



# ObjectScript リファレンス

Version 2024.1  
2024-06-03

ObjectScript 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

記号および省略形 .....	1
ObjectScript で使用する記号 .....	2
ObjectScript で使用する省略形 .....	12
ObjectScript 演算子 .....	17
単項プラス (+) .....	18
単項マイナス (-) .....	19
加算 (+) .....	20
減算 (-) .....	21
乗算 (*) .....	22
除算 (/) .....	23
整数除算 (¥) .....	24
モジュロ (#) .....	25
べき乗 (**) .....	26
より小さい (<) .....	28
より大きい (>) .....	29
以下 (<= または '>') .....	30
以上 (>= または '<') .....	31
否定 (!) .....	32
論理積 (& または '&&') .....	33
論理和 (! または '  ') .....	35
論理積否定 (NAND) ('&') .....	36
論理和否定演算 (NOR) ('!') .....	37
文字列の連結 ( ) .....	38
等価 (=) .....	40
不等価 ('=') .....	41
包含 ( ) .....	42
非包含 (' ') .....	43
後続 ( ) .....	44
非後続 (' ') .....	45
前後関係 ( ) .....	46
非前後関係 (' ') .....	47
パターン・マッチング (?) .....	48
間接 (@) .....	56
ObjectScript コマンド .....	61
BREAK (ObjectScript) .....	62
CATCH (ObjectScript) .....	67
CLOSE (ObjectScript) .....	73
CONTINUE (ObjectScript) .....	75
DO (ObjectScript) .....	77
DO WHILE (ObjectScript) .....	86
ELSE (ObjectScript) .....	90
ELSEIF (ObjectScript) .....	91
FOR (ObjectScript) .....	92
GOTO (ObjectScript) .....	100
HALT (ObjectScript) .....	104
HANG (ObjectScript) .....	107

IF (ObjectScript) .....	110
JOB (ObjectScript) .....	114
KILL (ObjectScript) .....	128
LOCK (ObjectScript) .....	134
MERGE (ObjectScript) .....	147
NEW (ObjectScript) .....	151
OPEN (ObjectScript) .....	157
QUIT (ObjectScript) .....	163
READ (ObjectScript) .....	169
RETURN (ObjectScript) .....	177
SET (ObjectScript) .....	182
TCOMMIT (ObjectScript) .....	195
THROW (ObjectScript) .....	197
TROLLBACK (ObjectScript) .....	202
TRY (ObjectScript) .....	207
TSTART (ObjectScript) .....	211
USE (ObjectScript) .....	214
VIEW (ObjectScript) .....	219
WHILE (ObjectScript) .....	223
WRITE (ObjectScript) .....	228
XECUTE (ObjectScript) .....	240
ZKILL (ObjectScript) .....	246
ZNSPACE (ObjectScript) .....	248
ZTRAP (ObjectScript) .....	253
ZWRITE (ObjectScript) .....	256
ZZDUMP (ObjectScript) .....	263
ZZWRITE (ObjectScript) .....	266
ルーチンおよびデバッグ・コマンド .....	269
PRINT (ObjectScript) .....	270
ZBREAK (ObjectScript) .....	273
ZINSERT (ObjectScript) .....	281
ZLOAD (ObjectScript) .....	286
ZPRINT (ObjectScript) .....	291
ZREMOVE (ObjectScript) .....	294
ZSAVE (ObjectScript) .....	298
ZZPRINT (ObjectScript) .....	302
ObjectScript 関数 .....	305
\$ASCII (ObjectScript) .....	306
\$BIT (ObjectScript) .....	310
\$BITCOUNT (ObjectScript) .....	314
\$BITFIND (ObjectScript) .....	316
\$BITLOGIC (ObjectScript) .....	318
\$CASE (ObjectScript) .....	322
\$CHANGE (ObjectScript) .....	325
\$CHAR (ObjectScript) .....	327
\$CLASSMETHOD (ObjectScript) .....	331
\$CLASSNAME (ObjectScript) .....	333
\$COMPILE (ObjectScript) .....	335
\$DATA (ObjectScript) .....	340
\$DECIMAL (ObjectScript) .....	344

\$DOUBLE (ObjectScript) .....	347
\$EXTRACT (ObjectScript) .....	352
\$FACTOR (ObjectScript) .....	359
\$FIND (ObjectScript) .....	361
\$FNUMBER (ObjectScript) .....	364
\$GET (ObjectScript) .....	370
\$INCREMENT (ObjectScript) .....	373
\$INNUMBER (ObjectScript) .....	378
\$ISOBJECT (ObjectScript) .....	384
\$ISVALIDDOUBLE (ObjectScript) .....	386
\$ISVALIDNUM (ObjectScript) .....	390
\$isvector (ObjectScript) .....	395
\$JUSTIFY (ObjectScript) .....	396
\$LENGTH (ObjectScript) .....	399
\$LIST (ObjectScript) .....	402
\$LISTBUILD (ObjectScript) .....	413
\$LISTDATA (ObjectScript) .....	419
\$LISTFIND (ObjectScript) .....	422
\$LISTFROMSTRING (ObjectScript) .....	426
\$LISTGET (ObjectScript) .....	429
\$LISTLENGTH (ObjectScript) .....	434
\$LISTNEXT (ObjectScript) .....	437
\$LISTSAME (ObjectScript) .....	440
\$LISTTOSTRING (ObjectScript) .....	444
\$LISTUPDATE (ObjectScript) .....	448
\$LISTVALID (ObjectScript) .....	451
\$LOCATE (ObjectScript) .....	453
\$MATCH (ObjectScript) .....	457
\$METHOD (ObjectScript) .....	460
\$NAME (ObjectScript) .....	462
\$NCONVERT (ObjectScript) .....	466
\$NORMALIZE (ObjectScript) .....	468
\$NOW (ObjectScript) .....	472
\$NUMBER (ObjectScript) .....	475
\$ORDER (ObjectScript) .....	480
\$PARAMETER (ObjectScript) .....	484
\$PIECE (ObjectScript) .....	486
\$PREFETCHOFF (ObjectScript) .....	495
\$PREFETCHON (ObjectScript) .....	497
\$PROPERTY (ObjectScript) .....	499
\$QLENGTH (ObjectScript) .....	501
\$QSUBSCRIPT (ObjectScript) .....	503
\$QUERY (ObjectScript) .....	505
\$RANDOM (ObjectScript) .....	509
\$REPLACE (ObjectScript) .....	511
\$REVERSE (ObjectScript) .....	514
\$SCONVERT (ObjectScript) .....	516
\$SELECT (ObjectScript) .....	518
\$SEQUENCE (ObjectScript) .....	520
\$SORTBEGIN (ObjectScript) .....	524
\$SORTEND (ObjectScript) .....	526

\$STACK (ObjectScript) .....	528
\$TEXT (ObjectScript) .....	532
\$TRANSLATE (ObjectScript) .....	536
\$vector (ObjectScript) .....	539
\$vectordefined (ObjectScript) .....	547
\$vectorop (ObjectScript) .....	550
\$VIEW (ObjectScript) .....	566
\$WASCII (ObjectScript) .....	572
\$WCHAR (ObjectScript) .....	574
\$WEXTRACT (ObjectScript) .....	575
\$WFIND (ObjectScript) .....	578
\$WISWIDE (ObjectScript) .....	580
\$WLENGTH (ObjectScript) .....	581
\$WREVERSE (ObjectScript) .....	583
\$XECUTE (ObjectScript) .....	585
\$ZABS (ObjectScript) .....	587
\$ZARCCOS (ObjectScript) .....	588
\$ZARCSIN (ObjectScript) .....	590
\$ZARCTAN (ObjectScript) .....	592
\$ZBOOLEAN (ObjectScript) .....	594
\$ZCONVERT (ObjectScript) .....	600
\$ZCOS (ObjectScript) .....	605
\$ZCOT (ObjectScript) .....	607
\$ZCRC (ObjectScript) .....	608
\$ZCSC (ObjectScript) .....	611
\$ZCYC (ObjectScript) .....	612
\$ZDASCII (ObjectScript) .....	613
\$ZDATE (ObjectScript) .....	615
\$ZDATEH (ObjectScript) .....	628
\$ZDATETIME (ObjectScript) .....	641
\$ZDATETIMEH (ObjectScript) .....	657
\$ZDCHAR (ObjectScript) .....	670
\$ZEXP (ObjectScript) .....	671
\$ZF (ObjectScript) .....	673
\$ZF(-1) (ObjectScript) .....	677
\$ZF(-2) (ObjectScript) .....	680
\$ZF(-3) (ObjectScript) .....	682
\$ZF(-4) (ObjectScript) .....	684
\$ZF(-5) (ObjectScript) .....	687
\$ZF(-6) (ObjectScript) .....	688
\$ZF(-100) (ObjectScript) .....	689
\$ZHEX (ObjectScript) .....	694
\$ZISWIDE (ObjectScript) .....	697
\$ZLASCII (ObjectScript) .....	698
\$ZLCHAR (ObjectScript) .....	699
\$ZLN (ObjectScript) .....	701
\$ZLOG (ObjectScript) .....	703
\$ZNAME (ObjectScript) .....	705
\$ZPOSITION (ObjectScript) .....	709
\$ZPOWER (ObjectScript) .....	711
\$ZQASCII (ObjectScript) .....	713

\$ZQCHAR (ObjectScript) .....	715
\$ZSEARCH (ObjectScript) .....	717
\$ZSEC (ObjectScript) .....	721
\$ZSEEK (ObjectScript) .....	722
\$ZSIN (ObjectScript) .....	724
\$ZSQR (ObjectScript) .....	726
\$ZSTRIP (ObjectScript) .....	727
\$ZTAN (ObjectScript) .....	731
\$ZTIME (ObjectScript) .....	733
\$ZTIMEH (ObjectScript) .....	738
\$ZVERSION(1) (ObjectScript) .....	741
\$ZWASCII (ObjectScript) .....	742
\$ZWCHAR (ObjectScript) .....	744
\$ZWIDTH (ObjectScript) .....	746
\$ZWPACK および \$ZBPACK (ObjectScript) .....	747
\$ZWUNPACK および \$ZBUNPACK (ObjectScript) .....	749
\$ZENKAKU (ObjectScript) .....	751
ObjectScript 特殊変数 .....	753
\$DEVICE (ObjectScript) .....	754
\$ECODE (ObjectScript) .....	755
\$ESTACK (ObjectScript) .....	758
\$ETRAP (ObjectScript) .....	761
\$HALT (ObjectScript) .....	765
\$HOROLOG (ObjectScript) .....	768
\$IO (ObjectScript) .....	773
\$JOB (ObjectScript) .....	775
\$KEY (ObjectScript) .....	776
\$NAMESPACE (ObjectScript) .....	779
\$PRINCIPAL (ObjectScript) .....	782
\$QUIT (ObjectScript) .....	783
\$ROLES (ObjectScript) .....	784
\$STACK (ObjectScript) .....	786
\$STORAGE (ObjectScript) .....	788
\$SYSTEM (ObjectScript) .....	791
\$TEST (ObjectScript) .....	794
\$THIS (ObjectScript) .....	796
\$THROWOBJ (ObjectScript) .....	797
\$TLEVEL (ObjectScript) .....	798
\$USERNAME (ObjectScript) .....	801
\$X (ObjectScript) .....	803
\$Y (ObjectScript) .....	805
\$ZA (ObjectScript) .....	807
\$ZB (ObjectScript) .....	809
\$ZCHILD (ObjectScript) .....	811
\$ZEOF (ObjectScript) .....	812
\$ZEOS (ObjectScript) .....	813
\$ZERROR (ObjectScript) .....	814
\$ZHOROLOG (ObjectScript) .....	820
\$ZIO (ObjectScript) .....	821
\$ZJOB (ObjectScript) .....	822

\$ZMODE (ObjectScript) .....	824
\$ZNAME (ObjectScript) .....	826
\$ZNSPACE (ObjectScript) .....	827
\$ZORDER (ObjectScript) .....	829
\$ZPARENT (ObjectScript) .....	830
\$ZPI (ObjectScript) .....	831
\$ZPOSITION (ObjectScript) .....	832
\$ZREFERENCE (ObjectScript) .....	833
\$ZSTORAGE (ObjectScript) .....	837
\$ZTIMESTAMP (ObjectScript) .....	839
\$ZTIMEZONE (ObjectScript) .....	842
\$ZTRAP (ObjectScript) .....	846
\$ZVERSION (ObjectScript) .....	851
構造化システム変数 .....	853
`\$GLOBAL (ObjectScript) .....	854
`\$JOB (ObjectScript) .....	858
`\$LOCK (ObjectScript) .....	860
`\$ROUTINE (ObjectScript) .....	865
マクロ・プリプロセッサ指示文 .....	869
#; .....	870
#deflarg .....	871
#define .....	872
#dim .....	875
#else .....	877
#elseif .....	878
#endif .....	879
#execute .....	880
#if .....	881
#ifDef .....	882
#ifNDef .....	883
#import .....	884
#include .....	886
#noshow .....	888
#show .....	889
#sqlcompile audit .....	890
#sqlcompile mode .....	891
#sqlcompile path .....	892
#sqlcompile select .....	894
#undef .....	896
##; .....	897
##beginquote ...##EndQuote .....	898
##continue .....	899
##expression .....	900
##function .....	902
##lit .....	904
##quote .....	905
##quoteExp .....	906
##sql .....	907
##stripq .....	908
##unique .....	909



付録A: 識別子のルールとガイドライン .....	911
A.1 ローカル変数名のルール .....	911
A.2 回避する必要があるローカル変数名 .....	911
A.3 グローバル変数名のルール .....	912
A.4 回避する必要があるグローバル変数名 .....	912
A.4.1 パーセント・グローバル .....	912
A.4.2 非パーセント・グローバル .....	912
A.5 ルーチン名とラベルのルール .....	914
A.6 ユーザが使用するために予約されているルーチン名 .....	914
A.7 クラス名のルール .....	915
A.8 回避する必要があるパッケージ名、クラス名、およびスキーマ名 .....	915
A.9 クラス・メンバ名のルール .....	916
A.10 回避する必要があるメンバ名 .....	916
A.11 IRISYS のカスタム項目 .....	917
付録B: 一般的なシステム制限 .....	919
B.1 文字列長の制限 .....	919
B.2 添え字の制限 .....	919
B.3 グローバル参照の最大長 .....	920
B.4 クラスの制限 .....	920
B.5 クラスおよびルーチンの制限 .....	921
B.6 その他のプログラミング制限 .....	923
付録C: システム・マクロ .....	925
C.1 これらのマクロを使用できるようにする方法 .....	925
C.2 マクロ・リファレンス .....	925
付録D: システム・フラグおよびシステム修飾子 (qspec) .....	929
D.1 例 .....	929
D.2 否定 .....	929
D.3 フラグ .....	930
D.4 コンパイラ修飾子 .....	930
D.5 エクスポート修飾子 .....	933
D.6 ShowClassAndObject 修飾子 .....	935
D.7 UnitTest 修飾子 .....	935
D.8 フラグに対応する修飾子 .....	936
D.9 フラグおよび修飾子のヘルプ .....	937
D.10 既定値の制御 .....	938
D.11 qspec の処理順序 .....	938
付録E: 正規表現 .....	939
E.1 ワイルドカードと修飾子 .....	939
E.2 リテラルと文字の範囲 .....	940
E.3 文字タイプ・メタ文字 .....	941
E.3.1 単一文字タイプ .....	941
E.3.2 Unicode プロパティ文字タイプ .....	942
E.3.3 POSIX 文字タイプ .....	945
E.4 グループ化構文 .....	946
E.5 アンカー・メタ文字 .....	947
E.5.1 文字列の先頭または最後 .....	947
E.5.2 単語境界 .....	948
E.6 論理演算子 .....	949
E.7 文字表現メタ文字 .....	949

E.7.1 16 進、8 進、および Unicode の表現 .....	949
E.7.2 制御文字表現 .....	950
E.7.3 記号名表現 .....	950
E.8 モード .....	950
E.8.1 正規表現シーケンスのモード .....	951
E.8.2 リテラルのモード .....	952
E.9 Comments .....	953
E.9.1 埋め込みコメント .....	953
E.9.2 行末コメント .....	953
E.10 エラー・メッセージ .....	953
E.11 関連項目 .....	954
付録F: 変換テーブル .....	955
F.1 概要 .....	955
F.2 テーブルのリスト .....	955
F.3 出力エスケープ .....	958
F.4 URL および URI の変換 .....	959
F.5 JS と JSML、JSON と JSONML の変換 .....	960
F.6 関連する API .....	960
F.7 関連概念 .....	961
F.8 関連項目 .....	961
付録G: インターシステムズ・アプリケーションでの数値の計算 .....	963
G.1 概要 .....	963
G.1.1 SQL 表現 .....	963
G.2 10 進形式 .....	963
G.3 \$DOUBLE 形式 .....	964
G.4 数値形式の選択 .....	965
G.5 変換 : 文字列 .....	966
G.5.1 数値としての文字列 .....	966
G.5.2 添え字としての数値文字列 .....	966
G.6 変換 : \$DOUBLE へ .....	967
G.7 変換 : 10 進数へ .....	967
G.7.1 \$DECIMAL(x) .....	967
G.7.2 \$DECIMAL(x, n) .....	967
G.8 変換 : 10 進数から文字列へ .....	968
G.9 算術演算 .....	968
G.9.1 同種の表現 .....	968
G.9.2 異種の表現 .....	968
G.9.3 丸め .....	968
G.10 比較演算 .....	969
G.10.1 同種の表現 .....	969
G.10.2 異種の表現 .....	969
G.10.3 以下、以上 .....	969
G.11 ブーリアン演算 .....	970
G.12 関連項目 .....	970

## 図一覧

図 C-1: 非並行モードおよび並行モードのクライアント/サーバ接続 .....	124
図 C-2: $\hat{X}$ と $\hat{Y}$ の初期構造 .....	149
図 C-3: $\hat{X}$ および $\hat{Y}$ の MERGE コマンドの結果 .....	149

# テーブル一覧

テーブル B-1: パターン・コード .....	50
--------------------------	----

# 記号および省略形

## ObjectScript で使用する記号

ObjectScript で演算子、接頭語などとして使用される文字のテーブルです。

### 記号のテーブル

以下は、InterSystems IRIS® Data Platform の ObjectScript で使用されるリテラル記号です。(このテーブルには、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、[InterSystems SQL で使用する記号](#)を示したテーブルもあります。

各記号の名前の後には、ASCII 数値が続きます。

記号	名前と使用法
[スペース]、または [タブ]	<p>空白 (タブ (9) またはスペース (32)) : <a href="#">ラベル</a>および一部の<a href="#">コメント</a>行以外の各コード行の行頭に空白 (タブあるいはスペース) が必要です。</p> <p><a href="#">コマンド</a>内には、コマンド名と最初の引数の間にスペースが 1 つが必要です。</p> <p><a href="#">最後のコマンドの引数</a>と同じ行にある後続のコマンドまたはコメントとの間には、後続の空白 (スペースあるいはタブ) が必要です。<a href="#">ラベル</a>と同じ行にある後続のコマンドまたはコメントとの間にも、後続の空白が必要です。</p>
[2 つのスペース、2 つのタブ、またはスペースとタブ 1 つずつ]	<p>2 つの空白 : 引数のないコマンドと、同じ行の<a href="#">次のコマンド</a>の間には、後続の2 つの空白が必要です。</p>
!	<p>感嘆符 (33) : <a href="#">OR 論理演算子</a> (完全な評価)。</p> <p><a href="#">READ</a> コマンドと <a href="#">WRITE</a> コマンドでは、新規の行を指定します。</p> <p>最初の文字として、ターミナル・プロンプトで<a href="#">インタラクティブなサブシェル</a>をロードします。</p>
"	<p>引用符 (34) : <a href="#">文字列リテラル</a> を囲みます。<a href="#">ダイナミック SQL</a> では、%Prepare() メソッドの文字列引数として、SQL コードを囲みます。</p> <p>ストレート引用符 (") と方向付き引用符 (" ") の違いについては、"<a href="#">パターン・マッチング (?)</a>" を参照してください。</p>
""	<p>二重引用符 : 0 の長さの文字列である NULL 文字列 ("" ) を指定します。</p> <p>引用符付きの文字列内では、<a href="#">リテラル引用符文字</a>を指定します。</p>

記号	名前と使用法
#	<p>シャープ記号 (35) : <b>モジュロ除算演算子</b>。ビット値を決定するために使用できます。例えば、\$ZA#2 は、ビット 1 の値 (0 または 1) を返します。整数除算 (¥) 演算子 \$ZJOB\1024#2 は、ビット 1024 の値 (0 または 1) を返します。</p> <p><b>READ</b> コマンドおよび <b>WRITE</b> コマンドでは改ページ。固定長の READ では読み取る文字数。</p> <p>クラス内からクラス・パラメータの値を参照するための接頭語。(例) #ParameterName。</p> <p>#define、#include、#if などの多数の<b>マクロ・プリプロセッサ指示文</b>の名前で使用する接頭語。## も参照してください。</p> <p>class 構文では、<b>パラメータ値</b>を返すためにパラメータ接頭語が使用されます。例えば、##class(%Library.Boolean).#XSDTYPE または myinstance.#EXTENTQUERYSPEC のようになります。</p> <p><b>ZBREAK</b> デバッグでは、指定のブレークポイントまたはウォッチポイントを無効化する繰り返しカウンタです。例えば、ZBREAK -label^rou#100 は、label^rou のブレークポイントを 100 回繰り返して無効化します。</p> <p>コールアウト・ルーチン ZFENTRY で、DOUBLE データ型を示す argtype 接頭語。(例) #D または #F。</p> <p><b>正規表現</b>の行末コメント文字 ((?x) モード内のみ)。</p>
##	<p>二重ポンド記号 : オブジェクト・<b>クラス呼び出し接頭語</b> : ##class(classname).methodname() または ##class(classname).#parametername。</p> <p><b>##super() syntax</b> を使用して、オーバーライドされたスーパークラス・メソッドを呼び出します。</p> <p>##continue、##expression、##function、##lit、##sql、##unique などの特定の<b>マクロ・プリプロセッサ指示文</b>の名前の接頭語。##sql は、ObjectScript : ##sql (SQL コマンド) 内から SQL コードの行を実行するために呼び出されます。</p>
##;	二重シャープ記号セミコロン : <b>1 行コメント</b> 文字。ObjectScript または 埋め込み SQL の列 1 で使用できます。
#;	シャープ記号セミコロン : <b>1 行コメント</b> 文字。列 1 で使用できます。
\$	<p>ドル記号 (36) : <b>システム関数接頭語</b>。(例) \$name(parameters)。</p> <p><b>特殊変数</b>接頭語。(例) \$name。</p> <p>\$Znnn (\$Z で始まる名前) は、<b>%ZLANG</b> 言語拡張ライブラリを使用して定義された、ユーザ定義関数または特殊変数にできます。インターシステムズが提供するシステム関数または特殊変数にすることも可能です。</p> <p><b>正規表現</b>の文字列の最後のアンカー。(例) (USA)\$。</p> <p><b>ZBREAK</b> デバッグでは、シングル・ステップ・ブレークポイントです。</p> <p>最初の文字として、ターミナル・プロンプトで<b>インタラクティブなサブシェルをロード</b>します。</p>
\$\$	<p>二重ドル記号 : <b>ユーザ指定関数呼び出し接頭語</b>。(例) \$\$myname(parameters)。コンテキストがユーザ指定関数への参照によって構築されたとき、\$\$ は <b>\$STACK</b> によって返されます。</p> <p>ルーチンを直接呼び出すためのルーチン名の接頭語。</p>
\$\$\$	三重ドル記号 : <b>マクロ呼び出し接頭語</b> 。

記号	名前と使用法
%	<p>パーセント記号 (37) : 以下で、名前の最初の文字として使用できます。(1) 特殊な有効範囲のルールを持つ “% <i>variable</i>” を示し、ロックに使用するローカル変数名。(2) 一般にシステム・ユーティリティを示すルーチン名。(3) パッケージ・クラス名。例えば、%SYSTEM.class や %Library.class および %Library パッケージ内のクラス名 (%String などのデータ型を含む)。(4) %Dialect、%New()、%OpenId() などの %Persistent オブジェクトのプロパティ名およびメソッド名。%On... メソッド名は <b>コールバック・メソッド</b> です。(5) <b>ラベル</b>。</p> <p><b>マクロ引数</b> の最初に記述する必要があります。</p> <p>一部の埋め込み SQL 変数の接頭語。(例) %msg、%ROWCOUNT。SQL キーワードの接頭語。(例) %STARTSWITH。</p> <p>% ( <b>インスタンス変数</b> ) 参照。</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) %%CLASS-NAME、%%CLASSNAMEQ、%%ID、%%TABLENAME。ObjectScript の計算フィールド・コードおよび <b>トリガ・コード</b> で使用されます。</p>
&	<p>アンパサンド (38) : <b>AND 論理演算子</b> (完全な評価)。<b>\$BITLOGIC</b> ビット文字列 AND 演算子。</p> <p>仮パラメータ・リストでは、パラメータが参照によって渡されることをマークする、オプションの機能しない <b>変数名の接頭語</b>。&amp; は、マーカであり、変数名の一部ではありません。(例) Calc(x,&amp;y)。</p> <p>埋め込みコードのシェル呼び出し接頭語。例えば、&amp;sql (SQL commands)。</p> <p><b>UNIX® バッチ・コマンド</b>。</p>
&&	<p>二重アンパサンド : <b>AND 論理演算子</b> (部分的な評価)。</p> <p><b>正規表現</b> の AND 論理演算子。</p>
記号	名前と使用法
'	<p>アポストロフィ (39) : <b>単項否定演算子</b>。リレーショナル演算子 '= (not equal to)、'&lt; (not less than)、'&gt; (not greater than)、<b>パターン・マッチ</b> 'operand?pattern) と組み合わせることが出来ます。</p> <p><b>単項否定演算子</b>。論理演算子 '&amp; (Not And) および '  (Not Or) と組み合わせることが出来ます。論理演算子 &amp;&amp; または    と組み合わせることはできません。</p> <p>ヨーロッパ式 <b>数値グループ・セパレータ</b>。</p>



記号	名前と使用法
( )	<p>括弧 (40,41) : <b>プロシージャや関数のパラメータ・リスト</b>を囲みます。空白の場合でも括弧は必須です。</p> <p>式を入れ子にします。入れ子によって InterSystems IRIS の既定である演算子の厳密な左から右への評価をオーバーライドし、式に優先順位を指定することができます。</p> <p><b>ローカル変数</b> : <code>a(1,1)</code>、<b>グローバル変数</b> : <code>^a(1,1)</code>、または<b>プロセス・プライベート・グローバル</b> : <code>^ a(1,1)</code> の配列添え字を指定します。</p> <p><b>代替パターン・マッチ</b> (? に続く) を囲みます。</p> <p><b>NEW</b> と <b>KILL</b> コマンドにより排他的に表示します。</p> <p><b>後置条件にスペースを含む場合に必要です。</b></p> <p><code>&amp;sql</code> シェル呼び出しコマンドに続く埋め込み SQL コードを囲みます。(例) <code>&amp;sql(SQL コマンド)</code></p> <p><b>正規表現</b>の一致文字列 (Boston) または文字列のリスト (Boston New York Paris)。<b>正規表現</b>のグループ化構文。</p> <p><b>JSON オブジェクトまたは配列の値</b>を設定する場合、ObjectScript リテラルまたは式を囲むために使用します。</p>
*	<p>アスタリスク (42) : <b>乗算演算子</b>。</p> <p><b>\$ZSEARCH</b> では、0、1、または複数の文字のワイルドカード。</p> <p><b>\$EXTRACT</b>、<b>\$LIST</b>、および <b>\$PIECE</b> では、文字列の末尾の最終項目を指定します。これを符号付整数と共に使用して、末尾からのオフセットを指定できます (例 : <code>*-2</code>、<code>*+1</code>)。</p> <p><b>WRITE</b> コマンドでは、文字の整数コードを指定します。例えば、<code>WRITE *65</code> は文字 “A” を書き込みます。</p> <p><b>\$ZTRAP</b> 文字列値の接頭語として、コール・スタック・レベルが変更されないように指定します。</p> <p><b>ZBREAK</b> では、ローカル変数を示す名前の接頭語です。<b>\$ZERROR</b>に返される特定のエラー・コードには、未定義のローカル変数、クラス、メソッド、またはプロパティを示す名前の接頭語があります。</p> <p><b>正規表現</b>の 0 文字以上の修飾子。</p>
**	<p>二重アスタリスク : <b>指数演算子</b>。例えば、<code>4**3=64</code> です。</p>
*+	<p>アスタリスク・プラス : <b>SET \$EXTRACT</b>、<b>SET \$LIST</b>、および <b>SET \$PIECE</b> では、文字列の最終項目より先のオフセットを指定します。値を追加するために使用されます。例えば、<code>*+1</code> を指定すると、項目が文字列の末尾に追加されます。</p>
*-	<p>アスタリスク・マイナス : <b>WRITE</b> コマンドでは、デバイス制御整数コードを指定します。例えば、<code>WRITE *-10</code> は終端の入力バッファを削除します。</p> <p><b>\$EXTRACT</b>、<b>\$LIST</b>、<b>\$LISTGET</b>、および <b>\$PIECE</b> では、文字列の最終項目からの逆方向のオフセットを指定します。例えば、<code>*-1</code> は最後から 2 番目の項目です。</p>
*/	<p>アスタリスク・スラッシュ : <b>複数行コメント</b>の末端文字。コメントは、<code>/*</code> で開始します。</p>

記号	名前と使用法
+	<p>プラス記号 (43) : <a href="#">単項算術プラス演算子</a>。数値評価を強制する文字列を返す文字列または関数に追加する場合。(例) <code>WRITE + "007.0"</code>、<code>WRITE + \$PIECE(str, ", ", 2)</code>。</p> <p><a href="#">加算演算子</a>。</p> <p>ラベルからの整数の<a href="#">行のカウント・オフセット</a> : <code>label+offset</code>。<a href="#">\$ZTRAP</a> では、プロシージャの先頭からの整数の行のカウント・オフセット : <code>+offset^procname</code>。</p> <p><a href="#">LOCK</a> コマンドおよび <a href="#">ZBREAK</a> コマンドでは、後に続く項目を有効化、適用、またはインクリメントする接頭語です。</p> <p><a href="#">正規表現</a> の 1 文字以上の修飾子。</p>
+=	<p>ファイルへの出力を行うコマンドと関数で、プラス記号と等号が両方指定されている場合は、出力データが既存のファイルの内容に追加されることを意味します。等号のみの場合は、既存のファイルの内容が出力データで上書きされることを意味します。“<a href="#">\$ZF(-100)</a>”を参照してください。</p>
,	<p>コンマ (44) : 関数およびプロシージャでは<a href="#">複数のパラメータの分離記号</a>。</p> <p>コマンドでは<a href="#">複数の引数の区切り文字</a>。</p> <p>配列変数では<a href="#">添え字レベルの分離記号</a>。</p> <p>ロケールに応じて、数値グループ・セパレータまたは<a href="#">小数点文字</a>。SetFormatItem() NLS クラス・メソッドを使用して構成可能です。</p> <p><a href="#">\$ECODE</a> ではエラー・コードを囲みます。(例) <code>,M7,</code></p>
,,	<p>二重コンマ : 関数では、未指定の<a href="#">位置パラメータ</a> (既定値を取る) のプレースホルダ。</p>
-	<p>マイナス記号 (45) : <a href="#">単項算術マイナス演算子</a>。</p> <p><a href="#">減算演算子</a>。</p> <p><a href="#">LOCK</a> コマンドおよび <a href="#">ZBREAK</a> コマンドでは、後に続く項目を無効化、デクリメント、または削除する接頭語です。</p> <p><a href="#">正規表現</a> の文字範囲演算子。(例) <code>[A-Z]</code>。</p>
--	<p>二重マイナス記号 : <a href="#">ZBREAK</a> コマンドでは、後に続く項目を削除する接頭語です。</p> <p><a href="#">正規表現</a> の 減算 (を除く) 論理演算子。</p>

記号	名前と使用法
.	<p>ピリオド (46) : ロケールに応じて、<b>小数点</b>文字または数値グループ・セパレータ。 SetFormatItem() NLS クラス・メソッドを使用して構成可能です。</p> <p>オブジェクト・インスタンスのメソッドやプロパティを参照するために使用する<b>オブジェクト・ドット構文</b> (myinstance.Name)。</p> <p>Windows および UNIX® : パス名またはパス名の一部として、現在のディレクトリを指定します。<b>\$ZSEARCH</b> で使用されます。</p> <p><b>グローバル名</b>や<b>ルーチン名</b>に組み込まれる場合があります。</p> <p><b>参照渡し</b>を指定する実際のパラメータ・リスト内の変数または配列名の接頭語。(例) SET x=\$\$Calc(num,.result)。</p> <p><b>パターン・マッチの繰り返し文字</b>。</p> <p><b>正規表現</b>の単一文字ワイルドカード。</p>
..	<p>二重ピリオド: <b>相対ドット構文</b> : 現在のオブジェクトのメソッドまたはプロパティを指定する接頭語。(例) WRITE ..foo()</p> <p>Windows および UNIX® : パス名またはパス名の一部として、現在のディレクトリの親ディレクトリを指定します。<b>\$ZSEARCH</b> で使用されます。</p>
..#	<p>二重ピリオド、シャープ記号 : <b>同じクラスのメソッド内からのクラス・パラメータ</b>を参照するための接頭語。(例) WRITE ..#MyParam</p>
...	<p>三重ピリオド(省略記号) : <b>可変数のパラメータ</b>を指定するために使用される、<b>仮パラメータ・リスト</b>または<b>実パラメータ・リスト</b>内の最後の (または唯一の) パラメータの後ろに追加される接尾語。(例) Calc(x,y,params...)。この構文は、通常は Method %DispatchMethod(Method As %String,Params...) などの<b>動的ディスパッチ・メソッド</b>と共に使用されます。</p> <p><b>ZWRITE</b> の出力では、末尾の省略記号は文字列が切り捨てられたことを示します。</p> <p><b>ターミナル・プロンプト</b>では、先頭の文字は、長い暗黙のネームスペースが最後の 24 文字に切り捨てられたことを示します。</p> <p>(コード内でのリテラルの省略記号の使用を、ドキュメント内の形式規約の使用と混同しないください。省略記号は、引数が複数回繰り返される、またはコード・セクションが意図的に省略されていることを示します。)</p>
/	<p>スラッシュ (47) : <b>除算演算子</b> (残余あり)。</p> <p><b>OPEN</b>、<b>CLOSE</b>、<b>USE</b> コマンドでは、入出力キーワード・パラメータ接頭語。<b>READ</b> と <b>WRITE</b> コマンドでは、デバイス・コントロール・ニーモニック接頭語。</p> <p><b>ZBREAK</b> コマンドでは、サブコマンドの接頭語です。</p>
//	<p>二重スラッシュ : <b>1 行コメント</b>文字。</p>
///	<p>三重スラッシュ : <b>1 行コメント</b>文字。マクロ・コメントの列 1 で使用できます。</p>
/*	<p>スラッシュ・アスタリスク : <b>複数行コメント</b> の開始文字。コメントは、*/ で終了します。</p>

記号	名前と使用法
:	<p>コロン (58) : コマンドでの<b>後置条件文字</b>。(例) WRITE:x=0 “nothing”。</p> <p>OPEN、USE、CLOSE、JOB、LOCK、READ、ZBREAK などのコマンドにおける、引数のプレースホルダ区切り文字または引数内のパラメータの区切り文字。(例) LOCK var1:10、+var2:15、OPEN “ TCP 4”:(4200:“PSTE”::32767:32767)</p> <p>\$CASE および \$SELECT 関数では、対になった項目である test:value を指定するのに使用されます。</p> <p>JSON オブジェクトでは、key:value ペアを指定するために使用されます。(例) SET JSONobj={ “name” : “Sam” }。</p> <p>\$JOB 特殊変数値では、プロセス ID (PID) とノード名を分離します。(例) 11368:MYCOMPUTER。</p> <p>SQL トリガ・コードなど、埋め込み ObjectScript コード内の ObjectScript ラベルを示す接頭語。この場合、1 列目でラベルをコード化することはできません。</p>
;	セミコロン (59) : 1 行コメント文字。
::	二重セミコロン : 維持された 1 行コメント文字。
<	より小さい (60) : より小さい演算子。
<=	以下記号 : 以下演算子。
'<	Not 演算子、より小さい : 以上演算子。
=	<p>等記号 (61) : 等しい比較演算子。</p> <p>SET コマンドでは、代入演算子。</p>
'=	Not 演算子、等記号 : 等しくない比較演算子。
>	より大きい (62) : より大きい演算子。
>=	以上記号 : 以上演算子。
'>	Not 演算子、より大きい : 以下演算子。
?	<p>疑問符 (63) : パターン・マッチ演算子。</p> <p>正規表現の 0 または 1 文字の修飾子接尾語。正規表現のモード接頭語。(例) (?i) case mode on; (?-i) case mode off。</p> <p>\$ZCONVERT 変換テーブルの結果で、変換できない文字を表します。</p> <p>\$ZSEARCH では、単一文字のワイルドカード。</p> <p>READ コマンドや WRITE コマンドでは行開始文字。</p> <p>ZBREAK コマンドでは、ヘルプ・テキストを表示します。</p> <p>ダイナミック SQL 内で %Execute() メソッドによって提供される入力パラメータ。</p>
?#	疑問符とシャープ記号 : 正規表現の埋め込みコメントの接頭語。(例) (?# this is a comment)。
@	<p>アット記号 (64) : 間接演算子。添え字間接演算子の場合、@array@(subscript) のようになります。</p>

記号	名前と使用法
A, a	文字 “A” (65,97) : <a href="#">パターン・マッチ・コード</a> (? に続く)。
C, c	文字 “C” (67,99) : <a href="#">パターン・マッチ・コード</a> (? に続く)。
E, e	文字 “E” (69,101) : <a href="#">科学的記数法演算子</a> (例えば、4E3=4000)。大文字の “E” は、標準の指数演算子です。小文字の “e” は、%SYSTEM.Process クラスの ScientificNotation() メソッドを使用して設定可能な指数演算子です。 <a href="#">パターン・マッチ・コード</a> (? に続く)。
I, i	文字 “I” (73,105) : <a href="#">\$NUMBER</a> 関数では整数文字。
i%	文字 “i” パーセント : <a href="#">インスタンス変数</a> 構文 : i%propertyname。
L, l	文字 “L” (76,108) : <a href="#">パターン・マッチ・コード</a> (? に続く)。
N, n	文字 “N” (78,110) : <a href="#">パターン・マッチ・コード</a> (? に続く)。
P, p	文字 “P” (80,112) : <a href="#">パターン・マッチ・コード</a> (? に続く)。
U, u	文字 “U” (85,117) : <a href="#">パターン・マッチ・コード</a> (? に続く)。

記号	名前と使用法
[	角括弧 (開始) (91) : <a href="#">包含関係演算子</a> 。
[ ]	角括弧 (91,93) : <a href="#">拡張グローバル参照</a> <code>^[ "namespace" ]global</code> で、ネームスペース名、ディレクトリ名、または NULL 文字列を囲むために使用します。 <code>^[ "^[ " ]ppgname</code> または <code>^[ "^[ " , " " ]ppgname</code> の構文で <a href="#">プロセス・プライベート・グローバル</a> を指定するために使用します。  <a href="#">構造化システム変数</a> (SSVN) 内において、ネームスペース名 <code>^[ "namespace" ]GLOBAL( )</code> を囲い、 <a href="#">拡張 SSVN 参照</a> を指定します。  <a href="#">XECUTE</a> コマンドまたは <a href="#">プロシージャ定義</a> では、パブリック変数リストを囲みます。(例) <code>[ a,b,c ]</code> 。  <a href="#">ZWRITE</a> または引数なしの <a href="#">WRITE</a> コマンドの表示では、オブジェクト参照 (oref) を囲みます。  <a href="#">正規表現</a> はリスト [ABCD] または範囲 [A-D] 内のすべての文字と一致します。  <a href="#">JSON 動的配列の式</a> は、%DynamicArray のインスタンスを返します。(例) <code>SET JSONArray=[ 1,2,3 ]</code> 。
[ : ]	角括弧とコロンの <a href="#">正規表現</a> の文字タイプ・キーワード。(例) <code>[ :alpha: ]</code> 。
¥	バックスラッシュ (92) : <a href="#">整数除算演算子</a> (残余なし)。モジュロ (#) 演算子と共に使用して、ビット値を決定できます。例えば、 <code>\$ZA\16#2</code> は、\$ZA 16 ビットの値 (0 または 1) を返します。  <a href="#">正規表現</a> のエスケープ接頭語。 <a href="#">JSON 文字列</a> のエスケープ接頭語。例えば <code>\</code> 。
]	角括弧 (終了) (93) : <a href="#">後続関係演算子</a> 。
]]	二重角括弧 (終了) : <a href="#">前後関係演算子</a> 。

記号	名前と使用法
^	<p>キャレット (94) : <a href="#">グローバル変数名接頭語</a>。(例) ^myglobal(i)</p> <p><a href="#">ルーチン呼び出し接頭語</a> (例) DO ^routine や DO label^routine</p> <p><a href="#">暗黙のネームスペース接頭語</a> (例) ^system^dir</p> <p><a href="#">\$BITLOGIC</a> ビット文字列 XOR (排他的 or) 演算子。</p> <p><a href="#">正規表現</a> の文字列の最初のアンカー。(例) ^A. <a href="#">正規表現</a> の文字タイプ・キーワードの逆。(例) [.^alpha:]^^</p>
^^	<p>二重キャレット: 現在のシステムに対する<a href="#">暗黙のネームスペース接頭語</a>。形式は ^^ または ^^dirpath です。</p>
^\$	<p>キャレット・ドル記号 : <a href="#">構造化システム変数</a>の接頭語。(例) ^\$GLOBAL() または ^\$ "namespace" GLOBAL()</p>
^\$[	<p>キャレット・ドル括弧 : <a href="#">構造化システム変数</a> (SSVN) 内において、ネームスペース名 ^\$[ "namespace" ]GLOBAL() を囲い、<a href="#">拡張 SSVN 参照</a> を指定します。</p>
^\$	<p>キャレット・ドル・バー : <a href="#">構造化システム変数</a> (SSVN) 内において、ネームスペース名 ^\$[ "namespace" ]GLOBAL() を囲い、<a href="#">拡張 SSVN 参照</a> を指定します。</p>
^%	<p>キャレット・パーセント : <a href="#">システム・グローバルの接頭語</a>。(例) ^%utility または ^%qStream</p>
^(	<p>キャレット括弧 : <a href="#">ネイキッド・グローバル参照</a>。最新の添え字付きグローバル名は暗黙です。(例) ^(1,2)</p>
^[	<p>キャレット角括弧 (開始) : “角括弧” 参照。</p>
^	<p>キャレット・バー : 後に続く文字によって、以下ようになります。</p> <p>引用符付きネームスペース名、ディレクトリ名、または NULL 文字列を一組のバーが囲むグローバル参照である、<a href="#">拡張グローバル参照</a>。バーとそのコンテンツは、グローバル名の一部ではありません。(例) ^  " "   globname、または ^  "namespace"   globname。</p> <p>接頭語 ^  が付いた<a href="#">プロセス・プライベート・グローバル</a>。バーは、プロセス・プライベート・グローバル名の一部です。(例) ^  ppname。また、このプロセス・プライベート・グローバル ^  " "   ppname の構文としても有効です。</p> <p>引用符付きネームスペース名、ネームスペース名に解決する変数、または NULL 文字列を一組のバーが囲む、<a href="#">拡張ルーチン参照</a>。(例) DO ^  "namespace"   routine。</p>
-	<p>アンダースコア (95) : <a href="#">結合演算子</a>。</p> <p><a href="#">名前の最初の文字</a>。</p>

記号	名前と使用法
{ }	<p>中括弧 (123,125) : <b>TRY</b> と <b>CATCH</b> のブロック、または <b>IF</b>、<b>FOR</b>、<b>DO WHILE</b>、および <b>WHILE</b> のコマンドを持つ<b>プロシージャ</b>で使用されるコード・ブロックの区切り文字。</p> <p><b>SQL 計算コード</b>では、フィールド名を囲みます。例えば、<code>SET {Age}=18</code> や <code>SET {f1} = {f2}</code> のようにします。</p> <p><b>XECUTE</b> コマンドでは、変数がプライベートとして扱われるコードを囲みます。</p> <p><b>正規表現</b>の修飾子。(例) {5} = 5 回、{3,6} = 3 回以上ただし 6 回以下。<b>正規表現</b>の <code>%p</code> 接頭語を使用する文字タイプ文字コードまたはキーワード。(例) <code>%p[LL]</code>, <code>%p[lower]</code>。<b>正規表現</b>の <code>%N</code> 接頭語を使用する単一文字キーワード。(例) <code>%N[comma]</code>。</p> <p><b>JSON ダイナミック・オブジェクトの式</b>は、<code>%DynamicObject</code> のインスタンスを返します。(例) <code>SET JSONobj={ "name" : "Sam" }</code>。</p>
{*}	中括弧内のアスタリスク : <b>SQL 計算コード</b> では、現在の SQL フィールド名を指定します。
	<p>垂直バー (124) : <b>\$BITLOGIC</b> ビット文字列 OR 演算子。</p> <p><b>正規表現</b>の OR 論理演算子。</p> <p>他の用途は、<code>^ </code> および <code>^\$</code> を参照してください。</p>
	<p>二重垂直バー (バー・バー) : <b>OR 論理演算子</b> (簡易評価)。</p> <p>複合 ID インジケータ。InterSystems IRIS で使用し、生成された複合オブジェクト ID (連結 ID) を表示します。これは、<b>複数のプロパティに対して定義された IDKey</b> (<code>prop1    prop2</code>)、または<b>親子リレーションシップの ID</b> (<code>parent    child</code>) になります。</p>
~	<p>チルダ (126) : <b>\$BITLOGIC</b> ビット文字列 NOT (1 の補数) 演算子。</p> <p>Windows パス名の場合、長い名前での 8.3 比較を示します。(例) <code>c:\%PROGRA~1\</code>。圧縮ディレクトリ名を変換するには、<code>%Library.File</code> クラスの <code>NormalizeDirectory()</code> メソッドを使用します。</p> <p>UNIX では、パス名は現在のユーザのホーム・ディレクトリを表します。例 : <code>~myfile</code> や <code>~/myfile</code>。</p>



## ObjectScript で使用する省略形

ObjectScript で使用可能なコマンド、関数、および特殊変数の省略形の表です。

### 省略形の表

以下は、InterSystems IRIS® Data Platform の ObjectScript で使用される略名です。すべてではありませんが、ほとんどの ObjectScript コマンド、関数、および特殊変数には、略名があります。コード文字として使用するその他の文字の使用法は、“ObjectScript で使用する記号” の表を参照してください。

省略形	完全名
\$A	\$ASCII 関数
B	BREAK コマンド
C	CLOSE コマンド
\$C	\$CHAR 関数
D	DO コマンド または DO WHILE コマンドの DO キーワード
\$D	\$DATA 関数 (引数あり) または \$DEVICE 特殊変数 (引数なし)
\$E	\$EXTRACT 関数
\$EC	\$ECODE 特殊変数
\$ES	\$ESTACK 特殊変数
\$ET	\$ETRAP 特殊変数
F	FOR コマンド
\$F	\$FIND 関数
\$FN	\$FNUMBER 関数
G	GOTO コマンド
\$G	\$GET 関数
^\$G	^\$GLOBAL 構造化システム変数
H	HALT コマンド (引数なし) または HANG コマンド (引数あり)
\$H	\$HOROLOG 特殊変数
I	IF コマンド
\$I	\$INCREMENT 関数 (引数あり) または \$IO 特殊変数 (引数なし)
\$IN	\$INUMBER 関数
J	JOB コマンド
\$J	\$JUSTIFY 関数 (引数あり) または \$JOB 特殊変数 (引数なし)
^\$J	^\$JOB 構造化システム変数
K	KILL コマンド
\$K	\$KEY 特殊変数
L	LOCK コマンド



省略形	完全名
\$L	\$LENGTH 関数
^\$L	^\$LOCK 構造化システム変数
\$LB	\$LISTBUILD 関数
\$LD	\$LISTDATA 関数
\$LF	\$LISTFIND 関数
\$LFS	\$LISTFROMSTRING 関数
\$LG	\$LISTGET 関数
\$LI	\$LIST 関数
\$LL	\$LISTLENGTH 関数
\$LS	\$LISTSAME 関数
\$LTS	\$LISTTOSTRING 関数
\$LU	\$LISTUPDATE 関数
\$LV	\$LISTVALID 関数
M	MERGE コマンド
N	NEW コマンド
\$NA	\$NAME 関数
\$NC	\$NCONVERT 関数
\$NUM	\$NUMBER 関数
O	OPEN コマンド
\$O	\$ORDER 関数
P	PRINT コマンド
\$P	\$PIECE 関数 (引数あり) または \$PRINCIPAL 特殊変数 (引数なし)
Q	QUIT コマンド
\$Q	\$QUERY 関数 (引数あり) または \$QUIT 特殊変数 (引数なし)
\$QL	\$QLENGTH 関数
\$QS	\$QSUBSCRIPT 関数
R	READ コマンド
\$R	\$RANDOM 関数
^\$R	^\$ROUTINE 構造化システム変数
\$RE	\$REVERSE 関数
RET	RETURN コマンド
S	SET コマンド
\$S	\$SELECT 関数 (引数あり) または \$STORAGE 特殊変数 (引数なし)
\$SC	\$SCONVERT 関数

省略形	完全名
\$SEQ	\$SEQUENCE 関数
\$ST	\$STACK 関数 (引数あり) または \$STACK 特殊変数 (引数なし)
\$SY	\$SYSTEM 特殊変数
\$T	\$TEXT 関数 (引数あり) または \$TEST 特殊変数 (引数なし)
TC	TCOMMIT コマンド
\$TL	\$TLEVEL 特殊変数
\$TR	\$TRANSLATE 関数
TRO	TROLLBACK コマンド
TS	TSTART コマンド
U	USE コマンド
V	VIEW コマンド
\$V	\$VIEW 関数
W	WRITE コマンド
\$WA	\$WASCII 関数
\$WC	\$WCHAR 関数
\$WE	\$WEXTRACT 関数
\$WF	\$WFIND 関数
\$WL	\$WLENGTH 関数
\$WRE	\$WREVERSE 関数
X	XECUTE コマンド
\$X	\$X 特殊変数 (省略形なし)
\$Y	\$Y 特殊変数 (省略形なし)
\$ZA	\$ZA 特殊変数 (省略形なし)
ZB	ZBREAK コマンド
\$ZB	\$ZBOOLEAN 関数 (引数あり) または \$ZB 特殊変数 (引数なし、省略形なし)
\$ZC	\$ZCYC 関数 (引数あり) または \$ZCHILD 特殊変数 (引数なし)
\$ZCVT	\$ZCONVERT 関数
\$ZD	\$ZDATE 関数
\$ZDA	\$ZDASCII 関数
\$ZDC	\$ZDCHAR 関数
\$ZDH	\$ZDATEH 関数
\$ZDT	\$ZDATETIME 関数
\$ZDTH	\$ZDATETIMEH 関数
\$ZE	\$ZERROR 特殊変数

省略形	完全名
\$ZF	<a href="#">\$ZF 関数</a> (省略形なし)。 <a href="#">\$ZF(-100)</a> 、 <a href="#">\$ZF(-3)</a> 、 <a href="#">\$ZF(-4)</a> 、 <a href="#">\$ZF(-5)</a> 、および <a href="#">\$ZF(-6)</a> 関数も参照。
\$ZH	<a href="#">\$ZHEX 関数</a> (引数あり) または <a href="#">\$ZHOROLOG 特殊変数</a> (引数なし)
ZI	<a href="#">ZINSERT コマンド</a>
\$ZI	<a href="#">\$ZIO 特殊変数</a>
\$ZJ	<a href="#">\$ZJOB 特殊変数</a>
ZK	<a href="#">ZKILL コマンド</a>
ZL	<a href="#">ZLOAD コマンド</a>
\$ZLA	<a href="#">\$ZLASCII 関数</a>
\$ZLC	<a href="#">\$ZLCHAR 関数</a>
\$ZM	<a href="#">\$ZMODE 特殊変数</a>
ZN	<a href="#">ZNSPACE コマンド</a>
\$ZN	<a href="#">\$ZNAME 特殊変数</a>
\$ZO	<a href="#">\$ZORDER 特殊変数</a>
\$ZPOS	<a href="#">\$ZPOSITION 特殊変数</a>
ZP	<a href="#">ZPRINT コマンド</a>
\$ZP	<a href="#">\$ZPARENT 特殊変数</a>
\$ZQA	<a href="#">\$ZQASCII 関数</a>
\$ZQC	<a href="#">\$ZQCHAR 関数</a>
ZR	<a href="#">ZREMOVE コマンド</a>
\$ZR	<a href="#">\$ZREFERENCE 特殊変数</a>
ZS	<a href="#">ZSAVE コマンド</a>
\$ZS	<a href="#">\$ZSTORAGE 特殊変数</a>
\$ZSE	<a href="#">\$ZSEARCH 関数</a>
\$ZT	<a href="#">\$ZTIME 関数</a> (引数あり) または <a href="#">\$ZTRAP 特殊変数</a> (引数なし)
\$ZTH	<a href="#">\$ZTIMEH 関数</a>
\$ZTS	<a href="#">\$ZTIMESTAMP 特殊変数</a>
\$ZTZ	<a href="#">\$ZTIMEZONE 特殊変数</a>
\$ZU	<a href="#">\$ZUTIL 関数</a> (Caché/Ensemble ドキュメントを参照)
\$ZV	<a href="#">\$ZVERSION 特殊変数</a>
ZW	<a href="#">ZWRITE コマンド</a>
\$ZWA	<a href="#">\$ZWASCII 関数</a>
\$ZWC	<a href="#">\$ZWCHAR 関数</a>



# ObjectScript 演算子

## 単項プラス (+)

---

単一のオペランドを数値として解釈します。

### 詳細

単項プラス演算子 (+) は、単一のオペランドを数値として解釈します。オペランドが文字列値を持つ場合、それを数値に変換します。無効な文字に遭遇するまで、文字列の文字を数値として順番に解析することで、これを実行します。そして、適格な数値に変換した文字列の先頭部分を返します。

### 例

以下に例を示します。

#### ObjectScript

```
WRITE + "32 dollars and 64 cents"           // 32
```

文字列の先頭に数値文字がない場合、単項プラス演算子は、オペランドをゼロとします。以下に例を示します。

#### ObjectScript

```
WRITE + "Thirty-two dollars and 64 cents" // 0
```

単項プラス演算子は、数値に対して何も作用しません。正数または負数の符号も変更しません。以下に例を示します。

#### ObjectScript

```
SET x = -23
WRITE " x: ", x, ! // -23
WRITE "+x: ", +x, ! // -23
```

# 単項マイナス (-)

オペランドを数値として解釈すると、このオペランドの符号を反転します。

## 詳細

単項マイナス演算子 (-) は、数値として解釈されるオペランドの符号を反転します。

## 例

以下に例を示します。

### ObjectScript

```
SET x = -60
WRITE " x: ", x,! // -60
WRITE "-x: ", -x,! // 60
```

オペランドが文字列値を持つ場合、単項マイナス演算子はその文字列を数値として解釈し、符号を反転します。数値は、単項プラス演算子と同様の方法で解釈されます。以下に例を示します。

### ObjectScript

```
SET x = -23
WRITE -"32 dollars and 64 cents" // -32
```

ObjectScript は、2 オペランド算術演算子より単項マイナス演算子を優先します。ObjectScript はまず数値式を検査し、単項マイナス演算子を実行します。その後、式を評価して結果を算出します。

以下の例では、ObjectScript は文字列を読み取り、数値 2 を検出するとそこで停止します。その後、単項マイナス演算子をその値に適用し、連結演算子 ( ) を使用して、2 番目の文字列からの値 "Rats" を数値に結合します。

### ObjectScript

```
WRITE -"2Cats_" "Rats" // -2Rats
```

数値式の絶対値を返すには、[\\$ZABS](#) 関数を使用します。

## 加算 (+)

---

2 つのオペランドを数値として解釈すると、それらの数値合計を算出します。

### 詳細

加算演算子 (+) は、2 つの数値として解釈されるオペランドの和を算出します。

### 例

以下の例は、2 つの定義済みローカル変数を加算します。

#### ObjectScript

```
SET x = 4
SET y = 5
WRITE "x + y = ", x + y // 9
```

以下の例は、先行する数字を持つ 2 つのオペランドに対し文字列算術を実行し、その結果を加算します。

#### ObjectScript

```
WRITE "4 Motorcycles" + "5 bicycles" // 9
```

以下の例は、数値として評価されたオペランドの先行ゼロが、演算子の結果に何も作用しないことを示します。

#### ObjectScript

```
WRITE "007" + 10 // 17
```



## 減算 (-)

2 つのオペランドを数値として解釈すると、それらの数値の差を算出します。

### 詳細

減算演算子は、数値として解釈される 2 つのオペランドの差を算出します。この演算子は、先行する有効なすべての数値文字をオペランドの数値として解釈し、減算後の剰余を算出します。

### 例

以下の例は、2 つの数値リテラルを減算します。

#### ObjectScript

```
WRITE 2936.22 - 301.45 // 2634.77
```

以下の例は、2 つの定義済みローカル変数の減算を実行します。

#### ObjectScript

```
SET x = 4  
SET y = 5  
WRITE "x - y = ", x - y // -1
```

以下の例は、先行する数字を持つ 2 つのオペランドに対して文字列算術を実行し、その結果を減算します。

#### ObjectScript

```
WRITE "8 apples" - "4 oranges" // 4
```

オペランドが先行数値文字を持たない場合、ObjectScript は、その値をゼロと見なします。以下に例を示します。

#### ObjectScript

```
WRITE "8 apples" - "four oranges" // 8
```

## 乗算 (\*)

---

2 つのオペランドを数値として解釈すると、それらを乗算します。

### 詳細

乗算演算子は、2 つの数値として解釈されるオペランドの積を算出します。この演算子も、オペランドの数値として先行する有効な数値文字をすべて使用して、積を算出します。

### 例

以下の例は、2 つの数値リテラルの乗算を実行します。

#### ObjectScript

```
WRITE 9 * 5.5 // 49.5
```

以下の例は、2 つの定義済みローカル変数の乗算を実行します。

#### ObjectScript

```
SET x = 4  
SET y = 5  
WRITE x * y // 20
```

以下の例は、先行する数字を持つ 2 つのオペランドに文字列算術を実行し、その結果を乗算します。

#### ObjectScript

```
WRITE "8 apples" * "4 oranges" // 32
```

オペランドが先行数値文字を持たない場合、乗算演算子は、その値をゼロと解釈します。

#### ObjectScript

```
WRITE "8 apples"*"four oranges" // 0
```

## 除算 (/)

2 つのオペランドを数値として解釈すると、それらを除算します。

### 詳細

除算演算子は、数値として解釈される 2 つのオペランドの除算結果を算出します。この演算子も、オペランドの数値として先行する有効な数値文字をすべて使用して、商を算出します。

### 例

以下の例は、2 つの数値リテラルを除算します。

#### ObjectScript

```
WRITE 9 / 5.5 // 1.6363636363636364
```

以下の例は、2 つの定義済みローカル変数を除算します。

#### ObjectScript

```
SET x = 4  
SET y = 5  
WRITE x / y // .8
```

以下の例は、先行する数字を持つ 2 つのオペランドに文字列算術を実行し、その結果を除算します。

#### ObjectScript

```
WRITE "8 apples" / "4 oranges" // 2
```

オペランドが先行数値文字を持たない場合、除算演算子は、その値をゼロと見なします。以下に例を示します。

#### ObjectScript

```
WRITE "eight apples" / "4 oranges" // 0  
// "8 apples"/"four oranges" generates a <DIVIDE> error
```

上記の 2 番目の演算は無効です。ゼロによる数値の除算は許可されていないからです。ObjectScript は、<DIVIDE> エラー・メッセージを返します。

## 整数除算 (¥)

---

2 つのオペランドを数値として解釈すると、それらの整数除算の結果を算出します。

### 詳細

整数除算演算子は、左のオペランドを右のオペランドで除算した整数の結果を算出します。剰余を返さず、結果も丸めません。

### 例

以下の例は、2 つの整数オペランドを整数除算します。ObjectScript は、結果の小数部分を返しません。

#### ObjectScript

```
WRITE "355 \ 113 = ", 355 \ 113 // 3
```

以下の例は、文字列算術演算を実行します。整数除算演算子も、オペランドの値として先行する数値文字をすべて使用して、整数の結果を算出します。

#### ObjectScript

```
WRITE "8 Apples" \ "3.1 oranges" // 2
```

オペランドが先行数値文字を持たない場合、ObjectScript は、その値をゼロと見なします。整数をゼロで除算しようとすると、ObjectScript は <DIVIDE> エラーを返します。

## モジュロ (#)

2 つのオペランドを数値として解釈すると、それらのモジュロ演算の結果を算出します。

### 詳細

モジュロ演算子は、数値として解釈される 2 つのオペランドにモジュロ演算の結果を算出します。2 つのオペランドが正の場合、モジュロ演算の結果は、右のオペランドで左のオペランドを整数除算した剰余です。

### 例

以下の例は、数値リテラルにモジュロ演算を実行し、その剰余を返します。

#### ObjectScript

```
WRITE "37 # 10 = ", 37 # 10, ! // 7
WRITE "12.5 # 3.2 = ", 12.5 # 3.2, ! // 2.9
```

以下の例は、文字列算術演算を実行します。文字列の演算を行う場合、モジュロ演算子が適用される前に、文字列は数値に変換されます (“[変数のタイプと変換](#)” で説明されています)。したがって、以下の 2 つの式は同一です。

#### ObjectScript

```
WRITE "8 apples" # "3 oranges", ! // 2
WRITE 8 # 3 // 2
```

InterSystems IRIS® データ・プラットフォームは、先行数値文字のない文字列を 0 と解釈するため、このようなオペランドを右側に使用すると、<DIVIDE> エラーが発生します。

## べき乗 (\*\*)

2 つのオペランドを数値として解釈すると、それらのべき乗した値を算出します。

### 詳細

指数演算子は、右のオペランドを指数として左のオペランドをべき乗した値を算出します。

- ・ `0**0`: ゼロのゼロ乗は 0 です。ただし、どちらかのオペランドが、IEEE 倍精度数の場合 (例えば、`0**$DOUBLE(0)` または `$DOUBLE(0)**0`)、ゼロのゼロ乗は 1 です。詳細は、“[\\$DOUBLE 関数](#)”を参照してください。
- ・ `0**n`: ゼロの正数 `n` 乗はゼロです。`0**$DOUBLE("INF")` の場合も同様です。ゼロの負数乗を求めようとするとエラーが発生します。このときに生成されるエラーは、標準的な負の数を指定した場合は `<ILLEGAL VALUE>` エラー、`$DOUBLE` の負数を指定した場合は `<DIVIDE>` エラーです。
- ・ `num**0`: ゼロ以外の数値 (正または負) のゼロ乗は 1 です。`$DOUBLE("INF")**0` の場合も同様です。
- ・ `1**n`: 1 の任意の数値 (正、負、またはゼロ) 乗は 1 です。
- ・ `-1**n`: -1 のゼロ乗は 1、-1 の 1 乗および -1 の -1 乗は -1 です。1 より大きい指数の場合は、以下を参照してください。
- ・ `num**n`: 正数 (整数または小数) の任意の数値 (整数または小数、正または負) 乗は正数になります。
- ・ `-num**n`: 負数 (整数または小数) の偶数 (正または負) 乗は正数になります。負数 (整数または小数) の奇数 (正または負) 乗は負数になります。
- ・ `-num**.n`: 負数の小数乗を求めようとすると、`<ILLEGAL VALUE>` エラーが発生します。
- ・ `$DOUBLE("INF")**n`: 無限数 (正または負) の 0 乗は 1 です。無限数 (正または負) の正数 (整数、小数、または INF) 乗は INF です。無限数 (正または負) の負数 (整数、小数、または INF) 乗は 0 です。
- ・ `$DOUBLE("NAN")`: 指数演算子のどちら側の NAN であっても、他方のオペランドの値に関係なく NAN が返ります。

非常に大きな指数を指定すると、値のオーバーフローやアンダーフローが発生することがあります。

- ・ `num**nnn`: 1 より大きな正または負の数の、大きな正数乗 (`9**153` や `-9.2**152` など) を求めようとすると、`<MAXNUMBER>` エラーが発生します。
- ・ `num**-nnn`: 1 より大きな正または負の数の、大きな負数乗 (`9**-135` や `-9.2**-134` など) はゼロになります。
- ・ `.num**nnn`: 1 未満の正または負の数の、大きな正数乗 (`.22**196` や `-.2**184` など) はゼロになります。
- ・ `.num**-nnn`: 1 未満の正または負の数の、大きな負数乗 (`.22**-196` や `-.2**-184` など) を求めようとすると、`<MAXNUMBER>` エラーが発生します。

InterSystems IRIS® データ・プラットフォームの数値として使用できる最大値を超える指数を使用すると、`<MAXNUMBER>` エラーが発生するか、自動的に IEEE 倍精度浮動小数点数に変換されます。この自動変換は、`%SYSTEM.Process` クラスの `TruncateOverflow()` メソッドを使用してプロセスごとに指定するか、または `Config.Miscellaneous` クラスの `TruncateOverflow` プロパティを使用してシステム全体で指定します。詳細は、“[\\$DOUBLE 関数](#)”を参照してください。

### 例

以下の例は、2 つの数値リテラルをべき乗します。

#### ObjectScript

```
WRITE "9 ** 2 = ", 9 ** 2, ! // 81
WRITE "9 ** -2 = ", 9 ** -2, ! // .01234567901234567901
WRITE "9 ** 2.5 = ", 9 ** 2.5, ! // 242.9999999994422343
```

以下の例は、2 つの定義済みローカル変数をべき乗します。

#### ObjectScript

```
SET x = 4, y = 3  
WRITE "x ** y = ", x ** y, ! // 64
```

以下の例は、文字列算術演算を実行します。指数演算子も、オペランドの値として先行する数値文字をすべて使用して、結果を算出します。

#### ObjectScript

```
WRITE "4 apples" ** "3 oranges" // 64
```

オペランドが先行数値文字を持たない場合、指数演算子は、その値をゼロと見なします。

以下の例は、指数を使用した数値の平方根の算出方法です。

#### ObjectScript

```
WRITE 256 ** .5 // 16
```

べき乗計算は、[\\$ZPOWER](#) 関数を使用しても可能です。

## より小さい (<)

---

2 つのオペランドを数値として解釈すると、左のオペランドが数値的に右のオペランドより小さいかどうかを判断します。

### 詳細

より小さい関係演算子は、左のオペランドが右のオペランドより数値的に小さいかどうかを判断します。ObjectScript は、両方のオペランドを数値的に評価して、左のオペランドが右のオペランドより数値的に小さい場合、True (1) のブーリアン値を返します。左のオペランドが右のオペランドと数値的に等しい、または大きい場合、False (0) のブーリアン値を返します。以下に例を示します。

### 例

#### ObjectScript

```
WRITE 9 < 6
```

これは、0 を返します。

#### ObjectScript

```
WRITE 22 < 100
```

これは、1 を返します。



## より大きい (>)

2 つのオペランドを数値として解釈すると、左のオペランドが数値的に右のオペランドより大きいかどうかを判断します。

### 詳細

より大きい関係演算は、左のオペランドが右のオペランドより数値的に大きいかどうかを判断します。ObjectScript は、2 つのオペランドを数値的に評価し、左のオペランドが右のオペランドより数値的に大きい場合、True (1) を返します。左のオペランドが右のオペランドと数値的に等しい、または小さい場合、False (0) の論理値を返します。以下に例を示します。

### 例

#### ObjectScript

```
WRITE 15 > 15
```

これは、0 を返します。

#### ObjectScript

```
WRITE 22 > 100
```

これは、0 を返します。

## 以下 (<= または '>')

2 つのオペランドを数値として解釈すると、左のオペランドが数値的に右のオペランド以下かどうかを判断します。

### 詳細

以下のようにして、以下関係演算子を記述できます。

- ・ より小さい関係演算子 (<) と等値演算子 (=) を結合します。2 つの演算子のどちらかが TRUE を返す場合、これらの演算子の組み合わせによって TRUE が返されます。
- ・ より大きい関係演算子 (>) と共に単項否定演算子 (!) を使用します。共に使用する 2 つの演算子は、より大きい関係演算の論理値を反転します。

ObjectScript は、左のオペランドが右のオペランドより数値的に小さい、または等しい場合、True (1) の結果を返します。左のオペランドが右のオペランドよりも数値的に大きい場合、False (0) の結果を返します。

### 例

以下のいずれか方法で、以下関係演算を記述できます。

```
operand_A <= operand_B  
operand_A '>' operand_B  
!(operand_A > operand_B)
```

以下の例は、以下関係演算で 2 つの変数をテストします。両方の変数の値が等しいため、結果は True となります。

#### ObjectScript

```
SET A="55",B="55"  
WRITE A'>B
```

これは、1 を返します。

## 以上 (>= または '<')

2 つのオペランドを数値として解釈すると、左のオペランドが数値的に右のオペランド以上かどうかを判断します。

### 詳細

以下のようにして、以上関係演算を記述できます。

- ・ より大きい関係演算子 (>) と等値演算子 (=) を結合します。2 つの演算子のどちらかが TRUE を返す場合、これらの演算子の組み合わせによって TRUE が返されます。
- ・ より小さい関係演算子 (<) と共に否定演算子 (!) を使用します。共に使用する 2 つの演算子は、より小さい関係演算の真偽値を反転します。

ObjectScript は、左のオペランドが右のオペランドより数値的に大きい、または等しい場合、True (1) の結果を返します。左のオペランドが右のオペランドよりも数値的に小さい場合、False (0) の結果を返します。

### 例

以下のいずれかの方法で、以上関係演算を記述できます。

```
operand_A >= operand_B  
operand_A '< operand_B  
!(operand_A < operand_B)
```

## 否定 (!)

---

ブーリアン型オペランドの真偽値を反転します。

### 詳細

否定演算子は、ブーリアン型オペランドの真偽値を反転します。オペランドが True (1) の場合、否定演算子は False (0) になります。オペランドが False (0) の場合、否定演算子は True (1) になります。

### 例

例えば、以下の文は False (0) の結果を返します。

#### ObjectScript

```
SET x=0
WRITE x
```

一方、以下の文は True (1) を返します。

#### ObjectScript

```
SET x=0
WRITE !x
```

比較演算子で否定演算子を使用すると、比較の意味が反転します。例えば、以下の文は False (0) の結果を返します。

#### ObjectScript

```
WRITE 3>5
```

しかし、以下の例は True (1) の結果を表示します。

#### ObjectScript

```
WRITE 3!'>5
```

## 論理積 (& または &&)

オペランドの両方の値が True (1) であるかどうかをテストします。

### 詳細

論理積演算子は、オペランドの両方の値が True (1) であるかどうかをテストします。オペランドが両方とも True の場合 (つまり、数値として計算した場合、ゼロ以外の値となる)、ObjectScript は、True (1) を返します。それ以外の場合、False (0) を返します。

論理積演算子には、以下の 2 つの形式があります。

- ・ & 演算子は、両方のオペランドを評価し、いずれかのオペランドの値がゼロの場合、False (0) を返します。それ以外は True (1) を返します。
- ・ && 演算子は、左のオペランドを評価し、そのオペランドの値がゼロの場合、False (0) を返します。左のオペランドがゼロ以外の場合にのみ、&& 演算子は右のオペランドを評価します。右のオペランドの評価がゼロの場合、False (0) を返します。それ以外は True (1) を返します。

### 例

以下の例は、2 つのゼロ以外のオペランドを True と評価して True (1) を返します。

#### ObjectScript

```
SET A=-4,B=1
WRITE A&B // TRUE (1)
```

#### ObjectScript

```
SET A=-4,B=1
WRITE A&&B // TRUE (1)
```

以下の例は、True と False のオペランドをそれぞれ評価して False (0) を返します。

#### ObjectScript

```
SET A=1,B=0
WRITE "A = ",A,!
WRITE "B = ",B,!
WRITE "A&B = ",A&B,! // FALSE (0)
SET A=1,B=0
WRITE "A&&B = ",A&&B,! // FALSE (0)
```

以下の例は、& 演算子と && 演算子の違いを示します。以下の例では、左のオペランドは False (0) と評価され、右のオペランドは定義されていません。この場合、& と && 演算子の結果が異なることに注意してください。

- ・ & 演算子は両方のオペランドを評価し、<UNDEFINED> エラーを生じます。

## ObjectScript

```
TRY {
    KILL B
    SET A=0
    WRITE "variable A defined?: ", $DATA(A), !
    WRITE "variable B defined?: ", $DATA(B), !
    WRITE A&B
    WRITE !, "Success"
    RETURN
}
CATCH exp
{
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE !, "System exception", !
        WRITE "Name: ", $ZCVT(exp.Name, "O", "HTML"), !
        WRITE "Data: ", exp.Data, !!
    }
    ELSE { WRITE "not a system exception" }
}
```

・ && 演算子は左のオペランドのみを評価し、False (0) を返します。

## ObjectScript

```
TRY {
    KILL B
    SET A=0
    WRITE "variable A defined?: ", $DATA(A), !
    WRITE "variable B defined?: ", $DATA(B), !
    WRITE A&&B
    WRITE !, "Success"
    RETURN
}
CATCH exp
{
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE !, "System exception", !
        WRITE "Name: ", $ZCVT(exp.Name, "O", "HTML"), !
        WRITE "Data: ", exp.Data, !!
    }
    ELSE { WRITE "not a system exception" }
}
```

# 論理和 (! または ||)

両方のオペランドが True かどうかをテストします。

## 詳細

論理和演算子は、いずれか一方のオペランドが True の値を持つ場合、あるいは両方のオペランドが True (1) の値を持つ場合、True (1) を返します。論理和演算子は、両方のオペランドが False (0) の場合にのみ False (0) を返します。

論理和演算子には、以下の 2 つの形式があります。

- ・ ! 演算子は、両方のオペランドを評価し、両方のオペランドの値がゼロの場合、False (0) を返します。それ以外は True (1) を返します。
- ・ || 演算子は、左のオペランドを評価します。左のオペランドがゼロ以外の値に評価された場合、|| 演算子は、右のオペランドを評価せずに True (1) を返します。左のオペランドがゼロの場合にのみ、|| 演算子は、右のオペランドを評価します。右のオペランドがゼロの場合、False (0) を返します。それ以外は True (1) を返します。

## 例

以下の例は、2 つの True の (ゼロではない) オペランドに論理和演算を実行し、True の結果を返します。

### ObjectScript

```
SET A=5,B=7
WRITE "A!B = ",A!B,!
SET A=5,B=7
WRITE "A||B = ",A||B,!
```

上記は、両方とも True (1) を返します。

以下の例は、False のオペランドと True のオペランドに論理和演算を実行し、True の結果を返します。

### ObjectScript

```
SET A=0,B=7
WRITE "A!B = ",A!B,!
SET A=0,B=7
WRITE "A||B = ",A||B,!
```

上記は、両方とも True (1) を返します。

以下の例は、2 つの False のオペランドを評価し、False の結果を返します。

### ObjectScript

```
SET A=0,B=0
WRITE "A!B = ",A!B,!
SET A=0,B=0
WRITE "A||B = ",A||B,!
```

上記は、両方とも False (0) を返します。

## 論理積否定 (NAND) ('&)

---

両方のオペランドに適用された & 論理積演算の論理値を反転します。

### 詳細

論理積否定演算子は、両方のオペランドに適用された & 論理積演算の論理値を反転します。いずれかのオペランド、もしくは両方のオペランドが False である場合、True (1) を返します。両方のオペランドが True の場合、False を返します。

&& 論理積演算子の前に、否定演算子を付けることはできません。'&& という形式はサポートされていません。ただし、以下の形式はサポートされています。

```
'(operand && operand)
```

### 例

以下の例では、2 つの対応する論理積否定演算を実行します。各演算で、1 つの False (0) と 1 つの True (1) のオペランドが評価され、True (1) の値が返されます。

#### ObjectScript

```
SET A=0,B=1
WRITE !,A'&B    // Returns 1
WRITE !,'(A&B) // Returns 1
```

以下の例では、&& 論理積演算を実行することにより、論理積否定演算を実行した後、否定演算を使用して、結果を反転させます。&& 演算は、最初のオペランドをテストし、ブーリアン値が False (0) のため、&& は 2 番目のオペランドをテストしません。否定演算は、式が True (1) を返すように、結果のブーリアン値を反転させます。

#### ObjectScript

```
SET A=0
WRITE !,'(A&&B)    // Returns 1
```



# 論理和否定演算 (NOR) (!)

両方のオペランドが False かどうかをテストします。

## 詳細

論理和否定演算子は、両方のオペランドが False の場合、True (1) の結果を返します。どちらか一方のオペランドが True の場合、あるいは両方のオペランドが True の場合、False (0) の結果を返します。

|| 論理和演算子の前に、否定演算子を付けることはできません。' || ' という形式はサポートされていません。ただし、以下の形式はサポートされています。

```
'(operand || operand)
```

## 例

以下の論理和否定演算の例は、2 つの False のオペランドを評価して、True の結果を返します。

### ObjectScript

```
SET A=0,B=0
WRITE "A!B = ",A!B    // Returns 1
```

### ObjectScript

```
SET A=0,B=0
WRITE "'(A!B) = ",'(A!B)    // Returns 1
```

以下の論理和否定演算の例は、1 つの True のオペランドと 1 つの False のオペランドを評価して、False の結果を返します。

### ObjectScript

```
SET A=0,B=1
WRITE "A!B = ",A!B    // Returns 0
```

### ObjectScript

```
SET A=0,B=1
WRITE "'(A!B) = ",'(A!B)    // Returns 0
```

以下の論理和否定演算の例では、左のオペランドを評価し、その結果が True (1) なので、右のオペランドを評価しません。否定演算は、式が False (0) を返すように、結果のブーリアン値を反転させます。

### ObjectScript

```
SET A=1
WRITE "'(A|B) = ",'(A|B)    // Returns 0
```

## 文字列の連結 ( )

2 つのオペランドを数値として解釈すると、それらを連結します。

### 詳細

文字列連結演算子 ( ) は、そのオペランドを文字列と解釈し、文字列値を返す二項 (2 オペランド) 演算子です。

連結演算子を使用して、文字列リテラル、数字、式、変数を結合します。以下の形式をとります。

operand\_operand

連結演算子は、右のオペランドを左のオペランドの後に結合させた文字列を結果として返します。連結演算子は、そのオペランドを特に解釈せず、文字列値として扱います。

### 例

以下の例は、2 つの文字列を結合します。

#### ObjectScript

```
WRITE "High_" "chair"
```

上記は、Highchair を返します。

ある数値リテラルを別の数値リテラル、または非数値文字列に連結すると、InterSystems IRIS は最初にそれぞれの数値をキャノニック形式に変換します。以下の例は、2 つの数値リテラルを結合します。

#### ObjectScript

```
WRITE 7.00_+008
```

これは、78 を返します。

以下の例は、数値リテラルと数値文字列を結合します。

#### ObjectScript

```
WRITE ++7.00_" +007"
```

これは、文字列 7+007 を返します。

以下の例は、2 つの文字列と NULL 文字列を結合します。

#### ObjectScript

```
SET A="ABC"_"_" "DEF"  
WRITE A
```

これは、"ABCDEF" を返します。

NULL 文字列は、文字列の長さに影響しないため、無数の NULL 文字列を文字列に結合できます。

文字列の長さには上限があります。["文字列長の制限"](#) を参照してください。文字列を連結しようと試みた結果、その文字列がこの最大文字列サイズを超えた場合、<MAXSTRING> エラーになります。

複数の連結を伴う ObjectScript 文は、アトミック (全か無か) 処理です。<MAXSTRING> エラーが発生した場合、連結によって拡大された変数は、連結前のその値を保持します。例えば、bigstr が長さ 2,000,000 の文字列である場合、連結 SET bigstr=bigstr\_"abc"\_bigstr を試みると <MAXSTRING> エラーになります。bigstr の長さは 2,000,000 のままになります。

## 連結エンコード文字列

ObjectScript 文字列には、それらの文字列を結合できるかどうかを制限できる内部的なエンコードが含まれるものがあります。

- ・ **ビット文字列**は、別のビット文字列、ビット文字列でない文字列、空の文字列 ("" ) のどれとであれ、結合できません。これを実行しようとすると、生成される文字列を使用するときに、`<INVALID BIT STRING>` エラーが返されます。
- ・ **リスト構造文字列**は、別のリスト構造文字列または空の文字列 ("" ) と結合できます。しかし、リスト以外の文字列とは連結できません。これを実行しようとすると、生成される文字列を使用するときに、`<LIST>` エラーが返されます。
- ・ **JSON 文字列**は、別の JSON 文字列、JSON 文字列でない文字列、空の文字列 ("" ) のどれとであれ、結合できません。これを実行しようとすると、生成される文字列を使用するときに、`<INVALID OREF>` エラーが返されます。

## 等価 (=)

---

2 つのオペランドが文字列として等しいかを判断します。

### 詳細

等価演算子は、2 つのオペランドが文字列として等しいかを判断します。等価演算子を 2 つの文字列で実行すると、ObjectScript は、2 つのオペランドの文字順序が同一で、スペースを含め他の異なる文字を持たない同一の文字列の場合に True (1) を返します。それ以外の場合は False (0) を返します。以下に例を示します。

#### ObjectScript

```
WRITE "SEVEN"="SEVEN"
```

上記は、True (1) を返します。

### 例

等価演算子は、どちらのオペランドも数值的に解釈しません。例えば、以下の文は、2 つのオペランドが数值的には等価ですが、False (0) を返します。

#### ObjectScript

```
WRITE "007"="7"
```

両方のオペランドが数値である場合、等値演算子を使用して、数值的に等しいかどうかを判断できます。以下に例を示します。

#### ObjectScript

```
WRITE 007=7
```

上記は、True (1) を返します。

また、単項プラス演算子を使用して、強制的に数値変換を実行できます。以下に例を示します。

#### ObjectScript

```
WRITE +"007"="7"
```

上記は、True (1) を返します。

2 つのオペランドのタイプが異なる場合、両方のオペランドが文字列に変換され、それらの文字列が比較されます。文字列への変換時に、丸めおよび有効桁数のために誤差が生じる可能性があります。以下に例を示します。

#### ObjectScript

```
WRITE "007"=7,!
// converts 7 to "7", so FALSE (0)
WRITE 007="7",!
// converts 007 to "7", so TRUE (1)
WRITE 17.1=$DOUBLE(17.1),!
// converts both numbers to "17.1", so TRUE (1)
WRITE 1.2345678901234567=$DOUBLE(1.2345678901234567),!
// compares "1.2345678901234567" to "1.23456789012346", so FALSE (0)
```

## 不等価 ('=)

両方のオペランドに適用された等価演算子の論理値を反転します。

### 詳細

不等価演算は、否定演算子と等価演算子を併用して指定できます。以下の 2 とおりの方法で、不等価演算を記述できます。

```
operand != operand  
!(operand = operand)
```

不等価演算は、等価演算子を両方のオペランドに適用した場合の論理値を反転します。2 つのオペランドが等しくない場合、結果は True (1) となります。2 つのオペランドが等しい場合、結果は False (0) となります。

## 包含 (I)

---

右のオペランドの一連の文字が、左の文字の部分文字列であるかを判断します。

### 詳細

包含関係演算子は、右のオペランドの一連の文字が、左の文字の部分文字列であるかを判断します。左のオペランドが、右のオペランドの文字列を含んでいる場合、結果は True (1) となります。左のオペランドが、右のオペランドの文字列を含んでいない場合、結果は False (0) となります。右のオペランドが NULL 文字列の場合、結果は常に True となります。

### 例

以下の例は、L の文字列が S の文字列を含むかどうかを判断します。L は S を含んでいるため、結果は True (1) となります。

#### ObjectScript

```
SET L="Steam Locomotive",S="Steam"  
WRITE L[S]
```

以下の例は、P の文字列が S の文字列を含むかどうかを判断します。文字列中の文字の並びが異なるため (P はピリオド、S は感嘆符を持つ)、結果は False (0) となります。

#### ObjectScript

```
SET P="Let's play.",S="Let's play!"  
WRITE P[S]
```

## 非包含 (D)

オペランド A がオペランド B で表される文字列を含まない場合に True を返し、オペランド B で表される文字列を含む場合は False を返します。

### 詳細

非包含演算は、以下のいずれかの等価形式で、二項包含関係演算子と単項否定演算子を使用して記述できます。

operand A '[ operand B

'(operand A [ operand B)

非包含演算は、オペランド A がオペランド B で表される文字列を含まない場合に True を返し、オペランド B で表される文字列を含む場合は False を返します。

### 例

#### ObjectScript

```
SET itemA="abc"  
SET itemB="123"  
WRITE itemA'[itemB // displays 1
```

## 後続 ([])

---

左のオペランドの文字が、ASCII 文字順で右のオペランドの文字の後に来るかどうかを判断します。

### 詳細

後続関係演算子は、左のオペランドの文字が、ASCII 文字順で右のオペランドの文字の後に来るかどうかを判断します。後続関係演算子は、両方の文字列が、それぞれの最左端の文字から始まるものとします。このテストは、以下のいずれかの場合に終了します。

- ・ 左のオペランドの中に、対応する右のオペランドの文字と異なる文字が発見された場合
- ・ いずれかのオペランドに、比較する文字がない場合

ObjectScript は、左のオペランド内の最初の一意の文字が、右のオペランド内の対応する文字よりも高い ASCII 値を持つ場合（つまり、左のオペランドの文字が、右のオペランドの文字よりも ASCII 文字順で後に来る場合）、True の値を返します。右のオペランドが左のオペランドより短い、それ以外は等しい場合も、ObjectScript は True の値を返します。

ObjectScript は、以下のいずれかの条件が当てはまる場合、False の値を返します。

- ・ 左のオペランド内の最初の一意の文字が、右のオペランド内の対応する文字よりも低い ASCII 値を持つ場合
- ・ 左のオペランドが右のオペランドと同一の場合
- ・ 左のオペランドが右のオペランドよりも短い、それ以外は同一の場合

### 例

以下の例は、文字列 LAMPOON が、ASCII 文字順で文字列 LAMP の後に来るかどうかを判断します。結果は True です。

#### ObjectScript

```
WRITE "LAMPOON" ] "LAMP"
```

以下の例は、B の文字列が A の文字列の後に来るかどうかを判断します。ASCII 文字順では、BO が BL の後に来るため結果は True です。

#### ObjectScript

```
SET A="BLUE",B="BOY"  
WRITE B]A
```



## 非後続 ('])

左のオペランドが ASCII の照合順序で右のオペランドの後に続いているかどうかを判断します。

### 詳細

非後続関係演算は、以下のどちらかの等価形式で、後続関係演算子と否定演算子を使用して記述できます。

```
operand A ']'operand B  
'(operand A ] operand B)
```

非後続関係演算は、両方のオペランドのすべての文字が同一である場合、あるいはオペランド A の最初の一意の文字が、オペランド B の対応する文字よりも低い ASCII 値を持つ場合、True の値を返します。また、オペランド A の最初の一意の文字が、オペランド B の対応する文字よりも高い ASCII 値を持つ場合、False の値を返します。

### 例

以下の例では、CDE の C は ABC の A の後ろに来るため、結果は False となります。

#### ObjectScript

```
WRITE "CDE"']"ABC",!  
WRITE '("CDE"]"ABC")
```

上記は、False (0) を返します。

## 前後関係 (]])

---

左のオペランドが数値添え字の照合順序で右のオペランドの後に順番に並んでいるかを判定します。

### 詳細

前後関係演算子は、左のオペランドが数値添え字の照合順序で右のオペランドの後に順番に並んでいるかを判定します。数値照合順序では、NULL 文字列が最初に置かれます。次に負数から数値順に並べた[キャノニック形式の数値](#)、ゼロ、整数と続き、最後に数値以外の値が置かれます。

前後関係演算子は、照合順序上で第 1 オペランドが第 2 オペランドよりも後に位置している場合に True (1) を、それ以外の場合に False (0) を返します。以下に例を示します。

#### ObjectScript

```
WRITE 122]]2
```

上記は、True (1) を返します。

#### ObjectScript

```
WRITE "LAMPOON" ]]"LAMP"
```

上記は、True (1) を返します。

## 非前後関係 (']'])

左のオペランドが右のオペランドよりも後に位置していないかどうかを判断します。

### 詳細

非前後関係演算は、以下のいずれかの等価形式でも、後続関係演算子と共に否定演算子を使用して記述できます。

```
operand A ']] operand B  
'(operand A ]] operand B)
```

オペランド A がオペランド B と同一である場合、あるいはオペランド B が、オペランド A の後に順番に並んでいる場合、ObjectScript は、True の値を返します。オペランド A がオペランド B の後ろに順番に並んでいる場合、False の値を返します。

# パターン・マッチング (?)

指定された文字列が指定されたパターンと一致しているかどうかを判断します。

## 概要

ObjectScript のパターン・マッチング演算子は、文字列が 2 連続の大文字を含んでいるかどうかを判断します。

ObjectScript のパターン・マッチング演算子は、左オペランドの文字が、その右オペランドのパターンと正確に一致しているかどうかを判断します。この演算子はブーリアン値を返します。パターン・マッチング演算子は、パターンが左オペランドの文字パターンに一致した場合に True (1) の結果を返します。また、一致しなかった場合は False (0) の結果を返します。

以下の例は、文字列 ssn に有効な米国社会保障番号 (数字 3 桁、ハイフン、数字 2 桁、ハイフン、数字 4 桁) が含まれているかどうかをテストします。

### ObjectScript

```
SET ssn="123-45-6789"  
SET match = ssn ? 3N1 "-" 2N1 "-" 4N  
WRITE match
```

左オペランド (テスト値) と右オペランド (パターン) は、右オペランドの先頭の ? で区別されます。これら 2 つのオペランドは、以下の等価プログラムの例で示すように、1 つ以上の空白スペースで区切られる場合、もしくは空白スペースで区切られない場合もあります。

### ObjectScript

```
SET ssn="123-45-6789"  
SET match = ssn?3N1 "-" 2N1 "-" 4N  
WRITE match
```

? 演算子の後に空白が入ることはできません。パターン内の空白は引用符で囲まれた文字列内に置く必要があり、パターンの一部として解釈されます。

パターン・マッチング演算の一般的な形式は以下のとおりです。

operand?pattern

operand	パターンのテスト対象となる文字群である、文字列や数字として評価される式です。
pattern	? 記号 (不一致テストの場合は '?') で始まるパターン・マッチング・シーケンスです。このパターン・シーケンスは、1 つ以上の pattern-elements (パターン要素) の列、あるいは 1 つ以上の pattern-elements の列として評価される間接参照のいずれかとなります。

pattern-element は以下のいずれかで構成されます。

- repeat-count pattern-codes
- repeat-count literal-string
- repeat-count alternation

repeat-count	リピート・カウント – パターンの各インスタンスをマッチさせる具体的な回数です。repeat-count は、整数またはピリオド・ワイルドカード文字 (.) に評価できる文字です。任意の回数を指定するには、ピリオドを使用します。
pattern-codes	1 つまたは複数のパターン・コード。複数のコードを指定した場合は、それらのコードのいずれかと一致するとパターンの条件が満たされます。
literal-string	二重引用符で囲まれたリテラル文字列。
alternation	一連の pattern-element。この pattern-element のいずれかを選択して、オペランド文字列のセグメントとのパターン・マッチングに使用します。これにより、パターン指定で論理 OR 機能が得られます。

特定の文字や文字列を比較する場合、二重引用符で囲んだリテラル文字列をパターンで使用します。他の状況では、ObjectScript が提供する特殊なパターン・コードを使用します。特定のパターン・コードに関連付けられる文字は、(ある程度の) ロケール依存となります。以下のテーブルは、使用可能なパターン・コードとその意味です。

テーブル B-1: パターン・コード

Code	意味
A	任意のアルファベットの大文字または小文字に一致。ロケールの 8 ビット文字セットは、アルファベット文字が何であるかを定義します。英語のロケール (Latin-1 文字セットに基づく) の場合、これには ASCII 値 65 ~ 90 (A ~ Z)、97 ~ 122 (a ~ z)、170、181、186、192 ~ 214、216 ~ 246、および 248 ~ 255 が含まれます。
C	任意の ASCII 制御文字 (ASCII 値の 0 から 31 までと拡張 ASCII 値の 127 から 159 まで) と一致。
E	出力不能文字、空白文字、および制御文字を含む任意の文字と一致。
L	任意のアルファベットの小文字に一致。ロケールの 8 ビット文字セットは、小文字が何であるかを定義します。英語のロケール (Latin-1 文字セットに基づく) の場合、これには ASCII 値 97 ~ 122 (a ~ z)、170、181、186、223 ~ 246、および 248 ~ 255 が含まれます。
N	10 個の ASCII 数字 (0 から 9、ASCII 48 から 57) のいずれとも一致。
P	任意の句読点文字に一致。ロケールの文字セットは、拡張 (8 ビット) ASCII 文字セットに対して、句読点文字が何であるかを定義します。英語のロケール (Latin-1 文字セットに基づく) の場合、これには ASCII 値 32 ~ 47、58 ~ 64、91 ~ 96、123 ~ 126、160 ~ 169、171 ~ 177、180、182 ~ 184、187、191、215、および 247 が含まれます。
U	任意のアルファベットの大文字に一致。ロケールの 8 ビット文字セットは、大文字が何であるかを定義します。英語のロケール (Latin-1 文字セットに基づく) の場合、これには ASCII 値 65 ~ 90 (A ~ Z)、192 ~ 214、および 216 ~ 222 が含まれます。
R B M	キリル 8 ビット・アルファベット文字のマッピングに一致。R は任意のキリル文字 (ASCII 値の 192 から 255 まで) と一致します。B は大文字のキリル文字 (ASCII 値の 192 から 223 まで) と一致します。M は小文字のキリル文字 (ASCII 値の 224 から 255 まで) と一致します。これらのパターン・コードは、Russian 8 ビット Windows ロケール (ruw8) のみで意味を持ちます。他のロケールでは、実行には成功しますが、いずれの文字とも一致しません。
ZFWCHARZ	日本語の全角文字セットのいずれの文字とも一致。ZFWCHARZ は、漢字などの全角文字や、一部のターミナル・エミュレータによって表示されたときに二重セルを占有する多くの漢字以外の文字に一致します。また、ZFWCHARZ は JIS2004 標準で定義された 303 のサロゲート・ペア文字とも一致して、各サロゲート・ペアを単一文字として扱います。例えば、サロゲート・ペア文字 \$WC(131083) は、?1ZFWCHARZ に一致します。このパターン・マッチング・コードには、日本語ロケールが必要となります。詳細は、“ <a href="#">\$ZENKAKU</a> ” 関数を参照してください。
ZHWKATAZ	日本語の半角かな文字セットのいずれの文字とも一致。これらは、Unicode 値 65377 (FF61) ~ 65439 (FF9F) までです。このパターン・マッチング・コードには、日本語ロケールが必要となります。詳細は、“ <a href="#">\$ZENKAKU</a> ” 関数を参照してください。

パターン・コードは大文字と小文字を区別しません。大文字と小文字いずれも指定できます。例えば、?5N と ?5n は等価です。複数のパターン・コードを指定して、特定の文字または文字列に一致させることができます。例えば、?1NU は数字または大文字のいずれかに一致します。

ASCII 文字セットは、制約の多い 7 ビット文字セットではなく拡張された 8 ビット文字セットを表します。

注釈 二重引用符文字と一致するパターンは、異なる NLS ロケールを使用して InterSystems IRIS 実装からデータが提供されている場合に特に、不整合な結果を生じる場合があります。まっすぐな二重引用符文字 (\$CHAR(34) = ") は、句読点文字として照合されます。方向性のある二重引用符文字 (巻いている引用符) は、句読点文字として照合されません。8 ビットの方向性のある二重引用符文字 (\$CHAR(147) = “ と \$CHAR(148) = ”) は、制御文字として照合されます。Unicode の方向性のある二重引用符文字 (\$CHAR(8220) = “ と \$CHAR(8221) = ”) は、句読点文字としても、制御文字としても照合されません。

パターン・マッチング演算子は、包含関係演算子 (I) とは異なります。包含関係演算子は、左辺のオペランドの部分文字列が右辺のオペランドと一致する場合も、True (1) を返します。また、包含関係式は、パターン・マッチング演算子で使える一連のオプションを提供しません。包含関係式では、単一の文字列のみ右辺のオペランドで使えます。特殊コードは使用できません。

例えば、変数 var2 に値 "abc" が含まれているとします。以下のパターン・マッチ式を考えてみます。

### ObjectScript

```
SET match = var2?2L
```

var2 は、2 つではなく 3 つの小文字を持っているため、match は False (0) の結果を返します。

以下に、基本的なパターン・マッチングの例を示します。

### ObjectScript

```
PatternMatchTest
SET var = "O"
WRITE "Is the letter O",!

WRITE "...an alphabetic character? "
WRITE var?1A,!

WRITE "...a numeric character? "
WRITE var?1N,!

WRITE "...an alphabetic or ",!," a numeric character? "
WRITE var?1AN,!

WRITE "...an alphabetic or ",!," a ZENKAKU Kanji character? "
WRITE var?1AZFWCHARZ,!

WRITE "...a numeric or ",!," a HANKAKU Kana character? "
WRITE var?1ZHWKATAZN
```

以下の項目を指定して、パターン・コードの範囲を拡張できます。

- ・ [パターンの発生回数](#)
- ・ [複数パターン](#)
- ・ [組み合わせパターン](#)
- ・ [不確定パターン](#)
- ・ [交互パターン](#)

## パターン・カウントの指定

以下の形式で、目的のオペランドで パターン が繰り返し発生できる回数の範囲を定義します。

n.n

1 番目の n は範囲の下限を、2 番目の n は上限を定義します。

以下の例では、変数 var3 に文字列 "AB" の複数のコピーが含まれていて、その他の文字は含まれていないものとします。1.4 は "AB" が 1 ~ 4 回出現すると認識されていることを示します。

## ObjectScript

```
SET match = var3?1.4"AB"
```

var3 = ABABAB の場合、この式は、var3 に "AB" が 3 つしか含まれていない場合でも、True (1) の結果を返します。別の例として、以下の式を考えてみます。

## ObjectScript

```
SET match = var4?1.6A
```

この式は、var4 に 1 個から 6 個までの英文字が含まれているかどうかをチェックします。var4 に英文字がまったく含まれていないか、7 文字以上の英文字が含まれているか、または英文字以外の文字が含まれている場合、False (0) の結果を返します。

どちらか一方の n を省略すると、ObjectScript は既定値を提供します。1 番目の n の既定値はゼロ (0) です。2 番目の n の既定値は任意の値です。以下の例を確認してください。

## ObjectScript

```
SET match = var5?.E1"AB".E
```

この例は、var5 に、パターン文字列 "AB" が少なくとも 1 つ含まれている限り、True (1) を返します。

## 複数パターンの指定

複数パターンを定義するには、任意の長さで n とパターンを組み合わせます。以下の例を確認してください。

## ObjectScript

```
SET match = date?2N1"/"2N1"/"2N
```

この式は、mm/dd/yy 形式で日付値をチェックします。文字列 "4/27/98" の場合、月の値が 1 桁なので False (0) を返します。1 桁の月と 2 桁の月の両方を検出するには、以下のように変更します。

## ObjectScript

```
SET match = date?1.2N1"/"2N1"/"2N
```

1 番目のパターン・マッチング (1.2N) は、1 桁または 2 桁の数字を受け取ります。前述のように、繰り返し回数の範囲を定義するため、オプションの小数点 (.) を使用します。

## 組み合わせパターンの指定

以下の形式を使用して、組み合わせパターンを定義します。

Pattern1Pattern2

パターンの組み合わせで、pattern1 に pattern2 が続く配列との比較で、目的のオペランドをチェックします。例えば、以下の式を考えてみます。

## ObjectScript

```
SET match = value?3N.4L
```

この式は、3 桁の数字の後ろに、0 から 4 つの小文字の英字が続いているかどうかのパターンをチェックします。目的のオペランドにパターンの組み合わせと同じものが 1 つ含まれている場合にのみ True (1) を返します。例えば、文字列 "345g" と "345gfij" は一致しますが、"345gfijhkbc" と "345gfij276hkbc" は一致しません。



## 不確定パターンの指定

以下の形式を使用して、不確定パターンを指定します。

```
.pattern
```

不確定パターンを使用して、目的のオペランドは、pattern の繰り返しをチェックします。繰り返しの数は（ゼロ回を含め）指定されません。例えば、以下の式を考えてみます。

### ObjectScript

```
SET match = value?.N
```

この式は、目的のオペランドがゼロもしくは 1 つ以上の数字を含み、その他のタイプの文字を含まない場合、True (1) を返します。

## 交互パターンの指定 (論理 OR)

交互パターンを使用すると、指定した複数のパターン列のいずれかとオペランドが一致するかどうかをテストできます。これにより、パターン・マッチングで論理 OR 機能が得られます。

交互パターンの構文は、以下のとおりです。

```
( pattern-element sequence {, pattern-element sequence }... )
```

したがって、以下のパターンは、val に文字 A が 1 回出現する、あるいは文字 B が 1 回出現する場合に True (1) を返します。

### ObjectScript

```
SET match = value?1(1"A",1"B")
```

交互パターンは、以下のパターン・マッチング式のように入れ子にできます。

### ObjectScript

```
SET match = value?.(.(1A,1N),1P)
```

例えば、電話番号の妥当性を検証する場合、少なくとも、電話番号は、3 桁目と 4 桁目の間にハイフン (-) のある 7 桁の番号である必要があります。以下に例を示します。

```
nnn-nnnn
```

また、電話番号に 3 桁の市外局番がある場合、括弧で囲むか、ハイフンで残りの番号から区別する必要があります。以下に例を示します。

```
(nnn) nnn-nnnn  
nnn-nnn-nnnn
```

以下のパターン・マッチング式は、電話番号の 3 つの有効な形式を示します。

### ObjectScript

```
SET match = phone?3N1"-"4N  
SET match = phone?3N1"-"3N1"-"4N  
SET match = phone?1("3N1") "3N1"-"4N
```

交互パターンを使用しない場合には、次の複合ブーリアン式が、あらゆる形式の電話番号の妥当性を確認するために必要となります。

## ObjectScript

```
SET match =
(
(phone?3N1"- "4N) ||
(phone?3N1"- "3N1"- "4N) ||
(phone?1"( "3N1" ) "3N1"- "4N)
)
```

交互パターンを使用する場合、電話番号の妥当性を検証するために、以下の単一のパターンが必要です。

## ObjectScript

```
SET match = phone?.1(1"( "3N1" ) " ,3N1"- ")3N1"- "4N
```

この例の交互パターンでは、電話番号の市外局番部分を、1“(“3N1”)” または 3N1”-” のいずれかで表現することができます。0 から 1 の交互カウント範囲は、オペランド phone が、0 または 1 のエリア・コードを持てることを示します。

1 より大きい反復カウントを持つ交互パターンは、使用できるパターンの組み合わせを多く生成できます。以下の交互パターンは、例で示されている文字列と一致する以外に、26 とおりの 3 文字の文字列と一致します。

## ObjectScript

```
SET match = "CAT"?3(1"C",1"A",1"T")
```

## 重要

交互パターンでは、不確定パターンの中で不確定パターンを入れ子にできます。例えば、.(A,.N) は、入れ子にした不確定パターンを含む交互パターンです。このような構造は、適正な長さの文字列に対する演算であっても、実行時間が極端に長くなる可能性があることに注意してください。Ctrl-C を押すことで、実行を停止できます。アプリケーションで不確定パターンを入れ子にする必要があるものの、実行時間がかかりすぎる場合は、[\\$MATCH](#) を使用して正規表現 (Regex) マッチングを試してみます。その方が効率的になることがあります。

## 不完全パターンの使用法

パターン・マッチングにより、文字列の一部しか一致しない場合、False (0) の結果を返します。つまり、パターンがすべて比較された後は、文字列が残っていないことを示します。以下の式は、パターンの最後の R が一致しないため、False (0) を返します。

## ObjectScript

```
SET match = "RAW BAR"?..U1P2U
```

## パターンの複数解釈

オペランドを比較する際に、1 つのパターンに複数の解釈が可能な場合があります。例えば、以下の式は 2 とおりの解釈ができます。

## ObjectScript

```
SET match = "/////A#####B$$$$$"?..E1U.E
```

1. 最初の .E が部分文字列の ///// と一致し、1U は A と一致し、2番目の .E は部分文字列の #####B\$\$\$\$\$ と一致します。
  2. 最初の .E が部分文字列は /////A##### と、1U は B と一致し、2番目の .E は部分文字列の .E と一致します。
- 少なくとも式の 1 つの解釈が True (1) である場合、式の値は True となります。

## 非マッチ演算

非マッチ演算は、パターン・マッチング演算子と共に否定演算子 ( ' ) を使用して記述できます。

```
operand'?pattern
```

非マッチ演算は、パターン・マッチングの真偽値を反転します。オペランドの文字がパターンと一致しない場合、非マッチ演算は True (1) を返します。パターンがオペランドの文字のすべてに一致する場合、非マッチ演算は False (0) を返します。

以下の例では、非マッチ演算が使用されています。

### ObjectScript

```
WRITE !,"abc" ?3L
WRITE !,"abc" '?3L
WRITE !,"abc" ?3N
WRITE !,"abc" '?3N
WRITE !,"abc" '?3E
```

## パターンの複雑さ

一部の複雑なパターンでは、適正な長さの文字列に対する演算であっても、実行時間が極端に長くなる場合があります。特に交互パターンと不確定パターンの組み合わせは、実行時に問題を引き起こす可能性があります。パターン・マッチング文を使用したプログラムの実行に、予想より大幅に時間がかかっている場合、**Ctrl-C** を使用して <INTERRUPT> エラーで実行を中止し、使用しているパターンを簡潔にします。パターンを簡潔にできない場合は、**\$MATCH** による Regex マッチングを試します。パターン・マッチングよりも効率的になることがあります。

複数の変更と不明確なパターンによるパターン・マッチは、長い文字列に適用される場合、多くのレベルをシステム・スタックに繰り返す場合があります。きわめてまれに、この繰り返しが数千回程度にまで増加し、スタックのオーバーフローおよびプロセスのクラッシュが発生することがあります。このような極端な状況が発生した場合、InterSystems IRIS は、現在のプロセスのクラッシュの危険を冒すよりはむしろ、<COMPLEX PATTERN> エラーを発行します。このような場合は、パターンを簡潔にするか、元の文字列のより短い部分にパターンを適用することをお勧めします。

## 正規表現に関する注記

ObjectScript では、多くのソフトウェア・ベンダよりサポートされる (バリエーションを用いた) パターン・マッチ構文である **正規表現** をサポートします。正規表現は、**\$LOCATE** および **\$MATCH** 関数と **%Regex.Matcher** クラスのメソッドで使用できます。

これらのパターン・マッチング・システムは完全に別個のものです。それぞれのパターン・マッチング・システムはその独自のコンテキストでのみ使用できます。ただし、以下の例で示すように、論理 AND および OR 構文を使用して、異なるパターン・マッチング・システムからパターン・マッチング・テストを組み合わせることは可能です。

### ObjectScript

```
SET var = "abcDef"
IF (var ?.e2U.e) && $MATCH(var, "^.{3,7}") { WRITE "It's a match!" }
ELSE { WRITE "No match" }
```

ObjectScript のパターンは、文字列が 2 連続の大文字を含んでいるかどうかを判断します。正規表現のパターンは、文字列が 3 ~ 7 文字かどうかを判断します。

## 間接 (@)

名前、パターン、コマンド引数、配列ノード、または \$TEXT 引数の構文を、間接オペランドの値を通じて間接的に含めることができます。

### 間接演算の概要

ObjectScript の間接演算子 (@) により、コマンド行、コマンド、またはコマンド引数の一部またはすべてを実行時に動的に置き換えることができます。InterSystems IRIS® データ・プラットフォームは、関連するコマンドを実行する前に、置換を実行します。

間接演算は、[添え字間接演算](#)の場合を除いて、間接演算子 (@) で指定され、以下の形式になります。

@variable

variable は、置換する値の取得元の変数です。変数は配列ノードにできます。間接演算子を使用するための手順は次の 2 つです。

1. 値を variable に割り当てます。適切な値は、意図したコンテキストによって異なります。  
置換する値で参照されるすべての変数は、プロシージャで使用されているとしてもパブリック変数であることに注意してください。
2. 該当するコンテキストに (つまり、コマンド行、コマンド、またはコマンド引数の一部またはすべてとして)、@variable を含めます。コンテキストに基づいて、間接演算子には 5 つのシナリオがあり、これらについて、このページで説明します。

以下は、ターミナル・セッションで示された簡単な例です。

#### Terminal

```
USER>set y="B"
USER>set @y = 123
USER>write B
123
```

### 制限と代替方法

間接演算は、明らかに便利な場合にのみ使用します。間接演算は、InterSystems IRIS がコンパイル・フェーズの間ではなく、実行時に必要な評価を実行するため、性能に影響を与える可能性があります。また、複雑な間接演算を使用する場合、明確なコードを記述する必要があります。間接演算は、コード解析を複雑にする場合があるからです。

間接演算は、他の方法より能率的かつ汎用的にコーディングできますが、必ずしも使用する必要はありません。XECUTE コマンドなど他の方法を使用して、間接演算と同様の機能を常に行うことができます。

間接演算は、ドット構文には使用できません。ドット構文が実行時ではなく、コンパイル時に構文解析されるためです。

**重要** オブジェクト・プロパティの値の取得または設定に間接演算子を使用しないでください。これを使用した場合、プロパティのアクセサ・メソッド (<PropertyName>Get および <PropertyName>Set) がバイパスされるため、エラーが生じることがあります。間接的にオブジェクト・プロパティの値を取得または設定する必要がある場合は、これを目的とした \$CLASSMETHOD、\$METHOD、および \$PROPERTY 関数を使用します。["オブジェクトへの動的アクセス"](#) を参照してください。

### シナリオ：名前間接演算

名前間接演算では、間接演算は、変数名、行ラベル、ルーチン名として評価されます。InterSystems IRIS は、変数の内容をコマンド実行前に要求された名前に置換します。

名前間接演算は、パブリック変数にのみアクセスできます。詳細は、“[ユーザ定義コード](#)”を参照してください。

名前付き変数を参照するために間接演算を使用する場合、間接演算の値は、必要な添え字すべてを含め、完全なグローバル変数名またはローカル変数名にする必要があります。

行ラベルを参照するために間接演算を使用する場合、間接演算の値は、構文的に有効な行ラベルにする必要があります。以下の例では、InterSystems IRIS は D を以下のように設定します。

- ・ N が 1 の場合、行ラベルの値は FIG
- ・ N が 2 の場合、行ラベルの値は GO
- ・ その他の場合の値は STOP

その後、InterSystems IRIS は D に値が与えられたラベルに制御を渡します。

### ObjectScript

```
B SET D = $SELECT(N = 1:"FIG",N = 2:"GO",1:"STOP")
; ...
LV GOTO @D
```

ルーチン名を参照するために間接演算を使用する場合、間接演算の値は、構文的に有効なルーチン名である必要があります。以下の例では、名前間接演算を DO コマンドで使用し、適切なプロシージャ名を渡します。実行時、変数 loc の内容が、要求されている名前に置換されます。

### ObjectScript

```
Start
READ !,"Enter choice (1, 2, or 3): ",num
SET loc = "Choice"_num
DO @loc
RETURN
Choice1()
; ...
Choice2()
; ...
Choice3()
; ...
```

名前間接演算は、名前の値のみを置換できます。以下の例で、2 番目の SET コマンドは、コンテキストに起因するエラー・メッセージを返します。等号の右側の式を評価する際、InterSystems IRIS は @var1 を数値ではなく、変数名に対する間接参照と解釈します。

### ObjectScript

```
SET var1 = "5"
SET x = @var1*6
```

正確に実行するには、上記の例を以下のように修正します。

### ObjectScript

```
SET var1 = "var2",var2 = 5
SET x = @var1*6
```

## シナリオ：パターン間接演算

パターン間接演算では、間接演算子はパターン・マッチを置換します。間接演算の値は、有効なパターンの必要があります。“[パターン・マッチング \(?\)](#)”を参照してください。)パターン間接演算は、可能性のあるパターンを複数選択し、それらを単一のパターンとして使用する場合に特に役立ちます。

以下の例では、間接演算をパターン・マッチングに使用し、郵便番号 (ZIP) が有効かどうかを判断します。コードは、5 桁 (nnnnn) あるいは 9 桁 (nnnnnnnnn) のいずれかです。

最初の SET コマンドは、5 桁の形式に対するパターンを設定します。次の SET コマンドは、9 桁の形式に対するパターンを設定します。この 2 番目の SET コマンドは、後置条件式 (`$LENGTH(zip) = 10`) が True (0 以外) の場合に実行されます。この条件は、ユーザが 9 桁を入力した場合にのみ発生します。

## ObjectScript

```
GetZip()
SET pat = "5N"
READ !,"Enter your ZIP code (5 or 9 digits): ",zip
SET:($LENGTH(zip)=10) pat = "5N1"-"-"4N"
IF zip'?@pat {
    WRITE !,"Invalid ZIP code"
    DO GetZip()
}
```

パターン・マッチングに間接演算を使用すると、アプリケーション中で使用するパターンをローカライズするのに便利です。この場合、パターンをそれぞれ別々の変数に格納しておき、実際のパターン・テストでは間接演算でそれらを参照できます(これは、名前間接演算の例でもあります)。このようなアプリケーションを移植するには、パターン変数自体を変更するだけで済みます。

## シナリオ：引数間接演算

引数間接演算では、間接演算の結果は、1 つ以上のコマンド引数として評価されます。一方、名前間接演算は、引数の一部のみを適用します。

この違いを理解するには、以下の例と“[名前間接演算](#)”の例を比較してください。

## ObjectScript

```
Start
SET rout = "^Test1"
READ !,"Enter choice (1, 2, or 3): ",num
SET loc = "Choice"_num_rout
DO @loc
QUIT
```

この場合の引数間接演算の例は、`@loc` が引数を完全な形式 (すなわち、`label^routine`) で提供します。名前間接演算の例では、`@loc` は引数の一部分のみ提供するため、`@loc` が名前間接演算の例となります (エントリ・ポイントが、別のルーチンではなく、現在のルーチンに存在する label 名)。

以下の 2 番目の SET コマンドは、名前間接演算 (変数名である引数の一部のみに適用) の例であり、一方、3 番目の SET コマンドは、引数間接演算 (引数全体に適用) の例です。

## ObjectScript

```
SET a = "var1",b = "var2 = 3*4"
SET @a = 5*6
SET @b
WRITE "a = ",a,!
WRITE "b = ",b,!
```

## シナリオ：添え字間接演算

添え字間接演算は、[名前間接演算](#)の拡張形式です。添え字間接演算では、間接演算の値はローカルあるいはグローバルの配列ノード名である必要があります。また、他の間接演算とは構文的に異なります。添え字間接演算は、以下の 2 つの間接演算子を使用します。

```
@array@(subscript)
```

`^client` というグローバル配列で、最初のレベルのノードにはクライアント名、2 番目にはクライアントの番地、3 番目にはクライアントの市区町村、都道府県、郵便番号が含まれていると想定します。この配列の最初のレコードにあるこれら 3 つのノードを出力するには、以下の形式で WRITE コマンドを使用します。

## ObjectScript

```
WRITE !,^client(1),!,^client(1,1),!,^client(1,1,1)
```

実行すると、このコマンドは以下のように出力する場合があります。

```
John Jones
42 Arnold St.
Boston, MA 02745
```

レコードの範囲を出力するには (例えば、先頭から 10 件のレコード)、WRITE コマンドが FOR ループの中で実行されるようにコードを変更します。以下に例を示します。

## ObjectScript

```
FOR i = 1:1:10 {
  WRITE !,^client(i),!,^client(i,1),!,^client(i,1,1)
}
```

FOR ループを実行するたびに変数 *i* は 1 ずつ増加し、その変数を使用して、出力する次のレコードを選択します。

この例は、前述の例よりも一般的ですが、配列名と出力レコード数の両方を明示的に指定する点で特殊です。

以下のように添え字間接演算を使用して、このコードをさらに一般的な形に変更し、ユーザが、3 つのノード・レベルに名前、町名、および都市情報を格納している任意の配列 (グローバルまたはローカル) から、一定範囲のレコードをリストできるようにします。

## ObjectScript

```
Start
  READ !,"Output Name, Street, and City info.",!
  READ !,"Name of array to access: ",name
  READ !,"Global or local (G or L): ",gl
  READ !,"Start with record number: ",start
  READ !,"End with record number: ",end
  IF (gl["L"]!(gl["l"])) {SET array = name}
  ELSEIF (gl["G"]!(gl["g"])) {SET array = "^"_name}
  SET x = 1,y = 1
  FOR i = start:1:end {DO Output}
  RETURN
Output()
  WRITE !,@array@(i)
  WRITE !,@array@(i,x)
  WRITE !,@array@(i,x,y)
  QUIT
```

Output サブルーチンにある WRITE コマンドは、添え字間接演算を使用して、要求されている配列とレコード範囲を参照します。

添え字間接演算の評価で、間接演算のインスタンスが、添え字のないグローバルまたはローカルの変数を参照する場合、間接演算の値は、変数名、および括弧を含む 2 番目の間接演算子の右側にあるすべての文字になります。

ローカル変数の添え字レベルの最大数は 255 です。添え字間接演算は、多次元オブジェクト・プロパティに 254 個を超える添え字を参照できません。グローバル変数の添え字レベルの最大数は添え字に応じて異なり、レベルが 255 を超える場合もあります。これは、“[グローバルについての正式な規則](#)” で説明しています。間接演算を使用して 255 より大きな添え字レベルをローカル変数に移入しようとすると、<SYNTAX> エラーが返されます。

クラス・パラメータは、ローカル変数またはグローバル変数をベースとして使用するのと同様の方法で、添え字間接演算のベースとして使用できます。例えば、以下の構文でクラス・パラメータを使用して、添え字間接演算を実行できます。

## ObjectScript

```
SET @..#myparam@(x,y) = "stringval"
```



## シナリオ：\$TEXT 引数間接演算

名前からわかるように、\$TEXT 引数間接演算は、[\\$TEXT](#) 関数の引数のコンテキストでのみ使用可能です。この間接演算の値は、有効な \$TEXT 引数とする必要があります。

\$TEXT 引数間接演算を使用する主な目的は、同じ結果を返す間接演算を複数の形式で記述しなくても済むようにすることにあります。例えば、ローカル変数 LINE に "START^MENU" のエントリ参照が含まれている場合、行ラベルとルーチン名に対し名前間接演算を使用して、行テキストを取得できます。以下がその例です。

### ObjectScript

```
SET LINETEXT = $TEXT(@$PIECE(LINE,"^",1)^@$PIECE(LINE,"^",2))
```

以下のように \$TEXT 引数間接演算を使用すると、より単純な方法で同じ結果を返すことができます。

### ObjectScript

```
SET LINETEXT = $TEXT(@LINE)
```



# ObjectScript コマンド

このドキュメントでは、ObjectScript がサポートしているコマンドについて詳しく説明します。このドキュメントでは、ObjectScript コマンドが以下の 2 つのグループに分類されています。

- ・ 汎用的なコマンド
- ・ ルーチンおよびデバッグ・コマンド

各グループ内のコマンドの説明はアルファベット順になっています。

ObjectScript コマンドの全般的な詳細は、“[コマンド](#)” を参照してください。

ほとんどのコマンドには省略形があります。通常は、コマンド名の最初の文字が省略形となりますが、文字 Z で始まるコマンドの場合はコマンド名の最初の 2 文字が省略形となります。各コマンドの説明では、最初に完全名を使用した構文が示され、その下に略名を使用した構文が示されます (略名がある場合)。

各コマンドの説明には、実際に構文に使用される句読点文字のみが含まれます。構文中のオプション要素を明確にするなど、形式規約のための句読点は使用されていません。その種の情報は、説明の直後にある引数のテーブルで説明されます。

1 つの例外は省略記号 (...) です。コンマの後の省略記号は、コンマの前の引数 (または引数のグループ) が、コンマ区切りリストとして複数回繰り返し可能であることを示します。中括弧の中の省略記号 { ... } は、1 つ以上のコマンドを含むコード・ブロックを中括弧内に記述できることを示します。中括弧はコード内で指定する必要があるリテラル文字です。

ほとんどのコマンドは、1 つ以上の引数を取ります。引数とは、コマンドのアクションを定義または制御する式 (例えば、関数とその引数、変数、演算子とそのオペランド、オブジェクト・プロパティ、オブジェクト・メソッドなど) です。コマンドで複数の引数を使用される場合は、一般的に引数リストとして参照されます。一部のコマンドでは、その引数自体が引数パラメータを取る場合があります。例えば、DO コマンドの各引数はパラメータ・リストを取ることができます。これらの情報は構文に示されます。

一部のコマンドは引数なしです。つまり、引数をまったく取らずに実行できます。また、どのような状況でも引数を取らないコマンドもあれば、特定の状況でのみ引数を取るコマンドもあります。後者のコマンドでは、引数のない場合と、引数リストがある場合で、その意味が異なります。

ほとんどのコマンドは、コマンド実行の可否を決定する条件を指定する、オプションの[後置条件式](#)を取ることができます。後置条件式は、コマンド名の後にコロンの (:) を使用して追加します。コマンド名とその後置条件式の間に、スペースや改行を挿入することはできません。厳密に言うと、後置条件式はコマンド引数ではありませんが、ここでは引数と一緒に説明しています。引数なしコマンドも後置条件式を取ることができます。

ほとんどの ObjectScript コマンドは、すべてのハードウェア・プラットフォーム上で同様に機能します。プラットフォーム固有のコマンド機能には、その機能を使用できるプラットフォームの種類が明示されています (Windows、UNIX® など)。プラットフォームの制限が記載されていないコマンドは、すべてのプラットフォームでサポートされます。

## BREAK (ObjectScript)

ブレークポイントで実行を中断します。ユーザの中断を有効または無効にします。

### 構文

```
BREAK:pc extend
B:pc extend
```

```
BREAK:pc flag
B:pc flag
```

### 引数

引数	説明
pc	オプション - 後置条件式
extend	オプション - 引用符付きの文字列として指定された、有効化または無効化されるブレークポイントの種類を示す文字コード。有効な値は、 <a href="#">BREAK の拡張引数の一覧</a> に示されています。flag 引数と一緒に使用することはできません。
flag	オプション - 中断動作を指定する整数フラグ。flag 値は、引用符で囲んでも囲まなくてもかまいません。有効な値は、0 と 4 (CTRL-C の中断を無効にします)、および 1 と 5 (CTRL-C の中断を有効にします) です。既定値は、コンテキストにより決定されます (詳細は、 <a href="#">BREAK flag</a> を参照)。extend 引数と一緒に使用することはできません。

### 説明

BREAK コマンドには、以下の 3 つの構文形式があります。

- ・ [引数なしの BREAK](#) は、現在の場所でコードの実行を中断します。
- ・ [BREAK extend](#) は、一定のブレークポイント間隔でコードの実行を中断します。
- ・ [BREAK flag](#) は、CTRL-C の中断を有効または無効にします。

### 必要な権限

コードの実行時に BREAK 文を使用するには、ユーザに、%Development リソースを U (使用) 権限で利用できるロール (%Developer や %Manager など) が割り当てられている必要があります。ロールをユーザに割り当てるには、SQL の [GRANT](#) 文を使用するか、管理ポータルの [\[システム管理\]→\[セキュリティ\]→\[ユーザ\]](#) オプションを使用します。定義を編集するユーザ名を選択し、[\[ロール\]](#) タブを選択して、そのユーザにロールを割り当てます。

### 主デバイス

主デバイスの入出力ダイレクトを使用するアプリケーションをデバッグする場合、デバッグ・プロンプトではリダイレクトがオフになり、デバッグ・コマンドからの出力はターミナルに表示されます。

### 引数

#### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に BREAK コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、["コマンド後置条件式"](#) を参照してください。

## extend

BREAK extend は、ブレークポイントの動作を指定する文字列コードをサポートします。引用符は必須です。これらの extend コードのテーブルは、“[BREAK の拡張引数](#)”を参照してください。

## flag

BREAK flag は、CTRL-C の中断を処理する 4 つの異なる方法をサポートします。

- ・ **BREAK 0** : まだシグナル送信されていない保留中の CTRL-C トラップをすべて破棄します。以降の CTRL-C のシグナル送信を無効にします。
- ・ **BREAK 1** : 保留中の CTRL-C トラップをすべて破棄します。以降の CTRL-C のシグナル送信を有効にします。  
つまり、BREAK 1 の実行後に CTRL-C を入力すると、CTRL-C がシグナル送信されます。
- ・ **BREAK 4** : 保留中の CTRL-C トラップを一切破棄しません。以降の CTRL-C のシグナル送信を無効にします。  
つまり、以降の BREAK 1 または BREAK 5 コマンドにより CTRL-C が有効になったときに保留中の CTRL-C トラップがシグナル送信されます。
- ・ **BREAK 5** : 保留中の CTRL-C トラップを一切破棄しません。以降の CTRL-C のシグナル送信を有効にします。  
つまり、保留中の CTRL-C トラップが、BREAK 5 直後に ObjectScript コマンドでシグナル送信されます。すべてではありませんが、ほとんどの ObjectScript コマンドは、CTRL-C に対してポーリングを実施します。

詳細および例は、“[中断を有効または無効にする BREAK フラグ](#)”を参照してください。

## 引数なしの BREAK

引数なしの BREAK は、遭遇時にコードの実行を中断します。引数なしの BREAK は、プログラムのソース・コードで使用できます。後置条件式は付けても付けなくてもかまいません。このコマンドは、その時点でプログラムの実行を中断し、制御をターミナル・プロンプトに戻します。引数なしの BREAK は、デバッグ目的で使用されます。

ルーチン内に引数なしの BREAK を含めると、ブレークポイントが設定されます。ブレークポイントはルーチンの実行を中断し、プロセスをターミナル・プロンプトに戻します。コードにブレークポイントを埋め込むことによって、デバッグを行う特定のコンテキストを設定できます。実行中に BREAK に遭遇すると、InterSystems IRIS はルーチンを中断し、ターミナルに戻ります。そして、他の ObjectScript コマンドを使用して、デバッグを実行できます。例えば、WRITE コマンドを使用して、中断中の箇所の変数値を調べ、SET コマンドを使用して、これらの変数やそれ以外の変数に新しい値を指定することができます。ルーチン行エディタ (XECUTE %) を呼び出して、ルーチンを修正することもできます。BREAK を使用してルーチンの実行を中断した後、引数なしの GOTO を使用して、通常の実行を再開することができます。または、GOTO コマンドの引数としてこの場所を指定して、他の場所から実行を再開することもできます。

**注釈** コード内では BREAK コマンドではなく、ZBREAK コマンドを使用して ObjectScript デバッグを呼び出すことを推奨します。デバッグには、より多彩なデバッグ機能があります。

%SYSTEM.Process クラスの BreakMode() メソッドを使用すると、現在のプロセスに対する引数なしの BREAK の動作を構成できます。Config.Miscellaneous クラスで BreakMode プロパティを設定することで、引数なしの BREAK のシステム全体の動作を構成できます。

すべての引数なしのコマンドと同様に、引数なしの BREAK と、同じ行にあるその直後のコマンドとの間に少なくとも 2 つの空白を挿入する必要があります。

## 引数なしの BREAK を条件付きで使用する

コード内の引数なしの BREAK コマンドに条件を指定すると、ルーチンを変更しなくても、単に変数を設定するだけで同じコードを再実行できるので便利です。例えば、ルーチンに以下の行があるとします。

## ObjectScript

BREAK : \$DATA ( debug )

変数 debug を設定すると、ルーチンを中断し、ジョブをターミナル・プロンプトに戻すことができます。また、変数 debug をクリアすると、ルーチンの実行を継続できます。

## 標準のブレークポイントを設定する BREAK の拡張引数

中断したいルーチンのすべての位置に、引数なし BREAK コマンドを置く必要はありません。BREAK には“拡張された”一連の引数 (extend) があり、この引数を使用して、引数なしの BREAK コマンドがコード全体に散在しているかのように、定期的にルーチンを中断させることができます。以下のテーブルは、さまざまな BREAK コマンド拡張引数です。

引数	説明
"S"	BREAK "S" (シングル・ステップ) を使用すると、コードのコマンド (生成されたトークン) を一度に 1 つずつ実行します。すべての ObjectScript コマンドがトークンを生成するわけではありません。一部のコマンドは複数のトークンを生成するため、複数のステップとして解析されます (下記を参照してください)。InterSystems IRIS は、DO コマンドや XECUTE コマンドによって呼び出されたコマンド、または FOR ループやユーザ定義関数で中断を止め、コマンドやループの完了時に次のトークンで再開します。
"S+"	BREAK "S+" は BREAK "S" と同様に機能します。ただし、DO コマンドや XECUTE コマンドによって呼び出されたコマンド、または FOR ループやユーザ定義関数での中断を InterSystems IRIS が含める点が異なります。
"S-"	BREAK "S-" を使用すると、現在のレベルでのブレーク・ステップ実行 ("S" または "L") が無効になり、前のレベルでのシングル・ステップ実行が有効になります。現在のレベルでは BREAK "C" と同じように動作し、前のレベルでは BREAK "S" と同じように動作します。
"L"	BREAK "L" (行ステップ) を使用して、各行の最初で処理を中断しながら、コードを単一ルーチン行ごとに実行します。トークンを生成しない行は無視されます (下記参照)。InterSystems IRIS は、DO コマンドや XECUTE コマンドによって呼び出されたコマンド、または FOR ループやユーザ定義関数で中断を止め、コマンドやループの完了時に次の行で再開します。
"L+"	BREAK "L+" は BREAK "L" と同様に機能します。ただし、DO コマンドや XECUTE コマンドによって呼び出されたコマンド、または FOR ループやユーザ定義関数で各ルーチン行の最初において InterSystems IRIS が中断を継続する点が異なります。
"L-"	BREAK "L-" を使用すると、現在のレベルでのブレーク・ステップ実行 ("S" または "L") が無効になり、前のレベルでの行ステップ実行が有効になります。現在のレベルでは BREAK "C" と同じように動作し、前のレベルでは BREAK "L" と同じように動作します。
"C"	BREAK "C" (クリア・ブレーク) を使用すると、現在のレベルでのブレーク・ステップ実行 ("L" および "S") がすべて停止します。BREAK が前のルーチン・レベルで有効になっている場合は、そのジョブが QUIT を実行した後、前のレベルでブレークが再開します。
"C-"	BREAK "C-" を使用すると、現在のレベルおよび前のすべてのレベルでのブレーク・ステップ実行 ("L" および "S") がすべて停止します。これにより、他のデバッグ機能に影響を与えることなく、すべてのレベルでステップ実行を削除できます。
"OFF"	BREAK "OFF" は、プロセスに対して構築されたすべてのデバッグを削除します。これは、すべてのブレークポイントとウォッチポイントを削除し、すべてのプログラム・スタック・レベルでステップ実行を無効にします。また、デバッグとトレース・デバイスに関連するものも削除しますが、それらを閉じません。

BREAK "S" および BREAK "L" は、トークンを生成する文で中断します。すべての ObjectScript コマンドおよび行がトークンを生成するわけではありません。例えば、BREAK "S" および BREAK "L" は、どちらもラベル行、コメント、およ

び TRY 文を無視します。BREAK "S" は CATCH 文でブレークしますが (CATCH ブロックが入力された場合)、BREAK "L" はブレークしません。

BREAK "S" と BREAK "L" 間で 1 つ異なる点は、多くのコマンド行が複数のトークンを生成するため、複数のステップで構成されることです。必ずしもこれが明白であるとは限りません。例えば、以下はすべて 1 つの行 (および 1 つの ObjectScript コマンド) で記述されていますが、BREAK "S" は、それぞれを 2 つのステップとして解析します。

### ObjectScript

```
KILL x,y
SET x=1,y=2
IF x=1,y=2 {
    WRITE "hello",! }
```

BREAK "S" は [コマンドの後置条件式](#) を解析します。後置条件式が True (コマンドを実行する) の場合、カーソルは後置条件式の後をポイントするように再配置されます。

ブレークポイント後のコード実行を再開するには、ターミナル・プロンプトで [GOTO](#) コマンド (短縮形は G) を発行します。詳細は、["コマンド行ルーチンのデバッグ"](#) を参照してください。

BREAK "OFF" コマンドの使用法は、以下の一連のコマンドの使用法と類似しています。

### ObjectScript

```
ZBREAK /CLEAR
ZBREAK /TRACE:OFF
ZBREAK /DEBUG:" "
ZBREAK /ERRORTRAP:ON
BREAK "C-"
```

## 中断を有効または無効にする BREAK フラグ

BREAK flag を使用して、**CTRL-C** などのユーザの中断を有効にするか無効にするかを制御できます。これらの無効/有効オプション間の実質的な相違点は以下のとおりです。

BREAK 0 および BREAK 1 は、重要な一連のコマンドが **CTRL-C** シグナルによって中断されないコード・ブロックを作成するために使用できます。ただし、そのようなブロックでのループは、インタラクティブ・ユーザが **CTRL-C** を使用して中断するのが難しくなる場合があります。これは、**CTRL-C** トラップの検出と **CTRL-C** シグナルのポーリングとの間にわずかな遅延が存在するためです。この遅延により、次の BREAK コマンド・ループで **CTRL-C** のユーザの中断を破棄できる場合があります。

BREAK 4 および BREAK 5 を含むプログラム・ブロックは、インタラクティブ・ユーザが **CTRL-C** を使用してこのブロックでのループ処理を中断できる機能に影響を与えることなく、重要な一連のコマンドが **CTRL-C** シグナルによって中断されないコードを作成するために使用できます。

BREAK の既定の flag 動作は、以下のようにログイン・モードに依存します。

- ・ ターミナル・プロンプトでログインする場合、既定は BREAK 1 となります。**CTRL-C** などの割り込みは、常に有効化されています。OPEN コマンドまたは USE コマンドに指定された B (/break) プロトコルには何も効果がありません。
- ・ ルーチンからコードを実行する場合、既定は BREAK 0 となります。**CTRL-C** などの割り込みは、OPEN または USE コマンドで指定される B (/break) プロトコルによって有効または無効にされています。

OPEN および USE モード・プロトコルの詳細は、["ターミナル入出力"](#) を参照してください。

## BREAK flag の例

以下の例では、[\\$ZJOB](#) を使用して、中断を有効にするか無効にするかを決定します。

## ObjectScript

```
BREAK 0
DO InterruptStatus
BREAK 1
DO InterruptStatus
WRITE "all done"
InterruptStatus()
IF $ZJOB\4#2=1 {WRITE "Interrupts enabled",!}
ELSE {WRITE "Interrupts disabled",!}
```

以下の例では、一連の数値のユーザ入力のために FOR ループで READ を使用します。これにより、READ 処理中のユーザの中断を無効にするように BREAK 0 が設定されます。ただし、ユーザが数値以外の値を入力する場合、BREAK 1 は、ユーザが入力値を拒否したり受け入れたりできるようにユーザの中断を有効にします。

## ObjectScript

```
SET y="^"
InputLoop
TRY {
  FOR {
    BREAK 0
    READ "input a number ",x
    IF x="" { WRITE !,"all done"  QUIT }
    ELSEIF 0=$ISVALIDNUM(x) {
      BREAK 1
      WRITE !,x," is not a number",!
      WRITE "you have four seconds to press CTRL-C",!
      WRITE "or accept this input value",!
      HANG 4 }
    ELSE { }
    SET y=y_x_^"
    WRITE !,"the number list is ",y,!
  }
}
CATCH { WRITE "Rejecting bad input",!
        DO InputLoop
      }
```

## 関連項目

- ・ [ZBREAK コマンド](#)
- ・ [GOTO コマンド](#)
- ・ [OPEN コマンド](#)
- ・ [USE コマンド](#)
- ・ [コマンド行ルーチンのデバッグ](#)
- ・ [ターミナル入出力](#)



## CATCH (ObjectScript)

例外発生時に実行するコード・ブロックを指定します。

### 構文

```
CATCH exceptionvar
{
    . . .
}
```

### 引数

引数	説明
exceptionvar	オプション — 例外変数。InterSystems IRIS オブジェクトへの参照 (OREF) を受け取るローカル変数として指定され、添え字がある場合とない場合があります。オプションで、この引数を括弧で囲むことができます。

### 説明

CATCH コマンドは、例外ハンドラを定義します。これは、TRY コード・ブロックで例外が発生したときに実行されるコードのブロックです。CATCH コマンドの後には、中括弧 ({} ) で囲まれたコード文のブロックが続きます。

TRY ブロックを指定する場合、CATCH ブロックが必須であり、TRY ブロックごとに対応する CATCH ブロックを含める必要があります。CATCH ブロックは 1 つのみが、各 TRY ブロックに対して許可されます。TRY ブロックのすぐ後に、CATCH ブロックが続く必要があります。TRY ブロックとその CATCH ブロックの間に実行可能コード行を含めることはできません。TRY ブロックとその CATCH ブロックの間、および CATCH コマンドと同じ行にラベルを含めることはできません。ただし、TRY ブロックとその CATCH ブロックの間にコメントを含めることはできます。

CATCH ブロックは、例外が発生したときに開始されます。例外が発生しない場合は、CATCH ブロックを実行しないでください。絶対に GOTO 文を使用して CATCH ブロックに入らないでください。

CATCH ブロックは、[QUIT](#) または [RETURN](#) で終了できます。QUIT は、現在のブロック構造を終了し、ブロック構造の外側にある次のコマンドで実行を続けます。例えば、入れ子の CATCH ブロック内で QUIT を発行すると、CATCH ブロックを終了して囲んでいるブロック構造に移動できます。引数付きの QUIT を使用して、CATCH ブロックを終了することはできません。これを行うと、コンパイル・エラーが返されます。CATCH ブロック内から完全にルーチンを終了するには、RETURN 文を発行します。

CATCH コマンドには、以下の 2 つの基本形式があります。

- ・ 引数なし
- ・ 引数付き

引数付きの形式が推奨されます。

### CATCH 例外処理

CATCH exceptionvar は、TRY ブロックからオブジェクト・インスタンス参照 (OREF) を受け取ります。これは、THROW コマンドによって明示的に渡されるか、システム・エラー発生時にシステムの実行環境から暗黙的に渡されます。OREF の詳細は、"[OREF の基本](#)" を参照してください。

このオブジェクト・インスタンス参照は、例外に関する情報を含むプロパティを提供します。

例外により、4 つの例外プロパティが CATCH に渡されます。これらのプロパティは、順番に、Name、Code、Location、および Data です。スローされた例外の場合、Location パラメータは渡されません。%IsA() インスタンス・メソッドを使用することで、これらのプロパティで渡された例外の種類を判別できます。

以下の例において、TRY ブロックは、**システム例外** (未定義のローカル変数) を生成したり、**SQL 例外** をスローしたり、**%Status 例外** をスローしたり、**一般例外** をスローしたりする場合があります。この汎用的な CATCH 例外ハンドラは、どのような例外が発生したかを判別して、適切なプロパティを表示します。システム例外に関する 4 つのプロパティをすべて表示します (Data プロパティは、一部の種類のシステム・エラーについて空の文字列になります)。SQL 例外に関する 2 つのプロパティ (Code および Data) を表示します。\$SYSTEM.Status.Error() に対して 2 つのプロパティを提供して、%Status 例外に関するエラー・メッセージの文字列を生成します。一般的な ObjectScript 例外に関する 3 つのプロパティ (Name、Code、および Data) を表示します。\$ZCVT 関数を使用して、ブラウザの表示用に角括弧を含む Name 値を形式設定します。

## ObjectScript

```
TRY {
    SET x=$RANDOM(4)
    IF x=0 { KILL undefvar
            WRITE undefvar }
    ELSEIF x=1 {
        SET oref=##class(%Exception.SQL).%New("-",999,"SQL error message")
        THROW oref }
    ELSEIF x=2 {
        SET oref=##class(%Exception.StatusException).%New("5002",,$LISTBUILD("My Status Error"))
        THROW oref }
    ELSE {
        SET oref=##class(%Exception.General).%New("<MY BAD>",999,"General error message")
        THROW oref }
    WRITE "this should not display",!
}
CATCH exp { WRITE "In the CATCH block",!
            IF 1=exp.%IsA("%Exception.SystemException") {
                WRITE "System exception",!
                WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
                WRITE "Location: ",exp.Location,!
                WRITE "Code: "
            }
            ELSEIF 1=exp.%IsA("%Exception.SQL") {
                WRITE "SQL exception",!
                WRITE "SQLCODE: "
            }
            ELSEIF 1=exp.%IsA("%Exception.StatusException") {
                WRITE "%Status exception",!
                DO $SYSTEM.Status.DisplayError(exp.AsStatus())
                RETURN
            }
            ELSEIF 1=exp.%IsA("%Exception.General") {
                WRITE "General ObjectScript exception",!
                WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
                WRITE "Code: "
            }
            ELSE { WRITE "Some other type of exception",! RETURN }
            WRITE exp.Code,!
            WRITE "Data: ",exp.Data,!
            RETURN
}
```

## 入れ子になった TRY/CATCH ブロック

CATCH ブロックは 1 つのみが、各 TRY ブロックに対して許可されます。ただし、ペアになった TRY/CATCH ブロックを入れ子にすることはできます。

以下のように、外側の CATCH ブロック内に TRY/CATCH ペアを挿入して入れ子にすることができます。

## ObjectScript

```
TRY {
    /* TRY code */
}
CATCH exvar1 {
    /* CATCH code */
    TRY {
        /* nested TRY code */
    }
    CATCH exvar2 {
        /* nested CATCH code */
    }
}
```



以下のように、外側の TRY ブロック内に TRY/CATCH ペアを挿入して入れ子にすることができます。

### ObjectScript

```
TRY {
    /* TRY code */
    TRY {
        /* nested TRY code */
    }
    CATCH exvar2 {
        /* nested CATCH code */
    }
}
CATCH exvar1 {
    /* CATCH code */
}
```

## 実行スタック

%Exception オブジェクトには、オブジェクト作成時に実行スタックが含まれます。この実行スタックには、StackAsArray() メソッドを使用してアクセスできます。以下の例では、この実行スタックを示しています。

### ObjectScript

```
TRY {
    WRITE "In the TRY block",!
    WRITE 7/0
}
CATCH exobj {
    WRITE "In the CATCH block",!
    WRITE $ZCVT($ZERROR,"O","HTML"),!
    TRY {
        WRITE "In the nested TRY block",!
        KILL fred
        WRITE fred
    }
    CATCH exobj2 {
        WRITE "In the nested CATCH block",!
        WRITE $ZCVT($ZERROR,"O","HTML"),!!
        WRITE "The Execution Stack",!
        DO exobj2.StackAsArray(.stk)
        ZWRITE stk
    }
}
```

## CATCH と \$ZTRAP、\$ETRAP

TRY ブロック内で \$ZTRAP または \$ETRAP を設定することはできません。ただし、CATCH ブロック内で \$ZTRAP または \$ETRAP を設定することはできます。また、TRY ブロックに入る前に \$ZTRAP または \$ETRAP を設定することもできます。

CATCH ブロック内で例外が発生した場合、指定された \$ZTRAP または \$ETRAP 例外ハンドラが実行されます。

## TRY / CATCH ループ

TRY ブロックにループ・バックする CATCH ブロックを TRY ブロックが呼び出すループは無限ではありません。最終的には <FRAMESTACK> エラーが発行されます。

## CATCH の無効化

ZBREAK /ERRORTRAP:OFF コマンドを実行すると CATCH 例外処理が無効になります。

## 引数

### exceptionvar

システム・エラー発生時に、THROW コマンドまたはシステムの実行環境から例外オブジェクト参照を受け取るのに使用されるローカル変数。システム・エラーの発生時に、exceptionvar は %Exception.SystemException 型のオブジェクトへ

の参照を受け取ります。ユーザ指定のエラーの発生時に、exceptionvar は `%Exception.General`、`%Exception.StatusException`、または `%Exception.SQL` 型のオブジェクトへの参照を受け取ります。詳細は、“インターシステムズ・クラス・リファレンス”の“`%Exception.AbstractException`”抽象クラスを参照してください。

オプションで、exceptionvar 引数を括弧で囲むことができます。したがって、`CATCH(var) { code block }` のようになります。この括弧構文は互換性の目的で提供されており、機能上の影響はありません。

## 例：システム例外

以下は、0 による除算実行時エラーによって呼び出される引数なしの CATCH の例を示しています。これは、\$ZERROR および \$ECODE のエラー値を表示します。引数なしの CATCH は、exceptionvar を渡すよりも信頼性が低いため、お勧めしません。CATCH ブロック内でエラーが発生した場合、\$ZERROR には、CATCH を呼び出したエラーではなく、この最新のエラーが含まれます。この例では、QUIT コマンドは CATCH ブロックを終了しますが、ブロック構造の外側にある次の行への“フォールスルー”は妨げません。

### ObjectScript

```
TRY {
    WRITE !,"TRY block about to divide by zero",!!
    SET a=7/0
    WRITE !,"this should not display"
}
CATCH {
    WRITE "CATCH block exception handler",!!
    WRITE "$ZERROR is: ",$ZERROR,!
    WRITE "$ECODE is :",$ECODE,!
    QUIT
    WRITE !,"this should not display"
}
WRITE !,"this is where the code falls through"
```

以下は、0 による除算実行時エラーによって呼び出され、引数を受け取る CATCH の例を示しています。このコーディング方法をお勧めします。myexp OREF 引数は、システムで生成された例外オブジェクトを受け取ります。これは、この例外インスタンスの Name、Code、および Location プロパティを表示します。この例では、RETURN コマンドはプログラムを終了するので、“フォールスルー”は発生しません。

### ObjectScript

```
TRY {
    WRITE !,"TRY block about to divide by zero",!!
    SET a=7/0
    WRITE !,"this should not display"
}
CATCH myexp {
    WRITE "CATCH block exception handler",!!
    WRITE "Name: ", $ZCVT(myexp.Name,"O","HTML"),!
    WRITE "Code: ",myexp.Code,!
    WRITE "Location: ",myexp.Location,!
    RETURN
}
WRITE !,"this is where the code falls through"
```

以下は、システム例外オブジェクトを受け取る CATCH の例を示しています。CATCH ブロック・コードは、`%Exception.SystemException` クラスの `AsSystemError()` メソッドを使用して、システム例外を \$ZERROR フォーマット済み文字列として表示します。(\$ZERROR も比較目的で表示されます。)その後、この CATCH ブロックは、エラー名、エラー・コード、エラー・データ、エラーの位置を個別のプロパティとして表示します。

## ObjectScript

```

TRY {
    WRITE !,"this global is not defined",!
    SET a=^badglobal(1)
    WRITE !,"this should not display"
}
CATCH myvar {
    WRITE !,"this is the exception handler",!
    WRITE "AsSystemError is: ",myvar.AsSystemError(),!
    WRITE "$ZERROR is: ",$ZERROR,!
    WRITE "Error name=", $ZCVT(myvar.Name, "O", "HTML"),!
    WRITE "Error code=", myvar.Code,!
    WRITE "Error data=", myvar.Data,!
    WRITE "Error location=", myvar.Location,!
    RETURN
}

```

## 例：スローされた例外

以下は、THROW コマンドによって呼び出される CATCH の例を示しています。myvar 引数は、4 つのプロパティを持つユーザ定義例外オブジェクトを受け取ります。この例では、THROW は省略された %Exception.General クラスの Location プロパティの値を提供しないことに注意してください。

## ObjectScript

```

TRY {
    SET total=1234
    WRITE !,"Throw an exception!"
    THROW ##class(%Exception.General).%New("Example Error",999,,"MyThrow")
    WRITE !,"this should not display"
}
CATCH myvar {
    WRITE !,"this is the exception handler"
    WRITE !,"Error data=",myvar.Data
    WRITE !,"Error code=",myvar.Code
    WRITE !,"Error name=",myvar.Name
    WRITE !,"Error location=",myvar.Location,!
    RETURN
}

```

以下の 2 つの例では、TRY ブロック内で誕生日を生成します。未来の誕生日を生成する場合、THROW を使用して一般例外を発行し、ユーザ定義例外を CATCH ブロックに渡します(例外をスローする日付を生成するためにこれらの例を複数回実行する必要がある場合があります)。

最初の例では、CATCH exceptionvar を指定しません。TRY ブロック内で定義された OREF 名を使用して、例外プロパティを指定します。

## ObjectScript

```

TRY {
    WRITE "In the TRY block",!
    SET badDOB=##class(%Exception.General).%New("BadDOB","999",,"Birth date is in the future")
    FOR x=1:1:20 { SET rndDOB = $RANDOM(7)$RANDOM(10000)
        IF rndDOB > $HOROLOG { THROW badDOB }
        ELSE { WRITE "Birthdate ", $ZDATE(rndDOB,1,,4)," is valid",! }
    }
}
CATCH {
    WRITE !,"In the CATCH block"
    WRITE !,"Birthdate ", $ZDATE(rndDOB,1,,4)," is invalid"
    WRITE !,"Error code=",badDOB.Code
    WRITE !,"Error name=",badDOB.Name
    WRITE !,"Error data=",badDOB.Data
    RETURN
}

```

2 番目の例では、CATCH exceptionvar を指定します。名前変更されたこの OREF を使用して、例外プロパティを指定します。これを使用することをお勧めします。

## ObjectScript

```

TRY {
    WRITE "In the TRY block",!
    SET badDOB=##class(%Exception.General).%New("BadDOB","999",,"Birth date is in the future")
    FOR x=1:1:20 { SET rndDOB = $RANDOM(7)_$RANDOM(10000)
        IF rndDOB > $HOROLOG { THROW badDOB }
        ELSE { WRITE "Birthdate ",$ZDATE(rndDOB,1,,4)," is valid",! }
    }
}
CATCH err {
    WRITE !,"In the CATCH block"
    WRITE !,"Birthdate ",$ZDATE(rndDOB,1,,4)," is invalid"
    WRITE !,"Error code=",err.Code
    WRITE !,"Error name=",err.Name
    WRITE !,"Error data=",err.Data
    RETURN
}

```

## 例：入れ子になった TRY/CATCH

以下は、0 による除算実行時エラーによって呼び出される CATCH の例を示しています。CATCH ブロックには、内側の CATCH ブロックとペアになる内側の TRY ブロックが含まれています。この内側の CATCH ブロックは、スローされた例外によって呼び出されます。デモンストレーションのために、この THROW は、このプログラム内ではランダムに呼び出されます。実際のプログラムでは、内側の CATCH ブロックは、AsSystemError() (捕捉エラー) と \$ZERROR (最新エラー) 間の不一致などの例外テストによって呼び出されます。

## ObjectScript

```

TRY {
    WRITE !,"Outer TRY block",!!
    SET a=7/0
    WRITE !,"this should not display"
}
CATCH myexp {
    WRITE "Outer CATCH block",!
    WRITE "Name: ", $ZCVT(myexp.Name, "O", "HTML"), !
    WRITE "Code: ", myexp.Code, !
    WRITE "Location: ", myexp.Location, !
    SET rndm=$RANDOM(2)
    IF rndm=1 {RETURN}
    TRY {
        WRITE !,"Inner TRY block",!
        SET oref=##class(%Exception.General).%New("<MY BAD>","999",,"General error message")
        THROW oref
        RETURN
    }
    CATCH myexp2 {
        WRITE !,"Inner CATCH block",!
        IF 1=myexp2.%IsA("%Exception.General") {
            WRITE "General ObjectScript exception",!
            WRITE "Name: ", $ZCVT(myexp2.Name, "O", "HTML"), !
            WRITE "Code: ", myexp2.Code, !
        }
        ELSE { WRITE "Some other type of exception",! }
        QUIT
    }
    WRITE !,"back to Outer CATCH block",!
    RETURN
}

```

## 関連項目

- [THROW コマンド](#)
- [TRY コマンド](#)
- [ZBREAK コマンド](#)
- [TRY-CATCH の使用法](#)

# CLOSE (ObjectScript)

ファイルまたはデバイスを終了します。

## 構文

```
CLOSE:pc closearg,...
C:pc closearg,...
```

closearg には、以下を指定できます。

```
device:parameters
```

## 引数

引数	説明
pc	オプション — 後置条件式。
device	閉じるデバイス。
parameters	オプション — デバイスの特性を設定するのに使用される 1 つまたは複数のパラメータ。1 つのパラメータは、CLOSE device:"D" などのように、引用符付きの文字列として指定できます。複数のパラメータは、括弧で囲み、コロンで区切る必要があります。

## 概要

CLOSE device は、指定されたデバイスの所有権を解放し、オプションで parameter を設定し、それを利用可能デバイス・プールに返します。

プロセスが指定のデバイスを所有していない場合、または指定のデバイスが開かれていないか、存在していない場合、InterSystems IRIS は、CLOSE を無視し、エラーを発行せずに返します。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に CLOSE コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### device

閉じるデバイス。device には、物理デバイス、TCP 接続、または[シーケンシャル・ファイルのパス名](#)を指定できます。これには、対応する OPEN コマンドで指定したものと同じデバイス ID (ニーモニック、または数字) を指定します。デバイス ID の指定に関する詳細は、“[OPEN](#)” コマンドを参照してください。

単一のデバイス ID、またはデバイス ID のコンマ区切りのリストを指定できます。CLOSE は、プロセスが現在開いている、リストされているデバイスをすべて閉じます。リストされているデバイスでも、存在しないデバイスや、このプロセスによって現在開かれていないデバイスは無視されます。

現在のデバイスのデバイス ID は、[\\$IO](#) 特殊変数に含まれています。

### parameters

指定のデバイスを閉じるときに使用されるパラメータ、またはコロンで区切られたパラメータのリスト。パラメータでは、大文字と小文字は区別されません。複数のパラメータを指定する場合は、括弧で囲み、コロンで区切る必要があります。

利用できるパラメータ値は以下のとおりです。

パラメータ	概要
"D"	シーケンシャル・ファイルを閉じて削除します。/DEL、/DEL=1、/DELETE、または /DELETE=1 のように指定することもできます。
"R":newname	シーケンシャル・ファイルを閉じ、その名前を変更します。/REN=newname または /RENAME=newname のように指定することもできます。
"K"	オペレーティング・システム・レベルではなく、InterSystems IRIS レベルで閉じます。Windows 以外のシステムでのみ使用されます。

指定された parameter が有効でない場合も、CLOSE はデバイスを閉じます。

詳細は、["シーケンシャル・ファイルの入出力"](#) を参照してください。

## デバイスの所有権を取得する

プロセスは、OPEN コマンドでデバイスの所有権を取得し、USE コマンドでそれをアクティブにします。閉じたデバイスがアクティブ（つまり、現在のデバイス）である場合、既定の入出力デバイスが現在のデバイスとなります。（既定の入出力デバイスは、ログイン時に確立されます）プロセスが終了するとき（例えば HALT の後など）には、開いていたデバイスはすべて自動的に閉じ、システムに戻ります。

既定のデバイスを閉じると、それ以降に、そのデバイスに出力（エラー・メッセージなど）が送られるとプロセスは停止します。この場合は、既定のデバイスを明示的に、再度開く必要があります。

## 例

以下の UNIX® の例では、デバイス C (/dev/tty02) が現在のデバイスでない場合にのみ、CLOSE コマンドでこのデバイスを閉じます。後置条件は、特殊変数 \$IO を使用して、現在のデバイスを確認します。

### ObjectScript

```
CloseDevC
SET C="/dev/tty02"
OPEN C
; ...
CLOSE:$IO'=C C
```

## 関連項目

- ・ [OPEN コマンド](#)
- ・ [入出力の概要](#)

# CONTINUE (ObjectScript)

FOR コマンド、WHILE コマンド、および DO WHILE コマンドにジャンプし、テストを再実行してループします。

## 構文

CONTINUE: *pc*

## 引数

引数	説明
pc	オプション - 後置条件式

## 概要

CONTINUE コマンドは、FOR コマンド、WHILE コマンド、DO WHILE コマンドに続くコード・ブロック内で使用されます。CONTINUE を使用して、FOR、WHILE、および DO WHILE コマンドに戻って実行を継続することができます。FOR、WHILE、および DO WHILE コマンドはテスト条件を評価し、その評価を基にしてコード・ブロック・ループを再実行します。したがって、CONTINUE コマンドの実行にはコード・ブロックの終了括弧 (}) に到達するのと同じ効果があります。

CONTINUE は引数を取りません (後置条件以外)。同じ行でその後に続くコマンドを、最低 2 つの空白スペースで分割する必要があります。

CONTINUE は、TRY または CATCH ブロックの外側にジャンプして、そのフロー制御文を返すことができます。

## 引数

### pc

コマンドを条件付きにする、オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に CONTINUE コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、"[コマンド後置条件式](#)" を参照してください。

## 例

以下の CONTINUE の例では、後置条件式を使用しています。ループして、1 から 10 (3を除く) のすべての数を入力します。

### ObjectScript

```
Loop
FOR i=1:1:10 {
    IF i # 2 {
        CONTINUE:i=3
        WRITE !,i," is odd" }
    ELSE { WRITE !,i," is even" }
    }
WRITE !,"done with the loop"
QUIT
```

次の例は、入れ子になった 2 つの FOR ループを示しています。CONTINUE は、内部ループの FOR に戻ります。

## ObjectScript

```
Loop
  FOR i=1:1:3 {
    WRITE !,"outer loop: i=",i
    FOR j=2:2:10 {
      WRITE !,"inner loop: j=",j
      IF j '= 8 {CONTINUE }
      ELSE { WRITE " crazy eight"}
    }
    WRITE !,"back to outer loop"
  }
QUIT
```

次の例は、TRY ブロックが存在する CONTINUE を示しています。CONTINUE は、TRY ブロックの外側の FOR 構文に戻ります。

## ObjectScript

```
TryLoop
  FOR i=1:1:10 {
    WRITE !,"Top of FOR loop"
    TRY {
      WRITE !,"In TRY block: i=",i
      IF i=7 {
        WRITE " lucky seven" }
      ELSE { CONTINUE }
    }
    CATCH exp {
      WRITE !,"CATCH block exception handler",!
      WRITE "Error code=",exp.Code
      RETURN
    }
    WRITE !,"Bottom of the FOR loop"
  }
QUIT
```

## 関連項目

- ・ [DO WHILE コマンド](#)
- ・ [FOR コマンド](#)
- ・ [WHILE コマンド](#)



## DO (ObjectScript)

ルーチン呼び出します。

### 構文

```
DO:pc doargument,...
D:pc doargument,...
```

doargument には、以下を指定できます。

```
entryref(param,...):pc
```

### 引数

引数	説明
pc	オプション — 後置条件式。
entryref	呼び出されるルーチンの名前。ルーチン名の前にはキャレットが付きます (DO ^myroutine)。オプションで、ラベル名 (DO Label2^myroutine) や、ラベルと行オフセット (DO Label2+3^myroutine) を指定します。 <a href="#">現在のルーチン</a> を呼び出す場合は、ルーチン名を省略して、ラベル、またはラベルと行オフセットのみを指定できます。以下に示すように、DO oref.Method() としてオブジェクト・メソッドを呼び出したり、別の構文形式を使用したりすることもできます。
param	オプション — 呼び出されるルーチンに渡されるパラメータ値。

### 説明

**注釈** DO と DO WHILE は異なる無関係のコマンドです。ここでは、DO コマンドについて説明します。[DO WHILE](#) コマンドでは、DO キーワードと WHILE キーワードは、複数のコード行によって分割されています。しかし、DO キーワードの後には左中括弧が続くので DO WHILE コマンドの識別は簡単です。

DO コマンドは、特定のオブジェクト・メソッド、サブルーチン、関数、またはプロシージャを呼び出します。InterSystems IRIS は、呼び出されたルーチンを実行し、DO コマンドの次のコマンドを実行します。ルーチンの呼び出しでは、パラメータは渡しても渡さなくてもかまいません。例えば、DO ^myrou1 または DO ^myrou(a,b,c) となります。

DO は呼び出されたルーチンからの戻り値を受け取れません。呼び出されたルーチンが引数付きの QUIT で終了している場合、DO コマンドは正常に終了しますが、QUIT の引数値は無視されます。

DO では、複数の引数を、コンマ区切りのリストで指定することができます。例えば、DO ^myrou1,^myrou2,^myrou3 となります。引数は指定した順序で実行されます。無効な引数が見つかった場合、実行は停止します。

DO は、通常、既存のコンパイル済みルーチンを実行する場合にターミナル・プロンプトで呼び出します。もちろん、ルーチンの内部から DO コマンドを呼び出すこともできます。

DO の各呼び出しは、新しいコンテキスト・フレームをプロセスのコール・スタックに配置します。\$STACK 特殊変数は、コール・スタックのコンテキスト・フレームの現在の番号を含みます。このコンテキスト・フレームは、新しい実行レベルを作成して、\$STACK と \$ESTACK をインクリメントし、DO の実行中に発行された NEW および SET \$ZTRAP の操作のスコープを提供します。正常に終了すると、DO は \$STACK と \$ESTACK をデクリメントし、NEW と SET \$ZTRAP の操作を元に戻します。

## 現在のルーチン

ターミナルから DO を呼び出した場合、DO は最初に、[現在ロードされているルーチン](#)を検索します。現在のプロセスに現在のルーチンがある場合は、DO は、ディスク上の対応するルーチンではなく、この現在のルーチンを実行します。

例えば、ZLOAD コマンドを使用して、ディスクからルーチン myroutine をロードするとします。これにより、このルーチンが現在のルーチンになります (\$ZNAME 特殊変数で表示されます)。その後、ZINSERT を使用して現在のルーチンを変更してから、DO ^myroutine を呼び出します。DO コマンドで実行されるのは、ディスク上の変更されていない myroutine ではなく、変更された myroutine です。[引数なしの ZREMOVE](#) コマンドで、現在ロードされているルーチンをアンロードします。引数なしの ZREMOVE の後に DO ^myroutine を実行すると、ディスクから、変更されていないバージョンの myroutine が実行されます。

現在のルーチンは、以下の 2 つの方法のいずれかで呼び出すことができます。

- ・ 明示的なルーチン名を使用する。DO ^myroutine、DO Label2^myroutine、または DO Label2+3^myroutine のようになります。
- ・ 暗黙的なルーチン名を使用して、ラベルとオフセット (オプション) を指定する。DO Main、DO Label2、または DO Label2+3 のようになります。

## 引数

### pc

オプションの後置条件式です。後置条件式が DO コマンド・キーワードに追加されている場合、InterSystems IRIS は後置条件式が True (0 以外の数値に評価される) の場合に DO コマンドを実行します。後置条件式が False (0 に評価される) の場合、InterSystems IRIS は DO コマンドを実行しません。

後置条件式が引数に追加されている場合、InterSystems IRIS は後置条件式が True (0 以外の数値に評価される) の場合に引数を実行します。後置条件式が False (0 に評価される) の場合、InterSystems IRIS はその引数をスキップし、次の引数 (存在する場合) もしくは次のコマンドの評価に進みます。次に、例を示します。

### ObjectScript

```
DO:0 $INCREMENT(myvar($INCREMENT(subvar))):1 /* myvar and subvar not incremented */
DO:1 $INCREMENT(myvar($INCREMENT(subvar))):0 /* myvar not incremented, subvar incremented */
```

InterSystems IRIS は左から右へ式を処理するため、後置条件式が評価される前に、式を含む引数部分 (パラメータ値、オブジェクト参照など) が評価され、エラーが発生する場合があります。後置条件式を追加して DO でオブジェクト・メソッドを呼び出す場合、オブジェクト・メソッド・パラメータの最大個数は 253 です。

詳細は、“[コマンド後置条件式](#)” を参照してください。

### entryref

呼び出すルーチン (オブジェクト・メソッド、[サブルーチン](#)、[プロシージャ](#)、または[ユーザ指定関数](#)) の名前。複数のルーチンを、コンマ区切りのリストで指定することができます。

entryref は以下のすべての形式で指定できます。

entryref 形式	説明
label+offset	現在のルーチン内の行ラベルを指定します。オプションの +offset は、パラメータを渡されていないサブルーチン呼び出しのためにだけ使用します。プロシージャの呼び出し、もしくはサブルーチンへパラメータを渡すためには使用できません。offset は負でない整数で、ラベルの何行後からサブルーチンの実行を開始するかを指定します。
label+offset^routine	ディスクに保存されている指定したルーチン内の行ラベルを指定します。InterSystems IRIS は、ディスクからそのルーチンをロードし、指定されたラベルから実行を開始します。+offset はオプションです。
^routine	ディスクにあるルーチンの名前。システムは、ディスクからそのルーチンをロードし、そのルーチン内の最初の実行可能行から実行を開始します。リテラル値である必要があります。変数は、routine の指定には使用できません。( ^ 文字は区切り文字で、ルーチン名の一部ではないことに注意してください。) ルーチンが変更されている場合、DO がそのルーチン呼び出すときに、InterSystems IRIS は更新バージョンのルーチンをロードします。ルーチンが現在のネームスペースにない場合、 <a href="#">拡張ルーチン参照</a> を使用して、ルーチンを含むネームスペースを ^ "namespace" routine のように指定できます。
oref.Method()	オブジェクト・メソッドを指定します。システムはオブジェクトにアクセスして、指定したメソッドを実行します。(存在する場合は) param で指定した引数とメソッドの引数リストを渡します。オブジェクトの呼び出しは、ドット構文を使用します。oref (オブジェクト参照) と Method() は、ドットで区切られます。空白スペースは許可されません。param 引数がない場合でも、開き括弧と閉じ括弧は必要です。  サポートされている構文形式は、DO oref.Method()、DO (oref).Method()、DO ..Method()、DO ##class(cname).Method()、DO i%prop(subs).Method() です。

IRISSYS % ルーチンの呼び出し時における offset の指定はできません。指定しようとすると、InterSystems IRIS は <NOLINE> エラーを発行します。

存在しないラベルを指定した場合、InterSystems IRIS は <NOLINE> エラーを返します。存在しないルーチンを指定した場合、InterSystems IRIS は <NOROUTINE> エラーを返します。存在しないメソッドを指定した場合、InterSystems IRIS は <METHOD DOES NOT EXIST> エラーを返します。既存のプロパティを (括弧で囲んで) メソッドとして指定した場合、InterSystems IRIS は <OBJECT DISPATCH> エラーを返します。拡張参照 (例えば、DO ^|"SYS"|MyProg) を使用して、存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを返します。拡張参照を使用して、特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを返し、続けてデータベース・パスを表示します (例: <PROTECT> \*^|^^c:\intersystems\iris\mgr\|MyRoutine)。これらのエラーの詳細は、"[\\$ZERROR](#)" 特殊変数を参照してください。

複数行の構文中を指す offset を指定すると、システムでは次の構文の最初で実行を開始します。

### param

サブルーチン、プロシージャ、ユーザ指定関数、またはオブジェクト・メソッドに渡されるパラメータ値です。単一の param 値、あるいはコンマで区切られた param 値のリストを指定できます。param リストは、括弧で囲まれます。param が指定されていない場合、プロシージャ、ユーザ指定関数、またオプションでサブルーチン呼び出す際に括弧が必要です。パラメータは、値または参照によって渡されます。同じ呼び出しで、値渡しによるパラメータと参照渡しによるパラメータを混合して指定できます。値渡しの場合、パラメータを値定数、式、または添え字なしのローカル変数名として指定できます。

(“[値渡し](#)”を参照してください。)参照渡しの場合、パラメータは、ローカル変数の名前、または .name. という形式の添え字なしの配列を参照する必要があります (詳細は“[参照渡し](#)”を参照してください)。

1 つの DO エントリポイントの param 値の最大合計数は 382、1 つの DO メソッドまたは DO 間接演算の param 値の最大合計数は 380 です。この合計には、最大で 254 の [実パラメータ](#) と 128 の [後置条件パラメータ](#) を含めることができます。

... 構文を使用して [可変個数のパラメータ](#) を指定できます。

## DO コマンドの entryref 引数

entryref 引数付きの DO コマンドは、1 つまたは複数の定義済みのコード・ブロックの実行を呼び出します。実行する各コード・ブロックは、entryref で指定されます。DO コマンドは、コンマで区切られたリストとして、実行する複数のコード・ブロックを指定できます。DO コマンドの実行とコンマで区切られたリストにある各 entryref の実行は、オプションの後置条件式で管理できます。

DO は、(パラメータ付き、あるいはパラメータなしの) サブルーチン、プロシージャ、ユーザ指定関数の実行を呼び出します。ブロック・コードの実行が終了した場合、DO コマンドの直後のコマンドで再開します。DO コマンドで呼び出されたコード・ブロックは、DO コマンドに値を返すことはできません。返された値はいずれも無視されます。したがって、DO はユーザ指定関数を実行しますが、関数の返り値の取得はできません。

DO は、ほとんどの ObjectScript の [システム関数](#) を呼び出すことができません。呼び出しを試みると、<SYNTAX> エラーが返されます。いくつかのシステム関数を DO コマンドの引数として呼び出すことができます: [\\$CASE](#)、[\\$CLASSMETHOD](#)、[\\$METHOD](#)、[\\$INCREMENT](#)、および [\\$ZF\(-100\)](#)。DO は関数の返り値を受け取ることはできません。すべての DO コマンド引数と同様に、これらの関数は [後置条件パラメータ](#) を取ることができます。例えば、次のようになります。DO `$CASE(exp,0:NoMul(),2:Square(num),3:Cube(num),:Exponent(num,exp)):0` プログラム例については、“[\\$CASE](#)” 関数を参照してください。

## パラメータ渡しなしの DO コマンド

パラメータ渡しなしの DO コマンドは、サブルーチンと一緒にの場合にのみ使用されます。DO entryref にパラメータを渡さない (つまり、param オプションを指定しない) で使用すると、呼び出し元のルーチンとそれによって呼び出されたサブルーチンで、同じ変数環境が共有されるという事実を利用できます。サブルーチンにより変数の更新が行われた場合、これらはすべて自動的に DO コマンドの後のコードで使用可能となります。

パラメータを渡さずに DO を使用する場合は、呼び出し元のルーチンと呼び出されるサブルーチンが同じ変数を参照していることを確認する必要があります。

**注釈**    プロシージャによる変数の処理は、まったく異なります。“[呼び出し可能なユーザ定義コードモジュール](#)”を参照してください。

以下の例では、Start (呼び出し元のルーチン) と Exponent (呼び出されるサブルーチン) が 3 つの変数 num、powr、および result へのアクセスを共有しています。Start では、num と powr がユーザ指定の値に設定されます。これらの値は、Exponent が DO コマンドによって呼び出されたときに、自動的に Exponent でも利用できます。Exponent は、num と powr を参照し、計算した値を result に代入します。Exponent が RETURN コマンドを実行する場合、制御は DO の後の WRITE コマンドに即座に戻ります。WRITE コマンドは、result を参照して計算された値を出力します。

### ObjectScript

```
Start ; Raise an integer to a specified power.
  READ !,"Integer= ",num QUIT:num=""
  READ !,"Power= ",powr QUIT:powr=""
  DO Exponent()
  WRITE !,"Result= ",result,!
  RETURN
Exponent()
  SET result=num
  FOR i=1:1:powr-1 { SET result=result*num }
  RETURN
```

以下の例で、DO は pat によって参照されるオブジェクトで、Admit() メソッドを呼び出します。このメソッドはパラメータを受け取らず、また値も返しません。

## ObjectScript

```
DO pat.Admit()
```

以下の例では、DO が、[現在のルーチン](#)内の Init サブルーチンと Read1 サブルーチン、および Test ルーチン内の Convert サブルーチンを連続して呼び出します。

## ObjectScript

```
DO Init,Read1,Convert^Test
```

以下の例では、DO は拡張参照を使用して、別のネームスペース (SAMPLES ネームスペース) の fibonacci ルーチンを呼び出しています。

## ObjectScript

```
NEW $NAMESPACE
SET $NAMESPACE="USER"
DO ^|"SAMPLES"|fibonacci
```

## DO と GOTO

DO コマンドを使用して、(パラメータ付き、あるいはなしの) サブルーチン、プロシージャ、ユーザ指定関数を呼び出します。呼び出しの終了時、InterSystems IRIS は DO コマンドに続く次のコマンドを実行します。

GOTO コマンドは、パラメータ渡しなしのサブルーチンを呼び出すためにのみ使用します。呼び出しの終了時、InterSystems IRIS は QUIT を呼び出し、実行を終了します。

## パラメータ渡しの DO

パラメータ渡しが使用される場合、DO entryref は、呼び出したサブルーチン、プロシージャ、ユーザ指定関数、またはオブジェクト・メソッドに 1 つ、または複数の値を明示的に渡します。渡す値は、コンマで区切られたリストとして param オプションで指定されたものです。パラメータ渡しを行う場合は、呼び出されるサブルーチンがパラメータ・リスト付きで定義されているかを確認する必要があります。サブルーチンの定義の形式は、以下のとおりです。

```
>label( param)
```

label では、サブルーチン、プロシージャ、ユーザ指定関数、またはオブジェクト・メソッドの[ラベル名](#)を指定します。param では、コンマで区切られた添え字なしのローカル変数名 1 つ、または複数からなるリストを指定します。以下はその例です。

## ObjectScript

```
Main
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,y,z)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  WRITE !,"In Sub1 ",a,b,c
  QUIT
```

DO コマンドで渡されるパラメータ・リストは、実パラメータ・リスト (実リスト) と呼ばれます。コード化されたルーチン・ラベルの一部として定義されるパラメータ変数のリストは、仮パラメータ・リスト (仮リスト) と呼ばれます。DO コマンドがルーチンを読み出すとき、実リストのパラメータは、その位置に従って、仮リスト内の対応する変数にマップされます。上述の例では、1 番目の実パラメータ (x) の値は、サブルーチンの仮パラメータ・リストにある 1 番目の変数 (a) に代入され、2 番目の実パラメータ (y) の値は 2 番目の変数 (b) に代入されます。その後、サブルーチンは、仮パラメータ・リストの変数を参照することによって、渡された値にアクセスできます。

実パラメータ・リスト内の変数が、仮パラメータ・リスト内のパラメータよりも多い場合、InterSystems IRIS は <PARAMETER> エラーを発行します。

仮リスト内の変数が、実リスト内のパラメータよりも多い場合は、余った変数は未定義のままになります。以下の例は、仮パラメータ c が未定義のままになっています。

### ObjectScript

```
Main
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,y)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  WRITE !,"In Sub1 "
  IF $DATA(a) {WRITE !,"a=",a}
  ELSE {WRITE !,"a is undefined"}
  IF $DATA(b) {WRITE !,"b=",b}
  ELSE {WRITE !,"b is undefined"}
  IF $DATA(c) {WRITE !,"c=",c}
  ELSE {WRITE !,"c is undefined"}
  QUIT
```

実パラメータ値が指定されていない場合、仮パラメータに対し既定値を指定できます。

DO コマンドの実パラメータ・リストから、対応するパラメータを削除して、変数を未定義のままにできます。ただし、実パラメータを削除した個所に、プレースホルダとしてコンマを入れる必要があります。以下の例は、仮パラメータ b が未定義のままになっています。

### ObjectScript

```
Main
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,,z)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  WRITE !,"In Sub1 "
  IF $DATA(a) {WRITE !,"a=",a}
  ELSE {WRITE !,"a is undefined"}
  IF $DATA(b) {WRITE !,"b=",b}
  ELSE {WRITE !,"b is undefined"}
  IF $DATA(c) {WRITE !,"c=",c}
  ELSE {WRITE !,"c is undefined"}
  QUIT
```

... 構文を使用して、可変個数のパラメータを指定できます。

### ObjectScript

```
Main
  SET x=3,x(1)=10,x(2)=20,x(3)=30
  DO Sub1(x...)
  QUIT
Sub1(a,b,c)
  WRITE a," ",b," ",c
  QUIT
```

DO コマンドは、パラメータの値渡し (DO Sub1(x,y,z) など)、または参照渡し (DO Sub1(.x,.y,.z) など) のいずれかを行います。同じ DO コマンド内で、値渡しと参照渡しを混合して指定することもできます。詳細は“[パラメータ渡し](#)”を参照してください。

以下の例は、値渡しと参照渡しの違いを示しています。



## ObjectScript

```
Main /* Passing by Value */
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(x,y,z)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(a,b,c)
  SET a=a+1,b=b+1,c=c+1
  WRITE !,"In Sub1 ",a,b,c
  QUIT
```

## ObjectScript

```
Main /* Passing by Reference */
  SET x=1,y=2,z=3
  WRITE !,"In Main ",x,y,z
  DO Sub1(.x,.y,.z)
  WRITE !,"Back in Main ",x,y,z
  QUIT
Sub1(&a,&b,&c) /* The & prefix is an optional by-reference marker */
  SET a=a+1,b=b+1,c=c+1
  WRITE !,"In Sub1 ",a,b,c
  QUIT
```

## 間接指定の DO

間接指定を使用して、DO にターゲットのサブルーチンの場所を提供することができます。例えば、さまざまなメニュー関数を別のルーチン内の異なる場所に組み込む方法で、一般的なメニュー・プログラムを実装するとします。メイン・プログラム・コードで、名前による間接指定を使用して、DO コマンドにメニューの各選択項目に対応するサブルーチンの場所を提供することができます。

InterSystems IRIS オブジェクト・ドット構文を使用して、間接指定をすることはできません。ドット構文が実行時ではなく、コンパイル時に構文解析されるためです。

名前による間接指定の場合、間接指定演算子 (@) の右側の式の値が名前 ([行ラベル](#) またはルーチン) でなければなりません。以下のコード部分では、名前による間接指定によって、DO コマンドに Menu ルーチン内のターゲット・サブルーチンの場所が提供されています。

## ObjectScript

```
READ !,"Enter the number for your choice: ",num QUIT:num=""
DO @("Item"_num)^Menu
```

DO コマンドは、Menu 内のサブルーチンを呼び出します。このサブルーチンのラベルは、Item とユーザ指定の num 値が連結されたもの (Item1、Item2 など) です。

間接指定の引数形式を使用して、完全な DO 引数の代わりに、式の値を使用することもできます。例えば、以下の DO コマンドを見てみましょう。

## ObjectScript

```
DO @(eref_":fstr>0")
```

このコマンドは、fstr の値が 0 より大きい場合に、eref の値で指定されたサブルーチンを呼び出します。

詳細は、“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

## 引数後置条件付きの DO

DO コマンドのターゲット・サブルーチンを選択するには、引数後置条件式を使用できます。後置条件式が False (0) で評価される場合、InterSystems IRIS は関連付けられているサブルーチン呼び出しを無視します。後置条件式が True (1) で評価される場合、InterSystems IRIS は関連付けられているサブルーチン呼び出しを実行してから、DO コマンドに戻ります。DO コマンドの引数とその引数の両方で、後置条件を使用できます。

例えば、以下のコマンドを考えてみます。

### ObjectScript

```
DO:F>0 A:F=1,B:F=2,C
```

DO コマンドには後置条件式があり、F が 0 以下の場合、DO は実行されません。DO コマンドの引数にも、後置条件式があります。DO は後置条件のこれらの引数を使用して、実行するサブルーチン (A、B、C) を選択します。True 条件を満たすすべてのサブルーチンが、指定されている順番に実行されます。つまり、この例では、後置条件式のない C は常に実行されます。F が 1 の場合は A と C が実行され、F が 2 の場合は B と C が実行され、F が 3 (または他の任意の数) の場合は C が実行されます。True の既定として C を指定する場合は、次のようにします。

### ObjectScript

```
DO:F>0 A:F=1,B:F=2,C:((F'=1)&&(F'=2))
```

この例では、実行されるサブルーチンは 1 つのみになります。

以下の例で、DO コマンドは後置条件式を取り、その引数もそれぞれ後置条件式を取ります。この場合、最初の引数は後置条件式が 0 であるため、実行されません。2 つ目の引数は後置条件式が 1 であるため実行されます。

### ObjectScript

```
Main
SET x=1,y=2,z=3
WRITE !,"In Main ",x,y,z
DO:1 Sub1(x,y,z):0,Sub2(x,y,z):1
WRITE !,"Back in Main ",x,y,z
QUIT
Sub1(a,b,c)
WRITE !,"In Sub1 ",a,b,c
QUIT
Sub2(d,e,f)
WRITE !,"In Sub2 ",d,e,f
QUIT
```

DO によって呼び出されるオブジェクト (oref) メソッドのほとんどは、引数後置条件式を取ることができます。ただし、\$SYSTEM オブジェクト・メソッドは、引数後置条件式を取ることができません。これを実行しようとすると、<SYNTAX> エラーが生成されます。

InterSystems IRIS では、式は必ず左から右の順番で評価されるため、引数後置条件式が評価される前に、式を含む引数が評価されます (エラーが生成される場合もあります)。

引数の後置条件式を使用するときは、副作用がないことを確認してください。例えば、以下のコマンドを考えてみます。

### ObjectScript

```
DO @^Control(i):z=1
```

この場合 ^Control(i) には、後置条件 z=1 が True の場合に呼び出すサブルーチンの名前が含まれています。z=1 であるかどうかにかかわらず、InterSystems IRIS は ^Control(i) の値を評価し、それに従って、現在のグローバル・ネイキッド・インジケータを再設定します。z=1 が False の場合、InterSystems IRIS は DO を実行しません。ただし、グローバル・ネイキッド・インジケータは、DO を実行したかのようにリセットされます。ネイキッド・インジケータの詳細は、["ネイキッド・グローバル参照"](#) を参照してください。

後置条件式の評価方法に関する詳細は、["コマンド後置条件式"](#) を参照してください。

## DO を使用した場合の \$TEST の振る舞い

DO を使用してプロシージャを呼び出すと、InterSystems IRIS では \$TEST の値を保持します。これは、プロシージャを終了するときの呼び出し時に、値をその状態にリストアすることで保持されます。ただし、DO を使用してサブルーチンを呼び出す場合 (パラメータを渡す場合も渡さない場合も) は、その呼び出しの間、InterSystems IRIS では \$TEST の値は保存しません。



\$TEST 値を DO 呼び出しの間も保存するためには、その呼び出しの前に、この値を明示的に変数に代入します。その後、呼び出しが続くコードで変数を参照できます。

## 関連項目

- ・ [GOTO コマンド](#)
- ・ [XECUTE コマンド](#)
- ・ [NEW コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [\\$CASE 関数](#)
- ・ [\\$ESTACK 特殊変数](#)
- ・ [\\$STACK 特殊変数](#)
- ・ [サブルーチン](#)
- ・ [プロシージャ](#)
- ・ [パラメータ渡し](#)

## DO WHILE (ObjectScript)

条件が存在する間に、コードを実行します。

### 構文

```
DO {code} WHILE expression,...
D {code} WHILE expression,...
```

### 引数

引数	説明
code	中括弧で囲まれた ObjectScript コマンドのブロックです。
expression	ブーリアン・テスト条件式、またはコンマで区切られたブーリアン・テスト条件式のリストです。オプションで、この式を括弧で囲むことができます。

### 説明

DO WHILE は code を実行し、expression を評価します。expression が True と評価される場合、DO WHILE はループし、code を再実行します。expression が True と評価されない場合、code は実行されず、DO WHILE の後続のコマンドが実行されます。

DO WHILE は常に、ブロック型形式で記述されることに注意してください。実行されるコードは DO と WHILE キーワードの間に配置され、{} で囲まれます。

開き中括弧や閉じ中括弧は、1 行で独立して記述するか、コマンドと同じ行に記述できます。開き中括弧や閉じ中括弧は、列 1 に記述してもかまいませんが、お勧めはできません。推奨されるプログラミング手法として、入れ子になったコード・ブロックの開始と終了を示すために、中括弧はインデントするようにしてください。開き中括弧の前後に空白を入れる必要はありません。閉じ中括弧の前後に空白を入れる必要はありません。WHILE キーワードと、括弧で囲んだ expression との間に空白は必要ありません。閉じ中括弧と WHILE キーワードの間にコメントが表示される場合があります。

DO キーワードは短縮される場合があります。

(関連しない DO コマンドとは異なり) DO WHILE は、新しい実行レベルを作成しません。NEW や SET \$ZTRAP など、DO WHILE ループ間に呼び出され、実行レベルの影響を受けるコマンドは、ループの終了後も引き続き有効です。

### 引数

#### code

1 つ、または複数の Objectscript コマンドのブロックです。コード・ブロックは数行にわたることがあります。コード・ブロックは、中括弧({}) で囲まれます。コード・ブロックとコマンド内の引数の内部にあるコマンドとコメントは、1 つ、または複数の空白スペースや改行で区切られます。しかし、すべての ObjectScript コマンドのように、各コマンド・キーワードとその最初の引数は必ず 1 つのスペースで区切る必要があります。

#### expression

単一の式、またはコンマで区切られた式のリストの形式をとるテスト条件。式リストでは、InterSystems IRIS は左から右の順で、個別の式を評価します。False である式に遭遇すると評価を終了します。すべての式が True と評価された場合、InterSystems IRIS は code コマンドを再実行します。DO WHILE は、各ループに対する expression をテストしながら、繰り返し実行されます。式が 1 つでも False と評価されると、InterSystems IRIS は残りの式を無視し、ループも行われません。そして、DO WHILE の後のコマンドから実行されます。

ObjectScript は、厳密に左から右の順に式を評価します。プログラマは、括弧を使用して優先順位を確立する必要があります。

注釈 InterSystems IRIS は code を実行する前に expression の検証を実行しません。したがって、DO WHILE はその code ループを常に 1 回実行します。expression を正常に実行できるかどうかは関係ありません。

## DO WHILE と WHILE

DO WHILE コマンドはループを一度実行し、そして expression をテストします。WHILE コマンドは、ループを実行する前に expression をテストします。

## DO WHILE と CONTINUE

DO WHILE コマンドの code ブロック内で **CONTINUE** コマンドに遭遇すると、実行が即座に WHILE キーワードにジャンプします。次に DO WHILE コマンドは WHILE expression テスト条件を評価し、その評価を基にして code ブロック・ループを再実行するかどうかを決定します。したがって、CONTINUE コマンドの実行は code ブロックの終了中括弧に到達するのとまったく同じ効果があります。

以下の例は、奇数ループの結果のみを表示します。

### ObjectScript

```
SET x=0
DO {SET x=x+1 IF x#2=0 {CONTINUE} WRITE !,"Loop ",x} WHILE x<20
WRITE !,"DONE"
```

## DO WHILE、QUIT、RETURN

code ブロック内の **QUIT** コマンドは DO WHILE ループを終了させ、WHILE キーワードに続くコマンドへ実行を移します。以下はその例です。

### ObjectScript

```
Testloop
SET x=1
DO {
    WRITE !,"Looping",x
    QUIT:x=5
    SET x=x+1
} WHILE x<10
WRITE !,"DONE"
```

このプログラムは、Looping1 から Looping5 までを書き込み、その後 DONE を書き込みます。

DO WHILE コード・ブロックは入れ子にできます。つまり、DO WHILE コード・ブロックに、別のフロー制御ループ (別の DO WHILE、あるいは FOR または WHILE コード・ブロック) を含むことができます。内側の入れ子となったループの QUIT では、内側のループが終了して、そのループの外側のループに実行が移ります。詳細は、以下の例を参照してください。

### ObjectScript

```
Nestedloops
SET x=1,y=1
DO {
    WRITE "outer loop ",!
    DO {
        WRITE "inner loop "
        WRITE " y=",y,!
        QUIT:y=7
        SET y=y+2
    } WHILE y<100
    WRITE "back to outer loop x=",x,!
    SET x=x+1
} WHILE x<6
WRITE "Done"
```

**RETURN** を使用すると、DO WHILE ループや入れ子のループ構造内を含む任意の場所からルーチンの実行を終了させることができます。RETURN は常に現在のルーチンを終了し、呼び出し元のルーチンに戻るか、呼び出し元のルーチン

ンがない場合はプログラムを終了します。RETURN は、コード・ブロック内から発行されたかどうかに関係なく、常に同じ動作を行います。

## DO WHILE と GOTO

code ブロック内の GOTO コマンドは、ループ外のラベルへ実行をダイレクトし、ループを終了します。(ラベルはタグと呼ばれることもあります。)code ブロック内の GOTO コマンドは、同じ code ブロック内のラベルへ実行をダイレクトする場合があります。入れ子になったコード・ブロックを終了するために、GOTO を使用できます。サポートはされていますが、他の GOTO の使用法はお勧めしません。

以下の例では、GOTO を使用して DO WHILE ループを終了します。

### ObjectScript

```
mainloop
DO {
    WRITE !,"In an infinite DO WHILE loop"
    GOTO label1
    WRITE !,"This should not display"
} WHILE 1=1
WRITE !,"This should not display"
label1
WRITE !,"Went to label1 and quit"
QUIT
```

以下の例では、GOTO を使用して DO WHILE ループ内で実行を移動します。

### ObjectScript

```
mainloop ; Example of a GOTO to within the code block
SET x=1
DO {
    WRITE !,"In the DO WHILE loop"
    GOTO label1
    WRITE !,"This should not display"
label1
    WRITE !,"Still in the DO WHILE loop after GOTO"
    SET x=x+1
    WRITE !,"x= ",x
    } WHILE x<3
WRITE !,"DO WHILE loop done"
```

## 例

以下の例では、最初の DO WHILE は式が True で、次の DO WHILE は False です。式が False の場合、そのコード・ブロックは一度だけ実行されます。

### ObjectScript

```
DoWhileTrue
SET x=1
DO {
    WRITE !,"Looping",x
    SET x=x+1
} WHILE x<10
WRITE !,"DONE"
```

このプログラムは、Looping1 から Looping9 までを書き込み、その後 DONE を書き込みます。

### ObjectScript

```
DoWhileFalse
SET x=11
DO {
    WRITE !,"Looping",x
    SET x=x+1
} WHILE x<10
WRITE " DONE"
```

このプログラムは、Looping11 DONE を書き込みます。

## 関連項目

- ・ [WHILE コマンド](#)
- ・ [FOR コマンド](#)
- ・ [IF コマンド](#)
- ・ [CONTINUE コマンド](#)
- ・ [GOTO コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [RETURN コマンド](#)

## ELSE (ObjectScript)

---

ブロック型 IF コマンドの文節です。

### 構文

```
ELSE { code }
```

完全な構文については、“[IF](#)” コマンドを参照してください。

### 概要

ELSE は独立したコマンドではなく、ブロック型 IF コマンドの文節です。単一の ELSE 節は IF コマンドの最終節として指定できます。また、この ELSE 節は省略することもできます。詳細と例は、“[IF](#)” コマンドを参照してください。

**注釈** ELSE コマンドの初期バージョンは、従来のアプリケーションで行型の IF コマンドと使用されていました。これらのコマンドは、{ } 括弧を使用しないため、独立したコマンドとして認識される場合があります。IF と ELSE の従来のフォームと新規のフォームは構文的には異なり、結合することができません。したがって、あるタイプの IF は別のタイプの ELSE と組み合わせることはできません。

初期の行型 ELSE コマンドは、E と省略することができますが、ブロック型の ELSE キーワードは省略できません。

ELSE キーワードの後には、開き中括弧 ( { ) と閉じ中括弧 ( } ) を続けて記述する必要があります。通常、これらの { } は、コード・ブロックを囲みます。ただし、コード・ブロックのない ELSE も、以下のように許可されます。

#### ObjectScript

```
      SET x=1
Loop
  IF x=1{
      WRITE "Once only"
      SET x=x+1
      GOTO Loop
    }
  ELSE{ }
  WRITE !, "All done"
```

ELSE キーワードには、空白の制約はありません。

### 関連項目

- ・ [IF コマンド](#)
- ・ [フロー制御コマンド](#)

# ELSEIF (ObjectScript)

---

ブロック型 IF コマンドの文節です。

## 構文

```
ELSEIF expression, ... { code }
```

完全な構文については、“[IF](#)” コマンドを参照してください。

## 概要

ELSEIF は独立したコマンドではなく、ブロック型 IF コマンドの文節です。1つの IF コマンドで、1 つまたは複数の ELSEIF 節を指定できます。この ELSEIF 節は省略することもできます。

ELSEIF はブーリアン式 (またはブーリアン式のコンマ区切りリスト) を評価し、すべての式が true に評価された場合、code ブロックを実行します。複数の ELSEIF 節がある場合、最初に true に評価された節のみが実行されます。詳細と例は、“[IF](#)” コマンドを参照してください。

## 関連項目

- ・ [IF コマンド](#)
- ・ [フロー制御コマンド](#)

## FOR (ObjectScript)

各ループの最初でテストしながら、コード・ブロックを繰り返し実行します。

### 構文

```
FOR var=forparameter { code } F var=forparameter { code }
FOR var=forparameter,forparameter2,... { code }
F var=forparameter,forparameter2,... { code }
```

forparameter には、以下を指定できます。

```
expr start:increment start:increment:end
```

### 引数

引数	説明
var	オプション - FOR コマンドで初期化されるローカル変数またはインスタンス変数。これは、一般に、code ブロックが実行されるたびにインクリメントされる数値のカウンタです。
expr	オプション - code ブロックを実行する前に、var に割り当てられる値。単一の値、または値のコンマ区切りリストになります。
start	オプション - code ブロックの最初の実行前に、var に割り当てられる数値。increment および (オプションの) end と併用して、FOR ループの複数の反復を制御します。
increment	オプション - FOR ループの各反復後の var のインクリメント (またはデクリメント) に使用される数値。
end	オプション - FOR ループを終了するために使用される数値。ループ処理は、インクリメントされた var が end 以上になったときに終了します。
code	中括弧で囲まれた ObjectScript コマンドのブロックです。

### 概要

FOR はブロック型のコマンドです。一般に、カウンタと、中括弧で囲まれた実行可能なコード・ブロックで構成されます。このコード・ブロックが実行される回数は、カウンタで決定されます。このカウンタは、各ループの先頭でテストされます。あまり一般的ではありませんが、インクリメントするカウンタを指定しない FOR コマンドもあります。これは、引数なし (終了するまでの無限ループ) にすることも、引数として式を指定する (1 回のループ) こともできます。

FOR コマンドには、以下の 2 つの基本的な形式があります。

- ・ [引数なし](#)
- ・ [引数付き](#)

### 引数なしの FOR

引数なしの FOR は、コード・ブロック内のコマンドで終了するまで、無限にループ・コード・ブロックを実行します。InterSystems IRIS は、QUIT、RETURN、またはループから抜け出す GOTO コマンドに遭遇するまで、中括弧内のコマンドを繰り返し実行します。以下の例では、x = 3 のときにループを終了します。



## ObjectScript

```
SET x=8
FOR { WRITE "Running loop x=",x,!
      SET x=x-1
      QUIT:x=3
    }
WRITE "Next command after FOR code block"
```

以下の例に示すように、エラーでも FOR ループから抜け出します。この FOR ループは、0 による除算エラーで終了されます。このエラーは、CATCH ブロックで捕捉されます。

## ObjectScript

```
TRY {
  SET x=8
  FOR { SET y=4/x
        WRITE "Running loop 4/",x,"=",y,!
        SET x=x-1
      }
  WRITE "Next command after FOR code block"
}
CATCH exp {
  WRITE !,"this is the exception handler",!
  IF 1=exp.%IsA("%Exception.SystemException") {
    WRITE "Name: ", $ZCVT(exp.Name,"O","HTML"),!
    WRITE "Location: ",exp.Location,!
    WRITE "Code: ",exp.Code
  }
  ELSE {WRITE "Unexpected exception type",! }
  RETURN
}
```

## 引数付きの FOR

FOR が実行する動作は、使用する引数形式によって異なります。

FOR var=expr は、code ブロックを 1 回実行し、var を expr の値に設定します。FOR var=expr1,expr2...,exprN は、code ブロックを N 回実行し、ループごとに var を連続する expr のそれぞれの値に設定します。

FOR var=start:increment は、code ブロックを終了するまで無限に実行します。最初の反復で、InterSystems IRIS は var を start の値に設定します。FOR コマンドが実行されるたびに、指定された increment の値で var の値がインクリメントされます。実行は、code ブロック内で QUIT、RETURN、または GOTO コマンドに遭遇するまで繰り返します。

FOR var=start:increment:end は、var を start の値に設定します。その後 InterSystems IRIS は、以下のテーブルに記述されている条件に基づいて code ブロックを実行します。

increment が正の場合	increment が負の場合
start > end の場合は code ブロックを実行しません。var が end 以上の場合、または InterSystems IRIS が QUIT、RETURN、または GOTO コマンドに遭遇した場合にループが終了します。	start < end の場合は、code ブロックを実行しません。var が end 以下の場合、または InterSystems IRIS が QUIT、RETURN、または GOTO コマンドに遭遇した場合にループが終了します。

ループの実行を開始するとき、InterSystems IRIS は start、increment、end の各値を評価します。ループ内でこれらの値が変更されてもそれは無視され、ループ実行回数に影響を与えません。

ループの終了時、var にはループの最後の実行結果によるインクリメントを反映する値が格納されています。ただし、var が end で指定された最大値を超えることはありません。

FOR ループには、複数のコンマ区切り forparameter 引数を含めることができます。しかし、var 引数は 1 つしか含めることができません。有効な構文は、以下のとおりです。

```
FOR var=start1:increment1:end1,start2:increment2:end2
```

forparameter 引数の内部、またはコンマで区切った各 forparameter 引数の間で改行を使用できます。code の区切り文字 { } の前後、および code ブロック内で改行を使用できます。

forparameter 引数内では空白スペースを使用できますが、必須ではありません。したがって、FOR i=1:1:10{code} と FOR i = 1 : 1 : 10 { code } は等しく有効な構文です。

## 引数

### var

var 引数には、単純なローカル変数、添え字付きローカル変数、または**インスタンス変数** (i%property など) を指定できます。FOR コマンドは、この変数を初期化 (または設定) するため、FOR コマンドの前に var を定義する必要はありません。

- ループ・カウンタ構文の使用時には、var は、FOR ループの現在のカウンタ値を保持するローカル変数になります。初期値として、start で指定された数値が格納されます。その後、FOR ループの反復ごとに increment を使用して値が再計算されます。
- var=expr 構文の使用時には、ローカル変数は expr 値に初期化されます。

### expr

ループ・コマンドを実行する前に InterSystems IRIS が var に割り当てる値です。expr の値は、リテラルまたは有効な式として指定できます。expr は単一の値、または値のコンマ区切りリストになります。単一の値の場合、InterSystems IRIS は、FOR ループを 1 回実行し、この var 値をコード・ブロックに提供します。値のコンマ区切りリストの場合、InterSystems IRIS は、値の数と同じ回数分 FOR ループを実行し、各ループで値を var に代入して、この var 値をコード・ブロックに提供します。

### start

FOR ループの最初の反復時に var に割り当てられる数値。start 値は、リテラルあるいは有効な式として指定できます。InterSystems IRIS は start をその数値に対して評価します。数値以外の start 値は 0 として評価されます。

### increment

FOR ループの各反復後に、var をインクリメントするのに使用される数値。start を指定している場合、increment は必須です。increment 値は、リテラルあるいは有効な式として指定できます。InterSystems IRIS は increment をその数値に対して評価します。increment には、整数または小数を指定できます。また、正の数 (インクリメント) または負の数 (デクリメント) のどちらでもかまいません。

increment の値が 0 または非数値の場合、FOR ループは、終了するまで無限に繰り返されることになります。これは、end=0 の場合にも当てはまります。ただし、start が end より大きいときには、ループは実行されず、0 の increment は無視されます。

### end

FOR ループを終了するのに使用される数値。var が、この値以上になると、FOR ループは、最後の 1 回を実行してから終了します。end 値は、リテラルあるいは有効な式として指定できます。InterSystems IRIS は end をその数値に対して評価します。

start = end の場合、FOR ループは 1 回だけ実行されます。start が end より大きく、increment が正の整数の場合は、FOR ループは実行されません。

### code

中括弧で囲まれた、1 つ以上の ObjectScript コマンドのブロックです。この実行可能なコード・ブロックには、複数のコマンド、ラベル、コメント、改行、インデント、および空白スペースを、必要に応じて含めることができます。FOR コマンドが終了すると、閉じ中括弧の後の次のコマンドが続けて実行されます。

開き中括弧や閉じ中括弧は、1 行で独立して記述するか、コマンドと同じ行に記述できます。開き中括弧や閉じ中括弧は、列 1 に記述してもかまいませんが、お勧めはできません。推奨されるプログラミング手法として、入れ子になったコード・ブロックの開始と終了を示すために、中括弧はインデントするようにしてください。開き中括弧の前後に空白を入れる必要はありません。閉じ中括弧の前に空白を入れる必要はありません。引数のないコマンドに続く中括弧の場合も同様です。中括弧の空白に関する唯一の要件は、閉じ中括弧とその後のコマンドを、スペース、タブ、または改行で区切る必要があるということです。

## FOR ループの終了

FOR ループは、QUIT、RETURN、CONTINUE、または GOTO を発行すると終了できます。

- **QUIT** は、現在の FOR ブロック構造を終了します。そのため、FOR ブロック内の QUIT では、InterSystems IRIS は FOR ブロックの後にある次の行で実行を開始します。QUIT は、現在の FOR ブロックを終了するだけです。FOR ブロックが別の FOR ブロック内（または他のブロック構造内）で入れ子になっている場合は、QUIT は内部の FOR ブロックを終了して外側のブロック構造に移動します。

FOR ブロック（および他のブロック構造）内の QUIT の動作は、ブロック構造内にない場合の QUIT の動作とは異なります。これらのブロック構造のいずれかの外側にある QUIT は、現在のコード・ブロックではなく、現在のルーチンを終了します。詳細については、コマンドのリファレンス・ページの **"QUIT"** コマンドを参照してください。

QUIT は、QUIT が FOR ブロック内にある場合に限り FOR ブロックを終了します。FOR ループがサブルーチンを呼び出している場合は、サブルーチン内で QUIT を発行すると、呼び出した FOR ループではなくサブルーチンを終了します。

- **RETURN** は、FOR ブロック構造内から発行されたかどうかには関係なく、現在のルーチンを終了します。
- **CONTINUE** は、現在の FOR ループを終了します。これにより、実行が即座に FOR コマンドに戻ります。次に FOR コマンドは引数をインクリメントして評価し、その評価を基にしてコード・ブロック・ループを再実行するかどうかを決定します。したがって、CONTINUE コマンドの実行にはコード・ブロックの終了中括弧に到達するのと同じ効果があります。
- **GOTO** は、制御を FOR コード・ブロックの外側に移動することで、現在の FOR ブロック構造を終了できます。FOR ループは、FOR コード・ブロック内に制御を渡す GOTO では終了されません。

## FOR ループまたは WHILE ループ

FOR または **WHILE** のいずれかを使用して、同じ操作を実行できます。つまり、イベントによって実行がループを抜けるまでループすることができます。ただし、どちらのループ構造を使用するかにより、コード・モジュールに対してシングル・ステップ (BREAK "S+" または BREAK "L+") デバッグの実行に影響があります。

FOR ループでは、スタックに新しいレベルがプッシュされます。WHILE ループでは、スタック・レベルは変更されません。FOR ループをデバッグする場合、FOR ループ内からスタックをポップすると (BREAK "C" GOTO または QUIT 1 を使用)、FOR コマンド構文の終了直後から、このコマンドでのシングル・ステップ・デバッグを続行できます。WHILE ループをデバッグする場合は、BREAK "C" GOTO または QUIT 1 を使用して発行してもスタックはポップされません。したがって、WHILE コマンドの終了後にシングル・ステップ・デバッグは続行されません。残りのコードはブレークなしで実行されます。

詳細は、**"BREAK"** コマンドおよび **"BREAK コマンドによるデバッグ"** を参照してください。

## 例

### 引数なしのFOR

以下の例では、引数なしの FOR を示しています。ユーザは、後の DO コマンドで Calc サブルーチンへ渡す数の入力を繰り返し求められます。FOR ループは、ユーザが NULL 文字列を入力する（数値を入力せずに [Enter] を押す）と QUIT コマンドを実行し、終了します。

## ObjectScript

```
Mainloop
  FOR {
    READ !, "Number: ", num
    QUIT: num = " "
    DO Calc(num)
  }
Calc(a)
  WRITE !, "The number squared is ", a*a
  QUIT
```

## FOR var=expr の使用

var=expr を指定すると、InterSystems IRIS は、expr のコンマで区切られた値の数と同じ回数分 FOR ループを実行します。expr の値は、リテラルまたは有効な式です。式を指定する場合、その式は単一の値に評価される必要があります。

以下の例では、FOR コマンドでコード・ブロックを 1 回実行します。num の値には 4 が設定されています。これにより、数値の 12 を書き出します。

## ObjectScript

```
Loop
  SET val=4
  FOR num=val {
    WRITE num*3,!
  }
  WRITE "Next command after FOR code block"
```

以下の例では、FOR コマンドでコード・ブロックを 1 回実行します。alpha(7) の値には “abcdefg” が設定されています。

## ObjectScript

```
Loop
  SET val="abc"
  FOR alpha(7)=val_"defg" {
    WRITE alpha(7),!
  }
  WRITE "Next command after FOR code block"
```

以下の例では、FOR コマンドでコード・ブロックを 8 回実行します。連続する各完全数がコード・ブロックに提供されます。

## ObjectScript

```
FOR pnum=6,28,496,8128,33550336,8589869056,137438691328,2305843008139952128 {
  WRITE "Perfect number ", pnum
  SET rp=$REVERSE(pnum)
  IF 54=$ASCII(rp,1) {
    WRITE " ends in 6",! }
  ELSEIF 56=$ASCII(rp,1),50=$ASCII(rp,2) {
    WRITE " ends in 28",! }
  ELSE {WRITE " is something unknown to mathematics",! }
}
```

## var=start:increment:end の使用

引数 start、increment、end は、それぞれ初期値、増分値、終値を指定します。これら 3 つの値は、数値として評価されます。これは整数または実数、正または負の値です。文字列値を提供する場合は、ループの開始時に同等の数値に変換されます。

初めて InterSystems IRIS がループに入ると、start 値を var に代入し、var 値と end 値を比較します。var 値が end 値より小さい場合（または increment 値が負のときは end 値より大きい場合）、InterSystems IRIS はループ・コマンドを実行します。その後 increment 値を使用して、var 値を更新します（負の increment が使用されている場合、var の値はデクリメントされます。）

ループの実行は、var の値が end の値を上回るまで（または InterSystems IRIS が QUIT、RETURN、または GOTO に遭遇するまで）続けられます。その時点で、var が end を超えないようにするために、InterSystems IRIS が変数の割り当

てを抑制して、ループの実行が終了します。インクリメントによって var 値が end 値と等しくなった場合、InterSystems IRIS は FOR ループを最後に 1 回実行した後、ループを終了します。

以下のコードは、最後の文字を除く string1 のすべての文字に WRITE コマンドの実行を繰り返し、順番に出力します。end 値は len-1 として指定されているので、最後の文字は出力されません。これは、テストがループのトップで実行され、そのループは変数値 (index) が end 値 (len-1) を (同等ではなく) 超えるときに終了するからです。

### ObjectScript

```
Stringwriteloop
SET string1="123 Primrose Path"
SET len=$LENGTH(string1)
FOR index=1:1:len-1 {
    WRITE $EXTRACT(string1,index)
}
```

## FOR var=start:increment の使用

この FOR コマンドの形式には、end 値がありません。ループを終了させるための QUIT、RETURN、または GOTO コマンドを含む必要があります。

start と increment の値は、数値として評価されます。これは整数または実数、正または負の値です。文字列値の場合は、ループの開始時に同等の数値に変換されます。ループの実行を開始するとき、InterSystems IRIS は start および increment の値を評価します。ループ内でこれらの値が変更されても、それは無視されます。

ループの開始時に start 値を var に代入し、ループ・コマンドを実行します。その後 increment 値を使用して、var 値を更新します(負の increment が使用されている場合、var の値はデクリメントされます。)InterSystems IRIS は、ループ内で QUIT、RETURN、または GOTO に遭遇するまでループの実行を繰り返します。

以下の例では、start:increment 構文を使用して、桁数が 3 桁未満の 7 の倍数をすべて返します。

### ObjectScript

```
FOR i(1)=0:7 {
    QUIT:$LENGTH(i(1))=3
    WRITE "multiple of 7 = ",i(1),! }
```

以下の例では、start:increment 構文を使用して、ユーザが提供した一連の数値の平均を計算します。ユーザが NULL 文字列を入力したとき(つまり、数値を入力せずに [Enter] を押したとき)にループを終了するため、後置条件 QUIT が含まれています。後置条件式 (num="") が TRUE と評価されると、InterSystems IRIS は QUIT を実行し、ループを終了します。

ループ・カウンタ(変数 i)は、入力された数字の数を記録するために使用されます。i は、0 に初期化されます。これは、ユーザの数値入力後にカウンタがインクリメントされるためです。ユーザが NULL を入力すると、InterSystems IRIS はループを終了します。ループが終了された後、平均を計算するため、SET コマンドは(ローカル変数として)iを参照します。

### ObjectScript

```
Averageloop
SET sum=0
FOR i=0:1 {
    READ !,"Number: ",num
    QUIT:num=""
    SET sum=sum+num
}
SET average=sum/i
```

## 複数の forparameter を持つ FOR を使用する

FOR コマンドには、1 つの var = 引数しか含めることができませんが、forparameter 引数は、コンマ区切りリストとして指定することで複数含めることができます。例えば、構文 var=expr1,expr2,expr3 により、code ブロックは、実行ごとに異なる var の値で 3 回実行されます。

これらの forparameter 引数は、厳密に左から右の順序で評価され実行されます。そのため、1 つの forparameter でエラーが発生しても、それより前の forparameter の実行が妨げられることはありません。

1 つの FOR コマンドには、expr 構文と start:increment:end 構文の両方のタイプのパラメータ構文を含めることができます。

以下の例では、expr 構文と start:increment:end 構文を組み合わせています。2 つの forparameter はコンマで区切られています。初めて FOR を使用するとき、InterSystems IRIS は expr 構文を使用し、y の値と等しい x を使用して Test サブルーチンを呼び出します。2 度目（およびそれ以降）の反復では、InterSystems IRIS は start:increment:end 構文を使用します。x を 1 に、その次は 2 を設定し、最後の繰り返しで x=10 となります。

### ObjectScript

```
Mainloop
  SET y="beta"
  FOR x=y,1:1:10 {
    DO Test
  }
QUIT
Test
  WRITE !,"Running test number ",x
QUIT
```

以下に示すのは、3 つの forparameter 引数と、start:increment:end 構文が含まれているプログラム例です。これは i に 1 を設定し、1 から 10 までは、1 桁の数値を 1 単位でインクリメントします。2 番目の forparameter は i の値 10 を取り、100 までは 10 単位でインクリメントします。3 番目の forparameter は i の値 100 を取り、1000 までは 100 単位でインクリメントします。この例では、10 と 100 の値を繰り返しています。

### ObjectScript

```
FOR i=1:1:10,i:10:100,i:100:1000 {WRITE i,!}
```

以下の例は、前述の例と同じ操作を実行していますが、10 と 100 の値は繰り返していません。

### ObjectScript

```
FOR i=1:1:9,i+1:10:99,i+10:100:1000 {WRITE i,!}
```

## 引数なしの FOR のインクリメント

引数なしの FOR は FOR var=start:increment 形式と同じように動作します。唯一の違いは、ループ実行回数を記録する方法を提供していないことです。

以下の例は、前述のループ・カウンタ例を引数なしの FOR を使用して、どのように書き直しているかを表しています。i=i+1 がループ・カウンタの代わりです。

### ObjectScript

```
Average2loop
  SET sum=0
  SET i=0
  FOR {
    READ !,"Number: ",num QUIT:num=""
    SET sum=sum+num,i=i+1
  }
  SET average=sum/i
  WRITE !!,"Average is: ",average
QUIT
```

## FOR および NEW

NEW コマンドは、var に影響を与える可能性があります。引数なしの NEW コマンドまたは排他的 NEW コマンド（特に var を除外しないもの）を FOR ループ本文内で発行すると、新しいフレーム・コンテキストで var が定義されない場合があります。



以下の例のように、var を含まない NEW コマンドは、FOR ループの実行に影響を与えません。

### ObjectScript

```
SET a=1,b=1,c=8
FOR i=a:b:c {
    WRITE !,"count is ",i
    NEW a,c
    WRITE " loop"
    NEW (i)
    WRITE " again"
}
```

## FOR および監視ポイント

FOR と組み合わせた監視ポイントの使用には制限があります。FOR コマンドの制御 (インデックス) 変数に対する監視ポイントを確立する場合、InterSystems IRIS は、各 FOR コマンド引数を最初に評価するときだけ、指定された監視ポイント・アクションをトリガします。この制約は、パフォーマンスの向上が目的です。

以下の例には、監視される変数 x に対して、3 種類の FOR コマンド引数が含まれています。1 つは初期値、増分値、制限 (最終値) を持つ値域、もう 1 つは単一の値、そしてもう 1 つは初期値、増分値を持ち、制限を持たない値域です。x の初期値が 1、20、50 のときに、ブレイクが実行されます。

```
USER>ZBREAK *x
USER>FOR x=1:1:10,20,50:2 {SET t=x QUIT:x>69}
<BREAK>
USER 2f0>WRITE
x=1
USER 2f0>g
USER>FOR x=1:1:10,20,50:2 {SET t=x QUIT:x>69}
<BREAK>
USER 2f0>WRITE
t=10
x=20
USER> 2f0>g
USER>FOR x=1:1:10,20,50:2 {SET t=x QUIT:x>69}
<BREAK>
USER 2f0>WRITE
t=20
x=50
USER 2f0>g
USER>WRITE
t=70
x=70
```

## 関連項目

- ・ [DO WHILE コマンド](#)
- ・ [WHILE コマンド](#)
- ・ [IF コマンド](#)
- ・ [CONTINUE コマンド](#)
- ・ [DO コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [RETURN コマンド](#)

## GOTO (ObjectScript)

制御を移動します。

### 構文

```
GOTO:pc
GOTO:pc goargument,...

G:pc
G:pc goargument,...
```

goargument には、以下を指定できます。

```
location:pc
```

### 引数

引数	説明
pc	オプション - 後置条件式
location	オプション - 制御の移動先

### 説明

GOTO コマンドには、以下の 2 つの形式があります。

- ・ [引数なし](#)
- ・ [引数付き](#)

### 引数なしの GOTO

引数なしの GOTO は、InterSystems IRIS が現在実行中のコードでエラーまたは [BREAK](#) コマンドに遭遇した後に、通常のプログラムの実行を再開します。引数なしの GOTO は、ターミナル・プロンプトでのみ使用できます。

以下の例は、引数なしの GOTO の使用法を示しています。この例では、2 つ目の WRITE は、〈BREAK〉エラーが原因で実行されません。GOTO を発行すると実行が再開され、2 つ目の WRITE が実行されます。

```
USER>WRITE "before" BREAK WRITE "after"
before
WRITE "before" BREAK WRITE "after"
                        ^
<BREAK>
USER 1S0>GOTO
after
USER>
```

BREAK コマンドの後には、2 つのスペースを配置する必要があることに注意してください。

引数を持たない GOTO の発行時に NEW コマンドが有効な場合、InterSystems IRIS は〈COMMAND〉エラーを発行し、新しいコンテキストが維持されます。NEW の後に再開するには、QUIT 1 コマンドを実行してから、引数を持たない GOTO を実行してください。

ターミナル・プロンプトで引数なしの GOTO を使用してもエラー後に実行を継続できます。["ターミナル・プロンプトでのエラー処理"](#) を参照してください。

### 引数付きの GOTO

引数 location 付きの GOTO は、指定した場所に制御を移動します。後置条件式をコマンドまたは引数のいずれかで指定した場合、InterSystems IRIS は、後置条件式が True (ゼロ以外) の場合だけ、制御を移します。



ターミナル・プロンプトから GOTO location を使用して、異なる場所で中断されているプログラムを再開できます。

GOTO コマンドの引数として \$CASE 関数を指定できます。

## 引数

### pc

コマンドを条件付きにする、オプションの後置条件式です。後置条件式が GOTO コマンド・キーワードに追加されている場合、InterSystems IRIS では後置条件式が True (0 以外の数値に評価される) の場合に GOTO コマンドを実行します。InterSystems IRIS では、後置条件式が False (0 に評価される) の場合、GOTO コマンドを実行しません。後置条件式が引数に追加されている場合、InterSystems IRIS では後置条件式が True (0 以外の数値に評価される) の場合に引数を実行します。False (0 に評価される) の場合、InterSystems IRIS はその引数をスキップし、次の引数 (存在する場合)、または次のコマンドを評価します。詳細は、“[コマンド後置条件式](#)” を参照してください。

### location

制御の移動先。ルーチン・コード内で GOTO を使用する場合は必須です。ターミナル・プロンプトから実行する場合はオプションです。location は、単一の値、またはコンマで区切られた値のリストとして (後置条件付きで) 指定できます。以下のいずれかの形式を使用できます。

label+offset は、[現在のルーチン](#)にある[行ラベル](#)を指定します。+offset は負でない整数です。オプションのオフセットでは、ラベルの何行後からサブルーチンの実行を開始するかを指定します。offset は、ラベル行を含むコード行、およびコメント行をカウントします。offset は、空白行やコメント内の空白行はカウントしません。ただし、offset は、[埋め込み SQL](#) の行をすべてカウントします。この場合は、すべての空白行も含まれます。

label+offset routine は、ディスクに保存されている指定したルーチン内の[行ラベル](#)を指定します。InterSystems IRIS は、ディスクからそのルーチンをロードし、指定されたラベルから実行を続けます。+offset は負でない整数です。オプションのオフセットでは、ラベルの何行後からサブルーチンの実行を開始するかを指定します。

^routine は、ディスクに保存されているルーチンを指定します。InterSystems IRIS は、ディスクからルーチンをロードし、ルーチン内の最初の実行可能コード行から実行を続けます。ルーチンが変更されている場合、GOTO がそのルーチンを呼び出すときに、InterSystems IRIS は更新バージョンのルーチンをロードします。[DO](#) コマンドとは異なり、GOTO はルーチンの実行後に呼び出し元のプログラムへは返りません。存在しないルーチンを指定した場合、InterSystems IRIS は <NOROUTINE> エラー・メッセージを返します。詳細は、“[\\$ZERROR](#)” 特殊変数を参照してください。

注釈 GOTO は、[拡張ルーチン参照](#)をサポートしません。別のネームスペースでルーチンを実行するには、[DO](#) コマンドを使用してください。

上記のいずれかの形式を含む変数として、location を参照することもできます。ただし、この場合は名前による間接指定を使用する必要があります。location には、仮パラメータ・リスト、または外部関数やプロシージャの名前でユーザ定義されているサブルーチン・ラベルを指定することはできません。存在しない label を指定した場合、InterSystems IRIS は <NOLINE> エラー・メッセージを返します。詳細は、“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

IRISSYS % ルーチンの呼び出し時における offset の指定はできません。指定しようとすると、InterSystems IRIS は <NOLINE> エラーを発行します。

## 例

以下の例では、ユーザが入力した age の値に従って、GOTO コマンドが 3 つの場所のいずれかに実行を移動させます。場所は、変数 loc に格納され、名前による間接指定 (@loc) で参照されるサブルーチン・ラベルです。

### ObjectScript

```
mainloop
  SET age=""
  READ !,"What is your age? ",age QUIT:age=""
  IF age<30 {
    SET loc="Young" }
  ELSEIF (age>29)&(age<60) {
```

```
    SET loc="Midage" }
ELSEIF age>59 {
    SET loc="Elder" }
ELSE {
    WRITE "data input error"
    QUIT }
GOTO @loc
Quit
Young
    WRITE !,"You're still young"
    QUIT
Midage
    WRITE !,"You're in your prime"
    QUIT
Elder
    WRITE !,"You have a lifetime of wisdom to impart"
    QUIT
```

このタイプの名前による間接指定を使用した GOTO は、プロシージャ・ブロック内では許可されていません。

上記の例の代わりに、IF コマンドを省略し、以下のように、引数に後置条件を使用して、コンマで区切られたリストを持つ GOTO のコードを記述することもできます。

### ObjectScript

```
GOTO Young:age<30,Midage:(age>29)&(age<60),Elder:age>59
```

同じ例を、適切なサブルーチンの場所を呼び出す DO コマンドを使用してコードを記述することもできます。ただしこの場合は、InterSystems IRIS が QUIT に遭遇したとき、DO の後のコマンドに制御が返されます。

以下の例は、offset がコード行をカウントする様子を示しています。これは、間に挟まれたラベル行とコメント行をカウントしますが、空白行はカウントしません。

### ObjectScript

```
Main
    GOTO Branch+7
    QUIT
Branch
    WRITE "Line 1",!
SubBranch
    WRITE "Line 3",!
    /* comment line */
    WRITE "Line 5",!

    WRITE "Line 6",!
    WRITE "Line 7",!
    WRITE "Line 8",!
    QUIT
```

## GOTO および QUIT

DO コマンドとは異なり、GOTO は制御を無条件で移動させます。InterSystems IRIS が DO によって呼び出されたサブルーチンで QUIT に遭遇すると、InterSystems IRIS は最後に実行した DO に続くコマンドに制御を渡します。

InterSystems IRIS が GOTO による移動の後に QUIT に遭遇しても、GOTO に続くコマンドには制御は返されません。それ以前に DO があった場合は、最後の DO に続くコマンドに制御が返されます。前に DO がなかった場合は、ターミナルに戻ります。

以下のコード・シーケンスでは、C 内の QUIT が、A 内の DO コマンドに続く WRITE コマンドに制御を返します。

## ObjectScript

```
testgoto
A
  WRITE !,"running A"
  DO B
  WRITE !,"back to A, all done"
  QUIT
B
  WRITE !,"running B"
  GOTO C
  WRITE !,"this line in B should never execute"
  QUIT
C
  WRITE !,"running C"
  QUIT
```

## コード・ブロックで GOTO を使用する

GOTO はコード・ブロックの終了に使用されますが、コード・ブロックの開始には使用できません。

FOR、IF、DO WHILE、または WHILE ループ内で GOTO を使用する場合、すべてのコード・ブロックの外の location、あるいは現在のコード・ブロック内の location に移動するか、もしくは入れ子にされたコード・ブロックからそれを囲むコード・ブロック内の location に移動できます。独立したコード・ブロック、あるいは現在のコード・ブロック内に入れ子になったコード・ブロックのどちらの場合でも、コード・ブロックから他のコード・ブロック内の location への移動はできません。例として、それぞれのコマンドを参照してください。

コード・ブロックの外部にある location に対する GOTO は、ループを終了します。コード・ブロックの内部にある location に対する GOTO は、ループを終了しません。入れ子にされたコード・ブロックから、それを囲むコード・ブロックへの GOTO は、内部（入れ子にされた）ループを終了しますが、外部のループは終了しません。

GOTO は TRY または CATCH コード・ブロックの終了に使用されますが、これらのいずれかのコード・ブロックの開始には使用できません。また、GOTO は、TRY または CATCH キーワードと同じ行のラベルに指定できません。指定しようとすると、<NOLINE> エラーが返されます。

## GOTO の制限

以下の GOTO 操作は許可されません。

- ・ GOTO は、プロシージャに入る、またはプロシージャを終了するために使用できません。
- ・ プロシージャ・ブロック内では、名前による間接指定での GOTO (GOTO @name) を使用することはできません。

## 関連項目

- ・ [DO コマンド](#)
- ・ [FOR コマンド](#)
- ・ [IF コマンド](#)
- ・ [DO WHILE コマンド](#)
- ・ [WHILE コマンド](#)
- ・ [BREAK コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [\\$CASE 関数](#)
- ・ [ターミナル・プロンプトでのエラー処理](#)
- ・ [BREAK コマンドによるデバッグ](#)

## HALT (ObjectScript)

現在のプロセスの実行を終了します。

### 構文

```
HALT:pc
H:pc
```

### 引数

引数	説明
pc	オプション - 後置条件式

### 概要

HALT コマンドは、現在のプロセスの実行を終了します。\$HALT 特殊変数が現在のコンテキスト内（または前のコンテキスト内）に定義されている場合、HALT コマンドを発行すると、現在のプロセスが終了されるのではなく、\$HALT で指定されている停止トラップ・ルーチンが呼び出されます。通常、停止トラップ・ルーチンは削除処理またはレポート処理を実行してから、2 番目の HALT コマンドを発行して実行を終了します。

HALT の振る舞いは、ルーチン・コードの実行により検出される場合も、ターミナル・プロンプトから入力される場合も同じです。どちらの場合も、現在のプロセスを終了します。

HALT の最小省略形は HANG コマンドと同じですが、HANG は、必須の引数 hangtime を取るため区別できます。

### HALT の効果

HALT が処理を終了すると、システムは自動的にこのプロセスのすべてのロックを解除し、このプロセスが所有していたデバイスをすべて閉じます。これにより、終了したプロセスがロックされた変数や未解放のデバイスを残さないようにします。

HALT がプロセスを終了するときに進行中のトランザクションがある場合、トランザクションの解決はプロセス・タイプによって異なります。バックグラウンド・ジョブ（非インタラクティブ・プロセス）中の HALT は、進行中のトランザクションを必ずロール・バックします。インタラクティブ・プロセス（ルーチンを実行するためにターミナルを使用するなど）中の HALT は、進行中のトランザクションの解決を求めるプロンプトを表示します。このプロンプトは、以下のとおりです。

```
You have an open transaction.
Do you want to perform a (C)ommit or (R)ollback? R=>
```

現在のトランザクションをコミットするには、“C” を指定します。現在のトランザクションをロール・バックするには、“R” を指定します（または単に Enter キーを押します）。

### 停止トラップ

HALT コマンドの実行は、停止トラップによって割り込まれます。停止トラップは、HALT 特殊変数を使用して作成します。

停止トラップが現在のコンテキスト内に作成されている場合、HALT コマンドにより、\$HALT で指定される停止トラップを実行します。HALT コマンド自身は実行されません。

停止トラップが低いレベルのコンテキスト・フレームにある場合、HALT コマンドは停止トラップを持つコンテキスト・フレームに到達するまで、フレーム・スタックからコンテキスト・フレームを削除します。その後、HALT は \$HALT で指定される停止トラップ・ルーチンを呼び出し、実行を中止します。

## 引数

### pc

コマンドを条件付きにする、オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に HALT コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

## 例

以下の例では、HALT によって、ユーザは現在のアプリケーションを終了し、オペレーティング・システムに戻ることができます。システムでは、ユーザが必要とするクリーンアップがすべて実行されます。コマンドは後置条件式を使用していることに注意してください。

### ObjectScript

```
Main
  READ !,"Do you really want to stop (Y or N)? ",ans QUIT:ans=""
  HALT:(ans["Y"]!(ans="y"))
  DO Start
Start()
  WRITE !,"This is the Start routine"
  QUIT
```

以下の例では、HALT は \$HALT で指定されている停止トラップ・ルーチンを呼び出します。この場合、実際に実行を停止するのは、2 番目の HALT コマンドです(デモンストレーションのために、この例では、表示された出力を参照する時間を取れるように HANG 文を使用します)。

### ObjectScript

```
Main
  NEW $ESTACK
  SET $HALT="OnHalt"
  WRITE !,"Main $ESTACK= ",$ESTACK // 0
  HANG 2
  DO SubA
  WRITE !,"this should never display"
SubA()
  WRITE !,"SubA $ESTACK= ",$ESTACK // 1
  HANG 2
  HALT // invoke the OnHalt routine
  WRITE !,"this should never display"
OnHalt()
  WRITE !,"OnHalt $ESTACK= ",$ESTACK // 0
  HANG 2
  // clean-up and reporting operations
  HALT // actually halt the current process
```

## \$SYSTEM.Process.Terminate()

現在のプロセスを停止するか、または他の実行中のプロセスを停止するには、\$SYSTEM.Process.Terminate() メソッドを使用できます。

以下の例は、現在のプロセスを停止します。

### ObjectScript

```
DO $SYSTEM.Process.Terminate()
```

以下の例は、PID 7732 のプロセスを停止します。

### ObjectScript

```
DO $SYSTEM.Process.Terminate(7732)
```

Terminate() メソッドの効果は、現在のプロセスに対しては HALT コマンドと同じで、その他のプロセスに対しては ^RESJOB ユーティリティと同じです。

## ^RESJOB および ^JOBEXAM

HALT コマンドは、現在のプロセスを停止するために使用します。

^RESJOB または ^JOBEXAM ユーティリティは、その他の実行中のプロセスを停止するのに使用できます。これらのユーティリティは、現在のプロセスを停止するのには使用できません。これらは、現在のプロセスを含めて、すべての実行中のプロセスに関する情報を表示するのに使用できます。

これらのユーティリティは、%SYS ネームスペースから呼び出す必要があります。これらのユーティリティを呼び出すには、適切な特権を持っている必要があります。ユーティリティ名では、大文字と小文字が区別されます。

- ・ プロセス ID (PID) がわかる場合、^RESJOB を使用することで、プロセスを直接停止できます。? オプションを使用することで、すべての実行中のプロセスのリストを表示できます。
- ・ ^JOBEXAM を使用すると、すべての実行中のプロセスのリストがまず表示されます。その後、停止 (終了)、中断、または再開するプロセスを指定できます。View ^JOBEXAM を使用しても、すべての実行中のプロセスのリストを表示できますが、プロセスを停止、中断、および再開するオプションはありません。

以下は、ターミナルからの ^RESJOB の呼び出し例です。

### Terminal

```
%SYS>DO ^RESJOB
Force a process to quit InterSystems IRIS
Process ID (? for status report): 7732
Process ID (? for status report):
%SYS>
```

プロンプトで、停止するプロセスのプロセス ID (PID) を入力します。^RESJOB はプロセスを停止し、次のプロセス ID の入力を求めるプロンプトを表示します。プロセス ID の入力が終了したら、プロンプトで Enter キーを押します。プロンプトで ? を指定すると、現在実行中のプロセスのリストを表示できます。

- ・ 現在のプロセス: 現在のプロセスを ^RESJOB を使用して停止しようとする、失敗して次のメッセージが表示されます。[ KILL ] ^RESJOB は、別のプロセス ID の入力を求めるプロンプトを表示します。
- ・ 実行していないプロセス: 実行していないプロセスの ID を指定すると、失敗して次のメッセージが表示されます。[ InterSystems IRIS ] ^RESJOB は、別のプロセス ID の入力を求めるプロンプトを表示します。
- ・ システム・プロセス: 特定のシステム・プロセスを停止するために ^RESJOB を使用することはできません。使用を試みると、失敗して次のメッセージが表示されます。[ name KILL ] ^RESJOB は、別のプロセス ID の入力を求めるプロンプトを表示します。
- ・ 進行中のトランザクション: ^RESJOB を使用して進行中のトランザクションがあるプロセスを停止することは、そのプロセス内で HALT コマンドを発行することと同じです。非インタラクティブ・プロセスは未完了のトランザクションをロール・バックします。インタラクティブ・プロセスは、そのターミナル・プロンプトで、未完了のトランザクションのコミットまたはロール・バックを求めるプロンプトを表示します。

## 関連項目

- ・ [\\$HALT](#) 特殊変数
- ・ [コマンド行ルーチンのデバッグ](#)

# HANG (ObjectScript)

指定された秒数だけ実行を中断します。

## 構文

```
HANG:pc hangtime,...
H:pc hangtime,...
```

## 引数

引数	説明
pc	オプション - 後置条件式
hangtime	待機時間 (秒) 正の数値に解決される式、または正の数値に解決されるコンマで区切られた数式のリスト。

## 説明

HANG は、ルーチンの実行を指定した時間 (秒数) だけ中断します。複数の引数がある場合、InterSystems IRIS は提示される順序で各引数の継続時間にわたって実行を中断します。HANG の時間はシステム時計を使用して計算されます。これにより、その精度が決定されます。

HANG の最小省略形 (H) は HALT コマンドと同じです。HANG は、必須の引数 hangtime を取るため区別できます。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に HANG コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

### hangtime

待機時間 (秒) この時間は、数値式で表されます。hangtime は、整数秒を指定するために整数として指定したり、小数秒を指定するために小数として指定したりできます。指数 (\*\*)、数値式、または他の数値演算子を使用することもできます。

hangtime を 0 (ゼロ) に設定すると、待機時間はありません。hangtime に負の数または非数値を指定した場合、0 に設定する場合と同じ結果となります。

複数の hangtime 引数を、数値式のコンマ区切りのリストとして指定することもできます。InterSystems IRIS は、提示される順序で各引数の継続時間にわたって実行を中断します。ここでも負の数は 0 と同様に処理されます。したがって、hangtime に 16,-15 を指定すると、待機時間は 16 秒となります。

以下の例で示すように、各 hangtime 引数が別々に実行される場合、hang 計算で現在の時刻を使用する処理に影響が及ぶ可能性があります。

### ObjectScript

```
SET start=$ZHOROLOG
SET a=$ZHOROLOG+5
HANG 4,a-$ZHOROLOG
SET end=$ZHOROLOG
WRITE !,"elapsed hang=",end-start
```

この例では、HANG が最初に実行を 4 秒間一時停止します。次の引数が解析されると、時間は変数が設定されてから 4 秒が経過した時点になっています。したがって、2 回目の一時停止期間はわずか 1 秒になります。HANG は各引数を



順番に実行するため、この例では合計待機時間は(約) 5 秒となります。その他の場合に予期される(約) 9 秒ではありません。

## 例

以下の例では、処理が 10 秒間中断されます。

### ObjectScript

```
WRITE !,$ZTIME($PIECE($HOROLOG,"",2))
HANG 10
WRITE !,$ZTIME($PIECE($HOROLOG,"",2))
```

以下の例では、処理が 1/2 秒間中断されます。\$ZTIMESTAMP は \$HOROLOG とは異なり、\$ZTIME 関数の precision パラメータが指定されている場合は秒の小数部を返すことができます。

### ObjectScript

```
WRITE !,$ZTIME($PIECE($ZTIMESTAMP,"",2),1,2)
HANG .5
WRITE !,$ZTIME($PIECE($ZTIMESTAMP,"",2),1,2)
```

以下のような値を返します。

```
14:34:19.75
14:34:20.25
```

## HANG と時間指定された READ との比較

HANG を使用して、ユーザが出力メッセージを読む間、ルーチンを停止させることができます。しかし、時間指定された READ コマンドを使用すると、このタイプの一時停止をより効果的に扱うことができます。時間指定された READ を使用すると、ユーザは準備ができた時点ですぐに処理を続行できます。しかし、HANG はあらかじめ待機時間が設定されているため、これは不可能です。

## HANG および ^JOBEXAM

HANG コマンドは、現在のプロセスの実行を一時停止するのに使用します。

^JOBEXAM ユーティリティは、その他の実行中のプロセスの実行を中断したり、再開したりするのに使用できます。現在のプロセスの実行を中断するには使用できません。^JOBEXAM を使用して、HANG コマンドによって一時停止されたプロセスの実行を再開することはできません。^JOBEXAM を使用して、HANG コマンドによって一時停止されたプロセスを中断した場合、^JOBEXAM の再開により、中断時に残っていた HANG 時間の部分を完了する必要があるプロセスがアクティブになります。

^JOBEXAM は、すべての実行中のプロセスに関する状態情報を表示します。HANG コマンドによって一時停止されているプロセスは、HANGW 状態として表示されます。^JOBEXAM によって中断されているプロセスは、SUSPW 状態として表示されます。

^JOBEXAM ユーティリティは、%SYS ネームスペースからターミナルで呼び出す必要があります。このユーティリティを呼び出すには、適切な特権を持っている必要があります。ユーティリティ名では、大文字と小文字が区別されます。以下のようにして ^JOBEXAM を実行できます。

- ・ DO ^JOBEXAM : すべての実行中のプロセスのリストを表示します。実行中のプロセスを終了、中断、または再開する文字コード・オプションが用意されています。
- ・ DO View^JOBEXAM : すべての実行中のプロセスのリストを表示します。プロセスを終了、中断、および再開するオプションはありません。



^JOBEXAM をプロセスに使用する場合は、^JOBEXAM を使用する時間を確保するために、HANG コマンドを使用してプロセスを一時停止することが効果的な場合があります。例えば、プロセスを一時停止する箇所でコードに以下を追加します。

```
set ^zMyProcessID = $JOB
while ( $DATA(^zMyDebugGlobal) = 1 )
{
    hang 1
}
```

この方法を使用するには、プロセスを実行する前に ^zMyDebugGlobal グローバルを 1 に設定します。続いて ^JOBEXAM を実行します。^zMyProcessID グローバルがプロセス ID (PID) に設定されることで、^JOBEXAM を実行するときに目的のプロセスを見つけやすくなります。検査したプロセスを再開するには、^zMyDebugGlobal グローバルを強制終了します。

## 関連項目

- ・ [READ](#) コマンド
- ・ [\\$ZTIME](#) 関数
- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数
- ・ [\\$JOB](#) 特殊変数
- ・ [\\$DATA](#) 関数

## IF (ObjectScript)

ブーリアン式を評価し、式の真理値を基にして、実行するコード・ブロックを選択します。

### 構文

```
IF expression1,... { code }
ELSEIF expression2,... { code }
ELSE { code }
```

あるいは

```
I expression1,... { code }
ELSEIF expression2,... { code }
ELSE { code }
```

### 引数

引数	説明
expression1	IF 節のブーリアン・テスト条件。単一の条件、またはコンマで区切られた条件のリスト。
expression2	ELSEIF 節のブーリアン・テスト条件。単一の条件、またはコンマで区切られた条件のリスト。
code	中括弧で囲まれた ObjectScript コマンドのブロックです。

### 概要

ここでは、IF、ELSEIF、および ELSE コマンド・キーワードについて説明します。これらはすべて、IF コマンドの構成要素節と見なされます。IF キーワードは、I と省略できます。他の 2 つのキーワードは省略できません。

1 つの IF コマンドには 1 つの IF 節が含まれ、その後に任意の数の ELSEIF 節、最後に 1 つの ELSE 節と続きます。ELSEIF 節と ELSE 節はオプションですが、ELSE 節は常に指定する方がわかりやすいプログラムになります。

IF コマンドは最初に IF 節の expression1 を評価し、expression1 が True の場合、それに続く中括弧内のコード・ブロックを実行し、IF コマンドを終了します。

expression1 が False の場合、IF 文の次の節に実行を移します。(存在する場合は) 最初の ELSEIF 節を評価します。ELSEIF 節内の expression2 が True の場合、それに続く中括弧内の ELSEIF コード・ブロックを実行し、IF コマンドを終了します。expression2 が False の場合、次の ELSEIF 節 (ある場合) が同様に評価されます。後続の各 ELSEIF 節は、それらの 1 つが True と評価されるまで、あるいはすべてが False と評価されるまでリストの順序でテストされます。

IF 節とすべての ELSEIF 節が False と評価された場合、実行は ELSE 節に移動します。それに続く中括弧内の ELSE コード・ブロックを実行し、IF コマンドを終了します。ELSE 節が省略されている場合、IF コマンドを終了します。

IF はブロック型のコマンドです。各コマンド・キーワードの後には、中括弧 ({} ) で囲まれたコード・ブロックが続きます。IF 節、ELSEIF 節、および ELSE 節は余白 (改行、インデント、空白スペース) を自由に使用できます。ただし、各 IF キーワードと ELSEIF キーワードはそれぞれ、ブーリアン・テスト式の最初の文字と同じ行に置き、1 つの空白スペースで区切る必要があります。ブーリアン・テスト式は複数行に記述でき、また複数の空白スペースを含むことができます。

開き中括弧や閉じ中括弧は、1 行で独立して記述するか、コマンドと同じ行に記述できます。開き中括弧や閉じ中括弧は、列 1 に記述してもかまいませんが、お勧めはできません。推奨されるプログラミング手法として、入れ子になったコード・ブロックの開始と終了を示すために、中括弧はインデントするようにしてください。開き中括弧の前後に空白を入れる必要はありません。閉じ中括弧の前後に空白を入れる必要はありません。引数のないコマンドに続く中括弧の場合も同様です。中括弧の空白に関する唯一の要件は、IF コマンドの最後の節の最後の閉じ中括弧とその後のコマンドを、スペース、タブ、または改行で区切る必要があるということです。

IF コマンドは別の IF コマンド内に入れ子にすることができます。複数レベルの入れ子がサポートされます。

IF コマンドは、\$TEST 特殊変数の値の読み取りや設定を行いません。ブーリアン・テスト式が True と評価されると、\$TEST の値にかかわらず、中括弧内のコード・ブロックを実行します。

## 引数

### expression1

IF 節のテスト条件。単一の式かコンマで区切られた式のリストの形式をとります。式リストでは、InterSystems IRIS は左から右の順で、個別の式を評価します。コンマ区切りリスト内で False に評価される式に遭遇すると評価を終了します。コンマ区切りリスト内のすべての式が True と評価されると、InterSystems IRIS は IF 節に関連するコード・ブロックを実行します。リスト内の式が 1 つでも False と評価されると、InterSystems IRIS は残りの式を無視し、IF 節に関連するコード・ブロックを実行しません。

通常、expression1 は True または False と評価されるブーリアン式です (x=7 など)。["演算子と式"](#) を参照してください。IF は、以下のようにリテラル値をブーリアンの True および False として解釈します。

- TRUE : ゼロ以外の数値、またはゼロ以外の数値に評価される数値文字列。例 : 1、7、-.007、"7-7"、"7dwarves"。
- FALSE : ゼロの数値、またはゼロの数値に評価される文字列。ゼロの数値に評価される非数値文字列。0、-0.00、7-7、"0"、"TRUE"、"FALSE"、"strike3"、空文字列("") など。

詳細は、["数値としての文字列"](#) を参照してください。

### expression2

ELSEIF 節のテスト条件。単一の式かコンマで区切られた式のリストの形式をとります。これは expression1 と同じ方法で評価されます。

## QUIT での IF

QUIT コマンドが IF コード・ブロック (あるいは ELSEIF コード・ブロックまたは ELSE コード・ブロック) 内にある場合、QUIT は、このコード・ブロックが存在しないかのように、通常の QUIT コマンドとして動作します。この動作は、その他の種類の中括弧のコード・ブロック (FOR、WHILE、DO...WHILE、TRY、CATCH) 内にある QUIT と異なります。

- IF コード・ブロックが、ループ構造 (FOR コード・ブロックなど) 内で入れ子にされている場合、QUIT はそのループ構造ブロックを終了し、このループ構造コード・ブロックに続くコマンドで実行を続けます。
- IF コード・ブロックが TRY ブロックまたは CATCH ブロック内にある場合、QUIT は TRY ブロックまたは CATCH ブロックを終了し、TRY ブロックまたは CATCH ブロックに続くコマンドで実行を続けます。
- IF コード・ブロックがループ構造内または TRY ブロックまたは CATCH ブロックで入れ子にされていない場合は、QUIT は現在のルーチンを終了します。

[RETURN](#) を発行すると、ブロック構造内で発行されたどうかに関係なく、現在のルーチンが終了します。

以下の例では、IF がループ構造内にない場合の QUIT の動作を示しています。QUIT はルーチンを終了します。

### ObjectScript

```
SET y=$RANDOM(10)
IF y#2=0 {
    WRITE y," is even",!
    QUIT
    WRITE "never written"
}
ELSE {
    WRITE y," is odd",!
    QUIT
    WRITE "never written"
}
WRITE "QUIT out of the IF (never written)"
```

以下の例では、IF がループ構造内にある場合の QUIT の動作を示しています。QUIT は、FOR ループを終了してルーチンの実行を続けます。

### ObjectScript

```
FOR x=1:1:8 {
  IF x#2=0 {
    WRITE x," is even",!
    QUIT:x=4
  }
  ELSE {
    WRITE x," is odd",!
  }
}
WRITE "QUIT out of the FOR loop (written)"
```

以下の例では、IF が TRY ブロック内にある場合の QUIT の動作を示しています。QUIT が TRY ブロックを終了した後、CATCH ブロックに続く次のコードでルーチンの実行が継続されます。

### ObjectScript

```
TRY {
  SET y=$RANDOM(10)
  IF y#2=0 {
    WRITE y," is even",!
    QUIT
    WRITE "never written"
  }
  ELSE {
    WRITE y," is odd",!
    QUIT
    WRITE "never written"
  }
  WRITE "QUIT out of the IF (never written)"
}
CATCH exp1 {
  WRITE "only written if an error occurred",!
  WRITE "Error Name: ", $ZCVT(exp1.Name,"O","HTML"),!
}
TRY {
  WRITE "on to the next TRY block"
}
CATCH exp2 {
  WRITE "only written if an error occurred",!
  WRITE "Error Name: ", $ZCVT(exp2.Name,"O","HTML"),!
}
```

## GOTO での IF

IF コード・ブロック内に GOTO がある場合、プログラムは特定の制限に従い実行されます。

GOTO 文は IF コマンドの外側、あるいは現在の節のコード・ブロック内に移動できます。GOTO 文は他のコード・ブロックには移動できません。つまり、現在の IF コマンドの別の節に属するコード・ブロックや、他の IF、FOR、DO WHILE、または WHILE コマンドに属するコード・ブロックには移動できません。

## 例

以下の例では、IF コマンドを使用して、回答者を 3 つのグループのいずれかに分類し、適切なサブルーチンを呼び出します。3 つのグループは、44 歳以下の女性のグループ、44 歳以下の男性のグループ、45 歳から 120 歳までの男女のグループです。この例では、性別評価式は包含関係演算子 ([ ]) を使用します。([演算子と式](#) を参照してください。)

### ObjectScript

```
Mainloop
NEW sex,age
READ !,"What is your sex? (M or F): ",!,sex QUIT:sex=""
READ !,"What is your age? ",!,age QUIT:age=""
IF "Ff"[sex,age<45 {
  DO SubA(age)
}
ELSEIF "Mm"[sex,age<45 {
  DO SubB(age)
}
```

```
    }  
    ELSEIF "FfMm"[sex,age>44,age<125 {  
        DO SubC(age)  
    }  
    ELSE {  
        WRITE !,"Invalid data value input"  
    }  
SubA(y)  
    WRITE !,"Young woman ",y," years old"  
SubB(y)  
    WRITE !,"Young man ",y," years old"  
SubC(y)  
    WRITE !,"Older person ",y," years old"
```

## 関連項目

- ・ [DO WHILE コマンド](#)
- ・ [FOR コマンド](#)
- ・ [WHILE コマンド](#)
- ・ [GOTO コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [\\$CASE 関数](#)

## JOB (ObjectScript)

プロセスをバックグラウンドで実行します。

### 構文

```
JOB:pc jobargument,...
J:pc jobargument,...
```

jobargument は以下のいずれかになります。

ローカル・ジョブ

```
routine(routine-params):(process-params):timeout
routine(routine-params)[joblocation]:(process-params):timeout
routine(routine-params)|joblocation|:(process-params):timeout
##class(className).methodName(args):(process-params):timeout
..methodName(args):(process-params):timeout
$CLASSMETHOD(className,methodName,args):(process-params):timeout
```

リモート・ジョブ

```
routine[joblocation]
routine|joblocation|
```

### 引数

引数	説明
pc	オプション — 後置条件式
routine	JOB で作成されたプロセスによって実行されるルーチン
routine-params	オプション — ルーチンに渡されるパラメータの、コンマ区切りのリストこのパラメータは、値、式、もしくは既存のローカル変数名です。指定された場合は、最後の括弧が必須です。ルーチン・パラメータは、ローカル・ジョブにのみ渡されます。
className.methodName(args) ..methodName(args)	JOB で作成されたプロセスによって実行されるクラス・メソッドclassName を \$SYSTEM にすることはできませんが、%SYSTEM にすることは可能です。className の代わりに .. を指定すると、JOB は、現在のクラス・コンテキスト (\$THIS クラス) を使用します。variable 引数のコンマ区切りのリストはオプションですが、括弧で囲む必要があります。引数は省略できません。 <a href="#">\$CLASSMETHOD</a> の使用の詳細は、“ <a href="#">オブジェクトへの動的アクセス</a> ”を参照してください。
process-params	オプション — コロンで区切られた位置パラメータのリスト。ジョブの環境内のさまざまな要素を設定するために使用されます。process-params リストは括弧で囲まれ、括弧で囲んだリストの前にコロンが付きます。すべての process-params はオプションです。括弧は必須です。位置パラメータを指定しないことを示すには、コロンを記述する必要がありますが、末尾のコロンは省略できます。process-params 引数は、ローカル・ジョブに対してのみ指定できます。
timeout	オプション — ジョブ起動プロセスの開始を待つ秒数。秒の小数部は整数部分に切り捨てられます。最初のコロンは必須です。timeout 引数は、ローカル・ジョブに対してのみ指定できます。省略した場合、InterSystems IRIS は無限に待機します。

引数	説明
joblocation	<p>オプション - ローカル・ジョブまたはリモート・ジョブを実行するシステムおよびディレクトリを指定するために使用される、明示的または暗黙的なネームスペース。<a href="#">暗黙のネームスペース</a>は、"<code>^^dir</code>" のように 2 つのキャレット文字が先行するディレクトリ・パスです。joblocation は、角括弧 (<code>[ ]</code>)、または垂直バー (<code> </code>) で囲まれます。</p> <p>クラス・メソッドをジョブ起動する場合、joblocation を指定することはできません。joblocation でリモート・システムを指定する場合、routine-params、process-params、または timeout を指定することはできません。</p> <p>joblocation でローカル・ジョブを指定する場合、最初のプロセス・パラメータ (nspace) を指定することはできません。joblocation パラメータと競合するからです。したがって、2 番目、3 番目、および 4 番目のプロセス・パラメータのみが指定され、指定されない nspace パラメータはコロンで示される必要があります。</p>

## 概要

JOB は、ジョブ、ジョブ起動プロセス、もしくはバックグラウンド・ジョブと呼ばれる個別のプロセスを作成します。作成されたプロセスは、現在のプロセスと関係なく、バックグラウンドで実行されます。通常は、ユーザ対話も行いません。ジョブ起動プロセスは、JOB コマンドで明示的に指定されたものを除いて、呼び出されたプロセスからの構成環境を継承します。例えば、ジョブ起動プロセスは、システムの既定のロケールではなく、親プロセスのロケール設定を継承します。

これに対し、DO コマンドで呼び出されたルーチンは、現在のプロセスの一部としてフォアグラウンドで実行されます。

JOB は使用中のローカル・システムに、ローカル・プロセスを作成するか、他のシステムでリモート・プロセスの作成を実行することもできます。リモート・ジョブに関する詳細は、"[リモート・ジョブ](#)" を参照してください。

ジョブが開始されると、InterSystems IRIS は、ユーザが記述した JOB`%ZSTART ルーチンを呼び出すことができます。ジョブが終了すると、InterSystems IRIS は、ユーザが記述した JOB`%ZSTOP ルーチンを呼び出すことができます。これらのエントリ・ポイントは、ジョブ・アクティビティのログの管理および生じた問題のトラブルシューティングに使用できます。詳細は、"[InterSystems IRIS `ZSTART および `ZSTOP ルーチンの使用法](#)" を参照してください。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に JOB を実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、"[コマンド後置条件式](#)" を参照してください。

### routine

開始されるプロセスです。以下の形式のいずれかで指定することができます。

プロセス指定	説明
label	現在のルーチン内の <a href="#">行ラベル</a> を指定します。
^routine	ディスクに保存されているルーチンを指定します。InterSystems IRIS は、ディスクからそのルーチンをロードし、そのルーチン内の最初の実行可能コード行から実行を開始します。
label^routine	ディスクに保存されている指定したルーチン内の <a href="#">行ラベル</a> を指定します。InterSystems IRIS は、ディスクからそのルーチンをロードし、指定されたラベルから実行を開始します。



プロセス指定	説明
label+offset label+offset^routine	行ラベルからの指定された行数のオフセットを指定します。オフセットを使用すると、プログラムのメンテナンスに関する問題が生じることがあるので、お勧めしません。IRISSYS % ルーチンの呼び出し時における offset の指定はできません。指定しようとすると、InterSystems IRIS は <NOLINE> エラーを発行します。

プロシージャ・ブロック内では、JOB コマンドを呼び出すと、プロシージャ・ブロックの範囲外にある子プロセスが開始します。そのため、プロシージャ・ブロック内の label 参照を解決できません。この結果、プロシージャ内のラベルを参照する JOB では、プロシージャはプロシージャ・ブロックを使用できません。

存在しないラベルを指定した場合、InterSystems IRIS は <NOLINE> エラーを返します。存在しないルーチンを指定した場合、InterSystems IRIS は <NOROUTINE> エラーを返します。これらのエラーの詳細は、“\$ZERROR” 特殊変数を参照してください。

### routine-params

コンマで区切られた値のリスト、式、または既存のローカル変数名です。括弧は必須です。このリストは、実パラメータ・リストと呼ばれます。ルーチンには、仮パラメータ・リストが必須です。また、このリスト内のパラメータ数は、実パラメータ・リスト内のパラメータ数以上でなければなりません。指定した実パラメータの方が多い場合、InterSystems IRIS は <PARAMETER> エラーを返します。呼び出されるルーチンに仮パラメータ・リストが含まれている場合は、それを囲む括弧を指定する必要があります。パラメータを渡さない場合も同様です。routine-params のリストから項目を省略することはできません。

ルーチン・パラメータは値によってのみ渡すことができますつまり、配列を渡すことはできないということです。この点で、値でも参照でもパラメータを渡すことができる DO コマンドとは異なります。この制限により、JOB コマンドでは配列を渡すことができません。配列は、参照でしか渡すことができないからです。

ジョブ起動プロセスにオブジェクト参照 (oref) を渡すことはできません。呼び出し元のプロセス・コンテキスト内に存在するオブジェクトへの参照は、新しいジョブ起動プロセス・コンテキスト内では参照できないためです。oref を渡そうとすると、ジョブ起動プロセスに空文字列(”) が渡されます。

ルーチンを開始すると、InterSystems IRIS はすべての式を評価し、実リストの各パラメータの値をその位置に従って、仮リストの対応変数にマップします。仮リスト内の変数が、実リスト内のパラメータよりも多い場合は、余った変数は未定義のままになります。

ルーチン・パラメータは、ローカル・プロセスにのみ渡されます。リモート・ジョブを作成しているときは、ルーチン・パラメータを指定することはできません。“リモート・ジョブ” を参照してください。

### process-params

コロンの区切られた位置パラメータのリスト。ジョブの環境内のさまざまな要素を設定するために使用されます。先頭のコロンと括弧は必須です。すべての位置パラメータはオプションです。process-params には、最大 6 つの位置パラメータを指定できます。これらの 6 つのパラメータは以下のとおりです。

```
(nspace:switch:principal-input:principal-output:priority:os-directory)
```

これらのパラメータの位置は決められているので、必ず示した順に指定してください。指定するパラメータの前に省略するパラメータがある場合は、プレースホルダとしてコロンを付ける必要があります。

プロセス・パラメータは、リモート・ジョブに対しては指定できません。

以下のテーブルは、プロセス・パラメータの説明です。



プロセス・パラメータ	概要
nspc	<p>プロセスの既定のネームスペースです。指定された routine は、このネームスペースから引き出されます。nspc を省略した場合、現在の既定のネームスペースが、そのジョブ起動プロセスの既定のネームスペースになります。無効なネームスペースを指定した場合は、ジョブが開始されない可能性があります。ローカル・ジョブでは、プロセス・パラメータとして joblocation 引数とネームスペースの両方を指定することはできません。nspc プロセス・パラメータを省略し、プレースホルダのコロンを保持する必要があります。</p>
switch	<p>以下のうち 1 つ以上の合計で構成されている整数</p> <p>0 または複数の以下のフラグを表すことができる整数ビット・マスク</p> <p>1 – 生成されたジョブにシンボル・テーブルを渡します。</p> <p>2 – JOB サーバは使用しません。</p> <p>4 – 主入出力デバイス (<a href="#">\$PRINCIPAL</a>) を使用して、開いている TCP/IP ソケットを、生成されたジョブに渡します。(非推奨。代わりに 16 を使用してください。以下を参照してください。)</p> <p>8 – 生成されたジョブの 2 桁の年に対するプロセス独自のウィンドウが、システム全体の既定のスライディング・ウィンドウ定義になるように設定します。特に設定されない場合、生成されたジョブは、JOB コマンドを実行したプロセスのスライディング・ウィンドウ定義を継承します。</p> <p>16 – 現在の主入出力デバイス (<a href="#">\$IO</a>) を使用して、開いている TCP/IP ソケットを、生成されたジョブに渡します。</p> <p>128 ~ 16384 (32 の倍数) – ジョブ起動子プロセスのパーティション・サイズ (KB) を指定する追加の整数値です。詳細は、"<a href="#">子プロセスのパーティション・サイズを指定する</a>" を参照してください。</p> <p>switch 値は、これらの整数の組み合わせの合計を設定できます。例えば、switch 値 13 (1 + 4 + 8) はシンボル・テーブル (1) を渡し、開いている TCP/IP ソケット (4) を渡し、システム全体の既定である 2 桁の年 (8) のための、プロセス独自のウィンドウを設定します。</p> <p>スイッチのブロックは、%SYSTEM.Util クラスの CheckSwitch() メソッドを使用して決定できます。</p>
principal-input	<p>プロセスの主入出力デバイスです。既定値は NULL デバイスです。</p>
principal-output	<p>プロセスの主出力デバイスです。既定値は、principal-input に指定するデバイスか、どちらのデバイスも指定していない場合は NULL デバイスです。</p> <p>UNIX® : どちらのデバイスも指定していない場合、プロセスは、JOB コマンドで開始されたプロセスの既定の主デバイス /dev/null を使用します。</p>
priority	<p>UNIX® – 子プロセスの優先度を指定する整数 (オペレーティング・システムの制約に従います)。指定しない場合は、子プロセスは親プロセスのベース優先度とシステム定義のジョブ優先度の変更子を使用します。<a href="#">\$VIEW</a> 関数を使用して、ジョブの現在の優先度を決定することができます。Windows の標準優先度は 7 です。UNIX® の優先度は、-20 ~ 20 の範囲で、標準優先度は 0 です。UNIX® では、root として実行していない限り、プロセス自身でその優先度を上げることはできません。</p>

プロセス・パラメータ	概要
os-directory	ファイル入出力用のオペレーティング・システムの作業ディレクトリ。既定では、親プロセスから継承した作業ディレクトリを使用します。このパラメータは、システムによっては無視される可能性があります。

#### Switch 4 および Switch 16

渡された TCP デバイスが子ジョブの主デバイスとして確立されるため、switch=4 の使用はお勧めしません。この場合、子ジョブは TCP リモート接続が切断されたことを検出したときに停止する場合があります、エラー・トラップを実行しません。代わりに、ユーザは、switch=16 を使用してから、子ジョブで %SYSTEM.INetInfo.TCPName() メソッドを使用し、渡された TCP デバイスの名前を取得する必要があります。この場合、子ジョブの主デバイスが渡された TCP デバイスではないため、子ジョブは TCP リモート接続が切断されたことを検出ときに実行を継続する場合があります。

#### timeout

ジョブがタイムアウトし、中止される前に、ジョブ起動プロセスの開始を待機する秒数です。最初のコロンは必須です。timeout は、整数値または式として指定する必要があります。ジョブ起動プロセスがタイムアウトになると、InterSystems IRIS はプロセスを中止し、\$TEST 特殊変数を 0 (FALSE) に設定します。実行は、呼び出しルーチンの次のコマンドから続行します。エラー・メッセージは発行されません。ジョブ起動プロセスが成功すると、InterSystems IRIS は \$TEST を 1 (TRUE) に設定します。\$TEST は、ユーザが設定することもできます。また、LOCK、OPEN、または READ のタイムアウトで設定されることもあります。

タイムアウトは、ローカル・プロセスに対してのみ指定できます。

#### 例

以下の例は、監視ルーチンをバックグラウンドで開始します。プロセスが 20 秒以内に開始されない場合、InterSystems IRIS は \$TEST を False (0) に設定します。

#### ObjectScript

```
JOB ^monitor:::20
WRITE $TEST
```

以下の例は、Disp という名前の行ラベルで、監視ルーチンの実行を開始します。

#### ObjectScript

```
JOB Disp^monitor
```

以下の例は、ADD ルーチンを開始します。このとき、変数 num1 内の値、値 8、および式  $a + 2$  の結果の値がルーチンに渡されます。ADD ルーチンには、少なくとも 3 つのパラメータを含む、仮パラメータ・リストが含まれます。

#### ObjectScript

```
JOB ^Add(num1,8,a+2)
```

以下の例は、ADD ルーチンを開始します。このルーチンは仮パラメータ・リストを持っていますが、パラメータは渡されません。この場合、ADD ルーチンは、呼び出し元のルーチンから値を受け取らないので、その仮パラメータに既定値を代入するコードが含まれていなければなりません。

#### ObjectScript

```
JOB ^Add( )
```

以下の例は、ラベル AA の現在のルーチンで実行するプロセスを作成します。プロセス・パラメータは、現在のシンボル・テーブルをルーチンに渡します。JOB サーバを使用できます。

## ObjectScript

```
JOB AA:(":":1)
```

以下の例では、ルーチン・パラメータ VAL1 および文字列 "DR." を、現在のネームスペースの、エントリ・ポイント ABC で開始される ^PROG ルーチンに渡します。このルーチンは 2 つの引数を要求します。InterSystems IRIS は現在のシンボル・テーブルをこのジョブに渡しません。また、使用できる場合は JOB サーバを使用し、主入出力デバイスとして tta5: を使用します。

## ObjectScript

```
JOB ABC^PROG(VAL1,"DR."):(:0:"tta5:")
```

以下は、10 秒でタイムアウトするクラス・メソッドのジョブ起動の例です。主入出力デバイスとして tta5: を使用します。

以下の例では、##class 構文を使用してクラス・メソッドを呼び出します。

## ObjectScript

```
JOB ##class(MyClass).Run():(:0:"tta5:"):10
```

以下の例では、\$CLASSMETHOD 関数を使用してクラス・メソッドを呼び出します。

## ObjectScript

```
JOB $CLASSMETHOD("MyClass","Run"):(:0:"tta5:"):10
```

以下の例では、[相対ドット構文](#) (..) を使用して現在のオブジェクトのメソッドを参照します。

## ObjectScript

```
JOB ..Cleanup():(:0:"tta5:"):10
```

または単純に以下ようになります。

## ObjectScript

```
JOB ..Cleanup()::10
```

詳細は、“[オブジェクト特有の ObjectScript の機能](#)” を参照してください。

## メモ

### InterSystems IRIS が割り当てるジョブ番号とメモリ・パーティション

ジョブ起動プロセスの開始後、InterSystems IRIS は、そのプロセスに個別のメモリ・パーティションと、一意のジョブ番号 (またはプロセス ID、PID と呼ばれます) を割り当てます。ジョブ番号は、特殊変数 \$JOB に保存されます。ジョブの状態 (そのジョブが JOB コマンドで開始されたか否かも含む) は、\$ZJOB 特殊変数に格納されます。

ジョブ起動プロセスは、それぞれ別個のメモリ・パーティションを保持しているので、そのプロセスを作成したプロセスやジョブ起動プロセスが共通のローカル変数環境を共有することはありません。ジョブ起動プロセスを開始するときには、パラメータ渡し (routine-params) により、現在のプロセスからジョブ起動プロセスに値を渡すことができます。

JOB コマンドが失敗する一般的な原因は以下のとおりです。

- ・ 空きパーティションがない
- ・ process-params で指定された特性でパーティションを作成する十分なメモリがない

## プラットフォームで異なるジョブ起動プロセスの許可

JOB コマンドにより作成されたプロセスは、インターシステムズのサービス・アカウント・ユーザとして実行します。つまり、インターシステムズのサービス・アカウントに、必要なすべてのリソースへの明示的なアクセス権が付与されていることを確認する必要があります。

生成されたジョブ・プロセスは JOB コマンドを発行したプロセスのユーザ ID 以外で実行することもできます。生成されたジョブ・プロセスのユーザ ID はプラットフォームによって、以下のように異なります。

- Windows プラットフォームでは、ジョブ・プロセスは InterSystems IRIS インスタンス用に確立されたユーザ ID を使用します。
- UNIX® プラットフォームでは、ジョブ・プロセスは JOB コマンドを発行したプロセスのユーザ ID を使用します。

したがって、ジョブを生成するときは、ジョブ・プロセスのユーザ ID に、ジョブ実行中に読み書きされるファイルを使用するために必要な許可があることを確認する必要があります。

## ジョブ間の通信

値渡しのパラメータは、1 方向のみ、およびジョブの起動時のみ発生します。プロセス相互の通信を行うには、双方が合意したグローバル変数を使用する必要があります。このような変数は、プロセス間の情報交換を可能にすることだけが目的なので、一般にスクラッチ・グローバルと呼ばれます。

- プロセスでは、`%SYSTEM.Event` クラス・メソッドを使用して、ジョブ間の通信を行えます。
- 特殊なプロセス・パラメータを指定することによって、現在のプロセスのすべてのローカル変数を、呼び出されたプロセスに渡すことができます。
- プロセスでは、IPC (プロセス間通信) デバイス (デバイス番号 224 ~ 255) を介してジョブ間の通信を行うこともできます。また、UNIX® オペレーティング・システムの場合は、UNIX® のパイプを介した通信が可能です。

## デバイス所有権の設定

InterSystems IRIS は、新しいプロセスのデバイス所有権を扱うコード (OPEN コマンドや USE コマンド) を呼び出し先ルーチンが含むと想定しています。既定のデバイスは、NULL デバイスです。

InterSystems IRIS は、サインイン時に開始されたプロセス以外のプロセスには、既定のデバイスを割り当てません。

## ジョブ優先度の設定

`%PRIO` ユーティリティを使用すると、UNIX® のジョブ起動プロセスの実行優先度を制御できます。使用可能なオプションは、NORMAL (負荷分散を使用して CPU の使用効率を調整)、LOW および HIGH です。優先度が HIGH の場合、ジョブ起動プロセスは、対話型プロセスと対等に CPU リソースを取り合います。

InterSystems IRIS で、ジョブ起動プロセスに対して既定の優先度を設定することもできます。

`BatchFlag()` メソッドを使用すると、プロセスをバッチ・モードで実行するように設定できます。バッチ・モードのプロセスは、バッチでないプロセスよりも優先度が低くなります。

## Raw パーティションで JOB コマンドを使用する (UNIX®)

以下のいずれかの方法で、JOB コマンドを Raw パーティションで使用できます。

- Raw パーティションで JOB コマンドを実行する
- 他のネームスペースで JOB コマンドを実行し、JOB コマンドのプロセス・パラメータ `nspc` として、ロー・パーティションを指定するこの `nspc` は、**暗黙のネームスペース**です。暗黙のネームスペースは、`^^^dir` のように 2 つのキャレット文字が先行するディレクトリ・パスです。暗黙のネームスペースの構文については、[拡張グローバル参照](#)を参照してください。

Raw パーティションで実行しているコマンドとジョブ起動プロセスの場合、ファイル名を参照するときは常に、[フル・パス名](#)を指定する必要があります。また、“.” や “..” で開始するパス名は使用してはなりません。これらは特殊な UNIX® ファイルであり、Raw パーティションには存在しないからです。これらの規則のいずれかに違反すると、<DIRECTORY> エラーが発生します。

現在のネームスペースのフル・パス名を取得するには、以下の例に示すように、NormalizeDirectory() メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%Library.File).NormalizeDirectory( " ")
```

また、ObjectScript の JOB コマンドの代わりに、UNIX® のジョブ制御構文 (&) を使用することもできます。

## リモート・ジョブ

リモート・ジョブを開始する前に、[ECP接続を確立](#)し、[netjob](#) パラメータを True に設定する必要があります。これにより、サーバはリモート ECP クライアント・システムからのジョブ要求を処理することができます。

リモート・ジョブ要求を受信する予定のシステムには、あらかじめその機能を設定しておく必要があります。

受信システムで管理ポータルに進み、[\[システム管理\]](#)、[\[構成\]](#)、[\[追加の設定\]](#)、[\[メモリ詳細\]](#) の順に選択します。表示して編集する [netjob](#) を見つけます。“true” の場合、ECP を介して着信するリモート・ジョブ要求は、このサーバで処理されます。既定は“true”です。

リモート・システムのライセンスでは、リモートで開始されるジョブを実行できる十分なユーザがサポートされている必要があります。“[インターシステムズ・クラス・リファレンス](#)”の説明に従って、%SYSTEM.License クラスのクラス・メソッドを使用して、使用できる InterSystems IRIS ライセンスの数を決定できます。

## リモート・ジョブ要求のジョブ構文

以下の構文を使用すると、ある InterSystems IRIS システムから別の InterSystems IRIS システムへ、リモート・ジョブを送信できます。

```
JOB routine[joblocation]
JOB routine|joblocation|
```

2 つの形式は類似しています。joblocation パラメータを囲むのは、角括弧 ([ ])、または垂直バー (|) のどちらでも使用できます。リモート・ジョブは、ルーチン・パラメータ、プロセス・パラメータ、またはタイムアウトを渡すことはできません。

- ・ joblocation – ジョブの場所を指定します。角括弧 ([ ]) や垂直バー (|) は必須です。

ジョブの場所を指定する構文の形式によって、InterSystems IRIS の動作が異なります。

joblocation 構文	結果
["namespace"]	InterSystems IRIS は、この明示的なネームスペースが、ローカル・システムまたはリモート・システムに既定のデータセットを持っているかどうかをチェックします。既定のデータセットがローカル・システムにある場合、システムは指定されたパラメータを使用してジョブを開始します。既定のデータセットがリモート・システムにある場合、システムはネームスペースの既定のデータセットのディレクトリでリモート・ジョブを開始します。
["dir","sys"]	InterSystems IRIS はこの場所を <a href="#">暗黙のネームスペース</a> の形式 ["^sys^dir"] に変換します。
["^sys^dir"]	ジョブは指定されたリモート・システムの指定されたディレクトリ内で実行します。InterSystems IRIS では、ルーチン・パラメータ、プロセス・パラメータ、またはタイムアウト指定は許可されていません。



joblocation 構文	結果
[“^^dir”]	ジョブは現在のシステムのローカル・ジョブとして、指定されたディレクトリ（ <b>暗黙のネームスペース</b> ）で指定されたパラメータを使用して動作します。暗黙のネームスペースは、“^^dir” のように 2 つのキャレット文字が先行するディレクトリ・パスです。
[“dir”, “”]	<COMMAND> エラーが発行されます。

### リモート・ジョブでのグローバル・マッピング (Windows)

要求システムでグローバル・マッピングが定義されているかどうかにかかわらず、InterSystems IRIS ではリモート・ジョブに対してグローバル・マッピングは用意されていません。グローバル・マッピングの不足を回避するためには、グローバル指定で、そのネームスペースにないグローバルの場所をポイントする拡張参照を使用してください。拡張参照で指定するネームスペースが、指定するシステムで定義されていない場合、<NAMESPACE> エラーが返されます。ネームスペースと拡張グローバル参照の構文に関する詳細は、“[グローバルについての正式な規則](#)”を参照してください。

### 特殊変数 \$ZCHILD と \$ZPARENT の使用

**\$ZPARENT** には、現在のプロセスをジョブ化したプロセスの PID (プロセス ID) が含まれます。現在のプロセスが JOB コマンドで作成されなかった場合は、0 が含まれます。

**\$ZCHILD** には、JOB コマンドによって最後に作成されたプロセスの PID が含まれています。このプロセスは成功したものではありません。

\$ZCHILD を使用すると、JOB コマンドを実行する前後の \$ZCHILD 値を比較することにより、**リモート・ジョブ**の実行状態を判断することができます。前後の値が異なり、後の値がゼロ以外の場合、後の \$ZCHILD 値は、新しく作成されたリモート・ジョブの PID であり、プロセスが正常に作成されたことを示します。後の値がゼロの場合、または後の値が前の値と同じ場合、リモート・ジョブは作成されていません。

\$ZCHILD からは、リモート・ジョブが作成されたことのみがわかります。リモート・ジョブが正常に実行されたかどうかはわかりません。リモート・プロセスがエラーなく実行され、完了まで実行されたかどうかを判断する最も良い方法は、実行中のコードにある種のログとエラー・トラップを用意することです。リモート・ジョブ・メカニズムは、リモート・プロセス・エラーやリモート・プロセスの終了、正常に実行されたかどうかなどについて、親プロセスに一切通知しません。

### JOB サーバの使用

JOB サーバは、JOB 要求の処理を待つ InterSystems IRIS プロセスです。JOB サーバに付属しているジョブ起動プロセスは、新しいプロセスを作成する必要があるというオーバーヘッドの付加を回避します。使用可能な場合、JOB サーバを使用するように設定された switch パラメータで JOB コマンドが発行されるたびに、InterSystems IRIS は、これに対処できる JOB サーバがあるかどうかをチェックします。JOB サーバがない場合は、Caché がプロセスを作成します。使用できる JOB サーバがある場合、JOB はその JOB サーバに付属します。

JOB サーバで実行されている JOB で HALT が実行された場合、JOB サーバは他の JOB 要求を受け取るまで、ハイバネーション状態になります。JOB サーバで動作していないジョブ起動プロセスは終了し、そのプロセスは削除されます。

JOB サーバ環境と、ジョブ起動プロセス環境では回避できない相違点がいくつかあります。ジョブ起動プロセスを JOB サーバで実行する場合に、これらがセキュリティ上の問題になる可能性があります。ジョブ起動プロセスは、InterSystems IRIS レベルで JOB コマンドを発行したプロセスのセキュリティ属性を受け付けます。

### 入出力デバイス

一度に 1 つのプロセスだけが、デバイスを所有できます。これは、ユーザがデバイス 0 を閉じる場合でも、JOB サーバで実行中のジョブが、主入出力デバイスに対して入力、または出力を実行できないことを意味します。

したがって、JOB サーバに入力を実行させたい場合は、以下を指定します。

- ・ 代替入力デバイス

- ・ 出力デバイス用の NULL デバイス (出力を表示させたくない場合)

このガイドラインに従わないと、JOB サーバで実行中のジョブは、主入出力デバイスに対する入力や出力を行う必要がある場合に停止することがあります。JOB の出力が頻繁に端末に表示される場合もありますが (SHARE 特権を持っている場合など)、通常は表示されません。

## 実行されないジョブに関するトラブルシューティング

ジョブが開始しない場合は、入出力指定をチェックしてください。InterSystems IRIS が要求デバイスを開くことができない場合、ジョブは開始されません。なお、NULL デバイス (UNIX® では /dev/null) はいつでも利用できます。

ジョブが開始されてもすぐに停止される場合は、適切なスワップ領域があることを確認してください。十分なスワップ領域がない場合、ジョブはエラーを受け取ります。JOB コマンドによって作成された新しいプロセスが、すぐに停止した場合や、プロセスの起動が完了する前に `RESJOB` ユーティリティで終了した場合、JOB コマンドは <HALTED> または <RESJOB> エラーを生成します。詳細は "[HALT](#)" コマンドを参照してください。

ジョブが開始されない場合は、JOB コマンドで正しいネームスペースを使用していることを確認してください。Exists() メソッドを使用して、ネームスペースが定義されているかどうかをテストすることができます。

### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a nonexistent namespace
```

JOB コマンドが目的どおりに実行されない場合は、以下を試してください。

- ・ ルーチンを DO コマンドで実行します。
- ・ InterSystems IRIS でライセンス付与されているプロセスの数を超えていないことを確認してください。"インターシステムズ・クラス・リファレンス" の説明に従って、`%SYSTEM.License` クラスのクラス・メソッドを使用して、使用できる InterSystems IRIS ライセンスの数を決定できます。
- ・ JOB コマンドに timeout パラメータがある場合は、システムの速度により指定されたタイムアウト時間に達していないかどうかを確認してください。

多数のジョブを同時に実行している場合、ルーチン・バッファもしくはライセンス・スロットを待っている間に、JOB コマンドが停止することがあります。Ctrl-C を使用して、JOB コマンドに割り込むことができます。

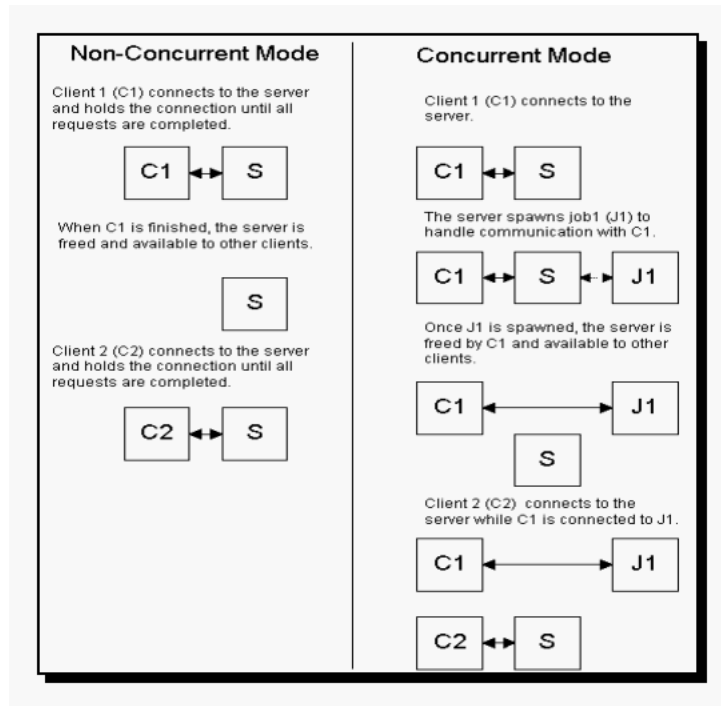
## JOB コマンドの終了

ジョブ起動プロセスは、終了する前にそれを作成したプロセスがログオフされた場合でも、最後まで続行されます。

## TCP デバイスを使用した JOB コマンド

JOB コマンドを使用して、TCP 並行サーバを実装することができます。TCP 並行サーバを使用すると、複数のクライアントを同時に処理できます。このモードでは、クライアントは、サーバが他のクライアントのサービスを終了するまで待機する必要はありません。代わりに、クライアントがサーバに要求を送信するたびに、開いているクライアントに対して必要に応じて個別のサブジョブを生成します。このサブジョブが生成された直後に (JOB コマンドから制御が戻ります)、他のクライアントが処理を要求することがあります。サーバは、そのクライアントのサブジョブも同様に生成します。

図 C-1: 非並行モードおよび並行モードのクライアント/サーバ接続



並行サーバは、ビット 4 またはビット 16 のビット・マスクが設定された switch プロセス・パラメータを指定して JOB コマンドを使用し、生成されたプロセスに入力および出力のプロセス・パラメータを渡します。

- switch にビット 4 を指定する場合、常に、プロセス・パラメータ principal-input および principal-output の両方の TCP デバイスを指定する必要があります。以下のように、principal-input と principal-output の両方に同デバイスを使用する必要があります。

#### ObjectScript

```
JOB child:( :4:tcp:tcp)
```

その後、生成されたプロセスは、この単一の入出力デバイスを以下のように設定します。

#### ObjectScript

```
SET tcp=$IO
```

- switch にビット 16 を指定する場合、以下のように、プロセス・パラメータ principal-input および principal-output の TCP デバイスに異なるデバイスを指定できます。

#### ObjectScript

```
USE tcp
JOB child:( :16:input:output)
```

JOB コマンドの前の USE tcp は、(主デバイスではなく) 現在のデバイスを TCP デバイスとして指定します。その後、生成されたプロセスはこれらのデバイスを以下のように設定できます。

#### ObjectScript

```
SET tcp=##class(%SYSTEM.INetInfo).TCPName()
SET input=$PRINCIPAL
SET output=$IO
WRITE tcp," ",input," ",output
```



4 ビットまたは 16 ビットが設定されている場合、JOB コマンドが、ジョブ起動プロセスに TCP ソケットを渡すことが重要となります。追加の機能ごとに適切なビット・コードを追加すると、JOB コマンドの他の機能と組み合わせることができます。例えば、switch が値 1 のビットを含むときは、シンボル・テーブルが渡されます。並行モードをオンにし、シンボル・テーブルを渡すには、switch の値を 5 (4+1) または 17 (16+1) にします。

JOB コマンドを実行する前に、TCP デバイスは以下の状態になる必要があります。

- ・ 接続されている
- ・ TCP ポートで待ち受け状態にある
- ・ 接続を受け入れ済みである

JOB コマンドの後、生成されているプロセスのデバイスは継続して TCP ポートで待ち受け状態にありますが、アクティブな接続は既に存在していません。アプリケーションは、JOB コマンドの実行後 \$ZA 特殊変数をチェックし、TCP デバイスで CONNECTED ビットがリセットされていることを確認する必要があります。

生成されたプロセスは、指定の TCP デバイスを principal-input デバイスおよび principal-output デバイスに使用し、指定のエントリ・ポイントで開始します。子プロセスの TCP デバイス名は、親プロセスの名前と同じです。TCP デバイスには、付属のソケットが 1 つあります。USE コマンドは、“M” モードで TCP デバイスを構築するために使用されます。これは“PSTE”と同等です。“P” (pad) オプションは、レコード・ターミネータ文字で出力を埋め込む際に必要です。このモードを設定すると、WRITE ! では LF (改行)、WRITE # では FF (改ページ) が送信されます。さらに、書き込みバッファがフラッシュされます。

生成したプロセスの TCP デバイスは、接続状態です。これは、デバイスをクライアントから開いた後に受け取る状態と同じです。生成されたプロセスは、TCP デバイスを明示的な USE 文で使用します。また、TCP デバイスを明示的に使用することもできます。

以下の例は、非常に単純な並行サーバです。このサーバは、クライアントからの接続を検出するたびに子ジョブを生成します。JOB は、並行サーバ・ビット 16 およびシンボル・テーブル・ビット 1 で構成される switch 値 17 を使用します。

## ObjectScript

```
server ;
    SET io="|TCP|1"
    SET ^serverport=7001
    OPEN io:(^serverport:"MA"):200
    IF $TEST=0 {
        WRITE !,"Cannot open server port"
        QUIT }
    ELSE { WRITE !,"Server port opened" }
loop ;
    USE io READ x ; Read for accept
    USE 0 WRITE !,"Accepted connection"
    USE io
    JOB child:(^17::) ; Concurrent server bit is on
    GOTO loop
child ;
    SET io=##class(%SYSTEM.INetInfo).TCPName()
    SET input=$PRINCIPAL
    SET output=$IO
    USE io:(:"M") ; Ensure that "M" mode is used
    WRITE $JOB,! ; Send job id on TCP device to be read by client
    QUIT
client ;
    SET io="|TCP|2"
    SET host="127.0.0.1"
    OPEN io:(host:^serverport:"M"):200 ; Connect to server
    IF $TEST=0 {
        WRITE !,"Cannot open connection"
        QUIT }
    ELSE { WRITE !,"Client connection opened" }
    USE io READ x#3:200 ; Reads from subjob
    IF x="" {
        USE 0
        WRITE !,"No message from child"
        CLOSE io
        QUIT }
    ELSE {
        USE 0
```

```
WRITE !,"Child is on job ",x
CLOSE io
QUIT }
```

子ジョブは、継承した TCP 接続を使用して、ジョブ ID (この場合は 3 文字とします) をクライアントに渡します。その後、子プロセスは終了します。クライアントは、サーバとの接続を開き、その状態で子のジョブ ID を読み取ります。この例では、変数 "host" の IPv4 の値 "127.0.0.1" がローカル・ホスト・マシンへのループバック接続を示しています (対応する IPv6 ループバックの値は "0:0:0:0:0:0:0:1" または "::1" です)。["host" にそのサーバの IP アドレス、または名前が設定される場合は、そのサーバからクライアントを別のマシンにセットアップすることもできます。](#) IPv4 および IPv6 の形式に関する詳細は、["IPv6 アドレスの使用"](#) を参照してください。

原則では、子プロセスとクライアントは広範囲の通信を実行できます。複数のクライアントが同時に対応するサーバの子と通信できます。

## 子プロセスのパーティション・サイズを指定する

バックグラウンド・ジョブのパーティション・サイズ ([\\$ZSTORAGE](#) size) は、以下のように決定されます。

- ・ ジョブ・サーバがアクティブな場合、すべてのジョブ・サーバ・プロセスについて、ジョブ・パーティション・サイズは常に 262144 (キロバイト単位) になります。
- ・ ジョブ・サーバが非アクティブの場合、ジョブ・パーティション・サイズは、非バックグラウンド・ジョブの既定のパーティション・サイズ ([bbsiz](#)) に既定で設定されます。これは、JOB を発行する親プロセスのパーティション・サイズに関わらず、JOB を実行した際に有効になるシステム全体の既定です。

ジョブ・サーバが有効化されているが、それらのすべてがアクティブであり、ユーザが新しいジョブを開始した場合、そのジョブ・プロセスはジョブ・サーバ・プロセスにならないため、その [\\$ZSTORAGE](#) 値は [bbsiz](#) に既定で設定されます。これはジョブ・サーバがアクティブの場合にも当てはまりますが、ジョブのロードがジョブ・サーバ以外のプロセスで実行されるためです。

- ・ ジョブ・サーバが非アクティブの場合、JOB 文でジョブ起動子プロセスのパーティション・サイズを指定できます。ジョブ起動プロセスのパーティション・サイズ (キロバイト単位) は、JOB コマンドの第 2 プロセス・パラメータで指定します。例えば、JOB ^myroutine(:8192) のように指定します。指定する値は、32 の倍数で、範囲は、128 ～ 16384 です。また、既定のパーティション・サイズを超える値は指定できません。これは、既定未満の値を指定する目的でのみ使用できます。

オプションで、通常 JOB の第 2 プロセス・パラメータに入れる他のプロセス情報と組み合わせて、パーティション・サイズのプロセス・パラメータの値を指定することもできます。JOB コマンド JOB ^myroutine(:544+1) について考えてみましょう。このコマンドは、ジョブを実行しているプロセスのシンボル・テーブルをジョブ起動プロセスに渡すこと、また、ジョブ起動プロセスのパーティション・サイズを 544 K にすることを指定しています。この第 2 パラメータで、2 つの値 (544 と 1) を 545 として渡すように指定することもできますが、544+1 とした方が分かりやすく、まったく同じ効果が得られます。

ジョブ自体が [SET \\$ZSTORAGE](#) を使用してプログラマチックに独自のパーティション・サイズを設定できることに注意してください。

## 関連項目

- ・ [^\\$JOB](#) 構造化システム変数
- ・ [\\$JOB](#) 特殊変数
- ・ [\\$TEST](#) 特殊変数
- ・ [\\$ZJOB](#) 特殊変数
- ・ [\\$ZCHILD](#) 特殊変数
- ・ [\\$ZPARENT](#) 特殊変数
- ・ [\\$ZF\(-1\)](#) 関数

- ・ [\\$ZF\(-2\) 関数](#)
- ・ [TCP クライアント/サーバ通信](#)
- ・ [スプール・デバイス](#)
- ・ [プロセスの管理](#)

## KILL (ObjectScript)

変数を削除します。

### 構文

```
KILL:pc killargument,...
K:pc killargument,...
```

killargument には、以下を指定できます。

```
variable,...
(variable,...)
```

### 引数

引数	説明
pc	オプション - 後置条件式。
variable	オプション - 変数名、またはコンマで区切られた変数名のリスト。括弧なし : 削除される変数。括弧付き : 削除されない変数。

### 概要

KILL コマンドには、以下の 3 つの形式があります。

- ・ KILL 引数なし (引数なしの KILL と呼ばれる)
- ・ KILL 変数リスト付き (包含的 KILL と呼ばれる)
- ・ KILL 括弧で囲まれた変数リスト付き (排他的 KILL と呼ばれる)

引数なしの KILL コマンドは、すべてのローカル変数を削除します。このコマンドは、プロセス・プライベート・グローバル、グローバル、またはユーザ定義の特殊変数を削除しません。

引数として変数またはコンマ区切りの変数リストを持つ KILL。

```
KILL variable,...
```

は、包含的 KILL と呼ばれます。これは、引数で指定した変数だけを削除します。変数を削除すると、指定された変数よりも低いレベルにあるその変数の添え字すべてが削除されます。変数は、ローカル変数、プロセス・プライベート・グローバル、またはグローバルのいずれかです。これらは、実際の定義済み変数である必要はありませんが、有効な変数名でなければなりません。その値がユーザ指定されている場合でも、特殊変数を削除できません。変更を試みると、〈SYNTAX〉エラーが生成されます。

引数として括弧に囲まれた変数、またはコンマ区切りの変数リストを持つ KILL。

```
KILL (variable,...)
```

は、排他的 KILL と呼ばれます。これは、引数で指定したローカル変数以外の、すべてのローカル変数を削除します。指定できる変数は、ローカル変数のみです。添え字の付いた変数を指定することはできません。ローカル変数を指定することで、変数およびその添え字すべてが保持されます。指定したローカル変数は、実際の定義済み変数である必要はありませんが、有効なローカル変数名でなければなりません。

**注釈** KILL は、InterSystems IRIS オブジェクトによって生成されたローカル変数を削除できます。したがって、システム構造 (%Save で現在使用されている %objTX など) や、システム・オブジェクト (ストアード・プロシージャ・コンテキスト・オブジェクトなど) に影響を与える可能性があるコンテキストで、引数なしの KILL や排他的 KILL を使用してはなりません。大半のプログラミング・コンテキストでは、KILL のこの形式は使用されません。

**\$DATA** 関数を使用することで、変数が定義済みであるか未定義であるか、および定義済み変数に添え字があるかを確認できます。変数を削除すると、その \$DATA のステータスが未定義に変わります。

変数を削除するために KILL を使用すると、ローカル変数の格納領域を解放します。現在のプロセスの最大格納領域 (KB 単位) を判別、または設定するには、**\$ZSTORAGE** 特殊変数を使用します。現在のプロセスの使用可能な格納領域 (KB 単位) を判別するには、**\$STORAGE** 特殊変数を使用します。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に KILL コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

### variable

括弧で囲まれていない場合 :KILL コマンドによって削除される変数。variable は単一の変数、またはコンマで区切られた変数のリストです。

括弧で囲まれている場合 :KILL コマンドで削除されないローカル変数。KILL コマンドは、それ以外のすべてのローカル変数を削除します。variable は単一の変数、またはコンマで区切られた変数のリストです。

## 例

以下の例では、包含的 KILL が、ローカル変数 a、b、および d を削除し、プロセス・プライベート・グローバル ^|ppglob およびその添え字すべてを削除します。他の変数には影響はありません。

### ObjectScript

```
SET ^|ppglob(1)="fruit"
SET ^|ppglob(1,1)="apples"
SET ^|ppglob(1,2)="oranges"
SET a=1,b=2,c=3,d=4,e=5
KILL a,b,d,^|ppglob
WRITE "a=", $DATA(a), " b=", $DATA(b), " c=", $DATA(c), " d=", $DATA(d), " e=", $DATA(e), !
WRITE " ^|ppglob(1)=", $DATA(^|ppglob(1)),
      " ^|ppglob(1,1)=", $DATA(^|ppglob(1,1)),
      " ^|ppglob(1,2)=", $DATA(^|ppglob(1,2))
```

以下の例では、包含的 KILL がローカル変数 a(1) およびその添え字 a(1,1)、a(1,2) および a(1,1,1) を削除します。ローカル変数 a、a(2)、および a(2,1) は削除されません。

### ObjectScript

```
SET a="food",a(1)="fruit",a(2)="vegetables"
SET a(1,1)="apple",a(1,1,1)="mackintosh",a(1,2)="banana"
SET a(2,1)="artichoke"
WRITE "before KILL:", !
WRITE $DATA(a), " ", $DATA(a(1)), " ", $DATA(a(1,1)), " ", $DATA(a(1,1,1)), " ",
      $DATA(a(2)), " ", $DATA(a(2,1)), !
KILL a(1)
WRITE "after KILL:", !
WRITE $DATA(a), " ", $DATA(a(1)), " ", $DATA(a(1,1)), " ", $DATA(a(1,1,1)), " ",
      $DATA(a(2)), " ", $DATA(a(2,1))
```

以下の例では、排他的 KILL が変数 d および e 以外のローカル変数をすべて削除します。

## ObjectScript

```
SET a=1,b=2,c=3,d=4,e=5
KILL (d,e)
WRITE "a",$DATA(a)," b",$DATA(b)," c",$DATA(c)," d",$DATA(d)," e",$DATA(e)
```

排他的 KILL がオブジェクト変数を削除するので、上記のプログラムはターミナル・セッションから作動しますが、オブジェクト・メソッド内部では作動しないということに注意してください。

以下の例では、包含的 KILL が 2 つプロセス・プライベート・グローバルを削除し、排他的 KILL が変数 d および e 以外のローカル変数をすべて削除します。

## ObjectScript

```
SET ^|a="one",^|b="two",^|c="three"
SET a=1,b=2,c=3,d=4,e=5
KILL ^|a,^|c,(d,e)
WRITE "a",$DATA(a)," b",$DATA(b)," c",$DATA(c)," d",$DATA(d)," e",$DATA(e),!
WRITE " ^|a",$DATA(^|a)," ^|b",$DATA(^|b)," ^|c",$DATA(^|c)
```

## KILL とオブジェクト

オブジェクト・インスタンス参照 (OREF) は、参照カウントを自動的に維持管理します。参照カウントとは、現在オブジェクトを参照している項目の数です。オブジェクトを参照する変数やオブジェクト・プロパティを SET で設定するたびに、オブジェクトの参照カウントは自動的にインクリメントされます。KILL で変数を削除すると、対応するオブジェクト参照カウントはデクリメントされます。この参照カウントが 0 になると、オブジェクトは自動的に消滅します (メモリから削除されます)。オブジェクト参照カウントは、SET コマンドで変数に新しい値が設定されたときや、変数が範囲外になったときもデクリメントされます。

永続オブジェクトの場合は、オブジェクトに対する変更を保存したければ、メモリからオブジェクトを削除する前に %Save() メソッドを呼び出します。%Delete() メソッドは、InterSystems IRIS オブジェクトの格納されたものを削除します。メモリ内のオブジェクトは削除しません。

OREF の詳細は、“[OREF の基本](#)” を参照してください。

## オブジェクト・メソッドを使用した KILL

KILL 式の左側でオブジェクト・メソッドを指定できます。以下の例では %Get() メソッドを指定しています。

## ObjectScript

```
SET obj=##class(test).%New() // Where test is class with a multidimensional property md
SET myarray=[(obj)]
SET index=0,subscript=2
SET myarray.%Get(index).md(subscript)="value"
WRITE $DATA(myarray.%Get(index).md(subscript)),!
KILL myarray.%Get(index).md(subscript)
WRITE $DATA(myarray.%Get(index).md(subscript))
```

## 包含的 KILL

包含的 KILL は、明示的に指定された変数だけを削除します。ローカル変数、プロセス・プライベート・グローバル、およびグローバルを含むリストを指定できます。また、添え字なしと添え字付きの、どちらの変数も指定できます。グローバル変数を削除できるのは、包含的 KILL だけです。

KillRange() メソッドを使用することで、添え字付きのグローバル変数の範囲を削除できます。

## 排他的 KILL

排他的 KILL は、明示的に指定された変数を除く、すべてのローカル変数を削除します。リストされた名前はコンマで区切られます。括弧は必須です。

排他的 KILL を呼び出す際に、例外リストで指定されたローカル変数を定義する必要はありません。

例外リストに含むことができるのは、ローカルの添え字なしの変数名だけです。例えば、複数の添え字ノードを持つ fruitbasket という名前のローカル変数配列がある場合、KILL (^fruitbasket) を指定することで、ローカル変数配列全体を保持することができます。排他的 KILL を使用して、個別の添え字ノードを選択的に保持することはできません。

例外リストには、オブジェクト参照 (OREF) を表すローカル変数を格納できます。例外リストに、オブジェクト参照のプロパティを格納することはできません。

排他的リストでは、プロセス・プライベート・グローバル、グローバル、または特殊変数を指定できません。これらの変数を指定しようとする、<SYNTAX> エラーが返されます。例外リストで指定されていないローカル変数は削除されます。以降このような変数を参照すると、<UNDEFINED> エラーが返されます。排他的 KILL は、プロセス・プライベート・グローバル、グローバル、または特殊変数に対して効力がありません。しかし、システム・オブジェクトによって生成されたローカル変数を削除することはできます。

## KILL を配列で使用する

包含的 KILL を使用して、配列全体、または配列内の選択したノードだけを削除できます。指定された配列は、ローカル変数、プロセス・プライベート・グローバル、またはグローバル変数のいずれかです。

- ローカル変数配列を削除するには、KILL のいずれの形式を使用することもできます。
- ローカル変数配列内の選択されたノードを削除するには、包含的 KILL を使用しなければなりません。
- グローバル変数配列を削除するには、包含的 KILL を使用しなければなりません。
- グローバル変数配列内の選択されたノードを削除するには、包含的 KILL を使用しなければなりません。

添え字ノード付きのグローバル変数に関する詳細は、“[グローバルについての正式な規則](#)”を参照してください。

配列は、包含的 KILL にその名前を指定するだけで削除できます。例えば以下のコマンドでは、グローバル配列 ^fruitbasket とその従属ノードすべてを削除します。

### ObjectScript

```
SET ^fruitbasket(1)="fruit"
SET ^fruitbasket(1,1)="apples"
SET ^fruitbasket(1,2)="oranges"
WRITE "Before KILL:",!
WRITE ^fruitbasket(1)=,$DATA(^fruitbasket(1)),
      ^fruitbasket(1,1)=,$DATA(^fruitbasket(1,1)),
      ^fruitbasket(1,2)=,$DATA(^fruitbasket(1,1)),!
KILL ^fruitbasket
WRITE "After KILL:",!
WRITE ^fruitbasket(1)=,$DATA(^fruitbasket(1)),
      ^fruitbasket(1,1)=,$DATA(^fruitbasket(1,1)),
      ^fruitbasket(1,2)=,$DATA(^fruitbasket(1,1))
```

配列ノードを削除するには、適切な添え字を指定します。例えば以下の KILL コマンドは、添え字 1、および 2 のノードを削除します。

### ObjectScript

```
SET ^fruitbasket(1)="fruit"
SET ^fruitbasket(1,1)="apples"
SET ^fruitbasket(1,2)="oranges"
SET ^fruitbasket(1,2,1)="navel"
SET ^fruitbasket(1,2,2)="mandarin"
WRITE ^fruitbasket(1)," contains ",^fruitbasket(1,1),
      " and ",^fruitbasket(1,2),!
WRITE ^fruitbasket(1,2)," contains ",^fruitbasket(1,2,1),
      " and ",^fruitbasket(1,2,2),!
KILL ^fruitbasket(1,2)
WRITE "1st level node: ",$DATA(^fruitbasket(1)),!
WRITE "2nd level node: ",$DATA(^fruitbasket(1,1)),!
WRITE "Deleted 2nd level node: ",$DATA(^fruitbasket(1,2)),!
WRITE "3rd level node under deleted 2nd: ",$DATA(^fruitbasket(1,2,1)),!
QUIT
```



配列ノードを削除すると、自動的にそのノードに従属するノードすべてと、削除されたノードへのポインタだけを含んでいた直前のノードがすべて削除されます。検出されたノードがその配列の唯一のノードである場合は、ノードと共に配列自体も削除されます。

複数のローカル変数配列を削除するには、上述のとおり、KILL の包含形式と排他形式の両方を使用できます。例えば以下のコマンドは、array1 および array2 を除く、すべてのローカル配列を削除します。

### ObjectScript

```
KILL (array1,array2)
```

複数の配列ノードを削除するには、KILL の包含形式だけしか使用できません。例えば以下のコマンドは、各配列から 1 ノードを削除しながら、指定された 3 つのノードを削除します。

### ObjectScript

```
KILL array1(2,4),array2(3,2),array3(1,7)
```

ノードは同じ配列内でも、他の配列内でもかまいません。

ZKILL コマンドを使用して、指定されたローカル、あるいはグローバル配列ノードを削除する場合もあります。KILL とは異なり、ZKILL は指定されたノードのすべての従属ノードを削除することはできません。

## パラメータ渡しの KILL

パラメータを渡す場合、値は、外部関数、または DO コマンドで呼び出されるサブルーチンに渡されます。外部関数またはサブルーチンに渡される値は、実パラメータ・リストと呼ばれるコンマで区切られたリストで指定します。指定されたそれぞれの値は、その位置に従って、外部関数またはサブルーチンで定義されている仮パラメータ・リストの対応する変数にマップされます。

実パラメータ・リストの指定方法に従って、パラメータ渡しは、値または参照のいずれかによる方法で行われます。この 2 つのパラメータ渡し方法の詳細は、“[パラメータ渡し](#)” を参照してください。

仮パラメータ・リストの変数に KILL を実行すると、それが値によって渡されていたか、参照によって渡されていたかによって結果が異なります。

[値によって変数の受け渡しを行っている](#)場合は、以下のようになります。

- 仮リストの変数に KILL を実行しても、呼び出される関数やサブルーチンのコンテキストの外部には影響はありません。これは、関数またはサブルーチンが呼び出されるときに、InterSystems IRIS が自動的にその時点の対応する実変数の値を保存しているからです。関数、またはサブルーチンを終了すると、保存された値が自動的に復元されます。

以下の値による受け渡しの例では、Subrt1 で実行する KILL によって、仮変数 x は削除されますが、実変数 a は影響を受けません。

### ObjectScript

```
Test
SET a=17
WRITE !,"Before Subrt1 a: ", $DATA(a)
DO Subrt1(a)
WRITE !,"After Subrt1 a: ", $DATA(a)
QUIT
Subrt1(x)
WRITE !,"pre-kill x: ", $DATA(x)
KILL x
WRITE !,"post-kill x: ", $DATA(x)
QUIT
```

[参照によって変数の受け渡しを行っている](#)場合は、以下のようになります。



- ・ 参照による受け渡しを行っている場合、仮リストの変数を包含して KILL を実行すると、対応する実変数にも KILL が実行されます。関数またはサブルーチンが終了すると、実変数も存在なくなります。
- ・ 仮リストの変数を除外して KILL を実行すると、参照により受け渡された仮変数と実変数は共に保存されます。

以下の参照による受け渡しの例では、Subrtl で実行する KILL によって、仮変数 x と実変数 a の両方が削除されます。

## ObjectScript

```
Test
SET a=17
WRITE !,"Before Subrtl a: ", $DATA(a)
DO Subrtl(.a)
WRITE !,"After Subrtl a: ", $DATA(a)
QUIT
Subrtl(&x)
WRITE !,"pre-kill x: ", $DATA(x)
KILL x
WRITE !,"post-kill x: ", $DATA(x)
QUIT
```

一般的には、仮パラメータ・リストに指定されている変数に KILL を実行すべきではありません。InterSystems IRIS は、パラメータ渡し(値による場合でも、参照による場合でも)を使用する関数、またはサブルーチンに遭遇すると、仮リストのそれぞれの変数に対して、NEW コマンドを暗黙的に実行します。Cache は、関数またはサブルーチンを終了すると、仮リストのそれぞれの変数に対して、今度は暗黙の KILL コマンドを実行します。参照による受け渡しを使用する仮変数の場合は、KILL を実行する前に、(仮変数に対して行われた変更を反映するために) 対応する実変数を更新します。

## KILL、トランザクション、およびロールバック

グローバル変数の KILL はジャーナリングされます。つまり、トランザクション内でこのアクティビティが生じてそのトランザクションがロールバックされると、このグローバル変数の削除がロールバックされます。一方、ローカル変数またはプロセス・プライベート・グローバル変数の KILL はジャーナリングされません。したがって、この変数の削除はトランザクション・ロールバックの影響を受けません。["TROLLBACK \(ObjectScript\)"](#) を参照してください。

## KILL、トランザクション、大規模なグローバル

トランザクション内でグローバル変数に KILL を実行すると、ロールバックが生じた場合に使用できるように、ジャーナル・ファイルに各ノードの古い値が記録されます。これにより、トランザクションに KILL が設定されていない場合と比べて、ジャーナル・ファイルのサイズが大きくなります。このグローバルが非常に大きい場合、ジャーナル・ファイルが大きくなる可能性があるうえ、ジャーナル・ファイルの書き込みアクティビティによってシステムの速度が低下することがあります。このため、トランザクション内で非常に大規模なグローバルの KILL を実行することは避けてください。

## 関連項目

- ・ [ZKILL](#) コマンド
- ・ [\\$DATA](#) 関数
- ・ [\\$STORAGE](#) 特殊変数

## LOCK (ObjectScript)

データ・リソースへのアクセスを制御するために、プロセスがロックを適用および解除できるようにします。

### 構文

```
LOCK:pc
L:pc

LOCK:pc +lockname#locktype:timeout,...
L:pc +lockname#locktype:timeout,...

LOCK:pc +(lockname#locktype,...):timeout,...
L:pc +(lockname#locktype,...):timeout,...
```

### 引数

引数	説明
pc	オプション - 後置条件式。
+ -	オプション - ロックを適用または削除する <b>ロック・オペレーション・インジケータ</b> (1 つの + 文字、- 文字、または文字なし) です。+ (プラス記号) は、既存のどのロックもアンロックせずに、指定されたロックを適用します。これは、 <b>増分ロック</b> を適用するために使用できます。- (マイナス記号) は、ロックをアンロック (またはデクリメント) します。このロック・オペレーション・インジケータを省略すると (文字なし)、InterSystems IRIS では、既存のすべてのロックをアンロックし、指定されたロックを適用します。
lockname	ロック、またはロック解除されるリソースと関連付けられている <b>ロック名</b> 。ローカル変数またはグローバル変数と同じ名前付け規約に従った正当な識別子でなければなりません。
#locktype	オプション - ロックまたはアンロックするロックのタイプを指定する <b>文字コード</b> であり、引用符で囲んで指定します。使用可能な値は、“S” (共有ロック)、“E” (エスカレート・ロック)、“I” (即時アンロック) および “D” (遅延アンロック) です。指定する場合、先頭の # 記号は必須です。例えば、#"S" となります。複数の文字コードを指定することができます。例えば、#"SEI" と指定できます。“S” および “E” は、ロックとアンロックのどちらのオペレーションにも指定されます。“I” および “D” は、アンロック・オペレーションのみに指定されます。省略されると、ロックのタイプは既定の排他ロックになり (非 S)、エスカレートせず (非 E)、常に現在のトランザクションの最後まで、アンロックされたロックの解除を延期します (非 I / 非 D)。
:timeout	オプション - 試行したロック・オペレーションがタイムアウトする前に待機する時間。 <b>timeout</b> は、小数部分が付かない秒数、小数部分が付く秒数、または 1/100 秒までの小数部として指定します (s、s.ff、または .ff)。オプションの #locktype の有無に関係なく指定できます。timeout を指定する場合、先頭の : 記号は必須です。例えば、LOCK ^a(1):10 または LOCK ^a(1)#"E":10 のようにします。値 0 を指定すると、1 回試行した後、タイムアウトします。1/100 秒未満の値は 0 として処理されます。省略した場合、InterSystems IRIS は無限に待機します。

### 概要

LOCK コマンドには、以下の 2 つの基本的な形式があります。

- ・ [引数なし](#)
- ・ [引数付き](#)

## 引数なしの LOCK

引数なしの LOCK は、すべてのネームスペースでプロセスによって現在保持されているロックをすべて解除 (アンロック) します。これにはローカルとグローバルの両方の、排他ロックと共有ロックが含まれます。また、累積した増分ロックもすべて含まれます。例えば、任意のロック名に 3 つの増分ロックが設定されている場合、InterSystems IRIS は 3 つのロックすべてを解除し、ロック・テーブルからそのロック名のエントリを削除します。

トランザクション中に引数なし LOCK を発行すると、InterSystems IRIS は現在プロセスにより保持されているすべてのロックを、トランザクションの終わりまでロック解除状態にします。トランザクションが終了すると、InterSystems IRIS はロックを解除し、対応するロック名のエントリをロック・テーブルから削除します。

以下の例は、トランザクション中のさまざまなロックに適用され、それらすべてのロックを解除するために引数なし LOCK を発行します。ロックは、トランザクションの終了までロック解除状態になります。HANG コマンドを実行すると、ロック・テーブルにあるロックの ModeCount を確認する時間を取ることができます。

### ObjectScript

```
TSTART
LOCK +^a(1)          // ModeCount: Exclusive
HANG 2
LOCK +^a(1)#"E"       // ModeCount: Exclusive/1+1e
HANG 2
LOCK +^a(1)#"S"       // ModeCount: Exclusive/1+1e,Shared
HANG 2
LOCK                  // ModeCount: Exclusive/1+1e->Delock,Shared->Delock
HANG 10
TCOMMIT               // ModeCount: locks removed from table
```

引数なし LOCK は、プロセスにより保持されているすべてのロックを、ロックを適用せずに解除します。プロセスを完了することでも、そのプロセスに保持されているすべてのロックを解除することができます。

## 引数付きの LOCK

引数付きの LOCK は、ロックとアンロックのオペレーションを行う、1 つまたは複数のロック名を指定します。InterSystems IRIS が行うロック・オペレーションは、使用する [ロック・オペレーション・インジケータ](#) 引数により異なります。

- LOCK lockname は、すべてのネームスペースでプロセスにより事前に保持されていたすべてのロックをアンロックしてから、指定されたロック名にロックを適用します。
- LOCK +lockname は、前のすべてのロックをアンロックすることなく、指定されたロック名にロックを適用します。これにより、さまざまなロックを累積させて、[増分ロック](#)を同じロックに適用できます。
- LOCK -lockname は、アンロック・オペレーションを、指定されたロック名に対して実行します。アンロックは、指定されたロック名に対するロック・カウントをデクリメントします。このロック・カウントが 0 までデクリメントされると、ロックは解除されます。

ロック・オペレーションは、即時にロックを適用する場合もあれば、別のプロセスによる競合ロックが解除されるまでロック要求を待機キューで保留にする場合もあります。待機中のロック要求は ([タイムアウト](#)が指定されていれば) タイムアウトする場合もあり、無制限に (プロセスの終了まで) 待機する場合もあります。

### 複数のロック名が指定された LOCK

1 つの LOCK コマンドで複数のロックを指定する場合は、以下の 2 とおりの方法を使用できます。

- 括弧なし : コンマ区切りリストとして複数の括弧なしのロック引数を指定すると、それぞれが独自の [タイムアウト](#) を持つ複数の非依存ロック・オペレーションを指定できます。(機能的には、この方法は、ロック引数ごとに個々の LOCK コマンドを指定する方法と同じです。) ロック・オペレーションは、厳密に左から右の順で実行されます。以下はその例です。

### ObjectScript

```
LOCK var1(1):10,+var2(1):15
```

括弧なしの複数のロック引数はそれぞれ、独自のロック・オペレーション・インジケータと、独自のタイムアウト引数を持つことができます。しかし、複数のロック引数を使用する場合、プラス記号ロック・オペレーション・インジケータなしのロック・オペレーションは、同じ LOCK コマンドの初期部分で適用されたロックを含む、既存のすべてのロックをアンロックします。例えば、コマンド LOCK `b(1,1), ^c(1,2,3), ^d(1) は、次のような 3 つの独立したロック・コマンドとして解析されます。1 つ目は、プロセスが事前に保持していたロックがあればそれを解除して `b(1,1) をロックし、2 つ目は即時に `b(1,1) を解除して ^c(1,2,3) をロックし、3 つ目は即時に ^c(1,2,3) を解除して ^d(1) をロックします。結果として、^d(1) のみがロックされます。

- 括弧付き：ロック名のコンマ区切りリストを括弧で囲むことで、複数のロックに対するロック・オペレーションを単独のアトミック処理として実行することができます。例えば、以下のようにすることができます。

### ObjectScript

```
LOCK +(var1(1),var2(1)):10
```

括弧内にリストされているすべてのロック・オペレーションは、単一のロック・オペレーション・インジケータと単一のタイムアウト引数によって管理されています。すべてのロックが適用されるか、またはいずれのロックも適用されないかのいずれかになります。プラス記号ロック・オペレーション・インジケータなしの、括弧で囲まれたリストは、既存のすべてのロックをアンロックし、リストされているすべてのロック名をロックします。

1 つの LOCK コマンド内のロック名の最大数は、複数の要因によって制限されます。その 1 つが、1 つのプロセスで利用できる引数スタックの数 (512) です。各ロック参照には、4 つの引数スタックと、添え字レベルごとに 1 つの追加の引数スタックが必要です。したがって、ロック参照に添え字がない場合、ロック名の最大数は 127 になります。ロックに 1 つの添え字レベルがある場合、ロック名の最大数は 101 になります。これはおおよその目安と考えてください。1 つの LOCK 内のロック名数は、他の要因によってさらに制限される場合があります。リモート・システムのロックの数に別の制限はありません。

## 引数

### pc

コマンドを条件付きにする、オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に LOCK コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。引数なしの LOCK コマンドまたは引数付きの LOCK コマンドには、後置条件式を指定できます。詳細は、“[コマンド後置条件式](#)” を参照してください。

### lock operation indicator

ロック・オペレーション・インジケータは、ロックの適用 (ロック) または削除 (アンロック) に使用されます。これは以下のいずれかの値を指定できます。

値	説明
なし	現在のプロセスに属している既存のすべてのロックをアンロックし、指定されたロックの適用を試みます。例えば、LOCK ^a(1) は、プロセスにより事前に保持されていたすべてのロックを（ローカルまたはグローバル、排他または共有、エスカレートまたは非エスカレートのどちらであっても）解除し、^a(1) のロックを試みるという、アトミック処理を実行します。これは次の 2 つの結果のいずれかになります。(1) 既存のすべてのロックはアンロックされ、^a(1) はロックされます。(2) 既存のすべてのロックはアンロックされ、^a(1) は別のプロセスによる競合ロックが解除されるまでロック待機状態になります。
プラス記号 (+)	アンロックを実行せずに、指定されたロックの適用を試みます。これにより、現在のプロセスにより保持されているロックに、ロックを追加できます。このオプションの使用法の 1 つとして、ロック名の <a href="#">増分ロック</a> の実行があります。
マイナス記号 (-)	指定されたロックをアンロックします。ロック名が 1 のロック・カウントを持つ場合、アンロックによりそのロックはロック・テーブルから削除されます。ロック名が 2 以上のロック・カウントを持つ場合、アンロックにより増分ロックの 1 つが削除されます（ロック・カウントをデクリメント）。既定では、このアンロックは排他的非エスカレート・ロックです。共有ロック/エスカレート・ロックの一方または両方を削除するには、対応する <a href="#">#locktype</a> を指定する必要があります。

LOCK コマンドに、複数のコンマ区切りのロック引数が含まれる場合、各ロック引数は、独自のロック・オペレーション・インジケータを持つことができます。InterSystems IRIS はこれを、複数の非依存 LOCK コマンドとして解析します。

## lockname

lockname はデータ・リソースのロックの名前であり、データ・リソースそのものではありません。つまり、プログラムは ^a(1) という名前のロックと、^a(1) という変数を競合なく使用できます。ロックとデータ・リソースの関係は、プログラミングの規約です。規約により、プロセスは対応するデータ・リソースを変更する前にロックを獲得する必要があります。

ロック名は、大文字と小文字を区別します。ロック名は、対応するローカル変数およびグローバル変数と同じ命名規則に従います。ロック名は、添え字付きでも添え字なしでもかまいません。ロックの添え字は、[変数の添え字](#)と同じ命名規則、最大長、およびレベル数を持ちます。InterSystems IRIS では、a、a(1)、A(1)、^a、^a(1,2)、^A(1,1,1) はすべて、有効で一意なロック名です。詳細は、“[変数](#)”を参照してください。

**注釈** パフォーマンス上の理由から、可能な場合はいつでも、添え字を使用したロック名を指定することをお勧めします。例えば、^a(1) よりも ^a とします。添え字付きのロック名は、ドキュメントの例で使用されています。

ロック名は、ローカル、グローバルのどちらにもできます。A(1) などのロック名は、ローカル・ロック名です。これは、使用しているプロセスだけに適用されますが、ネームスペース全体に適用されます。キャレット記号 (^) で開始するロック名は、グローバル・ロック名です。このロックに対するマッピングは、対応するグローバルと同じマッピングに従います。つまり、プロセス全体に適用でき、同じリソースへのプロセスのアクセスを制御するということです (“[グローバルについての正式な規則](#)”を参照してください)。

**注釈** プロセス・プライベート・グローバル名は、ロック名として使用できません。プロセス・プライベート・グローバル名をロック名として使用しようとすると、エラーは発生しませんが、何も処理が実行されずに終了します。

ロック名はローカル、またはグローバル変数で、添え字付き、または添え字なしで表されます。また、暗黙のグローバル参照、または他のコンピュータのグローバルへの拡張参照も可能です (“[拡張グローバル参照](#)”を参照してください)。

ロック名に対応するデータ・リソースが存在する必要はありません。例えば、同じ名前を持つグローバル変数の有無に関係なく、ロック名 ^a(1,2,3) をロックできます。ロックとデータ・リソースの関係は、合意された規約であるため、ロックはまったく異なる名前を持つデータ・リソースを保護するために使用できます。



## locktype

適用または削除するロックのタイプを指定する文字コードです。locktype はオプションの引数です。locktype を省略する場合、ロック・タイプは既定では排他・非エスカレート・ロックになります。locktype を省略する場合、ポンド記号 (#) 接頭語を省略する必要があります。locktype を指定する場合、ロック・タイプの構文は、必須のポンド記号 (#) に、1 つ以上のロック・タイプ文字コードを囲む引用符が続きます。ロック・タイプ文字コードは順不同で指定でき、大文字と小文字の区別をしません。以下は、ロック・タイプ文字コードです。

- ・ S : 共有ロック

同じリソース上で、競合しないロックを同時に保持するための複数のプロセスを許可します。例えば、2 つ (またはそれ以上) のプロセスは、同じリソース上で共有ロックを同時に保持しますが、排他ロックはリソースを 1 つのプロセスに制限します。既存の共有ロックがあると、他のすべてのプロセスの排他ロックは適用できなくなります。また既存の排他ロックがあると、他のプロセスの共有ロックは適用できなくなります。しかし、プロセスでまずリソースに共有ロックを適用し、その同じプロセスでリソースに排他ロックを適用することで、ロックを共有から排他にアップグレードできます。共有と排他ロックのカウントは、独立しています。したがって、そのようなリソースを解除するためには、排他ロックと共有ロックの両方を解除する必要があります。共有 (“S”) として指定されていないすべてのロックとアンロックのオペレーションは、排他が既定です。

共有ロックはインクリメントである場合があります。つまり、同じリソースにおいて複数の共有ロックがプロセスによって発行される場合があります。ロックおよびアンロック時に、共有ロックをエスカレート (“SE”) として指定できます。共有ロックのアンロック時に、アンロックを即時 (“SI”) または遅延 (“SD”) として指定できます。エスカレート・ロックおよび非エスカレート・ロックのタイプにおけるインクリメント数と共に現在の共有ロックを表示するには、“[ロック管理](#)” で説明されているシステム全体のロック・テーブルを参照してください。

- ・ E : エスカレート・ロック

ロック・テーブルをオーバーフローさせることなく、同時ロックの最大数を適用できます。既定では、ロックは非エスカレートです。ロックの適用時に、locktype “E” を使用して、そのロックをエスカレートとして指定できます。エスカレート・ロックのリリース時には、アンロック文で locktype “E” を指定する必要があります。排他ロックと共有 (“S”) ロックの両方をエスカレートと指定できます。

一般に、同じ添え字レベルで多数の同時ロックを適用する場合に、エスカレート・ロックを使用することになります。例えば、LOCK +^mylock(1,1)#"E", +^mylock(1,2)#"E", +^mylock(1,3)#"E"... のようになります。

同じロックを、非エスカレート・ロックおよびエスカレート・ロックとして同時に適用することができます。例えば、^mylock(1,1) や ^mylock(1,1)#"E" のようにすることができます。InterSystems IRIS は、ロック・テーブルで locktype “E” を指定して発行されたロックを個別にカウントします。ロック・テーブルでエスカレート・ロックおよび非エスカレート・ロックがどのように表示されるかについての詳細は、“[ロック管理](#)” を参照してください。

添え字レベルの “E” ロックの数がしきい値に達すると、その添え字レベルで要求される次の “E” ロックは親ノード (次に高位の添え字レベル) を自動的にロックしようとします。不可能な場合は、エスカレーションは発生しません。親ノードのロックに成功すると、1 つの親ノードのロックが下位の添え字レベルのロックの数 + 1 に対応するロック・カウントで設定されます。下位の添え字レベルのロックは解除されます。その後の下位の添え字レベルへの “E” ロックの要求は、この親ノードのロックのロック・カウントをさらにインクリメントします。親ノードのロック・カウントを 0 にデクリメントし、下位の添え字レベルにエスカレーション解除するには、適用済みのすべての “E” ロックを解除する必要があります。ロックの既定のしきい値は 1000 ロックです。1001 番目のロックが要求されると、ロックのエスカレーションが発生します。

ロックがエスカレートすると、ロック・オペレーションは、ロックされた特定のリソースではなく、適用されたロックの数のみを保持することに注意してください。そのため、ロックした同じリソースのロック解除に失敗すると、“E” ロックのカウントが同期されなくなります。

以下の例では、プログラムがロックしきい値 + 1 の “E” ロックを適用すると、ロックのエスカレーションが発生します。この例では、ロックのエスカレーションにより、次に高い添え字レベルにおいてロックが適用されると同時に、下位の添え字レベルにおいてロックが解除されることを示しています。

## ObjectScript

```

Main
TSTART
SET thold=$SYSTEM.SQL.Util.GetOption("LockThreshold")
WRITE "lock escalation threshold is ",thold,!
SET almost=thold-5
FOR i=1:1:thold+5 { LOCK +dummy(1,i)#"E"
    IF i>almost {
        IF ^$LOCK("dummy(1,"_i_)", "OWNER") '= "" {WRITE "lower level lock applied at ",i,"th lock
",! }
        ELSEIF ^$LOCK("dummy(1)", "OWNER") '= "" {WRITE "lock escalation",!
WRITE "higher level lock applied at ",i,"th lock
",!
        QUIT }
    }
    ELSE {WRITE "No locks applied",! }
}
}
TCOMMIT

```

ロック・エスカレーションに対してカウントされるのは“E”ロックだけであることに注意してください。以下の例では、既定（非“E”）のロックと“E”ロックの両方を同じ変数に設定します。ロックのエスカレーションは、変数における“E”ロックの合計数がロックしきい値に達した場合にのみ発生します。

## ObjectScript

```

Main
TSTART
SET thold=$SYSTEM.SQL.Util.GetOption("LockThreshold")
WRITE "lock escalation threshold is ",thold,!
SET noE=17
WRITE "setting ",noE," non-escalating locks",!
FOR i=1:1:thold+noE { IF i < noE {LOCK +a(6,i)}
    ELSE {LOCK +a(6,i)#"E"}
    IF ^$LOCK("a(6)", "OWNER") '= "" {
        WRITE "lock escalation on lock a(6,"i,"")",!
        QUIT }
    }
}
TCOMMIT

```

“E”ロックのロック解除は、上記の逆です。ロックがエスカレーションされている場合、子のレベルでのロック解除は、ゼロに達してロック解除されるまで親ノードのロックのロック・カウントをデクリメントします。これらのロック解除はカウントをデクリメントし、特定のロックとは一致しません。親ノードのロック・カウントが 0 に達すると、親ノードのロックが削除され、“E”ロックが下位の添え字レベルへとエスカレーション解除されます。それ以降の下位の添え字レベルのロックは、そのレベルで固有のロックを作成します。

“E” locktype は、他の任意の locktype と組み合わせることができます。例えば、“SE”、“ED”、“EI”、“SED”、“SEI”などです。“I” locktype と組み合わせると、現在のトランザクションの最後ではなく、呼び出された直後に“E”ロックを解除できます。この“EI” locktype は、ロックがエスカレートされる状況を最低限に抑えることができます。

通常、“E”ロックは、トランザクション内の SQL [INSERT](#)、[UPDATE](#)、および [DELETE](#) 処理において自動的に適用されます。ただし、“E”ロックをサポートする SQL データ定義構造について固有の制限事項が存在します。詳細は、固有の SQL コマンドを参照してください。

- ・ I：即時アンロック

トランザクションの終了まで待つのではなく、ロックを即時に解除します。

- 非増分（ロック・カウント 1）ロックのアンロック時に“I”を指定すると、ロックは即時に解除されます。既定では、アンロックは非増分ロックを即時に解除しません。代わりに、非増分ロックをアンロックすると、InterSystems IRIS はそのロックを、トランザクションの終了までロック解除状態に維持します。“I”を指定すると、この既定の動作はオーバーライドされます。
- 増分ロック（ロック・カウント > 1）のアンロック時に“I”を指定すると、増分ロックは即時に解除され、ロックカウントは 1 だけデクリメントされます。これは増分ロックの既定のアンロックと同じ動作です。

“I” locktype は、トランザクション中にアンロックを実行する場合に使用します。これには、ロックがトランザクションの内部または外部のどちらで適用されたかに関係なく、InterSystems IRIS のアンロック動作に対して同じ効果がありま

す。“I” locktype は、アンロックがトランザクションの外部で行われる場合は、オペレーションを実行しません。トランザクションの外部では、アンロックは必ず指定されたロックを即時に解除します。

“I” は、アンロック・オペレーションにのみ指定できます。ロック・オペレーションには指定できません。“I” は、共有ロック（#"SI"）または排他ロック（#"I"）のアンロックに指定できます。Locktypes “I” および “D” は、相互排他的です。“IE” は、エスカレート・ロックの即時アンロックに使用できます。

この即時アンロックを以下の例で示します。

### ObjectScript

```
TSTART
LOCK +^a(1)      // apply lock ^a(1)
LOCK -^a(1)      // remove (unlock) ^a(1)
                  // An unlock without a locktype defers the unlock
                  // of a non-incremented lock to the end of the transaction.
WRITE "Default unlock within a transaction.",!,"Go look at the Lock Table",!
HANG 10          // This HANG allows you to view the current Lock Table
LOCK +^a(1)      // reapply lock ^a(1)
LOCK -^a(1)#"I"  // remove (unlock) lock ^a(1) immediately
                  // this removes ^a(1) from the lock table immediately
                  // without waiting for the end of the transaction
WRITE "Immediate unlock within a transaction.",!,"Go look at the Lock Table",!
HANG 10          // This HANG allows you to view the current Lock Table
                  // while still in the transaction
TCOMMIT
```

#### ・ D：遅延アンロック

アンロックされたロックをトランザクション中に解除するタイミングを制御します。アンロック状態は、そのロックの直前のアンロックの状態まで遅延されます。そのため、ロックをアンロックするときに locktype “D” を指定すると、そのトランザクション中のロックの履歴に応じて、即時アンロックになるか、またはトランザクションの終わりまでロック解除状態になるかのいずれかの結果になります。複数回ロック/アンロックされたロックの動作は、現在のトランザクション中に 1 回のみロックされたロックの動作とは異なります。

“D” アンロックは、ロックを解除するアンロック（ロック・カウント 1）のみに有効であり、ロックをデクリメントするアンロック（ロック・カウント > 1）には無効です。ロックをデクリメントするアンロックは、常に即時に解除されます。

“D” は、アンロック・オペレーションにのみ指定できます。“D” は、共有ロック（#"SD"）または排他ロック（#"D"）に指定できます。“D” は、エスカレート（“E”）ロックに指定できますが、当然ながらアンロックもエスカレート（“ED”）として指定する必要があります。ロック・タイプ “D” と “I” は、相互排他的です。

トランザクション内でのこの “D” アンロックの使用を以下の例に示します。HANG コマンドを実行すると、ロック・テーブルにあるロックの ModeCount を確認する時間を取ることができます。

ロックが現在のトランザクション中に 1 回のみ適用された場合、“D” はロックを即時に解除します。これは “I” の振る舞いと同じです。詳細は、以下の例を参照してください。

### ObjectScript

```
TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
HANG 10
TCOMMIT
```

現在のトランザクション中にロックが複数回適用された場合、“D” アンロックは直前のアンロック状態に戻します。

- 最後のアンロックが標準アンロックであった場合、“D” アンロックは、その直前のアンロックの動作に戻します。つまり、トランザクションの終了までアンロックを遅延します。詳細は、以下の例を参照してください。



## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)      // Lock Table ModeCount: Exclusive
WRITE "1st unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive->Delock
WRITE "2nd unlock",! HANG 5
TCOMMIT

```

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)      // Lock Table ModeCount: Exclusive->Delock
WRITE "1st unlock",! HANG 5
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive->Delock
WRITE "2nd unlock",! HANG 5
TCOMMIT

```

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"I"  // Lock Table ModeCount: Exclusive/2
WRITE "1st unlock",! HANG 5
LOCK -^a(1)      // Lock Table ModeCount: Exclusive
WRITE "2nd unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive->Delock
WRITE "3rd unlock",! HANG 5
TCOMMIT

```

- 最後のアンロックが“I”アンロックであった場合、“D”アンロックは、その直前のアンロックの動作に戻ります。つまり、ロックを即時にアンロックします。詳細は、以下の例を参照してください。

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"I"  // Lock Table ModeCount: null (immediate unlock)
WRITE "1st unlock",! HANG 5
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
WRITE "2nd unlock",! HANG 5
TCOMMIT

```

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"I"  // Lock Table ModeCount: Exclusive
WRITE "1st unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
WRITE "2nd unlock",! HANG 5
TCOMMIT

```

- 最後のアンロックが“D”アンロックだった場合、“D”アンロックは直前の非“D”ロックの動作に従います。

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive
WRITE "1st unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
WRITE "2nd unlock",! HANG 5
TCOMMIT

```

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)      // Lock Table ModeCount: Exclusive/2
    WRITE "1st unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive
    WRITE "2nd unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive->Delock
    WRITE "3rd unlock",! HANG 5
TCOMMIT

```

## ObjectScript

```

TSTART
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK +^a(1)      // Lock Table ModeCount: Exclusive
LOCK -^a(1)#"I"  // Lock Table ModeCount: Exclusive/2
    WRITE "1st unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: Exclusive
    WRITE "2nd unlock",! HANG 5
LOCK -^a(1)#"D"  // Lock Table ModeCount: null (immediate unlock)
    WRITE "3rd unlock",! HANG 5
TCOMMIT

```

## timeout

タイムアウト前に、ロック要求を待機する時間(秒数または小数秒数)です。timeout はオプションの引数です。省略された場合、LOCK コマンドはリソースがロック可能になるまで無制限に待機します。ロックが適用できない場合、プロセスは停止します。timeout の構文には、必須のコロン(:) が最初にあり、その後には数値や、数値に評価される式が続きます。

有効な値は秒で、1/10 秒または 1/100 秒の小数部分はあってもなくてもかまいません。したがって、:5、:5.5、:0.5、:.5、:0.05、:.05 はすべて timeout の値として有効です。:0.01 より小さい値はゼロとして解析されます。ゼロの値は、タイムアウトの前に 1 つのロックの試みを呼び出します。負の数は、ゼロと同様です。

通常、ロック要求が指定ロックを取得する(保持する)のを妨げる競合ロックが別プロセスに存在する場合、ロックは待機します。ロック要求は、ロックが解除されて競合が解決されるか、またはロック要求がタイムアウトになるまで待機します。プロセスを終了しても保留中のロック要求が終了します(削除されます)。ロックの競合は、別プロセスが保持するのと同じロックをあるプロセスが要求する場合だけでなく、さまざまな状況によって発生する場合があります。ロックの競合およびロック要求の待機状態の詳細は、“[ロック管理](#)”を参照してください。

timeout を使用し、ロックに成功した場合、InterSystems IRIS は特殊変数 \$TEST を 1 (TRUE) に設定します。タイムアウト時間内にロックを適用できなかった場合、InterSystems IRIS は \$TEST を 0 (FALSE) に設定します。timeout を指定せずにロック要求を出した場合、\$TEST の現在値には何の影響も及ぼしません。\$TEST は、ユーザが設定することもできます。また、JOB、OPEN、または READ のタイムアウトで設定されることもあります。

以下の例では、ロックをロック名 ^abc(1,1) に適用し、プロセスにより保持されている既存のすべてのロックをアンロックします。

## ObjectScript

```
LOCK ^abc(1,1)
```

このコマンドは、排他ロックを要求します。他のプロセスは、このリソース上のロックを同時に保持することができません。他のプロセスが既にこのリソース上のロックを保持している場合(排他、または共有)、この例では、ロックが解除されるまで待機する必要があります。プロセスを停止しながら、無制限に待機できます。この状態を避けるため、タイムアウト値を指定することを強くお勧めします。

## ObjectScript

```
LOCK ^abc(1,1):10
```

LOCK により、コンマ区切りリストの lockname 引数を複数指定する場合、各 lockname リソースは専用の timeout (括弧なしの構文) を備えるか、または、指定した lockname リソースすべてが単一の timeout (括弧付きの構文) を共有することもできます。

- ・ 括弧なし: 各 lockname 引数には固有の timeout を指定できます。InterSystems IRIS はこれを複数の独立した LOCK コマンドとして解析するので、ロック引数の timeout は他のロック引数に影響を与えません。ロック引数は厳密に左から右の順で解析され、各ロック要求は、次のロック要求が試される前に、完了するかまたはタイムアウトします。
- ・ 括弧付き: すべての lockname 引数で timeout を共有します。LOCK は、タイムアウト時間内にすべてのロック (またはアンロック) を正常に適用する必要があります。すべてのロックが成功する前にタイムアウト時間が経過した場合、LOCK コマンドで指定されたどのロック・オペレーションも実行されず、制御はプロセスに戻ります。

InterSystems IRIS は、複数の演算を厳密に左から右の順に実行します。したがって、括弧なしの LOCK 構文では、\$TEST の値は複数の lockname ロック要求の最後 (右端) の結果を示しています。

以下の例では、現在のプロセスは ^a(1) を、別のプロセスにより排他的にロックされているためロックできません。これらの例では、指定されたロックへの適用を 1 回試みることになる、タイムアウト 0 を使用しています。

最初の例は、^x(1) および ^z(1) をロックします。^z(1) がタイムアウト前に成功したので、これは \$TEST=1 を設定します。

#### ObjectScript

```
LOCK +^x(1):0,+^a(1):0,+^z(1):0
```

2 番目の例は、^x(1) および ^z(1) をロックします。^z(0) がタイムアウトしたので、これは \$TEST=1 を設定します。^z(1) はタイムアウトを指定していなかったため、\$TEST に効果はありません。

#### ObjectScript

```
LOCK +^x(1):0,+^a(1):0,+^z(1)
```

3 番目の例は、括弧で囲まれたロックのリストがアトミック (全か無か) オペレーションであるため、ロックを適用しません。^a(1) がタイムアウトしたので、これは \$TEST=0 を設定します。

#### ObjectScript

```
LOCK +(^x(1),^a(1),^z(1)):0
```

## ロック・テーブルを使用したシステム全体のロックの表示および削除

InterSystems IRIS は、システム全体のロック・テーブルを保持しています。このテーブルには、その時点で有効なロックとそれを適用しているプロセスがすべて記録されています。管理ポータル・インタフェースまたは [LOCKTAB ユーティリティ](#) を使用すれば、システム・マネージャでロック・テーブルの既存ロックを表示したり、選択されたロックを削除することができます (“[ロック管理](#)” を参照してください)。%SYS.LockQuery クラスを使用して、ロック・テーブル情報を読み取ることもできます。%SYS ネームスペースから SYS.Lock クラスを使用することで、ロック・テーブルを管理できます。

管理ポータルを使用することで、システム全体で保持されているロックおよび保留中のロック要求を表示できます。管理ポータルに移動して、[システムオペレーション]→[ロック]→[ロック表示] の順に選択します。[ロック表示] テーブルの詳細は、“[ロック管理](#)” を参照してください。

管理ポータルを使用することで、システムにおいて現在保持されているロックを削除できます。管理ポータルに移動して、[システムオペレーション]→[ロック]→[ロックを管理] の順に選択します。任意のプロセス ([所有者]) について、“[削除]” または “[プロセスが保持するすべてのロックを削除]” をクリックします。

ロックを削除すると、そのロックのすべての形式 (ロックのすべての増分レベル、ロックのすべての排他バージョン、排他エスカレート・バージョン、および共有バージョン) が解除されます。ロックを削除すると、そのロック・キューで待機している次のロックが直ちに適用されます。

ロックは、SYS.Lock.DeleteOneLock() および SYS.Lock.DeleteAllLocks() メソッドを使用しても削除できます。

ロックの削除には、WRITE 権限が必要です。ロックの削除は、監査データベースに記録されます（有効になっている場合）。messages.log には記録されません。

## 増分ロックおよびアンロック

増分ロックでは、同じロックを複数回適用できます。つまりロックをインクリメントします。増分ロックのロック・カウントは > 1 です。その後、プロセスはこのロック・カウントをインクリメント、またはデクリメントすることができます。ロック・カウントが 0 までデクリメントされると、ロックは解除されます。ロック・カウントが 0 までデクリメントされるまで、他のプロセスはロックを獲得できません。ロック・テーブルは、排他ロックと共有ロック、およびその各タイプのエスカレート・ロックと非エスカレート・ロックの、個別のロック・カウントを維持します。最大増分ロック・カウントは 32,766 です。この最大ロック・カウントを超過しようとする、<MAX LOCKS> エラーになります。

ロックは以下のようにインクリメントできます。

- ・ プラス記号：同じロック名に対する複数のロック・オペレーションを、プラス記号ロック・オペレーション・インジケータを付けて指定します。例：LOCK +^a(1) LOCK +^a(1) LOCK +^a(1)、または LOCK +^a(1),+^a(1),+^a(1)、または LOCK +(^a(1),^a(1),^a(1))。このすべては、ロック・テーブル ModeCount の Exclusive/3 という結果になります。ロックを増分するためには、プラス記号を使用することをお勧めします。
- ・ 記号なし：複数のロックを実行するアトミック・オペレーションを指定することで、プラス記号ロック・オペレーション・インジケータを使用しなくても、ロックを増分することは可能です。例えば、LOCK (^a(1),^a(1),^a(1)) は、既存のすべてのロックをアンロックして、^a(1) を 3 回増分ロックします。これも、ロック・テーブル ModeCount の Exclusive/3 という結果になります。この構文は機能しますが、お勧めしません。

トランザクション中でないときに増分ロックをアンロックしても、ロック・カウントがデクリメントされるだけです。トランザクション中に増分ロックをアンロックした場合には、以下のデフォルトの動作になります。

- ・ アンロックのデクリメント：各デクリメント・アンロックは、ロック・カウントが 1 になるまで増分アンロックを即時に解除します。既定では、最終アンロックは、ロックをロック解除状態にして、トランザクションの終了までロックの解除を遅延します。オペレーションがアトミックであるかどうかに関係なく、マイナス記号ロック・オペレーション・インジケータを使用してロック解除するときは、必ずこのようになります。例：LOCK -^a(1) LOCK -^a(1) LOCK -^a(1)、または LOCK -^a(1),-^a(1),-^a(1)、または LOCK -(^a(1),^a(1),^a(1))。このすべてで、ロック・テーブルの先頭は ModeCount の Exclusive/3、末尾は Exclusive->Delock となります。
- ・ 既存のリソースのアンロック：既存のすべてのリソースをアンロックするオペレーションは、トランザクションの終了までに、増分ロックを即時にロック解除状態にします。例えば、LOCK x(3)（ロック・オペレーション・インジケータなしのロック）または引数なし LOCK では、増分ロックでロック・テーブルの先頭は ModeCount の Exclusive/3、末尾は Exclusive/3->Delock となる効果があります。

同じロックでも、排他ロック、共有ロック、排他エスカレート・ロック、および共有エスカレート・ロックには個別のロック・カウントが維持されます。以下の例では、最初のアンロックで、ロック ^a(1) の 4 つの個別ロック・カウントが 1 だけデクリメントされます。2 番目のアンロックでは、4 つすべての ^a(1) ロックでそれらを削除するように指定する必要があります。HANG コマンドを実行すると、ロック・テーブルにあるロックの ModeCount を確認する時間を取ることができます。

### ObjectScript

```
LOCK +(^a(1),^a(1)#"E",^a(1)#"S",^a(1)#"SE")
LOCK +(^a(1),^a(1)#"E",^a(1)#"S",^a(1)#"SE")
HANG 10
LOCK -(^a(1),^a(1)#"E",^a(1)#"S",^a(1)#"SE")
HANG 10
LOCK -(^a(1),^a(1)#"E",^a(1)#"S",^a(1)#"SE")
```

現在適用されているロックがないロック名をアンロックしようとしても、オペレーションは実行されず、エラーは返されません。

## 自動アンロック

プロセスが終了すると、InterSystems IRIS は引数なしの暗黙の LOCK を実行して、そのプロセスが適用していたロックをすべて消去します。これにより保持されていたロックとロック待機要求の両方が削除されます。

## グローバル変数のロック

ロックは、同時に同じ変数にアクセスする可能性のある、複数のプロセスの動作の同期をとるために、グローバル変数で使われるのが一般的です。グローバル変数はディスクに存在し、すべてのプロセスに対して利用可能であるという点で、ローカル変数とは異なります。そのため、2 つのプロセスが、同じグローバル変数に同時に書き込みを行う可能性があります。実際には、InterSystems IRIS は、更新を 1 つずつ処理します。したがって、ある更新によって、その次に更新が予定されていたものが上書きされ、実質的に、破棄されたりすることがあります。

グローバル・ロック名は ` 文字で開始します。

グローバル変数のロックを分かりやすく説明するために、2 人のデータ入力者が、新しく入学した学生のレコードを追加するために、同時に同じ学生入学アプリケーションを実行している場合を考えてみましょう。レコードは、`student という名前のグローバル配列に格納されています。学生 1 人 1 人に一意のレコードを確保するために、アプリケーションでは、学生が追加されるたびに、グローバル変数 `index がインクリメントされます。各学生レコードが配列の一意の場所に追加されること、およびある学生レコードが他のレコードを上書きしないことを保証するために、アプリケーションには LOCK コマンドが含まれています。

このアプリケーションに関連するコードを以下に示します。この場合 LOCK はグローバル配列 `student ではなく、グローバル変数 `index を制御します。`index は、2 つのプロセスによって共用されるスクラッチ・グローバルです。プロセスは `index をロックし、その現在の値を更新 (SET `index=`index+1) して初めて、配列にレコードの書き込みができます。他のプロセスが既にコードのこのセクションに入っている場合は、`index がロックされているので、そのプロセスが (引数なしの LOCK コマンドで) ロックを解除するまで、次のプロセスは待機しなければなりません。

### ObjectScript

```
READ !,"Last name: ",!,lname QUIT:lname="" SET lname=lname_","
READ !,"First name: ",!,fname QUIT:fname="" SET fname=fname_","
READ !,"Middle initial: ",!,minit QUIT:minit="" SET minit=minit_":"
READ !,"Student ID Number: ",!,sid QUIT:sid=""
SET rec = lname_fname_minit_sid
LOCK `index
SET `index = `index + 1
SET `student(`index)=rec
LOCK
```

以下の例では、前述の例を書き直し、`student 配列に追加されるノードにロックを設定するようにしています。関連部分のコードだけを示します。この場合、変数 `index は、新しい学生レコードが追加された後、更新されます。レコードを追加する次のプロセスは、更新された index 値を使用して、現在の配列ノードに書き込みを行います。

### ObjectScript

```
LOCK `student(`index)
SET `student(`index) = rec
SET `index = `index + 1
LOCK /* release all locks */
```

配列ノードのロックの位置は、トップ・レベルのグローバルがマップされている位置になることに注意してください。InterSystems IRIS は、ロックの位置の決定時に添え字を無視します。したがって、`student(name) のデータが格納されている場所に関係なく、`student(name) は `student のネームスペースにマップされます。

## ネットワークにおけるロック

ネットワーク・システムでは、1 つ、または複数のサーバがグローバル変数のロックを解決する役割を果たすことができます。

LOCK コマンドは、最大 255 までの、任意の数のサーバで使用できます。

[^\\$LOCK](#) を使用してリモート・ロックをリストできますが、リモート・ロックのロック状態はリストできません。

リモート・サーバ・システム上のクライアント・ジョブにより保持されるリモート・ロックは、クライアント・ジョブを削除するために [^RESJOB](#) ユーティリティを呼び出すときに解除されます。

## ローカル変数ロック

動作は以下のとおりです。

- ・ 特定のネームスペースのコンテキストで取得されるローカル (^ が付いていない) ・ロックは、ローカル・マシンのマネージャのデータセットから取得されます。これは、既定のネームスペースが、明示的なネームスペースであるか、またはネームスペースの明示的な参照によるためです。この場合、グローバルに対する既定のマッピングがローカル・データセットか、リモート・データセットかは関係ありません。
- ・ 暗黙のネームスペースのコンテキストで、またはローカル・マシンの暗黙のネームスペースへの明示的な参照を通じて取得されたローカル (^ が付いていない) ・ロックは、ローカル・マシンのマネージャのデータセットを使用して取得されます。[暗黙のネームスペース](#)は、"^^dir" のように 2 つのキャレット文字が先行するディレクトリ・パスです。

明示的または暗黙的なネームスペースの参照に関する詳細は、"[グローバルについての正式な規則](#)"を参照してください。

## 関連項目

- ・ [\\$TEST](#) 特殊変数
- ・ [^\\$LOCK](#) 構造化システム変数
- ・ [ロックと並行処理の制御](#)
- ・ [ロック管理](#)
- ・ [トランザクション処理での ObjectScript の使用法](#)
- ・ [ロックの監視](#)



## MERGE (ObjectScript)

グローバル・ノード、もしくはサブツリーをソースから宛先にマージします。

### 構文

```
MERGE:pc mergeargument,...
M:pc mergeargument,...
```

mergeargument には、以下を指定できます。

*destination=source*

### 引数

引数	説明
pc	オプション - 後置条件式
destination および source	マージされるローカル変数、プロセス・プライベート・グローバル、またはグローバル。クラス・プロパティとして指定された場合、source 変数は、多次元の (添え字付き) 変数である必要があります。

### 概要

MERGE destination=source は source を destination にコピーし、source のすべての下位を destination の下位にコピーします。これは、source を変更したり、destination のノードを削除することはありません。

MERGE は、ある変数のサブツリー (複数の添え字) を別の変数にコピーします。変数は、添え字付きのローカル変数、プロセス・プライベート・グローバル、またはグローバルのいずれかです。サブツリーとは、特定の変数の子孫であるすべての変数です。

MERGE は、source と destination が親子リレーションシップである場合に <COMMAND> エラーを発行します。

### MERGE の実行

MERGE はアトミック処理ではありません。

MERGE コマンドの実行には、他のほとんどの ObjectScript コマンドよりも時間がかかります。その結果、割り込みを受ける可能性も高くなります。割り込みによって、コピー先のサブツリーに、ソースから予測できないサブセットがコピーされる可能性があります。

他のプロセスが同時にデータ変更操作を実行中に MERGE を実行した場合、destination の内容は、MERGE の開始時点でのデータの状態から、MERGE 操作の終了時に KILL で削除された変数を除いたものになります。MERGE 処理中に行われた他のデータ変更は、destination の内容に反映されない場合があります。

### 引数

#### pc

オプションの後置条件式。InterSystems IRIS® データ・プラットフォームは、後置条件式が True (0 以外の数値に評価される) の場合に MERGE コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

## destination および source

マージされる変数。変数は、ローカル変数、プロセス・プライベート・グローバル、またはグローバルのいずれかです。destination が未定義の場合、MERGE は定義を行い、source に設定します。source が未定義の場合、MERGE は正常に終了しますが、destination を変更しません。

ユーザは、複数のコンマ区切りの destination=source の組み合わせを指定することができます。これらは、左から右の順に評価されます。

mergeargument は、destination=source ペアを [間接演算](#) で参照できます。例：MERGE @tMergeString

mergeargument には、myargs... など [可変個数のパラメータ](#) を指定する参照によって渡される配列を指定できます。

`^$GLOBAL` SSVN には source 変数を指定できます。これにより、グローバル・ディレクトリが destination 変数にコピーされます。MERGE は、各グローバル名を、NULL 値を持つ destination 添え字として追加します。以下に例を示します。

### ObjectScript

```
MERGE gbls=^$GLOBAL(" ")
ZWRITE gbls
```

## 例

以下の例では、グローバル変数 (^a) から別のグローバル変数 (^b) にサブツリーをコピーします。この場合は、マージの使用によって、より小さいグローバル ^b が作成され、これには ^a にある情報の ^a(1,1) サブツリーのみが含まれます。

### ObjectScript

```
SET ^a="cartoons"
SET ^a(1)="The Flintstones",^a(2)="The Simpsons"
SET ^a(1,1)="characters",^a(1,2)="place names"
SET ^a(1,1,1)="Flintstone family"
SET ^a(1,1,1,1)="Fred"
SET ^a(1,1,1,2)="Wilma"
SET ^a(1,1,2)="Rubble family"
SET ^a(1,1,2,1)="Barney"
SET ^a(1,1,2,2)="Betty"
MERGE ^b=^a(1,1)
WRITE ^b,! ,^b(2),! ,^b(2,1), " and ",^b(2,2)
```

以下の例では、コピー元のグローバル変数のサブツリーが結合された後の、コピー先のグローバル変数を示しています。

以下を実行するとします。

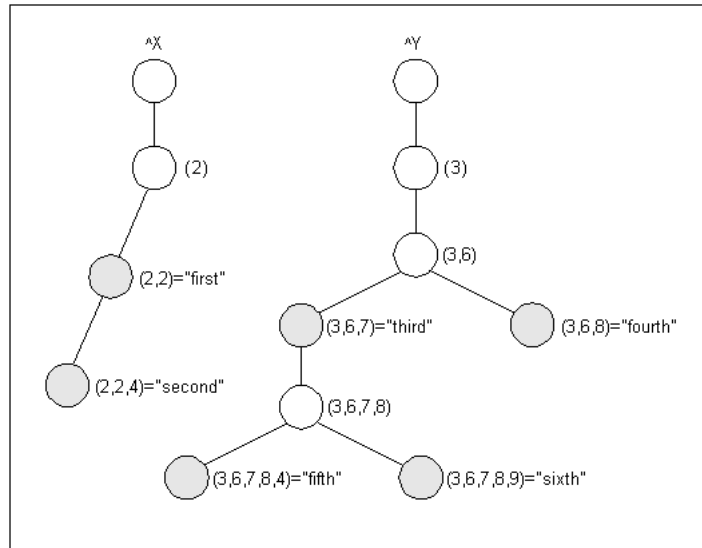
### ObjectScript

```
KILL ^X,^Y
SET ^X(2,2)="first"
SET ^X(2,2,4)="second"
SET ^Y(3,6,7)="third"
SET ^Y(3,6,8)="fourth"
SET ^Y(3,6,7,8,4)="fifth"
SET ^Y(3,6,7,8,9)="sixth"
WRITE ^X(2,2),! ,^X(2,2,4),!
WRITE ^Y(3,6,7),! ,^Y(3,6,8),!
WRITE ^Y(3,6,7,8,4),! ,^Y(3,6,7,8,9)
```

以下の図は、実行後の ^X と ^Y の論理構造を示しています。



図 C-2: ^X と ^Y の初期構造



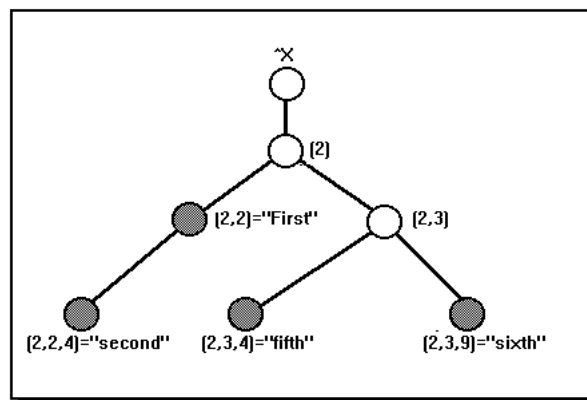
以下の MERGE コマンドを考えてみましょう。

### ObjectScript

```
MERGE ^X(2,3)=^Y(3,6,7,8)
```

前の文を実行すると、InterSystems IRIS は ^Y の一部を ^X(2,3) にコピーします。グローバル ^X は、以下の図のような構造になります。

図 C-3: ^X および ^Y の MERGE コマンドの結果



## ネイキッド・インジケータ

destination および source の両方がローカル変数である場合、ネイキッド・インジケータは変更されません。source がグローバル変数で、destination がローカル変数の場合は、ネイキッド・インジケータは source を参照します。

source および destination の両方がグローバル変数である場合、ネイキッド・インジケータの状態は、source が未定義 (\$DATA(source)=0) であれば変更されません。これ以外の場合 (\$DATA(source)=10 の場合)、ネイキッド・インジケータは、SET コマンドを MERGE コマンドと置き換え、source が値を持っているとして実行した場合と同じ値を取ります。ネイキッド・インジケータの詳細は、[最新のグローバル参照のチェック](#) を参照してください。

## 自己マージ

destination と source が同じ変数のとき、マージは行われず、ジャーナル・ファイルにも何も記録されません。しかし以前のセクションで説明された規定に基づいて、ネイキッド・インジケータは変更されます。

## ウォッチポイント

MERGE コマンドはウォッチポイントをサポートしています。ウォッチポイントが有効になっている場合、MERGE が監視対象変数の値を変更するたびに、InterSystems IRIS はウォッチポイントをトリガします。ウォッチポイントを設定するには、ZBREAK コマンドを使用します。

## 関連項目

- ・ [ZBREAK コマンド](#)
- ・ [コマンド行ルーチンのデバッグ](#)
- ・ [グローバルについての正式な規則](#)

# NEW (ObjectScript)

空のローカル変数環境を作成します。

## 構文

```
NEW:pc newargument,...
N:pc newargument,...
```

newargument には、以下を指定できます。

```
variable,...
(variable,...)
```

## 引数

引数	説明
pc	オプション - 後置条件式。
variable	オプション - 既存のローカル変数環境に追加される変数名。既存のローカル変数での NEW の結果は、variable が括弧に囲まれているか (排他的 NEW)、または括弧に囲まれていないか (包含的 NEW) によって異なります。variable は、有効なローカル変数名でなければなりませんが、定義済みの変数である必要はありません。未定義の変数を指定しても、エラーが出力されたり、変数が定義されたりすることはありません。

## 概要

NEW コマンドには、以下の 3 つの形式があります。

- ・ 引数なし
- ・ 引数あり：包含的 NEW (括弧なし)
- ・ 引数あり：排他的 NEW (括弧あり)

引数なしの NEW コマンドは、呼び出されるサブルーチンまたは外部関数用に、空のローカル変数環境を作成します。既存のローカル変数値は、新規のローカル環境では利用できません。この値は、以前のローカル環境に戻ることによって復元されます。

引数付きの NEW コマンドのアクションは、使用する引数の形式によって異なります。

- ・ NEW var1,var2,... (包含的 NEW) は、既存のローカル変数環境を保持し、それに指定された変数を追加します。指定されたローカル変数のいずれかが、既存のローカル変数と同じ名前を持つ場合、その名前を持つ変数の古い値は、現在の環境ではアクセス不可能となります。
- ・ NEW (var1,var2,...) (排他的 NEW) は、指定された変数を除く既存のすべての変数を、ローカル変数環境で置き換えます。

## 引数なしの NEW

引数なしの NEW は、呼び出されるサブルーチン、または外部関数用に空のローカル変数環境を提供します。既存のローカル変数環境 (呼び出し元のルーチン内) は保存され、サブルーチンまたは関数の終了時に復元されます。NEW の後に作成された変数は、サブルーチンまたは関数の終了時にすべて削除されます。

同じ行で NEW の後にコマンドが続く場合、(最低でも) 2 つのスペースで以下のコマンドと NEW を区別する必要があります。

引数なしの NEW は、FOR ループ本文内または InterSystems IRIS オブジェクトに影響を与える可能性のあるコンテキスト内では使用できません。

31 レベルを超える排他的 NEW コマンドや引数なしの NEW コマンドの発行を試みると、〈MAXSCOPE〉エラーが返されます。

## 包含的 NEW

包含的 NEW — NEW var1,var2 は、既存のローカル変数環境を保持し、それに指定された変数を追加します。既存の変数が指定された場合、“NEW” 変数によって、既存の変数が置き換えられます。既存の変数は、スタックに保存され、サブルーチンまたは関数の終了後に復元されます。

包含的 NEW では、コンマ区切りのリストとして指定できる変数の数は制限されません。また、ローカル変数環境レベルの数 (NEW コマンドの発行回数) も制限されません。

以下の例では、呼び出し元のルーチン (Main) のローカル変数環境が、変数 a、b、c で構成されていると仮定しています。DO が Subr1 を呼び出すと、NEW コマンドが Subr1 のローカル変数環境を再定義し、c を新しい変数にして変数 d を追加します。NEW の後、サブルーチンの環境は、既存の変数 a および b に新しい変数 c および d を加えて構成されています。変数 a および b は、継承されたもので、既存の値を保持しています。新しい変数 c および d は、未定義で作成されます。c は既存のローカル変数の名前であるため、システムは既存の値をスタックに格納し、Subr1 が QUIT を実行するときに c を復元します。Subr1 の最初の SET コマンドは a および b を参照して、d に値を代入していることに注意してください。このコンテキストで、変数 c は未定義です。

### ObjectScript

```
Main
  SET a=2,b=4,c=6
  WRITE !,"c in Main before DO: ",c
  DO Subr1(a,b,c)
  WRITE !,"c in Main after DO: ",c
  QUIT
Subr1(a,b,c)
  NEW c,d
  IF $DATA(c) {WRITE !,"c=",c}
  ELSE {WRITE !,"c in Subr1 is undefined"}
  SET d=a*b
  SET c=d*2
  WRITE !,"c in Subr1: ",c
  QUIT
```

このコードが実行されると、以下の結果が生成されます。

```
c in Main before DO: 6
c in Subr1 is undefined
c in Subr1: 16
c in Main after DO: 6
```

この結果は、前述の例のように値によるパラメータ渡しでも、[参照によるパラメータ渡し](#)でも同じになります。

### ObjectScript

```
Main
  SET a=2,b=4,c=6
  WRITE !,"c in Main before DO: ",c
  DO Subr1(.a,.b,.c)
  WRITE !,"c in Main after DO: ",c
  QUIT
Subr1(&a,&b,&c)
  NEW c,d
  IF $DATA(c) {WRITE !,"c=",c}
  ELSE {WRITE !,"c in Subr1 is undefined"}
  SET d=a*b
  SET c=d*2
  WRITE !,"c in Subr1: ",c
  QUIT
```

変数 `c` は `Subr1` に渡され、NEW を使用して直ちに再定義されます。この場合、変数 `c` を渡す必要はありません。`c` が渡されるかどうかにかかわらず、このプログラムの結果は同じです。サブルーチンの仮パラメータ・リスト内の任意の変数に対して NEW を実行する場合は、これらを未定義で表示し、渡された値をアクセス不可にします。

## 排他的 NEW

排他的 NEW — `NEW (var1,var2)` — は、指定された変数を除く既存のローカル変数環境全体を置換します。既存の変数が指定された場合、それが保持され、新しい環境でも参照可能になります。ただし、そのような変数に行われた変更は、関数またはサブルーチンの終了時に、既存の変数に反映されます。

包含的 NEW では、コンマ区切りリストとして最大 255 の変数を指定できます。この数を超えると、InterSystems IRIS では <SYNTAX> エラーが出力されます。

排他的 NEW (`NEW (x,y,z)`) は、ローカル変数を現在の範囲から一時的に削除します。これは、InterSystems IRIS オブジェクトによって生成されたローカル変数に影響を及ぼす可能性があります。例えば InterSystems IRIS は、InterSystems IRIS のオブジェクト・クエリに対するカーソル・ポインタである `%objcn` を保持します。これを現在の範囲から削除すると、他の内部構造との衝突を引き起こします。したがって、システム構造に影響を与える可能性のあるコンテキストでは、排他的 NEW を使用してはなりません。

31 レベルを超える排他的 NEW コマンドや引数なしの NEW コマンドの発行を試みると、<MAXSCOPE> エラーが返されます。

FOR コード・ブロックで排他的 NEW を使用している場合、除外する変数として FOR の `count` 変数を指定する必要があります。詳細は、“[FOR コマンド](#)”を参照してください。

以下の例では、呼び出し元のルーチン (`Start`) のローカル変数環境が、変数 `a`、`b`、および `c` で構成されていると仮定しています。DO が `Subr1` を呼び出すと、NEW コマンドが `Subr1` のローカル変数環境を再定義して、変数 `c` および `d` 以外のすべての変数を除外します。

NEW の後、サブルーチンの環境は、新しい変数 `c` および `d` だけで構成されています。新しい変数 `c` は、呼び出し元のルーチンの環境から維持されており、その既存の値を保持しています。新しい `d` は、未定義で作成されます。

`Subr1` の最初の SET コマンドは `c` を参照して、`d` に値を代入しています。2 番目の SET コマンドは、新しい値 (24) を `c` に代入しています。サブルーチンが QUIT を実行するとき、`c` は、呼び出し元のルーチンの環境では、この更新された値を保持します (元の値 6 ではありません)。

## ObjectScript

```
Start      SET a=2,b=4,c=6
          DO Subr1
          WRITE !,"c in Start: ",c
          QUIT
Subr1      NEW (c,d)
          SET d=c+c
          SET c=d*2
          WRITE !,"c in Subr1: ",c
          QUIT
```

このコードが実行されると、以下の結果が生成されます。

`c in Subr1:`

`c in Start:`

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に NEW コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

## variable

variable は単一の変数、またはコンマで区切られた変数名のリストです。指定できるのは添え字なしの変数名だけです。配列全体 (つまり添え字なしの配列名) に NEW を実行できます。未定義の変数名を指定したり、既存のローカル変数の名前を再使用することができます。包含的 NEW では、既存のローカル変数を指定する場合、InterSystems IRIS はローカル環境のその変数を再初期化しますが、プログラム・スタックで現在の値を保存し、サブルーチンまたは関数の終了後に復元します。

変数名、または変数名のコンマ区切りのリストが括弧で囲まれているとき (排他的 NEW)、InterSystems IRIS は反対のオペレーションを実行します。つまり、すべてのローカル変数は、指定された変数名を除いて再度初期化され、指定された変数は、以前の値を保持します。InterSystems IRIS はプログラム・スタックのすべての変数で現在の値を保存し、サブルーチンまたは関数の終了後に値を復元します。

NEW コマンド (包括的または排他的) は、以下で使用することはできません。

- ・ [グローバル](#)
- ・ [プロセス・プライベート・グローバル](#)
- ・ [ローカル変数添え字](#)
- ・ [プライベート変数](#)
- ・ [\\$ESTACK、\\$ETRAP、\\$NAMESPACE、および \\$ROLES を除く特殊変数](#)

NEW を上記のコンテキストで使用すると、<SYNTAX> エラーが返されます。

## NEW を使用する場所

NEW を使用すると、現在のプロセスのローカル変数環境をサブルーチン、ユーザ定義関数、または XECUTE された文字列による変更から保護することができます。NEW は、[DO](#) コマンドによって呼び出されたサブルーチン内で、最も頻繁に使用されます。

NEW コマンドの基本的な用途は、呼び出されるサブルーチン、または外部関数内のローカル変数環境の再定義です。パラメータ渡しなしで呼び出されたサブルーチン、または外部関数は、呼び出し元のルーチンのローカル変数環境を継承します。サブルーチン、または関数に対するこの環境を再定義するには、すべてのローカル変数 (引数なしの NEW)、指定されたローカル変数 (包含的 NEW)、または指定された変数を除くすべてのローカル変数 (排他的 NEW) に対して NEW を使用することができます。

プロシージャ内では、変数はプライベートまたはパブリックです。

- ・ 既定では、ローカル変数はそのプロシージャに対してプライベートです。そのプロシージャ・ブロックでは、その外部に存在する同名の変数とは対話しないプライベート変数を使用します。プライベート変数に対して NEW を実行することはできません。プロシージャ・ブロック内のプライベート変数に対して包含的または排他的 NEW を実行しようとすると、<SYNTAX> エラーが発生します。
- ・ プロシージャを宣言するときに、ローカル変数をパブリック変数として明示的にリストできます。プロシージャ・ブロック内のパブリック変数に対して NEW を実行できます。この NEW は、そのプロシージャ内の変数値にのみ影響します。プロシージャ内のパブリック変数に対して NEW を繰り返し実行できます。

詳細は [“プロシージャ変数”](#) を参照してください。

パラメータ渡しの DO コマンドによって呼び出されたサブルーチンの場合は、特に配慮すべき問題があります。詳細は、[“パラメータ渡しのサブルーチン”](#) を参照してください。

## NEW と KILL

NEW によって作成された変数には、対応する明示的な [KILL](#) コマンドは必要ありません。呼び出されたサブルーチンやユーザ定義関数が終了すると、InterSystems IRIS は、サブルーチンや関数内で NEW コマンドで初期化された各変数に対して、暗黙の KILL を実行します。

## 例

以下は、包含的 NEW の例です。これは、既存のローカル変数 a、b、および c を保持し、変数 d および e を追加し、d の以前の値を置換します。

### ObjectScript

```
Main
  SET a=7,b=8,c=9,d=10
  WRITE !,"Before NEW:",!, "a=",a,!, "b=",b,!, "c=",c,!, "d=",d
  DO Sub1
  WRITE !,"Returned to prior context:"
  WRITE !,"a=",a,!, "b=",b,!, "c=",c,!, "d=",d
  QUIT
Sub1
  NEW d,e
  SET d="big number"
  WRITE !,"After NEW:",!, "a=",a,!, "b=",b,!, "c=",c,!, "d=",d
  QUIT
```

以下のターミナルの例は、排他的 NEW を示しています。これは、指定された変数 a および c を除いた既存のすべてのローカル変数を削除します。

### Terminal

```
USER>SET a=7,b=8,c=9,d=10
USER>WRITE
a=7
b=8
c=9
d=10
USER>NEW (a,c)
USER 1S1>WRITE
a=7
c=9
USER 1S1>QUIT
USER>WRITE
a=7
b=8
c=9
d=10
USER>
```

## 特殊変数：\$ESTACK、\$ETRAP、\$NAMESPACE、および \$ROLES

大半の特殊変数では NEW を使用できません。NEW を使用すると<SYNTAX> エラーが返されます。しかし、\$ESTACK、\$ETRAP、\$NAMESPACE、および \$ROLES の 4 つは例外です。

**\$ETRAP** : NEW \$ETRAP コマンドを実行すると、エラー・トラップ用の新しいコンテキストがシステムで作成されます。その後、この新しいコンテキスト内で好みのエラー・トラップ・コマンドを使用して \$ETRAP を設定できます。前のコンテキストの \$ETRAP 値は保持されます。最初に NEW \$ETRAP コマンドを発行せずに \$ETRAP を設定すると、InterSystems IRIS はすべてのコンテキストで \$ETRAP をこの値に設定します。このため、常に、設定する前に \$ETRAP 特殊変数に NEW を実行することをお勧めします。

**\$NAMESPACE** : NEW \$NAMESPACE コマンドを実行すると、新規ネームスペース・コンテキストがシステムで作成されます。このコンテキストでネームスペースを変更できます。例えば、QUIT を使用して現在のコンテキストを終了すると、InterSystems IRIS は直前のネームスペースに戻ります。

## パラメータ渡しのサブルーチン

パラメータ渡しのサブルーチンが呼び出される場合、InterSystems IRIS は、サブルーチンの仮パラメータ・リストに指定されているすべての変数に対して、暗黙の NEW コマンドを実行します。次に、それらの変数に、DO コマンドの実パラメータリストで (値または参照によって) 渡された値を代入します。

DO コマンドが値によるパラメータ渡しを使用している場合や、仮リストが既存のローカル変数を指定している場合、InterSystems IRIS は既存の変数とその値をスタックに格納します。サブルーチンが (明示的または暗黙の QUIT で) 終

了するときに、InterSystems IRIS は、仮リスト変数それぞれに対して暗黙の KILL コマンドを実行して、スタックから元の値を復元します。

## 関連項目

- ・ [DO コマンド](#)
- ・ [KILL コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [SET コマンド](#)
- ・ [\\$NAMESPACE](#) 特殊変数



# OPEN (ObjectScript)

入出力処理のためにデバイスまたはファイルの所有権を取得します。

## 構文

```
OPEN:pc device:(parameters):timeout:"mnespace",...
O:pc device:(parameters):timeout:"mnespace",...
```

## 引数

引数	説明
pc	オプション — 後置条件式
device	開くデバイスです。デバイス ID または <a href="#">デバイス・エイリアス</a> で指定します。デバイス ID は整数 (デバイス番号)、デバイス名、または <a href="#">シーケンシャル・ファイルのパス名</a> のいずれかです。文字列の場合は引用符で囲まなければなりません。device の最大長は 256 文字です。
parameters	オプション — デバイスの特性を設定するのに使用されるパラメータ・リスト。パラメータ・リストは括弧で囲まれ、リスト内のパラメータはコロンで区切られます。パラメータは (パラメータ・リストの固定順で指定された) 位置、またはキーワード (順不同) のいずれかです。位置パラメータとキーワード・パラメータは、混在させることもできます。個別のパラメータとそれぞれの位置とキーワードは、デバイスに依存しています。
timeout	オプション — 整数として指定される、要求が成功するのを待機する秒数。秒の小数部は整数部分に切り捨てられます。省略した場合、InterSystems IRIS は無限に待機します。
mnespace	オプション — 引用符付き文字列として指定された、このデバイスで使用する制御ニーモニックを含むニーモニック空間名です。

## 説明

OPEN コマンドを使用して、入出力処理用に指定されたデバイスの所有者を取得することができます。OPEN は、CLOSE コマンドで所有が解放されるまで、デバイスの所有を保持します。

OPEN コマンドは、コンマを使用して各デバイスの指定を区切ることで、複数のデバイスを開くのに使用できます。デバイスの指定で、その引数はコロン (:) を使用して区切られています。引数が省略されている場合、位置を示すコロンを指定する必要があります。しかし、末尾のコロンは必要ありません。

OPEN コマンドは、ターミナル・デバイス、スプール・デバイス、TCP バインディング、プロセス間パイプ、名前付きパイプ、ジョブ内通信などのデバイスを開くために使用できます。

OPEN コマンドは、シーケンシャル・ファイルを開くためにも使用します。device 引数は、引用符付きの文字列としてファイル・パス名を指定します。parameters 引数は、シーケンシャル・ファイルを管理するパラメータを指定します。このパラメータには、指定されたファイルが存在しない場合は新規のファイルを生成するオプションを含めることもできます。timeout 引数はオプションですが、シーケンシャル・ファイルを開くときには指定することをお勧めします。

シーケンシャル・ファイルを開くオプションの既定は、%SYSTEM.Process クラスの OpenMode() メソッドや FileMode() メソッドを使用して現在のプロセスに対して設定され、Config.Miscellaneous クラスの OpenMode プロパティや FileMode プロパティを使用してシステム全体にわたって設定されます。シーケンシャル・ファイルを開く操作の詳細は、"[シーケンシャル・ファイル入出力](#)" を参照してください。

OPEN コマンドは、InterSystems IRIS データベース・ファイルのアクセスには使用されません。

Windows 上で、ObjectScript は各プロセスに対し、データベース・ファイル数と OPEN で開くファイル数の間で、開くファイル数を割り当てます。OPEN で開くファイルが多すぎて OPEN コマンドへの割り当てができない場合、<TOOMANYFILES> エラーを生じます。InterSystems IRIS ではプロセスごとに開くことができるファイルの最大数は 1,024 です。プロセスごとに開くことができるファイルの実際の最大数は、プラットフォーム固有の設定です。例えば、Windows の既定では、プロセスごとに開くことができるファイルの最大数は 998 です。詳細は、オペレーティング・システムのマニュアルを参照してください。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に OPEN コマンドを実行します。InterSystems IRIS では、後置条件式が False (0 に評価される) の場合、OPEN コマンドを実行しません。OPEN コマンドで複数のデバイスまたはファイルを開く場合でも、許可される後置条件式は 1 つのみです。詳細は、“[コマンド後置条件式](#)” を参照してください。

### device

開くデバイス。以下のいずれかを使用して、デバイスを指定することができます。

- ・ 正の整数として指定される、[物理的デバイス番号](#)。例えば、2 は常にスプーラ・デバイスです。この番号は InterSystems IRIS 内部のもので、プラットフォーム・オペレーティング・システムによって割り当てられたデバイス番号とは無関係です。
- ・ 引用符付きの文字列として指定された、デバイス ID。例えば " |TRM| : |4294318809" です。現在のデバイスに対するデバイス ID は、\$IO 特殊変数に含まれています。
- ・ 正の整数として指定される、[デバイス・エイリアス](#)。デバイス・エイリアスは、物理デバイス番号を参照します。
- ・ 引用符付き文字列として指定された、ファイル・パス名。これは、シーケンシャル・ファイルを開くために使用します。パス名はキャノニック形式 (c:¥myfiles¥testfile)、または現在のディレクトリ (¥myfiles¥testfile) に関連しています。UNIX のパス名には、チルダ (^) 展開を使用して、現在のユーザのホーム・ディレクトリを示すことができます。例：~myfile や ~/myfile。

device の最大長は、Windows および UNIX® では 256 文字です。

### parameters

開くデバイスの動作特性を設定するために使用する、パラメータのリスト。複数のパラメータが含まれる場合、括弧で囲まなければなりません (プログラム記述時には、パラメータが 1 つでも必ず括弧で囲むことをお勧めします)。コロンは、左括弧の前に配置します。括弧内では、コロンを使用して複数のパラメータを区切ります。

デバイス用のパラメータは、位置パラメータ、またはキーワード・パラメータのいずれかを使用して指定されます。また、位置パラメータとキーワード・パラメータを同じパラメータ・リスト内で混在させて使用することもできます。

多くの場合、矛盾したパラメータ、複製パラメータ、または無効なパラメータ値を指定するとエラーが返されます。可能な場合は常に、InterSystems IRIS は不適切なパラメータ値を無視し、適切な既定値を使用します。

パラメータのリストを指定しない場合は、デバイスの既定パラメータが使用されます。デバイスの既定パラメータは構成可能です。管理ポータルに進み、[\[システム管理\]](#)、[\[構成\]](#)、[\[デバイス設定\]](#)、[\[デバイス\]](#) の順に選択して、現在のデバイスのリストを表示します。対象のデバイスに対して [\[編集\]](#) をクリックし、[\[オープン・パラメータ :\]](#) オプションを表示します。この値を、OPEN コマンド・パラメータと同じ方法で指定します (括弧も含む)。例えば ("AVL" : 0 : 2048) です。

利用できるパラメータは、開くデバイスの種類によって異なります。デバイス・パラメータの詳細は、“[入出力の概要](#)” を参照してください。

### 位置パラメータ

位置パラメータは、パラメータ・リスト内で固定の順序で指定されなければなりません。位置パラメータは省略することもできますが (省略の場合は既定値を使用)、省略された位置パラメータの位置を示すためのコロンは保持する必要があります。

す。末尾のコロンは必須です。余分なコロンは無視されます。個別のパラメータとそれぞれの位置は、デバイスに依存しています。位置パラメータには、値と文字コード文字列の 2 種類があります。

値は整数 (例えばレコード・サイズ)、文字列 (例えばホスト名)、または値に評価される変数や式のいずれかです。

文字コード文字列は、開くデバイスの特性を指定するために個別の文字を使用します。大半のデバイスで、この文字コード文字列は位置パラメータの 1 つとなります。ユーザは文字列内の任意の文字数を順不同で指定することができます。文字コードは、大文字と小文字を区別しません。文字コード文字列は引用符で囲まれます。文字コード文字列でスペースや句読点を使用することはできません (¥ 記号で区切られた名前が続く K と Y (例: K¥name¥) は例外です)。位置パラメータを使用する例は、シーケンシャル・ファイルを開くときに "ANDFW" (既存のファイルへの追加 (A)、ファイルの新規作成 (N)、ファイルの削除 (D)、固定長レコード (F)、書き込みアクセス (W)) という文字コード文字列を指定する場合などです。文字コード文字列パラメータの位置と、それぞれの文字の意味は、デバイスに依存しています。

## キーワード・パラメータ

キーワード・パラメータは、パラメータ・リスト内の任意の順序で指定することができます。パラメータ・リストはキーワード・パラメータだけで構成することも、位置パラメータとキーワード・パラメータを混在させて構成することもできます。(一般的には、位置パラメータが (適切な順序で) 最初に指定され、その次にキーワード・パラメータが指定されます)。すべてのパラメータ (位置、およびキーワード) は、コロン (:) で区切られなければなりません。キーワード・パラメータのパラメータ・リストの標準的な構文は、以下のとおりです。

```
OPEN device: (/KEYWORD1=value1:/KEYWORD2=value2:.../KEYWORDn=valuen):timeout
```

個別のパラメータとそれぞれの位置は、デバイスに依存しています。標準規約として、位置パラメータやキーワード・パラメータのいずれかを使用して、同じパラメータと値を指定することができます。文字コード文字列は、/PARAMS キーワードを使用して キーワード・パラメータとして指定することができます。

## timeout

OPEN 要求の成功を待機する時間を指定します (秒)。先頭のコロンは必須です。timeout は、必ず整数値または式で指定します。timeout がゼロ (0) の場合、OPEN はファイルを開く操作を 1 回試行します。実行が失敗した場合、OPEN も即座に失敗します。試行に成功した場合は、ファイルが無事に開きます。timeout を設定していない場合、InterSystems IRIS は OPEN が成功するまで、またはプロセスが手動で終了されるまで、デバイスを開く試行を続行します。timeout オプションを使用し、デバイスを開く操作に成功した場合、InterSystems IRIS は特殊変数 \$TEST を 1 (TRUE) に設定します。タイムアウト時間内にデバイスを開くことができなかった場合、InterSystems IRIS は \$TEST を 0 (FALSE) に設定します。\$TEST は、ユーザが設定することもできます。また、JOB、LOCK、または READ のタイムアウトで設定されることもあります。

## mnespace

このデバイスによって使用されるデバイス制御ニーモニックを含む、ニーモニック・スペースの名前です。既定で、InterSystems IRIS はデバイスとシーケンシャル・ファイル用に ^%X364 (ANSI X3.64 互換) のニーモニック・スペースを提供しています。既定のニーモニック・スペースは、デバイスのタイプごとに割り当てられます。

管理ポータルに進み、[システム管理]、[構成]、[デバイス設定]、[IO設定] の順に選択します。[ファイル]、[ターミナル]、または [その他] のニーモニック・スペース設定を表示して編集します。

ニーモニック・スペースは、READ コマンドと WRITE コマンドによって使用されるデバイス制御ニーモニックに対するエントリ・ポイントを含むルーチンです。READ コマンドと WRITE コマンドは、/mnemonic(params) 構文を使用して、デバイス制御ニーモニックを呼び出します。このデバイス制御ニーモニックは、カーソルを画面内の指定された位置に移動するなどの処理を実行します。

mnespace 引数を使用して、既定のニーモニック・スペースの割り当てをオーバーライドします。このデバイスで使用されている制御ニーモニック・エントリ・ポイントを含む、ObjectScript ルーチンを指定します。二重引用符は必須です。このオプションは、READ コマンドや WRITE コマンドでデバイス制御ニーモニックを使用したい場合のみ指定します。ニーモニック・スペースが存在しない場合は、<NOROUTINE> エラーが返されます。ニーモニック空間の詳細は、"[入出力の概要](#)" を参照してください。

## 例

以下の例は、OPEN コマンドでデバイス 2 (スプーラ) の所有権を取得するものです。最初の位置パラメータ (3) で、グローバル `SPOOL` 内のファイル番号を指定し、2 番目の位置パラメータ (12) で、そのファイル内の行番号を指定しています。後から USE コマンドを使用してこれを現在のデバイスに指定する (つまり USE 2) 場合、ObjectScript はそれ以降の出力をスプーラに送信します。

### ObjectScript

```
OPEN 2:(3:12)
```

以下の例は、OPEN コマンドで、シーケンシャル・ファイル CUSTOMER の所有権をタイムアウト時間 10 秒以内に取得します。

### ObjectScript

```
OPEN "\myfiles\customer":::10
```

パラメータが指定されていないので括弧は省略されますが、それでもコロンは必要です。

以下の例は、Seqtest というシーケンシャル・ファイルを開きます。文字コード位置パラメータは "NRW" です。"N" 文字コードは、ファイルが存在しない場合に、この名前を持つ新規のシーケンシャル・ファイルを作成するように指定します。"R" と "W" 文字コードは、そのファイルが読み取りまたは書き込みのために現在開いていることを意味します。タイムアウトは 5 秒です。

### ObjectScript

```
NEW $NAMESPACE
SET $NAMESPACE="%SYS"
SET dir=##class(%SYSTEM.Process).CurrentDirectory() ; determine InterSystems IRIS directory
SET seqfilename=dir_"Samples\Seqtest"
OPEN seqfilename:("NRW"):5
WRITE !,"Opened a sequential file named Seqtest"
USE seqfilename
WRITE "a line of data for the sequential file"
CLOSE seqfilename:"D"
WRITE !,"Closed and deleted Seqtest"
QUIT
```

この例では、UnknownUser が %DB\_IRISYS ロールを割り当てている必要があります。

## デバイスの所有権と現在のデバイス

OPEN は、指定されたデバイスの所有権を設定します。プロセスは、プロセスが終了するか、後続の CLOSE コマンドでデバイスが解放されるまで、そのデバイスの所有権を維持します。あるプロセスがデバイスを所有している間は、他のプロセスがこのデバイスを取得したり、使用したりすることはできません。

プロセスは、同時に複数のデバイスを所有できます。しかし、現在のデバイスにできるデバイスは 1 つだけです。所有しているデバイスを現在のデバイスとして設定するには、USE コマンドを使用します。現在のデバイスのデバイス ID は、\$IO 特殊変数に含まれています。

プロセスは、常に少なくとも 1 つのデバイスを所有しています (device 0 として指定されます)。これが、そのプロセスの主デバイスです。このデバイスは、プロセスの起動時に割り当てられます。通常は、InterSystems IRIS へのサインオンに使用されたターミナルです。主デバイスの ID は、\$PRINCIPAL 特殊変数に含まれています。

プロセスが終了すると、InterSystems IRIS は、そのプロセスによって所有されている各デバイスに対して、暗黙の CLOSE を実行し、デバイスを使用可能デバイスのプールに返します。

## 所有デバイスのパラメータの変更

既にプロセスが所有しているデバイスのパラメータを変更するには、以下の方法があります。

- ・ デバイスをいったん閉じて、新しいパラメータ値で再び開く。

- ・ デバイスがターミナルまたは TCP デバイスの場合、既にかかれているデバイスに対する新しいパラメータ値で OPEN を発行できます。

他の OPEN コマンドでそのデバイスを指定する場合、最初の OPEN コマンドで設定されたデバイス・パラメータは、明示的に変更されない限り、有効のままとなります。デバイスの種類によっては、デバイスをいったん閉じて再び開いた場合に、後続の入出力が異なる可能性があります。

一部のデバイスでは、parameters オプションを省略し、後で USE コマンドの parameters オプションを使用して、必要な特性を設定することができます。

## 物理デバイス番号の使用法

InterSystems IRIS では、システムによって割り当てられている物理デバイス番号によって、任意のデバイスを識別できます。InterSystems IRIS では、どの実装タイプでも、以下の物理デバイス番号を認識します。

- ・ 0 = プロセスの主デバイス (通常はサインオンのときに使用したデバイス)
- ・ 2 = スプーラ (後で印刷するために出力を保持する)
- ・ 63 = [表示バッファ](#)。

OPEN 63 は、以下の例に示すように、ネームスペースを受け入れます。

### ObjectScript

```
OPEN 63: "SAMPLES"
```

存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行します。

デバイス 3 は無効なデバイスになります。この番号のデバイスをオープンしようとする、timeout の期限切れを待たずに <NOTOPEN> エラーが返されます。

デバイス番号に関する詳細は、“[入出力デバイスの詳細](#)” を参照してください。

## デバイス・エイリアスの使用法

エイリアスは数値デバイス ID の別名です。これは、有効なデバイス番号でなければならない、一意である必要があり、割り当てられたデバイス番号と競合してはなりません。

デバイスに対する数値エイリアスを作成できます。管理ポータルに進み、[システム管理]、[構成]、[デバイス設定]、[デバイス] の順に選択して、現在のデバイスとそのエイリアスのリストを表示します。対象のデバイスに対して [編集] をクリックし、その [エイリアス :] オプションを編集します。

デバイスのエイリアスを割り当てた後、OPEN コマンド、または [%IS ユーティリティ](#) を使用して、このエイリアスを使用するデバイスを開きます。

## 開くファイル数割り当ての超過

InterSystems IRIS は、データベース・ファイルと OPEN で開くファイルとの間で、プロセスごとに開くことができるファイル数を割り当てます。OPEN によって OPEN コマンドに割り当てられるファイル数が多すぎる場合、<TOOMANYFILES> エラーが返されます。InterSystems IRIS ではプロセスごとに開くことができるファイルの最大数は 1,024 です。プロセスごとに開くことができるファイルの実際の最大数は、プラットフォーム固有の設定です。例えば、Windows の既定では、プロセスごとに開くことができるファイルの最大数は 998 です。

## 既定のレコード長

OPEN コマンドにシーケンシャル・ファイルのレコード・サイズが指定されていない場合、InterSystems IRIS は、既定の 32,767 文字を想定します。



## 関連項目

- ・ [CLOSE コマンド](#)
- ・ [USE コマンド](#)
- ・ [\\$TEST 特殊変数](#)
- ・ [\\$IO 特殊変数](#)
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)
- ・ [TCP クライアント/サーバ通信](#)
- ・ [シーケンシャル・ファイルの入出力](#)
- ・ [スプール・デバイス](#)
- ・ [^%IS グローバルと %IS ユーティリティ](#)

# QUIT (ObjectScript)

ループ構造またはルーチンの実行を終了します。

## 構文

```
QUIT:pc expression
Q:pc expression
```

```
QUIT n
Q n
```

## 引数

引数	説明
pc	オプション – 後置条件式
expression	オプション – 呼び出すルーチンに返す値。有効な式。
n	オプション – ターミナル・プロンプトのみ: クリアするプログラム・レベルの数。正の整数に解決される式。

## 説明

QUIT コマンドは、以下の 2 つのコンテキストで使用されます。

- ・ プログラム・コード内の QUIT
- ・ ターミナル・プロンプトでの QUIT

## プログラム・コード内

QUIT コマンドは、現在のコンテキストの実行を終了し、囲んでいるコンテキストに移動します。QUIT は、ルーチン内で呼び出されると呼び出し元のルーチンに戻り、呼び出し元のルーチンがない場合は、プログラムを終了します。FOR、DO WHILE、または WHILE の各フロー制御構造内、あるいは TRY または CATCH ブロック内から発行されると、QUIT は構造を終了し、その構造の外側にある次のコマンドで実行を続けます。

類似する [RETURN](#) コマンドを使用すると、FOR、DO WHILE、または WHILE ループあるいは入れ子のループ構造、もしくは TRY または CATCH ブロック内を含めて、任意の場所でルーチンの実行が終了します。RETURN は常に現在のルーチンを終了し、呼び出し元のルーチンに戻るか、呼び出し元のルーチンがない場合はプログラムを終了します。

QUIT コマンドには、以下の 2 つの基本形式があります。

- ・ 引数なし
- ・ 引数付き

後置条件は、引数として考慮されません。\$QUIT 特殊変数は、引数付きの QUIT コマンドが現在のコンテキストを終了する必要があるかどうかを示します。そのために、M16 “許可されていない引数で QUIT しました” と M17 “引数付きの QUIT が必要です” の 2 つの [\\$ECODE](#) エラー・コードが用意されています。

## 引数なしの QUIT

引数なしの QUIT は、値を返さずに現在のコンテキストを終了します。これは、DO コマンドまたは XECUTE コマンドで開始された処理の実行レベルを終了するため、または FOR、DO WHILE、または WHILE の各フロー制御ループを終了するために使用されます。

DO、XECUTE、または(入れ子になっていない)フロー制御ループの各コマンドがターミナル・プロンプトから発行された場合、QUIT はターミナルに制御を戻します。終了した操作で、QUIT コマンドの前に NEW コマンドが配置されている場合、QUIT は、対象となっていた変数に自動的に KILL を実行し、元の値をリストアします。

## 引数付きの QUIT

QUIT expression は、ユーザ定義関数またはオブジェクト・メソッドを終了し、指定された式の結果を返します。FOR、DO WHILE、WHILE のいずれかのコマンド・ループの内部からルーチンを終了するために引数付きの QUIT を使用することはできません。引数付きの QUIT を使用して、TRY ブロックまたは CATCH ブロック内部から終了することはできません。

引数付きの QUIT がサブルーチン内で呼び出されると、以下のいずれかが発生します。

- ・ (関数の代わりに) サブルーチンの内部から引数付きの QUIT が呼び出された場合、QUIT 引数は評価され(副次的作用またはエラーが発生する場合あり)、引数の結果が破棄されます。実行はサブルーチンの呼び出し元に戻ります。
- ・ サブルーチンが DO によって呼び出され、その DO 引数の範囲にある場合は、QUIT はその引数(およびその評価によって引き起こされるすべての副次的作用)を評価しますが、引数は返しません。例えば、QUIT 4/0 で終わる DO に呼び出されたサブルーチンは<DIVIDE> エラーを生成します。サブルーチンが引数付きの QUIT で終了する DO で呼び出されていて、**\$ETRAP** で終了する場合も、同じ動作が発生します。

## ターミナル・プロンプト

ターミナル・プロンプトで QUIT を発行すると、プログラム・スタックからエンティティがクリアされます。

- ・ QUIT n は、指定された番号のレベルをプログラム・スタックからクリアします。**\$STACK** 特殊変数で、プログラム・スタックでの現在のレベル数を判別できます。  
  
n は、リテラル、変数、または算術式として指定できます。n が現在のレベル数よりも大きい場合、プログラム・スタックからすべてのレベルがクリアされ、エラーは発行されません。n の小数は、整数値に切り捨てられます。n がゼロ以外の負の整数に解決される場合、QUIT はプログラム・スタックからすべてのレベルをクリアします。n が 0 (ゼロ) に解決される場合、システムは<COMMAND> エラーを生成します。
- ・ QUIT は、プログラム・スタックからすべてのレベルを消去します。

**\$STACK** 特殊変数は、コール・スタックのコンテキスト・フレームの現在の番号を含みます。この番号は、ターミナル・プロンプトの一部としても表示されます。

以下の例では、整数の引数付きの QUIT を使用して、プログラム・スタックから指定された番号のレベルをクリアし、次に引数なしの QUIT を使用して残りのレベルすべてを削除します。

### Terminal

```
USER 5f0>QUIT 1
USER 4d0>QUIT 2
USER 2d0>QUIT
USER>
```

詳細は、“[ターミナル・プロンプトでのエラー処理](#)”を参照してください。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳



細は、“[コマンド後置条件式](#)”を参照してください。QUIT コマンドが引数を取らない場合、同じ行における後置条件と次のコマンドの間に 2 つ、またはそれ以上のスペースが必要です。

## expression

任意の有効な ObjectScript 式。これはユーザ定義の関数内でのみ使用され、呼び出し元のルーチンに評価結果を返します。

## 例

以下の 2 つの例では、フロー制御構造内から発行されたときの QUIT と RETURN の動作を比較します。QUIT は、FOR ループを終了し、MySubroutine の残りを続けて実行してから MyRoutine に戻ります。RETURN は MySubroutine を終了して MyRoutine に戻ります。

### ObjectScript

```
MyMain
  WRITE "In the main routine",!
  DO MySubroutine
  WRITE "Returned to main routine",!
  QUIT
MySubroutine
  WRITE "In MySubroutine",!
  FOR i=1:1:5 {
    WRITE "FOR loop:",i,!
    IF i=3 QUIT
    WRITE "  loop again",!
  }
  WRITE "MySubroutine line displayed with QUIT",!
  QUIT
```

### ObjectScript

```
MyMain
  WRITE "In the main routine",!
  DO MySubroutine
  WRITE "Returned to main routine",!
  QUIT
MySubroutine
  WRITE "In MySubroutine",!
  FOR i=1:1:5 {
    WRITE "FOR loop:",i,!
    IF i=3 RETURN
    WRITE "  loop again",!
  }
  WRITE "MySubroutine line not displayed with RETURN",!
  QUIT
```

以下の例では、最初の QUIT コマンドの実行は後置条件 (:x>46) によって制御されます。乱数が 46 より大きい場合は、InterSystems IRIS は Cube プロシージャを実行しません。最初の QUIT が後置条件を受け、文字列を num として呼び出し元のルーチンに返します。乱数が 46 以下の場合は、2 つ目の QUIT が式 x\*x\*x の結果を num として返します。

### ObjectScript

```
Main
  SET x = $RANDOM(99)
  WRITE "Number is: ",x,!
  SET num=$$Cube(x)
  WRITE "Cube is: ",num
  QUIT
Cube(x) QUIT:x>46 "a six-digit number."
  WRITE "Calculating the cube",!
  QUIT x*x*x
```

以下の 2 つの例では、TRY および CATCH を使用した QUIT と RETURN の動作を比較します。TRY ブロックは、0 による除算を試行するので、CATCH ブロックが呼び出されます。この CATCH ブロックには、QUIT または RETURN のいずれかで終了する入れ子の TRY ブロックが含まれます。(デモンストレーションのために、これらのプログラムには“フォーカスルー”を防ぐために推奨されているコード (QUIT または RETURN) は含まれていません)。

QUIT は、入れ子の TRY ブロックを終了して囲んでいるブロックに移動し、CATCH ブロックの残りで実行を続けます。CATCH ブロックを完了すると、TRY/CATCH 構造の外側にあるフォールスルー行を実行します。

### ObjectScript

```
TRY {
  WRITE "In the TRY block",!
  SET x = 5/0
  WRITE "This line should never display"
}
CATCH exp1 {
  WRITE "In the CATCH block",!
  WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  TRY {
    WRITE "In the nested TRY block",!
    QUIT
  }
  CATCH exp2 {
    WRITE "In the nested CATCH block",!
    WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  }
  WRITE "QUIT displays this outer CATCH block line",!
}
WRITE "fall-through at the end of the program"
```

RETURN はルーチンを終了します。そのため、入れ子の TRY ブロックと囲んでいるブロックをすべて終了し、TRY/CATCH 構造の外側にあるフォールスルー行を実行しません。

### ObjectScript

```
TRY {
  WRITE "In the TRY block",!
  SET x = 5/0
  WRITE "This line should never display"
}
CATCH exp1 {
  WRITE "In the CATCH block",!
  WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  TRY {
    WRITE "In the nested TRY block",!
    RETURN
  }
  CATCH exp2 {
    WRITE "In the nested CATCH block",!
    WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
  }
  WRITE "RETURN does not display this outer CATCH block line",!
}
WRITE "fall-through at the end of the program"
```

以下の例での QUIT コマンドの引数は、オブジェクト・メソッドです。InterSystems IRIS はメソッドの実行を終了して、呼び出し元のルーチンに制御を戻します。

### ObjectScript

```
QUIT inv.TotalNum()
```

## QUIT による変数のリストア

終了したプロセスで、QUIT コマンドの前に NEW コマンドが配置されている場合、QUIT は、対象となっていた変数に自動的に KILL を実行し、元の値をリストアします。

## QUIT とフロー制御構造

QUIT は、FOR ループ、DO WHILE ループ、および WHILE ループを終了するために使用されます。ループ構造の最後に続くコマンドで実行を継続します。これらのループ構造が入れ子になっている場合、QUIT を実行すると、その呼び出し元の内側のループが終了して、そのループの外側のループに実行が移ります。

QUIT コマンドが IF コード・ブロック (あるいは ELSEIF コード・ブロックまたは ELSE コード・ブロック) 内にある場合、QUIT は、このコード・ブロックが存在しないかのように、通常の QUIT コマンドとして動作します。

- ・ IF コード・ブロックが、ループ構造 (FOR コード・ブロックなど) 内で入れ子にされている場合、QUIT はそのループ構造ブロックを終了し、このループ構造コード・ブロックに続くコマンドで実行を続けます。
- ・ IF コード・ブロックがループ構造内で入れ子にされていない場合は、QUIT は現在のルーチンを終了します。

## 無限の FOR ループでの使用

無限の FOR ループは、引数なしの FOR です。終了されない限り、無限にループします。無限の FOR ループを制御するには、後置条件付きの QUIT と QUIT を呼び出す IF コマンドのどちらかをループ構造に含める必要があります。IF 内の QUIT は、囲まれている FOR ループを終了します。これらの QUIT が 1 つもない場合、ループは無限ループとして、無限に実行され続けます。

以下の例では、無限の FOR ループは、後置条件付きの QUIT を使用して終了します。ユーザが "Text =" プロンプトへの応答として値を入力する限り、このループの実行は継続します。ユーザが NULL 文字列を入力した (つまり Enter を押しただけ) の場合にのみ、QUIT が実行され、ループが終了します。

### ObjectScript

```
Main
  FOR
    {
      READ !, "Text =", var1
      QUIT:var1="" DO Subr1
    }
Subr1
  WRITE "Thanks for the input", !
  QUIT
```

このコマンドには、QUIT コマンドの後置条件と、それに続く同じ行の DO コマンドの間に、少なくとも 2 つのスペースが必要です。InterSystems IRIS は後置条件を引数ではなく、コマンド修飾子として処理するからです。

## 暗黙の QUIT

InterSystems IRIS は各ルーチンの最後に暗黙の QUIT を実行しますが、プログラムを読みやすくするために、明示的に QUIT を含めることもできます。

以下の場合、QUIT コマンドは必要ありません。InterSystems IRIS は自動的に暗黙の QUIT を発行し、異なるコード・ユニットの実行が "フォールスルー" しないように防ぎます。

- ・ InterSystems IRIS はルーチンの最後に暗黙の QUIT を実行します。
- ・ InterSystems IRIS はパラメータを持つラベルに遭遇した場合、暗黙の QUIT を実行します。パラメータを持つラベルは、括弧内にパラメータが何も含まれていない場合でも、括弧付きのラベルとして定義されます。すべてのプロシージャは、(パラメータが定義されていない場合でも) パラメータを持つラベルから始まります。多くのサブルーチンと関数にも、パラメータを持つラベルがあります。

どのような環境でも、明示的な QUIT はコード化できます。

## DO での振る舞い

QUIT は、DO コマンドによって呼び出されたサブルーチンの内部で実行された場合、サブルーチンを終了し、DO コマンドの次のコマンドに制御を戻します。

## XECUTE での振る舞い

QUIT は、XECUTE コマンドが実行されているコード行で実行された場合、その行の実行を終了し、XECUTE コマンドの次のコマンドに制御を戻します。引数は指定できません。

## ユーザ定義関数での振る舞い

QUIT は、ユーザ定義の関数で使用された場合、その関数を終了し、指定された式の結果の値を返します。expression 引数は必須です。

その用途は、ユーザ定義関数はパラメータ渡しの DO コマンドとほぼ同じです。DO コマンドとの違いは、式の値を、変数を介してではなく、直接返すことです。外部関数を呼び出すには、以下の形式を使用します。

`$$name(parameters)`

`name` には関数の名前を指定します。label、routine、または label routine の形式で指定できます。

`parameters` には、関数に渡される、コンマ区切りのパラメータ・リストを指定します。関数に関連付けられた label もパラメータ・リストを持っている必要があります。呼び出される関数のパラメータ・リストは、実パラメータ・リストと呼ばれます。関数ラベルのパラメータ・リストは、仮パラメータ・リストと呼ばれます。

以下の例では、FOR ループは最初に READ コマンドを使用して、2 乗する数字を取得、それを変数 `num` に保存します。(引数なしの FOR と後置条件付きの QUIT の後にスペースが 2 つ含まれていることに注意してください)。次に、WRITE コマンドを使用して、関数のパラメータとして `num` を指定した標準関数 `Square` を呼び出します。

この関数の唯一のコードは、2 乗を計算する式が続く QUIT コマンドです。QUIT コマンドに遭遇すると、InterSystems IRIS は式を評価し、関数を終了し、結果の値を直接 WRITE コマンドに返します。`num` の値は変更されません。

## ObjectScript

```
Test WRITE "Calculate the square of a number",!  
  FOR {  
    READ !, "Number:", num QUIT: num = "  
    WRITE !, $$Square(num), !  
    QUIT  
  }  
Square(val)  
  QUIT val*val
```

## 関連項目

- ・ [DO コマンド](#)
- ・ [DO WHILE コマンド](#)
- ・ [FOR コマンド](#)
- ・ [NEW コマンド](#)
- ・ [RETURN コマンド](#)
- ・ [WHILE コマンド](#)
- ・ [XECUTE コマンド](#)
- ・ [\\$ECODE 特殊変数](#)
- ・ [\\$ESTACK 特殊変数](#)
- ・ [\\$QUIT 特殊変数](#)
- ・ [\\$STACK 特殊変数](#)

## READ (ObjectScript)

入力を受け付け、それを変数に格納します。

### 構文

```
READ:pc readargument,...
R:pc readargument,...
```

readargument には、以下を指定できます。

```
fchar
prompt
variable:timeout
*variable:timeout
variable#length:timeout
```

### 引数

引数	説明
pc	オプション - 後置条件式
fchar	オプション - 1 つ、または複数の形式制御文字。許可されている文字は、!、#、?、および / です。
prompt	オプション - ユーザ入力を求めるプロンプト、またはメッセージとして表示される文字列リテラル。引用符で囲まれます。
variable	入力データを受け取る変数です。ローカル変数、プロセス・プライベート・グローバル、またはグローバルのいずれかです。添字なしでも添字付きでもかまいません。
length	オプション - 受け入れ可能な文字数。整数、または整数に評価される式または変数のいずれかとして指定されます。先頭の # 記号は必須です。読み取る文字数が指定されていない場合、InterSystems IRIS は、既定の 32,767 文字を想定します。
timeout	オプション - 整数として指定される、要求が成功するのを待機する秒数。秒の小数部は整数部分に切り捨てられます。先頭のコロン (:) は必須です。省略した場合、InterSystems IRIS は無限に待機します。

コンマで引数を区切ることで、複数の fchar 引数または prompt 引数を指定できます。

### 概要

READ コマンドは、現在のデバイスからの入力を受け入れます。現在のデバイスは、OPEN コマンドと USE コマンドを使用して設定されています。\$IO 特殊変数は、現在のデバイスのデバイス ID を含みます。既定では、現在のデバイスはユーザ端末です。

READ コマンドは、プログラムが現在のデバイスからの入力を受け取るまで、またはタイムアウトまで、プログラムの実行を一時停止します。このため現在のデバイスがユーザ端末の場合は、READ コマンドはバックグラウンド (非対話的) ジョブとして実行されているプログラムで使用することはできません。

variable 引数は、入力文字を受け取ります。READ は、variable が未定義の場合は最初に variable を定義し、variable が前の値を持つ場合は消去します。したがって、variable に入力データがない場合 (例えば、文字が入力される前に READ がタイムアウトになった場合)、variable は定義され、NULL 文字列を含みます。これは、唯一入力された文字が **ターミネータ文字** (例えば、ユーザ端末から [Enter] を押すなど) の場合でも同様です。**中断** (例えば [Ctrl-C]) の影響は、以下を参照してください。

固定長、および可変長の読み取りでは、variable は読み取り処理を終了するターミネータ文字を格納しません。単一文字の読み取りは variable を別に処理します。variable を使用する単一文字の読み取りは、以下を参照してください。

オプションのタイムアウト値を指定する場合、READ はすべての文字が入力される前にタイムアウトできます。READ がタイムアウトすると、タイムアウトの発生前に入力された文字は、variable に格納されます。この場合、ターミネータ文字の入力は必要ありません。タイムアウトの発生前に入力された文字が variable に転送され、READ が終了し、\$TEST が 0 に設定されます。

READ 処理には、可変長読み取り、固定長読み取り、単一文字読み取りの 3 種類あります。これら 3 つは、タイムアウト引数付き、またはなしで指定することができます。単一の READ コマンドは、3 種類の任意の組み合わせで複数の READ 処理を含むことができます。それぞれの読み取り処理は、左から右の順に個別に実行されます。READ コマンドは、任意の数のコンマで区切られた prompt 引数および fchar 引数を含むこともできます。

3 種類の READ 処理の詳細は、以下のとおりです。

- ・ 可変長読み取りの形式は、以下のとおりです。

READ variable

これは、任意の数の入力文字を受け取り、それを指定された variable に格納します。入力はターミネータ文字で終了します。通常は、[Enter] キーを押すことで終了できます。(ターミネータ文字ではなく) 入力文字は、variable に格納されます。

- ・ 固定長読み取りの形式は、以下のとおりです。

READ variable#length

これは、最大 length 個の入力文字を受け取り、それを指定された variable に格納します。入力は、指定された数の文字が入力されたとき、またはターミネータ文字に遭遇したときに自動的に終了します。例えば、4 固定長読み取りで 2 つの文字を入力し、[Enter] キーを押します。(ターミネータ文字ではなく) 入力文字は、variable に格納されます。

- ・ 単一文字読み取りの形式は、以下のとおりです。

READ \*variable

単一文字の読み取りは単一の入力文字を受け取り、指定された variable に同等の ASCII 数値を格納します。これは文字自体を \$ZB と \$KEY 特殊変数に格納します。入力は、単一文字が入力されたときに自動的に終了します。ターミネータ文字は単一文字入力と見なされ、格納されます。オプションの timeout 引数が指定されている場合にタイムアウトが発生すると、タイムアウトは variable を -1 に設定します。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

### fchar

以下の形式制御コードのいずれかを指定します。キーボードからのユーザ入力で使用されると、これらの制御は指定された prompt、またはユーザ入力エリアを表示する画面内の位置を決定します。

! は、新しい行を開始します。複数の感嘆符を指定できます。

# は、新規ページを開始します。端末で、現在の画面が消去され、新しい画面の一番上から開始されます。

?n は、n 番目の列に配置します。n には正の整数を指定します。

/keyword(parameters) は、デバイス制御モニターです。ビデオ端末のカーソルを位置決めするなどのデバイス特有の機能を実行します。スラッシュ文字 (/) の後にはキーワードが続きます。このキーワードには、オプションで 1 つ、また



は複数の括弧で囲まれたパラメータが続きます。複数のパラメータは、コンマで区切られます。キーワードは、現在のデバイスのニーモニック・スペース・ルーチン内のエントリ・ポイント・ラベルです。

デバイス・タイプの既定のニーモニック・スペースは、以下のいずれかの方法で指定できます。

- ・ 管理ポータルに進み、[システム管理]、[構成]、[デバイス設定]、[IO設定] の順に選択します。[ファイル]、[その他]、または [ターミナル] のニーモニック・スペース設定を表示して編集します。
- ・ デバイスに対して OPEN コマンド、または USE コマンドを実行するときに /mnospace パラメータを含めます。

形式制御は、複数指定することができます。例えば、#!!!?20 は、新規ページ (または画面) のトップから開始し、3 行下り、列 20 に移動することを意味しています。形式制御文字は、コンマで区切られた別の READ 引数と混在させることができます。以下はその例です。

## ObjectScript

```
READ #!!, "Please enter", !, "your name: ", x, !, "THANK YOU"
```

以下のように表示されます。

```
>
Please enter
your name: FRED
THANK YOU
>
```

## prompt

端末キーボードからのユーザ入力を求めるプロンプト、またはメッセージとして表示される文字列リテラルです。通常、prompt 引数の後に variable が続く場合は、表示されたリテラルの後にユーザ入力エリアが続きます。コンマで区切られた一連の prompt 引数および fchar 引数を使用すると、複数行のプロンプトやメッセージを指定できます。

## variable

入力データを受け取るローカル変数、プロセス・プライベート・グローバル、またはグローバルです。添字なしでも添字付きでもかまいません。指定された変数が既に存在しない場合、InterSystems IRIS はその変数を READ 処理の開始時に定義します。指定された変数が定義され、値を持つ場合、InterSystems IRIS はこの値を READ 処理の開始時に消去します。

文字を入力すると、その文字は variable に格納されます。オプションの timeout 引数が指定されていて、読み取り処理がタイムアウトによって中断された場合、その時点までに入力された文字は variable に格納されます (ただし **Ctrl-C 中断**に遭遇したときの variable の振る舞いには注意してください。詳細は以下のとおりです)。

印字**不能文字** (Tab など) は、variable に格納されます。**ターミネータ文字**は、すべての種類の読み取り処理を終了させるために使用されます。例えば端末から、[Enter] キーを押して読み取り処理を終了する場合などです。このターミネータ文字は、可変長や固定長の読み取りに対して variable に格納されません。単一文字読み取りに対しては、ターミネータ文字は variable に格納されます。

## length

固定長の読み取りに対して、受け入れ可能な最大文字数を指定する正の整数です。READ は、指定された文字数が入力されたとき、または**ターミネータ文字**に遭遇したときのいずれかに終了します。引数はオプションですが、指定される場合、先頭の # 記号は必須です。

読み取る文字数が指定されていない場合、InterSystems IRIS は、既定の 32,767 文字を想定します。

ゼロや負数を指定すると <SYNTAX> エラーが返されます。先行のゼロと数字の小数部は無視され、整数部のみ使用されます。length は整数に解決される変数あるいは式として指定できます。

READ a#1 と READ \*a は両方とも、単一文字を入力するために使用されることに注意してください。しかし、variable に格納されている値は異なります。a#1 は入力文字を変数 a に格納します。\*a は入力文字の ASCII 数値を変数 a に格

納します。両方とも入力文字を \$ZB 特殊変数に格納します。これら 2 種類の単一文字入力は、ターミネータ文字の処理方法や、タイムアウトの処理方法でも異なります。

## timeout

要求の成功を待機する時間を指定します。この引数はオプションですが、指定される場合、先頭のコロンは必須です。timeout は整数、または整数に評価される式として指定されなければなりません。timeout 引数は、以下のように \$TEST 特殊変数を設定します。

- timeout 引数を持つ READ が正常に終了する (タイムアウトしない) : \$TEST は 1 (TRUE) に設定されます。
- timeout 引数を持つ READ がタイムアウトになる : \$TEST は 0 (FALSE) に設定されます。
- timeout 引数を持たない READ です。\$TEST は、以前の値のままになります。

\$TEST は、ユーザが設定することもできます。また、LOCK、OPEN、または JOB のタイムアウトで設定されることもあります。

READ が完了する前にタイムアウトになり、文字の一部が入力された場合 (可変長、または固定長読み取り)、入力文字は variable に格納されます。文字が何も入力されない場合 (可変長、または固定長読み取り)、InterSystems IRIS は variable を (必要な場合は) 定義し、NULL 文字列に設定します。文字が単一文字 READ に対して入力されない場合、InterSystems IRIS は variable を (必要な場合は) 定義し、-1 に設定します。

## 例

以下の例は、可変長形式の READ を使用して、ユーザから任意の文字列を受け取ります。形式制御 ! によって、プロンプトは新しい行に表示されます。

### ObjectScript

```
READ !,"Enter your last name: ",lname
```

以下の例は、単一文字形式の READ を使用して、ユーザから 1 文字を受け取り、それを ASCII コードを表す数値として格納します。

### ObjectScript

```
READ !,"Enter option number (1,2,3,4): ",*opt
WRITE !,"ASCII value input=",opt
WRITE !,"Character input=",$KEY
```

以下の例は、固定長形式の READ を使用して、ユーザから 3 文字だけを受け取ります。

### ObjectScript

```
READ !,"Enter your 3-digit area code: ",area#3
```

以下は、12 文字までの固定長の名前 (gname)、固定長 (1 文字) のミドル・イニシャル (iname)、任意の長さの姓 (fname) という 3 種類の名前の例です。gname 変数と iname 変数は、10 秒後にタイムアウトになるようにコード化されています。

### ObjectScript

```
READ "Given name:",gname#12:10,!,"
"Middle initial:",iname#1:10,!,"
"Family name:",fname
WRITE $TEST
```

読み取り処理がタイムアウトすると、READ コマンドは次の読み取り処理を開始します。最初の 2 つの読み取り処理は、タイムアウトするかどうかにかかわらず \$TEST を設定します。3 番目の読み取り処理は \$TEST を設定しないので、この例の \$TEST の値は 2 番目の読み取り処理の結果 (成功かタイムアウト) を反映します。

以下の例は、間接指定を使用して READ コマンドに関連付けられたプロンプトを動的に変更します。



## ObjectScript

```
PromptChoice
  READ "Type 1 for numbers or 2 for names:",p,#!!!!
  IF p'=1,p'=2 {WRITE !,"Invalid input" RETURN }
  ELSE {DO DataInput(p) }
DataInput(dtype)
  SET MESS(1)="ENTER A NUMBER:"
  SET MESS(2)="ENTER A NAME:"
  SET x=1
  READ !,@MESS(dtype),val(x)
  IF val(x)=" " {WRITE !,"Goodbye" RETURN }
  ELSE {
    IF dtype=1,l=$ISVALIDNUM(val(x)) { WRITE !,"You input number: ",val(x),! }
    ELSEIF dtype=2 { WRITE !,"You input string: ",val(x),! }
    ELSE { WRITE !,"That is not a number",! }
  }
  SET x=x+1
  DO DataInput(dtype)
}
```

以下の例は、入力された最初の数値の桁数を基にして、固定長読み取りの長さを設定します。

## ObjectScript

```
FirstNum
  READ "ENTER LARGEST INTEGER (and press Return): ",val(1)
  SET ibuf=$LENGTH(val(1))
  WRITE !,"Your largest number is: ",val(1),!
  DO OtherNums(ibuf)
OtherNums(digits)
  SET x=2
  READ !,"ENTER NEXT INTEGER: ",val(x)#digits
  IF val(x)=" " { WRITE !,"Goodbye" RETURN }
  ELSEIF val(x)>val(1) { WRITE !,"Number is too big",!
    DO OtherNums(digits) }
  ELSE { WRITE !,"You input: ",val(x),!
    SET x=x+1
    DO OtherNums(digits) }
}
```

## READ は現在のデバイスを使用する

READ は現在の入出力デバイスから文字指向のデータを入力します。ユーザは OPEN コマンドを使用してデバイスを開き、USE コマンドを使用してそのデバイスを現在のデバイスとして設定する必要があります。InterSystems IRIS は、現在のデバイス ID を \$IO 特殊変数で管理します。

READ が最もよく使用されるのは、キーボードからユーザ入力を取得する場合ですが、シーケンシャル・ディスク・ファイル、または通信バッファなどの**バイト指向のデバイス**から文字を入力するためにも使用できます。

## 行呼び出し

行呼び出しモードを使用すると、ターミナル・デバイス上の READ コマンドは、前に入力された行を入力として受け取ることができます。この呼び出された入力行は、その後編集できます。ユーザは、ユーザ指定の入力を終結するのと同じ方法で、呼び出された行の入力をインタラクティブに終了する必要があります。InterSystems IRIS は、可変長ターミナル読み取り (READ variable) と固定長ターミナル読み取り (READ variable#length) の両方で、行呼び出しをサポートしています。InterSystems IRIS は、単一文字ターミナル読み取り (READ \*variable) では行呼び出しをサポートしていません。現在のプロセスの行呼び出しをアクティブにするには、%SYSTEM.Process クラスの LineRecall() メソッドを使用します。システム全体の行呼び出しを既定に設定するには、Config.Miscellaneous クラスの LineRecall プロパティを使用します。**“ターミナル入出力”**の説明に従って、ターミナルの OPEN プロトコルおよび USE プロトコルを設定することもできます。

## READ ターミナータ

InterSystems IRIS は、指定された長さに入力文字列が達したときに読み取り処理を終了します (単一文字 READ および固定長 READ の場合)。可変長 READ の場合、InterSystems IRIS は、入力文字列が現在のプロセスの最大長に達したときに読み取り処理を終了します。

InterSystems IRIS は、特定のターミネータ文字に遭遇した場合にも、読み取りを終了します。ターミネータ文字は、デバイスの種類によって異なります。例えば端末では、既定のターミネータは RETURN (一般的には [Enter] キー)、LINE FEED (ASCII 10) および ESCAPE (ASCII 27) です。

デバイスに対して OPEN コマンドや USE コマンドを実行するとき、既定のターミネータを変更することができます。OPEN や USE を使用して、ターミネータ・パラメータ値を指定することができます。ターミナルの OPEN プロトコルおよび USE プロトコルについては、“[ターミナル入出力](#)”を参照してください。

- 可変長 READ：InterSystems IRIS は、入力ターミネータを入力値と共に格納せず、\$KEY 特殊変数と \$ZB 特殊変数に記録します。
- 固定長複数文字 READ：InterSystems IRIS は、入力ターミネータを入力値と共に格納せず、\$KEY 特殊変数と \$ZB 特殊変数に記録します。
- 単一文字 READ：InterSystems IRIS は、入力ターミネータ (指定されている場合) を単一文字読み取りの入力値として格納します。また、入力ターミネータを \$KEY 特殊変数と \$ZB 特殊変数にも記録します。

\$KEY 特殊変数と \$ZB 特殊変数はターミナルからコマンド行を読み取るたびに設定されるため、前の READ コマンドで設定された \$KEY 値と \$ZB 値は上書きされることに注意してください。

## タイムアウトと \$ZA、\$ZB、および \$TEST 特殊変数

InterSystems IRIS は以下のように、\$TEST、\$ZA、および \$ZB の特殊変数で READ の完了状況を記録します。

READ の種類	変数データ	\$TEST 値	\$ZA 値	\$ZB 値
可変長：改行で終了	入力文字 (ない場合は NULL 文字列)	1	0	ターミネータ文字
可変長：入力後にタイムアウト	入力文字	0	2	NULL 文字列
可変長：入力なしでタイムアウト	NULL 文字列	0	2	NULL 文字列
固定長：すべての文字を入力	入力文字	1	0	入力された最後の文字
固定長：改行	入力文字 (ない場合は NULL 文字列)	1	0	ターミネータ文字
固定長：タイムアウト	入力文字 (ない場合は NULL 文字列)	0	2	NULL 文字列
単一文字：データ入力	入力文字の ASCII 値	1	0	入力文字
単一文字：ターミネータ文字を入力	ターミネータ文字の ASCII 値	1	0 <Enter>, 256 <Esc>	ターミネータ文字
単一文字：タイムアウト	-1	0	2	NULL 文字列

## \$ZB と \$KEY

\$ZB と \$KEY 特殊変数は、すべての種類の読み取りに対してまったく同じ値を返しますが、1 つ例外があります。固定長読み取りを実行し、指定された文字数を入力するとき、READ はターミネータなしで終了します。この場合、\$ZB は入力された最後の文字 (終了文字) を含み、\$KEY は NULL 文字列を含みます (ターミネータ文字はありません)。

## 中断

READ の呼び出し時に保留中の **CTRL-C** 中断が存在する場合、READ は、ターミナルからの読み取り前にこの中断を破棄します。

処理中の読み取りが **CTRL-C** 中断によって中断される場合、variable は前の状態に戻ります。例えば、読み取り処理に複数の文字を入力し、**CTRL-C** で中断した場合、variable は読み取り処理前の状態に戻ります。つまり、未定義の場合は未定義のまま、以前に何か値を持つ場合は、その前の値を含むということです。この振る舞いは、タイムアウトになる読み取り処理とは完全に異なります。読み取りのタイムアウトには、タイムアウトになる前に入力された文字を含む、variable の新規の状態が保持されます。READ コマンドが複数の読み取り処理を含む場合、実行中の読み取り処理だけが中断の影響を受けます。複数の読み取り処理を 1 つのユニットとして実行、または戻すには、トランザクション処理を使用します。

**CTRL-C** 中断の有効化および無効化については、“[中断を有効または無効にする BREAK フラグ](#)” および “[\\$ZJOB](#)” 特殊変数を参照してください。

## キーボード以外のデバイスからの入力を受け取る READ を実行する

前述したように、READ は、すべての文字指向のデバイスからの入力の取得に使用できます。この中には、シーケンシャル・ディスク・ファイルや端末キーボードなどのデバイスも含まれます。しかし最初に、OPEN コマンドや USE コマンドを使用して、読み取りに使用するデバイスを現在のデバイスとして設定する必要があります。

キーボード以外のデバイスの場合、可変長、単一文字、固定長の 3 つの入力形式のいずれも使用できます。特定の状況でどの形式を選択するかは、利用できる[ターミネータの種類](#)によって異なります。

例えば、行指向の形式のデータに、行のターミネータとして、CARRIAGE RETURN/LINE FEED が含まれているデバイスから読み取る場合は、可変長フォームを使用できます。この場合、InterSystems IRIS は各行をそのまま全部 variable に読み取ります。入力を終了するのは、行末の [Return] (ASCII コード 13) に達したときだけです(前に示したユーザ入力の例を思い出してください。Enter は入力ターミネータです)。

一方、レコードが一連の固定長フィールドで表されているデバイスから読み取りを行う場合は、固定長 (variable#length) 形式を使用します。例えば、4 つのフィールドで構成されている、レコード形式を使用したデバイスを持っているとします。フィールドの最高文字数は、それぞれ 8 文字、12 文字、4 文字、6 文字とします。データを読み取るには、以下のようなコードを使用できます。

### ObjectScript

```
READ field1#8,field2#12,field3#4,field4#6
```

この場合、#n 値が、各フィールドの入力ターミネータを設定します。

特定のデバイスでどのターミネータを使用するかは、OPEN コマンド、または USE コマンドでそのデバイスに指定するデバイス・パラメータによって設定できます。

ブロック型のデータを読み取るとき、\$ZB 特殊変数は入出力バッファに残るバイトを含みます。この関数は、文字指向のデータを読み取るときに使用する方法とは完全に異なります。\$ZB は読み取りターミネータ文字や、ブロック型入出力中に最後に入力された文字などを含みません。詳細は“[\\$ZB](#)”を参照してください。

## 印字不能文字の読み取り

印字不能文字とは、ASCII 印字可能文字の標準範囲外の文字を指します。つまり、ASCII コードの ASCII 32 より小さいか、ASCII 127 より大きい文字ということです。これらの文字と対応する単独のキーは、標準のキーボードにはありません。

ASCII コードが ASCII 32 より小さい文字は、通常、制御操作に使用されます。これらは、Ctrl キーと組み合わせてのみ入力できます。例えば、ETX (ASCII 3) は、Ctrl-C として入力されます。キーボードから入力されるときは、BREAK をアサートするために使用されます。

コードが ASCII 127 より大きい文字は、通常、グラフィック操作に使用されます。原則として、これらの文字はキーボードからは入力できませんが、他の種類のデバイスから読み取りできます。例えば、ASCII 179 は、垂直線を表す文字を生成します。

READ コマンドを使用すると、標準の ASCII 印字可能文字と同じように、印字不能文字を入力できます。ただし、このような文字それぞれで使用する、エスケープ・シーケンスを処理するコードを含む必要があります。エスケープ・シーケンスは、Esc 文字 (ASCII 27) から始まる文字シーケンスです。例えば、有効な入力応答として、ユーザがファンクション・キーを押すことができるようにする READ コードを記述することができます。ファンクション・キーを押すと、エスケープ・シーケンスが生成されます。エスケープ・シーケンスは端末の種類ごとに異なります。

ObjectScript は、エスケープ・シーケンスを指定 variable ではなく、\$ZB や \$KEY 特殊変数に保存することで、エスケープ・シーケンスの入力をサポートします。例えば、ファンクション・キーが押された場合、InterSystems IRIS は Esc コード (ASCII 27) を \$ZB と \$KEY に保存します。エスケープ・シーケンスを処理するには、各 READ の後の \$ZB や \$KEY の現在の値をテストするコードを含める必要があります。これは、後続のコードがこれらの特殊変数を更新し、以前の値をすべて上書きするからです。\$ZB と \$KEY は類似していますが、まったく同じものではありません。詳細は、“\$KEY”を参照してください。エスケープ・シーケンスなどの印字不能文字を表示するには、ZZDUMP コマンド、または \$ASCII 関数を使用します。

## シーケンシャル・ファイルの最後

シーケンシャル・ファイルの最後に到達したときの READ の振る舞いは、システム全体の既定によって異なります。管理ポータルに進み、[システム管理]、[構成]、[追加の設定]、[互換性] の順に選択します。SetZEOF の現在の設定を表示して編集します。このオプションは、シーケンシャル・ファイルの読み込み時に InterSystems IRIS が予期しないファイル終端に到達したときの振る舞いを決めます。“true”に設定すると、InterSystems IRIS は、ファイルの最後に到達したことを示す \$ZEOF 特殊変数を設定します。“false”に設定すると、InterSystems IRIS は <ENDOFFILE> エラーを発行します。既定は “false” です。

現在のプロセスに対して、このファイルの最後に到達したときの動作を変更するには、%SYSTEM.Process クラスの SetZEOF() メソッドを使用します。システム全体に対して、ファイルの最後に到達したときの既定の動作を設定するには、Config.Miscellaneous クラスの SetZEOF プロパティを使用します。

## 関連項目

- [OPEN コマンド](#)
- [WRITE コマンド](#)
- [ZZDUMP コマンド](#)
- [USE コマンド](#)
- [\\$KEY 特殊変数](#)
- [\\$TEST 特殊変数](#)
- [\\$ZA 特殊変数](#)
- [\\$ZB 特殊変数](#)
- [\\$ZEOF 特殊変数](#)
- [ターミナル入出力](#)
- [シーケンシャル・ファイルの入出力](#)

# RETURN (ObjectScript)

ルーチンの実行を終了します。

## 構文

```
RETURN:pc expression
RET:pc expression
```

## 引数

引数	説明
pc	オプション – 後置条件式
expression	オプション – ObjectScript 式

## 説明

RETURN コマンドは、ルーチンの実行を終了するために使用されます。多くのコンテキストでは、[QUIT](#) コマンドと同義です。RETURN と QUIT は、FOR、DO WHILE、または WHILE フロー制御構造、あるいは TRY または CATCH ブロックの内部から発行されたときに、異なる動作をします。

- RETURN を使用すると、FOR、DO WHILE、WHILE の各ループまたは入れ子のループ構造内を含む任意の場所からルーチンの実行を終了させることができます。RETURN は常に現在のルーチンを終了し、呼び出し元のルーチンに戻るか、呼び出し元のルーチンがない場合はプログラムを終了します。RETURN は、コード・ブロック内から発行されたかどうかに関係なく、常に同じ動作を行います。これには、TRY ブロックまたは CATCH ブロックが含まれます。
- これに対して QUIT は、FOR ループ、DO WHILE ループ、WHILE ループ、あるいは TRY ブロックまたは CATCH ブロックの内部から発行されたときに、現在の構造のみを終了します。QUIT は、現在のブロック構造を終了し、そのブロック構造の外側にある次のコマンドで現在のルーチンの実行を続けます。QUIT は、ブロック構造の外部または IF、ELSEIF、ELSE の各コード・ブロックの内部から発行されたときに、現在のルーチンを終了します。

RETURN コマンドには、以下の 2 つの形式があります。

- [引数なし](#)
- [引数付き](#)

後置条件は、引数として考慮されません。[\\$QUIT](#) 特殊変数は、引数付きの RETURN コマンドが現在のコンテキストを終了する必要があるかどうかを示します。そのために、M16 “許可されていない引数で QUIT しました” と M17 “引数付きの QUIT が必要です” の 2 つの [\\$ECODE](#) エラー・コードが用意されています。

## 引数なしの RETURN

引数なしの RETURN は、値を返さずに現在のコンテキストを終了します。これは、DO または XECUTE コマンドで開始されたプロセスの実行レベルを終了するために使用されます。

DO または XECUTE がターミナル・プロンプトから呼び出された場合、RETURN は制御をターミナル・プロンプトに戻します。終了したプロセスで、RETURN の前に NEW コマンドが配置されている場合、RETURN は対象となっていた変数に自動的に KILL を実行し、変数を元の値にリストアします。

## 引数付きの RETURN

RETURN expression は、ユーザ定義関数またはオブジェクト・メソッドを終了し、指定された式の結果を返します。引数付きの RETURN を使用して、FOR、DO WHILE、または WHILE コマンド・ループの内部、あるいは TRY ブロックまたは CATCH ブロックの内部からルーチンを終了できます。

引数付きの RETURN がサブルーチン内で呼び出されると、以下のいずれかが発生します。

- ・ (関数の代わりに) サブルーチンの内部から引数付きの RETURN が呼び出された場合、RETURN 引数は評価され (副次的作用またはエラーが発生する場合あり)、引数の結果が破棄されます。実行はサブルーチンの呼び出し元に戻ります。
- ・ サブルーチンが DO によって呼び出され、その DO 引数の範囲にある場合は、RETURN はその引数 (およびその評価によって引き起こされるすべての副次的作用) を評価しますが、引数は返しません。例えば、RETURN 4/0 で終わる DO に呼び出されたサブルーチンは <DIVIDE> エラーを生成します。サブルーチンが引数付きの RETURN で終了する DO で呼び出されていて、\$ETRAP で終了する場合も、同じ動作が発生します。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。RETURN コマンドが引数を取らない場合、同じ行における後置条件と次のコマンドの間に 2 つ、またはそれ以上のスペースが必要です。

### expression

任意の有効な ObjectScript 式。これはユーザ定義の関数内でのみ使用され、呼び出し元のルーチンに評価結果を返します。

## 例

以下の 2 つの例では、フロー制御構造内から発行されたときの RETURN と QUIT の動作を比較します。RETURN は MySubroutine を終了して MyRoutine に戻ります。QUIT は、FOR ループを終了し、MySubroutine の残りを続けて実行してから MyRoutine に戻ります。

### ObjectScript

```
MyMain
WRITE "In the main routine",!
DO MySubroutine
WRITE "Returned to main routine",!
QUIT
MySubroutine
WRITE "In MySubroutine",!
FOR i=1:1:5 {
    WRITE "FOR loop:",i,!
    IF i=3 RETURN
    WRITE " loop again",!
}
WRITE "MySubroutine line not displayed with RETURN",!
QUIT
```



## ObjectScript

```
MyMain
  WRITE "In the main routine",!
  DO MySubroutine
  WRITE "Returned to main routine",!
  QUIT
MySubroutine
  WRITE "In MySubroutine",!
  FOR i=1:1:5 {
    WRITE "FOR loop:",i,!
    IF i=3 QUIT
    WRITE "  loop again",!
  }
  WRITE "MySubroutine line displayed with QUIT",!
  QUIT
```

以下の例では、最初の RETURN コマンドの実行は後置条件 (:x>46) によって制御されます。乱数が 46 より大きい場合は、InterSystems IRIS は Cube プロシージャを実行しません。最初の RETURN が後置条件を受け、文字列を num として呼び出し元のルーチンに返します。乱数が 46 以下の場合、2 つ目の RETURN が式  $x*x*x$  の結果を num として返します。

## ObjectScript

```
Main
  SET x = $RANDOM(99)
  WRITE "Number is: ",x,!
  SET num=$$Cube(x)
  WRITE "Cube is: ",num
  QUIT
Cube(x) RETURN:x>46 "a six-digit number."
  WRITE "Calculating the cube",!
  RETURN x*x*x
```

以下の 2 つの例では、TRY および CATCH を使用した QUIT と RETURN の動作を比較します。TRY ブロックは、0 による除算を試行するので、CATCH ブロックが呼び出されます。この CATCH ブロックには、QUIT または RETURN のいずれかで終了する入れ子の TRY ブロックが含まれます。デモンストレーションのために、これらのプログラムには“フォールスルー”を防ぐために推奨されているコード (QUIT または RETURN) は含まれていません。

RETURN はルーチンを終了します。そのため、入れ子の TRY ブロックと囲んでいるブロックをすべて終了し、TRY/CATCH 構造の外側にあるフォールスルー行を実行しません。

## ObjectScript

```
TRY {
  WRITE "In the TRY block",!
  SET x = 5/0
  WRITE "This line should never display"
}
CATCH expl {
  WRITE "In the CATCH block",!
  WRITE "Error Name: ", $ZCVT(expl.Name, "O", "HTML"), !
  TRY {
    WRITE "In the nested TRY block",!
    RETURN
  }
  CATCH exp2 {
    WRITE "In the nested CATCH block",!
    WRITE "Error Name: ", $ZCVT(expl.Name, "O", "HTML"), !
  }
  WRITE "RETURN does not display this outer CATCH block line",!
}
WRITE "fall-through at the end of the program"
```

QUIT は、入れ子の TRY ブロックを終了して囲んでいるブロックに移動し、CATCH ブロックの残りで実行を続けます。CATCH ブロックを完了すると、TRY/CATCH 構造の外側にあるフォールスルー行を実行します。

## ObjectScript

```
TRY {
    WRITE "In the TRY block",!
    SET x = 5/0
    WRITE "This line should never display"
}
CATCH exp1 {
    WRITE "In the CATCH block",!
    WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
    TRY {
        WRITE "In the nested TRY block",!
        QUIT
    }
    CATCH exp2 {
        WRITE "In the nested CATCH block",!
        WRITE "Error Name: ", $ZCVT(exp1.Name, "O", "HTML"),!
    }
    WRITE "QUIT displays this outer CATCH block line",!
}
WRITE "fall-through at the end of the program"
```

以下の例での RETURN コマンドの引数は、オブジェクト・メソッドです。InterSystems IRIS はメソッドの実行を終了して、呼び出し元のルーチンに制御を戻します。

## ObjectScript

```
RETURN inv.TotalNum()
```

## RETURN による変数のリストア

終了したプロセスで、RETURN の前に NEW コマンドが配置されている場合、RETURN は対象となっていた変数に自動的に KILL を実行し、変数を元の値にリストアします。

## 暗黙の RETURN

InterSystems IRIS は各ルーチンの最後に暗黙の RETURN を実行しますが、プログラムを読みやすくするために、明示的に QUIT を含めることもできます。

以下の場合、RETURN コマンドは必要ありません。InterSystems IRIS は自動的に暗黙の RETURN を発行し、異なるコード・ユニットの実行が“フォールスルー”しないように防ぎます。

- ・ InterSystems IRIS はルーチンの最後に暗黙の RETURN を実行します。
- ・ InterSystems IRIS はパラメータを持つラベルに遭遇した場合、暗黙の RETURN を実行します。パラメータを持つラベルは、括弧内にパラメータが何も含まれていない場合でも、括弧付きのラベルとして定義されます。すべてのプロシージャは、(パラメータが定義されていない場合でも) パラメータを持つラベルから始まります。多くのサブルーチンと関数にも、パラメータを持つラベルがあります。

どのような環境でも、明示的な RETURN はコード化できます。

## DO での振る舞い

RETURN は、DO コマンドによって呼び出されたサブルーチンの内部で実行された場合、サブルーチンを終了し、DO コマンドの次のコマンドに制御を戻します。

## XECUTE での振る舞い

RETURN は、XECUTE が実行されているコード行で実行された場合、その行の実行を終了し、XECUTE コマンドの次のコマンドに制御を戻します。引数は指定できません。

## ユーザ定義関数での振る舞い

RETURN は、ユーザ定義の関数で使用された場合、その関数を終了し、指定された式の結果の値を返します。expression 引数は必須です。



その用途は、ユーザ定義関数はパラメータ渡しの DO コマンドとほぼ同じです。DO コマンドとの違いは、式の値を、変数を介してではなく、直接返すことです。外部関数を呼び出すには、以下の形式を使用します。

```
$$name(parameters)
```

name には関数の名前を指定します。label、^routine、または label^routine の形式で指定できます。

parameters には、関数に渡される、コンマ区切りのパラメータ・リストを指定します。関数に関連付けられた label もパラメータ・リストを持っている必要があります。呼び出される関数のパラメータ・リストは、実パラメータ・リストと呼ばれます。関数ラベルのパラメータ・リストは、仮パラメータ・リストと呼ばれます。

## プログラム・スタックからのレベルの消去

RETURN は、コンテキスト・フレームをプロセスのコール・スタックから削除します。\$STACK 特殊変数は、コール・スタックのコンテキスト・フレームの現在の番号を含みます。

ターミナルから RETURN n を実行すると、プログラム・スタックから一部またはすべてのレベルを消去することができます。引数なしの RETURN は、スタックからすべてのレベルを消去します。この動作は、[ターミナル・プロンプトからの QUIT の発行](#)と同じです。以下に例を示します。

### Terminal

```
USER>NEW
USER 1S1>NEW
USER 2N1>NEW
USER 3N1>NEW
USER 4N1>WRITE $STACK
4
USER 4N1>RETURN 2
USER 2N1>WRITE $STACK
2
USER 2N1>RETURN
USER>
```

## 関連項目

- ・ [DO コマンド](#)
- ・ [DO WHILE コマンド](#)
- ・ [FOR コマンド](#)
- ・ [NEW コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [WHILE コマンド](#)
- ・ [XECUTE コマンド](#)
- ・ [\\$ECODE 特殊変数](#)
- ・ [\\$QUIT 特殊変数](#)
- ・ [\\$STACK 特殊変数](#)

## SET (ObjectScript)

値を変数に代入します。

### 構文

```
SET:pc setargument,...
S:pc setargument,...
```

setargument には、以下を指定できます。

```
variable=value
(variable-list)=value
```

### 引数

引数	説明
pc	オプション — 後置条件式
variable	対応する value に設定される変数。variable は、ローカル変数、プロセス・プライベート・グローバル、グローバル、オブジェクト・プロパティ、または特殊変数にできます。(すべての特殊変数がアプリケーションによって設定されるわけではありません。詳細は、それぞれの特殊変数のドキュメントを参照してください。)
variable-list	1 つ、または複数の variable 引数で構成されている、括弧で囲まれたコンマ区切りのリスト。variable-list のすべての variable 引数に、同じ value が代入されます。
value	リテラル値、または値に評価される有効な ObjectScript 式。 <a href="#">JSON オブジェクト</a> または <a href="#">JSON 配列</a> を指定できます。

### 説明

SET コマンドは、値を変数に代入します。単一の変数、または、2 つの構文形式の任意の組み合わせで、複数の変数を設定できます。variable=value の組み合わせのコンマ区切りのリストを指定することで、変数に値を代入できます。次に例を示します。

#### ObjectScript

```
SET a=1,b=2,c=3
WRITE a,b,c
```

SET a=value,b=value,c=value,... コマンドの 1 回の呼び出しで実行できる代入数には、制限はありません。

指定された変数が存在しない場合、SET は変数を作成し、値を代入します。指定された変数が存在する場合、SET は前の値を指定された値で置換します。SET は左から右の順に実行されるので、以下のように値をある変数に割り当てた後に、その変数を他の変数に割り当てることができます。

#### ObjectScript

```
SET a=1,b=a
WRITE a,b
```

値は、文字列、数値、[JSON オブジェクト](#)、[JSON 配列](#)、またはこれらの値のいずれかに評価される式です。“空”の変数を定義するには、変数を空の文字列 (”) の値に設定します。

## 複数の変数に同じ値を設定する方法

SET を使用して、複数の変数に同じ値を割り当てるには、括弧で囲んだ変数のコンマ区切りリストを指定します。次に例を示します。

### ObjectScript

```
SET (a,b,c)=1
WRITE a,b,c
```

これら 2 つの SET 構文形式は、任意の方法で組み合わせることができます。次に例を示します。

### ObjectScript

```
SET (a,b)=1,c=2,(d,e,f)=3
WRITE a,b,c,d,e,f
```

SET (a,b,c,...)=value の 1 回の呼び出しで実行できる最大代入数は、128 です。この数を超えると、<SYNTAX> エラーが返されます。

### 複数の変数の設定に関する制約

- **\$LIST**: SET (a,b,c,...)=value という構文を使用して、等号の左側にある \$LIST 関数に値を割り当てることはできません。割り当てようとすると、<SYNTAX> エラーが返されます。\$LIST を使用して項目の 1 つを設定する場合、SET a=value,\$LIST(mylist,n)=value,c=value,... という構文を使用する必要があります。
- **\$EXTRACT** および **\$PIECE**: \$EXTRACT 関数または \$PIECE 関数で相対オフセット構文を使用する場合、SET (a,b,c,...)=value という構文を使用して、等号の左側にあるこれらの関数に値を割り当てることはできません。相対オフセット構文では、アスタリスクが文字列の末尾を表し、\*-n および \*+n が文字列の末尾からの相対オフセットを表します。例えば、SET (x,\$PIECE(mylist,"^",3))=123 は有効ですが、SET (x,\$PIECE(mylist,"^",\*))=123 では <UNIMPLEMENTED> エラーが返されます。相対オフセットを使用してこれらの関数のいずれかを設定する場合は、SET a=value,b=value,c=value,... 構文を使用する必要があります。
- **オブジェクト・プロパティ**: SET (a,b,c,...)=value という構文を使用して、等号の左側にあるオブジェクト・プロパティに値を割り当てることはできません。割り当てようとすると、「 MyPackage.MyClass Set MyProp SET 」のようなメッセージと共に、<OBJECT DISPATCH> エラーが返されます。オブジェクト・プロパティを設定する場合は、SET a=value,oref.MyProp=value,c=value,... という構文を使用する必要があります。

## SET と添え字

添え字付きの個別の値 (配列ノード) を、ローカル変数、プロセス・プライベート・グローバル、またはグローバルに設定できます。添え字は順不同です。変数添え字レベルがまだ存在していない場合、SET はこれを作成し、値を代入します。各添え字レベルは独立した変数として処理されます。これらの添え字レベルのみが定義されます。次に例を示します。

### ObjectScript

```
KILL myarray
SET myarray(1,1,1)="Cambridge"
WRITE !,myarray(1,1,1)
SET myarray(1)="address"
WRITE !,myarray(1)
```

この例では、変数 myarray(1,1,1) と myarray(1) は定義され、値を持ちます。しかし、変数 myarray と myarray(1,1) は定義されず、呼び出されたときは <UNDEFINED> エラーが返されます。

既定では、NULL 添え字は設定できません。例えば、SET ^x(")=123 は、<SUBSCRIPT> エラーになります。ただし、%SYSTEM.Process.NullSubscripts() メソッドを設定して、グローバルおよびプロセス・プライベート・グローバル変数に NULL 添え字を許可することができます。ローカル変数に NULL 添え字を設定することはできません。

添え字の最大長は 511 文字です。この長さを超えると <SUBSCRIPT> エラーになります。

ローカル変数の添え字レベルの最大数は 255 です。グローバル変数の添え字レベルの最大数は添え字レベル名に応じて異なり、レベルが 255 を超える場合もあります。(直接に、または[間接演算](#)で) ローカル変数の添え字レベルを 255 より大きく設定しようすると、<SYNTAX> エラーが返されます。添え字付き変数の詳細は、“[グローバルについての正式な規則](#)” を参照してください。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### variable

ターゲット変数がまだ存在していない場合、SET はこれを作成し、値を代入します。既に存在する場合、SET は既存の値を代入する値と置き換えます。

value の評価の結果の値を受け取る変数です。この変数は、[ローカル変数](#)、[プロセス・プライベート・グローバル](#)、[グローバル変数](#)のいずれかです。ローカル変数、プロセス・プライベート・グローバル、またはグローバル変数は、添え字付きでも添え字なしでもかまいません (詳細は“[SET と添え字](#)”を参照)。グローバル変数は、拡張グローバル参照で指定できます (“[グローバルについての正式な規則](#)”を参照してください)。

\$ECODE、\$ETRAP、\$DEVICE、\$KEY、\$TEST、\$X、\$Y など、特定の[特殊変数](#)を指定できます。

ローカル変数、プロセス・プライベート・グローバル、グローバル変数、または特殊変数は、現在のプロセスに対して固有で、マッピングすると、すべてのネームスペースからアクセスできます。グローバル変数は、それを作成したプロセスが終了しても存続します。

グローバルは、作成元のネームスペースに対して固有です。既定では、SET コマンドにより現在のネームスペースにグローバルが割り当てられます。SET ^["Samples"]myglobal="Ansel Adams" のような構文を使用すると、SET を使用して別のネームスペースのグローバル (myglobal) を定義できます。存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行し、続けてグローバル名とデータベース・パスを表示します (例: <PROTECT> ^myglobal,c:\intersystems\iris\mgr\)

変数は、\$PIECE 関数、または \$EXTRACT 関数の引数で指定される、変数の一部、またはセグメントのいずれかです。

変数は、`obj.property` 構文または `.. property` 構文を使用してオブジェクト・プロパティとして表現できます。また、[\\$PROPERTY](#) 関数を使用して表現することもできます。以下の構文を使用して、[i%property インスタンス変数リファレンス](#)を設定できます。

### ObjectScript

```
SET i%propname = "abc"
```

SET は任意の長さの変数名を受け入れますが、長い変数名は値を割り当てる前に 31 文字に切り捨てられます。変数名の最初の 31 文字が一意ではない場合、この名前への切り捨てを行うと、以下の例のように、変数値が意図せずに上書きされる場合があります。

## ObjectScript

```
SET abcdefghi jklmnopqrstuvwxyz2abc="30 characters"
SET abcdefghi jklmnopqrstuvwxyz2abcd="31 characters"
SET abcdefghi jklmnopqrstuvwxyz2abcde="32 characters"
SET abcdefghi jklmnopqrstuvwxyz2abcdef="33 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abc // returns "30 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abcd // returns "33 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abcde // returns "33 characters"
WRITE !,abcdefghijklmnopqrstuvwxyz2abcdef // returns "33 characters"
```

特殊変数とは、システム・イベントによって設定されるものです。SET を使用して、特定の特殊変数に値を割り当てることができます。ただし、ほとんどの特殊変数には、SET を使用して値を割り当てることができません。個別の特殊変数に関する詳細は、リファレンスを参照してください。

変数の種類および名前付け規則の詳細は、“[変数](#)”を参照してください。

## value

リテラル値、または任意の有効な ObjectScript 式。通常、value は数式、または文字列式のいずれかです。value には、[JSON オブジェクト](#)または [JSON 配列](#)を指定できます。

- ・ 数値はキャノニック形式に変換されてから代入されます。先頭と末尾にあるゼロ、先頭にある + 記号、および末尾にある小数点を取り除かれます。科学的記数法からの変換および算術演算の評価が実行されます。
- ・ 文字列の値は引用符で囲みます。文字列は、内部の二重引用符が一重引用符に変換されない限り、未変更のまま代入されます。NULL 文字列 (“”) は、有効な value として評価されます。
- ・ 引用符で囲まれた数値はキャノニック形式に変換されないため、代入の前に算術演算は実行されません。
- ・ 関係式または論理式が使用された場合、InterSystems IRIS は式の結果である真偽値 (0 または 1) を代入します。
- ・ 値を返すオブジェクトのプロパティとメソッドは有効な式になります。[相対ドット構文 \(.\)](#)を使用して、変数にプロパティまたはメソッドの値を代入します。

## JSON 値

SET コマンドを使用して変数を JSON オブジェクトまたは JSON 配列に設定できます。JSON オブジェクトの場合、value は中括弧で区切られた [JSON オブジェクト](#)です。JSON 配列の場合、value は角括弧で区切られた [JSON 配列](#)です。

これらの区切り内では、リテラル値は JSON リテラルです。ObjectScript リテラルではありません。無効な JSON リテラルを使用すると <SYNTAX> エラーが発生します。

- ・ 文字列リテラル:JSON 文字列を二重引用符で囲む必要があります。特定の文字を JSON 文字列内のリテラルとして指定するには、¥ エスケープ文字を指定し、その後にリテラルを指定する必要があります。JSON 文字列に二重引用符リテラル文字が含まれている場合、この文字は ¥” と記述されます。JSON 文字列構文では、二重引用符 (¥”)、バックスラッシュ (¥¥)、およびスラッシュ (¥/) をエスケープできます。行スペース文字 (バックスペース (¥b)、フォームフィード (¥f)、改行 (¥n)、キャリッジ・リターン (¥r)、およびタブ (¥t)) もエスケープできます。すべての Unicode 文字は、6 文字のシーケンス (バックスラッシュ、その後に小文字の u、その後に 4 桁の 16 進数) で表すことができます。例えば、\u0022 はリテラルの二重引用符文字を指定します。¥u03BC は、ギリシア語の小文字の Mu を指定します。
- ・ 数値リテラル:JSON では、数値を ObjectScript キャノニック形式に変換しません。JSON には独自の変換規則と検証規則があります:先頭のマイナス符号は 1 つのみ許可されます。先頭のプラス符号は許可されません。複数の先頭符号は許可されません。“E” 科学的記数文字は許可されていますが、評価されません。先頭のゼロは許可されません。末尾のゼロは保持されます。小数点区切り文字の両側に数字を記述する必要があります。したがって、JSON 数値 0、0.0、0.4、および 0.400 は有効です。ゼロ値の負の符号は保持されます。

IEEE 浮動小数点数の場合、追加の規則が適用されます。詳細は、“[\\$DOUBLE](#)”関数を参照してください。

JSON の小数は、ObjectScript の数値とは異なる形式で格納されます。ObjectScript の浮動小数点数は、最大精度に達すると丸められ、末尾のゼロは削除されます。JSON でパックされた BCD の小数は精度の向上が可能になり、末尾のゼロは保持されます。以下に例を示します。

### ObjectScript

```
SET jarray=[1.23456789123456789876000,(1.23456789123456789876000)]
WRITE jarray.%ToJSON()
```

- 特別な値: JSON は、次の特殊な値をサポートしています: `true`、`false`、および `null`。これらは、引用符で囲まれていないリテラルとして小文字で指定する必要がある、リテラル値です。これらの JSON 特殊値は、変数を使用して指定することも、ObjectScript 式で指定することもできません。
- ObjectScript: JSON 配列要素または JSON オブジェクト値に、ObjectScript リテラルまたは式を含めるには、文字列全体を括弧で囲む必要があります。JSON オブジェクト・キーに ObjectScript を指定することはできません。ObjectScript と JSON は、異なるエスケープ・シーケンス規則を使用します。ObjectScript で二重引用符文字をエスケープするには、二重にします。以下の例では、JSON 文字列リテラルおよび ObjectScript 文字列リテラルが JSON 配列で指定されています。

### ObjectScript

```
SET jarray=["This is a \"good\" JSON string","(This is a \"good\" ObjectScript string)"]
WRITE jarray.%ToJSON()
```

次の JSON 配列の例では、ObjectScript ローカル変数を指定し、キャノニック形式への ObjectScript 数値変換を実行します。

### ObjectScript

```
SET str="This is a string"
SET jarray=[(str),(--0007.000)]
WRITE jarray.%ToJSON()
```

次の例では、JSON オブジェクト値で ObjectScript 関数を指定しています。

### ObjectScript

```
SET jobj={"firstname":"Fred","namelen":($LENGTH("Fred"))}
WRITE jobj.%ToJSON()
```

## JSON オブジェクト

value には、中括弧で区切られた JSON オブジェクトを指定できます。variable は、`3@%Library.DynamicObject` のように OREF に設定されます。**ZWRITE** コマンドを指定されたローカル変数名と共に使用して、JSON の値を表示できます。

### ObjectScript

```
SET jobj={"inventory123":"Fred's \"special\" bowling ball"}
ZWRITE jobj
```

`%Get()` メソッドを使用することで、この OREF で指定されたキーの値を取得できます。`%ToJSON()` メソッドを使用することで、この OREF を JSON オブジェクト値に解決できます。以下に例を示します。

### ObjectScript

```
SET jobj={"inventory123":"Fred's \"special\" bowling ball"}
WRITE "JSON object reference = ",jobj,!
WRITE jobj.%Get("inventory123")," (data value in ObjectScript format)",!
WRITE jobj.%Get("inventory123","json")," (data value in JSON format)",!
WRITE jobj.%ToJSON()," (key and data value in JSON format)"
```



有効な JSON オブジェクトの形式は、以下のとおりです。

- ・ 左中括弧で始まり、右中括弧で終わります。空のオブジェクト {} は有効な JSON オブジェクトです。
- ・ 中括弧の内部は、key:value ペアまたはコンマで区切られた key:value ペアのリストです。キー・コンポーネントと値コンポーネントの両方が JSON リテラルです。ObjectScript リテラルではありません。
- ・ キー・コンポーネントは JSON の引用符付きの文字列リテラルでなければなりません。括弧で囲まれた ObjectScript リテラルまたは式にすることはできません。
- ・ 値コンポーネントには、JSON 文字列または JSON 数値リテラルを指定できます。JSON リテラルは JSON 検証条件に従います。値コンポーネントには、括弧で囲まれた ObjectScript リテラルまたは式を指定できます。値コンポーネントは、文字列、数値、JSON オブジェクト、または JSON 配列を指定する定義済み変数として指定できます。値コンポーネントには、入れ子になった JSON オブジェクトまたは JSON 配列を含めることができます。値コンポーネントに、小文字の true、false、null の 3 つの特殊な JSON 値のいずれかを引用符なしのリテラルとして指定することもできます。ただし、変数を使用してこれらの特殊な JSON 値を指定することはできません。

以下はすべて有効な JSON オブジェクトです。{"name":"Fred"}、{"name":"Fred","city":"Bedrock"}、{"bool":true}、{"1":true,"0":false,"Else":null}、{"name":{"fname":"Fred","lname":"Flintstone"},"city":"Bedrock"}、{"name":["Fred","Wilma","Barney"],"city":"Bedrock"}

JSON オブジェクトでは、以下の例に示すように、null プロパティ名を指定し、これに値を割り当てることができます。

## ObjectScript

```
SET jobj={}
SET jobj."null"="This is the "null" property value"
WRITE jobj.%Get(""),!
WRITE "JSON null property object value = ",jobj.%ToJSON()
```

返される JSON 文字列は、リテラルの二重引用符文字に JSON エスケープ・シーケンス (\") を使用します。

%Set() メソッドを使用して、key:value のペアを JSON オブジェクトに追加できます。

%Get() メソッドを使用することで、指定されたキーの値をさまざまな形式で返すことができます。構文は、以下のとおりです。

```
jobj.%Get(keyname,default,format)
```

default 引数は、keyname が存在しない場合に返される値です。

format 引数は、戻り値の形式を指定します。format が指定されていない場合、値は ObjectScript 形式で返されます。format="json" の場合、値は JSON 形式で返されます。format="string" の場合、すべての文字列値と数値は ObjectScript 形式で返されますが、JSON の特別な値である true と false は、ブーリアン整数ではなく、JSON のアルファベット文字列として返されます。JSON の特別な値である null は、ObjectScript 形式で長さ 0 の NULL 文字列として返されます。詳細は、以下の例を参照してください。

```
SET x={"yep":true,"nil":null}
WRITE "IRIS: ",x.%Get("yep")," JSON: ",x.%Get("yep","", "json")," STRING: ",x.%Get("yep","", "string"),!
/* IRIS: 1 JSON: true STRING: true */
WRITE "IRIS: ",x.%Get("nil")," JSON: ",x.%Get("nil","", "json")," STRING: ",x.%Get("nil","", "string")
/* IRIS: JSON: null STRING: */
```

詳細は、“[JSON の使用](#)”を参照してください。

## JSON 配列

value には、角括弧で区切られた JSON 配列を指定できます。variable は、1@%Library.DynamicArray のように OREF に設定されます。ZWRITE コマンドを指定されたローカル変数名と共に使用して、JSON の値を表示できます。

## ObjectScript

```
SET jary=["Fred","Wilma","Barney"]
ZWRITE jary
```

%Get() メソッドを使用して、指定した配列要素の値 (0 からカウント) を OREF で取得できます。%Get(n) は ObjectScript 値を返し、%Get(n,"json") は JSON 値を返します。%Get(n,"no such element","json") は、指定した配列要素が存在しない場合に返す既定値を指定します。%ToJSON() 関数を使用することで、この OREF を JSON 配列値に解決できます。以下に例を示します。

## ObjectScript

```
SET jary=["Fred","Wilma","Barney"]
WRITE "JSON array reference = ",jary,!
WRITE jary.%Get(1)," (array element value in ObjectScript format)",!
WRITE jary.%Get(1,"json")," (array element value in JSON format)",!
WRITE jary.%ToJSON()," (array values in JSON format)"
```

有効な JSON 配列の形式は、以下のとおりです。

- ・ 開始角括弧で始まり、終了角括弧で終わります。空の配列 [] は有効な JSON 配列です。
- ・ 角括弧の内部は、要素またはコンマで区切られた要素のリストです。配列の各要素には、JSON 文字列または JSON 数値リテラルを指定できます。JSON リテラルは JSON 検証条件に従います。配列要素には、括弧で囲まれた ObjectScript リテラルまたは式を指定できます。配列の要素は、文字列、数値、JSON オブジェクト、または JSON 配列を指定する定義済み変数として指定できます。配列の要素には、1 つ以上の JSON オブジェクトまたは JSON 配列を含めることができます。配列の要素に、小文字の true、false、null の 3 つの特殊な JSON 値のいずれかを引用符なしのリテラルとして指定することもできます。ただし、変数を使用してこれらの特殊な JSON 値を指定することはできません。

以下はすべて有効な JSON 配列です。[1],[5,7,11,13,17],[ "Fred","Wilma","Barney" ],[true,false],[ "Bedrock",[ "Fred","Wilma","Barney" ]],[{"name":"Fred"}, {"name":"Wilma"}],[ {"name":"Fred","city":"Bedrock"}, {"name":"Wilma","city":"Bedrock"}],[ {"names":["Fred","Wilma","Barney"]} ]

%Push() メソッドを使用して、配列の最後に新しい要素を追加できます。%Set() メソッドを使用して、新しい配列要素を追加したり、既存の配列要素を位置によって更新したりできます。

詳細は、“[JSON の使用](#)”を参照してください。

## オブジェクトでの SET コマンド

以下は、3 つの SET コマンドを含む例です。1 番目は変数を OREF (オブジェクト参照) に設定し、2 番目は変数をオブジェクト・プロパティの値に、3 番目はオブジェクト・プロパティを値に設定しています。

## ObjectScript

```
SET myobj=##class(%SQL.Statement).%New()
SET dmode=myobj.%SelectMode
WRITE "Default select mode=",dmode,!
SET myobj.%SelectMode=2
WRITE "Newly set select mode=",myobj.%SelectMode
```

ドット構文は、オブジェクト式で使います。ドットは、オブジェクト参照とオブジェクト・プロパティ名あるいはオブジェクト・メソッド名の間に配置します。

オブジェクト・プロパティに変数を設定する、または現在のオブジェクトにオブジェクト・メソッドの値を設定するには、二重ドット構文を使います。

## ObjectScript

```
SET x=..LastName
```



指定したオブジェクト・プロパティが存在しない場合、InterSystems IRIS は <PROPERTY DOES NOT EXIST> エラーを発行します。二重ドット構文を使用し、現在のオブジェクトが定義されていない場合、InterSystems IRIS は <NO CURRENT OBJECT> エラーを発行します。

詳細は、“[オブジェクト特有の ObjectScript の機能](#)” を参照してください。

以下のコマンドは、x を、GetNodeName() メソッドが返す値に設定します。

### ObjectScript

```
SET x=##class(%SYS.System).GetNodeName()
WRITE "the current system node is: ",x
```

オブジェクトの SET コマンドは、以下のようにカスケード・ドット構文を持つ式を使用できます。

### ObjectScript

```
SET x=patient.Doctor.Hospital.Name
```

この例では、patient.Doctor オブジェクト・プロパティは、Name プロパティを含む Hospital オブジェクトを参照します。そのため、このコマンドは x を指定された患者の医者と、関連している病院名に設定します。同じカスケード・ドット構文は、オブジェクト・メソッドと一緒に使用できます。

オブジェクトの SET コマンドは、以下のデータ型プロパティ・メソッドのように、システム・レベル・メソッドで使用できます。

### ObjectScript

```
SET x=patient.NameIsValid(Name)
```

この例では、NameIsValid() メソッドは、現在の patient オブジェクトの結果を返します。つまり、NameIsValid() は、Name プロパティのデータ型検証のために生成されたブーリアン・メソッドです。そのため、このコマンドは指定した名前が有効な場合は x を 1 に、無効の場合は 0 に設定します。

## オブジェクト・メソッドを使用した SET

SET 式の左側でオブジェクト・メソッドを指定できます。以下の例では %Get() メソッドを指定しています。

### ObjectScript

```
SET obj=##class(test).%New() // Where test is class with a multidimensional property md
SET myarray=(obj)
SET index=0,subscript=2
SET myarray.%Get(index).md(subscript)="value"
IF obj.md(2)="value" {WRITE "success"}
ELSE {WRITE "failure"}
```

## オブジェクトへの変数のリストの設定

以下の例に示すように、SET をオブジェクトと共に使用する場合、代入を複数回行くと、リスト内のすべての変数が同じ OREF に設定されます。

### ObjectScript

```
SET (a,b,c)=##class(Sample.Person).%New()
```

### ObjectScript

```
SET (dyna1,dyna2,dyn3) = ["default","default"]
```

各変数に別個の OREF を割り当てるには、以下のように各代入に対して別々の SET コマンドを使用します。

## ObjectScript

```
SET a=##class(Sample.Person).%New()  
SET b=##class(Sample.Person).%New()  
SET c=##class(Sample.Person).%New()
```

## ObjectScript

```
SET dyna1 = ["default","default"]  
SET dyna2 = ["default","default"]  
SET dyna3 = ["default","default"]
```

また、以下に示すように、[#dim プリプロセッサ指示文](#)を使用して、リスト内のすべての変数を別々の OREF に割り当てることもできます。

```
#dim a,b,c As %ClassDefinition = ##class(Sample.Person).%New()  
  
#dim dyn1,dyn2,dyn3 As %DynamicArray = ["default","default"]
```

## 例

以下は、同じ SET コマンドで複数の引数を指定する例です。具体的には、このコマンドは 3 つの変数に値を代入します。引数は左から右の順に評価されます。

## ObjectScript

```
SET var1=12,var2=var1*3,var3=var1+var2  
WRITE "var1=",var1,!,"var2=",var2,!,"var3=",var3
```

以下は、SET コマンドの (variable-list)=value 形式の例です。複数の変数に同じ値を代入します。具体的には、このコマンドは 3 つの変数に、値 0 を代入します。

## ObjectScript

```
SET (sum,count,average)=0  
WRITE "sum=",sum,!,"count=",count,!,"average=",average
```

以下は、拡張グローバル参照を使用して、異なるネームスペースで添え字付きのグローバル変数を設定する例です。

## ObjectScript

```
NEW $NAMESPACE  
SET $NAMESPACE="%SYS"  
SET ^["user"]nametest(1)="fred"  
NEW $NAMESPACE  
SET $NAMESPACE="USER"  
WRITE ^nametest(1)  
KILL ^nametest
```

## 評価の順序

InterSystems IRIS は、SET コマンドの引数を必ず左から右の順に評価します。各引数に対しては、以下の順序で評価を実行します。

1. 変数名を決定するために、間接指定のオカレンス、または等号の左側の添え字を左から右の順に評価します。詳細は、[“間接 \(@\)” のリファレンス・ページ](#)を参照してください。
2. 等号の右側の式を評価します。
3. 等号の右側の式に、等号の左側の変数名、または参照を代入します。

## トランザクション処理

グローバル変数の SET は現在のトランザクションの一部としてジャーナリングされます。このグローバル変数の割り当ては、トランザクションのロールバックの際にロールバックされます。ローカル変数またはプロセス・プライベート・グローバル変数の SET はジャーナリングされません。したがって、この割り当てはトランザクション・ロールバックの影響を受けません。

## 定義済み変数および未定義変数

ObjectScript コマンドおよび関数の多くは、変数を定義してから参照する必要があります。既定では、未定義変数を参照しようとすると、<UNDEFINED> エラーが生成されます。定義済みオブジェクトを参照しようとすると、<PROPERTY DOES NOT EXIST> エラーまたは <METHOD DOES NOT EXIST> エラーが生成されます。これらのエラー・コードに関する詳細情報は、[\\$ZERROR](#) を参照してください。

%SYSTEM.Process.Undefined() メソッドを設定することで、未定義変数を参照する際の InterSystems IRIS の動作を変更できます。

READ コマンドと [\\$INCREMENT](#) 関数は、未定義変数を参照して値を代入できます。[\\$DATA](#) 関数の場合、未定義または定義済みの変数を使用して、その状態を返します。[\\$GET](#) 関数は、定義済み変数の値を返します。また、未定義変数に値を代入することもできます。

## \$PIECE や \$EXTRACT を使用した SET

SET では、等号のどちらの側にも [\\$PIECE](#) 関数や [\\$EXTRACT](#) 関数を使用できます。詳細は、["\\$PIECE"](#) と ["\\$EXTRACT"](#) を参照してください。

[\\$PIECE](#) 関数や [\\$EXTRACT](#) 関数を等号の右側で使用すると、変数から部分文字列を抽出し、その値を等号の左側に指定されている変数に代入します。[\\$PIECE](#) は、指定された区切り文字を使用して部分文字列を抽出し、[\\$EXTRACT](#) は、文字カウントを使用して部分文字列を抽出します。

例えば、変数 x に、文字列 "HELLO WORLD" が含まれているとします。以下のコマンドは、それぞれ部分文字列 "HELLO" を抽出し、それを変数 y および z に代入します。

### ObjectScript

```
SET x="HELLO WORLD"
SET y=$PIECE(x," ",1)
SET z=$EXTRACT(x,1,5)
WRITE "x=",x,"!", "y=",y,"!", "z=",z
```

等号の左側で使用された場合、[\\$PIECE](#) および [\\$EXTRACT](#) は、等号の右側の式から得た値をターゲット変数の指定された部分に挿入します。ターゲット変数の指定された部分に既に値が存在する場合は、挿入された値と置き換えられます。

例えば、変数 x に文字列 "HELLO WORLD" が含まれ、変数 y に文字列 "HI THERE" が含まれているとして、以下のコマンドを見てみましょう。以下のコマンドでは、

### ObjectScript

```
SET x="HELLO WORLD"
SET y="HI THERE"
SET $PIECE(x," ",2)=$EXTRACT(y,4,9)
WRITE "x=",x
```

[\\$EXTRACT](#) 関数は、変数 y から文字列 "THERE" を抽出し、[\\$PIECE](#) 関数は、それを変数 x の 2 番目のフィールド位置に挿入します。このとき、既存の文字列 "WORLD" が置き換えられます。変数 x の値は、文字列 "HELLO THERE" になります。

ターゲット変数が存在しない場合は、システムがこれを作成し、必要に応じて区切り文字 ([\\$PIECE](#) の場合)、またはスペース ([\\$EXTRACT](#) の場合) が埋め込まれます。

以下の例では、SET \$EXTRACT を使用して z の値を文字列 x と y に挿入します。既存の値は上書きされます。

### ObjectScript

```
SET x="HELLO WORLD"
SET y="OVER EASY"
SET z="THERE"
SET $EXTRACT(x,7,11)=z
SET $EXTRACT(y,*-3,*)=z
WRITE "edited x=",x,!
WRITE "edited y=",y
```

これで、変数 x には文字列 "HELLO THERE" が格納され、y には文字列 "OVER THERE" が格納されます。この例の SET \$EXTRACT 操作の 1 つは負のオフセット (\*-3) を使用するため、これらの操作は個別のセットとして実行する必要があります。変数の一部が負のオフセットを使用する場合は、複数の変数を括弧で囲んだ 1 つの SET で設定することができません。

以下の例では、ルート・ノードにクライアントの名前、従属ノードに町名と番地、および市が含まれた構成になっているグローバル配列 ^client があるとします。例えば、^client(2,1,1) には、配列に格納された 2 番目のクライアントの住所の市が含まれます。

さらに、city ノード (x,1,1) には、市、州の略称、郵便番号を識別するフィールド値が含まれており、フィールド分離文字としてコンマを使用しているとします。例えば、典型的な city ノード値は、"Cambridge,MA,02142" となります。以下のコードの 3 つの SET コマンドは、それぞれ \$PIECE コマンドを使用して、配列ノード値の特定の部分を適切なローカル変数に代入しています。いずれの場合も、\$PIECE は、コンマ (",") を文字列の分離文字として参照していることに注意してください。

### ObjectScript

```
ADDRESSPIECE
SET ^client(2,1,1)="Cambridge,MA,02142"
SET city=$PIECE(^client(2,1,1),"",1)
SET state=$PIECE(^client(2,1,1),"",2)
SET zip=$PIECE(^client(2,1,1),"",3)
WRITE "City is ",city,!
      "State or Province is ",state,!
      ",Postal code is ",zip
QUIT
```

\$EXTRACT 関数を使用して同じ処理を行うこともできますが、これは、フィールドが固定長で、その長さが分かっている場合に限りです。例えば、city フィールドには最高 9 文字のみ、state フィールドと zip フィールドはそれぞれ 2 文字と 5 文字が含まれることが分かっている場合、SET コマンドを、\$EXTRACT 関数を使用して、以下のように書き換えることができます。

### ObjectScript

```
ADDRESSEXTRACT
SET ^client(2,1,1)="Cambridge,MA,02142"
SET city=$EXTRACT(^client(2,1,1),1,9)
SET state=$EXTRACT(^client(2,1,1),11,12)
SET zip=$EXTRACT(^client(2,1,1),14,18)
WRITE "City is ",city,!
      "State or Province is ",state,!
      "Postal code is ",zip
QUIT
```

フィールド分離文字のコンマに対応するために、9 から 11、12 から 14 というように、途中で数字が飛んでいることに注意してください。

以下の例は、A (最初は 1 に設定されています) 内の最初の部分文字列を、文字列 "abc" と置き換えます。

## ObjectScript

```
StringPiece
SET A="1^2^3^4^5^6^7^8^9"
SET $PIECE(A,"^")="abc"
WRITE !,"A=",A
QUIT
```

A="abc^2^3^4^5^6^7^8^9"

以下の例は、\$EXTRACT を使用して、A の最初の文字 (この場合も 1) を文字列 "abc" と置き換えます。

## ObjectScript

```
StringExtract
SET A="123456789"
SET $EXTRACT(A)="abc"
WRITE !,"A=",A
QUIT
```

A="abc23456789"

以下の例は、A の 3 番目から 6 番目の部分に文字列 "abc" を設定し、変数 B の最初の文字を、文字列 "abc" と置き換えます。

## ObjectScript

```
StringInsert
SET A="1^2^3^4^5^6^7^8^9"
SET B="123"
SET ($PIECE(A,"^",3,6),$EXTRACT(B))="abc"
WRITE !,"A=",A,!,"B=",B
QUIT
```

A="1^2^abc^7^8^9"

B="abc23"

以下の例は、\$X、\$Y、\$KEY、および以前に定義されなかったローカル変数 A の 4 番目の部分を値 20 に設定します。また、ローカル変数 K には、\$KEY の現在の値が設定されます。A には、前の 3 つの部分とキャレット (^) の区切り文字が含まれています。

## ObjectScript

```
SetVars
SET ($X,$Y,$KEY,$PIECE(A,"^",4))=20,X=$X,Y=$Y,K=$KEY
WRITE !,"A=",A,!,"K=",K,!,"X=",X,!,"Y=",Y
QUIT
```

A="^^^20" K="20" X=20 Y=20

## \$LIST や \$LISTBUILD を使用した SET

\$LIST 関数は、リストの作成と操作を実行します。これらは、要素の区切り文字を使用する代わりに、リスト内の各要素の長さ (およびタイプ) をエンコードします。これは、コード化された長さの指定を使用して、リスト操作の間に指定されたリスト要素を抽出します。\$LIST 関数は区切り文字を使用しないので、この関数を使用して作成されたリストは \$PIECE や他の文字区切り関数には入力はいりません。

等号の右側で使用された場合、これらの関数は以下を返します。

- ・ \$LIST は、指定されたリストの指定された要素を返します。
- ・ \$LISTBUILD は、指定された引数ごとに 1 つの要素を含むリストを返します。

SET 引数で、等号の左側で使用された場合、これらの関数は以下のタスクを実行します。

- ・ SET \$LIST は、指定された要素を等号の右側で指定された値に置き換えます。

#### ObjectScript

```
SET A=$LISTBUILD("red","blue","green","white")
WRITE "Created list A=", $LISTTOSTRING(A), !
SET $LIST(A,2)="yellow"
WRITE "Edited list A=", $LISTTOSTRING(A)
```

#### ObjectScript

```
SET A=$LISTBUILD("red","blue","green","white")
WRITE "Created list A=", $LISTTOSTRING(A), !
SET $LIST(A,*-1,*)=$LISTBUILD("yellow")
WRITE "Edited list A=", $LISTTOSTRING(A)
```

SET \$LIST と括弧を使用して、複数の変数に同じ値を割り当てることはできません。

- ・ SET \$LISTBUILD は、単一の処理でリストの複数要素を取り出します。\$LISTBUILD の引数は変数で、各変数は、\$LISTBUILD パラメータ・リストの位置に対応するリスト要素を受け取ります。関連しない位置の変数名は削除されます。

以下の例は、(等号の右側にある)\$LISTBUILD が最初に使用され、リストを返します。(等号の左側にある)\$LISTBUILD は、そのリストから 2 つの要素を取り出すために使用され、適切な変数を設定します。

#### ObjectScript

```
SetListBuild
SET J=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(A,B)=J
WRITE "A=",A,!, "B=",B
```

この例では、A="red"、B="green" です。

## 関連項目

- ・ [\\$LIST 関数](#)
- ・ [\\$LISTBUILD 関数](#)
- ・ [\\$EXTRACT 関数](#)
- ・ [\\$PIECE 関数](#)
- ・ [\\$X 特殊変数](#)
- ・ [\\$Y 特殊変数](#)

# TCOMMIT (ObjectScript)

トランザクションの正常終了を表します。

## 構文

```
TCOMMIT:pc
TC:pc
```

## 引数

引数	説明
pc	オプション - 後置条件式。

## 概要

TCOMMIT は、対応する TSTART によって開始されたトランザクションの正常終了を表します。

TCOMMIT は \$TLEVEL 特殊変数の値をデクリメントし、\$TLEVEL が 0 に達した場合のみ、トランザクションを終了させます。InterSystems IRIS は、\$TLEVEL が 0 に達した場合のみ、トランザクションを終了させます。通常、TCOMMIT が TSTART と同じ回数呼び出されたときに、トランザクションは終了します。[入れ子になったトランザクション](#)の間に行われた変更は、\$TLEVEL が 0 になるまでコミットされません。

\$TLEVEL が既に 0 のときに TCOMMIT を呼び出すと、<COMMAND> エラーが返されます。このケースが該当するのは、処理中のトランザクションがないとき、TCOMMIT コマンドの値が TSTART コマンドの値よりも大きいとき、または TROLLBACK コマンドの後に TCOMMIT を呼び出した場合です。対応する \$ZERROR 値は、<COMMAND>、エラーの位置 (例えば +3^mytest)、およびデータ・リテラル \*NoTransaction から構成されます。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、[コマンド後置条件式](#)を参照してください。

## 例

TCOMMIT は、TROLLBACK コマンドおよび TSTART コマンドと共に使用できます。これら 3 つのトランザクション処理コマンドを一緒に使用方法の例については、["TROLLBACK"](#)と["TSTART"](#)を参照してください。

## 入れ子にされた TSTART / TCOMMIT

TSTART/TCOMMIT を呼び出したモジュール、またはそれらが呼び出すモジュールで発行された他の TSTART/TCOMMIT とは独立して、そのモジュールが TSTART/TCOMMIT の組み合わせを正常に実行できるように、InterSystems IRIS は TSTART/TCOMMIT コマンドの入れ子をサポートします。トランザクションの現在の入れ子のレベルは、特殊変数 \$TLEVEL で記録されます。一番外側の該当する TCOMMIT が実行されると、つまり \$TLEVEL が 0 に戻ると、トランザクションはコミットされます。

入れ子になった個々のトランザクションは TROLLBACK 1 を呼び出すことでロールバックでき、現在のすべてのトランザクションは TROLLBACK を呼び出すことでロールバックできます。TROLLBACK は、TSTART が発行されているレベル数に関係なく、有効なトランザクション全体をロールバックし、\$TLEVEL を 0 に設定します。



## 同期コミット

TCOMMIT コマンドは、ディスクへのトランザクションに関連するジャーナル・データのフラッシュを要求します。このディスク書き込み操作の完了を待つかどうかは、構成可能なオプションです。

- ・ 現在のプロセスに対してこのオプションを構成するには、%SYSTEM.Process.SynchCommit() メソッドを使用します。
- ・ システム全体でこのオプションを構成するには、管理ポータルで、[システム管理]、[構成]、[追加の設定]、[互換性]の順に選択します。[SynchCommit] の現在の設定を表示して編集します。“true” に設定すると、TCOMMIT はジャーナル・データの書き込み操作が完了するまで待機します。“false” に設定すると、TCOMMIT はジャーナル・データの書き込み操作の完了を待機しません。既定は“false”です。SynchCommit 設定の変更を有効にするには、再起動が必要です。

## SQL とトランザクション

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。

## 関連項目

- ・ [TROLLBACK コマンド](#)
- ・ [TSTART コマンド](#)
- ・ [\\$TLEVEL 特殊変数](#)
- ・ [トランザクション処理での ObjectScript の使用法](#)



# THROW (ObjectScript)

次の例外ハンドラへ明示的に例外をスローします。

## 構文

```
THROW oref THROW:pc oref
```

## 引数

引数	説明
pc	オプション - 後置条件式。
oref	オプション - 例外ハンドラへスローされるオブジェクト参照 (OREF)。オプションですが、指定することを強くお勧めします。

## 説明

THROW コマンドは明示的に例外をスローします。例外は、システム・エラー、[%Status 例外](#)、またはユーザ定義の例外です。この例外は、[%Exception.AbstractException](#) オブジェクトから継承するオブジェクト参照 (OREF) としてスローされます。THROW コマンドは、この例外を次の例外ハンドラへスローします。

THROW oref の使用には、2 つの方法があります。

- TRY/CATCH : THROW oref を使用して、[TRY のコード・ブロック内](#)から例外を明示的に通知し、TRY ブロックから対応する CATCH ブロックの例外ハンドラに実行を渡します。
- その他の例外ハンドラ : THROW oref を使用して、[TRY ブロック外](#)において例外を明示的に通知します。これにより、現在の例外ハンドラ (\$ZTRAP など) がトリガされます。oref は [\\$THROWOBJ](#) 特殊変数から取得できます。

注釈 [引数なし](#)で THROW を使用することは非推奨であり、新しいコードではお勧めしません。

## システム・エラー

InterSystems IRIS は、未定義の変数を参照するなどの実行時エラーが発生したときにシステム・エラーを発行します。システム・エラーは [%Exception.SystemException](#) オブジェクト参照を生成して、oref のプロパティである Code、Name、Location、Data、ならびに、\$ZERROR および \$ECODE 特殊変数を設定し、次のエラー・ハンドラに制御を渡します。このエラー・ハンドラは、CATCH 例外ハンドラ、または \$ZTRAP あるいは \$ETRAP エラー・ハンドラです。システム・エラーは暗黙的エラーであり、THROW を使用しません。

エラー・ハンドラ内において THROW を使用すると、システム・エラー・オブジェクトをさらに別のエラー・ハンドラにスローできます。これは信号再伝達システム・エラーと呼ばれます。

THROW は、実行スタックの制御を次のエラー・ハンドラに渡します。例外が [%Exception.SystemException](#) オブジェクトである場合、次のエラー・ハンドラは CATCH、\$ZTRAP または \$ETRAP のいずれのタイプでもかまいません。それ以外の場合、例外を処理するためには、CATCH である必要があります。そうでないと、<THROW> エラーが生成されます。

## 引数

### oref

例外オブジェクトへの参照。これは [%Exception.AbstractException](#) を継承する任意のクラスのインスタンスです。システム・エラーの例外オブジェクトは、クラス [%Exception.SystemException](#) のインスタンスです。ユーザ指定の例外オブジェクトは、[%Status 例外](#) オブジェクト ([%Exception.StatusException](#))、一般的な例外オブジェクト ([%Exception.General](#))、ま

たは SQL 例外オブジェクト(%Exception.SQL) のいずれかになります。ユーザ例外オブジェクトを作成してデータを入れるのは、プログラマの責任です。

OREFの詳細は、“[OREF の基本](#)”を参照してください。

## TRY ブロックからの THROW

THROW oref は、TRY ブロックから対応する CATCH ブロックに発行できます。これにより、ユーザ定義の例外が明示的に通知されます。これにより、TRY ブロックから対応する CATCH ブロックに実行が渡されます。スローされた oref は、CATCH ブロックの exceptionvar 引数として設定されます。

引数付きの THROW を CATCH の例外ハンドラから発行するには、CATCH 以外の例外ハンドラにスローするか、または TRY ブロック (および関連する入れ子になった CATCH ブロック) を CATCH の例外ハンドラ内で入れ子にし、この入れ子になった TRY ブロックから THROW を発行します。

## %Status 例外およびユーザ定義の例外

[%Status 例外](#)またはユーザ定義の例外をトラップするには、oref 引数として %Exception.AbstractException オブジェクトに基づいたオブジェクトを指定します。例外クラスを定義してから、%New() を使用してそのクラスのインスタンスを作成し、例外情報を提供します。このタイプの例外は、CATCH 例外ハンドラで処理される必要があります。CATCH が存在しない場合、システムは <THROW> エラーを生成します。

ユーザ定義の例外では、\$ZERROR の値も \$ECODE の値も変更されません。これらの特殊変数のいずれかを使用するには、プログラムで SET コマンドを使用して、これらの変数を明示的に設定する必要があります。

## 一般的な例外

InterSystems IRIS は、THROW 引数として指定できる一般的な例外を提供します。これは %Exception.AbstractException 抽象クラスの %Exception.General サブクラスです。この使用法は、以下の例を参照してください。

### ObjectScript

```
TRY {
    WRITE "In the TRY block",!!
    SET mygenex = ##class(%Exception.General).%New("My exception","999",,
        "My own special exception")
    THROW mygenex
    WRITE "This shouldn't display",!
}
CATCH stuff {
    WRITE "In the CATCH block",!
    WRITE stuff.Name,!
    WRITE stuff.Code,!
    WRITE stuff.Data,!
    WRITE "End of the CATCH block",!
    RETURN
}
```

## TRY ブロック外における THROW

THROW を TRY ブロック外から発行すると、InterSystems IRIS によって <THROW> エラー “<THROW>+3^myprog \*%Exception.General MyErr 999 My user-defined error” が生成されます。この THROW の使用は、エラーの再通知に役立ちます。

THROW で指定されるオブジェクト参照 (oref) は、\$THROWOBJ 特殊変数で保存されます。例：  
9@%Exception.General.\$THROWOBJ 値は、次の正常な THROW 処理、または SET \$THROWOBJ="" によってクリアされます。

以下の例では、THROW は \$ZTRAP の例外ハンドラに例外をスローします。

## ObjectScript

```

MainRou
    WRITE "In the Main Routine",!!
    SET $ZTRAP=^ErrRou
    SET mygenex = ##class(%Exception.General).%New("My exception","999",,
                                                "My own special exception")
    THROW mygenex
    WRITE "This shouldn't display",!
    RETURN

```

## ObjectScript

```

ErrRou
    WRITE "In $ZTRAP",!
    SET oref=$THROWOBJ
    SET $THROWOBJ=""
    WRITE oref.Name,!
    WRITE oref.Code,!
    WRITE oref.Data,!
    WRITE "End of $ZTRAP",!
    RETURN

```

## 引数なしの THROW

引数なしの THROW は、現在のシステム・エラーを再び信号で送り、次の例外ハンドラへ制御を渡します。現在のシステム・エラーは、\$ZERROR 特殊変数によって参照されるエラーです。したがって、引数なしの THROW はコマンド ZTRAP \$ZERROR と同等です。

どのシステム・エラーが現在のシステム・エラーであるかは変更されることがあるため、引数なしの THROW の使用はお勧めできません。例えば、エラー・ハンドラが \$ZERROR 値を変更した場合、または、エラー・ハンドラ自体がシステム・エラーを生成した場合にこれが発生する可能性があります。そのため、THROW oref を使用して、次の例外ハンドラへローされるシステム・エラーを明示的に指定することをお勧めします。

## 例

以下の例では、%Exception.General クラスのインスタンスを使用してユーザ定義の例外をスローします。ここでは、後置条件付き構文で THROW を使用方法を示しています。この例では、Exceptions メソッドを呼び出し、引数として 5 以上の数を渡すと、例外がスローされて catch ブロックの WRITE 文が実行されます。

## ObjectScript

```

ClassMethod Exceptions(x As %Integer)
{
    set ex = ##class(%Exception.General).%New()
    set ex.Name = "Demo Exception",
        ex.Code = 100000,
        ex.Data = "Tutorial Example"
    try {
        write !, "Hello!",
        throw:(x >= 5) ex    // throw the exception    }
    catch err {
        write !, "x: ", ?20, x,
            !, "Error name: ", ?20, err.Name,
            !, "Error code: ", ?20, err.Code,
            !, "Error location: ", ?20, err.Location,
            !, "Additional data: ", ?20, err.Data, !
    }
    write !, "Finished!"
}

```

以下の例では、InterSystems IRIS のエラー名が山括弧で囲まれ、これらの例が Web ブラウザから実行されるので、\$ZCVT(myerr.Name,"O","HTML") が使用されます。多くの別のコンテキストでは、myerr.Name は必要な値を返します。

以下の例では、TRY ブロックで誕生日が生成されます。未来の日付の誕生日が生成された場合、THROW の使用により一般的な例外が発行され、ユーザ定義の例外の OREF が汎用 CATCH ブロックに渡されます。(例外をスローする日付を生成するために、この例を複数回実行する必要がある場合があります)

## ObjectScript

```

TRY {
    WRITE "In the TRY block",!
    SET badDOB=##class(%Exception.General).%New("<BAD DOB>","999",,"Birth date is in the future")
    FOR x=1:1:20 { SET rndDOB = $RANDOM(7)_$RANDOM(10000)
        IF rndDOB > $HOROLOG { WRITE !,"Birthdate ", $ZDATE(rndDOB,1,,4)," is invalid"
            THROW badDOB }
        ELSE { WRITE "Birthdate ", $ZDATE(rndDOB,1,,4)," is valid",! }
    }
}
CATCH err {
    WRITE !,"In the CATCH block"
    WRITE !,"Error code=",err.Code
    WRITE !,"Error name=", $ZCVT(err.Name,"O","HTML")
    WRITE !,"Error data=",err.Data
    RETURN
}

```

以下の例では、ユーザ定義の引数を使用した 2 つの THROW コマンドのいずれかを発行できます。\$RANDOM は、どの THROW を発行するか (ランダムな値 0 または 1)、または THROW を発行しないか (ランダムな値 2) を選択します。ブロックの実行が RETURN コマンドで終了しない限りは、コードの実行は TRY / CATCH ブロックの後に続行されます。

## ObjectScript

```

TRY {
    SET errdatazero="this is the zero error"
    SET errdataone="this is the one error"
    /* Error Randomizer */
    SET test=$RANDOM(3)
    WRITE "Error test is ",test,!
    IF test=0 {
        WRITE !,"Throwing exception 998",!
        THROW ##class(Sample.MyException).%New("TestZeroError",998,,errdatazero)
        THROW myvar
    }
    ELSEIF test=1 {
        WRITE !,"Throwing exception 999",!
        THROW ##class(Sample.MyException).%New("TestOneError",999,,errdataone)
    }
    ELSE { WRITE !,"No THROW error this time" }
}
CATCH exp {
    WRITE !,"This is the exception handler"
    WRITE !,"Error code=",exp.Code
    WRITE !,"Error name=",exp.Name
    WRITE !,"Error data=",exp.Data
    RETURN
}
WRITE !,"Execution after TRY block continues here"

```

以下の例は、システム・エラーが発生したときの THROW の使用法を示しています。THROW は、一般的に、システム・エラーを別のハンドラに転送するために CATCH 例外ハンドラで使用されます。これは、システム・エラーが予期しないタイプのシステム・エラーを受け取ったときに行われる可能性があります。これを行う場合、TRY ブロック (および対応する CATCH ブロック) を CATCH ブロック内に入れ子にする必要があることに注意してください。これは、入れ子になった CATCH ブロックにシステム・エラーを THROW するために使用されます。以下の例でこれを示します。Calculate を呼び出して除算演算を実行して、その答えを返します。考えられる結果は 3 種類です。その 1 番目として、y が任意のゼロ以外の数値の場合、除算演算は成功し、CATCH ブロック・コードは実行されません。2 番目として、y がゼロ (または非数値文字列) の場合、除算演算では、ゼロでの除算が試みられ、その CATCH ブロックにシステム・エラーがスローされます。これは、このエラーを“修正して”0 値を返す calcerr 例外ハンドラによって捕捉されます。3 番目として、y が定義されていない場合 (NEW y)、calcerr は予期しないシステム・エラーを捕捉して、このエラーを myerr 例外ハンドラへスローします。この 3 つの結果を示すために、このサンプル・プログラムでは \$RANDOM を使用して除数 (y) を設定しています。

## ObjectScript

```

Randomizer
SET test=$RANDOM(3)
IF test=0 { SET y=0 }
ELSEIF test=1 { SET y=7 }
ELSEIF test=2 { NEW y }

```

```

/* Note: if test=2, y is undefined */
Main
SET x=4
TRY {
    SET result=$$Calculate(x,y)
    WRITE !,"Calculated value=",result
}
CATCH myerr {
    WRITE !,"this is the exception handler"
    WRITE !,"Error code=",myerr.Code
    WRITE !,"Error name=", $ZCVT(myerr.Name,"O","HTML")
    WRITE !,"Error data=",myerr.Data
}
QUIT

Calculate(arg1,arg2) PUBLIC {
    TRY {
        SET answer=arg1/arg2
    }
    CATCH calcerr {
        WRITE "In the CATCH Block",!
        TRY {
            IF calcerr.Name="<DIVIDE>" {
                WRITE !,"handling zero divide error"
                SET answer=0 }
            ELSE { THROW calcerr }
            RETURN
        }
        CATCH {
            WRITE "Unexpected error",!
            WRITE "Error name=", $ZCVT(myerr.Name,"O","HTML"),!
        }
    }
    QUIT answer
}

```

## 関連項目

- ・ [CATCH コマンド](#)
- ・ [TRY コマンド](#)
- ・ [ZTRAP コマンド](#)
- ・ [\\$ETRAP 特殊変数](#)
- ・ [\\$THROWOBJ 特殊変数](#)
- ・ [\\$ZERROR 特殊変数](#)
- ・ [\\$ZTRAP 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)

## TROLLBACK (ObjectScript)

失敗に終わったトランザクションをロールバックします。

### 構文

```
TROLLBACK:pc
TRO:pc

TROLLBACK:pc 1
TRO:pc 1
```

### 引数

引数	説明
pc	オプション - 後置条件式
1	オプション - 整数 1。1 レベルの入れ子のロールバック。リテラルとして指定する必要があります。

### 説明

TROLLBACK は現在のトランザクションを終了し、[ジャーナルされたすべてのデータベース値](#)を、トランザクションの開始時に保持されていた値にリストアします。TROLLBACK には、以下の 2 つの形式があります。

- ・ TROLLBACK は、TSTART が発行されているレベル数に関係なく、処理中のすべてのトランザクションをロールバックし、\$TLEVEL を 0 にリセットします。
- ・ TROLLBACK 1 は、入れ子になったトランザクションの現在のレベル (最新の TSTART によって開始されたレベル) をロールバックし、\$TLEVEL を 1 つデクリメントします。この 1 という引数は、リテラル値の 1 でなければならず、1 に解決される変数や式にすることはできません。1 以外の数字はサポートされていません。

[\\$TLEVEL](#) 特殊変数から、入れ子になったトランザクションのレベルを決定できます。\$TLEVEL が 0 に設定されるときに呼び出される TROLLBACK は、何も影響を与えません。

%SYS.Journal.System クラスの GetImageJournalInfo() メソッドを使用して、ジャーナル・ファイル内で TSTART コマンドを検索し、開いているトランザクションを特定できます。TSTART は \$TLEVEL をインクリメントし、ジャーナル・ファイル・レコードに書き込みます。その場合、\$TLEVEL が 0 であれば、“BT” (Begin Transaction) レコードが、\$TLEVEL が 0 よりも大きければ、“BTL” (Begin Transaction with Level) レコードが書き込まれます。%SYS.Journal.System クラスの Sync() メソッドを使用して、成功したロールバック処理の後のジャーナル・バッファをフラッシュします。

TROLLBACK は、ロールバック処理の間、**Ctrl-C** による割り込みを無効にします。

### ロールバックの対象

TROLLBACK は、ジャーナルされたすべてのオペレーションをロールバックします。そのため、トランザクション内で加えられた変更に以下のような影響があります。

- ・ TROLLBACK は、SET オペレーションや KILL オペレーションを含め、グローバル変数に対する大半の変更をロールバックします。例外は次のリストに示します。

**注釈** 既定では、グローバル変数の SET または KILL は、他のプロセス (トランザクションの外部で実行されている可能性あり) から即座に確認できます。グローバル変数の SET または KILL が、他のプロセスから見られないようにするには、LOCK コマンドを使用してグローバル変数へのアクセスを調整する必要があります。

- ・ TROLLBACK は、トランザクション中の変更をグローバル変数の[ビット文字列値](#)にロールバックします。ただし、グローバル変数のビット文字列は、ロールバック操作によって以前の内部文字列表現に戻りません。



- ・ TROLLBACK は、挿入、更新、および削除の変更を SQL データにロールバックします。

ただし、アプリケーションによって行われたすべての変更がジャーナルされるわけではないため、TROLLBACK は、トランザクション内で実行される場合であっても、以下の変更はロールバックしません。

- ・ ローカル変数またはプロセス・プライベート・グローバルに対する変更
- ・ 特殊変数 (\$TEST など) に対する変更
- ・ 現在のネームスペースに対する変更
- ・ LOCK コマンドによるロックまたはアンロック・オペレーション
- ・ グローバル変数に対する \$INCREMENT の変更
- ・ グローバル変数に対する \$SEQUENCE の変更
- ・ 外部からデータベースに加えられた変更

## トランザクション・ロールバックのロギング

ロールバック処理中にエラーが発生した場合は、<ROLLFAIL> エラー・メッセージが発行され、messages.log オペレータ・メッセージ・ログ・ファイルにエラー・メッセージが記録されます。管理ポータル内の [システムオペレーション] オプションを使用して、messages.log を表示できます ([システムオペレーション]、[システムログ]、[メッセージログ])。

既定では、messages.log オペレータ・メッセージ・ログ・ファイルは、InterSystems IRIS システム管理ディレクトリ (mgr) にあります。この既定の場所は構成可能です。管理ポータル内の [システム管理] オプションに移動して、[構成]、[追加の設定]、[メモリ詳細] の順に選択します。[ConsoleFile] の現在の設定を表示して編集します。既定では、この設定は空白で、コンソール・メッセージは mgr ディレクトリの messages.log に転送されます。messages.log に異なるディレクトリの場所を指定することもできます。

## <ROLLFAIL> エラー

TROLLBACK がトランザクションを正常にロールバックできない場合、<ROLLFAIL> エラーが発生します。このプロセスの動作は、システム全体のジャーナル構成設定フラグ [エラー発生時に凍結する] (管理ポータルから、[システム管理]、[構成]、[システム構成]、[ジャーナル設定] の順に選択) の設定に応じて以下のように異なります。

- ・ [エラーの発生時に凍結する] が設定されていない場合 (既定)、プロセスは <ROLLFAIL> エラーを受け取ります。トランザクションは閉じられ、そのトランザクションに対して保持されているすべてのロックが解除されます。このオプションでは、データ整合性よりシステムの可用性が優先されます。
- ・ [エラーの発生時に凍結する] が設定されている場合、プロセスは停止し、ジョブ削除デーモン (CLNDMN) が、開いているトランザクションのロールバックを再試行します。CLNDMN が再試行している間、トランザクションに対して保持されているロックは変更されませんが、システムは停止する場合があります。このオプションでは、システムの可用性よりデータ整合性が優先されます。

詳細は、“データ整合性ガイド” の “ジャーナル入出力エラー” を参照してください。

<ROLLFAIL> が発生する場合、%msg は、<ROLLFAIL> エラーそのものと、ロールバックを発生させた直前のエラーを記録します。例えば、日付を範囲外の値で更新しようとしてロールバックに失敗すると、以下の %msg が返されます。

```
SQLCODE = -105 %msg = Unexpected error occurred: <ROLLFAIL>%0Ac+1^dpv during
TROLLBACK.Previous error: SQLCODE=-105, %msg='Field 'Sample.Person.DOB' (value '5888326')
failed validation'.
```

<ROLLFAIL> は、トランザクション内で、グローバルがリモート・データベースにアクセスした後、プログラムが明示的にそのリモート・データベースをディスマウントした場合のトランザクション・ロールバック時に発生します。

<ROLLFAIL> は、データベース変更を行う前にプロセスがジャーナリングを無効にして、トランザクション・ロールバックを起動したエラーが発生した場合のトランザクション・ロールバック時に発生します。<ROLLFAIL> は、すべてのデータベース変更が行われた後、TROLLBACK コマンドを発行する前にプロセスがジャーナリングを無効にした場合のトラン

ザクション・ロールバック時には発生しません。その代わりに、InterSystems IRIS は、ロールバック処理中にジャーナリングを一時的に有効にします。ロールバック処理の完了時に、InterSystems IRIS はジャーナリングを再び無効にします。

## 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、TROLLBACK によってロールバックできません。ただし、TROLLBACK 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。

## SQL とトランザクション

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。

### クエリ・キャッシュの削除

トランザクションの際に %SYSTEM.SQL クラスの Purge() メソッドを呼び出してクエリ・キャッシュを削除すると、クエリ・キャッシュは永久に削除されます。後続の TROLLBACK では、削除されたクエリ・キャッシュはリストアされません。

## グローバルと TROLLBACK 1

TROLLBACK 1 は、入れ子になったトランザクション内で変更されたすべてのグローバルをロールバックしてリストアします。ただし、入れ子のトランザクションをサポートしないリモート・システムにマップされたグローバルが変更された場合、その変更は入れ子の最も外側で生じたものとして処理されます。こうしたグローバルは、TROLLBACK が呼び出されるか、または \$TLEVEL が 1 のときに TROLLBACK 1 が呼び出されることで、\$TLEVEL が 0 にリセットされたときのみロールバックされます。

## ロックと TROLLBACK 1

TROLLBACK 1 は、入れ子になったトランザクションの実行中に設定されたロックを以前の状態にリストアしません。トランザクションの実行中に設定されたすべてのロックは、レベル 0 への TROLLBACK または TCOMMIT によってトランザクションが終了するまで、ロック・テーブルに残ります。その時点で、InterSystems IRIS は入れ子になったトランザクションの実行中に作成されたすべてのロックを解放し、既存のすべてのロックを TSTART より前の状態にリストアします。

入れ子になったトランザクションの TCOMMIT は対応するロックを解除しないため、次の TROLLBACK はコミットされたサブトランザクション内でロックを実行する場合があります。

## 引数

### pc

オプションの後置条件式です。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。



## 例

以下の例では、口座間でランダムな金額を送金する、単一レベルのトランザクションを使用しています。送金額が利用可能な残高を超えている場合、プログラムは TROLLBACK を使用して以下のようにトランザクションをロールバックします。

### ObjectScript

```
SetupBankAccounts
    SET num=12345
    SET ^CHECKING(num,"balance")=500.99
    SET ^SAVINGS(num,"balance")=100.22
    IF $DATA(^NumberOfTransfers)=0 {SET ^NumberOfTransfers=0}
BankTransfer
    WRITE "Before transfer:",!, "Checking=$", ^CHECKING(num,"balance"), " Savings=$", ^SAVINGS(num,"balance"),!

    // Transfer funds from one account to another
    SET transfer=$RANDOM(1000)
    WRITE "transfer amount $",transfer,!
    DO CkToSav(num,transfer)
    IF ok=1 {WRITE "successful transfer",!, "Number of transfers to date=", ^NumberOfTransfers,!}
    ELSE {WRITE "*** INSUFFICIENT FUNDS ***",!}
    WRITE "After transfer:",!, "Checking=$", ^CHECKING(num,"balance"), " Savings=$", ^SAVINGS(num,"balance"),!

    RETURN
CkToSav(acct,amt)
    TSTART
    SET ^CHECKING(acct,"balance") = ^CHECKING(acct,"balance") - amt
    SET ^SAVINGS(acct,"balance") = ^SAVINGS(acct,"balance") + amt
    SET ^NumberOfTransfers=^NumberOfTransfers + 1
    IF ^CHECKING(acct,"balance") > 0 {TCOMMIT SET ok=1 QUIT:ok}
    ELSE {TROLLBACK SET ok=0 QUIT:ok}
```

以下の例では、入れ子になったトランザクションでの TROLLBACK の影響を示します。各 TSTART は \$TLEVEL をインクリメントし、グローバルを設定します。内側の入れ子になったトランザクションで TCOMMIT を発行すると、\$TLEVEL がデクリメントされますが、入れ子になったトランザクションでの変更のコミットは延期されます。この場合、外側のトランザクションにある後続の TROLLBACK により、すべての変更がロールバックされます。このロールバックには、内側の“コミット済みの”入れ子になったトランザクションで行われた変更も含まれます。

### ObjectScript

```
SET ^a(1)=[- - -],^b(1)=[- - -]
WRITE !,"level:",$TLEVEL," ",^a(1)," ",^b(1)
TSTART
    LOCK +^a(1)
    SET ^a(1)="hello"
    WRITE !,"level:",$TLEVEL," ",^a(1)," ",^b(1)
    TSTART
        LOCK +^b(1)
        SET ^b(1)="world"
        WRITE !,"level:",$TLEVEL," ",^a(1)," ",^b(1)
    TCOMMIT
WRITE !,"After TCOMMIT"
WRITE !,"level:",$TLEVEL," ",^a(1)," ",^b(1)
TROLLBACK
WRITE !,"After TROLLBACK"
WRITE !,"level:",$TLEVEL," ",^a(1)," ",^b(1)
QUIT
```

以下の例では、TROLLBACK によってグローバル変数がロールバックされながら、ローカル変数がロールバックされないことを示します。

## ObjectScript

```
SET x="default",^y="default"
WRITE !,"level:",$TLEVEL
WRITE !,"local:",x," global:",^y
TSTART
SET x="first",^y="first"
WRITE !,"TSTART level:",$TLEVEL
WRITE !,"local:",x," global:",^y
TSTART
SET x=x_ second",^y=y_ second"
WRITE !,"TSTART level:",$TLEVEL
WRITE !,"local:",x," global:",^y
TSTART
SET x=x_ third",^y=y_ third"
WRITE !,"TSTART level:",$TLEVEL
WRITE !,"local:",x," global:",^y
TROLLBACK
WRITE !,"After Rollback:"
WRITE !,"TROLLBACK level:",$TLEVEL
WRITE !,"local:",x," global:",^y
```

以下の例では、グローバルに対する \$INCREMENT 変更がロールバックされないことを示します。

## ObjectScript

```
SET ^x=-1,^y=0
WRITE !,"level:",$TLEVEL
WRITE !,"Increment:",$INCREMENT(^x)," Add:",^y
TSTART
SET ^y=^y+1
WRITE !,"level:",$TLEVEL
WRITE !,"Increment:",$INCREMENT(^x)," Add:",^y
TSTART
SET ^y=^y+1,^z=^z_ second"
WRITE !,"level:",$TLEVEL
WRITE !,"Increment:",$INCREMENT(^x)," Add:",^y
TSTART
SET ^y=^y+1,^z=^z_ third"
WRITE !,"level:",$TLEVEL
WRITE !,"Increment:",$INCREMENT(^x)," Add:",^y
TROLLBACK
WRITE !,"After Rollback"
WRITE !,"level:",$TLEVEL
WRITE !,"Increment:",^x," Add:",^y
```

## 関連項目

- ・ [TCOMMIT コマンド](#)
- ・ [TSTART コマンド](#)
- ・ [\\$TLEVEL 特殊変数](#)
- ・ [トランザクション処理での ObjectScript の使用法](#)

# TRY (ObjectScript)

実行中のエラーを監視するコード・ブロックを指定します。

## 構文

```
TRY {
    . . .
}
```

## 説明

TRY コマンドは引数を取りません。中括弧で囲まれた ObjectScript コード文のブロックを識別するために使用されます。このコード・ブロックは、構造化された例外処理のための保護されたコードです。このコード・ブロック内で例外が発生した場合、InterSystems IRIS は \$ZERROR と \$ECODE を設定し、[CATCH](#) コマンドによって識別される、例外ハンドラに実行を移します。

例外は、0 による除算の試みなどの実行時エラーの結果として発生する場合があります。また例外は、[THROW](#) コマンドの発行により明示的に伝達されない場合があります。エラーが発生しなかった場合、CATCH コード・ブロックの後の次の ObjectScript 文で実行を続けます。

CATCH ブロックのすぐ後に、TRY ブロックが続く必要があります。TRY コード・ブロックと CATCH コマンドの閉じ中括弧の間に、実行可能コード文またはラベルを指定できません。しかし、TRY ブロックとその CATCH ブロックの間にコメントを指定できます。CATCH ブロックは 1 つのみが、各 TRY ブロックに対して許可されます。ただし、以下のようにペアになった TRY/CATCH ブロックを入れ子にすることはできます。

### ObjectScript

```
TRY {
    /* TRY code */
    TRY {
        /* nested TRY code */
    }
    CATCH {
        /* nested CATCH code */
    }
}
CATCH {
    /* CATCH code */
}
```

通常、ObjectScript プログラムは、複数の TRY ブロックで構成され、各 TRY ブロックのすぐ後に、関連付けられた CATCH ブロックが続きます。

## QUIT および RETURN

TRY ブロックは、[QUIT](#) または [RETURN](#) を使用して終了できます。QUIT は、現在のブロック構造を終了し、ブロック構造の外側にある次のコマンドで実行を続けます。たとえば、入れ子の TRY ブロック内で QUIT を発行すると、TRY ブロックを終了して囲んでいるブロック構造に移動します。TRY ブロック内で QUIT コマンドを発行すると、対応する CATCH ブロックの後の最初のコード行に実行を移します。引数付きの QUIT を使用して、TRY ブロックを終了することはできません。これを行うと、コンパイル・エラーが返されます。TRY ブロック内からルーチンを完全に終了するには、RETURN 文を発行します。

TRY ブロックの QUIT または RETURN コマンドで例外が生成される場合がまれにあります。これは、TRY が新しいコンテキストを作成してから、古いコンテキストの一部を削除した場合に起こる可能性があります。古いコンテキストに戻そうとすると、例外が発生します。TRY ブロックの QUIT または RETURN の例外は、CATCH ブロックの例外ハンドラを呼び出しません。

## \$ZTRAP

TRY コマンドと CATCH コマンドは実行レベル内でエラー処理を行います。TRY ブロック内で例外が発生すると、InterSystems IRIS は通常、TRY ブロックの直後に続く例外処理コードの CATCH ブロックを実行します。これは 推奨されるエラー処理動作です。

TRY ブロック内で \$ZTRAP を設定することはできません。

TRY ブロックに入る前に \$ZTRAP が設定されていて、例外が TRY ブロック内で発生する場合、InterSystems IRIS は、\$ZTRAP ではなく CATCH ブロックを取得します。

## \$ETRAP

TRY コマンドと CATCH コマンドは実行レベル内でエラー処理を行います。TRY ブロック内で例外が発生すると、InterSystems IRIS は通常、TRY ブロックの直後に続く例外処理コードの CATCH ブロックを実行します。これは 推奨されるエラー処理動作です。

TRY ブロック内で \$ETRAP を設定することはできません。

TRY ブロックに入る前に \$ETRAP が設定されており、TRY ブロック内で例外が発生した場合、InterSystems IRIS はこの可能性を予想していない限り、CATCH ではなく \$ETRAP を取得します。例外発生時に \$ETRAP と CATCH の両方が存在する場合、InterSystems IRIS は現在の実行レベルに適用されるエラー・コード (CATCH または \$ETRAP) を実行します。\$ETRAP は本来実行レベルとは関連付けられないため、InterSystems IRIS では、他に指定されていない限り現在の実行レベルに関連付けられていると見なします。\$ETRAP を設定して \$ETRAP のレベル・マーカを設定する前に NEW \$ETRAP を実行し、InterSystems IRIS が現在のレベルの例外ハンドラとして \$ETRAP ではなく CATCH を正しく取得するようにする必要があります。そうしないと、システム・エラー (THROW コマンドによりスローされるシステム・エラーを含む) により \$ETRAP 例外ハンドラが取得される可能性があります。

## GOTO および DO

**GOTO** または **DO** コマンドを使用して、TRY ブロック内のラベルで TRY ブロックに入ることができます。後から TRY ブロック内で例外が発生する場合、TRY キーワードで TRY ブロックに入ったかのように、CATCH ブロックの例外ハンドラが取得されます。ただし、コーディングをわかりやすくするために、GOTO や DO を使用して TRY ブロックに入ることは避けてください。

もちろん、TRY ブロック内または CATCH ブロック内から GOTO を発行できます。

GOTO や DO を使用して CATCH ブロックに入ることはお勧めしません。

## TRY ブロック内での DO

TRY 文を使用すると、THROW によって、フレーム・スタックの検索が、適切な CATCH ブロックを検出しようとします。フレーム・スタックが TRY ブロック内での実行を示すと、対応する CATCH ブロックで実行が再開します。ただし、InterSystems IRIS は、CATCH ブロックを実行する前に、現在の TRY ブロック内で任意の“ローカル”呼び出しを削除する必要があります。

TRY ブロックに DO 文が含まれていて、その TRY ブロックに再度入る結果になる場合、次の 2 つのうちのいずれかが発生します。

“ローカル” DO 呼び出し (現在の TRY ブロック内に留まる DO 呼び出し) : 前のフレーム・スタック・エントリが、同じ TRY ブロックにある DO 呼び出しの場合、その DO は、現在の TRY ブロック内の“ローカル”サブルーチン呼び出しと見なされます。この場合、CATCH にはすぐには入らず、フレーム・スタックがポップされ (場合によっては、最近割り当てられた NEW 変数が削除され)、現在の TRY ブロックでの DO 呼び出しで検索が再開します。新しい前のフレーム・スタック・エントリが、現在の TRY ブロック内からの DO でない場合、対応する CATCH ブロックに入ります。ただし、前のフレーム・スタック・エントリが同じ TRY の別の DO の場合、フレーム・スタックは (最近割り当てられた NEW 変数と共に) 再びポップされます。この動作は、前のフレーム・スタック・エントリが DO でなくなるまで継続され、その時点で CATCH ブロックに入ります。

“反復” DO 呼び出し (TRY ブロックから出るが、後の実行でその TRY ブロックに再度入る TRY ブロック内の DO 呼び出し) : CATCH ブロックを検索しているときに、前のフレーム・スタック・エントリが現在の TRY ブロック内の DO だが、その前のスタック・フレームのターゲット・ラベルが現在の TRY ブロック内 (入れ子にされた TRY ブロックを含む) にない場合、フレーム・スタックはポップされず (また、最近割り当てられたローカル変数がポップされず)、CATCH ブロックにすぐに入ります。この CATCH ブロックが別の THROW を実行すると、反復 DO フレームが依然としてフレーム・スタック上にあるため、現在の CATCH ブロックに再度入る可能性があります。

## 例

このセクションの例は、実行時エラー (%Exception.SystemException エラー) を示しています。THROW を発行することで呼び出されたユーザ指定の例外の例は、[THROW](#) および [CATCH](#) コマンドを参照してください。

以下の例では、TRY コード・ブロックが実行されます。ローカル変数 a を設定しようとしています。最初の例では、コードが正常に完了し、CATCH はスキップされます。2 番目の例では、<UNDEFINED> エラーでコードが失敗し、CATCH の例外ハンドラに実行が渡されます。

TRY が成功します。CATCH ブロックがスキップされます。実行が 2 番目の TRY ブロックで続きます。

### ObjectScript

```
TRY {
    WRITE "1st TRY block",!
    SET x="fred"
    WRITE "x is a defined variable",!
    SET a=x
}
CATCH exp
{
    WRITE !,"This is the CATCH exception handler",!
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception",!
        WRITE "Name: ", $ZCVT(exp.Name,"O","HTML"),!
        WRITE "Location: ", exp.Location,!
        WRITE "Code: ", exp.Code,!
        WRITE "Data: ", exp.Data,!
    }
    ELSE { WRITE "not a system exception",!!}
    WRITE "$ZERROR: ", $ZERROR,!
    WRITE "$ECODE: ", $ECODE
    RETURN
}
TRY {
    WRITE !,"2nd TRY block",!
    WRITE "This is where the code falls through",!
    WRITE "$ZERROR: ", $ZERROR,!
    WRITE "$ECODE: ", $ECODE
}
CATCH exp2 {
    WRITE !,"This is the 2nd CATCH exception handler",!
}
```

TRY が失敗します。実行が CATCH ブロックで続きます。CATCH ブロックが RETURN で終了するため、2 番目の TRY ブロックは実行されません。

### ObjectScript

```
TRY {
    WRITE "1st TRY block",!
    KILL x
    WRITE "x is an undefined variable",!
    SET a=x
}
CATCH exp {
    WRITE !,"This is the CATCH exception handler",!
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception",!
        WRITE "Name: ", $ZCVT(exp.Name,"O","HTML"),!
        WRITE "Location: ", exp.Location,!
        WRITE "Code: ", exp.Code,!
        WRITE "Data: ", exp.Data,!
    }
    ELSE { WRITE "not a system exception",!!}
    WRITE "$ZERROR: ", $ZERROR,!
}
```

```
        WRITE "$ECODE: ", $ECODE
    RETURN
}
TRY {
    WRITE !, "2nd TRY block", !
    WRITE "This is where the code falls through", !
    WRITE "$ZERROR: ", $ZERROR, !
    WRITE "$ECODE: ", $ECODE
}
CATCH exp2 {
    WRITE !, "This is the 2nd CATCH exception handler", !
}
```

TRY が終了します。以下の例では、TRY ブロックの実行がエラーではなく QUIT または RETURN のいずれかによって終了されるため、CATCH ブロックは実行されません。RETURN の場合、プログラムの実行が停止します。QUIT の場合、プログラムの実行は 2 番目の TRY ブロックで継続します。

## ObjectScript

```
TRY {
    WRITE "1st TRY block", !
    KILL x
    WRITE "x is an undefined variable", !
    SET decide=$RANDOM(2)
    IF decide=0 { WRITE "issued a QUIT", !
        QUIT }
    IF decide=1 { WRITE "issued a RETURN", !
        RETURN }
    WRITE "This should never display", !
    SET a=x
}
CATCH exp {
    WRITE !, "This is the CATCH exception handler", !
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception", !
        WRITE "Name: ", $ZCVT(exp.Name, "O", "HTML"), !
        WRITE "Location: ", exp.Location, !
        WRITE "Code: ", exp.Code, !
        WRITE "Data: ", exp.Data, !!
    }
    ELSE { WRITE "not a system exception", !! }
    WRITE "$ZERROR: ", $ZERROR, !
    WRITE "$ECODE: ", $ECODE
    RETURN
}
TRY {
    WRITE !, "2nd TRY block", !
    WRITE "This is where the code falls through", !
    WRITE "$ZERROR: ", $ZERROR, !
    WRITE "$ECODE: ", $ECODE
}
CATCH exp2 {
    WRITE !, "This is the 2nd CATCH exception handler", !
}
```

## 関連項目

- [CATCH コマンド](#)
- [THROW コマンド](#)
- [\\$ETRAP 特殊変数](#)
- [\\$ZTRAP 特殊変数](#)
- [TRY-CATCH の使用法](#)

# TSTART (ObjectScript)

トランザクションの開始を表します。

## 構文

```
TSTART:pc
TS:pc
```

## 引数

引数	説明
pc	オプション - 後置条件式

## 概要

TSTART は、トランザクションの開始を表します。以下の TSTART では、それに続く TCOMMIT や TROLLBACK コマンドを有効にするため、データベース・オペレーションがジャーナルされます。

TSTART は、\$TLEVEL 特殊変数の値をインクリメントします。\$TLEVEL 値 0 は、有効なトランザクションがないことを示します。最初の TSTART はトランザクションを開始し、\$TLEVEL を 1 にインクリメントします。後続の TSTART コマンドは、入れ子になったトランザクションを作成する場合があります、\$TLEVEL をさらにインクリメントします。

トランザクション内で発生するすべての処理がロールバック可能なわけではありません。例えば、トランザクション内でのグローバル変数の設定はロールバックできますが、トランザクション内でのローカル変数の設定はロールバックできません。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。

既定では、トランザクション内で発行されたロックは、そのトランザクション内で解除されたとしても、そのトランザクションが終了するまでは保持されます。この既定は、ロックの設定時にオーバーライドできます。詳細は“[LOCK](#)”コマンドを参照してください。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True の場合、TSTART コマンドを実行し、後置条件式が False の場合、TSTART コマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

## 入れ子になったトランザクション

TSTART をトランザクション内で発行すると、入れ子になったトランザクションが開始されます。TSTART を発行すると、\$TLEVEL 値がインクリメントされ、トランザクションの入れ子レベル数が示されます。入れ子になったトランザクションは、TCOMMIT を発行して入れ子のトランザクションをコミットするか、または TROLLBACK 1 を発行して入れ子のトランザクションをロールバックすることによって終了します。入れ子になったトランザクションを終了すると、\$TLEVEL 値が 1 だけデクリメントされます。

- 入れ子になったトランザクションに対して TROLLBACK 1 を発行すると、その入れ子のトランザクションに加えられた変更がロールバックされ、\$TLEVEL がデクリメントされます。TROLLBACK コマンドを発行すると、発行されている TSTART のレベル数に関係なく、トランザクション全体をロールバックできます。
- 入れ子のトランザクションに対する TCOMMIT の発行は、\$TLEVEL をデクリメントしますが、入れ子のトランザクションに対する実際のコミットは延期されます。入れ子になったトランザクションの処理中に行われた変更は、最も外側のトランザクションがコミットされたとき、つまり、TCOMMIT によって \$TLEVEL 値が 0 にデクリメントされたときのみコミットされ、このコミットは取り消すことはできません。



%SYS.Journal.System クラスの GetImageJournalInfo() メソッドを使用して、ジャーナル・ファイル内で TSTART コマンドを検索し、開いているトランザクションを特定できます。TSTART は、\$TLEVEL が 0 の場合は "BT" (Begin Transaction) ジャーナル・ファイル・レコードを、\$TLEVEL が 0 よりも大きい場合は "BTL" (Begin Transaction with Level) ジャーナル・ファイル・レコードをそれぞれ書き込みます。

入れ子構造のトランザクションの最大レベル数は 255 です。この入れ子レベルの制限を超えると、<TRANSACTION LEVEL> エラーになります。

## SQL とトランザクション

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。

## 例

以下の例では、口座間でランダムな金額を送金する、単一レベルのトランザクションを使用しています。送金額が利用可能な残高を超えている場合、プログラムは以下のようにトランザクションをロールバックします。

### ObjectScript

```
SetupBankAccounts
    SET num=12345
    SET ^CHECKING(num,"balance")=500.99
    SET ^SAVINGS(num,"balance")=100.22
    IF $DATA(^NumberOfTransfers)=0 {SET ^NumberOfTransfers=0}
BankTransfer
    WRITE "Before transfer:",!, "Checking=$",^CHECKING(num,"balance")," Savings=$",^SAVINGS(num,"balance"),!

    // Transfer funds from one account to another
    SET transfer=$RANDOM(1000)
    WRITE "transfer amount $",transfer,!
    DO CkToSav(num,transfer)
    IF ok=1 {WRITE "successful transfer",!, "Number of transfers to date=",^NumberOfTransfers,!}
    ELSE {WRITE "*** INSUFFICIENT FUNDS ***",!}
    WRITE "After transfer:",!, "Checking=$",^CHECKING(num,"balance")," Savings=$",^SAVINGS(num,"balance"),!

    RETURN
CkToSav(acct,amt)
    TSTART
    SET ^CHECKING(acct,"balance") = ^CHECKING(acct,"balance") - amt
    SET ^SAVINGS(acct,"balance") = ^SAVINGS(acct,"balance") + amt
    SET ^NumberOfTransfers=^NumberOfTransfers + 1
    IF ^CHECKING(acct,"balance") > 0 {TCOMMIT SET ok=1 QUIT:ok}
    ELSE {TROLLBACK SET ok=0 QUIT:ok}
```

以下の例では、TSTART を使用して、入れ子になったトランザクションを作成します。これらの例は、入れ子になったトランザクションの 3 つのロールバック・シナリオを示しています。

以下では、最も内側のトランザクションをロールバックし、中間のトランザクションをコミットし、最も外側のトランザクションをコミットします。

## ObjectScript

```

KILL ^a,^b,^c
TSTART SET ^a=1 WRITE "tlevel=", $TLEVEL,!
    TSTART SET ^b=2 WRITE "tlevel=", $TLEVEL,!
        TSTART SET ^c=3 WRITE "tlevel=", $TLEVEL,!
            TROLLBACK 1 WRITE "tlevel=", $TLEVEL,!
                TCOMMIT WRITE "tlevel=", $TLEVEL,!
            TCOMMIT WRITE "tlevel=", $TLEVEL,!
        IF $DATA(^a) {WRITE "^a=", ^a,!} ELSE {WRITE "^a is undefined",!}
        IF $DATA(^b) {WRITE "^b=", ^b,!} ELSE {WRITE "^b is undefined",!}
        IF $DATA(^c) {WRITE "^c=", ^c,!} ELSE {WRITE "^c is undefined",!}

```

以下では、最も内側のトランザクションをコミットし、中間のトランザクションをロールバックし、最も外側のトランザクションをコミットします。

## ObjectScript

```

KILL ^a,^b,^c
TSTART SET ^a=1 WRITE "tlevel=", $TLEVEL,!
    TSTART SET ^b=2 WRITE "tlevel=", $TLEVEL,!
        TSTART SET ^c=3 WRITE "tlevel=", $TLEVEL,!
            TCOMMIT WRITE "tlevel=", $TLEVEL,!
        TROLLBACK 1 WRITE "tlevel=", $TLEVEL,!
    TCOMMIT WRITE "tlevel=", $TLEVEL,!
    IF $DATA(^a) {WRITE "^a=", ^a,!} ELSE {WRITE "^a is undefined",!}
    IF $DATA(^b) {WRITE "^b=", ^b,!} ELSE {WRITE "^b is undefined",!}
    IF $DATA(^c) {WRITE "^c=", ^c,!} ELSE {WRITE "^c is undefined",!}

```

以下では、最も内側のトランザクションをコミットし、中間のトランザクションをコミットし、最も外側のトランザクションをロールバックします。

## ObjectScript

```

KILL ^a,^b,^c
TSTART SET ^a=1 WRITE "tlevel=", $TLEVEL,!
    TSTART SET ^b=2 WRITE "tlevel=", $TLEVEL,!
        TSTART SET ^c=3 WRITE "tlevel=", $TLEVEL,!
            TCOMMIT WRITE "tlevel=", $TLEVEL,!
        TCOMMIT WRITE "tlevel=", $TLEVEL,!
    TROLLBACK 1 WRITE "tlevel=", $TLEVEL,!
    IF $DATA(^a) {WRITE "^a=", ^a,!} ELSE {WRITE "^a is undefined",!}
    IF $DATA(^b) {WRITE "^b=", ^b,!} ELSE {WRITE "^b is undefined",!}
    IF $DATA(^c) {WRITE "^c=", ^c,!} ELSE {WRITE "^c is undefined",!}

```

この 3 番目のケースでは、TROLLBACK 1 と TROLLBACK は、どちらも \$TLEVEL を 0 にデクリメントするので、同じ結果になります。

## 関連項目

- ・ [TCOMMIT コマンド](#)
- ・ [TROLLBACK コマンド](#)
- ・ [\\$TLEVEL 特殊変数](#)
- ・ [トランザクション処理での ObjectScript の使用法](#)

## USE (ObjectScript)

現在のデバイスとして、デバイスを構築します。

### 構文

```
USE:pc useargument,...
U:pc useargument,...
```

useargument には、以下を指定できます。

```
device:(parameters):"mnespace"
```

### 引数

引数	説明
pc	オプション — 後置条件式。
device	現在のデバイスとして選択されるデバイスです。デバイス ID、またはデバイス・エイリアスのいずれかで指定されます。デバイス ID は整数 (デバイス番号)、デバイス名、またはシーケンシャル・ファイルのパス名のいずれかです。文字列の場合は引用符で囲まれます。
parameters	オプション — デバイスの特性を設定するのに使用されるパラメータ・リスト。パラメータ・リストは括弧で囲まれ、リスト内のパラメータはコロンで区切られます。パラメータは (パラメータ・リストの固定順で指定された) 位置、またはキーワード (順不同) のいずれかです。位置パラメータとキーワード・パラメータは、混在させることもできます。個別のパラメータとそれぞれの位置とキーワードは、デバイスに依存しています。
mnespace	オプション — 引用符付き文字列として指定された、このデバイスで使用する制御ニーモニックを含むニーモニック空間名です。

### 概要

USEdevice では、指定されたデバイスが現在のデバイスとして設定されます。このコマンドを実行する前に、プロセスでは OPEN コマンドを使用して、デバイスの所有権を設定しておく必要があります。

現在のデバイスは、別の USE コマンドを実行して、他に所有しているデバイスを現在のデバイスとして選択するか、プロセスを終了するまで、そのまま保持されます。

USE コマンドはターミナル・デバイス、スプール・デバイス、TCP バインディング、プロセス間パイプ、名前付きパイプ、ジョブ内通信などのデバイスを現在のデバイスとして設定することができます。USE コマンドは、シーケンシャル・ファイルを開くためにも使用できます。device 引数は、引用符付きの文字列としてファイル・パス名を指定します。

USE コマンドで利用可能な parameters は、デバイスによって異なります。多くの場合、利用できる parameters は OPEN コマンドで利用可能なものと同じです。しかし、OPEN コマンドでしか設定できないデバイス・パラメータもあり、逆に USE コマンドでしか設定できないデバイス・パラメータもあります。

USE コマンドは、コンマで区切られた複数の useargument を指定することができます。しかし、現在のデバイスは一度に 1 つしか持つことができません。複数の useargument を指定すると、最後の useargument で指定されたデバイスが現在のデバイスとなります。USE のこの形式は、複数のデバイスに対して parameters を設定するために使用され、現在のデバイスとして最後に名付けられたデバイスを設定します。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

### device

現在のデバイスとして選択するデバイス。対応する OPEN コマンドで指定したものと同一デバイス ID (または他のデバイス識別子) を指定します。デバイスの指定に関する詳細は、“[OPEN](#)” コマンドを参照してください。

### parameters

現在のデバイスとして使用されるデバイスの動作特性を設定するために使用される、パラメータのリストです。複数のパラメータが含まれる場合、括弧で囲まなければなりません (プログラム記述時には、パラメータが 1 つでも必ず括弧で囲むことをお勧めします。) コロンは、左括弧の前に配置します。括弧内では、コロンを使用して複数のパラメータを区切ります。

デバイス用のパラメータは、位置パラメータ、またはキーワード・パラメータのいずれかを使用して指定されます。また、位置パラメータとキーワード・パラメータを同じパラメータ・リスト内で混在させて使用することもできます。

多くの場合、矛盾したパラメータ、複製パラメータ、または無効なパラメータ値を指定するとエラーが返されます。可能な場合は常に、InterSystems IRIS は不適切なパラメータ値を無視し、適切な既定値を使用します。

利用可能なパラメータは、多くの場合、OPEN コマンドでサポートされているパラメータと同じです。シーケンシャル・ファイル、TCP デバイス、プロセス間通信パイプでは、OPEN コマンドでしか設定できないパラメータもあります。また、シーケンシャル・ファイルでは、USE コマンドでしか設定できないパラメータもあります。USE パラメータは、選択されるデバイスの種類や特定の実装タイプによって異なります。“[入出力の概要](#)”に、USE コマンド・キーワード・パラメータがデバイス・タイプ別にリストされています。

USE パラメータのリストを指定しない場合は、デバイスの既定の OPEN パラメータが使用されます。デバイスの既定パラメータは構成可能です。管理ポータルに進み、[\[システム管理\]](#)、[\[構成\]](#)、[\[デバイス設定\]](#)、[\[デバイス\]](#) の順に選択して、現在のデバイスのリストを表示します。対象のデバイスに対して [\[編集\]](#) をクリックし、[\[オープン・パラメータ :\]](#) オプションを表示します。この値を、OPEN コマンド・パラメータと同じ方法で指定します (括弧も含む)。例えば ("AVL":0:2048) です。

### 位置パラメータ

位置パラメータは、パラメータ・リスト内で固定の順序で指定されなければなりません。位置パラメータは省略することもできます (省略の場合は既定値を使用)、省略された位置パラメータの位置を示すためのコロンは保持する必要があります。末尾のコロンは必須です。余分なコロンは無視されます。個別のパラメータとそれぞれの位置は、デバイスに依存しています。位置パラメータには、値と文字コード文字列の 2 種類があります。

値は整数 (例えばレコード・サイズ)、文字列 (例えばホスト名)、または値に評価される変数や式のいずれかです。

文字コード文字列は、開くデバイスの特性を指定するために個別の文字を使用します。大半のデバイスで、この文字コード文字列は位置パラメータの 1 つとなります。ユーザは文字列内の任意の文字数を順不同で指定することができます。文字コードは、大文字と小文字を区別しません。文字コード文字列は引用符で囲まれます。文字コード文字列でスペースや句読点を使用することはできません (¥ 記号で区切られた名前が続く K と Y (例: K¥name¥) は例外です)。位置パラメータを使用する例は、シーケンシャル・ファイルを開くときに "ANDFW" (既存のファイルへの追加 (A)、ファイルの新規作成 (N)、ファイルの削除 (D)、固定長レコード (F)、書き込みアクセス (W)) という文字コード文字列を指定する場合などです。文字コード文字列パラメータの位置と、それぞれの文字の意味は、デバイスに依存しています。

### キーワード・パラメータ

キーワード・パラメータは、パラメータ・リスト内の任意の順序で指定することができます。パラメータ・リストはキーワード・パラメータだけで構成することも、位置パラメータとキーワード・パラメータを混在させて構成することもできます。(一般的

には、位置パラメータが (適切な順序で) 最初に指定され、その次にキーワード・パラメータが指定されます)。すべてのパラメータ (位置、およびキーワード) は、コロン (:) で区切られなければなりません。キーワード・パラメータのパラメータ・リストの標準的な構文は、以下のとおりです。

```
USE device: (/KEYWORD1=value1:/KEYWORD2=value2:.../KEYWORDn=valuen): "mnespace"
```

個別のパラメータとそれぞれの位置は、デバイスに依存しています。標準規約として、位置パラメータやキーワード・パラメータのいずれかを使用して、同じパラメータと値を指定することができます。文字コード文字列は、/PARAMS キーワードを使用して キーワード・パラメータとして指定することができます。

## mnespace

このデバイスによって使用されるデバイス制御ニーモニックを含む、ニーモニック・スペースの名前です。既定で、InterSystems IRIS はすべてのデバイスとシーケンシャル・ファイル用に %X364 (ANSI X3.64 互換) のニーモニック・スペースを提供しています。既定のニーモニック・スペースは、デバイスのタイプごとに割り当てられます。

管理ポータルに進み、[システム管理]、[構成]、[デバイス設定]、[IO設定] の順に選択します。[ファイル]、[その他]、または [ターミナル] のニーモニック・スペース設定を表示して編集します。

ニーモニック・スペースは、READ コマンドと WRITE コマンドによって使用されるデバイス制御ニーモニックに対するエントリ・ポイントを含むルーチンです。READ コマンドと WRITE コマンドは、/mnemonic(params) 構文を使用して、デバイス制御ニーモニックを呼び出します。このデバイス制御ニーモニックは、カーソルを画面内の指定された位置に移動するなどの処理を実行します。

mnespace 引数を使用して、既定のニーモニック・スペースの割り当てをオーバーライドします。このデバイスで使用されている制御ニーモニック・エントリ・ポイントを含む、ObjectScript ルーチンを指定します。二重引用符で囲むことが必要です。このオプションは、READ コマンドや WRITE コマンドでデバイス制御ニーモニックを使用したい場合にのみ指定します。ニーモニック空間が存在しない場合、InterSystems IRIS は <NOROUTINE> エラーを発行します。ニーモニック空間の詳細は、"[入出力の概要](#)" を参照してください。

## 例

以下の例では、USE コマンドはシーケンシャル・ファイル "STUDENTS" を現在のデバイスとして設定し、ファイルの開始行を基準にして 256 行目から順次読み取りが開始されるファイル・ポイントを設定します。

### ObjectScript

```
USE "STUDENTS":256
```

## デバイスの所有権

デバイスの所有権は OPEN コマンドで設定されます。唯一の例外は主デバイスで、これはユーザがサインオンしたときにプロセスに割り当てられるもので、通常は端末になります。USE コマンドで指定されたデバイスをそのプロセスが所有していない場合、InterSystems IRIS は <NOTOPEN> エラー・メッセージを返します。

## 現在のデバイス

現在のデバイスは、READ コマンドおよび WRITE コマンドによる入出力処理に使用されるデバイスです。READ コマンドは、現在のデバイスから入力を取得し、WRITE コマンドは、現在のデバイスに出力を送ります。

InterSystems IRIS は現在のデバイスの ID を特殊変数 \$IO に保持します。USE 要求が成功した場合、InterSystems IRIS は \$IO に指定デバイスの ID を設定します。%Library.Device クラスの GetType() メソッドは、現在のデバイスのデバイス・タイプを返します。

## 主デバイス

特殊なデバイス番号 0 (ゼロ) は、主デバイスを表します。各プロセスに 1 つの主デバイスが割り当てられています。InterSystems IRIS は主デバイスの ID を特殊変数 \$PRINCIPAL に保持します。主デバイスは、ユーザが InterSystems IRIS を開始したときに自動的に開きます。最初は、主デバイス (\$PRINCIPAL) と現在のデバイス (\$IO) は同じものです。

USE コマンドを実行した後、現在のデバイス (\$IO) は通常、最後に実行した USE コマンドで指定されていたデバイスになっています。

多くのプロセスが同じ主デバイスを持つことができますが、一度に 1 つしか持つことができません。あるプロセスがデバイスの OPEN コマンドの実行に成功した後は、明示的な CLOSE コマンドの実行、プロセス自体の停止、またはユーザによるセッション終了によって、そのプロセスがデバイスを解放するまで、他のプロセスがそのデバイスに対して OPEN コマンドを実行することはできません。

ターミナルから主デバイス以外のデバイスに対して OPEN と USE を発行することはできますが、InterSystems IRIS は > プロンプトに戻るたびに、暗黙に USE 0 を発行します。0 以外のデバイスの使用を継続するには、> プロンプトで入力する行ごとに USE コマンドを実行する必要があります。

以下のいずれかの場合に、自動的に主デバイスが現在のデバイスに設定されます。

- ・ ログオンしたとき。
- ・ USE 0 コマンドを発行したとき。
- ・ エラー・トラップが設定されていない状態でエラーが発生したとき。
- ・ 現在のデバイスを閉じたとき。
- ・ ターミナル・プロンプトに戻ったとき。
- ・ HALT コマンドを発行して InterSystems IRIS を終了したとき。

USE 0 は、主デバイスに対する OPEN コマンドを意味します。他のプロセスがそのデバイスを所有している場合、このプロセスは OPEN コマンドが発生した場合と同じように、暗黙の OPEN で停止します。

USE 0 は、主デバイスに対して OPEN 0 コマンドを実行することを意味しますが、(以前の OPEN コマンドが原因で) プロセスが所有していない他のデバイスに対して USE コマンドを実行すると、<NOTOPEN> エラーが返されます。

**注釈** 大多数の InterSystems IRIS プラットフォームでは、入力主デバイスを閉じることができますが、UNIX® 用の InterSystems IRIS ではできません。したがって、他のジョブの子であるジョブがログイン端末で入出力を実行しようとした場合、そのジョブは、InterSystems IRIS をログオフするまで停止します。このとき、出力が表示されることも、表示されないこともあります。

## UNIX® での NULL デバイスの使用

NULL デバイス (UNIX® の場合は /dev/null) に対して、OPEN コマンドおよび USE コマンドを実行すると、InterSystems IRIS は、その NULL デバイスをダミー・デバイスとして扱います。この後の READ コマンドは、即座に NULL 文字列 ("" ) を返します。また、WRITE コマンドの場合は、直ちに成功を返します。いずれも、実データは、読み取りまたは書き込みされていません。UNIX® ベースのシステムの場合、デバイス /dev/null は UNIX® システム・コール open、write、および read をバイパスします。

JOB コマンドで他のプロセスによって開始されたプロセスには、既定で、主デバイスとして /dev/null が設定されます。

InterSystems IRIS の外部から /dev/null を開く場合、例えば InterSystems IRIS の出力を UNIX® シェルから /dev/null に転送する場合は、他のデバイスに行われるのと同じように、UNIX® システム・コールによって処理が行われます。

## 関連項目

- ・ [OPEN コマンド](#)
- ・ [CLOSE コマンド](#)



- ・ [\\$IO](#) 特殊変数
- ・ [\\$PRINCIPAL](#) 特殊変数
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)
- ・ [TCP クライアント/サーバ通信](#)
- ・ [シーケンシャル・ファイルの入出力](#)
- ・ [スプール・デバイス](#)



## VIEW (ObjectScript)

データベース・ブロックの読み取りや書き込み、メモリ内のデータの変更を行います。

### 構文

```
VIEW:pc viewargument
V:pc viewargument
```

viewargument は以下のいずれかになります。

```
block
offset:mode:length:newvalue
```

### 引数

引数	説明
pc	オプション - 後置条件式
block	整数として指定されるブロック位置。
offset	バイト単位で表される、mode で指定されたメモリ領域内のベース・アドレスからのオフセット。
mode	修正されるデータを計算するために使用される、ベース・アドレスを含むメモリ領域。
length	修正されるデータの長さ。
newvalue	メモリの場所に格納される置き換え値。

### 概要

VIEW コマンドは、データベース・ブロックの読み取りや書き込み、メモリ内の場所への書き込みを行います。VIEW には以下の 2 つの引数形式があります。

- VIEW block は、InterSystems IRIS データベースとメモリ間でデータを移動させます。
- VIEWoffset:mode:length:newvalue は、offset、mode、および length で指定されるメモリ位置に newvalue を配置します。

**\$VIEW** 関数を使用して、メモリ内のデータを調べることができます。

**注釈** VIEW コマンドは使用しないことをお勧めします。どのような環境で使用されても、メモリ構造を破壊する可能性があります。

### VIEW の使用には注意が必要

VIEW コマンドの使用には注意が必要です。これは通常、デバッグや InterSystems IRIS データベース、InterSystems IRIS システム情報の修復に使用されます。VIEW の使用法を誤ると、メモリや InterSystems IRIS データベースは、簡単に破壊されます。

### VIEW の使用の制限

VIEW コマンドは、制限付きのシステム機能です。呼び出されるコードが IRISYS データベース内にあるので、コマンドは保護されます。詳細は、“[特別な機能](#)”を参照してください。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### block

整数として指定されるブロック位置。block が正の整数の場合、VIEW はその番号ブロックを表示バッファに読み取ります。block が負の整数の場合、VIEW は現在の表示バッファのブロックをそのブロック・アドレスに書き込みます。block と offset:mode:length:newvalue 引数は、相互排他的です。

ブロックが既にメモリ・バッファ内にある場合は、バッファ内の現在のコンテンツがコピーされます。

ブロック位置 0 は有効な位置ではありません。VIEW 0 を指定しようとすると、<BLOCKNUMBER> エラーが返されます。

### offset

バイト単位で表される、mode で指定されたメモリ領域内のベース・アドレスからのオフセット。

### mode

修正されるデータを計算するために使用される、ベース・アドレスを含むメモリ領域。可能な値に関する説明は、“[メモリ内のデータの修正](#)” を参照してください。

### length

修正されるデータの長さ。

バイト数を整数 1 ～ 4 または 1 ～ 8 で指定します。文字 C または P を使用して、現在のプラットフォームのアドレス・フィールド (ポインタ) のサイズを指定することもできます。

newvalue が文字列を定義している場合は、1 からカウントされる負の整数でバイト数を指定します。newvalue の長さがこの数を上回る場合、InterSystems IRIS は超えた分の文字を無視します。newvalue の長さがこの数を下回る場合、InterSystems IRIS は与えられた文字を保存し、残り部分のメモリ位置は変更されません。

逆順でバイト値を保管 (最低位バイトを最低アドレスに置く) するためには、文字 O を長さを示す数字に追加し、両方を二重引用符で囲みます。

### newvalue

メモリの場所に格納される置き換え値

## 例

以下の例は、InterSystems IRIS データベースの 6 番目のブロックを表示バッファに読み取ります。

### ObjectScript

```
VIEW 6
```

以下の例は、おそらくデータが修正された後に、表示バッファを InterSystems IRIS データベースの元の 6 番目のブロックに書き込みます。

### ObjectScript

```
VIEW -6
```

以下の例は、文字列 "WXYZ" を表示バッファのオフセット ADDR から 4 バイト目にコピーします。\$VIEW(ADDR, 0, -4) 式の結果は、値 "WXYZ" になります。

### ObjectScript

```
VIEW ADDR:0:-4:"WXYZ"
```

## 表示バッファ

データベース・バッファの読み取り、および書き込みに使用された場合、VIEW コマンドは、表示バッファ (デバイス 63) で使用されます。表示バッファは、特殊なメモリ領域です。VIEW 操作を実行する前に、開いておく必要があります。

表示バッファを (OPEN コマンドを使用して) 開くときは、その表示バッファに関連付けられる InterSystems IRIS データベース (IRIS.DAT) を指定します。VIEW コマンドを使用して、次に InterSystems IRIS データベースから個々のブロックを表示バッファに読み取ることができます。

ブロックを表示バッファに読み取った後、\$VIEW 関数を使用してデータを調べることができます。または、VIEW コマンドを使用して、データを修正できます。データを修正した場合は、もう一度 VIEW コマンドを使用して、修正済みのブロックを InterSystems IRIS データベースに書き込むことができます。

## InterSystems IRIS データベースでのデータの読み取りや書き込み

VIEW を使用して、InterSystems IRIS データベースのデータ・ブロックを読み取ったり書き込んだりする前に、まず、OPEN コマンドを使用して表示バッファを開きます。

1. 表示バッファを開きます。表示バッファは、デバイス番号 63 として指定されています。したがって、コマンドは以下のとおりです。

### ObjectScript

```
OPEN 63:location
```

location には、表示バッファに関連付けられる IRIS.DAT ファイルを含むネームスペースを指定します。location は、実装タイプごとに異なります。OPEN 63 コマンドは、InterSystems IRIS データベースで使用するブロック・サイズと等しいサイズのシステム・メモリの領域を割り当てることによって、表示バッファを作成します。

2. VIEW block 形式を使用して、関連付けられた InterSystems IRIS データベースからブロックを読み取ります。block には、正の整数を指定します。例えば以下ようになります。

### ObjectScript

```
VIEW 4
```

この例では、InterSystems IRIS データベースの 4 番目のブロックを表示バッファに読み取ります。表示バッファのサイズは、InterSystems IRIS データベースで使用するブロック・サイズと等しいので、表示バッファには、一度に 1 ブロックしか含めることができません。後に続くブロックを読み取るときには、毎回新しいブロックが現在のブロックを上書きします。InterSystems IRIS データベースからいずれのブロックを読み取るかを決定するには、ファイルの構造に精通していなければなりません。

3. \$VIEW 関数でブロックのデータを検証するか、VIEW コマンドでデータを修正します。
4. 表示バッファのデータの変更を行った場合は、それを InterSystems IRIS データベースに書き込みます。データを書き込むには、VIEW block 形式を使用しますが、block には、負の整数を指定します。ブロック番号は通常、表示バッファ内の現在のブロック番号と一致しますが、必ずしもその必要はありません。指定されたブロック番号によって、表示バッファ内のブロックで置き換える (上書きする) ファイル内のブロックが識別されます。例えば、VIEW -5 の場合、InterSystems IRIS データベースの 5 番目のブロックが表示バッファの現在のブロックと置き換えられます。
5. CLOSE 63 を使用して、表示バッファを閉じます。

## InterSystems IRIS データベース間でのブロック転送

表示バッファを開いても、InterSystems IRIS は自動的に既存のブロックを消去しません。したがって、以下の手順で、InterSystems IRIS データベースから他の InterSystems IRIS データベースにデータ・ブロックを転送することができます。

1. OPEN 63 を使用して、最初の InterSystems IRIS データベースが含まれるネームスペースを指定します。
2. VIEW を使用して、ファイルから目的のブロックを表示バッファに読み取ります。
3. 必要に応じて、VIEW を使用して表示バッファ内のデータを修正します。
4. OPEN 63 を再び使用して、2 番目の InterSystems IRIS データベースが含まれるネームスペースを指定します。
5. VIEW を使用して、表示バッファから 2 番目の InterSystems IRIS データベースにブロックを書き込みます。
6. CLOSE 63 を使用して表示バッファを閉じます。

## メモリ内のデータの修正

InterSystems IRIS データベースからのデータの読み取りや書き込み以外に、VIEW コマンドは、表示バッファ内、または他のシステム・メモリ領域のメモリ内にあるデータを修正することができます。

データを修正するには、以下の形式を使用します。

VIEWoffset:mode:length:newvalue

4 つの引数すべてが必須です。

メモリ内の指定した場所に新しい値を格納することで、データを修正します。この場所は、mode で示されたベース・アドレスからのバイト・オフセットで指定されます。length 引数で、影響を受けるメモリの大きさを指定できます。

mode で利用可能な値は、以下のテーブルのとおりです。

モード	メモリ管理領域	ベース・アドレス
n>0	プロセス n のアドレス空間。n は、そのプロセスに対する \$JOB の値、プロセス ID (PID)。	0
0	表示バッファ	表示バッファの開始位置
-1	プロセスのパーティション	パーティションの開始位置
-2	システム・テーブル	システム・テーブルの開始位置
-3	プロセスのアドレス空間	0
-6	インターシステムズ用に確保	
-7	整合性の確認ユーティリティによってのみ使用	特殊。“ <a href="#">高可用性ガイド</a> ”を参照してください。

## 関連項目

- [OPEN](#) コマンド
- [CLOSE](#) コマンド
- [\\$VIEW](#) 関数

## WHILE (ObjectScript)

条件が True のときに、コードを実行します。

### 構文

```
WHILE expression,... {
    code
}
```

### 引数

引数	説明
expression	テスト条件。1 つ、または複数のコンマ区切りのテスト条件を指定することができます。それらはすべて、コード・ブロックの実行に対して True でなければなりません。
code	中括弧で囲まれた ObjectScript コマンドのブロックです。

### 概要

WHILE は expression をテストし、expression が TRUE と評価された場合に、開き中括弧と閉じ中括弧の間のコード・ブロック (1 つ以上のコマンド) を実行します。WHILE は、expression が TRUE と評価されている限り、コードのブロックを繰り返し実行します。expression が True でない場合、中括弧内のコード・ブロックは実行されず、閉じ中括弧 (}) に続く次のコマンドが実行されます。

プログラマは WHILE の無限ループを回避するために、注意を払う必要があります。

開き中括弧や閉じ中括弧は、1 行で独立して記述するか、コマンドと同じ行に記述できます。開き中括弧や閉じ中括弧は、列 1 に記述してもかまいませんが、お勧めはできません。推奨されるプログラミング手法として、入れ子になったコード・ブロックの開始と終了を示すために、中括弧はインデントするようにしてください。開き中括弧の前後に空白を入れる必要はありません。閉じ中括弧の前に空白を入れる必要はありません。引数のないコマンドに続く中括弧の場合も同様です。中括弧の空白に関する唯一の要件は、閉じ中括弧とその後のコマンドを、スペース、タブ、または改行で区切る必要があるということです。

中括弧内のコードのブロックは、1 つ以上の ObjectScript コマンドと関数の呼び出しで構成される可能性があります。コード・ブロックは複数行にわたる場合があります。インデント、改行、空白スペースは、コード・ブロック内で許可されています。コード・ブロック内のコマンドとコマンド内の引数は、1 つ以上の空白スペース、または改行で分けられます。

### 引数

#### expression

ブーリアンのテスト条件。単一の式かコンマで区切られた式のリストの形式をとります。expression を TRUE (ゼロ以外の数値) として評価した場合、InterSystems IRIS は WHILE ループを実行します。通常、expression は  $x < 10$  や `"apple"="apple"` などの条件テストですが、ゼロ以外の数値に評価される値はすべて TRUE になります。例えば、7、00.1、“700”、“7dwarves”などは、すべて TRUE と評価されます。ゼロに評価される値はすべて FALSE になります。例えば、0、-0、および非数値文字列は、すべて FALSE と評価されます。

expression リストでは、InterSystems IRIS は左から右の順で、個別の式を評価します。0 (FALSE) に評価される式に遭遇すると、評価を終了します。FALSE に評価される式の右側にある式には、検証やテストが実施されません。

すべての式がゼロ以外の数値 (TRUE) に評価される場合、InterSystems IRIS は WHILE ループ・コード・ブロックを実行します。expression が TRUE に評価されている限り、InterSystems IRIS は各ループの先頭で expression をテストしながら、WHILE ループを繰り返し実行します。いずれかの式が FALSE に評価されると、InterSystems IRIS は、WHILE の閉じ中括弧の後の次のコード行を実行します。

## 例

以下の例では、WHILE ループを指定した回数実行します。ループを実行する前に、expression がテストされます。

### ObjectScript

```
Mainloop
  SET x=1
  WHILE x<10 {
    WRITE !," Looping",x
    SET x=x+1
  }
  WRITE !,"DONE"
  QUIT
```

以下の 2 つの例では、2 つの expression テストを実行します。2 つのテストはコンマで区切られています。両方のテストが True に評価される場合、WHILE ループが実行されます。そのため、これらのプログラムは、リストのすべての項目、またはリスト内の指定のサンプル・サイズの項目を返します。

### ObjectScript

```
SET mylist=$LISTBUILD("a","b","c","d","e")
SET ptr=0,sampcnt=1,sampmax=4
WHILE 1=$LISTNEXT(mylist,ptr,value),sampcnt<sampmax {
  WRITE value," is item ",sampcnt,!
  SET sampcnt=sampcnt+1
}
IF sampcnt<sampmax {WRITE "This is the whole list"}
ELSE {WRITE "This is a ",sampcnt-1," item sample of the list"}
```

### ObjectScript

```
SET mylist=$LISTBUILD("a","b","c","d","e")
SET ptr=0,sampcnt=1,sampmax=10
WHILE 1=$LISTNEXT(mylist,ptr,value),sampcnt<sampmax {
  WRITE value," is item ",sampcnt,!
  SET sampcnt=sampcnt+1
}
IF sampcnt<sampmax {WRITE "This is the whole list"}
ELSE {WRITE "This is a ",sampcnt-1," item sample of the list"}
```

## WHILE と DO WHILE

WHILE コマンドは、ループを実行する前に expression をテストします。DO WHILE コマンドはループを一度実行し、そして expression をテストします。

## WHILE および FOR

FOR または WHILE のいずれかを使用して、同じ操作を実行できます。つまり、イベントによって実行がループを抜けるまでループすることができます。ただし、どちらのループ構造を使用するかにより、コード・モジュールに対してシングル・ステップ (BREAK "S+" または BREAK "L+") デバッグの実行に影響があります。

FOR ループでは、スタックに新しいレベルがプッシュされます。WHILE ループでは、スタック・レベルは変更されません。FOR ループをデバッグする場合、FOR ループ内からスタックをポップすると (BREAK "C" GOTO または QUIT 1 を使用)、FOR コマンド構文の終了直後から、このコマンドでのシングル・ステップ・デバッグを続行できます。WHILE ループをデバッグする場合は、BREAK "C" GOTO または QUIT 1 を使用して発行してもスタックはポップされません。したがって、WHILE コマンドの終了後にシングル・ステップ・デバッグは続行されません。残りのコードはブレークなしで実行されます。

詳細は、“[BREAK](#)” コマンドおよび “[BREAK コマンドによるデバッグ](#)” を参照してください。

## WHILE および CONTINUE

WHILE コマンドのコード・ブロック内で CONTINUE コマンドに出会うと、実行が即座に WHILE コマンドに戻ります。次に WHILE コマンドは expression テスト条件を評価し、その評価を基にしてコード・ブロック・ループを再実行するかどうか



かを決定します。したがって、CONTINUE コマンドの実行にはコード・ブロックの終了中括弧に到達するのとまったく同じ効果があります。

## WHILE、QUIT、RETURN

code ブロック内の **QUIT** コマンドは WHILE ループを終了させ、終わりの中括弧に続くコマンドに制御を渡します。以下はその例です。

### ObjectScript

```
Testloop
  SET x=1
  WHILE x < 10
  {
    WRITE !,"Looping",x
    QUIT:x=5
    SET x=x+1
  }
  WRITE !,"DONE"
```

このプログラムは、Looping1 から Looping5 までを書き込み、その後 DONE を書き込みます。

WHILE コード・ブロックは入れ子にできます。つまり、WHILE コード・ブロックに、別のフロー制御ループ (別の WHILE、あるいは FOR または WHILE コード・ブロック) を含むことができます。内側の入れ子となったループの QUIT では、内側のループが終了して、そのループの外側のループに実行が移ります。詳細は、以下の例を参照してください。

### ObjectScript

```
Nestedloops
  SET x=1,y=1
  WHILE x<6 {
    WRITE "outer loop ",!
    WHILE y<100 {
      WRITE "inner loop "
      WRITE " y=",y,!
      QUIT:y=7
      SET y=y+2
    }
    WRITE "back to outer loop x=",x,!
    SET x=x+1
  }
  WRITE "Done"
```

**RETURN** を使用すると、WHILE ループや入れ子のループ構造内を含む任意の場所からルーチンの実行を終了させることができます。RETURN は常に現在のルーチンを終了し、呼び出し元のルーチンに戻るか、呼び出し元のルーチンがない場合はプログラムを終了します。RETURN は、コード・ブロック内から発行されたかどうかに関係なく、常に同じ動作を行います。

## WHILE と GOTO

コード・ブロック内の GOTO コマンドは、ループ外のラベルへ実行を移動しループを終了します。同じコード・ブロック内のラベルに実行を移動することもあります。この場合、ラベルは入れ子にされたコード・ブロックに存在します。

GOTO コマンドでは、別のコード・ブロック内のラベルに実行を移すことはできません。このような構造を実行することはできませんが、移動先のコード・ブロックのテスト条件が無効になるため、“不正”と見なされます。

以下は、適切な GOTO の形式です。



## ObjectScript

```
mainloop ; GOTO to outside of the code block
  WHILE 1=1 {
    WRITE !,"In an infinite WHILE loop"
    GOTO labell
    WRITE !,"This should not display"
  }
  WRITE !,"This should not display"
labell
  WRITE !,"Went to labell and quit"
```

## ObjectScript

```
mainloop ; GOTO to elsewhere within the same code block
  SET x=1
  WHILE x<3 {
    WRITE !,"In the WHILE loop"
    GOTO labell
    WRITE !,"This should not display"
  }
labell
  WRITE !,"Still in the WHILE loop after GOTO"
  SET x=x+1
  WRITE !,"x=",x
}
WRITE !,"WHILE loop done"
```

## ObjectScript

```
mainloop ; GOTO from an inner to an outer nested code block
  SET x=1,y=1
  WHILE x<6 {
    WRITE !,"Outer loop",!
    SET x=x+1
  }
labell
  WRITE "outer loop iteration ",x-1,!
  WHILE y<4 {
    WRITE !,"  Inner loop iteration ",y,!
    SET y=y+1
    WRITE "    return to "
    GOTO labell
    WRITE "    This should not display",!
  }
  WRITE "Inner loop completed",!
}
WRITE "All done"
```

## ObjectScript

```
mainloop ; GOTO from an outer to an inner nested code block
  SET x=1,y=1
  WHILE x<6 {
    WRITE !,"Outer loop",!
    SET x=x+1
    WRITE "outer loop iteration ",x-1,!
    WRITE "Jumping into the "
    GOTO labell
    WRITE "This should not display",!
    WHILE y<4 {
      WRITE !,"  Inner loop iteration ",y,!
      SET y=y+1
    }
  }
labell
  WRITE "inner loop ",!
}
WRITE "Inner loop completed",!
}
WRITE "All done"
```

以下の形式の GOTO は実行可能ですが、GOTO の移動先ブロックの条件テストが無効になる（無視される）ため、“不正”と見なされます。

## ObjectScript

```
mainloop ; GOTO into a code block
  SET x=1
  WRITE "Jumped into the "
  GOTO label1
  WHILE x>1,x<6 {
    WRITE "Top of WHILE loop x=",x,!
label1
    WRITE "Bottom of WHILE loop x=",x,!
    SET x=x+1
  }
```

## ObjectScript

```
mainloop ; GOTO from a code block into an IF clause block
  SET x=1
  WHILE x<6 {
    WRITE !,"WHILE loop iteration=",x,!
    SET x=x+1
    GOTO label1
    WRITE "This should never display",!
    IF x#2 { WRITE "in the IF clause",!
label1
    WRITE "GOTO entry into the IF clause",!
    WRITE x," is an odd number",!
    }
    ELSE {WRITE "in the ELSE clause",!
          WRITE x," is an even number",! }
    WRITE "Bottom of WHILE loop",!
  }
  WRITE "All done"
```

## 関連項目

- ・ [DO WHILE コマンド](#)
- ・ [FOR コマンド](#)
- ・ [IF コマンド](#)
- ・ [CONTINUE コマンド](#)
- ・ [GOTO コマンド](#)
- ・ [QUIT コマンド](#)
- ・ [RETURN コマンド](#)

## WRITE (ObjectScript)

出力を現在のデバイスに表示します。

### 構文

```
WRITE:pc writeargument,...
W:pc writeargument,...
```

writeargument には、以下を指定できます。

```
expression
f
*integer
*-integer
```

### 引数

引数	説明
pc	オプション - 後置条件式
expression	オプション - 出力デバイスに書き込まれる値。リテラル、変数、オブジェクト・メソッド、およびオブジェクト・プロパティを含めて、数値または引用符付き文字列のいずれかに評価される有効な ObjectScript 式です。
f	オプション - ターゲット・デバイスでの出力位置を定める 1 つまたは複数の形式制御文字。形式制御文字には、!、#、?、および / が含まれます。
*integer	オプション - 出力デバイスに書き込まれる文字を表す整数コード。ASCII では 0 から 255 の整数、Unicode では 0 から 65534 の整数です。評価結果が適切な範囲の整数となる有効な任意の ObjectScript 式となります。アスタリスクは必須です。
*-integer	オプション - デバイス制御操作を指定する負の整数コード。アスタリスクは必須です。

### 概要

WRITE コマンドは、現在の入出力デバイスに指定された出力を表示します (現在の入出力デバイスを設定するには、USE コマンドを使用して \$IO 特殊関数の値を設定します)。WRITE には、以下の 2 つの形式があります。

- ・ [引数なしの WRITE](#)
- ・ [引数付きの WRITE](#)

#### 引数なしの WRITE

引数なしの WRITE は、定義されたすべてのローカル変数の名前と値を一覧表示します。プロセス・プライベート・グローバル、グローバル変数、または特殊変数はリストしません。以下の形式で定義済みのローカル変数を 1 行に 1 個ずつリストします。

```
varname1=value1
varname2=value2
```

引数なしの WRITE は、すべてのタイプのローカル変数値を引用符付き文字列として表示します。キャノニック形式の数とオブジェクト参照は例外です。キャノニック形式の数は引用符に囲まれずに表示されます。[オブジェクト参照 \(OREF\)](#)

は myoref=<OBJECT REFERENCE>[1@%SQL.Statement] のように表示されます。JSON 配列または JSON オブジェクトはオブジェクト参照 (OREF) として表示されます。ビット文字列値およびリスト値は引用符付きの文字列として表示され、データ値はエンコード形式で表示されます。

数値および数値文字列の表示を以下の例に示します。

### ObjectScript

```
SET str="fred"
SET num=+123.40
SET canonstr="456.7"
SET noncanon1="789.0"
SET noncanon2="+999"
WRITE
```

```
canonstr=456.7
noncanon1="789.0"
noncanon2="+999"
num=123.4
str="fred"
```

引数なしの WRITE は、以下の WRITE 出力の例のように、大文字/小文字を区別する文字列照合順にローカル変数を表示します。

```
A="Apple"
B="Banana"
a="apple varieties"
a1="macintosh"
a10="winesap"
a19="northern spy"
a2="golden delicious"
aa="crabapple varieties"
```

引数なしの WRITE は、以下の WRITE 出力の例のように、数値の照合を使用して、添え字ツリー順にローカル変数の添え字を表示します。

```
a(1)="United States"
a(1,1)="Northeastern Region"
a(1,1,1)="Maine"
a(1,1,2)="New Hampshire"
a(1,2)="Southeastern Region"
a(1,2,1)="Florida"
a(2)="Canada"
a(2,1)="Maritime Provinces"
a(10)="Argentina"
```

引数なしの WRITE は、フォームフィード (\$CHAR(12)) やバックスペース (\$CHAR(8)) などの制御文字を実行します。そのため、制御文字を定義するローカル変数は、以下の例に示すように表示されます。

### ObjectScript

```
SET name="fred"
SET number=123
SET bell=$CHAR(7)
SET formfeed=$CHAR(10)
SET backspace=$CHAR(8)
WRITE
```

```
backspace="
bell="
formfeed="
name="fred"
number=123
```

複数のバックスペースは back というローカル変数を指定して以下のように表示されます: バックスペース 1 つ: back="、バックスペース 2 つ: back "、バックスペース 3 つ: bac "、バックスペース 4 つ: ba "k="、バックスペース 5 つ: b "ck="、バックスペース 6 つ: "ack="、バックスペース 7 つ以上: "ack="。

引数なしの WRITE は、同じ行でその後続くコマンドと最低 2 つの空白で区別されなければなりません。この後のコマンドが引数付きの WRITE である場合、その引数付きの WRITE には、適切な改行 f 形式制御引数を指定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。

### ObjectScript

```
SET myvar="fred"
WRITE WRITE           ; note two spaces following argumentless WRITE
WRITE WRITE myvar     ; formatting needed
WRITE WRITE !,myvar   ; formatting provided
```

引数なしの WRITE リストは、**CTRL-C** を発行すると中断され、〈INTERRUPT〉エラーが発生することがあります。

引数なしの WRITE を使用することで、定義済みのローカル変数をすべて表示できます。[\\$ORDER](#) 関数を使用することで、定義済みのローカル変数の限定されたサブセットを返すことができます。

## 引数付きの WRITE

WRITE は、1 つの writeargument またはコンマ区切りの writeargument のリストを取ります。WRITE コマンドは、expression、f、\*integer、および \*-integer の各引数の任意の組み合わせを取ります。

- WRITE expression は、[expression](#) 引数に対応するデータ値を表示します。expression には、変数の名前、リテラル、またはリテラル値に評価される式を指定できます。
- WRITE f は、目的の[出力書式設定](#)を提供します。引数付き形式の WRITE は、各引数値を区切ったり、文字列を示すための自動的な書式設定を提供しません。そのため、expression の値は f 書式設定で区切られない限り、単一の文字列として表示されます。
- WRITE \*integer は、[整数コードによって表される文字](#)を表示します。
- WRITE \*-integer は、[デバイス制御操作](#)を提供します。

WRITE 引数は、コンマで区切られます。例えば以下のようになります。

### ObjectScript

```
WRITE "numbers",1,2,3
WRITE "letters","ABC"
```

以下のように表示されます。

```
numbers123lettersABC
```

WRITE は、出力文字列の末尾に改行を追加しません。WRITE の出力を区切るには、f 引数の書式設定文字 (改行 (!) 文字など) を明示的に指定する必要があります。

### ObjectScript

```
WRITE "numbers ",1,2,3,!
WRITE "letters ", "ABC"
```

以下のように表示されます。

```
numbers 123
letters ABC
```

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。引数なし

の WRITE や引数付きの WRITE に対する後置条件式を指定することができます。詳細は、“[コマンド後置条件式](#)”を参照してください。

## expression

表示する値です。通常は、リテラル (引用符付き文字列、または数値) または変数になります。ただし、expression は、リテラル、変数、算術式、オブジェクト・メソッド、およびオブジェクト・プロパティを含む、任意の有効な ObjectScript 式にすることもできます。式に関する詳細は、“[ObjectScript の使用法](#)”を参照してください。

expression は、任意のタイプの変数にすることができます。これには、[ローカル変数](#)、[プロセス・プライベート・グローバル](#)、[グローバル変数](#)、および[特殊変数](#)も含まれます。変数には添え字を付けることができます。WRITE は指定した添え字ノードの値のみを表示します。

リテラルとして指定するか、変数として指定するかによって、データ値は以下のように表示されます。

- ・ **文字列**は括弧で囲まずに表示されます。一部の非表示文字 (\$CHAR 0、1、2、14、15、28、127) は表示されません。その他の非表示文字 (\$CHAR 3、16 ~ 26) はプレースホルダ文字として表示されます。制御文字 (\$CHAR 7 ~ 13、27) は実行されます。例えば、\$CHAR(8) はバックスペースを実行し、\$CHAR(11) は垂直タブを実行します。
- ・ **数字**はキャノニック形式で表示されます。算術演算は実行されます。
- ・ **拡張グローバル参照**は、グローバルの値として表示されます。グローバル変数が定義されたネームスペースは示されません。存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行し、続けてグローバル名とデータベース・パスを表示します (例:<PROTECT> ^myglobal,c:\intersystems\IRIS\mgr\)
- ・ **ObjectScript リスト**構造化データはエンコード形式で表示されます。
- ・ **InterSystems IRIS ビット文字列**はエンコード形式で表示されます。
- ・ **オブジェクト参照**は OREF 値として表示されます。例えば、##class(%SQL.Statement).%New() は OREF 2@%SQL.Statement と表示されます。JSON ダイナミック・オブジェクトまたは JSON 動的配列は OREF 値として表示されます。OREF の詳細は、“[OREF の基本](#)”を参照してください。
- ・ オブジェクト・メソッドとプロパティは、プロパティの値、またはメソッドによって返された値を表示します。Get メソッドによって返される値は引数の現在の値です。Set メソッドによって返される値は引数の前の値です。添え字を使用して多次元プロパティを指定できます。添え字 (または空の括弧) を使用して非多次元プロパティを指定すると、<OBJECT DISPATCH> エラーが返されます。
- ・ **%Status** は 1 (成功)、または複雑なエンコード失敗ステータスとして表示されます。失敗ステータスの最初の文字は 0 です。

## f

ターゲット・デバイスの出力を配置する形式制御です。形式制御は、コンマで区切らずに形式制御文字の任意の組み合わせを指定することができますが、式と区切るためにコンマを使用する必要があります。以下は、WRITE を端末に対して実行する例です。

## ObjectScript

```
WRITE #!!!?6,"Hello",!,"world!"
```

形式制御は新規の画面の最上部に配置し (#)、次に 3 つの改行を行い (!!!)、最後に 6 列をインデントします (?6)。WRITE は文字列 Hello を表示し、形式制御の改行 (!) を実行して、文字列 world! を表示します。改行は列 1 に配置されることに注意してください。したがってこの例では、Hello はインデントされて表示されますが、world! はインデントされません。

形式制御文字は、引数なしの WRITE では使用できません。

詳細は、“[WRITE で形式制御を使用する](#)”を参照してください。

## \*integer

\*integer 引数により、正の整数コードを使用して、現在のデバイスに文字を書き込めるようになります。これは、アスタリスクと、それに続く任意の有効な ObjectScript 式（評価結果が文字に対応する正の整数になる式）で構成されます。  
 \*integer 引数は、出力可能文字または制御文字に対応することもあります。0 から 255 の整数は、対応する 8 ビットの ASCII 文字に評価されます。256 から 65534 までの範囲にある整数は、対応する 16 ビットの Unicode 文字に評価されます。

以下の例に示すように、\*integer は整数コードを指定するか、整数コードに解決される式を指定できます。以下の例はすべて、“touché” という文字を返します。

### ObjectScript

```
WRITE !,"touch",*233
WRITE !,*67,*97,*99,*104,*233
SET accent=233
WRITE !,"touch",*accent      ; variables are evaluated
WRITE !,"touch",*232+1      ; arithmetic operations are evaluated
WRITE !,"touch",*00233.999  ; fractional numbers are truncated to integers
```

作曲家ドボルザーク (Anton Dvorak) の名前を、適切なチェコ語アクセント記号で表示するために、以下を使用します。

### ObjectScript

```
WRITE "Anton Dvo",*345,*225,"k"
```

式の評価から返された整数は、制御文字に対応することもあります。このような文字は、ターゲット・デバイスに従って解釈されます。  
 \*integer 引数を使用して、画面表示を管理する制御文字（例えば、改ページ \*12）や、ターミナルで通知音を出す \*7 などの特殊文字を挿入できます。

例えば、現在のデバイスがターミナルの場合、整数 0 から 30 は ASCII 制御文字として解釈されます。以下のコマンドは、ASCII コード 7 および 12 をターミナルに送ります。

### ObjectScript

```
WRITE *7 ; Sounds the bell
WRITE *12 ; Form feed (blank line)
```

次に、expression 引数と、改ページ文字を指定する \*integer の組み合わせ例を示します。

### ObjectScript

```
WRITE "stepping",*12,"down",*12,"the",*12,"stairs"
```

## \*integer and \$X, \$Y

整数式は、ターミナルへの書き込み時に \$X および \$Y の特殊変数を変更することはありません。したがって WRITE "a" と WRITE \$CHAR(97) の両方とも、\$X に含まれる列番号値をインクリメントしますが、WRITE \*97 は \$X をインクリメントしません。

\*integer を使用すると、\$X および \$Y の値を変更しないで、バックスペース (ASCII 8)、改行 (ASCII 10) またはその他の制御文字を発行できます。次のターミナルの例では、この整数式の使用法を示します。

バックスペース：

### ObjectScript

```
WRITE $X,"/",$CHAR(8),$X      ; displays: 01
WRITE $X,"/",$*8,$X           ; displays: 02
```

改行：



## ObjectScript

```
WRITE $Y,$CHAR(10),$Y
/* displays: 1
           2 */
WRITE $Y,*10,$Y
/* displays: 4
           4 */
```

詳細は、“\$X”と“\$Y”の特殊変数、および“[ターミナル入出力](#)”を参照してください。

## \*-integer

アスタリスクとそれに続く負の整数（デバイス制御コード）。WRITE は、以下の一般的なデバイス制御コードをサポートします。

コード	デバイス操作
*-1	次の READ で入力バッファをクリアします。
*-2	TCP デバイスまたは名前付きパイプを切断します。“ <a href="#">TCP クライアント/サーバ通信</a> ”および“ <a href="#">ローカル・プロセス間通信</a> ”を参照してください。
*-3	出力バッファをデバイスにフラッシュします。これにより、ディスク上のファイルへの書き込みが強制されます。
*-9	シーケンシャル・ファイルのコンテンツを、現在のファイル・ポインタの位置で切り捨てます。ファイルを切り捨てるには、そのファイルが開いている必要があります（少なくとも“RW”アクセスで OPEN コマンドを使用します）。また、現在のデバイスとして設定されている必要があります（USE コマンドを使用します）。“ <a href="#">シーケンシャル・ファイルの入出力</a> ”を参照してください。
*-10	入力バッファを即座にクリアします。
*-99	圧縮されたストリーム・データを送信します。“ <a href="#">TCP クライアント/サーバ通信</a> ”を参照してください。

## 入力バッファの制御

\*-1 および \*-10 の制御は、ターミナル・デバイスからの入力に使用されます。これら制御は、READ コマンドによって受け入れられていない任意の文字の入力バッファを消去します。\*-1 制御は、次の READ の入力バッファを消去します。\*-10 制御は、入力バッファを直ちに消去します。WRITE \*-1 または WRITE \*-10 の呼び出し時に保留中の CTRL-C 中断が存在する場合、WRITE は、入力バッファの消去前にこの中断を破棄します。

入力バッファは、キーボードから送られた文字を保持します。また、ルーチンが READ コマンドを実行する前に、ユーザが入力した文字も保持します。このためユーザは、プロンプトが画面に表示される前であっても、応答を先行入力できます。READ コマンドがバッファから文字を取得するとき、InterSystems IRIS は、質問と応答を一緒に表示するように、ターミナルにエコー・バックします。ルーチンがエラーを検出すると、\*-1 または \*-10 制御を使用して、これらの先行入力された答えを削除できます。詳細は、“[ターミナル入出力](#)”を参照してください。

[TCP クライアント/サーバ通信](#)での \*-1 の使用法については、“[入出力デバイス・ガイド](#)”を参照してください。

## 出力バッファの制御

\*-3 の制御は、出力バッファからデータをフラッシュするのに使用されます。これにより、物理デバイス上での書き込み操作が強制されます。つまり、まずデータがデバイス・バッファからオペレーティング・システムの入出力バッファにフラッシュされ、その後、オペレーティング・システムで、強制的にその入出力バッファが物理デバイスにフラッシュされます。この制御は一般に、ディスク上のシーケンシャル・ファイルに即座に書き込むことが強制される場合に使用されます。\*-3 は、Windows および UNIX プラットフォームでサポートされています。他のオペレーティング・システム上では、空命令となります。

[TCP クライアント/サーバ通信](#)での \*-3 の使用法については、“[入出力デバイス・ガイド](#)”を参照してください。

## 例

以下の例の WRITE コマンドは、変数 var1 の現在の値を、現在の出力デバイスに送信します。

### ObjectScript

```
SET var1="hello world"
WRITE var1
```

以下の例の両方の WRITE コマンドは、pi に対する Unicode 文字を表示します。最初に \$CHAR 関数を使用し、次に \*integer 引数を使用します。

### ObjectScript

```
WRITE !,$CHAR(960)
WRITE !,*960
```

以下の例では、名前と姓の値を、それぞれに関連するラベル・テキストと共に書き込みます。WRITE コマンドは、同一の行の複数の引数を結合します。これは、その次の例にある 2 つの WRITE コマンドと同じです。! 文字は、改行を行う形式制御です。(! 改行文字は、異なる 2 つの WRITE コマンドによってテキストが出力されるときに必要とされます。)

### ObjectScript

```
SET fname="Bertie"
SET lname="Wooster"
WRITE "First name: ",fname,!,"Last name: ",lname
```

以下と同等です。

### ObjectScript

```
SET fname="Bertie"
SET lname="Wooster"
WRITE "First name: ",fname,!
WRITE "Last name: ",lname
```

以下の例では、現在のデバイスがユーザのターミナルであると仮定しています。READ コマンドで、ユーザに名前と姓を入力するように求め、入力値をそれぞれ変数 fname、および lname に保存します。WRITE コマンドは、ユーザが確認できるように、fname および lname の値を表示します。出力名を区切るために、スペース文字 (" ") を含む文字列が含まれています。

### ObjectScript

```
Test
READ !,"First name: ",fname
READ !,"Last name: ",lname
WRITE !,fname," ",lname
READ !,"Is this correct? (Y or N) ",check#1
IF "Nn"[check {
    GOTO Test
}
```

以下の例は、client(1,n) ノードの現在の値を書き込みます。

## ObjectScript

```
SetElementValues
  SET client(1,1)="Betty Smith"
  SET client(1,2)="123 Primrose Path"
  SET client(1,3)="Johnson City"
  SET client(1,4)="TN"
DisplayElementValues
  SET n=1
  WHILE $DATA(client(1,n)) {
    WRITE client(1,n),!
    SET n=n+1
  }
  RETURN
```

以下の例は、オブジェクト・インスタンス・プロパティの現在の値を書き込みます。

## ObjectScript

```
SET myoref=##class(%SYS.NLS.Format).%New()
WRITE myoref.MonthAbbr
```

**myoref** はオブジェクト参照 (OREF)、**MonthAbbr** はオブジェクト・プロパティ名です。ドット構文は、オブジェクト式で使います。ドットは、オブジェクト参照とオブジェクト・プロパティ名あるいはオブジェクト・メソッド名の間に配置します。

以下の例は、オブジェクト・メソッド `GetFormatItem()` が返す値を書き込みます。

## ObjectScript

```
SET myoref=##class(%SYS.NLS.Format).%New()
WRITE myoref.GetFormatItem("MonthAbbr")
```

以下の例は、オブジェクト・メソッド `SetFormatItem()` が返す値を書き込みます。通常、`Set` メソッドによって返される値は、引数の前の値です。

## ObjectScript

```
SET myoref=##class(%SYS.NLS.Format).%New()
SET oldval=myoref.GetFormatItem("MonthAbbr")
WRITE myoref.SetFormatItem("MonthAbbr"," J F M A M J J A S O N D")
WRITE myoref.GetFormatItem("MonthAbbr")
WRITE myoref.SetFormatItem("MonthAbbr",oldval)
WRITE myoref.GetFormatItem("MonthAbbr")
```

オブジェクトに対する `WRITE` コマンドには、以下のようにカスケード・ドット構文を持つ式を使用できます。

## ObjectScript

```
WRITE patient.Doctor.Hospital.Name
```

この例では、**patient.Doctor** オブジェクト・プロパティが **Name** プロパティを含む **Hospital** オブジェクトを参照します。そのため、このコマンドは指定した患者の医者に関連している病院名を書き込みます。同じカスケード・ドット構文は、オブジェクト・メソッドと一緒に使用できます。

オブジェクトの `WRITE` コマンドは、以下のデータ型プロパティ・メソッドのように、システム・レベル・メソッドで使用できます。

## ObjectScript

```
WRITE patient.AdmitDateIsValid(date)
```

この例で、`AdmitDateIsValid()` プロパティ・メソッドは、現在の **patient** オブジェクトの結果を返します。`AdmitDateIsValid()` は、**AdmitDate** プロパティのデータ型検証のために作成されたブーリアン・メソッドです。そのため、このコマンドは指定した日付が有効な場合は 1 を、無効の場合は 0 を書き込みます。

すべてのオブジェクト式は、オブジェクト参照が参照するスーパークラス、あるいはクラスを宣言することによって指定できるということに注意してください。上述の例は以下のようにも記述できます。

### ObjectScript

```
WRITE ##class(Patient)patient.Doctor.Hospital.Name
```

### ObjectScript

```
WRITE ##class(Patient)patient.AdmitDateIsValid(date)
```

## \$X と \$Y を使用した WRITE

WRITE は、式の評価から生成される文字を一度に 1 つずつ左から右の順に表示します。InterSystems IRIS は \$X 特殊変数と \$Y 特殊変数の現在の出力位置を記録します。\$X は現在の列位置を定義し、\$Y は現在の行位置を定義します。1 文字表示されるたびに、\$X は 1 つインクリメントされます。

以下の例では、WRITE コマンドは 11 字の文字列 Hello world を出力した後に列番号を与えます。

### ObjectScript

```
WRITE "Hello world", " "_$X, " is the column number"
```

表示された文字列と \$X 値 (, " ", \$x) の間に空白スペースを入れると、評価前にその空白スペースで \$X がインクリメントされます。空白スペースを \$X (, " "\_\$x) に連結すると空白スペースは表示されますが、評価前に \$X の値がインクリメントされません。

連結された空白を使用しても、\$X と \$Y の表示は、以下の例のように \$X をインクリメントします。

### ObjectScript

```
WRITE $Y, " "_$X  
WRITE $X, " "_$Y
```

最初の WRITE では、\$X の値は \$Y 値の桁数分インクリメントされます (ユーザの希望どおりでない場合があります)。2 番目の WRITE では、\$X の値は 0 です。

\$X を使用して、WRITE コマンド間に現在の列位置を表示することができます。WRITE コマンド間の列位置を制御するために、? 形式制御文字を使用することができます。? 形式文字は、\$X が列 0 にあるときだけ意味を持ちます。以下の WRITE コマンドでは、? はインデントを行います。

### ObjectScript

```
WRITE ?5, "Hello world", !  
WRITE "Hello", !?5, "world"
```

## WRITE での形式制御の使用法

f 引数を使用すると、以下の任意の形式制御文字を含めることができます。ターミナルに出力する場合は、これらの制御によって、出力データの画面上の表示位置が決まります。形式制御文字は、任意の組み合わせで指定することができます。

### ! 形式制御文字

1 行進み、0 列目に配置されます (\$Y は 1 ずつインクリメントされ、\$X は 0 に設定されます)。実際の制御コード・シーケンスは、デバイスによって異なります。通常は ASCII 13 (RETURN)、または ASCII 13 と ASCII 10 (LINE FEED) です。

InterSystems IRIS は、引数付きの WRITE で暗黙の改行シーケンスを実行しません。ターミナルへの書き込み時に、! 形式制御文字を持つすべての WRITE を開始 (または終了) することをお勧めします。

! 形式制御は、複数指定することができます。例えば、5 行分進めるには `WRITE !!!!!` と記述します。! 形式制御と他の形式制御を組み合わせることもできます。しかし、!`#` または `!#`、および `?5,!` という組み合わせは、許可されてはいませんが、多くの場合は意味がありません (!`#` または `!#` は 1 行進めた後に新規の画面の最上部に配置し、`$Y` は 0 にリセットされます)。(5 でインデントし、1 行進めて、インクリメントを取り消す)。`?5!` の組み合わせは許可されていません。

現在のデバイスが TCP デバイスである場合、!`!` は RETURN および LINE FEED を出力しません。代わりに、バッファに残っている文字をフラッシュし、ネットワークを通じてターゲット・システムへ送信します。

## # 形式制御文字

CR (ASCII 13) 文字および FF (ASCII 12) 文字を純正 ASCII デバイスに送ると同じ効果があります (正確な動作は、オペレーティング・システムのタイプ、デバイス、および記録形式により異なります)。ターミナル上では、# 形式制御文字により現在の画面が消去され、新しい画面の一番上の 0 列目から出力が開始されます (`$Y` と `$X` は 0 にリセットされます)。

# 形式制御と他の形式制御を組み合わせることもできます。しかし、!`#` または `!#`、および `?5,#` という組み合わせは、許可されてはいませんが、多くの場合は意味がありません (!`#` または `!#` は 1 行進めた後に新規の画面の最上部に配置し、`$Y` は 0 にリセットされます)。`?5,#` は、5 でインデントし、新規の画面の最上部に配置し、インクリメントは行わないことを意味しています)。`?5#` の組み合わせは許可されていません。

## ?n 形式制御文字

この形式制御は、疑問符 (?) の後に整数という形式か、整数に評価される式で構成されています。これは、n 番目の列位置で出力を配置し (列 0 からカウント)、`$X` をリセットします。この整数が現在の列位置よりも小さい、または同じ場合 (`n < $X`)、この形式制御は無効です。新規の列位置を設定するときに、`$X` 特殊変数 (現在の列) を参照できます。例えば、`?$X+3` です。

## /mnemonic 形式制御文字

この形式制御は、スラッシュ (/) の後にニーモニック・キーワードという形式か、(オプションで) ニーモニックに渡されるリスト・パラメータで構成されています。

```
/mnemonic(param1,param2,...)
```

InterSystems IRIS は、アクティブなニーモニック・スペースで定義されたエントリ・ポイント名として mnemonic を解釈します。この形式制御は、カーソルを画面に配置するなどのデバイス関数を実行するために使用されます。アクティブなニーモニック空間がない場合は、エラーが返されます。mnemonic は、パラメータ・リストを必要とします (または必要としないこともあります)。

以下のいずれかの方法で、アクティブなニーモニック空間を設定できます。

- 管理ポータルに進み、[システム管理]、[構成]、[デバイス設定]、[IO設定] の順に選択します。ニーモニック・スペース設定を表示して編集します。
- デバイスに対して OPEN コマンド、または USE コマンドを実行するときに `/mnospace` パラメータを含めます。

以下は、ニーモニック・デバイス機能の例です。

ニーモニック	概要
/IC(n)	現在のカーソル位置に n 文字分のスペースを挿入します。同じ行の残り部分は右に移動させます。
/DC(n)	カーソルの右側の n 文字を削除し、その行を削除します。
/EC(n)	カーソルの右側の n 文字を消去し、その代わりに空白を残します。

ニーモニックに関する詳細は、“[入出力デバイス・ガイド](#)” を参照してください。

## 一連の形式制御の指定

InterSystems IRIS では一連の形式制御を指定して、形式制御と式を分割できます。複数の形式制御のシーケンスを指定するとき、形式制御の間にコンマ区切り文字を置く必要はありません (コンマは許可されます)。コンマ区切り文字は、形式制御を式から区切る場合に必要です。

以下の例では、WRITE コマンドは、出力を 2 行進め、READ コマンドへの入力で設定された列位置に、最初の出力文字を配置します。

### ObjectScript

```
READ !,"Enter the number: ",num
SET col=$X
SET ans=num*num*num
WRITE !!,"Its cube is: ",?col,ans
```

出力する列は、READ で入力された文字数によって異なります。

通常、形式制御は、各 WRITE コマンドのリテラル・オペランドとして指定します。変数は実行可能なオペランドとしてではなく文字列として解釈されるため、変数を使用して形式制御を指定することはできません。複数の WRITE コマンドによって使用される形式制御と式のシーケンスを作成する場合は、以下の例に示すように、[#define](#) プリプロセッサ指示文を使用してマクロを定義できます。

### ObjectScript

```
#define WriteMacro "IF YOU ARE SEEING THIS",!,"SOMETHING HAS GONE WRONG",##continue
$SYSTEM.Status.DisplayError($SYSTEM.Status.Error(x),!!
SET x=83
Module1
/* code */
WRITE $$$WriteMacro
Module2
/* code */
WRITE $$$WriteMacro
```

## WRITE で使用するエスケープ・シーケンス

WRITE コマンドは、READ コマンドと同様、エスケープ・シーケンスをサポートしています。エスケープ・シーケンスは通常、形式操作や制御操作に使用されます。エスケープ・シーケンスの解釈は、現在のデバイスの種類によって異なります。

エスケープ・シーケンスを出力するには、以下の形式を使用します。

### ObjectScript

```
WRITE *27,"char"
```

\*27 はエスケープ文字用の ASCII コードで、char は 1 つ、または複数の制御文字で構成されているリテラル文字列です。二重引用符は必須です。

例えば、現在のデバイスが、VT-100 互換のターミナルである場合、以下のコマンドは、現在のカーソル位置から行末までのすべての文字を消去します。

### ObjectScript

```
WRITE *27,"[2J"
```

複数のプラットフォームで実行可能なプログラムがデバイスに依存しないようにするには、プログラムの最初で SET コマンドを使用して、必要なエスケープ・シーケンスを変数に割り当てます。使用するプログラム・コードでは、実際のエスケープ・シーケンスの代わりに変数を参照することができます。他のプラットフォームとプログラムを適合させるには、SET コマンドで定義されたエスケープ・シーケンスに必要な変更を加えるだけです。

## WRITE とその他の Write コマンドとの比較

WRITE と、[ZWRITE](#)、[ZZDUMP](#)、および [ZZWRITE](#) コマンドとの比較は、“[表示 \(書き込み\) コマンド](#)” を参照してください。

### 関連項目

- ・ [USE コマンド](#)
- ・ [READ コマンド](#)
- ・ [ZWRITE コマンド](#)
- ・ [ZZDUMP コマンド](#)
- ・ [ZZWRITE コマンド](#)
- ・ [\\$X 特殊変数](#)
- ・ [\\$Y 特殊変数](#)
- ・ “[ターミナル入出力](#)” と “[プロセス間通信](#)” のエスケープ・シーケンスの記述
- ・ [ターミナル入出力](#)
- ・ [シーケンシャル・ファイルの入出力](#)
- ・ [スプール・デバイス](#)



## XECUTE (ObjectScript)

指定されたコマンドを実行します。

### 構文

```
XECUTE:pc xecutearg,...
X:pc xecutearg,...
```

xecutearg は、以下のいずれかにできます。

```
"cmdline":pc
(" (fparams) cmdline",params):pc
```

### 引数

引数	説明
pc	オプション — <a href="#">後置条件式</a> 。
cmdline	1 つ以上の有効な ObjectScript コマンドで構成される <a href="#">コマンド行</a> に解決される式。cmdline または (fparams) cmdline は引用符付きの文字列として指定する必要があります。
fparams	オプション — 括弧で囲まれたコンマ区切りのリストとして指定される <a href="#">仮パラメータ・リスト</a> 。仮パラメータは cmdline により使用される変数で、その値は params を渡すことによって指定されます。fparams は、引用符付きのコード文字列内の最初の項目です。
params	オプション — コンマ区切りのリストとして指定される <a href="#">パラメータ・リスト</a> 。これらは fparams に渡されるパラメータです。params が指定されている場合、同数以上の fparams を指定する必要があります。

### 概要

XECUTE は、1 つ以上の ObjectScript コマンド行を実行します。各コマンド行は xecutearg で指定されます。複数の xecutearg をコンマで区切って指定できます。これらの xecutearg は、左から右の順で実行され、それぞれの実行はオプションの[後置条件式](#)で管理されます。xecutearg には以下の 2 つの構文形式があります。

- ・ パラメータ渡しなし。この形式は括弧を使用しません。
- ・ パラメータ渡しあり。この形式は括弧が必要です。

XECUTE には、xecutearg の 2 つの形式を任意に組み合わせて含めることができます。

実際、各 xecutearg は、DO コマンドによって呼び出される一行サブルーチンに類似しており、引数の最後に達するか、QUIT コマンドに遭遇すると終了します。InterSystems IRIS が引数を実行した後、制御は xecutearg の直後に返されます。

XECUTE の各呼び出しは、新しいコンテキスト・フレームをプロセスのコール・スタックに配置します。\$STACK 特殊変数は、コール・スタックのコンテキスト・フレームの現在の番号を含みます。

XECUTE コマンドは、実質的に \$XECUTE 関数と同じ処理を行います。ただし、コマンドでは後置条件を使用できるが、関数では使用できないという点が異なります。コマンドは複数の xecutearg を指定できますが、関数が指定できる xecutearg は 1 つのみです。コマンドでは実行を完了するのに QUIT を必要としませんが、関数では実行パスごとに引数付きの QUIT が必要です。

## XECUTE に呼び出されるコマンドの実行時間

XECUTE 内で呼び出されたコードの実行時間は、ルーチンの本体で遭遇する同じコードの実行時間より遅い場合があります。これは、InterSystems IRIS が XECUTE を処理するたびに、XECUTE コマンドで指定されたソース・コード、あるいは参照されたグローバル変数に含まれるソース・コードをコンパイルするためです。

## XECUTE コマンド行の構文チェック

%Library.Routine クラスの CheckSyntax() メソッドを使用すると、xecutearg [コマンド行文字列](#)の構文チェックを実行できます。CheckSyntax() では、実行可能な ObjectScript コード行の前に 1 つ以上のスペースが必要です。インデントされていない行は、CheckSyntax() では、ラベル、またはラベルとそれに続く実行可能なコードとして解析されます。XECUTE では、実行可能なコードをインデントすることができますが、これは必須ではありません。また、ラベル名を指定することはできません。XECUTE も CheckSyntax() もマクロ・プリプロセッサ・コードは解析しません。

## 入れ子にされた XECUTE の呼び出し

ObjectScript は、XECUTE 引数内での XECUTE の使用をサポートします。しかし、入れ子にされた XECUTE は注意して使用しなければなりません。実行時の処理の正確な流れを判断するのが困難な場合があるからです。

## 引数

### pc

オプションの後置条件式。後置条件式が XECUTE コマンド・キーワードに追加されている場合、InterSystems IRIS は後置条件式が True (0 以外の数値に評価される) の場合にのみコマンドを実行します。InterSystems IRIS では、後置条件式が False (0 に評価される) の場合、XECUTE コマンドを実行しません。

後置条件式が xecutearg に追加されると、InterSystems IRIS は後置条件式が True (ゼロ以外の数値に評価される) の場合にのみ、その引数を評価します。後置条件式が False の場合には、InterSystems IRIS はその xecutearg をスキップして、次の xecutearg を評価します (存在する場合)。詳細は、["コマンド後置条件式"](#)を参照してください。

### cmdline

各 cmdline は、1 つ以上の ObjectScript コマンドを含む文字列でなければなりません。場合によっては、コマンドとその次のコマンドの間にスペースを 2 つ挿入する必要があります。cmdline 文字列には、最初にタブ文字、または最後に [Enter] が含まれていてはなりません。cmdline 文字列内で引用符を指定するには、二重引用符を使用します。以下の例は、2 つのコマンドを含む、単一の cmdline を示しています。

### ObjectScript

```
XECUTE "WRITE "hello " ",! WRITE "world" ",!"
```

cmdline は文字列なので、複数のコード行に分割することはできません。単一の cmdline 引数は、以下のように、[連結演算子](#)で結合された個別の文字列に分割できます。

### ObjectScript

```
XECUTE "WRITE "hello " ",!" _
      " WRITE "world" ",!"
```

単一の cmdline 引数は、以下のように、個別の複数のコンマ区切り cmdline 引数に分割できます。

### ObjectScript

```
XECUTE "WRITE "hello " ",!",
      "WRITE "world" ",!"
```

cmdline の最大長は、次の考慮事項に左右されます。InterSystems IRIS は、ソース cmdline 文字列とその生成オブジェクト・コードの両方を単一の文字列として格納します。結果として生成されたこの文字列は、InterSystems IRIS の [文字列の最大長](#) を超えることはできません。

`/* text */` 形式のコメントを、cmdline 内、連結された cmdline 文字列の間、またはコンマ区切りの cmdline 引数の間に埋め込むことができます。

### ObjectScript

```
XECUTE "SET x="hello " /* 1st val */ SET y="world" /* 2nd val */ "_
      " WRITE x,! /* part of 1st cmdline */ ",
      "WRITE y,! /* 2nd cmdline */ "
```

cmdline は、NULL 文字列 ("") に評価される場合があります。この場合、InterSystems IRIS は何も動作せず、次の xecutearg (存在する場合) で実行を継続します。

パラメータを渡す場合、fparams [仮パラメータ・リスト](#) は cmdline コマンドの前に置き、両方の要素を同じ引用符で囲む必要があります。cmdline と fparams を 1 つ以上のスペースで区切ることをお勧めしますが、スペースは必須ではありません。

### ObjectScript

```
SET x=1
XECUTE ("(in,out) SET out=in+3", x, .y)
WRITE y
QUIT
```

既定では、cmdline で使用するすべてのローカル変数は、パブリック変数です。変数を設定するコマンドを中括弧で囲むことにより、プライベート変数としてコマンド行内で変数を指定することができます。次に例を示します。

### ObjectScript

```
SET x=1
XECUTE ("(in,out) { SET out=in+3 }", x, .y)
WRITE y
QUIT
```

この特定の変数のプライベート変数の指定は、fparams [仮パラメータ・リスト](#) のすぐ後に角括弧で囲んでパブリック変数リストを指定することによって上書きできます。以下の例では、変数 x を含むパブリック変数リストを指定しています。

### ObjectScript

```
SET x=1
XECUTE ("(in,out) [x] { SET out=in+3 }", x, .y)
WRITE y
QUIT
```

## fparams

コンマで区切り、括弧で囲んだ仮パラメータのリスト。仮パラメータ名は、有効な識別子である必要があります。これらの仮パラメータは、別のコンテキストで実行されるため、その xecutearg 内でのみ一意である必要があります。XECUTE を発行したプログラム内、または別の xecutearg 内の同じ名前のローカル変数には影響を与えません。cmdline 内の任意またはすべての fparams を使用する必要はありません。ただし、fparams の数は、指定した params の数と同じか、それ以上である必要があります。そうでない場合は、<PARAMETER> エラーが生成されます。

## params

呼び出し元のプログラムから fparams に渡される、コンマ区切りリストで指定された実パラメータ。params は呼び出し元プログラム内で定義された変数であることが必要です。

前にドットを付けることで、参照によってパラメータを渡すことができます。これは、cmdline から値を渡すのに便利です。例は以下のようになります。詳細は、["参照渡し"](#) を参照してください。

## 例

以下の例では、グローバルを設定するコマンドにパラメータを渡します。2 つのコマンド行が用意されています。それぞれの実行は、[後置条件設定](#)に依存します。

### ObjectScript

```
SET bad=0,good=1
SET val="paid in full"
XECUTE ("(pay) SET ^acct1=pay",val):bad,("(pay) SET ^acct2=pay",val):good
```

ここでは、bad 後置条件の値により、最初の xecutearg はスキップされます。2 番目の xecutearg は、パラメータとして渡される val で実行され、コマンド行で使用される pay 仮パラメータに値を提供します。

以下の例では、参照渡し (.y) を使用して、cmdline から呼び出し元のコンテキストにローカル変数値を渡します。

### ObjectScript

```
CubeIt
SET x=5
XECUTE ("(in,out) SET out=in*in*in",x,.y)
WRITE !,x," cubed is ",y
```

以下の例では、XECUTE コマンドは、ローカル変数 x と y を参照します。x と y にはそれぞれ XECUTE が実行する 3 つの個別の ObjectScript コマンドから構成される文字列リテラルが含まれています。

### ObjectScript

```
SET x="SET id=ans QUIT:ans="" DO Idcheck"
SET y="SET acct=num QUIT:acct="" DO Actcheck"
XECUTE x,y
```

以下の XECUTE コマンドの例では、\$SELECT 文を使用しています。

### ObjectScript

```
XECUTE "SET A=$SELECT(A>100:B,1:D)"
```

以下の例では、A の値であるサブルーチンを実行しています。

### ObjectScript

```
SET A="WRITE ! FOR I=1:1:5 { WRITE ?I*5,I+1 }"
XECUTE A
```

## 一般化された演算の実装

XECUTE の代表的な使用法は、アプリケーション内で一般化された演算を実装することです。例えば、ユーザが任意の 2 つの数、または変数で数学演算を実行できるインライン数学カルキュレータを実装するとします。アプリケーションの任意の点から、このカルキュレータを利用可能にするには、特定のファンクション・キー (例えば F1) を使用して、カルキュレータ・サブルーチンをトリガします。

このようなカルキュレータを実装するコードの簡略化バージョンは、以下のようになります。

## ObjectScript

```
Start SET ops=$CHAR(27,21)
      READ !,"Total amount (or F1 for Calculator): ",amt
      IF $ZB=ops { DO Calc
                  ; . . .
                }
Calc  READ !,"Calculator"
      READ !,"Math operation on two numbers and/or variables."
      READ !,"First number or variable name: ",inpl
      READ !,"Mathematical operator (+,-,*,/): ",op
      READ !,"Second number or variable name: ",inp2
      SET doit="SET ans="_inpl_op_inp2
      EXECUTE doit
      WRITE !,"Answer (ans) is: ",ans
      READ !,"Repeat? (Y or N) ",inp
      IF (inp="Y")!(inp="y") { GOTO Calc+2 }
      QUIT
```

上記が実行されると、Calc ルーチンは、ユーザ入力の番号、変数、または希望の演算を受け入れ、これらを適切な SET コマンドを定義する文字列リテラルとして変数 doit に保存します。EXECUTE コマンドは、doit を参照して、そこに含まれるコマンド文字列を実行します。このコード・シーケンスは、アプリケーション内の任意の数のポイントから呼び出すことができ、ユーザは毎回異なる入力を行うことができます。EXECUTE は、提供された入力を使用して、毎回 SET コマンドを実行します。

## EXECUTE とオブジェクト

以下のように、EXECUTE を使用して、オブジェクト・メソッドとプロパティを呼び出し、戻り値を実行できます。

### ObjectScript

```
EXECUTE patient.Name
EXECUTE "WRITE patient.Name"
```

## EXECUTE と FOR

EXECUTE 引数に FOR コマンドが含まれている場合、FOR の範囲は、引数の剰余です。EXECUTE 引数の最も外側の FOR が終了すると、EXECUTE 引数も終了します。

## EXECUTE と DO

EXECUTE コマンドに DO コマンドが含まれていると、InterSystems IRIS は DO 引数で指定されているルーチンを実行します。Caché は QUIT に出会うと、DO コマンドのすぐ後に制御を戻します。

例えば以下のコマンドでは、InterSystems IRIS はルーチン ROUT を実行し、DO 引数のすぐ後に戻って文字列 "DONE" を書き込みます。

### ObjectScript

```
EXECUTE "DO ^ROUT WRITE !,""DONE"""
```

## EXECUTE と GOTO

EXECUTE 引数に GOTO コマンドが含まれていると、InterSystems IRIS は GOTO 引数で指定されているポイントに制御を移します。Caché は、QUIT に遭遇すると、転送を実行する GOTO 引数のすぐ後には戻りません。その代わりに、InterSystems IRIS はその GOTO を含む EXECUTE 引数のすぐ後に制御を戻します。

以下の例では、InterSystems IRIS はルーチン ROUT に制御を移し、EXECUTE 引数のすぐ後に制御を戻して、文字列 "FINISH" を書き込みます。文字列 "DONE" は書き込まれません。

### ObjectScript

```
EXECUTE "GOTO ^ROUT WRITE !,""DONE"" WRITE !,"FINISH"
```

## XECUTE と QUIT

各 XECUTE 引数の終わりには、暗黙の QUIT があります。

## \$TEXT での XECUTE

cmdline 内に \$TEXT 関数を含めることで、XECUTE を含むルーチン内にコード行を指定します。例えば、以下のプログラムでは、\$TEXT 関数が行を検索して実行します。

### ObjectScript

```
A
  SET H="WRITE !!,$PIECE($TEXT(HELP+1),",",",3)"
  XECUTE H
  QUIT
HELP
;; ENTER A NUMBER FROM 1 TO 5
```

ルーチン A を実行すると、“ENTER A NUMBER FROM 1 TO 5” が抽出され、書き込まれます。

## XECUTE と ZINSERT

XECUTE コマンドは、ルーチンから実行可能なコードの単独の行を定義し、挿入するために使用されます。ターミナルで ZINSERT コマンドを使用して、実行可能なコードの単独の行を行位置ごとに定義し、現在のルーチンに挿入することができます。ターミナルで ZREMOVE コマンドを使用して、現在のルーチンから実行可能なコードの 1 行、または複数の行を行位置ごとに削除することができます。

XECUTE コマンドは、新規のラベルの定義に使用することはできません。したがって、XECUTE はコード行の最初のコマンドの前に、最初の空白スペースを必要としません。ZINSERT コマンドは、新規のラベルの定義に使用できます。したがって、ZINSERT はコマンド行の最初のコマンドの前に、最初の空白スペース (または新規ラベルの名前) を必要とします。

## 関連項目

- [DO コマンド](#)
- [GOTO コマンド](#)
- [QUIT コマンド](#)
- [ZINSERT コマンド](#)
- [\\$TEXT 関数](#)
- [\\$XECUTE 関数](#)
- [\\$STACK 特殊変数](#)

## ZKILL (ObjectScript)

ノードの下位を保持しながら、そのノードを削除します。

### 構文

```
ZKILL:pc array-node,...
ZK:pc array-node,...
```

### 引数

引数	説明
pc	オプション - 後置条件式
array-node	配列ノードであるローカル変数、プロセス・プライベート・グローバル、またはグローバル、またはそれらのコンマ区切りのリスト

### 概要

ZKILL コマンドは、指定された array-node の値を削除しますが、そのノードの下位は削除しません。対照的に KILL コマンドは、指定された配列ノードの値と、そのノードの下位を削除します。配列ノードは、ローカル変数、プロセス・プライベート・グローバル、またはグローバル変数のいずれかです。

既定では、この後、この削除された array-node を参照すると〈UNDEFINED〉エラーが生成されます。%SYSTEM.Process.Undefined() メソッドを設定することで、未定義の添え字付き変数を参照する際に〈UNDEFINED〉エラーを生成しないように InterSystems IRIS の動作を変更できます。

### 引数

#### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

#### array-node

ローカル変数、[プロセス・プライベート・グローバル](#)、またはグローバル配列ノード。単一の式、またはコンマ区切りの式のリストを指定できます。添え字とノードに関する詳細は、“[グローバルについての正式な規則](#)” を参照してください。

[構造化システム変数 \(SSVN\)](#) (^\$GLOBAL など) で ZKILL を使用しようとすると、〈COMMAND〉エラーとなります。

### 例

以下の例では、ZKILL コマンドは a(1) を削除しますが、a(1,1) は削除しません。

#### ObjectScript

```
SET a(1)=1,a(1,1)=11
SET x=a(1)
SET y=a(1,1)
ZKILL a(1)
SET z=a(1,1)
WRITE "x=",x," y=",y," z=",z
```

上記は、x=1 y=11 z=11 を返しますが、a を発行します。



## ObjectScript

```
WRITE a(1)
```

これは、〈UNDEFINED〉 エラーを生成します。

## 関連項目

- ・ [KILL コマンド](#)

## ZNSPACE (ObjectScript)

現在のネームスペースを設定します。

### 構文

```
ZNSPACE:pc nspace
ZN:pc nspace
```

### 引数

引数	説明
pc	オプション - 後置条件式
nspace	既存のネームスペースの名前に評価する文字列式

### 説明

ZNSPACE nspace は、現在のネームスペースを nspace 値に変更します。nspace は、明示的なネームスペースでも[暗黙のネームスペース](#)でもかまいません。

- ・ ネームスペースを変更するお勧めの方法は、ターミナル・コマンド・プロンプトから、ZNSPACE を使用することです。
- ・ コード・ルーチン内で、現在のネームスペースを変更するお勧めの方法は、NEW \$NAMESPACE に続けて SET \$NAMESPACE=namespace を使用することです。詳細は、"[\\$NAMESPACE](#)" 特殊変数を参照してください。

ZNSPACE の使用の際、以下のメソッドでユーザの支援ができます。

- ・ 現在のネームスペースの名前を返すには、以下のように \$NAMESPACE 特殊変数または \$ZNSPACE 特殊変数を返すか、または %SYSTEM.SYS クラスの NameSpace() メソッドを呼び出します。

#### ObjectScript

```
WRITE $SYSTEM.SYS.NameSpace()
```

- ・ 現在のプロセスで利用できるネームスペース (明示的および暗黙) をすべてリストするには、以下のように %SYS.Namespace クラスの ListAll() メソッドを呼び出します。

#### ObjectScript

```
DO ##class(%SYS.Namespace).ListAll(.result)
ZWRITE result
```

ListAll() が暗黙のネームスペースをリストする場合は、キャレット (^) 区切り文字を使用してシステム名を区切ります。すべてのローカル・ネームスペースと (オプションで) リモートでマッピングされたネームスペースをすべてリストするには、以下のように %SYS.Namespace クラスの List クエリを呼び出します。

#### ObjectScript

```
SET ListRemote=1
SET stmt=##class(%SQL.Statement).%New()
SET status=stmt.%PrepareClassQuery("%SYS.Namespace", "List")
IF status'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(status) QUIT}
SET rset= stmt.%Execute(ListRemote)
DO rset.%Display()
```

このクエリは、各ネームスペースの名前、そのステータス (利用可否)、およびリモート・システムにマップされているかどうかを返します。

これらのリストはどちらも、ユーザがアクセス特権を持っていないものも含め、すべてのネームスペースをリストします。

- ・ ネームスペースが定義されているかどうかをテストするには、以下のように `%SYS.Namespace` クラスの `Exists()` メソッドを使用します。

### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

これらのメソッドは、“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。

UNIX® システムでは、システム全体の既定のネームスペースはシステム構成オプションとして設定されます。Windows システムでは、コマンド行スタートアップ・オプションを使用して設定されます。

ネームスペースの名前付け規約およびネームスペース名変換については、“[ネームスペース](#)”を参照してください。ネームスペースの使用については、“[ネームスペースとデータベース](#)”を参照してください。ネームスペースの作成および変更の詳細は、“[ネームスペースの構成](#)”を参照してください。

## 現在のネームスペースの変更

ZNSPACE コマンド、%CD ユーティリティ (DO ^%CD) を使用するか、特殊変数 \$NAMESPACE または \$ZNSPACE を設定することで、現在のネームスペースを変更することができます。ターミナル・プロンプトから、より拡張されたエラー・チェック機能を備えた、ZNSPACE、または %CD を使用することをお勧めします。

現在のネームスペースを一時的に変更する場合、処理を実行してから前のネームスペースに戻し、NEW \$NAMESPACE に続けて SET \$NAMESPACE を使用します。NEW \$NAMESPACE および SET \$NAMESPACE を使用することによって、ルーチンが終わったときまたは予期しないエラーが発生したときに前のネームスペースに自動的に戻るネームスペース・コンテキストを確立します。

## 暗黙のネームスペース

暗黙のネームスペースでは、システム名とディレクトリ・パスでネームスペースを指定します。これには、以下の 3 つの形式があります。

- ・ `"^^."` は、現在のネームスペースを指定します。これを使用すると、ネームスペースのプロンプトを明示的なネームスペースから対応する暗黙のネームスペースに変更できます。
- ・ `"^^dir"` は、現在のシステム上にあるネームスペース・ディレクトリ・パス `dir` を指定します。
- ・ `"^system^dir"` は、指定したリモート・システム上にあるネームスペース・ディレクトリ・パス `dir` を指定します。

`dir` にはディレクトリ・パスを指定します。詳細は、以下の例を参照してください。

Windows の例：

### ObjectScript

```
ZNSPACE "^^c:\InterSystems\IRIS\mgr\user\"
WRITE $NAMESPACE
```

Linux の例：

### ObjectScript

```
ZNSPACE "^RemoteLinuxSystem^/usr/IRIS/mgr/user/"
WRITE $NAMESPACE
```

現在のネームスペースのフル・パス名を返すには、以下の例に示すように、`NormalizeDirectory()` メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%Library.File).NormalizeDirectory(" ")
```

## 引数

## pc

オプション - オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

## namespace

新しいネームスペースの名前に評価される有効な文字列式。namespace は、明示的なネームスペースでも[暗黙のネームスペース](#)でもかまいません。

ネームスペース名は、大文字と小文字を区別しません。InterSystems IRIS は常に、明示的なネームスペース名をすべて大文字、暗黙のネームスペース名をすべて小文字で表示します。

namespace が存在しない場合、システムは <NAMESPACE> エラーを生成します。ネームスペースへのアクセス特権を持っていない場合、システムは <PROTECT> エラーを生成した後に、データベース・パスを表示します。例えば、%Developer ロールには %SYS ネームスペースへのアクセス特権がありません。このロールを持っていて、このネームスペースにアクセスしようすると、InterSystems IRIS は <PROTECT> \*c:\intersystems\iris\mgr\ というエラーを返します (Windows システムの場合)。

## 例

以下の例では、“accounting” と呼ばれるネームスペースが既に存在すると想定しています。そうでない場合は、<NAMESPACE> エラーが返されます。

ターミナルで、次のように入力します。

```
USER>ZNSPACE "Accounting"
ACCOUNTING>
```

例に示されているように、ターミナル・プロンプトは既定で現在のネームスペース名を表示します。ネームスペース名は常に、大文字で表示されます。

以下の例はネームスペースの存在をテストし、次に ZNSPACE を使用して現在のネームスペースを設定し、Terminal-Prompt() メソッドを使用してターミナル・プロンプトを、指定されたネームスペース、または USER に設定します。

## ObjectScript

```
WRITE !,"Current namespace is ", $NAMESPACE
SET ns="ACCOUNTING"
IF 1=##class(%SYS.Namespace).Exists(ns) {
    WRITE !,"Changing namespace to: ",ns
    ZNSPACE ns
    DO ##class(%SYSTEM.Process).TerminalPrompt(2)
    WRITE !,"and ", $NAMESPACE, " will display at the prompt"
}
ELSE {
    WRITE !,"Namespace ",ns," does not exist"
    SET ns="USER"
    WRITE !,"Changing namespace to: ",ns
    ZNSPACE ns
    DO ##class(%SYSTEM.Process).TerminalPrompt(2)
    WRITE !,"and ", $NAMESPACE, " will display at the prompt"
}
```

## 既定のディレクトリでのネームスペース

選択したネームスペースに、リモート・マシンの既定のディレクトリが存在する場合、ZNSPACE は、ユーザのプロセスの現在のディレクトリをそのネームスペースのディレクトリに変更しません。したがって、ユーザの現在のネームスペースは、選択したネームスペースとなりますが、現在のディレクトリは、ZNSPACE コマンドを発行する前のディレクトリのままです。

## 暗黙のネームスペースのマッピング

ZNSPACE は、[暗黙のネームスペース](#)からの追加の既定マッピングを作成します。これらのマッピングは、通常の（明示的な）ネームスペースに対するものと同じです。これらにより、プロセスは、IRISSYS データベースと IRISLIB データベース (IRIS¥mgr¥ ディレクトリと IRIS¥mgr¥irislib ディレクトリ) に物理的に置かれている % ルーチンと % グローバルを見つけ、実行できます。

\$NAMESPACE 特殊変数または \$ZNSPACE 特殊変数を設定すること、または暗黙的なネームスペースを使用して %CD ルーチンを実行することは、ZNSPACE コマンドを発行することと同じです。

### % ルーチン・マッピング

プロセスが ZNSPACE コマンドを使用してネームスペースを切り替える場合、通常はシステム・ルーチンのパス・マッピングがリセットされます。これは、通常の（明示的な）ネームスペースと暗黙的なネームスペースの両方に当てはまります。唯一の例外は、プロセスが暗黙的なネームスペースから暗黙的なネームスペースに切り替えられるときで、その場合、既存のマッピングは保存されます。暗黙的なネームスペースの詳細は、[拡張グローバル参照](#) を参照してください。

システム・ルーチンのこの再マッピングは、%SYSTEM.Process クラスの SysRoutinePath() メソッドを使用してオーバーライドできます。これは、既存のシステム・ルーチンのオーバーライドに使用できます。一般的に、これは、% ルーチンのデバッグ時に追加のマッピングを作成するために使用します。プロセスは、IRISSYS データベースに対する Write 権限を持っている必要があります。このメソッドを使用する際には、特に注意が必要です。

**注意**            インターシステムズが提供したシステム・ルーチンのマッピングは、変更しないことを強くお勧めします。変更すると、インターシステムズが提供する現在または今後のライブラリ・ルーチンやメソッドが破壊される可能性があります。

### % グローバル・マッピング

ユーザが初めて ZNSPACE (またはそれに相当するコマンド) を使用して[暗黙のネームスペース](#)に移動すると、システムは次のようにその暗黙のネームスペースのマッピングを作成します。InterSystems IRIS はまず、その暗黙のネームスペースで既存の % グローバルにマップします。そして、他のすべての % グローバルを IRISSYS にマップします。

暗黙的なネームスペースに対してこのマッピングが作成されると、マッピングは共有メモリに保存されます。つまり、それ以降、その暗黙的なネームスペースにユーザが移動するときは、この既存のマッピングが使用されることになります。

暗黙的なネームスペースのグローバル・マッピングを更新するには、この共有メモリ・ストレージをクリアする必要があります。システムの再起動は、共有メモリをクリアするための 1 つの方法です。

## ターミナル・プロンプト

既定で、ターミナル・プロンプトは現在のネームスペース名を表示します。この既定は構成可能です。

管理ポータルに進み、[システム管理]、[構成]、[追加の設定]、[開始] の順に選択します。[TerminalPrompt] の現在の設定を表示して編集します。Telnet ウィンドウのプロンプトで設定することもできます。

現在のプロセスに対するこの動作を設定するには、%SYSTEM.Process クラスの TerminalPrompt() メソッドを使用します。システム全体の既定の動作は、Config.Startup クラスの TerminalPrompt プロパティで設定できます。

ターミナル・プロンプトは、現在のネームスペースを明示的なネームスペースまたは[暗黙のネームスペース](#)として表すことができます。暗黙のネームスペース・パスが 27 文字を超える場合、プロンプトは切り捨てられ、省略記号と、それに続いて暗黙のネームスペース・パスの最後の 24 文字が表示されます。例:...ersystems\iris\mgr\user\>

## \$NAME 関数と \$QUERY 関数

\$NAME と \$QUERY 関数は、ネームスペース名を含むグローバル変数の拡張グローバル参照形式を返します。ユーザは、これらの関数がグローバル変数名の一部としてネームスペース名を返すか否かを制御することができます。`%SYSTEM.Process` クラスの `RefInKind()` メソッドを使用すると、現在のプロセスに対するこの拡張グローバル参照のスイッチを設定できます。システム全体の既定の動作は、`Config.Miscellaneous` クラスの `RefInKind` プロパティで設定できます。拡張グローバル参照の詳細は、“[拡張グローバル参照](#)”を参照してください。

## アプリケーション・コード内でのネームスペースの変更

オブジェクトと SQL コードは 1 つのネームスペースで動作していると見なすため、オブジェクト・インスタンスや SQL カーソルを開いたままネームスペースを変更すると、コードが正常に実行されない場合があります。通常は、さまざまなオブジェクト、SQL、および CSP サーバは、アプリケーション・コードを正しいネームスペースで実行するように自動的に確認するため、明示的にネームスペースを変更する必要はありません。

また、ネームスペースの変更は、他のコマンドに比べ高い処理能力を要するので、アプリケーション・コードはできるだけこれを回避します。

## 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$NAMESPACE](#) 特殊変数
- ・ [\\$ZNSPACE](#) 特殊変数
- ・ [ネームスペースの構成](#)

# ZTRAP (ObjectScript)

エラーを強制し、指定されたエラー・コードを表示します。

## 構文

```
ZTRAP:pc ztraparg
```

```
ZTRAP:pc $ZERROR
ZTRAP:pc $ZE
```

## 引数

引数	説明
pc	オプション - 後置条件式。
ztraparg	オプション - エラー・コードの文字列。エラー・コードの文字列は、文字列リテラル、または文字列に評価される式として指定しますが、文字列の最初の 4 文字のみが使用されます。
\$ZERROR	特殊変数 \$ZERROR。\$ZE と省略できます。

## 概要

ZTRAP コマンドは、コマンド後置条件と引数間接指定の両方を受け入れます。ZTRAP には、以下の 3 つの形式があります。

- ・ 引数なし
- ・ 文字列引数付き
- ・ [\\$ZERROR 付き](#)

引数なしの ZTRAP はエラーを強制し、エラー・コード <ZTRAP> を表示します。

ZTRAP ztraparg はエラーを強制し、エラー・コード <Zxxxx> を表示します。xxxx は、ztraparg によって指定された文字列の最初の 4 文字です。引用符付きの文字列リテラルではなく式を指定した場合、コンパイラは式を評価して、結果の文字列の最初の 4 文字を使用します。式を評価するとき、InterSystems IRIS は、プラス符号と、先頭および末尾のゼロを数字から切り離します。ztraparg の残りすべての文字は無視されます。

ZTRAP \$ZERROR は新しいエラーを強制しません。ZTRAP\$ZERROR は現行のプログラム・スタック・レベルで実行を停止し、別のエラー・ハンドラが見つかるまでスタック・レベルをポップします。その後、現行のエラー・コードを使用して、そのエラー・ハンドラで実行を続行します。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、["コマンド後置条件式"](#) を参照してください。

### ztraparg

文字列リテラルまたは文字列に評価される式。ztraparg には、以下のいずれかの値を指定できます。

- ・ 任意の文字を含む任意の長さの引用符付き文字列。ZTRAP は最初の 4 文字のみを使用して、エラー・コードを生成します。文字列が 4 文字未満の場合は、指定されたすべての文字列が使用されます。常に大文字となるシステム・エラー・コードとは異なり、大文字/小文字は維持されます。以下にコード例を示します。



## ObjectScript

```

ZTRAP "FRED" ; generates <ZFRED>
ZTRAP "Fred" ; generates <ZFred>
ZTRAP "Freddy" ; generates <ZFred>
ZTRAP "foo" ; generates <Zfoo>
ZTRAP " foo" ; generates <Z foo>
ZTRAP "@#$$" ; generates <Z@#$$>
ZTRAP "" ; generates <Z>
ZTRAP " "" " ; generates <Z">

```

- ・ 文字列として評価される式。

## ObjectScript

```

ZTRAP 1234 ; generates <Z1234>
ZTRAP 2+2 ; generates <Z4>
ZTRAP 10/3 ; generates <Z3.33>
ZTRAP +0.700 ; generates <Z.7>
ZTRAP $ZPI ; generates <Z3.14>
ZTRAP $CHAR(64)_$CHAR(37) ; generates <Z@%>
ZTRAP "" ; generates <Z>
ZTRAP " "" " ; generates <Z">

```

ZTRAP コマンドは、引数間接指定を受け入れます。詳細は、“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

**\$ZERROR によるエラー・ハンドラへの制御渡し**

ZTRAP の引数が特殊変数 \$ZERROR のときは、\$ZTRAP エラー・ハンドラ内で有効な特別な処理が実行されます。ZTRAP \$ZERROR は新しいエラーを強制しません。ZTRAP \$ZERROR は現行のプログラム・スタック・レベルで実行を停止し、別のエラー・ハンドラが見つかるまでスタック・レベルをポップします。その後、現行のエラー・コードを使用して、そのエラー・ハンドラで実行を続行します。このエラー・ハンドラは別のネームスペースにある場合があります。

**例**

以下の例は、式で ZTRAP コマンドを使用して、エラー・コードを生成する方法を示したものです。

## ObjectScript

```

; at this point the routine discovers an error ...
ZTRAP "ER23"
...

```

ルーチンが実行され、予想されたエラー条件が発見されたら、出力は以下のようになります。

```
<ZER23>label+offset^routine
```

以下の例は、後置条件を使用すると ZTRAP コマンドにどのような影響を与えるかを示しています。

## ObjectScript

```

;
ZTRAP:y<0 "yNEG"
;

```

ルーチンが実行され、y が負数である場合、出力は以下のようになります。

```
<ZyNEG>label+offset^routine
```

以下の例は、ZTRAP コマンドの引数間接指定をどのように使用するかを示したものです。

## ObjectScript

```

;
SET ERPTR="ERMSG"
SET ERMSG="WXYZ"
;
;
ZTRAP @ERPTR

```

出力は、以下のようになります。

<WXYZ>label+offset^routine

以下の例は、前のコンテキスト・レベルで定義された \$ZTRAP エラー・トラップ・ハンドラを起動する ZTRAP コマンドを示します。

## ObjectScript

```

Main
NEW $ESTACK
SET $ZTRAP="OnErr"
WRITE !,"$ZTRAP set to: ",$ZTRAP
WRITE !,"Main $ESTACK= ",$ESTACK // 0
WRITE !,"Main $ECODE= ",$ECODE," $ZERROR=", $ZERROR
DO SubA
WRITE !,"Returned from SubA" // not executed
WRITE !,"MainReturn $ECODE= ",$ECODE," $ZERROR=", $ZERROR
QUIT
SubA
WRITE !,"SubA $ESTACK= ",$ESTACK // 1
ZTRAP
WRITE !,"SubA $ECODE= ",$ECODE," $ZERROR=", $ZERROR
QUIT
OnErr
WRITE !,"OnErr $ESTACK= ",$ESTACK // 0
WRITE !,"OnErr $ECODE= ",$ECODE," $ZERROR=", $ZERROR
QUIT

```

## 関連項目

- ・ [\\$ZERROR 特殊変数](#)
- ・ [\\$ZTRAP 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)
- ・ [ラベル](#)

## ZWRITE (ObjectScript)

変数名とその値または式の値 (あるいはその両方) を表示します。

### 構文

```
ZWRITE:pc expression,...
ZW:pc expression,...
```

### 引数

引数	説明
pc	オプション - 後置条件式
expression	オプション - 表示する変数または式、変数のコンマ区切りリスト、または表示する式のいずれかまたはすべて。コンマ区切りリストには、任意の組み合わせの変数と式を含めることができます。

### 概要

ZWRITE コマンドは、変数の名前とその値をリストします。これらの変数とその子孫を、現在のデバイスに 1 行に 1 変数ずつ、正規順序の varname=value 形式でリストします。ZWRITE は、式の値もリストします。式は、値として、1 行に 1 ずつ、指定された順序でリストされます。ZWRITE コマンドには、以下の 2 つの基本形式があります。

- ・ [引数なし](#)
- ・ [引数付き](#)

ZWRITE は、オプションの後置条件式を指定できます。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

ZWRITE リストは、CTRL-C を発行すると中断され、〈INTERRUPT〉エラーが発生することがあります。

### 引数なしの ZWRITE

expression 引数なしの ZWRITE は、[引数なしの WRITE](#) と機能的に同じです。これは、プライベート変数を含むローカル変数環境 ([ローカル変数](#)) のすべての変数の名前と値を表示します。プロセス・プライベート・グローバルまたは特殊変数は表示しません。ASCII 順で名前ごとに変数をリストします。添え字ツリー順で添え字付き変数をリストします。

引数なしの ZWRITE は、ローカル変数に割り当てられた OREF 値を variable=<OBJECT REFERENCE>[oref] として表示します。JSON 配列または JSON オブジェクトに設定されたローカル変数でも、同じ結果が表示されます。OREF についてのさらなる詳細は表示されません。OREFの詳細は、“[OREF の基本](#)”を参照してください。

引数なしの ZWRITE は、ローカル変数に割り当てられたビット文字列を圧縮文字列として表示します。これは、(非表示文字を含むため) 空の文字列のようになる場合があります。ビット文字列についてのさらなる詳細は表示されません。

詳細は、“[WRITE](#)” コマンドを参照してください。

### 引数付きの ZWRITE

引数付きの ZWRITE は、1 つの expression 引数またはコンマ区切りの expression 引数リストを指定できます。これらの引数は、左から右の順に評価されます。各引数は、変数または式を指定できます。expression がコンマ区切りのリストである場合、各変数または式は個別の行に表示されます。

- ・ [変数](#):varname=value として表示されます。

- ・ **式**:評価され、その結果が value として表示されます。
- ・ **特殊変数**:value として表示されます。
- ・ **InterSystems IRIS リスト構造**:`$lb(element1,element2)` として表示されます。
- ・ **添え字付きグローバル**は、SQL データ値や SQL インデックス値の格納に使用されるものを含め、`$lb()` リスト構造として表示されます。
- ・ **オブジェクト参照**:<OBJECT REFERENCE>[oref] として表示されます。これは、引数なしの ZWRITE によって表示される値です。オブジェクト参照引数付きの ZWRITE は、この値に加えて追加の“一般情報”、“属性値”、および(該当する場合には)“スウィズル参照”と“計算参照”の情報を表示します。
- ・ **JSON 配列と JSON オブジェクト**:JSON 値として表示されます。
- ・ **ビット文字列**:`$ZCHAR` 圧縮バイナリ文字列と、ビット文字列のすべての“1”ビットのユーザが読み取れる形式の両方として表示されます。

ZWRITE は、1 つまたは複数の非数字文字を含む文字列を引用符付き文字列として表示します。

ZWRITE は、**キャノニック形式の数**として数値を表示します。ZWRITE は、キャノニック形式の数字を含む数値文字列を引用符なしのキャノニック形式の数として表示します。ZWRITE は、キャノニック形式でない数値文字列を引用符付きの文字列として表示します。キャノニック形式でない数値文字列に対して算術演算を行うと、文字列がキャノニック形式の数に変換されます。詳細は、以下の例を参照してください。

## ObjectScript

```
SET numcanon=7.9      // returns number
SET num=+007.90      // returns number
SET strnum="+7.9"     // returns string
SET strcanon="7.9"    // returns number
SET strnumop="+7.90"  // returns number
ZWRITE numcanon,num,strnum,strcanon,strnumop
```

ZWRITE は、長すぎる文字列の表示を切り捨て、文字列表示が切り捨てられていることを示す ... を追加します。

ZWRITE は、制御文字(`$LISTBUILD` および `$BIT` を使用して作成されたものを含む)を含む値を、読みやすい形式で表示します。この書式設定が、**最大文字列長**を超える、長すぎる文字列値の原因になる場合、ZWRITE は表示される文字列を切り捨て、文字列が切り捨てられていることを示す ... を追加します。

ZWRITE と、`WRITE`、`ZZDUMP`、および `ZZWRITE` コマンドとの比較は、“**表示 (書き込み) コマンド**”を参照してください。

## 変数

expression が変数である場合、ZWRITE は varname=value を別個の行に書き出します。変数は、**ローカル変数**、**プロセス・プライベート・グローバル**、**グローバル変数**、または**オブジェクト参照 (OREF)**にすることができます。

ZWRITE は未定義の変数は無視します。エラーの発行はしません。コンマ区切りの変数リスト内で未定義の変数を1つ以上指定すると、ZWRITE は未定義の変数無視して、定義された変数を返します。この動作により、複数の変数を、そのすべてが定義済みであるかどうかを判別するための確認なしで表示できます。未定義の変数を `WRITE`、`ZZDUMP`、または `ZZWRITE` に指定すると、InterSystems IRIS は <UNDEFINED> エラーを発行します。

変数には添え字を付けることができます。変数が定義済みのサブノードを持つ場合、ZWRITE は各サブノードに対して varname=value 行を、添え字ツリー順に書き出します。ルート・ノードを指定すると、ZWRITE は、そのルート・ノードが定義されていなくても、そのサブノードをすべて表示します。

拡張グローバル参照を使用して、現在のネームスペースにマップされていないグローバル変数を指定することができます。ZWRITE は、`%SYSTEM.Process` クラスの `RefInKind()` メソッドまたは `Config.Miscellaneous` クラスの `RefInKind` プロパティが拡張グローバル参照を削除するために設定されている場合でも、拡張グローバル参照を表示します。存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行し、続けてグローバル名とデータベース・パ

スを表示します (例:<PROTECT> ^myglobal,c:\intersystems\iris\mgr\)。添え字付き変数と拡張グローバル参照に関する詳細は、“[グローバルについての正式な規則](#)”を参照してください。

### 非表示文字

ZWRITE はすべての表示文字を表示します。非表示文字は [\\$CHAR](#) 関数を使用して表示し、非表示文字それぞれを連結された \$c(n) 値として表します。非表示制御文字は実行しません。詳細は、以下の例を参照してください。

#### ObjectScript

```
SET charstr=$CHAR(65,7,66,67,0,68,11,49,50)
ZWRITE charstr
```

### 式

expression がリテラル式である場合、ZWRITE はその式を評価し、結果の value を個別の行に書き出します。式に未定義の変数が含まれている場合、InterSystems IRIS は <UNDEFINED> エラーを発行します。

式が多次元のプロパティである場合、ZWRITE はプロパティの子孫を表示しません。ZWRITE で多次元のプロパティ全体を表示するには、それに [MERGE](#) を実行してローカル配列にマージするか、またはオブジェクト全体を表示します。

### InterSystems IRIS リスト構造

ZWRITE への InterSystems IRIS リスト構造 (%List) は、変数または式として指定できます。ZWRITE は、リスト構造を \$lb(element1,element2) として表示します。詳細は、以下の例を参照してください。

#### ObjectScript

```
SET FullList = $LISTBUILD("Red","Blue","Green","Yellow")
SET SubList = $LIST(FullList,2,4)
SET StrList = $LISTFROMSTRING("Crimson^Azure^Lime","^")
ZWRITE FullList,SubList,StrList
```

### 添え字付きグローバルとその下位ノード

以下は、添え字付きグローバル変数とそのすべての下位ノードの内容を表示する ZWRITE の例です。この例では、SQL テーブルに投影される、Sample.Person 永続クラスに対して定義されたデータを表示します。このグローバル変数は、その名前を永続クラス名 (SQL テーブル名ではありません) から取得し、大文字と小文字を区別します。また、データ・グローバルであることを示すため、“D” が付加されます。下位ノードには、[\\$LISTBUILD](#) (\$lb) 構文として表示されるリスト構造が含まれています。

#### ObjectScript

```
ZWRITE ^Sample.PersonD
```

単一のデータ・レコードを表示するには、以下に示すように、[RowID](#) 値をグローバル添え字として指定できます。

#### ObjectScript

```
ZWRITE ^Sample.PersonD(22)
```

[インデックス](#)の内容を表示するには、“I”を付加した永続クラス名を指定し、インデックス名を添え字として入力できます。インデックス名では、大文字と小文字が区別されます。

#### ObjectScript

```
ZWRITE ^Sample.PersonI("NameIDX")
```

リストで使用されるその他の出力不能文字も表示されます。

以下は、拡張グローバル参照を使用して、指定したネームスペース内の添え字付きグローバル変数の内容を表示する ZWRITE の例です。

### ObjectScript

```
ZWRITE ^[ "USER" ]Sample.PersonD
```

ネームスペース名は、別のネームスペースでも現在のネームスペースでもかまいません。ネームスペース名は、大文字と小文字を区別しません。

\$QUERY または \$NAME によって返されたグローバルからの拡張グローバル参照を削除するために設定可能な **ReflnKind** メソッドまたはプロパティが設定されているかどうかに関係なく、ZWRITE は常に拡張グローバル参照を表示します。

## オブジェクト参照

ZWRITE へのオブジェクト参照 (OREF) は、変数または式のいずれかとして指定できます。オブジェクト参照を指定した場合、ZWRITE は `variable=9@%SQL.Statement ; <OREF>` や `9@%SQL.Statement ; <OREF>` などの値を表示し、一般情報、属性値、および (該当する場合には) オブジェクトのプロパティのスウィズル参照と計算参照も 1 行に 1 属性ずつ表示します。

注釈 <OREF> 識別子の接尾語は、ブラウザ・インタフェースを介して ZWRITE を実行したときには、ブラウザが角括弧をタグと解釈するため、表示されない場合があります。

ZWRITE 引数が埋め込みオブジェクト・プロパティである場合、ZWRITE は、コンテナ・プロパティの配列要素の一般的な情報と属性値を、1 行に 1 属性ずつ表示します。表示形式は、`%SYSTEM.OBJ.Dump()` メソッドと同じです。

以下の例では、OREF に続いて “一般情報”、“属性値”、および “スウィズル参照” を表示します。

### ObjectScript

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

以下の例では、OREF に続いて “一般情報”、“属性値”、および “計算参照” を表示します。

### ObjectScript

```
SET doref=##class(%iKnow.Domain).%New("mytempdomain")
DO doref.%Save()
SET domId=doref.Id
ZWRITE doref
SET stat=##class(%iKnow.Domain).%DeleteId(domId)
```

以下の例では、OREF に続いて “一般情報”、“属性値”、“スウィズル参照”、および “計算参照” を表示します。

### ObjectScript

```
SET poref=##class(Sample.Person).%OpenId(1)
ZWRITE poref
```

### ObjectScript

```
SET myquery = "SELECT TOP 2 Name,DOB FROM Sample.Person"
SET oref = ##class(%SQL.Statement).%New()
SET qStatus = oref.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = oref.%Execute()
ZWRITE rset
```

OREFの詳細は、“[OREF の基本](#)”を参照してください。

## JSON 配列と JSON オブジェクト

引数なしの ZWRITE は、JSON 動的配列と JSON ダイナミック・オブジェクトをオブジェクト参照として表示します。以下に例を示します。

```
USER>SET jarray = ["apples","oranges"]
USER>SET jobj = {"fruit":"apples","count":24}
USER>ZWRITE
jarray=<OBJECT REFERENCE>[1@%Library.DynamicArray]
jobj=<OBJECT REFERENCE>[2@%Library.DynamicObject]
```

引数付きの ZWRITE は、JSON 動的配列と JSON ダイナミック・オブジェクトの値を表示します。詳細は、以下の例を参照してください。

JSON 配列:

```
USER>SET jarray = ["apples","oranges"]
USER>SET jobj = {"fruit":"apples","count":24}
USER>ZWRITE jarray
jarray = ["apples","oranges"] ; <DYNAMIC ARRAY>
USER>ZWRITE jobj
jobj = {"fruit":"apples","count":24} ; <DYNAMIC OBJECT>
```

引数なしの ZWRITE は、以下の例に示すように OREF 値を JSON オブジェクト内に表示します。

```
USER>SET oref = ##class(%SQL.Statement).%New()
USER>SET jobj = {"ObjRef":(oref)}
USER>ZWRITE
jobj=<OBJECT REFERENCE>[4@%Library.DynamicObject]
oref=<OBJECT REFERENCE>[6@%SQL.Statement]
```

JSON オブジェクト OREF 引数付きの ZWRITE は、("6@%SQL.Statement") のような形式を使用して OREF 値を JSON オブジェクト内に表示します。

```
USER>SET orefsql = ##class(%SQL.Statement).%New()
USER>SET jobjsql = {"ObjRef":(orefsql)}
USER>SET orefrs = ##class(%ResultSet).%New()
USER>SET jobjrs = {"ObjRef":(orefrs)}
USER>ZWRITE jobjsql
jobjsql={"ObjRef":("6@%SQL.Statement")} ; <DYNAMIC OBJECT>
USER>ZWRITE jobjrs
jobjrs={"ObjRef":("7@%Library.ResultSet")} ; <DYNAMIC OBJECT>
```

オブジェクト参照 (OREF) の詳細は、"[OREF の基本](#)" を参照してください。

ZWRITE は、JSON オブジェクト内の **\$DOUBLE** 値を以下のように処理します。

引数なしの ZWRITE

```
USER>SET jnum = {"doub":($DOUBLE(1.234))}
USER>SET jnan = {"doub":($DOUBLE("NAN"))}
USER>SET jinf = {"doub":($DOUBLE("-INF"))}
USER>ZWRITE
jinf=<OBJECT REFERENCE>[6@%Library.DynamicObject]
jnan=<OBJECT REFERENCE>[5@%Library.DynamicObject]
jnum=<OBJECT REFERENCE>[4@%Library.DynamicObject]
```

**\$DOUBLE** JSON オブジェクト引数付きの ZWRITE

```
USER>SET jnum = {"doub":($DOUBLE(1.234))}
USER>SET jnan = {"doub":($DOUBLE("NAN"))}
USER>SET jinf = {"doub":($DOUBLE("-INF"))}
USER>ZWRITE jnum
jnum={"doub":1.2339999999999999857} ; <DYNAMIC OBJECT>
USER>ZWRITE jnan
jnan={"doub":($DOUBLE("NAN"))} ; <DYNAMIC OBJECT>
USER>ZWRITE jinf
jinf={"doub":($DOUBLE("-INF"))} ; <DYNAMIC OBJECT>
```

ObjectScript 内での JSON の処理の詳細は、"[SET コマンド](#)" を参照してください。



## ビット文字列

ZWRITE へのビット文字列は、変数または式のいずれかとして指定できます。ZWRITE 引数が (\$BIT 関数を使用して作成された) InterSystems IRIS 圧縮ビット文字列である場合、ZWRITE は、圧縮バイナリ文字列の 10 進表現を \$ZWCHAR (\$zwc) 2 バイト (ワイド) 文字として表示します。

また、ZWRITE は、コンマ区切りリストとして、左から右の順に、圧縮されていない “1” ビットをリストするコメントを表示します。3 つ以上の連続した “1” ビットがある場合は、それらを 2 ドット構文 (n..m) を使用して範囲 (n と m も範囲に含む) としてリストします。例えば、ビット文字列 [1,0,1,1,1,0,1] は、/\*\$bit(1,3..6,8)\*/ と表示されます。ビット文字列 [1,1,1,1,1,1,1] は、/\*\$bit(1..8)\*/ と表示されます。ビット文字列 [0,0,0,0,0,0,0] は、/\*\$bit()\*/ と表示されます。以下の例は、ZWRITE ビット文字列の出力を示しています。

### ObjectScript

```
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 0
SET $BIT(a,5) = 1
SET $BIT(a,6) = 1
SET $BIT(a,7) = 1
SET $BIT(a,8) = 0
ZWRITE a
```

## 例

### 引数なしの ZWRITE

以下の例では、引数なしの ZWRITE が、定義済みのすべてのローカル変数を ASCII 名の順番でリストします。

### ObjectScript

```
SET A="A",a="a",AA="AA",aA="aA",aa="aa",B="B",b="b"
ZWRITE
```

これは、以下を返します。

```
A="A"
AA="AA"
B="B"
a="a"
aA="aA"
aa="aa"
b="b"
```

以下の例では、引数なしの ZWRITE が、キャノニック形式およびキャノニック形式でない数値をリストします。

### ObjectScript

```
SET w=10
SET x=++0012.00
SET y="6.5"
SET z="007"
SET a=w+x+y+z
ZWRITE
```

これは、以下を返します。

```
a=35.5
w=10
x=12
y=6.5
z="007"
```

### 引数付きの ZWRITE

以下の例では、ZWRITE は 3 つの変数を varname=value として、それぞれ個別の行で表示します。

## ObjectScript

```
SET alpha="abc"  
SET x=100  
SET y=80  
SET sum=x+y  
ZWRITE x,sum,alpha
```

以下の例では、ZWRITE は最初の引数の式を評価します。これは式を value として、変数を varname=value として返します。

```
SET x=100  
SET y=80  
ZWRITE x+y,y
```

以下の例では、さまざまな変数値を表示するときに ZWRITE と WRITE を比較します。ZWRITE は、引用符区切り文字列を返しますが、WRITE は返しません。

## ObjectScript

```
SET a=+007.00  
SET b=9E3  
SET c="+007.00"  
SET d=""  
SET e="Rhode Island"  
SET f="Rhode_"Island"  
ZWRITE a,b,c,d,e,f  
WRITE !,a,!,b,!,c,!,d,!,e,!,f
```

## 添え字サブノードを表示する ZWRITE

以下は、添え字付きプロセス・プライベート・グローバル変数のコンテンツを表示する ZWRITE の例です。ZWRITE は、変数の添え字を階層順に表示します。

## ObjectScript

```
SET ^|fruit(1)="apple",^||fruit(4)="banana",^||fruit(8)="cherry"  
SET ^|fruit(1,1)="Macintosh",^||fruit(1,2)="Delicious",^||fruit(1,3)="Granny Smith"  
SET ^|fruit(1,2,1)="Red Delicious",^||fruit(1,2,2)="Golden Delicious"  
SET ^|fruit="Fruits"  
WRITE "global arg ZWRITE:",!  
ZWRITE ^||fruit
```

ルート・ノードを指定すると、そのルート・ノード自体が定義されていなくても、すべてのサブノードが表示されます。

## ObjectScript

```
SET fruit(1)="apple",fruit(4)="banana",fruit(8)="cherry"  
SET fruit(1,1)="Macintosh",fruit(1,2)="Delicious",fruit(1,3)="Granny Smith"  
SET fruit(1,2,1)="Red Delicious",fruit(1,2,2)="Golden Delicious"  
WRITE "global arg ZWRITE:",!  
ZWRITE fruit
```

## 関連項目

- [WRITE コマンド](#)
- [ZZDUMP コマンド](#)
- [ZZWRITE コマンド](#)
- [表示 \(書き込み\) コマンド](#)

## ZZDUMP (ObjectScript)

16 進ダンプ形式で式を表示します。

### 構文

```
ZZDUMP:pc expression,...
```

### 引数

引数	説明
pc	オプション - 後置条件式
expression	16 進ダンプ形式で表示されるデータ。数字、引用符で囲まれた文字列、またはこれらのどちらかに解決される変数を指定できます。単一の expression、またはコンマ区切りの式のリストを指定できます。

### 概要

ZZDUMP は、16 進ダンプ形式で式を表示します。ZZDUMP は、主としてシステム・プログラマのための関数ですが、制御文字を含む文字列を表示する場合にも役立ちます。

ZZDUMP は以下の形式で数値または文字列値を返します。

```
position: hexdata printdata
```

### 引数

#### pc

InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に ZZDUMP コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

#### expression

数値、文字列リテラル、またはこれらのどちらかに解決される変数として expression を指定できます。単一の式、または式のコンマ区切りのリストを指定できます。式のコンマ区切りのリストを指定すると、各式に個別の ZZDUMP コマンドを発行するものと解析されます。コンマ区切りリストの実行は、最初のエラーが発生した時点で停止します。

expression は、任意のタイプの変数にすることができます。これには、[ローカル変数](#)、[プロセス・プライベート・グローバル](#)、[グローバル変数](#)、および[特殊変数](#)も含まれます。拡張参照を使用すると、別のネームスペースのグローバル変数を指定できます。存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行し、続けてグローバル名とデータベース・パスを表示します (例:<PROTECT> ^myglobal,c:\intersystems\iris\mgr\)

非表示文字は、その 16 進数値によって hexdata で、プレースホルダ・ドット (.) によって printdata で表されます。制御文字は実行されません。

### 例

以下の例は、ZZDUMP が 2 つの単一文字の文字列変数に対して 16 進のダンプを返すことを示しています。コンマ区切りの各式は、ZZDUMP の個別の呼び出しとして処理されます。

## ObjectScript

```
SET x="A"  
SET y="B"  
ZZDUMP x,y
```

```
0000: 41                                A  
0000: 42                                B
```

以下の例は、ZZDUMP が単独のダンプ行には長すぎる文字列変数に対して 16 進ダンプを返すことを示しています。2 番目のダンプ行 (0010:) に対する position は、16 進数です。

## ObjectScript

```
SET z="ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
ZZDUMP z
```

```
0000: 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50    ABCDEFGHIJKLMNOP  
0010: 51 52 53 54 55 56 57 58 59 5A                    QRSTUVWXYZ
```

以下の例は、ZZDUMP が 3 つの変数に対して 16 進ダンプを返すことを表しています。NULL 文字列変数には、16 進ダンプが (空白の行でも) 返されないことに注意してください。また、数値はキャノニック形式に変換されます (先頭および末尾のゼロとプラス記号は削除されます)。数値を含む文字列はキャノニック形式に変換されません。

## ObjectScript

```
SET x=+007  
SET y=" "  
SET z="+007"  
ZZDUMP x,y,z
```

```
0000: 37                                7  
0000: 2B 30 30 37                    +007
```

## Unicode

ZZDUMP 式の 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はワイド文字として表されます。以下の例は、Unicode 文字を含む変数を示しています。どのような場合でも、文字はすべてワイド文字として表示されます。

## ObjectScript

```
SET x=$CHAR(987)  
SET y=$CHAR(987)_"ABC"  
ZZDUMP x,y
```

```
0000: 03DB                                ?  
0000: 03DB 0041 0042 0043            ?ABC
```

## ZZDUMP と Write コマンドとの比較

ZZDUMP と、[WRITE](#)、[ZWRITE](#)、および [ZZWRITE](#) コマンドとを比較したテーブルは、“[表示 \(書き込み\) コマンド](#)”を参照してください。

## 関連項目

- [WRITE](#) コマンド
- [ZWRITE](#) コマンド
- [ZZWRITE](#) コマンド
- [\\$CHAR](#) 関数
- [\\$DOUBLE](#) 関数

- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$ZHEX](#) 関数
- ・ [表示 \(書き込み\) コマンド](#)

## ZZWRITE (ObjectScript)

変数または式の値を表示します。

### 構文

```
ZZWRITE:pc expression,...
```

### 引数

引数	説明
pc	オプション - 後置条件式。
expression	表示する変数または式、変数のコンマ区切りリスト、または表示する式のいずれかまたはすべて。コンマ区切りリストには、任意の組み合わせの変数と式を含めることができます。

### 説明

ZZWRITE コマンドは式を評価し、現在のデバイスの値を表示します。この式には、リテラル、[ローカル変数](#)、[プロセス・プライベート・グローバル](#)、[グローバル変数](#)、または[特殊変数](#)を指定できます。ZZWRITE は、式のコンマ区切りリストを評価できます。expression を 1 行に 1 つずつ、指定された順に結果を表示します。ZZWRITE は、それぞれの式の結果を %val=value として表示します。

引数なしの ZZWRITE は空命令となります。操作は何も実行されず、エラーも発行されません。

### ZZWRITE と ZWRITE

ZZWRITE は ZWRITE と同様に、InterSystems IRIS リスト、ビット文字列、%Status 文字列などの非表示文字およびエンコード・データを人が読める形式で表示します。制御文字は実行されません。どちらのコマンドも、オブジェクト参照 (oref) 値の広範な表示を提供します。これは、oref 値に続いて同じ "一般情報"、"属性値"、および (該当する場合には) %SYSTEM.OBJ.Dump() メソッドによって返される "スウィズル参照" および "計算参照" から構成されます。

ZZWRITE は、引数付きの [ZWRITE](#) と同じデータ値を表示しますが、以下の相違点があります。

- 変数名: ZZWRITE は、それぞれの式または変数の値を %val=value として表示します。ZWRITE は、ローカル、プロセス・プライベート、およびグローバルの各変数を varname=value として表示し、リテラル、式、および特殊変数を value として表示します。
- 未定義の変数: ZZWRITE は、未定義の変数に対して <UNDEFINED> エラーを発行します。ZWRITE は未定義の変数は無視します。
- 添え字: ZZWRITE は指定した添え字ノードの値を表示します。ZWRITE は、添え字ノードおよび定義されているすべてのサブノードを添え字ツリー順に表示します。
- 拡張グローバル参照: ZZWRITE は、拡張グローバル参照の値を %val=value として表示し (その他の expression と同様)、値が別のネームスペースで定義されていることを示しません。ZWRITE は、拡張グローバル参照の変数名を表示し、このグローバルを含むネームスペースを示します。

さまざまなデータ値の表示方法の詳細は、[ZWRITE](#) を参照してください。

ZZWRITE と、[WRITE](#)、[ZWRITE](#)、および [ZZDUMP](#) コマンドとを比較したテーブルは、"[表示 \(書き込み\) コマンド](#)" を参照してください。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### expression

評価する式、または式のコンマ区切りリスト。expression は、ローカル変数、プロセス・プライベート・グローバル、グローバル変数、または特殊変数で構成する(またはこれらを含める)ことができます。プライベート変数にすることはできません。変数には添え字を付けることができます。式は、厳密に左から順に評価されます。

拡張グローバル参照を使用して、現在のネームスペースにマップされていないグローバル変数を指定することができます。存在しないネームスペースを指定した場合、InterSystems IRIS は <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行し、続けてグローバル名とデータベース・パスを表示します (例:<PROTECT> ^myglobal,c:\intersystems\iris\mgr\)。添え字付き変数と拡張グローバル参照に関する詳細は、“[グローバルについての正式な規則](#)” および “[拡張グローバル参照](#)” を参照してください。

## 関連項目

- ・ [WRITE コマンド](#)
- ・ [ZWRITE コマンド](#)
- ・ [ZZDUMP コマンド](#)
- ・ [表示 \(書き込み\) コマンド](#)





# ルーチンおよびデバッグ・コマンド

以下は、ObjectScript がサポートするルーチンおよびデバッグ・コマンドのリファレンス・ページです。これらのコマンドは、このドキュメントの前のセクションで説明されている ObjectScript の汎用コマンドを補足します。コマンドはアルファベット順で示されます。

ObjectScript コマンド構文と規則の[基本的な説明](#)は、汎用コマンドのリファレンス・ページの前半を参照してください。ObjectScript コマンドの詳細は、“[コマンド](#)”を参照してください。

## PRINT (ObjectScript)

現在のデバイスに現在のルーチンからのコード行を表示します。

### 構文

```
PRINT:pc lineref1:lineref2
P:pc lineref1:lineref2
```

### 引数

引数	説明
pc	オプション - 後置条件式。
lineref1	オプション - 表示される行、もしくは表示される行の範囲の最初の行。リテラルとして指定します。ラベル名、数値オフセット (+n)、またはラベル名と数値オフセットのいずれかになります。省略した場合、現在のルーチン全体が表示されます。
:lineref2	オプション - 表示される行の範囲の最後の行。リテラルとして指定します。範囲を指定するには、lineref1 を指定する必要があります。

### 概要

PRINT コマンドは、現在ロードされているルーチンからコードの行を表示します。ルーチンをロードするには、[ZLOAD](#) を使用します。ZLOAD は、ルーチンの INT コード・バージョンをロードします。現在のルーチンの名前については、[\\$ZNAME](#) 特殊変数にアクセスします。

出力は、現在のデバイスに送られます。ターミナルから呼び出した場合、現在の出力デバイスは既定でターミナルになります。現在のデバイスは、USE コマンドで設定できます。現在のデバイスのデバイス ID については、[\\$IO](#) 特殊変数にアクセスします。

注釈 PRINT コマンドと [ZPRINT](#) コマンドは機能的に同一です。

PRINT は、ルーチンの INT コード・バージョンを表示します。INT コードではプリプロセッサ文はカウントされず、含まれることもありません。ルーチンの MAC バージョンの完全な空白行は、ソース・コード内にあっても、複数行コメント内にあっても、コンパイラによって削除されます。したがって、INT ルーチンには表示もカウントもされません。このため、PRINT では、MAC ルーチン内の以下の複数行コメントは、3 行ではなく 2 行で表示およびカウントされます。

#### ObjectScript

```
/* This comment includes
   a blank line */
```

MAC コード内のコメント `#;`、`##;`、および `///` は、INT コードには表示されないことがあるため、行数やオフセットに影響が生じる場合があります。詳細は、“[MAC コードのルーチンおよびメソッドに使用するコメント](#)”を参照してください。

PRINT で、[編集ポインタ](#)を出力した行の末尾に設定します。例えば、PRINT の次に `ZINSERT " SET y=2"` を指定すると、行がルーチンの末尾に挿入されます。また、PRINT +1:+4 の次に `ZINSERT " SET y=2"` を指定すると、行が第 5 行として挿入されます。[\\$TEXT](#) 関数は、現在のルーチンから 1 行を出力しますが、編集ポインタは変更しません。

PRINT には、以下の 2 つの形式があります。

- ・ 引数なし
- ・ 引数付き

引数なしの PRINT は、現在ロードされているルーチンのすべてのコードの行を表示します。

引数付きの PRINT は、指定されたコードの行を表示します。PRINT lineref1 は、lineref1 によって指定された行を表示します。PRINT lineref1:lineref2 は、lineref1 から lineref2 まで (両端を含む) の行を表示します。

lineref 引数は、ルーチンの INT コード・バージョンを使用して、行および行オフセットをカウントします。ソース (MAC) バージョンに対応する行および行オフセットを正しくカウントするために、ルーチンの変更後、PRINT のルーチンを再コンパイルする必要があります。

[\\$TEXT](#) 関数を使用して、単一の INT コード行を返すことができます。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に PRINT コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、["コマンド後置条件式"](#) を参照してください。

### lineref1

印刷する行、または表示や印刷する行の範囲の最初の行。以下のいずれかの構文形式で指定できます。

- ・ +offset : offset は、現在のルーチンの行番号を指定する正の整数です。+1 はルーチンの最初の行であり、ラベル行とすることもできます。+0 は常に空の文字列を返します。
- ・ label[+offset] : label はルーチン内の[ラベル](#)であり、offset はこのラベルからカウントする行数となります (ラベル自体の offset は 0 としてカウント)。offset オプションを省略、または label+0 と指定すると、InterSystems IRIS はラベル行を印刷します。label+1 は、ラベルの後の行を出力します。

ラベルは、31 文字よりも長くすることができますが、最初の 31 文字は一意である必要があります。PRINT は、指定された label の最初の 31 文字のみと一致します。ラベル名では大文字と小文字が区別され、Unicode 文字を含めることができます。

### lineref2

表示される範囲の最後の行。lineref1 と同じ方法で指定します。lineref2 を指定するには、lineref1 を指定する必要があります。lineref1 および lineref2 はコロン (:) 記号で区切ります。コロンと lineref2 の間に空欄を入れることはできません。

lineref2 に、行のシーケンス内で lineref1 よりも前のラベルまたはオフセットが指定されている場合、PRINT は lineref2 を無視して、lineref1 で指定される単一のコード行を表示します。

lineref2 に、存在しないラベルまたはオフセットが指定されている場合、PRINT は、lineref1 からルーチンの最後までを表示します。

## 例

以下のコード行が与えられた場合を示します。

### ObjectScript

```
AviationLetters
Abc
  WRITE "A is Abel",!
  WRITE "B is Baker",!
  WRITE "C is Charlie",!
Def WRITE "D is Delta",!
  WRITE "E is Epsilon",!
  /* Not sure about E */
  WRITE "F is Foxtrot",!
```

lineref 引数なしの PRINT は、コメント行を含めて、9 行すべてを表示します。

PRINT +0 は、空の文字列を表示します。

PRINT +1 は、ラベル `AviationLetters` を表示します。

PRINT +8 は、コメント行 `/* Not sure about E */` を表示します。

PRINT +10 は、空の文字列を表示します。

PRINT Defまたは PRINT Def+0 は、行 `Def WRITE "D is Delta",!` を表示します。これは、実行可能なコードも含むラベル行です。

PRINT Def+1 は、行 `WRITE "E is Epsilon",!` を表示します。

## 範囲の例

PRINT +0:+3 は、空の文字列を表示します。

PRINT +1:+3 は、最初の 3 行を表示します。

PRINT +3:+3 は、3 行目を表示します。

PRINT +3:+1 は、3 行目を表示します。lineref2 は無視されます。

PRINT +3:Abc+1 は、3 行目を表示します。lineref1 と lineref2 はともに同一行を指定しています。

PRINT +3:abc+1 は、3 行目からルーチンの最後までを表示します。行ラベルは大文字と小文字が区別されるので、範囲の端点が判別されません。

PRINT Abc+1:+4 は、3 行目と 4 行目を表示します。

PRINT Abc+1:Abc+2 は、3 行目と 4 行目を表示します。

PRINT Abc:Def は、2、3、4、5、および 6 行目を表示します。

PRINT Abc+1:Def は、3、4、5、および 6 行目を表示します。

PRINT Def:Abc は、行 `Def WRITE "D is Delta",!` を表示します。lineref2 はコード内で先に出現しているので、無視されます。

## 関連項目

- ・ [ZPRINT コマンド](#)
- ・ [ZINSERT コマンド](#)
- ・ [ZLOAD コマンド](#)
- ・ [ZREMOVE コマンド](#)
- ・ [ZSAVE コマンド](#)
- ・ [ZZPRINT コマンド](#)
- ・ [\\$TEXT 関数](#)
- ・ [\\$IO 特殊変数](#)
- ・ [\\$ZNAME 特殊変数](#)
- ・ [コメント](#)
- ・ [ラベル](#)
- ・ [スプール・デバイス](#)

## ZBREAK (ObjectScript)

---

ブレークポイントまたはウォッチポイントを設定します。

### 構文

```
ZBREAK:pc  
ZB:pc
```

```
ZBREAK:pc location:action:condition:execute_code  
ZB:pc location:action:condition:execute_code
```

```
ZBREAK:pc /command:option  
ZB:pc /command:option
```

## 引数

引数	説明
pc	オプション - 後置条件式。
location	<p>(ブレークポイントを設定する) コード行位置、(ウォッチポイントを設定する) ローカル変数 *var あるいは \$(シングル・ステップ・ブレークポイント) を指定します。指定した位置にブレークポイント/ウォッチポイントが既に定義されている場合、新規の設定により古い設定が置き換えられます。</p> <p>location の前に記号 +、-、または -- をオプションで付けることができます。先頭に記号がない location は、指定されたブレークポイントまたはウォッチポイントを設定します。先頭の - (マイナス) 記号は、ブレークポイントまたはウォッチポイントを無効にします。先頭の + (プラス) 記号は、無効にされたブレークポイントまたはウォッチポイントを再び有効にします。先頭の -- (マイナス-マイナス) 記号は、ブレークポイントまたはウォッチポイントを削除します。</p> <p>以下の記号は、location なしで指定できます。- (マイナス) = すべてのブレークポイントとウォッチポイントを無効にします。+ (プラス) = 無効にされたすべてのブレークポイントとウォッチポイントを再び有効にします。</p>
:action	<p>オプション - ブレークポイント、またはウォッチポイントがトリガされるときに起こるアクションを指定します。アルファベット・コードとして指定します。アクション・コードは大文字の場合も小文字の場合もありますが、引用符で囲まれている必要はありません。省略される場合、既定は "B" です。省略される場合で、:condition もしくは :execute_code が指定された場合、列はプレースホルダとして表示されます。</p>
:condition	<p>オプション - 中括弧または引用符で囲んだブーリアン式。ブレークポイントまたはウォッチポイントのトリガ時に評価されます。</p> <ul style="list-style-type: none"> <li>condition が真 (1) の場合は action が実行されます。</li> <li>condition が偽の場合、action は実行されず、execute_code に記述したコードは実行されません。</li> </ul> <p>condition を指定していない場合の既定値は真です。condition を省略し、execute_code を指定する場合、condition の前のコロンをプレースホルダとして挿入する必要があります。</p>
:execute_code	<p>オプション - condition が True の場合に実行される ObjectScript コードを指定します。コードがリテラルである場合、中括弧または引用符でそのコードを囲む必要があります。</p>
/command:option	<p>すべてのブレークポイントとウォッチポイントを制御するコマンド。必ず、前にスラッシュ (/) を付ける必要があります。利用できるコマンドは、/CLEAR、/DEBUG、/TRACE、/ERRORTRAP、/INTERRUPT、/STEP、および /NOSTEP です。以下に説明されているように、/CLEAR を除く上記のコマンドはすべて option を取ります。ほとんどの /command 名は、/C や /T など、1 文字の省略形で指定できます。</p>

## 概要

ZBREAK は、コードの特定の行のブレークポイント、および特定のローカル変数のウォッチポイントを設定して、デバッグを行うためにプログラムの実行を中断できるようにしたものです。ブレークポイントまたはウォッチポイントは、一度確立さ



れると、現在のプロセスの間、あるいは明示的に削除またはクリアされるまで維持されます。ブレイクポイントおよびウォッチポイントは、複数のネームスペースにわたって永続します。

ZBREAK コマンドのさまざまな使用法は、“[デバッグ](#)” で説明されています。FOR ループでのウォッチポイントの使用については、リファレンス・ページの “[FOR](#)” コマンドの “FOR および監視ポイント” で説明します。

## 必要な権限

コードの実行時に ZBREAK 文を使用するには、ユーザに、%Development リソースを U (使用) 権限で利用できるロール (%Developer や %Manager など) が割り当てられている必要があります。ロールをユーザに割り当てるには、SQL の [GRANT](#) 文を使用するか、管理ポータルの [システム管理]→[セキュリティ]→[ユーザ] オプションを使用します。定義を編集するユーザ名を選択し、[ロール] タブを選択して、そのユーザにロールを割り当てます。

ZBREAK を使用するには、コードが存在するデータベースに対する WRITE アクセス権が必要です。アクセス権がないと、ステッピング、ブレイクポイント、およびウォッチポイントは無効になります。例えば、%SYS (IRISSYS) または IRISLIB データベースへの WRITE アクセス権を持たないユーザは、そのデータベース内のルーチンをデバッグすることはできません。InterSystems IRIS は、これらのいずれかのデータベース内のルーチンが入力されるとデバッグを無効にし、そのルーチンが終了するまでデバッグをリストアしません。ここでは、そのルーチンによって他のコードも呼び出される場合、そのコードのデバッグも無効になるという効果があります。これは、そのコードのデータベースに対する WRITE アクセス権がユーザにあるかどうかに関係ありません。

## ブレイクポイントのリスト

引数なしの ZBREAK コマンドでは、現在のブレイクポイントおよびウォッチポイントがリストされます。リストされるのは、有効と無効の両方のブレイクポイントおよびウォッチポイントです。

## ローカル変数値のリスト

同じ行で ZBREAK コマンドの後に [引数なしの WRITE](#) コマンドを続けることができます (引数なしの ZBREAK の後にスペースを 2 つ挿入する必要があることに注意してください)。引数なしの WRITE は、実行時ではなく、ZBREAK 行の遭遇時にすべてのローカル変数の値をリストします。

## 既存のブレイクポイント/ウォッチポイントすべての有効化/無効化

ZBREAK コマンドでは、location を指定せずに、記号だけを後に付けることができます。マイナス記号 (-) 引数は、現在のブレイクポイントとウォッチポイントをすべて無効にします。プラス記号 (+) 引数は、過去に無効にされたすべてのブレイクポイントとウォッチポイントを再び有効にします。以下の location なしの ZBREAK コマンドがサポートされています。

形式	説明
ZBREAK -	既存のすべてのブレイクポイントとウォッチポイントを無効にします。
ZBREAK +	過去に無効にされたすべてのブレイクポイントとウォッチポイントを再び有効にします。削除されたブレイクポイントおよびウォッチポイントを再び有効にすることはできません。

## ZBREAK ヘルプ・テキスト

ターミナル・プロンプトでの ZBREAK に関するオンライン・ヘルプ・テキストを表示するには、以下のように ? マークを指定します。

```
USER>ZBREAK ?
```

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### location

ブレークポイントやウォッチポイントの設定または設定解除を行う場合は、location 引数が必要です。この引数では、ブレークポイントまたはウォッチポイントの指定値の前に記号 (プラス、マイナス、またはマイナスマイナス) が付く場合があります。記号とブレークポイントまたはウォッチポイントの指定値のさまざまな組み合わせがサポートされています。

- ・ `*varname` : ローカル変数。変数値が設定されるたびにウォッチポイントを確立します。先頭のアスタリスクは必須です。未定義のローカル変数にウォッチポイントを設定しても、エラーは発生しません。ウォッチポイントごとに、`action`、`condition`、または `execute_code` を任意で指定できます。これらの任意の引数は省略されることもありますが、指定された場合、もしくはスキップされた場合、必須のコロン・セパレータ (:) を指定する必要があります。
- ・ `$` : シングル・ステップ・ブレークポイント。
- ・ ブレークポイントを指定するために `label+offset^routine` として指定される、行参照。label、+offset、および ^routine (この順番) の任意の組み合わせを指定できます。label はパブリックまたはプライベートのプロシージャで使用できます。label を省略すると、ブレークポイント位置は、ルーチンの先頭からのオフセットによりカウントされます (ZBREAK 文は、オフセット行としてカウントされます)。+offset を省略すると、ブレークポイント位置は、指定されたラベル行になります。^routine を省略すると、ブレークポイント位置は、現在ロードされているルーチン内にあるものと見なされます。^routine を使用することで、現在ロードされているルーチン以外のルーチン位置を指定できます。存在しないラベル、オフセット、ルーチンに対してブレークポイントを設定しても、エラーは発生しません。

ブレークポイントを設定するには、`label+offset^routine:action:condition:execute_code` のように、`label+offset^routine` 位置、および任意の引数として `action`、`condition`、`execute_code` を指定します。これらの任意の引数は省略できますが、引数をスキップする場合、必須のコロン区切り文字 (:) を指定する必要があります。

ZBREAK では、編集ポインタは移動しません。

location 引数の先頭に、指定された位置で既存のブレークポイントまたはウォッチポイントをどう処理するかを表す記号を任意で付けることができます。記号なし (設定)、マイナス記号 (無効化)、プラス記号 (再有効化)、または 2 つのマイナス記号 (削除) を指定できます。`$` シングル・ステップ・ブレークポイントの前に記号を指定することもできます。存在しないブレークポイントまたはウォッチポイントを無効化、再有効化、または削除しようとすると、<COMMAND> エラーが発生します。

先頭の記号	意味
location	location でブレークポイントまたはウォッチポイントを設定します。
-location#delay	location でブレークポイントまたはウォッチポイントを無効化します。任意の #delay 整数は、終了する前にこのブレークポイントまたはウォッチポイントを無効化する反復回数を指定します。既定では、遭遇したブレークポイントまたはウォッチポイントをすべて無効化します。# 記号の前にスペースを入れることはできません。
+location	location でブレークポイントまたはウォッチポイントを再び有効化します。
--location	location でブレークポイントまたはウォッチポイントを削除します。ブレークポイントとウォッチポイントをすべて削除するには、ZBREAK /CLEAR を使用します。

行参照 location には、コマンドとコマンドの間の位置を指定する必要があります。コマンド式の中に設定した変数を location として使用することはできません。このタイプの変数設定は、[\\$DATA](#) 関数、[\\$ORDER](#) 関数、および [\\$QUERY](#) 関数の target パラメータで見られます。

## action

ブレークポイント/ウォッチポイントがトリガされた(引き起こされた)ときの動作を指定するコード。ブレークポイントの場合、動作はコード行が実行される前に発生します。またウォッチポイントの場合、動作はローカル変数を変更するコマンドの実行後に発生します。action は、ブレークポイントまたはウォッチポイントの設定時にのみ指定できます。無効化時および再有効化時には指定できません。

アクション・コード	説明
“B”	実行を中断し、行の位置を示すキャレット(^)と共に、ブレークが起こった行を表示します。GOTO で実行を再開します。“B” が既定です。
“L”	GOTO を使用して、行のシングル・ステップの実行を中断します。DO、XECUTE、もしくはユーザ定義関数の間で中断されたシングル・ステップ・モード。
“L+”	GOTO を使用して、行のシングル・ステップの実行を中断します。DO、XECUTE、もしくはユーザ定義関数にも適用されるシングル・ステップ・モード。
“S”	GOTO を使用して、コマンドのシングル・ステップの実行を中断します。DO、FOR、XECUTE、もしくはユーザ定義関数の間で中断されたシングル・ステップ・モード。
“S+”	GOTO を使用して、コマンドのシングル・ステップの実行を中断します。DO、FOR、XECUTE、もしくはユーザ定義関数にも適用されるシングル・ステップ・モード。
“T”	トレース・デバイスにトレース・メッセージを出力します。他の任意のアクション・コードと結合可能です。例えば、“TB” は実行の中断 (“B”) とトレース・メッセージ (“T”) の出力を意味します。“T” 自体は、実行を中断しません。このアクションは、以前の ZBREAK コマンドまたは現在の ZBREAK で ZBREAK /TRACE:ON を指定した場合にのみ効果があります。以下の <a href="#">"/TRACE</a> ” を参照してください。
“N”	このブレークポイント、またはウォッチポイントでは、何も実行されません。

## condition

ブーリアン式。True (1) の場合、action が受け取られ、(存在する場合) execute\_code は実行されます。False (0) の場合、action と execute\_code は無視されます。既定は True (1) です。

## execute\_code

実行される ObjectScript コード。このコードは、action が実行される前に実行されます。[\\$TEST](#) の値は、コードを実行する前に保存されます。コードの実行後、デバッグされたプログラムに存在する [\\$TEST](#) の値は復元されます。

execute\_code は [XECUTE](#) コマンドによって内部で実行されます。XECUTE はパブリック変数にのみアクセスできます。しかし、execute\_code に渡すパラメータを指定し、プライベート変数を XECUTE に渡すことができます。

XECUTE 引数には引用符付き文字列が含まれているため、ZBREAK execute\_code 引数を使用してコードを渡すときには引用符を二重にする必要があります。これを正確に実施することが困難な場合があります。それに代わる手段として、ZBREAK コマンドを使用すれば、execute\_code を引用符ではなく中括弧で囲むことができます。これにより、コードの中で引用符を二重にする必要がなくなります。例えば、変数 var の値を変更すると、この ZBREAK コマンドによってその新しい値が表示されます。値の表示に使用するコードは中括弧で囲みます。

```
ZBREAK *var::{"(arg) WRITE "now var=",arg,!",$GET(var,"<UNDEFINED>")})}
```

中括弧の代わりに引用符を使う場合に引用符を正しく二重にする最も簡単な方法は、まず実際の XECUTE コマンドとしてコードを書いたうえですべての引用符を二重にして、このコマンドに対応する ZBREAK execute\_code を作成することです。

例えば、以前の ZBREAK コマンドを書くときに、引数 execute\_code を中括弧ではなく引用符で囲むとします。まず、変数を表示する XECUTE コマンドを書きます。

### ObjectScript

```
XECUTE ("(arg) WRITE "now var=",arg,!",$GET(var,"<UNDEFINED>"))
```

次に、それに相当する ZBREAK コマンドを書きます。XECUTE コードを最後の引数にコピーし、コードを引用符で囲み、コードの中の引用符を二重にします。

### ObjectScript

```
ZBREAK *var:::""(arg) WRITE ""now var=""",arg,!",$GET(var,"<UNDEFINED>"))"
```

### /command:option

後続の ZBREAK コマンドに対する ZBREAK 環境を設定するために使用されるコマンド・キーワード。/CLEAR コマンドには、オプションはありません。他のコマンド・キーワードの後には、コロンで区切られたオプションが続きます。空白は使用できません。

キーワード	説明
/CLEAR	すべてのブレークポイントを削除します。
/DEBUG:device	デバッグ・デバイスのクリア、または設定を行います。
/TRACE	トレース・デバイスへのトレース・メッセージの送信を有効、または無効にします（後続の ZBREAK コマンドでの "T" アクション）。オプションは、:ON=トレースの有効化、:OFF=トレースの無効化、:ALL=すべての行をトレース、の 3 つです。デバイスを指定することで、:ON、もしくは :ALL で出力をリダイレクトできます。例えば、ZBREAK /TRACE:ON:device です。
/ERRORTRAP	<a href="#">\$ZTRAP</a> 、 <a href="#">\$ETRAP</a> 、および <a href="#">TRY/CATCH</a> エラー・トラップを有効または無効にします。オプションは、:ON と :OFF です。
/INTERRUPT	Ctrl-C アクションを指定します。オプションは、:NORMAL と :BREAK です。NORMAL の場合、Ctrl-C で実行を中断すると、<INTERRUPT> エラーが発生します。BREAK の場合、Ctrl-C で実行を中断すると <BREAK> エラーが発生し、新しいスタック・フレームが設定されます。
/STEP	コード・モジュールのステップングを有効にします。オプションは、:EXT（ユーザ言語拡張機能）、:METHOD（オブジェクト・メソッド）、:DESTRUCT（%Destruct オブジェクト・メソッド）です。
/NOSTEP	コード・モジュールのステップングを無効にします。オプションは、:EXT（ユーザ言語拡張機能）、:METHOD（オブジェクト・メソッド）、:DESTRUCT（%Destruct オブジェクト・メソッド）です。

### /TRACE

/TRACE コマンド・キーワードによって、トレース・メッセージの受信に使用するトレース出力デバイスを指定します。これは、ZBREAK に action="T" を付けて発行する前に指定する必要があります。その場合、別個の ZBREAK コマンドとして指定することも（ZBREAK /TRACE:ON:device）、ZBREAK S:"T"/TRACE:ON:device のように 2 つのコマンドを結合することもできます。

一度にアクティブにできるトレース出力デバイスは 1 つのみです。トレース出力デバイスに書き込まれるのはトレース・メッセージのみです。通常の WRITE 操作では引き続きユーザ・ターミナルに書き込まれます。

- ・ /TRACE:ON は、トレース・メッセージの受取先として InterSystems ターミナルをアクティブ化します。

- ・ `/TRACE:ON:device` は、トレース・メッセージの受取先として既存の出力デバイス（通常、.txt ファイル）をアクティブ化します。ZBREAK `/TRACE:ON:device` を呼び出す前に、[OPEN](#) コマンドを使用してデバイスを開く必要があります。開いていないデバイスを指定すると、InterSystems IRIS は `<NOTOPEN>` エラーを発行します。シーケンシャル・ファイルを開くには、ディレクトリが存在し、ファイルが存在するか、または `OPEN` コマンドで `"N"` オプションを指定してファイルを作成する必要があります。この一連の操作を、以下の Windows におけるターミナル・セッションの例に示します。

```
USER>SET btrace="C:\Logs\mydebugtrace.txt"
USER>OPEN btrace:"WN"
USER>ZBREAK /TRACE:ON:btrace
USER>ZBREAK
BREAK: Trace ON
Trace device=C:\Logs\mydebugtrace.txt
No breakpoints
No watchpoints
```

以前の ZBREAK `/TRACE:ON:device1` によってトレース出力デバイスが既にアクティブ化されている場合、ZBREAK `/TRACE:ON:device2` は `device1` トレース・デバイスを `device2` トレース・デバイスに置き換えます。

- ・ `/TRACE:ALL` は、行ステップを有効にし、各行についてメッセージをトレース・デバイスに書き込みます。この行ステップは停止せず、デバッグ対象のコードを続行します。ZBREAK `/TRACE:ALL` または ZBREAK `/TRACE:ALL:device` を指定できます。`/TRACE:ALL` は、トレース行ステップを有効にします。このオプションは、`/TRACE:ON:device` を使用してトレース・デバイスをアクティブにする前または後に呼び出すことができます。`/TRACE:ALL:device` は、トレース・デバイスをアクティブ化し、そのトレース・デバイスに対する行ステップを有効にします。`/TRACE:ALL:device` を呼び出す前に、[OPEN](#) コマンドを使用してデバイスを開く必要があります。開いていないデバイスを指定すると、InterSystems IRIS は `<NOTOPEN>` エラーを発行します。シーケンシャル・ファイルを開くには、ディレクトリが存在し、ファイルが存在するか、または `OPEN` コマンドで `"N"` オプションを指定してファイルを作成する必要があります。この一連の操作を、以下の Windows におけるターミナル・セッションの例に示します。

```
USER>SET steptrace="C:\Logs\mysteptrace.txt"
USER>OPEN steptrace:"WN"
USER>ZBREAK /TRACE:ALL:steptrace
USER>ZBREAK
ZBREAK
BREAK:L+ Trace ON
Trace device=C:\Logs\mysteptrace.txt
$ (single step) F:ET S:0 C: E:
No watchpoints
USER>
```

- ・ `/TRACE:OFF` は、現在のトレース出力デバイスを非アクティブにします。デバイスを閉じることはありません。ZBREAK `/TRACE:OFF` または ZBREAK `/TRACE:OFF:device` を指定できます。`/TRACE:OFF` は、現在のトレース出力デバイスを非アクティブにします。アクティブになっているトレース・デバイスがない場合、InterSystems IRIS は操作を何も実行せず、エラーも発行しません。`/TRACE:OFF:device` は指定されたトレース出力デバイスを非アクティブにします。指定された `device` が現在のトレース出力デバイスではない場合、`/TRACE:OFF:device` は `<NOTOPEN>` エラーを発行します。

## [/STEP と /NOSTEP](#)

`/STEP` および `/NOSTEP` コマンド・キーワードは、デバッガが特定のタイプのコード・モジュールのステップ実行を行うかどうかを制御します。

- ・ `:EXT` オプションは、[%ZLANG](#) で作成された、ユーザ記述の言語拡張機能を管理します。アプリケーションでは、この言語拡張機能は、1 つのコマンドまたは関数呼び出しとして表示されます。既定では、`action` 引数の設定に関係なく、デバッガはこれらの `%ZLANG` ルーチンのステップ実行を行いません。



- ・ `:METHOD` オプションは、アプリケーションによって呼び出されるオブジェクト・メソッドのステップングを管理します。既定では、デバッガはオブジェクト・メソッド・コードのステップ実行を行います。
- ・ `:DESTRUCT` オプションは、`%Destruct` オブジェクト・メソッドのステップングを管理します。`%Destruct` メソッドは、オブジェクトが破棄されたときに、暗黙的に呼び出されます。既定では、`%Destruct` オブジェクト・メソッドのコードがデバッガでステップ実行されます。これを無効にするには `/NOSTEP:DESTRUCT` を指定します。

以下の例に示すように、複数の `/STEP` または `/NOSTEP` オプションを指定できます。

### ObjectScript

```
ZBREAK /STEP:METHOD:DESTRUCT
```

## 例

以下の例は、引数なしの `ZBREAK` で、ブレークポイントとウォッチポイントがどのようにリストされるかを示します。最初の `ZBREAK` は、ブレークポイントもウォッチポイントもリストしません。次にプログラムが 2 つのウォッチポイントを `x` ローカル変数と `y` ローカル変数に設定し、`ZBREAK` がこれを表示します。次に、プログラムは `$` シングル・ステップ・ブレークポイントを設定し、`ZBREAK` はブレークポイントと 2 つのウォッチポイントを表示します。次に、プログラムは `x` ローカル変数ウォッチポイントを無効化します。これは `ZBREAK` の表示には影響しません。最後に、プログラムは `x` ローカル変数ウォッチポイントを削除します。このウォッチポイントが `ZBREAK` の表示から消えます。

### ObjectScript

```
ZBREAK
ZBREAK *x: "B"
ZBREAK *y: "B"
ZBREAK
ZBREAK $
ZBREAK
ZBREAK -*x
ZBREAK
ZBREAK --*x
ZBREAK
ZBREAK /CLEAR
```

`ZBREAK` のその他の使用例は、“[デバッグ](#)” を参照してください。

## 関連項目

- ・ [BREAK](#) コマンド
- ・ [FOR](#) コマンド
- ・ [OPEN](#) コマンド
- ・ [USE](#) コマンド
- ・ [デバッグ](#)

# ZINSERT (ObjectScript)

現在のルーチンに 1 つ以上のコード行を挿入します。

## 構文

```
ZINSERT:pc "code":location ,...
ZI:pc "code":location ,...
```

## 引数

引数	説明
pc	オプション - 後置条件式。
code	ObjectScript コードの行。(引用符で囲まれた) 文字列リテラル、または文字列リテラルを含む変数として指定されます。実行可能なコードの場合、最初の文字はスペースである必要があります。スペースで始まらない行は、ラベル名として扱われます。コード行には、ラベル名と、それに続いて実行可能なコードを含めることができます。
:location	オプション - ZINSERT がコードを挿入する前の行。ラベル名、数値オフセット (+n)、またはラベル名と数値オフセットのいずれかになります。location を省略すると、code は現在の行位置 (編集ポインタ) に挿入されます。

## 概要

このコマンドにより、ObjectScript のソース・コードの行が現在ロードされているルーチンに挿入され、編集ポインタが挿入行の直後に進みます。ObjectScript ソース・コードの複数の行を、コンマで区切った一連の code:location 引数として挿入できます。コード行は、指定した順序で別個の挿入操作として挿入されます。

ルーチンをロードするには、ターミナルから **ZLOAD** を使用します。ZLOAD は、ルーチンの INT コード・バージョンをロードします。INT コードではプリプロセッサ文はカウントされず、含まれることもありません。INT コードでは、ルーチンの MAC バージョンの完全な空白行はカウントされず、含まれることもありません。これは、ソース・コード内であっても、複数行コメント内であっても同じです。ルーチンがロードされると、そのルーチンは、すべてのネームスペースで、現在のプロセスに対して現在ロードされているルーチンになります。したがって、ルーチンのロード元のネームスペースだけでなく任意のネームスペースから、現在ロードされているルーチンの行の挿入または削除、ルーチンの表示、実行、アンロードを実行できます。

ターミナルから入力したり、EXECUTE コマンドまたは \$EXECUTE 関数で呼び出したりする場合は、ZINSERT コマンドのみ使用できます。ルーチンの本文で ZINSERT を指定すると、コンパイル・エラーが返されます。また、ルーチン内から ZINSERT を実行しようとしても、エラーが生成されます。

- ・ ZINSERT "code" は、指定された ObjectScript コード行を、現在のルーチンで現在の編集ポインタの位置に挿入します。
- ・ ZINSERT "code":location は、指定されたコード行を現在のルーチンの指定された行 location の後に挿入します。行位置は、ルーチンの開始からの行オフセット番号、ラベル、または指定されたラベルの行オフセット番号として指定できます。

ZINSERT がコード行を挿入した後、この新規のコード行の最後に編集ポインタを再設定します。これは、次の ZINSERT (またはコンマで区切られた一連の ZINSERT の引数に含まれる次のコード行) で明示的に location が指定されていないければ、次の ZINSERT は最後に挿入された行の直後にコード行を配置することを意味しています。

ZINSERT は、行ごとに増分コンパイルを実行します。**DO** コマンドを使用して、現在のルーチンを実行できます。

ZINSERT は、現在のルーチンのローカル・コピーにのみ影響します。ディスク上に保存されているルーチンは変更されません。挿入された行を保存するには、ZSAVE コマンドを使用して、現在のルーチンを保存する必要があります。



**\$ZNAME** 特殊変数で、現在のルーチンの名前を判別できます。**ZPRINT** を使用して、現在ロードされているルーチンの複数の行を表示できます。

## 編集ポインタ

注意 ZINSERT で編集ポインタを動かします。

編集ポインタは次のように設定します。

- ・ ZLOAD で、編集ポインタをルーチンの先頭に設定します。
- ・ ZINSERT で、編集ポインタを、挿入する行の直後に設定します。例えば、ZINSERT " SET x=1 ":+4 の次に ZINSERT " SET y=2" を指定すると、第 5 行および第 6 行が挿入されます。
- ・ ZREMOVE で、編集ポインタを、削除する行に設定します。例えば、ZREMOVE +4 の次に ZINSERT " SET y=2" を指定すると、第 4 行が削除され、第 4 行は挿入した行で置き換わります。
- ・ ZPRINT (または PRINT) で、編集ポインタを、出力した行の末尾に設定します。例えば、ZPRINT の次に ZINSERT " SET y=2" を指定すると、行がルーチンの末尾に挿入されます。また、ZPRINT +1:+4 の次に ZINSERT " SET y=2" を指定すると、行が第 5 行として挿入されます。**\$TEXT** 関数は、現在のルーチンから 1 行を出力しますが、編集ポインタは変更しません。
- ・ ZSAVE では、編集ポインタは変更されません。
- ・ DO では、編集ポインタは変更されません。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

### code

ObjectScript コードの行。(引用符で囲まれた) 文字列リテラル、または文字列リテラルを含む変数として指定されます。このコード行には、1 つ以上の ObjectScript コマンド、新規のラベル名、またはラベルと 1 つ以上のコマンドの両方を含めることができます。コードはルーチンに挿入されるので、形式は ObjectScript に従う必要があります。したがって、code 文字列リテラルの最初の文字は空白スペース (標準 ObjectScript インデント)、またはラベルのいずれかでなければなりません。また、引用符で囲む必要もあります。引用符は挿入しているコード行を囲むので、コード内の引用符は二重になります。

**%Library.Routine** クラスの **CheckSyntax()** メソッドを使用すると、code の行を挿入する前に、その行の構文チェックを実行できます。**CheckSyntax()** と **ZINSERT** の両方で、実行可能な ObjectScript コード行の前には 1 つ以上のスペースが必要です。また、インデントされていない行は、ラベル、またはラベルとそれに続く実行可能なコードとして解析されます。**CheckSyntax()** も **ZINSERT** もマクロ・プリプロセッサ・コードは解析しません。

### location

ZINSERT がコードを挿入する前の行。以下の形式のうちのいずれかをとることができます。

形式	説明
+offset	ルーチン先頭からのオフセット行数として、行の位置を特定する正の整数に評価される式。ZINSERT は、コード行を指定した行の直後に挿入します。ルーチンの先頭に行を挿入するには、+0 を指定します。このプラス符号は必須です。+offset を省略した場合、label で特定される行が検索されます。
label	現在のルーチン内の既存の行ラベル。リテラル値である必要があります。変数は、label の指定には使用できません。行ラベルでは、大文字と小文字が区別されます。省略した場合、+offset はルーチンの先頭からカウントされます。
label+offset	ラベルと、ラベルが付けられたセクション内の行のカウント・オフセットを指定します。+offset の値を省略、または label+0 を指定すると、InterSystems IRIS はラベル行を検索して、その直後に挿入します。

注釈 ZINSERT は現在のルーチン専用です。その位置に label routine を指定しようとすると、〈SYNTAX〉エラーが返されます。

コード行は番号 1 から開始されるので、+1 の位置を指定すると、ルーチンの最初の行の後にコード行が挿入されます。ルーチンの最初、またはラベル付けされたセクションの最初 (既存の最初の行の前) に行を挿入するときは、+0 のオフセットを使用します。例えば、以下のようになります。

### ObjectScript

```
ZINSERT "Altstart SET c=12,d=8":+0
```

上記の例では、ルーチンの最初にコード行を挿入します。+0 のオフセットを使用することで (または location を省略することで)、別の空の現在ルーチンに行を挿入できます。

`^ROUTINE` [グローバル](#)を使用して、INT ルーチンの行番号を返すことができます。`^ROUTINE` は、ディスクに保存されている INT ルーチンのバージョンを返します。そのため、現在のルーチンを変更しても、保存していなければ変更内容が返されないことに注意してください。`^ROUTINE` では、編集ポインタは変更されません。

ラベルは、31 文字よりも長くすることができますが、最初の 31 文字は一意である必要があります。ZINSERT は、指定された label の最初の 31 文字のみと一致します。ラベル名では大文字と小文字が区別され、Unicode 文字を含めることができます。

INT コード行には、すべてのラベル、コメント、および空白が含まれます。例外として、コンパイラによって削除される、MAC ルーチンのすべて空白の行は表示もカウントもされません。複数行コメント内の空白行も削除されます。MAC コード内のコメント `#;`、`##;`、および `///` は、INT コードには表示されないことがあるため、行数やオフセットに影響が生じる場合があります。詳細は、["MAC コードのルーチンおよびメソッドに使用するコメント"](#) を参照してください。

## ZINSERT と ZREMOVE

ZREMOVE コマンドを使用して、現在実行中のルーチンから 1 行、または複数のコード行を削除することができます。したがって ZREMOVE と ZINSERT を使用して、既存のコード行と新規の行を置換することができます。このような操作は、プロセスで現在実行されているルーチンのコピーにのみ影響を与えます。

注釈 ZINSERT は、指定された location の後に行を挿入します。ZREMOVE は、指定された位置で行を削除します。例えば、ZINSERT " SET x=1":+4 で行を挿入する場合、この行を削除するには、ZREMOVE +5 を指定する必要があります。

## ZINSERT、XECUTE、および \$TEXT

XECUTE コマンドは、ルーチンから実行可能なコードの単独の行を定義し、挿入するために使用されます。ZINSERT コマンドを使用して、ルーチンの外部から、実行可能なコードの単独の行を行位置ごとに定義し、挿入することができます。

XECUTE コマンドは、新規のラベルの定義に使用することはできません。したがって、XECUTE はコード行の最初のコマンドの前に、最初の空白スペースを必要としません。ZINSERT コマンドは、新規のラベルの定義に使用できます。したがって、ZINSERT はコマンド行の最初のコマンドの前に、最初の空白スペース (または新規ラベルの名前) を必要とします。

\$TEXT 関数を使用して、ルーチンから行位置によりコード行を抽出することができます。\$TEXT はテキスト文字列として指定されたコード行をコピーします。これによってその行の実行に影響が及んだり、現在のルーチンからの抽出時に現在の行位置 (編集ポインタ) が変更されることはありません。(\$TEXT を使用して現在のルーチン以外のルーチンからコードを抽出すると、現在の行位置が変更されます)。\$TEXT は、XECUTE コマンドにコード行を提供できます。\$TEXT は WRITE コマンドにコード行を提供することもできるので、ターミナルにコード行を提供します。

XECUTE コマンドは、ルーチンから実行可能なコードの単独の行を定義し、挿入するために使用されます。ZINSERT コマンドを使用して、ルーチンの外部から、実行可能なコードの単独の行を行位置ごとに定義し、挿入することができます。

## ZINSERT を使用したルーチンの作成

現在のルーチンがない場合、ZINSERT を使用して、名前のないルーチンを現在のルーチンとして作成できます。

1. ターミナル・プロンプトで、ObjectScript コードの最初の行を指定して ZINSERT コマンドを発行します。通常、この行は、ラベル名、またはラベル名とそれに続く実行可能な ObjectScript コードのいずれかです。この最初の行にラベル名が含まれている場合は、DO を使用して、このルーチンを保存することなく実行できます。ラベル名が含まれていない場合、このコードを実行するには、ZSAVE routine を使用して、このルーチンに名前を付けて保存する必要があります。
2. ターミナル・プロンプトで、追加の ZINSERT コマンドを発行して、現在のルーチンに行を追加します。
3. ルーチンを保存する場合は、ターミナル・プロンプトで ZSAVE routine を発行し、指定した名前でのルーチンを保存します。
4. 完了したら、引数なしの ZREMOVE を使用して、現在のルーチンをアンロードします。

## Tab キーの省略表現を使用したルーチンの作成

現在のルーチンがない場合、ターミナルでは Tab キーを使用するルーチンを作成するための省略表現も認識されます。

1 行目 (下記参照) を除き、Tab キーは事実上 ZINSERT コマンドの代替として機能します。

1. ターミナル・プロンプトで、目的のルーチン (routine) の名前を指定して Tab キーを押し、ルーチンの 1 行目を入力します。
2. 現在のルーチンに追加する行ごとに、Tab に続いてターミナル・プロンプトで行の内容を入力します。
3. Enter キーを押して、この入力を完了します。
4. ルーチンを実行するには、DO コマンドを呼び出して、引数としてルーチン名を渡します (ただし、ルーチン名の先頭にはキャレットを含めないでください。この後の例を参照してください)。
5. 実行が完了すると、routine を引数として指定しているかどうかに関係なく、ZREMOVE が起動して現在のルーチンがアンロードされます。

例えば、以下の一連のコマンドによって、1 から 10 までの番号を書き込む簡単なルーチン counter が作成され、呼び出されます。

```
USER>counter Tab for i=1:1:10 {
USER>Tab write i,!
USER>Tab }
USER>DO counter
```

この省略表現を使用しても ZPRINT は正常に動作します。ZLOAD コマンドと ZINSERT コマンドを明示的に使用して作成したルーチンと異なり、ZSAVE を使用した場合はルーチンの名前を指定できません。

## 例

以下の例では、現在のルーチンの 4 番目の行の後にコード行 `SET x=24` が挿入されます。挿入されたこのコード行はラベルで開始されていないため、最初のスペースは必要な行開始文字として含まれる必要があります。

### ObjectScript

```
ZINSERT " SET x=24":+4
```

以下の例では、3 つのコード行が挿入されます。現在のルーチンの 4 番目の行の後に `SET x=24` が挿入されます。続いて、2 番目のコード行で `location` が指定されていないため、編集ポインタの現在の位置 (`SET x=24` の直後) に `SET z=1` が挿入されます。その後、新しい行位置 +5 (`SET x=24` と `SET z=1` の間) に、`SET y=1` が設定されます。

### ObjectScript

```
ZINSERT " SET x=24":+4," SET z=1"," SET y=1":+5
```

以下の例では、現在ロードされているルーチンに、“Checktest”と呼ばれるラベルが含まれていると仮定します。ZINSERT コマンドは、Checktest 内の 6 番目の行の後に新しい行を挿入します (Checktest+6)。新規の行には、“Altcheck”というラベルと `SET y=0` というコマンドが含まれます。

### ObjectScript

```
ZINSERT "Altcheck SET y=0":Checktest+6
```

挿入されたコード行はラベル “Altcheck” で開始するので、引用符の後に最初のスペースは必要ないことに注意してください。

以下の例では、現在のルーチンの 4 行目の後にコード行 `SET x=24 WRITE !,"x is set to ",x` が挿入されます。挿入されたコード行は引用符で囲まれるので、WRITE コマンド内の引用符は二重になります。

### ObjectScript

```
ZINSERT " SET x=24 WRITE !,""x is set to """,x":+4
```

## 関連項目

- ・ [XECUTE](#) コマンド
- ・ [ZLOAD](#) コマンド
- ・ [ZPRINT](#) コマンド
- ・ [ZREMOVE](#) コマンド
- ・ [ZSAVE](#) コマンド
- ・ [\\$TEXT](#) 関数
- ・ [\\$ZNAME](#) 特殊変数

## ZLOAD (ObjectScript)

ルーチンを現在のルーチン・バッファにロードします。

### 構文

```
ZLOAD:pc routine
ZL:pc routine
```

### 引数

引数	説明
pc	オプション - 後置条件式。
routine	オプション - ロードするルーチン。単純なリテラルとして指定します。routine の値は括弧で囲みません。キャレット (^) 接頭語やファイル・タイプ接頭語は付きません。変数または式を使用して指定することはできません。省略した場合、InterSystems IRIS は現在のデバイスから名前のないルーチンをロードします。

### 概要

ZLOAD コマンドは、ObjectScript ルーチンの INT コード・バージョンを現在のルーチンとしてロードします。ZLOAD には、以下の 2 つの形式があります。

- ・ [引数なしの ZLOAD](#)
- ・ [引数付きの ZLOAD](#)

ターミナルから入力したり、XECUTE コマンドまたは \$XECUTE 関数で呼び出したりする場合は、ZLOAD コマンドのみ使用できます。このオペレーションはルーチンの実行に影響を与える可能性があるため、ルーチンの本体にコード化するべきではありません。ルーチン内で ZLOAD を指定すると、コンパイル・エラーが返されます。また、ルーチンから ZLOAD を実行しようとしても、エラーが生成されます。

ZLOAD を使用してルーチンを現在のルーチンとしてロードした後、[DO](#) コマンドを使用して現在のルーチンを実行できます。続いて、[ZINSERT](#) や [引数付きの ZREMOVE](#) を使用して現在のルーチンを編集し、[ZPRINT](#) でルーチン行を表示できます。さらに、[ZSAVE](#) を使用して、編集した現在のルーチンを保存 (オプションで名前を変更) して、最後に [引数なしの ZREMOVE](#) を使用して現在のルーチンをアンロードすることができます。

### 引数なしの ZLOAD

引数なしの ZLOAD コマンドは、名前のない ObjectScript ルーチンを、現在のプロセスを対象に現在のルーチンとしてルーチン・バッファにロードします。その後、[ZSAVE](#) routine を使用して、このルーチンに名前を付けることができます。このルーチンには名前が付いていないため、[\\$ZNAME](#) 特殊変数を使用して、現在のルーチンがロードされているかどうかを判断することはできません。

引数なしの ZLOAD には以下の 2 つの使用方法があります。

- ・ シーケンシャル・ファイルまたは他のデバイスからルーチンをロードする
- ・ ターミナルを使用してルーチンを作成する

引数なしの ZLOAD コマンドでは後置条件式を指定することができます。

### デバイスからのルーチンのロード

デバイスからルーチンをロードするには、以下を実行します。

1. OPEN コマンドを発行してデバイスを開きます。

2. USE コマンドを発行して、デバイスを現在のデバイスにします。
3. 引数なしの ZLOAD コマンドを発行して、デバイスからルーチンを現在のルーチンとしてロードします。

InterSystems IRIS が NULL 文字列行 (") を読み取るまで、行のロードを継続します。ロードされたこのルーチンは、ZSAVE routine コマンドで保存するまで名前がありません。

### ターミナルからのルーチンの作成

ターミナルから、引数なしの ZLOAD を使用して、名前のないルーチンを現在のルーチンとして作成します。

1. ターミナル・プロンプトで、引数なしの ZLOAD コマンドを発行します。
2. 次の行に、ルーチンの最初の ObjectScript コマンドを (括弧で囲まずに) 入力して、**Enter** キーを 2 回押します。通常、この行は、ラベル名、またはラベル名とそれに続く実行可能な ObjectScript コードです。実行可能な ObjectScript コードはインデントする必要があります。
3. ターミナル・プロンプトで、ZINSERT コマンドを発行して、この現在のルーチンに他の行を追加します。
4. オプションで、ターミナル・プロンプトで、ZSAVE routine を発行し、指定した名前でのこのルーチンを保存します。
5. 完了したら、引数なしの ZREMOVE を使用して、現在のルーチンをアンロードします。

または、ターミナルから、ZINSERT を使用して、名前のないルーチンを現在のルーチンとして作成することができます。ターミナルでは **Tab** も認識されます。

### 引数付きの ZLOAD

ZLOAD routine は、既存の ObjectScript ルーチンの INT コード・バージョンを、現在のプロセスの現在のルーチンとして、現在のネームスペースからルーチン・バッファにロードします。INT コードではプリプロセッサ文はカウントされず、含まれることもありません。

ZLOAD は、ルーチンをロードするときに暗黙的に引数なしの ZREMOVE を実行します。つまり、ZLOAD は、以前ロードされたすべてのルーチンを削除し、指定された routine に置き換えます。**\$ZNAME** 特殊変数で、現在ロードされているルーチンを判別できます。ルーチンをロードすると、ZLOAD は、ルーチンの先頭で行ポインタを配置します。

一度ロードされると、ユーザが別のルーチンを ZLOAD コマンドで明示的にロードするか、引数なしの ZREMOVE で削除するか、または DO コマンドや GOTO コマンドで別のルーチンを暗黙的にロードするまで、ルーチンはプロセスの現在のルーチンのままです。

ルーチンが現在のものである限り、(ZINSERT および ZREMOVE コマンドで) ルーチンを編集し、ZPRINT コマンドで行を 1 行以上表示したり、**\$TEXT** 関数で単一行を返すことができます。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### routine

現在のルーチンとしてロードする、現在のネームスペース内の既存の ObjectScript ルーチン。ルーチン名は、大文字と小文字を区別します。

ZLOAD を可能にするために routine への実行許可を持つ必要があります。この許可がない場合は、InterSystems IRIS は <PROTECT> エラーを生成します。

指定されたルーチンが存在しない場合、システムは <NOROUTINE> エラーを生成します。ZLOAD でルーチンをロードしようとして失敗した場合、現在ロードされているルーチンは削除されることに注意してください。



その後、このプロセスでエラーが発生すると、そのすべてに、現在ロードされているルーチンの名前が付加されます。これは、エラーがルーチンと関係があるかどうかにかかわらずネームスペース全体にわたって実行されます。詳細は、“\$ZERROR” 特殊変数を参照してください。

## ネームスペース

ZLOAD でロードできるルーチンは、現在のネームスペースに存在するルーチンだけです。ルーチンがロードされると、そのルーチンは、すべてのネームスペースで、このプロセスに対して現在ロードされているルーチンになります。したがって、ルーチンのロード元のネームスペースだけではなく任意のネームスペースから、現在ロードされているルーチンの行の挿入または削除、ルーチンの表示、実行、アンロードを実行できます。ZSAVE は、現在ロードされているルーチンを現在のネームスペースに保存します。したがって、ZLOAD のネームスペースが ZSAVE のネームスペースと異なる場合、ルーチンの変更されたバージョンは、ZSAVE を発行したときに現在のネームスペースであるネームスペースに保存されます。変更は、ZLOAD ネームスペース内のルーチンのバージョンでは保存されません。

## ZLOAD でのルーチンの振る舞い

ZLOAD routine を指定する場合、InterSystems IRIS はメモリ内のルーチン・バッファのプールのルーチンを検索します。ルーチンがない場合、InterSystems IRIS はルーチンの ObjectScript オブジェクト・コード・バージョンをバッファの 1 つにロードします。ObjectScript INT (中間) コードは、現在のネームスペースの対応する ^ROUTINE グローバルに残りますが、編集を行って、ZSAVE を使用して変更を保存した場合には更新されます。

例えば、ZLOAD MyTest は (まだロードされていない場合に) ルーチン MyTest のオブジェクト・コード・バージョンをロードします。MyTest ルーチンは、現在のネームスペースに存在している必要があります。

マルチユーザ環境では、LOCK プロトコルを設定し、複数のユーザが同時に同じルーチンをロードして変更するのを防ぐ必要があります。各ユーザは、対応するルーチンに対して ZLOAD を発行する前に、排他ロックを取得する必要があります。

routine を省略した場合、ZLOAD は、NULL 行を入力する (つまり、Return (Enter キー) を押す) ことでコードを終了するまで、現在のデバイス (通常はキーボード) から入力した新しいコード行をロードします。このルーチンは、次の ZSAVE コマンドで保存するまで名前がありません。

## ^rINDEX ルーチンのタイムスタンプとサイズ

以下のターミナルの例に示すように、^rINDEX グローバルを使用して、ルーチンの MAC、INT、および OBJ の各コード・バージョンのローカル・タイムスタンプと文字数を返すことができます。

```
USER>ZWRITE ^rINDEX("MyTest")
^rINDEX("MyTest","INT")=$1b("2019-12-13 06:37:49",475)
^rINDEX("MyTest","MAC")=$1b("2019-12-13 06:34:04.235011",452)
^rINDEX("MyTest","OBJ")=$1b("2019-12-13 06:37:49",476)
USER>
```

MAC のタイムスタンプは、MAC コードの変更後の最終保存日時です。INT と OBJ のタイムスタンプは、MAC コードの最終コンパイル日時です。INT コード・バージョンの変更後に ZSAVE を発行すると、INT および OBJ のタイムスタンプと文字数が更新されます。INT コードを変更せずに ZSAVE を発行すると、OBJ のタイムスタンプのみが更新されます。

## INT コードと ^ROUTINE グローバル

ルーチンの ObjectScript INT (中間) コードは、^ROUTINE グローバルに格納されます。^ROUTINE は、現在のネームスペースのルーチンにのみアクセスできます。^ROUTINE で表示されるのは、ディスク上にあるルーチンの INT コード・バージョンであり、現在ロードされているルーチンではありません。

- ・ ZWRITE コマンドを使用すると、指定したルーチンの INT コードを表示できます。

### ObjectScript

```
ZWRITE ^ROUTINE("MyRoutine")
```



この表示には、ルーチン MyRoutine の以下の ^ROUTINE 添え字が含まれます。

- ^ROUTINE("MyRoutine",0)="65309,36923.81262":このルーチンの INT コード・バージョンが最後にコンパイルされたローカル日時 (\$HOROLOG 形式)。このタイムスタンプは、再コンパイル前に MAC コードを変更していなくても更新されます。指定したルーチンが現在ロードされているルーチンである場合、現在ロードされているルーチンが変更されていれば、ZSAVE を発行すると、この値が更新されます。
- ^ROUTINE("MyRoutine",0,0)=8:このルーチンの INT コード・バージョンの行数。
- ^ROUTINE("MyRoutine",0,1)="Main":このルーチンの INT コード・バージョンの最初の行。この場合は、ラベル Main になります。
- ^ROUTINE("MyRoutine",0,2)=" WRITE "This is line 2",!":このルーチンの INT コード・バージョンの 2 行目。この場合は、インデントされた実行可能な ObjectScript コード行になります。追加のコード行も同じパターンに従います。

ZINSERT および ZREMOVE で現在のルーチンを変更した場合、ZSAVE を使用して変更を保存しない限り、^ROUTINE には反映されません。

ルーチンが MAC コード・ソースからロードされている場合、^ROUTINE の以下の添え字も表示されます。

- ^ROUTINE("MyRoutine","GENERATED")=1:INT コードが生成されたことを示します。
  - ^ROUTINE("MyRoutine","INC","%occStatus")="65301,60553":MAC バージョンに複数の #include ファイルが含まれる場合は、各 #include ファイルに対してこれらの添え字の 1 つが含まれ、#include ファイルが作成されたときのタイムスタンプを示します。
  - ^ROUTINE("MyRoutine","SIZE")=134:ソース・ファイルの INT コード・バージョンの文字数。
  - ^ROUTINE("MyRoutine","MAC")="65309,36920.45721":ルーチンの MAC バージョンが最後に保存されたローカル日時 (\$HOROLOG 形式)。このタイムスタンプは、MAC コードを変更して保存し、再コンパイルした場合にのみ更新されます。
- ・ 管理ポータルを使用して、^ROUTINE グローバルの内容を表示し、編集することができます。**[システムエクスプローラ]** > **[グローバル]** を選択し、左側の列のネームスペースのドロップダウン・リストから目的のネームスペースを選択します。
  - ・ **KILL** コマンドを使用して、ObjectScript INT (中間) コードを削除できます。

### ObjectScript

```
KILL ^ROUTINE("MyRoutine")
```

ルーチンの INT コードが利用できない場合 (KILL で削除されている場合)、ルーチンは引き続き実行できますが、現在ロードされているルーチンで INT コードを変更することはできません。ZLOAD、ZINSERT、および ZREMOVE ではエラーは発行されませんが、ZSAVE は <NO SOURCE> エラーで失敗します。

## ZLOAD と言語モード

ルーチンがロードされるとき、現在の言語モードはロードされたルーチンの言語モードに変換されます。呼び出されたルーチンの終了時、言語モードは呼び出し元のルーチンの言語モードに復元されます。しかし、ZLOAD でロードされたルーチンの終了時、言語モードは前の言語モードに復元されません。言語モードの確認と設定の詳細は、%SYSTEM.Process クラスの LanguageMode() メソッドを参照してください。

## 例

以下のターミナルの例では、排他ロックを確立してから、対応するルーチン MyRoutine をロードします。ソース・コードの最初の 10 行を表示して、ObjectScript コードの 1 行を行 2 として追加し、ソース・コードを再表示してから、変更を保存してロックを解放します。

```
USER>LOCK +^ROUTINE("MyRoutine")
USER>ZLOAD MyRoutine
USER>ZPRINT +1:+10
USER>ZINSERT " WRITE "Hello, World!"":+1
USER>ZPRINT +1:+11
USER>ZSAVE
USER>LOCK -^ROUTINE("MyRoutine")
```

以下のターミナルの例では、デバイス dev から最初のルーチンをロードします。

```
USER>OPEN dev
USER>USE dev
USER>ZLOAD
```

## 関連項目

- ・ [DO コマンド](#)
- ・ [OPEN コマンド](#)
- ・ [USE コマンド](#)
- ・ [XECUTE コマンド](#)
- ・ [ZREMOVE コマンド](#)
- ・ [ZSAVE コマンド](#)
- ・ [\\$XECUTE 関数](#)
- ・ [\\$ZNAME 特殊変数](#)

# ZPRINT (ObjectScript)

現在のデバイスに現在のルーチンからのコード行を表示します。

## 構文

```
ZPRINT:pc lineref1:lineref2
ZP:pc lineref1:lineref2
```

## 引数

引数	説明
pc	オプション - 後置条件式。
lineref1	オプション - 表示される行、もしくは表示される行の範囲の最初の行。リテラルとして指定します。ラベル名、数値オフセット (+n)、またはラベル名と数値オフセットのいずれかになります。省略した場合、現在のルーチン全体が表示されます。
:lineref2	オプション - 表示される行の範囲の最後の行。リテラルとして指定します。範囲を指定するには、lineref1 を指定する必要があります。

## 概要

ZPRINT コマンドは、現在ロードされているルーチンからコードの行を表示します。ルーチンをロードするには、[ZLOAD](#) を使用します。ZLOAD は、ルーチンの INT コード・バージョンをロードします。現在のルーチンの名前については、[\\$ZNAME](#) 特殊変数にアクセスします。

出力は、現在のデバイスに送られます。ターミナルから呼び出した場合、現在の出力デバイスは既定でターミナルになります。現在のデバイスは、USE コマンドで設定できます。現在のデバイスのデバイス ID については、[\\$IO](#) 特殊変数にアクセスします。

注釈 ZPRINT コマンドと [PRINT](#) コマンドは機能的に同一です。

ZPRINT は、ルーチンの INT コード・バージョンを表示します。INT コードではプリプロセッサ文はカウントされず、含まれることもありません。ルーチンの MAC バージョンの完全な空白行は、ソース・コード内にあっても、複数行コメント内にあっても、コンパイラによって削除されます。したがって、INT ルーチンには表示もカウントもされません。このため、ZPRINT では、MAC ルーチン内の以下の複数行コメントは、3 行ではなく 2 行で表示およびカウントされます。

### ObjectScript

```
/* This comment includes
   a blank line */
```

MAC コード内のコメント `#;`、`##;`、および `///` は、INT コードには表示されないことがあるため、行数やオフセットに影響が生じる場合があります。詳細は、“[MAC コードのルーチンおよびメソッドに使用するコメント](#)”を参照してください。

ZPRINT で、[編集ポインタ](#)を出力した行の末尾に設定します。例えば、ZPRINT の次に `ZINSERT " SET y=2"` を指定すると、行がルーチンの末尾に挿入されます。また、ZPRINT +1:+4 の次に `ZINSERT " SET y=2"` を指定すると、行が第 5 行として挿入されます。[\\$TEXT](#) 関数は、現在のルーチンから 1 行を出力しますが、編集ポインタは変更しません。

ZPRINT には、以下の 2 つの形式があります。

- ・ 引数なし
- ・ 引数付き

引数なしの ZPRINT は、現在ロードされているルーチンのすべてのコードの行を表示します。

引数付きの ZPRINT は、指定されたコードの行を表示します。ZPRINT lineref1 は、lineref1 によって指定された行を表示します。ZPRINT lineref1:lineref2 は、lineref1 から lineref2 まで (両端を含む) の行を表示します。

lineref 引数は、ルーチンの INT コード・バージョンを使用して、行および行オフセットをカウントします。ソース (MAC) バージョンに対応する行および行オフセットを正しくカウントするために、ルーチンの変更後、ZPRINT のルーチンを再コンパイルする必要があります。

[\\$TEXT](#) 関数を使用して、単一の INT コード行を返すことができます。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合に ZPRINT コマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、["コマンド後置条件式"](#) を参照してください。

### lineref1

印刷する行、または表示や印刷する行の範囲の最初の行。以下のいずれかの構文形式で指定できます。

- ・ `+offset` : `offset` は、現在のルーチンの行番号を指定する正の整数です。+1 はルーチンの最初の行であり、ラベル行とすることもできます。+0 は常に空の文字列を返します。
- ・ `label[+offset]` : `label` はルーチン内のラベルであり、`offset` はこのラベルからカウントする行数となります (ラベル自体の `offset` は 0 としてカウント)。offset オプションを省略、または `label+0` と指定すると、InterSystems IRIS はラベル行を印刷します。`label+1` は、ラベルの後の行を出力します。

ラベルは、31 文字よりも長くすることができますが、最初の 31 文字は一意である必要があります。ZPRINT は、指定された `label` の最初の 31 文字のみと一致します。ラベル名では大文字と小文字が区別され、Unicode 文字を含めることができます。

### lineref2

表示される範囲の最後の行。lineref1 と同じ方法で指定します。lineref2 を指定するには、lineref1 を指定する必要があります。lineref1 および lineref2 はコロン (:) 記号で区切ります。コロンと lineref2 の間に空欄を入れることはできません。

lineref2 に、行のシーケンス内で lineref1 よりも前のラベルまたはオフセットが指定されている場合、ZPRINT は lineref2 を無視して、lineref1 で指定される単一のコード行を表示します。

lineref2 に、存在しないラベルまたはオフセットが指定されている場合、ZPRINT は、lineref1 からルーチンの最後までを表示します。

## 例

以下のコード行が与えられた場合を示します。

### ObjectScript

```
AviationLetters
Abc
  WRITE "A is Abel",!
  WRITE "B is Baker",!
  WRITE "C is Charlie",!
Def WRITE "D is Delta",!
  WRITE "E is Epsilon",!
  /* Not sure about E */
  WRITE "F is Foxtrot",!
```

lineref 引数なしの ZPRINT は、コメント行を含めて、9 行すべてを表示します。

ZPRINT +0 は、空の文字列を表示します。

ZPRINT +1 は、ラベル `AviationLetters` を表示します。

ZPRINT +8 は、コメント行 `/* Not sure about E */` を表示します。

ZPRINT +10 は、空の文字列を表示します。

ZPRINT Def または ZPRINT Def+0 は、行 `Def WRITE "D is Delta",!` を表示します。これは、実行可能なコードも含むラベル行です。

ZPRINT Def+1 は、行 `WRITE "E is Epsilon",!` を表示します。

## 範囲の例

ZPRINT +0:+3 は、空の文字列を表示します。

ZPRINT +1:+3 は、最初の 3 行を表示します。

ZPRINT +3:+3 は、3 行目を表示します。

ZPRINT +3:+1 は、3 行目を表示します。lineref2 は無視されます。

ZPRINT +3:Abc+1 は、3 行目を表示します。lineref1 と lineref2 はともに同一行を指定しています。

ZPRINT +3:abc+1 は、3 行目からルーチンの最後までを表示します。行ラベルは大文字と小文字が区別されるので、範囲の端点が判別されません。

ZPRINT Abc+1:+4 は、3 行目と 4 行目を表示します。

ZPRINT Abc+1:Abc+2 は、3 行目と 4 行目を表示します。

ZPRINT Abc:Def は、2、3、4、5、および 6 行目を表示します。

ZPRINT Abc+1:Def は、3、4、5、および 6 行目を表示します。

ZPRINT Def:Abc は、行 `Def WRITE "D is Delta",!` を表示します。lineref2 はコード内で先に出現しているので、無視されます。

## 関連項目

- ・ [PRINT](#) コマンド
- ・ [ZINSERT](#) コマンド
- ・ [ZLOAD](#) コマンド
- ・ [ZREMOVE](#) コマンド
- ・ [ZSAVE](#) コマンド
- ・ [ZZPRINT](#) コマンド
- ・ [\\$TEXT](#) 関数
- ・ [\\$IO](#) 特殊変数
- ・ [\\$ZNAME](#) 特殊変数
- ・ [Comments](#)
- ・ [ラベル](#)
- ・ [スプール・デバイス](#)

## ZREMOVE (ObjectScript)

現在のルーチンから 1 行または行の範囲を削除するか、現在のルーチンをアンロードします。

### 構文

```
ZREMOVE:pc lineref1:lineref2 ,...
ZR:pc lineref1:lineref2 ,...
```

### 引数

引数	説明
pc	オプション - 後置条件式
lineref1	オプション - 削除される単一行、または行の範囲の最初の行の位置。リテラル (+5) または変数 (+a) で指定できます。lineref1 を省略すると、ZREMOVE はすべての現在のルーチンを削除します。
:lineref2	オプション - 削除される行範囲の最後の行の位置。

### 概要

ZREMOVE コマンドは、現在のプロセスの現在ロードされているルーチン进行操作します。現在のルーチンをロードするには、[ZLOAD](#) を使用します。ZLOAD は、ルーチンの INT コード・バージョンをロードします。INT コードではプリプロセッサ文はカウントされず、含まれることもありません。INT コードでは、ルーチンの MAC バージョンの完全な空白行はカウントされず、含まれることもありません。これは、ソース・コード内であっても、複数行コメント内であっても同じです。ルーチンがロードされると、そのルーチンは、すべてのネームスペースで、現在のプロセスに対して現在ロードされているルーチンになります。したがって、ルーチンのロード元のネームスペースだけではなく任意のネームスペースから、現在ロードされているルーチンの行の挿入または削除、ルーチンの表示、実行、アンロードを実行できます。

ターミナルから入力したり、XECUTE コマンドまたは \$XECUTE 関数で呼び出したりする場合は、ZREMOVE コマンドのみ使用できます。ルーチンの本文で ZREMOVE を指定すると、コンパイル・エラーが返されます。また、ルーチンから ZREMOVE を実行しようとしても、エラーが生成されます。

ZREMOVE には、以下の 2 つの形式があります。

- ・ [引数なし](#)の場合、現在のルーチンをアンロードします。
- ・ [引数付き](#)の場合、現在のルーチンから ObjectScript ソース・コードの 1 つ以上の行を削除します。

### 引数なし

引数なしの ZREMOVE は、現在ロードされているルーチンを削除 (アンロード) します。引数なしの ZREMOVE の後に [\\$ZNAME](#) を実行すると、現在のルーチンの名前ではなく空の文字列が返されます。また、[ZPRINT](#) で行は表示されません。ルーチンが削除されているため、[ZSAVE](#) を使用してルーチンを保存することはできません。保存しようすると、[<COMMAND>](#) エラーが返されます。

下記のターミナル・セッションでこの処理を示します。

```
USER>ZLOAD myroutine
USER>WRITE $ZNAME
myroutine
USER>ZREMOVE
USER>WRITE $ZNAME
USER>
```

引数なしの ZREMOVE では後置条件式を指定することができます。

引数付きの ZREMOVE では、現在のルーチンのすべての行を削除できますが、現在のルーチンそのものは削除されません。例えば、ZREMOVE +1:NonexistentLabel を実行すると現在のルーチンの行がすべて削除されますが、ZINSERT を使用して新しい行を挿入し、ZSAVE を使用してルーチンを保存できます。

## 引数付き

引数付きの ZREMOVE は、現在のルーチン内のコード行を削除します。ZREMOVE lineref1 は、指定された行を削除します。ZREMOVE lineref1:lineref2 は、最初の行参照で開始し、2 番目の行参照で終了する行の範囲を削除します。編集ポインタは、削除された行の直後に進みます。したがって、ZREMOVE lineref1 に続いて ZINSERT を実行すると、指定した行が置換されます。

ZREMOVE は、ObjectScript ソース・コードの複数の行（または複数の範囲）を削除できます。そのためには、一連の引数 lineref1 または lineref1:lineref2 をコンマで区切って組み合わせて指定します。指定したコード行またはコード行の範囲はそれぞれ個別の削除操作として指定の順序で削除されます。

ZPRINT を使用して、現在ロードされているルーチンの複数の行を表示できます。DO コマンドを使用して、現在のルーチンを実行できます。

ルーチンのローカル・コピーだけが影響を受け、ディスクに保存されたルーチンは影響を受けません。変更されたコードを保存するには、ZSAVE コマンドを使用してルーチンを保存する必要があります。

下記のターミナル・セッションでこの処理を示します。この例では、各行がその行を指定する文字列に変数を設定するダミー・ルーチン（myroutine）を使用します。

```
USER>ZLOAD myroutine

USER>ZPRINT +8
WRITE "this is line 8",!
USER>ZREMOVE +8

USER>PRINT +8
WRITE "this is line 9",!
USER>
```

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)”を参照してください。

### lineref1

削除する行、あるいは削除する行の範囲の最初の行。以下の形式のいずれかで指定することができます。

形式	説明
+offset	ルーチン内の行番号を指定します。1 から始まる正の整数。
label	ルーチン内のラベルを指定します。ZREMOVE label は、ラベル行だけを削除します。これには、その行のラベルに続くすべてのコードが含まれます。
label+offset	ラベル化されたセクション内で、ラベルとオフセットする行番号を指定します（ラベル行を行 1 としてカウントします）。

ラベルは、31 文字よりも長くすることができますが、最初の 31 文字は一意である必要があります。ZREMOVE は、指定された label の最初の 31 文字のみと一致します。ラベル名では大文字と小文字が区別され、Unicode 文字を含めることができます。



lineref1 を使用することで、削除する単一のコード行を指定できます。ルーチンの先頭からのオフセットとして (+lineref1)、または指定したラベルからのオフセットとして (label+lineref1) コード行を指定します。

- ・ ZREMOVE +7 : ルーチンの先頭から数えて 7 番目の行を削除します。
- ・ ZREMOVE +0 : 何も処理を行わず、エラーも生成しません。
- ・ ZREMOVE +999 : 999 がルーチン内の行数よりも大きい場合は、何も処理を行わず、エラーも生成しません。
- ・ ZREMOVE Test1 : ラベル行 Test1 を削除します。
- ・ ZREMOVE Test1+0 : ラベル行 Test1 を削除します。
- ・ ZREMOVE Test1+1 : ラベル行 Test1 の後の最初の行を削除します。
- ・ ZREMOVE Test1+999 : ラベル行 Test1 の後の 999 番目の行を削除します。この行は、別のラベル化されたモジュール内にある場合があります。999 がラベル Test1 からルーチンの最後までまでの行数よりも大きい場合は、何も処理を行わず、エラーも生成しません。

INT コード行には、ルーチンの MAC バージョンにあるラベル、コメント、および空白がすべて含まれます。例外として、コンパイラによって削除される、MAC ルーチンのすべて空白の行は表示もカウントもされません。複数行コメント内の空白行も削除されます。MAC コード内のコメント #;、##;、および /// は、INT コードには表示されないことがあるため、行数やオフセットに影響が生じる場合があります。詳細は、“[MAC コードのルーチンおよびメソッドに使用するコメント](#)”を参照してください。

## lineref2

削除する行の範囲の最後の行。lineref2 は、lineref1 で使用されるいずれかの形式で指定します。必ず、前にコロンの (:) を付ける必要があります。

行の範囲は +lineref1:lineref2 として指定します。ZREMOVE は、行の範囲 (lineref1 および lineref2 を含む) を削除します。lineref1 と lineref2 が同一行を指している場合、ZREMOVE は、その単一行を削除します。

lineref2 がルーチン・コード内で lineref1 よりも先に出現する場合は、何の処理も実行されず、エラーも発生しません。例えば、ZREMOVE +7:+2、ZREMOVE Test1+1:Test1、ZREMOVE Test2:Test1 では、何の処理も実行されません。

注釈 lineref2 でラベル名を指定する場合は注意が必要です。[ラベル名](#)では、大文字と小文字が区別されます。lineref2 に、ルーチンに存在しないラベル名が含まれる場合、ZREMOVE は、ルーチンの lineref1 から最後までの行の範囲を削除します。

## 例

このコマンドは、現在のルーチン内の 4 番目の行を削除します。

### ObjectScript

```
ZREMOVE +4
```

このコマンドは、ラベル Test1 の後にある 6 番目の行を削除します。Test1 が最初の行としてカウントされます。

### ObjectScript

```
ZREMOVE Test1+6
```

このコマンドは、現在のルーチンの 3 番目から 10 番目までの行を削除します。

### ObjectScript

```
ZREMOVE +3:+10
```

このコマンドは、現在のルーチン内で、ラベル行 Test1 からその直後の行までを削除します。

#### ObjectScript

```
ZREMOVE Test1:Test1+1
```

このコマンドは、現在のルーチン内で、ラベル Test1 からラベル Test2 までのすべての行を両方のラベルを含めて削除します。

#### ObjectScript

```
ZREMOVE Test1:Test2
```

## 関連項目

- ・ [PRINT](#) コマンド
- ・ [XECUTE](#) コマンド
- ・ [ZINSERT](#) コマンド
- ・ [ZLOAD](#) コマンド
- ・ [ZSAVE](#) コマンド
- ・ [\\$ZNAME](#) 特殊変数

## ZSAVE (ObjectScript)

現在のルーチンを保存します。

### 構文

```
ZSAVE:pc routine
ZS:pc routine
```

### 引数

引数	説明
pc	オプション - 後置条件式
routine	オプション - ルーチンの新しい名前。単純なりテラルとして指定します。有効な識別子である必要があります。ルーチン名は、大文字と小文字を区別します。routine の値は括弧で囲みません。キャレット (^) 接頭語やファイル・タイプ接頭語は付きません。変数または式を使用して指定することはできません。

### 概要

ZSAVE コマンドは、[現在のルーチン](#)を保存します。ZLOAD を使用してルーチンをロードします。次に、ZSAVE を使用して、ZINSERT コマンドおよび ZREMOVE コマンドでルーチンに加えた変更を保存します。

ターミナルから入力したり、XECUTE コマンドまたは \$XECUTE 関数で呼び出したりする場合は、ZSAVE コマンドのみ使用できます。このオペレーションはルーチンの実行に影響を与える可能性があるため、ルーチンの本体にコード化するべきではありません。ルーチン内で ZSAVE を指定すると、コンパイル・エラーが返されます。また、ルーチンから ZSAVE を実行しようとしても、エラーが生成されます。

ZSAVE では、[編集ポインタ](#)は移動しません。

ZLOAD を使用してルーチンを現在のルーチンとしてロードした場合、以下のように処理されます。

- ・ ZINSERT や ZREMOVE を使用して現在のルーチンを変更した後、ZSAVE でルーチンを保存した場合: ^rINDEX("MyRoutine","INT") グローバルと ^rINDEX("MyRoutine","OBJ") グローバルは最新のタイムスタンプと文字数に更新され、^ROUTINE("MyRoutine",0) グローバルは更新されます。
- ・ 現在のルーチンを変更せずに ZSAVE でルーチンを保存した場合: ^rINDEX("MyRoutine","OBJ") グローバルは更新されます。^rINDEX("MyRoutine","INT") グローバルまたは ^ROUTINE("MyRoutine",0) グローバルは変更されません。

詳細は、["INT コードと ROUTINE グローバル"](#) を参照してください。

ZSAVE には、以下の 2 つの形式があります。

- ・ [引数なし](#)
- ・ [引数付き](#)

### 引数なしの ZSAVE

引数なしの ZSAVE は、現在のルーチンを現在の名前で保存します。これは、ZLOAD で指定した名前か、以前に ZSAVE を使用して保存した名前です。ZSAVE は、ルーチンを現在のネームスペースに保存します。

以下の例では、USER ネームスペースからルーチンをロードし、ルーチンを変更してから、別のネームスペースに変更して ZSAVE を実行します。これらの操作の結果、USER ネームスペースと TESTNAMESPACE ネームスペースの両方に

MyRoutine という名前のルーチンが存在するようになります。TESTNAMESPACE の MyRoutine には、挿入されたコード行が含まれます。USER の MyRoutine には、挿入されたコード行は含まれません。

```
USER>ZLOAD MyRoutine

USER>ZPRINT +1:+4
WRITE "this is line 1",!
WRITE "this is line 2",!
WRITE "this is line 3",!
WRITE "this is line 4",!

USER>ZINSERT " WRITE 123,!":+3

USER>ZPRINT +1:+5
WRITE "this is line 1",!
WRITE "this is line 2",!
WRITE "this is line 3",!
WRITE 123,!
WRITE "this is line 4",!

USER>ZNSPACE "TESTNAMESPACE"

TESTNAMESPACE>ZPRINT +1:+5
WRITE "this is line 1",!
WRITE "this is line 2",!
WRITE "this is line 3",!
WRITE 123,!
WRITE "this is line 4",!

TESTNAMESPACE>ZSAVE
```

現在のルーチンにまだ名前がない場合、引数なしの ZSAVE は <COMMAND> エラーを生成します。

引数なしの ZSAVE コマンドでは後置条件式を指定することができます。

## 引数付きの ZSAVE

ZSAVE routine は、現在のルーチンを、指定された routine 名でディスクに保存します。また、指定された routine を現在のルーチンにします。例えば、MyRoutine という名前のルーチンをロードし、変更してから ZSAVE MyNewRoutine を使用して保存した場合、変更が含まれる MyNewRoutine が現在のルーチンになります。MyRoutine という名前のルーチンにはこれらの変更は含まれず、このルーチンは現在のルーチンとしてロードされていません。

ZSAVE routine は、現在のルーチンを現在のネームスペースに保存します。例えば、USER ネームスペースから MyRoutine という名前のルーチンをロードし、別のネームスペースに変更して ZSAVE MyNewRoutine を実行した場合、MyNewRoutine は、USER ネームスペースではなく変更先のネームスペースに保存されます。

XECUTE コマンドを使用して ZSAVE routine を呼び出す場合、システムは、Load フレームを作成して現在のルーチンを保持します。XECUTE コマンドが終了すると、InterSystems IRIS は、この Load フレームを使用して、現在のルーチンとして、XECUTE の前にルーチン名をリストアします。詳細は、以下の例を参照してください。

## ObjectScript

```
WRITE "Current routine name",!
WRITE "initial name: ",$ZNAME,!
SET x = "WRITE $ZNAME"
SET y = "ZSAVE mytest"
SET z = "WRITE "" changed to """,$ZNAME,!"
XECUTE x,y,z
WRITE "restored name: ",$ZNAME,!
```

引数なしの ZLOAD でロードしたルーチンに名前を付けるには、ZSAVE routine を使用します。

ZINSERT コマンドで作成した名前のないルーチンに名前を付けるには、ZSAVE routine を使用します。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### routine

ルーチンを保存する名前。routine は有効なルーチン名である必要があります。`$ZNAME("string",1)` 関数を使用すると、string が有効なルーチン名かどうかを判定できます。`$ZNAME` 特殊変数で、現在ロードされているルーチンの名前を判別できます。

通常、routine はルーチンの新しい名前ですが、現在のルーチン名にすることもできます。その名前のルーチンが既に現在のネームスペースに存在する場合、InterSystems IRIS はそれを上書きします。上書きするときに、Caché からの確認はないことに注意してください。ルーチン名の最初の 255 文字以内は一意である必要があります。220 文字を超える長さのルーチン名は避けてください。

routine を省略すると、システムは現在の名前でルーチンを保存します。現在の名前が存在しない場合、ZSAVE は `<COMMAND>` エラーを生成します。

## ZSAVE およびルーチンのリコンパイル

ソース・コードを変更するコマンドを発行した場合、ZSAVE はルーチンをリコンパイルして保存します。ルーチンのソース・コードが無効な場合、ZSAVE は失敗して `<NO SOURCE>` エラーが発生し、既存のオブジェクト・コードを置換しません。例えば、次のコマンドは %SS オブジェクト・コード・ルーチンをロードして、(存在しない) ソース・コードから行を削除しようとし、その後 `^test` グローバルに保存しようとしています。この処理は失敗して、`<NO SOURCE>` エラーが発生します。

```
ZLOAD %SS ZREMOVE +3 ZSAVE ^test
```

ソース・コードを変更するコマンドを発行しない場合、ZSAVE は指定された routine にオブジェクト・コードを保存します (明らかにリコンパイルは発生しません)。例えば、次のコマンドは %SS オブジェクト・コード・ルーチンをロードして、`^test` グローバルに保存します。この処理は成功します。

```
ZLOAD %SS ZSAVE ^test
```

## %Routine での ZSAVE

リモート・データセットに対する %routine に ZSAVE の実行を試みると、そのデータセットがプロセスの現在のデータセットであっても、`<PROTECT>` エラーが返されます。システム・ユーティリティのような変更不可のルーチンの名前には、その先頭にパーセント記号が使用されています。

## 並行 ZSAVE オペレーション

ネットワーク環境で ZSAVE を使用しているときは、2 つの異なるジョブがそれぞれのルーチンに同じ名前を割り当てて同時に保存する可能性があります。このオペレーションでは、一方のルーチンが他方を上書きし、予測できない結果を生じる可能性があります。こうした可能性がある場合は、ZSAVE オペレーションの前にルーチンに対して通知ロックを取得します。例えば、`LOCK ^ROUTINE("name")` を使用します。詳細は、“[LOCK](#)” コマンドを参照してください。ECP 経由でジョブを実行しているとき、ローカル・バッファの保護が他のクライアントには認識されないため、保存されたソースはこうした同時保存による障害を受けやすい状態になります。

## 例

以下のターミナル・セッションの例は、ZSAVE コマンドを実行して、現在ロードされているルーチンを保存します。

```
USER>DO ^myroutine
this is line 8
this is line 9
USER>ZLOAD myroutine

USER>PRINT +8
WRITE "this is line 8",!
USER>ZREMOVE +8

USER>PRINT +8
WRITE "this is line 9",!
USER>ZSAVE myroutine

USER>DO ^myroutine
this is line 9
USER>
```

## 関連項目

- ・ [XECUTE](#) コマンド
- ・ [ZLOAD](#) コマンド
- ・ [ZINSERT](#) コマンド
- ・ [ZREMOVE](#) コマンド
- ・ [\\$ZNAME](#) 特殊変数

## ZZPRINT (ObjectScript)

ルーチンからの 1 つ以上のソース・コード行を表示します。

### 構文

```
ZZPRINT:pc "entry+offset^routine":before:after
```

### 引数

引数	説明
pc	オプション — 後置条件式
entry	オプション — routine 内のエントリ (ラベル) の名前。ZZPRINT を省略した場合、routine の最初 (行 0) から開始されます。
offset	オプション — entry から (指定する場合)、または routine の最初から (entry を省略する場合) オフセットする行数を指定する整数。offset にはプラス記号 (+) の接頭語が必要です。通常、offset は正の整数です。offset は負の数になる場合があります。例えば、"subsect+-3^mytest" は、subsect の 3 行前の行を表示します。offset を省略した場合、entry の最初から表示されます。
routine	ソース・コード行を表示する起点となるルーチン。routine の名前は、常にキャレット (^) の接頭語が付きます。
before	オプション — 表示する "entry+offset^routine" で指定された行の前の行数を指定する正の整数。"entry+offset^routine" 行は含まれません。
after	オプション — 表示する "entry+offset^routine" で指定された行の後の行数を指定する正の整数。"entry+offset^routine" 行は含まれません。

### 説明

ZZPRINT コマンドは、指定された ObjectScript ルーチンからの 1 つ以上のソース・コード行を表示します。指定されたコード行、および指定されたソース・コード行の前後にある指定された行数を表示でき、コンテキスト内で確認できるようにします。entry 値は、表示の開始ポイントを指定します。ただし、ZZPRINT は、表示を指定された entry に制限しません。offset、before、および after で、それ以前またはそれ以降のエントリのソース・コード行を含めることができます。

表示される行には、すべてのラベル、コメント、およびスペースが含まれます (すべて空白の行は除きます)。空白行はコード内であっても複数行コメント内であっても、表示されることも、カウントされることもありません。このため、ZZPRINT では、以下の複数行コメントが 3 行ではなく、2 行で表示およびカウントされます。

#### ObjectScript

```
/* This comment includes
   a blank line */
```

ZZPRINT では、プリプロセッサ文がカウントされることも、表示されることもありません。

### コマンドの区切り文字およびスペースの要件

区切り引用符 (" ") が必要です。開始引用符は、コマンド名または後置条件式のちょうど 1 スペース後に配置する必要があります。開始引用符の後、または終了引用符の前後には任意の数のスペースを配置できます。



routine 名にはキャレット (^) の接頭語を付ける必要があります。offset を指定する場合、キャレットの接頭語の前に任意の数のスペースを配置できます。offset を指定しない場合、entry 名と routine 名のキャレットの接頭語との間にスペースを配置することはできません。

offset を指定するにはプラス記号 (+) が必要です。プラス記号は entry 名の直後に付ける必要があります (指定する場合)。整数の直前以外のプラス記号は、0 のオフセットを表します。例えば、キャレット (^) の接頭語、スペース、または数値でない文字の直前のプラス記号は、すべて 0 のオフセットを表します。整数の直前のプラス記号は、正のオフセットを表します。負の整数の直前のプラス記号は、負のオフセットを表します。

before の行数を指定するにはコロン (:) 区切り文字が必要です。after の行数を指定するには、before 値を指定するかどうかに関係なく、コロン区切り文字を指定する必要があります。これらのコロン区切り文字の前後には、任意の数のスペースを配置できます。

## 引数

### pc

オプションの後置条件式。InterSystems IRIS は、後置条件式が True (0 以外の数値に評価される) の場合にコマンドを実行します。InterSystems IRIS は、後置条件式が False (0 に評価される) の場合はコマンドを実行しません。詳細は、“[コマンド後置条件式](#)” を参照してください。

### entry

routine 内のエントリの名前。エントリ名は、大文字と小文字を区別します。routine が指定 entry を含まない場合、システムは <NOLABEL> エラーを生成します。

### offset

entry の最初から (指定する場合)、または routine の最初からオフセットする行数を指定する整数。offset を省略した場合、0 のオフセットと同じになります。entry からの 0 のオフセットは、エントリ・ラベル行自体 (行 1) になります。routine からの 0 のオフセットは、ルーチンの最初の前 (存在しない) 行 (行 0) になります。したがって、ルーチンの最初のコメント行を表示するには、1 のオフセットを指定する必要があります (" $+1^{\text{mytest}}$ ").

オフセット数には、すべてのソース・コード行が含まれます (プリプロセッサ文およびすべて空白の行は除きます)。これらは、表示されることも、カウントされることもありません。指定された (または暗黙的な) entry ポイントからカウントする際、routine の使用可能な行を超えるオフセットは、NULL 文字列を返します。

### routine

ソース・コード行を表示する起点となるルーチンの名前。ルーチン名は、大文字と小文字を区別します。指定されたルーチンが存在しない場合、システムは <NOROUTINE> エラーを生成します。

routine がこれを ZZPRINT できるようにするには、読み取り許可が必要です。この許可がない場合は、InterSystems IRIS は <PROTECT> エラーを生成します。

### before

指定された行の前の表示行数を指定する正の整数。これにより、直前の行のコンテキストでソース・コード行を表示できます。これは、指定されたプログラム行から逆方向にカウントした行数です。before の数には、" $\text{entry}+\text{offset}^{\text{routine}}$ " で指定されたプログラム行は含まれません。before の数が使用可能なコード行の数よりも大きい場合、ZZPRINT は、使用可能なコード行を表示します。エラーは発行されません。

### after

指定された行の後の表示行数を指定する正の整数。これにより、直後の行のコンテキストでソース・コード行を表示できます。これは、指定されたプログラム行から順方向にカウントした行数です。after の数には、" $\text{entry}+\text{offset}^{\text{routine}}$ " で指定されたプログラム行は含まれません。after の数が使用可能なコード行の数よりも大きい場合、ZZPRINT は、使用可能なコード行を表示します。エラーは発行されません。

after 値を指定する際、before 値は指定しても、省略してもかまいません。

- ・ 指定されたプログラム行、およびその行の後にある指定された行数を表示するには、after に対して正の整数を指定し、before に対して 0 の値を指定します。
- ・ 指定されたプログラム行の前にある routine のすべての行、およびその行の後にある指定された行数を表示するには、before 値を指定せずに after 値を指定します (代替のコロンのみ指定します)。before を指定せずに after を指定した場合、ZZPRINT は、routine の最初から、after の行数で指定された、指定プログラム行の後の場所まで表示します。
- ・ 指定されたプログラム行の前後にある指定された行数を表示するには、before および after に対して正の整数を指定します。

## 関連項目

- ・ [PRINT コマンド](#)
- ・ [ZPRINT コマンド](#)
- ・ [\\$TEXT 関数](#)
- ・ [Comments](#)
- ・ [ラベル](#)

# ObjectScript 関数

関数は演算を実行し、値を返します。この値は、演算結果になる場合と、演算の成功または失敗を表すインジケータになる場合があります。規約によって、値に変数を設定する InterSystems IRIS® データ・プラットフォーム関数は、変数を設定してから、演算が実行される前のその変数の値を返します。

このドキュメントでは、システム関数 (内部関数とも呼ばれます) について説明します。システム関数は、関数名の接頭語である \$ 文字と、名前の後の括弧によって識別されます。ドキュメントでは関数を参照する際、括弧は指定されていません。ユーザ指定関数 (外部関数とも呼ばれます) を作成することで、これらのシステム関数を補完できます。ユーザ指定関数は、接頭語 \$\$ で識別されます。

ObjectScript の特殊変数名も \$ 文字で始まりますが、特殊変数には括弧がありません。

ObjectScript 関数の全般的な詳細は、“[関数](#)” を参照してください。ユーザ独自の関数の定義方法の詳細は、“[ユーザ定義コード](#)” を参照してください。

システム関数を呼び出すには、以下の形式を使用します。

```
$name ( arguments )
```

引数	説明
name	関数名。先頭にドル記号 (\$) を付けます。ここでは、関数名はすべて大文字で表記されていますが、実際は大文字と小文字は区別されません。ほとんどの関数名には省略形があります。各関数の説明では、最初に完全名を使用した構文が示され、その下に略名を使用した構文が示されます (略名がある場合)。
arguments	<p>1 つ以上の値が関数に渡されます。関数引数は、常に括弧で囲まれ、関数名の後に記述します。関数に引数がない場合も括弧は必要です。</p> <p>複数の引数は、コンマで区切られています。引数は定位置にあり、関数で予期される引数と同じ順序で記述する必要があります。適切な数のコンマを提供することで、リスト内に引数が不足していることを示すことができます。引数リストの最後の引数を指定しない場合は、末尾のコンマは不要です。</p> <p>スペースは、引数リストの任意の場所で許可されます。name と開き括弧文字の間には、スペースは許可されません。</p>

各関数の説明には、実際に構文に使用される句読点文字のみが含まれます。構文中のオプション引数を明確にするなど、形式規約のための句読点は使用されていません。その種の情報は、説明の直後にある引数のテーブルで説明されます。

1 つの例外は省略記号 (...) です。コンマの後の省略記号は、コンマの前の引数 (または引数のグループ) が、コンマ区切りリストとして複数回繰り返し可能であることを示します。

プラットフォーム固有の関数は、関数をサポートするプラットフォーム名と共に示します。関数がプラットフォーム略語と共に示されていないときは、InterSystems IRIS® データ・プラットフォームのすべてのプラットフォームによってサポートされます。

## \$ASCII (ObjectScript)

文字を数値コードに変換します。

### 構文

```
$ASCII(expression,position)
$A(expression,position)
```

### 引数

引数	説明
expression	変換する文字
position	オプション - 文字列内での文字の位置。1 から数えます。既定値は 1 です。

### 概要

\$ASCII は expression で指定された単一の文字に対し文字コード値を返します。この文字は 8 ビット (拡張 ASCII) 文字あるいは 16 ビット (Unicode) 文字です。返り値は正の整数です。

expression 引数は、単一文字あるいは文字列に評価されます。expression が文字列に評価される場合、オプションの position 引数で、変換したい文字の位置を示す必要があります。

### 引数

#### expression

1 つ以上の文字を含む引用符付きの文字列として評価される式。この式には、変数名や数値、文字列リテラル、その他の有効な ObjectScript 式を指定できます。expression が複数の文字を含む文字列であるとき、position を使用して希望の文字を選択します。文字列に position が省略されると、\$ASCII は先頭の文字の数値コードを返します。

expression が NULL 文字列に評価される場合、\$ASCII は -1 を返します。例：\$ASCII(" ")。

二重引用符文字 (") の文字コード (34) を返すには、その文字を重複して使用し、引用符付きの文字列で囲んで指定します：\$ASCII(" ").

#### position

0 以外の正の整数を指定します。符号付き、もしくは符号なしでもかまいません。position では整数以外の数値を使用することも可能ですが、InterSystems IRIS は数値の小数部分は無視し、整数部分しか考慮しません。position がない場合、\$ASCII は expression の最初の文字の数値を返します。position の整数値が expression の文字数よりも多いか、または 1 より少ない場合、\$ASCII は -1 を返します。

### 例

以下の例は、文字 W の ASCII 数値 87 を返します。

#### ObjectScript

```
WRITE $ASCII("W")
```

以下の例は、Unicode 文字 "pi" と同等の数値 960 を返します。

#### ObjectScript

```
WRITE $ASCII($CHAR(959+1))
```

以下の例は、変数 Z の先頭にある文字の ASCII の数値と同等の 84 を返します。

#### ObjectScript

```
SET Z="TEST"
WRITE $ASCII(Z)
```

以下の例は、変数 Z の 3 番目にある文字の ASCII の数値と同等の 83 を返します。

#### ObjectScript

```
SET Z="TEST"
WRITE $ASCII(Z,3)
```

以下の例は、2 番目の引数が文字列にある文字数よりも大きい位置を指定したので、-1 を返します。

#### ObjectScript

```
SET Z="TEST"
WRITE $ASCII(Z,5)
```

以下の例では、FOR ループ内で \$ASCII を使用して、変数 x にあるすべての文字を ASCII の数値の同等値に変換します。\$ASCII リファレンスには、ループが実行されるたびに更新される position 引数が含まれます。position が x にある文字数よりも大きい数に達したとき、\$ASCII はループを終了する値 -1 を返します。

#### ObjectScript

```
SET x="abcdefghijklmnopqrstuvwxyz"
FOR i=1:1 {
    QUIT:$ASCII(x,i)=-1
    WRITE !,$ASCII(x,i)
}
QUIT
```

以下の例は、文字列 X の簡単なチェックサムを生成します。\$CHAR(CS) をこの文字列に連結すると、新しい文字列のチェックサムは常に 0 です。これにより、妥当性検証が簡単になります。

#### ObjectScript

```
CXSUM
SET x="Now is the time for all good men to come to the aid of their party"
SET CS=0
FOR i=1:1:$LENGTH(x) {
    SET CS=CS+$ASCII(x,i)
    WRITE !,"Checksum is:",CS
}
SET CS=128-CS#128
WRITE !,"Final checksum is:",CS
```

以下の例は、小文字、または大文字と小文字を組み合わせたものをすべて大文字に変換します。

## ObjectScript

```
ST
SET String="ThIs Is a MiXeDCaSe stRiNg"
WRITE !,"Input: ",String
SET Len=$LENGTH(String),Nstring=" "
FOR i=1:1:Len { DO CNVT }
QUIT
CNVT
SET Char=$EXTRACT(String,i),Asc=$ASCII(Char)
IF Asc>96,Asc<123 {
    SET Char=$CHAR(Asc-32)
    SET Nstring=Nstring_Char
}
ELSE {
    SET Nstring=Nstring_Char
}
WRITE !,"Output: ",Nstring
QUIT
```

## Unicode サポート

\$ASCII 関数は、8 ビット文字と 16 ビット文字の両方をサポートします。8 ビット文字は 0 から 255 の数値を返します。16 ビット (Unicode) 文字は 65535 までの数値コードを返します。

文字の Unicode 値は通常、0 ～ 9 と A ～ F (それぞれ 10 ～ 15) の 16 進数の 4 桁の数字で表されます。しかし、ObjectScript 言語の基本関数は、通常 16 進数ではなく、10 進法の 10 進数値の ASCII 数値による文字を識別します。

したがって、Unicode 標準が推奨する 16 進数値の代わりに、\$ASCII 関数は入力文字の 10 進数の Unicode 値を返すことによって、Unicode コード化をサポートします。この方法で、関数は Unicode をサポートしている間でも、後方互換性のままになります。10 進数から 16 進数への変換には、\$ZHEX 関数を使用します。

InterSystems IRIS Unicode サポートの詳細は、“[Unicode](#)” を参照してください。

## サロゲート・ペア

\$ASCII は、サロゲート・ペアを認識しません。サロゲート・ペアは、一部の中国語の文字を表示したり、日本語の JIS2004 標準をサポートするために使用されます。\$WISWIDE 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。\$WASCII 関数は、サロゲート・ペアを認識して、正しく解析します。\$ASCII と \$WASCII は、それ以外は同一です。ただし、\$ASCII は通常 \$WASCII より高速なため、サロゲート・ペアが出現しない場合は常に \$ASCII が推奨されます。

注釈     \$WASCII を \$ZWASCII と混同しないでください。\$ZWASCII は常にペアの文字を解析します。

## 関連する関数

\$CHAR 関数は \$ASCII と逆の機能を持っています。この関数を使用して、整数コードを文字に変換することができます。

\$ASCII は、単一の文字を整数に変換します。16 ビット (wide) 文字の文字列を整数に変換するには、\$ZWASCII を使用します。32 ビット (long) 文字の文字列を整数に変換するには、\$ZLASCII を使用します。64 ビット (quad) 文字の文字列を整数に変換するには、\$ZQASCII を使用します。64 ビット文字の文字列を IEEE 浮動小数点数 (\$DOUBLE データ型)に変換するには、\$ZDASCII を使用します。

## 関連項目

- ・ [READ](#) コマンド
- ・ [WRITE](#) コマンド
- ・ [\\$CHAR](#) 関数
- ・ [\\$WASCII](#) 関数
- ・ [\\$WISWIDE](#) 関数
- ・ [\\$ZLASCII](#) 関数

- ・ [\\$ZWASCII](#) 関数
- ・ [\\$ZQASCII](#) 関数
- ・ [\\$ZDASCII](#) 関数



## \$BIT (ObjectScript)

\$BIT は、ビット文字列内の指定された位置のビット値を返すか、または設定します。

### 構文

```
$BIT(bitstring,position)
```

```
SET $BIT(bitstring,position) = value
```

### 引数

引数	説明
bitstring	ビット文字列として評価される式。  \$BIT の場合、bitstring にはビット文字列に解決される任意の式を指定できます。これには、任意のタイプの変数、\$FACTOR、ユーザ定義関数、および <code>oref.prop</code> 、 <code>..prop</code> 、 <code>i%prop</code> <a href="#">インスタンス変数プロパティ参照</a> が含まれます。  SET \$BIT の場合、bitstring には <a href="#">i%Prop()</a> プロパティ・ <a href="#">インスタンス変数</a> を含む任意のタイプの変数を指定できます。
position	bitstring 内のビット位置。リテラルまたは正の整数に評価される式。ビット位置は 1 からカウントされます。
value	position で設定するビット値。リテラルまたは整数 0 か 1 に評価される式。

### 概要

\$BIT は、圧縮ビット文字列からビット値を返すために使用されます。\$BIT (bitstring,position) は、与えられたビット文字列式 bitstring 内で、指定された位置 position のビット値 (0 または 1) を返します。position に対して値が定義されていない場合、\$BIT はその position に対して 0 を返します。position は 1 からカウントされ、position が 1 未満 (0 または負の数) であるか、またはビット文字列の長さよりも長い場合、関数は 0 を返します。

bitstring が未定義の変数であるか、NULL 文字列 ("" ) である場合、\$BIT 関数は 0 を返します。bitstring が有効なビット文字列ではない場合、<INVALID BIT STRING> エラーが発生します。

\$BIT 関数およびその他のビット文字列関数の概説は、"[ビット文字列関数の概要](#)" を参照してください。

### SET \$BIT

SET \$BIT は、bitstring の圧縮ビット文字列で指定されたビット値を設定するために使用されます。bitstring が定義されていない場合、SET \$BIT は、bitstring 変数を圧縮ビット文字列として定義し、指定されたビット値を設定します。

SET \$BIT(bitstring,position) = value は bitstring によって指定されたビット文字列に対してアトミック・ビット・セットを実行します。value が 1 の場合、位置 position のビットは 1 に設定されます。value が 0 の場合、ビットはクリアされます (0 に設定)。整数の value 値 0 または 1 のみを使用してください。InterSystems IRIS は、"true" や "false" などの非数値をすべて 0 に変換します。

ビットの position は 1 からカウントされます。bitstring が、指定された position よりも短い場合、ビット文字列の指定された位置に、InterSystems IRIS により 0 ビットが付け加えられます。position に 0 を指定すると、システムによって <VALUE OUT OF RANGE> エラーが生成されます。

bitstring 変数は、未定義の変数、ビット文字列値に既に設定されている変数、または空の文字列 ("" ) に設定されている変数のいずれかにする必要があります。ビット文字列以外の値に既に設定されている変数で SET \$BIT を使用しようとすると、<INVALID BIT STRING> エラーが発生します。

SET \$BIT bitstring 引数は `oref.property` または `.. property` 構文をサポートしていません。

SET \$BIT bitstring 引数は、ローカル (非継承) プロパティとスーパー・クラスから継承されたプロパティの両方の `%property インスタンス変数` 構文をサポートします。継承されたプロパティを既存のコードに設定しようとして <FUNCTION> エラーが生成される場合、ルーチンをリコンパイルするとこのエラーが解消され、継承されたプロパティを設定できるようになります。

## ビット文字列の表示

例に示されているように、WRITE を使用することで、ビット文字列の各ビットの内容を \$BIT の返り値として表示できます。

InterSystems IRIS には、変数の内容を表示するためのコマンドがいくつか用意されています。ただし、\$BIT のビット文字列は圧縮バイナリ文字列であるため、WRITE は実用的な値を表示しません。ZZDUMP は、16 進表現の圧縮バイナリ文字列を表示しますが、これもほとんどの目的において実用的な値ではありません。

ZWRITE および ZZWRITE は、10 進表現の圧縮バイナリ文字列を \$ZWCHAR (\$zwc) 2 バイト (ワイド) 文字として表示します。ただし、コンマ区切りリストとして、左から右の順に、圧縮されていない “1” ビットをリストするコメントも表示します。3 つ以上の連続した “1” ビットがある場合は、それらを 2 ドット構文 (n..m) を使用して範囲 (n と m も範囲に含む) としてリストします。例えば、ビット文字列 [1,0,1,1,1,1,0,1] は、`/*$bit(1,3..6,8)*/` と表示されます。ビット文字列 [1,1,1,1,1,1,1,1] は、`/*$bit(1..8)*/` と表示されます。ビット文字列 [0,0,0,0,0,0,0,0] は、`/*$bit()*/` と表示されます。

\$DATA は、[0,0,0,0,0,0,0,0] のようなすべてがゼロのビット文字列を含め、圧縮バイナリ文字列変数について 1 を返します。\$GET は、その値に関わらず、圧縮バイナリ文字列について空の文字列を返します。\$GET は、未定義の変数についても空の文字列を返します。

## 例

以下の例で、設定されていないビットは常に 0 の値を持つことに注意してください。

次の例では、SET \$BIT を使用して、圧縮ビット文字列を作成します。その後、\$BIT を繰り返し呼び出して、ビット文字列のビットを表示します。

### ObjectScript

```
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 0
SET $BIT(a,5) = 0
SET $BIT(a,6) = 1
SET $BIT(a,7) = 1
SET $BIT(a,8) = 0
// Test single bits within the bitstring
WRITE "bit #2 value: ", $BIT(a,2), !
WRITE "bit #7 value: ", $BIT(a,7), !
WRITE "bit #8 value: ", $BIT(a,8), !
WRITE "bit #13 value: ", $BIT(a,13), !
// Write the bitstring
WRITE "bitstring value: "
FOR x=1:1:8 {WRITE $BIT(a,x) }
WRITE !, "compressed bitstring: "
ZZDUMP a
```

InterSystems IRIS は、ビット文字列の指定した位置に 0 ビットを付け加えるため、以下の例では、まったく同じビット文字列データ値が返されています。ただし、#8 ビットが定義されていないため、圧縮ビット文字列 a は圧縮ビット文字列 b と同一ではないことに注意してください。

## ObjectScript

```

SET $BIT(b,3) = 1
SET $BIT(b,6) = 1
SET $BIT(b,7) = 1
// Test single bits within the bitstring
WRITE "bit #2 value: ", $BIT(b,2), !
WRITE "bit #7 value: ", $BIT(b,7), !
WRITE "bit #8 value: ", $BIT(b,8), !
WRITE "bit #13 value: ", $BIT(b,13), !
// Write the bitstring
WRITE "bitstring value: "
FOR x=1:1:8 {WRITE $BIT(b,x) }
WRITE !!, "compressed bitstring: "
ZZDUMP b

```

このため、割り当て済みの値が 0 であっても、ビット文字列で最大定義済みビットを常に明示的に設定するプログラミング方法が推奨されます。

以下の例に示すように、**FOR expr** コンマ区切りリストを使用して、複数のビットを設定できます。

## ObjectScript

```

FOR i=3,6,7 { SET $BIT(b,i) = 1 }
// Test single bits within the bitstring
WRITE "bit #2 value: ", $BIT(b,2), !
WRITE "bit #7 value: ", $BIT(b,7), !
WRITE "bit #8 value: ", $BIT(b,8), !
WRITE "bit #13 value: ", $BIT(b,13), !
// Write the bitstring
WRITE "bitstring value: "
FOR x=1:1:8 {WRITE $BIT(b,x) }
WRITE !!, "compressed bitstring: "
ZZDUMP b

```

次の例では、\$BIT を連続して呼び出すことにより、**\$FACTOR** によって生成されたビット文字列のビットが返されます。

## ObjectScript

```
FOR i=1:1:32 {WRITE $BIT($FACTOR(2*31-1),i) }
```

次の例は、ランダムな 16 ビットのビット文字列を返します。

## ObjectScript

```

SET x=$RANDOM(65536)
FOR i=1:1:16 {WRITE $BIT($FACTOR(x),i) }

```

## ビット文字列関数の概要

ビット文字列関数は、コード化されたビット単位のデータを計算します。ビット文字列は、すべての ObjectScript コマンドや関数と共に使用できますが、通常ビット関数のコンテキスト内でのみ意味を持ちます。

\$BIT ビット文字列関数は、アトミック演算を実行します。したがって、ビット文字列演算を実行するときにロックは必要ありません。

\$BIT ビット文字列関数は、ビット文字列の内部圧縮を実行します。したがって、ビット文字列の実際のデータの長さと、その物理的なスペース割り当てとは、異なる可能性もあります。\$BIT ビット文字列関数はビット文字列のデータの長さを使用します。ほとんどの場合、物理的なスペース割り当てはユーザから見えないところにあります。ビット文字列圧縮は、\$BIT 関数のユーザからは見えないところで行われます。

しかし、圧縮 2 進数の表現は各ビット文字列ごとに最適化されるため、(別々に作成された) “同じ” 文字列が同じ内部表現を持つとは限りません。InterSystems IRIS は、スペース・ビット文字列と非スペース・ビット文字列の両方に対する最適化のために、4 つの別個のビット文字列内部表現から選択します。したがって、各ビットにおけるマッチング演算で期待通りの結果が得られても、文字列全体を比較して同様の結果が得られるとは限りません。

\$BIT ビット文字列関数は、InterSystems IRIS 用に最大ビット文字列長 262,104 ビット (32763 × 8) をサポートしています (特定の従来のインターシステムズ製品とは異なり、InterSystems IRIS では、ビット文字列長よりも長いビットを操作してもエラーは発生しません)。ただし、パフォーマンス上の理由から、長いビット文字列を 65,280 ビット未満のチャンクに分割することを強くお勧めします。これが、単一の 8KB データベース・ブロック内に収めることができる最大ビット数です。

ビット文字列のビットは左から右に順番に並び、一番左のビットは位置 1 となります。すべてのビット文字列は左から右に比較されます。

例では、ビット文字列はコンマで区切られたビットを使用し、一致する中括弧 ({...}) 内に示されています。例えば、1 ビット 4 つのビット列は {1,1,1,1} になり、最下位ビットが右になります。

すべてのビット文字列関数で、bitstring という変数はビット文字列であり、値、変数、または式として指定できます。

## 関連項目

- ・ [\\$BITCOUNT](#) 関数
- ・ [\\$BITFIND](#) 関数
- ・ [\\$BITLOGIC](#) 関数
- ・ [\\$FACTOR](#) 関数
- ・ [\\$ZBOOLEAN](#) 関数

## \$BITCOUNT (ObjectScript)

bitstring のビット数を返します。

### 構文

```
$BITCOUNT(bitstring,bitvalue)
```

### 引数

引数	説明
bitstring	ビット文字列として評価される式。任意のタイプの変数、\$FACTOR、ユーザ定義関数、または oref.prop、..prop、i%prop ( <a href="#">インスタンス変数</a> ) プロパティ参照を指定できます。
bitvalue	オプション - ビット文字列内をカウントする値 (0 または 1)

### 概要

\$BITCOUNT 関数は、ビット文字列内のビットの数をカウントします。ビット文字列は、ビットの連続としてシステムに解釈される、エンコードされた文字列です。\$BIT や \$BITLOGIC を使用して、ビット文字列を作成することができます。

\$BITCOUNT(bitstring) は、bitstring 内のビット数を返します。

\$BITCOUNT(bitstring, bitvalue) は、bitstring 内の bitvalue (0 または 1) タイプのビット数を返します。

最大のビット文字列長は 262,104 ビット (32763 x 8) です。

InterSystems IRIS のエンコードされたビット文字列ではない bitstring 値を指定すると、<INVALID BIT STRING> エラーが生成されます。詳細は、"[ビット文字列関数の概要](#)" を参照してください。

### 例

bitstring = [0,0,1,1,0] の場合、\$BITCOUNT(bitstring) の結果は 5 になります。

#### ObjectScript

```
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 1
SET $BIT(a,5) = 0
WRITE !,$BITCOUNT(a)
```

bitstring= [0,0,1,1,0] の場合、\$BITCOUNT(bitstring,0) の結果は 3 になります。

#### ObjectScript

```
SET $BIT(a,1) = 0
SET $BIT(a,2) = 0
SET $BIT(a,3) = 1
SET $BIT(a,4) = 1
SET $BIT(a,5) = 0
WRITE !,"number of zero bits:",$BITCOUNT(a,0)
WRITE !,"number of one bits: ",$BITCOUNT(a,1)
```

次の例は、\$FACTOR によって生成されたランダムな 16 ビットのビット文字列に含まれ、値が 1 であるビットの数を返します。

## ObjectScript

```
SET x=$RANDOM(65536)
FOR i=1:1:16 {WRITE $BIT($FACTOR(x),i) }
WRITE !,"Number of 1 bits=", $BITCOUNT($FACTOR(x),1)
```

## 関連項目

- ・ [\\$BIT](#) 関数
- ・ [\\$BITFIND](#) 関数
- ・ [\\$BITLOGIC](#) 関数
- ・ [\\$FACTOR](#) 関数
- ・ [\\$ZBOOLEAN](#) 関数

## \$BITFIND (ObjectScript)

ビット文字列内の指定されたビット値の位置を返します。

### 構文

```
$BITFIND(bitstring,bitvalue,position,direction)
```

### 引数

引数	説明
bitstring	ビット文字列として評価される式。bitstring が未定義の変数の場合、\$BITFIND は 0 を返します。
bitvalue	ビット文字列内を検索する値 (0 または 1)。
position	<p>オプション – 検索が開始されるビット位置。正の整数で指定されます。検索方向が順方向でも逆方向でも、ビット位置はビット文字列の先頭を起点に 1 からカウントされます。検索はこの位置から開始します。</p> <p>順方向に検索 (direction = 1 または未指定のまま) するときに、指定された位置がない場合、または位置の値が 0 の場合は、位置 1 が指定されているものと見なされます。</p> <p>逆方向に検索 (direction = -1) するときに、指定された位置がない場合、または位置の値が 0 の場合は、ビット文字列の最後の位置が指定されているものと見なされます。</p>
direction	<p>オプション – 方向フラグ。使用可能な値は 1 と -1 です。</p> <ul style="list-style-type: none"> <li>1 を指定すると、ビット文字列の先頭 (または position で指定された位置) から順方向 (左から右) に検索されます (既定の方向)。</li> <li>-1 はビット文字列の最後 (または position) から逆方向に検索します。</li> </ul>

### 概要

\$BITFIND(bitstring,bitvalue) は、ビット文字列 bitstring の中を左から右へ検索し、指定された bitvalue (0 または 1) が最初に見つかった位置を返します。ビット位置は、ビット文字列の先頭を起点に、1 からカウントされます。

\$BITFIND(bitstring,bitvalue,position) は、bitstring の position 以降の部分で、指定された bitvalue が最初に見つかった位置を返します。

\$BITFIND(bitstring,bitvalue,position,direction) は、ビット文字列の中を検索する方向を指定します。direction = 1 を指定するか、何も指定しないと、\$BITFIND による検索方向は順方向 (左から右) になります。direction = -1 を指定すると、\$BITFIND による検索方向は逆方向 (右から左) になります。検索が順方向か逆方向かに関係なく、ビット位置はビット文字列の先頭を起点に 1 からカウントされます。

希望のビット値が見つからない場合、または position がビット文字列内のビット数よりも大きい場合は、0 の値を返します。指定された bitstring が未定義の変数である場合、返り値は 0 です。指定された bitstring が有効なビット文字列ではない場合、<INVALID BIT STRING> エラーが発行されます。

“[ビット文字列関数の概要](#)” も参照してください。



## 例

20 個の 1 と 0 で構成するビット文字列を作成します。

```
set bitvalues = "00110001101010000111"
for i = 1:1:$length(bitvalues) { set $bit(bitstring,i) = $extract(bitvalues,i)}
```

値が 0 である最初のビット位置と値が 1 である最初のビット位置を返します。position または direction の値を指定しないと、\$BITFIND による検索は位置 1 から始まります。

```
write $BITFIND(bitstring,0) // returns 1
write $BITFIND(bitstring,1) // returns 3
```

位置 5 以降の範囲で、値が 0 である最初のビット位置と値が 1 である最初のビット位置を返します。\$BITFIND は位置 5 以降の範囲を検索します。

```
set position = 5
write $BITFIND(bitstring,0,position) // returns 5
write $BITFIND(bitstring,1,position) // returns 8
```

位置 5 以降の範囲で、値が 0 である最初のビット位置と値が 1 である最初のビット位置を返します。backward 引数には -1 を指定します。\$BITFIND は、位置 5 から逆方向に右から左へ検索を進めます。

```
write $BITFIND(bitstring,0,position,-1) // returns 5
write $BITFIND(bitstring,1,position,-1) // returns 4
```

ビット文字列の範囲外を検索の開始位置として、値が 1 である最初のビット位置を返します。

```
write $BITFIND(bitstring,1,-100) // Search from -100 forward: returns 3
write $BITFIND(bitstring,1,-100,-1) // Search from -100 backward: returns 0
write $BITFIND(bitstring,1,100) // Search from 100 forward: returns 0
write $BITFIND(bitstring,1,100,-1) // Search from 100 backward: returns 20
```

ビット文字列の中で値が 0 であるすべてのビット位置と値が 1 であるすべてのビット位置を返します。

### ObjectScript

```
set position = 0
write !,"Bit positions with value 1: "
for { set position=$BITFIND(bitstring,1,position+1) quit:'position' write position,", " }
write !,"Bit positions with value 0: "
for { set position=$BITFIND(bitstring,0,position+1) quit:'position' write position,", " }
```

## 関連項目

- [\\$BIT](#) 関数
- [\\$BITCOUNT](#) 関数
- [\\$BITLOGIC](#) 関数
- [\\$FACTOR](#) 関数
- [\\$ZBOOLEAN](#) 関数

## \$BITLOGIC (ObjectScript)

ビット文字列でビット演算を実行します。

### 構文

```
$BITLOGIC(bitstring_expression, length)
```

### 引数

引数	説明
bitstring_expression	1 つ以上のビット文字列変数と論理演算子 &、 、^、および ~ で構成した論理式。ローカル変数、プロセス・プライベート・グローバル、グローバル、オブジェクト・プロパティ、または定数 "" として指定されます。NULL 文字列 ("") のビット文字列長は 0 です。ビット文字列を返す関数 (\$FACTOR など) を使用してビット文字列を指定することはできません。
length	オプション - 取得したビット文字列の長さをビットで表したものの。length が指定されていない場合、既定は bitstring_expression の最長のビット文字列長になります。

### 概要

\$BITLOGIC は、1 つまたは複数のビット文字列値に対し、bitstring\_expression で指定されたようにビット演算を実行し、結果のビット文字列を返します。

ビット文字列は、一連のビットとして解釈されるコード化 (圧縮された) 文字列です。\$BIT、\$FACTOR、または \$BITLOGIC を使用して作成されたビット文字列、または NULL 文字列 ("") だけが、\$BITLOGIC 関数に提供されます。一般的に、ビット文字列はインデックス演算に使用されます。詳細は、“\$BIT” の “[ビット文字列関数の概要](#)” を参照してください。

\$BITLOGIC と \$ZBOOLEAN が使用するデータ形式は異なります。一方の結果を、もう一方の入力として使用することはできません。

### ビット文字列の最適化

基本的な \$BITLOGIC 操作は \$BITLOGIC(a) です。この操作は何も行わないように見えます。ビット文字列 a が入力され、同じビット文字列 a が出力されます。しかし、\$BITLOGIC は、いくつかの圧縮アルゴリズムから選択することによってこれが最適化するビット文字列圧縮を実行します。したがって、ビット文字列 a は、このビット文字列が作成されてから実質的な変更が加えられる場合、\$BITLOGIC 経由でこのビット文字列を渡すことにより、ビット文字列の最適化を再度行うことができます。詳細は “[\\$BIT](#)” を参照してください。

例えば、多数の削除操作の後、インデックス・ビット文字列は、全体的にまたは大部分がゼロで構成されているスパース・ビット文字列になる場合があります。\$BITLOGIC 経由でこのインデックス・ビット文字列を渡すことにより、パフォーマンスをかなり向上させることができます。

### ビット文字列の論理演算子

\$BITLOGIC は以下のテーブルのビット文字列演算子のみを評価します。

演算子	意味
&	AND
	OR
^	XOR (排他的 OR)
~	NOT (1 の補数)

bitstring\_expression は現在の最大 31 のビット文字列まで、単一のビット文字列 (~A)、2 つのビット文字列 (A&B)、または 2 つ以上のビット文字列 (A&B|C) を含めることができます。評価は左から右の順に行われます。論理演算子は標準的な ObjectScript の演算子の順序に従って、bitstring\_expression 内で括弧によってグループ化されることもあります。\$BITLOGIC 内で使用されている変数が未定義の場合は、NULL 文字列 ("") として処理されます。

\$BITLOGIC は NULL 文字列を、すべてのビットが 0 に設定されている、長さ不定のビット文字列として処理します。

**注釈** \$BITLOGIC に 2 つ以上のビット文字列オペランドを指定する場合は、中間結果を保持するために一時的なビット文字列を作成する必要があります。異常な状況下では (多くのビット文字列や極端に大きいビット文字列など)、このような一時的なビット文字列を保持するために割り当てられている領域が使い尽くされる場合があります。ビット文字列の組み合わせの処理ではこのような制限はないので、大きなビット文字列の処理を行うことができます。

NOT (~) 演算子は単項演算子 (例えば ~A) として使用されたり、他の演算子との組み合わせ (例えば A&~B) で使用されます。この演算子は、1 をすべて 0 に、0 をすべて 1 に変換する文字列上の補数演算を実行します。複数の NOT 演算子も使用可能です (例えば ~~~A)。

## length 引数

length が指定されていない場合、既定は bitstring\_expression の最長のビット文字列長になります。

length が指定されている場合は、結果のビット文字列の論理長を指定します。

- length が bitstring\_expression 内のビット文字列より 1 以上大きい場合、ビット文字列論理演算が実行される前に、これらのビット文字列は、その文字列長までゼロで埋められます。
- length が 1 つ以上の bitstring\_expression 内のビット文字列よりも小さい場合、ビット文字列論理演算が実行される前に、これらのビット文字列はその文字列まで切り捨てられます。
- length がゼロの場合、ビット文字列長 0 (NULL 文字列) を返します。

## 例

以下は、複数の単純なビット文字列を作成し、その文字列で \$BITLOGIC を使用する例を示しています。

### ObjectScript

```
// Set a to [1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
// Set b to [0,1]
SET $BIT(b,1) = 0
SET $BIT(b,2) = 1
WRITE !,"bitstring a=", $BIT(a,1), $BIT(a,2)
WRITE !,"bitstring b=", $BIT(b,1), $BIT(b,2)
SET c = $BITLOGIC(~b)
WRITE !,"The one's complement of b=", $BIT(c,1), $BIT(c,2)
// Find the intersection (AND) of a and b
SET c = $BITLOGIC(a&b) // c should be [0,1]
WRITE !,"The AND of a and b=", $BIT(c,1), $BIT(c,2)
SET c = $BITLOGIC(a&~b) // c should be [1,0]
WRITE !,"The AND of a and ~b=", $BIT(c,1), $BIT(c,2)
// Find the union (OR) of a and b
SET c = $BITLOGIC(a|b) // c should be [1,1]
WRITE !,"The OR of a and b=", $BIT(c,1), $BIT(c,2)
```

```
SET c = $BITLOGIC(a^b)    // c should be [1,0]
WRITE !,"The XOR of a and b=", $BIT(c,1), $BIT(c,2)
QUIT
```

以下の例は、入力ビット文字列より大きい length を指定した場合の結果を示します。ビット文字列論理演算が実行される前に、文字列がゼロで埋められます。

### ObjectScript

```
// Set a to [1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
WRITE !,"bitstring a=", $BIT(a,1), $BIT(a,2)
SET c = $BITLOGIC(~a,7)
WRITE !,"~a (length 7)="
WRITE $BIT(c,1), $BIT(c,2), $BIT(c,3), $BIT(c,4)
WRITE $BIT(c,5), $BIT(c,6), $BIT(c,7), $BIT(c,8)
```

ここでは、11 の 1 の補数 (~) は 00111111 です。ビット 3 から 7 は、~ 演算が実行される前は、ゼロに設定されていました。さらに、この例では 8 番目のビットが表示されます。このビットは指定した文字列長を超えているので、\$BITLOGIC 操作の影響を受けません。0 として表示されます。

以下の例は、入力ビット文字列より小さい length を指定した場合の結果を示します。ビット文字列論理演算が実行される前に、ビット文字列は指定した長さに切り捨てられます。指定した長さを超えるビットはすべて既定で 0 になります。

### ObjectScript

```
// Set a to [1,1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
SET $BIT(a,3) = 1
WRITE !,"bitstring a=", $BIT(a,1), $BIT(a,2), $BIT(a,3)
SET c = $BITLOGIC(a,2)
WRITE !," a (length 2)="
WRITE $BIT(c,1), $BIT(c,2), $BIT(c,3), $BIT(c,4)
SET c = $BITLOGIC(~a,2)
WRITE !,"~a (length 2)="
WRITE $BIT(c,1), $BIT(c,2), $BIT(c,3), $BIT(c,4)
```

以下の例では、length が指定されていない場合、既定は最長のビット文字列長になることを示しています。ビット文字列論理演算が実行される前に、短いビット文字列はゼロで埋められます。

### ObjectScript

```
// Set a to [1,1,1]
SET $BIT(a,1) = 1
SET $BIT(a,2) = 1
SET $BIT(a,3) = 1
// Set b to [1,1]
SET $BIT(b,1) = 1
SET $BIT(b,2) = 1
SET c = $BITLOGIC(a&~b)
WRITE !," a&~b="
WRITE $BIT(c,1), $BIT(c,2), $BIT(c,3)
SET c = $BITLOGIC(a&~b,3)
WRITE !," a&~b,3="
WRITE $BIT(c,1), $BIT(c,2), $BIT(c,3)
```

ここでは、2 つの \$BITLOGIC 操作 (length 引数が指定されている場合と指定されていない場合) は、両方とも同じ値 001 を返します。

## 関連項目

- [\\$BIT 関数](#)
- [\\$BITCOUNT 関数](#)
- [\\$BITFIND 関数](#)

- ・ [\\$ZBOOLEAN](#) 関数
- ・ [演算子](#)

## \$CASE (ObjectScript)

expression を比較し、最初に一致した case に関連する値を返します。

### 構文

```
$CASE(target, case:value, case2:value2, ..., :default)
```

### 引数

引数	説明
target	ケースに一致させられる値の式、またはリテラル
case	target の評価の結果と一致させられる値のリテラル、または式
value	対応する case に完全一致したときに返される値
default	オプション - target に一致する case がない場合に返される値

### 概要

\$CASE 関数は target と case のリスト (リテラルや式) を比較し、最初に一致する case の値を返します。case:value の組み合わせは無制限に指定できます。case は指定された順番 (左から右へ) に一致させられます。最初に完全一致したときにマッチングは終了します。

一致する case がない場合は、default が返されます。一致する case がなく、default が指定されていない場合は、InterSystems IRIS は <ILLEGAL VALUE> エラーを発行します。

InterSystems IRIS では、case:value ペアなしの \$CASE を指定できます。これは、target 値に関係なく、常に default 値を返します。

### 引数

#### target

\$CASE は expression を 1 度評価し、その結果を左から右の順に各 case に一致させます。

#### case

case はリテラルや式ですが、リテラルはコンパイル時に評価できるので、式よりも、リテラルを照合した方がはるかに効率的です。各 case は value と組み合わせる必要があります。case と value の組み合わせは無制限に指定できます。

#### value

value はリテラル、または式です。\$CASE を GOTO コマンドや DO コマンドの引数として使用する場合は、value は以下のとおり制限されます。

- ・ \$CASE 文を GOTO コマンドと共に使用する場合、各 value は有効な[行ラベル](#)でなければなりません。式は使用できません。
- ・ \$CASE 文を DO コマンドと共に使用する場合、各 value は有効な DO の引数でなければなりません。これら DO 引数はパラメータを含むことができます。すべての DO コマンド引数と同様に、\$CASE は DO によって呼び出されたときに[後置条件式](#)を取ることができます。

## default

default は、case:value の組み合わせのように指定します。ただし、コンマ区切り文字（項目の分割に使用）とコロン（項目の組み合わせに使用）の間では case は指定されません。default はオプションです。これが指定されている場合は、常に \$CASE 関数の最終引数になります。default 値は、value 引数と同じ GOTO と DO の制限に従います。

一致する case がなく、default が指定されていない場合は、InterSystems IRIS は <ILLEGAL VALUE> エラーを発行します。

## 例

以下の例では、曜日番号を指定し、対応する曜日名を返します。既定値 “entry error” は提供されません。

### ObjectScript

```
SET daynum=$ZDATE($HOROLOG,10)
WRITE $CASE(daynum,
    1:"Monday",2:"Tuesday",3:"Wednesday",
    4:"Thursday",5:"Friday",
    6:"Saturday",0:"Sunday",:"entry error")
```

以下の例では、野球のバッターの出塁記録を入力として指定し、適切な野球用語を書き出します。

### ObjectScript

```
SET hit=$RANDOM(5)
SET atbat=$CASE(hit,1:"single",2:"double",3:"triple",4:"home run",:"strike out")
WRITE hit," = ",atbat
```

以下の例では、\$CASE を DO コマンド引数として使用します。これは exp 指数値に適したルーチン呼び出します。

### ObjectScript

```
Start ; Raise an integer to a randomly-selected power.
    SET exp=$RANDOM(6)
    SET num=4
    DO $CASE(exp,0:NoMul(),2:Square(num),3:Cube(num),:Exponent(num,exp))
    WRITE !,num," ",result,!
    RETURN
Square(n)
    SET result=n*n
    SET result="Squared = "_result
    RETURN
Cube(n)
    SET result=n*n*n
    SET result="Cubed = "_result
    RETURN
Exponent(n,x)
    SET result=n
    FOR i=1:1:x-1 { SET result=result*n }
    SET result="exponent "_x_ = "_result
    RETURN
NoMul()
    SET result="multiply by zero"
    RETURN
```

以下は、入力された文字が英字かそれ以外の文字であるかをテストする例です。

### ObjectScript

```
READ "Input a letter: ",x
SET chartype=$CASE(x?1A,1:"letter",:"other")
WRITE chartype
```

以下の例では、\$CASE を使用して、どの添え字付き変数を返すかを判断しています。



## ObjectScript

```
SET dabbrv="W"  
SET wday(1)="Sunday",wday(2)="Monday",wday(3)="Tuesday",  
    wday(4)="Wednesday",wday(5)="Thursday",wday(6)="Friday",wday(7)="Saturday"  
WRITE wday($CASE(dabbrv,"Su":1,"M":2,"Tu":3,"W":4,"Th":5,"F":6,"Sa":7))
```

次の例は、case:value ペアを指定しません。これは、定義されていない default 文字列を返します。

## ObjectScript

```
SET dummy=3  
WRITE $CASE(dummy,:"not defined")
```

## \$CASE と \$SELECT

\$CASE と \$SELECT はどちらも、一連の式に対して左から右にマッチング操作を実行して、最初の一致に関連する値を返します。\$CASE はターゲットの値を一連の式とマッチングして、最初の一致に関連する値を返します。\$SELECT は一連のブーリアン式をテストして、最初に true に評価された式に関連する値を返します。

## 関連項目

- ・ [DO コマンド](#)
- ・ [GOTO コマンド](#)
- ・ [IF コマンド](#)
- ・ [\\$SELECT 関数](#)

## \$CHANGE (ObjectScript)

入力文字列の文字列単位の部分文字列の置換で構成される新しい文字列を返します。

### 構文

```
$CHANGE(string, searchstr, replacestr, occurrences, begin)
```

### 引数

引数	説明
string	部分文字列の置換が行われる文字列。有効な文字列または数値に解決される任意の式。
searchstr	置換される部分文字列。有効な文字列または数値に解決される任意の式。
replacestr	searchstr の代わりに挿入される部分文字列。有効な文字列または数値に解決される任意の式。
occurrences	オプション - searchstr が replacestr に置換される回数を指定する正の整数。省略した場合は、すべてが置換されます。begin と共に使用する場合、occurrences に -1 の値を指定できます。これは、begin のポイントから文字列の最後まですべての searchstr が置換されることを示します。occurrences は、string 内での出現数より大きな値にできます。
begin	オプション - searchstr のどの出現から置換を開始するかを指定する整数。省略した場合、または 0 か 1 を指定した場合、置換は最初に出現する searchstr から開始されます。

### 説明

\$CHANGE 関数は、入力文字列の文字列単位の置換で構成される新しい文字列を返します。string で searchstr 部分文字列を検索します。\$CHANGE が一致する値を 1 つ以上見つけると、searchstr 部分文字列を replacestr に置換し、結果の文字列を返します。replacestr パラメータ値は、searchstr より長い値も短い値も取ることができます。searchstr 部分文字列を削除するには、replacestr に空文字列("")を指定します。

\$CHANGE と \$REPLACE はどちらも、部分文字列の置換を実行します。どちらも、すべての一致部分文字列、または指定した数の一致部分文字列のみを置換できます。\$CHANGE は、searchstr 部分文字列の出現回数に基づいて、置換を開始する場所を特定します。\$REPLACE は、string の最初からの文字数に基づいて、置換を開始する場所を特定します。\$CHANGE は、置換の実行を開始する場所に関係なく、完全な string (部分文字列の置換を含む) を返します。\$REPLACE は、string の文字カウント開始位置からの部分を返します。

**注釈** \$CHANGE を使用すると文字列の長さが変わる場合があるため、ObjectScript %List や %List オブジェクト・プロパティなどのエンコードされた文字列値で \$CHANGE を使用しないでください。

## \$CHANGE、\$REPLACE、および \$TRANSLATE

\$CHANGE と [REPLACE](#) は文字列と文字列をマッチングして置換します。これらは、1 つ以上の文字の単一の指定された部分文字列を別の任意の長さの部分文字列に置換できます。[\\$TRANSLATE](#) は文字と文字をマッチングして置換します。\$TRANSLATE は、複数の指定された単一文字を対応する単一文字に置換できます。この 3 つの関数はすべて、一致する文字または部分文字列を削除 (NULL に変換) します。

\$CHANGE は、常に大文字と小文字を区別します。\$REPLACE のマッチングは、既定では大文字と小文字を区別しますが、大文字と小文字を区別しないものとして呼び出すこともできます。\$TRANSLATE のマッチングは、常に大文字と小文字を区別します。

\$REPLACE と \$CHANGE は、マッチングの開始ポイントおよび/または実行する置換の数を指定できます。\$REPLACE と \$CHANGE では、開始ポイントの定義方法が異なります。\$TRANSLATE は常に、ソース文字列内のすべての一致を置換します。

## 関連項目

- ・ [\\$REPLACE 関数](#)
- ・ [\\$TRANSLATE 関数](#)
- ・ [\\$EXTRACT 関数](#)
- ・ [\\$PIECE 関数](#)
- ・ [\\$REVERSE 関数](#)
- ・ [\\$ZCONVERT 関数](#)

# \$CHAR (ObjectScript)

式の整数値を、対応した ASCII または Unicode に変換します。

## 構文

```
$CHAR(expression,...)  
$C(expression,...)
```

## 引数

引数	説明
<i>expression</i>	変換される整数値

## 概要

\$CHAR は *expression* で指定した 10 進数 (10 進法) に対応する文字を返します。この文字は 8 ビット (ASCII) 文字あるいは 16 ビット (Unicode) 文字です。8 ビット文字の場合、*expression* の値は、0 から 255 までの正の整数に評価される必要があります。16 ビット文字の場合、256 から 65535 (16 進数 FFFF) までの整数を指定します。65535 より大きい値は、空の文字列を返します。65536 (16 進数 10000) から 1114111 (16 進数 10FFFF) は、Unicode のサロゲート・ペアを表します。これらの文字は \$WCHAR を使用して返されます。

コンマで区切られたリストとして *expression* を指定することもできます。その場合、\$CHAR はリストの各 *expression* に対応する文字を返します。

[\\$ASCII](#) 関数は \$CHAR と逆の機能を持っています。

## 引数

### *expression*

式は、整数値、整数値を含む変数の名前、整数値に評価する有効な ObjectScript 式のいずれかにできます。複数の整数値に対して文字を返すためには、式をコンマで区切ったリストを使用します。

[\\$ZHEX](#) 関数を使用して、10 進数 (10 進法) の文字コードではなく、16 進数の文字コードを使用して文字を指定することができます。以下の例では、両方の \$CHAR 文がギリシャ文字 pi を返します。

### ObjectScript

```
WRITE $CHAR(960),!  
WRITE $CHAR($ZHEX("3C0"))
```

## 例

以下の例は、FOR ループの中で \$CHAR を使用して、65 から 90 までのすべての ASCII コードの文字を出力しています。これらは、大文字のアルファベット文字になります。

### ObjectScript

```
FOR i=65:1:90 {  
    WRITE !,$CHAR(i) }
```

以下の例では、FOR ループで \$CHAR を使用して、日本語のひらがな文字を出力します。

### ObjectScript

```
FOR i=12353:1:12435 {  
    WRITE !,$CHAR(i) }
```

以下の 2 つの例は、複数の expression 値の使用方法です。第 1 の例は "AB" を返し、第 2 の例は "AaBbCcDdEeF-fGgHhIiJjKk" を返します。

## ObjectScript

```
WRITE $CHAR(65,66),!  
FOR i=65:1:75 {  
    WRITE $CHAR(i,i+32) }
```

## \$CHAR および WRITE

WRITE コマンドで文字を出力するために \$CHAR を用いる場合、出力文字は特殊変数 \$X と \$Y の位置を再設定します。これは NULL 文字 (ASCII 0) のときでも True ですが、NULL 文字列 ("") では異なります。原則として、出力不可能な文字を書き込むときの \$CHAR の使用は、慎重に行ってください。このような文字は、予想できないカーソルの動きと画面動作をすることがあります。

## \$CHAR と %List 構造

%List 構造 (%Library.List) は、出力不能文字を使用してエンコードされた文字列であるため、\$CHAR の特定の値は、単一の要素が含まれる %List 構造になります。%List 構造を返す \$CHAR の組み合わせは以下のとおりです。

- ・ \$CHAR(1) は、空のリスト \$1b() を返します。
- ・ \$CHAR(1,1) は、2 要素の空のリスト \$1b(,) を返します。
- ・ \$CHAR(2,1)、\$CHAR(2,2)、または \$CHAR(2,12) は、空の文字列が含まれるリスト \$1b("") を返します。
- ・ \$CHAR(2,4) は、\$1b(0) を返します。
- ・ \$CHAR(2,5) は、\$1b(-1) を返します。
- ・ \$CHAR(2,8) または \$CHAR(2,9) は、\$1b(\$double(0)) を返します。

3 文字以上が含まれ、単一要素リストが生成される \$CHAR の組み合わせは、以下のような構文になります。

```
$CHAR(count,flag,string)
```

count は、合計文字数です。例えば、\$CHAR(5,1,65,66,67) や \$CHAR(5,1)\_"ABC" です。

flag は、string をどのように表すかを指定する整数です。flag の有効な値には、1、2、4、5、6、7、8、9、12、および 13 があります。これらの flag の解釈は、ASCII におけるこの非表示文字の通常解釈とは関係ありません。

- ・ flag=1、flag=12、および flag=13 は、string のリテラル値をリスト要素として返します。
- ・ flag=2 は、count が偶数の場合にのみ有効です。これは、string から得られた 1 つ以上のワイド Unicode 文字 (多くの場合は 1 つ以上の中国語の文字) が含まれるリストを返します。
- ・ flag=4 は、文字の正の ASCII 数値コードをリスト要素として返します。flag=4 は、count > 10 の場合には使用できません。
- ・ flag=5 は、文字の負の整数の ASCII 数値コードをリスト要素として返します。flag=5 は、count > 10 の場合には使用できません。
- ・ flag=6 は、string から得られた正の整数をリスト要素として返します。
  - － \$CHAR(3,6,n) は、常に \$1b(0) を返します。
  - － count > 3 の場合、\$CHAR(count,6,string) は、ASCII 数値から得られた (通常は) 大きい正の整数を返します。末尾のゼロの数は string の最初の文字の ASCII 値に対応し、先頭の数値は文字列の 2 番目の文字の ASCII 値に対応します。例えば、\$CHAR(4,6,0,7) は \$1b(7) を返し、\$CHAR(4,6,3,7) は \$1b(7000) を返します。

flag=6 は、count > 11 の場合には使用できません。

- flag=7 は、string から得られた負の整数をリスト要素として返します。
  - \$CHAR(3,7,n) は、n の値に対応する数のゼロが付いた負の数値を返します。0 = -1、1 = -10、2 = -100、3 = -1000 のようになります。
  - count > 3 の場合、\$CHAR(count,7,string) は、(通常は) 大きい負の整数を返します。末尾のゼロの数は、string の最初の文字の ASCII 値に対応します。

flag=7 は、count > 11 の場合には使用できません。

- flag=8 は、\$DOUBLE(x) を返します。x は小さい数です。flag=8 は、count > 6 の場合には使用できません。
- flag=9 は、\$DOUBLE(x) を返します。x は大きい数です。flag=9 は、count > 10 の場合には使用できません。

string は、count - 2 文字の数値または文字列です。例えば、3 文字の string は、\$CHAR(5,flag,65,66,67) または \$CHAR(5,flag)\_"ABC" として表すことができます。string の値はリスト要素になり、その値は flag の指定に従って表されます。

詳細は、“[\\$LISTBUILD](#)” および “[\\$LISTVALID](#)” を参照してください。

## \$CHAR 引数での数値

expression に符号付きの数値を使用できます。InterSystems IRIS は負数を無視し、正もしくは符号なしの数のみを評価します。以下の例で、符号付きの整数を使用した \$CHAR は、最初と 3 番目の expression のみを返し、負数である 2 番目の expression を無視します。

### ObjectScript

```
WRITE !,$CHAR(65,66,67)
WRITE !,$CHAR(+65,-66,67)
```

ABC

AC

expression に浮動小数点値を使用できます。InterSystems IRIS は引数の小数部を無視し、整数部のみを評価します。以下の例は、\$CHAR が小数部分を無視して、文字コード 65、大文字 A を表す文字を作成しています。

### ObjectScript

```
WRITE $CHAR(65.5)
```

## Unicode サポート

\$CHAR は、10 進数 (10 進法) で表される Unicode 文字をサポートします。

文字の Unicode 値は、通常 0 ～ 9 と A ～ F を使用する 16 進数の 4 桁の数字で表されます。しかし、ObjectScript 言語の基本関数は、通常 16 進数ではなく、10 進数値の ASCII コードによる文字を識別します。

したがって、\$CHAR 関数は Unicode 標準が推奨する 16 進数値の代わりに、\$ASCII 関数は入力文字の 10 進数の Unicode 値を返すことによって、Unicode コード化をサポートします。\$CHAR(\$ZHEX("hexnum")) のように、\$ZHEX 関数で引用符を使用して、16 進数の Unicode 値を指定することができます。hexnum = \$ZHEX(decnum) のように、引用符なしで \$ZHEX を使用して、10 進数を 16 進数に変換することもできます。

詳細は “[Unicode](#)” を参照してください。

## サロゲート・ペア

\$CHAR は、サロゲート・ペアを認識しません。サロゲート・ペアは、一部の中国語の文字を表示したり、日本語の JIS2004 標準をサポートするために使用されます。\$WISWIDE 関数を使用して、文字列にサロゲート・ペアが含まれているかどうか

かを判断することができます。\$WCHAR 関数は、サロゲート・ペアを認識して、正しく解析します。\$CHAR と \$WCHAR は、それ以外は同一です。ただし、\$CHAR は通常 \$WCHAR より高速なため、サロゲート・ペアが出現しない場合は常に \$CHAR が推奨されます。

注釈     \$WCHAR を \$ZWCHAR と混同しないでください。\$ZWCHAR は常にペアの文字を解析します。

## \$CHAR に関連している関数

\$ASCII 関数は \$CHAR と逆の機能を持っています。文字を対応する数値に変換するために使用します。\$ASCII は、Unicode 文字を含むすべての文字を変換します。また、すべての InterSystems IRIS プラットフォームは関連する関数、\$ZLCHAR と \$ZWCHAR をサポートしています。これらは、\$CHAR に似ていますが、ワード (2 バイト) あるいはロングワード (4 バイト) で動作します。\$ZISWIDE を使用して、\$CHAR の expression 内にマルチバイト ("ワイド") 文字があるかどうかを判断します。

## 関連項目

- ・ [READ コマンド](#)
- ・ [WRITE コマンド](#)
- ・ [\\$ASCII 関数](#)
- ・ [\\$WCHAR 関数](#)
- ・ [\\$WISWIDE 関数](#)
- ・ [\\$ZHEX 関数](#)
- ・ [\\$ZLCHAR 関数](#)
- ・ [\\$ZWCHAR 関数](#)
- ・ [\\$X 特殊変数](#)
- ・ [\\$Y 特殊変数](#)



## \$CLASSMETHOD (ObjectScript)

指定されたクラスにある指定クラス・メソッドを実行します。

### 構文

```
$CLASSMETHOD(classname, methodname, arg1, arg2, arg3, ... )
```

### 引数

引数	説明
classname	オプション — 文字列として評価される表現。文字列の内容は、既存のアクセス可能なコンパイル済みクラスの名前と正確に一致する必要があります。InterSystems IRIS クラスを参照する場合は、キャノニック形式 (%Library.String) または省略形式 (%String) で名前を指定できます。  classname が省略されている場合、現在のクラス・コンテキストが使用されます(現在のクラス・コンテキストは <a href="#">\$THIS</a> を使用して確認できます)。classname が省略されている場合、プレースホルダとしてのコンマを指定する必要があります。
methodname	有効な文字列として評価する表現。文字列の値は、classname で指定したクラスにある既存のクラス・メソッドの名前と一致する必要があります。
arg1, arg2, arg3, ...	オプション — 指定したメソッドへの引数を順次置き換える一連の式。式の値には任意の型のものを使用できます。実装するユーザは、指定した表現の型とメソッドで想定している型が一致していること、また宣言された境界の範囲の値が使用されていることを確認する必要があります(指定したメソッドで引数の使用を想定していない場合、この関数呼び出しでは methodname 以外の引数を指定する必要はありません。メソッドが引数を必要とする場合、どの引数を指定する必要があるかを規定するルールとして、ターゲット・メソッドのルールが適用されます)。

### 説明

\$CLASSMETHOD は、ObjectScript プログラムが任意のクラス内の任意のクラス・メソッドを呼び出すことを許可します。クラス名とメソッド名はいずれも、実行時に計算することもできれば、文字列定数として指定しておくこともできます。クラス・メソッドではなくインスタンス・メソッドを呼び出すには、\$METHOD 関数を使用します。

メソッドが引数を取る場合は、メソッド名の後の引数リストで引数を指定します。最大 255 の引数値がメソッドに渡されます。

関数またはプロシージャとして \$CLASSMETHOD を呼び出すと、ターゲット・メソッドの呼び出しが確定します。  
\$CLASSMETHOD は、JOB コマンド、または DO コマンドを使用して呼び出して、返り値を破棄できます。すべての DO コマンド引数と同様に、\$CLASSMETHOD は DO によって呼び出されたときに[後置条件パラメータ](#)を取ることができます。

存在しないクラスを呼び出そうとすると、<CLASS DOES NOT EXIST> の後に現在のネームスペース名と指定されたクラス名が記述されたエラーが出力されます。例えば、存在しない classname “Fred” を呼び出そうとすると、エラー <CLASS DOES NOT EXIST> \*User.Fred が出力されます。classname に空の文字列を指定すると、<CLASS DOES NOT EXIST> \*(No name) が出力されます。

存在しないクラス・メソッドを呼び出そうとすると、<METHOD DOES NOT EXIST> エラーが発生します。

## 例

以下は、\$CLASSMETHOD を関数として使用する例です。

### ObjectScript

```
SET classname = "%Dictionary.ClassDefinition"
SET classmethodname = "NormalizeClassname"
SET singleargument = "%String"
WRITE $CLASSMETHOD(classname, classmethodname, singleargument), !
```

%Library.String が返されます。

以下は、2 つの引数を指定して \$CLASSMETHOD を使用する例です。

### ObjectScript

```
WRITE $CLASSMETHOD("%Library.Persistent", "%PackageName"), !
WRITE $CLASSMETHOD("%Library.Persistent", "%ClassName")
```

この呼び出しでは、%Library と %Persistent が返されます。

以下の例では、\$CLASSMETHOD を使用してダイナミック SQL クエリを実行します。

### ObjectScript

```
SET q1="SELECT Age,Name FROM Sample.Person "
SET q2="WHERE Age > ? AND Age < ? "
SET q3="ORDER by Age"
SET myquery=q1_q2_q3
SET rset=$CLASSMETHOD("%SQL.Statement", "%ExecDirect", ,myquery,12,20)
DO rset.%Display()
WRITE !,"Teenagers in Sample.Person"
```

## 関連項目

- ・ [\\$CLASSNAME](#) 関数
- ・ [\\$METHOD](#) 関数
- ・ [\\$PARAMETER](#) 関数
- ・ [\\$PROPERTY](#) 関数
- ・ [\\$THIS](#) 特殊変数

# \$CLASSNAME (ObjectScript)

クラス名を返します。

## 構文

`$CLASSNAME (oref)`

## 引数

引数	説明
oref	オプション - クラス・インスタンスへのオブジェクト参照 (OREF)。省略した場合は、現在のクラスの名前が返されます。

## 説明

`$CLASSNAME` はクラス名を返します。一般的には、オブジェクト参照 (OREF) を取得して、対応するクラス名を返します。引数なしの `$CLASSNAME` は、現在のクラスの名前を返します。`$CLASSNAME` は常に完全なクラス名 (`%SQL.Statement` など) を返し、パッケージ名を省略した短縮版のクラス名 (`Statement` など) は返しません。

`$CLASSNAME` は、機能的には `%Library.Base` スーパークラスの `%ClassName(1)` メソッドと同等です。`$CLASSNAME` 関数は、完全なクラス名を返す点で `%ClassName(1)` よりもパフォーマンスが優れています。短縮版のクラス名を返すには、`%ClassName()` もしくは `%ClassName(0)` のいずれかを使用することができます。

OREFの詳細は、“[OREF の基本](#)”を参照してください。

## 例

以下の例は、クラスのインスタンスを作成します。`$CLASSNAME` は、OREF インスタンスを取り、対応するクラス名を返します。

### ObjectScript

```
SET dynoref = ##class(%SQL.Statement).%New()
WRITE "instance class name: ", $CLASSNAME(dynoref)
```

以下は、パラメータの指定なしの `$CLASSNAME` で現在のクラス・コンテキストのクラス名を返す例です。この例では、`DocBook.Utils` クラスです。これは、`$THIS` 特殊変数に含まれるクラス名と同じです。

### ObjectScript

```
WRITE "class context: ", $CLASSNAME(), !
WRITE "class context: ", $THIS
```

以下の例では、`$CLASSNAME` 関数および `%ClassName(1)` メソッドが同じ値を返すことを示しています。また、`%ClassName()` メソッドを (引数なし、もしくは引数 0 で) 使用して、短縮版のクラス名を返すことも示しています。

### ObjectScript

```
CurrentClass
WRITE "current full class name: ", $CLASSNAME(), !
WRITE "current full class name: ", ...%ClassName(1), !
WRITE "current short class name: ", ...%ClassName(0), !
WRITE "current short class name: ", ...%ClassName(), !!
ClassInstance
SET x = ##class(%SQL.Statement).%New()
WRITE "oref full class name: ", $CLASSNAME(x), !
WRITE "oref full class name: ", x.%ClassName(1), !
WRITE "oref short class name: ", x.%ClassName(0), !
WRITE "oref short class name: ", x.%ClassName()
```

## 関連項目

- ・ [\\$CLASSMETHOD](#) 関数
- ・ [\\$METHOD](#) 関数
- ・ [\\$PARAMETER](#) 関数
- ・ [\\$PROPERTY](#) 関数
- ・ [\\$THIS](#) 特殊変数

## \$COMPILE (ObjectScript)

ソース・コードをコンパイルし、実行可能なオブジェクト・コードを生成します。

### 構文

```
$COMPILE (source, language, errors, object)
```

```
$COMPILE (source, language, errors, object, , , rname)
```

### 引数

引数	説明
source	コンパイルするソース・コードが含まれる添え字付き配列を指定するローカル変数またはグローバル変数。
language	ソース・コードのプログラミング言語を指定する整数フラグ。0 = ObjectScript。
errors	コンパイル時に発生するエラーを受け取る添え字なしのローカル変数。この変数は、報告されたエラーごとに 1 つの要素を持つリスト構造です。各エラーは、それ自体がエラーの位置とタイプを指定するリスト構造になっています（以下を参照）。
object	1 番目の構文 - コンパイルされたオブジェクト・コードを保持するために使用される配列の生成に使用する添え字なしのローカル変数またはグローバル変数。  2 番目の構文 - この引数はオプションです。指定した場合、object から以前の値がクリアされますが、設定されません。一般に、2 番目の構文では object を省略し、プレースホルダのコンマを指定します。
rname	2 番目の構文 - コンパイルされたオブジェクト・コードを ^rOBJ グローバルに格納するために使用されるルーチン名を指定する文字列。

### 説明

\$COMPILE は、ソース・コードをコンパイルし、OBJ (オブジェクト) コード (ルーチンの実行可能な形式) を生成します。\$COMPILE は、コンパイル・エラーを報告します。これを使って、実際にオブジェクト・コードを生成せずに、ソース・コードにコンパイル・エラーがあるかどうかを確認できます。\$COMPILE は、MAC コードではなく INT コードを入力として取ります。したがって、コンパイルの前に、ObjectScript マクロ・プリプロセッサなどのプリプロセッサでソース・コード内のマクロを解決する必要があります。

**注釈** 一般にソース・コードのコンパイルは、\$COMPILE 関数ではなく、任意の IDE を使用して実行されます。

\$COMPILE には、次の 2 つの構文形式があります。

- ・ \$COMPILE の 1 つ目の構文形式では、オブジェクト・コードが object 配列に返されます。まず、object 変数が削除されます。コンパイル後、object 配列がコンパイルされたオブジェクト・コードのサイズに設定されます。

object 配列には、^rOBJ グローバルの場合と同じ形式でオブジェクト・コードが格納されます。^rOBJ 内のオブジェクト・コードは、MERGE ^rOBJ(rname)=object(1) コマンドによって新しいオブジェクト・コードに置換できます。ただし、MERGE コマンドは複数のノードを設定するときにアトミックでないため、この操作と同時に別のプロセスが同じルーチンをロードしている場合は、予測できない結果が発生する可能性があります。

object 引数を省略すると、ソース・コードのコンパイルとエラー・チェックが実行されますが、オブジェクト・コードは作成されません。

- 2 番目の \$COMPILE 構文形式は、オブジェクト・コードを rname という名前のルーチンに直接返します。この OBJ コードは、`^rOBJ(rname)` を返すことによって表示できます。\$COMPILE 操作では、新しいオブジェクト・コードが完全に格納されるまで他のプロセスがルーチンのオブジェクト・コードをロードしないように、内部で `^rOBJ(rname)` がロックされます。

rname 引数を省略すると、ソース・コードのコンパイルとエラー・チェックが実行されますが、オブジェクト・コードは作成されません。

一般に、この構文形式では object 引数 (4 番目の引数) は省略されます。object 引数を指定すると、その object 変数は削除されますが、設定されません。省略された (プレースホルダのコンマで表される) 他の引数は、内部用のため、指定しないでください。

\$COMPILE は、次のように整数コードを返します。0 = エラーが検出されず、オブジェクト・コードが作成された場合。1 = エラーが検出され、オブジェクト・コードが作成された場合。-1 = エラーが検出され、オブジェクト・コードが作成されなかった場合。オブジェクト・コードを格納する引数 (1 番目の構文では object、2 番目の構文では rname) を省略した場合、同じ戻りコードが返されます。

ObjectScript コンパイラは、エラーが検出された時点で、その行が実行されたときにエラーを返すオブジェクト・コードを作成します。

## 引数

### source

(INT ルーチンの形式で) コンパイルするソース・コードを格納する配列。配列要素 source(0) にはソース・コードの行数を格納し、source(n) にはソース・コードの行番号 n を格納します。ソース行には 1 ~ n の連番を省略なしで付ける必要があります。実行可能な ObjectScript コードはインデントする必要があります。以下に例を示します。

### ObjectScript

```
SET mysrc(0)=6
SET mysrc(1)=" SET x=1"
SET mysrc(2)="Main" // a label
SET mysrc(3)=" WRITE "x is:",x,!"
SET mysrc(4)=" SET x=x+1"
SET mysrc(5)=" IF x=4 {WRITE "x is:",x," all done" QUIT}"
SET mysrc(6)=" GOTO Main"
SET rtn=$COMPILE(mysrc,0,errs,,, "myobj")
IF rtn=0 {WRITE "OBJ code successfully generated",!}
ELSE {WRITE "no OBJ code generated return code: ",rtn,! QUIT}
WRITE "Running the code",!!
DO ^myobj
```

source 引数には、添え字なしのローカル変数名か、場合によっては添え字付きのグローバル名を指定できます。

source(0) が未定義の場合、システムは、%SYSTEM.Process.Undefined() メソッドの設定にかかわらず、<UNDEFINED> エラーを生成します。

source(0) 値がソース・コードの行数より大きい場合や、連続するソース・コード行が見つからない場合、システムは <UNDEFINED> エラーを生成した後に、見つからないソース・コード行の名前を表示します。この動作は、%SYSTEM.Process.Undefined() メソッドを設定して変更できます。以下の例では、これらの種類のエラーを示します。

### ObjectScript

```
SET src(0)=4,src(1)="TestA ",src(2)=" WRITE 123",src(3)=" WRITE 456,!"
SET stat=$COMPILE(src,0,errs,TestA) /* generates <UNDEFINED> *src(4) */
```

### ObjectScript

```
SET src(0)=4,src(1)="TestA ",src(3)=" WRITE 123",src(4)=" WRITE 456,!"
SET stat=$COMPILE(src,0,errs,TestA) /* generates <UNDEFINED> *src(2) */
```

## ObjectScript

```
SET src(0)=3,src(1)="TestA ",src(3)=" WRITE 123",src(4)=" WRITE 456,!"
SET stat=$COMPILE(src,0,errs,TestA) /* generates <UNDEFINED> *src(2) */
```

## language

コンパイルするソースのタイプを指定する言語モード。0 を使用すると、ObjectScript が指定されます。

他の値を使用すると、従来のモードが指定されます。使用する場合は、事前にインターシステムズのサポート窓口にお問い合わせください。

## エラー

コンパイラが検出したエラーが設定される添え字なしのローカル変数。既存の値は削除されます。エラーが検出されなかった場合は、空の文字列 ("") がこの変数に設定されます。エラーが検出された場合は、エラーごとに 1 つの要素を持つ \$LIST 構造が errors 変数に設定されます。各エラーは、それ自体が \$LISTBUILD(line,offset,errnum,text) という形式の \$LIST 構造です。各要素の内容は以下のとおりです。

- ・ line = エラーが検出された行番号
- ・ offset = ソース行でのエラーのオフセット
- ・ errnum = エラーのタイプを示すエラー番号
- ・ text = エラーを説明するテキスト

## object

コンパイラのオブジェクト・コード出力を受け取る配列。object 引数には、添え字なしのローカル変数名か、場合によっては添え字付きのグローバル名を指定できます。object 配列の内容は、上記のとおりです。

## rname

オブジェクト・コードを添え字付きの ^rOBJ グローバルのどこに保存する必要があるかを指定するルーチン名。\$COMPILE は、新しいオブジェクト・コードを保存する前に、^rOBJ(rname) の既存の内容を削除します。以下の例では、rname="myobj" です。

OBJ コードを表示するには、以下を使用します。

## ObjectScript

```
WRITE ^rOBJ("myobj")
```

あるいは以下のようになります。

## ObjectScript

```
ZWRITE ^rOBJ("myobj")
```

OBJ コードを実行するには、以下を使用します。

## ObjectScript

```
DO ^myobj
```

OBJ コードの作成タイムスタンプと長さをリストするには、以下を使用します。

## ObjectScript

```
ZWRITE ^rINDEX("myobj")
```



\$COMPILE で作成されたコードには、対応する保存済みの MAC または INT コード・バージョンがないため、`^rINDEX()` では OBJ コード行のみがリストされることに注意してください。

## その他のコンパイル・インタフェース

InterSystems IRIS では、1 つ以上のクラスをコンパイルするためのクラス・メソッドが提供されています。

- ・ `$SYSTEM.OBJ.Compile()` は、指定したクラスおよびそのクラス内のすべてのルーチンをコンパイルします。
- ・ `$SYSTEM.OBJ.CompileList()` は、指定したクラスのリストおよびそれらのクラス内のすべてのルーチンをコンパイルします。
- ・ `$SYSTEM.OBJ.CompilePackage()` は、指定したパッケージ (スキーマ) 内のすべてのクラス/ルーチンをコンパイルします。
- ・ `$SYSTEM.OBJ.CompileAll()` は、現在のネームスペースのすべてのクラス/ルーチンをコンパイルします。
- ・ `$SYSTEM.OBJ.CompileAllNamespaces()` は、すべてのネームスペースのすべてのクラス/ルーチンをコンパイルします。

これらのメソッドには、コンパイル・オプションをより正確に指定するための修飾子とフラグがあります。”[システム・フラグおよびシステム修飾子 \(qspec\)](#)” を参照してください。

## コンパイルの中断

実行中のコンパイルは、**Ctrl-C** を発行するか、または `^RESJOB` ユーティリティを呼び出すことで中断できます。このようなコンパイルの中断は、すべての language モードでサポートされています。

## コンパイラのバージョン

`%SYSTEM.Version.GetCompilerVersion()` メソッドを使用すると、現在のコンパイラのバージョンを返すことができます。InterSystems IRIS は、同じメジャー・コンパイラ・バージョン番号でコンパイルされたオブジェクト・コードのみを実行できます。現在のマイナー・コンパイラ・バージョン以下のマイナー・コンパイラ・バージョン番号でコンパイルされたオブジェクト・コードを実行できません。

## 例

次の例では、\$COMPILE の 1 つ目の形式を使用して、4 行の ObjectScript プログラムをコンパイルしています。

### ObjectScript

```
SourceCode
    SET src(0)=4
    SET src(1)="TestA "
    SET src(2)=" WRITE "Hello " " "
    SET src(3)=" WRITE "World",!"
    SET src(4)=" QUIT"
CompileSource
    SET stat=$COMPILE(src,0,errs,TestA)
    IF stat=0 {WRITE "Compile successful" }
    ELSE {WRITE "status=",stat,!
           WRITE "number of compile errors=", $LISTLENGTH(errs) }
```

次の例では、\$COMPILE の 2 つ目の形式を使用して、同じ 4 行の ObjectScript プログラムをコンパイルしています。

## ObjectScript

```

SourceCode
  SET src(0)=4
  SET src(1)="TestB "
  SET src(2)=" WRITE "Hello " " "
  SET src(3)=" WRITE "World" ",!"
  SET src(4)=" QUIT"
CompileSource
  SET stat=$COMPILE(src,0,errs,,,"TestB")
  IF stat=0 {WRITE "Compile successful",!
    DO ^TestB }
  ELSE {WRITE "status=",stat,!
    WRITE "number of compile errors=", $LISTLENGTH(errs) }

```

次の例では、7 行の ObjectScript プログラムに対してコンパイル・エラー・チェックを実行しています。この \$COMPILE は、エラーの有無を検査するだけであり、正常なコンパイルによって生成されたオブジェクト・コードを受け取るための変数は提供しません。この例では、ソース・コードのすべての行にエラーが含まれていますが、\$COMPILE は 1、3、5、6、および 7 行目のコンパイル時エラーのみを返し、0 による除算エラー (2 行目) や未定義変数エラー (4 行目) などの実行時エラーは返しませんが。

## ObjectScript

```

SourceCode
  SET src(0)=7
  SET src(1)="?TestC "
  SET src(2)=" SET a=2/0"
  SET src(3)=" SET b=3+#2"
  SET src(4)=" SET c=xxx"
  SET src(5)=" SET? d=5"
  SET src(6)=" SET 123="abc""
  SET src(7)=" SETT f=7"
CompileSource
  SET stat=$COMPILE(src,0,errs)
  IF stat {WRITE $LISTLENGTH(errs)," Compile Errors ",!
    FOR i=1:1:$LISTLENGTH(errs) {
      WRITE !,i,": "
      SET errn=$LIST(errs,i)
      FOR j=1:1:$LISTLENGTH(errn) {
        WRITE $LIST(errn,j)," "
      }
    }
  }
  ELSE {WRITE "Compile successful",!
    WRITE "but no object code generated" }

```

## 関連項目

- ・ [クラスのコンパイルと配置](#)
- ・ [システム・フラグおよびシステム修飾子 \(qspec\)](#)
- ・ [XECUTE コマンド](#)
- ・ [ZLOAD コマンド](#)
- ・ [ZSAVE コマンド](#)
- ・ [マクロとインクルード・ファイルの使用](#)

## \$DATA (ObjectScript)

variable にデータがあるか否かを調べます。

### 構文

```
$DATA(variable, target)
$D(variable, target)
```

### 引数

引数	説明
variable	状態が確認される変数。ローカルあるいはグローバルの、添え字付きあるいは添え字なしの変数。この変数は未定義でもかまいません。単純なオブジェクト・プロパティ参照を variable として指定することはできません。構文 obj.property を使用すると、多次元プロパティ参照を variable として指定できます。
target	オプション - \$DATA が variable の現在の値を返す変数。

### 概要

\$DATA を使用して、処理する前に variable にデータがあるか否かを調べます。\$DATA は、指定した変数についての情報を返します。variable 引数は、あらゆる変数（ローカル変数、プロセス・プライベート・グローバル、またはグローバル）の名前にできます。また、添え字付き配列の要素を含めることができます。これは[多次元オブジェクト・プロパティ](#)であってもかまいませんが、非多次元オブジェクト・プロパティであってははいけません。

可能な状態値は、以下のとおりです。

状態値	意味
0	変数が未定義です。すべての参照によって <UNDEFINED エラー>が発生します。
1	変数が存在し、データを含みますが、下位ノードはありません。NULL 文字列(“”)もデータと見なされます。
10	変数が下位ノード（他の配列要素の下方ポインタを含みます）を持つ配列要素を識別しますが、データは含みません。このような変数への直接参照は、<UNDEFINED> エラーになります。例えば、y(1) が定義されているものの、y が定義されておらず、\$DATA (y) が 10 を返す場合、set x = y は <UNDEFINED> エラーを生成します。
11	変数が下位ノード（他の配列要素の下方ポインタも含みます）を持つ配列要素を識別し、データも含みます。この種類の変数は、式で参照できます。

モジュロ 2 (#2) 演算を使用して、ブーリアン値を \$DATA から返すことができます。\$DATA (var) #2 は、未定義のステータス・コード (0 および 10) には 0 を返し、定義済みのステータス・コード (1 および 11) には 1 を返します。

状態値 1 と 11 は、データの種類ではなく存在のみを示します。

%SYSTEM.Process クラスの Undefined() メソッドを使用すると、未定義の変数が見つかった場合の動作を設定できます。<UNDEFINED> エラーの詳細は、[“\\$ZERROR” 特殊変数](#)を参照してください。

### \$DATA によるロック、ルーチン、ジョブ、およびグローバルのテスト

- \$DATA(^\$LOCK(lockname)) は、ロックが存在するかどうかをテストします。戻り値が異なることに注意してください。0 = ロックが存在しません。10 = ロックが存在します。ロックの下位ノードは判断できません。値 1 と 11 が返されることはありません。詳細は [“\\$LOCK”](#) を参照してください。

- ・ `$DATA(^$ROUTINE(routinename))` は、ルーチンの OBJ コード・バージョンが存在するかどうかをテストします。戻り値が異なることに注意してください。0 = ルーチンの OBJ コードが存在しません。1 = ルーチンの OBJ コードが存在します。値 10 と 11 が返されることはありません。詳細は “[\\$ROUTINE](#)” を参照してください。
- ・ `$DATA(^$JOB(jobnum))` は、ジョブが存在するかどうかをテストします。戻り値が異なることに注意してください。0 = ジョブが存在しません。1 = ジョブが存在します。値 10 と 11 が返されることはありません。詳細は “[\\$JOB](#)” を参照してください。
- ・ `$DATA(^$GLOBAL(globalname))` は、グローバルが存在するかどうかをテストします。戻りコードは変数の場合と同じで、0、1、10、および 11 です。詳細は “[\\$GLOBAL](#)” を参照してください。

## 引数

### variable

データの存在に対してテストされる変数:

- ・ `variable` は、ローカル変数、グローバル変数、またはプロセス・プライベート・グローバル (PPG) 変数のいずれかにすることができます。これは添え字付きでも添え字なしでもかまいません。

グローバル変数の場合は、[拡張グローバル参照](#)を含めることができます。添え字付きグローバル変数の場合は、[ネイキッド・グローバル参照](#)を使用して指定できます。未定義の添え字付きグローバル変数を参照するときでも、`variable` はネイキッド・インジケータをリセットするので、以下で説明されているように、今後のネイキッド・グローバル参照に影響が及びます。

- ・ `variable` は [多次元オブジェクト・プロパティ](#)とすることができます。非多次元オブジェクト・プロパティとすることはできません。非多次元オブジェクト・プロパティで `$DATA` を使用しようとする、<OBJECT DISPATCH> エラーが発生します。

例えば、`%SQL.StatementMetadata` クラスには、多次元プロパティ `columnIndex` と、非多次元プロパティ `columnCount` があります。以下の例では、最初の `$DATA` は値を返し、2 番目の `$DATA` は <OBJECT DISPATCH> エラーを返します。

### ObjectScript

```
SET x=##class(%SQL.StatementMetadata).%New()
WRITE "columnIndex defined: ", $DATA(x.columnIndex), !
WRITE "columnCount defined: ", $DATA(x.columnCount)
```

- ・ `variable` が [\\$ROUTINE](#) 構造化システム変数の場合、返される状態値は 1 または 0 が可能です。

### target

オプションの引数です。ローカル変数名、プロセス・プライベート・グローバル名、またはグローバル名を指定します。この `target` 変数は定義する必要はありません。`target` が指定されている場合、`$DATA` は `target` に `variable` の現在のデータ値を書き込みます。`variable` が未定義の場合、`target` 値は変化しません。

ZBREAK コマンドでは、`target` 引数をウォッチポイントとして指定することはできません。

## 例

この例では、`^client` 配列から選択した範囲のレコードを書き込みます。`^client` 配列は 3 段階のスペース配列で構成されています。最初の添え字には顧客名、第 2 添え字には顧客の住所、第 3 添え字には顧客の口座、口座番号、残高が含まれています。顧客は 4 種類までの口座を持つことができます。3 つの添え字のどこかに未定義の要素があるかもしれません。一般的なレコードの内容は、以下のようになる場合があります。

```

^client(5) John Jones
^client(5,1) 23 Bay Rd./Boston/MA 02049
^client(5,1,1) Checking/45673/1248.00
^client(5,1,2) Savings/27564/3270.00
^client(5,1,3) Reserve Credit/32456/125.00
^client(5,1,4) Loan/81263/460.00

```

以下のコードは3つの各配列添え字の出力を処理するために、別々のサブルーチンを提供します。各サブルーチンの始めに \$DATA 関数を使用して、現在の配列要素をテストします。

Level 1、Level 2、Level 3 の \$DATA=0 テストは、現在の配列要素が未定義であるかどうかをテストします。True であれば、コードはQUIT して前のレベルに戻ります。

Level 1 と Level 2 の \$DATA=10 テストは、現在の配列要素がデータを含まずに、下位の要素へのポインタを含んでいるかどうかを調べます。True であれば、コードが "No Data" と書き出します。それから、コードは次の下位レベルの FOR ループ・プロセスに飛びます。このレベルの下に要素はないので、Level 3 の \$DATA=10 テストは実行されません。

Level 2 と Level 3 の WRITE コマンドは、\$PIECE 関数を使用して、現在の配列要素から適切な情報を取り出します。

## ObjectScript

```

Start Read !,"Output how many records: ",n
Read !,"Start with record number: ",s
For i=s:1:s+(n-1) {
  If $Data(^client(i)) {
    If $Data(^client(i))=10 {
      Write !," Name: No Data"
    }
    Else {
      Write !," Name: " ,^client(i)
    }
  }
  If $Data(^client(i,1)) {
    If $Data(^client(i,1))=10 {
      Write !,"Address: No Data"
    }
    Else {
      Write !,"Address: " , $Piece(^client(i,1),"/",1)
      Write " , " , $Piece(^client(i,1),"/",2)
      Write " , " , $Piece(^client(i,1),"/",3)
    }
  }
  For j=1:1:4 {
    If $Data(^client(i,1,j)) {
      Write !,"Account: " , $Piece(^client(i,1,j),"/",1)
      Write " #: " , $Piece(^client(i,1,j),"/",2)
      Write " Balance: " , $Piece(^client(i,1,j),"/",3)
    }
  }
}
Write !,"Finished."
Quit

```

実行されると、このコードは以下のような出力をする場合もあります。

```

Output how many records: 3
Start with record number: 10
Name: Jane Smith
Address: 74 Hilltop Dr., Beverly, MA 01965
Account: Checking #: 34218 Balance: 876.72
Account: Reserve Credit #: 47821 Balance: 1200.00
Name: Thomas Brown
Address: 46 Huron Ave., Medford, MA 02019
Account: Checking #: 59363 Balance: 205.45
Account: Savings #: 41792 Balance: 1560.80
Account: Reserve Credit #: 64218 Balance: 125.52
Name: Sarah Copley
Address: No Data
Account: Checking #: 30021 Balance: 762.28

```

## ネイキッド・グローバル参照

\$DATA は、グローバル変数と共に使用されるとき、ネイキッド・インジケータを設定します。ネイキッド・インジケータは、指定されたグローバル変数が定義されていない場合 (状態値は 0) でも設定されます。

同じグローバル変数への後続の参照は、以下の例で示されるようにネイキッド・グローバル参照を使用することができます。

### ObjectScript

```
IF $DATA(^A(1,2,3))#2 {  
  SET x=^(3) }  
}
```

ネイキッド・グローバル参照とグローバル変数での \$DATA の使用に関する詳細は、“[多次元ストレージの使用法\(グローバル\)](#)”を参照してください。

## ネットワーク環境でのグローバル参照

定義されていないグローバル変数を繰り返し参照するために \$DATA を使用すると (例えば、 $\hat{x}$  が定義されていない `$DATA(^x(1))` など)、そのグローバル変数が ECP データ・サーバで定義されているかどうかをテストするネットワーク処理が常に必要となります。

定義されたグローバル変数内で、定義されていないノードを繰り返し参照するために \$DATA を使用すると (例えば、 $\hat{x}$  にあるその他のノードがすべて定義済みの `$DATA(^x(1))`)、いったんグローバルの一部 ( $\hat{x}$ ) がクライアント・キャッシュに格納された後には、ネットワーク処理が必要なくなります。

詳細は、“スケーラビリティ・ガイド”の“[分散キャッシュ・アプリケーションの開発](#)”を参照してください。

## \$DATA に関連している関数

関連情報については、“\$GET”と“\$ORDER”を参照してください。\$ORDER は、データを含む配列にある次の要素を選択するので、配列添え字をループするときに \$DATA テストを実行する必要がありません。

## 関連項目

- ・ [KILL コマンド](#)
- ・ [SET コマンド](#)
- ・ [\\$GET 関数](#)
- ・ [\\$ORDER 関数](#)
- ・ [多次元ストレージの使用法 \(グローバル\)](#)

## \$DECIMAL (ObjectScript)

浮動小数値に変換した数値（具体的にはインターシステムズの [10 進形式](#) の値）を返します。

### 構文

```
$DECIMAL (num, n)
```

### 引数

引数	説明
num	変換される数値。これは通常 IEEE バイナリ浮動小数点数（つまり、インターシステムズの <a href="#">\$DOUBLE 形式</a> の数値）です。
n	オプション - 返される有効桁数を指定する整数。 <a href="#">\$DECIMAL</a> は、 <a href="#">値をその有効桁数に丸めて</a> 、 <a href="#">キャノニック形式の数値文字列</a> を返します。有効な値は、1 から 38、および 0 です。0 値の詳細は <a href="#">以下を参照</a> してください。n が 38 より大きい場合、<ILLEGAL VALUE> エラーが生成されます。

### 説明

[\\$DECIMAL](#) は、FLOAT [SQL データ型](#) に対応する、インターシステムズの [10 進形式](#) に変換された数値を返します。これは、[\\$DOUBLE](#) 関数で実行される処理の逆です。

注釈 [\\$DOUBLE 形式](#) の小数は、通常その [10 進数](#) 変換とはわずかに異なります。“[インターシステムズ・アプリケーションでの数値の計算](#)” を参照してください。

num 値は、数字または[数値文字列](#)で指定できます。num 値が、インターシステムズの [10 進形式](#) に変換可能な値の範囲外である場合、[\\$DECIMAL](#) は <MAXNUMBER> エラーを生成します。

[\\$DECIMAL](#) は、[キャノニック形式](#) で数値を返します。

- ・ [\\$DECIMAL\(num\)](#) は、インターシステムズの [10 進形式](#) の数値を返します。この結果の値は、[18 桁の精度または 19 桁の精度](#) です。
- ・ [\\$DECIMAL\(num,n\)](#) は、インターシステムズの [10 進形式](#) の数値を表す数値文字列を返します。一般に、数値文字列は対応する数値に変換されます。例外については、“[非常に大きな数値文字列](#)” を参照してください。

### 丸め

n が 1 から 38 (1 と 38 を含む) の [\\$DECIMAL\(num,n\)](#) を指定すると、有効桁数 n 以下のキャノニック形式の数値文字列が返されます。num が ObjectScript の [10 進形式](#) の数値の場合、最大で 19 桁の有効な小数点以下の桁数を返します。num が IEEE バイナリの場合、入力には少なくとも 767 桁の小数点以下の有効桁数を指定できますが、[\\$DECIMAL\(num,n\)](#) は 38 桁を超える有効桁数は表示しません。

キャノニック形式の数値文字列では、小数点の前に先行ゼロ、および小数点の後に後置ゼロを付けることはできません。

[\\$DECIMAL\(num,n\)](#) は 38 桁を超える有効桁数は表示しませんが、[\\$FNUMBER](#) 関数を使用すると、より多くの有効桁数を IEEE バイナリ値で表示できます。長いシーケンスの有効桁数を持つ文字列が与えられた場合、[\\$DOUBLE](#) 関数は最初の 38 桁のみを調べます。

丸めは、以下のように行われます。

- ・ n が指定されておらず、num の有効桁数が 19 桁より多い場合、[\\$DECIMAL](#) は数値を丸めて、[有効桁数 18 桁または 19 桁](#) のインターシステムズの [10 進形式](#) を返します。[\\$DECIMAL](#) は常に、絶対値が大きい方に丸めます。ただし、ObjectScript の [10 進形式](#) に変換される値が 18 桁の精度から 19 桁の精度の間に収まる場合は例外です。こ



の場合、結果は大きい方の 18 桁の値になります。以下に、有効桁数 20 桁の数値を \$DECIMAL(num) が有効桁数 19 桁に丸める例を示します。

#### Terminal

```
USER>w $DOUBLE(12345678901234567890123456789)
123456789012345682270000000000
USER>w $DECIMAL($DOUBLE(12345678901234567890123456789))
123456789012345682300000000000
```

- ・ n が正の整数の場合、IEEE 標準規格の丸めを使用して丸めが行われます。\$DECIMAL は、キャノニック形式の数値文字列で [10 進形式](#) の数値を返します。num の有効桁数が 38 桁より多い (加えて n=38) 場合、\$DECIMAL は、38 桁目の小数部分を丸め、これ以下の num の桁をすべてゼロで表します。
- ・ n=0 の場合、\$DECIMAL は、以下のようにキャノニック形式の数値文字列で [10 進形式](#) の数値を返します。
  - － n=0 で、num に対応する [10 進形式](#) の数値が 20 桁以下の場合、\$DECIMAL はその形式値を返します。
  - － n=0 で、num に対応する [10 進形式](#) の数値が 20 桁より多い場合、以下のような特殊な丸め (IEEE 丸めではない) が行われます。10 進数文字列値は有効桁数 20 桁に切り捨てられます。20 桁目が "5" でも "0" でもない場合、その切り捨てられた、有効桁数 20 桁の数値が結果になります。20 桁目が "5" または "0" の場合、20 桁目はそれぞれ "6" または "1" に置き換えられ、その 20 桁の数値文字列が結果になります。結果が後から 20 桁未満に丸められる場合、この特殊な丸めによって "double round" エラーを防ぐことができます。これを以下の例で示します。

#### Terminal

```
USER>WRITE $DECIMAL($DOUBLE(123456789012345678901234567),20)
1234567890123456781500000000
USER>WRITE $DECIMAL($DOUBLE(123456789012345678901234567),0)
1234567890123456781600000000
```

## 整数除算

特定の値の場合、インターシステムズの [10 進形式](#) および [\\$DOUBLE 形式](#) の数値は、異なる整数除算結果を生成します。以下はその例です。

#### ObjectScript

```
WRITE !,"Integer divide operations:"
WRITE !,"IRIS \: ", $DECIMAL(4.1)\.01 // 410
WRITE !,"Double \: ", $DOUBLE(4.1)\.01 // 409
```

詳細は、"[インターシステムズ・アプリケーションでの数値の計算](#)" を参照してください。

## INF と NAN

num が INF の場合は、<MAXNUMBER> エラーが生成されます。num が NAN の場合は、<ILLEGAL VALUE> エラーが生成されます。以下の例に、これらの無効な値を示します。

#### ObjectScript

```
SET i=$DOUBLE("INF")
SET n=$DOUBLE("NAN")
WRITE $DECIMAL(i),!
WRITE $DECIMAL(n)
```

## 例

以下の例では、既に InterSystems IRIS 形式になっている小数值に適用されたときに、\$DECIMAL が影響を与えないことを説明します。

## ObjectScript

```

SET x=$DECIMAL($ZPI)
SET y=$ZPI
IF x=y { WRITE !,"Identical:"
          WRITE !,"IRIS $DECIMAL: ",x
          WRITE !,"Native IRIS:   ",y }
ELSE { WRITE !,"Different:"
        WRITE !,"IRIS $DECIMAL: ",x
        WRITE !,"Native IRIS:   ",y }

```

以下の例では、pi の値が \$DOUBLE 値と標準の InterSystems IRIS 数値で返されます。この例では、\$DOUBLE 数と標準 InterSystems IRIS 数の間では、同等の演算は行われないことを示しています。また、その同等値は、IEEE を元の InterSystems IRIS に変換するための \$DECIMAL を使用してリストアすることはできません。

## ObjectScript

```

SET x=$DECIMAL($ZPI)
SET y=$DOUBLE($ZPI)
SET z=$DECIMAL(y)
IF x=y { WRITE !,"IRIS & IEEE Same" }
ELSEIF x=z { WRITE !,"IRIS & IEEE-to-IRIS same" }
ELSE { WRITE !,"All three different"
        WRITE !,"IRIS decimal: ",x
        WRITE !,"IEEE float:   ",y
        WRITE !,"IEEE to IRIS: ",z }

```

以下の例は、pi の \$DECIMAL 変換を \$DOUBLE 値で返します。これらの変換は、さまざまな n 引数値によって丸められます。

## ObjectScript

```

SET x=$DOUBLE($ZPI)
WRITE !,$DECIMAL(x)
/* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,1)
/* returns 3 */
WRITE !,$DECIMAL(x,8)
/* returns 3.1415927 (note rounding) */
WRITE !,$DECIMAL(x,12)
/* returns 3.14159265359 (note rounding) */
WRITE !,$DECIMAL(x,18)
/* returns 3.14159265358979312 */
WRITE !,$DECIMAL(x,19)
/* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,20)
/* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,21)
/* returns 3.141592653589793116 (19 digits) */
WRITE !,$DECIMAL(x,0)
/* returns 3.1415926535897931159 (20 digits) */

```

## 関連項目

- [ZZDUMP コマンド](#)
- [\\$DOUBLE 関数](#)
- [\\$FNUMBER 関数](#)
- [\\$NUMBER 関数](#)
- [インターシステムズ・アプリケーションでの数値の計算](#)
- [数値](#)
- [データ型](#)
- [演算子](#)

## \$DOUBLE (ObjectScript)

64 ビット浮動小数値に変換した数値 (具体的にはインターシステムズの [\\$DOUBLE 形式](#) の値) を返します。

### 構文

```
$DOUBLE (num)
```

### 引数

引数	説明
num	変換される数値。文字列 “NAN” と “INF” (およびそれらの異形) も指定できます。

### 説明

\$DOUBLE は、IEEE 倍精度 (64 ビット) バイナリ浮動小数点データ型に変換された数値 (これは、インターシステムズの [\\$DOUBLE 形式](#) としても知られ、DOUBLE および DOUBLE PRECISION [SQL データ型](#) に対応します) を返します。これは、[\\$DECIMAL](#) 関数で実行される処理の逆です。

\$DOUBLE 形式の数値には 20 桁まで格納できます。num が 20 桁より多い場合、\$DOUBLE は、小数部を適切な桁数に丸めます。num の整数部の有効桁が 20 より多い場合、\$DOUBLE は、整数部を 20 桁に丸め、追加の桁をゼロで表します。

**注釈** [\\$DOUBLE 形式](#) の小数は、通常その [10 進数](#) 変換とはわずかに異なります。“[インターシステムズ・アプリケーションでの数値の計算](#)” を参照してください。

**注釈** InterSystems IRIS 浮動小数点データ型 (“1E128” など) がサポートする最小/最大範囲を超える InterSystems IRIS 数値文字列リテラルは、自動的に IEEE 倍精度浮動小数点数に変換されます。この変換は数値リテラルに対してのみ実行され、算術演算の結果に対しては行われません。[%SYSTEM.Process](#) クラスの [TruncateOverflow\(\)](#) メソッドを使用すると、自動変換をプロセスごとに制御できます。システム全体の既定の動作は、[Config.Miscellaneous](#) クラスの [TruncateOverflow](#) プロパティで設定できます。

num 値は、数字または数値文字列で指定できます。\$DOUBLE 変換の前に、キャノン形式 (先頭と末尾にあるゼロを削除し、複数のプラス符号とマイナス符号に解決されるなど) に解決されます。数値でない文字列を num に指定すると 0 が返されます。混合数値文字列 (“7dwarves” または “7.5.4” など) を num に指定すると、数値でない文字が初めて現れた箇所まで入力値が切り捨てられ、その後、数値部が変換されます。[JSON 配列](#) または [JSON オブジェクト](#) に指定される \$DOUBLE 数値は、異なる検証規則と変換規則に従います。

### 等値比較と混合算術

\$DOUBLE で生成される数値は、正確には小数点以下の桁数と一致しないバイナリ表現に変換されるため、\$DOUBLE 値と [10 進数](#) 値の間での同等比較は予期しない結果を生じさせる場合があります、一般的にはこれを回避する必要があります。“[インターシステムズ・アプリケーションでの数値の計算](#)” を参照してください。

### 整数除算

特定の値の場合、[10 進](#) および [\\$DOUBLE 形式](#) の数値は、異なる整数除算結果を生成します。次に例を示します。

#### ObjectScript

```
WRITE !,"Divide operations:"
WRITE !,"IRIS   /: ",4.1/.01           // 410
WRITE !,"Double /: ", $DOUBLE(4.1)/.01 // 410
WRITE !,"Integer divide operations:"
WRITE !,"IRIS   \: ",4.1\0.01         // 410
WRITE !,"Double \: ", $DOUBLE(4.1)\.01 // 409
```

## リストの圧縮

ListFormat は、\$DOUBLE 形式の数値を、\$LIST エンコードされた文字列に格納する際に圧縮するかどうかを制御します。既定では、圧縮されません。圧縮された形式は、InterSystems IRIS によって自動的に処理されます。圧縮された形式がサポートされるかどうかを確認せずに、Java や C# などの外部クライアントに圧縮されたリストを渡さないでください。

%SYSTEM.Process クラスの ListFormat() メソッドを使用すると、プロセスごとの動作を制御できます。

Config.Miscellaneous クラスの ListFormat プロパティを設定するか、InterSystems IRIS の管理ポータルで [システム管理] から [構成]、[追加の設定]、[互換性] を選択して、システム全体の既定の動作を設定できます。

## INF と NAN

IEEE 標準に従って、\$DOUBLE は、文字列 INF (無限大) および NAN (非数値) を返します。INF は、正の数でも負の数でもかまいません(INF、-INF)。NAN は常に符号なしです。これらは有効な IEEE 返り値ですが、実際の数ではありません。

### 入力値としての INF と NAN

\$DOUBLE を INF と NAN に返すには、num 入力値と同等の文字列を指定します。これらの入力値は大文字と小文字を区別せず、先頭にプラス符号とマイナス符号を使用できます (INF は符号を解決し、NAN は符号を無視します)。NAN を返すには、“NAN”、“sNAN”、“+NAN”、“-NAN” を指定します。INF を返すには、“INF”、“+INF”、“Infinity” を指定します。-INF を返すには、“-INF”と “+-INF” を指定します。

### IEEEError

IEEEError は、解決できない数値変換に対する \$DOUBLE の応答方法を制御します。IEEEError が 0 に設定されている場合、\$DOUBLE は変換を解決できないときに INF と NAN を返します。IEEEError が 1 に設定されている場合、\$DOUBLE は変換を解決できないときに標準の InterSystems IRIS エラー・コードを生成します。既定値は 1 です。

%SYSTEM.Process クラスの IEEEError() メソッドを使用すると、プロセスごとの動作を制御できます。

Config.Miscellaneous クラスの IEEEError プロパティを設定するか、InterSystems IRIS の管理ポータルで [システム管理] から [構成]、[追加の設定]、[互換性] を選択して、システム全体の既定の動作を設定できます。

### INF と NAN の返り値

極端に大きな数字を指定したり、解決できない算術演算を指定すると、\$DOUBLE は INF と NAN を返します。これらの値が返されるのは、IEEEError が INF および NAN を返すように設定されている場合のみです。

極端に大きい浮動小数点数はサポートされません。\$DOUBLE のバイナリ浮動小数点数でサポートされる最大値は 1.7976931348623158079e308 です。\$DOUBLE のバイナリ浮動小数点数でサポートされる最小値は 1.0E-323 です。これより小さい num 値は 0 を返します。

注釈 InterSystems IRIS の 10 進数の浮動小数点数でサポートされる最大値は 9.223372036854775807e145 です。InterSystems IRIS の 10 進数の浮動小数点数でサポートされる最小値は 2.2250738585072013831e-308 (標準) または 4.9406564584124654417e-324 (非正規化) のいずれかです。

以下のテーブルは、返り値または解決できない算術演算によって生成されたエラーを示します。

入力値	IEEEError=0	IEEEError=1
> 1.0E308	INF	<MAXNUMBER>
< 1.0E-323	0	0
1/\$DOUBLE(0)	INF	<DIVIDE>
1/\$DOUBLE(-0)	-INF	<DIVIDE>
\$DOUBLE(1)/0	INF	<DIVIDE>
\$DOUBLE(0)/0	NAN	<ILLEGAL VALUE>
\$ZLOG(\$DOUBLE(0))	-INF	<DIVIDE>

## INF と NAN の比較

INF は数値として比較できます。例えば、 $INF = INF$ 、 $INF' = -INF$ 、 $-INF = -INF$ 、 $INF > -INF$  となります。

NAN は数値として比較できません。NAN (非数値) は数値演算子を使用して有意義に比較できないため、\$DOUBLE("NAN") と別の \$DOUBLE("NAN") を比較しようとする InterSystems IRIS 演算 (より大きい、等しい、より小さいなど) は失敗します。NAN <= または >= での比較は特殊なケースです。これについては、「インターシステムズ・アプリケーションでの数値の計算」を参照してください。

[\\$LISTSAME](#) は \$DOUBLE("NAN") リスト要素を別の \$DOUBLE("NAN") リスト要素と同一と見なします。

InterSystems IRIS は、異なる NAN 表現 (NAN、sNAN など) を区別しません。NAN のバイナリ表現に関係なく、すべての NAN を同一と見なします。

## \$ISVALIDNUM、\$INUMBER、および \$FNUMBER

これらの ObjectScript 関数は、\$DOUBLE 数をサポートしています。

[\\$ISVALIDNUM](#) は、INF および NAN をサポートしています。これらの文字列は数字ではありませんが、\$ISVALIDNUM 関数は、これらの値に対して数字として 1 を返します。\$DOUBLE が非数値文字列 (\$DOUBLE("") など) で指定されている場合、InterSystems IRIS は 0 の値を返します。このため、0 は数字なので、\$ISVALIDNUM(\$DOUBLE("")) は 1 を返します。

[\\$INUMBER](#) および [\\$FNUMBER](#) は、\$DOUBLE 値をサポートする "D" 形式オプションを提供します。[\\$INUMBER](#) は、数値を IEEE 浮動小数点数に変換します。[\\$FNUMBER](#) "D" のサポートは、INF および NAN の大文字と小文字の変換を含み、\$DOUBLE(-0) が 0 と -0 のどちらを返すかを選択します。

## 演算子を使用した INF と NAN

INF と NAN に対して算術演算子と論理演算子を実行できます。INF と NAN と共に演算子を使用することはお勧めしません。このような演算を実行すると、以下のような結果になります。

算術演算子：

加算	減算	乗算	除算 (/、¥、または # 演算子)
NAN+NAN=NAN	NAN-NAN=NAN	NAN*NAN=NAN	NAN/NAN=NAN
NAN+INF=NAN	NAN-INF=NAN	NAN*INF=NAN	NAN/INF=NAN
	INF-NAN=NAN		INF/NAN=NAN
INF+INF=INF	INF-INF=NAN	INF*INF=INF	INF/INF=NAN

論理演算子：

等しい (=)	NAN	INF
NAN	0	0
INF	0	1

より小さい (<)/より大きい (>)	NAN	INF
NAN	0	0
INF	0	0

パターン・マッチングや連結演算子などの他の演算子は、3 文字のアルファベット文字列として NAN と INF を扱います。

詳細は、“[インターシステムズ・アプリケーションでの数値の計算](#)” を参照してください。

## INF と NAN の例

以下の例に示すように、数値が使用可能な精度を超えた場合、\$DOUBLE は INF 値（負数に対しては -INF）を返します。

### ObjectScript

```
SET rtn=##class(%SYSTEM.Process).IEEERError(0)
SET x=$DOUBLE(1.2e300)
WRITE !,"Double: ",x
WRITE !,"Is number? ",$ISVALIDNUM(x)
SET y= $DOUBLE(x*x)
WRITE !,"Double squared: ",y
WRITE !,"Is number? ",$ISVALIDNUM(y)
```

数値が無効な場合、\$DOUBLE は NAN (非数値) 値を返します。例えば、以下の例に示すように、算術式に 2 つの INF 値が含まれる場合です (1 つの INF 値を含む算術式は INF を返します)。

### ObjectScript

```
SET rtn=##class(%SYSTEM.Process).IEEERError(0)
SET x=$DOUBLE(1.2e500)
WRITE !,"Double: ",x
WRITE !,"Is number? ",$ISVALIDNUM(x)
SET y= $DOUBLE(x-x)
WRITE !,"Double INF minus INF: ",y
WRITE !,"Is number? ",$ISVALIDNUM(y)
```

## JSON 数値リテラル

数値リテラルの JSON 検証については、[SET](#) コマンドで説明しています。JSON 配列または JSON オブジェクトで指定された \$DOUBLE 数値リテラルは、以下の追加規則に従います。

- INF 値、-INF 値、および NAN 値は、JSON 構造に格納できますが、%ToJSON() によって返すことはできません。これを実行しようとすると、以下の例に示すような、<ILLEGAL VALUE> エラーが返されます。

### ObjectScript

```
SET jary=[123,($DOUBLE("INF"))] // executes successfully
WRITE jary.%ToJSON()           // fails with <ILLEGAL VALUE> error
```

- \$DOUBLE(-0) は JSON 構造に -0.0 として格納されます。\$DOUBLE(0) または \$DOUBLE(+0) は JSON 構造に 0.0 として格納されます。以下に例を示します。

## ObjectScript

```
SET jary=[0,-0,($DOUBLE(0)),($DOUBLE(-0))]  
WRITE jary.%ToJSON() // returns [0,-0,0.0,-0.0]
```

## 例

以下の例は、20 桁の浮動小数点数を返します。

### ObjectScript

```
WRITE !,$DOUBLE(999.12345678987654321)  
WRITE !,$DOUBLE(.99912345678987654321)  
WRITE !,$DOUBLE(999123456789.87654321)
```

以下の例では、pi の値が \$DOUBLE 値と標準の InterSystems IRIS 数値で返されます。この例では、\$DOUBLE 数と標準 InterSystems IRIS 数の間で等値演算を行うべきでないこと、および標準 InterSystems IRIS 数について返される桁数の方が大きくなることを示しています。

### ObjectScript

```
SET x=$ZPI  
SET y=$DOUBLE($ZPI)  
IF x=y { WRITE !,"Same" }  
ELSE { WRITE !,"Different"  
      WRITE !,"standard: ",x  
      WRITE !,"IEEE float: ",y }
```

以下の例では、浮動小数点数は、同じ値の数値文字列と同等である必要がないことを示しています。

### ObjectScript

```
SET x=123.4567891234560  
SET y=123.4567891234567  
IF x=$DOUBLE(x) { WRITE !,"Same" }  
ELSE { WRITE !,"Different" }  
IF y=$DOUBLE(y) { WRITE !,"Same" }  
ELSE { WRITE !,"Different" }
```

### ObjectScript

```
SET x=1234567891234560  
SET y=1234567891234567  
IF x=$DOUBLE(x) { WRITE !,"Same" }  
ELSE { WRITE !,"Different" }  
IF y=$DOUBLE(y) { WRITE !,"Same" }  
ELSE { WRITE !,"Different" }
```

## 関連項目

- [ZZDUMP](#) コマンド
- [\\$DECIMAL](#) 関数
- [\\$FNUMBER](#) 関数
- [\\$NUMBER](#) 関数
- [インターシステムズ・アプリケーションでの数値の計算](#)
- [データ型](#)
- [演算子](#)



## \$EXTRACT (ObjectScript)

指定された位置にある部分文字列を文字列から取り出します。または指定された位置にある部分文字列を置換します。

### 構文

```
$EXTRACT(string,from,to)
$E(string,from,to)

SET $EXTRACT(string,from,to)=value
SET $E(string,from,to)=value
```

### 引数

引数	説明
string	部分文字列が識別されるターゲット文字列。string には、評価結果が引用符で囲んだ文字列または数値になる式を指定します。SET \$EXTRACT 構文では、string は変数または多次元プロパティにする必要があります。
from	オプション - 対象の string 内の開始位置を指定します。文字は 1 からカウントされます。許可される値は、n (string の先頭からの文字カウントを指定する正の整数)、* (string の末尾文字の指定)、および *-n (string の末尾から逆向きの文字のオフセット整数カウント) です。SET \$EXTRACT 構文は、*+n (string の末尾の先に追加する文字のオフセット整数カウント) もサポートします。to なしの from は、単一の文字を指定します。to ありの from は、文字の範囲を指定します。from が指定されていない場合、既定値は 1 です。
to	オプション - 文字の範囲の終了位置 (その文字を含む) を指定します。from と共に使用する必要があります。許可される値は、n (string の先頭からの文字カウントを指定する正の整数)、* (string の末尾文字の指定)、および *-n (string の末尾から逆向きの文字のオフセット整数カウント) です。SET \$EXTRACT 構文は、*+n (string の末尾の先に追加する文字範囲の終了位置のオフセット整数カウント) もサポートします。

### 概要

\$EXTRACT は、string の先頭または string の末尾からの文字カウントで、string 内の部分文字列を識別します。部分文字列には、1 文字または文字の範囲を指定できます。

\$EXTRACT には以下の 2 つの使用方法があります。

- string の部分文字列を返します。これには構文 \$EXTRACT(string,from,to) が使用されます。
- string の部分文字列を置換します。置換部分文字列は、元の部分文字列と同じ長さでも、長くても、短くてもかまいません。これには構文 SET \$EXTRACT(string,from,to)=value が使用されます。

### 部分文字列を返す方法

\$EXTRACT は string から文字位置により部分文字列を返します。この部分文字列の抽出の特性は、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。

#### ObjectScript

```
SET mystr="ABCD"
WRITE $EXTRACT(mystr)
```

- ・ \$EXTRACT(string,from) は、from で指定した位置の 1 文字を抽出します。from の値は、文字列の最初からカウントした整数、文字列の最後の文字を指定するアスタリスク、または、文字列の最後から逆方向にカウントして指定する負の整数を持つアスタリスクになります。

以下の例では、文字列 “ABCD” から 1 つの文字を抽出します。

#### ObjectScript

```
SET mystr="ABCD"
WRITE !,$EXTRACT(mystr,2)      // "B" the 2nd character
WRITE !,$EXTRACT(mystr,*)      // "D" the last character
WRITE !,$EXTRACT(mystr,*-2)    // "B" the offset 2 characters from end
WRITE !,$EXTRACT(mystr,*-0)    // "D" the last character by 0 offset
```

- ・ \$EXTRACT(string,from,to) は、from の位置から始まり to の位置で終了する文字列の範囲（その位置の文字を含む）を抽出します。例えば、変数 var2 が文字列 “1234Alabama567” を含むときは、以下の \$EXTRACT 関数は両方とも文字列 “Alabama” を返します。

#### ObjectScript

```
SET var2="1234Alabama567"
WRITE !,$EXTRACT(var2,5,11)
WRITE !,$EXTRACT(var2,*-9,*-3)
```

## 引数

### string

部分文字列が識別されるターゲット文字列。

**部分文字列を返す**ために \$EXTRACT を使用する場合、string は、引用符で囲まれた文字列リテラル、キャノニック形式の数値、変数、オブジェクト・プロパティ、または評価結果が文字列または数値になる有効な ObjectScript 式にできます。ターゲット文字列として NULL 文字列 (") を指定すると、\$EXTRACT はその他の引数値に関係なく、常に NULL 文字列を返します。

**1 つの部分文字列を置換する**ために、等号の左側で SET と共に \$EXTRACT を使用する場合、string は変数名または **多次元プロパティ**参照にすることができますが、非多次元オブジェクト・プロパティにすることはできません。

### from

from 引数は、単独の文字または文字の範囲の先頭を指定できます。

- ・ from が n (正の整数) の場合、\$EXTRACT は string の先頭から文字をカウントします。
- ・ from が \* (アスタリスク) の場合、\$EXTRACT は string 内の末尾の文字を返します。
- ・ from が \*-n (アスタリスクとそれに続く負の整数) の場合、\$EXTRACT は string の末尾からオフセット分だけ文字をカウントします。つまり、\*-0 は string 内の末尾の文字になり、\*-1 は string 内の最後から 2 番目の文字 (末尾からのオフセットが 1) になります。
- ・ SET \$EXTRACT 構文のみ - from が \*+n (アスタリスクとそれに続く正の整数) の場合、SET \$EXTRACT は string の末尾の先にオフセット分の文字を追加します。つまり、\*+1 の場合は string の末尾の先に 1 つの文字が追加され、\*+2 の場合は string の末尾の 2 つ先の位置に 1 つの文字が追加されます (末尾と追加位置の間には空白スペースが埋め込まれます)。\*+0 は string 内の最後の文字です。

from の整数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 に \*-n 値を指定したときに、n が文字列の文字数以上になる場合、\$EXTRACT は NULL 文字列を返します。from の値が 0 または負の数の場合、\$EXTRACT は NULL 文字列を返します。ただし、from が to と共に使用されていると、0 または負の数になる from の値は、値 1 と見なされます。

from を to 引数と共に使用するときは、from は抽出する範囲の先頭を識別します。また、to の値よりも小さく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 引数と共に使用するときは、1 未満の from の値 (ゼロか負の数) は 1 として扱われます。

## to

to 引数は、from 引数と共に使用する必要があります。正の整数、\* (アスタリスク)、または \*-n (アスタリスクとそれに続く負の整数) でなければなりません。to の値が from 以上の整数である場合、\$EXTRACT は指定した部分文字列を返します。to の値がアスタリスクのときは、\$EXTRACT は、from 文字から文字列の最後までを含む部分文字列を返します。to が文字列の長さより大きい整数の場合も、\$EXTRACT は、from 文字から文字列の最後までの部分文字列を返します。

from と to の位置が同じ場合、\$EXTRACT は単一の文字を返します。to の位置が from の位置よりも文字列の先頭に近い場合、\$EXTRACT は NULL 文字列を返します。

to 引数を省略すると、1 文字のみが返されます。from を指定すると、\$EXTRACT は from により識別される文字を返します。to と from の両方を省略すると、\$EXTRACT は string の最初の文字を返します。

SET \$EXTRACT 構文のみ -to が \*+n の場合、SET \$EXTRACT は string の末尾の先にオフセット分の文字の範囲を追加します (必要に応じて、空白スペースが埋め込まれます)。from が string の末尾の後の文字位置を示している場合、SET \$EXTRACT は文字を追加します。from が string の末尾の前の文字位置を示している場合、SET \$EXTRACT は文字の置換と追加の両方を実行します。

## \*-n および \*+n 引数値の指定

変数を使用して \*-n または \*+n を指定する場合は、常に、引数自体でアスタリスクと符号文字を指定する必要があります。

\*-n の有効な指定内容は以下のとおりです。

### ObjectScript

```
SET count=2
SET alph="abcd"
WRITE $EXTRACT(alph,*-count)
```

### ObjectScript

```
SET count=-2
SET alph="abcd"
WRITE $EXTRACT(alph,*+count)
```

\*+n の有効な指定内容は以下のとおりです。

### ObjectScript

```
SET count=2
SET alph="abcd"
SET $EXTRACT(alph,*+count)="F"
WRITE alph
```

これらの引数値内では、空白を使用できます。

## 例：部分文字列を返す方法

以下の例は文字列の 4 番目の文字 “D” を返します。

### ObjectScript

```
SET x="ABCDEFGHIJK"
WRITE $EXTRACT(x,4)
```

以下の例は文字列の最後の文字 “K” を返します。

#### ObjectScript

```
SET x="ABCDEFGHJK"
WRITE $EXTRACT(x,*)
```

以下の例では、\$EXTRACT 関数は、すべて文字列の最後から 2 番目の文字 “J” を返します。

#### ObjectScript

```
SET n=-1
SET m=1
SET x="ABCDEFGHJK"
WRITE !,$EXTRACT(x,*-1)
WRITE !,$EXTRACT(x,*-m)
WRITE !,$EXTRACT(x,*+n)
WRITE !,$EXTRACT(x,*-1,*-1)
```

アスタリスクと整数変数の間には、マイナス記号かプラス記号が必要です。

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。

#### ObjectScript

```
SET x="HELLO"
WRITE !,$EXTRACT(x)
WRITE !,$EXTRACT(x,1)
```

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。

#### ObjectScript

```
SET x="THIS IS A TEST"
WRITE $EXTRACT(x,1,7)
```

以下の例も、部分文字列 “THIS IS” を返します。from 変数が 1 未満の値を含むとき、\$EXTRACT はその値を 1 として処理します。以下の例は、先頭から 7 番目の文字までの部分文字列を返します。

#### ObjectScript

```
SET X="THIS IS A TEST"
WRITE $EXTRACT(X,-1,7)
```

以下の例は文字列の最後の 4 文字を返します。

#### ObjectScript

```
SET X="THIS IS A TEST"
WRITE $EXTRACT(X,*-3,*)
```

また、以下の例は文字列の最後の 4 文字も返します。

#### ObjectScript

```
SET X="THIS IS A TEST"
WRITE $EXTRACT(X,*-3,14)
```

以下の例では、オブジェクト・プロパティから部分文字列を抽出します。

## ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()  
SET tStatement.%SchemaPath="MyTests,Sample,Cinema"  
WRITE "whole schema path: ",tStatement.%SchemaPath,!  
WRITE "start of schema path: ",$EXTRACT(tStatement.%SchemaPath,1,10),!
```

## SET \$EXTRACT を使用した部分文字列の置換

\$EXTRACT を SET コマンドと一緒に使用して、指定した文字または文字列の範囲を別の値で置き換えることができます。また、文字列の末尾に文字を追加する場合にも使用できます。

等号の左側で SET と共に \$EXTRACT を使用する場合、string は有効な変数名にすることができます。変数が存在しない場合は、SET \$EXTRACT が変数を定義します。string 引数は、[多次元プロパティ](#)参照にすることもできますが、非多次元オブジェクト・プロパティにすることはできません。非多次元オブジェクト・プロパティで SET \$EXTRACT を使用しようとすると、<OBJECT DISPATCH> エラーが発生します。

\$EXTRACT (または \$PIECE や \$LIST) を伴った SET (a,b,c,...)=value 構文は、その関数で相対オフセット構文 \* (文字列の末尾を表す)、\*-n または \*+n (文字列の末尾からの相対オフセットを表す) を使用する場合、等号の左側では使用できません。その代わりに、SET a=value,b=value,c=value,... 構文を使用する必要があります。

SET \$EXTRACT のうちで最も単純な形式は一对一の置き換えです。

## ObjectScript

```
SET alph="ABZD"  
SET $EXTRACT(alph,3)="C"  
WRITE alph ; "ABCD"
```

string に文字を追加するには、to に string の長さより 1 大きい正の整数を指定するか、to に \*+1 を指定します。以下に、この例を示します。

## ObjectScript

```
SET alph="ABCD"  
SET $EXTRACT(alph,5)="E"  
WRITE alph ; "ABCDE"
```

## ObjectScript

```
SET alph="ABCD"  
SET $EXTRACT(alph,*+1)="E"  
WRITE alph ; "ABCDE"
```

文字列プラス 1 よりも大きい to を指定すると、\$EXTRACT は空白スペースを埋め込みます。

## ObjectScript

```
SET alph="ABCD"  
SET len=$LENGTH(alph)  
SET $EXTRACT(alph,len+2)="F"  
WRITE alph ; "ABCD F"
```

## ObjectScript

```
SET alph="ABCD"  
SET $EXTRACT(alph,*+2)="F"  
WRITE alph ; "ABCD F"
```

文字列を抽出して、異なる長さの文字列と置き換えることも可能です。例えば、以下のコマンドは foo から文字列 "Rhode Island" を抽出して、埋め込みなしで文字列 "Texas" に置き換えます。

### ObjectScript

```
SET foo="Deep in the heart of Rhode Island"
SET $EXTRACT(foo,22,33)="Texas"
WRITE foo      ; "Deep in the heart of Texas"
```

文字列を抽出して、その文字列から抽出された文字を削除しながら、NULL 文字列に設定できます。

### ObjectScript

```
SET alph="ABCzzzzzD"
SET $EXTRACT(alph,4,8)=" "
WRITE alph      ; "ABCD"
```

to より長い from を指定すると、置換は実行されません。

### ObjectScript

```
SET alph="ABCD"
SET $EXTRACT(alph,4,3)="X"
WRITE alph      ; "ABCD"
```

以下の例では、変数 x が存在しないものとします。

### ObjectScript

```
KILL x
SET $EXTRACT(x,1,4)="ABCD"
WRITE x      ; "ABCD"
```

SET コマンドは、変数 x を作成して、値 "ABCD" を代入します。

SET \$EXTRACT は、必要に応じて空白スペースによる先頭の埋め込みを実行しますが、末尾の埋め込みは実行しません。以下の例では、文字列の末尾より先の 6 番目の位置に値 "F" を挿入していますが、7 番目と 8 番目の位置には追加の文字を挿入していません。

### ObjectScript

```
SET alph="ABCD"
SET $EXTRACT(alph,6,8)="F"
WRITE alph      ; "ABCD F"
```

以下の例では、6 番目の位置に値 "F" を挿入して、指定した範囲の後に文字を追加しています。

### ObjectScript

```
SET alph="ABCD"
SET $EXTRACT(alph,6,8)="FGHIJ"
WRITE alph      ; "ABCD FGHIJ"
```

以下の例は、置き換える文字列の値の数よりも長い from,to 範囲を抽出することにより文字列を短くします。

### ObjectScript

```
SET x="ABCDEFGH"
SET $EXTRACT(x,3,6)="Z"
WRITE x
```

これは、3 番目に値 "Z" を挿入して、位置 4 と 5 と 6 の文字を削除します。変数 x の値は "ABZGH" になり、5 文字列長になります。

## \$EXTRACT と Unicode

\$EXTRACT 関数は、バイトではなく、文字を処理します。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 "_$CHAR(960)
SET a1=$EXTRACT(a,-33,4)
SET a2=$EXTRACT(a,4,4)
SET a3=$EXTRACT(a,4,99)
SET b1=$EXTRACT(b,-33,4)
SET b2=$EXTRACT(b,4,4)
SET b3=$EXTRACT(b,4,99)
WRITE !,"ASCII form returns ",!,a1,!,a2,!,a3
WRITE !,"Unicode form returns ",!,b1,!,b2,!,b3
```

詳細は“Unicode”を参照してください。

## サロゲート・ペア

\$EXTRACT は、サロゲート・ペアを認識しません。サロゲート・ペアは、一部の中国語の文字を表示したり、日本語の JIS2004 標準をサポートするために使用されます。\$WISWIDE 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。\$WEXTRACT 関数は、サロゲート・ペアを認識して、正しく解析します。\$EXTRACT と \$WEXTRACT は、それ以外は同一です。ただし、\$EXTRACT は通常 \$WEXTRACT より高速なため、サロゲート・ペアが出現しない場合は常に \$EXTRACT が推奨されます。

## \$PIECE と \$LIST と比較した \$EXTRACT コマンド

\$EXTRACT は、文字列の最初の部分から文字をカウントすることにより、部分文字列を決定します。\$EXTRACT は、入力として通常の任意の文字列を取ります。\$PIECE と \$LIST は両方とも、特別に用意された文字列を処理します。

\$PIECE は、文字列内のユーザ定義の区切り文字をカウントすることにより、部分文字列を決定します。

\$LIST は、リストの最初から要素（文字ではない）をカウントすることにより、エンコードされたリストから要素を決定します。\$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。

## 関連項目

- [SET コマンド](#)
- [\\$FIND 関数](#)
- [\\$LENGTH 関数](#)
- [\\$PIECE 関数](#)
- [\\$REVERSE 関数](#)
- [\\$WEXTRACT 関数](#)
- [\\$WFIND 関数](#)
- [\\$WISWIDE 関数](#)



# \$FACTOR (ObjectScript)

整数を \$BIT ビット文字列に変換します。

## 構文

```
$FACTOR(num, scale)
```

## 引数

引数	説明
num	数字として評価される式。num は、正の整数に変換されてから、ビット文字列に変換されます。負の整数は、正の整数（その絶対値）に変換されます。小数点以下を持つ数値は整数に丸められます。
scale	オプション - num に対する 10 のべき乗指数（科学的記数法）として使用する整数。既定は 0 です。

## 概要

\$FACTOR は、指定された整数の 2 進数表記に対応する \$BIT 形式のビット文字列を返します。これは、以下の操作を実行します。

- ・ 負の数を指定すると、\$FACTOR はその数の絶対値を取ります。
- ・ scale を指定すると、\$FACTOR は整数を  $10^{scale}$  で乗算します。
- ・ 小数を指定すると、\$FACTOR はこの数を整数に丸めます。数を丸めるときは、四捨五入が使用され小数の .5 は次に高い整数になります。
- ・ \$FACTOR は整数を、その 2 進数表記に変換します。
- ・ \$FACTOR はこの 2 進数を、\$BIT でエンコードされたバイナリ形式に変換します。

返されたバイナリ文字列は、位置 1 の最下位ビットを起点としてビット位置を指定します（位置 1 を 1 とします）。これは、さまざまな \$BIT 関数で使用するビット文字列に対応しています。

## 引数

### num

数（または数に評価する式）。\$FACTOR は scale 引数を適用し（指定されている場合）、この数を四捨五入によって整数に変換してから、対応するビット文字列を返します。num は正の数でも負の数でもかまいません。num が混合数値文字列（“7dwarves” または “5.6.8” など）の場合、\$FACTOR は、非数値文字が検出されるまで文字列の数値部分（7 および 5.6）を変換します。num がゼロの場合、ゼロに丸められた場合、NULL 文字列（”）の場合、または非数値文字列の場合、\$FACTOR は空の文字列を返します。\$DOUBLE の INF 値、-INF、および NAN は空の文字列を返します。

### scale

num に適用する、科学的記数法のべき乗指数を指定する整数。例えば、scale が 2 の場合、scale は 10 の 2 乗、つまり 100 を表し、この scale 値と num が乗算されます。例えば、\$FACTOR(7,2) は整数 700 に対応するビット文字列を返します。この乗算は、num を整数に丸める前に実行されます。既定では、scale は 0 です。

## 例

以下は、整数の 1 から 9 まではビット文字列に変換する例です。

## ObjectScript

```
SET x=1
WHILE x<10 {
  WRITE !,x,"="
  FOR i=1:1:8 {
    WRITE $BIT($FACTOR(x),i) }
  SET x=x+1 }

```

以下は、負の数と小数の正の整数への \$FACTOR 変換の例です。

## ObjectScript

```
FOR i=1:1:8 {WRITE $BIT($FACTOR(17),i)}
WRITE " Positive integer",!
FOR i=1:1:8 {WRITE $BIT($FACTOR(-17),i)}
WRITE " Negative integer (absolute value)",!
FOR i=1:1:8 {WRITE $BIT($FACTOR(16.5),i)}
WRITE " Positive fraction (rounded up)",!
FOR i=1:1:8 {WRITE $BIT($FACTOR(-16.5),i)}
WRITE " Negative fraction (rounded up)"

```

以下は、scale 引数が指定された場合に返されるビット文字列の例です。

## ObjectScript

```
SET x=2.7
WRITE !,x," scaled then rounded to an integer:",!!
FOR i=1:1:12 {
  WRITE $BIT($FACTOR(x),i) }
WRITE " binary = ", $NORMALIZE(x,0)," decimal",!
SET scale=1
SET y=x*(10**scale)
FOR i=1:1:12 {
  WRITE $BIT($FACTOR(x,scale),i) }
WRITE " binary = ", $NORMALIZE(y,0)," decimal",!
SET scale=2
SET y=x*(10**scale)
FOR i=1:1:12 {
  WRITE $BIT($FACTOR(x,scale),i) }
WRITE " binary = ", $NORMALIZE(y,0)," decimal"

```

## 関連項目

- ・ [\\$BIT 関数](#)
- ・ [\\$BITCOUNT 関数](#)
- ・ [\\$BITFIND 関数](#)
- ・ [\\$BITLOGIC 関数](#)
- ・ [\\$DOUBLE 関数](#)

## \$FIND (ObjectScript)

値により部分文字列を検索し、文字列の最終位置を指定する整数を返します。

### 構文

```
$FIND(string,substring,position)
$F(string,substring,position)
```

### 引数

引数	説明
string	検索されるターゲット文字列。変数名、数値、文字列リテラル、または文字列に解決される有効な ObjectScript 式を指定できます。
substring	検索される部分文字列。変数名、数値、文字列リテラル、または文字列に解決される有効な ObjectScript 式を指定できます。
position	オプション - 検索を開始するターゲット文字列内の位置。必ず正の整数になります。

### 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、string で substring を検索します。\$FIND は、大文字と小文字を区別します。substring が見つかり、\$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。

\$FIND は、substring に続く文字の位置を返すため、substring が string の最初の文字に一致する 1 つの文字である場合、\$FIND は 2 を返します。substring が NULL 文字列("") である場合、\$FIND は 1 を返します。

position オプションを使用して、検索の開始位置を指定できます。position 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。

\$FIND は、バイトではなく、文字をカウントします。したがって、8 ビットまたは 16 ビット (Unicode) 文字を含む文字列で使用できます。InterSystems IRIS Unicode サポートの詳細は、“[Unicode](#)” を参照してください。

### 例

例えば、変数 var1 が文字列 “ABCDEFGH”、変数 var2 が文字列 “BCD” を含む場合、以下の \$FIND は var2 文字列に続く文字 (“E”) の位置を示す値 5 を返します。

#### ObjectScript

```
SET var1="ABCDEFGH",var2="BCD"
WRITE $FIND(var1,var2)
```

以下の例は、部分文字列 “FOR” のすぐ右にある文字位置 4 を返します。

#### ObjectScript

```
SET X="FOREST"
WRITE $FIND(X,"FOR")
```

以下の例では、\$FIND は string に含まれていない部分文字列、NULL サブ文字列、および string の最初の文字であるサブ文字列を検索します。例では、それぞれ 0、1、および 2 が返されます。

## ObjectScript

```
WRITE !,$FIND("aardvark","z") ; returns 0
WRITE !,$FIND("aardvark","") ; returns 1
WRITE !,$FIND("aardvark","a") ; returns 2
```

以下の例は、string が NULL 文字列である場合に何が起こるかを示しています。

## ObjectScript

```
WRITE !,$FIND("", "z") ; returns 0
WRITE !,$FIND("", "") ; returns 1
```

次の例は、X の 7 番目の文字の後で、最初に "R" が現れる位置のすぐ右の文字位置 14 を返します。

## ObjectScript

```
SET X="EVERGREEN FOREST",Y="R"
WRITE $FIND(X,Y,7)
```

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。結果として 0 を返します。

## ObjectScript

```
SET X="EVERGREEN FOREST",Y="R"
WRITE $FIND(X,Y,20)
```

以下の例では、\$FIND を \$REVERSE と共に使用して、文字列の最後から検索操作を実行します。この例では、テキスト行内の最後の検索対象文字列が検出され、その文字列の位置として 33 が返されます。

## ObjectScript

```
SET line="THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."
SET position=$LENGTH(line)+2-$FIND($REVERSE(line),$REVERSE("THE"))
WRITE "The last THE in the line begins at ",position
```

次の例は、名前の間接指定を使用して部分文字列 "THIS" のすぐ右の文字位置 6 を返しています。

## ObjectScript

```
SET Y="x",X="" "THIS IS A TEST" ""
WRITE $FIND(@Y,"THIS")
```

詳細は、[間接 \(@\)](#) のリファレンス・ページを参照してください。

## \$FIND、\$EXTRACT、\$PIECE、および \$LIST

- ・ \$FIND は、値により部分文字列を検索し、位置を返します。
- ・ \$EXTRACT は、位置により部分文字列を検索し、部分文字列の値を返します。
- ・ \$PIECE は、区切り文字または区切り文字列により部分文字列を検索し、部分文字列の値を返します。
- ・ \$LIST は、特別にエンコードされた文字列を操作します。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。

\$FIND、\$EXTRACT、\$LENGTH、および \$PIECE 関数は、標準の文字列を操作します。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、入力値としてエンコードされた文字列を取りながら、標準の文字列として 1 つの要素の値を出力する、\$LIST の 1-引数形式と 2-引数形式です。

## サロゲート・ペア

\$FIND は、サロゲート・ペアを認識しません。サロゲート・ペアは、一部の中国語の文字を表示したり、日本語の JIS2004 標準をサポートするために使用されます。\$WISWIDE 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。\$WFIND 関数は、サロゲート・ペアを認識して、正しく解析します。\$FIND と \$WFIND は、それ以外は同一です。ただし、\$FIND は通常 \$WFIND より高速なため、サロゲート・ペアが出現しない場合は常に \$FIND が推奨されます。

## 関連項目

- ・ [\\$EXTRACT](#) 関数
- ・ [\\$LENGTH](#) 関数
- ・ [\\$LIST](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$REVERSE](#) 関数
- ・ [\\$WEXTRACT](#) 関数
- ・ [\\$WFIND](#) 関数
- ・ [\\$WISWIDE](#) 関数

## \$FNUMBER (ObjectScript)

指定された形式で数値をフォーマットします。オプションで、指定された精度まで丸めるか、またはゼロで埋めます。

### 構文

```
$FNUMBER(inumber,format,decimal)
$FN(inumber,format,decimal)
```

### 引数

引数	説明
<i>inumber</i>	フォーマットされる数値。数値リテラル、変数、数値に評価される有効な ObjectScript 式を指定できます。
<i>format</i>	オプション - 数のフォーマット方法を指定します。任意の順序の 0 以上の形式コードで構成される引用符付きの文字列として指定されます。形式コードの詳細は後述します。一部の形式コードは互換性がなく、エラーが生じます。 <i>decimal</i> 引数の有無によらず、既定の形式には、空の文字列 ( ) を指定できます。省略した場合、既定は空の文字列 ("" ) です。
<i>decimal</i>	オプション - 返される数に含まれる小数桁数。 <i>format</i> を省略する場合、 <i>decimal</i> を指定する前にプレースホルダのコンマを入れます。

### 概要

\$FNUMBER は、指定された *format* で、*inumber* により指定される数値を返します。

### 引数

#### *inumber*

数値として評価する式。\$FNUMBER が演算を実行する前に、InterSystems IRIS は、以下のように *inumber* に対して標準の数値解決を実行します。つまり、変数を解決し、連結などの文字列操作を実行し、文字列を数値に変換し、数値式演算を実行し、結果として得られた数値を**キャノニック形式**に変換します。これは、\$FNUMBER がフォーマットする数値です。

*inumber* が文字列である場合、InterSystems IRIS ではまず数値に変換され、最初の非数値文字で切り捨てが行われます。文字列の最初の文字が非数値文字の場合、InterSystems IRIS は文字列を 0 に変換します。

#### *format*

利用可能な形式コードは、以下のとおりです。単独でも、組み合わせても指定できます。アルファベット・コードでは、大文字と小文字は区別されません。

コード	説明
""	空の文字列。 <i>inumber</i> を <b>キャノニック形式の数値</b> で返します。この形式は、“L” 形式と同じです。
+	現在のロケールの PlusSign プロパティを接頭語とした非負数を返します (既定は “+”)。数が負のとき、現在のロケールの MinusSign プロパティを接頭語とした数を返します (既定は “-”)。
-	数字の絶対値を返します。常に MinusSign 文字なしの負の数値を返します。PlusSign 文字なしの正の数値を返します。“+” 形式コードと組み合わせると (“-+”)、正の数値はプラス記号付きで、負の数値は記号なしで返します。このコードは “P” 形式コードと一緒に使用できません。併用しようとすると、<SYNTAX> エラーが返されます。

コード	説明
,	現在のロケールの <a href="#">NumericGroupSeparator</a> プロパティを、小数点の左側にある整数に <a href="#">NumericGroupSize</a> ごとに入れた数値を返します。“,” を “.” または “N” と組み合わせた形式は、<FUNCTION> エラーになります。
.	現在のロケール設定に関係なく、標準的なヨーロッパ形式を使用して数字を返します。DecimalSeparator はコンマ (,)、NumericGroupSeparator はピリオド (.)、NumericGroupSize は 3、PlusSign はプラス記号 (+)、MinusSign はマイナス記号 (-) に設定されます。“.” を “,” または “O” と組み合わせた形式は、<FUNCTION> エラーになります。
D	<p>\$DOUBLE 特殊形式。このコードには以下の 2 つの効果があります。</p> <p>“D” は、\$DOUBLE(-0) が -0 を返すことを指定します。それ以外の場合、\$DOUBLE(-0) は 0 を返します。ただし、“-D” は負符号を上書きし、0 を返します。</p> <p>このコードでは、“D” または “d” を指定できます。返される INF または NAN は、これに対応して大文字または小文字で表されます。既定では大文字となります。</p>
E	E 記数法 ( <a href="#">科学的記数法</a> )。科学的記数法で数を返します。小数桁の decimal 数を省略すると、既定で 6 が使用されます。このコードでは、“E” または “e” を指定できます。返される値には、これに対応して大文字または小文字の記号が含まれます。返される値の指数部分は、指数に 3 桁を必要としない限り 2 桁の長さで、先頭に符号が付きます。“E” と “G” は互換性がなく、<FUNCTION> エラーになります。
G	E 記数法または固定 10 進小数記数法。科学的記数法への変換の結果、小数桁数が decimal 値（または既定の 6 小数桁）より大きくなる場合、数は科学的記数法で返されます。例えば、\$FNUMBER(1234.99, "G", 2) は 1.23E+03 を返します。科学的記数法への変換の結果、小数桁数が decimal 値（または既定の 6 小数桁）と同じか小さくなる場合、数は固定 10 進小数記数法（標準）で返されます。例えば、\$FNUMBER(1234.99, "G", 3) は 1235 を返します。このコードでは、“G” または “g” を指定できます。返される科学的記数法の値には、これに対応して大文字 “E” または小文字 “e” が含まれます。“E” と “G” は互換性がなく、<FUNCTION> エラーになります。
L	先行符号。符号がある場合、inumber の数字部分の前に置く必要があります。括弧は使用しません。このコードは “P”、または “T” 形式コードと共に使用できません。使用を試みると、<SYNTAX> エラーまたは <FUNCTION> エラーが返されます。先頭の符号が既定の形式です。
N	NumericGroupSeparator なし。数値グループ・セパレータを使用できないようにします。この形式コードには、コンマ (,) 形式コードとの互換性がありません。ドット形式コード (“N.”) と組み合わせて使用すると、数値は、ヨーロッパ式の小数点区切り文字を使用してフォーマットされますが、数値グループ・セパレータは使用されません。
O	ODBC ロケール。現在のロケールをオーバーライドし、代わりに PlusSign=+、MinusSign=-、DecimalSeparator=., NumericGroupSeparator=., NumericGroupSize=3 の値で標準 ODBC ロケールを使用します。“O” 形式コードは、単独では、ODBC の MinusSign と DecimalSeparator のみを使用します。この形式コードにはドット (.) 形式コードとの互換性がありません。コンマ形式コード (“O,”) と組み合わせて使用すると、数値は、ODBC の小数点区切り文字および ODBC の数値グループ・セパレータを使用してフォーマットされます。
P	括弧。MinusSign プロパティ値をつけずに、負の数を括弧付きで返します。その他の数は、括弧を付けずにスペース文字を両側に配置して返します。このコードは “+”、“-”、“L”、または “T” 形式コードと共に使用できません。使用を試みると、<SYNTAX> エラーが返されます。



コード	説明
T	後続符号。後に符号が続く数を返すか、あるいは接頭語が生成されます。しかし、必ずしも符号が付くわけではありません。非負数（正数またはゼロ）の後に符号を加えるときは、“+”形式コードも指定します。負数の後に符号を加えるときは、“-”形式コードを指定しないでください。使用される符号はそれぞれ、現在のロケールの PlusSign プロパティと MinusSign プロパティで決まります。末尾スペース文字があるのに符号がないときは、“+”が省略されている非負数、あるいは“-”を指定した負の数の場合です。括弧は使用しません。このコードは“L”、または“P”形式コードと共に使用できません。使用を試みると、<SYNTAX> エラーまたは <FUNCTION> エラーが返されます。

InterSystems IRIS では、1 未満の小数は、ゼロ整数なしの InterSystems IRIS キャノニック形式で表されます。0.66 は .66 になります。これが \$FNUMBER の既定です。ただし、ほとんどの \$FNUMBER format オプションは、先頭にゼロ整数を付けて 1 未満の小数を返します。つまり、.66 は、0.66 になります。format が "" (空の文字列)、“L” (空の文字列と同一機能)、および “D” の 2 引数 \$FNUMBER は、1 未満の小数をキャノニック形式 (.66) で返します。その他すべての 2 引数 \$FNUMBER format オプション、およびすべての 3 引数 \$FNUMBER format オプションでは、先頭にゼロ整数を 1 つ付けて 1 未満の小数を返します。つまり、000.66 と .66 は両方とも 0.66 になります。これは JSON 数値の小数の形式です。

\$DOUBLE 関数は、値 INF (無限大) および NAN (非数値) を返します。INF は負符号を取ります。形式コードは、INF が数であるかのように表します。例: +INF、INF-、(INF)。NAN は符号を取りません。NAN に影響を与える形式コードは “d” のみで、これにより NAN は小文字で返されます。“E” コードおよび “G” コードは、INF および NAN の値には影響を与えません。

## decimal

decimal 引数は、返り値に含める小数桁の数を指定します。decimal には、正の整数、有効な ObjectScript 変数、または正の整数に評価される式を指定します。decimal が負の数の場合、InterSystems IRIS はこれを 0 として扱います。decimal が小数の場合、InterSystems IRIS はこれを切り捨てて整数部分のみにします。

- decimal が inumber の小数桁数よりも大きい場合は、0 で埋めます。
- decimal が inumber の小数桁数よりも小さい場合、InterSystems IRIS は inumber を適切な小数桁数に丸めます。
- decimal が 0 の場合、inumber は、小数点区切り文字なしの整数として返されます。InterSystems IRIS は inumber を適切な整数に丸めます。

inumber が 1 未満で、decimal が 0 より大きい場合、\$FNUMBER は、format の値に関係なく、常に小数点区切り文字の前の整数の位置に 1 つのゼロを返します。小数のこの表示方法は InterSystems IRIS キャノニック形式と異なります。

decimal 引数を指定すると、数の丸めを行った後に返す小数桁の数を制御できます。例えば、変数 c に数値 6.25198 が含まれているとします。

## ObjectScript

```
SET c="6.25198"
SET x=$FNUMBER(c,"+",3)
SET y=$FNUMBER(c,"+",8)
WRITE !,x,! ,y
```

最初の \$FNUMBER は +6.252 を、2 番目は +6.25198000 を返します。

## 例

以下の例は、異なる形式を指定すると、どのように \$FNUMBER の動作に影響するかを示しています。これらの例は、現在のロケールが既定であると仮定した場合です。

以下の例では、符号コードの正の数への影響を示します。

## ObjectScript

```

SET a=1234
WRITE $FNUMBER(a),! ; returns 1234
WRITE $FNUMBER(a,""),! ; returns 1234
WRITE $FNUMBER(a,"+"),! ; returns +1234
WRITE $FNUMBER(a,"-"),! ; returns 1234
WRITE $FNUMBER(a,"L"),! ; returns 1234
WRITE $FNUMBER(a,"T"),! ; returns 1234 (with a trailing space)
WRITE $FNUMBER(a,"T+"),! ; returns 1234+

```

以下の例では、符号コードの負の数への影響を示します。

## ObjectScript

```

SET b=-1234
WRITE $FNUMBER(b,""),! ; returns -1234
WRITE $FNUMBER(b,"+"),! ; returns -1234
WRITE $FNUMBER(b,"-"),! ; returns 1234
WRITE $FNUMBER(b,"L"),! ; returns -1234
WRITE $FNUMBER(b,"T"),! ; returns 1234-

```

以下の例では、“P”形式コードの正および負の数への影響を示します。この例では、正の数が先頭と末尾の空白と共に返されることを示すため、数の前後にアスタリスクを記述しています。

## ObjectScript

```

WRITE "",$FNUMBER(-123,"P"),"*,! ; returns *(123)*
WRITE "",$FNUMBER(123,"P"),"*,! ; returns * 123 *

```

以下の例は 1,234,567.81 を返します。“,” format は、アメリカ形式で x を返し、数値グループ・セパレータとしてコンマを、小数点区切り文字としてピリオドを挿入します。

## ObjectScript

```

SET x=1234567.81
WRITE $FNUMBER(x,"",")

```

以下の例は 1.234.567,81 を返します。“.” format は、ヨーロッパ形式で x を返し、数値グループ・セパレータとしてピリオドを、小数点区切り文字としてコンマを挿入します。

## ObjectScript

```

SET x=1234567.81
WRITE $FNUMBER(x,".")

```

以下の 3 引数の例は 124,329.00 を返します。\$FNUMBER は、数値グループ・セパレータとしてコンマを挿入し、小数点区切り文字としてピリオドを追加し、x の値に小数桁として 2 つのゼロを付加します。

## ObjectScript

```

SET x=124329
WRITE $FNUMBER(x,"",2)

```

以下の 3 引数の例は 124329.00 を返します。省略した format はプレースホルダのコンマで表されます。decimal は、x の値に小数桁として 2 つのゼロを付加します。

## ObjectScript

```

SET x=124329
WRITE $FNUMBER(x,,2)

```

以下の 3 引数の例は 0.78 を返します。省略した format はプレースホルダのコンマで表されます。decimal は 2 小数桁に丸めます。また、decimal は、整数 0 を付加し、format の既定値をオーバーライドします。

## ObjectScript

```
SET x=.7799
WRITE $FNUMBER(x,2)
```

## 小数点区切り文字

\$FNUMBER は、返される数値の整数部分および小数部分の間の区切り文字として、現在のロケールの DecimalSeparator プロパティの値を使用します (既定は “.”)。“.” 形式コードが指定されると、現在のロケール設定に関係なくこの区切り文字は “,” です。

ユーザのロケールの DecimalSeparator 文字を決定するには、以下のように GetFormatItem() メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

## 数値グループ・セパレータおよびサイズ

format の文字列が “,” を含む場合、\$FNUMBER は、返される数値の整数桁のグループ間区切り文字として、現在のロケールの NumericGroupSeparator プロパティの値を使用します。これらのグループの大きさは、現在のロケールの NumericGroupSize プロパティによって決まります。

英語ロケールでは、既定でコンマ (“,”) が NumericGroupSeparator として、3 が NumericGroupSize として設定されます。多くのヨーロッパ言語のロケールでは、ピリオド (“.”) が NumericGroupSeparator として使用されます。ロシア語 (rusw)、ウクライナ語 (ukrw)、およびチェコ語 (csyw) ロケールでは、空白スペースが NumericGroupSeparator として使用されます。NumericGroupSize は、日本語を含め、すべてのロケールで既定で 3 に設定されます。(日本語のユーザについては、状況に応じて、整数を 3 桁単位で区切りたい場合もあれば、4 桁単位で区切りたい場合もあるかもしれません。)

format の文字列が “.” を含む場合 (かつ “N” を含まない場合)、\$FNUMBER は現在のロケール設定に関係なく、NumericGroupSeparator=”.” および NumericGroupSize=3 を使用して、返り値をフォーマットします。

ユーザのロケールの NumericGroupSeparator 文字と NumericGroupSize 数を決定するには、以下のように GetFormatItem() メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSize")
```

## プラス記号とマイナス記号

\$FNUMBER は、現在のロケールに対する PlusSign プロパティおよび MinusSign プロパティ (既定では “+” と “-”) を使用します。“.” 形式コードが指定されると、これらの記号は、現在のロケールに関係なく、“+” と “-” に設定されます。

ユーザのロケールの PlusSign 文字および MinusSign 文字を決定するには、以下のように GetFormatItem() メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PlusSign"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MinusSign")
```

## \$FNUMBER と \$INUMBER の相違点

大半の形式コードは、\$FNUMBER と \$INUMBER で類似した意味を持ちますが、各コードによって引き起こされる正確な動作は、実行される検証と変換の性質により、関数によって異なります。

特に、\$FNUMBER での “-” と “+” の形式コードは、\$INUMBER の場合と同じ意味になるわけではありません。\$FNUMBER で、“-” と “+” は相互排他的ではありません。“-” は (抑制することによって) MinusSign にのみ影響し、“+” は (挿入す

ることによって) PlusSign にのみ影響します。\$INUMBER では、“-”と“+”は相互排他的です。“-”は符号が許可されないという意味であり、“+”は符号が必ず必要であるという意味です。

## 関連項目

- ・ [\\$DOUBLE](#) 関数
- ・ [\\$JUSTIFY](#) 関数
- ・ [\\$INUMBER](#) 関数
- ・ [\\$ISVALIDNUM](#) 関数
- ・ [\\$NORMALIZE](#) 関数
- ・ [\\$NUMBER](#) 関数
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$GET (ObjectScript)

指定した変数のデータ値を返します。

### 構文

```
$GET(variable,default)
$G(variable,default)
```

### 引数

引数	説明
variable	ローカル変数、グローバル変数、またはプロセス・プライベート・グローバル変数、添え字付きまたは添え字なし。この変数は未定義でもかまいません。variable は、構文 obj.property を持つ多次元オブジェクト・プロパティとして指定できます。
default	オプション - 変数が未定義の場合に返される値。変数の場合、定義が必要です。

### 概要

\$GET は、指定した変数のデータ値を返します。未定義の変数の処理は、default 引数を指定しているか否かによって異なります。

- ・ \$GET(variable) は指定した変数値か、または変数が定義されていない場合は、NULL 文字列を返します。variable 引数は、添え字付き配列要素（ローカルあるいはグローバル）を含む、任意の変数の名前です。
- ・ \$GET(variable,default)は変数が定義されていない場合に、既定値を提供して返します。変数が定義されている場合、\$GET はその値を返します。

### 引数

#### variable

データ値が返される変数。

- ・ variable は、ローカル変数、グローバル変数、またはプロセス・プライベート・グローバル (PPG) 変数のいずれかにすることができます。これは添え字付きでも添え字なしでもかまいません。ObjectScript 特殊変数や構造化システム変数 (SSVN) にすることはできません。

定義済みの変数である必要はありません。\$GETは未定義の変数にNULL文字列を返します。変数の定義は行いません。変数は定義してからNULL文字列(“)を設定できます。グローバル変数の場合は、[拡張グローバル参照](#)を含めることができます。添え字付きグローバル変数の場合は、[ネイキッド・グローバル参照](#)を使用して指定できます。未定義の添え字付きグローバル変数を参照するときでも、variable はネイキッド・インジケータをリセットするので、以下で説明されているように、今後のネイキッド・グローバル参照に影響が及びます。

- ・ variable は[多次元オブジェクト・プロパティ](#)にすることはできますが、非多次元オブジェクト・プロパティにすることはできません。非多次元オブジェクト・プロパティで \$GET を使用しようすると、<OBJECT DISPATCH> エラーが発生します。

例えば、%SQL.StatementMetadata クラスには、多次元プロパティ columnIndex と、非多次元プロパティ columnCount があります。以下の例では、最初の \$GET は値を返し、2 番目の \$GET は <OBJECT DISPATCH> エラーを返します。

## ObjectScript

```
SET x=##class(%SQL.StatementMetadata).%New()
WRITE $GET(x.columnIndex,"columnIndex property is undefined"),!
WRITE $GET(x.columnCount,"columnCount property is undefined")
```

## default

variable が未定義の場合に返されるデータ値。添え字付き、または添え字なしのローカル変数やグローバル変数を含む、任意の式です。グローバル変数の場合は、[拡張グローバル参照](#)を含めることができます。添え字付きグローバル変数の場合は、[ネイキッド・グローバル参照](#)を使用して指定できます。存在する場合、default はネイキッド・インジケータをリセットするので、以下で説明されているように、今後のネイキッド・グローバル参照に影響が及びます。

default が未定義の変数である場合、variable が定義されていても、既定で \$GET は〈UNDEFINED〉エラーを発行します。%SYSTEM.Process.Undefined() メソッドを設定することで、未定義の変数を参照する際に〈UNDEFINED〉エラーを生成しないように InterSystems IRIS の動作を変更できます。Undefined() メソッドが〈UNDEFINED〉エラーを生成しないように設定されている場合、\$GET は、default が未定義の場合に variable を返します。

ObjectScript 特殊変数を default として指定できます。ただし、[\\$ZORDER](#) を指定すると、variable が定義されていても、〈UNDEFINED〉エラーになる場合があります。

## 例

以下の例では、変数 test は定義済みで、変数 xtest は未定義です (ZWRITE コマンドは、明示的に NULL 文字列値を返すので使用されます)。

## ObjectScript

```
KILL xtest
SET test="banana"
SET tdef=$GET(test),tundef=$GET(xtest)
ZWRITE tdef      ; $GET returned value of test
ZWRITE tundef    ; $GET returned null string for xtest
WRITE !,$GET(xtest,"none")
      ; $GET returns default of "none" for undefined variable
```

## \$DATA と比較した \$GET

\$GET は、\$DATA に未定義の変数 (\$DATA = 0) とデータ (\$DATA = 10) なしの下位ポインタである配列ノードの代わりを提供しています。variable が未定義であるかデータのないポインタ配列ノードのときは、\$GET は未定義のエラーなしで NULL 文字列 ("") を返します。例えば、以下の行を再コード化します。

## ObjectScript

```
IF $DATA(^client(i))=10 {
    WRITE !,"Name: No Data"
    GOTO Levell+3
}
```

以下のように実行します。

## ObjectScript

```
IF $GET(^client(i))="" {
    WRITE !,"Name: No Data"
    GOTO Levell+3
}
```

\$DATA テストは、未定義要素と下位ポインタのみの要素を識別するので、\$GET テストと比べてより多くの情報を提供します。以下の行で例を示します。

## ObjectScript

```
IF $DATA(^client(i))=0 { QUIT }
ELSEIF $DATA(^client(i))=10 {
    WRITE !!, "Name: No Data"
    GOTO Level1+3
}
```

以下のようには再コード化されません。

## ObjectScript

```
IF $GET(^client(i))="" { QUIT }
ELSEIF $GET(^client(i))="" {
    WRITE !!, "Name: No Data"
    GOTO Level1+3
}
```

2 行は、配列要素が未定義あるいはデータなしの下位ポインタのどちらであるかによって異なる動作をします。`$GET` がここで使用されると、最初の動作 (QUIT) のみが実行されます。`$DATA` を最初のテストで使用し、`$GET` を 2 番目に使用することができますが、その逆 (最初のテストに `$GET`、2 番目に `$DATA` を使用すること) はできません。

## \$GET と \$SELECT の既定

`$GET(variable,default)` は、指定された変数が未定義のときに既定値を返します。`$SELECT` 関数を使用することでも、同じ結果を得ることができます。

しかし `$SELECT` とは異なり、`$GET` の 2 番目の引数は常に評価されます。

`variable` と `default` が両方とも添え字付きグローバル参照を行ってネイキッド・インジケータを変更する場合、`$GET` が常にその引数の両方を評価するという事実が重要になります。引数は左から右の順に評価されるので、`$GET` が `default` 値を返すか否かにかかわらず、ネイキッド・インジケータは `default` グローバル参照に設定されます。グローバル変数とネイキッド・インジケータでの `$GET` の使用に関する詳細は、“[多次元ストレージの使用法 \(グローバル\)](#)”を参照してください。

## 未定義の変数の処理

`$GET` は、指定された変数が未定義の場合、処理の振る舞いを定義します。指定の変数が未定義の場合、`$GET` の返す基本的な形式は `NULL` 文字列 ("" ) です。

`$DATA` は、指定された変数が定義済みかどうかテストします。変数が未定義の場合は 0 を返します。

`%SYSTEM.Process` クラスの `Undefined()` メソッドを使用して、すべての未定義の変数に対する処理の動作を、プロセスごとに定義できます。システム全体の既定の動作は、`Config.Miscellaneous` クラスの `Undefined` プロパティで設定できます。`Undefined` の設定は、指定された変数の `$GET` または `$DATA` の処理に影響しません。

## 関連項目

- [\\$DATA 関数](#)
- [\\$SELECT 関数](#)
- [多次元ストレージの使用法 \(グローバル\)](#)



# \$INCREMENT (ObjectScript)

変数の数値に、指定されたインクリメントを加えます。

## 構文

```
$INCREMENT(variable,num)
$I(variable,num)
```

## 引数

引数	説明
variable	値がインクリメントされる変数。添え字付きあるいは添え字なしのローカル変数、プロセス・プライベート・グローバル、またはグローバル変数を指定できます。変数を定義する必要はありません。変数が定義されない、または NULL 文字列(“”)に設定されている場合、\$INCREMENT は初期値をゼロとして扱い、状況に応じてインクリメントします。ここではリテラル値を指定できません。単純なオブジェクト・プロパティ参照を variable として指定することはできません。構文 obj.property を使用すると、多次元プロパティ参照を variable として指定できます。
num	オプション - variable に加算したいインクリメント数値。値は、数(整数または整数以外の数、正の数または負の数)、数を含む文字列、または数に評価される式のいずれかにできます。先頭、および末尾の空白や複数の記号が評価されます。文字列は、最初の非数値文字に遭遇するまで評価されます。NULL 文字列(“”)は 0 として評価されます。  2 番目の引数に num を指定しない場合、InterSystems IRIS は既定で variable を 1 つインクリメントします。

## 概要

\$INCREMENT と \$SEQUENCE はどちらも、ローカル変数、グローバル変数、またはプロセス・プライベート・グローバルをインクリメントします。\$SEQUENCE は一般的に、グローバルで使用されます。

\$INCREMENT は、変数の既存の値に指定されたインクリメントを追加し、インクリメントされた値を返すことによって、変数の値をリセットします。詳細は、以下の例を参照してください。

### ObjectScript

```
SET a=7
SET result=$INCREMENT(a)
WRITE !,result      /* result is 8 (a+1)          */
WRITE !,a           /* variable a is also now 8 */
```

\$INCREMENT はアトミック処理として実行されます(そのため、LOCK コマンドの使用を必要としません)。

複数のプロセスが \$INCREMENT を介して同じグローバルを同時にインクリメントするとき、各プロセスは一意的増加数(num が負のときは、減少数)を受け取ります。状況によっては、タイミングの問題のために数がスキップされることもあります。グローバル変数での \$INCREMENT の使用に関する詳細は、“[多次元ストレージの使用法 \(グローバル\)](#)”を参照してください。

ロールバックされた[トランザクション](#)に \$INCREMENT がある場合、InterSystems IRIS はインクリメントされていない元の値をリストアップしません。

## 引数

### variable

データ値がインクリメントされる変数。これは変数にする必要があります。リテラルにすることはできません。この変数は定義されている必要はありません。`$INCREMENT` は未定義の変数を定義し、その値を `num` (既定では 1) に設定します。

`variable` 引数は、[ローカル変数](#)、[プロセス・プライベート・グローバル](#)、または[グローバル変数](#)のいずれかになります。添え字付きでも添え字なしでもかまいません。グローバル変数の場合は、[拡張グローバル参照](#)を含めることができます。添え字付きグローバル変数の場合は、[ネイキッド・グローバル参照](#)を使用して指定できます。

`variable` 引数は、[多次元プロパティ参照](#)でもかまいません。例えば、`$INCREMENT(..Count)` のように指定します。非多次元オブジェクト・プロパティとすることはできません。非多次元オブジェクト・プロパティをインクリメントしようとすると、`<OBJECT DISPATCH>` エラーが発生します。

`$INCREMENT` は、[特殊変数](#)をインクリメントできません。これは、`SET` を使用して変更できるものでも同じです。特殊変数をインクリメントしようとすると、`<SYNTAX>` エラーが発生します。

### num

インクリメント(またはデクリメント)する分量。`num` 引数は、`variable` の値をインクリメントする正の数、または `variable` の値をデクリメントする負の数のいずれかにできます。整数または小数を指定できます。`num` はゼロでもかまいません(インクリメントなし)。数値文字列は数値として扱われます。空文字列(“)または非数値文字列は、インクリメント・ゼロとして扱われます。インクリメントを指定しない場合、InterSystems IRIS は既定のインクリメント (1) を使用します。

## `$INCREMENT` または `$SEQUENCE`

`$SEQUENCE` と `$INCREMENT` は、代替として使用したり、互いに組み合わせて使用したりできます。`$SEQUENCE` は、特に複数の同時プロセスが関わる整数インクリメント処理の用途で使用されます。`$INCREMENT` は、より汎用的なインクリメント/デクリメント関数です。

- ・ `$SEQUENCE` は、グローバル変数をインクリメントします。`$INCREMENT` は、ローカル変数、グローバル変数、またはプロセス・プライベート・グローバルをインクリメントします。
- ・ `$SEQUENCE` は、数値を 1 だけインクリメントします。`$INCREMENT` は、任意の数値を任意の指定値単位でインクリメントまたはデクリメントします。
- ・ `$SEQUENCE` は、プロセスにインクリメントの範囲を割り当てることができます。`$INCREMENT` は、1 つのインクリメントのみを割り当てます。
- ・ `SET $SEQUENCE` は、グローバルの変更または未定義化(削除)の用途で使用できます。`$INCREMENT` は、`SET` コマンドの左側で使用することはできません。

## `$INCREMENT` とグローバル変数

グローバル変数、またはグローバル変数の添え字ノードで `$INCREMENT` を使用することができます。[拡張グローバル参照](#)を使用して、他のネームスペースにマップされたグローバル変数にアクセスすることができます。添え字付きグローバル変数の場合は、[ネイキッド・グローバル参照](#)になります。

InterSystems IRIS は、左から右の順に引数を評価します。`num` (インクリメントする数量) が添え字付きグローバルの場合、InterSystems IRIS はこのグローバル参照を使用してネイキッド・インジケータを設定するので、後続のすべてのネイキッド・グローバル参照に影響が及びます。

## `DO $INCREMENT`

`$INCREMENT` は、`DO` コマンドの引数として実行できます。`DO $INCREMENT` は、`$INCREMENT` を関数として呼び出す場合と以下の 2 つの点で異なります。

- DO は関数からの戻り値を無視します。このため、DO \$INCREMENT(variable,num) は variable をインクリメントしますが、インクリメントした値を返すことはありません。
- DO \$INCREMENT(variable,num) が <MAXINCREMENT> エラーを発行することはありません。\$INCREMENT が **インクリメントに失敗**した場合、DO コマンドはエラーを発行せずに完了し、variable はインクリメントされません。

DO \$INCREMENT に、引数の**後置条件式**を追加できます。例えば、DO \$INCREMENT(myvar):x は、x=0 の場合、myvar をインクリメントしません。後置条件式によって実行は阻止されますが、引数の評価は阻止されません。このため、DO \$INCREMENT(myvar(\$INCREMENT(subvar))):x は、x=0 の場合、myvar をインクリメントしませんが、subvar はインクリメントします。

## 文字列のインクリメント

\$INCREMENT は、通常、数値を含む変数をインクリメントするために使用します。しかし、文字列を含む variable も受け入れます。文字列で \$INCREMENT を使用するときには、以下の規則が適用されます。

- NULL 文字列 ("") は、値がゼロであるものとして扱います。
- NULL 文字列 ("123" または "+0012.30") は、数値を持つものとして扱います。文字列は、先頭と末尾のゼロやプラス記号が削除され、キャノン形式に変換されます。
- 数値/非数値の混合文字列 ("12AB" または "1,000") は、最初の子数値文字まで数値として扱われ、その時点で切り捨てられます (コンマは非数値文字であることに注意してください)。結果として得られる数値部分文字列は、先頭と末尾のゼロやプラス記号が削除され、キャノン形式に変換されます。
- 非数値文字列 ("ABC" または "\$12") は、値がゼロであるものとして扱われます。
- 科学的記数法**変換が実行されます。例えば、strvar が "3E2" の場合、\$INCREMENT は、それを値が 300 であるものとして扱います。大文字の "E" は標準の指数演算子です。小文字の "e" は、%SYSTEM.Process.ScientificNotation() メソッドを使用して構成できる指数演算子です。
- 算術演算が実行されません。例えば、strvar が "3+7" の場合、\$INCREMENT はプラス記号 (非数値文字として扱われる) で文字列を切り捨てて、strvar を 4 にインクリメントします。
- 1 つの \$INCREMENT 文で文字列変数を複数使用することは、回避すべきです。例えば、文字列変数を変数 strvar\_\$INCREMENT(strvar) のインクリメントに連結することは避けてください。予測できない結果が発生します。

## インクリメントの失敗

\$INCREMENT が variable をインクリメントできない場合、<MAXINCREMENT> エラーが発行されます。これは、num インクリメント値が非常に小さい場合、および/または variable 値が非常に大きい場合にのみ発生します。

インクリメント・ゼロ (num=0) では、サイズに関係なく常に元の数字が返されます。<MAXINCREMENT> エラーは発行されません。

<MAXINCREMENT> は引数の数値型が異なる場合に発生し、結果として生じる型変換および丸めではインクリメントは行われません。非常に大きな数値に対して \$INCREMENT を使用する場合、既定のインクリメント 1 (または、num のその他の小さな正や負の値) は小さすぎて意味がありません。同様に、非常に小さい小数の num 値を指定した場合も、値が小さすぎて意味がありません。\$INCREMENT は、元の variable の数値をインクリメントせずに返すのではなく、<MAXINCREMENT> エラーを生成します。

以下の例では、1.2E18 は、1 つずつインクリメントまたはデクリメントできる数字であり、1.2E20 は、大きすぎて 1 つずつインクリメントまたはデクリメントできない数字です。最初の 3 つの \$INCREMENT 関数は、数字 1.2E18 を正常にインクリメントまたはデクリメントします。4 つ目と 5 つ目の \$INCREMENT 関数では、インクリメントはゼロです。そのため、この関数は、元の数字のサイズに関係なく、常に元の数字を変更しないで返します。6 つ目と 7 つ目の \$INCREMENT 関数では、num のインクリメントは十分に大きいので、数字 1.2E20 は正常にインクリメントまたはデクリメントされます。8 つ目の \$INCREMENT 関数は、1.2E20 を 1 つずつインクリメントしようとし、その結果 <MAXINCREMENT> エラーを生成します。

## ObjectScript

```

SET x=1.2E18
WRITE "E18      :",x,!
WRITE "E18+1    :", $INCREMENT(x),!
WRITE "E18+4    :", $INCREMENT(x,4),!
WRITE "E18-6    :", $INCREMENT(x,-6),!
WRITE "E18+0    :", $INCREMENT(x,0),!
SET y=1.2E20
WRITE "E20      :",y,!
WRITE "E20+0    :", $INCREMENT(y,0),!
WRITE "E20-10000:", $INCREMENT(y,-10000),!
WRITE "E20+10000:", $INCREMENT(y,10000),!
WRITE "E20+1    :", $INCREMENT(y),!

```

<MAXINCREMENT> は、\$INCREMENT が関数として呼び出された場合にのみ発行されます。DO \$INCREMENT は、<MAXINCREMENT> エラーを発行しません。

## \$INCREMENT とトランザクション処理

\$INCREMENT は通常、新しいエントリをデータベースに追加する前に、カウンタをインクリメントするために使用します。\$INCREMENT を使用すると、LOCK コマンドの使用をしなくても、この操作を迅速に行うことができます。

このカウンタは、トランザクション中の 1 プロセスによってインクリメントされます。トランザクションの処理中に、並列トランザクション中の別のプロセスによってでもインクリメントできます。

いずれかのトランザクション（あるいは \$INCREMENT を使用する他のトランザクション）を（TROLLBACK コマンドで）ロールバックする必要があるとき、カウンタのインクリメントは無視されます。結果のカウンタ値が有効であるかどうかが明確ではないため、カウンタ変数はデクリメントされません。このようなロールバックを行った場合、他のトランザクションに多大な損害が及ぶ可能性が大きくなります。

分散データベース環境での \$INCREMENT の使用に関する詳細は、“[\\$Increment 関数とアプリケーション・カウンタ](#)”を参照してください。

## 例

以下の例は、myvar の値が n 分インクリメントされます。myvar は、事前に定義された変数である必要はないことに注意してください。

## ObjectScript

```

SET n=4
KILL myvar
SET VAL=$INCREMENT(myvar,n)      ; returns 4
WRITE !,myvar
SET VAL=$INCREMENT(myvar,n)      ; returns 8
WRITE !,myvar
SET VAL=$INCREMENT(myvar,n)      ; returns 12
WRITE !,myvar

```

以下の例は、\$INCREMENT を使用して、インクリメント値をプロセス・プライベート・グローバル ^|xyz に追加します。\$INCREMENT の 1 つの引数の形式は、1 をインクリメントします。\$INCREMENT の 2 つの引数の形式は、2 番目の引数で指定された値をインクリメントします。この場合、2 番目の引数は整数以外の値です。

## ObjectScript

```

KILL ^|xyz
WRITE !,$INCREMENT(^|xyz)        ; returns 1
WRITE !,$INCREMENT(^|xyz)        ; returns 2
WRITE !,$INCREMENT(^|xyz)        ; returns 3
WRITE !,$INCREMENT(^|xyz,3.14)   ; returns 6.14

```

以下の例は、0 をインクリメントする結果と、負の数をインクリメントする結果を示しています。

## ObjectScript

```
KILL xyz
WRITE !,$INCREMENT(xyz,0) ; initialized as zero
WRITE !,$INCREMENT(xyz,0) ; still zero
WRITE !,$INCREMENT(xyz) ; increments by 1 (default)
WRITE !,$INCREMENT(xyz) ; increments by 1 (=2)
WRITE !,$INCREMENT(xyz,-1) ; decrements by -1 (=1)
WRITE !,$INCREMENT(xyz,-1) ; decrements by -1 (=0)
WRITE !,$INCREMENT(xyz,-1) ; decrements by -1 (=-1)
```

以下の例は、混合（数値と非数値）num 文字列と NULL 文字列を使用してインクリメントする結果を示しています。

## ObjectScript

```
KILL xyz
WRITE !,$INCREMENT(xyz,"")
; null string initializes to 0
WRITE !,$INCREMENT(xyz,2)
; increments by 2
WRITE !,$INCREMENT(xyz,"")
; null string increments by 0 (xyz=2)
WRITE !,$INCREMENT(xyz,"3A4")
; increments by 3 (rest of string ignored)
WRITE !,$INCREMENT(xyz,"A4")
; nonnumeric string evaluates as zero (xyz=5)
WRITE !,$INCREMENT(xyz,"1E2")
; increments by 100 (scientific notation)
```

## 関連項目

- ・ [\\$SEQUENCE 関数](#)
- ・ [TROLLBACK コマンド](#)
- ・ [トランザクション処理での ObjectScript の使用法](#)

## \$INUMBER (ObjectScript)

数値を検証し、内部形式に変換します。

### 構文

```
$INUMBER(fnumber,format,erropt)  
$IN(fnumber,format,erropt)
```

### 引数

引数	説明
<i>fnumber</i>	内部形式に変換される数値。数値、文字列値、変数名、あるいは有効な ObjectScript 式にできます。
<i>format</i>	有効な表現である外部数値形式を示す形式指定。任意の順序の 0 以上の形式コードで構成される引用符付きの文字列として指定されます。 <a href="#">形式コード</a> の詳細は後述します。一部の形式コードは互換性がなく、エラーが生じます。 <i>erropt</i> 引数の有無によらず、既定の形式には、空の文字列 () を指定できます。
<i>erropt</i>	オプション — <i>fnumber</i> が <i>format</i> に基づいて無効であると見なされた場合に返される式。

### 概要

\$INUMBER 関数は、*format* で指定した形式を使用して、数値 *fnumber* を検証します。そして、内部 InterSystems IRIS 形式に変換します。

*fnumber* が指定 *format* に対応していない場合で、*erropt* を指定していない場合、システムは <ILLEGAL VALUE> エラーを生成します。*erropt* を指定した場合、無効な数値は *erropt* 文字列を返します。

### 引数

#### *format*

利用可能な形式コードは、以下のとおりです。形式の制限に基づいて \$INUMBER を指示するために単一、あるいは組み合わせて指定できます。形式コードが入力されない場合、\$INUMBER は、*fnumber* を可能な限り柔軟に検証します (詳細は [“最大の柔軟性を提供する NULL 形式”](#) を参照)。



コード	説明
+	必須符号。fnumber 値には明示的な符号が必要です。0 にも符号を付ける必要があります (+0、または -0)。“L” もしくは “T” 形式コードによる規制がない限り、符号は先行でも後続でもかまいません。括弧を使用することはできません。符号を必要としない値は NAN のみです。これは、コード “D+” を使用している場合、符号を付けても付けなくても指定できます。
-	符号なし。fnumber に符号がない場合もあります。
D	\$DOUBLE の数値。このコードは fnumber を IEEE 浮動小数点数に変換します。これは \$DOUBLE (fnumber) と同等です。“D” が指定されている場合、引用符付きの文字列 “INF” および “NAN” を fnumber 値として入力できます。INF と NAN は、大文字と小文字の任意の組み合わせで、先頭または末尾の符号や括弧があってもなくても指定できます (NAN の場合、符号は許可されますが無視されます)。変異形の INFINITY および SNAN もサポートされています。
E または G	E 記数法 (科学的記数法)。このコードでは、科学的記数法の形式の文字列として fnumber を指定できます。このコードは、科学的記数法で fnumber を指定することを許可しています (ただし必須ではありません)。
N	NumericGroupSeparator なし。数値グループ・セパレータを使用できないようにします。この形式コードには、コンマ (,) 形式コードとの互換性はありません。
O	ODBC ロケール。現在のロケールにオーバーライドし、代わりに PlusSign=+、MinusSign=-、DecimalSeparator=., NumericGroupSeparator=., NumericGroupSize=3 の値で標準 ODBC ロケールを使用します。この形式コードにはドット (.) 形式コードとの互換性はありません。
P	負の数は括弧で囲む必要があります。非負数は符号を付けません。先頭または末尾の空白を付けても省略してもかまいません。
L	先行符号。符号がある場合、fnumber の数字部分の前に置く必要があります。括弧は使用しません。
T	後続符号。符号がある場合、fnumber の数字部分の後に置く必要があります。括弧は使用しません。
,	fnumber で、現在のロケールのプロパティによって指定された形式が使用されることを予期します。NumericGroupSeparator (既定は “,”) は、fnumber に表示される場合と表示されない場合がありますが、表示される場合は、小数点の左側に NumericGroupSize (既定は 3) 桁ごとに一貫して表示されている必要があります。
.	現在のロケール設定に関係なく、標準的なヨーロッパ形式を要求します。DecimalSeparator はコンマ (,)、NumericGroupSeparator はピリオド (.)、NumericGroupSize は 3、PlusSign はプラス記号 (+)、MinusSign はマイナス記号 (-) を要求します。ピリオドはオプションですが、ピリオドが存在する場合は常に、小数点の左側に 3 桁ごとに表示されなければなりません。

## “+”、“-” および “P” 形式コードが存在しないとき

format に “+”、“-”、または “P” コードのいずれも含まれていないときには、fnumber に以下のいずれかを含めることができます。

- ・ 符号や括弧はなし。
- ・ PlusSign 個別指定プロパティ (既定は “+”) あるいは MinusSign 個別指定プロパティ (既定は “-”) の両方ではなくどちらか一方。この符号の位置は、指定すれば、“L” あるいは “T” 形式コードのどちらかにより決定。
- ・ 括弧で囲む。

## “L”、“T” および “P” 形式コードが存在しないとき

format が “L”、“T” または “P” 形式コードのいずれも含まないときは、fnumber に付いている符号は先行でも後続でもかまいません (ただし、両方には付きません)。



## 形式コード “,” および “.” が存在しないとき

format に “,” 形式コードまたは “.” 形式コードのいずれも含まれないとき、オプションとして fnumber は、(存在する場合は) DecimalSeparator の左右いずれかの任意の場所に NumericGroupSeparator 記号を付けることができます。しかし、各 NumericGroupSeparator は、その左右両隣に少なくとも 1 桁を持たなければなりません。format が “N” を含む場合、NumericGroupSeparator 記号は使用できません。

## 相互排他的形式コード

形式コードには、お互いに競合するものもあります。形式コードの以下の各組み合わせは、相互排他的であり、エラーが発生します。

- ・ “+” は <FUNCTION> エラーになります。
- ・ “-P” または “+P” は <SYNTAX> エラーになります。
- ・ “TP” または “LP” は <SYNTAX> エラーになります。
- ・ “TL” は <FUNCTION> エラーになります。
- ・ “,” は <FUNCTION> エラーになります。
- ・ “,N” は <FUNCTION> エラーになります。
- ・ “.O” は <FUNCTION> エラーになります。

さらに、無効な形式コード文字を指定した場合、<FUNCTION> エラーが発生します。

## 最大の柔軟性を提供する NULL 形式

format を NULL 文字列として指定できます。これは、NULL 形式と呼ばれています。NULL 形式が指定されると、\$INUMBERは以下の記号規則のうちのいずれかを使用して、fnumber 値を受け入れます。

- ・ 符号や括弧はなし
- ・ 前か後の両方ではなく、どちらか一方に MinusSign
- ・ 前か後の両方ではなく、どちらか一方に PlusSign
- ・ 括弧で囲む。

NULL 形式を指定したとき、fnumber は、オプションとして、(存在する場合は) DecimalSeparator の左右いずれかの任意の場所に NumericGroupSeparator 記号を持つことがあります。しかし、各 NumericGroupSeparator は、その左右両隣に少なくとも 1 桁を持たなければなりません。符号に関する規約は柔軟ですが、前と後の空白とゼロは無視されます。したがって、以下の 2 つのコマンド

### ObjectScript

```
WRITE !,$INUMBER("+1,23,456,7.8,9,100","")
WRITE !,$INUMBER("0012,3456,7.891+","")
```

は両方とも有効で、既定のロケールに対応してフォーマットされた同じ数を返します。しかし、以下の例はどうでしょうか。

### ObjectScript

```
WRITE $INUMBER("1,23,,345,7.,8,9","")
```

これは、隣接したコンマ、隣接したピリオドとコンマ、末尾のコンマなので無効です。これによって <ILLEGAL VALUE> エラーが発生します。

## すべての形式に共通する動作

指定した形式コードに関係なく、\$INUMBER は常に前と後にある空白スペースやゼロを無視しますが、以下の特性のいずれかがある場合 fnumber は無効であると見なします。

- ・ PlusSign と MinusSign 両方
- ・ 1 つ以上の PlusSign あるいは MinusSign
- ・ 括弧と PlusSign
- ・ 括弧と MinusSign
- ・ 1 つ以上の DecimalSeparator
- ・ 埋め込まれたスペース
- ・ 下記以外の文字
  - 数字
  - “(“
  - “)”
  - 先行または後続スペース
  - 現在のロケールで指定された DecimalSeparator (format が “.” を含まない場合)
  - 現在のロケールで指定された NumericGroupSeparator (format が “.” を含まない場合)
  - 現在のロケールで指定された PlusSign プロパティ (format が “.” を含まない場合)
  - 現在のロケールで指定された MinusSign プロパティ (format が “.” を含まない場合)
  - “.” (format が “.” を含む場合)
  - “,” (format が “.” を含む場合)
  - “+” (format が “.” を含む場合)
  - “-” (format が “.” を含む場合)
- ・ 文字列 “INF” と “NAN” およびそれらの変異形 (format が “D” を含む場合)

## 例

これらの例は、異なる形式がどのように \$INUMBER の動作に影響するかを示しています。すべて、現在のロケールが既定のロケールであると仮定します。

以下の例では、“L” 形式コードのため \$INUMBER は先行マイナス符号を受け入れ、-123456789.12345678 を返します。

### ObjectScript

```
WRITE $INUMBER("-123,4,56,789.1234,5678","L")
```

以下の例では、符号が先行していながら、“T” 形式コードで後続符号を使用するように指定しているので、\$INUMBER は <ILLEGAL VALUE> エラーを生じます。

### ObjectScript

```
WRITE $INUMBER("-123,4,56,789.1234,5678","T")
```

以下の例では、最初の \$INUMBER は成功し、負の数を返します。2 番目の \$INUMBER では、fnumber は符号を含んでいますが、“P” 形式コードでは、負の数に符号を付けるのではなく括弧で囲むように指定されているので、<ILLEGAL VALUE> エラーが発生します。

#### ObjectScript

```
WRITE !,$INUMBER("(123,4,56,789.1234,5678)","P")
WRITE !,$INUMBER("-123,4,56,789.1234,5678","P")
```

以下の例では、符号がありながら、“-” 形式コードで符号で使わないように指定しているので、\$INUMBER は <ILLEGAL VALUE> エラーを生じます。

#### ObjectScript

```
WRITE $INUMBER("-123,4,56,789.1234,5678","-")
```

以下の例では、\$INUMBER は失敗しますが、符号の不正使用を原因とするエラーは発生せず、erroppt として指定されている文字列 “ERR” をその値として返します。

#### ObjectScript

```
WRITE $INUMBER("-123,4,56,789.1234,5678","-", "ERR")
```

以下の例は、-23456789.123456789 を返します。先行符号の使用は、“L” で指定される形式に従っており、小数点の左側に 3 桁ごとに必ずコンマがあり、右側にはコンマがないことは、“,” コードで指定される厳密な形式に従っているため、\$INUMBER は指定した fnumber を有効な値として受け入れます。

#### ObjectScript

```
WRITE $INUMBER("-23,456,789.123456789","L,")
```

以下の例で、“E” コードは科学的記数法の文字列の数値への変換を許可しています。すべての形式コードは数値リテラルとして科学的記数法をサポートしますが、文字列として科学的記数法をサポートするのは“E”（または“G”）のみです。この例では、変数および連結を使用して科学的記数法の文字列値を提供します。

#### ObjectScript

```
SET num=1.234
SET exp=-14
WRITE $INUMBER(1.234E-14,"E","E-lit-err"),!
WRITE $INUMBER(num_"E"_exp,"E","E-string-err"),!
WRITE $INUMBER(1.234E-14,"L","L-lit-err"),!
WRITE $INUMBER(num_"E"_exp,"L","L-string-err"),!
```

以下の例では、小数および定数 pi の“L” コードおよび“D” コードにより返される値を比較します。“D” コードは IEEE 浮動小数点 (\$DOUBLE) 数に変換します。

#### ObjectScript

```
WRITE $INUMBER(1.23E-23,"L"),!
WRITE $INUMBER(1.23E-23,"D"),!
WRITE $INUMBER($ZPI,"L"),!
WRITE $INUMBER($ZPI,"D"),!
```

## \$INUMBER と \$FNUMBER の相違点

大半の形式コードは、\$INUMBER と \$FNUMBER で類似した意味を持ちますが、各コードによって引き起こされる正確な動作は、実行される検証と変換の性質により、関数によって異なります。

特に、\$INUMBER での“-”と“+”の形式コードは、\$FNUMBER の場合と同じ意味になるわけではありません。\$FNUMBER で、“-”と“+”は相互排他的ではありません。“-”は（抑制することによって）MinusSign にのみ影響し、“+”は（挿入す

ることによって) PlusSign にのみ影響します。\$NUMBER では、“-”と“+”は相互排他的です。“-”は符号が許可されないという意味であり、“+”は符号が必ず必要であるという意味です。

## 小数点区切り文字

\$NUMBER は、fnumber の整数部分および小数部分の間の区切り文字として、現在のロケールの DecimalSeparator プロパティの値を使用します (既定は “.”)。“.” 形式コードが指定されると、現在のロケールに関係なくこの区切り文字は “,” です。

ユーザのロケールの DecimalSeparator 文字を決定するには、以下のように GetFormatItem() メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

## 数値グループ・セパレータおよびサイズ

\$NUMBER は、fnumber の整数部分の桁のグループ間の区切り文字として、現在のロケールの NumericGroupSeparator プロパティの値を使用します (既定は “,”)。これらのグループの大きさは、現在のロケールの NumericGroupSize プロパティによって決まります (既定は “3”)。“.” 形式コードが指定されると、この区切り文字は “.” であり、現在のロケールに関係なく 3 桁ごとに表示されます。

ユーザのロケールの NumericGroupSeparator 文字と NumericGroupSize 数を決定するには、以下のように GetFormatItem() メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSize")
```

## プラス記号とマイナス記号

\$NUMBER は、現在のロケールの PlusSign プロパティ値および MinusSign プロパティ値 (既定では “+”と“-”)を使用します。“.” 形式コードが指定されると、これらの記号は、現在のロケールに関係なく、“+”と“-”に設定されます。

ユーザのロケールの PlusSign 文字および MinusSign 文字を決定するには、以下のように GetFormatItem() メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PlusSign"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MinusSign")
```

## 関連項目

- [\\$DOUBLE](#) 関数
- [\\$FNUMBER](#) 関数
- [\\$ISVALIDNUM](#) 関数
- [\\$NORMALIZE](#) 関数
- [\\$NUMBER](#) 関数
- [各国言語サポートのシステム・クラスの使用法](#)

## \$ISOBJECT (ObjectScript)

expression がオブジェクト参照 (OREF) であるか否かを返します。

### 構文

`$ISOBJECT(expr)`

### 引数

引数	説明
expr	ObjectScript 式。

### 概要

expr がオブジェクト参照 (OREF) の場合、\$ISOBJECT は 1 を返します。expr がオブジェクト参照 (OREF) でない場合、\$ISOBJECT は 0 を返します。

expr が無効なオブジェクトへの参照の場合、\$ISOBJECT は -1 を返します。通常の処理においては無効なオブジェクトは生じません。例えば、クラス内のインスタンスがアクティブなときにそのクラスをリコンパイルすると、無効なオブジェクトが生じます。

オブジェクト参照を削除するには、変数に NULL 文字列(″)を設定します。従来の %Close() メソッドは、オブジェクト参照の削除には使用できません。%Close() は処理をいっさい実行せずに、処理が成功したことを示すメッセージを返します。新規のコードを記述するときは、%Close() を使用しないでください。

OREFの詳細は、"[OREF の基本](#)"を参照してください。

### 引数

#### expr

任意の ObjectScript 式。

### 例

以下の例では、\$ISOBJECT によって、オブジェクト参照と非オブジェクト参照(この場合は、文字列参照)として返される値を示します。

#### ObjectScript

```
SET a="certainly not an object"
SET o=##class(%SQL.Statement).%New()
WRITE !,"non-object a: ", $ISOBJECT(a)
WRITE !,"object ref o: ", $ISOBJECT(o)
```

以下の例では、JSON 値がオブジェクト参照であることが示されています。

#### ObjectScript

```
SET a=["apple","banana","orange"]
SET b={"fruit":"orange","color":"orange"}
WRITE !,"JSON array: ", $ISOBJECT(a)
WRITE !,"JSON object: ", $ISOBJECT(b)
```

以下のダイナミック SQL の例では、[ストリーム・フィールド](#)がオブジェクト参照ではなく、OID であることが示されています。オブジェクト参照を返すには、SQL [%OBJECT](#) 関数を使用する必要があります。

## ObjectScript

```
set myquery=2
set myquery(1)="SELECT TOP 1 Name,Notes,%OBJECT(Notes) AS NoteObj "
set myquery(2)="FROM Sample.Employee WHERE %OBJECT(Notes) IS NOT NULL"
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 "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}

while rset.%Next()
{
    write "Stream field oid: ",$ISOBJECT(rset.Notes),!
    write "Stream field oref: ",$ISOBJECT(rset.NoteObj),!
}
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
```

以下の例では、オブジェクト参照を削除する方法を示します。%Close() メソッドは、オブジェクト参照を変更しません。オブジェクト参照に NULL 文字列を設定することで、オブジェクト参照は削除されます。

## ObjectScript

```
SET o=##class(%SQL.Statement).%New()
WRITE !,"objref o: ",$ISOBJECT(o)
DO o.%Close() ; this is a no-op
WRITE !,"objref o: ",$ISOBJECT(o)
SET o=""
WRITE !,"objref o: ",$ISOBJECT(o)
```

## 関連項目

- ・ [\\$SYSTEM](#) 特殊変数

## \$ISVALIDDOUBLE (ObjectScript)

\$DOUBLE 数値を検証し、ブーリアン値を返します。オプションで、範囲のチェックも行います。

### 構文

```
$ISVALIDDOUBLE(num,scale,min,max)
```

### 引数

引数	説明
num	検証される数値。数値、文字列値、変数名、あるいは有効な ObjectScript 式にできます。有効な数字の場合、num は IEEE 倍精度浮動小数点型に変換されます。
scale	オプション – min と max の範囲を比較する場合に有効な小数点以下の桁数。
min	オプション – 許容範囲内の最小値。指定した値は、IEEE 倍精度浮動小数点型に変換されます。指定されない場合、min は既定で \$DOUBLE(“-INF”) に設定されます。
max	オプション – 許容範囲内の最大値。指定した値は、IEEE 倍精度浮動小数点型に変換されます。指定されない場合、max は既定で \$DOUBLE(“INF”) に設定されます。

### 説明

\$ISVALIDDOUBLE 関数は、num が IEEE 倍精度浮動小数点数の検証テストに合格したかどうかを判別し、ブーリアン値を返します。オプションで、自動的に IEEE の数に変換される min 値と max 値を使用して範囲チェックも行います。scale 引数は、範囲チェック時に比較する小数桁数を指定するために使用されます。num は適切に構成された IEEE 倍精度数値であることをブーリアン値の 1 は意味し、指定された場合は範囲チェックに合格したことを意味します。\$ISVALIDDOUBLE は検証関数であり、num がデータ型 DOUBLE の数であるかどうかを判別したり、\$DOUBLE 関数を使用して生成されたものであるかどうかを判別したりはしません。

\$ISVALIDDOUBLE は、アメリカ形式の数 (ピリオド (.) を小数点区切り文字として使用する) を評価します。これは、ヨーロッパ形式の数 (コンマ (,) を小数点区切り文字として使用する) は評価しません。\$ISVALIDDOUBLE は、数値グループ区切り文字を含む数を有効と見なしません。現在のロケールに関わらず、コンマまたは空白スペースを含む数に対して 0 (無効) を返します。

### 引数

#### num

検証される数は整数、小数、または科学的記数法の数 (文字 “E” または “e” 付き) のいずれかです。文字列、式、または数に解決される変数のいずれかになることもあります。符号付きや符号なし、先頭、または末尾に 0 が付いていることもあります。

ObjectScript は、検証のために \$ISVALIDDOUBLE に数値を渡す前に、その数値を**キャノニック形式**に変換します。したがって、数式、数値の連結、先頭にある複数の + 記号および - 記号は、すべて関数によって評価前に変換されます。数値文字列は、評価前にキャノニック形式に変換されません。ただし、数値文字列の先頭に + 記号が付いている場合、**数値文字列**の数値評価 (およびキャノニック形式への変換) が強制実行されます。

検証は、以下の場合に失敗 (\$ISVALIDDOUBLE が 0 を返す) します。

- num が 0-9、先頭の +、- 符号、小数点(.), および “E” または “e” 以外の任意の文字を含む場合。科学的記数法では、大文字の “E” は標準の指数演算子です。小文字の “e” は、%SYSTEM.Process.ScientificNotation() メソッドを使用して構成できる指数演算子です。



- ・ num が複数の + または - 符号、小数点、文字 "E" または "e" を含む場合。num が数値の場合、ObjectScript は、先頭にある複数の + 記号および - 記号を変換します。num が数値文字列の場合、先頭の + 記号や - 記号を複数含めることはできません。
- ・ オプションの + または - 符号は、num の最初の文字ではありません。
- ・ 10 進法の指数を表す文字 "E"、もしくは "e" の後に数値文字列の整数が続かない場合。整数が続かない数 "E" では、<SYNTAX> エラーが返されます。
- ・ num が NULL 文字列の場合。

num が INF である場合 (+ 記号や - 記号の有無は関係ありません)、それは有効な \$DOUBLE 数です。\$ISVALIDDOUBLE により返されるブーリアンは、指定の範囲チェックに num が適合しているかどうかによって決まります。

num が NAN の場合、\$ISVALIDDOUBLE から 1 が返されます。

## scale

scale 引数は、範囲チェック時に比較する小数桁数を指定するために使用されます。scale には整数値を指定します。scale 値の小数桁数は無視されます。他の引数で指定された小数桁数よりも大きい scale 値を指定することができます。-1 の scale 値を指定することもできます。それ以外の負の scale 値はすべて、<FUNCTION> エラーが返されます。

非負数の scale 値を指定すると、min と max の範囲チェック実行前に、num がその小数桁数に丸められます。scale 値に 0 を指定すると、範囲チェック実行前に num は整数値 (3.9 = 4) に丸められます。scale 値に -1 を使用すると、範囲チェック実行前に num は整数値 (3.9 = 3) に切り捨てられます。指定されたすべての桁を丸めたり、切り捨てることなく比較するには、scale 引数を省略します。数値以外の scale 値や NULL 文字列は、scale 値 0 に相当します。

丸めは、-1 を除くすべての scale 値に対して実行されます。5 以上の値は常に切り上げられます。

scale 引数値により、num 値の端数を丸めた値、または切り捨てた値を使用して評価が行われます。num 変数の実際の値は、\$ISVALIDDOUBLE 処理によって変更されません。

scale 引数を省略する場合、コンマはプレースホルダとして保持しておきます。

## min と max

許容最小値や許容最大値、またはその両方を指定できます。あるいは、一切指定なしでもかまいません。指定された場合、(scale 演算後の) num の値は min の値以上で、max の値以下でなければなりません。値は範囲チェックに使用される前に IEEE 浮動小数点数に変換されます。min または max 値として NULL 文字列を指定した場合、これは 0 に相当します。値がこの条件に合わない場合、\$ISVALIDDOUBLE は 0 を返します。

NAN 値は、min または max の値にかかわらず、常に有効と見なされます。

引数を省略する場合、コンマはプレースホルダとして保持しておきます。例えば、scale を省略して min か max を指定するとき、または min を省略して max を指定するときなどです。末尾のコンマは無視されます。

## 例

以下の例では、\$ISVALIDDOUBLE の各呼び出しはすべて 1 (有効な値) を返します。

### ObjectScript

```
WRITE !,$ISVALIDDOUBLE(0)           ; All integers OK
WRITE !,$ISVALIDDOUBLE(4.567)       ; Fractional numbers OK
WRITE !,$ISVALIDDOUBLE("4.567")    ; Numeric strings OK
WRITE !,$ISVALIDDOUBLE(-.0)         ; Signed numbers OK
WRITE !,$ISVALIDDOUBLE(+---123)     ; Multiple signs resolved for numbers OK
WRITE !,$ISVALIDDOUBLE(+004.500)    ; Leading/trailing zeroes OK
WRITE !,$ISVALIDDOUBLE(4E2)         ; Scientific notation OK
```

以下の例では、\$ISVALIDDOUBLE の各呼び出しはすべて 0 (無効な値) を返します。

## ObjectScript

```
WRITE !,$ISVALIDDOUBLE("") ; Null string is invalid
WRITE !,$ISVALIDDOUBLE("4,567") ; Commas are not permitted
WRITE !,$ISVALIDDOUBLE("4A") ; Invalid character
WRITE !,$ISVALIDDOUBLE("--++-123") ; Multiple signs not resolved for strings
```

以下の例では、INF (無限大) および NAN (非数値) が厳密には数字でなくても、\$ISVALIDDOUBLE の各呼び出しはすべて 1 (有効な値) を返します。

## ObjectScript

```
WRITE !,$ISVALIDDOUBLE($DOUBLE($ZPI)) ; DOUBLE numbers OK
WRITE !,$ISVALIDDOUBLE($DOUBLE("INF")) ; DOUBLE INF OK
WRITE !,$ISVALIDDOUBLE($DOUBLE("NAN")) ; DOUBLE NAN OK
```

以下の例では、min 値を指定すると、-INF は除外されますが、INF は除外されません。

## ObjectScript

```
WRITE !,$ISVALIDDOUBLE($DOUBLE("-INF"),,9999999999)
WRITE !,$ISVALIDDOUBLE($DOUBLE("INF"),,9999999999)
```

以下の例は、min 引数や max 引数の使用法を示しています。以下はすべて、1 (数は有効で、範囲チェックにも合格する) を返します。

## ObjectScript

```
WRITE !,$ISVALIDDOUBLE(4,,3,5) ; scale can be omitted
WRITE !,$ISVALIDDOUBLE(4,2,3,5) ; scale can be larger than
; number of fractional digits
WRITE !,$ISVALIDDOUBLE(4,0,,5) ; min or max can be omitted
WRITE !,$ISVALIDDOUBLE(4,0,4,4) ; min and max are inclusive
WRITE !,$ISVALIDDOUBLE(-4,0,-5,5) ; negative numbers
WRITE !,$ISVALIDDOUBLE(4.00,2,04,05) ; leading/trailing zeros
WRITE !,$ISVALIDDOUBLE(.4E3,0,3E2,400) ; base-10 exponents expanded
```

以下の例は、min 引数と max 引数と一緒に scale 引数を使用する方法を示しています。以下はすべて、1 (数は有効で、範囲チェックにも合格する) を返します。

## ObjectScript

```
WRITE !,$ISVALIDDOUBLE(4.55,,4.54,4.551)
; When scale is omitted, all digits of num are checked.
WRITE !,$ISVALIDDOUBLE(4.1,0,4,4.01)
; When scale=0, num is rounded to an integer value
; (0 fractional digits) before min & max check.
WRITE !,$ISVALIDDOUBLE(3.85,1,3.9,5)
; num is rounded to 1 fractional digit,
; (with values of 5 or greater rounded up)
; before min check.
WRITE !,$ISVALIDDOUBLE(4.01,17,3,5)
; scale can be larger than number of fractional digits.
WRITE !,$ISVALIDDOUBLE(3.9,-1,2,3)
; When scale=-1, num is truncated to an integer value
```

## \$ISVALIDDOUBLE と \$ISVALIDNUM の比較

\$ISVALIDDOUBLE 関数と \$ISVALIDNUM 関数は両方ともアメリカ形式の数を検証し、ブーリアン値 (0 または 1) を返します。

- どちらの関数も、有効な数字として、\$DOUBLE によって返された INF、-INF、NAN の値を受け入れます。また、\$ISVALIDDOUBLE は、有効な数字として、大文字小文字を区別しない文字列 “NAN”、“INF”、および “Infinity”、“sNAN”、プラス記号またはマイナス記号が先頭に付いた任意のこれらの文字列を受け入れます。\$ISVALIDNUM は、これらの文字列すべてを無効として拒否し、0 を返します。

## ObjectScript

```

WRITE !,$ISVALIDNUM($DOUBLE("NAN"))      ; returns 1
WRITE !,$ISVALIDDOUBLE($DOUBLE("NAN"))    ; returns 1
WRITE !,$ISVALIDNUM("NAN")                 ; returns 0
WRITE !,$ISVALIDDOUBLE("NAN")              ; returns 1

```

- ・ これら 2 つの関数は、符号付き、または符号なしの整数 (− 0 を含む)、科学的記数法の数 (“E” または “e” 付き)、実数 (123.45)、および数値文字列 (“123.45”) を解析します。
- ・ どちらの関数も、ヨーロッパ形式の DecimalSeparator 文字 (コンマ (,)) および NumericGroupSeparator 文字 (アメリカ形式はコンマ (,), ヨーロッパ形式はピリオド (.)) は認識しません。例えば、現在のロケール設定に関わらず、どちらも文字列 “123,456” を無効な数値として拒否します。
- ・ 2 つの関数は、数字に対して先頭の複数の符号 (+ と −) を解析します。どちらも引用符付き数値文字列に入った先頭の複数の符号を受け入れません。

## 関連項目

- ・ [\\$DOUBLE](#) 関数
- ・ [\\$FNUMBER](#) 関数
- ・ [\\$INUMBER](#) 関数
- ・ [\\$ISVALIDNUM](#) 関数
- ・ [\\$NORMALIZE](#) 関数
- ・ [\\$NUMBER](#) 関数
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ISVALIDNUM (ObjectScript)

数値を検証し、ブーリアン値を返します。オプションで、範囲のチェックも行います。

### 構文

```
$ISVALIDNUM(num, scale, min, max)
```

### 引数

引数	説明
<i>num</i>	検証される数値。数値、文字列値、変数名、あるいは有効な ObjectScript 式にできます。
<i>scale</i>	オプション – <i>min</i> と <i>max</i> の範囲を比較する場合に有効な小数点以下の桁数。
<i>min</i>	オプション – 許容範囲内の最小値。
<i>max</i>	オプション – 許容範囲内の最大値。

### 概要

\$ISVALIDNUM 関数は *num* を検証し、ブーリアン値を返します。オプションで、*min* 値と *max* 値を使用して範囲チェックも行います。*scale* 引数は、範囲チェック時に比較する小数桁数を指定するために使用されます。ブーリアン値 1 は、*num* が適切に構成された数であることを意味し、指定された場合は範囲チェックに合格したことを意味します。

\$ISVALIDNUM は、アメリカ形式の数 (ピリオド (.) を小数点区切り文字として使用する) を評価します。これは、ヨーロッパ形式の数 (コンマ (,) を小数点区切り文字として使用する) は評価しません。\$ISVALIDNUM は、数値グループ区切り文字を含む数を有効と見なしません。現在のロケールに関係なく、コンマまたは空白スペースを含む数に対して 0 (無効) を返します。

### 引数

#### num

検証される数は整数、実数、または科学的記数法の数 (文字 “E” または “e” 付き) のいずれかです。文字列、式、または数に解決される変数のいずれかになることもあります。符号付きや符号なし、先頭、または末尾に 0 が付いていることもあります。検証は、以下の場合に失敗 (\$ISVALIDNUM が 0 を返す) します。

- ・ *num* が空の文字列 (“”) の場合。
- ・ *num* が 0-9、先頭の +、- 符号、小数点(.), および “E” または “e” 以外の任意の文字を含む場合。科学的記数法では、大文字の “E” は標準の指数演算子です。小文字の “e” は、%SYSTEM.Process.ScientificNotation() メソッドを使用して構成できる指数演算子です。
- ・ *num* が複数の + または - 符号、小数点、文字 “E” または “e” を含む場合。
- ・ オプションの + または - 符号は、*num* の最初の文字ではありません。
- ・ 10 進法の指数を表す文字 “E”、もしくは “e” の後に数値文字列の整数が続かない場合。

*scale* 引数値により、*num* 値の端数を丸めた値、または切り捨てた値を使用して評価が行われます。*num* 変数の実際の値は、\$ISVALIDNUM 処理によって変更されません。

*num* が \$DOUBLE、\$ISVALIDNUM によって返された INF、-INF、または NAN の値の場合、1 を返します。

InterSystems IRIS® データ・プラットフォームがサポートする浮動小数点数の最大値は 1.7976931348623157081E308 です。任意の InterSystems IRIS の数値演算でこれよりも大きい数を指定すると、<MAXNUMBER> エラーが発生します。\$ISVALIDNUM がサポートする、文字列として指定される InterSystems IRIS の 10 進数の浮動小数点数の最大値は

9.223372036854775807E145 です。これよりも大きい浮動小数点数文字列の場合は、[\\$ISVALIDDOUBLE](#) を使用します。詳細は、[“極端に大きな数字”](#) を参照してください。

## scale

scale 引数は、範囲チェック時に比較する小数桁数を指定するために使用されます。scale には整数値を指定します。scale 値の小数桁数は無視されます。他の引数で指定された小数桁数よりも大きい scale 値を指定することができます。-1 の scale 値を指定することもできます。それ以外の負の scale 値はすべて、<FUNCTION> エラーが返されます。

非負数の scale 値を指定すると、min と max の範囲チェック実行前に、num がその小数桁数に丸められます。scale 値に 0 を指定すると、範囲チェック実行前に num は整数値 (3.9 = 4) に丸められます。scale 値に -1 を使用すると、範囲チェック実行前に num は整数値 (3.9 = 3) に切り捨てられます。指定されたすべての桁を丸めたり、切り捨てることなく比較するには、scale 引数を省略します。数値以外の scale 値や NULL 文字列は、scale 値 0 に相当します。

丸めは、-1 を除くすべての scale 値に対して実行されます。5 以上の値は常に切り上げられます。

scale 引数を省略する場合、コンマはプレースホルダとして保持しておきます。

数値を丸める際、IEEE 浮動小数点数と標準の InterSystems IRIS 小数の精度が異なることに注意してください。\$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。これらの精度は 53 バイナリ・ビットで、これは 15.95 小数桁数の精度に相当します (バイナリ表現は正確に小数と一致するものではありません)。ほとんどの小数はこのバイナリ表現では正確に表すことができないため、IEEE 浮動小数点数は対応する InterSystems IRIS 標準の浮動小数点数とは若干異なる場合があります。サポートされるすべての InterSystems IRIS システム・プラットフォームにおいて、標準の InterSystems IRIS 小数は、小数桁数 18 桁の精度を持ちます。IEEE 浮動小数点数が小数として表示されるとき、バイナリ・ビットが 18 桁を大幅に超える小数桁数の小数に変換されることがよくあります。これは、IEEE 浮動小数点数が InterSystems IRIS 標準の小数より精度が高いことを意味するわけではありません。

## min と max

許容最小値や許容最大値、またはその両方を指定できます。あるいは、一切指定なしでもかまいません。指定された場合、(scale 演算後の) num の値は min の値以上で、max の値以下でなければなりません。min または max 値として NULL 文字列を指定した場合、これは 0 に相当します。値がこの条件に合わない場合、\$ISVALIDNUM は 0 を返します。

引数を省略する場合、コンマはプレースホルダとして保持しておきます。例えば、scale を省略して min か max を指定するとき、または min を省略して max を指定するときなどです。末尾のコンマは無視されます。

num、min、または max 値が \$DOUBLE 数の場合、この範囲チェックではこれら 3 つの数はすべて \$DOUBLE 数として処理されます。これによって、生成された \$DOUBLE 数の小数部の小部分によって予期しない範囲エラーが引き起こされるのを防ぐことができます。

## 例

以下の例では、\$ISVALIDNUM の各呼び出しはすべて 1 (有効な値) を返します。

### ObjectScript

```
WRITE !,$ISVALIDNUM(0)           ; All integers OK
WRITE !,$ISVALIDNUM(4.567)      ; Real numbers OK
WRITE !,$ISVALIDNUM("4.567")    ; Numeric strings OK
WRITE !,$ISVALIDNUM(-.0)        ; Signed numbers OK
WRITE !,$ISVALIDNUM(+004.500)    ; Leading/trailing zeroes OK
WRITE !,$ISVALIDNUM(4E2)        ; Scientific notation OK
```

以下の例では、\$ISVALIDNUM の各呼び出しはすべて 0 (無効な値) を返します。

### ObjectScript

```
WRITE !,$ISVALIDNUM("")         ; Null string is invalid
WRITE !,$ISVALIDNUM("4,567")    ; Commas are not permitted
WRITE !,$ISVALIDNUM("4A")       ; Invalid character
```

以下の例では、INF (無限大) および NAN (非数値) が厳密には数字でなくても、\$ISVALIDNUM の各呼び出しはすべて 1 (有効な値) を返します。

### ObjectScript

```
DO ##class(%SYSTEM.Process).IEEEEError(0)
WRITE !,$ISVALIDNUM($DOUBLE($ZPI)) ; DOUBLE numbers OK
WRITE !,$ISVALIDNUM($DOUBLE("INF")) ; DOUBLE INF OK
WRITE !,$ISVALIDNUM($DOUBLE("NAN")) ; DOUBLE NAN OK
WRITE !,$ISVALIDNUM($DOUBLE(1)/0) ; generated INF OK
```

以下の例は、min 引数や max 引数の使用法を示しています。以下はすべて、1 (数は有効で、範囲チェックにも合格する) を返します。

### ObjectScript

```
WRITE !,$ISVALIDNUM(4,,3,5) ; scale can be omitted
WRITE !,$ISVALIDNUM(4,2,3,5) ; scale can be larger than
                                ; number of fractional digits
WRITE !,$ISVALIDNUM(4,0,,5) ; min or max can be omitted
WRITE !,$ISVALIDNUM(4,0,4,4) ; min and max are inclusive
WRITE !,$ISVALIDNUM(-4,0,-5,5) ; negative numbers
WRITE !,$ISVALIDNUM(4.00,2,04,05) ; leading/trailing zeros
WRITE !,$ISVALIDNUM(.4E3,0,3E2,400) ; base-10 exponents expanded
```

以下の例は、min 引数と max 引数と一緒に scale 引数を使用する方法を示しています。以下はすべて、1 (数は有効で、範囲チェックにも合格する) を返します。

### ObjectScript

```
WRITE !,$ISVALIDNUM(4.55,,4.54,4.551)
    ; When scale is omitted, all digits of num are checked.
WRITE !,$ISVALIDNUM(4.1,0,4,4.01)
    ; When scale=0, num is rounded to an integer value
    ; (0 fractional digits) before min & max check.
WRITE !,$ISVALIDNUM(3.85,1,3.9,5)
    ; num is rounded to 1 fractional digit,
    ; (with values of 5 or greater rounded up)
    ; before min check.
WRITE !,$ISVALIDNUM(4.01,17,3,5)
    ; scale can be larger than number of fractional digits.
WRITE !,$ISVALIDNUM(3.9,-1,2,3)
    ; When scale=-1, num is truncated to an integer value
```

## \$ISVALIDNUM と \$ISVALIDDOUBLE の比較

\$ISVALIDNUM 関数と \$ISVALIDDOUBLE 関数は両方とも数を検証し、ブーリアン値 (0 または 1) を返します。

- どちらの関数も、有効な数字として、\$DOUBLE によって返された INF、-INF、NAN の値を受け入れます。また、\$ISVALIDDOUBLE は、有効な数字として、大文字小文字を区別しない文字列 “NAN”、“INF”、および “Infinity”、“sNAN”、プラス記号またはマイナス記号が先頭に付いた任意のこれらの文字列を受け入れます。\$ISVALIDNUM は、これらの文字列すべてを無効として拒否し、0 を返します。

### ObjectScript

```
WRITE !,$ISVALIDNUM($DOUBLE("NAN")) ; returns 1
WRITE !,$ISVALIDDOUBLE($DOUBLE("NAN")) ; returns 1
WRITE !,$ISVALIDNUM("NAN") ; returns 0
WRITE !,$ISVALIDDOUBLE("NAN") ; returns 1
```

- これら 2 つの関数は、符号付き、または符号なしの整数 (-0 を含む)、科学的記数法の数 (“E” または “e” 付き)、実数 (123.45)、および数値文字列 (“123.45”) を解析します。
- どちらの関数も、ヨーロッパ形式の DecimalSeparator 文字 (コンマ (,)) および NumericGroupSeparator 文字 (アメリカ形式はコンマ (.), ヨーロッパ形式はピリオド (.)) は認識しません。例えば、現在のロケール設定に関わらず、どちらも文字列 “123,456” を無効な数値として拒否します。



- 2 つの関数は、数字に対して先頭の複数の符号 (+ と -) を解析します。どちらも引用符付き数値文字列に入った先頭の複数の符号を受け入れません。

数値文字列が大きすぎて InterSystems IRIS 浮動小数点数によって表せない場合、既定では IEEE 倍精度数に自動的に変換されます。ただし、以下の例のように、そのような大きな数字は \$ISVALIDNUM テストに失敗します。

## ObjectScript

```
WRITE !,"E127 no IEEE conversion required"
WRITE !,$ISVALIDNUM("9223372036854775807E127")
WRITE !,$ISVALIDDOUBLE("9223372036854775807E127")
WRITE !,"E128 automatic IEEE conversion"
WRITE !,$ISVALIDNUM("9223372036854775807E128")
WRITE !,$ISVALIDDOUBLE("9223372036854775807E128")
```

## \$ISVALIDNUM、\$NORMALIZE、および \$NUMBER の比較

\$ISVALIDNUM、\$NORMALIZE、および \$NUMBER 関数はすべて、数値を検証します。\$ISVALIDNUM は、ブーリアン値 (0 または 1) を返します。\$NORMALIZE と \$NUMBER は、指定された数を検証して返します。

これら 3 つの関数では、検証の条件は異なります。必要に応じて、それぞれの関数を使い分けてください。

- アメリカ形式の数は、3 つすべての関数で検証されます。ヨーロッパ形式の数は、\$NUMBER 関数のみで検証されます。
- これら 3 つの関数は、符号付き、または符号なしの整数 (-0 を含む)、科学的記数法の数 ("E" または "e" 付き)、および小数部分を持つ数を検証します。ただし、\$NUMBER は小数部分を持つ数 (負の 10 進数の指数を持つ科学的記数法を含む) を拒否するように ("I" 形式を使用して) 設定できます。3 つの関数はすべて、数 (123.45) と数値文字列 ("123.45") の両方を解析します。
- この 3 つの関数では、先頭と末尾の 0 は削除されます。10 進文字は、ゼロでない値が続かない限り削除されます。
- DecimalSeparator: \$NUMBER は、format 引数 (または現在のロケールに対する既定値) に基づいて、小数点文字 (アメリカ形式はピリオド (.), ヨーロッパ形式はコンマ (,)) を検証します。現在のロケール設定に関わらず、その他の関数は、アメリカ形式の 小数のみを検証します。
- NumericGroupSeparator: \$NUMBER は、NumericGroupSeparator 文字 (アメリカ形式はコンマ (,) または空白スペース、ヨーロッパ形式はピリオド (.) または空白スペース) を許可します。また、位置に関わらず、任意の数の NumericGroupSeparator 文字を許可し、削除します。例えばアメリカ形式は、数 123456.99 として "12 3,,4,56.9,9" を検証します。\$NORMALIZE は、NumericGroupSeparator 文字を認識しません。この関数は、数値以外の文字を検出するまで、文字ごとに検証します。例えば "123,456.99" を、数 123 として検証します。\$ISVALIDNUM は、文字列 "123,456" を無効な数値として拒否します。
- 先頭の複数の符号 (+ と -) は、数に対する 3 つすべての関数によって解釈されます。ただし、\$NORMALIZE だけは、引用符付き数値文字列で先頭の複数の符号を受け入れます。
- 末尾の + と - 符号。3 つの関数ではすべて、数の末尾の符号を受け入れません。しかし、引用符付き数値文字列では、\$NUMBER は末尾の符号を 1 つ (唯一) 解析し、\$NORMALIZE は複数の末尾の符号を解析し、\$ISVALIDNUM は末尾に符号を含む文字列をすべて無効な数として受け入れません。
- 括弧。\$NUMBER は、引用符付き文字列内の符号なしの数を囲む括弧を、負の数を示すものとして解析します。\$NORMALIZE と \$ISVALIDNUM は、括弧を受け入れません。
- 複数の 10 進文字を含む、数値文字列。\$NORMALIZE は 2 番目の 10 進文字を検出するまで、文字ごとに検証を行います。例えばアメリカ形式は、"123.4.56" を数 123.4 として検証します。\$NUMBER と \$ISVALIDNUM は、1 つ以上の 10 進文字を含む任意の文字列を無効な数として受け入れません。

他の数値以外の文字を含む数値文字列: \$NORMALIZE は、アルファベット文字を検出するまで、文字ごとに検証を行います。これは、"123A456" を数 123 として検証します。\$NUMBER と \$ISVALIDNUM は文字列全体を検証し、"123A456" を無効な数として受け入れません。



- ・ NULL 文字列。\$NORMALIZE は、NULL 文字列をゼロ (0) として解析します。\$NUMBER と \$ISVALIDNUM は、NULL 文字列を受け入れません。

\$ISVALIDNUM 関数と \$NUMBER 関数は、オプションで min/max の範囲チェックを行います。

\$ISVALIDNUM、\$NORMALIZE、および \$NUMBER はすべて、指定された小数桁数に数を丸めます。\$ISVALIDNUM と \$NORMALIZE は、小数桁数の丸め、および小数部を持つ数の丸めまたは切り捨てを行って、整数を返します。例えば、\$NORMALIZE は 488.65 を 488.7 または 489 に丸めるかまたは、488 に切り捨てることができます。\$NUMBER は、小数桁および整数桁の両方を丸めることができます。例えば、\$NUMBER は 488.65 を 488.7、489、490、または 500 に丸めることができます。

## 関連項目

- ・ [\\$DOUBLE 関数](#)
- ・ [\\$FNUMBER 関数](#)
- ・ [\\$INUMBER 関数](#)
- ・ [\\$ISVALIDDDOUBLE 関数](#)
- ・ [\\$NORMALIZE 関数](#)
- ・ [\\$NUMBER 関数](#)
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$isvector (ObjectScript)

ベクトル値を検証し、ブーリアン値を返します。

```
$isvector(expr)
```

### 説明

\$isvector 関数は、入力引数がベクトルかどうかを判断します。入力引数がベクトルの場合、この関数は 1 (true) を返します。ベクトルは任意の有効なベクトルの型、任意の長さにすることができ、要素は連続していなくてもかまいません。有効なベクトルの型は、decimal、double、integer、string、および timestamp です。

入力引数がベクトルでない場合、この関数は 0 (false) を返します。

### 引数

**expr**

任意の ObjectScript 式。

### 例

次の例では、1 から 100 までのランダムな整数からなる長さ 10 のベクトルを定義して、1 つおきに要素を定義し、定義したベクトルがベクトルであるかどうかをチェックして、その結果を書き込みます。また、文字列がベクトルかどうかをチェックして、その結果も書き込みます。

```
for i=2:2:10 set $vector(vector,i,"integer") = $random(100)+1
set isvec = $isvector(vector)
write isvec // writes 1

set notvec = "This is not a vector."
set notvec = $isvector(notvec)
write notvec // writes 0
```

### 関連項目

- [\\$vector](#)
- [\\$vectorop](#)
- [\\$LIST](#)

## \$JUSTIFY (ObjectScript)

指定された幅の範囲内で式を右に寄せる関数です。指定された小数桁数に丸めます。

### 構文

```
$JUSTIFY(expression,width,decimal)
$J(expression,width,decimal)
```

### 引数

引数	説明
expression	右寄せされる値。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。
width	右寄せされる expression 内の文字数。正の整数、または正の整数に評価される式。
decimal	オプション - 小数桁数。正の整数、または正の整数に評価される式。InterSystems IRIS は、expression 内の小数桁の数をこの値まで丸めるかパディングします。decimal を指定すると、InterSystems IRIS は expression を数値として扱います。

### 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 **キャノニック形式の数値** 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 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(.) またはヨーロッパ形式のロケール用のコンマ(,) のどちらかです。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")
```

一般に、\$JUSTIFY は、小数桁がある数値を形式設定するために使用されます。すべての数値の小数桁数が同じになり、DecimalSeparator 文字が数値列内で揃うようにそれらの数値が右寄せされます。\$JUSTIFY は、特に WRITE コマンドを使用してフォーマットされた値を出力するのに便利です。

### 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すことができる)。

- ・ 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。NULL 文字列("") を指定すると、指定した width の空白スペースの文字列を作成できます。

- 数値揃えが必要な場合は、decimal を指定します。decimal を指定すると、\$JUSTIFY は expression を [キャノニック形式の数値](#) に変換します。それにより、先頭のプラス符号とマイナス符号が解決され、先頭および末尾のゼロが削除されます。それにより、最初の非数値文字列で expression が切り捨てられます。expression の先頭が非数値文字列 (通貨記号など) の場合、\$JUSTIFY は expression の値を 0 に変換します。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 からキャノニック形式の数値への変換後に、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埋めてから、その結果を右寄せします。\$JUSTIFY では、NumericGroupSeparator 文字、通貨記号、複数の DecimalSeparator 文字、および末尾のプラス符号やマイナス符号は認識されません。

## 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 値が 0、NULL 文字列 (''), または非数値文字列の場合、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は、\$JUSTIFY は小数部をこの小数桁までゼロで埋めて、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 値が、decimal 値には関係なく \$JUSTIFY によって変更されずに返されます。

## 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。

### ObjectScript

```
WRITE ">",$JUSTIFY("right",10),"<",<br>WRITE ">",$JUSTIFY("aligned",10),"<",<br>WRITE ">",$JUSTIFY("+0123.456",10),"<",<br>WRITE ">",$JUSTIFY("string longer than width",10),"<",<!--
```

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。

### ObjectScript

```
SET var1 = 250.50999
SET var2 = 875
WRITE !,$JUSTIFY(var1,20,2),!,$JUSTIFY(var2,20,2)
WRITE !,$JUSTIFY("_____",20)
WRITE !,$JUSTIFY("TOTAL",9),$JUSTIFY(var1+var2,11,2)
```

以下の行を返します。

```
                250.51
                875.00
TOTAL          1125.51
```

以下の例では、\$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

```
SET rtn=##class(%SYSTEM.Process).IEEEEError(0)
SET x=$DOUBLE(1.2e500)
WRITE !,"Double: ",x
WRITE !,">", $JUSTIFY(x,12,2),"<"
SET y=$DOUBLE(x-x)
WRITE !,"Double INF minus INF: ",y
WRITE !,">", $JUSTIFY(y,12,2),"<"
```

## \$JUSTIFY および \$FNUMBER

**\$FNUMBER** を使用して、表示する数値の形式を設定できます。**\$JUSTIFY** および **\$FNUMBER** は、両方とも指定した小数桁まで丸めることができます (またはゼロで埋めることができます)。**\$FNUMBER** は、NumericGroupSeparator 文字を追加するために使用することもできます。ただし、以下の点に注意してください。

- ・ 数値が **\$JUSTIFY** により右寄せされている場合、**\$FNUMBER** でその数値の形式を設定できません(**\$FNUMBER** は、先頭のスペースを非数値文字として解釈します)。
- ・ NumericGroupSeparator 文字や通貨記号を追加した場合、**\$JUSTIFY** で数値揃えを実行できません(**\$JUSTIFY** は、NumericGroupSeparators または通貨記号を非数値文字として解釈します)。

したがって、NumericGroupSeparators の追加、小数桁数の丸め、通貨記号の追加、および結果の数値の右寄せを適切に行うには、**\$FNUMBER** を使用して、丸めおよび NumericGroupSeparators の挿入を行います。その後、**\$JUSTIFY** を 2 引数構文で使用して、結果の文字列を右寄せします。

## ObjectScript

```
SET num=123456.789
SET fmtnum=$FNUMBER(num,"",2)
SET money="$"_fmtnum
SET rmoney=$JUSTIFY(money,15)
WRITE ">",rmoney,"<"
```

## 関連項目

- ・ **\$FNUMBER** 関数
- ・ **\$X** 特殊変数

## \$LENGTH (ObjectScript)

文字列内の文字数または区切られた部分文字列の数を返します。

### 構文

```
$LENGTH(expression,delimiter)
$L(expression,delimiter)
```

### 引数

引数	説明
expression	ターゲット文字列。数値、文字列リテラル、変数名、または文字列に解決される任意の有効な式を指定できます。
delimiter	オプションターゲット文字列にある部分文字列の境界を定める文字列。変数名、数値、文字列リテラル、または文字列に解決される任意の有効な式を指定できます。

### 概要

\$LENGTH は使用される引数によって、指定した文字列内の文字数あるいは指定した文字列内の区切られた部分文字列の数を返します。長さは文字数をカウントします。8 ビット文字および 16 ビットのワイド (Unicode) 文字は、両方とも 1 文字としてカウントされます。詳細は [“Unicode”](#) を参照してください。

- \$LENGTH(expression) は、文字列の文字数を返します。expression が NULL 文字列の場合、\$LENGTH は 0 を返します。expression が数値式の場合、その長さを決定する前に[キャノニック形式](#)に変換されます。expression が文字列数値式の場合、変換は実行されません。expression が [\\$DOUBLE](#) の INF 値、-INF 値、または NAN 値の場合は、返される長さは、それぞれ 3、4、および 3 になります。

この構文は、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。

- \$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。この数は常に文字列内の区切り文字数に 1 を加えたものです。

この構文は、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。

delimiter が NULL 文字列の場合は、\$LENGTH は 0 を返します。区切り文字がその他の有効な文字列リテラルで、文字列が NULL 文字列の場合は、\$LENGTH は 1 を返します。

### エンコードされた文字列

InterSystems IRIS は、内部的なエンコードを含む文字列をサポートしています。このエンコードのため、\$LENGTH を使用して文字列のデータ内容を調べることはできません。

- \$LENGTH は、\$LISTBUILD または \$LIST を使用して作成された[リスト構造文字列](#)には使用できません。InterSystems IRIS のリスト文字列はエンコードされているため、返される長さはリスト要素内の文字列の数を有意に示していません。唯一の例外は、\$LIST で 1 つの引数の形式と 2 つの引数の形式であり、エンコードされた InterSystems IRIS リスト文字列を入力として取りながら、標準の文字列として 1 つのリスト要素の値を出力します。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) の数を調べるには、[\\$LISTLENGTH](#) 関数を使用できます。
- \$LENGTH は、[ビット文字列](#)には使用できません。InterSystems IRIS のビット文字列はエンコードされているため、返される長さはビット文字列内のビット数を有意に示していません。文字列内のビット数を返す [\\$BITCOUNT](#) 関数を使用できます。

- ・ \$LENGTH は、[JSON 文字列](#)には使用できません。変数を JSON オブジェクトまたは JSON 配列に設定することによって割り当てられる値は、オブジェクト参照です。したがって、その変数値の長さはオブジェクト参照の長さであり、JSON 文字列にエンコードされたデータの長さとは関係ありません。

## サロゲート・ペア

\$LENGTH は、サロゲート・ペアを認識しません。サロゲート・ペアは、一部の中国語の文字を表示したり、日本語の JIS2004 標準をサポートするために使用されます。\$WISWIDE 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。\$WLENGTH 関数は、サロゲート・ペアを認識して、正しく解析します。\$LENGTH と \$WLENGTH は、それ以外は同一です。ただし、\$LENGTH は通常 \$WLENGTH より高速なため、サロゲート・ペアが出現しない場合は常に \$LENGTH が推奨されます。

## 例

以下の例では、両方の \$LENGTH 関数は、文字列の文字数 4 を返します。

### ObjectScript

```
SET roman="test"
WRITE !,$LENGTH(roman)," characters in: ",roman
SET greek=$CHAR(964,949,963,964)
WRITE !,$LENGTH(greek)," characters in: ",greek
```

以下の例では、最初の \$LENGTH 関数は 5 を返します。これは、指定した数値のキャノニック・バージョンである、長さ 74000 です。2 番目の \$LENGTH は、文字列 "+007.4e4" の長さ 8 を返します。

### ObjectScript

```
WRITE !,$LENGTH(+007.4e4)
WRITE !,$LENGTH("+007.4e4")
```

以下の例では、最初の WRITE は var1 内の文字数 11 を返します (スペース文字も含む)。2 番目の WRITE は、部分文字列の区切り文字としてスペース文字を使用した場合の var1 内の部分文字列数 2 を返します。

### ObjectScript

```
SET var1="HELLO WORLD"
WRITE !,$LENGTH(var1)
WRITE !,$LENGTH(var1," ")
```

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。

### ObjectScript

```
SET STR="ABC$DEF$EFG",DELIM="$"
WRITE $LENGTH(STR,DELIM)
```

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。

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。

### ObjectScript

```
SET Nstring = ""
WRITE $LENGTH(Nstring)
```

以下の例は、区切り文字やその文字列が NULL 文字列のときに返される値を示しています。



## ObjectScript

```
SET String = "ABC"
SET Nstring = ""
SET Delim = "$"
SET Ndelim = ""
WRITE !,$LENGTH(String,Delim) ; returns 1
WRITE !,$LENGTH(Nstring,Delim) ; returns 1
WRITE !,$LENGTH(String,Ndelim) ; returns 0
WRITE !,$LENGTH(Nstring,Ndelim) ; returns 0
```

## 関連項目

- ・ [\\$EXTRACT](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$WISWIDE](#) 関数
- ・ [\\$WLENGTH](#) 関数

## \$LIST (ObjectScript)

リストの要素を返すか、置換します。

### 構文

```
$LIST(list,position,end)
$LI(list,position,end)
$LIST(list,start1:end1,start2:end2...)
$LI(list,start1:end1,start2:end2...)
SET $LIST(list,position,end)=value
SET $LI(list,position,end)=value
```

### 引数

引数	説明
list	有効なリストとして解釈する式。リストにはエンコードが含まれるので、list は <a href="#">\$LISTBUILD</a> または <a href="#">\$LISTFROMSTRING</a> を使用して作成されるか、あるいは \$LIST を使用して別のリストから抽出されなければなりません。SET \$LIST 構文では、list は変数または多次元プロパティである必要があります。
position	オプション - list の中でサブリストの取得を開始する位置を指定する整数コード。許可される値は、n (list の先頭からの整数カウント)、* (list 内の最後の要素)、および *-n (list の末尾から逆向きの相対オフセット・カウント) です。SET \$LIST 構文は、*+n (list の末尾の先に追加する要素の相対オフセットの整数カウント) もサポートします。したがって、リスト内の 1 つ目の要素は 1、2 つ目の要素は 2、リスト内の最後の要素は * であり、最後から 2 番目の要素は *-1 です。position が省略されている場合は、既定値の 1 が使用されます。  古いコード内でリストの最後の要素を指定するには -1 を使用できます。このような -1 の使用法は非推奨であり、* や *-n または *+n という相対オフセット構文と組み合わせて使用しないでください。
end	オプション - list の中でサブリストの取得を終了する位置を指定するコード。list の先頭からの位置カウントを指定する正の整数、または list の終了からカウントした記号コードを指定できます。position と共に使用し、position と同じコード値を使用する必要があります。省略した場合、position で指定された単一要素のみが返されます。
start1:end1	オプション - list から取得する範囲の開始位置と終了位置を指定する整数のペア。start と end はコロンで接続します。list の先頭から数えた、要素の正の整数カウントのみを指定できます。範囲ペアに、list の最後を超える整数カウントを含めることができます。start は end 以下にする必要があります。  任意の数の start:end の範囲ペアをコンマ区切りリストとして指定できます。範囲ペアは任意の順序で指定できます。範囲ペアは重なっていてもかまいません。また、list 要素を繰り返し使用することができます。

### 概要

\$LIST は次のように使用できます。

- list から [単一の要素を取得](#)します。構文 `$LIST(list,position)` を使用します。list の先頭または末尾からのオフセットで要素の位置を検索します。リストから単一の要素を返します。
- list から [要素の単一の範囲を取得](#)します。これには構文 `$LIST(list,position,end)` が使用されます。list の先頭または末尾からのオフセットで要素を検索します。要素の範囲をリストとして返します。

- list から要素の複数の範囲を取得します。これには構文 `$LIST(list,position1:end1,position2:end2)` が使用されます。list の先頭からのオフセットで要素を検索します。また、取得した範囲を連結し、連結した結果をリストとして返します。
- list 内で要素を置換します。置換要素は、元の要素と同じ長さでも、長くても、短くてもかまいません。これには構文 `SET $LIST(list,position,end)=value` が使用されます。

## 単一の要素を返す

`$LIST(list, position)` 構文:

`$LIST` は単一の要素を返します。返される要素は position 引数で決まります。

- `$LIST(list)` は、list の最初の要素を返します。
- `$LIST(list,position)` は、list から position で指定した位置にある要素を返します。position に、list の末尾を超える正の整数カウントや、list の先頭より前になる負の整数カウントを指定することはできません。

注釈 `$LIST` をループ構造内で使用して、複数の連続した要素値を返さないでください。`$LIST` は実行ごとにリストを最初から評価する必要があるため、これが動作している間、非常に効率が悪くなります。複数の連続した要素値を返す場合、`$LISTNEXT` 関数を使用した方がはるかに効率的です。

## 単一のサブリストを返す

`$LIST(list,position,end)` 構文:

`$LIST` は、要素の単一のサブリストを返します。このサブリストには 1 つ以上の要素を含めることができます。返される要素は、使用する引数によって決まります。

- `$LIST(list,position,end)` は、list から取得した要素の範囲が含まれる“サブリスト”(エンコードされたリスト文字列)を返します。position で指定した開始位置から、end で指定した終了位置(この位置を含む)までが範囲になります。position と end が同じ要素を指定している場合、`$LIST` は、この要素をエンコードされたリストとして返します。  
position に、list の末尾を超える正の整数カウントや、list の先頭より前になる負の整数カウントを指定することはできません。end に、list の末尾を超える正の整数カウントを指定できますが、返されるのは既存の list 要素のみです。要素のパディングは実行されません。

## 複数のサブリストを返す

`$LIST(list,start1:end1,start2:end2)` 構文:

`$LIST` は、要素の複数範囲のサブリストを取得してから、それらを単一の戻りリストに連結します。start:end の範囲を 1 つ以上指定できます。複数の範囲はコンマで区切ります。返される要素が 1 つだけでも、戻り値は常にリスト構造になります。

範囲ペアは、コロンで組み合わせた 2 つの正の整数で構成される必要があります。これにより、list の先頭からカウントした要素の範囲を指定します。start および end の値は範囲に含まれます。start の値は end の値以下でなければなりません。例えば、3:4 は list の 3 番目の要素と 4 番目の要素を取得します。3:3 は list の 3 番目の要素を取得します。4:3 は無効な範囲であるため、要素は取得されずにスキップされます。start および end に正の整数以外の値を指定することはできません。要素は 1 からカウントされます。ゼロは使用しないでください。ゼロを指定すると、1 に変換されます。したがって、1:1、0:1、0:0、および 1:0 はすべて list の最初の要素を取得します。

start または end、あるいはその両方の値に、list の末尾を超える正の整数カウントを指定できます。指定した要素の範囲を含む範囲サブリストが生成されます。範囲に既存の list 要素が含まれる場合、それらはサブリストに含まれます。範囲に list の末尾を超える要素が含まれる場合、サブリストは適切な数の NULL 要素でパディングされます。NULL 要素が含まれる結果を表示するには、`ZWRITE` または `$LISTTOSTRING(list,,1)` を使用する必要があることに注意してください。WRITE では NULL 要素は表示されません。また、`$LISTTOSTRING` (既定) では <NULL VALUE> エラーが生成されます。

この構文は、`$LIST(mylist,2:2,4:7,10:20)` のように、重複しない範囲ペアを昇順で指定する場合に使用すると、最も効率的です。ただし、コンマ区切りの範囲ペアは、任意の順序で指定できます。list の要素を、異なる範囲ペアによって複数回取得できます。

以下の例では、4 つの範囲ペアを取得して、それらを単一のリストに連結します。

### ObjectScript

```
SET mylist=$LISTBUILD("a","b","c","d","e","f","g")
SET result=$LIST(mylist,2:2,1:4,7:9,1:2)
ZWRITE result
```

戻り値: `result=$lb("b","a","b","c","d","g",,"a","b")`

## 引数

### list

1 つ以上の要素を含む、エンコードされたリスト文字列。リストは、`$LISTBUILD` または `$LISTFROMSTRING` を使用して作成するか、`$LIST` 関数を使用して、他のリストから抽出することができます

要素を返す場合、list は変数またはオブジェクト・プロパティにすることができます。

要素を置換するために、等号の左側で SET と共に `$LIST` を使用する場合、list は変数または [多次元プロパティ参照](#) にすることができますが、非多次元オブジェクト・プロパティにすることはできません。

有効な list 引数は以下のとおりです。

### ObjectScript

```
SET myList = $LISTBUILD("Red","Blue","Green","Yellow")
WRITE !,$LIST(myList,2) ; prints Blue
SET subList = $LIST(myList,2,4)
WRITE !,$LIST(subList,2) ; prints Green
```

以下の例では、subList は有効な list 引数ではありません。ここでは、subList は通常の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からです。

### ObjectScript

```
SET myList = $LISTBUILD("Red","Blue","Green","Yellow")
SET subList = $LIST(myList,2)
WRITE $LIST(subList,1)
```

SET `$LIST` 構文形式では、list を非多次元オブジェクト・プロパティにすることはできません。

### position

返される (または置換される) リスト要素の位置 (要素カウント)。1 つの要素が返されます。リスト要素は 1 からカウントされます。position が省略されると、`$LIST` は最初の要素を返します。

- position が正の整数の場合、`$LIST` は list の先頭から要素をカウントします。position が list 内の要素数よりも大きい場合、InterSystems IRIS は `<NULL VALUE>` エラーを発行します。
- position が \* (アスタリスク) の場合、`$LIST` は list 内の最後の要素を返します。
- position が \*-n (アスタリスクとそれに続く負の数) の場合、`$LIST` は list の末尾からオフセット分だけ逆向きに要素をカウントします。したがって、\*-0 はリスト内の最後の要素であり、\*-1 は最後から 2 番目のリスト要素 (末尾からのオフセットが 1) です。position の相対オフセット・カウントが list 内の要素数と等しい (結果的に、0 番目の要素を指定している) 場合、InterSystems IRIS は `<NULL VALUE>` エラーを発行します。position の相対オフセット・カウントが list 内の要素数よりも大きい場合、InterSystems IRIS は `<RANGE>` エラーを発行します。

- SET \$LIST 構文のみ – position が `*+n` (アスタリスクとそれに続く正の整数) の場合、SET \$LIST は list の末尾のオフセット分先の位置に要素を追加します。つまり、`*+1` の場合は list の末尾の先に 1 つの要素が追加され、`*+2` の場合は list の末尾の 2 つ先の位置に 1 つの要素が追加されます (末尾と追加位置の間には NULL 文字列の要素が埋め込まれます)。
- position が 0 または `-0` の場合、InterSystems IRIS は <NULL VALUE> エラーを発行します。

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。要素の範囲は、常にエンコードされたリスト文字列として返されます。(position と end が同じ値のとき) 要素が 1 つだけ返されますが、その値はエンコードされた文字列として返されます。つまり、`$LIST(x,2)` は同じ要素ですが、`$LIST(x,2,2)` と同じデータ値にはなりません。

## end

整数で指定される、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 が小数の場合、整数部に切り捨てられます。

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。このような文字列はエンコードされているため、他の \$LIST 関数によってのみ処理される必要があります。

- position < end : end と position が正の整数であり、position < end である場合、\$LIST は、指定した要素のリストを含むエンコードされたサブリストを返します (position および end の要素を含む)。position が 0 または 1 の場合、サブリストの先頭は list 内の最初の要素になります。end が list 内の要素数よりも大きい場合、\$LIST は position からリストの末尾までのすべての要素を格納しているエンコードされたサブリストを返します。end が `*-n` の場合、position は正の整数か、この end 位置以上の `*-n` になります。つまり、`$LIST(fourlist,*-1,*)`、`$LIST(fourlist,*-3,*-2)`、`$LIST(fourlist,2,*-1)` は、すべて有効なサブリストになります。
- position = end : end および position が同じ要素に評価される場合、\$LIST は、その 1 つの要素を含むエンコードされたサブリストを返します。例えば、4 要素のリストでは、end と position が同じになる可能性も (`$LIST(fourlist,2,2)`、`$LIST(fourlist,*,*)`、または `$LIST(fourlist,*-2,*-2)`)、同じ要素を指定する可能性も (`$LIST(fourlist,4,*)`、`$LIST(fourlist,3,*-1)`) があります。
- position > end : position > end の場合、\$LIST は NULL 文字列 ("" ) を返します。例えば、4 要素のリストでは、`$LIST(fourlist,3,2)`、`$LIST(fourlist,7,*)`、または `$LIST(fourlist,*-1,*-2)` は、どれも NULL 文字列を返します。
- position=0、end : end が指定されていて、position がゼロ (0) または位置 0 に評価される負のオフセットの場合、position 0 は 1 と等しくなります。したがって、end がゼロより大きい要素の位置に評価される場合、\$LIST は、position 1 から end 位置までの要素を含むエンコードされたサブリストを返します。end が要素位置 0 に評価される場合も、position > end となるため、\$LIST は NULL 文字列 ("" ) を返します。
- SET \$LIST 構文のみ – end が `*+n` (アスタリスクとそれに続く正の整数) の場合、SET \$LIST は list の末尾のオフセット分先の位置に要素の範囲を追加します。position が `*+n` の場合、SET \$LIST は値の範囲を追加します。position が正の整数または `*-n` の場合、SET \$LIST は値の置換と追加の両方を行います。最後の要素を置換して要素を追加するには、SET \$LIST(mylist,\*+0,\*+n) を指定します。指定した範囲の開始位置が list の末尾を超える場合、リストには NULL 文字列の要素が必要な分だけ埋め込まれます。end が指定の値の範囲よりも大きい場合、末尾の埋め込みは実行されません。

## 非推奨の -1 の値

古いコードでは、-1 の position または end の値は、リスト内の最後の要素を表します。-1 の値は、\*、\*+n、または \*-n の構文と組み合わせて使用することはできません。

## \*-n および \*+n 引数値の指定

変数を使用して \*-n または \*+n を指定する場合は、常に、引数自体でアスタリスクと符号文字を指定する必要があります。

\*-n の有効な指定内容は以下のとおりです。

### ObjectScript

```
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LIST(alph,*-count)
```

\*+n の有効な指定内容は以下のとおりです。

### ObjectScript

```
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
SET $LIST(alph,*+count)="F"
WRITE $LISTTOSTRING(alph,"^",1)
```

### ObjectScript

```
SET count=-2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LIST(alph,*+count)
```

これらの引数値内では、空白を使用できます。

## \$LIST のエラー

以下の \$LIST 引数値では、エラーが発生します。

- list 引数が有効なリストとして評価されない場合は、\$LIST で <LIST> エラーが発生します。[\\$LISTVALID](#) 関数を使用すると、リストの有効性を判定できます。
- list 引数が NULL 値を含む有効なリストとして評価される場合、またはリストと NULL 値を連結した値である場合、\$LIST(list) 構文を指定すると <NULL VALUE> エラーが発生します。この構文は NULL 値を返そうとするからです。以下はすべて有効なリスト ([\\$LISTVALID](#) に従う) ですが、\$LIST では <NULL VALUE> エラーが発生します。

### ObjectScript

```
WRITE $LIST(""),!
WRITE $LIST($LB()),!
WRITE $LIST($LB(UndefinedVar)),!
WRITE $LIST($LB(),)
WRITE $LIST($LB())_ $LB("a","b","c"))
```

\$LIST(list,position) 構文の position 引数に NULL (存在しない) 要素を指定した場合、\$LIST は <NULL VALUE> エラーを生成します。その理由は、この構文が NULL 値を返そうとするためです。

### ObjectScript

```
SET mylist=$LISTBUILD("A",,"C")
WRITE $LIST(mylist,2) ; generates a <NULL VALUE> error
```

### ObjectScript

```
SET mylist2=$LISTBUILD("A","B","C")
WRITE $LIST(mylist2,4) ; generates a <NULL VALUE> error
```

- \$LIST(list,position) 構文に、0 の position 引数、または 0 番目の要素を指定する負の相対オフセットを指定した場合、\$LIST は <NULL VALUE> エラーを生成します。end が指定されている場合は、0 は position 値として有効で、1 と解釈されます。
- position または end 引数の \*-n の値で、list 内の要素の位置数よりも大きな n の値を指定した場合、\$LIST で <RANGE> エラーが発生します。



## ObjectScript

```
SET list2=$LISTBUILD("Brown","Black")
WRITE $LIST(list2,*-2) ; generates a <NULL VALUE> error
WRITE $LIST(list2,*-3) ; generates a <RANGE> error
```

\$LISTLENGTH( " ") が 0 であるため、\*-1 以上の position または end は <RANGE> エラーになります。

## ObjectScript

```
WRITE $LIST("",*-0) ; generates a <NULL VALUE> error
WRITE $LIST("",*-1) ; generates a <RANGE> error
WRITE $LIST("",0,*-1) ; generates a <RANGE> error
```

- position 引数の値または end 引数の値が -1 未満の場合は、\$LIST で <RANGE> エラーが発生します。

## SET \$LIST を使用した要素の置換

- SET \$LIST(list,position) を使用すると、1 つの要素の値の置換、またはリストへの 1 つの要素の追加を実行できます。この 2 引数形式では、新しい要素の値を指定します。
- SET \$LIST(list,position,end) を使用すると、1 つ以上の要素の削除、1 つ以上の要素の値の置換、またはリストへの 1 つ以上の要素の追加を実行できます。この 3 引数形式では、新しい要素の値をエンコードされたリストとして指定する必要があります。

SET \$LIST は \$LIST(list,start1:end1,start2:end2) 構文をサポートしません。

等号の左側で SET と共に \$LIST を使用する場合、list は有効な変数名にすることができます。変数が存在しない場合は、SET \$LIST が変数を定義します。list 引数は、[多次元プロパティ参照](#)にすることもできますが、非多次元オブジェクト・プロパティにすることはできません。非多次元オブジェクト・プロパティで SET \$LIST を使用しようとすると、<OBJECT DISPATCH> エラーが発生します。

SET (a,b,c,...)=value 構文で左辺に \$LIST を使用することはできません。その代わりに、SET a=value,b=value,c=value,... 構文 (または複数の SET 文) を使用する必要があります。

さらに、[\\$LISTUPDATE](#) を使用して、リスト内の 1 つ以上の要素を置換したり、要素位置を指定して要素をリストに追加したりできます。\$LISTUPDATE はリスト要素を置換し、各要素の置換に対してブーリアン・テストを実行します。SET \$LIST とは異なり、\$LISTUPDATE は初期リストを変更しませんが、指定された要素の置換が行われた初期リストのコピーを返します。

## 2 引数操作

以下の 2 引数操作を実行できます。リストとして要素の値を指定すると、リスト内にサブリストが作成されます。

- 要素の値を新しい値に置き換える

## ObjectScript

```
SET $LIST(fruit,2)="orange" ; count from beginning of list
SET $LIST(fruit,*)="pear" ; element at end of list
SET $LIST(fruit,*-2)="peach" ; offset from end of list
SET $LIST(fruit,2)=" " ; sets the value to the null string
```

- 要素をリストに追加する。\*+n 構文を使用すると、リストの末尾への追加も、リストの末尾より先の位置への追加も可能です。SET \$LIST は、必要に応じて NULL 値の要素を挿入して指定した位置までの間を埋め込みます。

## ObjectScript

```
SET $LIST(fruit,*+1)="plum"
```

- 1 つの要素を要素のサブリストに置き換える



## ObjectScript

```
SET $LIST(fruit,3)=$LISTBUILD("orange","banana")
```

## 3 引数操作

以下の 3 引数 (範囲) 操作を実行できます。範囲操作では、要素の値を 1 つだけ指定する場合でも、その値をリストとして指定する点に注意してください。

- ・ 1 つの要素を複数の要素に置き換える

## ObjectScript

```
SET $LIST(fruit,3,3)=$LISTBUILD("orange","banana")
```

- ・ 要素の値の範囲を新しい値の同じ数に置き換える

## ObjectScript

```
SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana")
```

- ・ 要素の値の範囲を大きな新しい値または小さな新しい値に置き換える

## ObjectScript

```
SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana","peach")
```

- ・ 要素の値の範囲を削除する (ここでは要素の値を NULL 文字列に設定しています。要素の位置は削除していません)

## ObjectScript

```
SET $LIST(fruit,2,3)=$LISTBUILD("", "")
```

- ・ 要素の値の範囲と、それらの位置を削除する

## ObjectScript

```
SET $LIST(fruit,2,3)=""
```

- ・ 要素の範囲をリストに追加する。\*+n 構文を使用すると、リストの末尾への追加も、リストの末尾より先の位置への追加も可能です。SET \$LIST は、必要に応じて NULL 値の要素を挿入して指定した位置までの間を埋め込みます。

## ObjectScript

```
SET $LIST(fruit,*+1,*+2)=$LISTBUILD("plum","pear")
```

SET \$LIST は、指定した要素の値のみを追加します。end の位置が指定の要素数よりも大きい場合、末尾の空の要素の位置は作成されません。

## 例

### \$LIST を使用して要素を返す例

以下の例は、\$LIST の 2 引数形式を使用して、リスト要素を返します。

以下の 2 つの \$LIST 文は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。

## ObjectScript

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LIST(colorlist),!
WRITE $LIST(colorlist,1)
```

以下の 2 つの \$LIST 文はリストの 2 番目の要素である“Orange”を返します。最初の文はリストの先頭からカウントします。2 番目の文はリストの末尾から逆向きにカウントします。

## ObjectScript

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LIST(colorlist,2),!
WRITE $LIST(colorlist,*-4)
```

以下の例は、\$LIST の 3 引数形式を使用して、1 つ以上の要素をエンコードされたリスト文字列として返します。リストには出力不能なエンコード文字が含まれているため、\$LISTTOSTRING を使用して、サブリストを出力可能な文字列に変換する必要があります。

以下の 2 つの \$LIST 文は、エンコードされたリスト文字列として、リストの 5 番目の要素である“Blue”を返します。最初の文はリストの先頭からカウントします。2 番目の文はリストの末尾から逆向きにカウントします。範囲として要素が指定されているため、これは 1 つの要素で構成されたリストとして取得されます。

## ObjectScript

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LISTTOSTRING($LIST(colorlist,5,5))
WRITE $LISTTOSTRING($LIST(colorlist,*-1,*-1))
```

以下の例は“Red,Orange,Yellow”を返します。これは、リスト内の先頭要素から始まり 3 番目の要素で終了する 3 つの要素のリスト文字列です。

## ObjectScript

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LISTTOSTRING($LIST(colorlist,1,3))
```

以下の例は“Green,Blue,Violet”を返します。これは、リスト内の 4 番目の要素から始まり最後の要素で終了する 3 つの要素のリスト文字列です。

## ObjectScript

```
SET colorlist=$LISTBUILD("Red","Orange","Yellow","Green","Blue","Violet")
WRITE $LISTTOSTRING($LIST(colorlist,4,*))
```

以下の例は、プロパティからのリスト要素を返します。

## ObjectScript

```
SET cfg=##class(%iKnow.Configuration).%New("Trilingual",1,$LB("en","fr","es"))
WRITE $LIST(cfg.Languages,2)
```

## SET \$LIST を使用して要素を置換、削除または追加する例

以下の例は、2 つめの要素を置き換える SET \$LIST を示します。

## ObjectScript

```
SET fruit=$LISTBUILD("apple","onion","banana","pear")
WRITE !,$LISTTOSTRING(fruit,"/")
SET $LIST(fruit,2)="orange"
WRITE !,$LISTTOSTRING(fruit,"/")
```

以下の例は、2 つめと 3 つめの要素を置き換える SET \$LIST を示します。

#### ObjectScript

```
SET fruit=$LISTBUILD("apple","potato","onion","pear")
WRITE !,$LISTTOSTRING(fruit,"/")
SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana")
WRITE !,$LISTTOSTRING(fruit,"/")
```

以下の例は、2 つめと 3 つめの要素を 4 つめの要素に置き換える SET \$LIST を示します。

#### ObjectScript

```
SET fruit=$LISTBUILD("apple","potato","onion","pear")
WRITE !,$LISTTOSTRING(fruit,"/")
SET $LIST(fruit,2,3)=$LISTBUILD("orange","banana","peach","tangerine")
WRITE !,$LISTTOSTRING(fruit,"/")
```

以下の例では、リストの末尾に 1 つの要素を追加する SET \$LIST を示します。

#### ObjectScript

```
SET fruit=$LISTBUILD("apple","orange","banana","peach")
WRITE $LL(fruit)," ", $LISTTOSTRING(fruit,"/",1),!
SET $LIST(fruit,*+1)="pear"
WRITE $LL(fruit)," ", $LISTTOSTRING(fruit,"/",1)
```

以下の例では、リストの末尾から 3 つ先に 1 つの要素を追加する SET \$LIST を示します。

#### ObjectScript

```
SET fruit=$LISTBUILD("apple","orange","banana","peach")
WRITE $LL(fruit)," ", $LISTTOSTRING(fruit,"/",1),!
SET $LIST(fruit,*+3)="tangerine"
WRITE $LL(fruit)," ", $LISTTOSTRING(fruit,"/",1)
```

以下の 4 つの例では、\*-n 構文を使用して、リストの末尾からオフセット分の位置の要素を置換する SET \$LIST を示します。SET \$LIST(x,\*-n) と SET \$LIST(x,n,\*-n) では異なる操作が行われます。SET \$LIST(x,\*-n) は、指定した要素の value を置換します。SET \$LIST(x,n,\*-n) は、指定した要素の範囲を削除してから、指定したリストを追加します。

最後から 2 番目の要素を 1 つの値に置き換えるには、SET \$LIST(x,\*-1) を使用します。

#### ObjectScript

```
SET fruit=$LISTBUILD("apple","banana","orange","potato","pear")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
SET $LIST(fruit,*-1)="peach"
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
```

リストの末尾からオフセット分の位置の 1 つの要素を削除するには、SET \$LIST(x,\*-n,\*-n)="" を使用します。

#### ObjectScript

```
SET fruit=$LISTBUILD("apple","banana","orange","potato","pear")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
SET $LIST(fruit,*-1,*-1)=""
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
```

リストの末尾からオフセット分の位置の 1 つの要素を要素のリストに置き換えるには、SET \$LIST(x,\*-n,\*-n)=list を使用します。

## ObjectScript

```
SET fruit=$LISTBUILD("apple","banana","potato","orange","pear")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
SET $LIST(fruit,*-2,*-2)=$LISTBUILD("peach","plum","quince")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
```

リストの末尾からオフセット分の位置の 1 つの要素をサブリストに置き換えるには、`SET $LIST(x,*-n)=list` を使用します。

## ObjectScript

```
SET fruit=$LISTBUILD("apple","banana","potato","orange","pear")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
SET $LIST(fruit,*-2,*-2)=$LISTBUILD("peach","plum","quince")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
```

以下の例は、3 つめの要素からリストの最後まで、リストから要素を削除する `SET $LIST` を示します。

## ObjectScript

```
SET fruit=$LISTBUILD("apple","orange","onion","peanut","potato")
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
SET $LIST(fruit,3,*)=""
WRITE !,"list length is ", $LISTLENGTH(fruit)," "
WRITE $LISTTOSTRING(fruit,"/")
```

## 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。

以下の例は、2 つのリストを示しています。y 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。z 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 y=$LISTBUILD("ABC"_$CHAR(68),"XYZ")
SET z=$LISTBUILD("ABC"_$CHAR(960),"XYZ")
WRITE !,"The ASCII list y elements: "
ZZDUMP $LIST(y,1)
ZZDUMP $LIST(y,2)
WRITE !,"The Unicode list z elements: "
ZZDUMP $LIST(z,1)
ZZDUMP $LIST(z,2)
```

InterSystems IRIS は、z 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。z 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。

## \$EXTRACT と \$PIECE と比較した \$LIST

`$LIST` は、リストの先頭 (または末尾) から要素 (文字ではありません) をカウントすることで、エンコードされたリストから 1 つの要素を決定します。

`$EXTRACT` は、文字列の先頭 (または末尾) から文字をカウントすることで、部分文字列を決定します。`$EXTRACT` は、入力として通常の文字列を取ります。

`$PIECE` は、文字列内のユーザ定義の区切り文字をカウントすることにより、部分文字列を決定します。`$PIECE` は、区切り文字として使用するための文字 (または文字列) の複数のインスタンスを含む通常の文字列を入力として取ります。

\$LIST は通常の文字列では使用できません。\$PIECE および \$EXTRACT はエンコードされたリストでは使用できません。

## 関連項目

- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#)
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数

## \$LISTBUILD (ObjectScript)

指定した式から要素のリストを作成します。

### 構文

```
$LISTBUILD(element,...)
$LB(element,...)

SET $LISTBUILD(var1,var2,...)=list
SET $LB(var1,var2,...)=list
```

### 引数

引数	説明
element	リスト要素値を指定する式です。単一の式、またはコンマ区切りの式リスト内の式を指定できます。省略された要素に対して、プレースホルダとしてのコンマを指定できます。
var	単一の変数、またはコンマ区切りの変数リスト内の変数を指定できます。省略された変数に対して、プレースホルダとしてのコンマを指定できます。var には、ローカル、プロセス・プライベート、グローバル、添え字なし、添え字付きなどの種類の変数を指定できます。
list	有効なリストとして評価される式。リストにはエンコードが含まれるので、list は \$LISTBUILD または <a href="#">\$LISTFROMSTRING</a> を使用して作成されるか、あるいは <a href="#">\$LIST</a> を使用して別のリストから抽出されなければなりません。

### 概要

\$LISTBUILD には、\$LISTBUILD と SET \$LISTBUILD の 2 つの構文形式があります。

- \$LISTBUILD(element1,element2,...) は 1 つ以上の式を取り、1 つの式に対して 1 つの要素を持つエンコードされたリスト構造を返します。要素は、指定された順序でリスト内に配置されます。要素は 1 からカウントされます。
- SET \$LISTBUILD(var1,var2,...)=list は、リストから複数の要素を取り出して、変数に取り込みます。これは、SET \$LISTGET(var1,var2,...)=list に類似しています。これらは、明示的な値が割り当てられていない変数の処理方法において違いがあります。SET \$LISTGET は、そういった変数を定義して (これにより <UNDEFINED> エラーが回避されます)、空の文字列 (NULL 値、前の値があれば上書き) を割り当てます。SET \$LISTBUILD はそういった変数を定義することはありません。変数に前の値がない場合は <UNDEFINED> エラーが生成され、変数に前の値がある場合はその値が保持されます。

### \$LISTBUILD(element1,element2,...)

リストの作成には、以下の関数を使用できます。

- \$LISTBUILD は、複数のデータ項目 (文字列または数値) から、データ項目ごとに 1 つのリスト要素を含むリストを作成します。\$LISTBUILD を使用して、データを含まないリスト要素を作成することもできます。
- \$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。
- NULL 文字列 ("" ) も、有効なリストと見なされます。NULL リスト (要素を含まないリスト) を表すには NULL 文字列 ("" ) を使用します。\$LISTLENGTH ( "" ) は、リスト要素を含まないため、0 の要素数を返します。
- [\\$CHAR の特定の出力不能文字の組み合わせ](#) (\$CHAR(1)、\$CHAR(2,1)、\$CHAR(3,1,asciicode) など) により、エンコードされた空のリストまたは 1 要素のリストを返すこともできます。

[\\$LISTVALID](#) 関数を使用すると、式が有効なリストであるかどうかを判定できます。

\$LISTBUILD は、\$LISTDATA、\$LISTFIND、\$LISTGET、\$LISTNEXT、\$LISTLENGTH、\$LISTSAME、\$LISTTOSTRING などの他の \$LIST 関数と共に使用します。

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方法で格納します。ワイド文字はバイト文字列として記述されます。

**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、エンコードされたリストを比較するときに、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に、異なる内部表現がある場合もあります。例えば、\$LISTBUILD(1) は \$LISTBUILD("1") と等しくなく、\$LISTBUILD(1.0) は \$LISTBUILD(1) と等しくありません。しかし、\$LIST および \$LISTTOSTRING などのリスト表示関数は、[キャノニック形式](#)で数値リスト要素値を返します。そのため、\$LIST(\$LISTBUILD(1),1)=\$LIST(\$LISTBUILD("1"),1) となります。

同じ理由から、\$LISTBUILD によって返されたエンコードされたリスト値は、\$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。

## SET \$LISTBUILD

SET コマンドの等号の左側で使用された場合、\$LISTBUILD 関数は、単一の処理でリストから複数の要素を取り出します。構文は、以下のとおりです。

```
SET $LISTBUILD(var1,var2,...)=list
```

SET \$LISTBUILD の var 引数は、コンマで区切られた変数のリストです。これらの変数はそれぞれ、対応する位置にある list 要素の値に設定されます。したがって、var1 は、最初の list 要素の値に設定され、var2 は 2 つ目の list 要素の値に設定されるようになります。var 引数が既存の変数である必要はありません。変数は、SET \$LISTBUILD によって値に割り当てられるときに、定義されます。

- var 引数の数は、list 要素の数より少なくても、多くてもかまいません。未指定の var 値では、前の値が保持されます。前に値が定義されていない場合は、未定義のままです。この動作を [SET \\$LISTGET](#) と比較してください。余分な list 要素は無視されます。
- var 引数と list 要素のいずれかまたは両方に、プレースホルダのコンマで表された、省略された値を含めることができます。省略された var 引数は未定義です。list 要素を省略すると、対応する var 値で前の値が維持されます。前に値が定義されていない場合は、未定義のままです。この動作を [SET \\$LISTGET](#) と比較してください。

SET \$LISTBUILD はアトミック処理です。コンパイル済みのプログラムでの var 引数の最大数は 1024 個です。ターミナルから実行した場合の var 引数の最大数は 128 個です。これらの制限を超過しようとする、<SYNTAX> エラーが発行されます。

var 引数がオブジェクト・プロパティ (object.property) である場合、プロパティは多次元でなければなりません。すべてのプロパティは、オブジェクト・メソッド内で [i%property インスタンス変数](#)として参照できます。

以下の例では、(等号の右側にある) \$LISTBUILD によって、4 つの要素を使用してリストが作成されます。

以下の例では、SET \$LISTBUILD によって、リストから最初の 2 つの要素が取り出され、2 つの変数に取り込まれます。

### ObjectScript

```
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(a,b)=colorlist
WRITE "a=",a," b=",b /* a="red" b="blue" */
```

以下の例では、SET \$LISTBUILD によって、リストから要素が取り出され、5 つの変数に取り込まれます。指定された list には 5 つ目の要素がないため、対応する var 変数 (e) には、その前の値が含まれます。



## ObjectScript

```
SET (a,b,c,d,e)=0
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTBUILD(a,b,c,d,e)=colorlist
WRITE "a=",a," b=",b," c=",c," d=",d," e=",e
/* a="red" b="blue" c="green" d="white" e=0 */
```

以下の例では、SET \$LISTBUILD によって、リストから要素が取り出され、4 つの変数に取り込まれます。指定された list には 3 つ目の要素がないため、対応する var 変数 (c) には、その前の値が含まれます。

## ObjectScript

```
SET (a,b,c,d)=0
SET colorlist=$LISTBUILD("red","blue",,"white")
SET $LISTBUILD(a,b,c,d)=colorlist
WRITE "a=",a," b=",b," c=",c," d=",d
/* a="red" b="blue" c=0 d="white" */
```

以下の例では、SET \$LISTBUILD によって、リストから要素が取り出され、4 つの変数に取り込まれます。list の 3 つ目の要素は入れ子になったリストであるため、対応する var 変数 (c) にはリスト値が含まれます。

## ObjectScript

```
SET (a,b,c,d)=0
SET colorlist=$LISTBUILD("red","blue",$LISTBUILD("green","yellow"),"white")
SET $LISTBUILD(a,b,c,d)=colorlist
WRITE "a=",a," b=",b," c=",c," d=",d
/* a="red" b="blue" c=$LB("green","yellow") d="white" */
```

## 例

ここに示す例の多くでは、[\\$LISTTOSTRING](#) 関数を使用して、表示のために \$LISTBUILD 戻り値に変換します。  
\$LISTBUILD は直接表示できないエンコードされた文字列を返します。

以下の例は、3つの要素を持つリスト "Red,Blue,Green" を作成します。

## ObjectScript

```
SET colorlist=$LISTBUILD("Red","Blue","Green")
WRITE $LISTTOSTRING(colorlist,"^")
```

以下の例では、"3°0'44"5.6"33'400" と表示される 6 つの数値要素のリストを作成します。\$LISTBUILD は、キャノニック形式と同じでない場合がある、最適化された 2 進数表記に基づいて、数値要素値をエンコードします。\$LIST および \$LISTTOSTRING などのリスト表示関数は、キャノニック形式で数値要素値を返します。

## ObjectScript

```
SET numlist=$LISTBUILD(003,0.00,44.0000000,5.6,+33,4E2)
WRITE $LISTTOSTRING(numlist,"^")
```

## 要素の省略

要素式を省略すると、エンコードされた要素が定義されますが、その要素のデータ値は定義されません。

以下の例では、\$LISTBUILD 文はどちらも、2 つ目の要素に未定義の値がある、有効な 3 要素のリストを作成します。要素を省略することと、要素に未定義の変数を指定することで、生じる結果はまったく同じです。\$LISTBUILD は、未定義の変数をリスト要素として取ることができ、これによってエラーが生成されることはありません。また、結果のリストは \$LISTVALID のテストに合格します。しかし、この未定義のリスト要素を \$LIST や \$LISTTOSTRING などのリスト関数で参照すると、<NULL VALUE> エラーが生成されます。

## ObjectScript

```
KILL a
SET list1=$LISTBUILD("Red",,"Green")
SET list2=$LISTBUILD("Red",a,"Green")
WRITE "List lengths:",$LISTLENGTH(list1)," ",$LISTLENGTH(list2),!
IF $LISTVALID(list1)=1,$LISTVALID(list2)=1 {
    WRITE "These are valid lists",! }
IF list1=list2 {WRITE "and they're identical"}
ELSE {WRITE "They're not identical"}
```

以下の例は、未定義の要素を、リスト内だけでなく、リストの末尾にも指定できることを示しています。末尾に未定義の要素があるリストは、有効なリストです。しかし、この未定義の要素をリスト関数で参照すると、〈NULL VALUE〉エラーが生成されます。

## ObjectScript

```
KILL z
SET list3=$LISTBUILD("Red",)
SET list4=$LISTBUILD("Red",z)
WRITE "List lengths:",$LISTLENGTH(list3)," ",$LISTLENGTH(list4),!
IF $LISTVALID(list3)=1,$LISTVALID(list4)=1 {
    WRITE "These are valid lists",! }
IF list3=list4 {WRITE "and they're identical"}
ELSE {WRITE "They're not identical"}
```

しかし、以下の例では、2 つ目の要素にデータ値 (空の文字列) を持つ 3 つの要素のリストを生成します。2 つ目の要素を参照する際に、エラーは発生しません。

## ObjectScript

```
SET list5=$LISTBUILD("Red","", "Green")
SET list5len=$LISTLENGTH(list5)
WRITE "List length: ",list5len,!
FOR i=1:1:list5len {
    WRITE "Element ",i," value: ",$LIST(list5,i),! }
```

## データを含まないリストまたは NULL 文字列データを含むリスト

\$LISTBUILD を使用して作成されたリストには、少なくとも 1 つのエンコードされたリスト要素が含まれます。その要素には、データが含まれる場合と含まれない場合があります。\$LISTLENGTH はデータではなく要素をカウントするため、\$LISTBUILD を使用して作成されたリストのリスト長さは 1 以上になります。

データ値が未定義である \$LISTBUILD 要素を参照すると、〈NULL VALUE〉エラーが生成されます。以下は、すべて“空”のリストを作成する有効な \$LISTBUILD 文です。しかし、前述のリストのような要素を参照しようとする、〈NULL VALUE〉エラーになります。

## ObjectScript

```
TRY {
    SET x=$LISTBUILD(UndefinedVar)
    SET y=$LISTBUILD(,)
    SET z=$LISTBUILD()
    IF $LISTVALID(x)=1,$LISTVALID(y)=1,$LISTVALID(z)=1 {
        WRITE "These are valid lists",! }
    WRITE "$LB(UndefinedVar) contains ",$LISTLENGTH(x)," elements",!
    WRITE "$LB(,) contains ",$LISTLENGTH(y)," elements",!
    WRITE "$LB() contains ",$LISTLENGTH(z)," elements",!
    /* Attempt to use null lists */
    WRITE "$LB(UndefinedVar) list value ",$LISTTOSTRING(x,"^"),!
    WRITE "$LB(,) list value ",$LISTTOSTRING(y,"^"),!
    WRITE "$LB() list value ",$LISTTOSTRING(z,"^"),!
}
CATCH exp { WRITE !,"In the CATCH block",!
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception",!
        WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
        WRITE "Location: ",exp.Location,!
        WRITE "Code: "
    }
    ELSE { WRITE "Some other type of exception",! RETURN }
    WRITE exp.Code,! }
```

```

        WRITE "Data: ",exp.Data,!
    RETURN
}

```

データに NULL 文字列値が含まれていますが、以下は、データを含むリスト要素を作成する有効な \$LISTBUILD 文です。

### ObjectScript

```

SET x=$LISTBUILD("")
WRITE "list contains ", $LISTLENGTH(x), " elements", !
WRITE "list value is ", $LISTTOSTRING(x, "^"), !
SET y=$LISTBUILD($CHAR(0))
WRITE "list contains ", $LISTLENGTH(y), " elements", !
WRITE "list value is ", $LISTTOSTRING(y, "^"), !

```

## 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。

### ObjectScript

```

SET nlist=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
WRITE "Nested list length is ", $LISTLENGTH($LIST(nlist,3)), !
WRITE "Full list length is ", $LISTLENGTH(nlist), !
WRITE "List is ", $LISTTOSTRING(nlist, "^"), !

```

## リストの連結

連結演算子 ( ) を使用して 2 つのリストを連結すると、その 2 つのリストを結合した別のリストが生成されます。

以下の例では、2 つのリストを連結することで、LISTBUILD を使用して作成したのと同じ要素を持つ同一のリストを作成します。

### ObjectScript

```

SET list1=$LISTBUILD("A","B")
SET list2=$LISTBUILD("C","D","E")
SET clist=list1_list2
SET list=$LISTBUILD("A","B","C","D","E")
IF clist=list {WRITE "they're identical",!}
ELSE {WRITE "they're not identical",!}
WRITE "concatenated ", $LISTTOSTRING(clist, "^"), !
WRITE "same list as ", $LISTTOSTRING(list, "^"), !

```

文字列をリストに連結することはできません。そのようにすると、結果に最初にアクセスしようとしたときに、<LIST> エラーが生成されます。

### ObjectScript

```

TRY {
SET list=$LISTBUILD("A","B")_ "C"
WRITE "$LISTBUILD completed without error", !
SET listlen=$LISTLENGTH(list)
WRITE "$LISTLENGTH completed without error", !
SET listval=$LISTTOSTRING(list, "^")
WRITE "$LISTTOSTRING completed without error", !
}
CATCH exp { WRITE !, "In the CATCH block", !
    IF l=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception", !
        WRITE "Name: ", $ZCVT(exp.Name, "O", "HTML"), !
        WRITE "Location: ", exp.Location, !
        WRITE "Code: "
    }
    ELSE { WRITE "Some other type of exception", ! RETURN }
    WRITE exp.Code, !
    WRITE "Data: ", exp.Data, !
    RETURN
}
}

```

連結の詳細は、“[文字列連結 \(\)](#)” を参照してください。

## 関連項目

- ・ [SET](#) コマンド
- ・ [ZZDUMP](#) コマンド
- ・ [\\$LIST](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#)
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数

## \$LISTDATA (ObjectScript)

指定した要素が存在してデータ値を持つかを示します。

### 構文

```
$LISTDATA(list,position,var)
$LD(list,position,var)
```

### 引数

引数	説明
list	有効なリストに評価する式
position	オプション - 指定したリストで、位置として解釈される式ゼロでない正の整数または -1 です。
var	オプション - 指定されたリスト位置にある要素値を含む変数。\$LISTDATA が値 1 を返す場合、var が書き込まれます。\$LISTDATA が値 0 を返す場合、var は変更されません。

### 概要

\$LISTDATA は、リスト内の指定された要素のデータを調べ、ブーリアン値を返します。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。

\$LISTDATA は、オプションで要素値を var 変数に書き込むことができます。

**注釈** \$LISTDATA をループ構造内で使用して、複数の連続した要素値を返さないでください。\$LISTDATA は実行ごとにリストを最初から評価する必要があるため、これが動作している間、非常に効率が悪くなります。複数の連続した要素値を返す場合、[\\$LISTNEXT](#) 関数を使用した方がはるかに効率的です。

### 引数

#### list

list は、複数の要素を含むエンコードされた文字列です。list は [\\$LISTBUILD](#) または [\\$LISTFROMSTRING](#) を使用して作成されるか、あるいは \$LIST を使用して別のリストから抽出されなければなりません。

[\\$LISTVALID](#) 関数を使用すると、式が有効なリストであるかどうかを判定できます。list 引数内の式が有効なリストとして評価されない場合は、<LIST> エラーが発生します。有効なリストの指定された位置にデータが含まれていない場合、\$LISTDATA は 0 を返します。

#### position

1 からカウントするリスト内の要素の整数位置。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 が存在しないリスト・メンバを参照する場合、\$LISTDATA は 0 を返します。0 の position は常に 0 を返します。position の値が -1 未満の場合に \$LISTDATA 関数を呼び出すと、<RANGE> エラーが発生します。

#### var

\$LISTDATA が値 1 を返す場合、InterSystems IRIS は、要求された要素の値を var に書き込みます。\$LISTDATA が値 0 を返す場合、var は変更されません。var 引数は、添え字付きもしくは添え字なしのローカル、グローバル、またはプロセス・プライベート・グローバル変数のいずれかにできます。これを定義する必要はありません。1 を返す \$LISTDATA

の最初の呼び出しにより var の定義と設定が行われます。\$LISTDATA の最初の呼び出しが 0 を返す場合、var は未定義のままになります。

var 引数は、非多次元オブジェクト・プロパティとすることはできません。非多次元オブジェクト・プロパティに値を書き込もうとすると、<OBJECT DISPATCH> エラーが発生します。

var 引数は、特殊変数とすることはできません。特殊変数に値を書き込もうとすると、<SYNTAX> エラーが発生します。

## 例

以下の 2 つの例では、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 \$LISTDATA 文は、0 を返します。

### ObjectScript

```
KILL y
SET x=$LISTBUILD("Red",,y,"","Green",)
WRITE !,$LISTDATA(x,2) ; second element is undefined
WRITE !,$LISTDATA(x,3) ; third element is a killed variable
WRITE !,$LISTDATA(x,-1) ; the last element is undefined
WRITE !,$LISTDATA(x,0) ; the 0th position
WRITE !,$LISTDATA(x,6) ; 6th position in 5-element list
```

以下の \$LISTDATA 文は、1 を返します。

### ObjectScript

```
SET x=$LISTBUILD("Red",,y,"","Green",)
WRITE !,$LISTDATA(x) ; first element (by default)
WRITE !,$LISTDATA(x,1) ; first element specified
WRITE !,$LISTDATA(x,4) ; fourth element, value=null string
WRITE !,$LISTDATA(x,5) ; fifth element
```

以下の 3 引数の \$LISTDATA 文は、要素値の存在についてテストし、evaluate 変数をその値で更新します。\$LISTDATA が 0 を返す場合、evaluate は変更されないままであることに注意してください。

### ObjectScript

```
SET x=$LISTBUILD("Red",,y,"","Green",)
FOR i=1:1:$LISTLENGTH(x) {
    WRITE "element ",i," data? ",$LISTDATA(x,i,evaluate)," value ",evaluate,!
}
WRITE i," list elements"
```

以下のすべての \$LISTDATA 文は、0 を返します。

### ObjectScript

```
WRITE !,$LISTDATA($LB()) ; null list
WRITE !,$LISTDATA($LB(UndefinedVar)) ; null list
WRITE !,$LISTDATA("") ; null string is a valid list
; but contain no data
WRITE !,$LISTDATA($LB(,)) ; two-element null list
```

以下の \$LISTDATA 文は、1 を返します。

### ObjectScript

```
WRITE !,$LISTDATA($LB("")) ; data is null string
WRITE !,$LISTDATA($LB($CHAR(0))) ; data is non-display character
```

## 関連項目

- ・ [\\$LIST 関数](#)

- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#)
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数



## \$LISTFIND (ObjectScript)

値を要求して指定した list を検索します。

### 構文

```
$LISTFIND(list,value,startafter)
$LF(list,value,startafter)
```

### 引数

引数	説明
list	有効なリストに評価する式。list は、複数の要素を含むエンコードされた文字列です。list は <a href="#">\$LISTBUILD</a> または <a href="#">\$LISTFROMSTRING</a> を使用して作成されるか、あるいは \$LIST を使用して別のリストから抽出されなければなりません。
value	対象の要素値を含む式。
startafter	オプション - リスト位置として翻訳される整数式。検索はこの位置の後の要素から開始されるため、0 は位置 1 から開始することを意味し、1 は位置 2 から開始することを意味します。startafter=-1 は有効な値ですが、常に一致なしが返されます。startafter 値の整数部分のみが使用されます。

### 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。一致は完全で、すべての要素値が含まれる必要があります。文字比較では大文字と小文字が区別されます。数字はキャノニック形式で比較されます。完全一致が見つかったら、\$LISTFIND は一致する要素の位置を返します。value が見つからなければ、\$LISTFIND は 0 を返します。

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。

一致が見つからない場合、\$LISTFIND は 0 を返します。startafter 引数が存在しないリスト・メンバを参照する場合も、\$LISTFIND は同様に 0 を返します。

[\\$LISTVALID](#) 関数を使用すると、list が有効なリストかどうかを判定できます。list が有効なリストでない場合、システムは <LIST> エラーを生成します。

startafter 引数の値が -1 未満の場合、\$LISTFIND 関数を呼び出すと <RANGE> エラーが発生します。

### 空の文字列および空のリスト

以下の例のように、\$LISTFIND 関数を使用して、空の文字列値を検索できます。

#### ObjectScript

```
SET x=$LISTBUILD("A","","C","D")
WRITE $LISTFIND(x,"") ; returns 2
```

\$LISTFIND を省略された要素を含むリストと共に使用することはできますが、省略された要素の検索には使用できません。以下の例では、省略された要素を含むリスト内の値を見つけます。

#### ObjectScript

```
SET x=$LISTBUILD("A",,"C","D")
WRITE $LISTFIND(x,"C") ; returns 3
```

以下の \$LISTFIND の例は 1 を返します。

#### ObjectScript

```
WRITE $LISTFIND($LB(""), "") ; returns 1
```

以下の \$LISTFIND の例は 0 を返します。

#### ObjectScript

```
WRITE $LISTFIND("", ""), ! ; returns 0
WRITE $LISTFIND($LB(), ""), ! ; returns 0
```

以下の list 例は、データを含むリストに連結された空のリストで構成されています。空のリストを追加すると、結果として得られる連結されたリストでの要素のリスト位置が変わります。

#### ObjectScript

```
SET x=$LISTBUILD("A", "B", "C", "D")
WRITE $LISTFIND(x, "B"), ! ; returns 2
WRITE $LISTFIND("_x", "B"), ! ; returns 2
WRITE $LISTFIND($LB(_x, "B"), ! ; returns 3
WRITE $LISTFIND($LB(, , , )_x, "B") ; returns 6
```

ただし、NULL 文字列を value に連結すると、\$LISTFIND には影響を与えません。

#### ObjectScript

```
SET x=$LISTBUILD("A", "B", "C", "D")
WRITE $LISTFIND(x, "B"), ! ; returns 2
WRITE $LISTFIND(x, "B_"), ! ; returns 2
WRITE $LISTFIND(x, "_B"), ! ; returns 2
```

## 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。

#### ObjectScript

```
SET x=$LISTBUILD("A", "B", "C", "D")
WRITE $LISTFIND(x, "B")
```

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

```
SET x=$LISTBUILD("A", "B", "C", "D")
WRITE $LISTFIND(x, "E")
```

以下の例は、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、文字列が見つからず 0 を返します。

#### ObjectScript

```
SET x=$LISTBUILD("A", "B", "C", "D")
WRITE $LISTFIND(x, "B", 2)
```

2 つ目の例では、要求された文字列の 1 つ目が startafter の位置より前に現れるため、要求された文字列の 2 つ目を見つけて 4 を返します。

#### ObjectScript

```
SET y=$LISTBUILD("A", "B", "C", "A")
WRITE $LISTFIND(y, "A", 2)
```

\$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

```
SET mylist = $LISTBUILD("ABC","BCD","BBB")
WRITE $LISTFIND(mylist,"B")
```

一致前に数値がキャノニック形式に変換されるため、以下の数値例はすべて 0 を返します。これらの場合、文字列数値とキャノニック形式の数値は一致しません。

#### ObjectScript

```
SET y=$LISTBUILD("1.0","+2","003","2*2")
WRITE $LISTFIND(y,1.0),!
WRITE $LISTFIND(y,+2),!
WRITE $LISTFIND(y,003),!
WRITE $LISTFIND(y,4)
```

以下の数値例は、数値がキャノニック形式で比較されるため一致します。

#### ObjectScript

```
SET y=$LISTBUILD(7.0,+6,005,2*2)
WRITE $LISTFIND(y,++7.000),! ; returns 1
WRITE $LISTFIND(y,0006),!   ; returns 2
WRITE $LISTFIND(y,8-3),!    ; returns 3
WRITE $LISTFIND(y,--4.0)    ; returns 4
```

以下の例は、指定した startafter 値が一致なしとなるので、すべて 0 を返します。

#### ObjectScript

```
SET y=$LISTBUILD("A","B","C","D")
WRITE $LISTFIND(y,"A",1),!
WRITE $LISTFIND(y,"B",2),!
WRITE $LISTFIND(y,"B",99),!
WRITE $LISTFIND(y,"B",-1)
```

以下の例は、\$LISTFIND を使用して入れ子になったリストを見つけるしくみを示しています。InterSystems IRIS は、複数要素の入れ子になったリストをリスト値のある単一のリスト要素として扱うことに注意してください。

#### ObjectScript

```
SET y=$LISTBUILD("A",$LB("x","y"),"C","D")
WRITE $LISTFIND(y,$LB("x","y"))
```

## 関連項目

- ・ [\\$LIST 関数](#)
- ・ [\\$LISTBUILD 関数](#)
- ・ [\\$LISTDATA 関数](#)
- ・ [\\$LISTFROMSTRING 関数](#)
- ・ [\\$LISTGET 関数](#)
- ・ [\\$LISTLENGTH 関数](#)
- ・ [\\$LISTNEXT](#)
- ・ [\\$LISTSAME 関数](#)
- ・ [\\$LISTTOSTRING 関数](#)

- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数

## \$LISTFROMSTRING (ObjectScript)

文字列からリストを作成します。

### 構文

```
$LISTFROMSTRING(string,delimiter,flag)
$LFS(string,delimiter,flag)
```

### 引数

引数	説明
string	InterSystems IRIS リストに変換される文字列。この文字列には、delimiter によって区切られた 1 つ以上の要素が含まれます。既定では、delimiter は作成される InterSystems IRIS リストの構成要素になりません。
delimiter	オプション — string 内の部分文字列 (要素) を分離するために使用される区切り文字。delimiter は、引用符付きの文字列として指定します。delimiter を指定しない場合、既定はコンマ (,) 文字です。
flag	オプション — 2 ビットのバイナリ・ビット・フラグ。利用可能な値は、0 (00)、1 (01)、2 (10)、および 3 (11) です。既定値は 0 です。

### 概要

\$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 \$LIST 関数を使用して操作されます。

ZWRITE コマンドを使用すると、エンコードされていない形式でリストを表示できます。

### 引数

#### string

文字列リテラル (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。既定では、delimiter 文字は出力リストにありません。したがって、文字列データ要素には delimiter 文字 (文字列) を使用できません。flag 引数の使用に関する以下のセクションでの説明のとおり、flag の値を 2 または 3 に設定した特定の条件下では、出力リストの文字列データ要素に delimiter 文字列を使用できます。

#### 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)。(引用符で囲まれた) 数値または文字列リテラル、変数名、文字列に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される特定の文字が設定されます。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できません。NULL 文字列を指定すると、<ILLEGAL VALUE> エラーになります。

#### flag

2 ビットのバイナリ・ビット・フラグ。

- 1ビットは、string 内で隣接する区切り文字の処理方法を指定します。これらは、エンコードされた戻りリストで省略された要素に対応します。0 は、省略された要素を空の文字列 ("") として表します。1 は、省略された要素を NULL 要素として表します。これを、以下の例に示します。

```
SET colorstr="Red,,Blue"
ZWRITE $LISTFROMSTRING(colorstr,,0)
// $lb("Red","","Blue")
ZWRITE $LISTFROMSTRING(colorstr,,1)
// $lb("Red",,"Blue")
```

- 2ビットは、string 内の引用符の処理方法を指定します。\$LISTFROMSTRING は、string 内の区切られたサブ文字列を引用符付きの文字列要素として返します。既定では、区切った部分文字列に使用している引用符は、リストの中でその文字列に対応する文字列データ要素で保持されます。flag を 2 または 3 に設定すると、区切った部分文字列の先頭と末尾にある引用符は削除され、その部分文字列の中で使用している区切り文字と区切り文字列は区切り文字として扱われません。これらの文字は、リストの中で元の文字列に対応する文字列データ要素の一部として扱われます。これを、以下の例に示します。

```
SET qstr="abc,3,,"New York, New York","004.0","5",""+0.600""
ZWRITE $LISTFROMSTRING(qstr,,0)
// $lb("abc","3",""New York"," New York","004.0",""5",""+0.600"")
ZWRITE $LISTFROMSTRING(qstr,,2)
// $lb("abc","3","New York, New York","004.0","5",""+0.600")
```

## 例

次のように定義された文字列について考えます。

### ObjectScript

```
SET namestring="Deborah Noah Martha Bowie"
```

この文字列をリストに変換するとします。

### ObjectScript

```
SET namelist=$LISTFROMSTRING(namestring," ")
```

以下のテーブルは、結果として得られるリスト要素を示します。

式	値
<code>\$LIST(namelist,1)</code>	Deborah
<code>\$LIST(namelist,2)</code>	Noah
<code>\$LIST(namelist,3)</code>	Martha

## 関連項目

- [\\$LISTTOSTRING](#) 関数
- [\\$LISTBUILD](#) 関数
- [\\$LIST](#) 関数
- [\\$PIECE](#) 関数
- [\\$LISTDATA](#) 関数
- [\\$LISTFIND](#) 関数
- [\\$LISTGET](#) 関数

- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#) 関数
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数



## \$LISTGET (ObjectScript)

list の要素、または要求された要素が未定義の場合は指定した既定値を返します。

### 構文

```
$LISTGET(list,position,default)
$LG(list,position,default)

SET $LISTGET(var1,var2,...)=list
SET $LG(var1,var2,...)=list
```

### 引数

引数	説明
list	有効なリストに評価する式
position	<p>オプション — list 内の開始位置を指定する整数コード。許可される値は、n (list の先頭からのカウント)、* (list 内の最後の要素)、および *-n (list の末尾から逆向きの相対オフセット・カウント) です。したがって、リスト内の 1 つ目の要素は 1、2 つ目の要素は 2、リスト内の最後の要素は * であり、最後から 2 番目の要素は *-1 です。position が小数の場合、整数部に切り捨てられます。position が省略されている場合は、既定値の 1 が使用されます。</p> <p>古いコード内でリストの最後の要素を指定するには -1 を使用できます。このような -1 の使用法は非推奨であり、* や *-n という相対オフセット構文と組み合わせて使用しないでください。</p>
default	<p>オプション — list 要素が未定義の値を持つときに、返り値を作成する式。default が省略されている場合は、既定値の NULL 文字列 (“”) が使用されます。default 値を指定するには position 引数を指定する必要があります。</p>
var	<p>単一の変数、またはコンマ区切りの変数リスト内の変数を指定します。省略された変数に対して、プレースホルダとしてのコンマを指定できます。var には、ローカル、プロセス・プライベート、グローバル、添え字なし、添え字付きなどの種類の変数を指定できます。</p>

### 概要

\$LISTGET には、\$LISTGET と SET \$LISTGET の 2 つの構文形式があります。

- ・ \$LISTGET(list,position,default) は、指定したリスト内の要求した要素を返します。position の値が、存在しない要素を参照する場合や、値が未定義の要素を指定する場合は、default の値が返されます。

\$LISTGET 関数は、\$LIST 関数の 1-引数形式および 2-引数形式とまったく同じですが、\$LIST が <NULL VALUE> エラーを発生する条件下では、\$LISTGET は既定値を返します。<NULL VALUE> エラーが発生する条件に関する詳細は、“\$LIST” 関数の説明を参照してください。

- ・ SET \$LISTGET(var1,var2,...)=list は、リストから複数の要素を取り出して、変数に取り込みます。これは、SET \$LISTBUILD(var1,var2,...)=list に類似しています。これらは、明示的な値が割り当てられていない変数の処理方法において違いがあります。SET \$LISTGET は、そういった変数を定義して (これにより <UNDEFINED> エラーが回避されます)、NULL 文字列値を割り当て、前の値があれば上書きします。SET \$LISTBUILD はそういった変数を定義することはありません。変数に前の値がない場合は <UNDEFINED> エラーが生成され、変数に前の値がある場合はその値が保持されます。

## 引数

### list

リストは、\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。NULL 文字列 ("" ) も、有効なリストと見なされます。[\\$LISTVALID](#) を使用すると、list が有効なリストかどうかを判定できます。無効なリストの場合、\$LISTGET で <LIST> エラーが発生します。

### position

返されるリスト要素の位置 (要素カウント)。要素は、文字列として返されます。リスト要素は 1 からカウントされます。position が省略されると、\$LISTGET は最初の要素を返します。

- ・ position が n (正の整数) の場合、\$LISTGET は list の先頭から要素をカウントします。position の値が list 内の要素数よりも大きい場合、\$LISTGET は default の値を返します。
- ・ position が \* の場合、\$LIST は list 内の最後の要素を返します。
- ・ position が \*-n (アスタリスクとそれに続く負の整数) の場合、\$LIST は list の末尾からの相対オフセット分だけ要素をカウントします。したがって、\*-0 はリスト内の最後の要素であり、\*-1 は最後から 2 番目のリスト要素 (末尾からのオフセットが 1) です。\*-n というオフセットが list の最初の要素の前の位置を指定する場合 (例: 3 つの要素を持つリストに対する \*-3)、\$LISTGET は default の値を返します。\*-n というオフセットがそれよりさらに前の位置を指定する場合は (例: 3 つの要素を持つリストに対する \*-4)、InterSystems IRIS は <RANGE> エラーを発行します。
- ・ position が 0 または -0 の場合、\$LISTGET は default の値を返します。

position 引数の数値部分は整数に評価されます。InterSystems IRIS は、小数を切り捨てて、その整数部分のみにします。position に -1 (リストの最後の要素を示す) を指定することは非推奨であり、新しいコード内では使用しないでください。position の値が -1 より小さい負数である場合は、<RANGE> エラーが発生します。

変数を使用して \*-n を指定する場合は、常に、引数自体でアスタリスクと符号文字を指定する必要があります。

\*-n の有効な指定内容は以下のとおりです。

### ObjectScript

```
SET count=2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LISTGET(alph,*-count,"blank")
```

### ObjectScript

```
SET count=-2
SET alph=$LISTBUILD("a","b","c","d")
WRITE $LISTGET(alph,*+count,"blank")
```

### default

文字列または数値に評価される式。position によって指定された要素が存在しない場合、\$LISTGET は default を返します。この状況が発生する可能性があるのは、position がリストの末尾より後ろである場合、position で値のない要素を指定している場合、position が 0 の場合、または list に要素が含まれていない場合です。ただし、position の値 \*-n によって指定された位置が、list の 0 番目の要素より前の場合は、InterSystems IRIS は <RANGE> エラーを発行します。

default 引数を省略すると、既定値として NULL 文字列 () が返されます。

## SET \$LISTGET

SET コマンドの等号の左側で使われた場合、\$LISTGET 関数は、単一の処理でリストから複数の要素を取り出します。構文は、以下のとおりです。

```
SET $LISTGET(var1,var2,...)=list
```

SET \$LISTGET の var 引数は、コンマで区切られた変数のリストです。これらの変数はそれぞれ、対応する位置にある list 要素の値に設定されます。したがって、var1 は、最初の list 要素の値に設定され、var2 は 2 つ目の list 要素の値に設定されるようになります。var 引数が既存の変数である必要はありません。変数は、SET \$LISTGET によって値に割り当てられるときに、定義されます。

- var 引数の数は、list 要素の数より少なくても、多くてもかまいません。未指定の var 値には NULL 文字列値が割り当てられます。前に値が定義されている場合、前の値は NULL 文字列に置換されます。前に変数が定義されていない場合、変数が定義されます。この動作を [SET \\$LISTBUILD](#) と比較してください。余分な list 要素は無視されます。
- var 引数と list 要素のいずれかまたは両方に、プレースホルダのコンマで表された、省略された値を含めることができます。省略された var 引数は未定義です。list 要素を省略すると、対応する var 値が NULL 文字列に設定されます。前に値が定義されている場合、前の値は削除されます。前に変数が定義されていない場合、変数が定義されます。この動作を [SET \\$LISTBUILD](#) と比較してください。

SET \$LISTGET はアトミック処理です。コンパイル済みのプログラムでの var 引数の最大数は 1024 個です。ターミナルから実行した場合の var 引数の最大数は 128 個です。これらの制限を超過しようとすると、<SYNTAX> エラーが発行されます。

var 引数がオブジェクト・プロパティ (object.property) である場合、プロパティは多次元でなければなりません。すべてのプロパティは、オブジェクト・メソッド内で [!%property](#) インスタンス変数として参照できます。

以下の例では、(等号の右側にある) \$LISTBUILD によって、4 つの要素を使用してリストが作成されます。

以下の例では、SET \$LISTGET によって、リストから最初の 2 つの要素が取り出され、2 つの変数に取り込まれます。

#### ObjectScript

```
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTGET(a,b)=colorlist
WRITE "a=",a," b=",b /* a="red" b="blue" */
```

以下の例では、SET \$LISTGET によって、リストから要素が取り出され、5 つの変数に取り込まれます。指定された list には 5 つ目の要素がないため、対応する var 変数 (e) は NULL 文字列 ("") の値で定義されます。

#### ObjectScript

```
SET (a,b,c,d,e)=0
SET colorlist=$LISTBUILD("red","blue","green","white")
SET $LISTGET(a,b,c,d,e)=colorlist
WRITE "a=",a," b=",b," c=",c," d=",d," e=",e
/* a="red" b="blue" c="green" d="white" e=" */
```

以下の例では、SET \$LISTGET によって、リストから要素が取り出され、4 つの変数に取り込まれます。指定された list には 3 つ目の要素がないため、対応する var 変数 (c) は NULL 文字列 ("") の値で定義されます。

#### ObjectScript

```
SET (a,b,c,d)=0
SET colorlist=$LISTBUILD("red","blue",,"white")
SET $LISTGET(a,b,c,d)=colorlist
WRITE "a=",a," b=",b," c=",c," d=",d
/* a="red" b="blue" c=" " d="white" */
```

以下の例では、SET \$LISTGET によって、リストから要素が取り出され、4 つの変数に取り込まれます。list の 3 つ目の要素は入れ子になったリストであるため、対応する var 変数 (c) にはリスト値が含まれます。

#### ObjectScript

```
SET (a,b,c,d)=0
SET colorlist=$LISTBUILD("red","blue",$LISTBUILD("green","yellow"),"white")
SET $LISTGET(a,b,c,d)=colorlist
WRITE "a=",a," b=",b," c=",c," d=",d
/* a="red" b="blue" c=$LB("green","yellow") d="white" */
```

## 例

以下の例の \$LISTGET 関数は、position で指定されたリスト要素の値を返します (position の既定は 1 です)。

### ObjectScript

```
SET list=$LISTBUILD("A","B","C")
WRITE !,$LISTGET(list)      ; returns "A"
WRITE !,$LISTGET(list,1)   ; returns "A"
WRITE !,$LISTGET(list,3)   ; returns "C"
WRITE !,$LISTGET(list,*)   ; returns "C"
WRITE !,$LISTGET(list,*-1) ; returns "B"
```

以下の例の \$LISTGET 関数は、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 2 つは、ユーザが default 値として定義した疑問符 (?) を返します。その後の 2 つは、ユーザが default 値を指定していないため、NULL 文字列を返します。

### ObjectScript

```
WRITE "returns:",$LISTGET($LISTBUILD("A","","C"),2,"?"),!
WRITE "returns:",$LISTGET($LISTBUILD("A","","C"),*-1,"?"),!
WRITE "returns:",$LISTGET($LISTBUILD("A","","C"),2),!
WRITE "returns:",$LISTGET($LISTBUILD("A","","C"),*-1)
```

以下の例は、リスト内のすべての要素の値を返します。この例は、リストの末尾の前後の位置も列挙します。値が存在しない場合は、default 値が返されます。

### ObjectScript

```
SET list=$LISTBUILD("a","b","","d",,"g")
SET llen=$LISTLENGTH(list)
FOR x=0:1:llen+1 {
    WRITE "position ",x,"=",,$LISTGET(list,x," no value"),!
}
WRITE "end of the list"
```

以下の例は、リスト内のすべての要素の値を逆の順序で返します。値が省略されると、default 値が返されます。

### ObjectScript

```
SET list=$LISTBUILD("a","b","","d",,"g")
SET llen=$LISTLENGTH(list)
FOR x=0:1:llen {
    WRITE "position *-",x,"=",,$LISTGET(list,*-x," no value"),!
}
WRITE "beginning of the list"
```

以下の例の \$LISTGET 関数は、NULL 文字列の list 要素値を返します。これらの関数は、default 値を返しません。

### ObjectScript

```
WRITE "returns:",$LISTGET($LB(""),1,"no value"),!
WRITE "returns:",$LISTGET($LB(""),*,"no value"),!
WRITE "returns:",$LISTGET($LB(""),*-0,"no value")
```

以下の例の \$LISTGET 関数は、すべて default 値を返します。

### ObjectScript

```
WRITE $LISTGET("",1,"no value"),!
WRITE $LISTGET($LB(),1,"no value"),!
WRITE $LISTGET($LB(UndefinedVar),1,"no value"),!
WRITE $LISTGET($LB(,),1,"no value"),!
WRITE $LISTGET($LB(,),*,"no value"),!
WRITE $LISTGET($LB(,),*-1,"no value"),!
WRITE $LISTGET($LB(""),2,"no value"),!
WRITE $LISTGET($LB(""),*-1,"no value")
```

## 関連項目

- ・ [\\$LIST](#) 関数
- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#)
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数

## \$LISTLENGTH (ObjectScript)

指定したリストにある要素の数を返します。

### 構文

```
$LISTLENGTH(list)
$LL(list)
```

### 引数

引数	説明
list	リストに評価する任意の式リストは、\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。

### 概要

\$LISTLENGTH は、list にある要素の数を返します。\$LISTLENGTH は、データが含まれているかどうかにかかわらず、それぞれ指定されたリスト位置をリスト要素としてカウントします。

[\\$LISTVALID](#) 関数を使用すると、list が有効なリストかどうかを判定できます。list が有効なリストでない場合、システムは <LIST> エラーを生成します。

\$LISTBUILD によって作成された“空”のリストは、(リスト要素にはデータは含まれませんが) エンコードされたリスト要素を定義します。\$LISTLENGTH は、(データを含む要素ではなく) リスト要素をカウントするため、“空”のリストの \$LISTLENGTH 数は 1 になります。

NULL リスト (要素を含まないリスト) を表すには NULL 文字列(″)を使用します。リスト要素を含まないため、\$LISTLENGTH 数は 0 になります。

### 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。

#### ObjectScript

```
WRITE $LISTLENGTH($LISTBUILD("Red","Blue","Green"))
```

以下の例はリストの 2 つ目の要素にデータが含まれていませんが、それでも 3 を返します。

#### ObjectScript

```
WRITE $LISTLENGTH($LISTBUILD("Red",,"Green"))
```

以下の例はすべて 1 を返します。\$LISTLENGTH は、空のリスト要素とデータを含むリスト要素を区別しません。

#### ObjectScript

```
WRITE $LISTLENGTH($LB()),!
WRITE $LISTLENGTH($LB(UndefinedVar)),!
WRITE $LISTLENGTH($LB("")),!
WRITE $LISTLENGTH($LB($CHAR(0))),!
WRITE $LISTLENGTH($LB("John Smith"))
```

以下の例は 0 を返します。[\\$LISTVALID](#) は NULL 文字列を有効なリストと見なしますが、これにリスト要素は含まれません。

## ObjectScript

```
WRITE $LISTLENGTH( " " )
```

2 つのプレースホルダ・コンマは空のリスト要素が 3 つあることを表しているため、以下の例は 3 を返します。

## ObjectScript

```
WRITE $LISTLENGTH($LB( , , ))
```

## \$LISTLENGTH と連結

2 つのリストを連結すると、常に \$LISTLENGTH はリストの長さの合計と等しくなります。これは、空のリストを連結したり、NULL 文字列をリストに連結する場合でも同様です。

以下の例はすべてリスト長 3 を返します。

## ObjectScript

```
WRITE $LISTLENGTH($LB()_ $LB("a","b")),!
WRITE $LISTLENGTH($LB("a")_ $LB(UndefinedVar)_ $LB("c")),!
WRITE $LISTLENGTH($LB(" ")_ $LB()_ $LB(UndefinedVar)),!
WRITE $LISTLENGTH(" "_ $LB("a","b","c")),!
WRITE $LISTLENGTH($LB("a","b")_ " "_ $LB("c"))
```

## \$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せず、これを単一のリスト要素として扱うため、3 を返します。

## ObjectScript

```
WRITE $LISTLENGTH($LB("Apple","Pear",$LB("Walnut","Pecan")))
```

以下の例は、\$LISTLENGTH が最も外側の入れ子になっているリストのみをカウントするため、すべて 1 を返します。

## ObjectScript

```
WRITE $LISTLENGTH($LB($LB($LB()))),!
WRITE $LISTLENGTH($LB($LB($LB("Fred")))),!
WRITE $LISTLENGTH($LB($LB("Barney"_ $LB("Fred")))),!
WRITE $LISTLENGTH($LB("Fred"_ $LB("Barney"_ $LB("Wilma"))))
```

## 関連項目

- ・ [\\$LIST](#) 関数
- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTNEXT](#)
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数



- ・ [\\$LISTVALID](#) 関数

## \$LISTNEXT (ObjectScript)

リストから要素を順番に取得します。

### 構文

```
$LISTNEXT(list,ptr,value)
```

### 引数

引数	説明
list	リストに評価する任意の式。
ptr	リスト内の次の要素に対するポインタ。ptr は、0 に初期化されたローカル変数として指定する必要があります。この値は list の先頭を指し示します。InterSystems IRIS は、内部のアドレス値アルゴリズム（予測可能な整数カウンタではない）を使用して ptr をインクリメントします。したがって、ptr の設定に使用できる値は 0 のみです。ptr をグローバル変数および添え字付き変数にすることはできません。
value	リスト要素のデータ値の保持に使用されるローカル変数。\$LISTNEXT を呼び出す前に value を初期化する必要はありません。value をグローバル変数および添え字付き変数にすることはできません。

### 概要

\$LISTNEXT は、list 内の要素を順番に返します。ユーザは、\$LISTNEXT の初回呼び出しの前に、ptr を 0 に初期化します。これにより、\$LISTNEXT はリストの先頭から要素を返し始めます。\$LISTNEXT の連続する呼び出しごとに ptr が進められ、次のリストの要素値が value に返されます。\$LISTNEXT 関数は 1 を返し、リスト要素の取得に成功したことを示します。

\$LISTNEXT がリストの最後に到達したときは、0 を返し、ptr を 0 にリセットします。value は前の呼び出しから変更されません。ptr は 0 にリセットされているので、\$LISTNEXT の次の呼び出しは、list の最初から開始されます。

**注釈** ptr は list の内部構造のインデックスであるため、\$LISTNEXT の使用中にリストを変更することはできません。list を変更すると、ptr の値が無効になり、次に \$LISTNEXT を呼び出したときに <FUNCTION> エラーが発生する可能性があります。

\$LISTVALID を使用すると、list が有効なリストかどうかを判定できます。無効なリストの場合、\$LISTNEXT で <LIST> エラーが発生します。

\$LISTNEXT が省略されたリスト要素 (NULL 値を持つ要素) を検出すると、1 を返します。これは、リスト要素の取得が成功し、ptr を次の要素に進め、value を未定義の変数にリセットすることを示します。これは、list=\$LB("a","b") で \$LISTNEXT を 2 回目に呼び出した場合や、有効なリストである list=\$LB()、list=\$LB(UndefinedVar)、または list=\$LB(.) を使用した場合など、省略されたリスト要素を検出したときに発生する可能性があります。

\$LISTNEXT("",ptr,value) は 0 を返し、ポインタを進めたり value を設定することはありません。

\$LISTNEXT(\$LB(""),ptr,value) は 1 を返し、ポインタを進め、value を NULL 文字列("") に設定します。

### 例

次の例は、リスト内のすべての要素を順番に返します。省略された要素を検出した場合、[\\$SELECT](#) は既定値 “omitted” を返します。

## ObjectScript

```
SET list=$LISTBUILD("Red","Blue",,"Green")
SET ptr=0,count=0
WHILE $LISTNEXT(list,ptr,value) {
    SET count=count+1
    WRITE !,count," : ", $SELECT($DATA(value):value,1:"omitted")
}
WRITE !,"End of list: ",count," elements found"
QUIT
```

## \$LISTNEXT とパフォーマンス

InterSystems IRIS リストは、多数のデータ値を処理する最も効率的な方法です。リストを使用すると、配列や他のデータ構造を使用するよりも処理する値を多く保持できます。これにより、[最大文字列長](#)を回避できます。

\$LISTNEXT を使用してリストから多数の要素を返すと、\$LIST を使用して同じ操作を実行するよりも大幅に効率的です。

以下の例では、mylist 内の要素にすばやくアクセスします。

## ObjectScript

```
SET ptr=0
WHILE $LISTNEXT(mylist,ptr,value) {
    /* perform some operation on value */
}
```

以下の同等の例よりも大幅に速くなります。

## ObjectScript

```
FOR i=1:1:$LISTLENGTH(mylist) {
    SET value=$LIST(mylist,i)
    /* perform some operation on value */
}
```

## \$LISTNEXT と入れ子のリスト

以下の例は、\$LISTNEXT 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 つの要素を返します。

## ObjectScript

```
SET list=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
SET ptr=0,count=0
WHILE $LISTNEXT(list,ptr,value) {
    SET count=count+1
    WRITE !,value
}
WRITE !,"End of list: ",count," elements found"
QUIT
```

## 関連項目

- ・ [\\$LIST](#) 関数
- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTSAME](#) 関数

- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数

## \$LISTSAME (ObjectScript)

2 つのリストを比較し、ブーリアン値を返します。

### 構文

```
$LISTSAME(list1,list2)  
$LS(list1,list2)
```

### 引数

引数	説明
list1	リストに評価する任意の式。リストは、\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。NULL 文字列(“”)も、有効なリストと見なされます。
list2	リストに評価する任意の式。リストは、\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。NULL 文字列(“”)も、有効なリストと見なされます。

### 概要

\$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、リスト要素を、その文字列表現を使用して比較します。\$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 は最初に同一でないリスト要素のペアを検出すると 0 を返し、それ以降の要素が有効なリスト要素かどうかを判別するためのチェックは行いません。\$LISTSAME 比較が無効な要素を検出した場合、<LIST> エラーが発行されます。

### 例

以下の例は、2 つのリストが同一であるため、1 を返します。

#### ObjectScript

```
SET x = $LISTBUILD("Red","Blue","Green")  
SET y = $LISTBUILD("Red","Blue","Green")  
WRITE $LISTSAME(x,y)
```

以下の例は、2 つのリストが同一でないため、0 を返します。

#### ObjectScript

```
SET x = $LISTBUILD("Red","Blue","Yellow")  
SET y = $LISTBUILD("Red","Yellow","Blue")  
WRITE $LISTSAME(x,y)
```

### 同一のリスト

\$LISTSAME は、2 つのリストで文字列表現が同一である場合に、2 つのリストを同一と見なします。

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 は文字列と数値を比較します。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。

## ObjectScript

```
WRITE $LISTSAME($LISTBUILD("365"),$LISTBUILD(365)),!
WRITE $LISTSAME($LISTBUILD("365"),$LISTBUILD(365.0)),!
WRITE $LISTSAME($LISTBUILD("365.5"),$LISTBUILD(365.5)),!
WRITE $LISTSAME($LISTBUILD("365.0"),$LISTBUILD(365.0))
```

\$LISTSAME 比較は、その他のリスト演算子で使用されるのと同じ等価テストではありません。他の演算子では、リストの内部表現を使用してテストします。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 x = $LISTBUILD("365")
SET y = $LISTBUILD(365)
IF x=y
{ WRITE !,"Equal sign: number/numeric string identical" }
ELSE { WRITE !,"Equal sign: number/numeric string differ" }
IF 1=$LISTSAME(x,y)
{ WRITE !,"$LISTSAME: number/numeric string identical" }
ELSE { WRITE !,"$LISTSAME: number/numeric string differ" }
```

等価(=) 比較は、これらのリスト (同一ではない) の内部表現をテストします。\$LISTSAME は、両方のリストに対して文字列変換を実行し、これらを比較して、同一であると判断します。

以下の例は、数値要素のさまざまな表現を持つ 2 つのリストを示します。\$LISTSAME は、これら 2 つのリストを同一と見なします。

## ObjectScript

```
SET x = $LISTBUILD("360","361","362","363","364","365","366")
SET y = $LISTBUILD(00360.000,(19*19),+"362",363,364.0,+"365","3_"66")
WRITE !,$LISTSAME(x,y)," lists are identical"
```

## 最大数値

2\*\*63 (9223372036854775810) を超える数または -2\*\*63 (-9223372036854775808) 未満の数は、\$LISTSAME リストの比較の最大数値範囲を超えています。以下の例のように、そのような大きな数字を比較すると、\$LISTSAME は 0 を返します。

## ObjectScript

```
SET bignum=$LISTBUILD(9223372036854775810)
SET bigstr=$LISTBUILD("9223372036854775810")
WRITE $LISTSAME(bignum,bigstr),!
SET bignum=$LISTBUILD(9223372036854775811)
SET bigstr=$LISTBUILD("9223372036854775811")
WRITE $LISTSAME(bignum,bigstr)
```

## NULL 文字列と NULL リスト

NULL 文字列 (空の文字列) を唯一の要素として含むリストは、有効なリストです。NULL 文字列自体も、有効なリストと見なされます。ただし、これら 2 つ (NULL 文字列と NULL リスト) は、以下の例で示すように、同一とは見なされません。

## ObjectScript

```
WRITE !,$LISTSAME($LISTBUILD(""),$LISTBUILD("")), " null lists"
WRITE !,$LISTSAME("", ""), " null strings"
WRITE !,$LISTSAME($LISTBUILD(""),""), " null list and null string"
```

通常、文字列は有効な \$LISTSAME 引数ではないので、\$LISTSAME は、<LIST> エラーを発行します。ただし、以下の \$LISTSAME 比較は正常に終了し、0 (値は同一ではない) を返します。NULL 文字列と文字列 “abc” が比較され、同一ではないことが判明します。これらの NULL 文字列の比較により、<LIST> エラーは発生しません。

## ObjectScript

```
WRITE !,$LISTSAME("","abc")
WRITE !,$LISTSAME("abc","")
```

以下の \$LISTSAME 比較により、<LIST> エラーが発生します。これは、リスト (NULL リストであっても) は文字列と比較できないためです。

## ObjectScript

```
SET x = $LISTBUILD("")
WRITE !,$LISTSAME("abc",x)
WRITE !,$LISTSAME(x,"abc")
```

## “Empty” リストの比較

\$LISTVALID では以下のすべてを有効なリストと見なします。

## ObjectScript

```
WRITE $LISTVALID(""),!
WRITE $LISTVALID($LB()),!
WRITE $LISTVALID($LB(UndefinedVar)),!
WRITE $LISTVALID($LB("")),!
WRITE $LISTVALID($LB($CHAR(0))),!
WRITE $LISTVALID($LB(),)
```

\$LISTSAME では以下のペアのみを同一と見なします。

## ObjectScript

```
WRITE $LISTSAME($LB(),$LB(UndefinedVar)),!
WRITE $LISTSAME($LB(),$LB(UndefinedVarA,UndefinedVarB)),!
WRITE $LISTSAME($LB(),$LB()_$LB())
```

## 空の要素

\$LISTBUILD は、要素間にコンマを追加するか、1 つ以上のコンマをいずれかの要素リストの末尾に追加することにより、空の要素を作成できます。\$LISTSAME は空の要素を認識し、これらを NULL 文字列要素と同等には扱いません。

以下の \$LISTSAME の例はすべて 0 (同一ではない) を返します。

## ObjectScript

```
WRITE $LISTSAME($LISTBUILD(365,,367),$LISTBUILD(365,367)),!
WRITE $LISTSAME($LISTBUILD(365,366,),$LISTBUILD(365,366)),!
WRITE $LISTSAME($LISTBUILD(365,366,,),$LISTBUILD(365,366,)),!
WRITE $LISTSAME($LISTBUILD(365,,367),$LISTBUILD(365,"",367))
```

## \$DOUBLE リスト要素

\$LISTSAME は、0、-0、\$DOUBLE(0)、および \$DOUBLE(-0) のすべてゼロ形式を同一と見なします。

\$LISTSAME は \$DOUBLE("NAN") リスト要素と別の \$DOUBLE("NAN") リスト要素と同一と見なします。ただし、NAN (非数値) は数値演算子を使用して有意義に比較できないため、\$DOUBLE("NAN") と別の \$DOUBLE("NAN") を比較しようとする InterSystems IRIS 演算 (等しい、より小さい、より大きいなど) は以下の例のように失敗します。

## ObjectScript

```
SET x = $DOUBLE("NAN")
SET a = $LISTBUILD(1,2,x)
SET b = $LISTBUILD(1,2,x)
WRITE !,$LISTSAME(a,b) /* 1 (NAN list elements same) */
WRITE !,x=x /* 0 (NAN values not equal) */
```



## 入れ子のリストおよび連結されたリスト

\$LISTSAME では、入れ子になったリストに対するサポートはありません。コンテンツが同一であっても、リストを含む 2 つのリストを比較することはできません。

### ObjectScript

```
SET x = $LISTBUILD("365")
SET y = $LISTBUILD(365)
WRITE !,$LISTSAME(x,y)," lists identical"
WRITE !,$LISTSAME($LISTBUILD(x),$LISTBUILD(y))," nested lists not identical"
```

以下の例で、これらのリストは同一と見なされないため、\$LISTSAME 比較はどちらも 0 を返します。

### ObjectScript

```
SET x=$LISTBUILD("Apple","Pear","Walnut","Pecan")
SET y=$LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan"))
SET z=$LISTBUILD("Apple","Pear","Walnut","Pecan","")
WRITE !,$LISTSAME(x,y)," nested list"
WRITE !,$LISTSAME(x,z)," null string is list item"
```

\$LISTSAME では、連結されたリストをサポートします。以下の例は、2 つのリストが同一と見なされるため、1 を返します。

### ObjectScript

```
SET x=$LISTBUILD("Apple","Pear","Walnut","Pecan")
SET y=$LISTBUILD("Apple","Pear")_$LISTBUILD("Walnut","Pecan")
WRITE !,$LISTSAME(x,y)," concatenated list"
```

## 関連項目

- ・ [\\$LIST](#) 関数
- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数
- ・ [\\$DOUBLE](#) 関数

## \$LISTTOSTRING (ObjectScript)

リストから文字列を作成します。

### 構文

```
$LISTTOSTRING(list,delimiter,flag)
$LISTS(list,delimiter,flag)
```

### 引数

引数	説明
list	InterSystems IRIS リスト。\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。
delimiter	オプション — 部分文字列の区切りに使用される区切り文字。delimiter は、引用符付きの文字列として指定します。delimiter を指定しない場合、既定はコンマ (,) 文字です。
flag	オプション — 3 ビットのバイナリ・ビット・フラグ。利用可能な値は、0 (000)、1 (001)、2 (010)、3 (011)、4 (100)、5 (101)、6 (110)、および 7 (111) です。既定値は 0 です。

### 概要

\$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。

ZWRITE コマンドを使用すると、エンコードされていない形式でリストを表示できます。

### 引数

#### list

1 つ以上の要素を持つ InterSystems IRIS リスト。リストは、\$LISTBUILD を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。

list 引数内の式が有効なリストとして評価されない場合は、<LIST> エラーが発生します。

#### ObjectScript

```
SET x=$CHAR(0,0,0,1,16,27,134,240)
SET a=$LISTTOSTRING(x,",") // generates a <LIST> error
```

#### 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)。(引用符で囲まれた) 数値または文字列リテラル、変数名、文字列に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される特定の文字が設定されます。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。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するか (""""), または \$CHAR(34) を使用します。

## flag

3 ビットのバイナリ・ビット・フラグ:

- 1 ビットは、list 内の省略された要素の処理方法を指定します。0 は <NULL VALUE> エラーを発行します。1 は省略された各要素に対して空の文字列を挿入します。

以下の例では、list に、省略された要素があります。このリスト要素を処理するため、flag=1 オプションが指定されています。

### ObjectScript

```
SET colorlist=$LISTBUILD("Red",,"Blue")
WRITE $LISTTOSTRING(colorlist,,1)
```

flag の 1 ビットが省略されているか、0 に設定されている場合 (flag=0、2、4、または 6)、\$LISTTOSTRING は <NULL VALUE> エラーを生成します。

flag=1 の場合、空の文字列値のある要素は、省略された要素と区別がつきません。

\$LISTBUILD("Red",,"Blue") および \$LISTBUILD("Red",,"Blue") は同じ \$LISTTOSTRING 値を返します。この flag=1 の動作は、“InterSystems SQL リファレンス”の説明のとおり、InterSystems SQL の [\\$LISTTOSTRING](#) の実装と互換性があります。

- 2 ビットは、特定の文字が含まれる文字列の引用符を指定します。これらの文字列は、delimiter 文字 (既定ではコンマ)、二重引用符文字 (")、改行文字 (LF = \$CHAR(10))、およびキャリッジ・リターン文字 (CR = \$CHAR(13)) です。このビットは flag=2 または 3 によって設定します (flag=6 または 7 の場合、このオプションは設定されますが、4 ビット・オプションによって上書きされます)。
- 4 ビットは、すべての文字列の引用符を指定します。このビットは flag=4、5、6、または 7 によって設定します

## 例

以下の例は、4 つの要素を持つリストを作成し、要素がコロンの (:) 文字で区切られた文字列に変換します。

### ObjectScript

```
SET namelist=$LISTBUILD("Deborah","Noah","Martha","Bowie")
WRITE $LISTTOSTRING(namelist,":")
```

このコードは、結果として "Deborah:Noah:Martha:Bowie" を返します。

以下の例は、4 つの要素を持つリストを作成し、要素が \*sp\* 文字列で区切られた文字列に変換します。

### ObjectScript

```
SET namelist=$LISTBUILD("Deborah","Noah","Martha","Bowie")
WRITE $LISTTOSTRING(namelist,"*sp*")
```

このコードは、結果として "Deborah\*sp\*Noah\*sp\*Martha\*sp\*Bowie" を返します。

以下の例では、省略された要素 1 つと空の文字列値のある要素 1 つを持つリストを作成します。\$LISTTOSTRING は、このリストをコロンの (:) 文字で区切られた要素を持つ 1 つの文字列に変換します。省略された要素があるため、<NULL

VALUE> エラーを避けるには flag=1 が必要です。ただし、flag=1 の場合、省略された要素と空の文字列値は区別されません。

### ObjectScript

```
SET namelist=$LISTBUILD("Deborah",,"","Bowie")
WRITE $LISTTOSTRING(namelist,":",1)
```

このコードは、結果として "Deborah:::Bowie" を返します。

以下の例では、コンマを含む要素のリストを作成します。既定では、\$LISTTOSTRING は delimiter としてコンマを使用します。最初の例では、コンマを含む要素は、コンマで区切られた 2 つの要素と区別できません。2 番目の例では、flag=3 が指定されているため、この要素に引用符が付いています。3 番目の例では、flag=7 が指定されているため、すべての要素に引用符が付いています。

### ObjectScript

```
SET pairlist=$LISTBUILD("A,B","C^D","E|F")
WRITE $LISTTOSTRING(pairlist,,1)
// returns A,B,C^D,E|F
WRITE $LISTTOSTRING(pairlist,,3)
// returns "A,B","C^D","E|F"
WRITE $LISTTOSTRING(pairlist,,7)
// returns "A,B","C^D","E|F"
```

\$LISTVALID では以下のすべてを有効なリストと見なします。flag=1 で、\$LISTTOSTRING はすべてについて NULL 文字列 (") を返します。

### ObjectScript

```
WRITE "1",$LISTTOSTRING("",,1),!
WRITE "2",$LISTTOSTRING($LB(),,1),!
WRITE "3",$LISTTOSTRING($LB(UndefinedVar),,1),!
WRITE "4",$LISTTOSTRING($LB(""),,1)
```

flag=0 で、\$LISTTOSTRING は以下に対してのみ NULL 文字列 (") を返します。

### ObjectScript

```
WRITE "1",$LISTTOSTRING("",,0),!
WRITE "4",$LISTTOSTRING($LB(""),,0)
```

その他は <NULL VALUE> エラーを生成します。

## 関連項目

- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LIST](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#) 関数
- ・ [\\$LISTSAME](#) 関数

- ・ [\\$LISTUPDATE](#) 関数
- ・ [\\$LISTVALID](#) 関数

## \$LISTUPDATE (ObjectScript)

オプションで、指定された 1 つのリスト要素または一連の要素を置き換えることにより、リストを更新します。

### 構文

```
$LISTUPDATE(list,position,bool:value...)
$LU(list,position,bool:value...)
```

### 引数

引数	説明
list	リストに評価する任意の式。リストは、\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。NULL 文字列(“)も、有効なリストと見なされません。
position	更新する list 内の位置を指定する、1 からカウントされる正の整数。position が list 内の要素の数より大きい場合、\$LISTUPDATE は、必要な場合はパディングして要素を追加します。
bool:	オプション - 指定された list 要素を更新するかどうかを指定するブーリアン変数。省略した場合、既定で bool は 1 になり、この要素を更新します。
value	指定された position で list を更新するために使用する値。value 引数のコンマ区切りのリスト、または任意の組み合わせの bool:value のペア引数を指定できます。

### 説明

\$LISTUPDATE は、位置を指定して 1 つ以上のリスト要素を置換または追加して更新したリストのコピーを返します。position は、要素の更新を開始するリスト内の位置を指定します。要素は、この位置から開始して順次更新されます。position は、リスト内の要素数よりも大きくすることができます。そのようにした場合、指定された位置に新規要素を挿入するために、リストに NULL 要素が(必要に応じて)追加されます。位置は正の整数でなければなりません。\$LISTUPDATE はリストの先頭に新規要素を挿入することはできません。position=0 を設定しても、処理は実行されません。\$LISTUPDATE は、リストの終わりを指定することもできません。ただし、リストの先頭からの位置数で指定することはできます。

\$LISTUPDATE は、複数の bool:value のペアを指定できます。複数の bool:value のペアは、コンマで区切られます。これらの bool:value のペアは、連続する要素を、position エlement から開始して更新します。bool=1 の場合、InterSystems IRIS はその要素を value で更新します。bool=0 の場合、InterSystems IRIS はその要素を更新せず、次の要素に進みます。

一連の bool:value のペア内で更新されない連続する要素は、指定する必要はありません。それらはプレースホルダとしてのコンマで表すことができます。bool: 引数は、各 value に対してオプションです。省略する場合、既定の 1 になります。bool を省略する場合は、コロン(:) 区切り文字も省略します。したがって、以下の方法で bool:value ペアを指定することは許可されます。

- 更新する要素:1:"newval"、または "newval" (bool は既定で 1 になります)。
- 更新しない要素:0:"newval"、またはプレースホルダとしてのコンマ。

\$LISTUPDATE は、一般に既存のリストの更新バージョンを返すために使用します。\$LISTUPDATE を使用して、list="" を指定することでリストを作成できます。

さらに、[SET \\$LIST](#) を使用して、既存のリスト内の 1 つ以上の要素を更新することもできます。

### 例

以下の例では、位置 2 でリスト要素を、指定された値に置き換えます。

### ObjectScript

```
SET caps=1
SET mylist = $LISTBUILD("Red","White","Blue")
SET newlist = $LISTUPDATE(mylist,2,caps:"WHITE")
ZWRITE newlist
```

以下の例では、位置 2 でリスト要素を、指定された値に置き換えません。

### ObjectScript

```
SET caps=0
SET mylist = $LISTBUILD("Red","White","Blue")
SET newlist = $LISTUPDATE(mylist,2,caps:"WHITE")
ZWRITE newlist
```

以下の例では、位置 2 でリスト要素を NULL に置き換えます。

### ObjectScript

```
SET caps=1
SET mylist = $LISTBUILD("Red","White","Blue")
SET newlist = $LISTUPDATE(mylist,2,caps:"")
ZWRITE newlist
```

以下の例では、位置 7 で、指定された値をリストに追加し、NULL 要素を位置 5 と 6 にパディングします。

### ObjectScript

```
SET bool=1
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green")
SET newlist = $LISTUPDATE(mylist,7,bool:"Purple")
ZWRITE newlist
```

以下の例では、位置 7 で、指定された値をリストに追加しません。NULL 要素がパディングされていない未変更のリストが返されます。

### ObjectScript

```
SET bool=0
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green")
SET newlist = $LISTUPDATE(mylist,7,bool:"Purple")
ZWRITE newlist
```

以下の 3 つの例は、機能的にはすべて同じです。それぞれは要素を位置 2 と 4 で置き換え、新規要素を位置 7 で追加します。これは要素 3 と 5 は置き換えません。要素 6 は NULL 要素として作成されます。

### ObjectScript

```
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green","Blue")
SET newlist = $LISTUPDATE(mylist,2,1:"ORANGE",0:"YELLOW",
                          1:"GREEN",0:"BLUE",
                          0:"INDIGO",1:"VIOLET")
ZWRITE newlist
```

ここでは bool 引数は、0 の場合にのみ指定されます。

### ObjectScript

```
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green","Blue")
SET newlist = $LISTUPDATE(mylist,2,"ORANGE",0:"YELLOW",
                          "GREEN",0:"BLUE",
                          0:"INDIGO","VIOLET")
ZWRITE newlist
```

ここではプレースホルダとしてのコンマは、更新されない要素をスキップするために使用されます。



## ObjectScript

```
SET mylist = $LISTBUILD("Red","Orange","Yellow","Green","Blue")
SET newlist = $LISTUPDATE(mylist,2,"ORANGE",,"GREEN",,"VIOLET")
ZWRITE newlist
```

## 関連項目

- ・ [\\$LIST](#) 関数
- ・ [\\$LISTBUILD](#) 関数
- ・ [\\$LISTDATA](#) 関数
- ・ [\\$LISTFIND](#) 関数
- ・ [\\$LISTFROMSTRING](#) 関数
- ・ [\\$LISTGET](#) 関数
- ・ [\\$LISTLENGTH](#) 関数
- ・ [\\$LISTNEXT](#) 関数
- ・ [\\$LISTSAME](#) 関数
- ・ [\\$LISTTOSTRING](#) 関数
- ・ [\\$LISTVALID](#) 関数

# \$LISTVALID (ObjectScript)

式がリストかどうかを判別します。

## 構文

```
$LISTVALID(exp)
$LV(exp)
```

## 引数

引数	説明
exp	任意の有効な式。

## 説明

\$LISTVALID は exp がリストかどうかを判別してブーリアン値を返します。exp がリストの場合、\$LISTVALID は 1 を返し、exp がリストでない場合、\$LISTVALID は 0 を返します。

リストは、\$LISTBUILD または \$LISTFROMSTRING を使用して作成されるか、\$LIST を使用して別のリストから抽出されます。空の文字列(“)を唯一の要素として含むリストは、有効なリストです。空の文字列(“)自体も、有効なリストと見なされます ([\\$CHAR の特定の出力不能文字の組み合わせ](#) (\$CHAR(1)、\$CHAR(2,1)、\$CHAR(3,1,asciicode) など)により、有効な空のリストまたは 1 要素のリストを返すこともできます)。

## 例

以下の例はすべて、有効なリストであることを示す 1 を返します。

### ObjectScript

```
SET w = $LISTBUILD("Red", "Blue", "Green")
SET x = $LISTBUILD("Red")
SET y = $LISTBUILD(365)
SET z = $LISTBUILD("")
WRITE !, $LISTVALID(w)
WRITE !, $LISTVALID(x)
WRITE !, $LISTVALID(y)
WRITE !, $LISTVALID(z)
```

以下の例はすべて 0 を返します。数値および文字列 (NULL 文字列を除く) は有効なリストではありません。

### ObjectScript

```
SET x = "Red"
SET y = 44
WRITE !, $LISTVALID(x)
WRITE !, $LISTVALID(y)
```

以下の例はすべて 1 を返します。連結された値、入れ子になった値、および省略された値のリストはすべて有効なリストです。

### ObjectScript

```
SET w=$LISTBUILD("Apple", "Pear")
SET x=$LISTBUILD("Walnut", "Pecan")
SET y=$LISTBUILD("Apple", "Pear", $LISTBUILD("Walnut", "Pecan"))
SET z=$LISTBUILD("Apple", "Pear", , "Pecan")
WRITE !, $LISTVALID(w_x) ; concatenated
WRITE !, $LISTVALID(y) ; nested
WRITE !, $LISTVALID(z) ; omitted element
```

以下の例はすべて 1 を返します。\$LISTVALID では以下の “空” のリストすべてを有効なリストと見なします。\$LISTBUILD は、未定義の変数をリスト要素として取ることができ、これによってエラーが生成されることはありません。

### ObjectScript

```
WRITE $LISTVALID(""),!  
WRITE $LISTVALID($LB()),!  
WRITE $LISTVALID($LB(UndefinedVar)),!  
WRITE $LISTVALID($LB("")),!  
WRITE $LISTVALID($LB($CHAR(0))),!  
WRITE $LISTVALID($LB(,))
```

## 関連項目

- ・ [\\$LIST 関数](#)
- ・ [\\$LISTBUILD 関数](#)
- ・ [\\$LISTDATA 関数](#)
- ・ [\\$LISTFIND 関数](#)
- ・ [\\$LISTFROMSTRING 関数](#)
- ・ [\\$LISTGET 関数](#)
- ・ [\\$LISTLENGTH 関数](#)
- ・ [\\$LISTNEXT 関数](#)
- ・ [\\$LISTSAME 関数](#)
- ・ [\\$LISTTOSTRING 関数](#)
- ・ [\\$LISTUPDATE 関数](#)

## \$LOCATE (ObjectScript)

文字列で、正規表現の最初の一致を検索します。

### 構文

```
$LOCATE(string, regexp, start, end, val)
```

### 引数

引数	説明
string	一致対象の文字列。
regexp	string に一致させる正規表現。正規表現は、1 つ以上のメタ文字で構成し、リテラル文字を含めることもできます。
start	オプション — regexp が一致を開始する string 内の開始位置を指定する整数。  start を省略する場合、マッチングが string の先頭から開始されます。start を省略し、end と val のいずれかまたは両方を指定する場合、プレースホルダのコンマを指定する必要があります。
end	オプション — 一致に成功した場合 \$LOCATE によってこの変数に整数が割り当てられます。この整数は、一致した文字列の後の次の文字位置です。InterSystems IRIS は、 <a href="#">参照によって</a> end を渡します。この引数はローカル変数にする必要があります。配列、グローバル変数、オブジェクト・プロパティへの参照は使用できません。
val	オプション — 一致に成功した場合 \$LOCATE によってこの変数に文字列値が割り当てられます。この文字列は一致した文字列で構成されます。InterSystems IRIS は、 <a href="#">参照によって</a> val を渡します。この引数はローカル変数にする必要があります。配列、グローバル変数、オブジェクト・プロパティへの参照は使用できません。

### 説明

\$LOCATE は、正規表現を指定された文字列の連続する部分文字列に対して一致させます。string 内で、最初の regexp の一致の開始位置を指定する整数です。1 から始まる値で位置をカウントします。regexp が、string の一部と一致しない場合、0 を返します。

オプションで、一致部分文字列を変数に割り当てることができます。

あるオプション引数を省略し、それよりも後の引数を指定する場合、プレースホルダのコンマを指定する必要があります。

ObjectScript は、\$LOCATE および \$MATCH 関数で構成された正規表現をサポートします。

- ・ \$LOCATE は、正規表現と string の部分一致を検出し、最初の一致の位置（およびオプションで値）を返します。
- ・ \$MATCH は、正規表現と string の完全一致を検出し、一致が発生したかどうかを示すブーリアンを返します。

%Regex.Matcher クラスの Locate() メソッドは、\$LOCATE と類似する機能を提供します。%Regex.Matcher クラス・メソッドは、正規表現を使用するための有効な追加機能を提供します。

その他の ObjectScript のマッチング処理では、[InterSystems IRIS パターン・マッチング](#)演算子が使用されます。

## 引数

### string

文字列として評価される式。この式には、変数名や数値、文字列リテラル、その他の有効な ObjectScript 式を指定できます。string には、制御文字を含めることができます。

string が空の文字列で、regex が空の文字列と一致できない場合は、\$LOCATE が 0 を返し、end および val は設定されません。

string が空の文字列で、regex が空の文字列と一致できる場合は、\$LOCATE が 1 を返し、end が 1、および val が空の文字列に設定されます。

### regex

目的の部分文字列を検索するために、string に一致させるために使用する正規表現。正規表現は、メタ文字とリテラルのいくつかの組み合わせで構成された文字列を評価する式です。メタ文字は、文字の種類と一致パターンを指定します。リテラルは、1 つ以上の単一文字、文字の範囲、または部分文字列との一致を指定します。広範囲にわたる正規表現の構文をサポートしています。詳細は、“[正規表現](#)”を参照してください。

### start

regex が一致を開始する string 内の開始位置を指定する整数。1 または 0 で、string の先頭からの開始を指定します。start の値が、文字列 +1 の長さと同じ場合、常に 0 が返されます。start の値が、文字列 +1 の長さよりも大きい場合、ERROR #8351 の <REGULAR EXPRESSION> エラーが返されます。

start の位置に関係なく、\$LOCATE は、最初の一致の位置を文字列の先頭からのカウントで返します。

### end

検索処理で一致が検出された場合 \$LOCATE によって整数が割り当てられる出力変数。割り当てられる値は、一致した部分文字列の後の最初の文字の位置を文字列の先頭からのカウントで表したものです。一致が文字列の最後で発生した場合は、この文字の位置が文字列全体の長さに 1 を加えた値になる可能性があります。一致が検出されなかった場合は、end の値は変更されません。end が未定義の場合、この値は未定義のままになります。

end 変数をオブジェクトのプロパティへの参照として使用することはできません。

start および end に同じ変数を使用することで、\$LOCATE を繰り返し呼び出して、文字列内のすべての一致を検出できます。この例を以下に示します。この例は、アルファベットの母音の位置を検出します。

### ObjectScript

```
SET alphabet="abcdefghijklmnopqrstuvwxyz"
SET pos=1
SET val=""
FOR i=1:1:5 {
    WRITE $LOCATE(alphabet,"[aeiou]",pos,pos,val)
    WRITE " is the position of the ",i,"th vowel: ",val,! }

```

### val

検索処理で一致が検出された場合 \$LOCATE によって文字列が割り当てられる出力変数。この文字列値は一致した文字列です。一致が検出されなかった場合は、val の値は変更されません。val が未定義の場合、この値は未定義のままになります。

val 変数をオブジェクトのプロパティへの参照として使用することはできません。

## 例

以下の例は、regex のリテラル “de” が文字列の 4 番目の文字と一致するため 4 を返します。

## ObjectScript

```
WRITE $LOCATE("abcdef","de")
```

以下の例は、regexp が 3 文字の小文字の文字列を指定し、ここで検出されるのは“fga”の部分文字列で、開始位置が 8 であるため 8 が返されます。

## ObjectScript

```
WRITE $LOCATE("ABC-de-fgabc123ABC","[[:lower:]]{3}")
```

以下の例は、指定された regexp のスペース文字 (§s) および非スペース文字 (§S) の形式が、文字列の 5 番目の文字から始まっていることが検出されたため 5 を返します。この例では、start 引数を省略し、一致した部分文字列の後の文字である end 変数を 11 に設定します。

## ObjectScript

```
WRITE $LOCATE("AAAAA# $ 456789","\S\S\S\S\S\S",,end)
```

以下の例は、regexp が 3 文字の文字列を指定し、start 引数が 6 文字目以降から始まる必要があることを指定しています。

## ObjectScript

```
SET end="",val=""
WRITE $LOCATE("abc-def-ghi-jkl","[[:alpha:]]{3}",6,end,val),!
WRITE "the position after the matched string is: ",end,!
WRITE "the matched value is: ",val
```

end 引数が 12 に設定され、val 引数が“ghi”に設定されます。

以下の例は、regexp が結果文字列と一致する前に、数値がキャノニック形式に解決されることを示しています。end 引数は 5 に設定され、4 文字の文字列“1.23”の末尾を超えた 1 文字を示します。

## ObjectScript

```
WRITE $LOCATE(123E-2,"\.d*",1,end,val),!
WRITE "end is: ",end,!
WRITE "value is: ",val,!
```

以下の例は、start 引数を string+1 の長さより大きい値に設定します。これにより、以下に示すようにエラーが発生します。

## ObjectScript

```
TRY {
  SET str="abcdef"
  SET strlen=$LENGTH(str)
  WRITE "start=string length, match=", $LOCATE(str,"p{L}",strlen),!
  WRITE "start=string length+1, match=", $LOCATE(str,"p{L}",strlen+1),!
  WRITE "start=string length+2, match=", $LOCATE(str,"p{L}",strlen+2),!
}
CATCH exp {
  WRITE !,"CATCH block exception handler:",!
  IF l=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception",!
    WRITE "Name: ", $ZCVT(exp.Name,"O","HTML"),!
    WRITE "Location: ", exp.Location,!
    WRITE "Code: ", exp.Code,!
    WRITE "%Regex.Matcher status:"
    SET err=##class(%Regex.Matcher).LastStatus()
    DO $SYSTEM.Status.DisplayError(err) }
  ELSE {WRITE "Unexpected exception type",! }
  RETURN
}
```

## 関連項目

- ・ [\\$CHAR 関数](#)
- ・ [\\$MATCH 関数](#)
- ・ [正規表現](#)
- ・ [パターン・マッチング \(?\)](#)



# \$MATCH (ObjectScript)

正規表現と文字列を一致させます。

## 構文

```
$MATCH(string, regexp)
```

## 引数

引数	説明
string	一致対象の文字列。
regexp	string に一致させる正規表現。正規表現は、1 つ以上のメタ文字で構成し、リテラル文字を含めることもできます。

## 説明

\$MATCH は、string と regexp が一致する場合に 1、string と regexp が一致しない場合に 0 を返すブーリアン関数です。既定では、マッチングで大文字と小文字が区別されます。

ObjectScript は、\$LOCATE および \$MATCH 関数で構成された[正規表現](#)をサポートします。

- ・ \$MATCH は、正規表現と string の完全一致を検出し、一致が発生したかどうかを示すブーリアンを返します。
- ・ \$LOCATE は、正規表現と string の部分一致を検出し、最初の一致の位置（およびオプションで値）を返します。

%Regex.Matcher クラスの Match() メソッドは、同じ機能を提供します。%Regex.Matcher クラスは、正規表現を使用するための追加の機能を提供します。

その他の ObjectScript のマッチング処理では、[InterSystems IRIS パターン・マッチング](#)演算子が使用されます。

## 引数

### string

[文字列](#)として評価される式。この式には、変数名や数値、文字列リテラル、その他の有効な ObjectScript 式を指定できます。string には、制御文字を含めることができます。

### regexp

string に一致させるために使用する正規表現。正規表現は、メタ文字とリテラルのいくつかの組み合わせで構成された[文字列](#)を評価する式です。メタ文字は、文字の種類と一致パターンを指定します。リテラルは、1 つ以上の単一文字、文字の範囲、または部分文字列との一致を指定します。広範囲にわたる正規表現の構文をサポートしています。詳細は、“[正規表現](#)”を参照してください。

## 例

以下の例は、最初の文字が大文字 (`¥p{LU}`) で、1 文字以上の追加の文字 (+ 修飾子) が続き、この 2 番目の文字とそれ以降のすべての文字が単語の文字 (英字、数字、およびアンダースコア文字) (`¥w`) であることを指定する正規表現と、文字列と一致させます。

## ObjectScript

```

SET strng(1)="Assembly_17"
SET strng(2)="Part5"
SET strng(3)="SheetMetalScrew"
SET n=1
WHILE $DATA(strng(n)) {
  IF $MATCH(strng(n),"\p{LU}\w+")
  { WRITE strng(n)," : successful match",! }
  ELSE { WRITE strng(n)," : invalid string",! }
  SET n=n+1 }

```

以下の例は、16 進数の正規表現 (hex 41) が “A” の文字と一致するため 1 を返します。

## ObjectScript

```
WRITE $MATCH("A","\x41")
```

以下の例は、指定された文字列が、正規表現で指定されたスペース文字 (§s) および非スペース文字 (§S) の形式と一致するため 1 を返します。

## ObjectScript

```
WRITE $MATCH("A# $ 4","\S\S\S\S\S\S\S")
```

以下の例は、指定された日付が、正規表現で指定された桁数とリテラルの形式と一致するため 1 を返します。

## ObjectScript

```

SET today=$ZDATE($HOROLOG)
WRITE $MATCH(today,"^\d\d/\d\d/\d\d\d\d$")

```

この形式では、日付と月のそれぞれが 2 桁で指定される必要があり、このため先頭が 0 の場合 10 未満の値が必要になります。

以下の例は、文字列の各文字が正規表現の対応する文字の範囲内にあるため 1 を返します。

## ObjectScript

```
WRITE $MATCH("HAL","[G-I][A-C][K-Z]")
```

以下の例は、不正な regexp 引数を指定しています。これにより、以下に示すようにエラーが発生します。

## ObjectScript

```

TRY {
  SET str="abcdef"
  WRITE "match=", $MATCH(str, "\p{ }"), !
}
CATCH exp {
  WRITE !, "CATCH block exception handler:", !
  IF l=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception", !
    WRITE "Name: ", $ZCVT(exp.Name, "O", "HTML"), !
    WRITE "Location: ", exp.Location, !
    WRITE "Code: ", exp.Code, !!
    WRITE "%Regex.Matcher status:"
    SET err=##class(%Regex.Matcher).LastStatus()
    DO $SYSTEM.Status.DisplayError(err) }
  ELSE {WRITE "Unexpected exception type", ! }
  RETURN
}

```

## 関連項目

- ・ [\\$CHAR](#) 関数
- ・ [\\$LOCATE](#) 関数

- ・ [\\$ZSTRIP](#) 関数
- ・ [正規表現](#)
- ・ [パターン・マッチング \(?\)](#)

## \$METHOD (ObjectScript)

インスタンス・メソッドへの呼び出しをサポートします。

### 構文

```
$METHOD( instance, methodname, arg1, arg2, arg3, ... )
```

### 引数

引数	説明
instance	オブジェクト参照として評価される表現。この表現の値は、クラスのメモリに存在するインスタンスの値である必要があります。
methodname	文字列として評価される式。文字列の値は、最初の引数で指定したクラスのインスタンスにある既存のメソッドの名前と一致する必要があります。
arg1, arg2, arg3, ...	指定したメソッドへの引数を置き換える一連の式。式の値には任意の型のものを使用できます。実装するユーザは、指定した表現の型とメソッドで想定されている型が一致していること、およびメソッドが想定する範囲内の値が使用されていることを確認する必要があります (指定されたメソッドが引数の使用を想定していない場合、この関数呼び出しでは classname と methodname 以外の引数を使用する必要はありません。メソッドが引数を必要とする場合、どの引数を指定する必要があるかを規定するルールとして、ターゲット・メソッドのルールが適用されます)。

### 説明

\$METHOD は、指定されたクラスの指定されたインスタンスで、指定されたインスタンス・メソッドを実行します。

この関数は、ObjectScript プログラムが、あるクラスの既存のインスタンス内の任意のメソッドを呼び出すことを許可します。最初の引数はオブジェクトへの参照でなければならないため、実行時に算出されます。メソッド名は、実行時に算出することもできれば、文字列リテラルとして指定しておくこともできます。メソッドが引数を取る場合は、メソッド名の後の引数リストで引数を指定します。最大 255 の引数値がメソッドに渡されます。メソッドが引数を必要とする場合、どの引数を指定する必要があるかを規定するルールとして、ターゲット・メソッドのルールが適用されます。インスタンス・メソッドではなくクラス・メソッドを呼び出すには、\$CLASSMETHOD 関数を使用します。

関数またはプロシージャとして \$METHOD を呼び出すと、ターゲット・メソッドの呼び出しが決定されます。\$METHOD は、DO コマンドを使用して呼び出して、戻り値を破棄できます。すべての DO コマンド引数と同様に、\$METHOD は DO によって呼び出されたときに後置条件パラメータを取ることができます。

クラス・インスタンスの 1 つのメソッド内で、\$METHOD を使用してそのインスタンスの別のメソッドを参照する場合は、instance を省略できます。ただし、普通は Instance の後に続くコンマは、この場合も必要です。

存在しないメソッドまたはクラス・メソッドとして宣言されているメソッドの呼び出しを実行すると、<METHOD DOES NOT EXIST> というエラーが出力されます。

### 例

以下は、\$METHOD を関数として使用する例です。

## ObjectScript

```
SET ListOfStuff = ##class(%Library.ListOfDataTypes).%New()  
FOR i = "First", "Second", "Third", "Fourth"  
{  
    DO ListOfStuff.Insert((i _ "-Element"))  
}  
SET methodname = "Count"  
SET elements = $METHOD(ListOfStuff,methodname)  
WRITE "Elements: ",elements,!  
SET i = $RANDOM(elements) + 1  
WRITE "Element #", i , " = " , $METHOD(ListOfStuff,"GetAt", i), !
```

## 関連項目

- ・ [\\$CLASSMETHOD](#) 関数
- ・ [\\$CLASSNAME](#) 関数
- ・ [\\$PARAMETER](#) 関数
- ・ [\\$PROPERTY](#) 関数
- ・ [\\$THIS](#) 特殊変数

## \$NAME (ObjectScript)

変数の名前値、もしくは添え字参照部分を返します。

### 構文

```
$NAME(variable, integer)
$NA(variable, integer)
```

### 引数

引数	説明
variable	名前値が返される変数。添え字付きあるいは添え字なしのローカル変数やグローバル変数を指定できます。定義済みの変数である必要はありません。ただし、定義済みのプライベート変数ではないことが必要です。variable が添え字付きのグローバルの場合は、 <a href="#">ネイキッド・グローバル参照</a> を指定できます。
integer	オプション - 添え字参照のどの部分 (レベル) を返すか指定する数値。正の整数、変数名、式を指定できます。この場合、variable は添え字付き参照である必要があります。

### 概要

\$NAME は variable として提供されている変数名参照の、フォーマットされた形式を返します。この変数が定義済みであるか、またはデータ値を持つかなどのチェックは行いません。\$NAME が返す値は、使用する引数によって決まります。

- ・ \$NAME(variable) は、キャノニック形式で (完全な拡張参照として) 指定した変数名の値を返します。
- ・ \$NAME(variable, integer) は、添え字参照の一部を返します。特に、integer は関数によって返される添え字数を制御します。

この関数を実行しても、ネイキッド・インジケータに影響はありません。

### 引数

#### variable

variable は、ローカル変数、プロセス・プライベート・グローバル変数、またはグローバル変数です。これは添え字なしでも添え字付きでもかまいません。variable は、グローバル変数である場合は[拡張グローバル参照](#)を使用できます。\$NAME は、指定された拡張グローバル参照を返しますが、その際に、指定されたネームスペースが存在するかどうかや、ユーザがそのネームスペースに対するアクセス特権を持っているかどうかを確認しません。この関数は、ネームスペース名を大文字にしません。variable が[ネイキッド・グローバル参照](#)の場合、\$NAME は、完全なグローバル参照を返します。variable がプライベート変数である場合は、コンパイル・エラーが発生します。

これは[多次元オブジェクト・プロパティ](#)であってもかまいませんが、非多次元オブジェクト・プロパティであってはけません。非多次元オブジェクト・プロパティで \$NAME を使用しようとすると、<OBJECT DISPATCH> エラーが発生します。

\$NAME は、SET を使用して変更できるものも含めて、[特殊変数](#)を返すことはできません。特殊変数を返そうとすると、<SYNTAX> エラーが発生します。

#### integer

variable が添え字付き参照のとき、integer 引数を使用されます。integer の値が 0 のとき、\$NAME は変数名のみを返します。integer 値が variable の添え字数よりも小さい場合、\$NAME は integer 値で示される添え字数のみを返します。integer が variable の添え字数よりも大きい場合、\$NAME は完全添え字参照を返します。

variable が添え字なしの場合、integer の値は無視され、\$NAME は変数名を返します。integer が NULL 文字列 ("")、または数値以外の文字列の場合、\$NAME は添え字なしの変数名を返します。

integer の値は、標準整数解析を受け取ります。例えば、先頭のゼロとプラス符号は無視されます。小数桁は切り捨てられ、無視されます。負の integer 値は、<FUNCTION> エラーとなります。

## 例

この例では、integer 引数が返すレベルを指定します。integer で指定した添え字数が添え字レベルの数 (この場合は 3) と一致するか、またはそれを超えるとき、\$NAME は 1 引数形式が指定されたかのように、すべての定義済みレベルを返します。integer レベルにゼロ (0)、NULL 文字列 ("")、または数値以外の文字列 ("A" など) を指定した場合、\$NAME は配列名を返します (この場合は "client")。

### ObjectScript

```
SET ^client(4)="Vermont"
SET ^client(4,1)="Brattleboro"
SET ^client(4,1,1)="Yankee Ingenuity"
SET ^client(4,1,2)="Vermonster Systems"
WRITE !,$NAME(^client(4,1,1),1) ; returns 1 level
WRITE !,$NAME(^client(4,1,1),2) ; returns 2 levels
WRITE !,$NAME(^client(4,1,1),3) ; returns 3 levels
WRITE !,$NAME(^client(4,1,1),4) ; returns all (3) levels
WRITE !,$NAME(^client(4,1,1),0) ; returns array name
WRITE !,$NAME(^client(4,1,1),"") ; returns array name
WRITE !,$NAME(^client(4,1,1)) ; returns all (3) levels
```

以下の例では、\$NAME は、ループの中でネイキッド参照と共に使用され、現在の (ユーザが提供した) 配列レベルのすべての要素の値を出力します。

### ObjectScript

```
READ !,"Array element: ",ary
SET x=@ary ; dummy operation to set current array and level
SET y=$ORDER(^("")) ; null string to find beginning of level
FOR i=0:0 {
    WRITE !,@$NAME(^(y))
    SET y=$ORDER(^(y))
    QUIT:y=""
}
```

最初の SET コマンドは、ユーザ提供の配列とその後のネイキッド参照の基準としてのレベルを確立するために、仮の割り当てを実行しています。\$ORDER 関数は、ネイキッド参照と共に使用され、現在のレベルで最初の添え字数 (負あるいは正) を返します。

FOR ループの WRITE コマンドは、ネイキッド参照と引数の間接指定を \$NAME と共に使用して、現在の要素の値を出力します。SET コマンドは、ネイキッド・グローバル参照を \$ORDER と共に使用して、データを含む次の要素の添え字を返します。最後に、後置条件付きの QUIT は、\$ORDER の戻り値を確認し、現在のレベルの最後を検索して、ループ処理を終了します。

名前や添え字の間接指定に、返した \$NAME 文字列値を使用したり、引数としてルーチンやユーザ定義関数に渡したりします。詳細は、[“間接\(@\)”](#) のリファレンス・ページを参照してください。指定したノードの下位ノードをリストするルーチン `DESCEND` を考えてみます。



## ObjectScript

```

DESCEND(ROOT) ;List descendant nodes
NEW REF
SET REF=ROOT
IF ROOT'[ " ( " {
    FOR {
        SET REF=$QUERY(@REF)
        QUIT:REF=" "
        WRITE REF,! }
}
ELSE {
    SET $EXTRACT(ROOT,$LENGTH(ROOT))=","
    FOR {
        SET REF=$QUERY(@REF)
        QUIT:REF'[ROOT
        WRITE REF,! }
}

```

以下の例は、\$NAME を使用して、前例で定義された ^DESCEND ルーチンに引数を渡す方法を示しています。

## ObjectScript

```

FOR var1="ONE","TWO","THREE" {
    DO ^DESCEND($NAME(^X(var1))) }

```

^X("ONE",2,3)

## \$NAME の使用

\$DATA 関数と \$ORDER 関数で使用する配列要素の名前値を返すには、通常 \$NAME を使用します。

配列要素に対する参照に式が含まれている場合、\$NAME はキャノニック形式の名前を返す前にその式を評価します。例えば以下ようになります。

## ObjectScript

```

SET x=1+2
SET y=$NAME(^client(4,1,x))
WRITE y

```

\$NAME は、変数 x を評価して、値 ^client(4,1,3) を返します。

## ネイキッド・グローバル参照

\$NAME はネイキッド・グローバル参照も受け入れ、そのキャノニック形式（つまり完全な（非ネイキッド）参照）で名前値を返します。ネイキッド参照は配列名なしで指定され、最近実行されたグローバル参照を示します。以下の例では、最初の SET コマンドはグローバル参照を作成して、2 つ目の SET コマンドはネイキッド・グローバル参照と共に \$NAME 関数を使用しています。

## ObjectScript

```

SET ^client(5,1,2)="Savings/27564/3270.00"
SET y=$NAME(^{3})
WRITE y

```

この場合、\$NAME は ^client(5,1,3) を返します。現在のレベルで、元の値である添え字値 (2) を、提供された添え字値 (3) に置き換えます。

詳細は、“[ネイキッド・グローバル参照](#)” を参照してください。

## 拡張グローバル参照

%SYSTEM.Process クラスの RefInKind() メソッドを使用して、\$NAME が拡張グローバル参照形式で名前値を返すかどうかを、プロセスごとに制御できます。システム全体の既定の動作は、Config.Miscellaneous クラスの RefInKind プロパティで設定できます。

拡張参照モードが有効な場合、以下の例は、`^MyRoutine` (2 つ目の例で示されています) だけでなく、定義済みのネームスペースと名前 `^[ "PAYROLL" ]MyRoutine` (1 つ目の例で示されています) を返します。

#### ObjectScript

```
DO ##class(%SYSTEM.Process).RefInKind(0)
WRITE $NAME(^[ "PAYROLL" ]MyRoutine)
```

#### ObjectScript

```
DO ##class(%SYSTEM.Process).RefInKind(1)
WRITE $NAME(^[ "PAYROLL" ]MyRoutine)
```

拡張グローバル参照構文の詳細は、“[拡張グローバル参照](#)”を参照してください。

## 関連項目

- ・ [\\$DATA](#) 関数
- ・ [\\$ORDER](#) 関数
- ・ [\\$GET](#) 関数

## \$NCONVERT (ObjectScript)

数値を、8 ビット文字の文字列でエンコードされた 2 進数値に変換します。

### 構文

```
$NCONVERT(n,format,endian)
$NC(n,format,endian)
```

### 引数

引数	説明
<i>n</i>	任意の数字。値、変数、式として指定できます。選択した <i>format</i> により、有効な値に対してさらに制限が課せられます。
<i>format</i>	S1、S2、S4、S8、U1、U2、U4、F4、または F8 のいずれかの形式コード。引用符で囲んだ文字列で指定します。
<i>endian</i>	オプション — ブーリアン値。0 = リトル・エンディアン、1 = ビッグ・エンディアン。既定値は 0 です。

### 説明

\$NCONVERT は、指定された *format* を使用して、数値 *n* を 8 ビットのエンコードされた文字列に変換します。これらの文字の値は、\$CHAR(0) から \$CHAR(255) の範囲内になります。

以下の *format* コードがサポートされます。

コード	概要
S1	1 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -128 ~ 127 です。
S2	2 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -32768 ~ 32767 です。
S4	4 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -2147483648 ~ 2147483647 です。
S8	8 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -9223372036854775808 ~ 9223372036854775807 です。
U1	1 つの 8 ビット・バイトの文字列にエンコードされた符号なし整数。最大値は 255 です。
U2	2 つの 8 ビット・バイトの文字列にエンコードされた符号なし整数。最大値は 65535 です。
U4	4 つの 8 ビット・バイトの文字列にエンコードされた符号なし整数。最大値は 4294967295 です。
F4	4 つの 8 ビット・バイトの文字列にエンコードされた IEEE 浮動小数点数。
F8	8 つの 8 ビット・バイトの文字列にエンコードされた IEEE 浮動小数点数。

*format* 制限の範囲を超える値を指定すると、<VALUE OUT OF RANGE> エラーが発生します。符号なしの *format* に負の数を指定すると、<VALUE OUT OF RANGE> エラーが発生します。*n* が数値以外の値である (数値以外の文字が含まれている) 場合、InterSystems IRIS は [文字列から数値への変換](#) を実行します。数値以外の文字で始まる文字列は、0 に変換されます。

InterSystems IRIS は、F4 および F8 以外のすべての形式で小数を整数値に丸めます。

IsBigEndian() クラス・メソッドを使用して、オペレーティング・システム・プラットフォームでどのビット順序を使用するかを決定できます (1 = ビッグ・エンディアン・ビット順、0 = リトル・エンディアン・ビット順)。

### ObjectScript

```
WRITE $SYSTEM.Version.IsBigEndian()
```

\$SCONVERT は \$NCONVERT の逆の操作を行います。

## 例

以下の例では、一連の符号なしの数値を 2 バイトにエンコードされた値に変換します。

### ObjectScript

```
FOR x=250:1:260 {
    ZZDUMP $NCONVERT(x,"U2") }
QUIT
```

以下の例では、同じ操作をビッグ・エンディアン順に実行します。

### ObjectScript

```
FOR x=250:1:260 {
    ZZDUMP $NCONVERT(x,"U2",1) }
QUIT
```

## 関連項目

- ・ [\\$SCONVERT](#) 関数

## \$NORMALIZE (ObjectScript)

数値を検証して返し、指定された小数桁数に丸めます。

### 構文

```
$NORMALIZE (num, scale)
```

### 引数

引数	説明
num	検証される数値。数値、文字列値、変数名、あるいは有効な ObjectScript 式にできます。
scale	返り値として num が丸められる有効な小数桁数。この数は、num の実際の小数桁数より大きい場合も、小さい場合もあります。許可される値は、0 (整数に丸める)、-1 (整数に切り捨てる)、および正の整数 (指定した小数桁数に丸める) です。最大 scale 値はありません。ただし、機能上の最大値が数値精度を超えることはありません。標準の InterSystems IRIS® データ・プラットフォームの小数では、機能上の scale の最大値は 18 (から整数桁数 -1 を引く) です。

### 概要

\$NORMALIZE 関数は num を検証し、num を正規化した形式を返します。また、scale 引数を使用して、小数桁数の丸め (または切り捨て) を行います。scale 引数を使用して実数を指定した小数桁数に丸めたり、実数を整数に丸めたり、実数を整数に切り捨てたりすることができます。

丸めの後、\$NORMALIZE は、返り値から末尾のゼロを削除します。このため、返される小数桁数は、以下の例に示すように scale で指定された数より小さくなる場合があります。

#### ObjectScript

```
WRITE $NORMALIZE($ZPI,11),!  
WRITE $NORMALIZE($ZPI,12),!    /* trailing zero removed */  
WRITE $NORMALIZE($ZPI,13),!  
WRITE $NORMALIZE($ZPI,14)
```

### 引数

#### num

検証される数は整数、実数、または科学的記数法の数 (文字 “E” または “e” 付き) のいずれかです。文字列、式、または数に解決される変数のいずれかになることもあります。符号付きや符号なし、先頭、または末尾に 0 が付いていることもあります。\$NORMALIZE は、文字ごとに検証を行います。以下の場合には、検証を中止し、文字列の検証された部分を返します。

- num が 0-9、+ または - 符号、小数点(.)、および “E” または “e” 以外の任意の文字を含む場合。科学的記数法では、大文字の “E” は標準の指数演算子です。小文字の “e” は、%SYSTEM.Process.ScientificNotation() メソッドを使用して構成できる指数演算子です。
- num が複数の小数点、文字 “E” または “e” を含む場合。
- num で + 符号または - 符号が数値の後にある場合、これらは末尾の符号と見なされ、それ以降の数値は解析されません。
- “E”、もしくは “e” の後に数値の整数が続いていない場合。

scale 引数値を使用すると、返り値は、num 値を丸めた、または切り捨てた値となります。num 変数の実際の値は、\$NORMALIZE 処理によって変更されません。

## scale

必須の scale 引数は、丸める小数桁数を指定するために使用されます。指定された値に応じて、scale は、小数桁数に影響を与えない、指定された小数桁数に丸める、整数に丸める、または整数に切り捨てる場合があります。

負数ではない scale 値を使用して、num を指定された小数桁数に丸めることができます。丸めるとき、値 5、またはそれ以上の値は切り上げられます。数の丸めを避けるには、scale を num で可能な小数桁数よりも大きい値に設定します。scale 値に 0 を指定すると、num は整数値 ( $3.9 = 4$ ) に丸められます。scale 値に -1 を指定すると、num は整数値 ( $3.9 = 3$ ) に切り捨てられます。数値以外の scale 値や NULL 文字列は、scale 値 0 に相当します。

scale には整数値を指定します。scale 値の小数桁数は無視されます。num で指定された小数桁数よりも大きい scale 値を指定することができます。-1 の scale 値を指定することもできます。それ以外の負の scale 値はすべて、〈FUNCTION〉エラーが返されます。

## 例

以下の例では、\$NORMALIZE の各呼び出しで、指定された丸め（または整数切捨て）を使用して正規化された num が返されます。

### ObjectScript

```
WRITE !,$NORMALIZE(0,0)           ; All integers OK
WRITE !,$NORMALIZE("",0)          ; Null string is parsed as 0
WRITE !,$NORMALIZE(4.567,2)       ; Real numbers OK
WRITE !,$NORMALIZE("4.567",2)     ; Numeric strings OK
WRITE !,$NORMALIZE("-+--0.109",2) ; Multiple leading signs OK
WRITE !,$NORMALIZE(+004.500,1)     ; Leading/trailing 0's OK
WRITE !,$NORMALIZE(4E2,-1)        ; Scientific notation OK
```

以下の例で、\$NORMALIZE の各呼び出しは、num の数値サブセットを返します。

### ObjectScript

```
WRITE !,$NORMALIZE("4,567",0)
; NumericGroupSeparators (commas) are not recognized
; here validation halts at the comma, and 4 is returned.
WRITE !,$NORMALIZE("4A",0)
; Invalid (nonnumeric) character halts validation
; here 4 is returned.
```

以下の例は、scale 引数を使用して、返り値を丸める（または切り捨てる）方法を示します。

### ObjectScript

```
WRITE !,$NORMALIZE(4.55,2)
; When scale is equal to the fractional digits of num,
; all digits of num are returned without rounding.
WRITE !,$NORMALIZE(3.85,1)
; num is rounded to 1 fractional digit,
; (with values of 5 or greater rounded up)
; here 3.9 is returned.
WRITE !,$NORMALIZE(4.01,17)
; scale can be larger than number of fractional digits,
; and no rounding is performed; here 4.01 is returned.
WRITE !,$NORMALIZE(3.85,0)
; When scale=0, num is rounded to an integer value.
; here 4 is returned.
WRITE !,$NORMALIZE(3.85,-1)
; When scale=-1, num is truncated to an integer value.
; here 3 is returned.
```

## \$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 値と共に \$NORMALIZE 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、返り値に 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。

## ObjectScript

```
SET x=1234.1234
SET y=$DOUBLE(1234.1234)
WRITE "Decimal: ", $NORMALIZE(x,2), !
WRITE "Double: ", $NORMALIZE(y,2), !
WRITE "Dec/Dub: ", $NORMALIZE($DECIMAL(y),2)
```

\$DOUBLE 値を 10 進形式用に正規化する場合、上記の例で示すとおり、\$DOUBLE 値を 10 進表現に変換してから結果を正規化する必要があります。

\$NORMALIZE は、\$DOUBLE("INF") 値および \$DOUBLE("NAN") 値を処理し、INF および NAN を返します。

## 無視される DecimalSeparator 値

\$NORMALIZE は、数値の処理を意図しています。num 文字列は、数値として解釈されます。InterSystems IRIS は、\$NORMALIZE に数値を渡す前に、その数値を**キャノニック形式**に変換します。このキャノニック形式の数への変換では、現在のロケールの DecimalSeparator プロパティ値を使用しません。

例えば、num に文字列 "00123.4500" を指定すると、現在の DecimalSeparator 値に関係なく、\$NORMALIZE はこれをキャノニック形式の数 123.45 として扱います。文字列 "00123,4500" を指定した場合、現在の DecimalSeparator 値に関係なく、\$NORMALIZE はこれをキャノニック形式の数 123 (最初の非数値文字で切り捨て) として扱います。

文字列を入力とする関数を使用したい場合、**\$INNUMBER** を用います。文字列の結果を生成する関数を使用したい場合、**\$FNUMBER** を用います。

## \$NORMALIZE と \$NUMBER の比較

\$NORMALIZE 関数と \$NUMBER 関数は両方とも数を検証し、指定された数を検証して返します。

これら 2 つの関数では、検証の条件は異なります。必要に応じて、それぞれの関数を使い分けてください。

- これら 2 つの関数は、符号付き、または符号なしの整数 (-0 を含む)、科学的記数法の数 ("E" または "e" 付き)、および実数を解析します。ただし、\$NUMBER は小数部分を持つ数 (負の 10 進数の指数を持つ科学的記数法を含む) を拒否するように ("I" 形式を使用して) 設定できます。2 つの関数は両方とも、数 (123.45) と数値文字列 ("123.45") を解析します。
- どちらの関数も、先頭と末尾のゼロを削除します。10 進文字は、ゼロでない値が続かない限り削除されます。
- NumericGroupSeparator を含む数値文字列。\$NUMBER は format 引数 (または現在のロケールに対する既定値) を基にして、NumericGroupSeparator 文字 (アメリカ形式はコンマ (,), ヨーロッパ形式はピリオド (.) もしくはアポストロフィ ()) と 10 進文字 (アメリカ形式はピリオド (.), ヨーロッパ形式はコンマ (,)) を解析します。また、NumericGroupSeparator 文字の任意の数を許可し、削除します。例えばアメリカ形式は、数 123456.99 として "123,,4,56.99" を検証します。\$NORMALIZE は、NumericGroupSeparator 文字を認識しません。この関数は、数値以外の文字を検出するまで、文字ごとに検証します。例えば "123,456.99" を、数 123 として検証します。
- 先頭の複数の符号 (+ と -) は、数に対する 2 つの関数両方によって解釈されます。\$NORMALIZE だけは、引用符付き数値文字列に入った先頭の複数の符号を受け入れます。
- 両方の関数は、数の末尾の符号を受け入れません。引用符付きの数値文字列では、\$NUMBER は 1 つの (そして唯一の) 末尾の符号を解析します。\$NORMALIZE は、複数の末尾符号を解析します。
- 括弧。\$NUMBER は、引用符付き文字列内の符号なしの数を囲む括弧を、負の数を示すものとして解析します。\$NORMALIZE は、数値以外の文字として括弧を処理します。
- 複数の 10 進文字を含む、数値文字列。\$NORMALIZE は 2 番目の 10 進文字を検出するまで、文字ごとに検証を行います。例えばアメリカ形式は、"123.4.56" を数 123.4 として検証します。\$NUMBER は、1 つ以上の 10 進文字を含む任意の文字列を無効な数として受け入れません。

他の数値以外の文字を含む数値文字列。\$NORMALIZE は、アルファベット文字を検出するまで、文字ごとに検証を行います。これは、"123A456" を数 123 として検証します。\$NUMBER は文字列全体を検証し、"123A456" を無効な数として受け入れません。



- ・ NULL 文字列。\$NORMALIZE は、NULL 文字列をゼロ (0) として解析します。\$NUMBER は、NULL 文字列を受け入れません。

\$NUMBER 関数は、オプションで min/max の範囲チェックを行います。範囲チェックは、\$ISVALIDNUM 関数を使用し行うこともできます。

\$NORMALIZE、\$NUMBER、および \$ISVALIDNUM はすべて、指定された小数桁数に数を丸めます。\$NORMALIZE は、小数桁数の丸め、および実数の丸めまたは切り捨てを行って、整数を返します。例えば、\$NORMALIZE は 488.65 を 488.7 または 489 に丸めるかまたは、488 に切り捨てることができます。\$NUMBER は、実数または整数を丸めることができます。例えば、\$NUMBER は 488.65 を 488.7、489、490、または 500 に丸めることができます。

## 関連項目

- ・ [\\$DOUBLE 関数](#)
- ・ [\\$FNUMBER 関数](#)
- ・ [\\$INUMBER 関数](#)
- ・ [\\$ISVALIDNUM 関数](#)
- ・ [\\$NUMBER 関数](#)
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$NOW (ObjectScript)

現在のプロセスのローカルの日付と時刻を、秒の小数部を含めて返します。

### 構文

```
$NOW ( tzmins )
```

### 引数

引数	説明
tzmins	<p>オプション - グリニッジ子午線を基準とした目的のタイム・ゾーン・オフセットを指定する正または負の整数値 (分)。0 は、グリニッジ子午線に相当します。正の整数は、グリニッジの西側のタイム・ゾーンに相当し、負の整数はグリニッジの東側のタイム・ゾーンに相当します。例えば、300 は グリニッジの 5 時間 (300 分) 西側の米国東部標準時に相当します。許可される値の範囲は -1440 ~ 1440 です。この範囲を超えると、&lt;ILLEGAL VALUE&gt; エラーが発生します。</p> <p>tzmins を省略すると、\$NOW 関数は \$TIMEZONE 特殊関数の値に基づいてローカルの日付と時刻を返します。\$NOW 関数がサポートする \$TIMEZONE 値の範囲は -1440 ~ 1440 です。値がこの範囲を超えると、&lt;ILLEGAL VALUE&gt; エラーが発生します。</p>

### 説明

\$NOW は以下の値を返すことができます。

- ・ 現在のプロセスのローカルの日付と時刻 (秒の小数部を含む)
- ・ 現在のプロセスの、指定されたタイム・ゾーンのローカルの日付と時刻 (秒の小数部を含む)

\$NOW 関数は、コンマで区切られた 2 つの数値から成る文字列を返します。最初の数値は、現在のローカル日付を表す整数です。2 番目は、現在のローカル時刻を表す小数です。これらの値はカウンタで、ユーザが読み取れる日付と時刻ではありません。

\$NOW は、InterSystems IRIS ストレージ (\$HOROLOG) 形式の日付と時刻 (秒の小数部を含む) を返します。\$NOW は現在のローカルの日付と時刻を dddddd,sssss.mmmf という形式で返します。

最初の整数 (dddddd) は 1840 年 12 月 31 日から起算した日数です。1 は 1841 年 1 月 1 日を表します。この日付の整数の最大値は 2980013 です。これは 9999 年 12 月 31 日に相当します。

2 番目の数値 (sssss.mmmf) は、その日の午前 0 時 00 分からの秒数 (および秒の小数部) を表します。InterSystems IRIS は、sssss フィールドを 0 から 86399 秒までインクリメントします。午前 0 時に 86399 に達すると、sssss フィールドは 0 にリセットされ、日付フィールドが 1 つインクリメントされます。午前 0 時から 1 秒以内は、秒数が 0 です。mmmf (例えば、0.123456) と表されることに注意してください。この数値は、値の文字列ソート順に影響を与える ObjectScript の[キャノニック形式](#) (例えば、.123456) ではありません。先頭にプラス記号 (+) を追加して、数値をキャノニック形式に強制的に変換してから、ソート操作を実行できます。

精度の小数桁数 mmmf は 6 から 9 です。末尾のゼロは削除されます。

\$NOW 関数は引数値を指定しても、指定しなくても呼び出せます。括弧は必須です。

引数値を指定しないと、\$NOW は現在のプロセスの、現在のローカルの日付と時刻を返します。これは、\$ZTIMEZONE 特殊変数に設定された値からローカル・タイム・ゾーンを決定します。\$ZTIMEZONE を設定すると \$NOW の時刻部分 が変更され、またこの変更によって \$NOW の日付部分も変更されることがあります。

**注意**            ローカル時刻値 \$NOW は、ローカルな時刻に対応しない場合があります。\$NOW は \$ZTIMEZONE 値を使用してローカル時刻を決定します。\$ZTIMEZONE は 1 年を通して連続的で、サマータイム (DST) などのローカル時刻調整には対応していません。

UTC 時刻からのオフセットは、グリニッジ子午線からのタイム・ゾーンのカウンタを使用して計算されます。これは、ローカル時刻とローカル・グリニッジ時刻の比較ではありません。グリニッジ標準時 (GMT) とは紛らわしい用語であり、グリニッジのローカル時刻は、冬の間は UTC と同じで、夏の間は、UTC と 1 時間違います。これは、英国夏時間と呼ばれるローカル時刻の変更が適用されるためです。\$NOW では、ローカル時刻の変更がすべて無視されます。

また、\$NOW はシステム時計と時刻値を再同期するため、\$NOW とその他の InterSystems IRIS 時間関数および特殊変数には、若干の差異が生じる場合があります。この差異は 0.05 秒以内ですが、この差異の範囲内で比較した際に、誤解を与える結果が生じる場合があります。例えば、WRITE \$NOW(),!, \$HOROLOG では以下のような結果が生じます。

```
61438.38794.002085
61438.38793
```

この異常は、再同期での 0.05 秒の差異と、\$HOROLOG での秒の小数部の切り捨てが重なって生じます。

引数のある \$NOW は、tzmins で指定されたタイム・ゾーンに対応する時刻と日付を返します。\$ZTIMEZONE の値は無視されます。

## 日付と時刻の分割

\$NOW の日付部分または時刻部分のみを取得するには、区切り文字としてコンマを指定して、[\\$PIECE](#) 関数を使用します。

### ObjectScript

```
SET dateval=$PIECE($NOW()," ",1)
SET timeval=$PIECE($NOW()," ",2)
WRITE !,"Date and time: ", $NOW()
WRITE !,"Date only: ",dateval
WRITE !,"Time only: ",timeval
```

## 例

以下の例では、現在のローカルの日付と時刻を返す 2 つの方法を示します。

### ObjectScript

```
WRITE $ZDATETIME($NOW(),1,1,3)," $NOW() date & time",!
WRITE $ZDATETIME($HOROLOG,1,1,3)," $HOROLOG date & time"
```

\$HOROLOG によって、[サマータイム](#)などのローカル時刻調整が行われます。\$NOW ではローカル時刻調整に合わせた調整を行いません。

以下の例は \$ZDATE を使用して、\$NOW の日付フィールドを日付形式に変換します。

### ObjectScript

```
WRITE $ZDATE($PIECE($NOW()," ",1))
```

04/29/2009 の形式で値を返します。

次の例では、\$NOW の時刻部分が 12 時間 (a.m. または p.m.) 表示の時間:分:秒.mmmf 形式の時刻に変換されます。

## ObjectScript

```
CLOCKTIME
NEW
SET Time=$PIECE($NOW(),",",2)
SET Sec=Time#60
SET Totmin=Time\60
SET Min=Totmin#60
SET Milhour=Totmin\60
IF Milhour=12 { SET Hour=12,Meridian=" pm" }
ELSEIF Milhour>12 { SET Hour=Milhour-12,Meridian=" pm" }
ELSE { SET Hour=Milhour,Meridian=" am" }
WRITE !,Hour,":",Min,":",Sec,Meridian
QUIT
```

## 時間関数の比較

現在の日付と時刻は、さまざまな方法で返すことができますが、その違いを以下に示します。

- ・ \$NOW は、現在のプロセスのローカルな日付と時刻を返します。\$NOW は、日付と時刻を InterSystems IRIS ストレージ形式で返します。秒の小数部は含まれます。
  - － \$NOW は、\$ZTIMEZONE 特殊変数の値からローカル・タイム・ゾーンを決定します。ローカル時刻は、[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せん。したがって、ローカルな時刻に対応しない場合があります。
  - － \$NOW(tzmins) は、指定された tzmins タイム・ゾーン引数に対応する時刻と日付を返します。\$ZTIMEZONE の値は無視されます。
- ・ \$HOROLOGY には、ローカル調整された日付と時刻が InterSystems IRIS ストレージ形式で格納されます。ローカル・タイム・ゾーンは、\$ZTIMEZONE 特殊変数の現在の値で決まり、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。これは、整数秒のみを返し、小数部分は切り捨てられます。
- ・ \$ZTIMESTAMP には、秒の小数部を含む UTC (協定世界時) の日付と時刻が InterSystems IRIS ストレージ形式で格納されます。

## 日付の設定

\$NOW および \$ZTIMESTAMP によって返される値は、%SYSTEM.Process クラスの FixedDate() メソッドを使用して設定することはできません。

\$HOROLOGY に含まれる値は、%SYSTEM.Process クラスの FixedDate() メソッドを使用して、現在のプロセスのユーザー指定の日付に設定することができます。

## 関連項目

- ・ [\\$ZDATE](#) 関数
- ・ [\\$ZDATEH](#) 関数
- ・ [\\$ZDATETIME](#) 関数
- ・ [\\$ZDATETIMEH](#) 関数
- ・ [\\$ZTIME](#) 関数
- ・ [\\$ZTIMEH](#) 関数
- ・ [\\$HOROLOGY](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数
- ・ [\\$ZTIMEZONE](#) 特殊変数

# \$NUMBER (ObjectScript)

数値を検証し、返します。オプションで数の丸めや範囲チェックを行います。

## 構文

```
$NUMBER(num,format,min,max)
$NUM(num,format,min,max)
```

## 引数

引数	説明
num	検証されてから InterSystems IRIS キヤノニック形式に変換される数値。数値、文字列値、変数名、あるいは有効な ObjectScript 式にできます。
format	オプション — num に適用する処理オプションを指定します。これらの処理法は、主に小数点を含む数字を認識し、処理する方法を指示します。
min	オプション — 許容範囲内の最小値。
max	オプション — 許容範囲内の最大値。

## 概要

\$NUMBER 関数は、指定した format を使用して、num 数値の変換と検証を実行します。さまざまな句読点形式の数字を受け取り、InterSystems IRIS キヤノニック形式の数字を返します。format を使用して数が整数かどうかを確認できます。min または max が指定された場合、数字をその範囲の値に収める必要があります。

\$NUMBER は、アメリカ形式の数、ヨーロッパ形式の数、ロシア/チェコ形式の数に使用できます。

[\\$DOUBLE](#) の INF 値、-INF 値、または NAN 値で \$NUMBER を使用すると、常に空の文字列を返します。

## 引数

### format

利用可能な format コードは、以下のとおりです。これらの format コードは、どのような順序でも指定できます。数値以外の format は、引用符付きの文字列として指定されなければなりません。以下の format コードはいずれも省略可能です。format が無効の場合、\$NUMBER は <SYNTAX> エラーを生成します。

- 10 進文字は、“.” または “,” のいずれかを使用しますが、“.” はアメリカ式、“,” はヨーロッパ式の小数点検証用の変換方式を示します。これらのうちいずれかの文字を指定するか、あるいは 10 進文字を使用しません。10 進文字を省略する場合、その数は現在のロケールの DecimalSeparator を取得します。下記の [“ヨーロッパ式およびアメリカ式の小数点区切り文字”](#) を参照してください。
- 丸め係数は、小数点何桁まで丸めを行うかを示す整数です。整数の前には、+ 符号または - 符号を付けます。丸め係数が正数（あるいは符号無し）の場合、number は指定された小数桁で丸めます。丸め係数が 0 の場合、number は整数に丸めます。丸め係数が負の整数の場合、number は小数点区切り文字の左にある数字に対し指定された桁数で丸めます。例えば、丸め係数が -2 の場合、234.45 は 200 に丸められます。数字の “5” は常に繰り上がります。したがって、丸め係数が 1 の場合、123.45 は 123.5 になります。丸めると、末尾のゼロは削除されます。したがって、4.0043 を小数点以下 2 桁に丸めると、4.00 ではなく 4 が返されます。IEEE 浮動小数点数の丸めについては、下記の [“\\$DOUBLE の数値”](#) を参照してください。
- 整数表示文字：文字 “I”（大文字または小文字）は、number が整数値に解決される必要があることを指定します。例えば、-07.00 は整数に解決されますが、-07.01 は整数に解決されません。数が整数に解決されない場合、

\$NUMBER は NULL 文字列を返します。以下の例では、最初の 3 つの \$NUMBER 関数のみが整数を返します。それ以外の 3 つの関数は、NULL 文字列を返します。

### ObjectScript

```
WRITE $NUMBER(-07.00,"I")," non-canonical integer numeric",!
WRITE $NUMBER(+ "-07.00","I")," string forced as integer numeric",!
WRITE $NUMBER("-7","I")," canonical integer string numeric",!
WRITE $NUMBER("-07.00","I")," non-canonical integer string numeric",!
WRITE $NUMBER(-07.01,"I")," fractional numeric",!
WRITE $NUMBER("-07.01","I")," fractional string numeric",!
```

## Min と Max

許容最小値や許容最大値、またはその両方を指定できます。あるいは、一切指定なしでもかまいません。指定する場合、num 値 (丸め実行後) は、min 値以上、max 値以下でなければなりません。min または max 値として NULL 文字列を指定した場合、これは 0 に相当します。値がこの基準に合わない場合、\$NUMBER は NULL 文字列を返します。

したがって、以下の例で 1 つ目は、num (4.0) が max (4) と同等なので、有効です。2 つ目の例も、num (4.003) が max (4) と形式範囲内 (小数点以下 2 桁) で同等なので、有効です。しかし 3 つ目の例は、\$NUMBER が num を形式範囲内で max より大きい値である (4.01) まで丸めを行ってしまうため、有効ではありません。その結果、NULL 文字列を返します。

### ObjectScript

```
WRITE !,$NUMBER(4.0,2,0,4)
WRITE !,$NUMBER(4.003,2,0,4)
WRITE !,$NUMBER(4.006,2,0,4)
```

コンマをプレースホルダとして保持し、引数を省略できます。以下の例で、1 行目は max 値を設定しますが、format または min の値は設定されません。2 行目では、format 値を設定しませんが、ゼロと等しい NULL 文字列の min 値を設定します。したがって、1 行目は -7 を返し、2 行目は min の基準を満たさないで、NULL 文字列を返します。

### ObjectScript

```
SET max=10
WRITE !,$NUMBER(-7,,max)
WRITE !,$NUMBER(-7,,"",max)
```

末尾のコンマは指定できません。以下の結果は <SYNTAX> エラーになります。

```
WRITE $NUMBER(mynum,,min,)
```

## 演算の順序

\$NUMBER は以下の一連の変換と検証を実行します。数字の妥当性検証に失敗した場合、\$NUMBER は NULL 文字列 (") を返します。その数値がすべての検証手順に合格した場合、\$NUMBER は、InterSystems IRIS キャノニック形式の数値に変換した結果を返します。

1. \$NUMBER は、10 進文字形式を使用して、どの文字がグループ区切り文字であるかを判別し、(数におけるそれらの位置に関係なく) すべてのグループ区切り文字を削除します。使用する規則は以下のとおりです。format で指定した小数点文字がピリオド (.) の場合、グループ区切り文字はコンマ (,) または空白スペースです。format で指定した小数点文字がコンマ (,) の場合、グループ区切り文字はピリオド (.) または空白スペースです。format に 10 進文字が指定されていない場合、グループ・セパレータは、現在のロケールの NumericGroupSeparator プロパティの値となります。(ロシア語 (rusw)、ウクライナ語 (ukrw)、チェコ語 (csyw) のロケールでは、数値グループ区切り文字として空白スペースが使用されます)。
2. \$NUMBER は、数字が規則に合っているかどうか妥当性の検証を行います。適格な number は以下のいずれかを含みます。
  - ・ 数値。



- ・ 上述の任意の小数点文字 (1 つまたは指定なしで、複数指定は不可)。
  - ・ 任意のプラス (+) あるいはマイナス (-) 符号 (先頭あるいは末尾に置き、複数指定は不可)。
  - ・ 負の値 (借方) を示す数を囲む任意の括弧。括弧内の数字には符号を付けません。
  - ・ 整数の後に続く "E" (大文字もしくは小文字) によって示される、任意の 10 進数の指数。"E" を指定する場合は、指数が存在する必要があります。指数の前に符号が付く場合もあります。
3. 整数表示文字が format 内に存在する場合、\$NUMBER は整数をチェックします。整数には小数点文字を記述できません。数値文字列 ("123.45") と数 (123.45) は、個別に解析されます。数値文字列は、小数点文字の後に数字の桁がない場合や、拡張科学的記数法や数の丸めにより小数桁が削除されている場合でも、整数テストには失敗します。数は、これらの検証テストに成功します。数が整数表示テストに失敗した場合、\$NUMBER は NULL 文字列 ("") を返します。
  4. \$NUMBER は、その数を InterSystems IRIS のキャノニック形式の数値に変換します。科学的記数法を展開し、括弧を負の符号文字と置き換え、先頭あるいは末尾にあるゼロを取り除き、後にゼロでない数字桁が続かない小数点文字を削除します。科学的記数法では、大文字の "E" は標準の指数演算子です。小文字の "e" は、%SYSTEM.Process.ScientificNotation() メソッドを使用して構成できる指数演算子です。
  5. \$NUMBER は (存在する場合は) 丸め係数を使用して、指定された桁に数字を丸めます。その後、後に数字が何も続かない場合は、先頭あるいは末尾にあるゼロと小数点文字を削除します。
  6. \$NUMBER は、最小値が指定されている場合、その値に対し数字の妥当性を検証します。
  7. \$NUMBER は、最大値が指定されている場合、その値に対し数字の妥当性を検証します。
  8. \$NUMBER は結果の数字を返します。

## ヨーロッパ式およびアメリカ式の小数点区切り文字

\$NUMBER は、キャノニック形式の数値を返し、数字グループ・セパレータをすべて削除して小数点区切り文字を 1 つ追加します。format の値 “,” または “.” を使用して、num で使用する小数点区切り文字を識別することができます。また、小数点区切り文字を指定することで、数字グループ・セパレータを暗黙的に指定することもできます。

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")
```

ユーザのロケールの NumericGroupSeparator 文字と NumericGroupSize 数を決定するには、以下のメソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSize")
```

以下の例では、コンマを小数点区切り文字に指定します。



## ObjectScript

```

SET num="123,456"
WRITE !,$NUMBER(num,"")
// converts to the fractional number "123.456"
// (comma is identified as decimal separator)
SET num="123,45,6"
WRITE !,$NUMBER(num,"")
// returns the null string
// (invalid number, too many decimal separators)
SET num="123.456"
WRITE !,$NUMBER(num,"")
// converts to the integer "123456"
// removing group separator
// (if comma is decimal, then period is group separator)
SET num="123.4.56"
WRITE !,$NUMBER(num,"")
// converts to the integer "123456"
// removing group separators
// (number and placement of group separators ignored)

```

## \$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 値が丸め係数と共に \$NUMBER に入力されると、返り値に丸め係数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。これは、10 進小数の結果がバイナリで表現できないためです。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。

## ObjectScript

```

SET x=1234.5678
SET y=$DOUBLE(1234.5678)
WRITE "Decimal: ",x," rounded ",$NUMBER(x,2),!
WRITE "Double: ",y," rounded ",$NUMBER(y,2)

```

\$DOUBLE 数値を使用する際、IEEE 浮動小数点数と標準の InterSystems IRIS 小数の精度が異なることに注意してください。\$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。これらの精度は 53 バイナリ・ビットで、これは 15.95 小数桁数の精度に相当します(バイナリ表現は正確に小数と一致するものではありません)。ほとんどの小数はこのバイナリ表現では正確に表すことができないため、IEEE 浮動小数点数は対応する InterSystems IRIS 標準の浮動小数点数とは若干異なる場合があります。サポートされるすべての InterSystems IRIS システム・プラットフォームにおいて、標準の InterSystems IRIS 小数は、小数桁数 18 桁の精度を持ちます。IEEE 浮動小数点数が小数として表示されるとき、バイナリ・ビットが 18 桁を大幅に超える小数桁数の小数に変換されることがよくあります。これは、IEEE 浮動小数点数が InterSystems IRIS 標準の小数より精度が高いことを意味するわけではありません。

\$NUMBER を使用して \$DOUBLE 値を丸め、特定の小数桁数を返す場合は、\$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。以下に例を示します。

```

USER>set mydouble=$double(33/100)

USER>write mydouble
.330000000000000001554
USER>w $number(mydouble,2)
.330000000000000001554
USER>set mydecimal=$decimal(mydouble)

USER>write $number(mydecimal,2)
.33

```

\$NUMBER は空の文字列として \$DOUBLE("INF") または \$DOUBLE("NAN") を返します。

## 関連項目

- [\\$DOUBLE 関数](#)
- [\\$FNUMBER 関数](#)
- [\\$INUMBER 関数](#)

- ・ [\\$ISVALIDNUM](#) 関数
- ・ [\\$NORMALIZE](#) 関数
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ORDER (ObjectScript)

次のローカル変数、またはローカル変数やグローバル変数の添え字を返します。

### 構文

```
$ORDER(variable,direction,target)
$o(variable,direction,target)
```

### 引数

引数	説明
variable	添え字付きのローカル変数、プロセス・プライベート・グローバル変数、またはグローバル変数です。配列の場合、添え字が必要です。配列名だけを指定することはできません。間接演算を使用して添え字なしのローカル変数を指定できます（以下の例を参照してください）。単純なオブジェクト・プロパティ参照を variable として指定することはできません。構文 obj.property を使用すると、 <a href="#">多次元プロパティ参照</a> を variable として指定できます。
direction	オプション - ターゲット配列を検索するための添え字順序です。添え字付き変数の値は、1 が昇順添え字（既定）で、-1 は降順です。添え字なしのローカル変数では、1（既定）が唯一許可される値です。
target	オプション - variable の次または前のノードの現在のデータ値を返します。次のノードか前のノードかは、direction の設定によって決定されます。target を指定するには direction 値を指定する必要があります。添え字なしのローカル変数では、direction は 1 に設定されなければなりません。variable が定義されていない場合、target 値はそのままで変化しません。target 引数は、^\$ROUTINE などの構造化システム変数（SSVNs）とは一緒に使用できません。

### 概要

\$ORDER は、主に、指定された開始ポイントから指定された添え字レベルで添え字付きの変数をループするのに使用されます。これは、照合順で一連の変数を返します。また、添え字の順序にギャップを設けることができます。

\$ORDER が返す値は、使用する引数によって決まります。

- \$ORDER(variable) は、variable が添え字付きであれば、次の定義された添え字数を返します。返される添え字は、variable に指定したものと同一レベルです。例えば、\$ORDER(^client(4,1,6)) は、次のサード・レベル添え字を返します。変数 ^client(4,1,7) が存在する場合、これは 7 になります。  
 \$ORDER(variable) は、variable が添え字なしのローカル変数の場合、アルファベット照合順で次の定義済みのローカル変数名を返します。例えば \$ORDER は、以下の順序で、以下の定義済みのローカル変数 a、a0a、a1、a1a、aa、b、bb、および c を返します（以下の例を参照してください）。
- \$ORDER(variable,direction) は、変数の次の添え字か前の添え字を返します。direction を 1（次の添え字、既定）または -1（前の添え字）として指定できます。  
 添え字なしのローカル変数では、\$ORDER は direction 1（次）の順序でのみ変数を返します。-1（前）の direction を指定することはできません。これを行おうとすると <FUNCTION> エラーになります。
- \$ORDER(variable,direction,target) は変数に対する添え字を返し、target を現在のデータ値に設定します。これは direction 設定に応じて、添え字付きの変数に対する次の添え字、または前の添え字のいずれかになります。添え字なしのローカル変数では、target に対する現在のデータ値を返すために direction は 1 に設定されなければなりません。target 引数は、^\$ROUTINE などの構造化システム変数（SSVNs）とは一緒に使用できません。ZBREAK コマンドでは、target 引数をウォッチポイントとして指定することはできません。

## 返される最初の添え字

\$ORDER ループは、指定された変数の次の変数、または最初の変数から開始できます。

- 指定されたポイントから開始：SET key=\$ORDER(^mydata(99)) は、99 よりも 1 つ大きい添え字 (添え字 100) を返します (存在する場合)。引数で指定するノード (ここでは添え字 99) は存在している必要はありません。すべての正の添え字を返すには、SET key=\$ORDER(^mydata(-1)) と指定できます。すべての負の添え字を返すには、SET key=\$ORDER(^mydata(0),-1) と指定できます。
- 最初から開始：SET key=\$ORDER(^mydata("")) は、照合順で最初の添え字付きの変数を返します。この方法は、レベルに正の添え字と同様に負の添え字が含まれているときに必要です。

以下の例では、数字の昇順で正負両方のファースト・レベル添え字を返しています。

```
SET mydata(1)="a",mydata(-3)="C",mydata(5)="e",mydata(-5)="E"
// Get first subscript
SET key=$ORDER(mydata(""))
WHILE (key!="") {
    WRITE key,!
    // Get next subscript
    SET key = $ORDER(mydata(key))
}
```

\$ORDER が指定されたレベルで添え字の最後に到達すると、NULL 文字列 ("") を返します。ループの中で \$ORDER を使用するときは、必ずこの値のテストを含める必要があります。

\$ORDER の開始コードと失敗コードの値は、どちらも NULL 文字列 ("") です。\$ORDER は NULL 文字列で起動し、終了するため、正と負の両方の添え字を持つノードを正しく返します。

\$ORDER を使用することで、定義済みのローカル変数の限定されたサブセットを返すことができます。[引数なしの WRITE](#) を使用することで、定義済みのローカル変数をすべて表示できます。

## 例

この例では、ローカル変数を返しています。\$ORDER は、添え字付きのグローバル変数や添え字付きのプロセス・プライベート・グローバル変数を返すこともできます。

以下の例では、WHILE ループで \$ORDER を使用して、mydata(n) グローバルのファースト・レベル添え字をすべて返しています。

### ObjectScript

```
SET mydata(1)="a",mydata(3)="c",mydata(7)="g"
// Get first subscript
SET key=$ORDER(mydata(""))
WHILE (key!="") {
    WRITE key,!
    // Get next subscript
    SET key = $ORDER(mydata(key))
}
```

以下の例では、WHILE ループで \$ORDER を使用して、mydata(1,n) グローバルのセカンド・レベル添え字をすべて返しています。ファースト・レベル添え字とサード・レベル添え字は無視されます。この例では、添え字の数と対応する変数値の両方を返しています。

### ObjectScript

```
SET mydata(1,1)="a",mydata(1,3)="c",mydata(1,3,1)="lcase",mydata(1,1)="A",mydata(1,7)="g"
SET key=$ORDER(mydata(1,""),1,target)
WHILE (key!="") {
    WRITE key," = ",target,!
    // Get next subscript
    SET key = $ORDER(mydata(1,key),1,target)
}
```

以下の例では、WHILE ループで \$ORDER を使用して、添え字なしのローカル変数を返しています。ローカル変数は照合順で返されます。この例では、ローカル変数名とそれらの値の両方を返しています。添え字なしのローカル変数をループする際、間接演算子 @ を使用する必要があります。この例では、照合順で b の次のローカル変数から開始しています (この場合、bminus)。その後、照合順でその後のすべての定義済みのローカル変数をループします。変数 foo と target をリストしないようにするために、これらの変数は、ローカル変数ではなく、プロセス・プライベート・グローバルとして定義されます。

### ObjectScript

```
SET a="great",b="good",bminus="pretty good",c="fair",d="poor",f="failure"
SET ^|foo="b"
SET ^|foo=$ORDER(^|foo,1,^|target)
WHILE ^|foo != "" {
WRITE ^|foo," = ",^|target,!
SET ^|foo=$ORDER(^|foo,1,^|target)
}
```

## \$ORDER の使用

\$ORDER は、通常、連続した整数の添え字を使用しない配列のノードを検索するために、ループ処理と共に使用されます。\$ORDER は、次の既存のノードの添え字を返すだけです。以下はその例です。

### ObjectScript

```
SET struct=""
FOR {
SET struct=$ORDER(^client(struct))
QUIT:struct=""
WRITE !,^client(struct)
}
```

上述のルーチンは、^client グローバル配列にある上位レベル・ノードの値すべてを出力します。

\$ORDER は既存のノードの添え字を返しますが、値が含まれていないノードもあります。ループの中で \$ORDER を使用して、データをコマンド (WRITE など) に送るときは、値なしノードの \$DATA チェックを含める必要があります。例えば、前述の例での WRITE コマンドは、以下のように後置条件テストを付けて指定できます。

### ObjectScript

```
WRITE:($DATA(^client(struct))#10) !,^client(struct)
```

このテストには、値なしポインタ・ノードと値なし端末ノードの場合が含まれます。コードが単純な FOR ループでは処理できないとき、ブロック構造化された DO にその一部を委任できます。

## グローバル参照

variable は、グローバル変数である場合、異なるネームスペース内のグローバルを指定して、[拡張グローバル参照](#)にすることができます。存在しないネームスペースを指定した場合、InterSystems IRIS® データ・プラットフォームは <NAMESPACE> エラーを発行します。特権を持たないネームスペースを指定した場合、InterSystems IRIS は <PROTECT> エラーを発行し、続けてグローバル名とデータベース・パスを表示します (例:<PROTECT>

^myglobal(1),c:\intersystems\iris\mgr\)。ユーザが読み取り許可を持たないネームスペースへの添え字マッピングがグローバル変数にある場合、<PROTECT> エラーの情報に元のグローバル参照が表示されます。これは、特権を持たないネームスペースの添え字は表示できないためです。ただし、<PROTECT> エラーのデータベース・パスには、元のデータベースではなく、保護されたデータベースが表示されます。

variable は、添え字付きのグローバル変数である場合、[ネイキッド・グローバル参照](#)にできます。ネイキッド・グローバル参照は配列名なしで指定され、最近実行されたグローバル参照を示します。以下はその例です。

## ObjectScript

```
SET var1=^client(4,5)
SET var2=$ORDER(^(""))
WRITE "var1=",var1,!,"var2=",var2
```

最初の SET コマンドは、参照に対する添え字レベルを含む現在のグローバル参照を設定します。\$ORDER 関数はネイキッド・グローバル参照を使用して、このレベルの最初の添え字を返します。例えば、この添え字付きグローバルが定義されている場合、^client(4,1) を示す値 1 を返します。^client(4,1) が定義されておらず、この添え字付きグローバルが定義されている場合は、^client(4,2) を示す値 2 を返す、というようになります。

\$ORDER の 3 つの引数はすべて、ネイキッド・グローバル参照を取るか、グローバル参照を指定することができます。ただし、direction が明示的なグローバル参照を指定している場合、後に続くネイキッド・グローバル参照は direction グローバル参照を使用しません。以下の例に示すように、引き続き、前に構築されたグローバル参照を使用します。

## ObjectScript

```
SET ^client(4,3)="Jones"
SET ^client(4,5)="Smith"
SET ^dir(1)=-1
SET rtn=$ORDER(^client(4,5),-1)
WRITE $ZREFERENCE,!
/* naked global ref is ^client(4,3) */
SET rtn=$ORDER(^client(4,5),^dir(1))
WRITE $ZREFERENCE,!
/* NOTE: naked global ref is ^client(4,3) */
SET rtn=$ORDER(^client(4,5),^dir(1),^(1))
WRITE $ZREFERENCE
/* NOTE: naked global ref is ^client(4,1) */
WRITE ^client(4,1),!
SET rtn=$ORDER(^client(4,5),^dir(1),^targ(1))
WRITE $ZREFERENCE
/* naked global ref is ^targ(1) */
WRITE ^targ(1),!
SET ^rtn(1)=$ORDER(^client(4,5),^dir(1),^targ(2))
WRITE $ZREFERENCE
/* naked global ref is ^rtn(1) */
WRITE ^targ(2)
```

詳細は、“[最新のグローバル参照のチェック](#)”を参照してください。

## \$ORDER と \$DOUBLE の添え字

\$DOUBLE 浮動小数点数は、添え字識別子として使用できます。ただし、添え字識別子として使用した場合、\$DOUBLE 値は文字列に変換されます。\$ORDER はこのタイプの添え字を返すとき、\$DOUBLE 浮動小数点数ではなく、数値文字列として返します。

## 関連項目

- [\\$DATA 関数](#)
- [\\$GET 関数](#)
- [\\$QUERY 関数](#)
- [\\$ZREFERENCE 特殊変数](#)
- [グローバルについての正式な規則](#)

## \$PARAMETER (ObjectScript)

指定されたクラス・パラメータの値を返します。

### 構文

```
$PARAMETER(class,parameter)
```

### 引数

引数	説明
class	オプション クラス名、またはクラス・インスタンスへの <a href="#">オブジェクト参照 (OREF)</a> 。省略した場合は、現在のクラス・インスタンスのオブジェクト参照が使用されます。省略した場合は、プレースホルダのコンマを指定する必要があります。
parameter	パラメータ名です。有効な文字列に評価される式。文字列の値は、class で指定されたクラスの既存のパラメータ名と一致する必要があります。

### 説明

\$PARAMETER は、指定されたクラスのパラメータの値を返します。\$PARAMETER は、現在のクラス・コンテキストまたは指定されたクラス・コンテキストで、この parameter を検索できます。class 名を引用符付き文字列として指定するか、[OREF](#) を指定するか、または class 引数を省略して既定として現在のインスタンスを使用することができます ([\\$THIS](#) を参照)。class はオプションで指定できますが、区切り文字としてコンマを指定する必要があります。

例えば以下のようになります。

#### ObjectScript

```
WRITE $PARAMETER("%Library.Boolean","XSDTYPE")
```

以下の例に示すように、オブジェクト構文を使用してパラメータの値を返すにはいくつかの方法があります。

#### ObjectScript

```
WRITE "ObjectScript function:",!
WRITE $PARAMETER("Sample.Person","EXTENTQUERYSPEC")
WRITE !,"class parameter:",!
WRITE ##class(Sample.Person).#EXTENTQUERYSPEC
WRITE !,"instance parameter:",!
SET myinst=##class(Sample.Person).%New()
WRITE myinst.%GetParameter("EXTENTQUERYSPEC")
WRITE !,"instance parameter:",!
WRITE myinst.#EXTENTQUERYSPEC
```

### 無効な値

- ・ \$PARAMETER("","XMLTYPE") : 無効な OREF (空の文字列、整数、小数など) を呼び出そうとすると、<INVALID OREF> エラーになります。
- ・ \$PARAMETER("bogus","XMLTYPE") : 存在しないクラスを呼び出そうとすると、<CLASS DOES NOT EXIST> の後に指定されたクラス名が記述されたエラーが発生します。パッケージ名が指定されていない場合、InterSystems IRIS は既定値を採用します。例えば、存在しない class “bogus” を呼び出そうとすると、エラー <CLASS DOES NOT EXIST> \*User.bogus が出力されます。
- ・ \$PARAMETER(,"XMLTYPE") : 現在のオブジェクト・インスタンスが確立されていない場合に既定に設定しようとする、<NO CURRENT OBJECT> エラーが発生します。



- ・ `$PARAMETER("%SYSTEM.Task", "")` : 無効な parameter 名 (例えば、空の文字列) を参照しようとしたり、パラメータを数字で参照しようとしたりすると、`<ILLEGAL VALUE>` エラーが発生します。
- ・ `$PARAMETER("%SYSTEM.Task", "MakeCoffee")` : 存在しない parameter 名を参照しようすると、空の文字列 ("" ) が返されます。

## 例

以下の例では、クラス名を指定して、XMLTYPE および XSDTYPE パラメータのクラス既定値を返します。

### ObjectScript

```
WRITE $PARAMETER("%SYSTEM.Task", "XMLTYPE"), !  
WRITE $PARAMETER("%Date", "XSDTYPE")
```

以下の例では、OREF を指定して、このインスタンスの XMLTYPE パラメータの値を返します。

### ObjectScript

```
SET oref=##class(%SYSTEM.Task).%New()  
WRITE $PARAMETER(oref, "XMLTYPE")
```

以下の例では、\$PARAMETER 構文とクラス構文を使用してシステム・パラメータを返します。

### ObjectScript

```
WRITE $PARAMETER("%SYSTEM.SQL", "%RandomSig"), !  
WRITE ##class(%SYSTEM.SQL).%RandomSig
```

## 関連項目

- ・ [\\$CLASSMETHOD](#) 関数
- ・ [\\$CLASSNAME](#) 関数
- ・ [\\$METHOD](#) 関数
- ・ [\\$PROPERTY](#) 関数
- ・ [\\$THIS](#) 特殊変数

## \$PIECE (ObjectScript)

区切り文字を使用して、部分文字列を返すか、置換します。

### 構文

```
$PIECE(string,delimiter,from,to)
$P(string,delimiter,from,to)

SET $PIECE(string,delimiter,from,to)=value
SET $P(string,delimiter,from,to)=value
```

### 引数

引数	説明
string	区切られた部分文字列が指定されるターゲット文字列。string には、評価結果が引用符で囲んだ文字列または数値になる式を指定します。SET \$PIECE 構文では、string は変数または多次元プロパティである必要があります。
delimiter	string 内の部分文字列の特定に使用される区切り文字。delimiter には、1 つ以上の文字で構成される引用符で囲まれた文字列として評価される式を指定します。
from	オプション - 評価結果が、string 内での部分文字列の位置、または部分文字列の範囲の先頭を指定するコードとなる式。部分文字列は、delimiter で区切られ、1 からカウントされます。許可される値は、n (string の先頭からの部分文字列カウントを指定する正の整数)、* (string の最後の部分文字列の指定)、および *-n (string の末尾から逆向きの部分文字列のオフセット整数カウント) です。SET \$PIECE 構文は、*+n (string の末尾の先に追加する部分文字列のオフセット整数カウント) もサポートします。したがって、1 つ目の区切り部分文字列は 1 であり、2 つ目の区切り部分文字列は 2 であり、最後の区切り部分文字列は * であり、最後から 2 番目の区切り部分文字列は *-1 です。from が省略されている場合は、既定値の 1 つ目の区切り部分文字列が使用されます。
to	オプション - 評価結果が、string 内の部分文字列の範囲の最終部分文字列を指定するコードとなる式。from と共に使用される必要があります。許可される値は、n (string の先頭からの部分文字列カウントを指定する正の整数)、* (string 内の最後の部分文字列の指定)、および *-n (string の末尾からの部分文字列のオフセット整数カウント) です。SET \$PIECE 構文は、*+n (string の末尾の先に追加する部分文字列の範囲のオフセット整数) もサポートします。string 内で to が from より前である場合は、何の処理も実行されず、エラーは発生しません。

### 概要

\$PIECE は、delimiter の存在により string 内の部分文字列を識別します。delimiter が string に含まれていない場合は、その文字列全体が単一の部分文字列として扱われます。

\$PIECE には以下の 2 つの使用方法があります。

- string の部分文字列を返します。これには構文 \$PIECE(string,delimiter,from,to) が使用されます。
- string の部分文字列を置換します。これにより、部分文字列が識別され、別の部分文字列に置き換えられます。置換部分文字列は、元の部分文字列と同じ長さでも、長くても、短くてもかまいません。これには構文 SET \$PIECE(string,delimiter,from,to)=value が使用されます。

### 部分文字列を返す方法

string から指定された部分文字列 (piece) を返す場合、返される部分文字列は、使用される引数によって変わります。

- ・ \$PIECE(string,delimiter) は、string の先頭の部分文字列を返します。delimiter が string 内に存在する場合は、最初の delimiter の前にある部分文字列です。delimiter が string 内に存在しない場合、返される部分文字列は string です。
- ・ \$PIECE(string,delimiter,from) は、位置が from 引数によって指定される部分文字列を返します。部分文字列は、区切り文字、および string の先頭と末尾によって区切られます。区切り文字自体は返されません。
- ・ \$PIECE(string,delimiter,from,to) では、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 (from および to で指定された部分を含む) の部分文字列が返されます。この 4 つの引数の \$PIECE 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 をすべて含んだ部分文字列を返します。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 の最後まですべての部分文字列が含まれます。

## 引数

### string

部分文字列を返すために \$PIECE を使用する場合、string は、引用符で囲まれた文字列リテラル、キャノニック形式の数値、変数、オブジェクト・プロパティ、または評価結果が文字列または数値になる有効な ObjectScript 式にできます。ターゲット文字列として NULL 文字列(“)を指定すると、\$PIECE はその他の引数値に関係なく、常に NULL 文字列を返します。

通常、ターゲット文字列には、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 内のデータ値として使用することはできません。

1 つの部分文字列を置換するために、等号の左側で SET と共に \$PIECE を使用する場合、string は変数または多次元プロパティ参照にすることができますが、非多次元オブジェクト・プロパティにすることはできません。

### delimiter

string 内の部分文字列の区切りとして使用される検索文字列です。引用符で囲まれた文字列リテラル、キャノニック形式の数値、変数、または評価結果が文字列または数値となる有効な ObjectScript 式にできます。

通常、区切り文字には、string 内で決してデータとして使用されることがなく、部分文字列を区切る区切り文字としてのみ使用される特定の文字が指定されます。例えば、delimiter が “”” である場合は、string の値 “Red`Orange`Yellow” には 3 つの区切り部分文字列が含まれます。

区切り文字には、複数文字から成る文字列を指定できますが、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。例えば、delimiter が “#” である場合は、string の値 “Red`Orange`#`Yellow#Green#`Blue” には、“Red`Orange” と “`Yellow#Green#`Blue” という 2 つの区切り部分文字列が含まれます。

一般に、string の先頭や末尾には delimiter は使用されません。string の先頭や末尾に delimiter が使用されている場合、\$PIECE は、その区切り文字を、NULL 文字列(“)値を含む部分文字列の境界を示すものとして扱います。例えば、delimiter が “”” である場合は、string の値 “`Red`Orange`Yellow`” には 5 つの区切り部分文字列が含まれており、1 つ目と 5 つ目の部分文字列は NULL 文字列値です。

指定された区切り文字が string に存在しない場合、\$PIECE は string 全体を返します。指定された区切り文字が NULL 文字列(“)である場合、\$PIECE は、NULL 文字列を返します。

### from

string 内の部分文字列の位置。n (正の整数) を使用して、string の先頭から区切り部分文字列をカウントします。\* を使用して、string 内の最後の区切り部分文字列を指定します。\*-n を使用して、string 内の最後の区切り部分文字列からオフセット分だけ区切り部分文字列をカウントします。

- ・ 1 は、string の最初の部分文字列を指定します (これは最初の delimiter の前にある部分文字列です)。指定された区切り文字が string にない場合、from 値が 1 であれば string が返されます。from が省略されている場合は、既定値の 1 が使用されます。

- ・ 2 は、string の 2 つ目の部分文字列を指定します (この部分文字列は、1 つ目と 2 つ目の delimiter の間にある部分文字列であるか、1 つ目の delimiter と string の末尾の間にある部分文字列です)。
- ・ \* は、string の最後の部分文字列を指定します (これは最後の delimiter の後ろにある部分文字列です)。指定された区切り文字が string にない場合、from 値が \* であれば string が返されます。
- ・ \*-1 は、string の最後から 2 番目の部分文字列を指定します。\*-n は、string の最後の部分文字列からオフセット分だけカウントします。\*-0 は string の最後の部分文字列であり、\* と \*-0 は機能的に同じです。
- ・ SET \$PIECE 構文のみ - \*+n (アスタリスクとそれに続く正の整数) が、string の末尾のオフセット分先の位置に区切り部分文字列を追加します。したがって、\*+1 の場合は、string の末尾の先に区切り部分文字列が追加され、\*+2 の場合は、string の末尾の 2 つ先の位置に区切り部分文字列が追加され、区切り文字が埋め込まれます。
- ・ from が NULL 文字列 ("")、ゼロ、または負数であるか、string 内の部分文字列数よりも大きいカウントまたはオフセットを指定している場合、\$PIECE は NULL 文字列を返します。

\$PIECE は from の数値をキャノニック形式に変換してから (先頭のプラス記号とマイナス記号を解決して先頭のゼロを削除します)、その数値の小数部分を切り捨てます。

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 内の部分文字列の番号。返される文字列には、from の位置の文字列と to の位置の文字列に加えて、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。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。

n (正の整数) を使用して、string の先頭から区切り部分文字列をカウントします。\* を使用して、string 内の最後の区切り部分文字列を指定します。\*-n を使用して、string 内の最後の区切り部分文字列からの逆方向にオフセット分だけ区切り部分文字列をカウントします。

SET \$PIECE 構文のみ - \*+n (アスタリスクとそれに続く正の整数) が、string の末尾の先に追加する部分文字列の範囲の末尾を指定します。

- ・ 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成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。to が区切られた部分文字列の数よりも大きい場合、返される文字列には、from 位置の部分文字列から string 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、\$PIECE は from の部分文字列を返します。この状況が発生する可能性があるのは、from と to が同じ値であるか、同じ部分文字列を参照する異なる値である場合です。
- ・ from が to より大きい値であるか、ゼロ (0) であるか、または NULL 文字列 ("") である場合は、\$PIECE は NULL 文字列を返します。

\$PIECE は to の数値をキャノニック形式に変換してから (先頭のプラス記号とマイナス記号を解決して先頭のゼロを削除します)、その数値の小数部分を切り捨てます。

## \*-n および \*+n 引数値の指定

変数を使用して \*-n または \*+n を指定する場合は、常に、引数自体でアスタリスクと符号文字を指定する必要があります。

\*-n の有効な指定内容は以下のとおりです。

## ObjectScript

```
SET count=2
SET alph="a^b^c^d"
WRITE $PIECE(alph, "^", *-count)
```

## ObjectScript

```
SET count=-2
SET alph="a^b^c^d"
WRITE $PIECE(alph,"^",*+count)
```

\*+n の有効な指定内容は以下のとおりです。

## ObjectScript

```
SET count=2
SET alph="a^b^c^d"
SET $PIECE(alph,"^",*+count)="F"
WRITE alph
```

これらの引数値内では、空白を使用できます。

## 例：区切り部分文字列を返す方法

以下の例では、それぞれの \$PIECE は、“,” 区切り文字で識別される指定の部分文字列を返します。

## ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
WRITE $PIECE(colorlist,","),! ; returns "Red" (substring 1) by default
WRITE $PIECE(colorlist,",",3),! ; returns "Blue" the third substring
WRITE $PIECE(colorlist,",",*),! ; returns "Black" the last substring
WRITE $PIECE(colorlist,",",*-1),! ; returns "Orange" the next-to-last substring
```

以下の例では、\$PIECE は数値の整数部分と小数部分を返します。

## ObjectScript

```
SET int=$PIECE(123.999, ".")
SET frac=$PIECE(123.999, ".", *)
WRITE "integer=",int," fraction =.",frac
```

以下の例は、colorlist の 3 番目から 5 番目の部分文字列である “Blue,Yellow,Orange” を、“,” で区切って返します。

## ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract=$PIECE(colorlist,",",3,5)
WRITE extract
```

以下の WRITE 文はすべて、最初の部分文字列 “123” を返し、from と to の 値が 1 のとき、これらの形式は同等であることがわかります。

## ObjectScript

```
SET numlist="123#456#789"
WRITE !,"2-arg=", $PIECE(numlist,"#")
WRITE !,"3-arg=", $PIECE(numlist,"#",1)
WRITE !,"4-arg=", $PIECE(numlist,"#",1,1)
```

以下の例では、string 内に delimiter が使用されていないため、\$PIECE 関数は両方とも string 文字列全体を返します。

## ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract1=$PIECE(colorlist,"#")
SET extract2=$PIECE(colorlist,"#",1,4)
WRITE "#    =",extract1,!,"#",1,4=",extract2
```

以下の例では、\$PIECE はオブジェクト・プロパティの 2 番目の部分文字列を返します。

## ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SchemaPath="MyTests,Sample,Cinema"
WRITE "whole schema path: ",tStatement.%SchemaPath,!
WRITE "2nd piece of schema path: ",$PIECE(tStatement.%SchemaPath,"",2),!
```

以下の 2 つの例では、より複雑な区切り文字を使用します。

この例は、区切り文字列 "#-" を使用して、文字列 numlist の 3 つの部分文字列を返します。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。

## ObjectScript

```
SET numlist="1#2-3#-#45##6#-#789"
WRITE !,$PIECE(numlist,"#-",1)
WRITE !,$PIECE(numlist,"#-",2)
WRITE !,$PIECE(numlist,"#-",3)
```

以下の例では、\$CHAR 関数を使用して指定され、連結演算子 ( ) を使用して string に挿入されている非 ASCII 区切り文字 (ここでは pi の Unicode 文字) を使用します。

## ObjectScript

```
SET a = $CHAR(960)
SET colorlist="Red"_a_"Green"_a_"Blue"
SET extract1=$PIECE(colorlist,a)
SET extract2=$PIECE(colorlist,a,2)
SET extract3=$PIECE(colorlist,a,2,3)
WRITE extract1,! ,extract2,! ,extract3
```

## SET \$PIECE を使用した部分文字列の置換

SET コマンドを使用して値を代入するとき、等号の右側と同様、左側にも \$PIECE を使用できます。\$PIECE は、等号の左側に使用されると、代入値で置き換えられる部分文字列の値を示します。

等号の左側で SET と共に \$PIECE を使用する場合、string は有効な変数名にすることができます。変数が存在しない場合は、SET \$PIECE が変数を定義します。string 引数は、[多次元プロパティ参照](#)にすることもできますが、非多次元オブジェクト・プロパティにすることはできません。非多次元オブジェクト・プロパティで SET \$PIECE を使用しようとすると、<OBJECT DISPATCH> エラーが発生します。

このコンテキストで \$PIECE (\$LIST および \$EXTRACT) を使用すると、単に値を返すだけではなく、既存の値を変更してしまうため、他の基本関数とは異なります。\$PIECE (または \$LIST や \$EXTRACT) を伴った SET (a,b,c,...)=value 構文は、その関数で相対オフセット構文 \* (文字列の末尾を表す)、\*-n または \*+n (文字列の末尾からの相対オフセットを表す) を使用する場合、等号の左側では使用できません。その代わりに、SET a=value,b=value,c=value,... 構文を使用する必要があります。

## 例：区切り部分文字列を置換する方法

以下の例は、colorlist の値を "Magenta,Green,Cyan,Yellow,Orange,Black" に変更します。

## ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
WRITE colorlist,!
SET $PIECE(colorlist,"",1)="Magenta"
WRITE colorlist,!
SET $PIECE(colorlist,"",*-3)="Cyan"
WRITE colorlist,!
```

置換部分文字列は、元の部分文字列より長くても短くてもかまわず、区切り文字を含んでいてもかまいません。



## ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
WRITE colorlist,!
SET $PIECE(colorlist,"",3)="Turquoise,Aqua,Teal"
WRITE colorlist,!
```

引数 from と to を指定すると、その範囲に該当する部分文字列が指定の値に置換されます。この例では 4 ～ 6 番目の区切り部分文字列が相当します。

## ObjectScript

```
SET colorlist="Red,Blue,Yellow,Green,Orange,Black"
WRITE !,colorlist
SET $PIECE(colorlist,"",4,6)="Yellow+Blue,Yellow+Red"
WRITE !,colorlist
```

区切り部分文字列カウントによって (n を使用)、または string の末尾からのオフセットによって (\*+n を使用)、1 つ以上の区切り部分文字列を追加できます。SET \$PIECE は、指定された位置に区切り部分文字列を追加するために、必要に応じて区切り文字を追加します。以下の例では、どちらも colorlist の値を "Green`Blue`Red" に変更して、追加の空文字列を区切られた部分文字列に埋め込みます。

## ObjectScript

```
SET colorlist="Green^Blue"
SET $PIECE(colorlist,"^",4)="Red"
WRITE colorlist
```

## ObjectScript

```
SET colorlist="Green^Blue"
SET $PIECE(colorlist,"^",*+2)="Red"
WRITE colorlist
```

string に delimiter が存在しない場合、\$PIECE は string を 単一の文字として扱い、前述どおり置換を行います。from 引数を指定していない場合、元の string は新しい値で置き換えられます。

## ObjectScript

```
SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"^")="Purple^Orange"
WRITE colorlist
```

delimiter が string に含まれず、from に 1 より大きい整数が指定されている場合、\$PIECE は、(from-1) 個の区切り文字を追加して、指定された値を string の末尾に追加します。

## ObjectScript

```
SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"^",3)="Purple"
WRITE colorlist
```

from が文字列の先頭より前の位置を指している場合は、InterSystems IRIS は何の処理も実行しません。

## ObjectScript

```
SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"",*-7)="Purple"
WRITE colorlist
```

from が文字列の先頭より前の位置を指しており、to が指定されている場合は、InterSystems IRIS は from を位置 1 として扱います。



## ObjectScript

```
SET colorlist="Red,Green,Blue"
WRITE colorlist,!
SET $PIECE(colorlist,"",*-7,1)="Purple"
WRITE colorlist
```

## 文字列変数の初期化

string 変数は、値を割り当てる前に定義する必要はありません。以下の例は、newvar を文字パターン ">>>>>TOTAL" に初期化します。

## ObjectScript

```
SET $PIECE(newvar,">",7)="TOTAL"
WRITE newvar
```

詳細は、SET コマンド・ドキュメントの ["\\$PIECE と \\$EXTRACT を使用した SET"](#) セクションを参照してください。

## 区切り文字が NULL 文字列

区切り文字が NULL 文字列の場合、from 引数や to 引数の値に関係なく、元の string が新しい値に置換されます。

以下の 2 つの例では、どちらも colorlist は "Purple" に設定されます。

## ObjectScript

```
SET colorlist="Red,Green,Blue"
WRITE !,colorlist
SET $PIECE(colorlist,"")="Purple"
WRITE !,colorlist
```

## ObjectScript

```
SET colorlist="Red,Blue,Yellow,Green,Orange,Black"
WRITE !,colorlist
SET $PIECE(colorlist,"",3,5)="Purple"
WRITE !,colorlist
```

## 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 (;) が含まれます。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-引数形式の \$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、文字列内の部分文字列の数を調べてから、\$PIECE を使用して、以下の例で示すように、個々の部分文字列を抽出します。

## ObjectScript

```
SET sentence="The quick brown fox jumped over the lazy dog's back."
SET delim=" "
SET countdown=$LENGTH(sentence,delim)
SET countup=1
FOR reps=countdown:-1:1 {
    SET extract=$PIECE(sentence,delim,countup)
    WRITE !,countup," ",extract
    SET countup=countup+1
}
WRITE !,"All done!"
```

## NULL 値

\$PIECE は、値が NULL 文字列の部分文字列と、存在しない部分文字列を区別しません。両方とも、NULL 文字列を返します。例えば、以下の 2 つの例は両方とも、from 値の 7 に対して NULL 文字列値を返します。

### ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract1=$PIECE(colorlist,"",6)
SET extract2=$PIECE(colorlist,"",7)
WRITE "6=",extract1,!,"7=",extract2
```

### ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black,"
SET extract1=$PIECE(colorlist,"",6)
SET extract2=$PIECE(colorlist,"",7)
WRITE "6=",extract1,!,"7=",extract2
```

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 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 内に NULL 値がある場合を示しています。部分文字列 1 と 3 が抽出されます。これらの部分文字列は存在しますが、両方とも NULL 文字列を含みます。(部分文字列 1 は、最初の区切り文字の前にある文字列として定義されています)。

### ObjectScript

```
SET colorlist=",Red,,Blue,"
SET extract1=$PIECE(colorlist,"",)
SET extract3=$PIECE(colorlist,"",3)
WRITE !,"sub1=",extract1,!,"sub3=",extract3
```

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。

### ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract=$PIECE(colorlist,"",0)
WRITE !,"Length=", $LENGTH(extract),!,"Value=",extract
```

### ObjectScript

```
SET colorlist="Red,Green,Blue,Yellow,Orange,Black"
SET extract=$PIECE(colorlist,"",8,20)
WRITE !,"Length=", $LENGTH(extract),!,"Value=",extract
```

## 数値評価の強制

\$PIECE (または任意の ObjectScript 関数) の前に単項 + 符号を配置すると、戻り値の数値評価が強制されます。これは、**キャノニック形式**で数値部分文字列を返します。これは、非数値部分文字列を 0 として返します。これは、混合数値文字列の先頭の数字部分を返します。これは、NULL 文字列値または存在しない部分文字列について 0 を返します。

この数値評価の強制を以下の例で示します。

### ObjectScript

```
SET str="snow white,7dwarves,+007.00,99.90,, -0,"
WRITE "Substrings:",!
FOR i=1:1:7 {WRITE i,"=", $PIECE(str,"",i)," "}
WRITE !,"Forced Numerics:",!
FOR i=1:1:7 {WRITE i,"=", +$PIECE(str,"",i)," "}
```

## 入れ子になった \$PIECE 処理

複雑な抽出を実行するには、\$PIECE 参照を相互に入れ子にすることができます。内側の \$PIECE は、外側の \$PIECE によって処理される部分文字列を返します。各 \$PIECE はそれぞれの区切り文字を使用します。例えば、以下は州の略語である "MA" を返します。

### ObjectScript

```
SET patient="John Jones/29 River St./Boston MA 02095"
SET patientstateaddress=$PIECE($PIECE(patient,"/",3)," ",2)
WRITE patientstateaddress
```

以下は、入れ子になった \$PIECE 処理のもう 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、nestlist の 2 番目の部分文字列である "A,B,C" を見つけます。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。

### ObjectScript

```
SET nestlist="1,2,3^A,B,C^@#!"
WRITE $PIECE($PIECE(nestlist,"^",2)," ",1,2)
```

## \$EXTRACT および \$LIST と比較した \$PIECE コマンド

\$PIECE は、文字列内のユーザ定義の区切り文字をカウントすることにより、部分文字列を決定します。\$PIECE は、区切り文字として使用するための文字 (または文字列) の複数のインスタンスを含む通常の文字列を入力として取ります。

\$EXTRACT は、文字列の最初の部分から文字をカウントすることにより、部分文字列を決定します。\$EXTRACT は、入力として通常の文字列を取ります。

\$LIST は、リストの最初から要素 (文字ではない) をカウントすることにより、エンコードされたリストから要素を決定します。\$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を除外することが特定のデータ型 (ビット文字列データなど) で不適切な場合は、\$LISTBUILD と \$LIST 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。区切り文字列は、\$LISTFROMSTRING 関数を使用してリストに変換できます。リストは、\$LISTTOSTRING 関数を使用して区切り文字列に変換できます。

\$PIECE 関数と \$LIST 関数によって使用されるデータ格納方法は互換性がなく、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。

## 関連項目

- [SET コマンド](#)
- [\\$EXTRACT 関数](#)
- [\\$LENGTH 関数](#)
- [\\$LIST 関数](#)
- [\\$LISTBUILD 関数](#)
- [\\$LISTFROMSTRING 関数](#)
- [\\$LISTTOSTRING 関数](#)
- [\\$REVERSE 関数](#)

## \$PREFETCHOFF (ObjectScript)

グローバルの事前フェッチを終了します。

### 構文

```
$PREFETCHOFF(gref,gref2)
```

### 引数

引数	説明
<i>gref</i>	オプション - グローバル参照。
<i>gref2</i>	オプション - 範囲の指定に使用するグローバル参照。

### 説明

\$PREFETCHOFF は、\$PREFETCHON で現在のプロセスに対して確立されたグローバル・ノードの事前フェッチを無効にします。

\$PREFETCHOFF には、以下の 3 つの形式があります。

- ・ `$PREFETCHOFF()` は、現在のプロセスで確立されたすべての事前フェッチを無効にします。
- ・ `$PREFETCHOFF(gref)` は、*gref* ノードとそのすべての下位ノードの事前フェッチを無効にします。*gref* の値は、\$PREFETCHON の値と正確に一致する必要があります。
- ・ `$PREFETCHOFF(gref,gref2)` は、*gref* から *gref2* の範囲のノードの事前フェッチを無効にします。*gref* および *gref2* は、同じグローバルのノードである必要があります。*gref* および *gref2* の値は、\$PREFETCHON の値と正確に一致する必要があります。値の範囲の一部を無効にすることはできません。

正常に終了すると、\$PREFETCHOFF() は 0 を返します。無効にする事前フェッチがない場合でも、0 を返します。

正常に終了すると、\$PREFETCHOFF(*gref*) および \$PREFETCHOFF(*gref*,*gref2*) はコンマで区切られた 6 つの整数の文字列を返します。これらの 6 つの値はそれぞれ、事前フェッチされたブロック数、実行された入出力数、事前フェッチ処理数、事前フェッチ・ディスク時間 (ミリ秒)、バックグラウンド・ジョブで事前フェッチされたブロック数、およびバックグラウンド・ジョブで実行された入出力数です。

失敗すると、\$PREFETCHOFF のすべての形式で -1 が返されます。`$PREFETCHOFF(gref)` および `$PREFETCHOFF(gref,gref2)` は、指定されたグローバルまたはグローバルの範囲に正確に一致する \$PREFETCHON がない場合、または指定された事前フェッチのグローバルまたはグローバルの範囲が既に無効になっている場合に、-1 を返します。

### 引数

#### *gref*

グローバル参照は、グローバルまたはプロセス・プライベート・グローバルのいずれかです。グローバルは、事前フェッチが無効になる時点では、定義の必要はありません。

このグローバルは @ 間接演算を使用して指定できます。“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

この引数には、[構造化システム変数名 \(SSVN\)](#) を指定することはできません。

## gref2

gref で範囲の指定に使用するグローバル参照。したがって、gref2 は gref と同じグローバル・ツリーの下位のグローバル・ノードである必要があります。

このグローバルは @ 間接演算を使用して指定できます。“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

## 例

以下の例は、事前フェッチを 2 つ確立し、その後それぞれを無効にします。

### ObjectScript

```
SET ret=$PREFETCHON(^a)
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET ret2=$PREFETCHON(^b)
IF ret2=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET retoff=$PREFETCHOFF(^a)
IF retoff=-1 { WRITE !,"prefetch turned off. Values:",ret }
ELSE { WRITE !,"prefetch not turned off" }
SET retoff2=$PREFETCHOFF(^b)
IF retoff2=-1 { WRITE !,"prefetch turned off. Values:",ret }
ELSE { WRITE !,"prefetch not turned off" }
```

以下の例は、事前フェッチを 2 つ確立し、その後現在のプロセスの事前フェッチをすべて無効にします。

### ObjectScript

```
SET ret=$PREFETCHON(^a)
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET ret2=$PREFETCHON(^b)
IF ret2=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET retoff=$PREFETCHOFF()
IF retoff=0 { WRITE !,"all prefetches turned off" }
ELSE { WRITE !,"prefetch not turned off" }
```

## 関連項目

- ・ [\\$PREFETCHON](#) 関数
- ・ [グローバル](#)

## \$PREFETCHON (ObjectScript)

指定されたグローバルの事前フェッチを確立します。

### 構文

```
$PREFETCHON(gref,gref2)
```

### 引数

引数	説明
<i>gref</i>	グローバル参照。
<i>gref2</i>	オプション - 範囲の指定に使用するグローバル参照。

### 説明

\$PREFETCHON は、現在のプロセスにおいてグローバルまたはグローバルの範囲に対して事前フェッチを有効にすることでパフォーマンスを高めます。\$PREFETCHON は、正常に終了する (事前フェッチが有効になる) と 1 を返します。\$PREFETCHON は、指定された事前フェッチを確立できなかった場合は 0 を返します。指定された範囲に 2 つの異なるグローバル名が含まれているか、事前フェッチを妨げる他の問題がある場合は、0 が返されることがあります。返り値 0 はエラーではありません。これによってプログラムの実行に割り込みが発生することではなく、指定された範囲内のグローバル参照の処理に障害が生じることもありません。それは、単に、それらのグローバル操作で事前フェッチによってパフォーマンスが向上することはないことを意味します。

注釈 グローバルの事前フェッチは、リモート・データベースではサポートされていません。

\$PREFETCHOFF は事前フェッチを無効にします。

\$PREFETCHON には、以下の 2 つの形式があります。

- ・ \$PREFETCHON(*gref*) は、*gref* ノードとそのすべての下位ノードを事前フェッチします。例えば、\$PREFETCHON(^abc(4)) は、^abc(4) のすべての下位ノードである、^abc(4,1)、^abc(4,2,2)などを事前フェッチします。この場合、^abc(5) は事前フェッチされません。
- ・ \$PREFETCHON(*gref*,*gref2*) は、*gref* から *gref2* までの範囲のノードを事前フェッチします。これには、*gref2* の下位ノードは含まれません。*gref* および *gref2* は同じグローバルのノードである必要があります。例えば、\$PREFETCHON(^abc(4),^abc(7,5)) は、^abc(4) から ^abc(7,5) の範囲のグローバル・ノードをすべて事前フェッチします。これには、^abc(4,2,2)、^abc(5)、および ^abc(7,1,2) が含まれます。ただし、この場合、^abc(7,5,1) は事前フェッチされません。

事前フェッチは、読み取りアクセスのみに制限されているわけではなく、多数の SET 操作が実行されている場合も正常に機能します。

### 引数

#### *gref*

グローバル参照は、グローバルまたはプロセス・プライベート・グローバルのいずれかです。グローバルは、事前フェッチを確立する時点では、定義の必要はありません。

このグローバルは @ 間接演算を使用して指定できます。“[間接\(@\)](#)” のリファレンス・ページを参照してください。

この引数には、[構造化システム変数名 \(SSVN\)](#) を指定することはできません。

## gref2

gref で範囲の指定に使用するグローバル参照。したがって、gref2 は gref と同じグローバル・ツリーの下位のグローバル・ノードである必要があります。

このグローバルは @ 間接演算を使用して指定できます。“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

## 事前フェッチとパフォーマンス

\$PREFETCHON を呼び出すと、1 つ以上の事前フェッチ・バックグラウンド・プロセス (デーモン) が必要に応じて開始されます。これらの事前フェッチ・デーモンは、すべてのプロセスによってシステム全体で共有されます。それぞれの事前フェッチ・デーモンは事前フェッチ要求を 1 つずつしか処理できないため、通常は、システム上で複数の事前フェッチ・デーモンを実行しておくことをお勧めします。ただし、多数の並列事前フェッチ・デーモンは、インタラクティブ・システム・アクセスに影響を与える可能性があります。

同じグローバル・ツリーのノードを含む多数のディスク・ブロックを読み取るアプリケーションを実行している場合、事前フェッチにより、パフォーマンスを向上させることができます。事前フェッチが最も効率的なのは、以下の場合です。

- ・ データに通常は昇順でアクセスする場合。つまり、グローバル・ツリーのデータ・ブロックに、通常は左から右へ順にアクセスする場合。ただし、必ずしも昇順でなければならないというわけではありません。事前フェッチが最適に動作するのは、グローバル・ツリーのデータ・ブロックに、通常は左から右へ順にアクセスする場合、または範囲内の少なくとも 1 つのデータ・ブロックに、論理ツリー内でその右横にあるほとんどのデータ・ブロックよりも先にアクセスする可能性が高い場合です。
- ・ 指定された範囲のほとんどのデータ・ブロックにアクセスする場合。ただし、ユーザが、データのごく一部にアクセスした後で操作をキャンセルすると、初期の事前フェッチで以降のフェッチと同じ数のブロックはフェッチされません。
- ・ 一度にアクティブにする事前フェッチの数が 100 未満の場合。

## 例

以下の例は、グローバル ^a に対する事前フェッチを確立します。

### ObjectScript

```
SET ^a="myglobal"
SET x=^a
SET ret=$PREFETCHON(^a)
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
SET ret=$PREFETCHOFF()
```

以下の例は、プロセス・プライベート・グローバル ^|a(1) から ^a|(50) の範囲に対する事前フェッチを確立します。

### ObjectScript

```
SET ret=$PREFETCHON(^|a(1),^|a(50))
IF ret=1 { WRITE !,"prefetch established" }
ELSE { WRITE !,"prefetch not established" }
```

## 関連項目

- ・ [\\$PREFETCHOFF 関数](#)
- ・ [グローバル](#)



# \$PROPERTY (ObjectScript)

インスタンスの特定プロパティへの参照をサポートします。

## 構文

```
$PROPERTY(instance,propertyname,index1,index2,index3... )
```

## 引数

引数	説明
instance	オプション - オブジェクト・インスタンス参照 (OREF) に評価される式。この表現の値は、目的のクラスのメモリにあるインスタンスの値を指定する必要があります。省略した場合、現在のオブジェクトが既定で使用されます。
propertyname	文字列として評価される式。この文字列の値は、instance で指定されたクラスで定義されている既存のプロパティ名と一致する必要があります。
index1, index2, index3, ...	オプション - propertyname が多次元値である場合、この一連の式はプロパティが表す配列へのインデックスとして扱われます (指定したプロパティが多次元でない場合、余分な引数があると実行時にエラーが発生します)。

## 説明

\$PROPERTY は、指定されたクラスにあるインスタンスで、プロパティの値を取得または設定します。この関数は、ObjectScript プログラムが、あるクラスの既存のインスタンス内の任意のプロパティの値を選択することを許可します。最初の引数はクラスのインスタンスでなければならないため、実行時に算出されます。プロパティ名は、実行時に算出することもできれば、文字列リテラルとして指定しておくこともできます。文字列の内容は、クラスで宣言されたプロパティの名前と正確に一致する必要があります。プロパティ名は、大文字と小文字を区別します。

プロパティが**多次元**であるとして宣言されている場合は、プロパティ名の後の引数は、多次元配列へのインデックスとして扱われます。最大で 255 の引数をインデックスに使用できます。

\$PROPERTY は割り当て式の左側にも記述できます。\$PROPERTY が割り当て演算子の左にある場合は、値の割り当て先の場所を指定します。右側にある場合は計算で使用する値であることを示します。

instance が有効な**メモリ内 OREF** ではない場合、<INVALID OREF> エラーが発生します。propertyname が有効なプロパティではない場合、<PROPERTY DOES NOT EXIST> エラーが発生します。index1 を指定し、かつ propertyname が多次元でない場合、<OBJECT DISPATCH> エラーが発生します。

多次元として宣言されていないプロパティから多次元値を取得したり、多次元ではないプロパティに多次元値を設定したりしようとすると、<FUNCTION> エラーが発生します。

## \$PROPERTY とメソッド

\$PROPERTY 関数は、渡されたプロパティの Get() メソッドまたは Set() メソッドを呼び出します。この関数は、“Instance.PropertyName” 構文と機能的には同じです。ここで、“Instance” および “PropertyName” は、この関数のシグニチャに記述する引数と同等です。このため、プロパティに Get() メソッドや Set() メソッドが存在する場合、その中では \$PROPERTY を呼び出さないでください。Get() および Set() メソッドの詳細は、“[プロパティ・メソッドの使用とオーバーライド](#)” を参照してください。

メソッド内で、現在のインスタンスのプロパティを参照するために \$PROPERTY を使用する場合は、instance を省略できます。ただし、普通は instance の後に続くコンマは、この場合も必要です。

## 例

以下の例は、現在の NLS 言語プロパティ値を返します。

### ObjectScript

```
SET nlsoref=##class(%SYS.NLS.Locale).%New()  
WRITE $PROPERTY(nlsoref,"Language")
```

以下は、\$PROPERTY を関数として使用する例です。

### ObjectScript

```
SET TestName = "%Library.File"  
SET ClassDef = ##class(%Library.ClassDefinition).%OpenId(TestName)  
FOR i = "Name", "Super", "Persistent", "Final"  
{  
    WRITE i, ": ", $PROPERTY(ClassDef, i), !  
}
```

以下は、割り当て演算子の右と左の両方で \$PROPERTY を使用する例です。

### ObjectScript

```
SET TestFile = ##class(%Library.File).%New("AFile")  
WRITE "Initial file name: ", $PROPERTY(TestFile, "Name"), !  
SET $PROPERTY(TestFile, "Name") =  
    $PROPERTY(TestFile, "Name") _ "Renamed"  
WRITE "File name afterward: ", $PROPERTY(TestFile, "Name"), !
```

以下の例は、現在のオブジェクト(この場合は SQL シェル)からプロパティ値を返します。\$PROPERTY は、最初の引数を省略して指定されています。

### Terminal

```
USER>DO $SYSTEM.SQL.Shell()  
SQL Command Line Shell  
-----  
  
The command prefix is currently set to: <<nothing>>.  
Enter <command>, 'q' to quit, '?' for help.  
[SQL]USER>>set path="a,b,c"  
  
path="a,b,c"  
[SQL]USER>>! WRITE "The schema search path is ", $PROPERTY(,"Path")  
The schema search path is "a,b,c"  
[SQL]USER>>
```

## 関連項目

- ・ [\\$CLASSMETHOD 関数](#)
- ・ [\\$CLASSNAME 関数](#)
- ・ [\\$METHOD 関数](#)
- ・ [\\$PARAMETER 関数](#)
- ・ [\\$THIS 特殊変数](#)

# \$QLENGTH (ObjectScript)

変数内の添え字レベルの数を返します。

## 構文

```
$QLENGTH(var)
$QL(var)
```

## 引数

引数	説明
var	変数名が含まれた文字列、またはこのような文字列に評価される式。この変数名では、添え字を指定しなくても、1 つ以上指定してもかまいません。

## 説明

\$QLENGTH は、var 内の添え字レベルの数を返します。\$QLENGTH は、var 内で指定された添え字レベルの数をカウントするだけです。\$QLENGTH が添え字レベルの数を返すために、var の変数が定義されている必要はありません。

## 引数

### var

変数を指定する、引用符で囲まれた文字列または文字列に評価される式。この変数は、ローカル変数、プロセス・プライベート・グローバル、またはグローバル変数のいずれかです。

文字列がグローバル参照の場合、var は、ネームスペース名を含めることで[拡張グローバル参照](#)を指定できます。var は引用符で囲まれた文字列であるため、リテラル引用符として正しく解析されるために、ネームスペース参照を囲む引用符は二重にする必要があります。例えば、`^|"SAMPLES"|myglobal(1,4,6)"` のようにします。"" という構文のプロセス・プライベート・グローバル内の引用符にも、同じ規則が適用されます。例えば、`^|"^"|ppgname(3,6)"` のようにします。\$QLENGTH は、指定されたネームスペースが存在するかどうかや、ユーザがそのネームスペースに対するアクセス特権を持っているかどうかを確認しません。

var では、変数名をキャノニック形式 (完全な拡張参照) で指定する必要があります。[ネイキッド・グローバル参照](#)、または間接参照と共に \$QLENGTH を使用するには、対応する完全な拡張参照を返すために \$NAME 関数を使用できます。

## 例

以下の例は、添え字付きグローバルと添え字なしのグローバルと共に使用する際の \$QLENGTH の結果を示しています。1 つ目の \$QLENGTH は、添え字のないグローバルを取り、0 を返します。2 つ目の \$QLENGTH は、2 つの添え字レベルのあるグローバルを取り、2 を返します。変数名を囲む引用符が二重になっていることに注目してください。これは、var が引用符で囲まれた文字列として指定されているためです。

### ObjectScript

```
WRITE !,$QLENGTH("^|"USER"|test")
; returns 0
SET name="^|"USER"|test(1,"customer")"
WRITE !,$QLENGTH(name)
; returns 2
```

以下の例は、3 つの添え字レベルを持つプロセス・プライベート・グローバルの \$QLENGTH の値を返します。\$ZREFERENCE 特殊変数には、最近参照された変数が含まれています。

## ObjectScript

```
SET ^||myppg("food","fruit",1)="apples"  
WRITE !,$QLENGTH($ZREFERENCE) ; returns 3
```

以下の例は、[ネイキッド・グローバル参照](#)として指定されるプロセス・プライベート・グローバルに対する \$QLENGTH 値を返します。\$NAME 関数は、ネイキッド・グローバル参照をキャノニック形式に拡張するために使用されます。

## ObjectScript

```
SET ^grocerylist("food","fruit",1)="apples"  
SET ^{2}="bananas"  
WRITE !,$QLENGTH($NAME(^{2})) ; returns 3
```

## 関連項目

- ・ [\\$QUERY](#) 関数
- ・ [\\$QSUBSCRIPT](#) 関数
- ・ [\\$NAME](#) 関数
- ・ [\\$ZREFERENCE](#) 特殊変数
- ・ [グローバルについての正式な規則](#)

# \$QSUBSCRIPT (ObjectScript)

変数名または添え字名を返します。

## 構文

```
$QSUBSCRIPT(namevalue,intexpr)
$QS(namevalue,intexpr)
```

## 引数

引数	説明
namevalue	文字列、もしくは文字列と評価される式で、添え字付きまたは添え字なしの、ローカル変数の名前、プロセス・プライベート・グローバル、またはグローバル変数を表します。
intexpr	返す名前を指定する整数コードです。変数名、添え字名またはネームスペース名のいずれかを返します。

## 説明

\$QSUBSCRIPT は、intexpr の値に応じて、変数名、または namevalue の指定した添え字名を返します。namevalue がグローバル変数の場合、明示的に指定されていれば、ネームスペース名を返すこともできます。\$QSUBSCRIPT は、既定のネームスペース名を返しません。

## 引数

### namevalue

引用符で囲まれた文字列または文字列に評価される式。ローカル参照またはグローバル参照です。Name(s<sub>1</sub>,s<sub>2</sub>,...,s<sub>n</sub>) という形式にできます。

文字列がグローバル参照のとき、ネームスペース参照を含むことができます。namevalue は引用符で囲まれた文字列であるため、リテラル引用符として正しく解析されるために、ネームスペース参照を囲む引用符を二重にする必要があります。

namevalue は、変数名をキャノニック形式 (完全な拡張参照) で参照する必要があります。[ネイキッド・グローバル参照](#)、または間接参照と共に \$QSUBSCRIPT を使用するには、対応する完全な拡張参照を返すために \$NAME 関数を使用できます。

### intexpr

返す値を指定する整数式コードです。namevalue 引数が形式 NAME(s<sub>1</sub>,s<sub>2</sub>,...,s<sub>n</sub>) を持つと仮定します。ここで n は最終添え字の順序数です。intexpr 引数は、下の値のうちいずれでも持つことが可能です。

Code	返回值
< -1	<FUNCTION> エラーを生成します。これらの数値は後の拡張子用に保存されます。
-1	グローバル変数 namevalue がネームスペース名を含む場合に、ネームスペース名を返します。それ以外では、NULL ("" ) 文字列を返します。
0	変数名を返します。グローバル変数に対して ^NAME を返し、プロセス・プライベート・グローバル変数に対して ^  NAME を返します。ネームスペース名を返しません。
<=n	整数 n によって指定される添え字のレベルに対する添え字名を返します。1 が最初の添え字レベルで、定義された添え字レベルのうち n が最高レベルとなります。
>n	整数 >n は、NULL 文字列 ("" ) を返します。ここで、n は定義された添え字の最高レベルです。

## 例

以下の例は、namevalue が 1 つの添え字レベルを持つ添字付きグローバルであり、指定されたネームスペースの場合、\$QSUBSCRIPT の値を返します。

### ObjectScript

```
SET global="^|"account"|%test("customer")"
WRITE !,$QSUBSCRIPT(global,-1) ; account
WRITE !,$QSUBSCRIPT(global,0) ; ^%test
WRITE !,$QSUBSCRIPT(global,1) ; customer
WRITE !,$QSUBSCRIPT(global,2) ; null string
```

以下の例は、namevalue が 2 つの添え字レベルを持つプロセス・プライベート・グローバルの場合、\$QSUBSCRIPT の値を返します。\$ZREFERENCE 特殊変数には、最近参照された変数が含まれています。

### ObjectScript

```
SET ^||myppg(1,3)="apples"
WRITE !,$QSUBSCRIPT($ZREFERENCE,-1) ; null string
WRITE !,$QSUBSCRIPT($ZREFERENCE,0) ; ^||myppg
WRITE !,$QSUBSCRIPT($ZREFERENCE,1) ; 1
WRITE !,$QSUBSCRIPT($ZREFERENCE,2) ; 3
```

以下の例は、[ネイキッド・グローバル参照](#)として指定されるグローバル変数に対する \$QSUBSCRIPT 値を返します。  
\$NAME 関数は、ネイキッド・グローバル参照をキャノニック形式に拡張するために使用されます。

### ObjectScript

```
SET ^grocerylist("food","fruit",1)="apples"
SET ^(2)="bananas"
WRITE !,$QSUBSCRIPT($NAME(^(2)),2) ; returns "fruit"
```

## 関連項目

- ・ [\\$QUERY](#) 関数
- ・ [\\$QLENGTH](#) 関数
- ・ [\\$NAME](#) 関数
- ・ [\\$ZREFERENCE](#) 特殊変数
- ・ [グローバルについての正式な規則](#)

# \$QUERY (ObjectScript)

ローカル配列とグローバル配列の物理的スキャンを行います。

## 構文

```
$QUERY(reference,direction,target)
$Q(reference,direction,target)
```

## 引数

引数	説明
reference	パブリック・ローカル変数名とグローバル変数名（およびオプションとしてその添え字）に評価される参照。単純なオブジェクト・プロパティを reference として指定することはできません。構文 obj.property を使用すると、多次元プロパティ参照を reference として指定できます。
direction	オプション - 配列を検索する方向。前方 = 1、後方 = -1。既定値は前方です。
target	オプション - \$QUERY の評価結果として返された reference の現在のデータ値を返します。例えば、reference が ^a(1) で、\$QUERY が ^a(2) を返す場合、target は値 ^a(2) になります。

## 概要

\$QUERY はパブリック・ローカル配列とグローバル配列を物理的にスキャンします。順番で、指定した配列ノードの次にある、定義されたノードの完全参照（名前と添え字）を返します。そのようなノードが存在しない場合、\$QUERY は NULL 文字列を返します。

## 引数

### reference

この引数は、パブリック変数またはグローバル変数に評価される必要があります。\$QUERY は、プライベート変数をスキャンできません。

この引数は**多次元オブジェクト・プロパティ**とすることができます。非多次元オブジェクト・プロパティとすることはできません。非多次元オブジェクト・プロパティで \$QUERY を使用しようとすると、<OBJECT DISPATCH> エラーが発生します。

返されるグローバル参照は、同じ下位レベルか、reference 引数で指定された上位レベルです。添え字を指定せずに reference を指定すると、\$QUERY は配列内で最初に定義されたノードを返します。

### direction

direction が指定されていない場合、既定の方向は前方向です。方向を指定したい場合は、引数値の 1 が配列を前方向に、引数値 -1 が配列を後方向に検索します。

### target

オプションで、target 変数を指定できます。target 変数を指定したい場合は、direction 引数を指定する必要があります。

\$QUERY の評価で返される値が NULL 文字列 ("") の場合、target 値は変更されません。

ZBREAK コマンドでは、target 引数をウォッチポイントとして指定することはできません。



## 例

以下の例は、ユーザ指定配列の全ノードのデータ値を出力するための汎用ルーチンを示しています。任意のレベル数の配列に対応できます。コードは、\$ORDER 関数の例にあるコードと同じ処理を実行します。23 行が必要になるのではなく、6 行のみが必要になります。処理できるレベル数についての制限はありません。

### ObjectScript

```
Start READ !,"Array name: ",ary QUIT:ary=""
SET queryary=$QUERY(@ary@(" "))
WRITE !,@queryary
FOR {
    SET queryary=$QUERY(@queryary)
    QUIT:queryary=""
    WRITE !,@queryary
}
WRITE !,"Finished."
QUIT
```

最初の SET コマンドは、添え字付き間接指定を持つ \$QUERY を使用して、データを含む最初の既存ノードにグローバル参照を初期化します。詳細は、“[間接 \(@\)](#)” を参照してください。(\$ORDER のように、\$QUERY は NULL 文字列を受け入れ、配列の最初の添え字を指定します)。最初の WRITE コマンドは、最初に見つかったノードの値を出力します。省略すると、最初のノードはスキップされます。

FOR ループでは、\$QUERY は、次のノードを検索して、グローバル参照を更新するのに使用されます。その後グローバル参照の内容は WRITE コマンドで出力します。後置条件付きの QUIT は、\$QUERY が配列の最後に到達したことを示す NULL 文字列(“”)を見つけると、ループを終了します。

ポインタ・ノード (\$DATA=11) と端末ノード (\$DATA=1) を見分ける場合以外には、\$DATA テストは必要ありません。

## 配列を検索する \$QUERY の使用法

\$QUERY を繰り返し使用すると、定義されているノードを順番に返ししながら、配列全体を左から右、上から下に検索できます。\$QUERY は reference で指定した添え字により決定されるポイントから開始できます。水平軸と垂直軸の両方を使用して処理します。以下はその例です。

### ObjectScript

```
SET exam=$QUERY(^client(4,1,2))
```

この例に基づいて、\$QUERY は 3 レベル配列を仮定し、以下の値のうちのいずれかを返す場合があります。

値	以下の場合に \$QUERY により返される
<code>^client(4,1,3)</code>	<code>^client(4,1,3)</code> が存在してデータを含む場合。
<code>^client(4,2)</code>	<code>^client(4,1,3)</code> またはそのデータは存在しないが、 <code>^client(4,2)</code> が存在してデータを含む場合。
<code>^client(5)</code>	<code>^client(4,1,3)</code> と <code>^client(4,2)</code> は存在しないか、またはデータを含まず、 <code>^client(5)</code> が存在してデータを含む場合。
NULL 文字列(“”)	以前のグローバル参照が一切存在せず、データも含まない場合。\$QUERY は配列の最後に到達しています。

\$QUERY は、direction 値 -1 を使用して、配列全体を逆方向に、右から左、下から上へと検索します。

## \$ORDER と比較しての \$QUERY

\$QUERY がグローバル参照全体を返すのに対し、[\\$ORDER](#) は次のノードの添え字のみを返すため、この 2 つの関数は異なります。\$ORDER は、1 つのレベルのノード全体で、水平軸のみに沿って進みます。

\$QUERY は、データを含む既存のノードのみを選択するという点でも、\$ORDER とは異なります。\$ORDER は、データを含むか否かに関係なく、既存のノードを選択します。\$ORDER は存在の暗黙のテストを行います (\$DATA'=0)、\$QUERY は存在とデータ両方の暗黙のテストを行います (\$DATA'=0 and \$DATA'=10)。しかし、\$QUERY は、データを含むポインタ・ノード (\$DATA=11) と終端ノード (\$DATA=1) を識別しません。この識別を行うには、コードに適切な \$DATA テストを含める必要があります。

\$ORDER と同様、\$QUERY は、通常、連続した整数の添え字を使用しない配列のノードを検索するために、ループ処理と共に使用されます。\$QUERY は、値を持つ次のノードのグローバル参照を返すだけです。\$QUERY は、グローバル配列をアクセスするための非常に密度の高いコードを提供しています。

\$NAME 関数と \$ORDER 関数同様、\$QUERY は、[ネイキッド・グローバル参照](#)と共に使用できます。これは配列名なしで指定され、直前に実行されたグローバル参照を指定します。以下はその例です。

### ObjectScript

```
SET a=^client(1)
SET x=2
SET z=$QUERY(^ (x))
```

最初の SET コマンドは、参照レベルを含む現在のグローバル参照を作成します。2 番目の SET コマンドは、添え字と共に使用する変数を設定しています。\$QUERY 関数は、[ネイキッド・グローバル参照](#)を使用して、^client(2) の次のノードのグローバル参照全体を返します。例えば返り値は、^client(2,1) あるいは ^client(3) になります。

## \$QUERY と \$ZREFERENCE

\$QUERY reference 引数がグローバル参照でない場合、\$ZREFERENCE は変更されません。

\$QUERY reference 引数がグローバル参照の場合：

- ・ \$QUERY がグローバル参照を返す場合、[\\$ZREFERENCE](#) はそのグローバル参照に設定されます。
- ・ \$QUERY が空の文字列を返す場合：
  - － \$QUERY reference 引数が添え字付きのグローバル参照の場合、\$ZREFERENCE は reference 引数に設定されます。ネイキッド参照の場合、reference 引数は拡張されます。
  - － \$QUERY reference 引数が添え字なしのグローバル参照の場合：
 

direction が前方向の場合、\$ZREFERENCE は、空文字列 "" である単一の添え字と共に reference 引数グローバルに設定されます。例えば、\$QUERY(a,1) は空文字列を返し、\$ZREFERENCE を ^a("") に設定します。

direction が後方向の場合、\$ZREFERENCE は変更されません。

## \$QUERY と拡張グローバル参照

%SYSTEM.Process クラスの RefInKind() メソッドを使用して、\$QUERY が拡張グローバル参照形式でグローバル参照を返すかどうかを、プロセスごとに制御できます。システム全体の既定の動作は、Config.Miscellaneous クラスの RefInKind プロパティで設定できます。

拡張グローバル参照の詳細は、"[拡張グローバル参照](#)" を参照してください。

## 関連項目

- ・ [\\$DATA](#) 関数
- ・ [\\$NAME](#) 関数
- ・ [\\$ORDER](#) 関数
- ・ [\\$QLength](#) 関数
- ・ [\\$QSubscript](#) 関数

- ・ [\\$ZREFERENCE](#) 特殊変数

# \$RANDOM (ObjectScript)

指定した範囲での擬似ランダム整数値を返します。

## 構文

```
$RANDOM(range)
$R(range)
```

## 引数

引数	説明
<i>range</i>	有効な乱数の範囲の上限を指定するために使用されるゼロ以外の正の整数。

## 概要

\$RANDOM は、0 から *range* -1 (包含的) の間の擬似ランダム・データ整数値を返します。したがって、\$RANDOM(3) は 0、1、2 を返します。ただし、3 は返しません。返される数は、指定した範囲で一様に分散されます。

\$RANDOM は、ほとんどの目的において十分にランダムです。厳密にランダムな値を必要とするアプリケーションでは、%SYSTEM.Encryption クラスの GenCryptRand() メソッドを使用する必要があります。

## 引数

### range

この値は、有効な乱数の範囲の上限を指定します。乱数の最大値は *range* マイナス 1 になります。*range* の値には、ゼロ以外の正の整数、整数変数名、またはゼロ以外の正の整数に評価される任意の有効な ObjectScript 式を指定できます。*range* の最大値は、1E17 (100000000000000000) です。この最大値を超える値を指定すると、<FUNCTION> エラーが発生します。\$RANDOM(1) は有効ですが、常に 0 を返します。\$RANDOM(0) は <FUNCTION> エラーになります。

## 例

以下の例は、0 から 24 まで (包含的) の乱数を返します。

### ObjectScript

```
WRITE $RANDOM(25)
```

小数点部分の乱数を返すには、連結演算子 (.)、または加算演算子 (+) を以下の例のように使用することができます。

### ObjectScript

```
SET x=$RANDOM(10)_$RANDOM(10)/10
WRITE !,x
SET y=$RANDOM(10)+($RANDOM(10)/10)
WRITE !,y
```

このプログラムは、小数点部分が 1 桁の .0 から 9.9 (包含的) までの数を返します。いずれかの演算子を使用して、InterSystems IRIS は先頭と末尾のゼロ (および小数部分がゼロの場合は小数点) を削除します。ただし、両方の \$RANDOM 関数がゼロ (0 や .0) を返す場合、InterSystems IRIS は、ゼロ (0) を返します。

以下は、2 つのさいころを振る場合を想定した例です。

## ObjectScript

```
Dice
  FOR {
    READ "Roll dice? ",reply#1
    IF "Yy"[reply,reply']=" {
      WRITE !,"Pair of dice: "
      WRITE $RANDOM(6)+1,"+", $RANDOM(6)+1,! }
    ELSE { QUIT }
  }
```

## \$REPLACE (ObjectScript)

入力文字列の文字列単位の部分文字列の置換で構成される新しい文字列を返します。

### 構文

```
$REPLACE(string,searchstr,replacestr,start,count,case)
```

### 引数

引数	説明
string	ソース文字列。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。string が空文字列 ("" ) の場合、\$REPLACE は空文字列を返します。
searchstr	string 内で検索する部分文字列。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。searchstr が空文字列 ("" ) の場合、\$REPLACE は string を返します。
replacestr	string の searchstr のインスタンスと置換される置換部分文字列。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。replacestr が空文字列 ("" ) の場合、\$REPLACE は searchstr の出現箇所を削除して string を返します。
start	オプション - 部分文字列の検索を開始する string 内の文字カウント位置。文字列は 1 からカウントされます。0、負の数字、非数値文字列または空の文字列は、1 と同じです。省略した場合、1 と見なされます。start > 1 の場合、部分文字列 (存在する場合) の置換を実行して、その文字で始まる string の部分文字列が返されます。start > \$LENGTH(string) の場合、\$REPLACE は空の文字列 ("" ) を返します。
count	オプション - 部分文字列を置換する回数。省略した場合、既定値は -1 です。これは、可能なすべての置換を実行します。0、-1 以外の負の数字、非数値文字列または空の文字列は、置換を実行しないことを意味する 0 と同じです。start が指定されている場合、count は、start の位置から部分文字列の置換を開始します。
case	オプション - string の searchstr のマッチングで大文字と小文字を区別するかどうかを示すブーリアン・フラグ。0 は大文字と小文字を区別します (既定値)。1 は大文字と小文字を区別しません。ゼロ以外の数は、1 と同様です。数値以外の値は、0 と同様です。プレースホルダのコンマは、start または count が指定されていない場合に指定できます。

### 説明

\$REPLACE 関数は、入力文字列の文字列単位の置換で構成される新しい文字列を返します。string で searchstr 部分文字列を検索します。\$REPLACE が一致する値を 1 つ以上見つけると、searchstr 部分文字列を replacestr に置換し、結果の文字列を返します。replacestr 引数値は、searchstr より長い値も短い値も取ることができます。searchstr 部分文字列を削除するには、replacestr に空文字列 ("" ) を指定します。

既定では、\$REPLACE は string の先頭から開始して、searchstr の各インスタンスを置換します。オプションの start 引数を使用して、文字列の中の指定された文字カウント位置で比較を開始できます。返される文字列は start の位置から始まり、それ以降の searchstr の各インスタンスを置換する string の部分文字列です。

オプションの count 引数を使用すると、一致する部分文字列の指定された数のみを置換できます。

既定では、\$REPLACE 部分文字列マッチングは大文字と小文字が区別されます。オプションの case 引数を使用して、大文字と小文字が区別されないマッチングを指定することができます。

注釈 \$REPLACE を使用すると文字列の長さが変わる場合があるため、ObjectScript %List や %List オブジェクト・プロパティなどのエンコードされた文字列値で \$REPLACE を使用しないでください。

## \$REPLACE、\$CHANGE、および \$TRANSLATE

\$REPLACE と \$CHANGE は文字列と文字列をマッチングして置換します。これらは、1 つ以上の文字の単一の指定された部分文字列を別の任意の長さの部分文字列に置換できます。\$TRANSLATE は文字と文字をマッチングして置換します。\$TRANSLATE は、複数の指定された単一文字を対応する単一文字に置換できます。この 3 つの関数はすべて、一致する文字または部分文字列を削除 (NULL に置換) します。

\$CHANGE では、常に大文字と小文字が区別されます。\$REPLACE のマッチングでは既定で大文字と小文字が区別されますが、大文字と小文字を区別せずに呼び出すこともできます。\$TRANSLATE のマッチングでは、常に大文字と小文字が区別されます。

\$REPLACE と \$CHANGE では、マッチングの開始ポイントおよび/または実行する置換の数を指定できます。\$REPLACE と \$CHANGE では、開始ポイントの定義方法が異なります。\$TRANSLATE では常にソース文字列の一致対象がすべて置換されます。

## 例

以下の例は、\$REPLACE の 2 つの使用法を示しています。最初の \$REPLACE では、入力文字列値は変更されません。2 番目の \$REPLACE では、関数の返り値と同じ値に設定することで、入力文字列値が変更されます。

### ObjectScript

```
SET str="The quick brown fox"
// creates a new string, does not change str value
SET newstr=$REPLACE(str,"brown","red")
WRITE "source string: ",str,!, "new string: ",newstr,!!
// creates a new string and replaces str with new string value
SET str=$REPLACE(str,"brown","silver")
WRITE "revised string: ",str
```

以下の例では、\$REPLACE を呼び出して、部分文字列のすべてのインスタンスのマッチングと置換、部分文字列の最初の 2 つのインスタンスのマッチングと置換を実行します。

### ObjectScript

```
SET str="1110/1110/1100/1110"
WRITE !,"before conversion ",str
SET newall=$REPLACE(str,"111","AAA")
WRITE !,"after replacement ",newall
SET newsome=$REPLACE(str,"111","AAA",1,2)
WRITE !,"after replacement ",newsome
```

以下の例では、\$REPLACE を呼び出して、大文字と小文字を区別するマッチングと大文字と小文字を区別しないマッチングを行い、文字列のすべての出現箇所を置換します。

### ObjectScript

```
SET str="Yes/yes/Y/YES/Yes"
WRITE !,"before conversion ",str
SET case=$REPLACE(str,"Yes","NO")
WRITE !,"after replacement ",case
SET nocase=$REPLACE(str,"Yes","NO",1,-1,1)
WRITE !,"after replacement ",nocase
```

以下の例では、\$REPLACE 関数と \$TRANSLATE 関数を比較します。



## ObjectScript

```
SET str="A mom, o plom, o comal, Pomama"  
WRITE !,"before conversion ",str  
SET s4s=$REPLACE(str,"om","an")  
WRITE !,"after replacement ",s4s  
SET c4c=$TRANSLATE(str,"om","an")  
WRITE !,"after translation ",c4c
```

\$REPLACE は "A man, o plan, o canal, Panama" を返します。

\$TRANSLATE は "A nan, a plan, a canal, Panana" を返します。

以下の例で、\$REPLACE の 4 引数形式は、開始ポイントから始まる文字列の一部のみを、文字列と文字列の置換を行って返します。

## ObjectScript

```
SET str="A mon, a plon, a conal, Ponama"  
WRITE !,"before start replacement ",str  
SET newstr=$REPLACE(str,"on","an",8)  
WRITE !,"after start replacement ",newstr
```

\$REPLACE は "a plan, a canal, Panama" を返します。

## 関連項目

- ・ [\\$CHANGE](#) 関数
- ・ [\\$TRANSLATE](#) 関数
- ・ [\\$EXTRACT](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$REVERSE](#) 関数
- ・ [\\$ZCONVERT](#) 関数

## \$REVERSE (ObjectScript)

文字列にある文字を逆の順に返します。

### 構文

```
$REVERSE(string)  
$RE(string)
```

### 引数

引数	説明
string	文字列または文字列に評価する式

### 概要

\$REVERSE は、string 内の文字を逆の順序で返します。string には 8 ビット文字または 16 ビット Unicode 文字を使用できます。InterSystems IRIS Unicode サポートの詳細は、“[Unicode](#)” を参照してください。

### サロゲート・ペア

\$REVERSE は、サロゲート・ペアを認識しません。サロゲート・ペアは、一部の中国語の文字を表示したり、日本語の JIS2004 標準をサポートするために使用されます。\$WISWIDE 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。\$WREVERSE 関数は、サロゲート・ペアを認識して、正しく解析します。\$REVERSE と \$WREVERSE は、それ以外は同一です。ただし、\$REVERSE は通常 \$WREVERSE より高速なため、サロゲート・ペアが出現しない場合は常に \$REVERSE が推奨されます。

### 例

以下の WRITE コマンドは、\$REVERSE からの返り値を示します。最初のコマンドは “CBA” を返し、2 つ目のコマンドは 321 を返します。

#### ObjectScript

```
WRITE !,$REVERSE("ABC")  
WRITE !,$REVERSE(123)
```

\$REVERSE 関数を他の関数と一緒に使用して、文字列の後方から検索を行うこともできます。以下の例では、\$FIND 関数と \$LENGTH 関数と共に \$REVERSE を使用して、テキスト行内で最後に出現する検索対象の文字列の位置を見つける方法を示しています。その文字列の位置として 33 が返されます。

#### ObjectScript

```
SET line="THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG."  
SET position=$LENGTH(line)+2-$FIND($REVERSE(line),$REVERSE("THE"))  
WRITE "The last THE in the line begins at ",position
```

### 関連項目

- ・ [\\$FIND](#) 関数
- ・ [\\$EXTRACT](#) 関数
- ・ [\\$LENGTH](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$WISWIDE](#) 関数

- ・ [\\$WLENGTH](#) 関数
- ・ [\\$WREVERSE](#) 関数

## \$SCONVERT (ObjectScript)

バイナリでエンコードされた値を数値に変換します。

### 構文

```
$SCONVERT(s,format,endian,position)
$SC(s,format,endian,position)
```

### 引数

引数	説明
<i>s</i>	数値をコンコードする 8 ビット・バイトの文字列。選択した <i>format</i> により、有効な値に対して制限が課せられます。
<i>format</i>	S1、S2、S4、S8、U1、U2、U4、F4、または F8 のいずれかの形式コード。引用符で囲んだ文字列で指定します。
<i>endian</i>	オプション — ブーリアン値。0 = リトル・エンディアン、1 = ビッグ・エンディアン。既定値は 0 です。
<i>position</i>	オプション — 8 ビット・バイトの文字列内の変換を開始する文字位置。文字位置は 1 からカウントします。既定は 1 です。 <i>position</i> を指定する場合、 <i>endian</i> またはプレースホルダのコンマも指定する必要があります。

### 説明

\$SCONVERT は、指定された *format* を使用して、8 ビット・バイトのエンコードされた文字列の *s* を数値に変換します。以下の *format* コードがサポートされます。

コード	概要
S1	1 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -128 ~ 127 です。
S2	2 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -32768 ~ 32767 です。
S4	4 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -2147483648 ~ 2147483647 です。
S8	8 つの 8 ビット・バイトの文字列にエンコードされた符号付整数。値の範囲は -9223372036854775808 ~ 9223372036854775807 です。
U1	1 つの 8 ビット・バイトの文字列にエンコードされた符号なし整数。最大値は 256 です。
U2	2 つの 8 ビット・バイトの文字列にエンコードされた符号なし整数。最大値は 65535 です。
U4	4 つの 8 ビット・バイトの文字列にエンコードされた符号なし整数。最大値は 4294967295 です。
F4	4 つの 8 ビット・バイトの文字列にエンコードされた IEEE 浮動小数点数。
F8	8 つの 8 ビット・バイトの文字列にエンコードされた IEEE 浮動小数点数。

文字列 *s* には、*format* コードで必要とされる 8 ビット・バイトの数に対応するように、指定された文字 *position* で始まりそれに続く分の十分な文字数を指定する必要があります。例えば、\$SCONVERT(*s*, "S4", 0, 9) の場合、デコードされた結果が文字位置 9、10、11、12 から返されるので、*s* の長さは少なくとも 12 文字にする必要があります。この範囲を超える値を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

\$SConvert は、8 ビット・バイト文字列での使用のみを想定しています。

引数 s が数値である場合、デコードされる前に、その[キャノニック数値形式](#)を含む文字列に変換されます。

IsBigEndian() クラス・メソッドを使用して、オペレーティング・システム・プラットフォームでどのビット順序を使用するかを決定できます (1 = ビッグ・エンディアン・ビット順、0 = リトル・エンディアン・ビット順)。

### ObjectScript

```
WRITE $SYSTEM.Version.IsBigEndian()
```

\$SConvert は \$NConvert の逆の操作を行います。

## 例

以下の例では、\$SConvert はバイナリで 2 バイトにエンコードされた値を数値に変換します。

### ObjectScript

```
SET x=$NConvert(258,"U2")
ZZDUMP x
SET y=$SConvert(x,"U2")
WRITE !,y
```

以下の例では、\$SConvert はバイナリで 2 バイトにエンコードされた値をビッグ・エンディアン順に数値に変換します。

### ObjectScript

```
SET x=$NConvert(258,"U2",1)
ZZDUMP x
SET y=$SConvert(x,"U2",1)
WRITE !,y
```

## 関連項目

- ・ [\\$NConvert](#) 関数

## \$SELECT (ObjectScript)

最初の true の式に関連した値を返します。

### 構文

```
$SELECT(expression:value,...)
$S(expression:value,...)
```

### 引数

引数	説明
expression	関連付けられた value 引数に対する選択テストです。
value	関連付けられた expression が true と評価された場合に返される値です。

### 概要

\$SELECT 関数は、最初に true (1) に評価された expression に関連する value を返します。各 \$SELECT 引数は、コロンで区切られた組み合わせの式です。左側は、1 (true) または 0 (false) に評価されるブーリアン expression です。右側は、返される value です。value には、任意の式を指定できます。任意の数の expression:value のペアをコンマで区切って指定できます。

以下の例では、最初の 3 つの式の真理値がテストされます。いずれも True に評価されない場合は、最後の式 (常に True に評価されます) がその値を返します。

#### ObjectScript

```
WRITE $SELECT(x=1:"1st is True",x=2:"2nd is True",x=3:"3rd is True",1:"The Default")
```

\$SELECT は、左から右へ expression を評価します。\$SELECT が true の値 (1) の 真偽値式を見つけると、コロンの右側に対応した式を返します。\$SELECT は、最左端の true の 真偽値式を見つけると、評価を停止します。引数リストの後のペアを評価することはありません。

\$SELECT 関数を入れ子にすることにより、複雑なロジックを構成できます。評価されるすべての真理条件と同じように、入れ子にした \$SELECT に [NOT 論理演算子](#) (!) を適用できます。

### 引数

#### expression

関連付けられた value 引数に対する選択テストです。InterSystems IRIS の有効な関係式あるいは論理式にできます。true と評価される expression がないときは、システムは <SELECT> エラーを生成します。実行中のルーチンがエラーによって中断されることを防止するために、最後の expression を、常に true に評価される値 1 にできます。

expression が文字列または数値の場合、0 以外の数値はすべて true に評価されます。数値 0 や非数値文字列は false に評価されます。

#### value

関連付けられた expression が true と評価された場合に返される値です。数値、文字列リテラル、変数名、その他有効な ObjectScript 式を使用できます。value に式を指定すると、関連付けられた expression が true に評価された後にのみ評価されます。value が添え字付きグローバル参照を含む場合は、評価されるときにネイキッド・インジケータを変更します。このため、\$SELECT 関数内、あるいは直後にネイキッド・グローバル参照を使用するときは注意してください。ネイキッド・インジケータに関する詳細は、"[ネイキッド・グローバル参照](#)" を参照してください。

## 例

〈SELECT〉エラーの発生を防止するためにも、適切な既定値を持つ最後の式として、常に値 1 を含める必要があります。詳細は、以下の例を参照してください。

### ObjectScript

```
Start
  READ !,"Which level?: ",a
  QUIT:a=""
  SET x=$SELECT(a=1:"Level1",a=2:"Level2",a=3:"Level3",1:"Start")
  DO @x
Level1()
  WRITE !,"This is Level 1"
Level2()
  WRITE !,"This is Level 2"
Level3()
  WRITE !,"This is Level 3"
```

ユーザが 1、2、3、または NULL 文字列以外の値を入力すると、制御はルーチンの先頭に渡されます。

\$SELECT を使用して、複数の IF 節を置き換えることができます。以下の例は、IF、ELSEIF、そして ELSE 節を使用して、数字が偶数か奇数かの判断をします。

### ObjectScript

```
OddEven()
  READ !,"Enter an Integer: ",x
  QUIT:x=""
  WRITE !,"The input value is "
  IF 0=$ISVALIDNUM(x) { WRITE "not a number" }
  ELSEIF x=0 { WRITE "zero" }
  ELSEIF "=$NUMBER(x,"I") { WRITE "not an integer" }
  ELSEIF x#2=1 { WRITE "odd" }
  ELSE { WRITE "even" }
  DO OddEven
```

また、以下の例は数字を受け入れ、その数字が偶数か奇数かの判断をします。\$SELECT を使用して、前述の例にある IF コマンドを置き換えます。

### ObjectScript

```
OddEven()
  READ !,"Enter an Integer: ",x
  QUIT:x=""
  WRITE !,"The input value is "
  WRITE $SELECT(0=$ISVALIDNUM(x):"not a number",x=0:"zero",
    "=$NUMBER(x,"I"):"not an integer",x#2=1:"odd",1:"even")
  DO OddEven
```

## \$SELECT と \$CASE

\$SELECT と \$CASE はどちらも、一連の式に対して左から右にマッチング操作を実行して、最初の一致に関連する値を返します。\$SELECT は一連のブーリアン式をテストして、最初に true に評価された式に関連する値を返します。\$CASE はターゲットの値を一連の式とマッチングして、最初の一致に関連する値を返します。

## 関連項目

- [DO コマンド](#)
- [GOTO コマンド](#)
- [IF コマンド](#)
- [\\$CASE 関数](#)



## \$SEQUENCE (ObjectScript)

複数のプロセスで共有されているグローバル変数をインクリメントします。

### 構文

```
$SEQUENCE(gvar)
$SEQ(gvar)

SET $SEQUENCE(gvar)=value
SET $SEQ(gvar)=value
```

### 引数

引数	説明
gvar	値がインクリメントされる変数。通常、gvar は、添え字付きあるいは添え字なしのグローバル変数 (^gvar) です。変数を定義する必要はありません。gvar が定義されていない場合、または NULL 文字列 ("") に設定されている場合、\$SEQUENCE は、初期値 0 を持つものとしてそれを扱い、状況に応じてインクリメントし、値 1 を返します。  gvar に対してリテラル値を指定することはできません。単純なオブジェクト・プロパティ参照を gvar として指定することはできません。構文 obj.property を使用すると、多次元プロパティ参照を gvar として指定できます。
value	オプション - 整数または空の文字列として評価される式。SET \$SEQUENCE 構文で使われます。

### 説明

\$SEQUENCE と \$INCREMENT はどちらも、ローカル変数、グローバル変数、またはプロセス・プライベート・グローバルをインクリメントします。\$SEQUENCE は一般的に、グローバルで使用されます。

\$SEQUENCE は、同じグローバル変数について一意の (重複していない) 整数インデックスを複数のプロセスで素早く取得する手段を提供します。各プロセスについて、\$SEQUENCE は整数値のシーケンス (範囲) を割り当てます。\$SEQUENCE は、この割り当てられたシーケンスを使用して gvar に値を割り当て、この gvar の値を返します。その後の \$SEQUENCE の呼び出しにより、そのプロセスに対して割り当てられたシーケンス内の次の値にインクリメントします。割り当てられたシーケンス内のすべての整数値をプロセスが消費すると、\$SEQUENCE の次の呼び出しで新しい整数値のシーケンスが自動的に割り当てられます。\$SEQUENCE は、割り当てる整数値のシーケンスのサイズを自動的に決定します。また、シーケンスの割り当てごとに、割り当てられるシーケンスのサイズを個別に決定します。この割り当てられたシーケンスは単一の整数になる場合もあります。

\$SEQUENCE はアトミック処理として実行されます (そのため、LOCK コマンドの使用を必要としません)。

\$SEQUENCE は、複数のプロセスで同じグローバルを同時にインクリメントするときに使用されます。\$SEQUENCE は、同時プロセスそれぞれに gvar グローバルの値の固有の範囲を割り当てます。これにより、各プロセスは \$SEQUENCE を呼び出して、割り当てられた値の範囲から連続する値を割り当てます。

\$SEQUENCE がプロセスに割り当てるシーケンスのサイズは、内部のタイムスタンプにより左右されます。プロセスが 2 回目に \$SEQUENCE を呼び出したときに、InterSystems IRIS は前のタイムスタンプを現在の時刻と比較します。これらの \$SEQUENCE 呼び出し間の時間に応じて、InterSystems IRIS は単一のインクリメントまたはインクリメントの計算シーケンスのいずれかをプロセスに割り当てます。

- ・ 割り当てられたシーケンスが 1 の場合。\$SEQUENCE は、\$INCREMENT のように動作します。

- ・ 割り当てられたシーケンスが > 1 の場合。\$SEQUENCE は、このプロセスごとのインクリメントのシーケンスを使用します。各プロセスがそれぞれの割り当てられたシーケンスを使用し、その後、新しいシーケンスが各プロセスに割り当てられます。

例えば、プロセス A とプロセス B がどちらも同じグローバルをインクリメントします。各プロセスが最初にグローバルをインクリメントする際は、単一のインクリメントです。次に各プロセスがグローバルをインクリメントする際は、InterSystems IRIS が 2 つの \$SEQUENCE 処理を比較して、インクリメントのシーケンスを計算します (このシーケンスは 1 つの整数になる場合があります)。その後の \$SEQUENCE 処理は、これらのプロセスごとのシーケンスをすべて使用した後、インクリメントの再割り当てを行います。この結果、次のようなインクリメントになります。A1、B2 (クロックを設定する単一のインクリメント)、A3 (A1 と A3 を比較してプロセス A に 4、5、6、7 を割り当てる)、B8 (B2 と B8 を比較してプロセス B に 9、10、11 を割り当てる)。完全なインクリメントのシーケンスは、次のようになります。A1、B2、A3、A4、B8、A5、A6、B9、A7、B10、B11。

割り当てられたシーケンスの一部がプロセスで使用されない場合、残りの数は未使用となり、インクリメント・シーケンスが連続しなくなります。

グローバル変数での \$SEQUENCE の使用に関する詳細は、“[多次元ストレージの使用法\(グローバル\)](#)”を参照してください。

## 引数

### gvar

インクリメントされる整数を含む変数。この変数を定義する必要はありません。\$SEQUENCE の最初の呼び出しにより、未定義の変数が 0 として定義され、その値が 1 にインクリメントされます。gvar 値は、正または負の整数とする必要があります。

通常、gvar 引数は、添え字付きあるいは添え字なしの[グローバル変数](#) (gvar) です。グローバル変数には、[拡張グローバル参照](#)を含めることができます。添え字付きグローバル変数の場合は、[ネイキッド・グローバル参照](#)を使用して指定できます。

gvar 引数は、ローカル変数または[プロセス・プライベート・グローバル](#)のいずれかになります。ただし、\$SEQUENCE は複数のプロセスにわたる使用を意図したものであるため、ほとんどの場合、この使用は意味がありません。ローカル変数またはプロセス・プライベート・グローバルでの \$SEQUENCE の使用は、1 の数値インクリメントでの \$INCREMENT の使用と同義です。ロック、ジャーナリング、トランザクション・ロールバックに関する下記の \$SEQUENCE に関する制限事項は、ローカル変数およびプロセス・プライベート・グローバルには適用されません。ローカル変数またはプロセス・プライベート・グローバルでの \$SEQUENCE の使用では、\$INCREMENT と同じエラーが発生します。これは、次のセクションに記載されているグローバル変数での \$SEQUENCE のエラー動作とは異なります。

gvar 引数は、[多次元プロパティ](#)参照でもかまいません。例えば、\$SEQUENCE(..Count) のように指定します。非多次元オブジェクト・プロパティとすることはできません。非多次元オブジェクト・プロパティをインクリメントしようとすると、<OBJECT DISPATCH> エラーが発生します。

\$SEQUENCE は、[特殊変数](#)をインクリメントできません。これは、SET を使用して変更できるものでも同じです。特殊変数をインクリメントしようとすると、<SYNTAX> エラーが発生します。

### value

シーケンシャル・インクリメントの開始整数を指定するために、[SET \\$SEQUENCE](#) で使用します。この値には任意の式を指定できますが、空文字列または正か負の整数に評価される必要があります。非数値文字列 (「centimeters」など) は 0 として評価されます。混合数値文字列 (「6centimeters」など) は整数部分 (6) に評価されます。整数以外の数値 (「6.5centimeters」など) に評価される文字列では、<ILLEGAL VALUE> エラーが生成されます。

## SET \$SEQUENCE

SET \$SEQUENCE を使用することで、\$SEQUENCE グローバルを削除したり、リセットしたりできます。SET \$SEQUENCE は、グローバル変数をリセットして、他のプロセスに割り当てられた整数のシーケンスの割り当てを解除します。

- ・ `SET $SEQUENCE(^gvar)=""` は、指定されたグローバル・ノードを削除し、キャッシュされた `$SEQUENCE` 数を持つすべてのジョブを通知して、それらの現在のインクリメント値を消去します。`$SEQUENCE` の最初の呼び出しは 1 にインクリメントされます。`SET $SEQUENCE(^gvar)=""` は、指定されたグローバル・ノードのみを削除します。そのノードの下位ノード (存在する場合) は削除しません。
- ・ `SET $SEQUENCE(^gvar)=n` ( $n$  は整数) は、指定されたグローバル・ノードを  $n$  にリセットして、キャッシュされた `$SEQUENCE` 数を持つすべてのジョブを通知して、それらの現在のインクリメント値を消去します。すべてのジョブにおける以降の `$SEQUENCE` の呼び出しでは、新しいインクリメント開始値  $n$  が使用されます。

既定では、`$SEQUENCE` は正の整数 (1 から開始) を割り当てます。ただし、`$SEQUENCE` は負の整数に設定することもできます。負の整数はゼロに向かってインクリメントされます。`$SEQUENCE` が負の整数に設定された場合、その後の呼び出しはインクリメントとしてゼロを割り当てて場合があります。

`SET $SEQUENCE` で `^gvar` を整数以外の値に設定しようとする <ILLEGAL VALUE> エラーが生成されます。`SET $SEQUENCE` が `^gvar` を数値以外の文字列に設定しようとする、`^gvar` は 0 に設定されます。

`KILL ^gvar` や `SET ^gvar` を使用して、`$SEQUENCE` グローバルの削除やリセットを行うことはできません。これらのコマンドは、プロセスに割り当てられた整数のシーケンスの割り当て解除を行わないためです。

## 非常に大きな数値のインクリメント

`$SEQUENCE` によって返される整数は、-9223372036854775807 から 9223372036854775806 ( $-2^{63}+1$  から  $2^{63}-2$ ) の範囲内にある数値です。この範囲外の整数を使用してグローバル変数での `SET $SEQUENCE` を試行すると、<ILLEGAL VALUE> エラーが生成されます。

以下の例では、グローバル変数での `$SEQUENCE` を 9.223372036854775800E18 に設定できますが、範囲制限を超えてこの数値をインクリメントすると <MAXINCREMENT> エラーが生成されます。この例を繰り返し実行することで、“低速インクリメント”と“高速インクリメント”を実行できます。この例の“高速インクリメント”では、`$SEQUENCE` が範囲の上限を超えて数のシーケンスを割り当てようとするため、実際に範囲の上限までインクリメントする前に <MAXINCREMENT> が出される場合があります。

### ObjectScript

```
TRY {
  SET rand=$RANDOM(2)
  SET $SEQUENCE(^bignum)=9.223372036854775800E18
  IF rand=0 { WRITE "slow increments:",!
    FOR x=1:1:10 {WRITE $SEQUENCE(^bignum)," increment #",x,!
      HANG .5 }
  }
  IF rand=1 {WRITE "fast increments:",!
    FOR i=1:1:10 {WRITE $SEQUENCE(^bignum)," increment #",i,!}
  }
}
CATCH exp { WRITE !,"In the CATCH block",!
  IF 1=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception",!
    WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
    WRITE "Location: ",exp.Location,!
  }
  ELSE { WRITE "unknown error",! }
}
```

これらの種類のエラーは、グローバル変数をインクリメントする場合にのみ発生します。

## \$SEQUENCE または \$INCREMENT

`$SEQUENCE` と `$INCREMENT` のどちらを使用しても、複数のプロセスから同じグローバル変数に固有の整数を割り当てることができますが、一般的には `$SEQUENCE` の方がパフォーマンスが優れています。`$SEQUENCE` がインデックスを割り当てる順序は、`$INCREMENT` の順序とは異なります。各プロセスに単一の整数インクリメントを割り当てる `$INCREMENT` の動作とは異なり、`$SEQUENCE` は、プロセスにインクリメントの連続的な範囲を割り当てることができます。これにより、プロセスの衝突と同期化を減らすことで、パフォーマンスを大幅に向上させることができます。連続的な

レコード ID はプロセスごとにグループ化されるため、レコードの挿入時におけるデータ・ブロック・パフォーマンスも向上させることができます。

\$SEQUENCE は、特に複数の同時プロセスが関わる整数インクリメント処理の用途で使用されます。\$INCREMENT は、より汎用的なインクリメント/デクリメント関数です。

- ・ \$SEQUENCE は、グローバル変数をインクリメントします。\$INCREMENT は、ローカル変数、グローバル変数、またはプロセス・プライベート・グローバルをインクリメントします。
- ・ \$SEQUENCE は、数値を 1 だけインクリメントします。\$INCREMENT は、任意の数値を任意の指定値単位でインクリメントまたはデクリメントします。
- ・ \$SEQUENCE は、プロセスにインクリメントの範囲を割り当てることができます。\$INCREMENT は、1 つのインクリメントのみを割り当てます。
- ・ SET \$SEQUENCE は、グローバルの変更または未定義化 (削除) の用途で使用できます。\$INCREMENT は、SET コマンドの左側で使用することはできません。

既定では、InterSystems IRIS SQL は [RowID の割り当て](#) を自動的に、複数のプロセスによるテーブルへの迅速な同時入力ができる \$SEQUENCE を使用して実行します。[\\$INCREMENT](#) を使用して RowID の割り当てを実行するように InterSystems IRIS SQL を構成できます。

## \$SEQUENCE とトランザクション処理

- ・ ロック : \$SEQUENCE は通常、新しいエントリをデータベースに追加する前に、カウンタをインクリメントするために使用します。\$SEQUENCE を使用すると、LOCK コマンドの使用をしなくても、この操作を迅速に行うことができます。gvar は、トランザクション中の 1 プロセスによってインクリメントされます。トランザクションの処理中に、並列トランザクション中の別のプロセスによってでもインクリメントできます。
- ・ ロールバック : \$SEQUENCE の呼び出しはジャーナルされません。したがって、トランザクションのロール・バックで gvar の値は変更されません。ロール・バックされたトランザクション中に \$SEQUENCE によって割り当てられた整数値は、以降の \$SEQUENCE の呼び出しによる割り当てで使用できません。

分散データベース環境での \$SEQUENCE の使用に関する詳細は、["アプリケーション・カウンタでの \\$Increment 関数の使用"](#) を参照してください。

## 関連項目

- ・ [\\$INCREMENT 関数](#)
- ・ [TROLLBACK コマンド](#)
- ・ [トランザクション処理での ObjectScript の使用法](#)

## \$SORTBEGIN (ObjectScript)

ソート・モードを開始し、グローバルに対する複数セットのパフォーマンスを向上させます。

### 構文

```
$SORTBEGIN(set_global)
```

### 引数

引数	説明
set_global	グローバル変数名

### 概要

\$SORTBEGIN は特別なソート・モードを開始します。このモードでは、指定したターゲット・グローバルに対する SET 操作がプロセス・プライベートの一時領域にリダイレクトされ、サブセットにソートされます。ターゲット・グローバル参照にデータをコピーする \$SORTEND を呼び出すと、このモードは終了します。特別なソート・モードの実行中は、すべてのターゲット・グローバル参照に対するセットと、その派生すべてが影響を受けます。

\$SORTBEGIN は、インデックスの構築などの大容量の不規則なデータをグローバルに書き込む必要がある場合などに実行パフォーマンスを助けるように設計されています。書き込まれたデータの量が利用可能なバッファ・プール・メモリに到達する、またはそれを超えたとき、パフォーマンスは非常に低下します。\$SORTBEGIN はこの問題を解決するため、ターゲット・グローバル内のデータの書き込みを規則的に行い、物理的なディスク・アクセスの必要回数を最小にすることを保証します。これは、1 つ以上の一時的バッファ (必要な場合は **IRISTEMP** データベース内のスペースを使用して) の中にデータを書き込んでソートし、\$SORTEND が呼び出されるときにターゲット・グローバルの中にデータを連続してコピーすることで実行します。

\$SORTBEGIN の実行中は、ターゲット・グローバルから読み取られるデータは SET 操作を反映しません。値を挿入しようとするグローバルと同じグローバルからグローバルの値を読み取る必要がある場合、\$SORTBEGIN を使用することはできません。

InterSystems IRIS オブジェクトと InterSystems SQL アプリケーションは、インデックスや一時的なインデックスの作成で \$SORTBEGIN を自動的に利用します。

\$SORTBEGIN ソート・モードは、オプションの 2 番目の引数が 0 に設定されている \$SORTEND を呼び出すことで、ターゲット・グローバルにデータを書き込まずに終了することができます。

成功した場合、\$SORTBEGIN は 0 ではない整数値を返します。失敗した場合、\$SORTBEGIN は 0 を返します。

### モード・エラーの並べ替え

\$SORTBEGIN と \$SORTEND の間で一部の操作を呼び出すと、InterSystems IRIS がエラー・コードを発行します。

- ・ set\_global のネームスペースのマッピングが \$SORTBEGIN と \$SORTEND で異なる場合、\$SORTEND を呼び出した際に <NAMESPACE> エラーが発生します。ただし、\$SORTBEGIN で暗黙的なネームスペースを使用して set\_global を指定した場合は、それ以降にネームスペースのマッピングが変更されても \$SORTEND には影響がありません。暗黙的なネームスペースのグローバル参照と明示的なネームスペースのグローバル参照を同じソート操作に混在させないでください。ネームスペースの変更の詳細は、“[ネームスペースの構成](#)” を参照してください。
- ・ \$SORTBEGIN グローバルを構築し、そのグローバルの祖先または子孫に対して \$SORTBEGIN を発行すると、InterSystems IRIS は <DUPLICATEARG> エラーを発行します。例えば、\$SORTBEGIN(^test(1,2,3)) を呼び出した場合は、\$SORTBEGIN(^test(1,2)) または \$SORTBEGIN(^test(1,2,3,4)) の関数呼び出しは <DUPLICATEARG> エラーとなります。

## 関連項目

- ・ [\\$SORTEND](#) 関数



## \$SORTEND (ObjectScript)

\$SORTBEGIN で開始されたソート・モードを終了します。

### 構文

```
$SORTEND(set_global,dosort)
```

### 引数

引数	説明
set_global	オプション - 対応する \$SORTBEGIN 内で指定されたグローバル変数。省略した場合、\$SORTEND は、現在のプロセスの \$SORTBEGIN 操作をすべて終了します。
dosort	オプション - フラグ引数。1 の場合、\$SORTBEGIN で開始されたソート処理を実行し、ソートされたデータを set_global 内にコピーします。0 の場合、データをコピーせずにソート処理を終了します。既定は 1 です。

### 概要

\$SORTEND は、特定のターゲット・グローバルに関して \$SORTBEGIN で開始した特別なソート・モードの終了を指定します。\$SORTENDset\_global の値は、対応する \$SORTBEGINset\_global と一致する必要があります。

set\_global を省略した場合、\$SORTEND は、現在のプロセスの、すべてのアクティブな \$SORTBEGIN 関数によって開始された現在のソート・モードをすべて終了します。したがって、\$SORTEND() または \$SORTEND(,1) は、プロセスの現在のソート・モードをすべて終了してコミットします。\$SORTEND(,0) は、プロセスの現在のソート・モードをすべて中止します。

- 成功した場合、\$SORTEND はグローバル・ノード・セットの総数を示す正の整数を返します。set\_global が指定された場合、これは、指定の set\_global 変数に適用されるセットの数です。set\_global が省略された場合、これは、現在のすべての \$SORTBEGINset\_global 変数に適用されるセットの数です。dosort フラグの設定に関係なく、整数が返されます。
- 成功しなかった場合、\$SORTEND は -1 を返します。例えば、\$SORTEND で、対応するアクティブな \$SORTBEGIN を持たない set\_global が指定された場合などです。
- 空命令の場合、\$SORTEND は 0 を返します。例えば、指定された set\_global 変数に適用されるセットがない場合、または set\_global を指定しないで \$SORTEND を発行したときに、アクティブな \$SORTBEGIN がない場合です。

set\_global のネームスペースのマッピングが \$SORTBEGIN と \$SORTEND で異なる場合、\$SORTEND を呼び出した際に <NAMESPACE> エラーが発生します。ただし、\$SORTBEGIN で暗黙的なネームスペースを使用して set\_global を指定した場合は、それ以降にネームスペースのマッピングが変更されても \$SORTEND には影響がありません。暗黙的なネームスペースのグローバル参照と明示的なネームスペースのグローバル参照を同じソート操作に混在させないでください。ネームスペースの変更の詳細は、“[ネームスペースの構成](#)” を参照してください。

### 例

以下の例は、グローバル ^mytest に 3 つのセットを適用します。\$SORTEND は 3 を返します。dosort は 1 であるため、\$DATA 関数の戻り値で示されているとおり、これらのセットが適用されます。



## ObjectScript

```

WRITE $SORTBEGIN(^myyestest),!
SET ^myyestest(1)="apple"
SET ^myyestest(2)="orange"
SET ^myyestest(3)="banana"
WRITE $SORTEND(^myyestest,1),!
WRITE $DATA(^myyestest(1)),!
WRITE $DATA(^myyestest(2)),!
WRITE $DATA(^myyestest(3))
KILL ^myyestest

```

以下の例は、グローバル ^mynotest に 3 つのセットを適用します。\$SORTEND は 3 を返します。dosort は 0 であるため、\$DATA 関数の返り値で示されているとおり、これらのセットは適用されません。

## ObjectScript

```

WRITE $SORTBEGIN(^mynotest),!
SET ^mynotest(1)="apple"
SET ^mynotest(2)="orange"
SET ^mynotest(3)="banana"
WRITE $SORTEND(^mynotest,0),!
WRITE $DATA(^mynotest(1)),!
WRITE $DATA(^mynotest(2)),!
WRITE $DATA(^mynotest(3))
KILL ^mynotest

```

以下の 2 つの例では、2 つの \$SORTBEGIN 操作を指定し、それらの操作でグローバル ^mytesta に 3 つの セットを適用して、グローバル ^mytestb に 2 つのセットを適用します。\$SORTEND は、set\_global が指定されていないため、現在の \$SORTBEGIN 操作をすべて終了して 5 を返します。最初の例ではこれらのセットがコミットされ、次の例では中止されていますが、どちらの例でも \$SORTEND は 5 を返します。

## ObjectScript

```

WRITE $SORTBEGIN(^mytesta),!
SET ^mytesta(1)="apple"
SET ^mytesta(2)="orange"
SET ^mytesta(3)="banana"
WRITE $SORTBEGIN(^mytestb),!
SET ^mytestb(1)="corn"
SET ^mytestb(2)="carrot"
WRITE "$SORTEND returns: ", $SORTEND(,1),!
WRITE "global sets committed?: ", $DATA(^mytesta(2))
KILL ^mytesta, ^mytestb

```

## ObjectScript

```

WRITE $SORTBEGIN(^mytesta),!
SET ^mytesta(1)="apple"
SET ^mytesta(2)="orange"
SET ^mytesta(3)="banana"
WRITE $SORTBEGIN(^mytestb),!
SET ^mytestb(1)="corn"
SET ^mytestb(2)="carrot"
WRITE "$SORTEND returns: ", $SORTEND(,0),!
WRITE "global sets committed?: ", $DATA(^mytesta(2))
KILL ^mytesta, ^mytestb

```

## 関連項目

- ・ [\\$SORTBEGIN 関数](#)

## \$STACK (ObjectScript)

プロセス・コール・スタックに保存されているアクティブ・コンテキストに関する情報を返す関数です。

### 構文

```
$STACK(context_level,code_string)
$ST(code_string)
```

### 引数

引数	説明
context_level	情報が要求されているコンテキストの 0 をベースにしたコンテキスト・レベル番号を指定する整数。サポートされている値は、0、正の整数、および -1 です。
code_string	オプション - 要求されているコンテキスト情報の種類を指定するキーワード文字列。サポートされている値は、“PLACE”、“MCODE”、および “ECODE” です。

### 概要

\$STACK 関数は、現在の実行スタックまたは現在のエラー・スタックのいずれかに関する情報を返します。これは、\$ECODE 特殊変数の値によって異なります。\$STACK は、現在の実行スタック (プロセス・コール・スタックとも呼ばれる) 情報を返すために通常使用されます。

ルーチンが DO コマンド、XECUTE コマンド、またはユーザ定義関数 (GOTO コマンドではありません) を呼び出すたびに、現在実行しているルーチンのコンテキストがコール・スタックに保存され、呼び出されたルーチンで新しく作成されたコンテキストで実行を開始します。呼び出されたルーチンが順に別のルーチンを呼び出す、という具合に、より多くの保存コンテキストをコール・スタックに置くことができます。

\$STACK 関数は、プロセス・コール・スタックに保存されている、これらアクティブ・コンテキストの情報を返します。\$STACK は、現在実行中のコンテキストの情報も返すことが可能です。しかしエラー処理の間は、アプリケーションでエラーが発生したときに利用できる、全コンテキストのスナップショットを返します。

[\\$STACK](#) 特殊変数で、現在のコンテキスト・レベルを判別できます。

### \$ECODE および \$STACK

\$STACK によって返される値は、[\\$ECODE](#) 特殊変数に依存します。\$ECODE がクリアされている (NULL 文字列が設定されている) 場合、\$STACK は現在の実行スタックを返します。\$ECODE に NULL 以外の値が含まれている場合、\$STACK は現在のエラー・スタックを返します。

エラー・スタック・コンテキスト情報は、\$ECODE 特殊変数の値に NULL でない値が含まれている場合のみ利用可能です。このケースが該当するのは、エラーが発生した場合、または \$ECODE に明示的に NULL でない値が設定されている場合です。この場合、\$STACK は、指定したコンテキスト・レベルのアクティブ・スタック・コンテキストではなくエラー・スタック・コンテキストの情報を返します。

エラー・スタック・コンテキスト情報の利用が不可能 (\$ECODE="" ) な場合、2 個の引数の形式の \$STACK で現在のコンテキスト・レベルを指定すると、InterSystems IRIS は現在実行中のコマンド情報を返します。現在の実行スタックにアクセスしている間の動作の一貫性を確保するには、\$STACK を呼び出す前に SET \$ECODE="" を指定します。

エラー処理およびエラー処理スタックに関する詳細は、“[TRY-CATCH の使用法](#)” を参照してください。

### \$STACK の 1-引数形式

\$STACK(context\_level) は、指定したコンテキスト・レベルがどのように作成されたのかを示す文字列を返します。以下のテーブルは、返される文字列の値を示します。

戻り値	説明
DO	指定したコンテキストが DO コマンドによって作成されたときに返されます。
XECUTE	指定したコンテキストが XECUTE コマンドまたは BREAK コマンドによって作成されたときに返されます。
\$\$	指定したコンテキストがユーザ定義関数参照によって作成されたときに返されます。
ECODE 文字列	指定したコンテキストがエラー・スタックに追加される原因となったエラーのエラー・コード値。例：,M26,。エラーが発生済みのコンテキスト・レベルでエラーが発生するとき、コンテキスト情報は次に高いエラー・スタック・レベルに置かれます。指定したエラー・スタック・コンテキスト・レベルのコンテキスト情報が再配置された情報である場合にのみ、コンテキスト情報が返されます。

指定されたコンテキスト・レベルがゼロ (0)、または定義されていない場合、\$STACK は NULL 文字列を返します。

\$STACK 関数の 1 - 引数形式でコンテキスト・レベルに -1 を指定することも可能です。この場合、\$STACK は、通常のプロセスで、利用可能な、情報の最大コンテキスト・レベルを返します。情報は現在実行中のコンテキストのコンテキスト・レベル番号です。しかしエラー処理中は、\$STACK(-1) はいずれか大きい方を返します。

- ・ プロセス・エラー・スタックの最大コンテキスト・レベル
- ・ 現在実行中のコンテキストのコンテキスト・レベル番号

## \$STACK の 2 - 引数形式

\$STACK(context\_level,code\_string) は、指定した code\_string に従い、指定したコンテキスト・レベルの情報を返します。code\_string は引用符付きの文字列として指定する必要があります。code\_string 値は、大文字と小文字を区別しません。例えば、\$STACK(1,"PLACE") と \$STACK(1,"place") は同等です。

以下は、それぞれを指定したときに返されるコード文字列と情報を解説しています。

- ・ PLACE - 指定したコンテキスト・レベルで最後に実行されたエントリ参照とコマンド番号を返します。DO とユーザ定義の関数コンテキストに対して、"label[+offset][routine name] +command" の形式で値が返されます。XECUTE コンテキストの場合、"@ +command" の形式が使用されます。
- ・ MCODE - 指定されたコンテキスト・レベルで最後に実行されたコマンドを含む、ソース・ルーチン行、XECUTE 文字列、または \$ETRAP 文字列を返します(ルーチン行は、\$TEXT 関数の場合と同様に返されます)。

**注釈** エラー処理中は、エラー・スタックを作成あるいは更新しているときにメモリが少ないと、ソース行を格納できないことがあります。この場合、MCODE コード文字列の返り値は NULL 文字列です。しかし、PLACE コード文字列の返り値は位置を示します。

- ・ ECODE - 指定したコンテキスト・レベルで発生したエラーのエラー・コード (エラー・スタック・コンテキスト内でのみ利用可能)。

要求された情報が指定したコンテキスト・レベルでは利用不可能な場合、\$STACK の 2 - 引数形式は NULL 文字列を返します。

<STORE> エラーや低いメモリ状態の後、\$STACK の 2 - 引数形式のアプリケーションで通常利用できる情報が利用できないことがあります。

## 例

以下の例は、\$STACK が返すことができる情報を示しています。

## ObjectScript

```
STAC ;
    SET $ECODE=""
    XECUTE "DO First"
    QUIT
First SET varSecond=$$Second()
    QUIT
Second() FOR loop=0:1:$STACK(-1) {
    WRITE !,"Context level:",loop,?25,"Context type: ",$STACK(loop)
    WRITE !,?5,"Current place: ",$STACK(loop,"PLACE")
    WRITE !,?5,"Current source: ",$STACK(loop,"MCODE")
    WRITE ! }
    QUIT 1
```

このターミナルの例では、上述のルーチン呼び出します。

## Terminal

```
USER>DO ^STAC
Context level: 0      Context type:
Current place: @ +1
Current source: DO ^STAC
Context level: 1      Context type: DO
Current place: STAC+2^STAC +1
Current source: XECUTE "DO First"
Context level: 2      Context type: XECUTE
Current place: @ +1
Current source: DO First
Context level: 3      Context type: DO
Current place: First^STAC +1
Current source: First SET Second=$$Second
Context level: 4      Context type: $$
Current place: Second+2^STAC +4
Current source: WRITE !,?5,"Current source: ",$STACK(loop,"MCODE")
```

## マルチ - 引数コマンドを数える \$STACK

マルチ - 引数コマンドを指定すると、コマンド数にはコマンド・キーワードと最初からのすべてのコマンド引数が含まれます。以下のマルチ - 引数コマンドを考えてみましょう。

## ObjectScript

```
TEST
SET X=1,Y=Z
```

InterSystems IRIS では、\$STACK 文の \$STACK(1,"PLACE") は、Y=Z 引数が別のコマンドであると解釈されるため、"TEST^TEST +2" を返します。

## クロスネームスペース・ルーチン呼び出し

ルーチンが異なるネームスペースのルーチン呼び出す場合、\$STACK はルーチン名の一部としてネームスペース名を返します。例えば、USER ネームスペースのルーチンが SAMPLES ネームスペースのルーチン呼び出す場合、\$STACK は ^|"SAMPLES"|routineName を返します。

\$STACK は、区切り文字としてキャレット (^) 文字を使用します。そのため、**暗黙のネームスペース名**にキャレット (^) 文字が含まれている場合、InterSystems IRIS はこのネームスペース名文字を @ 文字として表示します。

## 関連項目

- [DO コマンド](#)
- [XECUTE コマンド](#)
- [\\$ECODE 特殊変数](#)
- [\\$ESTACK 特殊変数](#)
- [\\$STACK 特殊変数](#)

- ・ [TRY-CATCH の使用法](#)
- ・ [スタックを表示する %STACK の使用法](#)

## \$TEXT (ObjectScript)

指定した位置で見つかったソース・コード行を返します。

### 構文

```
$TEXT(label+offset^routine)
$T(label+offset^routine)

$TEXT(@expr_atom)
$T(@expr_atom)
```

### 引数

引数	説明
label	オプション - ルーチンの行ラベル。リテラル値である必要があります。変数は、label の指定には使用できません。行ラベルでは、大文字と小文字が区別されます。省略した場合、+offset はルーチンの先頭からカウントされます。
+offset	オプション - 行のオフセット数として返される行を特定する正の整数に評価される式。省略した場合、label で特定される行が返されます。
^routine	オプション - ルーチンの名前。リテラル値である必要があります。変数は、routine の指定には使用できません。( ^ 文字は区切り文字で、ルーチン名の一部ではないことに注意してください。)ルーチンが現在のネームスペースにない場合、 <a href="#">拡張ルーチン参照</a> を使用して、ルーチンを含むネームスペースを ^   "namespace"   routine のように指定できます。省略した場合、現在ロードされているルーチンが既定となります。
@expr_atom	位置を示す間接演算を使用する式アトム。label+offset^routine などの形式に解決されます。

### 概要

\$TEXT は、指定した位置で見つかったソース・コード行を返します。\$TEXT は、指定した位置でソース・コードを見つけれないとき、NULL 文字列を返します。

ソース・コードの単一行を特定するには、label か +offset の一方または両方を指定する必要があります。既定では、\$TEXT は[現在ロードされているルーチン](#)にアクセスします。\$TEXT は、現在実行されているルーチンでコード化されるか、[ZLOAD](#)を使用して最後にロードされた静的ルーチンとして、現在ロードされているルーチンにアクセスします。^routine を使用して、現在ロードされているルーチン以外にルーチンの場所を指定することができます。間接指定 (@expr\_atom) を使用して、場所を指定することができます。

\$TEXT は、ルーチンの INT コード・バージョンから、指定した行を返します。INT コードではプリプロセッサ文はカウントされず、含まれることもありません。INT コードにはすべてのラベルと[ほとんどのコメント](#)が含まれますが、ルーチンの MAC バージョンの完全な空白行はカウントされず、含まれることもありません。これは、ソース・コード内であっても、複数行コメント内であっても同じです。

+offset 引数は、ルーチンの INT コード・バージョンを使用して行をカウントします。INT バージョンに対応する行および行オフセットを正しくカウントするために、ルーチンの変更後、\$TEXT のルーチンをリコンパイルする必要があります。

返されたソース・コードでは、行内の最初の空白文字がタブの場合、\$TEXT はこれを単一の空白文字で置換します。それ以外のすべてのタブおよび空白文字は、変更されずに返されます。したがって、\$PIECE(\$TEXT(line), " ", 1) は常にラベルを返し、\$PIECE(\$TEXT(line), " ", 2, \*) はラベル以外のすべてのコードを返します。

ルーチンがオブジェクト・コードのみとして配信される場合、;; コメントは、オブジェクト・コードに保持される唯一のコメント・タイプです。したがって、それらのルーチンでは、\$TEXT で ;; コメントのみを使用可能です。\$TEXT により ;; コメントが返されるようにするには、コメントのみの行またはラベルと同じ行に記述する必要があります。コマンドを記述した行や

関数やサブルーチンを宣言している行には記述できません。各種 InterSystems IRIS コメントの詳細は、“[コメント](#)”を参照してください。

**PRINT** コマンドまたは **ZPRINT** コマンドを使用して、[現在ロードされているルーチン](#)からソース・コードの単一行（または複数行）を表示することができます。ZPRINT（または PRINT）で、[編集ポインタ](#)を、出力した行の末尾に設定します。**\$TEXT** では、編集ポインタは変更されません。

## 引数

### label

現在のルーチン内のラベル、または routine 引数も提供されている場合は、指定されたルーチン内のラベルです。引用符なしのリテラルとして指定する必要があります。[ラベル名](#)では大文字と小文字が区別され、Unicode 文字を含めることができます。ラベルは、31 文字よりも長くすることができますが、最初の 31 文字は一意である必要があります。**\$TEXT** は、指定された label の最初の 31 文字のみと一致します。

offset オプションを省略、または label+0 と指定すると、InterSystems IRIS はラベル行を印刷します。label+1 は、ラベルの後の行を出力します。label がルーチン内で見つからない場合、**\$TEXT** は空の文字列を返します。

### offset

行カウントを指定する正の整数、または正の整数として評価される式。先頭のプラス記号 (+) は必須となります。このパラメータを単独で指定した場合は、+offset は、ルーチンの先頭からの行カウント指定となります (+1 でルーチンの最初の行となります)。label 引数と共に指定した場合、行カウントはラベル位置から数えられます (+0 でラベル行自体、+1 でラベル後の行となります)。+offset がルーチンの行数（または label からルーチンの末尾までの行数）より大きい場合、**\$TEXT** は空の文字列を返します。

+0 のオフセットを指定できます。label が指定される場合、**\$TEXT(mylabel+0)** は **\$TEXT(mylabel)** と同じになります。**\$TEXT(+0)** を呼び出した場合、現在ロードされているルーチンの名前が返されます。

InterSystems IRIS は、+offset の値をキャノニック形式の正の整数に解決します。負の整数オフセット値の場合は、[<NOLINE>](#) エラーが発生します。

InterSystems IRIS では、数値および数値文字列を[キャノニック形式](#)に解決します。これには先頭のプラス記号を削除することも含まれます。このような理由から、**\$TEXT** 関数でプラス記号を指定することで、これをオフセットとして使用する必要があります。

変数をオフセットとして使用するには、以下に示すように、変数の前にプラス記号を付ける必要があります。

### ObjectScript

```
SET x=7
WRITE $TEXT(x)      /* because there is no plus sign, search for a label named x */
WRITE $TEXT(+x)     /* locates the offset +7 code line */
```

### routine

このパラメータだけが指定される場合は、ルーチン・コードの先頭行を示します。label 引数と共にのみ指定すると、ルーチン内の指定したラベルで見つかった行が返されます。offset 引数と共にのみ指定すると、ルーチン内の指定したオフセットにある行が返されます。label と offset 両方が提供されている場合、ルーチン内の指定したラベル内の指定したオフセットで見つかった行が返されます。

routine 引数は、引用符なしのリテラルとして指定する必要があります。変数は、routine 名の指定には使用できません。先頭のキャレット (^) は必須です。

既定では、InterSystems IRIS は、現在のネームスペース内のルーチンを検索します。目的のルーチンが別のネームスペースにある場合は、[拡張グローバル参照](#)を使用してそのネームスペースを指定できます。例えば、**\$TEXT(mylabel+2^| "SAMPLES" |myroutine)** のようにします。ここでは、垂直バーのみが使用できることに注意してください。角括弧は使用できません。^routine のネームスペース部分を変数として指定できます。



指定されたルーチンやネームスペースが存在しない場合や、ユーザがそのネームスペースに対するアクセス特権を持っていない場合、\$TEXT は空の文字列を返します。

### expression atom (@expr\_atom)

\$TEXT 引数 (label+offset^routine) に評価する間接引数です。詳細は、“[間接 \(@\)](#)” のリファレンス・ページを参照してください。

## 例

以下の 4 つの例は、オブジェクト・コードのみで保存されたルーチンを示しています。最初の 2 つの例は ; ; コメントを記述した参照先の行を返します。3 番目と 4 番目の例は、NULL 文字列を返します。

### ObjectScript

```
Start ; ; this comment is on a label line
WRITE $TEXT(Start)
```

### ObjectScript

```
Start
; ; this comment is on its own line
WRITE $TEXT(Start+1)
```

### ObjectScript

```
Start
SET x="fred" ; ; this comment is on a command line
WRITE $TEXT(Start+1)
```

### ObjectScript

```
MyFunc() ; ; this comment is on a function declaration line
WRITE $TEXT(MyFunc)
```

以下の例は、label の最初の 31 文字のみが指定のラベルと一致することを示しています。

### ObjectScript

```
StartabcdefghijklmnopqrstuvwxyzA ; ; 32-character label
WRITE $TEXT(StartabcdefghijklmnopqrstuvwxyzB)
```

以下の例は、現在のルーチンにある指定のラベルで見つかった行を返す \$TEXT(label) 形式を示しています。ラベルも返します。ユーザが “?” を入力すると、行ラベルに沿って情報テキストが出力され、初期プロンプトに制御が戻ります。

### ObjectScript

```
Start
READ !,"Array name or ? for Info: ",ary QUIT:ary=""
IF ary="?" {
    WRITE !,$TEXT(Info),!
    GOTO Start
}
ELSE { DO ArrayOutput(ary) }
QUIT
Info ; ; This routine outputs the first-level subscripts of a variable.
QUIT
ArrayOutput(val)
SET i=1
WHILE $DATA(@val@(i)) {
    WRITE !,"subscript ",i," is ",@val@(i)
    SET i=i+1
}
QUIT
```

以下の例は、現在のルーチン内になければならない、指定したラベル内のオフセットで見つかった行を返す \$TEXT(label+offset) 形式を示しています。offset が 0 のときは、ラベルを使用したラベル行が返されます。この例では、FOR ループを使用して複数行のテキストにアクセスし、ラベルまたは複数行のコメント区切り文字の表示を回避します。

## ObjectScript

```

Start
  READ !,"Array name or ? for Info: ",ary QUIT:ary=""
  IF ary="?" {
    DO Info
    GOTO Start }
  ELSE { DO ArrayOutput(ary) }
  QUIT
Info FOR loop=2:1:6 { WRITE !,$TEXT(Info+loop) }
/*
  This routine outputs the first-level subscripts of a variable.
  Specifically, it asks you to supply the name of the variable
  and then writes out the current values for each subscript
  node that contains data. It stops when it encounters a node
  that does not contain data.
*/
QUIT
ArrayOutput(val)
  SET i=1
  WHILE $DATA(@val@(i)) {
    WRITE !,"subscript ",i," is ",@val@(i)
    SET i=i+1
  }
  QUIT

```

以下の例は、[拡張ルーチン参照](#)を使用して、SAMPLES ネームスペース内のルーチンからコードの行にアクセスします。これは、myroutine という名前のルーチンの ErrorTest ラベルの後のコードの最初の行にアクセスします。これは、どのネームスペースからも実行できます。

## ObjectScript

```
WRITE $TEXT(ErrorTest+1^|"SAMPLES"|myroutine)
```

## 引数間接演算

\$TEXT 引数全体の間接演算は、行とルーチンの両方への間接参照を実行するための、便利な方法です。例えば、変数 ENTRYREF に行ラベルとルーチン名の両方が含まれているときは、行とルーチンを別々に参照するのではなく、変数

```
$TEXT(@ENTRYREF)
```

を参照できます。

```
$TEXT(@$PIECE(ENTRYREF,"^",1)^@$PIECE(ENTRYREF,"^",2))
```

## 関連項目

- ・ [PRINT コマンド](#)
- ・ [ZINSERT コマンド](#)
- ・ [ZLOAD コマンド](#)
- ・ [ZPRINT コマンド](#)
- ・ [ZREMOVE コマンド](#)
- ・ [ZSAVE コマンド](#)
- ・ [ZZPRINT コマンド](#)
- ・ [Comments](#)
- ・ [ラベル](#)
- ・ [間接 \(@\)](#)

## \$TRANSLATE (ObjectScript)

ソース文字列の文字列単位の置換で構成される新しい文字列を返します。

### 構文

```
$TRANSLATE(string,identifier,associator)
$TR(string,identifier,associator)
```

### 引数

引数	説明
string	ソース文字列。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。
identifier	検索文字。string で検索する 1 つ以上の文字で構成される文字列。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。
associator	オプション - 置換文字。identifier でそれぞれの文字に位置的に対応する 1 つ以上の置換文字で構成される文字列。数値、文字列リテラル、変数名等、任意の有効な ObjectScript 式を使用できます。

### 概要

\$TRANSLATE 関数は、ソース文字列の 1 つ以上の文字単位の置換で構成される新しい文字列を返します。\$TRANSLATE 処理では、複数の文字が置換される場合もありますが、1 文字が (最大) 1 文字に置換されるだけの場合もあります。string 引数の処理を一度に 1 文字行います。入力文字列にある各文字と identifier 引数にある各文字とを比較します。\$TRANSLATE が一致を見つけると、その文字に対して以下のいずれかのアクションを実行します。

- ・ \$TRANSLATE の 2 引数形式は、返された文字列から identifier 引数の文字を取り除きます。
- ・ \$TRANSLATE の 3 引数形式は、string で見つかった identifier 文字を、associator 引数から位置的に対応する文字と置き換え、結果の文字列を返します。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 引数よりも少ない場合、associator 引数にある余分な文字は無視されます。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で削除されます。

\$TRANSLATE は、大文字と小文字を区別します。

通常、string、identifier、および associator 引数は、引用符付きの文字列として指定されます。これらの引数のいずれかの値が純粋な数値である場合、文字列の引用符は不要です。ただし、InterSystems IRIS は \$TRANSLATE に値を指定する前に引数値をキャノニック形式の数に変換するため、この使用は推奨されません。

### 例

以下の例は、\$TRANSLATE の 2 つの使用法を示しています。最初の \$TRANSLATE では、入力文字列値は変更されません。2 番目の \$TRANSLATE では、関数の返り値と同じ値に設定することで、入力文字列値が変更されます。

#### ObjectScript

```
SET str="The quick brown fox"
SET newstr=$$TRANSLATE(str,"qbf","QBF")
WRITE "source string: ",str,!,"new string: ",newstr,!
// creates a new string, does not change str value
SET str=$$TRANSLATE(str,"qbf","QBF")
WRITE "revised string: ",str
// creates a new string and replaces str with new string value
```

以下の例では、2 引数 \$TRANSLATE が現在のロケールの設定に基づいて数値グループ区切り文字を削除します。

### ObjectScript

```
AppropriateInput
SET ds=##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
IF ds="." {SET x="+1,462,543.33"}
ELSE {SET x="+1.462.543,33"}
TranslateNum
WRITE !,"before translation ",x
SET ngs=##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator")
IF ngs="," {SET x=$TRANSLATE(x,"")}
ELSEIF ngs="." {SET x=$TRANSLATE(x,".")}
ELSEIF ngs=" " {SET x=$TRANSLATE(x," ")}
ELSE {WRITE "Non-standard NumericGroupSeparator:", ngs
      RETURN }
WRITE !,"after translation ",x
```

以下の例では、3 引数 \$TRANSLATE がさまざまな日付区切り文字をスラッシュに置き換えます。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。

### ObjectScript

```
SET x(1)="06-23-2014"
SET x(2)="06.24.2014"
SET x(3)="06/25/2014"
SET x(4)="06|26|2014"
SET x(5)="06 27 2014"
FOR i=1:1:5{
    SET x(i)=$TRANSLATE(x(i),"- .|","/")
    WRITE "x(",i,") :",x(i),!
}
```

以下の例では、3 引数 \$TRANSLATE がアクセント付き文字をアクセントなし文字に置き換え、疑問文および感嘆文の接頭辞句読点を削除することで、スペイン語を基本 ASCII に“簡略化”します。

### ObjectScript

```
SET esp="{Sabes lo que ocurrirá en el año 2016?}"
WRITE "Spanish:",!,esp,!
SET iden=$CHAR(225)_$CHAR(233)_$CHAR(237)_$CHAR(241)_$CHAR(243)_$CHAR(250)_$CHAR(161)_$CHAR(191)
SET asso="aeinou"
WRITE "Identifier: ",iden,!
WRITE "Associator: ",asso,!
SET spanglish=$TRANSLATE(esp,iden,asso)
WRITE "Spanglish:",!,spanglish
```

もちろん、この変換は、実際のスペイン語テキストでの使用はお勧めしません。

## \$TRANSLATE、\$REPLACE、および \$CHANGE

\$TRANSLATE は文字と文字をマッチングして置換します。[\\$REPLACE](#) と [\\$CHANGE](#) は文字列と文字列をマッチングして置換します。[\\$REPLACE](#) と [\\$CHANGE](#) は、1 つ以上の文字の単一の指定された部分文字列を別の任意の長さの部分文字列に置換できます。[\\$TRANSLATE](#) は、複数の指定された文字を対応する指定された置換文字に置換できます。この 3 つの関数はすべて、一致する文字または部分文字列を削除 (NULL に変換) します。

\$TRANSLATE のマッチングでは、常に大文字と小文字が区別されます。[\\$REPLACE](#) のマッチングでは既定で大文字と小文字が区別されますが、大文字と小文字を区別せずに呼び出すこともできます。[\\$CHANGE](#) では、常に大文字と小文字が区別されます。

\$TRANSLATE では常にソース文字列の一致対象がすべて置換されます。[\\$REPLACE](#) と [\\$CHANGE](#) では、マッチングの開始ポイントおよび/または実行する置換の数を指定できます。[\\$REPLACE](#) と [\\$CHANGE](#) では、開始ポイントの定義方法が異なります。

## 関連項目

- [\\$CHANGE](#) 関数

- ・ [\\$EXTRACT](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$REPLACE](#) 関数
- ・ [\\$REVERSE](#) 関数
- ・ [\\$ZCONVERT](#) 関数

## \$vector (ObjectScript)

指定の位置のベクトル・データを割り当て、返し、削除します。

### ベクトル・データの割り当て

```
set $vector(vector,position,type) = value
```

### ベクトル・データを返す

```
$vector(vector,position)
$vector(vector,startPosition,endPosition)
$vector(vector,startPosition,*)
$vector(vector,startPosition,* - offset)
```

### ベクトル・データの削除

```
kill $vector(vector,position)
```

## 説明

\$vector 関数は、ベクトル内の要素に値を割り当て、要素を削除することもできます。指定したベクトル要素の位置の値、または要素位置のサブセットからのベクトル・スライスを返すこともできます。ベクトルに対して操作を実行するには、[\\$vectorop](#) 関数を使用します。

ベクトルとは、データベースの列を格納する InterSystems IRIS® データ構造です。ベクトルの各要素は同じデータ型となります。\$vector を使用して列指向データにアクセスすると、分析クエリやその他のオンライン分析処理 (OLAP) トランザクションを、従来の行ベース (リスト) クエリより桁違いに高速化できます。

**省略形** : \$ve

### ベクトル・データの割り当て

- ・ `set $vector(vector,position,type) = value` は、指定したインデックス位置にあるベクトル要素を指定した値に割り当てます。要素は指定したデータ型です。有効なデータ型は、"integer" (または "int")、"double"、"decimal"、"string"、および "timestamp" です。
- － vector が定義されていない場合、\$vector 関数はベクトルを作成し、データ型を設定して、要素の値を割り当てます。例えば、次のコードは、"integer" 型のベクトルを作成し、3 番目の要素に値 10 を割り当てます。1 番目と 2 番目の要素は未定義です。

```
set $vector(vector,3,"integer") = 10
```

- － vector が既に定義されている場合、\$vector 関数は要素の値を割り当てます。type は以前に定義されたベクトルのデータ型に設定する必要があります。例えば、"double" 型のベクトルで `set $vector(vector,1,"integer") = 3` を呼び出すと、<VECTOR> エラーが発生します。

例 : [ベクトルの作成とベクトル要素の更新](#)

### ベクトル・データを返す

- ・ `$vector(vector,position)` は、指定したインデックス位置にあるベクトル要素の値を返します。例えば、次のコードは、ベクトルの 2 番目の要素の値を変数 x に割り当てます。

**ObjectScript**

```
set x = $vector(vector,2)
```

- ・ `$vector(vector,startPosition,endPosition)` は、元のベクトルのインデックス位置 `startPosition` と `endPosition` の間の要素を含む、新しいベクトルを返します。例えば、次のコードは、元のベクトルの 3 番目から 5 番目の要素を含むベクトルを、変数 `vector2` に格納します。

**ObjectScript**

```
set vector2 = $vector(vector,3,5)
```

返されるベクトルは、元のベクトルと同じ型になります。

- ・ `$vector(vector,startPosition,*)` は、インデックス位置 `startPosition` からベクトルの末尾までの要素を含む、新しいベクトルを返します。アスタリスク (\*) はベクトルの最後の要素を表します。例えば、次のコードは、4 番目の位置以降のすべての要素を含むベクトルを、変数 `vector2` に格納します。

**ObjectScript**

```
set vector2 = $vector(vector,4,*)
```

- ・ `$vector(vector,startPosition,* - offset)` は、インデックス位置 `startPosition` から、ベクトルの末尾から位置の `offset` 数を引いた位置までの要素を含む、新しいベクトルを返します。例えば、次のコードは、4 番目の位置から、最後から 2 番目の位置までのすべての要素を含むベクトルを、変数 `vector2` に格納します。

**ObjectScript**

```
set vector2 = $vector(vector,4,* - 1)
```

例：ベクトルからのデータの取得

**ベクトル・データの削除**

- ・ `kill $vector(vector,position)` は、ベクトルから指定した位置の要素を削除し、その要素を未定義に設定します。例えば、次のコードは、ベクトルの 1 番目の要素を削除します。

**ObjectScript**

```
kill $vector(vector,1)
```

要素を削除した結果、ベクトルの要素がなくなった場合は、以下のようになります。

- － `vector` がグローバル変数の場合、これは未定義になります。
- － `vector` がローカル変数の場合、これは空の文字列 ("") に設定されます。

例：ベクトルの作成とベクトル要素の更新

**引数****vector**

入力ベクトルを指定するグローバル変数またはローカル変数。

- ・ `vector` がベクトルではない (`$isvector(vector) = 0`) 場合、`$vector` は <VECTOR> エラーを発生させます。
- ・ `vector` が未定義であるか、空の文字列 ("") を保持している場合は、次のようになります。
  - － `$vector` は空の文字列を返します。
  - － `set $vector(vector,position,type) = value` は、指定した引数を持つ新しいベクトルを `vector` に割り当てます。
  - － `kill $vector(vector,position)` は、操作を行いません。



## position

割り当てたり、返したり、削除したりするベクトル要素の位置を指定する正の整数。position が 1 より小さい場合、\$vector は <VECTOR> エラーを発生させます。

position がベクトルの長さを超える場合、\$vector の動作は操作によって異なります。

範囲外の位置でのベクトル操作	結果
set \$vector(vector,position) = value	value を position にあるベクトル要素に割り当て、ベクトルの長さを position まで増加させます。元の長さと同じ長さの間の位置にある値は未定義のままです。
\$vector(vector,position)	空の文字列 (" ") を返します。
kill \$vector(vector,position)	操作を行いません。

位置をアスタリスク (\*) として指定することもできます。これは、最後に割り当てたベクトルの位置を指定するのと同じです。

## type

ベクトルのデータ型。次の文字列のいずれかとして指定されます。

ベクトルの型	ベクトル要素	対応する SQL タイプ
"integer" または "int"	整数	BIGINT
"double"	IEEE 倍精度 (64 ビット) バイナリ浮動小数点データ型に変換された数値。この形式の詳細は、" <a href="#">\$double</a> " を参照してください。	DOUBLE
"decimal"	InterSystems IRIS の 10 進数の浮動小数点データ型に変換された数値。この形式の詳細は、" <a href="#">\$decimal</a> " を参照してください。	DECIMAL
"string"	最大長 N の文字列。N より長い文字列を指定すると、<ILLEGAL VALUE> エラーが発生します。	VARCHAR
"timestamp"	整数ベースの日付時刻形式。この形式の詳細は、" <a href="#">%Library.PosixTime</a> " を参照してください。 タイムスタンプの操作例は、" <a href="#">タイムスタンプ・ベクトルの格納と表示</a> " を参照してください。	POSIXTIME

[vector](#) に割り当てられるすべての要素はこのデータ型になります。

## value

ベクトル要素に割り当てる値。value をベクトルに格納する前に、\$vector 関数は、標準の ObjectScript の型の変換規則を使用して、この値を指定した [type](#) に変換します。例えば、次のテーブルは、格納された 3.14 の value が、データ型によってどのように変化するかを示しています。

割り当てられたベクトル値	格納されたベクトル値
<code>set \$vector(vector,1,"integer") = 3.14</code>	3
<code>set \$vector(vector,1,"double") = 3.14</code>	3.1400000000000001243 この不正確さは、浮動小数点データの格納における制限によるものです。
<code>set \$vector(vector,1,"decimal") = 3.14</code>	3.14
<code>set \$vector(vector,1,"string") = 3.14</code>	"3.14"
<code>set \$vector(vector,1,"timestamp") = 3.14</code>	3

ObjectScript の変換規則の詳細は、“[変数のタイプと変換](#)”を参照してください。

## startPosition

返すベクトル・スライスの最初のインデックス位置を指定する正の整数。

`$vector(vector,startPosition,endPosition)` 構文は、startPosition 要素と endPosition 要素の間の vector 要素を含む、新しいベクトルを返します。新しいベクトルの長さは endPosition から startPosition を引いたものとなります。新しいベクトルのすべての要素は、元のベクトルの要素と同じ型となります。

`$vector` 関数は、次の条件において、空の文字列 (" ") を返します。

- ・ endPosition は startPosition より小さい。
- ・ startPosition は、最後に定義された vector 要素の位置より大きい。
- ・ startPosition と endPosition の間にはいずれの要素も定義されていない。

## endPosition

返すベクトル・スライスの最後のインデックス位置を指定する正の整数。

`$vector(vector,startPosition,endPosition)` 構文は、startPosition 要素と endPosition 要素の間の vector 要素を含む、新しいベクトルを返します。新しいベクトルの長さは endPosition から startPosition を引いたものとなります。新しいベクトルのすべての要素は、元のベクトルの要素と同じ型となります。

endPosition がベクトルの長さを超える場合、`$vector` 関数は、startPosition からベクトルの末尾までの要素を返します。この場合、アスタリスク構文 (\*) を使用してベクトルの末尾を指定するのと同じになります。例えば、次のコードで、ベクトル v1 と v2 はどちらも、vector の 4 番目と 5 番目の要素のみを含みます。

## ObjectScript

```
for i=1:1:5 set $vector(vector,i,"integer") = $random(100)+1
set v1 = $vector(vector,4,10)
set v2 = $vector(vector,4,*)
```

`$vector` 関数は、次の条件において、空の文字列 (" ") を返します。

- ・ endPosition は startPosition より小さい。
- ・ startPosition と endPosition の間にはいずれの要素も定義されていない。

## offset

ベクトル・スライスの最後のインデックス位置をオフセットする要素の数を指定する整数。アスタリスク構文(\*)を指定したoffsetを使用すると、最後のベクトル要素を基準にしたスライスを返します。例えば、\$vector(vector,1,\*-1)は、2番目から最後まで要素を返し、\$vector(vector,1,\*-2)は、3番目から最後まで要素を返します。

## 例

### ベクトルの作成とベクトル要素の更新

3要素の文字列ベクトルを作成し、zwriteコマンドを使用してベクトルのコンテンツを表示します。ベクトルの型は"string"に設定され、ベクトルの作成後に変更することはできません。要素数とベクトル長さはどちらも3に設定されます。

#### ObjectScript

```
set $vector(v,1,"string") = "a"
set $vector(v,2,"string") = "b"
set $vector(v,3,"string") = "c"
zwrite v
```

```
v={ "type": "string", "count": 3, "length": 3, "vector": [ "a", "b", "c" ] } ; <VECTOR>
```

ベクトルから2番目の要素を削除します。要素数は2に減少します。ベクトル長さは最後に定義された要素の位置と等しいため、これは3のままです。要素1("a")と3("c")の間のコンマが連続していることから、位置2の要素は未定義となったことがわかります。

#### ObjectScript

```
kill $vector(v,2)
zwrite v
```

```
v={ "type": "string", "count": 2, "length": 3, "vector": [ "a", , "c" ] } ; <VECTOR>
```

位置10に新しい要素を追加します。ベクトル数は3に増加し、ベクトル長さは10に増加します。位置4から9の要素は未定義です。

#### ObjectScript

```
set $vector(v,10) = "d"
zwrite v
```

```
v={ "type": "string", "count": 3, "length": 10, "vector": [ "a", , "c", , , , , , "d" ] } ; <VECTOR>
```

位置10の値を更新します。ベクトルの長さや数は変わりませんが、\$vector関数によって要素の値が更新されます。

#### ObjectScript

```
set $vector(v,10) = "j"
zwrite v
```

```
v={ "type": "string", "count": 3, "length": 10, "vector": [ "a", , "c", , , , , , "j" ] } ; <VECTOR>
```

ループでベクトルを更新し、欠落した要素を追加します。\$char関数を使用して、小文字のASCIIコードを文字列表現に変換します。これで、ベクトルの最初の10個の要素が定義されました。

## ObjectScript

```
set asciiOffset = 96 // lowercase letters start with ASCII code 97
for i = 1:1:10 set $vector(v,i) = $char(i + asciiOffset)
zwrite v
```

```
v={ "type": "string", "count": 10, "length": 10,
    "vector": [ "a", "b", "c", "d", "e", "f", "g", "h", "i", "j" ] } ; <VECTOR>
```

## ベクトルからのデータの取得

1 から 100 までのランダムな整数を含む 10 個の要素のベクトルを作成します。zwrite コマンドを使用して、コンテンツを表示します。出力は変化します。

## ObjectScript

```
for i = 1:1:100 set $vector(v1,i,"integer") = $random(100)+1
zwrite v1
```

```
v1={ "type": "integer", "count": 10, "length": 10,
     "vector": [ 89, 40, 20, 99, 32, 61, 55, 34, 19, 47 ] } ; <VECTOR>
```

ベクトルのさまざまな要素を変数に設定します。返り値はベクトルと同じ型となります (この場合は `integer`)。範囲外の要素は例外で、空の文字列に設定されます。

## ObjectScript

```
set element1 = $vector(v1,1)
set element2 = $vector(v1,2)
set element100 = $vector(v1,100)
zwrite element1, element2, element100
```

```
element1=89
element2=40
element100=""
```

ベクトルの最初の 5 つの要素を取得します。ベクトルから連続する要素の範囲を取得するには、それらの値のみを含む新しいベクトルを返します。このベクトルはベクトル・スライスと呼ばれることもあります。

## ObjectScript

```
set v2 = $vector(v1,1,5)
zwrite v2
```

```
v2={ "type": "integer", "count": 5, "length": 5, "vector": [ 89, 40, 20, 99, 32 ] } ; <VECTOR>
```

元のベクトルから 1 番目の要素を削除し、再度、最初の 5 つの要素を取得します。返されるベクトル・スライスには 5 つの要素が含まれますが、1 番目の要素は未定義です。

## ObjectScript

```
kill $vector(v1,1)
set v2 = $vector(v1,1,5)
zwrite v2
```

```
v2={ "type": "integer", "count":4, "length":5, "vector":[ ,40,20,99,32]} ; <VECTOR>
```

元のベクトルから 5 番目の要素を削除し、再度、最初の 5 つの要素を取得します。返されるベクトル・スライスには未定義の要素 5 は含まれません。これは、この要素がベクトルの最後にあるためです。

### ObjectScript

```
kill $vector(v1,5)
set v2 = $vector(v1,1,5)
zwrite v2
```

```
v2={ "type": "integer", "count":3, "length":4, "vector":[ ,40,20,99]} ; <VECTOR>
```

元のベクトルの要素 6 から、最後から 2 番目の要素までを取得します。これらの要素を新しいベクトルに格納します。\* を基準にして最後から 2 番目の要素を指定します。\* は最後のベクトル要素のインデックス位置を参照します。

### ObjectScript

```
set v3 = $vector(v1,6,*-1)
```

```
v3={ "type": "integer", "count":4, "length":4, "vector":[61,55,34,19]} ; <VECTOR>
```

ベクトルの最後の要素を取得し、範囲外の末尾の位置を指定します。\$vector 関数は、元のベクトルの最後の要素のみを含む 1 要素のベクトルを返します。

### ObjectScript

```
set v4 = $vector(v1,*,100)
```

```
v4={ "type": "integer", "count":1, "length":1, "vector":[47]} ; <VECTOR>
```

## タイムスタンプ・ベクトルの格納と表示

ベクトルは、タイムスタンプを Posix 時刻と呼ばれる (Unix 時刻や Epoch 時刻と呼ばれることもあります) 整数ベースの形式にエンコードします。各整数は、1970 年 1 月 1 日 00:00:00 以降 (または以前) の秒数です。この形式の詳細は、“%Library.PosixTime” を参照してください。

現在の時刻を 5 秒間、1 秒ずつの増分で取得します。[\\$datetime](#) 関数を使用して時刻を Posix 形式に変換します。この関数での変換コードは -2 です。これらの時刻をベクトルに格納し、ベクトルのコンテンツを表示します。時刻は変化します。

### ObjectScript

```
set posix = -2
for i=1:1:5 set $vector(v,i,"timestamp") = $zdatetime($now(),posix) hang 1
zwrite v
```

```
v={ "type": "timestamp", "count":5, "length":5,
  "vector":[1639672461,1639672463,1639672464,1639672465,1639672466]} ; <VECTOR>
```

ベクトルに格納されている時刻を読み取り可能な形式で表示します。

## ObjectScript

```
for i=1:1:5 set ts=$vector(v,i) write !,$zdatetime($zdatetimeh(ts,posix))
```

```
12/16/2021 11:34:21
12/16/2021 11:34:23
12/16/2021 11:34:24
12/16/2021 11:34:25
12/16/2021 11:34:26
```

`%Library.PosixTime` の `LogicalToDisplay` メソッドと `DisplayToLogical` メソッドを使用して、タイムスタンプ表示を変換することもできます。例えば、次のルーチンは、未加工のタイムスタンプ日付を Posix 時刻形式に変換し、そのタイムスタンプ日付をベクトルに格納して、それを読み取り可能な形式で表示します。

## ObjectScript

```
set dates = "10/10/2010 10:10:10;11/11/2011 11:11:11;12/12/2012 12:12:12"
set delimiter = ";"
set $vector(v) = "timestamp"
set numDates = $length(dates,delimiter)

for i = 1:1:numDates
{
    set dateString = $piece(dates,delimiter,i)
    set datePosix = ##class(%Library.PosixTime).DisplayToLogical(dateString)
    set $vector(v,i) = datePosix
    write !,##class(%Library.PosixTime).LogicalToDisplay($vector(v,i))
}
```

## 関連項目

- [\\$isvector](#)
- [\\$vectordefined](#)
- [\\$vectorop](#)

## \$vectordefined (ObjectScript)

指定した位置のベクトル要素が定義されているかどうかを判断します。

```
$vectordefined(vector,position)
$vectordefined(vector,position,value)
```

### 説明

- ・ `$vectordefined(vector,position)` は、指定したインデックス位置のベクトル要素が定義されている場合は、1 (true) を返します。要素が定義されていない場合、この関数は 0 (false) を返します。

例：ベクトル要素が定義されているかどうかの確認

- ・ `$vectordefined(vector,position,value)` は、ベクトル要素が定義されている場合、その値をローカル変数 `value` に格納します。

例：ベクトル要素の値の変数への格納

`$vectordefined` を使用してベクトル要素にデータが含まれているかどうかを確認します。これは、`$data` 関数を使用して変数にデータが含まれているかどうかを確認するのと同様です。

省略形：\$vd

### 引数

#### vector

入力ベクトルを指定するグローバル変数またはローカル変数。

- ・ `vector` がベクトルではない (`$isvector(vector) = 0`) 場合、`$vectordefined` は <VECTOR> エラーを発生させます。
- ・ `vector` が未定義であるか、空の文字列 ("") を保持している場合、`$vectordefined` は 0 を返し、`value` 変数 (指定されている場合) は空の文字列に設定されます。

#### position

確認するベクトル要素の位置を指定する正の整数。`position` が 1 より小さい場合、`$vectordefined` は <VECTOR> エラーを発生させます。

#### value

`position` にあるベクトル要素の値を格納するローカル変数。

- ・ 要素が定義されている場合 (`$vectordefined` が 1 を返す)、`value` はその要素の値に設定されます。
- ・ 要素が定義されていない場合 (`$vectordefined` が 0 を返す)、`value` は空の文字列 ("") に設定されます。

`value` 変数がこれまで存在しなかった場合は、`$vectordefined` によってこれが作成されます。

### 例

#### ベクトル要素が定義されているかどうかの確認

1 から 100 までのランダムな整数からなる長さ 10 のベクトルを定義して、1 つおきにのみ要素を定義します。ベクトルを表示します。連続するコンマが示すように、奇数番号の要素位置には値が含まれていません。要素の値は変化します。



## ObjectScript

```
for i=2:2:10 set $vector(vector,i,"integer") = $random(100)+1
zwrite vector
```

```
vector={"type":"integer", "count":5, "length":10, "vector":[,46,,67,,45,,82,,2]} ;
<VECTOR>
```

偶数番号の要素が定義されているかどうかを確認します。以下の \$vectordefined 呼び出しからはすべて 1 が返されます。

## ObjectScript

```
write $vectordefined(vector,2)
write $vectordefined(vector,4)
write $vectordefined(vector,6)
write $vectordefined(vector,8)
write $vectordefined(vector,10)
```

奇数番号の要素が定義されているかどうかを確認します。以下の \$vectordefined 呼び出しからはすべて 0 が返されます。

## ObjectScript

```
write $vectordefined(vector,1)
write $vectordefined(vector,3)
write $vectordefined(vector,5)
write $vectordefined(vector,7)
write $vectordefined(vector,9)
```

## ベクトル要素の値の変数への格納

10 要素のベクトルを定義します。これらの要素はアルファベットの最初の 10 文字に対応します。ベクトルを表示します。

## ObjectScript

```
for i=1:1:10 set $vector(vector,i,"string") = $extract("abcdefghij",i)
zwrite vec
```

```
vector={"type":"string", "count":10, "length":10,
"vector":["a","b","c","d","e","f","g","h","i","j"]} ; <VECTOR>
```

ベクトルの最初と最後の文字をローカル変数に格納します。これらの変数の値は、それぞれ "a" と "j" です。

## ObjectScript

```
write $vectordefined(vector,1,firstLetter)
write $vectordefined(vector,10,lastLetter)
zwrite firstLetter, lastLetter
```

ベクトルの長さを超える位置に値を格納してみます。この位置の値は未定義のため、変数は空の文字列に設定されます。

## ObjectScript

```
write $vectordefined(vector,26,undefinedLetter)
zwrite undefinedLetter
```

代わりに、\$vector 関数を使用して、ベクトル要素を変数に格納することができます。

## ObjectScript

```
set firstLetter = $vector(vector,1)
set lastLetter = $vector(vector,10)
set undefinedLetter = $vector(vector,26)
```

## 関連項目

- ・ [\\$vector](#)
- ・ [\\$isvector](#)
- ・ [\\$vectorop](#)

## \$vectorop (ObjectScript)

ObjectScript を通じて定義されたベクトルに対し、さまざまな操作を実行します。

### 集約操作

```
$vectorop("count", vexpr [ , bitexpr ] )
$vectorop("max", vexpr [ , bitexpr ] )
$vectorop("min", vexpr [ , bitexpr ] )
$vectorop("sum", vexpr [ , bitexpr ] )
```

### フィルタ操作

```
$vectorop("defined", vexpr)
$vectorop("undefined", vexpr)
$vectorop("<", vexpr, expr1)
$vectorop("<=", vexpr, expr1)
$vectorop(">", vexpr, expr1)
$vectorop(">=", vexpr, expr1)
$vectorop("=", vexpr, expr1)
$vectorop("!=", vexpr, expr1)
$vectorop("between", vexpr, expr1, expr2)
```

### ベクトル単位のフィルタ操作

```
$vectorop("v<", vexpr1, vexpr2)
$vectorop("v<=", vexpr1, vexpr2)
$vectorop("v>", vexpr1, vexpr2)
$vectorop("v>=", vexpr1, vexpr2)
$vectorop("v=", vexpr1, vexpr2)
$vectorop("v!=", vexpr1, vexpr2)
$vectorop("vbetween", vexpr1, vexpr2, vexpr3)
```

### 数値演算

```
$vectorop("+", vexpr, expr [ , bitexpr ] )
$vectorop("-", vexpr, expr [ , bitexpr ] )
$vectorop("/", vexpr, expr [ , bitexpr ] )
$vectorop("**", vexpr, expr [ , bitexpr ] )
$vectorop("***", vexpr, expr [ , bitexpr ] )
$vectorop("#", vexpr, expr [ , bitexpr ] )
$vectorop("e-", vexpr, expr [ , bitexpr ] )
$vectorop("e/", vexpr, expr [ , bitexpr ] )
$vectorop("v+", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("v-", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("v/", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("v*", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("v**", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("v#", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("ceiling", vexpr)
$vectorop("floor", vexpr)
```

### 文字列操作

```
$vectorop("_", vexpr, expr [ , bitexpr ] )
$vectorop("e_", vexpr, expr [ , bitexpr ] )
$vectorop("v_", vexpr1, vexpr2 [ , bitexpr ] )
$vectorop("lower", vexpr)
$vectorop("upper", vexpr)
$vectorop("substring", vexpr, intexpr1 [ , intexpr2 ... ] )
$vectorop("trim", vexpr)
$vectorop("triml", vexpr)
$vectorop("trimr", vexpr)
```

### グループ化操作

```
$vectorop("group", vexpr, bitexpr, array)
$vectorop("countg", vexpr1, vexpr2, array [ , bitexpr ] )
$vectorop("maxg", vexpr1, vexpr2, array [ , bitexpr ] )
$vectorop("ming", vexpr1, vexpr2, array [ , bitexpr ] )
$vectorop("sumg", vexpr1, vexpr2, array [ , bitexpr ] )
$vectorop("countgb", vexpr, bitexpr, array)
```

## その他の操作

```
$vectorop("convert", vexpr, expr [ , restrict ] )
$vectorop("length", vexpr)
$vectorop("mask", vexpr, expr)
$vectorop("positions", vexpr, bitexpr)
$vectorop("set", vexpr, expr)
$vectorop("set", vexpr, expr, type, bitexpr)
$vectorop("vset", vexpr1, vexpr2, [ , bitexpr ] )
```

## 情報操作

```
$vectorop("bytesize", vexpr)
$vectorop("type", vexpr)
```

## 引数

引数	説明
vexpr	ベクトルとして評価する有効な ObjectScript 式。一部のベクトル操作では、このベクトルが特定の型である必要があります。
expr	有効な ObjectScript 式。ベクトル操作では、多くの場合、この引数が特定の型である必要があります。
bitexpr	ビット文字列式。この引数は、 <code>\$bit</code> で作成されたビット文字列のグローバル変数名、または <code>\$bit</code> に渡すことのできる式にできます。  多くの場合、これはベクトル操作のオプションの引数です。これが含まれている場合、bitexpr に対応する 1 の値があるベクトル内の位置でのみ、特定の操作が適用されます。
intexpr	整数に解決されるか、強制的に整数にできる有効な ObjectScript 式。
array	ベクトル操作でデータを入力する配列の名前。vectorop の呼び出しが行われる前に、未定義にすることができます。

**省略形** : \$vop

## 集約操作

集約操作はベクトルを受け取り、数値を返します。1 つのベクトル全体で操作を実行します。ベクトル内のある位置が未定義の場合は、その位置は無視されます。

すべての集約操作に、オプションの 3 番目の引数 bitexpr があります。この引数を使用して、操作を実行する特定のベクトル要素を指定できます。これが含まれている場合、true の値を持つビット文字列内の位置に対応する位置のベクトル要素のみが、集約操作に使用されます。

- `$vectorop("count", vexpr [ , bitexpr ] )` は、ベクトル vexpr 内の定義された要素の数をカウントします。未定義の要素はカウントから除外されます。結果として、返り値は常に、vexpr の長さ以下となります。

3 番目の引数が指定されている場合、定義された要素を含む位置が操作でカウントされるのは、bitexpr の同じ位置に true の値がある場合のみです。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i // defines a 10-element vector
write $vectorop("count",vec) // writes 10
```

- ・ \$vectorop("max", vexpr [, bitexpr ]) は、ベクトルのデータ型で vexpr の最大要素を返します。未定義の要素は無視されます。

3 番目の引数が指定されている場合、bitexpr が true の値を持つベクトル内の要素が定義されている位置のみが、返される可能性のある最大値として考慮されます。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
write $vectorop("max", vec) // writes 10
```

- ・ \$vectorop("min", vexpr [, bitexpr ]) は、ベクトルのデータ型で vexpr の最小要素を返します。未定義の要素は無視されます。

3 番目の引数が指定されている場合、bitexpr が true の値を持つベクトル内の要素が定義されている位置のみが、返される可能性のある最小値として考慮されます。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
write $vectorop("min",vec) // writes 1
```

- ・ \$vectorop("sum", vexpr [, bitexpr ]) は、ベクトル内のすべての要素を加算し、ベクトルのデータ型に応じて、integer、double、または decimal の値を返します。この操作は、文字列またはタイムスタンプのデータ型のベクトルで呼び出されると、<FUNCTION> エラーを発生させます。

3 番目の引数が指定されている場合、bitexpr が true の値を持つベクトル内の要素が定義されている位置のみが、合計に加算されます。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
write $vectorop("sum",vec) // writes 55
```

## フィルタ操作

\$vectorop 操作では、ベクトルをフィルタ処理するための多くの操作を提供します。フィルタ操作には、主に標準とベクトル単位の 2 種類があります。標準のフィルタ操作は、リストや配列に対して実行されるフィルタ操作と同様に、ベクトルに対して実行されます。ベクトル単位のフィルタ操作は、2 つの異なるベクトルの同じ位置にある要素を比較し、特定の操作の条件を満たす要素を持つベクトルを返します。

いずれかの入力ベクトルが未定義であったり、空の文字列である場合は、空の文字列が返されます。引数がベクトルでないその他の場合（引数が数値型の場合など）、この操作は<VECTOR> エラーを発生させます。

### 標準のフィルタ操作

フィルタ操作は、入力ベクトルと同じ長さのビット文字列を返します。ここで、定義されたベクトルの位置は、指定された操作が対応する位置のベクトル内の引数をどのように評価するかに応じて、true または false の値に置き換えられます。

- ・ \$vectorop("defined", vexpr) は、入力ベクトルと同じ長さのビット文字列を返します。定義されているベクトル内の要素に対応する位置には 1、定義されていないベクトル内の要素に対応する位置には 0 が含まれます。

### ObjectScript

```
// vec = <1,,3,4,,6,,,9,10>
zwrite $vectorop("defined",vec) // writes the bitstring 1011010011
```

- ・ \$vectorop("undefined", vexpr) は、入力ベクトルと同じ長さのビット文字列を返します。定義されていないベクトル内の要素に対応する位置には 1、定義されているベクトル内の要素に対応する位置には 0 が含まれます。

### ObjectScript

```
// vec = <1,,3,4,,6,,,9,10>
zwrite $vectorop("undefined",vec) // writes the bitstring 0100101100
```

- ・ \$vectorop("<", vexpr, expr1) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値より小さいベクトル内の要素に対応する位置には 1、expr1 の値以上のベクトル内の要素に対応する位置には 0 が含まれます。この操作では、定義されていない位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop("<",vec, 5) // writes the bitstring 1011000000
```

- ・ \$vectorop("<=", vexpr, expr1) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値以下のベクトル内の要素に対応する位置には 1、expr1 の値より大きいベクトル内の要素に対応する位置には 0 が含まれます。この操作では、定義されていない位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop("<=",vec, 5) // writes the bitstring 1011100000
```

- ・ \$vectorop(">", vexpr, expr1) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値より大きいベクトル内の要素に対応する位置には 1、expr1 の値以下のベクトル内の要素に対応する位置には 0 が含まれます。この操作では、定義されていない位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop(">",vec, 5) // writes the bitstring 0000010011
```

- ・ \$vectorop(">=", vexpr, expr1) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値以上のベクトル内の要素に対応する位置には 1、expr1 の値より小さいベクトル内の要素に対応する位置には 0 が含まれます。この操作では、定義されていない位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop("=>",vec, 5) // writes the bitstring 0000110011
```

- ・ \$vectorop("=", vexpr, expr1) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値に等しいベクトル内の要素に対応する位置には 1、expr1 の値に等しくないベクトル内の要素に対応する位置には 0 が含まれます。この操作では、定義されていない位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop("=",vec, 5) // writes the bitstring 0000100000
```

- ・ \$vectorop("!=", vexpr, expr1) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値に等しくないベクトル内の要素に対応する位置には 1、expr1 の値に等しいベクトル内の要素に対応する位置には 0 が含まれます。この操作では、定義されていない位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop("!=" ,vec, 5) // writes the bitstring 1111011111
```

- ・ \$vectorop("between", vexpr, expr1, expr2) は、入力ベクトルと同じ長さのビット文字列を返します。expr1 の値以上、かつ、expr2 の値以下のベクトル内の要素に対応する位置には 1 が含まれます。expr1 と expr2 の値の間にないベクトル内の要素に対応する位置、および未定義の位置には 0 が返されます。

### ObjectScript

```
// vec = <1,,3,4,5,6,,,9,10>
zwrite $vectorop("between",vec, 3, 9) // writes the bitstring 0011110010
```

## ベクトル単位のフィルタ操作

ベクトル単位のフィルタ操作は、複数のベクトルを受け取り、入力ベクトルの対応する位置にある要素を比較する指定の演算子に基づいて、0 または 1 の値のビット文字列を返します。返されるビット文字列は、比較に使用された 2 つのベクトルのうち短い方と同じ長さになります。長い方のベクトルの余分な要素は無視されます。

入力ベクトルがすべて同じ型ではない場合、この操作は <FUNCTION> エラーを発生させます。

- ・ \$vectorop("v<", vexpr1, vexpr2) は、vexpr1 の値が vexpr2 の対応する値より小さい場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。vexpr1 または vexpr2 のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

### ObjectScript

```
// vec1 = <1,2,1,2>
// vec2 = <2,1,1,2>
zwrite $vectorop("v<",vec1,vec2) // writes 1000
```

- ・ \$vectorop("v<=", vexpr1, vexpr2) は、vexpr1 の値が vexpr2 の対応する値以下の場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。vexpr1 または vexpr2 のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

### ObjectScript

```
// vec1 = <1,2,1,2>
// vec2 = <2,1,1,2>
zwrite $vectorop("v<=",vec1,vec2) // writes 1011
```

- ・ \$vectorop("v>", vexpr1, vexpr2) は、vexpr1 の値が vexpr2 の対応する値より大きい場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。vexpr1 または vexpr2 のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

### ObjectScript

```
// vec1 = <1,2,1,2>
// vec2 = <2,1,1,2>
zwrite $vectorop("v>",vec1,vec2) // writes 0100
```

- ・ \$vectorop("v>=", vexpr1, vexpr2) は、vexpr1 の値が vexpr2 の対応する値以上の場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。vexpr1 または vexpr2 のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

### ObjectScript

```
// vec1 = <1,2,1,2>
// vec2 = <2,1,1,2>
zwrite $vectorop("v>=",vec1,vec2) // writes 0111
```



- ・ `$vectorop("v=", vexpr1, vexpr2)` は、`vexpr1` の値が `vexpr2` の対応する値と等しい場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。`vexpr1` または `vexpr2` のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

#### ObjectScript

```
// vec1 = <1,2,1,2>
// vec2 = <2,1,1,2>
zwrite $vectorop("v=",vec1,vec2) // writes 0011
```

- ・ `$vectorop("v!=", vexpr1, vexpr2)` は、`vexpr1` の値が `vexpr2` の対応する値と等しくない場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。`vexpr1` または `vexpr2` のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

#### ObjectScript

```
// vec1 = <1,2,1,2>
// vec2 = <2,1,1,2>
zwrite $vectorop("v!=",vec1,vec2) // writes 1100
```

- ・ `$vectorop("vbetween", vexpr1, vexpr2, vexpr3)` は、`vexpr1` の値が `vexpr2` の対応する値以上、かつ `vexpr3` の対応する値以下の場合、各位置での値が 1 のビット文字列を返します。それ以外の場合、値は 0 です。`vexpr1`、`vexpr2`、または `vexpr3` のいずれかで要素が未定義の場合、ビット文字列内の対応する値は 0 です。

#### ObjectScript

```
// vec1 = <5,6,7>
// vec2 = <4,7,8>
// vec3 = <9,9,9>
zwrite $vectorop("vbetween",vec1,vec2,vec3) // writes 100
```

## 数値演算

数値演算は入力ベクトルに対して算術演算を実行し、演算の引数で定義された算術式の結果を含む新しいベクトルを返します。ベクトル引数が未定義または空の文字列の場合、これらの演算は空の文字列を返します。

ベクトル引数のデータ型が文字列の場合は、〈FUNCTION〉エラーがスローされます。

### 基本的な算術演算

基本的な算術演算はベクトルと単一の式を受け取ります。この式は暗黙的にベクトルのデータ型に強制されます。指定された値がベクトルのデータ型の範囲に合わない場合、この演算は〈MAXERROR〉エラーを発生させます。

基本的な算術演算では、オプションのビット文字列引数を指定できます。この引数が指定されている場合、演算はビット文字列の `true` の値に対応する位置にのみ適用され、結果としてこれらの位置のみが結果のベクトルで定義されます。

- ・ `$vectorop("+", vexpr, expr [, bitexpr ])` は、入力ベクトルと同じ長さおよび型で、定義されている `vexpr` の各要素に `expr` の値を加算したベクトルを返します。

この演算は、タイムスタンプ・データ型のベクトルに適用できます。この場合、`expr` は、`vexpr` の値に加算するマイクロ秒数です。

3 番目の引数が指定されている場合、`vexpr` と `expr` の要素の加算は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

#### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("+",vec,7) // writes <11,13,14,17>
```

- ・ `$vectorop("-", vexpr [, expr, bitexpr])` は、入力ベクトルと同じ長さおよび型で、定義されている `vexpr` の各要素から `expr` の値を減算したベクトルを返します。`expr` が指定されていない場合、返されるベクトルには、`vexpr` 内の定義されている各要素に "-" の符号を付けた値が含まれます。

この演算は、タイムスタンプ・データ型のベクトルに適用できます。この場合、`expr` は、`vexpr` の値から減算するマイクロ秒数です。

3 番目の引数が指定されている場合、`vexpr` と `expr` の要素の減算は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

#### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("-",vec,7) // writes <-3,-1,0,3>
```

- ・ `$vectorop("/", vexpr, expr [, bitexpr])` は、入力ベクトルと同じ長さおよび型で、各要素が `vexpr` の各要素を `expr` の値で除算した商の値を持つベクトルを返します。

この演算は、タイムスタンプ・データ型のベクトルに適用できます。この場合、`expr` は、`vexpr` の値に加算または減算するマイクロ秒数です。

3 番目の引数が指定されている場合、`vexpr` と `expr` の要素の除算は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

#### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("/",vec,2) // writes <2,3,3,5>
```

- ・ `$vectorop("*", vexpr, expr [, bitexpr])` は、入力ベクトルと同じ長さおよび型で、定義されている `expr` の各要素に `vexpr` の値を乗算したベクトルを返します。

3 番目の引数が指定されている場合、`vexpr` と `expr` の要素の乗算は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

#### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("*",vec,2) // writes <8,12,14,20>
```

- ・ `$vectorop("**", vexpr, expr [, bitexpr])` は、入力ベクトルと同じ長さおよび型で、定義されている `expr` の各要素の `vexpr` 乗のベクトルを返します。

3 番目の引数が指定されている場合、`vexpr` の要素の `expr` 乗は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

#### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("**",vec,2) // writes <16,36,49,100>
```

- ・ `$vectorop("#", vexpr, expr [, bitexpr])` は、入力ベクトルと同じ長さおよび型で、各要素が `vexpr` の各要素を `expr` の値で除算した余りの値を持つベクトルを返します。

3 番目の引数が指定されている場合、`vexpr` と `expr` の要素の除算の余りは、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ返されます。

#### ObjectScript

```
// vec = <4,6,7,11>
zwrite $vectorop("#",vec,3) // writes <1,0,1,2>
```

- ・ `$vectorop("e-", vexpr, expr [, bitexpr ])` は、入力ベクトルと同じ長さおよび型で、`expr` から、定義されている `vexpr` の各要素の値を減算したベクトルを返します。

3 番目の引数が指定されている場合、`expr` と `vexpr` の要素の減算は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("e-",vec,7) // writes <3,1,0,-3>
```

- ・ `$vectorop("e/", vexpr, expr [, bitexpr ])` は、入力ベクトルと同じ長さおよび型で、各要素が `expr` を `vexpr` の各要素の値で除算した商の値を持つベクトルを返します。

3 番目の引数が指定されている場合、`expr` と `vexpr` の要素の除算は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

### ObjectScript

```
// vec = <4,6,7,10>
zwrite $vectorop("e/",vec,8) // writes <2,1,1,0>
```

## ベクトル単位の算術演算

ベクトル単位の算術演算は、2 つのベクトルを受け取り、対応する位置にある要素に対して指定された演算を適用します。返されるベクトルは、2 つの入力ベクトルのうち短い方の長さになります。必要な 2 つの引数のいずれかがベクトルではない場合、演算は `<VECTOR>` エラーを発生させます。

ベクトル単位の算術演算では、オプションのビット文字列引数を指定できます。この引数が指定されている場合、演算はビット文字列の `true` の値に対応する位置にのみ適用され、結果としてこれらの位置のみが結果のベクトルで定義されます。

- ・ `$vectorop("v+", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` と `vexpr2` の対応する位置にある要素を足し合わせたベクトルを返します。`vexpr1` の型が `integer`、`decimal`、または `double` の場合、`vexpr2` は同じ型を持つ必要があります。同じ型でない場合、演算は `<FUNCTION>` エラーを発生させます。

`vexpr1` が `timestamp` のベクトル型である場合、`vexpr2` は `integer` のベクトル型を持つ必要があります。`vexpr2` は、`vexpr1` のタイムスタンプに加算するマイクロ秒数を表します。

3 番目の引数が指定されている場合、`vexpr1` と `vexpr2` の要素の加算は、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

### ObjectScript

```
// vec1 = <3,6,2,5>
// vec2 = <1,3,7,9>
zwrite $vectorop("v+",vec1,vec2) // writes <4,9,9,14>
```

- ・ `$vectorop("v-", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` の要素から `vexpr2` の要素を減算したベクトルを返します。`vexpr1` の型が `integer`、`decimal`、または `double` の場合、`vexpr2` は同じ型を持つ必要があります。同じ型でない場合、演算は `<FUNCTION>` エラーを発生させます。

`vexpr1` が `timestamp` のベクトル型である場合、`vexpr2` は `integer` のベクトル型を持つ必要があります。`vexpr2` は、`vexpr1` のタイムスタンプから減算するマイクロ秒数を表します。

3 番目の引数が指定されている場合、`vexpr1` と `vexpr2` の要素の減算は、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

## ObjectScript

```
// vec1 = <3,6,2,5>
// vec2 = <1,3,7,9>
zwrite $vectorop("v-",vec1,vec2) // writes <2,3,-5,-4>
```

- ・ `$vectorop("v/", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` の要素を `vexpr2` の要素で除算したベクトルを返します。`vexpr1` の型が `integer`、`decimal`、または `double` の場合、`vexpr2` は同じ型を持つ必要があります。同じ型でない場合、演算は〈FUNCTION〉エラーを発生させます。`vexpr1` と `vexpr2` が `integer` の場合、演算では整数の除算が実行され、結果は `integer` のベクトルとなります。

3 番目の引数が指定されている場合、`vexpr1` と `vexpr2` の要素の除算は、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

## ObjectScript

```
// vec1 = <3,6,2,5>
// vec2 = <1,3,7,9>
zwrite $vectorop("v*",vec1,vec2) // writes <3,2,0,0>
```

- ・ `$vectorop("v*", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` の要素に `vexpr2` の要素を乗算したベクトルを返します。`vexpr1` の型が `integer`、`decimal`、または `double` の場合、`vexpr2` は同じ型を持つ必要があります。同じ型でない場合、演算は〈FUNCTION〉エラーを発生させます。

3 番目の引数が指定されている場合、`vexpr1` と `vexpr2` の要素の乗算は、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

## ObjectScript

```
// vec1 = <3,6,2,5>
// vec2 = <1,3,7,9>
write $vectorop("v*",vec1,vec2) // writes <3,18,14,45>
```

- ・ `$vectorop("v**", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` の要素の `vexpr2` 乗のベクトルを返します。`vexpr1` の型が `integer`、`decimal`、または `double` の場合、`vexpr2` は同じ型を持つ必要があります。同じ型でない場合、演算は〈FUNCTION〉エラーを発生させます。

3 番目の引数が指定されている場合、`vexpr1` の要素の、対応する `vexpr2` 要素のべき乗は、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

## ObjectScript

```
// vec1 = <3,6,2,5>
// vec2 = <1,3,7,9>
zwrite $vectorop("v**",vec1,vec2) // writes <3,216,128,1953125>
```

- ・ `$vectorop("v#", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` の要素を `vexpr2` の要素で除算した余りを含むベクトルを返します。`vexpr1` の型が `integer`、`decimal`、または `double` の場合、`vexpr2` は同じ型を持つ必要があります。同じ型でない場合、演算は〈FUNCTION〉エラーを発生させます。

3 番目の引数が指定されている場合、`vexpr1` の要素と対応する `vexpr2` の要素の除算の余りは、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ返されます。

## ObjectScript

```
// vec1 = <3,6,2,5>
// vec2 = <1,3,7,9>
zwrite $vectorop("v#",vec1,vec2) // writes <0,0,2,5>
```

## 丸め演算

- ・ `$vectorop("ceiling", vexpr)` は、`vexpr` の各要素に切り上げ演算を適用した結果を要素の値とするベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。`vexpr` が `ineger`、`decimal`、または `double` 型のベクトルでない場合、演算は `<FUNCTION>` エラーを発生させます。

### ObjectScript

```
// vec = <2.4, 2.5, 2.7, 2.81, 2.19, 2.0>
zwrite $vectorop("ceiling",vec) // writes <3,3,3,3,3,2>
```

- ・ `$vectorop("floor", vexpr)` は、`vexpr` の各要素に切り下げ演算を適用した結果を要素の値とするベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。`vexpr` が `ineger`、`decimal`、または `double` 型のベクトルでない場合、演算は `<FUNCTION>` エラーを発生させます。

### ObjectScript

```
// vec = <2.4, 2.5, 2.7, 2.81, 2.19, 2.0>
zwrite $vectorop("floor",vec) // writes <2,2,2,2,2,2>
```

## 文字列操作

文字列操作は、入力ベクトルと同じ長さで、各位置に、指定した文字列操作の結果に基づく要素値が含まれる文字列ベクトルを返します。ある位置で値が未定義の場合、結果として得られるベクトルのその位置の値は未定義となります。

ベクトル入力のデータ型が `string` でない場合、操作は `<FUNCTION>` エラーを発生させます。

## 連結操作

連結操作は、文字列ベクトルを受け取り、別の引数をベクトルの各要素に連結します。

- ・ `$vectorop("_", vexpr, expr [, bitexpr ])` は、定義されている各要素が `expr` と連結されたベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

3 番目の引数が指定されている場合、`vexpr` と `expr` の要素の連結は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

### ObjectScript

```
// vec = <"b","great","larg">
zwrite $vectorop("_",vec,"est") // writes <"best","greatest","largest">
```

- ・ `$vectorop("e_", vexpr, expr [, bitexpr ])` は、定義されている各要素が `expr` と連結されたベクトルを返します。この操作が `$vectorop("_", ...)` と異なるのは、`vexpr` と `expr` の引数が使用される順序のみです。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

3 番目の引数が指定されている場合、`expr` と `vexpr` の要素の連結は、ベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

### ObjectScript

```
// vec = <"b","great","larg">
zwrite $vectorop("e_",vec,"est") // writes <"estb","estgreat","estlarg">
```

- ・ `$vectorop("v_", vexpr1, vexpr2 [, bitexpr ])` は、`vexpr1` の定義されている各要素が `vexpr2` の定義されている対応する要素と連結されたベクトルを返します。`vexpr1` または `vexpr2` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義になります。

3 番目の引数が指定されている場合、`vexpr1` と `vexpr2` の要素の連結は、両方のベクトルに要素が定義されており、`bitexpr` の値が `true` である位置に対してのみ実行されます。

### ObjectScript

```
// vec1 = <"take on", "take me", "I'll be">
// vec2 = <" me", " on", " gone">
zwrite $vectorop("v_",vec1,vec2) // writes <"take on me","take me on","I'll be gone">
```

## 大文字/小文字の操作

大文字/小文字の操作は、ベクトル内の文字列の大文字/小文字を変更します。

- ・ `$vectorop("lower", vexpr)` は、`vexpr` の各要素が小文字に変換されたベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

### ObjectScript

```
// vec = <"UpPeR", "UPPER", "upper">
zwrite $vectorop("lower",vec) // writes <"upper","upper","upper">
```

- ・ `$vectorop("upper", vexpr)` は、`vexpr` の各要素が大文字に変換されたベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

### ObjectScript

```
// vec = <"UpPeR", "UPPER", "upper">
zwrite $vectorop("upper",vec) // writes <"UPPER","UPPER","UPPER">
```

## 部分文字列

`$vectorop("substring", vexpr, intexpr1 [, intexpr2 ])` は、`vexpr` の定義された各要素から部分文字列を取得し、その部分文字列を含む新しいベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

部分文字列の範囲は、`intexpr1` とオプションの `intexpr2` によって決定されます。`intexpr1` のみが指定されている場合、結果として得られる文字列には、`intexpr1` の位置にある文字のみが含まれます。`intexpr2` が指定されている場合、`intexpr1` は部分文字列の開始位置を特定し、`intexpr2` は部分文字列の終了位置を特定します。`intexpr1` と `intexpr2` の位置にある文字はどちらも部分文字列に含まれます。`intexpr2` が特定の文字列の長さより大きい場合、この操作はエラーを発生させるのではなく、`intexpr1` で始まり文字列の末尾で終わる部分文字列を返します。

### ObjectScript

```
// vec = <"long","short",,,"headstrong","teleport">
zwrite $vectorop("substring",vec,3,6)
// writes <"ng","ort",,,"adst","lepo">
```

## トリミング操作

トリミング操作は、文字列の先頭または末尾、あるいはその両方から文字を削除します。

- ・ `$vectorop("trim", vexpr)` は、`vexpr` の各要素がトリミングされたベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

### ObjectScript

```
// vec = <" left", "right ", " middle ">
zwrite $vectorop("trim",vec) // writes <"left","right","middle">
```

- ・ `$vectorop("triml", vexpr)` は、`vexpr` の各要素がトリミングされたベクトルを返します。`vexpr` 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。



## ObjectScript

```
// vec = <" left", "right ", " middle ">
zwrite $vectorop("triml",vec) // writes <"left","right ", "middle ">
```

- ・ \$vectorop("trimr", vexpr) は、vexpr の各要素がトリミングされたベクトルを返します。vexpr 内のある位置が定義されていない場合、その位置は、結果のベクトルでも未定義のままになります。

## ObjectScript

```
// vec = <" left", "right ", " middle ">
zwrite $vectorop("trimr",vec) // writes <" left","right", " middle">
```

## グループ化操作

グループ化操作により、さまざまな操作をベクトルのサブセットに対して実行することができます。結果は多次元配列に格納されます。すべてのグループ化操作の返り値は未定義となります。

### 基本的なグループ化

\$vectorop("group", vexpr, bitexpr, array) は、bitexpr の対応する位置が true である vexpr の各値の添え字を、配列 array に入力します。false の bitexpr の位置に対応する vexpr の値は無視されます。

bitexpr の対応する位置が true の vexpr の位置で未定義のものがある場合、array は未定義に設定されます。配列に既に要素が定義されている場合、その配列はこの操作で未定義としてマークされませんが、新しい要素は追加されません。

vexpr が未定義または空の文字列の場合、bitexpr に true に設定されている位置があるかどうかに応じて、array の最上位ノードのみが変更される場合があります。

```
// vec = <10,8,6,15,3,17,18,2,7,100,101>
kill arr
zwrite $vectorop("group",vec,$factor(345),arr) // writes ""
zwrite arr

arr(3)=" "
arr(7)=" "
arr(10)=" "
arr(15)=" "
arr(18)=" "
```

### ベクトルのグループ化

ベクトルのグループ化操作は、2 つのベクトル (vexpr1 と vexpr2)、多次元配列 (array)、およびオプションでビット文字列 (bitexpr) を受け取ります。データをグループ化した後、そのグループに対して指定の操作を実行し、結果を多次元配列に格納します。

vexpr2 内の要素は、vexpr1 の要素によって識別される、位置的に対応するグループに分割されます。例えば、vexpr2 の位置 5 の要素は、vexpr1 の位置 5 の値によって識別されるグループに属します。array には、vexpr1 の一意の値ごとに 1 つの添え字が含まれ、その添え字にグループの合計が格納されます。vexpr2 で定義されているが、vexpr1 では定義されていない位置は、array の最上位ノードに関連付けられたグループに属します。

array 内の既存の添え字は上書きされません。いずれかの添え字が vexpr1 の要素である場合、グループの合計値は、その添え字に既に格納されている値の末尾に追加されます。

オプションの bitexpr 引数が渡される場合、グループは、bitexpr が true の値を持つ vexpr2 の値に対してのみ形成されます。

- ・ \$vectorop("countg", vexpr1, vexpr2, array [, bitexpr ]) は、グループ内の要素数をカウントし、それらを array の添え字に格納します。



### ObjectScript

```
// vec1 = <"g1","g2","g1","g1","g2","g2","g2","g2","g1">
// vec2 = <10,8,6,15,3,17,16,2,7,100,101>
kill arr
zwrite $vectorop("countg",vec1,vec2,arr) // writes ""
zwrite arr

arr=2
arr("g1")=4
arr("g2")=5
```

- ・ \$vectorop("maxg", vexpr1, vexpr2, array [ , bitexpr ]) は、グループ内の最大値を見つけ、それらを array の添え字に格納します。

### ObjectScript

```
// vec1 = <"g1","g2","g1","g1","g2","g2","g2","g2","g1">
// vec2 = <10,8,6,15,3,17,16,2,7,100,101>
kill arr
zwrite $vectorop("maxg",vec1,vec2,arr) // writes ""
zwrite arr

arr=101
arr("g1")=15
arr("g2")=17
```

- ・ \$vectorop("ming", vexpr1, vexpr2, array [ , bitexpr ]) は、グループ内の最小値を見つけ、それらを array の添え字に格納します。

### ObjectScript

```
// vec1 = <"g1","g2","g1","g1","g2","g2","g2","g2","g1">
// vec2 = <10,8,6,15,3,17,16,2,7,100,101>
kill arr
zwrite $vectorop("ming",vec1,vec2,arr) // writes ""
zwrite arr

arr=100
arr("g1")=6
arr("g2")=2
```

- ・ \$vectorop("sumg", vexpr1, vexpr2, array [ , bitexpr ]) は、グループの合計を見つけ、それらを array の添え字に格納します。

vexpr2 の型が string または timestamp の場合、この操作は <FUNCTION> エラーを発生させます。

### ObjectScript

```
// vec1 = <"g1","g2","g1","g1","g2","g2","g2","g2","g1">
// vec2 = <10,8,6,15,3,17,16,2,7,100,101>
kill arr
zwrite $vectorop("sumg",vec1,vec2,arr) // writes ""
zwrite arr

arr=201
arr("g1")=38
arr("g2")=46
```

## ビットマップのグループ化

\$vectorop("countgb", vexpr, bitexpr, array) は、true の値を持つ bitexpr の位置に対応する位置のサブセットから、vexpr 内の同一の値をグループ化します。この操作は、vexpr のこのサブセットを添え字として array に追加し、サブセット内にこの値が出現する回数を格納します。ベクトルのグループ化操作と同様、この関数は、array 引数に値を入力し、返り値は未定義です。

## ObjectScript

```
// vec = <"x", "y", "x", "y", "x", "y", "y", "y", "x">
kill arr
zwrite $vectorop("countgb",vec,$factor(205),arr) // writes "
zwrite arr

arr("x")=2
arr("y")=3
```

## その他の操作

InterSystems IRIS は、ベクトル内の情報の記述、ベクトルの型の変換、ベクトルの要素の設定など、さまざまな用途に使用できるベクトル操作を、他にも多数用意しています。

- ・ `$vectorop("convert", vexpr, expr[, restrict])` は、ObjectScript の変換規則に従って、`vexpr` の各要素が `expr` の型に強制変換された新しいベクトルを返します。`vexpr` 内の要素がターゲットの型の範囲に合わない場合、この操作は <MAXNUMBER> エラーを発生させます。`expr` は、文字列として表される有効なベクトル型である必要があります。

オプションの `restrict` 引数は、デフォルトでは 0 に設定されますが、これを 1 に設定すると、`vexpr` 内の対応する値が要求された型に適合しない位置では未定義の値を持つように、返されるベクトルを制限することができます。これが発生する可能性があるケースとして、以下の 3 つが挙げられます。

- `vexpr` 内の文字列値が有効な数値式ではない場合
- 変換により精度が失われる場合 (integer 型に変換する場合など)
- <MAXNUMBER> エラーが発生する場合

`vexpr` で未定義の位置は、返されるベクトルでも未定義です。

`vexpr` が未定義または空の文字列の場合、この操作は空の文字列を返します。`vexpr` がベクトルではない場合、または `expr` が有効なベクトル型に評価されない場合、この操作は <VECTOR> エラーを発生させます。

## ObjectScript

```
// vec = <"1", "one", "4", "seven", "10">
zwrite $vectorop("convert", vec, "int", 1) // writes <1,,4,,10>
```

- ・ `$vectorop("length", vexpr)` は、`vexpr` の長さを返します。ベクトルの長さは、暗黙的に、定義された要素を持つ最後の位置として定義されます。`vexpr` には、複数の未定義要素が含まれる場合があります。

`vexpr` が未定義または空の文字列の場合、この操作は 0 を返します。

## ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
write $vectorop("length",vec) // writes 10
```

- ・ `$vectorop("mask", vexpr, bitexpr)` は、`bitexpr` が 0 の値を持つ未定義の位置がある `vexpr` と同じ型のベクトルを返します。結果として得られるベクトルの長さは、`bitexpr` となります。`vexpr` に定義された値があるすべての位置で `bitexpr` の値が 0 のため、結果のベクトルに未定義の要素しか含まれない場合は、空の文字列が返されます。

`vexpr` が未定義または空の文字列の場合、この操作は空の文字列を返します。さらに、`bitexpr` が `vexpr` より長い場合、この操作は空の文字列を返します。

## ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
zwrite $vectorop("mask",vec,$factor(500)) // writes <,,3,,5,6,7,8,9>
```

- ・ `$vectorop("positions", bitexpr)` は、`bitexpr` が 1 である `vexpr` 内の各位置がその位置の整数自体に設定されている整数ベクトルを返します。`bitexpr` が 0 の位置は未定義です。

### ObjectScript

```
// bitexpr = $factor(46)
zwrite $vectorop("positions",bitexpr) // writes <,2,3,4,,6>
```

- ・ `$vectorop("set", vexpr, expr)` は、`vexpr` と同じ型および長さであるものの、各要素が `expr` に設定されている新しいベクトルを返します。`expr` の型は、強制的にベクトルの型に変換されます。指定された値がベクトルのデータ型の範囲に合わない場合、この操作は `<MAXNUMBER>` エラーを発生させます。

`vexpr` が未定義または空の文字列の場合、この操作は空の文字列を返します。`vexpr` は定義されている値だが、ベクトルでも空の文字列でもない場合、この操作は `<VECTOR>` エラーを発生させます。

### ObjectScript

```
// vec = <60,40,80,90>
$vectorop("set", vec, 500) // writes <500,500,500,500>
```

- ・ `$vectorop("set", vexpr, expr, type, bitexpr)` は、`vexpr` と同じ型および長さの新しいベクトルを返します。ここで、指定された `bitexpr` が true の値を持つすべての位置は、`expr` に設定され、ObjectScript の変換規則を使用して、暗黙的にそのベクトルの型に強制的に変換されます。返されるベクトルの長さは、`bitexpr` の長さとなります。指定された値がベクトルのデータ型の範囲に合わない場合、この操作は `<MAXNUMBER>` エラーを発生させます。

`vexpr` が未定義または空の文字列の場合、返されるベクトルの長さは、`bitexpr` が true の値を持つ最後の位置を介して定義されます。この場合、返されるベクトルの型は、`type` で指定される型になります。`vexpr` が定義されている場合、`type` はその型と一致する必要があります。`type` は常に有効なベクトルの型として定義される必要があり、定義されていない場合、この操作は `<VECTOR>` エラーを発生させます。

`bitexpr` が未定義または空の文字列の場合、返されるベクトルは、`vexpr` と同じになります。

`vexpr` が定義されているがベクトルではない場合、この操作は `<VECTOR>` エラーを発生させます。`vexpr` と `bitexpr` が未定義または空の文字列の場合、この操作は空の文字列を返します。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
zwrite $vectorop("set",vec,16,"int",$factor(63))
// writes <1,100,100,4,100,100,7,8,9,10>
```

- ・ `$vectorop("vset", vexpr1, vexpr2[, bitexpr])` は、`vexpr1` と同じ型および長さのベクトルを返します。ここで、各位置の要素は、`vexpr2` の対応する値 (これが定義されている場合)、または `vexpr1` の値 (`vexpr2` の値が未定義の場合) に設定されます。`vexpr1` が未定義または空の文字列の場合、この操作は `vexpr2` のコピーを返します。

4 番目の引数が指定されている場合、この操作は、`bitexpr` で true の値に対応する位置に対してのみ実行されます。返されるベクトルの長さは、`vexpr1` の長さ、または `bitexpr` が true の値を持ち、`vexpr2` が定義されている最後の位置の、いずれか大きい方です。

オプションの `bitexpr` 引数が含まれているが、未定義または空の文字列である場合、この操作は、`bitexpr` が false の値のみを持つと想定し、`vexpr1` のコピーを返します。`vexpr2` 内のある位置が、明示的に未定義であるか、`bitexpr` が `vexpr2` より長い場合、未定義である場合、結果のベクトルは、`bitexpr` が true の値を持つ位置で未定義の値を持ちます。`vexpr1` が未定義または空の文字列の場合、この操作は、`vexpr2` と同じ型で、指定された `bitexpr` が true の値を持つすべての位置が `vexpr2` の対応する位置の要素の値に設定されたベクトルを返します。

### ObjectScript

```
// vec1 = <1,,3,4,,6>
// vec2 = <,2,10,11,,12>
zwrite $vectorop("vset",vec1,vec2) // writes <1,2,10,11,,12>
```

## 情報操作

- ・ `$vectorop("bytesize", vexpr)` は、バイト数で表した `vexpr` の現在のサイズを返します。`vexpr` が未定義の場合、この操作は `<UNDEFINED>` エラーを発生させます。`vexpr` が空の文字列であるか、ベクトルでない場合、この操作は `<VECTOR>` エラーを発生させます。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
write $vectorop("bytesize",vec) // writes 320
```

- ・ `$vectorop("type", vexpr)` は `vexpr` の型を文字列で返します。可能な値は、“integer”、“decimal”、“double”、“timestamp”、“string” です。`vexpr` が未定義の場合、この操作は `<UNDEFINED>` エラーを発生させます。`vexpr` が空の文字列であるか、ベクトルでない場合、この操作は `<VECTOR>` エラーを発生させます。

### ObjectScript

```
for i = 1:1:10 set $vector(vec,i,"integer") = i
write $vectorop("",vec) // writes "integer"
```

## 関連項目

- ・ [\\$isvector](#)
- ・ [\\$vector](#)
- ・ [\\$vectordefined](#)
- ・ [\\$factor](#)
- ・ [ビットとビット文字列の操作](#)
- ・ [\\$LIST](#)

## \$VIEW (ObjectScript)

メモリ位置のコンテンツを返します。

### 構文

```
$VIEW(offset,mode,length)
$V(offset,mode,length)
```

### 引数

引数	説明
offset	<p>正の整数: バイト単位で表される、mode で指定されたメモリ領域内のベース・アドレスからのオフセット。解釈はモードに依存します (以下を参照)。</p> <p>-1: <a href="#">プロセス要約情報</a>を返すフラグ。</p>
mode	<p>オプション - ベース・アドレスがデータを位置決めするのに使用されるメモリ領域。既定値は -1 です。</p>
length	<p>オプション - Caché で返されるデータ長 (単位はバイト)。文字 “0” 逆順序の接頭語が含まれる場合もあります。既定値は 1 です。</p> <p><a href="#">プロセス要約情報</a>を返す場合の戻り値の形式のフラグ: 1 = キャレット区切りの文字列 (既定)、2 = \$list 構造の文字列。</p>

### 説明

\$VIEW は、メモリ位置のコンテンツを返します。

表示バッファは、[VIEW](#) コマンドを使用して InterSystems IRIS データベース (IRIS.DAT) から読み取るデータ・ブロックを格納するために使用される特殊メモリ領域です。ブロックを表示バッファに読み取った後、mode 0 と共に \$VIEW 関数を使用して、表示バッファのコンテンツを検査できます。[表示バッファ](#)にアクセスするには、デバイス 63 として開く必要があります。完了したら、デバイス 63 を閉じます。

\$VIEW を使用して[プロセスの要約情報](#)を返すこともできます。

通常、\$VIEW 関数は、InterSystems IRIS データベースとシステム情報のデバッグと修復を行うときに使用されます。

### 引数

#### offset

この引数の値は、mode 引数の値に応じて指定します。以下は、その説明です。

- mode が 0、-1、または -2 のときは、ベース・アドレスからのオフセットとして、0 から始まる正の整数をバイト単位で指定します。
- mode が -3 または正の整数のときは、プロセスのアドレス空間内のアドレスを指定します。値 -1 を使用すると、下記の“[プロセス要約情報](#)”の説明に従って、プロセス状態の要約を取得できます。
- mode が -5 のときは、現行ブロック内のグローバル・ノード数を指定する正の整数を指定します。この場合、奇数値はグローバル参照全体を返し、偶数値はポインタまたはデータを返します。

## mode

mode で利用可能な値は、以下のとおりです。mode が省略されたときの既定値は -1 です。値の中には実装固有なものもあります。実装制限は以下の表に示します。

モード	メモリ管理領域	ベース・アドレス
0	表示バッファ	表示バッファの開始位置
-1	プロセスのパーティション (既定)	パーティションの開始位置
-2	システム・テーブル	システム・テーブルの開始位置
-3	現行プロセスのアドレス空間	0
-5	グローバル参照とデータ	特殊。“ <a href="#">モード -5 の使用法</a> ” を参照。
-6	インターシステムズ用に確保	
-7	整合性の確認ユーティリティによってのみ使用	特殊。
n	n が正の数のときは、実行中のプロセス n のアドレス空間。ここで n はそのプロセスの pid (\$JOB 特殊変数の値) です。offset および length は mode -3 と同様に扱われます。	0

## length

長さ (バイト単位)、またはフラグ文字。解釈は mode および offset の値によって決定します。

- mode が 0、-1、または -2、-3、または正の整数 (pid) で、offset が正の整数の場合は、length 引数は以下のようになります。
  - データの長さを文字列で返す、-1 ～ [最大長](#) の負の整数。\$VIEW は、offset によって示されるアドレスで始まる、指定した文字数を返します。
  - 1 ～ 8 の正の整数は、データの小数値を返します。\$VIEW は、1 ～ 4 の連続したバイトか、offset によって示されるアドレスで始まる連続した 8 バイトを返します。
  - 引用符で囲んだ文字列の 文字 C または P は、32 ビット・システム上の 4 バイト・アドレス、または 64 ビット・システム上の 8 バイト・アドレスを示します。C または P を指定する場合は、整数値の length は指定しません。

逆順でバイト値を返す (下位バイトを最下位アドレスに置く) には、文字 O を length 値に追加し、結果の文字列を二重引用符で囲みます。

length 引数が省略された場合、既定値は 1 です。

- mode が -3 または正の整数 (pid) で offset が -1 のときは、length 引数は要約情報の形式を指定するフラグです。length に 1 を指定すると、この要約を区切り文字で返し、2 を指定するとこの要約を \$LIST 構造として返します。length 引数が省略された場合、既定値は 1 です。
- [mode が -5 のときは](#)、length 引数は指定しないでください。

## VIEW の使用の制限

\$VIEW 関数は、制限付きのシステム機能です。呼び出されるコードが IRISYS データベース内にあるので、コマンドは保護されます。詳細は、“[特別な機能](#)” を参照してください。

# プロセス要約情報

offset が -1 の場合、以下の例に示すように mode -3 を使用して、現在のプロセスのアドレス空間から要約情報を ^ 区切り文字として返すことができます。

## ObjectScript

```
WRITE $VIEW(-1,-3,1)
```

以下のように同じ情報を \$LIST 構造として返すことができます。

## ObjectScript

```
SET infolist = $VIEW(-1,-3,2)
ZWRITE infolist
```

指定されたプロセスのアドレス空間から要約情報を返すには、以下の例に示すようにそのプロセスのプロセス ID (pid) を mode 引数の正の整数として指定します。

## ObjectScript

```
SET pid=$PIECE($IO,"|",4)
WRITE $VIEW(-1,pid,1)
```

以下のターミナルの例は、dev フィールドの現在開いているデバイスを複数返します。これは、最初に現在のプロセスのみを返します。続いてスプール・デバイス (デバイス 2) を開き、開いているデバイスをコンマ区切りリストとして返します。

## Terminal

```
USER>WRITE $VIEW(-1,-3)
8484^*^|TRM|:|8484*,^116^...
USER>OPEN 2:(3:12)
```

```
USER>WRITE $VIEW(-1,-3)
8484^*^|TRM|:|8484*,2,^118^...
```

要約情報の返り値は、以下の形式で返されます。

pid^mode^dev^mem^dir^rou^stat^prio^uic^loc^blk^^^defns^lic^jbstat^mempeak^roles^loginroles

フィールドは以下のように定義されます。

フィールド	説明
pid	プロセス ID。クラス SYS.Process の Pid プロパティを参照してください。
mode	ターミナル・プロンプトで実行する場合、*。ジョブがコールイン接続の一部である場合、+ または -。デーモンの場合は省略されます。
dev	コンマ区切りリストとして返される現在の開いているデバイス。現在のデバイス (\$IO デバイス) はアスタリスク (*) 接尾語によって示されます。SYS.Process クラスの OpenDevices プロパティを参照してください。dev の値には末尾のコンマが含まれ、OpenDevices の値には含まれないことに注意してください。
mem	プロセスがデーモンでない場合は、プロセスのパーティション内で使用されているメモリ量 (KB 単位)。SYS.Process クラスの MemoryUsed プロパティに似ていますが、同一ではありません。
dir	既定のディレクトリ。
rou	ルーチン名。



フィールド	説明
stat	コンマ区切りの整数のカウントのペア bol,gcnt。bol は行の先頭を示すトークンで、実行される行数を指定します。gcnt はグローバル・カウントで、FOR ループと XECUTES コマンドの実行回数の合計を指定します。
prio	ユーザの現在のベース優先度。SYS.Process クラスの Priority プロパティを参照してください。
uic	廃止されました。既定は 0,0。
loc	位置。デーモン・プロセスのみ。
blk	バディ・ブロック・キューに使用可能な 2K ブロックの数。これは、ユーザ・メモリ領域（パーティション領域とも呼ばれる）の最大サイズです。SYS.Process クラスの MemoryAllocated プロパティを参照してください。
defns	既定のネームスペース。SYS.Process クラスの NameSpace プロパティを参照してください。
lic	ライセンス・ビット。
jbstat	ジョブのステータス。上位ビットと下位ビットを表す high,low として指定されます。詳細は、“\$ZJOB” 特殊変数を参照してください。
mempeak	プロセスのピーク・メモリ使用量（キロバイト）。この値は、約 64K です。SYS.Process クラスの MemoryPeak プロパティを参照してください。
roles	プロセスが現在持っているロール。コンマ区切りリストとして返されます。\$ROLES の値と同じです。SYS.Process クラスの Roles プロパティを参照してください。
loginroles	プロセスが開始時に現在持っていたロール。コンマ区切りリストとして返されます。SYS.Process クラスの LoginRoles プロパティを参照してください。

## モード -5 の使用法

表示バッファの現在のブロックにグローバルの一部が含まれている場合、mode に -5 を指定すると、グローバル参照とブロックに含まれている値が返されます。length 引数は、mode -5 では有効ではありません。

-5 の mode を使用した場合、offset 値は、ベース・アドレスのバイト・オフセットではなく、ブロックにあるグローバル・ノードを指定します。奇数値は、グローバル参照全体を返し、偶数値はポインタあるいはデータを返します。

例えば、表示バッファの n 番目のノードのグローバル参照全体を返すには、offset に  $n*2-1$  を指定します。n 番目のノードの値を返すには、 $n*2$  を指定します。ブロックの最終ノードのグローバル参照を返すには、offset に -1 を指定します。

\$VIEW は、照合順（つまり数値）でノードを返します。これは \$ORDER 関数が使用するものと同じ順序です。この順序を想定したコードを記述することにより、表示バッファでグローバルを使用してすばやく順次スキャンを実行できるようになります（ObjectScript ユーティリティの一部は、この方法を使用します）。\$VIEW は、offset が表示バッファの最後のノードを過ぎて位置を指定するとき、NULL 文字列（"）を返します。コードの中にこの値についてのテストを含むようにしてください。

現在のブロックがポインタ・ブロックのときは、戻り値はポインタである InterSystems IRIS ブロック数です。ブロックがデータ・ブロックのときは、戻り値はノードに対応したデータ値です。

現在のブロックがデータ・ブロックで、\$VIEW から <VALUE OUT OF RANGE> エラーが返される場合、そのオフセット位置に格納されている情報が長すぎる文字列であることを意味しています。そのオフセット位置に格納されているデータを取得するには、エラーをトラップし、その前のオフセット位置に格納されているグローバル参照を基準とした間接指定による \$GET を使用します。例えば、 $x = \$VIEW(offset, -5)$  の代わりに、 $x = \$GET(@$VIEW(offset-1, -5))$  を使用します。

\$VIEW が <DATABASE> エラーもしくは <FUNCTION> エラーを出した場合は、ブロックでの情報は有効なグローバル参照でも有効なデータでもないという意味になります。

以下の例は、表示バッファのコンテンツを検査するための、汎用コードを示しています。コードは最初に表示バッファを開いてから、読み取るためのブロック数を入力できるようにします。FOR ループは、現在のブロックすべてのオフセットを繰り返します。\$VIEW 関数は、-5 のモードを使用して、各オフセットで値を返します。WRITE コマンドは、結果の 1 組みのオフセット値を出力します。

ブロックの最後まで行くと、\$VIEW は NULL 文字列(“)を返します。IF コマンドは、この値をテストをして、“End of block”メッセージを書き込みます。その後、QUIT コマンドは、ループを終了して制御をプロンプトに戻し、ユーザが他のブロックで読み取りできるようにします。

## ObjectScript

```
Start OPEN 63
WRITE !,"Opening view buffer."
READ !,"Number of block to read in: ",block QUIT:block="
VIEW block
  FOR i=1:1 {
    SET x=$VIEW(i,-5)
    IF x=" ",i#2 {
      WRITE !,"End of block: ",block
      QUIT }
    WRITE !,"Offset = ",i
    WRITE !,"Value = ",x
  }
GOTO Start+2
CLOSE 63
QUIT
```

グローバル・ブロックについて、通常このルーチンによって作成される出力は、以下のとおりです。

```
Opening view buffer.
Number of block to read in:3720
Offset = 1
Value = ^client(5)
Offset = 2
Value = John Jones
Offset = 3
Value = ^client(5,1)
Offset = 4
Value = 23 Bay Rd./Boston/MA 02049
.
.
.
Offset = 126
Value = ^client(18,1,1)
Offset = 127
Value = Checking/45673/1248.00
End of block: 3720
Number of block to read in:
```

## 逆の順序のバイト値 (ビッグ・エンディアンのみ)

ビッグ・エンディアン・システムでは、length 引数の一部として文字 “O” 接尾語を使用することにより、逆順序でバイト値を返すことができます。length に文字 O を含めて指定すると、\$VIEW は逆順序でバイト値を返します (length は二重引用符で囲まなければなりません)。詳細は、以下の例を参照してください。

## ObjectScript

```
USE IO
FOR Z=0:0 {
  WRITE *-6
  SET NEXTBN=$VIEW(LINKA,0,"3O")
  QUIT:NEXTBN=0 }
```

前述の例では、\$VIEW の length 引数は “3O” です (3 と 文字 O)。ビッグ・エンディアン・システムで実行される場合、これは、逆の順序 (O) の次の 3 バイト (3) の長さを示しています。そのため、\$VIEW はメモリにある位置 (表示バッファ - 0 の mode で示されるように) で開始し、最上位バイト、2 番目の上位バイト、および 3 番目の上位バイトを返します。

リトル・エンディアン・システムでは、文字 “O” は空命令です。“3O” の length 値は、“3” の length 値と同じです。

IsBigEndian() クラス・メソッドを使用して、オペレーティング・システム・プラットフォームでどのビット順序を使用するかを決定できます (1 = ビッグ・エンディアン・ビット順、0 = リトル・エンディアン・ビット順)。

### ObjectScript

```
WRITE $SYSTEM.Version.IsBigEndian()
```

## 関連項目

- ・ [VIEW](#) 関数
- ・ [JOB](#) コマンド

## \$WASCII (ObjectScript)

サロゲート・ペアを認識して、文字に対応する数値コードを返します。

### 構文

```
$WASCII(expression,position)
$WA(expression,position)
```

### 引数

引数	説明
<i>expression</i>	変換する文字
<i>position</i>	オプション - 文字列内での文字の位置。1 から数えます。既定値は 1 です。

### 説明

\$WASCII は *expression* で指定された単一の文字に対し文字コード値を返します。\$WASCII は、サロゲート・ペアを単一の文字として認識します。返り値は正の整数です。

*expression* 引数は、単一文字あるいは文字列に評価されます。*expression* が文字列に評価される場合、オプションの *position* 引数で、変換したい文字の位置を示す必要があります。*position* は、サロゲート・ペアを単一の文字としてカウントします。[\\$WISWIDE](#) 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます (頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます)。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ~ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ~ DFFF の範囲の下位 16 ビット文字要素で構成されます。

\$WASCII 関数は、サロゲート・ペアを単一の文字として認識します。\$ASCII 関数は、サロゲート・ペアを 2 文字として処理します。その他の点では、\$WASCII と \$ASCII は機能的に同じです。ただし、\$ASCII は通常 \$WASCII より高速なため、サロゲート・ペアが出現しない場合は常に \$ASCII が推奨されます。文字から数値コードへの変換の詳細は、[\\$ASCII](#) 関数を参照してください。

### 例

以下の例は、サロゲート・ペアの Unicode 値を返す \$WASCII を示します。

#### ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
WRITE !,$ASCII(hipart)," = high-order value"
WRITE !,$ASCII(lopart)," = low-order value"
SET spair=hipart_lopart /* surrogate pair */
SET xpair=hipart_lopart /* NOT a surrogate pair */
WRITE !,$WASCII(spair)," = surrogate pair value"
WRITE !,$WASCII(xpair)," = Not a surrogate pair"
```

以下の例は、サロゲート・ペアの \$WASCII と \$ASCII の戻り値を比較します。

## ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
WRITE !,$ASCII(hipart)," = high-order value"
WRITE !,$ASCII(lopart)," = low-order value"
SET spair=hipart_lopart /* surrogate pair */
WRITE !,$ASCII(spair)," = $ASCII value for surrogate pair"
WRITE !,$WASCII(spair)," = $WASCII value for surrogate pair"
```

以下の例では、サロゲート・ペアをカウントする position への影響を示します。各 position の \$WASCII と \$ASCII の両方の値が返されます。\$WASCII はサロゲート・ペアを 1 つの位置としてカウントし、\$ASCII はサロゲート・ペアを 2 つの位置としてカウントします。

## ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
WRITE !,$ASCII(hipart)," = high-order value"
WRITE !,$ASCII(lopart)," = low-order value",!
SET str="AB"_lopart_hipart_lopart_"CD"_hipart_lopart_"EF"
FOR x=1:1:11 {
WRITE !,"position ",x," $WASCII ", $WASCII(str,x)," $ASCII ", $ASCII(str,x) }
```

## 関連項目

- ・ [\\$ASCII](#) 関数
- ・ [\\$WCHAR](#) 関数
- ・ [\\$WEXTRACT](#) 関数
- ・ [\\$WFIND](#) 関数
- ・ [\\$WISWIDE](#) 関数
- ・ [\\$WLENGTH](#) 関数
- ・ [\\$WREVERSE](#) 関数

## \$WCHAR (ObjectScript)

サロゲート・ペアを認識して、数値コードに対応する文字を返します。

### 構文

```
$WCHAR(expression,...)  
$WC(expression,...)
```

### 引数

引数	説明
expression	変換される整数値

### 説明

\$WCHAR は expression で指定したコード値に対応する文字を返します。10 進数値 65535 (16 進数 FFFF) およびそれより小さい値は、\$CHAR および \$WCHAR によって同様に処理されます。65536 (16 進数 10000) から 1114111 (16 進数 10FFFF) は、Unicode のサロゲート・ペアを表します。これらの文字は \$WCHAR を使用して返されます。

expression にコンマで区切られたリストのコード値が含まれている場合、\$WCHAR は文字列として対応する文字を返します。\$WCHAR は、サロゲート・ペアを単一の文字として認識します。[\\$WISWIDE](#) 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます (頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます)。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ~ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ~ DFFF の範囲の下位 16 ビット文字要素で構成されます。

\$WCHAR 関数は、サロゲート・ペアを単一の文字として処理します。\$CHAR 関数は、サロゲート・ペアを 2 文字として処理します。その他の点では、\$WCHAR と \$CHAR は機能的に同じです。ただし、\$CHAR は通常 \$WCHAR より高速なため、サロゲート・ペアが出現しない場合は常に \$CHAR が推奨されます。

数値コードから文字への変換の詳細は、[\\$CHAR](#) 関数を参照してください。

### 関連項目

- [\\$CHAR](#) 関数
- [\\$WASCII](#) 関数
- [\\$WEXTRACT](#) 関数
- [\\$WFIND](#) 関数
- [\\$WISWIDE](#) 関数
- [\\$WLENGTH](#) 関数
- [\\$WREVERSE](#) 関数

## \$WEXTRACT (ObjectScript)

指定された位置にある部分文字列を文字列から取り出すか、または指定された位置にある部分文字列を置換し、サロゲート・ペアを認識します。

### 構文

```
$WEXTRACT(string,from,to)
$WE(string,from,to)

SET $WEXTRACT(string,from,to)=value
SET $WE(string,from,to)=value
```

### 引数

引数	説明
string	部分文字列が識別されるターゲット文字列。string には、評価結果が引用符で囲んだ文字列または数値になる式を指定します。SET \$WEXTRACT 構文では、string は変数または多次元プロパティにする必要があります。
from	<p>オプション - 対象の文字列内の開始位置を指定します。文字は 1 からカウントされます。サロゲート・ペアは単一の文字としてカウントされます。許可される値は、n (string の先頭からの文字カウントで開始位置を指定する正の整数)、* (string の末尾文字の指定)、および *-n (string の末尾から逆向きの文字のオフセット整数カウント) です。SET \$WEXTRACT 構文は、*+n (string の末尾の先に追加する文字のオフセット整数カウント) もサポートします。指定されていない場合、既定は 1 です。2 引数形式 \$WEXTRACT(string,from) および 3 引数形式 \$WEXTRACT(string,from,to) には、さまざまな値が使用されます。</p> <p>to なし：単一の文字を指定します。string の先頭からカウントするには、正の整数に評価する式 (1 からカウントする) を指定します。ゼロ (0) または負の数を指定すると、空の文字列が返されます。string の末尾からカウントするには、*、または *-n を指定します。from が省略されている場合は、既定値の 1 が使用されます。</p> <p>to あり：文字の範囲の先頭を指定します。string の先頭からカウントするには、正の整数に評価する式 (1 からカウントする) を指定します。ゼロ (0) または負の数字は 1 として評価されます。string の末尾からカウントするには、*、または *-n を指定します。</p>
to	<p>オプション - 文字の範囲の終了位置 (その文字を含む) を指定します。from と共に使用する必要があります。許可される値は、n (string の先頭から文字カウントで終了位置を指定する、from に指定された値以上の正の整数)、* (string の末尾文字の指定)、および *-n (string の末尾からの逆向きでの文字のオフセット整数カウント) です。サロゲート・ペアは単一の文字としてカウントされます。文字列の末尾を超える to 値を指定できます。</p> <p>SET \$WEXTRACT 構文は、*+n (string の末尾の先に追加する文字範囲の終了位置のオフセット整数カウント) もサポートします。</p>

### 説明

\$WEXTRACT は、string の先頭から文字をカウントするか、または string の末尾からのオフセットで文字をカウントすることで、指定された位置での string 内の部分文字列を識別します。部分文字列には、1 文字または文字の範囲を指定できます。\$WEXTRACT は、サロゲート・ペアを単一の文字として認識します。

\$EXTRACT には、以下の 2 つの使用方法があります。

- string から部分文字列を返します。これには構文 \$WEXTRACT(string,from,to) が使用されます。



- string 内の部分文字列を置き換えます。置換部分文字列は、元の部分文字列と同じ長さでも、長くても、短くてもかまいません。これには構文 `SET $WEXTRACT(string,from,to)=value` が使用されます。

\$WEXTRACT と \$EXTRACT は機能的に同じです。ただしサロゲート・ペアの処理が異なります。

## サロゲート・ペア

\$WEXTRACT の from 引数と to 引数は、サロゲート・ペアを単一の文字としてカウントします。[\\$WISWIDE](#) 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます (頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます)。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ~ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ~ DFFF の範囲の下位 16 ビット文字要素で構成されます。

\$WEXTRACT 関数は、サロゲート・ペアを単一の文字として処理します。\$EXTRACT 関数は、サロゲート・ペアを 2 文字として処理します。文字列にサロゲート・ペアが含まれていない場合、\$WEXTRACT と \$EXTRACT のいずれかを使用できます。どちらも同じ値が返されます。ただし、\$EXTRACT は通常 \$WEXTRACT より高速なため、サロゲート・ペアが出現しない場合は常に \$EXTRACT が推奨されます。部分文字列の抽出の詳細は、[\\$EXTRACT](#) 関数を参照してください。

## 部分文字列を返す方法

\$WEXTRACT は、string から文字位置により部分文字列を返します。この部分文字列の抽出の特性は、使用する引数によって決まります。

- \$WEXTRACT(string) は、文字列の先頭文字を抽出します。
- \$WEXTRACT(string,from) は、from で指定した位置の 1 文字を抽出します。from の値は、文字列の最初からカウントした整数の文字数、文字列の最後の文字を指定するアスタリスク、または、文字列の最後から逆方向にカウントして指定する負の整数を持つアスタリスクになります。

以下の例では、サロゲート・ペアを含む文字列から 1 つの文字を抽出します。\$LENGTH はサロゲート・ペアを 2 文字とカウントしますが、\$WEXTRACT はサロゲート・ペアを 1 文字とカウントします。

### ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
SET spair=hipart_lopart /* surrogate pair */
WRITE "length of surrogate pair ", $LENGTH(spair), !
SET mystr="AB"_spair_"DEFG"
WRITE !, $WEXTRACT(mystr,4)      // "D" the 4th character
WRITE !, $WEXTRACT(mystr,*)      // "G" the last character
WRITE !, $WEXTRACT(mystr,*-5)    // "B" the offset 5 character from end
WRITE !, $WEXTRACT(mystr,*-0)    // "G" the last character by 0 offset
```

- \$WEXTRACT(string,from,to) は、from の位置から始まり、to の位置で終了する文字列の範囲 (その位置の文字を含む) を抽出します。以下の \$WEXTRACT 関数は、サロゲート・ペアを単一文字とカウントして、両方とも文字列 “Alabama” を返します。

### ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
SET spair=hipart_lopart /* surrogate pair */
SET var2=spair_"XXX"_spair_"Alabama"_spair
WRITE !, $WEXTRACT(var2,6,12)
WRITE !, $WEXTRACT(var2,*-7,*-1)
```

from と to の位置が同じ場合、\$WEXTRACT は単一の文字を返します。to の位置が from の位置よりも文字列の先頭に近い場合、\$WEXTRACT は NULL 文字列を返します。

## 部分文字列の置換

\$WEXTRACT を SET コマンドと一緒に使用して、指定した文字または文字列の範囲を別の値で置き換えることができます。また、文字列の末尾に文字を追加する場合にも使用できます。SET \$WEXTRACT はサロゲート・ペアを単一の文字として認識します。

等号の左側で SET と共に \$WEXTRACT を使用する場合、string は有効な変数名にすることができます。変数が存在しない場合は、SET \$WEXTRACT が変数を定義します。string 引数は、[多次元プロパティ](#)参照にすることもできますが、非多次元オブジェクト・プロパティにすることはできません。非多次元オブジェクト・プロパティで SET \$WEXTRACT を使用しようとする、<OBJECT DISPATCH> エラーが発生します。

\$WEXTRACT (または \$EXTRACT、\$PIECE、あるいは \$LIST) を伴った SET (a,b,c,...)=value 構文は、その関数で相対オフセット構文 \* (文字列の末尾を表す) と \*-n または \*+n (文字列の末尾からの相対オフセットを表す) を使用する場合、等号の左側では使用できません。その代わりに、SET a=value,b=value,c=value,... 構文を使用する必要があります。

部分文字列の置換の詳細は、"[\\$EXTRACT](#)" 関数を参照してください。

## 例

以下の例は、サロゲート・ペアの Unicode 値を返す 2 引数形式の \$WEXTRACT を示します。

### ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
SET spair=hipart_lopart /* surrogate pair */
SET x="ABC"_spair_"DEFGHIJK"
WRITE !,"$EXTRACT character "
ZZDUMP $EXTRACT(x,4)
WRITE !,"$WEXTRACT character "
ZZDUMP $WEXTRACT(x,4)
```

以下の例は、部分文字列の範囲にサロゲート・ペアを含む 3 引数形式の \$WEXTRACT を示します。

### ObjectScript

```
SET hipart=$CHAR($ZHEX("D806"))
SET lopart=$CHAR($ZHEX("DC06"))
SET spair=hipart_lopart /* surrogate pair */
SET x="ABC"_spair_"DEFGHIJK"
WRITE !,"$EXTRACT two characters "
ZZDUMP $EXTRACT(x,3,4)
WRITE !,"$WEXTRACT two characters "
ZZDUMP $WEXTRACT(x,3,4)
```

## 関連項目

- ・ [\\$EXTRACT](#) 関数
- ・ [\\$WASCI](#) 関数
- ・ [\\$WCHAR](#) 関数
- ・ [\\$WFIND](#) 関数
- ・ [\\$WISWIDE](#) 関数
- ・ [\\$WLENGTH](#) 関数
- ・ [\\$WREVERSE](#) 関数

## \$WFIND (ObjectScript)

サロゲート・ペアを認識して、値を基準に部分文字列を検索し、文字列の最終位置を指定する整数を返します。

### 構文

```
$WFIND(string,substring,position)
$WF(string,substring,position)
```

### 引数

引数	説明
string	検索されるターゲット文字列。変数名、数値、文字列リテラル、または文字列に解決される有効な ObjectScript 式を指定できます。
substring	検索される部分文字列。変数名、数値、文字列リテラル、または文字列に解決される有効な ObjectScript 式を指定できます。
position	オプション - 検索を開始するターゲット文字列内の位置。必ず正の整数を指定します。

### 説明

\$WFIND 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。位置の計算では、各サロゲート・ペアは単一の文字としてカウントされます。\$WFIND は \$FIND と機能的に同じですが、\$WFIND がサロゲート・ペアを認識する点が異なります。サロゲート・ペアを単一の文字として認識します。[\\$WISWIDE](#) 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます（頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます）。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ～ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ～ DFFF の範囲の下位 16 ビット文字要素で構成されます。

\$WFIND 関数は、サロゲート・ペアを単一の文字としてカウントします。\$FIND 関数は、サロゲート・ペアを 2 文字としてカウントします。その他の点では、\$WFIND と \$FIND は機能的に同じです。ただし、\$FIND は通常 \$WFIND より高速なため、サロゲート・ペアが出現しない場合は常に \$FIND が推奨されます。

部分文字列の検索の詳細は、[\\$FIND](#) 関数を参照してください。

### 例

以下の例は、\$WFIND が戻り値の単一の文字としてサロゲート・ペアをカウントする方法を示します。

#### ObjectScript

```
SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
SET str="ABC_spair_DEF"
WRITE !,$FIND(str,"DE")," $FIND location in string"
WRITE !,$WFIND(str,"DE")," $WFIND location in string"
```

以下の例は、\$WFIND で position 引数の単一の文字としてサロゲート・ペアをカウントする方法を示します。

## ObjectScript

```
SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
SET str="ABC"_spair_"DEF"
WRITE !,$FIND(str,"DE",6)," $FIND location in string"
WRITE !,$WFIND(str,"DE",6)," $WFIND location in string"
```

## 関連項目

- ・ [\\$FIND](#) 関数
- ・ [\\$WASCII](#) 関数
- ・ [\\$WCHAR](#) 関数
- ・ [\\$WEXTRACT](#) 関数
- ・ [\\$WISWIDE](#) 関数
- ・ [\\$WLENGTH](#) 関数
- ・ [\\$WREVERSE](#) 関数

## \$WISWIDE (ObjectScript)

文字列にサロゲート・ペアが含まれているかどうかを示すフラグを返します。

### 構文

```
$WISWIDE(string)
```

### 引数

引数	説明
string	文字列または文字列に評価する式

### 説明

\$WISWIDE は、string にサロゲート・ペアが含まれているかどうかを示すブーリアン値を返します。0=string は、サロゲート・ペアを含んでいません。1=string は、1 つ以上のサロゲート・ペアを含んでいます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます (頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます)。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ~ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ~ DFFF の範囲の低位 16 ビット文字要素で構成されます。

### 例

以下の例は、サロゲート・ペアのブーリアン値を返す \$WISWIDE を示します。

#### ObjectScript

```
SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06")) /* surrogate pair */
SET xpair=$CHAR($ZHEX("DC06"),$ZHEX("D806")) /* NOT a surrogate pair */
SET str="AB"_spair_"CD"
WRITE !,$WISWIDE(str)," = surrogate pair(s) in string?"
SET xstr="AB"_xpair_"CD"
WRITE !,$WISWIDE(xstr)," = surrogate pair(s) in string?"
```

### 関連項目

- [\\$WASCII 関数](#)
- [\\$WCHAR 関数](#)
- [\\$WEXTRACT 関数](#)
- [\\$WFIND 関数](#)
- [\\$WLENGTH 関数](#)
- [\\$WREVERSE 関数](#)

## \$WLENGTH (ObjectScript)

サロゲート・ペアを認識して、文字列の文字数を返します。

### 構文

```
$WLENGTH(string)
$WL(string)
```

### 引数

引数	説明
string	文字列または文字列に評価する式

### 説明

\$WLENGTH は、string の文字数を返します。\$WLENGTH は \$LENGTH と機能的に同じですが、\$WLENGTH がサロゲート・ペアを認識する点が異なります。サロゲート・ペアを単一の文字として認識します。[\\$WISWIDE](#) 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます（頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます）。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ～ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ～ DFFF の範囲の下位 16 ビット文字要素で構成されます。

\$WLENGTH 関数は、サロゲート・ペアを単一の文字としてカウントします。\$LENGTH 関数は、サロゲート・ペアを 2 文字としてカウントします。その他の点では、\$WLENGTH と \$LENGTH は機能的に同じです。ただし、\$LENGTH は通常 \$WLENGTH より高速なため、サロゲート・ペアが出現しない場合は常に \$LENGTH が推奨されます。

文字列の長さの詳細は、[\\$LENGTH](#) 関数を参照してください。

### 例

以下の例は、\$WLENGTH が単一の文字としてサロゲート・ペアをカウントする方法を示します。

#### ObjectScript

```
SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
SET str="AB" _spair_ "CD"
WRITE !,$LENGTH(str)," $LENGTH characters in string"
WRITE !,$WLENGTH(str)," $WLENGTH characters in string"
```

### 関連項目

- [\\$LENGTH](#) 関数
- [\\$WASCII](#) 関数
- [\\$WCHAR](#) 関数
- [\\$WEXTRACT](#) 関数
- [\\$WFIND](#) 関数
- [\\$WISWIDE](#) 関数

- ・ [\\$WREVERSE](#) 関数



# \$WREVERSE (ObjectScript)

サロゲート・ペアを認識して、文字列にある文字を逆の順に返します。

## 構文

```
$WREVERSE(string)
$WRE(string)
```

## 引数

引数	説明
string	文字列または文字列に評価する式

## 説明

\$WREVERSE は、string 内の文字を逆の順序で返します。\$WREVERSE は \$REVERSE と機能的に同じですが、\$WREVERSE がサロゲート・ペアを認識する点が異なります。[\\$WISWIDE](#) 関数を使用して、文字列にサロゲート・ペアが含まれているかどうかを判断することができます。

サロゲート・ペアは、単一の Unicode 文字と一緒にエンコードする 16 ビットの InterSystems IRIS 文字要素のペアです。サロゲート・ペアは、中国語、日本語の漢字、韓国語のハンジャ文字で使用されている特定の表意文字を表すために使用されます（頻繁に使用される中国語、漢字、およびハンジャ文字は、標準 16 ビットの Unicode エンコードで表されます）。サロゲート・ペアにより、InterSystems IRIS は日本語 JIS X0213:2004 (JIS2004) エンコーディング標準および中国語 GB18030 エンコーディング標準をサポートします。

サロゲート・ペアは、16 進数の D800 ～ DBFF の範囲の高位 16 ビット文字要素と、16 進数の DC00 ～ DFFF の範囲の下位 16 ビット文字要素で構成されます。

\$WREVERSE 関数は、サロゲート・ペアを単一の文字としてカウントします。\$REVERSE 関数は、サロゲート・ペアを 2 文字として処理します。その他の点では、\$WREVERSE と \$REVERSE は機能的に同じです。ただし、\$REVERSE は通常 \$WREVERSE より高速なため、サロゲート・ペアが出現しない場合は常に \$REVERSE が推奨されます。

文字列の反転の詳細は、[\\$REVERSE](#) 関数を参照してください。

## 例

以下の例は、\$WREVERSE が単一の文字としてサロゲート・ペアを処理する方法を示します。

### ObjectScript

```
SET spair=$CHAR($ZHEX("D806"),$ZHEX("DC06"))
SET str="AB" _spair_ "CD"
WRITE !,"String before reversing:"
ZZDUMP str
SET wrev=$WREVERSE(str)
WRITE !,"$WREVERSE did not reverse surrogate pair:"
ZZDUMP wrev
SET rev=$REVERSE(str)
WRITE !,"$REVERSE reversed surrogate pair:"
ZZDUMP rev
```

## 関連項目

- [\\$REVERSE](#) 関数
- [\\$WASCII](#) 関数
- [\\$WCHAR](#) 関数

- ・ [\\$WEXTRACT](#) 関数
- ・ [\\$WFIND](#) 関数
- ・ [\\$WISWIDE](#) 関数
- ・ [\\$WLENGTH](#) 関数

## \$EXECUTE (ObjectScript)

指定されたコマンド行を実行します。

### 構文

```
$EXECUTE(code,paramlist)
```

### 引数

引数	説明
<i>code</i>	引用符で囲んだ文字列で指定される、有効な ObjectScript コマンド行に解決される式。コマンド行には、1 つ以上の ObjectScript コマンドを含めることができます。最後のコマンドは、引数付きの QUIT である必要があります。
<i>paramlist</i>	オプション - <i>code</i> に渡すパラメータのリスト。複数のパラメータは、コンマで区切られます。

### 説明

\$EXECUTE 関数では、ユーザ記述の *code* を関数として実行し、渡されたパラメータを指定して値を返すことができます。*code* 引数は、1 つ以上の ObjectScript コマンドを含む引用符付き文字列に評価される必要があります。*code* の実行は、引数を返す QUIT コマンドで完了する必要があります。InterSystems IRIS は、この QUIT 引数を \$EXECUTE の返りコード値として返します。

*paramlist* 引数を使用して、パラメータを *code* に渡すことができます。パラメータを渡す場合、*code* の先頭に仮パラメータ・リストが記述されている必要があります。パラメータは、位置指定されます。*code* に記載される仮パラメータの数は、*paramlist* で指定する実パラメータの数以上である必要があります。

%Library.Routine クラスの CheckSyntax() メソッドを使用すると、*code* の構文チェックを実行できます。

\$EXECUTE の各呼び出しは、新しいコンテキスト・フレームをプロセスのコール・スタックに配置します。\$STACK 特殊変数は、コール・スタックのコンテキスト・フレームの現在の番号を含みます。

\$EXECUTE 関数は、実質的に EXECUTE コマンドと同じ処理を行います。ただし、\$EXECUTE 関数は後置条件や複数のコマンド行引数の使用をサポートしないという点が異なります。\$EXECUTE 関数では、各実行パスが引数付きの QUIT で終了する必要があります。EXECUTE コマンドでは、QUIT は不要であり、引数付きの QUIT が許可されることもありません。

### 引数

#### code

引用符で囲んだ文字列で指定される、有効な ObjectScript コマンド行に評価される式。*code* 文字列には、最初にタブ文字、または最後に <Return> が含まれていてはなりません。文字列は、有効な ObjectScript プログラム行より長くてもなりません。*code* の文字列には、可能な各実行パスの最後に、引数を返す QUIT コマンドを含める必要があります。

\$EXECUTE がパラメータを *code* に渡す場合、*code* 文字列は仮パラメータ・リストで開始する必要があります。仮パラメータ・リストは括弧で囲み、括弧内のパラメータはコンマで区切ります。

#### paramlist

*code* に渡すパラメータのリスト。コンマ区切りリストとして指定します。*paramlist* の各パラメータは、*code* 文字列内の仮パラメータに対応する必要があります。*paramlist* のパラメータ数は、*code* に記載された仮パラメータの数以下にすることができます。

前にドットを付けることで、参照によってパラメータを渡すことができます。これは、code から値を渡すのに便利です。例は以下のようになります。詳細は、“[参照渡し](#)”を参照してください。

## 例

以下の例では、\$EXECUTE 関数は cmdline で指定されたコマンド行を実行します。これは、num1 と num2 の 2 つのパラメータをこのコマンド行に渡します。

### ObjectScript

```
SET cmd="(dvnd,dvsr) IF dvsr=0 {QUIT 99} ELSE {SET ^testnum=dvnd/dvsr QUIT 0}"
SET rtn=$EXECUTE(cmd,num1,num2)
IF rtn=99
    {WRITE !,"Division by zero. ^testnum not set"}
ELSE
    {WRITE !,"global ^testnum set to",^testnum}
```

以下の例では、参照渡し (.y) を使用して、code から呼び出し元のコンテキストにローカル変数値を渡します。

### ObjectScript

```
CubeIt
SET x=7
SET rtn=$EXECUTE("(in,out) SET out=in*in*in QUIT 0",x,.y)
IF rtn=0 {WRITE !,x," cubed is ",y}
ELSE {WRITE !,"Error code=",SQLCODE}
```

次の例は、\$EXECUTE がどのように \$STACK 特殊変数をインクリメントするかを示しています。この例では、\$EXECUTE の内部から \$STACK の値を書き込むか、または \$EXECUTE から EXECUTE コマンドを呼び出して \$STACK の値を書き込みます。

### ObjectScript

```
StackIt
SET stackit=$RANDOM(3)
IF stackit=0 {GOTO StackIt}
WRITE "initial stack level ",$STACK,!
SET cmd="(stackit) IF stackit=1 {WRITE ""stack is """,$STACK,! QUIT 1} "_
    "ELSEIF stackit=2 {WRITE ""stack is """,$STACK EXECUTE ""WRITE """" stack is """"",$STACK,!}"
QUIT 1} "_
    "ELSE { QUIT 0}"
SET rtn=$EXECUTE(cmd,stackit)
IF rtn=1 { WRITE "return stack level ",$STACK }
ELSE {WRITE "unexpected value: rtn=",rtn}
```

## 関連項目

- [DO コマンド](#)
- [EXECUTE コマンド](#)
- [QUIT コマンド](#)
- [\\$STACK 特殊変数](#)

# \$ZABS (ObjectScript)

絶対値の関数です。

## 構文

`$ZABS(n)`

## 引数

引数	説明
<i>n</i>	任意の数字

## 概要

\$ZABS は、*n* の絶対値を返します。

## 引数

*n*

任意の数字。値、変数、式として指定できます。式は評価され、結果を正の値に変換します。複数のプラス記号やマイナス記号を使用できます。先頭のゼロや末尾のゼロは削除されます。

非数値文字列は0として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

## 例

以下の例は、ユーザが指定された数の絶対値を返します。

### ObjectScript

```
READ "Input a number: ",num
SET abs=$ZABS(num)
WRITE "The absolute value of ",num," is ",abs
```

## 関連項目

- ・ [演算子](#)

## \$ZARCCOS (ObjectScript)

逆 (アーク) コサイン関数です。

### 構文

`$ZARCCOS (n)`

### 引数

引数	説明
n	符号付き 10 進数

### 概要

\$ZARCCOS は、n の三角関数の逆 (アーク) コサインを返します。結果は、ラジアンで返されます (小数点第 18 位まで)。

### 引数

n

1 から -1 までの範囲の符号付き 10 進数です。値、変数、式として指定することができます。範囲外の数値では、`<ILLEGAL VALUE>` エラーが発生します。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

以下は、\$ZARCCOS によって返されるアーク・コサイン値です。

n	返されるアーク・コサイン
1	これは、0 を返します。
0	これは、1.570796326794896619 を返します。
-1	これは、pi (3.141592653589793238) を返します。

### 例

以下の例では、数のアーク・コサインとアーク・サインの比較を行います。

#### ObjectScript

```

READ "Input a number: ",num
IF num>1 { WRITE !,"ILLEGAL VALUE: number too big" }
ELSEIF num<-1 { WRITE !,"ILLEGAL VALUE: number too small" }
ELSE {
    WRITE !,"the arc cosine is: ",$ZARCCOS(num)
    WRITE !,"the arc sine is: ",$ZARCSIN(num)
}
QUIT

```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。どちらの場合も、1 のアーク・コサインは正確に 0 になります。

## ObjectScript

```
WRITE !,"the arc cosine is: ",$ZARCCOS(0.0)
WRITE !,"the arc cosine is: ",$ZARCCOS($DOUBLE(0.0))
WRITE !,"the arc cosine is: ",$ZARCCOS(1.0)
WRITE !,"the arc cosine is: ",$ZARCCOS($DOUBLE(1.0))
WRITE !,"the arc cosine is: ",$ZARCCOS(-1.0)
WRITE !,"the arc cosine is: ",$ZARCCOS($DOUBLE(-1.0))
```

## 関連項目

- ・ [\\$ZCOS](#) 関数
- ・ [\\$ZPI](#) 特殊変数



## \$ZARCSIN (ObjectScript)

逆 (アーク) サイン関数です。

### 構文

`$ZARCSIN(n)`

### 引数

引数	説明
n	符号付き 10 進数

### 概要

\$ZARCSIN は、n の三角関数の逆 (アーク) サインを返します。結果はラジアンで返されます。

### 引数

n

1 から -1 までの範囲の符号付き 10 進数です。値、変数、式として指定することができます。範囲外の数値では、`<ILLEGAL VALUE>` エラーが発生します。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

以下は、\$ZARCSIN によって返されるアーク・サイン値です。

n	返されるアーク・サイン
1	これは、1.570796326794896619 を返します。
0	これは、0 を返します。
-1	これは、-1.570796326794896619 を返します。

### 例

以下の例では、数のアーク・サインとアーク・コサインの比較を行います。

#### ObjectScript

```
READ "Input a number: ",num
IF num>1 { WRITE !,"ILLEGAL VALUE: number too big" }
ELSEIF num<-1 { WRITE !,"ILLEGAL VALUE: number too small" }
ELSE {
    WRITE !,"the arc sine is: ",$ZARCSIN(num)
    WRITE !,"the arc cosine is: ",$ZARCCOS(num)
}
QUIT
```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。どちらの場合も、0 のアーク・サインは正確に 0 になります。

## ObjectScript

```
WRITE !,"the arc sine is: ",$ZARCSIN(0.0)
WRITE !,"the arc sine is: ",$ZARCSIN($DOUBLE(0.0))
WRITE !,"the arc sine is: ",$ZARCSIN(1.0)
WRITE !,"the arc sine is: ",$ZARCSIN($DOUBLE(1.0))
WRITE !,"the arc sine is: ",$ZARCSIN(-1.0)
WRITE !,"the arc sine is: ",$ZARCSIN($DOUBLE(-1.0))
```

## 関連項目

- [\\$ZSIN](#) 関数
- [\\$ZPI](#) 特殊変数

## \$ZARCTAN (ObjectScript)

逆 (アーク) タンジェント関数です。

### 構文

`$ZARCTAN(n)`

### 引数

引数	説明
<i>n</i>	任意の正や負の数字です。

### 概要

`$ZARCTAN` は、*n* の三角関数の逆 (アーク) タンジェントを返します。結果は、1.57079 (pi の半分) から 0、0 から -1.57079 です。結果はラジアンで返されます。

### 引数

*n*

任意の正や負の数字です。値、変数、式として指定することができます。`$ZPI` 特殊変数を使用して、pi を指定することができます。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

以下は、`$ZARCTAN` によって返されるアーク・タンジェント値です。

<i>n</i>	返されるアーク・タンジェント
2	これは、1.107148717794090502 を返します。
1	これは、.7853981633974483098 を返します。
0	これは、0 を返します。
-1	これは、-.7853981633974483098 を返します。

### 例

以下は、数のアーク・タンジェントの計算を許可する例です。

#### ObjectScript

```
READ "Input a number: ",num
WRITE !,"the arc tangent is: ",$ZARCTAN(num)
QUIT
```

以下の例は、InterSystems IRIS 小数 (`$DECIMAL` の数値) の結果と `$DOUBLE` の数値の結果を比較します。どちらの場合も、pi/2 のアーク・タンジェントは 1 ではない小数値ですが、0 のアーク・タンジェントは 0 です。

## ObjectScript

```
WRITE !,"the arc tangent is: ",$ZARCTAN(0.0)
WRITE !,"the arc tangent is: ",$ZARCTAN($DOUBLE(0.0))
WRITE !,"the arc tangent is: ",$ZARCTAN($ZPI)
WRITE !,"the arc tangent is: ",$ZARCTAN($DOUBLE($ZPI))
WRITE !,"the arc tangent is: ",$ZARCTAN($ZPI/2)
WRITE !,"the arc tangent is: ",$ZARCTAN($DOUBLE($ZPI)/2)
```

## 関連項目

- ・ [\\$ZTAN](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZBOOLEAN (ObjectScript)

ビット単位論理演算関数

### 構文

```
$ZBOOLEAN(arg1,arg2,bit_op)
$ZB(arg1,arg2,bit_op)
```

### 引数

引数	説明
arg1	最初の引数。整数または文字列、あるいは、整数または文字列に解析される変数または式のいずれかです。すべての文字は 0 ～ 255 の ASCII 値 である必要があります。浮動小数点数は使用できません。
arg2	2 番目の引数。整数または文字列、あるいは、整数または文字列に解析される変数または式のいずれかです。すべての文字は 0 ～ 255 の ASCII 値 である必要があります。浮動小数点数は使用できません。
bit_op	実行される演算を示す整数（以下のテーブルを参照してください）。許可される値は、0 から 15（包含的）です。

### 概要

\$ZBOOLEAN は、2 つの引数 arg1 と arg2 に対して、bit\_op によって指定されたビット単位論理演算を実行します。  
\$ZBOOLEAN は、bit\_op 値で指定されたように arg1 と arg2 のビット単位結合の結果を返します。ZZDUMP コマンドを使用して、結果を表示することができます。

\$ZBOOLEAN は、文字列または数字のいずれかに対して演算を実行します。文字列の場合、文字列内の各文字に対して、論理積と論理和を実行します。数字の場合、数字全体を 1 つの単位として、論理積と論理和を実行します。数値文字列を数字に評価させるには、文字列の前にプラス符号 (+) を付けます。

**注釈** \$ZBOOLEAN は ASCII 255 より大きな値の Unicode 文字をサポートしません。\$ZBOOLEAN を 16 ビットの Unicode 文字の文字列に適用するには、まず \$ZWUNPACK を使用し、次に \$ZBOOLEAN、さらに \$ZWPACK を続ける必要があります。

\$ZBOOLEAN と \$BITLOGIC が使用するデータ形式は異なります。一方の結果を、もう一方の入力として使用することはできません。

このビット単位演算では、arg1 と arg2 のブーリアンの組み合わせが 16 種類考えられます。以下の表で、その組み合わせを示しています。

bit_op のビット・マスク	実行される演算
0	0
1	arg1 & arg2 (論理 AND)
2	arg1 & ~arg2
3	arg1
4	~arg1 & arg2
5	arg2

bit_op のビット・マスク	実行される演算
6	$\text{arg1} \wedge \text{arg2}$ (論理 XOR (排他的 or))
7	$\text{arg1} ! \text{arg2}$ (論理 OR (包含 or))
8	$\sim(\text{arg1} ! \text{arg2})$
9	$\sim(\text{arg1} \wedge \text{arg2})$
10	$\sim \text{arg2}$ (論理 NOT)
11	$\text{arg1} ! \sim \text{arg2}$
12	$\sim \text{arg1}$ (論理 NOT)
13	$\sim \text{arg1} ! \text{arg2}$
14	$\sim(\text{arg1} \& \text{arg2})$
15	-1 (0 の 1 の補数)

以下はその説明です。

& は論理 AND

! は論理 OR

~ は論理 NOT

^ は XOR (排他的 OR)

詳細は [“演算子と式”](#) を参照してください。

すべての \$ZBOOLEAN 演算は、bit\_op 値 0、3、5、10、12、および 15 を含む arg1 と arg2 の両方を解析します。

\$ZBOOLEAN arg1 引数と arg2 引数は、以下のタイプのいずれかに解析されます。

- ・ 整数。正、または負の、18 桁までの 10 進数の整数です。数 0 から 9 を除く文字、またはオプションで 1 つ、または複数の先頭のプラス符号やマイナス符号を許可しません。先頭のゼロは無視されます。
- ・ 文字列。文字列の長さもコンテンツも任意ですが、文字列は引用符で囲まれる必要があります。文字列 “123” と整数 123 は同じものではありません。NULL 文字列も使用できますが、arg2 が NULL 文字列の場合は、bit\_op の値に関係なく、\$ZBOOLEAN は必ず arg1 の値を返します。
- ・ 符号付き文字列。先頭にプラス符号、またはマイナス符号が付く文字列は、文字列のコンテンツにかかわらず、整数として解析されます。符号付き文字列は、整数の長さ規約に従います。符号付き NULL 文字列は、ゼロと同じです。

arg1 と arg2 を両方とも整数に解析するか、または、両方とも文字列に解析することを強くお勧めします。通常、arg1 と arg2 は同じデータタイプです。\$ZBOOLEAN 演算で整数と文字列を結合すると、多くの場合で有用な結果を得ることができなくなります。

## 引数

### arg1

ビット単位論理式の中の 1 番目の引数。文字列では、戻り値の長さは常に、この引数の長さと同じです。

### arg2

ビット単位論理式の中の 2 番目の引数。

## bit\_op

実行されるビット単位論理演算。0 から 15 (包含的) の数値コードとして指定されます。このコードはビット・マスクとして処理されるので、値は 16=0、17=1、18=2 となります。

bit\_op 値 0 と 15 は定数値を返しますが、引数の評価も行います。arg1 が整数 (または符号付き文字列) の場合、bit\_op 値 0 は 0 を返し、bit\_op 値 15 は -1 を返します (0 の 1 の補数)。arg1 が文字列の場合、bit\_op 0 は arg1 の各文字に対する低値 (hex 00) を返し、bit\_op 15 は arg1 の各文字に対する高値 (hex FF) を返します。arg2 が NULL 文字列 ("" ) の場合、両方の演算は arg1 のリテラル値を返します。

bit\_op 値 3、5、10、および 12 は、引数のいずれかに対してのみ論理演算を実行しますが、評価は両方の引数に対して行います。

- ・ bit\_op=3 は常に、arg2 の値に関係なく arg1 の値を返します。
- ・ bit\_op=5 は、2 つの引数が同じデータ型のとき、arg2 の値を返します。しかし、引数のうち 1 つが文字列で、もう 1 つの引数が整数 (または符号付き文字列) の場合、結果は予測不可能です。arg2 が NULL 文字列の場合、\$ZBOOLEAN は arg1 のリテラル値を返します。
- ・ bit\_op=10 は、両方の引数が整数の場合に、arg2 の 1 の補数を返します。arg1 が文字列の場合、演算は arg1 の各文字に対する高順序文字を返します。arg2 が文字列で arg1 が整数の場合、ビット単位演算は arg2 文字列に対して実行されます。arg2 が NULL 文字列の場合、\$ZBOOLEAN は arg1 のリテラル値を返します。
- ・ bit\_op=12 は、その値が整数 (または符号付き文字列) の場合、NULL 文字列を除く任意の arg2 の値に対して、arg1 の 1 の補数を返します。arg1 が文字列の場合、演算は arg1 の各文字の 1 の補数を (hex 値として) 返します。arg2 が NULL 文字列の場合、\$ZBOOLEAN は arg1 のリテラル値を返します。

## 例

以下の 3 つの例は、すべて同じ AND 演算を表します。これらの例では、小文字の ASCII 値と下線文字 ( \_ ) の ASCII 値に対し AND 演算を実行すると、対応する ASCII 値の大文字が得られます。

### ObjectScript

```
WRITE $ZBOOLEAN("abcd","_",1)
```

これは、ABCD を表示します。

小文字の "a" = [01100001] (ASCII 10 進数 97)

下線文字 "\_" = [01011111] (ASCII 10 進数 95)

大文字の "A" = [01000001] (ASCII 10 進数 65)

以下の演算例は、上記の AND 演算と同じですが、ASCII 10 進数値を引数として使用します。関数 \$ASCII("a") は、最初の引数に対し 97 という 10 進数の値を返します。

### ObjectScript

```
WRITE $ZBOOLEAN($ASCII("a"),95,1)
```

これは、65 を表示します。

以下の例は、2 番目の引数として \$CHAR 値を使用して、同じ AND 演算を実行します。

### ObjectScript

```
WRITE $ZBOOLEAN("a",$CHAR(95),1)
```

これは、A を表示します。



以下は、論理 OR の例を示しています。

#### ObjectScript

```
WRITE $ZBOOLEAN(1,0,7)
```

これは、1 を表示します。

#### ObjectScript

```
WRITE $ZBOOLEAN(1,1,7)
```

これは、1 を表示します。

#### ObjectScript

```
WRITE $ZBOOLEAN(2,1,7)
```

これは、3 を表示します。

#### ObjectScript

```
WRITE $ZBOOLEAN(2,2,7)
```

これは、2 を表示します。

#### ObjectScript

```
WRITE $ZBOOLEAN(3,2,7)
```

これは、3 を表示します。

以下の論理和の例は、文字列比較と数値比較との間の相違を示します。

#### ObjectScript

```
WRITE $ZBOOLEAN(64,255,7)
```

2 つの値を数値として比較し、255 を表示します。

#### ObjectScript

```
WRITE $ZBOOLEAN("64","255",7)
```

2 つの値を文字列として比較し、65 を表示します。

#### ObjectScript

```
WRITE $ZBOOLEAN(+"64",+"255",7)
```

プラス符号により、強制的に 2 つの値が数値として比較され、255 を表示します。

以下は、排他的 OR の例を示しています。

#### ObjectScript

```
WRITE $ZBOOLEAN(1,0,6)
```

これは、1 を表示します。

### ObjectScript

```
WRITE $ZBOOLEAN(1,1,6)
```

これは、0 を表示します。

### ObjectScript

```
WRITE $ZBOOLEAN(2,1,6)
```

これは、3 を表示します。

### ObjectScript

```
WRITE $ZBOOLEAN(2,2,6)
```

これは、0 を表示します。

### ObjectScript

```
WRITE $ZBOOLEAN(3,2,6)
```

これは、1 を表示します。

### ObjectScript

```
WRITE $ZBOOLEAN(64,255,6)
```

これは、191 を表示します。

以下の例は、すべてのバイトが 1 に設定された 4 バイトのエンティティを示しています。

### ObjectScript

```
WRITE $ZBOOLEAN(5,1,15)
```

これは、-1 を表示します。

以下の例では、x を、すべてのビットが 1 に設定された 3 バイトの文字列に設定します。

### ObjectScript

```
SET x=$ZBOOLEAN("abc",0,15)
WRITE !,$LENGTH(x)
WRITE !,$ASCII(x,1)," ",,$ASCII(x,2)," ",,$ASCII(x,3)
```

最初の WRITE は 3 を、2 番目の WRITE は 255 255 255 を表示します。

## 整数の処理

\$ZBOOLEAN はビット単位演算を実行する前に、それぞれの数値をサイズによって 8 バイトまたは 4 バイトの符号付きバイナリ値に変換します。\$ZBOOLEAN は、常に数値を一連のバイトとして変換します。ブーリアン演算は、これらのバイト文字列を、文字列引数として使用します。結果のタイプは、arg1 のタイプと同じです。

arg1、arg2 の一方が数値であり、8 バイトの符号付き整数 (18 小数桁より大きい) で表示できない場合は、<FUNCTION> エラーが発生します。arg1、arg2 の両方が数字であり、一方を 8 バイトで表す必要がある場合、両方とも 8 バイトのバイナリ値として解釈されます。

以上の変換が完了した後、arg1 と arg2 に、任意のブーリアンの組み合わせが 1 ビットずつ割り当てられて結果が生成されます。(上記の数値データの変換後) 返される結果は、常に arg1 と同じ長さになります。arg2 が arg1 より短い場合は、arg2 は、連続した arg1 の部分文字列と、左から右へ繰り返して組み合わせられます。

使用しているマシンのネイティブのバイト・オーダーにかかわらず、\$ZBOOLEAN は数値を一連のバイトとして、常に下位バイトから順にリトルエンディアンで変換します。

## \$ZBOOLEAN 値の内部構造

以下の表では、\$ZBOOLEAN の内部法則を表しています。\$ZBOOLEAN を使用する際に、これらの法則を理解する必要はありません。参照としてご利用ください。

arg1 と arg2 内の 2 つのビットを比較する場合、起こりうる組み合わせは 4 種類あります。ブーリアン演算は、下記の表に示されるとおり、bit\_op がビット・マスクを持つ場合のみ、true (1) という結果を生じます。

arg1 のビット	arg2 のビット	bit_op 10 進数のビット・マスク	bit_op 2 進数のビット・マスク
0	0	8	1000
0	1	4	0100
1	0	2	0010
1	1	1	0001

## EQV および IMP 論理演算子

\$ZBOOLEAN は、EQV および IMP 論理演算子を直接サポートしません。これらの論理演算子は、以下のように定義されます。

- EQV は、2 つの式間の論理等値です。`$ZBOOLEAN(arg1,arg2,9)` で表されます。これは、論理的には  $\sim(\arg1 \wedge \arg2)$  で、 $((\sim \arg1) \& (\sim \arg2))!$  と論理的に同等です ( $\arg1 \& \arg2$ )。
- IMP は、2 つの式間の論理含意です。`$ZBOOLEAN(arg1,arg2,13)` で表されます。これは、 $(\sim \arg1) ! \arg2$  と論理的に同等です。

## 関連項目

- [ZZDUMP コマンド](#)
- [演算子](#)

## \$ZCONVERT (ObjectScript)

文字列変換関数

### 構文

```
$ZCONVERT(string,mode,trantable,handle)  
$ZCVT(string,mode,trantable,handle)
```

### 引数

引数	説明
string	変換する文字列。引用符付きの文字列として指定します。文字列は、値、変数、式として指定できます。
mode	変換モードを指定する文字コード。大文字小文字変換または入力/出力エンコーディングのいずれかのタイプになります。引用符付きの文字列として mode を指定します。
trantable	オプション - <a href="#">使用する変換テーブル</a> 。整数または引用符で囲まれた文字列によって指定します。
handle	オプション - 文字列値を持つ添字なしのローカル変数。\$ZCONVERT の複数の呼び出しのために使用されます。 <a href="#">handle</a> は、\$ZCONVERT の終了時に変換できなかった string の残り部分を格納しており、次の \$ZCONVERT 呼び出しにこの残り部分を提供します。

### 概要

\$ZCONVERT は、文字列の形式を変換します。変換の種類は、使用する引数によって決まります。

### \$ZCONVERT による文字列の変換

\$ZCONVERT(string, mode) は、mode で指定されたように変換された文字を含む、string を返します。変換には、以下の 2 つのタイプがあります。

- ・ 大小文字変換
- ・ エンコード変換

大小文字変換は、文字列内の各文字の大文字、小文字を変換します。文字列内のすべての文字を大文字から小文字に、小文字から大文字に、あるいはタイトル文字形式に変換できます。文字列内の単語または文の最初の文字を大文字形式に変更できます。文字列の中で大文字であるか小文字であるかが既に指定されている文字、および大文字と小文字の区別のない文字 (アルファベット以外の文字など) は、変換されません。リテラルの引用符 (") を出力するには、引用符を 2 つ (") 入力します。ASCII コードでないものやカスタム変換などの、大小文字変換のオプションの詳細は、"[各国言語サポートのシステム・クラスの使用法](#)" を参照してください。

エンコード変換は、使用しているシステムの内部的なエンコード形式と外部のエンコード形式が異なる場合に string を変換します。外部のエンコード形式から使用しているシステムのエンコード形式への string の変換、つまり入力変換を実行できます。また、使用しているシステムのエンコード形式から外部のエンコード形式への string の変換、つまり出力変換を実行することもできます。ASCII コードでないものやカスタム変換などの、入出力変換のオプションの詳細は、"[各国言語サポートのシステム・クラスの使用法](#)" を参照してください。

mode に使用する値は、以下のとおりです。

モード・コード	意味
U、u	<b>大文字変換</b> 。string 内のすべての文字を大文字に変換します。
L、l	<b>小文字変換</b> 。string 内のすべての文字を小文字に変換します。
T、t	<b>タイトル文字変換</b> 。string 内のすべての文字をタイトル文字に変換します。タイトル文字は、1 つの文字に小文字、大文字、およびタイトル文字の 3 つの形式があるアルファベット文字 (主に東ヨーロッパ言語) についてのみ有効です。その他すべての文字については、タイトル文字変換は大文字変換と同じです。
W または w	<b>単語変換</b> 。string 内の各単語の最初の文字を大文字に変換します。空白スペース、引用符 (")、アポストロフィ (')、開始の括弧 ( ( ) が前に付く文字は、いずれも単語の最初の文字と見なされます。単語変換は、その他のすべての文字を小文字に変換します。単語変換は、ロケール固有です。英語の場合、前述の構文規則が他の言語ロケールと異なる可能性があります。
S または s	<b>文変換</b> 。string 内の各文の最初の文字を大文字に変換します。string の最初の空白以外の文字、およびピリオド (.)、疑問符 (?)、感嘆符 (!) が前に付く文字は、いずれも文の最初の文字と見なされます (前にある句読点文字と文字の間の空白は無視されます)。この文字 (表意文字) が文字 (アルファベット文字) の場合、大文字に変換されます。文変換は、その他のすべての文字 (アルファベット文字) を小文字に変換します。文変換は、ロケール固有です。英語の場合、前述の構文規則が他の言語ロケールと異なる可能性があります。
I、i	指定された文字列で、 <b>入力エンコード変換</b> を実行します。引数を 2 つ持つ形式では、現在のプロセス入出力変換ハンドルを使用して変換を実行します。現在のプロセス入出力変換ハンドルが定義されていない場合、InterSystems IRIS は既定のプロセス入出力変換テーブル名に基づいて、変換を実行します。
O、o	指定された文字列で、 <b>出力エンコード変換</b> を実行します。引数を 2 つ持つ形式では、現在のプロセス入出力変換ハンドルを使用して変換を実行します。現在のプロセス入出力変換ハンドルが定義されていない場合、InterSystems IRIS は既定のプロセス入出力変換テーブル名に基づいて、変換を実行します。

mode が NULL 文字列である場合、あるいは有効な文字列以外の値である場合、<FUNCTION> エラーが発生します。

## 大文字/小文字の変換

文字列内の文字をすべて大文字またはすべて小文字に変換できます。変換は Unicode 文字および ASCII 文字で有効です。以下の例では、ギリシャ文字を小文字から大文字に変換します。

### ObjectScript

```
FOR i=945:1:969 {WRITE $ZCONVERT($CHAR(i),"U")}
```

ただし、小文字の形式しかない文字が少数ですが存在します。例えば、ドイツ語の eszett 文字 (\$CHAR(223)) は小文字としてのみ定義されています。これを大文字に変換しようとしても、結果は同じ小文字になります。

### ObjectScript

```
IF $ZCONVERT($CHAR(223),"U")=$ZCONVERT($CHAR(223),"L") {
    WRITE "uppercase and lowercase letter are the same" }
ELSE {WRITE "uppercase and lowercase are different" }
```

このため、英数字文字列をすべて大文字/小文字に変換する場合、小文字に変換することが常に推奨されます。

以下の例のように、\$TRANSLATE 関数を使用して、同様の文字列の変換を実行できます。

## ObjectScript

```
WRITE $TRANSLATE(text,"ABCDEFGHIJKLMNOPQRSTUVWXYZ","abcdefghijklmnopqrstuvwxyz")
```

## 単語および文の変換

“W” および “S” モードは、空白でない文字が単語の最初の文字どうか、または文の最初の文字かどうかを判別します。その文字 (表意文字) が文字 (アルファベット文字) の場合は、大文字に変換されます。その他のすべての文字は小文字に変換されます。大小文字変換は、任意のアルファベットの文字に対して行われます。以下の例では、ギリシャ文字の変換を示します (\$CHAR(945) は小文字のアルファ、\$CHAR(913) は大文字のアルファ)。

## ObjectScript

```
SET greek=$CHAR(945,946,947,913,914,915)
WRITE $ZCONVERT(greek,"W")
```

ただし、単語や文を構成要素を決める規則は、ロケールによって異なります。例えば、以下の例では、スペイン語の反転した感嘆符 \$CHAR(161) を使用します。既定 (英語) のロケールでは、この文字を文または単語の先頭として認識しません。この例では、spanish のすべての文字は小文字に変換されます。

## ObjectScript

```
SET spanish=$CHAR(161)_"ola MuNdO! "_$CHAR(161)_"olA!"
SET english="heLLo wOrLd! heLLo!"
WRITE !,$ZCONVERT(english,"S")
WRITE !,$ZCONVERT(spanish,"S")
```

## タイトル文字変換

タイトル文字 (“T”) モードは、文字列内のすべての文字をタイトル文字形式に変換します。タイトル文字は、単語または文字列の位置に基づいて選択的に文字を大文字に変換しません。タイトル文字は、タイトルの単語の先頭の文字を表す文字のタイプです。標準のラテン文字ではほとんどの場合、タイトル文字は大文字と同じです。

一部の言語 (例えばクロアチア語) では、特定の文字は 2 つの文字で表されます。例えば、クロアチア語のアルファベットで、“lj” は 1 文字です。この文字には、小文字 “lj”、大文字 “LJ”、およびタイトル文字 “Lj” の 3 つの形式があります。このタイプの文字変換には、\$ZCONVERT タイトル文字変換を使用します。

## 3 つの引数形式 : エンコード変換

\$ZCONVERT (string, mode, trantable) は、string に対して入力エンコード変換、または出力エンコード変換を実行します。引数を 3 つ持つ形式では、mode 値に “I” または “O” を使用できます。mode 値は必ず定義します。“I” 変換では、string は %4B (文字 “K”) などの 16 進数文字列になります。16 進数文字列は大文字小文字を区別しません。

ZZDUMP を使用して、文字列の 16 進エンコーディングを表示できます。[\\$CHAR](#) を使用して、文字 (または文字列) をその 10 進数 (10進法) エンコーディングで指定することができます。[\\$ZHEX](#) を使用して、16 進数を 10 進数に、または 10 進数を 16 進数に変換することができます。変換された値が出力不能文字である場合、InterSystems IRIS はそれを NULL 文字として表示します。変換された文字をターゲット・デバイスが表示できない場合、InterSystems IRIS はその出力不能文字を疑問符 (?) 文字に置き換えます。

trantable 値は、使用する変換テーブルまたは変換ハンドルを指定する数値文字あるいは文字列にできます。trantable の値は、以下のいずれかになります。

- ・ プロセス入出力変換オブジェクトを指定する整数値。使用可能な値は、0 ～ 3 です (0 は現在のプロセス入出力変換オブジェクトを表します)。
- ・ 入出力変換テーブルを指定する大文字の文字列値。リストおよび詳細は、“[変換テーブル](#)” を参照してください。
- ・ 入出力変換テーブルを指定する文字列値は、NLS ロケールによって定義されます。例えば、Latin2 または CP1252 などです。リストおよび詳細は、“[変換テーブル](#)” を参照してください。

- ・ ユーザ定義の入出力変換テーブルを指定する文字列値。指定されたテーブルは、ロケールで定義し、1 つあるいは 2 つの変換テーブルを示すことができます。指定されたテーブルを使用して、独自のシステムへの (あるいはデータベースからの) エンコードを定義することができます。
- ・ NULL 文字列 (""). この場合、既定のプロセス入出力変換テーブルを使用します (同様の機能については、%NLS ユーティリティの \$\$GetPDefIO~%NLS() 関数を参照してください)。

## 4 つの引数形式：入力/出力文字列

handle 引数は、\$ZCONVERT が実行の最初に読み取り、実行の最後に関数呼び出しの間に情報を保持するために使用されます。これは次のように使用できます。文字列を string の先頭に連結して、極端に長い文字列を変換します。

文字列を string の先頭に連結するには、\$ZCONVERT を呼び出す前に handle を設定します。

### ObjectScript

```
SET handle="the "
WRITE $ZCVT("quick brown fox","O","URL",handle),!
/* the%20quick%20brown%20fox */
WRITE $ZCVT("quick brown fox","O","URL",handle),!
/* quick%20brown%20fox */
```

\$ZCONVERT は、実行完了時に handle をリセットします。前述の例では、handle は空の文字列にリセットされています。

handle 引数は、入力変換のために使用できます。handle の指定が役に立つのは、文字の部分セット (ストリーム読み取りなど) を扱う際にマルチバイト文字シーケンスを処理する場合です。これらのケースでは、\$ZCONVERT は handle 引数を使用して、マルチバイト・シーケンスの先頭バイトである可能性のある部分文字シーケンスを保持します。\$ZCONVERT の終了時に、完全な変換単位を構成しない入力文字がバッファ内に残っている場合は、これらの残り文字は handle に格納されて返されます。次の \$ZCONVERT の開始時に、handle にデータが格納されている場合は、これらの残り文字は通常の入力データの前に付加されます。これは、以下の例に示すように UTF8 変換で使用する場合に特に有用です。

### ObjectScript

```
SET handle=""
WHILE 'stream.AtEnd() {
    WRITE $ZCONVERT(stream.Read(20000),"I","UTF8",handle)
}
```

極端に長い文字列を変換するには、\$ZCONVERT を複数回呼び出して、1 つ以上の文字列の変換を実行する必要があります。\$ZCONVERT には、string の変換されていない残りの部分を保持するために、オプションの handle 引数があります。handle 引数を指定すると、\$ZCONVERT が呼び出されるたびに更新されます。文字列変換が完了すると、\$ZCONVERT は handle に空の文字列を設定します。

### ObjectScript

```
SET handle=""
SET out = $ZCVT(hugestring,"O","HTML",handle)
IF handle '=' "" {
    SET out2 = $ZCVT(handle,"O","HTML",handle)
    WRITE "Converted string is: ",out,out2
} ELSE {
    WRITE "Converted string is: ",out }
```

## 例

以下の例は、“HELLO” を返します。

### ObjectScript

```
WRITE $ZCONVERT("Hello","U")
```



以下の例は、“hello” を返します。

#### ObjectScript

```
WRITE $ZCVT("Hello","L")
```

以下の例は、“HELLO” を返します。

#### ObjectScript

```
WRITE $ZCVT("Hello","T")
```

以下の例は、連結演算子 ( ) を使用して、アクセント記号付き文字を付加し、大小文字変換を実行します。

#### ObjectScript

```
WRITE "TOUCH"_$CHAR(201),!, $ZCVT("TOUCH"_$CHAR(201),"L")
```

これは、以下を返します。

TOUCHÉ

touché

以下の例は、文字列内の山括弧を出力するために HTML エスケープ文字に変換し、“&lt;TAG&gt;” を返します。

#### ObjectScript

```
WRITE $ZCVT("<TAG>","O","HTML")
```

出力デバイスにより、この山括弧の表示方法が異なります。このプログラムをここで実行したり、ターミナル・プロンプトで実行してみてください。

以下の例は、\$ZCONVERT が、変換された文字を表示できない場合にどのように？文字に置き換えるかを示しています。この例では、UTF8 および現在のプロセス入出力変換オブジェクト (trantable 0) のどちらの変換も \$CHAR(63) を表示していますが、これは実際には？文字です。UTF8 は、\$CHAR(127) を超える変換された文字は表示できません。変換テーブル 0 は、\$CHAR(255) を超える変換された文字は表示できません。

#### ObjectScript

```
FOR i=1:1:300 {IF $ZCONVERT($CHAR(i),"I","UTF8") '= "?"
    { CONTINUE }
    ELSE {WRITE "UTF8 ",i,"=", $ZCONVERT($CHAR(i),"I","UTF8")}
    IF $ZCONVERT($CHAR(i),"I",0)="?"
        {WRITE " trantable 0 ",i,"=", $ZCONVERT($CHAR(i),"I",0),!}
    ELSE {WRITE !}
}
```

## 関連項目

- ・ [変換テーブル](#)
- ・ [\\$ASCII 関数](#)
- ・ [\\$CHAR 関数](#)
- ・ [\\$ZSTRIP 関数](#)
- ・ [パターン・マッチング \(?\) 演算子](#)
- ・ [各国言語サポートのシステム・クラスの使用法](#)



# \$ZCOS (ObjectScript)

コサイン関数です。

## 構文

`$ZCOS (n)`

## 引数

引数	説明
n	Pi から 2 Pi までの範囲のラジアン単位の角度。提供されている他の数値は、この範囲内の値に変換されます。

## 説明

\$ZCOS は、n の三角関数のコサインを返します。結果は符号付きの 10 進数で、範囲は -1 から +1 です。\$ZCOS(0) は 1 を返します。\$ZCOS(\$ZPI) は -1 を返します。

## 引数

n

Pi から 2 Pi までの範囲のラジアン単位の角度。値、変数、式として指定することができます。\$ZPI 特殊変数を使用して Pi 値を指定できます。Pi より小さいか、2 Pi より大きい、正または負の値を指定できます。InterSystems IRIS はこれらの値を Pi の対応する倍数に解析します。例えば、3 Pi と Pi、負の Pi と Pi、ゼロ (0) と 2 Pi は同値です。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

## 例

以下は、数のコサインを計算する例です。

### ObjectScript

```
READ "Input a number: ",num
IF $ZABS(num)>(2*$ZPI) { WRITE !,"number is a larger than 2 pi" }
ELSE {
    WRITE !,"the cosine is: ",$ZCOS(num)
}
QUIT
```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。どちらの場合でも、0 のコサインは正確に 1 で、pi のコサインは正確に -1 です。

### ObjectScript

```
WRITE !,"the cosine is: ",$ZCOS(0.0)
WRITE !,"the cosine is: ",$ZCOS($DOUBLE(0.0))
WRITE !,"the cosine is: ",$ZCOS(1.0)
WRITE !,"the cosine is: ",$ZCOS($DOUBLE(1.0))
WRITE !,"the cosine is: ",$ZCOS($ZPI)
WRITE !,"the cosine is: ",$ZCOS($DOUBLE($ZPI))
```

## 関連項目

- ・ [\\$ZSIN](#) 関数

- ・ [\\$ZARCCOS](#) 関数
- ・ [\\$ZPI](#) 特殊変数

# \$ZCOT (ObjectScript)

コタンジェント関数です。

## 構文

`$ZCOT(n)`

## 引数

引数	説明
<i>n</i>	ラジアン単位の角度

## 概要

\$ZCOT は、*n* の三角関数のコタンジェントを返します。結果は、符号付きの 10 進数の数字です。

## 引数

*n*

ラジアン単位の角度です。0 以外の値として指定されます。値、変数、式として指定することができます。値 0 を指定すると <ILLEGAL VALUE> エラーが生成されます。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

## 例

以下は、数のコタンジェントを計算する例です。

### ObjectScript

```
READ "Input a number: ",num
IF num=0 { WRITE !,"zero is an illegal value" }
ELSE {
    WRITE !,"the cotangent is: ",$ZCOT(num)
}
QUIT
```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。

### ObjectScript

```
WRITE !,"the cotangent is: ",$ZCOT(1.0)
WRITE !,"the cotangent is: ",$ZCOT($DOUBLE(1.0))
WRITE !,"the cotangent is: ",$ZCOT(-1.0)
WRITE !,"the cotangent is: ",$ZCOT($DOUBLE(-1.0))
WRITE !,"the cotangent is: ",$ZCOT($ZPI/2)
WRITE !,"the cotangent is: ",$ZCOT($DOUBLE($ZPI)/2)
```

ここで、 $\pi/2$  のコタンジェントは小数値となり、0 にはなりません。

## 関連項目

- ・ [\\$ZTAN](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZCRC (ObjectScript)

チェックサム関数

### 構文

```
$ZCRC(string,mode,expression)
```

### 引数

引数	説明
string	チェックサム演算が実行される文字列。
mode	使用するチェックサム・モードを指定する整数コード。
expression	オプション – “seed” の初期値。整数として指定します。省略した場合、既定はゼロ (0) です。

### 概要

\$ZCRCstring に対して冗長巡回検査を実行し、整数のチェックサム値を返します。\$ZCRC によって返される値は、使用する引数により異なります。

- ・ \$ZCRC(string,mode) は、文字列のチェックサムを計算して返します。mode の値により、\$ZCRC が計算するチェックサムのタイプが決まります。
- ・ \$ZCRC(string,mode,expression) は、mode により指定されたモードを使用して、string のチェックサムを計算して返します。expression は、複数の文字列をチェックする際 “seed” の初期値を提供します。これにより、複数の文字列で連続的に \$ZCRC 計算を実行し、それらの文字列を連結して \$ZCRC を実行したかのように、同じチェックサム値を得ることができます。

### 引数

#### string

バイト文字列。値、変数、式として指定できます。バイトの文字列のみを使用し、それ以外を使用した場合は、<FUNCTION> エラーが発生します。

#### mode

使用するチェックサム・アルゴリズム。すべてのチェックサム・モードは、8 ビット (ASCII) または 16 ビット Unicode (ワイド) 文字を使用できます。規定の mode 値は以下のとおりです。

モード	計算
0	8 ビットのバイト加算値。文字列内の文字の ASCII 値の合計です。したがって、\$ZCRC(2,0)=50、\$ZCRC(22,0)=100、\$ZCRC(23,0)=101、\$ZCRC(32,0)=101 となります。
1	XOR 8 ビットの 加算値
2	16 ビットのデータ・ツリー CRC-CCITT
3	16 ビットのデータ・ツリー CRC-16
4	XMODEM プロトコルの 16 ビット CRC
5	正しい 16 ビット CRC-CCITT
6	正しい 16 ビット CRC-16
7	正しい 32 ビット CRC-32。これは、OS X および Java ユーティリティ・パッケージの CRC32 クラスの cksum ユーティリティ・アルゴリズム 3 に相当します。
8	32 ビットの Murmur3 ハッシュ (x64 バリエント)
9	128 ビットの Murmur3 ハッシュ (x64 バリエント)

## expression

“seed” の初期値である引数。\$ZCRC は string に対して生成したチェックサムに expression を追加します。これにより、複数の文字列で連続的に \$ZCRC 計算を実行し、それらの文字列を連結して \$ZCRC を実行したかのように、保存済みのチェックサム値を得ることができます。\$ZCRC 式のチェイニングは、きわめて長い文字列を分割して <MAXSTRING> エラーを防止するうえで効果的です。Murmur3 ハッシュ (モード 8 および 9) では expression 引数がサポートされていません。

## 例

以下の例では、文字 A、B、および C を含む文字列で mode=0 を使用し、それぞれチェックサム 198 を返します。

### ObjectScript

```
write $ZCRC("ABC",0),!
write $ZCRC("CAB",0),!
write $ZCRC("BCA",0),!
```

チェックサムは、以下のようにして得られます。

### ObjectScript

```
write $ASCII("A")+ $ASCII("B")+ $ASCII("C") /* 65+66+67 = 198 */
```

以下の例は、文字列 “ABC” の各 mode により返される値を示します。

### ObjectScript

```
for i=0:1:9 { write "mode ",i," = ", $ZCRC("ABC",i),! }
```

以下の例は、前の \$ZCRC 値の出力を次の計算のシードとして使用できる様子を示します。1 から 7 までの各モードでは、“ABC” の CRC は “A”、“B”、“C” に対して連続して計算した CRC に等しく、それぞれの計算にはその前の文字の CRC がシードされています。モード 8 および 9 (Murmur3 ハッシュ) では、Murmur3 が連鎖式をサポートしていないため、CRC 計算は等しくなりません。

## ObjectScript

```
for mode = 1:1:9 {  
    set crc1 = $zcrc("ABC",mode)  
  
    set crc2 = $zcrc("A",mode)  
    set crc2 = $zcrc("B",mode,crc2)  
    set crc2 = $zcrc("C",mode,crc2)  
  
    write "mode ", mode," ", "crc1 = crc2: ", crc1 = crc2, !  
}
```

## 関連項目

- ・ [\\$ZCYC](#) 関数

# \$ZCSC (ObjectScript)

コセカント関数です。

## 構文

`$ZCSC(n)`

## 引数

引数	説明
n	ラジアン単位の角度

## 概要

\$ZCSC は、n の三角関数のコセカントを返します。結果は、符号付きの 10 進数の数字です。

## 引数

n

ラジアン単位の角度です。0 以外の数値として指定されます。値、変数、式として指定することができます。0 を指定すると <ILLEGAL VALUE> エラーが生成され、\$DOUBLE(0) を指定すると <DIVIDE> エラーが生成されます。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

## 例

以下は、数のコセカントを計算する例です。

### ObjectScript

```

READ "Input a number: ",num
IF num=0 { WRITE !,"ILLEGAL VALUE: zero not permitted" }
ELSE {
    WRITE !,"the cosecant is: ",$ZCSC(num)
}
QUIT

```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。どちらの場合も、pi/2 のコセカントは正確に 1 になります。

### ObjectScript

```

WRITE !,"the cosecant is: ",$ZCSC($ZPI)
WRITE !,"the cosecant is: ",$ZCSC($DOUBLE($ZPI))
WRITE !,"the cosecant is: ",$ZCSC($ZPI/2)
WRITE !,"the cosecant is: ",$ZCSC($DOUBLE($ZPI)/2)
WRITE !,"the cosecant is: ",$ZCSC($DOUBLE($ZPI/2))

```

## 関連項目

- ・ [\\$ZSEC](#) 関数
- ・ [\\$ZCOT](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZCYC (ObjectScript)

データ整合性の冗長巡回検査

### 構文

```
$ZCYC(string)  
$ZC(string)
```

### 引数

引数	説明
string	文字列

### 概要

\$ZCYC(string) は、文字列の巡回冗長検査値を計算して返します。これにより 2 点間通信プログラムでのデータ整合性を確認することができます。

送信プログラムは、データの塊を送信する際に、一緒に \$ZCYC を使用して計算された照合検査値も送信します。受信プログラムは、\$ZCYC を使用して受信したデータの検査値を計算し、検証します。2 つの検査値が一致すれば、受信したデータは送信したデータと同一であると判断できます。

\$ZCYC は、文字列の 8 ビット文字すべての 2 進数表記の排他的論理和 (XOR) を実行して、検査値を計算します。

8 ビット文字列の \$ZCYC 値は、\$ZCRC のモード 1 の値と一致します。

### 引数

#### string

文字列。値、変数、式として指定できます。文字列値は引用符で囲まれます。

### 例

以下の例では、最初の \$ZCYC は 65、2 番目は 3、3 番目は 64 を返します。

#### ObjectScript

```
SET x= $ZCYC("A")  
; 1000001 (only one character; no XOR )  
SET y= $ZCYC("AB")  
; 1000001 XOR 1000010 -> 0000011  
SET z= $ZCYC("ABC")  
; 1000001 XOR 1000010 -> 0000011 | 1000011 -> 1000000  
WRITE !,"x=",x," y=",y," z=",z
```

### 関連項目

- ・ [\\$ZCRC 関数](#)



## \$ZDASCII (ObjectScript)

8 バイト文字列を \$DOUBLE 浮動小数点数に変換します。

### 構文

```
$ZDASCII(string,position)
$ZDA(string,position)
```

### 引数

引数	説明
string	文字列または数値。値、変数、式として指定することができます。長さは 8 バイト以上でなければなりません。数値は、\$ZDASCII に指定される前に、先頭のプラス記号、および先頭と末尾のゼロを削除することによって、キャノニック形式に変換されます。結果として得られるキャノニック形式の数の長さは 8 バイト以上でなければなりません。
position	オプション - 正の、ゼロ以外の整数として表される、文字列の開始位置。既定は 1 です。位置は、8 バイト文字列ではなくシングル・バイト単位でカウントされます。指定された position (その位置の文字を含む) から少なくとも 8 バイトの string が存在する必要があります。position の数値は、小数桁数を切り捨て、先行するゼロやプラス記号などを削除することによって、整数として解析されます。

### 説明

\$ZDASCII が返す値は、使用する引数によって異なります。

- ・ \$ZDASCII(string) は、string の最初の文字位置から開始して、8 バイト文字列を \$DOUBLE (IEEE 浮動小数点) の数値に変換して返します。
- ・ \$ZDASCII(string,position) は、position で指定したバイト位置から開始して、8 バイト文字列を \$DOUBLE (IEEE 浮動小数点) の数値に変換して返します。

\$ZDASCII は、正の数または負の数のいずれかを返すことができます。

\$ZDASCII は、string の長さが 8 バイト未満の場合、指定された position の残りの string が 8 バイト未満の場合、または position が無効な値の場合に、<FUNCTION> エラーを発行します。

### 例

以下は、数値 12345678 を数値に変換する例です。

```
WRITE $ZDASCII(12345678)
```

これは .0000000000000000000000000000000000000000000068213200517013251261 を返します。

次の例も同じ値を返します。

```
WRITE !,$ZDASCII("12345678",1)
WRITE !,$ZDASCII(++001234567899,1)
WRITE !,$ZDASCII("++001234567899",5)
```

以下は、負の数値 -1234567 を数値に変換する例です。マイナス記号は 1 バイトとしてカウントされることに注意してください。

```
WRITE $ZDASCII(-1234567)
```



## \$ZDATE (ObjectScript)

日付を検証し、これを内部形式から指定の表示形式に変換します。

### 構文

```
$ZDATE(hdate,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erroppt,localeopt)
$ZD(hdate,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erroppt,localeopt)
```

### 引数

引数	説明
hdate	内部の日付形式値を指定する整数。この整数は、1840 年 12 月 31 日からの経過日数を表します。hdate に \$HOROLOGY が指定されている場合、\$HOROLOGY の日付部分のみが使用されます。後述の “hdate” を参照してください。
dformat	オプション — 返される日付の形式を表す整数コード。後述の “dformat” を参照してください。
monthlist	オプション — 連の月名を表す文字列または変数の名前。この文字列は、区切り文字から始まり、12 個のエントリは、この区切り文字で分けられる必要があります。後述の “monthlist” を参照してください。
yearopt	オプション — 年を 2 桁または 4 桁のどちらで表記するかを指定する整数コード。後述の “yearopt” を参照してください。
startwin	オプション — 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値。後述の “startwin” を参照してください。
endwin	オプション — 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最後を指定する数値。後述の “endwin” を参照してください。
mindate	オプション — 有効な日付範囲の下限。\$HOROLOGY 整数日付カウントの形式で指定し、0 は 1840 年 12 月 31 日を表します。正または負の整数として指定できます。後述の “mindate” を参照してください。
maxdate	オプション — 有効な日付範囲の上限。\$HOROLOGY 整数日付カウントの形式で指定します。後述の “maxdate” を参照してください。
erroppt	オプション — hdateが無効のときに返す式。この引数に値を指定すると、hdate 値が無効または範囲外の場合に生じるエラー・コードが抑制されます。エラー・メッセージを発行する代わりに、\$ZDATE は erroppt を返します。後述の “erroppt” を参照してください。
localeopt	<p>オプション — dformat、monthlist、yearopt、mindate、maxdate の既定値、およびその他の日付特性（日付区切り文字など）に対してどのロケールを使用するかを指定するブーリアン・フラグ。</p> <p>localeopt=0: 現在のロケール・プロパティ設定によって、これらの引数の既定値が決定されます。</p> <p>localeopt=1: ODBC 標準ロケールによって、これらの引数の既定値が決定されます。</p> <p>localeopt 未指定: dformat 値によって、これらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。日本およびイスラム暦の日付の dformat 値については、それぞれの既定値が使用されます。その他すべての dformat 値については、現在のロケール・プロパティ設定が既定値として使用されます。後述の “localeopt” を参照してください。</p>

指定の引数値間で省略された引数は、プレースホルダのコンマで示されます。末尾のプレースホルダのコンマは必要ありませんが、あってもかまいません。省略された引数を示すコンマの間に空白があってもかまいません。

## 概要

\$ZDATE 関数は、内部の格納形式 (\$HOROLOG) で指定した日付を、別の日付表示形式に変換します。\$ZDATE によって返される値は、使用する引数により異なります。

## 単純な \$ZDATE 形式

\$ZDATE(hdate) は \$ZDATE の最も基本的な形式で、指定した hdate に対応する表示形式の日付を返します。hdate は、1840 年 12 月 31 日から経過した日数の整数値で、範囲は、0 ~ 2980013 (12/31/1840 ~ 12/31/9999) です。

既定では、\$ZDATE(hdate) は 1900 年から 1999 年までの年を 2 桁で表します。1900 年より前と 1999 年より後の年は、4 桁で表します。次に例を示します。

### ObjectScript

```
WRITE $ZDATE(21400),! ; returns 08/04/1899
WRITE $ZDATE(50000),! ; returns 11/23/77
WRITE $ZDATE(60000),! ; returns 04/10/2005
WRITE $ZDATE(0),! ; returns 12/31/1840
```

\$HOROLOG の日付を \$ZDATE に提供する場合は、日付部分のみが使用されます。\$HOROLOG では、コンマで区切られた 2 つの整数として、日付と時刻が表されます。コンマ (数値以外の文字) が検出されると、\$ZDATE はその文字列の残りを無視します。以下の例で、\$ZDATE は、04/10/2005 と \$HOROLOG 形式の値による現在の日付を返します。

### ObjectScript

```
WRITE !,$ZDATE("60000,12345")
WRITE !,$ZDATE($HOROLOG)
```

## 日付の既定値のカスタマイズ

InterSystems IRIS の起動時、既定の日付形式は、dformat=1 に初期化されています。これは日付をスラッシュで区切るアメリカ日付形式 (MM/DD/[YY]YY) です。この形式およびその他の既定の形式を現在のロケール用に設定するには、SET ^SYS("NLS","Config","LocaleFormat")=1 のグローバル変数を設定します。これは、すべてのプロセスのすべての既定の形式を現在のロケール値に設定します。これらの既定は、このグローバルが変更されるまで維持されます。

**注釈** ここでは、localeopt が未定義または 0 に設定されているときに適用されるユーザ・ロケール定義について説明します。localeopt=1 の場合、\$ZDATE は、定義済みの ODBC ロケールを使用します。

NLS (各国言語サポート) を使用して現在のプロセスの既定の形式を上書きすることもできます。すべての既定の形式を指定したロケールの値に変更することも、個々の形式の値を変更することもできます。

- 既定の形式をすべて (既定の日付形式を含む) 指定のロケールのプロパティに設定するには、SET fmt=##class(%SYS.NLS.Format).%New("lname") メソッド呼び出しを実行します。lname は目的のロケールの NLS 名です (例えば、deuw=German、espw=Spanish、fraw=French、ptbw=Brazilian Portuguese、rusw=Russian、jpnw=Japanese)。全ロケールのリストは、管理ポータルで、**[システム管理]**、**[構成]**、**[国際言語設定]**、**[ロケール定義]** を参照してください。これらの既定を現在のロケールのプロパティに設定するには、"current" の lname または空の文字列 ("") を指定します。
- 既定の日付形式を指定した dformat 形式に設定するには、SetFormatItem() メソッド、つまり SET rtn=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",n) を呼び出します。n は既定にする dformat 値の数です。

以下の例では、すべての既定の形式を Russian ロケールに設定し、既定の形式 (Russian) で \$ZDATE の日付を返し、既定の形式を現在のロケールの既定にリセットします。Russian ロケールでは、日付部分の区切り文字としてスラッシュではなく、ピリオドが使用されることに注意してください。

## ObjectScript

```
WRITE !,$ZDATE($HOROLOG)
SET fmt=##class(%SYS.NLS.Format).%New("rusw")
WRITE !,$ZDATE($HOROLOG)
SET fmt=##class(%SYS.NLS.Format).%New("current")
WRITE !,$ZDATE($HOROLOG)
```

以下の例では、個々の既定の形式を設定しています。最初の \$ZDATE は既定の形式で日付を返します。最初の SetFormatItem() メソッドは、2 つ目の \$ZDATE で示されるように、既定値を dformat=4、またはヨーロッパ日付形式 (DD/MM/[YY]YY) に変更します。2 つ目の SetFormatItem() メソッドは、(dformat が -1、1、4 および 15 の場合に影響する) 日付区切り文字に対する既定値を変更します。この例では、3 つ目の \$ZDATE で示されるように、日付区切り文字は、ドット (".") に設定されています。最後に、このプログラムは元の日付形式の値をリストアします。

## ObjectScript

```
InitialVals
SET fmt=##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
SET sep=##class(%SYS.NLS.Format).GetFormatItem("DateSeparator")
WRITE !,$ZDATE($HOROLOG)
ChangeVals
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",4)
WRITE !,$ZDATE($HOROLOG)
SET y=##class(%SYS.NLS.Format).SetFormatItem("DateSeparator",".")
WRITE !,$ZDATE($HOROLOG)
RestoreVals
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",fmt)
SET y=##class(%SYS.NLS.Format).SetFormatItem("DateSeparator",sep)
WRITE !,$ZDATE($HOROLOG)
```

サポートされるロケールの既定の日付形式の詳細は、“[日付](#)”を参照してください。

## 引数

### hdate

1840 年 12 月 31 日から経過した日数を示す内部の日付形式値。既定では、この値は 0 から 2980013 の範囲内の整数である必要があります。hdate は、数値、文字列リテラル、または式として指定できます。InterSystems IRIS は、hdate を[キャノニック形式](#)に変換します。数値文字列 (\$HOROLOG 値など) は、最初の非数値文字で切り捨てられます。非数値文字列は、整数 0 として評価されます。整数に解決されない浮動小数点数を指定すると、<ILLEGAL VALUE> エラーが発生します。

既定では、最も早い有効な hdate は 0 です (1840 年 12 月 31 日)。**DateMinimum** プロパティの既定値は 0 であるため、既定では、日付は正の整数に制限されています。現在のロケールの **DateMinimum** プロパティが、過去の日付を表す負の整数以下の負の整数に設定されていれば、過去の日付を負の整数を使用して指定できます。**DateMinimum** の有効な最小値は -672045 であり、これは 0001 年 1 月 1 日に相当します。InterSystems IRIS では、ISO 8601 標準に準拠してグレゴリオ暦が“西暦 1 年”まで遡って適用された先発グレゴリオ暦が使用されています。その理由の 1 つは、グレゴリオ暦が採用された時期は国ごとに異なるからです。例えば、欧州大陸諸国の多くでは 1582 年に採用された一方で、英国と米国では 1752 年に採用されました。このため、ご使用の地域でグレゴリオ暦が採用された時期より前の InterSystems IRIS の日付は、その当時にその地域で採用されていた暦に基づいて記録された歴史上の日付に対応していない可能性があります。1840 年より前の日付の詳細は、“[mindate](#)” 引数を参照してください。

hdate 値が無効または範囲外の場合に生じるエラーは、[errorpt](#) 引数で示されます。

### dformat

返される日付の形式。有効な値は以下のとおりです。

値	意味
1	MM/DD/[YY]YY (07/01/97 または 03/27/2002) – <a href="#">アメリカの数値形式</a> 。日付区切り文字 (/ または .) は、現在のロケール設定から取得されます。
2	DD Mmm [YY]YY (01 Jul 97 あるいは 27 Mar 2002)
3	YYYY-MM-DD (1997-07-01 あるいは 2002-03-27) – ODBC 形式。既定では、この形式はユーザの現在のロケール設定 (localeopt=1) に依存しません。したがって、日付を ODBC 標準交換形式で指定します。日付に対するユーザの現在のロケール設定をこの形式で使用するには、localeopt=0 に設定します。
4	DD/MM/[YY]YY (01/07/97 または 27/03/2002) – <a href="#">ヨーロッパの数値形式</a> 。日付区切り文字 (/ または .) は、現在のロケール設定から取得されます。
5	Mmm [D]D, YYYY (Jul 1, 1997 あるいは Mar 27, 2002)
6	Mmm [D]D YYYY (Jul 1 1997 あるいは Mar 27 2002)
7	Mmm DD [YY]YY (Jul 01 1997 あるいは Mar 27 2002)
8	YYYYMMDD (19970701 あるいは 20020327) – 数値形式
9	Mmmmm [D]D, YYYY (July 1, 1997 あるいは March 27, 2002)
10	W (2) – 曜日の番号。0 (日) から 6 (土) までの数値で指定します。\$SYSTEM.SQL.DAYOFWEEK() メソッドと比較してください。
11	Www (Tue) – 曜日の略名
12	Wwwwww (Tuesday) – 正式な曜日の名前
13	[D]D/[M]M/YYYY (1/7/2549 または 27/11/2549) – タイ語の日付形式。日と月は、ヨーロッパ形式と同じですが、先頭に 0 は付きません。年は仏教紀元 (BE) で、グレゴリオ暦の年に 543 年追加して計算されます。
14	nnn (354) – 年の日付
15	DD/MM/[YY]YY (01/07/97 や 27/03/2002) – ヨーロッパ形式 (dformat=4 と同じ)。日付区切り文字 (/ または .) は、現在のロケール設定から取得されます。
16	YYYYc[M]Mc[D]Dc – 日本の日付形式。年、月、および日の数は、他の日付形式と同じです。先頭のゼロは省略されます。“年”、“月”、および“日”の日本語の文字 (ここでは c として表示) は、年、月、および日の数の後に挿入されます。これらの文字は、年=\$CHAR(24180)、月=\$CHAR(26376)、および日=\$CHAR(26085) です。
17	YYYYc[M]Mc[D]Dc – 日本の日付形式。dformat 16 と同じです。ただし、“年”および“月”の日本語の文字の後に空白が挿入されます。
18	[D]D Mmmmm YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月名を示す)。日の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 Muharram 0001。
19	[D]D [M]M YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月番号を示す)。日と月の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 1 0001。
20	[D]D Mmmmm YYYY – Observed Hijri (イスラム) 日付形式 (完全な月名を示す)。既定で Tabular Hijri (dformat 18) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。



値	意味
21	[D]D [M]M YYYY – Observed Hijri (イスラム) 日付形式 (完全な月番号を示す)。既定で Tabular Hijri (dformat 19) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。
-1	有効な dformat 値は、 <a href="#">ユーザのロケール</a> (localeopt=0 または未定義の場合)、あるいは ODBC ロケール (既定は dformat=3) のいずれかから取得します。dformat をユーザのロケールから取得した場合、これは fmt.DateFormat の値になります。fmt は、現在のプロセスに関連付けられた ##class(%SYS.NLS.Format) のインスタンスです。dformat を指定しなければ、これが既定の振る舞いになります。詳細は、 <a href="#">“日付の既定値のカスタマイズ”</a> を参照してください。

各項目の内容は次のとおりです。

構文	意味
YYYY	YYYY は 4 桁の年です。[YY]YY は、hdate がアクティブ・ウィンドウ内で 2 桁の場合は 2 桁の年、それ以外は 4 桁の年です。
MM	2 桁の月：01 から 12。[M]M は、1 月から 9 月では先頭のゼロが省略されることを示します。
DD	2 桁の日：01 から 31。[D]D は、1 日から 9 日では先頭のゼロが省略されることを示します。
Mmm	現在のロケールの MonthAbbr プロパティから取得した月の省略形。別の月の省略形 (あるいは長さに制限のない名前) は、\$ZDATE に monthlist 引数として指定したオプションのリストから取得することができます。以下は MonthAbbr の既定値です。 “Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”
Mmmmm	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。以下はその既定値です。 “January February March ...November December”
W	曜日を示す 0 から 6 の数字。 (例) Sunday は 0、Monday は 1、Tuesday は 2。
Www	現在のロケールの WeekdayAbbr プロパティによって指定される曜日の省略形。以下はその既定値です。 “Sun Mon Tue Wed Thu Fri Sat”
Wwwwww	現在のロケールの WeekdayName プロパティによって指定される曜日の正式な名前。以下はその既定値です。 “Sunday Monday Tuesday ...Friday Saturday”
nnn	指定された年の日付。常に 3 桁で示され、必要に応じて先頭にゼロが付きます。値は、001 ~ 365 (うるう年の場合は 366) です。

### dformat の既定値

dformat を省略するか、-1 に設定した場合、dformat の既定値は localeopt 引数と NLS の **DateFormat** プロパティに依存します。

- ・ localeopt=1 の場合、dformat の既定値は ODBC 形式です。monthlist、yearopt、mindate、および maxdate 引数の既定値も、ODBC 形式に設定されます。これは、dformat=3 を設定した場合と同じです。
- ・ localeopt=0 または未指定の場合、dformat の既定値は NLS の **DateFormat** プロパティから取得されます。**DateFormat=3** の場合、dformat の既定値は ODBC 形式です。ただし、DateFormat=3 は monthlist、yearopt、mindate、および maxdate 引数の既定値に影響しません。これらは、NLS の現在のロケール定義で指定されます。

ユーザのロケールの既定の日付形式を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
```

ヨーロッパ日付形式 (dformat=4、DD/MM/YYYY の順) は、ヨーロッパの多くの言語 (すべてではない) で既定です。これには、“/” を **DateSeparator** 文字として使用するイギリス英語、フランス語、ドイツ語、イタリア語、スペイン語、ポルトガル語、および “.” を **DateSeparator** 文字として使用するチェコ語 (csyw)、ロシア語 (rusw)、スロバキア語 (skyw)、スロベニア語 (svnw)、ウクライナ語 (ukrw) が含まれます。サポートされるロケールの既定の日付形式の詳細は、“[日付](#)” を参照してください。

### dformat の設定

dformat が 3 (ODBC 日付形式) の場合は、monthlist、yearopt、mindate、および maxdate 引数の既定値にも ODBC 形式の既定値が使用されます。現在のロケールの既定値は無視されます。

dformat が -1、1、4、13、または 15 (数値の日付形式) の場合は、現在のロケールの **DateSeparator** プロパティの値が \$ZDATE の月、日、年の区切り文字として使用されます。dformat が 3 の場合は、ODBC の日付区切り文字 (“-”) が使用されます。dformat がそれ以外の値の場合は、空白が日付区切り文字として使用されます。英語での **DateSeparator** の既定値は “/” で、この区切り文字はすべてのドキュメントで使用されています。

dformat が 11 または 12 (曜日名) で、localeopt=0 または未指定の場合、曜日名の値は現在のロケールのプロパティから得られます。localeopt=1 の場合、曜日名は ODBC ロケールから得られます。ユーザのロケールの既定の曜日名および曜日名の省略形を決定するには、以下の NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayName"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayAbbr"),!
```

dformat が 16 または 17 (日本の日付形式) の場合、返される日付形式はロケールの設定に依存しません。日本の日付形式は、任意の InterSystems IRIS インスタンスから取得できます。

dformat が 18、19、20、または 21 (イスラム暦の日付形式) で、localeopt が未指定の場合、引数の既定値は現在のロケールの既定値ではなく、イスラム暦の既定値になります。monthlist 引数の既定値は、ラテン文字で書き直されたアラビア語の月名になります。tformat、yearopt、mindate、および maxdate 引数の既定値は、ODBC の既定値になります。日付区切り文字の既定値は、ODBC の既定値や現在のロケールの **DateSeparator** プロパティ値ではなく、イスラム暦の既定値 (空白) になります。localeopt=0 の場合は、現在のロケール・プロパティの既定値がこれらの引数に使用されます。localeopt=1 の場合は、ODBC の既定値がこれらの引数に使用されます。

### monthlist

区切り文字で区切られた月名または月の省略名の文字列に解決される式です。monthlist にある名前は、現在のロケールの **MonthAbbr** プロパティに指定されている既定の月の省略形の値、または **MonthName** に指定されている月名の値を置き換えます。

monthlist は、dformat が 2、5、6、7、9、18、または 20 の場合にのみ有効です。dformat がそれ以外の値の場合、\$ZDATE は monthlist を無視します。

monthlist 文字列の形式は、以下のとおりです。

- 文字列の最初の文字は、区切り文字 (通常はスペース) です。monthlist 内の最初の月名の前に、各月名の間に、同じ区切り文字を置く必要があります。1 文字からなる任意の区切り文字を指定できます。この区切り文字は、返される日付値の月、日、年の間に表示されます。このため、スペースが通常推奨される区切り文字となります。
- 月名の文字列は、1 月から 12 月に対応する 12 個の区切り値を含む必要があります。12 より多い、または 12 より少ない月名を指定することは可能ですが、hdate の月に該当する月名がない場合、<ILLEGAL VALUE> エラーが発生します。



monthlist を省略するか、monthlist の値として -1 を指定した場合は、現在のロケールの **MonthAbbr** または **MonthName** プロパティで定義されている月名のリストが \$ZDATE で使用されます (localeopt=1、または monthlist の既定値が ODBC の月リスト (英語) である場合を除く)。localeopt が未指定で、dformat が 18 または 20 (イスラム暦の日付形式) の場合、monthlist の既定値はイスラム暦の月リスト (ラテン文字で表されたアラビア語の名前) になり、**MonthAbbr** および **MonthName** プロパティ値は無視されます。

ユーザのロケールの既定の月名および月名の省略形を指定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```

以下の例では、既定のロケールの月名をリストし、このプロセスのロケールを Russian ロケールに変更し、さらにロシア語の月名をリストして、ロシア語の月名で現在の日付を表示します。さらに、既定のロケールを現在のロケールに戻し、再度現在の日付を表示します。今回は既定の月名で表示されます。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
SET fmt=##class(%SYS.NLS.Format).%New("rusw")
WRITE fmt.MonthName,!
WRITE $ZDATE($HOROLOG,9),!
SET fmt=##class(%SYS.NLS.Format).%New()
WRITE $ZDATE($HOROLOG,9)
```

## yearopt

dformat 値が 1、2、4、7、または 15 の場合に、年を 2 桁の値で表示する時間枠を指定する整数コードです。その他すべての dformat 値では、yearopt は無視されます。有効な yearopt 値は、以下のとおりです。

値	意味
-1	現在のロケールの YearOption プロパティから有効な yearopt 値を取得します。既定値は 0 です。yearopt を指定しない場合は、これが既定の動作になります。
0	(%DATE ユーティリティから構築される) プロセス固有のスライディング・ウィンドウ機能が有効になっていない限り、20 世紀 (1900 年から 1999 年まで) の日付は 2 桁で表し、それ以外の日付は 4 桁で表します。スライディング・ウィンドウが有効の場合、指定範囲内の日付のみ 2 桁の年で表し、それ以外の日付は 4 桁で表します。
1	20 世紀の日付は 2 桁の年、それ以外は 4 桁で表します。
2	すべての日付を 2 桁の年で表します。
3	startwin と (オプションで) endwin に定義した一時的なスライディング・ウィンドウ範囲内に収まる日付を、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt=3 の場合、startwin と endwin は、\$HOROLOG 形式の絶対日付です。
4	すべての日付を 4 桁の年で表します。ODBC の年オプションです。
5	startwin と (オプションで) endwin に定義した一時的なスライディング・ウィンドウ範囲内に収まる日付を、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt が 5 の場合、startwin と endwin は相対年になります。
6	2 桁の年で現在の世紀にあるすべての日付を表します。それ以外は 4 桁の年で表します。

ユーザのロケールの既定の年のオプションを決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("YearOption")
```

yearopt を省略するか、yearopt の値として -1 を指定した場合は、現在のロケールの **YearOption** プロパティが \$ZDATE で使用されます (localeopt=1、または yearopt の既定値が ODBC の年オプションである場合を除く)。localeopt=0 または未指定で、dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、yearopt の既定値は ODBC の年オプション (4 桁の年) になります。イスラム暦の日付では、**YearOption** プロパティの値は無視されます。

## startwin

日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値です。詳細は、引数のセクションを参照してください。yearopt が 3 または 5 のとき、startwin を指定する必要があります。startwin は、その他の yearopt 値では無効になります。

yearopt=3 のとき、startwin はスライディング・ウィンドウの最初の日付を示す \$HOROLOGY 日付形式の絶対日付です。

yearopt = 5 のとき、startwin は、現在の年より前の年数として表される、スライディング・ウィンドウの最初の年を示す数値です。スライディング・ウィンドウは常に、startwin で指定した年の 1 月 1 日から開始します。

## endwin

日付を 2 桁の年で表すスライディング・ウィンドウの最後を指定する数値です。yearopt が 3 あるいは 5 のとき、endwin をオプションで指定する必要があります。endwin は、その他の yearopt 値では無効になります。

yearopt=3 のとき、endwin はスライディング・ウィンドウの最後の日付を示す \$HOROLOGY 日付形式の絶対日付です。

yearopt=5 のとき、endwin は、現在の年から後の年数を示すスライディング・ウィンドウの最後の年を示す数値です。スライディング・ウィンドウは常に、endwin で指定した年の 12 月 31 日で終了します。endwin を指定しない場合、既定は、startwin 以降 100 年後の 12 月 31 日です。

endwin を省略した場合または -1 を指定した場合、スライディング・ウィンドウの有効期限は 100 年間です。endwin 値を -1 とすると特殊なケースになります。endwin の値が -1 よりも大きい場合または小さい場合に erropt が返される場合でも、-1 の場合は必ず日付値が返されます。このため、100 年間のウィンドウを指定する場合は endwin を省略し、その他の場合でも endwin には負の値を指定しないようにすることをお勧めします。

startwin と endwin の両方を指定する場合、指定するスライディング・ウィンドウの年数は 100 年以内に収める必要があります。

## mindate

有効な日付範囲 (両端を含む) の下限を指定する式です。\$HOROLOGY 整数日付カウント (例えば、1/1/2013 は 62823 と表される) または \$HOROLOGY 文字列値の形式で指定できます。\$HOROLOGY 日付文字列 ("62823,43200" など) の時刻部分は含めることも省略することもできますが、解析されるのは mindate の日付部分のみです。mindate より前の hdate 値を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

mindate に指定できる値は以下のとおりです。

- ・ 正整数 : 通常は、mindate には、1840 年 12 月 31 日より後の最も早い有効日付を設定するための正の整数を指定します。例えば、mindate に 21550 を指定すると、1900 年 1 月 1 日が最も早い有効日付として設定されます。有効な最大値は 2980013 です (9999 年 12 月 31 日)。
- ・ 0 : 1840 年 12 月 31 日を最も早い日付として指定します。これは **DateMinimum** プロパティの既定値です。
- ・ -2 以下の負の整数 : 1840 年 12 月 31 日から遡ってカウントする最も早い日付を指定します。例えば、mindate に -14974 を指定すると、1800 年 1 月 1 日が最も早い有効日付として設定されます。負の mindate 値が有効となるのは、現在のロケールの **DateMinimum** プロパティがその値以下の負の整数に設定されている場合のみです。有効な最小値は -672045 です。
- ・ mindate を省略するか、-1 を指定した場合は、現在のロケールの **DateMinimum** プロパティ値が既定値になります (localeopt=1 または mindate の既定値が 0 の場合を除く)。localeopt が未指定で dformat=3 の場合、mindate の既

定値は 0 になります。localeopt が未指定で dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、mindate の既定値は 0 になります。

**DateMinimum** プロパティは、以下のように取得および設定できます。

### ObjectScript

```
SET min=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Permit18thCenturyDates
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
SET newmin=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "set DateMinimum value is ",newmin,!
RestrictTo19thCenturyDates
WRITE $ZDATE(-13000,1,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

mindate を指定する際は、maxdate を指定しても指定しなくてもかまいません。maxdate より大きい mindate を指定すると、<ILLEGAL VALUE> エラーが発生します。

### ODBC 日付形式 (dformat 3)

**DateMinimum** プロパティの適用は、localeopt 設定によって管理されます。localeopt=1 (dformat=3 の場合の既定) の場合、現在のロケール設定に関係なく、日付の最小値は 0 になります。したがって、ODBC 形式 (dformat=3) では、以下を使用して 1840 年 12 月 31 日より前の日付を指定できます。

- 指定した日付よりも前の mindate を指定します。

### ObjectScript

```
WRITE $ZDATE(-30,3,,,,,-365)
```

- 指定した日付よりも前の **DateMinimum** プロパティ値を指定し、localeopt=0 を設定します。

### ObjectScript

```
DO ##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-365)
WRITE $ZDATE(-30,3,,,,,0)
```

## maxdate

有効な日付範囲(両端を含む)の上限を指定する式です。\$HOROLOG 整数日付カウント(例えば、1/1/2100 は 94599 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付 (“94599,43200” など)の時刻部分は含めることも省略することもできますが、解析されるのは maxdate の日付部分のみです。

maxdate を省略するか、または -1 を指定すると、現在のロケールの **DateMaximum** プロパティから日付の上限が取得されます。既定値は、\$HOROLOG の日付部分で許容される最大値である、2980013 (西暦 9999 年 12 月 31 日に対応) です。ただし、**DateMaximum** プロパティの適用は localeopt の設定によって制御されます。localeopt=1 (dformat=3 の場合の既定値) の場合、現在のロケール設定に関係なく、日付の既定最大値は ODBC の値 (2980013) になります。イスラム暦の日付形式でも ODBC の既定値が使用されます。タイ日付形式 (dformat=13) の最大日付は \$HOROLOG 2781687 で、これは BE 9999 年 12 月 31 日に対応します。

maxdate より大きい hdate を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

2980013 より大きい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

maxdate を指定する際は、mindate を指定しても指定しなくてもかまいません。mindate より小さい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

## erropt

この引数に値を指定すると、hdate 値が無効または範囲外の場合に生じるエラーが抑制されます。\$ZDATE 関数は、`<ILLEGAL VALUE>` または `<VALUE OUT OF RANGE>` エラーを生成する代わりに erropt 値を返します。

- ・ 検証: InterSystems IRIS は、hdate に対して **キャノンニック数値変換** を実行します。hdate 文字列の解析は、数値以外の最初の文字で停止します。したがって、64687AD などの hdate 文字列は、64687 と同じです。数値以外の日付 (NULL 文字列を含む) は 0 に評価されます。したがって、hdate が空の文字列の場合は、\$HOROLOG の最初の日付 (12/31/1840) が返されます。ただし、hdate が整数に評価されない (ゼロ以外の小数を含む) 場合、`<ILLEGAL VALUE>` エラーが発生します。
- ・ 範囲: hdate は常に mindate から maxdate までの範囲内の整数に評価される必要があります。既定では、2980013 を超える日付値または 0 未満の日付値は `<VALUE OUT OF RANGE>` エラーを生成します。mindate を負の数値に設定すると、1840 年 12 月 31 日より前の有効な日付の範囲を拡張できます。ただし、dformat 18、19、20、または 21 (ヒジュラ・イスラム暦) の日付の場合、-445031 より前の日付は、mindate がより早い日付に設定されていても、`<ILLEGAL VALUE>` エラーを生成します。

erropt 引数は、hdate の無効な値または範囲外の値が原因で生成されるエラーのみを抑制します。他の引数の無効な値、あるいは範囲外の値が原因で発生したエラーについては、erropt が指定されているかどうかに関係なく、常にエラーが生成されます。例えば、\$ZDATE で endwin が startwin より前になるスライディング・ウィンドウを指定すると、常に `<ILLEGAL VALUE>` エラーが発生します。同様に、maxdate が mindate より小さい場合、`<ILLEGAL VALUE>` エラーが発生します。

### ZDateNull を使用した無効な日付の処理

hdate に無効な値が指定された際の \$ZDATE の動作は、ZDateNull を使用して設定できます。現在のプロセスに対するこの動作を設定するには、%SYSTEM.Process クラスの ZDateNull() メソッドを使用します。システム全体の既定の動作は、Config.Miscellaneous クラスの ZDateNull プロパティで設定できます。\$ZDATE はエラーを発行するか、または NULL 値を返すことができます。

システム全体の既定の振る舞いは構成可能です。管理ポータルに進み、**[システム管理]**、**[構成]**、**[追加設定]**、**[互換性]** の順に選択します。**[ZDateNull]** の現在の設定を表示して編集します。既定は “false” で、これは \$ZDATE がエラーを返すことを意味します。

## localeopt

このブーリアン引数には、ロケールで指定される引数 dformat、monthlist、yearopt、mindate、および maxdate の既定値のソースとして、ユーザの現在のロケール定義または ODBC のロケール定義が指定されます。

- ・ localeopt=0 の場合は、これらすべての引数で現在のロケール定義の既定値が使用されます。
- ・ localeopt=1 の場合は、これらすべての引数で ODBC の既定値が使用されます。
- ・ localeopt が未指定の場合は、dformat 引数によってこれらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。dformat が 18、19、20、または 21 の場合は、現在のロケール定義に関係なく、イスラム暦の日付形式の既定値が使用されます。dformat がそれ以外の値の場合は、現在のロケール定義の既定値が使用されます。詳細は、“**dformat**” の説明を参照してください。

ODBC ロケールは変更できません。これは、各国語サポート (NLS) の選択が異なる InterSystems IRIS プロセス間で移植できる日付文字列をフォーマットするために使用されます。localeopt=1 の場合、ODBC ロケールの日付定義は以下のとおりです。

- ・ 日付形式の既定は 3 です。したがって、dformat が未定義または -1 の場合は、日付形式 3 が使用されます。
- ・ 日付の区切り文字の既定は “/” です。ただし、日付形式の既定は 3 で、これは日付の区切り文字として “-” を常に使用します。
- ・ 年のオプションの既定は 4 桁です。
- ・ 日付の最小値は 0 で、最大値は 2980013 (\$HOROLOG の日数カウント) です。

- 英語の月名、月の省略形、曜日名、および曜日の省略形が使用されます。

## 例

### 日付形式の例

以下の例は、現在の日付に対して、さまざまな dformat 形式をどのように \$ZDATE が返すかを示しています。yearopt は既定値を取ります。日付の区切り文字、および月と曜日の名前とその省略形はロケールに依存します。この例では、以下に示す現在のユーザのロケール定義を使用しています。

#### ObjectScript

```
WRITE $ZDATE($HOROLOG), "    default date format",!
WRITE $ZDATE($HOROLOG,1),"    1=American numeric format",!
WRITE $ZDATE($HOROLOG,2),"    2=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,3),"    3=ODBC numeric format",!
WRITE $ZDATE($HOROLOG,4),"    4=European numeric format",!
WRITE $ZDATE($HOROLOG,5),"    5=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,6),"    6=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,7),"    7=Month abbreviation format",!
WRITE $ZDATE($HOROLOG,8),"    8=Numeric format no spaces",!
WRITE $ZDATE($HOROLOG,9),"    9=Month name format",!
WRITE $ZDATE($HOROLOG,10),"    10=Day-of-week format",!
WRITE $ZDATE($HOROLOG,11),"    11=Day abbreviation format",!
WRITE $ZDATE($HOROLOG,12),"    12=Day name format",!
WRITE $ZDATE($HOROLOG,13),"    13=Thai numeric format",!
WRITE $ZDATE($HOROLOG,14),"    14=Day-of-year format",!
WRITE $ZDATE($HOROLOG,15),"    15=European numeric format",!
WRITE $ZDATE($HOROLOG,16),"    16=Japanese date format",!
WRITE $ZDATE($HOROLOG,17),"    17=Japanese date format with spaces"
```

以下の例は、現在のユーザのロケールを既定にしたときのロケールの日付と localeopt=1 が ODBC ロケール定義を有効化したときの日付とを比較します。この例をより興味深いものにするために、現在のユーザのロケールをフランス語に設定します。

#### ObjectScript

```
SET fmt=##class(%SYS.NLS.Format).%New("fraw")
WRITE "default: local=", $ZDATE($HOROLOG), "    ODBC=", $ZDATE($HOROLOG,,,,,,,1),!
WRITE "-1:    local=", $ZDATE($HOROLOG,-1), "    ODBC=", $ZDATE($HOROLOG,-1,,,,,,,1),!!
FOR x=1:1:17 {
    WRITE x,": local=", $ZDATE($HOROLOG,x), "    ODBC=", $ZDATE($HOROLOG,x,,,,,,,1),! }
```

## 2 桁の年のスライディング・ウィンドウの例

明示的なスライディング・ウィンドウの使用法を示すために、以下の関数呼び出しを 1997 年に入力したと仮定します。59461 の hdate に相当する日付は、2003 年 10 月 19 日です。dformat に 1 を指定すると、2 桁または 4 桁の年が返されます。yearopt に 5 を指定すると、4 桁の年のスライディング・ウィンドウが指定されます。yearopt 設定に従い、startwin および endwin が、現在の年（この場合 1997 年）から相対的に、加算および減算によって計算されます。

#### ObjectScript

```
WRITE $ZDATE(59461,1,,5,90,10)
```

2 桁で年を表示するスライディング・ウィンドウは、1/1/1907 から 12/31/2006 までの範囲に指定されるため、InterSystems IRIS は 10/19/03 のように日付を表示します。

## 日付範囲の例

以下の例は、mindate と maxdate を使用して、妥当な誕生日をテストします。maxdate は、誕生日が未来になることはない想定し、mindate は、リストされた人で 124 歳を超える人はいないと想定します。日付は、\$HOROLOG 形式で指定されます。



## ObjectScript

```

PlausibleBirthdate
SET bdateh(1)=62142
SET bdateh(2)=16800
SET bdateh(3)=70000
DO $SYSTEM.Process.ZDateNull(1)
SET maxdate=$PIECE($HOROLOG,"",1)+1
SET mindate=maxdate-(365.25*124)
FOR x=1:1:3 {
    SET bdate=$ZDATE(bdateh(x),,,,,mindate,maxdate)
    IF bdate="" {WRITE "Birth date ",bdateh(x)," is out of range",!}
    ELSE {WRITE "Birth date ",bdateh(x)," is ",bdate,!}
}

```

上記の 2 つの \$ZDATE 入力値が誕生日テストの日付範囲の外側になります。つまり、16800 (12/30/1886) は 124 年より前で、70000 (08/26/2032) は未来です。既定では、このように \$ZDATE を呼び出すと、<VALUE OUT OF RANGE> エラーが発生しますが、ZDateNull(1) が設定されているため、空の文字列(“)が返されます。

## \$ZDATE で無効な値

以下の状況では、<FUNCTION> エラーが発生します。

- ・ 無効な dformat コード (-1 未満の整数値、17 より大きな整数値、または整数以外の値) を指定した場合。
- ・ yearopt が 3 あるいは 5 の場合に、startwin 値を指定しない場合。

以下の状況で、<ILLEGAL VALUE> エラーが発生します。

- ・ hdate に無効な値を指定し、erropt 値または ZDateNull (以下で説明します) を設定しない場合。
- ・ 指定された月数が monthlist の値よりも大きい場合。
- ・ maxdate が mindate よりも小さい場合。
- ・ endwin が startwin よりも小さい場合。
- ・ startwin と endwin で、期間が 100 年より長い一時的なスライディング・ウィンドウが指定された場合。

以下の状況では、<VALUE OUT OF RANGE> エラーが発生します。

- ・ 有効な日付範囲外の hdate を指定した場合。InterSystems IRIS の場合、これは 0 から 298013 までの数になります。%SYSTEM.Process クラスの ZDateNull() メソッドを使用して、現在のプロセスに対する日付範囲と無効な日付の動作を設定できます。
- ・ maxdate と mindate に対して想定される値によって定義された範囲外にある、範囲内にあれば有効であった日付を指定し、erropt 値を指定しない場合。

## ユーティリティの代わりに使用する \$ZDATE

\$ZDATE 関数と日付ユーティリティのいずれかを選択する必要がある場合、以下の点に注意してください。

- ・ %DO あるいは %D ユーティリティにある既存のエントリ・ポイントの代わりに \$ZDATE 関数を使用できます。
- ・ INT ^%D を呼び出さずに、\$ZDATE(\$HOROLOG,7) を直接呼び出すことができます。これにより、現在の日付が “Mmm DD [YY]YY” 形式で表示されます。
- ・ \$ZDATEH と \$ZDATE は、%DATE、%DI、%DO のエントリ・ポイントの呼び出しよりも高速に処理できます。

## 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$ZDATEH 関数](#)

- ・ [\\$ZDATETIME](#) 関数
- ・ [\\$ZDATETIMEH](#) 関数
- ・ [\\$ZTIME](#) 関数
- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数
- ・ %DATE ユーティリティ (<https://docs.intersystems.com/priordocexcerpts> で利用可能な旧ドキュメントに記載)
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ZDATEH (ObjectScript)

---

日付を検証し、これを表示形式から InterSystems IRIS の内部形式に変換します。

### 構文

```
$ZDATEH(date,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)  
$ZDH(date,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
```



## 引数

引数	説明
date	表示形式で、有効な日付文字列として評価される式。\$ZDATEH は、この日付文字列を \$HOROLOGY 形式に変換します。これは、現在の日付を表す明示的な日付 (さまざまな形式で指定)、あるいは "T" または "t" 文字列です。"T" または "t" 文字列は、オプションで符号付きの整数のオフセットを含むことができます。例えば、"T-7" は現在の日付の 7 日前を表します。後述の "date" を参照してください。
dformat	オプション - date の日付形式オプションを指定する整数コード。date が "T" の場合、dformat は 5、6、7、8、9、または 15 のいずれかとする必要があります。後述の "dformat" を参照してください。
monthlist	オプション - 一連の月名を表す文字列または変数の名前。この文字列は、区切り文字から始まり、12 個のエントリは、この区切り文字で分けられる必要があります。後述の "monthlist" を参照してください。
yearopt	オプション - 年を 2 桁または 4 桁のどちらで表記するかを指定する整数コード。後述の "yearopt" を参照してください。
startwin	オプション - 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値。後述の "startwin" を参照してください。
endwin	オプション - 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最後を指定する数値。後述の "endwin" を参照してください。
mindate	オプション - 有効な date 日付範囲の下限。\$HOROLOGY 整数日付カウントの形式で指定し、0 は 1840 年 12 月 31 日を表します。正または負の整数として指定できます。後述の "mindate" を参照してください。
maxdate	オプション - 有効な日付範囲の上限。\$HOROLOGY 整数日付カウントの形式で指定します。後述の "maxdate" を参照してください。
erropt	オプション - date が無効のときに返す式。この引数に値を指定すると、date 値が無効または範囲外の場合に生じるエラー・コードが抑制されます。エラー・メッセージを発行する代わりに、\$ZDATEH は erropt を返します。後述の "erropt" を参照してください。
localeopt	<p>オプション - dformat、monthlist、yearopt、mindate、maxdate の既定値、およびその他の日付特性 (日付区切り文字など) に対してどのロケールを使用するかを指定するブーリアン・フラグ。</p> <p>localeopt=0: 現在のロケール・プロパティ設定によって、これらの引数の既定値が決定されます。</p> <p>localeopt=1: ODBC 標準ロケールによって、これらの引数の既定値が決定されます。</p> <p>localeopt 未指定: dformat 値によって、これらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。日本およびイスラム暦の日付の dformat 値については、それぞれの既定値が使用されます。その他すべての dformat 値については、現在のロケール・プロパティ設定が既定値として使用されます。後述の "localeopt" を参照してください。</p>

指定の引数値間で省略された引数は、プレースホルダのコンマで示されます。末尾のプレースホルダのコンマは必要ありませんが、あってもかまいません。省略された引数を示すコンマの間に空白があってもかまいません。

## 概要

\$ZDATEH 関数は、指定した日付を検証し、\$ZDATE 関数でサポートされる形式から \$HOROLOG 形式に変換します。  
\$ZDATEH が実行する動作は、使用する引数によって決まります。

### 単純な \$ZDATEH 形式

\$ZDATEH(date) は、MM/DD/[YY]YY 形式から \$HOROLOG 形式の最初の整数に日付を変換します (\$HOROLOG は、2 つの整数で構成されます。最初の整数は日付、2 番目の整数は時刻です。)1900 年から 1999 年までは 2 桁または 4 桁で指定し、1900 年より前、または 1999 年より後の年は 4 桁で指定します。

### \$ZDATEH 形式のカスタマイズ

\$ZDATEH(date,dformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,errorpt) は、指定された dformat の日付を \$HOROLOG 形式に変換します。dformat、monthlist、yearopt、startwin、endwin、mindate、maxdate、および errorpt の値は、\$ZDATE によって使用される値と同一です。ただし、dformat に 5、6、7、8、9 を使用している場合、\$ZDATEH は dformat コード 1、2、3、5、6、7、8、9 に対して定義されている (dformat コード 4 は除きます)、あらゆる外部日付形式の日付を認識して変換します。また、特別な相対 date 形式も認識します。相対 date 形式は、T または t ("today" の意味) で始まる文字列の後に、オプションでプラス記号 (+) またはマイナス記号 (-)、および現在の日付より前の、あるいは後の日数を表す整数値が続きます。

## 引数

### date

\$HOROLOG 形式に変換する日付。引用符付き文字列として指定します。これは、明示的な日付、あるいは、"T" または "t" 文字列で表す暗黙の現在日付です。

明示的な日付は、dformat によってサポートされる形式で指定する必要があります。許可される形式は、dformat 引数に応じて異なります。dformat が指定されていない場合、あるいは 1、2、3、4 の場合、許可される日付形式は 1 つだけです。dformat が 5、6、7、8、9、または 15 の場合は、複数の日付形式を使用できます。

dformat が 5、6、7、8、または 9 の場合、\$ZDATEH では、曖昧ではないすべてのアメリカ日付形式を使用できます。dformat が 15 の場合、\$ZDATEH では、曖昧ではないすべてのヨーロッパ日付形式を使用できます。有効な日付形式のリストは、以下を参照してください。\$ZDATEH では、02/03/02 (2002 年 2 月 3 日) とヨーロッパ形式の 02/03/02 (2002 年 3 月 2 日) を区別できないので、dformat=4 は有効なアメリカ日付形式ではありません。許可されていない形式で日付を指定した場合や存在しない日付 (2002 年 2 月 31 日など) を指定した場合、\$ZDATEH では <ILLEGAL VALUE> エラー・コードが生成されます (\$ZDATEH は、うるう年の確認も実行します。したがって、2004 年 2 月 29 日は許可されますが、2003 年 2 月 29 日は許可されません)。

ロシア、ウクライナ、およびチェコのロケールでは、date の値は、DD.MM.YYYY のように、スラッシュではなくピリオドを日付部分の区切り文字として使用して指定する必要があります。

暗黙日付は、現在の (今日の) 日付を示す "T" または "t" の文字から成る文字列として指定されます。この文字列は、オプションでプラス記号、またはマイナス記号、および整数を含みます。これは、現在の日付からのオフセットの日数を指定します。例えば、"t+9" は現在の日付の 9 日後を表し、"t-12" は現在の日付の 12 日前を表します。暗黙日付は、dformat が 5、6、7、8、9、または 15 の場合にのみ使用できます。使用できる唯一の暗黙日付の形式は、"T" (または "t")、あるいは "T" (または "t") の後に符号と整数を続けて記述する形式です。整数以外の値、算術式、符号なしの整数、または整数なしの符号を指定した場合、InterSystems IRIS は <ILLEGAL VALUE> エラーを生成します。"T+0" および "T-0" は許可されており、現在の日付を返します。\$HOROLOG 日付が有効な日付範囲外となるオフセットを指定した場合は、InterSystems IRIS は <VALUE OUT OF RANGE> エラーを生成します。

既定では、最も早い有効な date は 1840 年 12 月 31 日です (内部 \$HOROLOG 表現では 0)。DateMinimum プロパティの既定値は 0 であるため、既定では、日付は正の整数に制限されています。現在のロケールの DateMinimum プロパティが、過去の日付を表す負の整数以下の負の整数に設定されていれば、過去の日付を負の整数を使用して指定できます。DateMinimum の有効な最小値は -672045 であり、これは 0001 年 1 月 1 日に相当します。InterSystems IRIS では、ISO 8601 標準に準拠してグレゴリオ暦が "西暦 1 年" まで遡って適用された先発グレゴリオ暦が使用されていま

す。その理由の 1 つは、グレゴリオ暦が採用された時期は国ごとに異なるからです。例えば、欧州大陸諸国の多くでは 1582 年に採用された一方で、英国と米国では 1752 年に採用されました。このため、ご使用の地域でグレゴリオ暦が採用された時期より前の InterSystems IRIS の日付は、その当時にその地域で採用されていた暦に基づいて記録された歴史上の日付に対応していない可能性があります。1840 年より前の日付の詳細は、“mindate” 引数を参照してください。

## dformat

日付の形式です。有効な値は以下のとおりです。

値	意味
-1	有効な dformat 値は、 <a href="#">現在のロケール</a> の DateFormat プロパティから取得します。dformat を指定しなければ、これが既定の振る舞いになります。
1	MM/DD/[YY]YY (07/01/97 または 03/27/2002) – <a href="#">アメリカの数値形式</a> 。現在のロケール用の正しい日付区切り文字 (/ または .) を指定する必要があります。
2	DD Mmm [YY]YY (01 Jul 97)
3	[YY]YY-MM-DD (1997-07-01) – ODBC 形式
4	DD/MM/[YY]YY (01/07/97 または 27/03/2002) – <a href="#">ヨーロッパの数値形式</a> 。現在のロケール用の正しい日付区切り文字 (/ または .) を指定する必要があります。
5	Mmm D, YYYY (Jul 1, 1997)、またはあらゆる <a href="#">曖昧ではないアメリカ日付形式</a> 。
6	Mmm D YYYY (Jul 1 1997)、またはあらゆる <a href="#">曖昧ではないアメリカ日付形式</a> 。
7	Mmm DD [YY]YY (Jul 01 1997)、またはあらゆる <a href="#">曖昧ではないアメリカ日付形式</a> 。
8	[YY]YYMMDD (19970701) – 数値形式、またはあらゆる <a href="#">曖昧ではないアメリカ日付形式</a> 。
9	Mmmmm D, YYYY (July 1, 1997)、またはあらゆる <a href="#">曖昧ではないアメリカ日付形式</a> 。
13	[D]D/[M]M/YYYY (1/7/2549 または 27/11/2549) – タイ語の日付形式。日と月は、ヨーロッパ形式と同じですが、先頭に 0 は付きません。年は仏教紀元 (BE) で、グレゴリオ暦の年に 543 年追加して計算されます。
15	DD/MM/[YY]YY、YYYY-MM-DD、日付区切り文字を使用したあらゆる <a href="#">曖昧ではないヨーロッパ日付形式</a> 、または日付区切り文字を使用しない YYYYMMDD。日付区切り文字は、現在のロケールで指定されている日付区切り文字に関係なく、英数字以外の任意の文字 (空白も含む) ならどれでもかまいません。また、monthlist 名および “T” も使用できます。
16	YYYYc[M]Mc[D]Dc – 日本の日付形式。年、月、および日の数は、他の日付形式と同じです。先頭のゼロは省略されます。“年”、“月”、および“日”の日本語の文字 (ここでは c として表示) は、年、月、および日の数の後に挿入されます。これらの文字は、年=\$CHAR(24180)、月=\$CHAR(26376)、および日=\$CHAR(26085) です。
17	YYYYc[M]Mc[D]Dc – 日本の日付形式。dformat 16 と同じです。ただし、“年”および“月”の日本語の文字の後に空白が挿入されます。
18	[D]D Mmmmm YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月名を示す)。日の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 Muharram 0001。
19	[D]D [M]M YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月番号を示す)。日と月の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 1 0001。

値	意味
20	[D]D Mmmmm YYYY – Observed Hijri (イスラム) 日付形式 (完全な月名を示す)。既定で Tabular Hijri (dformat 18) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。
21	[D]D [M]M YYYY – Observed Hijri (イスラム) 日付形式 (完全な月番号を示す)。既定で Tabular Hijri (dformat 19) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。

以下は、この指定の説明です。

構文	意味
YYYY	YYYY は 4 桁の年です。[YY]YY は、日付がアクティブ・ウィンドウ内で 2 桁の場合は 2 桁の年、それ以外は 4 桁の年です。日付形式 (dformat) 1 から 4 までを使用するときは、年の値を提供する必要があります。これらの日付形式では、不足している年の値は提供されません。日付形式 5 から 9 は、指定する日付に年が含まれない場合、現在の年が想定されます。
MM	2 桁の月。
D	日付が 10 未満の場合は 1 桁、それ以外は 2 桁の日。
DD	2 桁の日。
Mmm	現在のロケールの MonthAbbr プロパティから取得した月の省略形。以下は英語での既定値です。 “Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”。別の月の省略形 (あるいは長さに制限のない名前) は、\$ZDATEH に monthlist 引数として指定したオプションのリストから取得することができます。
Mmmmm	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。英語での既定値は、 “January February March ...November December” です。

### dformat の既定値

dformat を省略するか、-1 に設定した場合、dformat の既定値は localeopt 引数と NLS の **DateFormat** プロパティに依存します。

- ・ localeopt=1 の場合、dformat の既定値は ODBC 形式です。monthlist、yearopt、mindate、および maxdate 引数の既定値も、ODBC 形式に設定されます。これは、dformat=3 を設定した場合と同じです。
- ・ localeopt=0 または未指定の場合、dformat の既定値は NLS の **DateFormat** プロパティから取得されます。  
**DateFormat=3** の場合、dformat の既定値は ODBC 形式です。ただし、DateFormat=3 は monthlist、yearopt、mindate、および maxdate 引数の既定値に影響しません。これらは、NLS の現在のロケール定義で指定されます。

ユーザのロケールの既定の日付プロパティを決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateSeparator")
```

\$ZDATEH は、dformat=1 または 4 の場合に、[現在のロケールのDateSeparator プロパティ](#)の値 (/ または .) を、月、日、および年の区切り文字として使用します。

ヨーロッパ日付形式 (dformat=4、DD/MM/YYYY の順) は、ヨーロッパの多くの言語 (すべてではない) で既定です。これには、“/” を **DateSeparator** 文字として使用するイギリス英語、フランス語、ドイツ語、イタリア語、スペイン語、ポルトガル語、および “.” を **DateSeparator** 文字として使用するチェコ語 (csyw)、ロシア語 (rusw)、スロバキア語 (skyw)、スロベニア語 (svnw)、ウクライナ語 (ukrw) が含まれます。サポートされるロケールの既定の日付形式の詳細は、“[日付](#)”を参照してください。

## dformat の設定

dformat が 3 (ODBC 形式の日付) の場合は、monthlist、yearopt、mindate、および maxdate 引数の既定値にも ODBC 形式の既定値が使用されます。日付区切り文字は常に “-” になります。現在のロケールの既定値は無視されます。

dformat が 16 または 17 (日本の日付形式) の場合、日付形式はロケールの設定に依存しません。日本の日付形式は、任意の InterSystems IRIS インスタンスで使用できます。

dformat が 18、19、20、または 21 (イスラム暦の日付形式) で、localeopt が未指定の場合、引数の既定値は現在のロケールの既定値ではなく、イスラム暦の既定値になります。monthlist 引数の既定値は、ラテン文字で書き直されたアラビア語の月名になります。yearopt、mindate、および maxdate 引数の既定値は、ODBC の既定値になります。日付区切り文字の既定値は、ODBC の既定値や現在のロケールの **DateSeparator** プロパティ値ではなく、イスラム暦の既定値 (空白) になります。localeopt=0 の場合は、現在のロケール・プロパティの既定値がこれらの引数に使用されます。localeopt=1 の場合は、ODBC の既定値がこれらの引数に使用されます。

## monthlist

区切り文字で区切られた月名または月の省略名の文字列に解決される式です。monthlist にある名前は、現在のロケールの **MonthAbbr** プロパティに指定されている既定の月の省略形の値、または **MonthName** に指定されている月名の値を置き換えます。

monthlist は、dformat が 2、5、6、7、8、9、15、18、または 20 の場合にのみ有効です。dformat がそれ以外の値の場合、\$ZDATEH は monthlist を無視します。

monthlist 文字列の形式は、以下のとおりです。

- 文字列の最初の文字は、区切り文字 (通常はスペース) です。monthlist 内の最初の月名の前と、各月名の間に、同じ区切り文字を置く必要があります。1 文字からなる任意の区切り文字を指定できます。この区切り文字は、指定された date 値の月、日、年の間に指定する必要があります。このため、スペースが通常推奨される区切り文字となります。
- 月名の文字列は、1 月 から 12 月に対応する 12 個の区切り値を含む必要があります。12 より多い、または 12 より少ない月名を指定することは可能ですが、date の月に該当する月名がない場合、<ILLEGAL VALUE> エラーが発生します。

monthlist を省略するか、monthlist の値として -1 を指定した場合は、現在のロケールの **MonthAbbr** または **MonthName** プロパティで定義されている月名のリストが \$ZDATEH で使用されます (localeopt=1、または monthlist の既定値が ODBC の月リスト (英語) である場合を除く)。localeopt が未指定で、dformat が 18 または 20 (イスラム暦の日付形式) の場合、monthlist の既定値はイスラム暦の月リスト (ラテン文字で表されたアラビア語の名前) になり、**MonthAbbr** および **MonthName** プロパティ値は無視されます。

ユーザのロケールの既定の月名および月名の省略形を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```

## yearopt

年を 2 桁で表示するか 4 桁で表示するかを指定する数値コード。有効な値は以下のとおりです。

値	意味
-1	現在のロケールの YearOption プロパティから有効な yearopt 値を取得します。既定値は 0 です。yearopt を指定しない場合は、これが既定の動作になります。



値	意味
0	プロセス特有の (%DATE ユーティリティにより設定された) スライディング・ウィンドウが実行されていない限り、1900 年代 (1900 から 1999 まで) の日付は 2 桁で表示します。スライディング・ウィンドウが有効な場合は、指定範囲内の日付のみ 2 桁の年で表示します。1900 年代以外もしくはプロセス特有のスライディング・ウィンドウ以外の日付は、すべて 4 桁で表示します。
1	有効になっている一時的なスライディング・ウィンドウには関係なく、1900 年代の日付は 2 桁の年、それ以外は 4 桁の年で表示します。
2	一時的に実行されているスライディング・ウィンドウには関係なく、日付はすべて 2 桁の年で表します。日付はすべて 1900 年代と想定されます。このオプションにより、4 桁の年から 2 桁が削除されるため、このオプションを使用すると、世紀に関する情報が失われて復元不能になります。(日付がすべて同じ世紀内のものであれば、情報を失っても大きな問題にはなりません。)
3	startwin と (オプションで) endwin に定義した一時的なスライディング・ウィンドウ範囲内に収まる日付を、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt=3 の場合、startwin と endwin は、\$HOROLOG 形式の絶対日付です。
4	すべての日付を 4 桁の年で表します。2 桁で入力された日付は、無効として拒否されます。
5	startwin と (オプションで) endwin に定義した一時的なスライディング・ウィンドウ範囲内に収まる日付を、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt=5 の場合、startwin と endwin は相対年になります。
6	2 桁の年で現在の世紀にあるすべての日付を表します。それ以外は 4 桁の年で表します。

yearopt を省略するか、yearopt の値として -1 を指定した場合は、現在のロケールの YearOption プロパティが \$ZDATEH で使用されます (localeopt=1、または yearopt の既定値が ODBC の年オプションである場合を除く)。localeopt=0 または未指定で、dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、yearopt の既定値は ODBC の年オプション (4 桁の年) になります。イスラム暦の日付では、YearOption プロパティの値は無視されます。

### startwin

日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値です。yearopt の値に 3 または 5 を使用する場合は、startwin を指定する必要があります。startwin は、他の yearopt 値では無効になります。

yearopt=3 のとき、startwin はスライディング・ウィンドウの最初の日付を示す \$HOROLOG 日付形式の絶対日付です。

yearopt=5 のとき、startwin は、現在の年より前の年数で表される、スライディング・ウィンドウの最初の年を示す数値です。スライディング・ウィンドウは常に、startwin で指定した年の初日 (1 月 1 日) から開始します。

### endwin

日付を 2 桁の年で表すスライディング・ウィンドウの最後を指定する数値です。yearopt が 3 あるいは 5 のとき、endwin をオプションで指定する必要があります。endwin は、その他の yearopt 値では無効になります。

yearopt=3 のとき、endwin はスライディング・ウィンドウの最後の日付を示す \$HOROLOG 日付形式の絶対日付です。

yearopt=5 のとき、endwin は、現在の年から後の年数を示すスライディング・ウィンドウの最後の年を示す数値です。スライディング・ウィンドウは常に、endwin で指定した年の最後の日 (12 月 31 日) または、(endwin を省略した場合は) 暗示されている最後の年の最後の日 (12 月 31 日) で終了します。

endwin を省略した場合または -1 を指定した場合、スライディング・ウィンドウの有効期限は 100 年間です。endwin 値を -1 とすると特殊なケースになります。endwin の値が -1 よりも大きい場合または小さい場合に errorpt が返される場合でも、-1 の場合は必ず日付値が返されます。このため、100 年間のウィンドウを指定する場合は endwin を省略し、その他の場合でも endwin には負の値を指定しないようにすることをお勧めします。

startwin と endwin の両方を指定する場合、指定するスライディング・ウィンドウの年数は 100 年以内に収める必要があります。

## mindate

有効な日付範囲(両端を含む)の下限を指定する式です。\$HOROLOG 整数日付カウント(例えば、1/1/2013 は 62823 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付 (“62823,43200” など)の時刻部分は含めることも省略することもできますが、解析されるのは mindate の日付部分のみです。mindate より前の date 値を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

mindate に指定できる値は以下のとおりです。

- ・ 正整数：通常は、mindate には、1840 年 12 月 31 日より後の最も早い有効日付を設定するための正の整数を指定します。例えば、mindate に 21550 を指定すると、1900 年 1 月 1 日が最も早い有効日付として設定されます。有効な最大値は 2980013 です (9999 年 12 月 31 日)。
- ・ 0：1840 年 12 月 31 日を最も早い日付として指定します。これは **DateMinimum** プロパティの既定値です。
- ・ -2 以下の負の整数：1840 年 12 月 31 日から遡ってカウントする最も早い日付を指定します。例えば、mindate に -14974 を指定すると、1800 年 1 月 1 日が最も早い有効日付として設定されます。負の mindate 値が有効となるのは、現在のロケールの **DateMinimum** プロパティがその値以下の負の整数に設定されている場合のみです。有効な最小値は -672045 です。
- ・ mindate を省略するか、-1 を指定した場合は、現在のロケールの **DateMinimum** プロパティ値が既定値になります (localeopt=1 または mindate の既定値が 0 の場合を除く)。localeopt が未指定で dformat=3 の場合、mindate の既定値は 0 になります。localeopt が未指定で dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、mindate の既定値は 0 になります。

**DateMinimum** プロパティは、以下のように取得および設定できます。

## ObjectScript

```
SET min=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Permit18thCenturyDates
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
SET newmin=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "set DateMinimum value is ",newmin,!
RestrictTo19thCenturyDates
WRITE $ZDATEH("05/29/1805",1,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

mindate を指定する際は、maxdate を指定しても指定しなくてもかまいません。maxdate より大きい mindate を指定すると、<ILLEGAL VALUE> エラーが発生します。

## maxdate

有効な日付範囲(両端を含む)の上限を指定する式です。\$HOROLOG 整数日付カウント(例えば、1/1/2100 は 94599 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付 (“94599,43200” など)の時刻部分は含めることも省略することもできますが、解析されるのは maxdate の日付部分のみです。

maxdate を省略するか、または -1 を指定すると、現在のロケールの **DateMaximum** プロパティから日付の上限が取得されます。既定値は、\$HOROLOG の日付部分で許容される最大値である、2980013 (西暦 9999 年 12 月 31 日に対応) です。ただし、**DateMaximum** プロパティの適用は localeopt の設定によって制御されます。localeopt=1 (dformat=3 の場合の既定値) の場合、現在のロケール設定に関係なく、日付の既定最大値は ODBC の値 (2980013) になります。イスラム暦の日付形式でも ODBC の既定値が使用されます。タイ日付形式 (dformat=13) の最大日付は BE 9999 年 12 月 31 日で \$HOROLOG 2781687 に対応します。

maxdate より大きい date を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

2980013 より大きい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

maxdate を指定する際は、mindate を指定しても指定しなくてもかまいません。mindate より小さい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

## erroppt

この引数に値を指定すると、date 値が無効または範囲外の場合に生じるエラーが抑制されます。\$ZDATEH 関数は、`<ILLEGAL VALUE>` または `<VALUE OUT OF RANGE>` エラーを生成する代わりに erroppt 値を返します。

InterSystems IRIS は、date に対して標準の数値評価を実行します。これは常に mindate から maxdate までの範囲内の整数に評価される必要があります。したがって、7、"7"、+7、0007、7.0、"7 dwarves"、--7 は、すべて同じ日付値 (01/07/1841) に評価されます。既定では、2980013 を超える値または 0 未満の値は `<VALUE OUT OF RANGE>` エラーを生成します。小数値は `<ILLEGAL VALUE>` エラーを生成します。数値以外の文字列 (NULL 文字列を含む) は 0 に評価され、\$HOROLOG の最初の日付 (12/31/1840) が返されます。

erroppt 引数は、date の無効な値または範囲外の値が原因で生成されるエラーのみを抑制します。他の引数の無効な値、あるいは範囲外の値が原因で発生したエラーについては、erroppt が指定されているかどうかに関係なく、常にエラーが生成されます。例えば、\$ZDATEH で endwin が startwin より前になるスライディング・ウィンドウを指定すると、常に `<ILLEGAL VALUE>` エラーが発生します。同様に、maxdate が mindate より小さい場合、`<ILLEGAL VALUE>` エラーが発生します。

## localeopt

このブーリアン引数には、ロケールで指定される引数 dformat、monthlist、yearopt、mindate、および maxdate の既定値のソースとして、ユーザの現在のロケール定義または ODBC のロケール定義が指定されます。

- ・ localeopt=0 の場合は、これらすべての引数で現在のロケール定義の既定値が使用されます。
- ・ localeopt=1 の場合は、これらすべての引数で ODBC の既定値が使用されます。
- ・ localeopt が未指定の場合は、dformat 引数によってこれらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。dformat が 18、19、20、または 21 の場合は、現在のロケール定義に関係なく、イスラム暦の日付形式の既定値が使用されます。dformat がそれ以外の値の場合は、現在のロケール定義の既定値が使用されます。詳細は、"[dformat](#)" の説明を参照してください。

ODBC ロケールは変更できません。これは、各国語サポート (NLS) の選択が異なる InterSystems IRIS プロセス間で移植できる日付文字列をフォーマットするために使用されます。localeopt=1 の場合、ODBC ロケールの日付定義は以下のとおりです。

- ・ 日付形式の既定は 3 です。したがって、dformat が未定義または -1 の場合は、日付形式 3 が使用されます。
- ・ 日付の区切り文字の既定は "/" です。ただし、日付形式の既定は 3 で、これは日付の区切り文字として "-" を常に使用します。
- ・ 年のオプションの既定は 4 桁です。
- ・ 日付の最小値は 0 で、最大値は 2980013 (\$HOROLOG の日数カウント) です。
- ・ 英語の月名、月の省略形、曜日名、および曜日の省略形が使用されます。

## 例

以下の例は、1983 年 6 月 12 日の \$HOROLOG 日付を返します。

### ObjectScript

```
WRITE $ZDATEH("06/12/83")
```

これは、52027 を返します。

以下の例は、1902 年 6 月 12 日の \$HOROLOG 日付を返します。

### ObjectScript

```
WRITE $ZDATEH("06/12/02")
```



これは、22442 を返します。

注釈 既定では、年を 2 桁で表すと 1900 年代の日付を表すものと見なされます。21 世紀以降の日付を表すには、4 桁の年を指定するか、yearopt、startwin、endwin 引数を指定して、2 桁のスライディング・ウィンドウを変更します。スライディング・ウィンドウは、ユーザのロケールに設定することもできます。

以下の例は、複数の日付エン트리形式を許可するために dformat 引数を使用する方法を示します。

#### ObjectScript

```
WRITE !,$ZDATEH("November 2, 1954",5)
WRITE !,$ZDATEH("Nov 2, 1954",5)
WRITE !,$ZDATEH("Nov. 2 1954",5)
WRITE !,$ZDATEH("11/2/1954",5)
WRITE !,$ZDATEH("11.02.54",5)
WRITE !,$ZDATEH("11 02 1954",5)
```

すべて 41578 を返します。

以下の例で、現在の日付が 2007 年 1 月 16 日であるとしします。

#### ObjectScript

```
WRITE $HOROLOG
```

現在の日付を表す最初の整数である、60646,37854 を返します (2 つ目の整数は、経過秒数での現在時刻です)。

次の例は、“T”date を使用して今日の日付 (2007 年 1 月 16 日) を返します。

#### ObjectScript

```
WRITE $ZDATEH("T",5)
```

これは、60646 を返します。

次の例は、プラス 2 日およびマイナス 2 日のオフセットで、現在の日付を返します。

#### ObjectScript

```
WRITE !,$ZDATEH("T+2",5)
WRITE !,$ZDATEH("T-2",5)
```

60648 と 60644 を返します。

最後の例では、年を指定しない場合には \$ZDATEH で現在の年 (この場合は 2007 年) が想定されることを示しています。

#### ObjectScript

```
WRITE $ZDATEH("25 Nov",5)
```

これは、60959 を返します。

## \$ZDATEH で無効な値

以下の状況では、<FUNCTION> エラーが発生します。

- ・ 無効な dformat コード (-1、1、2、3、4、5、6、7、8、9、および 15 のどれでもない整数値、0、または整数以外の値) を指定した場合。
- ・ 無効な yearopt コード (-1 未満または 6 より大きい整数値、0、整数以外の値) を指定した場合。
- ・ yearopt が 3 あるいは 5 のときに、startwin 値を指定しない場合。

以下の状況では、<ILLEGAL VALUE> エラーが発生します。

- ・ 日付単位 (日、月、年) に無効な値を指定した場合。指定した場合は、<ILLEGAL VALUE> が発行されるのではなく、errop 値が返されます。
- ・ ODBC 日付形式の日付単位 (日、月、年) に余分な先頭のゼロを指定した場合。例えば、2007 年 2 月 3 日は、“2007-2-3” または “2007-02-03” で表せますが、“2007-002-03” に対して <ILLEGAL VALUE> を受け取ります。指定した場合は、<ILLEGAL VALUE> が発行されるのではなく、errop 値が返されます。
- ・ 指定された月数が monthlist の値よりも大きい場合。
- ・ maxdate が mindate よりも小さい場合。
- ・ endwin が startwin よりも小さい場合。
- ・ startwin と endwin で、期間が 100 年より長い一時的なスライディング・ウィンドウが指定された場合。

以下の状況では、<VALUE OUT OF RANGE> エラーが発生します。

- ・ 1840 年 12 月 31 日より前、あるいは 9999 年 12 月 31 日より後の日付 (あるいは “T” のオフセット) を指定して、errop 値を指定しない場合。
- ・ mindate と maxdate の範囲外にある、範囲内にあれば有効であった日付 (あるいは “T” のオフセット) を指定し、errop 値を指定しない場合。

## 5、6、7、8、9、または 15 を dformat に指定した場合の日付形式

\$ZDATEH の dformat に 5 ～ 9 を指定した日付形式では、曖昧ではないアメリカ形式の日付値をすべて使用できます。  
\$ZDATEH の dformat に 15 を指定した日付形式では、曖昧ではないヨーロッパ形式の日付値をすべて使用できます。  
これらの日付形式で年を指定していない場合は、現在の年が指定されているものと見なされます。

以下の形式がサポートされています。

アメリカ形式 : dformat が 5、6、7、8、または 9	ヨーロッパ形式 : dformat が 15
MM/DD	DD/MM
MM-DD	DD-MM
MM DD	DD MM
MM/DD/YY	DD/MM/YY
MM-DD-YY	DD-MM-YY
MM DD YY	DD MM YY
MM/DD/YYYY	DD/MM/YYYY
MM-DD-YYYY	DD-MM-YYYY
MM DD YYYY	DD MM YYYY
YYYYMMDD	YYYYMMDD
YYMMDD	YYMMDD
YYYY-MM-DD	YYYY-MM-DD
YYYY MM DD	YYYY MM DD
Mmm D	Mmm D
Mmm D, YY	Mmm D, YY
Mmm D, YYYY	Mmm D, YYYY
Mmm D YY	Mmm D YY
Mmm D YYYY	Mmm D YYYY
Mmm DD	Mmm DD
Mmm DD YY	Mmm DD YY
Mmm DD YYYY	Mmm DD YYYY
DD Mmm	DD Mmm
DD Mmm YY	DD Mmm YY
DD Mmm YYYY	DD Mmm YYYY
DD-Mmm	DD-Mmm
DD-Mmm-YY	DD-Mmm-YY
DD-Mmm-YYYY	DD-Mmm-YYYY
YYYY Mmm DD	YYYY Mmm DD

MMDD は、実装されていない形式です。

## ユーティリティの代わりに使用する \$ZDATEH

\$ZDATEH 関数と日付ユーティリティのいずれかを選択する必要がある場合、以下の点に注意してください。

- ・ %DATE あるいは %DI ユーティリティにある既存のエントリ・ポイントの代わりに \$ZDATEH 関数を使用できます。
- ・ \$ZDATEH と \$ZDATE は、%DATE、%DI、%DO のエントリ・ポイントの呼び出しよりも高速に処理できます。

## 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$ZDATE 関数](#)
- ・ [\\$ZDATETIME 関数](#)
- ・ [\\$ZDATETIMEH 関数](#)
- ・ [\\$ZTIME 関数](#)
- ・ [\\$HOROLOG 特殊変数](#)
- ・ [\\$ZTIMESTAMP 特殊変数](#)
- ・ [%DATE ユーティリティ](#) (<https://docs.intersystems.com/priordocexcerpts> で利用可能な旧ドキュメントに記載)
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ZDATETIME (ObjectScript)

日付と時刻を検証し、これを内部形式から指定の表示形式に変換します。

### 構文

```
$ZDATETIME(hdatetime,dformat,tformat,precision,monthlist,yearopt,startwin,endwin,mindate,maxdate,erroppt,localeopt)
$ZDT(hdatetime,dformat,tformat,precision,monthlist,yearopt,startwin,endwin,mindate,maxdate,erroppt,localeopt)
```

### 引数

引数	説明
hdatetime	内部日付/時刻形式で指定した、日付と時刻の値。後述の “ <a href="#">hdatetime</a> ” を参照してください。
dformat	オプション — 返される日付値の形式を表す整数コード。後述の “ <a href="#">dformat</a> ” を参照してください。
tformat	オプション — 返される時刻値の形式を表す整数コード。後述の “ <a href="#">tformat</a> ” を参照してください。
precision	オプション — 返される時刻値の小数点以下の有効桁数 (小数秒) を表す整数。後述の “ <a href="#">precision</a> ” を参照してください。
monthlist	オプション — 一連の月名を表す文字列または変数の名前。この文字列は、区切り文字から始まり、12 個のエントリは、この区切り文字で分けられる必要があります。後述の “ <a href="#">monthlist</a> ” を参照してください。
yearopt	オプション — 年を 2 桁または 4 桁のどちらで表記するかを指定する整数コード。後述の “ <a href="#">yearopt</a> ” を参照してください。
startwin	オプション — 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値。後述の “ <a href="#">startwin</a> ” を参照してください。
endwin	オプション — 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最後を指定する数値。後述の “ <a href="#">endwin</a> ” を参照してください。
mindate	オプション — 有効な日付範囲の下限。\$HOROLOG 整数日付カウントの形式で指定し、0 は 1840 年 12 月 31 日を表します。正または負の整数として指定できます。後述の “ <a href="#">mindate</a> ” を参照してください。
maxdate	オプション — 有効な日付範囲の上限で、整数の \$HOROLOG 日付カウントの形式で指定します。後述の “ <a href="#">maxdate</a> ” を参照してください。
erroppt	オプション — hdatetimeが無効のときに返す式。この引数に値を指定すると、hdatetime 値が無効または範囲外の場合に生じるエラー・コードが抑制されます。エラー・メッセージを発行する代わりに、\$ZDATETIME は erroppt を返します。後述の “ <a href="#">erroppt</a> ” を参照してください。

引数	説明
localeopt	<p>オプション — dformat、tformat、monthlist、yearopt、mindate、maxdate の既定値、およびその他の日付と時刻の特性に対してどのロケールを使用するかを指定するブーリアン・フラグ。</p> <p>localeopt=0: 現在のロケール・プロパティ設定によって、これらの引数の既定値が決定されます。</p> <p>localeopt=1: ODBC 標準ロケールによって、これらの引数の既定値が決定されます。</p> <p>localeopt 未指定: dformat 値によって、これらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。それ以外の場合は、現在のロケール・プロパティ設定が使用されます。後述の “<a href="#">localeopt</a>” を参照してください。</p>

指定の引数値間で省略された引数は、プレースホルダのコンマで示されます。末尾のプレースホルダのコンマは必要ありませんが、あってもかまいません。省略された引数を示すコンマの間に空白があってもかまいません。

## 説明

\$ZDATETIME は、指定された日付と時刻を検証し、それらを \$HOROLOGY 内部形式あるいは \$ZTIMESTAMP 内部形式から、別の日付時刻の表示形式に変換します。返される値は、指定する引数によって異なります。

- ・ \$ZDATETIME(hdatetime) は、現在のロケール用の既定の表示形式で、日付と時刻を返します。
- ・ \$ZDATETIME (datetime, dformat, tformat, precision, monthlist, yearopt, startwin, endwin, mindate, maxdate) は、指定する他の引数によってさらに定義された、dformat と tformat で指定される表示形式で、日付と時刻を返します。有効な日付の範囲は、mindate 引数と maxdate 引数で制限できます。

## 引数

### datetime

内部形式値として指定された日付と時刻。InterSystems IRIS 内部形式は、任意の開始ポイント (1840 年 12 月 31 日) からの日数で日付を表示し、その日の秒数で時刻を表示します。datetime の値は、以下のいずれかの形式の文字列である必要があります。

- ・ [\\$HOROLOGY](#) : コンマで区切られた、2 つの符号なし整数。1 つ目は、日付 (日数) を指定する整数で、2 つ目は、時刻 (秒数) を指定する整数です。
- ・ [\\$ZTIMESTAMP](#) : コンマで区切られた、2 つの符号なしの数。1 つ目は、日付 (日数) を指定する整数で、2 つ目は、時刻 (秒と小数秒数) を指定する整数です。時刻値は、小数点以下有効桁数 9 桁 (小数秒) まで表示できます。

datetime は、文字列値、変数、式として指定できます。

datetime で日付部分の値のみが指定され、コンマが指定されない場合、日付のみが返されます。datetime で日付部分の値の後にコンマが指定され、時刻値が指定されない場合、時刻値として 00:00:00 が指定されます。

既定では、最も早い有効な datetime 日付は 0 です (1840 年 12 月 31 日)。DateMinimum プロパティの既定値は 0 であるため、既定では、日付は正の整数に制限されています。現在のロケールの DateMinimum プロパティが、過去の日付を表す負の整数以下の負の整数に設定されていれば、過去の日付を負の整数を使用して指定できます。

DateMinimum の有効な最小値は -672045 であり、これは 0001 年 1 月 1 日に相当します。InterSystems IRIS では、ISO 8601 標準に準拠してグレゴリオ暦が “西暦 1 年” まで遡って適用された先発グレゴリオ暦が使用されています。その理由の 1 つは、グレゴリオ暦が採用された時期は国ごとに異なるからです。例えば、欧州大陸諸国の多くでは 1582 年に採用された一方で、英国と米国では 1752 年に採用されました。このため、ご使用の地域でグレゴリオ暦が採用された時期より前の InterSystems IRIS の日付は、その当時にその地域で採用されていた暦に基づいて記録された歴史上の日付に対応していない可能性があります。1840 年より前の日付の詳細は、“[mindate](#)” 引数を参照してください。

hdatetime 値が無効または範囲外の場合に生じるエラーは、`errop` 引数で示されます。

## dformat

返される日付の形式。有効な値は以下のとおりです。

値	意味
1	MM/DD/[YY]YY (07/01/97 または 02/22/2018) – <b>アメリカの数値形式</b> 。日付区切り文字 (/ または .) は、現在のロケール設定から取得されます。
2	DD Mmm [YY]YY (01 Jul 97)
3	YYYY-MM-DD (2018-02-22) – ODBC 形式。既定では、この形式は、ユーザの現在のロケール設定に依存しません。したがって、日付と時刻を ODBC 標準交換形式で指定します。(ODBC の時刻形式の既定は、以下の tformat で説明しています。)日付と時刻に対するユーザの現在のロケール設定をこの形式で使用するには、localeopt を 0 に設定します。
4	DD/MM/[YY]YY (01/07/97 または 22/02/2018) – <b>ヨーロッパの数値形式</b> 。日付区切り文字 (/ または .) は、現在のロケール設定から取得されます。
5	Mmm [D]D, YYYY (Jul 1, 1997)
6	Mmm [D]D YYYY (Jul 1 1997)
7	Mmm DD [YY]YY (Jul 01 1997)
8	YYYYMMDD (19970701) – 数値形式
9	Mmmmm [D]D, YYYY (July 1, 1997)
10	W (2) – 曜日の番号。0 (日) から 6 (土) までの数値で指定します。\$SYSTEM.SQL.DAYOFWEEK() メソッドと比較してください。
11	Www (Tue) – 曜日の略名
12	Wwwwww (Tuesday) – 正式な曜日の名前
13	[D]D/[M]M/YYYY (1/7/2549 または 27/11/2549) – タイ語の日付形式。日と月は、ヨーロッパ形式と同じですが、先頭に 0 は付きません。年は仏教紀元 (BE) で、グレゴリオ暦の年に 543 年追加して計算されます。
14	nnn (354) – 年の日付
15	DD/MM/[YY]YY (01/07/97 や 22/02/2018) – ヨーロッパ形式 (dformat=4 と同じ)。日付区切り文字 (/ または .) は、現在のロケール設定から取得されます。
16	YYYYc[M]Mc[D]Dc – 日本の日付形式。年、月、および日の数は、他の日付形式と同じです。先頭のゼロは省略されます。“年”、“月”、および“日”の日本語の文字 (ここでは c として表示) は、年、月、および日の数の後に挿入されます。これらの文字は、年=\$CHAR(24180)、月=\$CHAR(26376)、および日=\$CHAR(26085) です。
17	YYYYc[M]Mc[D]Dc – 日本の日付形式。dformat 16 と同じです。ただし、“年” および “月” の日本語の文字の後に空白が挿入されます。
18	[D]D Mmmmm YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月名を示す)。日の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 Muharram 0001 AH。
19	[D]D [M]M YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月番号を示す)。日と月の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 1 0001 AH。



値	意味
20	[D]D Mmmmm YYYY – Observed Hijri (イスラム) 日付形式 (完全な月名を示す)。既定で Tabular Hijri (dformat 18) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。
21	[D]D [M]M YYYY – Observed Hijri (イスラム) 日付形式 (完全な月番号を示す)。既定で Tabular Hijri (dformat 19) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。
-1	<a href="#">ユーザのロケール</a> の fmt.DateFormat から有効な dformat 値を取得します。fmt は、現在のプロセスに関連付けられた <code>##class(%SYS.NLS.Format)</code> のインスタンスです。dformat を指定しなければ、これが既定の振る舞いになります。詳細は、“日付と時刻の既定値のカスタマイズ”を参照してください。
-2	<p>\$ZDATETIME は、プラットフォーム固有の日付/時刻の起点からの経過秒数を表す整数を返します。これは、ISO C プログラミング言語標準の定義に従って time() ライブラリ関数から返される値です。例えば、POSIX 準拠システムでは、この値は協定世界時 1970 年 1 月 1 日 00:00:00 (January 1, 1970 00:00:00 UTC) を起点とする経過秒数です。入力値では秒の小数部を指定できますが、無視されます。</p> <p>(現在、この日付変換の際に、“ローカル時刻調整の境界日”での時刻変換の変則性が発生する可能性があります。この変則性は、tformat 値が 5、6、7、または 8 の場合に該当するものです)。</p> <p>この秒の整数値を PosixTime 値に変換するには、<a href="#">以下</a>のように UnixTimeToLogical() メソッドを使用します。</p> <p>次に示すプラットフォーム固有形式がサポートされています。32 ビット Linux では 32 ビット符号付き整数、64 ビット Linux では 64 ビット符号付き整数、Windows では 64 ビット符号なし整数です。</p> <p>tformat、precision、monthlist、yearopt、startwin、および endwin の各引数は無視されます。</p>
-3	<p>\$ZDATETIME は、<a href="#">\$HOROLOG</a> 内部形式で指定された datetime 値を取り、この値をローカル時刻から UTC 時刻に変換し、その結果をこの内部形式で返します。tformat、monthlist、yearopt、startwin、および endwin の各引数は無視されます。\$ZDATETIMEH はこの逆の処理を行います (現在、この日付変更の際に、時刻変換の変則性が発生する可能性があります。この変則性は tformat 値が 5、6、7、または 8 の場合に該当するものです。これにより、1970 年より前の日付、2038 年より後の日付、およびサマータイムの開始日と終了日などのローカル時刻調整の境界日が影響を受ける可能性があります)。</p>

以下は、この指定の説明です。

構文	意味
YYYY	YYYY は 4 桁の年です。[YY]YY は、hdatetime がアクティブ・ウィンドウ内で 2 桁の場合は 2 桁、それ以外は 4 桁です。
MM	2 桁の月 : 01 から 12。[M]M は、1 月から 9 月では先頭のゼロが省略されることを示します。
DD	2 桁の日 : 01 から 31。[D]D は、1 日から 9 日では先頭のゼロが省略されることを示します。
Mmm	現在のロケールの MonthAbbr プロパティから取得した月の省略形。別の月の省略形 (あるいは長さに制限のない名前) は、\$ZDATETIME に monthlist 引数として指定したオプションのリストから取得することができます。以下は MonthAbbr の既定値です。“Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”
Mmmmm	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。以下はその既定値です。“January February March ...November December”
W	曜日を表す 0 から 6 の数字。(例) Sunday は 0、Monday は 1、Tuesday は 2。

構文	意味
Www	現在のロケールの WeekdayAbbr プロパティが指定する曜日の省略形。以下はその既定値です。 “Sun Mon Tue Wed Thu Fri Sat”
Wwwwww	現在のロケールの WeekdayName プロパティが指定する曜日の正式な名前。以下はその既定値です。 “Sunday Monday Tuesday Wednesday Thursday Friday Saturday”
nnn	指定された年の日付。常に 3 桁で示され、必要に応じて先頭にゼロが付きます。値は、001 ~ 365 (うるう年の場合は 366) です。

### dformat の既定値

dformat を省略するか、-1 に設定した場合、dformat の既定値は localept 引数と NLS の **DateFormat** プロパティに依存します。

- ・ localept=1 の場合、dformat の既定値は ODBC 形式です。tformat、monthlist、yearopt、mindate、および maxdate 引数も、ODBC 形式に設定されます。これは、dformat=3 を設定した場合と同じです。
- ・ localept=0 または未指定の場合、dformat の既定値は NLS の **DateFormat** プロパティから取得されます。  
**DateFormat=3** の場合、dformat の既定値は ODBC 形式です。ただし、DateFormat=3 は tformat、monthlist、yearopt、mindate、および maxdate 引数の既定値に影響しません。これらは、NLS の現在のロケール定義で指定されます。

ユーザのロケールの既定の日付形式を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
```

ヨーロッパ日付形式 (dformat=4、DD/MM/YYYY の順) は、ヨーロッパの多くの言語 (すべてではない) で既定です。これには、“/” を **DateSeparator** 文字として使用するイギリス英語、フランス語、ドイツ語、イタリア語、スペイン語、ポルトガル語、および “.” を **DateSeparator** 文字として使用するチェコ語 (csyw)、ロシア語 (rusw)、スロバキア語 (skyw)、スロベニア語 (svnw)、ウクライナ語 (ukrw) が含まれます。サポートされるロケールの既定の日付形式の詳細は、“[日付](#)”を参照してください。

### dformat の設定

dformat が 3 (ODBC 日付形式) の場合は、monthlist、yearopt、mindate、および maxdate 引数の既定値にも ODBC 形式の既定値が使用されます。現在のロケールの既定値は無視されます。

dformat が -1、1、4、13、または 15 (数値の日付形式) の場合は、現在のロケールの **DateSeparator** プロパティの値が \$ZDATETIME の月、日、年の区切り文字として使用されます。dformat が 3 の場合は、ODBC の日付区切り文字 (“-”) が使用されます。dformat がそれ以外の値の場合は、空白が日付区切り文字として使用されます。英語での **DateSeparator** の既定値は “/” で、この区切り文字はすべてのドキュメントで使用されています。

dformat が 11 または 12 (曜日名) で、localept=0 または未指定の場合、曜日名の値は現在のロケールのプロパティから得られます。localept=1 の場合、曜日名は ODBC ロケールから得られます。ユーザのロケールの既定の曜日名および曜日名の省略形を決定するには、以下の NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayName"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("WeekdayAbbr"),!
```

dformat が -2 (Unix® 時刻:1970-01-01 00:00:00 からの UTC の経過秒数) である場合、UnixTimeToLogical() メソッドを使用することで、この値をエンコードされた PosixTime 値に変換できます。以下の例では、dformat -2 を使用して、ローカル時刻を UTC (グリニッジ) 時刻の経過秒数に変換します。その後、UnixTimeToLogical() を使用して、この秒数をエンコードされた %PosixTime 値に変換します。LogicalToOdbc() を使用して、この PosixTime 値を UTC 時刻の ODBC

形式の日付/時刻に変換し直します。また、LogicalToUnixTime() を使用して PosixTime を経過秒数に変換し直します。その後、dformat -2 を指定して \$ZDATETIMEH を使用し、この UTC の経過秒数をローカル日付/時刻に変換します。

## ObjectScript

```
WRITE "local datetime: ", $ZDATETIME($HOROLOG,3),!  
SET secs=$ZDATETIME($HOROLOG,-2)  
WRITE "UTC seconds since 1970: ",secs,!  
SET posix=##class(%PosixTime).UnixTimeToLogical(secs)  
WRITE "PosixTime encoded value: ",posix,!  
SET datetime=##class(%PosixTime).LogicalToOdbc(posix)  
WRITE "UTC datetime: ",datetime,!  
SET secs2=##class(%PosixTime).LogicalToUnixTime(posix)  
WRITE "UTC seconds since 1970: ",secs2,!  
SET htime=$ZDATETIMEH(secs2,-2)  
WRITE "local datetime: ", $ZDATETIME(htime,3)
```

Posix 時刻では小数点以下 6 桁の精度で秒の小数部がカウントされますが、Unix® 時刻は整数秒数になります。

dformat が 16 または 17 (日本の日付形式) の場合、返される日付形式はロケールの設定に依存しません。日本の日付形式は、任意の InterSystems IRIS インスタンスから取得できます。

dformat が 18、19、20、または 21 (イスラム暦の日付形式) で、localeopt が未指定の場合、引数の既定値は現在のロケールの既定値ではなく、イスラム暦の既定値になります。monthlist 引数の既定値は、ラテン文字で書き直されたアラビア語の月名になります。tformat、yearopt、mindate、および maxdate 引数の既定値は、ODBC の既定値になります。日付区切り文字の既定値は、ODBC の既定値や現在のロケールの **DateSeparator** プロパティ値ではなく、イスラム暦の既定値 (空白) になります。localeopt=0 の場合は、現在のロケール・プロパティの既定値がこれらの引数に使用されます。localeopt=1 の場合は、ODBC の既定値がこれらの引数に使用されます。

## tformat

時刻の値を表記する形式を指定する数値です。サポートされる値は以下のとおりです。

値	意味
-1	現在のロケールの TimeFormat プロパティから有効な tformat 値を取得します。既定値は 1 です。dformat が 3 以外であれば、tformat を指定しない場合はこれが既定の動作になります。
1	時刻を "hh:mm:ss" (24 時制) の形式で表します。
2	時刻を "hh:mm" (24 時間) の形式で表します。
3	時刻を "hh:mm:ss[AM/PM]" (12 時間) の形式で表します。
4	時刻を "hh:mm[AM/PM]" (12 時間) の形式で表します。
5	時刻を "hh:mm:ss+/-hh:mm" (24 時間) の形式で表します。時刻を現地時間で表します。プラス (+) またはマイナス (-) の接尾語は、協定世界時 (UTC) からの現地時間のシステム定義オフセットを示します。マイナス符号 (-hh:mm) は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より早い (西向きである) ことを示します。プラス符号 (+hh:mm) は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より遅い (東向きである) ことを示します。詳細は後述します。
6	時刻を "hh:mm+/-hh:mm" (24 時間) の形式で表します。時刻を現地時間で表します。プラス (+) またはマイナス (-) の接尾語は、協定世界時 (UTC) からの現地時間のシステム定義オフセットを示します。マイナス符号 (-hh:mm) は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より早い (西向きである) ことを示します。プラス符号 (+hh:mm) は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より遅い (東向きである) ことを示します。詳細は後述します。
7	時刻を "hh:mm:ssZ" (24 時間) の形式で表します。"Z" 接尾語は、時刻が現地時間ではなく、協定世界時 (UTC) で表示されていることを示します。

値	意味
8	時刻を“hh:mmZ”(24 時間)で表します。“Z” 接尾語は、時刻が現地時間ではなく、協定世界時 (UTC) で表示されていることを示します。

tformat を省略するか、-1 に設定した場合、tformat の既定値は localeopt 引数と NLS の **TimeFormat** プロパティに依存します。

dformat の値が 3、18、19、20、21 以外の場合、すべての時刻形式は既定で現在のロケール定義の **TimeSeparator** および **DecimalSeparator** プロパティ値になります。dformat=3 (ODBC の日付形式) および dformat=18、19、20、または 21 (イスラム暦の日付形式) の場合は、現在のロケールのプロパティ値に関係なく、時刻区切り文字はコロン (:) になり、**DecimalSeparator** はピリオド (.) になります。これらの既定は、localeopt を設定することにより、オーバーライドできます。

ユーザのロケールの既定の時刻プロパティを決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

tformat 値が (現地時間を返す) 1 から 4 までの場合、日付と時刻はスペースで区切られます。tformat 値が 5 から 8 までの場合、日付と時刻は文字 “T” で区切られます。

### 12 時間形式 (tformat 3 および 4)

12 時間形式では、午前と午後を時間接尾語で表します。ここでは、AM と PM で表示されています。ユーザのロケールの既定の時間接尾語を決定するには、以下のように GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
```

dformat の値が 3、18、19、20、21 以外の場合、**AM** および **PM** プロパティは既定で現在のロケール定義になります。dformat=3 (ODBC の日付形式) および dformat=18、19、20、または 21 (イスラム暦の日付形式) の場合は、現在のロケールのプロパティ値に関係なく、時間接尾語は常に “AM” および “PM” になります。**AM** および **PM** プロパティは、日本のロケール jpww を除くすべてのロケールで “AM” と “PM” になります。

日本のロケール (jpnw, jpww, zds, zdtw, zduw)、ポルトガル (ptbw)、ロシア (rusw)、およびウクライナ (ukrw) を除くすべてのロケールで、既定では深夜と正午は “MIDNIGHT” および “NOON” で表されます。

ただし、dformat=3 の場合、\$ZDATETIME では、ロケールの既定の設定に関係なく、常に ODBC 標準値を使用します。

### ローカル時刻 (tformat 5、6、7、および 8)

tformat を 5 または 6 に設定すると、hdatetime 入力値は現地日時であると想定され、現地日時として表示されます。hdatetime が現在の現地日時 (\$HOROLOGY) である場合は、\$ZTIMEZONE を変更すると、現在のプロセスについてこの現在の日時が変更されます。

オフセット接尾語は、グリニッジ子午線からの正または負のオフセット (時間および分) として、ローカル時刻調整設定を指定します。このローカル時刻調整は、必ずしもタイム・ゾーン・オフセットではありません。例えば、米国東部のタイム・ゾーンはグリニッジ標準時の 5 時間後 (-5:00) ですが、ローカル時刻調整 (サマータイム) によって、このタイム・ゾーンの時刻が 1 時間オフセットされて、-04:00 となります。\$ZTIMEZONE を設定すると、\$ZDATETIME(\$HOROLOGY, 1, 5) によって返される現在のプロセスの日時が変更されますが、システムのローカル時刻調整設定は変更されません。

注釈 tformat が 5 または 6 の場合は、UTC 時刻からのローカル時刻調整オフセットが返されます。これは、ローカル・タイム・ゾーン・オフセットでも、ローカル時刻と英国グリニッジのローカル時刻との比較でもありません。グリニッジ標準時 (GMT) という用語は誤解を招くことがあり、英国グリニッジのローカル時刻は、冬期は UTC と一致していますが、夏期は UTC から 1 時間ずれています。これは、英国夏時間と呼ばれるローカル時刻の変更が適用されるためです。

以下の例では、オペレーティング・システムのローカル時刻調整設定によって、および現在のプロセスのタイム・ゾーンの変更によって、tformat の 5 という値がどのような影響を受けるのかを示しています (この例では、ローカル時刻調整の境界がプログラム実行時に発生するかどうかを確認します)。

## ObjectScript

```
LocalDatetimeOffset
SET dst=$SYSTEM.Util.IsDST()
SET local=$ZDATETIME($HOROLOG,1,5)
WRITE local," is the local date/time and offset",!!
SET off=$PIECE(local,"+",2)
IF off="" {SET off=$PIECE(local,"-",2)
    WRITE "-",off," is local offset",!}
ELSE {WRITE "+",off," is local offset",!}
SET tz=$ZTIMEZONE
WRITE tz/60," is the local timezone offset, in hours",!!
IF dst=1 {WRITE " DST in effect, ",off," offset is not ",tz/60," time zone offset",!}
    ELSEIF dst=0 {WRITE " DST not in effect, offset ",off,"=timezone ",tz/60,!}
    ELSE {WRITE " DST setting cannot be determined",!}
ChangeTimeZoneForCurrentProcess
SET $ZTIMEZONE=tz+180
WRITE !,"changed the process time zone westward 3 hours",!
WRITE $ZDATETIME($HOROLOG,1,5)," is new local date/time and offset",!
WRITE "note that time has changed, but offset has not changed"
SET $ZTIMEZONE=tz
ConfirmNoDSTBoundary
SET dst2=$SYSTEM.Util.IsDST()
GOTO:dst'=dst2 LocalDatetimeOffset
```

tformat を 7 または 8 に設定すると、hdatetime 入力値は現地日付および時刻であると想定されます。時刻は対応する UTC 時間に変更されます (ローカル・タイムゾーンの設定を使用して計算)。返される日付も、この UTC 時間値に一致するように変更されます (必要な場合)。そのため、この日付は、現地日付とは異なる場合があります。



注釈 dformat 値が -2 および -3、tformat 値が 7 および 8、かつ tformat 値が 5 および 6 のときに生成される UTC オフセットを含む変換には、以下のプラットフォーム依存の変則性があります。

- ローカル時刻から UTC 時刻への変換は、ローカル時刻調整の境界の動作に依存します。この動作は、オペレーティング・システム・プラットフォームごとに異なる可能性があります。

ローカル時計を進めたとき (サマータイム開始日の “Spring ahead”) は、進めた 1 時間、ローカル時刻が失われます。この “失われた” 1 時間は、不正なローカル時刻値です。InterSystems IRIS の \$HOROLOGY は、不正なローカル時刻値を決して返してはいけません。ただし、ユーザがこの不正なローカル時刻値を手動で入力した場合は (例えば \$HOROLOGY を設定することによって)、\$ZDATETIME の変換結果は不確定であり、プラットフォームによって大きく左右されます。

ローカル時計を遅らせたとき (サマータイム終了日の “Fall back”) は、遅らせた 1 時間、ローカル時刻が繰り返されます。この 2 時間以内に時刻を変換すると、InterSystems IRIS では、このローカル時刻の 1 時間目に対して時刻変換処理を行うのか、2 時間目に対して行うのかを判断できません。\$ZDATETIME では、プラットフォーム固有のランタイム・ライブラリで使用されている前提を採用します。したがって、この時間枠の範囲では、オペレーティング・システム・プラットフォームによって時刻変換の結果が異なる可能性があります。

- ローカル時刻と UTC 時刻の間の変換では、指定された年と場所に対して適用されているローカル時刻調整ルールが使用される必要があります。これらのルールは現地法によって規定されており、過去に変更された可能性があり、将来に変更される可能性もあるため、\$ZDATETIME の変換は、オペレーティング・システム・プラットフォームによってエンコードされているこれらのルールの完全性と正確性に依存します。将来の年の予測では、現在のルールを使用する必要があり、これらのルールは変更される可能性があります。
- ローカル時刻と UTC 時刻の間の変換は、オペレーティング・システム・プラットフォームでサポートされている日付範囲に依存します。

日付がプラットフォームでサポートされている最も早い日付より前の場合、InterSystems IRIS は、1902-01-01 に対する標準時オフセットを使用します (この日付がプラットフォームでサポートされている場合)。日付 1902-01-01 がプラットフォームでサポートされていない場合、InterSystems IRIS は、1970-01-01 に対する標準時オフセットを使用します。現地時刻調整オフセット (サマータイムなど) は無視されます。

日付がプラットフォームでサポートされている最も遅い日付より後の場合、InterSystems IRIS は、2010-01-01 から 2037-12-31 の範囲内で対応する日付を計算し、その対応する日付に対する標準時オフセットを使用します。このアルゴリズムは、2100-02-28 までの日付に対して、正確な時刻オフセットを実現します (今後、日付/時刻を管理する法律の変更がない場合)。

注釈 \$ZDATETIME には、hdatetime 入力値が UTC 時間か現地時間かを判別する方法がありません。したがって、\$ZTIMESTAMP 値などの、もともと UTC である hdatetime では、tformat 値 5、6、7、または 8 を使用しないでください。時間を現地時間に再変換する演算で tformat 7 または 8 変換からの出力を使用する場合は、現地から UTC への変換時に日付が変更されている可能性があることに注意してください。

## precision

時刻を表すのに使用される、秒の小数部の精度を示す小数点以下の桁数を指定する整数値です。つまり、precision の値として 3 を入力すると、\$ZDATETIME は秒数を小数点以下 3 桁まで表示します。precision の値として 9 を入力すると、\$ZDATETIME は秒数を小数点以下 9 桁まで表示します。返す小数桁数をこの引数で指定します。精度を示す実際の有効桁数は hdatetime ソースによって決まります。例えば、\$HOROLOGY は小数秒を返しませんが、\$ZTIMESTAMP と \$NOW() は小数秒を返します。

サポートされている値は、以下のとおりです。

現在のロケールの **TimePrecision** プロパティから、precision 値を取得します。既定は 0 です。precision を指定しなければ、これが既定の振る舞いになります。

ゼロ (0) 以上の n の値は、時刻を小数点以下の n 桁まで表すことを示します。

precision は hdatetime 形式に小数点以下の値 (\$ZTIMESTAMP 形式) を含めることができ、選択した tformat オプションに秒が含まれている場合にのみ適用されます。末尾のゼロは、抑制されません。指定した有効桁数がシステムで利用できる有効桁数を超えている場合、有効桁数の余分な桁は、末尾のゼロとして返されます。9 の precision が指定された以下の例のように、Normalize() メソッドを使用することで、余分な末尾のゼロを抑制できます。

## ObjectScript

```
WRITE $ZDATETIME($ZTIMESTAMP,3,,9),!  
WRITE ##class(%TimeStamp).Normalize($ZDATETIME($ZTIMESTAMP,3,,9))
```

ユーザのロケールの既定の時刻精度を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimePrecision")
```

ロケールの TimePrecision を任意の桁数 (最大 15 桁) に設定できます。

## monthlist

区切り文字で区切られた月名または月の省略名の文字列に解決される式です。monthlist にある名前は、現在のロケールの **MonthAbbr** プロパティに指定されている既定の月の省略形の値、または **MonthName** に指定されている月名の値を置き換えます。

monthlist は、dformat が 2、5、6、7、9、18、または 20 の場合にのみ有効です。dformat がそれ以外の値の場合、\$ZDATETIME は monthlist を無視します。

monthlist 文字列の形式は、以下のとおりです。

- 文字列の最初の文字は、区切り文字 (通常はスペース) です。monthlist 内の最初の月名の前と、各月名の間に、同じ区切り文字を置く必要があります。1 文字からなる任意の区切り文字を指定できます。この区切り文字は、返される日付値の月、日、年の間に表示されます。このため、スペースが通常推奨される区切り文字となります。
- 月名の文字列は、1 月 から 12 月に対応する 12 個の区切り値を含む必要があります。12 より多い、または 12 より少ない月名を指定することは可能ですが、hdatetime の月に該当する月名がない場合、<ILLEGAL VALUE> エラーが発生します。

monthlist を省略するか、monthlist の値として -1 を指定した場合は、現在のロケールの **MonthAbbr** または **MonthName** プロパティで定義されている月名のリストが \$ZDATETIME で使用されます (localeopt=1、または monthlist の既定値が ODBC の月リスト (英語) である場合を除く)。localeopt が未指定で、dformat が 18 または 20 (イスラム暦の日付形式) の場合、monthlist の既定値はイスラム暦の月リスト (ラテン文字で表されたアラビア語の名前) になり、**MonthAbbr** および **MonthName** プロパティ値は無視されます。

ユーザのロケールの既定の月名および月名の省略形を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```

## yearopt

dformat 値に 0、1、2、4、7、または 15 を使用して、年を 2 桁で表示する時間ウィンドウを指定する数値コードです。yearopt の値は以下のとおりです。



値	意味
-1	現在のロケールの YearOption プロパティから有効な yearopt 値を取得します。この既定値は 0 です。yearopt を指定しない場合、これが既定の振る舞いになります。
0	(%DATE ユーティリティにより構築される) プロセス特有のスライディング・ウィンドウが有効になっていない限り、20 世紀 (1900 年から 1999 年まで) の日付は 2 桁で表示します。スライディング・ウィンドウが有効な場合は、指定範囲内の日付のみ 2 桁の年を表示します。1900 年代以外もしくはプロセス特有のスライディング・ウィンドウ以外の日付は、すべて 4 桁で表示します。
1	20 世紀の日付は 2 桁の年、それ以外は 4 桁で表します。
2	すべての日付を 2 桁の年で表します。
3	startwin と (オプションで) endwin に定義した一時的なスライディング・ウィンドウ範囲内に収まる日付を、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt=3 の場合、startwin と endwin は、\$HOROLOG 形式の絶対日付です。
4	すべての日付を 4 桁の年で表します。
5	startwin と (オプションで) endwin に定義したスライディング・ウィンドウ範囲内に収まる日付をすべて、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt=5 の場合、startwin と endwin は相対年になります。
6	2 桁の年で現在の世紀にあるすべての日付を表します。それ以外は 4 桁の年で表します。

ユーザのロケールの既定の年のオプションを決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("YearOption")
```

yearopt を省略するか、yearopt の値として -1 を指定した場合は、現在のロケールの **YearOption** プロパティが \$ZDATETIME で使用されます (localeopt=1、または yearopt の既定値が ODBC の年オプションである場合を除く)。localeopt=0 または未指定で、dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、yearopt の既定値は ODBC の年オプション (4 桁の年) になります。イスラム暦の日付では、**YearOption** プロパティの値は無視されます。

## startwin

日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値です。yearopt の値に 3 または 5 を使用する場合は、startwin を指定する必要があります。startwin は、他の yearopt 値では無効になります。

yearopt=3 のとき、startwin はスライディング・ウィンドウの最初の日付を示す \$HOROLOG 日付形式の絶対日付です。

yearopt=5 のとき、startwin は、現在の年より前の年数で表される、スライディング・ウィンドウの最初の年を示す数値です。

## endwin

日付を 2 桁の年で表すスライディング・ウィンドウの最後を指定する数値です。yearopt が 3 あるいは 5 のとき、endwin をオプションで指定する必要があります。endwin は、その他の yearopt 値では無効になります。

yearopt=3 のとき、endwin はスライディング・ウィンドウの最後の日付を示す \$HOROLOG 日付形式の絶対日付です。

yearopt=5 のとき、endwin は、現在の年から後の年数を示すスライディング・ウィンドウの最後の年を示す数値です。

yearopt=5 のとき、スライディング・ウィンドウは常に startwin で指定する年の最初の日付 (1 月 1 日) で開始し、endwin で指定する年の最後の日付 (12 月 31 日) あるいは最後と暗示される年の最後の日付 (endwin が省略されているとき) で終了します。

endwin を省略した場合または -1 を指定した場合、スライディング・ウィンドウの有効期限は 100 年間です。endwin 値を -1 とすると特殊なケースになります。endwin の値が -1 よりも大きい場合または小さい場合に erropt が返される場合で

も、-1 の場合は必ず日付値が返されます。このため、100 年間のウィンドウを指定する場合は endwin を省略し、その他の場合でも endwin には負の値を指定しないようにすることをお勧めします。

startwin と endwin の両方を指定する場合、指定するスライディング・ウィンドウの年数は 100 年以内に収める必要があります。

## mindate

有効な日付範囲(両端を含む)の下限を指定する式です。\$HOROLOG 整数日付カウント(例えば、2/22/2018 は 64701 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付("64701,43200" など)の時刻部分は含めることも省略することもできますが、解析されるのは mindate の日付部分のみです。mindate より前の hdatetime 値を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

mindate に指定できる値は以下のとおりです。

- ・ 正整数：mindate には、通常、1840 年 12 月 31 日より後の最も早い有効日付を設定するための正の整数を指定します。例えば、mindate に 21550 を指定すると、1900 年 1 月 1 日が最も早い有効日付として設定されます。有効な最大値は 2980013 です(9999 年 12 月 31 日)。
- ・ 0：1840 年 12 月 31 日を最も早い日付として指定します。これは **DateMinimum** プロパティの既定値です。
- ・ -2 以下の負の整数：1840 年 12 月 31 日から遡ってカウントする最も早い日付を指定します。例えば、mindate に -14974 を指定すると、1800 年 1 月 1 日が最も早い有効日付として設定されます。負の mindate 値が有効となるのは、現在のロケールの **DateMinimum** プロパティがその値以下の負の整数に設定されている場合のみです。有効な最小値は -672045 です。-672045 より前の mindate は、<ILLEGAL VALUE> エラーを生成します。
- ・ mindate を省略するか、-1 を指定した場合は、現在のロケールの **DateMinimum** プロパティ値が既定値になります(localeopt=1 または mindate の既定値が 0 の場合を除く)。localeopt が未指定で dformat=3 の場合、mindate の既定値は 0 になります。localeopt が未指定で dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、mindate の既定値は 0 になります。

**DateMinimum** プロパティは、以下のように取得および設定できます。

## ObjectScript

```
SET min=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Permit18thCenturyDates
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
SET newmin=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "set DateMinimum value is ",newmin,!
RestrictTo19thCenturyDates
WRITE $ZDATETIME(-13000,1,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

mindate を指定する際は、maxdate を指定しても指定しなくてもかまいません。maxdate より大きい mindate を指定すると、<ILLEGAL VALUE> エラーが発生します。

## ODBC 日付形式 (dformat 3)

**DateMinimum** プロパティの適用は、localeopt 設定によって管理されます。localeopt=1 (dformat=3 の場合の既定) の場合、現在のロケール設定に関係なく、日付の最小値は 0 になります。したがって、ODBC 形式 (dformat=3) では、以下を使用して 1840 年 12 月 31 日より前の日付を指定できます。

- ・ 指定した日付よりも前の mindate を指定します。

## ObjectScript

```
WRITE $ZDATETIME(-30,3,,,,,-365)
```

- ・ 指定した日付よりも前の **DateMinimum** プロパティ値を指定し、localeopt=0 を設定します。

## ObjectScript

```
DO ##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-365)
WRITE $ZDATETIME(-30,3,,,,,,,,,0)
```

## maxdate

有効な日付範囲(両端を含む)の上限を指定する式です。\$HOROLOG 整数日付カウント(例えば、1/1/2100 は 94599 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付("94599,43200" など)の時刻部分は含めることも省略することもできますが、解析されるのは maxdate の日付部分のみです。

maxdate を省略するか、または -1 を指定すると、現在のロケールの **DateMaximum** プロパティから日付の上限が取得されます。既定値は、\$HOROLOG の日付部分で許容される最大値である、2980013 (西暦 9999 年 12 月 31 日に対応)です。ただし、**DateMaximum** プロパティの適用は localeopt の設定によって制御されます。localeopt=1 (dformat=3 の場合の既定値)の場合、現在のロケール設定に関係なく、日付の既定最大値は ODBC の値 (2980013) になります。イスラム暦の日付形式でも ODBC の既定値が使用されます。タイ日付形式 (dformat=13) の最大日付は \$HOROLOG 2781687 で、これは BE 9999 年 12 月 31 日に対応します。

maxdate より大きい hdatetime 日付を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

2980013 より大きい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

maxdate を指定する際は、mindate を指定しても指定しなくてもかまいません。mindate より小さい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

## erript

この引数に値を指定すると、hdatetime 値が無効または範囲外の場合に生じるエラーが抑制されます。\$ZDATETIME 関数は、<ILLEGAL VALUE> または <VALUE OUT OF RANGE> エラーを生成する代わりに erript 値を返します。

- ・ 検証: InterSystems IRIS は、hdatetime に対して [キャノニック数値変換](#) を実行します。hdatetime の日付と時刻の部分は別々に解析されます。日付/時刻区切り文字として検出された最初のコンマを解析します。追加のコンマは、数値以外の文字として処理されます。

hdatetime 文字列の各部分の解析は、数値以外の最初の文字で停止します。したがって、64687AD,1234SECS などの hdatetime は、64687,1234 と同じです。数値以外の日付または時刻部分 (NULL 文字列を含む) は 0 に評価されます。したがって、hdatetime が空の文字列の場合は、\$HOROLOG の最初の日付 (12/31/1840) が返されます。

ただし、日付部分の値が整数に評価されない (ゼロ以外的小数を含む) 場合、<ILLEGAL VALUE> エラーが発生します。

- ・ 範囲: hdatetime の日付部分は mindate から maxdate までの範囲内の整数に評価される必要があります。既定では、2980013 を超える日付値または 0 未満の日付値は <VALUE OUT OF RANGE> エラーを生成します。mindate を負の数値に設定すると、1840 年 12 月 31 日より前の有効な日付の範囲を拡張できます。ただし、dformat 18、19、20、または 21 (ヒジュラ・イスラム暦) の日付の場合、-445031 より前の日付は、mindate がより早い日付に設定されていても、<ILLEGAL VALUE> エラーを生成します。

hdatetime の時刻部分に 86399 より大きい値を指定すると、<ILLEGAL VALUE> エラーが生成されます。負の hdatetime 時刻値を指定すると、<ILLEGAL VALUE> エラーが生成されます。

erript 引数は、hdatetime の無効な値または範囲外の値が原因で生成されるエラーのみを抑制します。他の引数の無効な値、あるいは範囲外の値が原因で発生したエラーについては、erript が指定されているかどうかに関係なく、常にエラーが生成されます。例えば、\$ZDATETIME で endwin が startwin より前になる値のスライディング・ウィンドウを指定すると、常に <ILLEGAL VALUE> エラーが発生します。同様に、maxdate が mindate より小さい場合、<ILLEGAL VALUE> エラーが発生します。

## localeopt

このブーリアン引数には、ロケールで指定される引数 dformat、tformat、monthlist、yearopt、mindate、および maxdate の既定値のソースとして、ユーザの現在のロケール定義または ODBC のロケール定義が指定されます。

- ・ localeopt=0 の場合は、これらすべての引数で現在のロケール定義の既定値が使用されます。
- ・ localeopt=1 の場合は、これらすべての引数で ODBC の既定値が使用されます。
- ・ localeopt が未指定の場合は、dformat 引数によってこれらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。dformat が 18、19、20、または 21 の場合は、現在のロケール定義に関係なく、イスラム暦の日付および時刻形式の既定値が使用されます。dformat がそれ以外の値の場合は、現在のロケール定義の既定値が使用されます。詳細は、“[dformat](#)” の説明を参照してください。

ODBC 標準ロケールは変更できません。これは、異なる各国語サポート (NLS) の選択を行った InterSystems IRIS プロセス間で移植できる、日付文字列および時刻文字列をフォーマットするために使用されます。ODBC ロケールの日付と時刻の定義は、以下のとおりです。

- ・ 日付形式の既定は 3 です。したがって、dformat が未定義または -1 の場合は、日付形式 3 が使用されます。
- ・ 日付の区切り文字の既定は “/” です。ただし、日付形式の既定は 3 で、これは日付の区切り文字として “-” を常に使用します。
- ・ 年のオプションの既定は 4 桁です。
- ・ 日付の最小値は 0 で、最大値は 2980013 (\$HOROLOG の日数カウント) です。
- ・ 英語の月名、月の省略形、曜日名、曜日の省略形、および “Noon” と “Midnight” の語が使用されます。
- ・ 時刻形式は既定で 1 になります。時刻区切り文字は “.” です。時刻精度は 0 (小数秒なし) です。午前と午後の指定子は、“AM” と “PM” です。

## 例

以下の例は、現在の現地日付と現地時刻を表示します。また、ロケールの既定の日付および時刻形式を取ります。

### ObjectScript

```
WRITE $ZDATETIME($HOROLOG)
```

以下の例は、現在の日付と時刻を表示します。\$ZTIMESTAMP には、現在の日付値と時刻値が協定世界時 (UTC) 日付/時刻として含まれます。dformat 引数は ODBC 日付形式を指定し、tformat 引数は 24 時間制を指定し、precision 引数は 6 桁の小数秒の有効桁数を指定します。

### ObjectScript

```
WRITE $ZDATETIME($ZTIMESTAMP,3,1,6)
```

2018/11/25 18:45:16.960000 のようにフォーマットされた、現在のタイムスタンプ日付と時刻を返します。

以下の例は、現地時間を UTC 時間に変換する方法と、その変換結果によりどのように日付が変更されるかを示します。ほとんどのタイム・ゾーンにおいて、以下の \$ZDATETIME 演算のいずれかで、時間変換により日付変更が生じます。

### ObjectScript

```
SET local = $ZDATETIME("60219,82824",3,1)
SET utcwest = $ZDATETIME("60219,82824",3,7)
SET utceast = $ZDATETIME("60219,00024",3,7)
WRITE !,local,! ,utcwest,! ,utceast
```

## \$ZDATETIME で無効な値

以下の状況では、<FUNCTION> エラーが発生します。

- ・ 無効な dformat コード (-3 未満または 17 より大きい整数値、0、整数以外の値) を指定した場合。
- ・ tformat に無効な値 (-1 未満または 10 より大きい整数値、0、整数以外の値) を指定した場合。
- ・ yearopt が 3 あるいは 5 のときに、startwin 値を指定しない場合。

以下の状況では、<ILLEGAL VALUE> エラーが発生します。

- ・ 日付または時刻に無効な値を指定して、erropt 値を提供しない場合。
- ・ 指定された月数が monthlist の値よりも大きい場合。
- ・ maxdate が mindate よりも小さい場合。
- ・ endwin が startwin よりも小さい場合。
- ・ startwin と endwin で、期間が 100 年より長い一時的なスライディング・ウィンドウが指定された場合。

以下の状況では、<VALUE OUT OF RANGE> エラーが発生します。

- ・ maxdate と mindate に想定される値によって定義された範囲外にある、範囲内にあれば有効であった日付を指定し、erropt 値を指定しない場合。

## 日付と時刻の既定値のカスタマイズ

InterSystems IRIS の起動時、既定の日付および時刻の形式は、アメリカの日付および時刻形式 (例えば MM/DD/[YY]YY) に初期化されています。この形式およびその他の既定の形式を現在のロケール用に設定するには、`SET ^SYS("NLS","Config","LocaleFormat")=1` のグローバル変数を設定します。これは、すべてのプロセスのすべての既定の形式を現在のロケール値に設定します。これらの既定は、このグローバルが変更されるまで維持されます。

注釈 ここでは、localeopt が未定義または 0 に設定されているときに適用されるユーザ・ロケール定義について説明します。localeopt=1 の場合、\$ZDATETIME は、定義済みの ODBC ロケールを使用します。

以下の例で、最初の \$ZDATETIME は、ロケールの既定形式で日付と時刻を返します。入力引数は、\$ZTIMESTAMP 特殊変数、既定の dformat 引数と tformat 引数、および 2 小数桁に設定された precision です。ほとんどのロケールでは、最初の \$ZDATETIME は、dformat=1 または、日付の区切り文字にスラッシュ、秒の小数部用の小数点区切り文字にドットを使用したアメリカ形式の日付と時刻を返します。

ChangeVals セクションでは、最初の SetFormatItem() メソッドは、2 つ目の \$ZDATETIME で示されるように、ロケールの日付形式の既定値を dformat=4、またはヨーロッパ日付形式 (DD/MM/[YY]YY) に変更します。2 つ目の SetFormatItem() メソッドは、(dformat が -1、1、4、および 15 の場合に影響する) 日付区切り文字に対するロケールの既定値を変更します。この例では、3 つ目の \$ZDATETIME で示されるように、日付区切り文字は、ドット (".") に設定されています。3 番目の SetFormatItem() メソッドは、最後の \$ZDATETIME で示されるように、このロケールの小数点区切り文字をヨーロッパ標準 (",") に変更します。その後、このプログラムは日付形式の初期値をリストアします。



## ObjectScript

```

InitializeLocaleFormat
    SET ^SYS("NLS","Config","LocaleFormat")=1
InitialVals
    SET fmt=##class(%SYS.NLS.Format).GetFormatItem("DateFormat")
    SET sep=##class(%SYS.NLS.Format).GetFormatItem("DateSeparator")
    SET dml=##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
    WRITE !,$ZDATETIME($ZTIMESTAMP,,,2)
ChangeVals
    SET x=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",4)
    WRITE !,$ZDATETIME($ZTIMESTAMP,,,2)
    SET y=##class(%SYS.NLS.Format).SetFormatItem("DateSeparator",".")
    WRITE !,$ZDATETIME($ZTIMESTAMP,,,2)
    SET z=##class(%SYS.NLS.Format).SetFormatItem("DecimalSeparator","")
    WRITE !,$ZDATETIME($ZTIMESTAMP,,,2)
RestoreVals
    SET x=##class(%SYS.NLS.Format).SetFormatItem("DateFormat",fmt)
    SET y=##class(%SYS.NLS.Format).SetFormatItem("DateSeparator",sep)
    SET z=##class(%SYS.NLS.Format).SetFormatItem("DecimalSeparator",dml)
    WRITE !,$ZDATETIME($ZTIMESTAMP,,,2)

```

## \$ZDATE と \$ZDATETIME

\$ZDATETIME は \$ZDATE に類似しています。\$ZDATETIME は、日付と時刻を組み合わせた値を変換するのに対し、\$ZDATE は、日付値のみを変換します。次に例を示します。

## ObjectScript

```
WRITE $ZDATE($HOROLOG)
```

02/22/2018 のようにフォーマットされた、現在の日付を返します。

## ObjectScript

```
WRITE $ZDATETIME($HOROLOG)
```

02/22/2018 13:53:57 のようにフォーマットされた、現在の日付と時刻を返します。

\$ZDATE は、5 から 8 までの tformat 値をサポートしていません。

## 関連項目

- [JOB コマンド](#)
- [\\$ZDATE 関数](#)
- [\\$ZDATEH 関数](#)
- [\\$ZDATETIMEH 関数](#)
- [\\$ZTIME 関数](#)
- [\\$HOROLOG 特殊変数](#)
- [\\$ZTIMESTAMP 特殊変数](#)
- [%DATE ユーティリティ \(<https://docs.intersystems.com/priordocexcerpts> で利用可能な旧ドキュメントに記載\)](#)
- [各国言語サポートのシステム・クラスの使用法](#)

## \$ZDATETIMEH (ObjectScript)

日付と時刻を検証し、これを表示形式から InterSystems IRIS の内部形式に変換します。

### 構文

```
$ZDATETIMEH(datetime,dformat,tformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
$ZDTH(datetime,dformat,tformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt,localeopt)
```

### 引数

引数	説明
datetime	入力する日付と時刻の値。表示形式で指定された日付と時刻を表す文字列です。 \$ZDATETIMEH は、この日付文字列を \$HOROLOGY 形式に変換します。datetime 値は、(多様な形式で指定される) 明示的な日付と時刻、または既定で 0 となる時刻値の付いた (多様な形式で指定される) 明示的な日付、あるいは指定された時刻値または既定で 0 となる時刻値の付いた現在の日付を表す文字列 “T” または “t” のいずれかになります。“T” または “t” 文字列は、オプションで符号付きの整数のオフセットを含むことができます。後述の “ <a href="#">datetime</a> ” を参照してください。
dformat	オプション — datetime の日付部分の日付形式を表す整数コード。datetime が “T” の場合、dformat は 5、6、7、8、9、または 15 のいずれかとする必要があります。後述の “ <a href="#">dformat</a> ” を参照してください。
tformat	オプション — datetime の時刻部分の時刻形式を表す整数コード。後述の “ <a href="#">tformat</a> ” を参照してください。
monthlist	オプション — 一連の月名を表す文字列または変数の名前。この文字列は、区切り文字から始まり、12 個のエントリは、この区切り文字で分けられる必要があります。後述の “ <a href="#">monthlist</a> ” を参照してください。
yearopt	オプション — 年を 2 桁または 4 桁のどちらで表記するかを指定する整数コード。後述の “ <a href="#">yearopt</a> ” を参照してください。
startwin	オプション — 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値。後述の “ <a href="#">startwin</a> ” を参照してください。
endwin	オプション — 日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最後を指定する数値。後述の “ <a href="#">endwin</a> ” を参照してください。
mindate	オプション — 有効な日付範囲の下限。\$HOROLOGY 整数日付カウントの形式で指定し、0 は 1840 年 12 月 31 日を表します。正または負の整数として指定できます。後述の “ <a href="#">mindate</a> ” を参照してください。
maxdate	オプション — 有効な日付範囲の上限。\$HOROLOGY 整数日付カウントの形式で指定します。後述の “ <a href="#">maxdate</a> ” を参照してください。
erropt	オプション — datetimeが無効のときに返す式。この引数に値を指定すると、datetime 値が無効または範囲外の場合に生じるエラー・コードが抑制されます。エラー・メッセージを発行する代わりに、\$ZDATETIMEH は erropt を返します。後述の “ <a href="#">erropt</a> ” を参照してください。



引数	説明
localeopt	<p>オプション - dformat、tformat、monthlist、yearopt、mindate、maxdate の既定値、およびその他の日付と時刻の特性 (日付区切り文字など) に対してどのロケールを使用するかを指定するブーリアン・フラグ。</p> <p>localeopt=0: 現在のロケール・プロパティ設定によって、これらの引数の既定値が決定されます。</p> <p>localeopt=1: ODBC 標準ロケールによって、これらの引数の既定値が決定されます。</p> <p>localeopt 未指定: dformat 値によって、これらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。日本およびイスラム暦の日付の dformat 値については、それぞれの既定値が使用されます。その他すべての dformat 値については、現在のロケール・プロパティ設定が既定値として使用されます。後述の "localeopt" を参照してください。</p> <p>オプション - どのロケールを使用するかを指定するブーリアン・フラグ。0 の場合、現在のロケールが、日付区切り文字や時刻区切り文字などの文字、文字列、および日付と時刻のフォーマットに使用するオプションを決定します。1 の場合は、ODBC ロケールが、これらの文字、文字列、およびオプションを決定します。既定値は 0 です。ただし、dformat=3 の場合、既定値は 1 です。下記を参照してください。</p>

指定の引数値間で省略された引数は、プレースホルダのコンマで示されます。末尾のプレースホルダのコンマは必要ありませんが、あってもかまいません。省略された引数を示すコンマの間に空白があってもかまいません。

## 説明

\$ZDATETIMEH は、指定された日付と時刻値を検証し、これを表示形式から内部形式に変換します。対応する \$ZDATETIME 関数は、日付と時刻値を内部形式から表示形式に変換します。内部形式は、\$HOROLOG または \$ZTIMESTAMP によって使用される形式です。コンマで区切られた 2 つの数値の文字列として、日付と時刻値を表します。

返される値は、使用する引数によって異なります。

\$ZDATETIMEH(datetime) は、日付と時刻値を "MM/DD/[YY]YY hh:mm:ss[.fff]" 形式から \$HOROLOG 形式に変換します。

構文	意味
MM	2 桁の月
DD	2 桁の日
[YY]YY	1900 年から 1999 年までは 2 桁または 4 桁。1900 年より前、または 1999 年より後の年は 4 桁
hh	24 時間形式の時間
mm	分
ss	秒
ffff	小数秒 (小数点以下 0 から 9 桁まで)

\$ZDATETIMEH(datetime,dformat,tformat,monthlist,yearopt,startwin,endwin,mindate,maxdate,erropt) は、(\$ZDATETIME を使用して) 最初に指定された日付と時刻値を \$HOROLOG 形式または \$ZTIMESTAMP 形式に変換します。dformat、tformat、yearopt、startwin および endwin の値は、\$ZDATETIME によって使用される値と同一です。

ただし、dformat に 5、6、7、8、または 9 を指定した場合、\$ZDATETIMEH は dformat コード 1、2、3、5、6、7、8、9 に対応するあらゆる外部アメリカ日付形式の日付を認識して変換します。有効なアメリカ日付形式のリストは、"\$ZDATEH" を

参照してください。dformat に 15 を使用している場合、\$ZDATETIMEH は、曖昧でないあらゆるヨーロッパ日付形式の日付を認識して変換します。有効なヨーロッパ日付形式のリストは、“\$ZDATEH” を参照してください。

また、dformat 値に 5、6、7、8、9、または 15 を指定した場合も、文字 “T” または “t” で指定した現在の日付を使用できます。T や t の後に、プラス記号 (+) またはマイナス記号 (-)、および現在の日付より前または後の日数を表す数値を記述することもできます。

\$ZDATETIMEH は、関数呼び出しで指定した時刻形式に関係なく、8 つの時刻形式の時刻をすべて認識して変換します。さらに、\$ZDATETIMEH は、接尾語 “AM、PM、NOON、MIDNIGHT” を認識して変換します。これらの接尾語は、大文字、小文字、または両方を混在させて表すことができます。これらの接尾語を文字に省略することもできます。

認識する形式は、以下のとおりです。

- ・ 既定の日付形式、MM/DD/[YY]YY
- ・ 形式 DDMmm[YY]YY
- ・ ODBC 形式 [YY]YY-MM-DD
- ・ 形式、DD/MM/[YY]YY
- ・ 形式 Mmm D, YYYY
- ・ 形式 Mmm D YYYY
- ・ 形式 Mmm DD YY
- ・ 形式 YYYYMMDD (数値形式)

## 指数

### datetime

\$HOROLOG 形式に変換する日付と時刻の文字列。以下のいずれかを指定できます。

- ・ 先頭に日付、その後に 1 つの空白スペース、時刻と続く、1 つの文字列に評価される式として指定します。
- ・ 日付のみを指定する文字列として評価される式。tformat が 7 または 8 (この場合、深夜 (0) からのローカル・タイム・ゾーンのアフセットが既定時刻となる) の場合を除き、時刻の既定値は深夜 (0) となります。
- ・ 文字コード “T” または “t” は、現在の日付を指定します。この文字の後に、オプションでプラス記号、またはマイナス記号、および現在の日付からのアフセットの日数で指定する整数を付けることができます。この日付式の後に、1 つの空白スペースと時刻式を付けることも、時刻を既定の深夜 (0) とすることもできます。(tformat が 7 または 8 の場合、時刻は既定の深夜 (0) からのローカル・タイム・ゾーンのアフセットとなります。)この現在の日付オプションを使用する場合、dformat の値として 5、6、7、8、9、または 15 を指定する必要があります。

有効な日付値、および時刻値は、現在のローカルの DateFormat 引数と TimeFormat 引数、および dformat 引数と tformat 引数に指定する値によって異なります。日付に関する詳細は、“\$ZDATEH” を参照してください。

既定では、最も早い有効な datetime 日付は 1840 年 12 月 31 日です (内部 \$HOROLOG 表現では 0)。DateMinimum プロパティの既定値は 0 であるため、既定では、日付は正の整数に制限されています。現在のローカルの DateMinimum プロパティが、過去の日付を表す負の整数以下の負の整数に設定されていれば、過去の日付を負の整数を使用して指定できます。DateMinimum の有効な最小値は -672045 であり、これは 0001 年 1 月 1 日に対応します。InterSystems IRIS では、ISO 8601 標準に準拠してグレゴリオ暦が “西暦 1 年” まで遡って適用された先発グレゴリオ暦が使用されています。その理由の 1 つは、グレゴリオ暦が採用された時期は国ごとに異なるからです。例えば、欧州大陸諸国の多くでは 1582 年に採用された一方で、英国と米国では 1752 年に採用されました。このため、ご使用の地域でグレゴリオ暦が採用された時期より前の InterSystems IRIS の日付は、その当時にその地域で採用されていた暦に基づいて記録された歴史上の日付に対応していない可能性があります。1840 年より前の日付の詳細は、“mindate” 引数を参照してください。

## dformat

日付の形式です。有効な値は以下のとおりです。

値	意味
1	MM/DD/[YY]YY (07/01/97 または 03/27/2002) – <a href="#">アメリカの数値形式</a> 。現在のロケール用の正しい日付区切り文字 (/ または .) を指定する必要があります。
2	DD Mmm [YY]YY (01 Jul 97 あるいは 27 Mar 2002)
3	[YY]YY-MM-DD (1997-07-01 や 2002-03-27) – ODBC 形式
4	DD/MM/[YY]YY (01/07/97 または 27/03/2002) – <a href="#">ヨーロッパの数値形式</a> 。現在のロケール用の正しい日付区切り文字 (/ または .) を指定する必要があります。
5	Mmm D, YYYY (Jul 1, 1997 あるいは Mar 27, 2002)
6	Mmm D YYYY (Jul 1 1997 あるいは Mar 27 2002)
7	Mmm DD [YY]YY (Jul 01 1997 あるいは Mar 27 2002)
8	YYYYMMDD (19930701 あるいは 20020327) – 数値形式
9	Mmmmm D, YYYY (July 1, 1997 あるいは March 27, 2002)
13	[D]D/[M]M/YYYY (1/7/2549 または 27/11/2549) – タイ語の日付形式。日と月は、ヨーロッパ形式と同じですが、先頭に 0 は付きません。年は仏教紀元 (BE) で、グレゴリオ暦の年に 543 年追加して計算されます。
15	DD/MM/[YY]YY、YYYY-MM-DD、日付区切り文字を使用した曖昧ではないあらゆるヨーロッパ日付形式、または日付区切り文字を使用しない YYYYMMDD。日付区切り文字は、現在のロケールで指定されている日付区切り文字に関係なく、英数字以外の任意の文字 (空白も含む) ならどれでもかまいません。また、monthlist 名および “T” も使用できます。有効なヨーロッパ日付形式のリストは、 <a href="#">“\$ZDATEH”</a> を参照してください。
16	YYYYc[M]Mc[D]Dc – 日本の日付形式。年、月、および日の数は、他の日付形式と同じです。先頭のゼロは省略されます。“年”、“月”、および“日”の日本語の文字 (ここでは c として表示) は、年、月、および日の数の後に挿入されます。これらの文字は、年=\$CHAR(24180)、月=\$CHAR(26376)、および日=\$CHAR(26085) です。
17	YYYYc[M]Mc[D]Dc – 日本の日付形式。dformat 16 と同じです。ただし、“年” および “月” の日本語の文字の後に空白が挿入されます。
18	[D]D Mmmmm YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月名を示す)。日の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 Muharram 0001。
19	[D]D [M]M YYYY – Tabular Hijri (イスラム) 日付形式 (完全な月番号を示す)。日と月の先頭のゼロは省略され、年の先頭のゼロは組み込まれます。InterSystems IRIS の日付 -445031 (07/19/0622 C.E.)= 1 1 0001。
20	[D]D Mmmmm YYYY – Observed Hijri (イスラム) 日付形式 (完全な月名を示す)。既定で Tabular Hijri (dformat 18) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。
21	[D]D [M]M YYYY – Observed Hijri (イスラム) 日付形式 (完全な月番号を示す)。既定で Tabular Hijri (dformat 19) になります。Tabular での計算をオーバーライドするには、新月周期の観測を追加するために %Calendar.Hijri クラスを使用します。

値	意味
-1	有効な dformat 値は、 <a href="#">現在のロケール</a> の DateFormat プロパティから取得します。dformat を指定しなければ、これが既定の振る舞いになります。
-2	<p>\$ZDATETIMEH は、UTC 秒の整数値を取り、対応するローカルの <a href="#">\$HOROLOG</a> の日付/時刻値を返します。これは、\$ZDATETIME dformat -2 の逆になります。詳細は、“<a href="#">\$ZDATETIME</a> dformat -2” を参照してください。</p> <p>datetime の入力値は、ISO C プログラミング言語標準の定義に従って time() ライブラリ関数から返される値です。例えば、POSIX 準拠システムでは、この値は協定世界時 1970 年 1 月 1 日 00:00:00 (January 1, 1970 00:00:00 UTC) を起点とする経過秒数です。</p> <p>tformat、monthlist、yearopt、startwin、および endwin の各引数は無視されます。</p>
-3	<p>\$ZDATETIMEH は、<a href="#">\$ZTIMESTAMP</a> 内部形式で指定された datetime 値を取り、この値を UTC 時刻からローカル時刻に変換し、その結果をこの内部形式で返します。tformat、monthlist、yearopt、startwin、および endwin の各引数は無視されます。\$ZDATETIME はこの逆の処理を行います（現在、この日付変更の際に、時刻変換の変則性が発生する可能性があります。この変則性は tformat 値が 7 または 8 の場合に該当するものです。これにより、1970 年より前の日付、2038 年より後の日付、およびサマータイムの開始日と終了日などのローカル時刻調整の境界日が影響を受ける可能性があります）。</p>

以下はその説明です。

構文	意味
YYYY	YYYY は 4 桁の年です。[YY]YY は、datetime がアクティブ・ウィンドウ内で 2 桁の日付の場合は 2 桁の年、それ以外は 4 桁の数字です。
MM	2 桁の月。
D	日付が 10 未満の場合は 1 桁、それ以外は 2 桁の日。
DD	2 桁の日。
Mmm	<p>Mmm は、現在のロケールの MonthAbbr プロパティから取得した月の省略形です。以下はその既定値です。</p> <p>“Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec”</p> <p>別の月の省略形（あるいは長さに制限のない名前）は、\$ZDATETIMEH に monthlist 引数として指定したオプションのリストから取得することができます。</p>
Mmmmm	<p>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。以下はその既定値です。“January February March ...November December”</p> <p>別の月の名前は、\$ZDATETIMEH に monthlist 引数として指定したオプションのリストから取得することができます。</p>

### dformat の既定値

dformat を省略するか、-1 に設定した場合、dformat の既定値は localeopt 引数と NLS の DateFormat プロパティに依存します。

- localeopt=1 の場合、dformat の既定値は ODBC 形式です。tformat、monthlist、yearopt、mindate、および maxdate 引数の既定値も、ODBC 形式に設定されます。これは、dformat=3 を設定した場合と同じです。
- localeopt=0 または未指定の場合、dformat の既定値は NLS の DateFormat プロパティから取得されます。  
DateFormat=3 の場合、dformat の既定値は ODBC 形式です。ただし、DateFormat=3 は tformat、monthlist、

yearopt、mindate、および maxdate 引数の既定値に影響しません。これらは、NLS の現在のロケール定義で指定されます。

ユーザのロケールの既定の日付プロパティを決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateFormat"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DateSeparator")
```

\$ZDATETIMEH は、dformat=1 または 4 の場合に、[現在のロケールの DateSeparator プロパティ](#)の値 (/ または .) を、月、日、および年の区切り文字として使用します。

ヨーロッパ日付形式 (dformat=4、DD/MM/YYYY の順) は、ヨーロッパの多くの言語 (すべてではない) で既定です。これには、“/” を **DateSeparator** 文字として使用するイギリス英語、フランス語、ドイツ語、イタリア語、スペイン語、ポルトガル語、および “.” を **DateSeparator** 文字として使用するチェコ語 (csyw)、ロシア語 (rusw)、スロバキア語 (skyw)、スロベニア語 (svnw)、ウクライナ語 (ukrw) が含まれます。サポートされるロケールの既定の日付形式の詳細は、“[日付](#)”を参照してください。

## dformat の設定

dformat が 3 (ODBC 形式の日付) の場合は、tformat、monthlist、yearopt、mindate、および maxdate 引数の既定値にも ODBC 形式の既定値が使用されます。現在のロケールの既定値は無視されます。

dformat が 16 または 17 (日本の日付形式) の場合、日付形式はロケールの設定に依存しません。日本の日付形式は、任意の InterSystems IRIS インスタンスで使用できます。

dformat が 18、19、20、または 21 (イスラム暦の日付形式) で、localeopt が未指定の場合、引数の既定値は現在のロケールの既定値ではなく、イスラム暦の既定値になります。monthlist 引数の既定値は、ラテン文字で書き直されたアラビア語の月名になります。tformat、yearopt、mindate、および maxdate 引数の既定値は、ODBC の既定値になります。日付区切り文字の既定値は、ODBC の既定値や現在のロケールの **DateSeparator** プロパティ値ではなく、イスラム暦の既定値 (空白) になります。localeopt=0 の場合は、現在のロケール・プロパティの既定値がこれらの引数に使用されます。localeopt=1 の場合は、ODBC の既定値がこれらの引数に使用されます。

## tformat

時刻値の入力形式を指定する数値。サポートされている値は、以下のとおりです。

値	意味
-1	現在のロケールの TimeFormat プロパティから、有効な tformat 値を取得します。既定は 1 です。dformat の値が 3 以外で、tformat を指定しない場合は、これが既定の動作になります。
1	時刻を “hh:mm:ss” (24 時間) の形式で指定します。これは dformat=3 である場合の既定です。
2	時刻を “hh:mm” (24 時間) の形式で指定します。
3	時刻を “hh:mm:ss[AM/PM]” (12 時間) の形式で指定します。
4	時刻を “hh:mm[AM/PM]” (12 時間) の形式で指定します。
5	時刻を “hh:mm:ss+/-hh:mm” (24 時間) の形式で指定します。時刻を現地時間で指定します。オプションの接尾語として、プラス (+) 接尾語またはマイナス (-) 接尾語とこれに続く協定世界時 (UTC) からの現地時間のオフセットが付加される場合がありますが、これは無視されます。マイナス符号 (-hh:mm) は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より早い (西向きである) ことを示します。プラス符号 (+hh:mm) は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より遅い (東向きである) ことを示します。



値	意味
6	時刻を“hh:mm+/-hh:mm”(24 時間)の形式で指定します。時刻を現地時間で指定します。オプションの接尾語として、プラス(+)接尾語またはマイナス(-)接尾語とこれに続く協定世界時(UTC)からの現地時間のオフセットが付加される場合がありますが、これは無視されます。マイナス符号(-hh:mm)は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より早い(西向きである)ことを示します。プラス符号(+hh:mm)は、返された時間と分のオフセット数により、現地時間がグリニッジ子午線より遅い(東向きである)ことを示します。
7	時刻を“hh:mm:ssZ”(24 時間)の形式で指定します。時刻は、協定世界時(UTC)で指定する必要があります。オプションの“Z”接尾語が付加されるか、または省略される場合がありますが、これは無視されます。この接尾語は、時刻が現地時間ではなく、協定世界時(UTC)であると想定されていることを示すだけです。
8	時刻を“hh:mmZ”(24 時間)で指定します。時刻は、協定世界時(UTC)で指定する必要があります。オプションの“Z”接尾語が付加されるか、または省略される場合がありますが、これは無視されます。この接尾語は、時刻が現地時間ではなく、協定世界時(UTC)であると想定されていることを示すだけです。

datetime 文字列に日付部分と時刻部分の両方が含まれている場合、時刻部分は、1 つのスペースと大文字の“T”のいずれかで日付部分から分離されます。時刻部分がある場合は、以下のようになります。

- 時刻形式 1 から 6 は、datetime が結果と同じタイム・ゾーンを使用して現地時間を指定すると想定します。
- 時刻形式 7 および 8 は、datetime が UTC 時刻を指定すると想定します。これらの形式は、日付と時刻の両方をシステム・ローカル時刻に変換します。

時刻形式 5 から 8 では、datetime の時刻値の後に大文字の“Z”あるいは“+”または“-”で始まる UTC オフセットのいずれかで構成される接尾語が続く場合があります。接尾語があっても、タイム・ゾーン変換には影響しません。

**注釈** dformat 値が -2 および -3、tformat 値が 7 および 8、かつ tformat 値が 5 および 6 のときに生成される UTC オフセットを含む変換には、以下のプラットフォーム依存の変則性があります。

- ローカル時刻調整の境界でのこの動作はオペレーティング・システム・プラットフォームによって異なります。ローカル時刻調整の変更が発生し、ローカル時計が逆方向にシフトすると(サマータイムの終了日の“戻し”)、そのローカル時刻の時間が繰り返されます。この 2 時間以内に時刻を変換すると、InterSystems IRIS では、このローカル時刻の 1 時間目に対して時刻変換処理を行うのか、2 時間目に対して行うのかを判断できません。\$ZDATETIME では、プラットフォーム固有のランタイム・ライブラリで使用されている前提を採用します。したがって、この時間枠の範囲では、オペレーティング・システム・プラットフォームによって時刻変換の結果が異なる可能性があります。
- InterSystems IRIS は、オペレーティング・システム・プラットフォームがサポートしているすべての日付に対して、標準時オフセットを使用してローカル時刻と UTC 時刻の変換を実行します。

指定の日付がプラットフォームでサポートされている最も早い日付より前の場合、InterSystems IRIS は、1902-01-01 に対する標準時オフセットを使用します(この日付がプラットフォームでサポートされている場合)。日付 1902-01-01 がプラットフォームでサポートされていない場合、InterSystems IRIS は、1970-01-01 に対する標準時オフセットを使用します。現地時刻調整オフセット(サマータイムなど)は無視されます。

指定の日付がプラットフォームでサポートされている最も遅い日付より後の場合、InterSystems IRIS は、2010-01-01 から 2037-12-31 の範囲内で対応する日付を計算し、その対応する日付に対する標準時オフセットを使用します。このアルゴリズムは、2100-02-28 までの日付に対して、正確な時刻オフセットを実現します(今後、日付/時刻を管理する法律の変更がない場合)。

ユーザのロケールの既定の時刻形式を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
```

## 12 時間形式 (tformat 3 および 4)

12 時間形式では、午前と午後を時間接尾語で指定します。ここでは、AM と PM で表示されています。ユーザのロケールの既定の時間接尾語を決定するには、以下のように GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
```

dformat の値が 3、18、19、20、21 以外の場合、**AM** および **PM** プロパティは既定で現在のロケール定義になります。dformat=3 (ODBC の日付形式) および dformat=18、19、20、または 21 (イスラム暦の日付形式) の場合は、現在のロケールのプロパティ値に関係なく、時間接尾語は常に“AM”および“PM”になります。**AM** および **PM** プロパティは、日本のロケール jpww を除くすべてのロケールで“AM”と“PM”になります。

日本のロケール (jpnw、jpuw、jpww、zdsu、zdtw、zduw)、ポルトガル (ptbw)、ロシア (rusw)、およびウクライナ (ukrw) を除くすべてのロケールで、既定では深夜と正午は“MIDNIGHT”および“NOON”で表されます。ただし、dformat=3 の場合、\$ZDATETIMEH では、ロケールの既定の設定に関係なく、常に ODBC 標準値を使用します。

## monthlist

区切り文字で区切られた月名または月の省略名の文字列に解決される式です。monthlist にある名前は、現在のロケールの **MonthAbbr** プロパティに指定されている既定の月の省略形の値、または **MonthName** に指定されている月名の値を置き換えます。

monthlist は、dformat が 2、5、6、7、9、15、18、または 20 の場合にのみ有効です。dformat がそれ以外の値の場合、\$ZDATETIMEH は monthlist を無視します。

monthlist 文字列の形式は、以下のとおりです。

- 文字列の最初の文字は、区切り文字 (通常はスペース) です。monthlist 内の最初の月名の前と、各月名の間に、同じ区切り文字を置く必要があります。1 文字からなる任意の区切り文字を指定できます。この区切り文字は、指定された datetime 値の月、日、年の間に指定する必要があります。このため、スペースが通常推奨される区切り文字となります。
- 月名の文字列は、1 月から 12 月に対応する 12 個の区切り値を含む必要があります。12 より多い、または 12 より少ない月名を指定することは可能ですが、datetime の月に該当する月名がない場合、<ILLEGAL VALUE> エラーが発生します。

monthlist を省略するか、monthlist の値として -1 を指定した場合は、現在のロケールの **MonthAbbr** または **MonthName** プロパティで定義されている月名のリストが \$ZDATETIMEH で使用されます (localeopt=1、または monthlist の既定値が ODBC の月リスト (英語) である場合を除く)。localeopt が未指定で、dformat が 18 または 20 (イスラム暦の日付形式) の場合、monthlist の既定値はイスラム暦の月リスト (ラテン文字で表されたアラビア語の名前) になり、**MonthAbbr** および **MonthName** プロパティ値は無視されます。

ユーザのロケールの既定の月名および月名の省略形を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthName"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("MonthAbbr"),!
```



## yearopt

dformat 値に 0、1、2、4、7 を使用して、年を 2 桁で表示する時間ウィンドウを指定する数値コードです。yearopt の値は以下のとおりです。

値	意味
-1	現在のロケールの YearOption プロパティから有効な yearopt 値を取得します。既定値は 0 です。yearopt を指定しない場合は、これが既定の動作になります。
0	(%DATE ユーティリティにより構築される) プロセス特有のスライディング・ウィンドウが有効になっていない限り、20 世紀 (1900 年から 1999 年まで) の日付は 2 桁で表示します。スライディング・ウィンドウが有効な場合は、指定範囲内の日付のみ 2 桁の年を表示します。1900 年代以外もしくはプロセス特有のスライディング・ウィンドウ以外の日付は、すべて 4 桁で表示します。
1	20 世紀の日付は 2 桁の年、それ以外は 4 桁で表します。
2	すべての日付を 2 桁の年で表します。
3	startwin と (オプションで) endwin に定義した一時的なスライディング・ウィンドウ範囲内に収まる日付を、2 桁の年で表します。それ以外の日付は 4 桁で表します。yearopt=3 の場合、startwin と endwin は、\$HOROLOG 形式の絶対日付です。
4	すべての日付を 4 桁の年で表します。
5	startwin と (オプションで) endwin に定義したスライディング・ウィンドウ範囲内に収まる日付をすべて、2 桁の年で表します。それ以外の日付は 4 桁の年で表します。yearopt=5 の場合、startwin と endwin は相対年になります。
6	2 桁の年で現在の世紀にあるすべての日付を表します。それ以外は 4 桁の年で表します。

yearopt を省略するか、yearopt の値として -1 を指定した場合は、現在のロケールの YearOption プロパティが \$ZDATETIMEH で使用されます (localeopt=1、または yearopt の既定値が ODBC の年オプションである場合を除く)。localeopt=0 または未指定で、dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、yearopt の既定値は ODBC の年オプション (4 桁の年) になります。イスラム暦の日付では、YearOption プロパティの値は無視されます。

## startwin

日付を 2 桁の年で表す必要があるスライディング・ウィンドウの最初を指定する数値です。yearopt の値に 3 または 5 を使用する場合は、startwin を指定する必要があります。startwin は、他の yearopt 値では無効になります。

yearopt=3 のとき、startwin はスライディング・ウィンドウの最初の日付を示す \$HOROLOG 日付形式の絶対日付です。

yearopt=5 のとき、startwin は、現在の年より前の年数で表される、スライディング・ウィンドウの最初の年を示す数値です。スライディング・ウィンドウは常に、startwin で指定した年の初日 (1 月 1 日) から開始します。

## endwin

日付を 2 桁の年で表すスライディング・ウィンドウの最後を指定する数値です。yearopt が 3 あるいは 5 のとき、endwin をオプションで指定する必要があります。endwin は、その他の yearopt 値では無効になります。

yearopt=3 のとき、endwin はスライディング・ウィンドウの最後の日付を示す \$HOROLOG 日付形式の絶対日付です。

yearopt=5 のとき、endwin は、現在の年から後の年数を示すスライディング・ウィンドウの最後の年を示す数値です。スライディング・ウィンドウは常に、endwin で指定した年の 12 月 31 日で終了します。endwin を指定しない場合、既定は、startwin 以降 100 年後の 12 月 31 日です。

endwin を省略した場合または -1 を指定した場合、スライディング・ウィンドウの有効期限は 100 年間です。endwin 値を -1 とすると特殊なケースになります。endwin の値が -1 よりも大きい場合または小さい場合に errop 返される場合で

も、-1 の場合は必ず日付値が返されます。このため、100 年間のウィンドウを指定する場合は endwin を省略し、その他の場合でも endwin には負の値を指定しないようにすることをお勧めします。

startwin と endwin の両方を指定する場合、指定するスライディング・ウィンドウの年数は 100 年以内に収める必要があります。

## mindate

有効な日付範囲(両端を含む)の下限を指定する式です。\$HOROLOG 整数日付カウント(例えば、1/1/2013 は 62823 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付文字列("62823,43200" など)の時刻部分は含めることも省略することもできますが、解析されるのは mindate の日付部分のみです。mindate より前の datetime 値を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

mindate に指定できる値は以下のとおりです。

- ・ 正整数：通常は、mindate には、1840 年 12 月 31 日より後の最も早い有効日付を設定するための正の整数を指定します。例えば、mindate に 21550 を指定すると、1900 年 1 月 1 日が最も早い有効日付として設定されます。有効な最大値は 2980013 です(9999 年 12 月 31 日)。
- ・ 0：1840 年 12 月 31 日を最も早い日付として指定します。これは **DateMinimum** プロパティの既定値です。
- ・ -2 以下の負の整数：1840 年 12 月 31 日から遡ってカウントする最も早い日付を指定します。例えば、mindate に -14974 を指定すると、1800 年 1 月 1 日が最も早い有効日付として設定されます。負の mindate 値が有効となるのは、現在のロケールの **DateMinimum** プロパティがその値以下の負の整数に設定されている場合のみです。有効な最小値は -672045 です。
- ・ mindate を省略するか、-1 を指定した場合は、現在のロケールの **DateMinimum** プロパティ値が既定値になります(localeopt=1 または mindate の既定値が 0 の場合を除く)。localeopt が未指定で dformat=3 の場合、mindate の既定値は 0 になります。localeopt が未指定で dformat が 18、19、20、または 21 (イスラム暦の日付形式) の場合、mindate の既定値は 0 になります。

**DateMinimum** プロパティは、以下のように取得および設定できます。

## ObjectScript

```
SET min=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "initial DateMinimum value is ",min,!
Permit18thCenturyDates
SET x=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",-51498)
SET newmin=##class(%SYS.NLS.Format).GetFormatItem("DateMinimum")
WRITE "set DateMinimum value is ",newmin,!
RestrictTo19thCenturyDates
WRITE $ZDATETIMEH("05/29/1805 12:00:00",1,,,,,-14974),!!
ResetDateMinimumToDefault
SET oldmin=##class(%SYS.NLS.Format).SetFormatItem("DateMinimum",min)
WRITE "reset DateMinimum value from ",oldmin," to ",min
```

mindate を指定する際は、maxdate を指定しても指定しなくてもかまいません。maxdate より大きい mindate を指定すると、<ILLEGAL VALUE> エラーが発生します。

## maxdate

有効な日付範囲(両端を含む)の上限を指定する式です。\$HOROLOG 整数日付カウント(例えば、1/1/2100 は 94599 と表される)または \$HOROLOG 文字列値の形式で指定できます。\$HOROLOG 日付("94599,43200" など)の時刻部分は含めることも省略することもできますが、解析されるのは maxdate の日付部分のみです。

maxdate を省略するか、または -1 を指定すると、現在のロケールの **DateMaximum** プロパティから日付の上限が取得されます。既定値は、\$HOROLOG の日付部分で許容される最大値である、2980013 (9999 年 12 月 31 日に対応)です。ただし、**DateMaximum** プロパティの適用は localeopt の設定によって制御されます。localeopt=1 (dformat=3 の場合の既定値) の場合、現在のロケール設定に関係なく、日付の既定最大値は ODBC の値 (2980013) になります。イスラム暦の日付形式でも ODBC の既定値が使用されます。タイ日付形式 (dformat=13) の最大日付は BE 9999 年 12 月 31 日で \$HOROLOG 2781687 に対応します。

maxdate より大きい datetime を指定すると、<VALUE OUT OF RANGE> エラーが発生します。

2980013 より大きい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

maxdate を指定する際は、mindate を指定しても指定しなくてもかまいません。mindate より小さい maxdate を指定すると、<ILLEGAL VALUE> エラーが発生します。

### erroppt

この引数に値を指定すると、datetime 値が無効または範囲外の場合に生じるエラーが抑制されます。\$ZDATETIMEH 関数は、<ILLEGAL VALUE> または <VALUE OUT OF RANGE> エラーを生成する代わりに erroppt 値を返します。

InterSystems IRIS は、datetime に対して標準の数値評価を実行します。これは常に mindate から maxdate までの範囲内の整数日付に評価される必要があります。したがって、7、"7"、+7、0007、7.0、"7 dwarves"、--7 は、すべて同じ日付値 (01/07/1841) に評価されます。既定では、2980013 を超える値または 0 未満の値は <VALUE OUT OF RANGE> エラーを生成します。小数値は <ILLEGAL VALUE> エラーを生成します。数値以外の文字列 (NULL 文字列を含む) は 0 に評価され、\$HOROLOG の最初の日付 (12/31/1840) が返されます。

erroppt 引数は、datetime の無効な値または範囲外の値が原因で生成されるエラーのみを抑制します。他の引数の無効な値、あるいは範囲外の値が原因で発生したエラーについては、erroppt が指定されているかどうかに関係なく、常にエラーが生成されます。例えば、\$ZDATETIMEH で endwin が startwin より前になる値のスライディング・ウィンドウを指定すると、常に <ILLEGAL VALUE> エラーが発生します。同様に、maxdate が mindate より小さい場合、<ILLEGAL VALUE> エラーが発生します。

### localeopt

このブーリアン引数には、ロケールで指定される引数 dformat、tformat、monthlist、yearopt、mindate、および maxdate の既定値のソースとして、ユーザの現在のロケール定義または ODBC のロケール定義が指定されます。

- ・ localeopt=0 の場合は、これらすべての引数で現在のロケール定義の既定値が使用されます。
- ・ localeopt=1 の場合は、これらすべての引数で ODBC の既定値が使用されます。
- ・ localeopt が未指定の場合は、dformat 引数によってこれらの引数の既定値が決定されます。dformat=3 の場合は、ODBC の既定値が使用されます。dformat が 18、19、20、または 21 の場合は、現在のロケール定義に関係なく、イスラム暦の日付および時刻形式の既定値が使用されます。dformat がそれ以外の値の場合は、現在のロケール定義の既定値が使用されます。詳細は、"[dformat](#)" の説明を参照してください。

ODBC 標準ロケールは変更できません。これは、異なる各国語サポート (NLS) の選択を行った InterSystems IRIS プロセス間で移植できる、日付文字列および時刻文字列をフォーマットするために使用されます。ODBC ロケールの日付と時刻の定義は、以下のとおりです。

- ・ 日付形式の既定は 3 です。したがって、dformat が未定義または -1 の場合は、日付形式 3 が使用されます。
- ・ 日付の区切り文字の既定は "/" です。ただし、日付形式の既定は 3 で、これは日付の区切り文字として "-" を常に使用します。
- ・ 年のオプションの既定は 4 桁です。
- ・ 日付の最小値は 0 で、最大値は 2980013 (\$HOROLOG の日数カウント) です。
- ・ 英語の月名、月の省略形、曜日名、曜日の省略形、および "Noon" と "Midnight" の語が使用されます。
- ・ 時刻形式は既定で 1 になります。時刻区切り文字は "." です。時刻精度は 0 (小数秒なし) です。午前と午後の指定子は、"AM" と "PM" です。

## \$ZDATETIMEH と秒の小数部

\$ZDATETIME とは異なり、\$ZDATETIMEH は時刻の有効桁数を指定することはできません。元の \$ZDATETIME 時刻形式の秒の小数部は、\$ZDATETIMEH が返す値に保持されます。

\$HOROLOG 形式も秒の小数部を返しません。

## \$ZDATETIMEH で無効な値

以下の状況では、<FUNCTION> エラーが発生します。

- ・ 無効な dformat コード (無効な整数値または整数以外の値) を指定した場合。
- ・ tformat に無効な値 (-1 未満または 8 より大きい整数値、0、整数以外の値) を指定した場合。
- ・ yearopt が 3 あるいは 5 の場合に、startwin 値を指定しない場合。

以下の状況で、<ILLEGAL VALUE> エラーが発生します。

- ・ 日付/時刻単位に無効な値を指定した場合。指定した場合は、<ILLEGAL VALUE> が発行されるのではなく、errop 値が返されます。
- ・ ODBC 日付形式の日付/時刻単位に余分な先頭のゼロを指定した場合。例えば、2007 年 2 月 3 日は、“2007-2-3” または “2007-02-03” で表せますが、“2007-002-03” に対して <ILLEGAL VALUE> を受け取ります。指定した場合は、<ILLEGAL VALUE> が発行されるのではなく、errop 値が返されます。
- ・ 指定された月数が monthlist にある値よりも大きい場合。
- ・ maxdate が mindate よりも小さい場合。
- ・ endwin が startwin よりも小さい場合。
- ・ startwin と endwin で、期間が 100 年より長い一時的なスライディング・ウィンドウが指定された場合。

以下の状況では、<VALUE OUT OF RANGE> エラーが発生します。

- ・ 1840 年 12 月 31 日より前、あるいは 9999 年 12 月 31 日より後の日付 (あるいは “T” のオフセット) を指定して、errop 値を指定しない場合。
- ・ mindate と maxdate の範囲外にある、範囲内にあれば有効であった日付 (あるいは “T” のオフセット) を指定し、errop 値を指定しない場合。

## 現在の日付

以下の例では、“T” または “t” 文字コードを使用して現在の日付を指定する方法を示します。ただし、dformat には 5、6、7、8、9、または 15 を指定する必要があります。

時刻が既定の 0 となっている現在の日付は、以下のとおりです。

### ObjectScript

```
WRITE $ZDATETIMEH("T",5)
```

時刻が既定の 0 となっている現在の日付の 3 日前は、以下のとおりです。

### ObjectScript

```
WRITE $ZDATETIMEH("T-3",5)
```

時刻が指定されている、現在の日付の 2 日後は、以下のとおりです。

### ObjectScript

```
WRITE $ZDATETIMEH("T+2 11:45:00",5)
```

## \$ZDATEH と \$ZDATETIMEH

\$ZDATETIMEH は、\$ZDATEH に似ていますが、異なる点として、日付と時刻の値の両方を内部 \$HOROLOG 形式に (時刻値を指定していない場合でも) 変換します。\$ZDATEH は、日付値のみを \$HOROLOG 形式に変換します。以下はその例です。

### ObjectScript

```
WRITE $ZDATEH("Nov 25, 2002",5)
```

これは、59133 を返します。

### ObjectScript

```
WRITE $ZDATETIMEH("Nov 25, 2002 10:08:09.539",5)
```

これは、59133,36489.539 を返します。

\$ZDATETIMEH に値を指定しない場合

### ObjectScript

```
WRITE $ZDATETIMEH("Nov 25, 2002",5)
```

これは、59133.0 を返します。

時刻値を指定せず、tformat を 7 または 8 にして \$ZDATETIMEH を指定する場合

### ObjectScript

```
WRITE $ZDATETIMEH("Nov 25, 2002",5,7)
```

59133,68400 のような値が返されます。ここで、時刻値は、深夜からのローカル・タイム・ゾーンのオフセットです。この場合、米国東部標準時間は、UTC から 5 時間オフセットされているので、ここでの時刻値は、19:00 (深夜から 5 時間オフセット) を表します。

## 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$ZDATE](#) 関数
- ・ [\\$ZDATEH](#) 関数
- ・ [\\$ZDATETIME](#) 関数
- ・ [\\$ZTIME](#) 関数
- ・ [\\$ZTIMEH](#) 関数
- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数
- ・ [%DATE](https://docs.intersystems.com/priordocexcerpts) ユーティリティ (<https://docs.intersystems.com/priordocexcerpts> で利用可能な旧ドキュメントに記載)
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ZDCHAR (ObjectScript)

\$DOUBLE 浮動小数点数を 8 バイト文字列に変換します。

### 構文

```
$ZDCHAR(n)  
$ZDC(n)
```

### 引数

引数	説明
n	IEEE 形式の浮動小数点数。値、変数、式として指定することができます。

### 説明

\$ZDCHAR は、n に対応する 8 バイト (quad) 文字の文字列を返します。文字の文字列のバイト数は、リトル・エンディアンの順序で、最下位バイトが先に表示されます。

数値 n には正または負の IEEE 浮動小数点数を指定できます。n が数値でない場合、\$ZDCHAR は空文字列を返します。IEEE 浮動小数点数の詳細は、“[\\$DOUBLE](#)” 関数を参照してください。

### 例

以下の例では、IEEE 浮動小数点数に対応する 8 バイト文字列が返されます。

```
WRITE $ZDCHAR($DOUBLE(1.4)),!  
WRITE $ZDCHAR($DOUBLE(1.4000000000000001))
```

これら 2 つの関数は、“fmmfö?” と “kmmfö?” を返します。

## \$ZDCHAR とその他の \$CHAR 関数

\$ZDCHAR は、IEEE 浮動小数点数を 8 バイト (64 ビット) quad 文字の文字列に変換します。

- ・ 整数を 64 ビット (quad) 文字の文字列に変換するには、\$ZQCHAR を使用します。
- ・ 整数を 32 ビット (long) 文字の文字列に変換するには、\$ZLCHAR を使用します。
- ・ 整数を 16 ビット (wide) 文字の文字列に変換するには、\$ZWCHAR を使用します。
- ・ 整数を 8 ビット文字の文字列に変換するには、\$CHAR を使用します。

### 関連項目

- ・ [\\$DOUBLE](#) 関数
- ・ [\\$ZDASCII](#) 関数
- ・ [\\$CHAR](#) 関数
- ・ [\\$ZWCHAR](#) 関数
- ・ [\\$ZLCHAR](#) 関数
- ・ [\\$ZQCHAR](#) 関数



# \$ZEXP (ObjectScript)

指数関数 (自然対数の逆関数)

## 構文

`$ZEXP (n)`

## 引数

引数	説明
n	任意のタイプの数。335.6 より大きな数の場合、<MAXNUMBER> エラーが発生します。-295.4 より小さい数は 0 を返します。

## 説明

\$ZEXP は指数関数  $e^n$  であり、 $e$  は定数 2.718281828 です。そのため、 $e$  の値を返すために、`$ZEXP(1)` と指定できます。`$ZEXP` は自然対数関数 `$ZLN` の逆関数です。

## 引数

**n**

任意の数字。値、変数、式として指定することができます。335.6 より大きな正数または -4944763837 未満の数の場合、<MAXNUMBER> エラーが発生します。-295.4 より小さい負の値は 0 を返します。値ゼロ (0) は 1 を返します。

非数値文字列は 0 として評価されるため、1 を返します。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)” を参照してください。

## 例

以下の例は、`$ZEXP` が `$ZLN` の逆関数であることを示しています。

### ObjectScript

```
SET x=7
WRITE $ZEXP(x),!
WRITE $ZLN(x),!
WRITE $ZEXP($ZLN(x))
```

以下の例は、負整数や正整数およびゼロに対して `$ZEXP` を返します。この例は、`$ZEXP(1)` として定数  $e$  を返します。

### ObjectScript

```
FOR x=-3:1:3 {
    WRITE !,"The exponential of ",x," = ",$ZEXP(x)
}
QUIT
```

これは、以下を返します。

```
The exponential of -3 = .04978706836786394297
The exponential of -2 = .1353352832366126919
The exponential of -1 = .3678794411714423216
The exponential of 0 = 1
The exponential of 1 = 2.718281828459045236
The exponential of 2 = 7.389056098930650228
The exponential of 3 = 20.08553692318766774
```



以下の例は、IEEE 浮動小数点数 ([\\$DOUBLE](#) 数) を使用します。最初の \$ZEXP は数値を返します。2 つ目の \$ZEXP は “INF” (または、IEEEError() メソッドの設定によっては <MAXNUMBER>) を返します。

#### ObjectScript

```
SET rtn=##class(%SYSTEM.Process).IEEEError(0)
WRITE $ZEXP($DOUBLE(1.0E2)),!
WRITE $ZEXP($DOUBLE(1.0E3))
```

以下の例は、空の文字列または数値以外の値は 0 として処理されることを示しています。

#### ObjectScript

```
WRITE $ZEXP(""),!
WRITE $ZEXP("INF")
```

いずれも 1 を返します。

## 関連項目

- ・ [\\$ZLN](#) 関数

## \$ZF (ObjectScript)

ObjectScript ルーチンから関数または非 ObjectScript プログラムを呼び出します。この関数は、コールアウト SDK のコンポーネントです。

### 構文

```
$ZF("function_name",args)
```

### 引数

引数	説明
function_name	呼び出す関数の名前
args	オプション - 関数に渡される一連の引数値

### 概要

さまざまな形式の \$ZF 関数を使用して、非 ObjectScript プログラム（シェルやオペレーティング・システムのコマンドなど）や、ObjectScript コードから関数を呼び出すことができます。他言語で記述された関数へのリンクやインタフェースを InterSystems IRIS に定義し、\$ZF を使用して ObjectScript コードから呼び出すことができます。

### その他の \$ZF 関数

\$ZF は、以下のように使用することもできます。

- ・ プログラムあるいはコマンドを実行する子プロセスを作成します。[\\$ZF\(-100\)](#)。
- ・ ダイナミック・リンク・ライブラリ (DLL) をロードし、そのライブラリから関数を実行します。[\\$ZF\(-3\)](#)、[\\$ZF\(-4\)](#)、[\\$ZF\(-5\)](#)、および [\\$ZF\(-6\)](#)。

\$ZF のこれらの実装は、[最初の引数に負の数](#)を指定します。これらについては、それぞれのリファレンスのページで説明しています。

### 引数

#### function\_name

引用符で囲まれた呼び出す関数の名前、または[負の数](#)。

#### args

args 引数の形式は、arg1, arg2, arg3, ...argn です。この引数は、引数が渡される方法の説明、および呼び出している C 関数へのエントリ・ポイントなどの項目から構成されます。

### \$ZF を使用して UNIX® システム・サービスを呼び出す

InterSystems IRIS は、\$ZF からの UNIX® システム呼び出しを使用するための、エラー・チェック機能をサポートしています。これらの呼び出しにより、非同期のイベントの確認や \$ZF でのアラーム・ハンドラの設定ができます。これらの UNIX® 機能を使用することにより、実際のエラー、〈CTRL-C〉の割り込み、および再起動が必要な呼び出しをそれぞれ区別できます。

関数宣言は、iris-cdzf.h に含まれており、以下のテーブルに示されているとおりです。

宣言	目的	メモ
int sigrtclr();	再試行フラグのクリア	sigrtchk() を使用する前に 1 回呼び出します。

宣言	目的	メモ
<code>int dzfalarm();</code>	新規 SIGALRM ハンドラの構築	\$ZF にエントリする際に、前のハンドラが自動的に保存されます。終了すると、自動的にリストアされます。ユーザ・プログラムで、他のシグナル処理を変更しないでください。
<code>int sigrtchk();</code>	非同期イベントのチェック	<p>これは、<code>open()</code>、<code>close()</code>、<code>read()</code>、<code>write()</code>、<code>ioctl()</code>、<code>pause()</code> のいずれかのシステム呼び出しが失敗した場合、またはプロセスがシグナルを受信したときの呼び出しが失敗した場合に、必ず呼び出す必要があります。これは、ユーザが次に何をすべきかを示すコードを返します。</p> <p>-1 シグナルではありません。入出力エラーを調べてください。errno 変数の内容を参照してください。</p> <p>0 他のシグナルです。割り込みが生じた時点から処理を再開してください。</p> <p>1 SIGINT/。SIGTERM “return 0” で \$ZF を終了します。システムは、これらのシグナルを適切にトラップします。</p>

デバイスを制御する通常の \$ZF 関数は、以下のように記述します。

### ObjectScript

```
IF ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
    ; Set some flags
    ; Call zferror
    ; return 0;
}
```

プロセスがシグナルを受け取ると、`open` システム呼び出しは失敗する可能性があります。通常、これはエラーではないため、処理を再開できます。しかし、シグナルによっては他の処理が必要な場合もあります。すべての可能性を考慮して、以下の C コードを記述してください。

```
sigrtclr();
WHILE (TRUE) {
    IF (sigrtchk() == 1) { return 1 or 0; }
    IF ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
        switch (sigrtchk()) {
            case -1:
                /* This is probably a real device error */
                ; Set some flags
                Call zferror
                return 0;
            case 0:
                /* A innocuous signal was received. Restart. */
                ; continue;
            case 1:
                /* Someone is trying to terminate your job. */
                Do cleanup work
                return 1 or 0;
        }
    }
    ELSE { break; }
}
/* Code to handle the normal situation: */
/* open() system call succeeded */
```

`dzfalarm()` 経由以外でシグナル・ハンドラを設定しないでください。

## エンコード・システム間で文字列を変換する

InterSystems IRIS は、\$ZF 引数タイプ、t (または T) 経由の入出力変換をサポートします。これは、以下の形式で指定できます。

引数	目的
t	現在のプロセス入出力変換オブジェクトを指定します。
t//	既定のプロセス入出力変換テーブル名を指定します。
t/name/	特定の入出力変換テーブル名を指定します。

\$ZF は、以下の C 構造にあるバイト文字列経由で、変換された文字列を外部プロシージャに移します。

```
typedef struct zarray {
    unsigned short len;
    unsigned char data[1]; /* 1 is a dummy value */
} *ZARRAYP;
```

これは、b (または B) 引数タイプに使用された構造でもあります。

以下の \$ZF サンプル関数は、可逆変換を実行します。

```
#include iris-cdzf.h
extern    int trantest();
ZFBEGIN
ZFENTRY("TRANTEST","t/SJIS/ T/SJIS/",trantest)

ZFEND

int trantest(inbuf,outbuf);

ZARRAYP inbuf;          /* Buffer containing string that was converted from
                           internal InterSystems IRIS encoding to SJIS encoding before it
                           was passed to this function */
ZARRAYP outbuf;         /* Buffer containing string in SJIS encoding that will
                           be converted back to internal InterSystems IRIS encoding before
                           it is passed back into the InterSystems IRIS environment */
{
    int i;
    /* Copy data one byte at a time from the input argument buffer
       to the output argument buffer */

    for (i = 0; i < inbuf->len; i++)
        outbuf->data[i] = inbuf->data[i];

    /* Set number of bytes of data in the output argument buffer */
    outbuf->len = inbuf->len;

    return 0; /* Return success */
}
```

**注釈** 概念的には、データは外部プロシージャがデバイスであるかのように、\$ZF 外部プロシージャへ流れ込んだり、また外部プロシージャから流れ出たりします。入出力変換の出力要素は、データが InterSystems IRIS 環境から“出ていく”ので、外部プロシージャに渡されるデータとして使用されます。入出力変換の入力要素は、データが InterSystems IRIS 環境に“入る”ので、外部プロシージャから渡されるデータとして使用されます。

入出力変換の出力要素が未定義で、アプリケーションがその入出力変換を使用して NULL 文字列以外のデータを渡そうとする場合、データの変換方法が不明であるため、InterSystems IRIS はエラーを返します。

入出力変換の入力要素が未定義で、タイプ文字列の引数が入出力変換を \$ZF 出力引数に関連付ける場合、未定義の変換を使用する出力引数は無意味なので、InterSystems IRIS はエラーを返します。

## ゼロで終了する Unicode 文字列、および Unicode 計算文字列

\$ZF 関数は、ゼロで終了する Unicode 文字列、および Unicode 計算文字列の引数タイプをサポートします。

ゼロ で終了する Unicode 文字列、および Unicode 計算文字列の引数タイプは、以下のコードを持ちます。

引数	目的
w	ゼロで終了する Unicode 文字列へのポインタ
s	Unicode 計算文字列へのポインタ

両方の引数タイプで、Unicode 文字の C データ型は unsigned short です。ゼロで終了する Unicode 文字列のポインタは、以下のように宣言されます。

```
unsigned short *p;
```

Unicode 計算文字列に対するポインタは、以下の C 構造に対するポインタとして宣言されます。

```
typedef struct zwarray {
    unsigned short len;
    unsigned short data[1]; /* 1 is a dummy value */
} *ZWARRAYP;
```

例えば以下ようになります。

```
ZWARRAYP *p;
```

len フィールドには、Unicode 文字配列の長さが格納されます。

data フィールドには、Unicode 計算文字列の文字が格納されます。Unicode 文字列の最大値は、\$ZF 文字列の最大値です。これは、更新できる構成パラメータで、既定は 32767 です。

各 Unicode 文字の長さは 2 バイトです。InterSystems IRIS は返される可能性のある最大長文字列のスペースを予約するため、出力引数として Unicode 文字列を宣言する際は、この文字の長さを考慮することが重要です。既定の文字列サイズを使用する場合、1 つの Unicode 文字列引数で使用するメモリの合計は以下のように計算します。

32767 (最大文字数) \* 2 (1 文字あたりのバイト数) = 65534 (合計バイト)

この値は、すべての \$ZF 引数に割り当てられる既定の最大メモリ領域の値 (67584) とほぼ同じ値です。この最大 \$ZF ヒープ領域は、更新可能な構成パラメータでもあります。

## エラー・メッセージ

\$ZF ヒープ領域がすべて使用されている場合、\$ZF は <OUT OF \$ZF HEAP SPACE> エラーを発行します。\$ZF 文字列スタックがすべて使用されている場合、\$ZF は <STRINGSTACK> エラーを発行します。\$ZF がバディブロックを割り当てることができない場合、<STORE> エラーを発行します。

## 関連項目

- ・ [\\$ZF\(-100\) 関数](#)
- ・ [\\$ZF\(-1\) 関数 \(非推奨\)](#)
- ・ [\\$ZF\(-2\) 関数 \(非推奨\)](#)
- ・ [\\$ZF\(-3\) 関数](#)
- ・ [\\$ZF\(-4\) 関数](#)
- ・ [\\$ZF\(-5\) 関数](#)
- ・ [\\$ZF\(-6\) 関数](#)
- ・ コールアウト SDK の使用法

## \$ZF(-1) (ObjectScript)

オペレーティング・システム・コマンドまたはプログラムを子プロセスとして同期的に実行します。(非推奨)。

### 構文

```
$ZF(-1,program)
```

### 引数

引数	説明
program	オプション - 子プロセスとして実行されるオペレーティング・システム・コマンドまたはプログラム。引用符付きの文字列として指定されます。program を省略した場合、\$ZF(-1) はオペレーティング・システム・シェルを起動します。

### 説明

注釈 \$ZF(-1) は非推奨の関数です。ここでは、既存のコードとの互換性についてのみ説明します。新しいコード開発ではすべて、[\\$ZF\(-100\)](#) を使用してください。

\$ZF(-1) は、InterSystems IRIS プロセスによるホスト・オペレーティング・システムのプログラムまたはコマンドの呼び出しを許可します。program で指定されたプログラム、またはコマンドを、現在のコンソールから生成された子プロセスとして実行します。この関数は同期的に実行されます。プロセスが返されるのを待機します。\$ZF(-1) は子プロセスの終了状態を返します。

\$ZF(-1) は以下のステータス・コードを返します。

- ・ 子プロセスの実行に成功した場合は 0 を返します。
- ・ オペレーティング・システム・シェルによって発行された終了ステータス・エラー・コードに基づいて、正の整数を返します。この終了ステータス・コード整数値は、ホスト・オペレーティング・システムによって決定されます。例えば、ほとんどの Windows コマンドの構文エラーに対して、\$ZF(-1) は 1 を返します。
- ・ 子プロセスがフォークできない場合は -1 を返します。

\$ZF(-1) は、生成された子プロセスからの応答を待機するので、子プロセスの実行中は InterSystems IRIS を正常にシャットダウンすることができません。

引数が指定されていない \$ZF(-1) は、既定のオペレーティング・システム・シェルを起動します。詳細は、“[\\$ZF\(-100\)](#) を使用したプログラムまたはシステム・コマンドの実行” を参照してください。

program で指定されたパス名にスペース文字が含まれる場合、パス名の処理はプラットフォームによって異なります。Windows および UNIX® の場合はパス名にスペース文字を使用することができますが、スペースを含むパス名は、追加の二重引用符 (") で全体を囲む必要があります。これは、Windows の cmd /c 文に従っています。詳細は、Windows のコマンド・プロンプトで cmd /? を指定してください。

**%Library.File** クラスの NormalizeFilenameWithSpaces() メソッドを使用すると、パス名内のスペースがホスト・プラットフォームに合わせて処理されるようにできます。

\$ZF(-1) は、**%System\_Callout:U** 特権を必要とします。詳細は、“[%System\\_Callout:USE 特権の追加](#)” を参照してください。

\$ZF(-1) がプロセスを生成できない場合、<FUNCTION> エラーを生成します。

ターミナルで、実行するオペレーティング・システム・コマンドの最初の文字に感嘆符 (!) またはドル記号 (\$) を使用して、\$ZF(-1) と同様の操作を実行できます。! または \$ のコマンド行接頭語によりオペレーティング・システム・コマンドが実

行され、呼び出されたプロセスからの結果が返され、ターミナルにそれらの結果が表示されます。\$ZF(-1) は、オペレーティング・システム・コマンドの結果を返さずに、オペレーティング・システム・コマンドの実行後、呼び出されたプロセスの終了ステータス・コードを返します。詳細は、“\$ZF(-100) を使用したプログラムまたはシステム・コマンドの実行” を参照してください。

## 監査

\$ZF(-1) を呼び出すたびに、OS コマンド監査レコードが監査ログに追加されます。このレコードには、以下のような情報が含まれます。

```
Execute O/S command Directory: c:\182u5\mgr\
Command: ls -lt 4002
```

## \$ZF(-1)、\$ZF(-2)、および \$ZF(-100)

これら 3 つの関数はほとんどの点で同一です。[\\$ZF\(-100\)](#) は、すべての目的に適した関数であり、\$ZF(-1) と \$ZF(-2) の両方の代わりに使用できます。

- \$ZF(-1) は OS シェルを使用して実行されます。この関数は同期的に実行されます。作成された子プロセスの完了を待つ間、現在のプロセスの実行を中断します。作成されたプロセスからステータス情報を受信し、作成されたプロセスの完了時に、これを終了ステータス・コード (整数値) として返します。\$ZF(-1) は、\$ZCHILD を設定しません。
- [\\$ZF\(-2\)](#) は OS シェルを使用して実行されます。この関数は非同期的に実行されます。現在のプロセスの実行を中断しません。子プロセスの生成において、直ちにステータスの値を返します。作成された子プロセスの完了を待たないので、そのプロセスからのステータス情報は受信できません。[\\$ZF\(-2\)](#) は、5 番目の引数が True の場合、\$ZCHILD を設定します。
- [\\$ZF\(-100\)](#) は、同期的または非同期的に実行できます。オペレーティング・システム・シェルを使用して実行することも、シェルを使用せずに実行することもできます。常に \$ZCHILD を設定します。[\\$ZF\(-1\)](#) と [\\$ZF\(-2\)](#) はどちらも、引数の指定なしで、オペレーティング・システム・シェルを起動します。[\\$ZF\(-100\)](#) では、オペレーティング・システム・シェルを起動するために program 引数 (および /SHELL フラグ) が必要です。

## 例

以下の Windows の例は、ユーザが記述したプログラムを実行します。この場合、.txt ファイルのコンテンツが表示されます。また、\$ZF(-1) のパス名の処理に NormalizeFilenameWithSpaces() を使用しています。スペースを含むパス名がホスト・プラットフォームに合わせて処理されます。スペースを含まないパス名は、変更されずに渡されます。[\\$ZF\(-1\)](#) は、指定されたファイルにアクセスできた場合は、Windows シェルの終了状態 0 を返します。アクセスに失敗した場合は、1 を返します。

### ObjectScript

```
SET fname="C:\My Test.txt"
WRITE fname,!
SET x=##class(%Library.File).NormalizeFilenameWithSpaces(fname)
WRITE x,!
WRITE $ZF(-1,x)
```

以下の Windows の例では、Windows のオペレーティング・システム SOL コマンドを呼び出します。SOL は、Windows オペレーティング・システムで提供されているソリティア・ゲームを表示するウィンドウを開きます。ソリティアのインタラクティブ・ウィンドウを閉じると、\$ZF(-1) は Windows シェルの終了状態 0 を返します。これは、処理が成功したことを示します。

### ObjectScript

```
SET x=$ZF(-1,"SOL")
WRITE x
```



以下の Windows の例では、存在しない Windows のオペレーティング・システム・コマンドを呼び出します。\$ZF(-1) は、Windows シェルの終了ステータス 1 を返します。これは、構文エラーを示します。

#### ObjectScript

```
WRITE $ZF(-1,"SOX")
```

以下の Windows の例では、Windows のオペレーティング・システム・コマンドを呼び出して、存在しないネットワーク名を指定します。\$ZF(-1) は、Windows シェルの終了エラー 2 を返します。

#### ObjectScript

```
WRITE $ZF(-1,"NET USE :k \\bogusname")
```

## 関連項目

- ・ [\\$ZF\(-2\) 関数 \(非推奨\)](#)
- ・ [\\$ZF\(-100\) 関数](#)
- ・ [\\$ZF\(-100\) を使用したプログラムまたはシステム・コマンドの実行](#)

## \$ZF(-2) (ObjectScript)

オペレーティング・システム・コマンドまたはプログラムを子プロセスとして非同期的に実行します。(非推奨)

### 構文

```
$ZF(-2,program)
```

### 引数

引数	説明
program	オプション - 子プロセスとして実行されるオペレーティング・システム・コマンドまたはプログラム。引用符付きの文字列として指定されます。program を省略した場合、\$ZF(-2) はオペレーティング・システム・シェルを起動します。

### 概要

**注釈** \$ZF(-2) は非推奨の関数です。ここでは、既存のコードとの互換性についてのみ説明します。新しいコード開発ではすべて、[\\$ZF\(-100\)](#) を使用してください。

\$ZF(-2) は、InterSystems IRIS プロセスによるホスト・オペレーティング・システムのプログラムまたはコマンドの呼び出しを許可します。\$ZF(-2) は、program で指定されたオペレーティング・システム・コマンドを、現在のコンソールから生成された子プロセスとして実行します。この関数は非同期的に実行されます。子プロセスを生成したら直ちに返し、プロセスの終了を待機しません。入力と出力のデバイスは、NULL デバイスが既定になります。

\$ZF(-2) は子プロセスの終了状態を返しません。その代わりに、子プロセスが正常に作成された場合、\$ZF(-2) は 0 を返します。子プロセスがフォークできない場合は \$ZF(-2) は -1 を返します。

\$ZF(-2) は、生成された子プロセスからの応答を待機しないので、子プロセスの実行中でも InterSystems IRIS を正常にシャットダウンすることができます。

\$ZF(-2) は、オペレーティング・システム・コマンドを実行する前に親プロセスの主デバイス (\$PRINCIPAL で指定) を終了します。これは、子プロセスが親プロセスと同時に実行されるためです。\$ZF(-2) が \$PRINCIPAL を終了しなかった場合は、親と子からの出力が混ざり合います。\$ZF(-2) の使用時に、子プロセスからの出力を回復させる場合は、コマンドで入出力をリダイレクトします。以下はその例です。

#### ObjectScript

```
SET x=$ZF(-2,"ls -l > mydir.txt")
```

引数が指定されていない \$ZF(-2) は、既定のオペレーティング・システム・シェルを起動します。詳細は、“[\\$ZF\(-100\)](#) を使用したプログラムまたはシステム・コマンドの実行” を参照してください。

program で指定されたパス名にスペース文字が含まれる場合、パス名の処理はプラットフォームによって異なります。Windows および UNIX® の場合はパス名にスペース文字を使用することができますが、スペースを含むパス名は、追加の二重引用符 (") で全体を囲む必要があります。これは、Windows の cmd /c 文に従っています。詳細は、Windows のコマンド・プロンプトで cmd /? を指定してください。

%Library.File クラスの NormalizeFilenameWithSpaces() メソッドを使用すると、パス名内のスペースがホスト・プラットフォームに合わせて処理されるようにできます。

\$ZF(-2) は、%System\_Callout:U 特権を必要とする操作です。詳細は、“%System\_Callout:USE 特権の追加” を参照してください。

## 監査

\$ZF(-2)を呼び出すたびに、OS コマンド監査レコードが監査ログに追加されます。このレコードには、以下のような情報が含まれます。

```
Execute O/S command Directory: c:\182u5\mgr\  
Command: ls -lt 4002 - Detached
```

Detached キーワードは、呼び出しが \$ZF(-2) であることを示します。\$ZF(-1) 呼び出しは、このキーワードを取りません。

## \$ZF(-2)、\$ZF(-1)、および \$ZF(-100)

これら 3 つの関数はほとんどの点で同一です。[\\$ZF\(-100\)](#) は、すべての目的に適した関数であり、\$ZF(-1) と \$ZF(-2) の両方の代わりに使用できます。

- ・ \$ZF(-2) は OS シェルを使用して実行されます。この関数は非同期的に実行されます。現在のプロセスの実行を中断しません。子プロセスの生成において、直ちにステータスの値を返します。作成された子プロセスの完了を待たないので、そのプロセスからのステータス情報は受信できません。\$ZF(-2) は、5 番目の引数が True の場合、\$ZCHILD を設定します。
- ・ [\\$ZF\(-1\)](#) は OS シェルを使用して実行されます。この関数は同期的に実行されます。作成された子プロセスの完了を待つ間、現在のプロセスの実行を中断します。作成されたプロセスからステータス情報を受信し、作成されたプロセスの完了時に、これを終了ステータス・コード (整数値) として返します。\$ZF(-1) は、\$ZCHILD を設定しません。
- ・ [\\$ZF\(-100\)](#) は、同期的または非同期的に実行できます。オペレーティング・システム・シェルを使用して実行することも、シェルを使用せずに実行することもできます。常に \$ZCHILD を設定します。\$ZF(-1) と \$ZF(-2) はどちらも、引数の指定なしで、オペレーティング・システム・シェルを起動します。\$ZF(-100) では、オペレーティング・システム・シェルを起動するために program 引数 (および /SHELL フラグ) が必要です。

## 関連項目

- ・ [\\$ZF\(-1\)](#) 関数 (非推奨)
- ・ [\\$ZF\(-100\)](#) 関数
- ・ [\\$PRINCIPAL](#) 特殊変数
- ・ \$ZF(-100) を使用したプログラムまたはシステム・コマンドの実行
- ・ %System\_Callout:USE 特権の追加

## \$ZF(-3) (ObjectScript)

DLL (ダイナミック・リンク・ライブラリ) をロードし、ライブラリ関数を実行します。この関数は、コールアウト SDK のコンポーネントです。

### 構文

```
$ZF(-3, dll_name, func_name, args)
```

### 引数

引数	説明
dll_name	ロードする DLL (ダイナミック・リンク・ライブラリ) の名前。引用符付き文字列として指定されます。DLL が既にロードされている場合は、dll_name を NULL 文字列 ("" ) として指定できます。
func_name	オプション - DLL 内で実行される関数名。引用符付き文字列として指定されます。
args	オプション - 関数に渡す引数の、コンマ区切りのリスト。

### 概要

\$ZF(-3) を使用して、DLL (ダイナミック・リンク・ライブラリ) をロードし、指定された関数をその DLL から実行することができます。\$ZF(-3) は、関数の返り値を返します。

\$ZF(-3) は、以下の方法のいずれかで呼び出されます。

DLL をロードするには、以下の構文を使用します。

#### ObjectScript

```
SET x=$ZF(-3, "mydll")
```

DLL をロードし、その DLL にある関数を実行するには、以下の構文を使用します。

#### ObjectScript

```
SET x=$ZF(-3, "mydll", "$myfunc", 1)
```

\$ZF(-3) を使用して DLL をロードすると、その DLL が現在の DLL となり、以前に \$ZF(-3) を呼び出したときにロードされた DLL を自動的にアンロードします。

以前の \$ZF(-3) でロードされた DLL にある関数を実行するには、以下のように現在の DLL を NULL 文字列で指定することで、前の関数の実行速度を上げることができます。

#### ObjectScript

```
SET x=$ZF(-3, "", "$myfunc2", 1)
```

(以前の \$ZF(-3) 呼び出しでロードされた) 現在の DLL を明示的にアンロードするには、以下の構文を使用します。

#### ObjectScript

```
SET x=$ZF(-3, "")
```

\$ZF(-3) は、1 つの DLL のみロードします。DLL をロードすると、以前の DLL がアンロードされます。また、現在ロードされている DLL を明示的にアンロードすることもできます。これにより、現在ロードされている DLL が存在しないことに

なります。(ただし、\$ZF(-3) のロードやアンロードが、\$ZF(-5) や \$ZF(-6) で使用するロードやアンロードに影響を与えることはありません。詳細は以下に説明しています。)

指定された DLL 名はフル・パス名、またはパス名の一部分です。パス名の一部分を指定する場合、InterSystems IRIS はそれを現在のディレクトリに正規化します。一般的に、DLL はバイナリ・ディレクトリ ("bin") に格納されます。バイナリ・ディレクトリの位置を確認するには、%SYSTEM.Util クラスの BinaryDirectory() メソッドを呼び出します。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。

## ダイナミック・リンク・ライブラリ

DLL は、実行時に呼び出されたり、ロードされるルーチンを含むバイナリ・ライブラリです。DLL がロードされるとき、InterSystems IRIS はその内部で GetZFTable() という名前の関数を検索します。GetZFTable() が存在する場合、DLL にある関数のテーブルに対するポインタを返します。このテーブルを使用して、\$ZF(-3) は指定された関数を DLL から呼び出します。

### 複数の DLL のロード

\$ZF(-3) を呼び出すと、1 回に 1 つの DLL をロードします。DLL をロードすると、以前の DLL はアンロードされます。複数の DLL を同時にロードするには、\$ZF(-5) または \$ZF(-6) を使用して DLL 関数を実行します。\$ZF(-3) を使用した DLL のロードやアンロードは、\$ZF(-5) や \$ZF(-6) で使用するためにロードされた DLL に影響を与えません。

### 別の DLL に依存する DLL のロード

Windows では、bin ディレクトリにインストールされる一部の InterSystems IRIS システム DLL は、bin ディレクトリ内の別の DLL に依存しています。Windows の検索規則では、bin をそのプロセスの PATH に追加しない限り、bin ディレクトリ内の依存関係は検索されません。\$ZF(-3) から、プロセスの PATH を使用して DLL の依存関係を解決できない場合、InterSystems IRIS は <DYNAMIC LIBRARY LOAD> エラーを発行します。

ただし、依存 DLL が [\\$ZF\(-4\)](#) を使用してロードされている場合、InterSystems IRIS は、最初に DLL のロード元のディレクトリで依存 DLL を検索します。InterSystems IRIS システムはこのために、Windows のロード操作を使用して、DLL のロード時に開始ディレクトリを一時的に PATH に追加します。\$ZF(-4) によってロードされたら、PATH を変更することなく \$ZF(-3) でこの依存 DLL を使用できます。

## 関連項目

- ・ [\\$ZF\(-5\) 関数](#)
- ・ [\\$ZF\(-6\) 関数](#)
- ・ 単純なライブラリ関数呼び出しでの \$ZF(-3) の使用

## \$ZF(-4) (ObjectScript)

\$ZF(-5) および \$ZF(-6) で使用されるユーティリティ関数を提供します。この関数は、コールアウト SDK のコンポーネントです。

### 構文

```
$ZF(-4,1,dll_name)
```

```
$ZF(-4,n,dll_id,func_name)
```

```
$ZF(-4,n,dll_id,decr_flag)
```

```
$ZF(-4,n,dll_index,dll_name)
```

```
$ZF(-4,n,dll_index,decr_flag)
```

### 引数

引数	説明
n	実行する操作のタイプを示すコード。1= 名前によって DLL をロードします。2= ID によって DLL をアンロードします。3= ID によって DLL の関数を検索します。4= インデックスによって DLL をアンロードします。5= システム DLL インデックス・テーブルにエントリを作成します。6= システム DLL インデックス・テーブルからエントリを削除します。7= プロセス DLL インデックス・テーブルにエントリを作成します。8= プロセス DLL インデックス・テーブルからエントリを削除します。
dll_name	ダイナミック・リンク・ライブラリ (DLL) の名前。n=1、5、または 7 で使用します。
dll_id	ロードされているダイナミック・リンク・ライブラリ (DLL) の ID 値。n=2 または 3 で使用します。
dll_index	DLL インデックス・テーブル内のダイナミック・リンク・ライブラリ (DLL) に対するユーザ定義のインデックス。一意でゼロ以外の正の整数にする必要があります。1024 から 2047 までの数値は、システムでの使用に予約されています。n=4、5、6、7、または 8 と一緒に使用します。
func_name	DLL 内で検索する関数の名前。n=3 の場合のみ使用します。
decr_flag	オプション - DLL 参照カウントをデクリメントするためのフラグ。n=2 または 4 で使用します。

### 概要

\$ZF(-4) は、DLL または DLL 内の関数の ID 値の確立に使用できます。この ID 値は、\$ZF(-5) が関数を実行する際に使用されます。

\$ZF(-4) は、DLL インデックス・テーブルに対するインデックスの確立に使用できます。このインデックス値は、\$ZF(-6) が関数を実行する際に使用されます。

- ・ \$ZF(-4,1) を使用することで、共有ライブラリを明示的にロードできます。これにより、ライブラリがロードされ、\$ZF(-5) でライブラリ関数にアクセスするために使用できるハンドルが返されます。
- ・ \$ZF(-3) を使用することで、単一の共有ライブラリを明示的にロードできます。これにより、単一のアクティブ・ライブラリがロードされ、そのメソッドが呼び出されます。
- ・ \$ZF(-4,5) または \$ZF(-4,7) でライブラリにインデックスを付けた後、\$ZF(-6) を使用して共有ライブラリを暗黙的にロードできます。

## ID 値の確立

DLL をロードしその ID を返すには、次の構文を使用します。

```
dll_id=$ZF(-4,1,dll_name)
```

\$ZF(-4,1) によってロードされた DLL から関数を検索し、その関数に対する ID を返すには、次の構文を使用します。

```
func_id=$ZF(-4,3,dll_id,func_name)
```

\$ZF(-4,3) によって配置された関数を実行するには、\$ZF(-5) を使用します。

\$ZF(-4,1) によってロードされた特定の DLL をアンロードするには、次の構文を使用します。

```
$ZF(-4,2,dll_id)
```

\$ZF(-4,1) によってロードされたすべての DLL をアンロードするには、次の構文を使用します。

```
$ZF(-4,2)
```

## DLL ロードのインクリメントおよびデクリメント

2 つのクラスで同じライブラリをロードした場合、そのライブラリは、\$ZF(-4,2,dll\_id) または \$ZF(-4,4,dll\_index) の最初の呼び出しによってアンロードされます。これにより、ライブラリにアクセスせずにもう片方のクラスが立ち往生する場合があります。このため、InterSystems IRIS は、各 DLL の参照カウントをサポートします。InterSystems IRIS は、\$ZF(-4,1,dll\_name) でライブラリがロードされた回数の参照カウントを保持します。\$ZF(-4,1,dll\_name) を呼び出すごとに参照カウントが増加します。

\$ZF(-4,2) は、オプションのデクリメント・フラグ引数 `decr_flag` を提供します。\$ZF(-4,2,dll\_id,1) を呼び出すごとに参照カウントが 1 ずつ減少します。\$ZF(-4,2,dll\_id,1) の呼び出しにより、参照カウントがゼロになった場合にライブラリがアンロードされます。\$ZF(-4,2,dll\_id) (または \$ZF(-4,2,dll\_id,0)) を呼び出すと、参照カウントは無視され、直ちにライブラリがアンロードされます。

\$ZF(-4,5) または \$ZF(-4,7) を呼び出すと、ライブラリ・インデックスが作成されます。その後、関数を実行するために \$ZF(-6) を呼び出すと、ライブラリが暗黙的にロードされ、参照カウントが増加します。\$ZF(-4,4,dll\_index,1) を呼び出すごとにこの参照カウントが 1 ずつ減少します。

`dll_name` と `dll_index` によって確立された参照カウント間の参照カウントの相互作用は以下のとおりです。

- ・ 参照カウントがゼロにならない限り、\$ZF(-4,1,dll\_name) でロードされたライブラリは、\$ZF(-4,4,dll\_index,1) の呼び出しによってアンロードされません。
- ・ 参照カウントに関係なく、\$ZF(-4,1,dll\_name) でロードされたライブラリは、\$ZF(-4,2,dll\_id) または \$ZF(-4,4,dll\_index) (デクリメント・フラグ引数なし) のいずれかによって直ちにアンロードされます。
- ・ 参照カウントがゼロになった場合でも、\$ZF(-6) で暗黙的にロードされたライブラリは、\$ZF(-4,2,dll\_id,1) によってアンロードされません。\$ZF(-4,4,dll\_index,1) によってのみアンロード可能です。
- ・ 参照カウントに関係なく、\$ZF(-6) で暗黙的にロードされたライブラリは、\$ZF(-4,2,dll\_id) または \$ZF(-4,4,dll\_index) (デクリメント・フラグ引数なし) のいずれかによって直ちにアンロードされます。

参照カウントに関係なく、また \$ZF(-4,1,dll\_name) でロードされたのか \$ZF(-6) で暗黙的にロードされたのかに関係なく、`dll_id` 引数のない \$ZF(-4,2) は、すべてのライブラリを直ちにアンロードします。

## 別の DLL に依存する DLL のロード

Windows では、`bin` ディレクトリにインストールされる一部のシステム DLL は、`bin` ディレクトリ内の別の DLL に依存しています。Windows の検索規則では、`bin` をそのプロセスの `PATH` に追加しない限り、`bin` ディレクトリ内の依存関係は検索されません。ただし、これらの DLL の 1 つが \$ZF(-4) または \$ZF(-6) を使用して呼び出された場合、InterSystems IRIS は、最初に DLL のロード元のディレクトリで依存 DLL を検索します。その場所で依存 DLL が見つからなかった場



合は、既定の検索 PATH が使用されます。InterSystems IRIS はこのために、Windows のロード操作を使用して、DLL のロード時に開始ディレクトリを一時的に PATH に追加します。この一時的な PATH の追加は、DLL が \$ZF(-4) または \$ZF(-6) によってロードされている場合に使用されます。この一時的な PATH の追加は、DLL が \$ZF(-3) によってロードされている場合には使用されません。

DLL の依存関係を解決できない場合、InterSystems IRIS は <DYNAMIC LIBRARY LOAD> エラーを発行します。

## インデックス値の確立

システム DLL インデックス・テーブルの DLL をインデックスするには、次の構文を使用します。

```
$ZF(-4,5,dll_index,dll_name)
```

プロセス DLL インデックス・テーブルの DLL をインデックスするには、次の構文を使用します。

```
$ZF(-4,7,dll_index,dll_name)
```

\$ZF(-4,5) または \$ZF(-4,7) によってインデックスされた関数を検索し、実行するには、\$ZF(-6) を使用します。

インデックス付きの DLL をアンロードするには、次の構文を使用します。

```
$ZF(-4,4,dll_index)
```

システム DLL インデックス・テーブルのインデックス・エントリを 1 つ削除するには、次の構文を使用します。

```
$ZF(-4,6,dll_index)
```

プロセス DLL インデックス・テーブルのインデックス・エントリを 1 つ削除するには、次の構文を使用します。

```
$ZF(-4,8,dll_index)
```

プロセス DLL インデックス・テーブルのすべてのインデックス・エントリを削除するには、次の構文を使用します。

```
$ZF(-4,8)
```

\$ZF(-4) と \$ZF(-5) の使用法の詳細は、“\$ZF(-5) を使用したシステム ライブラリへのアクセス” を参照してください。

\$ZF(-4) と \$ZF(-6) の使用法の詳細は、“ユーザ・インデックスによるライブラリへのアクセスでの \$ZF(-6) の使用” を参照してください。

## 関連項目

- ・ [\\$ZF\(-3\) 関数](#)
- ・ [\\$ZF\(-5\) 関数](#)
- ・ [\\$ZF\(-6\) 関数](#)
- ・ システム ID によるライブラリへのアクセスでの \$ZF(-5) の使用
- ・ ユーザ・インデックスによるライブラリへのアクセスでの \$ZF(-6) の使用

## \$ZF(-5) (ObjectScript)

\$ZF(-4) を使用してロードされた DLL 関数を実行します。この関数は、コールアウト SDK のコンポーネントです。

### 構文

```
$ZF(-5,dll_id,func_id,args)
```

### 引数

引数	説明
dll_id	\$ZF(-4) により提供されたダイナミック・リンク・ライブラリ (DLL) の ID 値
func_id	\$ZF(-4) により提供された DLL 内の関数の ID 値
args	オプション - 呼び出された関数に渡された 1 つ以上の引数

### 概要

\$ZF(-4) によってロードされた DLL 内の関数を実行するには、以下の構文を使用します。

```
return=$ZF(-5,dll_id,func_id,args)
```

### 関連項目

- ・ [\\$ZF\(-4\) 関数](#)
- ・ システム ID によるライブラリへのアクセスでの \$ZF(-5) の使用

## \$ZF(-6) (ObjectScript)

\$ZF(-4) を使用してインデックス作成された DLL 関数を実行します。この関数は、コールアウト SDK のコンポーネントです。

### 構文

```
$ZF(-6, dll_index, func_id, args)
```

### 引数

引数	説明
dll_index	\$ZF(-4) から発生した、DLL インデックス・テーブルの DLL ファイル名に対するユーザ指定のインデックス。
func_id	オプション - \$ZF(-4) から提供された、DLL 内の関数の ID 値。これを省略した場合は、呼び出しによって dll_index の妥当性を検証し、イメージをロードして、イメージの位置を返します。
args	オプション - 関数に渡す引数で、存在する場合はコンマで区切られたリストの形式で指定します。

### 概要

\$ZF(-6) は、DLL ファイル名に対するユーザ定義のインデックスを使用して、ダイナミック・リンク・ライブラリ (DLL) 関数の高速インタフェースを提供します。整数 (dll\_index) を割り当て、dll\_name と一意に関連付けることによって、\$ZF(-4) でこのユーザ定義のインデックスを確立します。このエントリは、プロセス DLL インデックス・テーブルか、システム DLL インデックス・テーブルに配置できます。

\$ZF(-5) と \$ZF(-6) はいずれも、\$ZF(-4) によって位置を確認された DLL からの関数の実行に使用できます。

\$ZF(-6) の使用法の詳細は、“ユーザ・インデックスによるライブラリへのアクセスでの \$ZF(-6) の使用” を参照してください。

### 別の DLL に依存する DLL のロード

Windows では、bin ディレクトリにインストールされる一部のシステム DLL は、bin ディレクトリ内の別の DLL に依存しています。Windows の検索規則では、bin をそのプロセスの PATH に追加しない限り、bin ディレクトリ内の依存関係は検索されません。ただし、これらの DLL の 1 つが \$ZF(-4) または \$ZF(-6) を使用して呼び出された場合、InterSystems IRIS は、最初に DLL のロード元のディレクトリで依存 DLL を検索します。その場所で依存 DLL が見つからなかった場合は、既定の検索 PATH が使用されます。InterSystems IRIS はこのために、Windows のロード操作を使用して、DLL のロード時に開始ディレクトリを一時的に PATH に追加します。この一時的な PATH の追加は、DLL が \$ZF(-4) または \$ZF(-6) によってロードされている場合に使用されます。この一時的な PATH の追加は、DLL が \$ZF(-3) によってロードされている場合には使用されません。

DLL の依存関係を解決できない場合、InterSystems IRIS は <DYNAMIC LIBRARY LOAD> エラーを発行します。

### 関連項目

- ・ [\\$ZF\(-3\) 関数](#)
- ・ [\\$ZF\(-4\) 関数](#)
- ・ [\\$ZF\(-5\) 関数](#)
- ・ ユーザ・インデックスによるライブラリへのアクセスでの \$ZF(-6) の使用

## \$ZF(-100) (ObjectScript)

オペレーティング・システム・コマンドまたはプログラムを子プロセスとして実行します。この関数は、コールアウト SDK のコンポーネントです。

### 構文

```
$ZF(-100, flags, program, args)
```

### 引数

引数	説明
flags	1 つ以上のキーワード・フラグを含む引用符付きの文字列。複数のキーワード・フラグは空白で区切られます。キーワード・フラグは、/keyword、/keyword=value、または /keyword+=value という形式を取ることができます。キーワードでは、大文字と小文字は区別されません。flags は、program の実行方法を指定します。
program	子プロセスとして実行されるオペレーティング・システム・コマンドまたはプログラム。引用符付きの文字列として指定されます。パスの指定はフル・パスでもプログラム名でもかまいません。オペレーティング・システムはその規則 (PATH 環境変数など) を使用して、指定されたプログラムを検索します。
args	オプション - program のオプションと引数のコンマ区切りのリスト。null 引数を "" として指定できます。ローカル配列と間接指定の .args または args... 構文を使用して、 <a href="#">可変個数の引数</a> を指定できます。

### 説明

\$ZF(-100) は、InterSystems IRIS プロセスによるホスト・オペレーティング・システムの実行可能プログラムまたはコマンドの呼び出しを許可します。例として、ファイルを列挙してコピーする Windows® コマンドの 2 つのサンプルを示します。

```
set listStatus = $zf(-100, "/SHELL", "dir", "/q")
set copyStatus = $zf(-100, "/SHELL", "copy", "myfile.txt", "c:\InterSystems")
```

\$ZF(-100) は、program で指定されたプログラムまたはコマンドを、現在のコンソールから生成された子プロセスとして実行します。オペレーティング・システムのシェルを呼び出しても呼び出さなくても、プログラムまたはコマンドを同期または非同期で呼び出すことができます。\$ZF(-100) は、\$ZF(-1) および \$ZF(-2) と類似の機能を提供します。\$ZF(-1) または \$ZF(-2) (どちらも非推奨の関数) よりも、この関数を使用することをお勧めします。

次の Unix® の例に示すように、ローカル配列と間接指定を使用して、可変個数の args を指定することができます：

```
SET args=2
SET args(1)="-01"
SET args(2)="myfile.c"
SET status = $ZF(-100, "/ASYNCR", "gcc", .args)
```

\$ZF(-100) は、[\\$ZCHILD](#) を起動プログラムの PID に設定します。

\$ZF(-100) は、[DO](#) コマンドの引数として実行できます。DO \$ZF(-100) は、\$ZF(-100) を関数として呼び出す場合と以下の 2 つの点で異なります。

- DO は、[返される整数ステータス・コード](#)を無視します。
- \$ZF(-100) に、引数の[後置条件式](#)を追加できます。例えば、DO: x \$ZF(-100, "", "gcc", .args): y \$ZF(-100, "/ASYNCR", "gcc", .args): z では、3 つの後置条件を指定しています。これは、x=0 であれば DO コマンドを実行せず、y=0 であれば "gcc" を同期的に実行せず、z=0 であれば "gcc" を非同期的に実行しません。後置条件式によって実行は阻止されますが、引数の評価は阻止されません。

## キーワード・フラグ

\$ZF(-100) の実行方法は、flags 文字列の値によって異なります。複数のキーワード・フラグは空白で区切られます。

- ・ /ASYNC:program を非同期的に実行します。完了するまで待機しません。既定では、同期的に実行します。  
/ASYNC が指定されておらず、かつ /STDIN、/STDOUT、または /STDERR が指定されていない場合、InterSystems IRIS は、これらのファイルに対してオペレーティング・システムの現在の記述子または標準ハンドルを使用しようとしています。
- ・ /ENV=environmentvars: 新しいプロセスで設定する環境変数を指定します。値を指定するには、以下の 2 つの方法があります。
  - 明示的に指定。形式は /ENV=(name:value) です。コンマで区切られた複数の名前と値のペアが存在する場合があります。
  - 多次元配列を介して指定。ここで、添え字は環境変数名で、値は環境変数の値です。

フラグ引数全体が文字列であるため、名前と値は常に変数名ではなく、文字列として扱われます。これらに、コロンやコンマなど、残りの文字列の解析を妨げる可能性のある文字が含まれる場合は、これらを引用符 (フラグ文字列内にあるため、二重引用符とする必要があります) で囲む必要があります。

次の例は、変数 MYNAME を "Tom"、MYARG を "comma" に設定してコマンドを実行します。

### ObjectScript

```
do $ZF(-100, "/ENV=(MYNAME:Tom,MYARG: \"comma\", \" \"), command)
```

次の例では、先ほどの例と同じ結果が生成されます。

### ObjectScript

```
set arr("MYNAME")="Tom"
set arr("MYARG")="comma, "
do $ZF(-100, "/ENV=arr...", command)
```

この例では、2 つの環境変数が示されていますが、いくつでもかまいません。明示的リストは括弧で囲む必要があります。

- ・ /SHELL:シェルを使用して program を実行します。既定では、シェルは使用されません。
- ・ /STDIN=filename:入出力ダイレクト入力ファイル。
- ・ /STDOUT=filename:入出力ダイレクト標準データ出力ファイル。filename が存在しない場合は、作成されます。filename が存在する場合、/STDOUT=filename は既存のデータを上書きします。/STDOUT+=filename は既存のデータに追加します。

このファイルは mgr ディレクトリにある必要があることに注意してください。完全なパス名は指定できません。ファイルの権限は、\$ZF(-100) を呼び出したプロセスに基づきます。

- ・ //STDERR=filename:入出力ダイレクト標準エラー出力ファイル。filename が存在しない場合は、作成されます。filename が存在する場合、/STDERR=filename は既存のエラー・ログ・データを上書きします。/STDERR+=filename は既存のエラー・ログ・データに追加します。

このファイルは mgr ディレクトリにある必要があることに注意してください。完全なパス名は指定できません。ファイルの権限は、\$ZF(-100) を呼び出したプロセスに基づきます。

また、/STDOUT と /STDERR に同じファイルを指定した場合、両方のタイプのデータがそのファイルに書き込まれます。

- ・ /LOGCMD: 結果として得られるコマンド行を **messages.log** に記録します。複雑なコマンドの引数を正しく理解することが難しい場合があります。このような場合、このキーワード・フラグを使用すると、開発者はコマンドに渡された引数が正しく設定されているかどうかを確認することができます(特に引用に関して)。ログ機能では、引用符やその他の区切り文字は追加されません。**messages.log** エントリは 1000 文字で切り捨てられます。
- ・ /NOQUOTE: コマンド、コマンド引数、またはファイル名の自動引用を禁止します。既定で \$ZF(-100) は、自動引用、およびほとんどのユーザ指定値に適しているパス内のスペースのエスケープ処理を提供します。必要に応じて、/NOQUOTE を指定することでこの既定をオーバーライドできます。そうすれば、ユーザが適切な引用符を自身で指定できます。["ユーザ指定値の引用"](#) を参照してください。

キーワード・フラグなしで \$ZF(-100) を指定するには、この引数に空の文字列を指定します。

```
SET status = $ZF(-100,"", "ls", "-l")
```

## 入出力ダイレクト

/STDIN=filename、/STDOUT=filename、および /STDERR=filename の入出力ダイレクトでは、UNIX® の規約に従います。UNIX と Windows の両方のシステムで、以下ようになります。

- ・ /STDIN=filename: このファイル名のファイルは、指定した cmd 文字列を実行するプロセスに指定されている stdin ファイル・ハンドルにリンクされます。
- ・ /STDOUT=filename: このファイル名のファイルがまだ存在しない場合は、作成されます。既存のファイルの場合、/STDOUT=filename はファイルをゼロ・サイズに切り捨てます。/STDOUT+=filename は既存のファイルのデータに追加します。このファイルは、指定した cmd 文字列を実行するプロセスに指定されている stdout ハンドルにリンクされます。これにより、生成されたコマンドの stdout 出力が含まれる新しいファイルが作成されます。
- ・ /STDERR=filename: このファイル名のファイルがまだ存在しない場合は、作成されます。既存のファイルの場合、/STDERR=filename はファイルをゼロ・サイズに切り捨てます。/STDERR+=filename は既存のファイルのデータに追加します。このファイルは、指定した cmd 文字列を実行するプロセスに指定されている stderr ハンドルにリンクされます。これにより、生成されたコマンドの stderr 出力が含まれる新しいファイルが作成されます。

/STDIN、/STDOUT、または /STDERR が指定されていない場合は、以下ようになります。

- ・ /ASYNC が指定されている場合、未指定のファイルの代わりに NULL デバイスを使用されます。NULL デバイスを参照するハンドルが、指定した cmd 文字列を実行するプロセスに、未指定ファイルのハンドルとして指定されます。
- ・ /ASYNC を指定していない場合、\$ZF(-100) 関数を実行する InterSystems IRIS ジョブで使用されたハンドルがコピーされ、指定した cmd 文字列を実行するプロセスに未指定ファイルのハンドルとして渡されます。

注釈 Windows システムでは、/ASYNC および /STDIN flags を絶対に省略しないでください。

作成できないファイルや開くことができないファイルを /STDIN、/STDOUT、または /STDERR で指定した場合、そのファイルの代わりに NULL デバイスを使用されます。

/STDOUT=filename および /STDERR=filename (または /STDOUT+=filename および /STDERR+=filename) が同じファイル名を指定した場合、指定されたファイルは、一度だけ開かれる、または作成されます。結果としてファイル・ハンドルが複製され、指定した cmd 文字列を実行するプロセスに stdout ファイル・ハンドルと stderr ファイル・ハンドルの両方として指定されます。\$ZF(-100) は、/STDOUT および /STDERR に同じファイルを指定した場合に、一方で +=filename を指定し、もう一方で =filename を指定すると、<ILLEGAL VALUE> エラーを生成します。

## ユーザ指定値の引用

既定では、\$ZF(-100) は、コマンドと、コマンドの引数に自動引用機能を提供します。これにより、実行可能ファイルがあるディレクトリの名前にスペースが含まれる場合や、コマンド引数で出力対象として指定されたファイルの名前にスパー



が含まれる場合、空白スペースが自動的に処理されます。`$ZF(-100)` は、必要に応じて、区切り二重引用符文字を提供します。この動作を以下の例に示します。

### ObjectScript

```
DO $ZF(-100,"/LOGCMD","c:\sdelete64.exe","-nobanner","c:\dir1\nested directory\deleteme\")
```

この例では、以下が **messages.log** に記録されます。`$ZF(-100)` は、最後の引数を引用符で囲んで、ファイル・パス内のスペースをエスケープしています。

```
06/14/18-14:25:05:988 (3788) 0 $ZF(-100) cmd=c:\sdelete64.exe -nobanner "c:\dir1\nested directory\deleteme\"
06/14/18-14:25:06:020 (3788) 0 $ZF(-100) ret=0
```

自動引用によってエスケープしたい内容が正しくエスケープされない場合は、自動引用を抑制する `/NOQUOTE` フラグを使用し、必要に応じて独自に引用を行います。指定した値に / 文字または空白が含まれている場合、二重引用符でその値を囲む必要があります。これを以下の例で示しています。

### ObjectScript

```
DO $ZF(-100,"/NOQUOTE /LOGCMD","c:\sdelete64.exe","""-nobanner""","""c:\dir2\""")
```

この例では、以下が **messages.log** に記録されます。

```
06/15/18-09:27:38:619 (3788) 0 $ZF(-100) cmd=c:\sdelete64.exe "-nobanner" "c:\dir2\"
06/15/18-09:27:38:650 (3788) 0 $ZF(-100) ret=0
```

この動作は、UNIX® システムと Windows システムで異なります。

- Windows システムでは、`/SHELL` が指定されていないと、コマンド行が作成されて渡されます。この場合、いくつかの引数を引用符で囲むことが必要になる場合があります。
- どのシステムでも、`/SHELL` が指定されると、コマンドラインが作成されて渡されます。この場合、いくつかの引数を引用符で囲むことが必要になる場合があります。

コマンドまたはコマンド引数内で見つかった二重引用符はオペレーティング・システムに応じてエスケープされます。

## 返されるステータス・コード

`$ZF(-100)` は、以下のステータス・コードを返します。

- 0: 子プロセスが正常に非同期的に起動された場合 (`/ASYNCR` フラグを指定)。program 実行のステータスが不明です。
- 1: 子プロセスをフォークできない場合。
- 同期的に起動された場合は整数 (`/ASYNCR` フラグを指定しない)。この終了ステータス・コード整数値は、ホスト・オペレーティング・システムで呼び出されたアプリケーションによって決定されます。通常は正の整数ですが、負の整数を返すアプリケーションもあります。例えば、ほとんどの Windows コマンドの構文エラーに対して、`$ZF(-100)` は 1 を返します。

`/SHELL` 引数が指定されている `$ZF(-100)` は、既定のオペレーティング・システム・シェルを起動します。詳細は、“`$ZF(-100)` を使用したプログラムまたはシステム・コマンドの実行” を参照してください。

program で指定されたパス名にスペース文字が含まれる場合、パス名の処理はプラットフォームによって異なります。Windows および UNIX® の場合はパス名にスペース文字を使用することができますが、スペースを含むパス名は、追加の二重引用符 `()` で全体を囲む必要があります。これは、Windows の `cmd /c` 文に従っています。詳細は、Windows のコマンド・プロンプトで `cmd /?` を指定してください。

`%Library.File` クラスの `NormalizeFilenameWithSpaces()` メソッドを使用すると、パス名内のスペースがホスト・プラットフォームに合わせて処理されるようにできます。



\$ZF(-100) は、**%System\_Callout:U** 特権を必要とします。詳細は、“%System\_Callout:USE 特権の追加”を参照してください。

\$ZF(-100) がプロセスを生成できない場合、**<FUNCTION>** エラーを生成します。

## エラー処理

\$ZF(-100) は、以下の場合、**<NOTOPEN>** エラーを生成します。

- ・ `/STDIN=filename`、`/STDOUT=filename`、または `/STDERR=filename` を開けない。
- ・ 指定されたプログラムを起動できない。

エラーが SYSLOG に記録されます。オペレーティング・システムのエラー番号とメッセージは、`%SYSTEM.Process.OSError()` メソッドから取得できます。

## 監査

\$ZF(-100) を呼び出すたびに、OS コマンド監査レコードが監査ログに追加されます。このレコードには、以下のような情報が含まれます。

```
Command: /Users/myname/IRIS/jlc/bin/clmanager 4002
Flags: /ASYNC/SHELL
```

## \$ZF(-100)、\$ZF(-1)、および \$ZF(-2)

これら 3 つの関数はほとんどの点で同一です。これらは以下の点で異なります。

- ・ \$ZF(-100) は、同期的または非同期的に実行できます。オペレーティング・システム・シェルを使用して実行することも、シェルを使用せずに実行することもできます。常に `$ZCHILD` を設定します。`$ZF(-1)` と `$ZF(-2)` はどちらも、引数の指定なしで、オペレーティング・システム・シェルを起動します。`$ZF(-100)` では、オペレーティング・システム・シェルを起動するために `program` 引数 (および `/SHELL` フラグ) が必要です。

\$ZF(-100) は、すべての目的に適した関数であり、`$ZF(-1)` と `$ZF(-2)` の両方の代わりに使用できます。

- ・ **\$ZF(-1)** (非推奨) は OS シェルを使用して実行されます。この関数は同期的に実行されます。作成された子プロセスの完了を待つ間、現在のプロセスの実行を中断します。作成されたプロセスからステータス情報を受信し、作成されたプロセスの完了時に、これを終了ステータス・コード (整数値) として返します。`$ZF(-1)` は、`$ZCHILD` を設定しません。
- ・ **\$ZF(-2)** (非推奨) は OS シェルを使用して実行されます。この関数は非同期的に実行されます。現在のプロセスの実行を中断しません。子プロセスの生成において、直ちにステータスの値を返します。作成された子プロセスの完了を待たないので、そのプロセスからのステータス情報は受信できません。`$ZF(-2)` は、5 番目の引数が `True` の場合、`$ZCHILD` を設定します。

## 関連項目

- ・ [\\$ZF\(-1\) 関数](#)
- ・ [\\$ZF\(-2\) 関数](#)
- ・ [\\$ZCHILD 特殊変数](#)
- ・ `$ZF(-100)` を使用したプログラムまたはシステム・コマンドの実行

## \$ZHEX (ObjectScript)

16 進数文字列を 10 進数値に、あるいは 10 進数値を 16 進数値に変換します。

### 構文

```
$ZHEX(num)  
$ZH(num)
```

### 引数

引数	説明
<i>num</i>	変換対象の数値として評価される式で、引用符付き文字列あるいは整数（符号付き、符号なし）のいずれかです。

### 概要

\$ZHEX は、16 進数の文字列を 10 進数の整数に、あるいは 10 進数の整数を 16 進数値に変換します。

*num* が文字列値の場合、\$ZHEX はその文字列を 16 進数表記の数と解釈し、10 進数値を返します。文字列値は必ず引用符で囲みます。

*num* が数値の場合、\$ZHEX は、その数値を 16 進数形式の数の文字列表記に変換します。最初か最後の数値が 8 バイトの符号付き整数で表記されていない場合、\$ZHEX は <FUNCTION> エラーを発行します。

%SYSTEM.Util クラスの HexToDecimal() および DecimalToHex() メソッドを使用して同じ 16 進数 / 10 進数変換を実行できます。

#### ObjectScript

```
WRITE $SYSTEM.Util.DecimalToHex("27")
```

#### ObjectScript

```
WRITE $SYSTEM.Util.HexToDecimal("27"),!  
WRITE $SYSTEM.Util.HexToDecimal("1B")
```

\$ZHEX と \$CHAR を併用することにより、16 進数の文字コードを使用して Unicode 文字を指定することができます：

\$CHAR(\$ZHEX("hexnum"))。

### 16 進数への強制的な解釈

整数値を強制的に 16 進数として解釈させる場合、*num* 引数の最後に 16 進数以外の文字を連結します。例えば以下ようになります。

#### ObjectScript

```
WRITE $ZHEX(16_"H")
```

これは、22 を返します。

### 引数

#### *num*

文字列値、数値、そのいずれかを含む変数、あるいはそのいずれかに評価される式です。

文字列値は 16 進数の数として読み取られ、正の 10 進数の整数値に変換されます。\$ZHEX は、“A” から “F” の大文字と小文字を 16 進数の桁として認識します。先行のゼロはすべて切り捨てます。また、プラスやマイナスの符号、および小数点は認識しません。16 進数に含まれない文字を見つけたときに、文字列の評価を中止します。したがって、“F”、“f”、“00000F”、“F.7”、“FRED” は、すべて 10 進数の 15 に評価されます。文字列の最初の文字が 16 進数でない場合、\$ZHEX はゼロの文字列に評価されます。したがって、“0”、“0.9”、“+F”、“-F”、“H” の文字列は、すべてゼロに評価されます。NULL 文字列(“”)は無効な値なので、<FUNCTION> エラーを発行します。

整数値は、10 進数の数として読み取られ、16 進数に変換されます。整数は、正の数でも負の数でもかまいません。\$ZHEX は、先頭のプラスまたはマイナスの符号を認識します。先行のゼロはすべて切り捨てます。入れ子にされた算術演算を評価します。ただし、小数点は認識しません。小数点文字を検出した場合は、<FUNCTION> エラーを発行します。したがって、整数 217、0000217、+217、--217 はすべて 16 進数の D9 に評価されます。また -217、-0000217、-+217 は、すべて FFFFFFFF7 に評価されます。浮動小数点の数、後続符号、数値以外の文字などのその他の値では、<FUNCTION> または <SYNTAX> エラーを生成します。

## 例

### ObjectScript

```
WRITE $ZHEX("F")
```

これは、15 を返します。

### ObjectScript

```
WRITE $ZHEX(15)
```

これは、F を返します。

### ObjectScript

```
WRITE $ZHEX("1AB8")
```

これは、6840 を返します。

### ObjectScript

```
WRITE $ZHEX(6840)
```

これは、1AB8 を返します。

### ObjectScript

```
WRITE $ZHEX("XXX")
```

これは、0 を返します。

### ObjectScript

```
WRITE $ZHEX(-1)
```

これは、FFFFFFFFFFFFFF を返します。

### ObjectScript

```
WRITE $ZHEX((3+(107*2)))
```

これは D9 を返します。

## 関連項目

- ・ [ZZDUMP](#) コマンド
- ・ [\\$ASCII](#) 関数
- ・ [\\$CHAR](#) 関数

## \$ZISWIDE (ObjectScript)

文字列に 16 ビットの文字が含まれているか否かをチェックします。

### 構文

```
$ZISWIDE(string)
```

### 引数

引数	説明
string	1 文字以上の文字列。引用符で囲みます。

### 概要

\$ZISWIDE は、文字列に 16 ビットの文字が含まれるか否かを確認するために使用するブーリアン関数です。以下の値を返します。

値	意味
0	すべての文字が、ASCII 値 255 あるいはそれ以下 (8 ビット文字) である場合。NULL 文字列 ("" ) も 0 を返します。
1	1 つ以上の文字が ASCII 値 255 以上 (多バイト文字) である場合。

\$ZISWIDE は、文字の値を確認し、それらが ASCII 配列 (0-255) 内にあり 8 ビットで表示できるか、または多バイト文字の範囲 (256 - 65535) 内にあり Unicode 文字の 16 ビットすべてを使用するかを決定します。

### 例

以下の例では、最初の 2 つのコマンドは、文字列に 1 バイト文字 (8 ビット) しか含まれないので、0 を返します。3 つ目のコマンドは、文字列に多バイト文字値 (2 番目の値) が含まれるため、1 を返します。

#### ObjectScript

```
WRITE $ZISWIDE("abcd"),",",
WRITE $ZISWIDE($CHAR(71,83,77)),",",
WRITE $ZISWIDE($CHAR(71,300,77))
```

この例では 0,0,1 を返します。

### 関連項目

- [\\$ZPOSITION](#) 関数
- [\\$ZWASCII](#) 関数
- [\\$ZWCHAR](#) 関数
- [\\$ZWIDITH](#) 関数

## \$ZLASCII (ObjectScript)

4 バイト文字列を数値に変換します。

### 構文

```
$ZLASCII(string,position)
$ZLA(string,position)
```

### 引数

引数	説明
string	値、変数、式として指定できる文字列。長さは 4 バイト以上でなければなりません。
position	オプション - 文字列の開始位置。既定値は 1 です。

### 概要

\$ZLASCII が返す値は、使用する引数によって異なります。

- ・ \$ZLASCII(string) は、string の最初の文字位置から開始して、4 バイト文字列を数値に変換して返します。
- ・ \$ZLASCII(string,position) は、position で指定された開始位置から始まる 4 バイトの文字列を数値に変換して返します。

正常に終了すると、\$ZLASCII は常に正の整数を返します。\$ZLASCII は、string の長さが無効の場合、または position の値が無効の場合に、-1 を返します。

### \$ZLASCII と \$ASCII

\$ZLASCII は、シングル 8 ビット・バイトではなく 4 バイト (32 ビット) の単語を操作すること以外は、\$ASCII と同じです。2 バイト (16 ビット) の単語には \$ZWASCII を使用し、8 バイト (64 ビット) の単語には \$ZQASCII を使用します。

\$ZLASCII(string,position) の機能は、以下と同等です。

```
$ASCII(string,position+3)*256 + $ASCII(string,position+2)*256 + $ASCII(string,position+1)*256 + $ASCII(string,position)
```

### \$ZLASCII と \$ZLCHAR

\$ZLCHAR 関数は、論理的には \$ZLASCII 関数の逆です。例えば以下ようになります。

#### ObjectScript

```
SET x=$ZLASCII("abcd")
WRITE !,x
SET y=$ZLCHAR(x)
WRITE !,y
```

“abcd” を指定すると、\$ZLASCII は 1684234849 を返し、1684234849 を指定すると、\$ZLCHAR は “abcd” を返します。

### 関連項目

- ・ [\\$ASCII 関数](#)
- ・ [\\$ZLCHAR 関数](#)
- ・ [\\$ZWASCII 関数](#)
- ・ [\\$ZQASCII 関数](#)

# \$ZLCHAR (ObjectScript)

整数を対応する 4 バイト文字列に変換します。

## 構文

```
$ZLCHAR(n)
$ZLC(n)
```

## 引数

引数	説明
n	0 から 4294967295 の範囲の正の整数です。値、変数、式として指定することができます。

## 概要

\$ZLCHAR は n の 4 バイト (long) 文字の文字列を返します。文字の文字列のバイト数は、リトル・エンディアンで、最下位バイトが先に表示されます。

n が範囲外または負の数の場合、\$ZLCHAR は NULL 文字列を返します。n がゼロの場合または数値でない文字列の場合、\$ZLCHAR は 0 を返します。

## \$ZLASCII と \$ZLCHAR

\$ZLASCII 関数は、論理的には \$ZLCHAR の逆です。例えば以下ようになります。

### ObjectScript

```
SET x=$ZLASCII("abcd")
WRITE !,x
SET y=$ZLCHAR(x)
WRITE !,y
```

“abcd” を指定すると、\$ZLASCII は 1684234849 を返し、1684234849 を指定すると、\$ZLCHAR は “abcd” を返します。

## \$ZLCHAR と \$CHAR

\$ZLCHAR は、シングル 8 ビット・バイトではなく 4 バイト (32 ビット) の単語を操作すること以外は、\$CHAR と同じです。2 バイト (16 ビット) の単語には \$ZWASCII を使用し、8 バイト (64 ビット) の単語には \$ZQASCII を使用します。

\$ZLCHAR の機能は、以下の形式の \$CHAR と同等です。

### ObjectScript

```
SET n=$ZLASCII("abcd")
WRITE !,n
WRITE !,$CHAR(n#256,n\256#256,n\ (256**2)#256,n\ (256**3))
```

\$ZLASCII で “abcd” と指定すると、1684234849 が返され、この \$CHAR 文で 1684234849 と指定すると、“abcd” が返されます。

## 関連項目

- ・ [\\$ZLASCII 関数](#)
- ・ [\\$CHAR 関数](#)
- ・ [\\$ZWCHAR 関数](#)



- ・ [\\$ZQCHAR](#) 関数

# \$ZLN (ObjectScript)

指定された数の自然対数を返します。

## 構文

`$ZLN(n)`

## 引数

引数	説明
<i>n</i>	ゼロ以外の正の数字。値、変数、式として指定できます。

## 説明

\$ZLN は、*n* の自然対数 (基数 *e*) を返します。

ゼロや負数を指定すると <ILLEGAL VALUE> エラーが発生します。

非数値文字列は 0 として評価されるため、<ILLEGAL VALUE> エラーが発生します。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

対応する自然対数べき関数は、[\\$ZEXP](#) です。

## 例

以下の例は、1 から 10 の整数の自然対数を記述します。

### ObjectScript

```
FOR x=1:1:10 {
    WRITE !,"The natural log of ",x," = ",$ZLN(x)
}
QUIT
```

これは、以下を返します。

```
The natural log of 1 = 0
The natural log of 2 = .6931471805599453089
The natural log of 3 = 1.098612288668109691
The natural log of 4 = 1.386294361119890618
The natural log of 5 = 1.609437912434100375
The natural log of 6 = 1.791759469228055002
The natural log of 7 = 1.945910149055313306
The natural log of 8 = 2.079441541679835929
The natural log of 9 = 2.197224577336219384
The natural log of 10 = 2.302585092994045684
```

以下の例は、\$ZLN と \$ZEXP の関係を示しています。

### ObjectScript

```
SET x=$ZEXP(1) ; x = 2.718281828459045236
WRITE $ZLN(x)
```

これは、1 を返します。

### ObjectScript

```
WRITE $ZLN(0)
```

これは、<ILLEGAL VALUE> エラーを発行します。

## 関連項目

- ・ [\\$ZEXP](#) 関数
- ・ [\\$ZLOG](#) 関数
- ・ [\\$ZPI](#) 特殊変数

# \$ZLOG (ObjectScript)

指定された正の数の式の、常用対数の値を返します。

## 構文

`$ZLOG(n)`

## 引数

引数	説明
n	ゼロ以外の正の数字。値、変数、式として指定できます。

## 説明

\$ZLOG は、n の常用対数値を返します。

ゼロや負数を指定すると <ILLEGAL VALUE> エラーが発生します。

非数値文字列は 0 として評価されるため、<ILLEGAL VALUE> エラーが発生します。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

対応する自然対数 (基数 e) 関数は、[\\$ZLN](#) です。

## 例

以下の例は、1 から 10 の整数の、基底 10 の対数を記述します。

### ObjectScript

```
FOR x=1:1:10 {
    WRITE !,"The log of ",x," = ",$ZLOG(x)
}
QUIT
```

これは、以下を返します。

```
The log of 1 = 0
The log of 2 = .301029995663981195
The log of 3 = .477121254719662437
The log of 4 = .60205999132796239
The log of 5 = .698970004336018805
The log of 6 = .778151250383643633
The log of 7 = .845098040014256831
The log of 8 = .903089986991943586
The log of 9 = .954242509439324875
The log of 10 = 1
```

### ObjectScript

```
WRITE $ZLOG($ZPI)
```

これは、.4971498726941338541 を返します。

### ObjectScript

```
WRITE $ZLOG(.5)
```

これは、-.301029995663981195 を返します。

## ObjectScript

```
WRITE $ZLOG(0)
```

これは、〈ILLEGAL VALUE〉 エラーを発行します。

## 関連項目

- ・ [\\$ZEXP](#) 関数
- ・ [\\$ZLN](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZNAME (ObjectScript)

指定された名前文字列を正当な識別子として検証する関数です。

### 構文

```
$ZNAME(string,type,lang)
```

### 引数

引数	説明
string	評価する名前。引用符付きの文字列として指定します。
type	オプション - 実行する名前検証のタイプを指定する整数コード。有効値は 0 から 6 です。既定値は 0 です。
lang	オプション - string の検証時に使用する言語モードを指定する整数コード。有効値は 0 から 12 です。既定では、現在の言語モードを使用します。

### 概要

\$ZNAME は string 引数が正当な識別子であれば、1 (True) を返します。それ以外は 0 (False) を返します。オプションの type 引数によって、文字列に対して実行する名前検証のタイプを設定します。この引数を省略すると、ローカル変数の名前付け規約が検証の既定として使用されます。オプションの lang 引数は、検証に適用する言語モード規約を指定します。

\$ZNAME が正当な識別子として検証する識別子の使用がロケールで許可されていない場合があります。ご使用のロケールに対する有効な識別子文字は、各国言語サポート (NLS) 識別子ロケール設定で定義され、これらはユーザが変更することはできません。NLS の詳細は、["各国言語サポートのシステム・クラスの使用法"](#) を参照してください。

\$ZNAME は、文字の検証のみを実行し、識別子の文字列長の検証は行われません。

### 引数

#### string

正当な識別子名として検証する引用符付き文字列。有効な文字列に含まれる文字は、検証する識別子のタイプ (type で指定)、言語モード (lang) とロケールの定義に依存します。string は、識別子名のみを指定します。キャレット (^) 接頭語やオプションのグローバルの区切りネームスペース名の接頭語などの接頭語文字、および配列添え字やパラメータの括弧などの接尾語文字を含めることはできません。既定では、InterSystems IRIS 内の以下の文字が有効な識別子の文字とされます。

- ・ 大文字 : A から Z (\$CHAR(65) ~ \$CHAR(90))
- ・ 小文字 : a から z (\$CHAR(97) ~ \$CHAR(122))
- ・ アクセント記号付きの文字 : (\$CHAR(192) ~ \$CHAR(255)。ただし、\$CHAR(215) と \$CHAR(247) を除く)
- ・ Unicode 文字: 非ラテン文字セットの文字 (ギリシャ語やキリル文字など)。例えば、\$CHAR(256) ~ \$CHAR(687) および \$CHAR(913) ~ \$CHAR(1153)。ただし、\$CHAR(930) と \$CHAR(1014) を除く。
- ・ 数字 : 0 から 9 (\$CHAR(48) ~ \$CHAR(57)) 一部の識別子に位置の制限あり
- ・ パーセント記号 : % (\$CHAR(37)) 一部の識別子に位置の制限あり

また、\$ZNAME は、有効な文字として、\$CHAR(170)、\$CHAR(181)、および \$CHAR(186) を受け入れます。

注釈 日本のローケルでは、アクセント記号付きのラテン文字を識別子でサポートしていません。日本の識別子には、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) 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。

## type

実行する名前検証のタイプを指定する整数コード。

値	意味	制限文字
0	ローカル変数名を検証	最初の文字のみ : % 後続の文字のみ : 数字 0 - 9
1	ルーチン名を検証	最初の文字のみ : % 後続の文字のみ : 数字 0 - 9 およびピリオド (.) 文字。ピリオドを、ルーチン名の最初または最後の文字にすることはできません。
2	ラベル (タグ) 名を検証	最初の文字のみ : %
3	グローバル名またはプロセス・プライベート・グローバル名を検証	最初の文字のみ : % 後続の文字のみ : 数字 0 - 9 およびピリオド (.) 文字。ピリオドを、グローバル名の最初または最後の文字にすることはできません。
4	完全修飾されたクラス名を検証	最初の文字のみ : % 後続の文字のみ : 数字 0 - 9 およびピリオド (.) 文字。ピリオドを、ルーチン名の最初または最後の文字にすることはできません。 (以下を参照。)
5	メソッド名を検証	最初の文字のみ : % 後続の文字のみ : 数字 0 - 9
6	プロパティ名を検証	最初の文字のみ : % 後続の文字のみ : 数字 0 - 9

type = 0 の場合 (または指定されていない場合)、検証をパスした識別子は、ローカルの変数名またはすべてのタイプの ObjectScript 名に使用できます。これは最も厳しい検証の形式です。正当な識別子の最初の文字は、パーセント記号 (%) か、有効な文字 (英字) に限られます。正当な識別子の 2 番目以降の文字は、有効な文字 (英字) または数字のいずれかに限られます。

type = 2 の場合、検証にパスした識別子を行ラベルとして使用することができます。これは、桁 (0-9) を最初の文字として許容する唯一の識別子タイプです。ラベル名のみを指定します。(トリガで使用する) コロン接頭語を指定したり、パラメータの括弧をラベル名の後に指定したりしないでください。

type = 3 の場合、検証にパスした識別子をグローバル名およびプロセス・プライベート・グローバル名に使用することができます。ただし、グローバル名およびプロセス・プライベート・グローバル名には、ワイド文字を含めることはできません。\$ZNAME は、ワイド文字をすべての名前検証タイプで有効な識別子文字であると判断します。そのため、type=3 の場合は、ワイド文字を含む識別子は \$ZNAME 検証を通過しますが、グローバル名またはプロセス・プライベート・グローバル名として使用されると、<WIDE CHAR> エラーが生成されます。



type = 4 の場合、検証にパスした識別子をクラス名として使用することができます。クラス名にはピリオドを含めることができますが、ピリオドの直後に数字や別のピリオドを配置することはできません。ピリオドの使用のこれらの制限は、type=1 および type=3 の検証には適用されません。すべての type で、string の最初または最後の文字にピリオドがあると、有効な識別子になりません。

## lang

検証に使用する言語モードを指定する整数コード。InterSystems IRIS は、現在の言語モードを変更することなく、指定された言語モードの規約を検証に適用します。(使用可能な現在の言語モードのリストについては、%SYSTEM.Process クラスの LanguageMode() メソッドを参照してください。)既定では、\$ZNAME によって現在の言語モードの規約が使用されます。InterSystems IRIS のすべての言語モードでは同じ名前付け規約が使用されるため、lang を省略して既定を使用できます。

## 例

以下の例では、\$ZNAME 関数で式が True (1) であると検証される場合を示しています。最後の 2 つの例にはピリオドが含まれていることに注意してください。ピリオドは、ルーチン名 (type=1) およびグローバル名 (type=3) で使用できます。

### ObjectScript

```
WRITE !,$ZNAME("A")
WRITE !,$ZNAME("A1")
WRITE !,$ZNAME("%A1",0)
WRITE !,$ZNAME("%A1",1)
WRITE !,$ZNAME("A.1",1)
WRITE !,$ZNAME("A.1",3)
```

以下の例では、最初の \$ZNAME は検証に失敗して 0 を返します。既定では、ローカル変数名を検証し、ローカル変数名の最初の文字に数字を使用できないからです。2 つ目の \$ZNAME は、検証を通過して 1 を返します。type=2 はラベルの検証を指定し、ラベル名の最初の文字には数字が使用できるからです。

### ObjectScript

```
WRITE "local var: ",$ZNAME("1A"),!
WRITE "label: ",$ZNAME("1A",2)
```

以下の例では、すべての type 値の検証に失敗します。すべてのタイプの InterSystems IRIS で、名前の最初の文字以外ではパーセント記号を使用できません。

### ObjectScript

```
FOR i=0:1:6 {
    WRITE "type ",i," is ",$ZNAME("A%i",i),!
}
```

以下の例は、ローカル変数名に有効な 8 ビットの識別子の完全セットを示しています。この有効な識別子の文字には、文字 (英字) ASCII 192 から ASCII 255 (算術記号である ASCII 215 と ASCII 247 を除く) が含まれています。

### ObjectScript

```
FOR n=1:1:255 {
    IF $ZNAME("A"_$CHAR(n),0) & $ZNAME($CHAR(n),0){
        WRITE !,$ZNAME($CHAR(n))," ASCII code=",n," Char.=",$CHAR(n) }
    ELSEIF $ZNAME($CHAR(n),0){
        WRITE !,$ZNAME($CHAR(n))," ASCII code=",n," 1st Char.=",$CHAR(n) }
    ELSEIF $ZNAME("A"_$CHAR(n),0){
        WRITE !,$ZNAME("A"_$CHAR(n))," ASCII code=",n," Subseq. Char.=",$CHAR(n) }
    ELSE { }
}
WRITE !,"All done"
```

以下の例は、InterSystems IRIS の検証にパスします。指定されたギリシャ文字は有効な Unicode 文字なので、\$ZNAME 検証を通過します。ただし、この名前は、グローバルまたはプロセス・プライベート・グローバル (type=3) では使用できず、一部のロケール (日本語ロケールなど) では使用できません。

### ObjectScript

```
WRITE $C(913)_$C(961)_$C(947)_$C(959),!  
FOR i=0:1:6 {  
    WRITE "type ",i," is ",$ZNAME($C(913)_$C(961)_$C(947)_$C(959),i),!  
}
```

## SQL 識別子

SQL 識別子には、ObjectScript 識別子の有効文字ではない句読点文字 (アンダースコア (\_)、アット記号 (@)、シャープ記号 (#)、ドル記号 (\$)) が含まれる場合もあります。SQL ルーチン名では先頭文字以外にパーセント記号 (%) を使用できません。詳細は、“InterSystems SQL の使用法” の “[識別子](#)” を参照してください。

## 関連項目

- ObjectScript の “[シンボル・テーブル](#)”
- InterSystems SQL の “[シンボル・テーブル](#)”

# \$ZPOSITION (ObjectScript)

式の中で、指定されたフィールド幅に収まる文字数を返します。

## 構文

```
$ZPOSITION(expression,field,pitch)
```

## 引数

引数	説明
expression	文字列式。
field	フィールド幅を指定する整数の式。
pitch	オプション - 全角文字に使用するための pitch 値を指定する数値式。既定値は 2 です。他に、1、1.25、1.5 の値を取ることもできます。

## 概要

\$ZPOSITION は、field 値内に収めることができる expression の文字数を返します。pitch 値は、全角文字に使用する幅を決定します。他の文字はすべて 1 (既定の幅) が割り当てられ、半角と認識されます。半角は 1 としてカウントされるため、field の値は field に収めることができる半角文字の数を示すことにもなります。

\$ZPOSITION は、幅の累計が field 値と等しくなるか、expression 内にそれ以上文字が入らなくなるまで、expression の文字の幅を 1 ずつ追加します。したがって結果は、指定された field 値内に収まる文字数になります。これには、完全に収まらない小数部の文字も含まれます。

## 例

以下の例では、変数 string に 2 つの半角文字と、それに続く全角文字 1 つが含まれていると想定しています。

### ObjectScript

```
WRITE $ZPOSITION(string,3,1.5)
```

これは、2.666666666666666667 を返します。

この例では、string の最初の 2 文字が指定されたフィールド幅に収まり、1 文字が残されます。string の 3 番目の文字は、1.5 の幅の全角文字 (pitch 引数で設定) ですが、2/3 (1/1.5) だけが収まりますが、全体は収まりきらないことになります。その結果、上記のような小数部分が返されます。

以下の例では、string には 1 つの全角文字と、それに続く 2 つの半角文字が含まれています。返される値は 2.5 です。

### ObjectScript

```
WRITE $ZPOSITION(string,3,1.5)
```

この結果は、上記の例と値が異なります。この場合、最初の 2 文字が合わせて 2.5 の幅を持ち、指定されたフィールド幅の余りが 0.5 になります。この幅 0.5 には 3 番目の文字の半分 (.5/1) のみが収まります。

最後に、string が 3 つの半角文字を含む文字列の場合、3 つのすべての文字は完全に (ぴったりと) 収まり、結果は 3 になります。

### ObjectScript

```
WRITE $ZPOSITION(string,3,1.5)
```

注釈 全角文字は、InterSystems IRIS プロセスにロードされているパターン照合テーブルを調べると判別できます。全角の属性を持つ文字はすべて、全角文字として認識されます。特別な ZFWCHARZ パットコードを使用して、この属性を確認することもできます (char?1ZFWCHARZ など)。全角の属性の詳細は、“[各国言語サポートのシステム・クラスの使用法](#)” の \$X/\$Y タブの説明を参照してください。

## 関連項目

- ・ [\\$ZWIDTH](#) 関数
- ・ [\\$ZENKAKU](#) 関数

# \$ZPOWER (ObjectScript)

数値の指定した累乗の値を返します。

## 構文

```
$ZPOWER ( num , exponent )
```

## 引数

引数	説明
num	累乗される数字
exponent	指数

## 概要

\$ZPOWER は、num 引数の値の n 乗を返します。

この関数は、指数演算子 (\*\*) と同様の処理を実行します 有効なオペランドと特定の値の組み合わせに対して返される値の詳細は、“[指数演算子 \(\\*\\*\)](#)” リファレンス・ページを参照してください。

## 引数

### num

累乗される数字。これには、整数、浮動小数点、負の数、正の数、またはゼロを指定します。InterSystems IRIS 標準の数値または IEEE 倍精度バイナリ浮動小数点数 (\$DOUBLE 値) で指定できます。値、変数、式として指定することができます。

“[数値としての文字列](#)” で説明されているように、num を引用符付きの文字列として指定した場合、文字列は数値として解析されます。NULL 文字列 ("")、および数値以外の文字列は、ゼロに評価されます。

### exponent

exponent には、整数、浮動小数点、負の数、正の数、またはゼロを指定します。InterSystems IRIS 標準の数値または IEEE 倍精度バイナリ浮動小数点数 (\$DOUBLE 値) で指定できます。数値、文字列値、変数、式として指定することができます。

“[数値としての文字列](#)” で説明されているように、exponent を引用符付きの文字列として指定した場合、文字列は数値として解析されます。NULL 文字列 ("")、および数値以外の文字列は、ゼロに評価されます。

次のように num と exponent を組み合わせるとエラーになります。

- num が負の数の場合、exponent は整数である必要があります。整数でない場合は、<ILLEGAL VALUE> エラーが生成されます。
- num が 0 の場合、exponent には正の数または 0 を指定する必要があります。それ以外の数を指定すると、<ILLEGAL VALUE> エラーまたは <DIVIDE> エラーが生成されます。
- \$ZPOWER (9,153) のような大きな値を exponent に指定すると、オーバーフローが発生して <MAXNUMBER> エラーが生成されるか、アンダーフローが発生して 0 が返されることがあります。どちらの結果になるかは、num が 1 (または -1) よりも大きいのか、および exponent が正か負かによって決まります。処理で InterSystems IRIS がサポートする最大数を超えた場合、<MAXNUMBER> エラーが発生します。詳細は、“[極端に大きな数字](#)” を参照してください。

有効なオペランドと特定の値の組み合わせに対して返される値の詳細は、"[指数演算子 \(\\*\\*\)](#)" リファレンス・ページを参照してください。

## 例

以下の例は、2 を 10 乗します。

### ObjectScript

```
SET x=0
WHILE x < 10 {
    SET rtn=$ZPOWER(2,x)
    WRITE !,"The ",x," power of 2=",rtn
    SET x=x+1 }
```

## 関連項目

- ・ [\\$ZSQR](#) 関数
- ・ [\\$ZEXP](#) 関数
- ・ [\\$ZLN](#) 関数
- ・ [\\$ZLOG](#) 関数
- ・ [べき乗 \(\\*\\*\) 演算子](#)

## \$ZQASCII (ObjectScript)

8 バイト文字列を数値に変換します。

### 構文

```
$ZQASCII(string,position)
$ZQA(string,position)
```

### 引数

引数	説明
string	文字列。値、変数、式として指定することができます。長さは 8 バイト以上でなければなりません。
position	オプション - 正の整数として表される、文字列の開始位置。既定は 1 です。位置は、8 バイト文字列ではなくシングル・バイト単位でカウントされます。position を文字列の最後のバイトにすることはできません。また、position が文字列の最後を超えることもできません。position の数値は、小数桁数を切り捨て、先行するゼロやプラス記号などを削除することによって、整数として解析されます。

### 説明

\$ZQASCII が返す値は、使用する引数によって異なります。

- ・ \$ZQASCII(string) は、string の最初の文字位置から開始して、8 バイト文字列を数値に変換して返します。
- ・ \$ZQASCII(string,position) は、position で指定したバイト位置から開始して、8 バイト文字列を数値に変換して返します。

\$ZQASCII は、正の整数または負の整数のいずれかを返すことができます。

\$ZQASCII は、string の長さが無効の場合、または position の値が無効の場合に、<FUNCTION> エラーを返します。

### 例

以下は、文字列 "abcdefgh" を数値に変換する例です。

#### ObjectScript

```
WRITE $ZQASCII("abcdefgh")
```

これは 7523094288207667809 を返します。

次の例も 7523094288207667809 を返します。

#### ObjectScript

```
WRITE !,$ZQASCII("abcdefgh",1)
WRITE !,$ZQASCII("abcdefghxx",1)
WRITE !,$ZQASCII("xxabcdefghxx",3)
```

### \$ZQASCII と \$ASCII

\$ZQASCII は、シングル 8 ビット・バイトではなく 8 バイト (64 ビット) の単語を操作すること以外は、\$ASCII と同じです。16 ビットの単語には \$ZWASCII を使用し、32 ビットの単語には \$ZLASCII を使用します。



## \$ZQASCII と \$ZQCHAR

\$ZQCHAR 関数は、論理的に \$ZQASCII の逆になります。例えば以下のようになります。

### ObjectScript

```
WRITE $ZQASCII("abcdefgh")
```

これは 7523094288207667809 を返します。

### ObjectScript

```
WRITE $ZQCHAR(7523094288207667809)
```

これは、"abcdefgh" を返します。

## 関連項目

- ・ [\\$ASCII](#) 関数
- ・ [\\$ZQCHAR](#) 関数

# \$ZQCHAR (ObjectScript)

整数を対応する 8 バイト文字列に変換します。

## 構文

```
$ZQCHAR(n)
$ZQC(n)
```

## 引数

引数	説明
<i>n</i>	-9223372036854775808 から 9223372036854775807 の範囲の整数。値、変数、式として指定できます。

## 説明

\$ZQCHAR は、*n* の 2 進数表記に対応する 8 バイト (quad) 文字の文字列を返します。文字の文字列のバイト数は、リトル・エンディアンの順序で、最下位バイトが先に表示されます。

*n* が範囲外の場合、\$ZQCHAR は NULL 文字列を返します。*n* がゼロの場合または数値でない文字列の場合、\$ZQCHAR は 0 を返します。

## \$ZQCHAR と \$CHAR

\$ZQCHAR は、シングル 8 ビット・バイトではなく 8 バイト (64 ビット) の単語を操作すること以外は、\$CHAR と同じです。16 ビットの単語には \$ZWCHAR を使用し、32 ビットの単語には \$ZLCHAR を使用します。

## \$ZQCHAR と \$ZQASCII

\$ZQASCII は、論理的には \$ZQCHAR 関数の逆です。例えば以下ようになります。

### ObjectScript

```
WRITE $ZQCHAR(7523094288207667809)
```

これは、abcdefgh を返します。

### ObjectScript

```
WRITE $ZQASCII("abcdefgh")
```

これは、7523094288207667809 を返します。

## 例

以下の例は、整数 7523094288207667809 の 8 バイト文字列を返します。

### ObjectScript

```
WRITE $ZQCHAR(7523094288207667809)
```

これは、"abcdefgh" を返します。

## 関連項目

- ・ [\\$ZQASCII 関数](#)

- ・ [\\$CHAR](#) 関数
- ・ [\\$ZLCHAR](#) 関数
- ・ [\\$ZWCHAR](#) 関数

# \$ZSEARCH (ObjectScript)

指定したターゲット・ファイルの完全なファイル指定（パス名とファイル名）を返します。

## 構文

```
$ZSEARCH(target)
$ZSE(target)
```

## 引数

引数	説明
target	ファイル名、パス名、または NULL 文字列のいずれか。1 つ、または複数の * や ? ワイルドカード文字が含まれることもあります。

## 説明

\$ZSEARCH は指定されたターゲット・ファイルまたはディレクトリの完全なファイル仕様（パス名とファイル名）を返します。ファイル名にはワイルドカードを含む場合もあるため、\$ZSEARCH はワイルドカードを満たす一連の完全修飾パス名を返します。

**注釈** オペレーティング・システムによっては、ディレクトリ・パスの区切り文字としてスラッシュ (/) 文字が使用されます。バックスラッシュ (\) 文字を使用するオペレーティング・システムもあります。この説明では、“スラッシュ”という用語は必要に応じてスラッシュとバックスラッシュのいずれかを意味します。

target 引数でパス名が指定されていない場合は、\$ZSEARCH によって現在作業中のディレクトリ内が検索されます。\$ZSEARCH は、以下の順番で一致プロセスの規則を適用します。

1. \$ZSEARCH は、target をスキャンして、パーセント文字 (%) で囲まれているか否かを確認します。このようなテキストを見つけた場合は、この文字列を環境変数として処理します。\$ZSEARCH は、文字列の名前変換を実行します。
2. \$ZSEARCH は、前の手順で得られた文字列をスキャンして、最後のスラッシュを見つけます。最後のスラッシュが見つかった場合は、検索されるパスまたはディレクトリとして、最後のスラッシュの前の文字までの文字列を使用します。最後のスラッシュが見つからない場合は、現在作業しているディレクトリを検索します。これは、現在のネームスペースから決定します。
3. 手順 2 で \$ZSEARCH が最後のスラッシュを見つけた場合、最後のスラッシュの後に続く target 文字列の一部を、ファイル名検索パターンとして使用します。手順 2 で最後のスラッシュが見つからなかった場合は、手順 1 で得た文字列全体を、ファイル名検索パターンとして使用します。

ファイル名検索パターンは、正当なファイル名文字列あるいはファイル名のワイルドカード式です。検索パターンに一致する最初のファイル名は、\$ZSEARCH 関数値として返されます。最初に一致したファイルがどれであるかは、プラットフォームに依存します（下記のメモ・セクションで説明しています）。

次回の \$ZSEARCH の呼び出しで target として NULL 文字列を指定すると、\$ZSEARCH は前回の target で続行され、検索パターンに一致する次のファイル名を返します。検索パターンに一致するファイルがない場合、\$ZSEARCH は NULL 文字列を返します。

以下の例に示すように、%Library.File クラスの NormalizeDirectory() メソッドを使用して、指定したファイルまたはディレクトリの完全なパス名が返されるようにすることもできます。

## ObjectScript

```
NEW $NAMESPACE
SET $NAMESPACE="%SYS"
WRITE ##class(%Library.File).NormalizeDirectory("IRIS.DAT"),!
NEW $NAMESPACE
SET $NAMESPACE="USER"
WRITE ##class(%Library.File).NormalizeDirectory("IRIS.DAT")
```

ただし、NormalizeDirectory() ではワイルドカードは使用できません。

## ワイルドカード

\$ZSEARCH では、引用符付きの target 文字列内で以下のワイルドカード式を使用できます。

ワイルドカード	一致
*	0 文字以上の文字列と一致。
?	1 文字と一致。Windows システムでは、名前要素の最後の 1 文字または 0 文字と一致します。

上記のワイルドカードは、ホスト・プラットフォームの使用規定に従います。Windows では、\$ZSEARCH は大文字と小文字を区別せずに検索を実行し、検索したファイルまたはディレクトリの実際の文字を返します。例えば、“j\*” は“JOURNAL”、“journal”、または“Journal”と一致する可能性があります。実際のディレクトリ名が“Journal”の場合は、“Journal”が返されます。

Windows および UNIX® システムでは、現在のディレクトリを指定する単一ドット(.) や、その親ディレクトリを指定する二重ドット(..) など、標準のパス名記号も使用できます。これらの記号は、ワイルドカードと組み合わせて使用できます。

## 引数

### target

target 引数に使用できる値のタイプは、以下のとおりです。

target のタイプ	説明
pathname	リストしたいファイル・グループ、またはファイルへのパスを指定する文字列に評価される式。パスの長さは、最大 1024 文字です。
filename	ファイル名。既定の位置は、現在のデータセットです。
NULL 文字列(“”)	前の \$ZSEARCH の後、次に一致するファイル名を返します。

## 例

以下の Windows の例は、USER ネームスペースの中でファイル拡張子が“.DAT”で終了するすべてのファイルを検索します。

### ObjectScript

```
NEW $NAMESPACE
SET $NAMESPACE="USER"
SET file=$ZSEARCH("*.DAT")
WHILE file="" {
    WRITE !,file
    SET file=$ZSEARCH("")
}
WRITE !,"That is all the matching files"
QUIT
```

これは、以下を返します。

```
c:\InterSystems\IRIS\mgr\user\IRIS.DAT
```

以下の Windows の例は、USER ネームスペースの中で文字 “i” で開始するすべてのファイルを検索します。

### ObjectScript

```
NEW $NAMESPACE
SET $NAMESPACE="USER"
SET file=$ZSEARCH("i*")
WHILE file="" {
    WRITE !,file
    SET file=$ZSEARCH("")
}
WRITE !,"That is all the matching files"
QUIT
```

これは、以下を返します。

```
c:\InterSystems\IRIS\mgr\user\IRIS.DAT
c:\InterSystems\IRIS\mgr\user\iris.lck
```

## ディレクトリのロック

適切な結果を返すため、\$ZSEARCH がすべてのファイルをディレクトリに返すまで（つまり、\$ZSEARCH が NULL 文字列を返すまで、または新規の \$ZSEARCH が開始するまで）プロセスはそのディレクトリを開いたままにしておきます。これは、ディレクトリの削除などの他の処理を妨げる可能性があります。\$ZSEARCH を開始するとき、\$ZSEARCH("") が NULL 文字列を返すまで繰り返します。また、すべてのファイルを検索したくない場合は、\$ZSEARCH(-1) などの、存在しないファイル名で \$ZSEARCH を実行します。

## Windows のサポート

Windows では、target 引数は、ワイルドカード文字 (\* と ?) を含む標準ファイル指定です。

- ・ \*ワイルドカードはドットと一致させるのに使用できますが、?ワイルドカードは使用できません。したがって、“MYFILE\*”は MYFILEFOLDER、MYFILE.DOC、および MYFILEBACKUP.DOC と一致しますが、“MYFILE?DOC”は MYFILE.DOC と一致しません。
- ・ ?ワイルドカードは、名前要素内の 0 文字とは一致しません。したがって、“MY?FILE.DOC”は MY2FILE.DOC と一致しますが、MYFILE.DOC とは一致しません。
- ・ ?ワイルドカードは、名前要素の最後の 0 文字と一致します。余分な末尾の ?ワイルドカードは無視されます。したがって、“MYFILE?.DOC”は MYFILE2.DOC と MYFILE.DOC とともに一致します。

特にディレクトリを指定しなければ、現在作業中のディレクトリを使用します。\$ZSEARCH はディレクトリで最初に一致したエントリを、アルファベット順に返します。これは完全ファイル指定、または完全修飾パス名を返します。ドライブ文字は、どのように指定したかに関係なく、常に大文字で返されます。

既定では、Windows ではファイル名拡張子の接尾語の最初の 3 文字だけがチェックされます。このため、\$ZSEARCH("\*.doc") では、.doc 接尾語を持つすべてのファイルだけでなく、.docx 接尾語を持つすべてのファイルも返されます。検索を .docx ファイルのみに制限する場合は、\$ZSEARCH("\*.docx") のように、4 文字の接尾語を指定する必要があります。末尾の ?ワイルドカードを使用して、4 文字以上の接尾語に検索を制限することはできません。

## UNIX® のサポート

UNIX® では、target 引数に UNIX® 標準のファイル指定を使用します。このパラメータでは、ワイルドカード文字 (\* と ?) も使用できます。特にディレクトリを指定しなければ、現在作業中のディレクトリを使用します。

UNIX® では、\$ZSEARCH は、ディレクトリ内の最初にアクティブなエントリを返します。UNIX® は、ディレクトリのエントリをアルファベット順に保存しないので、返り値はアルファベット順ではありません。Windows のプラットフォームとは異なり、\$ZSEARCH 関数は現在作業中のディレクトリが使用されない限り、完全ファイル指定、または完全修飾パス名を返しません。

## 関連項目

- ・ [OPEN コマンド](#)
- ・ [USE コマンド](#)
- ・ [\\$ZIO 特殊変数](#)
- ・ [シーケンシャル・ファイルの入出力](#)



# \$ZSEC (ObjectScript)

指定された角度の三角関数のセカントを返します。

## 構文

`$ZSEC(n)`

## 引数

引数	説明
n	範囲 0 から 2 Pi までのラジアン単位の角度。値、変数、式として指定することができます。

## 概要

\$ZSEC は、n の三角関数のセカントを返します。結果は、符号付きの 10 進数の数字です。0 のセカントは 1 です。pi のセカントは -1 です。

**注釈** InterSystems IRIS はホストのオペレーティング・システムのルーチンを使用して、三角関数の計算を行います。このため、異なるオペレーティング・システムから得られた結果は正確に一致しない場合があります。

## 引数

**n**

Pi から 2 Pi までの範囲のラジアン単位の角度。値、変数、式として指定することができます。\$ZPI 特殊変数を使用して Pi 値を指定できます。Pi より小さいか、2 Pi より大きい、正または負の値を指定できます。InterSystems IRIS はこれらの値を Pi の対応する倍数に解析します。例えば、3 Pi は Pi と等しく、負の Pi は Pi と等しいです。

非数値文字列は 0 として評価されるため \$ZSEC は 1 を返します。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

## 例

以下は、数のセカントを計算する例です。

### ObjectScript

```

READ "Input a number: ",num
IF $ZABS(num)>(2*$ZPI) { WRITE !,"number is a larger than 2 pi" }
ELSE {
    WRITE !,"the secant is: ",$ZSEC(num)
}
QUIT

```

## 関連項目

- ・ [\\$ZCSC](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZSEEK (ObjectScript)

現在のシーケンシャル・ファイルに新しいオフセットを確立します。

### 構文

```
$ZSEEK(offset,mode)
```

### 引数

引数	説明
offset	現在のファイルのオフセット (文字数単位)。整数で指定します。ゼロ、正の整数、または負の整数を指定できます。
mode	オプション - オフセットの相対位置を決定する整数値。0 = 先頭、1 = 現在位置、2 = 末尾。既定は 0 です。

### 概要

\$ZSEEK は、現在のデバイスに新しい offset を設定します。現在のデバイスは、シーケンシャル・ファイルである必要があります。現在のデバイスがシーケンシャル・ファイルではない場合、\$ZSEEK は <FUNCTION> エラーを発行します。

mode 引数は、offset の基点にするポイント (先頭、現在位置、または末尾) を決定します。

\$ZSEEK は、オフセットを実行した後にファイル内での現在位置を返します。引数なしの \$ZSEEK は、オフセットを実行せずにファイル内での現在位置を返します。

\$ZSEEK は、デバイスがシーケンシャル・ファイルの場合にのみ使用できます。ターミナルから、または開いているシーケンシャル・ファイルがない場合に \$ZSEEK を呼び出すと、<FUNCTION> エラーが返されます。現在のデバイスを特に設定していない場合、\$ZSEEK はデバイスが主デバイスであると見なします。

[\\$ZIO](#) 特殊変数は、現在のファイルのパス名を含みます。[\\$ZPOS](#) 特殊変数は、現在のファイル位置を含みます。これは、\$ZSEEK(0,1) または \$ZSEEK() (引数なし) で返される値と同じです。

### 引数

#### offset

mode によって設定されたポイントからのオフセット (文字数)。これは位置ではなくオフセットです。したがって、ファイルの先頭からのオフセット 0 は、位置 1 (ファイルの開始) になります。オフセット 1 は位置 2 で、ファイルの 2 番目の文字になります。

オフセットは、ファイルの末尾より後の位置にすることができます。\$ZSEEK は、指定したオフセットに対して空白を入力します。

mode が 1 または 2 の場合、オフセットを負の数字にすることができます。ファイルの先頭より前の位置になる負の数字を指定すると、<FUNCTION> エラーが返されます。

#### mode

有効な値は、以下のとおりです。

値	意味
0	オフセットは、ファイルの最初を基準とします（絶対位置）
1	オフセットは、現在の位置を基準とします
2	オフセットは、ファイルの最後を基準とします

mode 値を指定しない場合、\$ZSEEK はモード値を 0 であると想定します。

## 例

以下の Windows の例では、シーケンシャル・ファイルを開いて "AAA" を書き込み、\$ZSEEK(10) で、ファイルの先頭から 10 文字の位置にオフセットを設定します (7 個の空白を入力します)。続いて、その位置に "BBB" を書き込んでから、引数なしの \$ZSEEK() によって、ファイルの先頭からの結果のオフセット（この場合は 13）を返します。

### ObjectScript

```
SET $TEST=0
SET myfile="C:\InterSystems\IRIS\mgr\user\zseektestfile.txt"
OPEN myfile:("WNS"):10
  IF $TEST=0 {WRITE "OPEN failed" RETURN}
USE myfile
WRITE "AAA"
SET rtn=$ZSEEK(10)
WRITE "BBB"
SET rtnend=$ZSEEK()
CLOSE myfile
WRITE "set offset:",rtn," end position:",rtnend
```

以下の Windows の例では、シーケンシャル・ファイルに文字 "A" を 10 回書き込み、そのたびに、文字間に挿入する空白スペースを増やします。ここでは、\$ZPOS を使用して、現在のファイル位置を判断しています。

### ObjectScript

```
SET $TEST=0
SET myfile="C:\InterSystems\IRIS\mgr\user\zseektestfile2.txt"
OPEN myfile:("WNS"):10
  IF $TEST=0 {WRITE "OPEN failed" RETURN}
USE myfile
FOR i=1:1:10 {WRITE "A" SET rtn=$ZSEEK($ZPOS+i,0)}
CLOSE myfile
```

## 関連項目

- [OPEN コマンド](#)
- [USE コマンド](#)
- [CLOSE コマンド](#)
- [\\$ZIO 特殊変数](#)
- [\\$ZPOS 特殊変数](#)
- [シーケンシャル・ファイルの入出力](#)

## \$ZSIN (ObjectScript)

指定された角度の三角関数のサインを返します。

### 構文

`$ZSIN(n)`

### 引数

引数	説明
n	Pi から 2 Pi までの範囲のラジアン単位の角度。提供されている他の数値は、この範囲内の値に変換されます。

### 説明

\$ZSIN は、n の三角関数のサインを返します。結果は符号付きの 10 進数で、範囲は 1 から -1 です (メモ参照)。\$ZSIN(0) は 0 を返します。\$ZSIN(\$ZPI/2) は 1 を返します。

**注釈** \$ZSIN は (他のすべての三角関数のように)、利用できる小数桁数の数に丸められた pi を基にして、値を計算します。したがって、\$ZSIN(\$ZPI) から返される値は .00000000000000000462644 で、\$ZSIN(\$ZPI\*2) から返される値は -.00000000000000000092529 です。このため、これらの返り値を 0 と比較する制限テストを実行するべきではありません。

### 引数

n

Pi から 2 Pi までの範囲のラジアン単位の角度。値、変数、式として指定することができます。\$ZPI 特殊変数を使用して Pi 値を指定できます。Pi より小さいか、2 Pi より大きい、正または負の値を指定できます。InterSystems IRIS はこれらの値を Pi の対応する倍数に解析します。例えば、3 Pi は Pi と等しく、負の Pi は Pi と等しいです。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

### 例

以下は、数のサインを計算する例です。

#### ObjectScript

```
READ "Input a number: ",num
IF $ZABS(num)>(2*$ZPI) { WRITE !,"number is a larger than 2 pi" }
ELSE {
    WRITE !,"the sine is: ",$ZSIN(num)
}
QUIT
```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。どちらの場合も、pi のサインは 0 ではない小数値ですが、pi/2 のサインは正確に 1 です。

#### ObjectScript

```
WRITE !,"the sine is: ",$ZSIN($ZPI)
WRITE !,"the sine is: ",$ZSIN($DOUBLE($ZPI))
WRITE !,"the sine is: ",$ZSIN($ZPI/2)
WRITE !,"the sine is: ",$ZSIN($DOUBLE($ZPI)/2)
```

以下の例では、\$ZSIN 関数はすべてゼロ (0) を返します。

#### ObjectScript

```
WRITE !,"the sine is: ",$ZSIN(0.0)
WRITE !,"the sine is: ",$ZSIN(-0.0)
WRITE !,"the sine is: ",$ZSIN($DECIMAL(0.0))
WRITE !,"the sine is: ",$ZSIN($DOUBLE(0.0))
WRITE !,"the sine is: ",$ZSIN($DECIMAL(-0.0))
WRITE !,"the sine is: ",$ZSIN($DOUBLE(-0.0))
WRITE !,"the sine is: ",$ZSIN(-$DECIMAL(0.0))
WRITE !,"the sine is: ",$ZSIN(-$DOUBLE(0.0))
```

これは、AIX を含むすべてのプラットフォームに当てはまります。

## 関連項目

- ・ [\\$ZCOS](#) 関数
- ・ [\\$ZARCSIN](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZSQR (ObjectScript)

指定された数の平方根の値を返します。

### 構文

`$ZSQR(n)`

### 引数

引数	説明
<i>n</i>	正の数字、もしくはゼロ (NULL 文字列および数値以外の文字列値は、ゼロとして処理されます)。値、変数、式として指定できます。

### 説明

\$ZSQR は、*n* の平方根を返します。1 の平方根を 1 として返します。0 の平方根と NULL 文字列 ("") の平方根は、0 として返されます。負の数値を指定すると、<ILLEGAL VALUE> エラーを返します。絶対値関数 \$ZABS を使用して、負の数を正の数に変換することができます。

### 例

以下の例は、ユーザ指定の数の平方根を返します。

#### ObjectScript

```
READ "Input number for square root: ",num
IF num<0 { WRITE "ILLEGAL VALUE: no negative numbers" }
ELSE { WRITE $ZSQR(num) }
QUIT
```

以下は、その例です。

#### ObjectScript

```
WRITE $ZSQR(2)
```

これは、1.414213562373095049 を返します。

#### ObjectScript

```
WRITE $ZSQR($ZPI)
```

これは、1.772453850905516027 を返します。

### 関連項目

- ・ [\\$ZABS 関数](#)
- ・ [\\$ZPOWER 関数](#)

## \$ZSTRIP (ObjectScript)

指定された文字列から、文字のタイプと文字を削除します。

### 構文

```
$ZSTRIP(string,action,remchar,keepchar)
```

### 引数

引数	説明
string	削除される文字列。
action	string から削除する対象。action は、アクション・コードと、それに続く 1 つまたは複数のマスク・コードで構成されます。remchar を指定すると、マスク・コードはオプションとなります。action は引用符付き文字列として指定されます。
remchar	オプション — 削除する特定の文字の値の文字列。action がマスク・コードを含まない場合、remchar は削除する文字をリスト表示します。action がマスク・コードを含む場合、remchar は action 引数のマスク・コードで保護されていない、その他の削除対象文字を表示します。
keepchar	オプション — 削除されない特定の文字の値の文字列。action 引数のマスク・コードで削除が指定された文字から指定します。マスク・コードを指定してから、keepchar を指定する必要があります。

### 概要

\$ZSTRIP 関数は、指定された string から文字のタイプおよび個別の文字の値を削除します。action 引数では、実行する削除処理の種類を示すアクション・コードを指定し、(オプションで) 削除する文字のタイプを指定するマスク・コードを指定します。個別の文字の値を指定して、remchar 引数を使用して削除することができます。\$ZSTRIP は、文字のタイプ(すべての小文字など)とリストされた文字の値("AEIOU" の文字など)の両方を同時に削除できます。オプションで remchar 引数および keepchar 引数を使用して、削除または保持する文字の値を個別に指定することにより、action 引数のマスク・コードの効果を変更できます。

詳細は、"[パターン・マッチング \(?\)](#)" を参照してください。文字のタイプ、文字の順序、および文字の範囲の選択には、正規表現関数である [\\$LOCATE](#) および [\\$MATCH](#) を使用することもできます。

### 引数

#### action

削除する文字を示す文字列。アクション・コードとして指定され、オプションで 1 つまたは複数のマスク・コードを後に付けることができます。



## アクション・コード

アクション・コード	意味
*	マスク・コード (複数可) に一致するすべての文字を削除します
<	マスク・コード (複数可) に一致する先頭の文字を削除します
>	マスク・コード (複数可) に一致する末尾の文字を削除します
<>	マスク・コード (複数可) に一致する先頭および末尾の文字を削除します
=	マスク・コード (複数可) に一致する繰り返し文字を削除します。文字の繰り返しを検出すると、このコードは 1 つのインスタンスを残して重複する文字を削除します。このコードは、隣接して重複する文字のみを削除します。したがって、“aaaaaabc” から “a” を削除すると “abc” となりますが、“abaca” から “a” を削除しても文字列 “abaca” がそのまま返されます。重複文字のテストでは、大文字と小文字を区別します。
<=>	マスク・コード (複数可) に一致する先頭、末尾、および繰り返し文字を削除します

アクション・コードは、\* 文字または単一の <、>、もしくは = 文字の任意の組合せで構成できます。

文字のタイプを削除するには、action 文字列がアクション・コードと、それに続く 1 つまたは複数のマスク・コードで構成されている必要があります。特定の文字の値を削除するには、マスク・コードを省略して、remchar 値を指定します。マスク・コードと remchar 値は両方指定することができます。マスク・コードも remchar 値も指定しない場合、\$ZSTRIP は string を返します。

## マスク・コード

マスク・コード	意味
E	すべてを削除します
A	すべてのアルファベット文字を削除します
P	空白スペースと、句読記号文字を削除します
C	制御文字 (0-31、127-159) を削除します
N	数字を削除します (数値文字列は、\$ZSTRIP の適用前には、キャノニック形式に変換されないことに注意してください)。
L	アルファベットの小文字を削除します
U	アルファベットの大文字を削除します
W	空白を削除します (\$C(9)、\$C(32)、\$C(160))

マスク・コードは大文字でも小文字でも指定できます。

マスク・コード文字の前に単項否定演算子 (') を置くことで、このタイプの文字を削除しないように指定することができます。単項否定演算子マスク・コードを指定する前に、単項否定演算子を持たないマスク・コードを少なくとも 1 つ指定する必要があります。単項否定演算子を持たないマスク・コードはすべて、単項否定演算子を持つマスク・コードより前に置く必要があります。

## remchar

削除する特定の文字。引用符付きの文字列として指定します。これらの remchar 文字は、任意の順序で指定できます。重複してもかまいません。

マスク・コードを指定しない場合、\$ZSTRIP は action 引数を remchar 文字 (複数可) に適用します。マスク・コードを指定する場合、削除する文字を remchar で 1 つまたは複数追加して指定します。例えば、action 引数で、すべての数字 ("\*N") だけでなく、"E" (科学的記数法を表すために使用されるもの) の文字も削除する場合、2 番目の \$ZSTRIP で示すように、文字列 "E" を remchar 引数で追加します。

### ObjectScript

```
SET str="first:123 second:12E3"
WRITE $ZSTRIP(str,"*N"),!
WRITE $ZSTRIP(str,"*N","E")
```

### keepchar

削除しない特定の文字。例えば、すべての空白とアルファベット (\*WA) を削除して、大文字の M を残す場合は、文字列 "M" を keepchar 引数で追加します。

## 例

以下の例では、数値文字をすべて削除します。数値文字列はキャノニック形式に変換されないで、以下のように、+ および E の文字は削除されません。

### ObjectScript

```
SET str="+123E4"
WRITE $ZSTRIP(str,"*N")
```

これは、+E を返します。

以下の例においては、最初の \$ZSTRIP では句読点文字すべてが削除され、2 番目の \$ZSTRIP では空白文字を除くすべての句読点文字が削除されます。

### ObjectScript

```
SET str="ABC#$$^ DEF& *GHI****"
WRITE $ZSTRIP(str,"*P"),!
WRITE $ZSTRIP(str,"*P'W")
```

以下の例では、英字の小文字を除く ('L) すべての文字が削除されます。この例では remchar 引数を使用して小文字 x のみを削除し、それ以外的小文字は保存します。

### ObjectScript

```
SET str="xXx-Aa BXXbx Cxc Dd xxEeX^XXx"
WRITE $ZSTRIP(str,"*E'L","x")
```

これは、abcde を返します。

以下の例では、英字の小文字を除く ('L) すべての文字が削除されます。この例では、remchar 引数値を指定していませんが (区切りのコンマは指定しています)、大文字の A、B、および C を残すために keepchar 引数は指定しています。

### ObjectScript

```
SET str="X-Aa BXXb456X CXc Dd XxEeX^XFFFfXX"
WRITE $ZSTRIP(str,"*E'L",,"ABC")
```

これは、AaBbCcdef を返します。

以下の例では、マスク・コードを指定していませんが、英字の "X" および "x" が文字列内に出現した場合には、それらを削除するよう指定しています。文字列内の他の文字はすべて返されます。

## ObjectScript

```
SET str="+x $1x,x23XX4XX.X56XxxxxxX"  
WRITE $ZSTRIP(str,"*", "Xx")
```

これは、+ \$1,234.56 を返します。

以下の例では、マスク・コードを指定していませんが、先頭および末尾の文字として "x" を削除すること、および繰り返しの "x" が文字列内で出現した場合、それを削除することを指定しています。

## ObjectScript

```
SET str="xxxxx00xx0111xxx01x0000xxxxx"  
WRITE $ZSTRIP(str,"<=>", "x")
```

これは、00x0111x01x0000 を返します。

以下の例では、空白文字および英字の小文字を除くすべての数値文字、アルファベット文字、および句読点文字を削除します。単項否定演算子を持たないマスク・コードはすべて、単項否定演算子を持つマスク・コードよりいずれも前に置く必要があることに注意してください。

## ObjectScript

```
SET str="Aa66*% B&$b Cc987 #Dd Ee"  
WRITE $ZSTRIP(str,"*NAP'W'L")
```

これは、a b c d e を返します。

以下の例では、マスク・コード A (すべてのアルファベット文字) に一致する先頭文字、末尾文字、および繰り返し文字を削除します。

## ObjectScript

```
SET str="ABC123DDDEEFFffffGG5555567HI JK"  
WRITE $ZSTRIP(str,"<=>A")
```

これは、123DEffG5555567HI を返します。\$ZSTRIP はマスク (1) に含まれないタイプの文字に遭遇するまで、先頭文字 (ABC) を削除し、かつ非マスク文字 (空白) に遭遇するまで、文字列の最後から末尾文字を削除しています。マスク・タイプの繰り返し文字は、出現回数が 1 回に (DDDEE = DE) 削減されています。繰り返しテストは大文字と小文字が区別されることに注意してください (FFfff = Ff)。マスク・タイプ (55555) ではない繰り返し文字は、影響を受けません。

以下の例では、16 進数字 0 ~ 9 および A ~ F を除いたすべての文字を削除します。

## ObjectScript

```
SET str="123$ GYJF870B-QD @##%+ "  
WRITE $ZSTRIP(str,"*E'N", "ABCDEF")
```

これは、123F870BD を返します。

## 関連項目

- ・ [\\$EXTRACT](#) 関数
- ・ [\\$ZCONVERT](#) 関数
- ・ 正規表現の [\\$LOCATE](#) および [\\$MATCH](#) 関数
- ・ [パターン・マッチング \(?\)](#)

# \$ZTAN (ObjectScript)

指定された角度の三角関数のタンジェントを返します。

## 構文

`$ZTAN(n)`

## 引数

引数	説明
<i>n</i>	Pi から 2 Pi までの範囲のラジアン単位の角度。提供されている他の数値は、この範囲内の値に変換されます。

## 説明

\$ZTAN は、*n* の三角関数のタンジェントを返します。結果は、符号付きの 10 進数の数字です。

注釈 \$ZTAN は (他のすべての三角関数のように)、利用できる小数桁数の数に丸められた pi を基にして、値を計算します。したがって、\$ZTAN(\$ZPI) から返される値は `-.000000000000000000462644` で、\$ZTAN(-\$ZPI) から返される値は `.000000000000000000462644` です。このため、これらの返り値を 0 と比較する制限テストを実行するべきではありません。\$ZTAN(0) は 0 です。

## 引数

*n*

範囲 0 から 2 Pi までのラジアン単位の角度。値、変数、式として指定することができます。

非数値文字列は 0 として評価されます。混合数値文字列および非数値文字列の評価の詳細は、“[数値としての文字列](#)”を参照してください。

## 例

以下は、数のタンジェントを計算する例です。

### ObjectScript

```
READ "Input a number: ",num
WRITE !,"the tangent is: ",$ZTAN(num)
QUIT
```

以下の例は、InterSystems IRIS 小数 (\$DECIMAL の数値) の結果と \$DOUBLE の数値の結果を比較します。どちらの場合でも、0 のタンジェントは正確に 0 ですが、pi のタンジェントは 0 ではない負の小数値です。

### ObjectScript

```
WRITE !,"the tangent is: ",$ZTAN(0.0)
WRITE !,"the tangent is: ",$ZTAN($DOUBLE(0.0))
WRITE !,"the tangent is: ",$ZTAN($ZPI)
WRITE !,"the tangent is: ",$ZTAN($DOUBLE($ZPI))
WRITE !,"the tangent is: ",$ZTAN(1.0)
WRITE !,"the tangent is: ",$ZTAN($DOUBLE(1.0))
```

## 関連項目

・ [\\$ZARCTAN](#) 関数

- ・ [\\$ZSIN](#) 関数
- ・ [\\$ZPI](#) 特殊変数

## \$ZTIME (ObjectScript)

時刻を検証し、これを内部形式から指定の表示形式に変換します。

### 構文

```
$ZTIME(htime,tformat,precision,erropt,localeopt)
$ZT(htime,tformat,precision,erropt,localeopt)
```

### 引数

引数	説明
htime	数値、変数名、式として指定できる内部システム時刻
tformat	オプション — 時刻の値を返す形式を指定する整数値
precision	オプション — 時刻を表すときに、小数点以下の有効桁数を指定する数値省略した場合は、秒の小数部は切り捨てられます。
erropt	オプション — htime 引数が無効であると判断された場合に返される式
localeopt	オプション — どのロケールを使用するかを指定するブーリアン・フラグ。0 の場合は、現在のロケールが、時刻区切り文字などの文字、文字列、および時刻のフォーマットに使用するオプションを決定します。1 の場合は、ODBC ロケールが、これらの文字、文字列、およびオプションを決定します。ODBC ロケールは変更できません。これは、異なる各国語サポート (NLS) の選択を行った InterSystems IRIS プロセス間で移植できる、日付文字列および時刻文字列をフォーマットするために使用されます。既定値は 0 です。

指定の引数値間で省略された引数は、プレースホルダのコンマで示されます。末尾のプレースホルダのコンマは必要ありませんが、あってもかまいません。省略された引数を示すコンマの間に空白があってもかまいません。

### 概要

\$ZTIME 関数は、特殊変数 \$HOROLOG や \$ZTIMESTAMP の時刻形式で指定された内部システム時刻 htime を、出力可能形式に変換します。オプションの引数を使用しない場合、時刻は “hh:mm:ss” 形式で返されます。“hh” は 24 時間制で表す時刻、“mm” は分、“ss” は秒です。それ以外の場合、時刻は tformat と precision の引数値で指定された形式で返されます。

### 引数

#### htime

この値は、その日の午前 0 時 00 分から経過した秒数を表します。[\\$HOROLOG](#) 値の秒の構成要素で、[\\$PIECE\(\\$HOROLOG, ",", 2\)](#) を使用して抽出できます。htime は、整数、または precision で指定された精度の小数桁数を持つ小数です。

tformat 値が -1 ～ 4 の場合、htime の有効値は、0 ～ 86399 (-0 は 0 として処理される) の範囲の整数部分にする必要があります。この範囲外の値では、<ILLEGAL VALUE> エラーが発生します。tformat 値が 9 および 10 の場合、htime の有効値に負の数字および 86399 より大きい値を含めることもできます。

#### tformat

サポートされている値は、以下のとおりです。

tformat	説明
-1	現在のロケールの TimeFormat プロパティから、有効な形式値を取得します。既定は 1 です。tformat を指定せず、localeopt が未指定または 0 の場合は、これが既定になります。
1	時刻を “hh:mm:ss” (24 時間) で表します。
2	時刻を “hh:mm” (24 時間) で表します。
3	時刻を “hh:mm:ss[AM/PM]” (12 時間) で表します。
4	時刻を “hh:mm[AM/PM]” (12 時間) で表します。

ユーザのロケールの既定の時刻形式を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
```

12 時間形式では、午前と午後を時間接尾語で表します。ここでは、AM と PM で表示されています。ユーザのロケールの既定の時間接尾語を決定するには、以下の NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!
```

### precision

この関数は、precision 引数で指定された小数点以下の桁数まで、秒の小数部を表示します。例えば precision の値に 3 を入力すると、\$ZTIME は秒の小数部を小数点以下 3 桁まで表示します。また、precision が 9 の場合、\$ZTIME は秒の小数部を小数点以下 9 桁まで表示します。サポートされている値は、以下のとおりです。

値	説明
-1	現在のロケールの TimePrecision プロパティから precision 値を取得します。既定は 0 です。precision を指定しない場合は、これが既定の振る舞いになります。
n	ゼロ (0) 以上の n の値は、時刻を小数点以下の n 桁まで表すことを示します。
0	0 に設定された場合、または既定値の 0 である場合、秒の小数部は切り捨てられます。

ユーザのロケールの既定の時刻精度を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimePrecision")
```

### erropt

この引数は、無効な htime 値に対応するエラー・メッセージを抑制します。〈ILLEGAL VALUE〉エラー・メッセージを生成する代わりに、関数は erropt で示された値を返します。

### localeopt

この引数は、時刻のオプションのソースとして、ユーザの現在のロケール定義 (0) または ODBC ロケール定義 (1) を選択します。ODBC ロケールは変更できません。これは、異なる各国語サポート (NLS) の選択を行った InterSystems IRIS プロセス間で移植できる、日付文字列および時刻文字列をフォーマットするために使用されます。

ODBC ロケールの時刻定義は、以下のとおりです。



- ・ 時刻形式の既定は 1 です。時刻区切り文字は ":" です。時刻精度は 0 (小数秒なし) です。
- ・ 午前と午後の指定子は、"AM" と "PM" です。"Noon" と "Midnight" という単語が使用されます。

## 例

特殊変数 \$HOROLOGY を使用して現在の現地時刻を返すには、\$PIECE 関数を使用して、\$HOROLOGY の秒の部分指定する必要があります。以下は、24 時間形式 "13:55:11" で時刻を返します。

### ObjectScript

```
WRITE $ZTIME($PIECE($HOROLOGY,"",2),1)
```

次の例では、現在時刻に対して htime が \$PIECE(\$HOROLOGY,"",2) に設定されています。これらの例は、さまざまな \$ZTIME を使用し、異なる時刻形式を返す方法を示しています。

通常、以下の例は、"13:28:55" という形式の時刻を返します。しかし、この形式はロケールに依存します。

### ObjectScript

```
SET htime=$PIECE($HOROLOGY,"",2)
WRITE $ZTIME(htime)
```

以下の例は、"13:28:55" の形式で時刻を返します。

### ObjectScript

```
SET htime=$PIECE($HOROLOGY,"",2)
WRITE $ZTIME(htime,1)
```

以下の例は、"13:28:550.999" の形式で時刻を返します。

### ObjectScript

```
SET htime=$PIECE($HOROLOGY,"",2)
WRITE $ZTIME(htime,1,3)
```

以下の例は、"13:28:55.999999999" の形式で時刻を返します。

### ObjectScript

```
SET htime=$PIECE($HOROLOGY,"",2)
WRITE $ZTIME(htime,1,9)
```

以下の例は、"13:28" の形式で時刻を返します。

### ObjectScript

```
SET htime=$PIECE($HOROLOGY,"",2)
WRITE $ZTIME(htime,2)
```

以下の例は、"01:28:24PM" の形式で時刻を返します。

### ObjectScript

```
SET htime=$PIECE($HOROLOGY,"",2)
WRITE $ZTIME(htime,3)
```

以下の例は、"01:28PM" の形式で時刻を返します。

## ObjectScript

```
SET htime=$PIECE($HOROLOG,"",2)
WRITE $ZTIME(htime,4)
```

以下の例は、“13:45:56.021”の形式で時刻を返します。小数点以下有効桁数 3 桁で現在の UTC 時刻を返します。

## ObjectScript

```
SET t=$ZTIME($PIECE($ZTIMESTAMP,"",2),1,3)
WRITE "Current UTC time is ",t
```

## 無効な引数値

- ・ 無効な tformat 値を指定すると、<FUNCTION> エラーが返されます。
- ・ 0 ～ 86399 (以下) の許容範囲に含まれない値を htime に指定し、erropt 値を入力していない場合は、9 および 10 を除くすべての tformat に対して <ILLEGAL VALUE> エラーが返されます。

## 小数点区切り文字

\$ZTIME は、整数と小数の区切り文字として、現在のロケールの **DecimalSeparator** プロパティ値を使用します。DecimalSeparator の既定値は “.” であり、すべてのドキュメントの例でこの区切り文字を使用しています。

ユーザのロケールの既定の小数点区切り文字を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

## 時刻区切り文字

既定では、InterSystems IRIS は、現在のロケールの **TimeSeparator** プロパティ値を使用して、時間文字列の区切り文字を判別します。既定の区切り文字は “:” です。すべてのドキュメントの例でこの区切り文字を使用しています。

ユーザのロケールの既定の時刻区切り文字を決定するには、以下の NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator")
```

## 時間接尾語

既定では、InterSystems IRIS は現在のロケールのプロパティを使用して、時間の接尾語名を判別します。\$ZTIME で、これらのプロパティ (および、対応する既定値) は以下のとおりです。

- ・ AM (“AM”)
- ・ PM (“PM”)

このドキュメントでは、これらのプロパティに常に既定値を使用します。

ユーザのロケールの既定の時間接尾語を決定するには、以下の NLS クラス・メソッドを呼び出します。

## ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM")
```

## 関連項目

- ・ [\\$ZDATETIME](#) 関数
- ・ [\\$ZDATETIMEH](#) 関数
- ・ [\\$ZTIMEH](#) 関数
- ・ [\\$PIECE](#) 関数
- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ZTIMEH (ObjectScript)

時刻を検証し、これを表示形式から InterSystems IRIS の内部形式に変換します。

### 構文

```
$ZTIMEH(time,tformat,erroppt,localeopt)
$ZTH(time,tformat,erroppt,localeopt)
```

### 引数

引数	説明
time	変換される時刻値
tformat	オプション - 変換する時刻形式を指定する数値
erroppt	オプション - time 引数が無効であると判断された場合に返される式
localeopt	オプション - どのロケールを使用するかを指定するブーリアン・フラグ。0 の場合は、現在のロケールが、時刻区切り文字などの文字、文字列、および時刻のフォーマットに使用するオプションを決定します。1 の場合は、ODBC ロケールが、これらの文字、文字列、およびオプションを決定します。ODBC ロケールは変更できません。これは、異なる各国語サポート (NLS) の選択を行った InterSystems IRIS プロセス間で移植できる、日付文字列および時刻文字列をフォーマットするために使用されます。既定値は 0 です。

指定の引数値間で省略された引数は、プレースホルダのコンマで示されます。末尾のプレースホルダのコンマは必要ありませんが、あってもかまいません。省略された引数を示すコンマの間に空白があってもかまいません。

### 概要

\$ZTIMEH 関数は、\$ZTIME 関数で生成された形式の時刻値を、特殊変数 \$HOROLOGY と \$ZTIMESTAMP の形式に変換します。オプションの引数 tformat が指定されていない場合、入力時刻は “hh:mm:ss.fff...” 形式にする必要があります。これ以外の場合、時刻を正しく変換するには、\$ZTIME 関数で出力可能時刻の生成に使用されたものと同じ整数形式コードを使用する必要があります。

### 小数秒

\$ZTIME と異なり、\$ZTIMEH は精度を指定することはできません。\$ZTIME で返された元の時刻形式の小数秒は、\$ZTIMEH が返す値に保持されます。

### 引数

#### tformat

サポートされている値は、以下のとおりです。

Code	概要
-1	現在のロケールの TimeFormat プロパティから、有効な形式値を取得します。既定は 1 です。tformat を指定せず、localeopt が未指定または 0 の場合は、これが既定になります。
1	時刻を “hh:mm:ss” (24 時間) の形式で入力します。
2	時刻を “hh:mm” (24 時間) の形式で入力します。
3	時刻を “hh:mm:ss[AM/PM]” (12 時間) の形式で入力します。
4	時刻を “hh:mm[AM/PM]” (12 時間) の形式で入力します。

ユーザのロケールの既定の時刻形式を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
```

### erroppt

この引数は、無効な time 値に対応するエラー・メッセージを抑制します。〈ILLEGAL VALUE〉エラー・メッセージを生成する代わりに、関数は erroppt で示された値を返します。

### localeopt

この引数は、時刻のオプションのソースとして、ユーザの現在のロケール定義 (0) または ODBC ロケール定義 (1) を選択します。ODBC ロケールは変更できません。これは、異なる各国語サポート (NLS) の選択を行った InterSystems IRIS プロセス間で移植できる、日付文字列および時刻文字列をフォーマットするために使用されます。

ODBC ロケールの時刻定義は、以下のとおりです。

- ・ 時刻形式の既定は 1 です。時刻区切り文字は “:” です。時刻精度は 0 (小数秒なし) です。
- ・ 午前と午後の指定子は、“AM” と “PM” です。“Noon” と “Midnight” という単語が使用されます。

## 例

入力時刻が “14:43:38” の場合、以下の例は両方とも 53018 を返します。

### ObjectScript

```
SET time="14:43:38"
WRITE !,$ZTIMEH(time)
WRITE !,$ZTIMEH(time,1)
```

入力時刻が “14:43:38.974” の場合、以下の例は 53018.974 を返します。

### ObjectScript

```
SET time="14:43:38.974"
WRITE $ZTIMEH(time,1)
```

## 無効な引数値

無効な tformat コード (-1 未満または 4 より大きい整数値、0、非整数値) を指定すると、〈FUNCTION〉エラーが出力されます。

erroppt 値を指定しない場合は、以下の条件に該当すると 〈ILLEGAL VALUE〉エラーが出力されます。

- ・ 0 から 23 までの許容範囲外の時刻値で time を指定した場合

- ・ 0 から 59 までの許容範囲外の分値で time を指定した場合
- ・ 0 から 59 までの許容範囲外の秒値で time を指定した場合
- ・ 現在のロケールの TimeSeparator プロパティ値以外の区切り文字を使用する time 値を指定した場合

## 時刻区切り文字

既定では、InterSystems IRIS は、現在のロケールの TimeSeparator プロパティ値を使用して、時刻文字列の区切り文字を判別します。既定の区切り文字は “:” です。すべてのドキュメントの例でこの区切り文字を使用しています。

ユーザのロケールの既定の時刻区切り文字を決定するには、GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator")
```

## 時間接尾語

既定では、InterSystems IRIS は現在のロケールのプロパティを使用して、時間の接尾語名を判別します。\$ZTIMEH で、これらのプロパティ (および、対応する既定値) は以下のとおりです。

- ・ AM (“AM”)
- ・ PM (“PM”)
- ・ Midnight (“MIDNIGHT”)
- ・ Noon (“NOON”)

このドキュメントでは、これらのプロパティに常に既定値を使用します。

ユーザのロケールの既定の時間接尾語を決定するには、以下のように GetFormatItem() NLS クラス・メソッドを呼び出します。

### ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("AM"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("PM"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("Midnight"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("Noon")
```

## 関連項目

- ・ [\\$ZDATETIME 関数](#)
- ・ [\\$ZDATETIMEH 関数](#)
- ・ [\\$ZTIME 関数](#)
- ・ [\\$HOROLOG 特殊変数](#)
- ・ [\\$ZTIMESTAMP 特殊変数](#)
- ・ [各国言語サポートのシステム・クラスの使用法](#)

## \$ZVERSION(1) (ObjectScript)

---

オペレーティング・システムの種類を返します。

### 構文

```
$ZVERSION(1)
```

### 引数

サポートされる引数値は数値 1 のみです。

### 説明

\$ZVERSION(1) は、現在のオペレーティング・システムの種類を整数コードとして返します。Windows の場合は 2、UNIX® の場合は 3、および不明な場合は 0 の値を返します。

isWINDOWS および isUNIX の[システム提供マクロ](#)を使用することで、同じ情報をブーリアン値として返すことができます。

%SYSTEM.Version.GetOS() を使用することで、同じ情報を文字列として返すことができます。

[\\$ZVERSION](#) 特殊変数を使用することで、現在のオペレーティング・システムの種類を含む完全な InterSystems IRIS バージョン情報を返すことができます。

### 例

以下の例は、現在のオペレーティング・システムの種類を返します。

#### ObjectScript

```
#include %occInclude
WRITE "OS type as code: ", $ZVERSION(1), !
WRITE "OS type as Boolean: ", !
WRITE "Windows? ", $$$isWINDOWS, " UNIX? ", $$$isUNIX, !
WRITE "OS type as string: ", %SYSTEM.Version.GetOS(), !
WRITE "InterSystems IRIS Version: ", $ZVERSION
```

### 関連項目

- ・ [\\$ZVERSION](#) 特殊変数。



## \$ZWASCII (ObjectScript)

2 バイト文字列を数値に変換します。

### 構文

```
$ZWASCII(string,position)  
$ZWA(string,position)
```

### 引数

引数	説明
string	文字列。値、変数、式として指定することができます。長さは 2 バイト以上でなければなりません。
position	オプション - 正の整数として表される、文字列の開始位置。既定は 1 です。位置は、2 バイト文字列ではなくシングル・バイト単位でカウントされます。position を文字列の最後のバイトにすることはできません。また、position が文字列の最後を超えることもできません。position の数値は、小数桁数を切り捨て、先行するゼロやプラス記号などを削除することによって、整数として解析されます。

### 概要

\$ZWASCII が返す値は、使用する引数によって異なります。

- ・ \$ZWASCII(string) は、string の最初の文字位置から開始して、2 バイト文字列を数値に変換して返します。
- ・ \$ZWASCII(string,position) は、position で指定したバイト位置から開始して、2 バイト文字列を数値に変換して返します。

正常に終了すると、\$ZWASCII は常に正の整数を返します。\$ZWASCII は、string の長さが無効の場合、または position の値が無効の場合に、-1 を返します。

### 例

以下は、文字列 "ab" を数値に変換する例です。

#### ObjectScript

```
WRITE $ZWASCII("ab")
```

これは 25185 を返します。

次の例も 25185 を返します。

#### ObjectScript

```
WRITE !,$ZWASCII("ab",1)  
WRITE !,$ZWASCII("abxx",1)  
WRITE !,$ZWASCII("xxabxx",3)
```

次の例では、string または position が無効です。それぞれの場合に、\$ZWASCII 関数は -1 を返します。

## ObjectScript

```
WRITE !,$ZWASCII("a")
WRITE !,$ZWASCII("aba",3)
WRITE !,$ZWASCII("ababab",99)
WRITE !,$ZWASCII("ababab",0)
WRITE !,$ZWASCII("ababab",-1)
```

## \$ZWASCII と \$ASCII

\$ZWASCII は、シングル 8 ビット・バイトではなく 2 バイト (16 ビット) の単語を操作すること以外には、\$ASCII と同じです。4 バイト (32 ビット) の単語には \$ZLASCII を使用し、8 バイト (64 ビット) の単語には \$ZQASCII を使用します。

\$ZWASCII(string,position) は以下と同じ機能を持ちます。

$\$ASCII(string, position+1)*256+\$ASCII(string, position)$

## \$ZWASCII と \$ZWCHAR

\$ZWCHAR 関数は、論理的に \$ZWASCII の逆になります。例えば以下のようになります。

## ObjectScript

```
WRITE $ZWASCII("ab")
```

これは 25185 を返します。

## ObjectScript

```
WRITE $ZWCHAR(25185)
```

これは “ab” を返します。

## 関連項目

- ・ [\\$ASCII](#) 関数
- ・ [\\$ZWCHAR](#) 関数

## \$ZWCHAR (ObjectScript)

整数を対応する 2 バイト文字列に変換します。

### 構文

```
$ZWCHAR(n)  
$ZWC(n)
```

### 引数

引数	説明
<i>n</i>	0 から 65535 の範囲の正の整数です。値、変数、式として指定することができます。

### 概要

\$ZWCHAR は、*n* の 2 進数表記に対応する 2 バイト (wide) 文字の文字列を返します。文字の文字列のバイト数は、リトル・エンディアンで、最下位バイトが先に表示されます。機能的には、以下と同一です。

#### ObjectScript

```
WRITE $CHAR(n#256,n\256)
```

*n* が範囲外または負の数の場合、\$ZWCHAR は NULL 文字列を返します。*n* がゼロの場合または数値でない文字列の場合、\$ZWCHAR は 0 を返します。

### \$ZWCHAR と \$CHAR

\$ZWCHAR は、シングル 8 ビット・バイトではなく 2 バイト (16 ビット) ワードで動作することを除けば \$CHAR と同じです。4 バイト (32 ビット) ワードには \$ZLCHAR を使用し、8 バイト (64 ビット) ワードには \$ZQCHAR を使用します。

### \$ZWCHAR と \$ZWASCII

\$ZWASCII は、論理的には \$ZWCHAR 関数の逆です。例えば以下ようになります。

#### ObjectScript

```
WRITE $ZWCHAR(25185)
```

これは ab を返します。

#### ObjectScript

```
WRITE $ZWASCII("ab")
```

これは、25185 を返します。

### 例

以下の例は、整数 25185 に 2 バイト文字列を返します。

#### ObjectScript

```
WRITE $ZWCHAR(25185)
```

これは ab を返します。

## 関連項目

- ・ [\\$ZWASCII](#) 関数
- ・ [\\$CHAR](#) 関数
- ・ [\\$ZLCHAR](#) 関数
- ・ [\\$ZQCHAR](#) 関数

## \$ZWIDTH (ObjectScript)

式で文字の合計幅を返します。

### 構文

```
$ZWIDTH(expression,pitch)
```

### 引数

引数	説明
<i>expression</i>	文字列式。
<i>pitch</i>	オプション - 全角文字に使用する数ピッチ値。既定値は 2 です。他に、1、1.25、1.5 の値を取ることもできます (末尾がゼロの任意の数字を持つこれらの値は許可されます)。他のすべての <i>pitch</i> 値は、<FUNCTION> エラーになります。

### 概要

\$ZWIDTH は、*expression* で文字の合計幅を返します。*pitch* 値は、全角文字に使用する幅を決定します。それ以外すべての文字の幅は 1 で、半角と認識されます。

### 例

変数 STR に、2 つの半角文字と、その後に 1 つの全角文字が含まれているとします。

#### ObjectScript

```
WRITE $ZWIDTH(STR,1.5)
```

これは、3.5 を返します。

この例は、半角文字を 2 つの合計 2 に、全角文字の 1.5 (指定されたピッチ値) を追加して、合計 3.5 になります。

### 全角文字

全角文字は、InterSystems IRIS プロセスにロードされているパターン照合テーブルを調べると判別できます。全角の属性を持つ文字はすべて、全角文字として認識されます。特別な ZFWCHARZ パターン・コードを使用して、この属性を確認できます (char?1ZFWCHARZ)。全角の属性の詳細は、“[各国言語サポートのシステム・クラスの使用法](#)” の \$X/\$Y タブの説明を参照してください。

### 関連項目

- ・ [\\$ZPOSITION](#) 関数
- ・ [\\$ZZENKAKU](#) 関数

## \$ZWPACK および \$ZWBPACK (ObjectScript)

2 つの 8 ビット文字を 1 つの 16 ビット文字にパックします。

### 構文

```
$ZWPACK(string)
```

```
$ZWBPACK(string)
```

### 引数

引数	説明
string	2 つ以上の 8 ビット文字で構成される文字列。string は、偶数の文字でなければなりません。

### 概要

\$ZWPACK 関数は、8 ビット文字の文字列を 16 ビットのワイド文字の文字列としてリトル・エンディアン順にパックします。2 つの 8 ビット文字は、1 つの 16 ビット文字にパックされます。

\$ZWBPACK は同じタスクを実行しますが、8 ビット文字は文字列は 16 ビットのワイド文字にビッグ・エンディアン順に格納されます。

文字列のパックは、格納する文字列の文字数と文字列操作を半分にするための方法です。アンパックすると、元の 8 ビット文字の文字列を表示するために復元されます。これらの操作は、データ内で Unicode 文字が許可されているときは使用しないでください。

入力 string には、以下の要件があります。

- string は、偶数の文字で構成されなければなりません。空の文字列は許可され、空の文字列が返されます。奇数の文字を指定すると、<FUNCTION> エラーが返されます。
- string には、マルチバイト文字を含めることができません。string で \$ZISWIDE を使用して、マルチバイト文字が含まれていないかを確認できます。マルチバイト文字を含む string で \$ZWPACK あるいは \$ZWBPACK を使用すると、システムは <WIDE CHAR> エラーを生成します。

IsBigEndian() クラス・メソッドを使用して、オペレーティング・システム・プラットフォームでどのビット順序を使用するかを決定できます (1 = ビッグ・エンディアン・ビット順、0 = リトル・エンディアン・ビット順)。

### ObjectScript

```
WRITE $SYSTEM.Version.IsBigEndian()
```

### 例

以下の例は、\$ZWPACK が 8 ビット文字を 2 つの 16 ビットのワイド文字にパックする方法を示します。パックした文字列のワイド文字では、バイト順序が 16 進数 4241 4443 のようにリトル・エンディアンになります。

### ObjectScript

```
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWPACK"
SET wstr=$ZWPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWUNPACK"
SET nstr=$ZWUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
```

以下の例は、\$ZWBPACK が 8 ビット文字を 2 つの 16 ビットのワイド文字にパックする方法を示します。パックした文字列のワイド文字では、バイト順序が 16 進数 4142 4344 のようにビッグ・エンディアンになります。

#### ObjectScript

```
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWBPACK"
SET wstr=$ZWBPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWBUNPACK"
SET nstr=$ZWBUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
```

以下の例は、string を検証してからパックします。

#### ObjectScript

```
SET str=$CHAR(65,66,67,68)
IF $ZISWIDE(str) {
    WRITE !,str," contains wide characters"
    QUIT }
ELSEIF $LENGTH(str) # 2 {
    WRITE !,str," contains an odd number of characters"
    QUIT }
ELSE {
    WRITE !,str," passes validation" }
WRITE !,$LENGTH(str)," characters: ",str
SET wstr=$ZWPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
```

## 関連項目

- ・ [\\$LENGTH 関数](#)
- ・ [\\$ZISWIDE 関数](#)
- ・ [\\$ZWUNPACK と \\$ZWBUNPACK 関数](#)



# \$ZWUNPACK および \$ZWBUNPACK (ObjectScript)

1 つの 16 ビット文字を 2 つの 8 ビット文字にアンパックします。

## 構文

```
$ZWUNPACK(string)
```

```
$ZWBUNPACK(string)
```

## 引数

引数	説明
string	1 つ以上の 16 ビット文字で構成される文字列。

## 概要

\$ZWUNPACK は 1 つ以上の 2 バイトのワイド文字を取得して、それらを“アンパック”し、対応するシングル・バイト文字のペアをリトル・エンディアン順に返します。

\$ZWBUNPACK は同じタスクを実行しますが、2 バイトのワイド文字はビッグ・エンディアン順にアンパックされます。

文字列のパックは、格納する文字列の文字数と文字列操作を半分にするための方法です。アンパックすると、元の 8 ビット文字の文字列を表示するために復元されます。これらの操作は、データ内で Unicode 文字が許可されているときは使用しないでください。

入力 string は、\$ZWPACK または \$ZWBPACK を使用して作成された 16 ビットのワイド文字のみで構成される必要があります。空の文字列は許可され、空の文字列が返されます。string には、16 ビット Unicode 文字および 8 ビット文字を含めないでください。

string で \$ZISWIDE を使用して、マルチバイト文字が含まれているかを確認できます。ただし、文字列に 16 ビット文字と 8 ビット文字が混在していないことを確認するには、各文字に対して \$ZISWIDE を使用する必要があります。\$ZISWIDE は、Unicode とパックされた 16 ビット文字を区別しません。

IsBigEndian() クラス・メソッドを使用して、オペレーティング・システム・プラットフォームでどのビット順序を使用するかを決定できます (1 = ビッグ・エンディアン・ビット順、0 = リトル・エンディアン・ビット順)。

## ObjectScript

```
WRITE $SYSTEM.Version.IsBigEndian()
```

## 例

以下の例では、\$ZWPACK を使用してパックされた文字列 (“ABCD”) をアンパックします。2 つの 16 ビットのワイド文字を 4 つの 8 ビット文字にアンパックします。パックした文字列のワイド文字では、バイト順序が 16 進数 4241 4443 のようにリトル・エンディアンになります。

## ObjectScript

```
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWPACK"
SET wstr=$ZWPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWUNPACK"
SET nstr=$ZWUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
```

以下の例は、前の例と同じ操作を実行しますが、ビッグ・エンディアン順序を使用します。パックした文字列のワイド文字では、バイト順序が 16 進数 4142 4344 のようにビッグ・エンディアンになります。

#### ObjectScript

```
SET str=$CHAR(65,66,67,68)
WRITE !,$LENGTH(str)," characters: ",str
WRITE !,"$ZWBPACK"
SET wstr=$ZWBPACK(str)
WRITE !,$LENGTH(wstr)," packed characters: ",wstr
ZZDUMP wstr
WRITE !,"$ZWBUNPACK"
SET nstr=$ZWBUNPACK(wstr)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
```

以下の例は、8ビット文字の文字列を“アンパック”するときにはどのようなようになるかを示しています。アンパック操作は、各文字が 16 ビットのワイド文字になることを前提にしています。したがって、不足している 8 ビットは 16 進数 00 として提供します。この \$ZWUNPACK の使用は、お勧めしません。

#### ObjectScript

```
SET str=$CHAR(65,66,67)
WRITE !,$LENGTH(str)," characters: ",str
SET nstr=$ZWUNPACK(str)
WRITE !,$LENGTH(nstr)," unpacked characters:"
ZZDUMP nstr
```

以下の例は、16 ビット Unicode 文字の文字列を“アンパック”するときにはどのようなようになるかを示しています。この場合は、小文字のギリシャ文字です。この \$ZWUNPACK の使用は、お勧めしません。

#### ObjectScript

```
SET str=$CHAR(945,946,947)
WRITE !,$LENGTH(str)," characters: ",str
SET nstr=$ZWUNPACK(str)
WRITE !,$LENGTH(nstr)," unpacked characters: ",nstr
ZZDUMP nstr
```

## 関連項目

- ・ [\\$LENGTH 関数](#)
- ・ [\\$ZISWIDE 関数](#)
- ・ [\\$ZWPACK および \\$ZWBPACK 関数](#)

# \$ZZENKAKU (ObjectScript)

日本語のカタカナを半角から全角に変換します。

## 構文

```
$ZZENKAKU(expression,flag1,flag2)
```

## 引数

引数	説明
expression	半角文字を含む文字列。この文字列の文字には、カタカナ、アルファベット、または数字を使用できます。
flag1	オプション - 半角のカタカナを全角のひらがな (0) あるいは全角のカタカナ (1) に変換するかどうかを示すブーリアン・フラグ。
flag2	オプション - 濁点処理が必要か (1) あるいは不要か (0) を指定するブーリアン・フラグ

## 概要

\$ZZENKAKU は、日本語のカタカナを半角から全角に変換します。また、アルファベット ("ABC") やアラビア数字 (123) の文字列を半角から全角に変換します。

必要に応じ、\$ZZENKAKU を使用して、半角カタカナを全角ひらがなに変換することもできます。カタカナは主に外来語に使用しますが、これらは半角文字でも全角文字でも表記できます。日本語を書くときには、カタカナよりもひらがなの方が広く使用されています。ひらがなは必ず全角です。

flag1 が 0 の場合、\$ZZENKAKU は、出力可能な ASCII 文字をそれに対応する全角文字に変換し、半角のカタカナを全角のひらがなに変換します。flag1 の既定値は 0 です。

flag1 が 1 の場合、\$ZZENKAKU は、出力可能な ASCII 文字をそれに対応する全角文字に変換し、半角のカタカナを全角のカタカナに変換します。

flag2 が 1 で、半角のカタカナの後に濁点または半濁点がある場合、\$ZZENKAKU は、(適宜) 半角のカタカナと濁点を目的の全角のひらがなまたはカタカナに組み合わせます。flag2 の既定値は 1 です。

**Config.NLS.Locales** クラスの **PhysicalCursor** プロパティを設定することで、システム全体の動作として 1 文字に 2 つの物理的スペースを使用するように物理カーソルを設定できます。

**\$WASCII** 関数 (およびその他の \$W 関数) は、日本語の漢字のエンコードに使用される文字のサロゲート・ペアをサポートします。ZFWCHARZ および ZHWKATAZ の日本語パターン・マッチ・コードについては、"[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。

## 例

次の例では、半角カタカナ文字 "ア"、"メ"、"リ"、"カ" (アメリカ) を返します。

### ObjectScript

```
ZZDUMP $CHAR(65383,65426,65432,65398)
```

次の例では、これらの半角カタカナ文字を、対応する全角カタカナ文字に変換します。

### ObjectScript

```
ZZDUMP $ZZENKAKU($CHAR(65383,65426,65432,65398),1)
```

次の例では両方とも、これらの半角カタカナ文字を、対応する全角ひらがな文字に変換します。\$ZZENKAKU は、既定ではカタカナからひらがなへの変換を行います。

#### ObjectScript

```
ZZDUMP $ZZENKAKU($CHAR(65383,65426,65432,65398),0)
```

#### ObjectScript

```
ZZDUMP $ZZENKAKU($CHAR(65383,65426,65432,65398))
```

## 関連項目

- ・ [\\$ZPOSITION](#) 関数
- ・ [\\$ZWIDTH](#) 関数

# ObjectScript 特殊変数

特殊変数とは、システムによって管理されている変数です。特殊変数はシステム変数とも呼ばれますが、構造化システム変数との混乱を避けるため、ここでは特殊変数と呼びます。

特殊変数名の先頭には、\$ が付いています。特殊変数には後に括弧が付かず、引数も持たないので、関数と区別されます。特殊変数名は、大文字と小文字が区別されません。多くの特殊変数名には省略形があります。各特殊変数の説明では、最初に完全名を使用した構文が示され、その下に略名を使用した構文が示されます（略名がある場合）。

元来から、特殊変数はスカラー値を保持してきました。システムは自動的に特殊変数を更新し、オペレーティング環境のさまざまな機能を反映します。例えば、\$IO 特殊変数は現在のデバイスの ID を含みます。\$JOB 特殊変数は、現在のジョブ ID を含みます。

複数の特殊変数を設定できますが、大半は読み取り専用です。この読み取り専用の制約を除き、その他の変数と同様に特殊変数を処理できます。例えば、式で特殊変数を参照したり、別の（ユーザ定義の）変数に現在の値を代入できます。

実装固有の特殊変数の形式は、それをサポートしているプラットフォームの略語と共に示されています（Windows や UNIX®）。プラットフォーム略語が付いていないものは、すべてのプラットフォームでサポートされています。

特殊変数はアルファベット順で示されます。

## \$DEVICE (ObjectScript)

---

ユーザ定義のデバイス状況情報を含みます。

### 構文

```
$DEVICE  
$D
```

### 概要

\$DEVICE は、デバイス状況情報を記録するために使用されます。SET コマンドを使用して、\$DEVICE の値を配置することができます。慣例により、この値は以下の形式で、3 つの文字列として入出力オペレーションの結果を説明します。

standard\_error,user\_error,explanatory\_text

既定で、\$DEVICE には NULL 文字列が含まれます。

### 関連項目

- ・ [SET コマンド](#)

## \$ECODE (ObjectScript)

現在のエラー・コード文字列を含みます。

### 構文

```
$ECODE
$EC
```

### 概要

エラー発生時、InterSystems IRIS では、エラーに対応するエラー・コードを含む文字列 (コンマで囲まれた文字列) に \$ECODE 特殊変数を設定します。例えば、未定義のグローバル変数に対する参照が作成されると、InterSystems IRIS は \$ECODE 特殊変数を以下の文字列に設定します。

```
,M7,
```

\$ECODE には、形式 M# (# は整数) の ISO 11756-1999 標準エラー・コードが含まれる場合があります。例えば、M6 と M7 は、それぞれ“未定義のローカル変数”および“未定義のグローバル変数”です (M7 は、グローバルおよびプロセス・プライベート・グローバルの両方に対して発行されます)。完全なリストについては、“[ISO 11756-1999 標準エラー・メッセージ](#)”を参照してください。

また、\$ECODE には、一般的なシステム・エラー・コード (ターミナル・プロンプトで \$ZERROR 特殊変数に返されるエラー・コード) と同じエラー・コードが含まれる場合があります。しかし、\$ECODE は、これらのエラー・コードに “Z” を追加し、山括弧を削除します。したがって、\$ECODE エラーの ZSYNTAX は <SYNTAX> エラーで、ZILLEGAL VALUE は <ILLEGAL VALUE> エラー、ZFUNCTION は <FUNCTION> エラーです。\$ECODE は、追加エラー情報を提供するこれらのエラー・コードの追加エラー情報を保持しません。したがって、ZPROTECT は <PROTECT> エラーとなります。追加情報コンポーネントは、\$ECODE ではなくて \$ZERROR に保持されます。InterSystems IRIS エラー・コードに関する詳細は、“[\\$ZERROR](#)”を参照してください。完全なリストは、“InterSystems IRIS エラー・リファレンス”の“[一般的なシステム・エラー・メッセージ](#)”を参照してください。

\$ECODE に既に以前のエラー・コードがあるときにエラーが発生すると、既存のエラー・スタックはその新しいエラーの発生時にクリアされます。新しいエラー・スタックには現在のエラー発生時の状態を示すエントリのみが含まれるようになります (これが初期の \$ECODE の動作からの変更点です。以前は、古いエラー・スタックは明示的にクリアされるまで維持されていました)。

複数のエラー・コードがある場合、InterSystems IRIS は各エラー・コードを受信した順に現在の \$ECODE 値の最後に追加します。結果として得られる \$ECODE 文字列の各エラーは、以下のようにコンマで区切られます。

```
,ZSTORE,M6,ZILLEGAL VALUE,ZPROTECT,
```

上の例では、最新のエラーは <PROTECT> エラーです。

他のすべてのルーチンまたはライブラリ関数を呼び出すときに \$ECODE が保持されていると仮定すべきではありません。エラーが発見・処理され、\$ECODE に異なる値が残される可能性があります。永続的なコピーが必要な場合は、\$ETRAP コードに保存する必要があります。

また、\$ECODE は明示的にクリア、あるいは設定できます。

### \$ECODE の削除

以下のように空の文字列に設定すると、\$ECODE を削除できます。

#### ObjectScript

```
SET $ECODE=""
```

\$ECODE を空の文字列に設定すると、以下のような効果があります。



- ・ 既存の \$ECODE 値をすべて削除します。既存の \$ZERROR 値には効果がありません。
- ・ ジョブのエラー・スタックを削除します。つまり、\$STACK 関数の後続の呼び出しは、最後のエラー・スタックではなく、現在の実行スタックを返します。
- ・ \$ETRAP エラー・ハンドラの制御のエラー処理フローに影響します。詳細は、“[TRY-CATCH の使用法](#)”を参照してください。

NEW を使用して、\$ECODE 特殊変数をリセットすることはできません。変更を試みると、〈SYNTAX〉エラーが生成されます。

## \$ECODE の設定

\$ECODE を空の文字列以外の値に設定すると、エラーを強制的に発生させることができます。\$ECODE を NULL 以外の値に設定すると、ObjectScript ルーチンの実行中にインタプリタ・エラーが強制的に発生します。InterSystems IRIS は、指定した NULL 以外値に \$ECODE を設定した後、以下の手順を実行します。

1. 指定された値を \$ECODE に書き込みます。以前の値はオーバーライドされます。
2. これは、〈ECODETRAP〉エラーを生成します (\$ZERROR に 〈ECODETRAP〉を設定します)。
3. 作成したエラー・ハンドラに制御を渡します。エラー・ハンドラは、選択した \$ECODE 文字列の値を確認し、状況に応じて適切に処理します。

## \$ECODE の文字列オーバーフロー

\$ECODE の蓄積文字列長が 512 文字を超えると、文字列オーバーフローの原因となるエラー・コードは、\$ECODE 内のエラー・コードの現在のリストを削除して置き換えます。この場合、\$ECODE のエラー・リストは、オーバーフローを発生させたエラーで始まる、最新の文字列オーバーフローのエラー・リストです。最大文字列データ長に関する詳細は、“[ObjectScript の使用法](#)”を参照してください。

## 独自のエラー・コードの作成

\$ECODE 特殊変数の ANSI 標準形式は、1 つ以上のエラー・コードのコンマで囲まれたリストです。文字 U で始まるエラー・コードは、ユーザ用に予約されています。その他すべてのエラー・コードは InterSystems IRIS 用に予約されています。

ユーザ定義の \$ECODE 値は、InterSystems IRIS が自動的に生成する値と区別できるようにする必要があります。そのためには、以下のようにエラー・テキストの最初に “U” を配置します。エラー・コードをコンマで記述することも忘れないでください。例えば以下ようになります。

### ObjectScript

```
SET $ECODE="U,Upassword expired!,"
```

## \$ECODE ではなく \$ZERROR を確認

エラー・ハンドラでは、最新の InterSystems IRIS エラーについて、\$ECODE ではなく \$ZERROR を確認する必要があります。

## 関連項目

- ・ [ZTRAP](#) コマンド
- ・ [\\$STACK](#) 関数
- ・ [\\$ESTACK](#) 特殊変数
- ・ [\\$ZEOF](#) 特殊変数
- ・ [\\$ZERROR](#) 特殊変数

- ・ [\\$ZTRAP 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)
- ・ [システム・エラー・メッセージ](#)

## \$ESTACK (ObjectScript)

ユーザ定義のポイントから、コール・スタック上に保存されている、コンテキスト・フレーム数を含みます。

### 構文

```
$ESTACK  
$ES
```

### 概要

\$ESTACK には、ユーザ定義のポイントから、ジョブのコール・スタック上に保存されている、コンテキスト・フレーム数が含まれます。このポイントを指定するには、NEW コマンドを使用して \$ESTACK の新しいコピーを作成します。

\$ESTACK 特殊変数は、\$STACK 特殊変数と類似しています。どちらの変数にも、ジョブまたはプロセスのコール・スタックに現在保存されているコンテキスト・フレームの数が格納されます。コンテキストが変更されると、両方の変数がインクリメントされ、復元されます。主な違いは、NEW コマンドを使用することで、いつでも \$ESTACK のカウントをゼロにリセットすることができる点です。\$STACK のカウントをリセットすることはできません。

### コンテキスト・フレームとコール・スタック

InterSystems IRIS イメージが開始すると、コンテキストがコール・スタックに保存される前は、\$ESTACK と \$STACK の値は 0 です。DO でルーチンが別のルーチン呼び出すたびに、現在実行中のルーチンのコンテキストがコール・スタックに保存され、\$ESTACK と \$STACK がインクリメントされて、新しく作成されたコンテキストで、呼び出されたルーチンの実行が開始します。呼び出されたルーチンは、順番に別のルーチン呼び出すことができます。別のルーチンが呼び出されるたびに、\$ESTACK と \$STACK がインクリメントされ、コール・スタックに保存されたコンテキストが増えていきます。

DO コマンド、XECUTE コマンド、またはユーザ定義関数への呼び出しを発行すると、新しい実行コンテキストが作成されます。GOTO コマンドを発行しても、新しいコンテキストは作成されません。

DO コマンド、XECUTE コマンド、またはユーザ定義関数参照によって新しいコンテキストが作成されると、\$STACK と \$ESTACK の値がインクリメントされます。コンテキストが QUIT コマンドで終了するたびに、前のコンテキストはコール・スタックから戻され、\$STACK と \$ESTACK の値はデクリメントされます。

\$ESTACK および \$STACK 特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

### 新規の \$ESTACK の作成

NEW コマンドを使用して、コンテキストで \$ESTACK の新しいコピーを作成できます。InterSystems IRIS は以下を実行します。

1. \$ESTACK の古いコピーを保存します。
2. 値 0 を使用して、\$ESTACK の新規コピーを作成します。

この方法で、\$ESTACK レベル 0 コンテキストとして、特定のコンテキストを作成することができます。DO、XECUTE、またはユーザ定義関数によって新しいコンテキストが作成されると、この \$ESTACK の値がインクリメントされます。しかし、新しい \$ESTACK が作成された (レベル 0 の \$ESTACK) コンテキストを終了すると、\$ESTACK の前のコピーの値が復元されます。

### 例

次の例は、\$ESTACK に対する NEW コマンドの影響を示しています。この例では、MainRoutine によって \$STACK と \$ESTACK の初期値 (同一の値) が表示されます。続いて、Sub1 が呼び出されます。この呼び出しによって、\$STACK と \$ESTACK がインクリメントされます。NEW コマンドによって、値 0 の \$ESTACK が作成されます。Sub1 によって Sub2

が呼び出されて、\$STACK と \$ESTACK がインクリメントされます。MainRoutine に戻ると、\$STACK と \$ESTACK の初期値がリストアされます。

## ObjectScript

```
Main
WRITE !,"Initial: $STACK=", $STACK, " $ESTACK=", $ESTACK
DO Sub1
WRITE !,"Return: $STACK=", $STACK, " $ESTACK=", $ESTACK
QUIT
Sub1
WRITE !,"Sub1Call: $STACK=", $STACK, " $ESTACK=", $ESTACK
NEW $ESTACK
WRITE !,"Sub1NEW: $STACK=", $STACK, " $ESTACK=", $ESTACK
DO Sub2
QUIT
Sub2
WRITE !,"Sub2Call: $STACK=", $STACK, " $ESTACK=", $ESTACK
QUIT
```

次の例は、DO および XECUTE コマンドを発行することによって新しいコンテキストが作成されたときに \$ESTACK の値がどのようにインクリメントし、それらのコンテキストが終了したときにどのようにデクリメントするかを示しています。また、GOTO コマンドでは新しいコンテキストの作成も \$ESTACK のインクリメントも行われないことも示しています。

## ObjectScript

```
Main
NEW $ESTACK
WRITE !,"Initial Main: $ESTACK=", $ESTACK // 0
DO Sub1
WRITE !,"Return Main: $ESTACK=", $ESTACK // 0
QUIT
Sub1
WRITE !,"Sub1 via DO: $ESTACK=", $ESTACK // 1
XECUTE "WRITE !,\"Sub1 XECUTE: $ESTACK=", $ESTACK" // 2
WRITE !,"Sub1 post-XECUTE: $ESTACK=", $ESTACK // 1
GOTO Sub2
Sub1Return
WRITE !,"Sub1 after GOTO: $ESTACK=", $ESTACK // 1
QUIT
Sub2
WRITE !,"Sub2 via GOTO: $ESTACK=", $ESTACK // 1
GOTO Sub1Return
```

## ターミナル・プロンプトから呼び出す場合のコンテキスト・レベル

プログラムから呼び出されるルーチンは、ターミナル・プロンプトから DO コマンドを使用して呼び出すルーチンとは異なるコンテキスト・レベルで開始します。ターミナル・プロンプトで DO コマンドを入力すると、呼び出すルーチンの新しいコンテキストが作成されます。

呼び出すルーチンは、\$ESTACK レベル 0 コンテキストを作成してから、すべてのコンテキスト・レベル参照に \$ESTACK を使用することによって、補正することができます。

以下のルーチンを考慮してみましょう。

## ObjectScript

```
START
; Establish a $ESTACK Level 0 Context
NEW $ESTACK
; Display the $STACK context level
WRITE !,"$STACK level in routine START is ", $STACK
; Display the $ESTACK context level and exit
WRITE !,"$ESTACK level in routine START is ", $ESTACK
QUIT
```

プログラムから START を実行すると、以下が表示されます。

```
$STACK level in routine START is 0
$ESTACK level in routine START is 0
```

ターミナル・プロンプトで DO ^START を発行して START を実行すると、以下が表示されます。

```
$STACK level in routine START is 1
$ESTACK level in routine START is 0
```

## \$ESTACK とエラー処理

\$ESTACK は特に、エラー・ハンドラが特定のコンテキスト・レベルにコール・スタックを巻き戻す必要があるときのエラー処理中に利用できます。エラー処理の詳細は、“[TRY-CATCH の使用法](#)”を参照してください。

## 関連項目

- ・ [\\$STACK 関数](#)
- ・ [\\$STACK 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)
- ・ [スタックを表示する %STACK の使用法](#)

# \$ETRAP (ObjectScript)

エラー発生時に実行する ObjectScript コマンドの文字列を含みます。

## 構文

```
$ETRAP
$ET

SET $ETRAP="cmdline"
SET $ET="cmdline"
```

## 概要

\$ETRAP には、エラーの発生時に実行される 1 つ以上の ObjectScript コマンドを指定する文字列が含まれます。

**注釈** 新しいアプリケーション・コードでのエラー処理では、[TRY](#) および [CATCH](#) コマンドを使用することを強くお勧めします。ここでは、既存のコードのメンテナンスおよび移行をサポートする \$ETRAP について記載します。詳細は、["TRY-CATCH の使用法"](#) を参照してください。

SET コマンドを使用して、\$ETRAP に、1 つ以上の ObjectScript コマンドを含む cmdline 文字列の値を指定します。エラーが発生すると、InterSystems IRIS はユーザが \$ETRAP に入力したコマンドを実行します。例えば、\$ETRAP を、エラー処理ルーチンに制御を移す GOTO コマンドを含む文字列に設定するとします。

### ObjectScript

```
SET $ETRAP="GOTO LOGERR^ERRROU"
```

InterSystems IRIS は、エラー状態を生成する任意の ObjectScript コマンドの直後に、\$ETRAP のこのコマンドを実行します。InterSystems IRIS は、エラー状態が発生するコンテキスト・レベルで \$ETRAP コマンドを実行します。InterSystems IRIS は、\$ETRAP が設定されている場合、有効な [\\$ROLES](#) 値を保存します。\$ETRAP コードが実行されると、InterSystems IRIS は \$ROLES を保存されたその値に設定します。これにより、\$ETRAP エラー・ハンドラは、エラー・ハンドラの設定後にルーチンに付与された上位特権を使用しないようにします。

エラー・ハンドラを実行するように \$ETRAP を設定する場合 (GOTO コマンドを使用するなど)、このエラー・ハンドラは、label (現在のルーチンの[ラベル](#))、^routine (指定した外部ルーチンの先頭)、または label^routine (指定した外部ルーチンの指定したラベル) として指定できます。

\$ETRAP は、一部のコンテキスト (プロシージャ以外) で label+offset をサポートしています。オプションのこの +offset は、label からオフセットする行数を指定する整数です。インターシステムズでは、エラー・ハンドラの場合の指定時には行のオフセットを使用しないことを推奨します。

\$ETRAP の cmdline 文字列値が実行可能 ObjectScript コマンドであるため、文字列の長さを ObjectScript ルーチン行の最大長さ (32,741 文字) より長くすることはできません。\$ETRAP をこれより長い文字列に設定すると、<MAXSTRING> エラーになる可能性があります。cmdline 文字列の指定の詳細は、["XECUTE"](#) コマンドを参照してください。

## \$ETRAP を新しい値に設定する前の NEW の使用

NEW コマンドで最初に \$ETRAP の新しいコピーを作成しないコンテキストで、新しい値を \$ETRAP に割り当てる場合、InterSystems IRIS は、現在のコンテキストだけでなく、前のすべてのコンテキストにも、その新しい値を \$ETRAP の値として設定します。したがって、新しい値で \$ETRAP を設定する前に、NEW \$ETRAP コマンドを使用して \$ETRAP の新しいコピーを作成することを強くお勧めします。

## XECUTE コマンドと比較した \$ETRAP コマンド

\$ETRAP 文字列内のコマンドは、[XECUTE](#) 文字列でのコマンドとは異なり、新しいコンテキスト・レベルでは実行されません。また、\$ETRAP コマンド文字列は、常に暗黙的な QUIT コマンドにより終了されます。引数付きの QUIT コマンド

を必要とするユーザ定義の関数のコンテキストで \$ETRAP エラー処理コマンドが呼び出されると、暗黙的な QUIT コマンドは NULL 文字列の引数で終了します。

## 異なるコンテキスト・レベルでの \$ETRAP 値の設定

InterSystems IRIS は、既定では、\$ETRAP 特殊変数の値を新しい DO、XECUTE、およびユーザ定義関数コンテキストに伝送します。しかし、以下のように、NEW コマンドを発行することで、コンテキストで \$ETRAP の新しいコピーを作成できます。

### ObjectScript

```
NEW $ETRAP
```

\$ETRAP に対して NEW を発行すると、InterSystems IRIS は必ず次のアクションを実行します。

1. その時点で使用中の \$ETRAP のコピーを保存します。
2. \$ETRAP の新しいコピーを作成します。
3. \$ETRAP の新しいコピーに、保存されている古い \$ETRAP のコピーと同じ値を割り当てます。

ユーザは、その後、SET コマンドを使用して、\$ETRAP の新しいコピーに別の値を割り当てます。このようにして、現在のコンテキストで新しい \$ETRAP エラー処理コマンドを設定できます。

NULL 文字列に設定することで、\$ETRAP をクリアすることもできます。その後、InterSystems IRIS は、エラーの発生するイベントのコンテキスト・レベルで \$ETRAP コマンドを実行しなくなります。

QUIT コマンドにより現在のコンテキストが終了されると、InterSystems IRIS は、保存されている古い \$ETRAP の値をリストアします。

## 例

以下の例では \$ETRAP または \$ZTRAP を使用して同じ操作を実行します。〈DIVIDE〉エラーが発生したら、ErrMod に進みます。\$RANDOM 関数はランダムにいずれかのエラー・ハンドラを呼び出します。

### ObjectScript

```
MainMod
WRITE "MainMod stack:",$STACK,!
SET x=$RANDOM(2)
IF x=0 {WRITE "$ETRAP",!
        NEW $ETRAP
        SET $ETRAP="GOTO ErrMod"
        WRITE 5/0}
IF x=1 {WRITE "$ZTRAP",!
        SET $ZTRAP="ErrMod"
        WRITE 5/0}
QUIT
ModuleA
/* code not executed */
QUIT
ErrMod
WRITE !,"in ErrMod",!
WRITE "ErrMod stack:",$STACK
QUIT
```

以下の例では、\$ETRAP の値を新しいコンテキストに伝送する方法、およびエラーの発生後に各コンテキストで再度 \$ETRAP エラー処理コマンドを呼び出す方法を示します。この例の \$ETRAP コマンドは、エラーの破棄を試みません。むしろ、既定で制御が前の各コンテキスト・レベルの \$ETRAP エラー処理コマンドに返されます。



サンプル・コードは以下のとおりです。

```

ETR
  NEW $ETRAP
  SET $ETRAP="WRITE !,""$ETRAP invoked at Context Level ","$STACK"
    ; Initiate an XECUTE context that initiates a DO context
  XECUTE "DO A"
  QUIT
    ; Initiate a user-defined function context
A
  WRITE "In A",!
  SET A=$$B
  QUIT
B()
  ; User-defined function that generates an error
  WRITE "In B",!
  WRITE "impossible division ",5/0
  QUIT 1

```

このコードを使用したサンプルの Terminal セッションは、次のように実行されます。

### Terminal

```

USER>DO ^ETR
In A
In B
impossible division
$ETRAP invoked at context level 4
$ETRAP invoked at context level 3
$ETRAP invoked at context level 2
$ETRAP invoked at context level 1
USER>

```

## \$ETRAP とその他の ObjectScript エラー処理機能

\$ETRAP 特殊変数は、アプリケーションで発生するエラーの処理およびログ記録の制御を可能にする、いくつかの ObjectScript 言語機能の 1 つです。

- ・ エラー処理で推奨する InterSystems IRIS の機能は、ブロック構造の [TRY](#) および [CATCH](#) コマンドです。
- ・ [\\$ZTRAP](#) 特殊変数は、\$ETRAP よりもお勧めです。
- ・ \$ETRAP は、InterSystems IRIS で引き続きサポートされます。しかし、一般的には、新しいコードで \$ETRAP を他のエラー処理機能より優先して使用することは避けてください。

エラー処理の詳細は、"[TRY-CATCH の使用法](#)" を参照してください。

### \$ETRAP と \$ZTRAP

\$ZTRAP を使用してエラー・ハンドラを設定する場合、このハンドラは既存のどの \$ETRAP エラー・ハンドラよりも優先されます。InterSystems IRIS は暗黙的に `NEW $ETRAP` コマンドを実行し、\$ETRAP を NULL 文字列("") に設定します。

### \$ETRAP および TRY / CATCH

TRY コマンドと CATCH コマンドは実行レベル内でエラー処理を行います。TRY ブロック内で例外が発生すると、InterSystems IRIS は通常、TRY ブロックの直後に続く例外処理コードの CATCH ブロックを実行します。

**注釈** TRY ブロックで構成されたプログラム内では \$ETRAP を使用しないことを強くお勧めします。

TRY ブロック内で \$ETRAP を設定することはできません。これを試みると、コンパイル・エラーが生成されます。\$ETRAP は、TRY ブロックの前、または CATCH ブロック内で設定できます。

以前に \$ETRAP が設定されており、TRY ブロックで例外が発生した場合、InterSystems IRIS はこの可能性を予想していない限り、CATCH ではなく \$ETRAP を取得します。\$ETRAP と CATCH の両方が存在する場合、InterSystems IRIS は現在の実行レベルに適用されるエラー・コード (CATCH または \$ETRAP) を実行します。\$ETRAP は本来実行レベルとは関連付けられないため、InterSystems IRIS では、他に指定されていない限り現在の実行レベルに関連付けられていると見なします。\$ETRAP を設定して \$ETRAP のレベル・マーカを設定する前に、`NEW $ETRAP` を実行して、

InterSystems IRIS が現在のレベルの例外ハンドラとして \$ETRAP ではなく CATCH を取得するようにする必要があります。そうしないと、システム・エラー (THROW コマンドによりスローされるシステム・エラーを含む) により \$ETRAP 例外ハンドラが取得される可能性があります。

CATCH ブロック内で発生する例外は、現在のエラー・トラップ・ハンドラによって処理されます。

## 関連項目

- ・ [NEW コマンド](#)
- ・ [SET コマンド](#)
- ・ [THROW コマンド](#)
- ・ [TRY コマンド](#)
- ・ [\\$ECODE 特殊変数](#)
- ・ [\\$ZEOF 特殊変数](#)
- ・ [\\$ZTRAP 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)

# \$HALT (ObjectScript)

停止トラップ・ルーチン呼び出しを含みます。

## 構文

```
$HALT
```

## 概要

\$HALT には、現在の停止トラップ・ルーチンの名前が含まれます。停止トラップ・ルーチンとは、HALT コマンドが見つかったときにアプリケーションによって呼び出されるルーチンです。この停止トラップ・ルーチンは、HALT コマンドを発行する前に削除またはログ処理を実行します。また、プログラムの実行を停止する代わりに他の処理を実行することもあります。

\$HALT を停止トラップ・ルーチンに設定するには、SET コマンドを使用します。停止トラップ・ルーチンは、引用符で囲まれた文字列を使用して次の形式で指定します。

```
SET $HALT=location
```

ここでは、location は label (現在のルーチンまたはプロシージャ内のラベル)、^routine (指定された外部ルーチンの開始)、または label^routine (指定された外部ルーチン内の指定されたラベル) として指定できます。

\$HALT は、一部のコンテキスト (プロシージャ以外) で label+offset をサポートしています。オプションのこの +offset は、label からオフセットする行数を指定する整数です。インターシステムズでは、location の指定時には行のオフセットを使用しないことを推奨します。

プロシージャまたは IRISYS % ルーチンの呼び出し時における +offset の指定はできません。指定しようとすると、InterSystems IRIS は <NOLINE> エラーを発行します。

\$HALT は、現在のコンテキストの停止トラップ・ルーチンを定義します。現在のコンテキストに停止トラップが既に定義されている場合は、新しいものに置き換えられます。存在しないルーチン名を指定した場合、HALT コマンドはその \$HALT を無視し、スタックを戻して、前のコンテキスト・レベルの有効な \$HALT を探します。

現在のコンテキストの停止トラップを削除するには、\$HALT を NULL 文字列に設定します。NEW コマンドや KILL コマンドを使用して停止トラップを削除しようと試みると、<SYNTAX> エラーが返されます。

## 停止トラップの実行

HALT コマンドを発行すると、現在のコンテキストで \$HALT がチェックされます。現在のコンテキストに \$HALT が定義されていない場合 (存在しないルーチン名または NULL 文字列に設定されている場合)、前のコンテキストにスタックが戻され、そこで \$HALT が検索されます。このプロセスは、定義された \$HALT が見つかるか、またはスタックが完全に戻されるまで続行されます。InterSystems IRIS では、\$HALT の値を使用して、指定された停止トラップ・ルーチンに実行が転送されます。停止トラップ・ルーチンは、\$HALT が定義されたコンテキストで実行されます。エラー・コードは設定されず、エラー・メッセージも発行されません。

現在のコンテキストでも前のコンテキストでも有効な \$HALT が設定されていない場合、HALT コマンドを発行すると、スタックが完全に戻され、実際のプログラム停止が実行されます。

停止トラップ・ルーチンでは一般に、削除またはレポート処理が実行された後、HALT コマンドが発行されます。定義された \$HALT では、元の HALT コマンドによって停止トラップが呼び出されますが、実際のプログラム停止は実行されません。実際の停止を実行するには、停止トラップ・ルーチンに別の HALT コマンドを含める必要があります。

停止トラップ・ルーチンによって発行された HALT コマンドは、その停止トラップではトラップされません。ただし、下位のコンテキスト・レベルで設定された停止トラップでトラップすることができます。そのため、階層式の一連の停止トラップを単独の HALT コマンドで呼び出すことができます。

同様の処理が、エラー・トラップの [ZTRAP](#) コマンドと、関連する [\\$ZTRAP](#) または [\\$ETRAP](#) 特殊変数によって実行されます。

## \$HALT と ^%ZSTOP

\$HALT が設定されていて、さらに ^%ZSTOP にコードが定義されている場合に HALT が発行されると、\$HALT が最初 to 実行されます。\$HALT は、停止トラップ・ルーチンに HALT コマンドが含まれていない場合、プロセスの終了を防ぐことができます。

^%ZSTOP ルーチンは、プロセスが実際に終了しているときに実行されます。^%ZSTOP の詳細は、"[^%ZSTART および ^%ZSTOP ルーチンの使用法](#)" を参照してください。

## 例

以下の例は、\$HALT を使用して停止トラップを設定します。

### ObjectScript

```
SET $HALT="MyTrap^CleanupRoutine"
WRITE !,"the halt trap is: ",$HALT
```

指定したルーチンが存在するかどうかは、プログラマが確認する必要があります。

次の例は、\$HALT が定義されたコンテキストで停止トラップ・ルーチンがどのように実行されるかを示しています。この例では、\$HALT が \$ESTACK レベル 0 で定義され、HALT が \$ESTACK レベル 1 で発行され、停止トラップ・ルーチンが \$ESTACK レベル 0 で実行されます。

### ObjectScript

```
Main
  NEW $ESTACK
  SET $HALT="OnHalt"
  WRITE !,"Main $ESTACK= ",$ESTACK," $HALT= ",$HALT    // 0
  DO SubA
  WRITE !,"Returned from SubA"    // not executed
  QUIT
SubA
  WRITE !,"SubA $ESTACK= ",$ESTACK," $HALT= ",$HALT    // 1
  HALT
  WRITE !,"this should never display"
  QUIT
OnHalt
  WRITE !,"OnHalt $ESTACK= ",$ESTACK    // 0
  HALT
  QUIT
```

次の例は、\$HALT が \$ESTACK レベル 1 で定義されている点を除き、前の例と同じです。HALT コマンドは \$ESTACK レベル 1 で発行され、停止トラップ・ルーチンは \$ESTACK レベル 1 で実行されます。停止トラップ・ルーチンによって発行された HALT はスタックを戻しますが、前のコンテキスト・レベルで定義された \$HALT が見つからないため、プログラム実行を停止します。そのため、DO コマンドの後に続く WRITE コマンドは実行されません。

### ObjectScript

```
Main
  NEW $ESTACK
  WRITE !,"Main $ESTACK= ",$ESTACK," $HALT= ",$HALT    // 0
  DO SubA
  WRITE !,"Returned from SubA"    // not executed
  QUIT
SubA
  SET $HALT="OnHalt"
  WRITE !,"SubA $ESTACK= ",$ESTACK," $HALT= ",$HALT    // 1
  HALT
  WRITE !,"this should never display"
  QUIT
OnHalt
  WRITE !,"OnHalt $ESTACK= ",$ESTACK    // 1
  HALT
  QUIT
```

次の例は、階層式の一連の停止トラップを呼び出す方法を示しています。停止トラップ Halt0 は \$ESTACK レベル 0 で定義され、停止トラップ Halt1 は \$ESTACK レベル 1 で定義されます。HALT コマンドは \$ESTACK レベル 2 で発行されます。InterSystems IRIS はスタックを戻し、停止トラップ Halt1 を \$ESTACK レベル 1 で呼び出します。この停止トラップは HALT コマンドを発行します。InterSystems IRIS はスタックを戻し、停止トラップ Halt0 を \$ESTACK レベル 0 で呼び出します。この停止トラップは、プログラムの実行を停止する HALT コマンドを発行します。

## ObjectScript

```
Main
  NEW $ESTACK
  SET $HALT="Halt0"
  WRITE !,"Main $ESTACK= ", $ESTACK, " $HALT= ", $HALT    // 0
  DO SubA
  WRITE !,"Returned from SubA"    // not executed
  QUIT
SubA
  SET $HALT="Halt1"
  WRITE !,"SubA $ESTACK= ", $ESTACK, " $HALT= ", $HALT    // 1
  DO SubB
  WRITE !,"Returned from SubA"    // not executed
  QUIT
SubB
  WRITE !,"SubB $ESTACK= ", $ESTACK, " $HALT= ", $HALT    // 2
  HALT
  WRITE !,"this should never display"
  QUIT
Halt0
  WRITE !,"Halt0 $ESTACK= ", $ESTACK    // 0
  WRITE !,"Bye-bye!"
  HALT
  QUIT
Halt1
  WRITE !,"Halt1 $ESTACK= ", $ESTACK    // 1
  HALT
  QUIT
```

## 関連項目

- [HALT コマンド](#)

## \$HOROLOG (ObjectScript)

現在のプロセスのローカルな日付と時刻を含みます。

### 構文

```
$HOROLOG
$H
```

### 概要

\$HOROLOG には現在のプロセスの日付と時刻を記述します。以下の値を扱うことができます。

- ・ 現在のローカル日付とローカル時刻
- ・ タイム・ゾーンのオフセットに合わせて調整した現在のローカル日付とローカル時刻。
- ・ ユーザ指定のインクリメントされない日付。時刻は引き続き現在のローカル時刻です。

\$HOROLOG は、コンマで区切られた 2 つの整数値から成る文字列を含みます。これらの 2 つの整数は、InterSystems IRIS に格納する形式で現在のローカル日付とローカル時刻を表します。これらの整数はカウンタで、外部でユーザが読み取れる日付と時刻ではありません。\$HOROLOG は、次の形式で現在の日付と時刻を返します。

```
dddddd,sssss
```

最初の整数 ddddd は、1840 年 12 月 31 日からの現在の経過日数です。つまり、日 1 は 1841 年 1 月 1 日です。InterSystems IRIS は任意の開始ポイントからのカウンタを使用して日付を表すので、2000 年問題の影響は受けません。この日付整数の最大値は 2980013 で、9999 年 12 月 31 日に対応します。

2 番目の整数 sssss は現在の時刻で、その日の午前 0 時 00 分からの秒数で表されます。システムは、0 から 86399 秒で時間フィールドをインクリメントします。深夜 12 時に 86399 に到達すると、システムは時間フィールドを 0 にリセットしてから、日付フィールドを 1 つインクリメントします。\$HOROLOG では秒の小数部分が切り捨てられ、整数秒のみで表されます。

以下のように、Horolog() メソッドを呼び出すことで、同じ現在の日付と時刻を取得することができます。

#### ObjectScript

```
WRITE $SYSTEM.SYS.Horolog()
```

詳細は、“インターシステムズ・クラス・リファレンス” の %SYSTEM.SYS を参照してください。

### 日付と時刻の分割

\$HOROLOG の日付部分または時刻部分のみを取得するには、区切り文字としてコンマを指定して、[\\$PIECE](#) 関数を使用します。

#### ObjectScript

```
SET dateint=$PIECE($HOROLOG,",",1)
SET timeint=$PIECE($HOROLOG,",",2)
WRITE !,"Date and time: ",$HOROLOG
WRITE !,"Date only: ",dateint
WRITE !,"Time only: ",timeint
```

\$HOROLOG 値の日付部分のみを取得する場合は、次のプログラミング方法を使用することもできます。

## ObjectScript

```
SET dateint=+$HOROLOG
WRITE !,"Date and time: ",$HOROLOG
WRITE !,"Date only: ",dateint
```

プラス記号 (+) によって、\$HOROLOG 文字列が数値として解析されます。数値以外の文字 (コンマ) が検出されると、残りの文字列は切り捨てられ、数値部分が返されます。これが、文字列の日付整数部分です。

## 日付/時刻関数の比較

現在の日付と時刻は、さまざまな方法で返すことができますが、その違いを以下に示します。

- ・ \$HOROLOG には、ローカル調整された日付と時刻が InterSystems IRIS ストレージ形式で格納されます。ローカル・タイム・ゾーンは、\$ZTIMEZONE 特殊変数の現在の値で決まり、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。これは、整数秒のみを返し、小数部分は切り捨てられます。
- ・ \$NOW は、現在のプロセスのローカルな日付と時刻を返します。\$NOW は、日付と時刻を InterSystems IRIS ストレージ形式で返します。これには、秒の小数部が含まれます。小数桁数は、現在のオペレーティング・システムでサポートされている最大精度の桁数です。
  - \$NOW は、\$ZTIMEZONE 特殊変数の値からローカル・タイム・ゾーンを決定します。ローカル時刻は、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せん。したがって、ローカルな時刻に対応しない場合があります。
  - \$NOW(tzmins) は、指定された tzmins タイム・ゾーン・パラメータに対応する時刻と日付を返します。\$ZTIMEZONE の値は無視されます。
- ・ \$ZTIMESTAMP には、秒の小数部を含む UTC (協定世界時) の日付と時刻が InterSystems IRIS ストレージ形式で格納されます。秒の小数部は、Windows システムでは有効桁数 3 桁で、UNIX® システムでは有効桁数 6 桁で表されます。

## 日付・時刻の変換

\$ZDATE 関数を使用して、外部でユーザが読み取れる形式に \$HOROLOG の日付部分を変換できます。\$ZTIME 関数を使用して、外部でユーザが読み取れる形式に \$HOROLOG の時刻部分を変換できます。\$ZDATETIME 関数を使用して、日付と時刻の両方を変換できます。\$HOROLOG を使用した場合、これらの関数の時刻値の precision を設定すると、秒の小数部として常にゼロが返されます。

\$ZDATEH 関数を使用して、外部でユーザが読み取れる形式に \$HOROLOG の日付部分を変換できます。\$ZTIMEH 関数を使用して、外部でユーザが読み取れる形式に \$HOROLOG の時刻部分を変換できます。\$ZDATETIMEH 関数を使用して、\$HOROLOG 値の日付と時刻の両方を変換できます。

## 日付と時刻の設定

\$HOROLOG は、%SYSTEM.Process クラスの FixedDate() メソッドを使用して、現在のプロセスのユーザ指定の日付に設定することができます。SET コマンドを使用して、\$HOROLOG を変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## ObjectScript

```
DO ##class(%SYSTEM.Process).FixedDate(12345) // set $HOROLOG date
WRITE !,$ZDATETIME($HOROLOG,1,1,9)," $HOROLOG changed date"
WRITE !,$ZDATETIME($NOW(),1,1,9)," $NOW() no date change"
WRITE !,$ZDATETIME($ZDATETIMEH($ZTIMESTAMP,-3),1,1,9)," $ZTS UTC-to-local",
    " no date change"
DO ##class(%SYSTEM.Process).FixedDate(0) // restore $HOROLOG
WRITE !,$ZDATETIME($HOROLOG,1,1,9)," $HOROLOG current date"
```

FixedDate() は \$HOROLOG 値を変更しますが、\$NOW または \$ZTIMESTAMP の値は変更しないことに注意してください。



## タイム・ゾーン

既定では、\$HOROLOGY はローカル・タイム・ゾーンの日付と時刻を記述します。この既定のタイム・ゾーンはオペレーティング・システムで指定します。InterSystems IRIS では、この既定の設定を使用して \$ZTIMEZONE の既定値を設定します。

\$ZTIMEZONE を変更すると、現在のプロセスの \$HOROLOGY の値に影響します。これは、\$HOROLOGY の時間部分を変更し、またこの変更が \$HOROLOGY の日付部分を変更します。\$ZTIMEZONE はグリニッジ子午線からのタイム・ゾーンの固定オフセットです。サマータイムなどの季節的なローカル時刻調整には対応していません。

## サマータイム

\$HOROLOGY は、基本となるオペレーティング・システムが提供するアルゴリズムに基づいて、季節的な時刻調整に対応します。\$ZTIMEZONE 値を適用すると、InterSystems IRIS はオペレーティング・システムのローカル時刻を使用して、サマータイムなどの季節的な時刻調整に使用するために（必要に応じて）\$HOROLOGY を調整します。

現在の日付または指定した日時に対してサマータイムが適用されているかどうかを確認するには、IsDST() メソッドを使用します。以下の例は、現在の日時のサマータイム (DST) の状態を返します。この状態はプログラムの実行中に変化する可能性があるため、この例では 2 回確認しています。

### ObjectScript

```
CheckDST
SET x=$SYSTEM.Util.IsDST()
SET local=$ZDATETIME($HOROLOGY)
SET x2=$SYSTEM.Util.IsDST()
GOTO:x'=x2 CheckDST
IF x=1 {WRITE local," DST in effect"}
ELSEIF x=0 {WRITE local," DST not in effect"}
ELSE {WRITE local," DST setting cannot be determined"}
```

季節的な時刻調整の適用は、(少なくとも) 以下の 3 つの考慮事項によって異なります。

- ・ オペレーティング・システム：タイム・ゾーンが同一でも、コンピュータが異なれば、指定した日付に対する \$HOROLOGY も異なる可能性があります。これは、オペレーティング・システムによって、使用する時刻調整の適用アルゴリズムが異なるためです。サマータイムの開始日と終了日（およびその他の時刻調整）を管理する制度が変更されているので、古いオペレーティング・システムには現在の実態が反映されていない可能性があります。したがって、過去の \$HOROLOGY 値を使用する計算では、その時点で効力のあった開始日と終了日ではなく、現在の開始日と終了日を使用して調整が行われる可能性があります。
- ・ 各国政策の変更：季節的な時刻調整は、大半の欧州諸国では 1916 年、米国では 1918 年に採用されて以来、数多くの変更が行われてきました。サマータイムはさまざまな地域で、政府方針により採用され、廃止され、再び採用されています。サマータイムの開始日および終了日も何度となく変更されています。米国では、最近だけでも 1966 年、1974 ~ 1975 年、1987 年、および 2007 年に国内政策が変更されています。また、地域の法的処置により、国内政策が採用されることもあれば、不採用になることもあります。例えば、アリゾナ州ではサマータイムは導入されていません。
- ・ 地理的条件：サマータイム。DST の開始日にローカル時計を進め（“Spring ahead”）、DST の終了日に遅らせます（“Fall back”）。したがって、タイムゾーンが同一でも、北半球と南半球とではサマータイムの開始日と終了日が逆になることが普通です。また、赤道付近の国々や、アジアおよびアフリカの大半の国では、サマータイムは導入されていません。

## ローカル時刻調整のしきい値

\$HOROLOGY では、システム時計を参照して午前 0 時からの秒数を計算します。このため、サマータイムの開始や終了時など、ローカル時刻調整しきい値を超えたときにシステム時計が自動的に再設定されると、\$HOROLOGY の時刻値も該当する秒数だけ前または後に突然変更されます。したがって、2 つの \$HOROLOGY 時刻値を比較した場合、2 つの値の間の期間にローカル時刻調整しきい値が含まれていると、予期しない結果が生じることがあります。



\$NOW ではローカル時刻調整に合わせた調整を行いません。日付および時刻の値を比較する際に、2 つの値の間の期間にローカル時刻調整しきい値が含まれている場合には、これを使用することをお勧めします。

## 1840 年より前の日付

\$HOROLOG を使用して、1840 年から 9999 年までの範囲にない日付を直接表すことはできません。ただし、InterSystems SQL のユリウス日付機能を使用すると、この範囲外の日付を表すことができます。ユリウス日付は、紀元前 4711 年 (BCE) からカウントした符号なし整数として表されます。ユリウス日付には時刻コンポーネントはありません。

InterSystems IRIS の \$HOROLOG 日付を InterSystems IRIS のユリウス日付に変換するには、[TO\\_CHAR](#) SQL 関数、または `%SYSTEM.SQL` クラスの `TOCHAR()` メソッドを使用します。InterSystems IRIS のユリウス日付を InterSystems IRIS の \$HOROLOG 日付に変換するには、[TO\\_DATE](#) SQL 関数、または `%SYSTEM.SQL` クラスの `TODATE()` メソッドを使用します。

以下の例は、現在の \$HOROLOG 日付を取り、これをユリウス日付に変換しています。\$HOROLOG の前のプラス記号 (+) により、InterSystems IRIS ではこれが数値として扱われるので、コンマの位置で切り捨てられ、時刻整数が削除されます。

### ObjectScript

```
WRITE !,"Horolog date = ",+$H
SET x=$SYSTEM.SQL.TOCHAR(+$HOROLOG,"J")
WRITE !,"Julian date = ",x
```

以下の例は、ユリウス日付を取り、これを InterSystems IRIS の \$HOROLOG 日付に変換しています。

### ObjectScript

```
SET x=$SYSTEM.SQL.TODATE(2455030,"J")
WRITE !,"$HOROLOG date = ",x," = ", $ZDATE(x,1)
```

ただし、1721100 未満のユリウス日付値は変換できず、`<ILLEGAL VALUE>` エラーが生成されます。

ユリウス日付に関する詳細は、["TO\\_DATE"](#) および ["TO\\_CHAR"](#) を参照してください。

## 例

以下の例は、\$HOROLOG の現在の内容を表示します。

### ObjectScript

```
WRITE $HOROLOG
```

これは、64701,49170 の形式で値を返します。

以下の例は \$ZDATE を使用して、\$HOROLOG のデータ・フィールドをデータ形式に変換します。

### ObjectScript

```
WRITE $ZDATE($PIECE($HOROLOG,"",1))
```

02/22/2018 の形式で値を返します。

次の例では、\$HOROLOG の時刻部分が 12 時間 (a.m. または p.m.) 表示の時間:分:秒形式の時刻に変換されます。

## ObjectScript

```
CLOCKTIME
NEW
SET Time=$PIECE($HOROLOG,"",2)
SET Sec=Time#60
SET Totmin=Time\60
SET Min=Totmin#60
SET Milhour=Totmin\60
IF Milhour=12 { SET Hour=12,Meridian=" pm" }
ELSEIF Milhour>12 { SET Hour=Milhour-12,Meridian=" pm" }
ELSE { SET Hour=Milhour,Meridian=" am" }
WRITE !,Hour,":",Min,":",Sec,Meridian
QUIT
```

## 関連項目

- ・ [\\$NOW](#) 関数
- ・ [\\$ZDATE](#) 関数
- ・ [\\$ZDATEH](#) 関数
- ・ [\\$ZDATETIME](#) 関数
- ・ [\\$ZDATETIMEH](#) 関数
- ・ [\\$ZTIME](#) 関数
- ・ [\\$ZTIMEH](#) 関数
- ・ [\\$ZTIMESTAMP](#) 特殊変数
- ・ [\\$ZTIMEZONE](#) 特殊変数

## \$IO (ObjectScript)

現在の入出力デバイスの ID を含みます。

### 構文

```
$IO
$I
```

### 概要

\$IO は、すべての入出力処理が対象となる現在のデバイスのデバイス ID を含みます。入力デバイスと出力デバイスが異なる場合、\$IO には、現在の入力デバイスの ID が含まれます。

InterSystems IRIS は、ログイン時に \$IO の値を主入力/主出力デバイスに設定します。\$PRINCIPAL には、主デバイスの ID が含まれます。USE コマンドを使用して、現在のデバイスを変更できます。USE コマンド、CLOSE コマンド、BREAK コマンド、あるいはターミナルに戻るときのみ、この値を変更できます。

%Library.Device クラスの GetType() メソッドを使用して、現在のデバイスのデバイス・タイプを返すことができます。

UNIX® システムでは、\$IO で実際のデバイス名を指定します。

Windows システムでは、\$IO は主デバイスに対して一意の InterSystems IRIS 生成識別子を含みます。ターミナル・デバイス (TRM または TNT) は、垂直バー、コロン、および別の垂直バーで囲まれた擬似デバイス名で構成され、その後にデバイスのプロセス ID (pid) 番号が続きます。非ターミナル・デバイスでは、擬似デバイス名は垂直バーで囲まれ、その後一意の数値識別子が続きます。

ターミナルでは、|TRM|:|pid

Telnet ターミナルでは、|TNT|nodename:portnumber|pid

ファイル記述子では、|FD|file\_descriptor\_number

(ファイル記述子は CALLIN/CALLOUT リモート・アクセスと使用されます。)

TCP デバイスでは、|TCP|unique\_device\_identifier

名前付きパイプでは、|NPIPE|unique\_device\_identifier

既定のプリンタでは、|PRN|

既定以外のプリンタでは、|PRN|physical\_device\_name

主デバイスが NULL デバイス (バックグラウンド・プロセスの既定) の場合、\$IO は ":pid" が追加された NULL デバイス名を含み、一意の添え字に対して \$IO を使用できるようにします。\$IO に含まれる NULL デバイス名は、オペレーティング・システムによって異なります。

- Windows システムでは、\$IO には //./nul:pid が含まれます。
- UNIX® システムでは、\$IO で /dev/null:pid を指定します。

入力デバイスがパイプ、またはファイルを経由して転送される場合、\$IO は "00" を含みます。

既定のデバイス番号は、設定が可能です。管理ポータルに進み、[システム管理]、[構成]、[デバイス設定]、[デバイス] の順に選択します。必要なデバイスに対して [編集] をクリックし、[物理デバイス名:] オプションを表示して変更します。これを行うと、\$IO は実オペレーティング・システム・デバイス名ではなく、割り当てられたデバイス番号を含みます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## 関連項目

- ・ [USE コマンド](#)
- ・ [\\$PRINCIPAL 特殊変数](#)
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)

## \$JOB (ObjectScript)

現在のプロセスの ID を含みます。

### 構文

```
$JOB
$J
```

### 概要

\$JOB は、現在のプロセスの ID 番号を含みます。この ID 番号は、ホスト・オペレーティング・システムの実際のプロセス ID (PID) です。ID 番号は、各プロセスに対して一意のものです。

現在のプロセスで \$JOB に返される文字列の形式は、**%SYSTEM.Process** クラスの `NodeNameInPid()` メソッドの設定によって決まります。システム全体の既定の動作は、**Config.Miscellaneous** クラスの `NodeNameInPid` プロパティで設定できます。既定では、\$JOB は PID のみを返しますが、ユーザは \$JOB が PID とノード名の両方を返すようにこれらの関数を設定できます。(例) 11284:MYCOMPUTER。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉エラーが返されます。

ターミナル・プロンプトとして PID を設定するには、**%SYSTEM.Process** クラスの `TerminalPrompt(5)` メソッドを使用します。

### 現在のプロセスに関するその他の情報

以下のように、`ProcessId()` メソッドを呼び出すことで、現在の同じプロセス ID 番号を取得することができます。

#### ObjectScript

```
WRITE $SYSTEM.SYS.ProcessId()
```

詳細は、“インターシステムズ・クラス・リファレンス” の “**%SYSTEM.SYS**” クラスを参照してください。

\$JOB を使用すると、次のように現在のプロセスのジョブ番号を取得することができます。

#### ObjectScript

```
SET JobObj=##CLASS(%SYS.ProcessQuery).%OpenId($JOB)
WRITE JobObj.JobNumber
```

詳細は、“インターシステムズ・クラス・リファレンス” の “**%SYS.ProcessQuery**” クラスを参照してください。

現在のプロセスに関する状況情報は、**\$ZJOB** 特殊変数から取得することができます。

現在のプロセスの子プロセスまたは親プロセスの PID は、**\$ZCHILD** および **\$ZPARENT** 特殊変数から取得することができます。

ジョブ・テーブル内の現在のジョブの PID は、**^\$JOB** 構造化システム変数から取得することができます。

### 関連項目

- ・ [JOB コマンド](#)

## \$KEY (ObjectScript)

最新の READ からのターミネータ文字を含みます。

### 構文

```
$KEY  
$K
```

### 概要

\$KEY は、現在のデバイスで最後の READ コマンドを終了した文字、あるいは文字シーケンスを含みます。\$KEY と \$ZB は、機能的に非常に類似しています。これらを比較した詳細は、以下を参照してください。

- ・ ターミネータ文字 ([Enter] キーなど) で最後の READ が終了した場合、\$KEY にはそのターミネータ文字が含まれます。
- ・ タイムアウトや固定長読み込みの長さ制限で最後の READ が終了した場合、\$KEY には NULL 文字列が含まれます。ターミネータ文字に出会うことはありません。
- ・ 最後の READ が単一文字 (READ \*a) の読み込みで、文字が入力された場合、\$KEY には実際の入力文字が含まれます。

\$KEY と \$ZB は同一ではありませんが、非常に類似しています。以下の比較をご覧ください。

SET コマンドを使用して、\$KEY の値を指定することができます。ZZDUMP コマンドを使用して、\$KEY の値を表示することができます。

ターミナル・セッション中は、すべてのコマンド行の最後はキャリッジ・リターン (16 進数で 0D) として \$KEY で記録されます。また、\$KEY 特殊変数は、ターミナル・セッションを初期化するプロセスによって、キャリッジ・リターンに初期化されます。したがって、ターミナル・セッション中に READ コマンド、または SET コマンドによって設定された \$KEY の値を表示するには、\$KEY 値を同じコード行のローカル変数にコピーしなければなりません。

### 例

以下の例では、可変長 READ コマンドはターミナルからデータを受け取るか、10 秒後にタイムアウトになります。ユーザがタイムアウト前にデータを入力した場合、\$KEY には、データ入力を終了したユーザ入力のキャリッジ・リターン (16 進数の 0D) が含まれます。しかし READ がタイムアウトになった場合、\$KEY にはターミネータ文字が受け取られなかったことを示す NULL 文字列が含まれます。

#### ObjectScript

```
READ "Ready or Not: ",x:10  
ZZDUMP $KEY
```

以下の例では、固定長 READ コマンドはターミナルからデータを受け取るか、10 秒後にタイムアウトになります。ユーザが指定の文字数 (この場合は 1 文字) を入力した場合、READ 処理を終了するためにユーザは [Enter] キーを押す必要はありません。指定の文字数を入力する代わりに [Enter] キーを押すことで、READ プロンプトに応答できます。

読み込み処理がタイムアウトになった場合、\$KEY と \$ZB には、どちらも NULL 文字列が含まれます。固定長の READ 処理はターミネータ文字なしで終了されるため、ユーザが 1 文字の中間文字を入力した場合、\$KEY には NULL 文字列が含まれます。しかし、ユーザが中間文字を入力せずに [Enter] キーを押した場合、\$KEY にはユーザ入力のキャリッジ・リターンが含まれます。

## ObjectScript

```

READ "Middle initial: ",z#1:10
IF $ASCII($ZB)=-1 {
    WRITE !,"The read timed out" }
ELSEIF $ASCII($KEY)=-1 {
    WRITE !,"A character was entered" }
ELSEIF $ASCII($KEY)=13 {
    WRITE !,"A line return was entered" }
ELSE {
    WRITE !,"Unexpected result" }

```

## \$KEY と \$ZB の比較

\$KEY と \$ZB には、どちらも READ 処理を終了する文字が含まれます。これらの 2 つの特殊変数は同一ではありませんが、非常に類似しています。以下は主な相違点です。

- ・ \$KEY は SET コマンドを使用して設定することができますが、\$ZB は、SET にすることはできません。
- ・ 固定長 READ が正常に終了すると、\$ZB には最後の文字入力が含まれます (例えば、固定長 READ として 5 桁の郵便番号 "02138" が入力された場合、\$ZB には "8" が含まれます)。固定長 READ が正常に終了すると、\$KEY には NULL 文字列 ("") が含まれます。
- ・ \$KEY は、ブロック型の読み込み処理や書き込み処理をサポートしません。

## コマンド行の \$KEY

ターミナル・コマンド行から対話的にコマンドを実行する場合、[Enter] キーを押して各コマンド行を実行します。\$KEY と \$ZB 特殊変数は、このコマンド行ターミネータ文字を記録します。したがって、\$KEY や \$ZB を使用して読み込み処理の最終状況を返すには、同じコマンド行の一部として変数を設定しなければなりません。

例えば、コマンド

### Terminal

```
>READ x:10
```

をコマンド行から実行する場合、\$KEY をチェックします。これには、読み込み処理の結果が含まれません。コマンド行を実行した <RETURN> 文字が含まれます。読み込み処理の結果を返すには、以下のように同じコマンド行の \$KEY でローカル変数を設定します。

### Terminal

```
>READ x:10 SET rkey=$KEY
```

これにより、読み込み処理で設定された \$KEY の値が保持されます。この読み込み処理値を表示するには、以下のコマンド行文のいずれかを実行します。

### Terminal

```

>WRITE $ASCII(rkey)
; returns -1 for null string (time out)
; returns ASCII decimal value for terminator character
>ZZDUMP rkey
; returns blank line for null string (time out)
; returns hexadecimal value for terminator character

```

## 関連項目

- ・ [READ コマンド](#)
- ・ [SET コマンド](#)
- ・ [ZZDUMP コマンド](#)

- ・ [\\$ZB](#) 特殊変数



# \$NAMESPACE (ObjectScript)

現在のスタック・レベルのネームスペースを含みます。

## 構文

```
$NAMESPACE
SET $NAMESPACE=namespace
NEW $NAMESPACE
```

## 引数

引数	説明
namespace	既存のネームスペースの名前。引用符付きのリテラル文字列、または引用符付きの文字列に解決される式として指定します。ネームスペース名は、大文字と小文字を区別しません。

## 説明

\$NAMESPACE は、現在のスタック・レベルの現在のネームスペース名を含みます。\$NAMESPACE を使用すると、以下を実行できます。

- ・ 現在のネームスペースの名前を返します。
- ・ SET で現在のネームスペースを変更します。
- ・ NEW および SET で新しい一時ネームスペース・コンテキストを確立します。

コード・モジュール内で、現在のネームスペースを変更するお勧めの方法は、NEW \$NAMESPACE に続けて SET \$NAMESPACE=namespace を使用することです。

## 現在のネームスペース名を返す

\$NAMESPACE 特殊変数は、現在のネームスペース名を含みます。

また、以下のように %SYSTEM.SYS クラスの NameSpace() メソッドを呼び出すことで、現在のネームスペース名を取得することもできます。

### ObjectScript

```
WRITE $SYSTEM.SYS.NameSpace()
```

以下のように %Library.File クラスの NormalizeDirectory() メソッドを使用して、現在のネームスペースのフル・パス名を取得できます。

### ObjectScript

```
WRITE $NAMESPACE,!
WRITE ##class(%Library.File).NormalizeDirectory("")
```

以下のように、%SYS.Namespace クラスの Exists() メソッドを使用して、ネームスペースが定義されているかどうかをテストすることができます。

### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

これらのメソッドは、“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。

## SET \$NAMESPACE

SET コマンドを使用して、\$NAMESPACE を既存ネームスペースに設定できます。

コード・モジュール内で、ネームスペースを変更するお勧めの方法は、SET \$ZNSPACE または ZNSPACE コマンドを使用することではなく、NEW \$NAMESPACE に続けて SET \$NAMESPACE=namespace を使用することです。

SET \$NAMESPACE=namespace で、namespace を引用符付きの文字列リテラル、または引用符付きの文字列に評価される変数や式として指定します。namespace では大文字と小文字が区別されません。ただし、InterSystems IRIS は常に、明示的なネームスペース名をすべて大文字、暗黙のネームスペース名をすべて小文字で表示します。ネームスペース名には Unicode 文字を含めることができます。InterSystems IRIS は、アクセント記号付き小文字を対応アクセント記号付き大文字に変換します。

namespace 名は、明示的なネームスペース名 ("USER") でも、暗黙的なネームスペース ("^c:¥InterSystems¥IRIS¥mgr¥user¥") でもかまいません。暗黙的なネームスペースの詳細は、[ZNSPACE](#) コマンドを参照してください。

指定した namespace が存在しない場合、SET \$NAMESPACE は <NAMESPACE> エラーを生成します。ネームスペースへのアクセス特権を持っていない場合、システムは <PROTECT> エラーを生成した後に、データベース・パスを表示します。例えば、%Developer ロールには %SYS ネームスペースへのアクセス特権がありません。このロールを持っていて、このネームスペースにアクセスしようとすると、InterSystems IRIS は <PROTECT> \*c:\intersystems\iris\mgr\ というエラーを返します (Windows システムの場合)。

## NEW \$NAMESPACE

\$NAMESPACE の設定で、現在のネームスペースを変更できます。これは、メソッドやその他のルーチンでネームスペースを変更するお勧めの方法です。NEW \$NAMESPACE および SET \$NAMESPACE を使用することによって、メソッドが終わったときまたは予期しないエラーが発生したときに前のネームスペースに自動的に戻すネームスペース・コンテキストを確立します。

### ObjectScript

```

TRY {
    WRITE "before the method: ", $NAMESPACE, !
    DO MyNSMethod("DocBook")
    WRITE "after the method: ", $NAMESPACE
    RETURN
MyNSMethod(ns)
    NEW $NAMESPACE
    IF ##class(%SYS.Namespace).Exists(ns) {
        SET $NAMESPACE=ns
    }
    ELSE {SET $NAMESPACE="User" }
    WRITE "namespace changed in method: ", $NAMESPACE, !
    SET num=5/$RANDOM(2)
    QUIT
NextMethod()
    WRITE "This should not write", !
}
CATCH exp {
    WRITE "namespace after error in method: ", $NAMESPACE, !
    IF 1=exp.%IsA("%Exception.SystemException") {
        WRITE "System exception: ", $ZCVT(exp.Name, "O", "HTML"), !
    }
}

```

ルーチンを終了するか、またはエラー・トラップに分岐すると、このスタックされたネームスペースに戻ります。これについては、次のターミナルの例で示しています。

### Terminal

```

USER>NEW $NAMESPACE
USER 1S1>SET $NAMESPACE="SAMPLES"
SAMPLES 1S1>SET myoref=##class(%SQL.Statement).%New()
SAMPLES 1S1>QUIT
/* The QUIT reverts to the USER namespace */
USER>

```

## 例

以下の例は、呼び出し元プログラムと異なるネームスペースで実行されるルーチンを呼び出します。これは、NEW \$NAMESPACE を使用して、現在のネームスペースをスタックします。次に SET \$NAMESPACE を使用して、テストが継続している間はネームスペースを変更します。QUIT で、スタックされているネームスペースに戻ります。

### ObjectScript

```
WRITE "before: ", $NAMESPACE, !
DO Test
WRITE "after: ", $NAMESPACE, !
QUIT
Test
NEW $NAMESPACE
SET $NAMESPACE="USER"
WRITE "testing: ", $NAMESPACE, !
; routine code
QUIT
```

エラーを処理して古いネームスペースに戻る必要はありません。InterSystems IRIS は、ユーザが現在のスタック・レベルから離れると、古いネームスペースをリストアします。

次の例は、NEW \$NAMESPACE が省略されている点が前の例と異なります。QUIT の際には、ネームスペースが元に戻らないことに注意してください。

### ObjectScript

```
WRITE "before: ", $NAMESPACE, !
DO Test
WRITE "after: ", $NAMESPACE, !
QUIT
Test
NEW
SET $NAMESPACE="USER"
WRITE "testing: ", $NAMESPACE, !
; routine code
QUIT
```

プログラミングにおいて、現在のネームスペースを一時的に変更しているときには別のルーチンを呼び出すことをお勧めします。

## 関連項目

- ・ [NEW コマンド](#)
- ・ [SET コマンド](#)
- ・ [ZNSPACE コマンド](#)
- ・ [\\$ZNSPACE 特殊変数](#)
- ・ [ネームスペースの構成](#)

## \$PRINCIPAL (ObjectScript)

---

主入出力デバイスの ID を含みます。

### 構文

```
$PRINCIPAL  
$P
```

### 概要

\$PRINCIPAL は、現在のプロセスの主デバイスの ID を含みます。\$PRINCIPAL は、\$IO のように動作します。特定のデバイスのタイプと、システム・プラットフォームに関する詳細は、“\$IO” を参照してください。

主デバイスが閉じているときは、\$PRINCIPAL は変更できません。主入力と主出力のデバイスが異なるときは、\$PRINCIPAL は主入力デバイスの ID を反映します。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉 エラーが返されます。

### 例

次の例では、\$PRINCIPAL を使用して主デバイスをテストします。

#### ObjectScript

```
IF $PIECE($PRINCIPAL,"|",4) {  
    WRITE "Principal device is: ",$PRINCIPAL }  
ELSE { WRITE "Undefined" }
```

以下の例は、主デバイスを使用して書き込みます。

#### ObjectScript

```
USE $PRINCIPAL  
WRITE "output to $PRINCIPAL"
```

### USE \$PRINCIPAL と USE 0

以下の文は機能的に同じです。

#### ObjectScript

```
USE $PRINCIPAL  
USE 0
```

標準的なのは最初の形式であるため、こちらの使用をお勧めします。

### 関連項目

- ・ [USE コマンド](#)
- ・ [\\$IO 特殊変数](#)

## \$QUIT (ObjectScript)

現在のコンテキストを終了するのに必要な QUIT の種類を示すフラグを含みます。

### 構文

```
$QUIT
$Q
```

### 概要

\$QUIT には、現在のコンテキストを終了するときに、引数付き QUIT が必要であるか否かを示す値が含まれています。現在のコンテキストを終了するために引数付き QUIT が必要な場合、\$QUIT には 1 が含まれます。現在のコンテキストを終了するために引数付き QUIT が不要な場合は、\$QUIT に 0 が含まれます。

DO と XECUTE を呼び出すことで作成されるコンテキストでは、引数付き QUIT は終了するために必要ではありません。外部関数が作成するコンテキストでは、終了には引数付き QUIT が必要とされます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉 エラーが返されます。

### 例

以下の例は、DO コンテキスト、XECUTE コンテキスト、ユーザ定義の関数コンテキストの \$QUIT の値を示しています。サンプル・コードは以下のとおりです。

#### ObjectScript

```
QUI
DO
.  WRITE !,"$QUIT in a DO context = ", $QUIT
.  QUIT
XECUTE "WRITE !,"$QUIT in an XECUTE context = ", $QUIT"
SET A=$$A
QUIT
A()
WRITE !,"$QUIT in a User-defined function context = ", $QUIT
QUIT 1
```

このコードを使用したサンプル・セッションは、次のように実行されます。

```
USER>DO ^QUI
$QUIT in a DO context = 0
$QUIT in an XECUTE context = 0
$QUIT in a User-defined function context = 1
```

### \$QUIT とエラー処理

\$QUIT 特殊変数は、エラー・ハンドラが引数付き QUIT が必要なコンテキスト・レベルと、引数なしの QUIT が必要なコンテキスト・レベルで呼び出されるときに特に便利です。

エラー処理の詳細は、"[TRY-CATCH の使用法](#)" を参照してください。

### 関連項目

- [DO コマンド](#)
- [QUIT コマンド](#)
- [XECUTE コマンド](#)

## \$ROLES (ObjectScript)

現在のプロセスに割り当てられているロールを含みます。

### 構文

\$ROLES

### 概要

\$ROLES は、現在のプロセスに割り当てられているロールのリストを含みます。このロールのリストはコンマで区切られた文字列で構成されており、ユーザ・ロールと追加ロールの両方を含めることができます。

ロールをユーザに割り当てるには、SQL の GRANT 文を使用するか、管理ポータルの **[システム管理]**→**[セキュリティ]**→**[ユーザ]** オプションを使用します。定義を編集するユーザ名を選択し、**[ロール]** タブを選択して、そのユーザにロールを割り当てます。ロールを定義するには SQL の CREATE ROLE 文を使用し、削除するには SQL の DROP ROLE 文を使用します。ロールをユーザに割り当てる前にロールを定義する必要があります。ロールをユーザから削除するには、SQL の REVOKE 文を使用します。

JOB コマンドを使用してプロセスを作成すると、その親のプロセスと同じ \$ROLES および \$USERNAME の値が継承されます。

プロセスが入出力リダイレクトを実行するとき、このリダイレクトは、\$ROLES の現在値ではなく、ユーザのログイン \$ROLES 値を使用して実行されます。

### リストされていないロールに与えられたロール

別のロールにロールを与えるのは、InterSystems SQL だけでのみ利用できる概念です。ロールに与えられたロールは、SQL 特権をチェックするためのユーザのロールのリストを判断するために SQL 内で使用されます。ObjectScript からはアクセスできません。InterSystems IRIS システム・セキュリティからは他のロールにロールを与えることはできません。そのため、\$ROLES 特殊変数リストには、SQL 操作が現在のロールに与えたロールは含まれません。詳細は、“InterSystems SQL リファレンス” の “**GRANT**” コマンドを参照してください。

### SET \$ROLES

SET コマンドを使用すると、\$ROLES に含まれているリストの追加ロールの部分を変更できます。\$ROLES を設定すると、プロセスの追加ロールのみが変更されます。プロセスのユーザ・ロールは変更できません。\$ROLES を別の追加ロールのリストに設定すると、システム機能が制限されます。ただし、このような制限は \$ROLES を NULL 文字列に設定する場合には適用されません。この設定を行うと、追加ロールのリストが削除されます。

ロールを追加する前にロールを定義する必要があります。SQL の CREATE ROLE コマンドを使用して、ロールを定義できます。CREATE ROLE はロールに特権を付与しません。ロールに特権を付与するには、SQL の GRANT 文、または管理ポータル **[システム管理]**→**[セキュリティ]**→**[ロール]** インタフェースを使用します。

SET \$ROLES を使用してプロセス・ロールをエスカレートする前に、NEW \$ROLES 文を発行する必要があります。

### NEW \$ROLES

NEW \$ROLES は、\$ROLES と \$USERNAME の両方の現在値をスタックに配置します。\$ROLES に対してはセキュリティ制限なしに NEW コマンドを使用できます。

NEW \$ROLES を発行し、次に SET \$ROLES を発行して追加ロールを提供します。このようにすると、それらの追加ロールを使用するオブジェクト・インスタンスを作成できます。このルーチンを終了した場合は、InterSystems IRIS は、それらの追加ロールを持つオブジェクトを閉じてから、スタックされた \$ROLES 値に戻します。

### 例

以下の例は、現在のプロセスのロール・リストを返します。

## ObjectScript

```
WRITE $ROLES
```

以下の例では、最初にロール Vendor、Sales、および Contractor を作成します。既定ロール (ユーザ・ロールと追加ロールの両方が含まれる) のコンマ区切りのリストが表示されます。最初の SET \$ROLES は、追加ロールのリストを Sales と Contractor の 2 つのロールに置き換えます。2 番目の SET \$ROLES は、Vendor ロールを追加ロールのリストに連結します。最後の SET \$ROLES は、すべての追加ロールを削除して、追加ロール・リストを NULL 文字列に設定します。ユーザ・ロールは変更されません。

## ObjectScript

```
CreateRoles
    &sql(CREATE ROLE Vendor)
    &sql(CREATE ROLE Sales)
    &sql(CREATE ROLE Contractor)
    IF SQLCODE=0 {
        WRITE !,"Created new roles"
        DO SetRoles }
    ELSEIF SQLCODE=-118 {
        WRITE !,"Role already exists"
        DO SetRoles }
    ELSE { WRITE !,"CREATE ROLE failed, SQLCODE=",SQLCODE }
SetRoles()
    WRITE !,"Initial: ", $ROLES
    NEW $ROLES
    SET $ROLES="Sales,Contractor"
    WRITE !,"Replaced: ", $ROLES
    NEW $ROLES
    SET $ROLES=$ROLES_",Vendor"
    WRITE !,"Concatenated: ", $ROLES
    SET $ROLES=""
    WRITE !,"Nulled: ", $ROLES
```

## 関連項目

- ObjectScript : [SET](#) コマンド、[NEW](#) コマンド、[\\$USERNAME](#) 特殊変数
- InterSystems SQL : [CREATE ROLE](#)、[DROP ROLE](#)、[GRANT](#)、[REVOKE](#)、[%CHECKPRIV](#)

## \$STACK (ObjectScript)

コール・スタックに保存されているコンテキスト・フレーム数を収めた特殊変数です。

### 構文

```
$STACK  
$ST
```

### 概要

\$STACK には、プロセスのコール・スタックに現在保存されているコンテキスト・フレームの数が格納されます。現在実行しているコンテキストの 0 から始まるコンテキスト・レベル番号として、\$STACK を見ることもできます。そのため、InterSystems IRIS ジョブが開始すると、コンテキストがコール・スタックで保存される前は、\$STACK の値は 0 です。

DO コマンドでルーチンが他のルーチン呼び出すたびに、現在実行しているルーチンのコンテキストはコール・スタックに保存され、実行は呼び出されるルーチンの新しく作成されたコンテキストで始まります。呼び出されたルーチンは、順番に他のルーチン呼び出すことができます。各追加コールは、保存された他のコンテキストをコール・スタックに置きます。

XECUTE コマンドと外部関数参照も、新規の実行コンテキストを作成します。GOTO コマンドは作成しません。

DO コマンド、XECUTE コマンド、外部関数参照が新しいコンテキストを作成するたびに、\$STACK の値はインクリメントします。コンテキストが QUIT コマンドで終了するたびに、前のコンテキストはコール・スタックから戻され、\$STACK の値はデクリメントします。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉 エラーが返されます。

**\$ESTACK** は、\$STACK と同じです。ただし、NEW \$ESTACK コマンドを発行することで、いつでも 0 (ゼロ) の \$ESTACK レベルを作成できる点が異なります。NEW を使用して、\$STACK 特殊変数をリセットすることはできません。

### エラー処理

エラー発生時、すべてのコンテキスト情報はプロセス・エラー・スタックに即座に格納されます。これにより \$STACK の値が変更されます。この場合、エラー・ハンドラによって **\$ECODE** 値がクリアされるまで、**\$STACK** 関数を使用してコンテキスト情報にアクセスできます。つまり、\$ECODE 値が NULL でない場合、\$STACK 関数は、エラー・スタックに保存されたコンテキストと同じ指定されたコンテキスト・レベルのアクティブなコンテキストではなく、エラー・スタックに保存されたコンテキストの情報を返します。

### ターミナル・プロンプトから呼び出す場合のコンテキスト・レベル

プログラムから呼び出されるルーチンは、ターミナル・プロンプトから DO コマンドを使用して呼び出すルーチンとは異なるコンテキスト・レベルで開始します。ターミナル・プロンプトで入力する DO コマンドは、新しいコンテキストを作成します。以下の例では、ルーチンまたはターミナル・プロンプトから呼び出されたルーチン START を示しています。

以下のルーチンを考慮してみましょう。

#### ObjectScript

```
START  
; Display the context level and exit  
WRITE !,"Context level in routine START is ", $STACK  
QUIT
```

プログラムから START を実行すると、以下が表示されます。

```
Context level in routine START is 0
```

ターミナル・プロンプトで DO ^START を発行して START を実行すると、以下が表示されます。



Context level in routine START is 1

## 例

以下の例は、\$STACK が新しいコンテキストが作成されるときにインクリメントされ、コンテキストが終了するときにデクリメントされる方法を示しています。

サンプル・コードは以下のとおりです。

### ObjectScript

```
STA
WRITE !,"Context level in routine STA = ",$STACK
DO A
WRITE !,"Context level after routine A = ",$STACK
QUIT
A
WRITE !,"Context level in routine A = ",$STACK
DO B
WRITE !,"Context level after routine B = ",$STACK
QUIT
B
WRITE !,"Context level in routine B = ",$STACK
XECUTE "WRITE !,"Context level in XECUTE = ",$STACK"
WRITE !,"Context level after XECUTE = ",$STACK
QUIT
```

このコードを使用したサンプル・セッションは、次のように実行されます。

```
USER>DO ^STA
Context level in routine STA = 1
Context level in routine A = 2
Context level in routine B = 3
Context level in XECUTE = 4
Context level after XECUTE = 3
Context level after routine B = 2
Context level after routine A = 1
```

## 関連項目

- ・ [\\$STACK 関数](#)
- ・ [\\$ESTACK 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)
- ・ [スタックを表示する %STACK の使用法](#)

## \$STORAGE (ObjectScript)

ローカル変数ストレージに使用できるバイト数を含みます。

### 構文

```
$STORAGE  
$S
```

### 概要

\$STORAGE は、現在のプロセスのパーティションで、ローカル変数ストレージに利用できるバイト数を返します。\$STORAGE の初期値は、プロセスで使用できる最大メモリ量である \$ZSTORAGE の値により、確立されます。\$ZSTORAGE の値 (キロバイト) が増えると、\$STORAGE の値 (バイト) も増えます。ただし、\$ZSTORAGE と \$STORAGE の関係は、単純な 1:1 の比とはなりません。

\$STORAGE の値は、以下の処理によって影響を受けます。

- ・ \$STORAGE の値は、SET コマンドなどの使用により、ローカル変数がローカル変数の領域で定義されると減少します。\$STORAGE の減少は、ローカル変数の値を格納するために必要な領域の容量に関係します。ローカル変数の名前のサイズは、\$STORAGE に影響しませんが、添え字レベルの数値は \$STORAGE に影響します。\$STORAGE の値は、KILL コマンドなどを使用してローカル変数が削除されると増大します。
- ・ \$STORAGE は、NEW コマンド発行時に減少します。NEW は、新規の実行レベルを確立します。前回の実行レベルでローカル変数に対して設定された領域は (使用の有無に関係なしに)、新規の実行レベルでは使用できません。最初の NEW により、\$STORAGE は約 15000 減少します。以降の各 NEW により、\$STORAGE は 12288 減少します。\$STORAGE の値は、QUIT コマンドを発行して、実行レベルを終了させると増大します。
- ・ \$STORAGE は、IF や FOR などのフロー制御文、または TRY や CATCH などのブロック構造を定義したときに減少します。ストレージは、これらの構造を実行するためではなく、コンパイルするために割り当てられます。したがって、FOR 文は、ループの有無やループの回数に関係なく、同じストレージ容量を消費します。IF、ELSEIF、および ELSE 節は、実行される分岐の数に関係なく、それぞれ設定されたストレージ容量を消費します。領域は、コードをコンパイルしたプロセスから割り当てられます。通常、FOR ループは、ローカル変数をカウンタとして定義することに注意してください。

\$STORAGE の値は、プロセス・プライベート・グローバル、グローバル変数、または特殊変数の設定により影響を受けません。\$STORAGE の値には、ネームスペースの変更による影響はありません。

\$STORAGE 特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

### ローメモリと <STORE> エラー

\$STORAGE の値は、正または負の数値とすることができます。値 0 は、使用できるストレージがないことを示すのではなく、ストレージが非常に不足していることを示します。\$STORAGE が 0 未満に減少した場合、ある時点で <STORE> エラーが発生します。例えば、\$STORAGE が -7000 に減少した場合、別のローカル変数に対してストレージを割り当てると、ローカル変数値の格納や新しい実行レベルの確立に利用可能なストレージ領域が不足していることを示す <STORE> エラーが返され、失敗する可能性があります。

最初の <STORE> エラーは、\$STORAGE が 0 未満のある値になったときに発生しますが、正確な負の \$STORAGE 値のしきい値は状況により異なります。この <STORE> エラーは、\$ZSTORAGE を増加するか、KILL または QUIT 処理で一部の割り当て済みのストレージを解放することで、ストレージを追加する必要があることを示します。この最初の <STORE> エラーが発生すると、システムは、自動的に 1MB (1,048,576 バイト) の追加メモリをプロセスに提供し、エラー処理およびリカバリを行えるようにします。InterSystems IRIS は \$ZSTORAGE を変更せず、\$STORAGE が負の数値になるのを許容します。

この最初の〈STORE〉エラーが発生すると、InterSystems IRIS は、内部で「ローメモリ状態にあるプロセス」としてプロセスを指定します。このローメモリ状態にある間、プロセスがメモリの割り当てを継続し、\$STORAGE の値がさらに小さい負の数値に減少し続ける場合があります。このローメモリ状態にある間、プロセスが割り当てメモリの一部を解放し、\$STORAGE の値が上昇する場合があります。したがって、\$STORAGE の値は、新たな〈STORE〉エラーを出すことなく、値の範囲内で増減します。また、最初の〈STORE〉エラーの後に、InterSystems IRIS が一部の内部メモリを解放することで \$STORAGE の値がわずかに上昇する場合があります。

この最初の〈STORE〉エラーが発生すると、予備のメモリが提供されます。これにより、プロセスは、診断機能呼び出し、ディスクへの保存を実施し、正常に終了し、メモリを解放して、続行することが可能になります。

プロセスは、以下のいずれかが行われるまでローメモリ状態のままです。

- ・ プロセスで十分なメモリを使用可能にします。[\\$ZSTORAGE](#) の割り当てを増加させたり、KILL または QUIT 処理によって割り当て済みストレージを解放したりすることで、プロセスはこれを行うことができます。[\\$STORAGE](#) の値が 256K (または [\\$ZSTORAGE](#) の 25%、いずれか小さい方) を超えると、InterSystems IRIS によってそのプロセスのローメモリ状態が解除されます。使用可能メモリが負の数値に減少した場合、その時点で再び〈STORE〉エラーがプロセスによって出される場合があります。
- ・ プロセスは追加のメモリを消費します。[\\$STORAGE](#) の値が -1048576 に達すると、2 回目の〈STORE〉エラーが発生します。プロセスがこの時点に到達した場合、プロセスはこれ以上メモリを使用できず、それ以降のプロセス処理は予測不能になります。おそらくプロセスは直ちに終了します。

[\\$SYSTEM.Process.MemoryAutoExpandStatus\(\)](#) メソッドを呼び出すことで〈STORE〉エラーの理由を判別できます。

**注釈** オペレーティング・システムによって、実行中のアプリケーションの最大メモリ割り当てに上限が課される場合があります。InterSystems IRIS インスタンスがこのような上限の対象である場合、プロセスは [\\$ZSTORAGE](#) で指定されたメモリの一部を取得できず、その結果〈STORE〉エラーが発生することがあります。

## 例

以下の例では、[\\$ZSTORAGE](#) に少ない値が設定された場合、どのように [\\$STORAGE](#) も少なくなるかを示します。これらの 2 つの値の関係 (比) は可変であることに注意してください。

### ObjectScript

```
SET $ZS=262144
WRITE "$ZS=", $ZS, " $S=", $S, " ratio=", $NORMALIZE($S/$ZS, 3), !
FOR i=1:1:10 {
    IF $ZS>32768 {SET $ZS=$ZS-32768
        WRITE "$ZS=", $ZS, " $S=", $S, " ratio=", $NORMALIZE($S/$ZS, 3), !
    }
}
```

以下の例では、ローカル変数が割り当てられた場合、どのように [\\$STORAGE](#) が減少するか、またローカル変数が削除された場合、どのように増加するかを示します。

### ObjectScript

```
WRITE "$STORAGE=", $S, " initial value", !
FOR i=1:1:30 {SET a(i)="abcdefghijklmnopqrstuvwxyz"
    WRITE "$STORAGE=", $S, ! }
KILL a
WRITE !, "$STORAGE=", $S, " after KILL", !
```

以下の例では、割り当てられたローカル変数の添え字レベルの数が [\\$STORAGE](#) にどのように影響するかを示しています。

## ObjectScript

```

WRITE "No subscripts:",!
SET before=$$
SET a="abcdefghijklmnopqrstuvwxyz"
WRITE " memory allocated ",before-$$,!
KILL a
WRITE "One subscript level:",!
SET before=$$
SET a(1)="abcdefghijklmnopqrstuvwxyz"
WRITE " memory allocated ",before-$$,!
KILL a(1)
WRITE "Nine subscript levels:",!
SET before=$$
SET a(1,2,3,4,5,6,7,8,9)="abcdefghijklmnopqrstuvwxyz"
WRITE " memory allocated ",before-$$,!
KILL a(1,2,3,4,5,6,7,8,9)

```

以下の例では、NEW による新規実行レベル確立で、どのように \$STORAGE が減少するか（そのレベルでの使用が不可能となるか）を示します。

## ObjectScript

```

WRITE "increasing levels:",!
FOR i=1:1:10 {WRITE "$STORAGE=", $$, ! NEW }

```

以下の例では、ローメモリ状態になるまで、ローカル変数が割り当てられた際に、\$STORAGE がどのように減少していき、<STORE> エラーが出されるかを示しています。<STORE> エラーは、StoreErrorReason() メソッドを呼び出して、エラーの原因を判別する CATCH ブロックによって捕捉されます。CATCH ブロックに入ると大量のストレージを消費することに注意してください。CATCH ブロックに入った後、この例では、変数をもう一つ割り当てています。

## ObjectScript

```

TRY {
  WRITE !, "TRY block", !
  SET init=$ZSTORAGE
  SET $ZSTORAGE=456
  WRITE "Initial $STORAGE=", $STORAGE, !
  FOR i=1:1:1000 {
    SET pre=$STORAGE
    SET var(i)="1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    IF $STORAGE<0 {WRITE "var(", i, ") negative memory=", $STORAGE, ! }
    ELSEIF pre<$STORAGE {WRITE "var(", i, ") new allocation $$=", $STORAGE, ! }
    ELSE {WRITE "var(", i, ") $$=", $STORAGE, ! }
  }
}
CATCH myexp {
  WRITE !, "CATCH block exception handler", !!
  WRITE "Name: ", $ZCVT(myexp.Name, "O", "HTML"), !
  IF myexp.Name="<STORE>" {WRITE "store error reason=",
    $SYSTEM.Process.StoreErrorReason(), ! }
  WRITE "$S=", $STORAGE, !
  SET j=i
  SET var(j)="1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
  WRITE "var(", j, ") added one more variable $$=", $STORAGE, !
  SET $ZSTORAGE=init
  RETURN
}

```

## 関連項目

- ・ [\\$ZSTORAGE 特殊変数](#)
- ・ [ローカル変数](#)

# \$SYSTEM (ObjectScript)

システム・オブジェクトに関するシステム情報を含みます。

## 構文

```
$SYSTEM
$SY
$SYSTEM.class.method()
```

## 概要

\$SYSTEM は、特殊変数として、またはシステム情報を返すメソッドを呼び出すクラスとして呼び出すことができます。

## \$SYSTEM 特殊変数

特殊変数としての \$SYSTEM には、ローカル・システム名と現在の InterSystems IRIS インスタンスの名前がコロン (:) で区切られて含まれています。マシンの名前は、ローカルのオペレーティング・システムの大文字/小文字の使い分けの規約に従います。インスタンス名は大文字です。以下はその例です。

```
MyComputer:IRISInstance
```

LocalHostName() メソッドを使用して、ローカル・システム名を特定することもできます。

## ObjectScript

```
WRITE $SYSTEM,!
WRITE $SYSTEM.INetInfo.LocalHostName()
```

\$SY という省略形は、\$SYSTEM で特殊変数としてのみ使用できます。

## \$SYSTEM クラス

クラスとしての \$SYSTEM は、さまざまなシステム・オブジェクトへのアクセスを提供します。情報を返すメソッドや、アップグレードやロードなどのオペレーションを行い、状況情報を返すメソッドを呼び出すことができます。InterSystems IRIS は、以下を含むシステム・オブジェクトの複数のクラスをサポートします。

- ・ Version : InterSystems IRIS とそのコンポーネントのバージョン番号に関する情報
- ・ SYS : システムに関する情報
- ・ OBJ : オブジェクトに関する情報
- ・ SQL : SQLクエリに関する情報

オブジェクト・クラス名とメソッド名は、大文字と小文字を区別することに注意してください。これらの名前の大文字と小文字の指定を間違えると、<CLASS DOES NOT EXIST> エラー、または <METHOD DOES NOT EXIST> エラーが返されます。メソッド名で括弧を指定しない場合は、<SYNTAX> エラーが発行されます。

以下の同等の構文の例に示すように、\$SYSTEM メソッドとプロパティにはドット構文を使用してアクセスできます。

- ・ WRITE ##class(%SYSTEM.INetInfo).LocalHostName()
- ・ WRITE \$SYSTEM.INetInfo.LocalHostName()

“インターシステムズ・クラス・リファレンス”ドキュメントで説明されているように、\$SYSTEM は、%SYSTEM クラス・パッケージ内のシステム API クラスにアクセスできます。##class 構文では、%SYSTEM クラス・パッケージ名の大文字と小文字が区別されることに注意してください。\$SYSTEM 構文では、\$SYSTEM キーワードの大文字と小文字は区別されません。ドット構文の使用の詳細は、“[登録オブジェクトを使用した作業](#)”を参照してください。

%SYSTEM.OBJ の使用法の詳細は、“[フラグおよび修飾子](#)”を参照してください。

## 例

以下は、\$SYSTEMを使用して現在のネームスペースで使用可能なクラスのリストを表示するメソッドを呼び出す例です。

### ObjectScript

```
DO $SYSTEM.OBJ.ShowClasses()
```

これは、以下のような結果を表示します。

```
%SYS>d $system.OBJ.ShowClasses()  
%SYS.APIManagement  
%SYS.Audit  
%SYS.AuditString  
%SYS.ClusterInfo  
%SYS.DatabaseQuery  
...  
SYS.WSMon.wsProcess  
SYS.WSMon.wsResource  
SYS.WSMon.wsSystem
```

次のように、OBJクラスのすべてのメソッドをリストにすることができます。(クラス名を変更することで、この方法を使用して任意のシステム・クラスのリストを取得することができます)

### ObjectScript

```
DO $SYSTEM.OBJ.Help()
```

クラス内のメソッド情報を1つだけリストにするには、以下の例で示されているようにHelp引数リストでメソッド名を指定します。

### ObjectScript

```
DO $SYSTEM.OBJ.Help("Load")
```

以下は、メソッドを呼び出す\$SYSTEMの例です。

### ObjectScript

```
DO $SYSTEM.OBJ.Upgrade()  
WRITE !,"* * * * *"  
DO $SYSTEM.CSP.DisplayConfig()  
WRITE !,"* * * * *"  
WRITE !,$SYSTEM.Version.GetPlatform()  
WRITE !,"* * * * *"  
WRITE !,$SYSTEM.SYS.TimeStamp()
```

以下の例は、##class(%SYSTEM)構文を使用して、前述の例と同様のメソッドを呼び出します。

### ObjectScript

```
DO ##class(%SYSTEM.OBJ).Upgrade()  
DO ##class(%SYSTEM.CSP).DisplayConfig()  
WRITE !,##class(%SYSTEM.Version).GetPlatform()  
WRITE !,##class(%SYSTEM.SYS).TimeStamp()
```

前述の2つの例では、UnknownUserが%DB\_IRISSYSロールを割り当てている必要があります。

## 関連項目

- ・ [\\$ISOBJECT 関数](#)
- ・ [\\$ZVERSION 特殊変数](#)
- ・ [フラグおよび修飾子](#)

- ・ 登録オブジェクトを使用した作業

## \$TEST (ObjectScript)

タイムアウト・オプションを使用した最後のコマンドの結果である真理値を含みます。

### 構文

```
$TEST
$T
```

### 説明

\$TEST は、タイムアウトを使用した最後のコマンドの結果である真理値 (0 または 1) を含みます。\$TEST は、ターミナル・プロンプトで起動したか、ルーチン・コードにあったかに関係なく、以下のコマンドによって設定されます。

- ・ 時間制限付き JOB は、時間内に新規のジョブの開始が成功すると \$TEST を 1 に設定します。時間内に成功しなければ、\$TEST を 0 に設定します。
- ・ 時間制限付き LOCK は、時間内にロックが成功すると \$TEST を 1 に設定します。時間内に成功しなければ、\$TEST を 0 に設定します。
- ・ 時間制限付き OPEN は、時間内にオープンが成功すると \$TEST を 1 に設定します。時間内に成功しなければ、\$TEST を 0 に設定します。
- ・ 時間制限付き READ は、時間内に読み取りが終了すると \$TEST を 1 に設定します。時間内に成功しなければ、\$TEST を 0 に設定します。

これらのコマンドをタイムアウトなしで発行した場合は、\$TEST は設定されません。

### \$TEST の設定

SET コマンドを使用して、\$TEST にブーリアン値を設定することができます。値が 1 または 0 以外の任意の数値の場合は、\$TEST=1 に設定されます。値が 0 または非数値の文字列値の場合は、\$TEST=0 に設定されます。

\$TEST は、論理条件を返すことができる任意のコマンドまたは関数によって設定できます。

### \$TEST の保持

タイムアウトを指定していなかった JOB、LOCK、OPEN、または READ コマンドが正常に実行された場合は、\$TEST の既存値は変更されません。

DO コマンドは、プロシージャの呼び出し時には \$TEST の値を保持しますが、サブルーチンの呼び出し時にはこの値を保持しません。詳細は、“[DO](#)” コマンドを参照してください。

ZBREAK コマンドは、execute\_code の呼び出し時に \$TEST の値を保持します。詳細は、“[ZBREAK](#)” をコマンドを参照してください。

### 例

次のコードでは時間制限付きの読み取りが実行され、\$TEST を使用して読み取りの完了がテストされます。

#### ObjectScript

```
READ !,"Type a letter: ",a#1:10
IF $TEST { DO Success(a) }
ELSE { DO TimedOut }
Success(val)
WRITE !,"Received data: ",val
TimedOut()
WRITE !,"Timed out"
```



## \$TEST を設定しない処理

JOB、LOCK、OPEN、および READ の各コマンドで時間制限のない場合は、\$TEST に影響を与えません。後置条件式も、\$TEST に影響を与えません。

ブロック型 IF コマンドは、\$TEST を全く使用しません。

## 時間制限付きの処理の失敗

InterSystems IRIS は、時間制限付き処理が失敗した後は、エラー・メッセージを作成しません。アプリケーションは、\$TEST を確認してから適切なメッセージを作成する必要があります。

## 関連項目

- ・ [JOB コマンド](#)
- ・ [LOCK コマンド](#)
- ・ [OPEN コマンド](#)
- ・ [READ コマンド](#)

## \$THIS (ObjectScript)

---

現在のクラス・コンテキストを含みます。

### 構文

`$THIS`

### 説明

`$THIS` は、現在のクラス・コンテキストを含みます。インスタンス・メソッドのクラス・コンテキストは、現在のオブジェクト参照 (OREF) です。クラス・メソッドのクラス・コンテキストは、文字列値としての現在のクラス名です。例えば、クラス・メソッド内から `DO ..method()` コマンドまたは `SET ..property = value` コマンドを発行すると、..`コンテキストは、$THIS` の現在の値を使用して解決されます。オブジェクト・インスタンス内で参照を作成する場合は、相対ドット構文(..)をお勧めします。

`$THIS` は通常、オブジェクト・インスタンス内で、別のオブジェクトにある関数を呼び出すときに使用されます。この場合、`$THIS` を使用し、現在のクラス・コンテキストをその関数に渡して、現在のオブジェクト・インスタンスに値を返すようにすることができます。

`$THIS` に有効なオブジェクト参照が含まれていない場合、InterSystems IRIS は <NO CURRENT OBJECT> エラーを返します。

`$THIS` は、以下のようなコンテキストで使用できます。

```
SET x = ##class(otherclassname).method($THIS)
DO ##class(superclass)$THIS.method(args)
```

この特殊変数では、SET コマンドを使用して値を設定することはできません。設定しようとすると、<FUNCTION> エラーが返されます。

詳細は、“[オブジェクト特有の ObjectScript の機能](#)”にある“`$this` 構文”を参照してください。OREFの詳細は、“[OREF の基本](#)”を参照してください。

### 関連項目

- ・ `$CLASSNAME` 関数

## \$THROWOBJ (ObjectScript)

---

失敗した THROW からの OREF が含まれます。

### 構文

\$THROWOBJ

### 説明

\$THROWOBJ には、直近の失敗した THROW 処理によってスローされたオブジェクト参照 (OREF) が含まれます。InterSystems IRIS は、<THROW> エラーを発行する際に OREF を \$THROWOBJ に書き込みます。通常、TRY ブロックや CATCH ブロック内にないときに THROW を発行しようするとこれが発生します。

正常な THROW 処理により、\$THROWOBJ が空の文字列にリセットされます。

TRY、THROW、および CATCH については、“[TRY-CATCH メカニズム](#)”を参照してください。

OREFの詳細は、“[OREF の基本](#)”を参照してください。

### \$THROWOBJ の設定

以下のように \$THROWOBJ を明示的にリセットすることもできます。

#### ObjectScript

```
SET $THROWOBJ= " "
```

\$THROWOBJ では、SET コマンドを使用して空の文字列以外の値を設定することはできません。これを試みると、<ILLEGAL VALUE> エラーが返されます。

### 関連項目

- ・ [THROW コマンド](#)
- ・ [TRY-CATCH の使用法](#)

## \$TLEVEL (ObjectScript)

トランザクション処理に対する、現在の入れ子レベルを含みます。

### 構文

```
$TLEVEL  
$TL
```

### 概要

\$TLEVEL は、入れ子の開いているトランザクションの数である、現在のトランザクション・レベルを含みます。発行された TSTART コマンドの数がトランザクション・レベルを決定します。

- ・ 各 TSTART は \$TLEVEL を 1 つインクリメントします。
- ・ 各 TCOMMIT は \$TLEVEL を 1 つデクリメントします。
- ・ 各 TROLLBACK 1 は \$TLEVEL を 1 つデクリメントします。
- ・ TROLLBACK は \$TLEVEL を 0 にリセットします。

0 の \$TLEVEL はデクリメントできません。\$TLEVEL=0 が処理を実行しないとき、TROLLBACK (または TROLLBACK 1) が発行されます。\$TLEVEL=0 のときに TCOMMIT を発行すると、<COMMAND> エラーが返されます。

トランザクションの最大レベル数は 255 です。255 のトランザクション・レベルを超えると、<TRANSACTION LEVEL> エラーになります。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

### SQL と \$TLEVEL

\$TLEVEL は、以下のように SQL トランザクション文によっても設定できます。

- ・ 最初の START TRANSACTION は、\$TLEVEL を 1 に設定します。追加の START TRANSACTION 文は、\$TLEVEL に影響を与えません。
- ・ 各 SAVEPOINT 文 は \$TLEVEL を 1 つインクリメントします。
- ・ ROLLBACK TO SAVEPOINT pointname 文は、\$TLEVEL をデクリメントします。デクリメントの量は、指定するセーブポイントによって決定します。
- ・ COMMIT は \$TLEVEL を 0 にリセットします。
- ・ ROLLBACK は \$TLEVEL を 0 にリセットします。

\$TLEVEL を共有で使用しているにもかかわらず、ObjectScript トランザクション処理と SQL トランザクション処理は異なり、また互換性也没有ありません。アプリケーションは、同じトランザクション内のこれら 2 種類のトランザクション処理文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・レベルとターミナル・プロンプト

既定では、\$TLEVEL がターミナル・プロンプトから実行されるコマンド行またはプログラムの最後で 0 よりも大きい場合は、現在のトランザクション・レベルがターミナル・プロンプトの接頭語として表示されます。

- ・ \$TLEVEL=0 の場合は、ターミナル・プロンプトは現在のネームスペース名を表示します (既定)。例えば、USER> です。
- ・ \$TLEVEL>0 の場合は、ターミナル・プロンプトではネームスペース名の前に TLn: 接頭語が表示されます。n は 1 ~ 255 までの整数です。例えば、TL4:USER> です。

このターミナル・プロンプトは、[ZNSPACE](#) の説明に従って構成できます。

SQL シェル・プロンプトには、現在のトランザクション・レベルは表示されません。SQL シェルの終了時には、現在の \$TLEVEL 値がターミナル・プロンプトに表示されます。これには、SQL シェルを起動する前に設定されたトランザクション・レベルと SQL シェル内で発生したトランザクション・レベルの変更が含まれる場合があります。

## 例

以下の例では、TSTART が \$TLEVEL をインクリメントし、各 TCOMMIT が \$TLEVEL をデクリメントします。

### ObjectScript

```
WRITE !,"transaction level ", $TLEVEL // 0
TSTART
WRITE !,"transaction level ", $TLEVEL // 1
TSTART
WRITE !,"transaction level ", $TLEVEL // 2
TCOMMIT
WRITE !,"transaction level ", $TLEVEL // 1
TCOMMIT
WRITE !,"transaction level ", $TLEVEL // 0
```

以下の例では、TSTART を繰り返し呼び出すことで \$TLEVEL をインクリメントし、各 TROLLBACK 1 が \$TLEVEL をデクリメントします。

### ObjectScript

```
WRITE !,"transaction level ", $TLEVEL // 0
TSTART
WRITE !,"transaction level ", $TLEVEL // 1
TSTART
WRITE !,"transaction level ", $TLEVEL // 2
TROLLBACK 1
WRITE !,"transaction level ", $TLEVEL // 1
```

以下の例では、TSTART を繰り返し呼び出すことで \$TLEVEL をインクリメントし、TROLLBACK が \$TLEVEL を 0 にリセットします。

### ObjectScript

```
WRITE !,"transaction level ", $TLEVEL // 0
TSTART
TSTART
TSTART
WRITE !,"transaction level ", $TLEVEL // 3
TROLLBACK
WRITE !,"transaction level ", $TLEVEL // 0
```

以下の例は、\$TLEVEL が 0 の場合に、TROLLBACK コマンドが効果がないことを示します。

### ObjectScript

```
WRITE !,"transaction level ", $TLEVEL // 0
TROLLBACK
WRITE !,"transaction level ", $TLEVEL // 0
TROLLBACK 1
WRITE !,"transaction level ", $TLEVEL // 0
TROLLBACK
WRITE !,"transaction level ", $TLEVEL // 0
```

## 関連項目

- [TCOMMIT](#) コマンド
- [TROLLBACK](#) コマンド
- [TSTART](#) コマンド

- ・ [トランザクション処理での ObjectScript の使用法](#)

# \$USERNAME (ObjectScript)

現在のプロセスのユーザ名を含みます。

## 構文

\$USERNAME

## 概要

\$USERNAME は、現在のプロセスのユーザ名を含みます。これは以下の形式のうちのいずれかです。

- ・ 現在のユーザ名を返します。例：Mary。この値は、複数のセキュリティ・ドメインが許可されていない場合に返されます。
- ・ 現在のユーザの名前とシステム・アドレスを返します。例：Mary@jupiter。この値は、複数のセキュリティ・ドメインが許可されている場合に返されます。

複数のセキュリティ・ドメインを許可するには、管理ポータルに進み、[システム管理]→[セキュリティ]→[システム・セキュリティ]→[システムワイドセキュリティパラメータ]の順に選択します。[複数セキュリティ・ドメインを許可する]チェック・ボックスにチェックを付けます。この設定に対する変更は、新しく呼び出されたプロセスに適用され、現在のプロセスから返された値には影響を与えません。

SET コマンドまたは NEW コマンドを使用して、この値を変更することはできません。ただし、NEW \$ROLES は、現在の \$USERNAME の値もスタックに配置します。

一般的に、\$USERNAME 値は接続時に指定されたユーザ名になります。ただし、非認証のアクセスが許可されている場合、ユーザ・ターミナルまたは ODBC クライアントがユーザ名を指定せずに InterSystems IRIS に接続する場合があります。この場合、\$USERNAME には文字列 “UnknownUser” が含まれます。

JOB コマンドを使用してプロセスを作成すると、その親のプロセスと同じ \$USERNAME および \$ROLES の値が継承されます。

ユーザ名を作成するには SQL の CREATE USER 文を使用し、削除するには SQL の DROP USER 文を使用します。ユーザのパスワードを変更するには、SQL の ALTER USER 文を使用します。ロールをユーザに割り当てるには、SQL の GRANT 文を使用するか、ロールをユーザに追加するシステム・ユーティリティを使用します。現在のプロセスに割り当てられているロールのリストにアクセスするには、\$ROLES 特殊変数を使用します。ロールをユーザから削除するには、SQL の REVOKE 文を使用します。

\$USERNAME は、USER、CURRENT\_USER、SESSION\_USER の既定のフィールド値として InterSystems SQL で使用されます。

\$SYSTEM.Process.UserName() メソッドを呼び出すことで、現在のプロセスまたは指定されたプロセスのユーザ名を返すことができます。

## 例

以下の例は、現在のプロセスのユーザ名を返します。

### ObjectScript

```
WRITE $USERNAME
```

以下の例は、現在のプロセスのドメイン名を返します。

### ObjectScript

```
WRITE $PIECE($USERNAME, "@", 2)
```

## 関連項目

- ・ ObjectScript [\\$ROLES](#) 特殊変数
- ・ InterSystems SQL : [CREATE TABLE CREATE USER DROP USER ALTER USER GRANT REVOKE %CHECKPRIV](#)



## \$X (ObjectScript)

カーソルの現在の水平位置を含みます。

### 構文

\$X

### 概要

\$X は、カーソルの現在の水平位置を含みます。デバイスに文字が書き込まれると、InterSystems IRIS は \$X を更新して、水平カーソル位置を反映します。

出力される各印刷可能文字は、\$X を 1 つインクリメントします。キャリッジ・リターン (ASCII 13) または改ページ (ASCII 12) は、\$X を 0 (ゼロ) にリセットします。

\$X は、16 ビットの符号なし整数です。\$X の値が 16384 に達すると、0 に戻ります (残りの 2 ビットは、日本語ピッチのエンコードに使用されます)。

SET コマンドを使用して、\$X と \$Y に値を与えることができます。例えば、\$X 値と \$Y 値を更新せずに、物理的カーソル位置を変更する特殊なエスケープ・シーケンスを使用することができます。この場合、エスケープ・シーケンスを使用した後に、SET を使用して正しい値を \$X と \$Y に割り当てます。

### メモ

#### 各国言語サポート文字のマップ

各国言語サポート (National Language Support) ユーティリティ \$X/\$Y タブは、現在のロケールに対する \$X カーソルと \$Y カーソルの移動文字を定義します。詳細は、["各国言語サポートのシステム・クラスの使用法"](#) を参照してください。

#### ターミナル入出力での \$X

以下の表は、\$X での異なる文字の影響を示します。

エコーされる文字	ASCII コード	\$X での効果
<FORM FEED>	12	\$X=0
<RETURN>	13	\$X=0
<LINE FEED>	10	\$X=\$X
<BACKSPACE>	8	\$X=\$X-1
<TAB>	9	\$X=\$X+1
出力できる ASCII 文字	32-126	\$X=\$X+1
出力できない文字 (エスケープ・シーケンスなど)	127-255	" <a href="#">ObjectScript の使用法</a> " を参照してください。

OPEN コマンドと USE コマンドの S(ecret) プロトコルは、エコーをオフにします。また、入力中に \$X が変更されるのを防ぎ、正しいカーソル位置を示します。

WRITE \$CHAR() は \$X を変更します。WRITE \* は、\$X を変更しません。例えば、WRITE \$X, "/", \$CHAR(8), \$X は、バックスペースを実行 (/ 文字を削除) し、それに応じて \$X をリセットして、01 を返します。一方、WRITE \$X, "/", \*, \$X は、バックスペースを実行 (/ 文字を削除) しますが、\$X をリセットしないで 02 を返します (詳細は [WRITE コマンド](#) を参照)。

WRITE \* を使用すると、ターミナルに制御シーケンスを送信しても、\$X は真のカーソル位置を示します。制御シーケンスによってはカーソルを移動するものもあるので、SET コマンドを使用して直接 \$X を設定することができます。例えば、以下のコマンドは Digital VT100 端末 (あるいは同等の端末) でコラム 20 と行 10 にカーソルを移動し、それにしたがって \$X と \$Y を設定します。

#### ObjectScript

```
SET dy=10,dx=20
WRITE *27,*91,dy+1,*59,dx+1,*72
SET $Y=dy,$X=dx
```

デバイスは動作するのに出力しない ANSI 標準制御シーケンス (エスケープ・シーケンスなど) は、\$X 値、\$Y 値と正しいカーソル位置の間での矛盾を生じます。この問題を避けるために WRITE \* (整数式) 構文を使用して、文字列で各文字の ASCII 値を指定します。例えば、以下の形式ではなく

#### ObjectScript

```
WRITE !,$CHAR(27)_"[1m"
WRITE !,$X
```

同じ意味を持つ以下の形式を使用します。

#### ObjectScript

```
WRITE !,*27,*91,*49,*109
WRITE !,$X
```

通常、明示的にカーソルを移動するエスケープ・シーケンスの後、\$X と \$Y を更新して、実カーソル位置を反映します。

%SYSTEM.Process クラスの DX() メソッドを使用すると、\$X による現在のプロセスのエスケープ・シーケンスの処理方法を設定できます。システム全体の既定の動作は、Config.Miscellaneous クラスの DX プロパティで設定できます。

### TCP 通信とプロセス間通信での \$X

WRITE コマンドを使用して、クライアントあるいはサーバの TCP デバイスにデータを送信するとき、InterSystems IRIS は最初にバッファにデータを保存します。また、\$X を更新して、バッファの文字数を反映します。ASCII 文字はレコード部として考えられるため、この数に ASCII 文字 <RETURN> と <LINE FEED> を含みません。

WRITE ! コマンドを使用して \$X バッファをフラッシュすると、\$X が 0 にリセットされ、\$Y 値が 1 つインクリメントされます。WRITE # コマンドを使用して \$X および \$Y バッファをフラッシュした場合は、ASCII 文字 <FORM FEED> が別個のレコードとして書き込まれ、\$X と \$Y の両方が 0 にリセットされます。

### 関連項目

- [WRITE コマンド](#)
- [\\$Y 特殊変数](#)
- [入出力の概要](#)
- [ターミナル入出力](#)
- [ローカル・プロセス間通信](#)
- [TCP クライアント/サーバ通信](#)

## \$Y (ObjectScript)

カーソルの現在の垂直位置を含みます。

### 構文

\$Y

### 概要

\$Y は、カーソルの現在の垂直位置を含みます。デバイスに文字が書き込まれると、InterSystems IRIS は \$Y を更新して、垂直カーソル位置を反映します。

出力される各改行 (新規行) 文字 (ASCII 10) は、\$Y を 1 つインクリメントします。改ページ文字 (ASCII 12) は、\$Y を 0 にリセットします。

\$Y は、16 ビットの符号なし整数です。\$Y の値が 65536 に達すると、0 に戻ります。つまり、\$Y が 65535 の場合、次の出力文字によって 0 にリセットされます。

SET コマンドを使用して、\$X と \$Y に値を与えることができます。例えば、\$X 値と \$Y 値を更新せずに、物理的カーソル位置を変更する特殊なエスケープ・シーケンスを使用することができます。この場合、エスケープ・シーケンスを使用した後に、SET を使用して正しい値を \$X と \$Y に割り当てます。

### メモ

#### 各国言語サポート文字のマップ

各国言語サポート (National Language Support) ユーティリティ \$X/\$Y タブは、現在のロケールに対する \$X カーソルと \$Y カーソルの移動文字を定義します。詳細は、“[各国言語サポートのシステム・クラスの使用法](#)” を参照してください。

#### 端末入出力での \$Y

以下の表は、\$Y での異なる文字の影響を示します。

エコーされる文字	ASCII コード	\$Y での効果
<FORM FEED>	12	\$Y=0
<RETURN>	13	\$Y=\$Y
<LINE FEED>	10	\$Y=\$Y+1
<BACKSPACE>	8	\$Y=\$Y
<TAB>	9	\$Y=\$Y
出力できる ASCII 文字	32-126	\$Y=\$Y

OPEN コマンドと USE コマンドの S(ecret) プロトコルは、エコーをオフにします。また、入力中に \$Y が変更されるのを防ぎ、正しいカーソル位置を示します。

垂直位置を変更する WRITE \$CHAR() は \$Y も変更します。垂直位置を変更する WRITE \* は \$Y を変更しません。例えば、WRITE \$Y, \$CHAR(10), \$Y は改行を行い、\$Y をインクリメントします。一方、WRITE \$Y, \*10, \$Y は改行を行いますが、\$Y をインクリメントしません (詳細は、“[WRITE](#)” コマンドを参照してください)。

WRITE \* は \$Y を変更しないので、制御シーケンスを端末に送信しても、正しいカーソル位置を反映することができます。制御シーケンスによってはカーソルを移動するものもあるので、SET コマンドを使用して直接 \$Y を設定することができます。例えば、以下のコマンドは VT100 ターミナルでコラム 20 と行 10 にカーソルを移動し、それにしたがって \$X と \$Y を設定します。

## ObjectScript

```
SET dy=10,dx=20
WRITE *27,*91,dy+1,*59,dx+1,*72
SET $Y=dy,$X=dx
```

デバイスは動作するのに出力しない ANSI 標準制御シーケンス (エスケープ・シーケンスなど) は、\$X 値、\$Y 値と正しいカーソル位置の間での矛盾を生じます。この問題を避けるために WRITE \* 文を使用して、文字列で各文字の ASCII 値を指定します。例えば、以下のコードを使用する代わりに、

## ObjectScript

```
WRITE $CHAR(27)_"[1m"
```

同じ意味を持つ以下の形式を使用します。

## ObjectScript

```
WRITE *27,*91,*49,*109
```

通常、明示的にカーソルを移動するエスケープ・シーケンスの後、\$X と \$Y を更新して、実カーソル位置を反映します。

## 関連項目

- ・ [\\$X 特殊変数](#)
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)
- ・ [ローカル・プロセス間通信](#)

## \$ZA (ObjectScript)

現在のデバイス上の最後の READ の状況が含まれます。

### 構文

\$ZA

### 概要

\$ZA には、現在のデバイス上の最後の READ の状況が含まれます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉 エラーが返されます。

### ターミナル入出力での \$ZA

\$ZA は、各ビットが特定の情報を示すビット・フラグのシーケンスとして解釈されます。以下のテーブルは、可能な値、その意味、および [モジュロ](#) (#) 演算子と [整数除算](#) (¥) 演算子を使用してそれらをテストする方法を示します。

ビット	テスト	意味
0	\$ZA#2	ブレークが有効であったか否かにかかわらず、〈CTRL-C〉を受信しました。
1	\$ZA¥2#2	READ がタイムアウトになりました。
2	\$ZA¥4#2	入出力エラー
8	\$ZA¥256#2	InterSystems IRIS が無効なエスケープ・シーケンスを検出しました。
9	\$ZA¥512#2	ハードウェアがパリティ・エラーもしくはフレーミング・エラーを検出しました。
11	\$ZA¥2048#2	プロセスが主デバイスから切断されました。
12	\$ZA¥4096#2	COM ポートの場合：CTS (送信可)。モデムからそのコンピュータに送信される、転送が続行可能であることを示すシグナル。TCP デバイスの場合：デバイスはサーバ・モードで機能しています。
13	\$ZA¥8192#2	COM ポートの場合：DSR (データ・セット・レディ)。モデムからそのコンピュータに送信される、処理の準備が整っていることを示すシグナル。TCP デバイスの場合：デバイスは現在、リモート・ホストと対話している接続状態です。
14	\$ZA¥16384#2	True の場合、リングが設定されます。
15	\$ZA¥32768#2	True の場合、キャリア検出が設定されます。
16	\$ZA¥65536#2	CE_BREAK COM ポート・エラー状態
17	\$ZA¥131072#2	CE_FRAME COM ポート・エラー状態
18	\$ZA¥262144#2	CE_IOE COM ポート・エラー状態
19	\$ZA¥524288#2	CE_OVERRUN COM ポート・エラー状態
20	\$ZA¥1048576#2	CE_RXPARITY COM ポート・エラー状態
21	\$ZA¥2097152#2	CE_TXFULL COM ポート・エラー状態
22	\$ZA¥4194304#2	TXHOLD COM ポート・エラー状態。ClearCommError() によって返されるエラー・マスクで、fCtsHold、fDsrHold、fRltdHold、fXoffHold、fXoffSent のいずれかのフィールドが True の場合に、設定されます。

ビット	テスト	意味
24 & 25	\$ZA¥16777216#4	InterSystems IRIS で要求された DTR (データ端末レディ) 設定 : 0 = DTR オフ。1 = DTR オン。2 = DTR ハンドシェーク。(1) に設定されている場合、データの転送および受信の準備が整っていることを示します。

\$ZA がエラーを示す状況は多くありますが、\$ZTRAP にトラップして、プログラム・フローに割り込むことはありません。(ブレークを有効にして <CTRL-C> を実行すると、\$ZTRAP にトラップします)。これらのエラーに関するプログラムは、READ を実行するたびに \$ZA をチェックする必要があります。

COM ポートではビット 12 ~ 15、24 および 25 を使用して、モデム制御ピンのステータスを報告します。これは、ポートの InterSystems IRIS モデム制御チェックがオンになっているかどうかにかかわらず実行できます。ユーザは OPEN または USE コマンドの portstate パラメータ (バイト 8) を設定して、COM ポートの \$ZA エラー報告を有効または無効にできます。エラー報告が有効な場合、ポート・エラー状態はビット 16 ~ 22 に報告されます。詳細は、“[ターミナル入出力](#)”を参照してください。

%SYSTEM.Process クラスの DisconnectErr() メソッドを使用すると、現在のプロセスでのモデム切断が検知されるようになります。システム全体の既定の動作は、Config.Miscellaneous クラスの DisconnectErr プロパティで設定できます。

## 関連項目

- ・ [READ コマンド](#)
- ・ [\\$ZB 特殊変数](#)
- ・ [\\$ZTRAP 特殊変数](#)
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)
- ・ [シーケンシャル・ファイルの入出力](#)

## \$ZB (ObjectScript)

現在の入出力デバイスに関する状況情報を含みます。

### 構文

\$ZB

### 概要

\$ZB は、READ 処理後の現在の入出力デバイス特有の状況情報を含みます。

- ・ ターミナル、シーケンシャル・ファイル、または文字ベースの入出力デバイスからの読み込み時、\$ZB は読み込み処理の最終文字を含みます。これはターミネータ文字 ([Enter] など) や、読み込み処理がターミネータ文字を必要としない場合は入力の最終文字、ターミネータ文字は必要とされるが受け取られない場合 (例えば読み込み処理のタイムアウト) は、NULL 文字列となります。
- ・ ブロック型の入出力デバイスからの読み込み時、\$ZB は入出力バッファに残るバイト数を含みます。\$ZB は、ブロック型の入出力デバイスへの書き込み時の入出力バッファのバイト数も含みます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

\$ZB と \$KEY の両方とも、文字ベースのデバイスやファイルからの読み込み時に、READ 終端文字を返すために使用されます。文字ベースの読み込みでは、これらの 2 つの特殊変数は類似していますが、まったく別のものです。ブロック型の読み込みと書き込みでは、\$ZB を使用します。\$KEY ではブロック型の読み込み処理と書き込み処理がサポートされていないからです。詳細は、“[\\$KEY](#)” を参照してください。

### EOF の動作

既定では、InterSystems IRIS はシーケンシャル・ファイルの最後に到達すると、<ENDOFFILE> エラーを発行します。\$ZB は設定されません。ファイル終端検出時の \$ZB EOF 動作を構成できます。終端検出時、InterSystems IRIS はエラーを発行しませんが、\$ZB を "" (NULL 文字列) に、\$ZEOF を -1 に設定します。

EOF 処理を設定するには、管理ポータルで、[システム管理]、[構成]、[追加の設定]、[互換性] の順に選択します。SetZEOF の現在の設定を表示して編集します。“true” に設定すると、InterSystems IRIS は、\$ZB を "" (NULL 文字列) に、\$ZEOF を -1 に設定します。既定値は “false” です。

%SYSTEM.Process クラスの SetZEOF() メソッドを使用して、現在のプロセスでファイルの最後に到達したときの動作を制御できます。システム全体の既定の動作は、Config.Miscellaneous クラスの SetZEOF プロパティで設定できます。

### ターミナル、またはファイルからの読み込み

\$ZB は、ターミナル、シーケンシャル・ファイル、または他の文字ベースの入出力デバイスに関連する読み込み処理からの終端文字 (または文字シーケンス) を含みます。\$ZB には、以下のいずれかが含まれます。

- ・ キャリッジ・リターンなどの終端文字
- ・ エスケープ・シーケンス (16 文字まで)
- ・ 固定長 READx#n 内の n 番目の文字 (この場合、\$KEY 特殊変数は NULL 文字列を返します)
- ・ READ \*x の 1 文字
- ・ READ がタイムアウトになった後の NULL 文字列 (“”)

例えば、5 秒のタイムアウトを持つ以下の可変長読み込みを考慮してみましょう。

## ObjectScript

```
zbread
  READ !,"Enter number:",num:5
  WRITE !, num
  WRITE !, $ASCII($ZB)
  QUIT
```

READ プロンプトでユーザが 123 と入力し、<RETURN> ([Enter] キー) を押した場合、num 変数に 123 が保存され、\$ZB に <RETURN> (ASCII 10 進コード 13、16 進コード 0D) が保存されます。READ がタイムアウトになった場合、\$ZB は NULL 文字列を含み、\$ASCII("") は値 -1 を返します。

## コマンド行の \$ZB

ターミナル・コマンド行から対話的にコマンドを実行するとき、各コマンド行を実行するために <RETURN> ([Enter]) を押します。\$ZB と \$KEY 特殊変数は、このコマンド行ターミネータ文字を記録します。したがって、\$ZB や \$KEY を使用して読み込み処理の最終状況を返すには、同じコマンド行の一部として変数を設定しなければなりません。

例えば、コマンド

```
>READ x:10
```

をコマンド行から実行する場合、\$ZB をチェックします。これには、読み込み処理の結果が含まれません。コマンド行を実行した <RETURN> 文字が含まれます。読み込み処理の結果を返すには、以下のように同じコマンド行で \$ZB でローカル変数を設定します。

```
>READ x:10 SET rzb=$ZB
```

これにより、読み込み処理で設定された \$ZB の値が保持されます。この読み込み処理値を表示するには、以下のコマンド行文のいずれかを実行します。

```
>WRITE $ASCII(rzb)
; returns -1 for null string (time out),
; returns ASCII decimal value for terminator character
>ZZDUMP rkey
; returns blank line for null string (time out)
; returns hexadecimal value for terminator character
```

## 関連項目

- ・ [READ コマンド](#)
- ・ [WRITE コマンド](#)
- ・ [\\$KEY 特殊変数](#)
- ・ [\\$ZA 特殊変数](#)
- ・ [\\$ZEOF 特殊変数](#)
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)
- ・ [シーケンシャル・ファイルの入出力](#)
- ・ [スプール・デバイス](#)



## \$ZCHILD (ObjectScript)

---

最後の子プロセスの ID を含みます。

### 構文

```
$ZCHILD  
$ZC
```

### 概要

\$ZCHILD は、現在のプロセスが JOB コマンドまたは [\\$ZF\(-100\)](#) 関数を使用して作成した、最後の子プロセスの ID (PID) を含みます。JOB または \$ZF(-100) を使用して子プロセスを作成していない場合、\$ZCHILD は 0 (ゼロ) を返します。

\$ZCHILD が設定されていても、ジョブが正常に開始されたことを示しているわけではありません。プロセスが作成され、パラメータが正常に渡されたことを示しているだけです。

例えば、JOB を使用して存在しないルーチンを作成するとき、新しいジョブはすぐに <NOROUTINE> エラーになるにもかかわらず、\$TEST と \$ZCHILD は、両方とも JOB コマンドが成功したという報告をします。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

### 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$ZF\(-100\) 関数](#)
- ・ [^\\$JOB 構造化システム変数](#)
- ・ [\\$JOB 特殊変数](#)
- ・ [\\$TEST 特殊変数](#)
- ・ [\\$ZPARENT 特殊変数](#)

## \$ZEOF (ObjectScript)

ファイルの最後に到達したかどうかを示すフラグを含みます。

### 構文

\$ZEOF

### 概要

各シーケンシャル・ファイルの READ の後、InterSystems IRIS は、ファイルの最後に到達したかどうかを示す \$ZEOF 特殊変数を設定します。

InterSystems IRIS は、最後に使用されたデバイスのファイル状況に \$ZEOF を設定します。例えば、シーケンシャル・ファイルから読み取った後に主デバイスに書き込んだ場合、InterSystems IRIS は、\$ZEOF を、シーケンシャル・ファイルの EOF 状況から主デバイスの状況に再設定します。したがって、シーケンシャル・ファイルの READ の直後に \$ZEOF の値を確認する必要があります (さらに、必要に応じてその値を変数にコピーします)。

InterSystems IRIS は、\$ZEOF を以下の値に設定します。

-1 は、ファイルの最後に到達したとき

0 は、ファイルの最後ではないとき

この機能を使用するには、シーケンシャル・ファイルの <ENDOFFILE> エラーをオフにします。

- ・ 現在のプロセスに対してこれを無効にするには、**%SYSTEM.Process** クラスの SetZEOF() メソッドを呼び出します。
- ・ これをシステム全体で無効にするには、**Config.Miscellaneous** クラスの SetZEOF プロパティを設定するか、管理ポータルで **[システム管理]**、**[構成]**、**[追加の設定]**、**[互換性]** の順に選択します。SetZEOF の現在の設定を表示して編集します。このオプションは、シーケンシャル・ファイルの読み込み時に InterSystems IRIS が予期しないファイルの最後に到達したときの振る舞いを決めます。“true” に設定すると、InterSystems IRIS は、ファイルの最後に到達したことを示す \$ZEOF 特殊変数を設定します。“false” に設定すると、InterSystems IRIS は <ENDOFFILE> エラーを発行します。既定は “false” です。

ファイルの最後に到達すると、READ は <ENDOFFILE> エラーを発行せずに NULL 文字列を返し、\$ZB を null に、\$ZEOF を -1 に設定します。

\$ZEOF はファイル区切り文字、または入出力エラーを識別しません。\$ZEOF は、ファイル区切り文字を使用して、ファイルが正しく終了しているかを確認しません。入出力エラーは \$ZEOF ではなく、READ コマンド・エラーによって検知されます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

### 関連項目

- ・ [\\$ZB 特殊変数](#)
- ・ [シーケンシャル・ファイルの入出力](#)

# \$ZEOS (ObjectScript)

圧縮ストリーム読み取り時のストリーム終端ステータスが格納されます。

## 構文

\$ZEOS

## 説明

\$ZEOS には、(圧縮された) 着信ストリームの終端が受信され、処理されたかどうかを示すブーリアン値が格納されます。  
\$ZEOS=1 の場合は、圧縮データ・ストリームのストリーム終端が受信されています。\$ZEOS の値は、ストリームの圧縮/解凍がアクティブである (/GZIP=1) 場合にのみ意味があります。ストリームの圧縮/解凍をアクティブにするには、OPEN または USE コマンドから **/GZIP コマンド・キーワード**を発行します。

設定を /GZIP=0 に変更してストリームの圧縮/解凍を無効にする前に、\$ZEOS の値を確認する必要があります。圧縮された受信ストリームの終端が処理される前に /GZIP=0 を指定して USE コマンドを発行すると、<TRANSLATE> エラーが発生します。圧縮された受信ストリームの終端に達していない(\$ZEOS=0) 場合は、\$ZEOS=1 になるまでブロック **READ** コマンドを発行する必要があります。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## 例

次の例では、最初に /GZIP=1 を設定 (圧縮を有効化) します。次に、\$ZEOS=1 を調べるループを実行し、\$ZEOS=1 になるまで READ コマンドを発行します。その後、/GZIP=0 を設定 (圧縮を無効化) します。

### ObjectScript

```
OPEN dev:/GZIP=1
READ block#length
FOR {QUIT:$ZEOS
    READ x:10 }
USE dev:/GZIP=0
```

## 関連項目

- ・ [シーケンシャル・ファイルの入出力](#)
- ・ [TCP クライアント/サーバ通信](#)
- ・ [ターミナル入出力](#)

## \$ZERROR (ObjectScript)

最後のエラーの名前と位置を含みます。

### 構文

```
$ZERROR
$ZE
```

### 説明

\$ZERROR は、最新のエラーの名前、最新のエラーの場所 (該当する場所)、および (特定のエラー・コードの場合は) エラーの原因に関する詳細情報を含んでいます。\$ZERROR は、適切な言語モードの最新のエラーを常に含んでいます。

\$ZERROR 値は、エラーの直後に使用することを目的としています。\$ZERROR 値は複数のルーチン呼び出しにわたっては保持されない可能性があるため、後で使用するために \$ZERROR 値を保持したいユーザはこの値を変数にコピーする必要があります。ユーザは \$ZERROR を使用したらすぐに NULL 文字列 ("") に設定することを強くお勧めします。

\$ZERROR に含まれている文字列は、以下のいずれかの形式になります。

```
<error>
<error>entryref
<error> info
<error>entryref info
```

引数	説明
<error>	エラー名エラー名は、常に山括弧で囲まれた大文字で返されます。空白スペースが含まれることもあります。
entryref	エラーが発生したコードの行の参照。これは、ラベル名とラベルからの行オフセットで構成されます。この後には、^ とプログラム名が続きます。この entryref は、エラー名の山括弧を閉じたすぐ後に続きます。ターミナルから \$ZERROR を呼び出した場合には、この entryref 情報は意味をなさないので返されません。  ZLOAD を使用してルーチン・バッファに最後にロードされたルーチンへの参照。
info	特定のエラーの種類に固有の詳細情報 (以下の表を参照)。この情報は、空白スペースで <error> または <error>entryref と区切られています。info に複数のコンポーネントが存在する場合、それらはコンマで区切られます。

例えば、zerrortest という名前のプログラムは、ZerrorMain という以下のルーチンを含みます。これは、未定義のローカル変数 (fred) のコンテンツの記述を試みます。

### ObjectScript

```
ZerrorMain
TRY {
  SET $ZERROR=""
  WRITE "$ZERROR = ", $ZERROR, !
  WRITE fred }
CATCH {
  WRITE "$ZERROR = ", $ZCVT($ZERROR, "O", "HTML")
}
```

上の例では、最初の \$ZERROR は、\$ZERROR が NULL 文字列にリセットされたためエラーが発生していないので NULL 文字列 ("") を含みます。未定義の変数を記述しようとする、\$ZERROR が設定され、CATCH ブロックにスローされます。この \$ZERROR は <UNDEFINED>ZerrorMain+4^zerrortest \*fred を含みます。ここにはエラーの名前、位置、およびこの種類のエラーに固有の詳細情報が指定されていますこの場合、追加情報は未定義のローカル変数 fred の名前になります。アスタリスクの接頭語は、ローカル変数であることを示しています。(InterSystems IRIS エラー名は山

括弧で囲まれ、この例は Web ブラウザから実行されるため、この例では、\$ZCVT(\$ZERROR,"O","HTML") が使用されます。)

entryref は以下のように表示されます。

```
ZerrorMain+4^zerrortest -- 4 line offset from label ZerrorMain in program zerrortest
ZerrorMain^zerrortest -- no offset from label ZerrorMain in program zerrortest; error occurred in the
label line
+3^zerrortest -- 3 line offset from beginning of program zerrortest; no label precedes the error line
```

\$ZERROR 値の最大長は 512 文字です。その長さを超える値は、512 文字に切り捨てられます。

## AsSystemError() メソッド

%Exception.SystemException クラスの AsSystemError() メソッドは、\$ZERROR と同じ値を返します。詳細は、以下の例を参照してください。

### ObjectScript

```
TRY {
    KILL mylocal
    WRITE mylocal
}
CATCH myerr {
    WRITE "AsSystemError is: ",myerr.AsSystemError(),!
    WRITE "$ZERROR is:      ",$ZERROR
}
```

AsSystemError() は、TRY/CATCH の例外処理ブロック構造内の \$ZERROR よりも優れています。\$ZERROR は、例外処理中に発生するエラーによって上書きされる場合があるためです。

注釈 このメソッドは %Exception.SystemException に固有で、%Exception.SQL では使用できません。

## 一部のエラーに関する追加情報

特定の種類のエラーが発生した場合、\$ZERROR は以下の形式でエラーを返します。

```
<ERRORCODE>entryref info
```

info コンポーネントには、エラーの原因に関する追加情報が含まれます。以下に示すテーブルは、追加情報が格納されるエラーと、その情報の形式の一覧です。エラー・コードは、1 個のスペース文字で info コンポーネントから区切られます。

エラー・コード	情報コンポーネント
<UNDEFINED>	<p>未定義の変数の名前 (添え字が使用されている場合は添え字も含む)。これは、ローカル変数、プロセス・プライベート・グローバル、グローバル、または多次元クラス・プロパティのいずれかです。ローカル変数名の先頭にはアスタリスクが付いています。多次元プロパティの名前はピリオドから始まり、ローカルの変数名と区別されます。</p> <p>%SYSTEM.Process.Undefined() メソッドを設定することで、未定義の変数を参照する際に &lt;UNDEFINED&gt; エラーを生成しないように InterSystems IRIS の動作を変更できます。</p>
<SUBSCRIPT>	<p>エラー内の添え字参照。エラーを生成した行参照 (ルーチンと行オフセット)、添え字付き変数、およびエラーの添え字レベル。構造化システム変数 (SSVN) の場合、行参照 (ルーチンと行オフセット) のみが指定されます。</p> <p>%SYSTEM.Process.NullSubscripts() メソッドを設定することで、NULL 文字列の添え字を含むグローバル変数を参照する際に &lt;SUBSCRIPT&gt; エラーを生成しないように InterSystems IRIS の動作を変更できます。NULL 文字列の添え字は、ローカル変数では許可されません。</p>
<NOROUTINE>	アスタリスクを接頭語とした、参照されるルーチン名
<CLASS DOES NOT EXIST>	アスタリスクを接頭語とした、参照されるクラス名
<PROPERTY DOES NOT EXIST>	アスタリスクを接頭語とした、参照されるプロパティの名前と、それに続くコンマ区切り文字とそのプロパティが存在することになっているクラス名
<METHOD DOES NOT EXIST>	アスタリスクを接頭語とした、呼び出されるメソッドの名前と、それに続くコンマ区切り文字とそのメソッドが存在することになっているクラス名
<PROTECT>	<p>参照されるグローバルの名前と、そのグローバルが格納されているディレクトリの名前 (コンマ区切り)</p> <p>ディスマウントされたデータベースにアクセスする場合に、データベース名を指定します。</p>
<THROW>	アスタリスク接頭語、オブジェクト名の後に DisplayString() メソッドによって返された値が続きます。
<COMMAND>	<p>トランザクションをしていないときに TCOMMIT を呼び出すと、info コンポーネントは *NoTransaction になります。</p> <p>値を返さないユーザ定義関数を呼び出すと、info コンポーネントは、値を返したコマンドの位置を含むメッセージになります。</p>
<DIRECTORY>	先頭にアスタリスクの付いた、無効なディレクトリのフル・パス名。
<FRAMESTACK>	<p>&lt;FRAMESTACK&gt; エラーでプロセスが終了すると、&lt;FRAMESTACK&gt; エラーと追加情報がメッセージとして mgr/messages.log に書き込まれます。情報メッセージには、終了したプロセスのプロセス ID (pid) と、エラーを生成した行参照 (ルーチンと行オフセット) が示されます。例 : (pid) 0 &lt;FRAMESTACK&gt; at +13^  "USER"   test</p>

ルーチン (またはメソッド) のローカル変数名、および未定義のルーチン名、クラス名、プロパティ名、およびメソッド名の先頭にはアスタリスク (\*) が付いています。プロセス・プライベート・グローバルは、接頭語 || によって識別されます。グローバル変数は、接頭語 ^ (キャレット) によって識別されます。クラス名は、% の接頭語で表されます。

以下の例は、エラーの原因を指定する追加エラー情報を示しています。いずれの場合でも、指定された項目が存在しません。生成されたエラーの info コンポーネントは空白スペースでエラー名と区切られていることに注意してください。アスタリスク (\*) はローカルの変数、クラス、プロパティ、またはメソッドを示します。キャレット文字 (^) はグローバルを示し、| はプロセス・プライベート・グローバルを示します。

#### 〈UNDEFINED〉エラーの例

```

UndefTest ;
SET $NAMESPACE="SAMPLES"
KILL x,abc(2)
KILL ^xyz(1,1),^|"USER"|xyz(1,2)
KILL ^||ppg(1),^||ppg(2)
TRY {WRITE x }           // undefined local variable
  CATCH {WRITE $ZERROR,! }
TRY {WRITE abc(2)}       // undefined subscripted local variable
  CATCH {WRITE $ZERROR,! }
TRY {WRITE ^xyz(1,1) }   // undefined global
  CATCH {WRITE $ZERROR,! }
TRY {WRITE ^|"USER"|xyz(1,2) } // undefined global in another namespace
  CATCH {WRITE $ZERROR,! }
TRY {WRITE ^||ppg(1) }   // undefined process-private global
  CATCH {WRITE $ZERROR,! }
TRY {WRITE ^|^"|ppg(2) } // undefined process-private global
  CATCH {WRITE $ZERROR,! }

<UNDEFINED>UndefTest+5^MyProg *x
<UNDEFINED>UndefTest+7^MyProg *abc(2)
<UNDEFINED>UndefTest+9^MyProg ^xyz(1,1)
<UNDEFINED>UndefTest+11^MyProg ^xyz(1,2)
<UNDEFINED>UndefTest+13^MyProg ^||ppg(1)
<UNDEFINED>UndefTest+15^MyProg ^||ppg(2)

```

#### 〈SUBSCRIPT〉エラーの例

```

SubscriptTest ;
DO $SYSTEM.Process.NullSubscripts(0)
KILL abc,xyz
TRY {SET abc(1,2,3,"")=123 }
  CATCH {WRITE $ZERROR,! }
TRY {SET xyz(1,$JUSTIFY(1,1000))=1}
  CATCH {WRITE $ZERROR,! }

<SUBSCRIPT>SubscriptTest+3^MyProg *abc() Subscript 4 is ""
<SUBSCRIPT>SubscriptTest+5^MyProg *xyz() Subscript 2 > 511 chars

```

#### 〈NOROUTINE〉エラーの例

```

NoRoutineTest ;
KILL ^NotThere
TRY {DO ^NotThere }
  CATCH {WRITE $ZERROR,! }
TRY {JOB ^NotThere }
  CATCH {WRITE $ZERROR,! }
TRY {GOTO ^NotThere }
  CATCH {WRITE $ZERROR,! }

<NOROUTINE>NoRoutineTest+2^MyProg *NotThere
<NOROUTINE>NoRoutineTest+4^MyProg *NotThere
<NOROUTINE>NoRoutineTest+6^MyProg *NotThere

```

#### オブジェクト・エラーの例

```

WRITE $SYSTEM.XXQL.MyMethod()
<CLASS DOES NOT EXIST> *%SYSTEM.XXQL

DO $SYSTEM.SQL.MyMethod()
<METHOD DOES NOT EXIST> *MyMethod,%SYSTEM.SQL

SET x=##class(%SQL.Statement).%New()
WRITE x.MyProp
<PROPERTY DOES NOT EXIST> *MyProp,%SQL.Statement

```

#### 〈PROTECT〉エラーの例 (Windows 上) :

```

// user does not have access privileges for %SYS namespace
SET x=^|"%SYS"|var
<PROTECT> ^var,c:\intersystems\iris\mgr\

```

ユーザ定義関数の呼び出し時における〈COMMAND〉エラーの例。この例では、MyFunc QUIT コマンドは値を返しません。これにより、\$\$MyFunc の呼び出しの位置を示す entryref、および QUIT コマンドの位置を示す info メッセージを含む〈COMMAND〉エラーが生成されます。

## ObjectScript

```
Main
  TRY {
    KILL x
    SET x=$$MyFunc(7,10)
    WRITE "returned value is ",x,!
    RETURN
  }
  CATCH { WRITE "$ZERROR = ", $ZCVT($ZERROR,"O","HTML"),! }
MyFunc(a,b)
  SET c=a+b
  QUIT
```

関数を PUBLIC キーワードでプロシージャとして呼び出した際の同じ〈COMMAND〉エラー。

## ObjectScript

```
Main
  TRY {
    KILL x
    SET x=$$MyFunc(7,10)
    WRITE "returned value is ",x,!
    RETURN
  }
  CATCH { WRITE "$ZERROR = ", $ZCVT($ZERROR,"O","HTML"),! }
MyFunc(a,b) PUBLIC {
  SET c=a+b
  QUIT }
```

〈DIRECTORY〉エラーの例 (Windows 上) :

## ObjectScript

```
TRY { SET prev=$SYSTEM.Process.CurrentDirectory("bogusdir")
      WRITE "previous directory: ",prev,!
      RETURN }
CATCH { WRITE "$ZERROR = ", $ZCVT($ZERROR,"O","HTML"),!
       QUIT }
```

## メモ

### ZLOAD とエラー・メッセージ

以下の ZLOAD 操作では、ルーチン・バッファにロードされたルーチン名が後続のエラー・メッセージの entryref の部分に表示されます。この表示は、処理の間中、または ZREMOVE を使用して削除されるまで、あるいは別の ZLOAD によって削除されるか置き換えられるまで続きます。以下のターミナルの例は、このルーチン・バッファの内容を表示する方法を示します。

```
SAMPLES>ZLOAD Sample.Person.1
SAMPLES>WRITE 6/0
<DIVIDE>^Sample.Person.1
SAMPLES>WRITE fred
<UNDEFINED>^Sample.Person.1 *fred
SAMPLES>WRITE ^fred
<UNDEFINED>^Sample.Person.1 ^fred
SAMPLES>ZNAME "USER"
USER>WRITE 7/0
<DIVIDE>^Sample.Person.1
USER>ZREMOVE
USER>WRITE ^fred
<UNDEFINED> ^fred
```



## \$ZERROR とプログラム・スタック

\$ZERROR 文字列の <error> 部分には、最新のエラー・メッセージが含まれます。\$ZERROR 文字列の entryref 部分のコンテンツは、最新エラーのスタック・レベルを反映します。以下のターミナル・セッションは、無意味なコマンド GOBBLEDEGOOK の呼び出しを試みますが、これは <SYNTAX> エラーという結果になります。また、上記で指定された ZerrorMain も実行しますが、\$ZERROR 値 <UNDEFINED> という結果になります。このターミナル・セッション中の次に続く \$ZERROR 値は、以下のようにこのプログラムの呼び出しを反映します。

```
USER>gobbledegook
USER>WRITE $ZERROR
<SYNTAX>
USER>DO ^zerrortest
USER>WRITE $ZERROR
<UNDEFINED>ZerrorMain+2^zerrortest *FRED
USER 2d0>gobbledegook
USER 2d0>WRITE $ZERROR
<SYNTAX>^zerrortest
USER 2d0>QUIT
USER>WRITE $ZERROR
<SYNTAX>^zerrortest
USER>gobbledegook
USER>WRITE $ZERROR
<SYNTAX>
```

## \$ZTRAP が設定されているときの \$ZERROR

エラーが発生し、\$ZTRAP が設定されている場合、InterSystems IRIS は \$ZERROR でエラー・メッセージを返し、\$ZTRAP で指定したエラー・トラップ・ハンドラに分岐します (返される可能性があるエラー・テキストのリストは、["システム・エラー・メッセージ"](#) を参照してください)。

## \$ZERROR の設定

InterSystems IRIS モードのみで、SET コマンドを使用して \$ZERROR を 512 文字までの値に設定することができます。512 文字よりも長い値は 512 に切り捨てられます。

エラーの処理に続いて、\$ZERROR を NULL 文字列 ("") にリセットすることを強くお勧めします。

## 関連項目

- ・ [CATCH コマンド](#)
- ・ [ZTRAP コマンド](#)
- ・ [\\$ECODE 特殊変数](#)
- ・ [\\$ZTRAP 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)
- ・ [システム・エラー・メッセージ](#)

## \$ZHOROLOG (ObjectScript)

InterSystems IRIS の開始以後に経過した秒数を含みます。

### 構文

```
$ZHOROLOG  
$ZH
```

### 概要

\$ZHOROLOG は、直近の InterSystems IRIS の開始後から経過した秒数を含みます。これは、時刻の変更と日付の境界とは独立するカウントです。値は浮動小数点数として表され、秒と秒の小数部を示します。小数桁数は、プラットフォームによって異なります。\$ZHOROLOG は、小数部分の末尾の 0 を切り捨てます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉エラーが返されます。

**注釈** Windows オペレーティング・システムの制限により、Windows システムをハイバネート・モードまたはスタンバイ・モードにすると、\$ZHOROLOG によって予期しない値が返される場合があります。この問題は、\$HOROLOG または \$ZTIMESTAMP の値には影響しません。

### 例

以下の例は、現在の \$ZHOROLOG 値を出力したものです。

#### ObjectScript

```
WRITE $ZHOROLOG
```

これは、1036526.244932 のような値を返します。

以下の例は、\$ZHOROLOG を使用して、イベントの時間制限と、ベンチマークを行う方法を表しています。この例は、100 回の実行でアプリケーションの時間制限を行い、そして平均の実行時間を割り出します。

#### ObjectScript

```
Cycletime  
  SET start=$ZHOROLOG  
  FOR i=1:1:100 { DO Myapp }  
  SET end=$ZHOROLOG  
  WRITE !,"Average run was ",(end-start)/100," seconds."  
  QUIT  
Myapp  
  WRITE !,"executing my application"  
  ; application code goes here  
  QUIT
```

### 関連項目

- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数

# \$ZIO (ObjectScript)

現在のターミナル入出力デバイスに関する情報を含みます。

## 構文

```
$ZIO
$ZI
```

## 概要

\$ZIO は、現在の入出力デバイスに関する情報を含みます。

ターミナルであるターミナル・デバイスの場合、\$ZIO には文字列 TRM: が含まれます。現在のターミナル・デバイスがリモートで接続されていると、\$ZIO にはリモート接続に関する情報が含まれます。

TELNET で接続されているターミナル・デバイスの場合、\$ZIO には host|port が含まれます。

引数	説明
host	IPv4 形式のリモート・ホスト IP アドレス nnn.nnn.nnn.nnn (nnn は 10 進数)、または IPv6 形式のリモート・ホスト IP アドレス h:h:h:h:h:h:h:h (h は 16 進数)。IPv4 および IPv6 の形式に関する詳細は、“ <a href="#">IPv6 アドレスの使用</a> ”を参照してください。
port	リモート IP ポート番号

これら 2 つの値は垂直バー文字で区切ります。例えば、127.0.0.1|23 のようになります。詳細は、%Library.NetworkAddress クラス、データ型クラス %Library.String のサブクラスを参照してください。

現在のデバイスがターミナルでない場合

- ・ ファイルの場合、\$ZIO はファイルの完全なキャノニック形式のパス名を含みます。
- ・ ファイルでない場合、\$ZIO は NULL 文字列を含みます。

以下の例は、現在のデバイス情報を返します。

### ObjectScript

```
SET x=$CASE($ZIO,"TRM:":"a terminal",
            "CON:":"a console",
            "":"neither terminal nor file")
WRITE "The current device is ",x
```

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## 関連項目

- ・ [\\$IO](#) 特殊変数
- ・ [\\$PRINCIPAL](#) 特殊変数
- ・ [入出力の概要](#)
- ・ [ターミナル入出力](#)

## \$ZJOB (ObjectScript)

ジョブの状況情報を含みます。

### 構文

```
$ZJOB
$ZJ
```

### 概要

\$ZJOB は、各ビットがジョブ状況のある特定の機能を表す番号を含みます。\$ZJOB は、状況設定ビットの合計で構成される整数を返します。例えば、\$ZJOB = 5 の場合、これは、1 ビットと 4 ビットが設定されていることを意味します。

個別の \$ZJOB ビット設定をテストするために、[整数除算](#) (¥) 演算子と[モジュロ](#) (#) 演算子を使用できます。例えば、\$ZJOB\#2 とします。ここで x はビット数です。以下のテーブルは、ビット構成 (ビット位置の値により)、設定、およびその意味を示しています。

ビット	設定	意味
1	1	ターミナル・プロンプトから開始したジョブ
	0	ルーチンから開始したジョブ
2	1	JOB コマンドで開始したジョブ
	0	ターミナル・プロンプトにサインインすることによって、またはルーチンから開始したジョブ
4	1	〈INTERRUPT〉 が有効になりました。CTRL-C で実行中のプログラムを中断できます。詳細は、“ <a href="#">BREAK flag</a> ” を参照してください。
	0	OPEN、または USE コマンドで〈INTERRUPT〉が明示的に使用可能にされていないターミナル行で〈INTERRUPT〉 オフ
8	1	〈INTERRUPT〉 受信後にペンディング
	0	〈INTERRUPT〉 未受信。値 8 は、OPEN コマンド、USE コマンド、および CTRL-C によるエラー・トラップでクリアされます。
1024	1	他の条件に関係なく、ジャーナリングはオフ
	0	他の条件がジャーナリングを示していれば、このジョブのジャーナリングはオン

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉 エラーが返されます。

### 例

以下の例では、整数として \$ZJOB を返します。

#### ObjectScript

```
WRITE $ZJOB
```

以下の例では、各 \$ZJOB ビット値を返します。

## ObjectScript

```
WRITE "    bit 1=", $ZJOB\1#2,!
WRITE "    bit 2=", $ZJOB\2#2,!
WRITE "    bit 4=", $ZJOB\4#2,!
WRITE "    bit 8=", $ZJOB\8#2,!
WRITE "bit 1024=", $ZJOB\1024#2
```

\$ZJOB#2 を使用して、ビット 1 を返すこともできます。

## 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$JOB 特殊変数](#)

## \$ZMODE (ObjectScript)

現在の入出力デバイス OPEN パラメータを含みます。

### 構文

```
$ZMODE
$ZM
```

### 概要

\$ZMODE は、現在のデバイス OPEN コマンド、または USE コマンドで指定したパラメータを含みます。返される文字列には、キャノニック形式で現在の入出力デバイスをオープンするために使用されるパラメータが含まれます。これらのパラメータ値は、バックスラッシュ区切り文字で分けられます。TCP/IP 上の "M" のように開いているパラメータは、"PSTE" に正規化されます。"Y" や "K" パラメータ値は、常に最後に配置されます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉エラーが返されます。

### 例

以下の例は、\$ZMODE を使用して現在のデバイスのパラメータを返します。

#### ObjectScript

```
WRITE !,"The current OPEN modes are: ", $PIECE($ZMODE, "\")
WRITE !,"The NLS collation is: ", $PIECE($ZMODE, "\", 2)
WRITE !,"The network encoding is: ", $PIECE($ZMODE, "\", 4)
```

以下の例は、現在のデバイスのパラメータを USE コマンドを使用して設定します。これは、USE コマンドの前後に、\$ZMODE で現在のパラメータをチェックします。特定のパラメータが設定されたことをテストするため、この例はバックスラッシュ区切り文字がある \$PIECE 関数を使用し、包含関係演算子 (I) を使用して値のテストを行います ("[演算子と式](#)" を参照してください)。

#### ObjectScript

```
Zmodetest
WRITE !, $ZMODE
IF $PIECE($ZMODE, "\")["S" {
    WRITE !, "S is set" }
ELSE {WRITE !, "S is not set" }
USE 0:("": "IS": $CHAR(13,10))
WRITE !, $ZMODE
IF $PIECE($ZMODE, "\")["S" {
    WRITE !, "S is set" }
ELSE {WRITE !, "S is not set" }
QUIT
```

```
USER>DO ^zmodetest
```

```
RY¥Latin1¥K¥UTF8¥
```

```
S is not set
```

```
SIRY¥Latin1¥K¥UTF8¥
```

```
S is set
```

### 関連項目

- [OPEN](#) コマンド
- [USE](#) コマンド

- ・ [\\$IO 特殊変数](#)
- ・ [入出力の概要](#)

## \$ZNAME (ObjectScript)

---

現在のルーチン名を収めた特殊変数です。

### 構文

```
$ZNAME  
$ZN
```

### 概要

\$ZNAME には、現在のプロセスで実行されているルーチンの名前が含まれます。通常、これは [ZLOAD](#) によってロードされた現在のルーチンです。現在実行中のルーチンが存在しない場合、\$ZNAME は NULL 文字列を返します。

ZLOAD でルーチンをロードすると、そのルーチンは、すべてのネームスペースで、現在のプロセスに対して現在ロードされているルーチンになります。したがって、ロード元のネームスペースだけではなく任意のネームスペースから、\$ZNAME を使用して、現在ロードされているルーチンの名前を表示できます。

ルーチン名は、大文字と小文字を区別します。

ZLOAD でルーチンをロードしようとして失敗した場合、現在ロードされているルーチンは削除され、\$ZNAME は NULL 文字列に設定されることに注意してください。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉エラーが返されます。

\$ZNAME 値は、以下のコマンドのいずれかで設定することができます。

- ・ [ZLOAD](#) コマンド
- ・ [ZSAVE](#) コマンド
- ・ 引数なしの [ZREMOVE](#) コマンド (NULL 文字列に設定します)
- ・ [DO](#) コマンド
- ・ ^routine を使用した [GOTO](#) コマンド



# \$ZNSPACE (ObjectScript)

現在のネームスペース名を含みます。

## 構文

```
$ZNSPACE
```

## 概要

\$ZNSPACE は、現在のネームスペース名を含みます。\$ZNSPACE の設定で、現在のネームスペースを変更できます。以下のように現在のネームスペース名を取得します。

### ObjectScript

```
SET ns=$ZNSPACE
WRITE ns
```

また、以下のように **%SYSTEM.SYS** クラスの `Namespace()` メソッドを呼び出すことで、現在のネームスペース名を取得することもできます。

### ObjectScript

```
SET ns=%SYSTEM.SYS.Namespace()
```

以下のように、**%SYS.Namespace** クラスの `Exists()` メソッドを使用して、ネームスペースが定義されているかどうかをテストすることができます。

### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

これらのメソッドは、“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。

UNIX® システムでは、既定のネームスペースはシステム構成オプションとして設定されます。Windows システムでは、コマンド行スタートアップ・オプションを使用して設定できます。

ネームスペース名は、大文字と小文字を区別しません。InterSystems IRIS は常に、明示的なネームスペース名をすべて大文字、暗黙のネームスペース名をすべて小文字で表示します。

指定プロセスのネームスペース名を取得するには、以下の例に示すように **%SYS.ProcessQuery** クラスのメソッドを使用します。

### ObjectScript

```
WRITE ##CLASS(%SYS.ProcessQuery).%OpenId($JOB).NamespaceGet()
```

## 現在のネームスペースの設定

ZNSPACE コマンド、SET \$NAMESPACE、SET \$ZNSPACE、または %CD ユーティリティを使用して、現在のネームスペースを変更することができます。

- ・ ネームスペースを変更するお勧めの方法は、ターミナル・コマンド・プロンプトから、[ZNSPACE](#) コマンドを使用することです。SET \$ZNSPACE は、ZNSPACE コマンドと機能的には同じです。
- ・ コード・ルーチン内で、現在のネームスペースを変更するお勧めの方法は、NEW \$NAMESPACE に続けて SET \$NAMESPACE=namespace を使用することです。NEW \$NAMESPACE および SET \$NAMESPACE を使用すること

によって、メソッドが終わったときまたは予期しないエラーが発生したときに前のネームスペースに自動的に戻すネームスペース・コンテキストを確立します。詳細は、“[\\$NAMESPACE](#)” 特殊変数を参照してください。

SET \$ZNSPACE を使用して、プロセスの現在のネームスペースを変更できます。文字列リテラル、または引用符付き文字列に評価される変数や式として新しいネームスペースを指定します。明示的ネームスペース("namespace")、あるいは暗黙のネームスペース("^system^dir" や "^dir") を指定することができます。

現在のネームスペースを指定した場合、SET \$ZNSPACE は処理を実行せず、エラーも返しません未定義のネームスペースを指定した場合、SET \$ZNSPACE は <NAMESPACE> エラーを生成します。

NEW を使用して、\$ZNSPACE 特殊変数をリセットすることはできません。

## 例

次の例では、現在のネームスペースが USER でない場合、SET \$ZNSPACE コマンドによって現在のネームスペースが USER に変更されます。IF テストのため、ネームスペースはすべて大文字で指定される必要があることに注意してください。

### ObjectScript

```
SET ns="USER"
IF $ZNSPACE=ns {
    WRITE !,"Namespace already was ", $ZNSPACE }
ELSEIF 1=##class(%SYS.Namespace).Exists(ns) {
    WRITE !,"Namespace was ", $ZNSPACE
    SET $ZNSPACE=ns
    WRITE !,"Set namespace to ", $ZNSPACE }
ELSE { WRITE !,ns," is not a defined namespace" }
QUIT
```

この例では、UnknownUser が %DB\_IRISSYS ロールおよび %DB\_USER ロールを割り当てている必要があります。

## 関連項目

- [SET コマンド](#)
- [ZNSPACE コマンド](#)
- [\\$NAMESPACE 特殊変数](#)
- [ネームスペースの構成](#)

# \$ZORDER (ObjectScript)

次のグローバル・ノードの値を含みます。

## 構文

```
$ZORDER
$ZO
```

## 概要

\$ZORDER は、現在のグローバル参照の後、順番に次のグローバル・ノードの値を含みます (\$ORDER シーケンスではなく \$QUERY シーケンスで)。次のグローバル・ノードが存在しない場合、\$ZORDER にアクセスすると、\$ZORDER によって正常にアクセスされた最後のグローバルを示す <UNDEFINED> エラーが返されます。<UNDEFINED> の詳細は、"\$ZERROR" を参照してください。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## 例

以下の例は、WHILE ループを使用して \$ZORDER を繰り返し呼び出し、一連の添え字ノードを検索します。

### ObjectScript

```
SET ^| a="groceries"
SET ^| a(1)="fruit"
SET ^| a(1,1)="apples"
SET ^| a(1,2)="oranges"
SET ^| a(3)="nuts"
SET ^| a(3,1)="peanuts"
SET ^| a(2)="vegetables"
SET ^| a(2,1)="lettuce"
SET ^| a(2,2)="tomatoes"
SET ^| a(2,1,1)="iceberg"
SET ^| a(2,1,2)="romaine"
SET $ZERROR="unset"
WRITE !,"last referenced: ",^| |a(1,1)
WHILE $ZERROR="unset" {
    WRITE !,$ZORDER }
QUIT
```

上の例は、最後に参照されたグローバル (この例では、プロセス・プライベート・グローバル) ^|a(1,1) で始まります。\$ZORDER は ^|a(1,1) の値を含みませんが、以降の値を含みます。\$ZORDER への呼び出しは、(1,2)、(2)、(2,1)、(2,1,1)、(2,1,2)、(2,2)、(3)、(3,1) の順序で添え字ツリー・ノードを検索します。それぞれの WRITE \$ZORDER は連続する各ノードにデータ値を表示します。ノードがなくなると、エラー <UNDEFINED> ^| |a(3,1) を生成します。^|a(3,1) が定義されていないことに注意してください。^|a(3,1) が指定されるのは、\$ZORDER がその後に別のグローバルを検出できないためです。

## 関連項目

- [\\$ORDER](#) 関数
- [\\$QUERY](#) 関数
- [\\$ZERROR](#) 特殊変数

## \$ZPARENT (ObjectScript)

---

現在のプロセスの親プロセスの ID を含みます。

### 構文

```
$ZPARENT  
$ZP
```

### 概要

\$ZPARENT は、JOB コマンドで現在のプロセスを作成した親プロセスの ID を含みます。現在のプロセスが JOB コマンドで作成されていない場合は、\$ZPARENT は 0 (ゼロ) を含みます。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、〈SYNTAX〉エラーが返されます。

### 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$ZCHILD](#) 特殊変数
- ・ [\\$JOB](#) 特殊変数

## \$ZPI (ObjectScript)

---

円周率の値を含みます。

### 構文

`$ZPI`

### 概要

\$ZPI は、円周率 (3.141592653589793238) の値を小数点以下 18 桁まで含みます。

この値は、サイン関数 [\\$ZSIN](#) のような三角関数で頻繁に使用されます。

## \$ZPOSITION (ObjectScript)

---

シーケンシャル・ファイルの読み取り中の、現在のファイル位置を含みます。

### 構文

```
$ZPOSITION  
$ZPOS
```

### 説明

\$ZPOSITION には、シーケンシャル・ファイルの読み取り中の、現在のファイル位置が含まれます。この処理中に読み取られているシーケンシャル・ファイルが存在しない場合、\$ZPOSITION は 0 (ゼロ) を含みます。

シーケンシャルを読み取りするファイルを開いていると、そのデバイスからの READ ごとに、ファイルの中の次の読み取り開始位置に \$ZPOSITION が設定されます。\$ZPOSITION 値は、READ、READ \*、または READ #n の各操作が終了した時点での実際のファイル・オフセットをバイト単位で表したものです。マルチバイト文字セットを読み取る場合は適切な配慮が必要になります。

現在のファイル位置は、\$ZSEEK 関数を使用して設定できます。この特殊変数は、SET コマンドを使用して変更することはできません。使用しようとすると、<SYNTAX> エラーが返されます。

### 関連項目

- ・ [READ コマンド](#)
- ・ [\\$ZSEEK 関数](#)
- ・ [シーケンシャル・ファイルの入出力](#)

# \$ZREFERENCE (ObjectScript)

現在のグローバル参照を含みます。

## 構文

```
$ZREFERENCE
$ZR
```

## 概要

\$ZREFERENCE は、最後の**グローバル**参照の名前と添え字を含みます。これは、ネイキッド・インジケータとして知られています。

**注釈** 最後のグローバル参照とは、最後にアクセスされたグローバル・ノードです。通常は最後に行われたグローバルへの明示的な参照です。ただし、特定のコマンドは内部的に \$ORDER および \$QUERY 関数を使用してグローバル添え字を検索したり、内部的に他のグローバルを参照したりすることがあります。このような状況が発生すると、\$ZREFERENCE には最後にアクセスされたグローバル・ノードが含まれ、コマンド引数として渡されたグローバル・ノードではない場合があります。

最後のグローバル参照は、グローバル (^myglob)、または**プロセス・プライベート・グローバル** (^||myppg) のいずれかです。\$ZREFERENCE は、後続でその変数に対してどのプロセス・プライベート・グローバル接頭語が使用されるかに関わらず、その変数に対して最初で使用されていた形式でプロセス・プライベート・グローバル接頭語を返します。以下の \$ZREFERENCE の説明において、“グローバル”という言葉は、両方の種類の変数を指します。

最後のグローバル参照とは、コマンドまたは関数から最近参照されたグローバルのことです。ObjectScript は処理を左から右の順に実行するので、最後のグローバル参照は、常に一番右端のグローバルになります。コマンドまたは関数が複数の引数を取る場合、一番右端の引数で指定されたグローバルが最後のグローバル参照です。引数が複数のグローバル参照を含む場合、一番右端に指定されたグローバルが最後のグローバル参照です。この厳密な左から右の順は、括弧を使用して演算の順序を定義する場合でも、正しく保持されます。

明示的なグローバル参照が発行されるとき、InterSystems IRIS は \$ZREFERENCE を更新します。グローバル参照に評価される式 (ローカル変数など) を呼び出しても、\$ZREFERENCE は更新されません。

最後のグローバル参照が失敗していたとしても、\$ZREFERENCE には、最後のグローバル参照が含まれます。コマンドが未定義のグローバルを参照して <UNDEFINED> エラーが返された場合でも、グローバルが定義されたかのように、InterSystems IRIS はそのグローバル参照に \$ZREFERENCE を更新します。この動作は、%SYSTEM.Process.Undefined() メソッドの設定には影響されません。

このコマンドの実行に失敗していたとしても、多くの場合、\$ZREFERENCE には、最後のグローバル参照が含まれます。各グローバルが参照されると、InterSystems IRIS によって \$ZREFERENCE が更新されます。例えば、(数値を 0 で割ろうとすると) <DIVIDE> エラーを返すコマンドは、エラーが発生する前にコマンド内で参照された最後のグローバルに \$ZREFERENCE を更新します。しかし、<SYNTAX> エラーは \$ZREFERENCE を更新しません。

## 長いグローバル名

グローバル名が 31 文字より長い (^ などのグローバル接頭語を除く) 場合、\$ZREFERENCE は 31 文字に短縮したグローバル名を返します。長いグローバル名の処理の詳細は、“**グローバル**” を参照してください。

## ネイキッド・グローバル参照

最後のグローバル参照が**ネイキッド・グローバル参照**である場合、\$ZREFERENCE には現在のネイキッド・グローバル参照の完全な形式 (外部で読み取り可能) が含まれます。詳細は、以下の例を参照してください。

## ObjectScript

```
SET ^MyData(1)="fruit"
SET ^MyData(1,1)="apples" ; Full global reference
SET ^{2)="oranges"        ; Naked global reference,
                           ; implicitly ^MyData(1,2)
WRITE !,$ZREFERENCE        ; Returns ^MyData(1,2)"
```

ネイキッド・グローバル参照の詳細は、“[多次元ストレージの使用法 \(グローバル\)](#)” を参照してください。

## Extended Global Reference (拡張グローバル参照)

拡張グローバル参照は、現在のネームスペース以外のネームスペースにあるグローバルの参照に使用されます。拡張グローバル参照を使用して、コマンドがグローバル変数を参照する場合、\$ZREFERENCE 値はその拡張グローバル参照を含みます。InterSystems IRIS は、以下の状況において、拡張グローバル参照を返します。

- 最後のグローバル参照が、拡張グローバル参照を使用して別のネームスペースにあるグローバルを参照する場合。
- 最後のグローバル参照が、拡張グローバル参照を使用して現在のネームスペースにあるグローバルを参照する場合。
- 最後のグローバル参照がリモート参照 (リモート・システム上のグローバル) である場合。

いかなる場合でも、\$ZREFERENCE は、グローバル参照でどのように指定されたかに関わらず、ネームスペース名をすべて大文字で返します。

グローバル添え字と拡張グローバル参照の詳細は、“[グローバルについての正式な規則](#)” および “[拡張グローバル参照](#)” を参照してください。

## \$ZREFERENCE を更新する処理

\$ZREFERENCE 特殊変数は NULL 文字列(“)に初期化されます。現在のネームスペースを変更すると、\$ZREFERENCE は NULL 文字列にリセットされます。

以下の処理は、最近参照されたグローバルに \$ZREFERENCE を設定します。

- グローバルを引数として使用するコマンドまたは関数。複数のグローバルを使用する場合、\$ZREFERENCE は、最も右に位置するグローバルに設定されます (ただし、\$ORDER を除く)。
- グローバルを後置条件式として使用するコマンド。
- 未定義のグローバルを参照して、〈UNDEFINED〉エラーを生成するか、またはグローバルを定義する (\$INCREMENT の場合) コマンドまたは関数。

## \$ZREFERENCE の設定

SET コマンドを使用すると、この特殊変数を以下のように設定することができます。

- NULL 文字列(“)に設定します。これを実行すると、ネイキッド・インジケータを削除してしまいます。次のグローバル参照が[ネイキッド・グローバル参照](#)の場合、InterSystems IRIS は〈NAKED〉エラーを返します。
- 有効なグローバル参照 (定義済みまたは未定義) に設定します。これによって、その後のネイキッド参照では、ユーザによって設定された値が最後の実際のグローバル参照であるかのように使用されます。

SET コマンドを使用して、他の方法で \$ZREFERENCE を変更することはできません。変更を試みると、〈SYNTAX〉エラーが返されます。

## 例

以下の例は、最後のグローバル参照を返します。



## ObjectScript

```
SET ^a(1,1)="Hello" ; Full global reference
SET ^{(2)}=" world!" ; Naked global reference
WRITE $ZREFERENCE
```

これは、以下を返します。

```
^a(1,2)
```

以下の例は、複数の異なるコマンドからグローバル参照を返します。WRITE と ZWRITE は、同じグローバル参照でも異なる表示を行うことに注意してください。

## ObjectScript

```
SET (^barney,^betty,^wilma,^fred)="flintstone"
WRITE !,$ZREFERENCE
KILL ^flies
WRITE !,$ZREFERENCE
WRITE !,^fred
WRITE !,$ZREFERENCE,!
ZWRITE ^fred
WRITE !,$ZREFERENCE
```

これは、以下を返します。

```
^fred ; last of several globals set in left-to-right order
^flies ; KILL sets global indicator, though no global to kill
flintstone ; WRITE global
^fred ; global from WRITE
^fred="flintstone" ; ZWRITE global
^fred("") ; global from ZWRITE
```

以下の例は、拡張グローバル参照を返します。ネームスペース名は常に、大文字で返されることに注意してください。

## ObjectScript

```
SET ^["samples"]a(1,1)="Hello"
SET ^{(2)}=" world!"
WRITE $ZREFERENCE
QUIT
```

これは、以下を返します。

```
^["SAMPLES"]a(1,2)
```

以下の例は、最後のグローバル参照を返します。この場合は ^a(1) です。これは、\$LENGTH 関数の引数として使用されます。

## ObjectScript

```
SET ^a(1)="abcdefghijklmnopqrstuvwxyz"
SET ^b(1)="1234567890"
SET x=$LENGTH(^a(1))
WRITE $ZREFERENCE
QUIT
```

以下の例では、\$ZREFERENCE に設定されている値が最後のグローバル参照であるかのように返されます。この場合、値は ^a(1,1) です。

## ObjectScript

```
SET ^a(1,1)="abcdefghijklmnopqrstuvwxyz"
SET ^b(1,1)="1234567890"
WRITE !,^(1)
           ; Naked reference uses last global
SET $ZREFERENCE="^(1,1)"
WRITE !,$ZREFERENCE
WRITE !,^(1)
           ; Naked reference uses $ZREFERENCE
           ; value, rather than last global.
```

以下の例は、拡張グローバル参照を設定します。引用符が二重になっていることに注目してください。

## ObjectScript

```
KILL ^x
WRITE !,$ZREFERENCE
SET $ZREFERENCE="[ "samples"]a(1,2)"
WRITE !,$ZREFERENCE
```

## 関連項目

- ・ [ZNSPACE コマンド](#)
- ・ [\\$NAMESPACE 特殊変数](#)
- ・ [\\$ZNSPACE 特殊変数](#)
- ・ [ネームスペースの構成](#)
- ・ [グローバルについての正式な規則](#)
- ・ [ネイキッド・グローバル参照](#)

# \$ZSTORAGE (ObjectScript)

プロセスに使用可能な最大メモリです。

## 構文

```
$ZSTORAGE
$ZS
```

## 概要

\$ZSTORAGE には、ジョブのプロセス・プライベート・メモリ用の最大メモリ量 (KB 単位) が格納されます。このメモリは、ローカル変数、スタック、他のテーブルに利用できます。このメモリ制限には、ルーチン・オブジェクト・コードのスペースは含まれていません。このメモリは、必要に応じて (例えば配列を割り当てるときなどに) プロセスに割り当てられます。

一度このメモリがプロセスに割り当てられると、通常はプロセスが終了するまで、その割り当ては解除されません。ただし、大量の (例えば 32MB を超える) メモリが使用されてから解放された場合、可能であれば、InterSystems IRIS は割り当て解除されたメモリをオペレーティング・システムに戻します。

\$ZSTORAGE を使用して、最大メモリ・サイズを設定することもできます。例えば、以下の文はジョブの最大プロセス・プライベート・メモリを 524288 KB に設定します。

### ObjectScript

```
SET $ZSTORAGE=524288
```

\$ZSTORAGE を変更すると、**\$STORAGE** 特殊変数の初期値が変化します。これはプロセスで使用できる現在メモリ (バイト) です。

- ・ 最大値 : SET \$ZSTORAGE=-1 で \$ZSTORAGE を最大値 2147483647 に設定します。
- ・ 最小値 : SET \$ZSTORAGE=256 で \$ZSTORAGE を最小値 256 に設定します。
- ・ 既定値 : \$ZSTORAGE の既定値は、システム構成設定 [プロセスあたりの最大メモリ (KB)] で設定された値です。管理ポータルで、[システム管理]、[構成]、[システム構成]、[メモリと開始設定] を選択します。[プロセスあたりの最大メモリ (KB)] を変更すると、続いて開始されたプロセスの \$ZSTORAGE 値も変更されます。ただし、現在のプロセスの \$ZSTORAGE 値には影響を与えません。

\$ZSTORAGE の値が、最大値よりも大きい場合、自動的に既定の最大値または最小値に設定されます。\$ZSTORAGE は整数値に設定されます。InterSystems IRIS では、いかなる小数部も切り捨てられます (指定した場合)。

**注釈** オペレーティング・システムによって、実行中のアプリケーションの最大メモリ割り当てに上限が課される場合があります。InterSystems IRIS インスタンスがこのような上限の対象である場合、プロセスは \$ZSTORAGE で指定されたメモリの一部を取得できず、その結果 <STORE> エラーが発生することがあります。

## 例

以下の例では、\$ZSTORAGE を最大値および最小値に設定します。\$ZSTORAGE を、最小値より少ない値 (16) に設定しようとすると、自動的に \$ZSTORAGE は最小値 (256) に設定されます。

### ObjectScript

```
SET $ZS=256
WRITE "minimum storage=", $ZS, !
SET $ZS=16
WRITE "set to < minimum sets to minimum storage=", $ZS, !
SET $ZS=-1
WRITE "maximum storage=", $ZS, !
```

## 関連項目

[SET コマンド](#)

[KILL コマンド](#)

[\\$STORAGE 特殊変数](#)

# \$ZTIMESTAMP (ObjectScript)

UTC (Coordinated Universal Time) 形式で、現在の日付と時刻を含みます。

## 構文

```
$ZTIMESTAMP
$ZTS
```

## 概要

\$ZTIMESTAMP には、現在の日付と時刻が UTC (Coordinated Universal Time) 値として含まれます。UTC は、時刻と日付の世界標準です。この値は、ローカル時刻 (および日付) の値と異なる場合がほとんどです。

\$ZTIMESTAMP は、日付と時刻を次の形式の文字列として表します。

```
dddd,sssss.ffffff
```

dddd は、1840 年 12 月 31 日からの経過日数を示す整数です。sssss は、現在の日付の午前 0 時からの経過秒数を示す整数です。ffffff は、秒の小数部を示す可変桁数です。精度の小数桁数 ffffff は 6 から 9 です。末尾のゼロは削除されます。この形式は \$HOROLOGY と似ていますが、\$HOROLOGY には秒の小数部は含まれません。

現在の日付と時間 (UTC) が以下のようにになると仮定します。

```
2018/02/22 15:17:27.9843342
```

そのとき、\$ZTIMESTAMP は以下の値を持ちます。

```
64701.55047.9843342
```

\$ZTIMESTAMP は、タイム・ゾーンとは独立した UTC (Coordinated Universal Time) を報告します。そのため、ローカル時間帯に関係なく、一様に時間を記録するタイムスタンプとして \$ZTIMESTAMP を使用することができます。これは、ローカル時刻値とローカル日付値の両方と異なる場合があります。

\$ZTIMESTAMP の時刻値は、秒数とその小数部で時刻をカウントする 10 進数値です。秒以下の桁数は、0 から 9 桁で、コンピュータの時刻クロックの有効桁数によって決定します。\$ZTIMESTAMP は末尾の 0 や、この小数部での末尾の小数点を抑制します。午前 0 時から 1 秒以内は、秒数が 0 です。ffffff (例えば、0.1234567) と表されることに注意してください。この数値は、値の文字列ソート順に影響を与える ObjectScript の [キャノニック形式](#) (例えば、.1234567) ではありません。先頭にプラス記号 (+) を追加して、数値をキャノニック形式に強制的に変換してから、ソート操作を実行できます。

現在の日付と時刻は、さまざまな方法で返すことができますが、その違いを以下に示します。

- ・ \$ZTIMESTAMP には、秒の小数部を含む UTC の日付と時刻が InterSystems IRIS ストレージ (\$HOROLOGY) 形式で格納されます。
- ・ \$NOW は、現在のプロセスのローカル日付とローカル時刻を返します。ローカル時刻調整 (サマータイムなど) は適用されません。パラメータ値のない \$NOW は、\$ZTIMEZONE 特殊変数の値からローカル・タイム・ゾーンを決定します。パラメータ値のある \$NOW は、指定されたタイム・ゾーン・パラメータに対応する時刻と日付を返します。  
\$NOW(0) は、UTC による日付と時刻を返します。\$ZTIMEZONE の値は無視されます。\$NOW は、InterSystems IRIS ストレージ (\$HOROLOGY) 形式で日付と時刻を返します。秒の小数部は含まれます。
- ・ \$HOROLOGY には、ローカル調整された日付と時刻が InterSystems IRIS ストレージ形式で格納されます。秒の小数部は記録されません。\$HOROLOGY による秒の小数部の処理方法は、オペレーティング・システム・プラットフォームによって異なります。Windows では、小数部はすべて切り上げられます。UNIX® では、小数部は切り捨てられます。

注釈 ローカル時刻と UTC 時刻を比較する際には注意が必要です。

- ・ UTC 時刻をローカル時刻に変換するための推奨される方法は、\$ZDATETIMEH(utc,-3) 関数を使用することです。この関数によって、ローカル時刻調整が調整されます。
- ・  $\$ZTIMEZONE * 60$  の値を加算または減算するだけで、ローカル時刻と UTC 時刻を相互変換することはできません。これは、多くの場合、ローカル時刻調整 (季節によってローカル時刻を 1 時間調整する [サマータイム](#) など) に合わせてローカル時刻が調整されるためです。このようなローカル時刻調整は、\$ZTIMEZONE には反映されません。
- ・ UTC 時刻は、グリニッジ子午線からのタイム・ゾーンのカウントを使用して計算されます。これは、ローカル・グリニッジ時刻と同じではありません。グリニッジ標準時 (GMT) とは紛らわしい用語であり、グリニッジのローカル時刻は、冬の間は UTC と同じで、夏の間は、UTC と 1 時間違います。これは、英国夏時間と呼ばれるローカル時刻の変更が適用されるためです。
- ・ UTC からのタイムゾーン・オフセットとローカル時刻調整 (季節によるサマータイムへの移行など) の両方が、日付と時刻に影響します。ローカル時刻から UTC 時刻への変換 (またはその逆の変換) を行うと、日付と時刻の両方が変更される場合があります。

この特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## 協定世界時の変換

次の例のように、tformat 値 7 または 8 を指定して \$ZDATETIME および \$ZDATETIMEH 関数を使用することで、ローカル時刻情報を UTC として表すことができます。

### ObjectScript

```
WRITE !,$ZDATETIME($ZTIMESTAMP,1,1,2)
WRITE !,$ZDATETIME($HOROLOG,1,7,2)
WRITE !,$ZDATETIME($HOROLOG,1,8,2)
WRITE !,$ZDATETIME($NOW(),1,7,2)
WRITE !,$ZDATETIME($NOW(),1,8,2)
```

上記の \$ZDATETIME 関数はすべて、現在の時刻をローカル時刻ではなく UTC として返します。ローカル時刻からのこれらの時刻値変換は、[\\$NOW](#) がローカル時刻調整を行わないため、異なる場合があります。\$ZTIMESTAMP および [\\$HOROLOG](#) は、ローカル時刻調整を行い、必要に応じてこの調整に従って日付を調整します。\$ZTIMESTAMP の表示値と、tformat 値 7 または 8 で変換された表示値は同一ではありません。tformat 値 7 および 8 によって、時刻値の前に文字“T”、後に文字“Z”が挿入されます。また、\$HOROLOG 時刻には 1 秒以下の秒数が含まれないので、上の例の precision 2 によって小数部にゼロが埋め込まれます。

以下の構文形式のいずれかを使用して TimeStamp() クラス・メソッドを呼び出すことで、\$ZTIMESTAMP と同じタイム・スタンプ情報を取得することができます。

### ObjectScript

```
WRITE !,$SYSTEM.SYS.TimeStamp()
WRITE !,##class(%SYSTEM.SYS).TimeStamp()
```

詳細は、“[\\$SYSTEM](#)” 特殊変数と、“インターシステムズ・クラス・リファレンス” の “[%SYSTEM.SYS](#)” クラスを参照してください。

## 例

以下の例では、\$ZTIMESTAMP の値をローカル時刻に変換し、それを \$NOW() と \$HOROLOG の 2 つのローカル時刻表現と比較します。

## ObjectScript

```

SET stamp=$ZTIMESTAMP,clock=$HOROLOG,miliclock=$NOW()
WRITE !,"local date and time: ", $ZDATETIME(clock,1,1,2)
WRITE !,"local date and time: ", $ZDATETIME(miliclock,1,1,2)
WRITE !,"UTC date and time:      ", $ZDATETIME(stamp,1,1,2)
IF $PIECE(stamp,"",2) = $PIECE(clock,"",2) {
    WRITE !,"Local time is UTC time" }
ELSEIF $PIECE(stamp,"") '= $PIECE(clock,"") {
    WRITE !,"Time difference affects date" }
ELSE {
    SET localutc=$ZDATETIMEH(stamp,-3)
    WRITE !,"UTC converted to local: ", $ZDATETIME(localutc,1,1,2)
}
QUIT

```

次の例は、\$ZTIMESTAMP と \$HOROLOG によって返される値を比較し、\$ZTIMESTAMP の時刻部分がどのように変換されるかを示します (この簡単な例では、サマータイムなどのローカル時刻調整に合わせて1つの調整のみが行われます。その他のタイプのローカル時刻調整では、clocksecs および stampsecs に矛盾する値が含まれることがあります)。

## ObjectScript

```

SET stamp=$ZTIMESTAMP,clock=$HOROLOG
WRITE !,"local date and time: ", $ZDATETIME(clock,1,1,2)
WRITE !,"UTC date and time:      ", $ZDATETIME(stamp,1,1,2)
IF $PIECE(stamp,"") '= $PIECE(clock,"") {
    WRITE !,"Time difference affects date" }
SET clocksecs=$EXTRACT(clock,7,11)
SET stampsecs=$EXTRACT(stamp,7,11)-($ZTIMEZONE*60)
IF clocksecs=stampsecs {
    WRITE !,"No local time variant"
    WRITE !,"Local time is timezone time" }
ELSE {
    SET stampsecs=stampsecs+3600
    IF clocksecs=stampsecs {
        WRITE !,"Daylight Saving Time variant:"
        WRITE !,"Local time offset 1 hour from timezone time" }
    ELSE { WRITE !,"Cannot reconcile due to local time variant" }
}
QUIT

```

## 関連項目

- ・ [\\$NOW](#) 関数
- ・ [\\$ZDATETIME](#) 関数
- ・ [\\$ZDATETIMEH](#) 関数
- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMEZONE](#) 特殊変数

## \$ZTIMEZONE (ObjectScript)

グリニッジ子午線からのタイム・ゾーン・オフセットを含みます。

### 構文

```
$ZTIMEZONE  
$ZTZ
```

### 概要

\$ZTIMEZONE には以下の 2 つの使用方法があります。

- ・ お使いのコンピュータのローカル・タイム・ゾーン・オフセットを返します。
- ・ 現在のプロセスのローカル・タイム・ゾーン・オフセットを設定します。

\$ZTIMEZONE には、グリニッジ子午線からのタイム・ゾーン・オフセットが分単位で含まれています (グリニッジ子午線には、英国とアイルランドのすべてが含まれています)。このオフセットは、-1440 から 1440 の間の符号付き整数として表されます。グリニッジの西側タイム・ゾーンは、正の数として指定され、グリニッジの東側タイムゾーンは負の数として指定されます (すべてのタイム・ゾーンの違いは整数時間ではないので、タイム・ゾーンは分で表されます)。既定では、\$ZTIMEZONE はコンピュータのオペレーティング・システムに対するタイム・ゾーン設定に初期化されます。

#### 注意

\$ZTIMEZONE は、固定オフセットによりローカル時刻を調整します。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せん。InterSystems IRIS では、基本となるオペレーティング・システムからローカル時刻を取得します。オペレーティング・システムは、そのコンピュータで各ロケールに構成されているローカル時刻調整を適用します。したがって、\$ZTIMEZONE を使用して調整したローカル時刻には、\$ZTIMEZONE で指定したタイム・ゾーンではなく、構成したロケールのローカル時刻調整が適用されます。

UTC 時刻は、グリニッジ子午線 (\$ZTIMEZONE=0) からのタイム・ゾーンのカウントを使用して計算されます。これは、ローカル・グリニッジ時刻と同じではありません。グリニッジ標準時 (GMT) とは紛らわしい用語であり、グリニッジのローカル時刻は、冬の間は UTC と同じで、夏の間は、UTC と 1 時間違います。これは、英国夏時間と呼ばれるローカル時刻の変更が適用されるためです。

\$ZTIMEZONE を使用する関数およびプログラムでは、経過ローカル時間は必ず連続していますが、時刻値は季節によりローカル時刻に合わせて調整する必要があります。季節的なローカル時刻調整の詳細は、[\\$HOROLOG](#) を参照してください。

### タイム・ゾーンの設定

\$ZTIMEZONE を使用して、現在の InterSystems IRIS プロセスで使用されているタイム・ゾーンを設定することができます。\$ZTIMEZONE を設定しても、既定の InterSystems IRIS タイム・ゾーンや、使用しているコンピュータのタイム・ゾーン設定を変更することはありません。



注釈 \$ZTIMEZONE 特殊変数の変更は、いくつかの特殊な状況のために作られた機能です。\$ZTIMEZONE の変更は、InterSystems IRIS でローカルの日付/時刻の操作に使用するタイム・ゾーンの一貫した変更方法ではありません。\$ZTIMEZONE 特殊変数は、これを変更した結果として生じるすべての矛盾を処理できるプログラム以外では変更しないでください。

一部のプラットフォームでは、\$ZTIMEZONE 特殊変数の変更よりも適切な方法でタイム・ゾーンを変更できる場合があります。そのプラットフォームにプロセス固有のタイム・ゾーン設定が存在する場合 (POSIX システム上の TZ 環境変数など)、外部システム呼び出しを通じてプロセス固有のタイム・ゾーンを変更する方法は、\$ZTIMEZONE を変更する方法よりも適切である可能性があります。プロセス固有のタイム・ゾーンをオペレーティング・システム・レベルで変更すると、UTC からのローカル時刻オフセットが変更され、さらに、これに対応する、ローカル時刻調整の適用タイミングを決定するアルゴリズムも適用されます。このことが特に重要になるのは、既定のシステム・タイム・ゾーンが北半球のものである一方で、希望のプロセス・タイム・ゾーンが南半球のものである場合です。\$ZTIMEZONE を変更すると、ローカル時刻が UTC からの新しいタイム・ゾーン・オフセットに変更されますが、ローカル時刻調整の適用タイミングを決定するアルゴリズムは変更されません。

SET コマンドを使用して、\$ZTIMEZONE を指定された分の符号付き整数に設定します。最初の 0 と小数点は無視されます。\$ZTIMEZONE を設定する際に、数値以外の値を指定するか、または値を指定しないと、InterSystems IRIS は \$ZTIMEZONE を 0 (グリニッジ子午線) に設定します。

例えば、北米の東部標準時間 (EST) には、グリニッジに対して +5 時間の時差があります。したがって、現在の InterSystems IRIS プロセスを EST に設定するには、300 分と指定します。グリニッジの 1 時間前のタイム・ゾーンを指定するには、-60 分と指定します。グリニッジを指定するには、0 分と指定します。

\$ZTIMEZONE の設定：

- ・ 引数なしの \$NOW() ローカル時刻値に影響を及ぼします。これは、\$NOW() の時間部分を変更し、またこの変更が現在のプロセスの \$NOW() の日付部分を変更します。\$NOW() は、\$ZTIMEZONE 設定をそのまま反映し、ローカル時刻調整に合わせてその値を調整することはありません。
- ・ \$HOROLOGY ローカル時刻値に影響を及ぼします。\$HOROLOGY は \$ZTIMEZONE からタイム・ゾーン値を取得したうえで、サマータイムなどの季節的なローカル時刻調整を適用してその時刻を調整します。したがって \$HOROLOGY は常にローカル時刻に従いますが、\$HOROLOGY の経過時間は 1 年を通じて連続的ではありません。
- ・ %SYSTEM.Util クラス・メソッド IsDST()、UTCtoLocalWithZTIMEZONE()、および LocalWithZTIMEZONEtoUTC() に影響を及ぼします。
- ・ \$ZTIMESTAMP 値または \$ZHOROLOGY 値には影響しません。
- ・ \$ZDATE、\$ZDATEH、\$ZDATETIME、\$ZDATETIMEH、\$ZTIME、\$ZTIMEH の各関数によって実行される日付および時刻の形式変換には影響しません。
- ・ \$NOW(n) 関数には影響しません。
- ・ \$HOROLOGY の日付に固定値を設定する %SYSTEM.Process クラスの FixedDate() クラス・メソッドには影響しません。

\$ZTIMEZONE を変更した後に、以下の変則性が発生します。

- ・ \$ZDATETIME(\$HOROLOGY, 1, 7) は、通常 UTC 時刻を返しますが、\$ZTIMEZONE を変更すると UTC を返さなくなります。
- ・ \$ZTIMEZONE を変更すると、\$ZDATETIME(\$HOROLOGY, 1, 5) は、正しいタイム・ゾーン・オフセットを返さなくなります。
- ・ \$ZTIMEZONE を変更すると、ローカル時刻と UTC 時刻の間の \$ZDATETIME(\$HOROLOGY, -3) 変換および \$ZDATETIMEH(\$ZTIMESTAMP, -3) 変換が不正確になります。

## その他のタイム・ゾーン・メソッド

以下のように、TimeZone() クラス・メソッドを呼び出すことで、同じタイム・ゾーン情報を取得することができます。

### ObjectScript

```
WRITE $SYSTEM.SYS.TimeZone()
```

詳細は、“InterSystemsクラスリファレンス” の %SYSTEM.SYS クラスを参照してください。

次の例のように、tformat 値 5 または 6 を指定して \$ZDATETIME および \$ZDATETIMEH 関数を使用することで、日付および時刻の文字列の一部としてローカル時刻の差異を返すことができます。

### ObjectScript

```
WRITE !,$ZDATETIME($HOROLOG,1,5)
```

これは、次のような値を返します。

```
04/06/2011T12:31:16-04:00
```

この文字列の最後の部分 (-04:00) は、システムのローカル時刻調整設定を、グリニッジ子午線からのオフセット (時間と分で表現) として示したものです。このローカル時刻調整は、必ずしもタイム・ゾーン・オフセットではありません。上記の例では、タイム・ゾーンはグリニッジ標準時の 5 時間後 (-5:00) ですが、ローカル時刻調整 (サマータイム) によってこのタイム・ゾーンの時刻が 1 時間オフセットされて、-04:00 となります。\$ZTIMEZONE を設定すると、\$ZDATETIME(\$HOROLOG,1,5) によって返される現在のプロセスの日時が変更されますが、システムのローカル時刻調整設定は変更されません。

## \$ZDATETIMEH でのタイム・ゾーン設定の使用

dformat=-3 を指定して \$ZDATETIMEH を使用することで、協定世界時 (UTC) の日付と時刻の値をローカル時刻値に変換できます。この関数は、UTC 値 (\$ZTIMESTAMP) を入力として受け付けます。そして、ローカル・タイム・ゾーン設定を使用して、対応する日付と時刻を返します。サマータイムなどのローカル時刻調整の対象となっている場所であれば、その調整を適用した日付と時刻を返します。

### ObjectScript

```
SET clock=$HOROLOG
SET stamp=$ZDATETIMEH($ZTIMESTAMP,-3)
WRITE !,"local/local date and time: ", $ZDATETIME(clock,1,1,2)
WRITE !,"UTC/local date and time:   ", $ZDATETIME(stamp,1,1,2)
```

## \$ZTIMEZONE を使用したローカルと UTC の変換メソッド

%SYSTEM.Util クラスの 2 つのクラス・メソッドは、ローカルの日付および時刻と UTC の日付および時刻の間の変換を行います。つまり、UTCtoLocalWithZTIMEZONE() と LocalWithZTIMEZONEtoUTC() の変換です。これらのメソッドは、\$ZTIMEZONE の変更に影響されます。

### ObjectScript

```
WRITE $SYSTEM.Util.UTCtoLocalWithZTIMEZONE($ZTIMESTAMP),!
WRITE $HOROLOG,!
WRITE $SYSTEM.Util.LocalWithZTIMEZONEtoUTC($H),!
WRITE $ZTIMESTAMP
```

## 例

次の例は、現在のタイム・ゾーンを返します。

## ObjectScript

```
SET zone=$ZTIMEZONE
IF zone=0 {
    WRITE !,"Your time zone is Greenwich Mean Time" }
ELSEIF zone>0 {
    WRITE !,"Your time zone is ",zone/60," hours west of Greenwich" }
ELSE {
    WRITE !,"Your time zone is ",(-zone)/60," hours east of Greenwich" }
```

次の例は、タイム・ゾーンを設定することで日付および時刻を変更できることを示しています。

## ObjectScript

```
SET zonesave=$ZTIMEZONE
WRITE !,"Date in my current time zone: ",$ZDATE($HOROLOG)
IF $ZTIMEZONE=0 {
    SET $ZTIMEZONE=720 }
ELSEIF $ZTIMEZONE>0 {
    SET $ZTIMEZONE=($ZTIMEZONE-720) }
ELSE {
    SET $ZTIMEZONE=($ZTIMEZONE+720) }
WRITE !,"Date halfway around the world: ",$ZDATE($HOROLOG)
WRITE !,"Date at Greenwich Observatory: ",$ZDATE($ZTIMESTAMP)
SET $ZTIMEZONE=zonesave
```

次の例は、ローカル時刻がタイム・ゾーンの時刻と同じかどうかを判断します。

## ObjectScript

```
SET localnow=$HOROLOG, stamp=$ZTIMESTAMP
WRITE !,"local date and time: ",$ZDATETIME(localnow,1,1)
SET clocksecs=$PIECE(localnow,"",2)
SET stampsecs=$EXTRACT(stamp,7,11)-($ZTIMEZONE*60)
IF clocksecs=stampsecs {
    WRITE !,"No local time variant:"
    WRITE !,"Local time is timezone time" }
ELSE {
    IF clocksecs=stampsecs+3600 {
        WRITE !,"Daylight Saving Time variant:"
        WRITE !,"Local time offset 1 hour from timezone time" }
    ELSE { WRITE !,"Local time and time zone time are "
        WRITE !,(clocksecs-stampsecs)/60," minutes different" }
    }
QUIT
```

## 関連項目

- ・ [\\$HOROLOG](#) 特殊変数
- ・ [\\$ZTIMESTAMP](#) 特殊変数

## \$ZTRAP (ObjectScript)

現在のエラー・トラップ・ハンドラの場所が含まれます。

### 構文

```
$ZTRAP
$ZT
```

### 説明

\$ZTRAP には、現在のエラー・トラップ・ハンドラの場所が含まれます。エラー・トラップ・ハンドラをコード化する方法の詳細は、“[\\$ZTRAP でのエラー処理](#)”を参照してください。

- ・ ルーチンでは、\$ZTRAP はそのルーチンまたは他のルーチン内のラベルを参照できます。これには、[ラベル](#)名とルーチン名が含まれます。例えば、ルーチン MyRou 内のラベル MyHandler に \$ZTRAP を設定した場合、\$ZTRAP 特殊変数には MyHandler^MyRou が含まれます。
- ・ プロシージャでは、\$ZTRAP はそのプロシージャ内のラベルを参照する必要があります。これには、プロシージャ名からのラベルの行オフセットが含まれます。例えば、プロシージャ MyProc 内のプライベート・ラベル MyHandler に \$ZTRAP を設定した場合、\$ZTRAP 特殊変数には +17^MyProc が含まれます。

\$ZTRAP を設定するには、次の 3 つの方法があります。

- ・ `SET $ZTRAP="location"`
- ・ `SET $ZTRAP="*location"`
- ・ `SET $ZTRAP="~%ETN"`

KILL \$ZTRAP を実行することはできません。これを実行しようすると、〈SYNTAX〉エラーになります。現在の \$ZTRAP 値を削除するには、\$ZTRAP="" を指定します。

### Location

SET コマンドを使用することで、引用符付きの文字列として location を指定できます。

- ・ [ルーチン](#)では、location は label (現在のルーチン内の[行ラベル](#))、^routine (指定された外部ルーチンの開始)、または label^routine (指定された外部ルーチン内の指定されたラベル) として指定できます。プロシージャ内のラベルやプロシージャを参照する location をルーチンで指定しないでください。これは無効な location であり、InterSystems IRIS が \$ZTRAP を実行しようとする際に実行時エラーになります。
- ・ [プロシージャ](#)では、label (そのプロシージャ・ブロック内の[プライベート・ラベル](#)) として location を指定できます。プロシージャ・ブロック内の \$ZTRAP は、プロシージャ本体の外側にある location への移動には使用できません。プロシージャ・ブロック内の \$ZTRAP は、プロシージャ・ブロック内の location しか参照できません。したがって、プロシージャでは、\$ZTRAP を ^routine や label^routine に設定することはできません。設定しようすると、〈SYNTAX〉エラーになります。

プロシージャでは \$ZTRAP をプライベート・ラベル名に設定しますが、\$ZTRAP 値はプライベート・ラベル名ではなく、プロシージャ・ラベル (プロシージャの先頭) からプライベート・ラベルの行の場所までのオフセットです。(例) +17^myproc。

注釈 \$ZTRAP は、一部のコンテキスト(プロシージャ以外)で label+offset に対する従来のサポートを提供します。オプションのこの +offset は、label からオフセットする行数を指定する整数です。label は、同じルーチンに含める必要があります。+offset の使用は非推奨であり、コンパイル警告エラーが発生する場合があります。インターシステムズでは、location の指定時には行のオフセットを使用しないことを推奨します。

プロシージャまたは IRISYS % ルーチンの呼び出し時における +offset の指定はできません。指定しようとする、InterSystems IRIS は <NOLINE> エラーを発行します。

\$ZTRAP の location は、現在のネームスペースにある必要があります。\$ZTRAP では、[拡張ルーチン参照](#)をサポートしません。

存在しない行ラベル (現在のルーチンに存在しない location) を指定した場合、以下が発生します。

- ・ \$ZTRAP の表示：ルーチンでは、\$ZTRAP には label^routine が含まれます。(例) DummyLabel^MyRou。プロシージャでは、\$ZTRAP にはオフセットの最大許容値が含まれます。(例) +34463^MyProc。
- ・ \$ZTRAP の呼び出し：InterSystems IRIS が <NOLINE> エラー・メッセージを返します。

各スタック・レベルは専用の \$ZTRAP 値を備えることができます。\$ZTRAP を設定すると、システムでは前回のスタック・レベルの \$ZTRAP 値が保存されます。InterSystems IRIS では、現在のスタック・レベルが終了すると、この値がリストアされます。現在のスタック・レベルでエラー・トラップをオンにするには、エラー・トラップ・ハンドラの location を指定して、\$ZTRAP にエラー・トラップ・ハンドラを設定します。次に例を示します。

### ObjectScript

```
IF $ZTRAP="" {WRITE !,"$ZTRAP not set" }
ELSE {WRITE !,"$ZTRAP already set: ", $ZTRAP
      SET oldtrap=$ZTRAP }
SET $ZTRAP="Etrap1^Handler"
WRITE !,"$ZTRAP set to: ", $ZTRAP
// program code
SET $ZTRAP=oldtrap
WRITE !,"$ZTRAP restored to: ", $ZTRAP
```

エラーが発生すると、この形式ではコール・スタックが戻され、特定のエラー・トラップ・ハンドラに制御が移されます。

[SqlComputeCode](#) では、\$ZTRAP=\$ZTRAP を設定しないでください。これは、トランザクション処理およびエラー報告に関する重大な問題を引き起こす可能性があります。

エラー・トラップをオフにするには、\$ZTRAP を NULL 文字列( "") に設定します。これを行うと、現在 DO が実行されているスタック・レベルに設定されているエラー・トラップをクリアします。

\$ZTRAP を使用してエラー・ハンドラを設定する場合、このハンドラは既存のどの \$ETRAP エラー・ハンドラよりも優先されます。InterSystems IRIS は暗黙的に NEW \$ETRAP コマンドを実行し、\$ETRAP を NULL 文字列( "") に設定します。

注釈 ターミナル・プロンプトからの \$ZTRAP の使用は、現在のコード行に制限されます。SET \$ZTRAP コマンドと、エラーを生成するコマンドは、コードの同じ行に含まれている必要があります。ターミナルは、各コマンド行の先頭で \$ZTRAP をシステムの既定値にリストアします。

### \*Location

ルーチンでは、エラーが発生した後、コール・スタックをそのままにしておくこともできます。これを行うには、location の前および二重引用符内にアスタリスク(\*)を挿入します。プロシージャ内でこの形式を使用することは有効ではありません。これを使用しようとすると、<SYNTAX> エラーになります。この例に示すように、プロシージャではないサブルーチンでのみ使用できます。

## ObjectScript

```

Main
    SET $ZTRAP="*OnError"
    WRITE !,"$ZTRAP set to: ",$ZTRAP
    // program code
OnError
    // Error handling code
    QUIT

```

この形式では、\$ZTRAP で指定した[行ラベル](#)への GOTO が実行されるだけです。\$STACK と \$ESTACK は変更されません。\$ZTRAP エラー処理ルーチンのコンテキスト・フレームは、エラーが発生したコンテキスト・フレームと同じです。ただし、InterSystems IRIS は [\\$ROLES](#) を、\$ZTRAP が設定されていた実行レベルに有効であった値にリセットします。これにより \$ZTRAP エラー・ハンドラは、エラー・ハンドラの設定後にルーチンに付与された上位特権を使用しないようにします。\$ZTRAP エラー処理ルーチンが完了すると、スタックが前のコンテキスト・レベルに戻されます。この形式の \$ZTRAP は特に、予期しないエラーの分析に便利です。

アスタリスクによって \$ZTRAP オプションが設定されます。これは、location の一部ではありません。そのため、\$ZTRAP で WRITE または ZZDUMP を実行しても、このアスタリスクは表示されません。

### ^%ETN

ルーチンでは、SET \$ZTRAP="^%ETN" は、現在のエラー・トラップ・ハンドラとして、システムで提供されるエラー・ルーチン %ETN を設定します。%ETN は、これと呼び出したエラーの発生元であるコンテキストで実行されます。( %ET は %ETN の従来の名前です。これらの機能は同じですが、%ETN のほうが多少効率性に優れています)。エラー・ハンドラ ^%ETN は常に、\* ([アスタリスク](#)) が先行しているかのように動作します。

プロシージャ・ブロック内の \$ZTRAP は、プロシージャ本体の外側にある location への移動には使用できないため、プロシージャで SET \$ZTRAP="^%ETN" を使用することはできません。使用しようとすると、<SYNTAX> エラーが返されます。

%ETN とそのエントリポイント FORE^%ETN、BACK^%ETN、および LOG^%ETN の詳細は、"[\(従来\) ^%ETN を使用したエラー・ログ](#)" を参照してください。

## TRY / CATCH と \$ZTRAP

TRY ブロック内で \$ZTRAP を設定することはできません。これを試みると、コンパイル・エラーが生成されます。\$ZTRAP は、TRY ブロックの前、または CATCH ブロック内で設定できます。

TRY ブロック内で発生したエラーは、\$ZTRAP が以前に設定されているかどうかに関係なく、CATCH ブロックで処理されます。CATCH ブロック内で発生するエラーは、現在のエラー・トラップ・ハンドラによって処理されます。

以下の最初の例では、TRY ブロックで発生したエラーを示します。以下の 2 番目の例では、TRY ブロックにスローされた例外を示します。以下の両方の場合において、\$ZTRAP ではなく、CATCH ブロックが用いられています。

## ObjectScript

```

    SET $ZTRAP="Ztrap"
    TRY { WRITE 1/0 } /* divide-by-zero error */
    CATCH { WRITE "Catch taken" }
    QUIT
Ztrap
    WRITE "$ZTRAP taken"
    SET $ZTRAP=""
    QUIT

```

## ObjectScript

```

SET $ZTRAP="Ztrap"
TRY { SET myvar=##class(Sample.MyException).%New("Example Error",999,,errdatazero)
      WRITE !,"Throwing an exception!",!
      THROW myvar
    }
CATCH { WRITE "Catch taken" }
QUIT
Ztrap
WRITE "$ZTRAP taken"
SET $ZTRAP=""
QUIT

```

ただし、TRY ブロックは \$ZTRAP を設定および使用するコードを呼び出すことができます。以下の例では、0 による除算エラーが CATCH ブロックではなく、\$ZTRAP によりキャッチされています。

## ObjectScript

```

TRY { DO Errsub }
CATCH { WRITE "Catch taken" }
QUIT
Errsub
SET $ZTRAP="Ztrap"
WRITE 1/0 /* divide-by-zero error */
QUIT
Ztrap
WRITE "$ZTRAP taken"
SET $ZTRAP=""
QUIT

```

CATCH ブロックからの THROW コマンドでも、\$ZTRAP エラー・ハンドラを呼び出すことができます。

## 例

次の例では、\$ZTRAP がこのプログラムの OnError ルーチンに設定されます。これによって、(数値を 0 で除算しようとすると) エラーが発生する SubA が呼び出されます。エラーが発生すると、\$ZTRAP で指定した OnError ルーチンが呼び出されます。OnError は、\$ZTRAP が設定されたコンテキスト・レベルで呼び出されます。OnError は Main と同じコンテキスト・レベルにあるため、実行は Main に戻りません。

## ObjectScript

```

Main
NEW $ESTACK
SET $ZTRAP="OnError"
WRITE !,"$ZTRAP set to: ",$ZTRAP
WRITE !,"Main $ESTACK= ",$ESTACK // 0
WRITE !,"Main $ECODE= ",$ECODE
DO SubA
WRITE !,"Returned from SubA" // not executed
WRITE !,"MainReturn $ECODE= ",$ECODE
QUIT
SubA
WRITE !,"SubA $ESTACK= ",$ESTACK // 1
WRITE !,6/0 // Error: division by zero
WRITE !,"fine with me"
QUIT
OnError
WRITE !,"OnError $ESTACK= ",$ESTACK // 0
WRITE !,"$ECODE= ",$ECODE
QUIT

```

次の例は、1 つの例外を除き、前の例と同じです。つまり、\$ZTRAP location がアスタリスク (\*) で始まっています。SubA でエラーが発生すると、このアスタリスクによって、(\$ZTRAP が設定された) Main のコンテキスト・レベルではなく、(エラーが発生した) SubA のコンテキスト・レベルで OnError ルーチンが呼び出されます。そのため、OnError が完了すると、DO コマンドに続く行で実行が Main に戻ります。

## ObjectScript

```
Main
NEW $ESTACK
SET $ZTRAP="*OnError"
WRITE !,"$ZTRAP set to: ",$ZTRAP
WRITE !,"Main $ESTACK= ",$ESTACK    // 0
WRITE !,"Main $ECODE= ",$ECODE
DO SubA
WRITE !,"Returned from SubA"    // executed
WRITE !,"MainReturn $ECODE= ",$ECODE
QUIT
SubA
WRITE !,"SubA $ESTACK= ",$ESTACK    // 1
WRITE !,"6/0    // Error: division by zero
WRITE !,"fine with me"
QUIT
OnError
WRITE !,"OnError $ESTACK= ",$ESTACK    // 1
WRITE !,"$ECODE= ",$ECODE
QUIT
```

## 関連項目

- ・ [THROW コマンド](#)
- ・ [ZINSERT コマンド](#)
- ・ [\\$ECODE 特殊変数](#)
- ・ [\\$ESTACK 特殊変数](#)
- ・ [\\$ETRAP 特殊変数](#)
- ・ [\\$STACK 特殊変数](#)
- ・ [TRY-CATCH の使用法](#)



# \$ZVERSION (ObjectScript)

InterSystems IRIS の現在のバージョンを表す文字列を含みます。

## 構文

```
$ZVERSION  
$ZV
```

## 概要

\$ZVERSION には、現在動作している InterSystems IRIS® Data Platform のインスタンスのバージョンを表す文字列が含まれます。

以下の例は、\$ZVERSION 文字列を返します。

### ObjectScript

```
WRITE $ZVERSION
```

これは、以下のようなバージョンの文字列を返します。

```
IRIS for Windows (x86-64) 2018.1 (Build 487U) Tue Dec 26 2017 22:47:10 EST
```

この文字列には、InterSystems IRIS インストールのタイプ (CPU の型を含む製品番号とプラットフォーム)、バージョン番号 (2018.1)、現在のバージョンでのビルド番号 (ビルド番号の "U" は Unicode を意味します)、および InterSystems IRIS の現在のバージョンが作成された日付と時刻が含まれます。"EST" は東部標準時 (米国東部のタイム・ゾーン)、"EDT" は東部サマータイムを表しています (詳細は "[サマータイム](#)" を参照してください)。

次のように、GetVersion() クラス・メソッドを呼び出すことで、同じ情報を取得できます。

### ObjectScript

```
WRITE %SYSTEM.Version.GetVersion()
```

他の %SYSTEM.Version メソッドを呼び出してこのバージョン文字列のコンポーネントを取得し、以下を呼び出してリストにできます。

### ObjectScript

```
DO %SYSTEM.Version.Help()
```

バージョンおよびビルド番号の情報は、InterSystems IRIS ランチャーに移動し、[\[バージョン情報\]](#) を選択することで表示できます。

\$ZVERSION 特殊変数は、SET コマンドを使用して変更することはできません。変更を試みると、<SYNTAX> エラーが返されます。

## 例

以下の例は、バージョン文字列から生成された日付を取得し、InterSystems IRIS の現行バージョンの経過日数を算出します。この例は Windows プラットフォーム固有のものです。

## ObjectScript

```
SET createdate=$PIECE($ZVERSION," ",9,11)
WRITE !,"Creation date: ",createdate
WRITE !,"Current date:  ",$ZDATE($HOROLOG,6)
SET nowcount=$PIECE($HOROLOG,"")
SET thencount=$ZDATEH(createdate,6)
WRITE !,"This version is ",(nowcount-thencount)," days old"
```

以下の例では、クラス・メソッドを呼び出して同じ操作を実行します。

## ObjectScript

```
SET createdate=$SYSTEM.Version.GetBuildDate()
WRITE !,"Creation date: ",$ZDATE(createdate,6)
WRITE !,"Current date:  ",$ZDATE($HOROLOG,6)
SET nowcount=$PIECE($HOROLOG,"")
WRITE !,"This version is ",(nowcount-createdate)," days old"
```

## 関連項目

- ・ [\\$ZVERSION\(1\)](#) 関数
- ・ [\\$SYSTEM](#) 特殊変数

# 構造化システム変数

構造化システム変数名 (SSVN) は、グローバル変数のように組織化されている非スカラ・システム変数です。SSVN で、システム・データに関する情報を検索できる、移植可能なプログラムを記述することができます。構造化システム変数を使用することで、同じ ObjectScript コードで任意の InterSystems IRIS® データ・プラットフォーム実装からシステム・データ情報を取得できます。

各 SSVN は、以下のいずれかの添え字の値を含む構造を持ちます。

- ・ エンティティ識別子
- ・ リテラル
- ・ 属性キーワード

識別子である添え字と属性ノードに値を与えると、エンティティに関する情報が提供されます。

SSVN は以下の標準接頭語として、キャレット (^) とドル記号 (\$) を使用します。

1. 必要な情報に関するネームスペースのオプション (拡張構文) の指定
2. 指定された名前リストのうちの 1 つ

それから、括弧の中に 1 つ、あるいは複数の式の名前が続きます。これらの式は、添え字と呼ばれています。構文は、以下のとおりです。

```
^[ |namespace| ] ssvn_name ( expression )
```

通常、SET コマンドと KILL コマンドはデータ値を持つとは限らないので、構造化システム変数に使用できません。構造化システム変数が提供する情報のうちの大半は、特定の添え字値の存在です。多くの場合、\$DATA 関数、\$ORDER 関数、および \$QUERY 関数を使用して、添え字値を検査します。

InterSystems IRIS® データ・プラットフォームは、以下の構造化システム変数をサポートしています。

- ・ ^\$GLOBAL
- ・ ^\$JOB
- ・ ^\$LOCK
- ・ ^\$ROUTINE

各 SSVN の意味と添え字の使用方法は、以下のセクションで説明します。

各説明では、どの関数が特定の構造化システム変数で利用できるかを示します。各説明には、システム・テーブル情報を調べるために、\$DATA 関数、\$ORDER 関数、および \$QUERY 関数への引数としての構造化システム変数の使用法について、1 つまたは複数の例が含まれています。

## ^\$GLOBAL (ObjectScript)

グローバルとプロセス・プライベート・グローバルの情報を提供します。

### 構文

```
^$ | nspace | GLOBAL(global_name)
^$ | nspace | G(global_name)

^$ | | GLOBAL(global_name)
^$ | | G(global_name)
```

### 引数

引数	説明
nspace   または [ nspace ]	オプション — <a href="#">拡張 SSVN 参照</a> 。明示的なネームスペース名か、 <a href="#">暗黙のネームスペース</a> のいずれかです。角括弧 ([ "namespace" ])、または垂直バー (  "namespace"  ) で囲まれた、引用符付きの文字列に評価されなければなりません。ネームスペース名は、大文字と小文字を区別しません。ネームスペース名はすべて大文字で格納および表示されます。
global_name	添え字なしのグローバル名を含む文字列に評価される式。グローバル名は、大文字と小文字を区別します。^\$  GLOBAL() 構文を使用する場合は、プロセス・プライベート・グローバルに対応する添え字なしのグローバル名です。^ a の場合は ^a になります。

### 概要

^\$GLOBAL を \$DATA 関数、\$ORDER 関数、および \$QUERY 関数に対する引数として使用し、現在のネームスペース (既定)、あるいは指定したネームスペースにグローバル変数が存在するか否かに関する情報を返すことができます。また、^\$GLOBAL を使用して、プロセス・プライベート・グローバル変数が存在するか否かに関する情報を返すこともできます。

グローバル変数の詳細情報は、“[InterSystems IRIS 多次元ストレージの使用法](#)”を参照してください。

### プロセス・プライベート・グローバル

^\$GLOBAL を使用して、すべてのネームスペースで[プロセス・プライベート・グローバル](#)変数が存在するか否かに関する情報を返すことができます。プロセス・プライベート・グローバルの検索は、^\$||GLOBAL または ^\$|""|GLOBAL のいずれかとして指定できます。

例えば、プロセス・プライベート・グローバル ^| | a とその子孫に関する情報を取得するには、\$DATA(^\$| | GLOBAL(" ^a" )) を指定します。プロセス・プライベート・グローバルはネームスペース固有ではないため、この検索では、プロセス・プライベート・グローバルが定義されていれば現在のネームスペースに関係なく、^| | a に関する情報が返されます。

^\$GLOBAL では、global\_name そのものにプロセス・プライベート・グローバル構文を指定することはできないことに注意してください。プロセス・プライベート・グローバル構文で global\_name を指定しようとすると、<NAME> エラーが返されます。

### 引数

#### namespace

このオプションの引数を使用すると、^\$GLOBAL により、別のネームスペースで定義された global\_name を検索できます。これを、[拡張 SSVN 参照](#)といいます。ネームスペースは、引用符付きの文字列リテラルまたは変数として、明示的にネームスペース名を指定するか、もしくは[暗黙のネームスペース](#)を指定します。ネームスペース名は、大文字と小文字を

区別しません。括弧付き構文 ["USER"] か、環境構文 |"USER| のいずれかを使用します。nspace 区切り文字の前後にスペースを使用することはできません。

以下のメソッドを使用して、ネームスペースが定義されているかどうかをテストすることができます。

### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

\$NAMESPACE 特殊変数を使用して、現在のネームスペースを決定できます。現在のネームスペースを変更するお勧めの方法は、NEW \$NAMESPACE に続けて SET \$NAMESPACE="nspacename" を使用することです。

### global\_name

添え字なしのグローバル名を含む文字列に評価される式。グローバルでは大文字と小文字が区別されます。

- ・ ^\$GLOBAL("^a"):global\_name "^a" は、現在のネームスペースでこの**グローバル**とその子孫を検索します。プロセス・プライベート・グローバル "^|a" は検索しません。
- ・ ^\$|"USER"|GLOBAL("^a"):global\_name "^a" は、"USER" ネームスペースでこの**グローバル**とその子孫を検索します。プロセス・プライベート・グローバル "^|a" は検索しません。
- ・ ^\$||GLOBAL("^a"):global\_name "^a" は、すべてのネームスペースで**プロセス・プライベート・グローバル** "^|a" とその子孫を検索します。グローバル "^a" は検索しません。

## 例

以下の例は \$DATA、\$ORDER、\$QUERY 関数に対して、^\$GLOBAL を引数として使用方法を表しています。

### \$DATA に対する引数としての ^\$GLOBAL

^\$GLOBAL は \$DATA に対する引数として、指定するグローバル名が ^\$GLOBAL ノードとし存在するか否かを示す整数値を返します。以下のテーブルは、\$DATA が返すことができる整数値を示しています。

値	意味
0	グローバル名が存在しない
1	グローバル名は下位ノードを持たない、データを持つ既存のノード
10	グローバル名は下位ノードを持つ、データなしの既存のノード
11	グローバル名は下位ノードを持つ、データを持つ既存のノード

以下の例は、現在のネームスペースに指定したグローバル変数が存在するかどうかをテストしています。

### ObjectScript

```
KILL ^GBL
WRITE $DATA(^$GLOBAL(^GBL)),!
SET ^GBL="test"
WRITE $DATA(^$GLOBAL(^GBL)),!
SET ^GBL(1,1,1)="subscripts test"
WRITE $DATA(^$GLOBAL(^GBL))
```

これは 0、1、11 を順番に返します。

以下の例は、USER ネームスペースに指定したグローバル変数が存在するかどうかをテストしています。

## ObjectScript

```
SET $NAMESPACE="USER"
SET ^GBL(1)="test"
SET $NAMESPACE="%SYS"
WRITE $DATA(^$|"USER"|GLOBAL(^GBL))
```

これは、10 を返します。

以下の例は、任意のネームスペースに指定したプロセス・プライベート・グローバル変数が存在するかどうかをテストしています。

## ObjectScript

```
SET $NAMESPACE="USER"
SET ^||PPG(1)="test"
SET $NAMESPACE="%SYS"
WRITE $DATA(^$||GLOBAL(^PPG))
```

これは、10 を返します。

**\$ORDER に対する引数としての ^\$GLOBAL**

```
$ORDER(^$|namespace|GLOBAL(global_name),direction)
```

^\$GLOBAL は \$ORDER に対する引数として、指定したグローバル名に対する照合順序で、次または前のグローバル名を返します。そのようなグローバル名ノードが ^\$GLOBAL に存在しない場合は、\$ORDER は NULL 文字列を返します。

注釈 \$ORDER(^\$GLOBAL(name)) は、[IRISSYS データベース](#)から % グローバル名を返しません。

direction 引数は、次のグローバル名を返すか、前のグローバル名を返すかを指定します。direction 引数を指定しない場合、InterSystems IRIS は、指定したグローバル名に対する照合順序で、次のグローバル名を返します。詳細は、“[\\$ORDER](#)” 関数を参照してください。

以下のサブルーチンは、現在のネームスペースを検索し、GLOBAL というローカル配列にグローバル名を格納します。

## ObjectScript

```
GLOB
SET NAME=""
WRITE !,"The following globals are in ", $NAMESPACE
FOR I=1:1 {
    SET NAME=$ORDER(^$GLOBAL(NAME))
    WRITE !,NAME
    QUIT:NAME=""
    SET GLOBAL(I)=NAME
}
WRITE !,"All done"
QUIT
```

**\$QUERY に対する引数としての ^\$GLOBAL**

^\$GLOBAL は \$QUERY に対する引数として、指定したグローバル名に対する照合順序で、次のグローバル名を返します。そのようなグローバル名が ^\$GLOBAL のノードとして存在しなければ、\$QUERY は NULL 文字列を返します。

注釈 \$QUERY(^\$GLOBAL(name)) は、[IRISSYS データベース](#)から % グローバル名を返しません。

以下の例では、3 つのグローバル (^GBL1、^GBL2、^GBL3) が、USER ネームスペースに存在します。

## ObjectScript

```

NEW $NAMESPACE
SET $NAMESPACE="USER"
SET (^GBL1, ^GBL2, ^GBL3)="TEST"
NEW $NAMESPACE
SET $NAMESPACE="%SYS"
WRITE $QUERY(^$| "USER" | GLOBAL( "^GBL1" ) ), !
WRITE $QUERY(^$| "USER" | GLOBAL( "^GBL2" ) )
NEW $NAMESPACE
SET $NAMESPACE="USER"
KILL ^GBL1, ^GBL2, ^GBL3

```

最初の WRITE は ^\$| "USER" | GLOBAL( "^GBL2" ) を返します。

2 番目の WRITE は ^\$| "USER" | GLOBAL( "^GBL3" ) を返します。

## MERGE に対する引数としての ^\$GLOBAL

MERGE コマンドの source 引数として ^\$GLOBAL を使用すると、グローバル・ディレクトリが destination 変数にコピーされます。MERGE は、各グローバル名を、NULL 値を持つ destination 添え字として追加します。以下に例を示します。

## ObjectScript

```

MERGE gbls=^$GLOBAL( " " )
ZWRITE gbls

```

## 関連項目

- ・ [\\$DATA](#) 関数
- ・ [\\$ORDER](#) 関数
- ・ [\\$QUERY](#) 関数
- ・ [ZNSPACE](#) コマンド
- ・ [\\$NAMESPACE](#) 特殊変数
- ・ [ネームスペースの構成](#)

## ^\$JOB (ObjectScript)

InterSystems IRIS プロセスの (ジョブ) 情報を提供します。

### 構文

```
^$JOB(job_number)
^$J(job_number)
```

### 引数

引数	説明
<i>job_number</i>	ObjectScript コマンドを入力するときに作成される、システム特有のジョブ番号。アクティブな InterSystems IRIS プロセスは一意のジョブ番号を持ちます。システムにログインすると、ジョブは初期化されます。UNIX® システムでは、ジョブ番号は InterSystems IRIS が呼び出されたときに開始した子プロセスの PID です。 <i>job_number</i> は整数で指定する必要があります。16 進数値はサポートされていません。

### 概要

ローカル InterSystems IRIS システムで InterSystems IRIS ジョブの存在に関する情報を取得するために、\$DATA、\$ORDER、および \$QUERY 関数に対する引数として ^\$JOB 構造化システム変数を使用することができます。

### 例

以下の例は、\$DATA、\$ORDER、および \$QUERY 関数に対する引数として ^\$JOB を使用する方法を示しています。

#### \$DATA に対する引数としての ^\$JOB

```
$DATA(^$JOB(job_number))
```

\$DATA に対する引数としての ^\$JOB は、指定したジョブが ^\$JOB にノードとして存在するかどうかを示す整数値を返します。次のテーブルは、\$DATA が返すことができる整数値を示しています。

値	意味
0	ジョブは存在しない
1	ジョブが存在する

以下の例は InterSystems IRIS プロセスの存在を調べています。

#### ObjectScript

```
SET x=$JOB
WRITE !,$DATA(^$JOB(x))
```

変数 *x* は、現在のプロセスのジョブ番号に設定されます (例えば 4294219937)。WRITE はブーリアン 1 を返し、このプロセスが存在することを示します。

#### \$ORDER に対する引数としての ^\$JOB

```
$ORDER(^$JOB(job_number),direction)
```

\$ORDER に対する引数としての ^\$JOB は、指定したジョブ番号に対する照合順序で、次または前の ^\$JOB ジョブ番号を返します。そのようなジョブ番号が ^\$JOB ノードとして存在しない場合は、\$ORDER は NULL 文字列を返します。



direction 引数は、次のジョブ番号を返すか、前のジョブ番号を返すかを指定します。direction 引数を指定しない場合、InterSystems IRIS は、指定したジョブ番号に対する照合順序で、次のジョブ番号を返します。詳細は、“\$ORDER” 関数を参照してください。

以下のサブルーチンは InterSystems IRIS のジョブ・テーブルを検索し、JOB というローカル配列にあるジョブ番号を格納します。

#### ObjectScript

```
JOB
SET pid=""
FOR i=1:1 {
    SET pid=$ORDER(^$JOB(pid))
    QUIT:pid=""
    SET JOB(i)=pid
}
WRITE "Total Jobs in Job Table: ",i
QUIT
```

### \$QUERY に対する引数としての ^\$JOB

\$QUERY(^\$JOB(job\_number))

\$QUERY に対する引数としての ^\$JOB は、指定したジョブ番号に対する照合順序で、次の ^\$JOB ジョブ番号を返します。そのようなジョブ番号が ^\$JOB にノードとして存在しない場合は、\$QUERY は NULL 文字列を返します。

以下の例は、InterSystems IRIS ジョブ・テーブルで最初 2 つのジョブを返します。間接指定演算子 (@) を使用します。

#### ObjectScript

```
SET x=$QUERY(^$JOB(" "))
WRITE !,x
WRITE !,$QUERY(@x)
```

これは、以下のようなジョブの値を返します。

```
^$JOB("4294117993")
```

```
^$JOB("4294434881")
```

### 関連項目

- ・ [JOB コマンド](#)
- ・ [\\$DATA](#) 関数
- ・ [\\$ORDER](#) 関数
- ・ [\\$QUERY](#) 関数
- ・ [\\$JOB](#) 特殊変数
- ・ [\\$ZJOB](#) 特殊変数
- ・ [\\$ZCHILD](#) 特殊変数
- ・ [\\$ZPARENT](#) 特殊変数

## ^\$LOCK (ObjectScript)

ロック名情報を提供します。

### 構文

```
^$ | nspace | LOCK ( lock_name , info_type , pid )
^$ | nspace | L ( lock_name , info_type , pid )
```

### 引数

引数	説明
nspace   または [ nspace ]	オプション – 拡張 <a href="#">SSVN 参照</a> 。明示的なネームスペース名か、 <b>暗黙のネームスペース</b> のいずれかです。角括弧 ([ "namespace" ])、または垂直バー (  "namespace"  ) で囲まれた、引用符付きの文字列に評価されなければなりません。ネームスペース名は、大文字と小文字を区別しません。ネームスペース名はすべて大文字で格納および表示されます。
lock_name	文字列に評価される式。添え字付き、または添え字なしのロック変数名を含みます。リテラルの場合は引用符付きの文字列として指定する必要があります。
info_type	オプション – 引用符付きの文字列として指定される、すべて大文字の文字列のキーワードに解決される式。info_type は、lock_name について返される情報のタイプを指定します。可能なオプションは、“OWNER”、“FLAGS”、“MODE”、および “COUNTS” です。^\$LOCK をスタンドアロン関数として呼び出す場合は必須です。
pid	オプション – “COUNTS” キーワードと共に使用します。ロックの所有者のプロセス ID を指定する整数です。指定した場合は、最大で 1 つのリスト要素が “COUNTS” に返されます。省略した場合または 0 を指定した場合は、指定したロックの所有者ごとにリスト要素が 1 つ返されます。pid はその他の info_type キーワードには影響しません。

### 概要

^\$LOCK 構造化システム変数は、現在のネームスペースにあるロックまたはローカル・システム上の指定したネームスペースにあるロックに関する情報を返します。^\$LOCK の使用方法には以下の 2 種類があります。

- ・ info\_type を指定し、指定のロックに関する情報を返すスタンドアロン関数として使用する。
- ・ info\_type を指定せずに、\$DATA、\$ORDER、または \$QUERY の各関数の引数として使用する。

注釈    ^\$LOCK は、ローカル・システムのロック・テーブルからロック・テーブル情報を取得します。リモート・サーバ上のロック・テーブルからの情報は返しません。

### ECP 環境での ^\$LOCK

- ・ ローカル・システム: ローカル・システムにより保持されているロックの ^\$LOCK を呼び出す場合、^\$LOCK の動作は ECP なしの場合と同じになります。ただし 1 つの例外として、“FLAGS” info\_type は、ロックが ECP 環境にあることを示すアスタリスク (\*) を返します。つまり、リモート InterSystems IRIS インスタンスは、解除されてもロックを保持できるということです。
- ・ アプリケーション・サーバ: アプリケーション・サーバ上で、別のサーバで保持されているロックに対して、ECP (データ・サーバまたは別のアプリケーション・サーバのいずれか) を介して ^\$LOCK を呼び出す場合、^\$LOCK は何も情報を返しません。これはロックが存在していない場合の動作と同じです。
- ・ データ・サーバ: データ・サーバ上で、アプリケーション・サーバによって保持されているロックに対して ^\$LOCK を呼び出す場合、^\$LOCK の動作は、以下のようにローカル・システム上の場合と多少異なります。

- “OWNER”: ^\$LOCK を呼び出すデータ・サーバに接続されたアプリケーション・サーバによってロックが保持されている場合、^\$LOCK(lockname, “OWNER”) は “ECPCConfigurationName:MachineName:InstanceName” を、ロックを保持しているインスタンスに返しますが、ロックを保持している具体的なプロセスは識別しません。
- “FLAGS”: ^\$LOCK を呼び出すデータ・サーバに接続されたアプリケーション・サーバによってロックが保持されている場合、排他ロックに対する ^\$LOCK(lockname, “FLAGS”) は “Z” フラグを返します。この “Z” は、InterSystems IRIS では ECP 環境以外で使われなくなった従来のロックのタイプを示しています。
- “MODE”: ^\$LOCK を呼び出すデータ・サーバに接続されたアプリケーション・サーバによってロックが保持されている場合、排他ロックに対する ^\$LOCK(lockname, “MODE”) は、“X” の代わりに “ZAX” を返します。ZA は、InterSystems IRIS では ECP 環境以外で使われなくなった従来のロックのタイプを示しています。共有ロックの場合、^\$LOCK(lockname, “MODE”) は、ローカル・ロックに対するのと同じ “S” を返します。

## 引数

### nspc

このオプションの引数では、[拡張 SSVN 参照](#)を使用することで他のネームスペースのグローバルを指定できます。ネームスペースは、引用符付きの文字列リテラルまたは変数として、明示的にネームスペース名を指定するか、もしくは[暗黙のネームスペース](#)を指定します。ネームスペース名は、大文字と小文字を区別しません。括弧付き構文 [“USER”] か、環境構文 |“USER”| のいずれかを使用します。nspc 区切り文字の前後にスペースを使用することはできません。

以下のメソッドを使用して、ネームスペースが定義されているかどうかをテストすることができます。

### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

\$NAMESPACE 特殊変数を使用して、現在のネームスペースを決定できます。現在のネームスペースを変更するお勧めの方法は、NEW \$NAMESPACE に続けて SET \$NAMESPACE="nspcename" を使用することです。

### lock\_name

文字列に評価される式。添え字付き、または添え字なしのロック変数名を含みます。ロック変数(一般にはグローバル変数)は、LOCK コマンドを使用して定義します。

### info\_type

info\_type キーワードは、^\$LOCK をスタンドアロン関数として呼び出す場合は必須で、^\$LOCK を別の関数の引数として使用する場合はオプションです。info\_type は、引用符付きの文字列として大文字で指定する必要があります。

- ・ “OWNER” を指定すると、ロックの所有者のプロセス ID (pid) が返されます。ロックが共有ロックである場合は、そのロックの所有者すべてのプロセス ID のコンマ区切りリストが返されます。指定したロックが存在しない場合、^\$LOCK は空の文字列を返します。
- ・ “FLAGS” を指定すると、ロックの状態が返されます。返される値は “D” (ロック解除を保留中)、“P” (ロックを保留中)、“N” (ノード・ロック。下位ノードはロックされません)、“Z” (ZAX モードでのロック)、“L” (ロックが失われた状態。サーバは既にこのロックを持っていません) または “\*” (リモート・ロック) です。指定したロックが標準のロック状態にある場合、または指定したロックが存在しない場合、^\$LOCK は空の文字列を返します。
- ・ “MODE” を指定すると、現在のノードのロック・モードが返されます。返される値は、X (排他ロック・モード)、S (共有ロック・モード)、および ZAX (ZALLOCATE ロック・モード) です。指定したロックが存在しない場合、^\$LOCK は空の文字列を返します。
- ・ “COUNTS” を指定すると、指定したバイナリ・リスト構造としてロックのカウン트가返されます。排他ロックでは、このリストにある要素は 1 つです。共有ロックの場合、このリストにはロックの所有者ごとに 1 つずつの要素があります。pid 引数を使用して、指定したロック所有者のリスト要素のみが返るようにすることができます。各要素には、所有者の

pid、排他モードのインクリメント・カウント、および共有モードのインクリメント・カウントが記録されています。排他モードと共有モードのインクリメント・カウントが両方とも 0 (または、" ") である場合、このロックは 'ZAX' モードです。インクリメント・カウントの後には 'D' が付記されることがあります。これは、現在のトランザクションではロックは解除されていますが、トランザクションがコミットまたはロールバックされるまでロックの解放が遅延することを示しています。指定したロックが存在しない場合、`\$LOCK` は空の文字列を返します。

info\_type キーワードは、すべて大文字で指定する必要があります。無効な info\_type キーワードを指定すると、<SUBSCRIPT> エラーが生成されます。

## pid

ロック所有者のプロセス ID。"COUNTS" キーワードを使用している場合にのみ意味があります。"COUNTS" の返り値を最大でも 1 つのリスト要素に制限するために使用します。pid はどのプラットフォームでも整数で指定します。pid が lock\_name の所有者のプロセス ID と一致する場合、`\$LOCK` は、この所有者の "COUNTS" リスト要素を返します。pid が lock\_name の所有者のプロセス ID と一致しない場合、`\$LOCK` は空の文字列を返します。pid に 0 を指定することは、pid を省略することと同じです。`\$LOCK` は、"COUNTS" リスト要素をすべて返します。pid 引数は、"OWNER"、"FLAGS"、または "MODE" の各キーワードでも使用できますが、無視されます。

## 例

以下の例は、排他ロックの場合に info\_type キーワードにより返される値を示しています。

### ObjectScript

```
LOCK ^B(1,1) ; define lock
WRITE !,"lock owner: ",^$LOCK(^B(1,1),"OWNER")
WRITE !,"lock flags: ",^$LOCK(^B(1,1),"FLAGS")
WRITE !,"lock mode: ",^$LOCK(^B(1,1),"MODE")
WRITE !,"lock counts: "
ZZDUMP ^$LOCK(^B(1,1),"COUNTS")
LOCK ^B(1,1) ; delete lock
```

以下の例は、排他ロックの増減によって info\_type の "COUNTS" で返される値がどのように変化するかを示しています。

### ObjectScript

```
LOCK ^B(1,1) ; define exclusive lock
ZZDUMP ^$LOCK(^B(1,1),"COUNTS")
LOCK ^B(1,1) ; increment lock
ZZDUMP ^$LOCK(^B(1,1),"COUNTS")
LOCK ^B(1,1) ; increment lock again
ZZDUMP ^$LOCK(^B(1,1),"COUNTS")
LOCK ^B(1,1) ; decrement lock
ZZDUMP ^$LOCK(^B(1,1),"COUNTS")
LOCK ^B(1,1) ; decrement lock again
ZZDUMP ^$LOCK(^B(1,1),"COUNTS")
LOCK ^B(1,1) ; delete exclusive lock
```

以下の例は、共有ロックの増減によって info\_type の "COUNTS" で返される値がどのように変化するかを示しています。

### ObjectScript

```
LOCK ^S(1,1)#"S" ; define shared lock
ZZDUMP ^$LOCK(^S(1,1),"COUNTS")
LOCK ^S(1,1)#"S" ; increment lock
ZZDUMP ^$LOCK(^S(1,1),"COUNTS")
LOCK ^S(1,1)#"S" ; increment lock again
ZZDUMP ^$LOCK(^S(1,1),"COUNTS")
LOCK ^S(1,1)#"S" ; decrement lock
ZZDUMP ^$LOCK(^S(1,1),"COUNTS")
LOCK ^S(1,1)#"S" ; decrement lock again
ZZDUMP ^$LOCK(^S(1,1),"COUNTS")
LOCK ^S(1,1)#"S" ; delete shared lock
```

以下の例は \$DATA、\$ORDER、\$QUERY 関数に対して、`\$LOCK` を引数として使用する方法を表しています。

## \$DATA に対する引数としての ^\$LOCK

\$DATA(^\$|namespace|LOCK(lock\_name))

^\$LOCK は \$DATA に対する引数として、ロック名が ^\$LOCK 内のノードとして存在するか否かを示す整数値を返します。以下のテーブルは、\$DATA が返すことができる整数値を示しています。

値	意味
0	ロック名情報は存在しない
10	ロック名情報が存在する

このコンテキストで使用される \$DATA が返すことができるのは、0 または 10 だけであることに注意してください。10 は、指定したロックが存在することを意味します。ロックに下位ノードがあるかどうかを判断することや、1 や 11 を返すことはできません。

以下の例は、現在のネームスペースにロック名が存在するかどうかを調べています。最初の WRITE は 10 を返し（ロック名が存在する）、次の WRITE は 0 を返します（ロック名が存在しない）。

### ObjectScript

```
LOCK ^B(1,2) ; define lock
WRITE !,$DATA(^$LOCK("^B(1,2)"))
LOCK ^B(1,2) ; delete lock
WRITE !,$DATA(^$LOCK("^B(1,2)"))
```

## \$ORDER に対する引数としての ^\$LOCK

\$ORDER(^\$|namespace|LOCK(lock\_name),direction)

^\$LOCK は \$ORDER に対する引数として、指定したロック名に対する照合順序で、次または前の ^\$LOCK ロック名ノードを返します。そのようなロック名が ^\$LOCK ノードとして存在しない場合、\$ORDER は NULL 文字列を返します。

ロックは、大文字/小文字を区別する文字列照合順で返されます。名前付きロックの添え字は、数値の照合を使用して、添え字ツリー順で返されます。

direction 引数は、次のロック名を返すか、前のロック名を返すかを指定します。direction 引数を指定しない場合、InterSystems IRIS は、指定したロック名に対する照合順序で、次のロック名を返します。詳細は、“[\\$ORDER](#)” 関数を参照してください。

以下のサブルーチンは SAMPLES ネームスペースにあるロックを検索し、LOCKET というローカル配列にあるロック名を格納します。

### ObjectScript

```
LOCKARRAY
SET lname=""
FOR I=1:1 {
    SET lname=$ORDER(^$|"SAMPLES"|LOCK(lname))
    QUIT:lname=""
    SET LOCKET(I)=lname
    WRITE !,"the lock name is: ",lname
}
WRITE !,"All lock names listed"
QUIT
```

## \$QUERY に対する引数としての ^\$LOCK

\$QUERY(^\$|namespace|LOCK(lock\_name))

^\$LOCK は \$QUERY に対する引数として、指定したロック名に対する照合順序で、次のロック名を返します。^\$LOCK のノードとして定義された次のロック名が存在しない場合は、\$QUERY は NULL 文字列を返します。

ロックは、大文字/小文字を区別する文字列照合順で返されます。名前付きロックの添え字は、数値の照合を使用して、添え字ツリー順で返されます。

次の例では、現在のネームスペースに 5 つのグローバル・ロック名が (順不同で) 作成されます。

### ObjectScript

```
LOCK (^B(1),^A,^D,^A(1,2,3),^A(1,2))
WRITE !,"lock name: ", $QUERY(^$LOCK(""))
WRITE !,"lock name: ", $QUERY(^$LOCK("^C"))
WRITE !,"lock name: ", $QUERY(^$LOCK("^A(1,2)"))
```

\$QUERY は、添え字付きまたは添え字なしのすべてのグローバル・ロック変数名を文字列として処理し、文字列照合順序内で取得します。したがって、\$QUERY(\$LOCK("")) は、照合順序で最初のロック名 (^\$LOCK("^A") またはより照合順序の高い InterSystems IRIS 定義ロック) を取得します。\$QUERY(\$LOCK("^C")) は、照合順序内で、存在しない ^C の次に当たるロック名 ^\$LOCK("^D") を取得します。\$QUERY(\$LOCK("^A(1,2)")) は、照合順序内でその後に続く ^\$LOCK("^A(1,2,3)") を取得します。

## 関連項目

- ・ [LOCK コマンド](#)
- ・ [\\$DATA 関数](#)
- ・ [\\$ORDER 関数](#)
- ・ [\\$QUERY 関数](#)
- ・ [\\$NAMESPACE 特殊変数](#)
- ・ [ネームスペースの構成](#)



## ^\$ROUTINE (ObjectScript)

ルーチン情報を提供します。

### 構文

```
^$ | nspace | ROUTINE (routine_name)
^$ | nspace | R (routine_name)
```

### 引数

引数	説明
nspace   または [ nspace ]	オプション - <a href="#">拡張 SSVN 参照</a> 。明示的なネームスペース名か、 <a href="#">暗黙のネームスペース</a> のいずれかです。角括弧 ([ "namespace" ])、または垂直バー (  "namespace"  ) で囲まれた、引用符付きの文字列に評価されなければなりません。ネームスペース名は、大文字と小文字を区別しません。ネームスペース名はすべて大文字で格納および表示されます。
routine_name	ルーチン名を含む文字列に評価される式

### 概要

現在のネームスペース (既定)、または指定されたネームスペースからのルーチン情報を返すために、^\$ROUTINE 構造化システム変数を \$DATA、\$ORDER、\$QUERY 関数に対する引数として使用できます。^\$ROUTINE は、ルーチンの OBJ コード・バージョンのルーチン情報を返します。

InterSystems ObjectScript では、ルーチンは 3 つのコード・バージョンで存在します。MAC (ユーザが記述したコードで、マクロ・プリプロセッサ文が含まれることがあります)、INT (コンパイル済みの MAC コードで、マクロの事前処理を実行します)、および OBJ (実行可能なオブジェクト・コード) です。INT コード・バージョンの情報を返すには、[ROUTINE グローバル](#)を使用できます。OBJ コード・バージョンの情報を返すには、^\$ROUTINE を使用できます。

### 引数

#### namespace

このオプションの引数では、[拡張 SSVN 参照](#)を使用することで他のネームスペースのグローバルを指定できます。ネームスペースは、引用符付きの文字列リテラルまたは変数として、明示的にネームスペース名を指定するか、もしくは[暗黙のネームスペース](#)を指定します。ネームスペース名は、大文字と小文字を区別しません。括弧付き構文 ["USER"] か、環境構文 | "USER" | のいずれかを使用します。namespace 区切り文字の前後にスペースを使用することはできません。

以下のメソッドを使用して、ネームスペースが定義されているかどうかをテストすることができます。

#### ObjectScript

```
WRITE ##class(%SYS.Namespace).Exists("USER"),! ; an existing namespace
WRITE ##class(%SYS.Namespace).Exists("LOSER") ; a non-existent namespace
```

\$NAMESPACE 特殊変数を使用して、現在のネームスペースを決定できます。現在のネームスペースを変更するお勧めの方法は、NEW \$NAMESPACE に続けて SET \$NAMESPACE="namespace" を使用することです。

#### routine\_name

既存のルーチン名を含む文字列に評価される式。ルーチン名の最初の 255 文字以内は一意である必要があります。220 文字を超える長さのルーチン名は避けてください。

## 例

以下の例は \$DATA、\$ORDER、\$QUERY 関数に対して、^\$ROUTINE を引数として使用方法を表しています。

### \$DATA に対する引数としての ^\$ROUTINE

\$DATA(^\$|namespace|ROUTINE(routine\_name))

^\$ROUTINE は \$DATA に対する引数として、routine\_name の OBJ コード・バージョンが ^\$ROUTINE 内のノードとして存在するか否かを示す整数値を返します。以下のテーブルは、\$DATA が返すことができる整数値を示しています。

値	意味
0	ルーチン名は存在しない
1	ルーチン名が存在する

以下のターミナルの例では、myrou ルーチンの OBJ コード・バージョンが存在するかどうかをテストしています。この例では、myrou という名前のコンパイル済みの MAC ルーチンが USER ネームスペースに存在することを想定しています。

#### Terminal

```

USER>WRITE ^ROUTINE("myrou",0,"GENERATED") // INT code version exists
1
USER>WRITE $DATA(^$ROUTINE("myrou")) // OBJ code version exists
1
USER>KILL ^rOBJ("myrou") // Kills the OBJ code version

USER>DO ^myrou

DO ^myrou
^
<NOROUTINE> *myrou
USER>WRITE ^ROUTINE("myrou",0,"GENERATED") // INT code version exists
1
USER>WRITE $DATA(^$ROUTINE("myrou")) // OBJ code version does not exist
0
USER>
```

### \$ORDER に対する引数としての ^\$ROUTINE

\$ORDER(^\$|namespace|ROUTINE(routine\_name),direction)

^\$ROUTINE は \$ORDER に対する引数として、指定したルーチン名に対する照合順序で、次または前のルーチン名を返します。そのようなルーチン名が ^\$ROUTINE のノードとして存在しなければ、\$ORDER は NULL 文字列を返します。

direction 引数は、次のルーチン名を返すか、前のルーチン名を返すかを指定します。1 = 次、-1 = 前です。direction 引数を指定しない場合、InterSystems IRIS は、指定したルーチン名に対する照合順序で、次のルーチン名を返します。詳細は、“[\\$ORDER](#)” 関数を参照してください。

以下のサブルーチンは、USER ネームスペースを検索し、ROUTINE というローカル配列にあるルーチン名を格納します。

#### ObjectScript

```

SET rname=""
FOR I=1:1 {
    SET rname=$ORDER(^$|"USER"|ROUTINE(rname))
    QUIT:rname=""
    SET ROUTINE(I)=rname
    WRITE !,"Routine name: ",rname
}
WRITE !,"All routines stored"
QUIT
```



## \$QUERY に対する引数としての ^\$ROUTINE

`$QUERY(^$|namespace|ROUTINE(routine_name))`

^\$ROUTINE は \$QUERY に対する引数として、指定したルーチン名に対する照合順序で、次のルーチン名を返します。指定したルーチン名は、存在していなくてもかまいません。照合順序内でそれ以降にルーチン名がない場合は、\$QUERY(^\$ROUTINE) は NULL 文字列を返します。

次の例では、2 つの \$QUERY 関数が、USER ネームスペース内で、指定したルーチン名の次のルーチンを返します。

### ObjectScript

```
SET rname=""
WRITE !,"1st routine: ", $QUERY(^$|"USER"|ROUTINE(rname))
SET rname="%m"
WRITE !,"1st ", rname, " routine: ", $QUERY(^$|"USER"|ROUTINE(rname))
QUIT
```

## 関連項目

- ・ [\\$DATA](#) 関数
- ・ [\\$ORDER](#) 関数
- ・ [\\$QUERY](#) 関数
- ・ [\\$NAMESPACE](#) 特殊変数
- ・ [ネームスペースの構成](#)



# マクロ・プリプロセッサ指示文

このリファレンスでは、[マクロ](#)の定義で使えるプリプロセッサ指示文に関する情報を提供します。

注釈 これらの指示文では、大文字と小文字は区別されません。例えば、`#include` は `#INCLUDE` (大文字と小文字の他の組み合わせも同様) と同等に扱われます。

## #;

---

1 行コメントを作成します。

### 説明

このマクロ・プリプロセッサ指示文は、.int コードには表示されない 1 行コメントを作成します。このコメントは .mac コードまたはインクルード・ファイルでのみ表示されます。#; は、行の開始部 (1 列目) に表示されます。その行の残りの部分がコメントになります。以下の形式をとります。

#### ObjectScript

```
#; Comment here...
```

ここで、#; の後にコメントが続きます。

#; は行全体をコメントにします。現在の行の残りの部分をコメントにする、##; プリプロセッサ指示文と比較します。

## #deflarg

引数が 1 つのみのマクロを定義します。この引数にはコンマを含めることができます。

### 説明

この**マクロ**・プリプロセッサ指示文は、引数が 1 つのみのマクロを定義します。この引数にはコンマを含めることができます。#deflarg は、#define の特別なバージョンです。#define は、引数の間区切り文字として引数内のコンマを扱うためです。以下の形式をとります。

#### ObjectScript

```
#deflarg Macro(%Arg) Value
```

以下はその説明です。

- Macro は、定義されるマクロの名前で、1 つのみ引数を取ります。有効なマクロ名は、英数字文字列です。
- %Arg は、マクロの引数の名前です。マクロの引数を指定する変数の名前は、パーセント符号で始まる必要があります。
- Value は、マクロの値です。これには、実行時に指定される %Arg の値の使用も含まれます。

#deflarg の行には、**##**; コメントを記述できます。

マクロの定義に関する一般的な情報は、#define のエントリを参照してください。

例えば、以下の MarxBros マクロは、引数としてマルクス兄弟の名前をコンマで区切ったリストを受け入れます。

#### ObjectScript

```
#deflarg MarxBros(%MBNames) WRITE "%MBNames:",!, "The Marx Brothers!",!  
// some movies have all four of them  
$$$MarxBros(Groucho, Chico, Harpo, and Zeppo)  
WRITE !  
// some movies only have three of them  
$$$MarxBros(Groucho, Chico, and Harpo)
```

ここで、MarxBros マクロは、%MBNames という引数を 1 つ取ります。この引数は、マルクス兄弟の名前をコンマで区切ったリストを受け入れます。

## #define

---

マクロを定義します。

### 説明

このマクロ・プリプロセッサ指示文はマクロを定義します。以下の形式をとります。

#### ObjectScript

```
#define Macro[(Args)] [Value]
```

以下はその説明です。

- Macro は、定義されるマクロの名前です。有効なマクロ名は、英数字文字列です。
- Args (オプション) は、マクロが受け入れる 1 つ以上の引数です。これらは、(arg1, arg2, ...) の形式をとります。マクロの引数を指定する各変数の名前は、パーセント符号で始まる必要があります。引数の値にコンマを含めることはできません。
- Value (オプション) は、マクロに割り当てられる値です。この値には、任意の有効な ObjectScript コードを指定できます。リテラルなどの単純な値にすることも、式などの複雑な値にすることもできます。

マクロが値付きで定義されている場合、その値によって ObjectScript コード内のマクロが置き換えられます。値を持たないマクロを定義すると、コードでは他のプリプロセッサ指示文を使用して、そのマクロが存在するかどうかをテストし、その結果に応じてアクションを実行できます。

`##continue` を使用すると、`#define` 指示文を次の行に継続できます。`##;` を使用すると、`#define` 行にコメントを追加できます。ただし、`##continue` と `##;` を同じ行で使用することはできません。

### 値付きのマクロ

値付きのマクロは、ObjectScript コード内で単純なテキストを置き換える機能を提供します。ObjectScript コンパイラがマクロの呼び出し (`$$$MacroName` の形式で) に遭遇するたびに、ObjectScript コードの現在の位置で、マクロに指定されている値が置き換えられます。マクロの値は、任意の有効な ObjectScript コードにできます。これには以下が含まれます。

- 文字列
- 数値
- クラス・プロパティ
- メソッド、関数、または他のコードの呼び出し

マクロの引数にコンマを含めることはできません。コンマが必要な場合は、`#deflarg` 指示文を使用できます。

以下は、さまざまな方法で使われるマクロの定義の例です。

## ObjectScript

```
#define Macro1 22
#define Macro2 "Wilma"
#define Macro3 x+y
#define Macro4 $Length(x)
#define Macro5 film.Title
#define Macro6 +$h
#define Macro7 SET x = 4
#define Macro8 DO ##class(%Library.PopulateUtils).Name()
#define Macro9 READ !,"Name: ",name WRITE !,"Nice to meet you, ",name,!

#define Macro1A(%x) 22+%x
#define Macro2A(%x) "Wilma" _ ": %x"
#define Macro3A(%x) (x+y)*%x
#define Macro4A(%x) $Length(x) + $Length(%x)
#define Macro5A(%x) film.Title _ ": " _ film.%x
#define Macro6A(%x) +$h - %x
#define Macro7A(%x) SET x = 4+%x
#define Macro8A(%x) DO ##class(%Library.PopulateUtils).Name(%x)
#define Macro9A(%x) READ !,"Name: ",name WRITE !,"%x ",name,!
#define Macro9B(%x,%y) READ !,"Name: ",name WRITE !,"%x %y",name,!
```

## マクロの値の規則

マクロは任意の値を持つことができますが、マクロをリテラル式または完全な実行可能な行とすることが規則です。例えば、以下は有効な ObjectScript の構文です。

### ObjectScript

```
#define Macro7 SET x =
```

ここで、マクロは以下のようなコードで呼び出される可能性があります。

```
$$$Macro7 22
```

プリプロセッサは、以下のコードに展開します。

```
SET x = 22
```

これは、明らかに有効な ObjectScript 構文ですが、このようにマクロを使用することはお勧めしません。

## 値なしのマクロ

マクロは、値なしに定義できます。この場合、マクロの存在（または存在しないこと）は、特定の条件が存在することを指定します。次に、他のプリプロセッサ指示文を使用して、マクロが存在するかどうかをテストし、その結果に従ってアクションを実行できます。例えば、Unicode 実行可能プログラムまたは 8ビット実行可能プログラムのいずれかとしてアプリケーションがコンパイルされる場合、以下のようなコードとなる可能性があります。

### ObjectScript

```
#define Unicode
#ifDef Unicode
    // perform actions here to compile a Unicode
    // version of a program
#else
    // perform actions here to compile an 8-bit
    // version of a program
#endif
```

## JSON エスケープ円記号の制限

マクロは、\" エスケープ規則が含まれる JSON 文字列を受け入れません。マクロ値または引数は、リテラル円記号に JSON \" エスケープ・シーケンスを使用できません。このエスケープ・シーケンスは、マクロの本文またはマクロ拡張に渡される仮引数で使用するとは認められていません。代わりに、\" エスケープを \"u0022 に変換できます。この代わりの方法は、キー名と要素値の両方として使用される JSON 構文文字列に有効です。リテラル円記号が含まれる JSON

文字列が、JSON 配列または JSON オブジェクトの要素値として使用される場合は、別の方法として、\" が含まれる JSON 文字列を、同じ文字列値に評価する ObjectScript 文字列式に置き換えることができます。この文字列は括弧で囲む必要があります。



## #dim

ローカル変数のデータ型を指定し、必要に応じてその初期値を指定できます。データ型がオブジェクト・クラスの場合、IDE ではコード入力サポートが提供されます。

### 説明

この**マクロ**・プリプロセッサ指示文は、ローカル変数のデータ型を指定し、必要に応じてその初期値を指定できます。#dim は、コードを記述するための便利なオプションとして提供されています。ObjectScript は型のない言語で、変数のデータ型を宣言することも、#dim で指定されたデータ型を強制することはありません。#dim 指示文は、以下のいずれかの形式で指定できます。

#### ObjectScript

```
#dim VariableName As DataType
#dim VariableName As DataType = InitialValue
#dim VariableName As List Of DataType
#dim VariableName As Array Of DataType
```

以下はその説明です。

- VariableName には、定義する変数、またはコンマで区切られた変数のリストを指定します。
- DataType は、VariableName のデータ型、つまりクラス名です。
- InitialValue は、VariableName について必要に応じて指定する値です。これは SET VariableName=InitialValue と同じです。(この構文はリストや配列には使用できません)。

DataType は主に、コード入力をサポートしている IDE で使用されます。

### InitialValue

- VariableName にデータ変数のコンマ区切りのリストを指定し、DataType がデータ型クラスの場合、すべての変数が同じ値を受け取ります。以下に例を示します。

#### ObjectScript

```
#dim d,e,f As %String = "abcdef"
```

これは以下と同じです。

#### ObjectScript

```
SET d = "abcdef"
SET e = "abcdef"
SET f = "abcdef"
```

- VariableName にオブジェクト変数のコンマ区切りのリストを指定し、DataType がオブジェクト・クラスの場合、各変数に別個の OREF が割り当てられます。以下の例では、各変数に別個の OREF が割り当てられます。

#### ObjectScript

```
#dim j,k,l As Sample.Person = ##class(Sample.Person).%New()
```

これは以下と同じです。

#### ObjectScript

```
SET j = ##class(Sample.Person).%New()
SET k = ##class(Sample.Person).%New()
SET l = ##class(Sample.Person).%New()
```

ただし、`As DataType` 修飾を省略した場合（一般的なユース・ケースではありません）、すべての変数が同じ OREF を受け取ります。

#### ObjectScript

```
#dim j,k,l = ##class(Sample.Person).%New()
```

その結果は、以下と同じです。

#### ObjectScript

```
SET j = ##class(Sample.Person).%New()  
SET k = j  
SET l = j
```

- ・ `VariableName` に変数のコンマ区切りのリストを指定し、`DataType` がダイナミック・オブジェクト・クラスの場合、通常のオブジェクト・クラスの場合と同様に、各変数に別個の OREF が割り当てられます。以下の例では、各変数に別個の OREF が割り当てられます。

#### ObjectScript

```
#dim m,n,o As %DynamicObject = {"name":"Fred"}
```

以下の例でも、同様です。

#### ObjectScript

```
#dim p,q,r As %DynamicArray = ["element1","element2"]
```

ただし、`As DataType` 修飾を省略した場合（一般的なユース・ケースではありません）、通常のオブジェクト・クラスと同様に、すべての変数が同じ OREF を受け取ります。

## #else

一連のプリプロセッサ条件でフォールスルー・ケースの開始を指定します。

### 説明

このマクロ・プリプロセッサ指示文は、一連のプリプロセッサ条件でフォールスルー・ケースの開始を指定します。この後に、`#ifDef`、`#if`、または `#elseif` を置くことができます。`#endif` がこの後に続きます。以下の形式をとります。

#### ObjectScript

```
#else
// subsequent indented lines for specified actions
#endif
```

`#else` 指示文のキーワードは、1 行に単独で記述する必要があります。同一行で `#else` に続く内容はコメントと見なされるので解析されません。

`#if` と組み合わせて `#else` を使用する例は、`#if` 指示文の説明を参照してください。`#endif` と組み合わせて使用する例は、`#endif` 指示文の説明を参照してください。

**注釈**     リテラル値の 0 および 1 以外の引数を持つ `#else` をメソッド・コードで使用すると、コンパイラでは、スーパークラスにあるそのメソッドを呼び出さず、サブクラスにコードを生成します。このコードの生成を回避するには、0 または 1 の値の条件をテストします。この値を指定したほうが簡潔なコードになり、パフォーマンスも最適化できます。

## #elseif

---

#if で始まる一連のプリプロセッサ条件で 2 番目のケースの開始を指定します。

### 説明

このマクロ・プリプロセッサ指示文は、#if で始まる一連のプリプロセッサ条件で 2 番目のケースの開始を指定します。したがって、この後に、#if、または別の #elseif を置くことができます。この後に、別の #elseif、#else、または #endif を記述できます(#elseif 指示文は、#ifDef や #ifNDef では使用できません)。以下の形式をとります。

```
#elseif <expression>
  // subsequent indented lines for specified actions

  // next preprocessor directive
```

ここで、<expression> は有効な ObjectScript 式です。<expression> がゼロ以外の値の場合に True になります。

任意個数のスペースで #elseif と <expression> を区切ることもできます。ただし、<expression> 内ではスペースが認められません。同一行にて <expression> に続くものはコメントと見なされるので、解析はされません。

例については、“#if” を参照してください。

注釈    #elseif には、#ElIf の代替となる名前があります。2 つの名前は同一の動作をします。

---

## #endif

---

一連のプリプロセッサ条件を終了します。

### 説明

このマクロ・プリプロセッサ指示文は、一連のプリプロセッサ条件を終了します。この後に、#ifDef、#ifUnDef、#if、#elseif、および #else を記述できます。以下の形式をとります。

```
// #ifDef, #if, or #else specifying the beginning of a condition
// subsequent indented lines for specified actions
#endif
```

#endif 指示文のキーワードは、1 行に単独で記述する必要があります。同一行で #endif に続く内容はコメントと見なされるので解析されません。

例については、“#if” を参照してください。

## #execute

---

コンパイル時に ObjectScript の行を実行します。

### 説明

このマクロ・プリプロセッサ指示文は、コンパイル時に ObjectScript の行を実行します。以下の形式をとります。

```
#execute <ObjectScript code>
```

ここで、#execute に続く内容は、有効な ObjectScript コードです。このコードは、コンパイル時に値を持つ任意の変数またはプロパティを参照できます。また、コンパイル時に使用可能な任意のメソッドまたはルーチンを呼び出すことができます。ObjectScript コマンドと関数は、常に呼び出すことができます。

#execute は、コードの実行に成功したかどうかを示す値を返しません。コードの実行結果を示す状態コードなどの情報はアプリケーション・コード側で確認する必要があります。この確認操作には、別の #execute 指示文や他のコードを使用できます。

**注釈** ローカル変数と共に #execute を使用すると、予期しない結果が生じる場合があります。この理由は、以下のとおりです。

- ・ コンパイル時に使用される変数が、実行時に範囲外になる可能性があります。
- ・ 複数のルーチンまたはメソッドで、参照時に変数が利用できない場合があります。この問題は、アプリケーション・プログラマがコンパイル順序を制御しないと、悪化する場合があります。

例えば、コンパイル時に曜日を決定し、以下のコードを使用して保存できます。

### ObjectScript

```
#execute KILL ^DayOfWeek
#execute SET ^DayOfWeek = $ZDate($H,12)

WRITE "Today is ",^DayOfWeek,".",!;
```

この ^DayOfWeek グローバルは、コンパイルを実行するたびに更新されます。

## #if

条件テキストのブロックを開始します。

### 説明

この**マクロ**・プリプロセッサ指示文は、条件テキストのブロックを開始します。この指示文は、ObjectScript 式で引数の真理値をテストして、引数の真理値が True の場合にコード・ブロックをコンパイルします。このコード・ブロックは、`#else`、`#elseif`、または `#endif` のいずれかの指示文で終了します。

```
#if <expression>
  // subsequent indented lines for specified actions

  // next preprocessor directive
```

ここで、`<expression>` は有効な ObjectScript 式です。`<expression>` がゼロ以外の値の場合に True になります。

以下に例を示します。

#### ObjectScript

```
KILL ^MyColor, ^MyNumber
#define ColorDay $ZDate($H,12)
#if $$$ColorDay="Monday"
  SET ^MyColor = "Red"
  SET ^MyNumber = 1
#elseIf $$$ColorDay="Tuesday"
  SET ^MyColor = "Orange"
  SET ^MyNumber = 2
#elseIf $$$ColorDay="Wednesday"
  SET ^MyColor = "Yellow"
  SET ^MyNumber = 3
#elseIf $$$ColorDay="Thursday"
  SET ^MyColor = "Green"
  SET ^MyNumber = 4
#elseIf $$$ColorDay="Friday"
  SET ^MyColor = "Blue"
  SET ^MyNumber = 5
#else
  SET ^MyColor = "Purple"
  SET ^MyNumber = -1
#endif
WRITE ^MyColor, ", ", ^MyNumber
```

このコードは、コンパイル時に、`ColorDay` マクロの値を曜日の名前に設定します。`#if` で開始する条件文は、`ColorDay` の値を使用して、`^MyColor` 変数の値を設定する方法を決定します。このコードには、`ColorDay` (月曜日の後の各曜日に 1 つずつ) に適用できる複数の条件があります。これらをチェックするために、コードで `#elseif` 指示文を使用します。フォールスルー・ケースは、`#else` 指示文に続くコードです。`#endif` は、条件文を終了します。

任意個数のスペースで `#if` と `<expression>` を区切ることもできます。ただし、`<expression>` 内ではスペースが認められません。同一行にて `<expression>` に続くものはコメントと見なされるので、解析はされません。

**注釈** リテラル値の 0 および 1 以外の引数を持つ `#if` をメソッド・コードで使用すると、コンパイラでは、スーパークラスにあるそのメソッドを呼び出さず、サブクラスにコードを生成します。このコードの生成を回避するには、0 または 1 の値の条件をテストします。この値を指定したほうが簡潔なコードになり、パフォーマンスも最適化できます。

## #ifDef

---

マクロが定義済みであることを実行の条件とする条件コード・ブロックの開始を指定します。

### 説明

このマクロ・プリプロセッサ指示文は、マクロが定義済みであることを実行の条件とする条件コード・ブロックの開始を指定します。以下の形式をとります。

```
#ifDef macro-name
```

ここで、macro-name は、先頭に \$\$\$ 文字が付かずに表示されます。同一行にて macro-name に続くものはコメントと見なされるので、解析はされません。

コードの実行は、マクロが定義済みであることを条件とします。この実行は、#else 指示文に到達するか、#endif 指示文を終了するまで続きます。

#ifDef がチェックするのは、値が何かではなく、マクロが定義済みであるかどうかだけです。したがって、値が 0 (ゼロ) であるマクロであっても、マクロは存在するので #ifDef は条件コードを実行します。

また、#ifDef はマクロが存在するかどうかのみをチェックするので、他の可能性は 1 つのみ (マクロが定義されていない場合) です。これは #else 指示文で処理します。#elseif 指示文は、#ifDef では使用できません。

例えば、以下のコードは、マクロの存在に基づいて、簡単な 2 つのメッセージのいずれかを表示します。

### ObjectScript

```
#define Heads

#ifDef Heads
    WRITE "The coin landed heads up.",!
#else
    WRITE "The coin landed tails up.",!
#endif
```



## #ifNDef

マクロが定義済みでないことを実行の条件とする条件コード・ブロックの開始を指定します。

### 説明

このマクロ・プリプロセッサ指示文は、マクロが定義済みでないことを実行の条件とする条件コード・ブロックの開始を指定します。以下の形式をとります。

```
#ifNDef macro-name
```

ここで、macro-name は、先頭に \$\$\$ 文字が付かずに表示されます。同一行にて macro-name に続くものはコメントと見なされるので、解析はされません。

コードの実行は、マクロが定義されていないことを条件とします。この実行は、#else 指示文に到達するか、#endif 指示文を終了するまで続きます。#elseif 指示文は、#ifNDef では使用できません。

注釈 #ifNDef には、#ifUnDef の代替となる名前があります。2 つの名前は同一の動作をします。

例えば、以下のコードは、マクロが定義されていないことに基づいて、簡単なバイナリ・スイッチを提供します。

### ObjectScript

```
#define Multicolor 256

#ifNDef Multicolor
    SET NumberOfColors = 2
#else
    SET NumberOfColors = $$$Multicolor
#endif
WRITE "There are ",NumberOfColors," colors in use.",!
```

## #import

これ以降に記述されている埋め込み SQL DML 文のスキーマ検索パスを指定します。

### 説明

このマクロ・プリプロセッサ指示文は、これ以降に記述されている埋め込み SQL DML 文のスキーマ検索パスを指定します。

#import は、1 つまたは複数の検索するスキーマ名を指定して、未修飾のテーブル名、ビュー名、またはストアド・プロシージャ名にスキーマ名を指定します。単一のスキーマ名を指定できるほか、複数のスキーマ名をコンマ区切りリストで指定することもできます。スキーマは現在のネームスペースで検索されます。以下の例に示すとおり、Employees.Person テーブルが検出されます。

#### ObjectScript

```
#import Customers,Employees,Sales
&sql(SELECT Name,DOB INTO :n,:date FROM Person)
WRITE "name: ",n," birthdate: ",date,!
WRITE "SQLCODE=",SQLCODE
```

#import 指示文に指定されたすべてのスキーマが検索されます。Person テーブルは、#import にリストされたスキーマのうちのまさに 1 つで見つかる必要があります。#import は、スキーマ検索パス内での一致が必要なため、システム全体の既定のスキーマは使用されません。

ダイナミック SQL では、%SchemaPath プロパティを使用して、未修飾名を解決するためのスキーマ検索パスを指定します。

#import および #sqlcompile path の両方は、未修飾のテーブル名を解決するために使用される 1 つまたは複数のスキーマ名を指定します。これらの 2 つの指示文の違いを以下に示します。

- #import は、あいまいなテーブル名を検出します。#import は、指定されたすべてのスキーマを検索し、すべての一致を検出します。#sqlcompile path は、スキーマの指定されたリストを左から右の順に、最初の一致が検出されるまで検索します。そのため、#import はあいまいなテーブル名を検出できます。#sqlcompile path ではできません。例えば、#import Customers,Employees,Sales は Customers スキーマ、Employees スキーマ、および Sales スキーマの中に Person の出現をちょうど 1 つだけ検出する必要があります。このテーブル名の出現を複数検出した場合、「Table 'PERSON' is ambiguous within schemas」という SQLCODE -43 エラーが発生します。
- #import では、システム全体の既定値を使用できません。#import が、リストされているスキーマのいずれでも Person テーブルを検出できない場合、SQLCODE -30 エラーが発生します。#sqlcompile path が、リストされているスキーマのいずれでも Person テーブルを検出できない場合、システム全体の既定のスキーマが確認されます。
- #import 指示文は付加的です。#import 指示文が複数ある場合、すべての指示文のスキーマでちょうど 1 つの一致のみを解決する必要があります。2 番目の #import を指定しても、前の #import で指定されたスキーマ名のリストは無効化されません。#sqlcompile path 指示文を指定すると、以前の #sqlcompile path 指示文で指定されたパスは上書きされます。以前の #import 指示文で指定されたスキーマ名が #sqlcompile path によって上書きされることはありません。

InterSystems IRIS は、#import 指示文に存在しないスキーマ名を無視します。InterSystems IRIS は、#import 指示文の重複したスキーマ名を無視します。

テーブル名が修飾済みの場合、#import 指示文は適用されません。以下に例を示します。

#### ObjectScript

```
#import Voters
#import Bloggers
&sql(SELECT Name,DOB INTO :n,:date FROM Sample.Person)
WRITE "name: ",n," birthdate: ",date,!
WRITE "SQLCODE=",SQLCODE
```

この場合、InterSystems IRIS は Sample スキーマで Person テーブルを検索します。Voters スキーマと Bloggers スキーマでは検索されません。

- ・ #import は SQL DML 文に適用されます。これを使用して、SQL SELECT クエリの未修飾テーブル名とビュー名、および INSERT、UPDATE、および DELETE 演算の未修飾テーブル名とビュー名を解決できます。#import を使用すると、SQL の [CALL](#) 文にある未修飾プロシージャ名を解決することもできます。
- ・ #import は SQL DDL 文には適用されません。これは、CREATE TABLE やその他の CREATE 文、ALTER 文、DROP 文などのデータ定義文内の未修飾のテーブル名、ビュー名、およびプロシージャ名の解決には使用できません。この項目の定義を作成、変更、または削除する場合に、テーブル、ビュー、またはストアド・プロシージャに未修飾名を指定すると、InterSystems IRIS では #import の値が無視され、[システム全体の既定のスキーマ](#)が使用されます。

[#sqlcompile path](#) プリプロセッサ指示文と比較します。

## #include

---

プリプロセッサ指示文を記述した、指定の名前のファイルをロードします。

### 説明

このマクロ・プリプロセッサ指示文は、プリプロセッサ指示文を記述した、指定の名前のファイルをロードします。以下の形式をとります。

```
#include <filename>
```

filename は、.inc 接頭語を除く、インクルード・ファイルの名前です。インクルード・ファイルは通常、呼び出し元のファイルと同じディレクトリにあります。名前は大文字と小文字が区別されます。

システム提供の #include filename をすべてリストするには、以下のコマンドを発行します。

### ObjectScript

```
ZWRITE ^rINC("%occInclude",0)
```

これらの #include ファイルのうち、1 つのファイルの内容をリストするには、目的のインクルード・ファイルを指定します。以下に例を示します。

### ObjectScript

```
ZWRITE ^rINC("%occStatus",0)
```

INT ルーチンの生成時に事前処理された #include ファイルをリストするには、[^ROUTINE グローバル](#)を使用します。これらの #include 指示文を ObjectScript コードで参照する必要はないことに注意してください。

### ObjectScript

```
ZWRITE ^ROUTINE("myroutine",0,"INC")
```

注釈     ストアド・プロシージャのコードで #include を使用する場合は、以下のようにコロン文字 : を前に記述する必要があります。

### SQL

```
CREATE PROCEDURE SPxx() Language OBJECTSCRIPT {  
  :#include %occConstant  
    SET x=##lit($$NULLOREF)  
}
```

クラスの開始部にてファイルを組み込む場合、指示文にはシャープ記号を含めません。したがって、単一ファイルでは以下ようになります。

```
Include MyMacros
```

複数ファイルでは以下ようになります。

```
Include (MyMacros, YourMacros)
```

例えば、マクロを含む OS.inc ヘッダ・ファイルがあるとします。

```
#define Windows  
#define UNIX
```

## ObjectScript

```
#include OS

#ifdef Windows
    WRITE "The operating system is not case-sensitive.",!
#else
    WRITE "The operating system is case-sensitive.",!
#endif
```

## #noshow

---

インクルード・ファイルに記述されているコメント・セクションを終了します。

### 説明

このマクロ・プリプロセッサ指示文は、インクルード・ファイルに記述されているコメント・セクションを終了します。以下の形式をとります。

```
#noshow
```

#noshow は、#show 指示文の後に続きます。コメント・セクションがファイルの最後まで続く場合でも、それぞれの #show に対応する #noshow を指定することを強くお勧めします。使用例は、#show のエントリを参照してください。

## #show

インクルード・ファイルに記述されているコメント・セクションを開始します。

### 説明

このマクロ・プリプロセッサ指示文は、インクルード・ファイルに記述されているコメント・セクションを開始します。既定で、インクルード・ファイル内のコメントは、呼び出し元のコードには表示されません。したがって、#show から #noshow の範囲外にあるインクルード・ファイルのコメントは、参照元コードには表示されません。

この指示文の形式は、以下のとおりです。

```
#show
```

コメント・セクションがファイルの最後まで続く場合でも、それぞれの #show に対応する #noshow を指定することを強くお勧めします。

次の例で、#include の例として挙げたファイル **OS.inc** には以下のコメントが記述されています。

### ObjectScript

```
#show
// If compilation fails, check the file
// OS-errors.log for the statement "No valid OS."
#noshow
// Valid values for the operating system are
// Windows or UNIX (and are case-sensitive).
```

ここで、最初の 2 行のコメント (If compilation fails... で始まる) は、インクルード・ファイルを含むコードに表示され、次の 2 行のコメント (Valid values... で始まる) は、インクルード・ファイル自体にのみ表示されます。

## #sqlcompile audit

---

後続の埋め込み SQL 文を監査するかどうかを指定します。

### 説明

このマクロ・プリプロセッサ指示文は、後続の埋め込み SQL 文を監査するかどうかを指定するブーリアンです。以下の形式をとります。

```
#sqlcompile audit=value
```

ここで value は ON または OFF のいずれかです。

このマクロ・プリプロセッサ指示文に効力を発揮させるには、%System/%SQL/EmbeddedStatement [システム監査イベント](#) を有効にする必要があります。既定で、このシステム監査イベントは有効になっていません。



## #sqlcompile mode

非推奨。

### 説明

このマクロ・プリプロセッサ指示文は非推奨です。InterSystems IRIS 2020.1 では、ほとんどの操作 (SELECT、INSERT、UPDATE、DELETE を含む) の埋め込み SQL コードは、このプリプロセッサ指示文の設定に関係なく、この SQL コードを含むルーチンのコンパイル時ではなく、SQL コードの実行時 (ランタイム) にコンパイルされます。

以前のリリースでは、`#sqlcompile mode=value` によって、このプリプロセッサ指示文に続くコードの埋め込み SQL に対してコンパイル・モードが指定されていました。これによって、特定の埋め込み SQL の DML コマンドに、Embedded (コンパイル時に埋め込み SQL を処理) と Deferred (実行時まで埋め込み SQL の処理を遅延) のいずれかのコンパイル・モードが指定されていました。

InterSystems IRIS 2020.1 では、埋め込み SQL の DML コマンドはすべて実行時まで遅延され、実行時にクエリ・キャッシュとして処理されます。したがって、埋め込み SQL は常に、コンパイル時に存在しないテーブル、ユーザ定義関数、および他の SQL エンティティを参照できます。

埋め込み SQL 文は、コンパイル時に解析されます。埋め込み SQL 文に無効な SQL (SQL 構文エラーなど) が含まれている場合、コンパイラは、コード `** SQL Statement Failed to Compile **` を生成し、ObjectScript コードのコンパイルを続行します。このように、無効な埋め込み SQL を含むメソッドでクラスをコンパイルすると、SQL エラーが報告されますが、メソッドは生成されます。このメソッドを実行すると、無効な SQL によるエラーが発生します。

詳細は、“[埋め込み SQL の使用法](#)” を参照してください。

注釈 `#sqlcompile mode=Deferred` を、名前が似ているもののまったく異なる目的で使用される `$SYSTEM.SQL.Util.SetOption(“CompileModeDeferred”)` メソッドと混同しないようにしてください。

## #sqlcompile path

これ以降に記述されている埋め込み SQL DML 文のスキーマ検索パスを指定します。

### 説明

このマクロ・プリプロセッサ指示文は、これ以降に記述されている埋め込み SQL DML 文のスキーマ検索パスを指定します。以下の形式をとります。

```
#sqlcompile path=schema1[,schema2[,...]]
```

schema は、現在のネームスペースで未修飾の SQL テーブル名、ビュー名、またはプロシージャ名を検索する際に使用するスキーマ名です。単一のスキーマ名を指定できるほか、複数のスキーマ名をコンマ区切りリストで指定することもできます。複数のスキーマは指定の順序で検索されます。最初の一致が出現すると、検索は終了し、DML 操作が実行されます。いずれのスキーマにも一致が含まれていない場合、システム全体の既定のスキーマが検索されます。

スキーマは指定された順序で検索されるため、あいまいなテーブル名は検出されません。#import プリプロセッサ指示文も、スキーマ名のリストから未修飾の SQL テーブル名、ビュー名、プロシージャ名にスキーマ名を指定します。#import はあいまいな名前を検出します。

InterSystems IRIS は、#sqlcompile path 指示文に存在しないスキーマ名を無視します。InterSystems IRIS は、#sqlcompile path 指示文の重複したスキーマ名を無視します。

- #sqlcompile path は SQL DML 文に適用されます。これを使用して、SQL SELECT クエリの未修飾テーブル名とビュー名、および INSERT、UPDATE、および DELETE 演算の未修飾テーブル名とビュー名を解決できます。  
#sqlcompile path を使用すると、SQL の CALL 文にある未修飾プロシージャ名を解決することもできます。
- #sqlcompile path は SQL DDL 文には適用されません。これは、CREATE TABLE やその他の CREATE 文、ALTER 文、DROP 文などのデータ定義文内の未修飾のテーブル名、ビュー名、およびプロシージャ名の解決には使用できません。この項目の定義を作成、変更、または削除する場合に、テーブル、ビュー、またはストアド・プロシージャに未修飾名を指定すると、InterSystems IRIS では #sqlcompile path の値が無視され、システム全体の既定のスキーマが使用されます。

ダイナミック SQL では、%SchemaPath プロパティを使用して、未修飾名を解決するためのスキーマ検索パスを指定します。

以下の例では、未修飾テーブル名 Person を Sample.Person テーブルに解決します。最初に Cinema スキーマを検索しますが、これには Person というテーブルが含まれていないので、続いて Sample スキーマを検索します。

#### ObjectScript

```
#sqlcompile path=Cinema,Sample
&sql(SELECT Name,Age
      INTO :a,:b
      FROM Person)
WRITE "Name is: ",a,!
WRITE "Age is: ",b
```

検索パスの項目としてスキーマ名を指定した上で、以下のキーワードを指定できます。

- CURRENT\_PATH：前の #sqlcompile path プリプロセッサ指示文で定義されている現在のスキーマ検索パスを指定します。これは、以下の例のように、既存のスキーマ検索パスの先頭や末尾にスキーマを付加するときに広く使用します。

#### ObjectScript

```
#sqlcompile path=schema_A,schema_B,schema_C
#sqlcompile path=CURRENT_PATH,schema_D
```

- ・ **CURRENT\_SCHEMA** : 現在のスキーマのコンテナのクラス名を指定します。クラス・メソッドで #sqlcompile path を定義している場合、CURRENT\_SCHEMA は現在のクラス・パッケージにマップされたスキーマになります。MAC ルーチンで #sqlcompile path を定義している場合、CURRENT\_SCHEMA は構成の既定のスキーマになります。

例えば、#sqlcompile path=CURRENT\_SCHEMA を指定するクラス・メソッドをクラス **User.MyClass** で定義している場合、User パッケージの既定のスキーマ名は SQLUser なので、CURRENT\_SCHEMA は既定で SQLUser に解決されます。これは、さまざまなパッケージにスーパークラスおよびサブクラスがあり、未修飾テーブル名を使用する SQL クエリを持つスーパークラスでメソッドを定義する場合に便利です。CURRENT\_SCHEMA を使用して、テーブル名をスーパークラスのスーパークラス・スキーマおよびサブクラスのサブクラス・スキーマに解決できます。CURRENT\_SCHEMA の検索パスを設定していない場合、テーブル名は両方のクラスのスーパークラス・スキーマに解決されます。

トリガに #sqlcompile path=CURRENT\_SCHEMA を使用している場合は、スキーマ・コンテナ・クラス名が使用されます。例えば、クラス **pkg1.myclass** に #sqlcompile path=CURRENT\_SCHEMA を指定するトリガがあり、クラス **pkg2.myclass** が **pkg1.myclass** の拡張である場合、**pkg2.myclass** クラスをコンパイルするときに、このトリガの SQL 文に記述された非修飾テーブル名はパッケージ **pkg2** のスキーマに解決されます。

- ・ **DEFAULT\_SCHEMA** は、[システム全体の既定のスキーマ](#)を指定します。このキーワードを使用すると、リストされている他のスキーマを検索する前に、スキーマ検索パス内の項目としてシステム全体の既定のスキーマを検索できます。パスに指定されているすべてのスキーマを検索して一致が見つからなかった場合、システム全体の既定のスキーマは常に、スキーマ検索パスの検索後に検索されます。

スキーマ検索パスを指定している場合、SQL クエリ・プロセッサで未修飾の名前を解決するとき、その指定のスキーマ検索パスが最初に使用されます。指定されたテーブルまたはプロシージャが見つからない場合、SQL クエリ・プロセッサは #import (指定されている場合) で指定されているスキーマ、または構成されている[システム全体の既定のスキーマ](#)を検索します。指定されたテーブルがこれらの場所のどこにも見つからない場合は、SQLCODE -30 エラーが生成されます。

スキーマ検索パスの範囲は、それが定義されているルーチンまたはメソッドです。クラス・メソッドでスキーマ・パスを指定している場合、これはそのクラス・メソッドのみに適用され、同じクラスにある他のメソッドには適用されません。MAC ルーチンで指定したスキーマ検索パスの場合は、その指定した位置から別の #sqlcompile path 指示文が現れるまで、またはルーチンの最後に達するまでが範囲となります。

スキーマは現在のネームスペースに対して定義されます。

[#import](#) プリプロセッサ指示文と比較します。

## #sqlcompile select

これ以降に記述されている埋め込み SQL 文のデータ形式モードを指定します。

### 説明

このマクロ・プリプロセッサ指示文は、これ以降に記述されている埋め込み SQL 文のデータ形式モードを指定します。以下の形式をとります。

```
#sqlcompile select=value
```

valueは以下のいずれかになります。

- ・ Display — 画面に合わせてデータ形式を設定し、出力します。
- ・ Logical — データをメモリ内の形式のままにします。
- ・ ODBC — ODBC または JDBC 経由での表示に合わせてデータ形式を設定します。
- ・ Runtime — 表示形式 (DISPLAY または ODBC) から実行時間選択モード値に基づく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。出力値は現在のモードへ変換されます。

実行時間選択モード値の取得および設定の詳細は、“[SelectMode](#)” を参照してください。

- ・ Text — Display の同義語です。
- ・ FDBMS — 埋め込み SQL が FDBMS と同じデータをフォーマットできるようにします。

このマクロの値によって、SELECT 出力ホスト変数の埋め込み SQL 出力データ形式、および埋め込み SQL INSERT、UPDATE、および SELECT 入力ホスト変数の必須の入力データ形式が決まります。詳細は、“[埋め込み SQL とマクロ・プリプロセッサ](#)” を参照してください。

以下の埋め込み SQL の例は、さまざまなコンパイル・モードを使用して、**Sample.Person** テーブルにある **Name** (文字列フィールド)、**DOB** (日付フィールド)、および **Home** (リスト・フィールド) の 3 つのフィールドを返します。

#### ObjectScript

```
#SQLCOMPILE SELECT=Logical
&sql(SELECT Name,DOB,Home
      INTO :n,:d,:h
      FROM Sample.Person)
WRITE "name is: ",n,!
WRITE "birthdate is: ",d,!
WRITE "home is: ",h
```

#### ObjectScript

```
#SQLCOMPILE SELECT=Display
&sql(SELECT Name,DOB,Home
      INTO :n,:d,:h
      FROM Sample.Person)
WRITE "name is: ",n,!
WRITE "birthdate is: ",d,!
WRITE "home is: ",h
```

#### ObjectScript

```
#SQLCOMPILE SELECT=ODBC
&sql(SELECT Name,DOB,Home
      INTO :n,:d,:h
      FROM Sample.Person)
WRITE "name is: ",n,!
WRITE "birthdate is: ",d,!
WRITE "home is: ",h
```

## ObjectScript

```
#SQLCOMPILE SELECT=Runtime
&sql(SELECT Name,DOB,Home
      INTO :n,:d,:h
      FROM Sample.Person)
WRITE "name is: ",n,!
WRITE "birthdate is: ",d,!
WRITE "home is: ",h
```

## #undef

---

既に定義されているマクロの定義を削除します。

### 説明

マクロ・プリプロセッサ指示文は、既に定義されているマクロの定義を削除します。以下の形式をとります。

```
#undef macro-name
```

macro-name は、既に定義されているマクロです。

#undef は、#define または #deflarg の呼び出しの後に続きます。これは、#ifDef および関連付けられたプリプロセッサ指示文 (#else、#endif、および #ifNDef) と連動します。

以下の例では、マクロが定義済みであること、および未定義であることを条件とするコードを示します。

### ObjectScript

```
#define TheSpecialPart

#ifDef TheSpecialPart
    WRITE "We're in the special part of the program.",!
#endif

//
// code here...
//

#undef TheSpecialPart

#ifDef TheSpecialPart
    WRITE "We're in the special part of the program.",!
#else
    WRITE "We're no longer in the special part of the program.",!
#endif

#ifNDef TheSpecialPart
    WRITE "We're still outside the special part of the program.",!
#else
    WRITE "We're back inside the special part of the program.",!
#endif
```

これに対する .int コードは以下のとおりです。

```
WRITE "We're in the special part of the program.",!
//
// code here...
//
WRITE "We're no longer in the special part of the program.",!
WRITE "We're still outside the special part of the program.",!
```

---

## ##;

---

現在の行の残りの部分を .int コードには表示されないコメントにします。

## 説明

このマクロ・プリプロセッサ指示文は、現在の行の残りの部分を .int コードには表示されないコメントにします。このコメントは .mac コードまたはインクルード・ファイルでのみ表示されます。プリプロセッサ指示文のコメントには、常に ##; コメント文字を使用する必要があります。

### ObjectScript

```
#define alphalen ##function($LENGTH("abcdefghijklmnopqrstuvwxyz")) ##; + 100
    WRITE $$$alphalen," is the length of the alphabet"
```

##; コメント文字は、[#define](#)、[#deflarg](#)、または [#dim](#) の各プリプロセッサ指示文に記述できます。このコメント文字を [##continue](#) プリプロセッサ指示文に続けて使用することはできません。[// または ;](#) といった行の残りの部分をコメントと認識させる文字は、プリプロセッサ指示文内では使用しないでください。

##; を ObjectScript コード行または埋め込み SQL コード行の任意の場所で使用して、.int コードに表示されないコメントを指定することもできます。その行の残りの部分がコメントになります。

##; は、埋め込み HTML または埋め込み JavaScript の評価の前に評価されます。

列 1 に表示され行全体をコメントにする、[#;](#) と比較します。##; は、現在の行の残りの部分をコメントにします。##; が行の最初の列に表示された場合は、[#;](#) プリプロセッサ指示文と機能的に同じになります。

## ##beginquote ...##EndQuote

---

text 文字列を引用符で囲み、text 内のすべての引用符を二重にします。

### 説明

##beginquote text ##EndQuote マクロ・プリプロセッサ指示文は、text 文字列を引用符で囲み、text 内のすべての引用符を二重にします。##quote 指示文と目的は似ていますが、##BeginQuote ...##EndQuote では、以下の例に示すように、ペアになっていない括弧または引用符を使用できます。

```
SET code($i(code))=##beginquote SET def="SQL code-generation" &SQL(SELECT Name ##EndQuote
SET code($i(code))=##beginquote FROM Sample.Person) ##EndQuote
```



## ##continue

次の行にマクロ定義を継続して、複数の行でそのマクロ定義をサポートします。

### 説明

このマクロ・プリプロセッサ指示文は、次の行にマクロ定義を継続して、複数の行でそのマクロ定義をサポートします。マクロ定義の行の最後に現れて、以下の形式で継続していることを示します。

```
#define <beginning of macro definition> ##continue
    <continuation of macro definition>
```

マクロ定義では、複数の ##continue 指示文を使用できます。

以下はその例です。

### ObjectScript

```
#define Multiline(%a,%b,%c) ##continue
    SET v=" of Oz" ##continue
    SET line1="%a"_v ##continue
    SET line2="%b"_v ##continue
    SET line3="%c"_v

$$$Multiline(Scarecrow,Tin Woodman,Lion)
WRITE "Here is line 1: ",line1,!
WRITE "Here is line 2: ",line2,!
WRITE "Here is line 3: ",line3,!
```

##continue は、最終行を除くすべてのマクロ定義行の末尾に記す必要があります。これには、コメント行も含まれます。したがって、##continue は ##; または #; 1 行コメント または複数行の /\* コメントテキスト \*/ の各行を次のように終了させる必要があります。

```
#define <beginning of macro definition> ##continue
#; single line comment ##continue
/* Multi-line long ##continue
wordy comment */ ##continue
    <continuation of macro definition>
```

## ##expression

コンパイル時に ObjectScript 式を評価します。

### 説明

このマクロ・プリプロセッサ関数は、コンパイル時に ObjectScript 式を評価します。以下の形式をとります。

```
##expression(content)
```

ここで、content は、引用符で囲まれた文字列または他のプリプロセッサ指示文を含まない、有効な ObjectScript コードです（下記の子にされた ##expression を除く）。

プリプロセッサは、コンパイル時にこの関数の引数の値を評価し、ObjectScript .int コードの評価で ##expression(content) を置き換えます。##expression 内では変数を引用符で囲んで記述する必要があります。そうしない場合、その変数がコンパイル時に評価されます。

以下の例では、簡単な式をいくつか示します。

#### ObjectScript

```
#define NumFunc ##expression(1+2*3)
#define StringFunc ##expression("This is" _ a concatenated string")
    WRITE $$$NumFunc,!
    WRITE $$$StringFunc,!
```

以下の例は、現在のルーチンのコンパイル・タイムスタンプを含む式を定義しています。

#### ObjectScript

```
#define CompTS ##expression("Compiled: " _ $ZDATETIME($HOROLOG) _ ",!")
    WRITE $$$CompTS
```

ここで、##expression の引数は、3 か所で解析され、\_ 演算子を使用して連結されます。

- 最初の文字列、"Compiled: "。これは、二重引用符で区切られています。その中で、二重引用符のペアは、1 つの二重引用符が評価の後に表示されるように指定しています。
- 値、\$ZDATETIME(\$HOROLOG)。\$ZDATETIME 関数により変換および書式設定された、コンパイル時の \$HOROLOG 特殊変数の値。
- 最後の文字列、",!"。これも、二重引用符で区切られています。その中に、二重引用符が 1 組みあります（評価後、1 つの二重引用符となります）。定義される値は WRITE コマンドに渡されるので、WRITE コマンドに改行が含まれるように、最後の文字列には ,! が含まれています。

ルーチンの中間 (.int) コードには、以下のような行が含まれることになります。

#### ObjectScript

```
WRITE "Compiled: 05/29/2018 07:49:30",!
```

### ##expression とリテラル文字列

##expression で解析しても、リテラル文字列は認識されません。引用符内で文字を括弧で囲っても、特別に扱われません。例えば、以下の指示文を考えます。

```
#define MyMacro ##expression(^abc(")",1))
```

引用符が付いた右側の括弧は、引数を指定するための閉じ括弧であるかのように扱われます。

## ##expression の入れ子

InterSystems IRIS では、入れ子にした ##expression を使用できます。他の ##expression に展開するマクロを含む ##expression は、その展開が ObjectScript レベルで評価可能（つまり、プリプロセッサ指示文がない）であって、ObjectScript 変数に格納可能であれば、定義可能です。入れ子にした ##expression では、##expression 式を持つマクロが最初に展開された後、入れ子にした ##expression が展開されます。

##expression では、以下のマクロ関数を入れ子にすることもできます。##BeginLit...##EndLit、##function、##lit、##quote、##SafeExpression、##stripq、##unique。

## ##expression、サブクラス、および ##SafeExpression

メソッドが ##expression を含んでいる場合、クラスのコンパイル時にこれが検出されます。コンパイラでは ##expression の内容を解析しないため、この ##expression によってサブクラスで本来とは異なるコードが生成される可能性があります。これを回避するために、InterSystems IRIS では、サブクラスごとにコンパイラでメソッド・コードを再生成するようにしています。例えば、##expression(%classname) では現在のクラス名を挿入しますが、サブクラスのコンパイル時には、サブクラスのクラス名が挿入されている必要があります。InterSystems IRIS では、メソッドを強制的にサブクラス内で再生成することで、この状態が確実に発生するようになっています。

サブクラス内のコードが別のものにならないことがわかっている場合は、サブクラスごとのメソッドの再生成を回避してもかまいません。そのためには、##SafeExpression プリプロセッサ関数を ##expression の代わりに使用します。それ以外の場合、これら 2 つのプリプロセッサ関数は同じ機能となります。

## ##expression の動作

以下に示すように ##expression への引数には、ObjectScript [XECUTE](#) コマンドで値を設定します。

```
SET value="Set value="_expression XECUTE value
```

ここで、expression は、value の値を決定する ObjectScript 式です。この式には、マクロや ##expression プリプロセッサ関数は使用できません。

ただし、XECUTE value の結果には、マクロや別の ##expression を使用できます。この例のように、ObjectScript プリプロセッサは、これらのどれであってもさらに展開を進めます。

ルーチン **A.mac** に次の式が記述されているとします。

### ObjectScript

```
#define BB ##expression(10_"_"_$$aa^B())
SET CC = $$$BB
QUIT
```

また、ルーチン **B.mac** には次の式が記述されているとします。

### ObjectScript

```
aa()
QUIT "##expression(10+10+10)"
```

この結果、**A.int** の内容は次のようになります。

### ObjectScript

```
SET CC = 10_30
QUIT
```

## ##function

コンパイル時に ObjectScript 関数を評価します。

### 説明

このマクロ・プリプロセッサ関数は、コンパイル時に ObjectScript 関数を評価します。以下の形式をとります。

```
##function(content)
```

ここで、content は ObjectScript 関数で、ユーザ定義にできます。##function は、関数からの返り値で ##function(content) を置き換えます。

以下の例は、ObjectScript 関数から値を返します。

#### ObjectScript

```
#define alphalen ##function($LENGTH("abcdefghijklmnopqrstuvwxyz"))  
WRITE $$alphalen
```

以下の例では、GetCurrentTime.mac ファイルにユーザ定義関数があるとします。

#### ObjectScript

```
Tag1()  
KILL ^x  
SET ^x = "" _ $Horolog _ ""  
QUIT ^x
```

以下に示すように、ShowTimeStamps.mac と呼ばれる別個のルーチンでこのコードを呼び出すことができます。

#### ObjectScript

```
Tag2  
#define CompiletimeTimeStamp ##function($$Tag1^GetCurrentTime())  
#define RuntimeTimeStamp $$Tag1^GetCurrentTime()  
SET x=$$CompiletimeTimeStamp  
WRITE x,!  
SET y=$$RuntimeTimeStamp  
WRITE y,!
```

ターミナルでの出力は、以下のようになります。

#### Terminal

```
USER>d ^ShowTimeStamps  
64797,43570  
"64797,53807"  
USER>
```

ここで、出力の最初の行は、コンパイル時の \$Horolog の値で、2 行目は、実行時の \$Horolog の値です（出力の最初の行は引用符で囲まれておらず、2 行目は囲まれています。これは、x がその値の代わりに引用符で囲まれた文字列を使用するためです。したがって、引用符はターミナルに表示されませんが、y は引用符に囲まれた文字列を直接ターミナルに出力します）。

注釈     ##function 呼び出しの返り値が、呼び出しのコンテキストで意味と構文の両方が理解できることを確認するのは、アプリケーション・プログラマの責任です。

## ##function の入れ子

InterSystems IRIS では、##function 内の入れ子がサポートされます。##function では、##BeginLit...##EndLit、##function、##lit、##quote、##expression、##SafeExpression、##stripq、##unique、および ##this の各マクロ関数を入れ子にすることができます。

## ##lit

---

リテラル形式でその引数の内容を保持します。

### 説明

このマクロ・プリプロセッサ関数は、リテラル形式でその引数の内容を保持します。

```
##lit(content)
```

ここで、content は、有効な ObjectScript 式である文字列です。##lit プリプロセッサ関数は、受け取る文字列を評価せずに、リテラル・テキストとして扱うようにします。

以下はコードの例です。

```
#define Macro1 "Row 1 Value"
#define Macro2 "Row 2 Value"
  ##lit(;;) Column 1 Header ##lit(;) Column 2 Header
  ##lit(;;) Row 1 Column 1 ##lit(;) $$$Macro1
  ##lit(;;) Row 2 Column 1 ##lit(;) $$$Macro2
```

これは、.int コード内のテーブルを生成する一連の行を作成します。

```
;; Column 1 Header ; Column 2 Header
;; Row 1 Column 1 ; "Row 1 Value"
;; Row 2 Column 1 ; "Row 2 Value"
```

##lit プリプロセッサ関数を使用することにより、マクロが評価され、.int コードにセミコロンで区切って記述されます。

## ##quote

単一の引数を取り、その引数を引用符付きで返します。

### 説明

このマクロ・プリプロセッサ関数は、単一の引数を取り、その引数を引用符付きで返します。その引数に既に引用符文字が付いている場合、引用符文字を二重にすることによってこれらの引用符文字をエスケープします。以下の形式をとります。

```
##quote(value)
```

value は、引用符付きの文字列に変換されるリテラルです。value では、括弧文字または引用符文字はペアで使用する必要があります。例えば、以下は有効な value です。

#### ObjectScript

```
#define qtest ##quote(He said "Yes" after much debate)
ZZWRITE $$$qtest
```

これは、"He said ""Yes"" after much debate" を返します。##quote(This (") is a quote character) は有効な value ではありません。

value 文字列内の括弧はペアになっている必要があります。以下は有効な value です。

#### ObjectScript

```
#define qtest2 ##quote(After (a lot of) debate)
ZZWRITE $$$qtest2
```

[##beginquote .... ##EndQuote](#) 指示文では、ペアになっていない括弧文字または引用符文字を使用できます。

以下の例は、##quote の使用法を示しています。

#### ObjectScript

```
#define AssertEquals(%e1,%e2) DO AssertEquals(%e1,%e2,##quote(%e1)_ == "_##quote(%e2))
Main ;
    SET a="abstract"
    WRITE "Test 1:",!
    $$$AssertEquals(a,"abstract")
    WRITE "Test 2:",!
    $$$AssertEquals(a_"", "abstract")
    WRITE "Test 3:",!
    $$$AssertEquals("abstract","abstract")
    WRITE "All done"
    QUIT
AssertEquals(e1,e2,desc) ;
    WRITE desc_ is "$_SELECT(e1=e2:"true",1:"false"),!
    QUIT
```

## ##quote の入れ子

InterSystems IRIS では、##quote 関数内の入れ子がサポートされます。##quote では、##BeginLit...##EndLit、[##function](#)、[##lit](#)、[##quote](#)、[##expression](#)、[##SafeExpression](#)、[##stripq](#)、[##unique](#)、および [##this](#) の各マクロ関数を入れ子にすることができます。

## ##quoteExp

---

コンパイル時に評価される式を引数として取ります。この式には、入れ子/再帰 MPP 関数を含めることができます。

### 説明

このマクロ・プリプロセッサ関数は、コンパイル時に評価される式を引数として取ります。この式には、入れ子/再帰 MPP 関数を含めることができます。その後、このプリプロセッサ関数は、コンパイルされた結果を引用符付きの文字列として返します。その引数に既に引用符文字が付いている場合、引用符文字を二重にすることによってこれらの引用符文字をエスケープします。以下の形式をとります。

```
##quoteExp(expression)
```

expression には、##BeginLit...##EndLit、##expression、##function、##lit、##quote、##quoteExp、##SafeExpression、##stripq、##unique、および ##This のいずれの入れ子/再帰 MPP 関数も記述できます。

##quoteExp を使用すると、可変個数の添え字を受け入れ、その参照を引用符付き文字列として返す複雑な汎用グローバル・マクロを作成できます。このとき、添え字の値が数値と文字列のいずれとして渡されるかは関係ありません。#deflarg 指示文を使用して、複雑なグローバル参照としてマクロを定義します。この複雑なグローバル参照を引用符付き文字列として返すには、マクロに指定した添え字に関係なく、このマクロを ##quoteExp にラップします。このマクロは、##quoteExp に渡された expression 引数を評価し、この値を引用符付き文字列として返します。

以下に例を示します。

```
#deflarg complexGlobal(%subs)
^GLO("dd"##expression($s(%literalargs'=$lb("):", "_$LTS(%literalargs,","),1:""))
#deflarg complexGlobalQE(%subs) ##quoteExp($$$complexGlobal(%subs))
```



---

## ##sql

---

実行時に、指定の埋め込み SQL 文を呼び出します。

### 説明

このマクロ・プリプロセッサ指示文は、実行時に指定された埋め込み SQL 文を呼び出します。以下の形式をとります。

`##sql(SQL-statement)`

ここで SQL-statement は有効な埋め込み SQL 文です。##sql プリプロセッサ指示文は、&SQL 埋め込み SQL マーカーとまったく同じです。どちらの場合も、括弧で囲まれた SQL コードが、これを含むルーチンのコンパイル時ではなく、実行時にコンパイルされます（最初の実行）。詳細は“[埋め込み SQL のコンパイル](#)”を参照してください。

## ##stripq

---

単一の引数を取り、引用符を削除してその引数を返します。

### 説明

このマクロ・プリプロセッサ関数は、単一の引数を取り、引用符を削除してその引数を返します。これは、##quote マクロ関数とは逆の関数です。

```
##stripq(value)
```

value はリテラルまたは変数で、これが引用符で囲まれている場合は引用符が削除されます。

## ##unique

コンパイル時または実行時に使用するマクロ定義内に、新規で一意のローカル変数を作成します。

### 説明

このマクロ・プリプロセッサ関数は、コンパイル時または実行時に使用するマクロ定義内に、新規で一意のローカル変数を作成します。このプリプロセッサ関数は、`#define` 呼び出しまたは `#deflarg` 呼び出しの一部としてのみ使用できます。以下の形式をとります。

```
##unique(new)
##unique(old)
```

ここで、`new` は、新しい一意の変数の作成を指定し、`old` は、その同じ変数への参照を指定します。

`SET ##unique(new)` によって作成される変数は `%mmmu1` という名前のローカル変数です。後続の `SET ##unique(new)` では、`%mmmu2`、`%mmmu3`、以降同様の名前のローカル変数が作成されます。これらのローカル変数は、すべての `%` ローカル変数と同じ有効範囲ルール ([% 変数は常にパブリック変数です](#)) に従います。これらは、あらゆるローカル変数と同様に、`ZWRITE` を使用して表示でき、引数なしの `KILL` を使用して削除できます。

ユーザ・コードは他の ObjectScript 変数を参照できるように、`##unique(old)` 変数を参照できます。`##unique(old)` 構文は、作成された変数を参照するために、何度でも使用できます。

その後 `##unique(new)` を呼び出すと、新しい変数が作成されます。`##unique(new)` を再度呼び出した後に `##unique(old)` を呼び出すと、次に作成された変数が参照されます。

例えば、以下のコードは、`##unique(new)` と `##unique(old)` を使用して、2 つの変数の値を互いに入れ替えます。

### ObjectScript

```
#define Switch(%a,%b) SET ##unique(new)=%a, %a=%b, %b=##unique(old)
READ "First variable value? ",first,!
READ "Second variable value? ",second,!
$$$Switch(first,second)
WRITE "The first value is now ",first," and the second is now ",second,!
```

これらの変数の一意性を維持するには、以下のようにします。

- ・ `#define` または `#deflarg` プリプロセッサ指示文の外側に `##unique(new)` を設定しようとしない。
- ・ メソッドまたはプロシージャにあるプリプロセッサ指示文に `##unique(new)` を設定しない。これらはメソッド(`%mmmu1`) に固有の変数名を生成しますが、これは `%` 変数であるため、グローバルに有効範囲が設定されます。`##unique(new)` を設定する別のメソッドを呼び出しても `%mmmu1` が作成され、最初のメソッドで作成された変数が上書きされます。
- ・ `%mmmu1` 変数を直接設定しない。InterSystems IRIS は、システムで使用するためにすべての `%` 変数 (`%z` 変数および `%Z` 変数を除く) を予約しています。これらをユーザ・コードで設定しないでください。



# A

## 識別子のルールとガイドライン

ここでは、ObjectScript コード内およびクラス内の識別子のルールについて説明し、名前競合を回避するためのガイドラインを示します。ObjectScript には予約語がありません。そのため、識別子としてコマンドを使用しても結果は構文的に正しくなりますが、そのコードはそれを読む人にわかりにくくなる可能性があります。

ネームスペースおよびデータベースの識別子については、“[システム管理ガイド](#)”の関連するセクションを参照してください。

ユーザ、ロール、リソースなどのセキュリティ・エンティティの識別子については、“[承認ガイド](#)”の関連するセクションを参照してください。

“[ネームスペースで何にアクセス可能か](#)”も参照してください。

### A.1 ローカル変数名のルール

ローカル変数の名前について、ObjectScript では以下のルールが適用されます。

- 最初の文字は、英字かパーセント記号 (%) に限られます。
  - % で始まる名前にする場合、その直後の文字は `z` または `Z` にします。
- 残りの文字は、英字または数字にする必要があり、ASCII 255 より大きい文字 (Unicode 文字) も使用できます。
- 名前は、大文字と小文字を区別します。
- 名前は、最初の 31 文字が (該当するコンテキストで) 一意である必要があります。
  - 変数の添え字は、このカウントに含まれません。

### A.2 回避する必要があるローカル変数名

ローカル変数に対して以下の名前は使用しないでください。

- SQLCODE
  - InterSystems SQL が実行される可能性があるコンテキストでは、変数の名前として SQLCODE を使用しないでください。“[SQLCODE 値とエラー・メッセージ](#)”を参照してください。
- IO、IOF、IOBS、IOM、IOSL、IOT、IOST、IOPAR、MSYS、POP、RMSDF

™IS ユーティリティを使用するコンテキスト(実際には、これはまれです)では、これらの変数名を使用しないでください。”[入出力の概要](#)”を参照してください。

## A.3 グローバル変数名のルール

グローバル変数の名前については、以下のルールが適用されます。

- 最初の文字はキャレット(^)にし、次の文字は英字かパーセント記号(%)にする必要があります。グローバル名の場合、文字は ASCII 65 から ASCII 255 の範囲のアルファベット文字として定義されます。ASCII 255 より大きい文字は許可されていません。
- 残りの文字は、英字または数字にする必要があります(ただし、次の箇条書きに示すように例外が 1 つあります)。
- グローバル変数の名前には、1 つ以上のピリオド(.) 文字を含めることができますが、最初と最後の文字には使用できません。
- 名前は、大文字と小文字を区別します。
- 名前は、最初の 31 文字が(該当するコンテキストで)一意である必要があります。キャレット文字はこのカウントに含まれません。つまり、グローバル変数の名前は、キャレットを含めて最初の 32 文字が一意である必要があります。  
変数の添え字は、このカウントに含まれません。

- IRISSYS データベースでは、グローバル名はすべて予約されていますが、先頭に ^z、^Z、^%z、^%Z が付くものについては例外です。”[IRISSYS のカスタム項目](#)”を参照してください。

他のすべてのデータベースでは、先頭に ^IRIS または ^%IRIS が付くグローバル名はすべて予約されています。

”[回避する必要があるグローバル変数名](#)”も参照してください。

## A.4 回避する必要があるグローバル変数名

データベースを作成すると、InterSystems IRIS によって、それ自体が使用するためのいくつかのグローバルでそれが初期化されます。また、作成するネームスペースすべてに、システム・グローバルへのマッピングが含まれます。これには、書き込み可能システム・データベース内のグローバル・ノードも含まれます。

### A.4.1 パーセント・グローバル

パーセント・グローバルはすべてのネームスペースで使用できます。以下のルールが適用されます。

- 名前の先頭に ^%z または ^%Z が付く独自のグローバルを設定、変更、または削除できます(”[IRISSYS のカスタム項目](#)”を参照)。
- ™SYS を設定、変更、または削除することはできません(ドキュメントに記載されているようにノードを設定する場合を除く)。
- 前述の例外を除き、名前の先頭に ^% が付くグローバルを設定、変更、または削除することはできません。

### A.4.2 非パーセント・グローバル

システム・グローバルを上書きしないようにするために、ネームスペースでは以下のグローバルを設定、変更、または削除しないでください。

- ・ `^CacheTemp*` (InterSystems IRIS の一部のバージョンで使用するために予約されています)
- ・ `^DeepSee.*` (制限は、InterSystems IRIS Analytics を使用しているネームスペースにのみ適用されます)
- ・ `^Ens*` (制限は、相互運用対応ネームスペースにのみ適用されます。“相互運用プロダクションの概要”を参照してください。)
- ・ `^ERRORS`
- ・ `^HS` (制限は、HealthShare ネームスペースに適用されます)
- ・ `^InterSystems.Sequences` (制限は、InterSystems IRIS Hibernate Dialect を使用しているネームスペースにのみ適用されます)
- ・ `^IRIS*` (インターシステムズで使用するために予約されています)
- ・ `^IS.*` (InterSystems IRIS の[シャーディング](#)で使用するために予約されています)
- ・ `^ISC*` (ドキュメントに記載されているようにノードを設定する場合を除く)
- ・ `^mqh` (SQL クエリの履歴)
- ・ `^mtemp*`
- ・ `^OAuth2` (制限は、HealthShare ネームスペースに適用されます)
- ・ `^OBJ.GUID` (ドキュメントに記載されている場合を除く)
- ・ `^OBJ.DSTIME`
- ・ `^OBJ.JournalT`
- ・ `^odd*`
- ・ `^rBACKUP`
- ・ `^rINC` (インクルード・ファイルを含む)
- ・ `^rINCSAVE`
- ・ `^rINDEX`
- ・ `^rINDEXCLASS`
- ・ `^rINDEXEXT`
- ・ `^rINDEXSQL`
- ・ `^rMAC` (MAC コードを含む)
- ・ `^rMACSAVE`
- ・ `^rMAP`
- ・ `^rOBJ` (OBJ コードを格納する)
- ・ `^ROUTINE` (ルーチンを格納する)
- ・ `^SchemaMap` (制限は、HealthShare ネームスペースに適用されます)
- ・ `^SPOOL` (制限は、InterSystems IRIS スプーリングを使用しているネームスペースにのみ適用されます。“[スプール・デバイス](#)”を参照してください。)
- ・ `^SYS` (ドキュメントに記載されているようにノードを設定する場合を除く)
- ・ `^z*` および `^Z*` (**IRISSYS** データベースの場合を除き、インターシステムズで使用するために予約されています。“[IRISSYS のカスタム項目](#)”を参照してください。)

## A.5 ルーチン名とラベルのルール

ルーチンまたはラベルの名前について、ObjectScript では以下のルールが適用されます。

- 最初の文字は、英字かパーセント記号 (%) に限られます。  
ルーチン名の先頭を % にした場合、その直後の文字には z または Z を使用します。“[IRISSYS のカスタム項目](#)”を参照してください。
- 残りの文字は、英字または数字にする必要があります (ただし、例外が 1 つあります。次の箇条書きを参照してください)。これらの他の文字に、ASCII 128 より大きい任意の文字を使用できます。  
Unicode 文字を処理する際に、ローケル識別子は考慮されません。つまり、Unicode 文字で構成される識別子が、あるローケルで有効な場合、その識別子はどのローケルでも有効です。
- ルーチンの名前には、1 つ以上のピリオド (.) 文字を含めることができますが、最初と最後の文字には使用できません。
- 名前は、大文字と小文字を区別します。
- ルーチンの名前は、最初の 255 文字以内で一意 (該当するコンテキスト内) である必要があります。  
ラベルは、最初の 31 文字以内で一意 (該当するコンテキスト内) である必要があります。

特定の z および %z ルーチン名は、[ユーザが使用するために予約されています](#)。

## A.6 ユーザが使用するために予約されているルーチン名

InterSystems IRIS では、ユーザが使用するために、以下のルーチン名が予約されています。これらのルーチンは存在しませんが、それらを定義した場合、これらのイベントが発生したときにそれらがシステムによって自動的に呼び出されます。

- ^ZWELCOME ルーチンは、ObjectScript シェルが起動したときに実行されるカスタム・コードを含めることを目的としています。“[ObjectScript シェルの使用法](#)”を参照してください。
- ^ZAUTHENTICATE および ^ZAUTHORIZE ルーチンは、認証および承認のためのカスタム・コードを含めることを目的としています (代行認証と代行承認をサポートするため)。このようなルーチンのために、InterSystems IRIS にはテンプレートが用意されています。“[代行承認の使用法](#)”と“[代行認証](#)”を参照してください。
- ^ZMIRROR ルーチンは、InterSystems IRIS ミラーリングを使用する場合の、フェイルオーバー動作をカスタマイズするためのコードを含めることを目的としています。“[高可用性ガイド](#)”を参照してください。
- ^%ZSTART および ^%ZSTOP ルーチンは、ユーザ・ログインなどの特定のイベントが発生したときに実行されるカスタム・コードを含めることを目的としています。これらのルーチンは事前定義されません。それらを定義した場合、これらのイベントが発生したときにそれらがシステムによって呼び出されます。“[^%ZSTART ルーチンと ^%ZSTOP ルーチンによる開始動作と停止動作のカスタマイズ](#)”を参照してください。
- ^%ZLANGV00、および名前が ^%ZLANG で始まるその他のルーチンは、カスタム変数、コマンド、および関数を含めることを目的としています。“[^%ZLANG ルーチンによる言語の拡張](#)”を参照してください。
- ^%ZJREAD ルーチンは、^JCONVERT ルーチンを使用する場合にジャーナル・ファイルを操作するためのロジックを含めることを目的としています。“[ジャーナリングの概要](#)”を参照してください。



## A.7 クラス名のルール

どのクラスも、完全クラス名は `packagename.classname` の形式です。

クラス名のルールは、以下のとおりです。

- ・ `packagename` (パッケージ名) および `classname` (短いクラス名) は英字またはパーセント記号 (%) で始まる必要があります。  
 パッケージ名の先頭を % にした場合、その直後の文字には `z` または `Z` を使用します。“[IRISSYS のカスタム項目](#)”を参照してください。
- ・ `packagename` にはピリオドを含めることができます。  
 その場合、ピリオドの直後の文字は英字にする必要があります。  
`packagename` のピリオドで区切られた各部は、サブパッケージ名として扱われ、一意性のルールに従います。
- ・ 残りの文字は、英字または数字にする必要があります、ASCII 128 より大きい文字も使用できます。  
 Unicode 文字を処理する際に、ロケール識別子は考慮されません。つまり、Unicode 文字で構成される識別子が、あるロケールで有効な場合、その識別子はどのロケールでも有効です。
- ・ パッケージ名と短いクラス名は一意である必要があります。同様に、サブパッケージ名は、親パッケージ名内で一意である必要があります。  
 各クラスを定義したときに使用した大文字と小文字はシステムで保持され、クラス定義に指定した大文字と小文字に完全に一致させる必要があります。ただし、大文字と小文字のみが違う 2 つの識別子を指定することはできません。例えば、識別子 `id1` と `ID1` は一意性を保つ目的からは同一と見なされます。
- ・ 以下のように長さの制限があります。
  - パッケージ名 (すべてのピリオドを含む) は最初の 189 文字内で一意である必要があります。
  - 短いクラス名は最初の 60 文字内で一意である必要があります。

完全なクラス名は、[クラス・メンバ](#)の個別の長さ制限に寄与します。

## A.8 回避する必要があるパッケージ名、クラス名、およびスキーマ名

永続クラスについては、クラスの短い名前として SQL 予約語を使用しないでください。

クラスの短い名前として SQL 予約語を使用する場合、そのクラスに対して `SqlTableName` キーワードを指定することが必要になります。また、短いクラス名と SQL テーブル名の間に不一致があると、将来コードを読むときに注意が必要になります。

SQL 予約語のリストは、“[予約語](#)”を参照してください。

以下のパッケージ名を使用することは避けてください (ネームスペースによります)。これらのパッケージ名をスキーマ名として使用することもできません。

- ・ いずれのネームスペースでも、**IRIS** をパッケージ名として使用しないでください。これは、インターシステムズで使用するために予約されています。

- ・ いずれのネームスペースでも、**INFORMATION** をパッケージ名として使用しないでください。これは、すべてのネームスペースにマッピングされるシステム・パッケージです。
- ・ いずれの相互運用対応ネームスペースでも、**Ens**、**EnsLib**、**EnsPortal**、および **CSPX** をパッケージ名として使用しないでください。これらのパッケージは、アップグレード・プロセス中に完全に置換されます。これらのパッケージ内でクラスを定義した場合は、アップグレード前にそれらのクラスをエクスポートして、アップグレード後にインポートする必要があります。
- ・ いずれの相互運用対応ネームスペースでも、先頭に **Ens** (大文字と小文字の区別あり) が付くパッケージ名を使用しないでください。詳細は、“環境上の考慮事項”を参照してください。
- ・ HealthShare ネームスペースでは、**HS**、**HSFHIR**、**HSMOD**、および **SchemaMap** をパッケージ名として使用しないでください。

## A.9 クラス・メンバ名のルール

クラス・メンバについては、その項目の名前が範囲指定されていない限り、名前は以下のルールに従う必要があります。

- ・ 名前は、英字かパーセント記号 (%) で始まる必要があります。

SQL に投影されるクラス・メンバについては、その他の考慮事項があります (例えば、このメンバには永続クラスのほとんどのプロパティが含まれます)。最初の文字が % である場合、2 番目の文字は z または Z である必要があります。

- ・ 残りの文字は、英字または数字にする必要があり、ASCII 128 より大きい文字も使用できます。

- ・ メンバ名は (該当するコンテキスト内で) 一意である必要があります。

クラスを定義したときに使用した大文字と小文字はシステムで保持され、クラス定義に指定した大文字と小文字に完全に一致させる必要があります。ただし、2 つのクラス・メンバに、大文字と小文字のみが違う名前を指定することはできません。例えば、識別子 `id1` と `ID1` は一意性を保つ目的からは同一と見なされます。

- ・ メソッドまたはプロパティ名は最初の 180 文字内で一意である必要があります。
- ・ プロパティの名前の長さと、そのプロパティのインデックスの名前の長さを合わせた長さは、180 文字を超えてはいけません。
- ・ 各メンバの完全な名前 (未修飾のメンバ名および完全なクラス名を含む) は、220 文字以下である必要があります。
- ・ 2 つのメンバに同一の名前を付与しないでください。予測できない結果となる可能性があります。

メンバ名の範囲を指定することもできます。範囲指定したメンバ名を作成するには、名前の最初と最後の文字に二重引用符を使用します。これにより、範囲指定しない場合には許可されない文字を名前に含めることができます。以下に例を示します。

### Class Member

```
Property "My Property" As %String;
```

## A.10 回避する必要があるメンバ名

永続クラスについては、メンバの名前として SQL 予約語を使用しないでください。

これらの名前の 1 つに SQL 予約語を使用すると、そのクラスを SQL に投影する方法を指定するときに余分な作業が必要になります。例えば、プロパティの場合、`SqlFieldName` キーワードを指定する必要があります。また、クラスの識別子と SQL の識別子の間に不一致があると、将来コードを読むときに注意が必要になります。

SQL 予約語のリストは、“[予約語](#)”を参照してください。このリストには、`%SQLUPPER` や `%FIND` のように、名前が `%` で始まる項目が数多く含まれている点に注目してください。このような項目は、SQL に対する InterSystems 拡張機能です。将来のリリースで、その他の拡張機能が追加される可能性もあります。

## A.11 IRISSYS のカスタム項目

IRISSYS データベースに項目を作成できます。アップグレード時に、カスタム項目の名前付け規約に従っていないと、いくつかの項目が削除されます。

項目が上書きされないようにこのデータベースにコードまたはデータを追加するには、以下のいずれかを実行します。

- ・ `%SYS` ネームスペースに移動し、項目を作成します。このネームスペースの場合、既定のルーチン・データベースおよび既定のグローバル・データベースは、どちらも **IRISSYS** です。以下の名前付け規約を使用して、項目がアップグレード・インストールの影響を受けないようにします。
  - － クラス：パッケージを `Z` または `z` で始めます。
  - － ルーチン：名前を `Z`、`z`、`%Z`、または `%z` で始めます。
  - － グローバル：名前を `^Z`、`^z`、`^%Z`、または `^%z` で始めます。
- ・ どのネームスペースでも、以下の名前で作成します。
  - － ルーチン：名前を `%Z` または `%z` で始めます。
  - － グローバル：名前を `^%Z` または `^%z` で始めます。

ネームスペースの標準マッピングにより、これらの項目は **IRISSYS** に書き込まれます。

MAC コードとインクルード・ファイルはアップグレードによる影響を受けません。



# B

## 一般的なシステム制限

ここでは、InterSystems IRIS® データ・プラットフォームのシステム制限のいくつかを示します。

識別子の名前の制限については、“[識別子のルールとガイドライン](#)” を参照してください。

その他のシステム全体の制限は、“[構成パラメータ・ファイル・リファレンス](#)” を参照してください。

### B.1 文字列長の制限

文字列の長さには、3,641,144 文字という制限があります。

文字列は、入出力デバイスからの読み取りの結果だけに限られないことを認識することが重要です。その他のコンテキストでも、文字列は出現します。例えば、結果セット (SQL クエリ、多数の項目を含む \$LIST の構築、または XSLT 変換などの出力として返される) の行に含まれるデータのコンテキストにも文字列が出現します。

### B.2 添え字の制限

ローカル変数、プロセス・プライベート変数、グローバル変数、およびロック名にはすべて、添え字を使用でき、以下の制限が適用されます。

- ・ どの添え字にも最大長があります。以下に示す添え字の最大長を超えると <SUBSCRIPT> エラーになります。
  - ローカル配列の場合、添え字の最大長はエンコード後で 32,767 バイトです。
  - グローバル配列の場合、添え字の最大長はエンコード後で 511 バイトです。

どの場合も、該当の文字数は、添え字および現在のロケールに応じて異なることに注意してください。

また、最長許容整数は 309 桁であり、この制限を超えると <MAXNUMBER> エラーになります。したがって、309 文字よりも長い数値の添え字は、文字列として指定する必要があります。

- ・ ローカル変数の添え字レベルの最大数は 255 です。グローバルまたはプロセス・プライベート・グローバルの添え字レベルの最大数は 253 です。添え字レベルの最大数を超えると <SYNTAX> エラーになります。

## B.3 グローバル参照の最大長

グローバル参照（つまり、特定のグローバル・ノードやサブツリーへの参照）の合計の長さは、エンコード文字数で最長 511 文字です（入力文字数で考えると、511 文字より少ない場合があります）。

任意のグローバル参照のサイズを慎重に決定する場合は、以下のガイドラインを使用します。

1. グローバル名の場合、1 文字ごとに 1 を加算します。
2. 純粋な数字の添え字の場合、1 桁、符号、または小数点ごとに、1 を加算します。
3. 数値以外の文字を含む添え字の場合、1 文字ごとに、3 を加算します。

添え字が純粋な数字ではない場合、添え字の実際の長さは、文字列をエンコードするために使用される文字セットによって異なります。マルチバイトの文字は 3 バイトまでとすることができます。

ASCII 文字は 1 または 2 バイトとなる可能性があることに注意してください。照合で大文字と小文字を区別する場合、ASCII 文字は、文字に対して 1 バイト、および曖昧性解消のバイトに対して 1 バイトとすることができます。照合で大文字と小文字を区別しない場合、ASCII 文字は 1 バイトとなります。

4. 添え字ごとに、1 を追加します。

これらの数の合計が 511 より大きくなった場合、その参照は長過ぎということになります。

制限は決まっているため、長い添え字名やグローバル名にする必要がある場合、多数の添え字レベルを避けることをお勧めします。反対に、複数の添え字レベルを使用している場合は、長いグローバル名や添え字を避けます。使用している文字セットを制御できない場合があるので、グローバル名や添え字は短くすることが有用となります。

特定の参照について懸念がある場合、最長となりそうなグローバル参照と同等の長さか、それより若干長いテスト・バージョンのグローバル参照を作成することをお勧めします。それらテストのデータにより、アプリケーション構築前に、名前付け規約の修正についてのヒントが得られます。

## B.4 クラスの制限

以下の制限はクラスにのみ適用されます。

### クラス継承階層

制限：50。1 つのクラスは最大 50 階層までのサブクラスを作成できます。

### 外部キー

制限：1 クラスあたり 400。

### インデックス

制限：1 クラスあたり 400。

### メソッド

制限：1 クラスあたり 2000。

### パラメータ

制限：1 クラスあたり 1000。

## プロジェクション

制限：1 クラスあたり 200。

## プロパティ

制限：1 クラスあたり 1000。

## クエリ

制限：1 クラスあたり 200。

## SQL 制約

制限：1 クラスあたり 200。

## ストレージ定義

制限：1 クラスあたり 10。

## スーパークラス

制限：1 クラスあたり 127。

## トリガ

制限：1 クラスあたり 200。

## XData ブロック

制限：1 クラスあたり 1000。

# B.5 クラスおよびルーチンの制限

以下の制限はクラスとルーチンに適用されます。

## クラス・メソッド参照

制限：1 ルーチンまたは 1 クラスあたり 32768 の一意の参照。

以下は、メソッド名が同じでもクラス名が異なるため、2 つのクラス・メソッド参照としてカウントされます。

### ObjectScript

```
Do ##class(c1).abc(), ##class(c2).abc()
```

## クラス名参照

制限：1 ルーチンまたは 1 クラスあたり 32768 の一意の参照。

例えば、以下は 2 つのクラス名参照としてカウントされます。

```
Do ##class(c1).abc(), ##class(c2).abc()
```

同様に、以下も、2 つのクラス参照としてカウントされます。これは、%File から %Library.File への正規化がコンパイル時ではなく実行時に行われるためです。

```
Do ##class(%File).Open(x)
Do ##class(%Library.File).Open(y)
```

## インスタンス・メソッド参照

制限：1 ルーチンまたは 1 クラスあたり 32768。

X と Y が OREF の場合、以下は、1 つのインスタンス・メソッド参照としてカウントされます。

```
Do X.abc(), Y.abc()
```

多次元プロパティへの参照は、インスタンス・メソッドとしてカウントされます。これはコンパイラがこれらを区別できないためです。例えば、以下の文を考えてみます。

```
Set var = OREF.xyz(3)
```

コンパイラは、この文が xyz() メソッドを参照しているか、または多次元プロパティ xyz を参照しているかを区別できません。そのため、コンパイラはこれを、インスタンス・メソッド参照としてカウントします。

## 行数

制限：1 ルーチンあたり 65535 行（コメント行を含む）。この制限は、INT 表現のサイズに適用されます。

## リテラル (ASCII)

制限：1 ルーチンまたは 1 クラスあたり 65535 の ASCII リテラル。

ASCII リテラルは、どの文字も \$CHAR(255) を超えない 3 つ以上の文字を含む引用符付き文字列です。

ASCII リテラルと Unicode リテラルは別々に処理され、個別の制限があります。

## リテラル (Unicode)

制限：1 ルーチンまたは 1 クラスあたり 65535 の Unicode リテラル。

Unicode リテラルは、1 つ以上の文字が \$CHAR(255) を超える引用符付き文字列です。

ASCII リテラルと Unicode リテラルは別々に処理され、個別の制限があります。

## パラメータ数

制限：サブルーチン、メソッド、またはスタッド・プロシージャあたり 255 のパラメータ。

## プロシージャ数

制限：1 ルーチンあたり 32767。

## プロパティ読み取り参照

制限：1 ルーチンまたは 1 クラスあたり 32768。

この制限は、以下の例のように、プロパティの値の読み取りに関するものです。

**ObjectScript**

```
Set X = OREF.prop
```

## プロパティ設定参照

制限：1 ルーチンまたは 1 クラスあたり 32768。

この制限は、以下の例のように、プロパティの値の設定に関するものです。

**ObjectScript**

```
Set OREF.prop = value
```



## ルーチン参照

制限：1 ルーチンまたは 1 クラスあたり 65535。

この制限は、ルーチンまたはクラスの一意の参照（routine）の数に適用されます。

## ターゲット参照

制限：1 ルーチンまたは 1 クラスあたり 65535。

ターゲットは label routine（ラベルとルーチンの組み合わせ）です。

ターゲット参照はルーチン参照としてもカウントされます。例えば、以下は 2 つのルーチン参照と 3 つのターゲット参照としてカウントされます。

```
Do Label1^Rtn, Label2^Rtn, Label1^Rtn2
```

## TRY ブロック数

制限：1 ルーチンあたり 65535。

## 変数（プライベート）

制限（ObjectScript）：1 プロシージャあたり 32763。

## 変数（パブリック）

制限（ObjectScript）：1 ルーチンまたは 1 クラスあたり 65503。

変数名やその他の識別子の長さ制限の詳細は、“[識別子のルールとガイドライン](#)”を参照してください。

# B.6 その他のプログラミング制限

以下のテーブルに、コードを記述する際に関連するその他の制限を示します。

## %Status 値の制限

エラー・メッセージの長さ制限：32000 文字未満。

単一の %Status 値に結合できる %Status 値の最大数：150。

## { } による入れ子

制限：32767 レベル。

これは、中括弧を使用するあらゆる言語要素の入れ子の最大の深さです。例：IF { FOR { WHILE {...}}

## 1 行あたりの文字数

制限：1 行あたり 65535 文字。

## 数値

制限（10 進数形式またはネイティブ形式）：約 1.0E-128 ～ 9.22E145。“[インターシステムズ・アプリケーションでの数値の計算](#)”を参照してください。

制限（倍精度形式）：“[インターシステムズ・アプリケーションでの数値の計算](#)”を参照してください。



# C

## システム・マクロ

ここでは、最も一般的なシステム・マクロを使用する方法を説明します。

### C.1 これらのマクロを使用できるようにする方法

ここで説明するマクロは、`%RegisteredObject` のすべてのサブクラスで使用できます。`%RegisteredObject` の拡張でないルーチンまたはクラスでこれらのマクロを使用できるようにするには、以下の該当するファイルをインクルードします。

- ・ 状態関連のマクロの場合は、`%occStatus.inc` をインクルードします。
- ・ メッセージ関連のマクロの場合は、`%occMessages.inc` をインクルードします。

どのインクルード・ファイルが必要になるかについては、以下に示す各マクロを参照してください。

そのような文の構文は以下のとおりです。

#### ObjectScript

```
#include %occStatus
```

これらのインクルード・ファイルでは、大文字と小文字が区別されます。外部で定義されたマクロの使用法の詳細は、“[外部マクロ \(インクルード・ファイル\) の参照](#)” を参照してください。

### C.2 マクロ・リファレンス

マクロ名では大文字と小文字が区別されます。InterSystems IRIS では、以下のマクロが提供されます。

#### ADDSC(sc1, sc2)

ADDSC マクロは、`%Status` コード (sc2) を既存の `%Status` コード (sc1) に付加します。このマクロには `%occStatus.inc` が必要です。

#### EMBEDSC(sc1, sc2)

EMBEDSC マクロは、`%Status` コード (sc2) を既存の `%Status` コード (sc1) 内に埋め込みます。このマクロには `%occStatus.inc` が必要です。

**ERROR(errorcode, arg1, arg2, ...)**

ERROR マクロは、オブジェクト・エラー・コード (errorcode) を使用して、[%Status](#) オブジェクトを作成します。オブジェクト・エラー・コードの関連テキストは、%1、%2 などの形式の引数を数個受け付けます。その後、ERROR はそれらの引数を、errorcode (arg1, arg2 など) に続くマクロ引数に置換します。その際には、この追加引数の順序に基づいて置換されます。このマクロには [%occStatus.inc](#) が必要です。

システム定義のエラー・コードのリストは、“[一般的なエラー・メッセージ](#)” を参照してください。

**FormatMessage(language, domain, id, default, arg1, arg2, ...)**

FormatMessage マクロを使用すると、同じマクロ呼び出し内で、メッセージ・ディクショナリからテキストを取得し、メッセージ引数をテキストに置き換えることができます。[%String](#) を返します。

引数	説明
language	<a href="#">RFC1766</a> 言語コード。Web アプリケーション内では、 <code>%response.Language</code> を指定して既定のロケールを使用できます。
domain	メッセージ・ドメイン。Web アプリケーション内では、 <code>%response.Domain</code> を指定できます。
id	メッセージ ID。
default	language、domain、および id で識別されるメッセージが見つからない場合に使用する文字列。
arg1、arg2 など	メッセージ引数の置換テキスト。これらはすべてオプションなので、メッセージに引数がない場合でも <code>\$\$\$FormatMessage</code> を使用できます。

メッセージ・ディクショナリについては、“[文字列のローカライズとメッセージ・ディクショナリ](#)” を参照してください。

このマクロには [%occMessages.inc](#) が必要です。

[%Library.MessageDictionary](#) の `FormatMessage()` インスタンス・メソッドも参照してください。

**FormatText(text, arg1, arg2, ...)**

FormatText マクロは入力テキスト・メッセージ (text) を受け入れます。このテキスト・メッセージには %1、%2 などの形式で引数を含めることができます。FormatText はそれらの引数を、text 引数 (arg1, arg2 など) に続くマクロ引数に置換します。その際には、この追加引数の順序に基づいて置換されます。その後、結果の文字列を返します。このマクロには [%occMessages.inc](#) が必要です。

**FormatTextHTML(text, arg1, arg2, ...)**

FormatTextHTML マクロは入力テキスト・メッセージ (text) を受け入れます。このテキスト・メッセージには %1、%2 などの形式で引数を含めることができます。FormatTextHTML はそれらの引数を、text 引数 (arg1, arg2 など) に続くマクロ引数に置換します。その際には、この追加引数の順序に基づいて置換されます。またこのマクロは、その結果に HTML エスケープを適用します。その後、結果の文字列を返します。このマクロには [%occMessages.inc](#) が必要です。

**FormatTextJS(text, arg1, arg2, ...)**

FormatTextJS マクロは入力テキスト・メッセージ (text) を受け入れます。このテキスト・メッセージには %1、%2 などの形式で引数を含めることができます。FormatTextJS はそれらの引数を、text 引数 (arg1, arg2 など) に続くマクロ引数に置換します。その際には、この追加引数の順序に基づいて置換されます。またこのマクロは、その結果に JavaScript エスケープを適用します。その後、結果の文字列を返します。このマクロには [%occMessages.inc](#) が必要です。

## GETERRORCODE(sc)

GETERRORCODE マクロは、指定の **%Status** コード (sc) のエラー・コード値を返します。このマクロには **%occStatus.inc** が必要です。

## GETERRORMESSAGE(sc,num)

GETERRORMESSAGE マクロは、指定の **%Status** コード (sc) から、エラー・メッセージ値の、num で指定されている部分を返します。例えば、num=1 は SQLCODE エラー番号を返し、num=2 はエラー・メッセージ・テキストを返します。このマクロには **%occStatus.inc** が必要です。

## ISERR(sc)

ISERR マクロは、指定の **%Status** コード (sc) がエラー・コードの場合に True を返します。それ以外は False を返します。このマクロには **%occStatus.inc** が必要です。

## ISOK(sc)

ISOK マクロは、指定の **%Status** コード (sc) が正常に完了した場合に True を返します。それ以外は False を返します。このマクロには **%occStatus.inc** が必要です。

## OK

OK マクロは、正常終了を表す **%Status** コードを作成します。このマクロには **%occStatus.inc** が必要です。

## Text(text, domain, language)

Text マクロは、ローカライズに使用されます。Text マクロは、コンパイル時に新しいメッセージを生成し、実行時にはそのメッセージを取得するコードを生成します。このマクロには **%occMessages.inc** が必要です。

## TextHTML(text, domain, language)

TextHTML マクロは、ローカライズに使用されます。Text マクロと同じ処理を実行し、その結果に HTML エスケープを適用します。その後、結果の文字列を返します。このマクロには **%occMessages.inc** が必要です。

## TextJS(text, domain, language)

TextJS マクロは、ローカライズに使用されます。Text マクロと同じ処理を実行し、その結果に JavaScript エスケープを適用します。その後、結果の文字列を返します。このマクロには **%occMessages.inc** が必要です。

## ThrowOnError(sc)

ThrowOnError マクロは、指定された **%Status** コード (sc) を評価します。sc がエラー状態を示す場合、ThrowOnError は **THROW** 操作を実行して、**%Exception.StatusException** タイプの例外を例外ハンドラにスローします。このマクロには **%occStatus.inc** が必要です。詳細は、“[TRY-CATCH メカニズム](#)”を参照してください。

## THROWONERROR(sc, expr)

THROWONERROR マクロは式 (expr) を評価します。その式の値は **%Status** コードと見なされ、マクロは sc として渡された変数内に **%Status** コードを格納します。**%Status** がエラーの場合、THROWONERROR は **THROW** 処理を実行し、**%Exception.StatusException** タイプの例外を例外ハンドラへスローします。このマクロには **%occStatus.inc** が必要です。

**ThrowSQLCODE(sqlcode,message)**

ThrowSQLCODE マクロは、指定された SQLCODE とメッセージを使用し、[THROW](#) 処理を実行して、`%Exception.SQL` タイプの例外を例外ハンドラへスローします。このマクロには `%occStatus.inc` が必要です。詳細は、“[TRY-CATCH メカニズム](#)”を参照してください。

**ThrowSQLIfError(sqlcode,message)**

ThrowSQLIfError マクロは、指定された SQLCODE とメッセージを使用し、[THROW](#) 処理を実行して、`%Exception.SQL` タイプの例外を例外ハンドラへスローします。このマクロは、SQLCODE が 0 未満 (エラーを示す負の数字) の場合に、この例外をスローします。このマクロには `%occStatus.inc` が必要です。詳細は、“[TRY-CATCH メカニズム](#)”を参照してください。

**ThrowStatus(sc)**

ThrowStatus マクロは、指定された `%Status` コード (sc) を使用し、[THROW](#) 処理を実行して、`%Exception.StatusException` タイプの例外を例外ハンドラへスローします。このマクロには `%occStatus.inc` が必要です。詳細は、“[TRY-CATCH メカニズム](#)”を参照してください。

# D

## システム・フラグおよびシステム修飾子 (qspec)

クラス・ライブラリの多くのメソッドは qspec 引数を受け入れます。これにより、外部ソースの InterSystems IRIS® データ・プラットフォームへのインポート、コードのコンパイル方法、コードのエクスポートを制御できます。qspec 引数は、このページに記載されているサポートされているシステム・フラグとシステム修飾子を連結したものです。

この両機能は連携して機能します。つまり、qspec はフラグと修飾子の両方を含めることができますが、修飾子の前 (左) にフラグを配置する必要があります。修飾子間のスペースは許可されません。

フラグは UNIX® のコマンド行パラメータでモデル化されており、1 文字または 2 文字のシーケンスで構成されています。修飾子はさらに数が多く、名前も長くなり、それぞれがスラッシュ文字 (/) で始まります。フラグと修飾子は[否定](#)することができます。

多くのフラグには[同等または関連する修飾子](#)があり、この 2 つを同じ qspec で使用できます。[“qspec の処理順序”](#) も参照してください。

### D.1 例

以下の例では、ファイルをインポートする %SYSTEM.OBJ の Load() メソッドで qspec 引数を使用しています。この例で、qspec は c および k フラグを連結したものです。

```
Do $system.OBJ.Load(filename,"ck")
```

または、以下も同じ意味です。

```
Do $system.OBJ.Load(filename,"/compile/keepsources")
```

以下も同等です。

```
Do $system.OBJ.Load(filename,"c/keepsources")
```

これらのフラグと修飾子については、このページで後述します。

### D.2 否定

フラグを否定するには、その前にハイフン (-) を付けます。

修飾子を否定するには、/ ではなく /no を使用します (例 : /nodisplaylog)。または、修飾子の末尾に =0 を付けます (例 : /displaylog=0)。

## D.3 フラグ

フラグ	意味	既定値
b	サブクラスおよび SQL 使用で現在のクラスを参照しているクラスを含めます。	
c	ロード後、クラス定義をコンパイルします。	
d	表示。既定で設定されるフラグです。	X
e	エクステントによって使用されるグローバル・ストレージを記述するエクステント定義を削除し、データを削除します。	
h	非表示クラスを表示します。	
i	ロード時、XML エクスポート形式をスキーマに対して検証します。既定で設定されるフラグです。	X
k	ソースを保持します。このフラグを設定すると、生成されたルーチンのソース・コードが保持されます。	
l	非推奨 – コンパイル時のクラスのロックは、このフラグの設定に関係なく常に自動的に実行されます。	X
p	名前の先頭の文字が “%” であるクラスを追加します。	
r	処理を再帰的に実行します。先行依存するすべてのクラスをコンパイルします。	
s	システム。システム・メッセージまたはアプリケーション・メッセージを処理します。	
u	アップデートのみ実行します。最新状態であればクラスのコンパイルをスキップします。	
y	現在のクラスと関連するクラス (SQL 使用下での現在のクラスを参照しているクラスまたは現在のクラスから参照されるクラス) を含めます。	
o1、o2、o3、o4	最適化指定子。非推奨となっており、クラス・コンパイラでは無視されます。	

## D.4 コンパイラ修飾子

修飾子	意味	既定値
/autoinclude	このクラスのコンパイルに必要な、最新でないすべてのクラスを自動的に含めます。	1
/checkschema	インポートした XML ファイルをスキーマ定義に対して検証します。	1
/checkstoragedefined	クラスに、すべてのプロパティに対して定義されたストレージがあることを確認します。1 に設定されている場合、この修飾子は、コンパイル時にストレージ定義が変更されたらそれを知らせます。	0
/checksysutd	システム・クラスが最新であるかどうかを確認します。	0



修飾子	意味	既定値
/checkuptodate	最新のクラスまたは最新の拡張クラスのコンパイルをスキップします。	expandedonly
/compile	ロードされたクラスをコンパイルします。	0
/compileembedded	<a href="#">埋め込み SQL</a> は、これを含む ObjectScript コードのコンパイル時にコンパイルされます。既定では、埋め込み SQL は、SQL コードの初回実行時にコンパイルされます。	0
/cspcompileclass	CSP または CSR のロードによって作成されたクラスをコンパイルします。	1
/cspdeployclass	CSP ページがロードされたら生成したクラスを配置します。	0
/csphidden	CSP および CSR のコンパイルで生成されたクラスを非表示としてマーク付けします。	1
/defaultowner	クラスのロード時、 <a href="#">Owner</a> キーワードが定義されていない場合、この文字列で指定されているユーザ名をクラス所有者としてクラス定義に挿入します。この文字列の値が <a href="#">\$USERNAME</a> である場合、現在のユーザ名をクラス所有者としてクラス定義に挿入します。	—
/defines	定義するマクロのコンマ区切りリスト。必要に応じてそれらの値も指定します。	—
/deleteextent	エクステントによって使用されるグローバル・ストレージを記述するエクステント定義を削除し、データを削除します。	0
/diffexport	diff/merge ツールで実行できるよう、ファイルのエクスポートに時間やプラットフォーム情報を含めません。	0
/display	/displaylog および /displayerror のエイリアス修飾子。	—
/displayerror	エラー情報を表示します。	1
/displaylog	ログ情報を表示します。	1
/expand	/predecessorclasses、/subclasses、および /relatedclasses のエイリアス修飾子。	—
/exportgenerated	クラスをエクスポートするとき、生成したクラスもエクスポートし、そのクラスの生成元となったクラスも追加します。	0
/exportselectivity	このクラスのストレージ定義内に格納されている SELECTIVITY 値をエクスポートします。	1
/filterin	/application、/system、および /percent のエイリアス修飾子。	—
/generated	生成される項目（ルーチン、クラスなど）がパッケージ内の展開中のパターンまたはクラスのリストに含まれるかどうかを決定します。	1
/generatemap	マップ・ファイルを生成します。	1

修飾子	意味	既定値
/importselectivity	0 : XML ファイルから SELECTIVITY 値をインポートしません。1 : XML ファイルをインポートするときに、ストレージ定義に格納されている SELECTIVITY 値をインポートします。2 : 既存のクラスの SELECTIVITY 値を維持しますが、既存のクラスに、XML ファイルの内容に対して指定された SELECTIVITY がいない場合は、XML ファイルの SELECTIVITY 値を使用します。	2
/includesubpackages	サブ・パッケージを含めます。	1
/journal	クラス・コンパイル実行中にジャーナリングを有効にします。特に、コンパイルを実行しているプロセスでジャーナリングが無効になっている場合、/journal は既定値 0 になります。システム全体の既定値 1 にはなりません。	1
/keepsources	生成されたルーチンのソース・コードを保持します。	0
/lock	非推奨 – この修飾子の設定に関係なく、クラスはコンパイル時には常に自動的にロックされます。	1
/mapped	別のデータベースからマップされたクラスを含めます。別のデータベースからクラスをコンパイルすることを明確に指定すると (CompileList() メソッド)、/mapped の設定にかかわらず、そのクラスがコンパイルされます。/mapped は、コードがクラスを検索する場合 (例えば、CompileAll() メソッドを使用する場合) にのみ適用されます。Upgrade() メソッドを使用して 1 つのネームスペースのクラス定義データベースをアップグレードする場合、または UpgradeAll() メソッドを使用してすべてのネームスペースのクラス定義データベースをアップグレードする場合、/mapped = 1 に設定する必要があります。そうしないと、マップされたオブジェクトはアップグレードに含められません。	0
/mergeglobal	XML ファイルからグローバルをインポートする場合は、そのグローバルを既存のデータとマージします。	0
/multicompile	複数のユーザのジョブを有効にして、クラスをコンパイルします。	1
/percent	パーセント・クラスを含めます。	0
/predecessorclasses	先行依存するクラスを再帰的に含めます。	0
/relatedclasses	関連するクラスを再帰的に含めます。	0
/retainstorage	クラスのコンパイル時、コンパイラによってストレージ定義が生成されます。既定では、ストレージ定義が更新された場合、クラス定義は更新されたストレージ定義で更新されます。新しいバージョンのクラスが外部ソースからロードされた場合、その更新されたストレージ定義は新しいバージョンのクラス定義で定義されている内容に上書きされます。新しいバージョンのクラスにストレージ定義が含まれていない場合、既存のストレージ定義は削除されます。/retainstorage を設定すると、既存のストレージ定義が一時的に保存され、新しいバージョンのクラスのロード後にリストアされます。新しいバージョンのクラスでもストレージ定義が定義されている場合、既存のストレージ定義は上書きされ、保持されません。新しいバージョンのクラスでストレージ定義が定義されていない場合、前のバージョンのストレージ定義がリストアされます。	0

修飾子	意味	既定値
/subclasses	サブ・クラスを再帰的に含めます。	0
/system	システム・メッセージまたはアプリケーション・メッセージを処理します。	0

## D.5 エクスポート修飾子

フラグ	意味	既定値
/checksysutd	システム・クラスが最新であるかどうかを確認します。	0
/checkuptodate	投影の際にクラスが最新であるかどうかを確認します。	expandedonly
/createdirs	存在しない場合はディレクトリを作成します。	0
/cspdeployclass	CSP ページがロードされたら生成したクラスを配置します。	0
/diffexport	diff/merge ツールで実行できるよう、ファイルのエクスポートに時間やプラットフォーム情報を含めません。	0
/display	/displaylog および /displayerror のエイリアス修飾子。	—
/displayerror	エラー情報を表示します。	1
/displaylog	ログ情報を表示します。	1
/documatichost	JavaDoc の生成に使用されるホスト。	—
/documaticnamespace	JavaDoc の生成に使用されるネームスペース。	—
/documaticport	JavaDoc の生成に使用されるポート。	—
/exportgenerated	クラスをエクスポートするとき、生成したクラスもエクスポートし、そのクラスの生成元となったクラスも追加します。	0
/exportselectivity	このクラスのストレージ定義内に格納されている SELECTIVITY 値をエクスポートします。	1
/exportversion	エクスポート先のシステムの InterSystems プラットフォームとバージョンを指定します。プラットフォームは iris または cache として指定します。バージョンは、2 つの部分または 3 つの部分で構成されるリリース・バージョン (2020.1 または 2020.1.1 など) で指定します。例えば、/exportversion=iris2020.1.1 または /exportversion=cache2018.1.8 です。エクスポート・システムとインポート・システムが同じ InterSystems バージョンでない場合、IRIS は /exportversion 値を使用します。以前の InterSystems バージョンでは実装されていなかったクラス・キーワードを削除することで、バージョン間でのエクスポート形式の変更を処理します。/exportversion を指定しても、エクスポート・システムとインポート・システム間のコードの互換性は保証されません。	InterSystems IRIS の現在の バージョン
/generatemap	マップ・ファイルを生成します。	1
/generationtype	生成モード。	—

フラグ	意味	既定値
/genserialuid	serialVersionUID を生成します。	1
/importselectivity	0 : XML ファイルから SELECTIVITY 値をインポートしません。 1 : XML ファイルをインポートするときに、ストレージ定義に格納されている SELECTIVITY 値をインポートします。 2 : 既存のあらゆる SELECTIVITY 値を維持しますが、既存の値がないプロパティには XML ファイルの SELECTIVITY 値を使用します。	2
/includesubpackages	サブ・パッケージを含めます。	1
/javadoc	javadoc を作成しません。	1
/make	最新のコンパイルのタイムスタンプが、最新の生成時のタイムスタンプよりも新しい場合は、依存またはクラスのみを生成します。	0
/mapped	別のデータベースからマップされたクラスを含めます。	0
/mergegloabl	XML ファイルからグローバルをインポートする場合は、そのグローバルを既存のデータとマージします。	0
/newcollections	ネイティブの Java コレクションを使用します。	1
/percent	パーセント・クラスを含めます。	0
/pojo	POJO 生成モードを指定します。	0
/predecessorclasses	先行依存するクラスを再帰的に含めます。	0
/primitivedatatypes	%Integer、%Boolean、%BigInt、%Decimal に Java 基本関数を使用します。	0
/projectabstractstream	引数が抽象ストリームであるか、返りタイプが抽象ストリームであるメソッドを含むプロジェクト・クラスを指定します。	0
/projectbyrefmethodstopojo	pojo 実装に ByRef メソッドを投影します。	0
/recursive	クラスを再帰的にエクスポートします。	1
/relatedclasses	関連するクラスを再帰的に含めます。	0
/skipstorage	クラスのストレージ情報をエクスポートしません。	0
/subclasses	サブ・クラスを再帰的に含めます。	0
/system	システム・メッセージまたはアプリケーション・メッセージを処理します。	0
/unconditionallyproject	コードのコンパイルまたは正常な動作を妨げる問題を無視して投影を実行します。	0
/usedeepbase	メソッドまたはプロパティ定義に対してメソッドまたはプロパティが定義されている場合に、最も下部にあるベースを使用します。P が A、B、および C に定義されており、A が B を拡張し、B が C を拡張する場合、より深い位置にあるベースは C になります。	0

## D.6 ShowClassAndObject 修飾子

フラグ	意味	既定値
/detail	詳細情報を表示します。	0
/diffexport	diff/merge ツールで実行できるよう、ファイルのエクスポートに時間やプラットフォーム情報を含めません。	0
/hidden	非表示クラスを表示します。	0
/system	システム・メッセージまたはアプリケーション・メッセージを処理します。	0

## D.7 UnitTest 修飾子

フラグ	意味	既定値
/autoload	ディレクトリを自動ロードすることを指定します。そのサブディレクトリも自動ロードされます。詳細は、“%UnitTest.Manager” の “RunTest()” メソッドを参照してください。	—
/cleanup	ユニット・テスト完了時にグローバルを削除します。既定では、グローバルは削除されません。設定されている場合でも、分析グローバルは削除されません。	0
/debug	アサートに失敗した場合、アサートをブレイク (BREAK) 状態にします。	0
/delete	ロードしたクラスを削除するかどうかを判断します。	1
/display	/displaylog および /displayerror のエイリアス修飾子。	—
/displayerror	エラー情報を表示します。	1
/displaylog	ログ情報を表示します。	1
/findleakedvariables	有効にすると、テスト実施前にプロセスで現在設定されているパブリック変数が記録され、テスト完了後にそれらの設定と比較されます。SQLCODE などの事前設定された一連の既知のコンテキストおよび出力変数以外に、新たに定義された変数がそれらの値とともにテストの失敗として報告されます。	0
/load	クラスをロードするかどうかを判断します。ロードしない場合は、ディレクトリからクラス名のみが取得されます。	1
/loadudl	IDE によって作成された UDL ファイルをロードします。設定すると、.cls、.mac、.int、および .inc ファイルがロードされます。/loadudl と /loadxml を使用することで、ロードされるファイルの種類を制限できます。既定では、すべてのファイルがロードされます。UDL ファイルは常に、Unicode 文字が正しくロードされるように UTF8 としてロードされます。	1

フラグ	意味	既定値
/loadxml	XML 形式のソース・ファイルをロードします。設定すると、.xml ファイルがロードされます。/loadudl と /loadxml を使用することで、ロードされるファイルの種類を制限できます。既定では、すべてのファイルがロードされます。	1
/recursive	サブディレクトリでのテストを再帰的に実行するかどうかを判断します。	1
/run	テストを実行するかどうかを判断します。	1

## D.8 フラグに対応する修飾子

以下のテーブルに、既存のフラグとそれに対応する修飾子を示します。フラグによっては複数の修飾子にマップされるものがあり、その意味も使用目的に応じて異なります。

フラグ	グループ	修飾子	既定値
b	コンパイラ	/subclasses	0
c	コンパイラ	/compile	0
d	コンパイラ	/displayerror	1
d	コンパイラ	/displaylog	1
d	UnitTest	/displayerror	1
d	UnitTest	/displaylog	1
e	コンパイラ	/deleteextent	0
i	コンパイラ	/checkschema	1
k	コンパイラ	/keepsources	0
l	コンパイラ	/lock	1
p	コンパイラ	/percent	0
r	コンパイラ	/predecessorclasses	0
r	コンパイラ	/includesubpackages	1
s	コンパイラ	/system	0
y	コンパイラ	/relatedclasses	0
b	Export	/subclasses	0
d	Export	/displayerror	1
d	Export	/displaylog	1
g	Export	/exportselectivity	0
p	Export	/percent	0
r	Export	/includesubpackages	1
r	Export	/recursive	1

フラグ	グループ	修飾子	既定値
r	Export	/predecessorclasses	0
s	Export	/system	0
y	Export	/relatedclasses	0
h	ShowClassAndObject	/hidden	0
s	ShowClassAndObject	/system	0

## D.9 フラグおよび修飾子のヘルプ

フラグで使用可能な設定を確認するには、以下のコマンドを使用します。

### ObjectScript

```
Do $system.OBJ.ShowFlags()
```

これは、以下のような出力を生成します。

```
See $system.OBJ.ShowQualifiers() for comprehensive list of qualifiers as flags have been superseded by qualifiers
```

```

b - Include sub classes.
c - Compile. Compile the class definition(s) after loading.
d - Display. This flag is set by default.
```

```
...
Default flags for this namespace
You may change the default flags with the SetFlags(flags,system) classmethod.
```

同様に、修飾子で使用可能な設定を確認するには、以下のコマンドを使用します。

### ObjectScript

```
Do $system.OBJ.ShowQualifiers()
```

これは、以下のような出力を生成します。

```

      Name: /checkschema
Description: Validate imported XML files against the schema definition.
      Type: logical
      Flag: i
Default Value: 1
```

```

      Name: /checksysutd
Description: Check system classes for up-to-dateness
      Type: logical
Default Value: 0
```

```

      Name: /checkuptodate
Description: Skip classes or expanded classes that are up-to-date.
      Type: enum
      Flag: ll
      Enum List: none,all,expandedonly,0,1
Default Value: expandedonly
Present Value: all
Negated Value: none
```

```
...
```

これらのメソッドは、それぞれ現在のフラグと修飾子についての報告も行います。

## D.10 既定値の制御

ここには、フラグと修飾子の既定値 (該当する場合) をリストします。これらの既定値は、`%SYSTEM.OBJ` の `SetFlags()` メソッドを使用してオーバーライドできます。同様に、`SetQualifiers()` メソッドを使用して、現在のネームスペース (既定) またはシステム全体に対して修飾子を設定できます。

## D.11 qspec の処理順序

qspec は左から右に処理されます。特定のフラグまたは修飾子を設定すると、現在の設定が、使用している環境の既定値であるか、qspec 内のそれ以前のオカレンスであるかに関係なく、オーバーライドされます。

フラグは修飾子の左にリストする必要があるため、修飾子設定は常にフラグ設定より優先されることに注意してください。



# E

## 正規表現

InterSystems IRIS® データ・プラットフォームでは、ObjectScript 関数の [\\$LOCATE](#) と [\\$MATCH](#)、および `%Regex.Matcher` クラスのメソッドで使用する正規表現がサポートされています。

その他すべての部分文字列マッチング処理では、ObjectScript [パターン・マッチング](#) 演算子を使用します。

正規表現の InterSystems IRIS 実装は、正規表現の International Components for Unicode (ICU) 標準に基づきます。Perl 正規表現に精通したユーザは、InterSystems IRIS 実装に多くの類似点を見い出します。

### E.1 ワイルドカードと修飾子

ワイルドカード。行スペース文字の `$CHAR(10)`、`$CHAR(11)`、`$CHAR(12)`、`$CHAR(13)`、および `$CHAR(133)` を除いて、任意のタイプの任意の 1 文字と一致します。単一行モード (`?s`) を指定すると、行スペース文字の除外をオーバーライドできます (このリファレンス・ページで後述)。

単独で使用できます。`..` は、任意の 2 文字を表すために使用できます。また、`\d..` という組み合わせで、任意の 2 文字が続く 1 つの数字を表すために使用できます。

接尾語との組み合わせができますが、同じ行スペース文字制限があります。

- ・ `.?` は、任意のタイプの 0 文字か 1 文字と一致します。
- ・ `.*` は、任意のタイプの 0 文字以上の文字と一致します。
- ・ `.+` は、任意のタイプの 1 文字以上の文字と一致します。
- ・ `{3}` は、任意のタイプの 3 文字と一致します。

ワイルドカードのシーケンスを終了するには、円記号 (`¥`) 接頭語を使用して次のリテラルをエスケープ処理します。例えば、`regex ".*\H\d{2}"` は、英字の `H` の後に 2 桁の数字で終わる、任意のタイプの任意の文字の文字列と一致します。

?

単一文字接尾語 (0 または 1)。`regex` を 1 回または 0 回 `string` に適用します。正規表現の `\d?`、`[0-9]?`、または `[[[:digit:]]?` はすべて、単一の数字か空の文字列に一致します。正規表現の `.(?log)` は `blog` (1 回の出現) または `log` (0 回の出現) に一致します。正規表現 `abc?` は `abc` または `ab` のいずれかに一致できます。

+

反復接尾語 (1 回以上)。regexp を 1 回以上 string に適用します。例えば、`A+` は文字列の `AAAAA` に一致します。`.+` は、任意の文字タイプの任意の長さの文字列に一致しますが、空の文字列には一致しません。正規表現の `\d+`、`[0-9]+`、または `[[[:digit:]]+]` はすべて、任意の長さの数字の文字列に一致します。

括弧を複雑な反復パターンで使用できます。例えば、`(AB)+` は文字列の `ABABABAB` に一致します。`(\d\d\d\s)+` は、3 桁の数字と単一のスペース文字が交互に続く任意の長さのシーケンスに一致します。

\*

反復接尾語 (0 回以上)。regexp を 0 回以上 string に適用します。例えば、`A*` は文字列の `A`、`AAAAA`、および空の文字列に一致します。`.*` は、空の文字列を含めて、任意の文字タイプの任意の長さの文字列に一致します。正規表現の `\d*`、`[0-9]*`、または `[[[:digit:]]*]` はすべて、空の文字列または任意の長さの数字の文字列に一致します。

括弧を複雑な反復パターンで使用できます。例えば、`(AB)*` は文字列の `ABABABAB` に一致します。`(\d\d\d\s)*` は、3 桁の数字と単一のスペース文字が交互に続く任意の長さのシーケンスに一致します。

{n}

数量化接尾語 (n 回)。`{n}` 接尾語により regexp を正確に n 回適用します。例えば、`\d{5}` は、5 桁の数字に一致します。

{n,}

数量化接尾語 (n 回以上)。`{n,}` 接尾語により regexp を n 回以上適用します。例えば、`\d{5,}` は、5 桁以上の数字に一致します。

{n,m}

数量化接尾語 (範囲)。`{n,m}` 接尾語により regexp を n 回以上、m 回以下適用します。例えば `\d{7,10}` は、7 桁以上で 10 桁以下の数字に一致します。

## E.2 リテラルと文字の範囲

大半のリテラル文字は単純に正規表現に含めることができます。例えば、正規表現の `".*G.*"` は、文字列に英字の `G` が含まれている必要があることを指定します。

また、一部のリテラル文字は正規表現のメタ文字としても使用されます。リテラル文字として扱うメタ文字の前には、エスケープ接頭語 (円記号文字) を使用する必要があります。ドル記号 `\$`、アスタリスク `\*`、プラス記号 `\+`、ピリオド `\.`、疑問符 `\?`、円記号 `\\`、キャレット `\^`、垂直バー `\|`、開始括弧と終了括弧 `\( \)`、開始角括弧と終了角括弧 `\[ \]`、および開始中括弧と終了中括弧 `\{ \}` のリテラル文字では、エスケープ接頭語が必要です。終了角括弧の `]` では、常にエスケープ接頭語が必要なわけではありません。エスケープ接頭語は、明確にしたり整合性のために使用する必要があります。

引用符文字ではエスケープ接頭語を使用しません。リテラル引用符文字を使用するには、二重引用符 `"` にします。

以下に、リテラルの複数の正規表現一致を指定する方法を示します。

## [x]

指定文字または文字のリスト。したがって [A] は、英字の大文字の A のみに一致することを意味します。[ACE] は、英字の A、C、または E のいずれか 1 つと一致します。文字は任意の順序で記述できます。繰り返し文字が可能です。キャレット (^) を使用すると、否定を指定できます。例えば、[A] は、英字の大文字の A を除いた任意の文字に一致することを意味します。[XYZ] は、X、Y、または Z を除いた任意の文字に一致することを意味します。既定では、これらの文字の一致では大文字と小文字が区別されます。(?)i モード修飾子を前に付加すると、文字の一致で大文字と小文字が区別されないようにできます。

キャレット (^) をリテラル一致文字として指定するために、リストの最初の文字にすることはできません。ハイフン (\$CHAR(45)) をリテラル一致文字として指定するには、リストで最初か最後の文字にする必要があります。終了角括弧 (]) をリテラル一致文字として指定するには、リストで最初の文字にする必要があります。(最初の文字は、^ 否定演算子後の最初の文字を意味する場合があります。)円記号エスケープ接頭語リテラルも使用できます。例えば、[¥¥AB¥[CD] は、円記号 (¥)、開始角括弧 ([、および英字の A、B、C、および D に一致します。

## [x-z]

x で始まり、z で終わる指定文字の範囲。一般的には英字や数字に使用されますが、昇順の任意の ASCII シーケンスを範囲として使用することができます。したがって、[A-Z] はすべての大文字の範囲です。[A-z] は、英字の大文字と小文字だけでなく、英字における 6 つの ASCII 句読点文字も含まれる範囲です。昇順の ASCII シーケンスでない範囲を指定すると、<REGULAR EXPRESSION> エラーが発生します。また、複数の範囲を指定することもできます。したがって、[A-Za-z] はすべての大文字と小文字の範囲です。開始括弧に続く最初の文字としてキャレット (^) を使用すると、否定を指定できます。例えば、[A-F] は、A ~ F の文字以外の文字すべてに一致します。キャレットは、指定されているすべての範囲の否定を指定するので、[A-Za-z] は英文字を除くすべての文字に一致します。文字の範囲と単一文字のリストは、任意のシーケンスで組み合わせられます。したがって、[ABCa-fXYZ0-9] は、指定文字と指定範囲内の文字に一致します。

## (str) または (str1|str2)

OR 論理演算子 (|) で区切られた指定された文字列または文字列のリスト。したがって、(William) は string 内のこの部分文字列に一致し、(William|Willy|Wm¥.|Bill) はこれらの部分文字列のいずれかに一致します。エスケープ接頭語の ¥ を使用すると、垂直バーを文字列内のリテラルとして指定できます。既定では、これらの部分文字列の一致では大文字と小文字が区別されます。(?)i モード修飾子を前に付加すると、部分文字列の一致で大文字と小文字が区別されないようにできます。既定では、これらの部分文字列一致は string 内の任意の場所で可能です。¥b を前に付加すると、部分文字列の一致が単語境界で発生するように制限できます。

## E.3 文字タイプ・メタ文字

InterSystems IRIS の正規表現では、3 セットの文字タイプ・メタ文字がサポートされています。

- ・ 単一文字タイプ。例えば、¥d のようになります。
- ・ Unicode プロパティ文字タイプ。例えば、¥p{LL} のようになります。
- ・ POSIX 文字タイプ。例えば、[:alpha:] のようになります。

これらの文字タイプ・メタ文字は、任意の正規表現で任意に組み合わせで使用できます。

### E.3.1 単一文字タイプ

単一文字タイプ・メタ文字は、円記号 (¥) とその後の英字で示します。文字タイプは小文字で指定されます (¥d = 数字 : 0 ~ 9)。これらの文字タイプで否定をサポートしている場合、大文字により文字タイプの否定を指定します (¥D = 数字を除いた任意の文字)。

¥a

ベル文字 \$CHAR(7)。否定はサポートされていません。

¥d

数字。0 ～ 9。否定は ¥D です。

¥e

エスケープ文字 \$CHAR(27)。否定はサポートされていません。

¥f

書式送り文字 \$CHAR(12)。否定はサポートされていません。

¥n

改行文字 \$CHAR(10)。否定はサポートされていません。

¥r

キャリッジ・リターン文字 \$CHAR(13)。否定はサポートされていません。

¥s

スペース文字。\$CHAR(9)、\$CHAR(10)、\$CHAR(11)、\$CHAR(12)、\$CHAR(13)、\$CHAR(32)、\$CHAR(133)、および \$CHAR(160) を含めた、空白、タブ、または行スペースの文字。否定は ¥S です。

¥t

タブ文字 \$CHAR(9)。否定はサポートされていません。

¥w

単語構成文字。単語構成文字は、文字、数字、またはアンダースコア文字を使用できます。有効な文字には、Unicode 文字を含めて英字の大文字と小文字が含まれます。これらには、\$CHAR(170)、\$CHAR(181)、\$CHAR(186)、\$CHAR(192) ～ \$CHAR(214)、\$CHAR(216) ～ \$CHAR(246)、\$CHAR(248) ～ \$CHAR(256) の拡張 ASCII 文字が含まれます。否定は ¥W です。

¥d、¥s、および ¥w のメタ文字は、\$CHAR(256) よりも後の適切な Unicode 文字にも一致します。

その他の個々の制御文字のメタ文字シーケンスについては、“[制御文字表現](#)”を参照してください。

## E.3.2 Unicode プロパティ文字タイプ

Unicode プロパティ文字タイプの一致では、単一文字が以下の構文を使用して指定された文字タイプに一致します。

`\p{prop}`

例えば、¥p{LL} は、英字の小文字に一致します。prop キーワードは、1 文字または 2 文字の文字で構成されます。prop キーワードでは、大文字と小文字は区別されません。単一文字の prop キーワードは、最も包含的です。2 文字の prop キーワードはサブセットを指定します。

否定は ¥P{prop} です。例えば、¥P{LL} は、小文字でない任意の文字に一致します。

以下のリストは、最初の 256 文字で各 prop キーワードに一致する文字を示しています (256 文字のいずれにも一致しない prop キーワードには、サンプルの Unicode 文字が示されています)。

## C

制御文字とその他の文字 (0-31、127-159、173)

## CC

制御文字 (0-31、127-159)

## CF

書式設定文字 (173)

## CN

割り当てられていないコード・ポイント (例えば、888)

## CO

非公開使用文字 (例えば、57344)

## CS

サロゲート(例えば、55296)

## L

英字 (65-90、97-122、170、181、186、192-214、216-246、248-255)

## LL

英字の小文字 (97-122、170、181、186、223-246、248-255)

## LM

修飾子文字 (例えば、688)

## LO

LL、LU、LT、または LM 以外の文字 (例えば、443)

## LT

タイトル文字 (例えば、453)

## LU

英字の大文字 (65-90、192-214、216-222)

## M

記号 (例えば、768)

## MC

修飾文字 (例えば、2307)

## ME

囲み記号 (例えば、1160)

**MN**

アクセント記号 (例えば、768)

**N**

数字 (48-57、178-179、185、188-190)

**ND**

10 進数 (48-57)

**NL**

数値を表す文字 (例えば、5870)

**NO**

数字の添え字と小数 (178-179、185、188-190)

**P**

句読点 (33-35、37-42、44-47、58-59、63-64、91-93、95、123、125、161、171、183、187、191)

**PC**

接続句読点 (95)

**PD**

ダッシュ (45)

**PE**

終了句読点 (41、93、125)

**PS**

開始句読点 (40、91、123)

**PI**

最初の句読点 (171)

**PF**

最後の句読点 (187)

**PO**

その他の句読点 (33-35、37-39、42、44、46-47、58-59、63-64、92、161、183、191)

**S**

記号 (36、43、60-62、94、96、124、126、162-169、172、174-177、180、182、184、215、247)

**SC**

通貨記号 (36、162-165)

**SK**

組み合わせ記号 (94、96、168、175、180、184)

**SM**

算術記号 (43、60–62、124、126、172、177、215、247)

**SO**

その他の記号 (166–167、169、174、176、182)

**Z**

区切り文字 (32、160)

**ZL**

行区切り文字 (例えば、8232)

**ZP**

パラグラフ区切り文字 (例えば、8233)

**ZS**

空白文字 (32、160)

以下のコードを使用すると、prop キーワードで一致する文字を判別できます。

**ObjectScript**

```
READ prop#2:10
READ rangefrom:10
READ rangeto:10
FOR i=rangefrom:1:rangeto {
  IF $MATCH($CHAR(i),"\p{ "_prop_" }")=1 {
    WRITE i,"=", $CHAR(i),! } }
```

## E.3.3 POSIX 文字タイプ

POSIX 構文では、単一文字が以下の構文形式のいずれかを使用して ptype キーワードで指定された文字タイプに一致します。

```
\p{ptype}
[:ptype:]
```

例えば、[:lower:] や \p{lower} は、英字の小文字に一致します。[:^lower:] や \P{lower} のようにすると否定 (英字の小文字以外のすべてに一致) を指定できます。

ptype キーワードでは、大文字と小文字は区別されません。一般的な ptype キーワードは、以下のとおりです。

- ・ alnum – 文字と数字。
- ・ alpha – 文字。
- ・ blank – タブ (\$CHAR(9)) またはスペース (CHAR(32)、CHAR(160))。
- ・ cntrl – 制御文字 (\$CHAR(0) ~ \$CHAR(31)、\$CHAR(127) ~ \$CHAR(159))。
- ・ digit – 数字 (0 ~ 9)。
- ・ graph – スペース文字を除く出力可能文字 (\$CHAR(33) ~ \$CHAR(126)、\$CHAR(161) ~ \$CHAR(156))。

- `lower` – 英字の小文字。
- `math` – 算術文字 (記号のサブセット)。+<=>^|~¬±×÷ の文字が含まれます。
- `print` – スペース文字を含む出力可能文字 (\$CHAR(32) ~ \$CHAR(126), \$CHAR(160) ~ \$CHAR(156))。
- `punct` – 句読点文字 (記号文字を除きます)。!"#\$%&'()\*,-./:;?@[\\]\_{ } ¡ ¢ £ ¤ の文字が含まれます。
- `space` – \$CHAR(9), \$CHAR(10), \$CHAR(11), \$CHAR(12), \$CHAR(13), \$CHAR(32), \$CHAR(133), および \$CHAR(160) の文字を含む、空白、タブ、および行スペースの文字を含めた、スペース文字。
- `symbol` – 記号文字 (句読点文字を除きます)。\$+<=>^`|~¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸ の文字が含まれます。
- `upper` – 英字の大文字。
- `xdigit` – 16 進数。0 ~ 9 の数字、A ~ F の大文字、および a ~ f の小文字です。

さらに、`ptype` を使用すると、Unicode カテゴリを指定できます。例えば、`[:greek:]` は、Unicode ギリシャ語カテゴリ (\$CHAR(900) ~ \$CHAR(974)) の範囲にあるギリシャ文字がこれに含まれます) の文字に一致します。これらの POSIX Unicode カテゴリの部分リストには、`[:arabic:]`、`[:cyrillic:]`、`[:greek:]`、`[:hebrew:]`、`[:hiragana:]`、`[:katakana:]`、`[:latin:]`、`[:thai:]` が含まれます。これらの Unicode カテゴリは、例えば `[:script=greek:]` としても表記できます。

以下の例では、POSIX マッチングを使用して、`[:letter:]` 文字セットと `[:latin:]` 文字セットを最初の 256 文字で比較します。それらは、文字が 1 つ (\$CHAR(181)) 異なります。

#### ObjectScript

```
FOR i=0:1:255 {
  SET letr="foo"
  IF 1=$MATCH($CHAR(i), "[:letter:]") {
    SET letr=$CHAR(i)}
  IF 1=$MATCH($CHAR(i), "[:latin:]") {
    SET lat=$CHAR(i)}
  ELSE {SET lat="foo"}
  IF letr != lat {WRITE i, " ", $CHAR(i), !}
}
```

## E.4 グループ化構文

括弧を使用すると、繰り返し適用されるリテラルやメタ文字シーケンスを指定できます。例えば、正規表現 `([0-9])+` では、文字列内の各連続文字が数字かどうかテストします。

この使用法は、以下の例を参照してください。

#### ObjectScript

```
WRITE $MATCH("4567683285759", "([0-9])+"), !
// test for all numbers, no empty string
WRITE $MATCH("4567683285759", "([0-9])*"), !
// test for all numbers or for empty string
WRITE $MATCH("Now is the time", "\p{LU}(\p{L}|\s)+"), !
// test for initial uppercase letter, then all letters or spaces
WRITE $MATCH("MABoston-9a", "\p{LU}{2}(\p{LL}|\d|\\-)*"), !
// test for 2 uppercase letters, then all lowercase, numbers, dashes, or ""
WRITE $MATCH("1^23^456^789", "([0-9]+\\^?)+"), !
// test for one or more numbers followed by 0 or 1 ^ characters, apply test repeatedly
WRITE $MATCH("$1,234,567,890.99", "\\$([0-9]+,?)+\\.\\d\\d")
// test for $, then numbers followed by 0 or 1 comma, then decimal point, then 2 fractional digits
```

注釈 グループ化構文は正規表現を繰り返して適用するので、所要時間の長いマッチング処理を作成できます。



以下の例は注意が必要で、文字列のパターン一致エラーの位置に応じて、繰り返して適用されるグループ化構文の実行時間が急激に長くなります。不一致を宣言する前にテストが必要な組み合わせの数が増加すると、実行時間が長くなります。

## ObjectScript

```
SET a=$ZHOROLOG
WRITE $MATCH("1111111111,222222222,333333333","([0-9]+,?)+")
SET b=$ZHOROLOG-a
WRITE " duration: ",b,!
SET a=$ZHOROLOG
WRITE $MATCH("11111x11111,222222222,333333333","([0-9]+,?)+")
SET b=$ZHOROLOG-a
WRITE " duration: ",b,!
SET a=$ZHOROLOG
WRITE $MATCH("111111111,22x2222222,333333333","([0-9]+,?)+")
SET b=$ZHOROLOG-a
WRITE " duration: ",b,!
SET a=$ZHOROLOG
WRITE $MATCH("111111111,2222222x22,333333333","([0-9]+,?)+")
SET b=$ZHOROLOG-a
WRITE " duration: ",b,!
SET a=$ZHOROLOG
WRITE $MATCH("111111111,2222222x22,333333333","([0-9]+,?)+")
SET b=$ZHOROLOG-a
WRITE " duration: ",b
```

## E.5 アンカー・メタ文字

アンカーとは、関連する正規表現一致を一致文字列内の特定の場所に制限するメタ文字です。例えば、一致する場所を、文字列の先頭や最後、または文字列内のスペース文字の後のみにすることができます。

### E.5.1 文字列の先頭または最後

これらのアンカーにより、文字列の先頭や最後に一致が制限されます。

#### ^ または ¥A

文字列の先頭のアンカー接頭語。正規表現の一致が文字列の先頭で出現する必要があることを示します。

#### \$

文字列の最後のアンカー接尾語。正規表現の一致が文字列の最後で出現する必要があることを示します。行の最後の文字 (ASCII 10、11、12、または 13) は無視されます。¥Z と同じです。

#### ¥Z

文字列の最後のアンカー接尾語。正規表現の一致が文字列の最後で出現する必要があることを示します。行の最後の文字 (ASCII 10、11、12、または 13) は無視されます。\$ と同じです。

#### ¥z

文字列の最後のアンカー接尾語。正規表現の一致が文字列の最後で出現する必要があることを示します。行の最後の文字 (ASCII 10、11、12、または 13) は一致のための文字列として扱われます。

以下の例は、文字列の先頭アンカーにより \$LOCATE 一致を制限する方法を示しています。

## ObjectScript

```
SET str="ABCDEFGH"
WRITE $LOCATE(str,"A"),! // returns 1
WRITE $LOCATE(str,"D"),! // returns 4
WRITE $LOCATE(str,"^A"),! // returns 1
WRITE $LOCATE(str,"^D"),! // returns 0 (no match)
```

以下の例は、文字列の最後アンカーにより \$LOCATE 一致を制限する方法を示しています。

## ObjectScript

```
SET str="ABCDABCD"
WRITE $LOCATE(str,"(ABC)"),! // returns 1
WRITE $LOCATE(str,"D"),! // returns 4
WRITE $LOCATE(str,"(ABC)$"),! // returns 0 (no match)
WRITE $LOCATE(str,"(ABCD)$"),! // returns 5
WRITE $LOCATE(str,"D$"),! // returns 8
```

以下の例は、文字列最後のアンカーで改行文字がどのように扱われるかを示しています。

## ObjectScript

```
SET str="ABCDEFGH"$CHAR(10)

WRITE $LOCATE(str,"G$"),! // returns 7
WRITE $LOCATE(str,"G"$CHAR(10)"$"),! // returns 7
WRITE $LOCATE(str,$CHAR(10)"$"),!! // returns 8

WRITE $LOCATE(str,"G$Z"),! // returns 7
WRITE $LOCATE(str,"G"$CHAR(10)"$Z"),! // returns 7
WRITE $LOCATE(str,$CHAR(10)"$Z"),!! // returns 8

WRITE $LOCATE(str,"G$z"),! // returns 0
WRITE $LOCATE(str,"G"$CHAR(10)"$z"),! // returns 7
WRITE $LOCATE(str,$CHAR(10)"$z"),! // returns 8
```

## E.5.2 単語境界

一致が単語境界で発生するように制限できます。単語境界は、非単語構成文字の次の単語構成文字、または文字列の先頭の単語構成文字で特定されます。単語構成文字は、`¥w` 文字タイプ (英字、数字、およびアンダースコア文字) に一致する文字です。一般的にこれは、string の先頭か空白文字や句読点文字に続く単語の最初の文字です。単語境界の正規表現構文は以下のとおりです。

- ・ `¥b` は非単語構成文字と単語構成文字の境界の出現、または文字列先頭の単語構成文字と一致します。
- ・ `¥B` (否定) は単語構成文字と単語構成文字との境界、または非単語構成文字と非単語構成文字との境界の出現と一致します。

以下の例では `¥b` を使用して、部分文字列 `in` または `un` で始まる単語境界との一致を指定しています。

## ObjectScript

```
SET str(1)="unlucky" // match: "un" is at start of string
SET str(2)="highly unlikely" // match: "un" follows a space character
SET str(3)="fall in place" // match: "in" can be followed by a space
SET str(4)="the %integer" // match: % is a non-word character
SET str(5)="down-under" // match: - is a non-word character
SET str(6)="winning" // no match: "in" preceded by word character
SET str(7)="the 4instances" // no match: a number is a word character
SET str(8)="down_under" // no match: an underscore is a word character
FOR i=1:1:8 {
    WRITE $MATCH(str(i),".*\b[iu]n.*")," string",i,!
}
```

以下の例では `¥B` を使用して、単語境界にない場合の正規表現を検索しています。

## ObjectScript

```
SET str(1)="the thirteenth item"
WRITE $LOCATE(str(1),"\Bth") // returns 13 ("th" preceded by a word character)
SET str(2)="the^thirteenth^item"
```

以下の例では、単語構成文字を指定しない正規表現での %b および %B の使用方法を示しています。

## ObjectScript

```
SET str(1)="this##item"
WRITE $LOCATE(str(1),"\b#"),! // returns 5 (the first # at a word boundary)
WRITE $LOCATE(str(1),"\B#") // returns 6 (the first # not at a word boundary)
```

## E.6 論理演算子

論理 AND (&&)、論理 OR (!)、および減算 (-) の演算子で値を結合することで複合文字タイプを表現できます。複合文字タイプは角括弧で囲む必要があります。

暗黙的 OR：角括弧を論理演算子なしで使用すると、一致文字のリストまたは範囲を指定できます。これらのいずれかが true である必要があります。[\p{LU}1234]、[[:upper:]1234]、[\p{LU}1-4]、[[:upper:]1-4] などは、すべての大文字と数字の 1234 に一致します。

AND (&&)：論理 AND を使用すると、複数の文字タイプ・メタ文字を指定できます。どちらも true である必要があります。例えば、一致をギリシャ文字の大文字のみに制限するには、[\p{LU}&&\p{greek}] または [[:upper:]&&[:greek:]] を指定できます。

OR (!)：論理 OR を使用すると、複数の文字タイプ・メタ文字を指定できます。いずれかが true である必要があります。例えば、一致を数字かギリシャ文字に制限するには、[\p{N}|\p{greek}] または [[:digit:]]|[:greek:]] を指定できます。明示的 OR の使用はオプションです。論理演算子のない文字タイプのリストは、論理 OR と解釈されます。

SUBTRACT (-)：論理減算を使用すると、複数の文字タイプ・メタ文字を指定できます。最初は true で、2 番目は false になる必要があります。例えば、ギリシャ文字を除く大文字のすべてに一致を制限するには、[\p{LU}--\p{greek}] または [[:upper:]--[:greek:]] を指定できます。

## E.7 文字表現メタ文字

以下に個々の文字のメタ文字表現を示します。各シーケンスは単一の文字に一致します。

いくつかの個別制御文字 (\$CHAR(7)、\$CHAR(9)、\$CHAR(10)、\$CHAR(12)、\$CHAR(13)、および \$CHAR(27)) は、[単一文字タイプ](#)を使用しても表現できます。

### E.7.1 16 進、8 進、および Unicode の表現

¥xnn または ¥x{nnnn}

16 進表現。例えば、¥x5A は英字 'Z' です。16 進文字 (A ~ F) では、大文字と小文字が区別されないことに注意してください。先頭のゼロは含めることも、省略することもできます。

¥xnn は、1 桁か 2 桁の 16 進数に使用できます。16 進数でそれよりも桁数が多い場合、¥x{nnnn} 中括弧構文を使用する必要があります。nnn は、1 ~ 7 桁の 16 進値で最大値は 010FFFF にできます。例えば、¥x{005A} は英字 'Z' です。¥x{396} はギリシャ文字のゼータです。

## ¥0nnn

8 進表現。nnn 値は、2 ～ 4 桁の 8 進値ですが、左端の桁はゼロにする必要があります。例えば、キャリッジ・リターン文字 \$CHAR(13) は、\015 や \0015 で表現できます。最大値は \0377 で、\$CHAR(255) を示します。

## ¥unnnn

Unicode 表現。nnnn 値は、Unicode 文字に対応する 4 桁の 16 進数です。例えば、\u005A は英字 ‘Z’ (\$CHAR(90)) です。\\u03BB はギリシャ語の子文字のラムダ (\$CHAR(955)) です。

## E.7.2 制御文字表現

制御文字は、出力不能 ASCII 文字 (\$CHAR(0) ～ \$CHAR(31)) です。以下の構文を使用して表現できます。

`\cX`

X は、ASCII 制御文字 (0 ～ 31 の文字) に対応する文字または記号です。文字は、\$CHAR(1) ～ \$CHAR(26) に対応します。例えば、¥cH は \$CHAR(8) で、バックスペース文字です。X 文字では、大文字と小文字は区別されません。出力不能制御文字は、同じ ASCII 文字セット・シーケンスに続きます (\$CHAR(0) = ¥c@ または ¥c`、\$CHAR(27) = ¥c{ または ¥c[, \$CHAR(28) = ¥c| または ¥c¥、\$CHAR(29) = ¥c] または ¥c], \$CHAR(30) = ¥c^ または ¥c~, \$CHAR(31) = ¥c\_ )。

## E.7.3 記号名表現

この文字タイプは、単一の出力可能な句読点、空白、および記号文字の一致に使用できます。構文は、以下のとおりです。

`\N{charname}`

例えば、\N{comma} はコンマに一致します。メタ文字の ¥N は大文字である必要があります。

サポート対象文字名には、アクセント記号 (´)、アンド記号 (&)、アポストロフィ (’)、アスタリスク (\*)、短音記号 (˘)、セディラ (˙)、コロン (:)、コンマ (,)、短剣符 (†)、度数記号 (°)、除算記号 (÷)、ドル記号 (\$)、二重短剣符 (§)、全角ダッシュ (一)、二分ダッシュ (‐)、感嘆符 (!)、等号 (=)、終止符 (.), 抑音アクセント (˘)、無限大 (∞)、左中括弧 ([)、左括弧 ((), 左角括弧 ([), 長音記号 (ˉ)、乗算記号 (×)、加算記号 (+)、シャープ記号 (#)、プライム符号 (’), 疑問符 (?), 右中括弧 (]), 右括弧 ()), 右角括弧 (]), セミコロン (;)、スペース ( ), 平方根 (√)、チルダ (˘)、垂直線 (|) が含まれます。添え字 0 ～ 9 と上付き文字 0 ～ 9 もサポートされています。

## E.8 モード

モードにより、これに続く文字一致の解釈が変更されます。mode は単一の小文字により指定されます。モードの使用には、2 つの方法があります。

- ・ 正規表現シーケンスのモード。例えば、(?i) です。
- ・ 正規表現内における指定リテラルのモード。のいずれかの型にできます。例えば、(?i:(fred|ginger)) です。

以下の mode 文字がサポートされます。

(?)i

ケース・モード。アクティブな場合、大文字と小文字を正規表現にマッチングさせる際に、大文字と小文字の違いが無視されます。

(?)m

複数行モード。複数行の文字列に適用される際に、`^` (文字列の先頭) と `$` (文字列の最後) の[アンカー](#)の動作に影響を与えます。既定では、これらのアンカーは、文字列全体に適用されます。複数行モードがアクティブな場合、これらのアンカーは、複数行内における各行の先頭と終端に適用されます。行の先頭は、改行文字の 10、11、12、13、および 133 (および Unicode の 8232 と 8233) のいずれかにできます。

(?)s

単一行モード。オフの場合、ドット (`.`) [ワイルドカード](#) は、改行文字 10、11、12、13、および 133 (および Unicode の 8232 と 8233) に一致しません。オンの場合、ドット (`.`) ワイルドカードは、改行文字を含めて、すべての文字に一致します。キャリッジ・リターン (`$CHAR(13)`) と改行 (`$CHAR(10)`) のペアがこの順序で指定されていると、正規表現で単一文字としてカウントされます。

(?)x

フリー・スペース・モード。空白と[後続のコメント](#)が正規表現で使用できます。

## E.8.1 正規表現シーケンスのモード

regex モードは、適用された時点から正規表現の最後まで、または明示的にオフになるまで、正規表現の解釈を制御します。構文は、以下のとおりです。

```
(?n)  to turn mode on
(?-n) to turn mode off
```

`n` は、モード・タイプを指定する単一の小文字です。

以下の例は、ケース・モード (i) を示します。

### ObjectScript

```
WRITE $MATCH("A","(?i)[abc]"),!
WRITE $MATCH("a","(?i)[abc]")
```

以下の例は、ケース・モード (i) を示します。最初の正規表現では大文字と小文字が区別されます。2 番目の正規表現における先頭はケース・モード修飾子 (i) であり、正規表現では大文字と小文字が区別されません。

### ObjectScript

```
SET name(1)="Smith,John"
SET name(2)="dePaul,Lucius"
SET name(3)="smith,john"
SET name(4)="John Smith"
SET name(5)="Smith,J"
SET name(6)="R2D2,CP30"
SET n=1
WHILE $DATA(name(n)) {
  IF $MATCH(name(n),"^p{LU}p{LL}+,p{LU}p{LL}+")
  { WRITE name(n)," : case match",! }
  ELSEIF $MATCH(name(n),"(?i)^p{LU}p{LL}+,p{LU}p{LL}+")
  { WRITE name(n)," : non-case match",! }
  ELSE { WRITE name(n)," : not a valid name",! }
  SET n=n+1 }

```

以下の例は、単一行モード (s) を示します。このモードでは、`".*"` が改行文字のある文字列に一致します。

## ObjectScript

```

SET line(1)="This is a string without line breaks."
SET line(2)="This is a string with"_$CHAR(10)"one line break."
SET line(3)="This is a string"_$CHAR(11)"with"_$CHAR(12)"two line breaks."
SET i=1
WHILE $DATA(line(i)) {
  IF $MATCH(line(i),".*") {WRITE "line(",i,") is a single line string",! }
  ELSEIF $MATCH(line(i),"(?s).*") {WRITE "line(",i,") is a multiline string",! }
  ELSE {WRITE "string error",! }
  SET i=i+1 }

```

以下の例は、単一行モード (?s) を示します。キャリッジ・リターンと改行のペアがその順序出現すると、正規表現で 1 文字としてカウントされます。

## ObjectScript

```

SET str(1)="one"_$CHAR(13)$CHAR(10)"two" // CR/LF
SET str(2)="one"_$CHAR(10)$CHAR(13)"two" // LF/CR
SET i=1
WHILE $DATA(str(i)) {
  WRITE $LENGTH(str(i))," is the length of string ",i,!
  IF $MATCH(str(i),"(?s){7}") { WRITE "string ",i," matches 7 chars",! }
  ELSEIF $MATCH(str(i),"(?s){8}") { WRITE "string ",i," matches 8 chars",! }
  ELSE { WRITE "string match error",! }
  SET i=i+1
}

```

以下の例は、複数行モード (?m) を示します。終端アンカー (\$) により識別される部分文字列を探します。単一行モードでは、この終端部分文字列は必ず break (文字列における最後の部分文字列) になります。複数行モードでは、終端部分文字列は、複数行文字列内で行が終了する任意の部分文字列にできます。

## ObjectScript

```

SET line(1)="String without line break"
SET line(2)="String with"_$CHAR(10)" one line break"
SET line(3)="String"_$CHAR(11)" with"_$CHAR(12)" two line break"
SET i=1
WHILE $DATA(line(i)) {
  WRITE $LOCATE(line(i),"(String|with|break)$")," line(",i,") in single-line mode",!
  WRITE $LOCATE(line(i),"(?m)(String|with|break)$")," line(",i,") in multi-line mode",!!
  SET i=i+1 }

```

## E.8.2 リテラルのモード

また、以下の構文を使用して、モード修飾子をリテラル (またはリテラルのセット) に適用できます。

```
(?mode:literal)
```

このモード修飾子は、括弧内のリテラルにのみ適用されます。

以下のケース・モード (?i) 例では、この接頭語の大文字化処理に関係なく、先頭が de、del、dela、および della である名字 (lname) に一致します。残りの lname は大文字で始まり、小文字が 1 文字以上その後が続く必要があります。

## ObjectScript

```

SET lname(1)="deTour"
SET lname(2)="DeMarco"
SET lname(3)="DeLaRenta"
SET lname(4)="DelCarmine"
SET lname(5)="dellaRobbia"
SET i=1
WHILE $DATA(lname(i)) {
  WRITE $MATCH(lname(i),"(?i:de|del|dela|della)\p{LU}\p{LL}+")," = ",lname(i),!
  SET i=i+1 }

```

## E.9 Comments

正規表現内で、2 種類のコメントを指定できます。

- ・ 埋め込みコメント
- ・ 行末コメント ((?x) モード内のみ)

### E.9.1 埋め込みコメント

以下の構文を使用することで、埋め込みコメントを正規表現内で指定できます。

```
(?# comment)
```

以下の例は、正規表現内におけるコメントの使用法を示しています。このコメントでは、この書式一致はアメリカ式日付 (MM/DD/YYYY) 用であり、ヨーロッパ式日付 (DD/MM/YYYY) 用ではないことを記載しています。

#### ObjectScript

```
WRITE $MATCH("04/28/2012", "^([01]\d(?: months)/[0123]\d(?: days)/\d\d\d\d$")
```

### E.9.2 行末コメント

フリー・スペース・モード (?x) が有効な場合、以下の構文を使用して、コメントを正規表現の最後に指定できます。

```
# comment
```

以下の例は、フリー・スペース・モードでの最後のコメントを示しています。

#### ObjectScript

```
WRITE $MATCH("04/28/2012", "^([01]\d/[0123]\d/\d\d\d\d$")," no comment",!  
WRITE $MATCH("04/28/2012", "^([01]\d/[0123]\d/\d\d\d\d$# date test")," comment no (?x) mode",!  
WRITE $MATCH("04/28/2012", "(?x) ^([01]\d/[0123]\d/\d\d\d\d$# date test")," comment in (?x) mode",!
```

フリー・スペース・モードでは、空白を正規表現内に含めることができます。

## E.10 エラー・メッセージ

適切に regexp を指定しないと、<REGULAR EXPRESSION> エラーが発生します。エラーのタイプを判別するには、LastStatus() メソッドを以下の例のように呼び出すことができます。

## ObjectScript

```
TRY {
  WRITE "TRY block:",!
  WRITE $MATCH("A","\p{LU}"),! // good regexp
  WRITE $MATCH("A","\p{ }"),! // bad regexp
}
CATCH exp {
  WRITE !,"CATCH block exception handler:",!
  IF 1=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception",!
    WRITE "Name: ", $ZCVT(exp.Name,"O","HTML"),!
    WRITE "Location: ",exp.Location,!
    WRITE "Code: ",exp.Code,! }
  ELSE {WRITE "Unexpected exception type",! RETURN }
  WRITE "%Regex.Matcher status:"
  DO $SYSTEM.Status.DisplayError(##class(%Regex.Matcher).LastStatus())
  RETURN
}
```

これらのエラーのリストは、“[一般的なエラー・メッセージ](#)”を参照してください。

## E.11 関連項目

- ・ [\\$LOCATE](#) 関数
- ・ [\\$MATCH](#) 関数
- ・ [%Regex.Matcher](#)
- ・ [パターン・マッチング \(?\) 演算子](#) (正規表現を使用しません)



# F

## 変換テーブル

InterSystems IRIS® データ・プラットフォームは、文字の変換タスクで変換テーブル（入出力テーブルとも呼ばれます）を使用します。一部の API 呼び出し（および `$zconvert` 関数）は、引数として変換テーブルを受け入れることができます。ここでは、使用可能な変換テーブルのリファレンス情報を提供します。

### F.1 概要

文字の変換に変換テーブルが使用される一般的なシナリオには以下の 2 つがあります。

- ・ 多くのコンテキスト（URL、HTML、JSON など）では、特定の文字は許可されておらず、エスケープ・シーケンスで表す必要があります。このような場合、それらの文字と許可されている文字セットとの間で変換が必要となります。
- ・ データベース外のソースから読み取る、またはデータベース外の宛先に書き込む場合、そのエンティティは InterSystems IRIS が使用しているものとは異なる文字セットを期待している可能性があります。この場合、文字エンコードを変換する必要があります。

特定のコンテキストの“変換テーブル”は、実際にはテーブルのペアです。1 つのテーブルは既定の文字セットを他言語文字セット（または他言語コンテキスト）に変換する方法を指定し、もう 1 つのテーブルは反対方向に変換する方法を指定します。InterSystems IRIS では、このテーブルのペアを、入力モードと出力モードを持つ 1 つの単位として参照して変換を実行します。したがって、HTML との間での変換を管理するための HTML 変換テーブルがあり、CP1250 文字セットとの間での変換を管理するための CP1250 変換テーブルがあります。

### F.2 テーブルのリスト

以下は、InterSystems IRIS 変換テーブルのリストです。

#### **RAW**

8 ビット文字または 16 ビット Latin-1 文字（高位バイトの値が 00 の Unicode 文字）の変換を行いません。

RAW 変換は、非 Latin-1 ロケール（`rusw` など）を使用する InterSystems IRIS システムでは使用できません。

#### **SAME**

8 ビット文字を対応する Unicode 文字に変換します。

## HTML

HTML エスケープ文字を文字列に追加 (出力モード) または文字列から削除 (入力モード) します。“[出力エスケープ](#)” のテーブルを参照してください。

## JS または JSML

提供された JavaScript 変換テーブルを使用し、JavaScript 内で使用できるように文字列内の文字をエスケープ処理します。出力変換については、“[出力エスケープ](#)” のテーブルを参照してください。JS と JSML の比較は、“[JS と JSML、JSON と JSONML の変換](#)” を参照してください。入力変換については、“¥0”、“¥000”、“¥x00”、および “¥u0000” はすべて、NULL の有効なエスケープ・シーケンスです。

## JSON または JSONML

指定された変換テーブルを使用して JSON 形式に変換します。出力変換については、“[出力エスケープ](#)” のテーブルを参照してください。JSON と JSONML の比較は、“[JS と JSML、JSON と JSONML の変換](#)” を参照してください。入力変換については、“¥0”、“¥000”、“¥x00”、および “¥u0000” はすべて、NULL の有効なエスケープ・シーケンスです。

## URI

URI パラメータ・エスケープ文字を文字列に追加 (出力モード) または文字列から削除 (入力モード) します。URI は、文字 `!"#$%&'()*+,-./:;<=>?@[ ]^`{|}~` を以下のようにエンコードします。`%20%21%22%23%24%25%26%27%28%29%2A%2B%2C%2F%3A%3B%3C%3D%3E%3F%40%5B%5D%5E%60%7B%7C%7D`。スペース文字は `%20` としてエンコードします。

二重引用符文字 (`"My "perfect" code"` のような引用符付きの文字列に含まれる場合は二重にエスケープする必要があります) は `%22` としてエンコードされます。

チルダ (`~`) 文字はエンコードしません。“[出力エスケープ](#)” のテーブルを参照してください。

URI は、`$CHAR(255)` (Unicode 文字) より大きな文字を UTF-8 にエンコードしてから、UTF-8 値を 16 進数に `%` エンコードします。

“[URL および URI の変換](#)” も参照してください。

## URL

URL パラメータ・エスケープ文字を文字列に追加 (出力モード) または文字列から削除 (入力モード) します。URL は、文字 `"#%&+,:;<=>?@[ ]^`{|}~` を以下のようにエンコードします。`%20%22%23%25%26%2B%2C%3A%3B%3C%3D%3E%3F%40%5B%5D%5E%60%7B%7C%7D%7E`。

スペース文字は `%20` としてエンコードします。

二重引用符文字 (`"My "perfect" code"` のような引用符付きの文字列に含まれる場合は二重にエスケープする必要があります) は `%22` としてエンコードされます。

“[出力エスケープ](#)” のテーブルを参照してください。`$CHAR(255)` より大きな文字は、Unicode 16 進数で表されます (`$CHAR(256) = %u0100`)。

“[URL および URI の変換](#)” も参照してください。

## UTF8

UTF-8 エンコード16 ビットの Unicode 文字を一連の 8 ビット文字に変換 (出力モード) します。ASCII 16 ビットの Unicode 文字は、単一の 8 ビット文字に変換されます。例えば、16 進数 0041 (文字 “A”) は、8 ビット文字の 16 進数 41 に変換されます。非 ASCII の Unicode 文字は、2 つまたは 3 つの 8 ビット文字に変換されます。

0080 から 07FF の Unicode 16 進数は、2 つの 8 ビット文字に変換されます。これには、ラテン 1 補助とラテン拡張文字、およびギリシャ語、キリル文字、ヘブライ語、アラビア文字が含まれます。

0800 から FFFF の Unicode 16 進数は、3 つの 8 ビット文字に変換されます。これらは Unicode 基本多言語面の残りの部分を構成します。したがって、\$CHAR(0) から \$CHAR(127) の ASCII 文字は、RAW モードと UTF8 モードでは同じになり、\$CHAR(128) 以上の文字は変換されます。

入力モードではこの変換が逆になります。詳細は [“Unicode”](#) を参照してください。

## XML

XML エスケープ文字を文字列に追加 (出力モード) または文字列から削除 (入力モード) します。[“出力エスケープ”](#) のテーブルを参照してください。

## その他のテーブル

残りの変換テーブルは文字セット変換に固有であり、これらのテーブルにはこれらの文字セットと同じ名前が付付けられます。テーブルには以下のようなものがあります。

- ・ UnicodeLittle
- ・ UnicodeBig
- ・ CP1250
- ・ CP1251
- ・ CP1252
- ・ CP1253
- ・ CP1255
- ・ CP437
- ・ CP850
- ・ CP852
- ・ CP866
- ・ CP874
- ・ EBCDIC
- ・ Latin2
- ・ Latin9
- ・ LatinC
- ・ LatinG
- ・ LatinH
- ・ LatinT

現在の変換テーブルをリストする方法については、[“関連する API”](#) を参照してください。

## F.3 出力エスケープ

ここでは、特定の[変換テーブル](#)が出力モードで文字を変換する方法を示します。

	HTML	JS	JSON	URI	URL	XML
NULL \$CHAR(0)		¥x00	¥u0000	%00	%00	
\$CHAR(1) ~ \$CHAR(7)		¥x01 ~ ¥x07	¥u0001 ~ ¥u0007	%01 ~ %07	%01 ~ %07	
バックスペース \$CHAR(8)		¥b	¥b	%08	%08	
水平タブ \$CHAR(9)		¥t	¥t	%09	%09	
改行 \$CHAR(10)		¥n	¥n	%0A	%0A	
垂直タブ \$CHAR(11)		¥v	¥u000B	%0B	%0B	
書式送り \$CHAR(12)		¥f	¥f	%0C	%0C	
キャリッジ・リ ターン \$CHAR(13)		¥r	¥r	%0D	%0D	
\$CHAR(14) ~ \$CHAR(31)			¥u000E ~ ¥u001F	%0E ~ %1F	%0E ~ %1F	
\$CHAR(32)				%20	%20	
” (二重)	&quot;	¥”	¥”	%22	%22	&quot;
#				%23	%23	
\$				%24		
%				%25	%25	
&	&amp;			%26	%26	&amp;
‘ (アポストロ フィ) \$CHAR(39)	&#39;	¥’		%27		&apos;
(				%28		
)。				%29		
*				%2A		
+				%2B	%2B	
,				%2C	%2C	
/ (スラッシュ) \$CHAR(47)		¥/		%2F		
:				%3A	%3A	

	HTML	JS	JSON	URI	URL	XML
;				%3B	%3B	
<	&lt;			%3C	%3C	&lt;
=				%3D	%3D	
>	&gt;			%3E	%3E	&gt;
?				%3F	%3F	
@				%40	%40	
[				%5B	%5B	
¥		¥¥	¥¥	%5C	%5C	
]				%5D	%5D	
^				%5E	%5E	
`				%60	%60	
{				%7B	%7B	
				%7C	%7C	
}				%7D	%7D	
~					%7E	
\$CHAR(127)				%7F	%7F	
\$CHAR(128) ~ \$CHAR(159)				%C2%80 ~ %C2%9F	%80 ~ %9F	
\$CHAR(160)	&nbsp;			%C2%A0	%A0	
\$CHAR(161) ~ \$CHAR(191)				%C2%A1 ~ %C2%BF	%A1 ~ %BF	
\$CHAR(192) ~ \$CHAR(191)				%C3%80 ~ %C3%BF	%C0 ~ %FF	

## F.4 URL および URI の変換

URL または URI で使用できるのは、特定の 8 ビットの ASCII 文字のみです。その他のすべての文字は % で始まるエスケープ・シーケンスで表す必要があります。Unicode 文字を含む文字列を URL または URI に変換する場合、UTF-8 エンコードを使用して最初にローカル表現を 8 ビットの中間表現に変換します。その後、UTF-8 の結果を URL エンコードに変換します。URL を元の Unicode 文字列に戻すには、逆の処理を実行します。これを以下の例で示します。

## ObjectScript

```

SET ustring="US$ to "_$CHAR(8364)_" échange"
WRITE "initial string is: ",ustring,!
ConvertUnicodeToURL
SET utfo = $ZCONVERT(ustring,"O","UTF8")
SET urlo = $ZCONVERT(utfo,"O","URL")
WRITE "Unicode to URL conversion: ",urlo,!
ConvertURLtoUnicode
SET urli = $ZCONVERT(urlo,"I","URL")
SET utfi = $ZCONVERT(urli,"I","UTF8")
WRITE "URL to Unicode conversion: ",utfi

```

## F.5 JS と JSML、JSON と JSONML の変換

JS と JSON の変換では、Unicode 文字に UTF-8 エンコードを使用します。JSML と JSONML の変換では、Unicode 文字をエンコードなしで変換します。ASCII 文字 (\$CHAR(0) ~ \$CHAR(127)) では、JS と JSML のエンコードは同一です。ASCII 文字 (\$CHAR(0) ~ \$CHAR(127)) では、JSON と JSONML のエンコードは同一です。

以下の例では、JS の文字と JSML の文字の変換を比較します。

## ObjectScript

```

FOR i=1:1:256 {
  SET x=$ZCVT($C(i),"O","JS")
  SET y=$ZCVT($C(i),"O","JSML")
  IF x=y {
    WRITE ". "
  } ELSE {
    WRITE !!,$ZHEX(i),!,"JS: " ZZDUMP x WRITE !,"JSML: " ZZDUMP y
  }
}

```

## F.6 関連する API

現在使用可能な変換テーブルのリストは、以下の例に示す、**%SYS.NLS.Locale** の **XLTTTables** プロパティを参照してください。

## ObjectScript

```

SET nlsoref=##class(%SYS.NLS.Locale).%New()
WRITE $LISTTOSTRING(nlsoref.XLTTTables," ", " ")

```

**%Net.Charset** を使用して、InterSystems IRIS 内で文字セットを表すこともできます。このクラスは、以下のクラス・メソッドを含んでいます。

- ・ **GetDefaultCharset()** は、現在の InterSystems IRIS ロケールの既定の文字セットを返します (次の見出しを参照)。
- ・ **GetTranslateTable()** は、指定された入力文字セットの InterSystems IRIS 変換テーブルの名前を返します。
- ・ **TranslateTableExists()** は、指定された文字セットの変換テーブルがロードされているかどうかを示します。

メソッド・シグニチャについては、**%Net.Charset** のクラス・ドキュメントを参照してください。

## F.7 関連概念

InterSystems IRIS のロケールとは、ユーザの言語、通貨記号、書式、および特定の国または地域に固有のその他の規約を指定するメタデータのセットです。ロケールには変換テーブルが含まれます。InterSystems IRIS は、各国言語サポート (NLS) という語句を使用して、ロケール定義を表示して拡張するために使用するツールとロケール定義を一括して示します。

ロケール定義の外側で、特定の InterSystems IRIS インスタンスは、入出力アクティビティに特定の変換テーブルを使用するように既定で構成されています。これらは、既定の変換テーブルまたは既定の入出力テーブルです。現在の既定値を確認するには、**%SYS.NLS.Table** を使用します。詳細は、クラス・リファレンスを参照してください。

## F.8 関連項目

- ・ [\\$zconvert](#)
- ・ [各国言語サポートのシステム・クラスの使用法](#)
- ・ [%SYS.NLS.Locale](#)
- ・ [%SYS.NLS.Table](#)





# G

## インターシステムズ・アプリケーションでの数値の計算

ここでは、InterSystems IRIS® データ・プラットフォームによってサポートされている数値形式について詳しく説明します。

### G.1 概要

InterSystems IRIS で数値を表現する方法には 2 種類あります。

- ・ 1 つは InterSystems IRIS オリジナルの実装当初からあったもので、10 進形式と呼ばれます。  
クラス定義では、プロパティに **小数点形式の数値** を含めるときに、`%Library.Decimal` データ型クラスを使用します。
- ・ もう 1 つは IEEE Binary Floating-Point Arithmetic 標準 ([#754-2019](#)) に準拠した形式で、近年サポートされるようになりました。この後者の形式は `$DOUBLE` 形式と呼ばれ、値をこの形式で作成するために使用される ObjectScript 関数 (`$DOUBLE`) にちなんで名付けられたものです。  
クラス定義では、プロパティに **`$DOUBLE` 形式の数値** を含めるときに、`%Library.Double` データ型クラスを使用します。

#### G.1.1 SQL 表現

InterSystems SQL データ型の `DOUBLE` と `DOUBLE PRECISION` は、IEEE 浮動小数点数 (つまり `$DOUBLE`) を表します。SQL `FLOAT` データ型は、標準の InterSystems IRIS の 10 進数を表します。

### G.2 10 進形式

InterSystems IRIS では、10 進数は内部的に 2 つの部分で表されます。最初の部分は仮数、2 番目の部分は指数と呼ばれます。

- ・ 仮数は、対象となる値の有効桁数で構成されます。符号付 64 ビット整数として保存され、その値の右側には小数点があると見なされます。精度を失うことなく表現できる指数 0 の最大の正の整数は 9,223,372,036,854,775,807、最大の負の整数は -9,223,372,036,854,775,808 です。
- ・ 指数は、符号付バイトとして内部的に保存されます。値の範囲は 127 ~ -128 です。

これは、値の 10 進数の指数です。つまり、その数の値は、仮数に 10 の指数のべき乗を乗算したものになります。

例えば、ObjectScript リテラル値 1.23 の場合、仮数は 123 であり、指数は -2 です。

したがって、InterSystems IRIS がネイティブ形式で表現できる数値の範囲は、約 1.0E-128 ～ 9.22E145 になります。(最初の値は最小の指数を持つ最小の整数になります。2 番目の値は最も大きい整数で、小数点が左に移動して、それに従って指数が増加する形式で表されます。)

小数桁数が 18 の数値はすべて正確に表現できます。また、仮数が表現範囲内にある数値は、19 桁の値として正確に表現できます。

注釈 InterSystems IRIS では、数値を 10 進形式にする必要がない限り、仮数を正規化することはありません。したがって、仮数が 123 で指数が 1 の数と、仮数が 1230 で指数が 0 の数は同じものを表します。

## G.3 \$DOUBLE 形式

インターシステムズの \$DOUBLE 形式は、[IEEE-754-2019](#)、具体的には 64 ビットのバイナリ (倍精度) 表現に準拠します。つまり、以下の 3 つの部分で構成されます。

- ・ 符号ビット
- ・ 2 つの指数の 11 ビット乗。指数値には 1023 のバイアス分が含まれるので、\$DOUBLE(1.0) の指数の内部値は 0 ではなく 1023 になります。
- ・ 正の 52 ビット小数の仮数部。仮数部は常に正の数値として処理され正規化されるため、仮数部として表されていないくても、1 ビットは先頭のバイナリ桁であると見なされます。したがって、仮数部は数値としては 53 ビット長になり、値 1 の後に暗黙の 2 進小数点、その後に小数の仮数部が続きます。これは、暗黙的に  $2^{52}$  で除算された整数として考えることができます。

整数として、0 ～ 9,007,199,254,740,992 のすべての値は正しく表現できます。大きな整数を正しく表現できるかどうかは、そのビット・パターンによります。

このデータ表現には、InterSystems IRIS のネイティブ形式にはない以下の 3 つのオプション機能があります。

- ・ (負の数の平方根を取るなど) 不正な計算結果を NaN (非数) として表現する機能。
- ・ +0 および -0 の両方を表現する機能。
- ・ 無限大を表現する機能。
- ・ この標準は、 $2^{-1022}$  より小さい数の表現を提供します。これは、緩やかな精度の損失と呼ばれる手法で行われます。詳細は、[標準](#)を参照してください。

これらの機能は、個別プロセスの場合は `%SYSTEM.Process` クラスの `IEEEError()` メソッド、システム全般の場合は `Config.Miscellaneous` クラスの `IEEEError()` メソッドを介してプログラム制御されます。

## 重要

IEEE バイナリ浮動小数点表現を使用して計算すると、同じ IEEE 演算でも異なる結果が得られる場合があります。インターシステムズでは、以下に対して独自の実装を記述しています。

1. \$DOUBLE バイナリ浮動小数点と 10 進数との間の変換。
2. \$DOUBLE と数値文字列との間の変換。
3. \$DOUBLE とその他の数値型との間の比較。

これにより、\$DOUBLE 値を InterSystems IRIS データベースに挿入したり、InterSystems IRIS データベースからフェッチしたりするときに、すべてのハードウェア・プラットフォームで結果が同じになることが保証されます。

ただし、\$DOUBLE 型を含むその他すべての計算では、InterSystems IRIS はベンダが提供する浮動小数点ライブラリのサブルーチンを使用します。そのため、同じ演算のセットでも、プラットフォームによってわずかな差異が生じる可能性があります。ただし、すべてのケースにおいて、インターシステムズの \$DOUBLE の計算は、C の double 型で実行するローカルの計算と等しくなります。つまり、インターシステムズの \$DOUBLE の計算に対するプラットフォーム間の差異は、最大でも、それぞれ同じプラットフォーム上で実行される C プログラムの IEEE 値の計算結果の差異に留まります。

## G.4 数値形式の選択

使用する形式の選択は、計算の要件に大きく依存します。InterSystems IRIS の 10 進形式では 18 桁を超える精度が許可されますが、\$DOUBLE では 15 桁しか保証されません。

多くの場合、10 進形式は使用法がより簡単で、結果も正確です。予想どおりの結果が返されるため、通常、(通貨計算などの) 10 進値の計算に使用されます。ほとんどの 10 進数の小数は、バイナリ的小数ほど正確には表現できません。

それに対して、\$DOUBLE 形式の数値の範囲は、ネイティブ形式で許可される 1.0E145 よりもはるかに大きい 1.0E308 になります。数値の範囲が重要となるアプリケーションでは、\$DOUBLE を使用する必要があります。

データを外部で共有するアプリケーションでは、暗黙の変換対象とならないよう、データを \$DOUBLE 形式で維持することも検討できます。他のほとんどのシステムでは、基礎となるハードウェア・アーキテクチャによって直接サポートされることから、バイナリ浮動小数点の表現に IEEE 標準を使用しています。したがって 10 進形式の値は、(例えば ODBC/JDBC、SQL、または言語バインディング・インタフェースを使用して) 情報交換前に変換する必要があります。

InterSystems IRIS の 10 進数に定義された範囲内に \$DOUBLE 値がある場合、10 進数に変換して再度 \$DOUBLE 値に戻すと、常に同じ数値が生成されます。\$DOUBLE 値の精度は 10 進値よりも低いため、逆の場合はこのようにはなりません。

その理由から、可能であれば、いずれか 1 つのデータ表現のみを使用して計算を実行することをお勧めします。データ表現間で変換を繰り返すと、精度が落ちる場合があります。多くのアプリケーションでは、すべての計算に InterSystems IRIS の 10 進形式を使用できます。\$DOUBLE 形式は、IEEE 形式を使用するシステムとデータを交換するアプリケーションのサポートを目的としています。

\$DOUBLE ではなく、InterSystems IRIS の 10 進形式をお勧めする理由は、以下のとおりです。

- ・ InterSystems IRIS の 10 進形式は精度が高く、\$DOUBLE の桁数が 16 未満であるのに対し、ほぼ 19 桁の精度を持っています。
- ・ InterSystems IRIS の 10 進形式では、正確に 10 進数の小数を表現できます。例えば、0.1 という値は InterSystems IRIS の 10 進形式では正確に 0.1 になりますが、バイナリの浮動小数点では同値となる値がないので、\$DOUBLE 形式では語義的な近似値で 0.1 を表現する必要があります。

科学的数値では、以下の点でインターシステムズの 10 進形式よりもインターシステムズの \$DOUBLE の方が有利です。

- ・ \$DOUBLE は、ほとんどの計算ハードウェアで使用されている IEEE 倍精度浮動小数点とまったく同じデータ表現を使用します。
- ・ \$DOUBLE の方が範囲が広く、\$DOUBLE による最大値は 1.7E308 であるのに対し、インターシステムズの 10 進形式による最大値は 9.2E145 です。

## G.5 変換：文字列

値を文字列から数値に変換する場合や、記述された定数をプログラムのコンパイル時に処理する際、最初の 38 桁の有効桁数のみが仮数の値に影響します。それ以降のすべての桁はゼロとして処理されます。つまり、指数値の決定に使用され、仮数値には影響しません。

### G.5.1 数値としての文字列

InterSystems IRIS では、式に文字列が使用されている場合、その文字列の値は、その最初の文字で開始される文字列内の最も長い数値リテラルになります。そのようなリテラルが存在しない場合、文字列の計算値は 0 になります。

### G.5.2 添え字としての数値文字列

計算では、“04”と“4”の文字列間に違いはありません。しかし、このような文字列がローカル配列またはグローバル配列の添え字として使用される場合、InterSystems IRIS ではこれらの文字列が区別されます。

InterSystems IRIS では、先頭（存在する場合はマイナス記号の後）や 10 進小数の末尾に 0 を含む数値文字列は、添え字として使用する場合、文字列であるかのように処理されます。これらは、文字列として、数値を含んでいるため、計算に使用できます。しかし、ローカル変数またはグローバル変数の添え字としては、文字列として扱われ、文字列として照合されます。以下に、例をいくつか示します。

- ・ 4 と 04
- ・ 10 と 10.0
- ・ .001 と 0.001
- ・ -.3 と -0.3
- ・ 1 と +01

左側は添え字として使用した場合、数値と見なされ、右側は文字列として扱われます。（無関係な先頭と末尾の 0 の部分を除く左側の形式は、キャノニック形式とも呼ばれます。）

通常の照合では、以下の例のように、数値は文字列の前にソートされます。

#### ObjectScript

```
SET ^|TEST("2") = "standard"
SET ^|TEST("01") = "not standard"
SET NF = "Not Found"

WRITE ""2"", ": ", $GET(^|TEST("2"),NF), !
WRITE 2, ": ", $GET(^|TEST(2),NF), !
WRITE ""01"", ": ", $GET(^|TEST("01"),NF), !
WRITE 1, ": ", $GET(^|TEST(1),NF), !, !
SET SUBS=$ORDER(^|TEST(""))
WRITE "Subscript Order:", !
WHILE (SUBS '= "") {
    WRITE SUBS, !
    SET SUBS=$ORDER(^|TEST(SUBS))
}
```

## G.6 変換：\$DOUBLE へ

**注意** インターシステムズでは、10 進形式と \$DOUBLE 形式間の変換は、アプリケーションで明示的に制御することを推奨しています。

任意の数値を \$DOUBLE 形式に変換するには、**\$DOUBLE** 関数を使用します。また、この関数は、\$DOUBLE(<S>) 式を使用して、非数および無限大に対する IEEE 表現の明示的な構築を許可します。ここで <S> には、以下のいずれかを指定できます（大文字小文字は区別されません）。

- ・ 非数を生成する文字列 "nan"
- ・ 無限大を生成する文字列 inf、+inf、-inf、infinity、+infinity、または -infinity のいずれか
- ・ リテラル -0 または "-0"（同等）

## G.7 変換：10 進数へ

**注意** インターシステムズでは、10 進形式と \$DOUBLE 形式間の変換は、アプリケーションで明示的に制御することを推奨しています。

数値（あらゆる種類）を 10 進形式に変換するには、**\$DECIMAL** 関数を使用します。

### G.7.1 \$DECIMAL(x)

引数を 1 つ使用した形式の \$DECIMAL 関数では、指定した引数が 10 進形式に変換され、数値の小数部分が 19 桁に丸められます。\$DECIMAL では常に最も近い 10 進数値に丸められます。

### G.7.2 \$DECIMAL(x, n)

引数を 2 つ使用した形式の \$DECIMAL では、返される桁数を正確に制御できます。n が 38 より大きい場合、<ILLEGAL VALUE> エラーが生成されます。n が 0 より大きい場合、n 桁の有効桁数に丸められた x の値が返されます。

n が 0 の場合、値の決定には、以下のルールが使用されます。

1. x が無限大の場合、必要に応じて INF または -INF を返します。
2. x が非数の場合、NAN を返します。
3. x が正または負の 0 の場合、0 を返します。
4. x を 20 桁以下の有効桁数で正確に表現できる場合、その有効桁数のキャノニック形式の数値文字列を返します。
5. それ以外の場合は、10 進表現を 20 桁の有効桁数に切り捨てます。さらに
  - a. 20 番目の桁が 0 の場合、1 に置き換えます。
  - b. 20 番目の桁が 5 の場合、6 に置き換えます。

その後、結果の文字列を返します。

20 桁目が不正確に 0 または 5 になる場合を除き、20 桁目を 0 に切り捨てるこの丸めのルールには、以下の特性があります。

- ・ ある \$DOUBLE 値が、ある 10 進数値と異なる場合、これらの 2 つの値は常に、等しくない表現文字列を持ちます。

- ・ <MAXNUMBER> エラーを生成せずに \$DOUBLE 値を 10 進数に変換できる場合、その結果は \$DOUBLE 値を文字列に変換してから、その文字列を 10 進数値に変換する場合と同じです。2 つの変換を行う場合、double round エラーは発生しません。

## G.8 変換：10 進数から文字列へ

10 進数値は、例えば連結演算子のオペランドの 1 つとして使用した場合、既定で文字列に変換できます。変換を詳細に制御する必要がある場合には、\$FNUMBER 関数を使用します。

## G.9 算術演算

### G.9.1 同種の表現

10 進数値のみを含む式では、常に 10 進数の結果が生成されます。同様に、\$DOUBLE 値のみを含む式では、常に \$DOUBLE の結果が生成されます。さらに、

- ・ 10 進数値を含む計算結果がオーバーフローする場合、<MAXNUMBER> エラーとなります。リテラルの場合のように、\$DOUBLE への自動変換は行われません。
- ・ 10 進数式がアンダーフローする場合、式の結果として 0 が生成されます。
- ・ 既定では、オーバーフロー、0 による除算、および無効な演算の IEEE エラーは、無限大や非数の結果を生成するのではなく、それぞれ <MAXNUMBER>、<DIVIDE>、および <ILLEGAL VALUE> エラーとして示されます。この動作は、個別プロセスの場合は %SYSTEM.Process クラスの IEEEError() メソッド、システム全般の場合は Config.Miscellaneous クラスの IEEEError() メソッドで変更できます。
- ・  $0 ** 0$  (10 進数) という式では 10 進数値の 0 が生成されますが、 $\$DOUBLE(0) ** \$DOUBLE(0)$  という式では \$DOUBLE 値の 1 が生成されます。前者は InterSystems IRIS では常に当てはまっています。後者は IEEE 標準の要件です。

### G.9.2 異種の表現

10 進数表現と \$DOUBLE 表現の両方を含む式では、常に \$DOUBLE 値が生成されます。値の変換は、使用時に実行されます。したがって、次のような式の場合、

```
1 + 2 * $DOUBLE(4.0)
```

InterSystems IRIS は最初に、10 進数値として 1 と 2 を加算します。その後、結果の 3 を \$DOUBLE 形式に変換し、乗算を実行します。結果は \$DOUBLE(12) です。

### G.9.3 丸め

必要に応じて、数値は最も近い表現可能な値に丸められます。丸められる値が、2 つの使用可能な値と同等に近い場合、以下が実行されます。

- ・ \$DOUBLE 値は、IEEE 標準で定義されているとおりに、偶数に丸められます。
- ・ 10 進数値は、0 から離れた値、つまり (絶対値で) 大きい方の値に丸められます。



## G.10 比較演算

### G.10.1 同種の表現

\$DOUBLE(+0) と \$DOUBLE(-0) の比較では、これらの値は同等に扱われます。これは IEEE 標準に準拠しています。\$DOUBLE(+0) または \$DOUBLE(-0) が文字列に変換される場合、両方とも結果が “0” になるので、これは InterSystems IRIS の 10 進数の場合と同じです。

\$DOUBLE(nan) と他の数値 (\$DOUBLE(nan) を含む) の比較では、これらの値は、より大きくない、等しくない、より小さくない、になります。これは IEEE 標準に準拠しています。これは、文字列に変換してからそれらの文字列が等しいかどうかを確認することにより等値比較を行うという、通常の ObjectScript のルールから逸脱しています。

注釈 式 nan は、\$DOUBLE(nan) と同等です。これは、比較が文字列の比較として行われるためです。

注釈 ただし、[\\$LISTSAME](#) では、\$DOUBLE(nan) を含むリスト要素を、\$DOUBLE(nan) を含むリスト要素と同一と見なします。この場合にのみ、[\\$LISTSAME](#) は、等しくない値を等しいと見なします。

### G.10.2 異種の表現

10 進数値と \$DOUBLE 値の比較は、完全に正確です。比較は、いずれの値も丸められることなく実行されます。有限値のみが含まれる場合、これらの比較の答えは、両方の値を文字列に変換し、これらの文字列を既定の照合ルールに基づいて比較した場合に得られる結果と同じになります。

演算子 <, <=, >, および => を含む比較では常に、ブーリアン値、0 または 1 を 10 進数値として生成します。いずれかのオペランドが文字列である場合、そのオペランドは、比較を実行する前に 10 進数値に変換されます。他の数値オペランドは変換されません。前述のとおり、異なる数値型の比較は、完全に正確に実行され、変換は行われません。

文字列の比較演算子 (=, '=, ], ], [, '[, ], ]' など) では、すべての数値オペランドは、比較を行う前にまず文字列に変換されます。

### G.10.3 以下、以上

InterSystems IRIS では、演算子 < および >= はそれぞれ、演算子 ' > および ' < と同様に扱われます。

注意 オペランドのいずれかまたは両方が非数である可能性がある比較で、演算子 <= または >= が使用される場合、IEEE 標準の要求とは異なる結果になります。

A と B のいずれか (または両方) が非数の場合、 $A \geq B$  という式は、以下のように解釈されます。

1. 式は  $A > B$  に変換されます。
2. さらに  $'(A > B)'$  に変換されます。
3. 前述のように、非数を含む比較では、(a) 等しくない、(b) より大きくない、(c) より小さくない、という結果が得られます。したがって、括弧の中の式の結果は、False の値になります。
4. その値を反転させると、True の値が得られます。

注釈  $A \geq B$  という式は、これを  $((A > B) \mid (A = B))$  と表現すれば、書き換えて IEEE で要求される結果を得ることができます。

## G.11 ブーリアン演算

ブーリアン演算 (and、or、not、nor、nand など) では、すべての文字列オペランドが 10 進数に変換されます。数値オペランド (10 進数または \$DOUBLE) は変更されません。

数値 0 は FALSE として扱われ、その他のすべての数値 (\$DOUBLE(nan を含む) および \$DOUBLE(inf)) は TRUE として扱われます。結果は 0 または 1 (10 進数) です。

注釈     \$DOUBLE(-0) も False です。

## G.12 関連項目

- ・ [\\$DECIMAL](#) 関数
- ・ [\\$DOUBLE](#) 関数
- ・ [\\$FNUMBER](#) 関数
- ・ [IEEE-754-2019](#) 標準
- ・ Computing Surveys の David Goldberg 氏著による [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#) (1991 年 3 月発行)