また、SELECT 操作を行うと、[%ROWCOUNT](#) ローカル変数に選択された行数が設定されます。通常、SELECT 操作が成功すると、SQLCODE=0 が設定され、%ROWCOUNT には選択された行数が設定されます。埋め込み SQL コードに単純な SELECT 文が含まれる場合、選択されるデータは (多くても) 1 行であるため、SQLCODE=0 が設定され、%ROWCOUNT は 0 または 1 に設定されます。埋め込み SQL SELECT 文でカーソルが宣言されていて、データが複数の行からフェッチされる場合、操作はカーソルがデータの最後に達したときに完了します (SQLCODE=100)。その時点で、%ROWCOUNT は、選択された行の合計数に設定されます。詳細は、["FETCH"](#) を参照してください。

SELECT クエリから返される値を結果セットと呼びます。ダイナミック SQL では、SELECT は取得した値を [%SQL.Statement](#) クラスに格納します。詳細は、["ダイナミック SQL の使用法"](#) および [%SQL.Statement](#) のクラス・リファレンス・ページを参照してください。

SELECT は、SQL 関数、ホスト変数、またはリテラルから値を返す場合にも使用できます。SELECT クエリは、これらのデータベース以外の値を返すことと、テーブルまたはビューでデータを検索することを組み合わせることができます。デー

データベース以外の値を返すことのみに SELECT クエリを使用する場合、FROM 節はオプションです。詳細は、“[FROM](#)”を参照してください。

## シャーディング

シャーディングは SQL クエリに対して透過的で、特殊なクエリ構文は必要ありません。FROM 節で指定されているテーブルがシャーード化されているか、シャーード化されていないかをクエリが認識している必要はありません。同じクエリが、シャーード・テーブルとシャーード化されていないテーブルにアクセスできます。シャーード・テーブルとシャーード化されていないテーブルの間の結合をクエリに含めることもできます。

[シャーード・テーブル](#)は、CREATE TABLE コマンドを使用して定義します。シャーード・マスタ・データ・サーバ上のマスタ・ネームスペース内に定義する必要があります。このマスタ・ネームスペースには、シャーード化されていないテーブルを格納することもできます。

## トランザクション処理

クエリを実行するトランザクションは、READ COMMITTED または READ UNCOMMITTED のいずれかに定義されます。既定は READ UNCOMMITTED です。トランザクション内にないクエリは READ UNCOMMITTED として定義されます。

- ・ READ UNCOMMITTED モードでは、SELECT 文はデータの現在の状態を返します。それには、コミットされていない進行中のトランザクションによってデータに加えられた変更も含まれます。このような変更は、後でロールバックされる場合もあります。
- ・ READ COMMITTED モードの場合、SELECT 文の内容によって動作は異なります。通常、READ COMMITTED モードの SELECT 文は、コミットされたデータに対する挿入と更新による変更のみを返します。削除がコミットされておらず、ロールバックされる可能性がある場合でも、進行中のトランザクションによって削除されたデータ行は返されません。

ただし、SELECT 文に [%NOLOCK](#) キーワード、[DISTINCT](#) 節、または [GROUP BY](#) 節が含まれる場合、SELECT クエリは、コミットされていない現在のトランザクションの間にデータに加えられた変更を含め、現在のデータ状態を返します。SELECT 文内の集約関数でも、指定された列のデータの現在の状態は、コミットされていない変更分を含めて返されます。

詳細は、“[SET TRANSACTION](#)” および “[START TRANSACTION](#)” を参照してください。

## 関連項目

- ・ SELECT 節 : [DISTINCT](#)、[FROM](#)、[GROUP BY](#)、[HAVING](#)、[INTO](#)、[ORDER BY](#)、[TOP](#)、[WHERE](#)
- ・ [JOIN](#)、[UNION](#)
- ・ [CREATE VIEW](#)
- ・ [CREATE TABLE](#)、[ALTER TABLE](#)、[DROP TABLE](#)
- ・ [CREATE QUERY](#)、[DROP QUERY](#)
- ・ [INSERT](#)、[INSERT OR UPDATE](#)、[UPDATE](#)、[DELETE](#)
- ・ [データベースの問い合わせ](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

## SET ML CONFIGURATION (SQL)

ML 構成を既定として設定します。

### 構文

```
SET ML CONFIGURATION ml-configuration-name
```

### 引数

<i>ml-configuration-name</i>	ML 構成の名前です。
------------------------------	-------------

### 説明

SET ML CONFIGURATION コマンドは、指定した ML 構成を、その後のすべての TRAIN MODEL 文のシステムの既定として設定します。1 つの SET ML CONFIGURATION 文で 1 つの ML 構成のみシステムの既定として設定できます。

### 必要なセキュリティ特権

SET ML CONFIGURATION を呼び出すには、USE オブジェクト 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%CHECKPRIV コマンドまたは \$SYSTEM.SQL.Security.CheckPrivilege() メソッドを呼び出すことにより、現在のユーザが USE 特権を持っているかどうかを確認できます。

### 例

```
CREATE MODEL H2OMODEL PREDICTING (label) FROM data
SET ML CONFIGURATION %H2O
TRAIN MODEL H2OMODEL
```

### 関連項目

- ALTER ML CONFIGURATION、CREATE ML CONFIGURATION

# SET OPTION (SQL)

実行オプションを設定します。

## 構文

```
SET OPTION option_keyword = value
```

## 概要

SET OPTION 文では、コンパイル・モード、SQL 構成設定、および日付、時刻、数値規則を管理するロケール設定などの、実行オプションを設定します。SET OPTION 文ごとに、キーワード・オプションを 1 つのみ設定できます。

SET OPTION は、次のオプションをサポートします。

- ・ [AUTO\\_PARALLEL\\_THRESHOLD](#)
- ・ [COMPILEMODE](#)
- ・ [DEFAULT\\_SCHEMA](#)
- ・ [EXACT\\_DISTINCT](#)
- ・ [LOCK\\_ESCALATION\\_THRESHOLD](#)
- ・ [LOCK\\_TIMEOUT](#)
- ・ [PKEY\\_IS\\_IDKEY](#)
- ・ [SUPPORT\\_DELIMITED\\_IDENTIFIERS](#)
- ・ [ロケール・オプション](#) (日付、時刻、数値規則)

SET OPTION は、[ダイナミック SQL](#) (SQL シェルを含む) と[埋め込み SQL](#) で使用できます。

他の SET OPTION 引数 (ここには記載されていません) は、InterSystems IRIS で SQL 互換性に対して解析されますが、実行はしません。

SET OPTION はすぐに作成および実行され、通常一度しか実行されないため、InterSystems IRIS では、ODBC、JDBC、またはダイナミック SQL での SET OPTION にはクエリ・キャッシュは作成されません。

InterSystems IRIS でサポートされているオプションは以下のとおりです。

## AUTO\_PARALLEL\_THRESHOLD

AUTO\_PARALLEL\_THRESHOLD オプションは、自動並列処理が有効な場合に、並列処理をクエリに適用するかどうかを決定する整数 n に設定されます。並列処理にはパフォーマンス・コストが伴うため、並列処理に優位性があるかどうかを判断するしきい値を設定する必要があります。n の値が高いほど、InterSystems SQL クエリが[並列処理](#)を使用して実行される可能性は低くなります。既定値は 3200 です。これは、システム全体の設定です。値 n は、アクセスしたマップで並列処理を実行するために必要なタプルの最小数にほぼ相当します。

[AutoParallel](#) が無効な場合、AUTO\_PARALLEL\_THRESHOLD オプションには何の効果もありません。

このオプションは、`$SYSTEM.SQL.Util.SetOption()` メソッドの `AutoParallelThreshold` オプションを使用して設定することもできます。

詳細は、“[AutoParallelThreshold](#)” を参照してください。

## COMPILEMODE

COMPILEMODE オプションを使用して、現在のネームスペースのコンパイル・モードを、DEFERRED、IMMEDIATE、INSTALL、または NOCHECK に設定します。既定値は IMMEDIATE です。DEFERRED から IMMEDIATE コンパイル・



モードへ変更すると、Deferred Compile Queue のいずれのクラスもすぐにコンパイルされるようになります。すべてのクラスのコンパイルが成功すると、InterSystems IRIS は SQLCODE を 0 に設定します。エラーがある場合は -400 に設定します。クラス・コンパイルのエラーは `^mtemp2` ("Deferred Compile Mode", "Error") に記録されています。SQLCODE が -400 に設定された場合は、このグローバル構造を表示して、エラーの詳細メッセージを調べます。INSTALL コンパイル・モードは DEFERRED コンパイル・モードと似ていますが、テーブルにデータがない DDL インストールにのみ使用してください。

NOCHECK コンパイル・モードは IMMEDIATE と似ていますが、コンパイル時には以下の制約のチェックをスキップします：テーブルが削除される場合、InterSystems IRIS は削除するテーブルを参照する別のテーブルにある外部キー制約をチェックしません。外部キー制約が追加される場合、InterSystems IRIS は既存のデータをチェックしてその外部キーの有効性を保証するとはしません。NOT NULL 制約が追加される場合、InterSystems IRIS は既存のデータに NULL データがあるかチェックしたり、フィールドの既定値を割り当てたりすることはありません。UNIQUE または主キー制約が削除される場合、InterSystems IRIS はそのテーブルまたは別のテーブルの外部キーが、削除されるキーを参照しているかどうかはチェックしません。

このオプションは、`$$$SYSTEM.SQL.Util.SetOption()` メソッドの `CompileMode` オプションを使用して設定することもできます。

## DEFAULT\_SCHEMA

DEFAULT\_SCHEMA オプションは、すべてのネームスペースのシステム全体の既定スキーマを設定します。この既定は、明示的に変更するまで有効です。既定スキーマ名を使用して、すべての未修飾のテーブル、ビュー、またはストアド・プロシージャの名前にスキーマ名を指定できます。

リテラル・スキーマ名を指定するか、`_CURRENT_USER` を指定できます。`_CURRENT_USER` を既定のスキーマ名として指定すると、現在ログインしているプロセスのユーザ名が既定のスキーマ名として割り当てられます。詳細は、“[スキーマ名](#)”を参照してください。

## EXACT\_DISTINCT

EXACT\_DISTINCT ブーリアン・オプションは、システム全体で DISTINCT 処理 (TRUE) と FastDistinct 処理 (FALSE) のどちらを使用するかを指定します。[FastDistinct](#) 処理を使用することが、システム全体の既定です。

EXACT\_DISTINCT=TRUE の場合、GROUP BY および DISTINCT クエリはオリジナルの値を結果に返します。EXACT\_DISTINCT=FALSE で、FastDistinct が有効な場合、インデックスの効果的な使用により(可能な場合)、[DISTINCT](#) および [GROUP BY](#) を含んだ SQL クエリを実行するときの効率が向上します。ただし、このクエリによって返される値は、インデックス内に格納されるときと同じ方法で照合されます。このため、このクエリの結果はすべて大文字になることがあります。これは、大文字と小文字を区別するアプリケーションに影響する場合があります。

このオプションは、`$$$SYSTEM.SQL.Util.SetOption()` メソッドの `FastDistinct` ブーリアン・オプションを使用して設定することもできます。

詳細は、“[FastDistinct](#)”を参照してください。

## LOCK\_ESCALATION\_THRESHOLD

LOCK\_ESCALATION\_THRESHOLD オプションは、行のロックをテーブルのロックにエスカレートする時期を決定する整数 `n` に設定されます。既定値は 1000 です。1 つのトランザクション内で 1 つのテーブルに対して実行された挿入、更新、および削除の数が値 `n` に達すると、テーブルレベル・ロックがトリガされます。これは、すべてのネームスペースのシステム全体の設定です。例えば、ロックしきい値が 1000 で、プロセスがトランザクションを開始してから 2000 行を挿入したとします。1001 番目の行を挿入した後、プロセスは、引き続き個々の行をロックするのではなく、テーブルレベルのロックを取得しようとします。これにより、ロック・テーブルが一杯になるのを防ぐことができます。

このオプションは、`$$$SYSTEM.SQL.Util.SetOption()` メソッドの `LockThreshold` オプションを使用して設定することもできます。

詳細は、“[トランザクションのロックしきい値の変更](#)”を参照してください。



## LOCK\_TIMEOUT

LOCK\_TIMEOUT 数値オプションにより、現在のプロセスの既定のロック・タイムアウトを設定できます。LOCK\_TIMEOUT value は、SQL の実行中にロックを確立しようとする操作を待機する秒数です。ロックの競合が発生すると現在のプロセスでは LOCK、INSERT、UPDATE、DELETE、または SELECT の各操作に対してレコード、テーブル、または他のエンティティを直ちにロックできない場合に、このロックに対するタイムアウトを使用します。InterSystems SQL はタイムアウトになるまでロックを確立しようとしますが、タイムアウトになると SQLCODE -110 または -114 エラーを生成します。

使用できる値は正の整数と 0 です。タイムアウトの設定はプロセスごとに行います。現在のプロセスのロック・タイムアウト設定を確認するには、`$SYSTEM.SQL.Util.GetOption("ProcessLockTimeout")` メソッドを使用します。

現在のプロセスに対してロック・タイムアウトを設定していない場合、これは既定で現在のシステム全体のロック・タイムアウト設定になります。ODBC 接続を切断し、再接続する場合、現在のシステム全体のロック・タイムアウト設定を使用して再接続します。既定のシステム全体のロック・タイムアウトは、10 秒です。

ロックの競合、およびプロセスごとあるいはシステム全体の SQL ロック・タイムアウト設定の詳細は、“**LOCK**” コマンドを参照してください。

## PKEY\_IS\_IDKEY

PKEY\_IS\_IDKEY ブーリアン・オプションでは、主キーもシステム全体で ID キーであるか否かを指定します。利用可能な値は TRUE と FALSE です。TRUE であり、フィールドにデータがない場合、主キーが ID キーとして作成されます。つまり、テーブルの主キーがクラス定義の **IDKey インデックス** にもなります。フィールドにデータがある場合、IDKey インデックスは定義されません。主キーを IDKey インデックスとして定義すると、データ・アクセスはより効率的ですが、一度設定された主キー値は変更できません。設定後は、主キーに割り当てられた値の変更や、別のキーを主キーに割り当てることはできません。このオプションを使用すると、主キーを照合する既定値も変更され、主キー文字列値は EXACT 照合を既定値に設定します。FALSE の場合、主キーと ID キーが別々に定義され、効率が低くなります。ただし、主キー値は変更可能であり、主キー文字列値は既定で現在の**照合タイプの既定値**になります。これは既定では SQLUPPER です。

PKEY\_IS\_IDKEY オプションを設定するには、%Admin\_Manage:USE 特権が必要です。この特権がないと、SQLCODE -99 エラー（特権違反）になります。このオプションを一度設定すると、システム全体ですべてのプロセスに対して有効です。次の方法で、このオプションのシステム全体の既定を設定することもできます。

- ・ システム全体の `$SYSTEM.SQL.Util.SetOption()` メソッド構成オプション `DDLPrimaryKeyNotIDKey`。現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。これにより、「DDL ID」と表示されます。既定値は 1 です。
- ・ 管理ポータル構成設定。[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[DDL を使用して作成されたテーブルの ID キーとして主キーを定義する] の現在の設定を表示または変更します。

この PKEY\_IS\_IDKEY 設定は、他の SET OPTION PKEY\_IS\_IDKEY 経由でリセットするか、InterSystems IRIS 構成を再起動して、InterSystems IRIS システム構成の設定パラメータがリセットされるまで有効です。

## SUPPORT\_DELIMITED\_IDENTIFIERS

既定では、システム全体で**区切り識別子**がサポートされています。SUPPORT\_DELIMITED\_IDENTIFIERS ブーリアン・オプションを使用すると、区切り識別子のサポートをシステム全体について変更できます。利用可能な値は TRUE と FALSE です。TRUE の場合、二重引用符で区切られた文字列は、SQL 文の識別子と考えられます。FALSE の場合、SQL 文の文字列リテラルと考えられます。

SUPPORT\_DELIMITED\_IDENTIFIERS オプションを設定するには、%Admin\_Manage:USE 特権が必要です。この特権がないと、SQLCODE -99 エラー（特権違反）になります。このオプションを一度設定すると、システム全体ですべてのプロセスに対して有効です。別の SET OPTION SUPPORT\_DELIMITED\_IDENTIFIERS によってリセットされるか、`$SYSTEM.SQL.Util.SetOption()` メソッドの `DelimitedIdentifiers` オプションによってシステム全体で変更されるまで、SUPPORT\_DELIMITED\_IDENTIFIERS の設定は有効なままになります。

現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。

## ロケール・オプション

ロケール・オプションは、現在のプロセスの日付、時刻、数値規則の InterSystems IRIS ロケール設定に使用するキーワード・オプションです。利用可能なキーワード・オプションは、AM、DATE\_FORMAT、DATE\_MAXIMUM、DATE\_MINIMUM、DATE\_SEPARATOR、DECIMAL\_SEPARATOR、MIDNIGHT、MINUS\_SIGN、MONTH\_ABBR、MONTH\_NAME、NOON、NUMERIC\_GROUP\_SEPARATOR、NUMERIC\_GROUP\_SIZE、PM、PLUS\_SIGN、TIME\_FORMAT、TIME\_PRECISION、TIME\_SEPARATOR、WEEKDAY\_ABBR、WEEKDAY\_NAME、および YEAR\_OPTION です。これらのオプションはすべてリテラルに設定可能で、いずれも既定（アメリカ英語記述規則）をとります。TIME\_PRECISION オプションの設定は変更できます（以下を参照）。これらオプションのいずれかを無効な値に設定すると、InterSystems IRIS は、“SQLCODE -129 エラー（SET OPTION のロケール・プロパティの値が不正です）”を発行します。日付と時刻の形式およびオプションの詳細は、ObjectScript “[\\$ZDATETIME](#)” 関数を参照してください。

日付/時刻オプション・キーワード	説明
AM	文字列。既定値は 'AM' です。
DATE_FORMAT	整数。既定値は 1 です。使用できる値は 0 から 15 です。これらの日付形式の詳細は、ObjectScript “ <a href="#">\$ZDATE</a> ” 関数を参照してください。
DATE_MAXIMUM	整数。既定値は 2980013 (12/31/9999) です。前の日付は設定できますが、後の日付はできません。
DATE_MINIMUM	正整数。既定値は 0 (12/31/1840) です。後の日付は設定できますが、前の日付はできません。
DATE_SEPARATOR	文字。既定値は '/' です。
DECIMAL_SEPARATOR	文字。既定値は '.' です。
MIDNIGHT	文字列。既定値は 'MIDNIGHT' です。
MINUS_SIGN	文字。既定値は '-' です。
MONTH_ABBR	文字列。既定値は 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec' です（この文字列は、既定の区切り文字である空白文字で始まります）。
MONTH_NAME	文字列。既定値は 'January February March April May June ...November December' です。（この文字列は、既定の区切り文字である空白文字で始まります）。
NOON	文字列。既定値は 'NOON' です。
NUMERIC_GROUP_SEPARATOR	文字。既定値は ',' です。
NUMERIC_GROUP_SIZE	整数。既定値は 3 です。
PM	文字列。既定値は 'PM' です。
PLUS_SIGN	文字。既定値は '+' です。
TIME_FORMAT	整数。既定値は 1 です。使用できる値は 1 から 4 です。これらの時刻形式の詳細は、ObjectScript “ <a href="#">\$ZTIME</a> ” 関数を参照してください。
TIME_PRECISION	0 ～ 9 の整数。既定値は 0。秒の小数部の桁数。以下のように設定できます。
TIME_SEPARATOR	文字。既定値は ':' です。

日付/時刻オプション・キーワード	説明
WEEKDAY_ABBR	文字列。既定値は 'Sun Mon Tue Wed Thu Fri Sat' です (この文字列は、既定の区切り文字である空白文字で始まります)。
WEEKDAY_NAME	文字列。既定値は 'Sunday Monday Tuesday Wednesday Thursday Friday Saturday' です (この文字列は、既定の区切り文字である空白文字で始まります)。
YEAR_OPTION	整数。既定値は 0 です。使用できる値は 0 から 6 です。2 桁および 4 桁の年のこれらの表示方法の詳細は、ObjectScript “\$ZDATE” 関数を参照してください。

TIME\_PRECISION をシステム全体で構成するには、管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GETDATE()、CURRENT\_TIME、および CURRENT\_TIMESTAMP の既定の時間精度] の現在の設定を表示して編集します。これは、秒の小数部の精度の桁数を示します。既定値は 0 です。有効な値の範囲は、0 ～ 9 桁の精度です。実際の秒の小数部の有効桁数はプラットフォームにより異なります。

## 関連項目

- SQL 日付および時刻の関数: [CURRENT\\_TIMESTAMP](#)、[DATEPART](#)、[DATENAME](#)、[GETDATE](#)、[NOW](#)
- SQL 日付関数: [DAYNAME](#)、[DAYOFWEEK](#)、[DAYOFMONTH](#)、[DAYOFYEAR](#)、[WEEK](#)、[MONTH](#)、[MONTHNAME](#)、[QUARTER](#)、[YEAR](#)、[CURDATE](#)、[CURRENT\\_DATE](#)、[TO\\_DATE](#)
- SQL 時刻関数 : [HOUR](#)、[MINUTE](#)、[SECOND](#)、[CURTIME](#)、[CURRENT\\_TIME](#)
- [SQL およびオブジェクトの設定ページ](#)
- [SQLCODE エラー・メッセージ](#)
- ObjectScript 関数: [\\$ZDATE](#) [\\$ZDATETIME](#)[\\$ZTIME](#)

# SET TRANSACTION (SQL)

トランザクションのパラメータを設定します。

## 構文

```
SET TRANSACTION [%COMMITMODE commitmode]  
SET TRANSACTION [transactionmodes]
```

## 説明

SET TRANSACTION 文は、現在のプロセスの SQL トランザクションを管理するパラメータを設定します。これらのパラメータは、次のトランザクションの開始時に発効し、現在のプロセスの間、または明示的にリセットされるまでの間、有効となります。トランザクションの最後に、自動的に既定値にリセットされることはありません。

単一の SET TRANSACTION 文を使用して、commitmode パラメータまたは transactionmodes パラメータを設定できますが、両方を設定することはできません。

START TRANSACTION コマンドを使用しても同じパラメータを設定できます。このコマンドでは、パラメータの設定と新しいトランザクションの開始を両方実行することができます。メソッド呼び出しを使用しても、パラメータを設定することができます。

SET TRANSACTION ではトランザクションは開始されず、[\\$TLEVEL](#) トランザクション・レベル・カウンタもインクリメントされません。

SET TRANSACTION は、[ダイナミック SQL](#) (SQL シェルを含む) と [埋め込み SQL](#) で使用できます。

## %COMMITMODE

%COMMITMODE キーワードを使用すると、トランザクションのコミットを自動的に実行するかどうかを指定できます。使用可能なオプションは以下のとおりです。

- ・ IMPLICIT : トランザクションの自動コミットをオンにする (既定)。プログラムがデータベース変更操作 (INSERT、UPDATE、または DELETE) を発行すると、SQL はトランザクションを自動的に開始します。操作が正常に完了し、SQL が変更を自動的にコミットするか、操作がすべての行では正常に完了できず、SQL が操作全体を自動的にロールバックするまで、トランザクションは継続されます。各データベース操作 (INSERT、UPDATE、または DELETE) は、個別のトランザクションを構成します。データベース操作の実行が成功すると、自動的にロールバックのジャーナルがクリアされ、ロックが解放され、[\\$TLEVEL](#) がデクリメントされます。COMMIT 文は必要ありません。これがデフォルト設定です。
- ・ EXPLICIT : トランザクションの自動コミットをオフにする。プログラムが最初のデータベース変更操作 (INSERT、UPDATE、または DELETE) を発行すると、SQL はトランザクションを自動的に開始します。トランザクションは明示的に終了されるまで継続します。正常に終了したら、COMMIT 文を発行します。データベース変更操作が失敗したら、ROLLBACK 文を発行して、データベースをトランザクションが始まる前の時点に戻します。EXPLICIT モードでは、トランザクションあたりのデータベース操作の数は、ユーザ定義です。
- ・ NONE : トランザクション処理を自動化しない。START TRANSACTION 文で明示的に呼び出さない限り、トランザクションは開始されません。COMMIT 文または ROLLBACK 文を発行して、トランザクションを明示的に終了する必要があります。このため、データベース操作がトランザクションに含まれるかどうかと、トランザクション内のデータベース操作の数は、ユーザ定義です。

TRUNCATE TABLE は、自動的に開始されたトランザクション内では発生しません。TRUNCATE TABLE のジャーナリングおよびロールバックが必要な場合、明示的に START TRANSACTION を指定し、明示的な COMMIT または ROLLBACK で終わる必要があります。

現在のプロセスの %COMMITMODE 設定を確認するには、以下の ObjectScript の例のように、GetOption("AutoCommit") メソッドを使用します。

## ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
IF x=1 {
    WRITE "%COMMITMODE IMPLICIT (default behavior):",!,
        "each database operation is a separate transaction",!,
        "with automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "%COMMITMODE NONE:",!,
        "No automatic transaction support",!,
        "You must use START TRANSACTION to start a transaction",!,
        "and COMMIT or ROLLBACK to conclude one" }
ELSE {
    WRITE "%COMMITMODE EXPLICIT:",!,
        "the first database operation automatically",!,
        "starts a transaction; to end the transaction",!,
        "explicit COMMIT or ROLLBACK required" }
```

%COMMITMODE は、SetOption() メソッドを SET

status=\$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval) のように使用して、ObjectScript 内で設定できます。使用可能なメソッドの値は、0 (NONE)、1 (IMPLICIT)、および 2 (EXPLICIT) です。

## ISOLATION LEVEL

クエリを発行しているプロセスの ISOLATION LEVEL を指定します。ISOLATION LEVEL オプションを使用すると、進行中の変更に対してクエリが読み取りアクセスできるようにするかどうかを指定できます。別の同時プロセスがテーブルに対して挿入または更新を行っており、それらのテーブルへの変更がトランザクション内である場合、それらの変更は進行中で、ロールバックされる可能性があります。テーブルのクエリを行っているプロセスの ISOLATION LEVEL を設定することで、これらの進行中の変更をクエリ結果に含めるか排除するかを指定できます。

- ・ READ UNCOMMITTED は、すべての変更分がクエリ・アクセスで即座に使用できることを示します。これには、その後ロールバックされる可能性のある変更が含まれます。READ UNCOMMITTED では、クエリは同時挿入または更新プロセスを待機せずに結果を返すため、ロック・タイムアウト・エラーにより失敗することはありません。ただし、READ UNCOMMITTED の結果にはコミットされていない値が含まれる場合があります。挿入または更新操作が部分的にしか完了していないため、これらの値は内部的に不整合となり、その後ロールバックされる可能性があります。クエリ・プロセスが明示的なトランザクション内にない場合、またはトランザクションで ISOLATION LEVEL が指定されない場合、READ UNCOMMITTED が既定となります。READ UNCOMMITTED は、READ WRITE アクセスと互換性がありません。同じ文で両方を指定すると、SQLCODE -92 エラーが発生します。
- ・ READ VERIFIED は、その他のトランザクションのコミットされていないデータがすぐに使用可能であり、ロックが実行されないことを示しています。これには、その後ロールバックされる可能性のある変更が含まれます。ただし、READ UNCOMMITTED とは異なり、READ VERIFIED トランザクションは、クエリ条件を満たさない出力となるコミットされていないデータや新たにコミットされたデータによって無効にされる可能性があるすべての条件を再確認します。この条件の再確認のため、READ VERIFIED は READ UNCOMMITTED よりも正確ですが、効率性は劣ります。READ VERIFIED は、条件による確認が行われるデータに対する同期更新が行われる可能性がある場合のみ使用してください。READ VERIFIED は、READ WRITE アクセスと互換性がありません。同じ文で両方を指定しようとすると、SQLCODE -92 エラーになります。
- ・ READ COMMITTED は、コミットされた変更分のみがクエリ・アクセスで使用できることを示します。これにより、実行されている変更のグループではなく、後にロールバックできる変更のグループの間で、クエリをデータベース内で一貫性のある状態で実行できます。要求されたデータは変更されているが、コミット（またはロールバック）されていない場合、クエリはトランザクションが完了するまで待機します。このデータが使用可能になるまで待機している間にロック・タイムアウトが発生した場合、SQLCODE -114 エラーが発行されます。

## READ UNCOMMITTED と READ VERIFIED の違い

以下の例では、READ UNCOMMITTED と READ VERIFIED の違いが示されています。

## SQL

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```



クエリ・オプティマイザは、まず、Name インデックスから `>= 'M'` の条件を満たす名前を含むすべての RowID を収集する選択を行う場合があります。収集後、出力用の Name フィールドと SSN フィールドを取得するために、Person テーブルへのアクセスが行われます（一度に 1 つの RowID）。同時に実行されている更新トランザクションにより、クエリによるインデックスからの RowID の収集とテーブルへの行単位のアクセスの間に、'Smith' から 'Abel' までの RowID 72 の Person の Name フィールドが変更される可能性があります。この場合、インデックスからの RowID の収集に `Name >= 'M'` の条件を満たさなくなった行の RowID が含まれます。

READ UNCOMMITTED のクエリ処理は、Name `>= 'M'` の条件がインデックスによって満たされていることを前提としており、インデックスから収集された RowID ごとにテーブル内にある Name を出力します。したがって、この例では、条件を満たさない 'Abel' の Name が含まれる行が出力されます。

READ VERIFIED のクエリ処理は、インデックスによってこれまで満たされていた条件に属する出力 (Name) 用のテーブルからフィールドを取得していることを示し、インデックスが検証された時点からフィールド値が変更されている場合に条件を再確認します。再確認時、行が条件を満たさなくなったことを示し、出力からそれを除外します。出力で必要となる値のみ条件の再確認が行われます。この例において、`SELECT SSN FROM Person WHERE Name >= 'M'` は、RowID 72 の行を出力します。

### READ COMMITTED に対する例外

コミットされた ISOLATION LEVEL 読み取りが有効な場合、ISOLATION LEVEL READ COMMITTED の設定により、または `SetOption()` メソッドを `SET status=$SYSTEM.SQL.Util.SetOption("IsolationMode",1,.oldval)` のように使用して、SQL はコミットされたデータへの変更のみを取得できます。ただし、この規則には以下のような重要な例外があります。

- 行を削除したトランザクションが進行中で、その削除がその後ロールバックされる可能性があっても、**削除された行** がクエリによって返されることはありません。ISOLATION LEVEL READ COMMITTED では、挿入と更新は一貫性のある状態ですが、削除は一貫性のある状態ではありません。
- クエリに**集約関数**が含まれる場合、指定された ISOLATION LEVEL に関係なく、集約結果によりデータの現在の状態が返されます。そのため、進行中の（その後ロールバックされる可能性がある）挿入と更新は、集約結果に含まれます。進行中の（その後ロールバックされる可能性がある）削除は、集約結果に含まれません。これは、集約操作ではテーブルの多数の行のデータにアクセスする必要があるためです。
- DISTINCT 節**または **GROUP BY 節**を含む SELECT クエリは、ISOLATION LEVEL 設定による影響を受けません。これらの節のいずれかを含むクエリは、データの現在の状態を返します。それには、その後ロールバックされる可能性のある進行中の変更が含まれます。これは、これらのクエリ操作では、テーブルの多数の行のデータにアクセスする必要があるためです。
- %NOLOCK キーワード**を使用するクエリ

注釈 InterSystems IRIS で **ECP (Enterprise Cache Protocol)** を実装して READ COMMITTED を使用した場合、READ UNCOMMITTED と比べてパフォーマンスが著しく低下する場合があります。開発者は ECP を含むトランザクションを定義する際に、READ UNCOMMITTED を使用してパフォーマンスを得るか、READ COMMITTED を使用してデータの精度を得るかを十分に検討する必要があります。

詳細は、“[トランザクション処理](#)”を参照してください。

### 有効な ISOLATION LEVEL

SET TRANSACTION (トランザクションを開始しない)、START TRANSACTION (分離モードを設定してトランザクションを開始)、または `SetOption("IsolationMode")` メソッド呼び出しを使用して、プロセスの ISOLATION LEVEL を設定できます。

指定された ISOLATION LEVEL は、SET TRANSACTION、START TRANSACTION、または `SetOption("IsolationMode")` メソッド呼び出しによって明示的にリセットされるまで有効です。COMMIT または ROLLBACK はデータの変更にも有効で、データ・クエリの変更には有効でないため、COMMIT または ROLLBACK の操作は ISOLATION LEVEL の設定に影響を与えません。

クエリの開始時に有効になっている ISOLATION LEVEL は、クエリ中も有効なままになります。

現在のプロセスの ISOLATION LEVEL を判別するには、GetOption("IsolationMode") メソッド呼び出しを使用します。また、現在のプロセスの分離モードを設定するには、SetOption("IsolationMode") メソッド呼び出しを使用します。これらのメソッドでは、READ UNCOMMITTED (既定) を 0、READ COMMITTED を 1、READ VERIFIED を 3 に指定します。他の数値を指定すると、分離モードが変更されません。分離モードを現在の分離モードに設定すると、エラーまたは変更は発生しません。これらのメソッドの使用法を以下の例に示します。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after START TRANSACTION",!
DO $SYSTEM.SQL.Util.SetOption("IsolationMode",0,.stat)
IF stat=1 {
    WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after IsolationMode=0 call",! }
ELSE { WRITE "Set IsolationMode error" }
&sql(COMMIT)
```

分離モードとアクセス・モードは、常に互換性がある必要があります。以下の例のように、アクセス・モードを変更する場合は、分離モードを変更します。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after SET TRANSACTION",!
&sql(START TRANSACTION READ ONLY)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after changing access mode",!
&sql(COMMIT)
```

## 引数

### %COMMITMODE commitmode

トランザクションがデータベースにコミットされる方法を指定する引数(オプション)。利用可能な値は、EXPLICIT、IMPLICIT、そして NONE です。既定は IMPLICIT です。

### transactionmodes

トランザクションのアクセス・モードと分離モードを指定する引数(オプション)。分離モード、アクセス・モード、またはコママ区切りリストとして両方のモードの値を指定できます。

分離モードの有効な値は、ISOLATION LEVEL READ COMMITTED、ISOLATION LEVEL READ UNCOMMITTED、および ISOLATION LEVEL READ VERIFIED です。既定は、ISOLATION LEVEL READ UNCOMMITTED です。

アクセス・モードに有効な値は READ ONLY と READ WRITE です。アクセス・モード READ WRITE と互換性があるのは ISOLATION LEVEL READ COMMITTED のみです。

## 例

以下の埋め込み SQL の例では、2 つの SET TRANSACTION 文を使用して、トランザクション・パラメータを設定しています。SET TRANSACTION では、トランザクション・レベル(\$TLEVEL) がインクリメントされないことに注意してください。START TRANSACTION コマンドでトランザクションを開始して、\$TLEVEL をインクリメントします。



## ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction commit mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED)
WRITE !,"Set transaction isolation mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

## 関連項目

- ・ [COMMIT、ROLLBACK、SAVEPOINT、START TRANSACTION、\\$TLEVEL](#)
- ・ [トランザクション処理](#)

# START TRANSACTION (SQL)

トランザクションを開始します。

## 構文

```
START TRANSACTION [%COMMITMODE commitmode]
START TRANSACTION [transactionmodes]
```

## 概要

START TRANSACTION はトランザクションを開始します。START TRANSACTION は、現在のコミット・モードの設定に関係なく、即座にトランザクションを開始します。START TRANSACTION で開始したトランザクションは、現在のコミット・モードの設定に関係なく、明示的な COMMIT または ROLLBACK を発行して終了する必要があります。

START TRANSACTION を使用するかどうかは任意です。

- ・ プロセスがデータのクエリのみを行っている場合 (SELECT 文)、SET TRANSACTION を使用して ISOLATION LEVEL を設定できます。START TRANSACTION は不要です。
- ・ プロセスがデータを変更中の場合、START TRANSACTION を発行して SQL トランザクションを明示的に開始する必要があるかどうかは、プロセスの現在のコミット・モード設定 (自動コミット設定とも呼ばれます) によって決定されます。現在のプロセスのコミット・モードが IMPLICIT または EXPLICIT の場合は、START TRANSACTION を発行するかどうかは任意です。START TRANSACTION を省略した場合、データ変更操作 (DELETE、UPDATE、または INSERT) を呼び出すと、システムは自動的にトランザクションを開始します。START TRANSACTION を指定すると、トランザクションは即座に開始され、明示的な COMMIT または ROLLBACK で終了する必要があります。

START TRANSACTION はトランザクションを開始すると、\$TLEVEL トランザクション・レベル・カウンタを 0 から 1 にインクリメントし、トランザクションが進行中であることを示します。また、%INTRANSACTION 文によって設定されている SQLCODE をチェックして、トランザクションが進行中かどうかを確認することもできます。トランザクションが進行中の場合、START TRANSACTION を発行しても \$TLEVEL または %INTRANSACTION には効果がありません。

InterSystems SQL では、入れ子になったトランザクションはサポートされません。トランザクションが既に進行中の場合、START TRANSACTION を発行してもトランザクションを開始せず、エラー・コードも返しません。InterSystems SQL では、トランザクションの部分的なロールバックを許可するセーブポイントがサポートされています。

トランザクションが進行中ではない場合に SAVEPOINT 文を発行すると、SAVEPOINT はトランザクションを開始します。ただし、この方法でトランザクションを開始することはお勧めしません。

トランザクションの動作が正常完了できなかった場合は、SQLCODE -400 が発行されます。

## パラメータ設定

必要に応じて、START TRANSACTION を使用して、パラメータを設定することができます。指定したパラメータ設定は、即座に有効になります。ただし、START TRANSACTION で開始したトランザクションは、commitmode パラメータの設定に関係なく、明示的な COMMIT または ROLLBACK で終了する必要があります。パラメータ設定は、現在のプロセスの間または明示的にリセットされるまで、有効になります。トランザクションの最後に、自動的に既定値にリセットされることはありません。

単一の START TRANSACTION 文を使用して、commitmode パラメータまたは transactionmodes パラメータを設定できますが、両方を設定することはできません。両方を設定するには、SET TRANSACTION および START TRANSACTION または 2 つの START TRANSACTION 文を発行します。最初の START TRANSACTION は、トランザクションを開始します。

START TRANSACTION を発行した後に、トランザクション中に別の START TRANSACTION、SET TRANSACTION、またはメソッド呼び出しを発行して、これらのパラメータ設定を変更できます。commitmode パラメータを変更しても、現在のトランザクションを明示的な COMMIT または ROLLBACK で終了するという要件がなくなることはありません。

SET TRANSACTION 文を使用すると、トランザクションを開始せずに commitmode パラメータまたは transactionmodes パラメータを設定できます。また、これらのパラメータは、トランザクション内部または外部のメソッド呼び出しを使用しても設定できます。

## %COMMITMODE

%COMMITMODE キーワードを使用すると、現在のプロセスでトランザクションの開始とコミットの自動化を指定できます。START TRANSACTION %COMMITMODE は、現在のプロセスの今後すべてのトランザクションのコミット・モードの設定を変更します。START TRANSACTION 文で開始されたトランザクションには効果がありません。現在のまたは設定されているコミット・モードに関係なく、START TRANSACTION は即座にトランザクションを開始し、このトランザクションは、明示的な COMMIT または ROLLBACK を発行して終了する必要があります。

使用可能な %COMMITMODE のオプションは以下のとおりです。

- ・ IMPLICIT : トランザクションの自動コミットをオンにする (最初のプロセスの既定)。プログラムがデータベース変更操作 (INSERT、UPDATE、または DELETE) を発行すると、SQL はトランザクションを自動的に開始します。操作が正常に完了し、SQL が変更を自動的にコミットするか、操作がすべての行では正常に完了できず、SQL が操作全体を自動的にロールバックするまで、トランザクションは継続されます。各データベース操作 (INSERT、UPDATE、または DELETE) は、個別のトランザクションを構成します。データベース操作の実行が成功すると、自動的にロールバックのジャーナルがクリアされ、ロックが解放され、\$TLEVEL がデクリメントされます。COMMIT 文は必要ありません。
- ・ EXPLICIT : トランザクションの自動コミットをオフにする。プログラムが最初のデータベース変更操作 (INSERT、UPDATE、または DELETE) を発行すると、SQL はトランザクションを自動的に開始します。トランザクションは明示的に終了されるまで継続します。正常に終了したら、COMMIT 文を発行します。データベース変更操作が失敗したら、ROLLBACK 文を発行して、データベースをトランザクションが始まる前の時点に戻します。EXPLICIT モードでは、複数のデータベース変更操作で 1 つのトランザクションを構成できます。
- ・ NONE : トランザクション処理を自動化しない。トランザクションは、START TRANSACTION で明示的に呼び出されるまで開始されません。COMMIT 文または ROLLBACK 文を発行して、すべてのトランザクションを明示的に終了する必要があります。このため、データベース操作がトランザクションに含まれるかどうかと、トランザクション内のデータベース操作の数は、ユーザ定義です。

TRUNCATE TABLE は、自動的に開始されたトランザクション内では発生しません。TRUNCATE TABLE のジャーナリングおよびロールバックが必要な場合、明示的に START TRANSACTION を指定し、明示的な COMMIT または ROLLBACK で終わる必要があります。

SetOption() メソッドを SET status=\$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval) のように使用して、ObjectScript 内で %COMMITMODE を設定できます。使用可能なメソッドの値は、0 (NONE)、1 (IMPLICIT)、および 2 (EXPLICIT) です。

注釈 シャード・テーブルは常に、自動コミットなしのモード (SetOption("AutoCommit",0)) に設定されます。つまり、シャード・テーブルに対する挿入、更新、および削除はすべて、トランザクションの範囲外で実行されます。

## ISOLATION LEVEL

クエリを発行しているプロセスの ISOLATION LEVEL を指定します。ISOLATION LEVEL オプションを使用すると、進行中の変更に対してクエリが読み取りアクセスできるようにするかどうかを指定できます。別の同時プロセスがテーブルに対して挿入または更新を行っており、それらのテーブルへの変更がトランザクション内である場合、それらの変更は進行中で、ロールバックされる可能性があります。テーブルのクエリを行っているプロセスの ISOLATION LEVEL を設定することで、これらの進行中の変更をクエリ結果に含めるか排除するかを指定できます。

- ・ READ UNCOMMITTED は、すべての変更分がクエリ・アクセスで即座に使用できることを示します。これには、その後ロールバックされる可能性のある変更が含まれます。READ UNCOMMITTED では、クエリは同時挿入または更新プロセスを待機せずに結果を返すため、ロック・タイムアウト・エラーにより失敗することはありません。ただし、READ UNCOMMITTED の結果にはコミットされていない値が含まれる場合があります。挿入または更新操作が部分的に

しか完了していないため、これらの値は内部的に不整合となり、その後ロールバックされる可能性があります。クエリ・プロセスが明示的なトランザクション内でない場合、またはトランザクションで ISOLATION LEVEL が指定されない場合、READ UNCOMMITTED が既定となります。READ UNCOMMITTED は、READ WRITE アクセスと互換性がありません。同じ文で両方を指定すると、SQLCODE -92 エラーが発生します。

- ・ READ VERIFIED は、その他のトランザクションのコミットされていないデータがすぐに使用可能であり、ロックが実行されないことを示しています。これには、その後ロールバックされる可能性のある変更が含まれます。ただし、READ UNCOMMITTED とは異なり、READ VERIFIED トランザクションは、クエリ条件を満たさない出力となるコミットされていないデータや新たにコミットされたデータによって無効にされる可能性があるすべての条件を再確認します。この条件の再確認のため、READ VERIFIED は READ UNCOMMITTED よりも正確ですが、効率性は劣ります。READ VERIFIED は、条件による確認が行われるデータに対する同期更新が行われる可能性がある場合のみ使用してください。READ VERIFIED は、READ WRITE アクセスと互換性がありません。同じ文で両方を指定しようとすると、SQLCODE -92 エラーになります。
- ・ READ COMMITTED は、コミットされた変更分のみがクエリ・アクセスで使用できることを示します。これにより、実行されている変更のグループではなく、後にロールバックできる変更のグループの間で、クエリをデータベース内で一貫性のある状態で実行できます。要求されたデータは変更されているが、コミット（またはロールバック）されていない場合、クエリはトランザクションが完了するまで待機します。このデータが使用可能になるまで待機している間にロック・タイムアウトが発生した場合、SQLCODE -114 エラーが発行されます。

## READ UNCOMMITTED と READ VERIFIED の違い

以下の例では、READ UNCOMMITTED と READ VERIFIED の違いが示されています。

### SQL

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```

クエリ・オプティマイザは、まず、Name インデックスから  $\geq$  'M' の条件を満たす名前を含むすべての RowID を収集する選択を行う場合があります。収集後、出力用の Name フィールドと SSN フィールドを取得するために、Person テーブルへのアクセスが行われます（一度に 1 つの RowID）。同時に実行されている更新トランザクションにより、クエリによるインデックスからの RowID の収集とテーブルへの行単位のアクセスの間に、'Smith' から 'Abel' までの RowID 72 の Person の Name フィールドが変更される可能性があります。この場合、インデックスからの RowID の収集に Name  $\geq$  'M' の条件を満たさなくなった行の RowID が含まれます。

READ UNCOMMITTED のクエリ処理は、Name  $\geq$  'M' の条件がインデックスによって満たされていることを前提としており、インデックスから収集された RowID ごとにテーブル内にある Name を出力します。したがって、この例では、条件を満たさない 'Abel' の Name が含まれる行が出力されます。

READ VERIFIED のクエリ処理は、インデックスによってこれまで満たされていた条件に属する出力 (Name) 用のテーブルからフィールドを取得していることを示し、インデックスが検証された時点からフィールド値が変更されている場合に条件を再確認します。再確認時、行が条件を満たさなくなったことを示し、出力からそれを除外します。出力で必要となる値のみ条件の再確認が行われます。この例において、SELECT SSN FROM Person WHERE Name  $\geq$  'M' は、RowID 72 の行を出力します。

## READ COMMITTED に対する例外

コミットされた ISOLATION LEVEL 読み取りが有効な場合、ISOLATION LEVEL READ COMMITTED の設定により、または SetOption() メソッドを SET status=\$SYSTEM.SQL.Util.SetOption("IsolationMode",1,.oldval) のように使用して、SQL はコミットされたデータへの変更のみを取得できます。ただし、この規則には以下のような重要な例外があります。

- ・ 行を削除したトランザクションが進行中で、その削除がその後ロールバックされる可能性があっても、**削除された行** がクエリによって返されることはありません。ISOLATION LEVEL READ COMMITTED では、挿入と更新は一貫性のある状態ですが、削除は一貫性のある状態ではありません。
- ・ クエリに**集約関数**が含まれる場合、指定された ISOLATION LEVEL に関係なく、集約結果によりデータの現在の状態が返されます。そのため、進行中の（その後ロールバックされる可能性がある）挿入と更新は、集約結果に含ま

れます。進行中の（その後ロールバックされる可能性がある）削除は、集約結果に含まれません。これは、集約操作ではテーブルの多数の行のデータにアクセスする必要がありますためです。

- ・ **DISTINCT 節**または **GROUP BY 節**を含む SELECT クエリは、ISOLATION LEVEL 設定による影響を受けません。これらの節のいずれかを含むクエリは、データの現在の状態を返します。それには、その後ロールバックされる可能性のある進行中の変更が含まれます。これは、これらのクエリ操作では、テーブルの多数の行のデータにアクセスする必要がありますためです。
- ・ **%NOLOCK キーワード**を使用するクエリ

**注釈** InterSystems IRIS で **ECP (Enterprise Cache Protocol)** を実装して READ COMMITTED を使用した場合、READ UNCOMMITTED と比べてパフォーマンスが著しく低下する場合があります。開発者は ECP を含むトランザクションを定義する際に、READ UNCOMMITTED を使用してパフォーマンスを得るか、READ COMMITTED を使用してデータの精度を得るかを十分に検討する必要があります。

詳細は、“[トランザクション処理](#)”を参照してください。

### 有効な ISOLATION LEVEL

SET TRANSACTION (トランザクションを開始しない)、START TRANSACTION (分離モードを設定してトランザクションを開始)、または SetOption("IsolationMode") メソッド呼び出しを使用して、プロセスの ISOLATION LEVEL を設定できます。

指定された ISOLATION LEVEL は、SET TRANSACTION、START TRANSACTION、または SetOption("IsolationMode") メソッド呼び出しによって明示的にリセットされるまで有効です。COMMIT または ROLLBACK はデータの変更にのみ有効で、データ・クエリの変更には有効でないため、COMMIT または ROLLBACK の操作は ISOLATION LEVEL の設定に影響を与えません。

クエリの開始時に有効になっている ISOLATION LEVEL は、クエリ中も有効なままになります。

現在のプロセスの ISOLATION LEVEL を判別するには、GetOption("IsolationMode") メソッド呼び出しを使用します。また、現在のプロセスの分離モードを設定するには、SetOption("IsolationMode") メソッド呼び出しを使用します。これらのメソッドでは、READ UNCOMMITTED (既定) を 0、READ COMMITTED を 1、READ VERIFIED を 3 に指定します。他の数値を指定すると、分離モードが変更されません。分離モードを現在の分離モードに設定すると、エラーまたは変更は発生しません。これらのメソッドの使用法を以下の例に示します。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after START TRANSACTION",!
DO $SYSTEM.SQL.Util.SetOption("IsolationMode",0,.stat)
IF stat=1 {
    WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after IsolationMode=0 call",! }
ELSE { WRITE "Set IsolationMode error" }
&sql(COMMIT)
```

分離モードとアクセス・モードは、常に互換性がある必要があります。以下の例のように、アクセス・モードを変更する場合は、分離モードを変更します。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after SET TRANSACTION",!
&sql(START TRANSACTION READ ONLY)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after changing access mode",!
&sql(COMMIT)
```



## 引数

### commitmode

現在のプロセス中にその後のトランザクションがデータベースにコミットされる方法を指定する引数 (オプション)。有効な値は、EXPLICIT、IMPLICIT、および NONE です。既定では、既存のコミット・モードが維持されます。プロセスの最初のコミット・モードの既定値は、IMPLICIT です。

### transactionmodes

トランザクションのアクセス・モードと分離モードを指定する引数 (オプション)。分離モード、アクセス・モード、またはコンマ区切りリストとして両方のモードの値を指定できます。

分離モードの有効な値は、ISOLATION LEVEL READ COMMITTED、ISOLATION LEVEL READ UNCOMMITTED、および ISOLATION LEVEL READ VERIFIED です。既定は、ISOLATION LEVEL READ UNCOMMITTED です。

アクセス・モードに有効な値は READ ONLY と READ WRITE です。アクセス・モード READ WRITE と互換性があるのは ISOLATION LEVEL READ COMMITTED のみです。

## ObjectScript と SQL のトランザクション

ObjectScript と SQL のトランザクション・コマンドは完全に互換性があり、置き換え可能ですが、以下の例外があります。

ObjectScript TSTART と SQL START TRANSACTION はどちらも、トランザクションが進行中でない場合にトランザクションを開始します。ただし、START TRANSACTION では、入れ子になったトランザクションはサポートされません。そのため、入れ子になったトランザクションが必要な場合 (または必要になる可能性がある場合) には、トランザクションを TSTART で始めることをお勧めします。SQL 標準との互換性が必要な場合は、START TRANSACTION を使用してください。

ObjectScript トランザクション処理は、入れ子になったトランザクションを限定的にサポートします。SQL トランザクション処理はトランザクション内のセーブポイントをサポートします。

トランザクションに SQL データ変更文が含まれる場合、SQL の START TRANSACTION 文でトランザクションが開始され、COMMIT 文でコミットされます (これらの文は、%COMMITMODE 設定の設定によって明示的または暗黙的になります)。TSTART/TCOMMIT を入れ子にして使用するメソッドは、トランザクションを開始するものでなければ、トランザクションに組み込むことができます。メソッドとストア・プロシージャは、通常、設計でトランザクションの主要なコントローラにならない限り、SQL トランザクション制御文を使用しません。ストア・プロシージャは、独自のトランザクション制御モデルの ODBC/JDBC から呼び出されるため、通常 SQL トランザクション制御文を使用しません。

## 例

以下の埋め込み SQL の例では、2 つの START TRANSACTION 文を使用して、トランザクションを開始し、そのパラメータを設定しています。最初の START TRANSACTION では、トランザクションを開始し、コミット・モードを設定して、\$TLEVEL トランザクション・レベル・カウンタをインクリメントしています。2 番目の START TRANSACTION では、現在のトランザクションのクエリ読み取り操作に分離モードを設定しますが、トランザクションは既に開始されているため、\$TLEVEL をインクリメントしていません。SAVEPOINT 文は \$TLEVEL をインクリメントします。

### ObjectScript

```
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Start transaction commit mode, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED)
WRITE !,"Start transaction isolation mode, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

## 関連項目

- ・ [COMMIT、%INTRANSACTION、ROLLBACK、SAVEPOINT、SET TRANSACTION、\\$TLEVEL](#)
- ・ [トランザクション処理](#)



# TRAIN MODEL (SQL)

機械学習モデルをトレーニングします。

## 構文

```
TRAIN MODEL model-name
  [ AS preferred-name ]
  [ NOT DEFAULT ]
  [ FOR label-column ]
  [ WITH feature-column-clause ]
  [ FROM model-source ]
  [ USING json-object ]
```

## 引数

model-name	トレーニングする機械学習モデルの名前。
AS preferred-name	オプション - トレーニングされたモデルを保存する代替名。この後の説明を参照してください。
NOT DEFAULT	オプション - 既定のトレーニング済みモデルとして設定せずにモデルをトレーニングするための節。この後の説明を参照してください。
FOR label-column	オプション - 予測される列名 (ラベル列)。この後の説明を参照してください。
WITH feature-column-clause	オプション - 列名として、または列名のコンマ区切りリストとしての、モデル (特徴列) への入力。
FROM model-source	モデルの構築元にするテーブルまたはビュー。テーブル、ビュー、または結合の結果です。この後の説明を参照してください。
USING json-object-string	オプション - 1 つ以上のキーと値のペアを指定する JSON 文字列。この後の説明を参照してください。

## 説明

TRAIN MODEL 文はプロバイダに、指定したモデル定義を使用してモデルをトレーニングするよう伝えます。プロバイダは、ML 構成によって指定されます。

### FROM

FROM 節は、モデルをトレーニングするためのデータを提供します。

- ・ CREATE MODEL 文で FROM 節を指定しなかった場合、この節は必須です。
- ・ CREATE MODEL 文で FROM 節を指定した場合、この節はオプションです。

FROM の許容される使用と省略を示す例

#### TRAIN MODEL の FROM

```
CREATE MODEL model_b PREDICTING ( label ) WITH ( column_1, column_2, column_3 )
TRAIN MODEL model_b FROM table
```

#### CREATE MODEL の FROM

```
CREATE MODEL model_a PREDICTING ( label ) FROM table
TRAIN MODEL model_a
```

注釈 TRAIN MODEL 文から FROM を省略する場合は、CREATE MODEL から既定のクエリを使用します。

## WITH

WITHを使用すると、データの特徴列をモデル定義のスキーマに明示的に合わせることができます。各列は、標準の識別子です。

## FOR

FORを使用すると、データのラベル列をモデル定義のスキーマに明示的に合わせることができます。例えば、モデル定義のラベル列には `column_a` という名前が付けられているが、トレーニング・データでは `column_b` という名前が付けられている場合、次のように列を合わせることができます。

```
CREATE MODEL model_a PREDICTING ( column_a ) FROM table_a
TRAIN MODEL model_a FOR column_b FROM table_b
```

## 名前付け

ASを使用すると、トレーニングされたモデルに明示的に名前を付けることができます。

モデル定義とトレーニングされたモデルは、同じスキーマに存在しています。トレーニングされたモデルに AS を使用して明示的に名前が付けられていない場合、名前はモデル定義名に実行中の整数が付加されたものになります。

INFORMATION\_SCHEMA.ML\_TRAINED\_MODELS テーブルをクエリして、その違いを確認できます。

```
CREATE MODEL TitanicModel PREDICTING (Survived binary) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel
TRAIN MODEL TitanicModel
TRAIN MODEL TitanicModel
TRAIN MODEL TitanicModel AS TrainedTitanic
SELECT MODEL_NAME, TRAINED_MODEL_NAME FROM INFORMATION_SCHEMA.ML_TRAINED_MODELS
```

MODEL_NAME	TRAINED_MODEL_NAME
TitanicModel	TitanicModel_t1
TitanicModel	TitanicModel_t2
TitanicModel	TitanicModel_t3
TitanicModel	TrainedTitanic

## NOT DEFAULT

各モデル定義には、既定のトレーニング済みモデルがあります。ユーザが指定しない場合、最後にトレーニングされたモデルが既定になります。NOT DEFAULT 節を使用すると、新しいモデルをトレーニングし、それを既定のトレーニング済みモデルにしないことができます。

```
CREATE MODEL TitanicModel PREDICTING (Survived) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel As FirstModel
TRAIN MODEL TitanicModel As SecondModel NOT DEFAULT
SELECT MODEL_NAME, DEFAULT_TRAINED_MODEL_NAME FROM INFORMATION_SCHEMA.ML_MODELS
```

MODEL_NAME	DEFAULT_TRAINED_MODEL_NAME
TitanicModel	FirstModel

NOT DEFAULT を使用しない場合、DEFAULT\_TRAINED\_MODEL フィールドは別の方法で “SecondModel” を読み取ります。

## USING 節の考慮事項

トレーニング実行をよりカスタマイズするために、USING 節でプロバイダ固有のパラメータを渡すことができます。この節は、1 つ以上のキーと値のペアで構成される JSON 文字列を受け入れます。使用できるパラメータのリストは、プロバイダによって異なります。

例えば、AutoML をプロバイダとして使用してトレーニングする場合、ランダム・シードを変更できます。

```
TRAIN MODEL IsSpam USING {"seed": 3}
```

各プロバイダに渡すことができるパラメータの詳細は、“[プロバイダ](#)”を参照してください。

## NULL 値を渡す方法

TRAIN MODEL 文のラベル列で NULL 値のデータを渡すと、トレーニングされたモデルの動作が不定になります。このため、データ準備プロセスの一環として、NULL 値がないかどうかを注意深く調べる必要があります。

## 必要なセキュリティ特権

TRAIN MODEL を呼び出すには、%MANAGE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%MANAGE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## 例

```
TRAIN MODEL EmailFilter
```

```
TRAIN MODEL model_5 AS MyModel USING {"seed": 3}
```

```
TRAIN MODEL LoanDefault FROM LoanData
```

## 関連項目

- ・ [CREATE MODEL](#)、[VALIDATE MODEL](#)

## TRUNCATE TABLE (SQL)

テーブルからすべてのデータを削除し、カウンタをリセットします。

### 構文

```
TRUNCATE TABLE [restriction] tablename
```

### 説明

TRUNCATE TABLE コマンドは、テーブルからすべての行を削除し、すべてのテーブル・カウンタをリセットします。

TRUNCATE TABLE により、[RowID フィールド](#)、[IDENTITY フィールド](#)、および [SERIAL \(%Library.Counter\) フィールド](#) の連続した整数値を生成するために使用される内部カウンタがリセットされます。InterSystems IRIS では、TRUNCATE TABLE に従ってテーブルに挿入される最初の行にこれらのフィールドの値 1 を割り当てます。テーブルのすべての行で [DELETE](#) を実行しても、これらの内部カウンタはリセットされません。

TRUNCATE TABLE は、データがストリーム・フィールドに挿入されるときに[ストリーム・フィールド OID 値](#)を生成するために使用される内部カウンタをリセットします。テーブルのすべての行で [DELETE](#) を実行しても、この内部カウンタはリセットされません。

TRUNCATE TABLE は [%ROWCOUNT](#) ローカル変数を常に -1 に設定します。[%ROWCOUNT](#) を削除された行数に設定することはありません。

TRUNCATE TABLE では [ROWVERSION](#) カウンタはリセットされません。

TRUNCATE TABLE は、DELETE 処理中にベース・テーブル・トリガの実行を抑制して、トリガが引き出されないようにします。TRUNCATE TABLE は [%NOTRIGGER](#) 動作によって削除を実行するため、TRUNCATE TABLE を実行するには、ユーザに [%NOTRIGGER](#) 特権が付与されている必要があります ([GRANT](#) 文を使用)。TRUNCATE TABLE はこの点で、機能的には以下と同一です。

### SQL

```
DELETE %NOTRIGGER FROM tablename
```

**注釈** [DELETE](#) コマンドは、テーブルからすべての行を削除するのに也可以使用できます。DELETE は TRUNCATE TABLE より多くの機能を提供します。これには、[%ROWCOUNT](#) に削除された行数を返す機能も含まれます。DELETE は内部カウンタをリセットしません。

TRUNCATE TABLE は、その他のデータベース・ソフトウェアからのコード移行との互換性を提供します。

テーブルを切り捨てるには、以下の条件を満たしている必要があります。

- ・ テーブルは、現在の (または指定された) ネームスペースに存在している必要があります。  
ビューの名前が *tablename* 引数として指定されている場合、サブクエリが *tablename* 引数として指定されている場合、または指定されたテーブルが見つからない場合、InterSystems IRIS は SQLCODE -30 エラーを発行します。
- ・ トリガが定義されていない場合でも、ユーザは [%NOTRIGGER](#) **管理者特権** を持っている必要があります。この権限を持っていない場合、SQLCODE -99 エラーが発生し、*%msg* が "[%NOTRIGGER](#)" に設定されます。
- ・ ユーザは、テーブルに対する DELETE 特権を持っている必要があります。この権限を持っていない場合、SQLCODE -99 エラーが発生し、*%msg* が "[%NOTRIGGER](#)" に設定されます。[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが DELETE 特権を持っているかどうかを確認できます。`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを呼び出すことにより、指定のユーザが DELETE 特権を持っているかどうかを確認できます。特権の割り当てについては、"[GRANT](#)" コマンドを参照してください。

- ・ テーブルを READONLY として定義することはできません。読み取り専用テーブルを参照する TRUNCATE TABLE をコンパイルすると、SQLCODE -115 エラーが発生します。このエラーは実行時にのみ発生するのではなく、コンパイル時にも発生するようになったことに注意してください。“[永続クラスのその他のオプション](#)” の READONLY オブジェクトの説明を参照してください。
- ・ すべての行が削除可能である必要があります。既定では、1 行以上の行が削除不可能である場合、TRUNCATE TABLE 操作は失敗し、行は削除されません。

テーブルが別のプロセスによって EXCLUSIVE MODE または SHARE MODE でロックされている場合、TRUNCATE TABLE は失敗します。ロックされているテーブルに対して TRUNCATE TABLE 操作を試行すると、SQLCODE -110 エラーになり、%msg は “ ‘Sample.MyStuff’ DELETE RowID = '3' ” (指定された RowID はテーブルの最初の行です) になります。

行の削除が外部キーの参照整合性に違反する場合、TRUNCATE TABLE は失敗します。行は削除されず、TRUNCATE TABLE は SQLCODE -124 エラーを発行します。この既定の動作は以下のように変更できます。

## アトミック性

TRUNCATE TABLE は自動的に開始されたトランザクション内では発生しません。したがって、ジャーナリングやロールバック・オプションは提供されません。

ジャーナリングおよび TRUNCATE TABLE のロールバック・オプションが必要な場合、明示的に START TRANSACTION を指定し、明示的な COMMIT または ROLLBACK で終わる必要があります。

これは [SET TRANSACTION %COMMITMODE= NONE](#) または 0 (自動トランザクションなし) と同じです。TRUNCATE TABLE を呼び出してもトランザクションは開始されません。TRUNCATE TABLE 操作の失敗により、行の一部が削除されたり削除されなかったりすることで、データベースが整合性のない状態になる可能性があります。このモードでトランザクションのサポートを提供するには、START TRANSACTION を使用してトランザクションを開始し、COMMIT または ROLLBACK を使用してトランザクションを終了する必要があります。

シャード・テーブルの TRUNCATE TABLE は、明示的に SET TRANSACTION %COMMITMODE EXPLICIT を設定した場合であっても、常に SET TRANSACTION %COMMITMODE NONE を使用して実行されます。

## 制限引数

restriction 引数を使用するには、現在のネームスペースに対応する [admin-privilege](#) が必要となります。詳細は “[GRANT](#)” を参照してください。

restriction 引数を指定すると、以下のように処理を制限します。

- ・ %NOCHECK – 削除される行を参照する外部キーの参照整合性チェックを抑制します。
- ・ %NOLOCK – 削除される行の行のロックを抑制します。単独のユーザ/処理がデータベースを更新する際にのみ使用します。
- ・ %NOJOURN – 削除される行のジャーナリングを抑制し、削除中のトランザクションを無効化します。起動されたトリガを含め、行での変更はどれもジャーナリングされません。%NOJOURN が指定された文の後で ROLLBACK を実行した場合、その文で行われた変更はロールバックされません。

複数の restriction 引数を順不同で指定できます。複数の引数は、空白で区切られます。

親レコードの削除時に restriction 引数を指定すると、対応する子レコードの削除時に同じ restriction 引数が適用されます。

TRUNCATE TABLE は常に暗黙的な %NOTRIGGER 動作による削除を実行し、対応する admin-privilege を必要とします。

## 参照整合性

InterSystems IRIS では、システム全体の構成設定を使用して外部キーの参照整合性チェックを実行するかどうかが決まります。既定では、外部キーの参照整合性チェックを実行します。“[外部キーの参照整合性チェック](#)”の説明に従って、この既定値をシステム全体で設定できます。現在のシステム全体の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。

TRUNCATE TABLE 操作時には、すべての外部キー参照について、参照されるテーブルの該当する行に対する共有ロックが取得されます。この行は、トランザクションの終了までロックされています。これにより、参照される行は、TRUNCATE TABLE のロールバックがあってもそれより前に変更されることがなくなります。

## トランザクションでのロック

InterSystems IRIS は、TRUNCATE TABLE 操作時に標準のロックを実行します。一意フィールドの値は、現行のトランザクションの間、ロックされます。

既定のロックしきい値は、テーブルごとに 1000 ロックです。つまり、トランザクションの間にテーブルから 1000 件を超える一意のフィールド値を削除すると、ロックのしきい値に到達し、InterSystems IRIS は自動的にロック・レベルを一意フィールド値ロックからテーブル・ロックに上げます。これによってトランザクション時に、ロック・テーブルをオーバーフローすることなく、大規模な削除を実行できます。

現在のシステム全体のロックしきい値は、`$SYSTEM.SQL.Util.GetOption("LockThreshold")` メソッドを使用して確認できます。このシステム全体のロックしきい値は、以下の方法を使用して設定できます。

- ・ `$SYSTEM.SQL.Util.SetOption("LockThreshold")` メソッドを使用します。
- ・ 管理ポータルを使用する。[\[システム管理\]](#)、[\[構成\]](#)、[\[SQL およびオブジェクトの設定\]](#)、[\[SQL\]](#) の順に移動します。[\[ロック・エスカレーションしきい値\]](#) の現在の設定を表示して編集します。

ロックしきい値を変更するには、%Admin Manage Resource の USE 許可が必要です。InterSystems IRIS は、ロックしきい値の変更を現在のプロセスすべてに即座に適用します。

トランザクションでのロックの詳細は、“[トランザクション処理](#)”を参照してください。

## インポートされた SQL のコード

ImportDDL("IRIS") および Run() メソッドは TRUNCATE TABLE コマンドをサポートしません。これらのメソッドによってインポートされた SQL コード・ファイルの TRUNCATE TABLE コマンドは無視されます。これらのインポート・メソッドは DELETE コマンドをサポートします。

## 引数

### restriction

%NOCHECK、%NOLOCK の[制約キーワード](#)のうちの 1 つ、またはこれらのキーワードの空白で区切られたリストを表す引数 (オプション)。

### tablename

すべての行を削除する[テーブル](#)。テーブル名は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。未修飾の名前は、[スキーマ検索パス](#) (指定されている場合) または[既定のスキーマ名](#)を使用して、そのスキーマと照合されます。

## 例

以下の 2 つのダイナミック SQL の例では、DELETE と TRUNCATE TABLE を比較しています。それぞれの例では、テーブルを作成し、テーブルに行を挿入し、テーブル内のすべての行を削除してから、空になったテーブルに単一の行を挿入しています。



最初の例では **DELETE** を使用して、テーブル内のすべてのレコードを削除します。DELETE は RowID カウンタをリセットしないことに注意してください。

### ObjectScript

```
SET tcreate = "CREATE TABLE SQLUser.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)"
SET tinsert = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "_
               "SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET tinsert1 = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) VALUES ('Bob Jones',60123)"
SET tdelete = "DELETE SQLUser.MyStudents"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(tcreate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT,!

NEW %ROWCOUNT,%ROWID
SET qStatus = tStatement.%Prepare(tinsert)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tdelete)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tinsert1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT, " RowID ",rset.%ROWID,!
&sql(DROP TABLE SQLUser.MyStudents)
```

2 番目の例では TRUNCATE TABLE を使用して、テーブル内のすべてのレコードを削除します。%StatementTypeName は TRUNCATE TABLE に対して “DELETE” を返すことに注意してください。TRUNCATE TABLE は RowID カウンタをリセットすることに注意してください。

### ObjectScript

```
SET tcreate = "CREATE TABLE SQLUser.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)"
SET tinsert = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "_
               "SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET tinsert1 = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) VALUES ('Bob Jones',60123)"
SET ttrunc = "TRUNCATE TABLE SQLUser.MyStudents"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(tcreate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT,!

NEW %ROWCOUNT,%ROWID
SET qStatus = tStatement.%Prepare(tinsert)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(ttrunc)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " (TRUNCATE TABLE) rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tinsert1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT, " RowID ",rset.%ROWID,!
&sql(DROP TABLE SQLUser.MyStudents)
```

## 関連項目

- [DELETE、INSERT、UPDATE](#)
- [テーブルの定義](#)
- [トランザクション処理](#)



- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

# TUNE TABLE (SQL)

代表的なデータに基づいてテーブル統計を収集します。

## 構文

```
TUNE TABLE tablename [ tune_options ]
```

## 説明

TUNE TABLE コマンドは、テーブル内の現在のデータに基づいて既存のテーブルの統計を収集します。このデータは、テーブルが完全に生成されたときに予想されるデータの代表である必要があります。

TUNE TABLE は、テーブルの BlockCount とエクステント・サイズ、およびそれぞれのフィールドの選択性を、代表的なデータに基づいて計算し、設定します。通常、TUNE TABLE はこれらの値を 1 つ以上設定し、これらの新しい値をクエリが使用するように、その永続クラス(テーブル)を使用するクエリ・キャッシュをすべて削除します。ただし、TUNE TABLE がこれらのいずれの値も変更しない場合 (TUNE TABLE がそのテーブルに対して最後に実行されてからデータが変更されていない場合など)、クエリ・キャッシュは削除されず、テーブルのクラス定義にリコンパイルのフラグは設定されません。

TUNE TABLE は、SQL テーブル定義を更新します (そのため、テーブル定義を変更する特権が必要です)。一般に、TUNE TABLE は対応する永続クラス定義も更新します。これにより、クラスのコンパイルを必要とせずに、クエリ・オブティマイザで収集された統計を使用することができます。ただし、[クラスが導入済み](#)の場合、TUNE TABLE は SQL テーブル定義のみを更新します。クエリ・オブティマイザは、収集した統計をテーブル定義から間接的に使用します。

成功した場合、TUNE TABLE は SQLCODE = 0 を設定します。指定した tablename が存在しない場合、TUNE TABLE は SQLCODE -30 エラーを発行します。

## 特権

TUNE TABLE コマンドは特権を必要とする操作です。TUNE TABLE を実行するには、ユーザは [%ALTER\\_TABLE 管理特権](#)を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %ALTER\_TABLE " に設定されます。適切な付与特権を持っていれば、[GRANT](#) コマンドを使用して、ユーザまたはロールに %ALTER\_TABLE 特権を割り当てることができます。管理特権はネームスペース固有のものです。詳細は、["特権"](#) を参照してください。

ユーザは、指定されたテーブルに対する %ALTER 特権を持っている必要があります。ユーザがテーブルの所有者 (作成者)である場合、ユーザにはそのテーブルに対する %ALTER 特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する %ALTER 特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' " に設定されます。[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが %ALTER 特権を持っているかどうかを確認できます。[GRANT](#) コマンドを使用して、指定したテーブルに %ALTER 特権を割り当てることができます。詳細は、["特権"](#) を参照してください。

## TUNE TABLE のオプション

- ・ [%CLEAR\\_VALUES](#) : 指定した場合、SELECTIVITY、EXTENTSIZES などの既存の値がクラス定義とテーブル定義からクリアされます。このオプションを指定しない場合、既定のテーブル・チューニング動作が提供されます。
- ・ [%SAMPLE\\_PERCENT](#) percentage : テーブルのチューニング・ユーティリティでデータのサンプリングに使用されるテーブルの行の割合を指定します。この percentage は、'.##' または '##%' として指定できます。例えば、'.12' または '12%' を指定すると、このコマンドは、データをサンプリングするときにテーブル内の行の 12% を使用します。0 より大きく、100% 以下の値を使用して percentage を指定してください。この範囲外の値を指定すると、SQLCODE -1 エラーが発行されます。

通常、この値を指定する必要はありません。この値は、フィールドの潜在的な異常値がテーブル全体の行に均等に分散していない場合にのみ指定します。エクステント・サイズが 1000 行未満のテーブルについては、%SAMPLE\_PERCENT の値に関係なく、エクステント全体がテーブルのチューニングで使用されます。

- ・ %RECOMPILE\_CQ：指定した場合、チューニングされたテーブルについて単にクエリ・キャッシュを削除するのではなく、テーブルのチューニングは、新しいテーブル・チューニング統計を使用してクエリ・キャッシュ・クラスをリコンパイルします。このオプションを指定しない場合、既定のテーブル・チューニング動作が提供されます。

指定した tune\_options 値が存在しない場合、TUNE TABLE は SQLCODE -25 エラーを発行します。同じ tune\_options 値を 2 回指定した場合、TUNE TABLE は SQLCODE -326 エラーを発行します。

## クエリ・キャッシュ

TUNE TABLE を実行すると、クエリ・キャッシュが作成されます。[プラン表示] の表示は、クエリ・プランが作成されないことを示します。SQL 文は作成されません。クエリ・キャッシュはネームスペースに対して一般的です。特定のテーブルについてはリストされません。クエリ・キャッシュを使用して、同じ TUNE TABLE 文を再実行できます。

TUNE TABLE を実行すると、TUNE TABLE の前回の実行のクエリ・キャッシュを含め、指定したテーブルの既存のクエリ・キャッシュがすべて削除されます。オプションで、TUNE TABLE が新しいテーブル・チューニング値を使用してこれらのクエリ・キャッシュすべてをリコンパイルするように指定することもできます。

TUNE TABLE を実行してもテーブル・チューニング値が変更されない場合、クエリ・キャッシュは削除されません。

## 他の方法によるテーブル・チューニングの実行

テーブル・チューニングを実行するためのインタフェースが他にも 2 つあります。

- ・ [管理ポータル](#)の [SQL インタフェース](#)の [アクション] [ドロップダウン・リスト](#)を使用します。テーブルのチューニングを 1 つのテーブルに対して実行することもスキーマ内のすべてのテーブルに対して実行することもできます。
- ・ 1 つのテーブル、または現在のネームスペース内のすべてのテーブルに対して、\$SYSTEM.SQL.Stats.Table.GatherTableStats() メソッドを呼び出します。

詳細は、“[テーブルのチューニング](#)”を参照してください。

## 引数

### tablename

統計を収集する既存のテーブルの名前。[テーブル名](#)は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。

### tune\_options

指定する場合、1 つまたは複数の TUNE TABLE オプションをスペースで区切って任意の順序で指定します。これらの tune\_options は、大文字と小文字を区別しません。

## 例

以下の例では、Sample.MyTest table の 30% をサンプリングすることにより、テーブル統計を収集します。

### SQL

```
TUNE TABLE Sample.MyTest %SAMPLE_PERCENT '30%'
```

以下の例では、テーブル統計を収集し、新たに収集された統計に基づいてクエリ・キャッシュ・クラスをリコンパイルします。

## SQL

```
TUNE TABLE Sample.MyTest %RECOMPILE_CQ
```

以下の例では、テーブルの 40% のサンプルからテーブル統計を収集します。

## SQL

```
TUNE TABLE Sample.MyTest %SAMPLE_PERCENT '40%'
```

## 関連項目

- ・ [テーブルのチューニング](#)
- ・ [ExtentSize、Selectivity、および BlockCount](#)

## UNFREEZE PLANS (SQL)

1 つ以上の凍結したクエリ・プランを凍結解除します。

### 構文

```
UNFREEZE PLANS [[FROM] UPGRADE] BY ID statement-hash
UNFREEZE PLANS [[FROM] UPGRADE] BY TABLE table-name
UNFREEZE PLANS [[FROM] UPGRADE] BY SCHEMA schema-name
UNFREEZE PLANS [[FROM] UPGRADE]
```

### 説明

UNFREEZE PLANS コマンドは、凍結したクエリ・プランを凍結解除します。クエリ・プランを凍結するには、[FREEZE PLANS](#) コマンドを使用します。

FROM UPGRADE 節を指定せずに UNFREEZE PLANS を使用すると、プランの状態が / のすべてのクエリ・プランが凍結解除されます。FROM UPGRADE 節を指定して UNFREEZE PLANS を使用すると、プランの状態が / のすべてのクエリ・プランが凍結解除されます。この節の FROM キーワードはオプションです。

UNFREEZE PLANS には、クエリ・プランを凍結解除するための 4 つの構文形式が用意されています。

- 指定されたクエリ・プラン：UNFREEZE PLANS BY ID *statement-hash*。*statement-hash* 値は、二重引用符で区切る必要があります。
- テーブルのすべてのクエリ・プラン：UNFREEZE PLANS BY TABLE *table-name*。テーブル名またはビュー名を指定できます。クエリ・プランが複数のテーブルまたはビューを参照する場合、これらのテーブルまたはビューのいずれかを指定すると、クエリ・プランは凍結解除されます。
- スキーマ内のすべてのテーブルのすべてのクエリ・プラン：UNFREEZE PLANS BY SCHEMA *schema-name*。
- 現在のネームスペースのすべてのテーブルのすべてのクエリ・プラン：UNFREEZE PLANS。

このコマンドは、1 つ以上のクエリ・プランが凍結解除される場合は SQLCODE 0 を発行し、クエリ・プランが凍結解除されない場合は SQLCODE 100 を発行します。影響を受けた行 (%ROWCOUNT) は、凍結解除されたクエリ・プランの数を示します。

### その他のインタフェース

次の `$SYSTEM.SQL.Statement` メソッドを使用して 1 つのクエリ・プランまたは複数のクエリ・プランを凍結解除できます：1 つのプランの場合は `UnfreezeStatement()`、関係のすべてのプラン (クエリ・プランで参照されるテーブルまたはビュー) の場合は `UnfreezeRelation()`、スキーマのすべてのプランの場合は `UnfreezeSchema()`、現在のネームスペースのすべてのプランの場合は `UnfreezeAll()`。対応する `Freeze` メソッドがあります。

管理ポータルを使用して、クエリ・プランを凍結解除できます。詳細は、“[凍結プランのインタフェース](#)”を参照してください。

### 引数

#### statement-hash

引用符で囲まれた、クエリ・プランの SQL 文の定義の内部ハッシュ表現。場合によっては、同じ SQL 文のように見えても、文のハッシュ・エントリが異なることがあります。SQL 文の異なるコード生成を必要とする設定/オプションの相違によって、異なる文ハッシュが生成されます。これは、異なる内部最適化をサポートする異なるクライアント・バージョンや異なるプラットフォームで発生することがあります。“[SQL 文の詳細](#)”を参照してください。

### table-name

既存のテーブルまたはビューの名前。table-name は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。

### schema-name

既存のスキーマの名前。このコマンドは、指定されたスキーマ内のすべてのテーブルの、凍結されたすべてのクエリ・プランを凍結解除します。

## セキュリティおよび特権

UNFREEZE PLANS コマンドは、ユーザに %Development:USE 権限が必要な特権操作です。このような権限は管理ポータルを介して付与できます。この特権なしで UNFREEZE PLANS コマンドを実行すると、SQLCODE -99 エラーが発生し、コマンドは失敗します。以下の 2 つの例外があります。

- ・ 埋め込み SQL を介してコマンドが実行され、特権チェックを実行しない。
- ・ ユーザが特権チェックなしを明示的に指定する (例えば、checkPriv 引数を 0 に設定して %Prepare() を呼び出したり、%SQL.Statement で %ExecDirectNoPriv() を呼び出したりすることによって指定)。

## 関連項目

- ・ [FREEZE PLANS](#) コマンド
- ・ [凍結プラン](#)
- ・ [SQL 文](#)

## UNLOCK (SQL)

テーブルをアンロックします。

### 構文

```
UNLOCK [TABLE] tablename IN EXCLUSIVE MODE [IMMEDIATE]
```

```
UNLOCK [TABLE] tablename IN SHARE MODE [IMMEDIATE]
```

### 説明

UNLOCK コマンドは、LOCK コマンドがロックした SQL テーブルをアンロックします。このテーブルは、ユーザが適切な特権を持つ既存のテーブルである必要があります。tablename が一時テーブルである場合、コマンドは正常に完了しますが、処理は何も実行されません。tablename がビューである場合、SQLCODE -400 エラーが発生してコマンドが失敗します。

UNLOCK と UNLOCK TABLE は同義語です。

UNLOCK コマンドは LOCK 操作を取り消します。UNLOCK コマンドは、適用されているロックがない場合も正常に完了します。LOCK を使用してテーブルを複数回ロックすることができますが、明示的にロックした回数と同じ回数だけ明示的に UNLOCK を適用する必要があります。

### 特権

UNLOCK コマンドは特権を必要とする操作です。UNLOCK IN SHARE MODE を使用する前に、指定されたテーブルに対する SELECT 特権をプロセスに付与しておく必要があります。UNLOCK IN EXCLUSIVE MODE を使用する前に、指定されたテーブルに対する INSERT、UPDATE、および DELETE 特権をプロセスに付与しておく必要があります。IN EXCLUSIVE MODE では、テーブルの 1 つ以上のフィールドに対する INSERT、または UPDATE 特権が必要です。特権がない場合は、SQLCODE -99 エラー（特権違反）が返されます。[%CHECKPRIV](#) コマンドを呼び出すことにより、現在のユーザが必要な特権を持っているかどうかを確認できます。`$SYSTEM.SQL.Security.CheckPrivilege()` メソッドを呼び出すことにより、指定のユーザが必要なテーブルレベルの特権を持っているかどうかを確認できます。特権の割り当てについては、["GRANT"](#) コマンドを参照してください。

### 存在しないテーブル

存在しないテーブルをアンロックしようとすると、コンパイル・エラーが発生して UNLOCK が失敗し、[SQLCODE=-30 : 'SQLUser.mytable' ] というメッセージが表示されます。

### 引数

#### tablename

アンロックする[テーブル](#)の名前。tablename は既存のテーブルでなければなりません。tablename は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。テーブル名が未修飾の場合は、[既定のスキーマ名](#)が使用されます。[スキーマ検索パス](#)は無視されます。

### IN EXCLUSIVE MODE と IN SHARE MODE

キーワード句 IN EXCLUSIVE MODE は、通常の InterSystems IRIS ロックを解放します。キーワード句 IN SHARE MODE は、InterSystems IRIS レベルの共有ロックを解放します。

### IMMEDIATE

オプションの引数です。指定しない場合、InterSystems IRIS は現在のトランザクションの終了時にロックを解放します。指定した場合、InterSystems IRIS はロックを即時に解放します。



## 例

以下の埋め込み SQL の例は、テーブルを作成してロックしてから、アンロックします。

### ObjectScript

```
NEW SQLCODE,%msg
&sql(CREATE TABLE mytest (
    ID NUMBER(12,0) NOT NULL,
    CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),
    WORK_START DATE DEFAULT SYSDATE) )
IF SQLCODE=0 { WRITE !,"Table created" }
ELSE { WRITE !,"CREATE TABLE error: ",SQLCODE
      QUIT }
```

### ObjectScript

```
NEW SQLCODE,%msg
&sql(LOCK mytest IN EXCLUSIVE MODE)
IF SQLCODE=0 { WRITE !,"Table locked" }
ELSEIF SQLCODE=-110 { WRITE !,"Table is locked by another process",!,%msg }
ELSE { WRITE !,"Unexpected LOCK error: ",SQLCODE,!,%msg }
&sql(UNLOCK mytest IN EXCLUSIVE MODE)
IF SQLCODE=0 { WRITE !,"Table unlocked" }
ELSE { WRITE !,"Unexpected UNLOCK error: ",SQLCODE,!,%msg }
```

## 関連項目

- ・ [LOCK](#)
- ・ [INSERT UPDATE DELETE](#)
- ・ [SQLCODE エラー・メッセージ](#)

## UPDATE (SQL)

指定されたテーブルの指定された列に新しい値を設定します。

### 構文

```
UPDATE [%keyword] table-ref [[AS] t-alias]
  SET column1 = scalar-expression1 {,column2 = scalar-expression2} ...
  [FROM [optimize-option] select-table [[AS] t-alias] {, select-table2 [[AS] t-alias]} ]
  [WHERE condition-expression]
UPDATE [%keyword] table-ref [[AS] t-alias]
  [ (column1 {,column2} ...) ] VALUES (scalar-expression1 {,scalar-expression2} ...)
  [FROM ... ] [WHERE ...]
UPDATE [%keyword] table-ref [[AS] t-alias]
  VALUES :array()
  [FROM ... ] [WHERE ...]

UPDATE [%keyword] table-ref [[AS] t-alias]
  SET column1 = scalar-expression1 {,column2 = scalar-expression2} ...
  [WHERE CURRENT OF cursor]
UPDATE [%keyword] table-ref [[AS] t-alias] [ (column1 {,column2} ...) ]
  VALUES (scalar-expression1 {,scalar-expression2} ...)
  [WHERE CURRENT OF cursor]
UPDATE [%keyword] table-ref [[AS] t-alias]
  VALUES :array()
  [WHERE CURRENT OF cursor]
```

### 概要

UPDATE コマンドは、テーブルの列の既存値を変更します。テーブルのデータの更新には、直接行う方法、[ビュー](#)を使用した方法、または括弧で囲んだサブクエリを使用した方法があります。ビューを使用して更新する場合は、[CREATE VIEW](#) で説明されているように、必要条件や制限事項に従います。

UPDATE コマンドにより、1 つ以上の新しい列の値が、それらの列を含む既存のベース・テーブルの 1 つ以上の行に入ります。データ値の列への割り当ては、value-assignment-statement を使用して行われます。既定では、value-assignment-statement はテーブルのすべての行で更新されます。

UPDATE を使用する場合、一般的には condition-expression に基づいて更新する行を指定します。既定では、UPDATE 処理はテーブルのすべての行をチェックし、condition-expression を満たすすべての行を更新します。condition-expression を満たす行がない場合、UPDATE は正常に終了して SQLCODE=100 (データがこれ以上ありません) を設定します。

WHERE 節または WHERE CURRENT OF 節を指定することができます (両方は不可)。WHERE CURRENT OF 節が使用される場合、UPDATE は現在のカーソル位置のレコードを更新します。指定位置での実行の詳細は、“[WHERE CURRENT OF](#)” を参照してください。

また、UPDATE 操作により、[%ROWCOUNT](#) ローカル変数に更新された行数が設定され、[%ROWID](#) ローカル変数に最後に更新された行の RowID 値が設定されます。

既定では、UPDATE オペレーションは全か無かのイベントです。指定された行および列をすべて更新するか、まったくしないかのいずれかです。

列に値を割り当てるさまざまな方法の詳細は、以下の “[値の割り当て](#)” のセクションを参照してください。

### INSERT OR UPDATE

[INSERT OR UPDATE](#) 文は、INSERT 文のバリエーションであり、挿入操作と更新操作の両方を実行します。この文は最初に、挿入操作を実行しようとします。UNIQUE KEY 違反が原因で挿入要求に失敗した場合 (いずれかの一意キーのフィールドについて、この挿入対象として指定された行と同じ値を持つ既存の行が存在する場合)、挿入要求はその行に対する更新要求に自動的に変化し、INSERT OR UPDATE が指定されたフィールド値を使用して既存の行を更新します。

## SQLCODE エラー

既定では、複数行の UPDATE はアトミック処理です。1 行または複数行の更新できない行があると、UPDATE 処理は失敗し、どの行も更新されません。InterSystems IRIS は、UPDATE の成功または失敗を示す変数 SQLCODE を設定します。操作が失敗した場合は、%msg も設定します。

テーブルを更新するには、以下に示すように、更新がすべてのテーブル、列名、および値の各要件を満たしている必要があります。

テーブル：

- ・ テーブルは、現在の (または指定された) ネームスペースに存在する必要があります。指定されたテーブルが見つからない場合、InterSystems IRIS は SQLCODE -30 エラーを発行します。
- ・ テーブルを READONLY として定義することはできません。読み取り専用テーブルを参照する UPDATE をコンパイルしようとすると、SQLCODE -115 エラーが返されます。このエラーは実行時に発生するのではなく、コンパイル時に発生することに注意してください。["永続クラスのその他のオプション"](#) の READONLY オブジェクトの説明を参照してください。
- ・ 他のプロセスはテーブルを IN EXCLUSIVE MODE でロックできません。ロックされているテーブルを更新しようとすると、SQLCODE -110 エラーになり、%msg は `"RowID = '10' 'Sample.Person' UPDATE"` になります。SQLCODE -110 エラーは、UPDATE 文が更新する最初のレコードを検出した場合にのみ発生し、タイムアウト期間内はロックできません。
- ・ UPDATE で、存在しないフィールドを指定すると、SQLCODE -29 が発行されます。指定したテーブルに定義されているフィールド名をすべてリストするには、["列の名前と番号"](#) を参照してください。フィールドは存在するが UPDATE コマンドの WHERE 節の条件を満たすフィールド値がない場合は、影響を受ける行はないため、SQLCODE 100 (データの末尾) が発行されます。
- ・ まれに、%NOLOCK が指定された UPDATE で更新する行が見つかったが別のプロセスによりその行が即座に削除された場合には、SQLCODE -109 エラー：`"UPDATE"` が返されます。このエラーの %msg は、テーブル名と RowID をリストします。
- ・ ビュー経由でテーブルを更新する場合、ビューを WITH READ ONLY として定義することはできません。これを実行しようとすると、SQLCODE -35 エラーが返されます。ビューが[シャード・テーブル](#)に基づく場合、WITH CHECK OPTION で定義されたビューを介して UPDATE を実行することはできません。これを試みると、SQLCODE -35 が生じ、%msg が `"(sample.myview) with check option INSERT/UPDATE/DELETE"` に設定されます。詳細は、["CREATE VIEW"](#) コマンドを参照してください。

列名と値：

- ・ 更新に重複するフィールド名を含めることはできません。同じ名前の 2 つのフィールドを指定する更新を実行しようとすると、SQLCODE -377 エラーが返されます。
- ・ 別の同時プロセスによってロックされているフィールドは更新できません。これを実行しようとすると、SQLCODE -110 エラーが返されます。大量の更新を実行したことによって <LOCKTABLEFULL> エラーが発生した場合にも、この SQLCODE エラーが発生することがあります。
- ・ 整数カウンタ・フィールドは更新できません。これらのフィールドは変更不可です。変更すると、以下のエラーが生成されます：[RowID](#) フィールド (SQLCODE -107)、[IDENTITY](#) フィールド (SQLCODE -107)、[SERIAL](#) ([%Library.Counter](#)) フィールド (SQLCODE -105)、[ROWVERSION](#) フィールド (SQLCODE -138)。これらのフィールドの値はシステムで生成され、ユーザは変更できません。ユーザはカウンタ・フィールドの初期値を挿入できても、値を更新することはできません。

1 つの例外は、既存のデータを含むテーブルに [SERIAL](#) ([%Library.Counter](#)) フィールドを追加する場合です。この追加されたカウンタ・フィールドの既存のレコードには、NULL が含まれます。この場合、UPDATE を使用して NULL を整数値に変更できます。詳細は、["ALTER TABLE"](#) コマンドを参照してください。

- ・ **シャード・キー・フィールド**は更新できません。シャード・キーの一部であるフィールドを更新しようとすると、SQLCODE -154 エラーが生成されます。
- ・ 更新内容がフィールドの一意制約に違反する場合は、フィールド値を更新できません。一意制約または主キー制約に違反するようなフィールド (またはフィールドのグループ) の値の更新を試みると、SQLCODE -120 エラーが発生します。このエラーは、フィールドに **UNIQUE データ制約**がある場合、またはフィールドのグループに複数フィールドでの一意制約が適用されている場合に返されます。SQLCODE -120 %msg 文字列には、一意制約に違反するフィールドおよび値が含まれています。例えば、<Table 'Sample.MyTable', Constraint 'MYTABLE\_UNIQUE3', Field(s) FullName="Molly Bloom"; failed unique check> や <Table 'Sample.MyTable', Constraint 'MYTABLE\_PKEY2', Field(s) FullName="Molly Bloom"; failed unique check> などです。テーブルの一意の値や主キー・フィールドの制約、および制約の名前のリストの詳細は、[\[カタログの詳細\]](#)の[\[制約\]](#)を参照してください。
- ・ 更新で **VALUELIST パラメータ**にリストされていない値を指定する場合、フィールド値は更新できません。VALUELIST パラメータで定義された永続クラスのプロパティは、有効な値として、VALUELIST でリストされたいずれかのみを受け入れることができ、それ以外の場合は値なし (NULL) が指定されます。VALUELIST の有効な値では、大文字と小文字が区別されます。VALUELIST の値と一致しないデータ値で更新を試みると、SQLCODE -105 のフィールド値検証失敗のエラーが発生します。
- ・ 数値は**キャノニック形式**で挿入されますが、先頭と末尾の 0 や、先頭の複数の符号を付けて指定できます。ただし、SQL では、2 つの連続したマイナス記号は、1 行コメント文字として解析されます。したがって、先頭に 2 つの連続したマイナス符号を付けて数値を指定しようとすると、SQLCODE -12 エラーになります。
- ・ WHERE CURRENT OF 節を使用する場合は、現在のフィールド値を使用して新しい値を生成してフィールドを更新することはできません。例としては、SET Salary=Salary+100 や SET Name=UPPER(Name) が挙げられます。これを実行しようとすると、SQLCODE -69 エラーが返されます。SET <field> = <value expression> を WHERE CURRENT OF <cursor> で使用できません。
- ・ 指定された行の 1 行の更新が外部キーの参照整合性に違反する場合 (かつ %NOCHECK が指定されていない場合)、UPDATE はいずれの行も更新せず、SQLCODE -124 エラーが発行されます。これは、NOCHECK キーワードを使用して外部キーを定義している場合には適用されません。
- ・ 非ストリーム・フィールドをストリーム・データで更新することはできません。これにより、以下に示すように SQLCODE -303 エラーが発生します。
- ・ 挿入されるデータ値は表示モードから論理モードへの変換に合格する必要があります。InterSystems SQL では、データを logical モード形式で格納します。一部のデータ型では、論理形式が表示形式と異なることがあります。例えば、**日付**データは日数を示す整数値として格納され、**時刻**データは午前 0 時 00 分からの秒数として格納され、%List データはエンコードされた文字列として格納されます。文字列や数値など、その他のデータ型の場合、変換は必要ありません。論理格納値に変換できない形式で値を挿入しようとすると、エラーが発生します (日付の場合は SQLCODE -146、時刻の場合は SQLCODE -147)。モード変換の詳細は、“[データ表示オプション](#)”を参照してください。

## 引数

### %keyword

%NOCHECK、%NOFPLAN、%NOINDEX、%NOJOURN、%NOLOCK、%NOTRIGGER、%PROFILE、%PROFILE\_ALL の **キーワード・オプション**のうちの 1 つ、またはこれらのキーワードの空白で区切られたリストを指定する引数 (オプション)。

### table-ref

データを更新する既存の**テーブル**の名前。テーブルで更新を実行する**ビュー**を指定できます。この引数でテーブル値関数や JOIN 構文を指定することはできません。

テーブル名 (またはビュー名) は修飾 (schema.table)、未修飾 (table) のどちらでもかまいません。未修飾の名前は、**スキーマ検索パス** (指定されている場合) または**既定のスキーマ名**を使用して、そのスキーマと照合されます。

## AS t-alias

table-ref (テーブルまたはビュー) の名前のエイリアス (オプション)。エイリアスは有効な識別子である必要があります。AS キーワードはオプションです。

## FROM select-table

更新する行を特定する際に使用されるテーブルを指定するために使用される FROM 節 (オプション)。

複数のテーブルは、コンマ区切りのリストとして指定するか、ANSI 結合キーワードで関連付けることができます。テーブルあるいはビューのあらゆる組み合わせを指定できます。ここで 2 つの select-table の間にコンマを指定する場合、InterSystems IRIS は複数のテーブルに CROSS JOIN を実行して、JOIN 処理の結果テーブルからデータを取得します。ここで 2 つの select-table の間に ANSI 結合キーワードを指定する場合、InterSystems IRIS は指定された結合処理を実行します。詳細は、このドキュメントの “JOIN” のページを参照してください。

オプションで、クエリ実行を最適化するために、1 つ以上の optimize-option キーワードを指定できます。使用可能なオプションは、%ALLINDEX、%FIRSTTABLE select-table、%FULL、%INORDER、%IGNOREINDICES、%NOFLATTEN、%NOMERGE、%NOSVSO、%NOTOPOPT、%NOUNIONOROPT、%PARALLEL、および %STARTTABLE です。詳細は、“FROM” 節を参照してください。

## WHERE condition-expression

更新する行を決定する 1 つまたは複数のブーリアン述語を指定する引数 (オプション)。WHERE 節 (または WHERE CURRENT OF 節) が指定されていない場合、UPDATE はテーブル内のすべての行を更新します。詳細は、“WHERE” 節を参照してください。

## WHERE CURRENT OF cursor

UPDATE 操作が、現在のカーソル位置のレコードを更新することを指定する引数 (オプション)。WHERE CURRENT OF 節または WHERE 節を指定することができます (両方は不可)。詳細は、“WHERE CURRENT OF” を参照してください。

## column

既存の列名を指定する引数 (オプション)。複数の列名はコンマ区切りのリストで指定します。省略した場合は、すべての列が更新されます。

## scalar-expression

スカラ式で表す列データの値。複数のデータ値はコンマ区切りリストで指定します。各データ値は列の並びに対応します。

## :array()

埋め込み SQL のみ – ホスト変数として指定する値の配列。配列の最下位の添え字は指定しないでください。:myupdates()、:myupdates(5,)、および :myupdates(1,1,) はすべて、有効な指定になります。

## 値の割り当て

さまざまな方法で新しい値を、指定した列に割り当てることができます。

- SET キーワードを使用して、1 つ以上の column = scalar-expression のペアをコンマ区切りリストとして指定します。以下に例を示します。

```
SET StatusDate='05/12/06',Status='Purged'
```

- VALUES キーワードを使用して、列のリストとそれに対応するスカラ式のリストを指定します。以下に例を示します。

```
(StatusDate,Status) VALUES ('05/12/06','Purged')
```

スカラ式の値を列のリストに割り当てる場合、指定する列ごとにスカラ式が必要です。



- ・ 列リストを指定せずに VALUES キーワードを使用する場合、行の列の列順と暗黙的に対応するスカラ式のリストを指定します。以下の例では、テーブル内のすべての列を指定し、Address 列を更新するためのリテラル値を指定しています。

```
VALUES (Name,DOB,'22 Main St. Anytown MA 12345',SSN)
```

値を暗黙の列リストに割り当てるときには、DDL で列が定義されている順序で、すべての更新可能フィールドに値を指定する必要があります (更新できない RowID 列は指定されません)。これらの値は、新規の値を指定するリテラルまたは既存の値を指定するフィールド名のいずれかにすることができます。プレースホルダとしてのコンマを指定したり、末尾のフィールドを省略したりすることはできません。

- ・ 列リストなしで VALUES キーワードを使用すると、数値添え字が列番号と対応している、添え字付き配列が指定されます。列数には更新できない RowID が列番号 1 として含まれます。例えば以下ようになります。

```
VALUES :myarray()
```

この値の割り当ては、[埋め込み SQL](#) からホスト変数を使用することでのみ実行できます。その他すべての値の割り当てと異なり、この使用方法では (実行時に配列を追加することで) 更新する列の指定を実行時まで遅らせることができます。その他すべてのタイプの更新では、コンパイル時に更新する列を指定する必要があります。この構文はリンク・テーブルと一緒に使用できません。併用しようとすると、SQLCODE = -155 エラーが返されます。詳細は、[添え字付き配列としてのホスト変数](#) を参照してください。

これらの UPDATE の各タイプを示すプログラム例については、後述の [“例”](#) のセクションを参照してください。

## リスト構造

InterSystems IRIS は、リスト構造のデータ型 %List (データ型クラス %Library.List) をサポートしています。これは圧縮バイナリ形式であり、InterSystems SQL で対応するネイティブなデータ型にマップしません。また、データ型 VARBINARY に対応しており、その MAXLEN の既定値は 32749 です。このため、[ダイナミック SQL](#) では、型 %List のプロパティ値を設定するときに、UPDATE も INSERT も使用できません。詳細は、[“データ型”](#) を参照してください。

## ストリーム値

以下のように、ストリーム・フィールド内のデータ値を更新できます。

- ・ 任意のテーブル：文字列リテラル、または文字列リテラルが含まれるホスト変数。以下に例を示します。

### ObjectScript

```
SET literal="update stream string value"
//do the update; use a string
&sql(UPDATE MyStreamTable SET MyStreamField = :literal WHERE %ID=21)
```

- ・ シャード化されていないテーブル：ストリーム・オブジェクトへのオブジェクト参照 (OREF)。InterSystems IRIS は、このオブジェクトを開いてそのコンテンツをコピーし、ストリーム・フィールドを更新します。以下に例を示します。

### ObjectScript

```
SET oref=##class(%Stream.GlobalCharacter).%New()
DO oref.Write("Update stream string value non-shard 1")
//do the update; use an actual OREF
&sql(UPDATE MyStreamTable SET MyStreamField = :oref WHERE %ID=22)
```

または、ストリームの文字列版 OREF。以下はその例です。

## ObjectScript

```
SET oref=##class(%Stream.GlobalCharacter).%New()
DO oref.Write("Update stream string value non-shard 2")
//next line converts OREF to a string OREF
set string=oref_" "
//do the update
&sql(UPDATE MyStreamTable SET MyStreamField = :string WHERE %ID=23)
```

- ・ **シャード・テーブル** : ^IRIS.Stream.Shard グローバルに保存されている一時ストリーム・オブジェクトを使用するオブジェクト ID (OID)。

## ObjectScript

```
SET clob=##class(%Stream.GlobalCharacter).%New("Shard")
DO clob.Write("Update sharded table stream string value")
SET sc=clob.%Save() // Handle $$$ISERR(sc)
set ClobOid=clob.%Oid()
//do the update
&sql(UPDATE MyStreamTable SET MyStreamField = :ClobOid WHERE %ID=24)
```

非ストリーム・フィールドをストリーム・フィールドのコンテンツで更新することはできません。これを実行すると、SQLCODE -303 エラー：“UPDATE 割り当てでストリーム・フィールドから非ストリーム・フィールド への暗黙の変換はありません” が返されて失敗します。文字列フィールドをストリーム・データで更新するには、以下の例のように、最初に SUBSTRING 関数を使用してストリーム・データの最初の n 文字を文字列に変換する必要があります。

## SQL

```
UPDATE MyTable
SET MyStringField=SUBSTRING(MyStreamField,1,2000)
```

## 計算フィールド

COMPUTECODE を指定して定義されているフィールドは、以下のように、UPDATE 処理の一部として値を再計算できます。

- ・ COMPUTECODE: 値は、計算されて INSERT 時に保存されます。UPDATE 時には変更されません。
- ・ COMPUTEONCHANGE の COMPUTECODE: 値は、計算されて INSERT 時に保存されます。UPDATE 時には再計算されて保存されます。
- ・ DEFAULT と COMPUTEONCHANGE の COMPUTECODE: 既定値は、INSERT 時に保存されます。値は、UPDATE 時に計算されて保存されます。計算コードにプログラミング・エラーがある場合 (例えば、ゼロでの除算)、UPDATE 操作は SQLCODE -415 エラーで失敗します。
- ・ CALCULATED または TRANSIENT が指定された COMPUTECODE: このフィールドには値が格納されないで、値の UPDATE は実行できません。値は照会されるときに計算されます。ただし、計算済みフィールド内の値を更新しようとする、InterSystems IRIS は指定された値に対する検証を実行し、値が無効であればエラーを発行します。値が有効である場合、InterSystems IRIS は更新操作を実行せず、SQLCODE エラーを発行せず、ROWCOUNT をインクリメントします。

UPDATE 処理の新しいフィールド値が以前のフィールド値と同じである場合、実際の更新が発生しなければ、COMPUTEONCHANGE 計算フィールドは再計算されません。

ほとんどの場合、計算フィールドは読み取り専用として定義します。これによって、他のフィールド値が関与する計算の結果とされる値が、更新処理によって直接変更されないようになります。この場合、計算フィールドの値を上書きする UPDATE を使用しようとする、SQLCODE -138 エラーが発生します。

ただし、1 つ (以上) のソース・フィールド値の更新を反映できるように、計算フィールド値を改訂することもできます。このような改訂は、指定されたソース・フィールドを更新した後に計算フィールド値を再計算する更新トリガを使用することで実行できます。例えば、Salary データ・フィールドの更新によって、Bonus 計算フィールドを再計算するトリガを起動した



とします。この更新トリガによって Bonus が再計算され、Bonus が読み取り専用フィールドであっても正常に完了します。“[CREATE TRIGGER](#)” 文を参照してください。

CREATE TABLE [ON UPDATE](#) キーワード句を使用して、レコードが更新されるときにリテラルまたはシステム変数に設定されるフィールド (現在のタイムスタンプなど) を定義できます。

詳細は、“[INSERT または UPDATE 時の計算フィールドの値](#)” を参照してください。

## %SerialObject プロパティ

[%SerialObject](#) でデータを更新する際は、埋め込み %SerialObject を参照するテーブル (永続クラス) を更新する必要があります。%SerialObject を直接更新することはできません。参照するテーブルから、以下のいずれかを実行できます。

- 参照するフィールドを使用して、複数の %SerialObject プロパティの値を %List 構造として更新する。例えば、永続クラスに、プロパティ Street、City、および Country をこの順序で含むシリアル・オブジェクトを参照するプロパティ PAddress がある場合、SET PAddress=\$LISTBUILD('123 Main St.', 'Newtown', 'USA'), (PAddress) VALUES (\$LISTBUILD('123 Main St.', 'Newtown', 'USA')), または (PAddress) VALUES (:vallist) を更新します。%List には、シリアル・オブジェクト (またはプレースホルダとしてのコンマ) のプロパティの値が、このシリアル・オブジェクトで指定されている順序で含まれる必要があります。

このタイプの更新では、%SerialObject プロパティ値の検証は実行されない場合があります。したがって、%List 構造を使用して %SerialObject プロパティ値を更新した後に、\$SYSTEM.SQL.Schema.ValidateTable() メソッドを使用して [テーブル・データの検証](#) を実行することを強くお勧めします。

- アンダースコア構文を使用して、個々の %SerialObject プロパティの値を任意の順序で更新する。例えば、永続クラスに、プロパティ Street、City、および Country を含むシリアル・オブジェクトを参照するプロパティ PAddress がある場合、SET PAddress\_City='Newtown', PAddress\_Street='123 Main St.', PAddress\_Country='USA' を更新します。

このタイプの更新では、%SerialObject プロパティ値の検証が実行されます。

## FROM 節

UPDATE コマンドでは FROM キーワードを省略できます。更新するテーブル (またはビュー) を指定し、更新する行を WHERE 節を使用して選択するだけでもかまいません。

ただし、value-assignment-statement の後に [FROM](#) 節をオプションで含めることもできます。この FROM 節では、更新するレコードを特定するために使用される 1 つ以上のテーブルを指定します。この FROM 節は、通常は (常にではありませんが) 複数のテーブルを含む WHERE 節と共に使用します。FROM 節は複雑にすることができ、ANSI [結合構文](#) を含めることができます。SELECT FROM 節でサポートされている構文はすべて、UPDATE FROM 節で許可されています。この UPDATE FROM 節は、Transact-SQL との機能的な互換性を提供します。

以下の例は、この FROM 節の使用法を示しています。ここでは、Retirees テーブル内に同じ EmpId がある場合、それらのレコードが Employees テーブルで更新されます。

## SQL

```
UPDATE Employees AS Emp
SET retired='Yes'
FROM Retirees AS Rt
WHERE Emp.EmpId = Rt.EmpId
```

UPDATE table-ref と FROM 節で同じテーブルを参照する場合には、参照するテーブルが文字どおり同じであることも、テーブルの 2 つのインスタンスの結合であることもあります。これは、テーブルのエイリアスの使用方法によって異なります。

- 両方のテーブル参照にエイリアスがない場合には、両方とも同じテーブルを参照します。

```
UPDATE table1 value-assignment FROM table1,table2 /* join of 2 tables */
```

- 両方のテーブル参照に同じエイリアスがある場合には、両方とも同じテーブルを参照します。

```
UPDATE table1 AS x value-assignment FROM table1 AS x,table2 /* join of 2 tables */
```

- 両方のテーブル参照にエイリアスがあり、それぞれのエイリアスが異なる場合には、InterSystems IRIS は 2 つのテーブルのインスタンスの結合を実行します。

```
UPDATE table1 AS x value-assignment FROM table1 AS y,table2 /* join of 3 tables */
```

- 最初のテーブル参照にエイリアスがあり、2 番目にはない場合には、InterSystems IRIS は 2 つのテーブルのインスタンスの結合を実行します。

```
UPDATE table1 AS x value-assignment FROM table1,table2 /* join of 3 tables */
```

- 最初のテーブル参照にエイリアスがなく、2 番目にはエイリアスのあるテーブルへの単一の参照がある場合には、両方とも同じテーブルを参照し、このテーブルには指定されたエイリアスがあります。

```
UPDATE table1 value-assignment FROM table1 AS x,table2 /* join of 2 tables */
```

- 最初のテーブル参照にエイリアスがなく、2 番目にはテーブルへの複数の参照がある場合には、それぞれのエイリアスされたインスタンスは個別のテーブルと見なされ、それらのテーブルの結合が実行されます。

```
UPDATE table1 value-assignment FROM table1,table1 AS x,table2 /* join of 3 tables */
UPDATE table1 value-assignment FROM table1 AS x,table1 AS y,table2 /* join of 4 tables */
```

## %Keyword 引数

%keyword 引数を指定すると、以下のように処理を制限します。

- %NOCHECK — 一意の値のチェックおよび外部キーの参照整合性チェックは実行されません。データ型、最大長、データ制約などの検証条件に関して、列データの検証も実行されません。ビューを介して UPDATE を実行する場合、ビューの WITH CHECK OPTION 検証は実行されません。

注釈 %NOCHECK を使用すると、無効なデータが発生する可能性があるため、この %keyword 引数の使用は、信頼性の高いデータ・ソースから挿入または更新を一括で実行する場合に限定してください。

この制限を適用するには、ユーザは、現在のネームスペースに対して対応する %NOCHECK [管理特権](#)を持っている必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が " 'name' %NOCHECK " に設定されます。

%NOCHECK を指定する際に、更新で一意でないデータ値が生じないようにしたい場合は、UPDATE の前に [EXISTS](#) チェックを実行します。

外部キーの参照整合性チェックのみを無効にしたい場合は、%NOCHECK を指定するのではなく \$SYS-TEM.SQL.Util.SetOption("FilerRefIntegrity") メソッドを使用します。または、NOCHECK キーワードを使用して外部キーを定義し、外部キーの参照整合性チェックが実行されないようにすることができます。

- %NOPLAN — FROM 節の構文のみ: この操作の凍結されたプラン (ある場合) を無視して、新しいクエリ・プランを生成します。凍結されたプランは保持されますが、使用されません。詳細は、"[凍結プランの構成](#)" を参照してください。
- %NOINDEX — インデックス・マップは UPDATE 処理中には設定されません。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する %NOINDEX [管理特権](#)を持っている必要があります。これを実行しないと、SQLCODE -99 エラーが返されます。
- %NOJOURN — 更新操作の間、ジャーナリングを抑制し、トランザクションを無効化します。プルされたトリガを含め、行での変更はどれもジャーナリングされません。ただし、更新は依然として [ミラーリングされている環境](#) でジャーナリングされます。%NOJOURN が指定された文の後で ROLLBACK を実行した場合、その文で行われた変更はロール

バックされません。この制限を適用するには、現在のネームスペースに対応する %NOJOURN [管理特権](#) がユーザに必要です。これを実行しないと、SQLCODE -99 エラーが返されます。

- ・ %NOLOCK – UPDATE 時に行をロックしません。単独のユーザ/処理がデータベースを更新する際にのみ使用します。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する %NOLOCK [管理特権](#) を持っている必要があります。これを実行しないと、SQLCODE -99 エラーが返されます。
- ・ %NOTRIGGER – ベース・テーブル・トリガは UPDATE 処理中にはかかりません。BEFORE トリガおよび AFTER トリガのどちらも実行されません。この制限を適用するには、ユーザは、現在のネームスペースに対して対応する %NOTRIGGER [管理特権](#) を持っている必要があります。これを実行しないと、SQLCODE -99 エラーが返されます。
- ・ %PROFILE または %PROFILE\_ALL – これらのキーワード指示文のいずれかが指定された場合、SQLStats 収集コードが生成されます。これは、PTools を ON にして生成されるものと同じコードです。違いは、SQLStats 収集コードはこの特定の文に対してのみ生成されるという点です。コンパイルされるルーチン/クラス内のその他すべての SQL 文は、PTools が OFF であるかのようにコードを生成します。これにより、ユーザは、調査されない SQL 文の関係のない統計を収集することなく、アプリケーション内の特定の問題の SQL 文をプロファイリング/調査できます。詳細は、“[SQL 実行時統計](#)” を参照してください。

%PROFILE はメイン・クエリ・モジュールに対して SQLStats を収集します。%PROFILE\_ALL はメイン・クエリ・モジュールとそのすべてのサブクエリ・モジュールに対して SQLStats を収集します。

複数の %keyword 引数を順不同で指定できます。複数の引数は、空白で区切られます。

## 参照整合性

%NOCHECK を指定しない場合、InterSystems IRIS では、システム全体の構成設定を使用して外部キーの参照整合性チェックを実行するかどうかが決まります。既定では、外部キーの参照整合性チェックを実行します。“[外部キーの参照整合性チェック](#)” の説明に従って、この既定値をシステム全体で設定できます。現在のシステム全体の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。

この設定は、NOCHECK キーワードを使用して定義した外部キーには適用されません。

UPDATE 処理中は、更新するフィールド値を持つ外部キーの参照があるたびに、参照するテーブルの古い(更新前の)参照行と新しい(更新後の)参照行の両方で共有ロックを取得します。これらの行は参照整合性チェックと行の更新が完了するまでロックされます。その後、ロックは解除されます(トランザクションの終了までロック状態が継続することはありません)。このロックによって、参照整合性チェックから更新操作の完了までの間、参照先の行が変更されないようになります。古い行をロックすることにより、参照される行は、UPDATE のロールバックがあってもそれより前に変更されることがなくなります。新しい行をロックすることにより、参照整合性チェックから更新操作の完了までの間、参照先の行が変更されないようになります。

CASCADE、SET NULL、または SET DEFAULT で定義された外部キー・フィールドに対して、%NOLOCK を指定して UPDATE 操作を実行した場合は、対応する参照アクションが %NOLOCK によって実行されて外部キー・テーブルが変更されます。

## アトミック性

既定では、UPDATE、INSERT、DELETE、および TRUNCATE TABLE はアトミック処理として実行されます。UPDATE は、正常に完了するか、すべての操作がロールバックされるかのいずれかです。指定した行のいずれかを更新できない場合、指定した行は 1 行も更新できずにデータベースは UPDATE を発行する前の状態に戻ります。

現在のプロセスに対するこの既定は、[SET TRANSACTION %COMMITMODE](#) を呼び出すことによって SQL 内で変更できます。現在のプロセスに対するこの既定は、SetOption() メソッドを SET status=\$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval) のように呼び出すことによって ObjectScript 内で変更できます。以下の intval 整数オプションを使用できます。

- ・ 1 または IMPLICIT (自動コミットがオン) – 上記のように、これが既定の動作です。UPDATE ごとに個別のトランザクションが構成されます。

- ・ 2 または EXPLICIT (自動コミットがオフ) – 進行中のトランザクションがない場合は、UPDATE コマンドによってトランザクションは自動的に開始されます。ただし、COMMIT または ROLLBACK で明示的にトランザクションを終了する必要があります。EXPLICIT モードでは、トランザクションあたりのデータベース操作の数は、ユーザ定義です。
- ・ 0 または NONE (自動トランザクションなし) – UPDATE を呼び出してもトランザクションは開始されません。UPDATE 操作の失敗により、行の一部が更新されたり更新されなかったりすることで、データベースが整合性のない状態になる可能性があります。このモードでトランザクションのサポートを提供するには、START TRANSACTION を使用してトランザクションを開始し、COMMIT または ROLLBACK を使用してトランザクションを終了する必要があります。

シャード・テーブルは常に、自動トランザクションなしのモードに設定されます。つまり、シャード・テーブルに対する挿入、更新、および削除はすべて、トランザクションの範囲外で実行されます。

現在のプロセスのアトミック性設定を確認するには、以下の ObjectScript の例のように、GetOption("AutoCommit") メソッドを使用します。

### ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }
```

## トランザクションでのロック

%NOLOCK を指定しない場合、INSERT、UPDATE、および DELETE 操作時に自動的にレコードに標準のロックがかかります。影響を受ける各レコード (行) は、現在のトランザクションが継続している間はロックされます。

既定のロックしきい値は、テーブルごとに 1000 ロックです。つまり、トランザクションの間にテーブルで 1000 件を超えるレコードを更新すると、ロックのしきい値に到達し、InterSystems IRIS は自動的にロック・レベルをレコード・ロックからテーブル・ロックに上げます。これによってトランザクション時に、ロック・テーブルをオーバーフローすることなく、大規模な更新を実行できます。

InterSystems IRIS は、以下の 2 つのロック・エスカレーション策のどちらかを適用します。

- ・ “E” タイプのロック・エスカレーション: InterSystems IRIS は、次のことに該当する場合にこのタイプのロック・エスカレーションを使用します。(1) クラスで [%Storage.Persistent](#) が使用されている (これは、管理ポータル の SQL スキーマ表示で [\[カタログの詳細\]](#) から判断できます)。(2) クラスで、IDKey インデックスが指定されていないか、単一プロパティの IDKey インデックスが指定されている。“E” タイプのロック・エスカレーションについては、“ObjectScript リファレンス” の [LOCK](#) コマンドで説明されています。
- ・ 従来の SQL ロック・エスカレーション: クラスで “E” タイプのロック・エスカレーションが使用されない理由は、マルチプロパティの IDKey インデックスの存在にあると考えられます。この場合は、%Save ごとにロック・カウンタがインクリメントされます。つまり、トランザクション内の単一オブジェクトを 1001 回保存を行うと、InterSystems IRIS はロックのエスカレーションを試みます。

どちらのロック・エスカレーション策の場合も、\$SYSTEM.SQL.Util.GetOption("LockThreshold") メソッドを使用して、現在のシステム全体用ロックしきい値を決定できます。既定値は 1,000 です。このシステム全体のロックしきい値は、以下の方法を使用して設定できます。

- ・ \$SYSTEM.SQL.Util.SetOption("LockThreshold") メソッドを使用します。
- ・ 管理ポータルを使用する。[\[システム管理\]](#)、[\[構成\]](#)、[\[SQL およびオブジェクトの設定\]](#)、[\[SQL\]](#) の順に移動します。[\[ロック・エスカレーションしきい値\]](#) の現在の設定を表示して編集します。既定は 1000 ロックです。この設定を変更すると、変更後に開始される新しいプロセスは、新しい設定になります。



ロックしきい値を変更するには、%Admin Manage Resource の USE 許可が必要です。InterSystems IRIS は、ロックしきい値の変更を現在のプロセスすべてに即座に適用します。

結果として、自動ロック・エスカレーションでは、デッドロックの状況が起こる可能性があります。つまり、テーブル・ロックへのエスカレーションを試みたときに、テーブル内のレコード・ロックを保持する別プロセスとの競合が起こる可能性があります。これを避けるための方策としては、次のいくつかが考えられます。(1) ロック・エスカレーションがトランザクション内で起こる可能性が低くなるように、ロック・エスカレーションのしきい値を上げる。(2) ロック・エスカレーションが即座に起こるように、ロック・エスカレーションのしきい値を大幅に下げる。これにより、別プロセスが同一テーブル内のレコードをロックする機会が少なくなります。(3) トランザクションが継続している間はテーブル・ロックを適用し、レコード・ロックは実行しない。これは、LOCK TABLE、UNLOCK TABLE (テーブル・ロックがトランザクションの終了まで持続するよう、IMMEDIATE キーワードはなし) の順に指定することで、トランザクション開始時に実行できます。その後、%NOLOCK オプションを使用して更新を実行します。

自動ロック・エスカレーションは、ロック・テーブルのオーバーフローを防ぐことを目的としています。ただし、大量の更新などを実行したために <LOCKTABLEFULL> エラーが発生した場合は、UPDATE によって SQLCODE -110 エラーが発行されます。

トランザクションでのロックの詳細は、“[トランザクション処理](#)”を参照してください。

## カウンタのインクリメント

### ROWVERSION カウンタのインクリメント

テーブルにデータ型 [ROWVERSION](#) のフィールドが存在する場合、行の更新を実行すると、このフィールドの整数値が自動的に更新されます。ROWVERSION フィールドは、ネームスペース全体の行バージョン・カウンタの連続した次の整数を取得します。ROWVERSION フィールドに更新値を指定すると、SQLCODE -138 エラーが返されます。

### SERIAL (%Counter) カウンタのインクリメント

UPDATE 操作は、[SERIAL \(%Library.Counter\)](#) カウンタ・フィールドの値に影響しません。ただし、[INSERT OR UPDATE](#) を使用して実行される更新では、SERIAL フィールドに対する後続の挿入操作で整数シーケンスのスキップが発生します。詳細は“[INSERT OR UPDATE](#)”を参照してください。

## 特権

更新を実行するには、指定したテーブル (またはビュー) のテーブルレベルの UPDATE 特権、または指定した列の列レベルの UPDATE 特権のいずれかが必要です。行内のすべてのフィールドを更新する場合、列レベルの特権は GRANT コマンドで名付けられたすべてのテーブル列を管理し、テーブルレベルの特権は、特権が割り当てられた後に追加されたものも含み、すべてのテーブル列を管理します。

- ・ ユーザは指定されたテーブルに対する UPDATE 特権、または UPDATE フィールド・リスト内のすべての列に対する列レベルの UPDATE 特権を持っている必要があります。
- ・ ユーザは WHERE 節のフィールドに対する SELECT 特権を持っている必要があります (これらのフィールドを更新するかどうかにかかわらず)。これらのフィールドが UPDATE フィールド・リストに含まれる場合には、これらのフィールドに対する SELECT 特権と UPDATE 特権の両方が必要です。以下の例では、Name フィールドには (少なくとも) 列レベルの SELECT 特権が必要です。

### SQL

```
UPDATE Sample.Employee (Salary) VALUES (1000000) WHERE Name='Smith, John'
```

上記の例では、Salary フィールドには列レベルの UPDATE 特権のみが必要です。

ユーザがテーブルの所有者 (作成者) である場合、ユーザにはそのテーブルに対するすべての特権が自動的に付与されます。そうでない場合は、ユーザにテーブルに対する特権を付与する必要があります。持っていない場合、SQLCODE -99 エラーが発生し、%msg が “ ” 'name' ” に設定されます。[%CHECKPRIV](#) コマンドを呼

び出すことにより、現在のユーザが適切な特権を持っているかどうかを確認できます。[GRANT](#) コマンドを使用してユーザにテーブル特権を割り当てることができます。詳細は、“[特権](#)”を参照してください。

プロパティが [ReadOnly](#) として定義されている場合、対応するテーブル・フィールドも ReadOnly として定義されています。ReadOnly フィールドに値を割り当てるには、[InitialExpression](#) または [SqlComputed](#) を使用する必要があります。列レベルの ReadOnly (SELECT または REFERENCES) 特権を持っているフィールドの値 (NULL 値であっても) を更新しようとする、SQLCODE -138 エラー：“ INSERT/UPDATE ”が発生します。リンク・テーブル・ウィザードを使用してテーブルをリンクすると、フィールドを Read Only として定義することができるようになります。ソース・システム上のフィールドが読み取り専用ではないにもかかわらず、InterSystems IRIS でそのリンク・テーブルのフィールドを Read Only として定義した場合、このフィールドを参照する UPDATE を試みると、SQLCODE -138 エラーが発生します。

## 行レベル・セキュリティ

InterSystems IRIS の行レベル・セキュリティにより、UPDATE を使用して、セキュリティでアクセスが許可されるすべての行を変更できるようになります。セキュリティにより今後のアクセスが許可されない行が UPDATE によって作成された場合でも、行の更新が可能になります。UPDATE により行への今後の SELECT アクセスが妨げられないようにするには、WITH CHECK OPTION を持つビューで UPDATE を実行することをお勧めします。詳細は、“[CREATE VIEW](#)”を参照してください。

## 例

このセクションの例では、SQLUser.MyStudents テーブルを更新しています。以下の例では、SQLUser.MyStudents テーブルを作成し、そのテーブルにデータを生成しています。この例を繰り返し実行すると、重複データのあるレコードが累積するため、TRUNCATE TABLE を使用して、INSERT を呼び出す前に古いデータを削除します。この例は、UPDATE の例を呼び出す前に実行します。

### ObjectScript

```
CreateStudentTable
SET stuDDL=5
SET stuDDL(1)="CREATE TABLE SQLUser.MyStudents ( "
SET stuDDL(2)="StudentName VARCHAR(32),StudentDOB DATE,"
SET stuDDL(3)="StudentAge INTEGER COMPUTECODE {SET {StudentAge}="
SET stuDDL(4)="$PIECE(($PIECE($H,"",",",1)-{StudentDOB}))/365,"",",1)} CALCULATED,"
SET stuDDL(5)="Q1Grade CHAR,Q2Grade CHAR,Q3Grade CHAR,FinalGrade VARCHAR(2))"
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(.stuDDL)
IF qStatus=1 {WRITE "DDL %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rtn = tStatement.%Execute()
IF rtn.%SQLCODE=0 {WRITE !,"Table Create successful"}
ELSEIF rtn.%SQLCODE=-201 {WRITE "Table already exists, SQLCODE=",rtn.%SQLCODE,!}
ELSE {WRITE !,"table create failed, SQLCODE=",rtn.%SQLCODE,!
      WRITE rtn.%Message,! }
RemoveOldData
SET clearit="TRUNCATE TABLE SQLUser.MyStudents"
SET qStatus = tStatement.%Prepare(clearit)
IF qStatus=1 {WRITE "Truncate %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET truncrtn = tStatement.%Execute()
IF truncrtn.%SQLCODE=0 {WRITE !,"Table old data removed",!}
ELSEIF truncrtn.%SQLCODE=100 {WRITE !,"no data to be removed",!}
ELSE {WRITE !,"truncate failed, SQLCODE=",truncrtn.%SQLCODE," ",truncrtn.%Message,! }
PopulateStudentTable
SET studentpop=2
SET studentpop(1)="INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "
SET studentpop(2)="SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET qStatus = tStatement.%Prepare(.studentpop)
IF qStatus=1 {WRITE "Populate %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET poprtn = tStatement.%Execute()
IF poprtn.%SQLCODE=0 {WRITE !,"Table Populate successful",!
  WRITE poprtn.%ROWCOUNT," rows inserted"}
ELSE {WRITE !,"table populate failed, SQLCODE=",poprtn.%SQLCODE,!
      WRITE poprtn.%Message }
```

以下のクエリを使用して、これらの例の結果を表示できます。

## SQL

```
SELECT %ID,* FROM SQLUser.MyStudents ORDER BY StudentAge,%ID
```

以下の UPDATE の例のいくつかは、他の UPDATE の例で設定されるフィールド値に依存しているため、指定された順序で実行する必要があります。

以下の **ダイナミック SQL** の例では、SET field=value UPDATE は、選択したレコード内の指定したフィールドを変更します。MyStudents テーブルでは、7 歳未満の児童には成績が指定されていません。

## ObjectScript

```
SET studentupdate=3
SET studentupdate(1)="UPDATE SQLUser.MyStudents "
SET studentupdate(2)="SET FinalGrade='NA' "
SET studentupdate(3)="WHERE StudentAge <= 6"
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(.studentupdate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message }
```

以下の **カーソル・ベースの埋め込み SQL** の例では、SET field1=value1,field2=value2 UPDATE は、選択したレコード内のいくつかのフィールドを変更します。MyStudents テーブルで、これは指定した学生レコードを Q1 と Q2 の成績で更新します。

## ObjectScript

```
#sqlcompile path=Sample
NEW %ROWCOUNT,%ROWID
&sql(DECLARE StuCursor CURSOR FOR
      SELECT * FROM MyStudents
      WHERE %ID IN(10,12,14,16,18,20,22,24) AND StudentAge > 6)
&sql(OPEN StuCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH StuCursor)
      QUIT:SQLCODE
      &sql(Update MyStudents SET Q1Grade='A',Q2Grade='A'
          WHERE CURRENT OF StuCursor)
      IF SQLCODE=0 {
        WRITE !,"Table Update successful"
        WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
      ELSE {
        WRITE !,"Table Update failed, SQLCODE=",SQLCODE }
      }
&sql(CLOSE StuCursor)
```

以下の **ダイナミック SQL** の例では、field-list VALUES value-list UPDATE は、選択したレコード内のいくつかのフィールドの値を変更します。MyStudents テーブルでは、最終成績を受け取らない児童は、四半期成績も受け取りません。

## ObjectScript

```
SET studentupdate=3
SET studentupdate(1)="UPDATE SQLUser.MyStudents "
SET studentupdate(2)=(Q1Grade,Q2Grade,Q3Grade) VALUES ('x','x','x') "
SET studentupdate(3)="WHERE FinalGrade='NA'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.studentupdate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

以下の **ダイナミック SQL** の例では、VALUES value-list UPDATE は、選択したレコード内のすべてのフィールド値を変更します。この構文では、レコード内のすべてのフィールドの値を指定する必要があります。MyStudents テーブル



では、数人の児童が学校を退学しています。そのレコード ID と名前は保持され、名前に WITHDRAWN が付加されます。その他すべてのデータは削除され、DOB フィールドは退学の日付用に使用されます。

### ObjectScript

```
SET studentupdate=4
SET studentupdate(1)="UPDATE SQLUser.MyStudents "
SET studentupdate(2)="VALUES (StudentName||' WITHDRAWN',"
SET studentupdate(3)="$PIECE($HOROLOG,' ',1),00,'-','-', 'XX') "
SET studentupdate(4)="WHERE %ID IN(7,10,22)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.studentupdate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

以下の**ダイナミック SQL** の例では、**subquery** UPDATE はレコードの選択にサブクエリを使用しています。次にそれらのレコードを SET field=value 構文を使用して変更しています。SQLUser.MyStudents にある誕生日から StudentAge を計算する方法により、1 歳未満の児童の年齢は <Null> と算出されており、誕生日が NULL となっている児童は算出年齢がかなり高くなっています。ここで StudentName フィールドには、後で誕生日を確認するためにフラグが付けられています。

### ObjectScript

```
SET studentupdate=3
SET studentupdate(1)="UPDATE (SELECT StudentName FROM SQLUser.MyStudents "
SET studentupdate(2)="WHERE StudentAge IS NULL OR StudentAge > 21) "
SET studentupdate(3)="SET StudentName = StudentName||' *** CHECK DOB' "
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(.studentupdate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

以下の**埋め込み SQL** の例では、VALUES :array() UPDATE は、選択したレコード内の配列の列番号によって指定されたフィールド値を変更します。VALUES :array() の更新は、埋め込み SQL でのみ実行できます。この構文では、各値を DDL 列番号で指定する必要があります (列数には RowID 列 (列 1) を含めますが、この変更できないフィールドに値は指定しません)。MyStudents テーブルでは、4 歳以上 6 歳以下の児童には、'P' ('Present (在籍)') を示す) がその Q1Grade (列 5) および Q2Grade (列 6) のフィールドに指定されます。他のすべてのレコード・データは、変更されません。

### ObjectScript

```
SET array(5)="P"
SET array(6)="P"
&sql(UPDATE SQLUser.MyStudents VALUES :array()
      WHERE FinalGrade='NA' AND StudentAge > 3)
IF SQLCODE=0 {WRITE "Table Update successful",!
              WRITE "Rows updated=",%ROWCOUNT," Final RowID=",%ROWID }
ELSE {WRITE "Table Update failed, SQLCODE=",SQLCODE,! }
```

## 関連項目

- ・ [INSERT](#)
- ・ [INSERT OR UPDATE](#)
- ・ [DELETE](#)
- ・ [SELECT](#)
- ・ [VALUES](#)
- ・ [FROM](#)

- ・ [WHERE](#)
- ・ [WHERE CURRENT OF](#)
- ・ [CREATE TABLE](#)
- ・ [CREATE VIEW](#)
- ・ [データベースの変更](#)
- ・ [テーブルの定義](#)
- ・ [ビューの定義](#)
- ・ [トランザクション処理](#)
- ・ [SQL およびオブジェクトの設定ページ](#)
- ・ [SQLCODE エラー・メッセージ](#)

# USE DATABASE (SQL)

現在のネームスペースおよびデータベースを設定します。

## 構文

```
USE [DATABASE] dbname
```

## 概要

USE DATABASE コマンドは、現在のプロセスを、指定したネームスペースおよびそれに関連するデータベースに切り替えます。これにより、SQL 内でネームスペースを変更できるようになります。DATABASE キーワードはオプションです。

指定した dbname は、目的のネームスペース、およびデータベース・ファイルを含む対応ディレクトリの名前となります。dbname は識別子として指定します。ネームスペース名は、大文字と小文字を区別しません。ネームスペースの使用に関する詳細は、“[ネームスペースとデータベース](#)”を参照してください。

USER は SQL 予約語であるため、USER ネームスペースを指定するには、以下の SQL シェルの例で示すように区切り識別子を使用する必要があります。

```
USER>>USE DATABASE Samples
SAMPLES>>USE DATABASE "User"
USER>>
```

指定した dbname が存在しない場合、InterSystems IRIS によって SQLCODE -400 エラーが発行されます。

USE DATABASE コマンドは特権を必要とする操作です。USE DATABASE を使用する前に、適切な特権を有するユーザとしてログインする必要があります。特権がない場合は、SQLCODE -99 エラー（特権違反）が返されます。

\$SYSTEM.Security.Login() メソッドを使用して、以下のようにユーザに適切な特権を割り当ててください。

### ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

\$SYSTEM.Security.Login メソッドを呼び出すには、**%Service\_Login:Use** 特権が必要です。詳細は、“[%SYSTEM.Security](#)”を参照してください。

ObjectScript の [ZNSPACE](#) コマンド、または [SET \\$NAMESPACE](#) 文を使用して、異なるネームスペースに切り替えることもできます。

## データベース・ドライバによる実行

USE DATABASE コマンドがデータベース・ドライバにより実行されると、サーバ・プロセスはシミュレートされた接続リセットを実行します。サーバ・プロセスにより使用されるデータ構造は削除されます。ただし、コミット・モードは変更されません。READ COMMITTED 設定も変更されません。トランザクションが進行中の場合、トランザクションは実行を継続するだけであり、コミットやロールバックは実行されません。

## 引数

### dbname

現在のネームスペースとして現在のプロセスで使用するネームスペースおよび対応するデータベース。

## 関連項目

- [CREATE DATABASE](#) コマンド

- ・ [DROP DATABASE コマンド](#)

# VALIDATE MODEL (SQL)

モデルを検証します。

## 構文

```
VALIDATE MODEL model-name [ AS validation-run-name ]
    [ USE trained-model-name ]
    [ WITH feature-column-clause ]
FROM model-source
```

## 引数

model-name	検証するモデルの名前。
AS validation-run-name	オプション - 検証実行を保存する名前。この後の説明を参照してください。
USE trained-model-name	オプション - 検証する既定でないトレーニング済みモデルの名前。この後の説明を参照してください。
WITH feature-column-clause	オプション - モデルの検証に使用するデータセットの特定の列。
FROM model-source	モデルの検証元にするテーブルまたはビュー。テーブル、ビュー、または結合の結果です。この後の説明を参照してください。

## 説明

VALIDATE MODEL コマンドは、指定されたテスト・データセットでのパフォーマンスに基づいて、特定のトレーニングされたモデルの検証メトリックを計算します。各コマンドは検証実行を作成します。

## 名前付け

AS を使用すると、検証実行に明示的に名前を付けることができます。

検証実行に AS を使用して明示的に名前が付けられていない場合、名前はトレーニングされたモデルに実行中の整数が付加されたものになります。INFORMATION\_SCHEMA.ML\_VALIDATION\_RUNS テーブルをクエリして、その違いを確認できます。

```
CREATE MODEL TitanicModel PREDICTING (Survived) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel AS TitanicValidation FROM IntegratedML_dataset_titanic.passenger
SELECT MODEL_NAME, TRAINED_MODEL_NAME, VALIDATION_RUN_NAME FROM INFORMATION_SCHEMA.ML_VALIDATION_RUNS
```

MODEL_NAME	TRAINED_MODEL_NAME	VALIDATION_RUN_NAME
TitanicModel	TitanicModel_t1	TitanicModel_t1_v1
TitanicModel	TitanicModel_t1	TitanicModel_t1_v2
TitanicModel	TitanicModel_t1	TitanicModel_t1_v3
TitanicModel	TitanicModel_t1	TitanicValidation

## USE

USE を使用すると、検証を実行するトレーニングされたモデルを指定できます。トレーニングされたモデルが USE で明示的に指定されていない場合は、既定のトレーニング済みモデルが、指定されたモデル定義に対して検証されます。

INFORMATION\_SCHEMA.ML\_VALIDATION\_RUNS テーブルをクエリして、その違いを確認できます。

```
CREATE MODEL TitanicModel PREDICTING (Survived) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel AS FirstModel
TRAIN MODEL TitanicModel AS SecondModel
TRAIN MODEL TitanicModel AS ThirdModel
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel USE FirstModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel USE SecondModel FROM IntegratedML_dataset_titanic.passenger
SELECT MODEL_NAME, TRAINED_MODEL_NAME FROM INFORMATION_SCHEMA.ML_VALIDATION_RUNS
```

MODEL_NAME	TRAINED_MODEL_NAME
TitanicModel	ThirdModel
TitanicModel	ThirdModel
TitanicModel	FirstModel
TitanicModel	SecondModel

## FROM の考慮事項

モデルのトレーニングにはトレーニング・セットを使用しましたが、モデルの検証には他のデータ (テスト・データ・セット) を使用する必要があります。トレーニング・データを使用してモデルを検証する場合に評価されるのは適合度のみです。これに対し、モデルの予測パフォーマンスは他のデータで評価します。

このデータは、特徴列とラベル列を含め、トレーニング・データと同じスキーマである必要があります。

## 必要なセキュリティ特権

VALIDATE MODEL を呼び出すには、%USE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%USE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## 検証メトリック

VALIDATE MODEL の出力は、INFORMATION\_SCHEMA.ML\_VALIDATION\_METRICS テーブルで表示可能な一連の検証メトリックです。

回帰モデルでは、以下のメトリックが保存されます。

- ・ 分散
- ・ 決定係数
- ・ 平均二乗誤差
- ・ 二乗平均平方根誤差

分類モデルでは、以下のメトリックが保存されます。

- ・ 適合率 - 真陽性の数を予測される陽性数 (真陽性と疑陽性の合計) で割って計算されます。
- ・ 再現率 - 真陽性の数を実際の陽性数 (真陽性と偽陰性の合計) で割って計算されます。
- ・ F 値 - 次の式で計算されます。  

$$F = 2 * ( \text{適合率} * \text{再現率} ) / ( \text{適合率} + \text{再現率} )$$
- ・ 正解率 - 真陽性と真陰性の数を、テスト・セット全体の合計行数 (真陽性、偽陽性、真陰性、偽陰性の合計) で割って計算されます。
- ・ ROC-AUC - これは、受信者操作特性曲線の下を面積を計算した値です。この値が大きいほど、モデルはクラス間の違いをよりの確に認識できます。

## 例

```
VALIDATE MODEL PatientReadmission FROM Patient_test  
VALIDATE MODEL PatientReadmission AS PatientValidation USE PatientReadmission_H2OModel FROM Patient_test
```

## 関連項目

- ・ [CREATE MODEL](#)、[TRAIN MODEL](#)、[PREDICT](#)





# SQL 節

## DISTINCT (SQL)

非重複値のみを返すために指定する SELECT 節です。

### 構文

```
SELECT DISTINCT BY (item {,item2}) select-item {,select-item2}
```

```
SELECT DISTINCT [ALL] select-item {,select-item2}
```

### 引数

引数	説明
DISTINCT	オプション - select-item 値の組み合わせが一意である行を返します。
DISTINCT BY (item {,item2})	オプション - BY (item) の値 (1 つまたは複数) が一意である行の select-item を返します。
ALL	オプション - 結果セットにすべての行を返します。これが既定です。

### 概要

オプションの DISTINCT 節は、SELECT キーワードの後、オプションの [TOP 節](#) および最初の select-item の前に記述します。

DISTINCT 節は、[SELECT](#) 文の結果セットに適用されます。これは、返される行を、個別 (一意) の値ごとに 1 つの任意の行に制限します。DISTINCT 節が指定されない場合、既定では SELECT 条件を満たすすべての行を表示します。ALL 節は、DEFAULT 節を指定しないことと同義です。ALL を指定した場合の SELECT では、SELECT 条件を満たす、テーブル内のすべての行が返されます。

DISTINCT 節には、以下の 2 つの形式があります。

- SELECT DISTINCT: select-item の値の一意な組み合わせごとに 1 行を返します。1 つまたは複数の select-items を指定できます。例えば、以下のクエリでは、Home\_State および Age の値の一意な組み合わせごとに、Home\_State および Age の値を含む行が返されます。

#### SQL

```
SELECT DISTINCT Home_State, Age FROM Sample.Person
```

- SELECT DISTINCT BY (item): item の値の一意な組み合わせごとに 1 行を返します。単一の item、またはコンマ区切りの item のリストを指定できます。指定された item または item のリストは、括弧で囲む必要があります。BY キーワードの括弧の間は、スペースを指定しても省略してもかまいません。select-item リストには、指定した item を含めることができます (必須ではありません)。例えば、以下のクエリでは、Home\_State および Age の値の一意な組み合わせごとに、Name および Age の値を含む行が返されます。

#### SQL

```
SELECT DISTINCT BY (Home_State, Age) Name, Age FROM Sample.Person
```

item フィールドは、列名で指定する必要があります。有効な値としては、列名 (DISTINCT BY (City)) や、%ID (すべての行を返す) や、列名を指定するスカラー関数 (DISTINCT BY (ROUND(Age, -1))) や、列名を指定する [照合関数](#) (DISTINCT BY (%EXACT(City))) があります。列エイリアスでフィールドを指定できません。これを試行すると、SQLCODE -29 エラーが生成されます。列番号でフィールドを指定することはできません。これはリテラルとして解釈され、1 つの行を返します。DISTINCT 節内の item 値としてリテラルを指定すると、1 行が返されます。ど

の行が返されるかは不確定となります。したがって、7、'Chicago'、'、0、または NULL のどれを指定しても、1 行が返されます。ただし、コンマ区切りリスト内の item 値としてリテラルを指定すると、そのリテラルは無視され、DISTINCT が指定されたフィールド名の一意の組み合わせごとに任意の 1 つの行を選択します。

DISTINCT 節は、TOP 節の前に適用されます。両方が指定されている場合は、SELECT では、一意な値を含む行と、TOP 節で指定された一意な値の行数のみが返されます。

DISTINCT 節で指定された列に NULL (値を含まない) 行がある場合、DISTINCT では、以下の例で示すように、個別 (一意の) 値として NULL 値を含む 1 行が返されます。

## SQL

```
SELECT DISTINCT FavoriteColors FROM Sample.Person
```

## SQL

```
SELECT DISTINCT BY (FavoriteColors) Name,FavoriteColors FROM Sample.Person
ORDER BY FavoriteColors
```

[埋め込み SQL](#) の単純なクエリで DISTINCT 節を指定しても効果はありません。このタイプの埋め込み SQL の場合、SELECT では、データは必ず 1 行しか返されないためです。ただし、埋め込み SQL の[カーソルベースのクエリ](#)では、複数のデータ行を返すことができます。カーソルベースのクエリの DISTINCT 節では、一意な値の行のみが返されます。

## DISTINCT および ORDER BY

DISTINCT 節は、[ORDER BY](#) 節の前に適用されます。したがって、DISTINCT と ORDER BY の組み合わせは、最初に DISTINCT 節を満足する任意の行を選択し、次に ORDER BY 節に基づいてこれらの行を順序付けます。

## DISTINCT および GROUP BY

DISTINCT および GROUP BY は、指定されたフィールド (1 つまたは複数) でレコードをグループ化し、そのフィールドの一意な値ごとに 1 レコードを返します。それらの大きな違いは、DISTINCT では集約関数がグループ化の前に計算されるということです。GROUP BY では、集約関数はグループ化の後に計算されます。以下の例で、この違いを示します。

## SQL

```
SELECT DISTINCT BY (ROUND(Age,-1)) Age,AVG(Age) AS AvgAge FROM Sample.Person
/* AVG(Age) returns average of all ages in table */
```

## SQL

```
SELECT Age,AVG(Age) AS AvgAge FROM Sample.Person GROUP BY ROUND(Age,-1)
/* AVG(Age) returns an average age for each age group */
```

DISTINCT 節の指定では 1 つまたは複数の集約関数フィールドを使用できますが、集約関数は単一の値を返すため、これはあまり役に立ちません。したがって、以下の例では、1 つの行が返されます。

## SQL

```
SELECT DISTINCT BY (AVG(Age)) Name,Age,AVG(Age) FROM Sample.Person
```

**注意** 唯一の item または select-item として[集約関数](#)を含む DISTINCT 節を GROUP BY 節と共に使用すると、DISTINCT 節は無視されます。DISTINCT、集約関数、および GROUP BY を意図的に組み合わせるには、サブクエリを使用します。詳細とプログラムの例は、“[GROUP BY 節](#)”のリファレンス・ページを参照してください。

## 大文字/小文字の区別と DISTINCT の最適化

DISTINCT は、フィールドに対して定義された[照合タイプ](#)に基づいて、文字列値をグループ化します。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースに](#)

おける既定の文字列の照合を定義し、フィールド/プロパティの定義における既定以外のフィールドの照合タイプを指定することができます。

フィールド/プロパティの照合タイプが SQLUPPER である場合、グループ化されたフィールド値はすべて大文字で返されます。大文字/小文字の区別を元のままにして値をグループ化するには、またはグループ化されたフィールドの返り値を元の大文字/小文字区別で表示するには、**%EXACT** 照合関数を使用します。以下に例を示します。この例は、Home\_City フィールドが照合タイプ SQLUPPER で定義されており、‘New York’ と ‘new york’ の値が含まれていることを前提としています。

## SQL

```
SELECT DISTINCT BY (Home_City) Name,Home_City FROM Sample.Person
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in uppercase letters.
   Thus, 'NEW YORK' is returned. */
```

## SQL

```
SELECT DISTINCT BY (Home_City) Name,%EXACT(Home_City) FROM Sample.Person
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in original letter case.
   Thus, 'New York' or 'new york' may be returned, but not both. */
```

## SQL

```
SELECT DISTINCT BY (%EXACT(Home_City)) Name,Home_City FROM Sample.Person
/* groups together Home_City values by their original letter case
returns the name of each grouped city in original letter case.
Thus, both 'New York' and 'new york' are returned.
Optimization is not used. */
```

管理ポータルを使用して、DISTINCT 句を含むクエリのパフォーマンスを最適化できます。[\[システム管理\]](#)、[\[構成\]](#)、[\[SQL およびオブジェクトの設定\]](#)、[\[SQL\]](#) の順に選択します。[\[GROUP BY および DISTINCT クエリで元の値を生成する\]](#) オプションを表示して編集します (この最適化は [GROUP BY 節](#) にも有効です)。既定の設定は“いいえ”です。

この既定の設定では、アルファベット値が大文字の照合によってグループ化されます。この最適化は選択したフィールドのインデックスを利用します。そのため、インデックスが1つまたは複数の選択したフィールドに対して存在する場合にのみ意味があります。インデックス内に格納されるときにフィールド値が照合され、アルファベット文字列はすべて大文字で返されます。このシステム全体用オプションを設定した後に、特定のクエリのその設定を、%EXACT 照合関数を使用することで大文字/小文字区別を維持するように上書きできます。

詳細は、“システム管理ガイド” にリストされている [SQL およびオブジェクトの設定ページ](#)を参照してください。

また、`$SYSTEM.SQL.Util.SetOption()` メソッドの `FastDistinct` オプションを使用して、このオプションをシステム全体で設定することもできます。現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。これにより、`[DISTINCT` ] 設定が表示されます。既定値は 1 です。

## DISTINCT のその他の使用法

- ・ ストリーム・フィールド: DISTINCT は実際のデータに対してではなく、ストリーム・フィールドの OID に対して機能します。すべてのストリーム・フィールド OID は一意の値であるため、DISTINCT は実際のストリーム・フィールドの重複データ値に影響を与えません。DISTINCT BY (StreamField) は、ストリーム・フィールドが NULL のレコード数を、1 つの NULL レコードに削減します。詳細は、“[ストリーム・データ \(BLOB と CLOB\) の格納と使用](#)”を参照してください。
- ・ アスタリスク構文: 構文 DISTINCT \* は、正しい使用方法ですが役立ちません。これは定義上、すべての行に、重複しない一意の識別子が含まれているためです。構文 DISTINCT BY (\*) は、正しい使用方法ではありません。
- ・ サブクエリ: サブクエリでの DISTINCT 節の使用は正しい使用方法となりますが、サブクエリは単一の値を返すため、役には立ちません。
- ・ 選択する行データがない: DISTINCT 節は、テーブル・データにアクセスしない SELECT で使用できます。SELECT に FROM 節が含まれている場合には、DISTINCT を指定すると、このような非テーブル値を含む 1 行が返されま

す。DISTINCT (または TOP) を指定しないと、SELECT によって、FROM 節のテーブルにある行と同じ値を含む同数の行が返されます。SELECT に FROM 節が含まれていない場合の DISTINCT の使用は、正しい使用方法ですが役立ちません。詳細は、“FROM” 節を参照してください。

- ・ 集約関数：集約関数内で DISTINCT 節を使用することで、集約に含める対象の個別（一意）のフィールド値のみを選択できます。SELECT DISTINCT 節とは異なり、集約関数内の DISTINCT には、個別（一意）の値として NULL が含まれません。MAX および MIN 集約関数は、エラーなしで DISTINCT 節の構文を解析しますが、この構文はいかなる処理も実行しません。

## DISTINCT と %ROWID

DISTINCT キーワードを指定すると、[カーソル・ベースの埋め込み SQL クエリ](#)で %ROWID 変数が設定されなくなります。DISTINCT によって返される行が制限されない場合であっても、%ROWID は設定されません。以下に例を示します。

### ObjectScript

```
SET %ROWID=999
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT DISTINCT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"RowID: ",%ROWID," row count: ",%ROWCOUNT
    WRITE " Name=",name," State=",state
}
&sql(CLOSE EmpCursor)
```

クエリ動作のこの変更は、カーソル・ベースの埋め込み SQL SELECT クエリにのみ適用されます。ダイナミック SQL SELECT クエリと非カーソルの埋め込み SQL SELECT クエリでは、%ROWID が設定されることはありません。

## DISTINCT とトランザクション処理

DISTINCT キーワードを指定すると、クエリは、現在のトランザクションにまだコミットされていないデータを含む現在のすべてのデータを取得します。トランザクションの READ COMMITTED 分離モード・パラメータ（設定されている場合）は無視され、すべてのデータは READ UNCOMMITTED モードで取得されます。詳細は、“[トランザクション処理](#)”を参照してください。

## 例

以下のクエリでは、各個別の Home\_State 値ごとに 1 行が返されます。

### SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
ORDER BY Home_State
```

以下のクエリでは、各個別の Home\_State 値ごとに 1 行が返されますが、その行の追加フィールドが返されます。取得される行は予測できません。

### SQL

```
SELECT DISTINCT BY (Home_State) %ID,Name,Home_State,Office_State FROM Sample.Person
ORDER BY Home_State
```

以下のクエリでは、Home\_State 値と Office\_State 値の各個別の組み合わせごとに 1 行が返されます。データに応じて、前の例よりも多くの行が返されるか、前の例と同じ数の行が返されます。

## SQL

```
SELECT DISTINCT BY (Home_State,Office_State) %ID,Name,Home_State,Office_State FROM Sample.Person  
ORDER BY Home_State,Office_State
```

以下のクエリでは、DISTINCT BY が使用され、各個別の名前の長さごとに 1 行が返されます。

## SQL

```
SELECT DISTINCT BY ($LENGTH(Name)) Name,$LENGTH(Name) AS lname  
FROM Sample.Person  
ORDER BY lname
```

以下のクエリでは、DISTINCT BY が使用され、FavoriteColors %List 値の各個別の先頭要素ごとに 1 行が返されます。これは、FavoriteColors が NULL である個別の 1 行を表示します。

## SQL

```
SELECT DISTINCT BY ($LIST(FavoriteColors,1)) Name,FavoriteColors,$LIST(FavoriteColors,1) AS FirstColor  
FROM Sample.Person
```

以下のクエリは、Sample.Person から取得した最初の 20 個の異なる Home\_State 値を照合順の昇順で返します。“上位”行には、ORDER BY 節により、Sample.Person での全行の順序が反映されます。

## SQL

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person ORDER BY Home_State
```

以下のクエリは、メイン・クエリと WHERE 節サブクエリの両方で DISTINCT を使用しています。これによって、Sample.Employee にも含まれる、Sample.Person の最初の 20 個の異なる Home\_State 値が返されます。サブクエリ DISTINCT を使用しないと、Sample.Employee からランダムに選択した Home\_State 値と一致する Sample.Person の異なる Home\_State 値を取得します。

## SQL

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person  
WHERE Home_State IN(SELECT DISTINCT TOP 20 Home_State FROM Sample.Employee)  
ORDER BY Home_State
```

以下のクエリでは、個別の FavoriteColor 値のうち最初の 20 個が返されます。これには、ORDER BY 節により、Sample.Person 内での全行の順序が反映されます。FavoriteColors フィールドに NULL が含まれていることが判明しているため、FavoriteColors が NULL である重複しない 1 行が照合順の先頭になります。

## SQL

```
SELECT DISTINCT BY (FavoriteColors) TOP 20 FavoriteColors,Name FROM Sample.Person  
ORDER BY FavoriteColors
```

また、上記の例では FavoriteColors はリスト・フィールドなので、照合順序には要素のバイト長も加味されます。したがって、3 文字の要素 (RED) が先頭にある重複しないリスト値は、4 文字の要素 (BLUE) が先頭にあるリスト値の前にリストされます。

## 関連項目

- ・ [SELECT 文](#)
- ・ [GROUP BY 節](#)
- ・ [ORDER BY 節](#)
- ・ [TOP 節](#)



- ・ [集約関数](#)
- ・ [データベースの問い合わせ](#)
- ・ [照合](#)

## FROM (SQL)

クエリの対象となる 1 つ以上のテーブルを指定する SELECT 節です。

### 構文

```
SELECT ... FROM [optimize-option] table-ref
      [ [AS] t-alias ]
      [, [LATERAL] table-ref [ [AS] t-alias ] ] [, ...]
```

### 引数

引数	説明
optimize-hint	オプション - <a href="#">クエリ最適化オプション</a> を指定する、単一キーワード、または空白で区切られた一連のキーワード。詳細は、“ <a href="#">クエリでの最適化ヒントの指定</a> ”を参照してください。
table-ref	データの取得元の 1 つ以上の <a href="#">テーブル</a> 、 <a href="#">ビュー</a> 、 <a href="#">テーブル値関数</a> 、または <a href="#">サブクエリ</a> 。コンマで区切られたリストとしてか、JOIN 構文で指定されます。JOIN 構文でのビューの使用にはいくつかの制限が適用されます。サブクエリは、括弧で囲んで指定できません。
AS t-alias	オプション - テーブル名のエイリアス。有効な <a href="#">識別子</a> である必要があります。AS キーワードは省略可能で、これの有無に関係なくエイリアスを指定できます。
LATERAL	オプション - FROM 節で指定された以前のテーブルをラテラル参照できるようにします。詳細は、“ <a href="#">LATERAL キーワード</a> ”を参照してください。

### 概要

FROM 節は、[SELECT](#) 文内でデータを照会する 1 つ以上のテーブル（もしくはビューまたはサブクエリ）を指定します。テーブル・データが照会されていない場合には、以下のように、FROM 節はオプションです。

複数のテーブルは、コンマで区切られたリスト、または他の JOIN 構文によって区切られたリストとして指定されます。各テーブル名には、オプションとしてエイリアスを指定できます。

テーブル名のエイリアスは、SELECT 文で複数のテーブルのフィールド名を指定するときに使用されます。FROM 節で複数のテーブルを指定している場合は、SELECT の select-item 節で、フィールドを `tablename.fieldname` の形式で指定することにより、どのテーブルのフィールドを対象とするのかを示すことができます。この場合、テーブル名は長い名前が多いので、短いテーブル名のエイリアスを指定すると便利です (`t-alias.fieldname`)。

以下の例は、テーブル名のエイリアスの使用方法を示します。

### SQL

```
SELECT e.Name,c.Name
FROM Sample.Company AS c,Sample.Employee AS e
```

AS キーワードは省略できます。このキーワードは、互換性とわかりやすい表記のためのものです。

### テーブル参照に対するスキーマ名の指定

table-ref 名は、修飾 (`schema.tablename`) でも未修飾 (`tablename`) でもかまいません。未修飾のテーブル名（またはビュー名）のスキーマ名は、次のように、スキーマ検索パスまたはシステム全体の既定のスキーマ名を使用して指定されます。

1. [スキーマ検索パス](#)が指定されている場合、指定スキーマで一致テーブル名が検索されます。
2. スキーマ検索パスが指定されていないか、スキーマ検索パスで一致が生成されない場合、[システム全体の既定のスキーマ名](#)が使用されます。

## テーブルの結合

FROM 節で複数のテーブル名を指定すると、InterSystems SQL はこれらのテーブルに対して結合操作を実行します。実行される結合のタイプは、join キーワード句またはテーブル名の各ペアの間の記号で指定します。2 つのテーブル名がコンマで区切られているときは、交差結合が実行されます。結合のさまざまなタイプと構文の詳細は、["JOIN"](#) を参照してください。

結合が実行される順序は、SQL クエリ・オブティマイザによって自動的に決定され、クエリにテーブルがリストされる順序には基づきません。必要に応じて、クエリ・オブティマイザのオプションを指定して結合を実行する順序を制御できます。

最初の 2 つの SELECT 文は、2 つの個別のテーブルに対する行カウントを示し、3 番目の例は、両方のテーブルを指定する SELECT に対する行カウントを示しています。後者の例は、より大きなテーブル、デカルト積になります。この場合、1 番目のテーブルのそれぞれの行が、2 番目のテーブルのそれぞれの行に一致します。これは、[Cross Join](#) という処理です。

### SQL

```
SELECT COUNT(*)
FROM Sample.Company
```

### SQL

```
SELECT COUNT(*)
FROM Sample.Vendor
```

### SQL

```
SELECT COUNT(*)
FROM Sample.Company, Sample.Vendor
```

CROSS JOIN 構文を明示的に使用しても同じ動作を実行できます。

### SQL

```
SELECT COUNT(*)
FROM Sample.Company CROSS JOIN Sample.Vendor
```

多くの場合、Cross Join の広範囲なデータ重複は望ましくありません。その他の Join タイプをお勧めします。

SELECT 文で [WHERE 節](#) を指定すると、交差結合が実行され、WHERE 節の述語によって結果セットが決まります。これは ON 節を指定した INNER JOIN を実行することと同じです。したがって、以下の 2 つの例は同じ結果を返します。

### SQL

```
SELECT p.Name, p.Home_State, em.Name, em.Office_State
FROM Sample.Person AS p, Sample.Employee AS em
WHERE p.Name %STARTSWITH 'E' AND em.Name %STARTSWITH 'E'
```

### SQL

```
SELECT p.Name, p.Home_State, em.Name, em.Office_State
FROM Sample.Person AS p INNER JOIN Sample.Employee AS em
ON p.Name %STARTSWITH 'E' AND em.Name %STARTSWITH 'E'
```

FROM table-refリストで明示的な結合構文を指定すると(コンマを使用するのではない)、別のタイプの結合操作を実行できます。詳細は、["JOIN"](#) を参照してください。

## クエリ最適化オプション

既定で、InterSystems SQL クエリ・オブティマイザは、高度で柔軟性の高いアルゴリズムを使用して結合オペレーションや複数のインデックスを含む複雑なクエリのパフォーマンスを最適化します。ほとんどの場合、これらの既定によってパフォーマンスが最適化されます。ただし、インターシステムズのサポート窓口にお問い合わせいただくと、クエリ最適化についての解釈を 1 つ以上指定することで、クエリ・オブティマイザに“ヒント”を与えるように指示されることがあります。これらのヒントは、FROM で optimize-hint 引数を使用して指定します。複数の最適化ヒントは空白で区切り、任意の順序で指定できます。詳細は、“[クエリでの最適化ヒントの指定](#)”を参照してください。

FROM 節内の単純な SELECT 文、CREATE VIEW ビュー定義の SELECT 文、またはサブクエリ SELECT 文で optimize-hint FROM 節キーワードを使用できます。

## FROM 節におけるテーブル値関数

### SQL

```
SELECT Name,DOB FROM Sample.SP_Sample_By_Name('A')
```

以下のダイナミック SQL の例では、同じテーブル値関数を指定します。%Execute() メソッドを使用して ? 入力パラメータにパラメータ値を指定します。

### ObjectScript

```
SET myquery="SELECT Name,DOB FROM Sample.SP_Sample_By_Name(?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("A")
DO rset.%Display()
WRITE !,"End of A data",!!
SET rset = tStatement.%Execute("B")
DO rset.%Display()
WRITE !,"End of B data"
```

テーブル値関数は、SELECT 文または **DECLARE** 文の FROM 節でのみ使用できます。テーブル値関数名は、スキーマ名で修飾することも、未修飾 (スキーマ名なし) にすることもできます。未修飾の名前は、既定のスキーマを使用します。SELECT 文の FROM 節では、テーブル名を使用できる箇所などどこでもテーブル値関数を使用できます。ビューまたはサブクエリでも使用でき、コンマ区切りのリストまたは明示的な JOIN 構文を使用して他の table-ref 項目に結合できます。

テーブル値関数は、INSERT、UPDATE、または DELETE 文で直接使用することはできません。ただし、テーブル値関数を指定するこれらのコマンドのためのサブクエリを指定することはできます。

InterSystems SQL は、テーブル値関数のための EXTENTSIZE、またはテーブル関数列のための SELECTIVITY は定義しません。

## FROM 節のサブクエリ

FROM 節ではサブクエリを指定できます。これは、ストリーム化されたサブクエリとして知られています。サブクエリは、JOIN 構文での使用や AS キーワードを使用したオプションのエイリアス割り当てなども含めて、テーブルと同様に扱われます。FROM 節には、複数のテーブル、ビュー、およびサブクエリを任意の組み合わせで含むことができますが、“**JOIN**”で説明されている JOIN 構文の制約を受けます。

サブクエリは括弧で囲まれます。以下の例は、FROM 節のサブクエリを示しています。

### SQL

```
SELECT name,region
FROM (SELECT t1.name,t1.state,t2.region
      FROM Employees AS t1 LEFT OUTER JOIN Regions AS t2
      ON t1.state=t2.state)
GROUP BY region
```

サブクエリでは [TOP 節](#) を指定できます。サブクエリには、TOP 節と組み合わせて [ORDER BY 節](#) を記述できます。

サブクエリでは `SELECT *` 構文を使用できます。ただし、FROM 節から得られるのは数値としての式なので、`SELECT *` を指定したサブクエリでは 1 つの列のみが生成されるようにする必要があります。

サブクエリ内の結合を NATURAL 結合にしたり、USING 節を指定することはできません。

## FROM サブクエリと %VID

FROM サブクエリが呼び出されると、このサブクエリは、返されるサブクエリ行ごとに %VID を返します。%VID は整数カウンタ・フィールドです。その値は、システムによって割り当てられる一意の非 NULL かつ非ゼロ値であり、変更することはできません。%VID は、明示的に指定された場合にのみ返されます。これはデータ型 INTEGER として返されます。%VID の値は連続した整数であるため、サブクエリが順序付きデータを返す場合はとても有意義なものになります。TOP 節と組み合わせるときには、サブクエリでは ORDER BY 節以外は使用できません。

%VID は連続した整数であるため、%VID を使用して、ORDER BY 節が使用されたサブクエリ内の項目のランキングを特定できます。以下の例では、最も新しい 10 件のレコードが Name の順にリストされますが、それらのタイムスタンプ・ランキングは %VID の値を使用して簡単に確認できます。

### SQL

```
SELECT Name,%VID,TimeStamp FROM
  (SELECT TOP 10 * FROM MyTable ORDER BY TimeStamp DESC)
ORDER BY Name
```

%VID の一般的な用途の 1 つは、実行内容を 1 つの表示ウィンドウに収まる行数に合致する連続したサブセットに分割して、結果セットを“ウィンドウ表示”することです。例えば、20 件のレコードを表示してから、ユーザが Enter キーを押すまで待ち、次の 20 件のレコードを表示できます。

以下の例は、%VID を使用して、結果を 10 件のレコードごとのサブセットに分割して“ウィンドウ表示”します。

### SQL

```
SELECT %VID,* FROM
  (SELECT TOP 60 Name, Age FROM Sample.Person WHERE Age > 55 ORDER BY Name)
WHERE %VID BETWEEN ? AND ?
```

%VID の使用の詳細は、[“ビューの定義と使用”](#) を参照してください。

## LATERAL キーワード

FROM 節で以前に指定したテーブルのフィールド値をビューおよびテーブル値関数で参照できるようにすることで、LATERAL キーワードを使用して FROM 処理の順序をより明示的に制御できます。このようなラテラル参照は、サブクエリまたはテーブル値関数で生成する行に影響します。

LATERAL キーワードを指定したサブクエリまたはテーブル値関数では、ラテラル参照するフィールドが指定値と見なされます。ラテラル参照先のフィールドは、必ず同じ FROM 節にある以前の FROM アイテムから得られます。

FROM サブクエリに先行して記述した LATERAL キーワードは、クエリの中で意味的に先行する FROM アイテムのフィールドを、このサブクエリが参照する可能性があることを示します。このようなラテラル参照先のフィールドは、LATERAL が指定された FROM サブクエリよりも先に処理されます。

テーブル値関数に先行して記述した LATERAL キーワードは、以前の FROM アイテムのフィールドを、テーブル値関数で使用できることを示しています。このコンテキストでは、このキーワードはオプションであり、このような参照をテーブル値関数で使用するラテラル結合が暗黙的に適用されます。

## オプションの FROM 節

SELECT 項目リストで (直接的にも間接的にも) テーブル・データが参照されていない場合、FROM 節はオプションです。この種類の SELECT は、関数、演算子式、定数またはホスト変数からデータを返す場合に使用できます。テーブル・データを参照しないクエリでは、以下のようになります。

- FROM 節が省略された場合には、TOP キーワードの値に関係なく、返されるデータ行は最大でも 1 つです。TOP が 0 の場合には、データが返されません。DISTINCT 節は無視されます。特権は不要です。
- FROM 節を指定する場合には、現在のネームスペースに既存のテーブルを指定する必要があります。そのテーブルが参照されない場合でも、テーブルに SELECT 特権を設定する必要があります。TOP 節または DISTINCT 節を指定した場合、あるいはその節を WHERE 節または HAVING 節で制限した場合を除き、返される同一データ行の数は、指定したテーブルの行数と等しくなります。DISTINCT 節を指定すると、出力が単一のデータ行に限定されます。TOP キーワードを指定すると、出力は TOP 値で指定された行数に制限されます。TOP が 0 の場合には、データが返されません。

FROM 節を使用した場合でも使用しない場合でも、後続の節 (WHERE、GROUP BY、HAVING または ORDER BY) は指定できます。結果を返すかどうか、または同一の結果行をいくつ返すかを決定する場合には、WHERE 節または HAVING 節を使用できます。FROM 節を指定していない場合でも、これらの節はテーブルを参照できます。GROUP BY 節または ORDER BY 節を指定することもできますが、これらの節は無意味です。

以下は、テーブル・データを参照しない SELECT 文の例です。どちらの例も、情報を 1 行返します。

以下の例では、FROM 節が省略されています。DISTINCT キーワードは不要ですが、指定することはできます。SELECT 節は許可されていません。

## SQL

```
SELECT 3+4 AS Arith,
       {fn NOW} AS NowDateTime,
       {fn DAYNAME({fn NOW})} AS NowDayName,
       UPPER('MixEd cAsE EXPreSSioN') AS UpCase,
       {fn PI} AS PiConstant
```

以下の例では、FROM 節が含まれています。DISTINCT キーワードは、1 つのデータ行を返すために使用されています。FROM 節で参照するテーブルは、有効なテーブルである必要があります。ここでは ORDER BY 節が許可されていますが無意味です。ORDER BY 節では、有効な選択項目の別名を指定する必要があります。

## SQL

```
SELECT DISTINCT 3+4 AS Arith,
       {fn NOW} AS NowDateTime,
       {fn DAYNAME({fn NOW})} AS NowDayName,
       UPPER('MixEd cAsE EXPreSSioN') AS UpCase,
       {fn PI} AS PiConstant
FROM Sample.Person
ORDER BY NowDateTime
```

以下の例は両方とも、結果を返すかどうかを決定するために WHERE 節を使用しています。最初の例では FROM 節が含まれており、DISTINCT キーワードを使用して 1 つのデータ行を返します。2 番目の例では FROM 節が省略されているため、返されるデータ行は最大でも 1 つです。どちらの場合でも、WHERE 節で参照するテーブルは、SELECT 特権を設定した有効なテーブルである必要があります。

## SQL

```
SELECT DISTINCT
       {fn NOW} AS DataOKDate
FROM Sample.Person
WHERE FOR SOME (Sample.Person) (Name %STARTSWITH 'A')
```

## SQL

```
SELECT {fn NOW} AS DataOKDate
WHERE FOR SOME (Sample.Person) (Name %STARTSWITH 'A')
```

## 関連項目

- SELECT

- ・ JOIN
- ・ データベースの問い合わせ
- ・ テーブルの定義
- ・ SQL クエリの最適化
- ・ SQL 文と SQL 統計の分析
- ・ SQLCODE エラー・メッセージ



## GROUP BY (SQL)

1 つ以上の列に基づいて、クエリの結果行をグループ化する SELECT 節です。

### 構文

```
SELECT ...  
GROUP BY field {,field2}
```

### 概要

GROUP BY は [SELECT](#) コマンドの節です。オプションの GROUP BY 節は、FROM 節とオプションの WHERE 節の後、オプションの HAVING 節と ORDER BY 節の前に記述されます。

GROUP BY 節はクエリの結果行を受け取り、単独または複数のデータベース列によって結果行を個別のグループに分割します。SELECT を GROUP BY と併用するとき、GROUP BY フィールドの個別の各値に対して、1 行が取り出されます。GROUP BY は、NULL (値が指定されていない) を含むフィールドを別個の個別値グループとして扱います。

GROUP BY 節は概念的には InterSystems IRIS の集約関数の拡張キーワードである %FOREACH と似ていますが、%FOREACH はクエリ全体の母集団を制約することなく、サブ母集団で集約の選択ができるのに対し、GROUP BY はクエリ全体で実行します。

GROUP BY は、[INSERT](#) コマンドの SELECT 節で使用できます。GROUP BY は、UPDATE コマンドまたは DELETE コマンドでは使用できません。

### フィールドの指定

GROUP BY 節の最も簡潔な形式では、GROUP BY City など、1 つのフィールドを指定します。これにより、一意の City 値ごとに任意の 1 つの行を選択します。フィールドのコンマ区切りリストを指定することもできます。このリストの結合された値は、単一のグループ化項として扱われます。これにより、City および Age 値の一意の組み合わせごとに任意の 1 つの行を選択します。したがって、GROUP BY City,Age は、GROUP BY Age,City と同じ結果を返します。

フィールドは、列名で指定する必要があります。有効な field 値としては、列名 (GROUP BY City) や、%ID (すべての行を返す) や、列名を指定するスカラー関数 (GROUP BY ROUND(Age,-1)) や、列名を指定する[照合関数](#) (GROUP BY %EXACT(City)) があります。

列エイリアスでフィールドを指定できません。これを試行すると、SQLCODE -29 エラーが生成されます。列番号でフィールドを指定することはできません。これはリテラルとして解釈され、1 つの行を返します。集約フィールドを指定できません。これを試行すると、SQLCODE -19 エラーが生成されます。サブクエリを指定することはできません。これはリテラルとして解釈され、1 つの行を返します。

GROUP BY StreamField は実際のデータに対してではなく、ストリーム・フィールドの OID に対して機能します。すべてのストリーム・フィールド OID は一意の値であるため、GROUP BY は実際のストリーム・フィールドの重複データ値に影響を与えません。GROUP BY StreamField は、ストリーム・フィールドが NULL のレコード数を、1 レコードに削減します。詳細は、“[ストリーム・データ \(BLOB と CLOB\) の格納と使用](#)” を参照してください。

GROUP BY 節では、矢印構文 (->) 演算子を使用して、ベース・テーブルではないテーブル内のフィールドを指定できます。例えば、GROUP BY Company->Name のように使用します。詳細は、“[暗黙結合 \(矢印構文\)](#)” を参照してください。

GROUP BY 節内の field 値としてリテラルを指定すると、1 行が返されます。どの行が返されるかは不確定となります。したがって、7、'Chicago'、'、0、または NULL のどれを指定しても、1 行が返されます。ただし、コンマ区切りリスト内の field 値としてリテラルを指定すると、そのリテラルは無視され、GROUP BY が指定されたフィールド名の一意の組み合わせごとに任意の 1 つの行を選択します。

## GROUP BY および DISTINCT BY での集約関数

GROUP BY 節は、[集約関数](#)が計算される前に適用されます。以下の例では、COUNT 集約関数で、各 GROUP BY グループ内の行数をカウントします。

### SQL

```
SELECT Home_State,COUNT(Home_State)
FROM Sample.Person
GROUP BY Home_State
```

DISTINCT BY 節は、集約関数が計算された後に適用されます。以下の例では、COUNT 集約関数で、テーブル全体の行数をカウントします。

### SQL

```
SELECT DISTINCT BY(Home_State) Home_State,COUNT(Home_State)
FROM Sample.Person
```

テーブル全体に対する集約関数を計算するには、GROUP BY グループではなく、select-item サブクエリを指定します。

### SQL

```
SELECT Home_State,(SELECT COUNT(Home_State) FROM Sample.Person)
FROM Sample.Person
GROUP BY Home_State
```

選択リストが集約フィールドで構成されている場合、[DISTINCT 節](#)で GROUP BY 節を使用しないでください。例えば、次のクエリは Home\_State 値が同じ人の数を返すことを意図して作成したものです。

### SQL

```
/* This query DOES NOT apply the DISTINCT keyword */
/* It is provided as a cautionary example */
SELECT DISTINCT COUNT(*) AS mynum
FROM Sample.Person
GROUP BY Home_State
ORDER BY mynum
```

このクエリの場合、DISTINCT キーワードを適用していないので、想定した結果が得られません。DISTINCT 集約と GROUP BY 節の両方を適用するには、次の例で示すようにサブクエリを使用します。

### SQL

```
SELECT DISTINCT *
FROM (SELECT COUNT(*) AS mynum
      FROM Sample.Person
      GROUP BY Home_State) AS Sub
ORDER BY Sub.mynum
```

この例では、Home\_State 値が同じ人の数が正しく得られます。例えば、ある Home\_State 値を持つ人が 8 人いる場合、このクエリからは 8 が返されます。

クエリが集約関数のみで構成されていて、テーブルからのデータを返さない場合、集約関数に対して %ROWCOUNT=1 と空文字列 (または 0) の値が返されます。以下に例を示します。

### SQL

```
SELECT AVG(Age) FROM Sample.Person WHERE Name %STARTSWITH 'ZZZZ'
```

ただし、このタイプのクエリに GROUP BY 節が含まれている場合は、%ROWCOUNT=0 が返され、集約関数の値は未定義のままになります。

## 照合、大文字/小文字の区別、および最適化

このセクションでは、大文字/小文字の区別のみが異なるデータ値を GROUP BY が処理する方法について説明します。

- 大文字/小文字の区別が異なる値を同一グループに分類する (大文字を返す):

既定では、GROUP BY は、作成時に field に対して指定された照合に基づいて、文字列値をグループ化します。InterSystems IRIS には既定の文字列照合があり、ネームスペースごとにこれを設定できます。すべてのネームスペースにおける最初の文字列照合の既定値は SQLUPPER です。したがって、通常、指定がない限り、GROUP BY 照合では大文字と小文字は区別されません。

GROUP BY は、大文字の照合に基づいて SQLUPPER 照合でフィールドの値をグループ化します。大文字/小文字の区別のみが異なるフィールド値は、同一グループに分類されます。グループ化されたフィールド値は、すべて大文字で返されます。これには、GROUP BY が、実際のフィールド値にアクセスするのではなく、フィールドのインデックスを使用できるというパフォーマンス上の利点があります。そのため、インデックスが 1 つまたは複数の選択フィールドに対して存在する場合にのみ意味があります。この場合、結果的に、実際のデータ値がすべて大文字でなくても、GROUP BY フィールドの値はすべて大文字で返されます。

- 大文字/小文字の区別が異なる値を同一グループに分類する (実際の的大文字/小文字を返す):

GROUP BY は、大文字/小文字の区別が異なる値を同一グループに分類し、グループ化されたフィールドの値を実際のフィールドの大文字/小文字の値で返すことができます (ランダムに選択されます)。これには、返される値が実際の値であり、データ内の少なくとも 1 つの値の大文字/小文字の区別が表示されるという利点があります。フィールドのインデックスを使用できないというパフォーマンス上の欠点があります。select-item フィールドに %EXACT 照合関数を適用することで、個々のクエリに対してこれを指定することができます。

- 大文字/小文字の区別が異なる値を同一グループに分類しない (実際の的大文字/小文字を返す):

GROUP BY は、%EXACT 照合関数を GROUP BY フィールドに適用することによって、大文字/小文字を区別して値をグループ化することができます。これには、大文字/小文字の区別が異なる値をすべて別々のグループとして返すという利点があります。フィールドのインデックスを使用できない、というパフォーマンス上の欠点があります。

この動作は、管理ポータルを使用して、GROUP BY 節を含むすべてのクエリを対象としてシステム全体について構成できます。[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GROUP BY および DISTINCT クエリで元の値を生成する] チェック・ボックスを表示して編集します。既定では、このチェック・ボックスにはチェックが付いていません。この既定の設定では、アルファベット値が大文字の照合によってグループ化されます。(この最適化は DISTINCT 節にも有効です)。詳細は、“SQL およびオブジェクトの設定ページ” を参照してください。

また、\$SYSTEM.SQL.Util.SetOption() メソッドの FastDistinct オプションを使用して、このオプションをシステム全体で設定することもできます。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[DISTINCT] 1 設定が表示されます。既定値は 1 です。

この最適化は選択したフィールドのインデックスを利用します。そのため、インデックスが 1 つまたは複数の選択したフィールドに対して存在する場合にのみ意味があります。インデックス内に格納されるときにフィールド値が照合され、アルファベット文字列はすべて大文字で返されます。このシステム全体用オプションを設定した後に、特定のクエリのその設定を、%EXACT 照合関数を使用することで大文字/小文字区別を維持するように上書きできます。

以下の例は、これらの動作を示しています。これらの例では、Sample.Person に、SQLUPPER 照合、ならびに ‘New York’ および ‘new york’ という Home\_City フィールド値があるレコードが含まれているとします。

### SQL

```
SELECT Home_City FROM Sample.Person GROUP BY Home_City
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in uppercase letters.
   Thus, 'NEW YORK' is returned. */
```

## SQL

```
SELECT %EXACT(Home_City) FROM Sample.Person GROUP BY Home_City
/* groups together Home_City values by their uppercase letter values
   returns the name of a grouped city in original letter case.
   Thus, 'New York' or 'new york' may be returned, but not both. */
```

## SQL

```
SELECT Home_City FROM Sample.Person GROUP BY %EXACT(Home_City)
/* groups together Home_City values by their original letter case
   returns the name of each grouped city in original letter case.
   Thus, both 'New York' and 'new york' are returned as separate groups. */
```

## %ROWID

GROUP BY 節を指定すると、[カーソル・ベースの埋め込み SQL クエリ](#)で %ROWID 変数が設定されなくなります。GROUP BY によって返される行が制限されない場合であっても、%ROWID は設定されません。以下に例を示します。

### ObjectScript

```
SET %ROWID=999
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'M'
    GROUP BY Home_State)
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"RowID: ",%ROWID," row count: ",%ROWCOUNT
    WRITE " Name=",name," State=",state
}
&sql(CLOSE EmpCursor)
```

クエリ動作のこの変更は、カーソル・ベースの埋め込み SQL SELECT クエリにのみ適用されます。ダイナミック SQL SELECT クエリと非カーソル埋め込み SQL SELECT クエリでは、%ROWID が設定されることはありません。

## トランザクションでのコミットされた変更

GROUP BY 節を含むクエリでは、READ COMMITTED 分離レベルはサポートされていません。READ COMMITTED として定義されたトランザクションでは、GROUP BY 節がない SELECT 文は、コミットされている変更データのみを返します。つまり、このような文は現在のトランザクションの前のデータの状態を返します。GROUP BY 節のある SELECT 文は、コミットされているかどうかに関係なく、実行されたすべての変更データを返します。

## 引数

### field

1 つ以上のデータ取得元フィールド。1 つのフィールド名、またはコンマで区切られたフィールド名のリスト。

## 例

以下の例では、最初の文字で名前をグループ化しています。最初の文字、その最初の文字を共有する名前の数、および名前の値の 1 つの例を返します。名前は、実際の値の大文字/小文字の区別に関係なく、SQLUPPER 照合を使用してグループ化されます。Name select-item に含まれている最初の文字が大文字であることに注意してください。% EXACT 照合は、実際の名前の値を表示するために使用されます。

## SQL

```
SELECT Name AS Initial,COUNT(Name) AS SameInitial,%EXACT(Name) AS Example
FROM Sample.Person GROUP BY %SQLUPPER(Name,2)
```

## 関連項目

- ・ [SELECT](#)
- ・ [DISTINCT](#) 節
- ・ [JOIN](#)

# HAVING (SQL)

データ値のグループに対して 1 つ以上の制限条件を指定する SELECT 節です。

## 構文

```
SELECT field
FROM table GROUP BY field
HAVING condition-expression

SELECT aggregatefunc(field %AFTERHAVING)
FROM table [GROUP BY field]
HAVING condition-expression
```

## 概要

オプションの HAVING 節は、FROM 節とオプションの WHERE 節および GROUP BY 節の後、オプションの ORDER BY 節の前に記述されます。

**SELECT** 文の HAVING 節は、クエリで選択された特定の行の適格/不適格を決めます。適格な行とは、condition-expression が True である行です。condition-expression は、AND および OR 論理演算子によってリンク可能な論理テスト (述語) の一式です。詳細は、“**WHERE**” 節を参照してください。

HAVING 節は、データ・セット全体ではなく、グループに対して処理を実行できる WHERE 節に似ています。したがって、ほとんどの場合、HAVING 節は、%AFTERHAVING キーワードを使用した**集約関数**で使用するか、**GROUP BY** 節と組み合わせて使用するか、またはその両方で使用します。

HAVING 節の condition-expression では、**集約関数**を指定することもできます。WHERE 節の condition-expression では、集約関数は指定できません。詳細は、以下の例を参照してください。

## SQL

```
SELECT Name, Age, AVG(Age) AS AvgAge
FROM Sample.Person
HAVING Age > AVG(Age)
ORDER BY Age
```

HAVING 節は、サブ母集団の集約と、全体の母集団の集約とを頻繁に比較します。

## フィールドの指定

HAVING 節の condition-expression または %AFTERHAVING キーワード表現で指定されるフィールドは、フィールド名または集約関数として指定される必要があります。列番号でフィールドまたは集約関数を指定することはできません。列エイリアスでフィールドまたは集約関数を指定できません。これを試行すると、SQLCODE -29 エラーが生成されます。ただし、サブクエリを使用して列エイリアスを定義してから、このエイリアスを HAVING 節で使用できます。例えば以下のようになります。

## SQL

```
SELECT Y AS TeenYear, AVG(Y %AFTERHAVING) AS AvgTeenAge FROM
  (SELECT Age AS Y FROM Sample.Person WHERE Age < 20)
HAVING Y > 12 ORDER BY Y
```

## select-item リストの集約関数

HAVING 節は、返す行を選択します。既定では、この行の選択によって select-item リストの集約関数の値は決定されません。この理由は、HAVING 節は、select-item リストの集約関数の後で解析されるからです。

以下の例では、Age > 65 の行のみが返されます。ただし、AVG(Age) は、HAVING 節で選択された行だけでなく、すべての行に基づいて計算されます。

## SQL

```
SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
HAVING Age > 65
ORDER BY Age
```

これを WHERE 節と比較すると、WHERE 節は、返す行と、select-item リストの集約関数に含める行の値を選択します。

## SQL

```
SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
WHERE Age > 65
ORDER BY Age
```

HAVING 節は、集約値のみを返すクエリで使用できます。

- ・ 集約しきい値：HAVING 節は、集約しきい値を使用して、(クエリの集約値を含む) 1 行を返すか 0 行を返すかを決定します。したがって、HAVING 節を使用することで、集約しきい値に達した場合のみ集約計算を返すことができます。以下の例では、テーブルの行数が 100 行以上の場合のみ、テーブルのすべての行の Age 値の平均が返されます。行数が 100 行未満の場合、すべての行の Age 値の平均は無意味なことがあるので、返されません。

## SQL

```
SELECT AVG(Age) FROM Sample.Person HAVING COUNT(*)>99
```

- ・ 複数行：集約関数が含まれる HAVING 節が指定され、GROUP BY 節が指定されていない場合は、HAVING 節の条件を満たす行の数が返されます。集約関数値は、テーブルのすべての行に基づいて計算されます。

## SQL

```
SELECT AVG(Age) FROM Sample.Person HAVING %ID<10
```

これは、1 行を返す、集約関数が含まれる WHERE 節とは対照的です。集約関数値は、WHERE 節の条件を満たす行に基づいて計算されます。

## SQL

```
SELECT AVG(Age) FROM Sample.Person WHERE %ID<10
```

## %AFTERHAVING

%AFTERHAVING キーワードは select-item リストで集約関数と共に使用して、HAVING 節の条件の適用後に、集約演算を実行することを指定できます。

## SQL

```
SELECT Name, Age, AVG(Age) AS AvgAge,
AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
HAVING Age > 40 AND Age < 65
ORDER BY Age
```

以下の両方の考慮事項を満たした場合のみ、%AFTERHAVING キーワードは意味のある結果を返します。

- ・ select-item リストには、非集約のフィールド参照である項目を少なくとも 1 つ含める必要があります。このフィールド参照の対象は、FROM 節で指定された任意のテーブルの任意のフィールド、つまり、暗黙結合 (矢印構文)、%ID エイリアス、またはアスタリスク (\*) を使用して参照されるフィールドになります。
- ・ HAVING 節の条件は、少なくとも 1 つの非集約条件に適用する必要があります。したがって、HAVING Age>50、HAVING Age>AVG(Age)、または HAVING Age>50 AND MAX(Age)>75 は有効な条件ですが、HAVING Age>50 OR MAX(Age)>75 は有効な条件ではありません。



以下の例では、HAVING 節と GROUP BY 節を使用して、州の平均年齢を返すと共に、テーブルのすべての行の平均年齢よりも高齢な人について州の平均年齢を返しています。また、サブクエリを使用して、テーブルのすべての行の平均年齢を返しています。

## SQL

```
SELECT Home_State, (SELECT AVG(Age) FROM Sample.Person) AS AvgAgeAllRecs,
    AVG(Age) AS AvgAgeByState, AVG(Age %AFTERHAVING) AS AvgOlderByState
FROM Sample.Person
GROUP BY Home_State
HAVING Age > AVG(Age)
ORDER BY Home_State
```

## 引数

### condition-expression

どのデータ値が取得されるかを規定する 1 つまたは複数のブーリアン述語で構成される式。

## 論理述語

SQL の述語は以下のカテゴリに分類されます。

- ・ [等値比較述語](#)
- ・ [BETWEEN 述語](#)
- ・ [IN および %INLIST 述語](#)
- ・ [%STARTSWITH 述語](#)
- ・ [包含関係演算子 \(⊃\)](#)
- ・ [FOR SOME 述語](#)
- ・ [NULL 述語](#)
- ・ [EXISTS 述語](#)
- ・ [LIKE、%MATCHES、および %PATTERN 述語](#)
- ・ [%INSET および %FIND 述語](#)

**注釈** HAVING 節では、FOR SOME %ELEMENT コレクション述語を使用できません。この述語は、WHERE 節でのみ使用できます。

## 述語の大文字と小文字の区別

述語では、フィールドに対して定義された[照合タイプ](#)が使用されます。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

%INLIST、包含関係演算子 (⊃)、%MATCHES、および %PATTERN 述語は、フィールドの既定の照合を使用しません。常に大文字と小文字を区別する EXACT 照合が使用されます。

2 つのリテラル文字列の熟語の比較は、常に大文字小文字を区別します。

## 述語の条件と %NOINDEX

述語の条件の前に %NOINDEX キーワードを置くと、クエリ・オプティマイザが条件でインデックスを使用することを防ぐことができます。これは、多数の行を満たす範囲条件を指定する場合に最も便利です。例えば、HAVING %NOINDEX Age >= 1 のようにします。詳細は ["インデックスの最適化オプション"](#) を参照してください。

## 等値比較述語

以下は、使用できる比較述語です。

テーブル C-1: SQL 等値比較述語

述語	処理
=	等しい
<>	等しくない
!=	等しくない
>	より大きい
<	より小さい
>=	以上
<=	以下

以下の例は、比較述語を使用しています。21 より小さい Age ごとに 1 つのレコードが返されます。

### SQL

```
SELECT Name, Age FROM Sample.Person
GROUP BY Age
HAVING Age < 21
ORDER BY Age
```

SQL は照合 (値がソートされる順番) という点から比較演算子を定義します。まったく同様の方法で照合する場合の 2 つの値は等しくなります。2 つ目の値の後に照合される場合、値は別の値よりも大きくなります。文字列データ型の **フィールド照合** は、フィールドの既定の照合に基づきます。既定では、大文字と小文字は区別されません。そのため、2 つの文字列フィールドの値の比較または文字列フィールド値と文字列リテラルとの比較では、既定では大文字と小文字は区別されません。例えば、Home\_State フィールドの値が大文字の 2 文字の文字列の場合、以下のようになります。

式	値
'MA' = Home_State	値 MA に対して True
'ma' = Home_State	値 MA に対して True
'VA' < Home_State	値 VT、WA、WI、WV、WY に対して True
'ar' >= Home_State	値 AK、AL、AR に対して True

ただし、2 つのリテラル文字列の比較では大文字と小文字が区別され、WHERE 'ma' = 'MA' は常に FALSE です。

## BETWEEN 述語

以下の例は、BETWEEN 述語を使用しています。これは、「以上」と「以下」の組み合わせと同じ働きをします。18 と 35 を含む、18 から 35 までの Age ごとに 1 つのレコードが返されます。

### SQL

```
SELECT Name, Age FROM Sample.Person
GROUP BY Age
HAVING Age BETWEEN 18 AND 35
ORDER BY Age
```

詳細は、“**BETWEEN**” を参照してください。

## IN および %INLIST 述語

IN 述語は、構造化されていない一連の項目に値を一致させるために使用されます。

%INLIST 述語は、値をリスト構造の要素に一致させるための InterSystems IRIS の拡張機能です。

どちらの述語を使用しても、等値比較やサブクエリ比較を実行できます。

IN には 2 つの形式があります。1 つ目は、OR 演算子で複数の等値比較を結合する省略表現として使用します。以下はその例です。

### SQL

```
SELECT Name, Home_State FROM Sample.Person
GROUP BY Home_State
HAVING Home_State IN ( 'ME', 'NH', 'VT', 'MA', 'RI', 'CT' )
```

上記の文は、Home\_State が括弧のリスト内の値と等しい場合、True と評価します。リストの要素は定数または式を指定できます。等式テストと同様に、[照合](#)が IN 比較に適用されます。既定では、IN 比較でフィールド定義の照合タイプが使用されます。既定では、文字列フィールドは大文字と小文字が区別されない SQLUPPER として定義されます。

IN 述語の等値比較に日付または時刻を使用すると、適切なデータ型変換が自動的に実行されます。HAVING 節フィールドが TimeStamp 型の場合、Date 型または Time 型の値は Timestamp に変換されます。HAVING 節フィールドが Date 型の場合、TimeStamp 型または String 型の値は Date に変換されます。HAVING 節フィールドが Time 型の場合、TimeStamp 型または String 型の値は Time に変換されます。

以下の 2 つの例は、同じ等値比較を実行し、同じデータを返します。GROUP BY フィールドは、成功した等値比較ごとに 1 つのレコードのみを返すことを指定しています。DOB フィールドは Date データ型です。

### SQL

```
SELECT Name, DOB FROM Sample.Person
GROUP BY DOB
HAVING DOB IN ( {d '1951-02-02'}, {d '1987-02-28'} )
```

### SQL

```
SELECT Name, DOB FROM Sample.Person
GROUP BY DOB
HAVING DOB IN ( {ts '1951-02-02 02:37:00'}, {ts '1987-02-28 16:58:10'} )
```

詳細は、“[日付/時刻文](#)”を参照してください。

%INLIST 述語は、リスト構造の要素に対して等値比較を実行するために使用できます。%INLIST は EXACT 照合を使用します。そのため、既定では、%INLIST の文字列比較では大文字と小文字が区別されます。リスト構造の詳細は、SQL の [\\$LIST](#) 関数を参照してください。

以下の例では、%INLIST を使用して、FavoriteColors リスト・フィールドの要素に文字列値を一致させます。

### SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING 'Red' %INLIST FavoriteColors
```

ここでは、FavoriteColors に “Red” 要素が含まれるすべてのレコードが返されます。

以下の例では、Home\_State 列の値を northne (New England 北部の州) リストの要素に一致させます。

### SQL

```
SELECT Name, Home_State
FROM Sample.Person
HAVING Home_State %INLIST $LISTBUILD( "VT", "NH", "ME" )
```

サブクエリで IN または %INLIST を使用し、列の値 (あるいは他の式) がサブクエリの行の値と等しいかどうかをテストできます。以下はその例です。

## SQL

```
SELECT Name,Home_State FROM Sample.Person
HAVING Name IN
  (SELECT Name FROM Sample.Employee
   HAVING Salary < 50000)
```

サブクエリは、SELECT リスト内に必ず 1 項目を持ちます。

詳細は、“IN” および “%INLIST” を参照してください。

## %STARTSWITH 述語

InterSystems IRIS %STARTSWITH 比較演算子により、文字列や数字の先頭文字列との部分的マッチングを実行できます。以下の例は、%STARTSWITH を使用します。まず年齢で選択され、その中の “S” で始まる Name のレコードが返されます。

## SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age > 30
HAVING Name %STARTSWITH 'S'
ORDER BY Name
```

%STARTSWITH の比較では、他の文字列フィールドの比較と同様に大文字と小文字が区別されません。詳細は、“%STARTSWITH” を参照してください。

## 包含関係演算子 (I)

包含関係演算子は開始ブラケット記号 (I) です。これを使用して、部分文字列 (文字列または数値) とフィールドの値の任意の部分とのマッチングができます。比較は常に大文字と小文字が区別されます。以下の例は、HAVING 節で包含関係演算子を使用して、Home\_State の値に “K” を含むレコードを選択し、これらの州で %AFTERHAVING カウントを実行します。

## SQL

```
SELECT Home_State,COUNT(Home_State) AS States,
       COUNT(Home_State %AFTERHAVING) AS KStates
FROM Sample.Person
HAVING Home_State [ 'K'
```

## FOR SOME 述語

HAVING 節の FOR SOME 述語は、1 つ以上のフィールド値の条件テストに基づいて結果セットを返すかどうかを指定します。この述語の構文は以下のとおりです。

```
FOR SOME (table[AS t-alias]) (fieldcondition)
```

FOR SOME は、fieldcondition が True に評価される必要があるということを指定します。指定された条件に 1 つ以上のフィールド値が一致する必要があります。table には、単一のテーブル、またはコンマ区切りのテーブルのリストを指定可能であり、オプションでテーブル・エイリアスを指定できます。fieldcondition には、指定された table 内の 1 つまたは複数のフィールドのために 1 つまたは複数の条件を指定します。table 引数と fieldcondition 引数は、どちらも括弧で区切る必要があります。

以下の例は、FOR SOME 述語の使用法を示しています。

## SQL

```
SELECT Name, Age
FROM Sample.Person
HAVING FOR SOME (Sample.Person) (Age > 20)
ORDER BY Age
```

前述の例では、少なくとも 1 つのフィールドに 20 より大きい Age 値が含まれる場合、すべてのレコードが返されます。それ以外の場合、レコードは返されません。

詳細は、“[FOR SOME](#)” を参照してください。

## NULL 述語

定義されていない値を検出します。すべての NULL 値またはすべての NULL でない値を検出できます。

## SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING FavoriteColors IS NULL
```

## SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

GROUP BY 節を使用すると、指定されたフィールドの NULL でない値ごとに 1 つのレコードを返すことができます。

## SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
GROUP BY FavoriteColors
HAVING FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

詳細は、“[NULL](#)” を参照してください。

## EXISTS 述語

サブクエリが空のセットを評価するかどうかをテストするために、サブクエリを使用して処理します。

## SQL

```
SELECT t1.disease FROM illness_tab t1 WHERE EXISTS
  (SELECT t2.disease FROM disease_registry t2
   WHERE t1.disease = t2.disease
   HAVING COUNT(t2.disease) > 100)
```

詳細は、“[EXISTS](#)” を参照してください。

## LIKE、%MATCHES、および %PATTERN 述語

これらの 3 つの述語を使用してパターン・マッチングを実行できます。

- **LIKE** では、リテラルとワイルドカードを使用してパターン・マッチを実行できます。リテラル文字の既知の部分文字列、または既知のシーケンス内にいくつかの既知の部分文字列を含むデータ値を返す場合は、LIKE を使用します。LIKE は、大文字と小文字の比較に、そのターゲットの照合を使用します。
- **%MATCHES** では、リテラル、ワイルドカード、リスト、および範囲を使用してパターン・マッチを実行できます。リテラル文字の既知の部分文字列、指定された文字のリストまたは範囲にマッチする 1 つ以上のリテラル文字、または既知のシーケンス内にそうしたいくつかの部分文字列を含むデータ値を返す場合は、%MATCHES を使用します。%MATCHES は、大文字と小文字の比較に EXACT 照合を使用します。

- ・ [%PATTERN](#) を使用すると、文字タイプのパターンを指定できます。例えば、'1U4L1"/".A' (1 つの大文字、4 つの小文字、1 つのリテラル・コンマ、その後に任意数の大文字または小文字が続くパターン) などです。文字タイプの既知のシーケンスを含むデータ値を返す場合は、%PATTERN を使用します。%PATTERN はデータ値は重要でないけれども、その値の文字タイプ形式が重要である場合に役立ちます。%PATTERN では既知のリテラル文字も指定できます。リテラル比較では、常に大文字と小文字が区別される EXACT 照会を使用します。

文字列の最初の文字で比較を実行するには、[%STARTSWITH](#) 述語を使用します。

## 例

以下の例は、21 歳未満の人が 1 人以上いる各州の行を返します。行ごとに、その州の平均年齢、最低年齢、および最高年齢を返します。

### SQL

```
SELECT Home_State, MIN(Age) AS Youngest,  
       AVG(Age) AS AvgAge, MAX(Age) AS Oldest  
FROM Sample.Person  
GROUP BY Home_State  
HAVING Age < 21  
ORDER BY Youngest
```

以下の例は、21 歳未満の人が 1 人以上いる各州の行を返します。行ごとに、その州の平均年齢、最低年齢、および最高年齢を返します。また、%AFTERHAVING キーワードを使用して、州内の 21 歳未満の人の平均年齢 (AvgYouth) と 21 歳未満の人の最高年齢 (OldestYouth) を返します。

### SQL

```
SELECT Home_State, AVG(Age) AS AvgAge,  
       AVG(Age %AFTERHAVING) AS AvgYouth,  
       MIN(Age) AS Youngest, MAX(Age) AS Oldest,  
       MAX(Age %AFTERHAVING) AS OldestYouth  
FROM Sample.Person  
GROUP BY Home_State  
HAVING Age < 21  
ORDER BY AvgAge
```

%AFTERHAVING のその他の例は、個々の[集約関数](#)の説明を参照してください。

## 関連項目

- ・ [SELECT 文](#)
- ・ [WHERE 節](#)
- ・ [GROUP BY 節](#)
- ・ [述語の概要](#)
- ・ [データベースの問い合わせ](#)

# INTO (SQL)

選択された値をホスト変数に格納するように指定する SELECT 節です。

## 構文

```
INTO :hostvar1 [, :hostvar2]...
```

## 概要

INTO 節とホスト変数は、[埋め込み SQL](#)内でのみ使用されます。ホスト変数は、[ダイナミック SQL](#)では使用されません。ダイナミック SQL では、同様の出力変数機能が `%SQL.Statement` クラスに用意されています。ODBC、JDBC、またはダイナミック SQL を介して処理される SELECT クエリで INTO 節を指定すると、SQLCODE -422 エラーが発生します。

INTO 節は、[SELECT](#)、[DECLARE](#)、または [FETCH](#) 文で使用できます。INTO 節は、これら 3 つのどの文でもまったく同じです。このページの例はすべて、SELECT 文を対象としています。DECLARE および FETCH での使用法は、“[SQL カーソル](#)”を参照してください。

INTO 節は、SELECT の select-item リスト内で取得（または計算）された値を使用して、対応する出力ホスト変数を設定し、これらの返されたデータ値を ObjectScript で使用できるようにします。SELECT 内では、オプションの INTO 節は select-item リストと FROM 節の間に配置します。

**注意** 埋め込み SQL のコンパイル時に、出力ホスト変数は空文字列に初期化されます。これにより、実行時の `<UNDEFINED>` エラーを防止します。したがって、ホスト変数に意味のある値が含まれるのは、SQLCODE=0 の場合のみです。出力ホスト変数の値を使用する前に、必ず SQLCODEを確認してください。また、SQLCODE=100 の場合や、SQLCODE が負の数値の場合は、これらの変数の値を使用しないでください。

## ホスト変数

ホスト変数には単一の値のみを含めることができます。したがって、埋め込み SQL 内の SELECT では 1 行のデータのみが取得されます。既定では、この行はテーブルの最初の行になります。当然ながら、WHERE 条件を使用して対象行を制限することで、テーブルの他の行からデータを取得できます。

埋め込み SQL で、複数の行からデータを返すには、カーソルを宣言してから、連続する各行について FETCH を発行します。INTO 節のホスト変数は、DECLARE クエリ内または FETCH 内で指定できます。

INTO 節のホスト変数は、以下の 2 つのいずれかの方法（または両方の組み合わせ）で指定できます。

- ・ **ホスト変数リスト**：select-item ごとに 1 つの、ホスト変数のコンマ区切りリストで構成されます。
- ・ **ホスト変数配列**：単一の添え字付きホスト変数で構成されます。

含まれるプログラム内でのホスト変数の値の使用に関する重要な制限については、“[ホスト変数](#)”を参照してください。

**注釈** ホスト言語が変数のデータ型を宣言する場合、すべてのホスト変数は SELECT 文を呼び出す前にホスト言語で宣言される必要があります。取得したフィールド値のデータ型は、ホスト変数の宣言と一致する必要があります（ObjectScript では変数にデータ型を宣言しません）。

## ホスト変数リストの使用

INTO 節でホスト変数リストを指定する際は、以下の規則が適用されます。

- ・ INTO 節内のホスト変数の数は、select-item リスト内で指定されたフィールドの数と一致する必要があります。選択したフィールドの数とホスト変数の数が異なる場合、SQL は “cardinality mismatch” エラーを返します。
- ・ 選択したフィールドとホスト変数は、相対的な位置によって一致させます。したがって、これらの 2 つのリスト内の対応する項目は同じシーケンスで表示される必要があります。



- ・ リストされたホスト変数は、添え字なし変数と添え字付き変数のどのような組み合わせでもかまいません。
- ・ リストされたホスト変数は、集約値 (カウント値、合計値、平均値など) または関数値を返すことができます。
- ・ リストされたホスト変数は、%CLASSNAME と %TABLENAME の値を返すことができます。
- ・ リストされたホスト変数は、複数のテーブルを扱う SELECT からフィールド値を返すことも、FROM 節がない SELECT から値を返すこともできます。

以下の例では、4 つのフィールドを選択して、4 つのホスト変数のリストに格納します。この例のホスト変数は添え字付きです。

### ObjectScript

```
&sql(SELECT %ID,Home_City,Name,SSN
      INTO :mydata(1),:mydata(2),:mydata(3),:mydata(4)
      FROM Sample.Person
      WHERE Home_State='MA' )
IF SQLCODE=0 {
  FOR i=1:1:15 {
    IF $DATA(mydata(i)) {
      WRITE "field ",i," = ",mydata(i),! }
    }
  }
ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

さらなる例については、下記の [“ホスト変数リストの例”](#) を参照してください。

## ホスト変数配列の使用

ホスト変数配列では、単一の添え字付き変数に選択されたすべてのフィールド値を格納します。この配列には、select-item リスト内のフィールドの順序ではなく、テーブル内のフィールド定義の順序に従って値が格納されます。

INTO 節でホスト変数配列を使用する際は、以下の規則が適用されます。

- ・ select-item リストで指定されたフィールドは、選択されて、単一のホスト変数の添え字に格納されます。したがって、select-item リストの項目数をホスト変数の数と一致させる必要はありません。
- ・ ホスト変数の添え字には、テーブル定義内の対応するフィールド位置が割り当てられます。例えば、テーブル定義で定義された 6 番目のフィールドは mydata(6) に対応しています。指定された select-item に対応していない添え字はすべて、未定義のままになります。select-item 内の項目の順序は、添え字にどのように値が割り当てられるかに影響を与えません。
- ・ ホスト変数配列は、単一のテーブルからのフィールド値のみを返すことができます。
- ・ ホスト変数配列は、フィールド値のみを返すことができます。ホスト変数配列は、集約値 (カウント値、合計値、平均値など)、関数値、および %CLASSNAME や %TABLENAME の値を返すことはできません。(これらの値を返すには、ホスト変数リスト項目とホスト変数配列を組み合わせたホスト変数引数を指定します。)

以下の例では、4 つのフィールドを選択して、単一のホスト変数配列に格納します。

### ObjectScript

```
&sql(SELECT %ID,Home_City,Name,SSN
      INTO :mydata()
      FROM Sample.Person
      WHERE Home_State='MA' )
IF SQLCODE=0 {
  FOR i=0:1:15 {
    IF $DATA(mydata(i)) {
      WRITE "field ",i," = ",mydata(i),! }
    }
  }
ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

さらなる例については、下記の [“ホスト変数配列の例”](#) を参照してください。

詳細は、[“添え字付き配列としてのホスト変数”](#) を参照してください。

## 引数

### :hostvar1

ホスト言語で宣言された**出力ホスト変数**です。INTO 節内で指定すると、変数名の前にコロンの(:)が付きます。ホスト変数は、ローカル変数(添え字なしでも添え字付きでも可)か、オブジェクト・プロパティのいずれかです。複数の変数は、コンマ区切りリスト、単一の添え字付き配列変数、またはコンマ区切りリストと単一の添え字付き配列変数の組み合わせとして指定できます。

## フィールド値を返すホスト変数

以下の埋め込み SQL の例では、テーブルの最初のレコードから 3 つのフィールドを選択し(埋め込み SQL は常に単一のレコードを取得します)、INTO を使用して 3 つの対応する添え字なしホスト変数を設定します。これらの変数は ObjectScript WRITE コマンドによって使用されます。埋め込み SQL から戻った直ちに、プログラム内で SQLCODE 変数をテストすることをお勧めします。SQLCODE が 0 でない場合、出力ホスト変数の値は空文字列に初期化されます。

### ObjectScript

```
WRITE !,"Going to get the first record"
&sql(SELECT Home_State, Name, Age
      INTO :state, :name, :age
      FROM Sample.Person)
IF SQLCODE=0 {
  WRITE !,"  Name=",name
  WRITE !,"  Age=",age
  WRITE !," Home State=",state }
ELSE {
  WRITE !,"SQL error ",SQLCODE }
```

以下の埋め込み SQL の例では、2 つのテーブルの結合によって得られた 1 つの行からフィールド値を返します。複数のテーブルからフィールドを返す場合は、ホスト変数リストを使用する必要があります。

### ObjectScript

```
&sql(SELECT P.Name,E.Title,E.Name,P.%TABLENAME,E.%TABLENAME
      INTO :name(1),:title,:name(2),:ptname,:etname
      FROM Sample.Person AS P LEFT JOIN
           Sample.Employee AS E ON E.Name %STARTSWITH 'B'
      WHERE P.Name %STARTSWITH 'A')
IF SQLCODE=0 {
  WRITE ptname," = ",name(1),!
  WRITE etname," = ",title,!
  WRITE etname," = ",name(2) }
ELSE {
  WRITE !,"SQL error ",SQLCODE }
```

入力および出力のホスト変数の値の使用に関する制限については、“**ホスト変数**”を参照してください。

## リテラル値および集約値を返すホスト変数

出力ホスト変数は SQLCODE=0 の場合にのみ有効であるため、SQLCODE=100 (クエリがテーブル・データを返しませんが)を発行するクエリの結果を使用しないようにすることが重要です。SQLCODE=100 の場合、すべての出力ホスト変数は既定で空文字列に設定されます。これには、返されるリテラルと COUNT 集約も含まれます。

以下の埋め込み SQL の例では、ホスト変数 (today) を SELECT 文に渡し、SELECT 文での計算によって INTO 節の変数値 (:tomorrow) が得られます。このホスト変数値は、包含するプログラムに渡されます。このクエリはテーブルのフィールドを参照しないため、FROM 節が指定されていません。FROM 節が指定されていない埋め込み SQL クエリは、SQLCODE=100 を発行できません。FROM 節が指定された埋め込み SQL クエリは、SQLCODE=100 を発行できます。SQLCODE=100 の場合、テーブルのフィールド値ではない :tomorrow など、すべての出力ホスト変数が既定の NULL 文字列値に定義されます。

## ObjectScript

```

SET today=$HOROLOG
&sql(SELECT :today+1
      INTO :tomorrow )
IF SQLCODE=0 {
    WRITE !,"Tomorrow is: ",$ZDATE(tomorrow) }
ELSE {
    WRITE !,"SQL error ",SQLCODE }

```

以下の埋め込み SQL の例は、集約値を返します。この例では、COUNT 集約関数を使用してテーブルのレコードをカウントし、AVG を使用して Salary フィールド値の平均値を算出します。INTO 節は、これらの値を 2 つの添え字付きホスト変数として ObjectScript に返します。

どちらの select-items も集約なので、指定されたテーブルにデータが含まれていなくても、このプログラムは常に SQLCODE=0 を発行します。この場合、COUNT(\*)=0 と AVG(Salary) が既定の空文字列になります。

## ObjectScript

```

WRITE !,"Counting the records"
&sql(SELECT COUNT(*),AVG(Salary)
      INTO :agg(1),:agg(2)
      FROM Sample.Employee)
IF SQLCODE=0 {
    WRITE !,"Total Employee records= ",agg(1)
    WRITE !,"Average Employee salary= ",agg(2) }
ELSEIF SQLCODE=100 {
    WRITE !,"Total Employee records= ",agg(1) }
ELSE {
    WRITE !,"SQL error ",SQLCODE }

```

以下の埋め込み SQL の例は前の例と同じですが、以下の例はフィールド値も返す点が異なります。select-items にフィールド値が含まれるので、指定されたテーブルにデータが含まれない場合、このプログラムは SQLCODE=100 を発行できます。この例では、SQLCODE=100 の場合、COUNT(\*) は、0 ではなく既定の空文字列になります。

## ObjectScript

```

WRITE !,"Counting the records"
&sql(SELECT COUNT(*),AVG(Salary),Salary
      INTO :agg(1),:agg(2),:pay
      FROM Sample.Employee)
IF SQLCODE=0 {
    WRITE !,"Total Employee records= ",agg(1)
    WRITE !,"Average Employee salary= ",agg(2)
    WRITE !,"Sample Employee salary=",pay }
ELSE {
    WRITE !,"SQL error ",SQLCODE }

```

入力および出力のホスト変数の値の使用に関する制限については、“[ホスト変数](#)”を参照してください。

## ホスト変数配列

以下の 2 つの埋め込み SQL の例では、ホスト変数配列を使用して、単一の行から非公開ではないデータ・フィールド値を返します。これらの例では、select-item リストで %ID が指定されています。これは、既定では SELECT \* は RowID を返さないためです (ただし Sample.Person については RowID を返します)。RowID は常にフィールド 1 です。Sample.Person 内では、フィールド 4 および 9 は NULL を取ることができ、フィールド 5 はデータ・フィールドではなく (このフィールドは Sample.Address を参照します)、フィールド 10 は非公開です。

1 つ目の例では、指定された数のフィールドを返します (firstflds)。非公開のフィールドと非データ・フィールドは表示されませんが、この数に含まれます。多くのフィールドがあるテーブルから行を返す場合は、firstflds を使用することが適切です。この例は、親参照であるフィールド 0 を返すことができます。Sample.Person は子テーブルではないため、tfls(0) は未定義です。

## ObjectScript

```
&sql(SELECT *,%ID INTO :tflds()
      FROM Sample.Person )
IF SQLCODE=0 {
  SET firsttflds=14
  FOR i=0:1:firsttflds {
    IF $DATA(tflds(i)) {
      WRITE "field ",i," = ",tflds(i),! }
    } }
ELSE {WRITE "SQLCODE error=",SQLCODE,! }
```

2 つ目の例では、Sample.Person 内の非公開ではないすべてのデータ・フィールドを返します。Sample.Person 内で tflds(0) が未定義で、〈UNDEFINED〉エラーが発生するため、この例では、親参照であるフィールド 0 を返すを試みません。

## ObjectScript

```
&sql(SELECT *,%ID INTO :tflds()
      FROM Sample.Person )
IF SQLCODE=0 {
  SET x=1
  WHILE x '=' {
    WRITE "field ",x," = ",tflds(x),!
    SET x=$ORDER(tflds(x)) }
}
ELSE { WRITE "SQLCODE error=",SQLCODE,! }
```

以下の埋め込み SQL の例では、コンマ区切りのホスト変数リスト (非フィールド値用) とホスト変数配列 (フィールド値用) を組み合わせます。

## ObjectScript

```
&sql(SELECT %TABLENAME,Name,Age,AVG(Age)
      INTO :tname,:tflds(),:ageavg
      FROM Sample.Person
      WHERE Age > 50 )
IF SQLCODE=0 {
  WRITE "Table name is = ",tname,!
  FOR i=0:1:25 {
    IF $DATA(tflds(i)) {
      WRITE "field ",i," = ",tflds(i),! }
    }
  WRITE "Average age is = ",ageavg,! }
ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

## 関連項目

- ・ [SELECT、DECLARE、FETCH](#) 文
- ・ [VALUES](#) 節
- ・ [ホスト変数](#)
- ・ ObjectScript : [SET](#) コマンド

## ORDER BY (SQL)

結果セットでの行のソートを指定する SELECT 節です。

### 構文

```
ORDER BY orderItem
ORDER BY orderItem [ASC | DESC]
ORDER BY orderItem [ASC | DESC], orderItem2 [ASC | DESC]
```

### 概要

ORDER BY は、指定された 1 つ以上の順序付け項目（通常は列）で、クエリの結果セットの行をソートします。ORDER BY は、[SELECT](#) 文内の最後の節として、FROM、WHERE、GROUP By、および HAVING 節の後に指定されます。以下に例を示します。

#### SQL

```
SELECT Name, AVG(Age) AS AvgAge, Home_State
FROM Sample.Person
GROUP BY Home_State
ORDER BY AvgAge
```

- ORDER BY *orderItem* は、列などの指定された順序付け項目の値で、クエリの結果セットの行をソートします。行は、昇順に返されます。

以下の文は、照会された行を Home\_State 列を基準に昇順でソートして返します。

#### SQL

```
SELECT Name, Age, Home_State
FROM Sample.Person
ORDER BY Home_State
```

例：列名でのソート

- ORDER BY *orderItem* [ASC | DESC] は、値を昇順 (ASC) あるいは降順 (DESC) のいずれかでソートします。

以下の文は、照会された行を Home\_State 列を基準に降順でソートして返します。

#### SQL

```
SELECT Name, Age, Home_State
FROM Sample.Person
ORDER BY Home_State DESC
```

例：ORDER BY での TOP 節の使用

- ORDER BY *orderItem* [ASC | DESC], *orderItem2* [ASC | DESC] は、1 つ以上の順序付け項目で、値を順番にソートします。

以下の文では、照会された行を、最初に Home\_State 列を基準に昇順でソートし、次に Age 列を基準に降順でソートして返します。

#### SQL

```
SELECT Name, Age, Home_State
FROM Sample.Person
ORDER BY Home_State, Age DESC
```

例：

- 列名でのソート

- 列のエイリアスでのソート
- 列番号でのソート

この ORDER BY 節は、SELECT リスト内の [ウィンドウ関数](#) が実行された後 (ウィンドウ関数自体の ORDER BY 節を含む) に適用されます。このため、ウィンドウ関数から返される値は、SELECT クエリの ORDER BY 節の影響を受けません。

ORDER BY 節を省略すると、返される行の順序は指定されず、文の実行ごとに異なる可能性があります。

ORDER BY は、現在の [選択モード](#) 設定に関係なく、論理 (内部ストレージ) データ値で行をソートします。ORDER BY で行をソートする方法の詳細は、["照合での ORDER BY"](#) を参照してください。

## 引数

### orderItem

クエリ結果セットをソートする順序を指定する項目、または項目のコンマ区切りリスト。orderItem で、以下のいずれかを組み合わせとして項目を指定することができます。

- ・ テーブル内の列の名前。列を SELECT リストで指定する必要はありません。列名では、大文字と小文字は区別されません。テーブルにない列名を指定すると、SQLCODE -29 エラーが生成されます。
- ・ テーブル内の列のエイリアス。エイリアスは SELECT リストで指定する必要があります。列のエイリアスでは、大文字と小文字は区別されません。
- ・ 符号なしの数値リテラルとして指定される、テーブル内の列の数。この数は、SELECT リストで定義された列の順序に基づきます。SELECT リストの列に対応しない列番号を指定すると、SQLCODE -5 エラーが発生します。

orderItem の最初の文字が数字の場合、InterSystems IRIS® は列番号を指定していると思なします。整数のトランケーション・ルールは、整数以外の値を整数に解決するために適用されます。例えば、1.99 は 1 に解決されます。最初の文字が数字以外の場合、orderItem は列名または列のエイリアスと思なされます。

列番号を変数または式の結果として指定することはできません。列番号を括弧で囲むことはできません。

- ・ テーブル内の列で評価される式 (ORDER BY LENGTH(Name) など)。
- ・ [ウィンドウ関数](#) (ORDER BY ROW\_NUMBER() OVER (PARTITION BY State) など)。
- ・ [集約関数](#) (SELECT リストでも指定されている場合)。集約関数が ORDER BY 節のみで指定された場合、SQLCODE -73 エラーが生成されます。

いくつかの例外はありますが、orderItem は [リテラル](#) として指定する必要があります。列の名前を文字列として指定する変数またはその他の式を使用することはできません。orderItem が有効な識別子 (列の名前または列のエイリアス)、または符号なし整数 (列番号) として解釈されない場合、その orderItem は無視され、ORDER BY の実行はコンマ区切りリストで次の orderItem に進みます。orderItem 値が無視される例としては、以下のようなものがあります。

- ・ ダイナミック SQL ? 入力パラメータ
- ・ 埋め込み SQL :var ホスト変数
- ・ サブクエリ
- ・ 数値に解決される式
- ・ 符号付きの数字
- ・ 括弧で囲まれた数字

orderItem プロパティが非常に長い文字列であるか、複数の長い orderItems で ORDER BY を使用しようとする、InterSystems SQL はエラーを生成します。これを回避するには、関連するフィールドで [TRUNCATE 照合](#) を定義します。一般には、128 文字で切り捨てると安全です。

## 例

### 列名でのソート

以下の文は、列名でソートします。

#### SQL

```
SELECT Name,Home_State,DOB
FROM Sample.Person
ORDER BY Home_State,Name
```

ソート列が SELECT リストにあるかどうかに関係なく、列の名前でソートできます。例えば、この文は、SELECT リストに Home\_State 列が含まれていなくても、前の文と同じ順序で同じ行を返します。

#### SQL

```
SELECT Name,DOB
FROM Sample.Person
ORDER BY Home_State,Name
```

RowID がプライベートで、SELECT リストに示されていない場合でも、RowID 値でソートできます。[%ID 疑似列名](#)を、実際の RowID 名ではなく、orderItem として指定します。以下に例を示します。

#### SQL

```
SELECT Name,DOB
FROM Sample.Person
ORDER BY %ID
```

ORDER BY 節は、orderItem の一部として、テーブル名または[テーブル・エイリアス](#)を指定できます。

#### SQL

```
SELECT P.Name AS People,E.Name As Employees
FROM Sample.Person AS P,Sample.Employee AS E
ORDER BY P.Name
```

ORDER BY 節では、[矢印構文](#) (→) 演算子を使用して、ベース・テーブルではないテーブル内の列を指定できます。

#### SQL

```
SELECT Name,Company->Name AS CompName
FROM Sample.Employee ORDER BY Company->Name,Name
```

### 列のエイリアスでのソート

以下の文は、列のエイリアスでソートします。

#### SQL

```
SELECT Name,Home_State AS HS,DOB
FROM Sample.Person
ORDER BY HS,Name
```

この文は、エイリアスを持つ SELECT リスト内の式でソートします。

#### SQL

```
SELECT Name,Age,$PIECE(AVG(Age)-Age, '.',1) AS AgeDev
FROM Sample.Employee ORDER BY AgeDev,Name
```



## 列番号でのソート

以下の文は、列番号、つまり SELECT リストで指定されている取得された列の数値順でソートします。Home\_State (列 2) でソートしてから、Name (列 1) でソートします。

### SQL

```
SELECT Name,Home_State,DOB
FROM Sample.Person
ORDER BY 2,1
```

この文は、列番号を使用して SELECT リスト内の式でソートします。

### SQL

```
SELECT Name,Age,$PIECE(AVG(Age)-Age, '.',1)
FROM Sample.Employee ORDER BY 3,Name
```

SELECT \* 結果を列番号でソートすると、RowID がパブリックの場合、列 1 としてカウントされます。

### SQL

```
SELECT * FROM Sample.Person ORDER BY 1
```

## ORDER BY での TOP 節の使用

SELECT 文が ORDER BY および TOP 節を指定すると、返される“上位”行は、ORDER BY 節で指定された順序に基づきます。例えばこの文は、MyTable から、最も高い年齢の値を持つ 5 行を、年齢の高い方から順に返します。

### SQL

```
SELECT TOP 5 Name,Age FROM MyTable ORDER BY Age DESC
```

RowID でソートすると、TOP 節で選択される行が変わります。例えば、連続した RowID を持つ 100 行があるテーブルについて考えます。以下の文はそれぞれ行 1、2、3、4、5 および行 100、99、98、97、96 を返します。

### SQL

```
SELECT TOP 5 %ID FROM MyTable ORDER BY %ID
```

### SQL

```
SELECT TOP 5 %ID FROM MyTable ORDER BY %ID DESC
```

## リスト・データに基づくソート

以下の文は、InterSystems IRIS リスト・データを含む列でソートします。InterSystems IRIS リストは書式設定文字で始まるエンコードされた文字列であるため、この文では、\$LISTTOSTRING を使用して、リスト要素のエンコーディングではなく、実際の列値でソートしています。

### SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY $LISTTOSTRING(FavoriteColors)
```

## ホスト変数値に基づく項目のソート

CASE 式を使用して汎用のクエリを定義し、提供されるホスト変数の値に基づいてそのクエリを並べ替えることができます。例えば、以下の例では、var の値に応じて Name または Age で並べ替えることができます。

## SQL

```
SELECT Name, Age FROM Sample.Person ORDER BY
CASE WHEN :var1=1 then Name
      WHEN :var2=1 then Age END
```

この文は、2 つの CASE 式を指定します。真に評価された方の CASE で並べ替えられます。両方の CASE が真に評価された場合は、Country 順に並べ替えられ、Country 内で City 順に並べられます。

```
SELECT Country, City FROM Sample.Person ORDER BY
CASE WHEN :var1=1 then Country END,
      WHEN :var2=1 then City END
```

引数 ASC および DESC は、CASE END キーワードの後に指定します。

CASE 式内のフィールドは列名で指定する必要があります。列のエイリアスと列番号はこのコンテキストでは許可されていません。

## ダイナミック SQL と埋め込み SQL を使用した項目のソート

ダイナミック SQL は入力パラメータを使用して、リテラル値を ORDER BY 節に指定できます。入力パラメータを使用して、列名、列のエイリアス、列番号、照合キーワードを指定することはできません。以下のダイナミック SQL の例では、入力パラメータを使用して、名 (first name) で結果セットの行をソートします。

## ObjectScript

```
set myquery = 4
set myquery(1) = "SELECT TOP ? Name, Age,"
set myquery(2) = "CURRENT_DATE AS Today"
set myquery(3) = "FROM Sample.Person WHERE Age > ?"
set myquery(4) = "ORDER BY $PIECE(Name, ',', ?)"

set tStatement = ##class(%SQL.Statement).%New()
set qStatus = tStatement.%Prepare(.myquery)
if qStatus '= 1 {
    write "%Prepare failed:"
    do $System.Status.DisplayError(qStatus)
    quit }

set rset = tStatement.%Execute(10, 60, 2)
do rset.%Display()
write !, "%Display SQLCODE=", rset.%SQLCODE
```

以下のカーソルベースの埋め込み SQL の例は、同じ操作を実行しています。

## ObjectScript

```
SET topnum=10, agemin=60, firstname=2

&sql(DECLARE pCursor CURSOR FOR
      SELECT TOP :topnum Name, Age, CURRENT_DATE AS Today
      INTO :name, :years, :today FROM Sample.Person
      WHERE Age > :agemin
      ORDER BY $PIECE(Name, ',', :firstname) )

&sql(OPEN pCursor)
QUIT: (SQLCODE'=0)

FOR { &sql(FETCH pCursor)
      QUIT: SQLCODE
      WRITE "Name=", name, " Age=", years, " today=", today, !
    }

&sql(CLOSE pCursor)
```

## 制限事項

- SELECT クエリが ORDER BY 節を指定する場合、結果のデータは更新できません。したがって、後に続く DECLARE CURSOR FOR UPDATE 文を指定する場合、FOR UPDATE 節は無視され、カーソルは読み取り専用で宣言されます。

- ORDER BY が UNION に適用される場合、順序付け項目は数字か単純な列名にします。式は使用できません。列名を使用する場合、列の名前付けが UNION の最初の SELECT リストで行われたものとして結果の列を参照します。
- ORDER BY 節をサブクエリで使用する場合は、TOP 節と組み合わせる必要があります。これは TOP ALL 節になる場合もあります。例えば、以下のクエリは、ソートされたサブクエリで DISTINCT 節を使用しているため、有効ではありません。

```
SELECT Name FROM Sample.Person WHERE Name =
  (SELECT DISTINCT Name FROM Sample.Employee ORDER BY Title)
```

以下のクエリは、代わりにソートされたサブクエリで TOP ALL を使用しているため有効です。

## SQL

```
SELECT Name FROM Sample.Person WHERE Name =
  (SELECT TOP ALL Name FROM Sample.Employee ORDER BY Title)
```

- 400 文字を超える ORDER BY orderItem 値でクエリを実行すると、SQL -400 の深刻なエラーが発生する場合があります。この現象は、InterSystems IRIS システムの固定の制限である、エンコードされるグローバル参照の最大長の制限によるものです。この問題を防ぐためには、ORDER BY 節の基礎になるフィールドの照合設定でトランケーション・レングスを使用します。

例えば、このクエリの NarrativeSummary 列が 400 文字を超えているとします。

## SQL

```
SELECT LocationCity, NarrativeSummary FROM Aviation.Event
WHERE LocationCity %STARTSWITH 'Be'
ORDER BY NarrativeSummary
```

maxlen 切り捨て長を含む照合関数を追加すると、このクエリを正常に実行できます。

## SQL

```
SELECT LocationCity, NarrativeSummary FROM Aviation.Event
WHERE LocationCity %STARTSWITH 'Be'
ORDER BY %SQLUPPER(NarrativeSummary, 400)
```

## パフォーマンス

ORDER BY 節で使用されているリテラル値ごとに、異なるクエリ・キャッシュが生成されます。ORDER BY リテラルではリテラル置換は実行されません。これは、ORDER BY は整数を使用して列番号を指定できるためです。この整数が変わると、クエリはまったく異なる結果になります。

## 詳細

### 照合での ORDER BY

ソートは照合順で実行されます。既定では、文字列値の順序付けは、作成時に ORDER BY orderItem 列に対して指定された照合に基づいて行われます。InterSystems IRIS ネームスペースが SQLUPPER の既定の文字列照合を使用している場合、照合での ORDER BY では大文字と小文字を区別しません。

数値データ型フィールドの順序付けは、数値照合に基づいて行われます。式に対しては、既定照合は EXACT です。

照合関数を適用することで、列の既定照合をオーバーライドできます。例えば、ORDER BY %EXACT(Name) のように使います。列のエイリアスに照合関数を適用することはできません。これを実行しようすると、SQLCODE -29 エラーが生成されます。

既定の昇順照合シーケンスは、空文字列 (') よりも NULL を最小値とします。ORDER BY は、空白スペースのみで構成される文字列と空文字列を区別しません。

列に指定された照合が英数字である場合、先頭の数字は、整数順ではなく、文字照合順でソートされます。整数順で並べるには、[%PLUS](#) 照合関数を使用できますが、この関数は数値以外の文字すべてを 0 として扱います。

混合数値文字列を数値順に正しくソートするには、複数の ORDER BY orderItem を指定する必要があります。次の形式の Home\_Street 列について考えます。

```
Number StreetName StreetType
```

Number は、整数の番地です。StreetName と StreetType は、結合されて "Elm Street" などのような完全なストリート名を形成する文字列です。

以下の文では、住所を文字照合順にソートします。

## SQL

```
SELECT Name,Home_Street FROM Sample.Person
ORDER BY Home_Street
```

以下の文では、番地を整数順でソートし、ストリート名を文字照合順でソートしています。この文には式が含まれ、列のエイリアスや列番号ではなく、列名のみを操作します。

## SQL

```
SELECT Name,Home_Street FROM Sample.Person
ORDER BY $PIECE(%PLUS(Home_Street),' ',1),$PIECE(Home_Street,' ',2),$PIECE(Home_Street,' ',3)
```

## ASC 照合と DESC 照合

列識別子の前の、オプションの ASC (昇順) や DESC (降順) キーワードで指定されたように、各列のソートは昇順または降順の照合順で指定できます。ASC や DESC が指定されていない場合、ORDER BY は列を昇順でソートします。ASC または DESC キーワードは、ダイナミック SQL の ? 入力パラメータや埋め込み SQL の :var ホスト変数を使用して指定することはできません。

NULL は常に、ASC 順では最も低い値であり、DESC 順では最も高い値となります。

複数のコンマ区切りの ORDER BY 値は、並べ替え操作の階層を指定します。例えば以下の文では、SELECT 節リスト内の 3 番目にリストされた項目 (C) のデータ値を昇順でソートします。この順序内で、7 番目にリストされた項目 (J) の値を降順でソートします。この中で、1 番目にリストされた項目 (A) の値を昇順でソートします。

## SQL

```
SELECT A,B,C,M,E,X,J
FROM LetterTable
ORDER BY 3,7 DESC,1 ASC
```

ORDER BY 値のリスト内の重複した列は影響がありません。これは、2 番目のソートが最初のソート順序内であるためです。例えば、ORDER BY Name ASC, Name DESC では、Name 列を昇順でソートします。

## NLS 照合

既定以外の NLS (各国言語サポート) 照合を指定した場合は、すべての照合が並べられており、同一の国固有照合順序を使用していることを確認する必要があります。これには、テーブルで使用されるグローバルのみならず、IRISTEMP やプロセス・プライベート・グローバルなどの、一時ファイルでインデックスに使用されるグローバルも含まれます。詳細は、["SQL 照合と NLS 照合"](#) を参照してください。

## 関連項目

- [SELECT](#)

- ・ [UNION](#)
- ・ [TOP](#)
- ・ [照合](#)
- ・ [データベースの問い合わせ](#)
- ・ [SQLCODE エラー・メッセージ](#)

## TOP (SQL)

返す行数を指定する SELECT 節です。

### 構文

```
SELECT [DISTINCT clause] [TOP {[(int)] | ALL}]
      select-item{,select-item}
```

### 引数

引数	説明
int	返す行数を指定された整数値に制限します。int 引数には、正の整数、ダイナミック SQL の <a href="#">入力パラメータ (?)</a> 、または正の整数に解決される、埋め込み SQL の <a href="#">ホスト変数 (:var)</a> を指定できます。  ダイナミック SQL では、int 値はオプションで一重または二重の括弧で囲むことができます (推奨される構文は二重括弧です)。これらの括弧により、対応するクエリ・キャッシュの int 値のリテラル置換を抑制できます。
ALL	TOP ALL は、サブクエリまたは CREATE VIEW 文に使用したときのみ意味を持ちます。ORDER BY 節をこれらの状況で使用することがサポートされるように使用すると、CREATE VIEW で使用するクエリやサブクエリで TOP 節と <a href="#">ORDER BY</a> 節を組み合わせる必要がある要件が満たされます。TOP ALL では、返される行数に制限はありません。

### 説明

オプションの TOP 節は、SELECT キーワードとオプションの DISTINCT 節の後、最初の select-item の前に記述します。

TOP キーワードは、[ダイナミック SQL](#) と [カーソル・ベースの埋め込み SQL](#) で使用されます。非カーソルの埋め込み SQL では、TOP キーワードの意味のある使い方は TOP 0 のみとなります。他のすべての TOP int (int はゼロ以外の任意の整数) は有効ではありますが、意味はありません。なぜなら、非カーソルの埋め込み SQL 内の SELECT は常に、最大 1 行のデータを返すからです。

[SELECT](#) 文の TOP 節は、返される行数を int で指定された数に制限します。TOP 節が指定されない場合、既定では SELECT 条件を満たすすべての行を表示します。TOP 節が指定された場合、表示される行数は、int か、クエリ述語の要件を満たすすべての行のいずれか少ない方になります。ALL を指定した場合、SELECT は、クエリ述語の要件を満たすテーブル内のすべての行を返します。

クエリで [ORDER BY](#) 節が指定されていない場合、“上位” 行として返されるレコードは予想できません。ORDER BY 節を指定すれば、上位行は節で指定された順序に一致します。

[DISTINCT](#) 節は (指定された場合)、TOP の前に適用され、一意の値が (最大で) int 個返されるように指定します。

すべての行が送信されている場合、TOP は簡略化されます。このようにして、SQLCODE 100 を取得するまで選択する場合に、SQLCODE 100 を設定する [FETCH](#) では結果が即座に得られます。

ビューを介して、または [FROM](#) 節サブクエリを介してデータにアクセスする場合に、TOP 節ではなく (または TOP 節への追加として)、%vid ビュー ID を使用することで、返される行の数を制限できます。%vid の使用の詳細は、“[ビューの定義と使用](#)” を参照してください。

## TOP の int 値

int の数値には整数、数値文字列、ダイナミック SQL の[入力パラメータ \(?\)](#)、または整数値に解決される[入力ホスト変数 \(:var\)](#)を指定できます。

int 値では、返す行の行数を指定します。許可されている値は 0 と正の数です。int 値を算術式、フィールド名、サブクエリ列エイリアス、スカラ関数、または集約関数として指定することはできません。小数または数値文字列は、その整数値として解析されます。ゼロ (0) は有効な int 値です。TOP 0 を指定するとクエリは実行されますが、データは返されません。

TOP ALL は、クエリでキーワードとして指定する必要があります。ALL を ? 入力パラメータまたは :var ホスト変数値として指定することはできません。クエリ・パーサは、このように指定された文字列 “ALL” を値 0 の数値文字列として解釈します。

TOP int を数値文字列または整数として指定することができるため、TOP の[引数メタデータ](#)は、[xDBC データ型 4](#) (INTEGER) ではなく 12 (VARCHAR) として返されます。

int の数値には整数、数値文字列、ダイナミック SQL の[入力パラメータ \(?\)](#)、または整数値に解決される[入力ホスト変数 \(:var\)](#)を指定できます。

int 値では、返す行の行数を指定します。許可されている値は 0 と正の数です。int 値を算術式、フィールド名、サブクエリ列エイリアス、スカラ関数、または集約関数として指定することはできません。小数または数値文字列は、その整数値として解析されます。ゼロ (0) は有効な int 値です。TOP 0 を指定するとクエリは実行されますが、データは返されません。

TOP ALL は、クエリでキーワードとして指定する必要があります。ALL を ? 入力パラメータまたは :var ホスト変数値として指定することはできません。クエリ・パーサは、このように指定された文字列 “ALL” を値 0 の数値文字列として解釈します。

TOP int を数値文字列または整数として指定することができるため、TOP の[引数メタデータ](#)は、[データベース・ドライバのデータ型 4](#) (INTEGER) ではなく 12 (VARCHAR) として返されます。

## TOP とクエリ・キャッシュ

int 値は、括弧を付けて指定することも、付けないで指定することもできます。これらの括弧は、ダイナミック SQL クエリの[キャッシュ](#)方法に影響します (非カーソルの埋め込み SQL のクエリはキャッシュされません)。括弧なしの int 値は、クエリ・キャッシュでは ? パラメータ値に変換されます。これは、同じクエリを異なる TOP の int 値で呼び出すことは、毎回クエリを準備および最適化するのではなく、同じクエリ・キャッシュを呼び出すことを意味します。

括弧で囲むと、[リテラル置換を抑制](#)します。例えば、TOP ((7)) です。int が括弧で囲まれている場合、クエリ・キャッシュは特定の int 値を保持します。同じ TOP int 値のクエリを再呼び出しする場合、同じクエリ・キャッシュを使用します。別の TOP int 値でクエリを呼び出すと、SQL は新しいクエリを準備し、最適化し、キャッシュします。

TOP ALL は、? パラメータ変数としてキャッシュされません。ALL は、リテラルではなく、キーワードとして解析されます。そのため、TOP 7 と TOP ALL が指定された同じクエリでは、2 つの異なるクエリ・キャッシュが生成されます。

int 値は、括弧を付けて指定することも、付けないで指定することもできます。これらの括弧は、ダイナミック SQL クエリの[キャッシュ](#)方法に影響します (非カーソルの埋め込み SQL のクエリはキャッシュされません)。括弧なしの int 値は、クエリ・キャッシュでは ? パラメータ値に変換されます。これは、同じクエリを異なる TOP の int 値で呼び出すことは、毎回クエリを準備および最適化するのではなく、同じクエリ・キャッシュを呼び出すことを意味します。

括弧で囲むと、[リテラル置換を抑制](#)します。例えば、TOP ((7)) です。int が括弧で囲まれている場合、クエリ・キャッシュは特定の int 値を保持します。同じ TOP int 値のクエリを再呼び出しする場合、同じクエリ・キャッシュを使用します。別の TOP int 値でクエリを呼び出すと、SQL は新しいクエリを準備し、最適化し、キャッシュします。

TOP ALL は、? パラメータ変数としてキャッシュされません。ALL は、リテラルではなく、キーワードとして解析されます。そのため、TOP 7 と TOP ALL が指定された同じクエリでは、2 つの異なるクエリ・キャッシュが生成されます。



## TOP と ORDER BY

TOP は、通常、SELECT で [ORDER BY](#) 節と共に使用されます。既定の昇順の ORDER BY 照合シーケンスでは、NULL が最小 (“上位”) の値と見なされ、次に空文字列 (”) が続きます。

サブクエリの SELECT または [CREATE VIEW](#) の SELECT で ORDER BY 節を指定する場合は、TOP が必要です。このような場合、TOP int (返す行数を制限する) または TOP ALL を使用できます。

TOP ALL はサブクエリまたは CREATE VIEW 文でのみ使用できます。サブクエリで使用する ORDER BY 節または CREATE VIEW クエリで使用する ORDER BY 節は TOP 節と組み合わせる必要がありますが、TOP ALL を指定することでその要件を満たすことができます。TOP ALL では、返される行数に制限はありません。TOP ALL ...ORDER BY では、既定の SELECT の最適化は変更されません。ALL キーワードを括弧で囲むことはできません。

## TOP の最適化

既定では、SELECT はすべてのデータを返す時間が最速になるように最適化されます。TOP int 節と ORDER BY 節の両方を追加すると、最初の行を返す時間が最速になるように最適化されます (最適化を変更するには両方の節が必要です)。%SYS.PTools.StatsSQL クラスの TotalTimeToFirstRow プロパティを使用すると、最初の行を返すために必要な時間を返すことができます。

以下に特殊な最適化のケースを示します。

- ・ 返される行数を制限せずに TOP および ORDER BY の最適化方法を使用する必要がある場合があります (ページ単位で表示されるデータを返す場合など)。そのような場合、int 値を全行数より大きく指定して TOP 節を発行することができます。
- ・ 既定の SELECT の最適化を変更せずに、返される行数を制限してその順序を指定する必要がある場合があります。そのような場合、TOP 節、ORDER BY 節、および %NOTOPOPT キーワードを指定すると、すべてのデータを最速で返す最適化が保持されます。詳細は、“[FROM \(SQL\)](#)” を参照してください。

## TOP と集約および関数

集約関数またはスカラー関数は、単独の値のみを返します。クエリの select-item リストに、集約と関数のみが含まれている場合の TOP 節が適用されるかどうかについては、以下に示します。

- ・ select-item リストに COUNT(\*) や AVG(Age) などの集約関数が含まれており、フィールド参照が含まれていない場合、TOP の int 値や ORDER BY 節の存在に関係なく、複数の行は返されません。これらの節は検証されますが、無視されます。詳細は、以下の例を参照してください。

### SQL

```
SELECT TOP 5 AVG(Age),CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns 1 row */
```

### SQL

```
SELECT TOP 1 AVG(Age),CURRENT_TIMESTAMP(3) FROM Sample.Person ORDER BY Age
/* returns 1 row */
```

- ・ select-item リストに、1 つ以上のスカラー関数、式、リテラル (%TABLENAME など)、サブクエリ、またはホスト変数が含まれており、フィールド参照および集約は含まれていない場合、TOP 節が適用されます。詳細は、以下の例を参照してください。

### SQL

```
SELECT TOP 5 ROUND(678.987,2),CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns 5 identical rows */
```

実際に返される行数は、テーブル・フィールドを参照していない場合でも、テーブル内の行数が影響します。以下に例を示します。

## SQL

```
SELECT TOP 300 CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns either the number of rows in Sample.Person
or 300 rows, whichever is smaller */
```

クエリが述語の条件によって制限されている場合、テーブル・フィールドが select-item リストで参照されていない場合でも、返される行数はその条件によって制限されます。以下に例を示します。

## SQL

```
SELECT TOP 300 CURRENT_TIMESTAMP(3) FROM Sample.Person WHERE Home_State = 'MA'
/* returns either the number of rows in Sample.Person
where Home_State = 'MA'
or 300 rows, whichever is smaller */
```

- SELECT 文に FROM 節が含まれていない場合には、TOP 値に関係なく、返される行は最大でも 1 つです。以下に例を示します。

## SQL

```
SELECT TOP 5 ROUND(678.987,2),CURRENT_TIMESTAMP(3)
/* returns 1 row */
```

- DISTINCT 節は、TOP 節をさらに制限します。個別値の数が TOP 値より少ない場合は、個別値を持つ行のみが返されます。スカラー関数のみが参照される場合は、1 行のみが返されます。以下に例を示します。

## SQL

```
SELECT DISTINCT TOP 15 CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns 1 row */
```

- TOP 0 と指定すると、select-item リストの内容や、SELECT 文に FROM 節や DISTINCT 節が含まれているかどうかに関係なく、常に行はまったく返されません。

非カーソルの埋め込み SQL では、TOP 0 と指定したクエリでは行は返されず、SQLCODE=100 に設定されます。非カーソルの埋め込み SQL クエリで TOP 1 (または他の任意の TOP int 値) と指定すると、1 行が返され、SQLCODE=0 に設定されます。カーソル・ベースの埋め込み SQL では、フェッチ・ループが完了すると、TOP int 値に関係なく、常に SQLCODE=100 に設定されます。

## 例

以下のクエリは、データベースに格納される順番で Sample.Person から取得する最初の 20 行を返します。通常、このレコードの順序は予測できません。

## SQL

```
SELECT TOP 20 Home_State,Name FROM Sample.Person
```

以下のクエリは、Sample.Person から取得した最初の 20 個の異なる Home\_State 値を照合順の昇順で返します。

## SQL

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person ORDER BY Home_State
```

以下のクエリでは、非重複 FavoriteColor 値のうち最初の 40 個が返されます。“上位”の行は ORDER BY 節を反映し、Sample.Person 内のすべての行を照合順の降順 (DESC) に並べ替えます。FavoriteColors フィールドには NULL があることがわかっており、照合順の昇順では最初に表示されるので、既定の照合順の昇順より降順を使用します。

## SQL

```
SELECT DISTINCT TOP 40 FavoriteColors FROM Sample.Person
ORDER BY FavoriteColors DESC
```

また、上記の例では FavoriteColors はリスト・フィールドなので、照合順序には要素のバイト長も加味されます。そのため、6 文字の要素 (YELLOW、PURPLE、ORANGE) は共に照合され、その後に 5 文字の要素 (WHITE、GREENなど) がリストされます。

[ダイナミック SQL](#) では、int 値を入力パラメータ (“?”) として指定できます。以下の例では、%Execute メソッドで TOP ? の入力パラメータを 10 に設定しています。

### ObjectScript

```
SET myquery = "SELECT TOP ? Name, Age FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(10)
DO rset.%Display()
```

以下の[カーソル・ベースの埋め込み SQL](#) の例は、同じ操作を実行しています。

### ObjectScript

```
SET topnum=10
&sql(DECLARE pCursor CURSOR FOR
    SELECT TOP :topnum Name, Age INTO :name, :years FROM Sample.Person
)
&sql(OPEN pCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH pCursor)
    QUIT:SQLCODE
    WRITE "Name=", name, " Age=", years, !
}
&sql(CLOSE pCursor)
```

## 関連項目

- ・ [SELECT 文](#)
- ・ [DISTINCT 節](#)
- ・ [ORDER BY 節](#)
- ・ [データベースの問い合わせ](#)

# UNION (SQL)

2 つ以上の SELECT 文を組み合わせます。

## 構文

```
select-statement {UNION [ALL] [%PARALLEL] select-statement}
select-statement {UNION [ALL] [%PARALLEL] (query)}
(query) {UNION [ALL] [%PARALLEL] select-statement}
(query) {UNION [ALL] [%PARALLEL] (query)}
```

## 概要

UNION は 2 つ以上のクエリを単一のクエリに組み合わせ、1 つの結果にまとめられたデータを取り出します。UNION によって組み合わせられたクエリは、単一の [SELECT](#) 文で構成される単純なクエリになるか、または複合クエリになります。

SELECT 文間で UNION を使用可能にするには、それぞれの範囲で指定される列の数が一致している必要があります。列数が異なる SELECT を指定すると、SQLCODE -9 エラーが発生します。1 つの SELECT 内に NULL 列を指定し、列数を一致させるために別の SELECT 内のデータ列と組み合わせることができます。例を以下に示します。

### SQL

```
SELECT Name,Salary,BirthDate
FROM Sample.Employee
UNION ALL
SELECT Name,NULL,BirthDate
FROM Sample.Person
```

**注意** UNION で SELECT \* 構文を使用するには、テーブルに同じ列数が含まれている必要があります。そのため、列を追加したり削除してテーブル定義を今後変更すると、このソートの集合において予期しないエラーが発生する可能性があります。

InterSystems SQL は、UNION クエリのすべての範囲を自動的に評価して結果列のデータ型を決定し、次のように、**最も高い優先順位**を持つデータ型を返します：VARCHAR、DOUBLE、NUMERIC、BIGINT、INTEGER、SMALLINT、TINYINT。DATE など、その他のデータ型には優先順位は割り当てられません。例えば、他のコンテキストでは DATE データ型がより高い優先順位を持つ場合でも、以下のプログラムはデータ型 TINYINT を返します。

### SQL

```
SELECT MyTinyIntField FROM Table1
UNION ALL
SELECT MyDateField FROM Table2
```

上記以外のデータ型を返すには、以下の例に示すように、明示的な CAST 文を使用する必要があります。

### SQL

```
SELECT CAST(MyTinyInt AS DATE) FROM Table1
UNION ALL
SELECT MyDateField FROM Table2
```

UNION の範囲内の列の長さ、精度、またはスケールが異なる場合、結果列には最も大きい値が割り当てられます。

結果列名は、UNION の最初の範囲の列名 (または、列エイリアス) から取得されます。2 つの範囲で対応する列の名前が同じでない場合、すべての範囲で同じ列エイリアスを使用して、結果列を特定することをお勧めします。

任意の UNION 範囲の任意の列が NULL 可能な場合、結果列のメタデータは NULL 可能としてレポートされます。

UNION の結果の文字列フィールドには、対応する SELECT フィールドの[照合タイプ](#)が指定されますが、フィールドを照合しても一致しない場合には、EXACT 照合が割り当てられます。

## UNION と UNION ALL

一般の UNION では、重複する行 (すべての値が同一) が結果から削除されます。UNION ALL は、結果内の重複行を保持します。

精度の異なるフィールドは同じ値を持ちません。例えば、値 33 (データ型 NUMERIC(9)) と 33.00 (データ型 NUMERIC(9,2)) は同じであるとは見なされません。

照合の異なるフィールドは同じ値を持ちません。例えば、MyStringField と %SQLUPPER(MyStringField) は、両方の値がすべて大文字であっても、同じであるとは見なされません。

## TOP 節と ORDER BY 節

UNION 文は、結果を順序付ける [ORDER BY](#) 節で終結させることができます。この ORDER BY は文全体に適用されるため、サブクエリではなく、クエリの一番外側の部分で使用する必要があります。TOP 節と組み合わせる必要はありません。以下の例では、ORDER BY のこの使用法を示します。2 つの SELECT 文でデータを選択し、そのデータを UNION によって結合してから、ORDER BY で結果を順序付けます。

### SQL

```
SELECT Name,Home_Zip FROM Sample.Person
  WHERE Home_Zip %STARTSWITH 9
UNION
SELECT Name,Office_Zip FROM Sample.Employee
  WHERE Office_Zip %STARTSWITH 8
ORDER BY Home_Zip
```

SELECT リストの列に対応しない列番号を ORDER BY で使用すると、SQLCODE -5 エラーが発生します。SELECT リストの列に対応しない列名を ORDER BY で使用すると、SQLCODE -6 エラーが発生します。

UNION での SELECT 文のどちらか (または両方) に [ORDER BY](#) 節を含めることができますが、[TOP](#) 節と組み合わせる必要があります。この ORDER BY は、どの行を TOP 節によって選択するかを決定するために適用されます。以下の例では、ORDER BY のこの使用法を示します。2 つの各 SELECT 文で ORDER BY を使用して、それらの行を順序付けます。これにより、上位行として選択する行を決定します。選択されたデータを UNION によって結合してから、最後の ORDER BY で結果を順序付けます。

### SQL

```
SELECT TOP 5 Name,Home_Zip FROM Sample.Person
  WHERE Home_Zip %STARTSWITH 9
  ORDER BY Name
UNION
SELECT TOP 5 Name,Office_Zip FROM Sample.Employee
  WHERE Office_Zip %STARTSWITH 8
  ORDER BY Office_Zip
ORDER BY Home_Zip
```

TOP は、以下のように ORDER BY 節の位置に応じて、UNION の最初の SELECT に適用するか、UNION の結果に適用できます。

- UNION の結果に TOP...ORDER BY を適用します。UNION を FROM 節のサブクエリに記述している場合、TOP と ORDER BY はその UNION の結果に適用されます。以下はその例です。

### SQL

```
SELECT TOP 10 Name,Home_Zip
  FROM (SELECT Name,Home_Zip FROM Sample.Person
        WHERE Name %STARTSWITH 'A'
        UNION
        SELECT Name,Home_Zip FROM Sample.Person
        WHERE Home_Zip %STARTSWITH 8)
ORDER BY Home_Zip
```

- ・ TOP を最初の SELECT に適用し、ORDER BY を UNION の結果に適用します。以下はその例です。

## SQL

```
SELECT TOP 10 Name,Home_Zip
  FROM Sample.Person
  WHERE Name %STARTSWITH 'A'
UNION
SELECT Name,Home_Zip FROM Sample.Person
  WHERE Home_Zip %STARTSWITH 8
ORDER BY Home_Zip
```

## 囲みの括弧

UNION は、その SELECT 文の一方または両方、あるいは UNION 文全体を、オプションとして括弧で囲むことができます。1 組または複数組の括弧を指定できます。以下に示す括弧の使用は、すべて有効です。

```
(SELECT ...) UNION SELECT ...
(SELECT ...) UNION (SELECT ...)
((SELECT ...)) UNION ((SELECT ...))
(SELECT ... UNION SELECT ...)
((SELECT ...) UNION (SELECT ...))
```

使用する括弧の組ごとに、別個のクエリ・キャッシュが生成されます。

## UNION/OR 最適化

既定では、適切な場合に、SQL 自動最適化は UNION サブクエリを OR 条件に変換します。この UNION/OR 変換により、EXISTS やその他の下位の述語を InterSystems IRIS クエリ・オブティマイザ・インデックスで利用できる最上位の条件に移行できます。この既定の変換は、ほとんどの状況に適しています。ただし、状況によってはこの UNION/OR 変換によって大幅なオーバーヘッドが生じることがあります。%NOUNIONOROPT クエリ最適化オプションを指定すると、この FROM 節に関連付けられた WHERE 節内のすべての条件に対する自動の UNION/OR 変換が無効になります。そのため、複雑なクエリでは、この自動 UNION/OR 最適化を 1 つのサブクエリに対してのみ無効にして、その他のサブクエリには有効にすることができます。%NOUNIONOROPT の詳細は、“[FROM 節](#)”を参照してください。

サブクエリに関係する条件が UNION に適用される場合、各 UNION の末尾ではなくオペランド内に適用されます。これにより、サブクエリ最適化を各 UNION オペランドで適用できます。サブクエリ最適化オプションについては、“[FROM 節](#)”を参照してください。以下の例では、WHERE 節の条件は、UNION の結果ではなく、UNION 内の各サブクエリに適用されます。

## SQL

```
SELECT Name,Age FROM
  (SELECT Name,Age FROM Sample.Person
   UNION SELECT Name,Age FROM Sample.Employee)
WHERE Age IN (SELECT TOP 5 Age FROM Sample.Employee WHERE Age>55 ORDER BY Age)
```

## UNION ALL の集約の最適化

SQL による UNION ALL の自動最適化は、最上位の集約を UNION 範囲にプッシュします。これにより、%PARALLEL キーワードの有無に関係なく、パフォーマンスが大幅に向上します。以下に例を示します。

## SQL

```
SELECT COUNT(*) FROM (SELECT item1 FROM table1 UNION ALL SELECT item2 FROM table2)
```

これは、次のように最適化されます。

## SQL

```
SELECT SUM(y) FROM (SELECT COUNT(*) AS y FROM table1 UNION ALL SELECT COUNT(*) AS y FROM table2)
```



この最適化は、(COUNT だけでなく) 最上位のすべての集約関数に適用されます。これには、最上位の集約関数を複数持つクエリが含まれます。この最適化を適用するには、外側のクエリは WHERE や GROUP BY 節のない "1 行" のクエリである必要があります。これは %VID を参照することはできず、UNION ALL が FROM 節内の唯一のストリームである必要があります。集約は入れ子にすることはできません。また、使用されている集約関数では、%FOREACH() グループ化や DISTINCT は使用できません。

## 並列処理

%PARALLEL キーワードは、マルチプロセッサ・システムでの並列処理および分散処理をサポートします。これにより InterSystems IRIS は各クエリを同じマシン上の別個のプロセスに割り当て、UNION クエリに対して並列処理を実行します。場合によっては、この処理によりクエリが別のマシンに送信され、そこで処理されることもあります。これらの処理はパイプ経由で通信し、InterSystems IRIS はサブクエリの結果を保持する 1 つ以上の一時ファイルを作成します。メインの処理では、結果の行が結合され、最終結果が返されます。詳細は、UNION クエリの "[プラン表示](#)" を参照し、%PARALLEL キーワードを指定した場合と指定しない場合のプラン計画を比較してください。現在のシステム上のプロセッサ数を特定するには、%SYSTEM.Util.NumberOfCPUs() メソッドを使用します。

一般に、各行の生成に費やす手間がかかるほど、%PARALLEL はより有用になります。

%PARALLEL キーワードを指定すると、[自動の UNION から OR への最適化](#)が無効になります。

以下の例では、%PARALLEL キーワードの使用法を示します。

### SQL

```
SELECT Name FROM Sample.Employee WHERE Name %STARTSWITH 'A'
UNION %PARALLEL
SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

### SQL

```
SELECT Name FROM Sample.Employee WHERE Name %STARTSWITH 'A'
UNION ALL %PARALLEL
SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

%PARALLEL は、SELECT クエリとそのサブクエリで使用するためのものです。[INSERT](#) コマンド・サブクエリは %PARALLEL を使用できません。

UNION クエリによっては、%PARALLEL キーワードの追加が不適切な場合もあり、それによってエラーが発生することもあります。OUTER JOIN、相関フィールド、サブクエリを含む IN 述語条件、コレクション述語といった SQL 構造は、通常 UNION %PARALLEL の実行をサポートしません。UNION %PARALLEL は FOR SOME 述部ではサポートされますが、FOR SOME %ELEMENT コレクション述部ではサポートされません。UNION クエリで %PARALLEL を正常に使用できるかどうかを判断するには、それぞれの UNION 範囲を個別にテストします。[FROM %PARALLEL](#) キーワードを追加して、それぞれの範囲のクエリを別々にテストしてください。FROM %PARALLEL クエリの 1 つで並列処理のないクエリ・プランが生成された場合、その UNION クエリは %PARALLEL をサポートしないことになります。

## UNION ALL と集約関数

SQL 自動最適化では、UNION ALL [集約関数](#)を UNION 範囲サブクエリにプッシュします。SQL は各サブクエリの集約値を計算してから、結果を結合し、元の集約値を返します。例えば、以下のようになります。

### SQL

```
SELECT COUNT(Name) FROM (SELECT Name FROM Sample.Person
                           UNION ALL SELECT Name FROM Sample.Employee)
```

は、次のように最適化されます。



## SQL

```
SELECT SUM(y) FROM (SELECT COUNT(Name) AS y FROM Sample.Person
                    UNION ALL SELECT COUNT(Name) AS y FROM Sample.Employee)
```

この結果、パフォーマンスが大幅に向上する場合があります。この最適化は、**%PARALLEL** キーワード付き、またはなしで適用されます。この最適化は複数の集約関数に適用されます。

この最適化の変換は、以下の状況でのみ発生します。

- ・ 外側のクエリの FROM 節には UNION ALL 文のみが含まれる必要がある。
- ・ 外側のクエリに WHERE 節または GROUP BY 節を含めることはできない。
- ・ 外側のクエリに **%VID** (ビュー ID) フィールドを含めることはできない。
- ・ 集約関数に DISTINCT または **%FOREACH** キーワードを含めることはできない。
- ・ 集約関数を入れ子にすることはできない。

## 引数

### ALL

キーワードのリテラル (オプション)。指定した場合、重複するデータ値が返されます。省略した場合は、重複するデータ値が抑制されます。

### %PARALLEL

**%PARALLEL** キーワードを指定する引数 (オプション)。指定した場合、UNION のそれぞれの側は、別個のプロセスとして並行して実行されます。

### select-statement

データベースからデータを検索する **SELECT** 文。

### query

1 つまたは複数の SELECT 文を組み合わせるクエリ。

## 例

以下の例は、2 つのテーブルのそれぞれで見つかったすべての Name に対応する行を持つ結果を生成しています。Name が両方のテーブルで見つかった場合、行が 2 つ作成されます。Name が従業員の場合、State としての "オフィス" という用語と従業員の役職とを連結して、オフィスの場所をリストします。Name が人の場合、State としての "自宅" という用語と役職に対する <null> とを連結して、自宅の場所をリストします。ORDER BY 節は結果を処理し、行の組み合わせは Name を使って順番に並べられます。

## SQL

```
SELECT Name,Office_State||' office' AS State,Title
FROM Sample.Employee
UNION
SELECT Name,Home_State||' home',NULL
FROM Sample.Person
ORDER BY Name
```

以下の 2 つの例は、ALL キーワードの効果を示しています。最初の例では、UNION は一意の値のみを返します。2 番目の例では、UNION ALL は重複する値を含むすべての値を返します。

### SQL

```
SELECT Name
FROM Sample.Employee
WHERE Name %STARTSWITH 'A'
UNION
SELECT Name
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

### SQL

```
SELECT Name
FROM Sample.Employee
WHERE Name %STARTSWITH 'A'
UNION ALL
SELECT Name
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

## 関連項目

- ・ [SELECT](#)
- ・ [ORDER BY 節、TOP 節](#)
- ・ [CREATE QUERY、CREATE PROCEDURE](#)
- ・ [データベースの問い合わせ](#)
- ・ [SQLCODE エラー・メッセージ](#)

# VALUES (SQL)

フィールド内で使用するデータ値を指定する INSERT/UPDATE 節です。

## 構文

```
(field1{,fieldn})
VALUES (value1{,valuen})
```

## 概要

VALUES 節は [INSERT](#) 文、[UPDATE](#) 文、または [INSERT OR UPDATE](#) 文で使用され、フィールドに挿入するデータ値を指定します。通常は、以下のとおりです。

- INSERT クエリは、以下の構文を使用します。

```
INSERT INTO tablename (fieldname1,fieldname2,...)
VALUES (value1,value2,...)
```

- UPDATE クエリは、以下の構文を使用します。

```
UPDATE tablename (fieldname1,fieldname2,...)
VALUES (value1,value2,...)
```

VALUES 節の要素は、テーブル名の後で指定されているフィールドに連続して対応します。VALUES 節で 1 つの値だけが指定されている場合は、要素を括弧で囲む必要はありません。

以下の例は、単独の行を "Employee" テーブルに追加する INSERT 文を示しています。

### SQL

```
INSERT INTO Employee (Name,SocSec,Telephone)
VALUES("Boswell",333448888,"546-7989")
```

### SQL

```
INSERT INTO Employee (Name,SocSec,Telephone)
VALUES ('Boswell',333448888,'546-7989')
```

INSERT クエリと UPDATE クエリは、テーブル名の後にフィールド名のリストを明示的に指定せずに、VALUES 節を使用できます。テーブル名の後でフィールド名のリストを省略するには、クエリが次の 2 つの基準を満たしている必要があります。

- VALUES 節で指定されている値の数は、テーブル内のフィールドの数と同じです (ID フィールドは除く)。
- VALUES 節の値は、列 2 から始まるフィールドの内部列番号の順序でリストされます。列 1 は常にシステムによって作成された ID フィールド用として使用されるので、VALUES 節では指定されません。

例えば、以下にクエリがあります。

### SQL

```
INSERT INTO Sample.Person VALUES (5,'John')
```

以下に別のクエリがあります。

### SQL

```
INSERT INTO Sample.Person (Age,Name) VALUES (5,'John')
```

テーブル "Sample.Person" が 2 つのユーザ定義のフィールドを持っている場合、上と下のクエリは等しくなります。

この例では、値 5 が下位番号の列フィールドに割り当てられており、値 "John" がもう 1 つのフィールドに割り当てられています。

VALUES 節は、以下の埋め込み SQL の例のように配列の要素を指定できます。

### ObjectScript

```
&sql( UPDATE Person(Tel)
      VALUES :per('tel',)
      WHERE ID = :id )
```

また、UPDATE クエリは、指定されていない最後の部分文字列を持つ配列を参照できます。したがって、INSERT は配列要素の有無を使用して、値と既定値を新規に作成された行に割り当てます。UPDATE は配列要素の存在を使用して、対応するフィールドが更新されることを示します。例えば、以下のような 6 列のテーブルに対する配列があるとし

```
emp("profile",2)="Smith"
emp("profile",3)=2
emp("profile",3,1)="1441 Main St."
emp("profile",3,2)="Cableton, IL 60433"
emp("profile",5)=NULL
emp("profile",7)=25
emp("profile","next")="F"
```

列 1 は常に ID フィールド用のもので、ユーザ定義ではありません。挿入された "Employee" 行は列 2 の "Name" を "Smith" に設定し、列 3 の "Address" が 2 行の値を持つように設定し、列 4 の "Department" はここでは指定されていないので既定値に設定し、そして列 5 の "Location" は NULL に設定します。"Location" の既定値は、対応する配列要素が NULL 値で定義されているので、使用されません。配列要素 "7" と "next" は、"Employee" テーブルの列番号に対応しないので、クエリによって無視されます。ここでは、この配列を使用する UPDATE 文を示しています。

### ObjectScript

```
&sql(UPDATE Employee
      VALUES :emp('profile',)
      WHERE Employee = 379)
```

上記の定義と配列値により、この文は RowID = 379 である "Employee" 行の "Name" フィールド、"Address" フィールド、"Location" フィールドを更新します。

ただし、添え字を完全に省略すると SQLCODE -54 エラーが発生します。VALUES の後に (最後の添え字が省略された) 配列識別子が必要です。

例えば以下のように、複数の行を対象にした UPDATE クエリを持つ配列参照を使用することもできます。

### ObjectScript

```
&sql(UPDATE Employee
      VALUES :emp('profile',)
      WHERE Type = 'PART-TIME')
```

VALUES 節変数は、ドット構文を使用できません。したがって、以下の埋め込み SQL の例では正常に動作します。

```
SET sname = state.Name
&sql(INSERT INTO StateTbl VALUES :sname)
```

以下は不適切です。

```
&sql(INSERT INTO State VALUES :state.Name)
```

NULL と空白文字列値は異なります。詳細は "NULL" を参照してください。後方互換性では、古い既存データ内にあるすべての空白文字列 (') 値は NULL 値と見なされます。新規データでは、空白文字列は \$CHAR(0) としてデータ内に保存されます。SQL では、NULL は 'NULL' と記述されます。以下はその例です。

## SQL

```
INSERT INTO Sample.Person  
(SSN,Name,Home_City) VALUES ('123-45-6789','Doe,John',NULL)
```

SQL では、空白文字列は '' (2 つの一重引用符) と記述されます。以下はその例です。

## SQL

```
INSERT INTO Sample.Person  
(SSN,Name,Home_City) VALUES ('123-45-6789','Doe,John','')
```

ID フィールドに NULL 値は挿入できません。

## 引数

### field

フィールド名、またはコンマで区切られたフィールド名のリスト。

### value

値、あるいはコンマで区切られた値のリスト。各値は対応するフィールドに割り当てられています。

## 例

以下の例は、“Doe,John” のレコードを Sample.Person テーブルに挿入します。次に、このレコードを選択して、このレコードを削除します。2 番目の SELECT で削除を確認します。

## SQL

```
INSERT INTO Sample.Person (Name,SSN,Home_City) VALUES ("Doe,John","123-45-6789","Metropolis")  
SELECT Name,SSN,Home_City FROM Sample.Person WHERE Name ="Doe,John"  
DELETE FROM Sample.Person WHERE Name="Doe,John"  
SELECT Name,SSN FROM Sample.Person WHERE Name='Doe,John'
```

## 関連項目

- [INSERT](#)
- [INSERT OR UPDATE](#)
- [UPDATE](#)
- [SQLCODE エラー・メッセージ](#)

## WHERE (SQL)

1 つ以上の制限条件を指定する SELECT 節です。

### 構文

```
SELECT fields
FROM table
WHERE condition-expression
```

### 引数

引数	説明
condition-expression	どのデータ値が取得されるかを規定する 1 つまたは複数のブーリアン述語で構成される式。

### 概要

オプションの WHERE 節は、以下の目的のために使用できます。

- ・ どのデータ値が返されるかを制限する述語を指定するため。
- ・ 2 つのテーブル間の明示結合を指定するため。
- ・ ベース・テーブルと別のテーブルのフィールドの間の暗黙結合を指定するため。

WHERE 節は、1 つまたは複数の[述語](#)を指定し、[SELECT](#) クエリまたはサブクエリで検索されたデータを制限 (行をフィルタ削除) するために使用するのが最も一般的です。[UPDATE](#) コマンド、[DELETE](#) コマンド、または [INSERT](#) (あるいは [INSERT OR UPDATE](#)) コマンドの結果セット SELECT で WHERE 節を使用することもできます。

WHERE 節は、クエリ選択から特定の行を適格とするか、もしくは不適格とします。適格な行とは、condition-expression が True である行です。condition-expression は、1 つ以上の論理テスト (熟語) です。AND および OR 論理演算子で複数の述語をつなぐことができます。詳細と制約については、“[述語および論理演算子](#)”を参照してください。

述語に除算が含まれており、除数がゼロまたは NULL になりうる値がデータベースに存在する場合、評価の順序に依存してゼロによる除算を回避することはできません。その代わりに、[CASE](#) 文を使用してリスクを抑制します。

WHERE 節は、サブクエリを含む condition-expression を指定できます。サブクエリは括弧で囲む必要があります。

WHERE 節は、= (内部結合) シンボル結合演算子を使用して、2 つのテーブルの間の明示的な結合を指定できます。詳細は、“[JOIN](#)”を参照してください。

WHERE 節は、矢印構文 (->) 演算子を使用して、ベース・テーブルおよび別のテーブルのフィールドの間の明示的な結合を指定できます。詳細は、“[暗黙結合 \(矢印構文\)](#)”を参照してください。

### フィールドの指定

WHERE Age > 21 など、WHERE 節の最も簡潔な形式で、フィールドと値を比較する述語を指定します。有効なフィールド値としては、列名 (WHERE Age > 21) や、%ID、%TABLENAME、または %CLASSNAME や、列名を指定するスカラ関数 (WHERE ROUND(Age, -1)=60)、列名を指定する照合関数 (WHERE %SQLUPPER(Name) %STARTSWITH 'AB') があります。

列番号でフィールドを指定することはできません。

テーブルを再コンパイルすると、[RowID フィールド](#)の名前が変わる可能性があるため、WHERE 節では RowID を名前でも参照しないようにする必要があります (例: WHERE ID=22)。代わりに、%ID 疑似列名を使用して RowID を参照します (例: WHERE %ID=22)。

列エイリアスでフィールドを指定できません。これを試行すると、SQLCODE -29 エラーが生成されます。ただし、サブクエリを使用して列エイリアスを定義してから、このエイリアスを WHERE 節で使用できます。例えば以下ようになります。

## SQL

```
SELECT Interns FROM
  (SELECT Name AS Interns FROM Sample.Employee WHERE Age<21)
WHERE Interns %STARTSWITH 'A'
```

集約フィールドを指定できません。これを試行すると、SQLCODE -19 エラーが生成されます。ただし、サブクエリによって、集約関数値を WHERE 節で指定することはできます。例えば以下ようになります。

## SQL

```
SELECT Name, Age, AvgAge
FROM (SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person)
WHERE Age < AvgAge
ORDER BY Age
```

## 整数と文字列

整数データ型として定義されているフィールドを数値と比較する場合は、比較を実行する前に数値は**キャノニック形式**に変換されます。例えば、WHERE Age=007.00 は WHERE Age=7 として解析されます。この変換は、すべてのモードで発生します。

表示モードで、整数データ型として定義されているフィールドを文字列と比較する場合は、文字列は数値として解釈されます。例えば、空文字列('')は、数値以外の文字列と同様に、数値 0 と解析されます。この解析は、文字列を数値として処理するための ObjectScript 規則に従っています。例えば、WHERE Age='twenty' は WHERE Age=0、WHERE Age='20something' は WHERE Age=20 として解析されます。詳細は、“[数値としての文字列](#)”を参照してください。SQL は、この解析を表示モードでのみ実行します。論理モードまたは ODBC モードでは、整数を文字列値と比較すると NULL が返されます。

一重引用符を含む文字列のフィールドを比較するには、一重引用符を二重にします。例えば、WHERE Name %STARTSWITH 'O'' は、Obama でなく O'Neil および O'Connor を返します。

## 日付と時刻

InterSystems SQL では、日付と時刻の比較および保存には論理モードの内部表現が使用されています。日付と時刻は、論理モード、表示モード、または ODBC モードで返すことができます。例えば、1944 年 9 月 28 日は、論理モードでは 37891、表示モードでは 09/28/1944、ODBC モードでは 1944-09-28 と表現されます。condition-expression で日付または時刻を指定する場合に、SQL モードと日付形式または時刻形式が一致していなかったり、日付値または時刻値が無効であったりするとエラーが発生することがあります。

WHERE 節の condition-expression では、現在のモードに一致する日付形式または時刻形式を使用する必要があります。例えば、論理モードのときに、誕生日が 2005 年であるレコードを返すには、WHERE 節を WHERE DOB BETWEEN 59901 AND 60265 とします。表示モードの場合は、この WHERE 節を WHERE DOB BETWEEN '01/01/2005' AND '12/31/2005' とします。

condition-expression の日付形式または時刻形式が表示モードと一致していないと、エラーが発生します。

- 表示モードまたは ODBC モードで日付データを不適切な形式で指定すると、SQLCODE -146 エラーが生成されます。時刻データを不適切な形式で指定すると、SQLCODE -147 エラーが生成されます。
- 論理モードで日付データまたは時刻データを不適切な形式で指定すると、エラーは生成されませんが、データが何も返されないか、意図しないデータが返されます。これは、表示形式や ODBC 形式の日付または時刻が、論理モードでは日付値または時刻値として解析されないためです。WHERE DOB BETWEEN 37500 AND 38000 AND DOB <> '1944-09-28' という WHERE 節は、論理モードで実行すると意図しないデータを返します。ここでは、<> 述語で DOB=37891 (1944 年 9 月 28 日) の除外を指定していますが、実際にはこの日付を含む DOB 値の範囲が返されます。



無効な日付値または時刻値を指定した場合も、SQLCODE -146 または -147 エラーが返されます。無効な日付とは、表示モードや ODBC モードで指定はできても、InterSystems IRIS が同等の論理モード値に変換できない日付です。例えば、ODBC モードで `WHERE DOB > '1830-01-01'` を指定すると、1840 年 12 月 31 日より前の日付値は InterSystems IRIS で処理できないので SQLCODE -146 エラーが生成されます。また、ODBC モードで `WHERE DOB BETWEEN '2005-01-01' AND '2005-02-29'` と指定すると、2005 年はうるう年ではないので、この場合も SQLCODE -146 エラーが生成されます。

論理モードでは、表示モードや ODBC モードの値は日付値または時刻値として解析されないため、これらのモードの値は検証されません。したがって、論理モードで `WHERE DOB > '1830-01-01'` のような WHERE 節を指定しても、エラーは返されません。

## ストリーム・フィールド

ほとんどの場合、WHERE 節の述語でストリーム・フィールドを使用することはできません。これを使用すると、SQLCODE -313 エラーが発生します。ただし、以下のストリーム・フィールドの使用は WHERE 節で許可されています。

- ・ **ストリーム NULL テスト** : `streamfield IS NULL` 述語または `streamfield IS NOT NULL` 述語を指定できます。
- ・ ストリームの長さのテスト : WHERE 節の述語で `CHARACTER_LENGTH(streamfield)`、`CHAR_LENGTH(streamfield)`、または `DATALength(streamfield)` 関数を指定できます。
- ・ ストリーム部分文字列のテスト : WHERE 節の述語で `SUBSTRING(streamfield,start,length)` 関数を指定できます。

## リスト構造

InterSystems IRIS は、リスト構造のデータ型 `%List` (データ型クラス `%Library.List`) をサポートしています。これは圧縮バイナリ形式であり、InterSystems SQL で対応するネイティブなデータ型にマップしません。また、データ型 `VARBINARY` に対応しており、その `MAXLEN` の既定値は 32749 です。このため、**ダイナミック SQL** では、WHERE 節での比較に `%List` データを使用できません。詳細は、“**データ型**” を参照してください。

構造化されたリスト・データを参照するには、`%INLIST` 述語または `FOR SOME %ELEMENT` 述語を使用します。

`condition-expression` でリスト・フィールドのデータ値を使用するには、`%EXTERNAL` を使用して、リストの値と述語を比較できます。例えば、`FavoriteColors` リスト・フィールドの値が 1 つの要素 'Red' で構成されているレコードをすべて返すには、以下のようにします。

### SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors)='Red'
```

`%EXTERNAL` によってリストが `DISPLAY` 形式に変換されると、表示されるリスト項目は、空白スペースで区切られます。この“スペース”は実際には、`CHAR(13)` と `CHAR(10)` という 2 つの非表示文字です。`condition-expression` をリスト内の複数の要素に対して使用するには、これらの文字を指定する必要があります。例えば、`FavoriteColors` リスト・フィールドの値が 2 つの要素 'Orange' と 'Black' で (その順に) 構成されているレコードをすべて返すには、以下のようにします。

### SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors)='Orange' || CHAR(13) || CHAR(10) || 'Black'
```

## 変数

WHERE 節の述語は、以下を指定できます。

[%TABLENAME](#) または [%CLASSNAME](#) 疑似フィールド変数キーワード。[%TABLENAME](#) は、現在のテーブル名を返します。[%CLASSNAME](#) は、現在のテーブルに対応するクラスの名前を返します。クエリが複数のテーブルを参照する場合には、キーワードの接頭語としてテーブルのエイリアスを指定できます。例えば、[SUB\\_ACCESSIBLE\\_FILE](#) のように指定します。

以下の 1 つ以上の ObjectScript 特殊変数 (またはそれらの省略形) : [\\$HOROLOG](#)、[\\$JOB](#)、[\\$NAMESPACE](#)、[\\$TLEVEL](#)、[\\$USERNAME](#)、[\\$ZHOROLOG](#)、[\\$ZJOB](#)、[\\$ZNSPACE](#)、[\\$ZPI](#)、[\\$ZTIMESTAMP](#)、[\\$ZTIMEZONE](#)、[\\$ZVERSION](#)。

## 述語のリスト

SQL の述語は以下のカテゴリに分類されます。

- ・ [等値比較述語](#)
- ・ [BETWEEN 述語](#)
- ・ [IN および %INLIST 述語](#)
- ・ [%STARTSWITH 述語](#) および [包含関係演算子](#)
- ・ [NULL 述語](#)
- ・ [EXISTS 述語](#)
- ・ [FOR SOME 述語](#)
- ・ [FOR SOME %ELEMENT 述語](#)
- ・ [LIKE](#)、[%MATCHES](#)、および [%PATTERN 述語](#)
- ・ [%INSET](#) および [%FIND 述語](#)

## 述語の大文字と小文字の区別

述語では、フィールドに対して定義された[照合タイプ](#)が使用されます。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

[%INLIST](#)、[包含関係演算子 \(I\)](#)、[%MATCHES](#)、および [%PATTERN 述語](#) は、フィールドの既定の照合を使用しません。常に大文字と小文字を区別する EXACT 照合が使用されます。

2 つのリテラル文字列の熟語の比較は、常に大文字小文字を区別します。

## 述語の条件と %NOINDEX

述語の条件の前に [%NOINDEX](#) キーワードを置くと、クエリ・オプティマイザが条件でインデックスを使用することを防ぐことができます。これは、多数の行を満たす範囲条件を指定する場合に最も便利です。例えば、`WHERE %NOINDEX Age >= 1` のようにします。詳細は、["%ALLINDEX、%IGNOREINDEX、%NOINDEX の使用法"](#) を参照してください。

## 異常値に対する述語条件

ダイナミック SQL クエリ内の WHERE 節が NULL 以外の異常値を選択する場合、異常値リテラルを二重括弧で囲むことでパフォーマンスを大幅に向上させることができます。この二重括弧により、ダイナミック SQL は最適化の際に Outlier Selectivity (異常値の選択性) を使用します。例えば、企業がマサチューセッツ (MA) に所在している場合、従業員の大半はマサチューセッツに居住していることになります。Employees テーブルの Home\_State フィールドでは、'MA' が異常値です。この値を最適化して選択するには、`WHERE Home_State=(( 'MA' ))` を指定する必要があります。

この構文は、埋め込み SQL やビュー定義では使用すべきではありません。埋め込み SQL またはビュー定義では、Outlier Selectivity が常に使用され、特別なコーディングは不要です。

ダイナミック SQL クエリ内の WHERE 節は、NULL の異常値に対する最適化を自動的に実行します。例えば、WHERE FavoriteColors IS NULL などの節です。NULL が異常値である場合、IS NULL 述語と IS NOT NULL 述語に対する特別なコーディングは不要です。

Outlier Selectivity は、[テーブルのチューニング](#)・ユーティリティの実行により決定されます。

## 等値比較述語

以下は、使用できる等値比較述語です。

### テーブル C-2: SQL 等値比較述語

述語	処理
=	等しい
<>	等しくない
!=	等しくない
>	より大きい
<	より小さい
>=	以上
<=	以下

以下はその例です。

### SQL

```
SELECT Name, Age FROM Sample.Person
WHERE Age < 21
```

SQL は照合 (値がソートされる順番) という点から比較演算子を定義します。まったく同様の方法で照合する場合の 2 つの値は等しくなります。2 つ目の値の後に照合される場合、値は別の値よりも大きくなります。文字列フィールドの照合は、フィールドの既定の照合を利用します。InterSystems IRIS の既定の照合は大文字と小文字を区別しません。そのため、2 つの文字列フィールドの値の比較または文字列フィールド値と文字列リテラルとの比較では、既定では大文字と小文字は区別されません。例えば、Home\_State フィールドの値が大文字の 2 文字の文字列の場合、以下のようになります。

式	値
'MA' = Home_State	値 MA に対して True
'ma' = Home_State	値 MA に対して True
'VA' < Home_State	値 VT、WA、WI、WV、WY に対して True
'ar' >= Home_State	値 AK、AL、AR に対して True

ただし、2 つのリテラル文字列の比較では大文字と小文字が区別され、WHERE 'ma' = 'MA' は常に FALSE です。

## BETWEEN 述語

BETWEEN 比較演算子では、構文 BETWEEN lowval AND highval で指定された範囲内にあるデータ値を選択できます。この範囲には、lowval 値と highval 値も含まれます。これは、「以上」演算子と「以下」演算子の組み合わせと同じ働きをします。この比較は、以下の例を参照してください。

## SQL

```
SELECT Name, Age FROM Sample.Person
WHERE Age BETWEEN 18 AND 21
```

これにより、Sample.Person テーブルの Age 値が 18 ～ 21 の範囲にあるすべてのレコードを返します。BETWEEN 値は昇順で指定する必要があることに注意してください。BETWEEN 21 AND 18 のような述語はレコードを返しません。

ほとんどの述語と同様、以下の例に示すように、BETWEEN は NOT 論理演算子を使用して反転させることができます。

## SQL

```
SELECT Name, Age FROM Sample.Person
WHERE Age NOT BETWEEN 20 AND 55
ORDER BY Age
```

これにより、Sample.Person テーブルの Age 値が 20 より小さく 55 より大きいすべてのレコードを返します。

BETWEEN は通常、数値順に照合を行う数値の範囲に使用します。ただし、BETWEEN は、任意のデータ型の値の照合範囲に使用できます。

BETWEEN はマッチングの対象となる列と同じ照合タイプを使用します。既定では、文字列データ型の照合は大文字と小文字が区別されません。

詳細は、“[BETWEEN](#)” 述語を参照してください。

## IN および %INLIST 述語

IN 述語は、構造化されていない一連の項目に値を一致させるために使用されます。この構文は以下のとおりです。

```
WHERE field IN (item1,item2[,...])
```

等式テストと同様に、[照合](#)が IN 比較に適用されます。IN は、フィールドの既定の照合を使用します。既定では、フィールド文字列の値の比較では大文字と小文字が区別されません。

[%INLIST](#) 述語は、値を InterSystems IRIS リスト構造の要素に一致させるための InterSystems IRIS の拡張機能です。この構文は以下のとおりです。

```
WHERE item %INLIST listfield
```

%INLIST は EXACT 照合を使用します。そのため、既定では、%INLIST の文字列比較では大文字と小文字が区別されます。

どちらの述語を使用しても、等値比較やサブクエリ比較を実行できます。

詳細は、“[IN](#)” および “[%INLIST](#)” を参照してください。

## 部分文字列述語

以下を使用してフィールドの値と部分文字列を比較できます。

### テーブル C-3: SQL 部分文字列述語

述語	処理
<a href="#">%STARTSWITH</a>	値は、指定された部分文字列で始まる必要があります。
<a href="#">[</a>	包含関係演算子。値は、指定された部分文字列を含む必要があります。

## %STARTSWITH 述語

InterSystems IRIS %STARTSWITH 比較演算子により、文字列や数字の先頭文字列との部分的マッチングを実行できます。以下の例は、%STARTSWITH を使用して、Name の値が “S” で始まるレコードを選択します。

### SQL

```
SELECT Name, Age FROM Sample.Person  
WHERE Name %STARTSWITH 'S'
```

他の文字列フィールドの比較と同様に、%STARTSWITH の比較はフィールドの既定の照合を使用します。既定では、文字列フィールドでは大文字と小文字が区別されません。以下はその例です。

### SQL

```
SELECT Name, Home_City, Home_State FROM Sample.Person  
WHERE Home_City %STARTSWITH Home_State
```

詳細は、“%STARTSWITH” を参照してください。

## 包含関係演算子 (I)

包含関係演算子は開始ブラケット記号 (I) です。これを使用して、部分文字列 (文字列または数値) とフィールドの値の任意の部分とのマッチングができます。比較は常に大文字と小文字が区別されます。以下の例は、包含関係演算子を使用して、Name の値に “S” を含むレコードを選択します。

### SQL

```
SELECT Name, Age FROM Sample.Person  
WHERE Name [ 'S'
```

## NULL 述語

定義されていない値を検出します。すべての NULL 値またはすべての NULL でない値を検出できます。NULL 述語には以下の構文があります。

```
WHERE field IS [NOT] NULL
```

NULL 述語条件は、WHERE 節のストリーム・フィールドで使用可能な述語の 1 つです。

詳細は、“NULL” を参照してください。

## EXISTS 述語

サブクエリが空のセットを評価するかどうかをテストするために、サブクエリを使用して処理します。

### SQL

```
SELECT t1.disease FROM illness_tab t1 WHERE EXISTS  
(SELECT t2.disease FROM disease_registry t2  
WHERE t1.disease = t2.disease  
HAVING COUNT(t2.disease) > 100)
```

詳細は、“EXISTS” を参照してください。

## FOR SOME 述語

WHERE 節で FOR SOME 述語を使用して、1 つ以上のフィールド値の条件テストに基づいてレコードを返すかどうかを判断できます。この述語の構文は以下のとおりです。

```
FOR SOME (table [AS t-alias]) (fieldcondition)
```

FOR SOME は、fieldcondition が True に評価される必要があるということを指定します。指定された条件に 1 つ以上のフィールド値が一致する必要があります。table には、単一のテーブル、またはコンマ区切りのテーブルのリストを指定可能であり、各テーブルにはオプションでテーブル・エイリアスを指定できます。fieldcondition には、指定された table 内の 1 つ以上のフィールドのために 1 つ以上の条件を指定します。table 引数と fieldcondition 引数は、どちらも括弧で区切る必要があります。

以下の例では、FOR SOME 述語を使用して、結果セットを返すかどうか決定しています。

## SQL

```
SELECT Name, Age AS AgeWithWorkers
FROM Sample.Person
WHERE FOR SOME (Sample.Person) (Age < 65)
ORDER BY Age
```

前述の例では、少なくとも 1 つのフィールドに、指定された年よりも小さい Age 値が含まれる場合、すべてのレコードが返されます。それ以外の場合、レコードは返されません。

詳細は、“FOR SOME” を参照してください。

## FOR SOME %ELEMENT 述語

WHERE 節の FOR SOME %ELEMENT 述語には、以下の構文を使用します。

```
FOR SOME %ELEMENT(field) [AS e-alias] (predicate)
```

FOR SOME %ELEMENT 述語は、指定された predicate 節の値と field の要素を照合します。SOME キーワードは、field の要素の最低 1 つが、指定した predicate 条件を満たす必要があることを指定します。predicate には、キーワード %VALUE または %KEY を指定できます。

FOR SOME %ELEMENT 述語はコレクション述語です。

詳細は、“FOR SOME %ELEMENT” を参照してください。

## LIKE、%MATCHES、および %PATTERN 述語

これらの 3 つの述語を使用してパターン・マッチングを実行できます。

- **LIKE** では、リテラルとワイルドカードを使用してパターン・マッチを実行できます。リテラル文字の既知の部分文字列、または既知のシーケンス内にいくつかの既知の部分文字列を含むデータ値を返す場合は、LIKE を使用します。LIKE は、大文字と小文字の比較に、そのターゲットの照合を使用します。
- **%MATCHES** では、リテラル、ワイルドカード、リスト、および範囲を使用してパターン・マッチを実行できます。リテラル文字の既知の部分文字列、指定された文字のリストまたは範囲にマッチする 1 つ以上のリテラル文字、または既知のシーケンス内にそうしたいくつかの部分文字列を含むデータ値を返す場合は、%MATCHES を使用します。%MATCHES は、大文字と小文字の比較に EXACT 照合を使用します。
- **%PATTERN** を使用すると、文字タイプのパターンを指定できます。例えば、'1U4L1', '.A' (1 つの大文字、4 つの小文字、1 つのリテラル・コンマ、その後任意数の大文字または小文字が続くパターン) などです。文字タイプの既知のシーケンスを含むデータ値を返す場合は、%PATTERN を使用します。%PATTERN には既知のリテラル文字を指定できますが、特に、データ値は重要でないけれども、その値の文字タイプ形式が重要である場合に役立ちます。

文字列の最初の文字で比較を実行するには、%STARTSWITH 述語を使用します。

## 述語および論理演算子

AND および OR 論理演算子で複数の述語を連結することができます。括弧を使用すると複数の述語をグループにできます。InterSystems IRIS は、定義済みのインデックスおよび他の最適化を使用して WHERE 節の実行を最適化するため、AND および OR 論理演算子でつながれた述語が評価される順序は予測できません。このような理由から、複数の

述語の指定順序によるパフォーマンスへの影響は、ほとんどないか、まったくありません。熟語を厳密に左から右に評価する必要がある場合は、[CASE](#) 文を使用できます。

注釈 OR 論理演算子は、テーブル・フィールドを参照する [FOR SOME %ELEMENT](#) コレクション述語と、別のテーブル内のフィールドを参照する述語を関連付けるためには使用できません。以下はその例です。

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple')  
OR t2.Age < 65
```

この制限はオプティマイザがインデックスを使用する方法に依存するので、SQL はこの制限を、インデックスがテーブルに追加されるときにのみ実施できます。すべてのクエリでこのタイプの論理は使用しないことを強くお勧めします。

詳細は、“[論理演算子](#)” を参照してください。

## 関連項目

- ・ [SELECT](#) 文
- ・ [HAVING](#) 節
- ・ [述語の概要](#)
- ・ [データベースの問い合わせ](#)
- ・ [SQLCODE エラー・メッセージ](#)



# WHERE CURRENT OF (SQL)

カーソルを使用して現在の行を指定する UPDATE/DELETE 節です。

## 構文

WHERE CURRENT OF *cursor*

## 概要

WHERE CURRENT OF 節は[カーソル・ベースの埋め込み SQL](#) UPDATE または DELETE 文で使用して、カーソルが位置付けられたレコードを更新または削除することを指定できます。以下に例を示します。

### ObjectScript

```
&sql(DELETE FROM Sample.Employees WHERE CURRENT OF EmployeeCursor)
```

最後の FETCH コマンドによって "EmployeeCursor" カーソルから取得した行を削除します。

埋め込み SQL UPDATE または DELETE では、[WHERE](#) 節 (カーソルなし)、または宣言されたカーソルを持つ WHERE CURRENT OF で使用できます (両方は不可)。UPDATE または DELETE を指定した場合に、WHERE と WHERE CURRENT OF のいずれも指定しないときは、テーブル内のすべてのレコードが更新または削除されます。

## UPDATE の制限事項

WHERE CURRENT OF 節を使用する場合は、現在のフィールド値を使用して新しい値を生成してフィールドを更新することはできません。例としては、SET Salary=Salary+100 や SET Name=UPPER(Name) が挙げられます。これを実行しようとする、SQLCODE -69 エラーが返されます。SET <field> = <value expression> を WHERE CURRENT OF <cursor> で使用できません。

## 引数

### cursor

処理が現在の cursor の場所で実行されることを指定します。cursor はテーブルを指す[カーソル](#)です。

## 例

以下の埋め込み SQL の例は、WHERE CURRENT OF を使用した UPDATE 処理を示しています。

### ObjectScript

```
NEW %ROWCOUNT,%ROWID
&sql(DECLARE WPCursor CURSOR FOR
      SELECT Lang FROM SQLUser.WordPairs
      WHERE Lang='Sp')
&sql(OPEN WPCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH WPCursor)
      QUIT:SQLCODE
      &sql(UPDATE SQLUser.WordPairs SET Lang='Es'
          WHERE CURRENT OF WPCursor)
      IF SQLCODE=0 {
        WRITE !,"Update succeeded"
        WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
      ELSE {
        WRITE !,"Update failed, SQLCODE=",SQLCODE }
      }
&sql(CLOSE WPCursor)
```

以下の埋め込み SQL の例は、WHERE CURRENT OF を使用した DELETE 処理を示しています。

## ObjectScript

```
NEW %ROWCOUNT,%ROWID
&sql(DECLARE WPCursor CURSOR FOR
      SELECT Lang FROM SQLUser.WordPairs
      WHERE Lang='En')
&sql(OPEN WPCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH WPCursor)
      QUIT:SQLCODE
      &sql(DELETE FROM SQLUser.WordPairs
            WHERE CURRENT OF WPCursor)
      IF SQLCODE=0 {
        WRITE !,"Delete succeeded"
        WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
      ELSE {
        WRITE !,"Delete failed, SQLCODE=",SQLCODE }
      }
&sql(CLOSE WPCursor)
```

## 関連項目

- ・ [DECLARE、OPEN、FETCH、CLOSE](#)
- ・ [DELETE、UPDATE、INSERT OR UPDATE](#)
- ・ [SQL カーソル](#)
- ・ [SQLCODE エラー・メッセージ](#)

# SQL の述語条件

## 述語の概要

True または False に評価される論理条件を記述します。

### 述語の使用法

述語は、True または False のブーリアン値に評価される条件式です。

述語は次のように使用できます。

- ・ [SELECT](#) 文の [WHERE](#) 節または [HAVING](#) 節で使用し、特定のクエリに関連する行を特定します。すべての述語が [HAVING](#) 節で使用できるわけではないことに注意してください。
- ・ [JOIN](#) 操作の [ON](#) 節で、結合操作に関連する行を特定します。
- ・ [UPDATE](#) または [DELETE](#) 文の [WHERE](#) 節で使用し、変更する行を特定します。
- ・ [WHERE CURRENT OF](#) 文の [AND](#) 節で使用します。
- ・ [CREATE TRIGGER](#) 文の [WHEN](#) 節で使用して、トリガされるアクション・コードを適用するタイミングを決定します。
- ・ [DROP TABLE](#) などの [DROP](#) 文で使用して、ターゲットが存在しない場合にエラーが発生しないようにします。

### 述語のリスト

それぞれの述語の中では、1 つ以上の比較演算子を記号またはキーワードで指定します。InterSystems SQL は以下の比較演算子をサポートします。

比較演算子	説明
= (等しい) <> (等しくない) != (等しくない) > (より大きい) >= (以上) < (より小さい) <= (以下)	等値比較条件。数値比較または文字列照合順序比較に使用できます。数値比較では、空文字列("") は 0 と評価されます。等値比較の NULL は、常に空のセットを返します。代わりに、IS NULL 述語を使用します。“ <a href="#">関係演算子</a> ”を参照してください。
IS [NOT] NULL	フィールドに未定義 (NULL) の値があるかどうかをテストします。“ <a href="#">IS NULL</a> ”を参照してください。
IS [NOT] JSON	値が JSON 形式の文字列であるか、または JSON 配列や JSON オブジェクトへの <a href="#">oref</a> であるかをテストします。“ <a href="#">IS JSON</a> ”を参照してください。
EXISTS (subquery)	サブクエリを使用して、指定したテーブルに 1 つ以上の行が存在するかどうかをテストします。詳細は、“ <a href="#">EXISTS</a> ”を参照してください。
DROP-command IF EXISTS objectname	指定したターゲットが存在することを、DROP コマンドの実行の条件とし、これが存在しない場合にエラーにならないようにします。詳細は、“ <a href="#">EXISTS</a> ”を参照してください。

比較演算子	説明
BETWEEN x AND y	BETWEEN 条件は、 $\geq$ と $\leq$ の比較条件と一緒に使用します。一致条件は、指定した 2 つの範囲制限値の間にあることです (2 つの値も含まれます)。 <a href="#">“BETWEEN”</a> を参照してください。
IN (item1,item2[....,itemn]) IN (subquery)	フィールド値を、コンマ区切りリスト内のいずれかの項目、またはサブクエリにより返されるいずれかの項目と一致させる等値条件。詳細は、 <a href="#">“IN”</a> を参照してください。
%INLIST listfield	フィールド値を %List 構造リスト内のいずれかの要素と一致させる等値条件。詳細は、 <a href="#">“%INLIST”</a> を参照してください。
[	<a href="#">包含関係演算子</a> 。一致条件は、指定した文字列が含まれることです。包含関係演算子は、EXACT 照合を使用し、大文字と小文字を区別します。値は論理形式で指定する必要があります。
]	<a href="#">後続関係演算子</a> 。一致条件は、照合順において指定された項目の後に来ることです。値は論理形式で指定する必要があります。
%STARTSWITH 文字列	一致条件は、指定した文字列で始まることです。詳細は、 <a href="#">“%STARTSWITH”</a> を参照してください。
FOR SOME	ブーリアンの比較条件。FOR SOME 条件は、指定されたフィールドの少なくとも 1 つのデータ値に対して、True である必要があります。詳細は、 <a href="#">“FOR SOME”</a> を参照してください。
FOR SOME %ELEMENT	%VALUE または %KEY 述語節があるリスト要素比較条件。%VALUE は、リスト内の少なくとも 1 つの要素の値と一致しなければなりません。%KEY は、リスト内の要素の数以下でなければなりません。%VALUE 節と %KEY 節は、他の任意の比較演算子を使用できます。詳細は、 <a href="#">“FOR SOME %ELEMENT”</a> を参照してください。
LIKE	リテラルおよびワイルドカードを使用したパターン・マッチ条件。リテラル文字の既知の部分文字列を含むか、既知のシーケンス内にいくつかの既知の部分文字列を含むデータ値を返す場合は、LIKE を使用します。LIKE は、大文字と小文字の比較に、そのターゲットの照合を使用します。(包含関係演算子は、EXACT 照合を使用するという点で異なります。)詳細は、 <a href="#">“LIKE”</a> を参照してください。
%MATCHES	リテラル、ワイルドカード、リスト、および範囲を使用したパターン・マッチ条件。リテラル文字の既知の部分文字列を含むか、指定された文字のリストや範囲にマッチする 1 つ以上のリテラル文字を含むか、既知のシーケンス内にそうしたいくつかの部分文字列を含むデータ値を返す場合は、%MATCHES を使用します。%MATCHES は、大文字と小文字の比較に EXACT 照合を使用します。詳細は、 <a href="#">“%MATCHES”</a> を参照してください。

比較演算子	説明
%PATTERN	文字タイプを使用したパターン・マッチ条件。例えば、'1U4L1', 'A' (1 つの大文字、4 つの小文字、1 つのリテラル・コンマ、その後に任意数の大文字または小文字が続くパターン) などです。文字タイプの既知のシーケンスを含むデータ値が返されるようにするには、%PATTERN を使用します。%PATTERN には既知のリテラル文字を指定できますが、特に、データ値は重要でなくても、その値の文字タイプ形式が重要である場合に役立ちます。詳細は、“%PATTERN” を参照してください。
ALL ANY SOME	限定比較条件。詳細は、“ALL”、“ANY”、および“SOME” を参照してください。
%INSET %FIND	抽象的な、プログラムで指定された一時ファイルまたはビットマップ・インデックスを使用して RowId フィールドの値をフィルタ処理できるフィールド値比較条件。 <a href="#">%INSET</a> は、単純な比較をサポートします。 <a href="#">%FIND</a> は、ビットマップ・インデックスを使用する比較をサポートします。

## NULL

NULL とは、値が存在しないことです。当然ながら、これはすべてのブーリアン・テストに失敗します。NULL に等しい値、または等しくない値、NULL より大きい値または小さい値はありません。NULL=NULL であっても、述語としては失敗します。IN 述語は一連の OR で連結された等式テストであるので、IN 値リストに NULL を指定する意味はありません。したがって、何らかの述語条件を指定すると、NULL であるそのフィールドのすべてのインスタンスは削除されます。述語条件からの結果セットに NULL フィールドを組み込む唯一の方法は、IS NULL 述語を使用することです。詳細は、以下の例を参照してください。

## SQL

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors = $LISTBUILD('Red') OR FavoriteColors IS NULL
```

## 照合

述語では、フィールドに対して定義された[照合タイプ](#)が使用されます。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

クエリで照合タイプを指定する場合、比較の両側でそれを指定する必要があります。照合タイプを指定すると、インデックスの使用に影響がある場合があります。詳細は“[インデックス照合](#)”を参照してください。

特定の述語比較には、文字列内に埋め込まれている部分文字列 (包含関係演算子 (D)、%MATCHES 述語、および %PATTERN 述語) が含まれます。これらの述語は、必ず EXACT 照合を使用し、そのため必ず大文字と小文字を区別します。一部の照合は空白スペースを文字列の先頭に追加するので、これらの述語はフィールドの既定の照合の後に続く場合、その関数を実行できませんでした。ただし、LIKE 述語は、文字列内に埋め込まれた部分文字列に一致させるワイルドカードを使用できます。LIKE は、フィールドの既定の照合を使用します。これは既定では大文字と小文字を区別しません。

## 複合述語

条件式の最も単純な形が述語です。条件式は 1 つ以上の述語で構成されます。AND および OR 論理演算子で複数の述語をつなぐことができます。述語の前に単項否定演算子を置くと、述語の意味が反対になります。単項否定演算子は、その直後の述語のみに影響します。述語は、左から右の順に評価されます。述語をまとめるには、括弧を使用します。一連の述語の意味を反転させるには、開き括弧の前に単項否定演算子を付けます。括弧の前後、および括弧と論理演算子の間には空白は不要です。

IN 述語および %INLIST 述語は、複数の OR 等価述語と機能的に同等です。以下の例は同じものです。

### ObjectScript

```
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State='MA' OR Home_State='VT' OR Home_State='NH'"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

### ObjectScript

```
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State IN('MA','VT','NH')"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

### ObjectScript

```
SET list=$LISTBUILD("MA","VT","NH")
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State %INLIST(?)"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(list)
DO rset.%Display()
```

FOR SOME %ELEMENT 述語には、論理演算子を含めることができ、論理演算子を使用して他の述語とつなげることもできます。詳細は、以下の例を参照してください。

### SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red' OR %Value='White'
OR %Value %STARTSWITH 'B')
AND (Name BETWEEN 'A' AND 'F' OR Name %STARTSWITH 'S')
ORDER BY Name
```

(Name BETWEEN 'A' AND 'F' OR Name %STARTSWITH 'S') の前後の括弧に注意してください。このグループ化の括弧がないと、FOR SOME %ELEMENT 条件は Name %STARTSWITH 'S' に適用されません。

## OR を使用したコレクション述語

FOR SOME %ELEMENT はコレクション述語です。OR 論理演算子と共にこの述語を使用することは、以下のように制限されます。OR 論理演算子は、テーブル・フィールドを参照するコレクション述語と、別のテーブル内のフィールドを参照する述語を関連付けるためには使用できません。以下はその例です。

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple')
OR t2.Age < 65
```



この制限はオブティマイザがインデックスを使用する方法に依存するので、SQL はこの制限を、インデックスがテーブルに追加されるときにのみ実施できます。このタイプの論理は、すべてのクエリで使用しないことを強くお勧めします。

## 述語と %SelectMode

すべての述語は、論理 (内部保存) データ値を使用してそれらの比較を実行します。ただし、一部の述語は、述語値 (1 つまたは複数) に対して形式モード変換を実行して、それを ODBC 形式または表示形式から論理形式へ変換できます。その他の述語は形式モード変換は実行できないため、必ず論理形式で述語値を指定する必要があります。

形式モード変換を実行する述語は、照合フィールドのデータ型 (DATE または %List) から変換が必要かどうかを判断し、[%SelectMode 設定](#)から変換のタイプを決定します。%SelectMode が論理形式以外の値 (%SelectMode=ODBC または %SelectMode=Display) に設定されている場合は、述語値 (1 つまたは複数) を適切な ODBC 形式または表示形式で指定する必要があります。

- 等値述語は、形式モード変換を実行します。InterSystems IRIS は、述語値を論理形式に変換してから、それをフィールド値と照合します。%SelectMode が論理形式以外のモードに設定されている場合は、表示値と論理格納値とが異なるデータ型のために、述語値 (1 つまたは複数) を %SelectMode 形式 (ODBC または表示) で指定する必要があります。例えば、日付、時刻、および %List 形式設定済み文字列などがあります。InterSystems IRIS はこの形式変換を自動的に実行するため、論理形式でこのタイプの述語値を指定すると、通常は SQLCODE エラーとなります。例えば、SQLCODE -146 “[入力された日付を妥当な日付論理値に変換できません]” です (InterSystems IRIS は、指定された論理値が ODBC 値または表示値であると見なし、論理値に変換しようしますが、成功しません)。影響を受ける述語には、=、<、>、BETWEEN、および IN などがあります。
- パターン述語は、形式モード変換を実行できません。これは、InterSystems IRIS が述語値を意味のある値に変換できないためです。したがって、述語値は、%SelectMode 設定に関係なく、論理形式で指定する必要があります。述語値 (1 つまたは複数) を ODBC 形式または表示形式で指定すると、通常は、データが一致しないか、意図していないデータと一致することになります。影響を受ける述語には、%INLIST、LIKE、%MATCHES、%PATTERN、%STARTSWITH、[ (包含関係演算子)、および ] (後続関係演算子) などがあります。

[%INTERNAL](#)、[%EXTERNAL](#)、または [%ODBCOUT](#) 形式変換関数を使用すると、述語による操作対象となるフィールドを変換できます。これにより、述語値を別の形式で指定できるようになります。例えば、WHERE %ODBCOut (DOB) %STARTSWITH '1955-' のようにします。ただし、照合フィールドに対して形式変換関数を指定すると、そのフィールドのためのインデックスを使用できなくなります。これは、パフォーマンスに大きな悪影響を及ぼす可能性があります。

以下のダイナミック SQL の例の BETWEEN 述語 (等値述語) では、%SelectMode=1 (ODBC) 形式で日付を指定する必要があります。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB BETWEEN '1950-01-01' AND '1960-01-01'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下のダイナミック SQL の例では、%STARTSWITH 述語 (パターン述語) は形式モード変換を実行できません。1 つ目の例では、1950 年代の年について、%SelectMode=ODBC の形式で、日付のために %STARTSWITH を指定しようとしています。しかし、\$HOROLOG 195 で始まる誕生日 (1894 年内の日付) がテーブルに含まれていないため、行は選択されません。

## ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '195'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下の例では、ODBC形式で1950年代の年を選択するために%STARTSWITHを使用できるよう、照合するDOBフィールドに対して%ODBCOut形式変換関数を使用します。ただし、この使用法ではDOBフィールドでインデックスを使用できなくなることにご注意ください。

## ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOut(DOB) %STARTSWITH '195'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下の例では、%STARTSWITH 述語は、論理 (内部) 形式で日付のために %STARTSWITH を指定します。41 で始まるDOB論理値がある行 (1953年4月4日 (\$HOROLOG 41000) から1955年12月28日 (\$HOROLOG 41999) までの日付) が選択されます。DOBフィールドのインデックスが使用されます。

## ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '41'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

## 述語と PosixTime、Timestamp、および Date

等値述語比較は自動的にこれらのさまざまな日付および日付/時刻表現の変換を実行します。この変換は%SelectModeとは関係ありません。したがって、意味のあるすべての比較述語は以下のようになります。

```
WHERE MyPosixField = MyTimestampField
WHERE MyPosixField < CURRENT_TIMESTAMP
WHERE MyPosixField BETWEEN DATEADD('month',-1,CURRENT_TIMESTAMP) AND $HOROLOG
WHERE MyPosixField BETWEEN DATEADD('day',-1,CURRENT_DATE) AND LAST_DAY(CURRENT_DATE)
```

%STARTSWITHのようなパターン述語比較は、異なる日付および日付/時刻表現間の変換を実行しません。これらは実際に保存されている日付値に対して機能します。

## リテラル置換の抑制

述語引数を二重の括弧で囲むことで、コンパイルの事前解析時にリテラル置換を抑制できます。例えば、LIKE(('abc%'))のようにします。これにより、全体的な選択性や添え字境界の選択性が向上し、クエリのパフォーマンスが向上する場合があります。ただし、同じクエリを別の値で複数回呼び出すことは避けてください。それぞれのクエリ呼び出しでキャッシュされたクエリが個別に作成されることになってしまいます。

## 例

以下の例は、クエリの WHERE 節における多様な条件の使用を示しています。

### SQL

```
SELECT PurchaseOrder FROM MyTable
  WHERE OrderTotal >= 1000
     AND ItemName %STARTSWITH :partname
     AND AnnualOrders BETWEEN 50000 AND 100000
     AND City LIKE 'Ch%'
     AND CustomerNumber IN
       (SELECT CustNum FROM TheTop100
        WHERE TheTop100.City='Boston')
     AND :minorder > SOME
       (SELECT OrderTotal FROM Orders
        WHERE Orders.Customer = :cust)
```

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [CREATE TRIGGER](#)

# ALL (SQL)

値をサブクエリで取得された対応するすべての値と照合します。

## 構文

```
scalar-expression comparison-operator ALL (subquery)
```

## 概要

ALL キーワードは、比較演算子と併用し、スカラ式の値がサブクエリで取得した対応する値のすべてと一致した場合に True となる述語 (限定比較条件) を作成します。ALL 述語は、1 つの scalar-expression 項目とサブクエリの 1 つの SELECT 項目を比較します。SELECT 項目が複数あるサブクエリを実行すると、SQLCODE -10 エラーが生成されます。

述語条件を指定できる場所であればどこでも、ALL を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

該当する場合、システムは、ALL サブクエリに集合値サブクエリの最適化 (SVSO) を自動的に適用します。この最適化の詳細、および %NOSVSO キーワードを使用したこの最適化のオーバーライドの詳細は、“[FROM 節](#)” のリファレンス・ページで“クエリ最適化オプション”を参照してください。

## 引数

### scalar-expression

その値を subquery で生成された結果セットと比較するスカラ式 (通常はデータ列)。

### comparison-operator

次の比較演算子のいずれか : = (等しい)、< > または != (等しくない)、< (より小さい)、<= (以下)、> (より大きい)、>= (以上)、[ (包含)、または ] (後続)。

### subquery

括弧で囲まれたサブクエリで、これによって 1 つの列から返される結果セットが scalar-expression との比較に使用されます。

## 例

以下の例では、Person データベースの年齢で、Employee データベースのすべての年齢よりも少ないものを選択します。

### SQL

```
SELECT DISTINCT Age FROM Sample.Person
WHERE Age < ALL
  (SELECT Age FROM Sample.Employee)
ORDER BY Age
```

以下の例では、Person データベースの名前で、Employee データベースのすべての名前よりも短いものを選択します。

### SQL

```
SELECT $LENGTH(Name) AS NameLength, Name FROM Sample.Person
WHERE $LENGTH(Name) > ALL
  (SELECT $LENGTH(Name) FROM Sample.Employee)
OR $LENGTH(Name) < ALL
  (SELECT $LENGTH(Name) FROM Sample.Employee)
```

以下の例は、ミシシッピ川の西にあり、Manager と Director の役職を持つ従業員を含まない州のリストを返します。

### SQL

```
SELECT DISTINCT State
FROM Sample.USZipCode
WHERE Longitude < -93
      AND State != ALL
      (SELECT Home_State FROM Sample.Employee
       WHERE Title [ 'Manager' OR Title [ 'Director'])
ORDER BY State
```

### 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [ANY の述語条件](#)
- ・ [SOME の述語条件](#)
- ・ [述語の概要](#)

# ANY (SQL)

値をサブクエリで取得された少なくとも 1 つの一致する値と照合します。

## 構文

```
scalar-expression comparison-operator ANY (subquery)
```

## 概要

ANY キーワードは、比較演算子と併用し、スカラ式の値がサブクエリで取得した対応する値の 1 つ以上と一致した場合に True となる述語 (限定比較条件) を作成します。ANY 述語は、1 つの scalar-expression 項目とサブクエリの 1 つの SELECT 項目を比較します。SELECT 項目が複数あるサブクエリを実行すると、SQLCODE -10 エラーが生成されます。

注釈 キーワードの ANY および SOME は同義語です。

述語条件を指定できる場所であればどこでも、ANY を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

該当する場合、システムは、ANY サブクエリに集合値サブクエリの最適化 (SVSO) を自動的に適用します。この最適化の詳細、および %NOSVSO キーワードを使用したこの最適化のオーバーライドの詳細は、“[FROM 節](#)” のリファレンス・ページで“クエリ最適化オプション”を参照してください。

## 引数

### scalar-expression

その値を subquery で生成された結果セットと比較するスカラ式 (通常はデータ列)。

### comparison-operator

次の比較演算子のいずれか : = (等しい)、<> または != (等しくない)、< (より小さい)、<= (以下)、> (より大きい)、>= (以上)、[ (包含)、または ] (後続)。

### subquery

括弧で囲まれたサブクエリで、これによって返される結果セットが scalar-expression との比較に使用されます。

## 例

以下の例では、ミシシッピ川の西の州に住み、給与が \$75,000 よりも多い従業員を選択します。

### SQL

```
SELECT Name,Salary,Home_State FROM Sample.Employee
WHERE Salary > 75000
AND Home_State = ANY
  (SELECT State FROM Sample.USZipCode
   WHERE Longitude < -93)
ORDER BY Home_State
```

## 関連項目

- [SELECT 文](#)、[HAVING 節](#)、[WHERE 節](#)
- [ALL の述語条件](#)
- [SOME の述語条件](#)
- [述語の概要](#)

## BETWEEN (SQL)

値を値の範囲とマッチングします。

### 構文

```
scalar-expression BETWEEN lowval AND highval
```

### 説明

BETWEEN 述語では、lowval と highval で指定された範囲内にあるデータ値を選択できます。この範囲には、lowval 値と highval 値も含まれます。これは、「以上」演算子と「以下」演算子の組み合わせと同じ働きをします。この比較は、以下の例を参照してください。

#### SQL

```
SELECT Name, Age FROM Sample.Person  
WHERE Age BETWEEN 18 AND 21  
ORDER BY Age
```

これにより、Sample.Person テーブルの Age 値が 18 ～ 21 の範囲にあるすべてのレコードを返します。BETWEEN 値は昇順で指定する必要があることに注意してください。BETWEEN 21 AND 18 のような述語は、NULL 文字列を返します。スカラ式のいずれの値も指定された範囲に入らない場合、BETWEEN は NULL 文字列を返します。

ほとんどの述語と同様に、BETWEEN は NOT 論理演算子を使用して反転できます。BETWEEN も NOT BETWEEN も、NULL フィールドを返すために使用することはできません。NULL フィールドを返すには、[IS NULL](#) を使用します。以下の例では、NOT BETWEEN を示します。

#### SQL

```
SELECT Name, Age FROM Sample.Person  
WHERE Age NOT BETWEEN 20 AND 55  
ORDER BY Age
```

これにより、Sample.Person テーブルの Age 値が 20 より小さく 55 より大きいすべてのレコードを返します。

[述語条件](#)を指定できる場所であればどこでも、BETWEEN を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

### 照合タイプ

BETWEEN は通常、数値順に照合を行う数値の範囲に使用します。ただし、BETWEEN は、任意のデータ型の値の照合順範囲に使用できます。

BETWEEN はマッチングの対象となる列と同じ照合タイプを使用します。既定では、文字列データ型は大文字と小文字が区別されない SQLUPPER として照合されます。[現在のネームスペースにおける既定の文字列の照合](#)の定義および[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)の指定の詳細は、“[照合](#)”を参照してください。

クエリで列に別の照合タイプが割り当てられている場合、その照合タイプを BETWEEN substring にも適用する必要があります。詳細は、以下の例を参照してください。

以下の例では、BETWEEN はフィールドの既定である、大文字と小文字を区別しない照合 (SQLUPPER) を使用します。Name がアルファベット順で Home\_State よりも大きく、Home\_State がアルファベット順で Home\_City よりも大きなレコードを返します。



## SQL

```
SELECT Name,Home_State,Home_City
FROM Sample.Person
WHERE Home_State BETWEEN Name AND Home_City
ORDER BY Home_State
```

Home\_State フィールドが SQLUPPER として定義されているため、以下の例の BETWEEN 文字列比較では大文字と小文字が区別されません。つまり、lowval と highval は、機能的には同じになり、大文字小文字に関係なく 'MA' を選択します。

## SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State
      BETWEEN 'MA' AND 'Ma'
ORDER BY Home_State
```

以下の例では、%SQLSTRING 照合関数により、BETWEEN 文字列比較は大文字と小文字を区別します。Home\_State が 'MA' から 'Ma' の値のレコードが選択されます。このデータ・セットには、'MA'、'MD'、'ME'、'MO'、'MS'、および 'MT' が含まれます。

## SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE %SQLSTRING(Home_State)
      BETWEEN %SQLSTRING('MA') AND %SQLSTRING('Ma')
ORDER BY Home_State
```

以下の例では、BETWEEN 文字列比較は、大文字と小文字を区別せず、スペースと句読点を無視します。

## SQL

```
SELECT Name FROM Sample.Person
WHERE %STRING(Name) BETWEEN %SQLSTRING('OA') AND %SQLSTRING('OZ')
ORDER BY Name
```

ケース変換関数の詳細は、“[%SQLUPPER](#)”を参照してください。

以下の例では、INNER JOIN 操作の ON 節で使用されている BETWEEN を示しています。これは、大文字と小文字が区別されない文字列比較を実行しています。

## SQL

```
SELECT P.Name AS PersonName,E.Name AS EmpName
FROM Sample.Person AS P INNER JOIN Sample.Employee AS E
ON P.Name BETWEEN 'an' AND 'ch' AND P.Name=E.Name
```

## %SelectMode

論理形式ではない値に %SelectMode が設定されている場合は、BETWEEN 述語値を %SelectMode 形式 (ODBC または表示) で指定する必要があります。このことは主に、日付、時刻、および InterSystems IRIS 形式のリスト (%List) に当てはまります。論理形式で述語値 (1 つまたは複数) を指定すると、一般に、SQLCODE エラーが発生します。例えば、SQLCODE -146 “[入力された日付を妥当な日付論理値に変換できません]” が返されます。

以下のダイナミック SQL の例の BETWEEN 述語では、%SelectMode=1 (ODBC) 形式で日付を指定する必要があります。

## ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "  
SET q2 = "WHERE DOB BETWEEN '1950-01-01' AND '1960-01-01' "  
SET myquery = q1_q2  
SET tStatement = ##class(%SQL.Statement).%New()  
SET tStatement.%SelectMode=1  
SET qStatus = tStatement.%Prepare(myquery)  
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}  
SET rset = tStatement.%Execute()  
DO rset.%Display()  
WRITE !,"End of data"
```

## 引数

### scalar-expression

その値を lowval と highval の間 (包含的) の値の範囲と比較するスカラ式 (通常はデータ列)。

### expression

scalar-expression の各値とマッチングする値の範囲の先頭を指定する、照合順の最小値に解決される式。

### expression

scalar-expression の各値とマッチングする値の範囲の末尾を指定する、照合順の最大値に解決される式。

## 関連項目

- ・ [SELECT 文](#)、[HAVING 節](#)、[WHERE 節](#)
- ・ [述語の概要](#)
- ・ [照合](#)

# EXISTS (SQL)

指定されたオブジェクトの存在を確認します。

## 構文

### テーブルに少なくとも 1 行が存在するかを確認

```
EXISTS select-statement
```

### DROP コマンドの実行対象が存在しない場合にエラーを抑制

```
DROP-command IF EXISTS name
```

## 概要

述語 EXISTS は、主にテーブルに行が 1 つ以上存在するかどうかを調べるために、指定されたテーブルをテストします。EXISTS に続く SELECT 文は、何を含むかについてチェックされます。その節は多くの場合以下の形式です。

```
EXISTS (SELECT... FROM... WHERE...)
```

一般的な文は、以下のようになります。

## SQL

```
SELECT name
FROM Table_A
WHERE EXISTS
  (SELECT *
   FROM Table_B
   WHERE Table_B.Number = Table_A.Number)
```

この例では、サブクエリによって指定された単独あるいは複数の行が存在するかどうかを、述語がテストします。

このテストは、UNION 文ではなく SELECT 文で実行しなければならないことに注意してください。

以下の例のように、NOT EXISTS 節は、テーブル内に行が存在しないことをテストします。

## SQL

```
SELECT EmployeeName, Age
FROM Employees
WHERE NOT EXISTS (SELECT * FROM BonusTable
WHERE NOT (BonusTable.Result = 'Positive'
AND Employees.EmployeeNum = BonusTable.EmployeeNum))
```

[述語条件](#)を指定できる場所であればどこでも、EXISTS を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

該当する場合、システムは、EXISTS または NOT EXISTS サブクエリに集合値サブクエリの最適化 (SVSO) を自動的に適用します。この最適化の詳細、および %NOSVSO キーワードを使用したこの最適化のオーバーライドの詳細は、“[FROM 節](#)” のリファレンス ページで “クエリ最適化オプション” を参照してください。

以下の文に示すように、バリエーションの IF EXISTS を使用して、DROP TABLE などの DROP コマンドで、実行対象が存在するかどうかをそのコマンドの実行条件とすることができます。

## SQL

```
DROP TABLE IF EXISTS Records
```

この例では、テーブル Records が存在しない場合はエラーが発生しません。この文は SQLCODE 1 とメッセージを返します。この動作は、[管理ポータルまたは構成パラメータ・ファイル \(CPF\) で DDL 文を制御する設定](#)よりも優先されます。これにより、通知なしでエラーが抑制されます。

同様に、以下の文のように、CREATE TABLE コマンドを使用するときに IF NOT EXISTS を指定できます。

```
CREATE TABLE IF NOT EXISTS Records (...)
```

この例では、Records テーブルが既に存在する場合、コマンドは何も実行しません。エラーは発生せず、この文は SQLCODE 1 とメッセージを返します。この動作は、[管理ポータルまたは構成パラメータ・ファイル \(CPF\) で DDL 文を制御する設定](#)よりも優先されます。これによって、実質的に既存のテーブルが上書きされ、通知なしでエラーが抑制されます。詳細は、[CREATE TABLE](#)を参照してください。

## 引数

### select-statement

一般的に[条件式](#)を含む単純な[クエリ](#)。

### DROP-command

[DROP AGGREGATE](#)、[DROP DATABASE](#)、[DROP FUNCTION](#)、[DROP INDEX](#)、[DROP METHOD](#)、[DROP PROCEDURE](#)、[DROP QUERY](#)、[DROP ROLE](#)、[DROP TABLE](#)、[DROP TRIGGER](#)、[DROP USER](#)、[DROP VIEW](#) のいずれかのコマンド

## 関連項目

- ・ [SELECT 文](#)、[HAVING 節](#)、[WHERE 節](#)
- ・ [述語の概要](#)

## %FIND (SQL)

ビットマップ・チャンクの反復処理によって、値を一連の生成値とマッチングします。

### 構文

```
scalar-expression %FIND valueset [SIZE ((nn))]
```

### 説明

%FIND 述語では、valueset に指定された値と一致するデータ値を選択し、一連のビットマップ・チャンク内の値を反復処理することで、結果セットをフィルタ処理できます。この一致は、scalar-expression の値が valueset 内の値と一致したときに成功します。valueset の値がスカラ式のいずれの値とも一致しない場合、%FIND は NULL 文字列を返します。表示モードに関係なく、この照合は常に、論理 (内部保存) データ値に対して実行されます。

%FIND は、他の比較条件と同様に、SELECT 文の WHERE 節または HAVING 節の中で使用します。

%FIND では、抽象的な、プログラムで指定された一致する値のセットを使用して、フィールド値をフィルタ処理できます。具体的には、抽象的な、プログラムで指定されたビットマップを使用して、RowId フィールドの値をフィルタ処理できます。その場合、valueset はビットマップ・インデックスの添え字の層と同じように機能します。

このユーザ定義クラスは、抽象クラス %SQL.AbstractFind から派生します。この抽象クラスには、ContainsItem() ブーリアン・メソッドが定義されています。ContainsItem() メソッドは、scalar-expression の値を valueset の値とマッチングします。

一連のビットマップ・チャンク内の値に対する反復処理は、以下の 3 つのメソッドを使用して実行されます。

- ・ GetChunk(c) は、チャンク番号が c であるビットマップ・チャンクを返します。
- ・ NextChunk(.c) は、チャンク番号が c より大きい最初のビットマップ・チャンクを返します。
- ・ PreviousChunk(.c) は、チャンク番号が c より小さい最初のビットマップ・チャンクを返します。

これらの 4 つのメソッドの詳細は、%SQL.AbstractFind を参照してください。

### 照合タイプ

%FIND はマッチングの対象となる列と同じ照合タイプを使用します。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。現在のネームスペースにおける既定の文字列の照合を定義し、フィールド/プロパティの定義における既定以外のフィールドの照合タイプを指定することができます。列に別の照合タイプが割り当てられている場合、その照合タイプを %FIND substring にも適用する必要があります。ケース変換関数の詳細は、“%SQLUPPER” を参照してください。

### SIZE 節

オプションの %FIND SIZE 節には整数 nn が提供され、これによって valueset 内の値の桁数が指定されます。InterSystems IRIS はこの桁数を使用して、最適なクエリ・プランを決定します。nn は、10、100、1000、10000、その他のいずれかのリテラルで指定します。nn は、コンパイル時に定数値として使用可能であることが要求されるため、すべての SQL コードでリテラルとして指定する必要があります。埋め込み SQL を除くすべての SQL で、示されるように入れ子の括弧を指定する必要があります。

### %FIND と %INSET の比較

- ・ %INSET は、最も単純で最も一般的なインタフェースです。ContainsItem() メソッドをサポートします。
- ・ %FIND は、ビットマップ・インデックスを使ったビットマップ・チャンクの反復処理をサポートします。ObjectScript の \$ORDER 関数の機能をエミュレートし、NextChunk()、PreviousChunk()、GetChunk() の各反復処理メソッドと、ContainsItem() メソッドをサポートします。

## 引数

### scalar-expression

その値を valueset と比較するスカラ式 (通常はテーブルの RowId フィールド)。

### valueset

ビットマップ・チャンクの反復処理メソッドと ContainsItem() メソッドを実装するユーザ定義オブジェクトへのオブジェクト参照 (oref)。このメソッドは、一連のデータ値を取り、scalar-expression 内の値との一致があったときにブーリアン値を返します。

### SIZE ((nn))

クエリの最適化に使用される、桁数を表す整数 (10、100、1000 など) (オプション)。

## 関連項目

- ・ [SELECT](#) 文、[HAVING](#) 節、[WHERE](#) 節
- ・ [%INSET](#) 述語
- ・ [述語の概要](#)
- ・ [SEARCH\\_INDEX](#) 関数

# FOR SOME (SQL)

フィールド値の条件テストに基づいて、レコードを返すかどうかを決定します。

## 構文

```
FOR SOME (table [AS t-alias]) (fieldcondition)
```

## 説明

FOR SOME 述語を使用すると、テーブル内の 1 つまたは複数のフィールド値にブーリアン条件テストを実行した結果に基づいて、レコードを返すかどうかを決定できます。fieldcondition が True と評価された場合は、レコードが返されます。fieldcondition が False と評価された場合は、レコードは返されません。

**述語条件**を指定できる場所であればどこでも、FOR SOME を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

区切り括弧は、table (およびそのオプションの t-alias) 引数には必須となっています。区切り括弧は、fieldcondition 引数でも必須となります。空白は使用できますが、これら 2 組の括弧の間には必要ではありません。

一般に、FOR SOME は、別テーブル内のレコードの内容に基づいてテーブルからレコードを返すかどうかを決定するために使用されます。FOR SOME は、同一テーブル内のレコードの内容に基づいてテーブルからレコードを返すかどうかを決定するためにも使用できます。この後者の場合は、すべてのレコードが返されるか、レコードは返されないかのどちらかです。

## 複合条件

fieldcondition には、複数の条件式を含めることができます。条件のセットは括弧で囲みます。複数の条件は論理演算子 AND および OR を使用して指定します。これは & および ! 記号で指定することもできます。論理演算子の後に単項否定論理演算子を付けることもできます。既定では、条件は左から右の順に評価されます。括弧を使用して複数の条件をまとめることによって、評価の順序を変えることができます。

## SQL

```
SELECT Name,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name AND p.Name %STARTSWITH 'A')
ORDER BY Name
```

## SQL

```
SELECT Name,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name OR p.Name %STARTSWITH 'A')
ORDER BY Name
```

## 複数のテーブル

fieldcondition の前に、複数のテーブルをコンマ区切りリストとして指定できます。レコードを返すかどうかを決定する条件では、データの選択元のテーブルを参照できるほか、別のテーブルのフィールド値を参照することもできます。指定する各フィールドとそのテーブルを関連付けるために、通常はテーブルのエイリアスが必要となります。

## 引数

### table

table には、単一のテーブル、またはテーブルのコンマ区切りリストを指定できます。括弧で囲む必要があります。



## AS t-alias

上記の table 名のエイリアス (オプション)。エイリアスは有効な識別子である必要があり、区切り文字付き識別子とすることができます。

## fieldcondition

fieldcondition には、1 つまたは複数のフィールドを参照する、1 つまたは複数の条件式を指定します。fieldcondition は括弧で囲みます。AND (&) および OR (!) 論理演算子を使用して、複数の条件式を fieldcondition 内に指定できます。括弧で囲まれたサブクエリで、これによって 1 つの列から返される結果セットが scalar-expression との比較に使用されます。

## 例

以下の例では、FOR SOME は、Sample.Person テーブルから、その Name フィールド値が Sample.Employee テーブル内の Name フィールド値と一致するすべてのレコードを返します。

### SQL

```
SELECT Name,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name)
ORDER BY Name
```

以下の例では、FOR SOME は、同一テーブルのブーリアン・テストに基づいて Sample.Person テーブル内のレコードを返します。このプログラムは、65 より大きい Age 値が含まれるレコードが 1 つでもあれば、すべての Sample.Person レコードを返します。それ以外の場合はレコードを返しません。Sample.Person 内の 1 つ以上のレコードの Age フィールドに 65 より大きい値が含まれているため、すべての Sample.Person レコードが返されます。

### SQL

```
SELECT Name,Age,COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Person)(Age>65)
ORDER BY Age
```

ほとんどの述語と同様、以下の例に示すように、NOT 論理演算子を使用して FOR SOME の論理を反転できます。

### SQL

```
SELECT Name,Age,COUNT(Name) AS NameCount
FROM Sample.Person
WHERE NOT FOR SOME (Sample.Person)(Age>65)
ORDER BY Age
```

以下の例では、FOR SOME は、Sample.Person テーブルから、その Name フィールド値が Sample.Employee テーブル内の Name フィールド値と一致し、その人の住居の状態 (Home\_State) がその人のオフィスの状態 (Office\_State) と同じであるすべてのレコードを返します。

### SQL

```
SELECT Name,Home_State,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(p.Name=e.Name AND p.Home_State=e.Office_State)
ORDER BY Name
```

以下の例では、Sample.Person テーブル内と Sample.Employee テーブル内とで 1 つ以上の Name が一致すれば、すべてのレコードを返します。1 つ以上のレコードでこの条件が True であるため、すべての Sample.Person レコードが返されます。

## SQL

```
SELECT Name AS PersonName, Age, COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Employee AS e, Sample.Person AS p) (e.Name=p.Name)
ORDER BY Name
```

以下の例では、Sample.Person テーブル内と Sample.Company テーブル内とで 1 つ以上の Name が一致すれば、すべてのレコードを返します。個人名と企業名が(このデータ・セット内では)同じにはならないため、この条件はどのレコードでも True にはなりません。したがって、Sample.Person レコードは返されません。

## SQL

```
SELECT Name AS PersonName, Age, COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Company AS c, Sample.Person AS p) (c.Name=p.Name)
ORDER BY Name
```

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [述語の概要](#)
- ・ [FOR SOME %ELEMENT 述語](#)

## FOR SOME %ELEMENT (SQL)

リスト要素の値またはリスト要素の数を述語とマッチングします。

### 構文

```
FOR SOME %ELEMENT(field) [[AS] e-alias] (predicate)
```

### 説明

FOR SOME %ELEMENT 述語は、指定された predicate 節の値と field のリスト要素をマッチングします。SOME キーワードは、field の要素の少なくとも 1 つが、指定した predicate 節を満たす必要があることを指定します。

predicate 節には、述語条件が続く %VALUE キーワードまたは %KEY キーワードが含まれている必要があります。これらのキーワードは、大文字と小文字を区別しません。

%VALUE および %KEY の使用法を、以下の例で説明しています。

- ・ (%VALUE='Red') は、リスト要素の 1 つとして値 Red が含まれるすべての field 値と一致します。field には、単一の要素 Red のみを含めることも、複数の要素を含めてそのうちの 1 つを Red とすることもできます。
- ・ (%KEY=2) は、少なくとも 2 つの要素が含まれるすべての field 値と一致します。field には 2 つの要素のみを含めることも、2 つより多い数の要素を含めることもできます。%KEY 値は正の整数値にする必要があります。(%KEY=0) は、どのような field 値とも一致しません。

FOR SOME %ELEMENT は、NULL の field を一致させるために使用することはできません。

predicate 節は、等値条件だけでなく、任意の [述語条件](#) を使用できます。以下に predicate 節の例を示します。

```
(%VALUE='Red')
(%VALUE > 21)
(%VALUE %STARTSWITH 'R')
(%VALUE [ 'e' ])
(%VALUE IN ('Red', 'Blue'))
(%VALUE IS NOT NULL)
(%KEY=3)
(%KEY > 1)
(%KEY IS NOT NULL)
```

注釈 実行時に述語の値を指定した場合 ([? 入力パラメータ](#)または [:var 入力ホスト変数](#)を使用)、その結果の述語 %STARTSWITH 'abc' は、同等の結果の述語 LIKE 'abc%' よりもパフォーマンスが優れています。

AND、OR、および NOT [論理演算子](#)を使用して、複数の述語条件を指定できます。InterSystems IRIS は、結合した述語条件を各要素に適用します。したがって、2 つの %VALUE 述語または 2 つの %KEY 述語を AND テストを使用して適用することには意味がありません。

例えば、FOR SOME %ELEMENT を使用して、値 Red、Green、Red Green、Black Red、Green Yellow Red、Green Black、Yellow、または Black Yellow が含まれているフィールドを一致させます。

- ・ (%VALUE='Red') は、Red の要素を含むすべてのフィールド (Red、Red Green、Black Red、および Red Yellow Green) と一致します。
- ・ (%VALUE='Red' OR %VALUE='Green') は、Red と Green のいずれか (または順序を問わずその両方) の要素を含むフィールド (Red、Green、Red Green、Black Red、Green Yellow Red、Green Black) と一致します。この機能は、(%VALUE IN('Red', 'Green')) と同じです。
- ・ (%VALUE='Red' AND %VALUE='Green') は、どのフィールド値とも一致しません。各要素で Red と Green の両方について一致を確認しますが、値 Red と値 Green の両方を持つことができる要素がないためです。この述語は、2 要素の値 Red Green とは一致しません。

- ・ (%VALUE='Red' AND %KEY=2) は、Red Green、Black Red、Green Yellow Red と一致します。
- ・ (%VALUE='Red' OR %KEY=2) は、Red、Red Green、Black Red、Green Yellow Red、Green Black、Black Yellow と一致します。

FOR SOME %ELEMENT はコレクション述語です。これは、[述語条件](#)を指定できるコンテキストのほとんどで使用できます。詳細は、["述語の概要"](#)を参照してください。これは以下の制限に従います。

- ・ FOR SOME %ELEMENT は、HAVING 節では使用できません。
- ・ FOR SOME %ELEMENT は、JOIN 操作のフィールドを選択する述語として使用することはできません。
- ・ OR 論理演算子を使用して FOR SOME %ELEMENT を別の述語条件に関連付けることは、2 つの述語が異なるテーブル内のフィールドを参照している場合はできません。例えば、以下のような場合です。

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple') OR t2.Age < 65
```

この制限はオプティマイザがインデックスを使用する方法に依存するので、SQL はこの制限を、インデックスがテーブルに追加されるときにのみ実施できます。このタイプの論理は、すべてのクエリで使用しないことを強くお勧めします。

- ・ [シャード・テーブル](#)に対してクエリを実行するときに FOR SOME %ELEMENT を使用することはできません。["シャード・クラスタにおけるクエリ"](#)を参照してください。

## コレクション・インデックス

FOR SOME %ELEMENT の重要な用途は、コレクション・インデックスを使用して要素を選択することです。適切な KEYS インデックスまたは ELEMENTS インデックスを field に対して定義すると、InterSystems IRIS はフィールド値の要素を直接参照するのではなく、そのインデックスを使用します。

以下のコレクション・インデックスが定義されているとします。

### Class Member

```
INDEX fcIDX1 ON FavoriteColors(ELEMENTS);
```

以下のクエリは上記のインデックスを使用します。

### SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
```

以下のコレクション・インデックスが定義されているとします。

### Class Member

```
INDEX fcIDX2 ON FavoriteColors(KEYS) [ Type = bitmap ];
```

以下のクエリは上記のインデックスを使用します。

### SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%KEY=2)
```

コレクション・インデックスを使用した FOR SOME %ELEMENT の詳細は、["コレクションのクエリ"](#)を参照してください。

## 引数

### field

その要素を predicate と比較するスカラ式 (通常はデータ列)。

### AS e-alias

predicate 内の %KEY または %VALUE を修飾するために使用する要素のエイリアス (オプション)。通常、このエイリアスは、入れ子になった FOR SOME %ELEMENT 条件が predicate に含まれる場合に使用します。エイリアスは有効な識別子である必要があります。AS キーワードはオプションです。

### (predicate)

括弧で囲まれた述語条件。この条件内では、条件一致の内容を決定するために、%VALUE または %KEY (あるいはその両方) を使用します。%VALUE は要素値 (%VALUE='Red') と一致します。%KEY は要素の最小数 (%KEY=2) と一致します。また、e-alias を指定している場合、この条件でオプションとして %VALUE および %KEY を修飾することができます。この述語は、論理演算子 AND および OR を使用して、複数の条件式で構成することもできます。

## 例

以下の例は、FOR SOME %ELEMENT を使用して、FavoriteColors のリストに 'Red' 要素が含まれる行を返します。

### SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
```

以下の例では、述語 %VALUE にはコンマ区切りリストを指定する IN 文が含まれます。この例は、FavoriteColors のリストに要素 'Red' または要素 'Blue' (または両方) が含まれる行を返します。

### SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE IN ('Red','Blue'))
```

以下の例は、2 つの包含関係演算子 (I) と共に述語節を使用しています。これは、FavoriteColors のリストに小文字 'l' と小文字 'e' (包含関係演算子は小文字と大文字が区別されます) を含む要素がある行を返します。この場合、要素は 'Blue' や 'Yellow'、'Purple' があります。

### SQL

```
SELECT Name,FavoriteColors AS Preferences
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) AS fc (fc.%VALUE [ 'l' AND fc.%VALUE [ 'e']
```

この例では、要素のエイリアス (e-alias) の使用方法についても示しています。

以下のダイナミック SQL の例では、%KEY を使用して、FavoriteColors 内の要素の数に基づいて行が返されます。最初の %Execute() は %KEY=1 を設定し、1 つ以上の FavoriteColors 要素を持つすべての行を返します。2 番目の %Execute() は %KEY=2 を設定し、2 つ以上の FavoriteColors 要素を持つすべての行を返します。

## ObjectScript

```
SET q1 = "SELECT %ID,Name,FavoriteColors FROM Sample.Person "  
SET q2 = "WHERE FOR SOME %ELEMENT(FavoriteColors) (%KEY=?) "  
SET myquery = q1_q2  
SET tStatement = ##class(%SQL.Statement).%New()  
SET tStatement.%SelectMode=1  
SET qStatus = tStatement.%Prepare(myquery)  
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}  
SET rset = tStatement.%Execute(1)  
DO rset.%Display()  
WRITE !,"End of data %KEY 1",!!  
SET rset = tStatement.%Execute(2)  
DO rset.%Display()  
WRITE !,"End of data %KEY 2"
```

## 関連項目

- ・ [SELECT 文、WHERE 節](#)
- ・ [述語の概要](#)
- ・ [FOR SOME 述語](#)

## IN (SQL)

構造化されていないコンマ区切りリスト内の項目に値を一致させます。

### 構文

```
scalar-expression IN (item1,item2[,...])
scalar-expression IN (subquery)
```

### 引数

引数	説明
scalar-expression	その値を値のコンマ区切りリストまたは subquery で生成された結果セットと比較するスカラ式 (通常はデータ列)。
item	1 つ以上のリテラル値、入力ホスト変数、またはリテラル値に解決される式。任意の順序でリストし、コンマで区切ります。
subquery	括弧で囲まれたサブクエリで、これによって 1 つの列から返される結果セットが scalar-expression との比較に使用されます。

### 説明

IN 述語は、構造化されていない一連の項目に値を一致させるために使用されます。一般に、これは、列データの値を値のコンマ区切りリストと比較します。IN は、[等値比較](#)と[サブクエリ比較](#)を実行できます。

ほとんどの述語と同様に、IN は NOT 論理演算子を使用して反転できます。IN も NOT IN も、NULL フィールドを返すために使用することはできません。NULL フィールドを返すには、[IS NULL](#) を使用します。

[述語条件](#)を指定できる場所であればどこでも、IN を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

### 等値比較

IN 述語は、OR 演算子で複数の等値比較を結合する省略表現として使用します。以下はその例です。

#### SQL

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT')
```

上記の文は、Home\_State がコンマ区切りリスト内のいずれかの値と等しい場合、True と評価します。リストされる項目には、定数または式を指定できます。

IN 比較では、個々の item の照合タイプに関係なく、scalar-expression に対して定義された[照合タイプ](#)が使用されます。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

以下の 2 つの例では、照合のマッチングが scalar-expression 照合に基づいていることが示されています。Home\_State フィールドは、(大文字と小文字が区別されない) SQLUPPER 照合で定義されます。したがって、以下の例では、NH Home\_State 値が返されます。

#### SQL

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('ME','nH','VT')
```

以下の例では、NH Home\_State 値は返されません。



## SQL

```
SELECT Name, Home_State FROM Sample.Person
WHERE %EXACT(Home_State) IN ('ME','nH','VT')
```

値のリストに NULL を含めることに意味はありません。NULL は値なしであるため、すべての等式テストに失敗します。IN 述語 (または他の任意の述語) を指定すると、指定されたフィールドのインスタンスのうち NULL であるものがすべて排除されます。これについては、以下の 正しくない (が実行可能な) 例で示しています。

## SQL

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors IN ($LISTBUILD('Red'),$LISTBUILD('Blue'),NULL)
/* NULL here is meaningless. No FavoriteColor NULL fields returned */
```

NULL のフィールドを述語の結果セットに含めるための唯一の方法は、以下の例のように IS NULL 述語を指定することです。

## SQL

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors IN ($LISTBUILD('Red'),$LISTBUILD('Blue')) OR FavoriteColors IS NULL
```

IN 述語の等値比較に日付または時刻を使用すると、適切なデータ型変換が自動的に実行されます。WHERE フィールドが TimeStamp 型の場合、Date 型または Time 型の値は Timestamp に変換されます。WHERE フィールドが Date 型の場合、TimeStamp 型または String 型の値は Date に変換されます。WHERE フィールドが Time 型の場合、TimeStamp 型または String 型の値は Time に変換されます。

以下の 2 つの例は、同じ等値比較を実行し、同じデータを返します。DOB フィールドは Date データ型です。

## SQL

```
SELECT Name,DOB FROM Sample.Person
WHERE DOB IN ({d '1951-02-02'},{d '1987-02-28'})
```

## SQL

```
SELECT Name,DOB FROM Sample.Person
WHERE DOB IN ({ts '1951-02-02 02:37:00'},{ts '1987-02-28 16:58:10'})
```

詳細は、“[日付/時刻文](#)”を参照してください。

## %SelectMode

論理形式ではない値に %SelectMode が設定されている場合は、IN 述語値を %SelectMode 形式 (ODBC または表示) で指定する必要があります。このことは主に、日付、時刻、および InterSystems IRIS 形式のリスト (%List) に当てはまります。論理形式で述語値を指定すると、一般に、SQLCODE エラーが発生します。例えば、SQLCODE -146 “[入力された日付を妥当な日付論理値に変換できません]” が返されます。

以下のダイナミック SQL の例の IN 述語では、%SelectMode=1 (ODBC) 形式で日付を指定する必要があります。

## ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB IN('1956-03-05','1956-04-08','1956-04-18','1956-12-08')"
```

```
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

## サブクエリ比較

サブクエリで IN 述語を使用し、列の値 (あるいは他の式) がサブクエリの行の値と等しいかどうかをテストできます。以下に例を示します。

### SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Name IN
      (SELECT Name FROM Sample.Employee
       HAVING Salary < 50000)
```

サブクエリでは、SELECT リスト内に必ず 1 つの選択項目を持ちます。

以下の例では、IN サブクエリを使用して、Vendor の状態ではない Employee の状態を返します。

### SQL

```
SELECT Home_State
FROM Sample.Employee
WHERE Home_State NOT IN (SELECT Address_State FROM Sample.Vendor)
GROUP BY Home_State
```

次の例では、照合関数表現をサブクエリと共に使用される IN 述語と照合しています。

### SQL

```
SELECT Name,Id FROM Sample.Person
WHERE %EXACT(Spouse) NOT IN
      (SELECT Id FROM Sample.Person
       WHERE Age < 65)
```

IN は、サブクエリと、リテラル値のコンマ区切りリストの両方を指定することはできません。

## リテラル置換のオーバーライド

各 IN 述語引数を括弧で囲むことで、コンパイルの事前解析時にリテラル置換をオーバーライドできます。例えば、WHERE Home\_State IN (('ME'),('NH'),('VT'),('MA'),('RI'),('CT')) のように指定します。これにより、全体的な選択性や添え字境界の選択性が向上し、クエリのパフォーマンスが向上する場合があります。ただし、同じクエリを別の値で複数回呼び出すことは避けてください。それぞれのクエリ呼び出しでキャッシュされたクエリが個別に作成されることになってしまいます。

## IN および %INLIST

IN 述語と %INLIST 述語はどちらも、OR 等値比較に使用する複数の値を指定するために使用できます。%INLIST 述語は、値を %List 構造の要素に一致させるために使用します。ダイナミック SQL では、%INLIST 述語の値は、単一のホスト変数として指定できます。IN 述語の値は、個別のホスト変数として指定する必要があります。したがって、IN 述語の値の数を変更すると、別個のクエリ・キャッシュが作成されることになります。%INLIST には、複数の要素を持つ単一の述語値 %List を指定します。%List 要素の数を変更しても、別個のクエリ・キャッシュは作成されません。%INLIST はまた、桁数の SIZE 引数も提供し、SQL はこれを使用してパフォーマンスを最適化します。このため、多くの場合は IN(val1,val2,val3,..valn) ではなく %INLIST(\$LISTFROMSTRING(val)) を使用した方が有利です。

%INLIST では等値比較を実行できますが、サブクエリ比較は実行できません。

詳細は、%INLIST を参照してください。

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [%INLIST 述語](#)
- ・ [述語の概要](#)

## %INLIST (SQL)

値を %List 構造化リストの要素と一致させます。

### 構文

```
scalar-expression %INLIST list [SIZE ((nn))]
```

### 引数

引数	説明
scalar-expression	値を list 要素と比較するスカラー式 (通常はデータ列)。
list	1 つまたは複数の要素を含んだ %List 構造。
SIZE ((nn))	オプション - list 内の要素数の桁を指定する整数。10、100、1000、10000、といった値のいずれかでリテラルとして指定する必要があります。

### 説明

%INLIST 述語は、フィールドの値をリスト構造の要素に一致させるための、InterSystems IRIS の拡張機能です。%INLIST と IN はどちらも、指定した複数の値でこのような等値比較を実行できます。%INLIST は、これらの複数の値を、単一の list 引数の要素として指定します。したがって、%INLIST では、別のクエリ・キャッシュを作成することなく、マッチングする値の数を変更できます。

オプションの %INLIST SIZE 節では、整数 nn を提供し、これによって list 内のリスト要素数の桁を指定します。InterSystems IRIS はこの桁数を使用して、最適なクエリ・プランを決定します。list 内の要素の数に関係なく同じクエリ・キャッシュを使用できるので、SIZE を指定すると、list 内の要素の想定される概数に合わせて最適化されたクエリ・キャッシュを作成できます。SIZE リテラルを変更すると、別のクエリ・キャッシュが作成されます。nn は、10、100、1000、10000、その他のいずれかのリテラルで指定します。nn は、コンパイル時に定数値として使用可能であることが要求されるため、すべての SQL コードでリテラルとして指定する必要があります。コンパイルされたすべての SQL (ダイナミック SQL) で、次に示すように二重括弧を指定する必要があります。二重括弧は埋め込み SQL では使用されません。

%INLIST は、list の各要素との等値比較を実行します。%INLIST 比較では、scalar-expression に対して定義された[照合タイプ](#)が使用されます。したがって、list 要素の比較では、scalar-expression の照合に応じて、大文字と小文字が区別される場合も、されない場合もあります。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

NULL を比較値として指定することに意味はありません。NULL は値なしであるため、すべての等式テストに失敗します。%INLIST 述語 (または他の任意の述語) を指定すると、指定されたフィールドのインスタンスのうち NULL であるものがすべて排除されます。NULL のフィールドを述語の結果セットに含めるためには、IS NULL 述語を指定する必要があります。

ほとんどの述語と同様に、%INLIST は NOT 論理演算子を使用して反転できます。%INLIST も NOT %INLIST も、NULL フィールドを返すために使用することはできません。NULL フィールドを返すには、[IS NULL](#) を使用します。

マッチング式が %List 形式でない場合、%INLIST は SQLCODE -400 エラーを発行します。例えば、コレクション・プロパティの SqlListType が DELIMITED の場合は、リスト・フィールドの論理値は %List 形式ではありません。リスト構造の詳細は、SQL の [\\$LIST](#) 関数を参照してください。

[述語条件](#)を指定できる場所であればどこでも、%INLIST を使用できます。詳細は、[“述語の概要”](#) ページを参照してください。

コンマで区切られた値リストのように、構造化されていない一連の項目に値を結合するには、[IN](#) 述語を使用します。IN では、等値比較やサブクエリ比較を実行できます。

## %SelectMode

%INLIST 述語は、現在の %SelectMode 設定を使用しません。list の要素は、%SelectMode 設定に関係なく、論理形式で指定する必要があります。list 要素を ODBC 形式または表示形式で指定しようとすると、通常は、データが一致しないか、意図していないデータと一致することになります。

%EXTERNAL または %ODBCOUT 形式変換関数を使用すると、述語による操作対象となる scalar-expression フィールドを変換できます。これにより、表示形式または ODBC 形式で list 要素を指定できるようになります。ただし、形式変換関数を使用すると、そのフィールドのためのインデックスを使用できなくなり、それによってパフォーマンスに大きな影響を与える可能性があります。

以下のダイナミック SQL の例では、%INLIST 述語により、%SelectMode=1 (ODBC) 形式ではなく論理形式で、1978 年の日付値要素を含むリストを指定します。これらの \$HOROLOG 形式の日付に対応する日付が選択されます。

### ObjectScript

```
SET bday=$LISTBUILD(50039)
FOR i=50039:1:50403 {SET bday=bday_$LISTBUILD(i) }
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %INLIST ?"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(bday)
DO rset.%Display()
```

以下のダイナミック SQL の例では、%ODBCOUT 形式変換関数を使用して、述語によって、一致した DOB フィールドを変換します。これにより、%INLIST リスト要素を ODBC 形式で指定できるようになります。ただし、形式変換関数を指定すると、DOB フィールド値のためのインデックスを使用できなくなります。

### ObjectScript

```
SET births=$LISTBUILD("1978-01-15","1978-08-22","1978-10-01")
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %INLIST ?"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(births)
DO rset.%Display()
```

## %INLIST および IN

%INLIST および IN 述語はどちらも、等値比較に使用する複数の値を指定するために使用できます。以下の例は、同じ結果を返します。

### SQL

```
SELECT Name, Home_State
FROM Sample.Person
WHERE Home_State %INLIST $LISTBUILD('VT','NH','ME')
```

### SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE Home_State IN('VT','NH','ME')
```

ダイナミック SQL では、%INLIST 述語は単一のホスト変数として指定できますが、IN 述語は個別のホスト変数として指定する必要があります。したがって、IN 述語の値の数を変更すると、別個のクエリ・キャッシュが作成されることとなります。%INLIST 述語の値の数を変更しても、別のクエリ・キャッシュは作成されません。詳細は、“[クエリ・キャッシュ](#)”を参照してください。

## 例

以下の 2 つの例では、照合のマッチングが scalar-expression 照合に基づいていることが示されています。Home\_State フィールドは、大文字と小文字が区別されない SQLUPPER 照合で定義されます。これらの例の list は、New Hampshire を “NH” ではなく “nH” と指定しています。最初の例では NH Home\_State 値が返されますが、2 番目の例では NH Home\_State 値が返されません。

### SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE Home_State %INLIST $LISTBUILD("VT","nH","ME")
```

### SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE %EXACT(Home_State) %INLIST $LISTBUILD("VT","nH","ME")
```

以下の例では、SIZE リテラルが 10 のクエリ・キャッシュを作成します。10 は桁数でリスト内の要素の実数に対応するので、SIZE 10 の指定はこのクエリには最適です。リスト内の要素の数を変更しても、別のクエリ・キャッシュは作成されません。SIZE リテラルを変更すると、別のクエリ・キャッシュが作成されます。

### SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE Home_State %INLIST $LISTBUILD("VT","NH","ME") SIZE ((10))
```

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [\\$LISTBUILD 関数](#)
- ・ [IN 述語](#)
- ・ [述語の概要](#)

## %INSET (SQL)

値を一連の生成値とマッチングします。

### 構文

```
scalar-expression %INSET valueset [SIZE ((nn))]
```

### 説明

%INSET 述語では、valueset に指定された値と一致するデータ値を選択することで、結果セットをフィルタ処理できます。この一致は、scalar-expression の値が valueset 内の値と一致したときに成功します。valueset の値がスカラ式のいずれの値とも一致しない場合、%INSET は NULL 文字列を返します。表示モードに関係なく、この照合は常に、論理 (内部保存) データ値に対して実行されます。

%INSET は NULL 値に対して真になることはありません。したがって、valueset に NULL が入った scalar-expression の NULL には一致しません。

%INSET は、他の比較条件と同様に、SELECT 文の WHERE 節または HAVING 節の中で使用します。

%INSET では、抽象的な、プログラムで指定された一致する値のセットを使用して、フィールド値をフィルタ処理できます。具体的には、抽象的な、プログラムで指定された一時ファイルまたはビットマップ・インデックスを使用して、RowId フィールドの値をフィルタ処理できます。その場合、valueset はビットマップ・インデックスまたは通常のインデックスの最下位の添え字の層と同じように機能します。

このユーザ定義クラスは、抽象クラス %SQL.AbstractFind から派生します。この抽象クラスには、%INSET でサポートされる唯一のメソッドである ContainsItem() メソッドが定義されています。ContainsItem() メソッドは、valueset を返します。詳細は %SQL.AbstractFind を参照してください。

### 照合タイプ

%INSET はマッチングの対象となる列と同じ照合タイプを使用します。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。現在のネームスペースにおける既定の文字列の照合を定義し、フィールド/プロパティの定義における既定以外のフィールドの照合タイプを指定することができます。列に別の照合タイプが割り当てられている場合、その照合タイプを %INSET substring にも適用する必要があります。ケース変換関数の詳細は、“%SQLUPPER” を参照してください。

### SIZE 節

オプションの %INSET SIZE 節には整数 nn が提供され、これによって valueset 内の値の桁数が指定されます。InterSystems IRIS はこの桁数を使用して、最適なクエリ・プランを決定します。nn は、10、100、1000、10000、その他のいずれかのリテラルで指定します。nn は、コンパイル時に定数値として使用可能であることが要求されるため、すべての SQL コードでリテラルとして指定する必要があります。埋め込み SQL を除くすべての SQL で、示されるように入れ子の括弧を指定する必要があります。

### %INSET と %FIND の比較

- ・ %INSET は、最も単純で最も一般的なインタフェースです。ContainsItem() メソッドをサポートします。
- ・ %FIND は、ビットマップ・インデックスを使ったビットマップ・チャンクの反復処理をサポートします。ObjectScript の \$ORDER 関数の機能をエミュレートし、NextChunk()、PreviousChunk()、GetChunk() の各反復処理メソッドと、ContainsItem() メソッドをサポートします。
- ・

## 引数

### scalar-expression

その値を valueset と比較するスカラ式 (通常はテーブルの RowId フィールド)。

### valueset

ContainsItem() メソッドを実装するユーザ定義オブジェクトへのオブジェクト参照 (oref)。このメソッドは、一連のデータ値を取り、scalar-expression 内の値との一致があったときにブーリアン値を返します。

### SIZE ((nn))

クエリの最適化に使用される、桁数を表す整数 (10、100、1000 など) (オプション)。

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [%FIND 述語](#)
- ・ [述語の概要](#)



## IS JSON (SQL)

---

データ値が JSON 形式であるかどうか判別します。

### 構文

```
scalar-expression IS [NOT] JSON [keyword]
```

### 説明

IS JSON 述語は、データ値が JSON 形式であるかどうか判別します。

IS JSON は、(オプションの VALUE キーワードの有無を問わず) JSON 配列または JSON オブジェクトに対して True を返します。これには、空の JSON 配列 '[]' や空の JSON オブジェクト '{}' が含まれます。

VALUE キーワードと SCALAR キーワードは同義語です。

詳細は、ObjectScript SET コマンドのサブセクション “[JSON オブジェクトと JSON 配列](#)” を参照してください。

IS NOT JSON 述語は、WHERE 節の[ストリーム・フィールド](#)で使用可能な述語の 1 つです。その動作は IS NOT NULL と同じです。

[述語条件](#)を指定できる場所であればどこでも、IS JSON を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

### 引数

#### scalar-expression

JSON 形式であるかどうか確認するスカラ式。

#### keyword

オプションの引数です。VALUE、SCALAR、ARRAY、または OBJECT のいずれか。既定値は VALUE です。

### 例

以下の例では、述語が適切な形式の JSON 文字列 (JSON オブジェクトまたは JSON 配列) であるかどうかを判別しています。

#### SQL

```
SELECT TOP 5 Name FROM Sample.Person
WHERE '{"name":"Fred","spouse":"Wilma"}' IS JSON
```

IS JSON ARRAY は、JSON 配列の oref に対して True を返します。IS JSON OBJECT は、JSON オブジェクトの oref に対して True を返します。詳細は、以下の例を参照してください。

#### SQL

```
SET jarray=
WRITE "JSON array: ",jarray,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON ARRAY"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jarray)
DO rset.%Display()
```

## ObjectScript

```

SET jarray=[1,2,3,5,8,13,21,34]
WRITE "JSON array: ",jarray,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON OBJECT"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jarray)
DO rset.%Display()

```

## ObjectScript

```

SET jobj={"name":"Fred","spouse":"Wilma"}
WRITE "JSON object: ",jobj,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON OBJECT"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jobj)
DO rset.%Display()

```

以下の例は、IS NOT JSON 述語の動作を示しています。

## SQL

```

SELECT Title,%OBJECT(Picture) AS PhotoOref FROM Sample.Employee
WHERE Picture IS NOT JSON

```

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [JSON\\_ARRAY 関数、JSON\\_OBJECT 関数](#)
- ・ [JSON\\_ARRAYAGG 集約関数](#)
- ・ [述語の概要](#)

## IS NULL (SQL)

---

データ値が NULL かどうか判別します。

### 構文

```
scalar-expression IS [NOT] NULL
```

### 説明

IS NULL 述語は、未定義の値を検出します。すべての NULL 値またはすべての NULL でない値を検出できます。

#### SQL

```
SELECT Name, FavoriteColors FROM Sample.Person  
WHERE FavoriteColors IS NULL
```

#### SQL

```
SELECT Name, FavoriteColors FROM Sample.Person  
WHERE FavoriteColors IS NOT NULL
```

IS NULL/IS NOT NULL 述語は、WHERE 節の[ストリーム・フィールド](#)で使用可能な述語の 1 つです。詳細は、以下の例を参照してください。

#### SQL

```
SELECT Title,%OBJECT(Picture) AS PhotoOref FROM Sample.Employee  
WHERE Picture IS NOT NULL
```

[述語条件](#)を指定できる場所であればどこでも、IS NULL を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

IS NULL 述語を SQL [ISNULL](#) 関数と混同しないでください。

### 関連項目

- ・ [SELECT 文](#)、[HAVING 節](#)、[WHERE 節](#)
- ・ [述語の概要](#)

## LIKE (SQL)

値をリテラルおよびワイルドカードを含むパターン文字列とマッチングします。

### 構文

```
scalar-expression LIKE pattern [ESCAPE char]
```

### 引数

引数	説明
scalar-expression	値を pattern と比較するスカラ式 (通常はデータ列)。
pattern	scalar-expression の各値とマッチングする文字パターンを表す、引用符付きの文字列。pattern 文字列には、リテラル文字、アンダースコア ()、およびパーセント (%) ワイルドカード文字を含めることができます。
ESCAPE char	Optional - 1 文字を含む文字列。この char 文字は、pattern 内でその直後の文字をリテラルとして処理するよう指定するために使用されます。

### 概要

LIKE 述語を使用して、pattern で指定する 1 つ以上の文字と一致するデータの値を選択できます。pattern にはワイルドカード文字を含めることができます。pattern がスカラ式のいずれの値とも一致しない場合、LIKE は NULL 文字列を返します。

[述語条件](#)を指定できる場所であればどこでも、LIKE を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

LIKE 述語は、以下のワイルドカードをサポートします。

テーブル D-1: LIKE ワイルドカード文字

文字	以下と一致
-	単独の文字。
%	0 かそれ以上の文字のシーケンス (SQL 標準に従い、NULL は 0 文字のシーケンスとは見なされず、このワイルドカードによって選択されません)。

“例” のセクションに示されているように、ダイナミック SQL または埋め込み SQL では、pattern を使用して、ワイルドカード文字と入力パラメータまたは入力ホスト変数を連結文字列として表すことができます。

注釈     実行時に述語の値を指定した場合 (? [入力パラメータ](#)または [:var 入力ホスト変数](#)を使用)、その結果の述語 %STARTSWITH 'abc' は、同等の結果の述語 LIKE 'abc%' よりもパフォーマンスが優れています。

### 照合タイプ

pattern 文字列はマッチングの対象となる列と同じ[照合タイプ](#)を使用します。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。クエリに ESCAPE char 節が含まれる場合、エスケープ処理は照合の後に実行されます。

既定の照合タイプ SQLUPPER のフィールドに LIKE が適用された場合、LIKE 節は大文字と小文字を無視して一致するものを返します。SQLSTRING 照合タイプを使用することで、大文字と小文字を区別する LIKE の文字列比較を実行できます。

以下の例は、部分文字列“Ro”を含むすべての名前を返します。LIKE は大文字と小文字を区別しないため、LIKE '%Ro%' は、Robert、Rogers、deRocca、LaRonga、Brown、Mastroniなどを返します。

## SQL

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '%Ro%'
```

これを[包含関係演算子](#) (D) と比較します。これは、EXACT (大文字と小文字を区別する) 照合を使用します。

## SQL

```
SELECT Name FROM Sample.Person
WHERE Name [ 'Ro'
```

%SQLSTRING 照合タイプを使用することで、LIKE を使用して、大文字と小文字が区別される部分文字列“Ro”を含む名前のみを返すようにすることができます。これは、Mastroni または Brown は返しません。

## SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING(Name) LIKE '%Ro%'
```

上記の例で、%SQLSTRING が Name の値に追加した先行のスペースは、ワイルドカード % によって処理されました。さらに堅牢な例では、照合タイプを述語の両側に指定します。

## SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING(Name) LIKE %SQLSTRING('%Ro%')
```

ケース変換関数の詳細は、“[%SQLUPPER](#)”を参照してください。

## すべての値、空の文字列値、および NULL

pattern 値がパーセント (%) である場合、LIKE は、空文字列値を含む、指定されたフィールドのすべての値を選択します。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE '%'
```

NULL であるフィールドは選択されません。

空文字列の pattern 値を指定すると、空文字列値が返されます。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE ''
```

NULL の pattern 値を指定すると、意味のある操作ではなくなります。操作は正常に完了しますが、値が返されません。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE NULL
```

ほとんどの述語と同様に、LIKE は NOT 論理演算子を使用して反転できます。LIKE も NOT LIKE も、NULL フィールドを返すために使用することはできません。NULL フィールドを返すには、[IS NULL](#) を使用します。

## ESCAPE 節

ESCAPE では、pattern 内でワイルドカード文字をリテラル文字として使用できます。ESCAPE char が指定され、単一文字である場合、これは、pattern 内の直接の文字を、ワイルドカードや書式設定文字ではなく、リテラル文字として解釈するように指示します。以下の例では、ESCAPE を使用して、文字列 '\_SYS' を含む値を返しています。

### SQL

```
SELECT * FROM MyTable
WHERE symbol_field LIKE '%\_SYS%' ESCAPE '\'
```

### %SelectMode

LIKE 述語は、現在の %SelectMode 設定を使用しません。pattern は、%SelectMode 設定に関係なく、論理形式で指定する必要があります。pattern を ODBC 形式または表示形式で指定しようとする、通常は、データが一致しないか、意図していないデータと一致することになります。

%EXTERNAL または %ODBCOUT 形式変換関数を使用すると、述語による操作対象となる scalar-expression フィールドを変換できます。これにより、表示形式または ODBC 形式で pattern を指定できるようになります。ただし、形式変換関数を使用すると、そのフィールドのためのインデックスを使用できなくなり、それによってパフォーマンスに大きな影響を与える可能性があります。

以下のダイナミック SQL の例では、LIKE 述語で、%SelectMode=1 (ODBC) の形式ではなく論理形式で日付の pattern を指定します。41 で始まる DOB 論理値がある行 (1953 年 4 月 4 日 (\$HOROLOG 41000) から 1955 年 12 月 28 日 (\$HOROLOG 41999) までの日付) が選択されます。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB LIKE '41%'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下の例では、%ODBCOUT 形式変換関数を使用して、述語によって、一致した DOB フィールドを変換します。これにより、ODBC 形式で LIKE pattern を指定できるようになります。それにより、DOB フィールドの ODBC 値が 195 で始まる行 (1950 年から 1959 年の範囲内の日付) が選択されます。ただし、形式変換関数を指定すると、DOB フィールド値のためのインデックスを使用できなくなります。

### SQL

```
SELECT Name,DOB FROM Sample.Person
WHERE %ODBCOUT(DOB) LIKE '195%'
```

## リテラル置換のオーバーライド

LIKE 述語引数を二重の括弧で囲むことで、コンパイルの事前解析時にリテラル置換をオーバーライドできます。例えば、WHERE Name LIKE (('Mc%')) や WHERE Name LIKE (('son%')) のように指定します。これにより、全体的な選択性や添え字境界の選択性が向上し、クエリのパフォーマンスが向上する場合があります。ただし、同じクエリを別の値で複数回呼び出すことは避けてください。それぞれのクエリ呼び出しでキャッシュされたクエリが個別に作成されることになってしまいます。

## 例

以下の例では、WHERE 節を使用して、“son”を含む Name の値を選択します。これには先頭または末尾が“son”のものも含まれます。既定では、LIKE の文字列比較は大文字と小文字が区別されません。

## SQL

```
SELECT %ID,Name FROM Sample.Person
WHERE Name LIKE '%son%'
```

以下の埋め込み SQL の例は、前の例と同じ結果セットを返します。LIKE pattern で連結演算子を使用して入力ホスト変数 (:subname) がどのように指定されているかに注意してください。

## ObjectScript

```
SET subname="son"
&sql(DECLARE C1 CURSOR FOR SELECT %ID,Name INTO :id,:nameout FROM Sample.Person
      WHERE Name LIKE '%_:subname_%')
&sql(OPEN C1)
      QUIT:(SQLCODE'=0)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
    WRITE id," ",nameout,!
    &sql(FETCH C1) }
&sql(CLOSE C1)
```

以下のダイナミック SQL の例は、前の例と同じ結果セットを返します。LIKE pattern で連結演算子を使用して入力パラメータ (?) がどのように指定されているかに注意してください。

## ObjectScript

```
SET myquery = "SELECT %ID,Name FROM Sample.Person WHERE Name LIKE '%'_?_'%"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("son")
DO rset.%Display()
```

以下の例では、WHERE 節を使用して、“blue”を含む FavoriteColors の値を選択しています。FavoriteColors フィールドは %List フィールドです。ワイルドカード % は、以下の %List 書式設定文字を処理します。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE '%blue%'
```

以下の例では、HAVING 節を使用して、年齢が 1 で始まりその後に 1 文字続く人のレコードを選択しています。すべての年齢の平均と HAVING 節で選択された年齢の平均が表示されます。結果は年齢順に並べ替えられます。年齢が 10 ～ 19 までのすべての値が返されます。

## SQL

```
SELECT Name,
       Age,
       AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgTeen
FROM Sample.Person
HAVING Age LIKE '1_'
ORDER BY Age
```

## 関連項目

- SELECT 文、HAVING 節、WHERE 節
- %MATCHES 述語
- %PATTERN 述語
- %STARTSWITH 述語
- 述語の概要



## %MATCHES (SQL)

値をリテラル、ワイルドカード、および範囲を含むパターン文字列とマッチングします。

### 構文

```
scalar-expression %MATCHES pattern [ESCAPE char]
```

### 説明

%MATCHES 述語は、値とパターン文字列とをマッチングするための InterSystems IRIS の拡張機能です。%MATCHES は、マッチング演算に対して True または False を返します。pattern 文字列は、リテラル文字列、ワイルドカード文字、およびマッチング・リテラルのリストまたは範囲で構成できます。

パターン・マッチでは大文字と小文字が区別されます。パターン・マッチは、照合値でなく、scalar-expression の EXACT 値に基づいています。したがって、scalar-expression の [照合タイプ](#) で大文字と小文字が区別されない場合でも、%MATCHES 処理では大文字と小文字が常に区別されます。

%MATCHES は、次の pattern ワイルドカードをサポートします。

?	任意の型の任意の 1 文字と一致。
*	任意の型の 0 文字以上の文字と一致。
[abc]	角括弧内に指定された文字のいずれかと一致。
[a-z]	角括弧内に指定された範囲内の文字 (範囲の最初の文字と最後の文字も含む) と一致。
[^A-Z] [^a-z] [^0-9]	角括弧内に指定された文字以外の任意の文と一致。この構文を使用して、大文字以外、小文字以外、数字以外などを指定できます。ここで示したリテラル範囲のみがサポートされます。
¥	直後の文字を、ワイルドカード文字としてではなく、リテラル文字として処理します。円記号が既定のエスケープ文字ですが、オプションの ESCAPE 節を使用して、別の文字をエスケープ文字として指定することもできます。

ほとんどの述語と同様に、%MATCHES は NOT 演算子を使用して反転できます。例えば、item NOT %MATCHES pattern のようにします。%MATCHES も NOT %MATCHES も、NULL フィールドを返すために使用することはできません。NULL フィールドを返すには、[IS NULL](#) を使用します。

円記号 (¥) 文字が既定のエスケープ文字です。このエスケープ文字を使用して、指定されたパターンの位置でワイルドカード文字をリテラル・マッチとして使用するよう指定できます。例えば、疑問符を文字列の最初の文字としてマッチさせるには、'¥?\*' と指定します。疑問符を文字列の 4 番目の文字としてマッチさせるには、'????¥?\*' と指定します。文字列の任意の場所にある疑問符をマッチさせるには、'\*¥?\*' と指定します。アスタリスク文字のみで構成される文字列をマッチさせるには、'¥\*' と指定します。少なくとも 1 つのアスタリスク文字を含む文字列をマッチさせるには、'\*¥\*\*' と指定します。文字列の任意の場所にある円記号をマッチさせるには、'\*¥¥\*' と指定します。

[述語条件](#)を指定できる場所であればどこでも、%MATCHESを使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

%MATCHES は、Informix SQL との互換性を保つためにサポートされています。

## %SelectMode

%MATCHES 述語は、現在の %SelectMode 設定を使用しません。pattern は、%SelectMode 設定に関係なく、論理形式で指定する必要があります。pattern を ODBC 形式または表示形式で指定しようとすると、通常は、データが一致しないか、意図していないデータと一致することになります。

%EXTERNAL または %ODBCOUT 形式変換関数を使用すると、述語による操作対象となる scalar-expression フィールドを変換できます。これにより、表示形式または ODBC 形式で pattern を指定できるようになります。ただし、形式変換関数を使用すると、そのフィールドのためのインデックスを使用できなくなり、それによってパフォーマンスに大きな影響を与える可能性があります。

以下のダイナミック SQL の例では、%MATCHES 述語で、%SelectMode=1 (ODBC) の形式ではなく論理形式で日付の pattern を指定します。41 で始まる DOB 論理値がある行 (1953 年 4 月 4 日 (\$HOROLOG 41000) から 1955 年 12 月 28 日 (\$HOROLOG 41999) までの日付) が選択されます。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %MATCHES '41*'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下のダイナミック SQL の例では、%ODBCOUT 形式変換関数を使用して、述語によって、一致した DOB フィールドを変換します。これにより、ODBC 形式で %MATCHES pattern を指定できるようになります。それにより、DOB フィールドの ODBC 値が 195 で始まる行 (1950 年から 1959 年の範囲内の日付) が選択されます。ただし、形式変換関数を指定すると、DOB フィールド値のためのインデックスを使用できなくなります。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %MATCHES '195*'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

## 引数

### scalar-expression

値を pattern と比較するスカラ式 (通常はデータ列)。

### string

scalar-expression の各値とマッチングする文字パターンを表す、引用符付きの文字列。pattern 文字列には、リテラル文字列、疑問符 (?) およびアスタリスク (\*) ワイルドカード、許可する値を指定するための角括弧、直後の文字をリテラルとして処理するよう指定するための円記号 (¥) を含めることができます。pattern には、空文字列または NULL を指定することもできますが、その場合はマッチングが行われないか NULL 項目が返されます。

### ESCAPE char

オプションの引数です。1 文字を含む文字列。この char 文字は、pattern 内でその直後の文字をリテラルとして処理するよう指定するために使用されます。指定されない場合、既定のエスケープ文字は円記号 (¥) です。

## 例

以下の例は、“A” で始まる姓をすべて返します。

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES 'A*'
```

以下の例は、“A” で始まる名前をすべて返します。

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '*,A*'
```

以下の例は、“A” という文字を含むフルネーム (姓、名前、ミドルネーム) をすべて返します。

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '*A*'
```

以下の例は、“A”、“a”、“E”、“e” を含まないフルネームをすべて返します。

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name NOT %MATCHES '*[AaEe]*'
```

以下の例は、“A” ～ “D” で始まる名前を持つ 5 文字の姓をすべて返します。

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '?????,[A-D]*'
```

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [LIKE 述語](#)
- ・ [%PATTERN 述語](#)
- ・ [述語の概要](#)

## %PATTERN (SQL)

値を、リテラル、ワイルドカード、および文字タイプ・コードを含むパターン文字列とマッチングします。

### 構文

```
scalar-expression %PATTERN pattern
```

### 概要

%PATTERN 述語を使用すると、文字タイプ・コードおよびリテラルのパターンを scalar-expression で指定されたデータの値と比較できます。pattern がスカラ式の値と完全に一致した場合、その値が返されます。pattern がスカラ式のいずれの値とも完全一致しない場合は、%PATTERN は NULL 文字列を返します。

[述語条件](#)を指定できる場所であればどこでも、%PATTERN を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

%PATTERN は、ObjectScript のパターン・マッチ演算子と同じパターン・コード (? 演算子) を使用します。パターンは、反復数とその直後の値のペアを 1 つ以上並べて構成します。反復数は、整数、ピリオド(.) (“任意の文字数” を意味します)、またはピリオドと整数を組み合わせる範囲を指定できます。値は文字タイプ・コードの文字か (引用符を付けた) リテラル文字列を指定できます。

パターンは全体のデータ値と正確にマッチングする必要があるために、反復数/値のペアを複数並べて構成する場合もあります。そのため、多くのパターンは “.E” のペアで終わります。これは、残りのデータの値は任意のタイプの任意の文字数で構成できることを意味します。

パターン・マッチング・ペアの簡単な例をいくつか示します。

- ・ 1L は、1 文字の小文字を意味します。
- ・ 1”L” は、1 文字のリテラル “L” を意味します。
- ・ 1”617” は、1 つのリテラル文字列 “617” を意味します。
- ・ .U は、任意の数の大文字を意味します。
- ・ .E は、任意のタイプの任意の数の表示可能文字を意味します。
- ・ .3A は、最大 3 文字まで (3 文字以下) の任意数の文字 (大文字または小文字) を意味します。
- ・ 3.N は、3 桁以上の数字を意味します。
- ・ 3.6N は、3 ～ 6 桁の数字を意味します。

パターン・マッチでは大文字と小文字が区別されます。パターン・マッチは、照合値でなく、scalar-expression の EXACT 値に基づいています。したがって、scalar-expression の[照合タイプ](#)で大文字と小文字が区別されない場合でも、%PATTERN 処理で指定されるリテラル文字は、大文字と小文字が常に区別されてマッチングされます。

ダイナミック SQL では、SQL クエリは二重引用符で区切り、ObjectScript 文字列として指定されます。このような理由から、pattern 文字列内の二重引用符を二重にする必要があります。したがって、米ドル額のためのパターン '1"\$1.N1"."2N' は、ダイナミック SQL では '1"\$1.N1"."2N' と指定されます。

パターン・コードの詳細は、“[パターン・マッチング \(?\)](#)” リファレンス・ページを参照してください。

### %SelectMode

%PATTERN 述語は、現在の [%SelectMode](#) 設定を使用しません。pattern は、%SelectMode 設定に関係なく、論理形式で指定する必要があります。pattern を ODBC 形式または表示形式で指定しようとする、通常は、データが一致しないか、意図していないデータと一致することになります。

**%EXTERNAL** または **%ODBCOUT** 形式変換関数を使用すると、述語による操作対象となる scalar-expression フィールドを変換できます。これにより、表示形式または ODBC 形式で pattern を指定できるようになります。ただし、形式変換関数を使用すると、そのフィールドのためのインデックスを使用できなくなり、それによってパフォーマンスに大きな影響を与える可能性があります。

以下のダイナミック SQL の例では、%PATTERN 述語で、%SelectMode=1 (ODBC) の形式ではなく論理形式で日付の pattern を指定します。41 で始まる DOB 論理値がある行 (1953 年 4 月 4 日 (\$HOROLOG 41000) から 1955 年 12 月 28 日 (\$HOROLOG 41999) までの日付) が選択されます。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %PATTERN '1""41""3N' "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下のダイナミック SQL の例では、**%ODBCOUT** 形式変換関数を使用して、述語によって、一致した DOB フィールドを変換します。これにより、ODBC 形式で %PATTERN pattern を指定できるようになります。それにより、DOB フィールドの ODBC 値が 195 で始まる行 (1950 年から 1959 年の範囲内の日付) が選択されます。ただし、形式変換関数を指定すると、DOB フィールド値のためのインデックスを使用できなくなります。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %PATTERN '1""195""E' "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

## 引数

### scalar-expression

値を pattern と比較するスカラ式 (通常はデータ列)。

### pattern

scalar-expression の各値とマッチングする文字パターンを表す、引用符付きの文字列。pattern 文字列には、二重引用符で囲まれたリテラル文字、文字のタイプを指定する文字コード、およびワイルドカード文字としての数字およびピリオド (.) 文字を含めることができます。

## 例

以下の例は、%PATTERN 演算子を WHERE 節の中で使用して、1 文字目が任意の大文字で 2 文字目が “C” である Home\_State の値を選択します。

### SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN '1U1"C'
```

この例では、Home\_State に North Carolina (NC) や South Carolina (SC) の値を持つレコードが選択されます。

以下の例は、%PATTERN 演算子を WHERE 節の中で使用して、1 文字目が大文字で 2 文字目が小文字である Name の値を選択します。

## SQL

```
SELECT Name FROM Sample.Person
WHERE Name %PATTERN '1U1L.E'
```

ここでのパターンは以下のように解釈されます: 1U (1 文字の大文字)、その後に 1L (1 文字の小文字)、その後に .E (任意のタイプの任意の数の文字)。このパターンでは、“JONES”、O'Reilly”、“deGastyne” などの名前は除外されます。

以下の例は、%PATTERN 演算子を HAVING 節の中で使用して、名前が “Jo” で始まる人のレコードを選択し、検索されたレコードと返されたレコードのカウントを返します。

## SQL

```
SELECT Name,
       COUNT(Name) AS TotRecs,
       COUNT(Name %AFTERHAVING) AS JoRecs
FROM Sample.Person
HAVING Name %PATTERN '1U.L1',"1"Jo".E'
```

ここでは、Name フィールドの値は Lastname,Firstname というフォーマットで、オプションでミドル・ネームやイニシャルが含まれる場合があります。この氏名のフォーマットを反映するために、ここでのパターンは以下のように解釈されます: 1U (1 文字の大文字)、その後に .L (任意の数の小文字)、その後に 1", (1 文字のリテラルのコンマ)、その後に 1"Jo" (値が “Jo”) の 1 つのリテラル文字列)、その後に .E (任意のタイプの任意の数の文字)。

## 関連項目

- ・ [SELECT 文、HAVING 節、WHERE 節](#)
- ・ [LIKE 述語](#)
- ・ [%MATCHES 述語](#)
- ・ [述語の概要](#)

# SOME (SQL)

値をサブクエリで取得された少なくとも 1 つの一致する値と照合します。

## 構文

```
scalar-expression comparison-operator SOME (subquery)
```

## 概要

SOME キーワードは、比較演算子と併用し、スカラ式の値が [subquery](#) で取得した対応する値の 1 つ以上と一致した場合に True となる [述語](#) (限定比較条件) を作成します。SOME 述語は、1 つの scalar-expression 項目とサブクエリの 1 つの SELECT 項目を比較します。SELECT 項目が複数あるサブクエリを実行すると、SQLCODE -10 エラーが生成されます。

注釈 キーワードの SOME および ANY は同義語です。

[述語条件](#)を指定できる場所であればどこでも、SOME を使用できます。詳細は、“[述語の概要](#)” ページを参照してください。

## 引数

### scalar-expression

その値を subquery で生成された結果セットと比較するスカラ式 (通常はデータ列)。

### comparison-operator

次の比較演算子のいずれか : = (等しい)、<> または != (等しくない)、< (より小さい)、<= (以下)、> (より大きい)、>= (以上)、[ (包含)、または ] (後続)。

### subquery

括弧で囲まれたサブクエリで、これによって返される結果セットが scalar-expression との比較に使用されます。

## 例

以下の例では、ミシシッピ川の西の州に住み、給与が \$75,000 よりも多い従業員を選択します。

### SQL

```
SELECT Name,Salary,Home_State FROM Sample.Employee
WHERE Salary > 75000
AND Home_State = SOME
  (SELECT State FROM Sample.USZipCode
   WHERE Longitude < -93)
ORDER BY Home_State
```

## 関連項目

- [SELECT 文](#)、[HAVING 節](#)、[WHERE 節](#)
- [ALLANY](#)
- [述語の概要](#)



## %STARTSWITH (SQL)

最初の文字を指定する部分文字列と値をマッチングします。

### 構文

```
column %STARTSWITH substring
```

### 概要

- `column %STARTSWITH substring` は、substring で指定された文字で始まる列からデータ値を選択します。substring がいずれの列値とも一致しない場合、%STARTSWITH は NULL 文字列を返します。%STARTSWITH は、設定されている表示モードに関係なく、この照合を列の論理内部保存値に対して実行します。

%STARTSWITH は、InterSystems SQL クエリの任意の述語条件で使用できます。述語条件の詳細は、“[述語の概要](#)”を参照してください。

以下の文は、M の文字で始まる名前をすべて選択します。

#### SQL

```
SELECT Name FROM Sample.MyTest WHERE Name %STARTSWITH 'M'
```

値を照合するその他の方法については、“[その他の等値比較](#)”を参照してください。

例：

- 最初の文字に基づいた列データの選択
- 論理演算子を使用した列データの選択

### 引数

#### column

substring と比較される値が含まれる、テーブル内のデータ列。この引数には、%EXTERNAL(*column*) や %SQLUPPER(*column*) など、列テーブルに評価されるスカラ式を指定することもできます。

#### substring

column に指定した値と照合する 1 つまたは複数の先頭文字。この引数には、文字列または数値に解決される式を指定する必要があります。

### 例

#### 最初の文字に基づいた列データの選択

%STARTSWITH 述語では、さまざまな文字列型および数値型を処理できます。

#### 文字

以下の文は、M の文字から始まる個別の Home\_State 名ごとに 1 行を返します。

#### SQL

```
SELECT DISTINCT Home_State FROM Sample.Person  
WHERE Home_State %STARTSWITH 'M'  
ORDER BY Home_State
```

既定の照合設定では、%STARTSWITH のマッチングで大文字と小文字は区別されません。このため、この文は “M” または “m” で始まる名前に一致します。マッチングで大文字/小文字の区別を制御する方法の詳細は、“[照合タイプに基づいた選択の大文字/小文字の区別の管理](#)” を参照してください。

## 数値

以下の文は、HAVING 節を使用して、年齢が 2 で始まる人の行を選択します。結果セットには、すべての年齢の平均と、HAVING 節で選択された年齢の平均が表示されます。結果は年齢順に並べ替えられます。

## SQL

```
SELECT Name,
       Age,
       AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS Avg20
FROM Sample.Person
HAVING Age %STARTSWITH 2
ORDER BY Age
```

## 日付

以下の文は、DOB(誕生日)フィールドの内部日付の形式値との %STARTSWITH 比較を実行します。この場合、11/5/1988 (\$H=54000) ～ 08/1/1991 (\$H=54999) のすべての日付を選択します。

## SQL

```
SELECT Name,DOB
FROM Sample.Person
WHERE DOB %STARTSWITH 54
ORDER BY DOB
```

## リスト

column に[リスト・コレクション](#)が含まれている場合、%STARTSWITH は、[%EXTERNAL](#) 形式変換関数を使用して、そのリスト値を substring と比較できます。例えば、以下の文は、FavoriteColors リスト列が 'Bl' で始まる行に対するマッチングを行います。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Bl'
```

リスト・コレクションでは、%EXTERNAL によってリストが DISPLAY 形式に変換されると、表示されるリスト項目は、空白スペースで区切られます。この“スペース”は実際には、CHAR(13)とCHAR(10)という2つの非表示文字です。%STARTSWITH をリスト内の複数の要素と共に使用するには、これらの文字を指定する必要があります。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Orange' || CHAR(13) || CHAR(10) || 'B'
```

column に [%Library.List](#) 型のデータが含まれている場合、[%Library.List](#) データは LOGICAL 形式で格納されており、別個の DISPLAY 形式はないため、%EXTERNAL を使用する必要はありません。リスト・データを照合するには、[\\$LIST](#) 関数を使用します。以下に例を示します。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH $LISTFROMSTRING('Yellow,Orange')
```

**注釈** InterSystems SQL では、リストを連結文字列として格納しているため、%STARTSWITH を使用してリストの途中にある要素を照合することはできません。%STARTSWITH は、リストの先頭からのみ照合できます。これは、リスト・コレクションと [%Library.List](#) データの両方に適用されます。

## 先頭および末尾の空白

ほとんどの場合、%STARTSWITH は先頭の空白をその他の文字と同様に処理します。例えば、%STARTSWITH ' B' は、先頭の 1 つの空白の後に文字 B が続く列値を選択します。ただし、substring に空白のみが含まれる場合は、NULL 以外の値が選択され、先頭の空白は選択されません。

末尾の空白に対する %STARTSWITH の動作は、データ型および照合タイプにより異なります。

- ・ %STARTSWITH では、SQLUPPER 照合で文字列 substring 内の末尾の空白が無視されます。
- ・ ただし、数値、日付、またはリストの substring 内の末尾の空白は無視されません。

以下の文では、%STARTSWITH は結果セットを M で始まる名前に制限しています。Name は SQLUPPER 文字列データ型であるため、substring の末尾の空白は無視されます。

### SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH 'M'
```

以下の文では、substring の末尾の空白は数値の場合無視されないため、%STARTSWITH は結果セットからすべての行を削除します。

### SQL

```
SELECT Name, Age FROM Sample.Person
WHERE Age %STARTSWITH '6'
```

以下の文では、substring の末尾の空白はリスト値の場合無視されないため、%STARTSWITH は結果セットからすべての行を削除します。

### SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Blue'
```

ただし、以下の文では、結果セットは、Blue で始まりリスト区切り文字 (空白として表示されます) が続くリスト値で構成されます。つまり、この文は、'Blue' で始まり、複数の項目を含むリストに一致します。

### SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Blue' || CHAR(13) || CHAR(10)
```

## 論理演算子を使用した列データの選択

%STARTSWITH 関数は、[論理演算子](#) NOT、AND、および OR をサポートします。

以下の文は、M 以外の文字で始まる名前をすべて選択します。

### SQL

```
SELECT Name FROM Sample.MyTest WHERE NOT Name %STARTSWITH 'M'
```

以下の文は、M または N で始まる名前をすべて選択します。

### SQL

```
SELECT Name FROM Sample.MyTest WHERE Name %STARTSWITH 'M' OR Name %STARTSWITH 'N'
```

以下の文は、名が M で始まり、姓が N で始まる名前をすべて選択します。

## SQL

```
SELECT FirstName,LastName FROM Sample.MyTest WHERE FirstName %STARTSWITH 'M' AND LastName %STARTSWITH 'N'
```

## フィルタでの NULL 値の除外

%STARTSWITH 関数では、[column](#) が NULL 以外のデータ値に評価され、[substring](#) が空の値の場合、%STARTSWITH は NULL 以外の column データを返します。この動作を利用して、NULL 以外の値をフィルタで除外できます。例えば、以下の文は、結果セットを FavoriteColors 列内の NULL 以外の値に制限します。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH NULL
```

空の substring とは、値が以下のいずれかの場合を言います。

- ・ NULL
- ・ CHAR(0)
- ・ 空の文字列 (’ ’)
- ・ 1 つの空白のみが含まれる文字列 (’ ’)
- ・ CHAR(32) (空白文字)
- ・ CHAR(9) (タブ文字)

以下の文は、前の文と同じ結果を返します。

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH ' '
```

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH '   '
```

## SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH CHAR(9)
```

column が NULL に評価され、substring が空の値の場合、%STARTSWITH は column からデータを返しません。

空白文字のみで構成される column 値を返すには、%EXACT 照合を使用する必要があります。リスト・フィールドから NULL を除外する際に、column に %EXTERNAL 照合タイプは使用されません。

%STARTSWITH の NULL および空文字列の動作は、複合 substring とは異なります。これは、NULL および空文字列の定義が原因です。値と NULL を連結すると、結果は NULL になります。値と空文字列を連結すると、結果はその値になります。詳細は、以下の例を参照してください。

## SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B' || NULL
/* Selects all non-null rows */
```

## SQL

```
SELECT Name, FavoriteColors
FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B' || ''
/* Selects all values that begin with B */
```

## 照合タイプに基づいた選択の大文字/小文字の区別の管理

%STARTSWITH はマッチングの対象となるフィールドと同じ照合を使用します。文字列データ型のフィールドは、大文字と小文字を区別しない SQLUPPER 照合で定義されるため、既定の %STARTSWITH 選択でも大文字と小文字は区別されません。例えば、以下の文は、“M” または “m” で始まる自宅の州に一致します。

## SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
WHERE Home_State %STARTSWITH 'M'
ORDER BY Home_State
```

異なる照合タイプを WHERE 節の列に割り当てると、この照合タイプで %STARTSWITH substring のリテラル値がマッチングされます。例えば、EXACT (大文字と小文字を区別する) 照合を使用するよう検索列 Home\_State を指定すると、%STARTSWITH は、“M” で始まる自宅の州のみにマッチングします。

## SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
WHERE %EXACT(Home_State) %STARTSWITH 'M'
ORDER BY Home_State
```

一部の照合関数は、フィールド値の先頭に空白文字を追加します。これにより、同等の照合関数を部分文字列に適用する場合を除いて、%STARTSWITH で値がマッチングされない可能性があります。

例えば、EXACT 照合を使用する列 ExactName が含まれるテーブルがあるとします。%STARTSWITH の column 引数内の ExactName に SQLUPPER 照合を適用すると、%STARTSWITH は、空白文字で始まる値が含まれる列を検索します。したがって、以下のような比較は行を返しません。

## SQL

```
SELECT ExactName FROM Sample.MyTest WHERE %SQLUPPER(ExactName) %STARTSWITH 'Ra'
```

この問題を解決するには、同じ照合関数を部分文字列に適用する方法で、substring の先頭に空白文字を追加する必要があります。以下の例では、大文字と小文字を区別しないマッチングが EXACT 列に適用されています。

## SQL

```
SELECT ExactName FROM Sample.MyTest WHERE %SQLUPPER(ExactName) %STARTSWITH %SQLUPPER('Ra')
```

照合の既定値の変更、または大文字小文字の変換関数の使用の詳細は、“[照合](#)” を参照してください。

## 詳細

### 添え字の範囲

添え字から column を取得した場合は、%STARTSWITH をインデックス制限の範囲条件として使用して、検索する必要がある column 添え字値の範囲を縮小できます。このロジックは、指定された substring 接頭語値で添え字の範囲を開始し、添え字値が substring で始まらなくなったらすぐに停止するというものです。

### その他の等値比較

%STARTSWITH は、文字列の先頭文字の等値比較を行います。文字列比較演算子を使用して、その他のタイプの等値比較を実行できます。これは、以下の項目が含まれます。

- ・ 等記号を使用した文字列全体の等値比較。

## SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State = 'VT'
```

この例では、Home\_State フィールドの値が“VT”のレコードが選択されます。Home\_State が SQLUPPER として定義されているため、この文字列比較では大文字と小文字は区別されません。

不等記号 (<>) を使用した文字列全体の不等値比較を実行することもできます。

- ・ [包含関係演算子](#)を使用した、部分文字列と値との等値比較。

## SQL

```
SELECT Name FROM Sample.Person
WHERE Name [ 'y'
```

この例では、小文字の“y”が含まれる Name フィールドのレコードがすべて選択されます。既定では、大文字と小文字を区別しないようにフィールドが定義されている場合でも、包含関係演算子の比較で大文字と小文字が区別されます。

- ・ [InterSystems SQL Search](#) を使用したコンテキスト認識等値比較。SQL Search の用途の 1 つは、指定された単語や語句が値に含まれているかどうかを判断することです。SQL Search では大文字と小文字が区別されません。
- ・ IN キーワード演算子を使用した、文字列全体の複数の値との等値比較。例えば、以下の文は、指定された Home\_State フィールドのいずれかの値が含まれるレコードをすべて選択します。

## SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State IN ( 'VT','MA','NH','ME' )
ORDER BY Home_State
```

- ・ %PATTERN キーワード演算子を使用した、文字列全体の値のパターンとの等値比較。

## SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN '1U1"C'
ORDER BY Home_State
```

この例では、1U (大文字 1 文字) の後に 1"C (“C” の 1 文字) が続くパターンと一致する Home\_State フィールドの値を含むすべてのレコードが選択されます。Home\_State の省略形 “NC” または “SC” がこのパターンを満たします。

- ・ LIKE キーワード演算子を使用した、1 つ以上のワイルドカードを含む部分文字列と値との等値比較。

## SQL

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '_a%'
```

この例では、2 文字目に “a” が含まれる Name フィールドのレコードがすべて選択されます。この文字列比較では、Name 照合タイプを使用して、比較で大文字と小文字が区別されるかどうか指定されます。

これらの詳細とその他の比較条件の述語については、“[WHERE](#)” 節を参照してください。

## %SelectMode 設定

%STARTSWITH 述語は、現在の [%SelectMode](#) 設定を使用できません。substring は、%SelectMode 設定に関係なく、論理形式で指定する必要があります。述語値 (1 つまたは複数) を ODBC 形式または表示形式で指定すると、通常は、

データが一致しないか、意図していないデータと一致することになります。このことは主に、日付、時刻、および InterSystems IRIS 形式のリスト (%List) に当てはまります。

以下の動的 SQL の例では、%STARTSWITH 述語で、%SelectMode=1 (ODBC) の形式ではなく論理形式で日付の substring を指定する必要があります。41 で始まる DOB 論理値がある行 (1953 年 4 月 4 日 (\$HOROLOGY 41000) から 1955 年 12 月 28 日 (\$HOROLOGY 41999) までの日付) が選択されます。

### ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "  
SET q2 = "WHERE DOB %STARTSWITH '41%'"  
SET myquery = q1_q2  
SET tStatement = ##class(%SQL.Statement).%New()  
SET tStatement.%SelectMode=1  
SET qStatus = tStatement.%Prepare(myquery)  
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}  
SET rset = tStatement.%Execute()  
DO rset.%Display()  
WRITE !,"End of data"
```

### 言語によってあいまい照合が行われる文字

言語によっては、2 文字または文字の組み合わせは、初回の照合の通過について等価と見なされます。これは一般的に、Czech2 ロケールなどにおける、アクセント符号の付く文字または付かない文字です。そのようなロケールでは、CHAR(65)と CHAR(193)は両方とも、“A”として照合されます。%STARTSWITH は、これらの文字を等価と認識します。

以下の例では、照合における初回通過時の Czech2 CHAR(65) (A) と CHAR(193) (Á) を示します。

```
M  
MA  
MÁ  
MAC  
MÁČ  
MACX  
MÁD  
MÁD  
MB
```

クエリをコンパイルする場合、実行時に使用される国固有照合は不明です。このため、実行時に起こりうるシナリオに対応できるよう、%STARTSWITH 添え字検索コードを記述する必要があります。

### 関連項目

- [SELECT 文、HAVING 節、WHERE 節](#)
- [述語の概要](#)
- [照合](#)



# SQL 集約関数

## 集約関数の概要

列のすべての値を評価し、単一の集約値を返す関数です。

### サポートされている集約関数

集約関数は、1 つの列内の 1 つ以上の値に対して処理を行い、1 つの値を返します。サポートされる関数は以下のとおりです。

- ・ **SUM** — 指定した列内の合計値を返します。
- ・ **AVG** — 指定した列の平均値を返します。
- ・ **COUNT** — テーブル内の行数、または指定した列内の NULL でない値の数を返します。
- ・ **MAX** — 指定した列内で使用されている最大値を返します。
- ・ **MIN** — 指定した列内で使用されている最小値を返します。
- ・ **VARIANCE**、**VAR\_SAMP**、**VAR\_POP** — 指定した列の値の統計的分散を返します。
- ・ **STDDEV**、**STDDEV\_SAMP**、**STDDEV\_POP** — 指定した列の値の統計的標準偏差を返します。
- ・ **LIST** — 指定した列で使用されているすべての値を、コンマで区切られたリストの形式で返します。
- ・ **%DLIST** — 指定した列で使用されているすべての値を、InterSystems IRIS リスト構造内の要素として返します。
- ・ **XMLAGG** — 指定した列で使用されているすべての値を、連結された文字列として返します。
- ・ **JSON\_ARRAYAGG** — 指定した列で使用されているすべての値を、JSON フォーマットの配列として返します。

**CREATE AGGREGATE** コマンドを使用して、追加のユーザ定義集約関数 (UDAF) を定義できます。

集約関数では、NULL のフィールドは無視されます。例えば、指定したフィールドが NULL だった場合は、LIST および %DLIST には、その行の要素は含まれません。COUNT では、指定したフィールドの非 NULL 値のみが数えられます。

集約関数 (COUNT を除く) は、**ストリーム・フィールド**に適用できません。これを実行しようすると、SQLCODE -37 エラーが生成されます。COUNT は、ストリーム・フィールド値のカウントに使用できますが、多少の制約があります。

**注釈** 集約関数は、**ウィンドウ関数**と似ています。しかし、集約関数は行のグループから列の値を取得し、結果を単一値として返します。ウィンドウ関数は、行のグループから列の値を取得し、各行の値を返します。集約関数は、ウィンドウ関数内で指定できます。ウィンドウ関数を集約関数内で指定することはできません。**AVG()**、**MAX()**、**MIN()**、および **SUM()** は集約関数またはウィンドウ関数として使用できます。

### 集約関数の使用

集約関数は、以下のような箇所で使用できます。

- ・ **SELECT リスト**。リストされた selectItem としてか、サブクエリの selectItem 内で使用できます。
- ・ **HAVING 節**。ただし、HAVING 節は、明示的に集約関数を指定する必要があります。対応する selectItem 列エイリアスや selectItem シーケンス番号を使用して集約を指定することはできません。
- ・ **DISTINCT BY 節**。ただし、集約関数自体の指定には意味がなく、常に単一行が返されます。DISTINCT BY (MAX(Age) - Age) のように、式の一部として集約関数を指定すると、有意義なものになります。

集約関数は、以下のような箇所で直接使用することはできません。

- ・ **ORDER BY 節**。これを実行しようすると、SQLCODE -73 エラーが生成されます。ただし、対応する**列エイリアス**や selectItem のシーケンス番号を指定することにより、ORDER BY 節で集約関数を使用できます。
- ・ **WHERE 節**。これを実行しようすると、SQLCODE -19 エラーが生成されます。

- ・ GROUP BY 節。これを実行しようとすると、SQLCODE -19 エラーが生成されます。
- ・ TOP 節。これを実行しようとすると、SQLCODE -1 エラーが生成されます。
- ・ JOIN。ON 節で集約を指定しようとすると、SQLCODE -19 エラーが生成されます。USING 節で集約を指定しようとすると、SQLCODE -1 エラーが生成されます。

ただし、[列エイリアス](#)を指定するサブクエリを使用することで、これらの節 (TOP 節を除く) に集約関数値を指定できます。例えば、WHERE 節を使用して平均 Age 値より小さい Age 値を選択するために、サブクエリ内で AVG 集約関数を使用できます。

## SQL

```
SELECT Name, Age, AvgAge
FROM (SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person)
WHERE Age < AvgAge
ORDER BY Age
```

## 集約関数とフィールドの組み合わせ

InterSystems SQL では、クエリ内の他の SELECT 項目と共に集約関数を指定できます。COUNT(\*) などの集約関数は、別個のクエリ内に配置する必要はありません。

## SQL

```
SELECT TOP 5 COUNT(*), Name, AVG(Age)
FROM Sample.Person
ORDER BY Name
```

選択リスト内で集約関数を指定して、フィールド選択項目を指定しない場合、InterSystems SQL は 1 行を返します。TOP 節は、TOP 0 (行を返さない) である場合を除いて無視されます。

## SQL

```
SELECT TOP 7 AVG(Age), LIST(Age)
FROM Sample.Person
WHERE Age > 75
```

選択リスト内で集約関数と 1 つ以上のフィールド選択項目を指定した場合、InterSystems SQL は、フィールド項目に必要な数の行を返します。

## SQL

```
SELECT DISTINCT Age, AVG(Age), LIST(Age)
FROM Sample.Person
WHERE Age > 75
```

## 列の名前とエイリアス

既定では、集約関数の結果に割り当てられる列名は Aggregate\_n となります。ここで、n という数字接尾辞は、SELECT リストで指定されている列順序番号です。したがって、以下の例では、列名 Aggregate\_2 および Aggregate\_5 が作成されます。

## SQL

```
SELECT TOP 5 Home_State, COUNT(*), Name, Age, AVG(Age)
FROM Sample.Person
ORDER BY Name
```

別の列名 (列のエイリアス) を指定するには、AS キーワードを使用します。

## SQL

```
SELECT COUNT(*) AS PersonCount
FROM Sample.Person, Sample.Employee
```

列エイリアスを使用して、**ORDER BY** 節で集約フィールドを指定できます。以下の例では、年齢が平均年齢から離れる順番で人々をリストします。

## SQL

```
SELECT Name, Age,
       AVG(Age) AS AvgAge,
       ABS(Age - AVG(Age)) AS RelAge
FROM Sample.Person
ORDER BY RelAge
```

列エイリアスの詳細は、SELECT 文を参照してください。

## ORDER BY の使用

LIST、%DLIST、XMLAGG、および JSON\_ARRAYAGG の各関数は、複数の行のテーブル列の値を単一の集約値に結合します。すべての集約フィールドが評価された後に ORDER BY 節がクエリ結果セットに適用されるため、ORDER BY がこれらの集約内の値の順序に直接影響することはありません。特定の状況下では、これらの集約の結果が順番に表示されることがありますが、この順番を信頼すべきではありません。特定の集約結果値内にリストされる値を、明示的に順序付けすることはできません。

## DISTINCT キーワード節

すべての集約関数で、オプションの DISTINCT キーワード節がサポートされています。このキーワードは、集約操作を、非重複（一意）フィールド値のみに限定します。既定のフィールドの照合 (%SQLUPPER) を使用する場合、大文字と小文字のみが異なるフィールド値は個別値と見なされません。DISTINCT が指定されていない場合、既定で、重複値を含むすべての非 NULL 値に対して集約操作が実行されます。MIN および MAX 集約関数では、DISTINCT キーワードは、サポートされていますが操作は実行されません。

集約関数は、オプションの BY(item-list) 従属節を含めて、完全な DISTINCT キーワード節の構文をサポートします。詳細は、“[DISTINCT 節](#)”を参照してください。

集約関数の DISTINCT field1 節では、NULL の field1 値は無視されます。これは SELECT 文の DISTINCT 節とは異なります。SELECT DISTINCT 節では、個別のフィールド値ごとに 1 行が返されるのと同じように、個別の NULL ごとに 1 行が返されます。ただし、集約関数の DISTINCT BY(field2) field1 では、field2 の個別の NULL は無視されません。例えば、FavoriteColors に 50 個の個別値と複数の NULL が含まれている場合、返される DISTINCT 行数は 51、COUNT(DISTINCT FavoriteColors) は 50、COUNT(DISTINCT BY(FavoriteColors) %ID) は 51 になります。

## SQL

```
SELECT DISTINCT FavoriteColors,
       COUNT(DISTINCT FavoriteColors),
       COUNT(DISTINCT BY(FavoriteColors) %ID)
FROM Sample.Person
```

## DISTINCT および GROUP BY との組み合わせ

SELECT **DISTINCT** を selectItem 集約関数および GROUP BY 節と共に使用した場合は、DISTINCT キーワードが指定されていない場合と同じ結果が返されます。希望の結果を得るには、その集約関数をサブクエリに配置します。

例えば、人数別の状態（4 人の状態、6 人の状態などがあります）の数を返すとします。この結果が得られることを期待して、次のようにします。

## SQL

```
SELECT DISTINCT COUNT(*) AS PersonCounts
FROM Sample.Person
GROUP BY Home_State
```

しかし実際には、次のように DISTINCT キーワードを指定しない場合と同じ、各状態の人数が得られます。

## SQL

```
SELECT COUNT(*) AS PersonCounts
FROM Sample.Person
GROUP BY Home_State
```

目的の結果を得るには、次のようにサブクエリを使用する必要があります。

## SQL

```
SELECT DISTINCT *
FROM (SELECT COUNT(*) AS PersonCounts FROM Sample.Person
      GROUP BY Home_State)
```

## 行カウント

クエリが集約値を返す場合、%ROWCOUNT の値は以下のようにクエリに依存します。

- ・ 集約関数のみ: 集約値を計算し、%ROWCOUNT 1 を返します。集約関数のみのクエリで行が選択されない場合でも、%ROWCOUNT 1: COUNT=0 が返されます。その他の集約関数では、NULL が返されます。
- ・ **GROUP BY** で集約関数のみ: GROUP BY 節で選択された各グループの集約値を返します。%ROWCOUNT は選択されたグループの数です。クエリで行が選択されない場合、GROUP BY はグループを選択せず、クエリは %ROWCOUNT 0 を返します。
- ・ **DISTINCT** で集約関数のみ: 集約値を計算し、%ROWCOUNT 1 を返します。クエリで行が選択されない場合、DISTINCT は個別の値を選択せず、クエリは %ROWCOUNT 0 を返します。
- ・ **TOP** 節で集約関数のみ: ゼロでない TOP 値の場合、集約値を計算し、%ROWCOUNT 1 を返します。TOP=0 の場合は、%ROWCOUNT 0 を返します。集約は計算されません。
- ・ フィールドでの集約: クエリがフィールド値と集約関数を返す場合、返される行数は選択された行の数です。クエリで行が選択されない場合、%ROWCOUNT 0 が返され、集約は計算されません。

これらの結果は、サブクエリまたは式の selectItem が存在しても影響を受けません。

## 集約、トランザクション、およびロック

クエリに集約関数を含めると、クエリはすべての結果セット・フィールドにデータの現在の状態を返します。それには、コミットされていないデータ変更が含まれます。そのため、集約関数を含むクエリでは ISOLATION LEVEL READ COMMITTED 設定は無視されます。コミットされていないデータの現在の状態は以下のとおりです。

- ・ INSERT と UPDATE : 変更がまだコミットされておらず、ロールバックされる可能性があっても、変更された値は集約の計算に含まれます。
- ・ DELETE と TRUNCATE TABLE : 削除がまだコミットされておらず、ロールバックされる可能性があっても、削除された行は集約の計算に含まれません。

一般に、集約関数は多数の行のデータを使用するため、集計の計算に関係するすべての行にトランザクションのロックを発行することはできません。そのため集約の計算中に、別のユーザが、そのデータを変更するトランザクションを実行している可能性があります。

## 集約とシャード・テーブル

[シャード・テーブル](#)については、集約関数のサポートが制限されています。例えば、集約関数の DISTINCT、%FOREACH、および %AFTERHAVING 節はシャード・テーブルではサポートされていません。”[シャード・クラスタにおけるクエリ](#)”を参照してください。

## 関連項目

- ・ [AVG](#)、[COUNT](#)、[%DLIST](#)、[JSON\\_ARRAYAGG](#)、[LIST](#)、[MAX](#)、[MIN](#)、[STDDEV](#)、[STDDEV\\_SAMP](#)、[STDDEV\\_POP](#)、[SUM](#)、[VARIANCE](#)、[VAR\\_SAMP](#)、[VAR\\_POP](#)、[XMLAGG](#) の各集約関数
- ・ [CREATE AGGREGATE](#) コマンド
- ・ [SELECT](#) コマンド
- ・ [ウィンドウ関数の概要](#)

# AVG (SQL)

指定した列の値の平均値を返す集約関数です。

## 構文

```
AVG([ALL | DISTINCT [BY(col-list)]] expression
    [%FOREACH(col-list)] [%AFTERHAVING])
```

## 概要

AVG **集約関数**は、expression の値の平均値を返します。一般的に expression は、クエリで返される複数行の中のフィールド名（または、フィールド名を 1 つ以上含む式）です。

AVG は、テーブルまたはビューを参照する **SELECT** クエリまたは SELECT サブクエリで使用できます。AVG は、一般のフィールド値と共に、SELECT リストや HAVING 節で使用できます。

AVG は、WHERE 節では使用できません。SELECT がサブクエリの場合を除いて、JOIN の ON 節では、AVG を使用できません。

AVG は、すべての集約関数と同様に、オプションの **DISTINCT** 節を取ることができます。AVG(DISTINCT col1) は、これらの個別（一意）の col1 フィールド値のみの平均値を計算します。AVG(DISTINCT BY(col2) col1) は、col2 値が個別（一意）であるレコードのこれらの col1 フィールド値のみの平均値を計算します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

## データ値

expression の値が DOUBLE 以外の場合には、AVG は倍精度浮動小数点数を返します。AVG で返される値の有効桁数は 18 です。返される値の小数桁数は、expression の有効桁数と小数桁数によって異なります。AVG で返される値の小数桁数は、18 から expression の有効桁数を引き、そこに expression の小数桁数を加えたものと等しくなります。

expression の値が DOUBLE の場合には、小数桁数は 0 です。

通常、AVG は、数値フィールドや日付フィールドなど、数値を含むフィールドまたは式に適用されます。既定では、集約関数は Display 値ではなく、Logical (内部) データ値を使用します。タイプ・チェックは実行されないため、(ほとんど意味はありませんが) 非数値フィールドに対して実行できます。AVG では、空文字列('') のように数値でない値はゼロ (0) として扱われます。expression のデータ型が VARCHAR の場合、返り値は DOUBLE データ型になります。

データ・フィールドの NULL 値は、AVG 集約関数の値を取得する場合は無視されます。クエリから行が返されない場合や、すべての行のデータ・フィールド値が NULL の場合、AVG は NULL を返します。

## 単一値の平均

AVG に指定される expression の値がすべて同じ場合、計算結果の平均値はアクセスされるテーブルの行数 (除数) によって異なります。例えば、テーブルの特定の列ですべての行の値が同じ場合、その列の平均値の計算結果が個々の列の値とわずかに異なる場合があります。

次の例では、平均値の計算結果でわずかな差異が生じる理由について説明します。最初のクエリではテーブル行が参照されていないため、AVG は 1 で除算されます。2 番目のクエリではテーブル行が参照されているため、AVG はテーブル内の行数で除算されます。

## SQL

```
SELECT
    {fn PI} AS Pi,
    AVG({fn PI}) AS AvgPiDividedBy1
FROM Sample.Person
```



## SQL

```
SELECT
    Name,
    {fn PI} AS Pi,
    AVG({fn PI}) AS AvgPiDividedByNumRows
FROM Sample.Person
```

## 最適化

フィールドに**ビットスライス・インデックス**が定義されている場合、AVG 計算の SQL 最適化でこのインデックスを使用できます。

## 現在のトランザクションで発生する変更

すべての集約関数と同様に、AVG も必ず、現在のトランザクションの分離レベルに関係なく、コミットされていない変更も含めてデータの現在の状態を返します。詳細は、“[SET TRANSACTION](#)”と“[START TRANSACTION](#)”を参照してください。

## 引数

### ALL

AVG で、expression にある値すべての平均値を返すことを指定する引数 (オプション)。キーワードが指定されていない場合は、これが既定になります。

### DISTINCT

AVG で値の一意なインスタンスのみの平均値を計算するように指定する **DISTINCT 節** (オプション)。DISTINCT で BY(col-list) 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。

### expression

任意の有効な式。普通は、平均値算出の対象となるデータ値を含む列の名前を指定します。

### %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです (オプション)。%FOREACH の詳細は、“[SELECT](#)”を参照してください。

### %AFTERHAVING

**HAVING 節**にある条件を適用する引数 (オプション)。

AVG は、NUMERIC または DOUBLE **データ型**のいずれかを返します。AVG は、expression がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下のクエリは、Sample.Employee データベースに記録されている全従業員の平均給与額をリストにします。クエリから返されたすべての行には同じ平均値が入っているので、このクエリは、平均給与額から成る 1 行のみを返します。結果の見やすさを考慮し、|| 演算子を使用して数値にドル記号を付加し、AS 節を使用して列にラベルを表示しています。

## SQL

```
SELECT '$' || AVG(Salary) AS AverageSalary
FROM Sample.Employee
```

以下のクエリは、従業員の平均給与額を州別にまとめてリストにします。

## SQL

```
SELECT Home_State,'$' || AVG(Salary) AS AverageSalary
FROM Sample.Employee
GROUP BY Home_State
```

以下のクエリは、平均給与額よりも給与額が多い従業員の名前と給与額をリストにします。また、全従業員の平均給与額も表示します。この値は、このクエリで返されるすべての行で同じ値になります。

## SQL

```
SELECT Name,Salary,
       '$' || AVG(Salary) AS AverageAllSalary
FROM Sample.Employee
HAVING Salary>AVG(Salary)
ORDER BY Salary
```

以下のクエリは、平均給与額よりも給与額が多い従業員の名前と給与額をリストにします。また、平均給与額よりも給与が多い従業員の平均給与額も表示します。この値は、このクエリで返されるすべての行で同じ値になります。

## SQL

```
SELECT Name,Salary,
       '$' || AVG(Salary %AFTERHAVING) AS AverageHighSalary
FROM Sample.Employee
HAVING Salary>AVG(Salary)
ORDER BY Salary
```

以下のクエリは、従業員が4人以上いる州とその州の従業員の平均給与額、およびその州で給与額が\$20,000を超える従業員の平均給与額をリストにします。

## SQL

```
SELECT Home_State,
       '$' || AVG(Salary) AS AvgStateSalary,
       '$' || AVG(Salary %AFTERHAVING) AS AvgLargerSalaries
FROM Sample.Employee
GROUP BY Home_State
HAVING COUNT(*) > 3 AND Salary > 20000
ORDER BY Home_State
```

以下のクエリは、複数の形式の DISTINCT 節を使用します。Home\_City に NULL が1つ以上含まれる場合、BY 節に個別値として NULL が1つ含まれる可能性があるため、AVG(DISTINCT BY col-list の例では追加の Age 値が平均に含まれることがあります。

## SQL

```
SELECT AVG(Age) AS AveAge,AVG(ALL Age) AS Synonym,
       AVG(DISTINCT Age) AS AveDistAge,
       AVG(DISTINCT BY(Home_City) Age) AS AvgAgeDistCity,
       AVG(DISTINCT BY(Home_City,Home_State) Age) AS AvgAgeDistCityState
FROM Sample.Person
```

以下のクエリは、%FOREACH キーワードと %AFTERHAVING キーワードの両方を使用します。名前が“A”、“M”、または“W”で始まる人がいる州の行を返します。(HAVING 節と GROUP BY 節)。各州の行には以下の値があります。

- ・ LIST(Age %FOREACH(Home\_State)) : 州内のすべての人の年齢のリスト。
- ・ AVG(Age %FOREACH(Home\_State)) : 州内のすべての人の平均年齢。
- ・ AVG(Age %AFTERHAVING) : データベースに記録されている人のうち、HAVING 節の条件を満たすすべての人の平均年齢 (この数はすべての行で同じ値になります)。
- ・ LIST(Age %FOREACH(Home\_State) %AFTERHAVING) : 州内の人のうち、HAVING 節の条件を満たすすべての人の年齢のリスト。

- ・ `AVG(Age %FOREACH(Home_State) %AFTERHAVING)` : 州内の人のうち、HAVING 節の条件を満たすすべての人の平均年齢。

## SQL

```
SELECT Home_State,
       LIST(Age %FOREACH(Home_State)) AS StateAgeList,
       AVG(Age %FOREACH(Home_State)) AS StateAgeAvg,
       AVG(Age %AFTERHAVING ) AS AgeAvgHaving,
       LIST(Age %FOREACH(Home_State)%AFTERHAVING ) AS StateAgeListHaving,
       AVG(Age %FOREACH(Home_State)%AFTERHAVING ) AS StateAgeAvgHaving
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'A%' OR Name LIKE 'M%' OR Name LIKE 'W%'
ORDER BY Home_State
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [COUNT 集約関数](#)
- ・ [SUM 集約関数](#)

# COUNT (SQL)

テーブルや特定の列から、行の数を返す集約関数です。

## 構文

```
COUNT(*)
COUNT(expression)

COUNT(DISTINCT expression)
COUNT(DISTINCT BY(column) expression)
COUNT(ALL expression)

COUNT( ... expression %FOREACH(column))
COUNT( ... expression ... %AFTERHAVING)
```

## 説明

COUNT は、テーブルまたは列の行数を返す[集約関数](#)です。COUNT は、BIGINT データ型を返します。行が存在しない場合、COUNT は 0 または NULL を返します (クエリによって異なります)。詳細は ["カウントで行が返されない"](#) を参照してください。

**SELECT** クエリで COUNT を使用して、クエリで参照しているテーブルの行数をカウントして結果セットに返します。テーブルまたはビューを参照するサブクエリや HAVING 節で COUNT を使用することもできます。WHERE 節では COUNT を使用できません。SELECT がサブクエリでない限り、JOIN の ON 節でも COUNT を使用できません。

- ・ COUNT(\*) はテーブルまたはビューの行数を返します。COUNT(\*) は、列の値が重複する行や NULL 値がある行なども含め、すべての行をカウントします。

以下のクエリは、Sample.Person にある行の合計数を返します。

### SQL

```
SELECT COUNT(*) FROM Sample.Person
```

例： [テーブル行と列の値のカウント](#)

- ・ COUNT([expression](#)) は、テーブルの列名であるかデータの列に評価される式である expression に存在する値の数を返します。COUNT(expression) では NULL 値をカウントしません。

以下のクエリは、Sample.Person の Name 列にある NULL ではない値の数を返します。

### SQL

```
SELECT COUNT(Name) AS TotalNames FROM Sample.Person
```

例を以下に示します。

- [テーブルにある行と列の値のカウント](#)
- [ストリームにある列の値のカウント](#)
- [列の組み合わせにある NULL でない値のカウント](#)

- ・ COUNT(DISTINCT expression) は、[DISTINCT 節](#)を使用して expression 列にある個別 (固有) 値の個数を返します。ストリーム列では DISTINCT を使用できません。個別値としてカウントされる値は、列照合によって異なります。例えば、%SQLUPPER に既定の列照合を適用すると、文字の大文字と小文字が異なるだけの値は個別値としてカウントされません。大文字と小文字が異なるだけの値もすべて個別値としてカウントするには、COUNT(DISTINCT(%EXACT(expression))) を使用します。NULL 値は COUNT DISTINCT によるカウントの対象になりません。

この文は、Sample.Person にある固有な年齢の個数を返します。

## SQL

```
SELECT COUNT(DISTINCT Age) FROM Sample.Person
```

例：個別の列値のカウント

- ・ COUNT(DISTINCT BY(column) expression) は、column で指定された列にある重複値の行をフィルタで除外し、expression 列にある値の個数を返します。column 列にある NULL 値は個別値としてカウントされます。

以下の文は、個別（固有）な人名を収めた FavoriteColors 列にある値の個数を返します。

## SQL

```
SELECT COUNT(DISTINCT BY(Name) FavoriteColors) FROM Sample.Person
```

例：個別の列値のカウント

- ・ COUNT(ALL expression) は、expression にあるすべての値の個数を返します。ALL キーワードは、すべての重複値を含む、すべての非 NULL 値をカウントします。キーワードが指定されていない場合は、ALL が既定の動作になります。
- ・ COUNT(... expression %FOREACH(column)) は、expression 列の値を、column リストの個別値別にグループ化し、各グループにある値の個数を返します。%FOREACH と GROUP BY は似ています。GROUP BY がクエリ全体に対して機能することに対し、%FOREACH では、クエリ全体の母集団を制限することなく、母集団の一部に対する集約を選択できます。

以下のクエリは、Sample.Person で指定された各人物が記述された行を返します。各行には、その人物の名前と出身州、その州に住んでいる人物の名前の総数が記述されています。

## SQL

```
SELECT
    Name,
    Home_State,
    COUNT(Name %FOREACH(Home_State)) AS PersonCountInState
FROM Sample.Person
```

例：グループ化された値のカウント

- ・ COUNT(... expression ... %AFTERHAVING) は、HAVING 節で指定された条件を適用した後でのみ、expression にある行をカウントします。%AFTERHAVING を省略すると、HAVING の条件がカウントで考慮されません。

以下のクエリは、州別にグループ化した名前の個数、および州別にグループ化した名前のうち、“M” で始まる名前の個数を返します。

## SQL

```
SELECT
    Home_State,
    COUNT(Name) AS NameCount,
    COUNT(Name %AFTERHAVING) AS MNameCount
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'M%'
```

例：グループ化された値のカウント

## 引数

### expression

カウント対象のデータ値を含む有効な式です。expression には、列の名前、または結果がデータの列となる式を指定できます。expression をサブクエリとして指定することはできません。

### column

1 つの列名、またはコンマで区切った複数の列名のリストです。

- ・ COUNT(expression %FOREACH(column)) 構文では、データのグループ化に使用する列を column で指定し、COUNT で expression 列の値をカウントします。column にストリーム列を指定することはできません。
- ・ COUNT(DISTINCT BY(column) expression) 構文では、重複行をフィルタで除外するために使用する個別値を収めた列を column で指定し、COUNT で expression 列の値をカウントします。

## 例

### テーブルにある行と列の値のカウント

以下のクエリは、Sample.Person テーブルにある行の合計数を返します。この数には、1 つ以上の列に NULL 値がある行の数も算入されます。

#### SQL

```
SELECT COUNT(*) AS TotalPersons
FROM Sample.Person
```

以下のクエリは、Sample.Person にある名前、配偶者名、お気に入りの色の個数を返します。COUNT は、NULL 値を列カウントから除外します。したがって、列ごとの戻り値の個数は、COUNT(\*) で返される行の総数と異なるか、その数を下回ることがあります。

#### SQL

```
SELECT
  COUNT(Name) AS People,
  COUNT(Spouse) AS PeopleWithSpouses,
  COUNT(FavoriteColors) AS PeopleWithColorPref
FROM Sample.Person
```

### 個別の列値のカウント

以下のクエリは COUNT DISTINCT を使用して、Sample.Person にある個別の FavoriteColors 値の個数を返します。FavoriteColors 列には、いくつかのデータ値と NULL が記述されています。また、SELECT DISTINCT 節を使用して、個別の FavoriteColors 値ごとに 1 行が返されます。この行数は、COUNT(DISTINCT FavoriteColors) によるカウントよりも 1 大きくなります。DISTINCT は、単一の NULL を個別値としてその行を返しますが、COUNT DISTINCT は NULL をカウントしません。COUNT(DISTINCT BY(FavoriteColors) %ID) では、BY 節があることで NULL が個別値としてカウントされないの、返される値は行数と同じになります。

#### SQL

```
SELECT
  DISTINCT FavoriteColors,
  COUNT(DISTINCT FavoriteColors) AS DistColors,
  COUNT(DISTINCT BY(FavoriteColors) %ID) AS DistColorPeople
FROM Sample.Person
```

## グループ化された値のカウント

この例でのクエリは、**GROUP BY** を使用して、列の中で繰り返される値をグループ化し、その固有値 1 つごとに 1 行を返します。このクエリでは、続いて **COUNT** を使用して、別の列にある値のグループ別カウントを返します。

このクエリは、個別の **FavoriteColors** 値ごとに 1 行を返します。**FavoriteColors** が不要であるとする、このクエリは、**NULL** 値の行があればそれを返します。各行には、以下の 2 つのカウントが関連付けられています。

- ・ **FavoriteColors** オプションの値がある行の数。**NULL** 値の行はカウントされません。
- ・ 各 **FavoriteColors** オプションに関連付けられた名前数。**Name** に **NULL** 値がないとすると、このカウントには **FavoriteColors** が **NULL** 値であるカウントが算入されます。

### SQL

```
SELECT
    FavoriteColors,
    COUNT(FavoriteColors) AS ColorPreference,
    COUNT(Name) AS People
FROM Sample.Person
GROUP BY FavoriteColors
```

以下のクエリは、**Sample.Person** にある **Home\_State** 値ごとに該当する人物の行数を返します。

### SQL

```
SELECT
    Home_State,
    COUNT(*) AS AllPersons
FROM Sample.Person
GROUP BY Home_State
```

以下のクエリは、**%AFTERHAVING** を使用して、66 歳以上の人が 1 人以上いる各州の個人の行数と 66 歳以上の人数を返します。

### SQL

```
SELECT
    Home_State,
    COUNT(Name) AS AllPersons,
    COUNT(Name %AFTERHAVING) AS Seniors
FROM Sample.Person
GROUP BY Home_State
HAVING Age > 65
ORDER BY Home_State
```

このクエリは **%FOREACH** キーワードと **%AFTERHAVING** キーワードの両方を使用します。州ごとに 1 行を返すために **GROUP BY** を使用し、“A”、“M”、“W” のいずれかで始まる名前の人物のみを抽出するために **HAVING** を使用します。

### SQL

```
SELECT
    Home_State,
    COUNT(Name) AS NameCount,
    COUNT(Name %FOREACH(Home_State)) AS StateNameCount,
    COUNT(Name %AFTERHAVING) AS NameCountHaving,
    COUNT(Name %FOREACH(Home_State) %AFTERHAVING) AS StateNameCountHaving
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'A%' OR Name LIKE 'M%' OR Name LIKE 'W%'
ORDER BY Home_State
```

各州の行には以下のカウント値があります。

- ・ **COUNT(Name)** – データベースに記録されているすべての人物。**Name** が必須であるとする、このカウントは行の総数と同じです。



- ・ COUNT(Name %FOREACH(Home\_State)) – 州のすべての人物。
- ・ COUNT(Name %AFTERHAVING) – データベースの中で HAVING の条件を満たすすべての人物。Name が必須であるとする、この数は行の総数と同じです。
- ・ COUNT(Name %FOREACH(Home\_State) %AFTERHAVING) – 州の中で HAVING の条件を満たすすべての人物。

## 列の組み合わせにある NULL でない値のカウント

このクエリは、連結演算子 (||) と共に COUNT を使用して、FavoriteColors 列と Home\_State 列のどちらにも NULL 値がない行をカウントします。

### SQL

```
SELECT COUNT(FavoriteColors || Home_State) AS ColorState
FROM Sample.Person
```

## ストリームにある列の値のカウント

COUNT(expression) を [ストリーム列](#) の値のカウントに使用できますが、多少の制約があります。

- ・ 列のカウントには、重複値も含めて、NULL ではない値の総数が必ず算入されます。
- ・ COUNT DISTINCT expression 節でストリーム・フィールドを指定することはできません。これを実行しようとすると、SQLCODE -37 エラーが返されます。
- ・ %FOREACH column 節でストリーム・フィールドを指定することはできません。これを実行しようとすると、SQLCODE -37 エラーが返されます。

以下のクエリは、COUNT 関数の有効な使用法を示しています。Title は文字列フィールド、Notes と Picture はストリーム・フィールドです。

### SQL

```
SELECT DISTINCT Title, COUNT(Notes), COUNT(Picture %FOREACH(Title))
FROM Sample.Employee
```

ストリーム・フィールドを指定したこれらのクエリは有効ではありません。

### SQL

```
-- Invalid: DISTINCT keyword with stream field
SELECT Title, COUNT(DISTINCT Notes) FROM Sample.Employee
```

### SQL

```
-- Invalid: %FOREACH col-list contains stream field
SELECT Title, COUNT(Notes %FOREACH(Picture))
FROM Sample.Employee
```

## カウントで行が返されない

以下の各例では、COUNT でカウントする行が SELECT クエリで選択されない場合に COUNT が何を返すのかを示しています。クエリに応じて、COUNT は 0 または NULL を返します。

集約関数に対して指定する列を除き、FROM 節のテーブルにある列への参照が SELECT selectItem にない場合、COUNT は 0 を返します。

COUNT は 0 を返す唯一の集約関数です。他のすべての集約関数は NULL を返します。クエリは %ROWCOUNT 1 を返します。クエリのサンプルを以下に示します。

## SQL

```
SELECT
  COUNT(*) AS Recs, COUNT(Name) AS People,
  AVG(Age) AS AvgAge, MAX(Age) AS MaxAge,
  CURRENT_TIMESTAMP AS Now
FROM Sample.Employee
WHERE Name %STARTSWITH 'ZZZ'
```

SELECT selectItem に FROM 節のテーブルにある列への直接参照が記述されている場合、または TOP 0 が指定されている場合、COUNT は NULL を返します。クエリは %ROWCOUNT 0 を返します。クエリのサンプルを以下に示します。

## SQL

```
SELECT
  COUNT(*) AS Recs,
  COUNT(Name) AS People,
  $LENGTH(Name) AS NameLen
FROM Sample.Employee WHERE Name %STARTSWITH 'ZZZ'
```

テーブルを指定しない場合、COUNT(\*) は 1 を返します。クエリは %ROWCOUNT 1 を返します。クエリのサンプルを以下に示します。

## SQL

```
SELECT COUNT(*) AS Recs
```

## セキュリティおよび特権

COUNT(\*) 構文を使用するには、指定したテーブルに対するテーブルレベルの SELECT 特権が必要です。

COUNT(expression) 構文を使用するには、expression で指定した列に対する列レベルの SELECT 特権、または指定したテーブルに対するテーブルレベルの SELECT 特権が必要です。

- SELECT 特権があるかどうかを確認するには [%CHECKPRIV](#) を使用します。
- テーブルレベルの SELECT 特権があるかどうかを確認するには `$SYSTEM.SQL.Security.CheckPrivilege()` を使用します。
- 特権を割り当てるには [GRANT](#) を使用します。

## パフォーマンス

COUNT で高いパフォーマンスを得るには、以下のインデックスを定義することを検討します。

- テーブルを作成したときに [ビットマップ・エクステン・インデックス](#) が自動的に定義されていない場合は、COUNT(\*) 構文でビットマップ・エクステン・インデックスを定義します。
- COUNT(expression) 構文では、expression. で指定した列に [ビットスライス・インデックス](#) を定義します。  
COUNT(expression) のクエリ・プラン最適化によって、カウント対象とする列に既定の照合が自動的に適用されます。

## トランザクションの考慮事項

すべての集約関数と同様に、COUNT からは、トランザクションの現在の分離レベルに関係なく、コミットされていない変更も含め、データの現在の状態が返されます。COUNT は以下の動作に従います。

- 挿入されたレコードや更新されたレコードをカウントします。そのレコードの変更がコミットされていなくて、ロールバックされる可能性がある場合でもカウントします。
- 削除されたレコードはカウントしません。その削除がコミットされていなくて、ロールバックされる可能性がある場合でもカウントしません。

詳細は、["SET TRANSACTION"](#) および ["START TRANSACTION"](#) を参照してください。

## 関連項目

- ・ [集約関数](#)
- ・ [AVG](#)
- ・ [SUM](#)

## %DLIST (SQL)

値の InterSystems IRIS リストを作成する集約関数です。

### 構文

```
%DLIST([ALL | DISTINCT [BY(col-list)]]  
      string-expr  
      [%FOREACH(col-list)] [%AFTERHAVING])
```

### 説明

%DLIST [集約関数](#)は、指定された列の値をリストの要素として含む ObjectScript %List 構造を返します。

シンプルな %DLIST (または %DLIST ALL) は、選択された行の string-expr に対応する NULL 以外のすべての値で構成される InterSystems IRIS リストを返します。string-expr が NULL の行は、リスト構造の要素として含まれません。

%DLIST DISTINCT は、選択された行の string-expr に対応する NULL 以外のすべての個別 (一意) の値で構成される InterSystems IRIS リストを返します (%DLIST(DISTINCT col1))。NULL は、%List 構造の要素として含まれません。%DLIST(DISTINCT BY(col2) col1) は、col2 値が個別 (一意) であるレコードのこれらの col1 フィールド値のみが含まれる要素の %List を返します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

InterSystems IRIS リスト構造の詳細は、[%LIST](#) および関連する関数を参照してください。

### %DLIST と %SelectMode

[%SelectMode](#) プロパティを使用して、%DLIST から返されるデータの表示モードを 0 = 論理 (既定)、1 = ODBC、2 = 表示のいずれかに指定できます。

ODBC モードでは %DLIST は列の値のリストをコンマで区切り、\$LISTTOSTRING は (既定では) %List 列値内の要素をコンマで区切って返します。

### %DLIST と ORDER BY

%DLIST 関数は、複数の行のテーブル列の値を %List 構造の値リストに結合します。すべての集約フィールドが評価された後に ORDER BY 節がクエリ結果セットに適用されるため、ORDER BY がこのリスト内の値の順序に直接影響することはありません。特定の状況下では、%DLIST の結果が順番に表示されることがありますが、この順番を信頼すべきではありません。特定の集約結果値内にリストされる値を、明示的に順序付けすることはできません。

### 関連する集約関数

- ・ %DLIST は、値の InterSystems IRIS リストを返します。
- ・ [LIST](#) は、コンマで区切られた値のリストを返します。
- ・ [JSON\\_ARRAYAGG](#) は、JSON 配列の値を返します。
- ・ [XMLAGG](#) は、値を連結した文字列を返します。

### 引数

#### ALL

%DLIST で、string-expr にある値すべてのリストを返すことを指定する引数 (オプション)。キーワードが指定されていない場合は、これが既定になります。

## DISTINCT

%DLIST が一意の string-expr の値のみを含む %List 構造リストを返すように指定する **DISTINCT** 節 (オプション)。DISTINCT で **BY**(col-list) 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。

### string-expr

文字列として評価する SQL 式。通常は選択されたテーブルの列の名前です。

### %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです (オプション)。%FOREACH の詳細は、“**SELECT**” を参照してください。

### %AFTERHAVING

**HAVING** 節にある条件を適用する引数 (オプション)。

## 例

以下の例では、Sample.Person テーブルの Home\_State 列にある、先頭の文字が “A” であるすべての値の InterSystems IRIS リストを返します。

### SQL

```
SELECT %DLIST(Home_State)
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

この InterSystems IRIS リストには、値が重複する要素が含まれています。

以下の例では、Sample.Person テーブルの Home\_State 列にある、先頭の文字が “A” であるすべての個別値 (一意の値) の InterSystems IRIS リストを返します。

### SQL

```
SELECT %DLIST(DISTINCT Home_State)
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

以下の例では、各州の Home\_City 列にあるすべての値の InterSystems IRIS リストを作成し、これらの都市の値の数を州ごとにカウントします。それぞれの Home\_State の行には、その州のすべての Home\_City 値のリストが格納されます。これらのリストでは、都市名が重複している可能性があります。

### SQL

```
SELECT Home_State,
       %DLIST(Home_City) AS AllCities,
       COUNT(Home_City) AS CityCount
FROM Sample.Person
GROUP BY Home_State
```

以下の例に示すように、各州の Home\_City 列にあるすべての個別値のリストを生成したほうが役に立つことが普通です。

### SQL

```
SELECT Home_State,
       %DLIST(DISTINCT Home_City) AS CitiesList,
       COUNT(DISTINCT Home_City) AS DistinctCities,
       COUNT(Home_City) AS TotalCities
FROM Sample.Person
GROUP BY Home_State
```

この例は、各州の個別の都市名をカウントした整数と、都市名の合計をカウントした整数の両方を返します。

以下の例は、“A” で始まる Home\_State 値の %List 構造を返します。これは、個別の Home\_State 値 (DISTINCT Home\_State)、個別の Home\_City 値に対応する Home\_State 値 (DISTINCT BY(Home\_City) Home\_State、Home\_City に対応する一意の NULL が 1 つ含まれる場合があります)、およびすべての Home\_State 値を %List 要素として返します。

## SQL

```
SELECT %DLIST(DISTINCT Home_State) AS DistStates,
        %DLIST(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
        %DLIST(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

以下のダイナミック SQL の例では、%SelectMode プロパティを使用して、%List 構造の FavoriteColors 日付フィールドに ODBC 表示モードを指定しています。ODBC モードは、各列の値をコンマ区切りリストとして返し、\$LISTTOSTRING 関数は異なる区切り文字 (この例では ||) を指定し、異なる列の値を区切ります。

## ObjectScript

```
set myquery = "SELECT %DLIST(FavoriteColors) AS colors FROM Sample.Person WHERE Name %STARTSWITH 'A'"

set tStatement = ##class(%SQL.Statement).%New()
set tStatement.%SelectMode=1

set qStatus = tStatement.%Prepare(myquery)
if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}

set rset = tStatement.%Execute()
if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}

while rset.%Next()
{
    write $LISTTOSTRING(rset.colors,"||"),!
}
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message quit}

write !,"End of data"
```

以下の例では、%AFTERHAVING キーワードを使用します。ここでは、HAVING 節の条件 (先頭の文字が “M” である名前) を満たす Name 値が 1 つ以上ある Home\_State 行をすべて返します。最初の %DLIST 関数は、該当の州についてすべての名前のリストを返します。2 番目の %DLIST 関数は、HAVING 節の条件を満たす名前のみを含むリストを返します。

## SQL

```
SELECT Home_State,
        %DLIST(Name) AS AllNames,
        %DLIST(Name %AFTERHAVING) AS HaveClauseNames
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'M%'
ORDER BY Home_state
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [SELECT](#)
- ・ [\\$LIST 関数](#)
- ・ [JSON\\_ARRAYAGG 集約関数](#)
- ・ [LIST 集約関数](#)

- ・ [XMLAGG](#) 集約関数



## JSON\_ARRAYAGG (SQL)

JSON フォーマットの配列の値を作成する集約関数です。

### 構文

```
JSON_ARRAYAGG([ ALL | DISTINCT [BY(col-list)] ]
  string-expr
  [ %FOREACH(col-list) ] [ %AFTERHAVING ] )
```

### 引数

引数	説明
ALL	オプション - JSON_ARRAYAGG が JSON 配列 (string-expr のすべての値が含まれます) を返すように指定します。キーワードが指定されていない場合は、これが既定になります。
DISTINCT	オプション - 一意の string-expr 値のみを含む JSON 配列を JSON_ARRAYAGG が返すように指定する <a href="#">DISTINCT</a> 節。DISTINCT で BY(col-list) 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。
string-expr	文字列として評価する SQL 式。通常は選択されたテーブルの列の名前です。
%FOREACH(col-list)	オプション - 列名、またはコンマで区切られた列名のリスト。%FOREACHの詳細は、" <a href="#">SELECT</a> " を参照してください。
%AFTERHAVING	オプション - <a href="#">HAVING</a> 節にある条件を適用します。

### 概要

JSON\_ARRAYAGG [集約関数](#)は、指定された列の値の JSON フォーマットの配列を返します。JSON 配列のフォーマットの詳細は、[JSON\\_ARRAY](#) 関数を参照してください。

シンプルな JSON\_ARRAYAGG (または JSON\_ARRAYAGG ALL) は、選択された行の string-expr に対応するすべての値を含む JSON 配列を返します。配列では、string-expr が空の文字列 (') の行を ("¥u0000") によって表します。string-expr が NULL の行は、配列に含まれません。string-expr の値が 1 つのみでそれが空の文字列 (') である場合、JSON\_ARRAYAGG は JSON 配列 [ "¥u0000" ] を返します。string-expr の値がすべて NULL である場合、JSON\_ARRAYAGG は空の JSON 配列 [ ] を返します。

JSON\_ARRAYAGG DISTINCT は、選択された行の string-expr に対応するすべての異なる (一意の) 値で構成される JSON 配列を返します (JSON\_ARRAYAGG(DISTINCT col1))。NULL の string-expr は、JSON 配列に含まれません。JSON\_ARRAYAGG(DISTINCT BY(col2) col1) は、col2 値が個別 (一意) であるレコードのこれらの col1 フィールド値のみが含まれる JSON 配列を返します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

JSON\_ARRAYAGG string-expr にストリーム・フィールドを指定することはできません。ストリーム・フィールドを指定すると、SQLCODE -37 になります。

### エスケープ文字を格納するデータ値

- 二重引用符: string-expr の値に二重引用符文字 (") が含まれている場合、JSON\_ARRAYAGG はリテラルのエスケープ・シーケンス \" を使用してこの文字を表します。

- ・ バックスラッシュ: string-expr の値にバックスラッシュ文字 (¥) が含まれている場合、JSON\_ARRAYAGG はリテラルのエスケープ・シーケンス \\ を使用してこの文字を表します。
- ・ 一重引用符: string-expr の値にリテラル文字として一重引用符が含まれている場合、InterSystems SQL では、2 つの一重引用符 (') としてこれを二重にしてこの文字をエスケープする必要があります。JSON\_ARRAYAGG は一重引用符文字 ' としてこの文字を表します。

## 最大 JSON 配列サイズ

JSON\_ARRAYAGG の既定の返りタイプは VARCHAR(8192) です。この長さには、JSON 配列フォーマット文字およびフィールド・データ文字が含まれます。返される値が 8192 よりも長い必要があると予想される場合、CAST 関数を使用してより長い返り値を指定できます。例えば、CAST(JSON\_ARRAYAGG(value)) AS VARCHAR(12000)) と指定します。返された実際の JSON 配列が JSON\_ARRAYAGG の返りタイプの長さよりも長い場合、InterSystems IRIS は、エラーを発行せずに返りタイプの長さで JSON 配列を切り捨てます。JSON 配列を切り捨てると閉じ括弧文字 ] が削除されるため、返り値が無効になります。

## JSON\_ARRAYAGG と %SelectMode

%SelectMode プロパティを使用して、JSON 配列の要素のデータ表示値を 0 = 論理 (既定)、1 = ODBC、2 = 表示のいずれかに指定できます。string-expr に %List 構造が含まれている場合、その要素は ODBC モードではコンマで区切られて表され、論理モードおよび表示モードでは %List フォーマット文字が ¥ エスケープ・シーケンスで表されます。これらの JSON ¥ エスケープ・シーケンスの一覧表は、“\$ZCONVERT” で “エンコード変換” を参照してください。

## JSON\_ARRAYAGG と ORDER BY

JSON\_ARRAYAGG 関数は、複数の行のテーブル列の値を要素値の JSON 配列に結合します。すべての集約フィールドが評価された後に ORDER BY 節がクエリ結果セットに適用されるため、ORDER BY がこのリスト内の値の順序に直接影響することはありません。特定の状況下では、JSON\_ARRAYAGG の結果が順番に表示されることがありますが、この順番を信頼すべきではありません。特定の集約結果値内にリストされる値を、明示的に順序付けすることはできません。

## 関連する集約関数

- ・ LIST は、コンマで区切られた値のリストを返します。
- ・ %DLIST は、各値の要素を含む InterSystems IRIS リストを返します。
- ・ XMLAGG は、値を連結した文字列を返します。

## 例

この例では、Sample.Person テーブルの Home\_State 列にある、先頭の文字が “A” であるすべての値の JSON 配列を返します。

### SQL

```
SELECT JSON_ARRAYAGG(Home_State)
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

この JSON 配列には重複値が含まれます。

以下の例では、Sample.Person テーブルの Home\_State 列から先頭の文字が “A” であるすべての個別値 (一意の値) を抽出し、それを JSON 配列を含むホスト変数に格納し、返します。

### SQL

```
SELECT DISTINCT JSON_ARRAYAGG(Home_State) AS DistinctStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

以下の例では、各州の Home\_City 列にあるすべての値の JSON 配列を作成し、これらの都市の値の数を州ごとにカウントします。それぞれの Home\_State の行には、その州のすべての Home\_City 値の JSON 配列が格納されます。これらの JSON 配列では、都市名が重複している可能性があります。

## SQL

```
SELECT Home_State,
       COUNT(Home_City) AS CityCount,
       JSON_ARRAYAGG(Home_City) AS ArrayAllCities
FROM Sample.Person
GROUP BY Home_State
```

以下の例に示すように、各州の Home\_City 列にあるすべての個別値の JSON 配列を生成した方が役に立つことが普通です。

## SQL

```
SELECT DISTINCT Home_State,
       COUNT(DISTINCT Home_City) AS DistCityCount,
       COUNT(Home_City) AS TotCityCount,
       JSON_ARRAYAGG(DISTINCT Home_City) AS ArrayDistCities
FROM Sample.Person GROUP BY Home_State
```

この例は、各州の個別の都市名をカウントした整数と、都市名の合計をカウントした整数の両方を返します。

以下の動的 SQL の例では、**%SelectMode** プロパティを使用して、DOB 日付フィールドによって返される値の JSON 配列に ODBC 表示モードを指定しています。

## ObjectScript

```
SET myquery = 2
SET myquery(1) = "SELECT JSON_ARRAYAGG(DOB) AS DOBs "
SET myquery(2) = "FROM Sample.Person WHERE Name %STARTSWITH 'A'"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下の例では、**%FOREACH** キーワードを使用します。これは、Home\_State の age 値の JSON 配列を含むそれぞれ個別の Home\_State の行を返します。

## SQL

```
SELECT DISTINCT Home_State,
       JSON_ARRAYAGG(Age %FOREACH(Home_State))
FROM Sample.Person
WHERE Home_State %STARTSWITH 'M'
```

以下の例では、**%AFTERHAVING** キーワードを使用します。ここでは、HAVING 節の条件（先頭の文字が“M”である名前）を満たす Name 値が 1 つ以上ある Home\_State 行をすべて返します。最初の JSON\_ARRAYAGG 関数は、該当の州についてすべての名前の JSON 配列を返します。2 番目の JSON\_ARRAYAGG 関数は、HAVING 節の条件を満たす名前のみを含む JSON 配列を返します。

## SQL

```
SELECT Home_State,
       JSON_ARRAYAGG(Name) AS AllNames,
       JSON_ARRAYAGG(Name %AFTERHAVING) AS HavingClauseNames
FROM Sample.Person GROUP BY Home_State
HAVING Name LIKE 'M%' ORDER BY Home_State
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [JSON\\_ARRAY](#) 関数
- ・ [IS JSON](#) の述語条件
- ・ [LIST](#) 集約関数
- ・ [%DLIST](#) 集約関数
- ・ [XMLAGG](#) 集約関数
- ・ [SELECT](#) 文

## LIST (SQL)

コンマで区切られたリストの値を生成する集約関数です。

### 構文

```
LIST([ ALL | DISTINCT [BY(col-list)] ]
    string-expr
    [ %FOREACH(col-list) ] [ %AFTERHAVING ] )
```

### 引数

引数	説明
ALL	オプション - LIST が string-expr のすべての値のリストを返すように指定します。キーワードが指定されていない場合は、これが既定になります。
DISTINCT	オプション - 一意の string-expr 値のみを含むリストを LIST が返すように指定する <a href="#">DISTINCT 節</a> 。DISTINCT で BY(col-list) 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。
string-expr	文字列として評価する SQL 式。通常は選択されたテーブルの列の名前です。
%FOREACH(col-list)	オプション - 列名、またはコンマで区切られた列名のリスト。%FOREACHの詳細は、“ <a href="#">SELECT</a> ”を参照してください。
%AFTERHAVING	オプション - <a href="#">HAVING</a> 節にある条件を適用します。

### 概要

LIST [集約関数](#)は、指定された列の値のコンマ区切りのリストを返します。

LIST (または LIST ALL) は、選択された行の string-expr に対する、すべての値で構成されるコンマで区切られたリストを含む文字列を返します。コンマで区切られたリストでは、string-expr が空の文字列 (') の行をプレースホルダのコンマによって表します。string-expr が NULL の行は、コンマで区切られたリストに含まれません。string-expr の値が 1 つのみでそれが空の文字列 (') である場合、LIST は空の文字列を返します。

LIST DISTINCT は、選択された行の string-expr に対応するすべての異なる (一意の) 値で構成されるコンマ区切りリストを含む文字列を返します (LIST(DISTINCT col1))。NULL の string-expr は、コンマで区切られたリストに含まれません。LIST(DISTINCT BY(col2) col1) は、col2 値が個別 (一意) であるレコードのこれらの col1 フィールド値のみが含まれるコンマ区切りリストを返します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

### コンマを含むデータ値

LIST はコンマを使用して string-expr 値を区切るため、LIST をコンマを含むデータ値に使用しないでください。代わりに [%DLIST](#) または [JSON\\_ARRAYAGG](#) を使用してください。

### LIST と %SelectMode

[%SelectMode](#) プロパティを使用して、LIST から返されるデータの表示モードを 0 = 論理 (既定)、1 = ODBC、2 = 表示のいずれかに指定できます。

LIST では列の値がコンマで区切られ、ODBC モードでは %List 列値内の要素がコンマで区切られることに注意してください。したがって、%List 構造で LIST を使用している場合に ODBC モードを使用すると、生成される結果があいまいになります。

## LIST と ORDER BY

LIST 関数は、複数の行のテーブル列の値を 1 つのコンマ区切り値リストに結合します。すべての集約フィールドが評価された後に ORDER BY 節がクエリ結果セットに適用されるため、ORDER BY がこのリスト内の値の順序に直接影響することはありません。特定の状況下では、LIST の結果が順番に表示されることがありますが、この順番を信頼すべきではありません。特定の集約結果値内にリストされる値を、明示的に順序付けすることはできません。

## 最大 LIST サイズ

LIST の返り値は、[最大文字列長](#)を超えることはできません。

## 関連する集約関数

- LIST は、コンマで区切られた値のリストを返します。
- %DLIST は、各値の要素を含むリストを返します。
- JSON\_ARRAYAGG は、JSON 配列の値を返します。
- XMLAGG は、値を連結した文字列を返します。

## 例

以下の SQL の例では、Sample.Person テーブルの Home\_State 列から先頭の文字が "A" であるすべての値を抽出し、それをコンマ区切りリストとしてホスト変数に格納し、返します。

### SQL

```
SELECT LIST(Home_State) AS StateList
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

このリストには重複値が含まれます。

以下の SQL の例では、Sample.Person テーブルの Home\_State 列から先頭の文字が "A" であるすべての個別値（一意の値）を抽出し、それをコンマ区切りリストとしてホスト変数に格納し、返します。

### SQL

```
SELECT LIST(DISTINCT Home_State) AS DistinctStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

以下の SQL の例では、各州の Home\_City 列にあるすべての値のコンマ区切りリストを作成し、これらの都市の値の数を州ごとにカウントします。それぞれの Home\_State の行には、その州のすべての Home\_City 値のリストが格納されます。これらのリストでは、都市名が重複している可能性があります。

### SQL

```
SELECT Home_State,
       COUNT(Home_City) AS CityCount,
       LIST(Home_City) AS ListAllCities
FROM Sample.Person
GROUP BY Home_State
```

以下の例に示すように、各州の Home\_City 列にあるすべての個別値のコンマ区切りリストを生成したほうが役に立つことが普通です。

## SQL

```
SELECT Home_State,
       COUNT(DISTINCT Home_City) AS DistCityCount,
       COUNT(Home_City) AS TotCityCount,
       LIST(DISTINCT Home_City) AS DistCitiesList
FROM Sample.Person
GROUP BY Home_State
```

この例は、各州の個別の都市名をカウントした整数と、都市名の合計をカウントした整数の両方を返します。

以下の例は、“A” で始まる Home\_State 値のリストを返します。これは、個別の Home\_State 値 (DISTINCT Home\_State)、個別の Home\_City 値に対応する Home\_State 値 (DISTINCT BY(Home\_City) Home\_State、Home\_City に対応する一意の NULL が 1 つ含まれる場合があります)、およびすべての Home\_State 値を返します。

## SQL

```
SELECT LIST(DISTINCT Home_State) AS DistStates,
       LIST(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       LIST(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

以下のダイナミック SQL の例では、%SelectMode プロパティを使用して、DOB 日付フィールドによって返される値のリストに ODBC 表示モードを指定しています。

## ObjectScript

```
SET myquery = "SELECT LIST(DOB) AS DOBs FROM Sample.Person WHERE Name %STARTSWITH 'A'"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

以下の例では、%FOREACH キーワードを使用します。これは、Home\_State の age 値のリストを含むそれぞれ個別の Home\_State の行を返します。

## SQL

```
SELECT DISTINCT Home_State,
       LIST(Age %FOREACH(Home_State)) AgesForState
FROM Sample.Person WHERE Home_State %STARTSWITH 'M'
```

以下の例では、%AFTERHAVING キーワードを使用します。ここでは、HAVING 節の条件 (先頭の文字が “M” である名前) を満たす Name 値が 1 つ以上ある Home\_State 行をすべて返します。最初の LIST 関数は、該当の州についてすべての名前のリストを返します。2 番目の LIST 関数は、HAVING 節の条件を満たす名前のみを含むリストを返します。

## SQL

```
SELECT Home_State,
       LIST(Name) AS AllNames,
       LIST(Name %AFTERHAVING) AS HavingClauseNames
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'M%'
ORDER BY Home_State
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [%DLIST 集約関数](#)



- ・ [JSON\\_ARRAYAGG](#) 集約関数
- ・ [XMLAGG](#) 集約関数
- ・ [SELECT](#) 文

# MAX (SQL)

指定された列内の最大データ値を返す集約関数です。

## 構文

```
MAX([ ALL | DISTINCT [BY(col-list)] ]  
    expression  
    [ %FOREACH(col-list) ] [ %AFTERHAVING ] )
```

## 概要

MAX [集約関数](#)は、expression の最大値を返します。一般的に expression は、クエリで返される複数行の中のフィールドの名前 (または、フィールド名を 1 つ以上含む式) です。

MAX は、テーブルまたはビューを参照する [SELECT](#) クエリまたは SELECT サブクエリで使用できます。MAX は、一般のフィールド値と共に SELECT リストや HAVING 節で表示できます。

MAX は、WHERE 節では使用できません。SELECT がサブクエリの場合を除いて、JOIN の ON 節では MAX を使用できません。

その他の大半の集約関数と同様に、MAX は[ストリーム・フィールド](#)に適用できません。これを実行しようとすると、SQLCODE -37 エラーが生成されます。

その他の大半の集約関数とは異なり、MAX(DISTINCT BY(col2) col1) を含めて、ALL および DISTINCT キーワードは MAX 内で何も実行しません。これらは、SQL-92 の互換性のために用意されたものです。

## データ値

MAX によって使用される指定されたフィールドは、数値または非数値です。数値データ型フィールドの場合、最大値は最も大きな数値として定義されます。したがって -3 の方が -7 よりも大きくなります。非数値データ型フィールドの場合、最大値は文字列の[照合順](#)で最も大きな値として定義されます。したがって '-7' の方が '-3' よりも大きくなります。

空文字列('') 値は CHAR(0) として処理されます。

述語では、フィールドに対して定義された[照合タイプ](#)が使用されます。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

フィールドの定義済み照合タイプが SQLUPPER である場合、MAX は文字列をすべて大文字で返します。したがって、SELECT MAX(Name) は、データの元の大文字/小文字に関係なく 'ZWIG' を返します。ただし、比較は大文字の照合を使用して行われるため、HAVING Name=MAX(Name) 節では、'Zwig'、'ZWIG'、および 'zwig' の Name 値が含まれる行が選択されます。

数値の場合、返される小数桁数は、expression の小数桁数と同じになります。

データ・フィールドの NULL 値は、MAX 集約関数値を得る場合は無視されます。クエリから行が返されない場合や、すべての行のデータ・フィールド値が NULL の場合、MAX は NULL を返します。

## 引数

### ALL

すべての値に集約関数を適用する引数 (オプション)。ALL は、MAX によって返される値に影響を与えません。これは、SQL-92 の互換性のために用意されたものです。

### DISTINCT

それぞれの一意な値が考慮されるように指定する [DISTINCT 節](#) (オプション)。DISTINCT は、MAX によって返される値に影響を与えません。これは、SQL-92 の互換性のために用意されたものです。

## expression

任意の有効な式。通常、最大値が返される値を持つ列の名前。

## %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです (オプション)。%FOREACH の詳細は、“[SELECT](#)” を参照してください。

## %AFTERHAVING

HAVING 節にある条件を適用する引数 (オプション)。

MAX が返す値のデータ型は、expression の[データ型](#)と同じです。

注釈 MAX は、集約関数またはウィンドウ関数として指定できます。このリファレンス・ページでは、集約関数としての MAX の使用法について説明します。ウィンドウ関数としての MAX については、“[ウィンドウ関数の概要](#)” で説明しています。

## 現在のトランザクションで発生する変更

すべての集約関数と同様に、MAX も必ず、現在のトランザクションの分離レベルに関係なく、コミットされていない変更も含めてデータの現在の状態を返します。詳細は、“[SET TRANSACTION](#)” と “[START TRANSACTION](#)” を参照してください。

## 例

以下のクエリは、Sample.Employee データベース内の最高 (最大) 給与額を返します。

### SQL

```
SELECT '$' || MAX(Salary) As TopSalary
FROM Sample.Employee
```

以下のクエリは、給与額が \$25,000 より少ない従業員が 1 人以上存在する州ごとに行を 1 つ返します。この行には、%AFTERHAVING キーワードを使用して給与額が \$25,000 より少ない従業員の最大給与額を返し、さらにその州の全従業員の最小給与額と最大給与額を返します。

### SQL

```
SELECT Home_State,
       '$' || MAX(Salary %AFTERHAVING) AS MaxSalaryBelow25K,
       '$' || MIN(Salary) AS MinSalary,
       '$' || MAX(Salary) AS MaxSalary
FROM Sample.Employee
GROUP BY Home_State
HAVING Salary < 25000
ORDER BY Home_State
```

以下のクエリは、Sample.Employee データベース内で検出された照合順の最低 (最小) および最高 (最大) の名前を返します。

### SQL

```
SELECT Name, MIN(Name), MAX(Name)
FROM Sample.Employee
```

MIN および MAX は、比較する前に Name の値を大文字に変換します。

以下のクエリは、Sample.Employee データベース内の Home\_State が 'VT' である従業員の最高 (最大) 給与額を返します。

### SQL

```
SELECT MAX(Salary)
      FROM Sample.Employee
     WHERE Home_State = 'VT'
```

以下のクエリは、Sample.Employee データベース内の Home\_State ごとに、従業員数および最高 (最大) の従業員給与額を返します。

### SQL

```
SELECT Home_State,
       COUNT(Home_State) As NumEmployees,
       MAX(Salary) As TopSalary
  FROM Sample.Employee
 GROUP BY Home_State
 ORDER BY TopSalary
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [MIN 集約関数](#)

# MIN (SQL)

指定された列内の最小データ値を返す集約関数です。

## 構文

```
MIN([ ALL | DISTINCT [BY(col-list)] ]
    expression
    [ %FOREACH(col-list) ] [ %AFTERHAVING ])
```

## 概要

MIN [集約関数](#)は、expression の最小値を返します。一般的に expression は、クエリで返される複数行の中のフィールドの名前（または、フィールド名を 1 つ以上含む式）です。

MIN は、テーブルまたはビューを参照する [SELECT](#) クエリまたは SELECT サブクエリで使用できます。MIN は、一般のフィールド値と共に SELECT リストや HAVING 節で表示できます。

MIN は、WHERE 節では使用できません。SELECT がサブクエリの場合を除いて、JOIN の ON 節では MIN を使用できません。

その他の大半の集約関数と同様に、MIN は[ストリーム・フィールド](#)に適用できません。これを実行しようすると、SQLCODE -37 エラーが生成されます。

その他の大半の集約関数とは異なり、MIN(DISTINCT BY(col2) col1) を含めて、ALL および DISTINCT キーワードは MIN 内で何も実行しません。これらは、SQL-92 の互換性のために用意されたものです。

## データ値

MIN によって使用される指定されたフィールドは、数値または非数値です。数値データ型フィールドの場合、最小値は最も小さな数値として定義されます。したがって -7 の方が -3 よりも小さくなります。非数値データ型フィールドの場合、最小値は文字列の[照合順](#)で最も小さな値として定義されます。したがって '-3' の方が '-7' よりも小さくなります。

空文字列('') 値は CHAR(0) として処理されます。

述語では、フィールドに対して定義された[照合タイプ](#)が使用されます。既定では、文字列データ型フィールドは大文字と小文字が区別されない SQLUPPER 照合で定義されます。[現在のネームスペースにおける既定の文字列の照合](#)を定義し、[フィールド/プロパティの定義における既定以外のフィールドの照合タイプ](#)を指定することができます。

フィールドの定義済み照合タイプが SQLUPPER である場合、MIN は文字列をすべて大文字で返します。したがって、SELECT MIN(Name) は、データの元の大文字/小文字に関係なく 'AARON' を返します。ただし、比較は大文字の照合を使用して行われるため、HAVING Name=MIN(Name) 節では、'Aaron'、'AARON'、および 'aaron' の Name 値が含まれる行が選択されます。

数値の場合、返される小数桁数は、expression の小数桁数と同じになります。

データ・フィールドの NULL 値は、MIN 集約関数値を得る場合は無視されます。クエリから行が返されない場合や、すべての行に対するデータ・フィールド値が NULL の場合、MIN は NULL を返します。

## 引数

### ALL

すべての値に集約関数を適用する引数（オプション）。ALL は、MIN によって返される値に影響を与えません。これは、SQL-92 の互換性のために用意されたものです。

### DISTINCT

それぞれの一意な値が考慮されるように指定する引数（オプション）。DISTINCT は、MIN によって返される値に影響を与えません。これは、SQL-92 の互換性のために用意されたものです。

## expression

任意の有効な式。通常、最小値が返される値を持つ列の名前。

## %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです (オプション)。%FOREACH の詳細は、“[SELECT](#)” を参照してください。

## %AFTERHAVING

HAVING 節にある条件を適用する引数 (オプション)。

MIN が返す値のデータ型は、expression のデータ型と同じです。

注釈 MIN は、集約関数またはウィンドウ関数として指定できます。このリファレンス・ページでは、集約関数としての MIN の使用法について説明します。ウィンドウ関数としての MIN については、“[ウィンドウ関数の概要](#)” で説明しています。

## 現在のトランザクションで発生する変更

すべての集約関数と同様に、MIN も必ず、現在のトランザクションの分離レベルに関係なく、コミットされていない変更も含めてデータの現在の状態を返します。詳細は、“[SET TRANSACTION](#)” と “[START TRANSACTION](#)” を参照してください。

## 例

以下の例では、Salary 金額にドル記号 (\$) を連結します。

以下のクエリは、Sample.Employee データベース内の最低 (最小) 給与額を返します。

### SQL

```
SELECT '$' || MIN(Salary) AS LowSalary
FROM Sample.Employee
```

以下のクエリは、給与額が \$75,000 より多い従業員が 1 人以上存在する州ごとに行を 1 つ返します。この行には、%AFTERHAVING キーワードを使用して給与額が \$75,000 より多い従業員の最小給与額を返し、さらにその州の全従業員の最小給与額と最大給与額を返します。

### SQL

```
SELECT Home_State,
       '$' || MIN(Salary %AFTERHAVING) AS MinSalaryAbove75K,
       '$' || MIN(Salary) AS MinSalary,
       '$' || MAX(Salary) AS MaxSalary
FROM Sample.Employee
GROUP BY Home_State
HAVING Salary > 75000
ORDER BY MinSalaryAbove75K
```

以下のクエリは、Sample.Employee データベース内で検出された照合順の最低 (最小) および最高 (最大) の名前を返します。

### SQL

```
SELECT Name, MIN(Name), MAX(Name)
FROM Sample.Employee
```

MIN および MAX は、比較する前に Name の値を大文字に変換します。

以下のクエリは、Sample.Employee データベース内の Home\_State が VT である従業員の最低 (最小) 給与額を返します。

## SQL

```
SELECT MIN(Salary)
      FROM Sample.Employee
     WHERE Home_State = 'VT'
```

以下のクエリは、Sample.Employee データベース内の Home\_State ごとに、従業員数および最低（最小）の従業員給与額を返します。

## SQL

```
SELECT Home_State,
       COUNT(Home_State) As NumEmployees,
       MIN(Salary) As LowSalary
  FROM Sample.Employee
 GROUP BY Home_State
 ORDER BY LowSalary
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [MAX 集約関数](#)



## STDDEV、STDDEV\_SAMP、STDDEV\_POP (SQL)

データ・セットの統計標準偏差を返す集約関数。

### 構文

```
STDDEV([ ALL | DISTINCT [BY(col-list)] ]
      expression
      [ %FOREACH(col-list) ] [ %AFTERHAVING ])

STDDEV_SAMP([ ALL | DISTINCT [BY(col-list)] ]
            expression
            [ %FOREACH(col-list) ] [ %AFTERHAVING ])

STDDEV_POP([ALL | DISTINCT [BY(col-list)]]
           expression
           [%FOREACH(col-list)] [%AFTERHAVING])
```

### 説明

これら 3 つの標準偏差集約関数は、NULL 値を破棄した後に、expression の値分布の統計標準偏差を返します。つまり、データ・セットの平均値からの標準偏差が正数で表されます。返される値が大きければ、値のデータ・セット内の分散が大きいということになります。

STDDEV、STDDEV\_SAMP (サンプル)、および STDDEV\_POP (母集団) の各関数は、対応する分散集約関数から派生しています。

STDDEV	VARIANCE
STDDEV_SAMP	VAR_SAMP
STDDEV_POP	VAR_POP

標準偏差は、対応する分散値の平方根です。詳細は、それぞれの分散集約関数を参照してください。

これらの標準偏差関数は、テーブルまたはビューを参照する [SELECT](#) クエリまたはサブクエリで使用できます。これらの関数は、一般のフィールド値と共に SELECT リストや HAVING 節で表示できます。

これらの標準偏差関数は、WHERE 節では使用できません。SELECT がサブクエリの場合を除いて、JOIN の ON 節ではそれらを使用できません。

これらの標準偏差関数は、expression がデータ型 DOUBLE でない限り (この場合、この関数はデータ型 DOUBLE を返します)、精度が 36 で小数桁が 17 のデータ型 NUMERIC の値を返します。

通常、これらの関数は、数値を含むフィールドまたは式に適用されます。これらの関数では、空文字列("")などの数値でない値はゼロ (0) として扱われます。

これらの標準偏差関数は、データ・フィールド内の NULL 値を無視します。クエリから行が返されない場合や、すべての行のデータ・フィールド値が NULL の場合、これらの関数は NULL を返します。

標準偏差関数は、すべての集約関数と同様に、オプションの [DISTINCT](#) 節を取ることができます。STDDEV(DISTINCT col1) は、これらの個別 (一意) の col1 フィールド値の標準偏差を返します。STDDEV(DISTINCT BY(col2) col1) は、col2 値が個別 (一意) であるレコードの col1 フィールド値の標準偏差を返します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

### 引数

#### ALL

標準偏差関数が expression のすべての値の標準偏差を返すように指定する引数 (オプション)。キーワードが指定されていない場合は、これが既定になります。

## DISTINCT

標準偏差関数が個別（一意）の expression 値の標準偏差を返すように指定する **DISTINCT** 節（オプション）。DISTINCT で **BY(col-list)** 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。

### expression

任意の有効な式。通常は、標準偏差の分析対象となるデータ値を含む列の名前を指定します。

### %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです（オプション）。%FOREACH の詳細は、“[SELECT](#)” を参照してください。

### %AFTERHAVING

**HAVING** 節にある条件を適用する引数（オプション）。

## 現在のトランザクション中の変更

すべての集約関数と同様に、標準偏差関数も必ず、現在のトランザクションの分離レベルに関係なく、コミットされていない変更も含めてデータの現在の状態を返します。詳細は、“[SET TRANSACTION](#)” および “[START TRANSACTION](#)” を参照してください。

## 例

以下の例では、STDDEV を使用して、Sample.Employee 内の従業員の年齢における標準偏差と、1 人以上の従業員で表される個別の年齢における標準偏差を返します。

```
SELECT STDDEV(Age) AS AgeSD,STDDEV(DISTINCT Age) AS PerAgeSD
FROM Sample.Employee
```

以下の例では、STDDEV\_POP を使用して、Sample.Employee 内の従業員の年齢における母標準偏差と、1 人以上の従業員で表される個別の年齢における標準偏差を返します。

```
SELECT STDDEV_POP(Age) AS AgePopSD,STDDEV_POP(DISTINCT Age) AS PerAgePopSD
FROM Sample.Employee
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [VARIANCE、VAR\\_SAMP、VAR\\_POP](#) 集約関数
- ・ [AVG](#) 集約関数
- ・ [COUNT](#) 集約関数

## SUM (SQL)

指定の列の値の合計値を返す、集約関数です。

### 構文

```
SUM([ ALL | DISTINCT [BY(col-list)] ]  
    expression  
    [ %FOREACH(col-list) ] [ %AFTERHAVING ])
```

### 概要

SUM [集約関数](#)は、expression の値の合計を返します。一般的に expression は、クエリで返される複数行の中のフィールドの名前 (または、フィールド名を 1 つ以上含む式) です。

SUM は、テーブルまたはビューを参照する [SELECT](#) クエリやサブクエリで使用できます。SUM は、一般のフィールド値と共に SELECT リストや HAVING 節で使用できます。

SUM は、WHERE 節では使用できません。SELECT がサブクエリの場合を除いて、JOIN の ON 節では、SUM を使用できません。

SUM は、すべての集約関数と同様に、オプションの [DISTINCT](#) 節を取ることができます。SUM(DISTINCT col1) は、これらの個別 (一意) の col1 フィールド値のみの合計値を計算します。SUM(DISTINCT BY(col2) col1) は、col2 値が個別 (一意) であるレコードのこれらの col1 フィールド値のみの合計値を計算します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

### データ値

SUM は、データ型が INT、SMALLINT、または TINYINT の expression に対して、データ型 INTEGER を返します。SUM は、データ型が BIGINT の expression に対して、データ型 BIGINT を返します。SUM は、データ型が DOUBLE の expression に対して、データ型 DOUBLE を返します。他のすべての数値データ型に対して、SUM はデータ型 NUMERIC を返します。

SUM は、有効桁数が 18 の値を返します。返り値の小数桁数は expression の小数桁数と同じですが、以下の例外があります。expression が数値で、そのデータ型が VARCHAR または VARBINARY の場合、返り値の小数桁数は 8 になります。

既定では、集約関数は Display 値ではなく、Logical (内部) データ値を使用します。

通常、SUM は数値を含むフィールドまたは式に適用されます。タイプ・チェックはごくわずかしき実行されないため、(ほとんど意味はありませんが) 非数値フィールドに対して実行できます。SUM では、空文字列 (") のように数値でない値はゼロ (0) として扱われます。expression のデータ型が VARCHAR の場合、ODBC または JDBC への返り値は DOUBLE データ型になります。

データ・フィールドの NULL 値は、SUM 集約関数値を得る場合は無視されます。クエリから行が返されない場合や、すべての行のデータ・フィールド値が NULL の場合、SUM は NULL を返します。

### 最適化

SUM 計算の SQL 最適化では、[ビットスライス・インデックス](#)がフィールドに定義されている場合、それを使用することができます。

### 引数

#### ALL

SUM で、expression にある値すべての値の合計を返すことを指定する引数 (オプション)。キーワードが指定されていない場合は、これが既定になります。

## DISTINCT

SUM が expression の個別 (一意) の値の合計を返すように指定する **DISTINCT** 節 (オプション)。DISTINCT で **BY(col-list)** 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。

### expression

任意の有効な式。普通は、合計値算出の対象となるデータ値を含む列の名前を指定します。

### %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです (オプション)。**%FOREACH** の詳細は、“[SELECT](#)” を参照してください。

### %AFTERHAVING

**HAVING** 節にある条件を適用する引数 (オプション)。

SUM は expression と同じ**データ型**を返します。ただし、TINYINT、SMALLINT、および INTEGER は例外で、これらはすべて INTEGER データ型として返されます。

**注釈** SUM は、集約関数またはウィンドウ関数として指定できます。このリファレンス・ページでは、集約関数としての SUM の使用法について説明します。ウィンドウ関数としての SUM については、“[ウィンドウ関数の概要](#)” で説明しています。

## 現在のトランザクションで発生する変更

すべての集約関数と同様に、SUM も必ず、現在のトランザクションの分離レベルに関係なく、コミットされていない変更も含めてデータの現在の状態を返します。詳細は、“[SET TRANSACTION](#)” および “[START TRANSACTION](#)” を参照してください。

## 例

以下の例では、Salary 金額にドル記号 (\$) を連結します。

以下のクエリは、Sample.Employee データベースに記録されている全従業員の給与の合計を返します。

### SQL

```
SELECT '$' || SUM(Salary) AS Total_Payroll
FROM Sample.Employee
```

以下のクエリは、**%AFTERHAVING** を使用して、給与額が \$80,000 より多い人が 1 人以上いる各州について、すべての給与額の合計と \$80,000 を超える給与額の合計を返します。

### SQL

```
SELECT Home_State,
       '$' || SUM(Salary) AS Total_Payroll,
       '$' || SUM(Salary %AFTERHAVING) AS Exec_Payroll
FROM Sample.Employee
GROUP BY Home_State
HAVING Salary > 80000
ORDER BY Home_State
```

以下のクエリは、Sample.Employee データベースに記録されている役職ごとの給与の合計と平均値を返します。

## SQL

```
SELECT Title,
       '$' || SUM(Salary) AS Total,
       '$' || AVG(Salary) AS Average
FROM Sample.Employee
GROUP BY Title
ORDER BY Average
```

以下のクエリは、算術式と共に使用される SUM を示しています。Sample.Employee データベースに記録されている各役職に対して、現在の給与の合計と支払いに 10% 増加した給与の合計を返します。

## SQL

```
SELECT Title,
       '$' || SUM(Salary) AS BeforeRaises,
       '$' || SUM(Salary * 1.1) AS AfterRaises
FROM Sample.Employee
GROUP BY Title
ORDER BY Title
```

以下のクエリは、CASE 文を使用して論理式と共に使用される SUM を示しています。すべての定額給従業員をカウントし、SUM を使用して \$90,000 以上の所得があるすべての定額給従業員をカウントします。

## SQL

```
SELECT COUNT(Salary) As AllPaid,
       SUM(CASE WHEN (Salary >= 90000)
              THEN 1 ELSE 0 END) As TopPaid
FROM Sample.Employee
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [AVG 集約関数](#)
- ・ [COUNT 集約関数](#)

# VARIANCE、VAR\_SAMP、VAR\_POP (SQL)

データ・セットの統計的分散を返す集約関数。

## 構文

```
VARIANCE([ ALL | DISTINCT [BY(col-list)] ]
expression
[ %FOREACH(col-list) ] [ %AFTERHAVING ])

VAR_SAMP([ ALL | DISTINCT [BY(col-list)] ]
expression
[ %FOREACH(col-list) ] [ %AFTERHAVING ])

VAR_POP([ALL | DISTINCT [BY(col-list)] ]
expression
[%FOREACH(col-list)] [%AFTERHAVING])
```

## 説明

これら 3 つの分散集約関数は、NULL 値を破棄した後、expression の値の統計的分散を返します。つまり、データ・セットの平均値からの分散量が正数で表されます。返される値が大きければ、値のデータ・セット内の分散が大きいということになります。InterSystems SQL は、それらの各分散関数に対応する、標準偏差を返す集約関数も提供しています。

この統計的な分散の派生方法には、以下のようないくつかの差異があります。

- ・ VARIANCE：データ・セット内のすべての値が同じ値を持つ（変動なしの）場合は、0 を返します。データ・セットが 1 つのみの値で構成される（変動の可能性なしの）場合は、0 を返します。データ・セットに値がない場合は、NULL を返します。

VARIANCE の計算は以下ようになります。

$$\frac{(\text{SUM}(\text{expression}^2) * \text{COUNT}(\text{expression})) - \text{SUM}(\text{expression})^2}{\text{COUNT}(\text{expression}) * (\text{COUNT}(\text{expression}) - 1)}$$

- ・ VAR\_SAMP：サンプル分散。データ・セット内のすべての値が同じ値を持つ（変動なしの）場合は、0 を返します。データ・セットが 1 つのみの値で構成される（変動の可能性なしの）場合は、NULL を返します。データ・セットに値がない場合は、NULL を返します。VARIANCE と同じ分散計算を使用します。
- ・ VAR\_POP：母分散。データ・セット内のすべての値が同じ値を持つ（変動なしの）場合は、0 を返します。データ・セットが 1 つのみの値で構成される（変動の可能性なしの）場合は、0 を返します。データ・セットに値がない場合は、NULL を返します。

VAR\_POP の計算は以下ようになります。

$$\frac{(\text{SUM}(\text{expression}^2) * \text{COUNT}(\text{expression})) - (\text{SUM}(\text{expression})^2)}{\text{COUNT}(\text{expression}) * 2}$$

これらの分散集約関数は、テーブルまたはビューを参照する **SELECT** クエリまたはサブクエリで使用できます。これらの関数は、一般のフィールド値と共に SELECT リストや HAVING 節で表示できます。

これらの分散集約関数は、WHERE 節では使用できません。SELECT がサブクエリの場合を除いて、JOIN の ON 節ではそれらを使用できません。

これらの分散集約関数は、expression がデータ型 DOUBLE でない限り（この場合、この関数はデータ型 DOUBLE を返します）、精度が 36 で小数桁が 17 のデータ型 NUMERIC の値を返します。

通常、これらの分散集約関数は、数値を含むフィールドまたは式に適用されます。これらの関数では、空文字列 (") などの数値でない値はゼロ (0) として扱われます。

これらの分散集約関数は、データ・フィールド内の NULL 値を無視します。クエリから行が返されない場合や、すべての行のデータ・フィールド値が NULL の場合、これらの関数は NULL を返します。

統計的分散関数は、すべての集約関数と同様に、オプションの **DISTINCT** 節を取ることができます。VARIANCE(DISTINCT col1) は、これらの個別（一意）の col1 フィールド値の分散を返します。VARIANCE(DISTINCT BY(col2) col1) は、col2 値が個別（一意）であるレコードの col1 フィールド値の分散を返します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

## 引数

### ALL

統計的分散関数が expression のすべての値の分散を返すように指定する引数（オプション）。キーワードが指定されていない場合は、これが既定になります。

### DISTINCT

統計的分散関数が個別（一意）の expression 値の分散を返すように指定する **DISTINCT** 節（オプション）。DISTINCT で BY(col-list) 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。

### expression

任意の有効な式。通常は、分散の分析対象となるデータ値を含む列の名前を指定します。

### %FOREACH(col-list)

1 つの列名、またはコンマで区切った複数の列名のリストです（オプション）。%FOREACH の詳細は、“[SELECT](#)”を参照してください。

### %AFTERHAVING

**HAVING** 節にある条件を適用する引数（オプション）。

## 現在のトランザクション中の変更

すべての集約関数と同様に、分散関数も必ず、現在のトランザクションの分離レベルに関係なく、コミットされていない変更も含めてデータの現在の状態を返します。詳細は、“[SET TRANSACTION](#)” および “[START TRANSACTION](#)” を参照してください。

## 例

以下の例では、VARIANCE を使用して、Sample.Employee 内の従業員の年齢における分散と、1 人以上の従業員で表される個別の年齢における分散を返します。

```
SELECT VARIANCE(Age) AS AgeVar, VARIANCE(DISTINCT Age) AS PerAgeVar
FROM Sample.Employee
```

以下の例では、VAR\_POP を使用して、Sample.Employee 内の従業員の年齢における母分散と、1 人以上の従業員で表される個別の年齢における分散を返します。

```
SELECT VAR_POP(Age) AS AgePopVar, VAR_POP(DISTINCT Age) AS PerAgePopVar
FROM Sample.Employee
```

## 関連項目

- ・ [集約関数の概要](#)
- ・ [AVG 集約関数](#)



- ・ [COUNT](#) 集約関数
- ・ [STDDEV](#)、[STDDEV\\_SAMP](#)、[STDDEV\\_POP](#) 集約関数

## XMLAGG (SQL)

値を連結した文字列を作成する集約関数です。

### 構文

```
XMLAGG([ ALL | DISTINCT [BY(col-list)] ]
      string-expr
      [ %FOREACH(col-list) ] [ %AFTERHAVING ] )
```

### 引数

引数	説明
ALL	オプション – XMLAGG が string-expr のすべての値を結合した文字列を返すように指定します。キーワードが指定されていない場合は、これが既定になります。
DISTINCT	オプション – 一意の string-expr 値のみを含む連結文字列を XMLAGG が返すように指定する <a href="#">DISTINCT 節</a> 。DISTINCT で BY(col-list) 従属節を指定できます。col-list には 1 つのフィールド、またはコンマ区切りのフィールドのリストを指定できます。
string-expr	文字列として評価する SQL 式。一般的には、これはデータを取得する列の名前です。
%FOREACH(col-list)	オプション – 列名、またはコンマで区切られた列名のリスト。%FOREACH の詳細は、“ <a href="#">SELECT</a> ” を参照してください。
%AFTERHAVING	オプション – <a href="#">HAVING</a> 節にある条件を適用します。

### 概要

XMLAGG [集約関数](#)は、string-expr のすべての値を連結した文字列を返します。戻り値は VARCHAR データ型です。

- ・ シンプルな XMLAGG (または XMLAGG ALL) は、選択された行の string-expr のすべての値で構成される連結文字列を含む文字列を返します。string-expr が NULL の行は、無視されます。

以下の 2 つの例は両方とも、Sample.Person テーブルの Home\_State 列内にあるすべての値を連結した文字列を 1 つの値として返します。

#### SQL

```
SELECT XMLAGG(Home_State) AS All_State_Values
FROM Sample.Person
```

#### SQL

```
SELECT XMLAGG(ALL Home_State) AS ALL_State_Values
FROM Sample.Person
```

この結合文字列には重複値が含まれます。

- ・ XMLAGG DISTINCT は、選択された行の string-expr に対応するすべての個別 (一意の) 値で構成される連結文字列を返します (XMLAGG(DISTINCT col1))。string-expr が NULL の行は、無視されます。XMLAGG(DISTINCT BY(col2) col1) は、col2 値が個別 (一意) であるレコードのこれらの col1 フィールド値のみが含まれる連結文字列を返します。ただし、個別の col2 値には、個別値として NULL が 1 つ含まれる場合があります。

string-expr が NULL の行は、返り値から省略されます。空でない文字列の値が 1 つ以上返される場合、string-expr が空の文字列 (') の行は、返り値から省略されます。NULL でない string-expr の値のみが空の文字列 (') である場合、返り値は 1 つの空の文字列となります。

XMLAGG はデータ・ストリーム・フィールドをサポートしません。string-expr にストリーム・フィールドを指定すると、SQLCODE -37 が返されます。

## XML および XMLAGG

XMLAGG の一般的な使用方法の 1 つに、列の各データ・アイテムにタグ付けすることがあります。これは以下の例で示すように、XMLAGG と XMLELEMENT を組み合わせて行います。

### SQL

```
SELECT XMLAGG(XMLELEMENT("para",Home_State))
FROM Sample.Person
```

この結果、以下のような文字列が出力されます。

```
<para>LA</para><para>MN</para><para>LA</para><para>NH</para><para>ME</para>...
```

## XMLAGG と ORDER BY

XMLAGG 関数は、複数の行のテーブル列の値を 1 つの文字列に連結します。すべての集約フィールドが評価された後に ORDER BY 節がクエリ結果セットに適用されるため、ORDER BY がこの文字列内の値の順序に直接影響することはありません。特定の状況下では、XMLAGG の結果が順番に表示されることがありますが、この順番を信頼すべきではありません。特定の集約結果値内にリストされる値を、明示的に順序付けすることはできません。

## 関連する集約関数

- XMLAGG は、連結した値の文字列を返します。
- [LIST](#) は、コンマで区切られた値のリストを返します。
- [%DLIST](#) は、各値の要素を含む InterSystems IRIS リストを返します。
- [JSON\\_ARRAYAGG](#) は、JSON 配列の値を返します。

## 例

以下の例は、Sample.Person テーブルの FavoriteColors 列内にあるすべての個別値の連結文字列を作成します。したがって、すべての行の All\_Colors 列の値は同じになります。ある行が FavoriteColors に NULL 値を持つと、この値は連結文字列に含まれなくなります。データ値は、内部形式で返されます。

### SQL

```
SELECT Name,FavoriteColors,
       XMLAGG(DISTINCT FavoriteColors) AS All_Colors_In_Table
FROM Sample.Person
ORDER BY FavoriteColors
```

以下の例は、“A” で始まる Home\_State 値の連結文字列を返します。これは、個別の Home\_State 値 (DISTINCT Home\_State)、個別の Home\_City 値に対応する Home\_State 値 (DISTINCT BY(Home\_City) Home\_State、Home\_City に対応する一意の NULL が 1 つ含まれる場合があります)、およびすべての Home\_State 値を返します。

### SQL

```
SELECT XMLAGG(DISTINCT Home_State) AS DistStates,
       XMLAGG(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       XMLAGG(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

以下の例は、各州の Home\_City 列内にあるすべての個別値の連結文字列を作成します。同じ州のすべての行には、その州の個別の市の値すべてのリストが含まれます。

## SQL

```
SELECT Home_State, Home_City,
       XMLAGG(DISTINCT Home_City %FOREACH(Home_State)) AS All_Cities_In_State
FROM Sample.Person
ORDER BY Home_State
```

以下の例では、%AFTERHAVING キーワードを使用します。HAVING 節の条件（“C” または “K” で始まる名前）を満たす Name 値が 1 つ以上ある各 Home\_State の行を返します。最初の XMLAGG 関数は、該当の州にある名前をすべて連結して構成した文字列を返します。2 番目の XMLAGG 関数は、HAVING 節の条件を満たす名前のみを連結して構成した文字列を返します。

## SQL

```
SELECT Home_State,
       XMLAGG(Name) AS AllNames,
       XMLAGG(Name %AFTERHAVING) AS HaveClauseNames
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'C%' OR Name LIKE 'K%'
ORDER BY Home_State
```

例えば、以下の **AutoClub** のようなテーブルがあるとします。

Name	Make	Model	Year
Smith,Joe	Pontiac	Firebird	1971
Smith,Joe	Saturn	SW2	1997
Smith,Joe	Pontiac	Bonneville	1999
Jones,Scott	Ford	Mustang	1966
Jones,Scott	Mazda	Miata	2000

クエリは以下のとおりです。

## SQL

```
SELECT DISTINCT Name, XMLAGG(Make) AS String_Of_Makes
FROM AutoClub WHERE Name = 'Smith,Joe'
```

これは、以下を返します。

Name	String_Of_Makes
Smith,Joe	PontiacSaturnPontiac

クエリは以下のとおりです。

## SQL

```
SELECT DISTINCT Name, XMLAGG(DISTINCT Make) AS String_Of_Makes
FROM AutoClub WHERE Name = 'Smith,Joe'
```

これは、以下を返します。

Name	String_Of_Makes
Smith,Joe	PontiacSaturn

## 関連項目

- ・ [集約関数の概要](#)
- ・ [%DLIST 集約関数](#)
- ・ [JSON\\_ARRAYAGG 集約関数](#)
- ・ [LIST 集約関数](#)
- ・ [XMLELEMENT 関数](#)
- ・ [SELECT 文](#)



# SQL ウィンドウ関数



## ウィンドウ関数の概要

集約およびランキングの計算のための行当たりの“ウィンドウ・フレーム”を指定する関数。

### ウィンドウ関数と集約関数

ウィンドウ関数は、WHERE GROUP BY および HAVING 節が適用された後、SELECT クエリで選択された行に対して作用します。

ウィンドウ関数は、行のグループのフィールドの値を結合し、結果セットに生成された列の各行の値を返します。

ウィンドウ関数は、複数の行からの結果を結合するという点では[集約関数](#)と似ていますが、行自体を結合しないという点では集約関数と異なります。ただし、集約関数 [AVG\(\)](#)、[MIN\(\)](#)、[MAX\(\)](#)、[SUM\(\)](#) は、ウィンドウ関数としても呼び出されます。このコンテキストで、各行は、対応するウィンドウ・フレーム内の行のグループに対する関数の呼び出し結果を受け取ります。

### ウィンドウ関数の構文

ウィンドウ関数は、[SELECT](#) クエリ内の select-item として指定されます。ウィンドウ関数は、SELECT クエリの [ORDER BY](#) 節に指定することもできます。

ウィンドウ関数は、PARTITION BY 節、ORDER BY 節、および ROWS 節により指定された行ごとのウィンドウに関連してタスクを実行し、各行に対して値を返します。これら3つの節はオプションですが、指定する場合は、次の構文に示す順序で指定する必要があります。

```
window-function() OVER ( [ PARTITION BY partfield ]           [ ORDER BY orderfield ]           [ ROWS framestart
] | [ ROWS BETWEEN framestart AND frameend ] )
```

ここで、framestart と frameend は以下のようにできます。

```
UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW | UNBOUNDED FOLLOWING | offset FOLLOWING
```

キーワードとウィンドウ関数の名前では、大文字と小文字は区別されません。

### 簡単な例

CityTable には以下の値の行が含まれます。

Name	City
Able	New York
Betty	Boston
Charlie	Paris
Davis	Boston
Eve	Paris
Francis	Paris
George	London
Beatrix	Paris

ROW\_NUMBER() ウィンドウ関数は、指定されたウィンドウに基づいて、各行に一意の連続した整数を割り当てます。

```
SELECT Name, City, ROW_NUMBER() OVER (PARTITION BY City) FROM CityTable
```

この例では、City の値で行を分割し、以下を返します。

Name	City	Window_3
Able	New York	1
Betty	Boston	1
Charlie	Paris	1
Davis	Boston	2
Eve	Paris	2
Francis	Paris	3
George	London	1
Beatrix	Paris	4

```
SELECT Name, City, ROW_NUMBER() OVER (ORDER BY City) FROM CityTable
```

この例では、すべての行を単一のパーティションとして扱います。City の値で行を並べ替え、以下を返します。

Name	City	Window_3
Able	New York	4
Betty	Boston	1
Charlie	Paris	5
Davis	Boston	2
Eve	Paris	6
Francis	Paris	7
George	London	3
Beatrix	Paris	8

```
SELECT Name, City, ROW_NUMBER() OVER (Partition BY City ORDER BY Name) FROM CityTable
```

この例では、City の値で行を分割し、各 City のパーティションを Name の値で並べ替え、以下を返します。

Name	City	Window_3
Able	New York	1
Betty	Boston	1
Charlie	Paris	2
Davis	Boston	2
Eve	Paris	3
Francis	Paris	4
George	London	1
Beatrix	Paris	1

## NULL

PARTITION BY 節は、NULL (値が割り当てられていない) のフィールドを持つ行を、分割された 1 つのグループとして処理します。例えば、`ROW_NUMBER() OVER (PARTITION BY City)` は、City 値が 'Paris' の行に連続した整数を割り当てると同様に、City 値のない行に連続した整数を割り当てます。

ORDER BY 節は、NULL のフィールドを持つ行を、割り当てられているどの値よりも前に順序付けられている (最も小さい照合値を持つ) ものとして処理します。例えば、`ROW_NUMBER() OVER (ORDER BY City)` は、最初に City の値がない行に連続した整数を割り当ててから、City 値がある行に照合順で連続した整数を割り当てます。

ROWS 節は、NULL (値が割り当てられていない) のフィールドを、ゼロの値を持つものとして処理します。例えば、`SUM(Scores) OVER (ORDER BY Scores ROWS 1 PRECEDING)/2` は、Scores 値  $((0 + 0) / 2)$  を持たないすべての行に 0.00 を割り当て、これに 0 を加えて 2 で割ることで最初の Scores 値を処理します。

## サポートされているウィンドウ関数

以下のウィンドウ関数がサポートされます。

- AVG(field) – 指定したウィンドウ・フレーム内の行の field 列の値の平均を、そのウィンドウ・フレームのすべての行に割り当てます。例えば、`AVG(Salary) OVER (PARTITION BY Department) FROM Company.Employee` を使用して、各従業員の給与を、その従業員の部門内の従業員の給与の平均と比較できます。AVG() は [ROWS 節](#) をサポートします。使用方法の詳細は、[AVG\(\) 関数のリファレンス・ページ](#) を参照してください。
- COUNT(\* | field) – 指定したウィンドウ・フレーム内の各行に 1 から始まる数を割り当てます。field が指定されている場合、カウントは field のコンテンツが NULL でないときにのみインクリメントされます。この引数が指定されていない場合、カウントはすべての行でインクリメントされます。
- CUME\_DIST() – 指定したウィンドウ・フレーム内のすべての行に累積分布値を割り当てます。累積分布は、現在の行の値以下の値を持つ行をカウントし、そのカウントをウィンドウ内の行の総数で除算することによって計算されます。累積分布が計算される列の名前は、ORDER BY 節で指定されます。
- DENSE\_RANK() – 同じウィンドウ・フレーム内の各行に、1 から始まるランキングを表す整数を割り当てます。このランキングを表す整数は、RANK() ウィンドウ関数の場合とは異なり、常に連続した値となります。ウィンドウ関数フィールドに同じ値を含む行が複数ある場合は、ランキングの整数に重複値を含めることができます。
- FIRST\_VALUE(field) – 指定したウィンドウ・フレーム内の最初の行 (`ROW_NUMBER()=1`) の field 列の値を、そのウィンドウ・フレームのすべての行に割り当てます。例えば、`FIRST_VALUE(Country) OVER (PARTITION BY City)` のようになります。FIRST\_VALUE() は [ROWS 節](#) をサポートします。すべての値の前に NULL を照合し、最初の行の field の値が NULL の場合にウィンドウ内のすべての行が NULL になるようにすることに注意してください。
- LAST\_VALUE(field) – 指定したウィンドウ・フレーム内の最後の行の field 列の値を、そのウィンドウ・フレームのすべての行に割り当てます。LAST\_VALUE() は [ROWS 節](#) をサポートします。
- LAG(field[, offset[, default]]) – 指定したウィンドウ・フレーム内の指定の行の offset 行前にある field 列の値を割り当てます。offset が指定されていない場合、この関数は、既定で、指定された行の 1 行前の field 列の値を割り当てます。既定では、ウィンドウ・フレーム内で、指定された行の offset 行前に行がない場合、LAG() は値 NULL を割り当てます。ユーザは、値 default を含めることにより、これらの条件下で、代替値を割り当てすることもできます。
- LEAD(field[, offset[, default]]) – 指定したウィンドウ・フレーム内の指定の行の offset 行後にある field 列の値を割り当てます。offset が指定されていない場合、この関数は、既定で、指定された行の 1 行後の field 列の値を割り当てます。既定では、ウィンドウ・フレーム内で、指定された行の offset 行後に行がない場合、LEAD() は値 NULL を割り当てます。ユーザは、値 default を含めることにより、これらの条件下で、代替値を割り当てすることもできます。
- MAX(field) – 指定したウィンドウ・フレーム内の field 列の最大値を、そのウィンドウ・フレームのすべての行に割り当てます。例えば、`MAX(Salary) OVER (PARTITION BY Department) FROM Company.Employee` を使用して、各従業員の給与を、その従業員の部門内の従業員の最も高い給与と比較できます。MAX() は [ROWS 節](#) をサポートします。使用方法の詳細は、[MAX\(\) 関数のリファレンス・ページ](#) を参照してください。

- ・ MIN(field) – 指定したウィンドウ・フレーム内の field 列の最小値を、そのウィンドウ・フレームのすべての行に割り当てます。例えば、MIN(Salary) OVER (PARTITION BY Department) FROM Company.Employee を使用して、各従業員の給与を、その従業員の部門内の従業員の最も低い給与と比較できます。MIN() は [ROWS 節](#)をサポートします。使用方法の詳細は、[MIN\(\) 関数のリファレンス・ページ](#)を参照してください。
- ・ NTH\_VALUE(field, n) – 指定したウィンドウ・フレーム内の行番号 n (1 からカウント) の field 列の値を、そのウィンドウ・フレームのすべての行に割り当てます。NTH\_VALUE() は [ROWS 節](#)をサポートします。
- ・ NTILE(num-groups) – 指定したウィンドウ・フレーム内の行を、それぞれの要素の数がほぼ等しい、num-groups 個のグループに分割します。各グループは、1 から始まる番号で識別されます。
- ・ PERCENT\_RANK() – 同じウィンドウ・フレーム内の各行に、0 以上 1 以下の小数としてランキングのパーセンテージを割り当てます。ウィンドウ関数フィールドに同じ値を含む行が複数ある場合は、ランキングのパーセンテージに重複値を含めることができます。
- ・ RANK() – 同じウィンドウ・フレーム内の各行に、1 から始まるランキングを表す整数を割り当てます。ウィンドウ関数フィールドに同じ値を含む行が複数ある場合は、ランキングの整数に重複値を含めることができます。
- ・ ROW\_NUMBER() – 同じウィンドウ・フレーム内の各行に、一意の連続した整数を割り当てます。ウィンドウ関数フィールドの複数の行に同じ値が含まれている場合は、各行に一意の連続した整数が割り当てられます。
- ・ SUM(field) – 指定したウィンドウ・フレーム内の field 列の値の合計を、そのウィンドウ・フレームのすべての行に割り当てます。SUM() は [ROWS 節](#)をサポートします。使用方法の詳細は、[SUM\(\) 関数のリファレンス・ページ](#)を参照してください。

以下の例では、これらのウィンドウ関数内の ORDER BY 節により返された値を比較します。

```
SELECT Name,City,ROW_NUMBER() OVER (ORDER BY City) AS RowNum,
       RANK() OVER (ORDER BY City) AS RankNum,
       PERCENT_RANK() OVER (ORDER BY City) AS RankPct
FROM CityTable ORDER BY City
```

この例では、すべての行を単一のパーティションとして扱います。City の値で行を並べ替え、以下を返します。

Name	City	RowNum	RankNum	RankPct
Harriet		1	1	0
Betty	Boston	2	2	.111111111111111111
Davis	Boston	3	2	.111111111111111111
George	London	4	4	.333333333333333333
Able	New York	5	5	.444444444444444444
Charlie	Paris	6	6	.555555555555555555
Eve	Paris	7	6	.555555555555555555
Francis	Paris	8	6	.555555555555555555
Beatrix	Paris	9	6	.555555555555555555
Jackson	Rome	10	10	1

## ROWS 節

ROW 節は、AVG()、FIRST\_VALUE()、LAST\_VALUE()、NTH\_VALUE()、MIN()、MAX()、および SUM() ウィンドウ関数で使用できます。他のウィンドウ関数に対して指定することはできませんが、機能しません（結果は ROWS 節があってもなくても同じです）。

ROWS 節には次の 2 つの構文形式があります。

ROWS *framestart* ROWS BETWEEN *framestart* AND *frameend*

*framestart* および *frameend* は、次の 5 つの値を取ることが可能です。

```
UNBOUNDED PRECEDING           /* start at beginning of the current partition */ offset PRECEDING
/* start offset number of rows preceding the current row */ CURRENT ROW      /*
start at the current row */ offset FOLLOWING /* continue offset number of rows following the current
row */ UNBOUNDED FOLLOWING           /* continue to the end of the current partition */
```

ROWS 節の構文は、いずれかの方向の範囲を指定できます。例えば、ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING と ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED PRECEDING はまったく同じです。

ROWS *framestart* 構文は、指定されていない 2 番目の範囲の境界として、既定で CURRENT ROW が設定されます。したがって、以下は同等です。

ROWS UNBOUNDED PRECEDING	ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
ROWS 1 PRECEDING	ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
ROWS CURRENT ROW	ROWS BETWEEN CURRENT ROW AND CURRENT ROW
ROWS 1 FOLLOWING	ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING
ROWS UNBOUNDED FOLLOWING	ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

ROWS 節が指定されていない場合の既定値は、ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW です。

## ROWS 節の例

次のクエリは、多くの “ノイズ” (ランダム変数) を含むスコアを返します。ROWS 節を使用して、現在のスコアと照合順の直前のスコアおよび直後のスコアとを合計し、3 で割ることでこれらの変動を “平滑化” し、ゆるやかな変動の平均スコアを取得します。

```
SELECT Item,Score,SUM(Score)
  OVER (ORDER BY Score ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)/3
  AS CohortScore FROM Sample.TestScores
```

演算は (PrecedingScore + CurrentScore + FollowingScore)/3 のようになります。CohortScore の一番下と一番上の値は、2 つの Score 値に 0 を加えて 3 で割る ((0 + CurrentScore + FollowingScore)/3 および (PrecedingScore + CurrentScore + 0)/3) ため、正確ではありません。

## ウィンドウ関数の使用

ウィンドウ関数は、以下のような箇所で使用できます。

- ・ リストされた select-item としての [SELECT リスト](#)。

ウィンドウ関数は、select-item リスト内のサブクエリまたは集約関数に組み込むことはできません。

- ・ [ORDER BY 節](#)。

ウィンドウ関数は、ON、WHERE、GROUP BY、または HAVING 節の中では使用できません。これを実行しようすると、SQLCODE -367 エラーが返されます。

## 列の名前とエイリアス

既定では、ウィンドウ関数の結果に割り当てられる列名は Window\_n となります。ここで、n という数字接尾語は、SELECT リストで指定されている列順序番号です。したがって、以下の例では、列名 Window\_3 および Window\_6 が作成されます。

```
SELECT Name,State,ROW_NUMBER() OVER (PARTITION BY State),Age,AVG(Age),ROW_NUMBER() OVER (ORDER BY Age)
FROM Sample.Person
```

別の列名 (列のエイリアス) を指定するには、AS キーワードを使用します。

```
SELECT Name,State,ROW_NUMBER() OVER (PARTITION BY State) AS StateRow,Age
FROM Sample.Person
```

列エイリアスを使用して、[ORDER BY](#) 節でウィンドウ・フィールドを指定できます。

```
SELECT Name,State,ROW_NUMBER() OVER (PARTITION BY State) AS StateRow,Age
FROM Sample.Person
ORDER BY StateRow
```

ORDER BY 節で既定の列名 (Window\_3 など) を使用することはできません。

[列エイリアス](#)の詳細は、SELECT 文を参照してください。

## ORDER BY の使用

ORDER BY 節は、ウィンドウ関数が評価された後にクエリの結果セットに適用されるため、ORDER BY は select-item ウィンドウ関数によって割り当てられた値に影響を与えません。

## 関連項目

- ・ [SELECT 文](#)
- ・ [ORDER BY クエリ節](#)
- ・ [集約関数の概要](#)

## AVG (SQL)

---

指定したウィンドウ・フレーム内の行の field 列の値の平均を、そのウィンドウ・フレームのすべての行に割り当てるウィンドウ関数。

### 構文

```
AVG(field)
```

### 説明

AVG は、指定したウィンドウ・フレーム内の行の指定した列の値の平均を、そのウィンドウ・フレームのすべての行に返します。AVG は [ROWS 節](#)をサポートします。

このウィンドウ関数は、集約関数 [AVG](#) と同様に機能します。

### 引数

#### field

平均が計算される値の行を指定する列。

### 例

以下の例は、各従業員の給与を、その従業員の部門内の従業員給与の平均と比較しています。

#### SQL

```
SELECT AVG(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)
- ・ 集約関数：[AVG](#)



# COUNT (SQL)

指定したウィンドウ・フレーム内の各行に 1 から始まる数を割り当てるウィンドウ関数。

## 構文

```
COUNT(*|field)
```

## 説明

COUNT は、各行 (field 引数が指定されている場合はその特定の行) に 1 から始まるカウンタを作成します。

このウィンドウ関数は、集約関数 [COUNT](#) と同様に機能します。

## 引数

### field

カウントをインクリメントするフィールドを指定します。field が指定されている場合、カウントは field のコンテンツが NULL でないときにのみインクリメントされます。この引数が指定されていない場合、カウントはすべての行でインクリメントされます。

## 例

以下の例では、各従業員を個別の行として残したまま、各部門内の従業員数をカウントします。

### SQL

```
SELECT COUNT(Employee) OVER (PARTITION BY Department) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)
- ・ 集約関数：[COUNT](#)

## CUME\_DIST() (SQL)

---

指定したウィンドウ・フレーム内のすべての行に累積分布値を割り当てるウィンドウ関数。

### 構文

```
CUME_DIST()
```

### 説明

CUME\_DIST() は、すべての行に累積分布を割り当てます。累積分布は、確率変数が特定の値以下になる割合を表します。この分布は、現在の行の値以下の値を持つ行をカウントし、そのカウントをウィンドウ内の行の総数で除算することによって計算されます。累積分布が計算される列の名前は、**ORDER BY** 節で指定されます。

### 例

以下の例では、各部門内の各従業員の給与の累積分布を返します。

### SQL

```
SELECT CUME_DIST() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)

# DENSE\_RANK() (SQL)

同じウィンドウ・フレーム内の各行に 1 から始まるランクを割り当てるウィンドウ関数。

## 構文

```
DENSE_RANK()
```

## 説明

DENSE\_RANK() は、各行にランク (整数) を割り当てます。この整数は、([RANK\(\)](#) ウィンドウ関数とは異なり) 常に連続する整数となります。これらのランクは、ORDER BY 式で指定された値によって決まります。例えば、同じ値を持つすべての行には同じランクが与えられます。ただし、RANK() とは異なり、DENSE\_RANK() は複数の行が同じランクを共有していても連続する番号をスキップすることはありません。2 つの行がどちらもランク 1 の場合、割り当てられる次のランクは 2 となります。

DENSE\_RANK() では、ウィンドウ関数フィールドに同じ値を含む行が複数ある場合、重複値も許容されます。

## 例

以下の例では、各部門内での給与に基づいて、従業員のランクが返されます。

## SQL

```
SELECT DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)

## FIRST\_VALUE (SQL)

---

ウィンドウ・フレーム内の field 列の最初の値を、その列の他の各値に割り当てるウィンドウ関数。

### 構文

```
FIRST_VALUE(field)
```

### 説明

FIRST\_VALUE は、最初の行の field 列の値を使用して、特定のウィンドウ・フレームのすべての行に割り当てます。FIRST\_VALUE は [ROWS](#) 節をサポートします。

すべての値の前に NULL を照合し、最初の行の field の値が NULL の場合にウィンドウ内のすべての行が NULL になるようにすることに注意してください。

### 引数

#### field

そのウィンドウ・フレームのすべての行に割り当てる値を指定する列。

### 例

以下の例は、各都市の Country 列の最初の値を返します。

#### SQL

```
SELECT FIRST_VALUE(Country) OVER (PARTITION BY City)
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)

# LAG (SQL)

指定したウィンドウ・フレーム内の指定の行の offset 行前にある field 列の値を割り当てるウィンドウ関数。

## 構文

```
LAG(field[,offset[,default]])
```

## 説明

LAG は、offset で指定した行の field 列の値を割り当てます。既定では、ウィンドウ・フレーム内で、指定された行の offset 行前に行がない場合、LAG() は値 NULL を割り当てます。

ユーザは、値 default を含めることにより、これらの条件下で、代替値を割り当てることもできます。

## 引数

### field

割り当てられる値を指定する列。

### offset

field 列の値を取得する行を指定する整数 (オプション)。offset が指定されていない場合、この関数は、既定で、指定された行の 1 行前の field 列の値を割り当てます。

### default

ウィンドウ・フレーム内で指定された行の offset 行前に行がない場合に、返す値を指定する引数 (オプション)。default が指定されていない場合、値 NULL が割り当てられます。

## 例

以下の例では、各部門内の各従業員の以前の給与を返します。

### SQL

```
SELECT LAG(Salary) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

以下の例では、offset および default 引数を指定し、各部門の以前の従業員の給与を計算します。そのような従業員が存在しない場合は、値 0 が返されます。

### SQL

```
SELECT LAG(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)

## LAST\_VALUE (SQL)

---

ウィンドウ・フレーム内の field 列の最後の値を、その列の他の各値に割り当てるウィンドウ関数。

### 構文

```
LAST_VALUE(field)
```

### 説明

LAST\_VALUE は、最後の行の field 列の値を使用して、特定のウィンドウ・フレームのすべての行に割り当てます。LAST\_VALUE は [ROWS 節](#)をサポートします。

すべての値の前に NULL を照合し、最後の行の field の値が NULL の場合にウィンドウ内のすべての行が NULL になるようにすることに注意してください。

### 引数

#### field

そのウィンドウ・フレームのすべての行に割り当てる値を指定する列。

### 例

以下の例は、各都市の Country 列の最後の値を返します。

#### SQL

```
SELECT LAST_VALUE(Country) OVER (PARTITION BY City)
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)

# LEAD (SQL)

指定したウィンドウ・フレーム内の指定の行の offset 行後にある field 列の値を割り当てるウィンドウ関数。

## 構文

```
LEAD(field[, offset[, default]])
```

## 説明

LEAD は、offset で指定した行の field 列の値を割り当てます。既定では、ウィンドウ・フレーム内で、指定された行の offset 行後に行がない場合、LEAD() は値 NULL を割り当てます。

ユーザは、値 default を含めることにより、これらの条件下で、代替値を割り当てることもできます。

## 引数

### field

割り当てられる値を指定する列。

### offset

field 列の値を取得する行を指定する整数 (オプション)。offset が指定されていない場合、この関数は、既定で、指定された行の 1 行後の field 列の値を割り当てます。

### default

ウィンドウ・フレーム内で指定された行の offset 行後に行がない場合に、返す値を指定する引数 (オプション)。default が指定されていない場合、値 NULL が割り当てられます。

## 例

以下の例では、各部門内の各従業員の以降の給与を返します。

### SQL

```
SELECT LEAD(Salary) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

以下の例では、offset および default 引数を指定し、各部門の次の従業員の給与を計算します。そのような従業員が存在しない場合は、値 0 が返されます。

### SQL

```
SELECT LEAD(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)



## MAX (SQL)

---

指定したウィンドウ・フレーム内の field 列の最大値を、そのウィンドウ・フレームのすべての行に割り当てるウィンドウ関数。

### 構文

```
MAX(field)
```

### 説明

MAX は、field 列で指定される最大値を、そのウィンドウ・フレーム内のすべての行に割り当てます。MAX は [ROWS 節](#) をサポートします。

### 引数

field

どこから最大値を取得するのかを指定する列。

### 例

以下の例は、各従業員の給与を、その部門内の従業員の最も高い給与と比較しています。

#### SQL

```
SELECT MAX(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)
- ・ 集約関数 : [MAX](#)

## MIN (SQL)

---

指定したウィンドウ・フレーム内の field 列の最小値を、そのウィンドウ・フレームのすべての行に割り当てるウィンドウ関数。

### 構文

```
MIN(field)
```

### 説明

MIN は、field 列で指定される最小値を、そのウィンドウ・フレーム内のすべての行に割り当てます。MIN は [ROWS 節](#) をサポートします。

### 引数

**field**

どこから最小値を取得するかを指定する列。

### 例

以下の例は、各従業員の給与を、その部門内の従業員の最も低い給与と比較しています。

#### SQL

```
SELECT MIN(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)
- ・ 集約関数 : [MIN](#)

## NTH\_VALUE (SQL)

---

指定したウィンドウ・フレーム内の行番号 *n* の *field* 列の値を、そのウィンドウ・フレームのすべての行に割り当てるウィンドウ関数。

### 構文

```
NTH_VALUE(field, n)
```

### 説明

NTH\_VALUE は、行番号 *n* から取得した *field* 列の値を、ウィンドウ・フレーム内のすべての行に割り当てます。NTH\_VALUE は [ROWS 節](#)をサポートします。

### 引数

#### field

すべての行に割り当てる値を指定する列。

#### *n*

どの行の *field* 列値を使用するかを示す数値。

### 例

以下の例では、各部門内で 2 番目に高い給与を返します。

#### SQL

```
SELECT NTH_VALUE(Salary, 2) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)

# NTILE (SQL)

指定したウィンドウ・フレーム内の行を、それぞれの要素の数がほぼ等しい、num-groups 個のグループに分割するウィンドウ関数。

## 構文

`NTILE (num-groups)`

## 説明

NTILE は、ウィンドウ・フレーム内の行を、それぞれに含まれる要素の数がほぼ等しくなるように、num-groups 個のグループに分割します。各グループは、1 から始まる番号で識別されます。

## 引数

### num-groups

行をいくつのグループに分割するかを指定する数値。

## 例

以下の例では、各部門の従業員を、その給与に基づいて、4 つのグループに分割します。

### SQL

```
SELECT NTILE(4) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)

## PERCENT\_RANK() (SQL)

---

同じウィンドウ・フレーム内の各行に、0 以上 1 以下の小数としてランキングを割り当てるウィンドウ関数。

### 構文

```
PERCENT_RANK()
```

### 説明

PERCENT\_RANK は、各行に 0 以上 1 以下の百分位数ランキングを割り当てます。百分位数ランキングには、グループ内の特定の値以下のデータのパーセンテージが含まれます。ウィンドウ関数フィールドに同じ値を含む行が複数ある場合は、これらのランクに重複値を含めることができます。

### 例

以下の例では、各部門内での給与に基づいて、各従業員の百分位数ランクが計算されます。

### SQL

```
SELECT PERCENT_RANK() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)

# RANK() (SQL)

同じウィンドウ・フレーム内の各行に 1 から始まるランクを割り当てるウィンドウ関数。

## 構文

RANK ( )

## 説明

RANK は、各行に 1 から始まるランク (整数) を割り当てます。これらのランクは、ORDER BY 式で指定された値によって決まります。例えば、同じ値を持つすべての行には同じランクが与えられます。ただし、DENSE\_RANK() とは異なり、RANK() は複数の行が同じランクを共有する場合、連続する番号をスキップします。2 つの行がどちらもランク 1 の場合、割り当てられる次のランクは 3 となります。

ウィンドウ関数フィールドに同じ値を含む行が複数ある場合は、これらのランクに重複値を含めることができます。

## 例

以下の例では、各部門内での給与に基づいて、各従業員にランクが割り当てられます。

## SQL

```
SELECT RANK() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)

## ROW\_NUMBER() (SQL)

---

同じウィンドウ・フレーム内の各行に 1 から始まる一意の連続した整数を割り当てるウィンドウ関数。

### 構文

```
ROW_NUMBER()
```

### 説明

ROW\_NUMBER は、各行に 1 から始まる一意の連続した整数を割り当てます。ウィンドウ関数フィールドの複数の行に同じ値が含まれている場合は、各行に一意の連続した整数が割り当てられます。

### 例

以下の例では、(各部門内の) 各従業員に、その給与に基づいて、順番に番号が割り当てられます。

### SQL

```
SELECT ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

### 関連項目

- ・ [ウィンドウ関数の概要](#)



---

# SUM (SQL)

---

指定したウィンドウ・フレーム内の field 列の値の合計を、そのウィンドウ・フレームのすべての行に割り当てるウィンドウ関数。

## 構文

```
SUM(field)
```

## 説明

SUM は、フレーム内の field 列の値を合計し、そのフレームのすべての行に割り当てます。SUM は [ROWS 節](#)をサポートします。

このウィンドウ関数は、集約関数 [SUM](#) と同様に機能します。

## 引数

### field

合計される値を指定する列。

## 例

以下の例では、各部門の給与の合計を計算します。

### SQL

```
SELECT SUM(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

## 関連項目

- ・ [ウィンドウ関数の概要](#)
- ・ 集約関数：[SUM](#)



# SQL 関数

## ABS (SQL)

数値式の絶対値を返す、数値関数です。

### 構文

```
ABS(numeric-expression)  
{fn ABS(numeric-expression)}
```

### 概要

ABS は絶対値を返します。絶対値は、必ず 0 または正の数になります。*numeric-expression* が数値でない場合（例えば文字列 'abc' や空文字列 ''）、ABS は 0 を返します。NULL 値を渡すと ABS は <null> を返します。

ABS は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として使用できる点に注意してください。

この関数は、ObjectScript から ABS() メソッド・コールを使用して呼び出すこともできます。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.ABS(-0099)
```

### 引数

#### *numeric-expression*

絶対値を求める数。

ABS が返す値のデータ型は、*numeric-expression* のデータ型と同じです。

### 例

以下の例は、ABS の 2 つの形式を示しています。

#### SQL

```
SELECT ABS(-99) AS AbsGen, {fn ABS(-99)} AS AbsODBC
```

いずれも 99 を返します。

以下の例は、いくつかの数値が ABS でどのように扱われるかを示しています。InterSystems SQL は、*numeric-expression* を**キャノン形式**に変換し、先頭のゼロと末尾のゼロを削除し、指数を評価してから、ABS を呼び出します。

#### SQL

```
SELECT ABS(007) AS AbsoluteValue
```

これは、7 を返します。

#### SQL

```
SELECT ABS(-0.000) AS AbsoluteValue
```

これは、0 を返します。

#### SQL

```
SELECT ABS(-99E4) AS AbsoluteValue
```

これは、990000 を返します。

## SQL

```
SELECT ABS(-99E-4) AS AbsoluteValue
```

これは、.0099 を返します。

## 関連項目

- ・ SQL 関数 : [CONVERT TO\\_NUMBER](#)
- ・ ObjectScript 関数 : [\\$ZABS](#)

## ACOS (SQL)

指定されたコサインのアークコサインを、ラジアン表示で返すスカラ数値関数です。

### 構文

```
{fn ACOS(numeric-expression)}
```

### 概要

ACOS は数値を取り、そのコサインの逆関数値を浮動小数点数値で返します。numeric-expression には、-1 以上 1 以下の符号付き 10 進数値を指定します。この範囲外の数値を指定すると、実行時エラーが発生して、SQLCODE -400 (致命的なエラーが発生しました) が生成されます。NULL 値を渡すと、ACOS は NULL を返します。ACOS は、空文字列 (") のように数値でない文字列を数値 0 (ゼロ) として扱います。

ACOS は、有効桁数が 19 で小数桁数が 18 の値を返します。

ACOS は、{} 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

### 引数

#### numeric-expression

値が -1 から 1 の間である数値式。角度のコサインです。

ACOS は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。ACOS は、numeric-expression がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

### 例

以下の例は、2 つのコサイン値に対する ACOS の実行結果です。

#### SQL

```
SELECT {fn ACOS(0.52)} AS ArcCosine
```

これは、1.023945... を返します。

#### SQL

```
SELECT {fn ACOS(-1)} AS ArcCosine
```

これは、円周率 (3.14159...) を返します。

### 関連項目

- SQL 関数: [ASIN](#)、[ATAN](#)、[COS](#)、[COT](#)、[SIN](#)、[TAN](#)
- ObjectScript 関数: [\\$ZARCCOS](#)

# ASCII (SQL)

文字列式の最初の文字 (左端にある文字) を整数 ASCII コード値として返す、文字列関数です。

## 構文

```
ASCII(string-expression)  
{fn ASCII(string-expression)}
```

## 概要

NULL または空文字列値を渡すと、ASCII は NULL を返します。NULL を空文字列に返すことは SQL サーバとの整合性が取れています。

ASCII は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として呼び出せる点に注意してください。

## 引数

### *string-expression*

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。CHAR タイプや VARCHAR タイプの文字列式です。

## 例

以下の例はどちらも、文字 Z の ASCII 値である 90 を返します。

### SQL

```
SELECT ASCII('Z') AS AsciiCode
```

### SQL

```
SELECT {fn ASCII('ZEBRA')} AS AsciiCode
```

InterSystems SQL は数値を**キャノニック形式**に変換してから、ASCII 変換を実行します。以下の例は、数値 7 の ASCII 値である 55 を返します。

### SQL

```
SELECT ASCII(+007) AS AsciiCode
```

数値を文字列として指定すると、この数値の解析は実行されません。以下の例は、プラス (+) 文字の ASCII 値である 43 を返します。

### SQL

```
SELECT ASCII('+007') AS AsciiCode
```

## 関連項目

- SQL 関数: [CHAR](#)
- ObjectScript 関数: [\\$ASCII](#) [\\$ZLASCII](#) [\\$ZWASCII](#)



## ASIN (SQL)

与えられた角度の正弦のアークサインを、ラジアン表示で返すスカラ数値関数です。

### 構文

```
{fn ASIN(numeric-expression)}
```

### 概要

ASIN は、角度のサインの逆関数値を浮動小数点値で返します。numeric-expression には、-1 以上 1 以下の符号付き 10 進数値を指定します。この範囲外の数値を指定すると、実行時エラーが発生して、SQLCODE -400 (致命的なエラーが発生しました) が生成されます。NULL 値を渡すと、ASIN は NULL を返します。ASIN は、空文字列 (") のように数値でない文字列を数値 0 (ゼロ) として扱います。

ASIN は、有効桁数が 19 で小数桁数が 18 の値を返します。

ASIN は、{} 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

### 引数

#### numeric-expression

値が -1 から 1 の間である数値式。角度のサインです。

ASIN は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。ASIN は、numeric-expression がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

### 例

以下の例は、2 つのサイン値に対する ASIN の実行結果です。

#### SQL

```
SELECT {fn ASIN(0.52)} AS ArcSine
```

これは、0.5468509506... を返します。

#### SQL

```
SELECT {fn ASIN(-1.00)} AS ArcSine
```

これは、-1.5707963267... を返します。

### 関連項目

- SQL 関数: [ACOS](#)、[ATAN](#)、[COS](#)、[COT](#)、[SIN](#)、[TAN](#)
- ObjectScript 関数: [\\$ZARCSIN](#)

# ATAN (SQL)

与えられた角度の三角関数のアーク・タンジェントを、ラジアン表示で返すスカラ数値関数です。

## 構文

```
{fn ATAN(numeric-expression)}
```

## 概要

ATAN は任意の数値を取り、角度のタンジェントの逆関数値を浮動小数点値で返します。NULL 値を渡すと、ATAN は NULL を返します。ATAN は、空文字列 (") のように数値でない文字列を数値 0 (ゼロ) として扱います。

ATAN は、有効桁数が 36 で小数桁数が 18 の値を返します。

ATAN は、{ } 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

## 引数

### *numeric-expression*

数値式。角度のタンジェントです。

ATAN は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。ATAN は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、ATAN の実行結果です。

### SQL

```
SELECT {fn ATAN(0.52)} AS ArcTangent
```

これは、0.47951929199... を返します。

## 関連項目

- SQL 関数: [ACOS](#)、[ASIN](#)、[COS](#)、[COT](#)、[SIN](#)、[TAN](#)
- ObjectScript 関数: [\\$ZARCTAN](#)

## ATAN2 (SQL)

2 つの座標を受け取りアーク・タンジェントの角度をラジアンで返すスカラ数値関数。

### 構文

```
{fn ATAN2(y,x)}
```

### 概要

ATAN2 は半径のデカルト座標 (y,x) を受け取り、角度のタンジェントの逆関数値 (弧) を浮動小数点値として返します。両方の座標の符号を使用して、デカルト座標が決定されます。x が正の値である場合、ATAN2 は [ATAN\(y/x\)](#) と同じ値を返します。NULL 値を渡すと ATAN2 は NULL を返します。ATAN2 は、空文字列 (" ) のように数値でない文字列を数値 0 (ゼロ) として扱います。

ATAN2 は、有効桁数が 36 で小数桁数が 18 の値を返します。

ATAN2 は {} 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

### 引数

#### y

y 軸座標を指定する数値式。

#### x

x 軸座標を指定する数値式。

ATAN2 は、NUMERIC または DOUBLE [データ型](#) のいずれかを返します。ATAN2 では、numeric-expression がデータ型 DOUBLE の場合は DOUBLE を返し、それ以外の場合は NUMERIC を返します。

### 例

次の例では、ATAN2 を呼び出します。

```
SELECT {fn ATAN2(15,30)} AS ArcTangent
```

これは、0.46 を返します。

### 関連項目

- SQL 関数 : [ACOS](#)、[ASIN](#)、[ATAN](#)、[COS](#)、[COT](#)、[SIN](#)、[TAN](#)
- ObjectScript 関数 : [\\$ZARCTAN](#)

# CAST (SQL)

与えられた式を、特定のデータ型に変換する関数です。

## 構文

### 文字列

```
CAST(expression AS [ CHAR | CHARACTER | VARCHAR | NCHAR | NVARCHAR ])
```

```
CAST(expression AS [ CHAR VARYING | CHARACTER VARYING ])
```

```
CAST(expression AS
  [ CHAR(length) | CHARACTER(length) |
    VARCHAR(length) | CHAR VARYING(length) |
    CHARACTER VARYING(length) ])
```

### 数値

```
CAST(expression AS [ INT | INTEGER | BIGINT | SMALLINT | TINYINT ])
```

```
CAST(expression AS [ DEC | DECIMAL | NUMERIC ])
```

```
CAST(expression AS
  [ DEC(precision,scale) |
    DECIMAL(precision,scale) |
    NUMERIC(precision,scale) ])
```

```
CAST(expression AS DOUBLE)
```

```
CAST(expression AS [ MONEY | SMALLMONEY ])
```

### 日付と時刻

```
CAST(expression AS DATE)
```

```
CAST(expression AS TIME)
```

```
CAST(expression AS [ TIMESTAMP | DATETIME | SMALLDATETIME ])
```

```
CAST(expression AS POSIXTIME)
```

### ビット値

```
CAST(expression AS BIT)
```

### バイナリ値

```
CAST(expression AS [ BINARY | BINARY VARYING | VARBINARY ])
```

```
CAST(expression AS [ BINARY(length) |
  BINARY VARYING(length) |
  VARBINARY(length) ])
```

### 一意の識別子

```
CAST(expression AS GUID)
```

## 概要

SQL の CAST 関数は、式のデータ型を指定したデータ型に変換します。InterSystems SQL でサポートされているデータ型の一覧については、“[データ型](#)”を参照してください。

CAST は [CONVERT](#) と似ていますが、以下の相違点があります。

- ・ CONVERT のほうが CAST よりも柔軟です。例えば、CONVERT はストリーム・データの変換をサポートし、日付値と時刻値の書式を設定できます。
- ・ CAST のほうが CONVERT よりもデータベースでの互換性に優れています。CAST が ANSI SQL-92 標準を使用して実装されるのに対し、CONVERT の実装はデータベース固有です。InterSystems SQL が提供する CONVERT の実装には、MS SQL Server および ODBC との互換性があります。

サポートされていないデータ型で CAST を指定すると、InterSystems IRIS® によって SQLCODE -376 が生成されます。

## 文字列

- ・ CAST(expression AS [CHAR | CHARACTER | VARCHAR | NCHAR | NVARCHAR]) 数値または文字列の式を文字列データ型に変換します。これらのデータ型はすべて **%Library.String** にマッピングされます。
  - CHAR、CHARACTER、NCHAR は同等のデータ型であり、既定の長さは 1 文字です。
  - VARCHAR と NVARCHAR は同等のデータ型であり、既定の長さは 30 文字です。

以下の文からは、VARCHAR のデータ型として Name (文字列)、Age (数値)、DOB (日付値) が返されます。

### SQL

```
SELECT DISTINCT
  CAST(Name AS VARCHAR) AS VarCharName,
  CAST(Age AS VARCHAR) AS VarCharAge,
  CAST(DOB AS VARCHAR) AS VarCharDOB
FROM Sample.Person
```

この文は文字列値を単一の文字列にキャストし、元の文字列にあるその他の文字を切り捨てます。

### SQL

```
SELECT CAST('True' AS CHAR) -- T
```

例：文字列値のキャスト

- ・ CAST(expression AS [CHAR VARYING | CHARACTER VARYING]) は式を変換し、元の値にある文字と同じ個数の文字を返します。

この文は、浮動小数点で表現した  $\pi$  値を文字列値として返します。この文字列には、 $\pi$  の浮動小数点精度の桁数と同じ個数の文字があります。

### SQL

```
SELECT CAST({fn PI()} AS CHAR VARYING) AS StringPi -- 3.141592653589793238
```

例：文字列値のキャスト

- ・ CAST(expression AS [CHAR(length) | CHARACTER(length) | VARCHAR(length) | CHAR VARYING(length) | CHARACTER VARYING(length)]) は length で指定された文字数の文字列に式を変換します。余分な文字は切り捨てられます。

以下の文は、入力文字列式の先頭 8 文字で構成した文字列を返します。

### SQL

```
SELECT CAST('Grabscheid,Alfred N.' AS CHAR(8)) -- Grabsche
```

例：文字列値のキャスト

## 数値

- CAST(expression AS [INT | INTEGER | BIGINT | SMALLINT | TINYINT]) INT、INTEGER、BIGINT、SMALLINT、または TINYINT のデータ型に式を変換します。これらのデータ型では、小数部が切り捨てられます。

以下の例では、浮動小数点ではなく、整数で平均値が求められます。CAST によって数値が切り捨てられるので、平均年齢 42.9 は 42 になります。

### SQL

```
SELECT DISTINCT AVG(Age) AS AvgAge,
    CAST(AVG(Age) AS INTEGER) AS IntAvgAge
FROM Sample.Person
```

#### 例：数値のキャスト

- CAST(expression AS [DEC | DECIMAL | NUMERIC]) は DEC、DECIMAL、または NUMERIC のデータ型に式を変換します。これらのデータ型では、元の値の桁数が保持されます。InterSystems IRIS では [\\$DECIMAL](#) 関数を使用してこれらのデータ型が変換されます。この関数は \$DOUBLE 値を \$DECIMAL 値に変換します。これらのデータ型は [%Library.Numeric](#) データ型にマッピングされます。

以下の文は、1 ラジアン の正弦値である浮動小数点値を小数として返します。

### SQL

```
SELECT CAST({fn SIN(1)} AS DECIMAL) AS DecimalValue -- 0.841470984807897
```

#### 例：数値のキャスト

- CAST(expression AS [DEC(precision,scale) | DECIMAL(precision,scale) | NUMERIC(precision,scale)]) はデータ型の精度と記数法を指定します。
  - precision は、データ型で指定できる桁数の合計値を指定します。precision を指定すると、CAST から返される値に影響はありませんが、定義した型の一部として保持されます。
  - scale データ型の小数点以下桁数を指定します。CAST では、この指定した桁数で数値が四捨五入されます。

以下の文は、1 ラジアン の正弦値を小数点以下 4 桁の小数値として返します。

### SQL

```
SELECT CAST({fn SIN(1)} AS DECIMAL(8,4)) AS ScaledDecimalValue -- 0.8415
```

#### 例：数値のキャスト

- CAST(expression AS DOUBLE) は、IEEE 浮動小数点標準に準拠する DOUBLE データ型に式を変換します。詳細は、ObjectScript の ["\\$DOUBLE"](#) 関数を参照してください。

以下の文は、倍精度浮動小数点値として 1 ラジアン の正弦値を返します。

### SQL

```
SELECT CAST({fn SIN(1)} AS DOUBLE) AS DoubleValue -- .84147098480789650487
```

#### 例：数値のキャスト

- CAST(expression AS [MONEY | SMALLMONEY]) は、通貨の数値データ型である MONEY または SMALLMONEY に式を変換します。通貨データ型の小数点以下桁数は必ず 4 です。

以下の文は、通貨値として整数 10 を返します。

```
SELECT CAST(10 AS MONEY) AS MoneyValue -- 10.0000 (Display Mode)
```

## 日付と時刻

- CAST([expression](#) AS DATE) は、書式設定された日付式を DATE データ型に変換します。InterSystems IRIS では、コンテキストに応じて以下の形式で日付が表現されます。
  - ローケールに合わせた表示日付形式 (例 : mm/dd/yyyy)
  - ODBC 日付形式 (yyyy-mm-dd)
  - \$HOROLOG 整数日付ストレージ形式 (nnnnn)

\$HOROLOG の日付部分の値は、数値文字列ではなく、整数として指定する必要があります。

以下の文は、文字列を DATE データ型にキャストします。

### SQL

```
SELECT CAST('1936-11-26' AS DATE) AS StringToDate
```

例 : [フォーマットされた文字列の日付へのキャスト](#)

- CAST([expression](#) AS TIME) は、書式設定された時刻式を TIME データ型に変換します。InterSystems IRIS では、コンテキストに応じて以下の形式で時刻が表現されます。
  - ローケールに合わせた表示時刻形式 (例 : mm/dd/yyyy)
  - ODBC 日付形式 (hh:mm:ss)
  - \$HOROLOG 整数時刻ストレージ形式 (nnnnn)

\$HOROLOG の日付部分の値は、数値文字列ではなく、整数として指定する必要があります。

以下の文は、文字列を TIME データ型にキャストします。

### SQL

```
SELECT CAST('14:33:45.78' AS TIME) AS StringToTime
```

例 : [フォーマットされた文字列の時刻へのキャスト](#)

- CAST([expression](#) AS [TIMESTAMP | DATETIME | SMALLDATETIME]) は、日付とタイムスタンプを YYYY-MM-DD hh:mm:ss.nnn 形式で表現します。この値は、ObjectScript の [\\$ZTIMESTAMP](#) 特殊変数に対応します。

以下の文は、日付と時刻の文字列を TIMESTAMP データ型にキャストします。

### SQL

```
SELECT CAST('November 26, 1936 14:33:45.78' AS TIMESTAMP) AS StringToTS
```

例 : [フォーマットされた文字列のタイムスタンプへのキャスト](#)

- CAST([expression](#) AS POSIXTIME) は、日付とタイムスタンプを表現する式を、エンコードした 64 ビットの符号付き整数に変換します。このエンコード形式の詳細は、“[日付、時刻、PosixTime、およびタイムスタンプのデータ型](#)”を参照してください。

以下の文は、日付と時刻の文字列を POSIXTIME データ型にキャストします。

```
SELECT CAST('November 26, 1936 14:33:45.78' AS POSIXTIME) AS StringToPosix
```

例 : [POSIXTIME への日付のキャスト](#)



## ビット値

- CAST([expression](#) AS BIT) は、BIT データ型の単一のブーリアン値に式を変換します。  
以下の文は、それぞれ 1 と 0 の BIT 値を返します。

### SQL

```
SELECT
    CAST('1' AS BIT) As BitTrue,
    CAST('0' AS BIT) As BitFalse
```

例：ビット値のキャスト

## バイナリ値

- CAST([expression](#) AS [BINARY | BINARY VARYING | VARBINARY]) は、**%Library.Binary** (SQLType データ型 BINARY) にマッピングされる 3 つのデータ型のいずれかに式を変換します。
  - BINARY の既定の長さは 1 です。
  - BINARY VARYING および VARBINARY の既定の長さは 30 です。

CAST では、バイナリ値へのキャストでデータが変換されるのではなく、指定した length. の長さに値が切り詰められます。

- CAST([expression](#) AS [BINARY([length](#)) | BINARY VARYING([length](#)) | VARBINARY([length](#))]) は、返すバイナリ・データ型の最大文字長を設定します。

## 一意の識別子

- CAST([expression](#) AS GUID) GUID は、データ型 **%Library.UniqueIdentifier** の 36 文字の値を表します。37 文字以上の [expression](#) を指定した場合、CAST は [expression](#) の最初の 36 文字を返します。GUID 値を生成するには、**%SYSTEM.Util.CreateGUID()** メソッドを使用します。

## 引数

### expression

SQL 式。一般的には、テーブルの中で変換対象となるリテラルまたはデータ・フィールドです。

### length

キャスト後に返す最大文字数を示す整数です。

- length が [expression](#) の長さより短い場合、返されるデータは先頭から length 文字に切り詰められます。
- length が [expression](#) の長さより長い場合、CAST は切り捨てもパディングも実行しません。

### precision

キャストしたデータ型で返される合計桁数の最大値です (整数で指定)。precision は、定義されたデータ型の一部として保持されますが、CAST で返される値には影響しません。

精度の詳細は、“[精度と小数桁数](#)”を参照してください。

### scale

キャストしたデータ型で返される小数点以下桁数の最大値です (整数で指定)。返される値が scale の桁数で四捨五入されます。

- ・ scale = 0 を指定すると数値が整数に四捨五入されます。
- ・ scale = -1 を指定すると数値が整数に切り捨てられます。
- ・ scale を指定しないと、小数点以下桁数は既定値の 15 になります。

キャストした値の桁数よりも scale が大きい場合、表示モードでは適切な個数のゼロが返された値の末尾に付加されますが、論理モードと ODBC モードではこれらの桁は切り捨てられます。

小数点以下桁数の詳細は、“[精度と小数桁数](#)”を参照してください。

## 例

### 文字列値のキャスト

文字列をさまざまな数値、日付、時刻、文字列の値にキャストできます。

#### 文字列間のキャスト

文字列を他の文字データ型にキャストすると、1 文字、先頭の `length` 文字、または文字列全体が返されます。

#### SQL

```
SELECT
  CAST('Hello World' AS CHAR) AS StringToChar, -- H
  CAST('Hello World' AS CHAR(5)) AS StringToCharLength, -- Hello
  CAST('Hello World' AS CHAR VARYING) AS StringToCharVary -- Hello World
```

キャストの前に、InterSystems SQL によって埋め込み引用符文字および文字列連結が解決されます。先頭の空白や末尾の空白は削除されます。

#### SQL

```
SELECT
  CAST('Can't' AS VARCHAR) AS EmbeddedQuote, -- Can't
  CAST('Can' || 'not' AS VARCHAR) AS StringConcatenation -- Cannot
```

### 数値型への文字列のキャスト

文字列を数値型にキャストすると、InterSystems SQL は 1 桁のゼロ (0) を返します。

```
SELECT CAST('Hello World' AS DOUBLE) AS StringToNumeric -- 0
```

以下の例では、CAST 関数を使用して Name (文字列) をさまざまな数値データ型に変換した結果を示します。どの場合でも、返される値は 0 (ゼロ) です。

#### SQL

```
SELECT DISTINCT
  CAST(Name AS INT) AS IntName,
  CAST(Name AS SMALLINT) AS SmallIntName,
  CAST(Name AS DEC) AS DecName,
  CAST(Name AS NUMERIC) AS NumericName
FROM Sample.Person
```

### フォーマットされた文字列の日付へのキャスト

'yyyy-mm-dd' 形式の文字列を DATE データ型にキャストできます。これは、ODBC 日付形式に対応した文字列形式です。InterSystems SQL では、入力した式に対して値と範囲が確認されます。各要素の内容を以下に示します。

- ・ 年 (yyyy) には、00001 以上 9999 以下の数値を指定する必要があります。
- ・ 月 (mm) には、01 以上 12 以下の数値を指定する必要があります。
- ・ 日 (dd) には、該当する月で有効な数値を指定する必要があります。

InterSystems SQL は欠落している先頭のゼロを挿入します。以下に例を示します。

## SQL

```
SELECT CAST('2022-3-1' AS DATE) AS DateValue -- 03/01/2022 (Display Mode)
```

日付が無効な場合は 1840-12-31 (論理日付 0) を返します。例えば、2/29 はうるう年の場合にのみ有効です。

## SQL

```
SELECT CAST('2021-02-29' AS DATE) AS InvalidDate -- 12/31/1840 (Display Mode)
```

表示モードおよびロケールの日付表示形式によってキャストの表示が決まります。例えば、'2004-11-23' が '11/23/2004' と表示されます。

埋め込み SQL は、対応する \$HOROLOG 日付整数としてキャストを返します。無効な ODBC 日付や数値ではない文字列を DATE にキャストすると、論理モードでは 0 とされます。日付 0 は 1840-12-31 として表示されます。

### フォーマットされた文字列の時間へのキャスト

'hh:mm'、'hh:mm:ss'、'hh:mm:ss.nn' (length 秒の小数点以下桁数は任意) 形式の文字列は TIME データ型にキャストできます。これは、ODBC 時刻形式に対応した文字列形式です。InterSystems SQL では、入力した式に対して値と範囲が確認されます。各要素の内容を以下に示します。

- ・ 時間 (hh) には、00 以上 23 以下の数値を指定する必要があります。
- ・ 分 (mm) には、00 以上 59 以下の数値を指定する必要があります。
- ・ 秒 (ss) には、00 以上 60 未満の数値を指定する必要があります。秒に小数部は使用できますが、切り捨てられます。

InterSystems SQL は欠落しているゼロを追加します。以下に例を示します。

## SQL

```
SELECT
  CAST('2:45' AS TIME) AS StringToTime1, -- 02:45:00 (Display Mode)
  CAST('2:45:59' AS TIME) AS StringToTime2, -- 02:45:59 (Display Mode)
  CAST('2:45:59.98' AS TIME) AS StringToTime3 -- 02:45:59.98 (Display Mode)
```

時刻が無効な場合は 00:00:00 を返します (論理時刻 0)。

## SQL

```
SELECT CAST('11:52:60' AS TIME) AS InvalidTime -- 00:00:00 (Display Mode)
```

埋め込み SQL は、対応する \$HOROLOG 時刻整数としてキャストを返します。無効な ODBC 時刻や数値ではない文字列を TIME にキャストすると、論理モードでは 0 とされます。時刻 0 は 00:00:00 として表示されます。

### フォーマットされた文字列のタイムスタンプへのキャスト

有効な日付と時刻、有効な日付、または有効な時刻で構成した文字列は TIMESTAMP データ型にキャストできます。日付部分はさまざまな形式で指定できます ([TO\\_TIMESTAMP](#) 関数の説明を参照)。ここで得られるタイムスタンプは、YYYY-MM-DD hh:mm:ss 形式で表されます。

## SQL

```
SELECT
  CAST('1 MAR 2022 1:33pm' AS TIMESTAMP) As DateToTS1, -- 2022-03-01 13:33:00
  CAST('3/1/2022 13:33:00' AS TIMESTAMP) As DateToTS2 -- 2022-03-01 13:33:00
```

CAST はフォーマットされた日付を以下のように解決します。

- ・ 省略されている日付部分を 1841-01-01 (論理日付1) に設定します。
- ・ 省略されている時刻部分を 00:00:00 に設定します。
- ・ 月および日の先頭のゼロ (省略されている場合) を挿入します。

秒の小数部を指定している場合は、それをピリオド (.) またはコロン (:) で始めることができます。

- ・ ピリオドは標準の小数部であることを示します。以下に例を示します。
  - 12:00:00.4 の秒数は 10 分の 4 秒
  - 12:00:00.004 の秒数は 1000 分の 4 秒
- ・ コロンは、次に続く値が 1000 分の 1 秒単位であることを示します。12:00:00:4 の秒数は 1000 分の 4 秒です。コロンの後に続けることができる桁数は 3 に制限されています。

ある時刻データ型の文字列を別の時刻データ型にキャストすることもできます。以下の例では、文字列を TIME データ型にキャストし、その結果を TIMESTAMP データ型にキャストします。日付は現在のシステム日付に設定されます。

## SQL

```
SELECT CAST(CAST('14:33:45.78' AS TIME) AS TIMESTAMP) AS TimeToTstamp
```

## 日付値のキャスト

日付は、文字列データ型、数値データ型、または別の日付データ型にキャストできます。

### 文字列への日付のキャスト

日付を文字列データ型にキャストすると、日付全体、またはキャスト先のデータ型で決まる長さの日付が返されます。例えば、以下のクエリでは、現在の日付が "yyyy-mm-dd" として返されず、"yyyy-" のみが返されます。

## SQL

```
SELECT CAST(CURRENT_DATE AS CHAR(5)) AS TruncatedDate
```

CHAR VARYING および CHARACTER VARYING の両データ型では、完全な表示形式が返されます。

## SQL

```
SELECT CAST(CURRENT_TIME AS CHAR VARYING) AS FullDate
```

日付が mm/dd/yyyy などの異なる形式で表示される場合、文字列データ型は ODBC 日付形式 (yyyy-mm-dd) で日付を返します。以下に例を示します。

## SQL

```
SELECT CAST(TO_DATE('03/01/2022', 'MM/DD/YYYY') AS VARCHAR) AS DateFormat -- 2022-03-01
```

### 数値型への日付のキャスト

日付を数値データ型にキャストすると、その日付の \$HOROLOG 値が返されます。これは、1840 年 12 月 31 日を 0 とし、そこから起算した日数を表す整数値です。以下に例を示します。

## SQL

```
SELECT CAST(TO_DATE('01 MAR 2022') AS DECIMAL) AS DateToNumeric -- 66169
```

## POSIXTIME への日付のキャスト

日付を POSIXTIME データ型にキャストすると、エンコードされた 64 ビットの符号付き整数としてタイムスタンプが返されます。日付には時刻部分がないため、00:00:00 としてエンコードされたタイムスタンプに時刻部分が指定されます。

```
SELECT CAST(CURRENT_DATE AS POSIXTIME) As PosixDate
```

CAST は日付を検証します。expression の値が有効な日付でない場合は、SQLCODE -400 エラーが発行されます。

## TIMESTAMP、DATETIME、または SMALLDATETIME への日付のキャスト

日付を TIMESTAMP、DATETIME、または SMALLDATETIME のいずれかのデータ型にキャストすると、YYYY-MM-DD hh:mm:ss 形式のタイムスタンプが返されます。日付には時刻部分がないため、返されるタイムスタンプの時刻部分は常に 00:00:00 です。CAST は日付を検証します。expression の値が有効な日付でない場合は、SQLCODE -400 エラーが発行されます。

以下の例では、DATE データ型の列を TIMESTAMP にキャストします。比較のために POSIXTIME データ型も取り上げています。

```
SELECT TOP 5
  DOB,
  CAST(DOB AS TIMESTAMP) AS TStamp,
  CAST(DOB AS POSIXTIME) AS Posix
FROM Sample.Person
```

## 数値のキャスト

数値は、数値データ型または文字データ型にキャストできます。少ない桁数の値に数値をキャストすると、その数値は四捨五入ではなく、切り捨てられます。

### SQL

```
SELECT
  CAST(98.765 AS INT) AS TruncatedInt, -- 98
  CAST(98.765 AS CHAR) AS TruncatedChar1, -- 9
  CAST(98.765 AS CHAR(4)) AS TruncatedChar2 --98.7
```

負数を CHAR にキャストすると、負の符号だけが返されます。分数を CHAR にキャストすると、小数点だけが返されます。

### SQL

```
SELECT
  CAST(-50 AS CHAR) AS Negative, -- negative sign: -
  CAST(1/4 AS CHAR) AS Fraction -- decimal point: .
```

数値には以下の値を指定できます。

- ・ 0 から 9 までの数字
- ・ 小数点
- ・ 1 つまたは複数の先行符号 (+ または -)
- ・ 指数記号 (E または e の文字) とそれに続く 1 つ以下の正または負の符号

キャストを実行する前に、指数演算の実行、複数の符号の解決、先頭の正符号、末尾の小数点、先頭または末尾のゼロの削除によって、数値が**キャノニック形式**に解決されます。以下に例を示します。

## SQL

```
SELECT
  CAST(1e2 AS DECIMAL(6,2)) AS PositiveExponent, -- 100.000
  CAST(1e-2 AS DECIMAL(6,2)) AS NegativeExponent, -- 0.01
  CAST(+1000 AS DECIMAL(6,2)) AS LeadingSign, -- 1000.00
  CAST(-+1000 AS DECIMAL(6,2)) AS MultipleSigns, -- -1000.00
  CAST(00.100 AS DECIMAL(6,2)) AS LeadingTrailingZeros -- 0.10
```

2 つ続く負の符号はコメント文字として扱われます。数値の中に負の符号を 2 つ続けて記述すると、そのコード行の以降の部分はコメントとして扱われます。数値に桁区切り文字 (コンマ) を入れることはできません。詳細は、“リテラル” を参照してください。

数値はさまざまな数値型に変換できます。以下の例では、CAST で浮動小数点数の  $\pi$  をさまざまな数値データ型に変換しています。整数データ型には切り捨てが適用されます。

## SQL

```
SELECT
  CAST({fn PI()}) As INTEGER) As IntegerPi, -- 3
  CAST({fn PI()}) As SMALLINT) As SmallIntPi, -- 3
  CAST({fn PI()}) As DECIMAL) As DecimalPi, -- 3.141592653589793
  CAST({fn PI()}) As NUMERIC) As NumericPi, -- 3.141592653589793
  CAST({fn PI()}) As DOUBLE) As DoublePi -- 3.1415926535897931159
```

この例では、precision と scale の値が解析され、CAST によって返される値が変更されます。

## SQL

```
SELECT
  CAST({fn PI()}) As DECIMAL) As DecimalPi, -- 3.141592653589793
  CAST({fn PI()}) As DECIMAL(6,3)) As DecimalPiPS -- 3.142
```

数値を日付データ型または時刻データ型にキャストすると、SQL ではゼロ (0) として表示されます。日付または時刻としてキャストした数値は、埋め込み SQL から ObjectScript に渡すと、対応する \$HOROLOG 値として表示されます。

## ビット値のキャスト

expression を 0 または 1 として返すには、それをビット値にキャストします。

expression が以下のいずれかの値であると、CAST から 1 (真) が返されます。

- ・ 1 などのゼロ以外の数値。
- ・ “TRUE”、“True”、“true” など、単語 true の大文字と小文字を組み合わせた表記。省略形の “T” を使用することはできません。

以下の CAST 演算からはすべて 1 が返されます。

## SQL

```
SELECT CAST(1 AS BIT) AS One,
  CAST(7 AS BIT) AS Num,
  CAST(743.6 AS BIT) AS Frac,
  CAST(0.3 AS BIT) AS Zerofrac,
  CAST('tRuE' AS BIT) AS TrueWord
```

expression が以下のいずれかの値であると、CAST から 0 (偽) が返されます。

- ・ 単語 “true” とその大文字と小文字のさまざまな組み合わせ以外の、数値ではない任意の値
- ・ 空の文字列 (‘ ’)
- ・ 数値 0

以下の CAST 演算からはすべて 0 が返されます。

## SQL

```
SELECT CAST(0 AS BIT) AS Zero,
       CAST('FALSE' AS BIT) AS FalseWord,
       CAST('T' AS BIT) AS T,
       CAST('F' AS BIT) AS F,
       CAST(0.0 AS BIT) AS Zerodot,
       CAST('' AS BIT) AS EmptyString
```

## 詳細

## NULL および空文字列値のキャスト

NULL はどのデータ型にキャストしても、NULL として返されます。

## SQL

```
SELECT CAST(NULL AS DATE) AS NullValue
```

空の文字列 ( '') のキャスト結果はデータ型によって異なります。

データ型	空の文字列の戻り値
文字データ型	空の文字列 ( '')
数値データ型	0 (ゼロ)。末尾には小数部のゼロが適切な個数で表示されます。DOUBLE データ型からは 0 (ゼロ) が返されます。小数部のゼロは末尾に表示されません。
DATE データ型	12/31/1840 (論理日付 0)
TIME データ型	00:00:00 (論理時刻 0)
TIMESTAMP、DATETIME、および SMALLDATETIME データ型	空の文字列 ( '')
BIT データ型	0
すべてのバイナリ・データ型	空の文字列 ( '')

## 関連項目

- ・ [データ型](#)
- ・ [CONVERT](#)
- ・ [TO\\_CHAR](#)、[TO\\_DATE](#)、[TO\\_NUMBER](#)、[TO\\_POSIXTIME](#)、[TO\\_TIMESTAMP](#)



## CEILING (SQL)

指定された数値式に等しいか、より大きい最小整数値を返す数値関数です。

### 構文

```
CEILING(numeric-expression)  
{fn CEILING(numeric-expression)}
```

### 概要

CEILING は、*numeric-expression* 以上の最も近い整数値を返します。返される値は、小数桁数が 0 です。*numeric-expression* に NULL 値、空文字列 ("), または数値でない文字列を指定すると、CEILING は NULL を返します。

CEILING は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として呼び出せる点に注意してください。

この関数は、ObjectScript から CEILING() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.CEILING(numeric-expression)
```

### 引数

#### *numeric-expression*

上限値を計算する数。数はリテラルまたは文字列のいずれかになります。数は、科学的記数法で指定できます。

*numeric-expression* が数値型である場合、CEILING は *numeric-expression* と同じデータ型を返します。

### 例

以下の例では、CEILING を使用して小数をその上限となる整数に変換する方法を示します。

#### SQL

```
SELECT CEILING(167.111) AS CeilingNum1,  
       CEILING('167.456') AS CeilingNum2,  
       CEILING(167.999) AS CeilingNum3,  
       CEILING(167.0) AS CeilingNum4
```

すべて 168 を返します。

#### SQL

```
SELECT CEILING(-167.111) AS CeilingNum1,  
       CEILING('-167.456') AS CeilingNum2,  
       CEILING(-167.999) AS CeilingNum3,  
       CEILING(-167.0) AS CeilingNum4
```

すべて -167 を返します。

以下の例では科学的記数法を使用しています。

#### SQL

```
SELECT CEILING(10E-1) // returns 1  
SELECT CEILING('-14E-4') // returns 0  
SELECT CEILING('-10E-1') // returns -1
```

以下の例では、サブクエリを使用して、US Zip Codes (郵便番号) の大きなテーブルを、上限となる Latitude の整数ごとに 1 つの代表都市の郵便番号を含むテーブルに縮小します。

## SQL

```
SELECT City,State,CEILING(Latitude) AS CeilingLatitude
FROM (SELECT City,State,Latitude,CEILING(Latitude) AS CeilingNum
      FROM Sample.USZipCode)
GROUP BY CeilingNum
ORDER BY CeilingNum DESC
```

## 関連項目

- ・ [FLOOR](#)
- ・ [ROUND](#)

## CHAR (SQL)

---

文字列式で指定された ASCII コード値を持つ文字を返す、文字列関数です。

### 構文

```
CHAR(code-value)    {fn CHAR(code-value)}
```

### 説明

CHAR は指定した整数コード値に対応する文字を返します。InterSystems IRIS は Unicode システムであるため、0 から 65535 までのあらゆる Unicode 文字の整数コードを指定できます。code-value に許容範囲外の整数を指定すると、CHAR は NULL を返します。

code-value に数値でない文字列を指定すると、CHAR は空文字列('') を返します。NULL 値を渡すと、CHAR は NULL を返します。

CHAR は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として使用できる点に注意してください。

### 引数

#### code-value

文字に対応する整数コード。

### 例

以下の例は両方とも文字 Z を返します。

#### SQL

```
SELECT CHAR(90) AS CharCode
```

#### SQL

```
SELECT {fn CHAR(90)} AS CharCode
```

以下の例はギリシャ文字ラムダを返します。

#### SQL

```
SELECT {fn CHAR(955)} AS GreekLetter
```

### 関連項目

- SQL 関数: [ASCII](#)、[CHAR\\_LENGTH](#)、[CHARACTER\\_LENGTH](#)
- ObjectScript 関数: [\\$CHAR](#)、[\\$ZLCHAR](#)、[\\$ZWCHAR](#)

# CHARACTER\_LENGTH (SQL)

式の文字の数を返す関数です。

## 構文

```
CHARACTER_LENGTH(expression)
```

## 説明

CHARACTER\_LENGTH は、指定された *expression* について、そのバイト数ではなく、文字数を示す整数値を返します。*expression* は文字列、または数値やデータ・ストリーム・フィールドなどの他のデータ型とすることができます。返されたこの整数には、先頭と末尾の空白や文字列の終了文字の数も含まれます。CHARACTER\_LENGTH は、NULL 値を渡すと NULL を返し、空文字列('') の値を渡すと 0 を返します。

数値は、文字数がカウントされる前にキャノニック形式に解析されます。引用符に囲まれた数値文字列は解析されません。以下の例では、最初の CHARACTER\_LENGTH は 1 を返し (数値の解析によって先頭と末尾の 0 が削除されるため)、2 番目の CHARACTER\_LENGTH は 8 を返します。

## SQL

```
SELECT CHARACTER_LENGTH(007.0000) AS NumLen,  
       CHARACTER_LENGTH('007.0000') AS NumStringLen
```

注釈 CHARACTER\_LENGTH 関数、CHAR\_LENGTH 関数、および DATALENGTH 関数は同一です。これらのすべてがストリーム・フィールド引数を受け取ります。LENGTH 関数および \$LENGTH 関数は、ストリーム・フィールド引数を受け取りません。

LENGTH 関数は、文字数をカウントする前に末尾の空白や文字列の終了文字を削除する点でも、これらの関数とは異なります。また、\$LENGTH は、NULL を渡すと NULL、空文字列を渡すと 0 を返す点でも、これらの関数とは異なります。

## 引数

### *expression*

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる式。基本となるデータ型は (CHAR や VARCHAR など) 任意の文字タイプ、数値、またはデータ・ストリームとすることができます。

CHARACTER\_LENGTH は、INTEGER データ型を返します。

## 例

以下の例は、Sample.Employee テーブルの、州の省略形フィールド (Home\_State) にある文字の数を返します (米国のすべての州には 2 文字の郵便用省略形が付けられています)。

## SQL

```
SELECT DISTINCT CHARACTER_LENGTH(Home_State) AS StateLength  
FROM Sample.Employee
```

以下の例は、各従業員の名前とその名前の文字数を、文字数の昇順で返します。

## SQL

```
SELECT Name,  
       CHARACTER_LENGTH(Name) AS NameLength  
FROM Sample.Employee  
ORDER BY NameLength
```

以下の例は、Sample.Employee テーブル内の文字ストリーム・フィールド (Notes) とバイナリ・ストリーム・フィールド (Picture) の文字数を返します。

#### SQL

```
SELECT DISTINCT CHARACTER_LENGTH(Notes) AS NoteLen
FROM Sample.Employee WHERE Notes IS NOT NULL
```

#### SQL

```
SELECT DISTINCT CHARACTER_LENGTH(Picture) AS PicLen
FROM Sample.Employee WHERE Picture IS NOT NULL
```

以下の例は、CHARACTER\_LENGTH で Unicode 文字がどのように処理されるかを示しています。CHARACTER\_LENGTH では、文字のバイト長に関係なく、文字数がカウントされます。

#### SQL

```
SELECT CHARACTER_LENGTH($CHAR(960)_"FACE")
```

これは、5 を返します。

## 関連項目

- SQL 関数: [CHAR](#)、[CHAR\\_LENGTH](#)、[DATALENGTH](#)、[LENGTH](#)、[LEN](#)、[\\$LENGTH](#)
- ObjectScript 関数: [\\$LENGTH](#)

# CHARINDEX (SQL)

文字列内の部分文字列の位置を返す文字列関数です。オプションで検索開始位置を指定できます。

## 構文

```
CHARINDEX(substring,string[,start])
```

## 概要

CHARINDEX は部分文字列を検索します。一致が見つかった場合、最初に一致した部分文字列の開始位置を返します。先頭を 1 としてカウントされます。一致する部分文字列がない場合、CHARINDEX は 0 を返します。

空の文字列は文字列値です。そのため、文字列引数の値には空の文字列も指定できます。start 引数は空の文字列値を 0 として処理します。ただし、ObjectScript の空文字列は、NULL として InterSystems SQL に渡されることに注意してください。

InterSystems SQL では、NULL は文字列値ではありません。そのため、CHARINDEX 文字列引数のいずれかに NULL を指定すると、NULL が返されます。

CHARINDEX では string 引数または substring 引数に [%Stream.GlobalCharacter field](#) を使用することはできません。これを実行しようとする、SQLCODE -37 エラーが生成されます。SUBSTRING 関数を使用して、%Stream.GlobalCharacter フィールドを取得して %String データ型の値を返し、CHARINDEX で使用できます。

CHARINDEX は、大文字と小文字を区別します。大小文字変換関数の 1 つを使用して、文字列の大文字と小文字のインスタンスをどちらも配置します。

この関数には Transact-SQL 実装との互換性があります。

## CHARINDEX、POSITION、\$FIND、および INSTR

CHARINDEX、POSITION、\$FIND、および INSTR はすべて文字列内の指定された部分文字列を検索し、最初に一致した位置に対応する整数位置を返します。CHARINDEX、POSITION、および INSTR は、一致した部分文字列の最初の文字の整数位置を返します。\$FIND は、最初に一致した部分文字列の次の文字の整数位置を返します。CHARINDEX、\$FIND、および INSTR では、部分文字列の検索を開始する位置を指定できます。INSTR では、その開始位置から数えて何個目の部分文字列かを指定することもできます。

以下に、4 つの関数にすべてのオプション引数を指定した例を示します。string および substring の位置はそれぞれの関数で異なります。

### SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

部分文字列を検索する関数のリストは、[“文字列操作”](#) を参照してください。

## 引数

### substring

string 内で検索する部分文字列。

### string

部分文字列検索の検索対象となる文字列式。

## 起動

正の整数で指定する、部分文字列検索の開始地点を示す引数 (オプション)。string の最初から文字数がカウントされ、カウントは 1 を基準とします。string の先頭から検索するには、この引数を省略するか、0 または 1 の start を指定します。負の数値、空の文字列、NULL、または数値以外の値は 0 として処理されます。

CHARINDEX は、INTEGER [データ型](#)を返します。

## 例

以下の例は、ヌクレオチド配列で部分文字列 TTAGGG の最初の出現箇所を検索します。検索対象となる文字列内の、この部分文字列の文字位置 7 を返します。

### SQL

```
SELECT CHARINDEX('TTAGGG','TTAGTCTTAGGGACATTAGGG')
```

以下の例は、部分文字列 'Fred' を含むすべての Name フィールドの値を検索します。

### SQL

```
SELECT Name
FROM Sample.Person
WHERE CHARINDEX('Fred',Name)>0
```

以下の例では、SUBSTRING を使用して、DNA ヌクレオチド配列が含まれる %Stream.GlobalCharacter フィールドの最初の 1000 文字で部分文字列 TTAGGG の最初の出現箇所を CHARINDEX で検索できます。

### SQL

```
SELECT CHARINDEX('TTAGGG',SUBSTRING(DNASeq,1,1000)) FROM Sample.DNASequences
```

以下の例は、最初から数えて 10 文字目以降で部分文字列を一致させます。

### SQL

```
SELECT CHARINDEX('Re','Reduce, Reuse, Recycle',10)
```

これは 16 を返します。

以下の例は、文字列の長さを超えて start 位置を指定しています。

### SQL

```
SELECT CHARINDEX('Re','Reduce, Reuse, Recycle',99)
```

これは 0 を返します。

以下の例は、CHARINDEX が他の文字列値と同様に空の文字列 ('') を処理することを示しています。

### SQL

```
SELECT CHARINDEX('', 'Fred Astore'),
       CHARINDEX('A', ''),
       CHARINDEX('', '')
```

上記の例では、最初と 2 番目の CHARINDEX 関数は 0 (一致しない) を返します。3 番目の CHARINDEX 関数では、空文字列が位置 1 の空文字列に一致するため 1 が返されます。

以下の例は、CHARINDEX が文字列値として NULL を処理しないことを示します。いずれかの文字列に NULL を指定すると、常に NULL が返されます。



## SQL

```
SELECT CHARINDEX(NULL, 'Fred Astore'),  
       CHARINDEX('A', NULL),  
       CHARINDEX(NULL, NULL)
```

## 関連項目

- ・ [\\$FIND](#) 関数
- ・ [INSTR](#) 関数
- ・ [POSITION](#) 関数
- ・ [文字列操作](#)

## CHAR\_LENGTH (SQL)

式の文字の数を返す関数です。

### 構文

```
CHAR_LENGTH(expression)
```

### 概要

CHAR\_LENGTH は、指定された *expression* について、そのバイト数ではなく、文字数を示す整数値を返します。*expression* は文字列、または数値やデータ・ストリーム・フィールドなどの他のデータ型とすることができます。返されたこの整数には、先頭と末尾の空白や文字列の終了文字の数も含まれます。CHARACTER\_LENGTH は、NULL 値を渡すと NULL を返し、空文字列 (') の値を渡すと 0 を返します。

数値は、文字数がカウントされる前にキャノニック形式に解析されます。引用符に囲まれた数値文字列は解析されません。以下の例では、最初の CHAR\_LENGTH は 1 を返し (数値の解析によって先頭と末尾の 0 が削除されるため)、2 番目の CHAR\_LENGTH は 8 を返します。

### SQL

```
SELECT CHAR_LENGTH(007.0000) AS NumLen,  
       CHAR_LENGTH('007.0000') AS NumStringLen
```

**注釈** CHAR\_LENGTH 関数、CHARACTER\_LENGTH 関数、および DATALENGTH 関数は同一です。これらのすべてがストリーム・フィールド引数を受け取ります。LENGTH 関数および \$LENGTH 関数は、ストリーム・フィールド引数を受け取りません。

LENGTH 関数は、文字数をカウントする前に末尾の空白や文字列の終了文字を削除する点でも、これらの関数とは異なります。

また、\$LENGTH は、NULL を渡すと NULL、空文字列を渡すと 0 を返す点でも、これらの関数とは異なります。\$LENGTH は、データ型 SMALLINT を返す点で、他の length 関数とは異なります。他の length 関数はすべてデータ型 INTEGER を返します。

### 引数

#### *expression*

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる式。基本となるデータ型は (CHAR や VARCHAR など) 任意の文字タイプ、数値、またはデータ・ストリームとすることができます。

CHAR\_LENGTH は、INTEGER データ型を返します。

### 例

以下の例は、Sample.Employee テーブルの、州の省略形フィールド (Home\_State) にある文字の数を返します (米国のすべての州には 2 文字の郵便用省略形が付けられています)。

### SQL

```
SELECT DISTINCT CHAR_LENGTH(Home_State) AS StateLength  
FROM Sample.Employee
```

以下の例は、各従業員の名前とその名前の文字数を、文字数の昇順で返します。

## SQL

```
SELECT Name,
       CHAR_LENGTH(Name) AS NameLength
FROM Sample.Employee
ORDER BY NameLength
```

以下の例は、Sample.Employee テーブル内の文字ストリーム・フィールド (Notes) とバイナリ・ストリーム・フィールド (Picture) の文字数を返します。

## SQL

```
SELECT DISTINCT CHAR_LENGTH(Notes) AS NoteLen
FROM Sample.Employee WHERE Notes IS NOT NULL
```

## SQL

```
SELECT DISTINCT CHAR_LENGTH(Picture) AS PicLen
FROM Sample.Employee WHERE Picture IS NOT NULL
```

以下の埋め込み SQL の例は、CHAR\_LENGTH で Unicode 文字がどのように処理されるかを示しています。CHAR\_LENGTH では、文字のバイト長に関係なく、文字数がカウントされます。

## ObjectScript

```
SET a=$CHAR(960)_"FACE"
WRITE !,a
&sql(SELECT CHAR_LENGTH(:a) INTO :b)
IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }
ELSE {WRITE !,"The CHAR length is ",b }
```

これは、5 を返します。

## 関連項目

- SQL 関数: [CHAR](#)、[CHARACTER\\_LENGTH](#)、[DATALENGTH](#)、[LENGTH](#)、[LEN](#)、[\\$LENGTH](#)
- ObjectScript 関数: [\\$LENGTH](#)

## COALESCE (SQL)

NULL でない最初の式の値を返す関数です。

### 構文

```
COALESCE(expression,expression [,...])
```

### 概要

COALESCE 関数は式のリストを左から右に評価し、NULL でない最初の式の値を返します。すべての式が NULL と評価されると、NULL が返されます。

文字列は、先頭と末尾の空白が維持され、そのまま返されます。数値は、先頭と末尾のゼロが削除され、キャノニック形式で返されます。

NULL の処理の詳細は、“[NULL および空文字列](#)” を参照してください。

### 返り値のデータ型

数値以外の式 (文字列や日付など) の評価ではすべてが同じデータ型である必要があり、そのデータ型の値が返されます。互換性のないデータ型の式を指定すると、SQLCODE -378 エラーと、データ型の不一致を示すエラー・メッセージが返されます。[CAST](#) 関数を使用すると、*expression* を互換性のあるデータ型に変換できます。

数値式にはさまざまなデータ型を使用できます。複数のデータ型で数値式を指定した場合、考えられるすべての結果値との互換性が最も高く、[データ型の優先順位](#)が最も高い *expression* データ型で値が返されます。

リテラル値 (文字列、数値、または NULL) は、データ型 VARCHAR として扱われます。指定する式が 2 つだけの場合、リテラル値は数値式と互換性があります。最初の *expression* が数値式である場合はそのデータ型が返され、最初の *expression* がリテラル値である場合は VARCHAR データ型が返されます。

### 引数

#### *expression*

評価される一連の式。複数の式はコンマで区切ったリストで指定します。式リストでは、140 個までの式を指定できます。

### NULL を処理する関数の比較

以下の表は、さまざまな SQL 比較関数を示します。論理比較テストが True (A は B と同じ) の場合、各関数は特定の値を返し、False (A は B と同じではない) の場合、別の値を返します。これらの関数により、NULL の論理比較を実行できます。実際の [等値 \(または非等値\) 条件比較](#) で NULL を指定することはできません。

SQL 関数	比較テスト	返り値
COALESCE(ex1,ex2,...)	各引数で ex = NULL	True の場合、次の ex 引数をテスト すべての ex 引数が True (NULL) の 場合、NULL を返す  False の場合、ex を返す
IFNULL(ex1,ex2) [2 つの引数形式]	ex1 = NULL	True の場合、ex2 を返す  False の場合、NULL を返す
IFNULL(ex1,ex2) [3 つの引数形式]	ex1 = NULL	True の場合、ex2 を返す  False の場合、ex3 を返す
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True の場合、ex2 を返す  False の場合、ex1 を返す
ISNULL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す  False の場合、ex1 を返す
NVL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す  False の場合、ex1 を返す
NULLIF(ex1,ex2)	ex1 = ex2	True の場合、NULL を返す  False の場合、ex1 を返す

## 例

以下の例は、一連の値を取り、NULL 以外の値を持つ最初の値 (d) を返します。ObjectScript の空文字列 ("") が、InterSystems SQL では NULL として扱われる点に注意してください。

### ObjectScript

```
SELECT COALESCE("", "", "", "firstdata", "", "nextdata")
```

以下の例は、2 つの列の値を左から右に比較し、NULL でない最初の列の値を返します。FavoriteColors 列には NULL の行があり、Home\_State 列は NULL になりません。COALESCE では、2 つを比較するために、FavoriteColors must を文字列としてキャストする必要があります。

### SQL

```
SELECT TOP 25 Name, FavoriteColors, Home_State,
COALESCE(CAST(FavoriteColors AS VARCHAR), Home_State) AS CoalesceCol
FROM Sample.Person
```

以下の Dynamic SQL の例は、COALESCE を他の NULL を処理する関数と比較します。

## ObjectScript

```
SET myquery = "SELECT TOP 50 %ID, "_
               "IFNULL(FavoriteColors,'blank') AS Ifn2Col, "_
               "IFNULL(FavoriteColors,'blank','value') AS Ifn3Col, "_
               "COALESCE(CAST(FavoriteColors AS VARCHAR),Home_State) AS CoalesceCol, "_
               "ISNULL(FavoriteColors,'blank') AS IsnullCol, "_
               "NULLIF(FavoriteColors,$LISTBUILD('Orange')) AS NullifCol, "_
               "NVL(FavoriteColors,'blank') AS NvlCol" _
               " FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

## 関連項目

- ・ [CASE コマンド](#)
- ・ [IFNULL 関数](#)
- ・ [ISNULL 関数](#)
- ・ [NULLIF 関数](#)
- ・ [NVL 関数](#)

# CONCAT (SQL)

2 つの文字列式を連結した結果として、文字列を返すスカラ文字列関数です。

## 構文

```
{fn CONCAT(string1,string2)}
```

## 説明

- ・ {fn CONCAT(string1,string2)} は、2 つの文字列を連結し、連結後の文字列を返します。この構文は連結演算子 (||) の使用に相当します。STRING 関数を使用することで、2 つ以上の式を連結して 1 つの文字列にすることもできます。

以下の文は、上位 5 つの名前と姓をテーブルから選択し、LastName 列の値と FirstName 列の値を連結してコンマで区切ります。

### SQL

```
SELECT TOP 5
  FirstName, LastName,
  {fn CONCAT({fn CONCAT(LastName, ','), FirstName})} AS FullName
FROM Sample.Person
```

FirstName	LastName	FullName
Quigley	Ulman	Ulman,Quigley
Buzz	Woo	Woo,Buzz
Mario	Mastrolito	Mastrolito,Mario
Julie	Noodleman	Noodleman,Julie
Lawrence	Quincy	Quincy,Lawrence

例：2 つの文字列を連結

## 引数

### string1,string2

連結する文字列式。式は列の名前や文字リテラル、数値、または他のスカラ関数の結果を指定できます。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

数値または数値文字列の任意の組み合わせを連結できます。連結の結果は数値文字列です。InterSystems SQL では、連結の前に数値を**キャノニック形式**に変換します (指数が展開され、先頭のゼロと末尾のゼロは削除されます)。数値文字列は、連結の前にキャノニック形式に変換されません。

先頭または末尾の空白は文字列に連結できます。NULL 値を文字列に連結すると NULL になります。



## 例

### 2つの文字列の連結

以下の文では、Home\_State 列と Home\_City 列を連結して、場所の値を生成します。CONCAT 関数と連結演算子を使用して、2 回の連結が示されています。

#### SQL

```
SELECT TOP 5
  {fn CONCAT({fn CONCAT(HomeCity,', '), HomeState)} AS LocationWithConcatFunction,
  HomeCity||', '||HomeState AS LocationWithConcatOperator
FROM Sample.Person
```

LocationWithConcatFunction	LocationWithConcatOperator
Denver, CO	Denver, CO
Boston, MA	Boston, MA
Albuquerque, NM	Albuquerque, NM
Jacksonville, FL	Jacksonville, FL
Lexington, KY	Lexington, KY

以下の文は文字列と NULL を連結し、NULL の列を返します。

#### SQL

```
SELECT {fn CONCAT(HomeState,NULL)} AS StrNull
FROM Sample.Person
```

StrNull
NULL
NULL
NULL
NULL
NULL

以下の文は、連結前に数値がキャノニック形式に変換される例を示しています。このような状況を避けるには、この文の 2 番目の部分に示すように数値を文字列として指定します。

#### SQL

```
SELECT TOP 5
  {fn CONCAT(HomeState,0012.00E2)} AS StrNum,
  {fn CONCAT(HomeState,'0012.00E2')} AS StrStrNum
FROM Sample.Person
```

StrNum	StrStrNum
CO1200	CO0012.00E2
MA1200	MA0012.00E2
NM1200	NM0012.00E2
FL1200	FL0012.00E2
KY1200	KY0012.00E2

以下の文は、末尾の空白が保持されることを示しています。2 文字の状態フィールドを 10 個の空白と連結すると、連結した列のそれぞれの値は長さが 12 になります。

## SQL

```
SELECT TOP 5
  HomeState,
  CHAR_LENGTH({fn CONCAT(HomeState, '          ')}) AS StrSpace
FROM Sample.Person2
```

HomeState	StrSpace
CO	12
MA	12
NM	12
FL	12
KY	12

## 関連項目

- ・ [ASCII](#)
- ・ [CHAR](#)
- ・ [STRING](#)
- ・ [SUBSTRING](#)

## CONVERT (SQL)

与えられた式を、特定のデータ型に変換する関数です。

### 構文

```
CONVERT(type,expression)  
CONVERT(type,expression,formatCode)  
{fn CONVERT(expression,type)}
```

### 説明

CONVERT 関数は、特定のデータ型の式を別のデータ型の対応値に変換します。CONVERT は [CAST](#) と似ていますが、以下の相違点があります。

- CONVERT のほうが CAST よりも柔軟です。例えば、CONVERT はストリーム・データの変換をサポートし、日付値と時刻値の書式を設定できます。
- CAST のほうが CONVERT よりもデータベースでの互換性に優れています。CAST が ANSI SQL-92 標準を使用して実装されるのに対し、CONVERT の実装はデータベース固有です。InterSystems SQL が提供する CONVERT の実装には、MS SQL Server および ODBC との互換性があります。

### MS SQL Server の互換性

この CONVERT の実装は、MS SQL Server と互換性のある一般的な InterSystems IRIS® スカラ関数です。この関数は日付と時刻の書式設定、および[ストリーム・データ](#)の変換をサポートします。

- CONVERT([type](#),[expression](#)) は指定されたデータ型に式を変換します。InterSystems SQL でサポートされているデータ型の一覧については、“[データ型](#)”を参照してください。

以下の文は、10 進数 ( $\pi$  の近似値) を文字列に変換し、その数を 4 文字に切り捨てます。

#### SQL

```
SELECT CONVERT(CHAR(4),3.14159) -- '3.14'
```

例：

- [数値型間の変換](#)
  - [文字列間の変換](#)
  - [文字列へのストリーム・データの変換](#)
  - [数値型への文字列の変換](#)
  - [タイムスタンプへの日付の変換](#)
  - [数値型への日付の変換](#)
- CONVERT(*type*,*expression*,[formatCode](#)) は、指定されたデータ型に式を変換し、指定された形式コードに基づいて戻り値を書式設定します。

以下の文は、日付の文字列を TIMESTAMP データ型に変換します。この関数は、形式コード 103 (mm/dd/yy の形式を表す) に基づいて入力を変換します。すべての形式コードは、[formatCode](#) 引数の説明を参照してください。

#### SQL

```
SELECT CONVERT(TIMESTAMP,'1/1/99',103) -- '01/01/1999 00:00:00'
```

例：文字列への日付の変換

## ODBC 互換性

CONVERT のこの実装は、一般的な InterSystems IRIS ODBC スカラ関数です。この関数は日付と時刻の書式設定をサポートしていません。また、ストリーム・データの変換もサポートしません。

- ・ {fn CONVERT(expression,type)} は、指定されたデータ型に式を変換します。CONVERT のこの実装では、各データ型の引数の前に SQL\_ キーワードを付加する必要があります。これらのデータ型はパラメータを受け入れません。例えば、文字列データ型には最大長を設定できません。数値データ型には精度（最大桁数）と桁（小数部の最大桁数）を設定できません。

以下の文は 10 進数を文字列に変換します。返された文字列は切り捨てられません。SQL\_VARCHAR(4) などの最大長の指定は許可されません。

### SQL

```
SELECT {fn CONVERT(3.14159,SQL_VARCHAR) } -- '3.14159'
```

例：

- ・ [数値型間の変換](#)
- ・ [文字列間の変換](#)
- ・ [数値型への文字列の変換](#)
- ・ [タイムスタンプへの日付の変換](#)
- ・ [数値型への日付の変換](#)

## 引数

### type

変換した [expression](#) のデータ型です。指定できる型は、InterSystems IRIS の CONVERT() 構文を使用しているか ODBC の {fn CONVERT()} 構文を使用しているかによって異なります。

### CONVERT() 関数

InterSystems IRIS の CONVERT() 構文は “[データ型](#)” で説明されているデータ型をサポートします。指定できる一般的なデータ型は以下のとおりです。

- ・ 文字列データ型：CHAR、CHARACTER、VARCHAR。文字列のタイプによっては、必要に応じて最大長パラメータを指定できます。例：VARCHAR(10)
- ・ 数値データ型：INTEGER、DECIMAL、DOUBLE、MONEY。数値の種類によっては、必要に応じて精度と桁のパラメータを指定できます。例：DECIMAL(8,4)
- ・ 日付と時刻のデータ型：DATE、TIME、TIMESTAMP、POSIXTIME
- ・ ビットおよびバイナリ・データ型：BIT、BINARY、VARBINARY

### {fn CONVERT()} 関数

ODBC の {fn CONVERT()} 構文は、CONVERT() 構文よりも限定的なデータ型のセットをサポートします。サポートされるデータ型は CONVERT() 構文に指定するデータ型に対応していますが、先頭に SQL\_ キーワードを付加する必要があります。

指定できる有効なデータ型を 2 つのグループに分けて以下の表に示します。

- ・ 1 番目のグループはデータ値とデータ型の両方を変換します。例えば、%Date ソースを SQL\_VARCHAR に変換すると、日付がテキスト値に変わり、クエリではその値が VARCHAR データ型として処理されます。
- ・ 2 番目のグループは、データ型を変換しますがデータ値を変換しません。例えば、%Date ソースを INTEGER に変換しても %Date ソースは変化しませんが、クエリでは整数形式の日付が INTEGER データ型として処理されます。

ソース	有効な変換タイプ (型と値を変換)	有効な変換タイプ (型のみを変換)
任意の数値データ型	SQL_VARCHAR、SQL_DOUBLE、SQL_DATE、SQL_TIME	N/A
%String	SQL_DATE、SQL_TIME、SQL_TIMESTAMP	N/A
%Date	SQL_VARCHAR、SQL_POSIXTIME、SQL_TIMESTAMP	SQL_INTEGER、SQL_BIGINT、SQL_SMALLINT、SQL_TINYINT、SQL_DATE
%Time	SQL_VARCHAR、SQL_POSIXTIME、SQL_TIMESTAMP	SQL_INTEGER、SQL_BIGINT、SQL_SMALLINT、SQL_TINYINT、SQL_TIME
%PosixTime	SQL_TIMESTAMP、SQL_DATE、SQL_TIME	SQL_VARCHAR、SQL_INTEGER、SQL_BIGINT、SQL_SMALLINT、SQL_TINYINT
%TimeStamp	SQL_POSIXTIME、SQL_DATE、SQL_TIME	SQL_VARCHAR、SQL_INTEGER、SQL_BIGINT、SQL_SMALLINT、SQL_TINYINT
任意の非ストリーム・データ型	SQL_INTEGER、SQL_BIGINT、SQL_SMALLINT、SQL_TINYINT	SQL_DOUBLE

この関数にデータ型を指定するときは、以下の点に注意してください。

- ・ SQL\_VARCHAR は標準的な ODBC 表現です。SQL\_VARCHAR に変換する場合、日時は適切な ODBC 表現に変換されます。つまり、数値のデータタイプは文字列に変換されます。SQL\_VARCHAR から変換する場合、値は有効な ODBC Time、Timestamp、Date 表現である必要があります。
- ・ 時刻値を SQL\_TIMESTAMP または SQL\_POSIXTIME に変換すると、指定されていない日付は既定で 1841-01-01 になります。CONVERT() 構文では、日付は既定値の 1900-01-01 となります。
- ・ 日付値を SQL\_TIMESTAMP または SQL\_POSIXTIME に変換すると、時刻は既定で 00:00:00 になります。
- ・ 秒の小数部の前には、ピリオド (.) とコロン (:) のいずれかを記述できます。これらの記号は、意味が異なります。ピリオドは標準的な小数を示すため、12:00:00.4 は 10 分の 4 秒を示し、12:00:00.004 は 1000 分の 4 秒を示します。コロンは、それに続く値が 1000 分の 1 秒単位であることを示すため、12:00:00:4 は 1000 分の 4 秒を示します。コロンの後に続けることができる桁数は 3 に制限されています。
- ・ 整数データ型または SQL\_DOUBLE データ型への変換の場合、日付と時刻も含め、CONVERT 関数によってデータ値が数値表記に変換されます。SQL\_DATE の場合は 1841 年 1 月 1 日以降の日数となります。SQL\_TIME の場合、これは午前 0 時以降の経過秒数となります。CONVERT で数値以外の文字が検出されると、その文字で入力文字列が切り捨てられます。整数データ型では、小数点以下が切り捨てられ、数値の整数部分のみが返されます。

## expression

新しいデータ型への変換元とする式です。expression には、単一の文字列値などのスカラまたはテーブル列などの非スカラを指定できます。expression の有効な値は、[type](#) で指定したデータ型によって異なります。

- ・ 定義されたデータ型 (ObjectScript で提供されるホスト変数など) が expression がない場合、そのデータ型は既定で文字列データ型に設定されます。
- ・ expression にストリーム・データがある場合に {fn CONVERT(expression,type)} 構文を使用すると、CONVERT から SQLCODE -37 エラーが発行されます。
- ・ expression が NULL の場合、指定した型に関係なく、変換された値は NULL のままとなります。
- ・ expression が空の文字列 (') または数値以外の文字列である場合、返される値は指定した type によって異なります。
  - type が文字列データ型の場合、CONVERT からは指定した値が返されます。
  - type が数値データ型または TIME、SQL\_TIME、SQL\_DATE 型の場合、CONVERT からは 0 (ゼロ) が返されます。

無効な値を type に指定すると SQLCODE -141 エラーが発生します。

## formatCode

日付、日付時刻、時刻の各形式を指定する整数コードです。

formatCode を使用して、日付、時刻、タイムスタンプのデータ型から文字列に変換するときの出力を定義します。以下に例を示します。

## SQL

```
SELECT CONVERT(VARCHAR,TO_DATE('22 FEB 2022'),1) -- '02/22/22'
```

文字列から日付、時刻、タイムスタンプのデータ型に変換するときに、formatCode を使用して入力を定義することもできます。以下に例を示します。

## SQL

```
SELECT CONVERT(DATE,'22 FEB 2022',106) -- '02/22/2022'
```

CONVERT() 構文のみが formatCode をサポートします。

無効な形式または formatCode に一致しない形式で expression を指定すると、SQLCODE -141 エラーが生成されます。存在しない formatCode を指定すると、1900-01-01 00:00:00 が返されます。

以下の表に、サポートされている形式コードを示します。

- ・ 1 列目は 2 桁の年を出力するコードの一覧です。
- ・ 2 列目は 4 桁の年を出力するコード、または年を出力しないコードの一覧です。

2 桁の年コード	4 桁の年コード	形式
N/A	0 または 100	Mon dd yyyy hh:mmAM (または PM)
1	101	mm/dd/yy
2	102	yy.mm.dd
3	103	dd/mm/yy
4	104	dd.mm.yy
5	105	dd-mm-yy
6	106	dd Mon yy

2 桁の年コード	4 桁の年コード	形式
7	107	Mon dd, yy (dd < 10 のとき先頭に 0 は入りません)
N/A	8 または 108	hh:mm:ss
N/A	9 または 109	Mon dd yyyy hh:mm:ss:nnnAM (または PM)
10	110	mm-dd-yy
11	111	yy/mm/dd
12	112	yymmdd
N/A	13 または 113	dd Mon yyyy hh:mm:ss:nnn (24 時間制)
N/A	14 または 114	hh:mm:ss:nnn (24 時間制)
N/A	20 または 120	yyyy-mm-dd hh:mm:ss (24 時間制)
N/A	21 または 121	yyyy-mm-dd hh:mm:ss:nnn (24 時間制)
N/A	126	yyyy-mm-ddThh:mm:ss:nnn (24 時間制)
N/A	130	dd Mon yyyy hh:mm:ss:nnnAM (または PM)
N/A	131	dd/mm/yyyy hh:mm:ss:nnnAM (または PM)

### 値の範囲

許容される日付の範囲は 0001-01-01 から 9999-12-31 です。

### 既定値

CONVERT() 構文の場合、時刻値を TIMESTAMP、POSIXTIME、DATETIME、または SMALLDATETIME に変換すると、日付は既定で 1900-01-01 になります。{fn CONVERT()} 構文の場合、日付は既定で 1841-01-01 となります。

日付値を TIMESTAMP、POSIXTIME、DATETIME、または SMALLDATETIME に変換すると、時刻は既定で 00:00:00 になります。

### 既定の形式

formatCode を指定しない場合は、指定された値から CONVERT が形式を決定しようとします。形式を決定できない場合には、既定で formatCode 100 (mm-dd-yy) になります。

### 2 桁の年

00 から 49 までの 2 桁の年は 21 世紀の日付 (2000 年から 2049 年まで) に変換されます。

50 から 99 までの 2 桁の年は 20 世紀の日付 (1950 年から 1999 年まで) に変換されます。

### 秒の小数部

秒の小数部をピリオド (.) またはコロン (:) で始めることができます。これらの記号は、意味が異なります。

- ・ ピリオド (既定) — すべての formatCode 値で有効です。ピリオドは標準の小数部であることを示します。例えば、12:00:00.4 の秒数は 10 分の 4 秒、12:00:00.004 の秒数は 1000 分の 4 秒です。CONVERT では、小数部の桁数に制限はありません。
- ・ コロン — formatCode 値が 9/109、13/113、14/114、130、131 の場合にのみ有効です。コロンは、次に続く数が 1000 分の秒数であることを示します。例えば、12:00:00:4 の秒数は 1000 分の 4 秒 (12:00:00.004) です。小数部には最大で 3 桁を指定できます。



## 例

### 数値型間の変換

以下の例では、DECIMAL と DOUBLE のデータ型を使用した小数の変換を比較しています。InterSystems IRIS の CONVERT() 構文を使用しています。DOUBLE への変換によって精度が低下します。

#### SQL

```
SELECT CONVERT(DECIMAL,-123456789.0000123456789) AS DecimalVal, -- -123456789.0000123457
       CONVERT(DOUBLE,-123456789.0000123456789) AS DoubleVal -- -123456789.00001235306
```

以下の文は、ODBC の {fn CONVERT()} 構文を使用して上記と同様の変換を実行します。この構文は DECIMAL データ型をサポートしていないことから、値は DOUBLE データ型にのみ変換されます。

#### SQL

```
SELECT {fn CONVERT(-123456789.0000123456789,SQL_DOUBLE) } AS DecimalVal -- -123456789.00001235306
```

### 文字列間の変換

この例は、expression よりも文字列長が短い出力文字列長を指定し、VARCHAR から VARCHAR への変換で文字列を切り詰めています。切り詰めは、InterSystems IRIS の CONVERT() 構文でのみサポートされています。ODBC の {fn CONVERT()} 構文で唯一サポートされている文字列形式は SQL\_VARCHAR です。

#### SQL

```
SELECT CONVERT(VARCHAR(5),'Hello, World') As TruncatedValue -- 'Hello'
```

文字データ型で文字列長を指定していない場合、最大長は既定で 30 文字になります。

#### SQL

```
SELECT CONVERT(VARCHAR,'This string is more than 30 characters.') --This string is more than 30 ch
```

#### SQL

```
SELECT {fn CONVERT('This string is more than 30 characters.',SQL_VARCHAR) } --This string is more than
30 ch
```

CONVERT() 構文の場合、BINARY 型または VARBINARY 型への変換にも、この最大長が適用されます。または、長さの指定されていないデータ型は、[データ型](#) テーブルに示すように、MAXLEN が 1 の文字にマップされます。

### 文字列へのストリーム・データの変換

以下の例では、文字ストリーム・フィールドを VARCHAR テキスト文字列に変換します。また、CHAR\_LENGTH を使用して、文字ストリーム・フィールドの長さを表示します。

#### SQL

```
SELECT Notes,CONVERT(VARCHAR(80),Notes) AS NoteText,CHAR_LENGTH(Notes) AS TextLen
FROM Sample.Employee WHERE Notes IS NOT NULL
```

ODBC の {fn CONVERT()} 構文は文字ストリームをサポートしていません。

### 文字列への日付の変換

以下の例は、DOB(誕生日)列の日付を SQL\_VARCHAR データ型に変換します。ここで得られる文字列は yyyy-mm-dd 形式で表されます。

## SQL

```
SELECT DOB, CONVERT(VARCHAR, DOB) AS DOBtoVChar
FROM Sample.Person
```

## SQL

```
SELECT DOB, {fn CONVERT(DOB, SQL_VARCHAR)} AS DOBtoVChar
FROM Sample.Person
```

以下の例では、誕生日フィールド (DOB) から、形式設定した文字列への変換をいくつか示しています。それぞれの変換後、出力例の日付文字列がコメントに表示されます。

## SQL

```
SELECT DOB,
    CONVERT(VARCHAR(20), DOB) AS DOBDefault, -- Mar 20 1983 12:00AM
    CONVERT(VARCHAR(20), DOB, 100) AS DOB100, -- Mar 20 1983 12:00AM
    CONVERT(VARCHAR(20), DOB, 107) AS DOB107, -- Mar 20, 1983
    CONVERT(VARCHAR(20), DOB, 114) AS DOB114, -- 00:00:00.000
    CONVERT(VARCHAR(20), DOB, 126) AS DOB126 -- 1983-03-20T00:00:00:
FROM Sample.Person
```

既定の形式とコード 100 の形式は同じです。DOB フィールドには時刻値が格納されていないため、時刻を表示する形式 (ここでは、既定値、100、114、126) では、12:00AM (真夜中) を示すゼロ値が指定されます。コード 126 の形式は、日付と時刻の文字列を空白なしで表示します。

InterSystems IRIS の CONVERT() 構文のみが日付文字列の書式設定をサポートしています。ODBC の {fn CONVERT()} 構文ではサポートしていません。

## 数値型への文字列の変換

以下の例では、複数の文字列の混在が整数に変換されます。InterSystems IRIS では、最初の数値以外の文字で文字列を切り捨て、結果として得られる数値を**キャノニック形式**に変換します。

## SQL

```
SELECT CONVERT(INTEGER, '007 James Bond') -- 7
```

## SQL

```
SELECT {fn CONVERT('007 James Bond', SQL_INTEGER)} -- 7
```

## タイムスタンプへの日付の変換

以下の例は、DOB (誕生日) 列の日付をタイムスタンプ・データ型に変換します。ここで得られるタイムスタンプは、yyyy-mm-dd hh:mm:ss 形式で表されます。

## SQL

```
SELECT DOB, CONVERT(TIMESTAMP, DOB) AS DOBtoTstamp
FROM Sample.Person
```

## SQL

```
SELECT DOB, {fn CONVERT(DOB, SQL_TIMESTAMP)} AS DOBtoTstamp
FROM Sample.Person
```

## 数値型への日付の変換

以下の例は、DOB (誕生日) 列の日付を整数データ型に変換します。生成される整数は 1840 年 12 月 31 日からの \$HOROLOG カウントになります。

## SQL

```
SELECT DOB, CONVERT(INTEGER, DOB) AS DOBtoInt
FROM Sample.Person
```

## SQL

```
SELECT DOB, {fn CONVERT(DOB, SQL_INTEGER)} AS DOBtoInt
FROM Sample.Person
```

## ビット値の変換

BIT データ型変換を実行できます。許容される値は、1、0、または NULL です。その他の値を指定すると、InterSystems IRIS によって SQLCODE -141 エラーが発行されます。この例では NULL に対する 2 種類の BIT 変換を示します。

### ObjectScript

```
SET a=" "
&sql(SELECT CONVERT(BIT,:a),
      CONVERT(BIT,NULL)
      INTO :x,:y)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"the host variable is:",x
WRITE !,"the NULL keyword is:",y
```

埋め込み SQL を使用してホスト変数に空の文字列を格納している場合にのみ、ビット変換でその文字列を指定できます。CONVERT(BIT, ' ') のように空の文字列を直接指定すると、InterSystems IRIS によって SQLCODE -141 エラーが発行されます。

## 詳細

### CONVERT クラス・メソッド

以下の構文で示すように、データ型を SQL\_ キーワードとして指定し、CONVERT() メソッドを呼び出すことで、データ型を変換することもできます。

```
$SYSTEM.SQL.Functions.CONVERT(expression, SQL_convertToType, SQL_convertFromType)
```

以下に例を示します。

### ObjectScript

```
write $SYSTEM.SQL.Functions.CONVERT(66225, "SQL_VARCHAR", "SQL_DATE")
```

## 関連項目

- [CAST](#)
- [データ型](#)

## COS (SQL)

与えられた角度の三角関数のコサインを、ラジアン表示で返すスカラ数値関数です。

### 構文

```
{fn COS(numeric-expression)}
```

### 引数

引数	説明
<i>numeric-expression</i>	数値式。ラジアンで表示される角度です。

COS は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。COS は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

### 概要

COS は、任意の数値を取り、浮動小数点数でコサインを返します。返り値は -1 から 1 までの範囲の値です。NULL 値を渡すと、COS は NULL を返します。COS は、非数値文字列を数値 0 (ゼロ) として扱います。

COS は、有効桁数が 19 で小数桁数が 18 の値を返します。

COS は、{} 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

### 例

以下の例は、2 つのサイン値に対する COS の実行結果です。

#### SQL

```
SELECT {fn COS(0.52)} AS Cosine
```

これは、0.86781 を返します。

#### SQL

```
SELECT {fn COS(-.31)} AS Cosine
```

これは、0.95233 を返します。

### 関連項目

- SQL 関数: [ACOS](#)、[ASIN](#)、[ATAN](#)、[COT](#)、[SIN](#)、[TAN](#)
- Cach ObjectScript 関数: [\\$ZCOS](#)

# COT (SQL)

与えられた角度の三角関数のコタンジェントを、ラジアン表示で返すスカラ数値関数です。

## 構文

```
{fn COT(numeric-expression)}
```

## 概要

COT は、ゼロ以外の数値を取り、そのコタンジェントを浮動小数点数で返します。NULL 値を渡すと、COT は NULL を返します。数値 0 (ゼロ) では実行時エラーが発生して、SQLCODE -400 (致命的なエラーが発生しました) が生成されます。COT は、非数値文字列を数値 0 (ゼロ) として扱います。

COT は、有効桁数が 36 で小数桁数が 18 の値を返します。

COT は、{} 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

## 引数

### *numeric-expression*

数値式。ラジアンで表示される角度です。

COT は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。COT は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、COT の実行結果です。

### SQL

```
SELECT {fn COT(0.52)} AS Cotangent
```

これは、1.74653 を返します。

### SQL

```
SELECT {fn COT(124.1332)} AS Cotangent
```

これは、-0.040312 を返します。

## 関連項目

- SQL 関数: [ACOS](#)、[ASIN](#)、[ATAN](#)、[COS](#)、[SIN](#)、[TAN](#)
- ObjectScript 関数: [\\$ZCOT](#)

## CURDATE (SQL)

現在のローカル日付を返す、スカラ日付/時刻関数です。

### 構文

```
{fn CURDATE()}  
{fn CURDATE}
```

### 概要

CURDATE は、引数を取りません。現在のローカル日付を DATE データ型として返します。引数の括弧はオプションです。CURDATE は、このタイムゾーンの現在のローカル日付を返します。これはサマータイムなどのローカル時刻調整に合わせて調整されます。

論理モードの CURDATE は、\$HOROLOG 形式で現在のローカル日付を返します (例:64701)。表示モードの CURDATE は、そのロケールの既定形式で現在のローカル日付を返します。例えば、アメリカのロケールでは 02/22/2018、ヨーロッパのロケールでは 22/02/2018、ロシアのロケールでは 22.02.2018 となります。

異なる日付形式を指定するには、TO\_DATE 関数を使用します。既定の日付形式を変更するには、DATE\_FORMAT、YEAR\_OPTION、または DATE\_SEPARATOR オプションで SET OPTION コマンドを使用します。

現在の日付のみを返すには、CURDATE または CURRENT\_DATE を使用します。これらの関数は、その値を DATE データ型で返します。現在の日付と時間を TIMESTAMP データ型で返す場合は、CURRENT\_TIMESTAMP 関数、GETDATE 関数、および NOW 関数を使用できます。

InterSystems SQL の時刻関数および日付関数は、GETUTCDATE を除いてすべて、それぞれのローカル・タイム・ゾーン設定に固有です。タイム・ゾーンに依存しない世界時による現在のタイムスタンプを取得するには、GETUTCDATE または ObjectScript の \$ZTIMESTAMP 特殊変数を使用します。

埋め込み SQL を使用するときは、これらのデータ型の動作が異なります。DATE データ型は \$HOROLOG 形式の整数として値を格納し、SQL で表示されるときは日付表示形式に変換され、埋め込み SQL から返されるときは整数として返されます。TIMESTAMP データ型は、同じ形式で値を格納および表示します。日付および時刻のデータ型は、CONVERT 関数を使用して変更できます。

### 例

以下の例は、ともに現在の日付を返します。

#### SQL

```
SELECT {fn CURDATE()} AS Today
```

#### SQL

```
SELECT {fn CURDATE} AS Today
```

以下の例は、現在の日付を返します。この日付は \$HOROLOG 形式で格納されるため、整数として返されます。

#### SQL

```
SELECT {fn CURDATE()} AS CurrentDate
```

以下の例は、現在の日付以降の出荷日を示すすべてのレコードを返すために、CURDATE が SELECT 文でどのように使用されるかを示します。

## SQL

```
SELECT * FROM Orders
  WHERE ShipDate >= {fn CURDATE()}
```

## 関連項目

- ・ SQL 関数: [CURRENT\\_DATE](#)、[CURRENT\\_TIME](#)、[CURRENT\\_TIMESTAMP](#)、[CURTIME](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)



## CURRENT\_DATE (SQL)

現在のローカル日付を返す、日付/時刻関数です。

### 構文

CURRENT\_DATE

### 概要

CURRENT\_DATE は、引数を取りません。現在のローカル日付を DATE データ型として返します。引数の括弧は使用しません。CURRENT\_DATE は、このタイムゾーンの現在のローカル日付を返します。これはサマータイムなどのローカル時刻調整に合わせて調整されます。

論理モードの CURRENT\_DATE は、\$HOROLOG 形式で現在のローカル日付を返します (例:64701)。表示モードの CURRENT\_DATE は、そのロケールの既定形式で現在のローカル日付を返します。例えば、アメリカのロケールでは 02/22/2018、ヨーロッパのロケールでは 22/02/2018、ロシアのロケールでは 22.02.2018 となります。

異なる日付形式を指定するには、TO\_DATE 関数を使用します。既定の日付形式を変更するには、DATE\_FORMAT、YEAR\_OPTION、または DATE\_SEPARATOR オプションで SET OPTION コマンドを使用します。

現在の日付のみを返すには、CURRENT\_DATE または CURDATE を使用します。これらの関数は、その値を DATE データ型で返します。現在の日付と時間を TIMESTAMP データ型で返す場合は、CURRENT\_TIMESTAMP 関数、GETDATE 関数、および NOW 関数を使用できます。

InterSystems SQL の時刻関数および日付関数は、GETUTCDATE を除いてすべて、それぞれのローカル・タイム・ゾーン設定に固有です。タイム・ゾーンに依存しない世界時による現在のタイムスタンプを取得するには、GETUTCDATE または ObjectScript の \$ZTIMESTAMP 特殊変数を使用します。

埋め込み SQL を使用するとき、これらのデータ型の動作が異なります。DATE データ型は \$HOROLOG 形式の整数として値を格納し、SQL で表示されるときは日付表示形式に変換され、埋め込み SQL から返されるときは整数として返されます。TIMESTAMP データ型は、同じ形式で値を格納および表示します。日付および時刻のデータ型は、CONVERT 関数を使用して変更できます。

CURRENT\_DATE は、CREATE TABLE または ALTER TABLE の既定の仕様のキーワードとして使用されます。

### 例

以下の例では、表示モードに変換された現在の日付が返されます。

#### SQL

```
SELECT CURRENT_DATE AS Today
```

以下の例も現在の日付を返しますが、この日付は \$HOROLOG 形式で格納されるため、整数として返されます。

#### SQL

```
SELECT CURRENT_DATE
```

以下の例では、CURRENT\_DATE を WHERE 節で使用して過去 1000 日以内に生まれた人のレコードが返るようにする方法を示します。

#### SQL

```
SELECT Name, DOB, Age
FROM Sample.Person
WHERE DOB > CURRENT_DATE - 1000
```

## 関連項目

[CURDATE](#)、[CURRENT\\_TIME](#)、[CURRENT\\_TIMESTAMP](#)、[CURTIME](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)

## CURRENT\_TIME (SQL)

現在のローカル時刻を返す、日付/時刻関数です。

### 構文

```
CURRENT_TIME CURRENT_TIME(precision)
```

### 説明

CURRENT\_TIME は、このタイムゾーンの現在のローカル時刻を返します。これはサマータイムなどのローカル時刻調整に合わせて調整されます。

CURRENT\_TIME は、引数を取らないか、精度引数を取ります。空引数の括弧は使用できません。

論理モードの CURRENT\_TIME は、\$HOROLOG 形式で現在のローカル時刻を返します (例:37065)。表示モードの CURRENT\_TIME は、そのロケールの既定形式で現在のローカル時刻を返します (例:10:18:27)。

既定の時刻形式を変更するには、TIME\_FORMAT および TIME\_PRECISION オプションで SET OPTION コマンドを使用します。秒の小数部の精度は以下のように設定できます。

現在の時刻のみを返すには、CURRENT\_TIME または CURTIME を使用します。これらの関数は、その値を TIME データ型で返します。現在の日付と時間を TIMESTAMP データ型で返す場合は、CURRENT\_TIMESTAMP 関数、GETDATE 関数、および NOW 関数を使用できます。

InterSystems SQL の時刻関数および日付関数は、GETUTCDATE を除いてすべて、それぞれのローカル・タイム・ゾーン設定に固有です。タイム・ゾーンに依存しない世界時による現在のタイムスタンプを取得するには、GETUTCDATE または ObjectScript の \$ZTIMESTAMP 特殊変数を使用します。

埋め込み SQL を使用するときは、これらのデータ型の動作が異なります。TIME データ型は \$HOROLOG 形式の整数 (午前 0 時 00 分からの秒数) として値を格納し、SQL で表示されるときは時刻表示形式に変換され、埋め込み SQL から返されるときは整数として返されます。TIMESTAMP データ型は、同じ形式で値を格納および表示します。日付および時刻のデータ型は、CAST または CONVERT 関数を使用して変更できます。

CURRENT\_TIME は、CREATE TABLE または ALTER TABLE の既定の仕様のキーワードとして使用されます。CURRENT\_TIME が既定の仕様のキーワードとして使用されている場合、precision 引数を指定することはできません。

### 秒の小数部の精度

CURRENT\_TIME は秒の小数部の精度について小数点以下 9 桁までの値を返すことができます。精度桁数の既定は以下の方法で構成できます。

- SET OPTION で TIME\_PRECISION オプションを使用します。
- システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DefaultTimePrecision。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[Default time precision] が表示されます。既定値は 0 です。
- 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GETDATE()、CURRENT\_TIME、および CURRENT\_TIMESTAMP の既定の時間精度] の現在の設定を表示して編集します。

返される精度の小数点以下の桁数の既定値に 0 ~ 9 の整数を指定します。既定値は 0 です。実際に返される精度はプラットフォームに依存し、システムで使用可能な精度を超えた精度の桁はゼロとして返されます。

### 例

以下の例は、現在のシステム時刻を返します。

## SQL

```
SELECT CURRENT_TIME
```

以下の例は、3 桁の秒の小数部の精度で現在のシステム時刻を返します。

## SQL

```
SELECT CURRENT_TIME(3)
```

以下の埋め込み SQL の例は、現在の時刻を返します。この時刻は \$HOROLOG 形式で格納されるため、整数として返されます。

## ObjectScript

```
&sql(SELECT CURRENT_TIME INTO :a)
IF $SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"Current time is: ",a }
```

以下の例は、Contacts テーブルの指定された行の LastCall フィールドに、現在のシステム時刻を設定します。

## SQL

```
UPDATE Contacts SET LastCall = CURRENT_TIME
WHERE Contacts.ItemNumber=:item
```

## 関連項目

- SQL の概念: [データ型](#)、[日付/時刻文](#)
- SQL 時刻関数: [CAST](#)、[CONVERT](#)、[CURTIME](#)、[HOUR](#)、[MINUTE](#)、[SECOND](#)
- SQL タイムスタンプ関数: [CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)、[SYSDATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)
- InterSystems IRIS ObjectScript : [\\$ZTIME](#) 関数、[\\$HOROLOG](#) 特殊変数、[\\$ZTIMESTAMP](#) 特殊変数

## CURRENT\_TIMESTAMP (SQL)

現在のローカルな日付と時刻を返す日付/時刻関数です。

### 構文

```
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP(precision)
```

### 引数

引数	説明
<i>precision</i>	時刻の精度を秒の小数部の桁数として指定する正の整数。既定値は 0 (秒の小数部なし) です。この既定値は構成可能です。

CURRENT\_TIMESTAMP は、TIMESTAMP [データ型](#)を返します。

### 概要

CURRENT\_TIMESTAMP は、引数を取らないか、精度引数を取ります。空引数の括弧は使用しません。

CURRENT\_TIMESTAMP は、この[タイムゾーン](#)の現在のローカル日付とローカル時刻を返します。これは[サマータイム](#)などのローカル時刻調整に合わせて調整されます。

CURRENT\_TIMESTAMP は、%TimeStamp データ型形式 (yyyy-mm-dd hh:mm:ss.fff) または %PosixTime データ型形式 (エンコードされた 64 ビットの符号付き整数) のいずれかでタイムスタンプを返すことができます。返されるタイムスタンプ形式は以下のルールによって決まります。

- 現在のタイムスタンプがデータ型 %PosixTime のフィールドに対して指定されている場合、現在のタイムスタンプ値は POSIXTIME データ型形式で返されます。例として、WHERE PosixField=CURRENT\_TIMESTAMP や INSERT INTO MyTable (PosixField) VALUES (CURRENT\_TIMESTAMP) が挙げられます。
- 現在のタイムスタンプがデータ型 %TimeStamp のフィールドに対して指定されている場合、現在のタイムスタンプ値は TIMESTAMP データ型形式で返されます。例として、WHERE TSField=CURRENT\_TIMESTAMP や INSERT INTO MyTable (TSField) VALUES (CURRENT\_TIMESTAMP) が挙げられます。
- 現在のタイムスタンプがコンテキストなしで指定されている場合、現在のタイムスタンプ値は TIMESTAMP データ型形式で返されます。例として、SELECT CURRENT\_TIMESTAMP が挙げられます。

\$HOROLOG を使用すると、内部形式で現在のローカル日付とローカル時刻を保存して返すようにすることができます。

既定の日付/時刻形式を変更するには、各種日付/時刻オプションで [SET OPTION](#) コマンドを使用します。

CURRENT\_TIMESTAMP は、CREATE TABLE または ALTER TABLE を使用して日付/時刻フィールドを定義するときに、precision を付けても付けなくても[フィールドの既定値](#)として指定できます。CURRENT\_TIMESTAMP は、データ型 %Library.PosixTime または %Library.TimeStamp のフィールドの既定値として指定できます。現在の日付と時刻は、フィールドのデータ型によって指定された形式で格納されます。

### 秒の小数部の精度

CURRENT\_TIMESTAMP には、次の 2 つの構文形式があります。

- 引数の括弧がない場合、CURRENT\_TIMESTAMP は [NOW](#) と機能的に同じになります。システム全体の既定の時刻精度を使用します。

- ・ 引数の括弧がある場合、CURRENT\_TIMESTAMP(precision) は、CURRENT\_TIMESTAMP() precision 引数が必須である点を除いて、GETDATE と機能的に同じになります。CURRENT\_TIMESTAMP() は常に、指定された precision を返し、構成されたシステム全体の既定の時刻制度を無視します。

秒の小数部は常に、丸められずに、指定された精度に切り捨てられます。

- ・ TIMESTAMP データ型形式では、精度の最大桁数は 9 桁です。サポートされる実際の桁数は、precision 引数、構成されている既定の時刻精度、およびシステム機能によって決まります。構成されている既定の時刻制度よりも大きい precision が指定されている場合、追加の精度桁数は末尾の 0 として返されます。
- ・ POSIXTIME データ型形式では、精度の最大桁数は 6 桁です。POSIXTIME 値はすべて 6 桁の精度を使用して計算されます。指定されていない場合、これらの小数桁数は既定で 0 桁になります。サポートされる実際の桁数 (0 以外) は、precision 引数、構成されている既定の時刻精度、およびシステム機能によって決まります。

## 精度の構成

既定の精度は、以下の方法で構成できます。

- ・ [SET OPTION](#) で TIME\_PRECISION オプションを使用します。
- ・ システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DefaultTimePrecision。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[ ] が表示されます。既定値は 0 です。
- ・ 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GETDATE()、CURRENT\_TIME、および CURRENT\_TIMESTAMP の既定の時間精度] の現在の設定を表示して編集します。

返される精度の小数点以下の桁数の既定値に 0 ～ 9 の整数を指定します。既定値は 0 です。実際に返される精度はプラットフォームに依存し、システムで使用可能な精度を超えた precision の桁はゼロとして返されます。

## 日付/時刻関数の比較

[GETDATE](#) および [NOW](#) を使用して、現在のローカル日付とローカル時刻を TIMESTAMP データ型または POSIXTIME データ型の値として返すこともできます。GETDATE は精度をサポートし、NOW は精度をサポートしません。

[SYSDATE](#) は、SYSDATE が precision をサポートしない点を除いて、CURRENT\_TIMESTAMP と同じになります。CURRENT\_TIMESTAMP が優先される InterSystems SQL 関数で、SYSDATE は他のベンダとの互換性を保つために提供されます。

InterSystems SQL の時刻関数および日付関数は、[GETUTCDATE](#) を除いてすべて、それぞれのローカル・タイム・ゾーン設定に固有です。ユニバーサルな (タイム・ゾーンに依存しない) タイムスタンプを取得するには、TIMESTAMP データ型または POSIXTIME データ型の値としてユニバーサルな日付と時刻を返す [GETUTCDATE](#)、または ObjectScript [\\$ZTIMESTAMP](#) 特殊変数を使用できます。

現在のローカル日付のみを返すには、[CURDATE](#) または [CURRENT\\_DATE](#) を使用します。現在のローカル時刻のみを返すには、[CURRENT\\_TIME](#) または [CURTIME](#) を使用します。これらの関数は、その値を DATE または TIME データ型で返します。これらの関数は、いずれも精度をサポートしません。

TIMESTAMP データ型の格納形式と表示形式は同じです。POSIXTIME データ型の格納形式は、エンコードされた 64 ビットの符号付き整数です。TIME および DATE データ型は [\\$HOROLOG](#) 形式の整数として値を格納し、SQL で表示されるときは日付または時刻表示形式に変換されます。埋め込み SQL では既定で、論理 (格納) 形式で返されます。埋め込み SQL で返される値の形式は、[#sqlcompile select](#) マクロ・プリプロセッサ指示文を使用して変更できます。

日付および時刻のデータ型は、[CAST](#) または [CONVERT](#) 関数を使用して変更できます。

## 例

以下の例では、現在のローカル日付とローカル時刻が、システムの既定の時刻精度を使用する TIMESTAMP データ型形式、秒の小数部が 2 桁の precision を使用する TIMESTAMP データ型形式、および整数の秒数を使用する \$HOROLOG 内部ストレージ形式という 3 つの異なる方法で返されます。

### SQL

```
SELECT
    CURRENT_TIMESTAMP AS FullSecStamp,
    CURRENT_TIMESTAMP(2) AS FracSecStamp,
    $HOROLOG AS InternalFullSec
```

以下の例では、ローカルの既定の時刻精度が設定されています。最初の CURRENT\_TIMESTAMP では precision が指定されておらず、既定の時刻精度で現在の時刻が返されます。2 番目の CURRENT\_TIMESTAMP では precision が指定されており、構成されている既定の時刻精度がオーバーライドされます。precision 引数は、既定の時刻精度の設定よりも大きい値や小さい値にすることができます。

### SQL

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(2)
```

以下の例は、ローカル・タイムスタンプ (タイム・ゾーン依存) とユニバーサル・タイムスタンプ (タイム・ゾーン非依存) を比較します。

### SQL

```
SELECT CURRENT_TIMESTAMP, GETUTCDATE()
```

以下の例は、Orders テーブルの指定された行の LastUpdate フィールドに、現在のシステム日付と時刻を設定します。LastUpdate がデータ型 %TimeStamp である場合、CURRENT\_TIMESTAMP は現在の日付と時刻を ODBC タイムスタンプとして返します。LastUpdate がデータ型 %PosixTime である場合、CURRENT\_TIMESTAMP は現在の日付と時刻をエンコードされた 64 ビットの符号付き整数として返します。

### SQL

```
UPDATE Orders SET LastUpdate = CURRENT_TIMESTAMP
WHERE Orders.OrderNumber=:ord
```

以下の例は、Orders というテーブルを作成し、このテーブルには、受信した商品注文が記録されます。

### SQL

```
CREATE TABLE Orders (
    OrderId      INT NOT NULL,
    ClientId     INT,
    ItemName     CHAR(40) NOT NULL,
    OrderDate    TIMESTAMP DEFAULT CURRENT_TIMESTAMP(3),
    PRIMARY KEY (OrderId))
```

OrderDate 列には、注文を受信した日付と時刻が含まれます。TIMESTAMP データ型を使用し、有効桁数を 3 とする CURRENT\_TIMESTAMP 関数を使用して、現在のシステム日付と時刻を既定値として挿入します。

## 関連項目

- SQL の概念 : [データ型](#)、[日付/時刻文](#)
- SQL タイムスタンプ関数 : [CAST](#)、[CONVERT](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)、[SYSDATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)、[TO\\_POSIXTIME](#)、[TO\\_TIMESTAMP](#)
- SQL 現在日時関数 : [CURDATE](#)、[CURRENT\\_DATE](#)、[CURRENT\\_TIME](#)、[CURTIME](#)



- ・ ObjectScript : [\\$ZDATETIME](#) 関数、[\\$HOROLOG](#) 特殊変数、[\\$ZTIMESTAMP](#) 特殊変数

## CURTIME (SQL)

現在のローカル時刻を返す、スカラ日付/時刻関数です。

### 構文

```
{fn CURTIME()}
{fn CURTIME}
```

### 概要

CURTIME は、現在のローカル時刻を TIME データ型として返します。引数は取りません。引数の括弧はオプションです。CURTIME は、このタイムゾーンの現在のローカル時刻を返します。これはサマータイムなどのローカル時刻調整に合わせて調整されます。

論理モードの CURTIME は、\$HOROLOG 形式で現在のローカル時刻を返します (例:37065)。表示モードの CURTIME は、そのロケールの既定形式で現在のローカル時刻を返します (例:10:18:27)。

時間は 24 時間形式で表示されます。

既定の時刻形式を変更するには、TIME\_FORMAT および TIME\_PRECISION オプションで SET OPTION コマンドを使用します。

現在の時刻のみを返すには、CURTIME または CURRENT\_TIME を使用します。これらの関数は、その値を TIME データ型で返します。現在の日付と時間を TIMESTAMP データ型で返す場合は、CURRENT\_TIMESTAMP 関数、GETDATE 関数、および NOW 関数を使用できます。

InterSystems SQL の時刻関数および日付関数は、GETUTCDATE を除いてすべて、それぞれのローカル・タイム・ゾーン設定に固有です。タイム・ゾーンに依存しない世界時による現在のタイムスタンプを取得するには、GETUTCDATE または ObjectScript の \$ZTIMESTAMP 特殊変数を使用します。

埋め込み SQL を使用するときは、これらのデータ型の動作が異なります。TIME データ型は \$HOROLOG 形式の整数 (午前 0 時 00 分からの秒数) として値を格納し、SQL で表示されるときは時刻表示形式に変換され、埋め込み SQL から返されるときは整数として返されます。TIMESTAMP データ型は、同じ形式で値を格納および表示します。時刻および日付のデータ型は、CAST または CONVERT 関数を使用して変更できます。

### 例

以下の例は、ともに現在のシステム時刻を返します。

#### SQL

```
SELECT {fn CURTIME()} AS TimeNow
```

#### SQL

```
SELECT {fn CURTIME} AS TimeNow
```

以下の埋め込み SQL の例は、現在の時刻を返します。この時刻は \$HOROLOG 形式で格納されるため、整数として返されます。

#### ObjectScript

```
&sql(SELECT {fn CURTIME} INTO :a)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"Current time is: ",a }
```

以下の例は、Contacts テーブルの指定された行の LastCall フィールドに、現在のシステム時刻を設定します。

## SQL

```
UPDATE Contacts Set LastCall = {fn CURTIME()}  
WHERE Contacts.ItemNumber=:item
```

## 関連項目

- ・ SQL の概念 : [データ型](#) [日付/時刻文](#)
- ・ SQL 時刻関数 : [CAST](#) [CONVERT](#) [CURRENT\\_TIME](#) [HOUR](#) [MINUTE](#) [SECOND](#)
- ・ SQL タイムスタンプ関数 : [CURRENT\\_TIMESTAMP](#) [GETDATE](#) [GETUTCDATE](#) [NOW](#) [TIMESTAMPADD](#) [TIMESTAMPDIFF](#)
- ・ ObjectScript : [\\$ZTIME](#) 関数 [\\$HOROLOG](#) 特殊変数 [\\$ZTIMESTAMP](#) 特殊変数

## DATABASE

---

データベース名の修飾子を返すスカラ文字列関数です。

### 構文

```
{fn DATABASE()}
```

### 概要

DATABASE は、接続ハンドルに対応するデータベース名の現在の修飾子を返します。InterSystems IRIS では、DATABASE は常に空文字列 (') を返します。

---

# DATALENGTH (SQL)

---

式の文字の数を返す関数です。

## 構文

`DATALENGTH(expression)`

## 説明

DATALENGTH は、式に使用する文字の数を返します。

DATALENGTH 関数、CHAR\_LENGTH 関数、および CHARACTER\_LENGTH 関数は同一です。CHAR\_LENGTH 関数の使用は、新しいコードに推奨されます。DATALENGTH は、TSQL の互換性を持たせるために提供されています。詳細は "[CHAR\\_LENGTH](#)" を参照してください。

## 引数

### `expression`

列の名前、文字列リテラル、他のスカラー関数の結果などを表すことができる式。基本となるデータ型は (CHAR や VARCHAR など) 任意の文字タイプ、数値、またはデータ・ストリームとすることができます。

DATALENGTH は、INTEGER [データ型](#)を返します。

## 関連項目

- ・ [CHAR\\_LENGTH](#) 関数
- ・ [CHARACTER\\_LENGTH](#) 関数

## DATE (SQL)

タイムスタンプを取得して、日付を返す関数です。

### 構文

```
DATE(timestamp)
```

### 説明

- DATE(*timestamp*) はタイムスタンプの式を取得し、データ型が DATE の日付を返します。この関数は `CAST(timestamp as DATE)` と同等です。

以下の文は、タイムスタンプ文字列を日付に変換します。タイムスタンプの時刻部分は検証されますが、返されません。

#### SQL

```
SELECT DATE('2000-01-01 00:00:00') AS StringToDate -- Display Mode: 01/01/2000
```

例：さまざまな形式のタイムスタンプの日付への変換

### 引数

#### *timestamp*

タイムスタンプ、日付、または日時の表現を指定する式です。以下のいずれかのデータ型クラスとして *timestamp* を指定します。

- %Library.TimeStamp
- %Library.PosixTime
- %Library.Date
- %Library.Integer
- 暗黙的な論理タイムスタンプ形式の %Library.Numeric の値 (+\$HOROLOG など)。“[数値タイムスタンプ](#)”を参照してください。
- %Library.TimeStamp と互換性のある %Library.String の値。“[文字列タイムスタンプ](#)”を参照してください。

#### 数値タイムスタンプ

特殊変数 \$HOROLOG および \$ZTIMESTAMP は、現在の日付を数値文字列として指定します。DATE は、このような文字列を、1840 年 12 月 31 日を表す 0 にキャストします。

#### SQL

```
SELECT
  DATE($HOROLOG),      -- Returns 0 (12/31/1840)
  DATE($ZTIMESTAMP) -- Returns 0 (12/31/1840)
```

\$HOROLOG または \$ZTIMESTAMP を現在の日付として解釈するには、日付の前にプラス (+) 符号を付加することによって数値として解釈されるようにする必要があります。

## SQL

```
SELECT
  DATE(+$HOROLOG),    -- Returns the current date
  DATE(+$ZTIMESTAMP) -- Returns the current date
```

### 文字列タイムスタンプ

DATE 形式に変換した文字列タイムスタンプには、**%Library.TimeStamp** データ型との互換性が必要です。このデータ型で ODBC 日付形式の文字列を格納し、この形式に基づいて DATE 関数で入力文字列を検証します。文字列の検証で問題がなければ、DATE から対応する日付が返されます。検証が失敗すると、DATE からは 1840 年 12 月 31 日に相当する 0 が返されます。空の文字列 (") を指定した場合も 0 が返されます。NULL 引数は NULL を返します。

DATE では以下の検証が実行されます。

- ・ 文字列が ODBC 形式に対応していること。fff は秒の小数部です。

```
yyyy-mm-dd hh:mm:ss.fff
```

- ・ 文字列に少なくとも完全な日付 yyyy-mm-dd が指定されていること。時刻部分は省略できます。DATE では指定した時刻が検証されますが、その時刻は返されません。任意の時刻を指定できます。例えば、「yyyy-mm-dd hh:」と入力します。
- ・ 文字列に無効な形式の文字や文字列の末尾に文字が使用されていないこと。先頭のゼロは省略しても指定してもかまいません。
- ・ オプションの時刻部分も含め、文字列の数値要素に指定された範囲の有効な値が各要素に使用されていること。以下に例を挙げます。
  - 月の値が 1 ～ 12 の範囲にあること。
  - 日付の値が、うるう年も考慮して、指定された月の日付範囲にあること。
  - 日付が 0001-01-01 から 9999-12-31 の範囲にあること。

## 例

### さまざまな形式のタイムスタンプの日付への変換

以下の文は、データ型 **%Library.TimeStamp** のタイムスタンプを DATE データ型に変換します。

## SQL

```
SELECT
  {fn NOW} AS NowCol,
  DATE({fn NOW}) AS DateCol
```

## SQL

```
SELECT
  CURRENT_TIMESTAMP AS TSCol,
  DATE(CURRENT_TIMESTAMP) AS DateCol
```

## SQL

```
SELECT
  GETDATE() AS GetDateCol,
  DATE(GETDATE()) AS DateCol
```

以下の文は、**%Library.TimeStamp** 形式で記述された文字列を DATE に変換します。



## SQL

```
SELECT
  '2022-05-22 13:14:23' AS DateStrCol,
  DATE('2022-05-22 13:14:23') AS DateCol
```

以下の文は、%Library.PosixTime タイムスタンプを DATE に変換します。

## SQL

```
SELECT
  TO_POSIXTIME('2022-05-22', 'YYYY-MM-DD') AS PosixCol,
  DATE(TO_POSIXTIME('2022-05-22', 'YYYY-MM-DD')) AS DateCol
```

以下の文は、InterSystems IRIS® の論理形式で日付を表現した文字列値を DATE に変換します。この文字列値を正しく変換するには、文字列の前にプラス符号 (+) を付加することにより、数値が評価されるようにする必要があります。

## SQL

```
SELECT
  $H AS HoroCol,
  DATE(+ $H) AS DateCol
```

## SQL

```
SELECT
  $ZTIMESTAMP AS TSCol,
  DATE(+ $ZTIMESTAMP) AS DateCol
```

## 別の方法

コードを使用して ObjectScript でタイムスタンプと日付を上記と同様に変換するには DATE() メソッドを使用します。

## ObjectScript

```
WRITE $SYSTEM.SQL.Functions.DATE("2018-02-23 12:37:45")
```

## 関連項目

- ・ [CAST](#) 関数
- ・ [CURDATE](#) および [CURRENT\\_DATE](#) 関数
- ・ [CURRENT\\_TIMESTAMP](#) 関数
- ・ [GETUTCDATE](#) 関数
- ・ [NOW](#) 関数
- ・ [TO\\_TIMESTAMP](#) 関数
- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数

# DATEADD (SQL)

日付やタイムスタンプに対して日付部分の単位の数値（時間数や日数など）を加算または減算して計算されたタイムスタンプを返す日付/時刻関数です。

## 構文

```
DATEADD(datePart, numUnits, date)
```

## 説明

- DATEADD(**datePart**,**numUnits**,**date**) は、指定された日付部分を指定された単位の値でインクリメントすることで、日付または時刻の式を変更します。マイナスの単位を指定すると、DATEADD によってその単位の値で日付がデクリメントされます。
  - date が **%Library.PosixTime**（エンコードされた 64 ビットの符号付き整数）である場合、DATEADD からはデータ型 **%Library.PosixTime** のタイムスタンプが返されます。
  - date が他の型である場合、DATEADD からはデータ型 **%Library.TimeStamp** が yyyy-mm-dd hh:mm:ss.fff の形式で返されます。

以下の文は、現在の日付を 5 か月インクリメントします。

### SQL

```
SELECT DATEADD('month', 5, CURRENT_DATE)
```

DATEADD は、Sybase および Microsoft SQL Server と互換性があります。

## 引数

### datePart

日付または時刻の部分のフルネームまたは省略した名前です。datePart は大文字でも小文字でも指定できます。埋め込み SQL では、datePart をリテラル値またはホスト変数で指定します。日付と時刻の部分の有効な名前と省略形を以下の表に示します。また、その部分の 1 単位 (**numUnits** = 1) による日付のインクリメント量も示します。

名前	省略形	numUnits = 1
year	YYYY、YY	年に 1 をインクリメント。
quarter	qq、q	月に 3 をインクリメント。
month	mm、m	月に 1 をインクリメント。
week	wk、ww	日に 7 をインクリメント。
weekday	dw、w	日に 1 をインクリメント。
day	dd、d	日に 1 をインクリメント。
dayofyear	dy、y	日に 1 をインクリメント。
hour	hh、h	時間に 1 をインクリメント。
minute	mi、n	分に 1 をインクリメント。
second	ss、s	秒に 1 をインクリメント。

名前	省略形	numUnits = 1
millisecond	ms	秒を 0.001 インクリメント (precision of 3)
microsecond	mcs	秒を 0.000001 インクリメント (precision of 6)
nanosecond	ns	秒を 0.000000001 インクリメント (precision of 9)

日付部分をインクリメントまたはデクリメントすると、他の日付部分も適切に変更されます。例えば、時間のインクリメント結果が午前 0 時を超える場合、自動的に日がインクリメントされます。同様に、順次、月などがインクリメントされます。

DATEADD の動作は、datePart の指定で引用符を使用しているかどうかによってわずかに異なります。

- 引用符を使用している場合：DATEADD('month', 1, '2022-02-25')：datePart がリテラルとして扱われます。クエリを処理するときにリテラル置換が実行され、文字列 'month' が入力パラメータに置き換えられます。これにより、より一般的に使用できるクエリ・キャッシュが生成されます。
- 引用符を使用していない場合：DATEADD(month, 1, '2022-02-25')：datePart がキーワードとして扱われます。クエリの処理ではリテラル置換が実行されません。これにより、固有性が高いクエリ・キャッシュが生成されます。

datePart に無効なリテラル値を指定すると、SQLCODE -8 エラー・コードが発生します。一方、無効な datePart 値をホスト変数として指定すると、SQLCODE エラーは発行されず、datePart を解析するために呼び出された DATEPART 関数から NULL 値が返されます。

## numUnits

date に対する加算または減算に適用する datePart 単位の値です。数値で指定します。DATEADD では numUnits が整数に切り捨てられます。numUnits に数値部分がない場合、またはその先頭が数値ではない場合、DATEADD によってこの値が 0 に切り詰められ、最初に指定した date が返されます。

## date

加算または減算される日付または時刻の式です。InterSystems IRIS® の以下のいずれかのデータ型で指定します。

- %Date 論理値 (+\$H)。\$HOROLOG 形式とも呼ばれます。
- %PosixTime (%Library.PosixTime) 論理値 (エンコードされた 64 ビットの符号付き整数)。
- %TimeStamp (%Library.TimeStamp) 論理値 (YYYY-MM-DD HH:MM:SS.FFF)。ODBC 形式とも呼ばれます。
- %String または文字列互換の値。以下のいずれかの形式とします。

テーブル G-1: \$HOROLOG の日付と時刻の形式

形式	例
dddddd	SELECT DATEADD('yy', 1, '66716')
dddddd,sssss	SELECT DATEADD('yy', 1, '66716.256')
dddddd,sssss,fff	SELECT DATEADD('yy', 1, '66716,256.467')

各要素の内容は以下のとおりです。

- dddddd：1840 年 12 月 31 日からの経過日数 (整数値)。
- sssss：指定日の開始からの経過秒数 (整数値)。

- 桁：秒の小数部（整数値）。秒の小数部を指定した場合は、DATEADD の返り値にも秒の小数部が記述されます。

テーブル G-2: 日付形式

形式	例
MM/DD/YY	SELECT DATEADD('year',1,'12/31/99')
MM/DD/YYYY	SELECT DATEADD('year',1,'8/24/2022')
MM-DD-YY	SELECT DATEADD('year',1,'12-31-99')
MM-DD-YYYY	SELECT DATEADD('year',1,'8-24-2022')
MM.DD.YY	SELECT DATEADD('year',1,'12.31.99')
MM.DD.YYYY	SELECT DATEADD('year',1,'8.24.2022')
Mmm DD YY	SELECT DATEADD('year',1,'Dec 30 92')
Mmm DD YYYY	SELECT DATEADD('year',1,'January 23 2021')
Mmm DD, YY	SELECT DATEADD('year',1,'Dec 30, 92')
Mmm DD, YYYY	SELECT DATEADD('year',1,'January 23, 2021')

各要素の内容は以下のとおりです。

- MM：2 桁の月。
- DD：月の 2 桁の日数。
- Mmm：スペルアウト表記の月名。最低 3 文字（例：Mar）から完全な月名（例：March）まで指定できます。
- YY は 2 桁形式、YYYY は 4 桁形式の年です。

date は日付と時刻の文字列を組み合わせで指定できます。以下に例を示します。

## SQL

```
SELECT DATEADD('hh',1,'12/22/2021 8:15:23')
```

日付を指定せずに時刻を指定すると、DATEADD では既定で日付が 01/01/1900 になります。

テーブル G-3: 時刻形式

形式	例
HH:	SELECT DATEADD('hour',1,'10:')
HH:MM	SELECT DATEADD('mi',1,'10:30')
HH:MM:SS	SELECT DATEADD('ss',1,'10:30:59')
HH...SS.FFF	SELECT DATEADD('ms',1,'10:30:59.245')
HH...[AM PM]	SELECT DATEADD('mi',1,'10:30PM')

各要素の内容は以下のとおりです。

- HH : 1 日での 2 桁の時間値。
- MM : 1 時間での 2 桁の分の値。
- SS : 1 分間での 2 桁の秒の値。
- FFF : 秒の小数部の値。

date は日付と時刻の文字列を組み合わせで指定できます。以下に例を示します。

## SQL

```
SELECT DATEADD('hh',1,'12/22/2021 8:15:23')
```

時刻を指定せずに日付を指定すると、DATEADD では既定で時刻が 00:00:00 になります。

date 引数の制限事項と動作は以下のとおりです。

- ・ 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。年は 4 桁で指定される必要があります。入力値の日付部分を省略すると、DATEADD では既定で日付が '1900-01-01' になります。
- ・ 日付と時刻の値は、以下の有効な範囲内にいる必要があります。
  - 年 - 0001 から 9999
  - 月 - 1 から 12
  - 日 - 1 から 31
  - 時間 - 00 から 23
  - 分 - 0 から 59
  - 秒 - 0 から 59

月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。

- ・ 10(月および日)未満の日付値では、先頭のゼロを省略してもかまいません。その他の非標準的な整数値は許可されません。例えば、Day の値として '07' と '7' は有効ですが、'007'、'7.0'、'7a' は無効です。
- ・ 時刻値は省略してもかまいません。date に不完全な時刻を指定すると、指定していない部分には 0 が使用されます。
- ・ 10 よりも小さい時間値の先頭の 0 は記載する必要があります。

## 例

### さまざまな時間単位の日付への加算

以下の文は、指定された日付に 1 週を加算します。1 週を加算は 7 日の加算になるので、2022-03-05 00:00:00 が返されます。DATEADD では時刻部分が省略されます。

## SQL

```
SELECT DATEADD('week',1,'2022-02-26') AS NewDate
```

以下の文は、指定されたタイムスタンプに 5 か月を加算し、2022-04-26 12:00:00 を返します。5 か月を加算すると年もインクリメントされるので、DATEADD では月と年の両方が変更されます。

## SQL

```
SELECT DATEADD(MM,5,'2021-11-26 12:00:00') AS NewDate
```

以下の文も、タイムスタンプに 5 か月を加算し、2021-06-30 12:00:00 を返します。月のみをインクリメントすると無効な日付である 6 月 31 日になるので、DATEADD では日と月の両方が変更されます。

## SQL

```
SELECT DATEADD('mm',5,'2021-01-31 12:00:00') AS NewDate
```

以下の文は、タイムスタンプに 45 分を加算し、2022-02-26 12:45:00 を返します。

## SQL

```
SELECT DATEADD(MI,45,'2022-02-26 12:00:00') AS NewTime
```

以下の文も、タイムスタンプに 45 分を加算しますが、この場合は日がインクリメントされ、さらに月もインクリメントされます。その結果、2022-03-01 00:15:00 が返されます。

## SQL

```
SELECT DATEADD('mi',45,'2022-02-28 23:30:00') AS NewTime
```

以下の文は元のタイムスタンプを 45 分デクリメントし、2021-12-31 23:25:00 を返します。

## SQL

```
SELECT DATEADD(N,-45,'2022-01-01 00:10:00') AS NewTime
```

以下の文は、現在の日付に 60 日を加算し、各月の総日数に合わせて結果を調整します。

## SQL

```
SELECT DATEADD(D,60,CURRENT_DATE) AS NewDate
```

以下の文で最初の DATEADD は、指定された日に 92 日を加算し、2022-03-22 00:00:00 を返します。2 番目の DATEADD は、指定された日付に 1 四半期を加算し、2022-03-20 00:00:00 を返します。四半期のインクリメントを実行すると、月フィールドが 3 インクリメントされます。必要に応じて、DATEADD は年フィールドもインクリメントし、指定された月の総日数に合わせて結果を調整します。

## SQL

```
SELECT DATEADD('dd',92,'2021-12-20') AS NewDateD,  
       DATEADD('qq',1,'2021-12-20') AS NewDateQ
```

上記の文はすべて日付部分に省略形を使用していますが、完全な名前でも指定することもできます。例えば、以下の文は日付に 92 日を加算し、2022-03-22 00:00:00 を返します。

## SQL

```
SELECT DATEADD('day',92,'2021-12-20') AS NewDate
```

以下の埋め込み SQL コードは、ホスト変数を使用して前述の SQL 文と同じ DATEADD 処理を実行します。

## ObjectScript

```
set datePart = "day"  
set numUnits = 92  
set dateIn = "2021-12-20"  
  
&sql(SELECT DATEADD(:datePart,:numUnits,:dateIn) INTO :dateOut)  
  
write "in: ",dateIn,!, "out: ",dateOut
```

## 別の方法

[TIMESTAMPADD](#) ODBC スカラ関数を使用して、時刻と日付を変更できます。

ObjectScript コードで同様にタイムスタンプを変換するには `DATEADD()` メソッドを使用します。

```
$SYSTEM.SQL.Functions.DATEADD(datePart,numUnits,date)
```

## 関連項目

- ・ [DATEDIFF](#) 関数
- ・ [DATENAME](#) 関数
- ・ [DATEPART](#) 関数
- ・ [TIMESTAMPADD](#) 関数
- ・ [TIMESTAMPDIFF](#) 関数



# DATEDIFF (SQL)

2 つの日付の間に指定された datepart のために整数の差を返す、日付/時刻関数です。

## 構文

`DATEDIFF(datePart, startDate, endDate)`

## 説明

- DATEDIFF([datePart](#),[startDate](#),[endDate](#)) は、指定された日付部分 (秒、日、週など) に開始日と終了日の差である INTEGER 値 (startDate - endDate) を返します。endDate が startDate より前の場合、DATEDIFF は負の INTEGER 値を返します。

以下の文では、2 つのタイムスタンプ間に 353 日の差 (D) があるので 353 を返します。

### SQL

```
SELECT DATEDIFF(D, '2022-01-01 00:00:00', '2022-12-20 12:00:00')
```

例：日付間の差異の計算

DATEDIFF は、Sybase および Microsoft SQL Server と互換性があります。[TIMESTAMPDIFF](#) ODBC スカラ関数を使用して、同様に時刻と日付の比較を実行できます。

## 引数

### datePart

日付または時刻の部分の名前または省略名です。datePart は大文字でも小文字でも指定できます。埋め込み SQL では、datePart をリテラル値またはホスト変数で指定します。有効な日付と時刻の部分を下に示します。

名前	省略形
year	YYYY、YY
quarter	qq、q
month	mm、m
week	wk、ww
weekday	dw、w
day	dd、d
dayofyear	dy、y
hour	hh、h
minute	mi、n
second	ss、s
millisecond	ms
microsecond	mcs
nanosecond	ns

日付部分の weekday と dayofyear の値は day の値と機能的に同一です。

DATEDIFF では、四半期 (3 か月の期間) は処理されません。

開始日と終了日に秒の小数部も指定すると、DATEDIFF から秒の小数部の差が整数として返されます。以下に例を示します。

## SQL

```
SELECT DATEDIFF('ms','64701,56670.10','64701,56670.27'), /* returns 170 */
       DATEDIFF('ms','64701,56670.1111','64701,56670.27222') /* returns 161 */
```

DATEDIFF からは、startDate と endDate の小数桁数の精度に関係なく、秒の小数部がミリ秒 (3 桁の整数)、マイクロ秒 (6 桁の整数)、またはナノ秒 (9 桁の整数) として返されます。以下に例を示します。

DATEDIFF の動作は、datePart の指定で引用符を使用しているかどうかによってわずかに異なります。

- 引用符を使用している場合：DATEDIFF('month','2018-02-25',\$HOROLOGY)：datePart がリテラルとして扱われます。クエリを処理するときにリテラル置換が実行され、文字列 'month' が入力パラメータに置き換えられます。これにより、より一般的に使用できるクエリ・キャッシュが生成されます。
- 引用符を使用していない場合：DATEDIFF(month,'2018-02-25',\$HOROLOGY)：datePart がキーワードとして扱われます。クエリの処理ではリテラル置換が実行されません。これにより、固有性が高いクエリ・キャッシュが生成されます。

埋め込み SQL では、入力変数として無効な datePart を指定すると、SQLCODE -8 エラーが返されます。無効な datePart をリテラル値として指定すると、<SYNTAX> エラーが返されます。

ダイナミック SQL では、無効な datepart を指定すると、DATEDIFF から NULL が返されます。SQLCODE エラーは発行されません。

## startDate, endDate

DATEDIFF で差異の計算対象とする開始日付と終了日付。InterSystems IRIS® の以下のデータ型のいずれかとして指定します。

- %Date 論理値 (+\$H)。\$HOROLOG 形式とも呼ばれます。
- %PosixTime (%Library.PosixTime) 論理値 (エンコードされた 64 ビットの符号付き整数)。
- %TimeStamp (%Library.TimeStamp) 論理値 (YYYY-MM-DD HH:MM:SS.FFF)。ODBC 形式とも呼ばれます。
- %String または文字列互換の値。以下のいずれかの形式とします。

テーブル G-4: \$HOROLOG の日付と時刻の形式

形式	例
dddddd	SELECT DATEDIFF('yy','65726','66716')
dddddd,sssss	SELECT DATEDIFF('mi','65726,143','66716,256')
dddddd,sssss.fff	SELECT DATEDIFF('ns','65726,143.345','66716,256.467')

各要素の内容は以下のとおりです。

- dddddd：1840 年 12 月 31 日からの経過日数 (整数値)。
- sssss：指定日の開始からの経過秒数 (整数値)。

- 桁：秒の小数部 (整数値)。

テーブル G-5: 日付形式

形式	例
MM/DD/YY	SELECT DATEDIFF('yy', '11/25/80', '12/31/99')
MM/DD/YYYY	SELECT DATEDIFF('dd', '3/15/2017', '8/24/2022')
MM-DD-YY	SELECT DATEDIFF('yy', '11-25-80', '12-31-99')
MM-DD-YYYY	SELECT DATEDIFF('dd', '3-15-2017', '8-24-2022')
MM.DD.YY	SELECT DATEDIFF('yy', '11.25.80', '12.31.99')
MM.DD.YYYY	SELECT DATEDIFF('dd', '3.15.2017', '8.24.2022')
Mmm DD YY	SELECT DATEDIFF('ss', 'Sep 9 91', 'Dec 30 92')
Mmm DD YYYY	SELECT DATEDIFF('ss', 'October 10 2019', 'January 23 2021')
Mmm DD, YY	SELECT DATEDIFF('ss', 'Sep 9, 91', 'Dec 30, 92')
Mmm DD, YYYY	SELECT DATEDIFF('ss', 'October 10, 2019', 'January 23, 2021')

各要素の内容は以下のとおりです。

- MM：2 桁の月。
- DD：月の 2 桁の日数。
- Mmm：スペルアウト表記の月名。最低 3 文字 (例：Mar) から完全な月名 (例：March) まで指定できます。
- YY は 2 桁形式、YYYY は 4 桁形式の年です。

startDate および endDate は日付と時刻の文字列を組み合わせで指定できます。以下に例を示します。

## SQL

```
SELECT DATEDIFF('hh', '12/22/2021 8:15:23', '12/31/2021 10:30:23')
```

日付を指定せずに時刻を指定すると、DATEDIFF では既定で日付が 01/01/1900 になります。

テーブル G-6: 時刻形式

形式	例
HH:	SELECT DATEDIFF('hh','2:','10:')
HH:MM	SELECT DATEDIFF('mi','2:15','10:30')
HH:MM:SS	SELECT DATEDIFF('ss','2:15:23','10:30:59')
HH...SS:FFF	SELECT DATEDIFF('ms','2:15:23.335','10:30:59.245')
HH...SS.FFF	SELECT DATEDIFF('ms','2:15:23.335','10:30:59.245')
HH...[AM PM]	SELECT DATEDIFF('mi','2:15AM','10:30PM')

各要素の内容は以下のとおりです。

- HH : 1 日での 2 桁の時間値。
- MM : 1 時間での 2 桁の分の値。
- SS : 1 分間での 2 桁の秒の値。
- FFF : 秒の小数部の値。

startDate および endDate は日付と時刻の文字列を組み合わせで指定できます。以下に例を示します。

#### SQL

```
SELECT DATEDIFF('hh','12/22/2021 8:15:23','12/31/2021 10:30:23')
```

時刻を指定せずに日付を指定すると、DATEDIFF では既定で時刻が 00:00:00 になります。

startDate 引数と endDate 引数は異なるデータ型で指定できます。

startDate 引数と endDate 引数の制限事項と動作は以下のとおりです。

- ・ 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。年は 4 桁で指定される必要があります。入力値の日付部分を省略すると、DATEDIFF では既定で日付が '1900-01-01' になります。
- ・ 日付と時刻の値は、以下の有効な範囲内にある必要があります。
  - 年 - 0001 から 9999
  - 月 - 1 から 12
  - 日 - 1 から 31
  - 時間 - 00 から 23
  - 分 - 0 から 59
  - 秒 - 0 から 59

月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。無効な日付値を指定すると、SQLCODE -8 エラーになります。

- ・ 10(月および日)未満の日付値では、先頭のゼロを省略してもかまいません。その他の非標準的な整数値は許可されません。例えば、Day の値として '07' と '7' は有効ですが、'007'、'7.0'、'7a' は無効です。
- ・ 時刻値は省略してもかまいません。startDate または endDate に不完全な時刻を指定すると、指定していない部分には 0 が使用されます。
- ・ 10 よりも小さい時間値の先頭の 0 は記載する必要があります。この先頭の 0 を省略すると、SQLCODE -8 エラーになります。
- ・ 埋め込み SQL では、入力変数またはリテラルとして無効な startDate または endDate を指定すると、SQLCODE -8 エラーが返されます。
- ・ ダイナミック SQL では、無効な startDate または endDate を指定すると、DATEDIFF から NULL が返されます。SQLCODE エラーは発行されません。
- ・ 01 から 99 までの 2 桁の年は 1901 年から 1999 年までと見なされます。例えば、以下の文の startDate の年は 1914 年です。

## SQL

```
SELECT DATEDIFF('year','10/11/14','04/22/2022'),
       DATEDIFF('year','12:00:00','2022-04-22 12:00:00')
```

00 は 0000 年と見なされます。これは無効であり、エラーが返されます。

この日付をシステム全体で制御する既定のスライディング・ウィンドウを変更するには、%DATE ユーティリティを使用します。その方法は、<https://docs.intersystems.com/priordocexcerpts> の旧ドキュメントを参照してください。2 桁の年で指定した日付を解釈するためのスライディング・ウィンドウの設定については、\$ZDATE、\$ZDATEH、\$ZDATE-TIME、および \$ZDATETIMEH の各関数の説明を参照してください。

## 例

### 日付間の差異の計算

DATEDIFF は、startDate と endDate の間の合計日数を指定の単位で返します。例えば、以下の文は日付間の差を分数で計算します。また、日付と時刻の両方のコンポーネントが評価されます。1 日の差ごとに、1 日の分数である 1440 分が加算されます。

## SQL

```
SELECT DATEDIFF('mi','1910-08-21 08:32:04','1910-08-28 01:45:00')
```

DATEDIFF では、日付間の実際の期間が考慮されません。startDate と endDate の間に存在する指定の日付部分境界の数と見なされることがあります。例えば、連続する年の間のこうした差異すべてで、その期間が 365 日を超えていても 365 日未満であっても、DATEDIFF として 1 が返されます。

## SQL

```
SELECT DATEDIFF('yyyy','1910-08-21','1911-08-21') AS ExactYear,
       DATEDIFF('yyyy','1910-06-30','1911-01-01') AS HalfYear,
       DATEDIFF('yyyy','1910-01-01','1911-12-31') AS Nearly2Years,
       DATEDIFF('yyyy','1910-12-31 11:59:59','1911-01-01 00:00:00') AS NewYearSecond
```

同様に、以下の文では、2 つの値の間で実際の期間はわずか 6 秒ですが、連続する分の差異は 1 となります。

## SQL

```
SELECT DATEDIFF('mi','12:23:59','12:24:05') AS MinuteDiff
```

ここまでの文では日付部分に省略形を使用しましたが、完全名を指定することもできます。以下に例を示します。

## SQL

```
SELECT DATEDIFF('minute','12:23:59','12:24:05') AS MinuteDiff
```

以下の埋め込み SQL の例では、ホスト変数を使用して、前述の文と同じ DATEDIFF 処理を実行します。

## ObjectScript

```
set datePart="minute"
set startDate="12:23:59"
set endDate="12:24:05"

&sql(SELECT DATEDIFF(:datePart,:startDate,:endDate) INTO :diff)
WRITE diff, !
```

以下の文は、WHERE 節で DATEDIFF を使用して、先週入院した患者を選択します。

## SQL

```
SELECT Name,DateOfAdmission FROM Sample.Patients WHERE DATEDIFF(D,DateOfAdmission,$HOROLOG) <= 7
```

以下の文は、サブクエリを使用して、生年月日が現在の日付から 1500 日以内の人のレコードを返します。

## SQL

```
SELECT Name, Age, DOB
FROM (SELECT Name, Age, DOB, DATEDIFF('dy', DOB, $HOROLOG) AS DaysTo FROM Sample.Person)
WHERE DaysTo <= 1500
ORDER BY Age
```

## 時刻形式に依存しない時刻の差

DATEDIFF は、秒数が返されないように現在のプロセスの時刻形式を設定していても、秒単位とミリ秒単位で時刻の差を返します。以下に例を示します。

## ObjectScript

```
// Get current time format and set start and end times
set originalTimeFormat = ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
set startDate = "66211,34717.10"
set endDate = "66211,34720.27"

// Set time format that includes seconds (TimeFormat = 1)
do ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",1)
write "DATETIME (with seconds): ",!, $ZDATETIME(startDate,1,-1)," ", $ZDATETIME(endDate,1,-1),!
&sql(SELECT DATEDIFF('ss',:startDate,:endDate) INTO :diff)
write "DATEDIFF number of seconds: ",diff,!!

// Set time format that omits seconds (TimeFormat = 2)
do ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",2)
write "DATETIME (without seconds): ",!, $ZDATETIME(startDate,1,-1)," ", $ZDATETIME(endDate,1,-1),!
&sql(SELECT DATEDIFF('ss',:startDate,:endDate) INTO :diff)
write "DATEDIFF number of seconds: ",diff,!

// Revert to original time format
do ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",originalTimeFormat)
```

## 別の方法

ObjectScript コードでこの関数を呼び出すには、DATEDIFF() メソッドを使用します。

```
$SYSTEM.SQL.Functions.DATEDIFF(datePart,startDate,endDate)
```

無効な datepart を DATEDIFF() メソッドに指定すると、<ZDDIF> エラーが発生します。

## 関連項目

・ [DATEADD](#)

- DATENAME
- DATEPART
- TIMESTAMPADD
- TIMESTAMPDIFF



## DATENAME (SQL)

日付/時刻式の指定された部分の値を表す文字列を返す日付/時刻関数です。

### 構文

```
DATENAME (datepart, date-expression)
```

### 概要

DATENAME 関数は、日付/時刻値の指定された部分の名前（月名の "June" など）を返します。結果はデータ型 VARCHAR(20) として返されます。結果が数値である場合（日の数字 "23" など）でも、VARCHAR(20) 文字列として返されます。この情報を整数として返すには、[DATEPART](#) を使用します。複数の日付部分を含む文字列を返すには、[TO\\_DATE](#) を使用します。

DATENAME では、Sybase および Microsoft SQL Server との互換性が提供されています。

この関数は、ObjectScript から DATENAME() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.DATENAME (datepart, date-expression)
```

### Datepart 引数

datepart 引数は、以下の日付/時刻コンポーネントの 1 つ（1 つのみ）から成る文字列で、正式な名前（日付部分列）または省略形（省略形列）のいずれかになります。これらの datepart コンポーネント名と省略形では、大文字と小文字は区別されません。

日付部分	省略形	返り値
year	yyyy, yy	0001-9999
quarter	qq, q	1-4
month	mm, m	January, ..., December
week	wk, ww	1-53
weekday	dw, w	Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
dayofyear	dy, y	1-366
day	dd, d	1-31
hour	hh, h	0-23
minute	mi, n	0-59
second	ss, s	0-59
millisecond	ms	0-999 (3 桁の精度)
microsecond	mcs	0-999999 (6 桁の精度)
nanosecond	ns	0-999999999 (9 桁の精度)

リテラルとして無効な datepart 値を指定した場合は、SQLCODE -8 エラー・コードが発行されます。一方、ホスト変数として無効な datepart 値を指定した場合は、SQLCODE エラーは発行されず、DATENAME 関数は NULL 値を返します。

このテーブルに示されている返り値は、それぞれの日付部分の既定値です。これらの日付部分のいくつかの返り値は、各種日付/時刻オプションで [SET OPTION](#) コマンドを使用することで変更できます。

**week** : InterSystems IRIS は、InterSystems IRIS の既定のアルゴリズムと ISO 8601 標準アルゴリズムのいずれかを使用して、特定の日付がその年の何週目にあたるかを判定するように構成できます。詳細は、“[WEEK](#)” 関数を参照してください。

**weekday** : InterSystems IRIS の曜日に関する既定では、日曜日を週の最初の曜日として指定します (weekday=1)。ただし、週の最初の曜日を別の値に構成したり、月曜日を週の最初の曜日として指定する ISO 8601 標準を適用することができます。詳細は、“[DAYOFWEEK](#)” 関数を参照してください。

**millisecond** : InterSystems IRIS は、ミリ秒 (1000 分の 1 秒) 数が含まれる文字列を返します。date-expression に 3 桁より高い精度の小数部がある場合、InterSystems IRIS はそれを 3 桁に切り捨て、この数値を文字列として返します。date-expression が指定された精度であっても、3 桁の小数部の精度よりも低い場合、InterSystems IRIS は 3 桁までゼロでパディングしてから、この数字を文字列として返します。microsecond および nanosecond では、同じ切り捨てとゼロ・パディングが実行されます。

datepart は、引用符付き文字列として、または引用符なしで指定できます。以下のように構文に変化があると、実行される操作も少し異なります。

- 引用符あり: `DATENAME('month', '2018/02/25')` datepart は、クエリ・キャッシュの作成時にリテラルとして扱われます。InterSystems SQL は、リテラル置換を実行します。これにより、より一般的で再利用可能なクエリ・キャッシュが生成されます。
- 引用符なし: `DATENAME(month, '2018/02/25')` : datepart は、クエリ・キャッシュの作成時にキーワードとして扱われます。リテラル置換はありません。これにより、より特化したクエリ・キャッシュが生成されます。

## 日付式の形式

date-expression 引数は、以下のいずれかの形式になります。

- InterSystems IRIS %Date 論理値 (+\$H)
- InterSystems IRIS %PosixTime (%Library.PosixTime) 論理値 (エンコードされた 64 ビットの符号付き整数)
- InterSystems IRIS %TimeStamp (%Library.TimeStamp) 論理値 (YYYY-MM-DD HH:MM:SS.FFF)。ODBC 形式とも呼ばれます。
- InterSystems IRIS %String (互換性のある) 値

InterSystems IRIS %String (互換性のある) 値は、以下のいずれかの形式になります。

- 99999,99999 (\$H 形式)
- Sybase/SQL-Server-date Sybase/SQL-Server-time
- Sybase/SQL-Server-time Sybase/SQL-Server-date
- Sybase/SQL-Server-date (既定時刻は 00:00:00)
- Sybase/SQL-Server-time (既定日付は 01/01/1900)

Sybase/SQL-Server-date は、以下の 5 形式のいずれかです。

```
mmdelimiterddddelimiter[yy]yy dd Mmm[mm][,][yy]yy dd [yy]yy Mmm[mm] yyyy Mmm[mm] dd yyyy [dd] Mmm[mm]
```

delimiter は、スラッシュ (/)、ハイフン (-)、またはピリオド (.) です。

年が 2 桁で表示される場合、InterSystems IRIS はスライディング・ウィンドウを確認して日付を解釈します。スライディング・ウィンドウのシステム既定値は、%DATE ユーティリティで設定できます。これについては、<https://docs.intersystems.com/priordocexcerpts> にある旧ドキュメントに記載されています。現在のプロセスのスライディング・ウィンドウの設定の詳細は、ObjectScript の `$ZDATE`、`$ZDATEH`、`$ZDATETIME`、および `$ZDATETIMEH` の各関数のドキュメントを参照してください。

Sybase/SQL-Server-time は、以下の 3 形式のいずれかです。

HH:MM[:SS:SSS][{AM|PM}] HH:MM[:SS.S] HH[']{AM|PM}

date-expression で時刻形式が設定されているが、日付形式が指定されていない場合、既定で DATENAME は月曜日の曜日値を持つ日付 1900-01-01 になります。

## 範囲と値のチェック

DATENAME は、入力値に対して以下のチェックを実行します。値がチェックに失敗した場合は、NULL 文字列が返されます。

- 有効な date-expression は、日付文字列 (yyyy-mm-dd)、時刻文字列 (hh:mm:ss)、または日付/時刻文字列 (yyyy-mm-dd hh:mm:ss) で構成されます。日付と時刻の両方が指定されている場合は、両方が有効でなければなりません。例えば、時刻文字列が指定されていない場合は年数値を返すことができますが、無効な時刻文字列が指定されている場合は年数値を返すできません。
- 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。例えば、日の値が省略されている場合、年数値を返すことはできません。年は 4 桁で指定される必要があります。
- 時刻文字列は、適切な区切り文字で適切にフォーマットされている必要があります。時刻要素の値はゼロの場合もあるため、1 つ以上の時刻要素を省略でき (区切り文字はそのまま残す場合と省略する場合がある)、これらの要素はゼロ値として返されます。したがって、'hh:mm:ss'、'hh:mm:'、'hh:mm'、'hh::ss'、'hh::'、'hh'、および ':::' は、すべて有効です。時間 (Hour) 要素を省略するには、date-expression に文字列の日付部分がないことと、少なくとも 1 つの区切り文字 (:) があることが必要です。
- 日付/時刻値は、有効な範囲内にある必要があります。年は 0001 から 9999、月は 1 から 12、日は 1 から 31、時間は 0 から 23、分は 0 から 59、秒は 0 から 59 がそれぞれ有効範囲です。
- 月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。
- 日付値または時刻値が 10 よりも小さい場合、そのほとんどの先頭のゼロは、記載、省略のどちらでもかまいません。ただし、10 よりも小さい時間 (Hour) 値が日付/時刻文字列の一部の場合は、その先頭にゼロを付ける必要があります。その他の非標準的な整数値は許可されません。例えば、'07' または '7' は有効な日値ですが、'007'、'7.0'、または '7a' は無効です。
- date-expression で時刻形式が指定されているが、日付形式が指定されていない場合、DATENAME は時刻コンポーネント値の範囲検証を実行しません。

## 引数

### datepart

返す日付または時刻の情報のタイプ。日付または時刻の部分の名前 (または省略形)。この名前は小文字でも大文字でも指定できます。引用符で囲んでも囲まなくてもかまいません。datepart はリテラルまたはホスト変数として指定できます。

### date-expression

datepart 値を返す元となる日付、時刻、またはタイムスタンプ式です。date-expression には、タイプ datepart の値を含める必要があります。

## 例

以下の例では、各 DATENAME は 'Wednesday' を返します。指定された日付の曜日 ('dw') であるからです。

## SQL

```
SELECT DATENAME('dw','2018-02-21') AS DayName,
       DATENAME(dw,'02/21/2018') AS DayName,
       DATENAME('DW',64700) AS DayName
```

以下の例は、指定された日付の月名 ('mm') である 'December' を返します。

## SQL

```
SELECT DATENAME('mm','2018-12-20 12:00:00') AS MonthName
```

以下の例は、指定された日付の年名 (yy) である 2018 (文字列) を返します。

## SQL

```
SELECT DATENAME('yy','2018-12-20 12:00:00') AS Year
```

上記の例は、日付部分に省略形を使用していることに注意してください。ただし、以下の例のように、正式な名前を指定することもできます。

## SQL

```
SELECT DATENAME('year','2018-12-20 12:00:00') AS Year
```

以下の例では、現在の四半期、年間通算週、および年間通算日が返されます。各値は、文字列として返されます。

## SQL

```
SELECT DATENAME('Q',$HOROLOG) AS Q,
       DATENAME('WK',$HOROLOG) AS WkCnt,
       DATENAME('DY',$HOROLOG) AS DayCnt
```

以下の例では、ホスト変数として datepart および date-expression で渡します。

## SQL

```
SELECT DATENAME("year",$HOROLOG)
```

以下の例では、サブクエリを使用して、Sample.Person から誕生日が Wednesday のレコードが返されるようにします。

## SQL

```
SELECT Name AS WednesdaysChild,DOB
FROM (SELECT Name,DOB,DATENAME('dw',DOB) AS Wkday FROM Sample.Person)
WHERE Wkday='Wednesday'
ORDER BY DOB
```

## 関連項目

- SQL 関数 : [DATEADD](#)、[DATEDIFF](#)、[DATEPART](#)、[TO\\_DATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)
- ObjectScript 関数: [\\$ZDATETIME](#)

## DATEPART (SQL)

日付/時刻式の指定された部分の値を表す整数を返す、日付/時刻関数です。

### 構文

```
DATEPART (datepart, date-expression)
```

### 引数

引数	説明
datepart	返す日付または時刻の情報のタイプ。日付または時刻の部分の名前（または省略形）。この名前は大文字でも小文字でも指定できます。引用符で囲んでも囲まなくてもかまいません。datepart はリテラルまたはホスト変数として指定できます。
date-expression	datepart 値を返す元となる日付、時刻、またはタイムスタンプ式です。date-expression には、タイプ datepart の値を含める必要があります。

### 概要

DATEPART 関数は、指定された日付/時刻式についての datepart 情報をデータ型 Integer として返します。唯一の例外は sqltimestamp (sts) で、これはデータ型 %Library.Timestamp を返します。datepart 情報を文字列として返すには、[DATENAME](#) を使用します。

DATEPART が返すのは、date-expression の 1 要素のみの値です。複数の日付部分を含む文字列を返すには、[TO\\_DATE](#) を使用します。

この関数は、ObjectScript から DATEPART() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.DATEPART (datepart, date-expression)
```

DATEPART では、Sybase および Microsoft SQL Server との互換性が提供されています。

### Datepart 引数

datepart 引数は、以下の日付/時刻コンポーネントの 1 つで、正式な名前（日付部分列）または省略形（省略形列）のいずれかになります。これらの datepart コンポーネント名と省略形では、大文字と小文字は区別されません。

日付部分	省略形	返り値
year	yyyy, yy	0001–9999
quarter	qq, q	1–4
month	mm, m	1–12
week	wk, ww	1–53
weekday	dw, w	1–7 (Sunday、...、Saturday)
dayofyear	dy, y	1–366
day	dd, d	1–31
hour	hh, h	0–23
minute	mi, n	0–59
second	ss, s	0–59

日付部分	省略形	返り値
millisecond	ms	0-999 (3 桁の精度)
microsecond	mcs	0-999999 (6 桁の精度)
nanosecond	ns	0-999999999 (9 桁の精度)
sqltimestamp	sts	SQL_TIMESTAMP : yyyy-mm-dd hh:mm:ss

このテーブルに示されている返り値は、それぞれの日付部分の既定値です。これらの日付部分のいくつかの返り値は、各種日付/時刻オプションで **SET OPTION** コマンドを使用することで変更できます。

**week** : InterSystems IRIS は、InterSystems IRIS の既定のアルゴリズムと ISO 8601 標準アルゴリズムのいずれかを使用して、特定の日付がその年の何週目にあたるかを判定するように構成できます。詳細は、“**WEEK**” 関数を参照してください。

**weekday** : InterSystems IRIS の曜日に関する既定では、日曜日を週の最初の曜日として指定します (weekday=1)。ただし、週の最初の曜日を別の値に構成したり、月曜日を週の最初の曜日として指定する ISO 8601 標準を適用することができます。詳細は、“**DAYOFWEEK**” 関数を参照してください。ObjectScript の \$ZDATE および \$ZDATETIME 関数は、0 から 6 まで (1 から 7 ではない) の数値で曜日をカウントします。

**second** : date-expression に秒の小数部が含まれている場合、InterSystems IRIS は整数コンポーネントとして整数秒、および小数コンポーネントとして秒の小数部を含む 10 進数として second を返します。精度は切り捨てられません。

**millisecond** : InterSystems IRIS は、3 桁の精度の小数部を返します。末尾のゼロは削除されます。date-expression に 3 桁より高い精度の小数部がある場合、InterSystems IRIS はそれを 3 桁に切り捨てます。

**sqltimestamp** : InterSystems IRIS は、入力データをタイムスタンプ形式に変換し、必要に応じて、時間要素のためにゼロ値を指定します。sqltimestamp (略名 sts) datepart 値は、DATEPART と一緒にのみ使用します。この値を他のコンテキストで使用しようとししないでください。

datepart は、引用符で囲んだ文字列としてや、引用符なしで、または引用符で囲んだ文字列を括弧で囲んで指定できます。どう指定されているかに関係なく、datepart ではリテラル置換は実行されません。リテラル置換は date-expression で実行されます。datepart のすべての値は、sqltimestamp (または sts) を除き、データ型 INTEGER の値を返します。sqltimestamp は、その値をデータ型 TIMESTAMP の文字列として返します。

## 日付の入力形式

date-expression 引数は、以下のいずれかの形式になります。

- InterSystems IRIS %Date 論理値 (+\$H)
- InterSystems IRIS %PosixTime (%Library.PosixTime) 論理値 (エンコードされた 64 ビットの符号付き整数)
- InterSystems IRIS %TimeStamp (%Library.TimeStamp) 論理値 (YYYY-MM-DD HH:MM:SS.FFF)。ODBC 形式とも呼ばれます。
- InterSystems IRIS %String (互換性のある) 値

InterSystems IRIS %String (互換性のある) 値は、以下のいずれかの形式になります。

- 99999,99999 (\$H 形式)
- Sybase/SQL-Server-date Sybase/SQL-Server-time
- Sybase/SQL-Server-time Sybase/SQL-Server-date
- Sybase/SQL-Server-date (既定時刻は 00:00:00)
- Sybase/SQL-Server-time (既定日付は 01/01/1900)



Sybase/SQL-Server-date は、以下の 5 形式のいずれかです。

```
mmddelimiterddd delimiter[yy]yy dd Mmm[mm][,][yy]yy dd [yy]yy Mmm[mm] yyyy Mmm[mm] dd yyyy [dd] Mmm[mm]
```

delimiter は、スラッシュ (/)、ハイフン (-)、またはピリオド (.) です。

年が 2 桁で表示される場合、InterSystems IRIS はスライディング・ウィンドウを確認して日付を解釈します。スライディング・ウィンドウのシステム既定値は、%DATE ユーティリティで設定できます。これについては、<https://docs.intersystems.com/prior doc excerpts> にある旧ドキュメントに記載されています。現在のプロセスのスライディング・ウィンドウの設定の詳細は、ObjectScript の \$ZDATE、\$ZDATEH、\$ZDATETIME、および \$ZDATETIMEH の各関数のドキュメントを参照してください。

Sybase/SQL-Server-time は、以下の 3 形式のいずれかです。

```
HH:MM[:SS:SSS][{AM|PM}] HH:MM[:SS.S] HH[' ']{AM|PM}
```

date-expression で時刻形式が設定されているが、日付形式が指定されていない場合、既定で DATENAME は 2 (月曜日) の曜日値を持つ日付 1900-01-01 になります。

sqltimestamp では、時刻は 24 時間制で返されます。秒の小数部は切り捨てられます。

## 無効な引数のエラー・コード

無効な datepart オプションを指定した場合、DATEPART では SQLCODE -8 エラー・コードが生成され、%msg "badopt" DATEPART " が表示されます。

無効な date-expression 値 (アルファベット・テキスト文字列など) を指定した場合、DATEPART では SQLCODE -400 エラー・コードが生成され、%msg "Datepart(" DATEPART('year', 'badval') " が表示されます。検証に合格できない date-expression を指定した場合 (下記参照)、DATEPART では SQLCODE -400 エラー・コードが生成され、%msg "<ILLEGAL VALUE>datepart" が表示されます。

## 範囲と値のチェック

DATEPART は、date-expression 値に対して以下のチェックを実行します。値がチェックに失敗した場合は、NULL 文字列が返されます。

- 有効な date-expression は、日付文字列 (yyyy-mm-dd)、時刻文字列 (hh:mm:ss)、または日付/時刻文字列 (yyyy-mm-dd hh:mm:ss) で構成されます。日付と時刻の両方が指定されている場合は、両方が有効でなければなりません。例えば、時刻文字列が指定されていない場合は年数値を返すことができますが、無効な時刻文字列が指定されている場合は年数値を返すことができません。
- 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。例えば、日の値が省略されている場合、年数値を返すことはできません。年は 4 桁で指定される必要があります。
- 時刻文字列は、適切な区切り文字で適切にフォーマットされている必要があります。時刻要素の値はゼロの場合もあるため、1 つ以上の時刻要素を省略でき (区切り文字はそのまま残す場合と省略する場合がある)、これらの要素はゼロ値として返されます。したがって、'hh:mm:ss'、'hh:mm:', 'hh:mm', 'hh:ss', 'hh:', 'hh', および ':::' は、すべて有効です。時間 (Hour) 要素を省略するには、date-expression に文字列の日付部分がないことと、少なくとも 1 つの区切り文字 (:) があることが必要です。
- 日付/時刻値は、有効な範囲内にある必要があります。年は 0001 から 9999、月は 1 から 12、日は 1 から 31、時間は 0 から 23、分は 0 から 59、秒は 0 から 59 がそれぞれ有効範囲です。
- 月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。
- 日付値または時刻値が 10 よりも小さい場合、そのほとんどの先頭のゼロは、記載、省略のどちらでもかまいません。ただし、10 よりも小さい時間 (Hour) 値が日付/時刻文字列の一部の場合は、その先頭にゼロを付ける必要があります。



す。その他の非標準的な整数値は許可されません。例えば、'07' または '7' は有効な日値ですが、'007'、'7.0'、または '7a' は無効です。

- ・ date-expression で時刻形式が指定されているが、日付形式が指定されていない場合、DATEPART は時刻コンポーネント値の範囲検証を実行しません。

## 例

以下の例では、各 DATEPART は、日付時刻文字列の年の部分（この場合は、2018）を整数として返します。date-expression はさまざまな形式で指定でき、datepart は正式な名前または省略形で指定できることに注意してください。引用符で囲んでも囲まなくてもかまいません。

### SQL

```
SELECT DATEPART('yy','2018-02-22 12:00:00') AS YearDTS,
       DATEPART('year','2018-02-22') AS YearDS,
       DATEPART('YYYY','02/22/2018') AS YearD,
       DATEPART('YEAR',64701) AS YearHD,
       DATEPART('Year','64701,23456') AS YearHDT
```

以下の例では、\$HOROLOG 値に基づいて、現在の年と四半期が返されます。

### SQL

```
SELECT DATEPART('yyyy',$HOROLOG) AS Year,DATEPART('q',$HOROLOG) AS Quarter
```

以下の例では、曜日で並べ替えて、Sample.Person テーブルから誕生日の曜日が返されます。

### SQL

```
SELECT Name,DOB,DATEPART('weekday',DOB) AS bday
FROM Sample.Person
ORDER BY bday,DOB
```

以下の例では、各 DATEPART は、date-expression 文字列の分部分として 20 を返します。

### SQL

```
SELECT DATEPART('mi','2018-2-20 12:20:07') AS Minutes,
       DATEPART('n','2018-02-20 10:20:') AS Minutes,
       DATEPART('MINUTE','2018-02-20 10:20') AS Minutes
```

以下の例では、各 DATEPART は、date-expression 文字列の秒部分として 0 を返します。

### SQL

```
SELECT DATEPART('ss','2018-02-20 03:20:') AS Seconds,
       DATEPART('S','2018-02-20 03:20') AS Seconds,
       DATEPART('Second','2018-02-20') AS Seconds
```

以下の例では、TIMESTAMP データ型として完全な SQL タイムスタンプが返されます。DATEPART は、欠けている時間情報を埋めて、タイムスタンプ '2018/02/25 00:00:00' を返します。

### SQL

```
SELECT DATEPART('sqltimestamp','2/25/2018') AS DTStamp
```

以下の例は、日付と時刻を \$HOROLOG 形式で指定して、タイムスタンプの '2018/02/22 06:30:56' を返します。

### SQL

```
SELECT DATEPART('sqltimestamp','64701,23456') AS DTStamp
```

以下の例では、DATEPART を使用するサブクエリを使用して、誕生日がうるう年の日付 (2 月 29 日) である人が返されます。

### SQL

```
SELECT Name,DOB
FROM (SELECT Name,DOB,DATEPART('dd',DOB) AS DayNum,DATEPART('mm',DOB) AS Month FROM Sample.Person)
WHERE Month=2 AND DayNum=29
```

## 関連項目

- ・ [DATEDIFF](#) 関数
- ・ [DATENAME](#) 関数
- ・ [TIMESTAMPADD](#) 関数
- ・ [TIMESTAMPDIFF](#) 関数
- ・ [TO\\_DATE](#) 関数

# DATE\_TRUNC (SQL)

指定された精度に切り捨てられるタイムスタンプを返す日付/時刻関数。

## 構文

```
DATE_TRUNC( datePart, dateExpression )
```

## 説明

- DATE\_TRUNC([datePart](#), [date](#)) は、入力された date を指定された datePart まで切り捨てる日付式を返します。
  - date が **%Library.PosixTime** (エンコードされた 64 ビットの符号付き整数) である場合、DATE\_TRUNC からはデータ型 **%Library.PosixTime** のタイムスタンプが返されます。
  - date が他の型である場合、DATE\_TRUNC からはデータ型 **%Library.TimeStamp** が yyyy-mm-dd hh:mm:ss.fff の形式で返されます。

以下の文は、日付を日の単位までに切り捨てます。

### SQL

```
SELECT DATE_TRUNC('month', CURRENT_DATE)
```

DATE\_TRUNC は、Sybase および Microsoft SQL Server と互換性があります。

## 引数

### datePart

date をどこまで切り詰めるかを指定する、日付または時刻の部分のフルネームまたは省略した名前。datePart は大文字でも小文字でも指定できます。サポートされる日付と時刻の形式を、以下のテーブルに示します。

テーブル G-7:

日付部分	省略形
year	yyyy, yy
month	mm, m
quarter	qq, q
week	wk, ww
weekday	dw, w
day	dd, d
dayofyear	dy, y
hour	hh, h
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs

日付部分	省略形
nanosecond	ns

## date

切り詰める日付、時刻、またはタイムスタンプ式。この式は以下の型のうちのいずれかです。

- ・ %Date 論理値 (+\$H)。\$HOROLOG 形式とも呼ばれます。
- ・ %PosixTime (%Library.PosixTime) 論理値 (エンコードされた 64 ビットの符号付き整数)。
- ・ %TimeStamp (%Library.TimeStamp) 論理値 (YYYY-MM-DD HH:MM:SS.FFF)。ODBC 形式とも呼ばれます。
- ・ %String または文字列互換の値。以下のいずれかの形式とします。

テーブル G-8: \$HOROLOG の日付と時刻の形式

形式	例
dddd	SELECT DATE_TRUNC('yy', '66716')
dddd,sssss	SELECT DATE_TRUNC('yy', '66716,256')
dddd,sssss.fff	SELECT DATE_TRUNC('yy', '66716,256.467')

各要素の内容は以下のとおりです。

- dddd : 1840 年 12 月 31 日からの経過日数 (整数値)。
- sssss : 指定日の開始からの経過秒数。
- fff : 秒の小数部 (整数値)。秒の小数部を指定した場合は、DATE\_TRUNC の返り値にも秒の小数部が記述されます。

テーブル G-9: 日付形式

Format	例
MM/DD/YYYY	SELECT DATE_TRUNC('year', '8/24/2022')
MM-DD-YY	SELECT DATE_TRUNC('year', '12-31-99')
MM-DD-YYYY	SELECT DATE_TRUNC('year', '8-24-2022')
MM.DD.YY	SELECT DATE_TRUNC('year', '12.31.99')
MM.DD.YYYY	SELECT DATE_TRUNC('year', '8.24.2022')
Mmm DD YY	SELECT DATE_TRUNC('year', 'Dec 30 92')
Mmm DD YYYY	SELECT DATE_TRUNC('year', 'January 23 2021')
Mmm DD, YY	SELECT DATE_TRUNC('year', 'Dec 30, 92')
Mmm DD, YYYY	SELECT DATE_TRUNC('year', 'January 23, 2021')
MM/DD/YY	SELECT DATE_TRUNC('year', '12/31/99')

各要素の内容は以下のとおりです。

- MM : 2 桁の月。
- DD : 月の 2 桁の日数。

- Mmm : スペルアウト表記の月名。最低 3 文字 (例 : Mar) から完全な月名 (例 : March) まで指定できます。
- YY は 2 桁形式、YYYY は 4 桁形式の年です。

date は日付と時刻の文字列を組み合わせることで指定できます。以下に例を示します。

#### SQL

```
SELECT DATE_TRUNC('hh','12/22/2021 8:15:23')
```

日付を指定せずに時刻を指定すると、DATE\_TRUNC では既定で日付が 01/01/1900 になります。

#### テーブル G-10: 時刻形式

Format	例
HH:	SELECT DATE_TRUNC('hour','10:')
HH:MM	SELECT DATE_TRUNC('mi','10:30')
HH:MM:SS	SELECT DATE_TRUNC('ss','10:30:59')
HH...SS.FFF	SELECT DATE_TRUNC('ms','10:30:59.245')
HH...[AM PM]	SELECT DATE_TRUNC('mi','10:30PM')

各要素の内容は以下のとおりです。

- HH : 1 日での 2 桁の時間値。
- MM : 1 時間での 2 桁の分の値。
- SS : 1 分間での 2 桁の秒の値。
- FFF : 秒の小数部の値。

date は日付と時刻の文字列を組み合わせることで指定できます。以下に例を示します。

#### SQL

```
SELECT DATEADD('hh',1,'12/22/2021 8:15:23')
```

時刻を指定せずに日付を指定すると、DATE\_TRUNC では既定で時刻が 00:00:00 になります。

date 引数の制限事項と動作は以下のとおりです。

- ・ 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。年は 4 桁で指定される必要があります。入力値の日付部分を省略すると、DATE\_TRUNC では既定で日付が '1900-01-01' になります。
- ・ 日付と時刻の値は、以下の有効な範囲内にある必要があります。
  - 年 - 0001 から 9999
  - 月 - 1 から 12
  - 日 - 1 から 31
  - 時間 - 00 から 23
  - 分 - 0 から 59
  - 秒 - 0 から 59

月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。

- ・ 10(月および日)未満の日付値では、先頭のゼロを省略してもかまいません。その他の非標準的な整数値は許可されません。例えば、Day の値として '07' と '7' は有効ですが、'007'、'7.0'、'7a' は無効です。
- ・ 時刻値は省略してもかまいません。date で不完全な時刻が指定されている場合、指定されていない部分に対して 0 が指定されます。
- ・ 10 よりも小さい時間値の先頭の 0 は記載する必要があります。

## 例

以下の例は、[\\$HOROLOG](#) 形式で記述された日付に対して DATE\_TRUNC 関数を実行します。この例の結果は、2023-08-30 00:04:16 となります。

### SQL

```
SELECT DATE_TRUNC('ss', '66716,256.467')
```

以下の例は、文字列として記述された日付に対して DATE\_TRUNC 関数を実行します。結果は、1980-01-01 00:00:00 となります。

### SQL

```
SELECT DATE_TRUNC('yy', '11.25.80')
```

以下の例は、CURRENT\_DATE を使用して DATE\_TRUNC 関数を実行します。結果は日付によって異なりますが、日付は日の単位まで切り捨てられます。

### SQL

```
SELECT DATE_TRUNC('dd', CURRENT_DATE)
```

## 別の方法

ObjectScript コードでこの関数を呼び出すには、DATE\_TRUNC() メソッドを使用します。

```
$SYSTEM.SQL.Functions.DATE_TRUNC(datePart, dateExpression)
```

## 関連項目

- ・ [DATEADD](#)
- ・ [DATEDIFF](#)
- ・ [DATENAME](#)
- ・ [DATEPART](#)

# DAY (SQL)

---

日付式に対して月における日付を返す日付関数です。

## 構文

```
DAY(date-expression)  
{fn DAY(date-expression)}
```

## 説明

DAY は、特定の日付式に対して月における日付を返します。

DAY 関数はDAYOFMONTH 関数の別名です。DAY は、TSQL の互換性を果たせるために提供されています。詳細は "[DAYOFMONTH](#)" を参照してください。

## 引数

### *date-expression*

列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式。

## 関連項目

- ・ [DAYOFMONTH](#)



## DAYNAME (SQL)

日付式に対して曜日を返す日付関数です。

### 構文

```
{fn DAYNAME(date-expression)}
```

### 概要

DAYNAME は、指定された日付に該当する曜日名を返します。返り値は最長 15 文字の文字列です。既定として返される曜日名は、Sunday、Monday、Tuesday、Wednesday、Thursday、Friday、Saturday です。

これらの既定曜日名の値を変更するには、WEEKDAY\_NAME オプションで [SET OPTION](#) コマンドを使用します。

曜日名は、InterSystems IRIS 日付整数、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプに基づいて計算されます。

*date-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

タイムスタンプの時刻部分は評価されないので省略可能です。

DAYNAME は、指定された日付が有効であることをチェックします。年は 0001 ~ 9999、月は 01 ~ 12、日はその月に適切な数字 (例えば、02/29 はうるう年のみ有効) である必要があります。日付が無効の場合、DAYNAME は SQLCODE -400 エラー (深刻なエラーが発生しました) を発行します。

[DATENAME](#) 関数を使用して、同じ曜日情報を取得することもできます。[TO\\_DATE](#) を使用すると、その他の日付要素で曜日名または曜日の省略形を検索できます。曜日名に対応する整数を返すには、[DAYOFWEEK](#)、[DATEPART](#)、または [TO\\_DATE](#) を使用します。

この関数は、ObjectScript から DAYNAME() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.DAYNAME(date-expression)
```

### 引数

#### *date-expression*

InterSystems IRIS 日付整数、ODBC 日付、またはタイムスタンプとして評価される式。この式は、列の名前、他のスカラ関数の結果、または日付やタイムスタンプのリテラルとすることができます。

### 例

以下の 2 つの例は、指定された日付 (2018 年 2 月 21 日) の曜日が水曜日なので、Wednesday という文字列を返します。最初の例はタイムスタンプ文字列を取ります。

#### SQL

```
SELECT {fn DAYNAME('2018-02-21 12:35:46')} AS Weekday
```

2 番目の例は InterSystems IRIS 日付整数を取ります。

#### SQL

```
SELECT {fn DAYNAME(64700)} AS Weekday
```

以下の例はすべて、今日の曜日を返します。

## SQL

```
SELECT {fn DAYNAME({fn NOW()})} AS Wd_Now,  
       {fn DAYNAME(CURRENT_DATE)} AS Wd_CurrDate,  
       {fn DAYNAME(CURRENT_TIMESTAMP)} AS Wd_CurrTstamp,  
       {fn DAYNAME($ZTIMESTAMP)} AS Wd_ZTstamp,  
       {fn DAYNAME($HOROLOG)} AS Wd_Horolog
```

\$ZTIMESTAMP は協定世界時 (UTC) を返すことに注意してください。その他の time-expression 値はローカル時刻を返します。これは、DAYNAME 値に影響することがあります。

以下の例は、DAYNAME が無効な日付 (2017 年はうるう年ではない) をどのように処理するかを示しています。

## SQL

```
SELECT {fn DAYNAME("2017-02-29")}
```

## 関連項目

- SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DAYOFMONTH](#)、[DAYOFWEEK](#)、[DAYOFYEAR](#)、[TO\\_DATE](#)
- ObjectScript 関数 : [\\$ZDATE](#)
- ObjectScript 特殊変数 : [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## DAYOFMONTH (SQL)

日付式に対して月における日付を返す日付関数です。

### 構文

```
{fn DAYOFMONTH(date-expression)}
```

### 概要

DAYOFMONTH は、月における日付を 1 ～ 31 の整数で返します。date-expression には、InterSystems IRIS 日付整数、\$HOROLOG 値や \$ZTIMESTAMP 値、ODBC 形式の日付文字列、またはタイムスタンプを指定できます。

date-expression タイムスタンプには、データ型 %Library.PosixTime (エンコードされた 64 ビットの符号付き整数) またはデータ型 %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

タイムスタンプまたは \$HOROLOG 文字列の時刻部分は評価されないため省略可能です。

DAYOFMONTH 関数と DAY 関数は機能的に同一です。

この関数は、ObjectScript から DAYOFMONTH() メソッド・コールを使用して呼び出すこともできます。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.DAYOFMONTH("2018-02-25")
```

### タイムスタンプの date-expression

タイムスタンプ文字列の日 (dd) 部分は、1 から 31 までの範囲の整数でなければなりません。ただし、ユーザの指定値に対して、範囲チェックは行われません。31 よりも大きな数と小数は、指定されたとおりに返されます。(–) は区切り文字として使用されるため、負の数はサポートされません。入力では、先頭のゼロはオプションです。出力では、先頭のゼロは抑制されます。

日部分が '0'、'00'、または数以外の値の場合、DAYOFMONTH は NULL を返します。日付文字列の日部分が完全に省略されている場合 ('yyyy-mm hh:mm:ss')、または日付式が指定されていない場合も、同様に NULL が返されます。

日付/時刻文字列の要素は、SQL スカラ関数の YEAR、MONTH、DAYOFMONTH (または DAY)、HOUR、MINUTE、SECOND をそれぞれ使用して取得できます。DATEPART または DATENAME 関数を使用して、同じ要素を取得することもできます。DATEPART および DATENAME では、日の値に対して値と範囲のチェックが行われます。

### \$HOROLOG の date-expression

\$HOROLOG 値の月の日付を計算するときは、DAYOFMONTH では、うるう年の違いが計算されます (2000 年はうるう年で 1900 年と 2100 年はうるう年ではないという、1 世紀の日付調整など)。

DAYOFMONTH では、1840 年 12 月 31 日 よりも前の date-expression 値を負の整数として処理できます。詳細は、以下の例を参照してください。

#### SQL

```
SELECT {fn DAYOFMONTH(-306)} AS DayOfMonthFeb,      /* February 29, 1840 */
       {fn DAYOFMONTH(-305)} AS DayOfMonthMar,      /* March 1, 1840      */
       {fn DAYOFMONTH(-127410)} AS DayOfMonthFeb   /* February 29, 1492 */
```

LAST\_DAY 関数は、指定された日付の月の最後の日付を返します (\$HOROLOG 形式)。

## 引数

### date-expression

月における日付の値を返す元となる日付またはタイムスタンプ式です。列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式。

## 例

以下の例は、指定された日付がその月の 25 番目の日であるため 25 を返します。

### SQL

```
SELECT {fn DAYOFMONTH('2018-02-25')} AS DayNumTS,
       {fn DAYOFMONTH(64704)} AS DayNumH
```

以下の例も、月の日付として数値 25 を返します。年は省略されていますが、区切り文字列 (-) がプレースホルダとしての役割を果たしています。

### SQL

```
SELECT {fn DAYOFMONTH('-02-25 11:45:32')} AS DayNum
```

以下の例は、<null> を返します。

### SQL

```
SELECT {fn DAYOFMONTH('2018-02-00 11:45:32')} AS DayNum
```

### SQL

```
SELECT {fn DAYOFMONTH('2018-02 11:45:32')} AS DayNum
```

### SQL

```
SELECT {fn DAYOFMONTH('11:45:32')} AS DayNum
```

以下の DAYOFMONTH の例はすべて、月における今日の日付を示す値を返します。

### SQL

```
SELECT {fn DAYOFMONTH({fn NOW()})} AS DoM_Now,
       {fn DAYOFMONTH(CURRENT_DATE)} AS DoM_CurrD,
       {fn DAYOFMONTH(CURRENT_TIMESTAMP)} AS DoM_CurrTS,
       {fn DAYOFMONTH($HOROLOG)} AS DoM_Horolog,
       {fn DAYOFMONTH($ZTIMESTAMP)} AS DoM_ZTS
```

\$ZTIMESTAMP は協定世界時 (UTC) を返すことに注意してください。その他の time-expression 値はローカル時刻を返します。これは、DAYOFMONTH 値に影響することがあります。

以下の例は、先頭のゼロが抑制されることを示します。月の日付の値に応じて、1 または 2 の長さを返します。

### SQL

```
SELECT LENGTH({fn DAYOFMONTH('2018-02-05')}),
       LENGTH({fn DAYOFMONTH('2018-02-15')})
```

## 関連項目

- SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DAY](#)、[DAYNAME](#)、[DAYOFWEEK](#)、[DAYOFYEAR](#)、[LAST\\_DAY](#)、[MONTH](#)、[TO\\_DATE](#)

- ・ ObjectScript 関数: [\\$ZDATE](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

# DAYOFWEEK (SQL)

日付式に対する曜日を整数として返す日付関数です。

## 構文

```
{fn DAYOFWEEK(date-expression)}
```

## 概要

DAYOFWEEK は *date-expression* を受け取り、その日付の曜日に対応する整数を返します。曜日は週の最初の曜日からカウントされます。InterSystems IRIS の既定では、週の最初の曜日は日曜日です。そのため、既定では、返り値が示す曜日は以下のとおりです。

- ・ 1 – 日曜日
- ・ 2 – 月曜日
- ・ 3 – 火曜日
- ・ 4 – 水曜日
- ・ 5 – 木曜日
- ・ 6 – 金曜日
- ・ 7 – 土曜日

“[週の最初の曜日の設定](#)” の説明のとおり、システム全体で、または特定のネームスペースに対して、既定の週の最初の曜日をオーバーライドできます。

ObjectScript の \$ZDATE および \$ZDATETIME 関数は、0 から 6 まで (1 から 7 ではない) の数値で曜日をカウントします。

*date-expression* には、InterSystems IRIS 日付整数、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプを指定できます。

*date-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

タイムスタンプの時刻部分は評価されないので省略可能です。

[DATEPART](#) または [TO\\_DATE](#) 関数を使用して、同じ曜日情報を取得できます。曜日の名前を返すには、[DAYNAME](#)、[DATENAME](#)、または [TO\\_DATE](#) を使用します。

この関数は、ObjectScript から DAYOFWEEK() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.DAYOFWEEK(date-expression)
```

## 日付の検証

DAYOFWEEK は、入力値に対して以下のチェックを実行します。値がチェックに失敗した場合は、NULL 文字列が返されます。

- ・ 有効な *date-expression* は、日付文字列 (yyyy-mm-dd)、日付/時刻文字列 (yyyy-mm-dd hh:mm:ss)、InterSystems IRIS 日付整数、または [\\$HOROLOG](#) 値で構成されます。DAYOFWEEK は、*date-expression* の日付部分のみを評価します。
- ・ 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。年は 4 桁で指定される必要があります。
- ・ 日付値は、有効な範囲内にある必要があります。年は 0001 から 9999、月は 1 から 12、日は 1 から 31、

- ・ 月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。
- ・ 10 よりも小さい日付値の先頭のゼロは、記載、省略のどちらでもかまいません。その他の非標準的な整数値は許可されません。例えば、'07' または '7' は有効な日付値ですが、'007'、'7.0'、または '7a' は無効です。

## 週の最初の曜日の設定

既定では、週の最初の曜日は日曜日です。この既定値は、`SET ^%SYS("sql", "sys", "day of week")=n` と指定してシステム全体でオーバーライドできます。n の値は、1 (月曜日) ～ 7 (日曜日) です。週の最初の曜日を月曜日に設定するには、`SET ^%SYS("sql", "sys", "day of week")=1` と指定します。月曜日が週の最初の曜日の場合、水曜日の date-expression は 3 を返します。日曜日が週の最初の曜日の場合に返される 4 ではありません。InterSystems IRIS の既定 (日曜日が週の最初の曜日) をリセットするには、`SET ^%SYS("sql", "sys", "day of week")=7` と指定します。

特定のネームスペースの週の最初の曜日を設定するには、`SET ^%SYS("sql", "sys", "day of week", namespace)=n` と指定します。n の値は、1 (月曜日) ～ 7 (日曜日) です。USER ネームスペースの週の最初の曜日を月曜日に設定するには、`SET ^%SYS("sql", "sys", "day of week", "USER")=1` と指定します。ネームスペース・レベルで週の最初の曜日を設定した後、`SET ^%SYS("sql", "sys", "day of week")=n` と指定してシステム全体の設定を変更しても、そのネームスペースには反映されません。そのネームスペースの既定の週の最初の曜日を変更する機能をリストアするには、`^%SYS("sql", "sys", "day of week", namespace)` を削除する必要があります。以下の例を参照してください。

InterSystems IRIS では、曜日やその年の何週目かなどの日付設定を決定するための ISO 8601 標準もサポートされています。この規格は、主にヨーロッパの国で使用されます。ISO 8601 標準では、週の最初の曜日は月曜日です。ISO 8601 をアクティブにするには `SET ^%SYS("sql", "sys", "week ISO8601")=1` を指定し、非アクティブにするには、この値を 0 に設定します。week ISO8601 がアクティブになっており、InterSystems IRIS の day of week が定義されていないか既定 (7=日曜日) に設定されている場合は、ISO 8601 標準が InterSystems IRIS の既定よりも優先されます。InterSystems IRIS の day of week がこれ以外の値に設定されている場合は、その値によって DAYOFWEEK の week ISO8601 がオーバーライドされます。以下の例を参照してください。

## 引数

### date-expression

ODBC 形式または \$HOROLOG 形式の有効な日付 (時刻コンポーネントの指定あり、または指定なし)。列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式

## 例

以下の例では、両方の選択項目が数値 5 (日曜日が週の最初の曜日に設定されている場合) を返します。なぜなら、指定された date-expression (64701 = 2018 年 2 月 22 日) が木曜日であるためです。

### SQL

```
SELECT {fn DAYOFWEEK('2018-02-22')}||'|'|DATENAME('dw','2018-02-22') AS ODBCDoW,
       {fn DAYOFWEEK(64701)}||'|'|DATENAME('dw','64701') AS HorologDoW
```

以下の例では、すべての選択項目が今日の曜日に対応する整数を返します。

### SQL

```
SELECT {fn DAYOFWEEK({fn NOW()})} AS DoW_Now,
       {fn DAYOFWEEK(CURRENT_DATE)} AS DoW_CurrDate,
       {fn DAYOFWEEK(CURRENT_TIMESTAMP)} AS DoW_CurrTstamp,
       {fn DAYOFWEEK($ZTIMESTAMP)} AS DoW_ZTstamp,
       {fn DAYOFWEEK($HOROLOG)} AS DoW_Horolog
```



\$ZTIMESTAMP は協定世界時 (UTC) を返すことに注意してください。その他の time-expression 値はローカル時刻を返します。これは、DAYOFWEEK 値に影響することがあります。

以下の組み込み SQL の例は、ネームスペースの週の最初の曜日を変更する方法を示しています。まずシステム全体の週の最初の曜日を設定し (7)、次にネームスペースの週の最初の曜日を設定します (3)。その後システム全体の週の最初の曜日を変更しても (2)、プログラムによってネームスペース固有の設定が削除されるまで、ネームスペースの週の最初の曜日には反映されません。ネームスペース固有の設定を削除すると、ネームスペースの週の最初の曜日が直ちに現在のシステム全体の値にリセットされます。最後に、プログラムによって最初のシステム全体の設定がリストアされます。

**注釈** 以下のプログラムは、%SYS または USER ネームスペースで週の最初の曜日のネームスペース固有設定が存在するかどうかをテストします。存在する場合、このプログラムは処理を中止し、これらの設定の変更を回避します。

### ObjectScript

```

SetUp
    SET TestNsp="USER"
    SET ControlNsp="%SYS"
InitialDoWValues
    WRITE "Systemwide default DoW initial values",!
    DO TestDayofWeek()
    IF a=b {WRITE "No namespace-specific DoW defaults",!!}
    ELSE {WRITE "DoW initial settings are namespace-specific",!
        WRITE "Stopping this program"
        QUIT }
    SET initialDoW=%SYS("sql","sys","day of week")
SetSystemwideDoW
    KILL ^%SYS("sql","sys","day of week",TestNsp)
    KILL ^%SYS("sql","sys","day of week",ControlNsp)
    SET ^%SYS("sql","sys","day of week")=7
    WRITE "Systemwide DoW set",!
    DO TestDayofWeek()
SetNamespaceDoW
    SET ^%SYS("sql","sys","day of week",TestNsp)=3
    WRITE TestNsp," namespace DoW set",!
    &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
    DO TestDayofWeek()
ResetSystemwideDoW
    SET ^%SYS("sql","sys","day of week")=2
    WRITE "Systemwide DoW set with ",TestNsp," DoW set",!
    DO TestDayofWeek
KillNamespaceDoW
    KILL ^%SYS("sql","sys","day of week",TestNsp)
    WRITE "Namespace ",TestNsp," DoW killed",!
    DO TestDayofWeek
ResetSystemwideDoWDefault
    SET ^%SYS("sql","sys","day of week")=initialDoW
    WRITE "Systemwide DoW reset after ",TestNsp," DoW killed",!
    DO TestDayofWeek
TestDayofWeek()
    SET $NAMESPACE=TestNsp
    &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
    WRITE "Today is the ",a," day of week in ",$NAMESPACE,!
    SET $NAMESPACE=ControlNsp
    &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :b)
    WRITE "Today is the ",b," day of week in ",$NAMESPACE,!!
    RETURN

```

以下の埋め込み SQL の例は、既定の曜日と、ISO 8601 標準が適用された場合の曜日を表示します。この例では、InterSystems IRIS の day of week が未設定であるか既定値に設定されていると想定しています。

## ObjectScript

```
TestISO
  SET def=$DATA(^%SYS("sql","sys","week ISO8601"))
  IF def=0 {SET ^%SYS("sql","sys","week ISO8601")=0}
  ELSE {SET isoval=^%SYS("sql","sys","week ISO8601")}
    IF isoval=1 {GOTO UnsetISO }
    ELSE {SET isoval=0 GOTO DayofWeek }
UnsetISO
  SET ^%SYS("sql","sys","week ISO8601")=0
DayofWeek
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  WRITE "Today:",!
  WRITE "default day of week is ",a,!
  SET ^%SYS("sql","sys","week ISO8601")=1
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :b)
  WRITE "ISO8601 day of week is ",b,!
ResetISO
  SET ^%SYS("sql","sys","week ISO8601")=isoval
```

## 関連項目

- ・ SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DAYNAME](#)、[DAYOFMONTH](#)、[DAYOFYEAR](#)、[TO\\_DATE](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

# DAYOFYEAR (SQL)

日付式に対して年における日付を整数として返す日付関数です。

## 構文

```
{fn DAYOFYEAR(date-expression)}
```

## 概要

DAYOFYEAR は、指定された日付式の日付がその年の日付に対応する 1 から 366 までの整数を返します。DAYOFYEAR はうるう年の日付にも対応します。

年における日付は、InterSystems IRIS 日付整数、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプに基づいて計算されます。

*date-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

タイムスタンプの時刻部分は評価されないので省略可能です。

[\\$HOROLOG](#) 値の月の日付を計算するときは、DAYOFYEAR では、うるう年の違いが計算されます (2000 年はうるう年で 1900 年と 2100 年はうるう年ではないという、1 世紀の日付調整など)。

DAYOFYEAR では、1840 年 12 月 31 日 よりも前の *date-expression* 値を負の整数として処理できます。詳細は、以下の例を参照してください。

## SQL

```
SELECT {fn DAYOFYEAR(-306)} AS LastDayFeb, /* February 29, 1840 */
       {fn DAYOFYEAR(-305)} AS FirstDayMar /* March 1, 1840 */
```

最も早い有効な *date-expression* は -672045 です (1 年 1 月 1 日)。

[DATEPART](#) または [DATENAME](#) 関数を使用して、同じ日付カウントを取得することもできます。DATEPART および DATENAME では、日付式に対して値と範囲のチェックが行われます。

この関数は、ObjectScript から DAYOFYEAR() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.DAYOFYEAR(date-expression)
```

## 引数

### *date-expression*

列の名前、他のスカラー関数の結果、または日付やタイムスタンプのリテラルによる、日付式。

## 例

以下の例は、日付式の日 (2016 年 3 月 4 日) がその年の 64 番目の日であるため、ともに整数 64 を返します (うるう年の日付であることが自動的に計算されます)。

## SQL

```
SELECT {fn DAYOFYEAR('2016-03-04 12:45:37')} AS DayCount
```

## SQL

```
SELECT {fn DAYOFYEAR(63981)} AS DayCount
```

以下の例はすべて、今日の日付を示す値を返します。

### SQL

```
SELECT {fn DAYOFYEAR({fn NOW()})} AS DNumNow,  
       {fn DAYOFYEAR(CURRENT_DATE)} AS DNumCurrD,  
       {fn DAYOFYEAR(CURRENT_TIMESTAMP)} AS DNumCurrTS,  
       {fn DAYOFYEAR($HOROLOG)} AS DNumHorolog,  
       {fn DAYOFYEAR($ZTIMESTAMP)} AS DNumZTS
```

\$ZTIMESTAMP は協定世界時 (UTC) を返すことに注意してください。その他の time-expression 値はローカル時刻を返します。これは、DAYOFYEAR 値に影響することがあります。

以下の例では、サブクエリを使用して、誕生日の年間通算日順で並べ替えて Employee レコードが返されるようにします。

### SQL

```
SELECT Name,DOB  
FROM (SELECT Name,DOB,{fn DAYOFYEAR(DOB)} AS BDay FROM Sample.Employee)  
ORDER BY BDay
```

## 関連項目

- SQL 関数: [DATENAME](#)、[DATEPART](#)、[DAYNAME](#)、[DAYOFMONTH](#)、[DAYOFWEEK](#)
- ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

# DECODE (SQL)

与えられた式を評価し、指定された値を返す関数です。

## 構文

```
DECODE(expr {,search,result}[,default])
```

## 概要

search と result の複数の組み合わせを、コンマで区切って指定できます。default を 1 つ指定できます。DECODE 式のパラメータ (expr、search、result、および default を含む) の最大数は、およそ 100 です。search、result、default の各値は、式から導き出すことができます。

DECODE 式を評価するために、InterSystems IRIS は expr を各 search 値と 1 つずつ比較します。

- expr が search 値と等しい場合は、対応する result が返されます。
- expr がどの search 値とも等しくない場合は、default 値が返されます。default が省略されている場合は、NULL が返されます。

InterSystems IRIS が各 search 値を評価するのはそれを expr と比較する前だけで、すべての search 値を評価してから expr と比較されるわけではありません。したがって、InterSystems IRIS は、search が expr と一致した場合は、それ以降の search を評価しません。

DECODE 式では、InterSystems IRIS は 2 つの NULL を同等と見なします。expr が NULL の場合、InterSystems IRIS は、NULL である最初の search の result を返します。

DECODE は Oracle との互換性をサポートします。

## 返り値のデータ型

DECODE は、最初の result 引数のデータ型を返します。最初の result 引数のデータ型が特定されない場合、DECODE は VARCHAR を返します。数値の場合、DECODE は、可能なすべての result 引数値の中から、最大の長さ、有効桁数、および小数桁数を返します。

result および default のデータ型が異なる場合、返されるデータ型は可能なすべての返り値と最も互換性の高い型、つまり最も高いデータ型の優先順位を持つデータ型になります。例えば、result が整数で default が小数の場合、DECODE はデータ型 NUMERIC の値を返します。これは、NUMERIC が両方に互換性がある、最も高い優先順位を持つデータ型であるためです。

## 引数

### expr

解読される式。

### search

expr が比較される対象値。

### result

expr が search と一致する場合に返される値。

### default

expr が search と一致しない場合に返される既定値 (オプション)。

## 例

以下の例は、13 から 19 までの年齢を 'Teen' として “解釈” します。default は 'Adult' です。

### SQL

```
SELECT Name, Age, DECODE(Age,
    13, 'Teen', 14, 'Teen', 15, 'Teen', 16, 'Teen',
    17, 'Teen', 18, 'Teen', 19, 'Teen',
    'Adult') AS AgeBracket
FROM Sample.Person
WHERE Age > 12
```

以下の例では、NULL をデコードします。FavoriteColors に値がない場合は、DECODE はそれを文字列 'No Preference' に置換し、値がある場合は、FavoriteColors の値を返します。

### SQL

```
SELECT Name, DECODE(FavoriteColors,
    NULL, 'No Preference',
    $LISTTOSTRING(FavoriteColors, '^')) AS ColorPreference
FROM Sample.Person
ORDER BY Name
```

以下の例では、色の好みをデコードします。その人の好きな色が 1 つの場合は、色の名前が省略形に置き換えられます。好きな色が複数ある場合は、DECODE は FavoriteColors の値を返します。

### SQL

```
SELECT Name, DECODE(FavoriteColors,
    $LISTBUILD('Red'), 'R',
    $LISTBUILD('Orange'), 'O',
    $LISTBUILD('Yellow'), 'Y',
    $LISTBUILD('Green'), 'G',
    $LISTBUILD('Blue'), 'B',
    $LISTBUILD('Purple'), 'V',
    $LISTBUILD('White'), 'W',
    $LISTBUILD('Black'), 'K',
    $LISTTOSTRING(FavoriteColors, '^'))
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

この **ORDER BY** 節では元のフィールド値で並べ替えているという点に注意してください。以下の例では、DECODE の値で並べ替えます。

### SQL

```
SELECT Name, DECODE(FavoriteColors,
    $LISTBUILD('Red'), 'R',
    $LISTBUILD('Orange'), 'O',
    $LISTBUILD('Yellow'), 'Y',
    $LISTBUILD('Green'), 'G',
    $LISTBUILD('Blue'), 'B',
    $LISTBUILD('Purple'), 'V',
    $LISTBUILD('White'), 'W',
    $LISTBUILD('Black'), 'K',
    $LISTTOSTRING(FavoriteColors, '^')) AS ColorCode
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY ColorCode
```

以下の例は、Employee レコードの Company コード・フィールドの数値コードを解釈し、対応する部署名を返します。従業員の Company コードが 1 から 10 までに該当しない場合、DECODE は既定の “Admin (non-tech)” を返します。

## SQL

```
SELECT Name,  
DECODE (Company,  
    1, 'TECH MARKETING', 2, 'TECH SALES', 3, 'DOCUMENTATION',  
    4, 'BASIC RESEARCH', 5, 'SOFTWARE DEVELOPMENT', 6, 'HARDWARE DEVELOPMENT',  
    7, 'QUALITY TESTING', 8, 'FIELD SUPPORT', 9, 'PHONE SUPPORT',  
    10, 'TECH TRAINING',  
    'Admin (non-tech)') AS TechJobs  
FROM Sample.Employee
```

この式では、expr パラメータが Company で、search と result パラメータの 10 個の組み合わせが使用されています。“Admin (non-tech)” は default パラメータです。

## 関連項目

- [CASE](#)



## DEGREES (SQL)

ラジアンを角度に変換する数値関数です。

### 構文

```
DEGREES(numeric-expression)  
{fn DEGREES(numeric-expression)}
```

### 説明

DEGREES は、ラジアン単位で角度測定を行い、対応する角度測定を度単位で返します。DEGREES は、NULL 値を渡すと NULL 値を返します。

返される値は、既定の有効桁数が 36 で、既定の小数桁数が 18 です。

度をラジアンに変換するには、[RADIANS](#) 関数を使用できます。

### 引数

#### *numeric-expression*

セーブポイントの名前。[識別子](#)として指定します。ラジアン単位の角度のメジャー。数値に解決される式です。

DEGREES は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。DEGREES は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

DEGREES は、標準スカラ関数または {} 括弧構文による ODBC スカラ関数として指定できます。

### 例

以下の埋め込み SQL の例は、ラジアン値 0 から 6 に対応する同等の角度を返します。

#### ObjectScript

```
SET a=0  
WHILE a<7 {  
&sql(SELECT DEGREES(:a) INTO :b)  
IF SQLCODE'=0 {  
    WRITE !,"Error code ",SQLCODE  
    QUIT }  
ELSE {  
    WRITE !,"radians ",a," = degrees ",b  
    SET a=a+1 }  
}
```

### 関連項目

- SQL 関数: [CONVERT](#)、[RADIANS](#)、[TO\\_NUMBER](#)

## %EXACT (SQL)

文字を EXACT 照合形式に変換する照合関数です。

### 構文

```
%EXACT(expression)
```

```
%EXACT expression
```

### 概要

%EXACT は、EXACT 照合シーケンス内の expression を返します。この照合シーケンスは、次のように値を順序付けます。

1. すべての実際の値の前に NULL が照合されます。%EXACT は、NULL に影響しません。これは既定の照合と同じです。
2. **キャノニック形式の数値**は (数値として入力されたか、文字列として入力されたかに関係なく)、文字列値よりも先に数値順に照合されます。
3. 文字列値は、大文字/小文字を区別する文字列順序で照合されます。文字列に対する EXACT 照合シーケンスは、ANSI 標準の ASCII 照合シーケンスと同じものです。数字は大文字のアルファベット文字の前に照合され、大文字のアルファベット文字は小文字のアルファベット文字の前に照合されます。句読点文字は、そのシーケンスのそれぞれの場所で照合されます。

この結果、以下のような順序になります。

```
NULL
-2      /* canonical number collation */
0
1
2
10
22
88
''      /* empty string */
#       /* character-by-character string collation */
-00     /* non-canonical number collates as string */
0 Elm St. /* character-by-character string collation */
022     /* non-canonical number collates as string */
1 Elm St.
19 Elm St.
19 elm St. /* string collation is case-sensitive */
19Elm St.
2 Elm St.
201 Elm St.
21 Elm St.
Elm St.
```

%EXACT は一般的に、文字が含まれる文字列値を、大文字/小文字を区別する順序で照合するために使用されます。SQL の既定では、照合目的ですべての文字を大文字に変換します。

%EXACT は、InterSystems SQL の拡張機能であり、SQL 検索クエリ用として使用するものです。

%SYSTEM.Util クラスの Collation() メソッドを使用すると、ObjectScript で同じ照合変換を実行できます。

%EXACT は、入力文字列を完全な数値として (キャノニック形式)、または数字が他の任意の文字と同じように処理される混合文字列として照合します。これを %MVR と比べると、%MVR では文字列内の数値部分文字列に基づいて文字列をソートしている点が異なります。

### DISTINCT および GROUP BY

**DISTINCT** 節と**GROUP BY** 節は、大文字の既定の照合に基づいて値をグループ化し、実際のデータ値がすべて大文字でない場合でも、すべて大文字で値を返します。

- ・ %EXACT を使用して、大文字/小文字を区別する値で値をグループ化することができます:`SELECT Name FROM mytable GROUP BY %EXACT(Name)`
- ・ %EXACT を使用して、実際の大文字/小文字を区別する値をグループごとに返すことができます:`SELECT %EXACT(Name) FROM mytable GROUP BY Name`

注釈 既定では、SQL インデックスは文字列データを大文字の既定の照合で表します。このため、EXACT 照合を指定すると、インデックスの使用が妨げられ、潜在的にパフォーマンスに大きな影響を及ぼす可能性があります。

## 引数

### commitmode

列名、文字列リテラル、数値、または他の関数の結果となる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR2 など) とすることができます。

## 例

以下の例は、すべての番地を %EXACT 照合で並べます。

### SQL

```
SELECT Name, Street
FROM Sample.Person
ORDER BY %EXACT Street
```

以下の例は、%EXACT を使用して、'Smith' よりも照合順序が高いすべての Name 値を返します。最初の例は括弧構文を使用し、2 番目の例は括弧を省略しています。

### SQL

```
SELECT Name
FROM Sample.Person
WHERE %EXACT(Name) > 'Smith'
```

### SQL

```
SELECT Name
FROM Sample.Person
WHERE %EXACT Name > 'Smith'
```

## 関連項目

- ・ [ASCII 関数](#)
- ・ [%SQLSTRING 照合関数](#)
- ・ [%SQLUPPER 照合関数](#)
- ・ [%TRUNCATE照合関数](#)
- ・ [照合](#)

# EXP (SQL)

数値の指数値 (自然対数の逆関数) を返すスカラ数値関数です。

## 構文

```
{fn EXP(expression)}
```

## 概要

EXP は指数関数  $e^x$  であり、 $e$  は定数 2.718281828 です。そのため、 $e$  の値を返すために、{fn EXP(1)} と指定できます。EXP は自然対数関数 LOG の逆関数です。

EXP は、有効桁数が 36 で小数桁数が 18 の値を返します。NULL 値を渡すと EXP は NULL を返します。

EXP は ({} 括弧構文と共に) ODBC スカラ関数としてのみ使用できます。

## 引数

### expression

対数の指数値。数値式です。

EXP は、NUMERIC または DOUBLE データ型のいずれかを返します。EXP は、expression がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、定数  $e$  を返します。

### SQL

```
SELECT {fn EXP(1)} AS e_constant
```

これは、2.718281828... を返します。

以下の埋め込み SQL の例は、整数 0 から 10 までの指数値を返します。

### ObjectScript

```
SET a=0
WHILE a<11 {
&sql(SELECT {fn EXP(:a)} INTO :b)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE
QUIT }
ELSE {
WRITE !,"Exponential of ",a," = ",b
SET a=a+1 }
}
```

以下の例は、EXP が LOG の逆関数であることを示しています。

### SQL

```
SELECT {fn EXP(7)} AS Exp,
       {fn LOG(7)} AS Log,
       {fn EXP({fn LOG(7)})} AS ExpOfLog
```

3 番目の関数呼び出しで、入力された数値と計算された戻り値との間に小さな差異があることに注意してください。以下の例は、この計算上の誤差を処理する方法を示します。

以下の埋め込み SQL の例は、整数 1 から 10 までに対する LOG 関数と EXP 関数の関係を示します。

### ObjectScript

```
SET a=1
WHILE a<11 {
  &sql(SELECT {fn LOG(:a)} INTO :b)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
  ELSE {
    WRITE !,"Logarithm of ",a," = ",b }
  &sql(SELECT ROUND({fn EXP(:b)},12) INTO :c)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"Exponential of log ",b," = ",c
    SET a=a+1 }
}
```

ここでは、システムの計算制限によって生じる非常に小さな差異を修正する ROUND 関数が必要なことに注意してください。上記の例では、この目的で ROUND がとりあえず小数点以下 12 桁に設定されています。

### 関連項目

- SQL 関数 : [LOG LOG10 POWER ROUND](#)
- ObjectScript 関数 : [\\$ZEXP](#)

## %EXTERNAL (SQL)

DISPLAY 形式の式を返す形式変換関数です。

### 構文

```
%EXTERNAL(expression)
```

```
%EXTERNAL expression
```

### 概要

%EXTERNAL は、現在選択されているモード (表示モード) に関係なく、*expression* を DISPLAY 形式に変換します。DISPLAY 形式は、フィールドやデータ型の LogicalToDisplay メソッドによってどのようなデータ変換が実行されても、VARCHAR データ型でデータを表します。

通常、%EXTERNAL は、SELECT リストの select-item で使用されます。WHERE 節内で使用できますが、%EXTERNAL を使用すると指定されたフィールドでインデックスを使用できなくなるので、使用しないことをお勧めします。

%EXTERNAL を適用すると、列見出し名が “Expression\_1” などの値に変更されます。したがって、後述の例で示すように、通常は列名のエイリアスを指定することをお勧めします。

%EXTERNAL による変換にかかわらず、日付は、日付フィールドまたは関数によって返されるデータ型に依存しています。%EXTERNAL は、CURDATE、CURRENT\_DATE、CURTIME、および CURRENT\_TIME の値を変換します。CURRENT\_TIMESTAMP、GETDATE、GETUTCDATE、NOW、および \$HOROLOG の値は変換しません。

%EXTERNAL によって %List 構造が DISPLAY 形式に変換されると、表示されるリスト要素は、空白スペースで区切られます。この “スペース” は実際には、CHAR(13) と CHAR(10) という 2 つの非表示文字です。

%EXTERNAL は、InterSystems SQL の拡張です。

現在の選択モードに関係なく *expression* を LOGICAL 形式に変換するには、%INTERNAL 関数を使用します。現在の選択モードに関係なく *expression* を ODBC 形式に変換するには、%ODBCOUT 関数を使用します。

表示形式オプションの詳細は、“[データ表示オプション](#)” を参照してください。

### 引数

#### *expression*

変換される式。フィールド名、フィールド名を含む式、または変換可能なデータ型で値を返す関数 (DATE や %List など)。ストリーム・フィールドにすることはできません。

### 例

以下のダイナミック SQL の例では、DOB (誕生日) データ値を現在の選択モード形式で返し、%EXTERNAL 関数を使用して同じデータを返します。デモンストレーションのため、このプログラムでは、%SelectMode 値は、呼び出すごとにランダムに決定されます。

#### SQL

```
SELECT TOP 5 DOB,%EXTERNAL(DOB) AS ExtDOB
FROM Sample.Person
```

以下の例では、この関数の 2 つの構文形式を示します。それ以外は同じです。%List フィールドの %EXTERNAL (DISPLAY 形式)、%INTERNAL (LOGICAL 形式)、および %ODBCOUT (ODBC 形式) を指定します。

## SQL

```
SELECT TOP 10 %EXTERNAL(FavoriteColors) AS ExtColors,  
              %INTERNAL(FavoriteColors) AS IntColors,  
              %ODBCOUT(FavoriteColors) AS ODBCColors  
FROM Sample.Person
```

## SQL

```
SELECT TOP 10 %EXTERNAL FavoriteColors AS ExtColors,  
              %INTERNAL FavoriteColors AS IntColors,  
              %ODBCOUT FavoriteColors AS ODBCColors  
FROM Sample.Person
```

以下の例では、誕生日 (DOB) および丸めた誕生日 (DOB) の値を %EXTERNAL (DISPLAY 形式) に変換します。

## SQL

```
SELECT %EXTERNAL(DOB) AS DOB,  
       %INTERNAL(ROUND(DOB,-3)) AS DOBGroup,  
       %EXTERNAL(ROUND(DOB,-3)) AS RoundedDOB  
FROM Sample.Person  
GROUP BY (ROUND(DOB,-3))  
ORDER BY DOBGroup
```

## 関連項目

- ・ [%INTERNAL](#)、[%ODBCIN](#)、[%ODBCOUT](#)
- ・ [SQL の概念: データ型、日付/時刻文](#)



## \$EXTRACT (SQL)

文字列から、指定された位置にある文字を抽出する文字列関数です。

### 構文

```
$EXTRACT(string[,from[,to]])
```

### 説明

\$EXTRACT は string の指定した位置から部分文字列を返します。返される部分文字列の内容は、使用する引数によって決まります。

- ・ \$EXTRACT(string) は、文字列の先頭文字を抽出します。
- ・ \$EXTRACT(string,from) は、from で指定した位置の文字を抽出します。例えば、変数 var1 が文字列 “ABCD” を含むときは、以下のコマンドは “B” を抽出します (2 番目の文字)。

#### SQL

```
SELECT $EXTRACT('ABCD',2) AS Extracted
```

- ・ \$EXTRACT(string,from,to) は、from の位置で開始し to の位置で終了する文字列の範囲を抽出します。例えば、以下のコマンドは、文字列 “1234Alabama567” から文字列 “Alabama” (つまり、5 番目の位置から 11 番目の位置までにあるすべての文字) を抽出します。

#### SQL

```
SELECT $EXTRACT('1234Alabama567',5,11) AS Extracted
```

この関数は、データ型 VARCHAR を返します。

### 引数

#### string

string には、変数名、数値、文字列リテラル、または任意の有効な式を指定できます。

#### from

from 値には、正の整数を指定する必要があります (例外は、“[メモ](#)”を参照)。小数値の場合は、小数桁が切り捨てられ、整数部分のみが使用されます。

from 値が文字列内の文字数よりも大きいときは、\$EXTRACT は NULL 文字列を返します。

from が to 引数なしで指定されている場合は、その位置にある 1 文字が抽出されます。

to 引数と共に使用されている場合、from は抽出する範囲の先頭を表し、to 値よりも小さな値でなくてはなりません。from が to と等しい場合、\$EXTRACT は指定した位置にある 1 文字を返します。from が to よりも大きいときは、\$EXTRACT は NULL 文字列を返します。

#### to

to 引数は、from 引数と共に使用する必要があります。必ず正の整数を指定します。小数値の場合は、小数桁が切り捨てられ、整数部分のみが使用されます。

to の値が from の値以上である場合、\$EXTRACT は指定した部分列を返します。to が文字列の長さより大きい場合、\$EXTRACT は from の位置から文字列の終了までの部分文字列を返します。to が from よりも小さい場合、\$EXTRACT は NULL 文字列を返します。

## 例

以下の例は文字列の 4 番目の文字 “S” を返します。

### SQL

```
SELECT $EXTRACT('THIS IS A TEST',4) AS Extracted
```

以下の例は、先頭から 7 番目の文字までの部分文字列 “THIS IS” を返します。

### SQL

```
SELECT $EXTRACT('THIS IS A TEST',1,7) AS Extracted
```

以下の例は、“ABCD” から 2 番目の文字 (“B”) を抽出します。

### SQL

```
SELECT $EXTRACT("ABCD",2)
```

以下の例は、from 値が “1” のとき、1-引数形式は 2-引数形式と等しいことを示しています。両方の \$EXTRACT 関数は、“H” を返します。

### SQL

```
SELECT $EXTRACT("HELLO")  
SELECT $EXTRACT("HELLO",1)
```

## メモ

### \$PIECE および \$LIST と比較した \$EXTRACT

\$EXTRACT は、文字列からの整数位置により、部分文字列を返します。\$PIECE と \$LIST は両方とも、特別にフォーマットされた文字列を処理します。

\$PIECE は、文字列内に区切り文字が使用されている標準文字の文字列から、部分文字列を返します。

\$LIST は、要素（文字ではない）の整数位置により、エンコードされたリストから要素のサブリストを返します。\$LIST は、通常の文字列に対して使用できません。また、\$EXTRACT は、エンコードされたリストに対して使用できません。

\$EXTRACT、\$FIND、\$LENGTH、および \$PIECE 関数は、標準文字の文字列に対して処理を実行します。さまざまな \$LIST 関数は、エンコードされた文字列を操作します。この文字列は、標準の文字列とは互換性がありません。唯一の例外は、\$LISTGET 関数と、引数が 1 つおよび 2 つの形式の \$LIST 関数です。これらの関数は、入力としてエンコードされた文字の文字列を受け取り、単一要素値を標準文字の文字列として出力します。

### \$EXTRACT と Unicode

\$EXTRACT 関数は、バイトではなく、文字を操作します。したがって、以下の例で示すように、Unicode の文字列は ASCII 文字列と同様に処理されます。以下の埋め込み SQL の例では、pi (\$CHAR(960)) に対して Unicode 文字を使っています。

## ObjectScript

```
SET a="QT PIE"
SET b=("QT " _$CHAR(960))
&sql(SELECT
$EXTRACT(:a,-33,4),
$EXTRACT(:a,4,4),
$EXTRACT(:a,4,99),
$EXTRACT(:b,-33,4),
$EXTRACT(:b,4,4),
$EXTRACT(:b,4,99)
INTO :a1,:a2,:a3,:b1,:b2,:b3)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"ASCII form returns ",!,a1,!,a2,!,a3
    WRITE !,"Unicode form returns ",!,b1,!,b2,!,b3 }
```

## NULL と無効な引数

- ・ string が NULL 文字列の場合は、NULL 文字列が返されます。
- ・ from が文字列の長さよりも大きい場合は、NULL 文字列が返されます。
- ・ from がゼロまたは負の数のときに to が指定されていない場合は、NULL 文字列が返されます。
- ・ to がゼロ、負の数、または from よりも小さな数の場合は、NULL 文字列が返されます。
- ・ to が有効な値の場合は、from はゼロまたは負の数でもかまいません。\$EXTRACT は、この from 値を 1 として処理します。

無効な引数値に対して、SQLCODE エラーは生成されません。

以下の例は、負の from 値が 1 として評価されるため、\$EXTRACT は先頭から 7 番目の文字までの部分文字列 “THIS IS” を返します。

## SQL

```
SELECT $EXTRACT('THIS IS A TEST',-7,7)
```

以下の埋め込み SQL の例にある \$EXTRACT 関数呼び出しは、すべて NULL 文字列を返します。

## ObjectScript

```
SET a="THIS IS A TEST"
SET b=""
&sql(SELECT
$EXTRACT(:a,33),
$EXTRACT(:a,-7),
$EXTRACT(:a,3,2),
$EXTRACT(:a,-7,0),
$EXTRACT(:a,-7,-10),
$EXTRACT(:b,-33,4),
$EXTRACT(:b,4,4),
$EXTRACT(:b,4,99),
$EXTRACT(NULL,-33,4),
$EXTRACT(NULL,4,4),
$EXTRACT(NULL,4,99)
INTO :a1,:a2,:a3,:a4,:a5,:b1,:b2,:b3,:c1,:c2,:c3)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"FROM too big: ",a1
    WRITE !,"FROM negative, no TO: ",a2
    WRITE !,"TO smaller than FROM: ",a3
    WRITE !,"TO not a positive integer: ",a4,a5
    WRITE !,"LIST is null string: ",b1,b2,b3,c1,c2,c3 }
```

## 関連項目

- ・ SQL 関数: [\\$FIND](#) [\\$LENGTH](#) [\\$LIST](#) [\\$LISTGET](#) [\\$PIECE](#)

- ・ ObjectScript 関数: [\\$EXTRACT](#) [\\$FIND](#) [\\$LENGTH](#) [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTGET](#) [\\$PIECE](#)

## \$FIND (SQL)

文字列内の部分文字列の終了位置を返す文字列関数です。オプションで検索開始位置を指定できます。

### 構文

```
$FIND(string,substring[,start])
```

### 説明

\$FIND は、文字列内の部分文字列の末尾位置を示す整数を返します。\$FIND は、substring を求めて string を検索します。substring が見つかったら、\$FIND は substring に続く最初の文字の位置を整数で返します。substring が見つからなければ、\$FIND は 0 の値を返します。

start オプションを使用して、検索の開始位置を指定できます。start 値が string 内の文字数よりも大きい場合、\$FIND は 0 の値を返します。start が省略されている場合は、文字列位置 1 が既定です。start がゼロ、負の数、または非数値文字列の場合は、位置 1 が既定です。

\$FIND は、大文字と小文字を区別します。大小文字変換関数の 1 つを使用して、文字列の大文字と小文字のインスタンスをどちらも配置します。

### \$FIND、POSITION、CHARINDEX、および INSTR

\$FIND、POSITION、CHARINDEX、および INSTR はすべて文字列内の指定された部分文字列を検索し、最初に一致した位置に対応する整数値を返します。\$FIND は、最初に一致した部分文字列の次の文字の整数位置を返します。CHARINDEX、POSITION、および INSTR は、一致した部分文字列の最初の文字の整数位置を返します。CHARINDEX、\$FIND、および INSTR では、部分文字列の検索を開始する位置を指定できます。INSTR では、その開始位置から数えて何個目の部分文字列かを指定することもできます。

以下に、4 つの関数にすべてのオプション引数を指定した例を示します。string および substring の位置はそれぞれの関数で異なります。

#### SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

部分文字列を検索する関数のリストは、“[文字列操作](#)”を参照してください。

### 引数

#### string

検索するターゲット文字列。変数名、数値、文字列リテラル、または任意の有効な式を指定できます。

#### substring

検索する部分文字列。変数名、数値、文字列リテラル、または任意の有効な式を指定できます。

### 起動

正の整数で指定する、部分文字列検索の開始地点を示す引数（オプション）。string の最初から文字数がカウントされ、カウントは 1 を基準とします。string の先頭から検索するには、この引数を省略するか、0 または 1 の start を指定します。負の数値、空の文字列、または数値以外の値は 0 として処理されます。start を NULL に指定すると、\$FIND は <null> を返します。

\$FIND は、SMALLINT [データ型](#)を返します。

## 例

以下の例では、string が文字列 “ABCDEFGH” で、substring が文字列 “BCD” です。\$FIND 関数は、“BCD” の次の文字 (“E”) の位置を示す値 5 を返します。

### SQL

```
SELECT $FIND('ABCDEFGH','BCD') AS SubPoint
```

以下の例は、数値 987654321 で数字 7 を検索し、substring の後の位置を示す 4 を返します。

### SQL

```
SELECT $FIND(987654321,7) AS SubPoint
```

以下の例は、substring “AA”: の最初のインスタンスの後の文字の位置を示す 3 を返します。

### SQL

```
SELECT $FIND('AAAAAA','AA') AS SubPoint
```

以下の例では、\$FIND は文字列に含まれていない部分文字列を検索します。結果はゼロ (0) を返します。

### SQL

```
SELECT $FIND('AABBCDD','AC') AS SubPoint
```

以下の例では、\$FIND は 7 番目の文字から検索を開始します。この例は、7 文字目以降での “R” の出現箇所の次の位置を示す 14 を返します。

### SQL

```
SELECT $FIND('EVERGREEN FOREST','R',7) AS SubPoint
```

以下の例では、\$FIND は文字列の最後の文字の後から、検索を開始します。結果はゼロ (0) を返します。

### SQL

```
SELECT $FIND('ABCDEFGH','G',10) AS SubPoint
```

以下の例は、1 よりも小さい start が 1 として処理されることを示しています。

### SQL

```
SELECT  
$FIND("ABCDEFGH","F"),  
$FIND("ABCDEFGH","F",1),  
$FIND("ABCDEFGH","F",0),  
$FIND("ABCDEFGH","F",-35)
```

以下の例は、\$FIND を使用して、Unicode 文字の pi、つまり \$CHAR(960) が含まれる文字列を検索しています。最初の \$FIND は、pi の次の文字の位置を示す 5 を返します。2 番目の \$FIND は、4 文字目から検索を開始し、4 文字目が検索文字の pi に該当するため、同様に 5 を返します。3 番目の \$FIND は 5 文字目から検索を開始し、5 文字目以降での pi の出現箇所の次の位置を示す 13 を返します。位置 12 が文字列の最後の文字であっても、位置 13 が返されることに注意してください。

### ObjectScript

```
SELECT  
$FIND("QT "._$CHAR(960)_" HONEY "._$CHAR(960),$CHAR(960)),  
$FIND("QT "._$CHAR(960)_" HONEY "._$CHAR(960),$CHAR(960),4),  
$FIND("QT "._$CHAR(960)_" HONEY "._$CHAR(960),$CHAR(960),5)
```

## 関連項目

- ・ [CHARINDEX 関数](#)
- ・ [INSTR 関数](#)
- ・ [POSITION 関数](#)
- ・ [文字列操作](#)



## FLOOR (SQL)

与えられた数値式に等しいか、より小さい最大整数値を返す、数値関数です。

### 構文

```
FLOOR(numeric-expression)  
{fn FLOOR(numeric-expression)}
```

### 概要

FLOOR は、*numeric-expression* 以下の最も近い整数値を返します。返される値は、小数桁数が 0 です。  
*numeric-expression* に NULL 値、空文字列 ("), または数値でない文字列を指定すると、FLOOR は NULL を返します。

FLOOR は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として呼び出せる点に注意してください。

この関数は、ObjectScript から FLOOR() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.FLOOR(numeric-expression)
```

### 引数

#### *numeric-expression*

上限値を計算する数。数はリテラルまたは文字列のいずれかになります。文字列として指定された数は、科学的記数法にできます。

*numeric-expression* が数値型である場合、FLOOR は *numeric-expression* と同じデータ型を返します。

### 例

以下の例では、FLOOR を使用して小数をその下限となる整数に変換する方法を示します。

#### SQL

```
SELECT FLOOR(167.111) AS FloorNum1,  
       FLOOR('167.456') AS FloorNum2,  
       FLOOR(167.999) AS FloorNum3,  
       FLOOR(167.0) AS FloorNum4
```

すべて 167 を返します。

#### SQL

```
SELECT FLOOR(-167.111) AS FloorNum1,  
       FLOOR(-167.456) AS FloorNum2,  
       FLOOR(-167.999) AS FloorNum3,  
       FLOOR(-168.0) AS FloorNum4
```

すべて -168 を返します。

以下の例では科学的記数法を使用しています。

#### SQL

```
SELECT FLOOR(10E-1) // returns 1  
SELECT FLOOR('-14E-4') // returns -1  
SELECT FLOOR('-10E-1') // returns -1
```

以下の例では、サブクエリを使用して、US Zip Codes (郵便番号) の大きなテーブルを、下限となる Latitude の整数ごとに 1 つの代表都市の郵便番号を含むテーブルに縮小します。

## SQL

```
SELECT City,State,FLOOR(Latitude) AS FloorLatitude
FROM (SELECT City,State,Latitude,FLOOR(Latitude) AS FloorNum
      FROM Sample.USZipCode)
GROUP BY FloorNum
ORDER BY FloorNum DESC
```

## 関連項目

- ・ [CEILING](#)
- ・ [ROUND](#)

## GETDATE (SQL)

現在のローカル日付とローカル時刻を返す日付/時刻関数です。

### 構文

```
GETDATE([precision])
```

### 引数

引数	説明
precision	オプション - 時刻の精度を秒の小数部の桁数として指定する正の整数。既定値は 0 (秒の小数部なし) です。この既定値は構成可能です。precision 値はオプションであり、括弧が必要となります。

### 概要

GETDATE は、このタイムゾーンの現在のローカル日付とローカル時刻をタイムスタンプとして返します。これはサマータイムなどのローカル時刻調整に合わせて調整されます。

GETDATE は、%TimeStamp データ型形式 (yyyy-mm-dd hh:mm:ss.fff) または %PosixTime データ型形式 (エンコードされた 64 ビットの符号付き整数) のいずれかでタイムスタンプを返すことができます。返されるタイムスタンプ形式は以下のルールによって決まります。

1. 現在のタイムスタンプがデータ型 %PosixTime のフィールドに対して指定されている場合、現在のタイムスタンプ値は POSIXTIME データ型形式で返されます。例として、WHERE PosixField=GETDATE() や INSERT INTO MyTable (PosixField) VALUES (GETDATE()) が挙げられます。
2. 現在のタイムスタンプがデータ型 %TimeStamp のフィールドに対して指定されている場合、現在のタイムスタンプ値は TIMESTAMP データ型形式で返されます。ODBC タイプは TIMESTAMP で、LENGTH は 16、PRECISION は 19 です。例として、WHERE TSField=GETDATE() や INSERT INTO MyTable (TSField) VALUES (GETDATE()) が挙げられます。
3. 現在のタイムスタンプがコンテキストなしで指定されている場合、現在のタイムスタンプ値は TIMESTAMP データ型形式で返されます。例として、SELECT GETDATE() が挙げられます。

既定の日付/時刻形式を変更するには、各種日付/時刻オプションで SET OPTION コマンドを使用します。

GETDATE は CURRENT\_TIMESTAMP の同義語で、Sybase や Microsoft SQL Server との互換性を確保するために用意されています。CURRENT\_TIMESTAMP および NOW 関数を使用して、現在のローカル日付とローカル時刻を TIMESTAMP または POSIXTIME のいずれかの形式のタイムスタンプとして返すこともできます。精度は CURRENT\_TIMESTAMP ではサポートされていますが、NOW ではサポートされていません。

現在の日付のみを返すには、CURDATE または CURRENT\_DATE を使用します。現在の時刻のみを返すには、CURRENT\_TIME または CURTIME を使用します。これらの関数では、データ型として DATE または TIME を使用します。これらの関数は、いずれも精度をサポートしません。TIME データ型および DATE データ型では、値は、\$HOROLOG 形式で整数として格納されます。それらは、表示形式または論理 (格納) 形式のどちらかで表示できます。日付および時刻のデータ型は、CAST または CONVERT 関数を使用して変更できます。

### 協定世界時 (UTC)

InterSystems SQL のタイムスタンプ関数、日付関数、および時刻関数は、GETUTCDATE を除いてすべて、ローカル・タイム・ゾーン設定に固有のものとなります。GETUTCDATE は、現在の UTC (ユニバーサル) の日付と時刻を TIMESTAMP 値または POSIXTIME 値のいずれかとして返します。ObjectScript の \$ZTIMESTAMP 特殊変数を使用して、ユニバーサルな (タイム・ゾーンに依存しない) 現在のタイムスタンプを取得することもできます。

## 秒の小数部の精度

GETDATE では、小数点以下 9 桁までの精度の値を返すことができます。返される精度の桁数は precision 引数を使用して設定されます。precision 引数の既定値は、以下の方法で構成できます。

- ・ **SET OPTION** で TIME\_PRECISION オプションを使用します。
- ・ システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DefaultTimePrecision。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[ ] が表示されます。既定値は 0 です。
- ・ 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GETDATE()、CURRENT\_TIME、および CURRENT\_TIMESTAMP の既定の時間精度] の現在の設定を表示して編集します。

precision 引数には 0 ～ 9 の整数を指定します。既定値は 0 です。実際に返される精度はプラットフォームに依存し、システムで使用可能な精度を超えた precision の桁はゼロとして返されます。

秒の小数部は、丸められずに、常に指定された精度に切り捨てられます。

## 例

レポートの設計では、GETDATE は、レポートが作成されるたびに、そのときの時刻/日付を印刷するために使用できます。また GETDATE は、トランザクションが発生した時刻のログ記録など、活動を追跡するのに便利です。

GETDATE は、SELECT 文の選択リスト内、またはクエリの WHERE 節内で使用できます。以下の例は、現在の日付と時刻を TIMESTAMP 形式で返します。

### SQL

```
SELECT GETDATE() AS DateTime
```

以下の例は、2 桁の精度で現在の日付と時刻を返します。

### SQL

```
SELECT GETDATE(2) AS DateTime
```

以下の例は、ローカル・タイムスタンプ (タイム・ゾーン依存) とユニバーサル・タイムスタンプ (タイム・ゾーン非依存) を比較します。

### ObjectScript

```
SELECT GETDATE(), GETUTCDATE()
```

以下の例は、Orders テーブルの指定された行の LastUpdate フィールドに、現在のシステム日付と時刻を設定します。LastUpdate がデータ型 %TimeStamp である場合、GETDATE は現在の日付と時刻を ODBC タイムスタンプとして返します。LastUpdate がデータ型 %PosixTime である場合、GETDATE は現在の日付と時刻をエンコードされた 64 ビットの符号付き整数として返します。

### SQL

```
UPDATE Orders SET LastUpdate = GETDATE()
WHERE Orders.OrderNumber=:ord
```

CREATE TABLE で GETDATE を使用すると、特定のフィールドに既定値を指定できます。以下の例では、CREATE TABLE 文は GETDATE を使用して、StartDate フィールドに対して既定値を設定します。

## SQL

```
CREATE TABLE Employees(  
    EmpId      INT NOT NULL,  
    LastName   CHAR(40) NOT NULL,  
    FirstName  CHAR(20) NOT NULL,  
    StartDate  TIMESTAMP DEFAULT GETDATE())
```

## 関連項目

- ・ SQL の概念: [データ型](#)、[日付/時刻文](#)
- ・ SQL タイムスタンプ関数: [CAST](#)、[CONVERT](#)、[CURRENT\\_TIMESTAMP](#)、[GETUTCDATE](#)、[NOW](#)、[SYSDATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)、[TO\\_POSIXTIME](#)、[TO\\_TIMESTAMP](#)
- ・ SQL 現在日時関数: [CURDATE](#)、[CURRENT\\_DATE](#)、[CURRENT\\_TIME](#)、[CURTIME](#)
- ・ ObjectScript: [\\$ZDATETIME](#) 関数、[\\$HOROLOG](#) 特殊変数、[\\$ZTIMESTAMP](#) 特殊変数

# GETUTCDATE (SQL)

現在の UTC 日付と UTC 時刻を返す日付/時刻関数です。

## 構文

```
GETUTCDATE([precision])
```

## 説明

GETUTCDATE は、UTC (協定世界時) 日付と UTC 時刻をタイムスタンプとして返します。UTC 時刻は世界共通で、ローカル・タイム・ゾーンに依存しておらず、ローカル時刻調整 ([サマータイム](#)など) の影響は受けないため、異なるタイム・ゾーン間のユーザが同一データベースにアクセスする場合に一貫したタイムスタンプを適用する際にこの関数は便利です。

GETUTCDATE は、%TimeStamp データ型形式 (yyyy-mm-dd hh:mm:ss.fff) または %PosixTime データ型形式 (エンコードされた 64 ビットの符号付き整数) のいずれかで [タイムスタンプ](#) を返すことができます。返されるタイムスタンプ形式は以下のルールによって決まります。

1. 現在の UTC のタイムスタンプがデータ型 %PosixTime のフィールドに対して指定されている場合、このタイムスタンプ値は POSIXTIME データ型形式で返されます。例として、WHERE PosixField=GETUTCDATE() や INSERT INTO MyTable (PosixField) VALUES (GETUTCDATE()) が挙げられます。
2. 現在の UTC のタイムスタンプがデータ型 %TimeStamp のフィールドに対して指定されている場合、このタイムスタンプ値は TIMESTAMP データ型形式で返されます。ODBC タイプは TIMESTAMP で、LENGTH は 16、PRECISION は 19 です。例として、WHERE TSField=GETUTCDATE() や INSERT INTO MyTable (TSField) VALUES (GETUTCDATE()) が挙げられます。
3. 現在の UTC のタイムスタンプがコンテキストなしで指定されている場合、このタイムスタンプ値は TIMESTAMP データ型形式で返されます。例として、SELECT GETUTCDATE() が挙げられます。

既定の日付/時刻形式を変更するには、各種日付/時刻オプションで [SET OPTION](#) コマンドを使用します。

GETUTCDATE は、主に SELECT 文のセレクト・リストや、クエリの WHERE 節で使用されます。レポートの設計では、GETUTCDATE は、レポートが作成されるたびに、そのときの時刻/日付を印刷するために使用できます。また GETUTCDATE は、トランザクションが発生した時刻のログ記録など、活動を追跡するのに便利です。

GETUTCDATE は、フィールドの既定値を指定するために CREATE TABLE 内で使用できます。

## その他の SQL 関数

GETUTCDATE は、現在の UTC 日付と UTC 時刻を TIMESTAMP 形式または POSIXTIME 形式のタイムスタンプとして返します。

その他のタイムスタンプ関数は、ローカル日付とローカル時刻を返します。[GETDATE](#)、[CURRENT\\_TIMESTAMP](#)、[NOW](#)、および [SYSDATE](#) は、現在のローカル日付とローカル時刻を TIMESTAMP 形式または POSIXTIME 形式のタイムスタンプとして返します。

GETDATE と CURRENT\_TIMESTAMP では、precision 引数が用意されています。

NOW、引数なしの CURRENT\_TIMESTAMP、および SYSDATE では、precision 引数は用意されていません。これらの関数では、システム全体の既定の時刻精度が使用されます。

[CURDATE](#) と [CURRENT\\_DATE](#) は、現在のローカル日付を返します。[CURTIME](#) と [CURRENT\\_TIME](#) は、現在のローカル時刻を返します。これらの関数では、データ型として DATE または TIME を使用します。これらの関数は、いずれも精度をサポートしません。

TIMESTAMP データ型は、同じ形式で値を格納および表示します。POSIXTIME データ型は、エンコードされた 64 ビットの符号付き整数としてその値を格納します。TIME および DATE の各データ型では \$HOROLOG 形式の整数として値が格納されますが、その値はさまざまな形式で表示できます。

InterSystems SQL のタイムスタンプ関数は、GETUTCDATE を除いてすべて、それぞれのローカル・タイム・ゾーン設定に固有です。タイム・ゾーンに依存しない世界時による現在のタイムスタンプの取得には、ObjectScript の \$ZTIMESTAMP 特殊変数も使用できます。GETUTCDATE では precision を設定できますが、\$ZTIMESTAMP では常に 3 の精度が返されます。

## 秒の小数部の精度

GETUTCDATE では、小数点以下 9 桁までの精度の値を返すことができます。返される精度の桁数は precision 引数を使用して設定されます。precision 引数の既定値は、以下の方法で構成できます。

- ・ **SET OPTION** で TIME\_PRECISION オプションを使用します。
- ・ システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DefaultTimePrecision。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[ ] が表示されます。既定値は 0 です。
- ・ 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GET-DATE(), CURRENT\_TIME, CURRENT\_TIMESTAMP のデフォルトの時刻精度] の現在の設定を表示して編集します。

返される精度の小数点以下の桁数の既定値に 0 ～ 9 の整数を指定します。既定値は 0 です。実際に返される精度はプラットフォームに依存し、システムで使用可能な精度を超えた precision の桁はゼロとして返されます。

秒の小数部は、丸められずに、常に指定された精度に切り捨てられます。

## 引数

### precision

時刻の精度を秒の小数部の桁数として指定する正の整数 (オプション)。既定値は 0 (秒の小数部なし) です。この既定値は構成可能です。

## 例

以下の例は、現在の日付と時刻を TIMESTAMP 形式の UTC タイムスタンプ、およびローカル・タイムスタンプとして返します。

### SQL

```
SELECT GETUTCDATE() AS UTCDateTime,
       GETDATE() AS LocalDateTime
```

以下の例は、現在の UTC 日付と時刻で、秒の小数部を 2 桁の精度として返します。

### SQL

```
SELECT GETUTCDATE(2) AS DateTime
```

以下の例は、ローカル・タイムスタンプ (タイム・ゾーン依存) とユニバーサル・タイムスタンプ (タイム・ゾーン非依存) を比較します。

### SQL

```
SELECT GETDATE(), GETUTCDATE()
```

以下の例は、Orders テーブルの指定された行の LastUpdate フィールドに、現在の UTC 日付と UTC 時刻を設定します。LastUpdate がデータ型 %TimeStamp である場合、GETUTCDATE は現在の UTC 日付と UTC 時刻を ODBC タイ



ムスタンプとして返します。LastUpdate がデータ型 %PosixTime である場合、GETUTCDATE は現在の UTC 日付と UTC 時刻をエンコードされた 64 ビットの符号付き整数として返します。

## SQL

```
UPDATE Orders SET LastUpdate = GETUTCDATE()  
WHERE Orders.OrderNumber=:ord
```

以下の例では、CREATE TABLE 文は GETUTCDATE を使用して、OrderRcvd フィールドに対して既定値を設定します。

```
CREATE TABLE Orders(  
    OrderId      INT NOT NULL,  
    ItemName     CHAR(40) NOT NULL,  
    Quantity     INT NOT NULL,  
    OrderRcvd    TIMESTAMP DEFAULT GETUTCDATE())
```

## 関連項目

- ・ SQL の概念: [データ型](#)、[日付/時刻文](#)
- ・ SQL タイムスタンプ関数 : [CAST](#)、[CONVERT](#)、[CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[NOW](#)、[SYSDATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)、[TO\\_POSIXTIME](#)、[TO\\_TIMESTAMP](#)
- ・ SQL 現在日時関数: [CURDATE](#)、[CURRENT\\_DATE](#)、[CURRENT\\_TIME](#)、[CURTIME](#)
- ・ ObjectScript : [\\$ZDATETIME](#) 関数、[\\$HOROLOG](#) 特殊変数、[\\$ZTIMESTAMP](#) 特殊変数

## GREATEST (SQL)

一連の式から最大値を返す関数。

### 構文

```
GREATEST(expression, expression[, ...])
```

### 説明

GREATEST は、コンマで区切られた一連の式から最大値を返します。式は、左から順に評価されます。expression が 1 つしかない場合は、GREATEST はその値を返します。expression が NULL の場合、GREATEST は NULL を返します。

すべての expression 値がキャノニック形式の数値に解決される場合は、数値順に比較されます。引用符で囲まれた文字列にキャノニック形式の数値が含まれている場合は、数値順に比較されます。一方、引用符で囲まれた文字列にキャノニック形式以外の数値 ('00'、'0.4'、または '+4' など) が含まれている場合は、文字列として比較されます。右側にゼロが埋め込まれたキャノニック形式の数値 ('125.2500' など) は、数値順に比較されます。文字列の比較は、照合順に文字ごとに行われます。文字列値は、数値よりも大きくなります。

空の文字列は、任意の数値よりも大きいものの、他の文字列値よりは小さくなります。

戻り値が数値の場合、GREATEST は戻り値を (先頭と末尾のゼロが削除されるなど) キャノニック形式で返します。戻り値が文字列の場合は、GREATEST は、先頭または末尾の空白も含めてその文字列を変更せずに返します。

GREATEST は、コンマで区切られた一連の式から最大値を返します。LEAST は、コンマで区切られた一連の式から最小値を返します。COALESCE は、コンマで区切られた一連の式から最初の非 NULL 値を返します。

### 戻り値のデータ型

expression 値のデータ型が異なる場合、返されるデータ型は可能なすべての戻り値と最も互換性の高い型、つまり最も高いデータ型の優先順位を持つデータ型になります。例えば、ある expression が整数で別の expression が小数の場合、GREATEST はデータ型 NUMERIC の値を返します。これは、NUMERIC が両方に互換性がある、最も高い優先順位を持つデータ型であるためです。ただし、expression がリテラル値または文字列である場合、GREATEST はデータ型 VARCHAR を返します。

### 引数

#### expression

数値または文字列に解決される式。これらの式の値は、相互に比較されます。expression にはフィールド名、リテラル、算術式、ホスト変数、またはオブジェクト参照のいずれかを指定できます。最大で 140 個の式をコンマで区切って指定できます。

### 例

次の例では、各 GREATEST は 3 つのキャノニック形式の数値を比較しています。

#### SQL

```
SELECT GREATEST(22,2.2,-21) AS HighNum,  
       GREATEST('2.2','22','-21') AS HighNumStr
```

次の例では、各 GREATEST は、3 つの数値文字列を比較しています。ただし、各 GREATEST には、キャノニック形式以外の文字列が 1 つ含まれています。これらのキャノニック形式以外の値は、文字列として比較されます。文字列は数値よりも常に大きくなります。

## SQL

```
SELECT GREATEST('22', '+2.2', '-21'),  
       GREATEST('0.2', '22', '-21')
```

次の例では、各 GREATEST は 3 つの文字列を比較し、照合順序の最も高い値を返します。

## SQL

```
SELECT GREATEST('A', 'a', ''),  
       GREATEST('a', 'ab', 'abc'),  
       GREATEST('#', '0', '7'),  
       GREATEST('##', '00', '77')
```

次の例では、キャノニック形式の数値として扱われる、\$HOROLOG 整数としての生年月日と日付に変換される整数 58073 の 2 つの日付を比較しています。これは、21 世紀に生まれた各人の生年月日を返します。2000 年 1 月 1 日より前に生まれた人は、1999 年 12 月 31 日というデフォルトの生年月日で表示されます。

## SQL

```
SELECT Name, GREATEST(DOB, TO_DATE(58073)) AS NewMillenium  
FROM Sample.Person
```

## 関連項目

- ・ [LEAST](#) 関数
- ・ [COALESCE](#) 関数
- ・ [CONVERT](#) 関数
- ・ [TO\\_NUMBER](#) 関数

## HOUR (SQL)

日付/時刻式に対して時間を返す時刻関数です。

### 構文

```
{fn HOUR(time-expression)}
```

### 概要

HOUR は、指定された時刻または日付/時刻値に対して時間の部分を示す整数を返します。時間は、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプに基づいて計算されます。

*time-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

この既定の時刻形式を変更するには、[SET OPTION](#) コマンドを使用します。

時刻整数 (経過秒数) は指定できますが、時刻文字列 (hh:mm:ss) は指定できません。日付/時刻文字列 (yyyy-mm-dd hh:mm:ss) を指定する必要があります。日付/時刻文字列の秒 (:ss) または分および秒 (mm:ss) の部分を省略しても、時刻部分を返すことができます。日付/時刻文字列の時刻部分は有効な時刻値である必要があります。日付/時刻文字列の日付部分は検証されません。

時間は 24 時間法で表されます。時間 (hh) 部分は、0 から 23 までの範囲の整数である必要があります。入力では、先頭のゼロはオプションです。出力では、先頭のゼロは抑制されます。

時間部分が '0' または '00' の場合、HOUR は値 0 を返します。また、時刻式が指定されていない場合や、時刻文字列の時間部分が省略されている場合 (':mm:ss' または '::ss') やも、ゼロ時が返されます。

[DATEPART](#) または [DATENAME](#) を使用して、同じ時刻情報を取得できます。

この関数は、ObjectScript から HOUR() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.HOUR(time-expression)
```

### 引数

#### *time-expression*

列の名前や、他のスカラー関数の結果、または文字列や日付や数値リテラルである式。日付/時刻文字列または時刻整数に解決される必要があります。基本となるデータ型は、[%Time](#)、[%TimeStamp](#)、または [%PosixTime](#) とすることができます。

### 例

以下の 2 つの例は、*time-expression* 値が 18:45:38 なので、共に数値 18 を返します。

#### SQL

```
SELECT {fn HOUR('2017-02-16 18:45:38')} AS ODBCHour
```

#### SQL

```
SELECT {fn HOUR(67538)} AS HorologHour
```

以下の例も 18 を返します。時刻値の秒 (または分と秒) 部分は省略できます。

#### SQL

```
SELECT {fn HOUR('2017-02-16 18:45')} AS Hour_Given
```

以下の例は、日付/時刻文字列の時刻部分が省略されているので、ゼロ時を返します。

## SQL

```
SELECT {fn HOUR('2017-02-16')} AS Hour_Given
```

以下の例は、すべて現在の時刻の時間部分を返します。

## SQL

```
SELECT {fn HOUR(CURRENT_TIME)} AS H_CurrentT,
       {fn HOUR({fn CURTIME()})} AS H_CurT,
       {fn HOUR({fn NOW()})} AS H_Now,
       {fn HOUR($HOROLOG)} AS H_Horolog,
       {fn HOUR($ZTIMESTAMP)} AS H_ZTS
```

\$ZTIMESTAMP は協定世界時 (UTC) を返すことに注意してください。その他の time-expression 値はローカル時刻を返します。

以下の例は、先頭のゼロが抑制されることを示します。最初の HOUR 関数は長さ 2 を返し、その他は長さ 1 を返します。省略された時刻はゼロ時と見なされ、1 の長さを持ちます。

## SQL

```
SELECT LENGTH({fn HOUR('2018-02-15 11:45')}),
       LENGTH({fn HOUR('2018-02-15 03:45')}),
       LENGTH({fn HOUR('2018-02-15 3:45')}),
       LENGTH({fn HOUR('2018-02-15')})
```

以下の例では、HOUR 関数が、ロケールに指定された TimeSeparator 文字を認識していることを示しています。

## SQL

```
SELECT {fn HOUR('2018-02-16 18.45.38')}
```

## 関連項目

- SQL の概念: [データ型](#)、[日付/時刻文](#)
- SQL 関数: [MINUTE](#)、[SECOND](#)、[CURRENT\\_TIME](#)、[CURTIME](#)、[NOW](#)、[DATEPART](#)、[DATENAME](#)
- ObjectScript 関数: [\\$ZTIME](#)
- ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## IFNULL (SQL)

NULL テストを行い、適切な式を返す関数です。

### 構文

```
IFNULL(expression-1,expression-2 [,expression-3])
{fn IFNULL(expression-1,expression-2)}
```

### 概要

InterSystems IRIS は IFNULL を SQL 汎用関数と ODBC スカラ関数の両方としてサポートしています。双方は大変よく似た処理を実行しますが、機能上は異なります。SQL 汎用関数は、3 つの引数をサポートします。ODBC スカラ関数は、2 つの引数をサポートします。2 つの引数の SQL 汎用関数と ODBC スカラ関数の機能は同じではありません。expression-1 が NULL でない場合に異なる値を返します。

SQL 汎用関数は、expression-1 が NULL かどうかを評価します。expression-1 を返すことはありません。

- ・ expression-1 が NULL の場合は、expression-2 を返します。
- ・ expression-1 が NULL でない場合は、expression-3 を返します。
- ・ expression-1 が NULL でなく、expression-3 が指定されていない場合は、NULL を返します。

ODBC スカラ関数は、expression-1 が NULL かどうかを評価します。これは expression-1 または expression-2 のいずれかを返します。

- ・ expression-1 が NULL の場合は、expression-2 を返します。
- ・ expression-1 が NULL でない場合は、expression-1 を返します。

NULL の処理方法の詳細は “[NULL](#)” を参照してください。

### 返り値のデータ型

- ・ IFNULL(expression-1,expression-2) : expression-2 のデータ型を返します。expression-2 が数値リテラル、文字列リテラル、または NULL である場合、データ型 VARCHAR を返します。
- ・ IFNULL(expression-1,expression-2,expression-3) : expression-2 と expression-3 が異なるデータ型を持つ場合、より高い (より包括的な) データ型の優先順位を持つデータ型を返します。expression-2 または expression-3 が数値リテラルまたは文字列リテラルである場合、データ型 VARCHAR を返します。expression-2 または expression-3 が NULL である場合、非 NULL 引数のデータ型を返します。

expression-2 と expression-3 が異なる長さ、精度、または小数桁を持つ場合、IFNULL は2 つの式のうち、より大きい長さ、精度、または小数桁を返します。

- ・ {fn IFNULL(expression-1,expression-2)} : expression-1 のデータ型を返します。expression-1 が数値リテラル、文字列リテラル、または NULL である場合、データ型 VARCHAR を返します。

### DATE および TIME の表示変換

DATE や TIME データ型など、一部の expression-1 データ型では、論理モード (モード 0) から ODBC モード (モード 1) または表示モード (モード 2) への変換が必要です。expression-2 または expression-3 の値が同じデータ型でない場合は、この値を ODBC モードまたは表示モードに変換することはできません。DATE データ型の場合は SQLCODE エラー -146、TIME データ型の場合は SQLCODE エラー -147 が生成されます。例えば、IFNULL(DOB, 'nodate', DOB) は、ODBC モードまたは表示モードでは実行できません。SQLCODE -146 エラーが発行され、%msg が “

'nodate' ODBC/JDBC ” または ” 'nodate' ” に設定されます。ODBC モー

ドまたは表示モードでこの文を実行するには、値を適切なデータ型 `IFNULL(DOB,CAST('nodate' as DATE),DOB)` としてキャストする必要があります。これにより日付 0 になり、1840-12-31 として表示されます。

### %List の表示変換

**%List フィールド**は、エンコーディングが設定された文字列データ型です。expression-1 が %List フィールドである場合、expression-2 または expression-3 の適切な値は、選択モードによって異なります。

- ・ 論理モード (モード 0) または表示モード (モード 2) の場合、%List の値は、`$1b("element1","element2",...)` という形式の文字列データ型として返されます。したがって、以下の例に示すように、expression-2 または expression-3 の値は %List として指定する必要があります。

### SQL

```
SELECT TOP 20 Name,
IFNULL(FavoriteColors,$LISTBUILD('No Preference'),FavoriteColors) AS ColorChoice
FROM Sample.Person
```

- ・ ODBC モード (モード 1) では、%List の値は、`element1,element2,...` というコンマ区切り要素の文字列として返されます。したがって、以下の例に示すように、expression-2 または expression-3 の値は文字列として指定できます。

### ObjectScript

```
SELECT TOP 20 Name,
IFNULL(FavoriteColors'No Preference',FavoriteColors) AS ColorChoice
FROM Sample.Person
```

## 引数

### expression-1

NULL であるかどうか評価される式。

### expression-2

expression-1 が NULL の場合に返される式。

### expression-3

expression-1 が NULL でない場合に返される式 (オプション)。expression-3 が指定されないで expression-1 が NULL でない場合、NULL 値が返されます。

以下に、返されるデータ型について説明します。

## NULL を処理する関数の比較

以下の表は、さまざまな SQL 比較関数を示します。論理比較テストが True (A は B と同じ) の場合、各関数は特定の値を返し、False (A は B と同じではない) の場合、別の値を返します。これらの関数により、NULL の論理比較を実行できます。実際の **等値 (または非等値) 条件比較** で NULL を指定することはできません。



SQL 関数	比較テスト	返り値
IFNULL(ex1,ex2) [引数が 2 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、NULL を返す
IFNULL(ex1,ex2,ex3) [引数が 3 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、ex3 を返す
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
ISNULL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
NVL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
NULLIF(ex1,ex2)	ex1 = ex2	True の場合、NULL を返す False の場合、ex1 を返す
COALESCE(ex1,ex2,...)	各引数で ex = NULL	True の場合、次の ex 引数をテスト すべての ex 引数が True (NULL) の 場合、NULL を返す  False の場合、ex を返す

## 例

以下の例では、汎用関数と ODBC スカラ関数は、最初の式が NULL なので、どちらも 2 番目の式 (99) を返します。

### SQL

```
SELECT IFNULL(NULL,99) AS NullGen,{fn IFNULL(NULL,99)} AS NullODBC
```

以下の例では、汎用関数と ODBC スカラ関数は、異なる値を返します。汎用関数は、最初の式が NULL ではないので、<null> を返します。ODBC の例は、最初の式が NULL ではないので、最初の式 (33) を返します。

### SQL

```
SELECT IFNULL(33,99) AS NullGen,{fn IFNULL(33,99)} AS NullODBC
```

以下の例は、FavoriteColors が NULL の場合は文字列 'No Preference' を返し、そうでない場合は NULL を返します。

### SQL

```
SELECT Name,  
IFNULL(FavoriteColors,'No Preference') AS ColorChoice  
FROM Sample.Person
```

以下の例は、FavoriteColors が NULL の場合は文字列 'No Preference' を返し、そうでない場合は FavoriteColors の値を返します。

## SQL

```
SELECT Name,
IFNULL(FavoriteColors, 'No Preference', FavoriteColors) AS ColorChoice
FROM Sample.Person
```

以下の例は、FavoriteColors が NULL の場合は文字列 'No Preference' を返し、そうでない場合は文字列 'Preference' を返します。

## SQL

```
SELECT Name,
IFNULL(FavoriteColors, 'No Preference', 'Preference') AS ColorPref
FROM Sample.Person
```

以下の ODBC 構文の例は、FavoriteColors が NULL の場合は文字列 'No Preference' を返し、そうでない場合は FavoriteColors データ値を返します。

## SQL

```
SELECT Name,
      {fn IFNULL(FavoriteColors, $LISTBUILD('No Preference'))} AS ColorPref
FROM Sample.Person
```

## SQL

```
SELECT Name,
      {fn IFNULL(FavoriteColors, 'No Preference')} AS ColorChoice
FROM Sample.Person
```

以下の例では、WHERE 節で IFNULL を使用しています。これは、色の好みがない、21 歳未満の人を選択します。FavoriteColors が NULL の場合、IFNULL は Age フィールドの値を返し、その値が条件テストに使用されます。

## SQL

```
SELECT Name, FavoriteColors, Age
FROM Sample.Person
WHERE 21 > IFNULL(FavoriteColors, Age)
ORDER BY Age
```

類似の機能については、[NULL 述語](#) (IS NULL、IS NOT NULL) を参照してください。

## 関連項目

- ・ [CASE コマンド](#)
- ・ [COALESCE 関数](#)
- ・ [ISNULL 関数](#)
- ・ [NULLIF 関数](#)
- ・ [NVL 関数](#)
- ・ [NULL 述語](#)

## INSTR (SQL)

文字列内の部分文字列の位置を返す文字列関数です。オプションで、検索開始位置と、その位置から数えて何個目の部分文字列かを指定できます。

### 構文

```
INSTR(string,substring[,start[,occurrence]])
```

### 説明

INSTR は、string 内で substring を検索し、substring の最初の文字の位置を返します。string の先頭から何文字目であるかを示す整数として位置が返されます。substring が見つからない場合、0 (ゼロ) が返されます。いずれかの引数に NULL 値を渡すと、INSTR は NULL を返します。

INSTR では、部分文字列の検索の開始位置として start を指定できます。また、その開始位置から数えて部分文字列の occurrence を指定することもできます。

INSTR は、大文字と小文字を区別します。大小文字変換関数の 1 つを使用して、文字列の大文字と小文字のインスタンスをどちらも配置します。

この関数は、ObjectScript から INSTR() メソッド・コールを使用して呼び出すこともできます。

### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.INSTR("The broken brown briefcase","br",6,2)
```

### INSTR、CHARINDEX、POSITION、および \$FIND

INSTR、[CHARINDEX](#)、[POSITION](#)、および [\\$FIND](#) はすべて文字列内の指定された部分文字列を検索し、最初に一致した位置に対応する整数位置を返します。CHARINDEX、POSITION、および INSTR は、一致した部分文字列の最初の文字の整数位置を返します。\$FIND は、最初に一致した部分文字列の次の文字の整数位置を返します。CHARINDEX、\$FIND、および INSTR では、部分文字列の検索を開始する位置を指定することができます。INSTR では、その開始位置から数えて何個目の部分文字列かを指定することもできます。

以下に、4 つの関数にすべてのオプション引数を指定した例を示します。string および substring の位置はそれぞれの関数で異なります。

### SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,  
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,  
       $FIND('The broken brown briefcase','br',6) AS Find,  
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

部分文字列を検索する関数のリストは、[“文字列操作”](#) を参照してください。

### 引数

#### string

substring の検索が行われる文字列式。この表現は、列の名前、文字列リテラル、または他のスカラー関数の結果となります。基盤となったデータ型は、任意の文字型 (CHAR や VARCHAR2 など) で示すことができます。

#### substring

string 内での存在が想定される部分文字列。

## start

正の整数で指定される、部分文字列検索の開始地点を示す引数 (オプション)。string の最初から文字数がカウントされ、カウントは 1 を基準とします。string の先頭から検索するには、この引数を省略するか、1 の start を指定します。start 値に 0、空文字列、NULL、または数値以外の値を指定すると、INSTR は 0 を返します。start を負の数に指定すると、INSTR は <null> を返します。

## occurrence

start 位置から検索する際にどの substring を返すかを指定するゼロ以外の整数 (オプション)。既定では、最初に検出された部分文字列の位置が返されます。

INSTR は、INTEGER [データ型](#)を返します。

## 例

以下の例は、“b” が文字列の 11 番目の文字であるため 11 を返します。

### SQL

```
SELECT INSTR('The quick brown fox','b',1) AS PosInt
```

以下の例は、Sample.Person テーブル内の各名前に対する姓 (苗字) の長さを返します。残りの名前フィールドから姓を区切るために使用するコンマを配置し、次にその位置から 1 を減算します。

### SQL

```
SELECT Name,
INSTR(Name,',',1)-1 AS LNameLen
FROM Sample.Person
```

以下の例では、Sample.Person テーブル内の各名前にある文字 “B” の最初のインスタンスの位置を返します。INSTR は大文字と小文字を区別するため、%SQLUPPER 関数を使用して、検索を実行する前にすべての名前の値を大文字に変換します。%SQLUPPER は文字列の最初に空白スペースを加えるため、この例では 1 を減算して、実際の文字位置を取得します。指定した文字列を配置できないと、検索はゼロ (0) を返します。この例では、1 を減算しているので、これらの検索で表示される値は -1 です。

### SQL

```
SELECT Name,
INSTR(%SQLUPPER(Name),'B',1)-1 AS BPos
FROM Sample.Person
```

## 関連項目

- ・ [CHARINDEX](#) 関数
- ・ [\\$FIND](#) 関数
- ・ [POSITION](#) 関数
- ・ [文字列操作](#)

## %INTERNAL (SQL)

LOGICAL 形式の式を返す形式変換関数です。

### 構文

```
%INTERNAL(expression)
```

```
%INTERNAL expression
```

### 概要

%INTERNAL は、現在選択されているモード (表示モード) に関係なく、*expression* を LOGICAL 形式に変換します。LOGICAL 形式は、データのメモリ内形式です (処理が実行される形式)。通常、%INTERNAL は、SELECT リストの *select-item* で使用されます。

%INTERNAL は、WHERE 節内で使用できますが、%INTERNAL を使用すると指定されたフィールドでインデックスを使用できなくなり、%INTERNAL は、フィールドに既定の照合がある場合でも、すべての比較で無条件に大文字小文字を区別するので、使用しないことを強くお勧めします。

%INTERNAL を適用すると、列見出し名が “Expression\_1” などの値に変更されます。したがって、後述の例で示すように、通常は列名のエイリアスを指定することをお勧めします。

%INTERNAL は、データ型 %Date の値を INTEGER データ型の値に変換します。%INTERNAL は、データ型 %Time の値を NUMERIC (15,9) データ型の値に変換します。この変換が行われるのは、ODBC または JDBC のクライアントで InterSystems IRIS の論理 %Date 値および %Time 値が認識されないためです。

%INTERNAL による変換にかかわらず、日付は、日付フィールドまたは関数によって返されるデータ型に依存しています。%INTERNAL は、[CURDATE](#)、[CURRENT\\_DATE](#)、[CURTIME](#)、および [CURRENT\\_TIME](#) の値を変換します。[CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)、および [\\$HOROLOG](#) の値は変換しません。

ストリーム・フィールドは、すべての形式変換関数を含めて、ObjectScript の単項関数に対する引数に指定することはできません。ただし、%INTERNAL は例外です。%INTERNAL 関数では、*expression* の値として[ストリーム・フィールド](#)が許可されますが、そのストリーム・フィールドに対する操作は実行されません。

%INTERNAL は、InterSystems SQL の拡張です。

現在の選択モードに関係なく *expression* を DISPLAY 形式に変換するには、[%EXTERNAL](#) 関数を使用します。現在の選択モードに関係なく *expression* を ODBC 形式に変換するには、[%ODBCOUT](#) 関数を使用します。

表示形式オプションの詳細は、“[データ表示オプション](#)” を参照してください。

### 引数

#### *expression*

変換される式。フィールド名、フィールド名を含む式、または変換可能なデータ型で値を返す関数 (DATE や %List など)。

### 例

以下の例では、DOB (誕生日) データ値を現在の選択モード形式で返し、%INTERNAL 関数を使用して同じデータを返します。

#### SQL

```
SELECT TOP 5 DOB,%INTERNAL(DOB) AS IntDOB
FROM Sample.Person
```

以下の例では、この関数の2つの構文形式を示します。それ以外は同じです。%List フィールドの %EXTERNAL (DISPLAY 形式)、%INTERNAL (LOGICAL 形式)、および %ODBCOUT (ODBC 形式) を指定します。

## SQL

```
SELECT TOP 10 %EXTERNAL(FavoriteColors) AS ExtColors,  
              %INTERNAL(FavoriteColors) AS IntColors,  
              %ODBCOUT(FavoriteColors) AS ODBCColors  
FROM Sample.Person
```

## SQL

```
SELECT TOP 10 %EXTERNAL FavoriteColors AS ExtColors,  
              %INTERNAL FavoriteColors AS IntColors,  
              %ODBCOUT FavoriteColors AS ODBCColors  
FROM Sample.Person
```

## 関連項目

- ・ [%EXTERNAL](#)、[%ODBCIN](#)、[%ODBCOUT](#)
- ・ SQL の概念 : [データ型](#)、[日付/時刻文](#)

## ISNULL (SQL)

NULL テストを行い、適切な式を返す関数です。

### 構文

```
ISNULL(check-expression,replace-expression)
```

### 引数

引数	説明
<i>check-expression</i>	評価される式。
<i>replace-expression</i>	<i>check-expression</i> が NULL の場合に返される式。

ISNULL が返す値のデータ型は、*check-expression* のデータ型と同じです。

### 概要

ISNULL は *check-expression* を評価し、2 つの値のうち 1 つを返します。

- ・ *check-expression* が NULL の場合は、*replace-expression* を返します。
- ・ *check-expression* が NULL でない場合は、*check-expression* を返します。

*replace-expression* のデータ型は、*check-expression* のデータ型と互換性がある必要があります。

ISNULL 関数は NVL 関数と同じであり、Oracle との互換性のために提供されているという点に注意してください。

NULL の処理方法の詳細は “[NULL](#)” を参照してください。

### DATE および TIME の表示変換

*check-expression* のデータ型によっては (DATE、TIME データ型など)、論理モードから ODBC モードまたは表示モードへの変換が必要になります。*replace-expression* の値が同じデータ型でない場合は、この値を ODBC モードまたは表示モードに変換することはできません。DATE データ型の場合は SQLCODE エラー -146、TIME データ型の場合は SQLCODE エラー -147 が生成されます。例えば、ISNULL(DOB, 'nodate') は、ODBC モードまたは表示モードでは実行できません。SQLCODE -146 エラーが発行され、%msg が “ 'nodate' ODBC/JDBC ” または “ 'nodate' ” に設定されます。ODBC モードまたは表示モードでこの文を実行するには、値を適切なデータ型 ISNULL(DOB, CAST('nodate' as DATE)) としてキャストする必要があります。これにより日付 0 になり、1840-12-31 として表示されます。

### NULL を処理する関数の比較

以下の表は、さまざまな SQL 比較関数を示します。論理比較テストが True (A は B と同じ) の場合、各関数は特定の値を返し、False (A は B と同じではない) の場合、別の値を返します。これらの関数により、NULL の論理比較を実行できます。実際の等値 (または非等値) 条件比較で NULL を指定することはできません。



SQL 関数	比較テスト	返り値
ISNULL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
IFNULL(ex1,ex2) [引数が 2 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、NULL を返す
IFNULL(ex1,ex2,ex3) [引数が 3 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、ex3 を返す
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
NVL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
NULLIF(ex1,ex2)	ex1 = ex2	True の場合、NULL を返す False の場合、ex1 を返す
COALESCE(ex1,ex2,...)	各引数で ex = NULL	True の場合、次の ex 引数をテスト すべての ex 引数が True (NULL) の 場合、NULL を返す  False の場合、ex を返す

## 例

以下の例では、最初の ISNULL は、最初の式が NULL なので、2 番目の式 (99) を返します。2 番目の ISNULL の例は、最初の式が NULL ではないので、最初の式 (33) を返します。

### SQL

```
SELECT ISNULL(NULL,99) AS IsNullT,ISNULL(33,99) AS IsNullF
```

以下の例は、FavoriteColors が NULL の場合は文字列 'No Preference' を返し、そうでない場合は FavoriteColors の値を返します。

### SQL

```
SELECT Name,
ISNULL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

ISNULL の動作を IFNULL と比較します。

### SQL

```
SELECT Name,
IFNULL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

## 関連項目

- ・ [CASE コマンド](#)
- ・ [COALESCE 関数](#)
- ・ [IFNULL 関数](#)
- ・ [NULLIF 関数](#)
- ・ [NVL 関数](#)

# ISNUMERIC (SQL)

正しい数字かどうかをテストする数値関数。

## 構文

`ISNUMERIC(check-expression)`

## 概要

ISNUMERIC は *check-expression* を評価し、以下の値のいずれかを返します。

- ・ *check-expression* が有効な数字であれば 1 を返します。有効な数は、数値式、または有効な数を表す文字列のいずれかです。
  - － 数値式はまず**キャノニック形式**に変換され、複数の先頭の符号を解決します。したがって、`+-+++34` などの数値式は有効な数になります。
  - － 数値文字列は、評価前に変換されることはありません。数値文字列は、有効な数として評価されるには、先頭の符号が最大で 1 つでなければなりません。末尾の小数点がある数値文字列は、有効な数として評価されます。
- ・ *check-expression* が有効な数字でなければ 0 を返します。非数値文字を含む文字列は、有効な数字ではありません。先頭に複数の符号がある数値文字列 (`'+-+++34'` など) は、有効な数として評価されません。InterSystems IRIS のエンコード・リストは、その要素が有効な数であっても、常に 0 を返します。空の文字列 `ISNUMERIC('')` は 0 を返します。
- ・ *check-expression* が NULL の場合、NULL を返します。`ISNUMERIC(NULL)` は NULL を返します。

科学的記数法の指数が 308 (308 - (整数の数 - 1)) より大きい場合、ISNUMERIC では、指数が範囲外であることを示す SQLCODE -7 エラーが生成されます。例えば `ISNUMERIC(1E309)` や `ISNUMERIC(111E307)` は、どちらもこのエラー・コードを生成します。指数数値文字列が `'1E145'` 以下である場合は 1 を返し、`'1E145'` より大きい場合は 0 を返します。

ISNUMERIC 関数は ObjectScript の \$ISVALIDNUM 関数とよく似ています。しかし、入力値が NULL の場合、2 つの関数は異なる値を返します。

## 引数

### check-expression

評価される式。

## 例

以下の例では、ISNUMERIC 関数はすべて 1 を返します。

### SQL

```
SELECT ISNUMERIC(99) AS MyInt,
       ISNUMERIC('-99') AS MyNegInt,
       ISNUMERIC('-0.99') AS MyNegFrac,
       ISNUMERIC('-0.00') AS MyNegZero,
       ISNUMERIC('-0.09'+7) AS MyAdd,
       ISNUMERIC('5E2') AS MyExponent
```

以下の例では、FavoriteColors が NULL の場合は NULL を返し、それ以外の場合は FavoriteColors が数値フィールドでないため 0 を返します。

## SQL

```
SELECT Name,  
ISNUMERIC(FavoriteColors) AS ColorPref  
FROM Sample.Person
```

## 関連項目

- ・ [IFNULL](#) 関数
- ・ [ISNULL](#) 関数
- ・ [NULLIF](#) 関数
- ・ ObjectScript 関数: [\\$ISVALIDNUM](#)

## JSON\_ARRAY (SQL)

JSON 配列としてデータを返す変換関数です。

### 構文

```
JSON_ARRAY(expression [,expression][,...]
  [NULL ON NULL | ABSENT ON NULL])
```

### 説明

JSON\_ARRAY は式または (より一般的には) コンマ区切りの式リストを取得して、これらの値を含む JSON 配列を返します。JSON\_ARRAY は、SELECT 文で他のタイプの選択項目と組み合わせることができます。JSON\_ARRAY は、WHERE 節など、SQL 関数を使用できる他の場所に指定できます。

返される JSON 配列の形式は、以下のとおりです。

```
[ element1 , element2 , element3 ]
```

JSON\_ARRAY は、文字列 (二重引用符で囲まれます) または数字として、それぞれの配列要素値を返します。数字はキャノニック形式で返されます。数値文字列は、リテラルとして、二重引用符で囲まれて返されます。その他のデータ・タイプ (日付、\$List など) はすべて文字列として返されます。

JSON\_ARRAY では、テーブル内のすべてのフィールドを指定するための方法としてアスタリスク (\*) 構文を使用することはできません。COUNT(\*) 集約関数は使用できます。

返される JSON 配列の列には、Expression (既定) というラベルが付けられます。JSON\_ARRAY には列のエイリアスを指定できます。

### 選択モードおよび照合

現在の %SelectMode プロパティによって、返される JSON 配列値のフォーマットが決まります。選択モードを変更することによって、すべての日付要素および %List 要素が、その選択モードのフォーマットの文字列として JSON 配列に組み込まれます。

フォーマット変換関数 (%EXTERNAL、%INTERNAL、%ODBCIN、%ODBCOUT) を JSON\_ARRAY 内の個別のフィールド名に適用することによって、現在の選択モードをオーバーライドできます。フォーマット変換関数を JSON\_ARRAY に適用しても、JSON 配列の要素は文字列であるため、影響はありません。

照合関数を JSON\_ARRAY 内の個別のフィールド名または JSON\_ARRAY 全体に適用できます。

- JSON\_ARRAY に割り当てた照合関数は JSON 配列のフォーマットの後に照合を適用します。したがって、%SQLUPPER(JSON\_ARRAY(f1,f2)) は、すべての JSON 配列の要素の値を大文字に変換します。%SQLUPPER(JSON\_ARRAY(f1,f2)) は、配列の要素の前ではなく、JSON 配列の前にスペースを挿入します。そのため、数字は文字列として解析されません。
- JSON\_ARRAY 内の要素に適用した照合関数はその照合を適用します。そのため、JSON\_ARRAY('Abc',%SQLUPPER('Abc')) は [ "Abc", " ABC" ] を返し (先頭のスペースに注意)、JSON\_ARRAY(007,%SQLSTRING(007)) は [ 7, " 7" ] を返します。%SQLUPPER は、値の前にスペースを挿入するので、一般に、LCASE や UCASE などのケース変換関数を指定することが推奨されます。照合は要素と配列全体の両方に適用できます。%SQLUPPER(JSON\_ARRAY('Abc',%SQLSTRING('Abc')) ) は [ "ABC", " ABC" ] を返します。

### ABSENT ON NULL

オプションの ABSENT ON NULL キーワード句を指定すると、NULL (または NULL リテラル) の列値は、JSON 配列に含まれなくなります。JSON 配列にプレースホルダは含まれません。この結果、JSON 配列に含まれる要素の数が違って

くる可能性があります。例えば、以下のプログラムが返す JSON 配列では、一部のレコードは 3 番目の配列要素は Age になり、他のレコードでは 3 番目の要素は FavoriteColors になります。

```
SELECT JSON_ARRAY(%ID,Name,FavoriteColors,Age ABSENT ON NULL) FROM Sample.Person
```

キーワード句を指定しない場合、既定は NULL ON NULL になり、NULL は null という単語で (引用符による区切りなしで)、コンマ区切りの配列要素として表示されます。したがって、JSON\_ARRAY 関数から返される JSON 配列はすべて、同じ数の配列要素を持つことになります。

## 引数

### expression

式、または式のコンマ区切りのリスト。これらの式には、列名、集約関数、算術式、リテラル、およびリテラル NULL を含めることができます。

### ABSENT ON NULL/NULL ON NULL

返された JSON 配列での NULL 値の表現方法を指定するキーワード句 (オプション)。NULL ON NULL (既定) は、NULL (存在しない) データを null という言葉 (引用符なし) で表します。ABSENT ON NULL は、NULL データを JSON 配列から省略します。プレースホルダのコンマは残しません。このキーワード句は、空の文字列値に影響を与えません。

## 例

以下の例は、JSON\_ARRAY を適用して、コンマ区切りのフィールド値リストを持つ JSON 配列をフォーマットします。

### SQL

```
SELECT TOP 3 JSON_ARRAY(%ID,Name,Age,Home_State) FROM Sample.Person
```

以下の例は、JSON\_ARRAY を適用して、Name フィールド値を含む単一の要素を持つ JSON 配列をフォーマットします。

### SQL

```
SELECT TOP 3 JSON_ARRAY(Name) FROM Sample.Person
```

以下の例は、JSON\_ARRAY を適用して、リテラルおよびフィールド値を含む JSON 配列をフォーマットします。

### SQL

```
SELECT TOP 3 JSON_ARRAY('Employee from',%TABLENAME,Name,SSN) FROM Sample.Employee
```

以下の例は、JSON\_ARRAY を適用して、NULL およびフィールド値を含む JSON 配列をフォーマットします。

### SQL

```
SELECT JSON_ARRAY(Name,FavoriteColors) FROM Sample.Person  
WHERE Name %STARTSWITH 'S'
```

以下の例は、JSON\_ARRAY を適用して、結合されたテーブルのフィールド値を含む JSON 配列をフォーマットします。

### SQL

```
SELECT TOP 3 JSON_ARRAY(E.%TABLENAME,E.Name,C.%TABLENAME,C.Name)  
FROM Sample.Employee AS E,Sample.Company AS C
```

以下のダイナミック SQL の例では ODBC %SelectMode が設定されます。これは、JSON 配列の値を含むすべてのフィールドを表す方法を決定します。クエリは、%EXTERNAL フォーマット変換関数を適用することによって、特定の JSON 配列の要素に対応するこの選択モードをオーバーライドします。

## ObjectScript

```

SET myquery = 3
SET myquery(1) = "SELECT TOP 8 DOB,JSON_ARRAY(Name,DOB,FavoriteColors) AS ODBCMode, "
SET myquery(2) = "JSON_ARRAY(Name,DOB,%EXTERNAL(DOB),%EXTERNAL(FavoriteColors)) AS ExternalTrans
"
SET myquery(3) = "FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
WRITE "SelectMode is ODBC",!
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)}
DO rset.%Display()
WRITE !,"End of data"

```

以下の例は、JSON\_ARRAY を WHERE 節で使用して、OR 構文を使用しないで複数の列で Contains テストを実行します。

## SQL

```

SELECT Name,Home_City,Home_State FROM Sample.Person
WHERE JSON_ARRAY(Name,Home_City,Home_State) [ 'X'

```

## 関連項目

- ・ [SELECT 文](#)
- ・ [WHERE 節](#)
- ・ [JSON\\_OBJECT 関数](#)
- ・ [IS JSON の述語条件](#)
- ・ [述語の概要](#)
- ・ [データベースの問い合わせ](#)



## JSON\_OBJECT (SQL)

JSON オブジェクトとしてデータを返す変換関数です。

### 構文

```
JSON_OBJECT(key:value [,key:value][,...]  
[NULL ON NULL | ABSENT ON NULL])
```

### 説明

JSON\_OBJECT はコンマ区切りの key:value ペアのリスト (例 : 'mykey':colname) を取得して、これらの値を含む JSON オブジェクトを返します。key 名には、任意の文字列を一重引用符で囲んで指定できます。JSON\_OBJECT は key 名に名前付け規約や一意性チェックを適用しません。value には、列名またはその他の式を指定できます。

JSON\_OBJECT は、SELECT 文で他のタイプの選択項目と組み合わせることができます。JSON\_OBJECT は、WHERE 節など、SQL 関数を使用できる他の場所に指定できます。

返される JSON オブジェクトの形式は、以下のとおりです。

```
{ "key1" : "value1" , "key2" : "value2" , "key3" : "value3"  
}
```

JSON\_OBJECT は、文字列 (二重引用符で囲まれます) または数字として、オブジェクト値を返します。数字はキャノニク形式で返されます。数値文字列は、リテラルとして、二重引用符で囲まれて返されます。その他のすべてのデータ型 (例えば、日付や \$List) は、文字列として返され、現在の %SelectMode によって返される値の形式が決まります。DISPLAY または ODBC モードがクエリに選択されている場合、JSON\_OBJECT は key と value の両方の値を DISPLAY または ODBC モードで返します。

JSON\_OBJECT では、テーブル内のすべてのフィールドを指定するための方法としてアスタリスク (\*) 構文を使用することはできません。

返される JSON オブジェクトの列には、Expression (既定) というラベルが付けられます。JSON\_OBJECT には列のエイリアスを指定できます。

### 選択モードおよび照合

現在の %SelectMode プロパティによって、返される JSON オブジェクト値のフォーマットが決まります。選択モードを変更することによって、すべての日付値および %List 値が、その選択モードのフォーマットの文字列として JSON オブジェクトに組み込まれます。フォーマット変換関数 (%EXTERNAL、%INTERNAL、%ODBCIN、%ODBCOUT) を JSON\_OBJECT 内の個別のフィールド名に適用することによって、現在の選択モードをオーバーライドできます。フォーマット変換関数を JSON\_OBJECT に適用しても、JSON オブジェクトの key:value ペアは文字列であるため、影響はありません。

既定の照合によって、返される JSON オブジェクト値の照合が決まります。照合関数を JSON\_OBJECT に適用し、キーと値の両方を変換できます。通常、キーは大文字と小文字を区別するため、照合関数を JSON\_OBJECT に割り当てないでください。InterSystems IRIS は JSON オブジェクトへのフォーマットの後で照合を適用します。したがって、%SQLUPPER(JSON\_OBJECT('k1':f1, 'k2':f2)) では、JSON オブジェクトの key および value 文字列がすべて大文字に変換されます。%SQLUPPER は、オブジェクト内の値の前ではなく、JSON オブジェクトの前にスペースを挿入します。

JSON\_OBJECT 内では、照合関数を key:value ペアの値部分に適用できます。%SQLUPPER は、値の前にスペースを挿入するので、一般に、LCASE や UCASE などのケース変換関数を指定することが推奨されます。

### ABSENT ON NULL

オプションの ABSENT ON NULL キーワード句を指定すると、NULL (または NULL リテラル) の列値は、JSON オブジェクトに含まれなくなります。JSON オブジェクトにプレースホルダは含まれません。この結果、JSON オブジェクトに含まれる key:value ペアの数が変わってくる可能性があります。例えば、以下のプログラムが返す JSON オブジェクトでは、一部

のレコードは 3 番目の key:value ペアは Age になり、他のレコードでは 3 番目の key:value ペアは FavoriteColors になります。

```
SELECT JSON_OBJECT('id':%ID,'name':Name,'colors':FavoriteColors,'years':Age ABSENT ON NULL) FROM Sample.Person
```

キーワード句を指定しない場合、既定は NULL ON NULL になり、NULL は null という単語で (引用符による区切りなしで)、key:value ペアの値として表示されます。したがって、JSON\_OBJECT 関数から返される JSON オブジェクトはすべて、同じ数の key:value ペアを持つことになります。

## 引数

### key:value

key:value ペアまたはコンマで区切られた key:value ペアのリスト。key はユーザが指定するリテラル文字列で、一重引用符で区切られます。value は、列名、集約関数、算術式、数値リテラル、文字列リテラル、またはリテラル NULL になります。

### ABSENT ON NULL/NULL ON NULL

返された JSON オブジェクトでの NULL 値の表現方法を指定するキーワード句 (オプション)。NULL ON NULL (既定) は、NULL (存在しない) データを null という言葉 (引用符なし) で表します。ABSENT ON NULL は NULL データを JSON オブジェクトから省略します。値が NULL の場合は key:value のペアを削除して、プレースホルダのコンマを残しません。このキーワード句は、空の文字列値に影響を与えません。

## 例

以下の例は、JSON\_OBJECT を適用して、フィールド値を含む JSON オブジェクトをフォーマットします。

### SQL

```
SELECT TOP 3 JSON_OBJECT('id':%ID,'name':Name,'birth':DOB) FROM Sample.Person
```

以下の例は、JSON\_OBJECT を適用して、リテラルおよびフィールド値を含む JSON オブジェクトをフォーマットします。

### SQL

```
SELECT TOP 3 JSON_OBJECT('lit':'Employee from','t':%TABLENAME,
    'name':Name,'num':SSN) FROM Sample.Employee
```

以下の例は、JSON\_OBJECT を適用して、NULL およびフィールド値を含む JSON オブジェクトをフォーマットします。

### SQL

```
SELECT JSON_OBJECT('name':Name,'colors':FavoriteColors) FROM Sample.Person
WHERE Name %STARTSWITH 'S'
```

以下の例は、JSON\_OBJECT を適用して、結合テーブルのフィールド値を含む JSON オブジェクトをフォーマットします。

### SQL

```
SELECT TOP 3 JSON_OBJECT('e.t':E.%TABLENAME,'e.name':E.Name,'c.t':C.%TABLENAME,
    'c.name':C.Name) FROM Sample.Employee AS E,Sample.Company AS C
```

以下の例は、JSON\_OBJECT を WHERE 節で使用して、OR 構文を使用しないで複数の列で Contains テストを実行します。

### SQL

```
SELECT Name,Home_City,Home_State FROM Sample.Person
WHERE JSON_OBJECT('name':Name,'city':Home_City,'state':Home_State) [ 'X'
```

以下の動的 SQL の例では ODBC %SelectMode が設定されます。これは、JSON オブジェクト値を含むすべてのフィールドを表す方法を決定します。このクエリは、%EXTERNAL フォーマット変換関数を適用することによって、特定の JSON\_OBJECT 値に対応するこの選択モードをオーバーライドします。

### ObjectScript

```
SET myquery = 3
SET myquery(1) = "SELECT TOP 8 JSON_OBJECT('ODBCBday':DOB,'DispBday':%EXTERNAL(DOB)), "
SET myquery(2) = "JSON_OBJECT('ODBCcolors':FavoriteColors,'DispColors':%EXTERNAL(FavoriteColors))"
"
SET myquery(3) = "FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
WRITE "SelectMode is ODBC",!
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)}
DO rset.%Display()
WRITE !,"End of data"
```

## 関連項目

- ・ [SELECT 文](#)
- ・ [WHERE 節](#)
- ・ [JSON\\_ARRAY 関数](#)
- ・ [IS JSON の述語条件](#)
- ・ [述語の概要](#)
- ・ [データベースの問い合わせ](#)

## \$JUSTIFY (SQL)

指定された幅の範囲内で値を右に寄せる関数です。オプションで、指定された小数桁数に丸めます。

### 構文

```
$JUSTIFY(expression,width[,decimal])
```

### 概要

\$JUSTIFY は、指定した width 内で expression で指定した値を右寄せして返します。width 内で小数点を揃えるために、decimal 引数を使用できます。

- ・ `$JUSTIFY(expression,width)`: 引数が 2 つの構文では、expression を width 内で右寄せします。expression の変換は実行されません。expression には、数値、または非数値文字列を使用できます。
- ・ `$JUSTIFY(expression,width,decimal)`: 引数が 3 つの構文では、expression を **キャノン形式の数値** に変換し、decimal まで小数桁を丸めるかゼロで埋めてから、結果となる数値を width 内で右寄せします。expression が非数値文字列または NULL の場合、InterSystems IRIS はそれを 0 に変換し、パディングしてから、右寄せします。

\$JUSTIFY は、現在のロケールのための DecimalSeparator 文字を認識します。必要に応じて DecimalSeparator 文字を追加または削除します。DecimalSeparator 文字はロケールによって異なります。一般に、米国形式のロケール用のピリオドかヨーロッパ形式のロケール用のコンマ (,) のどちらかとなります。ユーザのロケールの DecimalSeparator 文字を決定するには、以下のメソッドを呼び出します。

#### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

指定する引数が少なすぎると、SQLCODE -380 が発行されます。指定する引数が多すぎると、SQLCODE -381 が発行されます。

### \$JUSTIFY、ROUND、および TRUNCATE

固定の小数桁数への丸めが重要な場合、例えば金額を表す場合などは、\$JUSTIFY を使用します。これは、丸め処理の後に指定された数の末尾のゼロを返します。decimal が expression の小数桁数より大きい場合、\$JUSTIFY はゼロパディングを行います。また \$JUSTIFY では、DecimalSeparator 文字が数値列内で揃うように数値が右寄せされます。

また **ROUND** では、指定された小数桁数への丸めを行いますが、その戻り値は常に正規化され、末尾のゼロが削除されます。例えば、`ROUND(10.004,2)` は 10.00 ではなく 10 を返します。\$JUSTIFY とは異なり、ROUND では、丸め (既定) または切り捨てを指定できます。

**TRUNCATE** では、指定された小数桁数への切り捨てを行います。ROUND とは異なり、切り捨てによって末尾にゼロが生じた場合、これらの末尾のゼロは保持されます。ただし、\$JUSTIFY とは異なり、TRUNCATE はゼロパディングを行いません。

ROUND および TRUNCATE では、小数点記号の左に丸める (または切り捨てる) ことができます。例えば、`ROUND(128.5,-1)` は 130 を返します。

### \$JUSTIFY および LPAD

引数が 2 つの形式の LPAD および引数が 2 つの形式の \$JUSTIFY はどちらも、先頭にスペースを使用してパディングすることで文字列を右寄せします。引数が 2 つのこれらの形式の違いは、入力 expression の長さより短い出力 width の扱い方です。LPAD は、指定された出力長に合わせて入力文字列を切り捨てます。\$JUSTIFY は、入力文字列に合わせて出力長を拡大します。詳細は、以下の例を参照してください。

## SQL

```
SELECT '>' || LPAD(12345,10) || '<' AS lpadplus,
       '>' || $JUSTIFY(12345,10) || '<' AS justifyplus,
       '>' || LPAD(12345,3) || '<' AS lpadminus,
       '>' || $JUSTIFY(12345,3) || '<' AS justifyminus
```

引数が 3 つの形式の LPAD では、スペース以外の文字での左パディングが可能です。

## 引数

## expression

右寄せする値 (任意で、指定された小数桁数の数値として表します)。

- 文字列揃えが必要な場合は、decimal は指定しないでください。expression にはどの文字でも含めることができます。\$JUSTIFY は、“width” で説明しているように expression を右寄せします。
- 数値揃えが必要な場合は、decimal を指定します。decimal が指定されている場合、InterSystems IRIS は expression を \$JUSTIFY に **キャノニック形式の数値** として提供します。先頭のプラス符号とマイナス符号を解決し、先頭および末尾のゼロを削除します。最初の非数値文字列で expression を切り捨てます。expression の先頭が非数値文字列 (通貨記号など) の場合、InterSystems IRIS は expression の値を 0 に変換します。キャノニック形式への変換では、NumericGroupSeparator 文字、通貨記号、複数の DecimalSeparator 文字、または末尾のプラス符号やマイナス符号を認識しません。InterSystems IRIS での数値からキャノニック形式の数値への変換方法、InterSystems IRIS での非数値文字を含む数値文字列の扱い方の詳細は、“**数値**” を参照してください。

\$JUSTIFY は expression をキャノニック形式の数値として受け取った後、\$JUSTIFY はその演算を実行し、“width” で説明しているように、このキャノニック形式の数値を decimal 小数桁数まで丸めるかゼロでパディングしてから、その結果を右寄せします。

## width

変換された expression を右寄せする width。width が expression の長さ (数値と小数桁の変換後) より長い場合、InterSystems IRIS は width まで右寄せし、必要に応じて空白スペースで左パディングします。width が expression の長さ (数値と小数桁の変換後) より短い場合、InterSystems IRIS は width を expression 値の長さに設定します。

width には、正の整数を指定します。width 値 0、空文字列 (’ ’)、NULL、または非数値文字列では、width は 0 として扱われます。これは、InterSystems IRIS が width を expression 値の長さに設定することを意味します。

## decimal

小数桁数。expression に含まれる小数桁が多い場合は、\$JUSTIFY は小数部をこの数の小数桁まで丸めます。expression に含まれる小数桁が少ない場合、\$JUSTIFY は小数部をこの小数桁までゼロでパディングして、必要に応じて Decimal Separator 文字を追加します。decimal=0 の場合、\$JUSTIFY は expression を整数値まで丸め、Decimal Separator 文字を削除します。

expression 値が 1 未満の場合、\$JUSTIFY は DecimalSeparator 文字の前に 0 を挿入します。

**\$DOUBLE** の INF 値、-INF 値、および NAN 値は、decimal 値には関係なく \$JUSTIFY によって変更されることなく返されます。

## 例

以下の例では、文字列に対して右寄せを実行します。数値変換は実行されません。

## SQL

```
SELECT TOP 20 Age,$JUSTIFY(Name,18),DOB FROM Sample.Person
```

以下の例では、指定された小数桁数で数値の右寄せを実行します。

## SQL

```
SELECT TOP 20 $JUSTIFY(Salary,10,2) AS FullSalary,
$JUSTIFY(Salary/7,10,2) AS SeventhSalary FROM Sample.Employee
```

以下の例では、指定された小数桁数での数値の右寄せと、同じ数値の文字列の右寄せを実行します。

### ObjectScript

```
"SELECT $JUSTIFY({fn ACOS(-1)},8,3) AS ArcCos3,
$JUSTIFY({fn ACOS(-1)},8) AS ArcCosAll
```

以下のダイナミック SQL の例では、\$DOUBLE 値の INF および NAN を使用して数値の右寄せを実行します。

### ObjectScript

```
DO ##class(%SYSTEM.Process).IEEEEError(0)
SET x=$DOUBLE(1.2e500)
SET y=x-x
SET myquery = 2
SET myquery(1) = "SELECT $JUSTIFY(?,12,2) AS INFtest,"
SET myquery(2) = "$JUSTIFY(?,12,2) AS NANtest"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(x,y)
DO rset.%Display()
```

## 関連項目

- ・ [LPAD](#) 関数
- ・ [ROUND](#) 関数
- ・ [TRUNCATE](#) 関数

## LAST\_DAY (SQL)

日付式に対してその月の最後の日付を返す日付関数です。

### 構文

```
LAST_DAY(date-expression)
```

### 概要

LAST\_DAY は、指定した月の最後の日付を \$HOROLOGY 形式の整数として返します。うるう年の違いが計算されます (2000 年はうるう年で 1900 年と 2100 年はうるう年ではないという、1 世紀の日付調整など)。

*date-expression* には、InterSystems IRIS 日付整数、\$HOROLOGY 値や \$ZTIMESTAMP 値、ODBC 形式の日付文字列、またはタイムスタンプを指定できます。

*date-expression* タイムスタンプには、データ型 %Library.PosixTime (エンコードされた 64 ビットの符号付き整数) またはデータ型 %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

%TimeStamp 文字列の時刻部分はオプションです。

無効な日付を指定すると、LAST\_DAY は 0 (表示モードでは 12/31/1840) を返します。無効な日付とは、日または月が 0、12 より大きい月、または該当年の該当月の日数を上回る日付です。年の範囲は 0001 ~ 9999 です。

この関数は、ObjectScript から LASTDAY() メソッド・コールを使用して呼び出すこともできます。

### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.LASTDAY("2018-02-22"),!  
WRITE $SYSTEM.SQL.Functions.LASTDAY(64701)
```

### 引数

#### *date-expression*

列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式。

### 例

以下の例では、月の最後の日付を InterSystems IRIS 日付整数として返しています。この値が整数として表示されるか、日付文字列として表示されるかは、現在の SQL Display モード設定に依存します。

以下の 2 つの例は、指定した日付が含まれる月の最後の日が 2 月 29 日であるため (2004 年はうるう年です)、両方とも ('2004-02-29' に対応する) 59594 を返します。

#### SQL

```
SELECT LAST_DAY('2004-02-25')
```

#### SQL

```
SELECT LAST_DAY(59590)
```

以下の例はすべて、現在の月の最後の日付に相当する InterSystems IRIS 日付整数を返します。



## SQL

```
SELECT LAST_DAY({fn NOW()}) AS LD_Now,  
       LAST_DAY(CURRENT_DATE) AS LD_CurrDate,  
       LAST_DAY(CURRENT_TIMESTAMP) AS LD_CurrTstamp,  
       LAST_DAY($ZTIMESTAMP) AS LD_ZTstamp,  
       LAST_DAY($HOROLOG) AS LD_Horolog
```

## 関連項目

- ・ SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DAY](#)、[DAYOFYEAR](#)、[MONTH](#)、[YEAR](#)、[TO\\_DATE](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## LAST\_IDENTITY (SQL)

最後に挿入、更新、削除、またはフェッチされた行の ID を返すスカラー関数です。

### 構文

LAST\_IDENTITY()

### 概要

LAST\_IDENTITY 関数は、[%ROWID](#) ローカル変数値を返します。[%ROWID](#) ローカル変数は、[埋め込み SQL](#) または ODBC の値に設定されます。[%ROWID](#) ローカル変数は、ダイナミック SQL、SQL シェル、または管理ポータル of SQL インタフェースによる値には設定されません。ダイナミック SQL は、その代わりに [%ROWID](#) オブジェクト・プロパティを設定します。

LAST\_IDENTITY 関数は引数を取りません。引数の括弧は必須です。

LAST\_IDENTITY は、現在のプロセスによって最後に影響を受けた行の IDENTITY フィールドの値を返します。テーブルに IDENTITY フィールドがない場合は、現在のプロセスによって最後に影響を受けた行の行 ID ([%ROWID](#)) を返します。戻り値は INTEGER データ型です。

- 埋め込み SQL の [INSERT](#)、[UPDATE](#)、[DELETE](#)、または [TRUNCATE TABLE](#) 文の場合は、LAST\_IDENTITY は最後に変更された行の IDENTITY または [%ROWID](#) 値を返します。
- 埋め込み SQL の [カーソル・ベースの SELECT](#) 文の場合は、LAST\_IDENTITY は最後に取得された行の IDENTITY または [%ROWID](#) 値を返します。ただし、カーソル・ベースの SELECT 文に [DISTINCT](#) キーワードまたは [GROUP BY](#) 節が含まれる場合は、LAST\_IDENTITY は変更されず、前の値がある場合はそれを返します。
- 埋め込み SQL の単一行 (非カーソル) の SELECT 文の場合は、LAST\_IDENTITY は変更されません。前の値がある場合は、それが返されます。

プロセスの開始時に、LAST\_IDENTITY は NULL を返します。NEW [%RowID](#) に続けて、LAST\_IDENTITY は NULL を返します。

操作の影響を受けた行がない場合は、LAST\_IDENTITY は変更されません。LAST\_IDENTITY は、前の値がある場合は、それを返します。NEW [%RowID](#) に続けて、LAST\_IDENTITY を呼び出すと NULL が返されますが、[%ROWID](#) を呼び出すと [<UNDEFINED>](#) エラーが生成されます。

IDENTITY フィールドの詳細は、["CREATE TABLE"](#) を参照してください。["%ROWID"](#) も参照してください。

### 例

以下の例では、2 つの埋め込み SQL プログラムを使用して LAST\_IDENTITY を返しています。最初の例では新しいテーブル Sample.Students を作成し、2 番目の例ではこのテーブルにデータを入力してから、[カーソル・ベースの SELECT](#) をデータに対して実行して、それぞれの操作について LAST\_IDENTITY を返しています。

2 つの埋め込み SQL プログラムは、示されている順序で実行してください (埋め込み SQL では参照されるテーブルが既に存在していなければ INSERT 文をコンパイルすることができないため、ここでは 2 つの埋め込み SQL プログラムを使用する必要があります)。

#### ObjectScript

```
WRITE !,"Creating table"
&sql(CREATE TABLE Sample.Students (
    StudentName VARCHAR(30),
    StudentAge INTEGER,
    StudentID IDENTITY))
IF SQLCODE=0 {
    WRITE !,"Created table, SQLCODE=",SQLCODE }
ELSEIF SQLCODE=-201 {
    WRITE !,"Table already exists, SQLCODE=",SQLCODE }
```

## ObjectScript

```

WRITE !,"Populating table"
NEW %ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Students (StudentName,StudentAge)
      SELECT Name,Age FROM Sample.Person WHERE Age <= '21')
IF SQLCODE=0 {
  WRITE !,%ROWCOUNT," records added, last RowID is ",%ROWID,! }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }
&sql(SELECT LAST_IDENTITY()
      INTO :insertID
      FROM Sample.Students)
WRITE !,"INSERT Last Identity is: ",insertID,!
/* Cursor-based SELECT Query */
&sql(DECLARE C1 CURSOR FOR
      SELECT StudentName INTO :name FROM Sample.Students
      WHERE StudentAge = '17')
&sql(OPEN C1)
      QUIT:(SQLCODE'=0)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
  WRITE name," is seventeen",!
  &sql(FETCH C1) }
&sql(CLOSE C1)
WRITE !,%ROWCOUNT," records queried, last RowID is ",%ROWID,!
&sql(SELECT LAST_IDENTITY()
      INTO :qId)
WRITE !,"SELECT Last Identity is: ",qId,!
&sql(DROP TABLE Sample.Students)

```

## 関連項目

- ・ [INSERT、UPDATE、DELETE、TRUNCATE TABLE](#)
- ・ [DECLARE、OPEN、FETCH、CLOSE](#)
- ・ [埋め込み SQL](#)

## LCASE (SQL)

文字列内のすべての大文字を小文字に変換するケース変換関数です。

### 構文

```
LCASE(string-expression)  
{fn LCASE(string-expression)}
```

### 概要

LCASE は、表示目的で大文字を小文字に変換します。これは、アルファベットでない文字に影響を与えません。句読点および先頭と末尾の空白を変更しません。

LCASE は、数値を文字列として解釈する変換を強制的に実行しません。InterSystems SQL は、先頭と末尾のゼロを削除して、数値をキャノニック形式に変換します (数値文字列はキャノニック形式に変換されません)。

LOWER 関数も、大文字から小文字への変換に使用できます。

LCASE は、[照合](#)に影響しません。%SQLUPPER 関数は、大文字と小文字を区別しない照合に対してデータ値を変換するために SQL で優先的に使用される方法です。照合のケース変換の詳細は、“[%SQLUPPER](#)”を参照してください。

### 引数

#### *string-expression*

文字列式。その中の文字が小文字に変換されます。式は列の名前や文字列リテラル、または他のスカラー関数の結果を指定できます。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

### 例

以下の例は、各人の名前を小文字で返します。

#### SQL

```
SELECT TOP 10 Name, {fn LCASE(Name)} AS LowName  
FROM Sample.Person
```

また、LCASE は、ギリシャ文字を大文字から小文字に変換する以下の例で示すように、Unicode (非 ASCII) アルファベット文字でも動作します。

#### SQL

```
SELECT LCASE($CHAR(920,913,923,913,931,931,913))
```

### 関連項目

- SQL 関数: [LOWER](#)、[UCASE](#)
- ObjectScript 関数: [\\$ZCONVERT](#)

# LEAST (SQL)

一連の式から最小値を返す関数。

## 構文

```
LEAST(expression, expression[, ...])
```

## 説明

LEAST は、コンマで区切られた一連の式から最小値を返します。式は、左から順に評価されます。expression が 1 つしかない場合は、LEAST はその値を返します。expression が NULL の場合、LEAST は NULL を返します。

すべての expression 値がキャノニック形式の数値に解決される場合は、数値順に比較されます。引用符で囲まれた文字列にキャノニック形式の数値が含まれている場合は、数値順に比較されます。一方、引用符で囲まれた文字列にキャノニック形式以外の数値 ('00'、'0.4'、または '+4' など) が含まれている場合は、文字列として比較されます。右側にゼロが埋め込まれたキャノニック形式の数値 ('125.2500' など) は、数値順に比較されます。文字列の比較は、照合順に文字ごとに行われます。文字列値は、数値よりも大きくなります。

空の文字列は、任意の数値よりも大きいものの、他の文字列値よりは小さくなります。

返り値が数値の場合、LEAST は返り値を (先頭と末尾のゼロが削除されるなど) キャノニック形式で返します。返り値が文字列の場合は、LEAST は、先頭または末尾の空白も含めてその文字列を変更せずに返します。

LEAST は、コンマで区切られた一連の式から最小値を返します。[GREATEST](#) は、コンマで区切られた一連の式から最大値を返します。[COALESCE](#) は、コンマで区切られた一連の式から最初の非 NULL 値を返します。

## 返り値のデータ型

expression 値のデータ型が異なる場合、返されるデータ型は可能なすべての返り値と最も互換性の高い型、つまり最も高い**データ型の優先順位**を持つデータ型になります。例えば、ある expression が整数で別の expression が小数の場合、LEAST はデータ型 NUMERIC の値を返します。これは、NUMERIC が両方に互換性がある、最も高い優先順位を持つデータ型であるためです。ただし、expression がリテラル値または文字列である場合、LEAST はデータ型 VARCHAR を返します。

## 引数

### expression

数値または文字列に解決される式。これらの式の値は、相互に、および返される最小値と比較されます。expression にはフィールド名、リテラル、算術式、ホスト変数、またはオブジェクト参照のいずれかを指定できます。最大で 140 個の式をコンマで区切って指定できます。

## 例

次の例では、各 LEAST は 3 つのキャノニック形式の数値を比較しています。

### SQL

```
SELECT LEAST(22,2.2,-21) AS HighNum,
       LEAST('2.2','22','-21') AS HighNumStr
```

次の例では、各 LEAST は、3 つの数値文字列を比較しています。ただし、各 LEAST には、キャノニック形式以外の文字列が 1 つ含まれています。これらのキャノニック形式以外の値は、文字列として比較されます。文字列は数字よりも常に大きくなります。

## SQL

```
SELECT LEAST('22','+2.2','21'),  
       LEAST('0.2','22','21')
```

次の例では、各 LEAST は 3 つの文字列を比較し、照合順序の最も低い値を返します。

## SQL

```
SELECT LEAST('A','a',''),  
       LEAST('a','aa','abc'),  
       LEAST('#','0','7'),  
       LEAST('##','00','77')
```

次の例では、キャノニック形式の数値として扱われる、\$HOROLOG 整数としての生年月日と日付に変換される整数 58074 の 2 つの日付を比較しています。これは、20 世紀に生まれた各人の生年月日を返します。1999 年の 12 月 31 日より後に生まれた人は、2000 年 1 月 1 日というデフォルトの生年月日で表示されます。

## SQL

```
SELECT Name,LEAST(DOB,TO_DATE(58074)) AS NewMillenium  
FROM Sample.Person
```

## 関連項目

- ・ [GREATEST 関数](#)
- ・ [COALESCE 関数](#)
- ・ [CONVERT 関数](#)
- ・ [TO\\_NUMBER 関数](#)

# LEFT (SQL)

文字列式の最初（一番左）の文字から、指定された数の文字を返す文字列関数です。

## 構文

```
{fn LEFT(string-expression,count)}
```

## 概要

LEFT は、文字列の最初から指定された数の文字を返します。LEFT は、文字列を埋め込みません。文字列内の文字数よりも多い文字数を指定した場合、LEFT はその文字列を返します。引数に NULL を渡すと、LEFT は NULL を返します。

LEFT は、{} 括弧構文による ODBC スカラ関数としてのみ使用できます。

## 引数

### string-expression

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、(CHAR や VARCHAR など) 任意の文字タイプとすることができます。

### count

string-expression の最初から返す文字の数を指定する整数。

## 例

以下の例は、Sample.Person テーブル内の各名前の左端から 7 文字を返します。

### SQL

```
SELECT Name,{fn LEFT(Name,7)}AS ShortName
FROM Sample.Person
```

以下の例は、LEFT による文字列自体よりも長い count の処理方法を示しています。

### SQL

```
SELECT Name,{fn LEFT(Name,40)}
FROM Sample.Person
```

埋め込みは実行されません。

## 関連項目

[LTRIM](#) [RIGHT](#) [RTRIM](#)



## LEN (SQL)

---

文字列式の文字の数を返す文字列関数です。

### 構文

```
LEN(string-expression)
```

### 説明

LEN は、文字列式の文字数を返します。

LEN 関数は LENGTH 関数の別名です。LEN は、TSQL の互換性を持たせるために提供されています。詳細は ["LENGTH"](#) を参照してください。

### 引数

#### *string-expression*

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

LEN は、INTEGER [データ型](#)を返します。

### 関連項目

- ・ [LENGTH](#)

# LENGTH (SQL)

文字列式の文字の数を返す文字列関数です。

## 構文

```
LENGTH(string-expression)
{fn LENGTH(string-expression)}
```

## 概要

LENGTH は、与えられた文字列式について、そのバイト数ではなく、文字数を示す整数を返します。string-expression には、文字列 (末尾の空白は削除される)、または数値 (InterSystems IRIS によってキャノニック形式に変換される) を指定できます。

LENGTH は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として使用できる点に注意してください。

LENGTH と他の length 関数 (\$LENGTH、CHARACTER\_LENGTH、CHAR\_LENGTH、および DATALENGTH) はすべて以下の処理を実行します。

- LENGTH は、SelectMode の設定に関係なく、表示値ではなく、フィールドの論理 (内部データ・ストレージ) 値の長さを返します。すべての SQL 関数は常に、フィールドの内部ストレージ値を使用します。
- LENGTH は、数値の**キャノニック形式**の長さを返します。キャノニック形式の数値では、先頭と末尾にあるゼロ、先頭の符号 (1 つのマイナス記号を除く)、末尾の小数点記号文字が取り除かれます。LENGTH は、数値文字列の文字列長を返します。数値文字列はキャノニック形式に変換されません。
- LENGTH 関数では、文字列から先頭の空白は取り除かれませんが、LTRIM 関数を使用して、文字列から先頭の空白を削除できます。

以下の処理を実行する場合、LENGTH は他の length 関数 (\$LENGTH、CHARACTER\_LENGTH、CHAR\_LENGTH、および DATALENGTH) と異なります。

- LENGTH には、末尾の空白や文字列の終了文字は含まれません。  
\$LENGTH、CHARACTER\_LENGTH 関数、CHAR\_LENGTH 関数、および DATALENGTH 関数では、末尾の空白や終了文字は取り除かれませんが、
- LENGTH は NULL を渡すと NULL、空文字列を渡すと 0 を返します。  
CHARACTER\_LENGTH、CHAR\_LENGTH、および DATALENGTH も、NULL 値を渡すと NULL、空文字列を渡すと 0 を返します。\$LENGTH は NULL 値を渡すと 0、空文字列を渡すと 0 を返します。
- LENGTH はデータ・ストリーム・フィールドをサポートしません。string-expression にストリーム・フィールドを指定すると、SQLCODE -37 が返されます。  
また、\$LENGTH ではストリーム・フィールドもサポートされません。CHARACTER\_LENGTH 関数、CHAR\_LENGTH 関数、および DATALENGTH 関数では、データ・ストリーム・フィールドはサポートされません。

## 引数

### string-expression

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

## 例

以下の例では、InterSystems IRIS は最初に各数値をキャノニック形式に変換します（先頭および末尾のゼロを削除し、先頭の符号を解決し、末尾の小数点記号文字を削除します）。以下の各 LENGTH は、長さ 1 を返します。

### SQL

```
SELECT {fn LENGTH(7.00)} AS CharCount,
       {fn LENGTH(+007)} AS CharCount,
       {fn LENGTH(007.)} AS CharCount,
       {fn LENGTH(00000.00)} AS CharCount,
       {fn LENGTH(-0)} AS CharCount
```

以下の例では、1 つ目の LENGTH は先頭のゼロを削除し、長さの値として 2 を返します。2 つ目の LENGTH は数値を文字列として扱い、先頭のゼロは削除せず、長さの値として 3 を返します。

### SQL

```
SELECT LENGTH(0.7) AS CharCount,
       LENGTH('0.7') AS CharCount
```

以下の例は、値 12 を返します。

### SQL

```
SELECT LENGTH('INTERSYSTEMS') AS CharCount
```

以下の例では、LENGTH がどのように先頭および末尾の空白を扱うかを示します。1 つ目の LENGTH は、末尾の空白は取り除くが先頭の空白は取り除かないため、15 を返します。2 つ目の LENGTH は、LTRIM が先頭の空白を取り除くため、12 を返します。

### SQL

```
SELECT LENGTH('    INTERSYSTEMS    ') AS CharCount,
       LENGTH(LTRIM('    INTERSYSTEMS    ')) AS CharCount
```

以下の例は、Sample.Person テーブル内の各 Name 値の文字数を返します。

### SQL

```
SELECT Name, {fn LENGTH(Name)} AS CharCount
FROM Sample.Person
ORDER BY CharCount
```

以下の例は、DOB（誕生日）フィールド内の文字数を返します。（LENGTH、CHAR\_LENGTH、CHARACTER\_LENGTH によって）返される長さは、日付の内部 (\$HOROLOGY) 形式であり、表示形式ではありません。DOB の表示長は、10 文字です。3 つの長さ関数は、すべて内部長 5 を返します。

### SQL

```
SELECT DOB, {fn LENGTH(DOB)} AS LenCount,
       CHAR_LENGTH(DOB) AS CCount,
       CHARACTER_LENGTH(DOB) AS CtrCount
FROM Sample.Person
```

以下の埋め込み SQL の例は、Unicode 文字の文字列の長さを返します。返される長さは、文字数 (7) であり、バイト数ではありません。

## ObjectScript

```
SET a=$CHAR(920,913,923,913,931,931,913)
&sql(SELECT LENGTH(:a) INTO :b )
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The Greek Sea: ",a,!,$LENGTH(a),!,b }
```

## 関連項目

- ・ SQL 関数 : [CHAR\\_LENGTH](#)、[CHARACTER\\_LENGTH](#)、[DATALENGTH](#)、[LEN](#)、[\\$LENGTH](#)
- ・ ObjectScript 関数: [\\$LENGTH](#)

## \$LENGTH (SQL)

文字列内の文字数、または文字列内の区切られた部分文字列数を返す文字列関数です。

### 構文

```
$LENGTH(expression[, delimiter])
```

### 概要

\$LENGTH は使用される引数に応じて、指定した文字列にある文字数あるいは指定した文字列にある部分文字列数を返します。

- ・ \$LENGTH(expression) は、文字列の文字数を返します。expression が空文字列 (') の場合、\$LENGTH は 0 を返します。expression が NULL の場合、\$LENGTH は 0 を返します。
- ・ \$LENGTH(expression, delimiter) は、文字列内の部分文字列数を返します。\$LENGTH は、指定された delimiter で相互に分離されている部分文字列数を返します。この数は常に expression 文字列内で見つかった区切り文字インスタンスの数に 1 を加えたものです。

### \$LENGTH(expression) と他の Length 関数

\$LENGTH(expression) と他の length 関数 (LENGTH、CHARACTER\_LENGTH、CHAR\_LENGTH、および DATALENGTH) はすべて以下の処理を実行します。

- ・ \$LENGTH は、SelectMode の設定に関係なく、表示値ではなく、フィールドの論理 (内部データ・ストレージ) 値の長さを返します。すべての SQL 関数は常に、フィールドの内部ストレージ値を使用します。
- ・ \$LENGTH は、数値の**キャノニック形式**の長さを返します。キャノニック形式の数値では、先頭と末尾にあるゼロ、先頭の符号 (1 つのマイナス記号を除く)、末尾の小数点記号文字が取り除かれます。\$LENGTH は、数値文字列の文字列長を返します。数値文字列はキャノニック形式に変換されません。
- ・ \$LENGTH 関数では、文字列から先頭の空白は取り除かれませんが、**LTRIM** 関数を使用して、文字列から先頭の空白を削除できます。

以下の処理を実行する場合、\$LENGTH は他の length 関数 (LENGTH、CHARACTER\_LENGTH、CHAR\_LENGTH、および DATALENGTH) と異なります。

- ・ \$LENGTH 関数では、末尾の空白や終了文字は取り除かれませんが、CHARACTER\_LENGTH 関数、CHAR\_LENGTH 関数、および DATALENGTH 関数でも、末尾の空白や終了文字は取り除かれませんが、LENGTH では、末尾の空白や文字列の終了文字は取り除かれます。
- ・ \$LENGTH は NULL 値を渡すと 0、空文字列を渡すと 0 を返します。  
LENGTH、CHARACTER\_LENGTH、CHAR\_LENGTH、および DATALENGTH は、NULL 値を渡すと NULL、空文字列を渡すと 0 を返します。
- ・ \$LENGTH はデータ・ストリーム・フィールドをサポートしません。string-expression にストリーム・フィールドを指定すると、SQLCODE -37 が返されます。  
また、LENGTH ではストリーム・フィールドもサポートされませんが、CHARACTER\_LENGTH 関数、CHAR\_LENGTH 関数、および DATALENGTH 関数では、データ・ストリーム・フィールドはサポートされません。
- ・ \$LENGTH は、データ型 SMALLINT を返します。他の length 関数はすべて、データ型 INTEGER を返します。

## NULL および空文字列引数

\$LENGTH(expression) では、空文字列 (') と NULL (値が存在しない) は区別されません。これは、空文字列 (') と NULL のどちらの場合にも 0 の長さを返します。

NULL 以外の区切り文字を指定した \$LENGTH(expression,delimiter) は、一致がない場合は、区切り部分文字列カウント 1 を返します。完全な文字列は、区切り文字を含まない単一の部分文字列です。これは、expression が空文字列 (') である場合や、expression が NULL である場合も同じです。ただし、空文字列自体は一致し、値 2 を返します。

以下のテーブルでは、文字列 ('abc')、空文字列 (')、または NULL の expression 値とノンマッチ文字列 (''), 空文字列 ('), または NULL の delimiter 値のペアの可能な組み合わせを示します。

\$LENGTH(NULL) = 0	\$LENGTH('') = 0	\$LENGTH('abc') = 3
\$LENGTH(NULL,NULL) = 0	\$LENGTH('',NULL) = 0	\$LENGTH(' abc ',NULL) = 0
\$LENGTH(NULL,',') = 1	\$LENGTH('','') = 2	\$LENGTH(' abc ','') = 1
\$LENGTH(NULL,'^') = 1	\$LENGTH(' ','^') = 1	\$LENGTH('abc','^') = 1

## 引数

### expression

ターゲット文字列。数値、文字列リテラル、変数名、または任意の有効な式を指定できます。

### delimiter

ターゲット文字列にある部分文字列の境界を示す文字列 (オプション)。文字列リテラルを使用する必要がありますが、長さは自由です。また、引用符で囲む必要があります。

\$LENGTH は、SMALLINT [データ型](#) を返します。

## 例

以下の例は、文字列の長さ 6 を返します。

### SQL

```
SELECT $LENGTH('ABCDEG') AS StringLength
```

以下の例は、ドル記号 (\$) で区切られた文字列内の部分文字列数 3 を返します。

### SQL

```
SELECT $LENGTH('ABC$DEF$EFG','$') AS SubStrings
```

指定された区切り文字が文字列内に見つからない場合は、文字列全体が 1 つの部分文字列となるため、\$LENGTH は 1 を返します。

### SQL

```
SELECT $LENGTH('ABCDEG','$') AS SubStrings
```

以下の例では、最初の \$LENGTH 関数は a 内の文字数 11 を返します (スペース文字も含む)。2 番目の \$LENGTH 関数は、部分文字列の区切り文字に b のスペース文字を使用し、a の部分文字列数である 2 を返します。

### SQL

```
SELECT $LENGTH("HELLO WORLD"), $LENGTH("HELLO WORLD", " ")
```

以下の例は、テストされた文字列が NULL 文字列なので 0 を返します。

#### SQL

```
SELECT $LENGTH(NULL) AS StringLength
```

以下の例は、指定された区切り文字が見つからないので 1 を返します。この場合は、部分文字列が 1 つ (NULL 文字列) あることになります。

#### SQL

```
SELECT $LENGTH(NULL, '$') AS SubStrings
```

以下の例は、区切り文字列が NULL 文字列なので 0 を返します。

#### SQL

```
SELECT $LENGTH('ABCDEFG', NULL) AS SubStrings
```

## メモ

### \$LENGTH、\$PIECE、および \$LIST

- ・ 引数が 1 つの \$LENGTH は、文字列の文字数を返します。この関数は、位置を基準に部分文字列を特定し、その部分文字列値を返す \$EXTRACT 関数と共に使用できます。
- ・ 引数が 2 つの \$LENGTH は、区切り文字列に基づいて、文字列の部分文字列数を返します。この関数は、区切り文字を基準に部分文字列を特定し、その部分文字列値を返す \$PIECE 関数と共に使用できます。
- ・ \$LENGTH は、\$LISTBUILD または \$LIST を使用して作成された、エンコードされたリストには使用できません。エンコードされたリスト文字列の部分文字列 (リスト要素) 数を調べるには、\$LISTLENGTH を使用します。

\$LENGTH、\$FIND、\$EXTRACT、および \$PIECE 関数は、標準文字の文字列に対して処理を実行します。さまざまな \$LIST 関数は、エンコードされた文字列を操作します。この文字列は、標準の文字列とは互換性がありません。唯一の例外は、\$LISTGET 関数と、引数が 1 つおよび 2 つの形式の \$LIST 関数です。これらの関数は、入力としてエンコードされた文字の文字列を受け取り、単一要素値を標準文字の文字列として出力します。

## 関連項目

- ・ SQL 関数: [CHAR\\_LENGTH](#)、[CHARACTER\\_LENGTH](#)、[DATALENGTH](#)、[\\$EXTRACT](#)、[\\$FIND](#)、[LENGTH](#)、[\\$LIST](#)、[\\$LISTGET](#)、[\\$PIECE](#)
- ・ ObjectScript 関数: [\\$EXTRACT](#)、[\\$FIND](#)、[\\$LENGTH](#)、[\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTGET](#)、[\\$PIECE](#)



## \$LIST (SQL)

リスト内の要素を返すリスト関数です。

### 構文

```
$LIST(list[,position[,end]])
```

### 説明

\$LIST は、リストから要素を返します。返される要素は、使用する引数によって決まります。

- ・ \$LIST(list) は、リストの最初の要素をテキスト文字列として返します。
- ・ \$LIST(list,position) は、指定された位置にある要素をテキスト文字列として返します。ここで、position の 1 の値はリスト内の最初の要素を表します。position 引数は、必ず整数として評価される必要があります。
- ・ \$LIST(list,position,end) は、指定された開始 position から指定された end 位置までの範囲の、リストの要素を含む “sublist” (エンコードされたリスト文字列) を返します。

この関数は、データ型 VARCHAR を返します。

### 引数

#### list

1 つ以上の要素を含む、エンコードされた文字の文字列。リストは、SQL [\\$LISTBUILD](#) 関数または ObjectScript [\\$LISTBUILD](#) 関数を使用して作成できます。区切り文字列からリストへの変換は、SQL [\\$LISTFROMSTRING](#) 関数または ObjectScript [\\$LISTFROMSTRING](#) 関数を使用して行うことができます。既存のリストからのリストの抽出は、SQL [\\$LIST](#) 関数または ObjectScript [\\$LIST](#) 関数を使用して行うことができます。

#### position

返されるリスト要素の位置。リスト要素は、1 から始まります。position が省略されている場合は、最初の要素が返されます。position の値が 0、またはリストの要素の数よりも大きい場合、InterSystems SQL は値を返しません。position の値がマイナス -1 (1) の場合、\$LIST はリストの最後の要素を返します。例えば、以下のコマンドは “Green” を返します。

#### SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),-1)
```

end 引数が指定されている場合、position は要素範囲の最初の要素を指定します。(position と end が同じ値のとき) 要素が 1 つだけ返されますが、その要素はエンコードされた文字列として返されます。したがって、\$LIST(x,2) (要素を通常の文字列として返す) は \$LIST(x,2,2) (要素をエンコードされた文字列として返す) と同じではありません。

#### end

要素範囲の最後の要素の位置。end を指定するには、position を指定する必要があります。end が指定されている場合、返される値はエンコードされたリスト文字列です。このようにエンコードされているため、他の \$LIST 関数で処理する必要があります。

end の値によって、次のように処理されます。

- ・ position よりも大きい場合、要素のリストを含むエンコードされた文字列が返されます。
- ・ position と等しい場合、1 つの要素を含むエンコードされた文字列が返されます。
- ・ position よりも小さい場合、値は返されません。

- ・ list の要素数よりも大きい場合、リストの最後の要素を指定するのと同じです。
- ・ マイナス -1 (-1) のとき、リストの最後の要素を指定するのと同じです。

end を指定する場合、position 値はゼロ (0) に指定できます。この場合、0 は 1 と同じです。

## リストを使用した作業

1 つのテーブルにリスト・フィールドを 1 つ以上含めることができます。リストはエンコードされた文字列であるため、これらのフィールドはデータ型 **%List** (%Library.List) またはデータ型 VARCHAR として定義できます。データ型 **%List** のフィールドは、**CType (クライアント・データ型) = 6** として定義できます。

データ型は、挿入または更新の際にフィールドで許可される値を制限するものではありません。したがって、ユーザは、リスト・フィールド内のデータ値がすべてリスト・エンコードされた文字の文字列であることを確認する必要があります。SQL \$LIST 関数がエンコードされていない文字列のデータ値を検出した場合、SELECT 操作は SQLCODE -400 で失敗し、%msg は " : <LIST>%0AmBuncommitted+1^%sqlcq.USER.cls61.1" のようになります。

SQL の \$LIST 関数にリストを渡すには、ホスト変数を使用するか、SQL 内で \$LISTBUILD を指定します。以下の埋め込み SQL の例では、両方の方法を示しています。

### ObjectScript

```
SET mylist=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:mylist,2),$LIST($LISTBUILD('Red','Blue','Green'),3)
INTO :a,:b )
IF SQLCODE'=0 {
    WRITE "Error code ",SQLCODE,! }
ELSE {
    WRITE !,"The host variable list element is ",a,!
    WRITE !,"The SQL $LISTBUILD list element is ",b,! }
```

リストは、\$LIST 関数を使用して、別のリストから抽出することもできます。

### SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),2,3)
```

以下の埋め込み SQL の例では、sublist は有効な list 引数ではありません。これは通常 of 文字列として返された 1 つの要素であり、エンコードされたリスト文字列ではないためです。引数が 3 つの形式の \$LIST のみが、エンコードされたリスト文字列を返します。この場合は、SQLCODE -400 の致命的なエラーが発生します。

### ObjectScript

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,2)
INTO :sublist )
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    &sql(SELECT $LIST(:sublist,1)
INTO :c )
    IF SQLCODE'=0 {
        WRITE !,"Error code ",SQLCODE }
    ELSE {
        WRITE !,"The sublist is"
        ZZDUMP c ; Variable not set
    }
}
```

## 例

以下の例では、どちらのコマンドもリストの先頭要素である “Red” を返します。最初の文は、既定で先頭要素を返します。2 番目の文は、position 引数が 1 に設定されているために先頭要素を返します。

## SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"))
SELECT $LIST($LISTBUILD("Red","Blue","Green"),1)
```

以下の例は、リストで 2 つ目の要素である “Blue” を返します。

### ObjectScript

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),2)
```

以下の埋め込み SQL の例は、“Red Blue” を返します。“Red Blue” は先頭要素で開始して、2 つ目の要素で終了する 2 要素のリスト文字列です。WRITE ではなく ZZDUMP を使用しますが、これは特殊 (印字不可能) 文字をリストに含むためです。

### ObjectScript

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,1,2)
INTO :b )
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The encoded sublist is"
    ZZDUMP b ; Prints "Red Blue "
}
```

以下の例は、長さがわからないリストの最後の要素を返します。最初の SELECT 文では、最後の要素が通常の文字列として返され、2 番目の文では、これがエンコードされたリスト文字列として返されます。

## SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),-1)
SELECT $LIST($LISTBUILD("Red","Blue","Green"),$LISTLENGTH($LISTBUILD("Red","Blue","Green")), -1)
```

## メモ

### 無効な引数値

list 引数の式が有効なリストとして評価されない場合は、SQLCODE -400 の致命的なエラーが発生します。

## SQL

```
SELECT $LIST("the quick brown fox",1)
```

position 引数または end 引数の値が -1 よりも小さい場合は、SQLCODE -400 の致命的なエラーが発生します。

### ObjectScript

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),-2,3)
```

position 引数の値が存在しないリスト・メンバを参照するときに end 引数を使用されていない場合は、SQLCODE -400 の致命的なエラーが発生します。

## SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),7)
```

ただし、end 引数を使用されている場合は、エラーは発生せず、NULL 文字列が返されます。

## SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),7,-1)
```

position 引数の値が、値が未定義の要素を指す場合は、SQLCODE -400 の致命的なエラーが発生します。

## SQL

```
SELECT $LIST($LISTBUILD("Red",,"Green"),2)
```

## 引数が 2 つの \$LIST と引数が 3 つの \$LIST

\$LIST(list,1) と \$LIST(list,1,1) は同じではありません。前者は文字列を返すのに対し、後者は要素が 1 つのリスト文字列を返します。返す要素がない場合、引数が 2 つの形式は値を返さず、3 つの形式は NULL 文字列を返します。

## Unicode

リスト要素内に Unicode 文字が 1 つでもあれば、リスト要素全体が Unicode (ワイド) 文字として表されます。リスト内の他の要素は影響されません。

以下の埋め込み SQL の例は、2 つのリストを示しています。a リストは、ASCII 文字のみが含まれる 2 つの要素で構成されます。b リストは 2 つの要素で構成され、最初の要素には Unicode 文字 (\$CHAR(960)、つまり pi 記号) が含まれ、2 番目の要素には ASCII 文字のみが含まれます。

## ObjectScript

```
SET a=$LISTBUILD("ABC"_$CHAR(68),"XYZ")
SET b=$LISTBUILD("ABC"_$CHAR(960),"XYZ")
&sql(SELECT $LIST(:a,1),$LIST(:a,2),$LIST(:b,1),$LIST(:b,2)
INTO :a1,:a2,:b1,:b2 )
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The ASCII list a elements: "
    ZSDUMP a1
    ZSDUMP a2
    WRITE !,"The Unicode list b elements: "
    ZSDUMP b1
    ZSDUMP b2 }
```

InterSystems IRIS は、b の先頭要素全体をワイド Unicode 文字でエンコードすることに注意してください。b の 2 番目の要素には Unicode 文字が含まれないため、InterSystems IRIS は 1 バイトの ASCII 文字を使用してエンコードします。

## 関連項目

- SQL 関数: [\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$PIECE](#)
- ObjectScript 関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTNEXT](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$LISTVALID](#)

# \$LISTBUILD (SQL)

文字列からリストを作成するリスト関数。

## 構文

```
$LISTBUILD(element [,...])
```

## 説明

\$LISTBUILD は 1 つまたは複数の式を取り、1 つの式に対して 1 つの要素があるリストを返します。

リストの作成には、以下の関数を使用できます。

- ・ \$LISTBUILD は、複数の文字列から、要素ごとに 1 つの文字列を含むリストを作成します。
- ・ \$LISTFROMSTRING は、複数の区切られた要素を持つ 1 つの文字列からリストを作成します。
- ・ \$LIST は、既存のリストからサブリストを抽出します。

\$LISTBUILD は、他の InterSystems SQL リスト関数 (\$LIST、\$LISTDATA、\$LISTFIND、\$LISTFROMSTRING、\$LISTGET、\$LISTLENGTH、\$LISTTOSTRING) と共に使用されます。

**注釈** \$LISTBUILD とその他の \$LIST 関数は、最適化された 2 進数の表現を使用してデータ要素を格納します。このため、\$LIST データで予期されているような等価テストが利用できない場合もあります。他のコンテキストでは同等と見なされるデータが、異なる内部表現を持つこともあります。例えば、\$LISTBUILD(1) と \$LISTBUILD("1") は同じではありません。

同じ理由から、\$LISTBUILD によって返されたリスト文字列値は、\$PIECE や 2 つの引数形式の \$LENGTH など、区切り文字を使用する文字検索や関数の解析には使用するべきではありません。\$LISTBUILD で生成されたリストの要素は、区切り文字によってマークされないので、任意の文字を含むことができます。

## 引数

### element

任意の式、または式のコンマ区切りのリスト。

## 例

以下の例では、3 つの文字列を受け取り、3 つの要素を持つリストを作成します。

### SQL

```
SELECT $LISTBUILD("Red","White","Blue")
```

## メモ

### 引数の省略

要素式を省略すると、値が NULL の要素が生成されます。例えば、以下の例には 2 つの \$LISTBUILD 文がありますが、どちらも、3 つの要素を持つリストを生成し、2 つ目の要素には未定義 (NULL) 値が格納されます。

### SQL

```
SELECT $LISTBUILD("Red",,"Blue"), $LISTBUILD("Red",'',"Blue")
```

また、\$LISTBUILD 式が未定義のときは、対応するリスト要素が未定義の値となります。以下の例では 2 要素リストを生成します。リストの先頭要素は "Red" で 2 つ目の要素は未定義値です。

## SQL

```
SELECT $LISTBUILD('Red',:z)
```

以下の例では、2 要素のリストを生成します。末尾のコンマは、2 つ目の要素が未定義の値を持つことを示します。

## SQL

```
SELECT $LISTBUILD('Red',)
```

## 引数を指定しない

引数を指定しないで \$LISTBUILD 関数を呼び出すと、データ値が未定義の要素を 1 つ含むリストが返されます。これは、NULL と同じではありません。以下は、“空” のリストを作成する有効な \$LISTBUILD 文です。

## SQL

```
SELECT $LISTBUILD(), $LISTBUILD(NULL)
```

以下は、空文字列を含むリスト要素を作成する有効な \$LISTBUILD 文です。

## SQL

```
SELECT $LISTBUILD(''), $LISTBUILD(CHAR(0))
```

## リストの入れ子

リストの要素自体がリストである場合もあります。例えば、以下の文は 3 つ目の要素が "Walnut,Pecan" という 2 つの要素を持つリストである、3 つの要素を持つリストを作成します。

## SQL

```
SELECT $LISTBUILD('Apple','Pear',$LISTBUILD('Walnut','Pecan'))
```

## リストの連結

SQL 連結演算子 (||) を使用して 2 つのリストを連結すると、別のリストが生成されます。例えば、以下の SELECT 項目は同じリスト "A,B,C" を作成します。

## SQL

```
SELECT $LISTBUILD('A','B','C') AS List,
       $LISTBUILD('A','B') || $LISTBUILD('C') AS CatList
```

以下の例で、最初の 2 つの選択項目は同じ 2 つの要素を持つリストを生成し、3 つ目の選択項目は NULL となり (NULL は何と連結しても NULL になります)、4 つ目と 5 つ目の選択項目は同じ 3 つの要素を持つリストを生成します。

## SQL

```
SELECT
  $LISTBUILD('A','B') AS List,
  $LISTBUILD('A','B') || '' AS CatEStr,
  $LISTBUILD('A','B') || NULL AS CatNull,
  $LISTBUILD('A','B') || $LISTBUILD('') AS CatEList,
  $LISTBUILD('A','B') || $LISTBUILD(NULL) AS CatNList
```

## Unicode

リスト要素の 1 つ、または複数の文字がワイド (Unicode) 文字の場合、その式のすべての文字がワイド文字として表されます。複数のシステムにまたがる互換性を保証するために、\$LISTBUILD はハードウェア・プラットフォームに関係なく、これらのバイトを常に同じ順序で格納します。ワイド文字はバイト文字列として記述されます。詳細は、ObjectScript の “\$LISTBUILD” 関数を参照してください。

## 関連項目

- ・ SQL 関数 : [\\$LIST](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$PIECE](#)
- ・ ObjectScript 関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTNEXT](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$LISTVALID](#)

## \$LISTDATA (SQL)

指定された要素が存在し、データ値を持つかどうかを示す値を返すリスト関数です。

### 構文

```
$LISTDATA(list[,position])
```

### 説明

\$LISTDATA は、リスト内の指定された要素のデータを調べます。position 引数で指定された要素が list に存在し、データ値を持つ場合、\$LISTDATA は値 1 を返します。要素が list に存在しないか、またはデータ値を持たない場合、\$LISTDATA は値 0 を返します。

この関数は、データ型 SMALLINT を返します。

### 引数

#### list

1 つ以上の要素を含む、エンコードされた文字の文字列。リストは、SQL [\\$LISTBUILD](#) 関数または ObjectScript [\\$LISTBUILD](#) 関数を使用して作成できます。区切り文字列からリストへの変換は、SQL [\\$LISTFROMSTRING](#) 関数または ObjectScript [\\$LISTFROMSTRING](#) 関数を使用して行うことができます。既存のリストからのリストの抽出は、SQL [\\$LIST](#) 関数または ObjectScript [\\$LIST](#) 関数を使用して行うことができます。

#### position

position 引数を省略した場合、\$LISTDATA は先頭要素を評価します。position 引数の値が -1 の場合は、リストの最終要素を指定するのと同じ意味です。position 引数の値が存在しないリスト・メンバを参照する場合は、\$LISTDATA は 0 を返します。

### 例

以下の例では、position 引数にさまざまな値を指定した結果を示します。

以下のすべての \$LISTDATA 文は、1 を返します。

#### SQL

```
SELECT
  $LISTDATA($LISTBUILD("Red",,Y,"","Green")),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),1),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),4),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),5),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),-1)
```

以下の \$LISTDATA 文は、同じ 5 つの要素のリストに対して 0 の値を返します。

#### SQL

```
SELECT
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),2),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),3),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),0),
  $LISTDATA($LISTBUILD("Red",,Y,"","Green"),6)
```

### メモ

#### 無効な引数値

list 引数の式が有効なリストとして評価されない場合は、SQLCODE -400 の致命的なエラーが発生します。



## SQL

```
SELECT $LISTDATA('fred')
```

position 引数の値が -1 よりも小さい場合は、SQLCODE -400 の致命的なエラーが発生します。

## SQL

```
SELECT $LISTDATA($LISTBUILD("Red","Blue","Green"),-3)
```

position が数値以外の値の場合は、このエラーは発生しません。

## SQL

```
SELECT $LISTDATA($LISTBUILD("Red","Blue","Green"),'g')
```

## 関連項目

- ・ SQL 関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$PIECE](#)
- ・ ObjectScript 関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTNEXT](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$LISTVALID](#)

## \$LISTFIND (SQL)

指定されたリストで、要求された値を検索するリスト関数です。

### 構文

```
$LISTFIND(list,value[,startafter])
```

### 概要

\$LISTFIND は、指定された *list* で、要求された *value* の最初のインスタンスを検索します。startafter 引数で指定された位置の次の要素から検索を開始します。startafter 引数を省略すると、\$LISTFIND は startafter 引数値を 0 と見なし、先頭要素 (要素 1) から検索を開始します。値が見つかったら、\$LISTFIND は一致する要素の位置を返します。値が見つからないか、startafter 引数の値が存在しないリスト・メンバを参照する場合は、\$LISTFIND は 0 を返します。

この関数は、データ型 SMALLINT を返します。

### 引数

#### list

有効なリストとして評価される式。list は、1 つ以上の要素を含むエンコードされた文字列です。list を作成するには、SQL または ObjectScript の \$LISTBUILD 関数または \$LISTFROMSTRING 関数を使用します。既存のリストから list を抽出するには、SQL または ObjectScript の \$LIST 関数を使用します。

#### value

検索要素を含む式。これは文字列です。

#### startafter

リスト位置として解釈される整数式 (オプション)。この位置の後の要素から検索を開始します。ゼロと -1 が有効値で、-1 は要素を返しませんが、ゼロが既定です。

### 例

以下の例は、要求された文字列の最初に発生した位置である 2 を返します。

#### SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue')
```

以下の例は 0 を返し、要求された文字列が見つからなかったことを表します。

#### ObjectScript

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green'),'Orange')
```

以下の 3 つの例は、startafter 引数の実行結果です。最初の例は、要求された文字列が startafter 位置にあるため、要求した文字列が見つからず 0 を返します。

#### SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue',2)
```

2 番目の例は、startafter がゼロ (既定値) に設定されているため、要求された文字列が先頭位置にあるのを見つけます。

## SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Red',0)
```

3 番目の例は、要求された文字列の 1 つ目が startafter 位置より前にあるため、2 つ目を見つけて 5 を返します。

## SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green","Yellow","Blue"),'Blue',3)
```

\$LISTFIND 関数は、完全な要素のみと一致します。したがって、以下の例ではすべての要素に“B”が含まれているものの、リスト要素が文字列“B”とは同じではないため、0 が返されます。

## ObjectScript

```
SELECT $LISTFIND($LISTBUILD("ABC","BCD","BBB"),'B')
```

## メモ

### 無効な引数値

list 引数の式が有効なリストとして評価されない場合、\$LISTFIND 関数を使用すると SQLCODE -400 の致命的なエラーが発生します。

## SQL

```
SELECT $LISTFIND("Blue",'Blue')
```

startafter 引数の値が -1 の場合、\$LISTFIND は常にゼロ (0) を返します。

## SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue',-1)
```

startafter 引数の値が -1 より小さい場合、\$LISTFIND 関数を呼び出すと、SQLCODE -400 の致命的なエラーが発生します。

## ObjectScript

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue',-3)
```

## 関連項目

- SQL 関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$PIECE](#)
- ObjectScript 関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTNEXT](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$LISTVALID](#)

## \$LISTFROMSTRING (SQL)

文字列からリストを作成するリスト関数。

### 構文

```
$LISTFROMSTRING(string[,delimiter])
```

### 説明

\$LISTFROMSTRING は、各要素が区切られた引用符付き文字列を受け取り、リストを返します。リストは、区切り文字列を使用しないエンコード形式でデータを表します。したがって、リストには可能な文字をすべて含めることができますが、ビット文字列データに最適です。リストは、ObjectScript および InterSystems SQL \$LIST 関数を使用して操作されます。

### 引数

#### string

文字列リテラル (一重引用符で囲まれた)、数値、あるいは文字列に評価される変数または式。この文字列には、delimiter によって区切られた 1 つ以上の部分文字列 (要素) を含めることができます。delimiter 文字は出力リストに表示されないため、文字列のデータ要素には delimiter 文字 (または文字列) を含めることはできません。

#### delimiter

入力文字列内の部分文字列を区切るのに使用される文字 (または文字列) (オプション)。(一重引用符で囲まれた) 数値または文字列リテラル、変数名、あるいは文字列として評価される式を指定できます。

通常、区切り文字には、文字列データ内で決して使用されることがなく、部分文字列を区切る文字としてのみ使用する特定の文字を設定します。区切り文字には、複数文字から成る文字列を指定することもできますが、それを構成する個々の文字は文字列データ内で使用できます。delimiter を指定しない場合、既定の区切り文字はコンマ (,) です。

### 例

以下の例は、空白スペースで区切られた名前の文字列を受け取り、リストを作成します。

#### SQL

```
SELECT $LISTFROMSTRING("Deborah Noah Martha Bowie",' ')
```

以下の例では、既定の区切り文字 (コンマ) を使用してリストを作成します。

#### SQL

```
SELECT $LISTFROMSTRING("Deborah,Noah,Martha,Bowie")
```

### 関連項目

- SQL 関数: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$PIECE](#)
- ObjectScript 関数: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

# \$LISTGET (SQL)

リスト内の要素または指定された既定値を返すリスト関数です。

## 構文

```
$LISTGET(list[,position[,default]])
```

## 説明

\$LISTGET は、指定したリスト内の要求した要素を標準文字の文字列として返します。position 引数値が存在しないメンバーを参照している場合、または未定義値を持つ要素を指している場合は、指定された既定値を返します。

\$LISTGET 関数は、\$LIST 関数の引数が 1 つおよび 2 つの形式とまったく同じですが、\$LIST が NULL 文字列を返す条件下では、\$LISTGET は既定値を返します。

この関数は、データ型 VARCHAR を返します。

\$LISTGET を使用すると、シリアル・コンテナ・フィールドからフィールド値を取得できます。以下の例では、Home がシリアル・コンテナ・フィールドであり、その 3 つ目の要素が Home\_State です。

## SQL

```
SELECT Name,$LISTGET(Home,3) AS HomeState
FROM Sample.Person
```

## 引数

### list

1 つ以上の要素を含む、エンコードされた文字の文字列。リストは、SQL [\\$LISTBUILD](#) 関数または ObjectScript [\\$LISTBUILD](#) 関数を使用して作成できます。区切り文字列からリストへの変換は、SQL [\\$LISTFROMSTRING](#) 関数または ObjectScript [\\$LISTFROMSTRING](#) 関数を使用して行うことができます。既存のリストからのリストの抽出は、SQL [\\$LIST](#) 関数または ObjectScript [\\$LIST](#) 関数を使用して行うことができます。

### position

position 引数は、必ず整数として評価される必要があります。省略する場合、関数はリストの先頭要素を既定で調べます。position 引数の値が -1 の場合は、リストの最終要素を指定するのと同じ意味です。

### default

これは文字列です。default 引数を省略すると、長さゼロの文字列を既定値とします。

## 例

以下の例の \$LISTGET 関数は、どちらもリストの先頭要素である “Red” を返します。

## SQL

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green")),
    $LISTGET($LISTBUILD("Red","Blue","Green"),1)
```

以下の例の \$LISTGET 関数は、どちらも、リストの最終要素である 3 つ目の要素 “Green” を返します。

## ObjectScript

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green"),3),
    $LISTGET($LISTBUILD("Red","Blue","Green"),-1)
```

以下の例の \$LISTGET 関数は、どちらも、リストの未定義の 2 つ目の要素が見つかった場合に、値を返します。最初の例は、ユーザが既定値として定義した疑問符 (?) を返します。2 つ目の例は、既定値が指定されていないため NULL 文字列を返します。

## SQL

```
SELECT
    $LISTGET($LISTBUILD("Red",,"Green"),2,'?'),
    $LISTGET($LISTBUILD("Red",,"Green"),2)
```

以下の例の \$LISTGET 関数は、どちらも 3 つの要素リストの最終要素よりも大きい位置を指定します。最初の例は、既定値が指定されていないため NULL 文字列を返します。2 つ目の例は、ユーザ指定の既定値 “ERR” を返します。

## SQL

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green"),4),
    $LISTGET($LISTBUILD("Red","Blue","Green"),4,'ERR')
```

以下の例の \$LISTGET 関数は、どちらも NULL 文字列を返します。

## SQL

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green"),0),
    $LISTGET(NULL)
```

## メモ

## 無効な引数値

list 引数の式が有効なリストに評価されない場合は、\$LISTGET が返す変数が未定義のままになるため、SQLCODE -400 の致命的なエラーが発生します。これは、以下の例に示すように default 値が指定されている場合でも発生します。

## SQL

```
SELECT $LISTGET('fred',1,'failsafe')
```

position 引数の値が -1 よりも小さい場合は、\$LISTGET が返す変数が未定義のままになるため、SQLCODE -400 の致命的なエラーが発生します。これは、以下の例に示すように default 値が指定されている場合でも発生します。

## SQL

```
SELECT $LISTGET($LISTBUILD("Red","Blue","Green"),-3,'failsafe')
```

position が数値以外の値の場合は、このエラーは発生しません。

## SQL

```
SELECT $LISTGET($LISTBUILD("Red","Blue","Green"),'g','failsafe')
```

## 関連項目

- SQL 関数 : [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTLENGTH](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$PIECE](#)

- ・ ObjectScript 関数: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

## \$LISTLENGTH (SQL)

指定されたリストの要素数を返すリスト関数です。

### 構文

```
$LISTLENGTH(list)
```

### 概要

\$LISTLENGTH は、*list* にある要素の数を返します。

この関数は、データ型 SMALLINT を返します。

### 引数

*list*

有効なリストとして評価される式。*list* は、1 つ以上の要素を含むエンコードされた文字列です。*list* を作成するには、SQL または ObjectScript の \$LISTBUILD 関数または \$LISTFROMSTRING 関数を使用します。既存のリストから *list* を抽出するには、SQL または ObjectScript の \$LIST 関数を使用します。

### 例

以下の例は、*list* に 3 つの要素が存在するため、3 を返します。

#### SQL

```
SELECT $LISTLENGTH($LISTBUILD("Red","Blue","Green"))
```

以下の SQL の例も、リストに 3 つの要素が存在するため、3 を返します。

#### SQL

```
SELECT $LISTLENGTH($LISTBUILD('Red','Blue','Green'))
```

次の例も 3 を返します。2 つ目の要素にはデータがありませんが、リストに 3 つの要素があるためです。

#### SQL

```
SELECT $LISTLENGTH($LISTBUILD("Red",,"Green"))
```

以下の SQL の例では、各 \$LISTLENGTH は 3 を返します。2 つ目の要素にはデータがありませんが、リストに 3 つの要素があるためです。

#### SQL

```
SELECT $LISTLENGTH($LISTBUILD('Red','','Green')),  
       $LISTLENGTH($LISTBUILD('Red',NULL,'Green')),  
       $LISTLENGTH($LISTBUILD('Red','','Green'))
```

### メモ

#### 無効なリスト

*list* が無効なリストの場合は、SQLCODE -400 の致命的なエラーが発生します。



## SQL

```
SELECT $LISTLENGTH("fred")
```

ObjectScript \$LISTBUILD 関数を使用して、NULL 文字列のみが含まれるリストを作成した場合、これは 1 つの要素からなる有効な list です。

## ObjectScript

```
SELECT $LISTLENGTH($LISTBUILD(""))
```

## NULL リスト

SQL の \$LISTLENGTH 関数と ObjectScript の \$LISTLENGTH 関数とでは、Null リスト (要素を含まないリスト) の扱い方が異なります。

以下の 3 つの例は、\$LISTLENGTH SQL 関数での NULL リストの処理結果を示します。最初の 2 つの例は、list が NULL 文字列で、NULL 文字列を返します。

## SQL

```
SELECT $LISTLENGTH("")
```

## SQL

```
SELECT $LISTLENGTH(NULL)
```

3 つ目の例は、list が値 \$CHAR(0) で、これは無効なリストのため、SQLCODE -400 の致命的なエラーが発生します。

## SQL

```
SELECT $LISTLENGTH('')
```

これは ObjectScript の \$LISTLENGTH 関数での Null リストの扱い方とは異なるということに注意してください。ObjectScript では、Null リスト (要素を含まないリスト) を表すために Null 文字列 ("") が使用されます。それにはリスト要素が含まれないため、以下の例で示すように、\$LISTLENGTH のカウントは 0 となります。

## ObjectScript

```
WRITE $LISTLENGTH("")
```

## \$LISTLENGTH と入れ子のリスト

以下の例は、\$LISTLENGTH が入れ子になっているリストの個々の要素を認識しないため、3 を返します。

## ObjectScript

```
SELECT $LISTLENGTH($LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan")))
```

## 関連項目

- SQL リスト関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)
- その他の SQL 関数: [\\$PIECE](#)
- ObjectScript リスト関数: [\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTNEXT](#)、[\\$LISTSAME](#)、[\\$LISTTOSTRING](#)、[\\$LISTVALID](#)

## \$LISTSAME (SQL)

2 つのリストを比較し、ブーリアン値を返すリスト関数です。

### 構文

```
$LISTSAME(list1,list2)
```

### 説明

\$LISTSAME は、2 つのリストの内容を比較し、リストが同一の場合、1 を返します。リストが同一でない場合、\$LISTSAME は 0 を返します。\$LISTSAME は、2 つのリストを要素ごとに比較します。2 つのリストが同一であるには、同じ数の要素が含まれ、list1 内の各要素が list2 内の対応する要素に一致している必要があります。

\$LISTSAME は、その文字列表現を使用するリスト要素を比較します。\$LISTSAME 比較では大文字と小文字が区別されます。\$LISTSAME は 2 つのリストを要素ごとに左から右の順に比較します。したがって、\$LISTSAME は対応しない最初のリスト要素のペアを検出すると 0 を返し、それ以降の項目が有効なリスト要素かどうかはチェックしません。

この関数は、データ型 SMALLINT を返します。

### 引数

#### list (list1 および list2)

list は、複数の要素を含むエンコードされた文字列です。リストは、SQL [\\$LISTBUILD](#) 関数または ObjectScript [\\$LISTBUILD](#) 関数を使用して作成できます。区切り文字列からリストへの変換は、SQL [\\$LISTFROMSTRING](#) 関数または ObjectScript [\\$LISTFROMSTRING](#) 関数を使用して行うことができます。既存のリストからのリストの抽出は、SQL [\\$LIST](#) 関数または ObjectScript [\\$LIST](#) 関数を使用して行うことができます。

以下に示すのは、有効なリストの例です。

- ・ `$LISTBUILD('a','b','c')`: 3 つの要素のリスト。
- ・ `$LISTBUILD('a','','c')`: 3 つの要素のリスト。2 番目の要素には NULL 文字列値があります。
- ・ `$LISTBUILD('a',,'c')` または `$LISTBUILD('a',NULL,'c')`: 3 つの要素リスト。2 番目の要素に値はありません。
- ・ `$LISTBUILD(NULL,NULL)` または `$LISTBUILD(,NULL)`: 2 つの要素リスト。それぞれの要素に値はありません。
- ・ `$LISTBUILD(NULL)` または `$LISTBUILD()`: 1 つの要素リスト。要素に値はありません。

list 引数が NULL の場合、\$LISTSAME は NULL を返します。list 引数が有効でないリストの場合 (および NULL でない場合)、InterSystems SQL は SQLCODE -400 の致命的なエラーを生成します。

### 例

以下の例では、\$LISTSAME を使用して 2 つのリスト引数を比較しています。

#### SQL

```
SELECT $LISTSAME($LISTBUILD("Red",,"Yellow","Green","", "Violet"),  
$LISTBUILD("Red",,"Yellow","Green","", "Violet"))
```

以下の SQL の例では、NULL であるか、存在しないか、または空の文字列要素があるリストを比較しています。

## SQL

```
SELECT $LISTSAME($LISTBUILD('Red',NULL,'Blue'),$LISTBUILD('Red','','Blue')) AS NullAbsent,
       $LISTSAME($LISTBUILD('Red',NULL,'Blue'),$LISTBUILD('Red','','Blue')) AS NullEmpty,
       $LISTSAME($LISTBUILD('Red','','Blue'),$LISTBUILD('Red','','Blue')) AS AbsentEmpty
```

\$LISTSAME 比較は、ObjectScript 等号によって使用されるのと同じ等価テストではありません。統合は、2 つのリストをエンコードされた文字列として (文字ごとに) 比較します。\$LISTSAME は、2 つのリストを要素ごとに比較します。以下の例で示すように、1 つの数字と 1 つの数値文字列を比較するとき、この相違が容易にわかります。

## ObjectScript

```
SET a = $LISTBUILD("365")
SET b = $LISTBUILD(365)
IF a=b
{ WRITE "Equal sign: lists a and b are the same",! }
ELSE { WRITE "Equal sign: lists a and b are not the same",! }
&sql(SELECT $LISTSAME(:a,:b)
      INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSEIF c=1 { WRITE "$LISTSAME: lists a and b are the same",! }
ELSE { WRITE "$LISTSAME: lists a and b are not the same",! }
```

以下の SQL の例では、キャノニック形式および非キャノニック形式の数字および数値文字列が含まれるリストを比較しています。数値と文字列のリスト要素を比較する際、文字列リスト要素では、数値をキャノニック形式で表す必要があります。これは、InterSystems IRIS では常に、比較の実行前に、数字がキャノニック形式に変換されるためです。以下の例で、\$LISTSAME は文字列と数字を比較します。最初の 3 つの \$LISTSAME 関数は 1 (同一) を返します。4 つ目の \$LISTSAME 関数は 0 (同一でない) を返します。これは、文字列表現がキャノニック形式でないためです。

## SQL

```
SELECT $LISTSAME($LISTBUILD('365'),$LISTBUILD(365)),
       $LISTSAME($LISTBUILD('365'),$LISTBUILD(365.0)),
       $LISTSAME($LISTBUILD('365.5'),$LISTBUILD(365.5)),
       $LISTSAME($LISTBUILD('365.0'),$LISTBUILD(365.0))
```

## 関連項目

- SQL 関数: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTTOSTRING](#) [\\$PIECE](#)
- ObjectScript 関数: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

## \$LISTTOSTRING (SQL)

リストから文字列を作成するリスト関数。

### 構文

```
$LISTTOSTRING(list[,delimiter])
```

### 説明

\$LISTTOSTRING は InterSystems IRIS リストを受け取り、それを文字列に変換します。結果の文字列では、リスト内の要素は delimiter によって区切られます。

リストは、区切り文字列を使用しないエンコード形式でデータを表します。したがって、リストには可能な文字をすべて含めることができますが、ビット文字列データに最適です。\$LISTTOSTRING は、このリストを区切られた要素を持つ 1 つの文字列に変換します。また、指定された文字 (または文字列) を区切り文字として設定します。これらの区切られた要素は、\$PIECE 関数を使用して処理できます。

**注釈** ここで指定する delimiter は、ソース・データに含まれる文字であってははいけません。InterSystems IRIS は、区切り文字の役割を果たす文字と、データ文字としての同じ文字を区別しません。

\$LISTTOSTRING を使用すると、区切り文字列として、シリアル・コンテナ・フィールドからフィールド値を取得できます。以下の例では、Home はシリアル・コンテナ・フィールドです。それには、リスト要素 Home\_Street、Home\_City、Home\_State、および Home\_Zip が含まれています。

### SQL

```
SELECT Name,$LISTTOSTRING(Home,'^') AS HomeAddress
FROM Sample.Person
```

### 引数

#### list

1 つ以上の要素を含む、エンコードされた文字の文字列。リストは、SQL [\\$LISTBUILD](#) 関数または ObjectScript [\\$LISTBUILD](#) 関数を使用して作成できます。区切り文字列からリストへの変換は、SQL [\\$LISTFROMSTRING](#) 関数または ObjectScript [\\$LISTFROMSTRING](#) 関数を使用して行うことができます。既存のリストからのリストの抽出は、SQL [\\$LIST](#) 関数または ObjectScript [\\$LIST](#) 関数を使用して行うことができます。

list 引数の式が有効なリストとして評価されない場合は、SQLCODE -400 エラーが発生します。

#### delimiter

出力文字列内の部分文字列を区切るのに使用される文字 (または文字列) (オプション)。(一重引用符で囲まれた) 数値または文字列リテラル、ホスト変数、あるいは文字列として評価される式を指定できます。

通常、区切り文字には、文字列データ内で決して使用されることがなく、部分文字列を区切る文字としてのみ使用する特定の文字を設定します。区切り文字には、複数文字から成る文字列を指定することもできますが、それを構成する個々の文字は文字列データ内で使用できます。

delimiter を指定しない場合、既定の区切り文字はコンマ (,) です。NULL 文字列 (') は区切り文字として指定可能ですが、この場合は、部分文字列が区切り文字なしで連結されます。一重引用符を区切り文字として指定するには、引用符を二重に指定します ('''' - 4 つの一重引用符を使用します)。

### 例

以下の例は、リスト・フィールドの値を、要素がコロン (:) 文字で区切られた文字列に変換します。

## SQL

```
SELECT
Name,
FavoriteColors AS ColorList,
$LISTTOSTRING(FavoriteColors,':') AS ColorStrings
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
```

## 関連項目

- ・ SQL 関数 : [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTSAME](#) [\\$PIECE](#)
- ・ ObjectScript 関数: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

## LOG (SQL)

与えられた数値式の自然対数を返す、スカラ数値関数です。

### 構文

```
{fn LOG(expression)}
```

### 概要

LOG は、*expression* の自然対数 (基数 e) を返します。LOG は、有効桁数が 21 で小数桁数が 18 の値を返します。

LOG は、{} 括弧構文による ODBC スカラ関数としてのみ使用できます。

### 引数

#### *expression*

数値式。

LOG は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。LOG は、*expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

### 例

以下の例は、整数の自然対数を返します。

#### SQL

```
SELECT {fn LOG(5)} AS Logarithm
```

これは、1.60943791... を返します。

以下の埋め込み SQL の例は、整数 1 から 10 までに対する LOG 関数と EXP 関数の関係を示します。

#### ObjectScript

```
SET a=1
WHILE a<11 {
&sql(SELECT {fn LOG(:a)} INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
ELSE {
    WRITE !,"Logarithm of ",a," = ",b }
&sql(SELECT ROUND({fn EXP(:b)},12) INTO :c)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
ELSE {
    WRITE !,"Exponential of log ",b," = ",c
    SET a=a+1 }
}
```

ここでは、システムの計算制限によって生じる非常に小さな差異を修正する ROUND 関数が必要なことに注意してください。上記の例では、この目的で ROUND がとりあえず小数点以下 12 桁に設定されています。

### 関連項目

- SQL 関数: [EXP](#)、[LOG10](#)、[ROUND](#)
- ObjectScript 関数: [\\$ZLN](#)

# LOG10 (SQL)

与えられた数値式の対数 (底 - 10) を返す、スカラー数値関数です。

## 構文

```
{fn LOG10(expression)}
```

## 説明

LOG10 は、expression の常用対数値を返します。LOG10 は、有効桁数が 21 で小数桁数が 18 の値を返します。

LOG10 は {} 括弧構文による ODBC スカラー関数としてのみ使用できます。

## 引数

### expression

数値式。

LOG10 は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。LOG10 は、expression がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、整数の常用対数を返します。

### SQL

```
SELECT {fn LOG10(5)} AS Log10
```

これは、.69897000433... を返します。

以下の埋め込み SQL の例は、1 から 10 の整数の常用対数を返します。

### ObjectScript

```
SET a=1
WHILE a<11 {
&sql(SELECT {fn LOG10(:a)} INTO :b)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE
QUIT }
ELSE {
WRITE !,"Log-10 of ",a," = ",b
SET a=a+1 }
}
```

## 関連項目

- SQL 関数 : [EXP](#)、[LOG](#)、[ROUND](#)
- ObjectScript 関数: [\\$ZLOG](#)

## LOWER (SQL)

文字列式内のすべての大文字を小文字に変換するケース変換関数です。

### 構文

```
LOWER(string-expression)
```

### 概要

LOWER 関数は、表示目的で大文字を小文字に変換します。これは、UPPER 関数とは逆の関数です。LOWER は、アルファベット以外の文字には影響しません。句読点、数、および先頭と末尾の空白を変更しません。

LOWER は、数値を文字列として解釈する変換を強制的に実行しません。InterSystems SQL は、先頭と末尾のゼロを削除して、数値をキャノニック形式に変換します。文字列として指定された数値はキャノニック形式に変換されず、先頭と末尾のゼロを保持します。

LCASE 関数も、大文字から小文字への変換に使用できます。

LOWER は、[照合](#)に影響しません。%SQLUPPER 関数は、大文字と小文字を区別しない照合に対してデータ値を変換するために SQL で優先的に使用される方法です。照合のケース変換の詳細は、“[%SQLUPPER](#)”を参照してください。

### 引数

#### string-expression

文字列式。その中の文字が小文字に変換されます。式は列の名前や文字列リテラル、または他のスカラー関数の結果を指定できます。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

### 例

以下の例は、各人の名前を小文字で返します。

#### SQL

```
SELECT Name, LOWER(Name) AS LowName
FROM Sample.Person
```

また、LOWER は、ギリシャ文字を大文字から小文字に変換する以下の例で示すように、Unicode (非 ASCII) アルファベット文字でも動作します。

#### SQL

```
SELECT LOWER($CHAR(920,913,923,913,931,931,913))
FROM Sample.Person
```

### 関連項目

- SQL 関数: [LCASE](#)、[UCASE](#)
- ObjectScript 関数: [\\$ZCONVERT](#)



## LPAD (SQL)

指定された長さになるまで左側にパディングされた文字列を返す文字列関数です。

### 構文

```
LPAD(string-expression, length[, padstring])
```

### 説明

LPAD は、文字列式の先頭にパディング文字を追加します。length 文字数にパディングされた文字列のコピーを返します。文字列式が length 文字より長い場合、返される文字列は length 文字まで切り捨てられます。

string-expression が NULL の場合、LPAD は NULL を返します。string-expression が空文字列 (') の場合、LPAD はすべてパディング文字で構成される文字列を返します。返される文字列は VARCHAR 型です。

LPAD をリンク・テーブルに対するクエリで使用できます。

LPAD は、先頭または末尾の空白を削除しません。先頭または末尾の空白も含めて文字列をパディングします。文字列をパディングする前に先頭または末尾の空白を削除するには、LTRIM、RTRIM、または TRIM を使用します。

### LPAD および \$JUSTIFY

引数が 2 つの形式の LPAD および引数が 2 つの形式の \$JUSTIFY はどちらも、先頭にスペースを使用してパディングすることで文字列を右寄せします。引数が 2 つのこれらの形式の違いは、入力 string-expression の長さより短い出力 length の扱い方です。LPAD は、指定された出力長に合わせて入力文字列を切り捨てます。\$JUSTIFY は、入力文字列に合わせて出力長を拡大します。詳細は、以下の例を参照してください。

#### SQL

```
SELECT '>' || LPAD(12345,10) || '<' AS lpadplus,
       '>' || $JUSTIFY(12345,10) || '<' AS justifyplus,
       '>' || LPAD(12345,3) || '<' AS lpadminus,
       '>' || $JUSTIFY(12345,3) || '<' AS justifyminus
```

### 引数

#### string-expression

列の名前、文字列リテラル、ホスト変数、他のスカラ関数の結果などを表すことができる文字列式。VARCHAR データ型に変換できる任意のデータ型を指定できます。string-expression をストリームにはできません。

#### length

返される文字列の文字数を指定する整数。

#### padstring

入力の string-expression のパディングに使用される、文字列または 1 文字で構成される文字列 (オプション)。padstring 文字 (1 つまたは複数) は、length 文字の出力文字列を作成するために必要な数だけ、string-expression の左側に付加されます。padstring には、文字列リテラル、列、ホスト変数、または別のスカラ関数の結果を指定できます。省略すると、既定で空白スペース文字になります。

### 例

以下の例では、列値を (必要に応じて) ^ 文字で左パディングして、長さが 16 文字の文字列を返します。Name 文字列に応じて、左パディングされるか右側が切り捨てられて、長さ 16 文字の文字列が返されます。

## SQL

```
SELECT TOP 15 Name,LPAD(Name,16,'^') AS Name16
FROM Sample.Person
```

以下の例では、列値を（必要に応じて）`^` 文字列で左パディングして、長さが 20 文字の文字列を返します。名前文字列のパディングは必要なだけ繰り返されます。返される文字列には、パディングされた文字列が部分的に含まれる場合があります。

## SQL

```
SELECT TOP 15 Name,LPAD(Name,20,'^=^') AS Name20
FROM Sample.Person
```

## 関連項目

- ・ [\\$JUSTIFY](#) 関数
- ・ [RPAD](#) 関数
- ・ [LTRIM](#) 関数
- ・ [RTRIM](#) 関数
- ・ [TRIM](#) 関数

# LTRIM (SQL)

先頭の空白を削除した文字列を返す文字列関数です。

## 構文

```
LTRIM(string-expression)
{fn LTRIM(string-expression)}
```

## 概要

LTRIM は文字列式から先頭の空白を削除し、その文字列を VARCHAR タイプとして返します。string-expression が NULL の場合、LTRIM は NULL を返します。string-expression が完全に空白スペースで構成されている文字列の場合、LTRIM は空文字列 (') を返します。

LTRIM は末尾の空白をそのまま残します。末尾の空白を削除するには、RTRIM を使用します。先頭や末尾のあらゆるタイプの文字を削除するには、TRIM を使用します。文字列の先頭に空白またはその他の文字でパディングするには、LPAD を使用します。空白の文字列を作成するには、SPACE を使用します。

LTRIM は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として使用できる点に注意してください。

## 引数

### string-expression

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

## 例

以下の例は、文字列から先頭の 5 つの空白を削除します。末尾の 5 つの空白はそのまま残します。

### SQL

```
SELECT {fn LTRIM("      Test string with 5 leading and 5 trailing spaces.      ")}
```

これは、以下を返します。

```
Before LTRIM
start:      Test string with 5 leading and 5 trailing spaces.      :end
After LTRIM
start:Test string with 5 leading and 5 trailing spaces.      :end
```

## 関連項目

[RTRIM](#) [TRIM](#) [LPAD](#) [SPACE](#)

## %MINUS (SQL)

数値をキャノニック照合形式に変換してから符号を反転する照合関数です。

### 構文

```
%MINUS(expression)
```

```
%MINUS expression
```

### 説明

%MINUS は、数値や数値文字列をキャノニック形式に変換してから符号を反転し、これらの *expression* 値を数値照合順に返します。

%MINUS が記号を反転することを除いて、%MINUS と [%PLUS](#) は機能的に同一です。正の数値の場合、数値の先頭にマイナス符号が付加されます。負の数値の場合、数値にあるマイナス符号は削除されます。ゼロの場合、符号はありません。

数値には、先頭と末尾のゼロ、先頭にある複数のプラス符号とマイナス符号、1 つの小数点 (.), および指数記号 (E) を指定できます。キャノニック形式で、すべての算術演算が実行され、指数が展開され、単体のマイナス符号が先頭に付くか符号なしになり、先頭と末尾のゼロが削除されます。

数値リテラルは、文字列を囲む区切り文字を付けて指定することも、付けなくて指定することもできます。数値以外の文字が文字列にあると、%MINUS では、最初の数値以外の文字で数値を切り捨て、数値部分をキャノニック形式で返します。数値でない文字列 (数値以外の文字で始まる文字列) を 0 として返します。%MINUS はさらに、NULL を 0 として返します。

%MINUS は、InterSystems SQL の拡張機能であり、SQL 検索クエリ用として使用するものです。

%SYSTEM.Util クラスの Collation() メソッドを使用すると、ObjectScript で同じ照合変換を実行できます。

#### ObjectScript

```
WRITE $SYSTEM.Util.Collation("++007.500",4)
```

%MINUS を %MVR と比べると、%MVR では文字列内の数値部分文字列に基づいて文字列をソートしている点異なります。

### 引数

#### expression

列名、数値や文字列リテラル、算術式、または別の関数の結果となる式。基本となるデータ型は、任意の文字タイプとすることができます。

### 例

以下の例では、%MINUS を使用して、数値の降順でホーム・ストリート番号を示すレコードを返します。

#### SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %MINUS(Home_Street)
```

上記の例では、ストリート・アドレスの整数部分が数値順に配列されています。これを以下の ORDER BY DESC の例と比較してください。こちらでは、レコードは照合順のストリート・アドレスで配列されています。

## SQL

```
SELECT Name,Home_Street  
FROM Sample.Person  
ORDER BY Home_Street DESC
```

## 関連項目

- ・ [%EXACT](#) 照合関数
- ・ [%PLUS](#) 照合関数
- ・ [照合](#)

## MINUTE (SQL)

日付/時刻式に対して分を返す時刻関数です。

### 構文

```
{fn MINUTE(time-expression)}
```

### 概要

MINUTE は、指定された時刻または日付/時刻値に対して分の部分を示す整数を返します。分は、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプに基づいて計算されます。

*time-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

既定の時刻形式を変更するには、[SET OPTION](#) コマンドを使用します。

時刻整数 (経過秒数) は指定できますが、時刻文字列 (hh:mm:ss) は指定できません。日付/時刻文字列 (yyyy-mm-dd hh:mm:ss) を指定する必要があります。

日付/時刻文字列の時刻部分は有効な時刻である必要があります。それ以外の場合には、SQLCODE -400 エラー <ILLEGAL VALUE> が生成されます。分 (mm) 部分は、0 から 59 までの範囲の整数である必要があります。入力では、先頭のゼロはオプションです。出力では、先頭のゼロは抑制されます。日付/時刻文字列の秒 (:ss) 部分を省略しても、分部分を返すことができます。

日付/時刻文字列の日付部分は検証されません。

分部分が '0' または '00' の場合、MINUTE はゼロ分を返します。また、時刻式が指定されていない場合、または時刻式の分部分が完全に省略されている場合 ('hh'、'hh:'、'hh:.'、または 'hh:ss') にも、ゼロ分が返されます。

[DATEPART](#) または [DATENAME](#) を使用して、同じ時刻情報を取得できます。

この関数は、ObjectScript から MINUTE() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.MINUTE(time-expression)
```

### 引数

#### *time-expression*

列の名前や、他のスカラー関数の結果、または文字列や日付や数値リテラルである式。日付/時刻文字列または時刻整数に解決される必要があります。基本となるデータ型は、[%Time](#)、[%TimeStamp](#)、または [%PosixTime](#) とすることができます。

### 例

以下の例は、日付/時刻文字列の時刻式が 45 分を表しているので、両方とも 45 を返します。

#### SQL

```
SELECT {fn MINUTE('2018-02-16 18:45:38')} AS ODBCMinutes
```

#### SQL

```
SELECT {fn MINUTE(67538)} AS HorologMinutes
```

以下の例も 45 を返します。ここに示すように、時刻値の秒部分は省略できます。

## SQL

```
SELECT {fn MINUTE('2018-02-16 18:45')} AS Minutes_Given
```

以下の例は、日付/時刻文字列から時刻式が省略されているので、ゼロ分を返します。

## SQL

```
SELECT {fn MINUTE('2018-02-16')} AS Minutes_Given
```

以下の例はすべて、現在の時刻の分部分を返します。

## SQL

```
SELECT {fn MINUTE(CURRENT_TIME)} AS Min_CurrentT,
       {fn MINUTE({fn CURTIME()})} AS Min_CurT,
       {fn MINUTE({fn NOW()})} AS Min_Now,
       {fn MINUTE($HOROLOG)} AS Min_Horolog,
       {fn MINUTE($ZTIMESTAMP)} AS Min_ZTS
```

以下の例は、先頭のゼロが抑制されることを示します。最初の MINUTE 関数は長さ 2 を返し、その他は長さ 1 を返します。省略された時刻はゼロ分と見なされ、1 の長さを持ちます。

## SQL

```
SELECT LENGTH({fn MINUTE('2018-02-22 11:45:00')}),
       LENGTH({fn MINUTE('2018-02-22 03:05:00')}),
       LENGTH({fn MINUTE('2018-02-22 3:5:0')}),
       LENGTH({fn MINUTE('2018-02-22')})
```

以下の埋め込み SQL の例では、MINUTE 関数が、ロケールに指定された TimeSeparator 文字を認識していることを示しています。

## ObjectScript

```
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator",".")
&sql(SELECT {fn MINUTE('2018-02-22 18.45.38')} INTO :a)
QUIT:(SQLCODE '= 0)
WRITE "minutes=",a
```

## 関連項目

- SQL の概念 : [データ型](#)、[日付/時刻文](#)
- SQL 関数 : [HOUR](#)、[SECOND](#)、[CURRENT\\_TIME](#)、[CURTIME](#)、[NOW](#)、[DATEPART](#)、[DATENAME](#)
- ObjectScript 関数: [\\$ZTIME](#)
- ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## MOD (SQL)

ある数を別の数で割った剰余を返す、スカラ数値関数です。

### 構文

```
MOD(dividend,divisor)  
{fn MOD(dividend,divisor)}
```

### 概要

MOD は、被除数を除数で割った余り (剰余) を返します。

MOD は、標準スカラ関数または {} 括弧構文による ODBC スカラ関数として指定できます。

- ・ *dividend* と *divisor* が正である場合、正の剰余またはゼロを返します。
- ・ *dividend* と *divisor* の両方が負である場合、負の剰余またはゼロを返します。
- ・ *dividend* または *divisor* が NULL である場合、NULL を返します。
- ・ *divisor* が 0 である場合、<DIVIDE> エラーの %msg を含む SQLCODE -400 を生成します。
- ・ *divisor* が *dividend* より大きい場合、*dividend* を返します。

MOD に対してレポートされる[精度](#) (いずれかの構文形式) は、算術式 *dividend*/*divisor* の精度レポートと同じです。

### ANSI 演算子の優先順位

1 つの負のオペランドが指定された MOD 関数の動作は、[ANSI の演算子優先順位を適用する] 構成設定によって異なります。

- ・ [ANSI の演算子優先順位を適用する] が適用されない場合、負のオペランドが指定された MOD の動作は [# モジューロ演算子](#)と同じです。どちらも、剰余ではなく、不足分 (次の倍数に達するために必要な数) を返します。例えば、12#7 は剰余の 5 を返します。-12#7 は不足分の 2 を返します。*dividend* が負である場合、不足分は正の値またはゼロです。*divisor* が負である場合、不足分は負の値またはゼロです。
- ・ [ANSI の演算子優先順位を適用する] が適用される場合 (InterSystems IRIS 2019.1 以降の既定)、負のオペランドが指定された MOD の動作は常に、剰余を返します。*dividend* が負である場合、負の剰余またはゼロを返します。*divisor* が負である場合、正の剰余またはゼロを返します。

[# モジューロ演算子](#)の動作は、[ANSI の演算子優先順位を適用する] 構成設定の影響を受けません。

### 引数

#### *dividend*

除算の分子 (被除数) である数字。

#### *divisor*

除算の分母 (除数) である数字。

*dividend* がデータ型 DOUBLE でない限り、MOD は NUMERIC [データ型](#)を返します。*dividend* が DOUBLE の場合、MOD は DOUBLE を返します。



## 例

以下の例は、MOD で返された余りを示しています。

### SQL

```
SELECT MOD(5,3) AS Remainder
```

これは、2 を返します。

### SQL

```
SELECT MOD(5.3,.5) AS Remainder
```

これは、.3 を返します。

## 関連項目

[CEILING](#)、[FLOOR](#)、[ROUND](#)、[TRUNCATE](#)

## MONTH (SQL)

日付式に対してその月を整数として返す日付関数です。

### 構文

```
MONTH(date-expression)  
{fn MONTH(date-expression)}
```

### 概要

MONTH は、月を示す整数を返します。月の整数は、InterSystems IRIS 日付整数、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプに基づいて計算されます。

*date-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

日付文字列の月 (mm) 部分は、1 から 12 までの範囲の整数である必要があります。入力では、先頭のゼロはオプションです。出力では、先頭と末尾のゼロは抑制されます。

*date-expression* の日付部分は検証され、1 から 12 までの範囲の月、および指定した月と年の有効な日の値を含む必要があります。それ以外の場合には、SQLCODE -400 エラー <ILLEGAL VALUE> が生成されます。

*date-expression* の時刻部分は検証されないため省略可能です。

MONTH は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として呼び出せる点に注意してください。

この関数は、ObjectScript から MONTH() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.MONTH(date-expression)
```

日付/時刻文字列の要素は、SQL 関数の YEAR、MONTH、DAY (または DAYOFMONTH)、HOUR、MINUTE、SECOND をそれぞれ使用して取得できます。[DATEPART](#) または [DATENAME](#) 関数を使用して、同じ要素を取得することもできます。日付要素は、[TO\\_DATE](#) を使用して取得できます。DATEPART および DATENAME では、月の値に対して値と範囲のチェックが行われます。

[LAST\\_DAY](#) 関数は、指定した月の最後の日付を返します。

### 引数

#### *date-expression*

列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式。

### 例

以下の例は、2 月は 1 年の中で 2 番目の月なので、両方とも 2 を返します。

#### SQL

```
SELECT MONTH('2018-02-22') AS Month_Given
```

#### SQL

```
SELECT {fn MONTH(64701)} AS Month_Given
```

以下の例では、DOB の年コンポーネントを無視して、レコードを月と日の誕生日順に配列しています。

## SQL

```
SELECT Name,DOB AS Birthdays
FROM Sample.Person
ORDER BY MONTH(DOB),DAY(DOB),Name
```

以下の例はすべて、現在の月を返します。

## SQL

```
SELECT {fn MONTH({fn NOW()})} AS MNow,
       MONTH(CURRENT_DATE) AS MCurrD,
       {fn MONTH(CURRENT_TIMESTAMP)} AS MCurrTS,
       MONTH($HOROLOG) AS MHorolog,
       {fn MONTH($ZTIMESTAMP)} AS MZTS
```

## 関連項目

- ・ SQL 関数: [DATEPART](#)、[DATENAME](#)、[DAYOFMONTH](#)、[LAST\\_DAY](#)、[MONTHNAME](#)、[TO\\_DATE](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## MONTHNAME (SQL)

日付式に対してその月の名前を返す日付関数です。

### 構文

```
{fn MONTHNAME(date-expression)}
```

### 概要

MONTHNAME では、入力として InterSystems IRIS 日付整数、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列、またはタイムスタンプが取得されます。

*date-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

タイムスタンプの時刻部分は評価されないので省略可能です。

MONTHNAME は、該当する月の名前 (January ~ December) を返します。この返り値は最長 15 文字の文字列です。

MONTHNAME は、指定された日付が有効であることをチェックします。年は 0001 ~ 9999、月は 01 ~ 12、日はその月に適切な数字 (例えば、02/29 はうるう年のみ有効) である必要があります。日付が無効の場合、MONTHNAME は SQLCODE -400 <ILLEGAL VALUE> エラーを発行します。

月の名前は、既定のアメリカ英語の省略のない月の名前になります。これらの月の名前の値を変更するには、MONTH\_NAME オプションで [SET OPTION](#) コマンドを使用します。

[DATENAME](#) 関数を使用して、同じ月名情報を取得することもできます。[TO\\_DATE](#) を使用すると、他の日付要素で月の名前または月名の省略形を検索できます。月に対応する整数を返すには、[MONTH](#)、[DATEPART](#)、または [TO\\_DATE](#) を使用します。

この関数は、ObjectScript から MONTHNAME() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.MONTHNAME(date-expression)
```

### 引数

#### *date-expression*

InterSystems IRIS 日付整数、ODBC 日付、またはタイムスタンプとして評価される式。この式は、列の名前、他のスカラ関数の結果、または日付やタイムスタンプのリテラルとすることができます。

### 例

以下の例は、日付式 (February 22, 2018) の月は 2 月なので、両方とも、文字列 "February" を返します。

#### SQL

```
SELECT {fn MONTHNAME('2018-02-22')} AS NameOfMonth
```

#### SQL

```
SELECT {fn MONTHNAME(64701)} AS NameOfMonth
```

以下の例はすべて、現在の月を返します。

## SQL

```
SELECT {fn MONTHNAME({fn NOW()})} AS MnameNow,  
       {fn MONTHNAME(CURRENT_DATE)} AS MNameCurrDate,  
       {fn MONTHNAME(CURRENT_TIMESTAMP)} AS MNameCurrTS,  
       {fn MONTHNAME($HOROLOG)} AS MNameHorolog,  
       {fn MONTHNAME($ZTIMESTAMP)} AS MNameZTS
```

以下の例は、MONTHNAME が無効な日付 (2017 年はうるう年ではない) をどのように処理するかを示しています。

## SQL

```
SELECT {fn MONTHNAME("2017-02-29")}
```

<ILLEGAL VALUE> を示す %msg と共に SQLCODE -400 エラー・コードが発行されます。

## 関連項目

- ・ SQL 関数: [DATEPART](#)、[DATENAME](#)、[DAYOFMONTH](#)、[MONTH](#)、[TO\\_DATE](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## NOW (SQL)

現在のローカル日付とローカル時刻を返す日付/時刻関数です。

### 構文

```
NOW()  
  
{fn NOW}  
{fn NOW() }
```

### 概要

NOW は、引数を取りません。引数の括弧は ODBC スカラ構文ではオプションですが、SQL 標準関数構文では必須です。

NOW は、この[タイムゾーン](#)の現在のローカル日付とローカル時刻をタイムスタンプとして返します。これは[サマータイム](#)などのローカル時刻調整に合わせて調整されます。

NOW は、%TimeStamp データ型形式 (yyyy-mm-dd hh:mm:ss.fff) または %PosixTime データ型形式 (エンコードされた 64 ビットの符号付き整数) のいずれかで[タイムスタンプ](#)を返すことができます。返されるタイムスタンプ形式は以下のルールによって決まります。

1. 現在のタイムスタンプがデータ型 %PosixTime のフィールドに対して指定されている場合、現在のタイムスタンプ値は POSIXTIME データ型形式で返されます。例として、WHERE PosixField=NOW() や INSERT INTO MyTable (PosixField) VALUES (NOW()) が挙げられます。
2. 現在のタイムスタンプがデータ型 %TimeStamp のフィールドに対して指定されている場合、現在のタイムスタンプ値は TIMESTAMP データ型形式 (yyyy-mm-dd hh:mm:ss) で返されます。ODBC タイプは TIMESTAMP で、LENGTH は 16、PRECISION は 19 です。時間は 24 時間形式で表示されます。先頭のゼロは、すべてのフィールドで維持されます。例として、WHERE TSField=NOW() や INSERT INTO MyTable (TSField) VALUES (NOW()) が挙げられます。
3. 現在のタイムスタンプがコンテキストなしで指定されている場合、現在のタイムスタンプ値は TIMESTAMP データ型形式で返されます。例として、SELECT NOW() が挙げられます。

既定の日付/時刻形式を変更するには、各種日付/時刻オプションで [SET OPTION](#) コマンドを使用します。

タイムスタンプ、日付、および時刻のデータ型は、[CAST](#) または [CONVERT](#) 関数を使用して変更できます。

### 秒の小数部の精度

既定では、NOW は秒の小数部の精度を返しません。精度引数はサポートされません。ただし、[システム全体の既定の時刻精度](#)を変更することで、システム全体ですべての NOW 関数が秒の小数部の精度でこの構成された桁数を返すことができます。システム全体の既定の時刻精度における最初の構成設定は 0 (秒の小数部なし) で、最大設定値は 9 です。

システム全体の既定の時刻精度をオーバーライドできる precision 引数を GETDATE で用意している点を除いて、[GETDATE](#) は NOW と機能的に同じになります。precision 引数を省略した場合、構成されているシステム全体の既定の時刻精度が GETDATE で使用されます。

[CURRENT\\_TIMESTAMP](#) には、次の 2 つの構文形式があります。引数の括弧がない場合、CURRENT\_TIMESTAMP は NOW と機能的に同じになります。引数の括弧がある場合、CURRENT\_TIMESTAMP(precision) は、CURRENT\_TIMESTAMP() precision 引数が必須である点を除いて、GETDATE と機能的に同じになります。CURRENT\_TIMESTAMP() は常に、指定された precision を返し、構成されたシステム全体の既定の時刻精度を無視します。

秒の小数部は、丸められずに、常に指定された精度に切り捨てられます。

[SYSDATE](#) は、引数なしの [CURRENT\\_TIMESTAMP](#) 関数と機能的に同じになります。

## その他の現在日時関数

[NOW](#)、[GETDATE](#)、[CURRENT\\_TIMESTAMP](#)、および [SYSDATE](#) はすべて、ローカル・タイム・ゾーン設定に基づいて、現在のローカル日付とローカル時刻を返します。

[GETUTCDATE](#) は、現在の UTC (協定世界時) 日付と UTC 時刻をタイムスタンプとして返します。UTC 時刻はローカル・タイム・ゾーンに依存しておらず、ローカル時刻調整 ([サマータイム](#)など) の影響は受けないため、異なるタイム・ゾーン間のユーザが同一データベースにアクセスする場合に一貫したタイムスタンプを適用する際にこの関数は便利です。[GETUTCDATE](#) は、秒の小数部の精度をサポートします。現在の UTC タイムスタンプは、ObjectScript の [\\$ZTIMESTAMP](#) 特殊変数でも提供されます。

現在の日付のみを返すには、[CURDATE](#) または [CURRENT\\_DATE](#) を使用します。現在の時刻のみを返すには、[CURRENT\\_TIME](#) または [CURTIME](#) を使用します。これらの関数では、DATE または TIME データ型が使用されます。TIME データ型および DATE データ型では、値は、[\\$HOROLOG](#) 形式で整数として格納されます。これらの関数は、いずれも精度をサポートしません。

## 例

以下の例は、3 つの構文形式が同等であることを示しています。すべて現在のローカル日付とローカル時刻をタイムスタンプとして返します。

### SQL

```
SELECT NOW(), {fn NOW}, {fn NOW()}
```

以下の例は、ローカル・タイムスタンプ (タイム・ゾーン依存) とユニバーサル・タイムスタンプ (タイム・ゾーン非依存) を比較します。

### SQL

```
SELECT NOW(), GETUTCDATE()
```

以下の例は、Orders テーブルの指定された行の LastUpdate フィールドに、現在のシステム日付と時刻を設定します。

### SQL

```
UPDATE Orders SET LastUpdate = {fn NOW()}
WHERE Orders.OrderNumber=:ord
```

## 関連項目

- SQL の概念 : [データ型](#)、[日付/時刻文](#)
- SQL タイムスタンプ関数 : [CAST](#)、[CONVERT](#)、[CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[GETUTCDATE](#)、[SYSDATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)、[TO\\_TIMESTAMP](#)
- SQL 現在日時関数 : [CURDATE](#)、[CURRENT\\_DATE](#)、[CURRENT\\_TIME](#)、[CURTIME](#)
- ObjectScript : [\\$ZDATETIME](#) 関数、[\\$HOROLOG](#) 特殊変数、[\\$ZTIMESTAMP](#) 特殊変数

## NULLIF (SQL)

---

2 つの式の値が同じ場合に NULL を返す関数。

### 構文

```
NULLIF(expression1,expression2)
```

### 概要

NULLIF 関数は、*expression1* の値が *expression2* の値に等しい場合に NULL を返します。そうでない場合は、*expression1* の値を返します。

NULLIF は、以下と同等です。

### SQL

```
SELECT CASE  
  WHEN value1 = value2 THEN NULL  
  ELSE value1  
END  
FROM MyTable
```

### 引数

#### *expression1*

列の名前、数値リテラル、文字列リテラル、ホスト変数、または他のスカラ関数の結果を指定できる式。

#### *expression2*

列の名前、数値リテラル、文字列リテラル、ホスト変数、または他のスカラ関数の結果を指定できる式。

NULLIF が返す値のデータ型は、*expression1* のデータ型と同じです。

### NULL を処理する関数の比較

以下の表は、さまざまな SQL 比較関数を示します。論理比較テストが True (A は B と同じ) の場合、各関数は特定の値を返し、False (A は B と同じではない) の場合、別の値を返します。これらの関数により、NULL の論理比較を実行できます。実際の等値 (または非等値) 条件比較で NULL を指定することはできません。



SQL 関数	比較テスト	返り値
NULLIF(ex1,ex2)	ex1 = ex2	True の場合、NULL を返す False の場合、ex1 を返す
ISNULL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
IFNULL(ex1,ex2) [引数が 2 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、NULL を返す
IFNULL(ex1,ex2,ex3) [引数が 3 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、ex3 を返す
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
NVL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
COALESCE(ex1,ex2,...)	各引数で ex = NULL	True の場合、次の ex 引数をテスト すべての ex 引数が True (NULL) の 場合、NULL を返す  False の場合、ex を返す

## 例

以下の例は、NULLIF 関数を使用して、Age=20 を持つすべてのレコードの表示フィールドを NULL に設定します。

### SQL

```
SELECT Name, Age, NULLIF(Age, 20) AS Nulled20
FROM Sample.Person
```

## 関連項目

- ・ [CASE コマンド](#)
- ・ [COALESCE 関数](#)
- ・ [IFNULL 関数](#)
- ・ [ISNULL 関数](#)
- ・ [NVL 関数](#)

## NVL (SQL)

NULL テストを行い、適切な式を返す関数です。

### 構文

```
NVL(check-expression,replace-expression)
```

### 概要

NVL は *check-expression* を評価し、2 つの値のうち 1 つを返します。

- ・ *check-expression* が NULL の場合は、*replace-expression* を返します。
- ・ *check-expression* が NULL でない場合は、*check-expression* を返します。

引数 *check-expression* および *replace-expression* は、あらゆるデータ型を持つことができます。それらのデータ型が異なる場合、SQL はそれらと比較する前に *replace-expression* を *check-expression* のデータ型に変換します。戻り値のデータ型は常に *check-expression* と同じです。ただし、*check-expression* が文字データでない場合は、戻り値のデータ型は VARCHAR2 になります。

NVL は Oracle との互換性のためにサポートされており、ISNULL 関数と同じである点に注意してください。

NULL の処理方法の詳細は "[NULL](#)" を参照してください。

### DATE および TIME の表示変換

*check-expression* のデータ型によっては、論理モードから ODBC モードまたは表示モードへの変換が必要になります。例えば、DATE データ型と TIME データ型です。*replace-expression* の値が同じデータ型でない場合は、この値を ODBC モードまたは表示モードに変換することはできません。DATE データ型の場合は SQLCODE エラー -146、TIME データ型の場合は SQLCODE エラー -147 が生成されます。例えば、ISNULL(DOB, 'nodate') は、ODBC モードまたは表示モードでは実行できません。SQLCODE -146 エラーが発行され、%msg が " 'nodate' ODBC/JDBC " または " 'nodate' " に設定されます。ODBC モードまたは表示モードでこの文を実行するには、値を適切なデータ型 ISNULL(DOB, CAST('nodate' as DATE)) としてキャストする必要があります。これにより日付 0 になり、1840-12-31 として表示されます。

### 引数

#### *check-expression*

評価される式。

#### *replace-expression*

*check-expression* が NULL の場合に返される式。

NVL が返す値のデータ型は、*check-expression* のデータ型と同じです。

### NULL を処理する関数の比較

以下の表は、さまざまな SQL 比較関数を示します。論理比較テストが True (A は B と同じ) の場合、各関数は特定の値を返し、False (A は B と同じではない) の場合、別の値を返します。これらの関数により、NULL の論理比較を実行できます。実際の等値 (または非等値) 条件比較で NULL を指定することはできません。

SQL 関数	比較テスト	返り値
NVL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
IFNULL(ex1,ex2) [引数が 2 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、NULL を返す
IFNULL(ex1,ex2,ex3) [引数が 3 つの形式]	ex1 = NULL	True の場合、ex2 を返す False の場合、ex3 を返す
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
ISNULL(ex1,ex2)	ex1 = NULL	True の場合、ex2 を返す False の場合、ex1 を返す
NULLIF(ex1,ex2)	ex1 = ex2	True の場合、NULL を返す False の場合、ex1 を返す
COALESCE(ex1,ex2,...)	各引数で ex = NULL	True の場合、次の ex 引数をテスト すべての ex 引数が True (NULL) の 場合、NULL を返す  False の場合、ex を返す

## 例

以下の例は、check-expression が NULL なので、replace-expression (99) を返します。

### SQL

```
SELECT NVL(NULL,99) AS NullTest
```

以下の例は、check-expression が NULL ではないので、check-expression (33) を返します。

### SQL

```
SELECT NVL(33,99) AS NullTest
```

以下の例は、FavoriteColors が NULL の場合は文字列 'No Preference' を返し、そうでない場合は FavoriteColors の値を返します。

### SQL

```
SELECT Name, NVL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

## 関連項目

- ・ [CASE コマンド](#)
- ・ [COALESCE 関数](#)

- ・ [IFNULL](#) 関数
- ・ [ISNULL](#) 関数
- ・ [NULLIF](#) 関数

# %OBJECT (SQL)

ストリーム・オブジェクトを開き、対応する oref を返すスカラ関数です。

## 構文

```
%OBJECT(stream)
```

## 概要

%OBJECT を使用してストリーム・オブジェクトが開かれ、[ストリーム・フィールド](#)の oref (オブジェクト参照) が返されます。

ストリーム・フィールドでの SELECT は、ストリーム・フィールドの整形形式 OID (オブジェクト ID) 値を返します。ストリーム・フィールドでの SELECT %OBJECT は、ストリーム・フィールドの oref (オブジェクト参照) を返します。

stream がストリーム・フィールドでない場合、%OBJECT は SQLCODE -128 エラーを発行します。

%OBJECT は、以下の関数に対する引数として使用できます。

- CHARACTER\_LENGTH(%OBJECT(streamfield)), CHAR\_LENGTH(%OBJECT(streamfield)), または DATALENGTH(%OBJECT(streamfield))
- SUBSTRING(%OBJECT(streamfield),start,length)

ストリーム・フィールドで SELECT を発行し、oid から oref を生成する \$Stream.Object.%Open() クラス・メソッドを呼び出してこのストリーム oid を開くことでも、同じ操作を実行することができます。

```
SET oref = ##class(%Stream.Object).%Open(oid)
```

oref の詳細は、“[OREF の基本](#)” を参照してください。oid の詳細は、上記と同じドキュメントの“[保存したオブジェクトの識別子：ID および OID](#)” を参照してください。

## 引数

### stream

ストリーム・フィールドの名前である式。

## 例

以下の例は、ストリーム・フィールドに対する SELECT %OBJECT が oref を返す方法を示しています。Notes と Picture はどちらもストリーム・フィールドです。

### SQL

```
SELECT TOP 3 Title,Notes,%OBJECT(Picture) AS Photo FROM Sample.Employee
```

## 関連項目

- [SELECT](#)
- [既定の SQL プロジェクションの概要](#)
- [SQL および ODBC へのストリーム・プロパティのプロジェクション](#)
- [BLOB と CLOB の格納と使用](#)

## %ODBCIN (SQL)

---

論理形式の式を返す形式変換関数です。

### 構文

```
%ODBCIN(expression)
```

```
%ODBCIN expression
```

### 概要

%ODBCIN は、フィールドやデータ型の OdbcToLogical メソッドで値を渡した後、*expression* を論理形式で返します。論理形式は、データのメモリ内形式です (処理が実行される形式)。

%ODBCIN は、InterSystems SQL の拡張です。

表示形式オプションの詳細は、“[データ表示オプション](#)” を参照してください。

### 引数

#### *expression*

変換される式。

### 例

以下の例は、既定の表示形式 %ODBCIN および同じフィールドの %ODBCOUT 形式を示しています。

#### SQL

```
SELECT FavoriteColors,%ODBCIN(FavoriteColors) AS InVal,  
       %ODBCOUT(FavoriteColors) AS OutVal  
FROM Sample.Person
```

以下の例は、WHERE 節で %ODBCIN を使用しています。

#### SQL

```
SELECT Name,DOB,%ODBCOUT(DOB) AS Birthdate  
FROM Sample.Person  
WHERE DOB BETWEEN %ODBCIN('2000-01-01') AND %ODBCIN('2018-01-01')
```

### 関連項目

[%EXTERNAL](#)、[%INTERNAL](#)、[%ODBCOUT](#)

# %ODBCOUT (SQL)

ODBC 形式の式を返す形式変換関数です。

## 構文

```
%ODBCOUT(expression)
```

```
%ODBCOUT expression
```

## 概要

%ODBCOUT は、フィールドやデータ型の LogicalToOdbc メソッドで値を渡した後、*expression* を ODBC 形式で返します。ODBC 形式は、ODBC を経由してデータを表示できる形式です。この形式は、データが ODBC/SQL に公開されるときに使用されます。使用可能な形式は ODBC で定義したものに対応します。

通常、%ODBCOUT は、SELECT リストの select-item で使用されます。WHERE 節内で使用できますが、%ODBCOUT を使用すると指定されたフィールドでインデックスを使用できなくなるので、使用しないことをお勧めします。

%ODBCOUT を適用すると、列見出し名が “Expression\_1” などの値に変更されます。したがって、後述の例で示すように、通常は列名のエイリアスを指定することをお勧めします。

%ODBCOUT による変換にかかわらず、日付は、日付フィールドまたは関数によって返されるデータ型に依存しています。%ODBCOUT は、[CURDATE](#)、[CURRENT\\_DATE](#)、[CURTIME](#)、および [CURRENT\\_TIME](#) の値を変換します。[CURRENT\\_TIMESTAMP](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)、および [\\$HOROLOG](#) の値は変換しません。

%ODBCOUT は、InterSystems SQL の拡張です。

表示形式オプションの詳細は、“[データ表示オプション](#)” を参照してください。

## 引数

### *expression*

変換される式。フィールド名、フィールド名を含む式、または変換可能なデータ型で値を返す関数 (DATE や %List など)。ストリーム・フィールドにすることはできません。

## 例

以下の例は、既定の表示形式 %ODBCIN および同じフィールドの %ODBCOUT 形式を示しています。

### SQL

```
SELECT FavoriteColors,%ODBCIN(FavoriteColors) AS InVal,  
       %ODBCOUT(FavoriteColors) AS OutVal  
FROM Sample.Person
```

## 関連項目

- ・ [%EXTERNAL](#)、[%INTERNAL](#)、[%ODBCIN](#)
- ・ SQL の概念: [データ型](#)、[日付/時刻文](#)

## %OID (SQL)

---

ID フィールドの OID を返すスカラー関数です。

### 構文

```
%OID(id_field)
```

### 概要

%OID は、フィールド名を取り、オブジェクトの整形形式 OID (オブジェクト ID) を返します。このフィールドは、ID フィールドまたは参照フィールド (外部キー・フィールド) のいずれかである必要があります。id\_field にその他のタイプのフィールドを指定すると、SQLCODE -1 エラーが発行されます。

### 引数

**id\_field**

ID フィールド、または参照フィールドのフィールド名。

### 例

以下の例は、参照フィールドで使用される %OID を示しています。

### SQL

```
SELECT Name, Spouse, %OID(Spouse)
FROM Sample.Person
WHERE Spouse IS NOT NULL
```

### 関連項目

- ・ [SELECT](#)
- ・ [%OBJECT](#)



# PI (SQL)

円周率の定数値を返すスカラ数値関数です。

## 構文

```
{fn PI()}  
{fn PI}
```

## 概要

PI は、引数を取りません。これは、有効桁数が 19 で小数桁数が 18 の [NUMERIC](#) データ型として、数学定数 pi を返します。

PI は、ODBC スカラ関数 (中括弧) 構文でのみ呼び出すことができます。引数の括弧はオプションです。

## 例

以下の例は、両方とも pi 値を返します。

### SQL

```
SELECT {fn PI()} AS ExactPi
```

### SQL

```
SELECT {fn PI} AS ExactPi
```

これは、3.141592653589793238 を返します。

## 関連項目

- SQL 関数: [ROUND](#)
- ObjectScript 特殊変数: [\\$ZPI](#)

## \$PIECE (SQL)

区切り文字で識別される部分文字列を返す文字列関数です。

### 構文

```
$PIECE(string-expression,delimiter[,from[,to]])
```

### 概要

\$PIECE は、指定された部分文字列 (断片) を string-expression から返します。返される部分文字列は、使用する引数によって異なります。

- ・ \$PIECE(string-expression,delimiter) は、string-expression の最初の部分文字列を返します。delimiter が string-expression 内に見つかった場合、最初の delimiter の前にある部分文字列が返されます。delimiter が string-expression 内に見つからない場合、返される部分文字列は string-expression です。
- ・ \$PIECE(string-expression,delimiter,from) は、string-expression の n 番目の部分文字列を返します。整数 n は from 引数で指定し、部分文字列は delimiter で区切ります。区切り文字は返されません。
- ・ \$PIECE(string-expression,delimiter,from,to) は、from で指定された部分文字列から to で指定された部分文字列までの範囲にある部分文字列を返します。4 つの引数の \$PIECE が返す文字列は、from から to までの部分文字列の間にあり、それらを区切っている delimiter を含んだすべての文字列を返します。to が部分文字列数よりも大きい場合、返される部分文字列には、string-expression 文字列の末尾までのすべての部分文字列が含まれます。

### 引数

#### string-expression

部分文字列を返す文字列。文字列リテラル、変数名、または文字列として評価される任意の有効な式を指定できます。

通常、文字列には、区切り文字として使用される文字 (または文字列) が含まれます。この文字または文字列は、string-expression 内でデータ値として使用することはできません。

ターゲット文字列として NULL 文字列 (NULL) を指定した場合、\$PIECE は <null>、つまり NULL 文字列を返します。

#### delimiter

string-expression 内の部分文字列を区切るために使用する検索文字列。(引用符で囲まれた) 数値または文字列リテラル、変数名、あるいは文字列として評価される式を指定できます。

通常、区切り文字には、文字列データ内で決して使用されることがなく、部分文字列を区切る文字としてのみ使用する特定の文字を設定します。区切り文字には、複数文字からなる検索文字列を指定することもできますが、それを構成する個々の文字は文字列データ内で使用できます。

区切り文字として NULL 文字列 (NULL) を指定した場合、\$PIECE は <null>、つまり NULL 文字列を返します。

#### from

string-expression 内の部分文字列の番号で、1 から始まることを指定する引数 (オプション)。これは、正の整数、整数変数の名前、または正の整数に評価される式でなければなりません。部分文字列は区切り文字で区切ります。

- ・ from 引数が省略された場合または 1 に設定された場合、\$PIECE は string-expression の最初の部分文字列を返します。指定された区切り文字が string-expression にない場合、from 値が 1 であれば、string-expression を返します。
- ・ from 引数がカウントによって string-expression の最後の部分文字列を指す場合、この部分文字列の後における区切り文字の有無に関係なく、この部分文字列が返されます。

- ・ to 引数が指定されず、from の値が NULL、空文字列、ゼロ、または負数の場合、\$PIECE は NULL 文字列を返します。しかし、to 引数が指定された場合、\$PIECE はその from 値を from=1 である場合と同じように扱います。
- ・ from の値が string-expression の部分文字列数よりも大きい場合、\$PIECE は NULL 文字列を返します。

引数 from が引数 to と共に使用される場合、from は文字列として返される部分文字列の範囲の先頭を指定し、to の値よりも小さくしなければなりません。

## to

from 引数によって先頭が指定された範囲を終了する、string-expression 内の部分文字列の番号を指定する引数 (オプション)。返される文字列には、from の位置の部分文字列と to の位置の部分文字列に加えて、その両者の間の部分文字列とそれらを区切る区切り文字が含まれます。to 引数は、正の整数、整数変数の名前、または正の整数に評価される式でなければなりません。引数 to は必ず from と共に使用し、from の値よりも大きい必要があります。

- ・ from が to よりも小さい場合、\$PIECE は、from 位置の部分文字列と to 位置の部分文字列を含む、この範囲内にあるすべての区切られた部分文字列からなる文字列を返します。この返された文字列には、範囲内にある部分文字列と区切り文字が含まれます。
- ・ 区切られた部分文字列の数よりも to が大きい場合、返される値には、from 位置の部分文字列から string-expression 文字列の末尾までのすべての文字列データ (部分文字列と区切り文字) が含まれます。
- ・ from が to と同じ場合、from 位置の部分文字列が返されます。
- ・ from が to よりも大きい場合、\$PIECE は NULL 文字列を返します。
- ・ to が NULL 文字列 (NULL) の場合、\$PIECE は NULL 文字列を返します。

## 例

以下の例は、"," 区切り文字で識別された最初の部分文字列である 'Red' を返します。

### SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black','(',')')
```

以下の例は、"," 区切り文字で識別された 3 番目の部分文字列である 'Blue' を返します。

### SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black','(',')',3)
```

以下の例は、colorlist の 3 番目から 5 番目の要素である 'Blue,Yellow,Orange' を、"," で区切って返します。

### SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black','(',')',3,5)
```

以下の \$PIECE 関数はどちらも '123' を返します。これは、from が 1 のとき、引数が 2 つの形式と 3 つの形式が同等であることを示しています。

### SQL

```
SELECT $PIECE('123#456#789','#') AS TwoArg
```

### SQL

```
SELECT $PIECE('123#456#789','#',1) AS ThreeArg
```

以下の例は、複数文字の区切り文字列 '#-' を使用して、3 番目の部分文字列 '789' を返します。この場合、区切り文字列を構成する個々の文字の '#' と '-' はデータ値として使用でき、指定された文字シーケンス (#-) のみが区切り文字列として認識されます。

#### SQL

```
SELECT $PIECE('1#2-3#-#45##6#-#789','#-',3)
```

以下の例は 'MAR;APR;MAY' を返します。これは 3 番目から 5 番目の部分文字列で構成され、delimiter ';' で区別されています。

#### SQL

```
SELECT $PIECE('JAN;FEB;MAR;APR;MAY;JUN',';',3,5)
```

以下の例は、\$PIECE を使用して従業員名と仕入先連絡先名から名字を抽出し、従業員が仕入先と同じ名字を持つインスタンスを返す JOIN を実行します。

#### SQL

```
SELECT E.Name,V.Contact
FROM Sample.Employee AS E INNER JOIN Sample.Vendor AS V
ON $PIECE(E.Name,'')=$PIECE(V.Contact,'')
```

## メモ

### データ値をアンパックする \$PIECE の使用法

\$PIECE は、通常、区切り文字で区切られた複数のフィールドを含むデータ値を“アンパック”するために使用されます。一般的な区切り文字には、スラッシュ (/)、コンマ (,)、スペース ( )、およびセミコロン (;) があります。以下の例の値は、\$PIECE を使用すると効率的に処理できます。

```
'John Jones/29 River St./Boston MA, 02095'
'Mumps;Measles;Chicken Pox;Diphtheria'
'45.23,52.76,89.05,48.27'
```

### \$PIECE および \$LENGTH

2 つの引数の \$LENGTH は、区切り文字に基づいて、文字列の部分文字列数を返します。\$LENGTH を使用して文字列の部分文字列数を調べ、\$PIECE を使用して個々の部分文字列を抽出します。

### \$PIECE および \$LIST

\$PIECE 関数と \$LIST 関数で使用するデータ格納方法には互換性がなく、組み合わせて使用することはできません。例えば、\$LISTBUILD を使用して作成されたリストに対して \$PIECE を使用すると、予測できない結果を生じる場合があります。そのため、使用しないでください。このことは、SQL 関数および対応する ObjectScript 関数の両方に当てはまります。

\$LIST 関数は、特定の区切り文字を使用せずに部分文字列を指定します。区切り文字または区切り文字シーケンスの設定が特定のデータ型 (ビット文字列データなど) に対して不適切な場合は、\$LISTBUILD と \$LIST SQL 関数を使用して部分文字列の格納と検索を行ってください。

### NULL 値

\$PIECE は、値が NULL 文字列 (NULL) の部分文字列と、存在しない部分文字列を区別しません。両方とも、<null>、つまり NULL 文字列値が返されます。例えば、以下の 2 つの例はいずれも、from 値の 7 に対して NULL 文字列値を返します。

#### SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',' ',7)
```

## SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',' ',',',7)
```

最初の例では、7 番目の部分文字列がないので、NULL 文字列が返されます。2 番目の例では、string-expression 文字列の最後に区切り文字で示された 7 番目の部分文字列があり、この 7 番目の部分文字列の値が NULL 文字列です。

以下の例は、string-expression 内に NULL 値がある場合を示しています。これは部分文字列 3 を抽出します。この部分文字列は存在しますが、NULL 文字列です。

## SQL

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black',' ',',',3)
```

以下の例も、NULL 文字列を返します。指定された部分文字列が存在しないためです。

## SQL

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black',' ',',',0)
```

## SQL

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black',' ',',',8,20)
```

以下の例では、string-expression 文字列内に delimiter が使用されていないため、\$PIECE 関数は string-expression 文字列全体を返します。

## SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',' ','#')
```

## 入れ子になった \$PIECE 操作

複雑な取り出しを実行するには、お互いの中で \$PIECE 参照を入れ子にすることができます。内側の \$PIECE で返された部分文字列が外側の \$PIECE の操作対象になります。各 \$PIECE はそれぞれの区切り文字を使用します。例えば、以下の例は、州の省略形の 'MA' を返します。

## SQL

```
SELECT $PIECE($PIECE('John Jones/29 River St./Boston MA 02095','/',3),' ',',',2)
```

以下は、入れ子になった \$PIECE 操作のもう 1 つの例で、区切り文字の階層が使用されています。最初に、内側の \$PIECE がキャレット (^) の区切り文字を使用して、文字列の 2 番目の部分文字列である 'A,B,C' を探します。次に、外側の \$PIECE がコンマ (,) 区切り文字を使用して、部分文字列 'A,B,C' の最初と 2 番目の部分文字列 ('A,B') を返します。

## SQL

```
SELECT $PIECE($PIECE('1,2,3^A,B,C^@#!','^',2),' ',',',1,2)
```

## 関連項目

- SQL 関数: [\\$EXTRACT \\$FIND \\$LENGTH \\$LIST](#)
- ObjectScript 関数: [\\$EXTRACT \\$FIND \\$LENGTH \\$LIST \\$PIECE](#)

## %PLUS (SQL)

数値をキャノニック照合形式に変換する照合関数です。

### 構文

```
%PLUS(expression)
```

```
%PLUS expression
```

### 説明

%PLUS は、数値や数値文字列をキャノニック形式に変換してから、これらの *expression* 値を数値照合順に返します。

数値には、先頭と末尾のゼロ、先頭にある複数のプラス符号とマイナス符号、1 つの小数点 (.)、および指数記号 (E) を指定できます。キャノニック形式で、すべての算術演算が実行され、指数が展開され、単体のマイナス符号が先頭に付くか符号なしになり、先頭と末尾のゼロが削除されます。

数値リテラルは、文字列を囲む区切り文字を付けて指定することも、付けなくて指定することもできます。数値以外の文字が文字列にあると、%PLUS では、最初の数値以外の文字で数値を切り捨て、数値部分をキャノニック形式で返します。数値でない文字列 (数値以外の文字で始まる文字列) を 0 として返します。%PLUS はさらに、NULL を 0 として返します。

%PLUS は、InterSystems SQL の拡張機能であり、SQL 検索クエリ用として使用するものです。

%SYSTEM.Util クラスの Collation() メソッドを使用すると、ObjectScript で同じ照合変換を実行できます。

#### ObjectScript

```
WRITE $SYSTEM.Util.Collation("++007.500",3)
```

%PLUS を %MVR と比べると、%MVR では文字列内の数値部分文字列に基づいて文字列をソートしている点が異なります。

### 引数

#### *expression*

列名、数値や文字列リテラル、算術式、または別の関数の結果となる式。基本となるデータ型は、任意の文字タイプとすることができます。

### 例

以下の例は %PLUS を使用して、Home\_Street の住所を数値の順に返します。

#### SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %PLUS(Home_Street)
```

上記の例では、ストリート・アドレスの整数部分が昇順の数値順に配列されています。これを以下の ORDER BY の例と比較してください。こちらでは、レコードは照合順のストリート・アドレスで配列されています。

#### SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY Home_Street
```

## 関連項目

- ・ [%EXACT](#) 照合関数
- ・ [%MINUS](#) 照合関数
- ・ [照合](#)

# POSITION (SQL)

文字列の中で部分文字列が占める位置を返す文字列関数です。

## 構文

```
POSITION(substring IN string)
```

## 概要

POSITION は、string 内の substring の最初の場所の位置を返します。位置情報は、整数として返されます。substring が見つからない場合、0 (ゼロ) が返されます。いずれかの引数に NULL 値が渡されると、POSITION は NULL を返します。

POSITION は、大文字と小文字を区別します。大小文字変換関数の 1 つを使用して、文字列の大文字と小文字のインスタンスをどちらも配置します。

## POSITION、INSTR、CHARINDEX、および \$FIND

POSITION、INSTR、CHARINDEX、および \$FIND はすべて文字列内の指定された部分文字列を検索し、最初に一致した位置に対応する整数位置を返します。CHARINDEX、POSITION、および INSTR は、一致した部分文字列の最初の文字の整数位置を返します。\$FIND は、最初に一致した部分文字列の次の文字の整数位置を返します。CHARINDEX、\$FIND、および INSTR では、部分文字列の検索を開始する位置を指定することができます。INSTR では、その開始位置から数えて何個目の部分文字列かを指定することもできます。

以下に、4 つの関数にすべてのオプション引数を指定した例を示します。string および substring の位置はそれぞれの関数で異なります。

## SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br', 'The broken brown briefcase', 6) AS Charindex,
       $FIND('The broken brown briefcase', 'br', 6) AS Find,
       INSTR('The broken brown briefcase', 'br', 6, 2) AS Inst
```

部分文字列を検索する関数のリストは、“[文字列操作](#)”を参照してください。

## 引数

### substring

検索する部分文字列。式は、列の名前、文字列リテラル、または他のスカラ関数の結果となります。基本データ型は、(CHAR や VARCHAR2 など) さまざまな文字タイプとして表示されます。

### IN string

substring の検索が行われる文字列式。

## 例

以下の例は、“b” が文字列の 11 番目の文字であるため 11 を返します。

## SQL

```
SELECT POSITION('b' IN 'The quick brown fox') AS PosInt
```

以下の例は、Sample.Person テーブル内の各名前に対する姓 (苗字) の長さを返します。残りの名前フィールドから姓を区切るために使用するコンマを配置し、次にその位置から 1 を減算します。



## SQL

```
SELECT Name,  
POSITION(' ' IN Name)-1 AS LNameLen  
FROM Sample.Person
```

以下の例では、Sample.Person テーブル内の各名前にある文字 “B” の最初のインスタンスの位置を返します。POSITION は大文字と小文字を区別するため、%SQLUPPER 関数を使用して、検索を実行する前にすべての名前の値を大文字に変換します。%SQLUPPER は文字列の最初に空白スペースを加えるため、この例では 1 を減算して、実際の文字位置を取得します。指定した文字列を配置できないと、検索はゼロ (0) を返します。この例では、1 を減算しているので、これらの検索で表示される値は -1 です。

## SQL

```
SELECT Name,  
POSITION('B' IN %SQLUPPER(Name))-1 AS BPos  
FROM Sample.Person
```

## 関連項目

- ・ [CHARINDEX](#) 関数
- ・ [\\$FIND](#) 関数
- ・ [INSTR](#) 関数
- ・ [文字列操作](#)

## POWER (SQL)

与えられた式の値を、指定の数で累乗して返す数値関数です。

### 構文

```
POWER(numeric-expression,power)  
{fn POWER(numeric-expression,power)}
```

### 概要

POWER は、ある数を累乗します。これは、有効桁数が 36 で小数桁数が 18 の値を返します。

POWER は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として呼び出せる点に注意してください。

POWER は、非数値文字列をいずれかの引数の 0 として解釈します。詳細は、“[数値としての文字列](#)”を参照してください。引数に NULL 値を渡すと、POWER は NULL を返します。

*numeric-expression* と *power* の以下を除くすべての組み合わせが有効です。

- ・ POWER(0, -*m*) : *numeric-expression* が 0 で *power* が負であると、SQLCODE -400 エラーとなります。
- ・ POWER(-*n*, .*m*) : *numeric-expression* が負で、*power* が小数であると、SQLCODE -400 エラーとなります。

### 引数

#### *numeric-expression*

基本の数値。正の整数、負の整数、または小数を指定できます。

#### *power*

*numeric-expression* を累乗する指数。正の整数、負の整数、または小数を指定できます。

POWER は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。POWER は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

### 例

以下の例は、5 を 3 乗します。

#### SQL

```
SELECT POWER(5,3) AS Cubed
```

これは、125 を返します。

以下の埋め込み SQL の例は、2 の最初の 16 のべき乗を返します。

#### ObjectScript

```
SET a=1  
WHILE a<17 {  
  &sql(SELECT {fn POWER(2,:a)}  
  INTO :b)  
  IF SQLCODE'=0 {  
    WRITE !,"Error code ",SQLCODE  
    QUIT }  
  ELSE {  
    WRITE !,"2 to the ",a," = ",b  
    SET a=a+1 }  
}
```

## 関連項目

- ・ SQL 関数 : [EXP LOG10 SQRT SQUARE](#)
- ・ ObjectScript 関数: [\\$ZPOWER](#)
- ・ ObjectScript [指数演算子 \(\\*\\*\)](#)

## PREDICT (SQL)

特定のトレーニングされたモデルを適用して、提供された各入力行の結果を予測する関数です。

### 構文

```
PREDICT( model-name )
PREDICT( model-name USE trained-model-name )
PREDICT( model-name WITH feature-columns-clause )
PREDICT( model-name USE trained-model-name \
    WITH feature-columns-clause )
```

### 説明

PREDICT は、トレーニングされた機械学習モデルを指定されたクエリに適用した結果を返します。これは、行単位で実行されます。

### USE

トレーニングされたモデルに、USE によって明示的に名前が付けられていない場合、PREDICT は指定されたモデル定義の既定のトレーニング済みモデルを使用します。

例えば、複数のモデルをトレーニングする場合は、次のようになります。

```
CREATE MODEL MyModel PREDICTING( label ) FROM data
TRAIN MODEL MyModel AS FirstModel
TRAIN MODEL MyModel AS SecondModel NOT DEFAULT
```

FirstModel は、MyModel の既定のモデルです。つまり、PREDICT クエリは予測に FirstModel を使用します。SecondModel を使用するよう指定する場合は、次のようになります。

```
PREDICT( MyModel USE SecondModel )
```

### WITH

FROM 節を使用してデータセットを指定する際、PREDICT 関数は、暗黙的に、指定したデータセットの特徴列を、モデル内の特徴列にマップします。WITH 節を使用することにより、以下のいずれかを行うことができます。

- データセットとモデル間の列のマッピングを指定します。以下に例を示します。

```
SELECT PREDICT(Trained_Model WITH age = year) FROM dataset
```

このクエリは、Trained\_Model の age 列を dataset の year 列に合わせます。

中括弧を使用して、複数の列をマップできます。

```
SELECT PREDICT(Trained_Model WITH {age = year, income = salary}) FROM dataset
```

WITH 節内でのこれらの列の順序は重要ではなく、欠落している列名は FROM 節から取得されます。

- 予測を行うための引数のリストを指定します。この形式の WITH を使用する場合、FROM 節は指定しません。以下に例を示します。

```
SELECT PREDICT(Flower_Model WITH (5.1, 3.5, 1.4, 0.2, 'setosa'))
```

このクエリは、式 (5.1, 3.5, 1.4, 0.2, 'setosa') に対して Flower\_Model を使用して予測を行います。

引数は、CREATE MODEL 文で指定したとおりの順序にする必要があります。欠落している引数は、空のコンマで指定できます。以下に例を示します。

```
SELECT PREDICT(Flower_Model WITH (5.1, , 1.4, , 'setosa'))
```

中括弧を使用して、複数の引数セットを指定できます。

```
SELECT PREDICT(Flower_Model WITH ({5.1, 3.5, 1.4, 0.2, 'setosa'}, {6.4, 3.2, 4.3, 1.2, 'versicolor'}))
```

## 必要なセキュリティ特権

PREDICT を呼び出すには、%USE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%USE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## 引数

### model-name

モデルの名前。

### USE trained-model-name

既定でないトレーニング済みモデルの名前を指定する引数 (オプション)。上記の[説明](#)を参照してください。

### WITH feature-columns-clause

オプションの引数です。トレーニングされたモデルの入力として提供する特定の列。上記の[説明](#)を参照してください。

## 例

```
CREATE MODEL HousePriceModel PREDICTING( HousePrice ) FROM housing_data_2019
TRAIN MODEL HousePriceModel
SELECT * FROM housing_data_2020 WHERE PREDICT( HousePriceModel ) > 500000
```

```
CREATE MODEL PatientReadmission PREDICTING ( IsReadmitted ) FROM patient_data
TRAIN MODEL PatientReadmission
SELECT *, PREDICT( PatientReadmission ) FROM new_patient_data
```

## 関連項目

- ・ [TRAIN MODEL](#)、[PROBABILITY](#)

## PROBABILITY (SQL)

特定のトレーニングされたモデルを適用して、指定されたラベルが予測されたラベル値である確率を返す関数です。これにより、その値の予測の相対的な強度を評価できます。

### 構文

```
PROBABILITY ( model-name FOR label-value )

PROBABILITY ( model-name USE trained-model-name
              FOR label-value )

PROBABILITY ( model-name FOR label-value
              WITH feature-columns-clause )

PROBABILITY ( model-name USE trained-model-name
              FOR label-value WITH feature-columns-clause ] )
```

### 説明

PROBABILITY 関数は、特定のモデルを特定のテーブルに適用して、テーブル内の各行について、モデルが指定された値を予測する確率を返します。この確率は、0 から 1 の値として返されます。この関数は、(回帰モデルではなく) 分類モデルでのみ使用できます。

### FOR

FOR は、PROBABILITY が確率を求める出力値を提供します。

例を以下に示します。

```
SELECT * FROM flower_dataset WHERE PROBABILITY(iris_flower FOR 'iris-setosa') > 0.6
```

iris\_flower モデルを使用して、結果が “iris-setosa” である確率が 0.6 を超える flower\_dataset の各行を返します。

FOR を省略すると、値は 1 になります。例を以下に示します。

```
SELECT PROBABILITY(IsSpam) FROM email_data
```

暗黙的に、次のクエリが生成されます。

```
SELECT PROBABILITY(IsSpam FOR 1) FROM email_data
```

FOR に指定した値が、指定されたトレーニング済みモデルに対して無効な場合、SQLCODE -400 エラーが発生して次のメッセージが表示されます。

```
[%msg: <PREDICT execution error: ERROR #2853: Specified positive label value not found
in the dataset.>]
```

### USE

トレーニングされたモデルに、USE によって明示的に名前が付けられていない場合、PROBABILITY は指定されたモデル定義の既定のトレーニング済みモデルを使用します。

例えば、複数のモデルをトレーニングする場合は、次のようになります。

```
CREATE MODEL MyModel PREDICTING( label ) FROM data
TRAIN MODEL MyModel AS FirstModel
TRAIN MODEL MyModel AS SecondModel NOT DEFAULT
```

FirstModel は、MyModel の既定のモデルです。つまり、PROBABILITY クエリは予測に FirstModel を使用します。SecondModel を使用するよう指定する場合は、次のようになります。

```
PROBABILITY( MyModel FOR label-value USE SecondModel )
```

## WITH

PROBABILITY は、WITH 節がない場合に、指定されたデータセットの特徴列をモデルの特徴列に暗黙的にマッピングする洗練された関数です。WITH 節を使用すると、データセットとモデル間の列のマッピングを指定できます。例を以下に示します。

```
SELECT PROBABILITY(iris_flower FOR 'iris-setosa' WITH petal_length = length_petal) FROM flower_dataset
```

このクエリは、iris\_flower モデルの petal\_length 列を flower\_dataset の length\_petal 列に合わせます。

## 必要なセキュリティ特権

PROBABILITY を呼び出すには、%USE\_MODEL 特権が必要です。ない場合、SQLCODE -99 エラーになります (特権違反)。%USE\_MODEL 特権を割り当てるには、[GRANT](#) コマンドを使用します。

## 引数

### model-name

トレーニングされたモデルの名前。

### FOR label-value

出力値。上記の[説明](#)を参照してください。

### USE trained-model-name

オプションの引数です。既定でないトレーニング済みモデルの名前。上記の[説明](#)を参照してください。

### WITH feature-columns-clause

オプションの引数です。トレーニングされたモデルの入力として提供する特定の列。上記の[説明](#)を参照してください。

## 例

```
CREATE MODEL PatientReadmission PREDICTING ( IsReadmitted ) FROM patient_data
TRAIN MODEL PatientReadmission
SELECT * FROM new_patient_data WHERE PROBABILITY( PatientReadmission FOR 1 ) > 0.8
```

## 関連項目

- ・ [TRAIN MODEL](#)、[PREDICT](#)

## QUARTER (SQL)

日付式に対して年の四半期を整数として返す日付関数です。

### 構文

```
{fn QUARTER(date-expression)}
```

### 概要

QUARTER は 1 ～ 4 の整数を返します。四半期は、InterSystems IRIS 日付整数、\$HOROLOG 値や \$ZTIMESTAMP 値、ODBC 形式の日付文字列、またはタイムスタンプに基づいて計算されます。

date-expression タイムスタンプには、データ型 %Library.PosixTime (エンコードされた 64 ビットの符号付き整数) またはデータ型 %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

四半期の期間は以下のとおりです。

四半期	期間
1	1 月 1 日から 3 月 31 日まで (90 日または 91 日)
2	4 月 1 日から 6 月 30 日まで (91 日)
3	7 月 1 日から 9 月 30 日まで (92 日)
4	10 月 1 日から 12 月 31 日まで (92 日)

QUARTER は、日付/時刻文字列の月の部分に基づきます。ただし、date-expression のすべてが検証され、1 から 12 までの範囲の月、および指定した月と年の有効な日の値を含む必要があります。それ以外の場合には、SQLCODE - 400 エラー <ILLEGAL VALUE> が生成されます。date-expression の時刻部分は省略可能ですが、存在する場合は有効である必要があります。

DATEPART または DATENAME 関数を使用しても、同じ四半期情報を取得できます。DATEADD または TIMESTAM-PADD 関数を使用して、指定した四半期数で日付をインクリメントできます。

この関数は、ObjectScript から QUARTER() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.QUARTER(date-expression)
```

### 引数

#### date-expression

列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式。

### 例

以下の例は、与えられた日付 (2 月 22 日) が 1 年の第 1 四半期内にいるので、どちらも 1 を返します。

#### SQL

```
SELECT {fn QUARTER('2018-02-22')} AS ODBCDateQ
```

#### SQL

```
SELECT {fn QUARTER(64701)} AS HorologDateQ
```

以下の例は、すべて現在の四半期を返します。



## SQL

```
SELECT {fn QUARTER({fn NOW()})} AS Q_Now,  
       {fn QUARTER(CURRENT_DATE)} AS Q_CurrD,  
       {fn QUARTER(CURRENT_TIMESTAMP)} AS Q_CurrTstamp,  
       {fn QUARTER($ZTIMESTAMP)} AS Q_ZTstamp,  
       {fn QUARTER($HOROLOG)} AS Q_Horolog
```

## 関連項目

- ・ SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DATEADD](#)、[MONTH](#)、[TO\\_DATE](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

## RADIANS (SQL)

角度をラジアンに変換する数値関数です。

### 構文

```
RADIANS(numeric-expression)  
{fn RADIANS(numeric-expression)}
```

### 説明

RADIANS は、度単位で角度測定を行い、対応する角度測定をラジアン単位で返します。RADIANS は、NULL 値を渡すと NULL を返します。

返される値は、既定の有効桁数が 36 で、既定の小数桁数が 18 です。

ラジアンを度に変換するには、[DEGREES](#) 関数を使用できます。

### 引数

#### *numeric-expression*

度単位の角度のメジャー。数値に解決される式です。

RADIANS は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。RADIANS は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

RADIANS は、標準スカラ関数または {} 括弧構文による ODBC スカラ関数として指定できます。

### 例

以下の埋め込み SQL の例は、30 度の増分で 0 から 365 の角度値に対応する同等のラジアンを返します。

#### ObjectScript

```
SET a=0  
WHILE a<366 {  
&sql(SELECT RADIANS(:a) INTO :b)  
IF SQLCODE'=0 {  
    WRITE !,"Error code ",SQLCODE  
    QUIT }  
ELSE {  
    WRITE !,"degrees ",a," = radians ",b  
    SET a=a+30 }  
}
```

### 関連項目

- SQL 関数: [CONVERT](#)、[DEGREES](#)、[TO\\_NUMBER](#)

# REPEAT (SQL)

指定した回数だけ文字列を繰り返す文字列関数です。

## 構文

```
REPEAT(expression,repeat-count)  
{fn REPEAT(expression,repeat-count)}
```

## 概要

REPEAT は、連結した *expression* の *repeat-count* インスタンスの文字列を返します。

*expression* が NULL の場合、REPEAT は NULL を返します。*expression* が空文字列の場合、REPEAT は空文字列を返します。

*repeat-count* が小数の場合、整数部のみを使用します。*repeat-count* が0 の場合、REPEAT は空文字列を返します。*repeat-count* が負の数の場合、または数値でない文字列の場合、REPEAT は NULL を返します。

## 引数

### *expression*

繰り返される文字列式

### *repeat-count*

繰り返す回数。整数で表現します。

## 例

以下の例は、REPEAT の 2 つの形式を示します。どちらの例も、文字列 'BANGBANGBANG' を返します。

### SQL

```
SELECT REPEAT('BANG',3) AS Tripled
```

### SQL

```
SELECT {fn REPEAT('BANG',3)} AS Tripled
```

## 関連項目

- ・ [REPLICATE](#)

## REPLACE (SQL)

文字列内の部分文字列を置換する文字列関数です。

### 構文

```
REPLACE(string,oldsubstring,newsubstring)
```

### 概要

REPLACE は部分文字列を検索して、一致するすべての文字列を置換します。マッチングでは、大文字と小文字が区別されます。一致が見つかれば、oldsubstring のすべてのインスタンスが newsubstring で置換されます。置換部分文字列は、置換対象となる部分文字列よりも長い場合もあります。部分文字列が見つからなければ、REPLACE は元の string を変更せずに返します。

REPLACE で返される値は、string のデータ型に関係なく、必ず VARCHAR データ型です。したがって、REPLACE(12.3, '.', '\_') などの置換操作が可能です。

REPLACE では string 引数、oldsubstring 引数または newsubstring 引数に [%Stream.GlobalCharacter](#) フィールドを使用することはできません。これを実行しようとすると、SQLCODE -37 エラーが生成されます。

空の文字列は文字列値です。そのため、引数の値には空の文字列を指定できます。ただし、ObjectScript の空文字列は、NULL として InterSystems SQL に渡されることに注意してください。

InterSystems SQL では、NULL はデータ値ではありません。このため、REPLACE 引数のいずれかに NULL を指定すると、一致があるかどうかに関係なく NULL が返されます。

この関数には Transact-SQL 実装との互換性があります。

### REPLACE、STUFF、および \$TRANSLATE

REPLACE および [STUFF](#) の両方とも、部分文字列の置換を実行します。REPLACE は、データ値によって部分文字列を検索します。STUFF は、文字列の位置と長さによって部分文字列を検索します。

REPLACE は、1 つの文字列と文字列をマッチングして置換します。[\\$TRANSLATE](#) は、文字と文字をマッチングして置換します。指定された 1 つまたは複数の単一文字のすべてのインスタンスを、指定された対応する置換単一文字に置き換えることができます。指定された 1 つまたは複数の単一文字のすべてのインスタンスを 1 つの文字列から削除することもできます。

既定では、3 つすべての関数で大文字と小文字が区別され、一致するすべてのインスタンスが置換されます。

部分文字列を検索する関数のリストは、["文字列操作"](#) を参照してください。

### 引数

#### string

部分文字列検索の検索対象となる文字列式。

#### oldsubstring

string 内で一致させる部分文字列。

#### newsubstring

oldsubstring を置換する部分文字列。

### 例

以下の例は、部分文字列 'P' のすべてのインスタンスを検索し、それを部分文字列 'K' で置き換えます。

## SQL

```
SELECT REPLACE('PING PONG','P','K')
```

以下の例は、部分文字列 'KANSAS' のすべてのインスタンスを検索し、それを部分文字列 'NEBRASKA' で置き換えます。

## SQL

```
SELECT REPLACE('KANSAS, ARKANSAS, NEBRASKA','KANSAS','NEBRASKA')
```

以下の例は、REPLACE が他の文字列値と同様、空の文字列 (" ) を処理することを示しています。

## SQL

```
SELECT REPLACE('',' ','Nothing'),
       REPLACE('PING PONG',' ','K'),
       REPLACE('PING PONG','P','')
```

以下の例は、REPLACE が NULL を返すことによって NULL 引数进行处理していることを示しています。一致のない最後の例を含み、以下の REPLACE 関数すべてが NULL を返します。

## SQL

```
SELECT REPLACE(NULL,'K','P'),
       REPLACE(NULL,NULL,'P'),
       REPLACE('PING PONG',NULL,'K'),
       REPLACE('PING PONG','P',NULL),
       REPLACE('PING PONG','Z',NULL)
```

以下の埋め込み SQL の例は、前述の NULL の例と同じです。ObjectScript の空文字列のホスト変数が、SQL 内で NULL として処理される方法を示しています。

## ObjectScript

```
SET a=""
&sql(SELECT
REPLACE(:a,'K','P'),
REPLACE(:a,:a,'P'),
REPLACE('PING PONG',:a,'K'),
REPLACE('PING PONG','P',:a),
REPLACE('PING PONG','Z',:a)
INTO :v,:w,:x,:y,:z)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"Output string=",v
WRITE !,"Output string=",w
WRITE !,"Output string=",x
WRITE !,"Output string=",y
WRITE !,"Output string=",z
```

## 関連項目

- [CHARINDEX](#) 関数
- [\\$FIND](#) 関数
- [STUFF](#) 関数
- [\\$TRANSLATE](#) 関数
- [文字列操作](#)

## REPLICATE (SQL)

---

指定した回数だけ文字列を繰り返す文字列関数です。

### 構文

```
REPLICATE(expression,repeat-count)
```

### 説明

REPLICATE は、指定した回数だけ文字列を繰り返します。

注釈 REPLICATE 関数は REPEAT 関数の別名です。REPLICATE は、TSQL の互換性を持たせるために提供されています。詳細は ["REPEAT"](#) を参照してください。

### 引数

#### **expression**

繰り返される文字列式

#### **repeat-count**

繰り返す回数。整数で表現します。

### 関連項目

・ [REPEAT](#)

# REVERSE (SQL)

文字列の文字を逆の順序で返すスカラ文字列関数です。

## 構文

```
REVERSE(string-expression)
```

## 説明

REVERSE は、*string-expression* の文字の順序を逆にして返します。例えば、'Hello World!' は、'!dlroW olleH' と返されます。これは、文字列の順序を逆にするだけで、その他の処理はしません。

返される文字列は、入力値のデータ型に関係なく、VARCHAR データ型となります。数値はキャノニック形式に変換され、数値文字列は順序を逆にする前にキャノニック形式に変換されることはありません。

先頭の空白や末尾の空白は順序を逆にすることによって影響されません。

NULL 値は逆にしても NULL です。

注釈 REVERSE は常に VARCHAR 文字列を返すので、一部のデータ型は順序を逆にすると無効になります。

- ・ 順序を逆にしたリストは有効なリストではなくなり、ストレージ形式から表示形式に変換することはできません。
- ・ 順序を逆にした日付は有効な日付ではなくなり、ストレージ形式から表示形式に変換することはできません。

## 引数

### *string-expression*

順序を逆にする文字列式。式は、列の名前や文字列リテラル、数値、または他のスカラ関数の結果を指定できます。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

## 例

以下の例では、Name フィールド値の順序を逆にします。ここでは、この結果、名前がミドル・イニシャルで並べ替えられます。

### SQL

```
SELECT Name, REVERSE(Name) AS RevName
FROM Sample.Person
ORDER BY RevName
```

Name および RevName は同じフィールドの異なる表現というだけなので、ORDER BY RevName と ORDER BY RevName, Name は同じ順序付けを実行します。

以下の例は、数値と数値文字列の順序を逆にします。

### SQL

```
SELECT REVERSE(+007.10) AS RevNum,
       REVERSE('+007.10') AS RevNumStr
```

以下の例は、\$DOUBLE の数値の順序を逆にします。

### SQL

```
SELECT
  CAST(1.1 AS DOUBLE) AS DoubleNumber,
  REVERSE(CAST(1.1 AS DOUBLE)) AS ReverseDouble
```

以下の例は、リストの順序を逆にしたときの動作を示しています。

### SQL

```
SELECT FavoriteColors,REVERSE(FavoriteColors) AS RevColors
FROM Sample.Person
```

以下の例は、日付の順序を逆にしたときの動作を示しています。

### SQL

```
SELECT DOB,%INTERNAL(DOB) AS IntDOB,REVERSE(DOB) AS RevDOB
FROM Sample.Person
```

## 関連項目

- ・ [CHAR](#)
- ・ [STRING](#)
- ・ [SUBSTRING](#)



# RIGHT (SQL)

文字列式の末尾（最右端）の文字から、指定された数の文字を返す、スカラ文字列関数です。

## 構文

```
{fn RIGHT(string-expression,count)}
```

## 概要

RIGHT は、*string-expression* の末尾（右端）から文字を *count* 文字数分返します。どちらかの引数に NULL 値を渡すと、RIGHT は NULL を返します。

RIGHT は {} 括弧構文による ODBC スカラ関数としてのみ使用できます。

## 引数

### *string-expression*

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、任意の文字タイプ（CHAR や VARCHAR など）とすることができます。

### *count*

*string-expression* の末尾（右端）から返す文字の数を指定する整数。

## 例

以下の例は、Sample.Person テーブル内の各名前の右端から 2 文字を返します。

### SQL

```
SELECT Name,{fn RIGHT(Name,2)}AS MiddleInitial  
FROM Sample.Person
```

以下の例は、RIGHT による文字列自体よりも長い *count* の処理方法を示しています。

### SQL

```
SELECT Name,{fn RIGHT(Name,40)} FROM Sample.Person
```

埋め込みは実行されません。

## 関連項目

・ [LEFT LTRIM RTRIM](#)

## ROUND (SQL)

指定された桁で数値の丸めまたは切り捨てを行う数値関数。

### 構文

```
ROUND(numeric-expr,scale[,flag])  
{fn ROUND(numeric-expr,scale[,flag])}
```

### 概要

この関数は、指定した小数桁数で数値の丸めまたは切り捨てを行うために使用できます。

ROUND は *numeric-expr* に対して小数点から数えて *scale* の桁に丸めまたは切り捨てを行います。丸めるとき、5 は常に切り上げられます。末尾のゼロは、ROUND による丸めまたは切り捨て操作後に削除されます。先頭のゼロは返されません。

- ・ *scale* が正の数である場合、丸めは小数点から右に数えたその桁数で実行されます。*scale* が小数桁数以上の数である場合、丸めやゼロの埋め込みは実行されません。
- ・ *scale* がゼロの場合は、最も近い整数に丸められます。つまり、丸めが小数点の右側 0 桁目で実行され、すべての小数桁と小数点が削除されます。
- ・ *scale* が負の数である場合、丸めは小数点から左に数えたその桁数で実行されます。*scale* が丸めの結果の整数桁数以上である場合は、ゼロが返されます。
- ・ *numeric-expr* がゼロ (ただし 00.00 や -0 などのように表されている) 場合、ROUND は *scale* の値とは無関係に小数桁のない 0 (ゼロ) を返します。
- ・ *numeric-expr* または *scale* が NULL の場合、ROUND は NULL を返します。

ROUND の戻り値は、末尾のゼロを削除して常に正規化されます。

### ROUND、TRUNCATE、および \$JUSTIFY

数値関数 ROUND と TRUNCATE の動作は似ています。両方とも小数桁または整数桁の有効桁数を減らすために使用できます。ROUND では、丸め (既定) または切り捨てを指定できます。TRUNCATE は丸めを行いません。ROUND は *numeric-expr* と同じデータ型を返します。TRUNCATE は *numeric-expr* をデータ型 NUMERIC として返します。ただしこれは、*numeric-expr* がデータ型 DOUBLE でない場合に限りです。DOUBLE である場合は、データ型 DOUBLE を返します。

ROUND では、指定された小数桁数への丸め (または切り捨て) を行いますが、その戻り値は常に正規化され、末尾のゼロが削除されます。例えば、ROUND(10.004, 2) は 10.00 ではなく 10 を返します。

TRUNCATE では、指定された小数桁数への切り捨てを行います。切り捨てによって末尾にゼロが生じた場合、これらの末尾のゼロは保持されます。ただし、*scale* が *numeric-expr* のキャノニック形式の小数桁数より大きい場合、TRUNCATE はゼロパディングを行いません。

固定の小数桁数への丸めが重要な場合、例えば金額を表す場合などは、\$JUSTIFY を使用します。\$JUSTIFY は、丸め処理後に指定数の末尾のゼロを返します。丸める桁数が小数桁数より大きい場合、\$JUSTIFY はゼロパディングを行います。また \$JUSTIFY では、DecimalSeparator 文字が数値列内で揃うように数値が右寄せされます。\$JUSTIFY は切り捨てを行いません。

### \$DOUBLE の数値

\$DOUBLE IEEE 浮動小数点数は、バイナリ表現でエンコードされます。ほとんどの 10 進数の小数は、このバイナリ表現では正確には表現できません。\$DOUBLE 値が *scale* 値および丸め *flag* (*flag*=0、既定値) と共に ROUND に入力されると、10 進小数の結果がバイナリで表現できないため、戻り値に *scale* で指定されているより多くの小数桁を含む場

合がよくあります。したがって、返り値は、以下の例で示すように、表現可能な最も近い \$DOUBLE 値に丸める必要があります。

## SQL

```
SELECT ROUND(1234.5678,2),ROUND($DOUBLE(1234.5678),2)
```

ROUND を使用して \$DOUBLE 値を切り捨てる場合 (flag=1)、\$DOUBLE の返り値は、scale で指定された小数桁数に切り捨てられます。また TRUNCATE 関数は、\$DOUBLE を scale で指定された小数桁数に切り捨てます。

ROUND を使用して \$DOUBLE 値を丸め、特定の scale を返す場合は、\$DOUBLE 値を 10 進表現に変換してから結果を丸める必要があります。

ROUND を flag=0 (丸める、既定) に設定すると、\$DOUBLE("INF") と \$DOUBLE("NAN") が空の文字列として返されます。

ROUND を flag=1 (切り捨てる) に設定すると、\$DOUBLE("INF") と \$DOUBLE("NAN") が INF と NAN として返されます。

## 引数

### numeric-expr

丸められる数字。数値式。

### scale

小数点からカウントして、どこまで丸めるかの桁数を指定する整数に評価される式。ゼロ、正の整数、または負の整数を指定できます。scale が小数の場合、最も近い整数に丸められます。

### flag

numeric-expr を丸めるか切り捨てるかを指定するブーリアン・フラグ (オプション)。0 は丸め、1 は切り捨てです。既定は 0 です。

## 例

以下の例は、scale の 0 (ゼロ) を使用していくつかの小数を整数に丸めます。5 は常に切り上げられます。

## SQL

```
SELECT ROUND(5.99,0) AS RoundUp,
       ROUND(5.5,0) AS Round5,
       {fn ROUND(5.329,0)} AS Roundoff
```

以下の例では、前の例と同じ小数を切り捨てます。

## SQL

```
SELECT ROUND(5.99,0,1) AS Trunc1,
       ROUND(5.5,0,1) AS Trunc2,
       {fn ROUND(5.329,0,1)} AS Trunc3
```

以下の ROUND 関数は、負の小数の丸めと切り捨てを行います。

## SQL

```
SELECT ROUND(-0.987,2,0) AS Round1,
       ROUND(-0.987,2,1) AS Trunc1
```

以下の例は、pi を小数桁数 4 に丸めます。

## SQL

```
SELECT {fn PI()} AS ExactPi, ROUND({fn PI()},4) AS ApproxPi
```

以下の例は、小数桁数より大きな scale を指定しています。

## SQL

```
SELECT {fn ROUND(654.98700,9)} AS Rounded
```

この場合は 654.987 が返されます (InterSystems IRIS が丸め処理を行う前に末尾のゼロを削除し、丸めもゼロの埋め込みも行われません)。

以下の例は、**Salary** の値を最も近い千ドルに丸めます。

## SQL

```
SELECT Salary,ROUND(Salary, -3) AS PayBracket  
FROM Sample.Employee  
ORDER BY Salary
```

**Salary** が 500 ドルより少ない場合は 0 (ゼロ) に丸められることに注意してください。

以下の例では、各 ROUND で、丸められる桁数と同じか、それを超える負の scale が指定されています。

## SQL

```
SELECT {fn ROUND(987,-3)} AS Round1,  
       {fn ROUND(487,-3)} AS Round2,  
       {fn ROUND(987,-4)} AS Round3,  
       {fn ROUND(987,-5)} AS Round4
```

最初の ROUND 関数では、丸めの結果の桁数が scale より大きくなるため、1000 が返されます。これ以外の 3 つの ROUND 関数は 0 (ゼロ) を返します。

## 関連項目

- ・ [\\$JUSTIFY 関数](#)
- ・ [TRUNCATE 関数](#)
- ・ [CEILING 関数](#)
- ・ [FLOOR 関数](#)
- ・ [MOD 関数](#)
- ・ ObjectScript関数: [\\$DOUBLE](#)、[\\$NORMALIZE](#)、[\\$NUMBER](#)

# RPAD (SQL)

指定された長さになるまで右側にパディングされた文字列を返す文字列関数です。

## 構文

```
RPAD(string-expression, length [, padstring])
```

## 説明

RPAD は、文字列式の最後にパディング文字を追加します。length 文字数にパディングされた文字列のコピーを返します。文字列式が length 文字より長い場合、返される文字列は length 文字まで切り捨てられます。

string-expression が NULL の場合、RPAD は NULL を返します。string-expression が空文字列 (') の場合、RPAD はすべてパディング文字で構成される文字列を返します。返される文字列は VARCHAR 型です。

RPAD をリンク・テーブルに対するクエリで使用できます。

RPAD は、先頭または末尾の空白を削除しません。先頭または末尾の空白も含めて文字列をパディングします。文字列をパディングする前に先頭または末尾の空白を削除するには、LTRIM、RTRIM、または TRIM を使用します。

## 引数

### string-expression

列の名前、文字列リテラル、ホスト変数、他のスカラー関数の結果などを表すことができる文字列式。VARCHAR データ型に変換できる任意のデータ型を指定できます。string-expression をストリームにはできません。

### length

返される文字列の文字数を指定する整数。

### padstring

入力の string-expression のパディングに使用される、文字列または 1 文字で構成される文字列 (オプション)。padstring 文字 (列) は、出力文字列が length 文字になるまで必要なだけ string-expression の右側に追加されます。padstring には、文字列リテラル、列、ホスト変数、他のスカラー関数の結果などを指定できます。省略すると、既定で空白スペース文字になります。

## 例

次の例では、列値を (必要に応じて) ^ 文字で右パディングして、長さが 16 の文字列を返します。ある Name 文字列は右パディングされ、ある Name 文字列は右側が切り捨てられて、長さ 16 の文字列が返されます。

### SQL

```
SELECT TOP 15 Name, RPAD(Name, 16, '^') AS Name16
FROM Sample.Person
```

次の例では、列値を (必要に応じて) ^^ 文字列で右パディングして、長さが 20 の文字列を返します。名前文字列のパディングは必要なだけ繰り返されます。返される文字列には、パディングされた文字列の一部が含まれる場合があります。

### SQL

```
SELECT TOP 15 Name, RPAD(Name, 20, '^=^') AS Name20
FROM Sample.Person
```

## 関連項目

- ・ [\\$JUSTIFY](#) 関数
- ・ [LPAD](#) 関数
- ・ [LTRIM](#) 関数
- ・ [RTRIM](#) 関数
- ・ [TRIM](#) 関数

# RTRIM (SQL)

末尾の空白を削除した文字列を返す文字列関数です。

## 構文

```
RTRIM(string-expression)
{fn RTRIM(string-expression)}
```

## 概要

RTRIM は文字列式から末尾の空白を削除し、その文字列を VARCHAR タイプとして返します。string-expression が NULL の場合、RTRIM は NULL を返します。string-expression が完全に空白スペースで構成されている文字列の場合、RTRIM は空文字列 (') を返します。

RTRIM は、削除される入力式のデータ型に関係なく、常に VARCHAR データ型を返します。

RTRIM は先頭の空白をそのまま残します。先頭の空白を削除するには LTRIM を使用します。先頭や末尾のあらゆるタイプの文字を削除するには、TRIM を使用します。文字列の末尾に空白またはその他の文字でパディングするには、RPAD を使用します。空白の文字列を作成するには、SPACE を使用します。

RTRIM は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として使用できる点に注意してください。

## 引数

### string-expression

列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

## 例

以下の例は、文字列から末尾の 5 つの空白を削除します。先頭の 5 つの空白はそのまま残します。

### SQL

```
SELECT {fn RTRIM("      Test string with 5 leading and 5 trailing spaces.      ")}
```

これは、以下を返します。

```
Before RTRIM
start:      Test string with 5 leading and 5 trailing spaces.      :end
After RTRIM
start:      Test string with 5 leading and 5 trailing spaces.:end
```

## 関連項目

- [LTRIM TRIM RPAD SPACE](#)

## SEARCH\_INDEX (SQL)

インデックスの Find() メソッドから一連の値を返す関数です。

### 構文

```
SEARCH_INDEX([[schema_name.]table-name.]index-name  
[,findparam[,...]])
```

### 説明

SEARCH\_INDEX は、*index-name* Find() メソッドを呼び出し、一連の値を返します。オプションでパラメータをこの Find() メソッドに渡すことができます。例えば、SEARCH\_INDEX(Sample.Person.NameIDX) は Sample.Person.NameIDXFind() メソッドを呼び出します。

SEARCH\_INDEX は、WHERE 節内で **%FIND** 述語と共に使用して、オブジェクトの *oref* を指定することができます。これは一連の値をカプセル化する抽象表現を提供します。これらの値は一般に、クエリ実行時に呼び出されるメソッドによって返される行 ID です。SEARCH\_INDEX は、インデックスの Find() メソッドを呼び出してこの *oref* を返します。

インデックスは、SQL 文が参照するテーブル内になければなりません。SQLCODE -151 エラーは、指定された *index-name* が SQL 文により使用されるテーブル内に存在していない場合にのみ生成されます。SQLCODE -152 エラーは、指定された *index-name* が完全修飾でなく、SQL 文により使用されるテーブル内であいまいである (既存の複数のインデックスを参照している可能性がある) 場合に生成されます。

インデックスが存在するものの、対応する Find() メソッドがない場合、ランタイム SQLCODE -149 エラー “SQL 関数でエラーが発生しました” が生成され、エラーの内容は <METHOD DOES NOT EXIST> となります。

SEARCH\_INDEX の使用法の詳細は、“[SQL Search](#)” テキスト検索ツールを参照してください。

### 引数

#### table-name

*index-name* が定義されている既存の [テーブル](#) の名前を指定する引数 (オプション)。ビューにすることはできません。テーブルの *schema\_name* はオプションです。指定を省略すると、FROM 節で指定されているすべてのテーブルが検索対象になります。

#### index-name

検索対象のインデックス。既存インデックスの [インデックス・マップ](#) の *SqlName*。

#### findparam

インデックスの Find() メソッドに渡されるパラメータまたはパラメータのコンマ区切りリスト (オプション)。

### 例

以下の例は、SEARCH\_INDEX での **%FIND** 述語の使用法を示しています。

#### SQL

```
SELECT Name FROM Sample.Person AS P  
WHERE P.Name %FIND SEARCH_INDEX(Sample.Person.NameIDX)
```

### 関連項目

- ・ [CREATE INDEX](#)
- ・ [%FIND 述語](#)



- ・ [%INSET 述語](#)
- ・ [インデックスの定義と構築](#)
- ・ [クエリ処理でのインデックスの使用](#)

## SECOND (SQL)

日付/時刻式に対して秒を返す時刻関数です。

### 構文

```
{fn SECOND(time-expression)}
```

### 概要

SECOND は、0 ～ 59 の整数を返します。小数の秒を返すこともできます。秒は、[\\$HOROLOG](#) 値や [\\$ZTIMESTAMP](#) 値、ODBC 形式の日付文字列 (時刻値なし)、またはタイムスタンプに基づいて計算されます。

*time-expression* タイムスタンプには、データ型 [%Library.PosixTime](#) (エンコードされた 64 ビットの符号付き整数) またはデータ型 [%Library.TimeStamp](#) (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

既定の時刻形式を変更するには、[SET OPTION](#) コマンドを使用します。

タイムスタンプ文字列 (yyyy-mm-dd hh:mm:ss) または [\\$HOROLOG](#) 文字列のどちらかを指定する必要があります。[\\$HOROLOG](#) 文字列は、完全な日付/時刻文字列 (63274,37279) か、[\\$HOROLOG](#) の時刻整数部分のみ (37279) にできます。時刻文字列 (hh:mm:ss) は指定できません。これは、実際の秒数に関係なく、必ず 0 を返します。

日付/時刻文字列の時刻部分は有効な時刻である必要があります。それ以外の場合には、SQLCODE -400 エラー <ILLEGAL VALUE> が生成されます。秒 (ss) 部分は、0 から 59 までの範囲の整数である必要があります。入力では、先頭のゼロはオプションです。出力では、先頭のゼロは抑制されます。

日付/時刻文字列の日付部分は検証されません。

秒部分が '0' または '00' の場合、SECOND は 0 秒を返します。時刻式がない ODBC 日付が指定されている場合、または時刻式の秒部分が完全に省略されている場合 ('hh'、'hh:mm'、'hh:mm:'、または 'hh::') にも、ゼロ秒が返されます。

[DATEPART](#) または [DATENAME](#) を使用して、同じ時刻情報を取得できます。

この関数は、ObjectScript から SECOND() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.SECOND(time-expression)
```

### 秒の小数部

*time-expression* で指定されると、SECOND は秒の小数部を返します。末尾のゼロは切り捨てられます。秒の小数部が指定されない場合 (例: 38.00)、小数点区切り文字も切り捨てられます。

時刻値 ([\\$HOROLOG](#)) の標準 InterSystems IRIS 内部表現は、秒の小数部をサポートしません。タイムスタンプは、秒の小数部をサポートします。

以下の SQL 関数、SECOND、CURRENT\_TIMESTAMP、DATENAME、DATEPART、および GETDATE は、秒の小数部をサポートします。CURTIME、CURRENT\_TIME、および NOW は秒の小数部をサポートしません。

SQL [SET OPTION](#) 文では、秒の小数部に既定の精度 (小数桁数) を設定できます。

ObjectScript [\\$ZTIMESTAMP](#) 特殊変数は、秒の小数部を表すために使用できます。ObjectScript 関数の [\\$ZDATETIME](#)、[\\$ZDATETIMEH](#)、[\\$ZTIME](#)、および [\\$ZTIMEH](#) は、秒の小数部をサポートします。

### 引数

#### *time-expression*

列の名前や、他のスカラ関数の結果、または文字列や日付や数値リテラルである式。タイムスタンプ文字列または [\\$HOROLOG](#) 文字列のどちらかに解決する必要があります。基本となるデータ型は、[%Time](#)、[%TimeStamp](#)、または [%PosixTime](#) とすることができます。

## 例

以下の例は、時刻式が 38 秒を表しているので、どちらも 38 を返します。

### SQL

```
SELECT {fn SECOND('2018-02-16 18:45:38')} AS ODBCSeconds
```

### SQL

```
SELECT {fn SECOND(67538)} AS HorologSeconds
```

以下の例は、.9 秒 を返します。先頭や末尾のゼロは、切り捨てられます。

### SQL

```
SELECT {fn SECOND('2018-02-16 18:45:00.9000')} AS Seconds_Given
```

以下の例は、日付/時刻文字列の秒部分が省略されているので、ゼロ秒を返します。

### SQL

```
SELECT {fn SECOND('2018-02-16 18:45')} AS Seconds_Given
```

以下の例は、日付/時刻文字列から時刻式が省略されているので、ゼロ秒を返します。

### SQL

```
SELECT {fn SECOND('2018-02-16')} AS Seconds_Given
```

以下の例はすべて、現在の時刻の秒部分を整数で返します。

### SQL

```
SELECT {fn SECOND(CURRENT_TIME)} AS Sec_CurrentT,  
       {fn SECOND({fn CURTIME()})} AS Sec_CurT,  
       {fn SECOND({fn NOW()})} AS Sec_Now,  
       {fn SECOND($HOROLOG)} AS Sec_Horolog,  
       {fn SECOND($ZTIMESTAMP)} AS Sec_ZTS
```

以下の例は、先頭のゼロが抑制されることを示します。最初の SECOND 関数は長さ 2 を返し、その他は長さ 1 を返します。省略された時刻はゼロ秒と見なされ、1 の長さを持ちます。

### SQL

```
SELECT LENGTH({fn SECOND('2018-02-15 11:45:22')}),  
       LENGTH({fn SECOND('2018-02-15 03:05:06')}),  
       LENGTH({fn SECOND('2018-02-15 3:5:6')}),  
       LENGTH({fn SECOND('2018-02-15')})
```

以下の例では、SECOND 関数が、ロケールに指定された TimeSeparator 文字を認識していることを示しています。

### SQL

```
SELECT {fn SECOND('2018-02-16 18.45.38')}
```

## 関連項目

- SQL の概念 : [データ型](#)、[日付/時刻文](#)
- SQL 関数 : [HOUR](#)、[MINUTE](#)、[CURRENT\\_TIME](#)、[CURTIME](#)、[NOW](#)、[DATEPART](#)、[DATENAME](#)
- ObjectScript 関数 : [\\$ZTIME](#)

- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

# SIGN (SQL)

与えられた数値式の正負符号を返す、数値関数です。

## 構文

```
SIGN(numeric-expression)  
{fn SIGN(numeric-expression)}
```

## 概要

SIGN は、以下の値を返します。

- numeric-expression がゼロより小さい場合、-1 を返します。
- numeric-expression がゼロの場合、0 (ゼロ) を返します。0、+0、または -0。
- numeric-expression がゼロより大きい場合、1 を返します。
- numeric-expression が NULL または非数値文字列の場合は、NULL です。

SIGN は、ODBC スカラ関数 ({} 括弧構文) または SQL 汎用関数のいずれかとして使用されます。

SIGN は、その値を決定する前に、numeric-expression を[キャノン形式](#)に変換します。例えば、SIGN(+++3) と SIGN(-3+5) はどちらも、正数を示す 1 を返します。

**注釈** InterSystems SQL では、2 つのマイナス記号 (ハイフン) はインラインの[コメント](#)文字を表します。そのため、2 つの連続するマイナス記号を指定する SIGN 引数は、引用符で囲まれた数値文字列として記述する必要があります。

## 引数

### numeric-expression

正負符号が返される数

## 例

以下の例は、SIGN の結果の例です。

### SQL

```
SELECT SIGN(-49) AS PosNeg
```

これは、-1 を返します。

### SQL

```
SELECT {fn SIGN(-0.0)} AS PosNeg
```

これは、0 を返します。

### SQL

```
SELECT SIGN(++-16.748) AS PosNeg
```

これは、1 を返します。

## SQL

```
SELECT {fn SIGN(NULL)} AS PosNeg
```

これは、<null> を返します。

## 関連項目

- ・ [+ \(正\)](#) および [- \(負\)](#) 単項演算子
- ・ [ABS](#) 関数
- ・ [ISNUMERIC](#) 関数
- ・ [%PLUS](#) および [%MINUS](#) 照合関数

# SIN (SQL)

与えられた角度の三角関数のサインを、ラジアン表示で返す、スカラ数値関数です。

## 構文

```
{fn SIN(numeric-expression)}
```

## 概要

SIN は、任意の数値を取り、浮動小数点数でサインを返します。NULL 値 を渡すと、SIN は NULL を返します。SIN は、非数値文字列を数値 0 (ゼロ) として扱います。

SIN は、有効桁数が 19 で小数桁数が 18 の値を返します。

SIN は、{ } 括弧構文による ODBC スカラ関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

## 引数

### *numeric-expression*

数値式。ラジアンで表示される角度です。

SIN は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。SIN は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、SIN の実行結果です。

### SQL

```
SELECT {fn SIN(0.52)} AS Sine
```

これは、0.496880 を返します。

## 関連項目

- SQL 関数: [ACOS](#)、[ASIN](#)、[ATAN](#)、[COS](#)、[COT](#)、[TAN](#)
- ObjectScript 関数: [\\$ZSIN](#)

## SPACE (SQL)

空白の文字列を返す文字列関数です。

### 構文

```
SPACE(count)

{fn SPACE(count)}
```

### 概要

SPACE は、空白スペースの文字列を count スペース分の長さだけ返します。count が数値文字列、10 進数、または混合数値文字列の場合、InterSystems IRIS は count をその整数部分に解決します。count が負の数または非数値文字列の場合は 0 に解決します。

文字列から空白スペースを削除するには、LTRIM (先頭の空白) または RTRIM (末尾の空白) を使用します。

**注釈** SPACE 関数を SPACE [照合タイプ](#) と混同しないでください。SPACE 照合は値の先頭にスペースを 1 つ追加します。それにより、値を文字列として解釈させます。SPACE 照合を指定するため、[CREATE TABLE](#) では %SPACE 照合キーワードが用意され、ObjectScript では %SYSTEM.Util クラスの Collation() メソッドが用意されています。

### 引数

#### count

返される空白スペースの数を指定する整数式。

### 例

以下の例は、name フィールドの空白の文字列の長さを返します。

#### SQL

```
SELECT SPACE(LENGTH(name))
FROM Sample.Person
```

### 関連項目

・ [LTRIMRTRIMTRIM](#)



## %SQLSTRING (SQL)

値を文字列としてソートする照合関数です。

### 構文

```
%SQLSTRING(expression[,maxlen])
```

```
%SQLSTRING expression
```

### 概要

%SQLSTRING は、expression を (大文字と小文字を区別する) 文字列としてソートされる形式に変換します。%SQLSTRING は、後続の空白 (スペースやタブなど) を削除し、文字列の最初の部分に空白を 1 つ追加します。この追加された空白は、NULL と数値を強制的に文字列として照合します。先頭と末尾のゼロは数値から削除されます。

%SQLSTRING がすべての値に空白を追加するため、NULL 値を文字列長 1 を持つ空白として照合します。%SQLSTRING は空白 (スペース、タブなど) だけを含む値を SQL 空文字列 (") として照合します。%SQLSTRING が空文字列 (長さがゼロ) に空白を追加すると、空文字列 \$CHAR(0) の内部表現が追加された空白として照合し、文字列長 2 という結果になります。

オプションの maxlen 引数は、インデックス指定や照合の際に expression 文字列を、指定された文字数に切り捨てます。例えば、maxlen トランケーションを持つ文字列を挿入する場合、完全な文字列が挿入され、SELECT 文で検索できます。この文字列に対するインデックス・グローバルは、指定された長さに切り捨てられます。これは、ORDER BY と比較演算は、切り捨てられたインデックス文字列のみを評価することを意味します。このようなトランケーションは、InterSystems IRIS 添え字で最大文字長を超える文字列に対してインデックスを指定するときに特に便利です。maxlen 引数では、長いフィールドにインデックスを指定する必要がある場合、トランケーション・レングス・パラメータを使用することができます。

%SQLSTRING は、expression の変換後に maxlen での切り捨てを実行します。変換された expression よりも maxlen の長さが長い場合は、パディングは追加されません。InterSystems IRIS では、文字列が文字列長の制限を超えることはできません。maxlen に対して明示的に最大値が強制されることはありませんが、該当する場合、InterSystems IRIS は <MAXSTRING> エラーを発行します。

%SYSTEM.Util クラスの Collation() メソッドを使用すると、ObjectScript で同じ照合変換を実行できます。

#### ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",8)
```

この関数は、ObjectScript から SQLSTRING() メソッド・コールを使用して呼び出すこともできます。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.SQLSTRING("The quick, BROWN fox.")
```

これらのメソッドの両方で、SQLSTRING 変換後の切り捨てがサポートされます。切り捨てられた長さには、追加された空白も含まれます。

#### ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",8,6),!  
WRITE $SYSTEM.SQL.SQLSTRING("The quick, BROWN fox.",6)
```

大文字/小文字を区別しない文字列変換に関しては、"%SQLUPPER" を参照してください。

注釈 システム全体の既定の照合を %SQLUPPER (大文字/小文字の区別なし) から %SQLSTRING (大文字/小文字の区別あり) に変更するには、以下のコマンドを使用します。

### ObjectScript

```
WRITE $$SetEnvironment^%apiOBJ("collation","%Library.String","SQLSTRING")
```

このコマンドを発行した後に、インデックスを削除し、すべてのクラスを再コンパイルしてから、再度インデックスを構築する必要があります。他のユーザがテーブルのデータにアクセスしている間はインデックスを再構築しないでください。再構築すると、クエリ結果が不正確になる可能性があります。

## 引数

### expression

列名、文字列リテラル、または他の関数の結果となる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。expression にはサブクエリを指定できます。

### maxlen

月における日付の値を返す元となる日付またはタイムスタンプ式 (オプション)。列の名前、他のスカラー関数の結果、または日付やタイムスタンプのリテラルである式。照合された値が、maxlen の値までに切り捨てられることを指定する正の整数。maxlen には、追加されている先頭の空白が含まれることに注意してください。((maxlen)) のように二重括弧で maxlen を囲むと、[リテラル置換を抑制](#)できます。

## 例

以下のクエリは、WHERE 節で %SQLSTRING を使用して、大文字と小文字を区別して選択を実行します。

### SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING Name %STARTSWITH %SQLSTRING 'Al'
ORDER BY Name
```

既定では、%STARTSWITH の文字列比較は大文字と小文字が区別されません。この例では、%SQLSTRING フォーマットを使用して、大文字と小文字を区別して比較を行います。これは “Al” で始まるすべての名前を返します (Allen、Alton など)。%STARTSWITH を使用する場合は、%SQLSTRING 照合を文の両側に適用する必要があります。

以下の例は、%SQLSTRING を文字列トランケーションと共に使用して、各名前の最初の 2 文字を返します。文字列トランケーションは、%SQLSTRING が先頭の空白を追加するため、3 (2 ではなく) になります。ORDER BY 節は、この 2 文字フィールドを使用して、行におおまかな照合順を配置します。

### SQL

```
SELECT Name, %SQLSTRING(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

この例は、切り捨てられた値を、大文字と小文字の区別を変更せずに返します。

以下の例では、%SQLSTRING をサブクエリに適用しています。

### SQL

```
SELECT TOP 5 Name, %SQLSTRING((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

## 関連項目

- ・ [CREATE TABLE](#)
- ・ [%STARTSWITH](#) 述語
- ・ [%SQLUPPER](#) 照合関数
- ・ [%TRUNCATE](#) 照合関数
- ・ [照合](#)

## %SQLUPPER (SQL)

値を大文字の文字列としてソートする照合関数です。

### 構文

```
%SQLUPPER(expression[,maxlen])
%SQLUPPER expression
```

### 概要

既定の照合は SQLUPPER です。

%SQLUPPER は、expression を (大文字と小文字を区別しない) 大文字の文字列としてソートされる形式に変換します。%SQLUPPER は、すべてのアルファベット文字を大文字に変換し、後続の空白 (スペースやタブなど) を削除し、文字列の最初の部分に空白を 1 つ追加します。この追加された空白は、NULL と数値を文字列として照合します。

SQL では、数値を関数に渡す前に、先頭と末尾にあるゼロを削除したり、指数を展開したりすることにより、[キャノニック形式](#)に変換します。数値文字列はキャノニック形式に変換されません。

%SQLUPPER はすべての値の先頭に空白を追加するため、NULL 値を文字列長 1 を持つ空白として照合します。%SQLUPPER は、空白 (スペース、タブなど) だけを含む任意の値を SQL [空文字列](#) (') として照合します。%SQLUPPER が空文字列 (長さがゼロ) の前に空白を追加すると、空文字列 \$CHAR(0) の内部表現が追加された空白として照合し、文字列長 2 という結果になります。

オプションの maxlen 引数は、インデックス指定や照合の際に変換された expression 文字列を、指定された文字数に切り捨てます。例えば、maxlen トランケーションを持つ文字列を挿入する場合、完全な文字列が挿入され、SELECT 文で検索できます。この文字列に対するインデックス・グローバルは、指定された長さに切り捨てられます。これは、ORDER BY と比較演算は、切り捨てられたインデックス文字列のみを評価することを意味します。このようなトランケーションは、InterSystems IRIS 添え字で[最大文字長](#)を超える文字列に対してインデックスを指定するときに特に便利です。maxlen 引数では、長いフィールドにインデックスを指定する必要がある場合、トランケーション・レンジス・パラメータを使用することができます。

%SQLUPPER は、expression の変換後に maxlen での切り捨てを実行します。変換された expression よりも maxlen の長さが長い場合は、パディングは追加されません。InterSystems IRIS では、文字列が[文字列長の制限](#)を超えることはできません。maxlen に対して明示的に最大値が強制されることはありませんが、該当する場合、InterSystems IRIS は <MAXSTRING> エラーを発行します。

%SYSTEM.Util クラスの Collation() メソッドを使用すると、ObjectScript で同じ照合変換を実行できます。

#### ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",7)
```

この関数は、ObjectScript から SQLUPPER() メソッド・コールを使用して呼び出すこともできます。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.SQLUPPER("The quick, BROWN fox.")
```

これらのメソッドの両方で、SQLUPPER 変換後の切り捨てがサポートされます。切り捨てられた長さには、先頭に追加された空白も含まれます。

#### ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",7,6),!
WRITE $SYSTEM.SQL.SQLUPPER("The quick, BROWN fox.",6)
```

大文字/小文字を区別する文字列変換に関しては、"[%SQLSTRING](#)" を参照してください。

注釈 システム全体の既定の照合を %SQLUPPER (大文字/小文字の区別なし) から %SQLSTRING (大文字/小文字の区別あり) に変更するには、以下のコマンドを使用します。

### ObjectScript

```
WRITE $$SetEnvironment^%apiOBJ("collation", "%Library.String", "SQLSTRING")
```

このコマンドを発行した後に、インデックスを削除し、すべてのクラスを再コンパイルしてから、再度インデックスを構築する必要があります。他のユーザがテーブルのデータにアクセスしている間はインデックスを再構築しないでください。再構築すると、クエリ結果が不正確になる可能性があります。

## その他の大文字/小文字変換関数

%SQLUPPER 関数は、大文字と小文字を区別しない比較または照合に対してデータ値を変換する、SQL で優先的な方法です。%SQLUPPER は、データの最初の部分に空白を追加します。それによって、数値データと NULL 値を文字列として解釈させます。

以下は、データ値を大文字/小文字変換するその他の関数です。

- **UPPER** および **UCASE**: 文字を大文字に変換し、数字文字、句読点、埋め込みの空白、および先頭と末尾の空白に影響を与えません。数値を文字列として解釈する変換を強制的に実行しません。
- **LOWER** および **LCASE**: 文字を小文字に変換し、数字文字、句読点、埋め込みおよび先頭と末尾の空白に影響を与えません。数値を文字列として解釈する変換を強制的に実行しません。
- **%SQLSTRING**: 文字の大文字小文字を変換しません。ただし、データの最初の部分に空白を追加します。それによって、数値データと NULL 値を文字列として解釈させます。

## 英数字の照合順序

大小文字変換関数は、以下のような異なるアルゴリズムを使用して数字で始まるデータ値を照合します。

%MVR	%SQLUPPER、%SQLSTRING、およびその他の大小文字変換関数
6 Oak Avenue, 66 Main Street, 66 Oak Street, 641 First Place, 665 Ash Drive, 672 Main Court, 709 Oak Avenue, 5988 Clinton Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 6572 First Avenue, 6643 First Street, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court,	5988 Clinton Avenue, 6 Oak Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 66 Main Street, 66 Oak Street, 6643 First Street, 665 Ash Drive, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court, 709 Oak Avenue,

## 引数

### expression

列名、文字列リテラル、または他の関数の結果となる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。expression にはサブクエリを指定できます。

## maxlen

照合された値が、maxlen の値までに切り捨てられることを指定する整数 (オプション)。maxlen には、追加されている先頭の空白が含まれることに注意してください。((maxlen)) のように二重括弧で maxlen を囲むと、[リテラル置換を抑制](#)できます。

## 例

以下のクエリは、文字列トランケーションを持つ %SQLUPPER を使用して、大文字の各名前の最初の 2 文字を返します。文字列トランケーションは、%SQLUPPER が先頭の空白を追加するため、3 (2 ではなく) になります。ORDER BY 節は、この 2 文字フィールドを使用して、行におおまかな照合順を配置します。

### SQL

```
SELECT Name, %SQLUPPER(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

以下の例では、%SQLUPPER をサブクエリに適用しています。

### SQL

```
SELECT TOP 5 Name, %SQLUPPER((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

## 関連項目

- ・ [CREATE TABLE](#)
- ・ [%STARTSWITH](#) 述語
- ・ [%SQLSTRING](#) 照合関数
- ・ [%TRUNCATE](#) 照合関数
- ・ [照合](#)

# SQRT (SQL)

与えられた数値式の平方根を返す、数値関数です。

## 構文

```
SQRT(numeric-expression)
{fn SQRT(numeric-expression)}
```

## 概要

SQRT は、numeric-expression の平方根を返します。numeric-expression は正の数である必要があります。負の numeric-expression (-0 以外) の場合、SQLCODE -400 エラーが生成されます。NULL 値を渡すと、SQRT は NULL を返します。

SQRT は、有効桁数が 36 で小数桁数が 18 の値を返します。

SQRT は、正規スカラ関数または {} 括弧構文を使用した ODBC スカラ関数として指定できます。

## 引数

### numeric-expression

平方根を計算する対象となる正の数に解決される式です。

SQRT は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。SQRT は、numeric-expression がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、2 つの SQRT 構文形式を示しています。どちらも 49 の平方根を返します。

### SQL

```
SELECT SQRT(49) AS SRoot, {fn SQRT(49)} AS ODBCRoot
```

以下の埋め込み SQL の例は、0 から 10 までの整数の平方根を返します。

### ObjectScript

```
SET a=0
WHILE a<11 {
&sql(SELECT SQRT(:a) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
ELSE {
    WRITE !,"The square root of ",a," = ",b
    SET a=a+1 }
}
```

## 関連項目

- SQL 関数: [POWER ROUND SQUARE](#)
- ObjectScript 関数: [\\$ZSQR](#)

## SQUARE (SQL)

数値の二乗を返すスカラ数値関数です。

### 構文

```
SQUARE(numeric-expression)
```

### 説明

SQUARE は、*numeric-expression* の二乗を返します。NULL 値を渡すと、SQUARE は NULL を返します。

SQUARE によって返される精度とスケールは、[SQL 乗算演算子](#)が返す、精度およびスケールと同じです。

### 引数

#### *numeric-expression*

数値に解決される式です。

SQUARE は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。SQUARE は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

### 例

以下の埋め込み SQL の例は、0 から 10 までの整数の二乗を返します。

#### ObjectScript

```
SET a=0
WHILE a<11 {
&sql(SELECT SQUARE(:a) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
ELSE {
    WRITE !,"The square of ",a," = ",b
    SET a=a+1 }
}
```

### 関連項目

- SQL 関数: [POWER](#)、[ROUND](#)、[SQRT](#)
- ObjectScript 関数: [\\$ZPOWER](#)



# STR (SQL)

数値を文字列に変換する関数。

## 構文

```
STR(number[, length[, decimals]])
```

## 概要

STR は数値を STRING 形式に変換し、length および decimals の値に基づいて数値を切り捨てます。length 引数は、数値の整数部分がすべて入るのに十分な大きさをとる必要があります。また、decimals を指定する場合、その数字は小数桁数に 1 を加えた数にします (小数点があるため)。length の大きさが十分でない場合、STR は length と同じ長さのアスタリスク (\*) の文字列を返します。

STR は、文字列変換する前に数値を**キャノニック形式**に変換します。そのため、算術演算が実行され、数値の先頭および末尾のゼロと先頭のプラス記号が取り除かれます。

number 引数が NULL の場合、STR は NULL を返します。number 引数が空文字列 (') の場合、STR は空文字列を返します。STRING は、空白を保持します。

## 引数

### number

数値として評価する式。これは、フィールド名、数値、または別の関数の結果を指定できます。フィールド名が指定されている場合は、論理値が使用されます。

### length

すべての文字 (数字、小数点、符号、空白スペース) を含む目的の出力文字列の合計の長さを指定する整数 (オプション)。既定は 10 です。

### decimals

小数点の右側に入る桁数を指定する整数 (オプション)。既定は 0 です。

## 例

以下の例では、STR により数値を文字列に変換しています。

### SQL

```
SELECT STR(123),  
       STR(123,4),  
       STR(+00123.45,3),  
       STR(+00123.45,3,1),  
       STR(+00123.45,5,1)
```

最初の STR 関数は先頭の 7 個の空白と数字 123 で構成される文字列を返します。先頭に 7 つの空白が入るのは、既定の文字列長が 10 であるためです。2 番目の STR 関数は文字列 “123” を返します。長さ 4 の文字列を返すために先頭の空白が必要になります。3 番目の STR 関数は文字列 “123” を返します。数値はキャノニック形式で示し、decimals は既定の 0 になります。4 番目の STR 関数は、“\*\*\*” を返します。これは文字列長が、指定した数全体を収容しきれない大きさであるためです。アスタリスクの数は文字列の長さを示します。5 番目の STR 関数は、“123.4” を返します。length は、小数桁を含めるために 5 にする必要があります。

## 関連項目

・ [STRING](#)、[%SQLUPPER](#)、[%SQLSTRING](#)

## STRING (SQL)

式を文字列に変換して連結する関数。

### 構文

```
STRING(string1[,string2][,...][,stringN])
```

### 説明

STRING は、1 つ以上の string を STRING 形式に変換し、これらの文字列を単一の文字列に連結します。ケース変換は実行されません。

STRING は、文字列変換する前に数値を**キャノニック形式**に変換します。そのため、算術演算が実行され、数値の先頭および末尾のゼロと先頭のプラス記号が取り除かれます。

string 引数のいずれかが NULL または**空の文字列** (') の場合、STRING はその他すべての引数を連結し、NULL と空の文字列をその連結から削除します。すべての string 引数が NULL の場合、STRING は NULL を返します。すべての string 引数が空文字列 (') の場合、STRING は空文字列を返します。STRING は、空白を保持します。

**%SQLSTRING** 関数を使用すると、データを文字列の大文字と小文字を区別して比較するための値に変換します。また、**%SQLUPPER** 関数は、データを文字列の大文字と小文字を区別せずに比較するための値に変換します。

### 引数

#### string

フィールド名、文字列リテラル、数値、または他の関数の結果となる式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。フィールド名が指定されている場合は、論理値が使用されます。

### 例

以下の例では、STRING により3つの部分文字列を単一の文字列に連結しています。また、空白スペース、空文字列、および NULL の処理についても示しています。

#### SQL

```
SELECT STRING('a','b','c'),  
       STRING('a',' ','c'),  
       STRING('a','','c'),  
       STRING('a',NULL,'c')
```

以下の例では、STRING により数値を文字列に変換しています。これらすべての STRING 関数は、文字列 '123' を返します。

#### SQL

```
SELECT STRING(123),  
       STRING(+00123.00),  
       STRING('1',23),  
       STRING(1,(10*2)+3)
```

以下の例では、STRING を使用してフィールドからサンプル・データを取得し、これを文字列に変換しています。

#### SQL

```
SELECT STRING(Name,Age)  
FROM Sample.Person
```

## 関連項目

- ・ [%SQLUPPER](#)、[%SQLSTRING](#)、[STR](#)

## STUFF (SQL)

文字列内の部分文字列を置換する文字列関数です。

### 構文

```
STUFF(string, start, length, substring)
```

### 概要

STUFF は、部分文字列を別の部分文字列と置き換えます。位置と長さにより置き換える部分文字列を特定し、それを substring と置き換えます。

この関数には Transact-SQL 実装との互換性があります。

置換 substring は、元の値よりも長い場合もあります。元の値を削除するために、substring を空文字列 (") にすることができます。

start の値は、string の現在の長さ以内である必要があります。0 の start を指定すると、string の先頭に substring を追加できます。空の文字列または数値以外の値は 0 として処理されます。

start、length、または substring 引数に NULL を指定すると、NULL が返されます。

STUFF では string 引数または substring 引数に [%Stream.GlobalCharacter field](#) を使用することはできません。これを実行しようとすると、SQLCODE -37 エラーが生成されます。

### REPLACE と STUFF

REPLACE および STUFF の両方とも、部分文字列の置換を実行します。REPLACE は、データ値によって部分文字列を検索します。STUFF は、文字列の位置と長さによって部分文字列を検索します。

部分文字列を検索する関数のリストは、[“文字列操作”](#) を参照してください。

### 引数

#### string

部分文字列置換の置換対象となる文字列式。

#### start

正の整数として指定される置換の開始地点。string の最初から文字数がカウントされ、カウントは 1 を基準とします。許可される値は 0 ～ string の長さまでです。文字を追加するには、0 の start および 0 の length を指定します。空の文字列または数値以外の値は 0 として処理されます。

#### length

正の整数として指定される、置換する文字数。文字を挿入するには、0 の length を指定します。start の後のすべての文字を置き換えるには、既存の文字数より大きい length を指定します。空の文字列または数値以外の値は 0 として処理されます。

#### substring

開始地点と長さで識別される部分文字列を置き換えるのに使用する文字列式。置き換える部分文字列よりも長くするか短くすることができます。空の文字列でもかまいません。

### 例

以下の例は、BOLT を BELT に変える、単一文字の置換を示しています。

## SQL

```
SELECT STUFF('BOLT',2,1,'E')
```

以下の例は、8 文字の部分文字列 (Kentucky) をこれより長い 12 文字の部分文字列、およびこれより短い 2 文字の部分文字列で置き換えます。

## SQL

```
SELECT STUFF('In my old Kentucky home',11,8,'Rhode Island'),  
       STUFF('In my old Kentucky home',11,8,'KY')
```

以下の例は、部分文字列を挿入します。

## SQL

```
SELECT STUFF('In my old Kentucky home',19,0,' (KY)')
```

以下の例は、部分文字列を文字列の最初に追加します。

## SQL

```
SELECT STUFF('In my old Kentucky home',0,0,'The sun shines bright ')
```

以下の例は、空の文字列で置き換えることにより、8 文字の部分文字列を削除します。

## SQL

```
SELECT STUFF('In my old Kentucky home',11,8,'')
```

## 関連項目

- ・ [REPLACE 関数](#)
- ・ [\\$EXTRACT 関数](#)
- ・ [SUBSTRING 関数](#)
- ・ [SUBSTR 関数](#)
- ・ [文字列操作](#)

## SUBSTR (SQL)

指定された文字列式から得られた部分文字列を返す文字列関数です。

### 構文

```
SUBSTR(string-expression,start[,length])
```

### 引数

引数	説明
<i>string-expression</i>	部分文字列が得られる文字式。式は列の名前や文字列リテラル、または他のスカラ関数の結果を指定できます。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。
<i>start</i>	<i>string-expression</i> 内で、部分文字列が開始される場所を指定する整数。正の数の開始位置は、文字列の最初からの文字数を指定します。 <i>string-expression</i> 1 の最初の文字はポジション 1 にあります。負の数の開始位置は、文字列の最後からの文字数を指定します。 <i>start</i> が 0 (ゼロ) の場合は 1 として扱います。
<i>length</i>	オプション - 返される部分文字列の長さを指定する正の整数。この値は、部分文字列が開始位置から右に <i>length</i> の文字数を進んで終了することを指定します。省略された場合、部分文字列は <i>start</i> から <i>string-expression</i> の最後まで進みます。 <i>length</i> が 0 または負の数の場合、InterSystems IRIS は NULL を返します。

### 概要

*start* は負の数になる場合もあるので、元の文字列の先頭または末尾のいずれからでも部分文字列を取得できます。

SUBSTR に引数として渡される浮動小数点は、小数部を切り捨てることにより、整数に変換されます。

- *start* が 0、-0、または 1 の場合、返される部分文字列は、その文字列の最初の文字で始まります。
- *start* が負の数である場合、返される部分文字列の先頭が、その文字列の末尾からこの文字数分の位置になります。-1 は文字列の最終文字を表します。負の数が大きく、文字列の末尾から逆向きに数えた値が文字列の先頭よりも前の位置になる場合、返される部分文字列の先頭は、文字列の最初の文字になります。
- *start* が文字列の範囲を超えていると、NULL が返されます。
- *length* が文字列内の残りすべての文字より長い場合、*start* から文字列の最後までの部分文字列が返されます。
- *length* が 1 未満の場合、NULL が返されます。
- *start* または *length* のいずれかが NULL の場合、NULL が返されます。

SUBSTR をストリーム・データと共に使用することはできません。*string-expression* がストリーム・フィールドの場合、SUBSTR は SQLCODE -37 を発行します。SUBSTRING を使用して、ストリーム・データから部分文字列を抽出します。

SUBSTR は、Oracle に対する互換性がサポートされています。

### 例

以下の例は、3 番目の文字 (C) から文字列の最後までを指定するので、部分文字列 CDEFG を返します。

## SQL

```
SELECT SUBSTR('ABCDEFGF',3) AS Sub
```

以下の例は、3 番目の文字 (C) から 4 文字分 (F まで) の部分文字列を指定するので、部分文字列 CDEF を返します。

## SQL

```
SELECT SUBSTR('ABCDEFGF',3,4) AS Sub
```

以下の例は、InterSystems IRIS が、まず元の文字列の末尾から逆方向に 5 文字をカウントし、そこから 4 文字分を返すように指定しているので、CDEF を返します。

## SQL

```
SELECT SUBSTR('ABCDEFGF',-5,4) AS Sub
```

## 関連項目

- ・ SQL 関数 : [SUBSTRING](#)
- ・ ObjectScript 関数: [\\$EXTRACT \\$PIECE](#)

## SUBSTRING (SQL)

ストリーム・データを含め、任意のデータ型のデータから部分文字列を返す文字列関数です。

### 構文

```
SUBSTRING(string-expression,start[,length])
SUBSTRING(string-expression FROM start [FOR length])

{fn SUBSTRING(string-expression,start[,length])}
```

### 引数

引数	説明
<i>string-expression</i>	部分文字列が得られる文字式。列の名前、文字列リテラル、他のスカラ関数の結果などを表すことができる式。このフィールドは、文字列 (CHAR や VARCHAR など)、数値、またはデータ型 %Stream.GlobalCharacter または %Stream.GlobalBinary のデータ・ストリーム・フィールドなど、任意のデータ型にすることができます。
<i>start</i>	<i>string-expression</i> 内で、部分文字列が開始する場所を指定する整数。 <i>string-expression</i> の最初の文字はポジション 1 です。開始位置が文字列の長さ以上である場合、SUBSTRING は空文字列 (") を返します。開始位置が 1 以下である場合 (ゼロまたは負の数)、部分文字列はポジション 1 で開始しますが、部分文字列の <i>length</i> は開始位置により削減されます。
<i>length</i>	オプション - 返される部分文字列の長さを指定する整数。 <i>length</i> を指定しない場合、既定で残りの文字列が返されます。

### 概要

SUBSTRING は任意のデータ型のデータを取り、そのデータの部分文字列をデータ型 %String として返します。部分文字列はもちろん、文字列として返される完全なデータ値でもかまいません。

*start* の値は、部分文字列の開始ポイントを管理します。

- ・ *start* が 1 の場合、部分文字列は *string-expression* の先頭から始まります。
- ・ *start* が 1 より大きい場合、部分文字列は *string-expression* の先頭からカウントされるその文字位置から始まります。
- ・ *start* が 1 より小さい場合、部分文字列は *string-expression* の先頭から始まりますが、*length* の値は対応する量によってディクリメントされます。したがって、*start* が 0 の場合、*length* の値は 1 まで減少します。*start* が -1 の場合、*length* の値は 2 まで減少します。

*length* の値は、部分文字列のサイズを管理します。

- ・ *length* が正の値の場合 (1 以上)、部分文字列は *start* 位置から右に向かって *length* 文字数まで進んで終わります (この有効な長さは、*start* の数が 1 より小さいと減少する場合があります)。
- ・ *length* が、文字列内の残りの文字より長い場合、*string-expression* の開始位置の右側から最後まですべての文字を返します。
- ・ *length* が 0 未満の場合、NULL が返されます。
- ・ *length* が負の数の場合、InterSystems IRIS は SQLCODE -140 エラーを発行します。

SUBSTRING は、ODBC スカラ関数 ({ } 括弧構文) や SQL 汎用関数として使用されます。



## 返り値

string-expression が %String データ型の場合は、SUBSTRING の返り値は、string-expression データ型と同じデータ型になります。これにより、SUBSTRING でユーザ定義の文字列データ型を、特殊なエンコードを使用して処理できるようになります。

string-expression が %String データ型でない場合 (%Stream.GlobalCharacter など)、SUBSTRING の返り値は %String になります。

すべての SUBSTRING 引数の値が NULL の場合、SUBSTRING は NULL を返します。

## ストリーム・データ

ほとんどの SQL 文字列関数とは異なり、SUBSTRING はストリーム・データと共に使用できます。string-expression は、データ型 %Stream.GlobalCharacter または %Stream.GlobalBinary のフィールドにできます。SUBSTRING は、抽出されたストリーム・データのサブセットを %String データ型として返します。start が 1 の場合に length を省略すると、SUBSTRING は完全なストリーム・データ値を %String として返します。

このため、SUBSTRING を使用して、文字ストリーム・データを文字列として他の SQL 文字列関数に提供できます。以下の例では、SUBSTRING を使用して、DNA ヌクレオチド配列が含まれる %Stream.GlobalCharacter フィールドの最初の 1000 文字で部分文字列 TTAGGG の最初の出現箇所を CHARINDEX で検索し、その位置を整数として返すことができます。

### SQL

```
SELECT CHARINDEX('TTAGGG',SUBSTRING(DNASeq,1,1000)) FROM Sample.DNASequences
```

## SUBSTRING と SUBSTR

- ・ SUBSTRING は、string-expression の先頭からカウントされる start 位置から部分文字列を抽出します。SUBSTR は、部分文字列を文字列の最初の部分または最後の部分のいずれかから抽出できます。
- ・ SUBSTRING はストリーム・データと共に使用できますが、SUBSTR はストリーム・データと共に使用できません。

## 例

以下の例は、“forward” 文字列を返します。

### SQL

```
SELECT {fn SUBSTRING( 'forward pass',1,7 )} AS SubText
```

以下の例は、“pass” 文字列を返します。

### SQL

```
SELECT {fn SUBSTRING( 'forward pass',9,4 )} AS SubText
```

以下の例は、各名前の最初の 4 文字を返します。

### SQL

```
SELECT Name,SUBSTRING(Name,1,4) AS FirstFour
FROM Sample.Person
```

以下の例は、SUBSTRING の他の構文形式を示しています。この例は、機能的に以前の例と同じです。

### SQL

```
SELECT Name,SUBSTRING(Name FROM 1 FOR 4) AS FirstFour
FROM Sample.Person
```

以下の例では、length が 1 より小さい start の値によってどのように減少するかを示しています (0 の start 値は length を 1 減らし、-1 の start 値は length を 2 減らす、というようになります)。この場合、length は 3 減らされるため、1 つの文字 (“A”) だけが返されます。

### SQL

```
SELECT {fn SUBSTRING( 'ABCDEFG',-2,4 )} AS SubText
```

### 関連項目

- ・ SQL 関数 : [SUBSTR](#)
- ・ ObjectScript 関数 : [\\$EXTRACT](#)、[\\$PIECE](#)

# SYSDATE (SQL)

現在のローカルな日付と時刻を返す日付/時刻関数です。

## 構文

SYSDATE

## [説明]

SYSDATE は引数を取らず、%TimeStamp データ型形式 (yyyy-mm-dd hh:mm:ss.fff) または %PosixTime データ型形式 (エンコードされた 64 ビットの符号付き整数) のいずれかの [タイムスタンプ](#) として現在のローカル日付とローカル時刻を返します。SYSDATE は、この [タイムゾーン](#) の現在のローカル日付とローカル時刻を返します。これは [サマータイム](#) などのローカル時刻調整に合わせて調整されます。

既定では、SYSDATE は時刻を整数秒単位で返します。この既定値は構成可能です。

注釈 SYSDATE は、引数なしの [CURRENT\\_TIMESTAMP](#) 関数と同義です。InterSystems SQL では、CURRENT\_TIMESTAMP の使用を推奨します。SYSDATE 関数は他のバージョンの SQL との互換性のために提供されています。

## 関連項目

- ・ [CURRENT\\_TIMESTAMP](#)

## %SYSTEM\_SQL.DefaultSchema()

---

現在のネームスペースで現在のプロセスの既定のスキーマを返す関数。

### 構文

```
%SYSTEM_SQL.DefaultSchema()
```

### 説明

%SYSTEM\_SQL.DefaultSchema() は、引数を取りません。現在のネームスペースで現在のプロセスの既定のスキーマを返します。引数の括弧は必須です。

### 関連項目

- ・ [クラス・リファレンス](#)

# TAN (SQL)

与えられた角度の三角関数のタンジェントを、ラジアン表示で返すスカラー数値関数です。

## 構文

```
{fn TAN(numeric-expression)}
```

## 概要

TAN は、任意の数値を取り、タンジェントを返します。NULL 値を渡すと、TAN は NULL を返します。TAN は、非数値文字列を数値 0 (ゼロ) として扱います。

TAN は、有効桁数が 36 で小数桁数が 18 の値を返します。

TAN は {} 括弧構文による ODBC スカラー関数としてのみ使用できます。

[DEGREES](#) 関数を使用してラジアンを度数に変換できます。[RADIANS](#) 関数を使用して度数をラジアンに変換できます。

## 引数

### *numeric-expression*

数値式。ラジアンで表示される角度です。

TAN は、NUMERIC または DOUBLE [データ型](#)のいずれかを返します。TAN は、*numeric-expression* がデータ型 DOUBLE の場合には DOUBLE を返し、それ以外の場合には NUMERIC を返します。

## 例

以下の例は、TAN の実行結果です。

### SQL

```
SELECT {fn TAN(0.52)} AS Tangent
```

これは、0.572561 を返します。

## 関連項目

- SQL 関数: [ACOS](#)、[ASIN](#)、[ATAN](#)、[COS](#)、[COT](#)、[SIN](#)
- ObjectScript 関数: [\\$ZTAN](#)

## TIMESTAMPADD (SQL)

指定された日付部分の間隔の数をタイムスタンプに加えて計算された新しいタイムスタンプを返すスカラ日付/時刻関数です。

### 構文

```
{fn TIMESTAMPADD(interval-type, integer-exp, timestamp-exp)}
```

### 引数

引数	説明
<i>interval-type</i>	キーワードとして指定される、 <i>integer-exp</i> が表す時刻/日付間隔のタイプ。
<i>integer-exp</i>	<i>timestamp-exp</i> に追加される整数値式。
<i>timestamp-exp</i>	タイムスタンプ値式。 <i>integer-exp</i> の値によって増加します。

### 概要

TIMESTAMPADD 関数は、指定された日付部分を指定された単位数でインクリメントすることで、日付/時刻式を変更します。例えば、*interval-type* が SQL\_TSI\_MONTH で、*integer-exp* が 5 の場合は、TIMESTAMPADD は *timestamp-exp* に 5 か月をインクリメントします。また、*integer-exp* に負の整数を指定して、日付部分をデクリメントすることもできます。

TIMESTAMPADD は、入力 *timestamp-exp* と同じデータ型のタイムスタンプを返します。このタイムスタンプは、**%Library.TimeStamp** データ型形式 (yyyy-mm-dd hh:mm:ss.fff) または **%Library.PosixTime** データ型形式 (エンコードされた 64 ビットの符号付き整数) のいずれかにできます。

TIMESTAMPADD は {} 括弧構文による ODBC スカラ関数としてのみ使用できる点に注意してください。

同様の日付/時刻変更の演算は、[DATEADD](#) 汎用関数を使用して、タイムスタンプで実行できます。

### 間隔のタイプ

*interval-type* 引数は、以下のタイムスタンプ間隔のいずれかになります。

- SQL\_TSI\_FRAC\_SECOND
- SQL\_TSI\_SECOND
- SQL\_TSI\_MINUTE
- SQL\_TSI\_HOUR
- SQL\_TSI\_DAY
- SQL\_TSI\_WEEK
- SQL\_TSI\_MONTH
- SQL\_TSI\_QUARTER
- SQL\_TSI\_YEAR

これらのタイムスタンプの間隔は、引用符なしで、または一重引用符や二重引用符で囲んで指定できます、大文字と小文字は区別されません。

タイムスタンプ間隔をインクリメントまたはデクリメントすると、他の間隔も適切に変更されます。例えば、時間のインクリメント結果が午前 0 時を超える場合、自動的に日がインクリメントされます。同様に、順次、月などがインクリメントされます。TIMESTAMPADD は、各月の日数やうるう年を考慮に入れて、常に有効な日付を返します。例えば、1 月 31 日に 1 か

月をインクリメントすると 2 月 28 日 (該当月の有効な最終日付) が返されますが、指定された年がうるう年の場合は 2 月 29 日が返されます。

3 桁の精度の秒の小数部でインクリメントあるいはデクリメントできます。1000 分の 1 秒単位の整数として秒の小数部を指定します (001 から 999)。

DATEADD と TIMESTAMPADD では四半期 (3 か月の間隔) が処理されます。DATEDIFF と TIMESTAMPDIFF では四半期は処理されません。

## %TimeStamp 形式

timestamp-exp 引数が **%Library.TimeStamp** データ型形式 (yyyy-mm-dd hh:mm:ss.ffff) である場合、以下のルールが適用されます。

- timestamp-exp で時刻値のみが指定されている場合、結果のタイムスタンプの計算前に、timestamp-exp の日付部分は '1900-01-01' に設定されます。
- timestamp-exp が日付値のみを指定する場合、timestamp-exp の時刻部分は結果のタイムスタンプが計算される前は '00:00:00' に設定されます。
- timestamp-exp に秒の小数部を含めても省略してもかまいません。timestamp-exp には任意の桁数の精度を含めることができますが、interval-type SQL\_TSI\_FRAC\_SECOND では正確に 3 桁の精度が指定されます。3 桁以外の SQL\_TSI\_FRAC\_SECOND を指定しようとすると、予期しない結果が発生する可能性があります。

## 範囲と値のチェック

TIMESTAMPADD は、**%Library.TimeStamp** 入力値に対して以下のチェックを実行します。

- TIMESTAMPADD 処理を実行するには、timestamp-exp の指定された部分がすべて有効である必要があります。
- 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。年は 4 桁で指定される必要があります。無効な日付値を指定すると、SQLCODE -400 エラーになります。
- 日付値は、有効な範囲内にある必要があります。年は 0001 から 9999、月は 1 から 12、日は 1 から 31、時間は 00 から 23、分は 0 から 59、秒は 0 から 59 がそれぞれ有効範囲です。月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。無効な日付値を指定すると、SQLCODE -400 エラーになります。
- インクリメント (またはデクリメント) 結果として返される年は、0001 から 9999 までの範囲内にある必要があります。この範囲を超えると、<null> が返されます。
- 10 よりも小さい日付値の先頭のゼロは、記載、省略のどちらでもかまいません。その他の非標準的な整数値は許可されません。例えば、'07' または '7' は有効な日付値ですが、'007'、'7.0'、または '7a' は無効です。10 よりも小さい日付値は、常に先頭の 0 がついた状態で返されます。
- 時刻値は、全体または一部を省略できます。timestamp-exp で不完全な時刻が指定されている場合、指定されていない部分に対して 0 が指定されます。
- 10 よりも小さい時間値の先頭の 0 は記載する必要があります。この先頭の 0 を省略すると、SQLCODE -400 エラーになります。

## 例

以下の例は、元のタイムスタンプに 1 週間を加算します。

### SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_WEEK,1,'2017-12-20 12:00:00')}
```

1 週を加算は 7 日の加算になるので、2017-12-27 12:00:00 が返されます。

以下の例は、元のタイムスタンプに 5 か月を加算します。

#### SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MONTH,5,'2017-12-20 12:00:00')}
```

この例は 5 か月を加算すると年がインクリメントされるので、2018/05/20 12:00:00 を返します。

以下の例も、元のタイムスタンプに 5 か月を加算します。

#### SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MONTH,5,'2018-01-31 12:00:00')}
```

上記の例は 2018/06/30 12:00:00 を返します。この TIMESTAMPADD は、月だけでなく日の値も変更します。これは単純に月をインクリメントすると、無効な日付の 6 月 31 日となるためです。

以下の例は、元のタイムスタンプに 45 分をインクリメントします。

#### SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MINUTE,45,'2017-12-20 00:00:00')}
```

上記の例は 2017/12/20 00:45:00 を返します。

以下の例は、元のタイムスタンプに 45 分をデクリメントします。

#### SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MINUTE,-45,'2017-12-20 00:00:00')}
```

上記の例は 2017/12/19 23:15:00 を返します。この場合は、時刻をデクリメントすると、日もデクリメントされます。

## 関連項目

- ・ [TIMESTAMPDIFF](#)、[DATEADD](#)、[DATENAME](#)、[DATEPART](#)、[TO\\_POSIXTIME](#)、[TO\\_TIMESTAMP](#)



# TIMESTAMPDIFF (SQL)

指定された日付部分における 2 つのタイムスタンプ間の差異を示す整数値を返すスカラ日付/時刻関数です。

## 構文

```
{fn TIMESTAMPDIFF(interval,startDate,endDate)}
```

## 概要

- ・ {fn TIMESTAMPDIFF(*interval*,*startDate*,*endDate*)} は、指定された日付部分の間隔 (秒、日、週など) における開始タイムスタンプと終了タイムスタンプとの差異 (*startDate* - *endDate*) を返します。この関数は、2 つのタイムスタンプ間の間隔数を表す INTEGER 値を返します。*endDate* が *startDate* より前の場合、TIMESTAMPDIFF は負の INTEGER 値を返します。

以下の文は、後者のタイムスタンプが、前者のタイムスタンプよりも 12 日大きいので、12 を返します。どちらのタイムスタンプも既定時刻は 00:00:00 です。

### SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_DAY,'2022-4-1','2022-4-13')}
```

例：タイムスタンプ間の差異の計算

## 引数

### interval

返されるタイムスタンプの差異が表す時刻間隔または日付間隔のタイプ。以下のいずれかのタイムスタンプ間隔として指定します。

- ・ SQL\_TSI\_FRAC\_SECOND - 秒の小数部で表した間隔
- ・ SQL\_TSI\_SECOND - 秒で表した間隔
- ・ SQL\_TSI\_MINUTE - 分で表した間隔
- ・ SQL\_TSI\_HOUR - 時間で表した間隔
- ・ SQL\_TSI\_DAY - 日で表した間隔
- ・ SQL\_TSI\_WEEK - 週で表した間隔
- ・ SQL\_TSI\_MONTH - 月で表した間隔
- ・ SQL\_TSI\_YEAR - 年で表した間隔

### startDate、endDate

比較する開始日と終了日を表すタイムスタンプ値の式。以下のいずれかの値として指定します。

- ・ %Library.TimeStamp データ型形式 (yyyy-mm-dd hh:mm:ss.fff)
- ・ %Library.PosixTime データ型形式 (エンコードされた 64 ビットの符号付き整数)

これらのタイムスタンプ間隔は、引用符なしで、または一重引用符か二重引用符で囲んで指定できます。大文字と小文字は区別されません。

startDate または endDate のいずれかで %Library.TimeStamp 形式を使用する場合、以下のルールが適用されます。

- 一方のタイムスタンプ式で時刻値のみが指定されており、interval で日付間隔（日、週、月、または年）が指定されている場合、結果の間隔数の計算前に、タイムスタンプの欠けている日付部分は、既定で '1900-01-01' になります。
- 一方のタイムスタンプ式で日付値のみが指定されており、interval で時刻間隔（時、分、秒、秒の小数部）が指定されている場合、結果の間隔数の計算前に、タイムスタンプの欠けている時刻部分は、既定で '00:00:00.000' になります。
- 任意の桁数の精度の秒の小数部を含めることができます。省略してもかまいません。SQL\_TSI\_FRAC\_SECOND は、秒の小数部の差異を 1000 分の 1 秒（3 桁の精度）単位の整数として返します。%Library.PosixTime 値には、常に 6 桁の精度が含まれます。

## 例

### タイムスタンプ間の差異の計算

以下の文は、後者のタイムスタンプ (2021/12/20 12:00:00) が、前者のタイムスタンプよりも 7 か月分大きいので、7 を返します。

#### SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MONTH,
    '2021-5-19 00:00:00', '2021-12-20 12:00:00')}
```

以下の文は、後者のタイムスタンプ (12:00:00) が、前者のタイムスタンプ (02:34:12) よりも 566 分大きいので、566 を返します。

#### SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MINUTE, '02:34:12', '12:00:00')}
```

以下の文は、後者のタイムスタンプが前者のタイムスタンプよりも値が 1 日（1440 分）小さいので -1440 を返します。

#### SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MINUTE, '2021-12-06', '2021-12-05')}
```

## 制約

- TIMESTAMPDIFF は、ODBC スカラ関数としてのみ使用できます。そのため、{} 括弧構文が必要です。タイムスタンプに対して同様の日時比較演算を実行するには、DATEDIFF 関数を使用します。

## 詳細

### 範囲と値のチェック

差異の計算を実行する前に、TIMESTAMPDIFF は入力値に対して以下のチェックを実行します。

- startDate および endDate の指定された部分がすべて有効である。時刻値は、全体または一部を省略できます。startDate または endDate で不完全な時刻が指定されている場合、TIMESTAMPDIFF は、指定されていない部分に 0 を指定します。
- 日付文字列が完全であると同時に、要素数、各要素の桁数、および該当する区切り文字に適切な形式が使用されている。年は 4 桁で指定される必要があります。無効な日付値を指定すると、SQLCODE -8 エラーになります。
- 日付値が有効な範囲内にある。年は 0001 から 9999、月は 1 から 12、日は 1 から 31、時間は 00 から 23、分は 0 から 59、秒は 0 から 59 がそれぞれ有効範囲です。月の日数は、該当月と該当年に合ったものでなければなりません。

ん。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。無効な日付値を指定すると、SQLCODE -8 エラーになります。

- ・ 日付値にキャノニック形式の整数値のみが含まれている。例外：月または日の値が 10 (10 月または 10 日) 未満の場合、先頭に 0 を含めることができます。したがって、07 または 7 は有効な日の値ですが、007、7.0、または 7a は無効です。
- ・ 10 未満の時間値の先頭に 0 が含まれている。この先頭の 0 を省略すると、SQLCODE -8 エラーになります。

## 関連項目

- ・ [TIMESTAMPADD](#)
- ・ [DATEDIFF](#)
- ・ [TO\\_POSIXTIME](#)
- ・ [TO\\_TIMESTAMP](#)

## TO\_CHAR (SQL)

日付、タイムスタンプ、または値数をフォーマットされた文字列へ変換する文字列関数です。

### 構文

```
TO_CHAR(expression,format)
```

```
TO_CHAR(expression)
```

```
TOCHAR(...)
```

### 説明

- TO\_CHAR([expression](#),[format](#)) は、日付、時刻、タイムスタンプ（日付と時刻）、または数式を、指定された形式文字列に応じて文字列に変換します。

以下の文は、現在の日付を 'MONTH DD, YYYY' の形式に変換します。ここで、MONTH は完全な月名、DD は 2 桁の日、YYYY は 4 桁の年数です。

#### SQL

```
SELECT TO_CHAR(CURRENT_DATE, 'MONTH DD, YYYY')
```

例：

- 日付をフォーマットされた日付文字列に変換する
- 時刻をフォーマットされた時刻文字列に変換する
- タイムスタンプをフォーマットされた日付と時刻の文字列に変換する
- 数をフォーマットされた数値文字列に変換する

TO\_CHAR(expression) は、日付、時刻、タイムスタンプ、または数式を、式のタイプの既定の論理モード形式に応じて変換します。

- 日付式と時刻式は、InterSystems SQL の論理 [\\$HOROLOG](#) 形式に変換します。これは、日付と時刻を表す 2 つのコンマ区切り整数の文字列です。最初の整数は 1840 年 12 月 31 日からの日数です。2 番目の整数は、現在の日付の午前 0 時からの秒数です。
- タイムスタンプ式は YYYY-MM-DD HH:MI:SS 形式に変換されます。
- 数式は整数に変換されます。先頭のゼロとプラス記号は削除され、数は最初の非数値文字（コンマやピリオドなど）で切り捨てられます。

以下の文は、タイムスタンプとして表される現在の日付と時刻を、YYYY-MM-DD HH:MI:SS 形式の文字列に変換します。

#### SQL

```
SELECT TO_CHAR(CURRENT_TIMESTAMP)
```

- TOCHAR(...) は、TO\_CHAR(...) と同等です。

## 引数

### expression

**format** で指定された形式に従って文字列に変換される、論理日付、時刻、タイムスタンプ、または数の式。expression が NULL の場合、TO\_CHAR は NULL を返します。

### 日付式

日付式を変換するには、expression は数値または **\$HOROLOG** 形式の文字列である必要があります。

expression が無効な日付 (February 30 など) である場合、InterSystems IRIS® は、SQLCODE -400 エラーを発行します。

expression が 1840 年 12 月 31 日以前の日付を表している場合、日付を変換するには、ユリウス日付形式 (format 引数 = 'J') を使用する必要があります。詳細は、“[ユリウス日の変換](#)”を参照してください。

### 時刻式

時刻式を変換するには、expression が以下の形式のいずれかである必要があります。

- ・ **\$HOROLOG** 時刻整数 (**\$HOROLOG** の時刻コンポーネント)。ここで expression は、0 ～ 86399 の範囲の有効な論理時刻整数です。日付と時刻の両方のコンポーネントを含む (64701, 42152 など) 完全な **\$HOROLOG** 値を指定しないでください。TO\_CHAR 時刻変換では、**\$HOROLOG** の最初のコンポーネントである日付コンポーネントのみをフォーマットされた時刻文字列に変換し、2 番目のコンポーネントである時刻コンポーネントは無視します。
- ・ 論理タイムスタンプ値。expression の値は (文字列データ型ではなく) YYYY-MM-DD hh:mm:ss 形式の **%TimeStamp** データ型である必要があります。format で時刻形式のみを指定する場合、TO\_CHAR は、タイムスタンプの日付コンポーネントを無視し、時刻コンポーネントのみを変換します。例えば、**SYSDATE** は論理タイムスタンプです。
- ・ ODBC 標準時刻形式の時刻値。expression の値は hh:mm:ss 形式にする必要がありますが、文字列になります。
- ・ 現在の NLS ロケール設定を使用したローカル時刻形式による時刻値。例えば、NLS TimeSeparator が “” に設定されている場合、expression の値は hh^mm^ss 形式の文字列になります。

expression が無効な時刻 (6:61 P.M. など) である場合、InterSystems IRIS は、SQLCODE -400 エラーを発行します。

### タイムスタンプ式

タイムスタンプ式を変換するには、expression は、YYYY-MM-DD HH:MI:SS の形式にするか、以下の有効なバリエーションのいずれかにする必要があります。

- ・ 10 より小さい月および日付の値の場合、先頭のゼロはオプションです。先頭のゼロが省略された場合、返される日付でもゼロが省略されます。
- ・ 秒の値は省略できますが、その場所を示すコロンは指定する必要があります (HH:MI: など)。返される時刻では、秒は既定で 00 となります。
- ・ 秒の値には、秒の小数部を含めることができます (HH:MI:SS.fff など)。返される時刻では、秒の小数部は切り捨てられます。
- ・ format に時間の形式を指定しなくても、タイムスタンプには時間の部分を含める必要があります。

expression が正しいタイムスタンプの形式でない場合、TO\_CHAR はそれを整数と解釈し、整数でない文字が最初に出現したところで解釈を終了します。

format が日付またはタイムスタンプ形式の場合、TO\_CHAR は expression を **\$HOROLOG** 日付整数と解釈します。例えば、2010-03-23 12-15:23 では、時刻の値の部分に誤ってハイフンが使用されていますが、これは **\$HOROLOG** 日付の 2010 (1846-07-03 12:00:00 AM) と解釈されます。

expression が無効な日付または時刻 (February 30、6:61 P.M など) である場合、InterSystems IRIS は、SQLCODE - 400 エラーを発行します。

## 数式

数式を変換するには、expression は、数値データ型または数値文字列にする必要があります。TO\_CHAR では文字列を最初の非数値整数で切り捨てます。先頭に数値のない文字列は 0 と解釈されます。

## format

expression 変換用に日付、タイムスタンプ、または数の形式を指定する文字コード。

- format に無効な日付、時刻、またはタイムスタンプ・コードの要素を指定した場合 (YYYYY、MIN、HH48 など)、TO\_CHAR は無効なコード要素の形式コード・リテラルを返します。その他すべての有効なコード要素については、日付、時刻、またはタイムスタンプの変換値を返します。
- TO\_CHAR が format コード要素を認識できない場合 (format が空文字列の場合など) や数字の形式における桁数が expression 値より小さい場合、TO\_CHAR は元の文字の代わりにシャープ記号 (#) を返します。これは、expression が少なくとも 2 桁の整数で始まる場合に当てはまります。そうでない場合、TO\_CHAR は NULL を返します。

以下の表に、有効な形式コードを示します。これらを、それぞれの式のタイプ (日付、時刻、日付と時刻 (タイムスタンプ)、数) で指定できます。

テーブル G-11: 日付形式

形式コード	意味
D	曜日 (1 ~ 7)。既定では、1 が日曜日 (週の最初の日) ですが、この指定は構成可能です。詳細は、“ <a href="#">DAYOFWEEK (SQL)</a> ” を参照してください。
DD	2 桁の日付 (01 ~ 31)。
DY	現在のロケールの WeekdayAbbr プロパティによって指定される日の略名。 既定値 : Sun Mon Tue Wed Thu Fri Sat
DAY	現在のロケールの WeekdayName プロパティが指定する曜日名。 既定値 : Sunday Monday Tuesday Wednesday Thursday Friday Saturday
MM	2 桁の月数 (01 ~ 12、01 = 1 月)。
MON	現在のロケールの MonthAbbr プロパティによって指定される月の略名。 既定値 (大文字と小文字の区別なし) : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
MONTH	現在のロケールの MonthName プロパティによって指定される正式な月名。 既定値 (大文字と小文字の区別なし) : January February March April May June July August September October November December
YYYY	4 桁の年数。
YYY	下 3 桁の年数。
YY	下 2 桁の年数。
Y	下 1 桁の年数。
RRRR	4 桁の年数。
RR	下 2 桁の年数。

形式コード	意味
DDD	年間通算日 (指定された年の 1 月 1 日からの日数)
J	ユリウス日 (紀元前 4712 年 1 月 1 日からの日数)。詳細は、“ <a href="#">ユリウス日の変換</a> ”を参照してください。

YYYYMMDD、DDMMYYYY、および YYYYMM の形式文字列を除いて、区切り文字は日付形式要素間で必要です。最後の形式では年と月の値が返され、月の日付は無視されます。

format のコード定義に説明のあるロケールは、ObjectScript の [\\$ZDATE](#) と [\\$ZDATEH](#) のドキュメントに記述されているロケールと同じです。

#### テーブル G-12: 時刻形式

形式コード	意味
HH	時 (1 ~ 12)
HH12	時 (1 ~ 12)
HH24	時 (0 ~ 23)
MI	分 (0 ~ 59)
SS	秒 (0 ~ 59)
SSSSS	午前 0 時 00 分からの秒数 (0 ~ 86388)
AM と PM	午前/午後のインジケータ (AM = 午前、PM = 午後)。適切な AM または PM の接尾語が付加された 12 時間形式に時刻値を変換します。返される AM または PM の接尾語は、指定した形式コードからではなく、時刻値から導出されます。format では、AM または PM のいずれかを使用できます。これらは、機能的には同じです。

時刻を文字列に変換する際、format はこの表に示した時刻形式コードのみを含む文字列である必要があります。format に他のコードが含まれる場合、TO\_CHAR は expression を代わりに日付として解釈します。

タイムスタンプをフォーマットされた日付/時刻文字列に変換する際、format は“日付形式”または“時刻形式”の表に示した日付/時刻形式コードを含む文字列である必要があります。この変換を実行するには、expression は、有効な論理タイムスタンプ値である必要があります。

#### テーブル G-13: 数の形式

形式コード	説明	例
9	指定された桁数の値を返します。 <ul style="list-style-type: none"> <li>正の値は先頭にスペースが付きます。</li> <li>負の値はマイナス記号が付きます。</li> <li>固定小数点の整数部にゼロを戻すゼロ値を除いては、先頭のゼロは空白になります。</li> </ul>	9999
0	先頭または末尾のゼロを付けて返します。	09999 99990
\$	先頭に \$ を付けて値を返します。正の数の場合、ドル記号の前には空白が入ります。	\$9999



形式コード	説明	例
B	整数部がゼロの場合、(format 引数の 0 の有無にかかわらず) 固定小数点の整数部に空白を返します。	B9999
S	正の場合は先頭または末尾のプラス記号 "+" を付け、負の場合は先頭または末尾のマイナス記号 "-" を付けて値を返します。	S9999 9999S
D	指定された位置に小数点区切り文字を配置して返します。使用される DecimalSeparator は、ロケールで定義されたものと同じです。既定はピリオド "." です。format 引数内には、"D" は 1 つしか許可されません。	99D99
G	指定された位置に数値グループ・セパレータを配置して返します。使用される NumericGroupSeparator は、ロケールで定義されたものと同じです。既定はコンマ "," です。小数点区切りの右側に数値グループ・セパレータが出現することはありません。	9G999
FM	先頭や末尾に空白がない値を返します。	FM90.9
,	指定された位置にコンマを付けて返します。10 進法の右側にコンマが表れることはありません。format 引数をコンマで始めることはできません。	9,999
.	指定した位置に 10 進小数点 (ピリオド ".") を返します。format 引数内には、"." は 1 つしか許可されません。	99.99

format では、小数区切りおよび数値グループ・セパレータを、リテラル文字として、またはロケールの DecimalSeparator と NumericGroupSeparator の現在の値として指定できます。現在のロケール値は ObjectScript を使用して以下のように確認できます。

#### ObjectScript

```
write ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator"),!  
write ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator")
```

format に含まれる整数桁数が入力数値式より小さい場合、TO\_CHAR は数値を返さず、複数のシャープ記号(##)の文字列を返します。シャープ記号の数は、現在の format 引数の長さに 1 を加えたものです。

format に含まれる小数桁数が入力数値式より小さい場合、TO\_CHAR は指定された桁数に数値を丸めます。10 進形式が指定されていない場合、TO\_CHAR は数値を整数に丸めます。

## 例

### 日付をフォーマットされた日付文字列に変換する

以下の文では、TO\_CHAR を使用して、\$HOROLOG 日付整数または完全な \$HOROLOG 文字列値を、フォーマットされた日付文字列または日付と時刻の文字列に変換します。

#### SQL

```
SELECT  
  TO_CHAR(66256,'YYYY-MM-DD') AS Date2FormattedDate,  
  TO_CHAR(66256,'YYYY-MM-DD HH24:MI:SS') AS Date2FormattedDateTime,  
  TO_CHAR('66256,50278','YYYY-MM-DD') AS DateTime2FormattedDate,  
  TO_CHAR('66256,50278','YYYY-MM-DD HH24:MI:SS') AS DateTime2FormattedDateTime
```

以下の文では、各 TO\_CHAR 呼び出しは日付整数を取り、format 文字列引数に従ってフォーマットされた日付文字列を返します。



## SQL

```
SELECT
  TO_CHAR(66256,'MM/DD/YYYY'),          /* returns 02/22/2018 */
  TO_CHAR(66256,'DAY MONTH DD, YYYY') /* returns Thursday February 22, 2018 */
```

以下の文は、日付整数をフォーマットされた日付文字列に変換します。無効な format 文字は、リテラルとして出力文字列に渡されます。これは文字列 `The date 05/27/2022 should be noted` を返します。

## SQL

```
SELECT TO_CHAR(66256,'The date MM/DD/YYYY should be noted')
```

以下の文は、日付式を、指定された年の 1 月 1 日からの経過日数として定義された、その年の日数に変換します。この構文を使用するには、日付式を \$HOROLOGY 形式にする必要があります。2 番目の TO\_CHAR 呼び出しの時刻値は無視されます。2 つの TO\_CHAR 呼び出しでは、年間通算日形式の要素 (DDD) と年要素 (YYYY および YY) は、異なる順序で表示されます。

## SQL

```
SELECT
  TO_CHAR('66235','DDD days into YYYY'),
  TO_CHAR('66235,12345','Year YY: DDD days elapsed')
```

以下の文では、TO\_CHAR は、セパレータをマイナス記号と解釈するため、正しくない日付値を返します。したがって、この式は  $2022 - 5 - 2 = 2015$  と評価されます。これは、\$HOROLOGY 整数形式で日付 1846-07-08 に対応します。

## SQL

```
SELECT TO_CHAR(2022-05-02,'YYYY-MM-DD') -- Incorrect usage
```

## 時刻をフォーマットされた時刻文字列に変換する

以下の文では、'66256' が時刻値 05:58:21 PM と解釈されます。

## SQL

```
SELECT TO_CHAR('66256','HH12:MI:SS PM')
```

以下の文では、2 つの論理タイムスタンプの時刻部分をフォーマットされた時刻文字列に変換します。format は秒の小数部をサポートしていないため、expression の秒の小数部は切り捨てられます。

## SQL

```
SELECT TO_CHAR(SYSDATE,'HH12:MI:SS PM'),
       TO_CHAR(CURRENT_TIMESTAMP(6),'HH12:MI:SS PM')
```

## タイムスタンプをフォーマットされた日付と時刻の文字列に変換する

以下の文は、現在のシステム日付 (タイムスタンプ) と、システム日付を表示するためにこのタイムスタンプを 2 つの異なるフォーマットで変換した値を返します。

## SQL

```
SELECT
  SYSDATE,
  TO_CHAR(SYSDATE,'MM/DD/YYYY HH:MI:SS'),
  TO_CHAR(SYSDATE,'DD MONTH YYYY at SSSSS seconds')
```

format 文字列で使用されている、形式コードでない文字列はすべて、結果文字列のその位置に戻されます。

## 数をフォーマットされた数値文字列に変換する

以下の文は、数値 1000 をさまざまな桁数の形式コードの文字列に変換します。

- ・ 最初の変換では、数値の桁数は、指定された桁の形式コードより多くなります。TO\_CHAR は、数値の桁数に等しいシャープ記号を返します。
- ・ 2 番目の変換では、数値の桁数は、指定された桁の形式コードと同じになります。TO\_CHAR は、数値を文字列形式で返します。数値は符号なしの正の整数であるため、TO\_CHAR は数値文字列の先頭にゼロを付加します。
- ・ 3 番目の変換では、数値の桁数は、指定された桁の形式コードより少なくなります。TO\_CHAR は、文字列形式で数値を返し、先頭に2つのゼロを付加します。1 つは、数値が符号なしの正の整数であるため、もう1つは余分な桁の形式コード用です。

### SQL

```
SELECT
  TO_CHAR(1000,'999'), -- '####'
  TO_CHAR(1000,'9999'), -- ' 1000'
  TO_CHAR(1000,'99999') -- '   1000'
```

以下の文は、区切り文字の使用例を示しています。

- ・ 最初の変換は、文字列 ' 1,000.00' を返します。
- ・ 2 番目の変換も同じ値を返しますが、区切り文字はロケールの設定により表示が異なります。

### SQL

```
SELECT
  TO_CHAR(1000,'9,999.99'),
  TO_CHAR(1000,'9G999D99')
```

以下の文は、正の符号と負の符号の使用例を示しています。先頭の空白は、符号の形式がない場合に正の数の前のみ表示されます。先頭の空白は、符号の配置にかかわらず、負の数の前、または符号付きの数値の前には表示されません。

### SQL

```
SELECT
  TO_CHAR(10,'99.99'), -- ' 10.00'
  TO_CHAR(-10,'99.99'), -- '-10.00'
  TO_CHAR(10,'S99.99'), -- '+10.00'
  TO_CHAR(-10,'S99.99'), -- '-10.00'
  TO_CHAR(10,'99.99S'), -- '10.00+'
  TO_CHAR(-10,'99.99S') -- '-10.00-'
```

以下の文は、"FM" 形式の使用例を示します。これは、符号なしの正の数に対し、既定の先頭の空白をオーバーライドします。

### SQL

```
SELECT
  TO_CHAR(12345678.90,'99,999,999.99'), -- ' 12,345,678,90'
  TO_CHAR(12345678.90,'FM99,999,999.99') -- '12,345,678,90'
```

以下の文は、先頭のドル記号の使用例を示しています。ドル記号の前には必ず符号または空白文字が入ります。

### SQL

```
SELECT
  TO_CHAR(1234567890,'$9G999G999G999'), -- '$1,234,567,890'
  TO_CHAR(1234567890,'S$9G999G999G999'), -- '+$1,234,567,890'
  TO_CHAR(12345678.90,'$99G999G999D99') -- '$1,234,567,8.90'
```

以下の文は、format 引数が入力数値式より小数桁数が小さい場合の動作を示します。返される数字は、それぞれ、1234567.5 と 1234568 に丸められます。

## SQL

```
SELECT
  TO_CHAR(1234567.4999, '9999999.9'),
  TO_CHAR(1234567.91, '9999999')
```

## 詳細

### ユリウス日の変換

ユリウス日付形式を使用すると、1840 年 12 月 31 日より前の日付を文字列に変換できます。この形式を使用するには、TO\_CHAR の [format](#) 引数を 'J' または 'j' に指定します。この形式を使用して、データ型 **%Date** または **%TimeStamp** の日付値を、必要に応じて先頭にゼロを付けた 7 桁のユリウス日の整数に変換します。以下に例を示します。

## SQL

```
SELECT
  TO_CHAR('1776-07-04', 'J') AS UnitedStatesStart, --2369916
  TO_CHAR('-0031-09-02', 'J') AS RomanEmpireStart  --1709980
```

返される整数は、紀元前 4712 年 1 月 1 日からの日数です。ユリウス日に変換できる [expression](#) の最大値は '9999-12-31' (ユリウス日のカウント 5373484) です。最小値は '-4712-01-01' (ユリウス日のカウント 0000001) です。

既定では、**%Date** データ型は [1840 年 12 月 31 日](#) より前の日付を表しません。ただし、このデータ型の MINIVAL パラメータを再定義すると、それより前の日付を西暦 1 年 1 月 1 日まで、負の整数として表すことができます。この負の整数による日付の表記には、ここで説明するユリウス日付形式とは互換性がありません。詳細は、“[データ型](#)”を参照してください。

ユリウス日のカウント値 1721424 は、ユリウス暦の 1 年 1 月 1 日 (1-01-01) を返します。ユリウス日のカウント値がこの値より小さい場合は、BCE (紀元前) の日付が返されます。これは、年の前にマイナス記号を付けて表示されます。

TO\_CHAR では、日付式に対応するユリウス日のカウントを返すことができます。TO\_DATE では、以下の例のように、ユリウス日のカウントに対応する日付式を返すことができます。

## SQL

```
SELECT
  TO_CHAR('1776-07-04', 'J') AS JulianCount, -- 2369916
  TO_DATE(2369916, 'J') AS JulianDate      -- 1776-07-04
```

## 関連する SQL 関数

- TO\_CHAR は、日付整数、タイムスタンプ、または数を文字列に変換します。
- [TO\\_DATE](#) は、日付に対して逆の操作を実行します。これは、フォーマットされた日付文字列を日付整数に変換します。
- [TO\\_TIMESTAMP](#) は、タイムスタンプに対して逆の操作を実行します。これは、フォーマットされた日付/時刻文字列を標準のタイムスタンプに変換します。
- [TO\\_NUMBER](#) は、数値に対して逆の操作を実行します。これは、数値文字列を数値に変換します。
- [CAST](#) および [CONVERT](#) は、DATE、TIMESTAMP、および NUMBER の各データ型に対して変換を実行します。

## 代替案

ObjectScript で同等の変換を実行するには、TOCHAR() メソッドを使用します。

```
$SYSTEM.SQL.Functions.TOCHAR(expression,format)
```

## 関連項目

- ・ SQL 関数: [CONVERT](#)、[TO\\_DATE](#)、[TO\\_NUMBER](#)
- ・ ObjectScript 関数: [\\$FNUMBER](#)、[\\$ZDATE](#)

# TO\_DATE (SQL)

フォーマットされた文字列を日付に変換する日付関数です。

## 構文

```
TO_DATE(dateString)
TO_DATE(dateString,format)
TODATE(...)
```

## 概要

TO\_DATE 関数は、さまざまな形式の日付文字列をデータ型 DATE の日付整数値に変換します。この関数は、さまざまな文字列形式で日付を入力し、その値を標準の InterSystems IRIS® 表現で保存するために使用します。

TO\_DATE は、1840 年 12 月 31 日からの日数を示す整数値を返します。0 は 1840 年 12 月 31 日を表し、最小値の -672045 は 0001 年 1 月 1 日を表し、最大値の 2980013 は 9999 年 12 月 31 日を表します。

- TO\_DATE(dateString) は、既定の形式 DD MON YYYY を使用して日付文字列を解析します。

以下の文は、日付文字列 22 Feb 2022 を整数 66162 に変換します。

### SQL

```
SELECT TO_DATE('22 FEB 2022')
```

例：既定の日付形式

- TO\_DATE(dateString,format) は、指定された形式文字列を使用して日付文字列を解析します。dateString の日付要素は、format の形式要素に対応している必要があります。

以下の文は、前の構文と同じ変換を実行しますが、カスタム形式を使用して日付文字列を指定することができます。

### SQL

```
SELECT TO_DATE('2-22-22','M-DD-YY')
```

例：

- 指定された日付形式
- スタンドアロンの日付要素形式
- 年間通算日の変換 (DDD 形式)
- 2 桁の年の変換 (RR および RRRR 形式)
- 既定の日付列値の設定

- TODATE(...) は、TO\_DATE(...) と同等です。

## 引数

### dateString

dateString 引数は、日付に変換される日付文字列を指定する文字列式です。dateString の基本となるデータ型は CHAR または VARCHAR2 である必要があります。

dateString の各文字は、以下の例外を除いて format 文字列に対応している必要があります。

- 区切り文字のない dateString の場合を除き、先頭のゼロは指定しても省略してもかまいません。

- ・ 年は 2 桁または 4 桁で指定できます。
- ・ 月名は、完全な名前または月名の最初の 3 文字で指定できます。正確に入力する必要があるのは最初の 3 文字のみです。月の名前では、大文字と小文字は区別されません。
- ・ 日付に追加された時刻の値は無視されます。

指定可能な最も早い日付は 1840 年 12 月 31 日で、これは InterSystems IRIS では論理整数 0 として表されます。さらに早い日付を指定するには、ユリウス日付形式を使用します。“[ユリウス日 \(J 形式\)](#)”を参照してください。

## format

format 引数は、[dateString](#) の形式を指定する日付文字列です。TO\_DATE は、format の対応する位置にある形式要素に基づいて、dateString の日付要素を変換します。format の要素は以下のルールに従う必要があります。

- ・ 形式要素では、大文字と小文字は区別されません。
- ・ ほとんど任意の数と順序で形式要素を使用できます。
- ・ 形式文字列は、dateString の区切り文字に一致する、英数字でない区切り文字（スペース、スラッシュ、ハイフンなど）で要素を区切ります。この指定された日付区切り文字の使用は、NLS（各国言語サポート）ロケールに対して定義されている DateSeparator に依存しません。
- ・ MMDDYYYY、DDMMYYYY、YYYYMMDD、YYYYDDMM の各日付形式文字列には、区切り文字は必要ありません。不完全な日付形式 YYYYMM も使用でき、DD 値は 01 と見なされます。これらの形式では、MM と DD の値には先頭のゼロが必要です。

以下の表は、有効な日付形式要素を示しています。大文字の形式が示されていますが、これらの要素では大文字と小文字が区別されません。

要素	意味
DD	2 桁の日付 (01 ~ 31)format に日付の区切り文字を記述している場合、各要素の先頭にゼロは不要です。
MM	2 桁の月数 (01 ~ 12、01 = 1 月)。format に日付の区切り文字を記述している場合、各要素の先頭にゼロは不要です。  日本語と中国語では、月数は数値とそれに続く表意文字“月”で構成されます。
MON	現在のロケールの MonthAbbr プロパティによって指定される月の略名。既定で、これは英語による月名の最初の 3 文字です。その他のロケールでは、月名の略名が 4 文字以上になる場合や、月名の最初の数文字で構成されない場合があります。ピリオド文字の使用は認められていません。大文字と小文字は区別されません。
MONTH	現在のロケールの MonthName プロパティによって指定される正式な月名。大文字と小文字は区別されません。
YYYY	4 桁の年数
YY	下 2 桁の年数。2 桁の年数の最初の 2 桁は既定で 19 になります。
RR / RRRR	2 桁の年数から 4 桁の年数への変換詳細は、“ <a href="#">2 桁の年の変換 (RR および RRRR 形式)</a> ”を参照してください。
DDD	年間通算日。1 月 1 日からの日数“ <a href="#">年間通算日の変換 (DDD 形式)</a> ”を参照してください。
J	ユリウス日 1840 年 12 月 31 日より前の日付を表すために使用します。“ <a href="#">ユリウス日 (J 形式)</a> ”を参照してください。

TO\_DATE format には、D (曜日番号)、DY (曜日の省略形)、または DAY (曜日名) 要素を含めることもできます。ただし、これらの形式要素は、検証されることや戻り値の決定に使用されることはありません。これらの format 要素の詳細は、“[TO\\_CHAR](#)” を参照してください。

format を省略すると、TO\_DATE は既定の形式 DD MON YYYY を使用して日付文字列を解析します。例えば、'22 Feb 2018' です。既定の日付形式をシステム全体で変更するには、[TODATEDefaultFormat](#) 構成パラメータを変更します。

現在の設定を確認するには、`$SYSTEM.SQL.CurrentSettings()` を呼び出します。これにより、[ TO\_DATE ] の設定が表示されます。

## 例

### 既定の日付形式

以下の文は、既定の日付形式を使用して解析される日付文字列を指定します。これらの文字列の両方は、60537 の DATE データ型内部値に変換されます。

#### SQL

```
SELECT
    TO_DATE('29 September 2018'),
    TO_DATE('29 SEP 2018')
```

以下の文は、既定の format を使用した 2 桁の年で日付文字列を指定します。2 桁の年数は既定で 1900 ～ 1999 になっていることに注意してください。このため、内部 DATE 値は 24012 になります。

#### SQL

```
SELECT
    TO_DATE('29 September 06'),
    TO_DATE('29 SEP 06')
```

### 指定された日付形式

以下の文は、さまざまな形式で日付文字列を指定します。これらの文字列のすべては、64701 の DATE データ型内部値に変換されます。

#### SQL

```
SELECT
    TO_DATE('2018 Feb 22','YYYY MON DD'),
    TO_DATE('FEBRUARY 22, 2018','month dd, YYYY'),
    TO_DATE('2018***02***22','YYYY***MM***DD'),
    TO_DATE('02/22/2018','MM/DD/YYYY')
```

以下の文は、要素の区切り文字を必要としない日付形式を指定します。これは、日付の内部値 64701 を返します。

#### SQL

```
SELECT
    TO_DATE('02222018','MMDDYYYY'),
    TO_DATE('22022018','DDMMYYYY'),
    TO_DATE('20182202','YYYYDDMM'),
    TO_DATE('20180222','YYYYMMDD')
```

以下の文は、YYYYMM 日付形式を指定します。これは要素の区切り文字を必要としません。欠落している日付要素には 01 を設定して、64800 (2018 年 6 月 1 日) という日付を返します。

#### SQL

```
SELECT TO_DATE('201806','YYYYMM')
```

## スタンドアロンの日付要素形式

**format** 引数では、スタンドアロンの日付文字列として、DD、DDD、MM、または YYYY を指定できます。これらの形式文字列は、月、年、または月と年の両方が省略されるため、InterSystems IRIS では、これらが現在の月および年を参照しているものと解釈されます。

DD は、現在年の現在月の指定日の日付を返します。以下に例を示します。

### SQL

```
SELECT TO_DATE('24','DD')
```

DDD は、現在年で、指定された日数が経過した日の日付を返します。以下に例を示します。

### SQL

```
SELECT TO_DATE('300','DDD')
```

MM は、現在年の指定された月の初日の日付を返します。以下に例を示します。

### SQL

```
SELECT TO_DATE('8','MM')
```

YYYY は、指定された年の現在月の初日の日付を返します。以下に例を示します。

### SQL

```
SELECT TO_DATE('2022','YYYY')
```

## 2 桁の年の変換 (RR および RRRR 形式)

YY 形式では、2 桁の年の値に 19 が付加されて 4 桁の値に変換されます。例えば、07 は 1907 になり、93 は 1993 になります。RR 形式および RRRR 形式では、2 桁の年から 4 桁の年への変換をより柔軟に行います。

RR 形式の変換は現在の年に基づきます。

- ・ 現在の年が、ある世紀の前半にある場合、以下ようになります。
  - 00 ～ 49 までの 2 桁の年は現在の世紀の 4 桁の年に展開されます。
  - 50 ～ 99 までの 2 桁の年は前の世紀の 4 桁の年に展開されます。

以下の文は、現在の年が 2000 から 2050 までの間の場合に TO\_DATE が返す日付の表示形式を示しています。

### SQL

```
SELECT
  TO_DATE('29 September 00','DD MONTH RR'), -- 09/29/2000
  TO_DATE('29 September 18','DD MONTH RR'), -- 09/29/2018
  TO_DATE('29 September 49','DD MONTH RR'), -- 09/29/2049
  TO_DATE('29 September 50','DD MONTH RR'), -- 09/29/1950
  TO_DATE('29 September 77','DD MONTH RR')  -- 09/29/1977
```

- ・ 現在の年が、ある世紀の後半にある場合、すべての 2 桁の年が現在の世紀の 4 桁の年に展開されます。

以下の文は、現在の年が 2050 から 2099 までの間の場合に TO\_DATE が返す日付の表示形式を示しています。



## SQL

```
SELECT
  TO_DATE('29 September 00','DD MONTH RR'), -- 09/29/2000
  TO_DATE('29 September 21','DD MONTH RR'), -- 09/29/2021
  TO_DATE('29 September 49','DD MONTH RR'), -- 09/29/2049
  TO_DATE('29 September 50','DD MONTH RR'), -- 09/29/2050
  TO_DATE('29 September 77','DD MONTH RR')  -- 09/29/2077
```

RRRR 形式を使用することにより、2 桁と 4 桁の年を混合して入力することができます。TO\_DATE は、4 桁の年を変更せずに渡します。TO\_DATE は、この例で前述の RR 形式アルゴリズムを使用して、2 桁の年を 4 桁の年に変換します。

以下の文は、現在の年が 2000 から 2050 までの間の場合に TO\_DATE が返す日付の表示形式を示しています。

## SQL

```
SELECT
  TO_DATE('29 September 2021','DD MONTH RRRR'), -- 09/29/2021
  TO_DATE('29 September 21','DD MONTH RRRR'), -- 09/29/2021
  TO_DATE('29 September 1949','DD MONTH RRRR'), -- 09/29/1949
  TO_DATE('29 September 49','DD MONTH RRRR'), -- 09/29/2049
  TO_DATE('29 September 1950','DD MONTH RRRR'), -- 09/29/1950
  TO_DATE('29 September 50','DD MONTH RRRR')  -- 09/29/1950
```

## 年間通算日の変換 (DDD 形式)

DDD 形式を使用して、年間通算日 (1 月 1 日から経過した日数) を実際の日付に変換できます。この変換を実行するには、以下の点に留意してください。

- ・ **format** 引数には、DDD 形式要素を含める必要があります。また、オプションで YYYY、YY、RR、RRRR などの年形式を含めることができます。これらの要素は任意の順序で指定できますが、要素の間に区切り文字を追加する必要があります。年を省略した場合、TO\_DATE によって現在の年が既定値として使用されます。
- ・ **dateString** 引数には、対応する day および year の値を含める必要があります。
  - day は、1 から 365 までの整数です (year がうるう年の場合は 366)。
  - year は、InterSystems IRIS の標準の日付範囲内 (1841 から 9999) の年です。

以下の文は、2022 年の 60 日目を返します。

## SQL

```
SELECT TO_DATE('2022:60','YYYY:DDD') -- 03/01/2022
```

TO\_DATE は月の要素を変更せずに渡します。形式文字列に DD と DDD 要素の両方が含まれる場合、TO\_DATE は DDD 要素を処理し、DD 要素を無視します。例えば、以下の文は、2020 年 12 月 31 日ではなく、2020 年 2 月 29 日 (2020 年の 60 日目) を返します。

## SQL

```
SELECT TO_DATE('2020-12-31-60','YYYY-MM-DD-DDD')
```

TO\_DATE は、年間通算日そのものではなく、年間通算日を含む日付式を返します。この日付の値を返すには、**TO\_CHAR** を使用します。

## 既定の日付列値の設定

**CREATE TABLE** コマンドを使用してテーブルを作成する際に、TO\_DATE 関数を使用して列の既定値を設定できます。以下に例を示します。

## SQL

```
CREATE TABLE MyTable
(ID NUMBER(12,0) NOT NULL,
End_Year DATE DEFAULT TO_DATE('12-31-2021','MM-DD-YYYY') NOT NULL)
```

## 詳細

## ユリウス日 (J 形式)

ユリウス日付形式を使用すると、1840 年 12 月 31 日より前の日付を表すことができます。この形式を使用するには、TO\_DATE の format 引数を 'J' または 'j' に指定します。この形式を使用して、7 桁の内部数値 (ユリウス日のカウント) を形式設定済みの日付に変換できます。例えば、以下の文は、論理形式または ODBC 形式では 1585-01-31 を、表示形式では 01/31/1585 をそれぞれ返します。

## SQL

```
SELECT TO_DATE(2300000,'J')
```

ユリウス日のカウント値 1721424 は、ユリウス暦の 1 年 1 月 1 日 (1-01-01) を返します。ユリウス日のカウント値がこの値より小さい場合は、BCE (紀元前) の日付が返されます。これは、年の前にマイナス記号を付けて表示されます。

既定では、%Date データ型は 1840 年 12 月 31 日より前の日付を表しません。ただし、このデータ型の MINIVAL パラメータを再定義すると、それより前の日付を西暦 1 年 1 月 1 日まで、負の整数として表すことができます。この負の整数による日付の表記には、ここで説明するユリウス日付形式とは互換性はありません。詳細は、“[データ型](#)”を参照してください。

ユリウス日のカウントは常に内部的に 7 桁の数で表記され、必要に応じて先頭にゼロが付きます。TO\_DATE では、先頭にゼロを付けずにユリウス日のカウントを入力できます。許可されている最大のユリウス日付は 5373484 で、12/31/9999 を返します。許可されている最小のユリウス日付は 0000001 で、01/01/-4712 (BCE 4713 年 1 月 1 日) を返します。値がこの範囲外の場合には、SQLCODE -400 エラーが生成されます。

1721424 (1/1/1) より前のユリウス日のカウントには、Oracle など、他のソフトウェア実装と互換性があります。このカウントは、通常の用法における BCE 日付と同一ではありません。通常の用法では、0 年は存在せず、日付の範囲は 12/31/-1 から 1/1/1 になります。Oracle の用法では、1721058 から 1721423 までのユリウス日は単純に無効となり、エラーが返されます。InterSystems IRIS では、ユリウス日がこの範囲内にあると、存在しない 0 年がプレースホルダとして返されます。したがって、BCE 日付を使用する計算は、1 年調整して通常の用法に対応させる必要があります。これは、TO\_CHAR および TO\_DATE の使用による日付とユリウス日のカウントの相互変換に影響を与えることはありませんが、ユリウス日のカウントを使用して行う計算の一部に影響が及ぶ場合があります。また、これらの日付のカウントでは、グレゴリオ暦の導入による日付の変更が考慮されないことに注意してください。

TO\_DATE では、ユリウス日のカウントに対応する日付式を返すことができます。TO\_CHAR では、以下の例のように、日付式に対応するユリウス日のカウントを返すことができます。

## SQL

```
SELECT
  TO_CHAR('1776-07-04','J') AS JulianCount, -- 2369916
  TO_DATE(2369916,'J') AS JulianDate      -- 1776-07-04
```

## 関連する SQL 関数

- ・ TO\_DATE は、フォーマットされた日付文字列を日付整数に変換します。
- ・ TO\_CHAR は、逆の操作を実行します。これは日付整数をフォーマットされた日付文字列に変換します。
- ・ TO\_TIMESTAMP は、フォーマットされた日付/時刻文字列を標準のタイムスタンプに変換します。
- ・ CAST および CONVERT は、DATE データ型の変換を実行します。

## 代替案

ObjectScript で同等の日付変換を実行するには、TODATE() メソッドを使用します。

```
$SYSTEM.SQL.Functions.TODATE(dateString,format)
```

## 関連項目

- ・ SQL 関数: [CAST](#)、[CONVERT](#)、[TO\\_CHAR](#)、[TO\\_TIMESTAMP](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)、[\\$ZDATEH](#)

## TO\_NUMBER (SQL)

文字列式を NUMBER データ型の値に変換する文字列関数です。

### 構文

```
TO_NUMBER(stringExpression)
```

```
TONUMBER(stringExpression)
```

### 概要

- TO\_NUMBER([stringExpression](#)) は、入力文字列式をデータ型 NUMERIC のキャノニック形式の数値に変換します。入力文字列式がデータ型 DOUBLE の場合、TO\_NUMBER はデータ型 DOUBLE の数値を返します。以下の表に示されていないその他すべてのタイプは、stringExpression タイプを返します。

stringExpression タイプ	返されるタイプ
VARCHAR、VARBINARY、TIME	NUMERIC
BIT	TINYINT
DATE	INTEGER
TIMESTAMP POSIXTIME	BIGINT

以下のクエリは、特定の通りの番地を数値の昇順で返します。ORDER BY Home\_Street を指定して番地を変換せずに、数値にも変換しない場合、番地は文字列の照合順 (1、10、100、2、20、200 など) に従います。

### SQL

```
SELECT Name,Home_Street FROM Sample.Person WHERE Home_Street LIKE '%Oakhurst%'
ORDER BY TO_NUMBER(Home_Street)
```

例：

- [文字列から数値への変換操作](#)
- [変換された文字列の形式モード](#)

- TONUMBER(stringExpression) は、TO\_NUMBER(stringExpression) と同等です。

### 引数

#### stringExpression

変換される文字列式。式には、列の名前、文字列リテラル、または他の関数の結果を指定できます。基本となるデータ型は CHAR または VARCHAR2 です。

### 例

#### 文字列から数値への変換操作

以下の例は、数値文字列をキャノニック形式の数値に変換する場合に TO\_NUMBER が実行する異なる操作を示しています。SQL コメントに示されている、返される結果は論理モードです。変換された数値の表示方法の詳細は、["変換された文字列の形式モード"](#) を参照してください。

TO\_NUMBER は、先頭のプラス符号またはマイナス符号を解決します。

**SQL**

```
SELECT TO_NUMBER('-+123 feet') -- -123
```

**SQL**

```
SELECT TO_NUMBER('+--123 feet') -- 123
```

また、TO\_NUMBER は、指数記数法 ("E" または "e") を展開します。

**SQL**

```
SELECT TO_NUMBER('1e3') -- 1000
```

**SQL**

```
SELECT TO_NUMBER('1E-3') -- .001
```

TO\_NUMBER は、非数値文字 (文字や数値グループ・セパレータなど) を検出すると変換を停止します。

**SQL**

```
SELECT TO_NUMBER('7dwarves') -- 7
```

文字列式の先頭文字が数値でない場合、または式が空文字列 (') か -0 の場合、TO\_NUMBER は 0 を返します。

**SQL**

```
SELECT TO_NUMBER('question3') -- 0
```

**SQL**

```
SELECT TO_NUMBER('') -- 0
```

**SQL**

```
SELECT TO_NUMBER('-0') -- 0
```

TO\_NUMBER は、算術演算を解決しません。例えば、以下の文字列の場合、TO\_NUMBER は文字 "+" で変換を停止し、2 を返します。

**SQL**

```
SELECT TO_NUMBER('2+4') -- 2
```

文字列式に NULL を指定すると、TO\_NUMBER は NULL を返します。

**変換された文字列の形式モード**

返されるクエリ結果の数値形式は、論理モード、ODBC モード、表示モードのいずれを使用するかによって異なる場合があります。

文字列式が DOUBLE でない限り、TO\_NUMBER 関数は NUMERIC 型の数値を返します。NUMERIC データ型の既定の小数桁数は 2 です。したがって、クエリを表示モードで実行すると、InterSystems IRIS は結果セットを小数点以下 2 桁で表示します。

**SQL**

```
SELECT TO_NUMBER('-15 degrees F') -- Display Mode: -15.00
```

それ以下の小数桁数は 2 桁に丸められます。

## SQL

```
SELECT TO_NUMBER('-15.835 degrees F') -- Display Mode: -15.84
```

末尾のゼロは小数点以下 2 桁に解決されます。

## SQL

```
SELECT TO_NUMBER('-15.60000 degrees F') -- Display Mode: -15.60
```

データベース・ドライバを介して使用する場合も、TO\_NUMBER は、小数桁数 2 桁の NUMERIC 型として返します。論理モードまたは ODBC モードでは、返される値はキャノニック形式の数値です。また、小数桁数は使用されず、末尾のゼロは省略されます。

## SQL

```
SELECT TO_NUMBER('-15 degrees F') -- Logical/ODBC Mode: -15
```

## SQL

```
SELECT TO_NUMBER('-15.835 degrees F') -- Logical/ODBC Mode: -15.835
```

## SQL

```
SELECT TO_NUMBER('-15.60000 degrees F') -- Logical/ODBC Mode: -15.6
```

入力文字列式がデータ型 DOUBLE の場合、TO\_NUMBER はデータ型 DOUBLE の値を返します。すべての形式モードで、変換後の数値の全桁が表示されます。

## SQL

```
SELECT TO_NUMBER(CAST('-15.6 degrees F' AS DOUBLE)) -- -15.59999999999999644
```

## 詳細

### 関連する SQL 関数

- ・ TO\_NUMBER は、文字列をデータ型 NUMERIC の数値に変換します。
- ・ [TO\\_CHAR](#) は、TO\_NUMBER とは逆の操作を実行し、数を文字列に変換します。
- ・ [CAST](#) と [CONVERT](#) を使用することで、文字列を任意のデータ型の数値に変換できます。例えば、文字列をデータ型 INTEGER の数値に変換できます。
- ・ [TO\\_DATE](#) は、フォーマットされた日付文字列を日付整数に変換します。
- ・ [TO\\_TIMESTAMP](#) は、フォーマットされた日付/時刻文字列を標準のタイムスタンプに変換します。

### 関連項目

- ・ [データ型](#)

# TO\_POSIXTIME (SQL)

フォーマットされた日付文字列を %PosixTime タイムスタンプに変換する日付/時刻関数です。

## 構文

```
TO_POSIXTIME(date_string[,format])
```

## 説明

TO\_POSIXTIME 関数は、さまざまな形式の日付/時刻文字列をデータ型 **%Library.PosixTime** の %PosixTime タイムスタンプに変換します。TO\_POSIXTIME は、64 ビットの符号付き整数としてエンコードされた 1970-01-01 00:00:00 の任意の開始ポイントからの経過秒数に基づいて計算された値として %PosixTime タイムスタンプを返します。この日付からの実際の経過秒数（および秒の小数部）は、[UNIX® タイムスタンプ](#)（数値）です。InterSystems IRIS は、UNIX® タイムスタンプをエンコードして、%PosixTime タイムスタンプを生成します。%PosixTime タイムスタンプ値はエンコードされるため、1970-01-01 00:00:00 は 1152921504606846976 として示されます。1970-01-01 00:00:00 より前の日付は、負の整数値で示されます。詳細は、“%PosixTime データ型” を参照してください。

TO\_POSIXTIME ではタイムゾーンは変換されません。ローカル日付とローカル時刻はローカル %PosixTime タイムスタンプに変換され、UTC 日付と UTC 時刻は UTC %PosixTime タイムスタンプに変換されます。

%PosixTime でサポートされる最初の日付は 0001-01-01 00:00:00 で、論理値 -6979664624441081856 を持ちます。サポートされる最後の日付は 9999-12-31 23:59:59.999999 で、論理値 1406323805406846975 を持ちます。これらの制限は、ODBC 日付形式の表示に関する制限に対応します。**%Library.PosixTime** の MINVAL パラメータと MAXVAL パラメータを使用することで、これらの値をさらに制限できます。IsValid() メソッドを使用することで、数値が有効な %PosixTime 値であるかどうかを判定できます。

%PosixTime 値は、常に小数点以下 6 桁の精度の秒の小数部をエンコードします。%PosixTime の変換前、6 桁未満の精度の date\_string は 6 桁になるように 0 が追加され、7 桁以上の精度の date\_string は 6 桁に切り捨てられます。

date\_string でタイムスタンプのコンポーネントを省略している場合、TO\_POSIXTIME では不足コンポーネントを提供します。date\_string と format の両方で年が省略されている場合、現在の年が yyyy の既定値として使用されます。date\_string のみ年が省略されている場合、00 が既定値として使用されますが、年の format 要素に従って 4 桁の年に拡張されます。日または月の値が省略されている場合、01 が dd の既定値として使用され、01-01 が mm-dd の既定値として使用されます。不足する時刻コンポーネントの既定値は 00 です。秒の小数部はサポートされていますが、明示的に指定する必要があります。既定では、秒の小数部は指定されません。

TO\_POSIXTIME は、2 桁の年から 4 桁の年への変換をサポートしています。TO\_POSIXTIME は、12 時間形式の時刻から 24 時間形式の時刻への変換をサポートしています。うるう年の検証も含めて、日付および時刻要素値の範囲の検証機能があります。範囲検証違反がある場合は、SQLCODE -400 エラーが生成されます。

この関数は、ObjectScript から TOPOSIXTIME() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.TOPOSIXTIME(date_string,format)
```

TO\_POSIXTIME 関数は、フィールドに既定値を指定するときにデータ定義で使用することができます。以下はその例です。

```
CREATE TABLE mytest
(ID NUMBER(12,0) NOT NULL,
End_Year DATE DEFAULT TO_POSIXTIME('12-31-2018','MM-DD-YYYY') NOT NULL)
```

TO\_POSIXTIME は、CREATE TABLE 文や ALTER TABLE ADD COLUMN 文で使用できます。このコンテキストでは、date\_string にリテラル値のみを使用できます。詳細は、“[CREATE TABLE](#)” コマンドを参照してください。



## %PosixTime の表現

%PosixTime は、date\_string の精度に関係なく、6 桁の精度の秒の小数部をエンコードします。ODBC モードと表示モードでは、精度の末尾の 0 が切り捨てられます。

- ・ 論理モード：エンコードされた 64 ビット (19 文字) の符号付き整数。
- ・ ODBC モード：YYYY-MM-DD HH:MM:SS.FFFFFFFF。"%PosixTime LogicalToOdbc() メソッド" を参照してください。
- ・ 表示モード：\$ZDATETIME で説明されているように、現在のロケールの既定の日付/時刻形式 (dformat -1 および tformat -1) が使用されます。"%PosixTime LogicalToDisplay() メソッド" を参照してください。

## 関連する SQL 関数

- ・ TO\_POSIXTIME は、フォーマットされた日付/時刻文字列を %PosixTime タイムスタンプに変換します。
- ・ TO\_CHAR は、逆の操作を実行し、%PosixTime タイムスタンプをフォーマットされた日付/時刻文字列に変換します。
- ・ UNIX\_TIMESTAMP は、フォーマットされた日付/時刻文字列を UNIX® タイムスタンプに変換します。
- ・ TO\_DATE は、フォーマットされた日付文字列を日付整数に変換します。
- ・ CAST および CONVERT は、%PosixTime データ型の変換を実行します。

## 引数

### date-string

%PosixTime タイムスタンプに変換される文字列式。この式では日付値、時刻値、または日付と時刻値を指定できます。

### format

date\_string に対応する日付/時刻形式の文字列 (オプション)。省略した場合、既定の DD MON YYYY HH:MI:SS が使用されます。

## 日付/時刻文字列

date\_string 引数では、日付/時刻文字列リテラルが指定されます。時刻コンポーネントなしで日付文字列を指定すると、TO\_POSIXTIME によって時刻値 00:00:00 が指定されます。日付コンポーネントなしで時刻文字列を指定したときは、TO\_POSIXTIME によって現在の年の日付 01-01 (1 月 1 日) が指定されます。

date\_string の入力にはどのような種類の日付/時刻文字列でも指定できます。date\_string の各文字は、以下の例外を除いて、format 文字列に対応している必要があります。

- ・ 区切り文字のない date\_string の場合を除き、先頭のゼロは指定しても省略してもかまいません。
- ・ 年は 2 桁または 4 桁で指定できます。
- ・ 月の省略形 (format MON) は、そのロケールでの月の省略形と一致している必要があります。ロケールによっては、月の省略形は、月の名前の先頭の連続文字でない場合があります。月の省略形では、大文字と小文字は区別されません。
- ・ 月の名前 (format MONTH) は、完全な月名として指定する必要があります。ただし、TO\_POSIXTIME では、format MONTH で完全な月名を必要としません。これは完全な月名の最初の文字 (複数可) を受け取り、月のリストからその先頭の文字シーケンスに対応する最初の月を選択します。そのため、英語では、"J" = "January"、"Ju" = "June"、"Jul" = "July" となります。指定されるすべての文字は、完全な月名の連続文字と一致する必要があります。完全な月名を逸脱した文字はチェックされません。例えば、"Fe"、"Febru"、および "FebruaryLeap" はすべて有効な値です。"Febs" は無効な値です。月の名前では、大文字と小文字は区別されません。



- 時刻値は、該当するロケール用に定義されている時刻区切り文字を使用して入力できます。出力タイムスタンプは必ず、ODBC 標準時刻区切り文字のコロン (:) とピリオド (.) で時刻値を表します。時刻要素を省略すると、既定で 00:00:00 が使用されます。

## 形式

format は、以下の規則に従って指定された 1 つまたは複数の形式要素の文字列です。

- 形式要素では、大文字と小文字は区別されません。
- 形式要素をどのような順序で、またはいくつ使うかについては、たいいていの場合制限はありません。
- 形式文字列は、date\_string の区切り文字に一致する、英数字でない区切り文字 (スペース、スラッシュ、ハイフンなど) で要素を区切ります。これらの区切り文字は出力文字列には表示されません。出力文字列では標準のタイムスタンプ区切り文字であるハイフン (日付値)、コロン (時刻値)、および秒の小数部がある場合は必要に応じてピリオドが使用されます。この区切り文字の使用は、NLS ロケールに対して定義されている DateSeparator に依存しません。
- 日付形式文字列 MMDDYYYY、DDMMYYYY、YYYYMMDDHHMISS、YYYYMMDDHHMI、YYYYMMDDHH、YYYYMMDD、YYYYDDMM、HHMISS、および HHMI には、区切り文字は必要ありません。不完全な日付形式 YYYYMM も使用でき、DD 値は 01 と見なされます。ただし、このような場合、最後の要素を除くすべての要素 (MM 値と DD 値など) の先頭にゼロを記述する必要があります。
- 有効な形式要素がない format の文字は無視されます。

## 形式の要素

以下のテーブルは、format 引数に対する有効な日付形式要素を示しています。

要素	意味
DD	2 桁の日付 (01–31)。format に日付の区切り文字を記述している場合、各要素の先頭にゼロは不要です。
MM	2 桁の月数 (01 ~ 12、01 = 1 月)。format に日付の区切り文字を記述している場合、各要素の先頭にゼロは不要です。 日本語と中国語では、月数は数値とそれに続く表意文字 “月” で構成されます。
MON	現在のロケールの MonthAbbr プロパティによって指定される月の略名。既定で、これは英語による月名の最初の 3 文字です。その他のロケールでは、月名の略名が 4 文字以上になる場合や、月名の最初の数文字で構成されない場合があります。ピリオド文字の使用は認められていません。大文字と小文字は区別されません。
MONTH	現在のロケールの MonthName プロパティによって指定される正式な月名。大文字と小文字は区別されません。
YYYY	4 桁の年数。
YY	下 2 桁の年数。2 桁の YY の年数の最初の 2 桁は既定で 19 になります。
RR / RRRR	2 桁の年数から 4 桁の年数への変換(以下を参照)。
DDD	年間通算日。1 月 1 日からの日数。(以下を参照)。
HH	時間。午前/午後のインジケータ (AM または PM) が指定されているかどうかに応じて、01 ~ 12、または 00 ~ 23 によって指定されます。HH12 または HH24 と指定できます。
MI	分。00 ~ 59 によって指定されます。
SS	秒。00 ~ 59 によって指定されます。

要素	意味
FF	秒の小数部。FF は、1 桁以上の小数桁数が指定されていることを示します。date_string では、任意の小数桁数を指定できます。TO_POSIXTIME は、date_string で指定されている精度に関係なく、正確に 6 桁の精度を返します。
AM と PM	午前/午後のインジケータ。12 時間形式を指定します (以下を参照)。
A.M. と P.M.	午前/午後のインジケータ (ピリオド付き)。12 時間形式を指定します (以下を参照)。

TO\_POSIXTIME format には、入力値 date\_string と一致する D (曜日番号)、DY (曜日の省略形)、または DAY (曜日名) 要素を含めることもできます。ただし、これらの形式要素は、検証されることや戻り値の決定に使用されることはありません。これらの format 要素の詳細は、“[TO\\_CHAR](#)” を参照してください。

## 2 桁の年の変換 (RR および RRRR 形式)

RR 形式では、2 桁の年から 4 桁の年への変換を行います。TO\_POSIXTIME は、[\\$ZDATETIME](#) で説明されているように、現在のロケールの YearOption プロパティを使用する既定の日付形式 (dformat -1) を用いてこの変換を行います。

## 年間通算日 (DDD 形式)

DDD を使用して、年間通算日 (1 月 1 日から経過した日数) を実際の日付に変換します。形式文字列 DDD YYYY は、整数の日数と 4 桁の年で構成される、対応する date\_string と組み合わせる必要があります (2 桁の年は、DDD と共に使用する場合は RR (YY ではない) として指定する必要があります)。形式文字列 DDD の既定は、現在の年となります。経過した日数は、1 ~ 365 の範囲の正の整数である必要があります (YYYY がうるう年の場合は 366)。4 桁の年は、標準の InterSystems IRIS の日付範囲内 (1841 から 9999) で指定する必要があります (年を省略した場合、現在の年が既定値として使用されます)。DDD および年 (YYYY、RRRR、または RR) 形式要素は、任意の順序で指定できます。これらの間の区切り文字は必須です。この区切り文字として空白を使用することができます。以下の例は、この年間通算日の使用法を示しています。

### SQL

```
SELECT TO_POSIXTIME('2018:160','YYYY:DDD')
```

形式文字列に DD と DDD 要素の両方が含まれる場合、DDD 要素が優勢です。以下の例に示すとおり、2008-12-31 00:00:00 ではなく 2008-02-29 00:00:00 が返されます。

### SQL

```
SELECT TO_POSIXTIME('2018-12-31-60','YYYY-MM-DD-DDD')
```

TO\_POSIXTIME では、年間通算日に対応する日付式を返すことができます。TO\_CHAR では、日付式に対応する年間通算日を返すことができます。

## 1970 年より前の日付

TO\_POSIXTIME は、1970 年 1 月 1 日より前の日付を負の数値として示します。%PosixTime は、0001 年 1 月 1 日より前の日付や、9999 年 12 月 31 日より後の日付を示すことはできません。このような日付を入力しようとすると、SQLCODE -400 エラーになります。TO\_DATE 関数は、ユリウス日の日付形式で 0001 年 1 月 1 日より前の紀元前 (BCE) の日付を示すことができます。ユリウス日の変換は、7 桁の正の内部整数値 (ユリウス日のカウント) を表示形式または ODBC 形式の日付に変換します。ユリウス日では時刻値がサポートされません。

## 12 時間形式の時刻

%PosixTime タイムスタンプは、時刻を常に 24 時間形式で示します。date\_string は、12 時間形式または 24 時間形式で時刻を示すことができます。TO\_POSIXTIME は、以下のいずれかが適用される場合を除き 24 時間形式を想定しています。

- date\_string 時刻値の後ろに 'am' か 'pm' (ピリオドなし) が続く場合。これらの午前/午後のインジケータでは、大文字と小文字が区別されません。また、時刻値に付加することも、1 つ以上のスペースで区切ることもできます。
- format が、1 つまたは複数のスペースによって時刻形式から分離されており、'a.m.' または 'p.m.' 要素 (どちらか 1 つ) を含む時刻形式の後に続く場合。例えば、DD MON YYYY HH:MI:SS.FF P.M. のようになります。この format は 2:23pm、2:23:54.6pm、2:23:54 pm、2:23:54 p.m.、および 2:23:54 (AM と想定) のような 12 時間形式 date\_string 値をサポートします。午前/午後のインジケータでは、大文字と小文字が区別されません。ピリオド付きの午前/午後のインジケータを使用する場合、1 つ以上のスペースで区切る必要があります。

## 例

以下の埋め込み SQL の例では、現在のローカル日付/時刻を %PosixTime 値に変換しています(format では任意の小数桁数を示すのに "ff" が使用されます。この例では、3 桁の精度になります。%PosixTime は、末尾に 0 を 3 つ追加して、これを 6 桁の精度としてエンコードします)。この例では、次に %Posix LogicalToOdbc() メソッドを使用し、精度の末尾の 0 を削除して、この値を ODBC タイムスタンプとして表示しています。

### ObjectScript

```
SET tstime=$ZDATETIME($ZTIMESTAMP,3,1,3)
WRITE "local datetime in : ",tstime,!
&sql(SELECT
    TO_POSIXTIME(:tstime,'yyyy-mm-dd hh:mi:ss.ff')
    INTO :ptime)
IF SQLCODE=0 {
    WRITE "Posix encoded datetime: ",ptime,!
    SET ODBCout=##class(%PosixTime).LogicalToOdbc(ptime)
    WRITE "local datetime out: ",ODBCout }
ELSE { WRITE "SQLCODE error:",SQLCODE }
```

以下の例は、さまざまな形式で日付文字列を指定します。最初の例では既定の形式を使用しており、それ以外の例では format を指定しています。これらはすべて、date\_string をタイムスタンプ値 2018-06-29 00:00:00 に変換します。

### SQL

```
SELECT
    TO_POSIXTIME('29 JUN 2018'),
    TO_POSIXTIME('2018 Jun 29','YYYY MON DD'),
    TO_POSIXTIME('JUNE 29, 2018','month dd, YYYY'),
    TO_POSIXTIME('2018***06***29','YYYY**MM**DD'),
    TO_POSIXTIME('06/29/2018','MM/DD/YYYY'),
    TO_POSIXTIME('29/6/2018','DD/MM/YYYY')
```

以下の例では、YYYYMM 日付形式を指定しています。これは要素セパレータを必要としません。TO\_POSIXTIME では、不足している日付値と時刻値が補われます。

### SQL

```
SELECT TO_POSIXTIME('201806','YYYYMM')
```

この例では、タイムスタンプ 2018-06-01 00:00:00 を返しています。

以下の例では、HH:MI:SS.FF 時刻形式のみを指定しています。TO\_POSIXTIME では、不足している日付値が補われます。いずれの場合も、日付 2018-01-01 (2018 が現在の年の場合) が返されます。

### SQL

```
SELECT TO_POSIXTIME('11:34','HH:MI:SS.FF'),
    TO_POSIXTIME('11:34:22','HH:MI:SS.FF'),
    TO_POSIXTIME('11:34:22.00','HH:MI:SS.FF'),
    TO_POSIXTIME('11:34:22.7','HH:MI:SS.FF'),
    TO_POSIXTIME('11:34:22.7000000','HH:MI:SS.FF')
```

秒の小数部は指定どおりに渡され、埋め込みや切り捨ては行われません。

## 関連項目

- ・ SQL コマンド : [CREATE TABLE](#)、[ALTER TABLE](#)
- ・ SQL 関数 : [CAST](#)、[CONVERT](#)、[TO\\_CHAR](#)、[TO\\_DATE](#)、[TO\\_NUMBER](#)、[TO\\_TIMESTAMP](#)、[UNIX\\_TIMESTAMP](#)
- ・ ObjectScript 関数: [\\$ZDATETIME](#)、[\\$ZDATETIMEH](#)
- ・ ObjectScript 特殊変数: [\\$ZTIMESTAMP](#)

# TO\_TIMESTAMP (SQL)

フォーマットされた文字列をタイムスタンプに変換する日付関数です。

## 構文

```
TO_TIMESTAMP(dateString,format)
TO_TIMESTAMP(dateString)
```

## 説明

TO\_TIMESTAMP 関数は、さまざまな形式の日付/時刻文字列を標準の InterSystems IRIS® タイムスタンプ表現に変換します。返されるタイムスタンプは、TIMESTAMP データ型で、先頭にはゼロが付加され、24時間形式の時刻となります。

```
yyyy-mm-dd hh:mi:ss
```

返されるタイムスタンプには、既定で、先頭のゼロが含まれ、24時間形式の時刻が使用されます。

TO\_TIMESTAMP では、秒の小数部、年の 2 桁から 4 桁への変換、および時刻の 12 時間形式から 24 時間形式への変換をサポートしています。うるう年の検証も含めて、日付および時刻要素値の範囲の検証機能があります。範囲検証違反がある場合は、SQLCODE -400 エラーが生成されます。

TO\_TIMESTAMP は、ODBC 形式で標準のタイムスタンプを返します。エンコードされた 64 ビットのタイムスタンプを返すには、代わりに [TO\\_POSIXTIME](#) を使用します。変換を行うその他の SQL 関数の詳細は、"[関連する SQL 関数](#)" を参照してください。

- TO\_TIMESTAMP(dateString,format) は、指定された形式文字列を使用して日付文字列を変換します。dateString の日付と時刻要素は、format の対応する位置にある形式要素と互換性がある必要があります。

以下の文は、前の構文の日付を指定する文字列を異なる形式に変換します。

### SQL

```
SELECT TO_TIMESTAMP('June 29, 2022 12:34 PM', 'MONTH DD, YYYY')
```

例：

- 日付文字列から複数のタイムスタンプ形式への変換
- 既定のタイムスタンプ列値の設定
- 2 桁の年の変換 (RR および RRRR 形式)
- 年間通算日の変換 (DDD 形式)

- TO\_TIMESTAMP(dateString) は、既定の形式 DDMONYYYYDD :MON :YYYY を使用して日付文字列を変換します。dateString 引数は、この形式と互換性がある必要があります。時刻を省略したり、時刻の一部のみを指定した場合、TO\_TIMESTAMP は時刻を 00:00:00 として返します。

以下の文は、日付文字列 29 Jun 2022 をタイムスタンプ 2022-06-29 12:34:00 に変換します。

### SQL

```
SELECT TO_TIMESTAMP('29 Jun 2022 12:34')
```

例：日付文字列から複数のタイムスタンプ形式への変換

## 引数

### dateString

dateString 引数は、タイムスタンプに変換される日付文字列を指定する文字列式です。dateString には、日付値、時刻値、またはその両方を含めることができます。

dateString の各文字は、以下の例外を除いて **format** 文字列に対応している必要があります。

- 区切り文字のない dateString の場合を除き、先頭のゼロは指定しても省略してもかまいません。
- 年は 2 桁または 4 桁で指定できます。
- 月の省略形 (format MON) は、そのロケールでの月の省略形と一致している必要があります。ロケールによっては、月の省略形は、月の名前の先頭の連続文字でない場合があります。月の省略形では、大文字と小文字は区別されません。
- 月の名前 (format MONTH) は、完全な月名として指定する必要があります。ただし、TO\_TIMESTAMP では、完全な月名が MONTH の format と一致する必要はありません。これは完全な月名の最初の文字を受け取り、月のリストからその先頭の文字シーケンスに対応する最初の月を選択します。そのため、英語では、“J” = “January”、“Ju” = “June”、“Jul” = “July” となります。指定されるすべての文字は、完全な月名の連続文字と一致する必要があります。完全な月名を逸脱した文字はチェックされません。例えば、“Fe”、“Febru”、および “FebruaryLeap” はすべて有効な値ですが、“Febs” は無効な値です。月の名前では、大文字と小文字は区別されません。
- 時刻値は、該当するロケール用に定義されている時刻区切り文字を使用して指定できます。出力タイムスタンプは必ず、ODBC 標準時刻区切り文字のコロン (:) とピリオド (.) で時刻値を表します。時、分、秒にはコロン (:), 秒の小数部にはピリオド (.) が使用されます。時刻要素を省略すると、既定で 00:00:00 が使用されます。既定では、秒の小数部のないタイムスタンプが返されます。

dateString でタイムスタンプのコンポーネントを省略している場合、TO\_TIMESTAMP が不足するコンポーネントを提供します。

- 時刻を指定せずに日付を指定すると、TO\_TIMESTAMP は、返される時刻値に 00:00:00 を設定します。時刻の一部のみを省略すると、TO\_TIMESTAMP は、その部分に 00 を設定します。秒の小数部はサポートされますが、明示的に指定する必要があります。
- 日付なしで時刻を指定すると、TO\_TIMESTAMP は、返される日付値を現在の年の 01-01 (1 月 1 日) に設定します。日付のみを省略すると、日付は既定で 01 となります。月も省略すると、月と日付が既定で 01-01 となります。
- 年を省略すると、既定で 00 になります。これは、format の年要素に応じて、4 桁の年に拡張されます。format でも年を省略すると、YYYY が既定で現在の年になります。

TO\_TIMESTAMP の日付は、0001 年 1 月 1 日から 9999 年 12 月 31 日までを指定できます。それより前の日付を表すには、**TO\_DATE** 関数を使用します。

### format

format 引数は、dateString の形式を指定する日付/時刻文字列です。TO\_TIMESTAMP は、format の対応する位置にある形式要素に基づいて、dateString の日付要素を変換します。format の要素は以下のルールに従う必要があります。

- 形式要素では、大文字と小文字は区別されません。
- 形式要素をどのような順序で、またはいくつ使うかについては、たいいていの場合制限はありません。
- 形式文字列は、dateString の区切り文字に一致する、英数字でない区切り文字 (スペース、スラッシュ、ハイフンなど) で要素を区切ります。これらの区切り文字は出力文字列には表示されません。出力文字列では標準のタイムスタンプ区切り文字であるハイフン (日付値)、コロン (時刻値)、および秒の小数部がある場合は必要に応じてピリオドが使用されます。この区切り文字の使用は、NLS (各国言語サポート) ロケールに対して定義されている DateSeparator に依存しません。



- 日付形式文字列 MMDDYYYY、DDMMYYYY、YYYYMMDDHHMISS、YYYYMMDDHHMI、YYYYMMDDHH、YYYYMMDD、YYYYDDMM、HHMISS、および HHMI には、区切り文字は必要ありません。不完全な日付形式 YYYYMM も使用でき、DD 値は 01 と見なされます。これらの形式では、最後の要素を除き、すべての要素で先頭のゼロを指定します。
- format の無効な形式の要素は無視されます。

以下の表は、有効な日付と時刻の形式要素を示しています。大文字の形式が示されていますが、これらの要素では大文字と小文字が区別されません。

要素	意味
DD	2 桁の日付 (01 ~ 31)。format に日付の区切り文字を記述している場合、各要素の先頭にゼロは不要です。
MM	2 桁の月数 (01 ~ 12、01 = 1 月)。format に日付の区切り文字を記述している場合、各要素の先頭にゼロは不要です。  日本語と中国語では、月数は数値とそれに続く表意文字“月”で構成されます。
MON	現在のロケールの MonthAbbr プロパティによって指定される月の略名。既定で、これは英語による月名の最初の 3 文字です。その他のロケールでは、月名の略名が 4 文字以上になる場合や、月名の最初の数文字で構成されない場合があります。ピリオド文字の使用は認められていません。大文字と小文字は区別されません。
MONTH	現在のロケールの MonthName プロパティによって指定される正式な月名。大文字と小文字は区別されません。
YYYY	0001 ~ 9999 の範囲の 4 桁の年。
YY	下 2 桁の年数。2 桁の年数の最初の 2 桁は既定で 19 になります。
RR / RRRR	2 桁の年数から 4 桁の年数への変換 詳細は、“ <a href="#">2 桁の年の変換 (RR および RRRR 形式)</a> ”を参照してください。
DDD	年間通算日。1 月 1 日からの日数詳細は、“ <a href="#">年間通算日の変換 (DDD 形式)</a> ”を参照してください。
HH	時間。午前/午後のインジケータ (AM または PM) が指定されているかどうかに応じて、1 ~ 12、または 00 ~ 23 によって指定されます。HH12 または HH24 と指定できます。
MI	分。00 ~ 59 によって指定されます。
SS	秒。00 ~ 59 によって指定されます。
FF	秒の小数部。TO_TIMESTAMP は、値の埋め込みや切り捨てを行わずに dateString で指定されたままの小数値を返します。FF を指定するには、小数点記号の文字 (.) を指定する必要があります。FF を指定しない場合、dateString で指定された秒の小数部は無視されます。

要素	意味
AM	<p>午前/午後のインジケータ。12 時間形式の時刻を指定します。</p> <p>TIMESTAMP データ型では、時刻が常に 24 時間形式で示されます。dateString は、12 時間形式でも 24 時間形式でも時刻を示すことができます。TO_TIMESTAMP では、dateString の時刻部分の最後が午前/午後のインジケータでなければ、24 時間形式と見なします。以下はその例です。</p> <p>DD MON YYYY HH:MI:SS.FF P.M.</p> <p>この形式では、2:23pm、2:23:54.6pm、2:23:54 pm、2:23:54 p.m.、2:23:54 (午前と見なされます) などの、dateString で指定される 12 時間形式の値をサポートしています。午前/午後のインジケータでピリオドを指定する場合は、このインジケータと時刻を 1 つ以上のスペースで区切る必要があります。</p>
PM	
A.M.	
P.M.	

TO\_TIMESTAMP format には、D (曜日番号)、DY (曜日の省略形)、または DAY (曜日名) 要素を含めることもできます。ただし、これらの形式要素は、検証されることや戻り値の決定に使用されることはありません。これらの format 要素の詳細は、“[TO\\_CHAR](#)” を参照してください。

format を省略すると、TO\_TIMESTAMP は既定の形式 DD MON YYYY HH:MI:SS を使用して日付文字列を解析します。例えば、'01 Feb 3456 07:08:09' のようになります。

## 例

### 日付文字列から複数のタイムスタンプ形式への変換

以下の文は、さまざまな形式で日付文字列を指定します。最初の例は既定の形式を使用し、それ以外は TO\_TIMESTAMP が日付文字列を解析するために使用するフォーマット引数を指定します。TO\_TIMESTAMP は、これらすべての日付文字列をタイムスタンプ 2022-06-29 00:00:00 に変換します。

#### SQL

```
SELECT
  TO_TIMESTAMP('29 JUN 2022'),
  TO_TIMESTAMP('2022 Jun 29','YYYY MON DD'),
  TO_TIMESTAMP('JUNE 29, 2022','month dd, YYYY'),
  TO_TIMESTAMP('2022**06**29','YYYY**MM**DD'),
  TO_TIMESTAMP('06/29/2022','MM/DD/YYYY'),
  TO_TIMESTAMP('29/6/2022','DD/MM/YYYY')
```

以下の文は、YYYYMM 日付形式を指定します。これは要素セパレータを必要としません。TO\_TIMESTAMP は、欠落している日付値と時刻値を補い、タイムスタンプ 2022-06-01 00:00:00 を返します。

#### SQL

```
SELECT TO_TIMESTAMP('202206','YYYYMM')
```

以下の文は、HH:MI:SS.FF 時刻形式のみを指定します。TO\_TIMESTAMP は、欠落している日付値を補い、どのような場合でも YYYY-01-01 (YYYY は現在の年) の日付値を返します。時刻値は、日付文字列で指定された秒の小数部によって異なります。TO\_TIMESTAMP は、秒の小数部を指定どおりに渡し、埋め込みや切り捨ては行いません。

#### SQL

```
SELECT TO_TIMESTAMP('11:34','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.00','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.7','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.700000','HH:MI:SS.FF')
```



以下の文では、秒の小数部が含まれる時刻形式を指定するための他の方法を示しています。これら 3 つの TO\_TIMESTAMP の呼び出しはすべて、時刻部分の値が 11:34:22.9678 である ODBC 形式のタイムスタンプを返します。最初の 2 つの呼び出しでは、省略された日付部分は、既定で現在の年の 1 月 1 日になります。3 番目の呼び出しは、日付部分を指定します。

## SQL

```
SELECT TO_TIMESTAMP('113422.9678','HHMISS.FF'),
       TO_TIMESTAMP('9678.113422','FF.HHMISS'),
       TO_TIMESTAMP('9678.20220629113422','FF.YYYYMMDDHHMISS')
```

## 既定のタイムスタンプ列値の設定

TO\_TIMESTAMP は、テーブルの列に既定のタイムスタンプ値を指定できます。例えば、以下の文は、TIMESTAMP のタイプの列である ReviewDate の既定値を受け入れるテーブルを作成します。

```
CREATE TABLE Sample.MyEmpReviews (
  EmpNum INTEGER UNIQUE NOT NULL,
  ReviewDate TIMESTAMP DEFAULT TO_TIMESTAMP(365,'DDD'))
```

ReviewDate の値を指定せずに行を挿入すると、ReviewDate は、現在の年の 365 日目の既定のタイムスタンプに設定されます。

## SQL

```
INSERT INTO Sample.MyEmpReviews (EmpNum) VALUES (1)
```

TO\_TIMESTAMP を使用すると、CREATE TABLE と ALTER TABLE ADD COLUMN の両方の文に既定の列値を設定できます。これらの既定値を設定する場合、dateString はリテラル値である必要があります。

## 2 桁の年の変換 (RR および RRRR 形式)

YY 形式では、2 桁の年の値に 19 が付加されて 4 桁の値に変換されます。例えば、07 は 1907 になり、93 は 1993 になります。RR 形式および RRRR 形式では、2 桁の年から 4 桁の年への変換をより柔軟に行います。

RR 形式の変換は現在の年に基づきます。

- ・ 現在の年が、ある世紀の前半にある場合、以下のようになります。
  - 00 ～ 49 までの 2 桁の年は現在の世紀の 4 桁の年に展開されます。
  - 50 ～ 99 までの 2 桁の年は前の世紀の 4 桁の年に展開されます。

以下の文は、現在の年が 2000 から 2050 までの間の場合に TO\_TIMESTAMP が返すタイムスタンプの表示形式を示しています。

## SQL

```
SELECT
  TO_TIMESTAMP('29 September 00','DD MONTH RR'), -- 2000-09-29 00:00:00
  TO_TIMESTAMP('29 September 18','DD MONTH RR'), -- 2018-09-29 00:00:00
  TO_TIMESTAMP('29 September 49','DD MONTH RR'), -- 2049-09-29 00:00:00
  TO_TIMESTAMP('29 September 50','DD MONTH RR'), -- 1950-09-29 00:00:00
  TO_TIMESTAMP('29 September 77','DD MONTH RR')  -- 1977-09-29 00:00:00
```

- ・ 現在の年が、ある世紀の後半にある場合、すべての 2 桁の年が現在の世紀の 4 桁の年に展開されます。

以下の文は、現在の年が 2050 から 2099 までの間の場合に TO\_TIMESTAMP が返す日付の表示形式を示しています。

## SQL

```
SELECT
  TO_TIMESTAMP('29 September 00','DD MONTH RR'), -- 2000-09-29 00:00:00
  TO_TIMESTAMP('29 September 21','DD MONTH RR'), -- 2021-09-29 00:00:00
  TO_TIMESTAMP('29 September 49','DD MONTH RR'), -- 2049-09-29 00:00:00
  TO_TIMESTAMP('29 September 50','DD MONTH RR'), -- 1950-09-29 00:00:00
  TO_TIMESTAMP('29 September 77','DD MONTH RR')  -- 1977-09-29 00:00:00
```

RRRR 形式を使用することにより、2 桁と 4 桁の年を混合して入力することができます。TO\_TIMESTAMP は、4 桁の年を変更せずに渡します。TO\_TIMESTAMP は、この例で前述の RR 形式アルゴリズムを使用して、2 桁の年を 4 桁の年に変換します。

以下の文は、現在の年が 2000 から 2050 までの間の場合に TO\_TIMESTAMP が返す日付の表示形式を示しています。

## SQL

```
SELECT
  TO_TIMESTAMP('29 September 2021','DD MONTH RRRR'), -- 2021-09-29 00:00:00
  TO_TIMESTAMP('29 September 21','DD MONTH RRRR'),  -- 2021-09-29 00:00:00
  TO_TIMESTAMP('29 September 1949','DD MONTH RRRR'), -- 1949-09-29 00:00:00
  TO_TIMESTAMP('29 September 49','DD MONTH RRRR'),  -- 2049-09-29 00:00:00
  TO_TIMESTAMP('29 September 1950','DD MONTH RRRR'), -- 1950-09-29 00:00:00
  TO_TIMESTAMP('29 September 50','DD MONTH RRRR')   -- 1950-09-29 00:00:00
```

## 年間通算日の変換 (DDD 形式)

DDD 形式を使用して、年間通算日 (1 月 1 日から経過した日数) を実際のタイムスタンプに変換できます。この変換を実行するには、以下の点に留意してください。

- **format** 引数の日付部分には、DDD 形式要素を含める必要があります。また、オプションで YYYY、YY、RR、RRRR などの年形式を含めることができます。これらの要素は任意の順序で指定できますが、要素の間に区切り文字を追加する必要があります。年を省略した場合、TO\_TIMESTAMP によって現在の年が既定値として使用されます。
- **dateString** 引数には、対応する day および year の値を含める必要があります。
  - day は、1 から 365 までの整数です (year がうるう年の場合は 366)。
  - year は、0001 から 9999 の範囲の年です。

以下の文は、2022 年の 60 日目を返します。

## SQL

```
SELECT TO_TIMESTAMP('2022:60','YYYY:DDD') --2022-03-01 00:00:00
```

TO\_TIMESTAMP は月の要素を変更せずに渡します。形式文字列に DD と DDD 要素の両方が含まれる場合、TO\_TIMESTAMP は DDD 要素を処理し、DD 要素を無視します。例えば、以下の文は、2020 年 12 月 31 日ではなく、2020 年 2 月 29 日 (2020 年の 60 日目) のタイムスタンプを返します。

## SQL

```
SELECT TO_TIMESTAMP('2020-12-31-60','YYYY-MM-DD-DDD')
```

TO\_TIMESTAMP は、年間通算日そのものではなく、年間通算日を含むタイムスタンプ式を返します。この日付の値を返すには、TO\_CHAR を使用します。

## 詳細

### 関連する SQL 関数

- ・ `TO_TIMESTAMP` は、フォーマットされた日付/時刻文字列を標準のタイムスタンプに変換します。
- ・ `TO_CHAR` は、逆の操作を実行します。これは標準のタイムスタンプをフォーマットされた日付/時刻文字列に変換します。
- ・ `TO_DATE` は、フォーマットされた日付文字列を日付整数に変換します。
- ・ `CAST` および `CONVERT` は、TIMESTAMP データ型の変換を実行します。

## 代替案

ObjectScript コードで同等のタイムスタンプ変換を実行するには、`TOTIMESTAMP()` メソッドを使用します。

```
$SYSTEM.SQL.Functions.TOTIMESTAMP(date_string,format)
```

## 関連項目

- ・ SQL コマンド : [CREATE TABLE](#)、[ALTER TABLE](#)
- ・ SQL 関数 : [CAST](#)、[CONVERT](#)、[TO\\_CHAR](#)、[TO\\_DATE](#)、[TO\\_NUMBER](#)、[TO\\_POSIXTIME](#)
- ・ ObjectScript 関数 : [\\$ZDATE](#) [\\$ZDATEH](#)

## \$TRANSLATE (SQL)

文字と文字の置換を行う文字列関数です。

### 構文

```
$TRANSLATE(string, identifier[, associator])
```

### 概要

\$TRANSLATE 関数は、返り値の文字列内の文字と文字の置き換えを行います。string 引数の処理を一度に 1 文字行います。string にある各文字と identifier 引数にある各文字とを比較します。\$TRANSLATE が一致するものを見つけると、文字の位置を書きとめます。

- ・ 引数が 2 つの形式の \$TRANSLATE は、identifier 引数にある文字のすべてのインスタンスを出力文字列から取り除きます。
- ・ 引数が 3 つの形式の \$TRANSLATE は、string 内で見つかった各 identifier 文字のすべてのインスタンスを、位置的に対応する associator 文字と置き換えます。置換は、文字列単位ではなく文字単位で行われます。identifier 引数に含まれる文字が associator 引数よりも多い場合、identifier 引数にある余分な文字は出力文字列から削除されます。identifier 引数に含まれる文字が associator 引数よりも少ない場合、associator 引数にある余分な文字は無視されます。

\$TRANSLATE は、大文字と小文字を区別します。

\$TRANSLATE は、NULL を文字と置き換えるためには使用できません。

指定する引数が少なすぎると、SQLCODE -380 が発行されます。指定する引数が多すぎると、SQLCODE -381 が発行されます。

### \$TRANSLATE および REPLACE

\$TRANSLATE は文字と文字をマッチングして置換します。[REPLACE](#) は文字列と文字列をマッチングして置換します。REPLACE では、1 つ以上の文字から成る指定された単一の部分文字列を別の部分文字列に置換することや、指定された部分文字列の複数のインスタンスを削除することができます。\$TRANSLATE は、複数の指定された文字に対応する指定された置換文字に置換できます。

既定では、どちらの関数も大文字小文字を区別して、string の先頭から開始し、一致するすべてのインスタンスを置換します。REPLACE には、これらの既定を変更するために使用できる引数があります。

### 引数

#### string

ターゲット文字列。フィールド名、リテラル、ホスト変数、または SQL 式のいずれかにできます。

#### identifier

string 内で検索する文字。文字列または数値のリテラル、ホスト変数、または SQL 式のいずれかにできます。

#### associator

オプションの引数です。identifier でそれぞれの文字に対応する置き換え文字。文字列または数値のリテラル、ホスト変数、または SQL 式のいずれかにできます。

## 例

以下の例では、引数が 2 つの \$TRANSLATE によって、句読点 (コンマ、スペース、ピリオド、アポストロフィ、ハイフン) を削除することで Name 値を変更し、アルファベット文字のみから成る名前を返します。identifier では、アポストロフィが二重になることに注意してください。これは、アポストロフィを文字列区切りではなくリテラル文字としてエスケープするためです。

### SQL

```
SELECT TOP 20 Name,$TRANSLATE(Name,',',' -') AS AlphaName
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

以下の例では、引数が 3 つの \$TRANSLATE によって、コンマおよびスペースをキャレット (^) 文字に置き換えることで Name 値を変更し、3 つの部分 (姓、名、ミドルネームのイニシャル) に区切られた名前を返します。associator では、identifier 内の文字数と同数の “” を指定する必要があることに注意してください。

### SQL

```
SELECT TOP 20 Name,$TRANSLATE(Name,', ',' ^') AS PiecesNamePunc
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

以下の例では、引数が 3 つの \$TRANSLATE によって、コンマおよびスペースをキャレット (^) 文字に置き換え (identifier および associator 内で指定されている)、ピリオド、アポストロフィ、およびハイフンを削除することで (identifier 内では指定されており、associator からは省略されている)、Name 値を変更します。

### SQL

```
SELECT TOP 20 Name,$TRANSLATE(Name,',',' -',' ^') AS PiecesNameNoPunc
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

## 関連項目

- ・ [REPLACE 関数](#)
- ・ [STUFF 関数](#)
- ・ [文字列操作](#)

## TRIM (SQL)

指定された先頭や末尾の文字列を削除する文字列を返す、文字列関数です。

### 構文

```
TRIM([end_keyword] [characters FROM] string-expression)
```

### 概要

TRIM は、指定値の最初および/または最後から指定された文字を削除します。文字の削除では、既定で大文字と小文字が区別されます。いずれかの端からの文字の削除は、characters で指定されていない文字が出現すると停止します。既定では、string-expression の両端から空白スペースを削除します。

TRIM は、削除される入力式のデータ型に関係なく、常に VARCHAR データ型を返します。

TRIM またはその他の SQL 関数に提供される前に、数値から先頭のゼロが自動的に削除されることに注意してください。先頭のゼロを保持するには、数値を文字列として指定する必要があります。

オプションの end\_keyword 引数に利用可能な値は、以下のとおりです。

LEADING	characters 内の文字が string-expression の先頭から削除されることを指定するキーワード。
TRAILING	characters 内の文字が string-expression の末尾から削除されることを指定するキーワード。
BOTH	characters 内の文字が string-expression の先頭と末尾の両方から削除されることを指定するキーワード。BOTH は既定で、end_keyword が指定されていない場合に使用されます。

代わりに、[LTRIM](#) を使用して先頭の空白を削除、あるいは [RTRIM](#) を使用して末尾の空白を削除することもできます。

文字列の先頭または末尾に空白またはその他の文字でパディングするには、[LPAD](#) または [RPAD](#) を使用します。

LENGTH 関数を使用すると、空白スペースが文字列から削除されているか、文字列に追加されているかを判断できます。

### 削除する文字

- すべての文字：TRIM は、characters が string-expression 内のすべての文字を含む場合、空の文字列を返します。
- 一重引用符：TRIM は、一重引用符が characters と string-expression の両方で二重になる場合、これらを削除できます。したがって、TRIM(BOTH 'a' 'b' FROM 'bb' 'ba' 'acaaa' '') は 'c' を返します。
- 空白スペース：TRIM は、characters が省略される場合、空白スペースを string-expression から削除します。characters が指定される場合、これは空白スペースを削除する空白スペース文字を含む必要があります。
- %List：string-expression が %List である場合、TRIM は、先頭の文字ではなく、末尾の文字のみを削除できます。これは、%List には先頭のエンコーディング文字が含まれるためです。TRIM を先頭文字に適用するには、%List を文字列に変換する必要があります。
- NULL：いずれかの文字列式が NULL の場合、TRIM は NULL を返します。

### 引数

#### end\_keyword

string-expression のどちら側の端部を削除するかを指定するキーワード (オプション)。利用可能な値は LEADING、TRAILING、BOTH です。既定値は BOTH です。

## characters

string-expression から削除する文字列を指定する文字列式 (オプション)。ここで指定されていない文字が出現するまで、指定された末尾から指定された文字のすべてのインスタンスが削除されます。したがって、`TRIM(BOTH 'ab' FROM 'bbbaacaaa')` は 'c' を返します。この例では、`BOTH` キーワードはオプションです。

characters を指定しない場合、`TRIM` は空白スペースを削除します。

characters を指定する場合、`FROM` キーワードが必要です。end\_keyword が指定され、characters が指定されていない場合、`FROM` キーワードが許可されます (ただし必須ではない)。これらの引数のいずれも指定されていない場合、`FROM` キーワードは許可されません。

## string-expression

削除される文字列式。string-expression は、列の名前、文字列リテラル、または他の関数の結果となります。基本データ型は、(CHAR や VARCHAR2 など) さまざまな文字タイプとして表示されます。

characters と end\_keyword の両方が省略される場合、`FROM` キーワードは省略されます。

## 例

以下の例は、end\_keyword および characters の既定を使用して、"abc" から先頭と末尾の空白を削除します。select-items は ' ' を文字列の両端に連結して、空白を示します。

### SQL

```
SELECT '^'||'   abc   '||'^' AS UnTrimmed,'^'||TRIM('   abc   ')||'^' AS Trimmed
```

これは、文字列 ^ abc ^ と ^abc^ を返します。

以下の例は、string-expression から先頭の空白スペースを削除するための有効なすべての構文です。

```
SELECT TRIM(LEADING '   abc   '),TRIM(LEADING FROM '   def   '),TRIM(LEADING ' ' FROM '   ghi   ')
```

以下の例は、文字列 "xxxabcxxx" の先頭から文字 "x" を削除して、"abcxxx" にします。

### SQL

```
SELECT TRIM(LEADING 'x' FROM 'xxxabcxxx') AS Trimmed
```

以下の例は両方とも、文字列 "xxxabcxxx" の先頭と末尾から文字 "x" を削除し、"abc" にします。最初の例は、`BOTH` を指定し、2 番目の例は既定値として `BOTH` を取ります。

### SQL

```
SELECT TRIM(BOTH 'x' FROM 'xxxabcxxx') AS Trimmed
```

### SQL

```
SELECT TRIM('x' FROM 'xxxabcxxx') AS Trimmed
```

以下の例は、"abczzzxyyyy" という文字列から末尾の文字列 "xyz" の全インスタンスを削除し、"abc" という結果を出します。

### SQL

```
SELECT TRIM(TRAILING 'xyz' FROM 'abczzzxyyyy') AS Trimmed
```

以下の例では、FirstName のすべての文字を削除して FullName を削除し、前に空白スペースが付いた姓を返します。例えば、FirstName/Fullname 'Fred'/'Fred Rogers' は 'Rogers' を返します。この例では、FirstName 'Annie' は

‘Ann’、‘Anne’、‘Ani’、‘Ain’、‘Annee’、‘Annie’ を LastName から削除しますが、‘Anna’ は完全には削除しません。TRIM は大文字と小文字を区別するためです。‘A’ のみが削除され、‘a’ は削除されません。

## SQL

```
SELECT TRIM(LEADING FirstName FROM FullName) FROM Sample.Person
```

以下の例は、FavoriteColors 値から先頭の文字 “B” または “R” を削除します。TRIM を先頭文字に適用するには、リストを文字列に変換する必要があります。

## SQL

```
SELECT TOP 15 Name, FavoriteColors,
    TRIM(LEADING 'BR' FROM $LISTTOSTRING(FavoriteColors)) AS Trimmed
FROM Sample.Person WHERE FavoriteColors IS NOT NULL
```

## 関連項目

- ・ SQL 関数 : [LTRIM](#)、[RTRIM](#)、[LPAD](#)、[RPAD](#)
- ・ ObjectScript 関数 : [\\$ZSTRIP](#)



# TRUNCATE (SQL)

指定された桁で数値を切り捨てるスカラ数値関数。

## 構文

```
{fn TRUNCATE(numeric-expr,scale)}
```

## 概要

TRUNCATE は、*numeric-expr* を小数点から *scale* 桁目で切り捨てます。数値の丸めやゼロの埋め込みはしません。TRUNCATE 処理の前に、先頭と末尾のゼロは削除されます。

- *scale* が正の数である場合、切り捨ては小数点から右に数えたその桁数で実行されます。*scale* が小数桁数以上の数である場合、切り捨てやゼロの埋め込みは実行されません。
- *scale* が 0 の場合、数値は整数に切り捨てられます。つまり、切り捨ては小数点右側の 0 桁目で実行され、小数点以下と小数点そのものが切り捨てられます。
- *scale* が負の数である場合、切り捨ては小数点から左に数えたその桁数で実行されます。*scale* が数の整数桁数以上である場合は、ゼロが返されます。
- *numeric-expr* がゼロ (ただし 00.00 や -0 などのように表されている) 場合、TRUNCATE は *scale* の値とは無関係に小数桁のない 0 (ゼロ) を返します。
- *numeric-expr* または *scale* が NULL の場合、TRUNCATE は NULL を返します。

TRUNCATE は {} 括弧構文による ODBC スカラ関数としてのみ使用できます。

ROUND を使用して、数値で同様のトランケーション処理を実行できます。TRIM を使用して、文字列で同様のトランケーション処理を実行できます。

## TRUNCATE、ROUND、および \$JUSTIFY

数値関数 TRUNCATE と ROUND の動作は似ています。両方とも小数桁または整数桁の有効桁数を減らすために使用できます。ROUND では、丸め (既定) または切り捨てを指定できます。TRUNCATE は丸めを行いません。ROUND は *numeric-expr* と同じデータ型を返します。TRUNCATE は *numeric-expr* をデータ型 NUMERIC として返します。ただしこれは、*numeric-expr* がデータ型 DOUBLE でない場合に限りです。DOUBLE である場合は、データ型 DOUBLE を返します。

TRUNCATE では、指定された小数桁数への切り捨てを行います。切り捨てによって末尾にゼロが生じた場合、これらの末尾のゼロは保持されます。ただし、*scale* が *numeric-expr* のキャノニック形式の小数桁数より大きい場合、TRUNCATE はゼロパディングを行いません。

ROUND では、指定された小数桁数への丸め (または切り捨て) を行いますが、その戻り値は常に正規化され、末尾のゼロが削除されます。例えば、ROUND(10.004, 2) は 10.00 ではなく 10 を返します。

固定の小数桁数への丸めが重要な場合、例えば金額を表す場合などは、\$JUSTIFY を使用します。\$JUSTIFY は、丸め処理の後に指定された数の末尾のゼロを返します。丸める桁数が小数桁数より大きい場合、\$JUSTIFY はゼロパディングを行います。また \$JUSTIFY では、DecimalSeparator 文字が数値列内で揃うように数値が右寄せされます。\$JUSTIFY は切り捨てを行いません。

## 引数

### *numeric-expr*

切り捨てられる数字。数値または数値式。

## scale

小数点からカウントして、どこまで切り捨てるかの桁数を指定する整数に評価される式。ゼロ、正の整数、または負の整数を指定できます。scale が小数の場合、最も近い整数に丸められます。

TRUNCATE は、DOUBLE、INTEGER、NUMERIC のデータ型を返します。

- ・ numeric-expr が DOUBLE 型の場合、TRUNCATE は DOUBLE を返します。
- ・ numeric-expr が INTEGER 型で scale が 0 以下の場合、TRUNCATE は INTEGER を返します。
- ・ numeric-expr が NUMERIC 型か INTEGER 型で、scale が 0 より大きい場合、TRUNCATE は NUMERIC を返します。

## 例

次の 2 つの例では、数値を小数第 2 位で切り捨てます。scale は整数として指定されています。

### SQL

```
SELECT {fn TRUNCATE(654.321888,2)}
```

### SQL

```
SELECT {fn TRUNCATE(654.321888,2)}
```

両方の例とも、654.32 を返します (小数第 2 位で切り捨て)。

以下の例は、小数桁数より大きな scale を指定しています。

### SQL

```
SELECT {fn TRUNCATE(654.321000,9)}
```

この場合は 654.321 が返されます (InterSystems IRIS が切り捨て処理を行う前に末尾のゼロを削除し、切り捨ててもゼロの埋め込みも行われません)。

次の例では、0 の scale を指定しています。

### SQL

```
SELECT {fn TRUNCATE(654.321888,0)}
```

これは 654 を返します (小数点以下すべてと小数点が切り捨てられます)。

次の例では、負の scale を指定しています。

### SQL

```
SELECT {fn TRUNCATE(654.321888,-2)}
```

これは 600 を返します (整数桁が 2 桁切り捨てられ、0 で置き換わります。丸めは実行されません)。

次の例では、負の scale を数値の整数桁数と同じに指定しています。

### SQL

```
SELECT {fn TRUNCATE(654.321888,-3)}
```

これは 0 を返します。

## 関連項目

- ・ SQL 関数: [\\$JUSTIFY](#)、[ROUND](#)、[RTRIM](#)、[TRIM](#)、
- ・ ObjectScript 関数: [\\$NORMALIZE](#)

## %TRUNCATE (SQL)

指定された長さになるまで文字列を切り捨ててから EXACT 照合に適用する照合関数です。

### 構文

```
%TRUNCATE(expression[, length])
```

### 説明

%TRUNCATE は、指定された長さになるまで *expression* を切り捨ててから EXACT 照合シーケンスで返します。

EXACT 照合は、まず純粋な数値 (x=+x としての値) を数値順で並べ、次に他のすべての文字列は文字列照合シーケンス順で並べます。EXACT 文字列照合シーケンスは、ANSI 標準の ASCII 照合シーケンスと同じものです。数字は大文字のアルファベット文字の前に照合され、大文字のアルファベット文字は小文字のアルファベット文字の前に照合されます。句読点文字は、そのシーケンスのそれぞれの場所で照合されます。

%TRUNCATE は NULL を変更せずに渡します。

%TRUNCATE は、InterSystems SQL の拡張機能であり、SQL 検索クエリ用として使用するものです。

この関数は、ObjectScript から TRUNCATE() メソッド・コールを使用して呼び出すこともできます。

#### ObjectScript

```
WRITE $SYSTEM.SQL.Functions.TRUNCATE("This long string",9)
```

### 引数

#### *expression*

列名、文字列リテラル、または他の関数の結果となる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR2 など) とすることができます。expression にはサブクエリを指定できます。

#### *length*

切り捨てを行った結果の長さを示す引数 (オプション)。整数として指定します。expression の先頭から length の文字数までを返します。length を省略すると、%TRUNCATE 照合は %EXACT 照合と同じになります。((length)) のように二重括弧で length を囲むと、[リテラル置換を抑制](#)できます。

### 例

以下の例は %TRUNCATE を使用して、Name 値の最初の 4 文字を返します。

#### SQL

```
SELECT TOP 5 Name,%TRUNCATE(Name,4) AS ShortName
FROM Sample.Person
```

以下の例では、%TRUNCATE をサブクエリに適用しています。

#### SQL

```
SELECT TOP 5 Name, %TRUNCATE((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

以下の例では、GROUP BY 節で %TRUNCATE を使用して、各文字で始まる名前の数を返すアルファベット・リストを作成しています。

## SQL

```
SELECT Name AS FirstLetter,COUNT(Name) AS NameCount
FROM Sample.Person GROUP BY %TRUNCATE(Name,1) ORDER BY Name
```

以下の 2 つの例は、%TRUNCATE で EXACT 照合がどのように行われるかを示しています。最初の例にある ORDER BY では、Home\_Street が切り捨てられて 2 文字になります。番地の最初の 2 文字はほぼ必ず数値になるので、Home\_Street フィールドは、最初の 2 つの数字の数値順に並べられます。

## SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %TRUNCATE(Home_Street,2)
```

2 番目の例にある ORDER BY では、Home\_Street が切り捨てられて 4 文字になります。4 番目の文字が (空白など) 数字でない番地があるので、先頭が 4 文字以上の数字である Home\_Street 値は、最初に数値順に並べられます。数値でない文字列が最初の 4 文字にある Home\_Street 値は、文字列順に並べられます。

## SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %TRUNCATE(Home_Street,4)
```

## 関連項目

- ・ [CREATE TABLE](#)
- ・ [%STARTSWITH](#) 述語
- ・ [%EXACT](#) 照合関数
- ・ [%SQLSTRING](#) 照合関数
- ・ [%TRUNCATE](#) 照合関数
- ・ [照合](#)

## \$TSQL\_NEWID (SQL)

---

グローバルに一意な ID を返す関数です。

### 構文

```
$TSQL_NEWID( )
```

### 説明

\$TSQL\_NEWID は、[グローバルに一意な ID \(GUID\)](#) を返します。GUID は、不定期に接続されるシステム上のデータベースを同期するのに使用されます。GUID は 36 文字の文字列で、32 文字の 16 進数値が、ハイフンで 5 つのグループに分割されて構成されています。このデータ型は %Library.UniqueIdentifier です。

\$TSQL\_NEWID は、InterSystems SQL で指定することで、[InterSystems Transact-SQL \(TSQL\)](#) がサポートされます。対応する TSQL 関数は [NEWID](#) です。

\$TSQL\_NEWID 関数は引数を取りません。引数の括弧は必須です。

%Library.GUID 抽象クラスは、グローバルに一意な ID をサポートします。これには、グローバルに一意な ID をクラスに割り当てるのに使用できる AssignGUID() メソッドが含まれます。GUID 値を生成するには、%SYSTEM.Util.CreateGUID() メソッドを使用します。

### 例

以下の例では、GUID を返しています。

#### SQL

```
SELECT $TSQL_NEWID( )
```

### 関連項目

- ・ TSQL : [NEWID](#) 関数

# UCASE (SQL)

文字列内のすべての小文字を大文字に変換するケース変換関数です。

## 構文

```
UCASE(string-expression)  
{fn UCASE(string-expression)}
```

## 概要

UCASE は、表示目的で小文字を大文字に変換します。これは、非アルファベット文字に影響を与えません。数字、句読点および先頭と末尾の空白スペースを変更しません。

UCASE は、{} 括弧構文による ODBC スカラ関数、または SQL 汎用関数として使用できる点に注意してください。

UCASE は、数値を文字列として解釈する変換を強制的に実行しません。InterSystems SQL は数値から先頭と末尾のゼロを削除します。文字列として指定された数値は、先頭と末尾のゼロを保持します。

UCASE は、[照合](#)に影響しません。%SQLUPPER 関数は、大文字と小文字を区別しない照合に対してデータ値を変換するために SQL で優先的に使用される方法です。照合のケース変換の詳細は、“[%SQLUPPER](#)”を参照してください。

この関数は、ObjectScript から UPPER() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.UPPER(expression)
```

## 引数

### *string-expression*

文字列式。その中の文字が大文字に変換されます。式は列の名前や文字列リテラル、または他のスカラ関数の結果を指定できます。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

## 例

以下の例は、各人の名前を大文字で返します。

### SQL

```
SELECT Name, {fn UCASE(Name)} AS CapName  
FROM Sample.Person
```

また、UCASE は、ギリシャ文字を小文字から大文字に変換する以下の例で示すように、Unicode (非 ASCII) アルファベット文字でも動作します。

### SQL

```
SELECT UCASE($CHAR(950,949,965,963))
```

## 関連項目

- SQL 関数: [LCASE](#)、[%SQLUPPER](#)、[UPPER](#)
- ObjectScript 関数: [\\$ZCONVERT](#)

## UNIX\_TIMESTAMP (SQL)

日付式を UNIX タイムスタンプに変換する日付/時刻関数です。

### 構文

```
UNIX_TIMESTAMP([date-expression])
```

### 説明

UNIX\_TIMESTAMP は、UNIX® タイムスタンプ ('1970-01-01 00:00:00' からの秒数 (および秒の小数部)) を返します。

date-expression が指定されていない場合、現在の UTC タイムスタンプが date-expression の既定値として使用されます。したがって、UNIX\_TIMESTAMP() は、システム全体の既定の 3 の精度を前提とする

UNIX\_TIMESTAMP(GETUTCDATE(3)) と同等になります。

date-expression が指定されている場合、UNIX\_TIMESTAMP は、UNIX タイムスタンプの秒数を計算して、指定されている date-expression 値を UNIX タイムスタンプに変換します。UNIX\_TIMESTAMP は、正または負の秒数を返すことができます。

UNIX\_TIMESTAMP は、その値をデータ型 %Library.Numeric として返します。秒の小数部の精度を返すことができます。date-expression が指定されていない場合、現在構成されているシステム全体の精度が使用されます。date-expression が指定されている場合、その date-expression の精度が使用されます。

### date-expression 値

オプションの date-expression は、以下の値として指定できます。

- ・ ODBC タイムスタンプ値 (データ型 %Library.TimeStamp) : YYYY-MM-DD HH:MI:SS.FFF。
- ・ PosixTime タイムスタンプ値 (データ値 %Library.PosixTime) : エンコードされた 64 ビットの符号付き整数。
- ・ \$HOROLOGY データ値 (データ型 %Library.Date) : 1840 年 12 月 31 日からの日数 (1 は 1841 年 1 月 1 日を示します)。
- ・ 秒の小数部を含む/含まない \$HOROLOGY タイムスタンプ : 64412,54736。

UNIX\_TIMESTAMP ではタイムゾーンの変換は行われません。date-expression が UTC 時刻である場合、UTC UnixTime が返されます。date-expression がローカル時刻である場合、ローカル UnixTime 値が返されます。

### 秒の小数部の精度

秒の小数部は、丸められずに、常に指定された精度に切り捨てられます。

- ・ %Library.TimeStamp データ型形式の date-expression の最大精度は 9 にできます。サポートされる実際の桁数は、date-expression precision 引数、構成されている既定の時刻精度、およびシステム機能によって決まります。構成されている既定の時刻制度よりも大きい precision が指定されている場合、追加の精度桁数は末尾の 0 として返されます。
- ・ %Library.PosixTime データ型形式の date-expression の最大精度は 6 です。POSIXTIME 値はすべて 6 桁の精度を使用して計算されます。指定されていない場合、これらの小数桁数は既定で 0 桁になります。

### 精度の構成

既定の精度は、以下の方法で構成できます。

- ・ SET OPTION で TIME\_PRECISION オプションを使用します。



- ・ システム全体の \$SYSTEM.SQL.Util.SetOption() メソッド構成オプション DefaultTimePrecision。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。これにより、[ ] が表示されます。既定値は 0 です。
- ・ 管理ポータルに進み、[システム管理]、[構成]、[SQL およびオブジェクトの設定]、[SQL] の順に選択します。[GET-DATE(), CURRENT\_TIME, CURRENT\_TIMESTAMP のデフォルトの時刻精度] の現在の設定を表示して編集します。

返される精度の小数点以下の桁数の既定値に 0 ～ 9 の整数を指定します。既定値は 0 です。実際に返される精度はプラットフォームに依存し、システムで使用可能な精度を超えた precision の桁はゼロとして返されます。

## 日付/時刻関数の比較

UNIX\_TIMESTAMP は、任意の日付からの経過秒数として表される日付と時刻を返します。

TO\_POSIXTIME は、UNIX タイムスタンプから計算される、エンコードされた 64 ビットの符号付きの値 (%PosixTime タイムスタンプ) を返します。

GETUTCDATE は、%TimeStamp (ODBC タイムスタンプ) データ型または %PosixTime (エンコードされた 64 ビットの符号付き整数) データ型の値として、ユニバーサルな (タイム・ゾーンに依存しない) 日付と時刻を返します。%PosixTime 値は、対応する UNIX タイムスタンプ値から計算されます。%PosixTime のエンコードにより、迅速にタイムスタンプの比較や計算を行うことが容易になります。%Library.PosixTime クラスには、UNIX タイムスタンプを PosixTime タイムスタンプに変換する UnixTimeToLogical() メソッド、および PosixTime タイムスタンプを UNIX タイムスタンプに変換する LogicalToUnixTime() メソッドがあります。これらのメソッドは、どちらもタイムゾーンの変換を行いません。

ObjectScript \$ZTIMESTAMP 特殊変数を使用して、ユニバーサルな (タイムゾーンに依存しない) タイムスタンプを返すことができます。

ObjectScript \$ZDATETIME 関数の dformat -2 は、InterSystems IRIS \$HOROLOGY 日付を取り、UNIX タイムスタンプを返します。\$ZDATETIMEH dformat -2 は、UNIX タイムスタンプを取り、InterSystems IRIS %HOROLOGY 日付を返します。これらの ObjectScript 関数は、ローカル時刻を UTC 時刻に変換します。UNIX\_TIMESTAMP は、ローカル時刻を UTC 時刻に変換しません。

## 引数

### date-expression

列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式 (オプション)。UNIX\_TIMESTAMP は、タイムゾーン間の変換は行いません。date-expression が省略されている場合、現在の UTC タイムスタンプが既定値として使用されます。

## 例

以下の例では、UTC UNIX タイムスタンプを返しています。最初の select-item では date-expression の既定値が使用されており、2 番目の select-item では明示的な UTC タイムスタンプが指定されています。

### SQL

```
SELECT
    UNIX_TIMESTAMP() AS DefaultUTC,
    UNIX_TIMESTAMP(GETUTCDATE(3)) AS ExplicitUTC
```

以下の例では、現在のローカル日付とローカル時刻のローカル UNIX タイムスタンプ、および UTC 日付値と UTC 時刻値の UTC UNIX タイムスタンプを返しています。最初の select-item ではローカル CURRENT\_TIMESTAMP、2 番目の select-item では \$HOROLOGY (ローカル日付とローカル時刻)、3 番目の select-item では現在の UTC 日付と UTC 時刻が指定されています。

## SQL

```
SELECT
    UNIX_TIMESTAMP(CURRENT_TIMESTAMP(2)) AS CurrTSLocal,
    UNIX_TIMESTAMP($HOROLOG) AS HorologLocal,
    UNIX_TIMESTAMP(GETUTCDATE(3)) AS ExplicitUTC
```

以下の例では、UNIX\_TIMESTAMP (ローカル時刻を変換しません) と \$ZDATETIME (ローカル時刻を変換します) を比較しています。

### ObjectScript

```
SET unixutc=$ZDATETIME($HOROLOG,-2)
SET myquery = "SELECT UNIX_TIMESTAMP($HOROLOG) AS UnixLocal,? AS UnixUTC"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(unixutc)
DO rset.%Display()
```

## 関連項目

- SQL の概念 : [データ型](#)、[日付/時刻文](#)
- SQL タイムスタンプ関数 : [CAST](#)、[CONVERT](#)、[GETDATE](#)、[GETUTCDATE](#)、[NOW](#)、[SYSDATE](#)、[TIMESTAMPADD](#)、[TIMESTAMPDIFF](#)、[TO\\_POSIXTIME](#)、[TO\\_TIMESTAMP](#)
- ObjectScript : [\\$ZDATETIME](#) 関数と [\\$ZDATETIMEH](#) 関数、[\\$HOROLOG](#) 特殊変数、[\\$ZTIMESTAMP](#) 特殊変数

# UPPER (SQL)

文字列式内のすべての小文字を大文字に変換するケース変換関数です。

## 構文

```
UPPER(expression)
```

```
UPPER expression
```

## 概要

UPPER 関数は、すべてのアルファベット文字を大文字に変換します。これは、LOWER 関数とは逆の関数です。UPPER は、数字、句読点、および先頭と末尾の空白を変更しません。

UPPER は、数値を文字列として解釈する変換を強制的に実行しません。InterSystems SQL は数値から先頭と末尾のゼロを削除します。文字列として指定された数値は、先頭と末尾のゼロを保持します。

この関数は、ObjectScript から UPPER() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.UPPER(expression)
```

UPPER は、照合ではなく、アルファベットのケース変換を実行する標準関数です。大文字の照合には、[%SQLUPPER](#) を使用します。これは、数値、NULL 値、および空文字列の照合に優れています。

## 引数

### *expression*

列名、文字列リテラル、または他の関数の結果となる文字列式。基本となるデータ型は、任意の文字タイプ (CHAR や VARCHAR など) とすることができます。

## 例

以下の例は、名前の大文字フォームが “JO” で始まる場合を選択して、すべての名前を返します。

### SQL

```
SELECT Name
FROM Sample.Person
WHERE UPPER(Name) %STARTSWITH UPPER('JO')
```

以下の例は、名前が “JO” で始まる場合を選択して、すべての大文字表記の名前を返します。

### SQL

```
SELECT UPPER(Name) AS CapName
FROM Sample.Person
WHERE Name %STARTSWITH UPPER('JO')
```

以下の例は、小文字のギリシャ文字デルタを大文字に変換します。この例では、キーワードと引数の区切り文字として括弧ではなくスペースを使用する UPPER 構文を使用しています。

### SQL

```
SELECT UPPER {fn CHAR(948)},{fn CHAR(948)}
FROM Sample.Person
```

## 関連項目

- ・ [%SQLUPPER](#) 照合関数

- ・ [%STARTSWITH](#) 述語条件
- ・ [LOWER](#) 関数
- ・ [UCASE](#) 関数
- ・ [照合](#)

# USER (SQL)

現在のユーザのユーザ名を返す関数です。

## 構文

USER

```
{fn USER}  
{fn USER()}
```

## 概要

USER 関数は引数を取らず、現ユーザのユーザ名を返します (承認 ID として参照)。汎用関数では括弧を使用できません。ODBC スカラ関数は、空の括弧を指定したり、省略できます。

ユーザ名は CREATE USER コマンドを使用して定義します。

USER は、主に SELECT 文のセレクト・リストや、クエリの WHERE 節で使用されます。レポートの設計では、USER はレポートが作成された現在のユーザを印刷するために使用されます。

## 例

以下の例は、現在のユーザ名を返します。

### SQL

```
SELECT USER AS CurrentUser
```

以下の例は、姓 (\$PIECE(Name,' ',1)) または名 (ミドル・イニシャルは含まない) が現在のユーザ名と一致するレコードを選択します。

### SQL

```
SELECT Name FROM Sample.Person  
WHERE %SQLUPPER(USER)=%SQLUPPER($PIECE(Name,' ',1))  
OR %SQLUPPER(USER)=%SQLUPPER($PIECE($PIECE(Name,' ',2),' ',1))
```

## 関連項目

- ・ [CREATE USER、GRANT](#)

## VECTOR\_COSINE (SQL)

2 つのベクトル間のコサイン類似度を求め、結果を返します。

### 構文

```
VECTOR_COSINE(vec1, vec2)
```

### 説明

VECTOR\_COSINE 関数は、2 つの入力ベクトルのコサイン類似度を求めます。これらのベクトルは、integer、double、decimal のいずれかの数値型である必要があります。結果は、これらの間のコサインの値で、-1 ~ 1 の double として表されます。この値は、2 つのベクトルのドット積を 2 つのベクトルの長さの積で除算した値として計算されます。この式は以下のように表されます。

$$\text{cosine} = \frac{\vec{V}_1 \cdot \vec{V}_2}{\|\vec{V}_1\| \|\vec{V}_2\|}$$

コサイン類似度は、2 つのベクトルの類似度の指標の 1 つです。コサイン類似度が正の場合、2 つのベクトルは類似していると見なされます。コサイン類似度が負の場合、2 つのベクトルは類似していないと見なされます。

### 引数

#### vec1、vec2

同じ数値型で同じ長さの 2 つのベクトル。これらのベクトルが異なる数値型の場合、この関数は SQLCODE -259 で失敗します。

string など、数値型ではないベクトルを指定した場合、この関数は SQLCODE -258 で失敗します。

これら 2 つのベクトルの長さが異なる場合、この関数は SQLCODE -257 エラーで失敗します。

### 例

以下の例では、2 つの integer 型のベクトルに VECTOR\_COSINE 関数を使用しています。

#### SQL

```
SELECT VECTOR_COSINE(TO_VECTOR('6,4,5',integer), TO_VECTOR('1,4,3',integer))
```

以下の例では、VECTOR\_COSINE 関数を使用して、各行に格納されているベクトルと入力ベクトル間の類似度に基づいて、結果に最も類似する説明テキストを返します。実際には、埋め込みベクトルには 3 つより多くの要素がありますが、この例では簡潔にするために、埋め込みベクトルを 3 つの要素を持つものとして表しています。

#### SQL

```
SELECT TOP 5 Description FROM Sample.DescAndEmbedding emb  
ORDER BY VECTOR_COSINE(Embedding,TO_VECTOR('0.3,0.4,0.6',double)) DESC
```

## 関連項目

- ・ [VECTOR\\_DOT\\_PRODUCT](#)
- ・ [ベクトル検索の使用法](#)

## VECTOR\_DOT\_PRODUCT (SQL)

2 つのベクトル間のドット積を求め、結果を返します。

### 構文

```
VECTOR_DOT_PRODUCT(vec1, vec2)
```

### 説明

VECTOR\_DOT\_PRODUCT 関数は、2 つの入力ベクトルのドット積を求めます。これらのベクトルは、integer、double、decimal のいずれかの数値型である必要があります。結果は、double として表されたドット積の値で、2 つのベクトル間の類似度を判断する場合に役立ちます。

2 つのベクトルのドット積は、2 つの入力ベクトルの同じ位置にある要素の各ペアの積の合計です。この計算式は以下のように表されます。

$$A \cdot B = \sum_{i=1}^n A_i B_i$$

これらの入力ベクトルが単位ベクトルの場合、ドット積は 2 つのベクトル間の角度のコサインと同じ値になります。ドット積が大きいほど、ベクトルの類似度が高くなります。この計算は、[VECTOR 型](#)の効率性を利用しているため、単位ベクトルに対して演算を行う場合は、この関数を使用することをお勧めします。

単位ベクトルではない 2 つのベクトル間の類似度を求める場合は、[VECTOR\\_COSINE](#) 関数を使用します。

### 引数

#### vec1、vec2

同じ数値型の 2 つのベクトル。これらのベクトルが異なる数値型の場合、この関数は SQLCODE -259 で失敗します。

String など、数値型ではないベクトルを指定した場合、この関数は SQLCODE -258 で失敗します。

2 つのベクトルは同じ長さである必要があります。これら 2 つのベクトルの長さが異なる場合、SQLCODE -257 エラーが返されます。

### 例

以下の例では、2 つの integer 型のベクトルに VECTOR\_DOT\_PRODUCT を使用しています。

#### SQL

```
SELECT VECTOR_DOT_PRODUCT(TO_VECTOR('6,4,5',integer), TO_VECTOR('1,4,3',integer))
```

以下の例では、VECTOR\_DOT\_PRODUCT 関数を使用して、各行に格納されているベクトルと入力ベクトル間の類似度に基づいて、結果に最も類似する説明テキストを返します。実際には、埋め込みベクトルには 3 つより多くの要素がありますが、この例では簡潔にするために、埋め込みベクトルを 3 つの要素を持つものとして表しています。



## SQL

```
SELECT TOP 5 Description FROM Sample.DescAndEmbedding emb  
ORDER BY VECTOR_COSINE(Embedding,TO_VECTOR('1,1,1',double)) DESC
```

## 関連項目

- ・ [VECTOR\\_COSINE](#)
- ・ [ベクトル検索の使用法](#)

## WEEK (SQL)

日付式に対して年における週を整数として返す日付関数です。

### 構文

```
{fn WEEK(date-expression)}
```

### 概要

WEEK は、date-expression を受け取り、その日付に対するその年の初めからの週の数返します。

既定では、週は \$HOROLOG 日付 (1840 年 12 月 31 日からの日数を表す正または負の整数) を使用して計算されます。したがって、週は年と年をまたいでカウントされ、Week 1 は前年の最後の週から開始された 7 日間を終了した週となります。週は常に日曜日から始まり、したがって、暦年の最初の日曜日は、Week 1 から Week 2 への切り替えを意味します。その年の最初の日曜日が 1 月 1 日である場合、その日曜日は Week 1 になります。その年の最初の日曜日が 1 月 1 日より後の場合、その日曜日は Week 2 の第 1 日目になります。このため、Week 1 は、一般的には、7 日間より短くなります。DAYOFWEEK 関数を使用すると、週の曜日を決定できます。1 年の週の合計数は、一般的に 53 です。うるう年には 54 になります。

InterSystems IRIS では、1 年の週の決定について ISO 8601 規格もサポートしています。この規格は、主にヨーロッパの国で使用されます。InterSystems IRIS が ISO 8601 用に構成されている場合、WEEK は週を月曜日からカウントし、週はその週の木曜日が含まれる年に割り当てられます。例えば、2004 年の Week 1 は、週の木曜日が 2004 年 1 月 1 日で 2004 年最初の木曜日であるため、2003 年 12 月 29 日の月曜日から 2004 年 1 月 4 日の日曜日までです。2005 年の Week 1 は、木曜日が 2005 年 1 月 6 日で 2005 年最初の木曜日であるため、2005 年 1 月 3 日から 2005 年 1 月 9 日までになります。1 年の合計週数は、通常は 52 ですが 53 のこともあります。ISO 8601 のカウントを有効にするには、SET ^%SYS("sql", "sys", "week ISO8601")=1 とします。

date-expression には、InterSystems IRIS 日付整数、\$HOROLOG 値や \$ZTIMESTAMP 値、ODBC 形式の日付文字列、またはタイムスタンプを指定できます。

date-expression タイムスタンプには、データ型 %Library.PosixTime (エンコードされた 64 ビットの符号付き整数) またはデータ型 %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

タイムスタンプの時刻部分は評価されないので省略可能です。

DATEPART または DATENAME 関数を使用しても、DAYOFWEEK と同じ週情報が返されます。

この関数は、ObjectScript から WEEK() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.WEEK(date-expression)
```

### 日付の検証

WEEK は、入力値に対して以下のチェックを実行します。値がチェックに失敗した場合は、NULL 文字列が返されます。

- ・ 日付文字列は完全であると同時に、要素数、各要素の桁数、および区切り文字に適切な形式が使用されている必要があります。年は 4 桁で指定される必要があります。
- ・ 日付値は、有効な範囲内にある必要があります。年は 0001 から 9999、月は 1 から 12、日は 1 から 31、
- ・ 月の日数は、該当月と該当年に合ったものでなければなりません。例えば、日付 '02-29' が有効なのは、指定された年がうるう年の場合のみです。
- ・ 10 よりも小さい日付値の先頭のゼロは、記載、省略のどちらでもかまいません。その他の非標準的な整数値は許可されません。例えば、'07' または '7' は有効な日付値ですが、'007'、'7.0'、または '7a' は無効です。

## 引数

### date-expression

列の名前や、他のスカラ関数の結果、または日付やタイムスタンプ・リテラルである式。

## 例

以下の例は、2005 年 1 月 2 日（日曜日）と 2006 年 1 月 1 日（日曜日）の週の曜日と通算週を返します。

### SQL

```
SELECT {fn DAYOFWEEK("2005-1-2")},{fn WEEK("2005-1-2")},
       {fn DAYOFWEEK("2006-1-1")},{fn WEEK("2006-1-1")}
```

以下の例は、与えられた日付が 2004 年の第 9 週目にあたるので、9 を返します。

### SQL

```
SELECT {fn WEEK('2004-02-25')} AS Wk_Date,
       {fn WEEK('2004-02-25 08:35:22')} AS Wk_Tstamp,
       {fn WEEK(59590)} AS Wk_DInt
```

以下の例は、54 を返します。これは、この例のすぐ後に続く例で示されているように、この特定の日付が 2 日目に開始する Week 2 で始まるうるう年にあるためです。

### SQL

```
SELECT {fn WEEK('2000-12-31')} AS Week
```

### SQL

```
SELECT {fn WEEK('2000-01-01')} || {fn DAYNAME('2000-01-01')} AS WeekofDay1,
       {fn WEEK('2000-01-02')} || {fn DAYNAME('2000-01-02')} AS WeekofDay2
```

以下の例はすべて、現在の週を返します。

### SQL

```
SELECT {fn WEEK({fn NOW()})} AS Wk_Now,
       {fn WEEK(CURRENT_DATE)} AS Wk_CurrD,
       {fn WEEK(CURRENT_TIMESTAMP)} AS Wk_CurrTS,
       {fn WEEK($HOROLOG)} AS Wk_Horolog,
       {fn WEEK($ZTIMESTAMP)} AS Wk_ZTS
```

以下の埋め込み SQL の例は、InterSystems IRIS の既定の年における週と、ISO 8601 規格が適用された年における週を示しています。

### ObjectScript

```
TestISO
SET def=$DATA(^%SYS("sql","sys","week ISO8601"))
IF def=0 {SET ^%SYS("sql","sys","week ISO8601")=0}
ELSE {SET isoval=%SYS("sql","sys","week ISO8601")}
    IF isoval=1 {GOTO UnsetISO}
    ELSE {SET isoval=0 GOTO WeekOfYear}
UnsetISO
SET ^%SYS("sql","sys","week ISO8601")=0
WeekOfYear
&sql(SELECT {fn WEEK($HOROLOG)} INTO :a)
WRITE "For Today:",!
WRITE "default week of year is ",a,!
SET ^%SYS("sql","sys","week ISO8601")=1
&sql(SELECT {fn WEEK($HOROLOG)} INTO :b)
WRITE "ISO8601 week of year is ",b,!
ResetISO
SET ^%SYS("sql","sys","week ISO8601")=isoval
```

## 関連項目

- ・ SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DAYOFWEEK](#)、[MONTH](#)、[QUARTER](#)、[TO\\_DATE](#)、[YEAR](#)
- ・ ObjectScript 特殊変数: [\\$HOROLOG](#)、[\\$ZTIMESTAMP](#)

# XMLCONCAT (SQL)

XML 要素を連結する関数です。

## 構文

```
XMLCONCAT(XmlElement1,XmlElement2[,...])
```

## 概要

XMLCONCAT 関数は、複数の XMLELEMENT 関数からの値を単一の文字列として返します。XMLCONCAT は、テーブルまたはビューを参照する [SELECT](#) クエリまたは SELECT サブクエリで使用できます。XMLCONCAT は、一般のフィールド値と共に SELECT リストで使用できます。

## 引数

### XmlElement

XMLELEMENT 関数。連結する複数の XmlElement を指定します。

## 例

以下のクエリは、2 つの XMLELEMENT 関数からの値を連結します。

### SQL

```
SELECT Name,XMLCONCAT(XMLELEMENT("Para",Name),
                        XMLELEMENT("Para",Home_City)) AS ExportString
FROM Sample.Person
```

返されるデータ行は、次のようになります。

```
ExportString
<Para>Emerson,Molly N.</Para><Para>Boston</Para>
```

以下のクエリは、XMLELEMENT 関数内で XMLCONCAT を入れ子にします。

### SQL

```
SELECT XMLELEMENT("Item",Name,
                  XMLCONCAT(
                      XMLELEMENT("Para",Home_City,' ',Home_State),
                      XMLELEMENT("Para",'is residence')))
      AS ExportString
FROM Sample.Person
```

返されるデータ行は、次のようになります。

```
ExportString
<Item>Emerson,Molly N.<Para>Boston MA</Para><Para>is residence</Para></Item>
```

## 関連項目

[SELECT](#) 文

[XMLAGG](#) 関数

[XMLELEMENT](#) 関数

## XMLELEMENT (SQL)

1 つまたは複数の式の値を囲むように XML マークアップ・タグをフォーマットする関数。

### 構文

```
XMLELEMENT([NAME] tag,expression[,expression])
XMLELEMENT([NAME] tag,XMLATTRIBUTES(expression [AS alias]),expression, ...)
```

### 概要

XMLELEMENT 関数は、tag で指定された XML (または HTML) マークアップ開始タグと終了タグを付けた expression の値を返します。例えば、XMLELEMENT(NAME "Para",Home\_City) は、<Para>Chicago</Para> のような値を返します。XMLELEMENT を使用して空要素タグを生成することはできません。

XMLELEMENT は、テーブルまたはビューを参照する [SELECT](#) クエリまたは SELECT サブクエリで使用できます。XMLELEMENT は、一般のフィールド値と共に SELECT リストで使用できます。

tag 引数では、二重引用符を使用してリテラル文字列を囲みます。その他ほぼすべてのコンテキストにおいて、InterSystems SQL では一重引用符を使用してリテラル文字列を囲みます。二重引用符は[区切り識別子](#)の指定に使用します。したがって、この機能を使用するには、区切り識別子のサポートを有効にする必要があります。区切り識別子は既定で有効になっています。

[ダイナミック SQL](#) の %Prepare() メソッドのように、SQL コードを二重引用符で区切った文字列として指定する場合は、以下のように二重引用符を 2 つ指定して tag の二重引用符をエスケープする必要があります。

#### ObjectScript

```
SET myquery = "SELECT XMLELEMENT('"Para"',Name) FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
```

一般的に expression は、クエリで返される複数行の中のフィールド名 (または、フィールド名を 1 つ以上含む式) です。expression は任意のタイプのフィールドにできます。指定された expression の値は、次の形式のように、開始タグと終了タグで囲んで返します。

```
<tag>value</tag>
```

タグを付ける値が空文字列 (") または NULL の場合、次のように返します。

```
<tag></tag>
```

expression に複数のコンマ区切り要素が含まれる場合、次の形式のように、結果が連結されます。

```
<tag>value1value2</tag>
```

expression が [データ・ストリーム・フィールド](#) の場合、ストリーム値は生成された XML 値内で以下を使用してエスケープされます。...

```
<tag>value
</tag>
```

XMLELEMENT 関数は入れ子にして使用できます。XMLELEMENT と XMLFOREST 関数は任意に組み合わせて入れ子にして使用できます。XMLELEMENT 関数は、[XMLCONCAT](#) を使用して連結できます。ただし、XMLELEMENT は、式全体の XML タイプの解析を実行しません。例えば、XMLELEMENT は CASE 文の節内の文字変換を実行できません (以下の例を参照)。

## XMLATTRIBUTES 関数

XMLATTRIBUTES 関数は、XMLELEMENT 関数内でのみ使用できます。expression の要素が XMLATTRIBUTES 関数の場合、次の形式で示すように、指定された式はタグの属性になります。

```
<tag ID='63' >value</tag>
```

XMLATTRIBUTES 関数は、XMLELEMENT 関数内で 1 つのみ指定できます。これは expression で任意の要素にできますが、慣例では最初の expression の要素とします。属性値は一重引用符で囲まれ、属性値とタグの閉じ山括弧 (>) との間に空白が挿入されます。

## XMLELEMENT と XMLFOREST の比較

- ・ XMLELEMENT は、単一のタグ内に expression リストの値を連結します。XMLFOREST は、各 expression アイテムに個別のタグを割り当てます。
- ・ XMLELEMENT は、タグの値を指定する必要があります。XMLFOREST は、既定のタグの値を使用することも、個々のタグの値を指定することもできます。
- ・ XMLELEMENT は、XMLATTRIBUTES を使用してタグの属性を指定できます。XMLFOREST は、タグの属性を指定できません。
- ・ XMLELEMENT は、NULL のタグ文字列を返します。XMLFOREST は、NULL のタグ文字列を返しません。

## 句読点文字値

データ値に、XML/HTML でタグまたはその他のコーディングとして解釈される句読点文字が含まれる場合、XMLELEMENT と XMLFOREST はこの文字を対応するエンコード形式に変換します。

アンド記号 (&) は &amp; となります。

アポストロフィ (') は &apos; となります。

引用符 (") は &quot; となります。

左山括弧 (<) は &lt; となります。

右山括弧 (>) は &gt; となります。

入力文字列でアポストロフィを表すには、can"t のように、2 つのアポストロフィを指定します。列のデータには、アポストロフィを 2 つ付ける必要はありません。

## 引数

### NAME tag

XML マークアップ・タグの名前。NAME キーワードはオプションです。この引数には、NAME "tag"、"tag"、および NAME という 3 つの構文形式があります。最初の 2 つは機能的に同じです。指定する場合は、tag を二重引用符で囲む必要があります。tag の大文字/小文字の区別は保持されます。

XMLELEMENT は tag の値の検証を実行しません。ただし、XML 規格では、有効な tag 名には文字 !"#\$%&'()\*+,-./:;<=>?@[\\]^\_`{|}~ やスペース文字を含めないこと、および最初の文字に "-、".、または数字を使用しないことが定められています。

NAME キーワードを tag の値なしで指定すると、既定のタグの値、<Name> ...</Name> となります。NAME キーワードは、大文字と小文字が区別されません。結果のタグは、最初の文字が大文字で表示されます。

## expression

任意の有効な式。普通は、タグ付けの対象となるデータ値を含む列の名前を指定します。列またはその他の式のコンマ区切りリストを指定できます。これらすべては同じ tag で囲まれます。最初のコンマ区切り要素を、XMLATTRIBUTES 関数にすることができます。XMLATTRIBUTES 要素は、1 つのみ指定できます。

## 例

次の例は、通常のデータおよび XML タグが付けられたデータとして、Sample.Person の各個人の Name フィールドの値を返します。

### SQL

```
SELECT Name,
       XMLELEMENT("Para",Name) AS ExportName
FROM Sample.Person
```

返されるデータ行は、次のようになります。

Name	ExportName
Emerson,Molly N.	<Para>Emerson,Molly N.</Para>

次の例では、Sample.Person 内の重複していない Home\_City 値と Home\_State 値の組み合わせごとに、データに XML のタグ <Address> ... </Address> を付けて返します。市区町村名と都道府県名が連結されないようにするには、空白の expression を指定します。

```
SELECT DISTINCT
       XMLELEMENT(NAME "Address",Home_City,' ',Home_State) AS CityState
FROM Sample.Person
ORDER BY Home_City
```

上の例では、オプションの NAME キーワードが使用されています。以下の例は、NAME キーワードを tag 値なしで使用します。

### SQL

```
SELECT DISTINCT
       XMLELEMENT(NAME,Home_City,' ',Home_State) AS CityState
FROM Sample.Person
ORDER BY Home_City
```

この場合、同じデータを返しますが、既定のタグ <Name> ... </Name> が付けられます。

以下の例は、[文字ストリーム・データ](#)を返します。

### SQL

```
SELECT XMLELEMENT("Para",Name) AS XMLNotes,XMLELEMENT("Para",Notes) AS XMLText
FROM Sample.Employee
```

返されるデータ行は、次のようになります。

XMLName	XMLText
<Para>Emerson,Molly N.</Para>	<Para>Molly worked at DynaMatix Holdings Inc. as a Marketing Manager</Para>

以下の例は、XMLELEMENT 関数を入れ子にして使用できることを示します。

### SQL

```
SELECT XMLELEMENT("Para",Home_State,
                  XMLELEMENT("Emphasis",Name),Age)
FROM Sample.Person
```



返されるデータ行は、次のようになります。

```
<Para>CA<Emphasis>Emerson,Molly N.</Emphasis>24</Para>
```

以下の例は、サブクエリの値を使用する XMLELEMENT 関数を示しています。

## SQL

```
SELECT XMLELEMENT("Para",Name,DOB, XMLELEMENT("Emphasis",%ID),Age,
  (SELECT XMLELEMENT("NameSub",Name) FROM Sample.Person WHERE %ID=2)) AS ExportName
FROM Sample.Person WHERE %ID=1
```

返されるデータ行は、次のようになります。

```
<Para>Zucherro,Rob F.38405<Emphasis>1</Emphasis>71<NameSub>Quixote,Mark N.</NameSub></Para>
```

以下の例は、XMLELEMENT が CASE 文の節内に値をタグ付けできないことを示します。

## SQL

```
SELECT XMLELEMENT("Para",Home_State,
  XMLELEMENT("Para",Name),
  CASE WHEN Age < 21 THEN NULL
    ELSE XMLELEMENT("Para",Age) END )
FROM Sample.Person
```

返されるデータ行は、次のようになります。

```
<Para>CA<Para>Emerson,Molly N.</Para>&lt;Para&gt;24&lt;/Para&gt;</Para>
```

以下のクエリは、Sample.Person の Name フィールドの値を返します。これはデータに XML のタグ付けをし、このタグ内で ID フィールドをタグ属性として使用したものです。

## SQL

```
SELECT XMLELEMENT("Para",XMLATTRIBUTES(%ID),Name) AS ExportName
FROM Sample.Person
```

返されるデータ行は、次のようになります。

```
ExportName
<Para ID='101' >Emerson,Molly N.</Para>
```

属性のエイリアスの指定方法を、以下の例に示します。

## SQL

```
SELECT XMLELEMENT("Para",XMLATTRIBUTES(%ID AS ItemKey),Name)
FROM Sample.Person
```

返されるデータ行は、次のようになります。

```
<Para ItemKey='101' >Emerson,Molly N.</Para>
```

## 関連項目

[XMLAGG 関数](#)

[XMLCONCAT 関数](#)

[XMLFOREST 関数](#)

[SELECT 文](#)

## XMLFOREST (SQL)

式の値を囲むように複数の XML マークアップ・タグをフォーマットする関数。

### 構文

```
XMLFOREST(expression [AS tag], ...)
```

### 概要

XMLFOREST 関数は、各 expression の値に、tag で指定された独自の XML マークアップ開始タグと終了タグを付けて返します。例えば、XMLFOREST(Home\_City AS City,Home\_State AS State) は、  
<City>Chicago</City><State>IL</State> のような値を返します。XMLFOREST を使用して空要素タグを生成することはできません。

XMLFOREST は、テーブルまたはビューを参照する [SELECT](#) クエリまたは SELECT サブクエリで使用できます。XMLFOREST は、一般の列の値と共に SELECT リストで使用できます。

指定された expression の値は、次の形式のように、開始タグと終了タグで囲んで返します。

```
<tag>value</tag>
```

一般的に expression は、列名、または 1 つ以上の列名を含む式です。expression は、[データ・ストリーム・フィールド](#)を含む任意のタイプのフィールドにすることができます。XMLFOREST は、それぞれの expression を次のようにタグ付けします。

- AS tag が指定された場合、XMLFOREST は指定されたタグを結果の値に付けます。tag の値は大文字と小文字を区別します。
- AS tag が省略され、expression が列名の場合、XMLFOREST は列名を結果の値にタグ付けします。列名の既定のタグは、常に大文字です。
- expression が列名でない場合 (集約関数、リテラル、2 つの列の連結など)、AS tag 節は必須です。
- expression が[ストリーム・フィールド](#)の場合、ストリーム値は生成された XML 値内で以下を使用してエスケープされます。... :

```
<tag>value  
</tag>
```

XMLFOREST は、コンマ区切りリストの各アイテムに個別のタグを割り当てます。XMLELEMENT は、単一のタグ内にコンマ区切りリストのすべてのアイテムを連結します。

XMLFOREST 関数は入れ子にして使用できます。入れ子にした XMLFOREST 関数および XMLELEMENT 関数は任意の組み合わせが可能です。XMLFOREST 関数は、[XMLCONCAT](#) を使用して連結できます。

### NULL 値

XMLFOREST 関数は、実際のデータの値がある場合のみタグを返します。expression の値が NULL の場合、タグを返しません。例を以下に示します。

### SQL

```
INSERT INTO Sample.Xmltest (f1,f2,f3) values (NULL,'Row 1',NULL)
```

### SQL

```
SELECT XMLFOREST(f1,f2,f3) from Sample.Xmltest
```

これは <F2>Row 1</F2> を返します。

空文字列 (') は、文字列データ型フィールドのデータ値と見なされます。タグ付けする f3 値が空文字列 (') の場合、XMLFOREST は以下を返します。

```
<F3></F3>
```

XMLFOREST は、NULL の処理方法において、XMLELEMENT と異なります。XMLELEMENT は、フィールドの値が NULL の場合でも、常にタグの値を返します。このため、XMLELEMENT では NULL と空文字列は区別されません。どちらも、<tag></tag> と表示されます。

## 句読点文字値

データ値に、XML/HTML でタグまたはその他のコーディングとして解釈される句読点文字が含まれる場合、XMLFOREST と XMLELEMENT はこの文字を対応するエンコード形式に変換します。

アンド記号 (&) は &amp; となります。

アポストロフィ (') は &apos; となります。

引用符 (") は &quot; となります。

左山括弧 (<) は &lt; となります。

右山括弧 (>) は &gt; となります。

入力文字列でアポストロフィを表すには、can"t のように、2 つのアポストロフィを指定します。列のデータには、アポストロフィを 2 つ付ける必要はありません。

## 引数

### expression

任意の有効な式。普通は、タグ付けの対象となるデータ値を含む列の名前を指定します。コンマ区切りリストとして指定すると、リスト内の式それぞれが専用の XML マークアップ・タグで囲まれます。

### AS tag

XML マークアップ・タグの名前を指定する引数 (オプション)。tag を指定する場合、AS キーワードは必須です。tag の大文字/小文字の区別は保持されます。

tag を二重引用符で囲むかどうかはオプションです。二重引用符を省略した場合、tag は XML の名前付け標準に従います。二重引用符で tag を囲むと、これらの名前付けの制限が取り除かれます。

XMLFOREST は、有効な tag 名に XML の名前付け標準を適用します。名前に文字

!"#\$%&'()\*+,-./:;<=>?@[\\]^\_`{|}~ やスペース文字を含めることはできません。また、名前の最初の文字に "-、".、または数字を使用することもできません。

AS tag 節を付けずに expression を指定すると、タグの値は expression 列の名前 (大文字表記) になります。例えば、<HOME\_CITY>Chicago</HOME\_CITY> のようになります。

## 例

次のクエリは、通常のデータおよび XML タグが付けられたデータとして、Sample.Person の Name 列の値を返します。

### SQL

```
SELECT Name,XMLFOREST(Name) AS ExportName
FROM Sample.Person
```

返されるデータ行は、次のようになります。ここで、既定のタグは列名になります。

```
Name                ExportName
Emerson,Molly N.    <NAME>Emerson,Molly N.</NAME>
```

以下の例は複数の列を指定します。

## SQL

```
SELECT XMLFOREST(Home_City,
                  Home_State AS Home_State,
                  AVG(Age) AS AvAge) AS ExportData
FROM Sample.Person
```

Home\_City フィールドはタグを指定しません。タグはすべて大文字で列名から生成されます：<HOME\_CITY>。Home\_State フィールドの AS 節はオプションです。タグ名を指定するとタグの書体を制御できるため、ここで指定されています：<HOME\_STATE>ではなく<Home\_State>。AVG(Age) の AS 節は必須です。これは、値が合計であり列の値ではないので、列名がないためです。返されるデータ行は、次のようになります。

```
ExportData
<HOME_CITY>Chicago</HOME_CITY><Home_State>IL</Home_State>
<AvAge>48.0198019801980198</AvAge>
```

以下の例は、[文字ストリーム・データ](#)を返します。

## SQL

```
SELECT XMLFOREST(name AS Para,Notes AS Para) AS XMLJobHistory
FROM Sample.Employee
```

返されるデータ行は、次のようになります。

```
XMLJobHistory
<Para>Emerson,Molly N.</Para><Para>Molly worked at DynaMatix Holdings Inc. as a Marketing Manager
</Para>
```

以下の例は、サブクエリの値を使用する XMLFOREST 関数を示しています。

## SQL

```
SELECT XMLFOREST(Name,DOB,Age,
                  (SELECT XMLFOREST(Name,DOB) FROM Sample.Person WHERE %ID=2) AS ExportName)
FROM Sample.Person where %ID=1
```

返されるデータ行は、次のようになります。

```
<NAME>Zahn,Rob F.</NAME><DOB>38405</DOB><AGE>71</AGE><ExportName><NAME>Quinn,Mark
N.</NAME><DOB>30999</DOB></ExportName>
```

## 関連項目

[XMLAGG 関数](#)

[XMLELEMENT 関数](#)

[XMLCONCAT 関数](#)

[SELECT 文](#)

# YEAR (SQL)

日付式に対して年を返す日付関数です。

## 構文

```
YEAR(date-expression)
{fn YEAR(date-expression)}
```

## 概要

YEAR は、InterSystems IRIS 日付整数 (\$HOROLOG 日付)、ODBC 形式の日付文字列、またはタイムスタンプを入力として取ります。YEAR は対応する年を整数で返します。

date-expression タイムスタンプには、データ型 **%Library.PosixTime** (エンコードされた 64 ビットの符号付き整数) またはデータ型 **%Library.TimeStamp** (yyyy-mm-dd hh:mm:ss.fff) のいずれかを指定できます。

年 (yyyy) 部分は 0001 ~ 9999 の範囲の 4 桁の整数にする必要があります。入力では、先頭のゼロはオプションです。出力では、先頭のゼロは抑制されます。2 桁の年は、4 桁の年に拡張されません。

date-expression の日付部分は検証され、1 から 12 までの範囲の月、および指定した月と年の有効な日の値を含む必要があります。それ以外の場合には、SQLCODE -400 エラー <ILLEGAL VALUE> が生成されます。

date-expression の時刻部分が存在する場合は検証されますが、省略可能です。

**注釈** InterSystems IRIS の日付の内部表現と互換性を持たせるために、すべての年の値は、0001 から 9999 までの 4 桁の整数で表記することを強くお勧めします。

TO\_DATE および TO\_CHAR の各 SQL 関数は、0001 年より前の年を表現できる“ユリウス日”をサポートしています。ObjectScript では、このようなユリウス日をサポートするメソッドを呼び出すことができます。

年形式の既定は、4 桁の年です。この年表示の既定を変更するには、YEAR\_OPTION オプションで **SET OPTION** コマンドを使用します。

YEAR、MONTH、DAY、DAYOFMONTH、HOUR、MINUTE、SECOND、**DATEPART** または **DATENAME** 関数を使用して、同じ要素を取得することもできます。

この関数は、ObjectScript から YEAR() メソッド・コールを使用して呼び出すこともできます。

```
$SYSTEM.SQL.Functions.YEAR(date-expression)
```

## 引数

### date-expression

InterSystems IRIS 日付整数、ODBC 日付文字列、またはタイムスタンプとして評価される式。この式は、列の名前、他のスカラー関数の結果、または日付やタイムスタンプのリテラルとすることができます。

## 例

以下の例は、整数 2018 を返します。

### SQL

```
SELECT YEAR('2018-02-22 12:45:37') AS ODBCDate_Year
```

### SQL

```
SELECT {fn YEAR(64701)} AS HorologDate_Year
```

以下の例は、現在の年を返します。

### SQL

```
SELECT YEAR(GETDATE()) AS Year_Now
```

以下の例は、2 つの関数から現在の年を返します。CURRENT\_DATE 関数は、DATE データ型を返します。NOW 関数は、TIMESTAMP データ型を返します。YEAR は、両方の入力データ型に 4 桁の年を表す整数を返します。

### SQL

```
SELECT {fn YEAR(CURRENT_DATE)}, {fn YEAR({fn NOW()})}
```

## 関連項目

- ・ SQL 関数 : [DATENAME](#)、[DATEPART](#)、[DAYOFYEAR](#)、[QUARTER](#)、[WEEK](#)、[TO\\_DATE](#)
- ・ ObjectScript 関数: [\\$ZDATE](#)

# SQL 単項演算子

## – (負の数)

---

式を負の数値として返す、単項演算子です。

### 構文

*-expression*

### 概要

単項演算子は、数値データ型カテゴリのすべてのデータ型の式の中でも、1 つの式でしか処理を実行しません。

– (負の数) は InterSystems SQL の拡張です。

### 引数

*expression*

数値式。

### 例

以下の例は、Sample.Person からの Age 列、Age の平均の – (負の数) 値、および平均年齢を引いた Age の 3 つの数値フィールドを返します。

### SQL

```
SELECT Age,  
       -(AVG(age)) AS NegAvg,  
       Age-AVG(Age) AS AgeRelAvg  
FROM Sample.Person
```

### 関連項目

[+ \(正の数\)](#)



## + (正の数)

---

式を正の数値として返す、単項演算子です。

### 構文

`+expression`

### 概要

単項演算子は、1 つの式でしか処理を実行しません。この式は、数値データ型カテゴリのすべてのデータ型の式にできます。

+ (正の数) は InterSystems SQL の拡張です。

### 引数

`expression`

数値式。

### 関連項目

[- \(負の数\)](#)



# SQL リファレンス資料

## Data Types (SQL)

SQL 列に記述できるデータの種別を指定します。

### InterSystems SQL でのデータ型

データ型は、テーブル列で保持できる値の種別を指定します。InterSystems SQL では、CREATE TABLE または ALTER TABLE を使用してフィールドを定義するときにデータ型を指定します。データ定義言語 (DDL) のデータ型または InterSystems IRIS のデータ型クラスを定義できます。以下に例を示します。

#### DDL Using SQL Data Types

```
CREATE TABLE Employees (
    FirstName VARCHAR(30),
    LastName VARCHAR(30),
    StartDate TIMESTAMP)
```

#### DDL Using InterSystems IRIS Data Types

```
CREATE TABLE Employees (
    FirstName %String(MAXLEN=30),
    LastName %String(MAXLEN=30),
    StartDate %TimeStamp)
```

### InterSystems IRIS へのデータ型のマッピングの表示

各 DDL データ型はそれと同等の InterSystems IRIS データ型にマッピングされます。システムで標準のマッピングを表示するには以下の手順に従います。

1. 管理ポータルで、[システム管理] を選択します。
2. [構成] > [SQL とオブジェクトの設定] に移動し、[システム DDL マッピング] をクリックします。

[システム定義DDLマッピング] ページに、以下の各列があるテーブルが表示されます。

- ・ Name – 指定できる DDL データ型の名前です。DDL データ型名では大文字と小文字が区別されません。
- ・ Datatype – InterSystems IRIS クラスで DDL データ型のマッピング先となっているデータ型の名前です。クラス名では大文字と小文字が区別されます。

[システム定義DDLマッピング] ページで既存のデータ型を変更でき、また削除できます。%1 などのリテラル・データ型パラメータと \$\$maxval^apiSQL(%1,%2) などの関数パラメータなどのマッピングを以下の表に示します。これらのパラメータの詳細は、“[データ型マッピング・パラメータ](#)” を参照してください。

DDL データ型	対応する InterSystems IRIS データ型クラス
BIGINT	%Library.BigInt  BIGINT 列に NULL ときわめて小さい負数の両方を記述する可能性がある場合は、標準のインデックス照合をサポートするために、インデックスの NULL 標識の再定義が必要になることがあります。詳細は、“ <a href="#">NULL のインデックス付け</a> ” を参照してください。
BIGINT(%1)	%Library.BigInt  %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。このデータ型は BIGINT と同等です。

DDL データ型	対応する InterSystems IRIS データ型クラス
BINARY	%Library.Binary(MAXLEN=1)
BINARY VARYING	%Library.Binary(MAXLEN=1)
BINARY VARYING(%1)	%Library.Binary(MAXLEN=%1) %1 はデータ型の最大長を設定します。
BINARY(%1)	%Library.Binary(MAXLEN=%1) %1 はデータ型の最大長を設定します。
BIT	%Library.Boolean このデータ型の詳細は、“ <a href="#">BIT データ型</a> ”を参照してください。
BLOB	%Stream.GlobalBinary
CHAR	%Library.String(MAXLEN=1)
CHAR VARYING	%Library.String(MAXLEN=1)
CHAR VARYING(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
CHAR(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
CHARACTER	%Library.Binary(MAXLEN=1)
CHARACTER VARYING	%Library.String(MAXLEN=1)
CHARACTER VARYING(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
CHARACTER(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
CLOB	%Stream.GlobalCharacter
DATE	%Library.Date
DATETIME	%Library.DateTime
DATETIME2	%Library.DateTime
DEC	%Library.Numeric (MAXVAL=999999999999999, MIN- VAL=-999999999999999, SCALE=0)

DDL データ型	対応する InterSystems IRIS データ型クラス
DEC(%1)	<pre>%Library.Numeric(MAXVAL=&lt; '\$max-val^%apiSQL(%1,0)'&gt;,MINVAL=&lt; '\$min-val^%apiSQL(%1,0)'&gt;,SCALE=0)</pre> <p>このデータ型は関数パラメータを使用し、位取りを 0 として、入力精度パラメータ (%1) に基づき、MINVAL と MAXVAL を設定します。これらのパラメータの詳細は、“<a href="#">精度と小数桁数</a>” を参照してください。</p> <p>例：DEC(4) は %Library.Numeric(MAXVAL=9999,MINVAL=-9999,SCALE=0) にマッピングされます。</p>
DEC(%1,%2)	<pre>%Library.Numeric (MAXVAL=&lt; '\$max-val^%apiSQL(%1,%2)'&gt;, MINVAL=&lt; '\$min-val^%apiSQL(%1,%2)'&gt;, SCALE=%2)</pre> <p>このデータ型は関数パラメータを使用して、入力精度パラメータ (%1) および位取りパラメータ (%2) に基づき、MINVAL、MAXVAL、SCALE を設定します。これらのパラメータの詳細は、“<a href="#">精度と小数桁数</a>” を参照してください。</p> <p>例：DEC(8,4) は %Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4) にマッピングされます。</p>
DECIMAL	<pre>%Library.Numeric (MAXVAL=9999999999999999, MINVAL=-9999999999999999, SCALE=0)</pre>
DECIMAL(%1)	<pre>%Library.Numeric(MAXVAL=&lt; '\$max-val^%apiSQL(%1,0)'&gt;,MINVAL=&lt; '\$min-val^%apiSQL(%1,0)'&gt;,SCALE=0)</pre> <p>このデータ型は関数パラメータを使用し、位取りを 0 として、入力精度パラメータ (%1) に基づき、MINVAL と MAXVAL を設定します。これらのパラメータの詳細は、“<a href="#">精度と小数桁数</a>” を参照してください。このデータ型は 64 ビット符号付き整数です。</p> <p>例：DECIMAL(6) は %Library.Numeric(MAXVAL=999999,MINVAL=-999999,SCALE=0) にマッピングされます。</p>
DECIMAL(%1,%2)	<pre>%Library.Numeric (MAXVAL=&lt; '\$max-val^%apiSQL(%1,%2)'&gt;, MINVAL=&lt; '\$min-val^%apiSQL(%1,%2)'&gt;, SCALE=%2)</pre> <p>このデータ型は関数パラメータを使用して、入力精度パラメータ (%1) および位取りパラメータ (%2) に基づき、MINVAL、MAXVAL、SCALE を設定します。これらのパラメータの詳細は、“<a href="#">精度と小数桁数</a>” を参照してください。</p> <p>例：DECIMAL(8,4) は %Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4) にマッピングされます。</p>

DDL データ型	対応する InterSystems IRIS データ型クラス
DOUBLE	%Library.Double  これは IEEE 浮動小数点で標準のデータ型です。このデータ型を持つ SQL 列は、既定では有効桁数が 20 の値を返します。最大値と最小値の制限などの重要な詳細は、 <a href="#">\$DOUBLE</a> 関数の説明を参照してください。
DOUBLE PRECISION	%Library.Double  これは IEEE 浮動小数点で標準のデータ型です。このデータ型を持つ SQL 列は、既定では有効桁数が 20 の値を返します。最大値と最小値の制限などの重要な詳細は、 <a href="#">\$DOUBLE</a> 関数の説明を参照してください。
FLOAT	%Library.Double  これは IEEE 浮動小数点で標準のデータ型です。このデータ型を持つ SQL 列は、既定では有効桁数が 20 の値を返します。
FLOAT(%1)	%Library.Double  これは IEEE 浮動小数点で標準のデータ型です。このデータ型を持つ SQL 列は、既定では有効桁数が 20 の値を返します。
IMAGE	%Stream.GlobalBinary
INT	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)
INT(%1)	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)  %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。このデータ型は INT と同等です。
INTEGER	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)
LONG	%Stream.GlobalCharacter
LONG BINARY	%Stream.GlobalBinary
LONG RAW	%Stream.GlobalBinary
LONG VARCHAR	%Stream.GlobalCharacter
LONG VARCHAR(%1)	%Stream.GlobalCharacter  %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。

DDL データ型	対応する InterSystems IRIS データ型クラス
LONGTEXT	%Stream.GlobalCharacter %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。
LONGVARBINARY	%Stream.GlobalBinary
LONGVARBINARY(%1)	%Stream.GlobalBinary %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。
LONGVARCHAR	%Stream.GlobalCharacter
LONGVARCHAR(%1)	%Stream.GlobalCharacter
MEDIUMINT	%Library.Integer (MAXVAL=8388607,MINVAL=-8388608) このデータ型は、MySQL との互換性を維持するために用意されています。
MEDIUMINT(%1)	%Library.Integer (MAXVAL=8388607,MINVAL=-8388608) %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。
MEDIUMTEXT	%Stream.GlobalCharacter
MONEY	%Library.Currency
NATIONAL CHAR	%Library.String(MAXLEN=1)
NATIONAL CHAR VARYING	%Library.String(MAXLEN=1)
NATIONAL CHAR VARYING(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
NATIONAL CHAR(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
NATIONAL CHARACTER	%Library.String(MAXLEN=1)
NATIONAL CHARACTER VARYING	%Library.String(MAXLEN=1)
NATIONAL CHARACTER VARYING(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
NATIONAL CHARACTER(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。
NATIONAL VARCHAR	%Library.String(MAXLEN=1)
NATIONAL VARCHAR(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の最大長を設定します。



DDL データ型	対応する InterSystems IRIS データ型クラス
NCHAR	%Library.String(MAXLEN=1)
NCHAR(%1)	%Library.String(MAXLEN=%1) %1 はデータ型の <b>最大長</b> を設定します。
NTEXT	%Stream.GlobalCharacter
NUMBER	%Library.Numeric(SCALE=0) このデータ型は 64 ビット符号付き整数です。
NUMBER(%1)	%Library.Numeric(MAXVAL=< '\$\$max- val^%apiSQL(%1)'> ,MINVAL=< '\$\$min- val^%apiSQL(%1)'> ,SCALE=0)  このデータ型は <b>関数パラメータ</b> を使用し、位取りを 0 とし て、入力精度パラメータ (%1) に基づき、MINVAL と MAXVAL を設定します。このデータ型は 64 ビット符号 付き整数です。  例 : NUMBER(6) は %Library.Numeric(MAX- VAL=999999,MINVAL=-999999,SCALE=0) にマッピング されます。
NUMBER(%1,%2)	%Library.Numeric (MAXVAL=< '\$\$max- val^%apiSQL(%1,%2)'> , MINVAL=< '\$\$min- val^%apiSQL(%1,%2)'> , SCALE=%2)  このデータ型は <b>関数パラメータ</b> を使用して、入力精度パ ラメータ (%1) および位取りパラメータ (%2) に基づき、 MINVAL、MAXVAL、SCALE を設定します。  例 : NUMBER(8,4) は %Library.Numeric(MAX- VAL=9999.9999,MINVAL=-9999.9999,SCALE=4) にマッ ピングされます。
NUMERIC	%Library.Numeric (MAXVAL=9999999999999999, MIN- VAL=-9999999999999999, SCALE=0)
NUMERIC(%1)	%Library.Numeric(MAXVAL=< '\$\$max- val^%apiSQL(%1,0)'> ,MINVAL=< '\$\$min- val^%apiSQL(%1,0)'> ,SCALE=0)  このデータ型は <b>関数パラメータ</b> を使用し、位取りを 0 とし て、入力精度パラメータ (%1) に基づき、MINVAL と MAXVAL を設定します。このデータ型は 64 ビット符号 付き整数です。  例 : NUMERIC(6) は %Library.Numeric(MAX- VAL=999999,MINVAL=-999999,SCALE=0) にマッピング されます。

DDL データ型	対応する InterSystems IRIS データ型クラス
NUMERIC(%1,%2)	<p>%Library.Numeric (MAXVAL=&lt;'\$\$max-val'^%apiSQL(%1,%2)'&gt;, MINVAL=&lt;'\$\$min-val'^%apiSQL(%1,%2)'&gt;, SCALE=%2)</p> <p>このデータ型は関数パラメータを使用して、入力精度パラメータ (%1) および位取りパラメータ (%2) に基づき、MINVAL、MAXVAL、SCALE を設定します。</p> <p>例 : NUMERIC(8,4) は %Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4) にマッピングされます。</p>
NVARCHAR	%Library.String(MAXLEN=1)
NVARCHAR(%1)	<p>%Library.String(MAXLEN=%1)</p> <p>%1 はデータ型の最大長を設定します。</p>
NVARCHAR(%1,%2)	%Library.String(MAXLEN=%1)
NVARCHAR(MAX)	<p>%Stream.GlobalCharacter</p> <p>このデータ型は LONGVARCHAR と同等であり、TSQL との互換性を維持するために用意されています。</p>
POSIXTIME	%Library.PosixTime
RAW(%1)	<p>%Library.Binary(MAXLEN=%1)</p> <p>%1 はデータ型の最大長を設定します。</p>
REAL	<p>%Library.Double</p> <p>これは IEEE 浮動小数点で標準のデータ型です。このデータ型を持つ SQL 列は、既定では有効桁数が 20 の値を返します。</p>
ROWVERSION	<p>%Library.RowVersion</p> <p>このデータ型は、システムによって割り当てられる連番の整数です。詳細は、“<a href="#">ROWVERSION データ型</a>”を参照してください。</p>
SERIAL	<p>%Library.Counter</p> <p>このデータ型はシステムによって生成されます。</p>
SMALLDATETIME	%Library.DateTime(MINVAL="1900-01-01 00:00:00",MAXVAL="2079-06-06 23:59:59")
SMALLINT	%Library.SmallInt
SMALLINT(%1)	<p>%Library.SmallInt</p> <p>%1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。このデータ型は SMALLINT と同等です。</p>

DDL データ型	対応する InterSystems IRIS データ型クラス
SMALLMONEY	%Library.Currency
SYSNAME	%Library.String(MAXLEN=128)
TEXT	%Stream.GlobalCharacter
TIME	%Library.Time
TIME(%1)	%Library.Time(PRECISION=%1)  PRECISION は秒の小数部の桁数で、0 ~ 9 の範囲の整数値です。
TIMESTAMP	%Library.PosixTime
TIMESTAMP2	%Library.TimeStamp
TINYINT	%Library.TinyInt
TINYINT(%1)	%Library.TinyInt  %1 パラメータは無視されます。これは、MySQL との互換性を維持するために用意されています。このデータ型は TINYINT と同等です。
UNIQUEIDENTIFIER	%Library.UniqueIdentifier
VARBINARY	%Library.Binary(MAXLEN=1)
VARBINARY(%1)	%Library.Binary(MAXLEN=%1)  %1 はデータ型の <b>最大長</b> を設定します。
VARCHAR	%Library.String(MAXLEN=1)
VARCHAR(%1)	%Library.String(MAXLEN=%1)  %1 はデータ型の <b>最大長</b> を設定します。
VARCHAR(%1,%2)	%Library.String(MAXLEN=%1)  %1 はデータ型の <b>最大長</b> を設定します。
VARCHAR(MAX)	%Stream.GlobalCharacter  このデータ型は LONGVARCHAR と同等であり、TSQL との互換性を維持するために用意されています。
VARCHAR2(%1)	%Library.String(MAXLEN=%1)  %1 はデータ型の <b>最大長</b> を設定します。
VECTOR	%Library.Vector

## データ型マッピング・パラメータ

多くの場合、[システム定義DDLマッピング] テーブルには、各データ型に指定できるパラメータがわかるように、データ型ごとに複数のエントリが表示されます。このマッピング・テーブルには、パラメータの既定値も表示されます。一般的には、

DDL データ型よりもデータ型クラスの方に、使用できるデータ値を定義するためのパラメータがより多く用意されています。リテラル・パラメータまたは関数パラメータを指定できます。また、追加のデータ型クラス・パラメータも定義できます。

## リテラル・パラメータ

リテラル・パラメータは、DDL データ型と、%n 形式の InterSystems IRIS データ型で識別します。n はデータ型引数の番号です。例えば、VARCHAR(%1) は %String(MAXLEN=%1) にマッピングされます。

一般的なリテラル・パラメータとして、最大文字列長、最小値と最大値、精度と位取りの値などがあります。

### 最大長

データ型クラスでは、MAXLEN パラメータで文字列データ型の最大長を指定します。多くの場合、DDL 型では、それに対応する名前なしパラメータでこれらの値を定義します。

このフィールドの定義では、データ型は最大 64 文字の文字列です。

### DDL Using SQL Data Type

```
ProductName VARCHAR(64)
```

### DDL Using InterSystems IRIS Data Type

```
ProductName %String(MAXLEN=64)
```

このパラメータを指定するときは以下の点に注意します。

- MAXLEN 値が指定されていないフィールドには、[最大文字列長](#)までの任意の長さの値を設定できます。最大長の文字列フィールドを定義するには、VARCHAR("")を指定します。これにより、データ型 %Library.String(MAXLEN="") のプロパティが作成されます。VARCHAR() では、データ型 %Library.String(MAXLEN=1) のプロパティが作成されます。MAXLEN 値が指定されていないバイナリ・フィールドを定義するには、VARBINARY("")を指定します。これにより、データ型 %Library.Binary(MAXLEN="") のプロパティが作成されます。VARBINARY() では、データ型 %Library.Binary(MAXLEN=1) のプロパティが作成されます。
- 大きい MAXLEN 値の指定：MAXLEN 値が大きすぎると、ODBC アプリケーションに影響することがあります。ODBC アプリケーションは、サーバからメタデータに基づいて、必要なフィールド・サイズを決定しようとします。そのため、このアプリケーションは、実際に必要な量よりも多くのバッファ・スペースを割り当てる可能性があります。このため、InterSystems IRIS では、システム全体の既定の ODBC VARCHAR の最大長を 4096 に設定します。このシステム全体の既定値は、管理ポータルを使用して構成可能です。[\[システム管理\]](#) から、[\[構成\]](#)、[\[SQL およびオブジェクトの設定\]](#)、[\[SQL\]](#) の順に選択します。[\[VARCHAR の既定の長さ\]](#) オプションを表示または設定します。現在の設定を確認するには、\$SYSTEM.SQL.CurrentSettings() を呼び出します。InterSystems ODBC ドライバは、TCP バッファからデータを取得し、それをアプリケーションのバッファに変換します。そのため、MAXLEN のサイズが ODBC クライアントに影響することはありません。

### 最大値と最小値

データ型クラスでは、MINVAL パラメータと MAXVAL パラメータで数値データ型の最小値と最大値を指定します。多くの場合、データ型クラスにはリテラル・パラメータではなく関数パラメータを使用して、これらの値を定義します。DDL 型には、これに相当するパラメータがありません。

このフィールド定義では、データ型は 0 から 100 までの整数です。

```
Capacity %Integer(MINVAL=0,MAXVAL=100)
```

### 精度と小数桁数

データ型クラスの PRECISION パラメータと SCALE パラメータは、それぞれ精度 (最大桁数) と位取り (最大小数点以下桁数) を指定する整数値です。多くの場合、データ型クラスでは、これらの値をリテラル・パラメータではなく関数パラメータで定義します。NUMERIC などの DDL データ型では、精度と位取りを名前なしパラメータとしてまとめて指定することが普通です。

以下のフィールド定義では、精度が 6 で位取りが 2 の数を定義します。

## DDL

```
UnitPrice NUMERIC(6,2) // Range: -9999.99 to 9999.99
```

精度と位取りのパラメータでは、数値データ型を以下のように定義します。

- ・ 精度 – 最大許容値と最小許容値です。0 から 19 までの整数に s を加算した値を指定します。s は小数点以下桁数です。一般的に精度は数値の合計桁数ですが、その正確な値は、**%Library** クラス・データ型のマッピングによって決まります。この最大整数値は 9223372036854775807 です。19 + s より大きな精度は既定で 19 + s になります。
- ・ 位取り – 最大許容小数点以下桁数です。整数で指定します。s が精度 p 以上の場合には小数以下の数値のみを使用でき、実際の p 値は無視されます。許容される最大のスケールは 18 です。これは .999999999999999999 に相当します。18 より大きなスケールは既定で 18 になります。

数値のフォーマットの詳細は、**\$FNUMBER** 関数の説明を参照してください。

## 関数パラメータ

DDL データ型のパラメータを InterSystems IRIS データ型に入力できるように、事前に変換する必要がある場合に関数パラメータを使用します。以下の例では、DDL データ型数値の精度と位取りの各パラメータを InterSystems IRIS データ型の MAXVAL パラメータと MINVAL パラメータに変換しています。

例えば、管理ポータル**の**[システム定義DDLマッピング]テーブルに表示するために、DECIMAL DDL データ型と %Numeric クラスとのマッピングを考えます。

### DDL Using SQL Data Type

```
DECIMAL(%1,%2)
```

### DDL Using InterSystems IRIS Data Type

```
%Numeric(MAXVAL=<|'$maxval'^%apiSQL(%1,%2)'>|,MINVAL=<|'$minval'^%apiSQL(%1,%2)'>|,SCALE=%2)
```

パラメータ %1 と %2 で、目的とするデータ型数値の精度と位取りをそれぞれ指定します。例えば、データ型が DECIMAL(4,2) のフィールドには、精度が 4 桁で小数点以下が 2 桁の数値を格納します。InterSystems SQL では、これらのパラメータを使用して、フィールドの最小許容値 (-99.99) と最大許容値 (99.99) が導き出されます。

%Numeric クラスの場合、InterSystems IRIS で SCALE パラメータ (SCALE=%2) は設定されますが、設定すべき PRECISION パラメータがありません。代わりに、以下の各変換関数を使用して MAXVAL パラメータと MINVAL パラメータが設定されます。

- ・ maxval^%apiSQL(precision,scale) は、精度と位取りを設定した、有効な数値の最大値 MAXVAL を返します。
- ・ minval^%apiSQL(precision,scale) は、精度と位取りを設定した、有効な数値の最小値 MINVAL を返します。

これらの変換関数の構文は以下のとおりです。

```
dataTypeClass(param=<|'func'|>|, param2=<|func2|>|, ...)
```

- ・ dataTypeClass – マッピング先となるデータ型クラスの名前です。例：%Numeric
- ・ param – 設定するデータ型クラス・パラメータの名前です。例：MAXVAL
- ・ func – パラメータの設定に使用する関数呼び出しです。例：maxval^%apiSQL(%1,%2)

<|'func'|> 式は、指定された値を使用して func のパラメータを置き換えること、続いて指定の値でこの関数を呼び出すことを DDL プロセッサに指示します。続いて、この関数の呼び出しから返された値で <|'func'|> 式が置き換えられます。

## その他のパラメータ

DDL データ型では定義できないものの、データ型クラスでは定義できるデータ定義パラメータがあります。これには、[許可されるデータ値の列挙リスト](#)、[許可されるデータ値のパターン・マッチング](#)、MAXLEN の最大長を超えるデータ値の[自動切り捨て](#)などのデータ検証操作が含まれます。

## 新しい DDL データ型の作成

データ型の設定は、システム・データ型のパラメータ値のデータ型マッピングをオーバーライドするか、新しいユーザ・データ型を定義することによって変更できます。システム・データ型を変更して、インターシステムズ既定のマッピングをオーバーライドできます。ユーザ定義のデータ型を作成して、インターシステムズが提供していないデータ型マッピングを追加することができます。

新しい DDL データ型とそのマッピングを作成するには以下の手順に従います。

1. 管理ポータルで、[システム管理]を選択します。
2. [構成] > [SQL とオブジェクトの設定] に移動し、[ユーザ定義DDLマッピング] をクリックします。
3. [新規ユーザ定義DDLマッピング作成] をクリックして、データ型を入力するフォームを開きます。
4. フォームの [名前] フィールドに、DDL データ型の仕様を入力します。例えば、「VARCHAR(100)」と入力します。
5. [データタイプ] フィールドに、InterSystems IRIS の既存データ型クラスの名前または作成したデータ型の名前を入力します。例えば、「MyString100(MAXLEN=100)」と入力します。
6. [保存] をクリックします。

[ユーザ定義DDLマッピング] テーブルに新しいエントリが表示されます。このテーブルでエントリを変更または削除できます。

ユーザ定義データ型をデータ型クラスとして作成できます。例えば、最大で 10 文字を取得して残りの入力データを切り捨てる文字列データ型を作成できます。このデータ型 Sample.TruncStr は以下のように作成します。

### Class Definition

```
Class Sample.TruncStr Extends %Library.String
{
    Parameter MAXLEN=10;
    Parameter TRUNCATE=1;
}
```

このデータ型をテーブル定義で使用するには、以下のようにデータ型クラス名を指定します。

```
CREATE TABLE Sample.ShortNames (Name Sample.TruncStr)
```

データ型クラスを作成するときは以下の点に注意します。

- ・ MINVAL や MAXVAL などのパラメータを関数に設定するには[関数パラメータ](#)を使用します。
- ・ Stream のコレクション型を持つ InterSystems IRIS プロパティに DDL データ型をマップする必要がある場合は、文字ストリーム・データには %Stream.GlobalCharacter を、バイナリ・ストリーム・データには %Stream.GlobalBinary をそれぞれ指定します
- ・ [SystemDataTypes] テーブルの DDL データ型列にないデータ型が DDL で検出されると、次に [UserDataTypes] テーブルが調べられます。どちらのテーブルにもデータ型マッピングがない場合は、そのデータ型は変換されず、DDL で指定されているクラス定義に直接渡されます。

例えば、以下のフィールド定義が DDL 文にあるとします。

## SQL

```
CREATE TABLE TestTable (
    Field1 %String,
    Field2 %String(MAXLEN=45)
)
```

上記の定義では、%String、%String(MAXLEN=%1)、または %String(MAXLEN=45) のマッピングが、[System-DataTypes] と [UserDataTypes] に見つからない場合、%String と %String(MAXLEN=45) 型が適切なクラス定義に直接渡されます。

## 特定のデータ型の処理

### 日付、時刻、およびタイムスタンプのデータ型

InterSystems SQL 標準の日付、時刻の関数を使用して、日付、時刻、タイムスタンプのデータ型を定義できます。また、日付とタイムスタンプを相互に変換することもできます。例えば、CURRENT\_DATE や CURRENT\_TIMESTAMP をデータ型で定義されたフィールドへの入力として使用したり、DATEADD、DATEDIFF、ATENAME、または DATEPART を使用して、このデータ型で保存された日付値を操作したりできます。

日付、時刻、タイムスタンプのデータ型クラスと SQL のタイプとの対応を以下の表に示します。データ型クラスでは、SQL で計算するときにこのタイプを使用します。カスタムのデータ型クラスを作成すると、これらのマッピングを使用して、どの SQL タイプを指定するべきかをクラス定義の [SqlCategory](#) キーワードで指定できます。以下に例を示します。

#### Class Definition

```
Class MyApp.MyDateDT [ ClassType = DataType, SQLCategory = DATE ]
{
    // class members
}
```

データ型クラス	対応する SQL タイプ	留意事項
---------	--------------	------



データ型クラス	対応する SQL タイプ	留意事項
<ul style="list-style-type: none"> <li>・ %Library.Date クラス</li> <li>・ +\$HOROLOG (\$HOROLOG の日付部分) の論理値を持つプロパティまたは列</li> </ul>	DATE	<p>既定では、DATE および対応する %Library.Date データ型には正の整数のみを指定できます。0 を指定すると 1840-12-31 になります。1840-12-31 より前の日付をサポートするには、テーブル内にデータ型 %Library.Date(MINVAL=-nnn) の日付フィールドを定義する必要があります。MINVAL は、1840-12-31 から最大値 -672045 (0001-01-01) まで遡ってカウントする負の日数です。%Library.Date には、日付値を -672045 から 2980013 の範囲の符号なし整数または負の整数として格納できます。日付値は以下のように入力できます。</p> <ul style="list-style-type: none"> <li>・ 論理モードでは、65619 (2020 年 8 月 28 日) のような +HOROLOG 整数値を使用できます。</li> <li>・ 表示モードでは、変換メソッド DisplayToLogical() を使用します。このモードでは、現在のロケールの表示形式の日付 (例: '8/28/2020') を使用できます。また、論理日付値 (+HOROLOG 整数値) も使用できます。</li> <li>・ ODBC モードでは、変換メソッド OdbcToLogical() を使用します。このモードでは、ODBC 標準形式の日付 (例: '2020-08-28') を使用できます。また、論理日付値 (+HOROLOG 整数値) も使用できます。</li> </ul>

データ型クラス	対応する SQL タイプ	留意事項
<ul style="list-style-type: none"><li>・ %Library.Time クラス</li><li>・ \$PIECE(\$HOROLOG, "", 2) の論理値 (\$HOROLOG の時刻部分)を持つクラス</li></ul>	TIME	

データ型クラス	対応する SQL タイプ	留意事項
		<p>%Library.Time は、日付値を 0 から 86399 までの符号なし整数 (午前 0 時 00 分からの秒数) として格納します。時刻値は以下のように入力できます。</p> <ul style="list-style-type: none"> <li>・ 論理モードでは、84444 (23:27:24) のような \$PIECE(\$HOROLOG,"",2) 整数値を使用できます。</li> <li>・ 表示モードでは、変換メソッド DisplayToLogical() を使用します。このモードでは、現在のロケールの表示形式の日付 (例: '23:27:24') を使用できます。</li> <li>・ ODBC モードでは、変換メソッド OdbcToLogical() を使用します。このモードでは、ODBC 標準形式の時刻 (例: '23:27:24') を使用できます。また、論理時刻値 (0 から 86399 までの範囲の整数) も使用できます。</li> </ul> <p>TIME は秒の小数部をサポートするので、このデータ型はユーザが指定した最大 9 までの精度の桁数 (F) に合わせて HH:MM:SS.FF にも使用できます。秒の小数部をサポートするには、PRECISION パラメータを設定します。例えば、TIME(0) (%Time(PRECISION=0)) は、最も近い秒に丸めます。TIME(2) (%Time(PRECISION=2)) は、小数点以下 2 桁の精度に丸めます (または 0 で埋めます)。</p> <p>指定されたデータで精度も指定している場合 (CURRENT_TIME(3) など)、保存される小数点以下の桁は以下のようになります。</p> <ul style="list-style-type: none"> <li>・ TIME で精度を指定せず、データで精度を指定している場合、データの精度を使用します。</li> <li>・ TIME で精度を指定せず、データでも精度を指定していない場合、<a href="#">システム全体で構成された時間精度</a>を使用します。</li> <li>・ TIME で精度を指定し、データで精度を指定していない場合、データ精度として <a href="#">システム全体で構成された時間精度</a>を使用します。</li> <li>・ TIME で精度を指定し、データ精度が TIME の精度より低い場合、データ精度</li> </ul>

データ型クラス	対応する SQL タイプ	留意事項
		<p>を使用します。</p> <ul style="list-style-type: none"> <li>TIME で精度を指定し、データ精度が TIME の精度より高い場合、TIME の精度を使用します。</li> </ul> <p>SQL メタデータは時刻精度の小数点以下桁数を“スケール”としてレポートします。ここでは、データ全体の長さに対し、ワードの“精度”を使用します。TIME データ型を使用するフィールドは、精度とスケールのメタデータについて次のようにレポートします。TIME(0) (%Time (PRECISION=0)) のメタデータ精度は 8 (nn:nn:nn)、スケールは 0 です。TIME(2) (%Time (PRECISION=2)) のメタデータ精度は 11 (nn:nn:nn:ff)、スケールは 2 です。TIME (%Time または %Time (PRECISION= " ")) は、指定されたデータから秒の小数部の精度を取得するため、メタデータの精度 18 を持ち、スケールは定義されていません。データ型、精度、およびスケールのメタデータを返す方法の詳細は、“<a href="#">select-item のメタデータ</a>”を参照してください。</p>
<ul style="list-style-type: none"> <li>%Library.TimeStamp クラス</li> <li>YYYY-MM-DD HH:MI:SS.FF の論理値を持つクラス</li> </ul>	TIMESTAMP	<p>%Library.PosixTime の最大精度は 6 桁ですが、%Library.TimeStamp の最大精度は、秒の小数点以下桁数を最大 9 桁として、システム・プラットフォームの精度から導き出されます。したがって、同じプラットフォームにおいて、%Library.TimeStamp の方が %Library.PosixTime よりも精度が高くなる場合があります。%Library.TimeStamp の正規化により、9 桁の精度を超える入力値は自動的に 9 桁の小数秒に切り捨てられます。</p> <p>注釈    %Library.DateTime は %Library.TimeStamp のサブクラスです。これは、DATEFORMAT という名前の型パラメータを定義します。また、TSQL アプリケーションで習慣的に使用されている不正確な日時入力に対処するよう、DisplayToLogical() および OdbcToLogical() メソッドをオーバーライドします。</p>

データ型クラス	対応する SQL タイプ	留意事項
<ul style="list-style-type: none"> <li>・ %MV.Date クラス</li> </ul>	MVDATE	このデータ型は、MultiValue との互換性を維持する目的でのみサポートされています。
<ul style="list-style-type: none"> <li>・ \$HOROLOG-46385 の論理日付値を持つクラス。 \$HOROLOG-46385 は ObjectScript の日付を MultiValue の日付に変換するために使用する式です。</li> </ul>		
<ul style="list-style-type: none"> <li>・ 上記の論理値のいずれにも該当しないデータ型</li> </ul>	DATE	このクラスを定義するときは、論理日付値を %Library.Date 論理値に変換する LogicalToDate() メソッドと、それとは逆方向に変換する DateToLogical() メソッドを定義します。

データ型クラス	対応する SQL タイプ	留意事項
<ul style="list-style-type: none"><li>・ %Library.PosixTime クラス</li><li>・ エンコードした 64 ビットの符号付き整数の論理値を持つユーザ定義データ型クラス</li></ul>	POSIXTIME	

データ型クラス	対応する SQL タイプ	留意事項
		<p>%PosixTime は、1970-01-01 00:00:00 以降の秒数（および秒の小数部）から計算されるエンコードされたタイムスタンプです。その日付以降のタイムスタンプは正の %PosixTime 値で表され、その日付以前のタイムスタンプは負の %PosixTime 値で表されます。%PosixTime は、最大 6 桁の秒の小数部の精度をサポートします。%PosixTime でサポートされる最初の日付は 0001-01-01 00:00:00 で、論理値 -6979664624441081856 を持ちます。サポートされる最後の日付は 9999-12-31 23:59:59.999999 で、論理値 1406323805406846975 を持ちます。</p> <p>エンコードされた 64 ビットの整数で %PosixTime 値は常に表されるため、%Date 値や %TimeStamp 値と常に明確に区別できます。例えば、1970-01-01 00:00:00 の %PosixTime 値は 1152921504606846976、2017-01-01 00:00:00 の %PosixTime 値は 1154404733406846976、1969-12-01 00:00:00 の %PosixTime 値は -6917531706041081856 になります。</p> <p>%PosixTime は %TimeStamp データ型よりもディスク領域やメモリの使用量が少なく、%TimeStamp よりもパフォーマンスが高いため、%TimeStamp よりも優れています。</p> <p>ODBC <b>表示モード</b>を使用することで、%PosixTime 値と %TimeStamp 値を統合できます。</p> <ul style="list-style-type: none"> <li>・ %PosixTime データ型と %TimeStamp データ型の論理モード値は完全に異なります。%PosixTime は符号付き整数であり、%TimeStamp は ODBC 形式のタイムスタンプが含まれる文字列です。</li> <li>・ 表示モード: %PosixTime の表示では、現在のロケール時刻とロケール日付の形式パラメータ（例: 02/22/2018 08:14:11）が使用されます。一方、%TimeStamp は ODBC 形式のタイムスタンプとして表示されます。</li> <li>・ ODBC モード: %PosixTime も %TimeStamp も ODBC 形式のタイムスタンプとして表示されます。精度の小数桁数は異なる場合があります。</li> </ul> <p><b>TO_POSIXTIME</b> 関数または TOPOSIXTIME() メソッドを使用して、%TimeStamp 値</p>

データ型クラス	対応する SQL タイプ	留意事項
上記の論理値のいずれにも該当しない時刻データ型	TIME	を %PosixTime に変換できます。IsValid() メソッドを使用することで、数値が有効な %PosixTime 値であるかどうかを判定できます。
上記の論理値のいずれにも該当しないタイムスタンプ・データ型	TIMESTAMP	このクラスを作成するときは、論理時間値を %Library.Time 論理値に変換する LogicalToTime() メソッドと、それとは逆方向に変換する TimeToLogical() メソッドを定義します。
上記の論理値のいずれにも該当しないタイムスタンプ・データ型	TIMESTAMP	このクラスを定義するときは、論理タイムスタンプ値を %Library.TimeStamp 論理値に変換する LogicalToTimeStamp() メソッドと、それとは逆方向に変換する TimeStampToLogical() メソッドを定義します。

演算子 =、<、>、または < を使用して、POSIXTIME を DATE 値または TIMESTAMP 値と比較できます。詳細は [“述語の概要”](#) を参照してください。

InterSystems IRIS では、FMTIMESTAMP カテゴリ値と DATE カテゴリ値を比較する場合に、DATE と比較する前に FMTIMESTAMP 値から時間を削除することはありません。この動作は、TIMESTAMP と DATE の値の比較、および TIMESTAMP と MVDATA の値の比較に対する動作と同じになりました。これは、他の SQL ベンダによるタイムスタンプと日付の比較方法とも互換性があります。つまり、FMTIMESTAMP 320110202.12 と DATE 62124 の比較は、SQL等値 (=) 演算子を使用した比較では等しくなります。アプリケーションでは、FMTIMESTAMP 値を DATE または FMDATE 値に変換して値の日付部分のみを比較する必要があります。

#### 1840 年 12 月 31 日より前の日付

日付は一般的に、DATE データ型、または TIMESTAMP データ型で表されます。

DATE データ型は、\$HOROLOG 形式で日付を格納し、任意の開始日 1840 年 12 月 31 日以降の日数を正の整数値としてカウントします。既定では、日付は正の整数値でのみ表すことができます (MINVAL=0。これは 1840 年 12 月 31 日に相当します)。ただし、%Library.Date MINVAL 型パラメータを変更することで、1840 年 12 月 31 日より前の日付の格納を可能にできます。MINIVAL を負の数値に設定すると、1840 年 12 月 31 日より前の日付を負の整数値として格納できます。使用可能な最も前の MINIVAL 値は -672045 です。これは西暦 1 年 (CE) 1 月 1 日に対応します。DATE データ型で BCE (紀元前) の日付を表すことはできません。

TIMESTAMP データ型は、既定で、許容される最も早いタイムスタンプである 1840-12-31 00:00:00 となります。ただし、MINVAL パラメータを変更すれば、1840 年 12 月 31 日より前の日付を保存できるフィールドまたはプロパティを定義できます (例: MyTS %Library.TimeStamp(MINVAL='1492-01-01 00:00:00')). 許容される最も早い MINVAL 値は 0001-01-01 00:00:00 です。これは西暦 1 年 1 月 1 日に相当します。%TimeStamp データ型で BCE (紀元前) の日付を表すことはできません。

注釈 これらの日付のカウントでは、グレゴリオ暦の導入 (1582 年 10 月に制定、ただし英国およびその植民地で採用されたのは 1752 年) による日付の変更が考慮されないことに注意してください。

ロケールの最小の日付は、以下のように再定義できます。



## ObjectScript

```
SET oldMinDate = ##class(%SYS.NLS.Format).GetFormatItem("DATEMINIMUM")
IF oldMinDate=0 {
    DO ##class(%SYS.NLS.Format).SetFormatItem("DATEMINIMUM",-672045)
    SET newMinDate = ##class(%SYS.NLS.Format).GetFormatItem("DATEMINIMUM")
    WRITE "Changed earliest date to ",newMinDate
}
ELSE { WRITE "Earliest date was already reset to ",oldMinDate}
```

上記の例では、ロケールの MINIVAL を、使用可能な最も早い日付 (1/1/01) に設定します。ロケールに基づく日付構成の詳細は、“[各国言語サポート \(NLS\) の構成](#)” を参照してください。

注釈 InterSystems IRIS は、負の論理 DATE 値 (MINVAL<0 の %Library.Date 値) による[ユリウス日](#)の使用をサポートしていません。このため、このような MINVAL<0 値には、[TO\\_CHAR](#) 関数で返されるユリウス日付形式との互換性がありません。

## 文字列

InterSystems IRIS では、文字列を扱うのに固定量のメモリを許可しているため、[文字列長の制限](#)があります。一般に、極端に長い文字列には [%Stream.GlobalCharacter](#) データ型のいずれかを割り当てる必要があります。

データベース・ドライバの接続には、文字列長の制限が適用されません。InterSystems IRIS インスタンスと ODBC ドライバ機能でサポートされているプロトコルが異なる場合は、2 つのプロトコルの中で下位のものが使用されます。実際に使用されたプロトコルは、InterSystems ODBC ログに記録されます。

既定では、InterSystems IRIS はシステム全体の ODBC VARCHAR の最大長を 4096 に設定します。この [ODBC の最大長は構成可能です](#)。

## リスト構造

InterSystems IRIS は、リスト構造のデータ型 %List (データ型クラス %Library.List) をサポートしています。これは圧縮バイナリ形式であり、InterSystems SQL で対応するネイティブなデータ型にマップしません。その内部表現では、データ型 VARBINARY に対応しており、その MAXLEN の既定値は 32749 です。InterSystems IRIS は、リスト構造データ型 %ListOfBinary (データ型クラス %Library.ListOfBinary) をサポートしています。これはデータ型 VARBINARY に対応しており、その MAXLEN の既定値は 4096 です。

このため、[ダイナミック SQL](#) では、WHERE 節での比較に %List データを使用できません。また、型 %List のプロパティ値を設定するときに、INSERT も UPDATE も使用できません。

ダイナミック SQL では、リスト構造化されたデータのデータ型が VARCHAR として返されます。クエリ内のフィールドのデータ型が %List または %ListOfBinary かどうかを確認するには、[select-item 列メタデータ](#)のブーリアン・フラグ [isList](#) を使用できます。これらのデータ型の CType (クライアント・データ型) 整数コードは 6 です。

ODBC クライアントまたは JDBC クライアントを使用する場合には、LogicalToOdbc 変換を使用して、%List データが VARCHAR 文字列データに投影されます。リストは、各要素がコンマで区切られた文字列として投影されます。この型のデータは、WHERE 節、INSERT 文および UPDATE 文で使用できます。既定では、InterSystems IRIS はシステム全体の ODBC VARCHAR の最大長を 4096 に設定します。この [ODBC の最大長は構成可能です](#)。

“%Library.List” でこのクラスの詳細も参照してください。WHERE 節でのリストの使用に関する詳細は、“[%INLIST](#)” 述語および “[FOR SOME %ELEMENT](#)” 述語の説明を参照してください。リスト・データを文字列として処理する方法の詳細は、“[%EXTERNAL](#)” 関数を参照してください。

InterSystems SQL は、[\\$LIST](#)、[\\$LISTBUILD](#)、[\\$LISTDATA](#)、[\\$LISTFIND](#)、[\\$LISTFROMSTRING](#)、[\\$LISTGET](#)、[\\$LISTLENGTH](#)、[\\$LISTTOSTRING](#) の 8 つのリスト関数をサポートしています。ObjectScript は、この他に 3 つのリスト関数をサポートしています。[\\$LISTVALID](#) は式がリストであるかどうかを判断するときに、[\\$LISTSAME](#) は 2 つのリストを比較するときに、[\\$LISTNEXT](#) はリストから要素を順番に取得するときに使用します。

## BIT データ型

BIT (%Library.Boolean) データ型では、0、1、および NULL を有効な値として使用できます。

- ・ 論理モードおよび ODBC モードでは、使用できる値は 0、1、および NULL だけです。
- ・ 表示モードでは、DisplayToLogical メソッドによって、最初に非 NULL の入力値を以下のように 0 または 1 に変換します。
  - ゼロ以外の数値または数値文字列 = 1。3、'0.1'、'-1'、'7dwarves' など。
  - 非数値文字列 = 0。'true'、'false' など。
  - 空の文字列 = 0。'' など。

## ストリーム・データ型

ストリーム・データ型は、InterSystems IRIS クラス・プロパティのデータ型 `%Stream.GlobalCharacter` (CLOB の場合) および `%Stream.GlobalBinary` (BLOB の場合) に対応します。これらのデータ型クラスは、指定された [LOCATION パラメータ](#) でストリーム・フィールドを定義するか、このパラメータを省略して既定のシステム定義のストレージ位置にすることができます。

ストリーム・データ型のフィールドは、ほとんどの SQL スカラ関数、集約関数、または単項関数で引数として使用できません。削除しようとすると、SQLCODE -37 エラー・コードが発行されます。例外であるいくつかの機能は、“[ストリーム・データ \(BLOB と CLOB\) の格納と使用](#)” にリストされています。

ストリーム・データ型のフィールドは、ほとんどの SQL 述語条件で引数として使用できません。削除しようとすると、SQLCODE -313 エラー・コードが発行されます。ストリーム・フィールドを受け取る述語は、“[ストリーム・データ \(BLOB と CLOB\) の格納と使用](#)” にリストされています。

[シャード・テーブル](#)にストリーム・データ型のフィールドを含めることはできません。

インデックス内、および挿入/更新の実行時におけるストリーム・データ型の使用も制限されています。ストリームの制限の詳細は、“[ストリーム・データ \(BLOB と CLOB\) の格納と使用](#)” を参照してください。

## SERIAL データ型

SERIAL (`%Library.Counter`) データ型のフィールドでは、ユーザ指定の正の整数値を使用できます。または、InterSystems IRIS では、連続する正の整数値をそのフィールドに割り当てることができます。`%Library.Counter` は `%Library.BigInt` を拡張します。

INSERT 操作では、SERIAL フィールドに以下のいずれかの値が指定されます。

- ・ 値なし、0 (ゼロ)、または非数値: InterSystems IRIS では、指定された値が無視され、その代わり、このフィールドの現在のシリアル・カウンタ値が 1 ずつインクリメントされ、インクリメント後の整数がフィールドに挿入されます。
- ・ 正の整数値: InterSystems IRIS では、フィールドにユーザ指定の値が挿入され、このフィールドのシリアル・カウンタ値がこの整数値に変更されます。

したがって、SERIAL フィールドには、一連のインクリメンタル整数値が含まれます。これらの値は、連続または一意とは限りません。例えば、SERIAL フィールドでは、1、2、3、17、18、25、25、26、27 といった一連の値が有効です。連続する整数は、InterSystems IRIS で生成されるか、またはユーザが指定します。連続しない整数はユーザが指定します。SERIAL フィールドの値を一意にする場合には、そのフィールドに UNIQUE 制約を適用する必要があります。

UPDATE 操作は、自動的に割り当てられた SERIAL カウンタ・フィールドの値に影響しません。ただし、[INSERT OR UPDATE](#) を使用して実行される更新では、SERIAL フィールドに対する後続の挿入操作で整数シーケンスのスキップが発生します。

UPDATE 操作では、SERIAL フィールドに現在、値がない (NULL) 場合、またはその値が 0 の場合にのみ、そのフィールドを変更できます。それ以外の場合には、SQLCODE -105 エラーが生成されます。

InterSystems IRIS では、テーブル内の SERIAL フィールドの数には制約がありません。

## ROWVERSION データ型

ROWVERSION データ型は、システムによって割り当てられる、1 で始まる一意の正の整数を含む読み取り専用フィールドを定義します。InterSystems IRIS は、挿入、更新、または %Save の各処理の一部として連続した整数を割り当てます。これらの値をユーザが変更することはできません。

InterSystems IRIS は、ネームスペース全体で 1 つの行バージョン・カウンタを維持します。ROWVERSION フィールドを含むネームスペース内のすべてのテーブルが、同じ行バージョン・カウンタを共有します。このように、ROWVERSION フィールドでは行レベルのバージョン・コントロールが可能なため、ユーザはネームスペース内の 1 つ以上のテーブルの行に対して行った変更の順序を決定できます。

テーブルごとに指定できる ROWVERSION データ型のフィールドは 1 つのみです。

一意キーや主キーに ROWVERSION フィールドを含めることはできません。ROWVERSION フィールドを IDKey インデックスの一部にすることはできません。

ROWVERSION の使用の詳細は、“[RowVersion フィールド](#)” を参照してください。

## ROWVERSION カウンタと SERIAL カウンタ

ROWVERSION と SERIAL (%Library.Counter) データ型フィールドは両方とも、INSERT 操作の一部として内部カウンタから連続した整数を受け取ります。ただし、これらの 2 つのカウンタは大きく異なるうえ、使用目的も異なります。

- ・ ROWVERSION カウンタはネームスペース・レベルにあります。SERIAL カウンタはテーブル・レベルにあります。これらの 2 つのカウンタは相互から完全に独立しており、RowID カウンタからも独立しています。
- ・ ROWVERSION カウンタは挿入、更新、または %Save 操作によってインクリメントされます。SERIAL カウンタは挿入操作によってのみインクリメントされます。INSERT OR UPDATE を使用して実行される更新によって、SERIAL カウンタ・シーケンスにギャップが生じる可能性があります。
- ・ ROWVERSION フィールドの値をユーザ指定にすることはできません。この値には常に ROWVERSION カウンタから指定されます。ユーザが SERIAL フィールドの値を指定していない場合、このフィールドの値には、挿入操作中のテーブルの内部カウンタの値が指定されます。挿入によって SERIAL の整数値が指定される場合、現在のカウンタ値ではなくその値が挿入されます。
  - 挿入によって現在の内部カウンタ値よりも大きい SERIAL フィールドの値が指定される場合、InterSystems IRIS はその値をフィールドに挿入し、内部カウンタをその値にリセットします。
  - 挿入によって現在のカウンタ値よりも小さい SERIAL フィールドの値が指定されると、InterSystems IRIS は内部カウンタをリセットしません。
  - 挿入によって SERIAL フィールドの値を負の整数または小数に指定できます。InterSystems IRIS は、小数を切り捨てて、その整数部分のみにします。指定された SERIAL フィールドの値が 0 または NULL の場合、InterSystems IRIS はユーザ指定の値を無視し、現在の内部カウンタ値を挿入します。

既存の SERIAL フィールドの値を更新することはできません。

- ・ ROWVERSION フィールドの値は常に一意です。ユーザ指定の SERIAL フィールドの値の挿入が可能なため、一意の SERIAL フィールドの値を保証するには、UNIQUE フィールド制約を指定する必要があります。
- ・ ROWVERSION カウンタはリセットできません。TRUNCATE TABLE は SERIAL カウンタをリセットします。すべての行に対して DELETE を実行しても SERIAL カウンタはリセットされません。
- ・ ROWVERSION フィールドは、テーブルごとに 1 つのみ許可されます。SERIAL フィールドは、1 つのテーブルに複数指定できます。

## InterSystems ODBC / JDBC で使用される DDL データ型

InterSystems ODBC は、DDL データ型のサブセットを使用して、このデータ型のサブセットに、その他のデータ型をマップします。これらのマッピングを元に戻すことはできません。例えば、CREATE TABLE mytable (f1 BINARY) 文は、

mytable (f1 VARBINARY) として ODBC に投影される InterSystems IRIS クラスを作成します。InterSystems IRIS のリスト・データ型は、VARCHAR 文字列として ODBC に投影されます。

ODBC は、BIGINT、BIT、DATE、DOUBLE、GUID、INTEGER、LONGVARBINARY、LONGVARCHAR、NUMERIC、OREF、POSIXTIME、SMALLINT、TIME、TIMESTAMP、TINYINT、VARBINARY、VARCHAR のデータ型を公開します。既定では、InterSystems IRIS はシステム全体の ODBC VARCHAR の最大長を 4096 に設定します。この [ODBC の最大長は構成可能です](#)。

これらの ODBC/JDBC データ型の値の 1 つが InterSystems SQL にマップされると、以下の演算が実行されます。DOUBLE データが [\\$DOUBLE](#) を使用してキャストされます。NUMERIC データが [\\$DECIMAL](#) を使用してキャストされます。

GUID データ型は、InterSystems SQL の UNIQUEIDENTIFIER データ型に相当します。GUID / UNIQUEIDENTIFIER フィールドに有効な値を指定できないと、#7212 一般エラーが生成されます。GUID 値を生成するには、%SYSTEM.Util.CreateGUID() メソッドを使用します。

## VECTOR

InterSystems SQL では、特定の最適化に非常に適した、圧縮形式でデータを格納する VECTOR 型をサポートしています。ベクトルは型が指定され、ベクトル内のすべてのデータはそのベクトルの型で格納および書式設定されます。ベクトルの型は、integer、decimal、double にすることができます。

VECTOR 型には、以下の 3 つのコンストラクタがあります。

```
VECTOR(type, length)
VECTOR(type)
VECTOR
```

**注釈** CREATE TABLE 文で VECTOR 型の列を定義する場合、その型のコンストラクタでベクトルの長さを指定することをお勧めします。長さが指定されている場合、その列に別の長さのベクトルが追加されると、システムはエラーをスローし、列内のすべてのベクトルが同じ長さになるようにします。この均一性により、[VECTOR\\_DOT\\_PRODUCT](#) や [VECTOR\\_COSINE](#) などのベクトル関数は、その列のすべてのベクトルで正常に実行されるか、その列のすべてのベクトルで失敗することが保証されます。

## データ型間の変換

特定の型から別の型にデータ型を変換するには、[CAST](#) または [CONVERT](#) 関数を使用します。

CAST は、一部の文字列データ型と数値データ型に加えて、DATE、TIME、および TIMESTAMP と POSIXTIME のタイムスタンプ・データ型への変換をサポートします。

CONVERT には、2 とおりの構文形式があります。両方の形式が、DATE、TIME、および TIMESTAMP と POSIXTIME のタイムスタンプ・データ型との相互変換、およびその他のデータ型間での変換をサポートします。

VARCHAR に値を CAST または CONVERT する場合、マッピングの既定サイズは 30 文字です。サイズの指定がない VARCHAR を MAXLEN 1 にマッピングする場合でも同様です。この既定サイズの 30 文字は、InterSystems IRIS 以外のソフトウェア要件との互換性を保持するために提供されています。

## データ型の優先順位

操作によって返すことができる値が複数存在し、それらの値のデータ型が複数にわたる場合は、InterSystems IRIS により、返り値に優先順位の最も高いデータ型が割り当てられます。例えば、NUMERIC データ型には INTEGER データ型として考えられる値がすべて含まれますが、INTEGER データ型には NUMERIC データ型として考えられる値がすべて含まれるわけではありません。したがって、NUMERIC の方が優先順位が上であり、INTEGER よりも包括的です。

例えば、[CASE](#) 文で返すことができる結果値のデータ型として INTEGER と NUMERIC がある場合、結果値がこのどちらであっても、実際に返される結果値のデータ型は必ず NUMERIC になります。

データ型の優先順位は、高いもの（最も包括的）から低いものの順に以下のとおりです。

LONGVARBINARY  
 LONGVARCHAR  
 VARBINARY  
 VARCHAR  
 GUID  
 TIMESTAMP  
 DOUBLE  
 NUMERIC  
 BIGINT  
 INTEGER  
 DATE  
 TIME  
 SMALLINT  
 TINYINT  
 BIT

## データ型の正規化および検証

%Library.DataType スーパークラスには、特定のデータ型のサブクラスがあります。これらのデータ型クラスには、入力値をデータ型形式に正規化するための Normalize() メソッド、入力値がそのデータ型に有効かどうかを判断するための IsValid() メソッド、および LogicalToDisplay() や DisplayToLogical() などのさまざまなモード変換メソッドが用意されています。

以下の例では、%TimeStamp データ型のための Normalize() メソッドを示しています。

### ObjectScript

```
SET indate=64701
SET tsdate=##class(%Library.TimeStamp).Normalize(indate)
WRITE "%TimeStamp date: ",tsdate
```

### ObjectScript

```
SET indate="2018-2-22"
SET tsdate=##class(%Library.TimeStamp).Normalize(indate)
WRITE "%TimeStamp date: ",tsdate
```

以下の例では、%TimeStamp データ型のための IsValid() メソッドを示しています。

### ObjectScript

```
SET datestr="July 4, 2018"
SET stat=##class(%Library.TimeStamp).IsValid(datestr)
IF stat=1 {WRITE datestr," is a valid %TimeStamp",! }
ELSE {WRITE datestr," is not a valid %TimeStamp",!}
```

### ObjectScript

```
SET leapdate="2016-02-29 00:00:00"
SET noleap="2018-02-29 00:00:00"
SET stat=##class(%Library.TimeStamp).IsValid(leapdate)
IF stat=1 {WRITE leapdate," is a valid %TimeStamp",! }
ELSE {WRITE leapdate," is not a valid %TimeStamp",!}
SET stat=##class(%Library.TimeStamp).IsValid(noleap)
IF stat=1 {WRITE noleap," is a valid %TimeStamp",! }
ELSE {WRITE noleap," is not a valid %TimeStamp",!}
```

## クエリ・メタデータを使用して返されるデータ型

ダイナミック SQL を使用すると、クエリ内で指定した列のデータ型など、クエリに関するメタデータが返されます。

以下のダイナミック SQL の例は、Sample.Person および Sample.Employee 内の各列に対する ODBC データ型の列名および整数コードを返します。



## ObjectScript

```

SET myquery="SELECT * FROM Sample.Person"
SET tStatement=##class(%SQL.Statement).%New()
SET tStatus=tStatement.%Prepare(myquery)
SET x=tStatement.%Metadata.columnCount
WHILE x>0 {
    SET column=tStatement.%Metadata.columns.GetAt(x)
    WRITE !,x," ",column.colName," ",column.ODBCType
    SET x=x-1 }
WRITE !,"end of columns"

```

## ObjectScript

```

SET myquery="SELECT * FROM Sample.Employee"
SET tStatement=##class(%SQL.Statement).%New()
SET tStatus=tStatement.%Prepare(myquery)
SET x=tStatement.%Metadata.columnCount
WHILE x>0 {
    SET column=tStatement.%Metadata.columns.GetAt(x)
    WRITE !,x," ",column.colName," ",column.ODBCType
    SET x=x-1 }
WRITE !,"end of columns"

```

ODBC では ObjectScript の %List データ型値をコンマで区切られた値の文字列で表すので、リスト構造化されたデータ (Sample.Person 内の FavoriteColors 列など) は、データ型 12 (VARCHAR) を返します。

ストリーム・データ (Sample.Employee 内の Notes 列や Picture 列など) は、データ型 -1 (LONGVARCHAR) または -4 (LONGVARBINARY) を返します。

%Library.RowVersion は %Library.BigInt のサブクラスであるため、ROWVERSION フィールドはデータ型 -5 を返します。

詳細は、“[ダイナミック SQL の使用](#)” および “[%SQL.Statement](#)” を参照してください。

## データ型の整数コード

クエリ・メタデータなどのコンテキストでは、列に定義されたデータ型が整数コードで返される場合があります。CType (クライアント・データ型) 整数コードは、%SQL.StatementColumn clientType プロパティにリストされます。詳細は、“[select-item のメタデータ](#)” を参照してください。

SQLType データ型コードは ODBC と JDBC で使用されます。前述の例で示すように、ODBC データ型コードは %SQL.Statement.%Metadata.columns.GetAt() メソッドで返されます。[SQL シェル・メタデータ](#)は ODBC データ型コードも返します。JDBC のコードと ODBC のコードはほぼ同じですが、日付と時刻のデータ型の表現が異なります。これらの ODBC および JDBC の値は以下のとおりです。

ODBC	JDBC	データ型
-11	-11	GUID
-7	-7	BIT
-6	-6	TINYINT
-5	-5	BIGINT
-4	-4	LONGVARBINARY
-3	-3	VARBINARY
-2	-2	BINARY
-1	-1	LONGVARCHAR
0	0	不明な型
1	1	CHAR
2	2	NUMERIC

3	3	DECIMAL
4	4	INTEGER
5	5	SMALLINT
6	6	FLOAT
7	7	REAL
8	8	DOUBLE
9	91	DATE
10	92	TIME
11	93	TIMESTAMP
12	12	VARCHAR

詳細は、“[ダイナミック SQL の使用](#)”を参照してください。

InterSystems IRIS は、中国語、ヘブライ語、日本語、韓国語のロケールなど、マルチバイトの文字セットを使用する ODBC アプリケーション向けに Unicode SQL タイプもサポートします。

ODBC	データ型
-10	WLONGVARCHAR
-9	WVARCHAR

この機能を有効にする方法については、“[ODBC データ・ソースとしての InterSystems データベースの使用法 \(Windows\)](#)”を参照してください。

## 関連項目

- ・ [CAST、CONVERT](#)
- ・ [TO\\_CHAR、TO\\_DATE、TO\\_NUMBER、TO\\_VECTOR](#)

## 日付/時刻文 (SQL)

ODBC の日付、時刻、またはタイムスタンプを検証して変換します。

### 構文

```
{d 'yyyy-mm-dd'}
{d nnnnnn}

{t 'hh:mm:ss[.fff]'}
{t nnnnn.nnn}

{ts 'yyyy-mm-dd [hh:mm:ss.fff]'}
{ts 'mm/dd/yyyy [hh:mm:ss.fff]'}
{ts nnnnnnn}
```

### 概要

これらの文は整数または ODBC の日付形式、時刻形式、またはタイムスタンプ形式の文字列を受け取り、それを対応する InterSystems IRIS の日付形式、時刻形式、またはタイムスタンプ形式に変換します。これらの文は、データ形式の決定と、値および範囲のチェックを行います。

#### [d 'string']

[d 'string'] 日付文は、ODBC 形式の日付を検証します。この日付が有効である場合は、1840-12-31 以降の整数カウント値として、(論理モードで) InterSystems IRIS [\\$HOROLOG](#) 日付形式で格納します。InterSystems IRIS によって、既定の時刻値が追加されることはありません。1840-12-31 より前の日付をサポートするには、テーブル内にデータ型 %Library.Date(MINVAL=-nnn) の日付フィールドを定義する必要があります。MINVAL は、1840-12-31 (0 日目) から最大値 -672045 (0001-01-01) まで遡ってカウントする負の日数です。

以下のように指定するとします。

- -672045 (0001-01-01) 未満または 2980013 (9999-12-31) より大きい整数は、SQLCODE -400 <VALUE OUT OF RANGE> エラーとなります。
- 無効な日付 (ODBC 形式でない日付やうるう年以外の 02-29 の日付など) : InterSystems IRIS は SQLCODE -146 エラー: “yyyy-mm-dd' ODBC/JDBC ” を生成します。
- ODBC タイムスタンプ値: InterSystems IRIS はタイムスタンプの日付部分と時刻部分の両方を検証します。両方が有効な場合は、日付部分のみが変換されます。日付と時刻のいずれかが無効な場合、システムは SQLCODE -146 エラーを生成します。

#### [t 'string']

[t 'string'] 時刻文は、ODBC 形式の時刻を検証します。この時刻が有効である場合は、深夜 0 時以降の秒数の整数カウント値として、(論理モードで) 指定の小数部と共に InterSystems IRIS [\\$HOROLOG](#) 時刻形式で格納します。InterSystems IRIS 表示モードおよび ODBC モードでは、秒の小数部は表示されません。秒の小数部は、これらの表示形式から切り捨てられます。

以下のように指定するとします。

- 0 (00:00:00) 未満または 86399.99 (23:59:59.99) より大きい整数は、SQLCODE -400 <ILLEGAL VALUE> エラーとなります。
- 無効な時刻 (ODBC 形式でない時刻や 23 より大きな時間) : InterSystems IRIS は SQLCODE -147 エラー: “hh:mi:ss.fff' ODBC/JDBC ” を生成します。
- ODBC タイムスタンプ値: InterSystems IRIS は SQLCODE -147 エラーを生成します。



## {ts 'string'}

{ts 'string'} タイムスタンプ文は、日付/時刻を検証し、ODBC タイムスタンプ形式で返します。指定された秒の小数部が常に保持され、表示されます。また、{ts 'string'} タイムスタンプ文は、日付も検証し、時刻値 00:00:00 を指定して ODBC タイムスタンプ形式で返します。

以下のように指定するとします。

- ・ 正または負の整数日付 (-672045 ~ 2980013): InterSystems IRIS は時刻値として 00:00:00 を追加してから、生成されたタイムスタンプを ODBC 形式で格納します。例えば、64701 の場合は 2018/02/22 00:00:00 が返されます。これは有効な \$HOROLOG 日付整数です。\$HOROLOG 0 は 1840-12-31 です。
- ・ ODBC 形式の有効なタイムスタンプ: InterSystems IRIS は指定された値を変更せずに格納します。これは InterSystems IRIS のタイムスタンプ形式が ODBC タイムスタンプ形式と同じためです。
- ・ ロケールの既定の日付形式および時刻形式を使用した有効なタイムスタンプ (例、2/29/2016 12:23:46.77) : InterSystems IRIS は指定された値を ODBC 形式で格納して表示します。
- ・ 無効なタイムスタンプ (うるう年以外で日付部分に 02-29 を指定したタイムスタンプや、時刻部分で 23 より大きな時間が指定されたタイムスタンプなど) : InterSystems IRIS は値として文字列 “error” を返します。
- ・ 有効な日付 (ODBC 形式またはロケール形式) で時刻指定なし: InterSystems IRIS は時刻値として 00:00:00 を追加してから、生成されたタイムスタンプを ODBC 形式で格納します。必要に応じて、先行のゼロが指定されます。例えば、2/29/2016 の場合は 2016/02/29 00:00:00 が返されます。
- ・ 形式は正しいが無効な日付 (ODBC 形式またはロケール形式) で時刻指定なし: InterSystems IRIS は時刻値として 00:00:00 を追加します。その後、指定に従って日付部分を格納します。例えば、02/29/2019 の場合は 02/29/2019 00:00:00 が返されます。
- ・ 形式が正しくない無効な日付 (ODBC 形式、ロケール形式、または \$HOROLOG 形式) で時刻指定なし: InterSystems IRIS は文字列 “error” を返します。例えば、2/29/2019 (先行のゼロはなく、日付値は無効) の場合は、“error” が返されます。00234 (先行のゼロがある \$HOROLOG) では、“error” が返されます。

詳細は、“\$HOROLOG” を参照してください。

## 例

次のダイナミック SQL の例は、ODBC 形式で指定された日付 (先行にゼロがある場合もない場合も) を検証し、\$HOROLOG 相当値 64701 として格納します。この例では、%SelectMode 0 (論理) 値が表示されます。

### ObjectScript

```
SET myquery = 2
SET myquery(1) = "SELECT {d '2018-02-22'} AS date1,"
SET myquery(2) = "{d '2018-2-22'} AS date2"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

次のダイナミック SQL の例は、ODBC 形式で指定された時刻 (先行にゼロがある場合もない場合も) を検証し、\$HOROLOG 相当値 43469 として格納します。この例では、%SelectMode 0 (論理) 値が表示されます。

## ObjectScript

```

SET myquery = 3
SET myquery(1) = "SELECT {t '12:04:29'} AS time1,"
SET myquery(2) = "{t '12:4:29'} AS time2,"
SET myquery(3) = "{t '12:04:29.00000'} AS time3"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()

```

次のダイナミック SQL の例は、ODBC 形式で指定され、秒の小数部のある時刻を検証し、\$HOROLOG 相当値 43469 として秒の小数部を追加して格納します。末尾のゼロは切り捨てられます。この例では、%SelectMode 0 (論理) 値が表示されます。

## ObjectScript

```

SET myquery = 3
SET myquery(1) = "SELECT {t '12:04:29.987'} AS time1,"
SET myquery(2) = "{t '12:4:29.987'} AS time2,"
SET myquery(3) = "{t '12:04:29.987000'} AS time3"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()

```

次のダイナミック SQL の例は、いくつかの形式による時刻および日付の値を検証し、相当する ODBC タイムスタンプとして格納します。必要に応じて時刻値 00:00:00 が指定されます。この例では、%SelectMode 0 (論理) 値が表示されます。

## ObjectScript

```

SET myquery = 6
SET myquery(1) = "SELECT {ts '2018-02-22 01:43:38'} AS ts1,"
SET myquery(2) = "{ts '2018-02-22'} AS ts2,"
SET myquery(3) = "{ts '02/22/2018 01:43:38.999'} AS ts3,"
SET myquery(4) = "{ts '2/22/2018 01:43:38'} AS ts4,"
SET myquery(5) = "{ts '02/22/2018'} AS ts5,"
SET myquery(6) = "{ts '64701'} AS ts6"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
IF rset.%Next() {
WRITE rset.ts1,!
WRITE rset.ts2,!
WRITE rset.ts3,!
WRITE rset.ts4,!
WRITE rset.ts5,!
WRITE rset.ts6
}

```

## 既定のユーザ名とパスワード (SQL)

---

既定のログイン識別を提供します。

### 概要

InterSystems IRIS® データ・プラットフォームには、データベースにログインして操作を開始するための既定のユーザ名とパスワードが用意されています。既定のユーザ名は “\_SYSTEM” (大文字)、パスワードは “SYS” です。

## SQLCODEのエラー・コード

---

SQL エラー・コード。

### 説明

ほとんどの InterSystems SQL 操作は、実行を試みると SQLCODE 値が発行されます。発行される SQLCODE 値は、0、100、および負の整数値です。

- ・ SQLCODE=0 は、SQL 処理が正常に終了したことを示します。SELECT 文の場合、通常、これはテーブルからのデータの取得に成功したことを意味します。ただし、SELECT で[集約演算](#) (SELECT SUM(myfield) など) を実行する場合、myfield にデータがなくても集約演算は成功し、SQLCODE=0 が発行されます。この場合、SUM は NULL を返し、%ROWCOUNT=1 となります。
- ・ SQLCODE=100 は、SQL 操作は成功したが処理するデータが見つからないことを意味します。これは、さまざまな理由で発生します。SELECT の場合、指定したテーブルにデータがない、テーブルにクエリ条件を満たすデータがない、行の検索がテーブルの最終行に達した、などの理由があります。UPDATE または DELETE の場合、指定したテーブルにデータがない、テーブルに WHERE 節の条件を満たすデータの行がない、などの理由があります。このような場合、%ROWCOUNT=0 となります。
- ・ SQLCODE=-n はエラーを示します。負の整数値は、発生したエラーの種類を指定します。SQLCODE=-400 は汎用の深刻なエラー・コードです。

SQLCODE エラー・コードと対応するエラー・メッセージの詳細、および SQLCODE エラー・コード値のリストは、“[SQL エラー・メッセージ](#)” を参照してください。

# フィールド制約

フィールド内容の規則を指定します。

## 概要

フィールド制約は、フィールドに指定可能なデータ値を規定する規則を指定します。フィールドには以下の制約を指定できます。

- ・ NOT NULL : このフィールドにはすべてのレコードで値を指定する必要があります (空文字列も指定可能)。
- ・ UNIQUE : レコードでこのフィールドに値を指定する場合は、一意の値である必要があります (1 つの空文字列も指定可能)。ただし、このフィールドに値を持たない (NULL) レコードは複数作成できます。
- ・ DEFAULT : このフィールドには、すべてのレコードで必ず値を指定する必要があり、指定がなければ InterSystems IRIS で既定値が指定されます (空文字列も指定可能)。既定値は、NULL、空文字列、またはそのデータ型に適切なその他の任意の値にできます。
- ・ UNIQUE NOT NULL : このフィールドにはすべてのレコードで一意の値を指定する必要があります (1 つの空文字列も指定可能)。主キーとして使用できます。
- ・ DEFAULT NOT NULL : このフィールドには、すべてのレコードで必ず値を指定する必要があり、指定がなければ InterSystems IRIS で既定値が指定されます (空文字列も指定可能)。
- ・ UNIQUE DEFAULT : 非推奨 – このフィールドには、すべてのレコードで必ず一意の値を指定する必要があり、指定がなければ InterSystems IRIS で既定値が指定されます (1 つの空文字列も指定可能)。既定値は、NULL、空文字列、またはそのデータ型に適切なその他の任意の値にできます。既定値が一意に生成される値 (CURRENT\_TIMESTAMP など) であるか、一度しか使用されないことがわかっている場合にのみ使用してください。
- ・ UNIQUE DEFAULT NOT NULL : 非推奨 – このフィールドには、すべてのレコードで必ず一意の値を指定する必要があり、指定がなければ InterSystems IRIS で既定値が指定されます (1 つの空文字列も指定可能)。既定値は、空文字列、またはそのデータ型に適切なその他の任意の値にできます。NULL にすることはできません。既定値が一意に生成される値 (CURRENT\_TIMESTAMP など) であるか、一度しか使用されないことがわかっている場合にのみ使用してください。主キーとして使用できます。
- ・ IDENTITY : このフィールドにはすべてのレコードで、システム生成され変更できない一意の整数値が InterSystems IRIS によって指定されます。その他のフィールド制約キーワードは無視されます。主キーとして使用できます。

データ値はフィールドのデータ型に適切なものである必要があります。空文字列は数値フィールドでは指定できない値です。

これらのフィールド制約の詳細は、“[CREATE TABLE](#)” コマンドのページで説明されています。

# 予約語 (SQL)

InterSystems IRIS® データ・プラットフォームの SQL 予約語のリスト。

## 構文

```
%AFTERHAVING | %ALLINDEX | %ALPHAUP | %ALTER | %BEGTRANS |
%CHECKPRIV | %CLASSNAME | %CLASSPARAMETER | %DEBUGFULL | %DELDATA |
%DESCRIPTION | %EXACT | %EXTERNAL | %FILE | %FIRSTTABLE | %FLATTEN |
%FOREACH | %FULL | %ID | %IDADDED | %IGNOREINDEX | %IGNOREINDICES |
%INLIST | %INORDER | %INTERNAL | %INTEXT | %INTRANS | %INTRANSACTION |
%KEY | %MATCHES | %MCODE | %MERGE | %MINUS | %MVR | %NOCHECK |
%NODELDATA | %NOFLATTEN | %NOFLPLAN | %NOINDEX | %NOLOCK |
%NOMERGE | %NOPARALLEL | %NOREDUCE | %NORUNTIME | %NOSVSO | %NOTOPOPT |
%NOTRIGGER | %NOUNIONROPT | %NUMROWS | %ODBCIN | %ODBCOUT |
%PARALLEL | %PLUS | %PROFILE | %PROFILE_ALL | %PUBLICROWID | %ROUTINE |
%ROWCOUNT | %RUNTIMEIN | %RUNTIMEOUT | %STARTSWITH |
%STARTTABLE | %SQLSTRING | %SQLUPPER | %STRING | %TABLENAME |
%TRUNCATE | %UPPER | %VALUE | %VID
ABSOLUTE | ADD | ALL | ALLOCATE | ALTER | AND | ANY | ARE | AS |
ASC | ASSERTION | AT | AUTHORIZATION | AVG | BEGIN | BETWEEN |
BIT | BIT_LENGTH | BOTH | BY | CASCADE | CASE | CAST |
CHAR | CHARACTER | CHARACTER_LENGTH | CHAR_LENGTH |
CHECK | CLOSE | COALESCE | COLLATE | COMMIT | CONNECT |
CONNECTION | CONSTRAINT | CONSTRAINTS | CONTINUE | CONVERT |
CORRESPONDING | COUNT | CREATE | CROSS | CURRENT |
CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP |
CURRENT_USER | CURSOR | DATE | DEALLOCATE | DEC | DECIMAL |
DECLARE | DEFAULT | DEFERRABLE | DEFERRED | DELETE | DESC |
DESCRIBE | DESCRIPTOR | DIAGNOSTICS | DISCONNECT | DISTINCT |
DOMAIN | DOUBLE | DROP | ELSE | END | ENDEXEC | ESCAPE | EXCEPT |
EXCEPTION | EXEC | EXECUTE | EXISTS | EXTERNAL | EXTRACT |
FALSE | FETCH | FIRST | FLOAT | FOR | FOREIGN | FOUND | FROM | FULL |
GET | GLOBAL | GO | GOTO | GRANT | GROUP | HAVING | HOUR |
IDENTITY | IMMEDIATE | IN | INDICATOR | INITIALLY |
INNER | INPUT | INSENSITIVE | INSERT | INT | INTEGER | INTERSECT |
INTERVAL | INTO | IS | ISOLATION | JOIN | LANGUAGE | LAST |
LEADING | LEFT | LEVEL | LIKE | LOCAL | LOWER | MATCH | MAX | MIN |
MINUTE | MODULE | NAMES | NATIONAL | NATURAL | NCHAR |
NEXT | NO | NOT | NULL | NULLIF | NUMERIC | OCTET_LENGTH | OF | ON |
ONLY | OPEN | OPTION | OR | OUTER | OUTPUT | OVERLAPS |
PAD | PARTIAL | PREPARE | PRESERVE | PRIMARY | PRIOR | PRIVILEGES |
PROCEDURE | PUBLIC | READ | REAL | REFERENCES | RELATIVE |
RESTRICT | REVOKE | RIGHT | ROLE | ROLLBACK | ROWS |
SCHEMA | SCROLL | SECOND | SECTION | SELECT | SESSION_USER |
SET | SHARD | SMALLINT | SOME | SPACE | SQLERROR | SQLSTATE |
STATISTICS | SUBSTRING | SUM | SYSDATE | SYSTEM_USER | TABLE |
TEMPORARY | THEN | TIME | TIMEZONE_HOUR | TIMEZONE_MINUTE |
TO | TOP | TRAILING | TRANSACTION | TRIM | TRUE | UNION | UNIQUE |
UPDATE | UPPER | USER | USING | VALUES | VARCHAR | VARYING | WHEN |
WHenever | WHERE | WITH | WORK | WRITE
```

## 概要

SQL 内では、特定の文字が予約されています。SQL 予約語は、以下の場合を除いて、SQL 識別子 (テーブル名、列名、AS エイリアス、その他のエンティティ名など) としては使用できません。

- ・ 単語が二重引用符で区切られており ("word")、かつ
- ・ 区切り識別子がサポートされている。詳細は、“[識別子](#)” を参照してください。

以下のリストは、このように予約された単語のみを含みます。すべての SQL キーワードを含むわけではありません。上にリストしたいいくつかの語は、先頭が “%” 文字です。これはそれらが InterSystems SQL 専用の拡張キーワードであることを示しています。一般に、テーブル名や列名などの識別子として先頭が “%” の語を使用することはお勧めしません。新しい InterSystems SQL 拡張キーワードが将来追加される可能性があるからです。

単語が SQL の予約語であるかどうかを調べるには、以下の例に示すように、IsReservedWord() メソッドを呼び出します。予約語は文字列に引用符を付けて指定します。予約語では大文字と小文字が区別されません。

\$SYSTEM.SQL.IsReservedWord() はブーリアン値を返します。

## ObjectScript

```
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("VARCHAR")
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("varchar")
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("VarChar")
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("FRED")
```

このメソッドは、ODBC または JDBC からストアード・プロシージャ %SYSTEM.SQL\_IsReservedWord("nnnn") として呼び出すこともできます。

## 特殊変数

システム定義変数。

### 構文

```
$HOROLOG
$JOB
$NAMESPACE
$TLEVEL
$USERNAME
$ZHOROLOG
$ZJOB
$ZPI
$ZTIMESTAMP
$ZTIMEZONE
$ZVERSION
```

### 説明

InterSystems SQL は、ObjectScript 特殊変数のいくつかを直接サポートします。これらの変数には、システム定義の値が含まれます。これらは、InterSystems SQL でリテラル値を指定できる場所であればどこでも使用できます。

SQL 特殊変数名では、大文字と小文字が区別されません。ほとんどは省略形を使用して指定できます。

変数名	省略形	戻り値のデータ型	用途
<a href="#">\$HOROLOG</a>	\$H	%String/VARCHAR	現在のプロセスのローカル日付と時刻
<a href="#">\$JOB</a>	\$J	%String/VARCHAR	現在のプロセスのジョブ ID
<a href="#">\$NAMESPACE</a>	なし	%String/VARCHAR	現在のネームスペース名
<a href="#">\$TLEVEL</a>	\$TL	%Integer/INTEGER	現在のトランザクションの入れ子レベル
<a href="#">\$USERNAME</a>	なし	%String/VARCHAR	現在のプロセスのユーザ名
<a href="#">\$ZHOROLOG</a>	\$ZH	%Numeric/NUMERIC(21,6)	InterSystems IRIS の起動時以降の経過秒数
<a href="#">\$ZJOB</a>	\$ZJ	%Integer/INTEGER	現在のプロセスのジョブの状態
<a href="#">\$ZPI</a>	なし	%Numeric/NUMERIC(21.18)	数値定数 PI
<a href="#">\$ZTIMESTAMP</a>	\$ZTS	%String/VARCHAR	協定世界時 (UTC) 形式による現在の日付と時刻
<a href="#">\$ZTIMEZONE</a>	\$ZTZ	%Integer/INTEGER	GMT を基準にしたローカル・タイム・ゾーン・オフセット
<a href="#">\$ZVERSION</a>	\$ZV	%String/VARCHAR	InterSystems IRIS の現在のバージョン

詳細は、“[ObjectScript リファレンス](#)” の対応する ObjectScript の特殊変数を参照してください。

### 例

以下の例は、現在の日付と時刻を含む結果セットを返します。



## SQL

```
SELECT TOP 5 Name,$H  
FROM Sample.Person
```

以下の例は、タイム・ゾーンが米国本土内にある場合にのみ結果セットを返します。

## SQL

```
SELECT TOP 5 Name,Home_State  
FROM Sample.Person  
WHERE $ZTIMEZONE BETWEEN 300 AND 480
```

# 文字列操作 (SQL)

文字列操作の関数および演算子

## 概要

InterSystems SQL では、以下の複数のタイプの文字列操作をサポートします。

- ・ 文字列は、長さ、文字の位置、または部分文字列値により操作できます。
- ・ 文字列は、指定の区切り文字または区切り文字列により操作できます。
- ・ 文字列は、パターン・マッチングと単語認識検索によりテストできます。
- ・ リストと呼ばれる特別にエンコードされた文字列には、区切り文字を使用しない埋め込み部分文字列識別子が含まれます。さまざまな \$LIST 関数は、これらのエンコードされた文字列を操作します。この文字列は、標準の文字列とは互換性がありません。唯一の例外は、\$LISTGET 関数と、引数が 1 つおよび 2 つの形式の \$LIST 関数です。これらの関数は、入力としてエンコードされた文字の文字列を受け取り、単一要素値を標準文字の文字列として出力します。

InterSystems SQL は、文字列関数、文字列条件式、および文字列演算子をサポートします。

ObjectScript の文字列操作では、大文字と小文字が区別されます。文字列の文字は、大文字や小文字に変換するか、または両方を混在させて保持することができます。[文字列照合](#)では大文字と小文字が区別されるようにすることもでき、区別されないようにすることもできます。既定では、SQL 文字列照合は、大文字と小文字が区別されない SQLUPPER です。InterSystems SQL は、さまざまな大文字/小文字、[照合関数](#)、および演算子を提供しています。

文字列が数値引数に対して指定される場合、ほとんどの InterSystems SQL 関数は、以下の文字列から数値への変換を実行します。数値以外の文字列は 0 に変換されます。数値文字列はキャノニック形式の数値に変換されます。混合数値文字列は最初の実数値文字で切り捨てられ、キャノニック形式の数値に変換されます。

## 文字列連結

以下の関数は部分文字列を単一の文字列に連結します。

- ・ [CONCAT](#) : 2 つの部分文字列を連結して、単一の文字列を返します。
- ・ [STRING](#) : 2 つ以上の部分文字列を連結して、単一の文字列を返します。
- ・ [XMLAGG](#) : 列内のすべての値を連結して、単一の文字列を返します。詳細は、“[集約関数](#)”を参照してください。
- ・ [LIST](#) : コンマ区切りを含む列内のすべての値を連結して、単一の文字列を返します。詳細は、“[集約関数](#)”を参照してください。

連結演算子 (||) も、2 つの文字列を連結するために使用できます。

## 文字列の長さ

以下の関数は文字列の長さを決定するために使用できます。

- ・ [CHARACTER\\_LENGTH](#) および [CHAR\\_LENGTH](#) : 末尾の空白を含む、文字列の文字数を返します。NULL は NULL を返します。
- ・ [LENGTH](#) : 末尾の空白を除く、文字列の文字数を返します。NULL は NULL を返します。
- ・ [\\$LENGTH](#) : 末尾の空白を含む、文字列の文字数を返します。NULL は 0 として返されます。

## トランケーションとトリミング

以下の関数は文字列を切り捨てるか削除するために使用できます。トランケーションは、文字列の長さを制限し、指定された長さを超えるすべての文字を削除します。トリミングは、先頭および/または末尾の空白を文字列から削除します。

- ・ トランケーション : [CONVERT](#)、[%SQLSTRING](#)、および [%SQLUPPER](#)。
- ・ トリミング : [TRIM](#)、[LTRIM](#)、および [RTRIM](#)。

## 部分文字列検索

以下の関数では、文字列内の部分文字列を検索してその位置を返します。

- ・ [POSITION](#) : 部分文字列値による検索を実行し、最初の一致を見つけて、部分文字列の最初の位置を返します。
- ・ [CHARINDEX](#) : 部分文字列値による検索を実行し、最初の一致を見つけて、部分文字列の最初の位置を返します。開始位置を指定できます。
- ・ [\\$FIND](#) : 部分文字列値による検索を実行し、最初の一致を見つけて、部分文字列の最後の位置を返します。開始位置を指定できます。
- ・ [INSTR](#) : 部分文字列値による検索を実行し、最初の一致を見つけて、部分文字列の最初の位置を返します。開始位置、およびそこから数えて何個目の部分文字列かを指定できます。

以下の関数では、文字列内の位置または区切り文字を指定して部分文字列を検索し、その部分文字列を返します。

- ・ [\\$EXTRACT](#) : 文字列位置による検索を実行し、開始位置によって指定されたか、開始位置と終了位置によって指定された部分文字列を返します。文字列の最初から検索します。
- ・ [SUBSTRING](#) : 文字列位置による検索を実行し、開始位置によって指定されたか、開始位置と長さによって指定された部分文字列を返します。文字列の最初から検索します。
- ・ [SUBSTR](#) : 文字列位置による検索を実行し、開始位置によって指定されたか、開始位置と長さによって指定された部分文字列を返します。文字列の最初または最後から検索します。
- ・ [\\$PIECE](#) : 区切り文字による検索を実行し、最初の区切られた部分文字列を返します。開始地点を指定するか、既定を文字列の最初に設定できます。
- ・ [\\$LENGTH](#) : 区切り文字による検索を実行し、区切られた部分文字列の数を返します。文字列の最初から検索します。
- ・ [\\$LIST](#) : 特別にエンコードされたリスト文字列の部分文字列のカウントによる検索を実行します。文字列のカウントにより部分文字列を検索し、部分文字列の値を返します。文字列の最初から検索します。

包含関係演算子 (D) は、部分文字列が単一の文字列に表示されるかどうかを調べるために使用することもできます。

[%STARTSWITH](#) 比較演算子は、指定した文字を、文字列の先頭とマッチングします。

## 部分文字列の検索と置換

以下の関数は、文字列内の部分文字列を検索し、それを別の部分文字列と置き換えます。

- ・ [REPLACE](#) : 文字列の値による検索を実行し、部分文字列を新しい部分文字列と置き換えます。文字列の最初から検索します。
- ・ [STUFF](#) : 文字列の位置と長さによる検索を実行し、部分文字列を新しい部分文字列と置き換えます。文字列の最初から検索します。

## 文字タイプと単語認識比較

[%PATTERN](#) 比較演算子は、文字列を指定の文字タイプのパターンとマッチングします。

ワイルドカード検索を含む、指定された単語や語句の文字列の単語認識検索を実行できます。詳細は、“[InterSystems SQL Search の使用法](#)”を参照してください。

