ObjectScript の使用法

Version 2023.1
2024-01-02

InterSystems Corporation　1 Memorial Drive　Cambridge MA 02142　www.intersystems.com

ObjectScript の使用法
InterSystems IRIS Data Platform　Version 2023.1　2024-01-02

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

目次

1 ObjectScript の概要 1
1.1 機能 1
1.2 言語の概要 2
1.3 コマンドと関数の呼び出し 2
1.3.1 文とコマンド 3
1.3.2 関数 3
1.3.3 式 4
1.3.4 変数 4
1.3.5 演算子 5

2 構文規則 7
2.1 大文字と小文字の区別 7
2.1.1 識別子 7
2.1.2 キーワード名 8
2.1.3 クラス名 8
2.1.4 ネームスペース名 8
2.2 Unicode 8
2.2.1 Unicode の文字 8
2.2.2 リストの圧縮 9
2.3 空白 9
2.4 コメント 10
2.4.1 INT コードのルーチンおよびメソッドに使用するコメント 10
2.4.2 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 10
2.4.3 メソッド・コードの外部のクラス定義に使用するコメント 11
2.5 リテラル 11
2.5.1 文字列リテラル 12
2.5.2 数値リテラル 13
2.6 識別子 14
2.6.1 識別子の句読点文字 14
2.7 ラベル 15
2.7.1 ラベルの使用法 15
2.7.2 ラベル付けされたコード・セクションの終了 16
2.8 ネームスペース 17
2.8.1 拡張参照 18
2.9 予約語 18

3 データ型とデータ値 19
3.1 文字列 19
3.1.1 最大文字列長 19
3.1.2 NULL 文字列と $CHAR(0) 20
3.1.3 引用符のエスケープ 20
3.1.4 連結文字列 20
3.1.5 文字列比較 21
3.1.6 ビット文字列 21
3.2 数値 22
3.2.1 数値の基本 22
3.2.2 数値のキャノニック形式 24
3.2.3 数値としての文字列 25

3.2.4 数字の連結 26
3.2.5 浮動小数点数 26
3.2.6 科学的記数法 28
3.2.7 極端に大きな数字 28
3.3 オブジェクト 30
3.4 永続多次元配列 (グローバル) 31
3.5 未定義の値 31
3.6 ブーリアン値 32
3.7 日付 33

4 変数 35
4.1 変数のカテゴリ 35
4.1.1 添え字付き変数 35
4.1.2 オブジェクト・プロパティ 36
4.2 ローカル変数 37
4.2.1 名前付け規約 37
4.2.2 ローカル変数の範囲 38
4.2.3 オブジェクト値 39
4.3 プロセス・プライベート・グローバル 39
4.3.1 名前付け規約 40
4.3.2 プロセス・プライベート・グローバルのリスト 41
4.4 グローバル 42
4.5 特殊変数 43
4.6 変数のタイプと変換 44
4.7 変数の宣言 45

5 演算子と式 47
5.1 演算子と式の概要 47
5.1.1 演算子記号の表 47
5.1.2 演算子の優先順位 49
5.1.3 式 51
5.1.4 代入 53
5.2 文字列から数値への変換 53
5.2.1 数値文字列 54
5.2.2 非数値文字列 55
5.3 算術演算子 55
5.3.1 10 進数および $DOUBLE 浮動小数点数 55
5.3.2 単項プラス演算子 (+) 56
5.3.3 加算演算子 (+) 56
5.3.4 単項マイナス演算子 (-) 57
5.3.5 減算演算子 (-) 57
5.3.6 乗算演算子 (*) 58
5.3.7 除算演算子 (/) 58
5.3.8 整数除算演算子 (¥) 59
5.3.9 モジュロ演算子 (#) 59
5.3.10 指数演算子 (**) 59
5.4 数値関係演算子 61
5.4.1 二項に対するより小さい関係演算子 (<) 61
5.4.2 二項に対するより大きい関係演算子 (>) 61
5.4.3 以下演算子 (<= または '>) 62
5.4.4 以上演算子 (>= または '<) 62
5.5 論理比較演算子 63

5.5.1 論理演算子の優先順位 63
5.5.2 単項否定演算子 (') 63
5.5.3 二項論理積演算子 (& または &&) 64
5.5.4 二項論理和演算子 (! または ||) 65
5.5.5 論理積否定 (NAND) 演算子 ('&) 66
5.5.6 論理和否定 (NOR) 演算子 ('!) 67
5.6 文字列連結演算子 (_) 67
5.6.1 連結エンコード文字列 68
5.7 文字列関係演算子 69
5.7.1 二項等値演算子 (=) 69
5.7.2 不等関係演算子 ('=) 70
5.7.3 二項包含演算子 ([) 70
5.7.4 非包含演算子 ('[) 70
5.7.5 二項後続関係演算子 (]) 71
5.7.6 非後続演算子 (']) 71
5.7.7 二項前後関係演算子 (]]) 72
5.7.8 非前後関係演算子 (']]) 72
5.8 パターン・マッチ演算子 (? または '?) 72
5.8.1 ObjectScript のパターン・マッチング 73
5.8.2 パターンの発生回数の指定 76
5.8.3 複数パターンの指定 77
5.8.4 組み合わせパターンの指定 77
5.8.5 不確定パターンの指定 78
5.8.6 交互パターンの指定 (論理 OR) 78
5.8.7 不完全パターンの使用法 79
5.8.8 パターンの複数解釈 79
5.8.9 非マッチ演算 80
5.8.10 パターンの複雑さ 80
5.9 間接演算子 (@) 80
5.9.1 名前間接演算 81
5.9.2 パターン間接演算 82
5.9.3 引数間接演算 83
5.9.4 添え字間接演算 83
5.9.5 $TEXT 引数間接指定 85

6 正規表現 87
6.1 ワイルドカードと修飾子 88
6.2 リテラルと文字の範囲 89
6.3 文字タイプ・メタ文字 90
6.3.1 単一文字タイプ 91
6.3.2 Unicode プロパティ文字タイプ 91
6.3.3 POSIX 文字タイプ 93
6.4 グループ化構文 94
6.5 アンカー・メタ文字 95
6.5.1 文字列の先頭または最後 95
6.5.2 単語境界 96
6.6 論理演算子 97
6.7 文字表現メタ文字 97
6.7.1 16 進、8 進、および Unicode の表現 97
6.7.2 制御文字表現 98
6.7.3 記号名表現 98

6.8 モード 98
6.8.1 正規表現シーケンスのモード 99
6.8.2 リテラルのモード 100
6.9 コメント 101
6.9.1 埋め込みコメント 101
6.9.2 行末コメント 101
6.10 エラー・メッセージ 101

7 コマンド 103
7.1 コマンド・キーワード 103
7.2 コマンド引数 104
7.2.1 複数の引数 105
7.2.2 パラメータおよび後置条件付きの引数 105
7.2.3 引数なしコマンド 106
7.3 コマンド後置条件式 107
7.3.1 後置条件構文 107
7.3.2 後置条件の評価 107
7.4 単一行での複数コマンド 108
7.5 変数割り当てコマンド 108
7.5.1 SET 108
7.5.2 KILL 110
7.5.3 NEW 110
7.6 コード実行コンテキスト・コマンド 110
7.7 コードの呼び出し 111
7.7.1 DO 111
7.7.2 JOB 112
7.7.3 XECUTE 112
7.7.4 QUIT および RETURN 112
7.8 フロー制御コマンド 113
7.8.1 条件付きの実行 113
7.8.2 FOR 114
7.8.3 WHILE と DO WHILE 116
7.9 入出力コマンド 116
7.9.1 表示 (書き込み) コマンド 116
7.9.2 READ 119
7.9.3 OPEN、USE、および CLOSE 119

8 呼び出し可能なユーザ定義コードモジュール 121
8.1 プロシージャ、ルーチン、サブルーチン、関数、メソッドの概要 122
8.1.1 ルーチン 122
8.1.2 サブルーチン 123
8.1.3 関数 124
8.2 プロシージャの定義 125
8.2.1 プロシージャの呼び出し 125
8.2.2 プロシージャの構文 126
8.2.3 プロシージャ変数 127
8.2.4 パブリック・プロシージャとプライベート・プロシージャ 129
8.3 パラメータ渡し 130
8.3.1 値渡し 131
8.3.2 参照渡し 131
8.3.3 可変個数のパラメータ 132
8.4 プロシージャ・コード 134

8.5 プロシージャ範囲内の間接指定、XECUTE コマンド、および JOB コマンド 136
8.6 プロシージャ内でのエラー・トラップ 136
8.7 従来のユーザ定義コード 137
8.7.1 サブルーチン 137
8.7.2 関数 138
9 ObjectScript マクロとマクロ・プリプロセッサ 143
9.1 マクロの使用 143
9.1.1 カスタム・マクロの作成 143
9.1.2 カスタム・マクロの保存 146
9.1.3 マクロの呼び出し 146
9.1.4 外部マクロ (インクルード・ファイル) の参照 146
9.2 システム・プリプロセッサ・コマンド・リファレンス 147
9.2.1 #; 148
9.2.2 #def1arg 148
9.2.3 #define 149
9.2.4 #dim 151
9.2.5 #else 152
9.2.6 #elseIf 153
9.2.7 #endif 153
9.2.8 #execute 153
9.2.9 #if 154
9.2.10 #ifDef 155
9.2.11 #ifNDef 155
9.2.12 #import 156
9.2.13 #include 157
9.2.14 #noshow 158
9.2.15 #show 158
9.2.16 #sqlcompile audit 159
9.2.17 #sqlcompile mode 159
9.2.18 #sqlcompile path 159
9.2.19 #sqlcompile select 161
9.2.20 #undef 162
9.2.21 ##; 163
9.2.22 ##beginquote ...##EndQuote 163
9.2.23 ##continue 163
9.2.24 ##expression 164
9.2.25 ##function 166
9.2.26 ##lit 167
9.2.27 ##quote 167
9.2.28 ##quoteExp 168
9.2.29 ##sql 168
9.2.30 ##stripq 169
9.2.31 ##unique 169
9.3 システムにより提供されるマクロの使用 170
9.3.1 システムにより提供されるマクロをアクセス可能にする方法 170
9.3.2 システムにより提供されるマクロのリファレンス 170
9.4 マクロが拡張される条件 173
10 埋め込み SQL 175
10.1 埋め込み SQL 175

11 多次元配列 177
11.1 多次元配列の概要 177
11.1.1 多次元ツリー構造 177
11.1.2 スパース多次元ストレージ 178
11.1.3 多次元配列の種類 178
11.2 多次元配列の操作 178
11.3 詳細 179

12 文字列演算 181
12.1 基本的な文字列演算と関数 181
12.1.1 $EXTRACT の高度な機能 182
12.2 区切り文字列演算 183
12.2.1 高度な $PIECE 関数 184
12.3 リスト構造文字列演算 184
12.3.1 スパース・リストおよびサブリスト 186
12.4 リストと区切り文字列の比較 186
12.4.1 リストの利点 186
12.4.2 区切り文字列の利点 186

13 ロック管理 189
13.1 システム全体での現在のロックの管理 189
13.1.1 ロック・テーブルを使用したロックの表示 190
13.1.2 ロック・テーブルを使用したロックの削除 192
13.2 ^LOCKTAB ユーティリティ 192
13.3 待機ロック要求 193
13.3.1 配列ノードに対するロック要求のキュー 194
13.3.2 ECP ローカルおよびリモート・ロック要求 195
13.4 デッドロックの回避 195

14 トランザクション処理 197
14.1 アプリケーションでのトランザクション管理 197
14.1.1 トランザクション・コマンド 197
14.1.2 トランザクションでの LOCK の使用 198
14.1.3 トランザクションでの $INCREMENT と $SEQUENCE の使用法 199
14.1.4 アプリケーション内でのトランザクション・ロールバック 199
14.1.5 アプリケーション内のトランザクション処理の例 200
14.2 自動トランザクション・ロールバック 200
14.3 トランザクション処理に関するシステム全体の問題 201
14.3.1 トランザクション処理でのバックアップとジャーナリング 201
14.3.2 非同期エラーの通知 201
14.4 現在のすべてのトランザクションの一時停止 202

15 エラー処理 203
15.1 TRY-CATCH メカニズム 203
15.1.1 TRY-CATCH と一緒に THROW を使用する 204
15.1.2 $$$ThrowOnError および $$$ThrowStatus マクロの使用法 205
15.1.3 %Exception.SystemException と %Exception.AbstractException クラスの使用 206
15.1.4 TRY-CATCH を使用する際の他の考慮事項 206
15.2 %Status エラー処理 207
15.2.1 %Status エラーの作成 208
15.2.2 %SYSTEM.Error 209
15.3 従来のエラー処理 209
15.3.1 従来のエラー処理の概要 209

15.3.2 $ZTRAP でのエラー処理 213
15.3.3 $ETRAP でのエラー処理 217
15.3.4 エラー・ハンドラによるエラー処理 220
15.3.5 強制エラー 221
15.3.6 ターミナル・プロンプトでのエラー処理 222
15.4 アプリケーション・エラーのログ作成 225
15.4.1 %ETN を使用したアプリケーション・エラーのログ作成 225
15.4.2 管理ポータルを使用したアプリケーション・エラー・ログの表示 226
15.4.3 %ERN を使用したアプリケーション・エラー・ログの表示 226
16 コマンド行ルーチンのデバッグ 229
16.1 保護されたデバッグ・シェル 229
16.1.1 制限されるコマンドと関数 230
16.2 ObjectScript デバッガによるデバッグ 231
16.2.1 ブレークポイントとウォッチポイントの使用法 232
16.2.2 ブレークポイントとウォッチポイントの設定 232
16.2.3 ブレークポイントとウォッチポイントを無効にする 236
16.2.4 ブレークポイントとウォッチポイントの実行を遅らせる 237
16.2.5 ブレークポイントとウォッチポイントの削除 237
16.2.6 シングル・ステップ・ブレークポイントの動作 237
16.2.7 実行のトレース 238
16.2.8 割り込みキーと Break コマンド 239
16.2.9 現在のデバッグ環境情報の表示 240
16.2.10 デバッグ・デバイスの使用法 241
16.2.11 ObjectScript デバッガの例 242
16.2.12 ObjectScript デバッガ・エラーの理解 243
16.3 BREAK コマンドによるデバッグ 243
16.3.1 ルーチンの実行を中断する引数なし BREAK の使用法 244
16.3.2 ルーチンの実行を中断する引数付き BREAK の使用法 244
16.3.3 ターミナル・プロンプトのプログラム・スタック情報表示 245
16.3.4 FOR ループおよび WHILE ループ 246
16.3.5 BREAK あるいはエラー後の実行の再開 247
16.3.6 ターミナル・プロンプトで使用する NEW コマンド 248
16.3.7 ターミナル・プロンプトで使用する QUIT コマンド 248
16.3.8 InterSystems IRIS エラー・メッセージ 249
16.4 スタックを表示する %STACK の使用法 249
16.4.1 %STACK の実行 249
16.4.2 プロセス実行スタックの表示 250
16.4.3 スタックの表示の理解 251
16.5 その他のデバッグ・ツール 255
16.5.1 $SYSTEM.OBJ.ShowReferences によるオブジェクトへの参照の表示 255
16.5.2 エラー・トラップ・ユーティリティ 255

図一覧

図 15-1: コール・スタックのフレーム 211
図 15-2: $ZTRAP エラー・ハンドラ 217
図 15-3: $ETRAP エラー・ハンドラ 219

テーブル一覧

テーブル 3-1: 日付形式 33
テーブル 4-1: ObjectScript の型の変換規則 44
テーブル 5-1: ObjectScript 演算子 47
テーブル 5-2: パターン・コード 75
テーブル 7-1: 表示形式 117
テーブル 7-2: 値の表示方法 118
テーブル 14-1: トランザクション・コマンド 198
テーブル 16-1: ターミナル・プロンプトのスタック・エラー・コード 246
テーブル 16-2: %STACK ユーティリティ情報 251
テーブル 16-3: フレーム・タイプと使用可能な値 251

1

ObjectScript の概要

ObjectScript は、InterSystems IRIS® で複雑なビジネス・アプリケーションを迅速に開発できるように設計されたオブジェクト・プログラミング言語です。以下のさまざまなアプリケーションに適しています。

- ビジネス・ロジック
- アプリケーションの統合
- データ処理

ObjectScript のソース・コードは、InterSystems IRIS 仮想マシン内で実行されるオブジェクト・コードにコンパイルされます。このオブジェクト・コードは、文字列操作やデータベース・アクセスなど、ビジネス・アプリケーションで一般的な処理を実行するために、高度に最適化されています。ObjectScript プログラムは、InterSystems IRIS がサポートするすべてのプラットフォーム間で完全に移植可能です。

以下のいずれのコンテキストでも、ObjectScript を使用できます。

- ターミナルのコマンド行からインタラクティブに使用
- InterSystems IRIS オブジェクト・クラスのメソッドの実装言語として使用
- InterSystems IRIS 内に含まれ実行される個別プログラムである ObjectScript ルーチンを生成するために使用
- InterSystems SQL のストアド・プロシージャとトリガに対する実装言語として使用

ObjectScript を詳細に学習するには、以下も参照してください。

- ObjectScript チュートリアル – 大半の言語要素に対しインタラクティブな概要について
- ObjectScript リファレンス – 個別のコマンドと関数について

1.1 機能

以下は、ObjectScript の主な機能です。

- 文字列と連動する強力な組み込み関数
- メソッド、プロパティ、多態を含むオブジェクトのネイティブ・サポート
- アプリケーションでコントロール・フローを管理する多様なコマンド
- 入出力デバイスを処理するコマンド・セット
- ローカルとグローバル (永続) の両方で多次元なスパース配列 (要素番号が連続しない配列) をサポート

- 効率的な埋め込み SQL のサポート
- 間接指定、実行時の評価とコマンド実行のサポート

1.2 言語の概要

以下は、ObjectScript の主な要素の概要です。

ObjectScript は予約語が定義されていないため、任意の単語を識別子 (変数名など) として自由に使用できます。このために、ObjectScript は、組み込みのコマンド一式と特殊文字 (関数名の “$” 接頭語など) を使用して、他の言語要素から識別子を区別します。

例えば、変数に値を割り当てる場合、SET コマンドを使用できます。

ObjectScript

```
SET x = 100
WRITE x
```

ObjectScript では、以下のプログラムに示すように、任意の有効な名前を識別子名として使用できます (推奨ではありません)。このプログラムと上記のプログラムは、機能的に同じです。

ObjectScript

```
SET SET = 100
WRITE SET
```

ObjectScript のコンポーネントには、コマンド名や関数名のように、大文字と小文字が区別されないものがあります。また、変数名、ラベル、クラス名、およびメソッド名のように、大文字と小文字が区別される ObjectScript のコンポーネントもあります。詳細は、″大文字と小文字の区別″ を参照してください。

ObjectScript のほとんどの場所で、空白を挿入または省略できます。ただし、空白の使用法のうち 2 つには重要な意味があります。

1. コマンドとその引数は少なくとも 1 つのスペースで区切る必要があります。
2. 各コマンド行は少なくとも 1 つのスペースでインデントする必要があります。コマンドを行の最初の文字位置から開始したり、続行することはできません。

また、コメントもインデントする必要があります。ただし、ラベルは行の最初の文字位置で開始する必要があります。マクロ・プリプロセッサ文など、他の一部の構文は、行の最初の文字位置で開始できます。詳細は、″空白″ を参照してください。

ObjectScript は、コマンド・ターミネータ文字や行ターミネータ文字を使用しません。

1.3 コマンドと関数の呼び出し

一番単純な ObjectScript 構文は、以下のような形式で、式でコマンドを呼び出します。

ObjectScript

```
WRITE x
```

これは、変数 x に WRITE コマンドを実行します (これにより、x の値を表示します)。上記の例で、x は 式 です。ObjectScript 式は、値を算出するために評価される 1 つ以上の “トークン” です。それぞれのトークンは、リテラル、変数、1 つ以上の演算の結果 (2 つの数字の加算の合計など)、関数の評価からの返り値、あるいはこれらのうちの組み合わせなどになります。文の有効な構文には、そのコマンド、関数、式、および演算子が関係しています。

1.3.1 文とコマンド

ObjectScript プログラムは、多くの文から構成されています。それぞれの文が、プログラムで実行する特定の動作を定義します。文には、コマンドと引数があります。

以下に ObjectScript 文があります。

ObjectScript

```
SET x="World"
WRITE "Hello",!,x
```

WRITE はコマンドです。これは、その名前のとおり、引数として指定された内容を現在の主出力デバイスに書き込みます。この場合、WRITE は、リテラル文字列 “Hello”、“!” 文字 (改行またはキャリッジ・リターンを発行する WRITE コマンドに固有の記号演算子)、および実行時に最新の値に置き換えられるローカル変数 x の 3 つの引数を書き込みます。引数はコンマで区切られ、引数と引数の間に空白を挿入することもできます (いくつかの制限があります)。空白については、本ドキュメントの ″構文規則″ の章で説明しています。

ほとんどの ObjectScript のコマンド (および関数や特殊変数) には、長い形式と短い (省略された) 形式 (通常は 1、2 文字) があります。例えば、次のプログラムは前述のプログラムと同じものですが、コマンド名が省略された形で使用されています。

ObjectScript

```
S x="World"
W "Hello",!,x
```

短い形式のコマンド名は、長いコマンド名の入力を好まない開発者が使用する 1 つの手段です。長い形式とまったく同じコマンドを示します。このドキュメントでは、長い形式のコマンド名を使用します。完全なリストは、″ObjectScript リファレンス″ の ″ObjectScript で使用する省略形″ を参照してください。

コマンドの詳細は、″コマンド″ の章、あるいは ″ObjectScript リファレンス″ のそれぞれのリファレンス・ページを参照してください。

1.3.2 関数

関数とは、処理 (例えば、文字列をその等価の ASCII コード値に変換するなど) を実行して値を返すコードのことです。関数は、コマンド行内で呼び出されます。この呼び出しにより関数にパラメータ値が渡され、このパラメータ値を使用して処理を実行します。次に関数は単一の値 (結果) を呼び出し元コマンドに返します。式 を使用できる場所ならどこでも関数を使用できます。

InterSystems IRIS では、大量のシステム指定の関数 (“内部” 関数とも呼ばれる) を提供しており、これらを変更することはできません。これらの関数は、1 つのドル記号 (“$”) で始まり、パラメータが括弧で囲まれているため、識別可能です。パラメータが指定されない場合でも、括弧は必須です(特殊変数名も単一のドル記号で始まりますが、括弧は含まれません)。

多くのシステム指定の関数名には、省略形があります。このドキュメントでは、完全な関数名が使用されます。省略形は関数の参照ページに示され、″ObjectScript リファレンス″ の ″ObjectScript で使用する省略形″ には完全なリストが用意されています。

関数は常に値を返します。通常、返り値はコマンドに渡され、`SET namelen=$LENGTH("Fred Flintstone")` または `WRITE $LENGTH("Fred Flintstone")` のようになるか、あるいは別の関数に指定されて `WRITE`

`$LENGTH($PIECE("Flintstone^Fred","^",1))` のようになります。返り値の受取先を指定しないと、通常 <SYNTAX> エラーが発生します。ただし、一部の関数では、返り値の受取先を指定することは必須ではありません。関数の実行によって行われる処理 (ポインタの移動など)、または関数のパラメータの 1 つを設定することは、関連する処理です。このような場合、DO または JOB コマンドを使用して、返り値を受け取らずに関数を呼び出すことができます。例えば、`DO $CLASSMETHOD(clname,clmethodname,singlearg)` のように指定します。

関数には、1 つのパラメータまたは複数のパラメータを指定することも、パラメータを指定しないこともできます。関数のパラメータは定位置にあり、コンマで区切られます。多くのパラメータはオプションです。パラメータを省略する場合、InterSystems IRIS はそのパラメータの既定値を使用します。パラメータは定位置のため、指定されたパラメータのリスト内でパラメータを通常省略することはできません。場合によっては ($LISTTOSTRING など)、パラメータのリスト内でパラメータを省略し、プレースホルダのコンマを指定できます。オプションのパラメータについては、最後の指定されたパラメータの右側にプレースホルダのコンマを指定する必要はありません。

大半の関数は、同じパラメータを繰り返して指定できません。しかし、$CASE、$CHAR、$SELECT は例外として使用できます。

一般的に、パラメータはリテラル、変数、または別の関数の返り値として指定できます。パラメータをリテラルとして指定しなければいけない場合もあります。ほとんどの場合、関数パラメータとして指定するには、変数をまず定義する必要があります。そうしないと、<UNDEFINED> エラーが発生します。一部のケースでは ($DATA など)、パラメータ変数を定義しなくてもよい場合があります。

一般的に、関数パラメータは関数に値を指定する入力パラメータです。関数は入力パラメータとして指定された変数の値を変更しません。関数は値を返すと共に出力パラメータを設定する場合もあります。例えば、$LISTDATA は、指定された位置にリスト要素があるかどうかを示すブーリアン値を返します。また、(オプションで) リスト要素の値に対して 3 番目のパラメータを設定します。

すべての関数は SET コマンド (`SET x=$LENGTH(y)` など) の右側で指定できます。一部の関数は SET コマンドの左側でも指定できます (`SET $LIST(list,position,end)=x` など)。SET の左側で指定可能な関数は、参照ページの構文ブロックでそのように識別されます。

システム指定の関数は、InterSystems IRIS の一部として提供されます。"ObjectScript ランゲージ・リファレンス" では、システム指定の関数のそれぞれについて説明しています。クラスで提供される関数はメソッドと呼ばれます。InterSystems IRIS で提供されるメソッドについては、"インターシステムズ・クラス・リファレンス" で説明されています。

システム関数の他に、ObjectScript は ユーザ定義関数 (“外部” 関数ともいいます) もサポートします。また、ユーザ定義関数の定義と呼び出しの詳細は、"ユーザ定義コード" を参照してください。

1.3.3 式

式は、1 つの値を算出するために評価されるトークン一式です。例えば、リテラル文字列 “hello” は式です。また `l + 2` も式です。x などの変数、`$LENGTH()` などの関数、`$ZVERSION` などの特殊変数も式として評価されます。

プログラムでは、式をコマンドや関数の引数として使用します。

ObjectScript

```
SET x = "Hello"
WRITE x,!
WRITE 1 + 2,!
WRITE $LENGTH(x),!
WRITE $ZVERSION
```

1.3.4 変数

ObjectScript で変数は、実行時の値が格納される場所の名前です。変数は、SET コマンドなどで定義する必要がありますが、決まった形式は必要ありません。ObjectScript の変数に決まった形式はありません。つまり、変数にはデータ型が

割り当てられていないため、任意のデータ値を取ることができます(互換性を維持するために、$DOUBLE 関数を使用して、形式の決まっていない浮動小数点数を特定の数値データ型形式に変換できます)。

ObjectScript は、さまざまな種類の変数をサポートします。

- ローカル変数 ー 変数を作成した InterSystems IRIS プロセスからのみアクセス可能でプロセス終了時に自動的に削除される変数。ローカル変数にはどのネームスペースからでもアクセスできます。
- プロセス・プライベート・グローバル ー InterSystems IRIS プロセスからのみアクセス可能でプロセス終了時に削除される変数。プロセス・プライベート・グローバルには、どのネームスペースからでもアクセスできます。プロセス・プライベート・グローバルは、サイズの大きいデータ値を一時的に格納する場合に特に便利です。
- グローバル ー InterSystems IRIS データベースに格納される永続変数。グローバルは、すべてのプロセスからアクセス可能であり、そのグローバル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、各ネームスペースに固有です。
- 配列変数 ー 1 つあるいはそれ以上の添え字を持つ変数。すべてのユーザ定義変数は、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、オブジェクト・プロパティを含め配列として使用されます。
- 特殊変数 (システム変数とも呼ばれる) ー InterSystems IRIS 操作環境における特定の状況の値を含む組み込み変数一式の 1 つ。すべての特殊変数は定義されます。InterSystems IRIS では、すべての特殊変数を初期値 (場合によっては NULL 文字列値) に設定します。特殊変数にはユーザが設定できるものと、InterSystems IRIS でのみ設定できるものがあります。特殊変数は配列変数ではありません。
- オブジェクト・プロパティ ー オブジェクトの特定のインスタンスに関連し、格納される値。

ObjectScript は、変数に対する、あるいは変数間でのさまざまな処理をサポートします。変数については、本ドキュメントの ″変数″ の章でさらに詳しく説明しています。

1.3.5 演算子

ObjectScript は、多くの組み込み演算子を定義します。演算子には、加算 (“+”) および乗算 (“*”) といった算術演算子、論理演算子、パターン・マッチ演算子などがあります。詳細は、本ドキュメントの “演算子と式” の章を参照してください。

2
構文規則

この章では、InterSystems IRIS® Data Platform の ObjectScript 構文の基本的な規則について説明します。

2.1 大文字と小文字の区別

ObjectScript には、大文字と小文字を区別する場合とそうでない場合があります。通常、ObjectScript のユーザ定義部は、大文字と小文字を区別しますが、キーワードは区別しません。

- 大文字と小文字の区別あり：変数名 (ローカル、グローバル、およびプロセス・プライベート・グローバル) および変数の添え字、クラス名、メソッド名、プロパティ名、プロパティのインスタンス変数の i% 序文、ルーチン名、マクロ名、マクロ・インクルード・ファイル (.inc ファイル) 名、ラベル名、ロック名、パスワード、埋め込みコード指示文マーカ文字列、埋め込み SQL ホスト変数名。
- 大文字と小文字を区別しない：コマンド名、関数名、特殊変数名、ネームスペース名 (下記参照)、ユーザ名およびロール名、プリプロセッサ指示文 (#include など)、文字コード (LOCK、OPEN、または USE)、キーワード・コード ($STACK)、パターン・マッチ・コード、埋め込みコード指示文 (&html、&js、&sql)。%ZLANG ルーチンをカスタマイズすることによって追加するカスタム言語要素は大文字と小文字を区別しませんが、作成するときは大文字を使用する必要があります。これらの言語要素を参照するときは、大文字でも小文字でもかまいません。テキスト分析のインデックス作成では、小文字に変換することでテキストを正規化しているので、ドメイン名も含め、大半の NLP 値は大文字と小文字を区別しません。
- 通常は大文字と小文字を区別しない：デバイス名、ファイル名、ディレクトリ名、ディスク・ドライブ名で、大文字と小文字を区別するかどうかはプラットフォームによる。指数記号は、通常大文字と小文字を区別しません。大文字の “E” は、常に有効な指数記号です。小文字の “e” は、**%SYSTEM.Process** の ScientificNotation() メソッドを使用している現在のプロセス、または **Config.Miscellaneous** クラスの ScientificNotation プロパティを使用しているシステム全体に対して、有効または無効に構成できます。

2.1.1 識別子

ユーザ定義の 識別子 (変数、ルーチン名、ラベル名) は、大文字と小文字を区別します。String、string、STRING はすべて異なる変数を指しています。グローバル変数名は、ユーザが定義する場合も、あるいはシステムが提供する場合も大文字と小文字を区別します。

注釈　　これに対して、InterSystems IRIS SQL 識別子では大文字と小文字を区別しません。

2.1.2 キーワード名

コマンド、関数、システム変数キーワード (およびその省略形) は、大文字と小文字を区別しません。例えば、Write、write、WRITE、W、w は同じ Write コマンドを示しています。

2.1.3 クラス名

クラスに関連するすべての識別子 (クラス名、プロパティ名、メソッド名など) は大文字と小文字を区別します。しかし、一意性を確保するためには、このような名前では大文字と小文字は区別されないと見なします。つまり、2 つのクラス名は大文字小文字だけでは区別できません。

2.1.4 ネームスペース名

ネームスペース名は、大文字と小文字を区別しません。つまり、ネームスペース名は、大文字と小文字を好きなように組み合わせて入力することができます。ただし、InterSystems IRIS は常に、ネームスペース名を大文字で格納することに注意してください。したがって、InterSystems IRIS は、ユーザが指定した小文字ではなく、大文字でネームスペース名を返す場合があります。ネームスペース名の名前付け規約の詳細は、"ネームスペース" を参照してください。

2.2 Unicode

InterSystems IRIS は、Unicode 国際文字セットをサポートします。Unicode 文字は 16 ビット文字であり、ワイド文字と呼ばれることもあります。$ZVERSION 特殊変数 (`Build nnnU`) および $SYSTEM.Version.IsUnicode() メソッドは、InterSystems IRIS インストールで Unicode がサポートされていることを示すものです。

ほとんどの用途に、InterSystems IRIS は Unicode 基本多言語面 (16 進数 0000 から FFFF) のみをサポートします。これには、最も一般的に使用される国際文字が含まれています。内部的に、InterSystems IRIS では UCS-2 エンコーディングを使用します。これは基本多言語面の場合、UTF-16 と同じです。$WCHAR、$WISWIDE、および関連する関数を使用することで、Unicode 基本多言語面に存在しない文字で作業できます。

InterSystems IRIS は標準の UTF-16 エンコードと同様に、1 文字あたり 16 ビット (2 バイト) を割り当てることで、Unicode 文字列をメモリにエンコードします。ただし、Unicode 文字列をグローバルに保存する際に、すべての文字の数値が 255 以下の場合、InterSystems IRIS は 1 文字あたり 8 ビット (1 バイト) を使用して文字列を格納します。文字列に 255 より大きな数値の文字が含まれる場合、InterSystems IRIS は圧縮アルゴリズムを適用して、文字列がストレージ内で占有する領域を減らします。

Unicode と UTF-8 間の変換、および他の文字エンコーディングへの変換については、$ZCONVERT 関数を参照してください。ZZDUMP を使用して、文字列の 16 進エンコーディングを表示できます。$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。

2.2.1 Unicode の文字

InterSystems IRIS では、Unicode 文字を使用できる名前もありますが、Unicode 文字を含めることができない名前もあります。Unicode 文字は、255 より大きな 10 進文字コード値のアルファベット文字として定義されます。例えば、小文字のギリシャ文字ラムダは Unicode 文字で $CHAR(955) になります。

InterSystems IRIS では、以下の例外を除き、どのような場合でも Unicode 文字を使用できます。

- 変数名 : ローカル変数名には Unicode 文字を使用できます。一方、グローバル変数名とプロセス・プライベート・グローバル変数名には、Unicode 文字を使用できません。あらゆるタイプの変数の添え字が Unicode 文字で指定できます。

- データベースを暗号化する際に管理者が使用するユーザ名とパスワードには Unicode 文字を使用できません。

注釈　日本語ロケールでは、InterSystems IRIS の名前におけるアクセント記号付きラテン文字がサポートされていません。日本語の名前には、(日本語の文字の他に) ラテン文字 A-Z と a-z (65-90 と 97-122)、およびギリシャ語の大文字 (913-929 と 931-937) を使用できます。

2.2.2 リストの圧縮

ListFormat は、$LIST のエンコードされた文字列に格納する際、Unicode 文字列を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。圧縮形式は、InterSystems IRIS によって自動的に処理されます。圧縮形式がサポートされてい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。

2.3 空白

特定の環境では、ObjectScript は構文的に意味のあるものとして空白を処理します。指定がない限り、空白はブランク、タブ、改行と同じ意味を示します。以下はその規則の概要です。

- コードの各行または 1 行コメントの先頭には空白を記述する必要があります。以下の場合には先頭の空白が不要となります。
  - ラベル (タグまたはエントリ・ポイントとも呼ばれる): ラベルは直前の空白文字なしで列 1 にて記述する必要があります。行にラベルがある場合、そのラベルと同一行上の他のコードやコメントの間には空白が必要です。ラベルがパラメータ・リストを有する場合、ラベル名とパラメータ・リストの最初の括弧の間にある空白をなくすこともできます。パラメータ・リスト内のパラメータの前、間、または後には空白を入れることができます。
  - マクロ指示文: #define などのマクロ指示文は直前の空白文字なしで列 1 に記述できます。これは推奨の規約となりますが、マクロ指示文の前の空白は認められています。
  - 複数行コメント: 複数行コメントの最初の行の前には 1 つ以上のスペースを配置する必要があります。複数行コメントの 2 行目以降には、先頭に空白を置く必要はありません。
  - 空白行: 行に文字がない場合は、スペースを含める必要はありません。空白文字のみで構成される行も認められており、コメントとして扱われます。
- コマンドと最初の引数の間には、必ずスペース文字を 1 つだけ記述する必要があります (タブ文字ではありません)。コマンドで後置条件を使用する場合、コマンドとその後置条件との間にはスペース文字を記述しません。
- 後置条件式にスペースがある場合、式全体を括弧で囲む必要があります。
- 組になったコマンド引数の間には、いくつもの空白を置くことができます。
- 1 行にコードと 1 行のコメントが含まれる場合は、その間に空白を置く必要があります。
- 通常、それぞれのコマンドは独自の行に記述されますが、同じ行に複数のコマンドを入力することができます。この場合、これらのコマンドの間には空白が必要です。引数がないコマンドの後には、空白を 2 つ (スペース文字を 2 つ、タブ文字を 2 つ、またはスペース文字とタブ文字をそれぞれ 1 つずつ) 記述する必要があります。これら必須の空白の後に、さらに空白を記述してもかまいません。

2.4 コメント

コードでコメントを使用してインラインのドキュメントを提供するのは適切な方法です。コメントは、コードの変更またはメンテナンス時に大切な情報源になります。ObjectScript では、いくつかの種類のコメントをサポートしており、さまざまな場所で使用できます。

- INT コードのルーチンおよびメソッドに使用するコメント
- 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
- メソッド・コード以外のクラス定義に使用するコメント

2.4.1 INT コードのルーチンおよびメソッドに使用するコメント

ObjectScript コードは MAC コードとして記述され、その MAC コードから INT (中間) コードが生成されます。MAC コードに記述したコメントは通常、対応する INT コードで利用できます。ZLOAD コマンドを使用して INT コード・ルーチンをロードしてから、ZPRINT コマンドまたは $TEXT 関数を使用して、以下のコメントを含む INT コードを表示できます。以下のタイプのコメントを使用できます。これらのコメントはすべて 2 列目以降から開始する必要があります。

- `/* */` 複数行コメントは、1 行または複数行にわたって記述できます。`/*` は、行の最初の要素として、または他の要素の後に配置できます。`*/` は、行の最後の要素として、または他の要素の前に配置できます。`/* */` の間のすべての行が INT コードに表示されます。これには、`/*` または `*/` のみで構成される行も含まれますが、完全に空白の行は含まれません。複数行コメント内の空白行は INT コードから省略されるため、行数に影響する可能性があります。
- `//` コメントは、行の残りの部分がコメントであることを示します。このコメントは、行の最初の要素として、または他の要素の後に配置できます。
- `;` コメントは、行の残りの部分がコメントであることを示します。このコメントは、行の最初の要素として、または他の要素の後に配置できます。
- `;` コメントの特殊なケースである `;;` コメントは、ルーチンがオブジェクト・コードとして配信される場合にのみ、$TEXT 関数で使用できます。行内でこのコメントの前にコマンドがない場合に、$TEXT 関数でのみ使用可能です。

  注釈　InterSystems IRIS は、`;;` コメントをオブジェクト・コード (実際に解釈され、実行されるコード) に保持するため、このコメントを挿入するとパフォーマンスに影響します。そのため、ループ内には配置しないでください。

複数行コメント (`/* comment */`) はコンマ区切り文字前後の、コマンドもしくは関数の引数の間に置くことができます。複数行コメントを引数内に置いたり、コマンドのキーワードと最初の引数の間、もしくは関数キーワードと最初の括弧の間に置くことはできません。同一行で複数行コメントを 2 つのコマンドの間に置くことはできます。その場合、コマンドを区切るために必要な 1 つの空白として機能します。同一行で複数行コメント (*/) の直後にコマンドを続けたり、同一行で単数行コメントを続けることもできます。行中に `/* comment */` を挿入する例を以下に示します。

ObjectScript

```
WRITE $PIECE("Fred&Ginger"/* WRITE "world" */,"&",2),!
WRITE "hello",/* WRITE "world" */" sailor",!
SET x="Fred"/* WRITE "world" */WRITE x,!
WRITE "hello"/* WRITE "world" *// WRITE " sailor"
```

2.4.2 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

以下のコメント・タイプは MAC コードに記述できますが、対応する INT コードでは別の動作となります。

- #; コメントは、任意の列から開始できますが、その行の最初の要素でなければなりません。#; コメントは、INT コードには表示されません。コメントもコメント・マーカ (#;) も INT コードには表示されず、空白行も保持されません。このため、#; コメントによって、INT コードの行番号が変わる可能性があります。
- ##; コメントは任意の列から開始できます。また、その行で最初の要素になることができ、他の要素に続けることもできます。##; コメントは、INT コードには表示されません。##: は、ObjectScript コード内や埋め込み SQL コード内、あるいは #define、#deflarg、または ##continue の各マクロ・プリプロセッサ指示文と同一行に使用できます。

  ##; コメントが列 1 から開始される場合、コメントもコメント・マーカ (##;) も INT コードには表示されず、空白行も保持されません。ただし、##; コメントが列 2 以降から開始される場合、コメントもコメント・マーカ (##;) も INT コードには表示されませんが、空白行は保持されます。この使用法では、##; コメントによって INT コードの行番号が変わることはありません。
- /// コメントは、任意の列から開始できますが、その行の最初の要素でなければなりません。/// が列 1 から開始される場合、INT コードには表示されず、空白行も保持されません。/// が列 2 以降から開始される場合、このコメントは INT コードに表示され、// コメントであるかのように扱われます。

2.4.3 メソッド・コードの外部のクラス定義に使用するコメント

メソッド定義の外部のクラス定義で、いくつかの種類のコメントを使用できます。これらのコメントはすべて任意の列から開始できます。

- // コメントおよび /* */ コメントは、クラス定義で使用するコメントです。
- /// コメントには、その直後のクラスまたはクラス・メンバのクラス・リファレンスのコンテンツを記述します。クラス自体に対しては、クラス定義の開始前に配置される /// コメントには、クラスリファレンスのコンテンツとしてそのクラスの説明が記述されます。このコンテンツは、クラスに関する説明のキーワード値にもなります。クラス内では、メンバの直前 (クラス定義の開始位置以降、または前のメンバの後) に配置される /// コメントにはすべて、そのメンバのクラス・リファレンスのコンテンツが記述されます。複数行の記述は、HTML の 1 つのブロックとして扱われます。/// コメントの規則およびクラス・リファレンスの詳細は、"クラスの定義と使用" の "クラスの定義とコンパイル" の章にある "クラス・ドキュメントの作成" または "インターシステムズ・クラス・リファレンス" の %CSP.Documatic のエントリを参照してください。

2.5 リテラル

リテラルとは、以下の "Hello" と "5" のように、一連の文字で構成して特定の文字列や数値を表すようにした定数値です。

ObjectScript

```
SET x = "Hello"
SET y = 5
```

ObjectScript は、2 種類のリテラルがあります。

- 文字列リテラル
- 数値リテラル

2.5.1 文字列リテラル

文字列リテラルは、0 個以上の文字を引用符で区切って記述したリテラルです（文字列リテラルと異なり、数値リテラルでは区切り文字を必要としません）。ObjectScript の文字列リテラルでは、区切り文字として二重引用符を使用し（例えば `"myliteral"`）、InterSystems SQL の文字列リテラルでは、区切り文字として一重引用符を使用します（例えば `'myliteral'`）。このような区切り文字としての引用符は、文字列の長さには算入されません。

文字列リテラルには、スペース文字や制御文字を初めとして、任意の文字を使用できます。文字列リテラルの長さは、バイト数ではなく、文字列を構成する文字の数で表します。文字列に 0 から 255 のコードの文字のみが含まれる場合（Latin-1 または ASCII 拡張文字とも呼ばれる）、各文字は 8 ビット（1 バイト）となります。文字列に 255 より大きなコードの文字が 1 文字以上含まれる場合（Unicode またはワイド文字とも呼ばれる）、各文字は 16 ビット（2 バイト）となります。文字列の文字の格納に使用されるバイト数を表示するには、次の例に示すように、ZZDUMP コマンドを使用します。

最大文字列サイズは 3,641,144 文字です。

以下の例は、8 ビット文字による文字列、16 ビット Unicode 文字（ギリシャ文字）による文字列、および 8 ビット文字と 16 ビット Unicode 文字を組み合わせた文字列を示しています。

ObjectScript

```
  DO AsciiLetters
  DO GreekUnicodeLetters
  DO CombinedAsciiUnicode
  RETURN
AsciiLetters()
  SET a="abc"
  WRITE a
  WRITE !,"the length of string a is ",$LENGTH(a)
  ZZDUMP a
  QUIT
GreekUnicodeLetters()
  SET b=$CHAR(945)_$CHAR(946)_$CHAR(947)
  WRITE !!,b
  WRITE !,"the length of string b is ",$LENGTH(b)
  ZZDUMP b
  QUIT
CombinedAsciiUnicode()
  SET c=a_b
  WRITE !!,c
  WRITE !,"the length of string c is ",$LENGTH(c)
  ZZDUMP c
  QUIT
```

入力不能文字を文字列に使用することもできます。以下の Unicode の例のように、$CHAR 機能を使用すると、入力不能文字を指定できます。

ObjectScript

```
  SET greekstr=$CHAR(952,945,955,945,963,963,945)
  WRITE greekstr
```

表示不能文字を文字列に使用することもできます。このような文字として、出力不能文字（制御文字）があります。WRITE コマンドでは、出力不能文字がボックス記号で表示されます。WRITE コマンドを使用すると、制御文字が実行されます。以下の例の文字列では、NULL 文字（$CHAR(0)）、タブ文字（$CHAR(9)）、キャリッジ・リターン文字（$CHAR(13)）に入れ替わる出力可能な文字を使用しています。

ObjectScript

```
  SET a="a"_$CHAR(0)_"b"_$CHAR(9)_"c"_$CHAR(13)_"d"
  WRITE !,"the length of string a is ",$LENGTH(a)
  ZZDUMP a
  WRITE !,a
```

プログラムで実行している WRITE では印刷不能文字として表示される制御文字が、ターミナルのコマンドラインから実行した WRITE では実行されることがあります。ベル文字（$CHAR(7)）と垂直タブ文字（$CHAR(11)）はその例です。

引用符文字 (") を文字列に含めるには、次の例に示すようにその文字を重複して使用します。

ObjectScript

```
SET x="This quote"
SET y="This "" quote"
WRITE x,!," string length=",$LENGTH(x)
ZZDUMP x
WRITE !!,y,!," string length=",$LENGTH(y)
ZZDUMP y
```

値を含まない文字列は NULL 文字列と呼ばれ、2 つの引用符 ("") で示されます。NULL 文字列は、定義済みの値と見なされます。その長さは 0 です。以下の例にあるように、NULL 文字列は、NULL 文字 ($CHAR(0)) で構成した文字列とは異なります。

ObjectScript

```
SET x=""
WRITE "string=",x," length=",$LENGTH(x)," defined=",$DATA(x)
ZZDUMP x
SET y=$CHAR(0)
WRITE !!,"string=",y," length=",$LENGTH(y)," defined=",$DATA(y)
ZZDUMP y
```

文字列の詳細は、このドキュメントの“データ型とデータ値”の章にある "文字列" を参照してください。

2.5.2 数値リテラル

数値リテラルとは、ObjectScript が数値として計算する値のことです。このリテラルでは区切り文字が不要です。InterSystems IRIS では、数値リテラルがキャノニック形式 (数値リテラルの最も単純な数値形式) に変換されます。

ObjectScript

```
SET x = ++0007.00
WRITE "length:     ",$LENGTH(x),!
WRITE "value:      ",x,!
WRITE "equality:   ",x = 7,!
WRITE "arithmetic: ",x + 1
```

引用符で区切った文字列リテラルとして数値を表現することもできます。数値の文字列リテラルはキャノニック形式に変換されませんが、演算操作では数値として使用できます。

ObjectScript

```
SET y = "++0007.00"
WRITE "length:     ",$LENGTH(y),!
WRITE "value:      ",y,!
WRITE "equality:   ",y = 7,!
WRITE "arithmetic: ",y + 1
```

詳細は、このドキュメントの“データ型とデータ値”の章にある "数値としての文字列" を参照してください。

ObjectScript では、以下の値を使用した値が数値として扱われます (ここにない値を使用すると数値扱いにはなりません)。

値	数量
0 から 9 までの数字。	1 個以上の任意の個数。
符号演算子、単項マイナス演算子 (-)、単項プラス演算子 (+)。	個数は任意ですが、他のすべての文字の前に記述する必要があります。
decimal_separator 文字 (既定ではピリオドまたは小数点文字、ヨーロッパのロケールではコンマ)。	最大 1 個。
文字 “E” (科学的記数法で使用)。	最大 1 個。2 つの数値の間に記述する必要があります。

これらの文字の使用と解釈の詳細は、“データ型とデータ値” の章の ″数値の基本″ を参照してください。

ObjectScript は、以下の種類の数値を使用できます。

- 整数 (100、0、-7 などの自然数)。
- 小数 : 小数 (3.767、.0442 などの実数)。ObjectScript では、InterSystems IRIS 標準の浮動小数点数 ($DECIMAL 数) または IEEE 倍精度浮動小数点数 ($DOUBLE 数) のいずれかの内部表現で小数を記述できます。詳細は、″$DOUBLE 関数″ を参照してください。
- 科学的記数法 : 指数表現の数字 (2.8E2 など)。

2.6 識別子

識別子 とは、変数、ルーチン、またはラベルの名前です。一般に、正当な識別子は文字や数字で構成されています。まれな例外を除き、句読点文字は識別子に使用できません。識別子は大文字と小文字を区別します。

ユーザ定義のコマンド、関数、および特殊変数に対する名前付け規約は、識別子の名前付け規約よりも制限 (許可された文字のみ) を受けます。″専用のシステム/ツールおよびユーティリティ″ の ″^%ZLANG ルーチンによる言語の拡張″ を参照してください。

ローカル変数、プロセス・プライベート・グローバル、およびグローバルの名前付け規則は、このドキュメントの “変数” の章に示されています。

2.6.1 識別子の句読点文字

特定の識別子は、1 つ以上の句読点文字を持つことができます。以下はその概要です。

- 識別子の最初の文字をパーセント (%) 記号にすることができます。% 文字で始まる InterSystems IRIS 名 (%Z または %z で始まるものは除く) は、システム要素として予約されています。詳細は、″サーバ側プログラミングの入門ガイド″ の “識別子のルールとガイドライン” を参照してください。
- (ローカル変数名ではなく) グローバルまたはプロセス・プライベート・グローバルの名前は、1 つ以上のピリオド (.) 文字を持つ場合があります。ルーチン名は、1 つ以上のピリオド (.) 文字を持つ場合があります。ピリオドを、識別子の最初または最後の文字にすることはできません。

グローバルおよびプロセス・プライベート・グローバルは、次に示すような 1 文字以上のキャレット (^) 文字で識別されることに注意してください。

```
  Globals:                                  Process-Private Globals:
^globname                                   ^||ppgname
^|""|globname                               ^|"^"|ppgname
^|"mynspace"|globname                  ^|"^",""|ppgname
^["mynspace"]globname                ^["^"]ppgname
```

これらの接頭文字は変数名を構成するものではなく、ストレージのタイプと(グローバルの場合は)このストレージで使用されるネームスペースを識別します。実際の名前は、最後の垂直バーまたは閉じ角括弧の後から始まります。

2.7 ラベル

ObjectScript コードの行は、オプションでラベル(タグとして)を含むことができます。ラベルは、コード内の行の場所を参照するハンドルとして機能します。ラベルはインデントされない識別子で、列 1 で指定されます。すべての ObjectScript コマンドは、インデントする必要があります。

ラベルには以下の名前付け規約があります。

- 最初の文字は、英数字またはパーセント記号 (%) でなくてはなりません。ラベルは、先頭を数字とすることができる唯一の ObjectScript 名であることに注意してください。2 番目以降の文字はすべて英数字でなくてはなりません。ラベルは Unicode 文字を含めることができます。
- 最大長は 31 文字です。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。ラベル参照は、ラベルの最初の 31 文字のみと一致します。ただし、ラベルまたはラベル参照 (最初の 31 文字だけでなく) の文字はすべて、ラベル文字の名前付け規約に従う必要があります。
- ラベルは大文字と小文字を区別します。

注釈　CREATE PROCEDURE や CREATE TRIGGER などの SQL コマンドで指定した ObjectScript コードのブロックではラベルを使用できます。この場合は、ラベルの先頭文字の前の 1 カラムに、接頭文字としてコロン (:) を記述します。ラベルのそれ以降の部分は、ここで説明する命名と使用方法の要件に従います。

ラベルではパラメータの括弧を含めたり省略することができます。括弧を含めた場合、その中を空にしたり、またはコンマ区切りのパラメータ名を 1 つ以上含めることができます。括弧付きのラベルによりプロシージャ・ブロックが識別されます。

行は、ラベルだけで構成できます。ラベルの次に 1 以上のコマンド、またはラベルの次に 1 コメントなどです。コマンドまたはコメントが、同じ行のラベルの後に続く場合、スペースまたはタブ記号を使って両者を区別する必要があります。

以下は、すべて一意のラベルです。

ObjectScript

```
maximum
Max
MAX
86
agent86
86agent
%control
```

$ZNAME 関数を使用すれば、ラベル名を検証することができます。ラベル名を検証する場合には、パラメータの括弧を含めないでください。

ZINSERT コマンドを使用すれば、ラベル名をソース・コードに挿入できます。

2.7.1 ラベルの使用法

ラベルは、コード・セクションの認識およびフロー・コントロールの管理に役立ちます。

DO および GOTO コマンドでは、ターゲットの場所をラベルとして指定することができます。$ZTRAP 特殊変数では、エラー・ハンドラの場所をラベルとして指定することができます。JOB コマンドでは、実行されるルーチンをラベルとして指定することができます。

また、ソース・コード行を特定するには、PRINT、ZPRINT、ZZPRINT、ZINSERT、ZREMOVE、および ZBREAK コマンド、さらには $TEXT 関数によりラベルを使用します。

ただし、CATCH コマンドと同一のコード行にて、または TRY ブロックと CATCH ブロックの間にてラベルを指定することはできません。

2.7.2 ラベル付けされたコード・セクションの終了

ラベルはエントリ・ポイントを提供しますが、カプセル化されたコード・ユニットを定義しません。これは、ラベル付けされたコードが実行されれば、実行が停止またはリダイレクトされない限り、次のラベル付けされたコード・ユニットへと実行が続くことを意味します。コード・ユニットの実行を停止するには、以下の 3 つの方法があります。

- コード実行が QUIT または RETURN に遭遇する。
- コード実行が TRY の閉じ中括弧に (“}”) に遭遇する。これが発生すると、関連付けられた CATCH ブロックに続くコードの次の行へと実行を継続します。
- コード実行が次のプロシージャ・ブロック (パラメータの括弧付きラベル) に遭遇する。括弧付きラベル行と遭遇すると、括弧内にパラメータがない場合においても、実行が停止します。

以下の例では、“label0” 以下から “label1” 以下までコードが連続して実行されます。

ObjectScript

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
          WRITE "Finished Routine0",! }
  ELSE {DO label1
          WRITE "Finished Routine1",! }
  QUIT
label0
  WRITE "In Routine0",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine0",!
label1
  WRITE "In Routine1",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine1",!
```

以下の例では、ラベル付けされたコード・セクションが QUIT または RETURN コマンドのいずれかで終了します。これによって実行が止まります。RETURN は常に実行を止めて、QUIT は現在のコンテキストの実行を止めることに注意してください。

ObjectScript

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
          WRITE "Finished Routine0",! }
  ELSE {DO label1
          WRITE "Finished Routine1",! }
  QUIT
label0
  WRITE "In Routine0",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1
      QUIT }
  WRITE "Quit the FOR loop, not the routine",!
  WRITE "At the end of Routine0",!
  QUIT
  WRITE "This should never print"
label1
  WRITE "In Routine1",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
```

```
  WRITE "At the end of Routine1",!
  RETURN
  WRITE "This should never print"
```

以下の例では、2 番目と 3 番目のラベルがプロシージャ・ブロック (パラメータの括弧にて指定されたラベル) を指定します。プロシージャ・ブロックのラベルと遭遇すると、実行が停止します。

ObjectScript

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
          WRITE "Finished Routine0",! }
  ELSE {DO label1
          WRITE "Finished Routine1",! }
  QUIT
label0
  WRITE "In Routine0",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine0",!
label1()
  WRITE "In Routine1",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine1",!
label2()
  WRITE "This should never print"
```

2.8 ネームスペース

ネームスペース名は、明示的なネームスペース名または暗黙的なネームスペース名になります。明示的なネームスペース名は大文字と小文字を区別しません。入力時においての大文字と小文字に関係なく、常に大文字で格納と返送が行われます。

明示的なネームスペース名では、最初の文字は、英字またはパーセント記号 (%) に限られます。残りの文字は、英字、数字、ハイフン (-)、またはアンダースコア (_) にする必要があります。この名前は、255 文字以内にする必要があります。

InterSystems IRIS で、明示的なネームスペース名をルーチン名またはクラス名に変換するとき (キャッシュしたクエリ・クラス/ルーチン名の作成時など) は、句読文字が % = p、_ = u、- = d のように小文字に置き換えられます。暗黙のネームスペース名には、これら以外の句読文字が使用されていることがありますが、そのようなネームスペース名の変換では、それらの句読文字が小文字の "s" に置き換えられます。したがって、7 つの句読点文字は次のようになります。@ = s、: = s、/ = s、¥ = s、[= s、] = s、^ = s。

予約済みのネームスペース名は、%SYS、BIN、BROKER、および DOCUMATIC です。

InterSystems SQL CREATE DATABASE コマンドを使用した場合、SQL データベースの作成により、対応する InterSystems IRIS ネームスペースが作成されます。

InterSystems IRIS のインスタンスでは、ネームスペースはディレクトリとして存在します。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("")
```

ネームスペースの使用については、"サーバ側プログラミングの入門ガイド" の "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については、"システム管理ガイド" の "ネームスペースの構成" を参照してください。

2.8.1 拡張参照

拡張参照とは他のネームスペースにあるエンティティへの参照のことです。ネームスペース名は引用符で囲んだ文字列リテラル、ネームスペース名に解決する変数、暗黙的なネームスペース名、または現在のネームスペースを指定するプレースホルダである NULL 文字列 (″″) として指定できます。拡張参照には、以下の 3 つのタイプがあります。

- 拡張グローバル参照: 他のネームスペースのグローバル変数を参照します。次の構文形式をサポートしています。`^["namespace"]global` および `^|"namespace"|global`。詳細は、このドキュメントの “変数” の章の ″グローバル変数″ のセクションを参照してください。
- 拡張ルーチン参照: 他のネームスペースのルーチンを参照します。
  - DO コマンド、$TEXT 関数、およびユーザ定義関数は次の構文形式をサポートしています。`|"namespace"|routine`。
  - JOB コマンドは次の構文形式をサポートしています。`routine|"namespace"|`、`routine["namespace"]`、または `routine:"namespace"`。

  これらのすべての場合において、拡張ルーチン参照の開始には ^ (キャレット) 文字を使用し、指定したエンティティが (ラベルやオフセットではなく) ルーチンであることを示します。このキャラットはルーチン名の一部ではありません。例えば、`DO ^|"SAMPLES"|fibonacci` は、SAMPLES ネームスペースに置かれている fibonacci という名のルーチンを呼び出します。`WRITE $$fun^|"SAMPLES"|house` コマンドは、SAMPLES ネームスペースに置かれている、ルーチン house のユーザ定義関数 fun() を呼び出します。
- 拡張 SSVN 参照: 他のネームスペースの構造化システム変数 (SSVN) を参照します。次の構文形式をサポートしています。`^$["namespace"]ssvn` および `^$|"namespace"|ssvn`。詳細は、^$GLOBAL、^$LOCK、および ^$ROUTINE 構造化システム変数を参照してください。

もちろん、すべての拡張参照は名前によって明示的に、もしくは NULL 文字列のプレースホルダ指定により、現在のネームスペースを指定できます。

2.9 予約語

ObjectScript に予約語はありません。したがって、あらゆる有効な識別子を、変数名、関数名、ラベルとして使用できます。同時に、コマンド名、関数名、他の文字列である識別子の使用を避けるのが適切です。また、ObjectScript コードは埋め込み SQL をサポートするため、関数、オブジェクト、変数、SQL 予約語である他のエンティティの名前を避けることが賢明です。そうしなければ、障害が発生する原因となります。

3
データ型とデータ値

ObjectScript は、タイプのない言語です (変数タイプを宣言する必要はありません)。変数は、文字列、数値、オブジェクト値を持つことができます。それでも ObjectScript では、異種データをいつ使用するかを把握しておくことは重要な情報となります。

3.1 文字列

文字列とは、文字、数字、句読点などの文字の組み合わせを、1 組の引用符 (") で囲んだものです。

ObjectScript

```
SET string = "This is a string"
WRITE string
```

文字列についての項目は以下のとおりです。

- 最大文字列長
- NULL 文字列と $CHAR(0)
- 引用符のエスケープ
- 連結文字列
- 文字列比較
- ビット文字列

3.1.1 最大文字列長

InterSystems IRIS® Data Platform は、3,641,144 文字の最大文字列長をサポートします。この最大文字列長を超えると、<MAXSTRING> エラーになります。

プロセスで文字列が使用されると、文字列用のメモリを、そのプロセス用のパーティション・メモリ領域から割り当てるのではなく、オペレーティング・システムの malloc() バッファから割り当てます。したがって、実際の文字列の値へのメモリ割り当ては、プロセス・パラメータ (プロセスあたりの最大メモリ (KB)) あたりの最大メモリの設定に制限されず、プロセスの $STORAGE 値に影響しません。

3.1.2 NULL 文字列と $CHAR(0)

- SET mystr="" : NULL または空の文字列を設定します。文字列が定義され、ゼロの長さで、データは含まれません。

  ObjectScript

  ```
  SET mystr=""
  WRITE "defined:",$DATA(mystr),!
  WRITE "length: ",$LENGTH(mystr),!
  ZZDUMP mystr
  ```

- SET mystr=$CHAR(0) : 文字列を NULL 文字に設定します。文字列が定義され、1 の長さで、16 進数値 00 の 1 文字が含まれます。

  ObjectScript

  ```
  SET mystr=$CHAR(0)
  WRITE "defined:",$DATA(mystr),!
  WRITE "length: ",$LENGTH(mystr),!
  ZZDUMP mystr
  ```

これら 2 つの値は同じではないことに注意してください。ただし、ビット文字列ではこれらの値は同一のものとして処理されます。

InterSystems SQL では、これらの値が独自の方法で解釈されます。"InterSystems SQL の使用法" の “言語要素” の章にある "NULL および空文字列" を参照してください。

3.1.3 引用符のエスケープ

" (二重引用符) 文字の前にもう 1 つ " 文字を置くと、文字列内のリテラルとして引用符を組み込むことができます。

ObjectScript

```
SET string = "This string has ""quotes"" in it."
WRITE string
```

ObjectScript 文字列リテラルに、他のエスケープ文字シーケンスはありません。

リテラルの引用符は、その他のインターシステムズのソフトウェアでは、別のエスケープ・シーケンスを使用して指定されます。これに該当するエスケープ・シーケンスについての表は、$ZCONVERT 関数を参照してください。

3.1.4 連結文字列

_ 連結演算子を使用して、2 つの文字列を 1 つに結合できます。

ObjectScript

```
SET a = "Inter"
SET b = "Systems"
SET string = a_b
WRITE string
```

連結演算子を使用して、出力不能文字を文字列に組み込むことができます。以下の文字列には、改行 ($CHAR(10)) 文字が含まれています。

ObjectScript

```
SET lf = $CHAR(10)
SET string = "This"_lf_"is"_lf_"a string"
WRITE string
```

注釈　出力不能文字の表示方法は、ディスプレイ・デバイスにより決定されます。例えば、ターミナルとブラウザでは、改行文字や他のポジショニング文字の表示方法が異なります。さらに、ブラウザが異なれば、ポジショニング文字 $CHAR(11) および $CHAR(12) の表示方法も異なります。

InterSystems IRIS エンコード文字列 (ビット文字列、リスト構造文字列、JSON 文字列) には、連結演算子の使用に関する制限があります。詳細は、"連結エンコード文字列" を参照してください。

数値を結合する際は、他の事項も考慮する必要があります。詳細は、"数字の連結" を参照してください。

3.1.5 文字列比較

等号 (=) および不等号 ('=) の演算子を使用して、2 つの文字列を比較できます。文字列の等値比較では大文字と小文字が区別されます。これらの演算子を使用して、文字列を数字と比較する際には注意が必要です。それは、この比較が文字列比較であり、数値比較ではないからです。したがって、キャノニック形式の数字を含む文字列のみが、対応する数字と等価になります。("-0" はキャノニック形式の数ではありません)。詳細は、以下の例を参照してください。

ObjectScript

```
WRITE "Fred" = "Fred",!  // TRUE
WRITE "Fred" = "FRED",!  // FALSE
WRITE "-7" = -007.0,!    // TRUE
WRITE "-007.0" = -7,!    // FALSE
WRITE "0" = -0,!         // TRUE
WRITE "-0" = 0,!         // FALSE
WRITE "-0" = -0,!        // FALSE
```

<、>、<=、>= などの演算子を使用して、文字列比較を実行することはできません。これらの演算子は数値として文字列を扱い、常に数値比較を実行します。非数値文字列は、これらの演算子を使用して比較された場合、いずれも数値 0 が割り当てられます。

3.1.5.1 大文字小文字の区別と文字列比較

文字列の等値比較では大文字と小文字が区別されます。$ZCONVERT 関数を使用すれば、比較する文字列の文字をすべて大文字に、またはすべて小文字に変換することができます。文字ではない符号や記号は不変となります。

小文字の文字形式しか有していない文字も多少あります。例えば、ドイツ語の eszett ($CHAR(223)) は小文字としてのみ定義されます。これを大文字に変換しても、結果は同じ小文字となります。これにより、英数字文字列を大文字または小文字のみの文字列に変換する場合は、常に小文字に変換することをお勧めします。

3.1.6 ビット文字列

ビット文字列は、番号付きビットとブーリアン値の論理セットを表します。文字列内のビットには、ビット番号 1 から始まる番号が付けられます。ブーリアン値 1 に明示的に設定されていない番号付きビットは、0 と評価されます。したがって、明示的に設定された番号付きビット以外のものを参照すると、ビット値 0 が返されます。

- ビット値は、ビット文字列関数 $BIT および $BITLOGIC を使用することでのみ、設定できます。
- ビット値は、ビット文字列関数 $BIT、$BITLOGIC、および $BITCOUNT を使用することでのみ、アクセスできます。

ビット文字列には論理長があります。これは、0 または 1 のいずれかに明示的に設定された最上位のビット位置です。この論理長には、$BITCOUNT 関数を使用することでのみアクセスできます。ただし、通常は、アプリケーション・ロジックでは論理長を使用しないでください。ビット文字列関数の場合、未定義のグローバル変数またはローカル変数は、ビット値 0 を返す指定された番号付きビットを持ち、$BITCOUNT 値 が 0 であるビット文字列と同等です。

ビット文字列は、内部形式を持つ通常の ObjectScript 文字列として保存されます。この内部文字列の表現には、ビット文字列関数ではアクセスできません。この内部形式のため、ビット文字列の文字列長は、文字列内のビット数に関する情報を判断する際に意味がありません。

ビット文字列の内部形式という理由により、連結演算子をビット文字列と共に使用することはできません。これを実行しようとすると、<INVALID BIT STRING> エラーが返されます。

同じ状態 (同じブーリアン値を持つ) の 2 つのビット文字列が、異なる内部文字列表現を持つ場合があります。したがって、文字列表現は、アプリケーション・ロジックで調査したり、比較したりしないでください。

ビット文字列関数の場合、未定義の変数として指定されたビット文字列は、すべてのビットが 0 で長さが 0 のビット文字列と同等です。

通常の文字列とは異なり、ビット文字列では、空の文字列と文字 $CHAR(0) は互いに等しく、0 ビットを表すものとして処理されます。これは、$BIT では非数値の文字列がすべて 0 として処理されるからです。したがって、以下のようになります。

ObjectScript

```
SET $BIT(bstr1,1)=""
SET $BIT(bstr2,1)=$CHAR(0)
SET $BIT(bstr3,1)=0
IF $BIT(bstr1,1)=$BIT(bstr2,1) {WRITE "bitstrings are the same"} ELSE {WRITE "bitstrings different"}

WRITE $BITCOUNT(bstr1),$BITCOUNT(bstr2),$BITCOUNT(bstr3)
```

トランザクション中にグローバル変数で設定されたビットは前の値に戻され、その後にトランザクションのロールバックが行われます。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前の文字列長または前の内部文字列表現に戻りません。ローカル変数は、ロールバック操作によって元に戻りません。

論理ビットマップ構造は、ビット文字列の配列で表現できます。この配列の各要素が固定のビット数を持つ 1 つの「チャンク」を表しています。未定義はすべてのビットが 0 のチャンクと同等であるため、配列はスパースになることがあります。すべてのビットが 0 のチャンクを表す配列要素は、存在している必要は一切ありません。この理由および上記のロールバック動作のため、アプリケーション・ロジックでは、ビット文字列の長さや $BITCOUNT(str) または $BITCOUNT(str,0) を使用してアクセスできる 0 値のビットのカウントに依存しないようにする必要があります。

3.2 数値

数値に関連するトピックは、以下のとおりです。

- 数値の基本
- 数値のキャノニック形式
- 数値としての文字列
- 数字の連結
- 浮動小数点数
- 科学的記数法
- 極端に大きな数字

3.2.1 数値の基本

数値リテラルに句読点は必要ありません。任意の有効な数値文字を使用して数を指定できます。InterSystems IRIS では、数値が構文的に有効と評価されると、その数値がキャノニック形式に変換されます。

数値リテラルの構文要件は以下のようになります。

- 0 ～ 9 の 10 進数を含むことができ、少なくともそれらの数値文字のいずれかを含んでいること。先頭または末尾にゼロを使用できます。ただし、InterSystems IRIS で数値をキャノニック形式に変換すると、先頭の整数ゼロが自動的に削除されます。このため、先頭の整数ゼロが重要な数値は、文字列として入力する必要があります。例えば、米国の郵便番号は、02142 のように先頭が整数ゼロの可能性もあるため、数値ではなく文字列として処理する必要があります。
- 任意のシーケンスで、先頭に任意の数のプラス符号とマイナス符号を使用できます。ただし、プラス符号とマイナス符号は、科学的記数法の文字 “E” を除き、他の任意の文字の後に位置することはできません。数値式では、非符号文字の後の符号は加算または減算として評価されます。数値文字列では、非符号文字の後の符号は非数値文字として評価され、そこで文字列の数値部が終わります。

  InterSystems IRIS では、PlusSign および MinusSign プロパティ値を現在のロケールに使用して、これらの符号文字 (既定では “+” および “–”) を決定します。これらの符号文字はロケールに依存します。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")
  ```

- 小数点区切り文字を 1 つ使用できます。数値式では、2 つ目の小数点区切りを用いると <SYNTAX> エラーとなります。数値文字列では、2 つ目の小数点区切りは最初の非数値文字として評価され、そこで文字列の数値部が終わります。小数点区切り文字は、数値式の最初の文字または最後の文字となることができます。小数点区切り文字の選択はロケールに依存します。アメリカ形式では、小数点区切りとして既定のピリオド (.) が使用されます。ヨーロッパ形式では、コンマ (,) が使用されます。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")
  ```

- 科学的記数法では、“E” (または “e”) を最大で 1 つ使用して 10 を基数とする指数を指定できます。この科学的記数文字 (“E” または “e”) の前には整数または小数、その後には整数を記述する必要があります。

数値リテラル値は、以下をサポートしていません。

- 数値グループ・セパレータは使用できません。これらはロケールに依存します。アメリカ形式ではコンマが使用され、ヨーロッパ形式ではピリオドが使用されます。$INUMBER 関数を使用して数値グループ・セパレータを削除し、$FNUMBER 関数を使用して数値グループ・セパレータを追加することができます。
- 通貨記号、16 進数文字、その他の非数値文字は使用できません。演算子の前後を除き、空白スペースは使用できません。
- 末尾にプラス符号やマイナス符号を使用できません。ただし、$FNUMBER 関数は、末尾に符号の付いた文字列として数を表示でき、$NUMBER 関数でこの形式の文字列を取得して、先頭に符号の付いた数に変換します。
- 負の数 (借方) として数を表す場合、括弧に入れることはできません。ただし、$FNUMBER 関数は、括弧で囲んだ文字列で負の数を表示でき、$NUMBER 関数でこの形式の文字列を取得して、先頭に負符号の付いた数に変換します。

数または数値式には、1 組みの括弧を使用できます。これらの括弧は数の一部ではありませんが、処理の優先順位を決定します。既定では、InterSystems IRIS は、厳密に左から右の順序ですべての処理を実行します。

3.2.2 数値のキャノニック形式

ObjectScript では、数に対する数値演算のすべてをキャノニック形式で実行します。例えば、数値 +007.00 の長さは 1 になり、文字列 "+007.00" の長さは 7 になります。

InterSystems IRIS で数をキャノニック形式に変換するときには、以下の手順が実行されます。

1. 科学的記数法の指数が解決されます。例えば、3E4 は 30000 に変換され、3E-4 は .0003 に変換されます。
2. 先頭の符号が解決されます。まず、複数の符号が 1 つの符号に解決されます (例えば、2 つのマイナス符号が 1 つのプラス符号になります)。次に、先頭にあるプラス符号が削除されます。$FNUMBER 関数を使用すると、InterSystems IRIS のキャノニック形式の正の数にプラス記号を明示的に指定 (追加) できます。

   注釈　ObjectScript は、先頭のプラス記号とマイナス記号の任意の組み合わせを解決します。SQL では、2 つの連続したマイナス記号は、1 行コメント文字として解析されます。したがって、先頭に 2 つの連続したマイナス記号を付けて SQL で数値を指定すると、SQLCODE -12 エラーになります。

3. 先頭と末尾のゼロがすべて削除されます。これには、1 より小さい小数の先頭の整数ゼロの削除も含まれます。例えば、0.66 は .66 になります。
   - キャノニック形式の小数に整数のゼロを追加するには、$FNUMBER 関数または $JUSTIFY 関数を使用します。.66 は 0.66 になります。
   - キャノニック形式以外の小数から整数ゼロを削除するには、単項プラス演算子を使用して、数値文字列からキャノニック形式の数値への変換を行います。例えば、+$PIECE("65798,00000.66",",",2) のタイムスタンプの秒の小数部00000.66 は .66 になります。

   この変換の一環として、ゼロの小数は 0 に単純化されます。表現方法 (0.0、.0、.000) にかかわらず、すべてのゼロ値は 0 に変換されます。
4. 末尾の小数点区切りが削除されます。
5. -0 は 0 に変換されます。
6. 算術演算と数値の連結が実行されます。InterSystems IRIS は、こうした演算を厳密に左から右の順に実行します。このような演算は、キャノニック形式の数で実行されます。詳細は、後述の "数字の連結" を参照してください。

InterSystems IRIS のキャノニック形式の数は、その他のインターシステムズのソフトウェアで使用されているキャノニック形式の数とは異なります。

- ODBC: 整数ゼロの小数は、1 つのゼロ整数を含む ODBC に変換されます。そのため、.66 と 000.66 は、どちらも 0.66 になります。$FNUMBER 関数または $JUSTIFY 関数を使用すると、InterSystems IRIS のキャノニック形式の小数に整数のゼロを追加できます。
- JSON:先頭の 1 つのマイナス記号のみが許容されます。先頭のプラス記号や複数の記号は許容されません。

  指数は許容されますが、解決されません。3E4 は、3E4 として返されます。

  先頭のゼロは許容されません。末尾のゼロは削除されません。

  整数ゼロの小数には、1 つのゼロ整数が必要です。したがって、.66 と 000.66 は無効な JSON の数値になりますが、0.66 と 0.660000 は有効な JSON の数値となります。

  末尾の小数点記号は許容されません。

  ゼロの値は、変換されません。0.0、-0、および -0.000 は、有効な JSON の数値として変更なしで返されます。

3.2.3 数値としての文字列

文字列を数値として処理する標準的な規則は以下のとおりです。詳細は、このドキュメントの“演算子と式”の章の″文字列から数値への変換″を参照してください。特別な処理については、″非常に大きな数値文字列″を参照してください。

- すべての数値演算では、キャノニック形式の数字を含む文字列が、対応する数字と機能的に同一になります。例えば、"3" = 3、"-2.5" = -2.5 となります。(-0 はキャノニック形式の数ではありません。)
- 算術演算では、非キャノニック形式の数値文字のみを含む文字列が、対応する数字と機能的に同一になります。例えば、"003" + 3 = 6、"++-2.5000" + -2.5 = -5 となります。
- 「より大きい/より小さい」などの比較演算では、非キャノニック形式の数値文字のみを含む文字列が、対応する数字と機能的に同一になります。例えば、次の文は True となります。"003" > 2、"++-2.5000" >= -2.5
- 等値演算 (=、'=) では、非キャノニック形式の数値文字のみを含む文字列が、数値ではなく文字列として扱われます。例えば、次の文は True となります。"003" = "003"、"003" '= 3、"+003" '= "003"

数値としての文字列の解析に関するその他のガイドラインを以下に述べます。

- 混合数値文字列とは、数字で始まり、1 つまたは複数の数値でない文字が続く文字列です。例えば“7 dwarves”です。InterSystems IRIS の数値演算とブーリアン演算 (等値演算は除く) では共に、数値ではない文字を検出するまで混合数値文字列を数値として解析します。数値でない文字を検出した時点で残りの文字列を無視します。以下に混合数値文字列の算術演算の例を示します。

  ObjectScript

  ```
  WRITE "7dwarves" + 2,!   // returns 9
  WRITE "+24/7" + 2,!      // returns 26
  WRITE "7,000" + 2,!      // returns 9
  WRITE "7.0.99" + 2,!     // returns 9
  WRITE "7.5.99" + 2,!     // returns 9.5
  ```

- 非数値文字列とは、数値文字の前に数値ではない文字がある任意の文字列です。空白は数値ではない文字と見なされます。InterSystems IRIS の数値演算とブーリアン演算 (等値演算は除く) では共に、このような文字列を数値 0 (ゼロ) を持つものとして解析します。以下に非数値文字列の算術演算の例を示します。

  ObjectScript

  ```
  WRITE "dwarves 7" + 2,!   // returns 2
  WRITE "+ 24/7" + 2,!      // returns 2
  WRITE "$7000" + 2,!       // returns 2
  ```

- 文字列の接頭語にプラス符号を置いて、等値演算の数値として評価させることができます。数値文字列はキャノニック形式の数値として解析され、非数値文字列は 0 として解析されます。(マイナス符号の接頭語でも等値演算の数値として文字列を評価されます。当然、マイナス符号は 0 以外の値の符号の反転となります。) 以下では、プラス符号により等値演算での数値評価が強制される例を示します。

  ObjectScript

  ```
  WRITE +"7" = 7,!            // returns 1 (TRUE)
  WRITE +"+007" = 7,!         // returns 1 (TRUE)
  WRITE +"7 dwarves" = 7,!    // returns 1 (TRUE)
  WRITE +"dwarves" = 0,!      // returns 1 (TRUE)
  WRITE +"" = 0,!             // returns 1 (TRUE)
  ```

″ObjectScript リファレンス″ に示したとおり、個別のコマンドと関数に対する数値文字列の例外処理は共通となります。

3.2.3.1 非常に大きな数値文字列

通常、数値文字列は ObjectScript の 10 進数値に変換されます。ただし、非常に大きな数値の場合 (9223372036854775807E127 より大きい)、常に数値文字列を 10 進数値に変換できるとは限りません。数値文字列を 10 進数値に変換すると <MAXNUMBER> エラーが発生する場合、InterSystems IRIS では代わりに IEEE バイナリ値に変換します。InterSystems IRIS は次の操作を実行して、数値文字列を数に変換します。

1. 数値文字列を 10 進数の浮動小数点数に変換します。この操作で <MAXNUMBER> エラーが発生する場合は、手順 2 に進みます。それ以外の場合、10 進数値がキャノニック形式の数として返されます。
2. $SYSTEM.Process.TruncateOverflow() メソッド・ブーリアン値を確認します。0 (既定) の場合、手順 3 に進みます。それ以外の場合、オーバーフローする 10 進数値が返されます (メソッドの説明を参照)。
3. 数値文字列を IEEE バイナリ浮動小数点数に変換します。この操作で <MAXNUMBER> エラーが発生する場合は、手順 4 に進みます。それ以外の場合、IEEE バイナリ値がキャノニック形式の数として返されます。
4. $SYSTEM.Process.IEEEError() メソッド・ブーリアン値を確認します。この値に応じて、INF / -INF が返されるか、<MAXNUMBER> エラーが発生します。

3.2.4 数字の連結

連結演算子 (_) を使用して、ある数と別の数を連結できます。InterSystems IRIS は、まずそれぞれの数値をキャノニック形式に変換してから、その結果に対して文字列連結を実行します。したがって、12_34、12_+34、12_--34、12.0_34、12_0034.0、12E0_34 はすべて 1234 になります。12._34 の連結は 1234 になりますが、12_.34 は 12.34 になります。12_-34 の連結は文字列 “12-34” になります。

InterSystems IRIS は、キャノニック形式に変換した後の数に対して、数値の連結と算術演算を実行します。こうした演算は厳密に左から右の順に実行されます。ただし、演算の優先順位を決める括弧が指定されている場合を除きます。以下の例では、この結果について説明しています。

ObjectScript

```
WRITE 7_-6+5 // returns 12
```

この例では、連結によって文字列 "7-6" が返されます。これが、キャノニック形式の数でないことは明白です。InterSystems IRIS は、最初の非数値文字 (埋め込みのマイナス記号) を切り捨てることで、この文字列をキャノニック形式の数に変換します。その後で、このキャノニック形式の数を使用して次の演算 7 + 5 = 12 を実行します。

3.2.5 浮動小数点数

InterSystems IRIS は、浮動小数点数を表すために使用できる 2 つの異なる数値タイプをサポートしています。

- 10 進数の浮動小数点 : 既定では、InterSystems IRIS は、固有の 10 進数の浮動小数点標準 ($DECIMAL 数) を使用して小数を表します。これは、ほとんどの場合の優先形式であり、IEEE バイナリ浮動小数点よりも、高いレベルの精度が得られます。これは、InterSystems IRIS がサポートしているすべてのシステム・プラットフォームで一貫しています。10 進数の浮動小数点は、データ・ベースの値で優先的に使用されます。特に、0.1 などの小数は、10 進数の浮動小数記数法を使用して正確に表現できますが、小数 0.1 (およびほとんどの 10 進数の小数) は IEEE バイナリ浮動小数点では近似値で表現できるだけです。

  内部的に、10 進算術は M*(10**N) 形式の数を使用して実行されます。M は -9223372036854775808 から 9223372036854775807 の整数値を含む整数の仮数で、N は -128 から 127 の整数値を含む 10 進指数です。仮数は 64 ビット符号付整数で表され、指数は 8 ビット符号付バイトで表されます。

  10 進浮動小数点の平均精度は小数桁数 18.96 桁です。仮数が 1000000000000000000 から 9223372036854775807 の小数の精度は正確に 19 桁で、922337203685477581 から 999999999999999999 の有効な小数の精度は正確に 18 桁です。IEEE バイナリ浮動小数点は、精度は低いものの (精度は小数桁数約 15.95 桁)、小数文字列としての IEEE バイナリ表現の正確な無限精度の値は、1000 桁以上の有効小数桁数を持つことができます。

以下の例では、$DECIMAL 関数は小数と 25 桁の整数を取り、19 桁の精度/有効桁数に丸められた小数を返します。

Terminal

```
USER>WRITE $DECIMAL(1234567890.123456781818181)
1234567890.123456782
USER>WRITE $DECIMAL(12345678901234567818181 81)
12345678901234567820000000
```

- IEEE バイナリ浮動小数点 : IEEE 倍精度バイナリ浮動小数点は、小数を表現する業界標準の方法です。IEEE 浮動小数点数は、バイナリ表現でエンコードされます。バイナリ浮動小数点表現は、ほとんどのコンピュータがバイナリ浮動小数点演算のための高速ハードウェアを搭載しているため、通常、高速計算を実行する場合に優先的に使用されます。

  内部的に、IEEE バイナリ算術は S*M*(2**N) 形式の数を使用して実行されます。S は -1 または +1 の値を含む符号で、M は 1 番目と 2 番目のバイナリ・ビット間に 2 進小数点がある 53 ビットのバイナリ小数値を含む仮数で、N は -1022 から 1023 の整数値を含む 2 進指数です。このため、表現は 64 ビットで構成されます。S は 1 つの符号ビット、指数 N は次の 11 ビットに格納され (2 つの追加の値が予約されます)、仮数 M は >=1.0 かつ <2.0 で最後の 52 ビットを含み、合計 53 バイナリ・ビットの精度となります。(M の最初のビットは常に 1 であるため、64 ビット表現に出現する必要はありません。)

  倍精度バイナリ小数点の精度は 53 バイナリ・ビットで、小数点以下約 15.95 桁の精度に相当します。(対応する 10 進数の精度は 15.35 から 16.55 桁の間で変動します。)

  バイナリ表現は、0.1 などの小数をバイナリ小数の有限シーケンスとして表現できないため、10 進数の小数と正確には一致しません。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 の 10 進浮動小数点数とは若干異なる場合があります。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 の 10 進浮動小数点数より精度が高いことを意味するわけではありません。IEEE 浮動小数点数は、InterSystems の10 進数よりも大きい数および小さい数を表現できます。

  以下の例では、$DOUBLE 関数は 17 桁の整数のシーケンスを取り、小数点以下の有効桁数約 16 桁の値を返します。

  Terminal

```
USER>FOR i=12345678901234558:1:12345678901234569 {W $DOUBLE(i),!}
12345678901234558
12345678901234560
12345678901234560
12345678901234560
12345678901234562
12345678901234564
12345678901234564
12345678901234564
12345678901234566
12345678901234568
12345678901234568
12345678901234568
```

  IEEE バイナリ浮動小数点は、特殊な値 INF (無限大) と NAN (非数値) をサポートします。詳細は、"$DOUBLE 関数" を参照してください。

  INF および NAN 値を扱うための `IEEEError` 設定と、$LIST 構造データ内の IEEE 浮動小数点数の圧縮を扱うための `ListFormat` 設定を使用して、IEEE 浮動小数点数の処理を構成できます。どちらも、**%SYSTEM.Process** クラス・メソッド $SYSTEM.Process.IEEEError() を使用して、現在のプロセスに表示および設定できます。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値を設定できます。

$DOUBLE 関数を使用して、InterSystems IRIS 標準の浮動小数点数を IEEE 浮動小数点数に変換できます。また、$DECIMAL 関数を使用して、IEEE 浮動小数点数を InterSystems IRIS 標準の浮動小数点数に変換できます。

既定では、InterSystems IRIS は小数をキャノニック形式に変換して、先頭のゼロをすべて削除します。したがって、0.66 は .66 となります。$FNUMBER (ほとんどの形式) および $JUSTIFY (3 パラメータ形式) は常に、最低 1 桁の整数を伴う小数を返します。これらの関数のいずれかを使用した場合、.66 は 0.66 となります。

$FNUMBER および $JUSTIFY を使用すれば、指定した小数桁数に対して丸めや埋め込みを行うことができます。InterSystems IRIS では、5 以上を切り上げて、4 以下を切り捨てます。埋め込みでは、ゼロを必要な小数桁だけ追加します。少数を整数に丸めた場合、小数点区切り文字は削除されます。整数をゼロ・パディングして小数にした場合、小数点区切り文字が追加されます。

3.2.6 科学的記数法

ObjectScript で科学的記数法 (指数表現) を指定するには、以下の形式を使用します。

```
[-]mantissaE[-]exponent
```

各項目の内容は次のとおりです。

要素	説明
-	オプション－1 つ以上の単項マイナス演算子または単項プラス演算子。これらのマイナス記号文字とプラス記号文字は構成可能です。キャノニック形式への変換では、科学的記数法を解決した後にこれらの演算子が解決されます。
mantissa	整数または小数。先頭と末尾のゼロおよび末尾の小数点区切り文字が含まれる場合があります。
E	指数表現を区切る演算子。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構成できる指数演算子です。
-	オプション－単一の単項マイナス演算子または単項プラス演算子。負数の指数を指定するために使用できます。これらのマイナス記号文字とプラス記号文字は構成可能です。
exponent	指数 (10 のべき乗値) を指定する整数。先頭にゼロを使用できます。小数点区切り文字は記述できません。

例えば、10 を表現するには 1E1、2800 を表現するには 2.8E3、0.05 を表現するには 5E-2 と記述します。

mantissa、E、およびexponent の間には、スペースは許可されません。この構文内では、括弧、連結、およびその他の演算子は許可されません。

科学的記数法を解決することが、数値をキャノニック形式に変換する第一歩であるため、一部の変換処理は利用できません。mantissa と exponent は数値リテラルである必要があり、変数または算術式にすることはできません。exponent は、(最大でも) 1 つのプラスまたはマイナス記号の付いた整数である必要があります。

詳細は、%SYSTEM.Process クラスの ScientificNotation() メソッドの説明を参照してください。

3.2.7 極端に大きな数字

正確に表現できる最大の整数は、19 桁の整数の -9223372036854775808 と 9223372036854775807 です。なぜなら、これらの数値は 64 ビット符号付き整数で表現できる最大数だからです。これを超える整数は、この 64 ビット制限に収まるように自動的に丸められます。以下に例を示します。

ObjectScript

```
SET x=9223372036854775807
WRITE x,!
SET y=x+1
WRITE y
```

同様に、128 を超える指数も、64 ビット符号付き整数の表現になるように自動的に丸められます。以下に例を示します。

ObjectScript

```
WRITE 9223372036854775807e-128,!
WRITE 9223372036854775807e-129
```

この丸めにより、19 桁の整数を超える数になる算術演算は、下位の桁がゼロに置き換えられます。これにより、以下のような状況となる可能性があります。

ObjectScript

```
SET longnum=9223372036854775790
WRITE longnum,!
SET add17=longnum+17
SET add21=longnum+21
SET add24=longnum+24
WRITE add17,!,add24,!,add21,!
IF add24=add21 {WRITE "adding 21 same as adding 24"}
```

InterSystems IRIS でサポートされる最大の 10 進数の浮動小数点数は 9.223372036854775807E145 です。サポートされる最大の $DOUBLE 値 (INFINITY への IEEE オーバーフローが無効であると想定) は 1.797693134862315708 1E308 です。$DOUBLE 型は、InterSystems IRIS の 10 進数型よりも広い範囲の値をサポートしますが、InterSystems IRIS の 10 進数型の方が高精度をサポートします。InterSystems IRIS の 10 進数型の精度は約 18.96 桁 (普通は 19 桁ですが、時折 18 桁の精度) で、$DOUBLE 型の通常の精度は約 15.95 桁 (53 バイナリ桁) です。既定では、InterSystems IRIS は数値リテラルを 10 進数の浮動小数点数として表現します。ただし、数値リテラルが InterSystems IRIS の 10 進数で表現できる数値より大きい場合 (9.223372036854775807E145 より大きい場合)、InterSystems IRIS はその数値を $DOUBLE 表現に自動的に変換します。

1.797693134862315708 1E308 (308 桁または 309 桁) より大きな数値の場合、<MAXNUMBER> エラーが発生します。

10 進数の浮動小数点からバイナリ浮動小数点への自動変換のため、丸めの動作は、9.223372036854775807E145 (整数に応じて 146 桁または 147 桁) で変更されます。詳細は、以下の例を参照してください。

ObjectScript

```
TRY {
  SET a=1
  FOR i=1:1:310 {SET a=a_1 WRITE i+1," digits = ",+a,! }
}
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
}
```

ObjectScript

```
TRY {
  SET a=9
  FOR i=1:1:310 {SET a=a_9 WRITE i+1," digits = ",+a,! }
}
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
}
```

309 桁より大きな数値は数値文字列として表現できます。この値は数値ではなく文字列として保存されるため、丸めも <MAXNUMBER> エラーも適用されません。

ObjectScript

```
SET a="1"
FOR i=1:1:360 {SET a=a_"1" WRITE i+1," characters = ",a,! }
```

許可される最大桁数を超える数値になる指数は、<MAXNUMBER> エラーを生成します。指数の最大許容数は、その指数を受け入れる数のサイズによって異なります。1 桁の仮数の場合、最大指数は 307 または 308 です。

InterSystems IRIS の小数または IEEE 倍精度数を使用する際の大きな数字に関する考慮事項の詳細は、"サーバ側プログラミングの入門ガイド" の付録 "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。

3.3 オブジェクト

オブジェクト値は、メモリ内のオブジェクトのインスタンスを参照します。オブジェクト参照 (OREF) をローカル変数に割り当てることができます。

ObjectScript

```
SET myperson = ##class(Sample.Person).%New()
WRITE myperson
```

オブジェクトのインスタンスのメソッドやプロパティを参照するには、ドット構文を使用します。

ObjectScript

```
SET myperson.Name = "El Vez"
```

変数にオブジェクトが使用されているかどうかを判断するには、$ISOBJECT 関数を使用します。

ObjectScript

```
SET str = "A string"
SET myperson = ##class(Sample.Person).%New()

IF $ISOBJECT(myperson) {
    WRITE "myperson is an object.",!
} ELSE {
    WRITE "myperson is not an object."
}

IF $ISOBJECT(str) {
     WRITE "str is an object."
} ELSE {
    WRITE "str is not an object."
}
```

グローバルにはオブジェクト値を代入できません。代入すると、実行時エラーが発生します。

オブジェクト値を変数 (またはオブジェクトのプロパティ) に割り当てると、以下の例にあるように、オブジェクトの内部参照カウントが増分される悪影響があります。

ObjectScript

```
SET x = ##class(Sample.Person).%New()
WRITE x,!
SET y = ##class(Sample.Person).%New()
WRITE y,!
SET z = ##class(Sample.Person).%New()
WRITE z,!
```

オブジェクトの参照番号が 0 に到達した場合、システムは自動的にそのオブジェクトを破壊します (%OnClose() コールバック・メソッドを呼び出し、そのオブジェクトをメモリから削除します)。

3.4 永続多次元配列 (グローバル)

グローバルは、多次元データベースのスパース配列です。グローバルは、グローバル変数名がキャレット記号 (^) で開始する以外は、他の配列タイプと同一です。データは、任意の数の添え字をつけてグローバルに格納できます。InterSystems IRIS の添え字にはタイプはありません。

以下は、グローバル使用の例です。グローバル ^x を設定すると、その値を検証できます。

ObjectScript

```
SET ^x = 10
WRITE "The value of ^x is: ", ^x,!
SET ^x(2,3,5) = 17
WRITE "The value of ^x(2,3,5) is: ", ^x(2,3,5)
```

グローバルの詳細は、このドキュメントの“多次元配列”の章と”グローバルの使用法”のドキュメントを参照してください。

3.5 未定義の値

ObjectScript 変数を明示的に宣言または定義する必要はありません。変数に値を代入するとすぐ、この変数が定義されます。この最初の代入が行われるまで、この変数へのすべての参照は未定義となります。$DATA 関数を使用して、変数が定義済みであるか未定義であるかを判断できます。

$DATA は、1 つあるいは 2 つの引数を取ります。1 つの引数を取る場合、変数が値を持つかどうかをテストします。

ObjectScript

```
WRITE "Does ""MyVar"" exist?",!
IF $DATA(MyVar) {
        WRITE "It sure does!"
} ELSE {
        WRITE "It sure doesn't!"
}

SET MyVar = 10
WRITE !,!,"How about now?",!
IF $DATA(MyVar) {
       WRITE "It sure does!"
} ELSE {
       WRITE "It sure doesn't!"
}
```

$DATA は、変数が値を持つ場合は True (1) を、値を持たない場合 (データがない場合) は False (0) のブーリアン値を返します。2 つの引数を取る場合、テストを実行し、2 番目の引数の変数を、テストされた変数の値と同じ値に設定します。

ObjectScript

```
IF $DATA(Var1,Var2) {
   WRITE "Var1 has a value of ",Var2,".",!
} ELSE {
    WRITE "Var1 is undefined.",!
}

SET Var1 = 3
IF $DATA(Var1,Var2) {
   WRITE "Var1 has a value of ",Var2,".",!
} ELSE {
    WRITE "Var1 is undefined.",!
}
```

3.6 ブーリアン値

論理コマンドあるいは演算子と共に使用するなどの特定の場合、値はブーリアン値 (真偽値) に解釈されます。このような場合、式がゼロ以外の数値に評価される場合は 1 (True)、ゼロの数値に評価される場合は 0 (False) と解釈されます。数値文字列はそれ自身の数値として評価されますが、非数値文字列は 0 (False) に評価されます。

例えば、以下の値は True になります。

ObjectScript

```
IF 1 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF 8.5 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF "1 banana" { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF 1+1 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF -7 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF +"007"=7 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
```

以下の値は False になります。

ObjectScript

```
IF 0 { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF 3-3 { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF "one banana" { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF "" { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF -0 { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF "007"=7 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
```

数値としての文字列の評価の詳細は、このドキュメントの“演算子と式”の章の”文字列から数値への変換”を参照してください。

3.7 日付

ObjectScript には、組み込みの日付タイプはありません。その代わりに、文字列として表される日付値を操作または形式設定するための関数が多数あります。日付形式は以下のようになります。

テーブル 3-1: 日付形式

形式	説明
$HOROLOG	これは、$HOROLOG ($H) 特殊変数で返される形式です。これはコンマで区切られた 2 つの整数からなる文字列です。最初の整数は、1840 年 12 月 31 日以降の日数、2 番目は今日の日付の午前 0 時以降の秒数です。$HOROLOG では秒の小数部はサポートされません。$NOW 関数は、秒の小数部を使用した $HOROLOG 形式の日付を提供します。InterSystems IRIS は、日付を $HOROLOG 形式に設定したり、この形式の日付を検証したりするための多数の関数を用意しています。
ODBC Date	これは、ODBC と他の多くの外部表現で使用される形式です。文字列の形式は、“YYYY-MM-DD HH:MM:SS”となります。ODBC の日付値は順番に並べられます。つまり、ODBC 日付形式でデータをソートした場合、このデータは自動的に年代順に並べられます。
ロケールの日付	これは現在のロケールで使用される形式です。ロケールによって、日付の形式は以下のように異なります。 “アメリカ”の日付は mm/dd/yyyy (dateformat 1) の形式に設定されます。“ヨーロッパ”の日付は dd/mm/yyyy (dateformat 4) の形式に設定されます。csyw、deuw、engw、espw、eurw、fraw、itaw、mitw、ptbw、rusw、skyw、svnw、turw、ukrw を除くすべてのロケールでは、dateformat 1 を使用します。これらのロケールでは dateformat 4 を使用します。 アメリカの日付では、秒の小数部を表すためにピリオド (.) を小数点区切り文字として使用します。ヨーロッパの日付では、秒の小数部を表すためにコンマ (,) を小数点区切り文字として使用します。ただし、engw、eurw、skyw ではピリオドを使用します。 すべてのロケールでは、スラッシュ (/) を日付区切り文字として使用しますが、例外としてチェコ語 (csyw)、ロシア語 (rusw)、スロバキア語 (skyw)、スロベニア語 (svnw)、およびウクライナ語 (ukrw) ではピリオド (.) を使用します。
System Time	これは、$ZHOROLOG ($ZH) 特殊変数で返される形式です。システムが実行されている秒数 (とその一部) を含む浮動小数点数です。InterSystems IRIS を停止して再起動すると、この数値がリセットされます。一般的にこの形式は、処理のタイミングを計り検証するために使用します。

以下は、異なる日付形式の使用方法についての例です。

ObjectScript

```
SET now = $HOROLOG
WRITE "Current time and date ($H): ",now,!

SET odbc = $ZDATETIME(now,3)
WRITE "Current time and date (ODBC): ",odbc,!

SET ldate = $ZDATETIME(now,-1)
WRITE "Current time and date in current locale format: ",ldate,!

SET time = $ZHOROLOG
WRITE "Current system time ($ZH): ",time,!
```

4

変数

変数は、値が格納される場所の名前です。ObjectScript では、変数にデータ型がなく、宣言の必要はありません。

一般的には、SET コマンドを使用して、変数に値を代入することによって変数を定義します。null 文字列 ("") の値も変数に割り当てられます。コマンドおよび関数の多くは、変数を定義してから、その変数を使用する必要があります。変数が未定義である場合、既定においては、変数を参照すると <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、未定義の変数を参照する際に <UNDEFINED> エラーを生成しないように InterSystems IRIS® Data Platform の動作を変更できます。

READ コマンド、$INCREMENT 関数、$BIT 関数、引数を 2 つ使用した形式の $GET 関数など、演算によっては、未定義の変数を使用できます。これらの関数は値を変数に代入します。$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。

4.1 変数のカテゴリ

ObjectScript には、いくつかの変数の種類があります。

- ローカル変数
- プロセス・プライベート・グローバル変数 (PPG)
- グローバル変数 (グローバルとも呼ばれる)
- i%property インスタンス変数
- 特殊変数 (システム変数とも呼ばれる)

それぞれのカテゴリは目的別に使用されます。また、有効範囲の規則が異なる場合もあります。

多くのコンピュータ言語とは異なり、ObjectScript では変数を宣言する必要はありません。変数は、値が割り当てられたときに作成されます。ObjectScript は "タイプのない" 言語です。変数はあらゆるタイプのデータを受け取ることができます。これらのトピックの詳細は、"変数の宣言" および "変数のタイプと変換" を参照してください。

4.1.1 添え字付き変数

ローカル変数、プロセス・プライベート変数、およびグローバル変数はすべて、添え字を使用できます。以下のように、すべてのタイプの変数について、添え字の規約は類似したものとなります。

- 添え字は数値または文字列にすることができます。添え字には Unicode 文字をはじめ、あらゆる文字を含めることができます。有効な数値の添え字は、正と負の数、ゼロ、および小数を含みます。空の文字列 ("") は有効な添え字ではありません。

- 添え字の値は、大文字と小文字で区別されます。
- 数値の添え字はキャノニック形式に変換されます。したがって、^a(7)、^a(007)、^a(7.000)、および ^a(7.) はすべて同じ添え字となります。文字列の添え字はキャノニック形式に変換されません。したがって、^a(“7”)、^a(“007”)、^a(“7.000”)、および ^a(“7.”) は異なる添え字となります。文字列の添え字 ^a(“7”) は数値の添え字 ^a(7) と同じです。
- 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。

実際の最大数はいくつかの要因によって変わります。グローバルの最大数に関する詳細は、″グローバルの使用法″ の ″グローバル構造″ の章を参照してください。

ロック名の添え字は変数の添え字と同一の規約に従います。

4.1.1.1 配列変数

配列変数は、1 つ以上の添え字レベルを持つ変数です。添え字は括弧で囲まれています。添え字のレベルはコンマで区切られます。どの変数も (特殊変数を除く)、次の例に示すように 配列として使用できます。

ObjectScript

```
SET a(1) = "A local variable array"
SET a(1,1,1) = "Another local variable array"
SET ^||a(1) = "A process-private global array"
SET ^a(1) = "A global array"
SET obj.a(1) = "A multidimensional array property"
```

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名の長さに応じて異なります。添え字の規約と制限に関する詳細は、″グローバルの使用法″ の ″グローバル構造″ の章を参照してください。

4.1.2 オブジェクト・プロパティ

オブジェクト・プロパティは、オブジェクトの特定のインスタンスに対応し、格納される値です。厳密に言うと、オブジェクト・プロパティは変数ではありませんが、構文的には他のローカル変数と同様にオブジェクト・プロパティを使用できます。

プロパティ参照には複数のタイプがあります (`oref.property`、`..property`、および `i%property`)。オブジェクト参照 (`oref`) は式ではなく、ローカル変数とする必要があります。`property` 参照は、単一値プロパティまたは添え字付き多次元プロパティを参照できます。`oref.prop1.prop2` のように、チェーン形式のプロパティ参照もできます。

以下の例は、変数として使用されるプロパティ参照を示しています。

ObjectScript

```
// Create an Address object
SET address = ##class(Sample.Address).%New()
// Use the properties of the object
SET address.City = "Boston"
WRITE "City: ",address.City,!
```

変数としてのプロパティ参照の使用には一定の制限があります。いずれのタイプのプロパティも、SET $BIT() や SET $LISTBUILD() を使用して変更することはできません。非多次元プロパティを、SET $EXTRACT()、SET $LIST()、SET $PIECE() を使用して変更することはできません。$DATA()、$GET()、および $INCREMENT() は、プロパティが多次元である場合にのみ、プロパティを取得できます。プロパティを MERGE コマンドで使用することはできません。これらの処理は、インスタンス変数構文 `i%PropertyName` を使用してオブジェクト・メソッド内で実行できます。

4.2 ローカル変数

ローカル変数は、現在の InterSystems IRIS プロセスに格納される変数です。作成したプロセスだけにアクセス可能となります。マッピングによりすべてのネームスペースからアクセスできます。プロセスの終了時、そのプロセスのローカル変数はすべて削除されます。

InterSystems IRIS では、ローカル変数の SET と KILL は、ジャーナル化されたトランザクション・イベントとして扱われません。このようなイベントでは、トランザクションをロールバックしても、これらの処理には何の影響もありません。

4.2.1 名前付け規約

次の名前付け規約を使って、ローカル変数を定義します。

- ローカル変数名は、有効な識別子である必要があります。最初の文字は必ず、文字、もしくはパーセント記号 (%) のいずれかです。“%” 記号で始まる変数名は、“パーセント変数” と呼ばれ、その有効範囲は異なります。“%Z” または “%z” で始まる変数のみがアプリケーション・コードで使用できます。その他すべてのパーセント変数は、規則に従ってシステムで使用するために予約されています。この規則の詳細は、″サーバ側プログラミングの入門ガイド″ の “識別子のルールとガイドライン” を参照してください。パーセント記号 (%) は、ローカル変数名の最初の文字以外には使用できません。ローカル変数名のその他の文字は、文字または数字になります。
- 変数名には、あらゆる単語を使用できます。ただし、変数名には ObjectScript コマンドや SQL の予約語を使用しないことを強くお勧めします。
- ローカル変数名は、大文字と小文字を区別します。例えば、MYVAR、MyVar および myvar は、3 つの異なるローカル変数となります。
- ASCII 255 よりもコード値が大きい文字 (Unicode 文字) をローカル変数名に使用できます。
- ローカル変数名は、現在のプロセスに一意であることが必要です。その他のプロセスは、同じ名前のローカル変数を持つことができます。プロセス・プライベート・グローバルまたはグローバルは、ローカル変数と同じ名前を持つことができます。例えば、myvar、^||myvar および ^myvar は、3 つの異なる変数となります。
- ローカル変数名の長さは、最大 31 文字です。31 文字より長い名前を指定することもできますが、最初の 31 文字のみが使用されます。このため、ローカル変数名は最初の 31 文字内で一意である必要があります。
- ローカル変数には、添え字を使用できます。添え字を使用することで、ローカル変数を値の配列として、ローカル変数を定義できます。添え字は数字または文字列とすることができ、Unicode 文字も使用できます。

注釈　%IS ユーティリティは、いくつかのローカル変数をすべて大文字の名前で設定します。%IS が呼び出される状況では、これらの変数名の使用を回避する必要があります。詳細は、″入出力デバイス・ガイド″ の ″入出力デバイスとコマンド″ を参照してください。

4.2.1.1 無効な名前

上記の名前付け規約に準拠しないローカル変数名の場合、<SYNTAX> エラーが生成されます。ただし例外が 1 つあり、変数名の先頭がアンダースコア文字でその後に文字が続く場合は、<_CALLBACK SYNTAX> エラーが生成されます。例えば、`SET x=_a` です。

キャレット (^) が前に付く有効な変数名は、ローカル変数ではなく、グローバル変数です。

アット記号 (@) が前に付く有効な変数名は、ローカル変数名が後に続く間接演算子です。

4.2.2 ローカル変数の範囲

ObjectScript コードでは、ローカル変数はすべてパブリックです。このため、現在のコンテキスト内で ObjectScript コードのプロセスによって実行されるいずれの処理からもローカル変数にアクセスできます。ローカル変数値へのアクセスには、以下のような制約があります。

- NEW コマンドは、新しいローカル変数コンテキストを作成します。引数なしの NEW は、既存のローカル変数に何も定義されていない新しいコンテキストを作成します。NEW var は、ローカル変数 var が定義されていない新しいコンテキストを作成します。QUIT コマンドは、前のローカル変数コンテキストに戻します。
- プロシージャ・ブロック内では、ローカル変数は既定でプライベートです。プライベート・ローカル変数は、そのプロシージャ・ブロック内でのみ定義されます。

  プロシージャ・ブロック内のローカル変数は以下のように動作します。

  - プライベート変数。プロシージャ・ブロック内で使用されるローカル変数は、プライベート変数です。パブリック変数として宣言されるか % 変数でない限り、そのプロシージャ・ブロック内でのみ定義されます。既定では、スタジオで生成されるすべてのオブジェクト・メソッドがプロシージャ・ブロックを使用するため (プロシージャ・ブロック・クラス・キーワードは、クラス定義内で設定されます)、メソッドで生成されるすべての変数がプライベート変数です。プロシージャ・ブロック内のプライベート変数に NEW コマンドを使用することはできません。
  - パブリック変数。プロシージャ・ブロックでは、ローカル変数のリストをパブリック変数として明示的に宣言できます。パブリック変数として宣言された変数の値にはプロシージャ・ブロックの外部からアクセスできます。パブリック変数のこのコンマ区切りリストには、存在しない変数や % 変数を含めることができます。プロシージャ・ブロック内のパブリック変数には NEW コマンドを使用できます。

    2 つのローカル変数 var1 と var2 のパブリック変数リストは、`MyProc(x,y) [var1,var2] PUBLIC { code body }` のように指定します (PUBLIC キーワードはプロシージャがパブリックであることを指定するもので、パブリック変数リストとは関係ないことに注意してください)。パブリック変数は、コンマ区切りのリストで指定します。添え字なしのローカル変数のみを指定できます。パブリック変数リストに添え字なしの変数を指定すると、その添え字レベルもすべてパブリックになります。単純なオブジェクト参照 (OREF) のみを指定できます。パブリック変数リストに OREF を指定すると、そのオブジェクト・プロパティもすべてパブリックになります。パブリック変数リストには、未定義の変数を含めることができます。
  - % 変数。名前が“%”で始まるローカル変数は、自動的にパブリック変数として宣言されます。これにより、プロセス内のすべてのコードから参照できる変数を、パブリックとして明示的にリストすることなく定義できます。“%Z”または“%z”で始まる変数のみをアプリケーション・コードで使用できます。その他すべての % 変数は、規則に従ってシステムで使用するために予約されています。この規則の詳細は、″サーバ側プログラミングの入門ガイド″ の“識別子のルールとガイドライン”を参照してください。プロシージャ・ブロック内の % 変数には NEW コマンドを使用できます。

  これらのメカニズムについては、“呼び出し可能なユーザ定義コードモジュール”の章の ″プロシージャ変数″ のセクションでより詳細に説明しています。
- XECUTE コマンドは、既定でローカル変数をパブリックとして定義します。XECUTE コマンド内で、明示的にローカル変数をプライベートとして定義できます。ローカル変数をプライベートまたはパブリックのいずれかとして明示的に定義する方法の詳細は、″XECUTE″ を参照してください。

引数の指定なしで WRITE または ZWRITE コマンドを使用すると、現在定義されているローカル変数をすべてリストできます。$QSUBSCRIPT 関数を使って、指定したローカル変数のコンポーネント (名前と添え字) を返したり、$QLENGTH 関数を使って、添え字の層の数を返すことができます。また、KILL コマンドを使用して、ローカル変数を削除できます。

4.2.3 オブジェクト値

オブジェクト値は、メモリ内のオブジェクトのインスタンスを参照します。オブジェクト値をローカル変数に割り当てることができます。

ObjectScript

```
SET person = ##class(Sample.Person).%New()
WRITE person,!
```

注釈　person の値は、文字列に変換されたオブジェクト参照 (OREF) の値です。文字列あるいはその値は、データベースからオブジェクトにロードするためには使用できません。

ドット構文を使用して、メソッドとオブジェクトのプロパティを参照できます。

ObjectScript

```
SET person.Name = "El Vez"
```

$ISOBJECT 関数を使用して、変数がオブジェクトを含むかどうかを判断します。

ObjectScript

```
SET str = "A string"
SET person = ##class(Sample.Person).%New()

WRITE "Is string an object? ", $IsObject(str),!
WRITE "Is person an object? ", $IsObject(person),!
```

グローバルにはオブジェクト値を代入できません。割り当てると、実行時エラーが発生します。

オブジェクト値を変数 (あるいはオブジェクト・プロパティ) に割り当てると、オブジェクトの内部参照カウントがインクリメントされるという悪影響があります。オブジェクトの参照番号が 0 に到達した場合、InterSystems IRIS は自動的にそのオブジェクトを破壊します (%OnClose() コールバック・メソッドを呼び出し、そのオブジェクトをメモリから削除します)。例えば以下のようになります。

ObjectScript

```
SET person = ##class(Sample.Person).%New() // one reference to Person
SET alias = person // two references

SET person = "" // 1 reference

SET alias = "" // no references left, object destroyed
```

4.3 プロセス・プライベート・グローバル

プロセス・プライベート・グローバルは、これを作成したプロセスによってのみアクセス可能な変数です。プロセスの終了時、プロセス・プライベート・グローバルはすべて削除されます。

- プロセス固有 : プロセス・プライベート・グローバルは、作成元のプロセスからのみアクセス可能で、そのプロセスが完了すると存在しなくなります。これは、ローカル変数と同様です。
- 常にパブリック : プロセス・プライベート・グローバルは常にパブリック変数です。これは、グローバル変数と同様です。
- ネームスペース非依存 : プロセス・プライベート・グローバルは、ネームスペースから独立して存在します。マッピングによりすべてのネームスペースからアクセスできます。したがって、現在のネームスペースに関係なく、作成、アク

セス、および削除できます。これは、どちらもネームスペースに固有であるローカル変数ともグローバル変数とも異なります。

- 引数なしの KILL、NEW、WRITE、または ZWRITE の影響を受けない：プロセス・プライベート・グローバルを KILL、WRITE、または ZWRITE の引数として指定できます。これは、グローバル変数と同様です。

プロセス・プライベート・グローバルは、サイズの大きなデータ値に使用することができます。多くの場合、Mgr/Temp ディレクトリで使用するための置換値の役割を果たし、プロセスの終了時に自動クリーンアップを行います。

InterSystems IRIS では、プロセスのプライベート・グローバルの SET と KILL は、ジャーナル化されたトランザクション・イベントとして扱われません。このようなイベントでは、トランザクションをロールバックしても、これらの処理には何の影響もありません。

4.3.1 名前付け規約

プロセス・プライベート・グローバル名は、次のいずれかの形式となります。

```
^||name
^|"^"|name
^["^"]name
^["^",""]name
```

これらの 4 つの接頭語は同等で、4 つはいずれも同じプロセス・プライベート・グローバルを参照します。最初の形式 (^||name) がもっとも一般的で、新しいコードに推奨されるものです。2 番目 ～ 4 番目の形式は、グローバルを定義する既存のコードとの互換性のために提供されています。これにより、name をプロセス・プライベート・グローバルまたは標準グローバルとして、定義するかどうかを決定する変数を指定できます。詳細は、以下の例を参照してください。

ObjectScript

```
SET x=1       // toggle storage type
IF x=1 {
  SET a="^"   // for a process-private global
}
ELSE {
  SET a=""    // for a standard global
}
SET ^|a|name="a value"
```

プロセス・プライベート・グローバルは、次の名前付け規約を使用します。

- プロセス・プライベート・グローバル名は、有効な識別子である必要があります。最初の文字 (2 番目の垂直バーの後) は必ず、文字、もしくはパーセント記号 (%) のいずれかにする必要があります。パーセント記号 (%) は、プロセス・プライベート・グローバル名の最初の文字以外には使用できません。“%Z” または “%z” で始まるパーセント変数のみがアプリケーション・コードで使用できます (^||%zmyppg や ^||%Z123 など)。その他すべてのパーセント変数は、規則に従ってシステムで使用するために予約されています。この規則の詳細は、″サーバ側プログラミングの入門ガイド″ の “識別子のルールとガイドライン” を参照してください。

  プロセス・プライベート・グローバル名の 2 番目およびその後に続く文字は、文字、数字、ピリオド記号 (.) になります。ピリオドを、名前の最初または最後の文字にすることはできません。

- プロセス・プライベート・グローバル名には、Unicode 文字 (ASCII 255 よりもコード値が大きい文字) を使用できません。プロセス・プライベート・グローバル名に Unicode 文字を使用すると <WIDE CHAR> エラーが発生します。
- プロセス・プライベート・グローバル名は、大文字と小文字を区別します。
- プロセス・プライベート・グローバル名は、プロセス内で一意である必要があります。
- プロセス・プライベート・グローバル名は、接頭語を除き最大 31 文字です。31 文字より長い名前を指定することもできますが、最初の 31 文字のみが使用されます。このため、プロセス・プライベート・グローバル名は最初の 31 文字内で一意である必要があります。

- プロセス・プライベート・グローバルには、添え字を使用できます。添え字を使用することで、プロセス・プライベート・グローバルを値の配列として定義できます。添え字は数字または文字列とすることができ、Unicode 文字も使用できます。添え字の規約や添え字レベルの数に対する制約など、添え字の使用に関する情報については、"グローバルの使用法" の "グローバル構造" を参照してください。

4.3.2 プロセス・プライベート・グローバルのリスト

^$GLOBAL() の ^$||GLOBAL() 構文形式を使用して、現在のプロセスに属するプロセス・プライベート・グローバルに関する情報を返すことができます。

^GETPPGINFO ルーチンを使用して、現在のすべてのプロセス・プライベート・グローバルの名前およびそれらのスペース割り当てをブロック単位で表示できます。^GETPPGINFO はプロセス・プライベート・グローバルの添え字も値もリストしません。プロセス ID (pid) を指定することで、特定プロセスのプロセス・プライベート・グローバルを表示できます。また、ワイルドカード文字列 (*) を指定することで、すべてのプロセスのプロセス・プライベート・グローバルを表示できます。^GETPPGINFO を呼び出すには %SYS ネームスペースで作業する必要があります。

以下の例では、^GETPPGINFO を使用して、現在のすべてのプロセスのプロセス・プライベート・グローバルをリストします。

ObjectScript

```
SET ^||flintstones(1)="Fred"
SET ^||flintstones(2)="Wilma"
NEW $NAMESPACE
SET $NAMESPACE="%SYS"
DO ^GETPPGINFO("*")
```

^GETPPGINFO ルーチンは、以下のように引数を取ります。

```
do ^GETPPGINFO("pdf","options","outfile")
```

これらの引数は以下のとおりです。

- pdf はプロセス ID または * ワイルドカードにすることができます。
- options は次の文字列のあらゆる組み合わせにすることができます。
  - b (バイト数で値を返す)
  - Mnn (nn 個以上のブロックを使用するプロセス・プライベート・グローバルがあるプロセスのみをリストする)

    プロセス・プライベート・グローバルがないプロセスをリストに含めるには M0 を使用します。

    プロセス・プライベート・グローバルがないプロセスはリストから除外し、グローバル・ディレクトリ・ブロックのみがあるプロセスを含めるには M1 を使用します (これが既定です)。

    プロセス・プライベート・グローバルがないプロセスをリストから除外し、グローバル・ディレクトリ・ブロックのみがあるプロセスも除外するには M2 を使用します。
  - S (outfile と共に使用して画面表示を抑制)
  - T (プロセスの合計数のみを表示)
- outfile は、^GETPPGINFO 出力を受け取る CSV (コンマ区切り値) 形式ファイルのファイル・パスです。

次の例では、プロセス・プライベート・グローバルを ppgout という名前の出力ファイルに書き込みます。S オプションで画面表示を抑制し、出力を 500 ブロック以上使用するプロセス・プライベート・グローバルがあるプロセスのみを M500 オプションで表示します。

ObjectScript

```
NEW $NAMESPACE
SET $NAMESPACE="%SYS"
DO ^GETPPGINFO("*","SM500","/home/myspace/ppgout")
```

4.4 グローバル

グローバルは、InterSystems IRIS データベースに自動的に格納される特殊な変数です。これは固有のネームスペースにマッピングされるので、拡張参照を使用しない限り、そのネームスペース内でのみアクセスできます。グローバルは任意のプロセスからアクセスできます。グローバルは、そのグローバルを作成したプロセスの終了後も存続します。グローバルは、明示的に削除されるまで存続します。

InterSystems IRIS では、グローバルの SET と KILL は、ジャーナル化されたトランザクション・イベントとして扱われます。このようなイベントでは、トランザクションをロールバックすると、これらの処理が取り消されて元の状態に戻ります。グローバルに対する変更に他のプロセスからアクセスできないように、変更したトランザクションがコミットされるまでロックを使用できます。詳細は、“トランザクション処理”の章を参照してください。

ObjectScript プログラムでは、他の変数と同様にグローバルを使用できます。グローバル名は、構文的に“^”文字を文字またはパーセント“%”記号の前に置いて区別します。

ObjectScript

```
SET mylocal = "This is a local variable"
SET ^myglobal = "This is a global stored in the current namespace"
```

グローバルの命名規約は、以下のとおりです。

- グローバルは、グローバル接頭語とグローバル名で構成されています。グローバル接頭語は、通常、キャレット (^) 文字で、現在のネームスペースでのグローバルであることを指定します。グローバル接頭語は、`^|"samples"|` などの拡張参照とすることもでき、この場合は別のネームスペースでのグローバルであることを指定します。
- グローバル名は、有効な識別子である必要があります。(接頭語の文字の後の) 最初の文字は必ず、文字、もしくはパーセント記号 (%) のいずれかです。“%Z”または“%z”で始まるパーセント変数のみがアプリケーション・コードで使用できます (`^%zmyglobal` や `^%Z123` など)。これらの ^%Z グローバルと ^%z グローバルは IRISSYS データベースに書き込まれるので、InterSystems IRIS のアップグレードを経ても保持されます。その他すべてのパーセント変数は、規則に従ってシステムで使用するために予約されています。この規則の詳細は、″サーバ側プログラミングの入門ガイド″ の“識別子のルールとガイドライン”を参照してください。

  グローバル名の 2 番目およびその後に続く文字は、文字、数字、ピリオド記号 (.) になります。ピリオドを、名前の最初または最後の文字にすることはできません。
- グローバル名には、Unicode 文字 (ASCII 255 よりもコード値が大きい文字) を使用できません。グローバル名に Unicode 文字を使用すると <WIDE CHAR> エラーが発生します。
- グローバル名は、大文字と小文字を区別します。
- グローバル名は、そのネームスペース内で重複しないようにします。
- グローバル名は、接頭語を除き最大 31 文字です。31 文字より長い名前を指定することもできますが、最初の 31 文字のみが使用されます。このため、グローバル名は最初の 31 文字内で一意である必要があります。
- グローバルに添え字を使用できます。添え字を使用することで、グローバルを値の配列として定義できます。 添え字は数字または文字列とすることができ、Unicode 文字も使用できます。添え字の規約や添え字レベルの数に対する制約など、添え字の使用に関する情報については、″グローバルの使用法″ の ″グローバル構造″ を参照してください。

- インターシステムズで使用する目的で保持されているグローバル名もあります。詳細は、"サーバ側プログラミングの入門ガイド" の "回避する必要があるグローバル変数名" を参照してください。

または、グローバルは、カレット文字のすぐ後に垂直バーまたは角括弧のペアを使用して、ネームスペースやディレクトリを定義する拡張参照を指定できます (`^|"samples"|myglobal` または `^|""|myglobal` など)。これらの拡張グローバル参照をプロセス・プライベート・グローバルと混同しないでください。

$ZREFERENCE 特殊変数を使用して、最後に使用されたグローバルの名前を特定できます。$QSUBSCRIPT 関数を使って、指定したグローバルのコンポーネントを返したり、$QLENGTH 関数を使って、添え字の層の数を返すことができます。

グローバルの詳細は、"グローバルの使用法" を参照してください。

4.5 特殊変数

ObjectScript には、特定のシステム情報をアプリケーションで有効にするために使用する組み込み特殊変数 (システム変数ともいいます) が多数あります。特殊変数はすべて InterSystems IRIS で供給される変数で、"$" 記号の接頭辞を使用した名前を使用します。ユーザがその他の特殊変数を定義することはできません。特殊変数一式をマッピングすると、すべてのネームスペースからアクセスできます。

その値は、ユーザの現在の動作環境に合わせて設定されます。特殊変数の一部は、はじめ NULL 文字列 ("") に設定されているため、特殊変数を参照しても <UNDEFINED> エラーは生成されません。特殊変数の値は現在のプロセスに固有のもので、別のプロセスからアクセスできません。

ユーザは、SET コマンドを使用して一部の特殊変数を設定できますが、それ以外の特殊変数をユーザが変更することはできません。詳細は、個々の特殊変数を参照してください。

次の例では、特殊変数 $HOROLOG を使用します。

ObjectScript

```
SET starttime = $HOROLOG
HANG 5
WRITE !,$ZDATETIME(starttime)
WRITE !,$ZDATETIME($HOROLOG)
```

特殊変数 $HOROLOG は、現在のシステムの日付と日時を格納します。SET コマンドはこの特殊変数を使用して、ユーザ定義のローカル変数 starttime にこの値を設定します。HANG コマンドは、プログラムを 5 秒間停止します。最後に、2 つの $ZDATETIME 関数は、starttime と現在のシステムの日付と時間をユーザが読み取れる形式で返します。

以下はその他の特殊変数の例です。

ObjectScript

```
WRITE !,"$JOB = ",$JOB   // Current process ID
WRITE !,"$ZVERSION = ",$ZVERSION   // Version info
```

多くの特殊変数は読み取り専用で、SET コマンドを使用して設定することができません。$DEVICE などのその他の特殊変数は、読み取り/書き込み可能で、SET コマンドで設定できます。

特殊変数には、添え字を使用できません。特殊変数は、$INCREMENT 関数を使用してインクリメントすることや KILL コマンドを使用して削除することができません。特殊変数は、"ObjectScript の使用法" の "コマンド" の章の "表示 (書き込み) コマンド" で説明するように、WRITE、ZWRITE、ZZWRITE、または ZZDUMP コマンドを使用して表示できます。

特殊変数のリストと詳細説明は、"ObjectScript リファレンス" を参照してください。

4.6 変数のタイプと変換

ObjectScript の変数には決まった形式はありません。つまり、特定のデータ型はありません (これは、JavaScript、VBScript、および Document Data Base/JSON にも当てはまります)。したがって、文字列値を変数に割り当て、その後同じ変数に数値を割り当てることができます。最適化として、InterSystems IRIS は文字列、整数、数字、オブジェクトに異なる内部表現を使用しますが、アプリケーション・プログラマからは見えません。InterSystems IRIS は、使用されるコンテキストを基にして、変数の値を自動的に変換 (または解釈) します。

以下はその例です。

ObjectScript

```
// set some variables
SET a = "This is a string"
SET b = "3 little pigs"
SET int = 22
SET num = 2.2
SET obj = ##class(Sample.Person).%New()

// Display them
WRITE "Here are the variables themselves: ",!
WRITE "a: ",a,!
WRITE "b: ",b,!
WRITE "int: ",int,!
WRITE "num: ",num,!
WRITE "obj: ",obj,!,!

// Now use them as other "types"
WRITE "Here are the numeric interpretation of",!
WRITE "a, b, and obj: ",!
WRITE "+a: ",+a,!
WRITE "+b: ",+b,!
WRITE "+obj: ",+obj,!,!

WRITE "Here are concatenations of int and num:",!
WRITE "Concatenating int: ","I found " _ int _ " apples.",!
WRITE "Concatenating num: ","There are " _ num _ " pounds per kilogram.",!
```

InterSystems IRIS は、以下のように値を変換します。

テーブル 4-1: ObjectScript の型の変換規則

変換元	変換先	規則
番号	文字列	数値を表す文字列が使用され、前の例では変数 num は 2.2 のようになります。
文字列	数値	文字列の先頭文字は数値リテラルとして解釈されます。これについては、“演算子と式”の章の“文字列から数値への変換”のセクションで説明しています。例えば、“-1.20abc” は -1.2 と解釈され、“abc123” は 0 と解釈されます。
オブジェクト	数値	指定されたオブジェクト参照の内部オブジェクト・インスタンス番号が使用されます。その値は整数です。
オブジェクト	文字列	n@cls 形式の文字列が使用されます。n は内部オブジェクトのインスタンス番号で、cls は指定されたオブジェクトのクラス名です。
数値	オブジェクト	許可されていません。
文字列	オブジェクト	許可されていません。

4.7 変数の宣言

他の言語と異なり、ObjectScript で変数を宣言する必要はありません。ただし、#dim プリプロセッサ指示文をコードの記述の補助として使用できます。

スタジオでコードを記述する場合は、#dim プリプロセッサ指示文を使用できます。#dim は、対象変数タイプについての情報を提供します。スタジオは、スタジオ・アシスト機能と共にこの情報を使用してコードを処理します。この情報は、ドキュメントに使用することも、コードを参照する可能性のある他のユーザに提供することもできます(スタジオ・アシストに関する詳細は、"スタジオの使用法" の "スタジオ・オプションの設定" の章の "エディタ・オプション" のセクションを参照してください)。

#dim の構文形式は、以下のとおりです。

ObjectScript

```
#dim VariableName As DataTypeName
#dim VariableName As List Of DataTypeName
#dim VariableName As Array Of DataTypeName
```

ここで、VariableName は、データ型を指定する変数で、DataTypeName は、そのデータ型を指定します。スタジオには、DataTypeName の値を選択できるメニューが用意されています。

#dim では、以下のように変数の初期値を指定することもできます。

ObjectScript

```
#dim President As %String = "Obama"
```

5
演算子と式

InterSystems IRIS® Data Platform は、多様な演算子をサポートします。数理演算動作、論理比較などさまざまな動作を実行します。演算子は、最終的に 1 つの値に評価される変数や他のエンティティである式に対して作用します。この章では、式とさまざまな ObjectScript 演算子について説明します。

5.1 演算子と式の概要

演算子とはシンボル文字で、対応するオペランドで実行する処理を指定するものです。各オペランドは、1 つ以上の式または式アトムから構成されます。演算子とその演算子に対応するオペランドを一緒に使用する場合、以下の形式になります。

[operand] 演算子 operand

演算子の中には、1 つのオペランドのみを使用する単項演算子があります。2 つのオペランドを使用するものは、二項演算子といいます。

式は、演算子とそこで使用されるオペランドで構成されます。このような式は、オペランドに対する演算子の演算結果を生成します。式は、含まれる演算子タイプに基づいて分類されます。

- 算術式は算術演算子を含み、オペランドを数値として解釈し、数値結果を算出します。
- 文字列式は文字列演算子を含み、オペランドを文字列として解釈し、文字列の結果を返します。
- 論理式は関係演算子と論理演算子を含み、オペランドを論理解釈し、True (1) あるいは False (0) のブーリアン値を返します。

5.1.1 演算子記号の表

ObjectScript には、以下の演算子があります。

テーブル 5-1: ObjectScript 演算子

演算子	実行される演算
+	正数 (単項)、加算 (2 項)
-	負数 (単項)、減算 (2 項)
*	乗算
/	除算

演算子	実行される演算
¥	整数除算
#	モジュロ (剰余)
**	べき乗
<	より小さい
>	より大きい
<= '>	以下
>= '<	以上
'	論理補数 (NOT)
& &&	論理 AND (&& は AND の“簡易版”)
! ||	論理 OR (|| は OR の“簡易版”)
'&	論理 NAND
'!	論理 NOR
_	連結
=	等しい (および代入)
'=	等しくない
[	包含
'[	非包含
]	追従
']	非後続
]]	前後関係演算
']]	非前後関係演算
? '?	パターン・マッチ
@	間接演算

詳細は、以下のセクションで説明しています。

5.1.2 演算子の優先順位

ObjectScript で演算子の評価順序は、必ず左から右です。したがって、式の演算は表示された順番で実行されます。これは、特定の演算子の優先順位が他の演算子よりも高くなることがある他の言語と異なります。式で明示的に小括弧を使用して、特定の演算子を先に処理させることができます。

ObjectScript

```
WRITE "1 + 2 * 3 = ", 1 + 2 * 3,!  // returns 9
WRITE "2 * 3 + 1 = ", 2 * 3 + 1,!  // returns 7
WRITE "1 + (2 * 3) = ", 1 + (2 * 3),!  // returns 7
WRITE "2 * (3 + 1) = ", 2 * (3 + 1),!  // returns 8
```

InterSystems SQL では、演算子の優先順位を構成でき、ObjectScript の演算子の優先順位と一致させる (または一致しないようにする) ことができます。

5.1.2.1 単項マイナス演算子

ObjectScript は、二項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。

ObjectScript

```
WRITE -123 - 3,!       // returns -126
WRITE -123 + -3,!      // returns -126
WRITE -(123 - 3),!     // returns -120
```

5.1.2.2 括弧と優先順位

式の評価の順序は、それぞれの式を対の小括弧で入れ子にして変更できます。小括弧は、囲んだ式 (算術式と関係式の両方) をグループ化し、ObjectScript が式で実行する演算の順序を制御します。以下の例を考えてみます。

ObjectScript

```
SET TorF = ((4 + 7) > (6 + 6)) // False (0)
WRITE TorF
```

上記では、小括弧で 4 と 7、および 6 と 6 を加算しているため、論理式は `11 > 12` となり、結果は False になります。以下のコードと比較します。

ObjectScript

```
SET Value = (4 + 7 > 6 + 6) // 7
WRITE Value
```

この場合、演算の優先順位は左から右になります。したがって、最初に 4 と 7 を加算します。その合計の 11 と 6 を比較し、11 は 6 より大きくなるため、論理演算の結果は 1 (True) になります。その後、1 に 6 を加算するため、結果は 7 になります。

優先順位により結果のタイプが異なることに注意してください。上記の例で、最初の式の演算は最終的にブーリアン値を返し、2 番目の式は数値を返します。

以下の例では、複数レベルの入れ子を示します。

ObjectScript

```
WRITE 1+2*3-4*5,!  // returns 25
WRITE 1+(2*3)-4*5,!  // returns 15
WRITE 1+(2*(3-4))*5,!  // returns -5
WRITE 1+(((2*3)-4)*5),!  // returns 11
```

内側の入れ子の式から 1 レベルずつ外側に進み、各レベルで左から右へと式が評価されます。

Tipヒント 極めて単純な ObjectScript 式を除いて、すべての式全体を括弧で囲むことをお勧めします。これにより、評価の順序のあいまいさが解消され、コードの本来の意図について今後疑問が出ることもなくなります。

例えば、すべての演算子と同様に“&&”演算子は、左から右の順に実行されるため、以下のコード例の最後の文は 0 に評価されます。

ObjectScript

```
SET x = 3
SET y = 2
IF x && y = 2 {
  WRITE "True",! }
ELSE {
  WRITE "False",! }
```

これは評価が次のように実行されるからです。

1. 最初に、x が定義されていて、0 以外の値であるかどうかが確認されます。x は 3 なので評価は続行されます。
2. 次に、y が定義されていて、0 以外の値であるかどうかが確認されます。y は 2 なので評価は続行されます。
3. 次に、3 && 2 の値が評価されます。3 も 2 も 0 ではないので、この式は True で、1 に評価されます。
4. 次に、返された値を 2 と比較します。1 は 2 ではないので、この評価は 0 を返します。

多くのプログラミング言語に精通している人にとって、これは予想外の結果です。x に0 でない値が定義されていて、y が 2 のときに True を返すことを意図している場合は、次のように括弧が必要です。

ObjectScript

```
SET x = 3
SET y = 2
IF x && (y = 2) {
  WRITE "True",! }
ELSE {
  WRITE "False",! }
```

5.1.2.3 関数と優先順位

関数など、式のタイプによって副次的作用が発生する場合があります。以下の論理式を考えてみます。

ObjectScript

```
IF var1 = ($$ONE + (var2 * 5)) {
   DO ^Test
}
```

ObjectScript は、最初に var1、次に関数 $$ONE、その次に var2 を評価します。その後、var2 を 5 倍し、最後に ObjectScript は、加算の結果が var1 の値に等しいかどうかをテストします。等しい場合、DO コマンドを実行して Test ルーチンを呼び出します。

別の例として、以下の論理式を考えてみます。

ObjectScript

```
SET var8=25,var7=23
IF var8 = 25 * (var7 < 24) {
  WRITE !,"True" }
ELSE {
  WRITE !,"False" }
```

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。この場合、ObjectScript はまず var8=25 を評価し、結果が 1 になります。次に、この 1 に括弧内の式の結果を

乗算します。var7 は、24 より小さいので、括弧内の式は、1 と評価されます。したがって、ObjectScript は、1 * 1 の乗算を行い、結果は 1 (True) となります。

5.1.3 式

ObjectScript 式は、値を算出するために評価される 1 つ以上の“トークン”です。最も単純な式は、リテラルあるいは変数です。

ObjectScript

```
SET expr = 22
SET expr = "hello"
SET expr = x
```

配列、演算子、多くの ObjectScript 関数の 1 つを使用して、さらに複雑な式を記述できます。

ObjectScript

```
SET expr = +x
SET expr = x + 22
SET expr = array(1)
SET expr = ^data("x",1)
SET expr = $Length(x)
```

式は、オブジェクト・プロパティ、インスタンス・メソッド呼び出し、クラス・メソッド呼び出しから構成されます。

ObjectScript

```
SET expr = person.Name
SET expr = obj.Add(1,2)
SET expr = ##class(MyApp.MyClass).Method()
```

ルーチン呼び出しの前に $$ を置き、ObjectScript ルーチン呼び出しを式で直接実行できます。

ObjectScript

```
SET expr = $$MyFunc^MyRoutine(1)
```

式は、返す値の種類で分類できます。

- 算術式は算術演算子を含み、オペランドを数値として解釈し、数値結果を算出します。

  ObjectScript

  ```
  SET expr = 1 + 2
  SET expr = +x
  SET expr = a + b
  ```

  算術式で使用される文字列は、数値として評価されます (有効な数値がない場合は 0 となります)。また、単項加算演算子 (+) を使用すると、文字列値を数値に暗黙に変換します。

- 文字列式は文字列演算子を含み、オペランドを文字列として解釈し、文字列の結果を返します。

  ObjectScript

  ```
  SET expr = "hello"
  SET expr = "hello" _ x
  ```

- 論理式は関係演算子と論理演算子を含み、オペランドを論理解釈し、True (1) あるいは False (0) のブーリアン値を返します。

ObjectScript

```
SET expr = 1 && 0
SET expr = a && b
SET expr = a > b
```

- オブジェクト式 は、結果としてオブジェクト参照を生成します。

ObjectScript

```
SET expr = object
SET expr = employee.Company
SET expr = ##class(Person).%New()
```

5.1.3.1 論理式

論理式は、論理演算子、数値関係演算子、文字列関係演算子 を使用します。式を評価し、1 (True) または 0 (False) のブーリアン値を返します。論理式は、通常、以下を使用します。

- IF コマンド
- $SELECT 関数
- 後置条件式

ブーリアン・テストでは、ゼロ以外の数値に評価される式はいずれもブーリアン 1 (True) の値を返します。ゼロの数値に評価される式はいずれもブーリアン 0 (False) の値を返します。InterSystems IRIS では、非数値文字列がゼロの数値を有しているものとして評価します。詳細は、"文字列から数値への変換" を参照してください。

論理演算子を使用することにより、複数のブーリアン論理式を組み合わせることができます。すべての InterSystems IRIS の式と同様に、これらは厳密に左から順に評価されます。論理演算子のタイプには、正規論理演算子 (& および !) と簡易論理演算子 (&& および ||) の 2 つがあります。

正規論理演算子を使用して論理式を組み合わせる場合、InterSystems IRIS では、すべての式の評価が完了する前にブーリアン値の結果が分かっていたとしても、指定された式をすべて評価します。これにより、すべての式の妥当性が保証されます。

簡易論理演算子を使用して論理式を組み合わせる場合、InterSystems IRIS ではブーリアン値結果の割り出しに必要な式のみを評価します。例えば、複数の AND テストをする場合、0 を返す最初の式によって全体のブーリアン値結果が決まります。この式の右側にある論理式は評価されません。これにより、不必要な時間的浪費となる式の評価を回避できます。

コンマ区切りリストを引数値として指定できるコマンドもあります。この場合、InterSystems IRIS ではリストの各引数を独立コマンド文のように扱います。したがって、`IF x=7,y=4,z=2` は `IF x=7 THEN IF y=4 THEN IF z=2` と解析され、簡易論理演算子の文の `IF (x=7)&&(y=4)&&(z=2)` と機能的に同一となります。

以下の例では、IF テストにて正規論理演算子 (&) を使用しています。したがって、最初の関数が 0 (False) を返すため、自動的に式全体の結果が False になりますが、すべての関数が実行されます。

ObjectScript

```
LogExp
 IF $$One() & $$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

以下の例では、IF テストにて簡易論理演算子 (&&) を使用しています。したがって、最初の関数の実行により 0 (False) を返すため、自動的に式全体の結果が False になります。2 番目の関数は実行されません。

ObjectScript

```
LogExp
 IF $$One() && $$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

以下の例では、IF テストにてコンマ区切り引数を指定しています。コンマは論理演算子ではありませんが、簡易 && 論理演算子の指定と同じ効果があります。最初の関数の実行により 0 (False) を返すため、自動的に式全体の結果が False になります。2 番目の関数は実行されません。

ObjectScript

```
LogExp
 IF $$One(),$$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

5.1.4 代入

ObjectScript の SET コマンドは、代入演算子 (=) を併用して変数に値を代入します。代入コマンドの右側に式が置かれます。

ObjectScript

```
 SET value = 0
 SET value = a + b
```

ObjectScript では、代入コマンドの左側にも特定の関数を使用できます。

ObjectScript

```
 SET pies = "apple,banana,cherry"
 WRITE "Before: ",pies,!

 // set the 3rd comma-delimited piece of pies to coconut
 SET $Piece(pies,",",3) = "coconut"
 WRITE "After: ",pies
```

5.2 文字列から数値への変換

文字列は数値、部分的数値、または非数値となり得ます。

- 数値文字列はすべて数字で構成されます。例えば、"123"、"+123"、".123"、"++0007"、"-0" となります。

- 部分的数値の文字列とは、数字で始まり、非数値の文字が続く文字列です。例えば、"3 blind mice"、"-12 degrees" となります。
- 非数値の文字列は数値ではない文字で始まります。例えば、" 123"、"the 3 blind mice"、"three blind mice" となります。

5.2.1 数値文字列

数値文字列または部分的文字列を算術式で使用する場合、文字列は数値として解釈されます。文字列で左から右に数値文字をスキャンしてこの数値を取得し、数値リテラルと解釈できる開始文字の最長シーケンスを見つけます。以下の文字が可能です。

- 0 から 9 までの数字。
- PlusSign プロパティ値および MinusSign プロパティ値。既定では、それらは “+” および “-” の文字となりますが、ロケールに依存します。%SYS.NLS.Format.GetFormatItem() メソッドを使用して、現在の設定を返します。
- DecimalSeparator プロパティ値。既定では、これは “.” の文字となりますが、ロケールに依存します。%SYS.NLS.Format.GetFormatItem() メソッドを使用して、現在の設定を返します。
- 文字の “e” および “E” は、シーケンス内で 4E3 などの科学的記数法を表す場合に、数値文字列の一部含むことができます。

NumericGroupSeparator プロパティ値 (既定では、“,” の文字) は、数値文字とはみなされないことに注意してください。したがって、文字列 "123,456" は部分的数値文字列となり、数 "123" に分解されます。

数値文字列および部分的数値文字列は、算術演算 (加算および減算) および「より大きい/より小さい」などの比較演算 (<、>、<=、>=) に優先して、キャノニック形式に変換されます。数値文字列は等値比較 (=、'=) に優先して、キャノニック形式へ変換されません。この理由は、それらの演算子が文字列比較にも使用されるためです。

以下の例は、数値文字列の算術比較を示しています。

ObjectScript

```
WRITE "3" + 4,!             // returns 7
WRITE "003.0" + 4,!         // returns 7
WRITE "++--3" + 4,!         // returns 7
WRITE "3 blind mice" + 4,!  // returns 7
```

以下の例は、数値文字列の < (より小さい) 比較を示しています。

ObjectScript

```
WRITE "3" < 4,!             // returns 1
WRITE "003.0" < 4,!         // returns 1
WRITE "++--3" < 4,!         // returns 1
WRITE "3 blind mice" < 4,!  // returns 1
```

以下の例は、数値文字列の <= 比較を示しています。

ObjectScript

```
WRITE "4" <= 4,!             // returns 1
WRITE "004.0" <= 4,!         // returns 1
WRITE "++--4" <= 4,!         // returns 1
WRITE "4 horsemen" <= 4,!    // returns 1
```

以下の例は、数値文字列の等値比較を示しています。非キャノニック形式の数値文字列は、数値ではなく、文字列として比較されます。-0 は非キャノニック形式の数値文字列です。したがって、数値ではなく、文字列として比較されます。

ObjectScript

```
WRITE "4" = 4.00,!          // returns 1
WRITE "004.0" = 4,!         // returns 0
WRITE "++--4" = 4,!         // returns 0
WRITE "4 horsemen" = 4,!    // returns 0
WRITE "-4" = -4,!           // returns 1
WRITE "0" = 0,!             // returns 1
WRITE "-0" = 0,!            // returns 0
WRITE "-0" = -0,!           // returns 0
```

5.2.2 非数値文字列

文字列の先頭の文字が数値文字ではない場合、すべての算術演算に対する文字列の数値は 0 となります。<、>、'>、<=、'<、および >= による比較演算でも、数値ではない文字列は 0 として扱われます。等号は数値的等値演算子および文字列比較演算子の両方として使用されるので、= および '= の演算では文字列比較が優先されます。PlusSign プロパティ値 (既定では +) を追加することで、文字列の数値的評価を強制することができます (例: "+123")。x および y が異なる非数値文字列 (x="Fred"、y="Wilma" など) である場合、結果は以下の論理値となります。

x, y	x, x	+x, y	+x, +y	+x, +x
x=*y* は False	*x*=*x* は True	+*x*=*y* は False	+*x*=+*y* は True	+*x*=+*x* は True
x'=*y* は True	*x*=*y* は False	+*x*'=*y* は True	+*x*'=+*y* は False	+*x*'=+*x* は False
x<*y* は FALSE	*x*<*x* は FALSE	+*x*<*y* は FALSE	+*x*<+*y* は FALSE	+*x*<+*x* は FALSE
x<=*y* は TRUE	*x*<=*x* は TRUE	+*x*<=*y* は TRUE	+*x*<=+*y* は TRUE	+*x*<=+*x* は TRUE

5.3 算術演算子

算術演算子は、オペランドを数値として解釈し、数値結果を生成します。文字列について演算する場合、算術演算子は、"文字列から数値への変換" のセクションで説明したルールに従って、文字列をその数値として扱います。

5.3.1 10 進数および $DOUBLE 浮動小数点数

InterSystems IRIS では、ObjectScript の 10 進数浮動小数点数と IEEE 倍精度バイナリ浮動小数点数の 2 つの小数点数の表現がサポートされています。

- InterSystems IRIS では、既定で、数値定数は ObjectScript の 10 進数浮動小数点値として表現されます。これを $DECIMAL 数値と言います。$DECIMAL 数値は、2.2 などの正確な小数値を表すことができます。$DECIMAL() 関数を使用して、IEEE 倍精度バイナリ浮動小数点数を対応する ObjectScript 10 進数浮動小数点数に明示的に変換します。
- InterSystems IRIS では、IEEE 倍精度バイナリ浮動小数値もサポートされています。$DOUBLE() 関数を使用して、数値定数を対応する IEEE 倍精度バイナリ浮動小数値 ($DOUBLE 数値) に明示的に変換します。$DOUBLE 数値は、2.2 などのおよその小数値を表すことしかできません。$DOUBLE 表現は、ほとんどのコンピュータがバイナリ浮動小数点演算のための高速ハードウェアを搭載しているため、通常、高速科学的計算を実行する場合に優先的に使用されます。

ObjectScript は、以下の状況では、数値を対応する $DOUBLE 値に自動的に変換します。

- 算術演算に $DOUBLE 値が含まれている場合、ObjectScript は演算内のすべての数値を $DOUBLE に変換します。例えば、`2.2 + $DOUBLE(.1)` は、`$DOUBLE(2.2) + $DOUBLE(.1)` と同じです。

- 演算の結果の数値が、ObjectScript 10 進数浮動小数点で表すには大きすぎる (9.223372036854775807E145 より大きい) 場合、ObjectScript は、<MAXNUMBER> エラーを発行するのではなく、この数値を $DOUBLE に自動的に変換します。

5.3.2 単項プラス演算子 (+)

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。次に例を示します。

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。次に例を示します。

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。次に例を示します。

ObjectScript

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

5.3.3 加算演算子 (+)

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。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使用し、オペランドの数値の和を算出します。

以下の例は、2 つの数値リテラルを加算します。

ObjectScript

```
WRITE 2936.22 + 301.45 //  3237.67
```

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
```

5.3.4 単項マイナス演算子 (-)

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。次に例を示します。

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、上記の単項プラス演算子と同様の方法で解釈されます。次に例を示します。

ObjectScript

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

ObjectScript は、二項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。

5.3.5 減算演算子 (-)

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。

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。次に例を示します。

ObjectScript

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

5.3.6 乗算演算子 (*)

二項乗算演算子は、数値として解釈される 2 つの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。

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、二項乗算演算子は、その値にゼロを割り当てます。

ObjectScript

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

5.3.7 除算演算子 (/)

二項除算演算子は、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。

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 // 1.636363636363636364
```

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、二項除算演算子は、その値をゼロと見なします。次に例を示します。

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。

5.3.8 整数除算演算子 (¥)

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。

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。

5.3.9 モジュロ演算子 (#)

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。

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" のセクション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 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。

5.3.10 指数演算子 (**)

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。
- `$DOUBLE("INF")**`*n* : 無限数 (正または負) のゼロ乗は 1、無限数 (正または負) の正数 (整数、小数、または INF) 乗は INF です。無限数 (正または負) の負数 (整数、小数、または INF) 乗は 0 です。
- `$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 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。

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。

5.4 数値関係演算子

文字列関係演算子と数値関係演算子という、2 種類の関係演算子があります。数値関係演算子は、オペランドの数値を使用してブーリアン値の結果を返します。文字列に対する演算では、“文字列から数値への変換” のセクションに説明があるルールに従い、文字列は数値関係演算子によってそれぞれが該当する数値として扱われます。

数値関係演算子を非数値文字列の比較で使用しないでください。

IEEE の倍精度小数 ($DOUBLE 数) と InterSystems IRIS 標準の浮動小数点数 ($DECIMAL 数) の間では、値を丸めずに、正確な値で比較が行われます。$DOUBLE 数と $DECIMAL 数の等値比較は、予期しない結果が発生することが多いので避ける必要があります。IEEE 倍精度数を含む算術演算の詳細は、“サーバ側プログラミングの入門ガイド” の付録 ″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。

5.4.1 二項に対するより小さい関係演算子 (<)

二項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テストします。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。次に例を示します。

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。

5.4.2 二項に対するより大きい関係演算子 (>)

二項より大きい関係演算子は、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。次に例を示します。

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。

5.4.3 以下演算子 (<= または '>)

以下のようにして、以下関係演算子を記述できます。

- 二項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。
- 二項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、二項より大きい関係演算子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。

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。

5.4.4 以上演算子 (>= または '<)

以下のようにして、以上関係演算子を記述できます。

- 二項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。
- 二項より小さい関係演算子 (<) 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、二項より小さい関係演算子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。

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)
```

5.5 論理比較演算子

論理比較演算子は、オペランド値を比較し、True (1) か False (0) のブーリアン値を返します。

5.5.1 論理演算子の優先順位

ObjectScript は、厳格に演算子を左から右へ評価するため、その他の演算子が関与する論理比較では、必要な優先順位を実現するために、演算をグループ化する括弧を使用する必要があります。例えば、以下のプログラムで True (1) を返すために、二項論理和演算 (!) テストが要求されます。

ObjectScript

```
SET x=1,y=0
IF x=1 ! y=0 {WRITE "TRUE"}
ELSE {WRITE "FALSE" }
// Returns 0 (FALSE), due to evaluation order
```

しかし、この論理比較を適切に実行するには、その他の演算を入れ子にする括弧を使用する必要があります。以下の例では、期待される結果が得られます。

ObjectScript

```
SET x=1,y=0
IF (x=1) ! (y=0) {WRITE "TRUE"}
ELSE {WRITE "FALSE" }
// Returns 1 (TRUE)
```

5.5.2 単項否定演算子 (')

単項否定演算子は、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、単項否定演算子は False (0) になります。オペランドが False (0) の場合、単項否定演算子は True (1) になります。

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
```

比較演算子で単項否定演算子を使用すると、演算子が実行する演算の意味が反転します。事実上、演算の結果が反転されます。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
```

5.5.3 二項論理積演算子 (& または &&)

二項論理積演算子は、オペランドの両方の値が True (1) であるかどうかを判断します。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。

二項論理積演算子には、& と && という 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。

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)
```

これは、1 を返します。

ObjectScript

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

これは、1 を返します。

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)
```

上記は、両方とも False (0) を返します。

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"}
}
```

5.5.4 二項論理和演算子 (! または ||)

二項論理和演算子は、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。二項論理和演算子は、両方のオペランドが False (0) の場合にのみ False (0) を返します。

二項論理積演算子には、! (感嘆符) および || (2 本の縦バー) 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。

以下の例は、2 つの True の (ゼロではない) オペランドに二項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 のオペランドに二項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。

5.5.5 論理積否定 (NAND) 演算子 ('&)

以下に相当するいずれかの形式で、二項論理積演算子 (&) と単項否定演算子を併用して、論理積否定演算 (NAND) 演算を指定できます。

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

論理積否定演算は、両方のオペランドに適用された & 二項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。

&& 二項論理積演算子の前に、単項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)
```

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、&& 二項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、単項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。単項否定演算は、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
```

5.5.6 論理和否定 (NOR) 演算子 ('!)

論理和否定演算 (NOR) は、以下のいずれかの等価形式で、単項否定演算子と二項論理和演算子！を組み合わせて記述できます。

```
operand '! operand '(operand ! operand)
```

論理和否定演算は、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。

|| 二項論理和演算子の前に、単項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)
```

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。単項否定演算は、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
```

5.6 文字列連結演算子 (_)

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。

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。

最大文字列サイズは 3,641,144 文字です。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。

5.6.1 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。

5.7 文字列関係演算子

数値関係演算子と文字列関係演算子という、2 種類の関係演算子があります。文字列関係演算子は、オペランドを文字列として解釈してブーリアン値の結果を返します。文字列関係演算子は、否定論理演算子 (') を先頭に付けて、論理結果を反転することができます。

5.7.1 二項等値演算子 (=)

二項等値演算子は、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。二項等値演算子を 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。次に例を示します。

ObjectScript

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

上記は、True (1) を返します。

二項等値演算子は、どちらのオペランドも数値的に解釈しません。例えば、以下の文は、2 つのオペランドが数値的には等価ですが、False (0) を返します。

ObjectScript

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

上記は、False (0) を返します。

両方のオペランドが数値である場合、二項等値演算子を使用して、数値的に等しいかどうかを判断できます。次に例を示します。

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。次に例を示します。

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。次に例を示します。

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)
```

5.7.2 不等関係演算子 ('=)

不等関係演算は、単項否定演算子と二項等値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、不等関係演算を記述できます。

```
operand '= operand
'(operand = operand)
```

不等関係演算は、二項等値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。

5.7.3 二項包含演算子 ([)

二項包含関係子は、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。

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
```

上記は、True (1) を返します。

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
```

上記は、False (0) を返します。

5.7.4 非包含演算子 ('[)

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。

ObjectScript

```
SET P="Beatles", S="Mick Jagger"
WRITE P'[S
```

これは、1 を返します。

5.7.5 二項後続関係演算子 (])

二項後続関係演算子は、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。二項後続関係演算子は、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が、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が、それ以外は同一の場合

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"
```

上記は、True (1) を返します。

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
```

上記は、True (1) を返します。

5.7.6 非後続演算子 ('])

非後続演算は、以下のどちらかの等価形式で、二項後続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)
```

非後続演算は、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。

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。

5.7.7 二項前後関係演算子 (]])

二項前後関係演算子は、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。

二項前後関係演算子は、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。次に例を示します。

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。

5.7.8 非前後関係演算子 ('])

非前後関係演算は、以下のいずれかの等価形式でも、二項後続関係演算子と共に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)
```

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。

5.8 パターン・マッチ演算子 (? または '?)

InterSystems IRIS では、以下の 2 系統のパターン・マッチングをサポートしています。

- ObjectScript のパターン・マッチング：ここで説明するパターン・マッチングは、疑問符 (?) またはその否定 ('?) でパターン文字列の先頭部を定める構文です。
- 正規表現：多くのソフトウェア・ベンダよりサポートされる (バリアントを用いた) パターン・マッチ構文です。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。それらについては、このドキュメントの個別の章にて説明しています。

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。

5.8.1 ObjectScript のパターン・マッチング

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証番号 (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
```

引数	説明
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

引数	説明
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 5-2: 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 $WC(131083) は、?1ZFWCHARZ に一致します。このパターン・マッチング・コードには、日本語ロケールが必要となります。詳細は、"$ZZENKAKU" 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、"$ZZENKAKU" 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。

"インターシステムズの用語集" に記載のとおり、ASCII 文字セットとは、文字数が少ない 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指しています。

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、二項包含関係演算子 ([) とは異なります。二項包含関係演算子は、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、二項包含関係式は、パターン・マッチング演算子で使用できる一連のオプションを提供しません。二項包含関係式では、単一の文字列のみ右辺のオペランドで使用できます。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

5.8.2 パターンの発生回数の指定

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。以下の例を考えてみます。

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。

5.8.3 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。以下の例を考えてみます。

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。

5.8.4 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。

5.8.5 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。

5.8.6 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。次に例を示します。

```
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。次に例を示します。

```
(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。

5.8.7 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、文字列が残っていてはなら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
```

5.8.8 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。

5.8.9 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単項否定演算子 (') 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
```

5.8.10 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。

5.9 間接演算子 (@)

ObjectScript の間接演算子 (@) は、変数に間接的に値を割り当てることができます。間接演算とは、コマンド行、コマンド、コマンド引数の一部あるいはすべてを、データ・フィールドの内容で、実行時にダイナミックに置換する手段です。InterSystems IRIS 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、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実行できます。

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、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、置換する値を取得する変数を識別します。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す。変数は配列ノードにできます。

以下のルーチンは、間接演算がその右側にある変数全体の値を参照することを示します。

ObjectScript

```
IndirectionExample
 SET x = "ProcA"
 SET x(3) = "ProcB"
 ; The next line will do ProcB, NOT ProcA(3)
 DO @x(3)
 QUIT
ProcA(var)
 WRITE !,"At ProcA"
 QUIT
ProcB(var)
 WRITE !,"At ProcB"
 QUIT
```

InterSystems IRIS には、以下の 5 つの間接演算があります。

- 名前間接演算
- パターン間接演算
- 引数間接演算
- 添え字間接演算
- $TEXT 引数間接指定

@変数が発生するコンテキストによって、実行する間接演算が異なります。各間接演算の説明を以下に示します。

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。

5.9.1 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、このドキュメントの ″ユーザ定義コード″ の章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。以下の例では、InterSystems IRIS は変数 B に 6 を設定します。

ObjectScript

```
 SET Y = "B",@Y = 6
```

重要　呼び出しが間接演算を使用して、オブジェクト・プロパティの値を取得または設定しようとした場合、エラーが発生する可能性があります。この種の呼び出しは、プロパティのアクセサ・メソッド (<PropertyName>Get と <PropertyName>Set) をバイパスしようとするので、使用しないでください。代わりに、このような用途を目的としている $CLASSMETHOD、$METHOD、および $PROPERTY の各関数を使用します。詳細は、″クラスの定義と使用″ の “ObjectScript でのオブジェクトの使用法” の章にある “オブジェクトへの動的アクセス” のセクションを参照してください。

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
```

5.9.2 パターン間接演算

パターン間接演算は、特殊な形式の間接演算です。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・マッチに関しては、"パターン・マッチング" で説明されています)。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。

5.9.3 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,!
```

5.9.4 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。以下はその例です。

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、"グローバルの使用法" の "グローバル構造" 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"
```

5.9.5 $TEXT 引数間接指定

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す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)
```

6
正規表現

InterSystems IRIS® Data Platform 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、InterSystems IRIS パターン・マッチング演算子を使用します。

この章では、正規表現の以下の機能について説明します。

- ワイルドカードと修飾子。例えば、`.*` は、任意のタイプの任意の数の文字に一致します。
- リテラルと文字の範囲。例えば、`[A-Z]` は、A から Z までの範囲にある単一の大文字に一致します。
- 文字タイプ・メタ文字は、文字のグループに一致するシーケンスです。
  - 単一文字タイプ。例えば、`\d` は、数字に一致します。
  - Unicode プロパティ文字タイプ。例えば、`\p{LL}` は、小文字に一致します。
  - POSIX 文字タイプ。例えば、`[:print:]` は、印刷可能文字に一致します。
- グループ化構文では括弧を使用して、繰り返して正規表現を適用します。例えば、`(\p{LL})+` では、それぞれの文字が小文字かどうかを判別します。
- アンカーにより、一致が可能な場所を制限します。例えば、`\b(day)` は、単語境界に出現する “day” のみに一致します。
- 論理演算子。例えば、`[[:upper:]&&[:greek:]]` は、大文字のギリシャ文字に一致します。
- 文字表現メタ文字は、単一文字に一致するシーケンスです。
  - 16 進、8 進、および Unicode の表現。例えば、`\x5A` は、英字の Z を 16 進数で表したものです。
  - 制御文字表現。例えば、`\cM` は、キャリッジ・リターン制御文字です。
  - 記号名表現。例えば、`\N{equals sign}` は、= 文字です。
- モード。例えば、`(?i)` では、以後すべてにおける一致では大文字と小文字は区別されません。
- コメント。例えば、`(?# date and 24-hour time)` では、このコメントを正規表現の文字列に挿入します。
- エラー・メッセージ。

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。

6.1 ワイルドカードと修飾子

式	説明
.	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、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、regexp ".*\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)。regexp 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。

6.2 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 ("") 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。

式	説明
[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) (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。

6.3 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て使用できます。

6.3.1 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)。

文字	説明
¥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 から 9 までの数字です。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。

6.3.2 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。

以下のテーブルは、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)。

Unicode 文字のキーワード	一致する値
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)

Unicode 文字のキーワード	一致する値
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)
PF	最後の句読点 (187)
PI	最初の句読点 (171)
PO	その他の句読点 (33-35、37-39、42、44、46-47、58-59、63-64、92、161、183、191)
PS	開始句読点 (40、91、123)
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)

Unicode 文字のキーワード	一致する値
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),!} }
```

6.3.3 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 ー 文字と数字。
- alpha ー 文字。
- blank ー タブ ($CHAR(9)) またはスペース (CHAR(32)、CHAR(160))。
- cntrl ー 制御文字 ($CHAR(0) ～ $CHAR(31)、$CHAR(127) ～ $CHAR(159))。
- digit ー 数字 (0 ～ 9)。
- graph ー スペース文字を除く出力可能文字 ($CHAR(33) ～ $CHAR(126)、$CHAR(161) ～ $CHAR(156))。
- lower ー 英字の小文字。
- math ー 算術文字 (記号のサブセット)。`+<=>^|~¬±×÷` の文字が含まれます。
- print ー スペース文字を含む出力可能文字 ($CHAR(32) ～ $CHAR(126)、$CHAR(160) ～ $CHAR(156))。
- punct ー 句読点文字 (記号文字を除きます)。`!"#%&'()*,-./:;?@[\]_{}¡«·»¿` の文字が含まれます。
- space ー $CHAR(9)、$CHAR(10)、$CHAR(11)、$CHAR(12)、$CHAR(13)、$CHAR(32)、$CHAR(133)、および $CHAR(160) の文字を含む、空白、タブ、および行スペースの文字を含めた、スペース文字。
- symbol ー 記号文字 (句読点文字を除きます)。`$+<=>^`|~¢£¤¥¦§¨©¬®¯°±´¸×÷` の文字が含まれます。
- upper ー 英字の大文字。
- xdigit ー 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),!}
}
```

6.4 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 $MATCH("1111111111,2222222222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("11111x11111,2222222222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,22x22222222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,2222222x222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,22222222x22,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b
```

6.5 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。

6.5.1 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。

アンカー	説明
^ ¥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に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に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に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="ABCDEFG"
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="ABCDEFG"_$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
```

6.5.2 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)
```

6.6 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。

6.7 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。

6.7.1 16 進、8 進、および Unicode の表現

表現	説明
¥xnn ¥x{nnn}	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{nnn} 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。

6.7.2 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_)。

6.7.3 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 (—)、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 (¯)、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。

6.8 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。.例えば、(?i:(fred|ginger)) です。

以下の mode 文字がサポートされます。

mode 文字	説明
(?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。

6.8.1 正規表現シーケンスのモード

regexp 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 }
```

6.8.2 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 文字以上その後に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 }
```

6.9 コメント

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)

6.9.1 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 $MATCH("04/28/2012","^[01]\d(?# months)/[0123]\d(?# days)/\d\d\d\d$")
```

6.9.2 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。

6.10 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、″InterSystems IRIS エラー・リファレンス″ の ″一般的なエラー・メッセージ″ で 8300 ～ 8352 を参照してください。

7
コマンド

コマンドは、InterSystems IRIS® Data Platform における ObjectScript プログラミングの基本的なコード・ユニットです。ObjectScript の実行タスクは、すべてコマンドで実行されます。すべてのコマンドは、コマンド・キーワードと (ほとんどの場合) これに続く 1 つまたは複数のコマンド引数から構成されます。

この章では、ObjectScript コマンドの以下の側面について説明します。

- コマンド・キーワード
- コマンド引数
- 後置条件
- 単一行での複数コマンド

この章では、次に ObjectScript コマンドの以下のグループについて簡単に説明します。

- 変数の割り当てを管理するコマンド – SET、KILL、および NEW
- 他のコマンドを実行するためのコンテキスト・コマンド – TRY および CATCH エラー処理構造。TSTART、TCOMMIT、および TROLLBACK トランザクション処理。LOCK リソース・ロック
- コードの呼び出し (およびコードの終了) を行うコマンド – DO、JOB、および XECUTE。QUIT および RETURN
- フロー制御コマンド – IF、ELSEIF、および ELSE。FOR。WHILE および DO WHILE
- 入出力コマンド – 4 つの表示 (書き込み) コマンド。READ。OPEN、USE、および CLOSE

これは、ObjectScript コマンドの完全な一覧ではありません。これらおよび他の多くの ObjectScript コマンドについては、″ObjectScript リファレンス″ で詳しく説明しています。

7.1 コマンド・キーワード

ObjectScript では、すべてのコマンド文は明示的です。ObjectScript コードの実行可能行は、すべてコマンド・キーワードで始まる必要があります。例えば、変数に値を代入するには、SET キーワードを指定し、続いて変数の引数および代入する値を指定する必要があります。

コマンド・キーワードは、常にキーワードで始まります。以下の例を考えてみます。

ObjectScript

```
WRITE "Hello"
```

“WRITE” という単語がコマンド・キーワードです。コマンドは、実行するアクションを指定します。WRITE コマンドは、その名前から分かるように、引数 として指定された内容を主デバイスに書き込みます。この場合、WRITE コマンドは文字列 “Hello” を書き込みます。

ObjectScript のコマンド名では、大文字と小文字は区別されません。ほとんどのコマンド名は省略形で表現できます。したがって、“WRITE”、“Write”、“write”、“W”、および “w” はいずれも、WRITE コマンドの有効な形式です。コマンドの省略形のリストは、″ObjectScript リファレンス″ の ″省略形テーブル″ を参照してください。

コマンド・キーワードは予約語ではありません。したがって、コマンド・キーワードを変数のユーザ割り当て名、ラベル、またはその他の識別子として使用できます。

ObjectScript プログラムでは、コード行の最初のコマンドをインデントにする必要があります。コマンド・キーワードは列 1 では使用できません。ターミナル・コマンド行プロンプトから、または XECUTE コマンドからコマンドを発行する場合、インデントは必要ありません (インデントにしてもかまいません)。

コードの実行可能行は、それぞれが独自のコマンド・キーワードを持つ 1 つまたは複数のコマンドを含むことができます。単一行における複数コマンドは 1 つ以上のスペースで区切ります。1 つ以上のコマンドは同一行のラベルの後に続きます。ラベルとコマンドは 1 つ以上のスペースで区切ります。

ObjectScript では、コマンド末端や行末の区切り文字が必要なく、または不許可となります。コマンドの後には、行内コメントを指定して、コマンド行の残りの部分がコメントであることを示すことができます。コマンドの末端とコメント構文の間には空白スペースが必要になりますが、`##;` および `/* comment */` の構文については例外となります。`/* comment */` の複数行コメントは、1 つのコマンド内かつコマンド末端にて指定可能です。

7.2 コマンド引数

コマンド・キーワードの後ろには、オブジェクト (複数も可) またはコマンドの範囲を指定する 1 つ以上の引数を置くことができます。また、引数を置かない場合もあります。コマンドが 1 つ以上の引数を取得する場合、コマンド・キーワードと最初の引数の間にスペースを 1 つだけ置く必要があります。例えば以下のようになります。

ObjectScript

```
SET x = 2
```

最初の引数の最初の文字が (上記のように) 正確に 1 つのスペースでコマンド自体から区切られている限り、引数内または引数間でスペースを使用できます。したがって、以下はすべて有効です。

ObjectScript

```
SET a = 1
SET b=2
SET c=3,d=4
SET e= 5   ,  f =6
SET g
    =             7
WRITE a,b,c,d,e,f,g
```

コマンドが 後置条件 を取る場合、コマンド・キーワードと後置条件間にスペースを置くことはできません。また、後置条件と 1 番目の引数の最初に必ず 1 つのスペースを置きます。以下の形式の QUIT コマンドはすべて有効です。

ObjectScript

```
QUIT x+y
QUIT x + y
QUIT:x<0
QUIT:x<0 x+y
QUIT:x<0 x + y
```

引数間にスペースは必要ありませんが、複数の空白スペースを引数間で使用する場合もあります。これらの空白スペースは、コマンドの実行に影響を与えません。また、改行、タブ、コメントもコマンド引数内や引数間に含まれますが、コマンドの実行に影響はありません。詳細は、このドキュメントの“構文規則”の章の″空白″を参照してください。

7.2.1 複数の引数

多くのコマンドでは、複数の独立引数を指定することができます。コマンド引数の区切り記号はコンマ“,”です。つまり、1 つのコマンドの後ろに、コンマで区切られたリストとして複数の引数を指定します。例えば以下のようになります。

ObjectScript

```
SET x=2,y=4,z=6
```

Set コマンドは、3 つの引数を使用して、指定した 3 つの変数に値を割り当てます。ここでは、複数の引数が繰り返されます。つまり、コマンドは、指定された順番で各引数に個別に割り当てられます。内部的には、InterSystems IRIS がこれを 3 つの別個 SET コマンドと解析します。デバッグ時においては、これらの複数の引数それぞれが別個の手順となります。

コマンド・リファレンス・ページで紹介しているコマンド構文では、繰り返して記述できる引数の後にコンマと省略記号 `,...` が続きます。このコンマは引数に必要な区切り文字です。また、省略記号 (...) は、不特定の数の繰り返し引数を指定できることを示しています。

繰り返し引数は、厳密に左から右の順で実行されます。したがって、以下のコマンドが有効となります。

ObjectScript

```
SET x=2,y=x+1,z=y+x
```

ただし、以下のコマンドは無効です。

ObjectScript

```
SET y=x+1,x=2,z=y+x
```

各繰り返し引数は、指定した順序で独立して実行されるため、無効な引数が検出されるまで有効な引数が実行されます。以下の例では、SET x では値が x に代入され、SET y では <UNDEFINED> エラーが発生し、SET z が評価されていないため、<DIVIDE> (0 による除算) エラーは検出されません。

ObjectScript

```
KILL x,y,z
SET x=2,y=z,z=5/0
WRITE "x is:",x
```

7.2.2 パラメータおよび後置条件付きの引数

いくつかのコマンド引数は、パラメータも受け取ります (″関数″で説明する関数パラメータと混同しないでください)。指定する引数がパラメータを取得できる場合、パラメータの区切り記号はコロン“:”です。

以下のコマンド例では、引数の区切り記号として使用されているコンマと、パラメータ区切り記号として使用されているコロンを示しています。ここでは、2 つの引数があり、各引数には 3 つのパラメータがあります。

ObjectScript

```
VIEW X:y:z:a,B:a:y:z
```

少数のコマンド (DO、XECUTE、および GOTO) では、引数に続くコロンは、その引数を実行するかどうかを決める後置条件式を指定します。

7.2.3 引数なしコマンド

引数を持たないコマンドは、引数なしコマンドといいます。キーワードに追加された後置条件式は引数とはみなされません。

常に引数を持たないコマンドが少数あります。例えば、HALT、CONTINUE、TRY、TSTART および TCOMMIT は、引数なしコマンドです

いくつかのコマンドは必要に応じて引数なしになります。例えば、BREAK、CATCH、FOR、GOTO、KILL、LOCK、NEW、QUIT、RETURN、TROLLBACK、WRITE、および ZWRITE には、すべて引数なしの構文形式があります。このような場合、同じコマンドでも引数ありと引数なしでは、意味が多少異なる場合があります。

引数なしコマンドを行の最後で使用する場合、末尾のスペースは必要ありません。引数なしコマンドを他のコマンドと同一のコード行で使用する場合、引数なしコマンドと後続のコマンドの間には 2 つ (あるいはそれ以上) のスペースを置く必要があります。例えば以下のようになります。

ObjectScript

```
QUIT:x=10  WRITE "not 10 yet"
```

この場合、QUIT は後置条件式を持つ引数なしのコマンドです。このコマンドと次のコマンドの間に、少なくとも 2 つのスペースが必要です。

7.2.3.1 引数なしコマンドと中括弧

中括弧で区切られたコマンド・ブロック内で引数なしコマンドを使用する場合、空白の制限はありません。

- 引数なしコマンドの直後に左中括弧が続く場合、コマンド名と中括弧の間に空白は必要ありません。何も指定しないか、1 つあるいは複数のスペース、タブ、または改行を指定します。これは、FOR などのように引数を使用できる引数なしコマンド、および ELSE などのように引数を使用できない引数なしコマンドのどちらにも当てはまります。

  ObjectScript

  ```
  FOR  {
     WRITE !,"Quit out of 1st endless loop"
     QUIT
  }
  FOR{
     WRITE !,"Quit out of 2nd endless loop"
     QUIT
  }
  FOR
  {
     WRITE !,"Quit out of 3rd endless loop"
     QUIT
  }
  ```

- 右中括弧は区切り文字として機能するため、引数なしコマンドの直後に右中括弧が続く場合、後続のスペースは必要ありません。例えば以下の方法で、引数なしの QUIT を使用します。

  ObjectScript

  ```
  IF 1=2 {
     WRITE "Math error"}
  ELSE {
     WRITE "Arthmetic OK"
     QUIT}
  WRITE !,"Done"
  ```

7.3 コマンド後置条件式

ほとんどの場合、ObjectScript のコマンドを指定する際、後置条件 を追加することができます。

後置条件とは、コマンドや (時には) コマンド引数に追加されるオプションの式で、InterSystems IRIS がそのコマンドやコマンド引数を実行するかどうかを制御します。後置条件式の評価が True (ゼロ以外の値) の場合、InterSystems IRIS はそのコマンドあるいはコマンド引数を実行します。また、後置条件式の評価が False (ゼロ) の場合、InterSystems IRIS はそのコマンドあるいはコマンド引数を実行せず、次のコマンドあるいはコマンド引数に実行を移します。

すべての ObjectScript コマンドでは、フロー制御コマンド (IF、ELSEIF、ELSE、FOR、WHILE、および DO WHILE) 以外、およびブロック構造エラー処理コマンド (TRY、THROW、CATCH) 以外で、後置条件式を使用できます。

ObjectScript コマンド DO と XECUTE は、コマンド・キーワードとコマンド引数の両方に後置条件式を追加できます。後置条件式は常にオプションです。例えば、コマンドの引数の中でも、後置条件式が追加されている引数と追加されていない引数があります。

コマンド・キーワードとコマンドの引数の両方が後置条件を持つ場合、キーワード後置条件が最初に評価されます。このキーワード後置条件が True に評価された場合にのみ、コマンド引数の後置条件が評価されます。コマンド・キーワードの後置条件が False の場合、コマンドは実行されず、プログラムは次のコマンドに移って実行を続けます。コマンド引数の後置条件が False の場合、引数は実行されず、コマンドは次の引数を左から右の順で実行を続けます。

7.3.1 後置条件構文

コマンドに後置条件を追加するには、コマンド・キーワードの直後にコロン (:) と式を置きます。後置条件式を持つコマンドの構文は以下のようになります。

```
Command:pc
```

Command はコマンド・キーワード、コロンは必須のリテラル文字で、pc は任意の有効な式にできます。

コマンド後置条件には、以下の構文規則があります。

- スペース、タブ、改行、コメントは、コマンド・キーワードと後置条件式の間、あるいはコマンド引数と後置条件式の間には使用できません。コロン文字の前後にスペースを使用することはできません。
- スペース、タブ、改行、コメントは、後置条件式内では使用できません。ただし、後置条件式全体が括弧で囲まれている場合や、括弧で囲まれた引数リストが後置条件式にある場合は使用できます。スペース、タブ、改行、コメントは括弧内で使用できます。
- 後置条件式の後に置く必要があるスペースは、コマンド・キーワードの場合と同様です。キーワード後置条件式の最後の文字と 1 番目の引数の最初の文字の間には、スペースを正確に 1 つ置きます。引数なしコマンドでは、後置条件の直後に右中括弧が続かない限り、後置条件式の最後の文字と同じ行にある次のコマンドの間に、2 つ以上のスペースを置く必要があります。(括弧を使用する場合、閉じ括弧は後置条件式の最後の文字として処理されます。)

後置条件式は、厳密に言えばコマンド引数ではありません (しかし、ObjectScript リファレンス・ページで、後置条件式は、引数セクションの一部として示されています)。後置条件は常にオプションです。

7.3.2 後置条件の評価

InterSystems IRIS は、後置条件式を True あるいは False として評価します。通常、これらは 1 あるいは 0 の値で評価することをお勧めします。しかし、InterSystems IRIS はあらゆる値の後置条件式を実行し、式の値が 0 の場合は False、ゼロ以外の場合は True として評価します。

- InterSystems IRIS は、有効なゼロ以外の数値を True として評価します。算術演算のように、有効な数値に対し同じ評価基準を使用します。したがって、1、“1”、007、3.5、-.007、7.0、“3 little pigs”、$CHAR(49)、0_″1″ はすべて、True と評価されます。
- InterSystems IRIS は、ゼロ (0) 値およびすべての非数値 (空白文字 (″ ″) が含まれる文字列や NULL 文字列 (″″) など) を False として評価します。したがって、0、-0.0、“A”、“-”、“$”、“The 3 little pigs”、$CHAR(0)、$CHAR(48)、″0_1″ はすべて、False と評価されます。
- 一般的な等式の規則が適用されます。したがって、0=0、0=″0″、″a″=$CHAR(97)、0=$CHAR(48)、(″″=$CHAR(32)) は、True と評価されます。一方、0=″″、0=$CHAR(0)、(″″=$CHAR(32)) は False と評価されます。

次の例では、どの WRITE コマンドが実行されるかは、変数 count の値によって異なります。

ObjectScript

```
FOR count=1:1:10 {
  WRITE:count<5 count," is less than 5",!
  WRITE:count=5 count," is 5",!
  WRITE:count>5 count," is greater than 5",!
}
```

7.4 単一行での複数コマンド

ObjectScript のソース・コードの単一行には、複数のコマンドおよびそれらの引数を含めることができます。これらは厳密に左から右の順序で実行され、別々の行で記述したコマンドと機能的に同一となります。引数付きのコマンドは、単一の空白文字によって後続コマンドと区切る必要があります。引数なしのコマンドは、2 つの空白文字によって後続コマンドと区切る必要があります。ラベルには、同一行にて 1 つ以上のコマンドを後続させることができます。コメントは、同一行にて 1 つ以上のコマンドの後に続けることができます。

ソース・コードの行の最大長については、″サーバ側プログラミングの入門ガイド″ の付録 ″一般的なシステム制限″ を参照してください。スタジオを使用してソース・コードを記述/編集する場合、この制限は異なることがあります。

7.5 変数割り当てコマンド

変数の管理に必要な機能を提供するため、ObjectScript は以下のコマンドを備えています。

- SET は、値を変数に代入します。
- KILL は、変数への値の割り当てを削除します。
- NEW は、変数割り当ての新しいコンテキストを作成します。

7.5.1 SET

SET コマンドは、変数に値を割り当てます。単一の変数または複数の変数に、一度に値を割り当てることができます。

以下は SET の基本的な構文です。

ObjectScript

```
SET variable = expression
```

これにより、1 つの変数の値が設定されます。また、以下のようにいくつかの手順があります。

- ObjectScript は、value 式を評価し、値を決定します (可能な場合)。この手順では、式に未定義変数、無効な構文 (例えばゼロによる除算)、他のエラーが含まれている場合、エラーを生成します。
- 変数が存在しない場合は、ObjectScript が変数を生成します。
- 一度変数が生成されると、あるいは変数が既に存在している場合、ObjectScript はその式の変数に値を設定します。

以下の構文を使用して、複数の変数それぞれに値を設定します。

ObjectScript

```
SET variable1 = expression1, variable2 = expression2, variable3 = expression3
```

以下の構文を使用して、1 つの式と等しい複数の変数を設定します。

ObjectScript

```
SET (variable1,variable2,variable3)= expression
```

例えば、以下のコードを使用して、Person クラス・インスタンスの Gender プロパティ値を設定します。

ObjectScript

```
SET person.Gender = "Female"
```

ここで、person は、Person クラスの関連インスタンスへのオブジェクト参照です。

複数の Person オブジェクトの Gender プロパティを同時に設定することもできます。

ObjectScript

```
SET (per1.Gender, per2.Gender, per3.Gender) = "Male"
```

ここで、per1、per2、per3 は、Person クラスの 3 つの異なるインスタンスへのオブジェクト参照です。

SET を使用して、値を返すメソッドを呼び出すことができます。メソッドを呼び出す場合、SET では、メソッドの返り値と同一の変数、グローバル参照、またはプロパティを設定できます。引数の形式は、メソッドがインスタンスかクラス・メソッドかどうかにより異なります。クラス・メソッドを呼び出すには、以下の構文を使用します。

ObjectScript

```
SET retval = ##class(PackageName.ClassName).ClassMethodName()
```

ClassMethodName() は呼び出すクラス・メソッド名、ClassName はメソッドを含むクラス名、PackageName はクラスを含むパッケージ名です。メソッドの返り値は retval ローカル変数に割り当てられます。##class() 構文は、コードに必要なリテラル部です。

インスタンス・メソッドを呼び出すには、ローカルでインスタンスを生成されたオブジェクトへのハンドルのみが必要です。

ObjectScript

```
SET retval = InstanceName.InstanceMethodName()
```

InstanceMethodName() は呼び出すインスタンス・メソッド名、InstanceName はメソッドを含むインスタンス名です。メソッドの返り値は retval ローカル変数に割り当てられます。

詳細は、"ObjectScript リファレンス" の "SET" コマンドを参照してください。

7.5.2 KILL

KILL コマンドは、メモリから変数を削除し、また、ディスクからも削除できます。以下はその基本的な形式です。

ObjectScript

```
KILL expression
```

expression は、削除する 1 つ以上の変数です。以下は、KILL の一番簡単な形式です。

ObjectScript

```
KILL x
KILL x,y,z
```

“排他的 KILL”と呼ばれる KILL の特殊な形式は、指定された変数以外のすべてのローカル変数を削除します。排他的 KILL を使用するには、括弧内にその引数を置きます。例えば、変数 x、y、z がある場合、以下を実行して x 以外の y、z、または任意のローカル変数を削除できます。

ObjectScript

```
KILL (x)
```

引数がない場合、KILL はすべてのローカル変数を削除します。

詳細は、″ObjectScript リファレンス″ の ″KILL″ コマンドを参照してください。

7.5.3 NEW

NEW コマンドは、新しいローカル変数のコンテキストを作成します。つまり、既存のローカル変数値を古いコンテキストに保存し、ローカル変数に値が代入されていない新しいコンテキストを開始します。プロシージャを使用するアプリケーションでは、NEW を使用して、アプリケーション全体、あるいはアプリケーションの主要なサブシステムの変数を初期化します。

以下の構文形式がサポートされます。

ObjectScript

```
NEW          // initiate new context for all local variables
NEW x        // initiate new context for the specified local variable
NEW x,y,z    // initiate new context for the listed local variables
NEW (y)      // initiate new context for all local variables except the specified variable
NEW (y,z)    // initiate new context for all local variables except the listed variables
```

詳細は、″ObjectScript リファレンス″ の ″NEW″ コマンドを参照してください。

7.6 コード実行コンテキスト・コマンド

以下のコマンドは、コマンドのグループの実行をサポートするために使用されます。

- エラー処理のための TRY / CATCH ブロック構造。エラー処理のためにブロック構造を作成するには、TRY コマンドおよび CATCH コマンドを使用することをお勧めします。TRY ブロックには、目的の操作を実行する複数のコマンドが含まれています。各 TRY ブロックは、TRY ブロックでエラーが発生したときに呼び出される CATCH エラー処理ブロックと組み合わされています。各 TRY ブロックのすぐ後に、対応する CATCH ブロックが続く必要があります。複数の TRY / CATCH ブロックのペアを必要に応じてプログラムで作成できます。TRY / CATCH ブロックのペアを入れ子にすることも可能です。TRY ブロック内から THROW コマンドを使用して、対応する CATCH ブロックを明

示的に呼び出すことができます。詳細は、このドキュメントの“エラー処理”の章の″TRY-CATCH メカニズム″を参照してください。″ObjectScript リファレンス″の″TRY″コマンド、″THROW″コマンド、および″CATCH″コマンドを参照してください。

- トランザクション処理用の TSTART コマンド、TCOMMIT コマンド、および TROLLBACK コマンド。(全か無かの単一のコード単位として) コマンドのグループをアトミックに実行する必要がある場合、コマンドのグループを TSTART コマンドで始め、TCOMMIT コマンドで終えることをお勧めします。実行中に問題が発生した場合、TROLLBACK コマンドを発行して、コマンドのグループで実行された操作をロールバックする必要があります。詳細は″トランザクション処理″の章を参照してください。″ObjectScript リファレンス″の″TSTART″コマンド、″TCOMMIT″コマンド、および″TROLLBACK″コマンドを参照してください。
- リソースをロックまたはロック解除するための LOCK コマンド。詳細は″トランザクション処理″の章を参照してください。詳細は、″ObjectScript リファレンス″の″LOCK″コマンドを参照してください。

7.7 コードの呼び出し

この節では、1 つまたは複数のコマンドの実行を呼び出すために使用するコマンドについて説明します。

- DO
- JOB
- XECUTE
- QUIT および RETURN

7.7.1 DO

ObjectScript で任意のルーチン、プロシージャ、またはメソッドを呼び出すには、DO コマンドを使用します。以下は DO の基本的な構文です。

ObjectScript

```
DO ^CodeToInvoke
```

CodeToInvoke は、InterSystems IRIS システム・ルーチンあるいはユーザ定義ルーチンです。キャレット文字“^”は、ルーチン名の直前に付ける必要があります。

また、ルーチン内でプロシージャの開始場所を示すラベル (タグともいいます) を参照して、プロシージャを実行することもできます。ラベルは、キャレットの直前に置かれます。以下はその例です。

ObjectScript

```
SET %X = 484
DO INT^%SQROOT
WRITE %Y
```

このコードは、%X システム変数値を 484 に設定します。その後 DO コマンドを使用して、InterSystems IRIS システム・ルーチン %SQROOT の INT プロシージャを呼び出し、%X の平方根を算出して、%Y に格納します。その後、WRITE コマンドを使用して、%Y の値を表示します。

メソッドの呼び出し時、DO は 1 つの引数として、メソッドを指定する式全体を取得します。引数の形式は、メソッドがインスタンスかクラス・メソッドかどうかにより異なります。クラス・メソッドを呼び出すには、以下の構文を使用します。

ObjectScript

```
DO ##class(PackageName.ClassName).ClassMethodName()
```

ClassMethodName() は呼び出すクラス・メソッド名、ClassName はメソッドを含むクラス名、PackageName はクラスを含むパッケージ名です。##class() 構文は、コードに必要なリテラル部です。

インスタンス・メソッドを呼び出すには、ローカルでインスタンスを生成されたオブジェクトへのハンドルのみが必要です。

ObjectScript

```
DO InstanceName.InstanceMethodName()
```

InstanceMethodName() は呼び出すインスタンス・メソッド名、InstanceName はメソッドを含むインスタンス名です。

詳細は、"ObjectScript リファレンス" の "DO" コマンドを参照してください。

7.7.2 JOB

DO はフォアグラウンドでコードを実行し、JOB はバックグラウンドで実行します。これは、通常ユーザとの対話なしに、現在のプロセスで単独に実行されます。ジョブ起動プロセスは、明示的に指定された場合を除き、すべてのシステムの既定値を継承します。

詳細は、"ObjectScript リファレンス" の "JOB" コマンドを参照してください。

7.7.3 XECUTE

XECUTE コマンドは、1 つ以上の ObjectScript コマンドを実行し、引数として取得する式を評価します (その引数は、1 つ以上の ObjectScript コマンドを含む文字列に評価される必要があります)。実際、各 XECUTE 引数は、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 XECUTE 引数の直後に返されます。

詳細は、"ObjectScript リファレンス" の "XECUTE" コマンドを参照してください。

7.7.4 QUIT および RETURN

QUIT コマンドと RETURN コマンドは、両方ともメソッドを含むコード・ブロックの実行を終了させます。引数がない場合、呼び出し元のコードを単に終了します。引数がある場合、返り値としてその引数を使用します。QUIT は、現在のコンテキストを終了し、囲んでいるコンテキストに移動します。RETURN は、現在のプログラムを終了し、プログラムが呼び出された場所に移動します。

以下のテーブルは、QUIT を使用するか RETURN を使用するかを選択する方法を示しています。

Location	QUIT	RETURN
ルーチン・コード（ブロック構造になっていない）	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。
TRY または CATCH ブロック	TRY / CATCH ブロック構造のペアを終了し、ルーチンの次のコードに移動します。入れ子にされた TRY または CATCH ブロックから発行された場合、1 レベルを終了し、囲んでいる TRY または CATCH ブロックに移動します。	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。
DO または XECUTE	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。
IF	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。ただし、FOR、WHILE、または DO WHILE ループで入れ子にされている場合、そのブロック構造を終了し、コード・ブロックの後の次の行で続行します。	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。
FOR、WHILE、DO WHILE	ブロック構造を終了し、コード・ブロックの後の次の行で続行します。入れ子にされたブロックから発行された場合、1 レベルを終了し、囲んでいるブロックに移動します。	ルーチンを終了し、呼び出し元のルーチン（存在する場合）に戻ります。

詳細は、"ObjectScript リファレンス" の "QUIT" コマンドおよび "RETURN" コマンドを参照してください。

7.8 フロー制御コマンド

コードでロジックを構築するためには、フロー制御が必要です。コードのブロックを条件付きで実行するかまたはバイパスするか、またはコードのブロックを繰り返し実行します。そのために、ObjectScript は以下のコマンドをサポートします。

- IF、ELSEIF、および ELSE
- FOR
- WHILE と DO WHILE

7.8.1 条件付きの実行

コードのブロックを条件付きで実行するには、ブーリアン（True/False）テストに基づいて、IF コマンドを使用できます。（後置条件式を使用して、個別の ObjectScript コマンドの条件付きの実行を行うことができます。）

IF は、式を引数として取得し、式が True か False かを判断します。True の場合、式の後ろに続くコードのブロックが実行され、False の場合は実行されません。通常、これらは 1 あるいは 0 の値で評価することをお勧めします。しかし、

InterSystems IRIS はあらゆる値の条件付きの実行を行い、式の値が 0 の場合は False、ゼロ以外の場合は True として評価します。詳細は、このドキュメントの ″演算子と式″ の章を参照してください。

複数の IF ブーリアン・テスト式を、コンマ区切りのリストで指定することができます。これらのテストは、一連の論理 AND テストとして左から順に評価されます。したがって、IF は、すべてのテスト式が True に評価されると、True に評価されます。IF は、テスト式の 1 つが False と評価されると False に評価されます。残りのテスト式は評価されません。

コードは、常に多数のコマンドを含む コード・ブロック で表示されます。コード・ブロックとは、単に { } 括弧内の 1 行以上のコード行のことで、コード・ブロックの前と中には改行を置くことができます。以下の例を考えてみます。

7.8.1.1 IF、ELSEIF、および ELSE

IF 文は、複数の条件を評価し、条件によってどのコードを実行するのかを指定できます。単純なコマンドと異なり、文とは 1 つ以上のコマンド・キーワード、条件式、コード・ブロックを含みます。IF 文は、以下のものから構成されます。

- 1 つまたは複数の条件式を含む 1 つの IF 節。
- それぞれが 1 つまたは複数の条件式を含む任意の数の ELSEIF 節。ELSEIF 節 (オプション)。ELSEIF 節は複数でもかまいません。
- 条件式のない、1 つ以下の ELSE 節。ELSE 節は、オプションです。

以下は、IF 文の例です。

ObjectScript

```
READ "Enter the number of equal-length sides in the polygon: ",x
  IF x=1 {WRITE !,"It's so far away that it looks like a point"}
  ELSEIF x=2 {WRITE !,"I think that's a line, not a polygon"}
  ELSEIF x=3 {WRITE !,"It's an equalateral triangle"}
  ELSEIF x=4 {WRITE !,"It's a square"}
  ELSE {WRITE !,"It's a polygon with ",x," number of sides" }
WRITE !,"Finished the IF test"
```

詳細は、″ObjectScript リファレンス″ の ″IF″ コマンドを参照してください。

7.8.2 FOR

FOR 文を使用すると、コードのセクションを繰り返し実行できます。数値や文字列値を基に FOR ループを生成できます。

通常、FOR は、コードのそれぞれのループの開始でインクリメントあるいはディクリメントされる数値制御変数の値を基に、コード・ブロックを 0 回以上実行します。制御変数が最後の値に達したとき、制御は FOR ループを終了します。終了値がない場合、ループは QUIT コマンドに遭遇するまで実行を続けます。制御がループを終了すると、制御変数は、最後のループの実行による値を保持します。

以下は、数値 FOR ループの形式です。

ObjectScript

```
FOR ControlVariable = StartValue:IncrementAmount:EndValue {
        // code block content
}
```

すべての値は正数あるいは負数で、スペースは許可されていますが、等符号とコロンの前後には不要です。FOR に続くコード・ブロックは、変数に割り当てられた各値を繰り返します。

例えば、以下の FOR ループは 5 回実行します。

ObjectScript

```
WRITE "The first five multiples of 3 are:",!
FOR multiple = 3:3:15 {
   WRITE multiple,!
}
```

また、変数を使用して、最後の値を決めることもできます。以下の例は、ループが繰り返される回数を指定します。

ObjectScript

```
SET howmany = 4
WRITE "The first ",howmany," multiples of 3 are "
FOR multiple = 1:1:howmany {
    WRITE (multiple*3),", "
    IF multiple = (howmany - 1) {
        WRITE "and "
    }
    IF multiple = howmany {
        WRITE "and that's it!"
    }
}
QUIT
```

この例は、制御変数 multiple を使用して 3 倍しているため、式は `multiple*3` となります。また、IF コマンドを使用して、最後の計算の前に“and”を挿入しています。

注釈　この例の IF コマンドは、ObjectScript の優先順位を含む分かりやすい例です (優先順位は、階層に関係なく常に左から右です)。IF 式“multiple = howmany − 1”の括弧がない場合、あるいは全体が括弧で囲まれている場合、式の最初の部分“multiple = howmany”は、False (0) となります。したがって、式全体は“0 − 1”となり、結果は −1 であるため、この式は True となります (ループで、最後の繰り返し以外、それぞれの文字列に“and”を挿入します)。

FOR の引数は、値のリストに設定された変数にすることもできます。この場合、コード・ブロックは、変数に割り当てられたリストで、各項目に対して繰り返します。

ObjectScript

```
FOR b = "John", "Paul", "George", "Ringo" {
   WRITE !, "Was ", b, " the leader? "
 READ choice
}
```

QUIT コマンドを特定の状況で実行されるようコード・ブロック内に配置し、FOR を終了させるように指定すると、最終値のない FOR の数値形式を記述できます。これにより、繰り返し回数を示すカウンタを提供し、カウンタの値に基づかない条件を使用して、FOR を制御できます。例えば、以下のループはカウンタを使用し、推測された回数をユーザに知らせます。

ObjectScript

```
    FOR i = 1:1 {
    READ !, "Capital of MA? ", a
    IF a = "Boston" {
        WRITE "...did it in ", i, " tries"
        QUIT
        }
    }
```

カウンタが不要な場合、引数なしの FOR を使用できます。

ObjectScript

```
    FOR  {
        READ !, "Know what? ", wh
        QUIT:(wh = "No!")
        WRITE "   That's what!"
    }
```

詳細は、"ObjectScript リファレンス" の "FOR" コマンドを参照してください。

7.8.3 WHILE と DO WHILE

2 つの関連するフロー制御コマンド、WHILE と DO WHILE は、それぞれがコード・ブロックでループを実行し、条件に応じて終了します。2 つのコマンドは、条件の評価方法が異なります。WHILE はコード・ブロック全体の前に条件を評価し、DO WHILE はブロックの後に条件を評価します。FOR と同様、コード・ブロック内に QUIT コマンドを記述してループを終了します。

2 つのコマンドの構文は、以下のとおりです。

```
DO {code} WHILE condition
WHILE condition {code}
```

以下の例は、ユーザが指定した値までフィボナッチの数列で値を 2 回表示します (最初は DO WHILE 次はWHILE を使用)。

ObjectScript

```
fibonacci() PUBLIC { // generate Fibonacci sequences
    READ !, "Generate Fibonacci sequence up to where? ", upto
    SET t1 = 1, t2 = 1, fib = 1
    WRITE !
    DO {
        WRITE fib,"  "  set fib = t1 + t2, t1 = t2, t2 = fib
    }
    WHILE ( fib '> upto )

    SET t1 = 1, t2 = 1, fib = 1
    WRITE !
    WHILE ( fib '> upto ) {
        WRITE fib,"  "
        SET fib = t1 + t2, t1 = t2, t2 = fib
    }
 }
```

WHILE、DO WHILE、FOR の明確な違いは、WHILE はループの実行前、DO WHILE はループの実行後に制御式の値をテストする必要があり、一方 FOR はループ内ならどこででもテストできる点です。つまり、コード・ブロックに 2 つの部分があり、式の評価によって 2 番目を実行する場合、FOR 文が一番適切です。それ以外の場合、式をコード・ブロックの前か後のいずれに評価するかによって選択します。

詳細は、"ObjectScript リファレンス" の "WHILE" コマンドおよび "DO WHILE" コマンドを参照してください。

7.9 入出力コマンド

ObjectScript の入出力コマンドは、InterSystems IRIS の内部あるいは外部でデータを取得する基本的な機能を提供します。以下のとおりです。

- Write コマンド
- READ
- OPEN、USE、および CLOSE

7.9.1 表示 (書き込み) コマンド

ObjectScript は、現在の出力デバイスにリテラルと変数値を表示 (書き込み) する、以下の 4 つのコマンドをサポートしています。

- WRITE コマンド

- ZWRITE コマンド
- ZZDUMP コマンド
- ZZWRITE コマンド

7.9.1.1 引数なしの表示コマンド

- 引数なしの WRITE は、定義済みの各ローカル変数の名前と値を表示します (行ごとに 1 つの変数を表示します)。パブリック変数とプライベート変数の両方がリスト表示されます。グローバル変数、プロセス・プライベート・グローバルまたは特殊変数はリスト表示されません。変数は照合シーケンス順でリスト表示されます。添え字ツリー順で添え字付き変数をリストします。

  すべてのデータ値は、二重引用符文字で区切られ、引用符で囲まれた文字列として表示されます。ただし、キャノニック形式の数とオブジェクト参照を除きます。オブジェクト参照 (OREF) 値が割り当てられた変数は、`variable=<OBJECT REFERENCE>[oref]` として表示されます。%List 形式の値またはビット文字列の値は、引用符で囲まれた文字列として、それらの値がエンコードされた形式で表示されます。このようなエンコードされた形式には非表示文字が含まれている可能性があるため、%List やビット文字列の表示は空の文字列になることがあります。

  WRITE は、特定の非表示文字を表示しません。そうした非表示文字を表現するための、プレースホルダやスペースは表示されません。WRITE は、制御文字を実行します (改行やバックスペースなど)。
- 引数なしの ZWRITE は、引数なしの WRITE と同様に機能します。
- 引数なしの ZZDUMP は、無効なコマンドであり、<SYNTAX> エラーを生成します。
- 引数なしの ZZWRITE は、空の文字列を返す空命令です。

7.9.1.2 引数ありの表示コマンド

以下の表に、4 つのコマンドの引数付きの形式の機能をリストします。4 つのすべてのコマンドは、1 つの引数または引数のコンマ区切りリストを取ります。4 つのすべてのコマンドは、引数として、ローカル変数、グローバル変数またはプロセス・プライベート変数、リテラル、式、または特殊変数を取ります。

以下の表には、%Library.Utility.FormatString() メソッドの既定の戻り値も示されています。FormatString() メソッドは、ZZWRITE とほとんど同じですが、戻り値の一部として `%val=` をリスト表示しないことと、オブジェクト参照 (OREF) 識別子のみを返す点が異なります。FormatString() を使用すると、ZWRITE/ZZWRITE 形式の戻り値に変数を設定できるようになります。

テーブル 7-1: 表示形式

	WRITE	ZWRITE	ZZDUMP	ZZWRITE	FormatString()
各値を別個の行に表示	なし	あり	あり (1 行に 16 文字)	あり	1 つの入力値のみ
識別された変数名	なし	あり	なし	%val= で表される	なし
未定義の変数を <UNDEFINED> エラーと表示	あり	なし (スキップされ、変数名は返されない)	あり	あり	あり

4 つのすべてのコマンドは、式を評価して、キャノニック形式で数値を返します。

テーブル 7-2: 値の表示方法

	WRITE	ZWRITE	ZZDUMP	ZZWRITE	FormatString()
16 進表現	なし	なし	あり	なし	なし
文字列を囲む引用符（数値と区別するため）	なし	あり	なし	あり（文字列リテラルは %val="value" として返される）	あり
添え字ノードの表示	なし	あり	なし	なし	なし
別のネームスペースのグローバル変数（拡張グローバル参照）の表示	あり	あり（拡張グローバル参照構文が示される）	あり	あり	あり
非表示文字の表示	なし、表示されない（制御文字は実行される）	あり、$c(n) として表示される	あり、16 進数として表示される	あり、$c(n) として表示される	あり、$c(n) として表示される
リスト値の形式	エンコードされた文字列	$lb(val) 形式	エンコードされた文字列	$lb(val) 形式	$lb(val) 形式
%Status 形式	エンコードされたリストを含んでいる文字列	$lb(val) 形式のリストを含んでいる文字列、エラーとメッセージを明示する /*...*/ コメントが付加される。	エンコードされたリストを含んでいる文字列	$lb(val) 形式のリストを含んでいる文字列、エラーとメッセージを明示する /*...*/ コメントが付加される。	$lb(val) 形式のリストを含んでいる文字列、エラーとメッセージを明示する /*...*/ コメントが付加される（既定の場合）。
ビット文字列形式	エンコードされた文字列	$zwc 形式、1 つのビットをリストする /* $bit() */ コメントが付加される。例: %[illegible]	エンコードされた文字列	$zwc 形式、1 つのビットをリストする /* $bit() */ コメントが付加される。例: %[illegible]	$zwc 形式、1 つのビットをリストする /* $bit() */ コメントが付加される（既定の場合）。例: %[illegible]
オブジェクト参照（OREF）形式	OREF のみ	<OBJECT REFERENCE>[oref] 形式の OREF。一般情報や属性値などの詳細がリストされる。すべてのサブノードがリストされる	OREF のみ	<OBJECT REFERENCE>[oref] 形式の OREF。一般情報や属性値などの詳細がリストされる。	OREF のみ（引用符で囲まれた文字列形式）

JSON の動的配列と JSON のダイナミック・オブジェクトは、該当するコマンドのすべてで OREF 値として返されます。JSON のコンテンツを返すには、以下の例に示すように、%ToJSON() を使用する必要があります。

```
SET jobj={"name":"Fred","city":"Bedrock"}
WRITE "JSON object reference:",!
ZWRITE jobj
WRITE !!,"JSON object value:",!
ZWRITE jobj.%ToJSON()
```

詳細は、"ObjectScript リファレンス" の "WRITE"、"ZWRITE"、"ZZDUMP"、および "ZZWRITE" の各コマンドを参照してください。

7.9.2 READ

READ コマンドを使用すると、現在の入力装置からエンド・ユーザが入力した内容を受け取り、格納することができます。READ コマンドは、以下の引数を持つことができます。

ObjectScript

```
READ format, string, variable
```

format は、ユーザの入力エリアが画面に表示される場所を制御し、string は入力プロンプトの前に画面に表示され、variable は入力データを格納します。

以下の形式コードを使用して、ユーザ入力エリアを制御します。

形式コード	結果
!	新しい行を開始します。
#	新しいページを開始します。ターミナルでは、現在の画面をクリアして、新規画面の一番上から開始します。
?n	n 番目の列に移動します。n は正の整数です。

詳細は、"ObjectScript リファレンス" の "READ" コマンドを参照してください。

7.9.3 OPEN、USE、および CLOSE

さらに高度なデバイス処理を実行するために、InterSystems IRIS は多彩なオプションを提供します。つまり、OPEN コマンドでデバイスを開き、USE コマンドで現在のデバイスを指定して、CLOSE コマンドで実行中のデバイスを閉じることができます。この処理の詳細は、"入出力デバイス・ガイド" で説明されています。

詳細は、"ObjectScript リファレンス" の "OPEN" コマンド、"USE" コマンド、および "CLOSE" コマンドを参照してください。

8

呼び出し可能なユーザ定義コードモジュール

このトピックでは、InterSystems IRIS® で ObjectScript コードのユーザ定義モジュール (メソッド、関数、プロシージャ、ルーチン、またはサブルーチン) を作成する方法および呼び出す方法について説明します。最も一般的なコード形式はメソッドであり、クラスに定義できます。このトピックでは、メソッドについて具体的には説明しませんが、既定ではメソッドはプロシージャです。このため、プロシージャに関する内容はすべてメソッドにも適用されます。

他の言語と同様に、ObjectScript は直接呼び出すことができる名前付きのコード・ブロックを生成できます。このようなブロックをプロシージャともいいます。ObjectScript 用語で厳密に言うと、プロシージャであるコード・ブロックは、特定の構文と構造を持ちます。

プロシージャ定義の構文は、以下のようになります。

ObjectScript

```
ProcedureName(Parameters) [PublicVariables]
 {
    /* code goes here */
    RETURN ReturnValue
 }
```

ここで ProcedureName としているプロシージャの要素は以下のとおりです。

- パラメータ (0 か 1) ― あらゆるタイプが可能です。一般的な ObjectScript 構文と同様、プロシージャの定義時にタイプを宣言する必要はありません。既定で、パラメータは (参照渡しではなく) 値によって渡されます。指定がない限り、その範囲はプロシージャに対しローカルです。パラメータの一般的な詳細は、"プロシージャ・パラメータ" を参照してください。
- パブリック変数への参照 (0 かそれ以上) ― あらゆるタイプが可能です。プロシージャは、変数の値を参照することも、設定することもできます。パブリック変数の詳細は、"プロシージャ変数" セクションを参照してください。
- プロシージャの Public 宣言 (オプション) ― 既定では、プロシージャはプライベートであるため、同じルーチン内からのみ、そのプロシージャを呼び出すことができます (ObjectScript 用語で、ルーチンとは 2 つ以上のプロシージャ、あるいはユーザ定義のコード・ブロックを含むファイルのことです)。また、プロシージャ名の後に PUBLIC キーワードを使用して、パブリック・プロシージャを生成できます。パブリック・プロシージャは他のルーチンやメソッドからも呼び出せます。パブリック・プロシージャとプライベート・プロシージャの詳細は、"パブリック・プロシージャとプライベート・プロシージャ" を参照してください。
- コード ― プロシージャ内のコードは、ObjectScript で使用できるすべての機能を備えています。プロシージャ・コードには、Java も使用できます。コードは中括弧で区切られ、プロシージャ・ブロックとも呼ばれます。
- 返り値 (オプション) ― プロシージャが返す値です。標準の ObjectScript 式である必要があります。プロシージャ内のフロー制御では、計算式の値、RETURN 文、またはその両方を使用して、さまざまな返り値を指定できます。

注釈　通常は、サブルーチンやユーザ定義関数を記述するよりも、プロシージャを記述する方が望ましいです。プロシージャ・パラメータは、プロシージャでのスコープで自動的にローカルになります。これらのパラメータはプロシージャにとってプライベートであり、シンボル・テーブルとやり取りしないので、他の値を上書きしないようにする NEW コマンドは不要です。また、パブリック変数を明示的に宣言することで、例えば銀行全体の金利といったアプリケーション内のグローバル変数を参照できます。あるいは、他のアプリケーションに有効なプロシージャ内で、変数の値を生成、設定することもできます。

プロシージャは、ObjectScript ルーチンの一種です。

InterSystems IRIS は、多くのシステム関数を備えています。内部関数とも呼ばれるこの関数の詳細は、"ObjectScript ランゲージ・リファレンス" で説明しています。システム関数の呼び出しは、接頭語 $ で識別されます。

8.1 プロシージャ、ルーチン、サブルーチン、関数、メソッドの概要

このトピックでは、プロシージャを使用して、独自のコードを実装する方法について説明します。ユーザ定義の機能実装では、この方法をお勧めします。InterSystems のドキュメントでは、プロシージャ、ルーチン、サブルーチン、関数、およびメソッドについて説明しています。すべての項目が機能を共有していますが、それぞれに独自の特徴もあります。

名前付きのユーザ定義のコード・ブロックを記述するのに、最も柔軟で強力な推奨される形式は、プロシージャです。以下はプロシージャの特徴です。

- プライベートあるいはパブリックです。
- ゼロ個以上のパラメータを受け取ることができます。
- その中で生成された変数を、ローカルの範囲として自動的に維持します。
- 範囲外の変数を参照し変更できます。
- あらゆるタイプの値を返す、あるいは値を返しません。

以下はプロシージャとの比較です。

- サブルーチンは常にパブリックで、値を返しません。
- 関数は常にパブリックで、ローカル変数を明示的に宣言する必要があります (そうしなければ、外部変数を上書きします)。また、必ず返り値があります。
- 既定で、メソッドはクラス定義の一部として指定されるプロシージャで、1 つ以上のオブジェクトあるいはクラスで呼び出すことができます。メソッドを関数として明示的に宣言すると、付随するすべての特徴も関数に属するため、これはお勧めできません。
- ルーチンは、ObjectScript プログラムです。1 つ以上のプロシージャ、サブルーチン、関数、あるいはこれらの組み合わせになります。

注釈　ObjectScript は、マクロ機能を通して、ユーザ定義コードに関連する形式もサポートしています。

8.1.1 ルーチン

ルーチンは、ユーザ記述コードの呼び出し可能なブロックで、ObjectScript プログラムです。ルーチンは汎用的な処理を実行します。ルーチン名は、保存時に選択する .MAC ファイル名から付けられます。ルーチンが値を返すかどうかによって、以下の構文の 1 つあるいは両方を使用して、ルーチンを呼び出すことができます。

ObjectScript

```
DO ^RoutineName
SET x = $$^RoutineName
```

ルーチンはネームスペース内で定義されます。拡張ルーチン参照を使用し、以下のようにして現在のネームスペース以外のネームスペースで定義されたユーザ定義ルーチンを実行できます。

ObjectScript

```
DO ^|"USER"|RoutineName
```

一般的に、ルーチンは、サブルーチン、メソッド、プロシージャのコンテナとして機能します。

ルーチンは、コード・ブロックの最初にあるラベル (タグともいいます) で識別されます。このラベルがルーチン名です。(通常) ラベルの後ろに続く括弧内には、呼び出し元のプログラムからルーチンに渡すパラメータ・リストが記述されます。

ルーチンをファイルに保存する場合、ファイル名には、アンダースコア (_)、ハイフン (-)、セミコロン (;) を使用できません。このような文字を含む名前は無効です。

8.1.2 サブルーチン

サブルーチンは、ルーチン内で名前の付いたコード・ブロックです。通常、サブルーチンはラベルから始まり、QUIT 文で終了します。パラメータを受け取ることはできますが、値は返しません。サブルーチンを呼び出すには、以下の構文を使用します。

```
DO Subroutine^Routine
```

Subroutine は、Routine ファイル (Routine.MAC) 内のコード・ブロックです。

以下は、サブルーチンの形式です。

ObjectScript

```
Label(parameters) // comment
 // code
 QUIT // note that QUIT has no arguments
```

サブルーチンの詳細は、従来のコードを説明している以下の "サブルーチン" セクションを参照してください。

{ } 括弧でコードと QUIT 文を囲む場合、サブルーチンはプロシージャとなるため、プロシージャとして処理されます。そのような場合、QUIT 文が重複するので省略してもかまいません。例えば、以下の 2 つのサブルーチンの定義は同等です。

ObjectScript

```
Label(parameters) PUBLIC {
 // code
 QUIT
}
```

および、

ObjectScript

```
Label(parameters) PUBLIC {
 // code
}
```

8.1.3 関数

InterSystems IRIS は、"ObjectScript ランゲージ・リファレンス" で説明しているように、多くのシステム関数 (内部関数とも呼ばれる) を備えています。このセクションでは、ユーザ定義 (extrinsic) 関数について説明します。

関数は、ルーチン内で名前の付いたコード・ブロックです。通常、関数はラベルから始まり、RETURN 文で終了します。パラメータを受け取り、値も返します。関数の呼び出しにおいて、以下の 2 つが有効な形式の構文となります。

```
SET rval=$$Function()  /* returning a value */
DO Function^Routine   /* ignoring the return value */
```

Function は、Routine ファイル (Routine.MAC) 内のコード・ブロックです。両方の構文形式で、拡張ルーチン参照を使用し、異なるネームスペースにある関数を実行できます。

以下は、関数の形式です。

ObjectScript

```
Label(parameters)
 // code
 RETURN ReturnValue
```

{} 括弧でコードと RETURN 文を囲むと、関数はプロシージャとなるため、プロシージャとして処理されます。プロシージャは既定によりプライベートであるため、以下のように PUBLIC キーワードを指定することが必要になる場合もあります。

ObjectScript

```
Label(parameters) PUBLIC {
 // code
 RETURN ReturnValue }
```

以下の例では、簡単な関数 (MyFunc) を指定したうえで、それを呼び出し、2 つのパラメータを渡して、さらに返される値を受け取ります。

ObjectScript

```
Main ;
   TRY {
     KILL x
     SET x=$$MyFunc(7,10)
        WRITE "returned value is ",x,!
        RETURN
   }
      CATCH { WRITE $ZERROR,!
   }
MyFunc(a,b)
   SET c=a+b
   RETURN c
```

関数を呼び出すコードは返される値を無視できますが、関数の RETURN コマンドでは返される値を指定することが必要となります。引数なしの RETURN にて関数の終了を試みると、<COMMAND> エラーが生成されます。<COMMAND> エラーによって関数を呼び出したコールの場所が指定され、呼び出された関数内の引数なし RETURN コマンドのオフセット位置を指定するメッセージが続きます。詳細は "$ZERROR" を参照してください。

関数の詳細は、従来のコードを説明している以下の "関数" セクションを参照してください。

8.2 プロシージャの定義

他のプログラミング言語と同様、プロシージャは特定のタスクを実行する一連の ObjectScript コマンド (大規模なルーチン) です。If 文の構造のように、プロシージャのコードは中括弧 ({}) で囲まれています。

プロシージャは、パブリックかプライベートのいずれかで変数を定義できます。例えば、以下は MyProc というプロシージャの例です。

ObjectScript

```
MyProc(x,y) [a,b] PUBLIC {
 Write "x + y = ", x + y
}
```

これは、2 つのパラメータ x と y を取る MyProc というパブリック・プロシージャを定義しています。2 つのパブリック変数 a と b も定義しています。このプロシージャで使用される他のすべての変数 (この場合、x と y) はプライベート変数です。

既定では、プロシージャはプライベートであるため、同じルーチン内からのみ呼び出すことができますまた、プロシージャ名の後に PUBLIC キーワードを使用して、パブリック・プロシージャを生成できます。パブリック・プロシージャは他のルーチンからも呼び出せます。

プロシージャでは、定義済みのパラメータは必要ありません。パラメータでプロシージャを生成する場合、ラベルの直後に括弧で囲んだ変数リストを置きます。

8.2.1 プロシージャの呼び出し

プロシージャを呼び出すには、DO コマンドを発行してプロシージャを指定するか、$$ 構文を使用して関数として呼び出します。プロシージャが他のプログラムから実行できるかどうか (パブリック)、あるいは存在するプログラム内でのみ実行できるかどうか (プライベート) を制御できます。DO コマンドで呼び出す場合、プロシージャは値を返しません。関数呼び出しとして呼び出す場合は、値を返します。$$ 形式は最も機能的であり、一般的に好まれる形式です。

8.2.1.1 $$ 接頭語の使用法

式を使用できるコンテキスト内で、ユーザ定義関数を呼び出すことができます。以下の形式で、ユーザ定義関数を呼び出します。

```
$$name([param[ ,...]])
```

以下を示します。

- name 関数の名前を指定します。関数の定義場所によって、以下のように名前を指定できます。
  - label は、現在のルーチンにある行ラベルです。
  - label^routine は、ディスクにある名前付きルーチンの行ラベルです。
  - ^routine は、ディスクに保存されているルーチンです。ルーチンは、実行する関数のコードのみを含んでいる必要があります。
- param は関数に渡される値を指定します。提供されるパラメータは、実パラメータ・リストと呼ばれます。これらは、その関数に関して定義された仮パラメータ・リストと一致する必要があります。例えば、関数コードには 2 つのパラメータがあり、1 番目は数値、2 番目は文字リテラルとします。この場合、1 番目のパラメータに文字リテラル値、2 番目に数値を指定すると、関数は不正確な値を返すか、あるいはエラーを生成する可能性があります。仮パラメータ・リストのパラメータでは、関数から NEW コマンドが必ず呼び出されます。詳細は、NEW コマンドの説明を参照してください。パラメータは、値または参照によって渡されます。詳細は、"パラメータ渡し" を参照してください。関数の仮

パラメータ・リストに記載されている数より少数のパラメータを関数に渡すと、パラメータの既定値が使用されます（定義されている場合）。既定値がない場合、パラメータは未定義のままとなります。

8.2.1.2 DO コマンドの使用法

DO コマンドを使用して、ユーザ定義関数を呼び出すことができます（DO コマンドを使用して、システム関数を呼び出すことはできません）。DO コマンドで呼び出した関数は、値を返しません。つまり、関数は必ず返り値を生成しますが、DO コマンドはこの返り値を無視します。したがって、ユーザ定義関数を呼び出す DO コマンドの用途は大幅に制限されます。

DO コマンドを使用してユーザ定義関数を呼び出すには、以下の構文でコマンドを発行します。

DO label(param[,...])

DO コマンドは、label という関数を呼び出し、param で指定されたパラメータ（存在する場合）を関数に渡します。$$ 接頭語を使用しないことと、パラメータの括弧が必須であることに注意してください。label と param を指定する際には、$$ 接頭語を使用するユーザ定義関数を呼び出すときと同じ規則が適用されます。

関数は、常に値を返す必要があります。しかし、DO コマンドで関数を呼び出す場合、返された値は呼び出し元のプログラムで無視されます。

8.2.2 プロシージャの構文

以下はプロシージャの構文です。

```
label([param[=default]][,...]) [[pubvar[,...]]] [access]
 {
  code
 }
```

以下の構文で呼び出します。

```
DO label([param[,...]])
```

あるいは以下のようになります。

```
 command $$label([param][,...])
```

以下はその説明です。

引数	説明
label	プロシージャ名です。標準ラベルです。必ず 1 列目から始まります。label に続くパラメータの括弧は必須です。
param	プロシージャに必要な各パラメータの変数です。必要なパラメータは、仮パラメータ・リスト と呼ばれます。パラメータ自体はオプションですが（何もない、1 つある、あるいは 1 つ以上の param がある）、括弧は必須です。複数の param 値がある場合、コンマによって区切られます。パラメータは、値または参照で仮パラメータ・リストに渡すことができます。ルーチンであるプロシージャにはパラメータのタイプ情報が記述されていませんが、メソッドであるプロシージャにはこの情報が記述されています。仮パラメータの最大数は 255 個です。

引数	説明
default	直前の param に対するオプションの既定値です。各パラメータに対して既定値を提供しても、省略してもかまいません。仮パラメータに対応する実パラメータが指定されていない場合、または実パラメータが参照渡しされたときに当該のローカル変数に値がない場合は、既定値が適用されます。この既定値はリテラル、つまり引用符で囲まれた数字あるいは文字列です。null 文字列 () も既定値として指定できます。null 文字列は変数を定義するため、これは既定値を指定しないこととは異なります。一方でパラメータ値が未指定の場合や既定値がない場合、変数は未定義となります。リテラル以外の既定値を指定すると、InterSystems IRIS は <PARAMETER> エラーを発行します。
pubvar	パブリック変数です。プロシージャによって使用され、他のルーチンとプロシージャで使用できる、パブリック変数の任意のリストです。これは、このプロシージャ内で定義され、他のルーチンでも使用可能な変数、あるいは他のルーチンで定義され、このプロシージャでも使用可能な変数のリストです。pubvar を指定する場合、それを角括弧で囲みます。pubvar を指定しない場合、角括弧は省略できます。複数の pubvar 値がある場合、コンマによって区切ります。Public として宣言されていない変数は、すべてプライベート変数です。プライベート変数は、プロシージャの現在の呼び出しに対してのみ有効です。その変数は、プロシージャの呼び出し時には未定義です。また、プロシージャが終了する際に破棄されます。プロシージャがそのプロシージャ外のコードを呼び出すと、プライベート変数は保存されますが、制御がプロシージャに戻るまでは使用できません。また、すべての % 変数は、ここにリストされているかどうかにかかわらず、常にパブリックです。パブリック変数のリストは、このルーチンに対して指定された 1 つ以上の param を含むことができます。
access	プロシージャが Public か Private を宣言する任意のキーワードです。利用できる値は 2 つあります。PUBLIC キーワードは、このプロシージャがどのルーチンからでも呼び出しできることを宣言します。PRIVATE キーワードは、このプロシージャが定義されたルーチン内でのみ呼び出されることを宣言します。既定は PRIVATE です。
code	中括弧で囲まれたコード・ブロックです。左中括弧 ({) は、その後ろに続く先頭の文字から少なくとも 1 つのスペースを置く、あるいは 1 行の改行が必要です。同じ行の右中括弧 (}) の後ろにはコードを記述できず、ブランクあるいはコメントのみ置くことができます。通常、右中括弧は 1 列目に記述します。このコード・ブロックは、ラベルによってのみ入力されます。

コマンドと引数の間には改行を挿入できません。

それぞれのプロシージャはルーチンの一部として実装され、各ルーチンは複数のプロシージャを持つことができます。

標準的な ObjectScript 構文の他に、ルーチンを管理する特殊な規則があります。ルーチン行はオプションとして、ObjectScript コードの最初にラベル (タグとも呼びます) を、最後にコメントを置くことができます。

インターシステムズでは、ルーチンの 1 行目にルーチン名を表すラベル、その後にタブまたはスペースを置いた後、ルーチンの目的を簡単に説明する短いコメントを入れることをお勧めします。行にラベルを置く場合、タブやスペースを含む残りの行からラベルを離す必要があります。つまり、スタジオを使用してルーチンに行を追加し、ラベルとタブ/スペースに続けて ObjectScript コードを入力するか、ラベルを置かずにタブかスペースに続けて ObjectScript を入力します。したがって、いずれの場合でも、それぞれの行には、最初のコマンドの前にタブかスペースが必要です。

1 行コメントを示す場合、二重スラッシュ (//) かセミコロン (;) のいずれかを使用します。コメントがコードの後ろに続く場合、スラッシュやセミコロンの前にスペースが必要です。その行にコメントしか記述しない場合、スラッシュやセミコロンの前に、タブかスペースが必要です。当然のことながら、1 行コメントは改行できません。複数行コメントの場合が、コメントの先頭に /* を、末尾には */ を付けます。

8.2.3 プロシージャ変数

プロシージャとメソッドは両方共、プライベート変数とパブリック変数をサポートします。以下の文はすべてプロシージャとメソッドに同等に適用されます。

プロシージャ内で使用する変数は、自動的に プライベート になります。したがって、これらは宣言する必要がなく、NEW コマンドは不要です。このプロシージャが呼び出すプロシージャの変数を共有したい場合、変数をパラメータとして他のプロシージャに渡します。

パブリック変数の宣言も可能です。パブリック変数は、このプロシージャまたはメソッドの呼び出し元と呼び出し先のすべてのプロシージャおよびメソッドで使用できます。アプリケーションで環境変数として動作するよう、比較的少数のパブリック変数をこの方法で定義します。パブリック変数を定義するには、プロシージャ名とパラメータの後に角括弧内 ([ ]) で変数をリストします。

以下の例では、宣言された 2 つのパブリック変数 [a, b] と 2 つのプライベート変数 (c、d) が含まれるプロシージャが定義されます。

ObjectScript

```
publicvarsexample
    ; examples of public variables
    ;
    DO proc1()   ; call a procedure
    QUIT    ; end of the main routine
    ;
proc1() [a, b]
    ; a private procedure
    ; "c" and "d" are private variables
    {
    WRITE !, "setting a"  SET a = 1
    WRITE !, "setting b"  SET b = 2
    WRITE !, "setting c"  SET c = 3
    SET d = a + b + c
    WRITE !, "The sum is: ", d
    }
```

Terminal

```
USER>WRITE

USER>DO ^publicvarsexample

setting a
setting b
setting c
The sum is: 6
USER>WRITE

a=1
b=2
USER>
```

8.2.3.1 パブリック変数とプライベート変数

プロシージャ内では、ローカル変数は パブリック あるいは プライベート のいずれででも定義できます。パブリック・リスト (pubvar) は、プロシージャ内のどの変数参照がパブリック変数のセットに追加されるかを宣言します。プロシージャ内の他の変数参照はすべて、プロシージャの現在の呼び出しによってのみ参照される、プライベート・セットに追加されます。

プライベート変数は、プロシージャの呼び出し時には未定義です。また、プロシージャが終了する際に破棄されます。

プロシージャ内のコードがプロシージャ外のコードを呼び出す場合、プライベート変数は、プロシージャに戻ったときに復元されます。呼び出されたプロシージャやルーチンは、パブリック変数にアクセスできます (それ自身のプライベート変数も同様に使用できます)。したがって、pubvar はこのプロシージャで参照されるパブリック変数と、プロシージャが呼び出すルーチンによって参照できこのプロシージャで使用する変数の両方を指定します。

パブリック・リストが空の場合、すべての変数はプライベートです。この場合、角括弧は任意となります。

名前の先頭に % 文字が付く変数は、通常、システムや特定用途のために使用する変数です。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。ユーザ・コードでは、%z または %Z で始まる % 変数のみを使用してください。すべての % 変数は、暗黙的にパブリックです。これらの変数は、(ドキュメント用に) パブリック・リストに記載することもできますが、必須ではありません。

8.2.3.2 プライベート変数と NEW コマンドで生成される変数

プライベート変数は、NEW で生成される変数とは異なることに注意してください。プロシージャの変数を、他のプロシージャやサブルーチンで使用したい場合、その変数は必ず Public に宣言し、パブリック・リストに記述する必要があります。このプロシージャによってパブリック変数が導入されていれば、それに対して NEW を実行する意味があります。この場合、パブリック変数は、プロシージャが終了するときに自動的に破棄されます。しかし、パブリック変数が保持していた以前の値は保護されます。以下はコードの例です。

ObjectScript

```
MyProc(x,y)[name]{
 NEW name
 SET name="John"
 DO xyz^abc
}
```

name 変数はパブリックであるため、このコードによって、この変数の値 John を abc ルーチンの xyz プロシージャで参照できます。name に対して NEW コマンドを実行すると、MyProc プロシージャを呼び出したときに既に存在しているすべてのパブリック変数 name が保護されます。

NEW コマンドは、プライベート変数ではなくパブリック変数にのみ作用します。プロシージャ内で x がパブリック・リストになく、% 変数でない場合、NEW x または NEW (x) を指定すると違反になります。

8.2.3.3 仮リスト・パラメータをパブリックにする

プロシージャに仮リスト・パラメータ (例えば MyProc(x,y) の x あるいは y) が存在し、それを他のプロシージャ (あるいはこのプロシージャ外のサブルーチン) で使用する場合は、それをパブリック・リストに置く必要があります。

以下はコードの例です。

ObjectScript

```
MyProc(x,y)[x] {
 DO abc^rou
 }
```

y ではなく x の値が、abc^rou ルーチンで使用可能となります。

8.2.4 パブリック・プロシージャとプライベート・プロシージャ

プロシージャは、パブリックまたはプライベートです。プライベート・プロシージャは、プロシージャが定義されたルーチン内でのみ呼び出すことができます。一方パブリック・プロシージャは、どのルーチンからも呼び出すことができます。PUBLIC キーワードと PRIVATE キーワードが省略されている場合、既定はプライベートです。

以下はその例です。

ObjectScript

```
MyProc(x,y) PUBLIC { }
```

上記はパブリック・プロシージャを定義しています。

ObjectScript

```
MyProc(x,y) PRIVATE { }
```

あるいは以下のコードを入力します。

ObjectScript

```
MyProc(x,y) { }
```

上記 2 つのコードは、プライベート・プロシージャを定義しています。

8.3 パラメータ渡し

プロシージャの重要な特徴は、パラメータ渡しに対するサポートです。これは、プロシージャに値または変数をパラメータとして渡すための機能です。当然のことながら、パラメータ渡しは必須ではありません。例えば、パラメータ渡しとしていないプロシージャを使用して、乱数の生成や、既定以外の形式でシステム日付を返す操作などが可能です。それでも、多くの場合、プロシージャではパラメータ渡しを使用します。

パラメータ渡しを設定するには、以下を指定します。

- プロシージャ呼び出し時の実パラメータ・リスト
- プロシージャ定義時の仮パラメータ・リスト

InterSystems IRIS がユーザ定義プロシージャを実行するとき、実パラメータ・リストのパラメータが、それに対応する仮リストのパラメータに位置を基準としてマップされます。したがって、実リストの先頭にあるパラメータの値は仮リストの先頭にある変数に配置され、2 番目の値は 2 番目の変数に配置され、以降の値も同様に配置されます。これらのパラメータの照合は、名前ではなく、パラメータの位置で行われます。したがって、実パラメータに使用する変数と仮パラメータに対して使用する変数は、同じ名前を持つ必要はありません (また、通常持つべきではありません)。プロシージャでは、渡された値に該当する変数を仮リストの中で参照することにより、渡された値にアクセスします。

実パラメータ・リストと仮パラメータ・リストの間で、収録されているパラメータの数が異なっていることがあります。

- 実パラメータ・リストにあるパラメータが、仮パラメータ・リストにあるパラメータより少ない場合、仮パラメータ・リストと一致しない要素は未定義となります。以下の例のように、未定義の仮パラメータに使用するデフォルト値を指定できます。

  ObjectScript

  ```
  Main
     /* Passes 2 parameters to a procedure that takes 3 parameters */
     SET a="apple",b="banana",c="carrot",d="dill"
     DO ListGroceries(a,b)
     WRITE !,"all done"
  ListGroceries(x="?",y="?",z="?") {
     WRITE x," ",y," ",z,!   }
  ```

- 実パラメータ・リストにあるパラメータが、仮パラメータ・リストにあるパラメータより多い場合は、以下の例のように、<PARAMETER> エラーが発生します。

  ObjectScript

  ```
  Main
     /* Passes 4 parameters to a procedure that takes 3 parameters.
        This results in a <PARAMETER> error  */
     SET a="apple",b="banana",c="carrot",d="dill"
     DO ListGroceries(a,b,c,d)
     WRITE !,"all done"
  ListGroceries(x="?",y="?",z="?") {
     WRITE x," ",y," ",z,!   }
  ```

仮リストにある変数が実リストにあるパラメータよりも多く、既定値が各パラメータに指定されていない場合、余った変数は未定義のままとなります。プロシージャのコードに適切な If テストを記述し、プロシージャの各参照で、使用可能な値が

得られることを確認する必要があります。実パラメータ数と仮パラメータ数の一致を簡素化するには、可変個数のパラメータを指定します。

実パラメータの最大数は 254 個です。

ユーザ定義プロシージャにパラメータを渡す場合は、値渡しまたは参照渡しが可能です。同じプロシージャの呼び出しの中で、値渡しと参照渡しを混用できます。

- プロシージャには、パラメータを値で渡すことも参照で渡すこともできます。
- サブルーチンには、パラメータを値で渡すことも参照で渡すこともできます。
- ユーザ定義関数には、パラメータを値で渡すことも参照で渡すこともできます。
- システム関数には、パラメータを値でのみ渡すことができます。

8.3.1 値渡し

値渡しをする場合は、実パラメータ・リストでリテラル値、式、またはローカル変数 (添え字付きでも添え字なしでも可) を指定します。式を指定した場合、InterSystems IRIS は最初にその式を評価し、次に結果の値を渡します。ローカル変数を指定した場合、InterSystems IRIS はその変数の現在の値を渡します。指定したすべての変数名が必ず存在し、値が設定されている必要があります。

プロシージャの仮パラメータ・リストには、添え字なしのローカル変数名を記述します。このリストで明示的な添え字付き変数を指定することはできません。一方、可変個数のパラメータを指定すると、添え字付き変数が暗黙的に作成されます。

InterSystems IRIS は、プロシージャ内で使用される非パブリック変数をすべて暗黙に生成し宣言します。したがって、呼び出し元コードで同じ名前を持つ既存の変数は上書きされません。これらの変数の既存の値 (存在する場合) をプログラム・スタックに置きます。QUIT コマンドまたは RETURN コマンドを呼び出すとき、InterSystems IRIS は各仮変数に対して暗黙の KILL コマンドを実行し、スタックから前の値を復元します。

以下の例で、SET コマンドは 3 つの異なる形式を使用して、参照先の Cube プロシージャに同じ値を渡します。

ObjectScript

```
 DO Start()
 WRITE "all done"
Start() PUBLIC {
 SET var1=6
 SET a=$$Cube(6)
 SET b=$$Cube(2*3)
 SET c=$$Cube(var1)
 WRITE !,"a: ",a," b: ",b," c: ",c,!
 RETURN 1
 }
Cube(num) PUBLIC {
 SET result = num*num*num
 RETURN result
}
```

8.3.2 参照渡し

参照渡しの場合、以下の形式を使用して、実パラメータ・リストのローカル変数名あるいは添え字なしの配列名を指定します。

```
.name
```

参照渡しでは、プロシージャでまだ参照していない段階であれば、指定された変数名または配列名が実際に存在している必要はありません。通常、配列は参照渡しのみを使用します。添え字付き変数は参照渡しできません。

- 実パラメータ・リスト：実パラメータ・リストのローカルな変数名または配列名の前にはピリオドを記述する必要があります。これは、変数が値ではなく、参照で渡されることを指定するものです。
- 仮パラメータ・リスト：仮パラメータ・リストでは特殊な構文を必要とせずに、参照渡しで変数を受け取ります。仮パラメータ・リストでは、ピリオドを接頭語として使用することはできません。ただし、仮パラメータ・リストの変数名の前にアンパサンド (&) を接頭語として使用することはできます。規約により、この & 接頭語の使用は、変数が参照渡しされることを示します。& 接頭語はオプションであり、処理を実行するものではありません。ソース・コードを読みやすく、かつ保守しやすくするために役立つ規約です。

"引数の渡し方の指示" で説明しているように、メソッド定義では参照渡しを指定できます。

参照渡しでは、実リストの各変数および配列名が、関数の仮リストの対応する変数名に結合されます。参照渡しは、参照するルーチンと関数の間を双方向で通信する効果的なメカニズムを提供します。関数が仮リストにある変数を変更すると、対応する実リスト上の参照渡しの変数も変更されます。これは、KILL コマンドにも適用されます。仮リストにある参照渡しの変数が関数によって削除されると、対応する実リスト上の変数も削除されます。

実リストで指定した変数あるいは配列名が存在しない場合、関数参照により未定義として処理されます。関数が対応する仮リストの値を変数に代入すると、実の変数あるいは配列もこの値で定義されます。

以下の例では、参照渡しを値渡しと比較しています。変数 a を値で渡し、変数 b を参照で渡します。

ObjectScript

```
Main
   SET a="6",b="7"
     WRITE "Initial values:",!
     WRITE "a=",a," b=",b,!
   DO DoubleNums(a,.b)
     WRITE "Returned to Main:",!
     WRITE "a=",a," b=",b
DoubleNums(foo,&bar) {
  WRITE "Doubled Numbers:",!
  SET foo=foo*2
  SET bar=bar*2
  WRITE "foo=",foo," bar=",bar,!
  }
```

以下の例では参照渡しを使用して、変数 result を通じて参照元のルーチンと関数の間で双方向通信します。ピリオド接頭語は、result が参照渡しされることを指定します。この関数を実行すると、実パラメータ・リストに result が作成され、関数の仮パラメータ・リストにある z にバインドされます。計算された値は z に代入され、result の参照元のルーチンに渡されます。仮パラメータ・リストの z に対する & 接頭語はオプションであり、機能もありませんが、ソース・コードの明確化に役立ちます。num と powr は、参照渡しではなく値渡しであることに注意してください。以下は、値渡しと参照渡しを組み合わせた例です。

ObjectScript

```
Start ; Raise an integer to a power.
 READ !,"Integer= ",num  RETURN:num=""
 READ !,"Power= ",powr   RETURN:powr=""
 SET output=$$Expo(num,powr,.result)
 WRITE !,"Result= ",output
 GOTO Start
Expo(x,y,&z)
 SET z=x
 FOR i=1:1:y {SET z=z*x}
 RETURN z
```

8.3.3 可変個数のパラメータ

プロシージャでは、可変個数のパラメータが使用可能であることを指定できます。そのように指定するには、vals... のように最後のパラメータ名に 3 つのドットを付加します。このドットを付加するパラメータは、パラメータのリストの最後にあるものとする必要があります。パラメータのリストに記述した唯一のパラメータであってもかまいません。この ... 構文を記述することで、1 つ以上のパラメータを渡すことができるほか、パラメータをまったく渡さないこともできます。

リストのパラメータ間、ならびにリストの最初のパラメータの前と最後のパラメータの後には、スペースと新しい行が認められています。3 つのドットの間の空白は認められていません。

この構文を使用するには、最後のパラメータ名の後に ... が続くシグニチャを指定します。この手法でメソッドに渡す複数のパラメータには、データ型による値またはオブジェクトによる値を設定できるほか、これらの設定によるパラメータを混在して使用することもできます。可変個数のパラメータの使用を指定するパラメータには、任意の有効な識別子名を設定できます。

ObjectScript では、添え字付き変数を作成することで、可変個数のパラメータを渡す処理を扱います。この場合は、渡す変数ごとに 1 つの添え字が作成されます。最上位の変数には、渡すパラメータの個数を記述し、変数の添え字には、渡す値を記述します。

これを、以下の例に示します。ここでは、invals... を仮パラメータ・リストにある唯一のパラメータとしています。ListGroceries(invals...) は、可変個数の値を値渡しで受け取ります。

ObjectScript

```
Main
   SET a="apple",b="banana",c="carrot",d="dill",e="endive"
   DO ListGroceries(a,b,c,d,e)
   WRITE !,"all done"
ListGroceries(invals...) {
  WRITE invals," parameters passed",!
   FOR i=1:1:invals {
      WRITE invals(i),! }
   }
```

以下の例では、2 つの定義済みパラメータの後に最後のパラメータとして args... を使用しています。このプロシージャは、3 つ目のパラメータから、可変個数の追加パラメータを受け取ることができます。最初の 2 つのパラメータは必須で、定義済みパラメータ `DO AddNumbers(a,b,c,d,e)` またはプレースホルダのコンマ `DO AddNumbers(,,c,d,e)` のいずれかを指定します。

ObjectScript

```
Main
   SET a=7,b=8,c=9,d=100,e=2000
   DO AddNumbers(a,b,c,d,e)
   WRITE "all done"
AddNumbers(x,y,morenums...) {
  SET sum = x+y
  FOR i=1:1:$GET(morenums, 0) {
     SET sum = sum + $GET(morenums(i)) }
  WRITE "The sum is ",sum,!
  }
```

以下の例では、2 つの定義済みパラメータの後に最後のパラメータとして args... を使用しています。このプロシージャが受け取るパラメータの個数は 2 つのみで、可変個数の追加パラメータ args... は 0 です。

ObjectScript

```
Main
   SET a=7,b=8,c=9,d=100,e=2000
   DO AddNumbers(a,b)
   WRITE "all done"
AddNumbers(x,y,morenums...) {
  SET sum = x+y
  FOR i=1:1:$GET(morenums, 0) {
     SET sum = sum + $GET(morenums(i)) }
  WRITE "The sum is ",sum,!
  }
```

指定に応じて、`AddNumbers(x,y,morenums...)` は 2 個以上、255 個以下の範囲でパラメータを受け取ることができます。定義済みのパラメータに既定値を指定して `AddNumbers(x=0,y=0,morenums...)` とすると、このプロシージャは 0 個以上、255 個以下の範囲でパラメータを受け取ることができます。

以下の例では、唯一のパラメータとして args... を使用しています。これは参照渡しで可変個数の値を受け取ります。

ObjectScript

```
Main
   SET a=7,b=8,c=9,d=100,e=2000
   DO AddNumbers(.a,.b,.c,.d,.e)
   WRITE "all done"
AddNumbers(&nums...) {
  SET sum = 0
  FOR i=1:1:$GET(nums, 0) {
     SET sum = sum + $GET(nums(i)) }
  WRITE "The sum is ",sum,!
  RETURN sum
  }
```

変数パラメータ・リスト params... が参照によって渡されたパラメータを受け取り、params... をルーチンに渡すと、その中間ルーチンは、参照によっても渡される追加パラメータ (params 配列の追加ノード) を追加できます。以下に例を示します。

ObjectScript

```
Main
   SET a(1)=10,a(2)=20,a(3)=30
   DO MoreNumbers(.a)
   WRITE !,"all done"
MoreNumbers(&params...) {
   SET params(1,6)=60
   SET params(1,8)=80
   DO ShowNumbers(.params) }
ShowNumbers(&tens...) {
   SET key=$ORDER(tens(1,1,""),1,targ)
   WHILE key'="" {
     WRITE key," = ",targ,!
     SET key=$ORDER(tens(1,1,key),1,targ)
   }
  }
```

以下の例では、この args... 構文を仮パラメータ・リストと実パラメータ・リストの両方で使用できることを示しています。この例の可変個数のパラメータ (invals...) は、ListNums に値で渡され、そこで 2 倍の値にされたうえで invals... として ListDoubleNums に渡されます。

ObjectScript

```
Main
   SET a="1",b="2",c="3",d="4"
   DO ListNums(a,b,c,d)
   WRITE !,"back to Main, all done"
ListNums(invals...) {
  FOR i=1:1:invals {
    WRITE invals(i),!
    SET invals(i)=invals(i)*2 }
  DO ListDoubleNums(invals...)
  WRITE "back to ListNums",!
}
ListDoubleNums(twicevals...) {
  WRITE "Doubled Numbers:",!
  FOR i=1:1:twicevals {
    WRITE twicevals(i),! }
}
QUIT
```

"可変個数の引数の指定" も参照してください。

8.4 プロシージャ・コード

中括弧内のコードがプロシージャのコードですが、従来の ObjectScript コードとは異なります。以下はその詳細です。

- プロシージャに入ることができるのは、プロシージャ・ラベルの場所のみです。label+offset 構文を使ってプロシージャにアクセスすることはできません。

- プロシージャのラベルは、プロシージャに対しプライベートであり、プロシージャ内からのみアクセスできます。PRIVATE キーワードは不要ですが、プロシージャ内のラベルで PRIVATE キーワードを使用できます。PUBLIC キーワードは、プロシージャ内のラベルで使用すると構文エラーが発生します。$TEXT システム関数は、プロシージャ・ラベル名を使用している label+offset をサポートしていますが、名前でプライベート・ラベルにアクセスできません。
- ラベルの重複はプロシージャ内では許可されていませんが、特定の状況ではルーチン内で許可されています。特に、異なるプロシージャ内でラベルは重複して使用できます。また、同じラベルはプロシージャ内や、プロシージャが定義されたルーチン内の他の場所で表示できます。例えば、以下の 3 つの `Label1` は使用できます。

ObjectScript

```
Rou1 // Rou1 routine
Proc1(x,y) {
Label1 // Label1 within the proc1 procedure within the Rou1 routine
}

Proc2(a,b,c) {
Label1 // Label1 within the Proc2 procedure (local, as with previous Label1)
}

Label1 // Label1 that is part of Rou1 and neither procedure
```

- プロシージャにルーチン名なしの Do コマンドやユーザ定義関数がある場合、ラベルが存在しているとプロシージャ内のラベルを参照します。そうでない場合、ルーチン内ではあるがプロシージャの外部にあるラベルを参照します。
- ルーチン名を持つ DO またはユーザ定義関数がプロシージャにある場合、そのプロシージャでは必ずその外部にある行を特定します。その名前がプロシージャを含むルーチンを識別する場合も同様です。例えば以下のようになります。

ObjectScript

```
ROU1 ;
PROC1(x,y) {
 DO Label1^ROU1
Label1 ;
 }
Label1 ; The DO calls this label
```

- プロシージャに GOTO が含まれる場合、プロシージャ内のプライベート・ラベルに移動する必要があります。GOTO では、プロシージャを終了できません。
- label+offset 構文は、いくつかの例外はありますが、プロシージャ内ではサポートされていません。
  - $TEXT は、プロシージャ・ラベルから label+offset をサポートします。
  - Break またはエラー発生後にプロシージャに戻る方法として、GOTO label+offset を、プロシージャ・ラベルから始まるダイレクト・モード行で使用できます。
  - ZBREAK コマンドでは、プロシージャ・ラベルから始まる label+offset を指定できます。
  - プロシージャが呼び出されたときに有効になっていた $TEST 状態は、そのプロシージャが終了すると復元されます。
  - プロシージャの終了を意味する `}` は、行の最初の文字位置を含め、どの文字位置にでも置くことができます。コードは、行の `}` の前に記述します。その後ろには記述できません。
  - 右中括弧の直前に、暗黙的な QUIT が置かれます。
  - 間接指定と XECUTE コマンドは、プロシージャの範囲外に存在するかのように動作します。

8.5 プロシージャ範囲内の間接指定、XECUTE コマンド、および JOB コマンド

プロシージャ内の名前間接指定、引数間接指定、XECUTE コマンドは、プロシージャの範囲内では実行されません。したがって、XECUTE コマンドは、プロシージャ外にあるサブルーチンの暗黙的な DO のように動作します。

間接演算子と XECUTE は、パブリック変数にのみアクセスできます。その結果、間接演算子または XECUTE コマンドが変数 x を参照する場合、プライベート変数である x がプロシージャに存在していても、パブリック変数である x を参照します。例えば以下のようになります。

ObjectScript

```
SET x="set a=3" XECUTE x ; sets the public variable a to 3
SET x="label1" DO @x ; accesses the public subroutine label1
```

同様に、間接指定または XECUTE で参照するラベルは、プロシージャ外のラベルです。プロシージャ内での GOTO は、その内部のラベルに移動する必要があるので、GOTO @A はプロシージャ内でサポートされていません。

間接演算と XECUTE コマンドの詳細は、このドキュメントの他の個所で説明されています。

同様に、プロシージャの範囲内で JOB コマンドを発行する場合、メソッド外の子プロセスを開始します。つまり、次のようなコードになります。

ObjectScript

```
    KILL ^MyVar
    JOB MyLabel
    QUIT $$$OK
MyLabel
    SET ^MyVar=1
    QUIT
```

子プロセスでラベルが見えるようにするには、プロシージャ・ブロックにメソッドまたはクラスを含めることはできません。

8.6 プロシージャ内でのエラー・トラップ

エラー・トラップがプロシージャ内から設定された場合、プロシージャのプライベート・ラベルに直接設定する必要があります。(これは、+offset やルーチン名を含む従来のコードとは異なります。この規則は、エラー・トラップの実行とは、基本的にエラー・トラップまでスタックを巻き戻し、GOTO を実行することである、という考えに一致しています。)

プロシージャ内でエラーが発生した場合、$ZERROR はプライベートの label+offset ではなく、プロシージャの label+offset に設定されます。

エラー・トラップを設定するには、通常の $ZTRAP を使用します。ただし、値はリテラルにする必要があります。以下はその例です。

ObjectScript

```
SET $ZTRAP = "abc"
// sets the error trap to the private label "abc" within this block
```

エラー・トラップの詳細は、"エラー処理" を参照してください。

8.7 従来のユーザ定義コード

InterSystems IRIS にプロシージャが追加される前、ユーザ定義コードはサブルーチンと関数の形式でサポートされていました(これらは現在、プロシージャとして実装できます)。ここでは、これらの従来のユーザ定義コードについて説明していますが、これは既存のコードを説明することが主な目的です。これらを継続して使用することはお勧めしません。

8.7.1 サブルーチン

8.7.1.1 構文

以下は、ルーチンの構文です。

```
label [ ( param [ = default  ][ , ...] ) ]
   code
   QUIT
```

以下の構文で呼び出します。

```
DO label [ ( param [ , ...]  ) ]
```

あるいは以下のようになります。

```
GOTO label
```

引数	説明
label	サブルーチンの名前標準ラベルです。必ず 1 列目から始まります。label の後に続くパラメータの括弧はオプションです。パラメータが指定されたサブルーチンは、GOTO では呼び出されません。パラメータの括弧は、直前に実行されるコードがサブルーチン内に“フォールスルーする”のを防ぎます。InterSystems IRIS は、パラメータの括弧付きラベルに遭遇した場合(中身が空でも)、暗黙の QUIT を実行し、ルーチンの次の行に実行が移らないように実行を終了します。
param	呼び出し元のプログラムからサブルーチンに渡されるパラメータの値。GOTO コマンドを使用して呼び出されるサブルーチンは、param 値を持つことはできず、パラメータの括弧を使うこともできません。DO コマンドを使用して呼び出されるサブルーチンには、param 値があってもなくてもかまいません。param 値がない場合、空のパラメータの括弧は記述しても省略してもかまいません。サブルーチンに必要な各パラメータに対し、param 変数を指定します。必要なパラメータは、仮パラメータ・リスト と呼ばれます。このリストには、何も指定しなくても、1 つ以上の param を定義してもかまいません。複数の param 値がある場合、コンマによって区切られます。InterSystems IRIS は、参照された param 変数に対して、自動的に NEW を呼び出します。パラメータは、値または参照で仮パラメータ・リストに渡すことができます。
default	直前の param に対するオプションの既定値です。各パラメータに対して既定値を提供しても、省略してもかまいません。仮パラメータに対応する実パラメータが指定されていない場合、または実パラメータが参照渡しされたときに当該のローカル変数に値がない場合は、既定値が適用されます。この既定値はリテラル、つまり引用符で囲まれた数字あるいは文字列です。null 文字列("") も既定値として指定できます。null 文字列は変数を定義するため、これは既定値を指定しないこととは異なります。一方でパラメータ値が未指定の場合や既定値がない場合、変数は未定義となります。リテラル以外の既定値を指定すると、InterSystems IRIS は <PARAMETER> エラーを発行します。

引数	説明
code	コード・ブロックです。コード・ブロックは、通常ラベルを呼び出してアクセスします。しかし、コード・ブロック内で別のラベルを呼び出したり、label + offset の GOTO コマンドを使用しても入力 (再入力) できます。コード・ブロックでは、他のサブルーチン、関数、プロシージャを入れ子にして呼び出すことができます。入れ子になった呼び出しは、リンクされた一連の GOTO コマンドではなく、DO コマンドまたは関数呼び出しを使用して実行することをお勧めします。コード・ブロックは、通常、明示的な QUIT コマンドで終了します。必須ではありませんが、QUIT コマンドの記述をお勧めします。外部ラベルに GOTO を使用して、サブルーチンを終了することもできます。

8.7.1.2 概要

サブルーチンは、最初の行の 1 列目にあるラベルで識別されるコード・ブロックです。サブルーチンの実行は、通常、明示的な QUIT 文によって終了します。

サブルーチンは、DO コマンドまたは GOTO コマンドで呼び出します。

- DO コマンドはサブルーチンを実行し、その後、呼び出し元のルーチンの実行を再開します。したがって、InterSystems IRIS がサブルーチンで QUIT コマンドに遭遇した場合、呼び出し元のルーチンに戻り、DO コマンドに続く次の行を実行します。
- GOTO コマンドは、サブルーチンを実行しますが、呼び出し元のプログラムに制御を戻しません。InterSystems IRIS は、サブルーチンの中で QUIT コマンドに遭遇したときに実行を終了します。

DO コマンドで呼び出されたサブルーチンにパラメータを渡すことができますが、GOTO コマンドで呼び出されたサブルーチンには渡せません。値、もしくは参照によってパラメータを渡せます。詳細は、"パラメータ渡し" を参照してください。

サブルーチンと呼び出し元のルーチンは、同じ変数を使用できます。

サブルーチンは値を返しません。

8.7.2 関数

関数は既定でプロシージャ (推奨) となります。ただし、プロシージャではない関数を定義することも可能です。このセクションでは、このような関数について説明しています。

8.7.2.1 構文

以下はプロシージャではない関数の構文です。

```
label ( [param [ = default  ]] [ , ...] )
   code
   QUIT expression
```

以下の構文で呼び出します。

```
command $$label([param[ ,...]])
```

あるいは以下のようになります。

```
DO label([param[ ,...]])
```

引数	説明
label	関数名。標準ラベルです。必ず 1 列目から始まります。label に続くパラメータの括弧は必須です。
param	関数に必要な各パラメータの変数です。必要なパラメータは、仮パラメータ・リストと呼ばれます。このリストには、何も指定しなくても、1 つ以上の param を定義してもかまいません。複数の param 値がある場合、コンマによって区切られます。InterSystems IRIS は、参照された param 変数に対して、自動的に NEW を呼び出します。パラメータは、値または参照で仮パラメータ・リストに渡すことができます。
default	直前の param に対するオプションの既定値です。各パラメータに対して既定値を提供しても、省略してもかまいません。仮パラメータに対応する実パラメータが指定されていない場合、または実パラメータが参照渡しされたときに当該のローカル変数に値がない場合は、既定値が適用されます。この既定値はリテラル、つまり引用符で囲まれた数字あるいは文字列です。null 文字列 () も既定値として指定できます。null 文字列は変数を定義するため、これは既定値を指定しないこととは異なります。一方でパラメータ値が未指定の場合や既定値がない場合、変数は未定義となります。リテラル以外の既定値を指定すると、InterSystems IRIS は <PARAMETER> エラーを発行します。
code	コード・ブロックです。コード・ブロックでは、他のサブルーチン、関数、プロシージャを入れ子にして呼び出すことができます。入れ子になった呼び出しは、DO コマンドまたは関数呼び出しで実行する必要があります。GOTO コマンドでは、関数のコード・ブロックを終了できません。コード・ブロックは、式を持つ明示的な QUIT コマンドを使用してのみ終了できます。
expression	有効な ObjectScript 式で指定される関数返り値です。式を持つ QUIT コマンドは、ユーザ定義関数では必須です。式から生じる値は、関数の結果として呼び出し元に返されます。

8.7.2.2 概要

このセクションでは、ユーザ定義関数について説明します。ユーザ定義関数の呼び出しは、接頭語 $$ で識別されます。(ユーザ定義関数は、外部関数 ともいいます)。

ユーザ定義関数により、InterSystems IRIS が提供する関数に新たな関数を追加できます。通常、どのプログラムからでも呼び出せる汎用的な処理を実装するために関数を使用します。

関数は、常に ObjectScript コマンドから呼び出されます。式として評価され、呼び出し元のコマンドに単一の値を返します。例えば以下のようになります。

ObjectScript

```
SET x=$$myfunc()
```

8.7.2.3 関数パラメータ

原則として、ユーザ定義関数はパラメータ渡しを使用します。ただし、外部に提供する値を持たない場合も、関数は機能します。例えば、乱数を生成する関数の定義や、システム日付を既定以外の形式で返す関数の定義ができます。このような場合、パラメータの括弧は、パラメータ・リストが空でも関数定義と関数呼び出しの両方に必要です。

パラメータ渡しでは、以下が必要です。

- 関数呼び出しの実パラメータ・リスト
- 関数定義の仮パラメータ・リスト

InterSystems IRIS がユーザ定義関数を実行するとき、実パラメータ・リストのパラメータが、それに対応する仮リストのパラメータに位置を基準としてマップされます。例えば、実リストにある最初のパラメータの値は、仮リストにある最初の変数に配置し、2 番目の値は 2 番目の変数に配置し、以降の値も同様に配置します。これらのパラメータの照合は、名前ではなく、パラメータの位置で行われます。したがって、実パラメータに使用する変数と仮パラメータに対して使用する変数は、同じ名前を持つ必要はありません(また、通常持つべきではありません)。関数は、仮リスト内の該当する変数を参照することにより、渡された値にアクセスします。

仮リストにある変数が実リストにあるパラメータよりも多く、既定値が各パラメータに指定されていない場合、余った変数は未定義のままとなります。したがって、関数のコードには適切な If テストを記述して、各関数の参照において使用可能な値が必ず渡されることを確認する必要があります。

ユーザ定義関数にパラメータを渡す場合、値または参照で渡すことができます。同じ関数呼び出し内で、値または参照渡しの両方を混合して指定することができます。詳細は、"パラメータ渡し" を参照してください。

8.7.2.4 返り値

ユーザ定義関数を定義するための構文は以下のとおりです。

```
label ( parameters  )
   code
   QUIT expression
```

関数は、QUIT コマンドの後に式を含む必要があります。InterSystems IRIS は、QUIT に遭遇すると関数の実行を終了し、関連式が生成する単一値を呼び出し元プログラムに返します。

式を含めずに QUIT コマンドを指定すると、InterSystems IRIS はエラーを生成します。

8.7.2.5 変数

呼び出し元のプログラムと呼び出される関数は、同じ変数一式を使用します。以下は特に考慮すべき点です。

- InterSystems IRIS は仮リストの各パラメータに対し、暗黙の NEW コマンドを実行します。以下の例では、myfunc が呼び出されると x が再度初期化されます。

  ObjectScript

  ```
  mainprog
   SET x=7
   SET y=$$myfunc(99)
  myfunc(x)
   WRITE x
   QUIT 66
  ```

- システムは、関数を開始した時点でシステム変数 $TEST の現在値を保存し、関数の終了時にその値を復元します。他の方法を使用して値を保存するコードを明示的に記述しない限り、関数実行中に $TEST 値を変更しても、関数の終了時にその値は破棄されます。

8.7.2.6 関数の位置

ユーザ定義関数は、それを参照するルーチン内に定義する、あるいは複数のプログラムから参照できるよう別々のルーチン内に定義できます。しかし、1 つのルーチンにすべてのユーザ定義関数を定義することをお勧めします。これにより、関数の検証や更新の際に、関数定義の検索が簡単にできます。

8.7.2.7 ユーザ定義関数の呼び出し

$$ 接頭語または DO コマンドを使用して、ユーザ定義関数を呼び出すことができます。$$ 形式は最も機能的であり、一般的に好まれる形式です。

$$ 接頭語の使用法

式を使用できるコンテキスト内で、ユーザ定義関数を呼び出すことができます。以下の形式で、ユーザ定義関数を呼び出します。

```
$$name([param [,...]])
```

以下はその説明です。

- name 関数の名前を指定します。関数が定義される場所に応じて、以下のように名前を指定できます。
  - label ー 現在のルーチン内の行ラベル
  - label^routine ー ディスクにある名前付きルーチン内の行ラベル。
  - ^routine ー ディスクに保存されているルーチン。ルーチンは、実行する関数のコードのみを含んでいる必要があります。

  ルーチンはネームスペース内で定義されます。拡張ルーチン参照を使用し、以下のようにして現在のネームスペース以外のネームスペースで定義されたルーチン内にあるユーザ定義関数を実行できます。

  ObjectScript

  ```
  WRITE $$myfunc^|"USER"|routine
  ```

- param は関数に渡される値を指定します。提供されるパラメータは、実パラメータ・リストと呼ばれます。これらは、その関数に関して定義された仮パラメータ・リストと一致する必要があります。例えば、関数コードには 2 つのパラメータがあり、1 番目は数値、2 番目は文字リテラルとします。この場合、1 番目のパラメータに文字リテラル値、2 番目に数値を指定すると、関数は不正確な値を返すか、あるいはエラーを生成する可能性があります。仮パラメータ・リストのパラメータでは、関数から NEW コマンドが必ず呼び出されます。詳細は、NEW コマンドの説明を参照してください。パラメータは、値または参照によって渡されます。詳細は、"パラメータ渡し" を参照してください。関数の仮パラメータ・リストに記載されている数より少数のパラメータを関数に渡すと、パラメータの既定値が使用されます（定義されている場合）。既定値がない場合、パラメータは未定義のままとなります。

DO コマンドの使用法

DO コマンドを使用して、ユーザ定義関数を呼び出すことができます（DO コマンドを使用して、システム関数を呼び出すことはできません）。DO コマンドで呼び出した関数は、値を返しません。つまり、関数は必ず返り値を生成しますが、DO コマンドはこの返り値を無視します。したがって、ユーザ定義関数を呼び出す DO コマンドの用途は大幅に制限されます。

DO コマンドを使用してユーザ定義関数を呼び出すには、以下の構文でコマンドを発行します。

```
DO label(param[ ,...])
```

DO コマンドは、label という関数を呼び出し、param で指定されたパラメータ（存在する場合）を関数に渡します。$$ 接頭語を使用しないことと、パラメータの括弧が必須であることに注意してください。label と param を指定する際には、$$ 接頭語を使用するユーザ定義関数を呼び出すときと同じ規則が適用されます。

関数は、常に値を返す必要があります。しかし、DO コマンドで関数を呼び出す場合、返された値は呼び出し元のプログラムで無視されます。

9

ObjectScript マクロとマクロ・プリプロセッサ

ObjectScript コンパイラには、プリプロセッサが組み込まれており、ObjectScript にはプリプロセッサ指示文に対するサポートが含まれています。これらの指示文を使用すると、アプリケーション (メソッドとルーチンの両方) で使用するマクロを作成できます。これらのマクロは、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。InterSystems IRIS® 自体にもさまざまな事前定義済みマクロが組み込まれています。これらについては、このドキュメントの該当コンテキストで説明します。

9.1 マクロの使用

このセクションでは、以下のトピックについて説明します。

- カスタム・マクロの作成
- カスタム・マクロの保存
- マクロの呼び出し
- 外部マクロ (インクルード・ファイル) の参照

9.1.1 カスタム・マクロの作成

マクロは、1 行の置換定義によって、ObjectScript の多様な機能をサポートできます。その基本形式では、#define 指示文を使用してマクロを作成します。例えば以下のコードは、`StringMacro` というマクロを生成し、そのマクロを文字列“Hello, World!”に置換します。

ObjectScript

```
#define StringMacro "Hello, World!"
```

(##continue を使用すると、#define 指示文を次の行に継続できます。)

ObjectScript では、以下のような“$$$”構文を使用して、マクロを呼び出すことができます。

ObjectScript

```
  WRITE $$$StringMacro
```

この場合、文字列“Hello, World!”を表示します。以下は、サンプルの全コードです。

ObjectScript

```
#define StringMacro "Hello, World!"
   WRITE $$$StringMacro
```

以下はサポート対象の機能です。

- 上記の例のような文字列置換
- 数値置換

  ObjectScript

  ```
  #define NumberMacro 22
  ```

  ObjectScript

  ```
  #define 25M ##expression(25*1000*1000)
  ```

  通常、ObjectScript のマクロ定義では、文字列には引用符が必要ですが、数値には必要ありません。

- 変数置換

  ObjectScript

  ```
  #define VariableMacro Variable
  ```

  マクロ名は、既に定義済みの変数名に置換されます。変数が未定義の場合、<UNDEFINED> エラーが発生します。

- コマンドと引数の呼び出し

  ObjectScript

  ```
  #define CommandArgumentMacro(%Arg) WRITE %Arg,!
  ```

  マクロ引数名は、上記の %Arg 引数のように、“%” 文字で始まる必要があります。このマクロは、%Arg 引数を使用する WRITE コマンドを呼び出します。

- 関数、式、演算子の使用

  ObjectScript

  ```
  #define FunctionExpressionOperatorMacro ($ZDate(+$Horolog))
  ```

  ここでは、マクロ全体が式で、その値は、$ZDate 関数の返り値になります。$ZDate は、システム変数 $Horolog に格納されているシステム時刻に “+” 演算子を付けた演算の結果として生じる式を処理します。上記のように、式を括弧で囲むことをお勧めします。これにより、式が使用されている文とのやり取りが最小限に抑えられます。

- 他のマクロの参照

  ObjectScript

  ```
  #define ReferenceOtherMacroMacro WRITE $$$ReferencedMacro
  ```

  このマクロは、WRITE コマンドへの引数として、他のマクロの式の値を使用します。

  注釈　1 つのマクロが別のマクロを参照する場合、参照先のマクロは、参照元のマクロの前にコンパイル済みのコード行に表示されている必要があります。

9.1.1.1 マクロの名前付け規約

- 最初の文字は、英数字またはパーセント記号 (%) でなくてはならない。
- 2 番目以降の文字は英数字でなくてはならない。マクロ名には、スペース、アンダースコア、ハイフンなどの記号文字を含めることはできません。
- マクロ名では大文字と小文字が区別される。
- マクロ名は最長 500 文字。
- マクロ名には日本語の全角文字および半角かな文字を使用可能。詳細は、このドキュメントの “演算子と式” の章の ″パターン・マッチング″ のセクションにある “パターン・コード” 表を参照してください。
- ISCname.inc ファイルがシステム用に予約されているため、マクロ名を ISC で開始することはできません。

9.1.1.2 マクロの空白規約

- 規約では、マクロ指示文をインデントさせずに、列 1 にて記述することになっています。しかし、マクロ指示文はインデントさせても構いません。
- 1 つ以上のスペースをマクロ指示文に続けることができます。マクロ内では、任意数のスペースをマクロ指示文、マクロ名、マクロ値の間に入れることができます。
- マクロ指示文は単一行の文となります。マクロ指示文、マクロ名、およびマクロ値はすべて同一行で記述する必要があります。##continue を使用すると、マクロ指示文を次の行に継続できます。
- #if 指示文と #elseIf 指示文はテスト式の形をとります。テスト式にはスペースを含めることはできません。
- #if 式、#elseIf 式、#else 指示文、および #endif 指示文はすべて、それぞれの独立した行に記述します。同一行にてこれらの指示文のいずれか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。

9.1.1.3 マクロ・コメントとスタジオ・アシスト

マクロには、その定義の一部として渡すコメントを記述できます。/* と */、//、#;、;、および ;; で区切ったコメントは、すべて通常どおりに機能します。コメントの基本的な情報については、″ObjectScript の使用法″ の “構文規則” の章にある “コメント” セクションを参照してください。

/// で始まるコメントには特別な機能があります。インクルード・ファイルにあるマクロでスタジオ・アシストを使用する場合、マクロの定義の直前の行に /// コメントを記述します。これにより、そのマクロの名前がスタジオ・アシストのポップアップに表示されます (現在のファイルにあるすべてのマクロがスタジオ・アシストのポップアップに表示されます)。例えば、以下のコードを #include 指示文で参照する場合、最初のマクロの名前はスタジオ・アシストのポップアップに表示されますが、2 番目のマクロの名前は表示されません。

ObjectScript

```
/// A macro that is visible with Studio Assist
#define MyAssistMacro 100
 //
 // ...
 //
 // A macro that is not visible with Studio Assist
#define MyOtherMacro -100
```

インクルード・ファイルでマクロを使用可能にする方法については、“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。スタジオ・アシストに関する詳細は、″スタジオの使用法″ の “スタジオ・オプションの設定” の章の “エディタ・オプション” のセクションを参照してください。

9.1.2 カスタム・マクロの保存

マクロは、呼び出されるファイルか、別々のインクルード・ファイルのいずれかに置くことができます (通常はインクルード・ファイルです)。マクロをクラス全体で使用可能にする (つまり、どのメンバからでも呼び出し可能にする) には、その定義をインクルード・ファイルに記述し、そのファイルをクラスでインクルードします。

マクロをインクルード・ファイルに記述するには、スタジオで以下の手順に従います。

1. [ファイル] メニューから [保存] あるいは [名前を付けて保存] を選択します。
2. [名前を付けて保存] ダイアログで、[ファイル・タイプ] フィールドから Include File (*.inc) 値を指定します。

   注釈　このフィールドの Macro Routine (*.mac) の値は、ObjectScript マクロの正しいファイル・タイプではありません。

3. ディレクトリとファイル名を入力し、マクロを保存します。

9.1.3 マクロの呼び出し

定義するメソッドまたはルーチンの中にマクロの定義が存在する場合、または定義するメソッドまたはルーチンで #include 指示文を使用して外部ソースの定義を参照している場合に、マクロを呼び出すことができます。#include の詳細は、"外部マクロの参照" または #include のリファレンスを参照してください。

ObjectScript コード内からマクロを呼び出す場合、"$$$" を名前の前に付けて呼び出します。したがって、MyMacro というマクロを定義している場合、$$$MyMacro という名前で呼び出します。このマクロ名の大文字と小文字は区別されます。

変数値を指定できないコンテキストで、マクロを呼び出して代わりの値を使用できます。以下に例を示します。

ObjectScript

```
#define StringMacro "Hello",!,"World!"
   WRITE $$$StringMacro
```

ObjectScript

```
#define myclass "Sample.Person"
  SET x=##class($$$myclass).%New()
```

マクロはテキスト置換であることに注意してください。マクロが置換された後、結果として出力された文の構文が正確かどうかをチェックします。したがって、式を定義するマクロは、式を必要とするコンテキスト内で呼び出される必要があります。また、コマンドとその引数に対するマクロは、ObjectScript の独立した行として置くことができます。

9.1.4 外部マクロ (インクルード・ファイル) の参照

複数のマクロをそれぞれ別々のファイルに保存した場合でも、#include 指示文を使用すれば、そのマクロを利用できます。この指示文では大文字と小文字が区別されません。

クラス内またはルーチン開始部にマクロを含む場合、指示文は以下の形式になります。

ObjectScript

```
#include MacroIncFile
```

ここで、MacroIncFile は、MacroIncFile.inc と呼ばれるマクロを含むインクルード・ファイルを参照しています。参照先ファイルが #include の引数となっている場合には、.inc 接尾語が参照先ファイルの名前に含まれないことに注意してください。

例えば、MyMacros.inc というファイルに 1 つ以上のマクロがある場合、以下の呼び出しに組み込むことができます。

ObjectScript

```
#include MyMacros
```

YourMacros.inc というファイルに他のマクロがある場合、以下の呼び出しでそれらすべてを組み込むことができます。

ObjectScript

```
#include MyMacros
#include YourMacros
```

ルーチンで #include を使用する場合、各インクルード・ファイルで別々の行に個別の #include 文を指定する必要があります。

クラス定義の開始部にインクルード・ファイルを組み込むには、構文を以下の形式とします。

```
include MyMacros
```

クラス定義の開始部に複数のインクルード・ファイルを組み込むには、構文を以下の形式とします。

```
include (MyMacros, YourMacros)
```

この include 構文には、先頭にシャープ記号がないことに注意してください。この構文は #include には使用できません。また、スタジオでのコンパイルにより、単語の形式が変換されるので、最初の文字が大文字化され、後続の文字は小文字となります。

ObjectScript コンパイラは、外部マクロを含めることを許可する /defines 修飾子を提供します。詳細は、"ObjectScript リファレンス" の $SYSTEM 特殊変数の参照ページ内の "コンパイラ修飾子" のテーブルを参照してください。

また、#include のリファレンスも参照してください。

9.2 システム・プリプロセッサ・コマンド・リファレンス

InterSystems IRIS では、以下のシステム・プリプロセッサ指示文がサポートされています。

- #;
- #def1arg
- #define
- #dim
- #else
- #elseIf
- #endif
- #execute
- #if
- #ifDef
- #ifNDef
- #import

- #include
- #noshow
- #show
- #sqlcompile audit
- #sqlcompile mode
- #sqlcompile path
- #sqlcompile select
- #undef
- ##;
- ##beginquote ... ##EndQuote
- ##continue
- ##expression
- ##function
- ##lit
- ##quote
- ##quoteExp
- ##sql
- ##stripq
- ##unique

注釈　マクロ・プリプロセッサ指示文には、大文字と小文字の区別がありません。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。

9.2.1 #;

#; プリプロセッサ指示文は、.int コードには表示されない単行の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。

9.2.2 #def1arg

#def1arg 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記述できます。#def1arg 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

```
#def1arg 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。

#def1arg 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

```
#def1arg 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。

9.2.3 #define

#define 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、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。

9.2.3.1 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、#def1arg 指示文を使用できます。

以下は、さまざまな方法で使用さ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。

9.2.3.2 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
```

9.2.3.3 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こ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。

9.2.4 #dim

#dim 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、オプションで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もありません。以下の形式をとります。

ObjectScript

```
#dim VariableName As DataTypeName
#dim VariableName As DataType = InitialValue
#dim VariableName As List Of DataType
#dim VariableName As Array Of DataType
```

各項目の内容は次のとおりです。

- 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 には、VariableName の型を指定します。データ型の指定はオプションです。As DataType を省略して =InitialValue だけを指定できます。
- 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、主としてスタジオ・アシストで使用されます。例えば、[スタジオアシスト] ドロップダウン・メニューを使用して、ユーザ定義のデータ型のパッケージとクラスを選択できます。詳細は、このドキュメントの “変数” の章の "変数の宣言" を参照してください。

9.2.4.1 InitialValue

- VariableName にデータ変数のコンマ区切りのリストを指定し、DataType を省略すると、すべての変数に同じ初期値が割り当てられます。例を以下に示します。

  ObjectScript

  ```
  #dim a,b,c = ##class(Sample.Person).%New()
  ```

  これは以下と同じです。

ObjectScript

```
SET (a,b,c) = ##class(Sample.Person).%New()
```

各変数に同じ OREF を割り当てます。

- VariableName にデータ変数のコンマ区切りのリストを指定し、DataType が標準の %Library データ型の場合、すべての変数に同じデータ型と初期値が割り当てられます。例を以下に示します。

ObjectScript

```
#dim d,e,f As %String = ##class(Sample.Person).%New()
```

これは以下と同じです。

ObjectScript

```
SET (d,e,f) = ##class(Sample.Person).%New()
```

各変数に同じ OREF を割り当てます。

- 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 %Library データ型でない場合、各変数に別個の OREF が割り当てられます。例を以下に示します。

ObjectScript

```
#dim j,k,l As Sample.Person = ##class(Sample.Person).%New()
```

各変数に別個の OREF を割り当てます。

- 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・データ型の場合、各変数に別個の OREF が割り当てられます。例を以下に示します。

ObjectScript

```
#dim m,n,o As %DynamicObject = {"name":"Fred"}
#dim p,q,r As %DynamicArray = ["element1","element2"]
```

各変数に別個の OREF を割り当てます。

9.2.5 #else

#else 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。

9.2.6 #elseIf

#elseIf 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。

使用例は、#if を参照してください。

注釈　#elseIf には、#EllF 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。

9.2.7 #endif

#endif 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。

使用例は、#if を参照してください。

9.2.8 #execute

#execute 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。

9.2.9 #if

#if 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。

次に例を示します。

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。

9.2.10 #ifDef

#ifDef 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
```

9.2.11 #ifNDef

#ifNDef 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.",!
```

9.2.12 #import

#import プリプロセッサ指示文は、そ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。以下はその例です。

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 は、SQL の CALL 文にある未修飾プロシージャ名の解決にも使用できます。
- #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。

9.2.13 #include

#include 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のようになります。

```
Include MyMacros
```

複数ファイルでは以下のようになります。

```
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
```

9.2.14 #noshow

#noshow 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。

9.2.15 #show

#show 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。

9.2.16 #sqlcompile audit

#sqlcompile audit 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。デフォルトでは、このシステム監査イベントは有効になっていません。

9.2.17 #sqlcompile mode

#sqlcompile mode 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、"InterSystems SQL の使用法" の "埋め込み SQL" の章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。

9.2.18 #sqlcompile path

#sqlcompile path プリプロセッサ指示文は、そ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 は、SQL の CALL 文にある未修飾プロシージャ名の解決にも使用できます。
- #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スコープは、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スコープとなります。

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。

9.2.19 #sqlcompile select

#sqlcompile select プリプロセッサ指示文は、そ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には、%SYSTEM.SQL クラスの GetSelectMode メソッドを使用できます。実行時間選択モード値の設定には、%SYSTEM.SQL クラスの SetSelectMode メソッドを使用できます。
- 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、"InterSystems SQL の使用法" の “埋め込み SQL の使用法” の章の "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
```

9.2.20 #undef

#undef 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
```

ここで、macro-name は、既に定義されているマクロです。

#undef は、#define または #def1arg 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.",!
```

9.2.21 ##;

##; 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、#def1arg、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。

9.2.22 ##beginquote ...##EndQuote

##beginquote text ##EndQuote 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
```

9.2.23 ##continue

##continue 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>
```

9.2.24 ##expression

##expression 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 (下記の入れ子にされた ##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",!
```

9.2.24.1 ##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。

9.2.24.2 ##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、##BeginLit...##EndLit、##function、##lit、##quote、##SafeExpression、##stripq、##unique、##this の各マクロ関数を入れ子にすることもできます。

9.2.24.3 ##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。

9.2.24.4 ##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
```

9.2.25 ##function

##function 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。

9.2.25.1 ##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。

9.2.26 ##lit

##lit 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
```

.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。

9.2.27 ##quote

##quote 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
```

9.2.27.1 ##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。

9.2.28 ##quoteExp

##quoteExp 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。#def1arg 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。

```
#def1arg complexGlobal(%subs)
^GLO("dd"##expression($s(%literalargs'=$lb(""):","_$LTS(%literalargs,","),1:"")))
#def1arg complexGlobalQE(%subs) ##quoteExp($$$complexGlobal(%subs))
```

9.2.29 ##sql

##sql 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。

9.2.30 ##stripq

##stripq 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。

9.2.31 ##unique

##unique 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 #def1arg 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 #def1arg 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。

9.3 システムにより提供されるマクロの使用

このセクションでは、InterSystems IRIS で使用できる定義済みマクロのいくつかに関連するトピックについて説明します。トピックは以下のとおりです。

- システムにより提供されるマクロをアクセス可能にする方法
- システムにより提供されるマクロのリファレンス

9.3.1 システムにより提供されるマクロをアクセス可能にする方法

これらのマクロは、%RegisteredObject のすべてのサブクラスで使用可能です。%RegisteredObject の拡張でないルーチンまたはクラスの中でこれらのマクロを使用可能にするには、以下の適切な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、%occStatus.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、セクション“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。

9.3.2 システムにより提供されるマクロのリファレンス

マクロ名では大文字と小文字が区別される。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、"InterSystems IRIS エラー・リファレンス" の“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。これは %String を返します。

引数	説明
language	RFC1766 言語コード。Web アプリケーション内では、%response.Language 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、%response.Domain 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 $$$FormatMessage を使用できます。

メッセージ・ディクショナリの詳細は、"InterSystems Business Intelligence の実装" の "ローカライズの実行" 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) を受け付けます。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) を受け付けます。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) を受け付けます。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、このドキュメントの “エラー処理” の章の "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、このドキュメントの “エラー処理” の章の "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、このドキュメントの“エラー処理”の章の ″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、このドキュメントの“エラー処理”の章の ″TRY-CATCH メカニズム″ を参照してください。

9.4 マクロが拡張される条件

プリプロセッサ指示文は、MAC バージョンのコードに含まれています。MAC コードをコンパイルすると、ObjectScript コンパイラによって事前処理が実行され、INT (中間の読み取り可能な ObjectScript) コードおよび OBJ (実行可能オブジェクト) コードが生成されます。

注釈　プリプロセッサは、ObjectScript パーサによって埋め込み SQL が処理される前にマクロを拡張します。プリプロセッサは、埋め込みまたは遅延いずれかのコンパイル・モードの埋め込み SQL をサポートしています。プリプロセッサは、ダイナミック SQL でのマクロ拡張は行いません。

ObjectScript パーサは、プリプロセッサ指示文を解析する前に、複数の行から成るコメントを削除します。そのため、/* と*/ で囲んで記述した複数行コメントの中で指定したマクロ・プリプロセッサ指示文は、実行されません。

以下のグローバルは MAC コード情報を返します。これらのグローバルとその添え字を表示するには、ZWRITE を使用します。

- ^rINDEX(routinename,″MAC″) には、MAC コードが変更後に最後に保存された日時を表すタイムスタンプと、この MAC コード・ファイルの文字数が含まれます。文字数には、コメントおよび空白行が含まれます。MAC コードが最後に保存された日時を表すタイムスタンプ、コンパイルされた日時を表すタイムスタンプ、および使用された #include ファイルに関する情報は、.INT ファイルの ^ROUTINE グローバルに記録されます。.INT コードの詳細は、″ZLOAD″ コマンドを参照してください。
- ^rMAC(routinename) には、MAC ルーチンの各コード行の添え字ノードが含まれます。また、^rMAC(routinename,0,0) には行数、^rMAC(routinename,0) には MAC ルーチンが最後に保存された日時のタイムスタンプ、^rMAC(routinename,0,"SIZE") には文字数がそれぞれ含まれます。
- ^rMACSAVE(routinename) には、MAC ルーチンの履歴が含まれます。ここには、保存された過去の MAC ルーチン・バージョン 5 つ分の ^rMAC(routinename) と同じ情報が含まれます。現在の MAC バージョンに関する情報は含まれません。

10
埋め込み SQL

InterSystems IRIS® Data Platform の ObjectScript 内に SQL を埋め込むことができます。

10.1 埋め込み SQL

埋め込み SQL を使用すると、ObjectScript プログラム内に SQL コードを含めることができます。構文は `&sql( )` です。例えば、以下のようになります。

ObjectScript

```
&sql( SELECT Name INTO :n FROM Sample.Person )
WRITE "name is: ",n
```

埋め込み SQL は、これを含むルーチンのコンパイル時にはコンパイルされません。埋め込み SQL のコンパイルは、SQL コードの最初の実行時 (ランタイム) に行われます。

詳細は、"InterSystems SQL の使用法" の "埋め込み SQL の使用法" の章を参照してください。

11

多次元配列

InterSystems IRIS® Data Platform は、多次元配列をサポートします。多次元配列は、1 つ以上の要素を含む永続変数で、それぞれが一意の添え字を持ちます。ユーザは異なる種類の添え字を組み合わせて使用できます。例は、以下の MyVar 配列です。

- MyVar
- MyVar(22)
- MyVar(-3)
- MyVar("MyString")
- MyVar(-123409, "MyString")
- MyVar("MyString", 2398)
- MyVar(1.2, 3, 4, "Five", "Six", 7)

配列ノード MyVar は ObjectScript 変数であり、この変数タイプの規約に従います。

MyVar の添え字は、正と負の数、文字列、あるいはこれらの組み合わせになります。添え字には Unicode 文字をはじめ、あらゆる文字を含めることができます。数値添え字はキャノニック形式の数として格納および参照されます。文字列の添え字は大文字と小文字を区別するリテラルとして格納および参照されます。キャノニック形式の数 (または、キャノニック形式の数へと変わる数) およびキャノニック形式の数を含む文字列は同等の添え字となります。

11.1 多次元配列の概要

多次元配列とは、添え字で示される n 次元の永続配列です。個別ノードは "グローバル" ともいい、InterSystems IRIS データ・ストレージの構成要素でもあります。以下のような他の特性もあります。

- ツリー構造で存在します。
- スパース (まばらな) 配列
- 3 つの基本的な種類の 1 つです。

11.1.1 多次元ツリー構造

多次元配列の全体的な構造は ツリー と呼ばれ、上から下に拡張します。ルート、上記の MyVar が頂点です。ルートと他の添え字が付いた形式を ノード と呼びます。下にノードを持たないノードは、葉と呼ばれます。下にノードを持つノー

ドは、親や先祖と呼ばれます。親を持つノードは、子や子孫と呼ばれます。また、同じ親を持つ子は兄弟と呼ばれます。すべての兄弟がツリーに追加されると、数値あるいはアルファベット順で自動的に並べ替えられます。

11.1.2 スパース多次元ストレージ

多次元配列はスパース(まばらな)配列です。つまり、上記の例では、それぞれ定義されたノードに対し、7 つのメモリ位置のみを使用しています。また、配列を宣言したり、ディメンジョンを指定したりする必要がないため、以下のようなメモリに関する利点があります。すなわち、多次元配列に対しては事前に予約される領域がない、多次元配列は必要になるまで領域を使用しない、および多次元配列が使用する領域はすべて動的に割り当てられる、の 3 点です。例えば、配列を使用して、チェッカーと呼ばれるゲームでのプレーヤの区画を追跡するとします。チェッカーボードは、8 × 8 です。8 × 8 のチェッカーボード・サイズの配列を必要とする言語では、チェッカーによって占められる位置が 24 個を超えることがないにもかかわらず、64 個のメモリ位置が使用されます。一方で、ObjectScript では、配列は最初に必要とする位置は 24 個となり、必要な位置はゲームの進行と共にさらに少なくなります。

11.1.3 多次元配列の種類

多次元配列は、3 つの基本的な種類のいずれかになります。いずれの場合も、配列は、1 つ以上の添え字を持つという点でスカラ式と区別できます。

- 添え字を持つローカル変数はすべて配列です。例えば、x(1) を作成すると、x が配列として定義されます。
- 添え字を持つグローバルはすべて配列です。例えば、^y(1) を作成すると、^y が配列として定義されます。
- クラス内のプロパティは、その定義に `MultiDimensional` キーワードがあれば多次元配列にすることができます。例えば、以下のようになります。

  ```
  Property MyProp as %String [ MultiDimensional ];
  ```

  つづいて、以下のような文で値を設定できます。

  ```
  myObj.MyProp(1) = "hello world"
  ```

  永続クラスまたはシリアル・クラスの多次元プロパティは、プロパティを保存するカスタム・コードを記述しない限り、オブジェクトを保存してもディスクに保存されません。

11.2 多次元配列の操作

Read コマンドと Write コマンドをそれぞれ使用して、多次元配列からデータの読み取りと書き込みができます。

InterSystems IRIS は、多次元配列と連動する包括的なコマンドと関数一式を提供します。

- Set コマンドは、配列に値を設定します。
- Kill コマンドは、配列構造のすべて、または一部を削除します。
- Merge コマンドは、配列構造のすべてまたは一部を 2 番目の配列構造にコピーします。
- $Order および $Query コマンドは、配列のコンテンツ全体に対して繰り返すことができます。
- $Data コマンドは、配列内にノードが存在するかどうかをテストできます。

このコマンドと関数一式は、多次元グローバルと多次元ローカル配列で生成できます。グローバルは“^”(キャレット文字) で始まるため、簡単に識別できます。

11.3 詳細

多次元配列の詳細は、"グローバルの使用法" を参照してください。

12
文字列演算

InterSystems IRIS® Data Platform の ObjectScript は、それぞれ独自の目的と機能に合わせて文字列に関連する演算グループを提供します。これらは、基本的な文字列演算と関数、区切り文字列演算、およびリスト構造文字列演算です。

12.1 基本的な文字列演算と関数

ObjectScript の基本的な文字列演算では、文字列でさまざまな処理を実行できます。その処理は、以下のとおりです。

- $LENGTH 関数は、文字列の文字数を返します。以下はそのコードの例です。

  ObjectScript

  ```
  WRITE $LENGTH("How long is this?")
  ```

  これは、文字列の長さ、17 を返します。詳細は、"ObjectScript リファレンス" で "$LENGTH" のリファレンス・ページを参照してください。

- $JUSTIFY は、文字列の左側をスペース文字で埋め、右揃えした文字列を返します (数値演算も実行できます)。以下はコードの例です。

  ObjectScript

  ```
  WRITE "one",!,$JUSTIFY("two",8),!,"three"
  ```

  これは、文字列 “two” の位置を 8 文字に調整し、返します。

  ```
  one
       two
  three
  ```

  詳細は、"ObjectScript リファレンス" で "$JUSTIFY" のリファレンス・ページを参照してください。

- $ZCONVERT は、文字列の形式を変換します。大文字と小文字の変換 (大文字変換、小文字変換、タイトル文字変換など) とコード変換 (さまざまな形式でコード化された文字間の変換) をサポートします。以下はコードの例です。

  ObjectScript

  ```
  WRITE $ZCONVERT("cRAZy cAPs","t")
  ```

  これは、以下を返します。

  ```
  CRAZY CAPS
  ```

詳細は、"ObjectScript リファレンス" で "$ZCONVERT" のリファレンス・ページを参照してください。

- $FIND 関数は、文字列を部分文字列で検索し、その部分文字列の次にある文字の位置を返します。以下はコードの例です。

  ObjectScript

  ```
  WRITE $FIND("Once upon a time...", "upon")
  ```

  これは、“upon” の直後の文字位置である 10 を返します。詳細は、"ObjectScript リファレンス" で "$FIND" のリファレンス・ページを参照してください。

- $TRANSLATE 関数は、文字列内で文字単位の置換を行います。以下はコードの例です。

  ObjectScript

  ```
  SET text = "11/04/2008"
  WRITE $TRANSLATE(text,"/","-")
  ```

  これは、日付のスラッシュをハイフンに置き換えます。詳細は、"ObjectScript リファレンス" で "$TRANSLATE" のリファレンス・ページを参照してください。

- $REPLACE 関数は、文字列内で文字列単位の置換を行います。演算の対象とする文字列の値は変更しません。以下はコードの例です。

  ObjectScript

  ```
  SET text = "green leaves, brown leaves"
  WRITE text,!
  WRITE $REPLACE(text,"leaves","eyes"),!
  WRITE $REPLACE(text,"leaves","hair",15),!
  WRITE text,!
  ```

  これは、2 つの異なる処理を行います。最初の呼び出しの $REPLACE では、文字列 `leaves` を文字列 `eyes` に置き換えます。2 番目の呼び出しの $REPLACE では、(4 番目の引数で指定された位置である) 15 番目の文字の前にあるすべての文字を破棄し、文字列 `leaves` を文字列 `hair` に置き換えます。text 文字列の値は、どの $REPLACE 呼び出しでも変更されません。詳細は、"ObjectScript リファレンス" で "$REPLACE" のリファレンス・ページを参照してください。

- $EXTRACT 関数は、文字列の中で指定の位置から部分文字列を返します。以下はコードの例です。

  ObjectScript

  ```
  WRITE $EXTRACT("Nevermore"),$EXTRACT("prediction",5),$EXTRACT("xon/xoff",1,3)
  ```

  これは、3 つの文字列を返します。1-引数形式は文字列の最初の文字、2-引数形式は文字列から指定された文字、3-引数形式は指定の文字から始まり指定の文字で終わる部分文字列をそれぞれ返します。ここでは改行がないため、以下の値を返します。

  ```
  Nixon
  ```

  詳細は、次のセクションまたは "ObjectScript リファレンス" で "$EXTRACT" のリファレンス・ページを参照してください。

12.1.1 $EXTRACT の高度な機能

$EXTRACT 関数と SET コマンドを組み合わせて使用することで、文字列の左側をスペースで埋めることができます。

ObjectScript

```
SET x = "abc"
WRITE x,!
SET $EXTRACT(y, 3) = x
SET x = y
WRITE x
```

このコードは、文字列“abc”を取得し、文字列y の 3 番目に置きます。y には値が指定されていないので、$EXTRACT では、文字列に対する埋め込み文字として機能する空白と見なされます。

$EXTRACT を使用して、変数の特定位置に新たな文字列を挿入することもできます。新、旧文字列の長さが同じかどうかにかかわらず、指定の文字を抜き出し、そこに提供された部分文字列を置きます。例えば以下のようになります。

ObjectScript

```
SET x = "1234"
WRITE x,!
SET $EXTRACT(x, 3) = "abc"
WRITE x,!
SET $EXTRACT(y, 3) = "abc"
WRITE y
```

このコードは、x の値として“1234”を設定します。次に $EXTRACT を使用して、x の 3 番目の文字を抜き出し、その位置に“abc”を挿入します。その結果、文字列“12abc4”が作成されます。

12.2 区切り文字列演算

InterSystems IRIS には、文字列を部分文字列として処理できる機能があります。この機能により、データの一部分を 1 つの完全なデータとして格納する処理が可能です。以下はその関数です。

- $PIECE ー 指定の区切り文字に基づいて文字列内の特定の部分を返します。その範囲や、複数の区切り文字を元に、1 つの文字列から複数の個所を返すこともできます。
- $LENGTH ー 指定の区切り文字に基づいて 1 つの文字列にある構成要素の数を返します。

$PIECE 関数独自の重要な機能として、専用の区切り文字 (例えば“^”) で区切って記述した複数の部分文字列を単一の文字列として使用できます。長い文字列はレコードとして、部分文字列はそのフィールドとして動作します。

$PIECE の構文は以下のとおりです。

ObjectScript

```
WRITE $PIECE("ListString","QuotedDelimiter",ItemNumber)
```

ListString は使用されている完全なレコードを含む、引用符で囲まれた文字列です。QuotedDelimiter は指定の区切り文字で、引用符内に記述する必要があります。 ItemNumber は、返されるように指定された部分文字列です。例えば以下の構文で、スペースで区切られた以下のリストで 2 番目のアイテムを表示します。

ObjectScript

```
WRITE $PIECE("Kennedy Johnson Nixon"," ",2)
```

これは、“Johnson”を返します。

また、リストから複数の値を返すこともできます。以下はその構文です。

ObjectScript

```
WRITE $PIECE("Nixon***Ford***Carter***Reagan","***",1,3)
```

これは、“Nixon***Ford***Carter”を返します。この値は、実際の部分文字列を参照する必要があります。また、3 番目の引数 (ここでは 1) は、4 番目の引数 (ここでは 3) よりも小さい値になる必要があることに注意してください。

以下のリストのように、区切り文字は任意に決めることができます。

ObjectScript

```
SET x = $PIECE("Reagan,Bush,Clinton,Bush,Obama",",",3)
SET y = $PIECE("Reagan,Bush,Clinton,Bush,Obama","Bush",2)
WRITE x,!,y
```

これは、以下を返します。

```
Clinton
,Clinton,
```

最初のコードの区切り文字はコンマです。2 番目は、文字列“Bush”が区切り文字であるため、返される文字列はコンマを含みます。区切り文字を正確に理解するために、次のセクションではリストに関連する関数を説明しています。

12.2.1 高度な $PIECE 関数

要素値を区切ってリストに記述する $PIECE を呼び出すことで、リストの中で固有の項目に部分文字列を配置し、他の部分は空として、十分な数の項目をリストに追加できます。例えば、あるコードで、リストの 1 番目、4 番目、そして 20 番目に項目を設定するとします。

ObjectScript

```
SET $PIECE(Alphalist, "^", 1) = "a"
WRITE "First, the length of the list is ",$LENGTH(Alphalist,"^"),".",!
SET $PIECE(Alphalist, "^", 4) = "d"
WRITE "Then, the length of the list is ",$LENGTH(Alphalist,"^"),".",!
SET $PIECE(Alphalist, "^", 20) = "t"
WRITE "Finally, the length of the list is ",$LENGTH(Alphalist,"^"),".",!
```

この $LENGTH 関数は、1、4、20 の値を順番に返します。その長さに相当する必要な数のアイテムが、このコードで作成されているからです。ただし、アイテム 2、3、そして 5 から 19 には、値が設定されていません。そのため、これらの値を表示させようとしても、何も表示されません。

区切り文字列アイテムは、区切り文字列も含むことができます。このようなサブリストから値を取得するには、以下のように $PIECE 関数の呼び出しを入れ子にします。

ObjectScript

```
SET $PIECE(Powers, "^", 1) = "1::1::1::1::1"
SET $PIECE(Powers, "^", 2) = "2::4::8::16::32"
SET $PIECE(Powers, "^", 3) = "3::9::27::81::243"
WRITE Powers,!
WRITE $PIECE( $PIECE(Powers, "^", 2), "::", 3)
```

このコードは 2 行の出力を返します。1 行目は文字列 Powers で、すべての区切り文字を含みます。2 行目は 8 で、これは Powers の 2 番目の要素に含まれるサブリストの 3 番目の要素の値です。(Powers リストで、n 番目のアイテムは、2 を 1 乗から 5 乗までしたサブリストです。したがって、例えばサブリストの最初のアイテムは、n の 1 乗となります)。

詳細は、″ObjectScript リファレンス″ で ″$PIECE″ のリファレンス・ページを参照してください。

12.3 リスト構造文字列演算

ObjectScript は、“リスト”という特別な文字列の種類を定義します。これは、要素と呼ぶ部分文字列のエンコード・リストから構成されます。これらの InterSystems IRIS リストは、以下のリスト関数のみを使用して処理できます。

- リスト作成：
  - $LISTBUILD は、パラメータ値として各要素を指定することでリストを作成します。
  - $LISTFROMSTRING は、デリミタを含む文字列を指定することでリストを作成します。この関数はデリミタを使用して、文字列を要素に分割します。
  - $LIST は、既存のリストからその一部を抽出してリストを作成します。
- リスト・データ取得：
  - $LIST は、リスト要素の値をその位置別に返します。リストの先頭または末尾から位置をカウントできます。
  - $LISTNEXT は、リストの先頭から順番にリスト要素の値を返します。$LIST および $LISTNEXT はともにリストから要素を順番に返すために使用されますが、大量のリスト要素を返す場合は、$LISTNEXT の方が大幅に高速な処理が可能です。
  - $LISTGET は、リスト要素の値を位置別に返すか、または既定値を返します。
  - $LISTTOSTRING は、リストの要素の値すべてを区切り文字列として返します。
- リストの操作：
  - SET $LIST は、リストの要素を挿入、更新、または削除します。SET $LIST は、リストの要素または要素の範囲を 1 つ以上の値で置換します。SET $LIST はリストの要素を複数の要素で置換できるため、これを使用してリストに要素を挿入できます。SET $LIST はリストの要素を NULL 文字列で置換できるため、これを使用してリストの要素または要素の範囲を削除できます。
- リストの評価: :
  - $LISTVALID は、文字列がリストに有効かどうかを判断します。
  - $LISTLENGTH は、リスト内の要素の数を判断します。
  - $LISTDATA は、指定のリスト要素がデータを含んでいるかどうか判断します。
  - $LISTFIND は、指定した値がリストにあるかどうかを判断して、リストの位置を返します。
  - $LISTSAME は、2 つのリストが同一かどうか判断します。

リストはエンコードされた文字列であるため、InterSystems IRIS ではリストの扱いが標準の文字列の場合とは多少異なります。したがって、リストでは標準の文字列関数を使用しないようにします。さらに、ほとんどのリスト関数は、標準の文字列で使用すると <LIST> エラーを発生します。

以下のプロシージャは、さまざまなリスト関数の使用法を示しています。

ObjectScript

```
ListTest() PUBLIC {
    // set values for list elements
    SET Addr="One Memorial Drive"
    SET City="Cambridge"
    SET State="MA"
    SET Zip="02142"

    // create list
    SET Mail = $LISTBUILD(Addr,City,State,Zip)

    // get user input
    READ "Enter a string: ",input,!,!

    // if user input is part of the list, print the list's content
    IF $LISTFIND(Mail,input) {
        FOR i=1:1:$LISTLENGTH(Mail) {
            WRITE $LIST(Mail,i),!
        }
     }
}
```

このプロシージャは、リストの重要な機能を示しています。

- $LISTFIND は、テストされた値がリスト項目に一致する場合にのみ 1 (True) を返します。
- $LISTFIND と $LISTLENGTH は式で使用します。

リスト関数の詳細は、"ObjectScript リファレンス" の関連リファレンス・ページを参照してください。

12.3.1 スパース・リストおよびサブリスト

リストの指定された位置に要素値を追加する関数では、適正位置に値を配置するために十分な数のリスト要素を追加します。以下はその例です。

ObjectScript

```
SET $LIST(Alphalist,1)="a"
SET $LIST(Alphalist,20)="t"
WRITE $LISTLENGTH(Alphalist)
```

この例にある 2 番目の $LIST で 20 番目のリスト要素を作成しているので、$LISTLENGTH は値 20 を返しますが、2 ～ 19 番目の要素には値が設定されていません。そのため、これらの値を表示させようとすると、<NULL VALUE> エラーが発生します。$LISTGET を使用して、このエラーを回避できます。

List の要素自体をリストとすることもできます。このようなサブリストから値を取得するには、以下のように $LIST 関数呼び出しを入れ子にします。

ObjectScript

```
SET $LIST(Powers,2)=$LISTBUILD(2,4,8,16,32)
WRITE $LIST($LIST(Powers,2),5)
```

このコードは、Powers リストの 2 番目の要素に含まれるサブリストの 5 番目の要素値、32 を返します。(Powers リストで 2 番目のアイテムは、2 を 1 乗から 5 乗までしたサブリストです。したがって、例えばサブリストの最初のアイテムは、2 の 1 乗になります)。

12.4 リストと区切り文字列の比較

12.4.1 リストの利点

- リストは特定の区切り文字を必要としません。$PIECE 関数では複数のデータ項目を含む文字列を管理できますが、これは専用のデリミタとしての文字 (または文字列) の設定にもよります。区切り文字を使用する場合は、データ項目の中にデータとして区切り文字が存在すると、データの区切り位置が混乱する可能性が常に存在しています。リストは区切り文字による混乱を避けるうえで効果的です。したがって、あらゆる文字または文字の組み合わせをデータとして入力できます。
- データの要素は、$PIECE を使用して区切り文字列から取得するよりも、$LIST または $LISTNEXT を使用してリストから取得する方が高速です。データを順番に取得する場合は、$LIST よりも $LISTNEXT の方が大幅に高速ですが、双方とも $PIECE よりはきわめて高速です。

12.4.2 区切り文字列の利点

- 区切り文字列では、$FIND 関数を使用して、より柔軟なデータ内容の検索ができます。$LISTFIND は、完全一致を必要とするので、リストの部分一致文字列を検索できません。したがって、上記の例では、アドレス “One Memorial

Drive”が文字“One”で始まっているにもかかわらず、$LISTFIND を使用してメール・リストの文字列“One”を検索しても 0 (失敗を示す) が返されます。

- 区切り文字列は標準文字列であるので、標準文字列関数をすべて使用できます。InterSystems IRIS リストはエンコードされた文字列なので、InterSystems IRIS リストでは $List 関数のみが使用できます。

13
ロック管理

LOCK コマンドを使用すれば、プロセスはロックの適用と解放を行うことができます。ロックは、グローバル変数などのデータ・リソースへのアクセスを制御します。アクセス制御は規約に従って機能します。ロックおよび対応する変数は同じ名前となることがありますが (通常は同一)、互いに独立しています。ロックの変更によって同一名の変数が影響を受けることはなく、変数の変更によっても同一名のロックが影響を受けることはありません。

ロック自体は、他のプロセスが関連データを修正しないようにしません。InterSystems IRIS® Data Platform が一方的なロックを強制しないからです。規約では、互いに競合するプロセスが、すべて同じ変数にロック設定を実行していることが必要とされています。

ロックはローカル (現在のプロセスによってのみアクセス可能) またはグローバル (すべてのプロセスによってアクセス可能) にすることができます。ロックの名前付け規約はローカル変数およびグローバル変数の名前付け規約と同じです。

ロックは、それをロックしたプロセスによってアンロックされるまで有効です。システム管理者によってアンロックされるか、またはプロセスの終了時に自動的にアンロックされます。

この章では、以下の項目について説明します。

- 管理ポータル・ロック・テーブル – システム全体で保持されたすべてのロックと、保持されたロックの解除を待機するすべてのロック要求を表示します。ロック・テーブルは、保持されたロックの解除にも使用できます。
- ^LOCKTAB ユーティリティ – ロック・テーブルと同じ情報を返します。
- 待機ロック要求 – InterSystems IRIS において保持されたロックの解除を待機するロック要求がキューに格納される方法を説明します。
- デッドロックの回避 (ロック要求の相互ブロック)

ロック方法の開発についての詳細は、"ロックと並行処理の制御" の文書を参照してください。

13.1 システム全体での現在のロックの管理

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、有効なロックのすべておよびそれらをロックしているプロセスと、すべての待機ロック要求が記録されています。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、または選択されたロックを削除することができます。また、%SYS.LockQuery クラスを使用して、ロック・テーブルの情報を読み込むこともできます。%SYS ネームスペースから SYS.Lock クラスを使用して、ロック・テーブルを管理することもできます。

13.1.1 ロック・テーブルを使用したロックの表示

管理ポータルを使用すれば、システム全体で現在保持または要求 (待機) されているすべてのロックを表示できます。管理ポータルで、[**システム処理**]→[**ロック**]→[**ロック表示**] の順に選択します。[**ロック表示**] ウィンドウには、ディレクトリごとのアルファベット順で ([**ディレクトリ**])、なおかつ各ディレクトリ内にはロック名による照合順で([**リファレンス**])、ロック (およびロック要求) の一覧が表示されます。各ロックはプロセス ID ([**所有者**]) で特定され、プロセスの作成時にオペレーティング・システムにより付与されるユーザ名 (OS User Name) を表示し、[**モードカウント**] (ロック・モードおよびロック・インクリメント・カウント) を有します。ロックおよびロック要求の最新リストを表示させるためには、[**再表示**] アイコンを使用する必要があります。このインタフェースの詳細は、"監視ガイド" の "管理ポータルを使用した InterSystems IRIS の監視" の章にある "ロックの監視" を参照してください。

[ModeCount] は、特定の [**リファレンス**] についての特定の [**所有者**] プロセスによって、保持されたロックを指定することができます。以下は、保持されたロックに対する [ModeCount] 値の例です。

ModeCount 値	意味
Exclusive	排他ロック、非エスカレート (`LOCK +^a(1)`)
Shared	共有ロック、非エスカレート (`LOCK +^a(1)#"S"`)
Exclusive_e	排他ロック、エスカレート (`LOCK +^a(1)#"E"`)
Shared_e	共有ロック、エスカレート (`LOCK +^a(1)#"SE"`)
Exclusive–>Delock	ロック解除状態にある排他ロック。ロックはアンロックされていますが、現在のトランザクションの終了までロックの解除が延期されます。これは、標準のアンロック (`LOCK -^a(1)`) または遅延アンロック `LOCK -^a(1)#"D"`) により発生できます。
Exclusive,Shared	共有ロックと排他ロックの両方 (いずれの順序でも適用)。エスカレート・ロックも指定可能です。例 :Exclusive_e,Shared_e
Exclusive/n	増分排他ロック (`LOCK +^a(1)` が n 回発行される)。ロック・カウントが 1 の場合、カウントは表示されません (ただし、以下参照のこと)。増分共有ロックも指定可能です。例 :Shared/2
Exclusive/n–>Delock	ロック解除状態にあるインクリメンタル排他ロック。ロックのインクリメントはすべてアンロックされていますが、現在のトランザクションの終了までロックの解除が延期されます。トランザクション内で、個々のインクリメントをアンロックすることで、これらのインクリメントはすぐに解除されます。ロックは、ロック・カウントが 1 のときにアンロックが発行されるまで、ロック解除状態になりません。この ModeCount 値 (ロック解除状態のインクリメンタル・ロック) は、それより前のすべてのロックが単独の操作 (引数なしの LOCK コマンドまたはロック・オペレーション・インジケータ (`LOCK ^xyz(1)`) のないロックのどちらか) によってアンロックされたときに発生します。
Exclusive/1+1e	2 つの排他ロック。一方は非エスカレートで、もう一方はエスカレートです。インクリメント・カウントはこれら 2 種類の排他ロックに対して別々に維持します。共有ロックも指定可能です。例 : Shared/1+1e
Exclusive/n,Shared/m	共有ロックと排他ロックの両方。ともに整数インクリメントとなります。

保持されたロック [ModeCount] は、インクリメントの有無にかかわらず、共有または排他、エスカレートまたは非エスカレートのロックのあらゆる組み合わせを表すことができます。排他ロックまたは共有ロック (エスカレートまたは非エスカレート) はロック解除状態にすることができます。

[ModeCount] によって、`WaitExclusiveExact` などのロック待機プロセスを指定することが可能です。以下は、待機ロック要求に対する [ModeCount] の値です。

ModeCount 値	意味
WaitSharedExact	保持または事前に要求されたまったくの同一ロックに対する共有ロックの待機 :`LOCK +^a(1,2)#"S"` はロック `^a(1,2)` に対する待機となります。
WaitExclusiveExact	保持または事前に要求されたまったくの同一ロックに対する排他ロックの待機 :`LOCK +^a(1,2)` はロック `^a(1,2)` に対する待機となります。
WaitSharedParent	保持または事前に要求されたロックの親に対する共有ロックの待機 :`LOCK +^a(1)#"S"` はロック `^a(1,2)` に対する待機となります。
WaitExclusiveParent	保持または事前に要求されたロックの親に対する排他ロックの待機 :`LOCK +^a(1)` はロック `^a(1,2)` に対する待機となります。
WaitSharedChild	保持または事前に要求されたロックの子に対する共有ロックの待機 :`LOCK +^a(1,2)#"S"` はロック `^a(1)` に対する待機となります。
WaitExclusiveChild	保持または事前に要求されたロックの子に対する排他ロックの待機 :`LOCK +^a(1,2)` はロック `^a(1)` に対する待機となります。

[ModeCount] によって、ロック要求をブロックしているロック (またはロック要求) を指定します。これは必ずしも [**リファレンス**] と同じではなく、ロック・キュー先頭で現在保持されているロックを指定します。このロック・キューには当ロック要求が待機しています。[**リファレンス**] は必ずしも、このロック要求を即座にブロックする被要求ロックを指定するものではありません。

[ModeCount] は、特定の [**リファレンス**] についての特定の [**所有者**] プロセスに対するその他のロック・ステータス値を指定することができます。以下は、それらのその他の [ModeCount] ステータス値です。

ModeCount 値	意味
LockPending	排他ロックの保留中です。サーバが排他ロックの付与プロセス中にある場合に、このステータスが発生することがあります。ロック・ペンディングの状態のロックは削除できません。
SharePending	共有ロックの保留中です。サーバが共有ロックの付与プロセス中にある場合に、このステータスが発生することがあります。ロック・ペンディングの状態のロックは削除できません。
DelockPending	アンロックの保留中です。サーバが保持ロックのアンロック・プロセス中にある場合に、このステータスが発生することがあります。ロック・ペンディングの状態のロックは削除できません。
Lost	ネットワークがリセットされたため、ロックが失われました。

[**所有者のルーチン情報を表示**] を選択して [**ルーチン**] 列を有効にします。これにより、所有者のプロセスが実行しているルーチンの名前が、そのルーチンで実行されている現在の行の番号が先頭に追加されて表示されます。

[SQL オプションを表示] を選択し、[ネームスペースのSQLテーブル名を表示] リストからネームスペースを選択して [SQL テーブル名] 列を有効にします。この列には、選択したネームスペースにある各プロセスに関連付けられた SQL テーブルの名前が表示されます。プロセスが SQL テーブルと関連付けられていない場合、この列の値は空になります。

[ロック表示] ウィンドウを使用して、ロックを削除することはできません。

13.1.2 ロック・テーブルを使用したロックの削除

システム上で現在保持されているロックを削除するには、管理ポータルに移動して、[システム処理]→[ロック]→[ロックの管理] の順に選択します。目的のプロセス ([所有者]) に対して、[削除] または [プロセスが保持するすべてのロックを削除] のいずれかをクリックします。

ロックの削除により、全形式の該当ロック (すべてのインクリメント・レベルのロック、すべての排他、排他エスカレート、および共有バージョンのロック) が解放となります。1 つの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即座に適用となります。

また、SYS.Lock.DeleteOneLock() および SYS.Lock.DeleteAllLocks() メソッドを使用して、ロックを削除することもできます。

ロックを削除するには、WRITE 権限が必要になります。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。

13.2 ^LOCKTAB ユーティリティ

また、%SYS ネームスペースから InterSystems IRIS ^LOCKTAB ユーティリティを使用して、ロックを表示および削除することもできます。^LOCKTAB は、以下の形式のいずれかで実行できます。

- DO ^LOCKTAB : ロックを表示および削除できます。個々のロックの削除、指定のプロセスが所有しているすべてのロックの削除、またはシステム上のすべてのロックの削除を行う文字コード・コマンドが用意されています。
- DO View^LOCKTAB : ロックを表示できます。ロックを削除するオプションは用意されていません。

これらのユーティリティ名は、大文字と小文字が区別されることに注意してください。

以下のターミナル・セッション例は、^LOCKTAB が現在のロックを表示する方法を示しています。

```
%SYS>DO ^LOCKTAB

                              Node Name: MYCOMPUTER
                  LOCK table entries at 07:22AM  01/13/2018
                16767056 bytes usable, 16774512 bytes available.

Entry Process     X#   S# Flg   W# Item Locked
   1) 4900         1              ^["^^c:\intersystems\iris\mgr\"]%SYS("CSP","Daemon")
   2) 4856         1              ^["^^c:\intersystems\iris\mgr\"]ISC.LMFMON("License Monitor")
   3) 5016         1              ^["^^c:\intersystems\iris\mgr\"]ISC.Monitor.System
   4) 5024         1              ^["^^c:\intersystems\iris\mgr\"]TASKMGR
   5) 6796         1              ^["^^c:\intersystems\iris\mgr\user\"]a(1)
   6) 6796        1e              ^["^^c:\intersystems\iris\mgr\user\"]a(1,1)
   7) 6796              2        1 ^["^^c:\intersystems\iris\mgr\user\"]b(1)Waiters: 3120(XC)
   8) 3120         2              ^["^^c:\intersystems\iris\mgr\user\"]c(1)
   9) 2024         1    1         ^["^^c:\intersystems\iris\mgr\user\"]d(1)

Command=>
```

^LOCKTAB の表示では、保持された排他ロックを X# 列がリストして、保持された共有ロックを S# 列がリストします。X# または S# の数字は、ロック・インクリメント・カウントを示します。“e” 接尾語は、ロックがエスカレートとして定義されていることを示します。“D” 接尾語は、ロックがロック解除状態であることを示します。ロックはアンロックされていますが、現在のトランザクションの終了まで別のプロセスが使用することはできません。W# 列では待機ロック要求の数をリストします。

上記の表示で示すように、process 6796 は増分共有ロック ^b(1) を保持しています。Process 3120 はこのロックを待つロック要求を 1 つ有しています。このロック要求は、^b(1) の子 (C) に対する排他 (X) ロックのためにあります。

このユーティリティのヘルプを表示するには、Command=> プロンプトで疑問符 (?) を入力します。これには、この表示の解読方法およびロックを削除する文字コード・コマンド (ある場合) についての詳細が含まれています。

注釈　　Flg 列の値で示されるように、ロック保留状態にあるロックは削除できません。

^LOCKTAB ユーティリティを終了するには Q を入力します。

13.3 待機ロック要求

あるプロセスが排他ロックを保持する場合、別のプロセスが同じロック、または保持されたロックの上層や下層レベルのノード上のロックを取得しようとすると、待機状態が発生します。添え字付きグローバル (配列ノード) をロックする場合、以下のように、ユーザがロックするものと他のプロセスでロック可能なものとの区別が重要になります。

- ユーザがロックするもの:ユーザは指定したノードに対してのみ明示的なロックを有します。その上層や下層のレベルのノードに対してはロックを有しません。例えば、^student(1,2) をロックする場合、ユーザは ^student(1,2) に対してのみ明示的なロックを有します。上層レベルのノード (^student(1) など) の解放によってこのノードを解放することはできません。なぜなら、上層レベルのノードに対して明示的なロックを有していないためです。もちろん、ユーザは任意の順序で上層または下層のノードを明示的にロックすることができます。
- 他プロセスが何をロックできるか:ユーザがノードをロックすることにより、他のプロセスがその同一ノード、または上層や下層レベルのノード (該当ノードの親や子) をロックできません。親の ^student(1) をロックすることはできません。なぜなら、これをロックすると、子の ^student(1,2) (プロセスによって既に明示的にロックされている) も暗示的にロックされるためです。子の ^student(1,2,3) をロックすることはできません。なぜなら、プロセスが既に親の ^student(1,2) をロックしているためです。それらの他のプロセスは、指定の順序によりロック・キューにおいて待機となります。これらはロック・テーブルにリストされ、キュー内において他に優先することを指定された最上層レベルのノードで待機となります。これはロックされたノード、またはロックを待つノードとなる場合があります。

例えば以下のようになります。

1. プロセス A は ^student(1,2) をロックします。
2. プロセス B は ^student(1) のロックを試行しますが、禁じられます。これは、プロセス B が ^student(1) をロックすると、^student(1,2) も (暗示的に) ロックされるためです。ただし、プロセス A は ^student(1,2) に対するロックを保持しています。ロック・テーブルには、これが WaitExclusiveParent ^student(1,2) としてリストされます。
3. プロセス C は ^student(1,2,3) のロックを試行しますが、禁じられます。ロック・テーブルには、これが WaitExclusiveParent ^student(1,2) としてリストされます。プロセス A は ^student(1,2) に対するロックを保持しており、そのため ^student(1,2,3) に対する暗示的なロックを保持しています。ただし、プロセス C はキュー内でプロセス B よりも下層となるため、プロセス B が ^student(1) をロックして解放するのをプロセス C は待つ必要があります。
4. プロセス A は ^student(1,2,3) をロックします。待機ロックはそのままで変更ありません。
5. プロセス A は ^student(1) をロックします。待機ロックは以下のように変化します。
   - プロセス B は WaitExclusiveExact ^student(1) としてリストされます。プロセス B は、プロセス A が保持するものと同一のロック (^student(1)) をロックすることを待ちます。
   - プロセス C は WaitExclusiveChild ^student(1) としてリストされます。プロセス C はプロセス B よりもキュー内で下層となるため、要求されたロックをプロセス B がロックして解放するのを待ちます。そのときには、プロセ

ス C がプロセス B ロックの子をロックできるようになります。同様に、プロセス A が ^student(1) を解放するのをプロセス B は待ちます。

6. プロセス A は ^student(1) をアンロックします。待機ロックは WaitExclusiveParent ^student(1,2) に戻ります。(手順 2 および 3 と同一)。
7. プロセス A は ^student(1.2) をアンロックします。待機ロックは WaitExclusiveParent ^student(1,2,3) に変化します。プロセス B は ^student(1) (現在のプロセス A ロック ^student(1,2,3) の親) のロックを待ちます。プロセス C はプロセス B が ^student(1) (プロセス C により要求された ^student(1,2,3) ロックの親) をロックしてからアンロックするのを待ちます。
8. プロセス A は ^student(1,2,3) をアンロックします。プロセス B は ^student(1) をロックします。プロセス C は現在、プロセス B によって禁じられています。プロセス C は WaitExclusiveChild ^student(1) としてリストされます。プロセス C は ^student(1,2,3) (現在のプロセス B ロックの子) のロックを待ちます。

13.3.1 配列ノードに対するロック要求のキュー

配列ロックに対する InterSystems IRIS の基本的なキュー・アルゴリズムでは、リソースが直接競合していない場合でも、同一リソースに対するロック要求は受信した順で厳密にキューに置かれます。この様子を以下の例に示します。ここでは、同じグローバル配列上の 3 つのロックが、以下に示す順序で別々のプロセスによって要求されます。

```
Process A: LOCK ^x(1,1)
Process B: LOCK ^x(1)
Process C: LOCK ^x(1,2)
```

これらの要求のステータスは以下のとおりです。

- プロセス A は ^x(1,1) に対するロックを保持しています。
- プロセス B は、プロセス A が ^x(1,1) に対するロックを解除するまで ^x(1) をロックできません。
- プロセス C もブロックされていますが、プロセス A のロックに伴うブロックではありません。プロセス B が ^x(1) の明示的なロックと、^x(1,2) の暗示的なロックを待機しており、これによりプロセス C がブロックされます。

シーケンスでロックを保持しているプロセスの後にある次のジョブを高速化するために、このような手法が採用されています。キューの中でプロセス C がプロセス B を飛び越えることができるようにすると、プロセス C は速く実行できますが、プロセス B に重大な遅延が発生する可能性があります。プロセス C のようなジョブが多数存在する場合は、特に顕著になります。

親ノードに対するロックを保持しているプロセスには、そのノードの子に対して要求されたロックが直ちに付与されます。これは、要求は受信した順で処理されるという一般的なルールの例外です。例えば、前の例の拡張を以下で考えます。

```
Process A: LOCK ^x(1,1)
Process B: LOCK ^x(1)
Process C: LOCK ^x(1,2)
Process A: LOCK ^x(1,2)
```

この例では、プロセス A には、プロセス B とプロセス C よりも前に、^x(1,2) に対して要求されたロックが直ちに付与されています。プロセス A が既に ^x(1,1) に対するロックを保持しているからです。

注釈　このプロセスのキューイング・アルゴリズムは、添え字を持つすべてのロック要求に適用されます。ただし、添え字のないロック要求 (LOCK +^x) と添え字のあるロック要求 (LOCK +^x(1,1)) の両方が待機中である場合、添え字のないロック (LOCK ^x など) の解除は特殊なケースです。この場合、どのロック要求が付与されるかは予測できず、プロセスのキューイングに従わない場合があります。

13.3.2 ECP ローカルおよびリモート・ロック要求

ロックを解除する際、ECP クライアントは、パフォーマンスを向上させるためにその他のシステム上の待機に優先してローカルの待機にロックを付与できます。リモート・ロック待機で許容できないほどの遅延が発生しないように、これが発生できる回数は制限されています。

13.4 デッドロックの回避

既存の共有ロックを保持している際の (+) 排他ロックの要求は、潜在的な危険を伴います。これは、"デッドロック"として知られる状況を引き起こす可能性があるからです。この状況は、既に他のプロセスによって共有ロックとしてロックされているロック名に対して、2 つのプロセスがそれぞれ排他ロックを要求するときに発生します。その結果、各プロセスは、他のプロセスが既存の共有ロックを解除するまで待機する間、停止します。

次の例では、これがどのように行われるかについて説明します (数字は処理の順番を示します)。

プロセス A	プロセス B
1.LOCK ^a(1)#"S" プロセス A は共有ロックを取得します。	
	2.LOCK ^a(1)#"S" プロセス B は共有ロックを取得します。
3.LOCK +^a(1) プロセス A は排他ロックを要求し、プロセス B がその共有ロックを解除するまで待機します。	
	4.LOCK +^a(1) プロセス B は排他ロックを要求し、プロセス A がその共有ロックを解除するまで待機します。これによってデッドロックが発生します。

これは最も単純な形式のデッドロックです。デッドロックは、保持されているロックの親ノードまたは子ノード上のロックをプロセスが要求する際にも発生する可能性があります。

デッドロックを防止するには、プラス記号なしで排他ロックを要求する (そうすることで共有ロックをロック解除する) 必要があります。以下の例では、両方のプロセスが排他ロックを要求する際に、それぞれが有している以前のロックを解除することでデッドロックを防止しています (数字は操作の順序を示しています)。どのプロセスが排他ロックを取得するかに注意してください。

プロセス A	プロセス B
1.LOCK ^a(1)#"S" プロセス A は共有ロックを取得します。	
	2.LOCK ^a(1)#"S" プロセス B は共有ロックを取得します。
3.LOCK ^a(1) プロセス A は共有ロックを解除してから排他ロックを要求し、プロセス B がその共有ロックを解除するまで待機します。	
	4.LOCK ^a(1) プロセス B は共有ロックを解除してから排他ロックを要求します。プロセス A は直ちに、要求した共有ロックを取得します。プロセス B は、プロセス A がその共有ロックを解除するまで待機します。

他のデッドロック回避方法には、LOCK + および LOCK - コマンドを実行する順序に関する、厳密なプロトコルに従う方法もあります。すべてのプロセスが、この同じ順序に従っている限り、デッドロックが発生することはありません。単純なプロトコルは、すべてのプロセスにおいて照合順序でロックを適用および解除するものです。

デッドロック状況の影響を最小限に抑えるために、プラス記号のロックを使用する際に timeout 引数を指定する必要があります。例えば、LOCK +^a(1):10 オペレーションは 10 秒後にタイムアウトします。

デッドロックが発生した場合は、管理ポータルまたは ^LOCKTAB ユーティリティを使用して、問題になっているロックのいずれかを削除できます。管理ポータルから、**[ロックを管理]** ウィンドウを開き、デッドロック・プロセスに対して **[削除]** オプションを選択します。

14
トランザクション処理

トランザクションとは、動作の論理ユニットです。InterSystems IRIS® Data Platform のトランザクション処理は、データベースの論理整合性の維持に役立ちます。

例えば銀行業務で、ある口座から別の口座に送金する場合、送金元のテーブル内のフィールドから差し引いた額を、送金先のテーブルのフィールドに追加する必要があります。両方の更新で 1 つのトランザクションとなることを指定すると、両方の処理が実行されるか、両方とも実行されないかのいずれかになります。つまり、一方の処理のみを実行できません。

アプリケーション内で、単一の SQL INSERT、UPDATE、DELETE 文、あるいは単一のグローバル SET や KILL は、それ自体で完全なトランザクションを構築しないことがあります。このような場合、トランザクション処理コマンドを使用して、完全なトランザクションを構成する一連の操作を定義します。1 つのコマンドでトランザクションの開始を示し、一連の多数のコマンドを置いた後に、他のコマンドでトランザクションの終了を示します。

通常、トランザクションは全体的に実行されます。プログラム・エラーやシステムの動作不良が原因で不完全なトランザクションになった場合、部分的に完了したトランザクションはロールバックされます。

アプリケーション開発者は、アプリケーション内でトランザクション・ロールバックを処理する必要があります。InterSystems IRIS は、リカバリ、HALT や ResJob の実行中など、システム障害時やさまざまな状況でトランザクション・ロールバックを自動的に処理します。

LogRollback 構成オプションが設定されている場合、InterSystems IRIS は、**messages.log** ファイルにロールバックを記録します。管理ポータルの **[システム処理]→[システムログ]→[メッセージログ]** オプションを使用して、**messages.log** を表示できます。

14.1 アプリケーションでのトランザクション管理

InterSystems IRIS では、以下のいずれかを使用して、アプリケーションのトランザクションを定義します。

- マクロ・ソース・ルーチンの SQL 文
- ObjectScript コマンド

両方の手法が機能します。SQL INSERT、UPDATE、DELETE 文あるいは ObjectScript の SET コマンドと KILL コマンドのいずれかを使用して、トランザクションを構築するデータベースを変更するかどうかは関係ありません。

14.1.1 トランザクション・コマンド

InterSystems IRIS は、ANSI SQL 操作 COMMIT WORK と ROLLBACK WORK をサポートしています (InterSystems SQL では、キーワード WORK はオプションとなります)。また、InterSystems SQL 拡張機能 SET TRANSACTION、START

TRANSACTION、SAVEPOINT、および %INTRANS もサポートしています。また、InterSystems IRIS には、M 言語 Type A 標準のトランザクション・コマンドのうち、実装されているものもあります。

以下のテーブルは、これらの SQL と ObjectScript コマンドの概要です。

テーブル 14-1: トランザクション・コマンド

SQL コマンド	ObjectScript コマンド	定義
SET TRANSACTION		トランザクションを開始せずに、トランザクション・パラメータを設定します。
START TRANSACTION	TSTART	トランザクションの開始を表します。
%INTRANS	$TLEVEL	トランザクションが現在進行中かどうかを検出します。 ・ %INTRANS によって使用される <0 は、トランザクション処理中を意味しますが、ジャーナリングは無効です。$TLEVEL では使用されません。 ・ 0 = トランザクション処理中ではありません。 ・ >0 は、トランザクション処理中を意味します。
SAVEPOINT		トランザクション内のポイントを指定します。セーブポイントまでの部分的なロール・バックに使用できます。
COMMIT	TCOMMIT	トランザクションの正常終了を示します。
ROLLBACK	TROLLBACK	トランザクションの失敗を示します。トランザクションの開始後に実行されたすべてのデータベース更新は、ロールバックされるか、元の状態に戻ります。

これらの ObjectScript と SQL の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。

14.1.2 トランザクションでの LOCK の使用

複数のプロセスがアクセスする可能性のあるグローバルにアクセスする際、常にそのグローバルで LOCK コマンドを使用して、データベースの整合性を保護する必要があります。グローバル変数に対応したロックを発行し、グローバルの値を変更し、次にロックをアンロックします。LOCK コマンドは、指定したロックのロックとアンロックの両方に使用されます。グローバルの値を変更しようとするその他のプロセスはロックを要求し、そのロックは最初のプロセスがロックを解除するまで待機します。

トランザクションでロックを使用する際、以下の 3 つの考慮事項があります。

- ロック/アンロック操作は、ロールバックを実行しません。
- トランザクション内で、プロセスによって保持されたロックをアンロックすると、以下の 2 つのうちどちらかが発生する場合があります。
  - ロックがすぐにアンロックされる。ロックは直後に別のプロセスによって取得可能になります。
  - ロックがロック解除状態に置かれる。ロックはアンロックされていますが、現在のトランザクションの終了まで別のプロセスによって取得可能な状態になりません。

ロックがロック解除状態にある場合は、InterSystems IRIS はトランザクションがコミットされるかロールバックされるまで、アンロックを延期します。トランザクション内では、ロックはアンロック状態で表示され、同じ値の後続のロックが許可されます。しかし、トランザクションの範囲外では、ロックはロックされたままになります。詳細は、このドキュメントの“ロック管理”の章を参照してください。

- タイムアウトになるロック操作は、$TEST を設定します。トランザクション時に $TEST で設定される値はロールバックしません。

14.1.3 トランザクションでの $INCREMENT と $SEQUENCE の使用法

$INCREMENT または $SEQUENCE 関数呼び出しは、トランザクションの一部ではありません。したがって、トランザクション・ロールバックの一部としてロールバックされません。これらの関数を使用すると、LOCK コマンドを使用せずにインデックス値を取得できます。この機能は、トランザクションの継続中にカウンタ・グローバルをロックしたくない場合に役立ちます。

$INCREMENT は、1 つ以上のプロセスからインクリメント要求を受け取った順に個々の整数値を割り当てます。$SEQUENCE は複数プロセスに対して高速手段を提供するものであり、整数値のシーケンス (範囲) を各増分プロセスに割り当てることにより、同一のグローバル変数に対して一意 (重複なし) の整数を取得します。

注釈　$INCREMENT は、トランザクション内の 1 プロセスによってインクリメントされます。また、トランザクションの処理中、並行するトランザクションの別のプロセスによってもインクリメントされる場合があります。最初のトランザクションをロールバックすると、インクリメントが“スキップされ”、“使用されない”番号が発生する可能性があります。

14.1.4 アプリケーション内でのトランザクション・ロールバック

トランザクション中にエラーに遭遇した場合、以下の 3 つの方法でロールバックできます。

- SQL ロールバック・コマンド ROLLBACK WORK の発行
- ObjectScript ロールバック・コマンド TROLLBACK の発行
- %ETN の呼び出し

注釈　トランザクションをロールバックする場合、既定クラスの IDKey はデクリメントされません。IDKey の値は、$INCREMENT 関数によって自動的に変更されます。

14.1.4.1 SQL あるいは ObjectScript ロールバック・コマンドの発行

アプリケーション開発者は、2 種類のロールバック・コマンドを使用して、トランザクションが失敗した場所を指定し、不完全なトランザクションを自動的にロールバックします。

- マクロ・ソース・ルーチンで、##sql(ROLLBACK WORK) を使用
- マクロあるいは中間ソース・コードで、ObjectScript TROLLBACK コマンドを使用

以下の例のように、ロールバック・コマンドは、エラー・トラップと併用する必要があります。

ObjectScript

```
ROU         ##sql(START TRANSACTION) set $ZT="ERROR"
             SET ^ZGLO(1)=100
             SET ^ZGLO=error
             SET ^ZGLO(1,1)=200
             ##sql(COMMIT WORK) Write !,"Transaction Committed" Quit
ERROR         ##sql(ROLLBACK WORK)
             Write !,"Transaction failed." Quit
```

この例のコードでは、トランザクションがコミットされる前にプログラム・エラーが発生した場合、$ZT が ERROR サブルーチンを実行するように設定されています。ROU 行は、トランザクションを開始し、エラー・トラップを設定します。ROU+1 行と ROU+3 行では、グローバル ^ZGLO のノードを設定します。しかし、変数 error が未定義の場合、ROU+2 でプログラム・エラーが発生し、ROU+3 行は実行されません。プログラムは、ERROR サブルーチンを実行し、^ZGLO(1) の設定は取り消されます。ROU+2行が削除されていたら、^ZGLO は 2 回とも値を設定し、トランザクションはコミットされて、“Transaction committed”というメッセージが表示されます。

14.1.4.2 %ETN への呼び出し

ロールバック・コマンドでトランザクション・ロールバックを処理しない場合、エラー・トラップ・ユーティリティ %ETN が不完全なトランザクションを検出し、トランザクションのコミットまたはロールバックのいずれかをユーザに指示します。不完全なトランザクションをコミットすると、通常、論理データベースの整合性の低下を招くため、アプリケーション内でロールバックを処理する必要があります。

トランザクションの進行中に発生したエラーの後に %ETN を実行する場合、以下のロールバック・プロンプトが表示されます。

```
You have an open transaction.
Do you want to perform a Commit or Rollback?
Rollback =>
```

10 秒のタイムアウト時間内に何も応答がない場合、既定でロールバックされます。ジョブ起動ジョブあるいはアプリケーション・モード・ジョブでは、メッセージなしでトランザクションがロールバックされます。

%ETN 自体は、トランザクションのロールバックを引き起こしませんが、通常、%ETN は InterSystems IRIS の停止によって終了します。ObjectScript の停止時にトランザクション・ロールバックが発生し、システムが %HALT を実行して InterSystems IRIS プロセスのクリーンアップを実行します。%ETN には、BACK^%ETN というエントリ・ポイントがあり、halt ではなく quit で終了します。ルーチンが BACK^%ETN ではなく ^%ETN や FORE^%ETN を呼び出す場合、エラー処理プロセスの一部としてトランザクション・ロールバックを実行しません。

14.1.5 アプリケーション内のトランザクション処理の例

以下の例は、マクロ・ソース・ルーチンでトランザクションを処理する方法です。SQL コードでデータベース更新を実行します。以下の SQL 文は、ある口座から別の口座に資金を送金します。

ObjectScript

```
Transfer(from,to,amount)   // Transfer funds from one account to another
{
   TSTART
    &SQL(UPDATE A.Account
         SET A.Account.Balance = A.Account.Balance - :amount
         WHERE A.Account.AccountNum = :from)
    If SQLCODE TRollBack  Quit "Cannot withdraw, SQLCODE = "_SQLCODE
    &SQL(UPDATE A.Account
         SET A.Account.Balance = A.Account.Balance + :amount
         WHERE A.Account.AccountNum = :to)
    If SQLCODE TROLLBACK  QUIT "Cannot deposit, SQLCODE = "_SQLCODE
    TCOMMIT
    QUIT "Transfer succeeded"
}
```

14.2 自動トランザクション・ロールバック

以下の場合、トランザクション・ロールバックが自動的に発生します。

- InterSystems IRIS の起動 (リカバリが必要な場合)。起動されると、InterSystems IRIS はリカバリが必要なことを判定し、コンピュータ上の不完全なトランザクションをすべてロールバックします。

- HALT コマンド (現在のプロセスの場合) または ^RESJOB ユーティリティ (その他のプロセスの場合) を使用したプロセスの終了。バックグラウンド・ジョブ (非インタラクティブ・プロセス) を停止すると、進行中の現在のトランザクションに対する変更が自動的にロールバックされます。インタラクティブ・プロセスを停止すると、進行中の現在のトランザクションに対する変更をコミットするかロールバックするかを確認するプロンプトが表示されます。^RESJOB をプログラマ・モード・ユーザ・プロセスで発行すると、ユーザにメッセージが表示され、現在のトランザクションをコミットするかロールバックするか尋ねられます。

また、システム管理者は、^JOURNAL ユーティリティを実行して、クラスタ固有のデータベースで不完全なトランザクションをロールバックできます。^JOURNAL ユーティリティのメイン・メニューから `Restore Globals From Journal` オプションを選択すると、ジャーナル・ファイルがリストアされ、すべての不完全なトランザクションがロールバックされます。

14.3 トランザクション処理に関するシステム全体の問題

このセクションでは、トランザクション処理に関するさまざまなシステム全体の問題について説明します。バックアップに関する問題の詳細は、"データ整合性ガイド" の "バックアップとリストア" の章を参照してください。ECP に関する問題の詳細は、"スケーラビリティ・ガイド" の "分散キャッシュによるユーザ数に応じた水平方向の拡張" の章の付録 "ECP リカバリ・プロセス、保証、および制限" を参照してください。

14.3.1 トランザクション処理でのバックアップとジャーナリング

トランザクション処理を実装する際、以下のバックアップおよびジャーナリングの手順を検討してください。

InterSystems IRIS の各インスタンスにはジャーナルがあります。ジャーナルとは、最後のバックアップ以後、データベースに対して行われた変更を、時間順で記録している一連のファイルからなるログです。InterSystems IRIS のトランザクション処理は、データの論理的整合性を維持するために、ジャーナリングと共に機能します。

ジャーナルには、トランザクションにあるグローバルに対する SET 処理と KILL 処理が記録されます。これは、影響を受けるグローバルが保存されているデータベースのジャーナル設定には関係なく記録されます。さらに、**[グローバル・ジャーナル状態]** を "Yes" に設定したデータベースのグローバルに対するすべての SET 処理と KILL 処理も記録されます。

バックアップは、トランザクション処理中に実行できます。ただし、その結果として生ずるバックアップ・ファイルには、トランザクションの一部が含まれる場合があります。災害が発生し、バックアップからリストアする必要が生じた場合は、まずバックアップ・ファイルをリストアしてから、リストアしたデータベースのコピーにジャーナル・ファイルを適用します。ジャーナル・ファイルを適用すると、バックアップの時点から災害発生時までの間にジャーナルに記述された更新がすべてリストアされます。トランザクションの一部を完了し、コミットされていないトランザクションをロールバックすることによってデータベースのトランザクションの整合性をリストアするには、ジャーナルの適用が必要です。これは、バックアップを実行した時点では、データベースにすべてのトランザクションが記録されていない可能性があるためです。詳細は、以下を参照してください。

- "データ整合性ガイド" の "ジャーナリング" の章
- "データ整合性ガイド" の "バックアップとリストア" の章の "ジャーナルの重要性" のセクション

14.3.2 非同期エラーの通知

%SYSTEM.Process クラスの AsynchError() メソッドを使用して、ジョブが非同期エラーで割り込まれるかどうかを指定できます。

- %SYSTEM.Process.AsynchError(1) は、非同期エラーの受け入れを可能にします。
- %SYSTEM.Process.AsynchError(0) は、非同期エラーの受け入れを不可能にします。

Config.Miscellaneous クラスの AsynchError プロパティは、新規プロセスに対するシステム全体の既定として、プロセスが非同期エラーで割り込まれるかどうかを設定します。既定値は 1 で、“割り込まれる”を意味します。

特定のジョブで複数の非同期エラーが検出される場合、システムは少なくとも 1 つ非同期エラーを引き起こします。しかし、どのエラーが起こるかは判別できません。

非同期エラーには、以下が実装されています。

- <LOCKLOST> ー このジョブで一度ロックされたもののうち、いくつかがリセットされました。
- <DATALOST> ー このジョブで実行されたデータ修正のうち、いくつかはエラーとしてサーバから返されました。
- <TRANLOST> ー このジョブで開された分散型トランザクションが、サーバによって非同期的にロール・バックされました。

非同期エラーを受信しているジョブを無効にした場合でも、次に ZSync コマンドを実行すると、非同期エラーが引き起こされます。

TStart、TCommit、または LOCK 操作、およびネットワーク・グローバル参照が行われるたびに、InterSystems IRIS は保留になっている非同期エラーをチェックします。ネットワーク上の SET 操作と KILL 操作は非同期であるため、SET の生成時と非同期エラーの報告時の間に、任意の数の他の指示が挿入される場合があります。

14.4 現在のすべてのトランザクションの一時停止

%SYSTEM.Process クラスの TransactionsSuspended() メソッドを使用して、現在のプロセスの現在のトランザクションすべてを一時停止できます。これはブーリアン・メソッドです (1 = 現在のすべてのトランザクションを一時停止する、0 = 現在のすべてのトランザクションを再開する)。デフォルトは 0 です。

トランザクションが一時停止されている間、変更はトランザクション・ログに記録されないため、ロールバックすることはできません。トランザクションが一時停止された期間の前後に現在のトランザクションで行われた変更は、ロールバックすることができます。

トランザクションでグローバル変数を変更し、そのトランザクションが一時停止されている間にその変数を再度変更すると、ロールバックが試行されたときにエラーが発生する可能性があります。

ブーリアン・パラメータを指定せずに TransactionsSuspended() を呼び出すと、その設定を変更せずに現在のブーリアン設定が返されます。

15
エラー処理

エラー発生時の InterSystems IRIS® Data Platform の動作を管理することをエラー処理といいます。エラー処理では、以下の処理の 1 つまたは複数が実行されます。

- エラーの原因となる条件の修正
- エラー発生後に動作を再開させるための処理の実行
- 実行フローの変更
- エラー情報の記録

InterSystems IRIS は、次の 3 種類のエラー処理の方法をサポートしており、これらを同時に使用することもできます。

- TRY-CATCH メカニズム
- %Status エラー・コード処理
- 従来のエラー処理

重要　　ObjectScript のエラー処理には、TRY-CATCH メカニズムが推奨されます。

15.1 TRY-CATCH メカニズム

InterSystems IRIS は、エラー処理のために TRY-CATCH メカニズムをサポートします。このメカニズムを使用すると、それぞれが TRY ブロックと呼ばれる区切られたコードのブロックを作成できます。TRY ブロック中にエラーが発生した場合、TRY ブロックに関連付けられた、例外処理のコードを含む CATCH ブロックに制御が渡されます。TRY ブロックには THROW コマンドを含めることもできます。これらのコマンドはそれぞれ、TRY ブロック内から明示的に例外を発行し、実行を CATCH ブロックに移します。

最も基本的な形式でこのメカニズムを使用するには、ObjectScript コード内に TRY ブロックを含めます。このブロック内で例外が発生する場合、関連付けられた CATCH ブロック内のコードが実行されます。TRY-CATCH ブロックの形式は以下のとおりです。

```
TRY {
     protected statements
} CATCH [ErrorHandle] {
     error statements
}
further statements
```

以下はその説明です。

- TRY コマンドは、中括弧で囲まれた ObjectScript のコード文を識別します。TRY は引数を取りません。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。TRY ブロック内で例外が発生した場合、InterSystems IRIS は例外プロパティ (oref.Name、oref.Code、oref.Data、oref.Location)、$ZERROR、および $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。これは、例外のスローとして知られています。
- protected statementsは、通常の実行の一部である、ObjectScript 文です。(これらの文には、THROW コマンドへの呼び出しを含めることができます。このシナリオについては、以下のセクションで説明します。)
- CATCH コマンドは、例外ハンドラを定義します。これは、TRY 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。
- ErrorHandle 変数は、例外オブジェクトへのハンドルです。これは、InterSystems IRIS が実行時エラーに対して生成した例外オブジェクト、または THROW コマンド (以下のセクションで説明します) の呼び出しにより明示的に発行された例外オブジェクトのいずれかとなります。
- error statements は、例外が発生したときに呼び出される ObjectScript 文です。
- further statements は、例外がない場合に protected statements の実行の後に続く、または例外が発生して CATCH ブロックから制御が渡される場合に error statements の実行の後に続く、ObjectScript 文です。

protected statements の実行中のイベントに応じて、以下のイベントのいずれかが発生します。

- エラーが発生しない場合、CATCH ブロックの外側に表示される further statements で実行が続行します。
- エラーが発生する場合、制御が CATCH ブロックに渡され、error statements が実行されます。実行は、CATCH ブロックの内容によって異なります。
  - CATCH ブロックに THROW コマンドまたは GOTO コマンドが含まれている場合、指定された場所に制御が直接移動します。
  - CATCH ブロックに THROW コマンドまたは GOTO コマンドが含まれていない場合、CATCH ブロックから制御が渡され、further statements で実行が続行します。

15.1.1 TRY-CATCH と一緒に THROW を使用する

InterSystems IRIS は、実行時エラーの発生時に暗黙的な例外を発行します。明示的な例外を発行するには、THROW コマンドを使用できます。THROW コマンドは、TRY ブロックから CATCH 例外ハンドラへ実行を移します。THROW コマンドの構文は以下のとおりです。

```
THROW expression
```

expression は、例外処理のために InterSystems IRIS に用意されている %Exception.AbstractException クラスから継承するクラスのインスタンスです。%Exception.AbstractException の詳細は、以下のセクションを参照してください。

TRY/CATCH ブロックと THROW を一緒に発行する場合の形式は以下のとおりです。

```
TRY {
     protected statements
     THROW expression
     protected statements
}
CATCH exception {
     error statements
}
further statements
```

ここで、THROW コマンドは明示的に例外を発行します。TRY-CATCH ブロックの他の要素については、前のセクションで説明しています。

THROW の動作は、スローが発生する場所と THROW の引数によって異なります。

- TRY ブロック内の THROW は、CATCH ブロックに制御を渡します。
- CATCH ブロック内の 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 オブジェクトの場合、次のエラー・ハンドラは、どのタイプ (CATCH または従来のエラー処理) でもかまいません。それ以外の場合は、CATCH による例外処理が必要となります。そうしないと、<NOCATCH> エラーがスローされます。

引数付きの THROW のために、制御が CATCH ブロックに渡される場合、ErrorHandle には引数からの値が含まれています。システム・エラーのために制御が CATCH ブロックに渡される場合、ErrorHandle は %Exception.SystemException オブジェクトです。ErrorHandle が指定されていない場合、制御が CATCH ブロックに渡された理由が示されません。

例えば、以下のように 2 つの数字を除算するコードがあるとします。

```
div(num,div) public {
 TRY {
  SET ans=num/div
 } CATCH errobj {
  IF errobj.Name="<DIVIDE>" { SET ans=0 }
  ELSE { THROW errobj }
 }
 QUIT ans
}
```

0 による除算エラーが発生すると、このコードは結果として 0 を返すように特別に設計されています。この他のエラーの場合、THROW はスタック上のエラーを次のエラー・ハンドラに送信します。

15.1.2 $$$ThrowOnError および $$$ThrowStatus マクロの使用法

InterSystems IRIS は、例外処理で使用するために、マクロを提供します。これらのマクロは、呼び出されると CATCH ブロックに例外オブジェクトをスローします。

以下の例は、%Prepare() メソッドによりエラー・ステータスが返されると、$$$ThrowOnError マクロを呼び出します。

ObjectScript

```
  #include %occStatus
  TRY {
    SET myquery = "SELECT TOP 5 Name,Hipness,DOB FROM Sample.Person"
    SET tStatement = ##class(%SQL.Statement).%New()
    SET status = tStatement.%Prepare(myquery)
    $$$ThrowOnError(status)
    WRITE "%Prepare succeeded",!
    RETURN
  }
  CATCH sc {
    WRITE "In Catch block",!
    WRITE "error code: ",sc.Code,!
    WRITE "error location: ",sc.Location,!
    WRITE "error data:",$LISTGET(sc.Data,2),!
  RETURN
  }
```

以下の例は、%Prepare() メソッドにより返されたエラー・ステータスの値をテストした後に$$$ThrowStatus を呼び出します。

ObjectScript

```
#include %occStatus
TRY {
  SET myquery = "SELECT TOP 5 Name,Hipness,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET status = tStatement.%Prepare(myquery)
  IF ($System.Status.IsError(status)) {
    WRITE "%Prepare failed",!
    $$$ThrowStatus(status) }
  ELSE {WRITE "%Prepare succeeded",!
    RETURN }
}
CATCH sc {
  WRITE "In Catch block",!
  WRITE "error code: ",sc.Code,!
  WRITE "error location: ",sc.Location,!
  WRITE "error data:",$LISTGET(sc.Data,2),!
RETURN
}
```

システム指定のマクロについては、このドキュメントの“ObjectScript マクロとマクロ・プリプロセッサ”の章で説明します。

15.1.3 %Exception.SystemException と %Exception.AbstractException クラスの使用

InterSystems IRIS は、例外処理で使用するために、%Exception.SystemException クラスと %Exception.AbstractException クラスを提供します。%Exception.SystemException は %Exception.AbstractException クラスから継承しており、システム・エラーに使用されます。カスタム・エラーの場合、%Exception.AbstractException から継承するクラスを作成します。%Exception.AbstractException には、エラーの名前や、エラーが発生した場所などのプロパティが含まれています。

システム・エラーが TRY ブロック内でキャッチされると、システムは %Exception.SystemException クラスの新しいインスタンスを作成し、そのインスタンス内にエラー情報を配置します。アプリケーション・プログラマは、カスタム例外をスローするとき、オブジェクトにエラー情報を入れる責任があります。

例外オブジェクトには、以下のプロパティがあります。

- Name ー <UNDEFINED> などのエラー名。
- Code ー エラー番号
- Location ー エラーの label+offset^routine の場所
- Data ー エラーによって報告される任意の追加データ (エラーの原因となる項目の名前など)

15.1.4 TRY-CATCH を使用する際の他の考慮事項

TRY-CATCH ブロックを使用する際に発生する可能性がある状況を、以下に説明します。

15.1.4.1 TRY-CATCH ブロック内での QUIT

TRY または CATCH ブロック内の QUIT コマンドは、TRY-CATCH ブロック全体の後で、ブロックから次の文に制御を渡します。

15.1.4.2 TRY-CATCH と実行スタック

TRY ブロックは、実行スタックで新しいレベルを導入しません。つまり、NEW コマンドのアクセス範囲境界ではありません。error statements は、そのエラーと同じレベルで実行されます。この結果として、protected statements 内に DO コマンドがあり、DO のターゲットも protected statements 内にある場合には、予期しない結果が発生する可能性があります。このような場合には、$ESTACK 特殊変数が、相対的な実行レベルに関する情報を提供できます。

15.1.4.3 従来のエラー処理で TRY-CATCH を使用する

TRY-CATCH エラー処理は、実行スタックの異なるレベルで使用される $ZTRAP エラー・トラップと互換性があります。例外は、$ZTRAP を TRY 節の protected statements 内で使用できないことです。THROW コマンドでは、ユーザ定義のエラーは TRY-CATCH のみに制限されます。ZTRAP コマンドを使用したユーザ定義のエラーは、任意のタイプのエラー処理で使用できます。

15.2 %Status エラー処理

InterSystems IRIS クラス・ライブラリのメソッドの多くは、%Status データ型を使用して、成功、または失敗の情報を返します。例えば、%Persistent オブジェクトのインスタンスを保存するときに使用される %Save() メソッドは、オブジェクトが保存されたか否かを示す %Status 値を返します。

- メソッドの実行が成功すると、1 の %Status が返ります。
- メソッドの実行が失敗すると、エラー・ステータスおよび 1 つ以上のエラー・コードとテキスト・メッセージを含むエンコードされた文字列として %Status が返ります。ステータス・テキスト・メッセージはユーザのロケールの言語でローカライズされています。

%SYSTEM.Status クラス・メソッドを使用すれば、%Status の値を検査および操作できます。

InterSystems IRIS には、%Status でエンコードされた文字列をさまざまな形式で表示 (書き込み) するための複数のオプションがあります。詳細は、このドキュメントの “コマンド” の章の “表示 (書き込み) コマンド” を参照してください。

以下の例では、myquery テキストのエラーによって %Prepare が失敗します (“ZOP” は “TOP” とすべき)。このエラーは IsError() メソッドにより検出され、他の %SYSTEM.Status がエラー・コードとテキストを表示します。

ObjectScript

```
  SET myquery = "SELECT ZOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET status = tStatement.%Prepare(myquery)
  IF ($System.Status.IsError(status)) {
      WRITE "%Prepare failed",!
      DO StatusError() }
  ELSE {WRITE "%Prepare succeeded",!
        RETURN }
StatusError()
  WRITE "Error #",$System.Status.GetErrorCodes(status),!
  WRITE $System.Status.GetOneStatusText(status,1),!
  WRITE "end of error display"
  QUIT
```

下の例は上の例と同じですが、ステータス・エラーが %occStatus インクルード・ファイルの $$$ISERR() マクロによって検出される点で異なります。$$$ISERR() (およびその反転型である $$$ISOK()) は、%Status =1 であるかどうかをチェックします。エラー・コードは $$$GETERRORCODE() マクロによって返されます。

ObjectScript

```
#include %occStatus
  SET myquery = "SELECT ZOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET status = tStatement.%Prepare(myquery)
  IF $$$ISERR(status) {
      WRITE "%Prepare failed",!
      DO StatusError() }
  ELSE {WRITE "%Prepare succeeded",!
        RETURN}
StatusError()
  WRITE "Error #",$$$GETERRORCODE(status),!
  WRITE $System.Status.GetOneStatusText(status,1),!
  WRITE "end of error display"
  QUIT
```

システム指定のマクロについては、このドキュメントの“ObjectScript マクロとマクロ・プリプロセッサ”の章で説明します。

%New() など、%Status を生成しても返さないメソッドもあります。%New() は、成功時にはクラスのインスタンスに oref を返し、失敗時には NULL 文字列を返します。以下の例で示すように、%objlasterror システム変数にアクセスして、この種のメソッドのステータス値を取得することができます。

ObjectScript

```
SET session = ##class(%CSP.Session).%New()
IF session="" {
    WRITE "session oref not created",!
    WRITE "%New error is ",!,$System.Status.GetErrorText(%objlasterror),! }
ELSE {WRITE "session oref is ",session,! }
```

詳細は、%SYSTEM.Status のクラスを参照してください。

15.2.1 %Status エラーの作成

Error() メソッドを使用すれば、ユーザ独自のメソッドからシステム定義の %Status エラーを呼び出すことができます。この場合、返したいエラー・メッセージに対応するエラー番号を指定します。

ObjectScript

```
WRITE "Here my method generates an error",!
SET status = $System.Status.Error(20)
WRITE $System.Status.GetErrorText(status),!
```

以下の例で示すように、返すエラー・メッセージに %1、%2、%3 パラメータを含むことができます。

ObjectScript

```
WRITE "Here my method generates an error",!
SET status = $System.Status.Error(214,"3","^fred","BedrockCode")
WRITE $System.Status.GetErrorText(status),!
```

エラー・メッセージをローカライズして、ユーザの優先言語で表示することもできます。

ObjectScript

```
SET status = $System.Status.Error(30)
WRITE "In English:",!
WRITE $System.Status.GetOneStatusText(status,1,"en"),!
WRITE "In French:",!
WRITE $System.Status.GetOneStatusText(status,1,"fr"),!
```

エラー・コードとメッセージ (英語) の一覧については、″InterSystems IRIS エラー・リファレンス″ の“一般的なエラー・メッセージ”の章を参照してください。

一般的なエラー・コード 83 および 5001 を使用すれば、いずれの一般エラー・メッセージにも対応していないカスタム・メッセージを指定することができます。

また、AppendStatus() メソッドを使用して、複数のエラー・メッセージの一覧を作成することができます。さらに、GetOneErrorText() または GetOneStatusText() を使用して、この一覧のそれぞれの位置ごとの各エラー・メッセージを取得することもできます。

ObjectScript

```
CreateCustomErrors
  SET st1 = $System.Status.Error(83,"my unique error")
  SET st2 = $System.Status.Error(5001,"my unique error")
  SET allstatus = $System.Status.AppendStatus(st1,st2)
DisplayErrors
  WRITE "All together:",!
  WRITE $System.Status.GetErrorText(allstatus),!!
  WRITE "One by one",!
  WRITE "First error format:",!
  WRITE $System.Status.GetOneStatusText(allstatus,1),!
  WRITE "Second error format:",!
  WRITE $System.Status.GetOneStatusText(allstatus,2),!
```

15.2.2 %SYSTEM.Error

%SYSTEM.Error クラスは一般的なエラー・オブジェクトです。これは %Status エラー、例外オブジェクト、$ZERROR エラー、または SQLCODE エラーから作成することができます。**%SYSTEM.Error** クラス・メソッドを使用すれば、%Status を例外に、または例外を %Status に変換することができます。

15.3 従来のエラー処理

このセクションでは、InterSystems IRIS を使用した従来のエラー処理のさまざまな側面について説明します。以下の内容が含まれます。

- 従来のエラー処理の概要
- $ZTRAP でのエラー処理
- $ETRAP でのエラー処理
- エラー・ハンドラによるエラー処理
- 強制エラー
- ターミナル・プロンプトからのエラー処理

15.3.1 従来のエラー処理の概要

InterSystems IRIS には、アプリケーションでエラー・ハンドラを使用して従来のエラー処理を実行できるようにするための機能が備わっています。エラー・ハンドラは、アプリケーションの実行中に発生するあらゆるエラーを処理します。エラー発生時に実行する ObjectScript コマンドは、特殊変数で指定します。 これらのコマンドは、エラーを直接処理するか、エラーを処理するルーチンを呼び出します。

エラー・ハンドラの作成には、以下の基本的な処理を実行します。

1. エラー処理を実行する 1 つ以上のルーチンを生成します。エラー処理を実行するコードを作成します。このコードは、アプリケーション全体に対する一般的なコードであっても、特定の処理における特定のエラー状態に対するものであってもかまいません。これにより、アプリケーションの特定個所で、カスタマイズされたエラー処理を実行できます。
2. アプリケーション内に 1 つ以上のエラー・ハンドラを設定し、それぞれのエラー・ハンドラで固有の適切なエラー処理を使用します。

エラーが発生してもエラー・ハンドラが設定されていなかった場合の振る舞いは、その InterSystems IRIS セッションの開始方法によって異なります。

1. ターミナル・プロンプトで InterSystems IRIS にサインオンし、エラー・トラップは設定していない場合、InterSystems IRIS は主デバイスにエラー・メッセージを表示し、プログラム・スタックの設定を変更せずにターミナル・プロンプトに戻ります。プログラマは、プログラムの実行を後で再開できます。
2. アプリケーション・モードで InterSystems IRIS を実行し、エラー・トラップを設定していない場合、InterSystems IRIS は主デバイスにエラー・メッセージを表示し、HALT コマンドを実行します。

15.3.1.1 内部的なエラー・トラップの振る舞い

特殊変数 $ZTRAP (および $ETRAP) による InterSystems IRIS のエラー処理とその有効範囲を最大限に活用するには、InterSystems IRIS が制御をあるルーチンから別のルーチンに移す方法を理解すると役立ちます。

InterSystems IRIS は、以下のいずれかが発生するたびに、“コンテキスト・フレーム” というデータ構造を構築します。

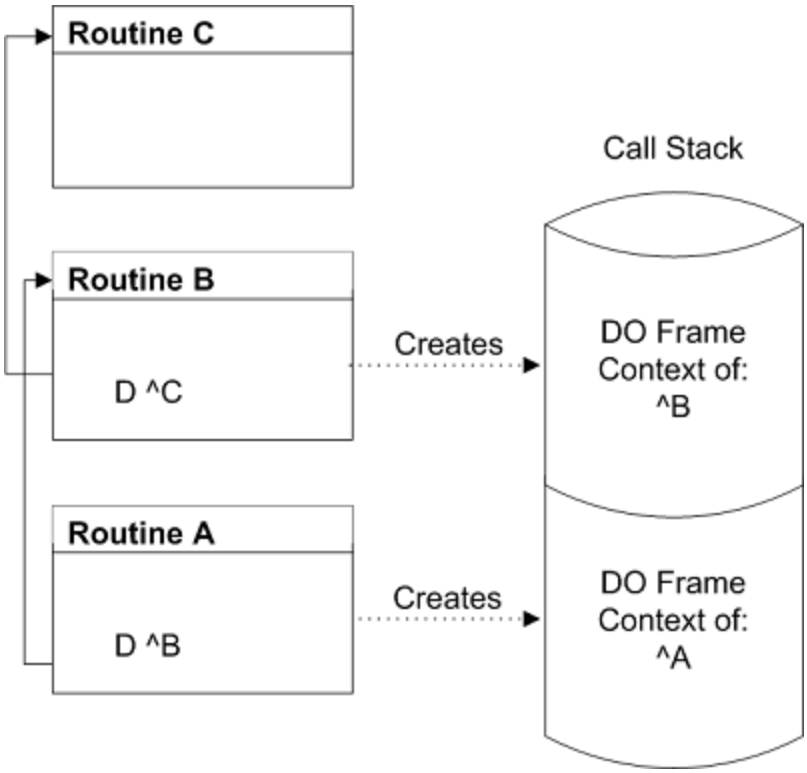
- ルーチンが Do コマンドで別のルーチンを呼び出す場合 (この種類のフレームは “DO フレーム” とも呼ばれます)。
- XECUTE コマンド引数により、ObjectScript コードが実行される場合 (この種類のフレームは “XECUTE フレーム” とも呼ばれます)。
- ユーザ定義関数を実行した場合

プロセスのアドレス部にあるプライベート・データ構造の 1 つ、コール・スタックにフレームが構築されます。InterSystems IRIS はルーチン用として、フレームに以下の要素を格納します。

- $ZTRAP 特殊変数の値 (存在する場合)
- $ETRAP 特殊変数の値 (存在する場合)
- サブルーチンから返される位置情報

例えば、ルーチン A がルーチン B を `DO ^B` で呼び出すと、InterSystems IRIS は、コール・スタックに A のコンテキストを保持する DO フレームを構築します。また、ルーチン B がルーチン C を呼び出すと、コール・スタックに B のコンテキストを保持する DO フレームを追加します。

図 15-1: コール・スタックのフレーム



上図のルーチン A がターミナル・プロンプトで DO コマンドを使用して呼び出される場合、図にはありませんが、追加の DO フレームがコール・スタックの一番下に存在しています。

15.3.1.2 現在のコンテキスト・レベル

現在のコンテキスト・レベルに関する情報を返すために以下を使用できます。

- $STACK 特殊変数は、現在の相対スタック・レベルを含みます。
- $ESTACK 特殊変数は、現在のスタック・レベルを含みます。任意のユーザ定義のポイントで、0 (レベル・ゼロ) に初期化できます。
- $STACK 関数は、現在のコンテキストと、コール・スタックに格納されているコンテキストに関する情報を返します。

$STACK 特殊変数

$STACK 特殊変数は、プロセスのコール・スタックに現在保存されているフレームの数を含みます。$STACK 値は、基本的に、現在実行しているコンテキストの (ゼロ・ベースの) レベル番号です。したがって、イメージが開始されるとき、コマンドが処理されるまで、$STACK の値は 0 です。

詳細は、"ObjectScript リファレンス" の "$STACK" 特殊変数を参照してください。

$ESTACK 特殊変数

$ESTACK 特殊変数は、$STACK 特殊変数に類似していますが、New コマンドで変数を 0 にリセットできるため (また以前の値を保存できます)、エラー処理には $ESTACK の方が便利です。したがって、プロセスは、特定のコンテキストで $ESTACK をリセットし、$ESTACK レベル 0 のコンテキストとしてマークすることができます。その後エラーが発生した場合、エラー・ハンドラは $ESTACK 値をテストし、コール・スタックをそのコンテキストまで戻すことができます。

詳細は、"ObjectScript リファレンス" の "$ESTACK" 特殊変数を参照してください。

$STACK 関数

$STACK 関数は、現在のコンテキストと、コール・スタックに格納されているコンテキストに関する情報を返します。各コンテキストで、$STACK 関数は以下の情報を提供します。

- コンテキスト・タイプ (Do、Xecute、ユーザ定義関数)
- コンテキストで処理された最後のコマンドのエントリ参照とコマンド番号
- コンテキストで処理された最後のコマンドを含むソース・ルーチン行あるいは XECUTE 文字列
- コンテキストで発生した任意のエラーの $ECODE 値 ($ECODE が NULL でない場合、エラー処理中にのみ利用可能)

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。その後、コンテキスト情報は、エラー・ハンドラが $ECODE 値をクリアするまで $STACK 関数によりアクセスできます。つまり、$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。

詳細は、"ObjectScript リファレンス" の "$STACK" 関数を参照してください。

エラーが発生し、エラー・スタックが既に存在している場合、InterSystems IRIS は、エラー・スタックの同じコンテキスト・レベルに別のエラーに関する情報が存在しない限り、コンテキスト・レベルで新しいエラーに関する情報を記録します。この場合、エラー情報は、(エラー情報が既に記録されているか否かにかかわらず) エラー・スタックの次のレベルに配置されます。

したがって、新規エラーのコンテキスト・レベルによっては、エラー・スタックは拡張される (1 つ以上のコンテキスト・レベルを追加される) か、あるいは、既存のエラー・スタックのコンテキスト・レベルに存在する情報が新規エラー情報を格納するために上書きされることがあります。

$ECODE 特殊変数をクリアすると、プロセス・エラー・スタックが消去されることに注意してください。

15.3.1.3 エラー・コード

エラー発生時、InterSystems IRIS は $ZERROR 特殊変数と $ECODE 特殊変数に、エラーを示す値を設定します。$ZERROR および $ECODE の値は、エラーの直後に使用することを目的としています。これらの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値を保持したい場合は、この値を変数にコピーする必要があります。

$ZERROR 値

InterSystems IRIS は、$ZERROR に以下を含む文字列を設定します。

- 山括弧で囲まれた InterSystems IRIS のエラー・コード
- エラー発生場所のラベル、オフセットおよびルーチン名
- 一部のエラーでは、エラーの原因となった項目の名前などの追加情報

%Exception.SystemException クラスの AsSystemError() メソッドは、$ZERROR と同じ形式で同じ値を返します。

以下の例では、InterSystems IRIS がエラーに遭遇した際に、$ZERROR に設定されるメッセージ・タイプを示します。以下の例では、未定義のローカル変数 abc が、ルーチン MyTest のラベル PrintResult より行オフセット 2 の位置で呼び出されます。$ZERROR は以下を含みます。

```
<UNDEFINED>PrintResult+2^MyTest *abc
```

存在しないクラスが行オフセット 3 の位置で実行されると、以下のエラーが発生します。

```
<CLASS DOES NOT EXIST>PrintResult+3^MyTest *%SYSTEM.XXQL
```

既存クラスの存在しないメソッドが行オフセット 4 の位置で実行されると、以下のエラーが発生します。

```
<METHOD DOES NOT EXIST>PrintResult+4^MyTest *BadMethod,%SYSTEM.SQL
```

128 文字までの任意の文字列として、特殊変数 $ZERROR を明示的に設定することもできます。以下はその例です。

ObjectScript

```
 SET $ZERROR="Any String"
```

$ZERROR 値は、エラーの直後に使用することを目的としています。$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 $ZERROR を使用したらすぐに NULL 文字列 (″″) に設定することを強くお勧めします。詳細は、″ObjectScript リファレンス″ の ″$ZERROR″ 特殊変数を参照してください。$ZERROR エラーの取り扱いについての詳細は、″インターシステムズ・クラス・リファレンス″ の %SYSTEM.Error クラス・メソッドを参照してください。

$ECODE 値

エラー発生時、コンマに囲まれた文字列値を InterSystems IRIS は $ECODE に設定します。この値は、エラーに対応する ANSI 標準エラー・コードを含んでいます。例えば、未定義のグローバル変数を参照する場合、InterSystems IRIS は $ECODE セットに以下の文字列を設定します。

```
,M7,
```

エラーに対応 ANSI 標準エラー・コードを持たない場合、コンマに囲まれた文字列値 (文字 Z が先頭についた InterSystems IRIS エラー・コードが含まれる) に $ECODE を InterSystems IRIS は設定します。例えば、プロセスがシンボル・テーブルの領域をすべて使用した場合、InterSystems IRIS は $ZERROR 特殊変数にエラー・コード <STORE> を格納し、$ECODE に以下の文字列を設定します。

```
,ZSTORE,
```

エラー発生後、エラー・ハンドラは $ZERROR 特殊変数あるいは $ECODE 特殊変数を検証して、特定のエラー・コードをテストできます。

注釈　エラー・ハンドラは、特定のエラーに関し、$ECODE 特殊変数ではなく $ZERROR 特殊変数を検証する必要があります。

詳細は、″ObjectScript リファレンス″ の ″$ECODE″ 特殊変数を参照してください。

15.3.2 $ZTRAP でのエラー処理

$ZTRAP でエラーを処理するには、$ZTRAP 特殊変数を location に設定し、引用符で囲まれた文字列として指定します。$ZTRAP 特殊変数はエントリ参照に対して設定します。このエントリ参照によって、エラー発生時における制御の移管先となる location を指定します。次にその場所に $ZTRAP コードを記述します。

$ZTRAP を空以外の値に設定すると、これは既存のどの $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 ″″ に設定します。

15.3.2.1 プロシージャでの $ZTRAP の設定

プロシージャ内では、そのプロシージャ内の行ラベル (プライベート・ラベル) にのみ $ZTRAP 特殊変数を設定できます。プロシージャ・ブロック内から $ZTRAP を外部ルーチンに設定することはできません。

$ZTRAP 値を表示するとき、InterSystems IRIS はプライベート・ラベルの名前を返しません。代わりに、プロシージャの先頭からのそのプライベート・ラベルがある場所までのオフセットを返します。

詳細は、″ObjectScript リファレンス″ の ″$ZTRAP″ 特殊変数を参照してください。

15.3.2.2 ルーチンでの $ZTRAP の設定

ルーチン内では、$ZTRAP 特殊変数を現在のルーチンのラベル、外部ルーチン、または外部ルーチン内のラベルに設定できます。外部ルーチンを参照できるのは、そのルーチンがプロシージャ・ブロック・コードではない場合だけです。以下の例では、LogErr^ErrRou をエラー・ハンドラとして構築します。エラー発生時には、InterSystems IRIS が ^ErrRou ルーチン内の LogErr ラベルにあるコードを実行します。

ObjectScript

```
SET $ZTRAP="LogErr^ErrRou"
```

$ZTRAP 値を表示するとき、InterSystems IRIS はラベル名と (該当する場合には) ルーチン名を表示します。

ラベル名は最初の 31 文字内で一意である必要があります。ラベル名およびルーチン名では、大文字と小文字が区別されます。

ルーチン内では、$ZTRAP には 3 つの形式があります。

- SET $ZTRAP="location"
- SET $ZTRAP="*location"。この形式では、この関数を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。
- SET $ZTRAP="^%ETN" は、システムで提供されるエラー・ルーチン %ETN を実行します。このルーチンはルーチンを呼び出すエラーが発生したコンテキストにおいて実行されます。プロシージャ・ブロック内から ^%ETN (またはいずれかの外部ルーチン) を実行することはできません。コードが [Not ProcedureBlock] であることを指定するか、%ETN エントリポイント BACK^%ETN を呼び出す以下のようなルーチンを使用してください。

  ```
  ClassMethod MyTest() as %Status
    {
    SET $ZTRAP="Error"
    SET ans = 5/0      /* divide-by-zero error */
    WRITE "Exiting ##class(User.A).MyTest()",!
    QUIT ans
  Error
    SET err=$ZERROR
    SET $ZTRAP=""
    DO BACK^%ETN
    QUIT $$$ERROR($$$CacheError,err)
    }
  ```

  %ETN およびそのエントリポイントの詳細は、"アプリケーション・エラーのログ作成" を参照してください。$ZTRAP と共に使用する際の処理の詳細は、"SET $ZTRAP=^%ETN" を参照してください。

詳細は、"ObjectScript リファレンス" の "$ZTRAP" 特殊変数を参照してください。

15.3.2.3 $ZTRAP コードの記述

$ZTRAP が指し示す location はさまざまな操作を実行して、エラーを表示、ログ記録、または修正することができます。実行したいエラー処理操作の如何にかかわらず、$ZTRAP コードは以下の 2 つのタスクの実行により開始する必要があります。

- $ZTRAP を他の値、すなわちエラー・ハンドラの location または空文字列 ("") のいずれかに設定します。(KILL $ZTRAP. とすることはできないので、SET を使用する必要があります。)これを行う理由は、エラー処理中に別のエラーが発生した場合において、そのエラーが現在の $ZTRAP エラー・ハンドラを呼び出してしまうためです。現在のエラー・ハンドラがユーザの関わるエラー・ハンドラである場合、無限ループとなってしまいます。
- ある変数を $ZERROR に設定します。後でコード内で $ZERROR 値を参照する場合は、$ZERROR 自体ではなく、この変数を参照します。これを行う理由は、$ZERROR には最新のエラーが含まれているうえ、$ZERROR 値が、内部ルーチン呼び出しなどの複数のルーチン呼び出しにわたって保持されない可能性があるからです。エラー処理中に別のエラーが発生した場合、$ZERROR の値がその新しいエラーによって上書きされることになります。

  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。

以下の例では、これらの不可欠な $ZTRAP コード文を示します。

ObjectScript

```
MyErrHandler
  SET $ZTRAP=""
  SET err=$ZERROR
  /* error handling code
     using err as the error
     to be handled */
```

15.3.2.4 $ZTRAP の使用法

アプリケーションの各ルーチンは、$ZTRAP を設定して、独自の $ZTRAP エラー・ハンドラを作成できます。エラー・トラップの発生時、InterSystems IRIS は以下の手順を実行します。

1. 特殊変数 $ZERROR にエラー・メッセージを設定します。
2. プログラム・スタックを、エラー・トラップが設定された時点 (SET $ZTRAP= の実行時) の状態にリセットします。つまり、システムは、エラー・トラップが設定された時点まで、スタックに存在するすべてのエントリを削除します。($ZTRAP がアスタリスク (*) で始まる文字列に設定されている場合、プログラム・スタックはリセットされません。)
3. $ZTRAP の値により指定された位置でプログラムを再開します。$ZTRAP の値は保持されます。

注釈　$ZERROR 変数を、128 文字までの文字列として明示的に設定できます。通常、$ZERROR は NULL 文字列にのみ設定しますが、$ZERROR を値に設定してもかまいません。

15.3.2.5 エラー・トラップによる New コマンドの削除

エラー・トラップが発生しプログラム・スタック・エントリが削除されると、InterSystems IRIS は、スタックされたすべての NEW コマンドも削除し、SET $ZTRAP= を含むサブルーチン・レベルに戻します。しかし、NEW コマンドが $ZTRAP の設定前あるいは設定後にスタックに追加されたかどうかにかかわらず、そのサブルーチン・レベルで実行されたすべての NEW コマンドは維持されます。

以下はその例です。

ObjectScript

```
Main
  SET A=1,B=2,C=3,D=4,E=5,F=6
  NEW A,B
  SET $ZTRAP="ErrSub"
  NEW C,D
  DO Sub1
  RETURN
Sub1()
  NEW E,F
  WRITE 6/0    // Error: division by zero
  RETURN
ErrSub()
  WRITE !,"Error is: ",$ZERROR
  WRITE
  RETURN
```

Sub1 のエラーがエラー・トラップを起動した場合、Sub1 にスタックされた E と F における以前の値は削除されます。しかし、A、B、C、D はスタックされたままになります。

15.3.2.6 $ZTRAP フロー制御オプション

$ZTRAP エラー・ハンドラがエラー処理のために呼び出され、クリーンアップとエラー記録処理を実行した後、このエラー・ハンドラには 3 つのフロー制御オプションがあります。

- エラーを処理し、アプリケーションを継続します。

- 制御を別のエラー・ハンドラに渡します。
- アプリケーションを終了します。

アプリケーションの継続

$ZTRAP がエラーを処理した後には、GOTO の発行によりアプリケーションを続行することができます。通常のアプリケーション処理を継続するために、$ZERROR または $ECODE 特殊変数値をクリアする必要はありません。しかし、$ZTRAP をクリアし (空の文字列を設定)、別のエラーが発生した際に無限のエラー処理ループに入らないようにする必要があります。詳細は、“エラー・ハンドラによるエラー処理”を参照してください。

エラー処理を実行した後、$ZTRAP エラー・ハンドラは GOTO コマンドを使用して、アプリケーション内で事前設定されたリスタートまたは継続ポイントに制御を移し、通常のアプリケーション処理を再開できます。

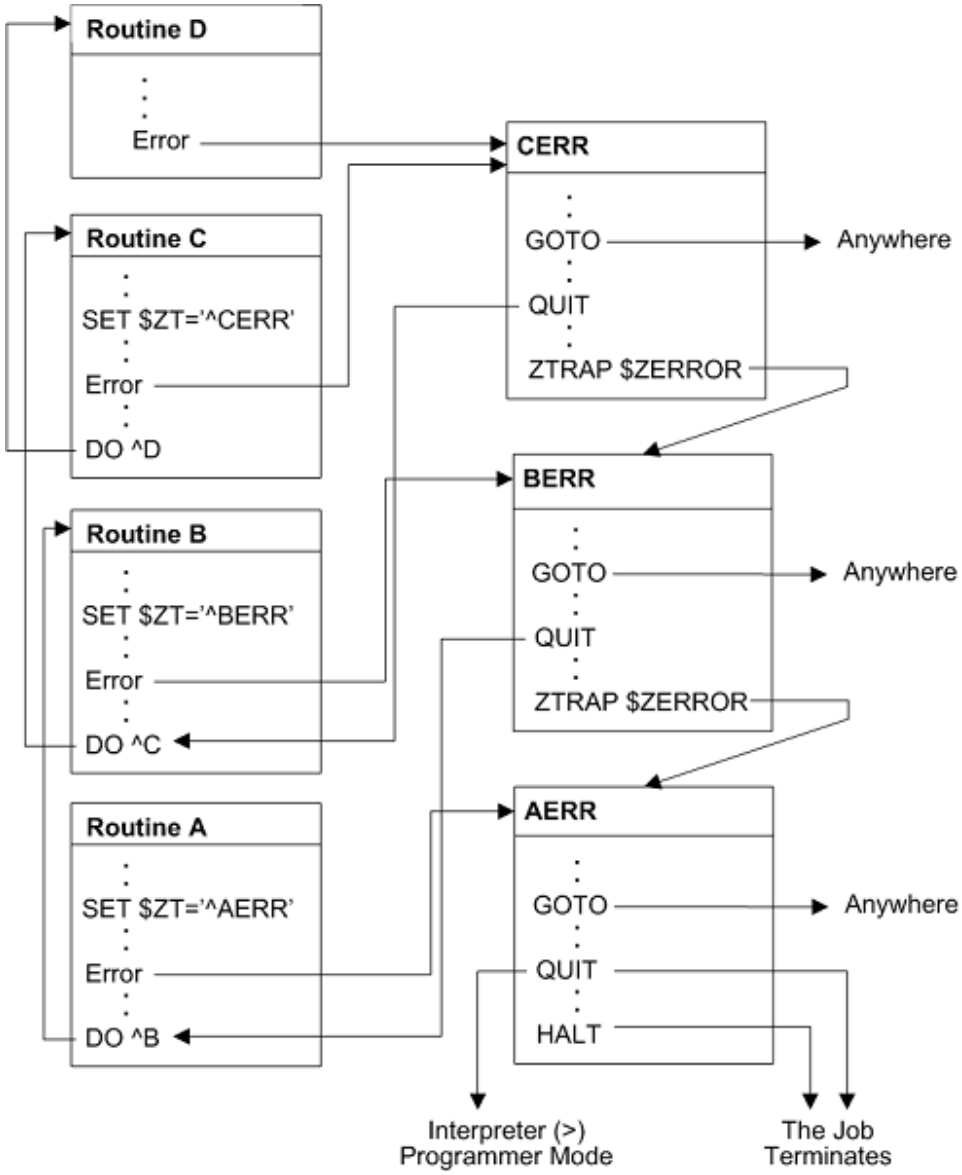
エラー・ハンドラがエラーを処理すると、$ZERROR 特殊変数が値に設定されます。エラー・ハンドラが処理を終了すると、この値は必ずしもクリアされるわけではありません。ルーチンによっては、$ZERROR を NULL 文字列に設定し直すものがあります。エラー・ハンドラを呼び出すエラーが次に発生したときに、$ZERROR は上書きされます。この理由から、$ZERROR 値はエラー・ハンドラのコンテキスト内でのみ使用する必要があります。この値を保持する場合は、これを変数にコピーして、$ZERROR 自体ではなくその変数を参照します。その他のコンテキストで $ZERROR を使用すると、信頼できる結果が生成されません。

別のエラー・ハンドラへの制御の移動

エラー条件を $ZTRAP エラー・ハンドラで修正できない場合、特殊な形式の ZTRAP コマンドを使用して、別のエラー・ハンドラに制御を移すことができます。コマンド **ZTRAP $ZERROR** は、エラー条件を再度示し、エラー・ハンドラを使って、InterSystems IRIS で次のコール・スタック・レベルまでコール・スタックが戻るようにします。InterSystems IRIS で次のエラー・ハンドラのレベルにコール・スタックを戻したら、そのエラー・ハンドラで処理が続行します。次のエラー・ハンドラは、$ZTRAP または $ETRAP で設定された可能性があります。

下図は、$ZTRAP エラー処理ルーチンのフロー制御を示しています。

図 15-2: $ZTRAP エラー・ハンドラ



15.3.3 $ETRAP でのエラー処理

エラー・トラップの発生時に $ETRAP が設定されている場合、InterSystems IRIS は以下の手順を実行します。

1. $ECODE および $ZERROR の値を設定します。
2. $ETRAP の値であるコマンドを処理します。

既定で、DO、XECUTE、またはユーザ定義の各関数コンテキストは、それを呼び出したフレームの $ETRAP エラー・ハンドラを継承します。つまり、どのコンテキスト・レベルであっても、指定された $ETRAP エラー・ハンドラが最終的に $ETRAP として定義されます。その定義が現在のレベルよりもいくつか低いスタック・レベルで行われていた場合でも同様です。

15.3.3.1 $ETRAP エラー・ハンドラ

$ETRAP 特殊変数には、エラーの発生時に実行される 1 つ以上の ObjectScript コマンドを含めることができます。SET コマンドを使用して、$ETRAP を、制御をエラー処理ルーチンに移す 1 つ以上の InterSystems IRIS コマンドを含む文字列に設定します。以下の例では、制御が LogError コード・ラベル (ルーチン ErrRoutine の一部) に移されます。

ObjectScript

```
SET $ETRAP="DO LogError^ErrRoutine"
```

$ETRAP 特殊変数内のコマンドの後ろには常に、暗黙の QUIT コマンドが続きます。引数付きの QUIT コマンドを必要とするユーザ定義の関数コンテキストで $ETRAP エラー・ハンドラが呼び出されると、暗黙の QUIT コマンドは NULL 文字列の引数で終了します。

$ETRAP はグローバル・スコープを持ちます。これは、通常、$ETRAP の設定の前には **NEW $ETRAP** が必要であることを意味します。そうしなければ、現在のコンテキストで $ETRAP の値が設定されている場合、その値がそのコンテキストの範囲を超えて渡された後、制御は上位レベルのコンテキストにあるのに、$ETRAP に格納された値は存在し続けることになります。このため、**NEW $ETRAP** を指定しないと、$ETRAP が設定されたときのコンテキストがもう存在しない場合、$ETRAP が予期しないタイミングで実行される可能性があります。

詳細は、"ObjectScript リファレンス" の "$ETRAP" 特殊変数を参照してください。

15.3.3.2 コンテキスト固有の $ETRAP エラー・ハンドラ

以下の手順を実行することで、あらゆるコンテキストでコンテキスト固有の $ETRAP エラー・ハンドラを作成できます。

1. NEW コマンドを使用して、$ETRAP の新しいコピーを作成します。
2. $ETRAP を新しい値に設定します。

ルーチンで最初に $ETRAP の新しいコピーを作成せずに $ETRAP を設定すると、現在のコンテキスト、それを呼び出したコンテキスト、およびコール・スタックに保存されている可能性のあるその他のコンテキストに、新しい $ETRAP エラー・ハンドラが作成されます。このため、$ETRAP を設定する前に、その新しいコピーを作成することをお勧めします。

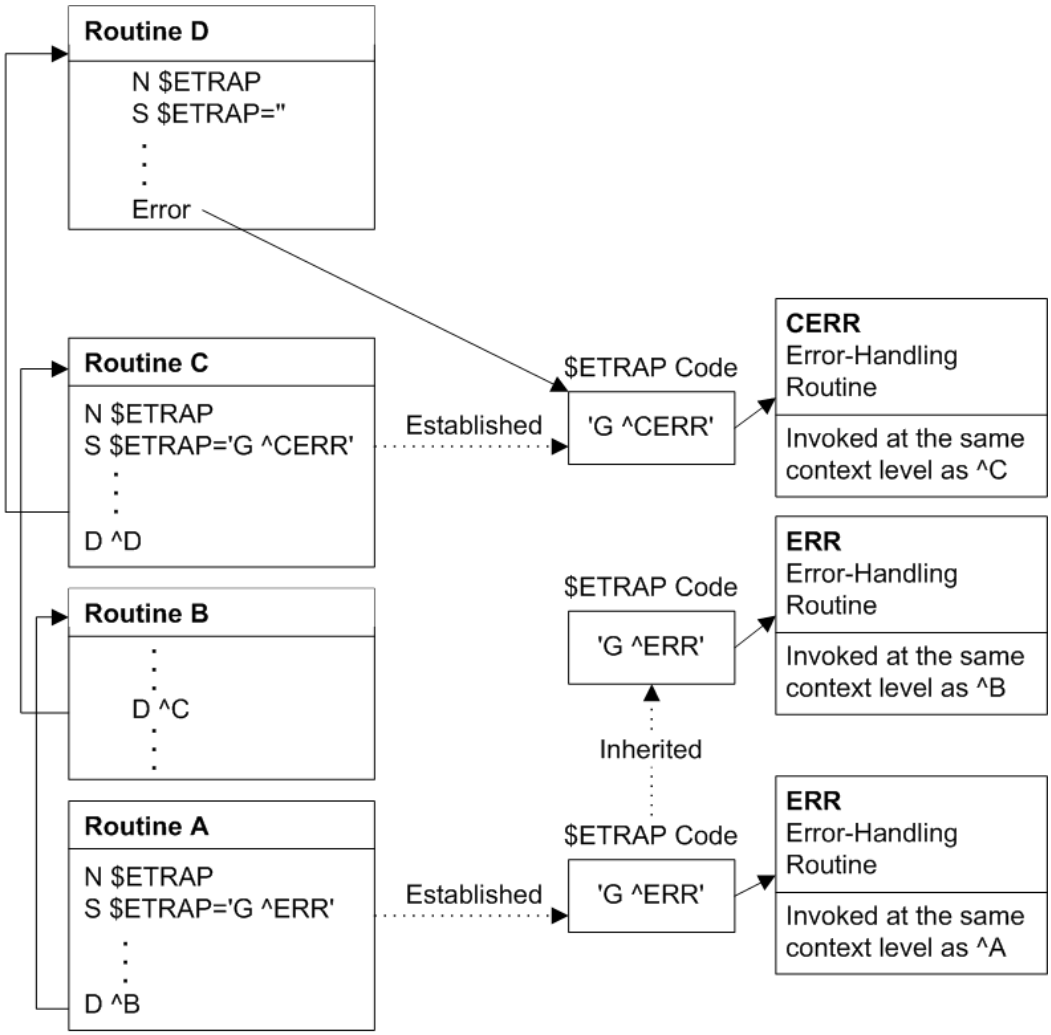
$ETRAP の新しいコピーを作成することで $ETRAP はクリアされないことに注意してください。$ETRAP の値は、NEW コマンドによって変更されないまま保持されます。

以下の図は、$ETRAP エラー・ハンドラのスタックを作成する $ETRAP 割り当てのシーケンスを示しています。この図が示している内容は次のとおりです。

- ルーチン A は、$ETRAP の新しいコピーを作成して、それを “GOTO ^ERR” に設定し、ルーチン B を呼び出すための DO コマンドを含んでいます。
- ルーチン B は、$ETRAP に対して何も行わず (その結果、ルーチン A の $ETRAP エラー・ハンドラを継承し)、ルーチン C を呼び出すための DO コマンドを含んでいます。
- ルーチン C は $ETRAP の新しいコピーを作成して、それを “GOTO ^CERR” に設定し、ルーチン D を呼び出すための DO コマンドを含んでいます。
- ルーチン D は、$ETRAP の新しいコピーを作成した後、それをクリアして、コンテキストに $ETRAP エラー・ハンドラを残しません。

ルーチン D でエラーが発生した場合 ($ETRAP エラー・ハンドラが定義されていないコンテキスト)、InterSystems IRIS はルーチン D の DO フレームをコール・スタックから削除して、制御をルーチン C の $ETRAP エラー・ハンドラに移します。ルーチン C の $ETRAP エラー・ハンドラはさらに ^CERR にディスパッチしてエラーが処理されます。ルーチン C でエラーが発生した場合、InterSystems IRIS は制御をルーチン C の $ETRAP エラー・ハンドラに移しますが、エラーが発生しているのは $ETRAP エラー・ハンドラが定義されているコンテキストであるため、スタックを戻すことはしません。

図 15-3: $ETRAP エラー・ハンドラ



15.3.3.3 $ETRAP フロー制御オプション

$ETRAP エラー・ハンドラが、エラーを処理し、クリーンアップまたはエラー・ログ処理を実行するために呼び出された場合、次のフロー制御オプションがあります。

- エラーを処理し、アプリケーションを継続します。
- 制御を別のエラー・ハンドラに渡します。
- アプリケーションを終了します。

エラーの処理とアプリケーションの継続

エラーを処理するために $ETRAP エラー・ハンドラが呼び出されると、InterSystems IRIS は、エラー状態が解除されるまでそのエラー状態がアクティブであると見なします。エラー状態を解除するには、$ECODE 特殊変数を NULL 文字列に設定します。

ObjectScript

```
SET $ECODE=""
```

$ECODE のクリアによって、プロセスのエラー・スタックもクリアされます。

エラー状態が解除されたら、通常は、GOTO コマンドを使用して、アプリケーションで事前設定されている再起動または継続のポイントに制御を移します。場合によっては、エラー状態の解除後に、終了して前のコンテキスト・レベルに戻る方が簡単なこともあります。

別のエラー・ハンドラへの制御の移動

エラー状態が解除されていない場合、$ETRAP エラー・ハンドラが呼び出されたコンテキストを QUIT コマンドが終了すると、InterSystems IRIS はコール・スタック上の別のエラー・ハンドラに制御を渡します。このため、$ECODE をクリアせずに、$ETRAP コンテキストから QUIT を実行することで、前のレベルのエラー・ハンドラに制御を渡すことができます。

ルーチン C から呼び出されたルーチン D に、制御を ^CERR に移すエラーが含まれる場合、前もって $ECODE が "" (空の文字列) に設定されていない ^CERR の QUIT コマンドは、前のコンテキスト・レベルの $ETRAP エラー・ハンドラに制御を移します。一方、$ECODE のクリアによってエラー状態が解除された場合、^CERR の QUIT は、ルーチン B の DO ^C コマンドに続く文に制御を移します。

アプリケーションの終了

コール・スタックに前のレベルのエラー・ハンドラが存在しておらず、$ETRAP エラー・ハンドラがエラー状態を解除せずに QUIT を実行した場合、アプリケーションが終了します。アプリケーション・モードでは その後 InterSystems IRIS は停止し、制御がオペレーティング・システムに渡されます。それに続いて、ターミナル・プロンプトが表示されます。

エラー状態が解除されているかどうかにかかわらず、$ETRAP エラー・ハンドラのコンテキストを終了するには、QUIT コマンドを使用することに留意してください。引数なしの QUIT が必要なコンテキスト・レベルと引数ありの QUIT が必要なコンテキスト・レベル (ユーザ定義関数のコンテキスト) で、同じ $ETRAP エラー・ハンドラを呼び出すことができるため、特定のコンテキスト・レベルで必要とされる QUIT コマンド形式を示すために $QUIT 特殊変数が指定されます。

$QUIT 特殊変数は、引数ありの QUIT が必要なコンテキストの場合は 1 を返し、引数なしの QUIT が必要なコンテキストの場合は 0 を返します。

$ETRAP エラー・ハンドラは、$QUIT を使用することで、以下のようにどちらの状況にも対処できます。

ObjectScript

```
Quit:$QUIT "" Quit
```

該当する場合、$ETRAP エラー・ハンドラは、HALT コマンドを使用してアプリケーションを終了することができます。

15.3.4 エラー・ハンドラによるエラー処理

エラー・ハンドラでエラーが発生するとき、実行フローは、現在実行中のエラー・ハンドラ・タイプにより異なります。

15.3.4.1 $ZTRAP エラー・ハンドラでのエラー

$ZTRAP エラー・ハンドラで新規エラーが発生すると、InterSystems IRIS は、最初に検出したエラー・ハンドラに制御を渡し、必要に応じてコール・スタックを戻します。したがって、$ZTRAP エラーが現在のスタック・レベルで $ZTRAP をクリアせず、別のエラーが同じエラー・ハンドラで続けて発生した場合、$ZTRAP ハンドラが同じコンテキスト・レベルで再度呼び出されるため、無限ループの原因となります。これを回避するには、エラー・ハンドラの開始時点において $ZTRAP に他の値を設定します。

15.3.4.2 $ETRAP エラー・ハンドラでのエラー

$ETRAP エラー・ハンドラで新しいエラーが発生すると、InterSystems IRIS は、$ETRAP エラー・ハンドラが呼び出されたコンテキスト・レベルが削除されるまでコール・スタックを戻します。その後、InterSystems IRIS はコール・スタック上の次のエラー・ハンドラ (存在する場合) に制御を渡します。

15.3.4.3 $ZERROR 特殊変数と $ECODE 特殊変数のエラー情報

元のエラーの処理中に別のエラーが発生した場合、2 番目のエラー情報により、$ZERROR 特殊変数の最初のエラー情報が置き換えられます。しかし、InterSystems IRIS は、$ECODE 特殊変数に新規情報を追加します。2 番目のエラーのコンテキスト・レベルによっては、InterSystems IRIS は新しい情報をプロセス・エラー・スタックに追加する場合もあります。

$ECODE 特殊変数の既存の値が NULL 以外の場合、InterSystems IRIS は、新しいコンマ区切りの部分として、新規エラー・コードを現在の $ECODE 値に追加します。以下のいずれかの状況が発生するまで、$ECODE 特殊変数にエラー・コードが追加されます。

- 以下のようにして、$ECODE を明示的にクリアします。

  ObjectScript

  ```
  SET $ECODE = ""
  ```

- $ECODE の長さは、最大文字列長を超えています。

その後、次の新規エラー・コードにより、$ECODE にある現在のエラー・コード・リストが置き換えられます。

エラーが発生し、エラー・スタックが既に存在している場合、InterSystems IRIS は、エラー・スタックの同じコンテキスト・レベルに別のエラーに関する情報が存在しない限り、コンテキスト・レベルで新しいエラーに関する情報を記録します。この場合、エラー情報は、(エラー情報が既に記録されているかどうかにかかわらず) エラー・スタックの次のレベルに配置されます。

したがって、新規エラーのコンテキスト・レベルによっては、エラー・スタックは拡張される (1 つ以上のコンテキスト・レベルを追加される) か、あるいは、既存のエラー・スタックのコンテキスト・レベルに存在する情報が新規エラー情報を格納するために上書きされることがあります。

$ECODE 特殊変数をクリアすると、プロセス・エラー・スタックが消去されることに注意してください。

詳細は、"ObjectScript リファレンス" の "$ECODE" 特殊変数と "$ZERROR" 特殊変数を参照してください。$ZERROR エラーの取り扱いについての詳細は、"インターシステムズ・クラス・リファレンス" の %SYSTEM.Error クラス・メソッドを参照してください。

15.3.5 強制エラー

$ECODE 特殊変数を設定、あるいは ZTRAP コマンドを使用して、制御環境でエラーを発生させます。

15.3.5.1 $ECODE の設定

$ECODE 特殊変数に NULL 以外の文字列を設定し、エラーを発生させることができます。ルーチンで $ECODE に NULL 以外の文字列を設定すると、InterSystems IRIS は $ECODE に特定の文字列を設定し、その後、エラー状態を生成します。この状況で $ZERROR 特殊変数は、以下のエラー・テキストと共に設定されます。

```
<ECODETRAP>
```

その後、通常のアプリケーション・エラーのように、エラー・ハンドラに制御を渡します。

エラー・ハンドラにロジックを追加し、$ECODE の設定により発生させるエラーをチェックできます。エラー・ハンドラは、<ECODETRAP> エラー (例えば、"$ZE["ECODETRAP""" に対して $ZERROR をチェックするか、または選択する特定の文字列値に対して $ECODE をチェックできます。

15.3.5.2 アプリケーション固有エラーの生成

$ECODE の ANSI 標準フォーマットは、1 つ以上のコンマに囲まれたエラー・コードのリストです。

- 接頭語 "Z" 付きのエラーは、実装固有のエラーです。

- 接頭語“U”付きのエラーは、アプリケーション固有のエラーです。

エラー・ハンドラで $ECODE に接頭語“U”を持つ適切なエラー・メッセージを設定し、ANSI 標準に合う独自のエラー・コードを生成できます。

ObjectScript

```
SET $ECODE=",Upassword expired,"
```

15.3.6 ターミナル・プロンプトでのエラー処理

エラー・ハンドラ・セットを持たないターミナル・プロンプトで InterSystems IRIS にサインオンした後エラーを生成する場合、InterSystems IRIS は、入力したコード行でエラーが発生すると、以下の手順を実行します。

1. InterSystems IRIS は、プロセスの主デバイスで、エラー・メッセージを表示します。
2. エラーが発生したコール・スタック・レベルでプロセスを中断します。
3. プロセスをターミナル・プロンプトに戻します。

15.3.6.1 エラー・メッセージ形式の理解

エラー・メッセージとして、InterSystems IRIS は以下の 3 行を表示します。

1. エラーが発生したすべてのソース・コード行
2. ソース・コードの行の下で、エラーが発生したコマンドを指定するキャレット (^)
3. $ZERROR のコンテンツを含む行

以下のターミナル・プロンプトの例では、2 番目の SET コマンドに未定義のローカル変数エラーがあります。

Terminal

```
USER>WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
hello

WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                              ^
<UNDEFINED> *zzz
USER>
```

以下の例では、ターミナル・プロンプトから実行される `mytest` という名前のプログラムにコードと同じ行があります。

Terminal

```
USER>DO ^mytest
hello

  WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                                ^
<UNDEFINED>WriteOut+2^mytest *zzz
USER 2d0>
```

この場合、$ZERROR は、`mytest` の中で `WriteOut` という名前のラベルから 2 行のオフセット位置でエラーが発生したことを示しています。プロンプトが変更され、新規のプログラム・スタック・レベルが開始されたことを示している点に注意してください。

15.3.6.2 ターミナル・プロンプトの理解

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示します。トランザクションが 1 つ以上開いている場合、$TLEVEL トランザクション・レベル・カウントも含まれます。ZNSPACE コマンドの説明に従って、既定のプロンプトを異なる内容で構成することができます。以下の例は、それらの既定の設定を示します。

Terminal

```
USER>
```

Terminal

```
TL1:USER>
```

ルーチンの実行中にエラーが発生すると、システムは、現在のプログラム・スタックを保存して、新規スタック・フレームを開始します。以下のような拡張プロンプトが表示されます。

Terminal

```
USER 2d0>
```

この拡張プロンプトは、プログラム・スタックに 2 つのエントリがあり、その最後の方が DO (d で示されています) の呼び出しであることを示しています。このエラーにより、プログラム・スタックに 2 つのエントリが置かれることに注意してください。次の DO 実行エラーにより、以下のプロンプト結果となります。

Terminal

```
USER 4d0>
```

説明の詳細は、“コマンド行ルーチンのデバッグ” の章の ″ターミナル・プロンプトのプログラム・スタック情報表示″ を参照してください。

15.3.6.3 エラーのリカバリ

以下のいずれかの手順を実行できます。

- ターミナル・プロンプトからコマンドを発行します。
- 変数とグローバル・データを表示、修正します。
- エラーと他のルーチンを含むルーチンを編集します。
- 他のルーチンを実行します。

これらの手順により、別のエラーを発生させる可能性もあります。

これらの手順の実行後、実行を再開する、あるいはプログラム・スタックの一部/すべてを削除することをお勧めします。

次に続くコマンドでの実行の再開

ターミナル・プロンプトから引数なしの GOTO を入力すれば、エラーを引き起こしたコマンドの次のコマンドで実行を再開できます。

Terminal

```
USER>DO ^mytest
hello

  WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                                ^
<UNDEFINED>WriteOut+2^mytest *zzz
USER 2d0>GOTO
world

USER>
```

別の行での実行の再開

ターミナル・プロンプトからラベル引数付きの GOTO を発行すると、別の行で実行を再開できます。

Terminal

```
USER 2d0>GOTO ErrSect
```

プログラム・スタックの削除

ターミナル・プロンプトから引数なしの QUIT コマンドを発行すると、プログラム・スタック全体を削除できます。

Terminal

```
USER 4d0>QUIT
USER>
```

プログラム・スタックの部分削除

ターミナル・プロンプトから整数の引数付きの QUIT n コマンドを発行すると、最後の (または最後にある複数の) プログラム・スタック・エントリを削除できます。

Terminal

```
USER 8d0>QUIT 1

USER 7E0>QUIT 3

USER 4d0>QUIT 1

USER 3E0>QUIT 1

USER 2d0>QUIT 1

USER 1S0>QUIT 1

USER>
```

この例では、プログラム・エラーによって 2 つのプログラム・スタック・エントリが作成されているので、GOTO を発行して実行を再開するには、“d” スタック・エントリに移動している必要があります。他に発生する内容に応じて、“d” スタック・エントリは偶数 (USER 2d0>) または奇数 (USER 3d0>) となります。

NEW $ESTACK を使用すれば、指定のプログラム・スタック・レベルで終了できます。

Terminal

```
USER 4d0>NEW $ESTACK

USER 5E1>
/* more errors create more stack frames */

USER 11d7>QUIT $ESTACK

USER 4d0>
```

NEW $ESTACK コマンドにて、エントリが 1 つプログラム・スタックに追加されます。

15.4 アプリケーション・エラーのログ作成

InterSystems IRIS には、例外をアプリケーション・エラー・ログに記録する方法がいくつか用意されています。

- %ETN ユーティリティはエラーを記録します。これは、^%ETN として、またはそのエントリポイント FORE^%ETN、BACK^%ETN、または LOG^%ETN のいずれかを使用して呼び出すことができます。
- %Exception.AbstractException.Log() メソッド。

15.4.1 %ETN を使用したアプリケーション・エラーのログ作成

%ETN ユーティリティは、例外をアプリケーション・エラー・ログに記録した後、終了します。%ETN (またはそのエントリポイントのいずれか) をユーティリティとして呼び出すことができます。

ObjectScript

```
DO ^%ETN
```

または、$ZTRAP 特殊変数を %ETN (またはそのエントリポイントのいずれか) に設定することができます。

ObjectScript

```
SET $ZTRAP="^%ETN"
```

%ETN またはそのエントリポイントのいずれかを指定できます。

- FORE^%ETN (フォアグラウンド) は、例外を標準アプリケーション・エラー・ログに記録した後、HALT によって終了します。これは、ロールバック処理を起動します。これは、%ETN と同じ処理です。
- BACK^%ETN (バックグラウンド) は、例外を標準アプリケーション・エラー・ログに記録した後、QUIT によって終了します。これは、ロールバック処理を起動しません。
- LOG^%ETN は、例外を標準アプリケーション・エラー・ログに記録した後、QUIT によって終了します。これは、ロールバック処理を起動しません。この例外は、標準の %Exception.SystemException またはユーザ定義の例外です。

  例外を定義するには、LOG^%ETN を呼び出す前に $ZERROR を意味のある値に設定します。この値は、ログ・エントリのエラー・メッセージ・フィールドとして使用されます。ユーザ定義の例外を直接 LOG^%ETN: `DO LOG^%ETN("This is my custom exception")` に指定することもできます。この値は、ログ・エントリのエラー・メッセージ・フィールドとして使用されます。$ZERROR を NULL 文字列に設定すると (`SET $ZERROR=""`)、LOG^%ETN は <LOG ENTRY> エラーを記録します。$ZERROR を <INTERRUPT> に設定すると (`SET $ZERROR="<INTERRUPT>"`)、LOG^%ETN は <INTERRUPT LOG> エラーを記録します。

  LOG^%ETN は、$HOROLOG 日付とエラー番号の 2 つの要素を持つ %List 構造を返します。

以下の例では、$ZERROR を変数に即座にコピーするお勧めのコーディング方法を使用しています。LOG^%ETN は %List 値を返します。

ObjectScript

```
SET err=$ZERROR
/* error handling code */
SET rtn = $$LOG^%ETN(err)
WRITE "logged error date: ",$LIST(rtn,1),!
WRITE "logged error number: ",$LIST(rtn,2)
```

LOG^%ETN または BACK^%ETN を呼び出すと、使用可能なプロセス・メモリが自動的に増加し、処理が実行されて、$ZSTORAGE の元の値がリストアされます。ただし、<STORE> エラーの発生後に LOG^%ETN または BACK^%ETN を呼

び出した場合、$ZSTORAGE の元の値をリストアすることによって別の <STORE> エラーがトリガされる可能性があります。このため、<STORE> エラーに対してこれらの %ETN エントリポイントが呼び出された場合は、増加した使用可能メモリが維持されます。

15.4.2 管理ポータルを使用したアプリケーション・エラー・ログの表示

管理ポータルで、[**システム処理**]→[**システムログ**]→[**アプリケーションエラーログ**] の順に選択します。これにより、アプリケーション・エラー・ログが発生したネームスペースの [**ネームスペース**] リストが表示されます。ヘッダを使用して、リストをソートできます。

ネームスペースの [**日付**] を選択すると、アプリケーション・エラー・ログが発生した日付と、その日付で記録されているエラーの数が表示されます。ヘッダを使用して、リストをソートできます。[**フィルタ**] を使用して、[日付] と [数量] の値に文字列をマッチングできます。

日付の [**エラー**] を選択すると、その日付のエラーが表示されます。[エラー#] の整数は、時系列順にエラーに割り当てられます。[エラー#] の `*COM` は、対象の日付のすべてのエラーに適用されるユーザ・コメントです。ヘッダを使用して、リストをソートできます。[**フィルタ**] を使用して、文字列をマッチングできます。

エラーの [**詳細**] を選択すると、[**エラー詳細**] ウィンドウが表示され、特殊変数の値や [**スタック**] の詳細など、エラー発生時の状態に関する情報が示されます。個々のエラーにユーザ・コメントを指定できます。

[**ネームスペース**]、[**日付**]、および [**エラー**] のリストにはチェックボックスが含まれており、これを使用して対応する 1 つ以上のエラーのエラー・ログを削除できます。削除するものにチェックを付けて、[**削除**] ボタンを選択します。

15.4.3 %ERN を使用したアプリケーション・エラー・ログの表示

%ERN ユーティリティは、%ETN エラー・トラップ・ユーティリティによって記録されたアプリケーション・エラーを検証します。%ERN は、現在のネームスペースについて記録されているすべてのエラーを返します。

%ERN ユーティリティを使用するには、以下の手順を実行します。

1. ターミナル・プロンプトで「DO ^%ERN」と入力します。このユーティリティの名前では大文字と小文字が区別されます。ユーティリティ内でのプロンプトへの応答では、大文字と小文字は区別されません。いずれのプロンプトでも、「?」を入力すると、そのプロンプトの構文オプションが一覧表示され、「?L」を入力すると、定義されている値がすべて一覧表示されます。Enter キーを押すと、前のレベルに戻ることができます。

2. `For Date:` プロンプトには、エラーが発生した日付を入力します。%DATE ユーティリティに使用可能ないずれの日付形式でも使用できます。年を省略すると、現在の年であると見なされます。このプロンプトからは、日付とその日付で記録されているエラーの数が返されます。または、以下の構文を使用して、このプロンプトからエラーのリストを取得できます。

   - 「?L」を入力すると、エラーが生じた日付すべてのリストが、日付の新しい順に、記録済みエラーの数と共に表示されます。(T) 列は、何日前かを表します。(T) = 今日で、(T-7) = 7 日前です。対象の日のすべてのエラーにユーザ・コメントが定義されている場合、そのコメントが角かっこに囲まれて表示されます。リストの後に、`For Date:` プロンプトが再表示されます。日付または「T-*n*」を入力できます。

   - 「[text」を入力すると、部分文字列 text を含むすべてのエラーが一覧表示されます。「<text」を入力すると、エラー名コンポーネントに部分文字列 text を含むすべてのエラーが一覧表示されます。「^text」を入力すると、エラー位置コンポーネントに部分文字列 text を含むすべてのエラーが一覧表示されます。リストの後に、`For Date:` プロンプトが再表示されます。日付を入力します。

3. `Error:` プロンプトには、確認したいエラーの整数番号を入力します。対象の日の最初のエラーの場合は「1」、2 つ目のエラーの場合は「2」というように入力します。または、疑問符 (?) を入力すると、取得可能な応答のリストが表示されます。このユーティリティは、エラーに関する情報として、エラー名、エラー位置、時間、システム変数値、およびエラー発生時に実行されていたコード行を表示します。

Error: プロンプトに「*」を入力すると、コメントを表示できます。「*」を入力すると、対象の日のエラーすべてに適用されている現在のユーザ定義のコメントが表示されます。次に、そのすべてのエラーに適用されている既存のコメントに置き換わる新しいコメントを入力するよう求められます。

4. Variable: プロンプトには、変数に関する情報を取得するためのさまざまなオプションを指定できます。ローカル変数の名前 (添え字なしまたは添え字付き) を指定した場合、%ERN は、スタック・レベルとその変数の値 (定義されている場合)、およびそのすべての下位ノードを返します。グローバル変数、プロセス・プライベート変数、または特殊システム変数を指定することはできません。

   「?」を入力すると、Variable: プロンプトで指定できるその他の構文オプションが一覧表示されます。

   - *A : Variable: プロンプトに指定した場合、Device: プロンプトが表示されます。Return キーを押すと、現在のターミナル・デバイスに結果が表示されます。
   - *V : Variable: プロンプトに指定した場合、Variable(s): プロンプトが表示されます。このプロンプトには、添え字なしのローカル変数を 1 つ、または添え字なしのローカル変数のコンマ区切りのリストを指定します。添え字付きの変数は拒否されます。これに応じて、%ERN から Device: プロンプトが表示されます。Return キーを押すと、現在のターミナル・デバイスに結果が表示されます。%ERN は、指定された各変数の値 (定義されている場合) とそのすべての下位ノードを返します。
   - *L : Variable: プロンプトに指定した場合、変数を現在のパーティションにロードします。すべてのプライベート変数を (パブリックとして) ロードしてから、ロードされたプライベート変数と競合しないすべてのパブリック変数をロードします。

16
コマンド行ルーチンのデバッグ

この章では、InterSystems IRIS® Data Platform アプリケーションのテストとデバッグの手法について説明します。アプリケーション開発で重要な部分は、ルーチンのデバッグ、つまりプログラム・コードのテストと修正です。InterSystems IRIS には、ルーチンをデバッグする 2 つの方法があります。

- ルーチン・コードで BREAK コマンドを使用して実行を中断し、実行内容を検証します。
- ZBREAK コマンドで ObjectScript デバッガを呼び出して実行を中断し、コードと変数の両方を検証できます。

InterSystems IRIS には、この章で説明するように、ルーチンを中断し、完全なデバッグ機能をサポートするシェルに入る機能が含まれています。InterSystems IRIS には保護されたデバッグ・シェルも含まれています。これには、割り当てられた特権の超越や迂回をユーザができないという利点があります。

16.1 保護されたデバッグ・シェル

保護されたデバッグ・シェルは、機密データへのアクセスの制御に関して優れています。ユーザは、環境によって、変数のステッピングや表示などの基本的なデバッグを実行できますが、ルーチンの実行パスや結果を変更するいかなる操作も許可されません。これによって、操作、悪意のあるロール・エスカレーション、より高い権限で実行するコードの挿入などの問題を引き起こす可能性のあるアクセスを防止できます。

既定では、デバッグ・プロンプトのユーザは、現在のレベルの特権を保持します。デバッグ・プロンプトの保護されたシェルを有効にし、このことによってユーザが発行できるコマンドを制限するには、そのユーザに対して保護されたデバッグ・シェルを有効にする必要があります。

現在のユーザに対して有効にした場合、保護されたデバッグ・シェルは、BREAK コマンドが実行された時点、ブレークポイントまたはウォッチポイントが検出された時点、または捕えられていないエラーが発生した時点で開始されます。

保護されたデバッグ・シェル内では、ユーザは以下のものを実行できません。

- 変数を変更する可能性があるコマンド。
- 変数を変更する可能性がある関数。
- 他のルーチンを呼び出す可能性があるコマンド。
- ルーチンのフローまたは環境に影響を及ぼすコマンド。

保護されたデバッグ・シェル内で、制限されたコマンドまたは関数をユーザが実行しようとすると、InterSystems IRIS はそれぞれ <COMMAND> エラーまたは <FUNCTION> エラーをスローします。

16.1.1 制限されるコマンドと関数

この節では、保護されたデバッグ・シェル内で制限されるアクティビティのリストを示します。

- 制限される ObjectScript コマンド
- 制限される ObjectScript 関数
- 制限されるオブジェクト構成

16.1.1.1 制限される ObjectScript コマンド

保護されたデバッグ・シェルで制限される ObjectScript コマンドは以下のとおりです。

- CLOSE
- DO
- FOR
- 引数付きの GOTO
- KILL
- LOCK
- MERGE
- OPEN
- QUIT
- READ
- RETURN
- SET
- TCOMMIT
- TROLLBACK
- TSTART
- VIEW
- XECUTE
- ZINSERT
- ZKILL
- ZREMOVE
- ZSAVE
- ZW と ZZDUMP 以外のユーザ・コマンド

16.1.1.2 制限される ObjectScript 関数

保護されたデバッグ・シェルで制限される ObjectScript 関数は以下のとおりです。

- $CLASSMETHOD
- $COMPILE

- $DATA(,var) ー 引数が 2 つもののみ
- $INCREMENT
- $METHOD
- $ORDER(,,var) ー 引数が 3 つもののみ
- $PROPERTY
- $QUERY(,,var) ー 引数が 3 つもののみ
- $XECUTE
- $ZF
- $ZSEEK
- 任意の外部関数

16.1.1.3 制限されるオブジェクト構成

いかなるメソッドまたはプロパティの参照も許可されません。プロパティ参照は、propertyGet メソッドを起動する可能性があるため制限されます。制限されるオブジェクト・メソッドおよびプロパティ構文の構成の例の一部を以下に示します。

- #class(classname).ClassMethod()
- oref.Method()
- oref.Property
- $SYSTEM.Class.Method()
- ..Method()
- ..Property

注釈　参照によって変数の受け渡しを行わない場合でも、メソッドによってパブリック変数が変更される可能性があります。プロパティ参照は propGet メソッドを起動する可能性があるため、いかなるプロパティ・アクセスも許可されません。

16.2 ObjectScript デバッガによるデバッグ

ObjectScript デバッガを使用すると、ルーチンのコードにデバッグ・コマンドを直接挿入し、ルーチンをテストできます。その後、コードの実行時にコマンドを発行し、アプリケーションの実行状況と処理フローをテストできます。以下は主な機能です。

- ZBREAK コマンドを使用してコードにブレークポイントを設定し、その場所に達したときに特定の動作を実行します。
- ローカル変数にウォッチポイントを設定し、その変数の値が変更されたときに特定の動作を実行します。
- ブレークポイントとウォッチポイントの実行中は、別々のウィンドウで InterSystems IRIS とやり取りします。
- 実行パスが変更されたときは常に実行をトレースし、その記録を (ターミナルあるいは別のデバイスに) 出力します。
- 実行スタックを表示します。
- デバイスでアプリケーションを実行中、デバッグの入出力はセカンド・デバイスで行われます。これにより、InterSystems IRIS アプリケーションを全画面で実行しながら、アプリケーションのターミナル入出力を中断せずにデバッグできます。

16.2.1 ブレークポイントとウォッチポイントの使用法

ObjectScript デバッガには、プログラムの実行を中断する 2 つの方法があります。

- ブレークポイント
- ウォッチポイント

ブレークポイントは、ZBREAK コマンドで指定する InterSystems IRIS ルーチン内の位置です。実行中のルーチンが指定行に達すると、InterSystems IRIS はルーチンの実行を中断します。オプションとして、定義したデバッグを実行する場合もあります。最大 20 個のルーチンまでブレークポイントを設定できます。また、1 つのルーチン内に最大 20 個までのブレークポイントを設定できます。

ウォッチポイントは、ZBREAK コマンドで識別する変数です。SET コマンドあるいは KILL コマンドで値を変更すると、ルーチンの実行またはZBREAK コマンド内で定義したデバッグの実行、あるいはその両方を中断できます。最大 20 個のウォッチポイントを設定できます。

定義したブレークポイントとウォッチポイントは、1 つのセッション内でのみ維持されます。したがって、ブレークポイントとウォッチポイントの定義を 1 つのルーチンあるいは XECUTE コマンド文字列に格納すると、セッション間でそれらを簡単に元に戻せるため便利です。

16.2.2 ブレークポイントとウォッチポイントの設定

ZBREAK コマンドを使用して、ブレークポイントとウォッチポイントを設定します。

16.2.2.1 構文

```
ZBREAK location[:action:condition:execute_code]
```

以下を示します。

引数	説明
location	必須項目。(ブレークポイントを設定する)コード位置、(ウォッチポイントを設定する)ローカル変数あるいはシステム変数を指定します。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。
action	オプション – 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) ときの動作を指定します。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。動作は、大文字あるいは小文字で指定できますが、引用符で囲む必要があります。
condition	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。 ・ condition が True (1) の場合、action が実行されます。 ・ condition が False の場合、action は実行されず、execute_code 内のコードも実行されません。 condition が未指定の場合、既定では True です。
execute_code	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。このコードは、動作が実行される前に実行されます。特殊なシステム変数 $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。

注釈　　疑問符 (?) を付けて ZBREAK を使用すると、ヘルプが表示されます。

16.2.2.2 コード位置でのブレークポイントの設定

$TEXT 関数への呼び出しで使用するルーチン行参照として、コード位置を指定します。実行がコードの指定した位置に到達した場合は常に、コード行が実行される前にブレークポイントが発生します。ルーチン名を指定しない場合、InterSystems IRIS は、現在のルーチンを参照しているものと見なします。

16.2.2.3 ブレークポイント実行コード内の引数なしの GOTO

引数なしの GOTO は、ブレークポイントの実行コードで使用できます。これは、デバッガの Break プロンプトで引数なしの GOTO を実行するのと同じ効果があるため、次のブレークポイントまで引き続き実行されます。

例えば、テスト中のルーチンが現在のネームスペースにある場合、位置の値は以下のようになります。

値	ブレーク位置
label^rou	ルーチン rou にある行ラベル label の行の前で中断
label+3^rou	ルーチン rou にある行ラベル label から 3行目の前で中断
+3^rou	ルーチン rou の 3 行目で中断

テスト中のルーチンが現在メモリにロードされている場合 (つまり、暗黙あるいは明示的に ZLOAD が実行された場合)、以下のような位置の値を使用できます。

値	ブレーク位置
label	label の行ラベルの前で中断
label+3	label から 3 行目の前で中断
+3	3 行目の前で中断

16.2.2.4 ローカル変数名とシステム変数名でのウォッチポイントの設定

以下の状況で、ローカル変数名を使用してウォッチポイントを発生させることができます。

- ローカル変数を生成したとき
- SET コマンドがローカル変数値を変更したとき
- KILL コマンドがローカル変数を削除したとき

変数名は、*a のようにアスタリスクが前に付きます。

配列変数名を指定すると、ObjectScript デバッガはすべての下位ノードを監視します。例えば、配列 a にウォッチポイントを設定し、これを a(5) あるいは a(5,1) に変更すると、ウォッチポイントがトリガされます。

ウォッチポイントの設定時、既存の変数は必要ありません。

以下の特殊なシステム変数も使用できます。

システム変数	トリガ・イベント
$ZERROR	エラー発生時は常に、エラー・トラップの呼び出し前にトリガされます。
$ZTRAP	エラー・トラップが設定、あるいはクリアされる際は常にトリガされます。
$IO	明示的に SET コマンドを実行するとトリガされます。

16.2.2.5 Action 引数の値

以下のテーブルは、ZBREAK の action 引数に使用できる値の説明です。

引数	説明
"B"	既定値です。"T" 動作を含む場合を除き、ZBREAK *a:"TB" のように明示的に "B" 動作を組み込み、ブレークを起こす必要があります。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。その後、ターミナル・プロンプトが表示され、やり取りが可能となります。引数なしの GOTO コマンドで実行を再開します。
"L"	GOTO コマンドがシングル・ステップの実行を開始し、各行の最初で停止する以外は、"B" と同じです。DO コマンド、ユーザ定義関数、XECUTE コマンドに遭遇した場合、シングル・ステップ・モードは、そのコマンドあるいは関数が完了するまで一時中断します。
"L+"	GOTO コマンドがシングル・ステップの実行を開始し、各行の最初で停止する以外は、"B" と同じです。DO コマンド、ユーザ定義関数、XECUTE コマンドは、シングル・ステップ・モードを中断しません。
"S"	GOTO コマンドがシングル・ステップの実行を開始し、各コマンドの最初で停止する以外は、"B" と同じです。DO コマンド、ユーザ定義関数、FOR コマンド、XECUTE コマンドに遭遇した場合、シングル・ステップ・モードは、そのコマンドあるいは関数が完了するまで一時中断します。

引数	説明
"S+"	GOTO コマンドがシングル・ステップの実行を開始し、各コマンドの最初で停止する以外は、"B" と同じです。DO コマンド、ユーザ定義関数、FOR コマンド、XECUTE コマンドは、シングル・ステップ・モードを中断しません。
"T"	他の引数と共に使用できます。トレース・デバイスにトレース・メッセージを出力します。この引数は、後述の ZBREAK /TRACE:ON コマンドでトレースを ON に設定した後にのみ機能します。トレース・デバイスは、ZBREAK /TRACE コマンドで定義しない限り、主デバイスになります。この引数をブレークポイントで使用すると、TRACE: ZBREAK at label2^rou2 というメッセージが表示されます。この引数をウォッチポイントで使用すると、監視中の変数名と実行中のコマンド名を指定するトレース・メッセージが表示されます。例えば次のコードの場合、変数 a は監視され、ルーチン test の行 test+1 で変更されました。TRACE: ZBREAK SET a=2 at test+1^test。"T" 動作を組み込む場合、ZBREAK *a:"TB" に "B" 動作も明示的に組み込む必要があります。これにより、実際のブレークが発生します。
"N"	このブレークポイント/ウォッチポイントでは、何も動作しません。condition 式は常に評価され、execute_code が実行されているかどうかを判定します。

16.2.2.6 ZBREAK の例

以下の例では、ローカル変数 a が削除されると常に実行を中断するウォッチポイントを設定します。動作は何も指定されていないため、"B" が指定されているものとします。

```
ZBREAK *a::"'$DATA(a)"
```

以下の例は、(ルーチン内から発行されるコマンドではなく) ダイレクト・モードの ObjectScript コマンドに対して動作する、上記のウォッチポイントを示しています。キャレット (^) は、実行を中断させるコマンドを示します。

Terminal

```
USER>KILL a
KILL a
^
<BREAK>
USER 1s0>
```

以下の例は、実行を中断し、label2^rou 行の先頭にシングル・ステップ・モードを設定するブレークポイントです。

```
ZBREAK label2^rou:"L"
```

以下の例は、ルーチン実行時にどのようにブレークが実行されるかを示しています。キャレット (^) は、実行が中断した位置を示します。

Terminal

```
USER>DO ^rou
label2 SET x=1
 ^
<BREAK>label2^rou
USER 2d0>
```

以下の例では、"N" 動作のため、行 label3^rou のブレークポイントは実行を中断しません。しかし、行 label3^rou に達した際に x<1 の場合、flag を x に設定します。

```
ZBREAK label3^rou:"N":"x<1":"SET flag=x"
```

以下の例は、a の値が変更されるたびに、^GLO のコードを実行するウォッチポイントを設定します。二重コロンは、condition 引数がないことを示しています。

```
ZBREAK *a:"N"::"XECUTE ^GLO"
```

以下の例は、b の値が変更されるたびに、トレース・メッセージを表示するウォッチポイントを設定します。トレース・メッセージは、トレース・モードが ZBREAK /TRACE:ON コマンドで ON になっている場合にのみ表示されます。

```
ZBREAK *b:"T"
```

以下の例は、変数 a が 5 に設定されると、シングル・ステップ・モードで実行を中断するウォッチポイントを設定します。

```
ZBREAK *a:"S":"a=5"
```

以下の例は、ブレークの発生時、キャレット記号 (^) は、変数 a を 5 に設定するコマンドを指定します。

Terminal

```
USER>DO ^test
FOR i=1:1:6 SET a=a+1
 ^
<BREAK>
test+3^test
USER 3f0>WRITE a
5
```

16.2.3 ブレークポイントとウォッチポイントを無効にする

以下のいずれかを無効にできます。

- 特定のブレークポイントとウォッチポイント
- すべてのブレークポイントあるいはウォッチポイント

16.2.3.1 特定のブレークポイントとウォッチポイントを無効にする

マイナス記号をブレークポイントあるいはウォッチポイントの位置指定の前に置くと、ブレークポイントあるいはウォッチポイントを無効にできます。以下のコマンドは、位置 label2^rou で事前に指定されたブレークポイントを無効にします。

```
ZBREAK -label2^rou
```

無効になったブレークポイントは“消失”しますが、その定義は維持されます。したがって、プラス記号をその前に置くと、無効にされたブレークポイントを有効にできます。以下のコマンドは、以前無効にされたブレークポイントを有効にします。

```
ZBREAK +label2^rou
```

16.2.3.2 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する

位置を指定せずにプラス記号あるいはマイナス記号を使用して、すべてのブレークポイントまたはウォッチポイントを無効できます。

記号	説明
ZBREAK -	定義済みの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。
ZBREAK +	定義済みのすべてのブレークポイントとウォッチポイントを有効にします。

16.2.4 ブレークポイントとウォッチポイントの実行を遅らせる

指定した数の反復の間、ブレークポイント/ウォッチポイントの実行を遅らせることもできます。実行する度ではなく定期的にブレークさせるように、ループ内にコード行を記述している場合もあります。このためには、通常どおりにブレークポイントを設定した後、位置の引数に続くカウントを無効にします。

以下の ZBREAK コマンドは、100 回反復される間、label2^rou のブレークポイントを無効にします。101 回目にこの行が実行されると、指定したブレークポイントの動作が発生します。

```
ZBREAK label2^rou       ; establish the breakpoint
ZBREAK -label2^rou#100  ; disable it for 100 iterations
```

重要　　行が繰り返し実行されても、この行には FOR コマンドが含まれるため、遅らせたブレークポイントはデクリメントされません。

16.2.5 ブレークポイントとウォッチポイントの削除

次のように、ブレークポイント/ウォッチポイントの位置の前にマイナス記号を 2 つ付けて、ブレークポイントまたはウォッチポイントを個別に削除できます。

```
ZBREAK --label2^rou
```

ブレークポイント/ウォッチポイントを削除した後は、それを再度定義することにより削除したポイントをリセットできます。

すべてのブレークポイントを削除するには、以下のコマンドを発行します。

```
ZBREAK /CLEAR
```

このコマンドは、InterSystems IRIS プロセスが停止すると、自動的に実行されます。

16.2.6 シングル・ステップ・ブレークポイントの動作

シングル・ステップ実行を使用して、コードの各行あるいは各コマンドの最初で実行を停止できます。シングル・ステップ・ブレークポイントを設定し、各ステップで実行される動作と実行コードを指定できます。以下の構文を使用して、シングル・ステップ・ブレークポイントを定義します。

```
ZBREAK $:action[:condition:execute_code]
```

他のブレークポイントと異なり、ZBREAK $ はブレークを引き起こしません。ブレークは、シングル・ステップとして自動的に発生するからです。ZBREAK $ を使用すると、ルーチンのステップ実行時にデバッガがブレークする各ポイントで、動作を指定しコードを実行できます。これは、行あるいはコマンドの実行をトレースするのに特に役立ちます。例えば以下のコードは、アプリケーション ^TEST で実行された行をトレースします。

Terminal

```
USER>ZBREAK /TRACE:ON
USER>BREAK "L+"
USER>ZBREAK $:"T"
```

"T" 単体で (つまり、その他のアクション・コードなしで) 指定すると、通常自動的に発生するシングル・ステップ・ブレークが抑制されます (また、他のアクション・コードを使用するかどうかにかかわらず、"N" アクション・コードを指定すれば、シングル・ステップ・ブレークが抑制されます)。

トレースとブレークの両方を発生させるには、以下の方法でシングル・ステップ・ブレークポイントの定義を設定します。

Terminal

```
USER>ZBREAK $:"TB"
```

16.2.7 実行のトレース

以下の ZBREAK 形式を使用して、ZBREAK コマンドの "T" 動作を有効にするかどうかを制御できます。

ZBREAK /TRACE:state[:device]

state には、以下を指定できます。

状態	説明
ON	トレース可能
OFF	トレース不可能
ALL	次のコードと同等なコードを実行する場合、アプリケーションのトレース可能: ZBREAK /TRACE:ON[:device] BREAK "L+" ZBREAK $:"T"

device で ALL あるいは ON 状態キーワードを使用する場合、トレース・メッセージは、主デバイスではなく、指定されたデバイスに転送されます。デバイスを開いていない場合、InterSystems IRIS は、WRITE オプションと APPEND オプションが付いたシーケンシャル・ファイルとしてそのデバイスを開こうとします。

OFF 状態キーワードを使用してデバイスを指定すると、現在ファイルが開いている場合に InterSystems IRIS はそのファイルを閉じます。

注釈　ZBREAK /TRACE:OFF により、ZBREAK /TRACE:ALL で設定したシングル・ステップ・ブレークポイントの定義が削除、または無効にされることはありません。また、ZBREAK /TRACE:ALL で設定した“L+”シングル・ステップが消去されることもありません。また、ZBREAK --$ と BREAK "C" コマンドを発行し、シングル・ステップを削除する必要があります。あるいはその代わりに、BREAK "OFF" シングル・コマンドを使用して、プロセスのデバッグをすべて無効にできます。

トレース・メッセージは、“T”動作に対応するブレークポイントで生成されます。例外として、トレース・メッセージの形式は、すべてのブレークポイントで以下のようになります。

Trace: line_reference での ZBREAK

line_reference は、参照するブレークポイントの行です。

コマンドでステップを実行する場合、トレース・メッセージの形式は、シングル・ステップのブレークポイントと若干異なります。

Trace: line_reference source_offset での ZBREAK

line_reference は参照するブレークポイントの行で、source_offset は、ブレークが発生したソース行位置への 0 ベースのオフセットです。

オペレーティング・システムの注釈

- Windows ― 別のデバイスへのトレース・メッセージは、COM1: など COM ポートに接続されているターミナル・デバイスに対して、Windows プラットフォームでサポートされています。コンソールあるいはターミナル・ウィンドウは使用できません。トレース・デバイスへのシーケンシャル・ファイルを指定できます。
- UNIX® ― 以下の方法で、UNIX® プラットフォームで別のデバイスにトレース・メッセージを送信します。

  1. /dev/tty01 にログインします。

2. 次の tty コマンドを入力し、デバイス名を検証します。

```
$ tty
/dev/tty01
```

3. 以下のコマンドを発行し、デバイス競合を避けます。

```
$ exec sleep 50000
```

4. 作業ウィンドウに戻ります。
5. InterSystems IRIS を開始します。
6. 以下のトレース・コマンドを発行します。

```
ZBREAK /T:ON:"/dev/tty01"
```

7. プログラムを実行します。

   “T” 動作を実行するブレークポイントあるいはウォッチポイントを設定すると、/dev/tty01 に接続されたウィンドウに、トレース・メッセージが表示されます。

16.2.7.1 トレース・メッセージ形式

ブレークポイントをコードに設定すると、以下のメッセージが表示されます。

```
Trace: ZBREAK at label2^rou2
```

ウォッチポイント変数を設定すると、以下のメッセージが表示されます。

```
Trace: ZBREAK SET var=val at label2^rou2
Trace: ZBREAK SET var=Array Val at label2^rou2
Trace: ZBREAK KILL var at label2^rou2
```

- var は、監視される変数です。
- val は、その変数に設定された新しい値です。

NEW コマンドを発行すると、トレース・メッセージは受け取りません。しかし、NEW レベルの変数で次に SET あるいは KILL を発行すると、変数のトレースが実行されます。変数がルーチンに参照渡しされると、その名前が変更された場合も、変数は引き続き追跡されます。

16.2.8 割り込みキーと Break コマンド

通常、割り込みキー (一般的に CTRL-C) を続けて押すと、トラップ可能な (<INTERRUPT>) エラーが生成されます。<INTERRUPT> エラーではなく、ブレークを発生させる割り込み処理を設定するには、ZBREAK コマンド **ZBREAK /INTERRUPT:Break** を使用します。

これにより、デバイスのアプリケーション・レベルで割り込みが無効の場合も、割り込みキーを押したときにブレークが発生します。

ターミナルから読み取り中に割り込みキーを押すと、ブレーク・モード・プロンプトを表示するため、Enter キーを押す必要があります。ブレークではなくエラーを生成するよう割り込み処理をリセットするには、コマンド **ZBREAK /INTERRUPT:NORMAL** を発行します。

16.2.9 現在のデバッグ環境情報の表示

定義されたすべてのブレークポイントあるいはウォッチポイントを含め、現在のデバッグ環境の情報を表示するには、引数なしの ZBREAK コマンドを発行します。

引数なしの ZBREAK コマンドは、以下のデバッグ環境の状態を示します。

- CTRL-C がブレークを生じさせるかどうか
- "T" 動作で指定されたトレースの出力が、ZBREAK コマンドで表示されるかどうか
- 有効/無効状態、動作、条件、実行可能コードを指定するフラグを持つ、すべての定義済みブレークポイントの位置
- 有効/無効状態、動作、条件、実行可能コードを指定するフラグを持つウォッチポイントのすべての変数

このコマンドからの出力は、デバッグ・デバイスとして定義されたデバイスに表示されます。ZBREAK /DEBUG コマンドで異なるデバッグ・デバイスを定義しない限り、ユーザの主デバイスに出力されます。デバイスの定義は、"デバッグ・デバイスの使用法" のセクションで説明しています。

以下のテーブルは、各ブレークポイントとウォッチポイントに提供されるフラグの説明です。

表示セクション	意味
ブレークポイント/ウォッチポイントの識別	ブレークポイントのルーチンの行ウォッチポイントのローカルの変数
F:	ZBREAK コマンドで定義された動作タイプの情報を提供するフラグです。
S:	ZBREAK -コマンドで定義されたブレークポイント/ウォッチポイントの実行を遅らせる回数です。
C:	ZBREAK コマンドで設定された Condition 引数です。
E:	ZBREAK コマンドで設定された Execute_code 引数です。

以下のテーブルは、ブレークポイント/ウォッチポイントの表示で F: 値をどのように解釈するかを示しています。F: 値は、最初の列にある適切な値のリストです。

値	意味
E	有効なブレークポイント/ウォッチポイント
D	無効なブレークポイント/ウォッチポイント
B	ブレークの実行
L	"L" の実行
L+	"L+" の実行
S	"S" の実行
S+	"S+" の実行
T	トレース・メッセージの出力

16.2.9.1 既定表示

最初に InterSystems IRIS を開いて ZB を使用するとき、出力は以下のようになります。

Terminal

```
USER>ZBREAK
BREAK:
No breakpoints
No watchpoints
```

これは、以下を示します。

- トレースの実行は OFF です。
- CTRL-C が押されると、ブレークは無効になります。
- ブレークポイント/ウォッチポイントは未定義です。

16.2.9.2 既存のブレークポイントとウォッチポイントの表示

この例では、2 つのブレークポイントと 1 つのウォッチポイントの定義を示しています。

Terminal

```
USER>ZBREAK +3^test:::{WRITE "IN test"}
USER>ZBREAK -+3^test#5
USER>ZBREAK +5^test:"L"
USER>ZBREAK -+5^test
USER>ZBREAK *a:"T":"a=5"
USER>ZBREAK /TRACE:ON
USER>ZBREAK
BREAK: TRACE ON
+3^test F:EB S:5 C: E:"WRITE ""IN test"""
+5^test F:DL S:0 C: E:
a F:ET S:0 C:"a=5" E:
```

最初の 2 つの ZBREAK コマンドは、ブレークポイントの遅延を定義しています。その次の 2 つの ZBREAK は無効なブレークポイントを、5 番目の ZBREAK はウォッチポイントを定義しています。6 番目の ZBREAK コマンドはトレースの実行を有効にし、引数のない最後の ZBREAK コマンドは、現在のデバッグ設定情報を表示します。

例えば、ZBREAK は以下を表示します。

- トレースは ON です。
- CTRL-C が押されると、ブレークは無効になります。

その後出力は、2 つのブレークポイントと 1 つのウォッチポイントを示します。

- 最初のブレークポイントの F フラグは “EB”、S フラグは 5 です。つまり、ブレークポイントは、行の 5 回目の実行で発生します。E フラグは、ブレーク用のターミナル・プロンプトが表示される前に実行される実行可能コードを表示します。
- 2 番目のブレークポイントの F フラグは “DL” です。これは、ブレークポイントの無効を意味しますが、有効の場合ブレークが発生し、ブレークポイント位置の後ろの各コード行をシングル・ステップで実行します。
- ウォッチポイントの F フラグは “ET” です。これは、ウォッチポイントが有効であることを意味します。トレース実行が ON の場合、トレース・メッセージがトレース・デバイスに表示されます。トレース・デバイスを定義していない場合、トレース・デバイスは主デバイスになります。
- C フラグは、condition が True の場合にのみトレースが表示されることを意味します。

16.2.10 デバッグ・デバイスの使用法

デバッグ・デバイスは、以下のようなデバイスです。

- ZBREAK コマンドは、デバッグ環境情報を表示します。

- ターミナル・プロンプトは、ブレーク発生時に表示されます。

注釈　Windows プラットフォームでは、別のデバイスへのトレース・メッセージは、COM1: など COM ポートに接続されているターミナル・デバイスでのみサポートされています。

InterSystems IRIS を実行すると、デバッグ・デバイスが主デバイスとして自動的に設定されます。いつでも、デバッグ入出力は、コマンド ZBREAK /DEBUG:"*device*" で代わりのデバイスに送信できます。

注釈　また、オペレーティング・システム固有の動作も実行可能です。

UNIX® システムで tty01 デバイスをブレーク状態にするには、以下のコマンドを発行します。

```
ZBREAK /D:"/dev/tty01/"
```

CTRL-C あるいはブレークポイント、ウォッチポイントがトリガされてブレークが発生すると、デバイスに接続されたウィンドウに表示されます。そのウィンドウが、アクティブ・ウィンドウになります。

デバイスが開いていない場合、自動的に OPEN コマンドが実行されます。デバイスが指定した位置で既に開いている場合、あらゆる既存の OPEN パラメータが優先されます。

重要　指定したデバイスがインタラクティブでない (例えばターミナル) 場合、ブレークから戻ることができません。しかし、システムでこの制約が強制されているわけではありません。

16.2.11 ObjectScript デバッガの例

最初に、以下に示す test という簡単なプログラムをデバッグします。目標は、変数 a に 1 を、変数 b に 2 を、変数 c に 3 を格納することです。

```
test; Assign the values 1, 2, and 3 to the variables a, b, and c
 SET a=1
 SET b=2
 SET c=3 KILL a WRITE "in test, at end"
 QUIT
```

しかし、test を実行すると、b と c のみが正しい値を取得します。

Terminal

```
USER>DO ^test
in test, at end
USER>WRITE
b=2
c=3
USER>
```

明らかにこのプログラムの問題は、変数 a が行 4 で KILL コマンドによって削除されることですが、この問題を特定するためにデバッガを使用する必要があると想定します。

ZBREAK コマンドを使用して、ルーチン test の各コード行 ("L" 動作) をシングル・ステップに設定できます。ステップを実行して a の値を書き込むことにより、4 行目に問題があることを判定します。

Terminal

```
USER>NEW
USER 1S1>ZBREAK
BREAK
No breakpoints
No watchpoints
USER 1S1>ZBREAK ^test:"L"
USER 1S1>DO ^test
SET a=1
^
<BREAK>test+1^test
```

```
USER 3d3>WRITE a
<UNDEFINED>^test
USER 3d3>GOTO
SET b=2
^
<BREAK>test+2^test
USER 3d3>WRITE a
1
USER 3d3>GOTO
SET c=3 KILL a WRITE "in test, at end"
^
<BREAK>test+3^test
USER 3d3>WRITE a
1
USER 3d3>GOTO
in test, at end
QUIT
^
<BREAK>test+4^test
USER 3d3>WRITE a
WRITE a
^
<UNDEFINED>^test
USER 3d3>GOTO
USER 1S1>
```

これでこの行を検証し、KILL a コマンドを発見できます。さらに複雑なコードでは、その行から ("S" 動作) コマンドでシングル・ステップを実行したい場合もあります。

問題が、DO、FOR、XECUTE コマンド、ユーザ定義関数で発生した場合、"L+" あるいは "S+" 動作を使用して、コードの下位レベルにある行あるいはコマンドからシングル・ステップを実行します。

16.2.12 ObjectScript デバッガ・エラーの理解

ObjectScript デバッガは、適切な InterSystems IRIS エラー・メッセージを持つ条件引数あるいは実行引数でエラーが発生すると、フラグを立てます。

エラーが execute_code 引数内に存在する場合、実行コードがエラー・メッセージの前に表示されると、実行コードは条件コードに囲まれます。条件特殊変数 ($TEST) は、常に実行コードの最後で 1 に設定されるため、残りのデバッガ処理コードは適切に動作します。制御がルーチンに戻ると、ルーチン内の $TEST 値はリストアされます。

サンプル・プログラム test に対し、以下の ZBREAK コマンドを発行するとします。

Terminal

```
USER>ZBREAK test+1^test:"B":"a=5":"WRITE b"
```

このプログラム test で変数 b は、行 test+1 で定義されていないため、エラーが発生します。以下のようなエラーが表示されます。

```
IF a=5 XECUTE "WRITE b" IF 1
^
<UNDEFINED>test+1^test
```

condition を定義しなかった場合、人工的に True の条件を実行コードの前後に定義できます。以下はその例です。

Terminal

```
USER>IF 1 WRITE b IF 1
```

16.3 BREAK コマンドによるデバッグ

InterSystems IRIS の BREAK コマンドには、3 つの形式があります。

- ルーチン・コードに挿入された引数なしの BREAK により、その場所においてブレークポイントが設定されます。コード実行中にこのブレークポイントと遭遇すると、実行が中断されて、ターミナル・プロンプトに戻ります。
- 文字列引数付きの BREAK では、行ごとまたはコマンドごとでのコードのステップ実行を可能にするブレークポイントを設定/削除します。
- 整数の引数付きの BREAK コマンドでは、CTRL-C によるユーザ割り込みを有効/無効化します。(詳細は "BREAK" コマンドを参照してください。)

16.3.1 ルーチンの実行を中断する引数なし BREAK の使用法

実行中のルーチンを中断し、プロセスをターミナル・プロンプトに戻すには、ルーチンで実行を一時的に中断させたい位置に、引数なしの BREAK を入力します。

InterSystems IRIS が BREAK に遭遇すると、以下のステップが実行されます。

1. 実行中のルーチンを中断します。
2. プロセスをターミナル・プロンプトに戻します。主デバイスの入出力リ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と、リダイレクトはデバッグ・プロンプトでオフになり、デバッグ・コマンドからの出力がターミナルに表示されます。

   これにより、ObjectScript コマンドを発行し、データを修正できます。また、エラーまたは追加の BREAK があっても、他のルーチンやサブルーチンを実行できます。デバッグ・ターミナル・プロンプトから ObjectScript コマンドを発行すると、このコマンドは直ちに実行されます。実行中のルーチンには挿入されません。このコマンド実行は、通常のターミナル・プロンプトの動作と同じです。1 つ違う点は、タブ文字によって続行されたコマンドがデバッグ・ターミナル・プロンプトから実行されることです。タブ文字によって続行されたコマンドは、通常のターミナル・プロンプトからは実行されません。

ルーチンが中断された位置で実行を再開するには、引数なしの GOTO コマンドを発行します。

コード内の引数なしの BREAK コマンドに後置条件を指定すると、ルーチンを変更しなくても、単に後置条件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。

```
CHECK 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。

詳細は、このドキュメントの "コマンド後置条件式" を参照してください。

16.3.2 ルーチンの実行を中断する引数付き BREAK の使用法

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。InterSystems IRIS では、コードのステップ実行を可能にする引数オプションがいくつか用意されています。コードのステップ実行は、シングル・ステップ (BREAK "S") またはコマンド行 (BREAK "L") を使用して行えます。これらの文字コード引数のすべてのリストについては、BREAK コマンドを参照してください。

BREAK "S" と BREAK "L" の一つの違いは、多数のコマンド行が複数ステップで構成されることにあります。このことは必ずしも明白になっているわけではありません。例えば、以下はすべて単一行 (および単一 ObjectScript コマンド) ですが、それぞれが 2 ステップで解析されます :SET x=1,y=2、KILL x,y、WRITE "hello",!、IF x=1,y=2。

BREAK "S" と BREAK "L" は、ラベル行、コメント、および TRY 文を無視します (ただし、両方ともに TRY ブロックの閉じ中括弧でブレークします)。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 でプロセスをターミナル・プロンプトに戻した場合、ブレークの状態はスタックされません。したがって、ブレークの状態を変更した場合、引数なしの GOTO を発行して実行中のルーチンに戻ったときに、その新規の状態が有効なままになります。

DO、XECUTE、FOR、またはユーザ定義関数が実行されるたびに、InterSystems IRIS はブレークの状態をスタックします。BREAK "C" を選択してブレークを OFF にすると、システムは DO、XECUTE、FOR、またはユーザ定義関数の最後にブレークの状態をリストアします。そうしない場合、InterSystems IRIS はスタックされた状態を無視します。

このように、下位のサブルーチン・レベルでブレークを有効にすると、ルーチンが上位のサブルーチン・レベルに戻った後、ブレークが継続されます。一方、上位レベルで有効だったブレークを下位サブルーチン・レベルで無効にした場合、上位レベルに戻ると、ブレークが再開されます。BREAK "C-" を使用して、すべてのレベルでブレークを無効にすることができます。

BREAK "L+" または BREAK "S+" を使用して、DO、XECUTE、FOR、またはユーザ定義関数内でブレークを有効にすることができます。

BREAK "L-" を使用すると、現在のレベルでのブレークが無効になりますが、前のレベルでの改行が有効になります。BREAK "S-" を使用すると、現在のレベルでのブレークが無効になりますが、前のレベルでのシングル・ステップ・ブレークが有効になります。

16.3.2.1 デバッグの停止

プロセスで作成されたすべてのデバッグを削除するには、BREAK "OFF" コマンドを使用します。このコマンドは、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 "OFF" コマンドを実行することは、以下の一連のコマンドを発行することと同じです。

ObjectScript

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

16.3.3 ターミナル・プロンプトのプログラム・スタック情報表示

BREAK コマンドがルーチンの実行を中断したとき、あるいはエラーが発生したとき、プログラム・スタックは、スタックされた情報の一部を維持します。このような状態になった場合、この情報の概要がターミナル・プロンプトの一部として表示されます (namespace>)。例えば、この情報は、USER 5d3> の形式で表示される場合があります。各要素は以下のとおりです。

文字	説明
5	5 つのスタック・レベルがあることを示します。スタック・レベルは DO、FOR、XECUTE、NEW、ユーザ定義関数の呼び出し、エラー状態、またはブレーク状態が原因となる可能性があります。
d	最後にスタックされた項目が DO であることを示します。
3	3 つの NEW 状態、パラメータ渡し、またはユーザ定義関数がスタックにあることを示します。NEW コマンド、パラメータ渡し、ユーザ定義関数がスタックされていない場合、この値は 0 です。

ターミナル・プロンプトの文字コードを以下のテーブルに示します。

テーブル 16-1: ターミナル・プロンプトのスタック・エラー・コード

プロンプト	定義
d	DO
e	ユーザ定義関数
f	FOR ループ
x	XECUTE
B	BREAK 状態
E	エラー状態
N	NEW 状態
S	サインオン状態

以下の例では、スタック・フレーム追加時において、コマンド行の文が結果のターミナル・プロンプトを伴って表示されています。

Terminal

```
USER>NEW
USER 1S1>NEW
USER 2N1>XECUTE "NEW  WRITE 123 BREAK"
<BREAK>
USER 4x1>NEW
USER 5B1>BREAK
<BREAK>
USER 6N2>
```

QUIT 1 を使用して、プログラム・スタックを戻すことができます。以下は、スタックの巻き戻し時におけるターミナル・プロンプトの例です。

Terminal

```
USER 6f0>QUIT 1  /* an error occurred in a FOR loop. */
USER 5x0>QUIT 1  /* the FOR loop was in code invoked by XECUTE. */
USER 4f0>QUIT 1  /* the XECUTE was in a FOR loop. */
USER 3f0>QUIT 1  /* that FOR loop was nested inside another FOR loop. */
USER 2d0>QUIT 1  /* the DO command was used to execute the program. */
USER 1S0>QUIT 1  /* sign on state. */
USER>
```

16.3.4 FOR ループおよび 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 (通常はカウンタのインクリメント) 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・モジュールに対するシングル・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。

16.3.5 BREAK あるいはエラー後の実行の再開

BREAK あるいはエラーの後にターミナル・プロンプトに戻ると、InterSystems IRIS は、BREAK あるいはエラーの原因となったコマンドの位置を記録します。その後、ターミナル・プロンプトで引数なしの GOTO を入力するだけで、次のコマンドで実行を再開できます。

Terminal

```
USER 4f0>GOTO
```

引数付きの GOTO を入力すると、以下のように、ブレークあるいはエラーが発生する同じルーチン内の別の行頭から実行を再開できます。

Terminal

```
USER 4f0>GOTO label3
```

また、異なるルーチンの行頭から実行を再開できます。

Terminal

```
USER 4f0>GOTO label3^rou
```

または、引数なしの QUIT コマンドで、プログラム・スタックをクリアすることもできます。

Terminal

```
USER 4f0>QUIT
USER>
```

16.3.5.1 サンプル・ダイアログ

以下はサンプルで使用されるルーチンです。

```
MAIN ; 03 Jan 2019 11:40 AM
 SET x=1,y=6,z=8
 DO ^SUB1 WRITE !,"sum=",sum
 QUIT

SUB1 ; 03 Jan 2019 11:42 AM
 SET sum=x+y+z
 QUIT
```

BREAK "L" により、ルーチン SUB1 でブレークは発生しません。

Terminal

```
USER>BREAK "L"
USER>DO ^MAIN
SET x=1,y=6,z=8
^
<BREAK>MAIN+1^MAIN
USER 2d0>GOTO
DO ^SUB1 WRITE !,"sum=",sum
^
<BREAK>MAIN+2^MAIN
USER 2d0>GOTO
sum=15
QUIT
^
<BREAK>MAIN+3^MAIN
USER 2d0>GOTO
USER>
```

BREAK"L+" により、ルーチン SUB1 でブレークも発生します。

Terminal

```
USER>BREAK "L+"
USER>DO ^MAIN
SET x=1,y=6,z=8
^
<BREAK>MAIN+1^MAIN
USER 2d0>GOTO
DO ^SUB1 WRITE !,"sum=",sum
^
<BREAK>MAIN+2^MAIN
USER 2d0>GOTO
SET sum=x+y+z
^
<BREAK>SUB1+1^SUB1
USER 3d0>GOTO
QUIT
^
<BREAK>SUB1+2^SUB1
USER 3d0>GOTO
sum=15
QUIT
^
<BREAK>MAIN+3^MAIN
USER 2d0>GOTO
USER>
```

16.3.6 ターミナル・プロンプトで使用する NEW コマンド

引数なしの NEW コマンドは、事実上すべての記号をシンボル・テーブルに格納します。したがって、空のシンボル・テーブルを実行できます。エラーあるいは BREAK の発生後に、このコマンドが特に役立ちます。

シンボル・テーブルの妨げにならずに他のルーチンを実行するには、ターミナル・プロンプトで引数なしの NEW コマンドを発行します。その後、システムは以下を実行します。

- プログラム・スタックに現在のフレームをスタックします。
- 新規スタック・フレームのターミナル・プロンプトを返します。

例えば以下のようになります。

Terminal

```
USER 4d0>NEW
USER 5B1>DO ^%T
3:49 PM
USER 5B1>QUIT 1
USER 4d0>GOTO
```

5B1> プロンプトは、システムが BREAK から入った現在のフレームをスタックしたことを示します。1 は、NEW コマンドが変数情報をスタックしたことを示し、この変数情報は、QUIT 1 を発行することで削除できます。実行を再開する場合は、QUIT 1 を発行して古いシンボル・テーブルをリストアし、GOTO を発行して実行を再開します。

NEW コマンド、パラメータ渡し、ユーザ定義関数を使用するたびに、システムは情報をスタックに格納し、後で現在のサブルーチンまたは XECUTE レベルの明示的または暗黙的 QUIT により特定の編集を削除して他の値をリストアする必要があることを示します。

NEW コマンド、パラメータ渡し、ユーザ定義関数が実行されたかどうか (その結果一部の変数がスタックされます)、また実行された場合、既存情報をどのスタックまで戻すかどうかがわかると役立ちます。

16.3.7 ターミナル・プロンプトで使用する QUIT コマンド

ターミナル・プロンプトから引数なしの QUIT コマンドを入力すると、すべての項目をプログラム・スタックから削除できます。

Terminal

```
USER 4f0>QUIT
USER>
```

プログラム・スタックからいくつかの項目のみを除外するには (例えば、現在実行中のサブルーチンから出て、前の DO レベルに戻る場合)、整数の引数で QUIT を使用します。例えば、QUIT 1 はプログラム・スタックの最後の項目を削除し、QUIT 3 は最後の 3 つの項目を削除します。以下はそのコード例です。

Terminal

```
9f0>QUIT 3
6d0>
```

16.3.8 InterSystems IRIS エラー・メッセージ

InterSystems IRIS は、<ERROR> のように、<> 山括弧内にエラー・メッセージを表示します。その後には、エラー発生時にルーチンが実行していた行の参照が続きます。キャレット記号 (^) は、行参照とルーチンを区切ります。また、エラー発生時に実行するコマンドの最初の文字の下に、キャレット記号と共に中間コード行が表示されます。例えば以下のようになります。

```
SET x=y+3 DO ^ABC
^
<UNDEFINED>label+3^rou
```

このエラー・メッセージは、rou ルーチンの label+3 行で、<UNDEFINED> エラー (変数 y への参照) が発生したことを示しています。この時点で、このメッセージも特殊変数 $ZERROR の値となります。

16.4 スタックを表示する %STACK の使用法

%STACK ユーティリティを使用して、以下を実行できます。

- プロセス実行スタックのコンテンツの表示
- NEW コマンドあるいはパラメータ渡しで “隠された” 値を含むローカル変数値の表示
- $IO や $JOB などのプロセス状態変数値の表示

16.4.1 %STACK の実行

以下のコマンドを入力して、%STACK を実行します。

Terminal

```
USER>DO ^%STACK
```

この例で示されているように、%STACK ユーティリティは、変数を持たない現在のプロセス・スタックを表示します。

```
Level  Type     Line                    Source
  1    SIGN ON
  2    DO                                ~DO ^StackTest
  3    NEW ALL/EXCL                      NEW (E)
  4    DO                 TEST+1^StackTest          SET A=1 ~DO TEST1 QUIT  ;level=2
  5    NEW                               NEW A
  6    DO                TEST1+1^StackTest          ~DO TEST2 ;level = 3
  7    ERROR TRAP                        SET $ZTRAP="TrapLabel^StackTest"
  8    XECUTE            TEST2+2^StackTest  ~XECUTE "SET A=$$TEST3()"
  9    $$EXTFUNC                ^StackTest ~SET A=$$TEST3()
 10    PARAMETER                         AA
 11    DIRECT BREAK      TEST3+1^StackTest          ~BREAK
 12    DO                       ^StackTest ~DO ^%STACK
```

現在の実行スタック表示の下で、%STACK により、スタックの表示動作を求める Stack Display Action プロンプトが表示されます。このプロンプトで疑問符 (?) を入力すると、ヘルプを表示できます。このプロンプトで Return キーを押すと、%STACK を終了できます。

16.4.2 プロセス実行スタックの表示

Stack Display Action で入力した内容によって、以下の 4 種類の形式で現在のプロセス実行スタックを表示できます。

- *F を入力すると、変数なしのスタックを表示
- *V を入力すると、特定のローカル変数付きスタックを表示
- *P を入力すると、すべてのローカル変数付きスタックを表示
- *A を入力すると、プロセス状態変数リストに先行するすべてのローカル変数付きスタックを表示

次に、%STACK により Display on Device プロンプトが表示されます。ここでは、この情報の表示先を指定できます。Return キーを押すと、この情報を現在のデバイスに表示できます。

16.4.2.1 変数なしスタックの表示

変数なしのプロセス実行スタックは、初めて %STACK ユーティリティを起動したとき、または Stack Display Action プロンプトで *F を入力したときに表示されます。

16.4.2.2 特定の変数付きスタックの表示

Stack Display Action プロンプトで、*V を入力します。これにより、このスタックで記録したいローカル変数の名前の入力を求められます。単一の変数、またはコンマで区切られた変数のリストを指定します。すべてのローカル変数の名前と値が返されます。以下の例では、変数 e が記録され、Return キーを押すと情報がターミナルに表示されます。

```
Stack Display Action: *V
Now loading variable information ...   2  done.
Variable(s): e
Display on
Device: <RETURN>
```

16.4.2.3 すべての定義済み変数付きスタックの表示

Stack Display Action プロンプトで *P を入力すると、すべての定義済みローカル変数の現在値と共に、プロセス実行スタックを表示できます。

16.4.2.4 状態変数を含むすべての変数付きスタックの表示

Stack Display Action プロンプトで *A を入力すると、すべての利用可能なレポートを表示できます。レポートは、以下の順序で発行されます。

- プロセス状態内部変数

- すべてのローカル変数の名前と値を伴ったプロセス実行スタック

16.4.3 スタックの表示の理解

スタックの各項目は、フレーム と呼ばれます。以下のテーブルは、各フレームに提供される情報を示します。

テーブル 16-2: %STACK ユーティリティ情報

見出し	説明
レベル	スタック内のレベルを識別します。スタック内の一番古い項目が 1 番です。関連付けられたレベル番号のないフレームは、そのすぐ上に表示されるレベルと同じになります。
タイプ	スタックのフレーム・タイプを識別します。タイプは次のようになります。DIRECT BREAK：BREAK コマンドが発生し、ダイレクト・モードに戻りました。DIRECT CALLIN：InterSystems IRIS コールイン・インタフェースを使用して、InterSystems IRIS プロセスが InterSystems IRIS 外部のアプリケーションから開始されました。DIRECT ERROR：エラーが発生し、ダイレクト・モードに戻りました。DO：DO コマンドが実行されました。ERROR TRAP：ルーチンが $ZTRAP を設定した場合、このフレームは、エラーによって実行が継続する位置を識別します。FOR：FOR コマンドが実行されました。NEW：NEW コマンドが実行されました。NEW コマンドが引数付きの場合、引数は表示されます。SIGN ON：InterSystems IRIS プロセスの実行が開始されました。XECUTE：XECUTE コマンドが実行されました。$XECUTE 関数が実行されました。$$EXTFUNC：ユーザ定義関数が実行されました。
行	使用可能な場合、label+offset^routine の形式で、フレームに関連付けられた ObjectScript ソース行を識別します。
ソース	使用可能な場合、ソース・コードを表示します。ソースが長すぎて表示領域に表示できない場合、水平スクロールを使用できます。デバイスが行表示対応の場合、ソースは折り返され、その後の行の先頭に “...” が付きます。

以下のテーブルは、各フレーム・タイプごとにレベル、行、ソース値が使用できるかどうかを示します。レベルの下の "No" は、レベル番号がインクリメントされず、表示されないことを示します。

テーブル 16-3: フレーム・タイプと使用可能な値

フレーム・タイプ	レベル	行	ソース
DIRECT BREAK	あり	あり	あり
DIRECT CALL IN	あり	なし	なし
DIRECT ERROR	あり	あり	あり
DO	あり	あり *	あり
ERROR TRAP	なし	なし	なし（しかし、新規 $ZTRAP 値は表示されます）
FOR	なし	あり	あり
NEW	なし	なし	NEW の形式（包括的または排他的）と影響を受けた変数を示します。

フレーム・タイプ	レベル	行	ソース
PARAMETER	なし	なし	仮パラメータ・リストを示します。パラメータが参照渡しされる場合は、同じメモリ位置を指すその他の変数を示します。
SIGN ON	あり	なし	いいえ
XECUTE	あり	あり *	あり
$$EXTFUNC	あり	あり *	あり
* 行の値は、ターミナル・プロンプトから呼び出された場合、空白になります。			

16.4.3.1 %STACK 表示からの移動

%STACK 表示が複数画面にわたる場合、画面の左下にプロンプト“-- more --”が表示されます。最後のページには、プロンプト“-- fini --”が表示されます。? を入力すると、%STACK 表示内での移動に使用するキー操作が表示されます。

```
- - - Filter Help - - -
<space> Display next page.
<return> Display one more line.
T Return to the beginning of the output.
B Back up one page (or many if arg>1).
R Redraw the current page.
/text Search for \qtext\q after the current page.
A View all the remaining text.
Q Quit.
? Display this screen
# specify an argument for B, L, or W actions.
L set the page length to the current argument.
W set the page width to the current argument.
```

“-- more --”あるいは“-- fini --”プロンプトが表示される場合は常に、上記に表示されたいずれかのコマンドを入力します。

B、L、W コマンドを入力する場合、コマンド文字の前に数値引数を入力します。例えば、2B と入力すると 2 ページ前に戻ります。また、20L を入力すると、20 行の長さにページが設定されます。

ページは、実際に表示される行数に合わせて設定してください。そうしない場合、ページを上下すると一部の行が表示されない可能性があります。既定のページの長さは 23 です。

16.4.3.2 特定のスタック・レベルでの変数の表示

指定されたスタック・フレーム・レベルに存在する変数を表示するには、“Stack Display Action”プロンプトで ?# と入力します。# には、スタック・フレーム・レベルを入力します。以下の例は、レベル 1 の変数を要求した場合の表示を示します。

```
Stack Display Action: ?1
The following Variables are defined for Stack Level: 1
E
Stack Display Action:
```

この情報を、**%SYS.ProcessQuery** VariableList クラス・クエリを使用して表示することもできます。

16.4.3.3 変数付きのスタック・レベルの表示

“Stack Display Action” プロンプトに ?? と入力すると、すべてのスタック・レベルで定義された変数を表示できます。?? と入力した場合に表示される例は、以下のとおりです。

```
Stack Display Action: ??
Now loading variable information ... 19
Base Stack Level: 5
A
Base Stack Level: 3
A B C D
Base Stack Level: 1
E
Stack Display Action:
```

16.4.3.4 プロセス状態変数の表示

$IO のようなプロセス状態変数を表示するには、″Stack Display Action″ プロンプトに *S と入力します。次の表に示すように、これらの定義された変数 (プロセス状態内部) が表示されます。

プロセス状態内部	ドキュメント
$D =	$DEVICE 特殊変数
$EC = ,M9,	$ECODE 特殊変数
$ES = 4	$ESTACK 特殊変数
$ET =	
$H = 64700,50668	$HOROLOG 特殊変数
$I = |TRM|:|5008	$IO 特殊変数
$J = 5008	$JOB 特殊変数
$K = $c(13)	$KEY 特殊変数
$P = |TRM|:|5008	$PRINCIPAL 特殊変数
$Roles = %All	$ROLES 特殊変数
$S = 268315992	$STORAGE 特殊変数
$T = 0	$TEST 特殊変数
$TL = 0	$TLEVEL 特殊変数
$USERNAME = glenn	$USERNAME 特殊変数
$X = 0	$X 特殊変数
$Y = 17	$Y 特殊変数
$ZA = 0	$ZA 特殊変数
$ZB = $c(13)	$ZB 特殊変数
$ZC = 0	$ZCHILD 特殊変数
$ZE = <DIVIDE>	$ZERROR 特殊変数
$ZJ = 5	$ZJOB 特殊変数
$ZM = RY¥Latin1¥K¥UTF8¥	$ZMODE 特殊変数
$ZP = 0	$ZPARENT 特殊変数

プロセス状態内部	ドキュメント
$ZR = ^||a	$ZREFERENCE 特殊変数
$ZS = 262144	$ZSTORAGE 特殊変数
$ZT =	$ZTRAP 特殊変数
$ZTS = 64700,68668.58	$ZTIMESTAMP 特殊変数
$ZU(5) = USER	$NAMESPACE
$ZU(12) = c:¥intersystems¥iris¥mgr¥	NormalizeDirectory()
$ZU(18) = 0	Undefined()
$ZU(20) = USER	UserRoutinePath()
$ZU(23,1) = 5	
$ZU(34) = 0	
$ZU(39) = USER	SysRoutinePath()
$ZU(55) = 0	LanguageMode()
$ZU(56,0) = $Id: //iris/2018.1.1/kernel/common/src/emath.c#1 $ 0	
$ZU(56,1) = 1349	
$ZU(61) = 16	
$ZU(61,30,n) = 262160	
$ZU(67,10,$J) = 1	JobType
$ZU(67,11,$J) = glenn	UserName
$ZU(67,12,$J) = TRM:	ClientNodeName
$ZU(67,13,$J) =	ClientExecutableName
$ZU(67,14,$J) =	CSPSessionID
$ZU(67,15,$J) = 127.0.0.1	ClientIPAddress
$ZU(67,4,$J) = 0^0^0	State
$ZU(67,5,$J) = %STACK	Routine
$ZU(67,6,$J) = USER	Namespace
$ZU(67,7,$J) = |TRM|:|5008	CurrentDevice
$ZU(67,8,$J) = 923	LinesExecuted
$ZU(67,9,$J) = 46	GlobalReferences
$ZU(68,1) = 0	NullSubscripts()
$ZU(68,21) = 0	SynchCommit()
$ZU(68,25) = 0	
$ZU(68,27) = 1	
$ZU(68,32) = 0	ZDateNull()

プロセス状態内部	ドキュメント
$ZU(68,34) = 1	AsynchError()
$ZU(68,36) = 0	
$ZU(68,40) = 0	SetZEOF()
$ZU(68,41) = 1	
$ZU(68,43) = 0	OldZU5()
$ZU(68,5) = 1	BreakMode()
$ZU(68,6) = 0	
$ZU(68,7) = 0	RefInKind()
$ZU(131,0) = MYCOMPUTER	
$ZU(131,1) = MYCOMPUTER:IRIS	
$ZV = IRIS for Windows (x86-64) 2018.1.0 (Build 527U) Tue Feb 20 2018 22:47:10 EST	$ZVERSION 特殊変数

16.4.3.5 スタックと変数の両方あるいはいずれかの出力

以下の動作を選択すると、出力デバイスを選択できます。

- *P
- *A
- 表示したい変数を選択した後に、*V

16.5 その他のデバッグ・ツール

デバッグ・プロセスで役立つツールが他にもあります。これには、以下のものがあります。

- $SYSTEM.OBJ.ShowReferences によるオブジェクトへの参照の表示
- エラー・トラップ・ユーティリティ ー %ETN および %ERN

16.5.1 $SYSTEM.OBJ.ShowReferences によるオブジェクトへの参照の表示

指定されたオブジェクトへの参照を格納したすべての変数をプロセスのシンボル・テーブルに表示するには、%SYSTEM.OBJ クラスの ShowReferences(oref) メソッドを使用します。oref は指定されたオブジェクトの OREF (オブジェクト参照) です。OREF の詳細は、"クラスの定義と使用" の "登録オブジェクトを使用した作業" の章にある "OREF の基本" セクションを参照してください。

16.5.2 エラー・トラップ・ユーティリティ

エラー・トラップ・ユーティリティ、%ETN と %ERN は、変数を格納し、エラーに関する他の適切な情報を記録することで、エラーの解析に役立ちます。

16.5.2.1 %ETN アプリケーション・エラー・トラップ

エラー・トラップを設定すると、アプリケーション・エラーの発生時、%ETN ユーティリティを実行できるため便利です。ユーティリティは、エラー時のジョブに関する重要な情報 (実行スタックや各変数の値など) を保存します。この情報はアプリケーションのエラー・ログに保存されており、%ERN ユーティリティを使用して表示するか、または管理ポータルの [**アプリケーションエラーログの表示**] ページ ([**システム処理**]→[**システムログ**]→[**アプリケーションエラーログ**]) を使用して確認できます。

以下のコードを使用して、このユーティリティにエラー・トラップを設定します。

```
SET $ZTRAP="^%ETN"
```

注釈　プロシージャでは、外部ルーチンに $ZTRAP を設定できません。この制限のため、プロシージャでは ^%ETN を使用できません (プロシージャであるクラス・メソッドを含む)。ただし、%ETN を呼び出すローカル・ラベルには $ZTRAP を設定できます。

エラーが発生し、%ETN ユーティリティを呼び出すと、以下のようなメッセージが表示されます。

```
Error has occurred: <SYNTAX> at 10:30 AM
```

%ETN は HALT コマンド (プロセスを終了) で終了するため、アプリケーション・モードでルーチンを使用している場合にのみ、%ETN エラー・トラップを設定できます。ターミナル・プロンプトでは、エラーが発生したときに、ターミナルにエラーが表示され、デバッガ・プロンプトで直ちにエラーの分析を行えるので、便利です。以下のコードは、InterSystems IRIS がアプリケーション・モードの場合にのみ、エラー・トラップを設定します。

```
SET $ZTRAP=$SELECT($ZJ#2:"",1:"^%ETN")
```

16.5.2.2 %ERN アプリケーション・エラー・レポート

%ERN ユーティリティは、%ETN エラー・トラップ・ユーティリティによって記録されたアプリケーション・エラーを検証します。%ERN の使用法の詳細は、このドキュメントの“エラー処理”の章の”%ERN を使用したアプリケーション・エラー・ログの表示”を参照してください。

以下のコードでは、すべての変数を“*LOAD”でロードしてからルーチンをロードすることを示すために、ルーチン REPORT の ZLOAD が発行されています。エラー発生時、DO に関する情報などを記録するプログラム・スタックが空でない場合、ジョブの状態を再現できます。

Terminal

```
USER>DO ^%ERN

For Date: 4/30/2018   3 Errors

Error: ?L

1) "<DIVIDE>zMyTest+2^Sample.MyStuff.1"  at 10:27 am.   $I=|TRM|:|10044   ($X=0  $Y=17)
     $J=10044  $ZA=0   $ZB=$c(13)   $ZS=262144 ($S=268242904)
               WRITE 5/0

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16 pm. $I=|TRM|:|10044   ($X=0  $Y=57)
     $J=10044  $ZA=0   $ZB=$c(13)   $ZS=2147483647 ($S=2199023047592)
 SET ^REPORT(%DAT,TYPE)=I

3) <UNDEFINED>zMyTest+2^Sample.MyStuff.1 *undef"  at 10:13 pm.   $I=|TRM|:|12416   ($X=0  $Y=7)
     $J=12416  $ZA=0   $ZB=$c(13)   $ZS=262144 ($S=268279776)
               WRITE undef

Error: 2

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16 pm. $I=|TRM|:|10044   ($X=0  $Y=57)
     $J=10044  $ZA=0   $ZB=$c(13)   $ZS=2147483647 ($S=2199023047592)
 SET ^REPORT(%DAT,TYPE)=I

Variable: %DAT
  %DAT="Apr 30 2018"

Variable: TYPE
```

```
  TYPE=""

Variable: *LOAD
USER>ZLOAD REPORT

USER>WRITE

%DAT="Apr 30 2018"
%DS=""
%TG="REPORT+1"
I=88
TYPE=""
XY="SET $X=250 WRITE *27,*91,DY+1,*59,DX+1,*72 SET $X=DX,$Y=DY"
USER>
```