

InterSystem IRIS 開発者ガイド

Current Document Version

Version 0.9	2020-03-30	Hiroshi Sato
Version 1.0	2020-10-15	Hiroshi Sato
Version 1.1	2021-03-31	Hiroshi Sato
Version 1.2	2021-07-29	Hiroshi Sato
Version 1.3	2022-08-03	Hiroshi Sato

Document Modifications

	Description of Change	Modified By	Date
0.9	First Beta Version	Hiroshi Sato	2016-02-19
1.0	For 2020.3	Hiroshi Sato	2020.10.15
1.1	Login Password Change	Hiroshi Sato	2021-03-31
1.2	Refer to DC contents for Visual Studio Code	Hiroshi Sato	2021-07-27
1.3	Logo Change etc.	Hiroshi Sato	2022-08-03

内容

InterSystem IRIS 開発者ガイド	1
Document Modifications	2
はじめに	16
第 1 部 基本編	17
第 1 章 InterSystems IRIS の概要	17
1.1 InterSystems IRIS の概要	17
InterSystems IRIS のコンセプト	17
1-2 IRIS のエンジンである多次元データエンジンとは	18
1-3 IRIS を特徴づけるユニークな機能「統一データアーキテクチャ」	19
1-4 IRIS を統合的に操作するユーティリティ「IRIS キューブ」	21
1-5 統合開発環境「スタジオ」	22
1-6 IRIS 全体からリレーショナルアクセスまで視覚的に管理できる「管理ポータル」	25
1-7 まとめ	28
データベース管理システムとしての特徴	28
開発環境としての特徴	28
第 2 章 IRIS のセットアップ	29
2-1 セットアップの要件	29
Windows 10,11 Profetional	29
Windows 10,11 Home	29
MacOS	29
2-5 IRIS Docker イメージのセットアップ	30
ネームスペース	30
スタジオ...IRIS の統合開発環境 (IDE)	36
ウェブターミナル...ブラウザ版端末エミュレータ	37
管理ポータル...IRIS の統合管理環境	38
第 3 章 InterSystems ObjectScript とは	40
3-1 言葉の定義	40
ルーチン	40
クラス	40

グローバル	40
IRIS SQL	42
マクロとインクルードファイル	42
MAC,INT,OBJ,デプロイコード	43
3-2 InterSystems ObjectScript プログラミング例	44
3-3 プログラム要素	47
ラベル (random, input, getnumbername,interesting など)	47
コメント	47
プロシジャ、関数、サブルーチン	48
変数	50
配列	52
演算子	53
パターンマッチング	53
コマンド	55
\$SYSTEM メソッド/変数	59
ロック	59
システム関数	61
リスト	62
その他のシステム関数	64
日付と時間	65
追加のプログラミング	66
3-4 IRIS Object 基本コンセプト	68
オブジェクトとプロパティ	68
メソッド	69
3-5 InterSystems ObjectScript を使った簡単なプログラミング	70
第 4 章 IRIS のオブジェクトモデルとは	76
4-1 IRIS オブジェクトの概要	76
IRIS オブジェクトアーキテクチャ	76
4-2 オブジェクト指向データベース開発	82
クラスとオブジェクト	82

抽象化とモデリング	83
継承とポリモフィズム	83
カプセル化.....	83
拡張	84
オブジェクト永続性.....	84
4-3 IRIS オブジェクトモデル	85
オブジェクト参照 OREF、OID、ID	85
クラスタイプ	86
一時オブジェクトクラス.....	86
永続オブジェクトクラス.....	87
シリアルオブジェクトクラス.....	87
データタイプクラス.....	87
継承	88
多重継承	89
クラスのコンパイル.....	89
4-4 IRIS クラス	90
命名基準	91
クラスパラメータ	93
4-5 パッケージ	94
パッケージ概要.....	94
パッケージ名	95
パッケージを定義する	95
パッケージの使用	95
パッケージと SQL	96
組み込みパッケージ.....	96
4-6 メソッド	97
メソッド引数	97
既定値の指定方法	97
参照渡しによる呼び出し.....	97
可変引数	98

メソッドの可視性	99
メソッドの継承	99
メソッド言語	100
メソッドキーワード	100
インスタンスメソッドとクラスメソッド	101
メソッドの種類	102
4-7 プロパティ	103
プロパティキーワード	104
プロパティ可視性	104
プロパティの振る舞い	104
プロパティアクセサ	105
プロパティの属性	105
データタイププロパティ	105
オブジェクト値プロパティ	106
コレクションプロパティ	106
ストリームプロパティ	106
4-8 クラスクエリ	107
クエリの基本	107
クエリの構造	107
クエリキーワード	107
クラスクエリ定義の作成	108
4-9 インデックス	108
インデックスキーワード	109
インデックス照合	109
4-10 ObjectScript でのオブジェクトの使用	110
メソッドの実行	110
戻り値について	110
インスタンスメソッドの実行	110
クラスメソッドの実行	110
エラー条件	111

オブジェクトの新規作成.....	111
オブジェクトのオープン.....	111
オブジェクトの変更.....	112
参照プロパティの変更.....	112
埋め込みオブジェクトプロパティの変更.....	112
List プロパティの変更.....	113
Array プロパティの変更.....	113
ストリームプロパティの変更.....	113
オブジェクトの保存.....	114
オブジェクトの削除.....	114
1 つのオブジェクトの削除.....	114
エクステンント中の全オブジェクトの削除.....	114
クエリの実行.....	114
クエリメタデータメソッド.....	115
クエリの実行の準備をする.....	115
クエリ実行.....	115
クエリ結果の処理.....	116
クエリのクローズ.....	116
4-11 データタイプ.....	117
利用可能なタイプ.....	117
演算操作.....	118
パラメータ.....	118
キーワード.....	118
データ形式とその変換メソッド.....	118
列挙型プロパティ.....	119
4-12 オブジェクト永続性.....	120
永続インタフェース.....	120
オブジェクトの保存.....	120
オブジェクトのオープン.....	121
オブジェクトの削除.....	121

オブジェクトの存在チェック	121
オブジェクトエクステンツ	121
エクステンツクエリ	122
ストレージ定義とストレージクラス	122
スキーマ進化	123
ストレージ定義の再設定	123
4-13 リレーションシップ	124
リレーションシップの基礎	124
リレーションシップキーワード	124
リレーションシップの定義	125
依存リレーションシップ	125
リレーションシップのメモリ上の振る舞い	125
リレーションシップの永続データの振る舞い	126
参照整合性	126
依存リレーションシップの永続データの振る舞い	126
4-14 オブジェクト用 ObjectScript	127
.. シンタックス	127
##class シンタックス	127
クラスメソッドの起動	128
メソッドのキャスト	128
\$this シンタックス	128
##super シンタックス	128
i%<PropertyName> シンタックス	129
.. #<Parameter> シンタックス	129
4-15 XData ブロック	129
4-16 クラス定義クラス	130
クラス定義の閲覧	130
クラス定義の変更	130
4-17 オブジェクト同時実行制御	131
同時実行制御オプション	131

バージョンチェック	132
4-18 IRIS オブジェクトを使ったサンプルプログラミング	133
サンプルプログラム「sales.xml」	133
第 5 章 IRIS SQL	141
5-1 テーブルに対するクラスの投影	141
テーブル名	143
継承	144
定数プロパティ	144
データ型	145
計算プロパティ	145
参照プロパティ	145
埋め込みオブジェクトプロパティ	146
リレーションシップ	146
リストプロパティ	146
配列プロパティ	147
インデックス	147
クエリ	149
クラスメソッド	150
5-2 標準 SQL (ANSI 92) サポートの例外	151
DQL 文	151
DML 文	152
DDL 文	153
5-3 IRIS による SQL 拡張	154
標準に追加された IRIS SQL 演算子	154
5-4 埋め込み SQL と動的 SQL	159
非カーソルベース SQL	159
カーソルベース SQL	160
動的 SQL	162
5-5 IRIS SQL サーバ	163
ODBC を使ったアクセス	163

5-6 IRIS SQL のその他の機能	167
全文検索機能	167
パフォーマンス最適化	168
SQL 外部キーの定義	169
トリガの定義	173
ストアドプロシージャの定義	174
5-7 IRIS SQL サンプルプログラム	178
第 6 章 Caché Server Pages による Web アプリケーション構築	181
6-1 CSP の機能	181
6-2 CSP の学習を始める前に	183
実運用 Web サーバと IRIS 提供プライベート Web サーバの準備	183
Web サーバと CSP ゲートウェイの構成	184
CSP で知っておくべきこと	184
CSP サンプルについて	184
CSP ドキュメントについて	185
6-3 最初の CSP ページを作成する	186
クラスによる CSP ページの作成手順	186
HTML タグを使った CSP ページの作成	193
6-4 CSP のアーキテクチャ	197
各コンポーネントの役割	197
CSP による情報の流れ	198
静的ファイル	198
6-5 CSP の HTTP リクエスト	199
CSP 実行環境	199
HTTP リクエスト処理	200
Web サーバと CSP ゲートウェイ	201
CSP サーバ	202
CSP サーバのイベントフロー	202
CSP サーバ URL とクラス名解決	203
%CSP.Page クラス	204

Page メソッド	204
CSP エラーの操作	206
%CSP.Request オブジェクト	208
%CSP.Response オブジェクトと OnPreHTTP メソッド	211
6-6 CSP セッション管理	213
CSP.Session によるセッション	213
%CSP.Session オブジェクト	215
セッションタイムアウト	217
状態管理	217
リクエスト間のデータを追跡する	218
ページ内にデータを格納する	218
クッキーにデータを格納する	220
セッションにデータを格納する Data プロパティ	220
データベースにデータを格納する	221
サーバコンテキスト保存 Preserve プロパティ	221
認証と暗号化	221
セッションキー	222
暗号化された URL と CSPToken	222
プライベートページ	224
暗号化した URL パラメータ	225
6-7 CSP によるタグを使った開発	226
自動と手動ページコンパイル	228
CSP マークアップ言語	228
CSP ページ言語	229
実行時表現	230
サーバ側メソッド	234
SQL <script>タグ	236
生成クラスを制御する	238
HTTP 出力のエスケープ	241
サーバ側メソッド	245

6-8 データベースアプリケーションの構築.....	258
ページ上のオブジェクトを使う	258
テーブルのオブジェクトデータを表示する	259
フォーム内にオブジェクトデータを表示する	260
フォームサブミット要求を処理する	262
プロパティに結び付ける.....	266
<csp:serach>タグによる CSP 検索ページ.....	267
6-9 CSP を使ったサンプルアプリケーション	269
ページ構成.....	269
クラス構成.....	269
セットアップ	270
クラスロード	エラー! ブックマークが定義されていません。
データロード	エラー! ブックマークが定義されていません。
イメージなどのセットアップ.....	エラー! ブックマークが定義されていません。
動作確認	270
7 章 REST	275
7-1 %CSP.REST クラス	276
7-2 REST 用の URL マップを作る	277
7-3 データ形式を指定する.....	279
8 章 JSON	280
8.1 概要	280
8.2 動的オブジェクトと配列	280
8.3 動的オブジェクトの使用	282
8.3.1 動的オブジェクトの作成.....	282
8.3.2 動的オブジェクトの参照、設定	282
8.3.3 動的オブジェクトにプロパティがあるかチェックする	283
8.3.4 動的オブジェクトからプロパティを削除する	283
8.3.5 プロパティの値を得る	284
8.3.6 プロパティの数を得る	284
8.3.7 プロパティを繰り返し処理する	285

8.4 動的配列の使用	286
8.4.1 動的配列の作成	286
8.4.2 動的配列に項目を追加する	287
8.4.3 配列の項目を修正する	287
8.4.4 動的配列に該当プロパティがあるかどうかチェックする	288
8.4.5 動的配列から項目を削除する	288
8.4.6 配列項目の値を得る	289
8.4.7 項目数を得る	289
8.4.8 配列項目を繰り返す	290
8.5 動的オブジェクトまたは動的配列を JSON にシリアルライズする	291
8.5.1 基本テクニック	291
10 章 他プログラミング言語との連携	292
10.1 一般的に推奨するインタフェース	293
10.1.1 REST/JSON	293
10.1.2 Web サービス (SOAP)	293
10.2 Java 言語との連携	294
10.2.1 JDBC	294
10.2.2 Java eXTreme	294
10.2.3 Hibernate	294
10.3 .NET との連携	295
10.3.1 ADO.NET	295
10.3.2 .NET eXTreme	295
10.3.3 ADO.NET Entity Framework	295
10.4 使用推奨しないインタフェース	296
11 章 XML のサポート	297
11-1 クラスのオブジェクトを XML ドキュメントにエクスポートする	297
XML 対応クラスの例	298
XML ドキュメントの例	299
XML ライターの作成の概要	300
11-2 XML ドキュメントを IRIS にインポートする	302

11-3	DOM の生成	305
	DOM として XML ドキュメントを開く	305
	ファイルの DOM への変換	306
	オブジェクトの DOM への変換	307
	DOM ノードのナビゲート	308
11-4	%XML.TextReader クラスの使用	310
	テキストリーダーメソッドの作成	310
11-5	XPath の使用	317
	IRIS における XPath 式の評価の概要	317
	XPath の結果の使用法	319
	サブツリー結果を持つ XPath 式の評価	320
	スカラ結果を持つ XPath 式の評価	321
11-6	XSLT の使用	323
	IRIS における XSLT 変換の実行の概要	323
11-7	XML スキーマからのクラスの生成	328
	ウィザードの使用法	329
	クラスからの XML スキーマの生成	330
	簡単な例	332
	より複雑なスキーマの例	333
12 章	Web サービス	338
12-1	IRIS Web サービスの概要	338
	IRIS Web サービスの作成	338
	Web アプリケーションの一部としての Web サービス	338
	WSDL	339
12-2	IRIS Web クライアントの概要	340
	IRIS Web クライアントの作成	340
12-3	IRIS Web サービスの作成	341
	基本要件	342
	簡単な例	345
	Web サービスの生成	347

12-4 Web クライアントの作成	353
SOAP ウィザードの概要	353
SOAP ウィザードの使用法	353
生成された Web クライアント・クラスの使用法	354
13 章 その他	356
13-1 ファイル IO	356
13-2 インターネットライブラリー	358
13-3 %Populate クラス	359
13-4 UNITTEST	360

はじめに

Caché の後継製品として InterSystems IRIS がリリースされてから早くも 4 年目を迎えました。

従来 Caché 向けに提供していたものを IRIS 用に手直しして、新たに提供することになりました。

IRIS は猛烈な勢いで進化し続けており、Caché には存在しなかった機能もたくさんありますし、今後もどんどん増えていくことでしょう。

新機能については、おいおい追加していくことを考えていますが、まずはベースラインの機能についてある程度網羅的にカバーしていきたいと思います。

第 1 部 基本編

第 1 章 InterSystems IRIS の概要

1.1 InterSystems IRIS の概要

IRIS のコンセプトと 2 つの特徴である「多次元データエンジン」と「マルチモデル」を説明します。

InterSystems IRIS のコンセプト

IRIS は、様々なデータタイプ（キーバリュー、リレーショナル、オブジェクト、ドキュメント、グラフ、マルチバリュー等）を柔軟に 1 つの物理データとして取り扱うことのできるマルチモデル型のデータプラットフォームです。

処理に応じて別々の物理データを管理する必要がありませんので、**One fact in one place** という本来あるべきデータ管理が実現できます。

インターシステムズは、最新の IRIS の評価キットを様々な形で提供しています。

このドキュメントでは、その中で **Docker** を使用した評価キットの使用を前提に話を進めていきます。

この評価キットに付随するライセンスでは、製品版と同じ機能（一部機能制限があります）を利用できる使用許諾が与えられ、使用期限もありません。（ライセンスという観点では、使用期限はありませんが、提供している **Docker** イメージには使用期限があります。新しいバージョンの **Docker** イメージを使用することで、期限を延長できます。）

但し、商用アプリケーションの開発には使用できません。

またシングルユーザーライセンスであるため、スタンドアロンでのみ使用可能です。

1-2 IRIS のエンジンである多次元データエンジンとは

オブジェクトデータベースは、プログラムで使用するオブジェクトをシリアルライズ（直列化）して外部媒体（通常は、ハードディスク）に保持する「永続性」という概念から始まりました。

IRIS の多次元データという概念は、オブジェクトデータベースが世の中に出現する前に考案されたものですが、同じ様な考え方を踏襲しています。

一般的には変数はメモリー上に存在し、プログラムが終了すると消失しますが、変数に永続性という属性を付与することにより、プログラム終了後にも変数構造を保持し、後からそのデータの参照、更新、あるいは削除を行えるようにしました。

その際にその変数構造も単純なスカラー変数だけではなく、複雑な構造を保持できる多次元配列変数についても永続化できるようにしました。

IRIS の多次元データエンジンとは、多次元データ構造にアクセスするためのデータ管理機構の総称です。

そのアクセスの実装は、IRIS のプログラミング言語である InterSystems ObjectScript の言語仕様として定義されています。

1-3 IRIS を特徴づけるユニークな機能「統一データアーキテクチャ」

IRIS の統一データアーキテクチャをひとことで言うと、格納されたデータに対して様々な形でアクセスできる機能になります。

IRIS では、オブジェクトやリレーショナルの定義をクラスとして体系付け、そのクラスにプロパティとメソッドを定義します。

プロパティは、リレーショナルデータベースのカラム、フィールドに相当するものです。

メソッドは一般的なリレーショナルデータベースには存在しない概念ですが、オブジェクト指向でのデータの振舞いを定義するものになります。

さらに IRIS では一般的なリレーショナルデータベースと同様にインデックスやビュー、トリガーなどの定義もできます。

そして実体としてのデータは多次元配列変数に格納されますが、その多次元配列変数とクラス定義情報を関連づけるためのストレージ定義も自動的に生成されます。

リレーショナルアクセスによるデータビューとオブジェクトアクセスによるデータビューは全く同じストレージ構造を参照します。

オブジェクトアクセスとリレーショナルアクセスによるデータの作成、更新および削除時には自動的にインデックス情報も適切に維持管理します。

図 1 – 1 は、統一データアーキテクチャに基づいて、どのように実行コードが生成されるかを図にしたものです。

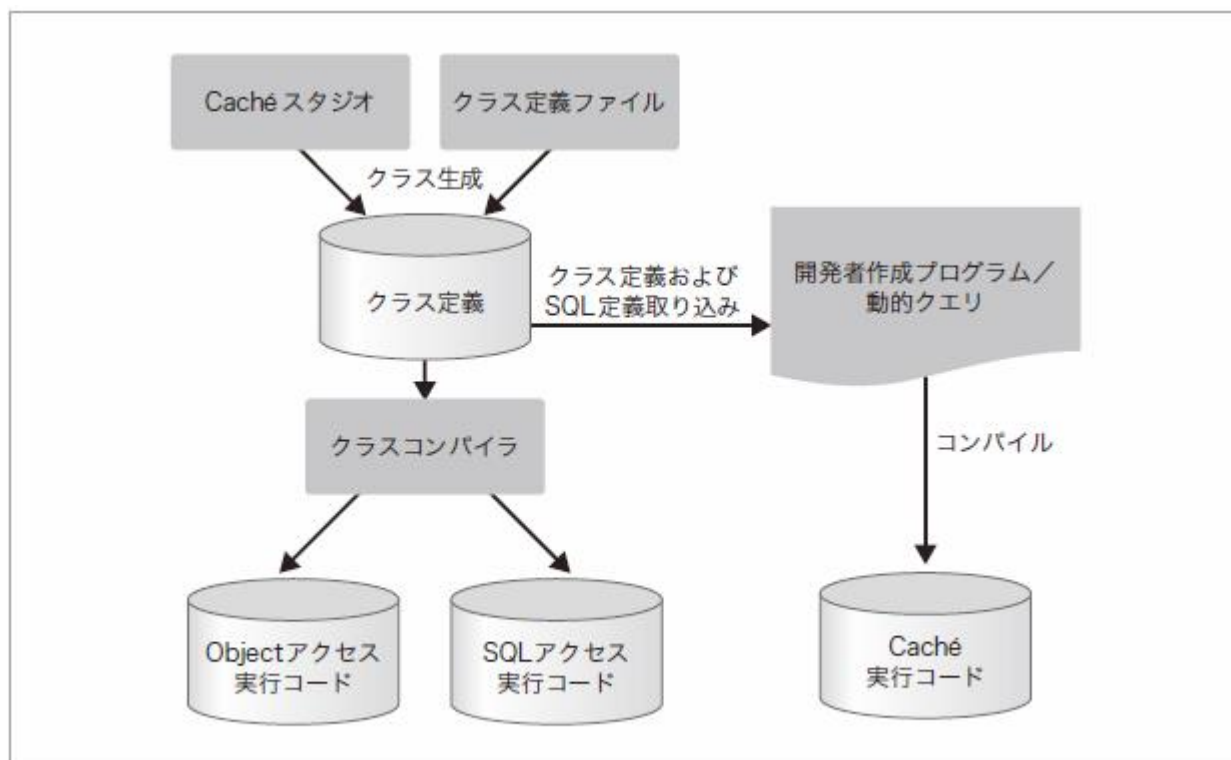


図 1-1 : Caché統一データ実行アーキテクチャ

後述する開発環境である「スタジオ」または Visual Studio Code でクラスを定義しコンパイルを実行するとそのクラスにオブジェクトアクセスするための実行コードとリレーショナルアクセスするための実行コードを自動生成します。

開発者が作成するプログラムの中でリレーショナルアクセスを行う場合にも、そのクラス定義を参照してそのストレージ構造にアクセスするコードを自動生成します。

JDBC や ODBC 経由でアクセスする際にも対応するストレージ構造にアクセスするコードが動的に自動生成されるため、クライアントのソフトウェアは行いたい処理を SQL 文で記述することに集中することができます。

1-4 IRIS を統合的に操作するユーティリティ「IRIS キューブ」

IRIS をウィンドウズ上でスタンドアロンでインストールすると、トレイアイコンとして IRIS キューブを作成します。

また IRIS をサーバーにインストールし、ウィンドウズ PC に IRIS クライアントオプションをインストールすると、そのクライアントに IRIS キューブを作成します。

そのアイコンをクリック（右、左どちらでも良い）するとメニューが表示され、IRIS の機能にアクセスすることができます。（図 1 – 2）



図 1-2 IRIS Cube をクリックして表示されるメニュー

このメニューに表示されている「スタジオ」は、クラスやメソッドを定義する開発環境です。

また、「ターミナル」は作成したメソッドなどの実行環境です。この 2 つを使い、データベースを構築することができます。

また、IRIS 全体の管理やリレーショナルアクセスをより視覚的に行うものが「管理ポータル」です。

その下の「ドキュメント」は IRIS のマニュアルですが、チュートリアルも含めて体系的にまとめられています。

また、全文検索機能を持っているので、任意のキーワードを指定してトピックを検索することもできます。

1-5 統合開発環境「スタジオ」

IRIS は、単なるデータベース管理システムではなくプログラミング言語を含んだ統合開発環境となっています。

一般的な開発環境では、プログラミング言語とデータベースシステムは完全に切り離されています。

例えば、Java 言語の場合には、一般的には DBMS とは JDBC を使用して連携します。一方、IRIS には、ObjectScript というプログラミング言語が含まれており、これらのプログラミング言語の仕様としてデータベースアクセス機能が完全に統合されています。

図 1-3 は「スタジオ」の初期画面ですが、この統合開発環境で、クラス定義やプログラムの開発からデバッグまで行うことができます。(スタジオはウィンドウズでのみ動作します。)

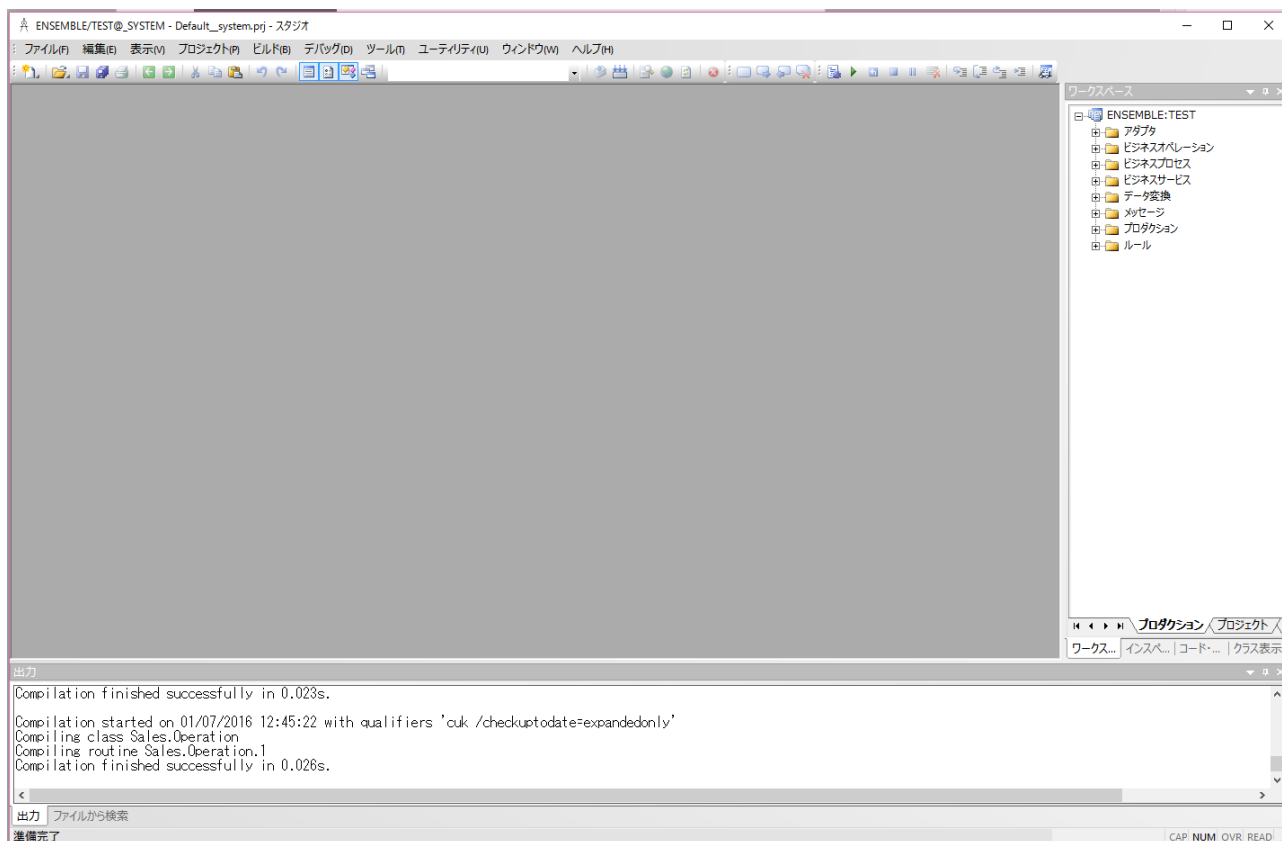

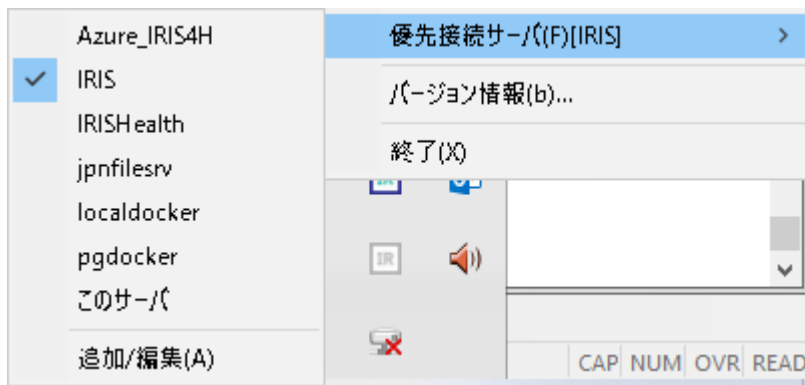


図 1-3 IRIS スタジオ初期画面

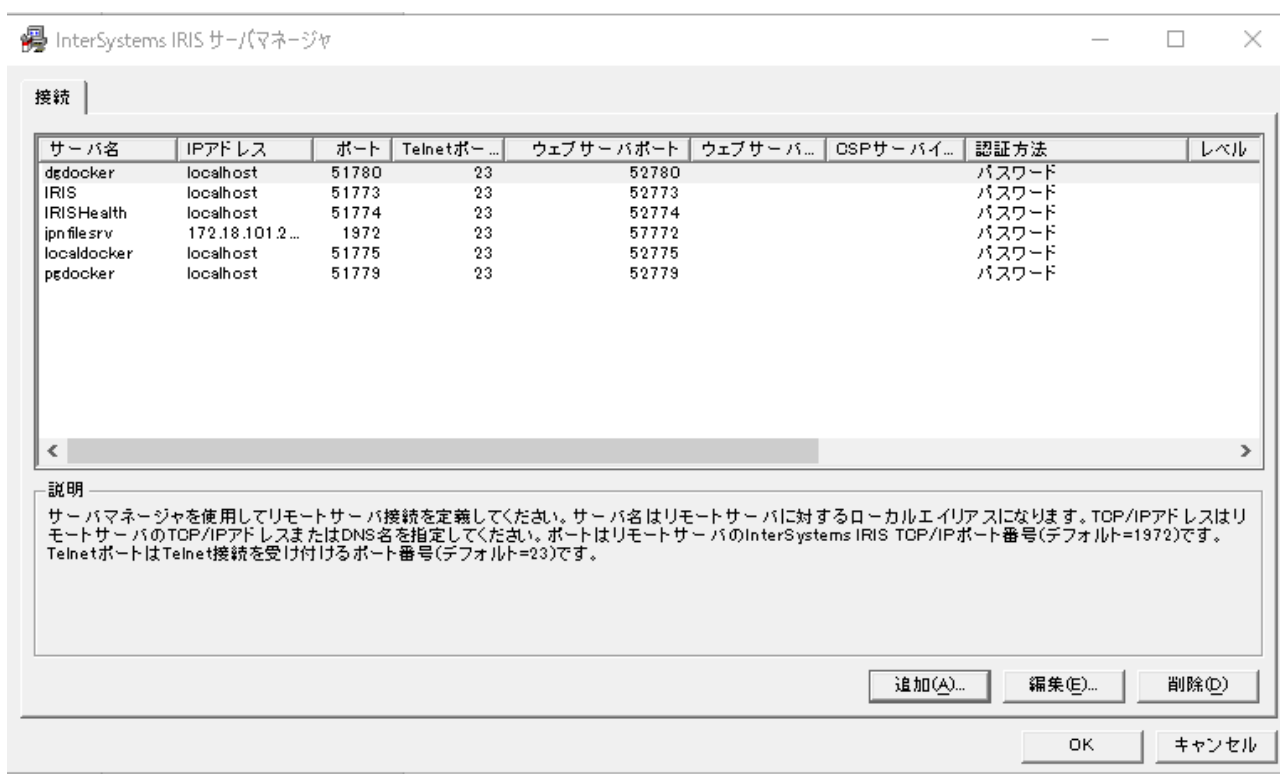
機能の多くはウィザード化されているので、毎回同じ様な入力に煩わされることもないと思います。

2 章で説明しているセットアップ環境にアクセスするためには、以下の設定が必要です。

ウィンドウズのタスクバーから  のアイコンをクリックして、優先接続サーバーをクリックして、次に追加/編集メニューをクリックします。



InterSystems IRIS サーバマネージャが起動するので、追加ボタンをクリックします。



接続追加情報に以下の情報を入力します。

接続を追加

サーバ名: dgdocker

IPアドレス: localhost

ポート: 51780

Telnetポート: 23

ウェブサーバポート: 52780

ウェブサーバIPアドレス:

GSPサーバインスタンス:

コメント:

認証方法

☒ パスワード

☐ Kerberos

OK キャンセル

入力した情報が追加されたか確認します。

InterSystems IRIS サーバマネージャ

接続

サーバ名	IPアドレス	ポート	Telnetポー...	ウェブサーバポート	ウェブサーバ...	GSPサーバ...	認証方法	レベル
dgdocker	localhost	51780	23	52780			パスワード	
IRIS	localhost	51773	23	52773			パスワード	
IRISHealth	localhost	51774	23	52774			パスワード	
jpn.filesrv	172.18.101.2...	1972	23	52772			パスワード	
localdocker	localhost	51775	23	52775			パスワード	
pgdocker	localhost	51779	23	52779			パスワード	

説明

サーバマネージャを使用してリモートサーバ接続を定義してください。サーバ名はリモートサーバに対するローカルエイリアスになります。TCP/IPアドレスはリモートサーバのTCP/IPアドレスまたはDNS名を指定してください。ポートはリモートサーバのInterSystems IRIS TCP/IPポート番号(デフォルト=1972)です。TelnetポートはTelnet接続を受け付けるポート番号(デフォルト=23)です。

追加(A)... 編集(E)... 削除(D)

OK キャンセル

1-6 IRIS 全体からリレーショナルアクセスまで視覚的に管理できる「管理ポータル」

「管理ポータル」は環境に依存することなく、ブラウザを使って管理できる IRIS ポータルサイトです。（図 1 - 4）

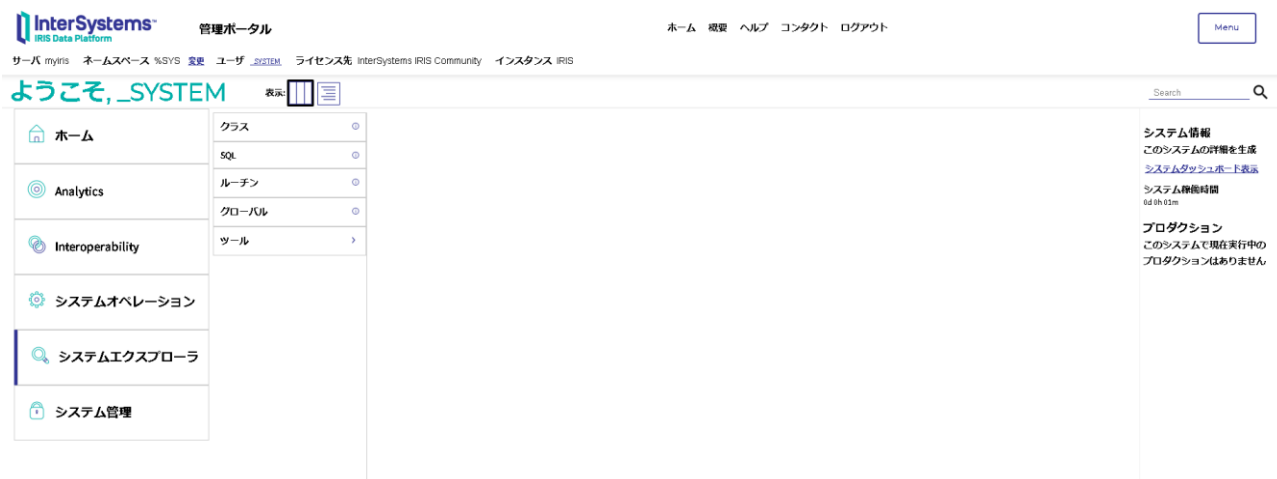


図 1-4 IRIS 管理ポータル初期画面

システムエクスプローラ>SQL を選ぶと、図 1－5 にあるような SQL メニューが表示されます。

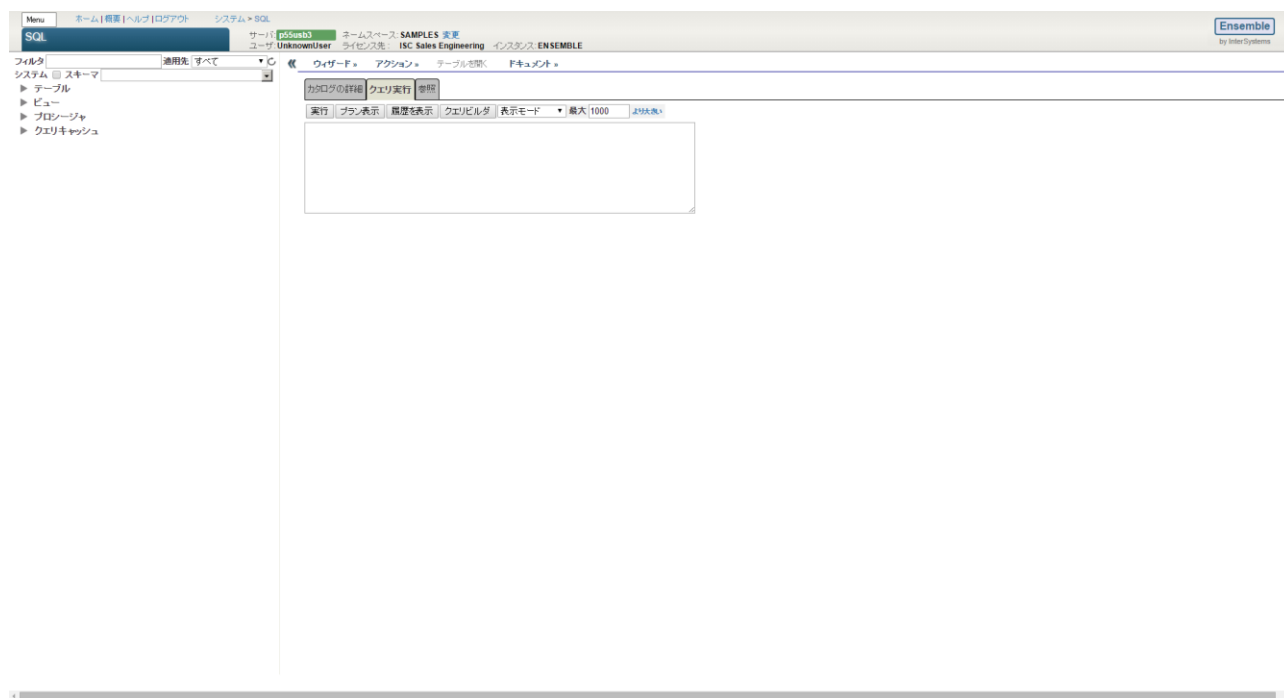


図 1-5 IRIS 管理ポータル SQL メニュー

SQL 文の実行や新規ビューの作成、テキストファイルからのデータインポートやエクスポートなど、日常的な運用に必要な SQL 関連機能が利用可能です。

1 つ例を挙げましょう。 図 1-6 は User ネームスペースのテーブル Sales.Customer を選んで、カタログの詳細タブを表示させたものです。

InterSystems™ IRIS Data Platform 管理ポータル

ホーム 概要

サーバ myiris ネームスペース USER 変更 ユーザ _SYSTEM ライセンス先 InterSystems IRIS Community インスタンス IRIS

システム > SQL

フィルタ Sales.* 適用先 すべて

システム スキーマ Sales

テーブル

- > Sales.Customer
- > Sales.OrderItem
- > Sales.Product
- > Sales.PurchaseOrder
- > ビュー
- > プロシージャ
- > クエリキャッシュ

ウィザード » アクション » テーブルを開く ツール » ドキュメント »

カタログの詳細 クエリ実行 参照 SQLステートメント

すべてのスキーマを表示 フィルタされたスキーマを表示

(4) スキーマのテーブル: Sales

名前	所有者	最終コンパイル	外部	読み専用	クラス名	EXTENTSIZE
Customer	_SYSTEM	2020-03-26 05:13:34	0	0	Sales.Customer	10
OrderItem	_SYSTEM	2020-03-26 05:13:34	0	0	Sales.OrderItem	100000
Product	_SYSTEM	2020-03-26 05:13:34	0	0	Sales.Product	10
PurchaseOrder	_SYSTEM	2020-03-26 05:13:34	0	0	Sales.PurchaseOrder	100000

図 1-6 IRIS 管理ポータル SQL テーブルのフィールド確認画面

1-7 まとめ

IRIS のデータベース管理システムと開発環境としての特徴を挙げます。

データベース管理システムとしての特徴

- ・リレーショナル、オブジェクト、キーバリュー、ドキュメント、マルチバリューなど様々なデータモデルをサポートしたマルチモデル対応
- ・スキーマ進化に柔軟に対応
- ・1 ユーザーから数万ユーザーのサポートまでプログラムを変更することなく拡張可能なスケーラビリティ
- ・データの経年劣化がなくデータの再編成およびそれに付随するインデックスの再構築作業等の必要がないため長期的な運用コストを削減できる
- ・システムリソースの消費が少なくハードウェアコストを低減できる

開発環境としての特徴

- ・オブジェクト指向の完全なサポートとそれを支援する強力なクラスライブラリを有す
- ・様々なデータモデルのサポート
- ・複数のスキーマモデルをサポート（スキーマレス、完全スキーマ（クラス定義）、セミスキーマ（XML、ドキュメントモデル））
- ・Tunable Consistency(プログラミングレベルで制御可能なデータ整合性の選択肢を提供)
- ・人気のある開発環境（開発言語、開発フレームワーク、IDE 等）との様々な連携機能
- ・豊富なインターオペラビリティ機能やコード自動生成を含む組み込みフレームワークによる迅速なアプリケーション開発の実現
- ・Web アプリケーション開発を支援する強力な開発フレームワークとアプリケーションサーバー機能の提供（CSP）

第 2 章 IRIS のセットアップ

本章では、IRIS のセットアップとユーティリティの基本操作を説明し、InterSystems ObjectScript によるプログラミング例を示します。

IRIS の基本となる考え方を学習しましょう。

2-1 セットアップの要件

このドキュメントの内容で IRIS をセットアップするために必要なソフトウェアの要件について説明します。

Windows 10,11 Professional

事前に Docker for Windows のインストールが必要です。

Windows 10,11 Home

wsl2 を設定した後、Docker for Windows のインストールが必要です。(ビルド 18362 以上)

MacOS

事前に Docker for MAC のインストールが必要です。

2-5 IRIS Docker イメージのセットアップ

このドキュメントに記載している内容を確認したり、サンプルを実行するためには、以下の場所から **Docker** イメージをダウンロードする必要があります。

<https://github.com/wolfman0719/DG>

ダウンロードした ZIP ファイルを適当な場所で解凍します。

Git の clone コマンドでもダウンロードできます。

```
Git clone https://github.com/wolfman0719/DG
```

Docker for Windows の場合には、コマンドプロンプトまたは Windows PowerShell を起動し、解凍したファイルがあるディレクトリに移動します。

Docker Toolbox の場合には、Docker Quick Start Terminal を起動し、解凍したファイルがあるディレクトリに移動します。

Mac の場合には、ターミナルを起動し、同様にディレクトリを移動します。

以下のコマンドを実行します。

```
docker-compose up -d --build
```

ネームスペース

開発者またはアプリケーションのユーザーが IRIS にログインして作業を行う場所を「ネームスペース」と呼んでいます。

ネームスペースは、1 つ以上のデータベースより構成されます。

データベースの実体は、オペレーティングシステムのファイルシステム上に構成される IRIS.DAT という名前のファイルです。

このデータベースの中に、IRIS が使用するプログラム、データ、クラス定義などが格納されます。

Docker イメージ上で既にいくつかのネームスペースとそれに関連付けられるデータベースがインストールされています。

ネームスペースとは、ユーザからみた「作業領域」です。IRIS のデータにアクセスするときや、クラス定義などを作成するとき、または独自のプログラムや CSP アプリケーションを動かすときには、まずネームスペースを指定するところから作業を始めます。

IRIS のデータベースとは、データのみならず、データにアクセスするルーチンやクラス定義などが格納されている場所のことです。

この場所を利用するためには、ユーザはネームスペースを指定しなければなりません。

USER ネームスペースは、ユーザが自由に使うことができるネームスペースです。

評価などの場合は、このネームスペースを使うのが、一番手間がかからず便利です。

しかし、実際にアプリケーションを構築して運用する際には、そのアプリケーション専用のネームスペースを作成する必要があります。

ネームスペースの構成は、管理ポータルで行います。図 2-17 のシステム管理メニューの [構成] をクリックして、図 2-18 にあるようなシステム構成メニューを表示させます。

ようこそ, _SYSTEM

表示:  

 ホーム	構成 >	システム構成 >	メモリと開始設定 ○
	セキュリティ >	接続性 >	ネームスペース ○
 Analytics	ライセンス >	ミラー設定 >	ローカルデータベース ○
	暗号化 >	データベースバックアップ >	リモートデータベース ○
 Interoperability		ウェブゲートウェイ管理 ○	シャード構成 ○
		SQLとオブジェクトの設定 >	ジャーナル設定 ○
 システムオペレーション		デバイス設定 >	
		国際言語設定 >	
 システムエクスプローラ		レポートサーバー >	
		追加の設定 >	
 システム管理			

図 2-18 システム構成メニュー

次に、システム構成メニューの「ネームスペース」をクリックして、図 2-19 にあるような、ネームスペースの指定画面を表示させます。

The screenshot shows the 'Namespaces' page in the InterSystems Management Portal. At the top, there's a navigation bar with 'サーバ myiris', 'ネームスペース %SYS', 'ユーザ _SYSTEM', 'ライセンス先 InterSystems IRIS Community', and 'インスタンス IRIS'. Below this is a breadcrumb 'システム > 構成 > ネームスペース'. The main heading is 'ネームスペース' with a button '新規ネームスペース作成' and a refresh icon with text '最終更新: 2020-03-26 14:27:43.856'. A section title reads '現在のネームスペースおよびそれらのグローバル/ルーチンに対するデフォルトデータベース:'. Below this is a table with columns: Namespace, Globals, Routines, Temp Storage, and a set of links for each namespace. The table lists three namespaces: %ALL, %SYS, and USER.

Namespace	Globals	Routines	Temp Storage	
%ALL	IRISTEMP	IRISTEMP	IRISTEMP	グローバルマッピング ルーチンマッピング パッケージマッピング 削除
%SYS	IRISSYS	IRISSYS	IRISTEMP	グローバルマッピング ルーチンマッピング パッケージマッピング -
USER	USER	USER	IRISTEMP	グローバルマッピング ルーチンマッピング パッケージマッピング 削除

図 2-19 ネームスペース指定画面

この画面で、「新規ネームスペース作成」をクリックすると、図 2-20 のような「新規ネームスペース作成画面」になります。

新規ネームスペース

保存

キャンセル

下記のフォームを使用して新規ネームスペースを作成してください。:

ネームスペース名	<input type="text"/>
	必須です。
コピー元	<input type="text"/>
このネームスペースでグローバルのデフォルト・データベースは	<input checked="" type="radio"/> ローカル・データベース <input type="radio"/> リモート・データベース
グローバルのための既存のデータベースを選択	<input type="text"/> <input type="button" value="新規データベース作成..."/>
	必須です。
このネームスペースでルーチンのデフォルト・データベースは	<input checked="" type="radio"/> ローカル・データベース <input type="radio"/> リモート・データベース
ルーチンのための既存のデータベースを選択	<input type="text"/> <input type="button" value="新規データベース作成..."/>
このネームスペースにデフォルトのウェブアプリケーションを作成	<input checked="" type="checkbox"/>
次からネームスペースマッピングをコピー	<input type="text"/>
相互運用プロダクション用にネームスペースを有効化	<input checked="" type="checkbox"/>

図 2-20 新規ネームスペース作成画面

この画面で、作成する「ネームスペース名」と「デフォルトデータベース」を指定します。

なお、ネームスペースは、指定した デフォルトデータベース以外のデータベースのデータやルーチンへのアクセスも定義可能です。

そのため、複数のデータベースに存在するデータを、あたかも 1 つのデータベースのようなイメージでアクセスできるのです。

例えば、テスト環境用データベース「CUSTTST」と本番環境用データベース「CUSTMGT」があり、アプリケーションで使用する固定データがデータベース「CUSTMST」に格納されているとします。その際の指定方法は次のようになります。

- ・ デフォルトデータベース : グローバル : 「CUSTTST」または「CUSTMGT」
- ・ グローバルマッピング : 「CUSTMST」の ^MasterData

実際のアプリケーション作成の際には、テスト環境であっても、本番環境であっても、ユーザはアクセスするデータベースの位置を意識しなくても、ネームスペースの名前だけ知っていればデータベースへ接続できます。

また、接続先データベースの変更も、この画面で指定するだけでよく、アプリケーションロジックの変更は必要ありません。

スタジオ…IRIS の統合開発環境（IDE）

スタジオは、IRIS 用のスクリプト記述言語である InterSystems ObjectScript の IDE です（図 2-22）。スタジオを使用するためには、別途 Windows 用の IRIS 評価キットをダウンロードする必要があります。

<https://download.intersystems.com/download/login.csp>

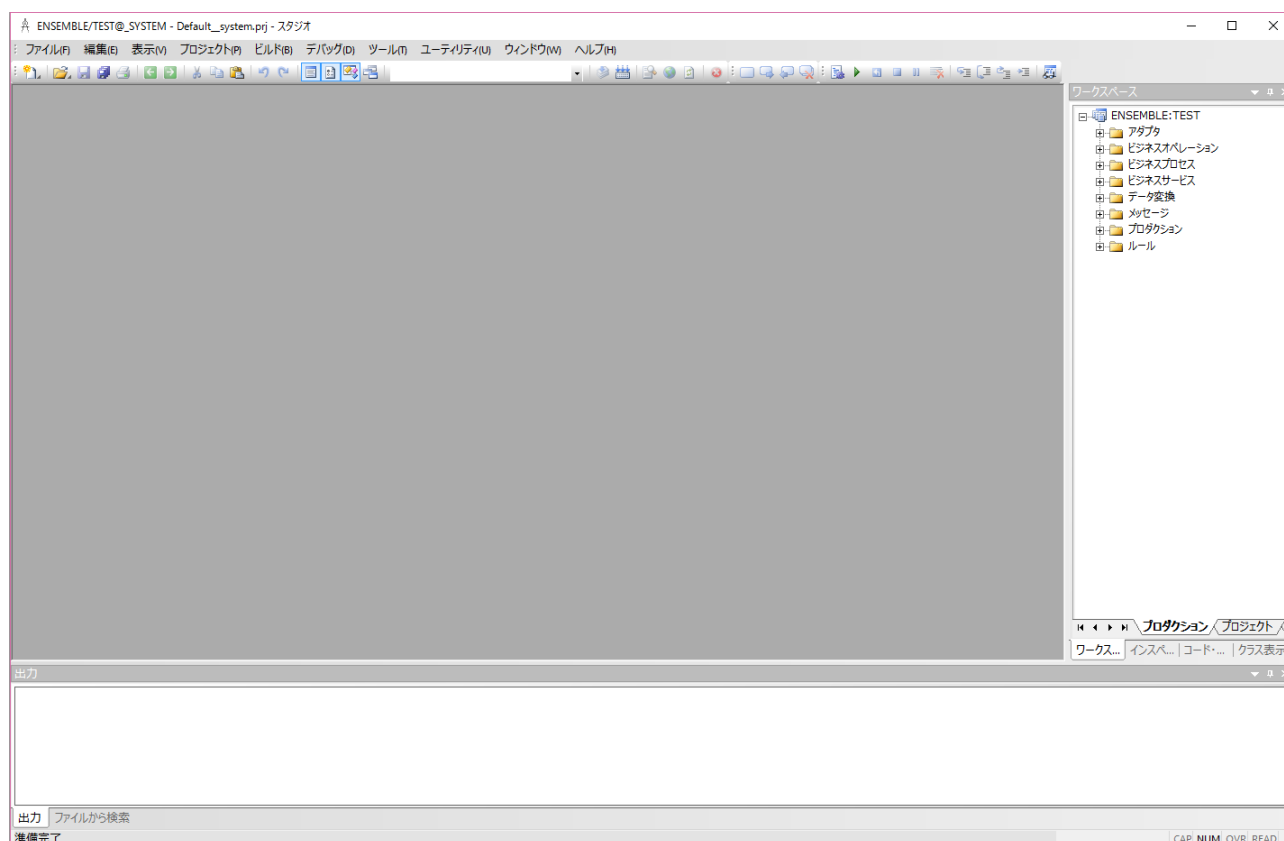


図 2-22 IRIS スタジオ

ここでは、クラス定義の作成、CSP（Caché Server Page）ファイルの作成、Web Service 用クラス定義の作成や SOAP クライアントウィザードや XML スキーマウィザードが利用できます。

また、このスタジオでは、プロジェクト単位でアプリケーションに必要なソースをまとめて保存できます。

スタジオとは別に Microsoft の Visual Studio Code も IDE として利用することもできます。

使用方法の詳細は以下のサイトをご確認ください。

[VSCoDe を使ってみよう](#)

ウェブターミナル…ブラウザ版端末エミュレータ

このターミナルで、InterSystems ObjectScript のコマンド、システムユーティリティなどを実行できます（図 2-23）。

お使いのブラウザから以下の url を指定してください。

<http://localhost:52780/terminal/>

ユーザー名: _system

パスワード: SYS

```
CWTV4.9.3 myiris:IRIS: SYSTEM  
Welcome to WebTerminal! Type /help special command to see how to use all the features.  
USER > _
```

最後の *h* は必須です。

アプリケーションを作成する際、作成したルーチン、クラスのメソッドの動作確認にも利用可能です。

管理ポータル…IRIS の統合管理環境

以下からアクセスできます。

http://localhost:52780/csp/sys/UtilHome.csp?IRISUsername=_system&IRISPassword=SYS

管理ポータル（図 2-24）では、次の 3 つの機能が提供されています。

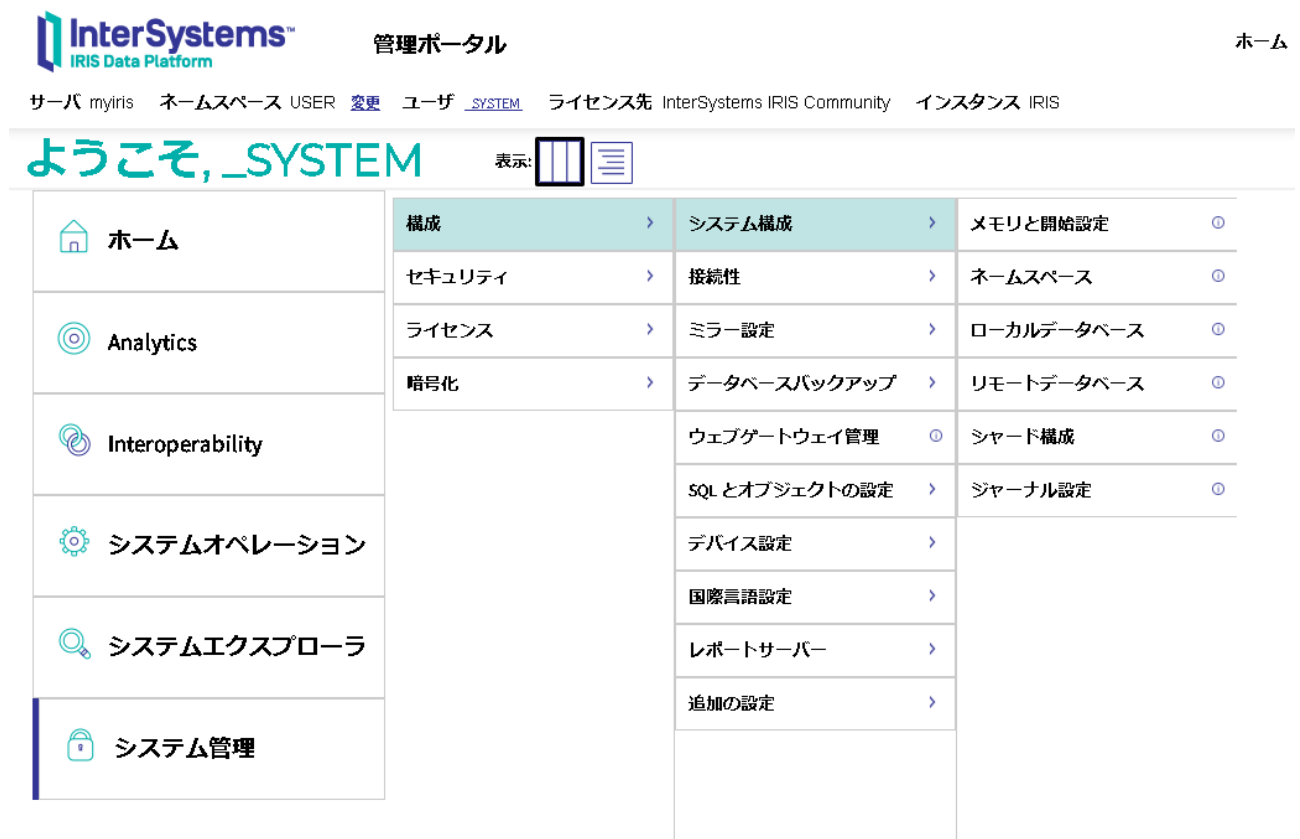


図 2-24 システム管理ポータル

システム管理（システム管理者タスク）

システムの構成情報（ネームスペースおよびデータベースの作成、ECP（Enterprise Cache Protocol）の設定、起動環境調整用項目など）、セキュリティ管理、ライセンス管理、データベースの暗号化など。

システムエクスプローラ（データ管理タスク）

クラス定義、ルーチン、グローバル変数の一覧やインポート／エクスポート機能。また、SQL 文操作メニューなど。

システムオペレーション（システム運用タスク）

バックアップ、ジャーナル管理、ロック管理など。

ドキュメント…IRIS のヘルプドキュメント

このドキュメントは、IRIS のウェブ技術である Caché Server Pages（CSP）を使用しています。

<https://docs.intersystems.com/irislatestj/csp/docbook/DocBook.UI.Page.cls>

なお、図 2-25 のドキュメントページは、IRIS 2019.4 バージョンのものであります。

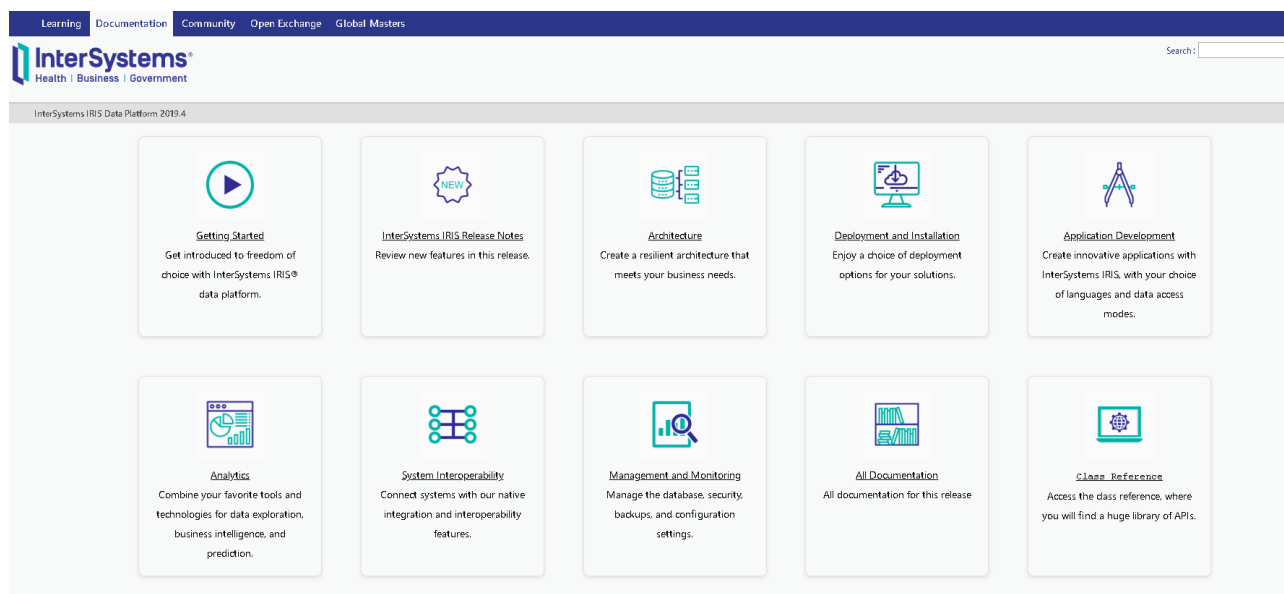


図 2-25 IRIS ドキュメント

第 3 章 InterSystems ObjectScript とは

InterSystems ObjectScript は、複雑なビジネスアプリケーションの迅速な開発のために設計されたオブジェクト指向型のプログラミング言語です。

InterSystems ObjectScript のソースコードは、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の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。

IRIS2021.2 からは Embedded Python が追加され、Python を使用した IRIS のサーバーサイドプログラミングも可能になりました。

Python でプログラミングを行う場合は、必ずしも ObjectScript 言語を習得する必要はありません。

3-1 言葉の定義

ルーチン

ルーチンは ObjectScript コードのブロック（かたまり）です。

通常スタジオまたは Visual Studio Code で記述します。

クラス

クラスは複数のメソッドを含むことができます。メソッドは通常 ObjectScript コードを実行します。

クラスはそのほかにプロパティ、パラメータ、クエリー、インデックス、トリガーなどを持つことができます。

クラスはスタジオまたは Visual Studio Code で開発します。

グローバル

グローバル（またはグローバル変数）は、永続性のある変数です。

グローバルは IRIS のデータベースに格納されます。

グローバルは通常 ObjectScript のコマンドで値を取得します。

グローバルの値は管理ポータルで操作することもできます。

IRIS データベースに格納されるものは全てグローバルとして格納されます。

- ・ クラス定義、メソッド/ルーチンコードなど

最終的には、データへの全ての変更は、関連するグローバル変数の変更になります。

ObjectScript、SQL、Java、その他の方法など、データを変更する方法に関わらず、最下層で操作されるのはグローバル構造です。

通常は、ユーザー（開発者）は、クラスを通してデータにアクセスします。

彼らが特定のグローバル構造を知る必要はありません。

IRIS は、内部構造としてデータベースストレージにバランストツリー構造を使います。

このことには、いくつかの含みがあります。

ストレージは論理的にまばらに管理されます。

但し、データのあるノードだけが物理的に格納されます。

通常、IRIS はリレーショナルデータベースに比較して少ないディスク容量でデータを格納できます。

その階層的な構造により、リレーショナルデータベースに比較して現実のデータを本来に近い形でモデル化することもできます。

一回の I/O で大量のデータの読み書きができますので、より大量のデータをメモリーにキャッシュできます。

データの再編成、インデックスの再作成、データ圧縮などの作業は必要ありません。

IRIS SQL

IRIS は、InterSystems SQL と呼ばれる SQL の実装を提供しています。

InterSystems SQL は、2 つの方法で利用できます。

動的 SQL (%SQL.Statement システムクラスを利用する)

埋め込み SQL

マクロとインクルードファイル

マクロはスタジオ内で利用できます。

マクロは `#define` ディレクティブで定義します。

そして `$$$` を前に付けて呼出します。

```
#define StringMacro "こんにちは！"  
write $$$StringMacro
```

インクルードファイルは、マクロ定義だけを含むファイルです。

`#include` ディレクティブを使ってインクルードファイルをルーチン、メソッド、クラスに含めることができます。

MAC,INT,OBJ,デプロイコード

記述したコード（ルーチン、メソッド、クラス内など）が最終的にデータベースエンジンで実行可能となるまでには複数の手順を通ります。

全てはまず最初に **MAC** コード ("マクロ") に変換されます。

これは **ObjectScript** ルーチンです。

コンパイラーがその **MAC** コードを **INT** コード ("中間") に変換します。

このコードは最下層のグローバルを操作するコードです。

このコードは参照可能で中身を見ることができ、デバッグの際に便利です。

最後に **INT** コードは **OBJ** コード ("オブジェクト") に変換されます。

そしてデータベースエンジンで実行可能となります。

一度 **OBJ** コードが生成されると、**MAC** コードと **INT** コードは削除可能です。

（セキュリティや著作権保護のため）

この状態をデプロイモードと呼びます。

OBJ コードの中身はバイナリデータなので、直接人が理解できる形式で見ることができません。

3-2 InterSystems ObjectScript プログラミング例

簡単なルーチンを書いてみましょう

スタジオ（または Visual Studio Code）を起動して、以下のルーチンを書いてみましょう。

（ファイル(F)>新規作成 カテゴリから一般を選択 テンプレートから ObjectScript ルーチンを選択）

```
interesting() public {  
    write "今日の日付: "_$ZDATE($HOROLOG,3)  
    write !,"インストールのバージョン: "_$ZVERSION  
    write !,"ユーザー名: "_$USERNAME  
    write !,"セキュリティロール: "_$ROLES  
}
```

ルーチンをコンパイルして名前を"demoroutine.MAC"にして保存します。

（ビルド(B)>コンパイル(C)）

ターミナルを起動して、次のようにタイプします。

```
USER>do interesting^demoroutine()
```

以下のように表示されるはずです。

```
今日の日付: 2015-07-29  
インストールのバージョン: Cache for Windows (x86-64) 2015.2 (Build 599U)  
Mon Apr 6 2015 19:17:55 EDT  
ユーザー名: UnknownUser  
セキュリティロール: %All
```

次にスタジオで以下のプロシジャを追加してみましょう。

```
getnumbername(number) {  
  set name=$CASE(number,1:"一",2:"二",3:"三",  
4:"四",5:"五",6:"六",7:"七",8:"八",  
9:"九",10:"十",:"その他")  
  quit name  
}
```

ルーチンをコンパイルします。

ターミナルで次のようにタイプします。

```
USER>do getnumbername^demoroutine
```

以下のように表示されるはずです。

```
<NOLINE>
```

このプロシジャは **public** ではないので、ルーチンの外から実行することができません。
その結果として行がない (<NOLINE>) というエラーが表示されました。

続いて以下のプロシジャを追加しましょう。

```
input() public {  
    read "1 から 10 までの数字を入力してください:",input  
    set name=$$getnumbername(input)  
    write !, "この数字の名前は: "_name  
}
```

ルーチンをコンパイルします。

ターミナルで次のようにタイプします。

```
USER>do input^demoroutine
```

```
1 から 10 までの数字を入力してください: 5  
この数字の名前は: 五
```

最後に以下のプロシジャを追加してみましょう。

```
random() public {  
    set rand=$RANDOM(10)+1 // rand は 1 から 10 の範囲の整数  
    write "乱数は: "_rand  
    set name=$$getnumbername(rand)  
    write !, "この数字の名前: "_name  
}
```

このルーチンは、いくつかのプログラム要素を含んでいます。

3-3 プログラム要素

ラベル (random, input, getnumbername, interesting など)

これらはプロシジャの開始点となります。

"random"と"input"は、getnumbername プロシジャを呼び出しています。

"write", "quit", "set", "read"はコマンドです。

"_"は、結合のための演算子です。

\$CASE と\$ZDATE はシステム関数です。

\$HOROLOG, \$ZVERSION, \$USERNAME, \$ROLES は、システム変数です。

コメント

コメントには、//または/* */を使います。

プロシジャ、関数、サブルーチン

ラベルは、以下のコードブロックのエントリーポイントです。

- プロシジャ
- 関数
- サブルーチン

これらは、パブリック、またはプライベートとして定義できます。

入力引数を受け取ることができます。

サブルーチンを除いて値を返すこともできます。

プロシジャ形式:

```
ラベル(引数) スコープキーワード [デフォルトはプライベート] {  
    ゼロまたは複数行のコード  
    [QUIT オプションの戻り値]}
```

関数形式:

```
ラベル(引数) スコープキーワード [デフォルトはパブリック]  
    ゼロまたは複数行のコード  
QUIT オプションの戻り値
```


サブルーチン形式:

ラベル(引数) スコープキーワード [デフォルトはパブリック]
 ゼロまたは複数行のコード
QUIT

変数のスコープ制御の観点では、プロシジャが他のプログラミング言語で一般的に使用される変数スコープを持つので、推奨する形式です。

関数形式およびサブルーチン形式は、レガシー（過去の遺産）な形式です。

新規に開発するアプリケーションでは、プロシジャ形式を使用することを推奨します。

変数

変数は 2 種類あります：

ローカル変数は、データをメモリーに保持します。

これらの変数のスコープを限定することができます。

ローカル変数名の最初の文字は、レター（アルファベットの句読点、数字を除く文字、日本語の任意の文字）になります。

グローバル変数は、データをデータベースに保持します。

明示的に削除（Kill）しない限り、変数のスコープは、システム（ネームスペース内）全体です。

グローバル変数名は、^で始まります。

グローバル変数名には日本語を使用することができません。

変数に格納できる文字列の最大長は、3,641,144 文字になります。

変数は `set` コマンドで作成されます。

`set x = 5`

まだ `set` されていない変数を使おうとすると、<UNDEFINED>エラーを受け取ります。

配列

配列は、多次元変数で他の変数とは、違う形で取り扱うことができます。

ターミナルで以下のコマンドを入力してみましょう。

```
set array(1)= “ジョン”  
set array(2)= “ポール”  
set array(3)= “ジョージ”  
set array(4)= “リンゴ”  
zwrite array
```

```
set array( “名前” )= “患者名”  
set array( “名前” , “名” )= “太郎”  
set array( “名前” , “氏” )= “山田”  
zwrite array
```

```
kill array  
zwrite array
```

```
set ^array(1)= “猫”  
set ^array(2)= “犬”  
set ^array(3)= “象”  
zwrite ^array
```

演算子

ObjectScript は、大抵のプログラミング言語で共通に使われる演算子を使います。

```
+ - / * \ # ** _ = > < >= <= && ||
```

パターンマッチング：？（例： 3N1"- "2N1"- "4N）

コンテナ(含む)：[（例： x[y）

フォロー（続く）：] （例： x]y）

ObjectScript は、算術的計算を厳密に左から右に評価します。

異なる実行順を強制するには括弧を使います。

```
write 5 + 7 / 2 - 1 * 3
```

パターンマッチング

パターンマッチを実行するためには、以下の構造でパターンマッチを指定します。

```
<数字><パターンコード>[<数字><パターンコード>...]
```

以下にいくつかの数字指定のサンプルを示します。

```
1(次に続くものがちょうど1つ)
1.3（次に続くものが1から3の間）
1.（以下に続くものが1以上）
.3（以下に続くものがゼロから3の間）
```

以下にいくつかのパターンコードを示します。

A(アルファベット)
N (数字)
AN (アルファベットと数字のいずれかまたはその組み合わせ)
U (大文字)
L (小文字)
P (句読点)
ZFWCHARZ (日本語の全角文字)
ZHWKATAZ (日本語の半角カタカナ文字)
E (任意の文字)
"abc" (リテラル文字列"abc")

例えば、値が米国の社会保障番号のパターンにマッチするかどうかチェックしたい場合、そのパターンは、3桁の数字、1つの"-", 2桁の数字、1つの"-", 4桁の数字です。そのパターンマッチは、以下のように表現できます。

```
3N1"-2N1"-4N
```

また値が米国の電話番号のパターンにマッチするかどうかチェックしたい場合、そのパターンは、1つの"(", 3桁の数字、1つの") ", 3桁の数字、1つの"-", 4桁の数字です。このパターンマッチは、以下のように表現できます。

```
1 "(" 3N1 " ) " 3N1 "- 4N
```

パターンマッチングを行うためには、?を使用します。

```
set ssn = "123-45-6789"  
if ssn?3N1"-2N1"-4N {write "正しい ssn です。"}
```

コマンド

オブジェクトスクリプトは、多くのコマンドをサポートします。

その多くは、他言語の同等なコマンドと同じようなことを実行します。

```
set, kill, read, write  
if {...} elseif {...} else {...}  
for, while, do/while  
quit/quit x  
try/catch  
open, use, close  
tstart, tcommit, trollback  
zbreak  
hang  
merge [ ^newglobal = array ]
```

set の使用法

変数の生成を行うために **Set** コマンドを使うことができます。

また、**Set** を使って数値演算を行うこともできます。

```
Set x = 4 + 2
Write x
Set x = "-5 度"
Write x
Write +x
Write -x
Set x = 4 * 2
Write x
Set x = 5 / 2
Write x
Set x = 5 ** 2
Write x
Set x = 5 \ 2
Write x
Set x = 5 # 2
Write x
```


If/Elseif/Else

条件の評価に基づいて処理を分岐させるために"If コードブロック"を使います。

そのシンタックスは：

```
if 条件 1 {コード} elseif 条件 2 {コード} else {コード}
```

elseif は、オプションです。 1 つ以上の elseif 文を使うことができます。

else もオプションです。ただし、else は一回だけ使用できます。

```
read "チーム:", t if (t="ジャガーズ") { write !, "行けジャグス!" }  
else { write !, "ブー ", t, "!" }
```

For の使用

```
for i = 1:1:8 { write !, "I ", i, " the sandbox." } write !  
for b = "ジョン","ポール","ジョージ","リンゴ" { write !, b, "はドラマーですか? " read yn }
```

以下の例では、終了ポイントがありません。

なので、エンドレスループを避けるためにループの中に **Quit** コマンドを入れなければなりません。

```
for i=1:1 { read !,"誰がスーパーボールで勝ったのですか? ",a if a="SEAHAWKS"  
    { write i,"回で、"正解しました!"  
Quit
```

引数なし For

引数なしの For 文を実行することもできます。

```
for {任意のコード}
```

ループ内に quit 文がないと、このコードは永久に実行し続けます。

この文を使う際には、コードループ内に条件付 Quit 文を使用したほうが良いでしょう。

そうしないとエンドレスループになります。

```
set x = 8 for {write "x = ",x,! set x = x - 1 quit:x=3} write "終了!"
```

While と Do/While

While と Do/While は、コードブロックを複数回実行するための 2 つの異なった方法です。

```
do {コード} while 条件
```

```
while 条件 {コード}
```

Try/Catch を使う

コードブロックの実行をモニターするために Try/Catch を使うことができます。
そして実行エラーが発生した時に制御をほかの場所に移します。

```
try { write "こんにちわ", 1/0 } catch { write !,"エラーコード: ", $ZERROR }
```

\$SYSTEM メソッド/変数

\$SYSTEM によって広範囲にわたるシステムメソッドとシステム変数を提供します。
これらの詳細は、ターミナルプロンプトから以下のコマンドをタイプして下さい:

```
>do $system.OBJ.Help()
```

このパッケージ内の特定のメソッドに関する情報を得ることもできます。

```
>do $system.OBJ.Help("Compile")
```

ロック

ObjectScript は、お使いの環境内の同時実行性を管理するユーティリティを提供しています。

これらは、"lock" コマンドによって提供されます。

ロックは、"ロックテーブル"によって管理されます。

このテーブルは、固定のサイズを持ち、満杯になると、"LOCK TABLE FULL" というメッセージが `messages.log` というファイルに書き込まれます。

配列の1つのノードをロックすることは、その直接の親とともにそのノードの子供もロックします。

ロックには2つのタイプがあります。

排他ロックは、他の全てのプロセスがノードをアクセスできないようにします。

内容を更新する際に使えます。

共有ロックは、他の複数プロセスがノードを読むことを許可しますが、更新は許可しません。

その内容を読んでいて、使用中には更新してほしくない場合に使えます。

システム関数

ObjectScript は、多数のシステム関数を提供します。

各関数は、\$で始まります。

\$CASE 関数を使用したルーチン例を提示しました。

\$DATA は特定の変数が存在するかどうかを教えてください。

\$GET は変数の値を返します。

\$LISTBUILD はリスト配列を作ります。\$LIST はリストの要素を表示します。

\$LISTLENGTH はその配列に何個の要素があるかを返します。

\$LISTIND は、ある要素の位置を返します。

リストにない項目に\$LIST コマンドを発行しようとすると、<LIST>エラーになります。

リスト

次にリスト操作の例を示します。

リストを操作する様々な関数があります。

括弧()内は省略形

\$LISTBUILD (\$LB) リストの作成

```
set mail = $LISTBUILD("西新宿","新宿区","東京都","1600012")
```

\$LIST(\$LI) リストから項目を取得

```
write $LIST(mail,2)
```

\$LISTLENGTH(\$LL) リスト内の項目数を返す

```
write $LISTLENGTH(mail)
```

\$LISTFIND(\$LF) リスト項目の位置を取得する

```
write $LISTFIND(mail,"東京都")
```

\$LISTVALID(\$LV) リストを操作している場合に 1 を返す

リスト形式ではない変数に対してリスト関数を実行しようとする、<LIST>エラーになります。

```
write $LISTBUILD(mail)
```

\$LISTFROMSTRING(\$LFS) 区切り文字列から新しいリストを作成する

```
set x = "ドナルド^ダック^ディズニーランド^東京"  
set newlist = $LISTFROMSTRING(x,"^")
```

\$LISTTOSTRING(\$LTS) リストから区切り文字列を作成する

```
set xx = $LISTTOSTRING(newlist,"*")
```

\$LISTSAME(\$LS) 2つのリストを比較し、同じならば1を返す。
違うならば、0を返す

```
write $LISTTOSTRING(mail,newlist)
```

その他のシステム関数

さらに以下のようなシステム関数が用意されています。

\$EXTRACT は、部分文字列を返すまたは置き換える

\$FIND は、部分文字列の場所を返す

\$JUSTIFY は、文字列を右揃えにする

\$TRANSLATE は、1 文字単位で置換する

\$REPLACE は、文字列単位で置換する

\$PIECE は、区切り文字列の入力から部分文字列を返す

\$LENGTH は文字列の長さを返す

\$CHAR(9)はタブキーを返す

\$CHAR(10)はラインフィードを返す

\$CHAR(13)はキャリッジリターンを返す

\$CHAR(13,10)はキャリッジリターン/ラインフィードを返す

\$ASCII は入力文字の ASCII 値を返す

日付と時間

現在のタイムスタンプは、システム変数\$HOROLOG(\$H)で表現されます。

\$H は 3 つの部分より成り立っています。

1. 1840 年 12 月 31 日を起点とした経過日数を示す整数値
2. カンマ
3. 深夜を起点とした秒数を示す整数値

2014 年 5 月 1 日 午後 12:00:00 を表す \$H 値は、633308,43200 になります。

UTC 時間を表すシステム変数\$ZTIMESTAMP もあります。

日付

システム関数\$ZDATE は \$H 形式を日付に変換します。

```
write $ZDATE($H,1)
write $ZDATE($H,3)
write $ZDATE($H,16)
```

システム関数\$ZDATETIME は日付と時間に対して同様のことを行います

```
write $ZDATETIME($H,1)
write $ZDATETIME($H,3)
write $ZDATETIME($H,16)
```

日付と時間を \$H 形式に変換する同じような\$ZDATEH と\$ZDATETIMEH システム関数もあります。

追加のプログラミング

ルーチンとプロシジャを試してみる

スタジオを起動して、前に作成した **demoroutine** という名前のルーチンに以下のプロシジャを追加してみましょう。

```
square(input) public {  
    set answer = input * input  
    set input = input + 10  
    quit answer  
}
```

そのルーチンをコンパイルしましょう

以下のコマンドをターミナルで実行してみましょう

```
set x = 5  
write $$square^demoroutine(x)  
write x
```

引数を参照渡しすることもできます。

この場合、送られた変数自身がサブルーチンで変更されます。

```
set x = 5  
write $$square^demoroutine(.x)  
write x
```

エラーメッセージ

書き終わったコードを実行してみると以下のようなエラーが表示されることがあります。

<SYNTAX>: コマンドの記述に間違いがあるか、コマンドに余計なスペースがあります。

<NOROUTINE>: 実行しようとするルーチンが存在しません。

<NOLINE>: 実行しようとするプロシジャが存在しません。

<UNDEFINED>: 定義されていない変数を使用しようとしています。

以下の内容をタイプしてください。 どんなエラーになりますか？ 何故エラーが返されるのでしょうか？

```
write "hi"  
write x  
do ^%DD  
do ALL^%DD  
do ALLL^%FREECNT  
write xxx
```

3-4 IRIS Object 基本コンセプト

オブジェクトとプロパティ

オブジェクトとは一緒に 1 つの集まりとして格納したり渡されたりする値の集まりのためのコンテナです。

例えば、"Sample.Person"というクラスはその中にいくつかのサンプルデータを持っています。（ネームスペース `user` にあります）

そのテーブルをオープンして、最初の行に等しい値を変数にセットすると、その変数はオブジェクトになります。

Sample.Person データベースには、"Name"と"DOB"というプロパティがあるので、その変数は以下のように利用します。

```
set person = ##class(Sample.Person).%OpenId(1)
write person.Name
write person.DOB
```

この変数の値を操作することができて、その後にデータをデータベースに書き戻すことができます。

```
set person.Name = "ミッキーマウス"
set person.DOB = $Piece($H,",",1)
do person.%Save()
```

データベースを SQL で探索してみて、一連のプログラミングの実行結果を確かめてみましょう。

データベースに新しいレコードを追加することもできます。

```
set newperson = ##class(Sample.Person).%New()
set newperson.Name = "ミニーマウス"
set newperson.SSN = "123-45-6789"
do newperson.%Save()
```

データベースを SQL で探索してみて、一連のプログラミングの実行結果を確かめてみましょう。

```
select * from Sample.Person where Name [ 'マウス'
```

メソッド

2つのタイプのメソッドがあります。

インスタンスメソッドは、クラスのオープンしているインスタンスに対して操作します。

呼ばれた時にそのインスタンスに対して何かを行うか、インスタンスのデータをフィードしてそのメソッドを呼び出します。

インスタンスメソッドのシグニチャーは"Method"という言葉で始まります。

%Save()はインスタンスメソッドです。

クラスメソッドは、実行する前にインスタンスをオープンしておく必要はありません。クラスメソッドのシグニチャーは、"ClassMethod"という言葉で始まります。

%New()と%OpenId()はクラスメソッドです。

3-5 InterSystems ObjectScript を使った簡単なプログラミング

それでは、InterSystems ObjectScript の実際のプログラミング例を挙げてみましょう。

このプログラム例のデータ構造と処理概要は次のようになります。

データ構造

注文グローバル注文ヘッダ注文 ID、顧客 ID、注文日

注文明細注文 ID、明細 NO、商品 ID、単価、割引、数量

商品グローバル商品 ID、商品名、単価

顧客グローバル顧客 ID、顧客名、住所、電話番号、割引率

注文インデックスグローバル商品 ID、顧客 ID

処理概要

注文処理

- ① 顧客 ID の入力
- ② 商品のリストの表示
- ③ 商品 ID の指定
- ④ 数量指定
- ⑤ ③～④の繰り返し
- ⑥ 注文内容表示
- ⑦ 確定

注文確認

- ① 顧客 ID の指定
- ② 顧客 ID に関連づけられた注文 ID の表示
- ③ 注文 ID の指定
- ④ 注文内容の表示

商品別売り上げ実績

- ① 商品 ID の指定
- ② 商品 ID が含まれる注文の検索
- ③ 商品 ID が含まれる明細の取得

- ④ 売上げの加算
- ⑤ ②～④の繰り返し
- ⑥ 実績の表示

実際のプログラミング例

それでは実際のプログラミング例を挙げてみましょう。以降の章で、それぞれの章の内容に沿った形に、このプログラムを修正していきます。まずは、基本となる **InterSystems ObjectScript** について把握しておきましょう。（セットアップ環境には、予めここで説明している内容は、既に含まれています。）

```

placeOrder(customerId) PUBLIC {
if customerId="" quit
if '$data(^customer(customerId)) write "顧客 ID が登録されていません" quit
set customerInfo = ^customer(customerId)
set customerName = $list(customerInfo,1)
write customerName_"様、いつもご利用ありがとうございます",!!
set pid = ""
write "商品 ID",?20, "商品名",?40,"単価",!
write "-----",!
for {
set pid = $order(^product(pid))
if pid = "" quit
set productInfo = ^product(pid)
set productName = $list(productInfo,1)
set unitPrice = $list(productInfo,2)
write pid,?20,productName,?40,unitPrice,!
}
set itemNo = 0
write !
for {
read "注文したい商品 ID を入力してください  ", pid,!
if pid="" quit
if '$data(^product(pid)) write "商品が存在しません",! continue
set productInfo = ^product(pid)
set unitPrice = $list(productInfo,2)
set discount = $list(customerInfo,4)
read "数量を入力してください  ", amount,!
if amount'?.N write "正しい数値を入力してください",! continue
set itemNo = itemNo + 1
set items(itemNo) = $listbuild(pid,unitPrice,discount,amount)
}
if itemNo = 0 quit
tstart

```



```

try {
set orderId = $increment(^order)
set ^orderI("customerIndex",customerId,orderId) = ""
set ^order(orderId) = $listbuild(customerId,$piece($horolog,",",1))
for i = 1:1:itemNo {
set ^order(orderId,i) = items(i)
set pid = $list(items(i),1)
set ^orderI("productIndex",pid,orderId) = ""
}
}

catch error {
trollback
}

tcommit

}

orderByCustomer(customerId) public {
if customerId="" quit
set customerName = $list(^customer(customerId),1)
write customerName_"様のご注文履歴は、次のようになっています",!!
write "注文 ID",?20,"注文日",!
write "-----",!
set oid = ""
for {
set oid = $order(^orderI("customerIndex",customerId,oid))
if oid = "" quit
set orderDate = $list(^order(oid),2)
write oid,?20,$zdate(orderDate,3),!
}
write !
read "内容を表示したい注文番号を入力してください",oid,!!
if oid = "" quit
if '$data(^order(oid)) write "注文が存在しません",! quit
write "ご注文内容は、以下のとおりです",!!

```

```

set orderInfo = ^order(oid)
set orderDate = $list(orderInfo,2)
write "お客様名",?20,"注文日",!
write customerName,?20,orderDate,!
write "明細番号",?10,"商品 ID",?20,"商品名",?40,"単価",?50,"割引率",?60,"数量",?70,"
小計",!
write "-----
----",!
set lid = ""
for {
set lid = $order(^order(oid,lid))
if lid = "" quit
set item = ^order(oid,lid)
set pid = $list(item,1)
set productName = $list($get(^product(pid)),1)
set unitPrice = $list(item,2)
set discount = $list(item,3)
set amount = $list(item,4)
set subTotal = $normalize((unitPrice * amount * discount / 100),0)
write lid,?10,pid,?20,productName,?40,unitPrice,?50,discount,?60,amount,?7
0,subTotal,!
}
}
totalByProduct(productId) public {
if productId = "" quit
set oid = "",total = 0
for {
set oid = $order(^orderI("productIndex",productId,oid))
if oid = "" quit
set subTotal = 0
set itemNo = ""
for {
set itemNo = $order(^order(oid,itemNo))
if itemNo = "" quit

```

```
set itemInfo = ^order(oid,itemNo)
set pid = $list(itemInfo,1)
if pid = productId {
set unitPrice = $list(itemInfo,2)
set discount = $list(itemInfo,3)
set amount = $list(itemInfo,4)
set subTotal = subTotal + $normalize((unitPrice * discount / 100 *
amount),0)
}
}

set total = total + subTotal
}
write "この商品の売り上げ累計は、",total,"円です",!
}

populate() public {
set ^product(1) = $listbuild("シャンプー",500)
set ^product(2) = $listbuild("リンス",600)
set ^product(3) = $listbuild("ボディークリーム",400)
set ^product(4) = $listbuild("スクラブウォッシュ",800)
set ^product(5) = $listbuild("ヘアジェル",1200)
set ^customer(1) = $listbuild("太田 明","東京都新宿区西新宿 6-10-11","03-5321-6201",90)
set ^customer(2) = $listbuild("前島 健作","東京都新宿区西新宿 8-10-11","03-5322-6209",90)
}
```

第 4 章 IRIS のオブジェクトモデルとは

4-1 IRIS オブジェクトの概要

本節では、IRIS オブジェクトの基本概念であるクラス辞書とクラスライブラリについて説明します。

IRIS オブジェクトアーキテクチャ

IRIS オブジェクトには、次のようなコンポーネントが含まれています。

クラス辞書

クラス定義のリポジトリ（メタデータ）。このリポジトリは、IRIS データベース内に保存されます。IRIS SQL エンジンによっても使用され、オブジェクトや IRIS データへのリレーショナルアクセスに対して責任を持ちます。

クラスコンパイラ

クラス定義を、実行可能なコードに変換する一連のプログラムです。

オブジェクトランタイムシステム

実行中のプログラム内でオブジェクト操作（オブジェクトのインスタンス化、メソッドの呼び出し、多態性など）をサポートする、IRIS 仮想マシンの組み込み機能のことです。

IRIS クラスライブラリ

IRIS のインストールとともに組み込まれる一連のクラスです。これには、アプリケーション内で直接使用するためのクラス（電子メールクラスなど）と、ユーザ定義のクラス（永続またはデータ型）に基本的な振る舞いを提供するためのクラスなどが含まれます。

言語バインディング

IRIS オブジェクトへの外部アクセスを提供する、コードジェネレータと実行時コンポーネントの組み合わせです。バインディングの種類として、IRIS eXTreme などがあります。

ゲートウェイ

IRIS オブジェクトに外部システムに対するアクセスを提供する、サーバ側のコンポーネントです。このゲートウェイには、IRIS SQL ゲートウェイと .Net ゲートウェイなどが含まれます。

クラス定義とクラス辞書

すべてのクラスには、クラス全体の特性（スーパークラスなど）と、どのメンバ（プロパティやメソッドなど）を含むかを指定する「クラス定義」が含まれます。これらのクラス定義は、IRIS データベース内に保存される「クラス辞書」内に含まれています。

クラス定義の作成

クラス定義を作成するには、表 4-1 にある 4 種類の方法があります。

表 4-1：クラス定義の作成方法

方法	概要
Caché スタジオ	Caché スタジオという開発環境を使う
XML	外部的にクラス定義を格納したり（ソースコントロールシステムなどで）、単純にコードを共有したりするのに使用する。適切な XML クラス定義を生成し、それを Caché にロードすることによって、新規のクラス定義を作成することも可能
Caché API	Caché は、クラス辞書へのオブジェクトアクセスを提供する、一連の「クラス定義クラス」を含む。これらを使用して、クラス定義の参照や変更、作成を行うことが可能
SQL DDL	DDL 文によるすべてのリレーショナルテーブル定義は、同等のクラス定義に自動的に変換され、クラス辞書内に配置される

また、IRIS は、クラス定義を自動的に作成するウィザード等も含んでいます。

クラス辞書

すべての IRIS のネームスペースには、そのネームスペースで使用可能なクラスを定義する、独自のクラス辞書が含まれています。

その中でも特別なデータベースとして「IRISLIB」があります。

ここには、IRIS クラスライブラリのクラスに対する、実行可能なコードや定義が含まれています。

これらのクラスは **system** クラスとして参照され、すべてのクラスには % 記号で始まる名前が付いています（厳密に言うと、%記号で始まるパッケージ名を持っています）。

すべてのネームスペースはクラス定義のために自動的に構成されるため、それぞれのクラス辞書は、独自のクラスだけではなく、システムクラスや IRISLIB データベース内のコードも利用できます。

このメカニズムにより、すべてのネームスペースは IRIS クラスライブラリ内のクラスも直接使用できるのです。

なお、クラス辞書には、次の 2 つの異なるデータ型が含まれています。

- データ定義：ユーザが作成する実際のクラス定義
- コンパイルデータ：クラス定義のコンパイルの結果として生成されるデータ

コンパイルデータには、継承解析の結果が含まれます。

つまり、指定されたクラスに対して定義および継承されたすべてのメンバをリストしているのです。

クラスコンパイラは、これを使用して、他のコンパイルをより効率的にします。

アプリケーションも、これを使用して、（適切なインタフェース経由で）クラスメンバに関する実行時の情報を取得できます。

クラス辞書は、そのデータを、一連のグローバル（永続配列）に保存します。

これらの配列の構造は、今後の IRIS のバージョンでは変更されることもありうるため、アプリケーション側ではこれらの構造を直接参照したり変更したりしないでください。

クラスライブラリ

IRIS のクラスライブラリには、事前に組み込まれた一連のクラスが含まれています。

これらは、すべての IRIS システムで、IRISLIB データベース内に自動的にインストールされます。

クラスライブラリのコンテンツは、IRIS が提供するオンラインクラスリファレンスを使用して表示できます（図 4-1）。

%SYSTEM.OBJ

abstract クラス %SYSTEM.OBJ extends [Help](#)

The %SYSTEM.OBJ class provides an interface for managing objects.

You can use it via the special \$system object:

```
Do $system.OBJ.Load("MyFile.xml","ck")
```

Inventory

Parameters	Properties	Methods	Queries	Indices	ForeignKeys	Triggers
		73	2			

Summary

メソッド			
CloseObjects	Compile	CompileAll	CompileAllNamespaces
CompileInfoClass	CompileInfoClose	CompileInfoExecute	CompileInfoFetch
CompileInfoFetchRows	CompileList	CompilePackage	CompileProject
Delete	DeleteAll	DeletePackage	DeleteProject
DisplayError	Dump	Export	ExportAllClasses
ExportAllClassesIndividual	ExportAllClassesToStream	ExportCPP	ExportDynCPP
ExportJava	ExportJavaPackage	ExportDDL	ExportPackage
ExportPackageToStream	ExportPattern	ExportPatternToStream	ExportToStream
ExportUDL	GetClassList	GetConcurrencyMode	GetDependencies
GetPackageList	GetQualifiers	GetTransactionMode	Help
ImportDir	IsUpToDate	IsValidClassname	Load
LoadDir	LoadLanguage	LoadStream	MakeClassDeployed
New	ObjectListClose	ObjectListExecute	ObjectListFetch
Open	OpenId	RebuildExtentIndex	RebuildExtentIndexOne
RedirectBindSrvUserOutput	SaveObjects	SetConcurrencyMode	SetFlags
SetQualifiers	SetTransactionMode	ShowClasses	ShowFlags
ShowMacros	ShowObjects	ShowQualifiers	ShowReferences
UnCompile	UpdateConfigParam	Upgrade	UpgradeAll
ValidateIndices	Version		

図 4-1

クラスライブラリには多くのパッケージが含まれ、それぞれのパッケージにはクラスのファミリーが含まれます。

それらのうち、内部的なクラスは、IRIS オブジェクトを実装の一部として使用し、その他のクラスは、アプリケーションで使用されるように設計されています。

表 4-2 にクラスライブラリの主なパッケージを掲載しました。

表 4-2：クラスライブラリの主なパッケージ

パッケージ	説明
%Activate	アクティベートによって使用されるクラス
%Compiler	クラスコンパイラによって使用される内部クラス
%CSP	Caché Server Pages (CSP) によって使用されるクラス
%csr	標準 CSP ルールを実装する、生成された一連の内部クラス
%Library	振る舞いクラス (%Persistent など) の中心となる一連のクラス。さまざまなデータ型、コレクション、ストリームクラスも含まれます
%Net	さまざまなインターネット関連の機能を提供する、電子メールや HTTP 要求などのクラス
%Projection	ユーザクラスに対するクライアント側のコードを生成するプロジェクションクラス
%SQL	Caché SQL によって使用される内部クラス
%Studio	Caché スタジオ によって使用される内部クラス
%SYSTEM	\$System 特殊変数を經由してアクセスできる、さまざまな System API クラス
%XML	Caché 内でサポートされる XML や SAX を提供するのに使用されるクラス

4-2 オブジェクト指向データベース開発

現代、多くのプログラムコードはオブジェクト指向に基づいており、ビジネスロジックの大部分も、オブジェクトとして実装されています。

IRIS とその他の言語による開発との大きな違いは、アプリケーションのデータベース層の内部構造です。

その他の数多くのデータベースは、従来の方法論（ISAM やリレーショナルモデル）に依存しています。

IRIS は、データベース開発そのものに、強力なオブジェクト指向のプログラミング手法を提供します。

クラスとオブジェクト

クラスは、アプリケーション内のデータと特定のエンティティの振る舞いを説明するテンプレートですが、オブジェクトは、そうしたクラスの特定のインスタンスです。

オブジェクトは、特定のプロセスやセッションのメモリ内で操作されたり、データベースに保存され、データベースから検索したりすることもできます。

IRIS における重要な機能は、新規のクラスを実行時に定義できるということです。

つまり、新規のクラスを生成して、アプリケーション自身で拡張を行うことができます。

抽象化とモデリング

抽象化とは、現実世界のエンティティの必要不可欠な詳細（従業員、送り状、文書など）を取得し、そこからアプリケーション内で使用される表現を定義するプロセスです。

また、モデリングとは、データを抽象化し、それを一連のクラスとして表すプロセスです。

抽象化とモデリングは、オブジェクト指向プログラミングにおいて最も重要な要素で、オブジェクトモデルを使用して、現実世界の実体に対応するアプリケーションコンポーネントを定義できます。

これとは対照的に、従来のリレーショナルデータベースでは、現実世界のエンティティを表現するためには、二次元のテーブルに適合させる必要があり、データと関連する振る舞いに対するメカニズムは存在しません。

継承とポリモフィズム

継承とは、既存のクラスを取得し、その振る舞いの要素を変更することです。

新規のクラスを生成するときに、新しくデータや振る舞いを追加するか、指定したスーパークラスからデータと振る舞いを継承します。

そして、継承された振る舞いを選択的に置換（オーバーライド）することができます。

ポリモフィズム（多態）とは、異なるタイプのオブジェクトの、同じメソッドを呼び出すことができる能力です。

各オブジェクトは、メソッドの独自の特定の実装を実行します。

これにより、既存アプリケーションの新規オブジェクトタイプを、アプリケーションを変更せずに使用できるようになります。

カプセル化

カプセル化とは、エンティティの具体的な実装からは独立して、エンティティのプロパティとメソッドを維持することを意味します。

これにより、エンティティのユーザが、内部動作に関する知識を持つことなく、そのエンティティを使用できるようになります。

オブジェクト指向プログラミングでは、振る舞いをカプセル化するクラスを使用して、カプセル化のプロセスを簡単にしています。

永続オブジェクトがデータベースからロードされる時、そのクラスに定義されたメソッドに自動的に関連付けられます。

拡張

特定のアプリケーション用に振る舞いを調整することを拡張と言います。一般的には、拡張は次のいずれかの方法で実現します。

- **タイプの定義**：アプリケーション固有のデータを表す新規データ型を定義します
- **イベント処理**：アプリケーションで、特定のイベントが発生したときに呼び出される一連のメソッドを定義します
- **サブクラス化**：アプリケーションで、既存のクラスのサブクラスを生成して、以前に開発したコンポーネントを新しい用途に使用できるようにします

オブジェクト永続性

オブジェクト永続性とは、オブジェクトのインスタンスをデータベース内に格納することを意味します。

IRIS は、ネイティブな **SQL** クエリを使用して、オブジェクトに対する完全クエリを実行できます。

IRIS は、オブジェクトをリレーショナル表現には変換せずにネイティブな状態で保存します。

永続オブジェクトは、IRIS のデータベースエンジンに対してオブジェクトを直接保存したり、ロードするメソッドを持っています。

また、IRIS では、スキーマ進化をサポートしているため、データモデルを変更した場合でも、以前に保存されたオブジェクトを新しい定義に対応すべく変換することなく使用できます。

4-3 IRIS オブジェクトモデル

IRIS オブジェクトモデルは、一時的なオブジェクトのみならず、永続オブジェクトの振る舞いと機能も定義が可能です。

オブジェクト参照 OREF、OID、ID

IRIS では、オブジェクトはディスクまたはメモリ上に存在します。ディスク上のオブジェクトとはデータベースに保存したものです。

メモリ内のオブジェクトとはデータベースからロードしたもので、プログラマーが操作可能なものです。

ロードされたものか保存されたものかによって、オブジェクトの参照方法は異なります。

OID 値と ID 値

IRIS では、ユニバーサルオブジェクト識別子 (OID) とローカルオブジェクト識別子 (ID) を区別しています。

OID 値は、データベースに保存されているすべてのオブジェクトを一意に識別します。
ID 値は、特定のエクステンツ (特定クラスのすべてのインスタンスや、クラスのインスタンスとそのサブクラスなど) に属するオブジェクトを一意に識別します。

永続オブジェクトを参照するメソッドは、OID 値と ID 値の 2 つあります。例えば、次のコードのように、オブジェクトインスタンスを開く際に、OID 値を使用することができます。

```
Set object = ##class(MyApp.Person).%Open(oid)
```

または、次のコードのように ID 値を使用して開くこともできます。

```
Set object = ##class(MyApp.Person).%OpenId(id)
```

ID 値を使用する例では、その名前に追加された **Id** が含まれます。

実際のアプリケーションでは、ID 値を使用する場合はほとんどです。

OREF 値と参照カウント

OREF 値とは、オブジェクトの特定のメモリ内インスタンスを参照する値のことです。OREF 値は、メモリ内オブジェクトを参照しますが、そのオブジェクトを参照している項目の数である参照カウントを自動的に管理します。

変数やオブジェクトのプロパティに、そのオブジェクトへの参照を設定すると、オブジェクトの参照カウントは自動的に増分されます。

また、オブジェクトへの参照を放棄すると、参照カウントは減分されます。

そして、参照カウントが 0 に達すると、オブジェクトは自動的に消滅し（メモリから削除され）、%OnClose メソッドが呼び出されます。

クラスタイプ

IRIS は、特別な振る舞いに対する、さまざまなクラスタイプをサポートしています。

クラスは、データ型クラスとオブジェクトクラスに分けられます。

データ型クラスは、文字列や整数、日付などのリテラル値を表し、他のオブジェクトのリテラルプロパティを作成するのに使用されます。

これはプロパティを持たず、インスタンス化されません。

一時オブジェクトクラス

%RegisteredObject から派生したクラスは、登録オブジェクトまたは一時オブジェクトと呼ばれます。

これらのオブジェクトには、このオブジェクトのメモリ内の振る舞いの管理に関する組み込みメソッドが含まれます。

ただし、多くのアプリケーションは、%RegisteredObject を直接継承するのではなく、永続クラスや埋め込み可能なクラスを使用します。

永続オブジェクトクラス

%Persistent から派生したクラスのインスタンス生成は、永続オブジェクトとなります。これらのオブジェクトは、%Persistent クラスを継承する登録オブジェクトです。

永続オブジェクトには、それ自体をデータベースに保存できる機能があります。

データベースからメモリに永続オブジェクトをロードしても、その永続オブジェクトが参照する他のオブジェクトはすぐにはロードされませんが、そのオブジェクトを参照すると、自動的にメモリに展開します（これは、スウィズリングまたは遅延ロードと呼ばれます）。

また、永続オブジェクトがプロパティとして使用された場合は、参照プロパティと呼ばれます。

シリアルオブジェクトクラス

%SerialObject から派生したクラスのインスタンス生成は、シリアルオブジェクトまたは埋め込み可能なオブジェクトとなります。

これらのオブジェクトは、%SerialObject クラスを継承する登録オブジェクトです。

また、これらのオブジェクトはメモリ内で独立して存在できますが、データベースに保存される場合は、永続オブジェクト内への埋め込みとしてのみ存在が可能です。

データタイプクラス

データタイプクラスは、リテラル値を定義し管理します。

オブジェクトクラスとは異なり、データ型は独立しておらず、明示的にインスタンス化されることもありません。

データタイプクラスは、単にそのクラスを含むオブジェクトの属性に過ぎず、プロパティを持つことができません。

データタイプとは、ClassType キーワードが「datatype」に設定されているクラスで、データタイプインタフェースに属する、特定の一連のメソッドを実装することもできます。このインタフェースには、データの妥当性の検証と SQL の相互運用のために設計された、多数の処理が含まれます。

継承

Super キーワードを使用して、既存のクラスを継承することができます。この方法で定義されたクラスは、サブクラスと呼び、派生元のクラスはスーパークラスと呼ばれます。

IRIS のオブジェクトモデルには、クラス定義としてスーパークラスのリストを表す **Super** キーワードがあります。

IRIS スタジオのクラス定義では、このリストは **Extends** キーワードを使用して表現されます。

```
Class User.MyClass Extends %Persistent
{
}
```

このクラスは、スーパークラスの仕様のすべてを継承します。

それには、プロパティ、メソッド、クラスパラメータ、使用可能なクラスキーワード、および、継承されたプロパティと継承されたメソッドのパラメータとキーワードが含まれます。

Final としてマークされている項目を除いて、サブクラスは、継承されたコンポーネントをオーバーライドできます。

スーパークラスからメソッドを継承するクラスに加えて、クラスのプロパティは、システムプロパティの振る舞いクラスから追加メソッドを継承します。

また、データ型属性の場合は、データ型クラスから継承します。

多重継承

IRIS は、Java 等の一般的なオブジェクト指向言語がサポートしていない多重継承をサポートしています。

多重継承によって、クラスは複数のスーパークラスから、その振る舞いやクラスタイプを継承できます。

多重継承を構築するには、**SUPER** キーワードを使用し、次に定義されたクラスが継承するクラスを指定します。

クラスをコンパイルするとき、クラスコンパイラは、スーパークラスのリストに従ってクラスメンバの値を設定します。

クラスメンバとは、そのクラスのパラメータ、メソッド、およびプロパティのことですが、クラスのキーワードではありません。

複数のスーパークラスに同じ名前のメンバがある場合、リストの後方にあるスーパークラスが優先されます。

このオーバーライドは、クラスのキーワードでは発生しないことに注意してください。

つまり、サブクラスは最初のスーパークラスのキーワード値のみを継承します。

クラスのコンパイル

IRIS のクラス定義は、クラスコンパイラによってルーチン（コード）にコンパイルされます。

クラスは、コンパイルされるまでアプリケーションでは使用できません。

IRIS のクラスコンパイラは、C++や Java などの言語とは 2 つの点で大きく異なります。

1 つは、IRIS では、コンパイルの結果は、ファイルシステムではなく、データベース内の共有リポジトリに保存されます。

もう 1 つは、コンパイラは、永続クラスに対するサポートを提供します。

4-4 IRIS クラス

IRIS のクラス定義と、ユーザが実行時にインスタンスを作成できる、いわゆる「コンパイル済みクラス」とは区別します。

クラス定義とは、メンバと呼ばれる多数の要素と、それに関連付けられた値で構成されています。

クラス定義のメンバには、プロパティやメソッドが含まれます。

データ型のクラス定義はメソッドだけを含んでいますが、それはデータ型クラスがリテラル値しか持っていないためです。

クラスを定義するには、主にスタジオまたは、**Visual Studio Code** を使用します。

また、**SQL DDL** 文を使用して **SQL** テーブルを定義することによって、**IRIS** はそれに対応するクラス定義を作成することもできます。

命名基準

クラスと、識別子と呼ばれるクラスメンバは、特定の名前付け規約に従います。

一般識別子ルール

すべての識別子は、コンテキスト内で一意となり、2つのクラスが同時に同じ名前を持つことはできません。

IRIS では、パッケージ、クラス、および、メンバの名前について次のような制限があります。

- 各パッケージの名前には、189 文字までの一意の文字を含めることができる
- 各クラスの名前には、60 文字までの一意の文字を含めることができる
- 各メソッドおよびプロパティの名前には、31 文字までの一意の文字を含めることができる
- 各メンバの正式な名前（固有のメンバ名、クラス名、パッケージ名、およびセパレータがあればそれを含む）は、220 文字以下である必要がある

なお、大文字と小文字は識別されますが、大文字か小文字か以外に違いがない名前を付けることはできません。

例えば、id1 と ID1 を同時に付けることはできません。

また、識別子は、必ずアルファベット文字（日本語文字の利用も可能ですが、様々な制約があり、使用の推奨はしておりません）で始まりますが、2 文字目以降は数字を含むこともあります。

識別子にはスペースや句読点を含むことはできませんが、パッケージ名には「.」文字が含まれることもあります。

特定の識別子は「%」で始まり、これはシステム項目であることを示します。

例えば、IRIS ライブラリに含まれる多くのメソッドやパッケージの名前は、% 文字で始まります。

クラス名

すべてのクラスは、一意に識別できる名前を持っています。完全なクラス名は、パッケージ名とクラス名という 2 つの部分で構成されています。

名前の中で、最後の「.」文字以降がクラス名になります。クラス名は、そのパッケージ内で一意でなければならず、パッケージ名も、IRIS ネームスペース内で一意である必要があります。

永続クラスは、自動的に SQL テーブルとして投影されるので、クラス定義は SQL 予約語ではないテーブル名を指定する必要があります。

もし永続クラスの名前が SQL 予約語だった場合は、クラス定義は `SQLTableName` キーワードに対して予約語でない有効な値を指定する必要があります。

クラスメンバ名

すべてのクラスメンバ（プロパティやメソッドなど）は、そのクラス内で一意の名前を持つ必要があります。

また、永続のメンバは、その識別子として SQL 予約語を使用することはできません。

クラスキーワード

クラス定義は、一連のキーワードを含み、その値はクラスの全体的な振る舞いを決定します。

表 4-3 にあるような各キーワードは、明示的に指定しなければ既定値になります。

表 4-3：クラスキーワード一覧

キーワード	概要
Abstract	このクラスのインスタンスは作成できないことを指定する。データ型クラスに対して、クラスはプロパティタイプとして使用できない
ClassType	クラスの振る舞いを指定する。使用可能な値は、datatype、persistent、および、serial
ClientDataType	データ型クラスが、ActiveXやJavaから使用された場合の型を指定する。既定値はない。データ型クラスでは、クライアントデータ型を指定する必要がある
Description	クラスの説明を指定する。説明は、オンラインクラスリファレンスによって表示される
Final	クラスが最終であることを指定する。このクラスから派生クラスを作成することはできない
SQLRowIDName	SQLによって投影されるオブジェクト行IDに対する、代わりのフィールド名を指定する
SQLTableName	SQLプロジェクションでクラスを識別するために使用されるテーブル名を指定する。既定では、CacheはSQLテーブルの名前としてクラスの名前を使用する
Super	クラスに対する1つまたは複数のスーパークラスを指定する。既定では、クラスはスーパークラスを持たない

クラスパラメータ

クラスパラメータは、指定されたクラスのすべてのオブジェクトに対する特別の定数値を定義します。

クラス定義を作成するときに、このクラスパラメータの値を設定できます。

既定では、各パラメータの値は **NULL** 文字列です。

パラメータの値を設定するには、それに対する値を明示的に提供する必要があります。

コンパイル時に、パラメータの値はクラスのすべてのインスタンスに対して構築されます。

この値は、実行時には変更できません。

4-5 パッケージ

IRIS オブジェクトは、「パッケージ」をサポートしています。パッケージとは、関連するクラスを特定のネームスペース内にグループとしてまとめる方法です。

パッケージ概要

パッケージは、関連するクラスを、共通の名前のもとに 1 つのグループとしてまとめる簡単な方法です。

例えば、アプリケーションに **Accounting** システムと **Inventory** システムがあるとする
と、図 4-2 のように、これらのアプリケーションを構成するクラスを、**Accounting** パッケージと **Inventory** パッケージに編成します。

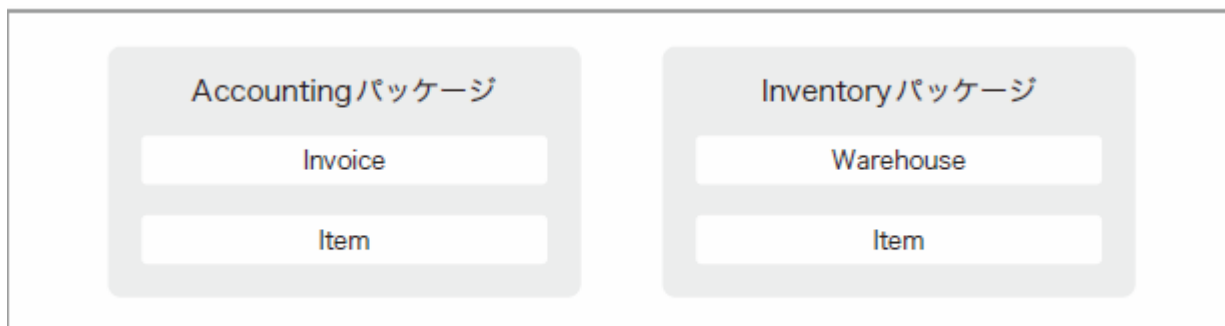


図 4-2: クラスパッケージの例

パッケージは単なる名前付け規約です。

つまり、クラスの命名方法を提供するだけで、それ以上の基本機能を提供するものではありません。

クラスと同様に、パッケージ定義は、IRIS ネームスペース内に存在します。

パッケージは、複数のネームスペースにまたがることができません。

パッケージ名

パッケージ名は、単なる文字列です。パッケージ名にはピリオドが含まれますが、パッケージの階層はありません。

例えば、あるクラスに「**Test.Subtest.TestClass**」と名前を付けた場合、クラス名が「**TestClass**」で、パッケージ名が「**Test.Subtest**」になります。なお、パッケージ名の長さ和使用法には、次のような制限があります。

- パッケージ名はピリオドも含めて、189 文字以下に制限される
- ネームスペース内では、各パッケージ名は一意である必要がある

パッケージを定義する

パッケージは、暗にクラス名の意味を含みます。

パッケージを作成するには、IRIS スタジオで新規クラスウィザードを使用する方法が簡単です。

このウィザードには、新しいパッケージ名を入力できるボックスがあります。

もう 1 つの方法は、単にクラスを作成する方法です。

この場合、パッケージは自動的に定義されます。

例えば、定義内でクラス名にパッケージ名を追加すると、パッケージを定義できます。

パッケージの使用

パッケージ内のクラス名の使用方法は、2 通りあります。

- 完全修飾名を使用する方法（例：**Package.Class**）
- 短いクラス名だけを使用し、クラスがいずれのパッケージに属するのかをクラスコンパイラに解決させる方法

なお、完全修飾名を使うほうが簡単です。

パッケージと SQL

すべてのパッケージは、SQL スキーマに対応しています。

例えば、クラスが **Team.Player**（**Team** パッケージの **Player** クラス）である場合、対応するテーブルは **Team.Player**（**Team** スキーマの **Player** テーブル）です。

既定のパッケージは、**User** で、**SQLUser** スキーマに対応しています。

つまり、**User.Person** という名前のクラスは、**SQLUser.Person** という名前のテーブルに対応します。

パッケージ名にピリオドが含まれる場合は、ピリオドの代わりに「_」（アンダースコア）を使用します。

例えば、

MyTest.Test.MyClass クラス（**MyTest.Test** パッケージの **MyClass** クラス）は、**MyTest_Test.MyClass** テーブル（**MyTest_Test** スキーマの **MyClass** テーブル）になります。

組み込みパッケージ

旧バージョンと互換性を保つため、次の 2 つの組み込みパッケージが用意されています。

- **%Library** : パッケージ名を持たないすべての%クラスは、暗黙的に**%Library** パッケージに属する
- **User** : パッケージ名を持たないすべての非%クラスは、暗黙的に **User** パッケージに属する

4-6 メソッド

メソッドは、オブジェクトと関連し、オブジェクトのインスタンスが実行できる操作です。

メソッドは **IRIS** プロセスで実行されます。

IRIS オブジェクトをクライアントで使っているあいだは、**IRIS** は、適切なサーバープロセス内で自動的にメソッドを呼び出します。

メソッド引数

メソッドは、引数を介して受け取る変数を使って動作します。

メソッドは、任意の数の引数を取ることができ、メソッドが持つ引数とそれぞれの引数のデータ型を指定します。

また、メソッドの定義では、どの引数が参照渡しでどの引数が値渡しであるかを指定します。

引数の既定は値渡しです。

既定値の指定方法

引数の既定値を指定するには、等号に続いて必要な値を設定します。

```
Method Test(flag As %Integer = 0)
{
}
```

参照渡しによる呼び出し

既定では、メソッド引数は値渡しされます。

ByRef 修飾子を使用して、引数を参照渡しすることができます。

```
/// 2つの整数値の交換
Method Swap(ByRef x As %Integer, ByRef y As %Integer)
{
    Set temp = x
    Set x = y
    Set y = temp
}
```

また、**Output** 修飾子を使用して、参照渡しで渡す値を持たない引数を指定することもできます。

可変引数

IRIS では、配列を使って任意の数の引数を渡すことができます。

```
Class TEST.ARGTEST1 Extends %RegisteredObject {
    ClassMethod NewMethod1(Arg... As %String) As %Boolean { kill ^a merge ^a =
        Arg }
}
```

このクラスをコンパイルして、IRIS ターミナルで確認してみます。

```
TEST>DO ##class(TEST.ARGTEST1).NewMethod1(1,2,3,4,5)
TEST>ZWRITE ^a
^a=5
^a(1)=1
^a(2)=2
^a(3)=3
^a(4)=4
^a(5)=5
```

戻り値

各メソッドを定義する際には、それが値を戻すかどうかを指定して、値を戻す場合は戻り値のタイプも指定します。

メソッドの戻りタイプはクラスです。戻りタイプがデータ型クラスの場合、メソッドはリテラル値を戻します。

そうでない場合、メソッドはクラスのインスタンスへの参照を戻します。

メソッドの可視性

インスタンスメソッドは、パブリックまたはプライベートのどちらでも定義できます。プライベートの場合は、それらが属するクラスの任意のインスタンスのメソッドによってのみアクセスできます。

パブリックの場合は、他のメソッドや他の呼び出しからでもアクセス可能です。

クラスメソッドはパブリックのみ定義できます。

PRIVATE キーワードを使用したクラスメソッドは、他の呼び出しのすべてのメソッドからアクセスできますが、生成される **IRIS** クラスリファレンスには表示されません。

IRIS では、プライベートメソッドは継承され、メソッドを定義するクラスのサブクラスから見ることはできません。

他の言語では、これらを **Protected** メソッドと呼ぶことがあります。

メソッドの継承

クラスは、その 1 つまたは複数のスーパークラスからメソッドを継承します。

継承された実装をオーバーライドするメソッドのローカル実装がある場合は、そのシグニチャが一致している必要があります。

メソッド言語

IRIS でメソッドを作成する際の言語は、ObjectScript です。

メソッドキーワード

1 つまたは複数のメソッドキーワードを使用して、メソッド定義を修正できます。

各キーワードはオプションで、明示的に指定されなければ既定値になります。

使用可能なメソッドキーワードは、表 4-4 のとおりです。

表 4-4: メソッドキーワード一覧

メソッドキーワード	概要
ClassMethod	メソッドがクラスメソッドであることを指定する。既定では、メソッドはインスタンスメソッド。サブクラスは、ClassMethod キーワードの値を継承し、その値を修正できない
Description	メソッドの説明。Cachéはこの説明を使用しない。既定では、メソッドに説明はない。サブクラスは、Descriptionメソッドキーワードの値を継承しない
Final	サブクラスがメソッドをオーバーライドできないように指定する。既定では、メソッドは最終ではない。Finalメソッドキーワードは、サブクラスによって継承される
Private	メソッドが、そのクラスまたはサブクラスの他のメソッドによってのみ呼び出せることを指定する。サブクラスはメソッドをプライベートとして継承し、キーワードの値を変更できる。既定では、メソッドはpublic。他の言語では、この種の可視性を説明するためにprotectedという用語を使用し、サブクラスから見えなくするためにprivateという用語を使用していることがあるので注意が必要
ReturnType	メソッドに対する呼び出しによって戻される値のデータ型を指定する。ReturnTypeを空の文字列に設定すると、戻り値がないことになる。ReturnType キーワードの値はサブクラスに継承され、サブクラスで変更できる。ReturnType キーワードの値をオーバーライドする場合は、変更できるのは元の型のサブクラスに限定される。既定では、メソッドには戻り値はない
SQLProc	メソッドがSQLストアードプロシージャであることを指定する。既定では、メソッドはストアードプロシージャではない。ストアードプロシージャはサブクラスに継承される

インスタンスメソッドとクラスメソッド

一般的に、メソッドとは、インスタンスメソッドを意味します。

それは、クラスの特定のインスタンスから起動され、ビジネスロジックの実行やインスタンスに関する情報の表示など、そのインスタンスに関連するアクションを実行します。

インスタンスメソッドに加えて、IRIS オブジェクトはクラスメソッドをサポートします。

クラスメソッドは特定のオブジェクトと必ずしも関連する必要はなく、開いているオブジェクトから呼び出される必要もありません。

メソッドの種類

IRIS は、次の 4 タイプのメソッドをサポートします。

- コードメソッド：実装が単純なコード行であるメソッド
- 式メソッド：ある特定の状況で、クラスコンパイラによって、指定された式の直接インライン代替メソッドに置換される場合があるメソッド
- 呼び出しメソッド：既存の IRIS ルーチンのメソッドラップを作成する、特別なメカニズム
- メソッドジェネレータ：クラスのコンパイル中にクラスコンパイラによって呼び出される小さなプログラム

4-7 プロパティ

プロパティは、オブジェクトの状態を定義するクラスメンバです。

オブジェクトプロパティの値にアクセスし、それを操作する方法は、オブジェクトのクラス定義によって異なります。

プロパティには、次の 2 種類があります。

- 属性プロパティ：値を保持する
- リレーションシッププロパティ：オブジェクト間の関係を保持する

IRIS では、属性プロパティを単にプロパティと呼び、リレーションシッププロパティをリレーションシップと呼びます。

Java と C++ のような多くのオブジェクト言語は、真のプロパティではなく、パブリックのアクセサメソッドによって操作されるプライベート変数を持っています。

IRIS では、プロパティと変数は異なります。

プロパティキーワード

1 つまたは複数のプロパティキーワードを使用して、プロパティ定義を修正できます。

各キーワードはオプションで、明示的に指定されなければ既定値になります。

使用可能なプロパティキーワードは、表 4-5 のとおりです。

表 4-5: プロパティキーワード一覧

プロパティキーワード	概要
Calculated	このプロパティを含むオブジェクトがインスタンス化されるときに、このプロパティにメモリ内ストレージを割り当てないことを指定する。既定では、プロパティは Calculated ではない。サブクラスは Calculated キーワードを継承し、オーバーライドできない
Description	プロパティの説明。Cacheはこの説明を使用しない。既定では、プロパティに説明はない。サブクラスは Description プロパティキーワードの値を継承しない
Final	サブクラスがプロパティをオーバーライドできないことを指定する。既定では、プロパティは Final ではない。Final プロパティキーワードは、サブクラスによって継承される
InitialExpression	プロパティの初期値を指定する。既定では、プロパティは初期値を持たない。サブクラスは InitialExpression キーワードの値を継承し、それをオーバーライドできる
Private	プロパティがプライベートであることを指定する。既定では、プロパティはプライベートではない。サブクラスは Private キーワードの値を継承し、それをオーバーライドできない
Required	ディスクに格納する前に、プロパティの値を設定する必要があることを指定する。既定では、プロパティは Required ではない。サブクラスは Required キーワードの値を継承し、それをオーバーライドできる
Transient	プロパティをデータベースに格納しないことを指定する。既定では、プロパティは Transient ではない。サブクラスは Transient キーワードの値を継承し、それをオーバーライドできない
Type	プロパティに関連付けられるクラス名を指定する。そのクラスはデータ型クラスであり、永続クラス。また、埋め込み可能なクラス。既定では、プロパティのタイプは %String。サブクラスは、Type キーワードの値を継承する

プロパティ可視性

プロパティは、パブリックまたはプライベートとして指定します。

パブリックの場合、それらはどのような場所でもアクセスできます。

プライベートの場合、それらが属するオブジェクトのインスタンスメソッドによってのみアクセスできます。

プロパティの振る舞い

プロパティには、自動的に関連付けられた多くのメソッドがあります。

これらのメソッドは、標準の継承を介して継承されません。

その代わりに、各プロパティに対する一連のメソッドを作成するために、特別なプロパティの振る舞いメカニズムを使用します。

各プロパティは、図 4-3 にあるように、2 つの場所からの一連のメソッドを継承します。

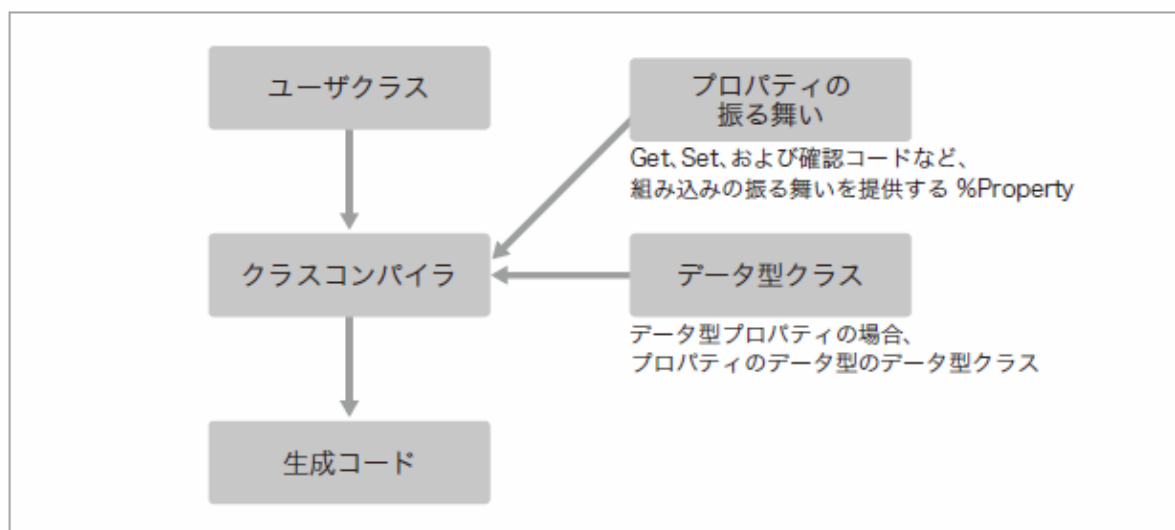


図 4-3: プロパティのメソッド継承

プロパティアクセサ

プロパティを参照する IRIS のドット構文は、値を取得および設定するための `get` と `set` のアクセサメソッドへのインタフェースになっています。

各プロパティに対して、コードが `oref.Prop` を参照するときは常に、システムから提供された `PropGet` メソッド、または `PropSet` メソッドが呼び出されたかのように実行されます。

プロパティの属性

プロパティの属性とは、特定のオブジェクトと関連する値です。

このようなプロパティは、通常は、特定のオブジェクトインスタンスの状態を表します。プロパティのタイプは、プロパティに関連付けられているクラスによって決定されます。既定では、プロパティは `%String` データ型を使用します。

データタイププロパティ

最も単純なプロパティのタイプは、データ型プロパティです。

これはリテラル値で、その振る舞いはプロパティに関連付けられたデータ型クラスによって管理されます。

オブジェクト値プロパティ

オブジェクト値プロパティは、永続オブジェクト、または埋め込みオブジェクトのどちらかのオブジェクトに対する参照です。

いずれの場合も、メモリ上にある間は、プロパティ値は **OREF** です。

ディスク上では、永続オブジェクトへの参照は **OID** になります。

一方で、埋め込みオブジェクトは、埋め込みオブジェクトのプロパティをすべて含む、1つの連続した文字列になります。

コレクションプロパティ

コレクションプロパティには、すべて同じタイプの個々の要素のグループ（またはコレクション）が含まれます。

コレクションプロパティを指定するには、プロパティ定義の最初にコレクションタイプを指定します。IRIS は、**List** と **Array** という 2 種類のコレクションをサポートしています。

ストリームプロパティ

ストリームは、大量の文字またはバイナリデータを含むプロパティの作成に使用されます。

バイナリストリームは、**%Binary** タイプと同種のデータを含み、写真などの大規模なバイナリオブジェクトを格納します。

文字ストリームは、**%String** タイプと同種のデータを含み、大量のテキストの保存を目的としています。

ストリームデータは、プロパティの定義に応じて、外部ファイルまたはグローバルのいずれかに保存できます。

4-8 クラスクエリ

クラスクエリは、指定された条件に適合するクラスのインスタンスを検索するツールを提供します。

これにより、アプリケーションが事前に定義された検索条件を持つことができるように、クラス構成の一部である名付けされた **SQL** 文を作成できるようになります。

例えば、名前などの複数のプロパティによってインスタンスを検索したり、パリからマドリッドまでのすべての航空便など、一連の特定の条件に適合するインスタンスのリストを提供したりできます。

クエリの基本

クラスクエリを作成したり、使用したりする場合のプロセスは、次の手順になります。

- ① クエリの定義
- ② クエリの呼び出し
- ③ クエリが返す `%SQL.StatementResult` オブジェクト（結果セット）からデータをフェッチ
- ④ 結果セットを閉じる

クエリの構造

最も単純なクエリは、**SQL SELECT** 文を含む `%SQLQuery` のタイプです。

このようなクエリは、アプリケーションに対するすべての処理を実行します。

クエリキーワード

クエリキーワードを使用すると、クエリ定義を修正できます。

各キーワードはオプションで、明示的に指定されなければ既定値になります。

使用可能なクエリキーワードは、表 4-6 のとおりです。

表 4-6：クエリキーワード一覧

クエリキーワード	概要
Description	クエリの説明。Cacheは、この説明を使用しない。既定では、クエリの説明はない
Final	サブクラスがクエリをオーバーライドできないことを指定する。既定では、クエリはFinalではない
SQLQuery	クエリが実行されるときに実行するコードを指定する。クエリがSQLを使用する場合は、このキーワードを使用する
SQLProc	SQLストアードプロシージャとしてもアクセス可能なクエリを指定する。既定では、クエリはストアードプロシージャとして投影されない
Type	クエリがSQLまたはCache ObjectScriptコードを含むかどうかを指定する。既定では、クエリはSQLコードを含む

クラスクエリ定義の作成

クエリは独自のコンテンツと、その結果の各行に対して返されるフィールドの順序に関する情報を含む必要があります。

このような情報を提供するため、ユーザ記述のクエリには仕様と呼ばれるものが含まれます。

仕様は、クエリを修飾する 1～2 個のパラメータを含みます。そのパラメータは次のとおりです。

- **CONTAINID**：このオプションのパラメータは、いずれのフィールドが特定の行の ID を含むかを指定する
- **ROWSPEC**：このパラメータは、名前、データ型、ヘッダ、およびクエリの結果セットの各行のフィールド順序に関する情報を提供する

4-9 インデックス

インデックスは、インデックスが定義されるクラス的全範囲を自動的にカバーします。**Person** クラスがサブクラス **Student** を持つ場合、**Person** で定義されたすべてのインデックスは、**Person** オブジェクトと **Student** オブジェクトの両方を含みます。

Student クラスで定義されたインデックスは、**Student** オブジェクトだけを含みます。

インデックスは、それらのクラスに属する 1 つまたは複数のプロパティで並べ替えできます。

これは、返される結果の順序を制御するのに大変便利です。

インデックスキーワード

インデックスキーワードを使用して、インデックス定義を修正できます。

各キーワードはオプションで、明示的に指定されなければ既定値になります。

使用可能なインデックスキーワードは、表 4-7 のとおりです。

表 4-7: インデックスキーワード一覧

インデックスキーワード	概要
Data	インデックスに格納する追加のプロパティを指定する。既定では、追加のデータは含まれない
Description	インデックスの説明。Cacheはこの説明を使用しない。既定では、インデックスの説明はない
Extent	このインデックスが、このクラスのエクステントに属するすべてのオブジェクトを識別することを指定する。エクステントインデックスには、プロパティは含まれない
IDKey	既定ID形式の代わりに、OIDのID部分として使用するプロパティ、または一連のプロパティを指定する
PrimaryKey	このインデックスがSQLに対する主キーとして外部システムに報告されることを指定する。主キーとして定義されたいずれのインデックスも、各オブジェクトに対して一意の値を持つ必要がある。既定では、インデックスは主キーではない
Property	インデックスに含めたり並び替えたりするための、プロパティまたはプロパティのリストを指定する。インデックスは、最低でも1つのプロパティを含む必要がある
Type	標準インデックスまたはビットマップインデックスであるかどうかを指定する
Unique	各オブジェクトがインデックスでプロパティまたはプロパティの組み合わせに対して一意の値を持つことを指定する。既定では、インデックスは一意の値に制約されない。一意のキーワードはビットマップインデックスと一緒に使用することはできない

インデックス照合

Properties インデックスキーワードで指定された各プロパティは、オプションの照合タイプを持つこともできます。

照合タイプをプロパティに対して指定すると、データが並べ替えられ、インデックスに格納される前に各インスタンスのプロパティの値が、指定された方法で変換されます。

照合が指定されない場合は、プロパティの照合値が使用されます。

この値は、プロパティのデータ型から継承されます。

照合タイプを指定する場合、通常は、インデックスに対して照合タイプを指定するのではなくプロパティに対して照合タイプを指定する方法を推奨しています。

4-10 ObjectScript でのオブジェクトの使用

ObjectScript プログラム内でオブジェクトを使用するためには、次の内容を把握している必要があります。

メソッドの実行

オブジェクトの作成や変更を含む多くのアクションには、メソッドが必要です。

IRIS は、インスタンスメソッドとクラスメソッドの 2 種類のメソッドをサポートしています。

インスタンスメソッドは、そのインスタンスに関連する複数の動作を実行します。

クラスメソッドは、クラスのオブジェクトがメモリ内にあるかないかに関係なく呼び出されるメソッドです。

戻り値について

メソッドには、実行時に値を戻すものがありますが、値を戻さないメソッドもあります。メソッドが値を戻すのに、アプリケーションがその値を必要としない場合、値を戻さないメソッドの構文を使用できます。

戻り値の確認なしでメソッドを呼び出すには、**Do** コマンドを使用し、メソッドを呼び出して、確認を行う場合は **Set** コマンドを使用します。

インスタンスメソッドの実行

インスタンスメソッドを実行するには、最初にオブジェクトのインスタンスを作成するか、データベースからそのオブジェクトを開く必要があります。

クラスメソッドの実行

オブジェクトインスタンスなしで、クラスメソッドを実行できます。

また、同じクラスのメソッドから、クラスメソッドを実行することもできます。

その場合は、インスタンスメソッドと同じ形式で、メソッドを呼び出すことができます。

エラー条件

クラスやインスタンスメソッドを呼び出すとき、次の条件のときに、実行時エラーが発生することがあります。

- オブジェクト外部から、プライベートメソッドを呼び出そうと試みたとき
- クラスやオブジェクトインスタンスに対して定義されていないメソッドを呼び出したとき
- 存在しないクラスのメソッドを呼び出したとき

オブジェクトの新規作成

新規のオブジェクトを生成する一般的な構文では、クラスメソッドである%New を使用します。

```
Set oref = ##class(Classname).%New()
```

oref は、新規オブジェクトの OREF であり、Classname は作成するクラスの名前です。Classname は、大文字と小文字が区別され、パッケージ名が含まれる場合があります。

オブジェクトのオープン

次の構文を使用して、既存の永続オブジェクトを開くことができます。

```
Set oref = ##class(Classname).%OpenId(id)
```

oref は、開くオブジェクトの OREF であり、Classname はそのクラスです。

id は、開くオブジェクトに関連付けられているオブジェクト識別子です。

オブジェクトの変更

メソッドとともに、各オブジェクトは状態を定義するプロパティを持ちます。

多くのプロパティは、名前や生年月日などの値を保持しています。

また、オブジェクト間の関係を保持する、リレーションシップと呼ばれるプロパティもあります。

オブジェクトのプロパティの状態を変更する一般的な構文は、次のとおりです。

`Set oref.PropertyName = value`

`oref` は、特定のオブジェクトの **OREF** であり、`PropertyName` はデータと対応するプロパティ名で、`value` は実際のデータです。

参照プロパティの変更

参照プロパティに保存されたデータは、独立オブジェクトとして存在し、参照を含むオブジェクトとは別にインスタンス化できます。

2つのオブジェクト間のリンクを構築するには、**ObjectScript** の呼び出しを経由して、参照されたオブジェクトは他のオブジェクトの参照プロパティと関連付ける必要があります。

埋め込みオブジェクトプロパティの変更

埋め込みオブジェクトプロパティに保存されたデータは、メモリ内では個別のオブジェクトの一部として存在します。

しかし、個別の永続オブジェクト内に埋め込まれたデータとしてのみ保存が可能です。

埋め込みオブジェクトプロパティを変更するには、次の2つの方法があります。

- 埋め込みオブジェクトの新規インスタンスを生成し、そのインスタンスを埋め込みオブジェクトのプロパティと関連付ける
- カスケードドット構文を使用して、埋め込みオブジェクトプロパティを直接生成し、設定する

List プロパティの変更

リストは整列した情報のコレクションです。

各リスト要素は、リスト内のその位置によって識別されます。

スロットのデータ値を設定したり、スロットにデータを挿入したりできます。

スロットに新規の値を設定すると、その値はリスト内に保存されます。

既存のスロットに値を設定する場合、新規データは前のデータを上書きし、スロットの割り当てを変更しません。

既存のスロットにデータを挿入する場合、新しいリスト項目により、それ以降のすべてのスロットのスロット番号が増分されます。

Array プロパティの変更

Array とは、整列していないコレクション情報です。

1 つまたは複数の名前と値の組み合わせで構成されて、その要素の名前はキーとして提供されます。

値はそのキーと対応するデータです。

新規に要素を追加することと、既存の要素のデータを変更することには、構文的な違いはありません。

ストリームプロパティの変更

ストリームプロパティを変更するには、次の構文を使用します。

`Do oref.PropertyName.Write(data)`

`oref` は、ストリームを含むオブジェクトの OREF であり、`PropertyName` はストリームプロパティの名前、

`Write` は `Stream` クラスのメソッド、`data` はプロパティと対応する実際のデータです。

オブジェクトの保存

次の構文を使用して、すべての永続オブジェクトを保存できます。

`Do oref.%Save()`

`oref` は、保存するオブジェクトのオブジェクト参照です。

オブジェクトの削除

エクステント内のオブジェクトは、単独でも複数でも削除できます。

1 つのオブジェクトの削除

データベースからオブジェクトを削除するには、`%DeleteId` メソッドを使用します。
このメソッドは、削除中に発生したエラーの情報を保存するオプションを提供します。

エクステント中の全オブジェクトの削除

`%DeleteExtent` メソッドを使用して、クラスのすべてのインスタンスや、そのサブクラスのインスタンスを削除できます。
このメソッドは、削除中に発生したエラーの情報を保存するオプションを提供します。

クエリの実行

IRIS には、クエリを実行して結果を処理するための `%SQL.StatementResultSet` オブジェクトがあります。

クエリメタデータメソッド

クエリを使用するには、そのクエリを%SQL.StatementResultSet オブジェクトと関連させます。

その後、いくつかの方法で操作できるようになります。

クエリの実行の準備をする

事前に定義されたクエリの実行は、%SQL.StatementResultSet オブジェクトをインスタンス化することから始めます。

```
Set statement = ##class(%SQL.Statement).%New()  
Set status = statement.%PrepareClassQuery("Sample.Peson","ByName")  
Set rset = statement.%Execute()
```

rset は、新規の%SQL.StatementResultSet オブジェクトの OREF です。

クエリ実行

クエリを実行するには、次の構文を使用します。

```
Do rset.%Execute(arglist)
```

rset は、%SQL.StatementResultSet オブジェクトの OREF であり、arglist は、クエリに渡される引数のコンマ区切りリストです。

クエリ結果の処理

データの最初の行にアクセスするには、次の構文を使用します。

```
Do rset.%Next()
```

rset は、%SQL.StatementResultSet オブジェクトの OREF です。

クエリのクローズ

結果セットのすべての行を処理したら、次のようにクエリを閉じます。

```
Do rset.%Close()  
Set sc = rset.%Close()
```

コードの 2 行目の **sc** はローカル変数で、メソッドがエラー情報を含むステータスコードをここに返します。

4-11 データタイプ

IRIS は、変数として使用できる多くのデータタイプをサポートしています。

各データタイプは、それ自体がクラスです。

データタイプクラスは、`string` や `integer` などの特定のリテラルタイプを示し、関連するテーブルのフィールドとオブジェクトプロパティの両方の振る舞いを決定します。

IRIS では、ユーザ独自のデータタイプを作成することも可能です。

利用可能なタイプ

IRIS は、`%String`、`%Integer`、`%Float`、`%Date`、`%Time` など、頻繁に使用するデータタイプライブラリを提供します（表 4-8）。

表 4-8: Caché データタイプクラス

クラス名	保持する値	類似の SQL タイプ
<code>%Binary</code>	バイナリデータ	VARBINARY
<code>%Boolean</code>	ブーリアン値	N/A
<code>%Currency</code>	通貨値	MONEY、SMALLMONEY
<code>%Date</code>	日付	DATE
<code>%Double</code>	IEEE 浮動小数値	DOUBLE、DOUBLE PRECISION
<code>%Float</code>	浮動小数値	FLOAT、REAL
<code>%Integer</code>	整数	BIT、INT、INTEGER、SMALLINT、TINYINT
<code>%List</code>	\$List 形式のデータ	N/A
<code>%Name</code>	姓名形式の名前	N/A
<code>%Numeric</code>	可変精度数値	DEC、DECIMAL、NUMBER、NUMERIC
<code>%Status</code>	エラーステータスコード	N/A
<code>%String</code>	文字列	CHAR、CHAR VARYING、CHARACTER、CHARACTER VARYING、NATIONAL CHAR、NATIONAL CHAR VARYING、NATIONAL CHARACTER、NATIONAL CHARACTER VARYING、NATIONAL VARCHAR、NCHAR、NVARCHAR、VARCHAR、VARCHAR2
<code>%Time</code>	時刻値	TIME
<code>%TimeStamp</code>	日付と時間の値	TIMESTAMP

演算操作

データタイプの基本的な特徴を説明しましょう。

クラス内でデータタイプを使用する

データタイプクラスの主な機能は、クラス内でプロパティのタイプを指定することです。基本の定義形式は次のとおりです。

`Property City As %String;`

データタイプは%Library パッケージの一部なので、明示的にパッケージ名を示すことなく、それらを単純に呼び出すことができます。

パラメータ

データ型クラスは、さまざまなパラメータをサポートします。これらのパラメータは多様なアクションを実行し、データ型によって異なります。

キーワード

クライアントシステムとの相互運用性を提供するため、データタイプクラスには、「CLIENTDATATYPE」「ODBCTYPE」「SQLCATEGORY」という 3 つのキーワードが含まれています。

データ形式とその変換メソッド

データを処理する場合、IRIS は、状況に応じて複数の異なる形式を使用します。

データタイプは、これらの形式間で自動的にデータを変換します。

データタイプのサブクラスである独自のデータ型を作成する場合には、ユーザが使用しているデータタイプを使用しているプロパティは、さまざまな形式間での変換のメソッドを自動的に含んでいます。

列挙型プロパティ

プロパティは、列挙値としても知られる複数選択値をサポートします。

このプロパティを作成する、2つのデータタイプクラスパラメータは、**VALUELIST** と **DISPLAYLIST** です。

プロパティに有効な値のリストを指定するには、**VALUELIST** パラメータを使用します。

VALUELIST の形式は、区切り文字で区切った論理値のリストで、区切り文字が最初の文字になります。

DISPLAYLIST は、リストがある場合に、プロパティの **LogicalToDisplay** メソッドで返される対応する表示値を表す追加リストです。

4-12 オブジェクト永続性

IRIS には、オブジェクトの永続性が組み込まれており、自動的に提供されます。

永続性を持たせるためのコードの記述の必要も、オブジェクトマッピングやリレーショナルマッピングの必要もありません。

また、ミドルウェアやデータベース接続ツールに頭を悩ませる必要もありません。

%Persistent クラス

オブジェクト永続性は、永続インタフェースのメソッドを定義する **%Persistent** と、スキーマ進化と **SQL** プロジェクションを管理するクラスコンパイラによって提供されています。

永続化の主なスーパークラスは、**%Persistent** または他の永続クラスのいずれかである必要があります。

永続インタフェース

%Persistent クラスは、永続インタフェースとして知られる一連のメソッドを定義します。

これは、オブジェクト永続性と連携するための手段を提供します。

永続インタフェースは、オブジェクトをデータベースに保存する機能、データベースからオブジェクトをロードする機能、オブジェクトを削除する機能、および、存在を確認する機能を提供します。

オブジェクトの保存

永続オブジェクトをデータベースに保存するには、**%Save** メソッドを実行します。

%Save メソッドは、保存の操作が成功したか、または失敗したかを示す **%Status** 値を返します。

オブジェクトに対して **%Save** を呼び出すと、保存しようとしているオブジェクトからアクセスできるすべての変更済みオブジェクトが自動的に保存されます。

つまり、すべての埋め込みオブジェクト、コレクション、ストリーム、参照オブジェクト、および、オブジェクトに関与するリレーションシップが必要に応じて自動的に保存されます。

オブジェクトのオープン

オブジェクトは、%OpenId メソッドを使用して開くことができます。

%OpenId メソッドは、ID 値を入力として受け取り、参照をメモリ内のオブジェクトに返します。

オブジェクトが見つからない、または開くことができない場合は、NULL 値を返します。

オブジェクトの削除

永続インタフェースには、データベースからオブジェクトを削除する「%DeleteId メソッド」「%DeleteExtent メソッド」などという複数の方法があります。

オブジェクトの存在チェック

特定のオブジェクトインスタンスがデータベース内に保存されているかをテストする方法は 2 つあります。

最初の方法は、%Persistent クラスから提供されている%ExistsId メソッドを使用することです。

これは、ID 値を取るクラスメソッドで、指定されたオブジェクトがデータベース内に保存されている場合は真を、それ以外の場合は偽を返します。

もう 1 つの方法は、SQL 文を使用し、SQLCODE の値をテストする方法です。

オブジェクトエクステンツ

データベースに保存されている類似したタイプの一連のオブジェクトインスタンスは、オブジェクトエクステンツと呼ばれます。

単独の永続クラスでは、エクステンツはリレーショナルデータベースのテーブルの行の集まりに相当します。

エクステンクエリ

すべての永続クラスは、**Extent** と呼ばれるクラスクエリを自動的に含みます。

これは、オブジェクトエクステンメント内の、一連のすべてのオブジェクト ID 値を提供します。

ストレージ定義とストレージクラス

%Persistent クラスは、データベース内のオブジェクト保存と検索用の高レベルのインタフェースを提供します。

オブジェクトの保存とロードの実際の作業は、ストレージクラスによって実行されます。

すべての永続オブジェクトは、ストレージクラスを使用して、データベースのオブジェクトの保存、ロード、および、削除に使用される実際のメソッドを生成します。

これらの内部メソッドは、ストレージインタフェースと呼ばれます。

ストレージインタフェースには、**%LoadData**、**%SaveData**、および、**%DeleteData** などのメソッドが含まれます。

これらのメソッドはアプリケーションから直接呼ばれることはなく、永続インタフェースのメソッドによって適宜呼び出されます。

永続クラスによって使用されるストレージクラスは、ストレージ定義によって指定します。

ストレージ定義には、ストレージインタフェースによって使用される追加のパラメータや、ストレージクラスを定義する一連のキーワードや値が含まれます。

永続クラスには、1 つ以上のストレージ定義を含みますが、一度に 1 つしかアクティブになりません。

アクティブなストレージ定義は、クラスの **StorageStrategy** キーワードを使用して指定されます。

既定では、永続クラスは **Default** と呼ばれるストレージ定義を 1 つ持っています。

%CacheStorage クラス

%CacheStorage は、永続オブジェクトから使用される既定のストレージクラスです。

これは、永続クラスに対する既定のストレージ構造を自動的に作成し保持します。

新規の永続クラスを生成するときは、常に%CacheStorage クラスが自動的に使用されます。

%CacheStorage クラスでは、ストレージ定義のさまざまなキーワードを使用して、クラスに使用されるストレージ構造の特定の機能を制御できます。

%CacheSQLStorage クラス

%CacheSQLStorage は、オブジェクトの永続性を提供するために SQL の SELECT、INSERT、UPDATE、および DELETE 文を使用する、特別なストレージクラスです。

スキーマ進化

%CacheStorage クラスは、自動的なスキーマ進化をサポートします。

既定の%CacheStorage クラスを使用する永続クラスをコンパイルするときに、クラスコンパイラは、クラスによって定義されたプロパティを解析して、自動的にそのプロパティを追加します。

ストレージ定義の再設定

開発プロセス中は、永続クラスにプロパティの追加、変更、および削除などの変更をいくらかでも加えることができます。

しかし、変更のたびにクラスコンパイラは互換性構造を維持しようと試みるため、ストレージ定義はより複雑になっていきます。

クラスコンパイラで、すっきりしたストレージ構造が再生成されるようにするには、クラスのストレージ定義を削除して、そのクラスをリコンパイルします。

ただし、この操作を行うためには保存済のデータを全て削除して新たにデータを作成しなおす必要があります。

4-13 リレーションシップ

リレーションシップとは、2つの特定のタイプのオブジェクトの関連性です。

2つのオブジェクト間にリレーションシップを作成するには、それぞれがリレーションシップの片方を定義するリレーションシッププロパティを持っている必要があります。

リレーションシップの基礎

リレーションシップを使用する基本的な概要は、リレーションシップキーワードとリレーションシップの定義です。

リレーションシップキーワード

リレーションシップには、次の3つのキーワードがあります。

- **TYPE** : 関係のあるクラスのタイプ。これは、永続クラスである必要がある
- **INVERSE** : 関連するクラス内で対応するリレーションシッププロパティの名前
- **CARDINALITY** : リレーションシップの、こちら側のカーディナリティ

カーディナリティとは、独立したリレーションシップなのか依存したリレーションシップなのかを定義することで、ここでは、IRIS側からリレーションシップがどのように見えるかの定義を指します。

リレーションシップの定義

リレーションシップを作成するためには、補完的なリレーションシッププロパティの一组が必要です。

スタジオでは、新規プロパティウィザードで、「リレーションシップオプション」をチェックして、リレーションシッププロパティを定義できます。

リレーションシッププロパティを定義すると、スタジオは、関連するクラスの逆リレーションシッププロパティを自動的に作成します。

依存リレーションシップ

一方に **PARENT** を、もう一方に **CHILDREN** のカーディナリティを持つように定義することを依存リレーションシップと言います。

依存リレーションシップは、次のような特性を持っています。

- 子オブジェクトの存在は、親に依存する。親オブジェクトが削除された場合、子もすべて自動的に削除される

- 子オブジェクトは、一度特定の親オブジェクトに関連付けると、他の親オブジェクトと関連付けることはできない。

これは、子オブジェクトの永続 ID 値が、親の永続 ID に一部基づいているため

- 依存リレーションシップは、親子テーブルとして、SQL に投影される

リレーションシップのメモリ上の振る舞い

プログラミング的には、リレーションシップはプロパティとして振る舞います。

シングルバリューリレーションシップは、単一の参照プロパティのように動作します。

マルチバリューリレーションシップは、コレクションのようなインタフェースを持つ `RelationshipObject` クラスのインスタンスです。

リレーションシップの永続データの振る舞い

すべてのリレーションシップには、2つの側面がありますが、実際は片方だけがそのリレーションシップに関連した情報を、ディスク上に保存しています。

これはシングルバリュー側の、リレーションシップの **PARENT** もしくは **ONE** です。

リレーションシップのマルチバリューサイドのインスタンスを開くときは、**IRIS** は、そのリレーションシッププロパティを標準オブジェクト参照として処理します。

つまり、**PARENT** または **ONE** オブジェクトをメモリ内でスウィズル（遅延ロード）するのです。

参照整合性

リレーションシップは、制約を強制し、参照操作を実行することで参照整合性を維持します。

リレーションシップを保存する際に、**IRIS** は、その参照のターゲットの存在をチェックします。

ターゲットが存在しない場合、**IRIS** は、エラーを返してその保存操作は、失敗します。

依存リレーションシップの永続データの振る舞い

親子構造のリレーションシップで、親を削除すると子も削除されます。

具体的には、子は、その親の下位レベルに保存されています。

4-14 オブジェクト用 ObjectScript

ObjectScript には、オブジェクトメソッドに特別な機能を提供する多数の機能があります。

..**シンタックス**

..**シンタックス**は、現在のオブジェクトの他メソッドまたはプロパティを参照するメカニズムを提供します。

##class **シンタックス**

この**シンタックス**では、クラスの既存のインスタンスがないとき、または開かれているインスタンスがないときにクラスメソッドを呼び出します。

また、別のクラスからのメソッドとして、あるクラスからメソッドをキャストします。

クラスメソッドの起動

クラスメソッドを起動するには、次のどちらかの構文を使用します。

```
>Do ##class(Package.Class).Method(Args)
>Set localname = ##class(Package.Class).Method(Args)
```

メソッドのキャスト

あるクラスのメソッドを、別のクラスのメソッドとしてキャストするには、次のどちらかの構文を使用します。

```
>Do ##class(Package.Class1)Class2Instance.Method(Args)
>Set localname = ##class(Package.Class1)Class2Instance.Method(Args)
```

クラスメソッドとインスタンスメソッドの両方ともキャストできます。

\$this シンタックス

このシンタックスは、現在のインスタンスの **OREF** へのハンドルを提供します。

例えば、現在のインスタンスを別のクラスに渡したり、別のクラスが現在のインスタンスのメンバを参照したりするのに使用します。

##super シンタックス

サブクラスメソッド内から、オーバーライドされたスーパークラスメソッドを呼び出すには、**##super** シンタックスを使用します。

i%<PropertyName> シンタックス

永続プロパティを持つクラスをインスタンス化するとき、IRIS は、インスタンス変数と呼ばれるものを生成し、プロパティの値を保持します。

プロパティ値を設定または参照するときには、IRIS は、**Get** および **Set** アクセサメソッド を呼び出します。

これらは、最終的にインスタンス変数を参照します。アクセサメソッドには、**<PropertyName>Get** および **<PropertyName>Set** という形式の名前が付けられます。

なお、**<PropertyName>**は、アクセスされるプロパティ名です。

..#<Parameter> シンタックス

このシンタックスを使用すると、クラスの方法内からクラスパラメータへの参照を行うことができます。

4-15 XData ブロック

XData ブロックとは、クラス定義に追加できる XML コードのブロックです。

XData ブロックをクラス定義に追加するには、次の 2 つの方法があります。

- クラスエディタを使用して、クラス定義を編集する方法
- 新規 XData ウィザードを使用する方法

クラスエディタを使用して XData ブロックを追加するには、クラスエディタの空の行にカーソルを置き、XData 宣言を入力します。

4-16 クラス定義クラス

IRIS ライブラリには、IRIS 統一ディクショナリへのオブジェクトアクセスを提供するための、一連のクラス定義クラスが含まれています。

これらのクラスを使用して、クラス定義の調査、クラス定義の修正、新規クラスの生成を行うプログラムを作成できます。

またはドキュメントを自動生成するプログラムを記述することも可能です。

これらのクラスは、%Dictionary パッケージに含まれます。

クラス定義の閲覧

クラス定義クラスによって、データベースアプリケーションで使用するのと同じ手法を使用して、IRIS ディクショナリ内のクラス定義を閲覧可能です。

%SQL.StatementResultSet オブジェクトを使用すると、一連のクラスの処理を繰り返し実行できます。

また、特定のクラス定義を表す永続オブジェクトもインスタンス化できます。

クラス定義の変更

%Dictionary.ClassDefinition オブジェクトを開き、任意の変更を行ったあと、%Save メソッドにより保存すれば、既存のクラス定義を修正できます。

4-17 オブジェクト同時実行制御

IRIS では、同一オブジェクトに対する、追加、変更、削除などの処理を並行して行うことができます。

同時実行制御オプション

多くの%Persistent クラスのメソッドにより、ロックがどのように同時処理コントロールに使用されるかを決定するために、オプションの同時処理設定を指定できます。

そして、アプリケーションに対して、適切なレベルの同時処理オプションを設定することが重要です。

concurrency に設定する同時処理オプションは、次のとおりです。

- ロックしない：ロックを使用しない

- アトミックリード：複数のノードにデータが保持されている場合のみ、オブジェクトがロードされるときに共有ロックを行い、読み終わるとそのロックを開放する。

新しいオブジェクトの場合にはロックを取得しない。

保存の際には排他ロックを取得する。

- 共有：オブジェクトがロードされるときに共有ロックを行い、読み終わるとそのロックを解放する。

新しいオブジェクトの場合にはロックを取得しない。

保存の際には排他ロックを取得する。

- 共有／保持：そのオブジェクトに対して共有ロックを取得する。

メモリー上でそのオブジェクトを解放するとそのロックも解放する

保存の際には新しいオブジェクトを含めて排他ロックを取得する。

このロックは、オブジェクトをメモリー上で解放するまで保持する。

- 排他：既存のオブジェクトを開くとき、または新しいオブジェクトをデータベースに初めて保存するときに排他ロックを取得する

メモリー上でオブジェクトを解放するとそのロックも解放する

バージョンチェック

IRIS は、VERSIONPROPERTY と呼ばれるクラスパラメータを使用して、データのバージョン確認が可能です。

すべての永続クラスに、このクラスパラメータが存在します。

4-18 IRIS オブジェクトを使ったサンプルプログラミング

第 3 章で紹介したサンプルプログラムを、IRIS オブジェクトを使った形に修正します。
このコードはセットアップした環境に既に存在しています。

サンプルプログラム Sales.Operation クラス

```
Include %occStatus

Class Sales.Operation
{

ClassMethod populate(no As %Integer) As %Boolean
{
    do ##class(Sales.Customer).%KillExtent()
    do ##class(Sales.Product).%KillExtent()

    do ##class(Sales.Customer).Populate(no)
    do ##class(Sales.Product).Populate(no)

    quit $$$OK
}

ClassMethod placeOrder(customerId As %Integer) As %Boolean
{

    if customerId="" write "顧客 ID を入力してください",! quit '$$$OK

    if '##class(Sales.Customer).%ExistsId(customerId) write "顧客 ID が登録されて  
いません",! quit '$$$OK

    set customer = ##class(Sales.Customer).%OpenId(customerId)
```

```
write customer.Name_"様、いつもご利用ありがとうございます",!!

set result=##class(%SQL.Statement).%ExecDirect("select id,name,unitprice from
Sales.Product")
set resultset = result.%CurrentResult

write "商品 ID",?20,      "商品名",?40,"単価",!
write "-----",!

While resultset.%Next(.sc) {
    If $$$ISERR(sc) quit
    write
resultset.id,?20,resultset.name,?40,##class(Sales.Product).UnitPriceLogicalToDisplay(resultset.unitprice),!
}

write !

set itemNo = 0

for {
    read "注文したい商品 ID を入力してください  ", pid,!

    if pid="" quit

    if '##class(Sales.Product).%ExistsId(pid) write "商品が存在しません",!
continue

    read "数量を入力してください  ", amount,!
    if amount?.N write "正しい数値を入力してください",! continue
    set itemNo = itemNo + 1
    set items(itemNo) = $listbuild(pid,amount)
}

if itemNo = 0 quit '$$$OK
```

```

set order = ##class(Sales.PurchaseOrder).%New()
set order.Customer = customer
  set order.PurchaseDate = $piece($horolog,"",1)

  for i = 1:1:itemNo {
    set item = ##class(Sales.OrderItem).%New()
    set pid = $list(items(i),1)
    set product = ##class(Sales.Product).%OpenId(pid)
    set item.Product = product
    set item.Amount = $list(items(i),2)
    do order.Items.Insert(item)
  }

  set sc = order.%Save()

  if $$$ISERR(sc) {
    Do $system.Status.DisplayError(sc)
  }
  Else {
    write "注文を受け付けました",!
  }
  quit $$$OK
}

ClassMethod orderByCustomer(customerId As %String) As %Boolean
{

  if customerId="" write "顧客 ID を入力してください",! quit '$$$OK

  if ##class(Sales.Customer).%ExistsId(customerId) write "顧客 ID が登録されて
  いません" quit '$$$OK

  set customer = ##class(Sales.Customer).%OpenId(customerId)

```

```

write !!,customer.Name_"様のご注文履歴は、以下のようにになっています",!!

set result=##class(%SQL.Statement).%ExecDirect("select id,purchasedate from
Sales.PurchaseOrder where customer->id = ?",.customerId)
set resultset = result.%CurrentResult

write "注文 ID",?20,"注文日",!
write "-----",!

While resultset.%Next(.sc) {
    If $$$ISERR(sc) quit
    write
resultset.id,?20,##class(%Date).LogicalToDisplay(resultset.purchasedate),!
}

write !

read "内容を表示したい注文番号を入力してください",oid,!!

if oid = "" quit '$$$OK

if '##class(Sales.PurchaseOrder).%ExistsId(oid) write "注文が存在しません",!
quit '$$$OK

set porder = ##class(Sales.PurchaseOrder).%OpenId(oid)

write "ご注文内容は、以下のとおりです",!!

write "お客様名",?20,"注文日",!
write
porder.Customer.Name,?20,porder.PurchaseDateLogicalToOdbc(porder.PurchaseDa
te),!!

```



```

    write "明細番号",?10,"商品 ID",?20,"商品名",?40,"単価",?50,"割引率",?60,"数量",?70,"小計",!

    write "-----",!

    for i=1:1:porder.Items.Count() {
        set item = porder.Items.GetAt(i)
        write
        $justify(i,8),?10,$justify(item.Product.%Id(),6),?20,item.Product.Name,?40,$justify(item.Product.UnitPriceLogicalToDisplay(item.Product.UnitPrice),6),?50,$justify(porder.Customer.Discount,4)," %",?60,$justify(item.Amount,4),?70,$justify(item.SubTotalLogicalToDisplay(item.SubTotal),8),!
    }
    quit $$$OK
}

ClassMethod totalByProduct(productId As %Integer) As %Boolean
{
    if productId="" write "商品 ID を入力してください",! quit '$$$OK

    if '##class(Sales.Product).%ExistsId(productId) write "商品 ID が登録されていません" quit '$$$OK

    set total = 0

    set result=##class(%SQL.Statement).%ExecDirect("select id from Sales.OrderItem where product->id = ?",.productId)
    set resultset = result.%CurrentResult

    While resultset.%Next(.sc) {

        If $$$ISERR(sc) quit

        set id = resultset.id
        set item = ##class(Sales.OrderItem).%OpenId(id)

        set total = total + item.SubTotal
    }
}

```

```
}  
  
write "この商品の売り上げ累計は、",total,"円です",!  
  
quit $$$OK  
}  
  
}
```

その後、このクラスをコンパイルして、ターミナルでログインし、データを生成します。

```
USER>DO ##class(Sales.Operation).populate(5)
```

それでは実際に注文作成して、注文履歴を確認してみましょう。注文作成は次のようにします。

```
USER>DO ##class(Sales.Operation).placeOrder(3)
```

北村 ゆきえ様、いつもご利用ありがとうございます

```
商品 ID 商品名 単価
-----
1 ヘアジェル 1400
2 スクラブウォッシュ 9200
3 シャンプー 4000
4 ボディーソープ 4600
5 ボディーソープ 600
注文したい商品 ID を入力してください 2
数量を入力してください 3
注文したい商品 ID を入力してください 4
数量を入力してください 2
注文したい商品 ID を入力してください

USER>
```

続いて、注文履歴を確認してみましょう。顧客別注文状況を表示させるには、次のようにします。

```
USER>DO ##class(Sales.Operation).orderByCustomer(3)
```

北村 ゆきえ様のご注文履歴は、以下のようになっています

注文 ID 注文日

1 09/10/2008

内容を表示したい注文番号を入力してください 1

ご注文内容は、以下のとおりです

お客様名 注文日

北村 ゆきえ 2008-09-10

明細番号 商品 ID 商品名 単価 割引率 数量 小計

1 4 ボディーソープ 4,600 90 % 2 8,280

2 2 スクラブウォッシュ 9,200 90 % 3 24,840

特定の商品の売り上げ累計を表示させるには、次のようにします。

```
USER>DO ##class(Sales.Operation).totalByProduct(1)
```

この商品の売り上げ累計は、3360 円です

第 5 章 IRIS SQL

IRIS SQL は、IRIS の統一データアーキテクチャ内の、IRIS クラス辞書でモデリングされたクラス階層を、IRIS オブジェクトとは異なる方法で表現しています。

オブジェクト指向の観点からすると、クラスのデータコンポーネントはリレーショナルテーブルになり、その行と列は個々のインスタンスとオブジェクトのプロパティにそれぞれ対応します。

しかし、クラス階層の全属性を、リレーショナルの観点から表現できるわけではありません。

その理由の 1 つは、オブジェクトはプロパティとメソッド（つまり、データと振る舞い）を持つ一方で、テーブルはデータだけを持つからです。

本章では、その IRIS SQL の特徴を説明します。

5-1 テーブルに対するクラスの投影

IRIS でテーブルを定義する 1 つの方法は、まず、スタジオ（Visual Studio Code）を使用して永続クラス定義を生成し、その定義ファイルを保存した後にコンパイルすることです。

この手順を踏むと、そのクラス定義に対応するリレーショナルテーブル定義が自動的に生成されます（各クラスはテーブルを表し、各プロパティは列を表します）。

例えば、次のようにスタジオで **Person** クラスを定義します。

```
Class MyApp.Person Extends %Persistent [ ClassType = persistent]
{
    Property Name As %String(MAXLEN=50) [Required];
    Property SSN As %String(MAXLEN=15) [Required];
    Property DateOfBirth As %Date;
    Property Sex As %String(MAXLEN=1);
}
```

これで、コンパイル時に、リレーショナルテーブル「Person」定義が MyApp スキーマ内に生成されます。

これは、次のように DDL によって定義した場合と同じです。

```
CREATE TABLE MyApp.Person (
  Name VARCHAR(50) not null,
  SSN VARCHAR(15) not null,
  DateOfBirth date,
  Sex varchar(1)
)
```

すべてのパッケージは、SQL スキーマに対応しています。

例えば、クラスが Team.Player (Team パッケージの Player クラス) である場合、対応するテーブルは Team.Player (Team スキーマの Player テーブル) です。

既定では、パッケージ名を省略した場合には、User パッケージが使われ、スキーマは、SQLUser になります (User は SQL の予約語のため使用できないため)。

したがって、「User.Person」という名前のクラスは「SQLUser.Person」という名前のテーブルに対応します。

パッケージ名に「(. ピリオド)」が含まれる場合は、ピリオドの代わりに「_ (アンダースコア)」を使用します。

例えば、次のようになります。

```
MyTest.Test.MyClass クラス (「MyTest.Test」パッケージの「MyClass」クラス)
↓
MyTest_Test.MyClass テーブル (「MyTest_Test」スキーマの「MyClass」テーブル)
```

基本的に、永続クラスは一意の SQL テーブルに、各インスタンスは、そのテーブル内で
行として投影されます。

その他の項目は、表 5-1 のように投影されます。

表 5-1 : オブジェクト SQL プロジェクション

投影元 (オブジェクトコンセプト)	投影先 (リレーショナルコンセプト)
パッケージ	スキーマ
クラス名	テーブル名
オブジェクト ID	ID フィールド
データ型プロパティ	フィールド
参照プロパティ	参照フィールド
埋め込みオブジェクト	一連のフィールド
リストプロパティ	リストフィールド
配列プロパティ	子テーブル
ストリームプロパティ	BLOB
インデックス	インデックス
メソッド	ストアードプロシージャ(クラスメソッドのみ)

それでは、以降で SQL プロジェクションについて説明します。

テーブル名

クラス名そのものに制約はないのですが、投影先のテーブル名として SQL 予約語は使えないため、そうした名前を持つ永続クラスを生成しようとする、IRIS クラスコンパイラはエラーメッセージを返します。

その場合は、クラス名を変更するか、クラス名とは異なるテーブル名をプロジェクションに対して指定する必要があります。

その指定方法は、次のように **SQLTABLENAME** キーワードを使用します。

SQLTABLENAME = TableNameForSQL;

オブジェクト ID と IDKEY

各オブジェクトは、オブジェクト ID (オブジェクト識別子) によって一意に識別されます。

対応するテーブルの各インスタンスは、テーブル定義で作成された ID 列の各エントリ値によって一意に識別されます。

ID 列は、オブジェクトに「ID」と名の付くプロパティがない限り、「ID」という名前を持っています。

オブジェクトが「ID」と名の付くプロパティを持っている場合は、ID 列は名称として「ID1」を持ちます。

ID は列として表示されますが、テーブル用の IDKEY になっているため、インデックスを作成する必要はありません。

継承

継承はリレーショナルモデルには存在しない概念のため、クラスコンパイラは、クラスを平坦化したものをリレーショナルテーブルとして投影します。

投影されたテーブルには、継承されたフィールドを含む、そのクラスに適切なすべてのフィールドが含まれます。

そのため、サブクラスのプロジェクションは、次のような項目で構成されるテーブルとなっています。

- スーパークラスのプロジェクションのすべての列
- サブクラスだけにあるプロパティに基づいたその他の列
- サブクラスのインスタンスだけで構成される、スーパークラスのテーブル内にある行のサブセット

定数プロパティ

プロパティが投影されると、多くの場合、リレーショナルテーブル内に列（フィールド）として表れ、その列はプロパティから名前を取得します。

プロパティの投影された列に異なる名前を付ける場合は、「SQLFIELDNAME」キーワードを使用します。

データ型

データ型は、そのプロパティの **SQL** カテゴリとそのパラメータを使用したフィールドとして投影されます。

この **SQL** カテゴリは、「**SQLCATEGORY**」キーワードで定義します。

リレーショナルアクセスとオブジェクトアクセスの両方とも、同じデータ型クラスを使用して、データの妥当性検証と形式変換のために同じデータ型メソッドを呼び出し、同じ方法でデータ型パラメータの値を使用します。

これは、データ型クラスがシステムで提供されたものでも、ユーザが記述したものでも変わりません。

計算プロパティ

計算プロパティ（**Calculated** プロパティ）は、ディスク上に保存されないため、設定しない限り、とくに **SQL** には投影されません。

この列をテーブルに投影するには、「**SQLCOMPUTED**」キーワードを使用します。

このキーワードによって、プロパティに対して **SQL** 計算フィールドの値を計算するコードを指定します。

プロパティの **SQL** 計算フィールドのコードを組み込むには、計算を行うコード「**SQLCOMPUTECODE**」キーワードとを関連付けます。

計算を行うコードには、他のプロパティに基づいて算出された値やシステムから導出された値を含むことができます。

このコードで、クラスメソッドやユーザ定義の関数も呼び出すこともできます。

参照プロパティ

参照プロパティは、永続オブジェクトへの参照を行うプロパティです。

これは、参照されるオブジェクト **ID** の **ID** 部分が入るフィールドとして投影されます。例えば、顧客オブジェクトが、**SalesRep** オブジェクト（顧客担当の営業員）を参照する **Rep** プロパティを持っているとします。

ある顧客の担当が「**ID 12**」の営業員である場合は、その顧客の **Rep** 列も「**12**」になります。

この値は、参照されているオブジェクトの ID 列の特定行の値と一致するため、JOIN あるいはその他の処理を行うために使用できます。

埋め込みオブジェクトプロパティ

埋め込みオブジェクトプロパティは、親クラスのテーブルで、複数列として投影されます。

プロジェクション内の 1 列に、区切り文字と制御文字をすべて含む連続した形式でオブジェクト全体が格納されます。

残りの各列は、オブジェクトの各プロパティに対応します。

リレーションシップ

リレーションシップとは、2 つのオブジェクトの関連性を表す特別なプロパティです。

リレーションシップの SQL プロジェクションは、そのカーディナリティ (Cardinality) 値によって異なります。

カーディナリティ値は、それが独立 (一対多) リレーションシップか依存 (親子) リレーションシップかを定義するのと同様に、こちら側からリレーションシップがどのように「見える」のかを定義します (例: 1 対多、親対子など)。

単一値側 (1 や親) は、単体指定参照フィールドとして投影されます。

リストプロパティ

リストプロパティは、\$LIST 形式の文字列として投影されます。

これは、コレクション内のすべての要素が、単独の文字列として連結され投影されるということを意味します。

例えば、Person クラスが、自動車のナンバープレートの番号を表す文字列のリスト「Cars リストプロパティ」を持っていたとします。

このプロパティは、Cars という単一の列として表されます。

配列プロパティ

配列プロパティは、子テーブルとして投影されます。

この子テーブルの名前は、その配列プロパティを含むクラスの名前と、配列プロパティ名自体をつなげたものになります。

子テーブルには、次のような 3 列が含まれます。

- 親クラスの各インスタンスの ID を含む列（列名は、配列を含むクラスの名前となる）。この列の値は、子に対しては一意ではないが、親に対しては一意である値を参照することに注意
- 各配列メンバの識別子を含む列（列名は、常に `element_key` となる）
- クラスの全インスタンスの配列メンバを含む列（列名は、配列プロパティの名前となる）

インデックス

IRIS SQL では、テーブルの列の値のインデックスを定義することができます。

これらのインデックスは、データベースに対して、オブジェクトの更新アクセスあるいは SQL アクセスの「INSERT」「UPDATE」「DELETE」演算が実行された場合にも自動的に維持されます。

各インデックスは、そのクラスまたはテーブル内で一意の名前を持ちます。

この名前は、レポート、インデックス構築、インデックス削除などのデータベース管理に使用します。

IRIS SQL は、特殊なケースでも使用可能な、さまざまなインデックスタイプやオプションをサポートしています。

標準インデックス

標準インデックスは、永続多次元配列でインデックス値と値を含む行の **RowId** を関連付けます。

明示的に、下記で説明するビットマップインデックス、ビットスライスインデックスと定義されていないインデックスは、すべて標準インデックスです。

- 複数プロパティのインデックス：2 つ以上のプロパティの組み合わせに対するインデックス
- ユニークインデックス：クラスに対してユニーク制約を与えるために使用する標準インデックス

ビットマップインデックス

ビットマップインデックスは、特別なタイプのインデックスです。

このインデックスには、既存のインデックス値に一致する、一連のオブジェクト ID 値を表す一連のビット文字列を使用します。

IRIS のビットマップインデックスには、次のような重要な特徴があります。

- 高度に圧縮されていて、通常のインデックスに比べて一般的に容量が非常に小さいため、ディスクおよびキャッシュの使用量を大幅に削減できる
- ビットマップ演算はトランザクション処理に最適なため、標準インデックスに比べてパフォーマンスが低下しない
- ビットマップ論理演算によって、クエリパフォーマンスが向上する
- IRIS SQL エンジンには、ビットマップインデックスのための特別な最適化機能を備えている

ビットスライスインデックス

ビットスライスインデックスは、計算に使われる数値データにのみ使用します。

ビットスライスインデックスは、各数値データを 2 進のビット文字列として表現します。

高速な集計処理に使うことに特化した特別なインデックスです。

クエリ

クエリとは、1つ以上の **SELECT** 文を使ってデータを取得する文のことです。

UNION 節によって複数の **SELECT** 文を組み合わせることも可能です。

また、クエリは、上位レベルのクエリの後続く単独の **ORDER BY** 節によって、クエリが検索するデータを再構成することができます。

IRIS SQL 内では、クエリによって、テーブルのデータを見たり変更したりできます。大別すると、クエリには、データの検索 (**SELECT** 文) とデータの変更 (**INSERT** 文、**UPDATE** 文、**DELETE** 文) の 2 種類があります。

SQL クエリは、次のようなさまざまな方法で使用できます。

- ObjectScript で埋め込み SQL を使う
- ObjectScript で、ダイナミック SQL を使う
- 他のさまざまな環境から ODBC インターフェイスまたは JDBC インターフェイスを使う

なお、クエリは、IRIS オブジェクトもしくは ObjectScript ルーチンの一部です。

クラスメソッド

IRIS SQL では、ストアドプロシージャをクラスメソッドとして定義できます。

実際のところ、ストアドプロシージャは、SQL でも使用できるようにしたクラスメソッドに過ぎません。

ストアドプロシージャでは、IRIS オブジェクトの全機能が使えます。

IRIS SQL には、次のような 2 種類のストアドプロシージャタイプがあります。

- 一連のレコードを返すもの（結果セットストアドプロシージャと呼ばれる）
- レコードを返さないもの（メソッドストアドプロシージャと呼ばれる）

このうち、値を返すメソッドストアドプロシージャは、「ストアド関数」とも呼ばれています。

5-2 標準 SQL (ANSI 92) サポートの例外

IRIS SQL は、次のいくつかの例外を除いて、すべてのエントリーレベル SQL-92 標準をサポートしています。

- テーブル定義に CHECK 制限を別途追加することはできない
- SERIALIZABLE 分離レベルはサポートされていない
- 算術演算子の優先順位が SQL-92 標準とは異なる

IRIS SQL には演算子の優先順位はなく、算術式は必ず左から右の順番で解析されます。

この規則は、ObjectScript でも同様です。

そのため、「 $3+3\times 5=30$ 」という式は、カッコを使って「 $3+(3\times 5)=18$ 」としなければなりません。

- 区切り識別子の小文字と大文字を区別しない

SQL-92 標準では、区切り識別子の小文字と大文字を区別しなければなりません。

- HAVING 節に含まれるサブクエリで集合参照できない

SQL-92 標準では、HAVING 節に含まれるサブクエリで、その HAVING 節で利用可能な集合を参照できますが、この機能はサポートされていません。

DQL 文

DQL 文は、データベースの内容に対して問い合わせを実行する文です。

これには、SELECT 文を使用します。

SELECT 文は、1 つ以上のテーブルまたはビューから、1 行以上のデータを選択します。この値が、クエリの結果セットに含まれます。

SELECT Name FROM Sample.Person

SELECT 文内で複数の列を表すには、コンマで区切ります。

`SELECT Name,SSN FROM Sample.Person`

また、SELECT 文の列にエイリアスを与えたり、式やリテラル値を含んだりすることもできます。

その他にも、標準 SQL の次の節も利用可能です。

- FROM 節：クエリに対してデータを提供する 1 つ以上のテーブルを指定する
- WHERE 節：クエリによって返される値を制限する 1 つ以上の条件を指定する
- ORDER BY 節：1 列以上の列を指定して、クエリによって返される値を並べ替える

IRIS SQL では、2 つのテーブルを組み合わせ、制限条件にあてはまる 3 番目のテーブルを作成する JOIN 句も使用できます。

DML 文

SQL クエリを使用して、データベースの内容を変更できます。

テーブルにインデックスが定義されている場合は、SQL はインデックスも自動的に更新します。

データや参照整合性の制約が定義されている場合は、SQL は自動的にそれらを実行します。

IRIS SQL においても、例外ではありません。

- INSERT 文：SQL テーブルに新規の行を挿入する
- UPDATE 文：SQL テーブルの既存の行の値を変更する
- DELETE 文：SQL テーブルの既存の行を削除する

DDL 文

IRIS SQL では、「ALTER」「CREATE」「DROP」といった標準 DDL 文によるテーブル定義が可能です。

この DDL 文は、次のようなさまざまな方法で実行することができます。

- ODBC 呼び出し
- JDBC 呼び出し
- DDL スクリプトファイル
- メソッドまたはルーチンでの埋め込み SQL

これらの実行方法は、以降、順を追って説明します。

5-3 IRIS による SQL 拡張

IRIS SQL は、多数の便利な拡張機能をサポートしています。

これらは、データに対してオブジェクトアクセスとリレーショナルアクセスを同時に提供できるという「統一データアーキテクチャ」に基づいています。

標準に追加された IRIS SQL 演算子

%ID

クエリの現在行のオブジェクト ID 値を返す擬似列名です。

次のコードはその使用例です。

```
SELECT %ID FROM Sample.Person ORDER BY %ID
```

この擬似列名によって、プログラム側では実際の列名がわからない場合でも、任意の行のオブジェクト ID を見つけることができます。

%STARTSWITH

文字列の値が、指定された文字で始まるかをテストする演算子です。

次のコードはその使用例です。

```
SELECT %ID, Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
```

%STARTSWITH 演算子は、SQL の LIKE 演算子と似ていますが、機能が限定されている（前方一致検索のみ）分、インデックスが有効利用できるのもので、一般的には %STARTSWITH の方が効率的です。

ユーザ定義関数

IRIS SQL では、SQL クエリでクラスメソッドを実行することができます。

これによって、SQL 構文をユーザが自由に拡張できるようになっています。

そのための手順は、次のとおりです。

① 永続 IRIS クラスで、リテラル（非オブジェクト）の返り値を持つクラスメソッドを定義する SQL クエリでは、インスタンスメソッドを実行するためのオブジェクトインスタンスがないため、クラスメソッドでなければなりません。

また、SQL ストアドプロシージャとして定義される必要があります。例えば、次のようにして、MyApp.Person クラスで Cube メソッドを定義します。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent, language = cache]

{

/// 数字の 3 乗をもとめる
ClassMethod Cube(val As %Integer) As %Integer [SqlProc]
{
Quit val * val * val
}
}
```

② SQL クエリで、メソッドを組み込み SQL 関数であるかのように呼び出す

そのためには、次のように実行します。

```
SELECT %ID, Age, MyApp.Person_Cube(Age) FROM MyApp.Person
```

このクエリは、Age のそれぞれの値に対して Cube メソッドを呼び出し、返り値を結果に表示します。

暗黙結合

IRIS SQL は、関連するテーブルから値を取得するための JOIN に代わる省略手段として、特別な->演算子を提供しています。

これにより、特定のケースで明示的に JOIN を指定するという複雑な作業を省くことができます。

参照されたテーブルから値を取得する例を 1 つ挙げてみましょう。

例えば、Company と Employee という 2 つのクラスを定義するとします。

【Company クラス】

```
Class Sample.Company Extends %Persistent [ClassType = persistent ]
{
  /// 会社名
  Property Name As %String;
}
```

【Employee クラス】

```
Class Sample.Employee Extends %Persistent [ClassType = persistent ]
{
  /// 従業員名
  Property Name As %String;
  /// この従業員が働く会社
  Property Company As Company;
}
```

Employee クラスは、Company オブジェクトへの参照プロパティを含んでいるため、オブジェクトアクセスを使ったアプリケーションでは、ドット構文を使用してこの参照を表すことができます。

例えば、会社員（employee）が所属する会社（company）を見つけるには、次のようにします。

```
name = employee.Company.Name
```

SQL 文で同じ作業を実行するには、OUTER JOIN 句によって、次のように Employee と Company テーブルを結合する必要があります。

```
SELECT Sample.Employee.Name AS EmpName, Sample.Company.Name AS  
CompName  
FROM Sample.Employee LEFT OUTER JOIN Sample.Company  
ON Sample.Employee.Company = Sample.Company.ID
```

しかし、-> 演算子を使えば、次のように、より簡潔に記述することができます。

```
SELECT Name AS EmpName, Company->Name AS CompName  
FROM Sample.Employee
```

テーブルに参照列がある場合は、いつでも-> 演算子を使用できます。

この列の値は、参照されたテーブルの ID です。クエリ内で列の式を使用できる場所であれば、どこでも-> 演算子を使用できます。

例えば、WHERE 句を使った次のような SQL 文は、

```
SELECT Sample.Employee.Name  
FROM Sample.Employee, Sample.Company  
WHERE Sample.Employee.Company = Sample.Company.ID AND  
Sample.Company.Name = 'XYZ Corp'
```

->演算子を使うと、

```
SELECT Name  
FROM Sample.Employee  
WHERE Company->Name = 'XYZ Corp'
```

のように、簡潔になります。

また、ORDER BY 節でも、次のように、-> 演算子を使用することができます。

```
SELECT Name AS EmpName, comp.Name AS CompName  
FROM Sample.Employee  
ORDER BY Company->Name
```

依存リレーションシップ

依存リレーションシップとは、一方に PARENT（親）を、もう一方に CHILDREN（子）のカーディナリティを持つように定義したものです。

次のような特徴を持っています。

- 子オブジェクトの存在は親に依存する（親オブジェクトが削除された場合、子もすべて自動的に削除される）
- 子オブジェクトは、特定の親オブジェクトに関連付けられると、他の親オブジェクトと関連付けることはできない（子オブジェクトの永続 ID 値が、親の永続 ID に一部基づいているため）
- ディスクアクセスの最適化のために、子オブジェクトのインスタンスがディスク上では親オブジェクトと一体となって保存されている
- 2つのクラスはコンパイル時にリンクされる（そのため、依存リレーションシップの両側は同一クラス、同一の基本クラスおよびその派生クラスには定義できない）

この依存リレーションシップは、IRIS SQL に親子テーブルとして投影されます。

5-4 埋め込み SQL と動的 SQL

IRIS SQL は、ObjectScript コードに SQL 文を埋め込む、「埋め込み SQL」機能をサポートしています。

埋め込み SQL 文は、以前は、コンパイル時に、最適化された実行可能なコードに変換されていましたが、今ではコンパイル時ではなく、実行時に作成、実行される動的 SQL と同等の動きとなっています。

これにより、%SQL.Statement クラスを使用して、IRIS でも ODBC や JDBC アプリケーションと同様の方法でプログラムを作成することができます。

非カーソルベース SQL

埋め込み SQL の 1 つである、非カーソルベース SQL とは、「INSERT」「UPDATE」「DELETE」および DDL 文、GRANT 文、REVOKE 文、SELECT 文といった単純な SQL 文のことです。

次の埋め込み SQL 文は、ID 番号が 43 の Patient の名前のみを検索する例です。

```
&sql(SELECT Name INTO :name  
FROM Patient  
WHERE %ID = 43)
```

ただし、SELECT 文で非カーソルベース SQL として使えるのは、1 行だけ返すクエリだけです。

次のように複数行を返す SELECT 文を非カーソルベース SQL として記述しても、最初の 1 行だけが返されます。

```
&sql(SELECT Name INTO :name  
FROM Patient  
WHERE Age = 43)
```

しかし、クエリによっては、実際にどの行が最初に返されるかの保証はありません。

カーソルベース SQL

埋め込み SQL で、複数の行を返すクエリを実行したい場合は、カーソルベース SQL を使います。

カーソルベース SQL を使うには、まず **DECLARE** で名前を設定する必要があります。この名前は、カーソルの「**OPEN**」「**FETCH**（データの取り出し）」「**CLOSE**」の際に使用します。

カーソル名は、クラスまたはルーチンの中で一意である必要があります、**DECLARE** 文は、そのカーソルを使用する文よりも前にある必要があります。

次のコードは、カーソルを使用してクエリを実行し、主デバイスへ結果を表示する例です。

```
&sql(DECLARE C1 CURSOR FOR
SELECT %ID,Name
INTO :id, :name
FROM Sample.Person
ORDER BY Name
)
&sql(OPEN C1)
&sql(FETCH C1)

While (SQLCODE = 0) {
Write id, ": ", name,!
&sql(FETCH C1)
}

&sql(CLOSE C1)
```


この例では、次の手順を踏んでいます。

① **DECLARE** 文によって、カーソル **C1** を宣言し、**Name** の順に並べられた一連の **Person** 行を返す

DECLARE 文では、名前とカーソルを定義する **SQL SELECT** 文の両方を指定します。

② **OPEN** 文によって、カーソルをオープンする

OPEN 文では、カーソルより後の部分を実行するためにカーソルそのものを作成します。

OPEN 呼び出しに成功すると、**SQLCODE** 変数が **0** に設定されます（オープンの成功を意味します）。

最初に **OPEN** 呼び出しを実行せずに、カーソルからデータを **FETCH** することはできません。

③ データの最後に達するまで、カーソルの **FETCH** を呼び出す

この③の **FETCH** 呼び出し後に、さらにフェッチするデータがある場合は、**SQLCODE** 変数は **0** に設定されます（フェッチ成功を意味します）。

FETCH への各呼び出し後、返り値は **DECLARE** 文の **INTO** 節により指定されたホスト変数にコピーされます。

④ **CLOSE** 文によって、カーソルをクローズする

CLOSE 文は、クエリの実行に使用したテンポラリストレージをクリーンアップします。**CLOSE** 呼び出しに失敗したプログラムでは、リソースの開放もれが起こる可能性があります。

動的 SQL

前述した埋め込み SQL との相違点は次のとおりです。

- 動的 SQL の入力パラメータは「?」を使用し、埋め込み SQL はホスト変数を使用する
- 動的 SQL からの出力値は、%SQL.StatementResult オブジェクトの API を使用し、埋め込み SQL はホスト変数を使用する
- 動的 SQL では、クエリのメタ情報を簡単に見ることができる

動的 SQL は、%SQL.Statement クラスによってサポートしています。

プログラムでは、%SQL.Statement クラスのインスタンスを生成して、それをクエリの作成、実行、繰り返しに使用します。

ObjectScript で%SQL.Statement オブジェクトを生成するためには、次のようにします。

```
Set statement = ##class(%SQL.Statement).%New()
```

この時点で、%SQL.Statement オブジェクトが SQL 文を生成できる状態になり、%SQL.Statement クラスの%Prepare メソッドを使用して、SQL 文を生成します。動的 SQL では、SELECT 文に限らず、この%Prepare メソッドを使用して、DDL、INSERT、UPDATE、DELETE などの文を作成できます。

5-5 IRIS SQL サーバ

IRIS SQL は、IRIS の一部として自動的にインストールされます。

IRIS が構成された後に、IRIS SQL としての、構成追加や設定をする必要はありません。

ほとんどの IRIS SQL 構成オプションは、DDL オペレーションやキャッシュされたクエリの管理に対応します。

Java や C 言語でプログラミングする際に、この IRIS SQL を利用する仕組みのことを IRIS SQL サーバと呼びます。

ODBC を使ったアクセス

IRIS SQL における C 言語の呼び出しレベルのインターフェイスは ODBC です。

多くのデータベース製品とは異なり、IRIS ODBC ドライバはネイティブのドライバであり、他のメーカ独自のインターフェイス上に構築されたものではありません。

ODBC システムのアーキテクチャは、次の 5 つの部分で構成されています。

- クライアントアプリケーション
- ODBC ドライバマネージャ
- ODBC クライアントドライバ
- データベースサーバ
- 初期化ファイル

クライアントアプリケーション

Microsoft の ODBC API に従って呼び出しを実行するアプリケーションのこと。

ODBC は、クライアントから DSN (Data Source Name) への接続を確立します。

これは、IRIS の特定のネームスペースと、その他の属性を指定するマッピングです。

クライアントアプリケーションから特定の DSN に接続するには、その DSN を ODBC ドライバマネージャに登録しておく必要があります。

ODBC ドライバマネージャ

この ODBC ドライバマネージャは、ODBC API を使用してアプリケーションからの呼び出しを受信し、これを IRIS クライアントドライバのような、登録済みのクライアントドライバに渡します。

また、クライアントアプリケーションがクライアントドライバ（さらにはデータベースサーバと）通信するために、必要なタスクを実行します。

ODBC クライアントドライバ

データベース固有のアプリケーションで、ODBC ドライバマネージャを通してクライアントアプリケーションからの呼び出しを受け取り、データベースサーバとの通信を提供するドライバです。

また、アプリケーションからの要求があれば、ODBC に関連するさまざまなデータ変換を実行します。

データベースサーバ

ODBC クライアントアプリケーションからの呼び出しを最終的に受け取る、実際のデータベースのことです。

これは、呼び出し元である ODBC クライアントドライバと同じマシンまたは異なるマシンのいずれに置いてもかまいません。

IRIS データベースは、複数の DSN をサポートすることができ、それぞれが複数の接続をサポートします。

初期化ファイル

ドライバマネージャの一連の構成情報ファイルです。

オペレーティングシステムによっては、クライアントドライバ情報を含むものもあります。

UNIX では、odbc.ini という実際のファイルで、Windows では、レジストリエントリです。

ODBC によるアクセスは、次の手順で確立されます。

- ① クライアントアプリケーションに含まれている ODBC 呼び出しが特定の DSN へ接続しようと試みて、クライアントアプリケーションにリンクされている ODBC ドライバマネージャが、その呼び出しを受け取る
- ② ODBC ドライバマネージャが IRIS ODBC 初期化ファイルを読み取り、IRIS ODBC クライアントドライバの位置を取得して、クライアントドライバをメモリにロードする
- ③ ODBC クライアントドライバがメモリにロードされ、IRIS ODBC 初期化ファイルを使用して DSN への接続情報の位置を確認する

この接続情報には、IRIS が稼働しているホスト、インストールされている IRIS の名称、およびデータを含むネームスペースなどの情報が含まれています。

ODBC クライアントドライバは、DSN と初期化ファイルから取得した情報を使用して、指定された IRIS とネームスペースに接続します。

- ④ 接続確立後、ODBC クライアントドライバによって IRIS データベースサーバとの通信が管理される

JDBC を使ったアクセス

IRIS SQL には、標準対応のレベル 4 JDBC クライアントが含まれています。

JDBC のクライアントを構成するためには、IRIS が自動的にインストールした JDBC ドライバ「cachejdbc.jar」を CLASSPATH 環境変数に設定する必要があります。

Java クライアントが JDBC を使って IRIS にアクセスするには、「ユーザ名」「パスワード」「サーバの IP アドレス」「ポート番号」「IRIS ネームスペース」を指定した URL が必要です。次のコードは、その例です。

```
String url = "jdbc:IRIS://127.0.0.1:51780/USER"; String user = "_SYSTEM";  
String password = "SYS";
```

JDBC を操作する Java プログラムでは、次の内容の確認が必要です。

- IRIS がインストール済みで稼働していることの確認
- IRIS を実行しているマシンの IP アドレスの確認
- 待ち受け状態になっている IRIS の TCP/IP ポート番号の確認

- 有効な SQL ユーザ名とパスワードかどうかの確認
- 接続用 URL に、有効な Cache ネームスペースが記述されているかどうかの確認

JDBC をサポートするものであれば、あらゆるツール、アプリケーション、開発環境で IRIS の JDBC が利用できます。

5-6 IRIS SQL のその他の機能

その他にも、IRIS SQL には、プログラミングを支援するさまざまな機能があります。

全文検索機能

%Text.Text クラスを使うことにより、全文検索機能を実現できます。

全文検索用のインデックスの切り出しは、Text Analytics 機能を利用します。

IRIS SQL では、テキストのインデックス作成と検索を実行するために、%Library.Text クラスと%Text パッケージが用意されています。

この機能を既存の%String プロパティで使用するには、%String を%Text に変更して、LANGUAGECLASS を設定します。

日本語テキストの場合の宣言は、次のようにします。

```
Property myDocument As %Text [ LANGUAGECLASS=" %TextJapanese" ];  
Index myFullTextIndex On myDocument As %iFind.Index.Basic(INDEXOPTION =  
0, LANGUAGE = "ja", LOWER = 1);
```

%Text プロパティを宣言して、そのオプションでインデックスを作成しておけば、次のように、SQL の%FIND 演算子を使用して全文検索を実行できます。

```
SELECT title FROM sample.book  
WHERE %ID %FIND SEARCH_INDEX(sample.book.titleIDX, ' 日本語' )
```

パフォーマンス最適化

IRIS SQL クエリの性能を最大限に発揮させるには、次の 2 種類の項目に注意する必要があります。

- インデックス
- EXTENTSIZE と SELECTIVITY

インデックス

インデックスをどのプロパティに定義するべきかを決定する際にも注意が必要です。

インデックスの設定が少なすぎたり、間違っていたりする場合は、主要なクエリの速度が大きく低下します。

また、インデックスの設定が多すぎる場合も、INSERT や UPDATE のパフォーマンスが低下する可能性があります。

最適なインデックスを定義することで、クエリパフォーマンスは向上します。

EXTENTSIZE と SELECTIVITY

クエリオプティマイザは、特定の SQL クエリを実行するもっとも効率のよい方法を決定します。

その際、クエリで使用されているテーブルの EXTENTSIZE 値と、そのクエリに使用される列の SELECTIVITY 値が検証されます。

オプティマイザが正しい方法を選択するためには、これらの値が正しく設定されていることが必要です。

- EXTENTSIZE 値：テーブルに保存されている行数。テーブル定義用クラス内の EXTENTSIZE パラメータで設定する
- SELECTIVITY 値：標準的な列の値を検索するクエリの結果として返される、テーブル内の行の割合。

テーブル定義用クラス内の SELECTIVITY パラメータで設定する

テーブルが、実際のデータを持つ場合は、IRIS システム管理ポータルでのテーブルのチューニング機能によって、SELECTIVITY 値が自動的に計算され設定されます。

運用システムでの実稼働あるいはベンチマーク等を実施する場合には、ある程度データが蓄積された段階で、このテーブルチューニングを実施することが必須です。

この操作を行わないと想定されるクエリー性能が得られない場合があります。

この操作は管理ポータル>システムエクスプローラ>SQL から右ペイン上のアクションメニューからテーブルチューニングを選択して実行します。

その結果は、クラスのストレージ定義内で値が上書きされます。

プログラムによっては、テーブルのチューニング機能によって生成された値を使用する必要があるため、事前によく調査した場合を除いて、SELECTIVITY 値は手動で設定しないでください。

SQL 外部キーの定義

SQL 外部キーは、テーブル内の 1 つまたは複数のフィールドと、他のテーブルのキー（一意のインデックス）間の整合性制約を定義します。

一般的には、オブジェクトプログラミングでは外部キーを使用しません。

その代わりに、オブジェクトアクセスでの操作性に優れたリレーションシップを使用します。

リレーションシップは、手動で定義する外部キー定義と同等の整合性制約を、SQL とオブジェクトアクセスの両方に自動的に適用します。

1 対多の多側のデータが大量の場合には、リレーションシップではなく外部キーを使用したほうが性能面で優位です。

SQL 外部キーをクラス定義に追加するには、次のような 2 つの方法があります。

- クラスエディタを使用して、クラス定義の値を編集する方法
- 新規外部キーウィザードを使用する方法

クラスエディタを使用して、クラス定義の値を編集する方法

クラスエディタを使用して **SQL** 外部キーを定義するには、クラスエディタの空の行にカーソルを置き、次のように外部キー宣言を記述します。

```
Class MyApp.Company Extends %Persistent [ClassType = persistent]
{
Property State As %String;
ForeignKey StateFKey(State) References StateTable(StateKey);
}
```

新規外部キーウィザードを使用する方法

新規外部キーウィザードを開くには、スタジオのツールバーにある [クラス] から [追加] を選んで、表示されたコンテキストメニューの [新規外部キー] を選択します (図 5-1)。

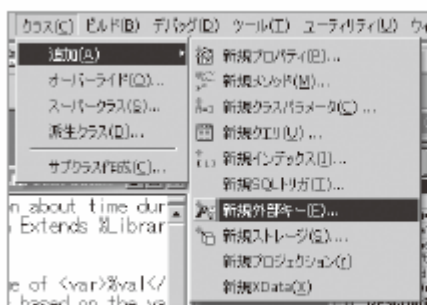


図 5-1 : ツールバーのメニューから [新規外部キー] を選択する方法

または、クラスインスペクタで右クリックして [新規外部キー] を選択する方法もあります (図 5-2)。

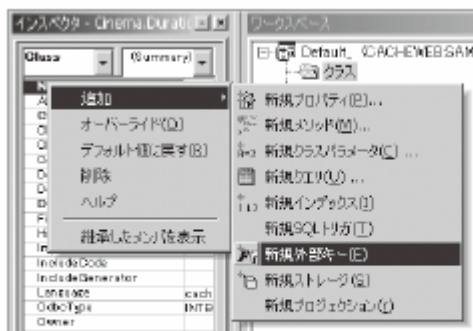


図 5-2: クラスインスペクタから [新規外部キー] を選択する方法

また、ツールバーにある [新規外部キー] アイコンを選択することも可能です。

すると、「新規外部キーウィザード」画面が表示され、各種の情報入力が促されます。

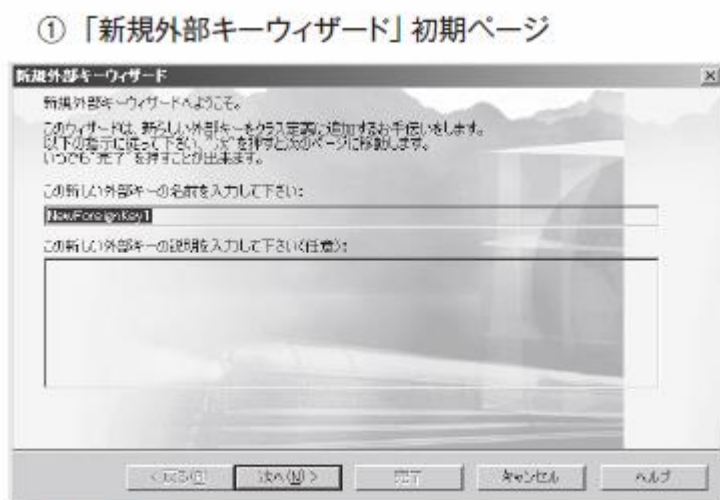


図 5-3: 「新規外部キーウィザード」初期ページ

- ・ [この新しい外部キーの名前を入力して下さい]

必須項目です。

これは、有効な外部キー名である必要があります。また、定義済みの既存の外部キーと同じ名前を付けることはできません。

- ・ [この新しい外部キーの説明を入力して下さい]

任意の項目です。

この説明は、このクラスドキュメントがオンラインクラスライブラリドキュメントで表示されるときに使用されます。

HTML タグを記述することもできます。

② 「新規外部キーウィザード」属性ページ



図 5-4: 「新規外部キーウィザード」属性ページ

この 2 番目の属性ページで、外部キーによって制約するクラスの 1 つ以上のプロパティを選択します。

③ 「新規外部キーウィザード」キー構築ページ



図 5-5: 「新規外部キーウィザード」キー構築ページ

この 3 番目のキー構築ページで、外部キープロパティ制約に使用する値を指定するための、クラスとクラス内のキーの両方を選択します。

必要な情報をすべて入力したら、[完了] ボタンをクリックします。

指定しなかった情報には既定値が設定されます。

この新規外部キーウィザードの完了後、クラスエディタウィンドウは更新され、新規の外部キー定義が表示されるようになります。

なお、この外部キーをさらに変更したい場合は、クラスエディタまたはクラスインスペクタを使用します。

トリガの定義

トリガとは、特定の **SQL** イベントに応答して実行されるコードのことです。

各トリガには、1 行以上の **ObjectScript** コードが含まれます。

このコードは、トリガが定義されたイベントが発生した場合に、**SQL** エンジンによって呼び出されます。

なお、特定のテーブルにトリガを定義するには、次のような 2 つの方法があります。

- スタジオを使用して、テーブルに対応するクラス定義に **SQL** トリガ定義を追加する
- **DDL CREATE TRIGGER** コマンドを使用して、トリガを生成する

スタジオを使用して、テーブルに対応するクラス定義に SQL トリガ定義を追加する

次のコード例を確認してください。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
  /// このトリガは、挿入ごとに LogTable を更新する
  Trigger LogEvent [ Event = INSERT, Time = AFTER ]
  {
    // 挿入行の id を取得
    Set id = {ID}
```

DDL CREATE TRIGGER コマンドを使用して、トリガを生成する

各トリガは、テーブルに「INSERT」「UPDATE」「DELETE」イベントのいずれかが発生したときに動作するように定義されます。

また、トリガが、イベント発生の前に動作するか後で動作するかも指定可能です。

1つのイベントに複数のトリガを定義することもできます。

この場合は、トリガの「Order」キーワードによって、複数のトリガが動作する順番を管理します（Order 値が低いトリガから順番に動作します）。

複数のトリガが同じ Order 値を持つ場合は、動作する順番は不定になります。

ストアドプロシージャの定義

ストアドプロシージャを定義するには、クラス定義内で定義する方法と DDL を使用する方法の 2 種類あります。

- クラス定義内で定義する方法

- DDL を使用する方法

クラス定義内で定義する方法

メソッドストアードプロシージャを定義するには、次のコード例のように、クラスメソッドを定義して、その **SqlProc** キーワードを設定します。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
  /// このプロシージャは、テリトリの全売り上げを計算する
  ClassMethod FindTotal(territory As %String) As %Integer [SqlProc]
  {
    // 全売り上げを計算するために埋め込み SQL を使う
    &sql(SELECT SUM(SalesAmount) INTO :total
    FROM Sales
    WHERE Territory = :territory
    )
    Quit total
  }
}
```

このクラスのコンパイル後、「**MyApp.Person_FindTotal**」というストアードプロシージャとして、**FindTotal** メソッドが **SQL** に投影されます。

メソッドの **SqlName** キーワードによって、**SQL** がプロシージャに使用している名前を変更することができます。

結果セットストアードプロシージャを定義するには、次のコードのように、クラスクエリを定義して、その **SqlProc** キーワードを設定します。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
/// このプロシージャは、名前順に並んだ人の集合を返す
Query ListPersons(name As %String = "") As %SQLQuery [ SqlProc ]
{
SELECT ID, Name
FROM Person
ORDER BY Name
}
}
```

このクラスのコンパイル後、「**MyApp.Person_ListPersons**」というストアードプロシージャとして、**ListPersons** クエリが **SQL** に投影されます。

クエリの **SqlName** キーワードによって、**SQL** がプロシージャに使用している名前を変更することができます。

IRIS SQL から **MyApp.Person_ListPersons** が呼び出されると、クエリの **SQL** 文によって定義されたレコードセットを自動的に返します。

ユーザが記述した **SQL** 文ではないコードに基づいて、レコードセットストアードプロシージャを生成することもできます。

DDL を使用する方法

CREATE FUNCTION と CREATE PROCEDURE コマンドによって、DDL を使用したストアドプロシージャを定義できます。

こうして定義されたストアドプロシージャは、次の 2 種類の方法で使用できます。

- SQL の CALL 文を使用して、ODBC や JDBC からストアドプロシージャを呼び出す（埋め込み SQL や動的 SQL では、この CALL 文をサポートしていない）
- SQL クエリにおいて、ストアド関数（メソッドベースのストアドプロシージャで値を返すもの）を組み込み関数であるかのように使用する

5-7 IRIS SQL サンプルプログラム

それでは、第 3 章の ObjectScript のサンプルプログラムを、IRIS SQL を使う形に書き換えてみます。（セットアップ環境に既に存在します）

```

ClassMethod placeOrderWithFullSQL(customerId As %Integer) As %Boolean
{
    if customerId="" write "顧客 ID を入力してください",! quit '$$$OK

    set      result=##class(%SQL.Statement).%ExecDirect("select      Name      from
Sales.Customer where id = ?",.customerId)
    set resultset = result.%CurrentResult

    if 'resultset.%Next(.sc) write "顧客 ID が登録されていません" quit '$$$OK

    write resultset.Name_"様、いつもご利用ありがとうございます",!!
    set result = ##class(%SQL.Statement).%New()
    set result.%SelectMode = 2 //表示モード

    set sc = result.%Prepare("select id,name,unitprice from Sales.Product")
    if $$$ISERR(sc) Do $system.Status.DisplayError(sc) quit '$$$OK
    set resultset = result.%Execute()

    write "商品 ID",?20,      "商品名",?40,"単価",!
    write "-----",!

    While resultset.%Next(.sc) {
        If $$$ISERR(sc) quit
        write resultset.id,?20,resultset.name,?40,resultset.unitprice,!
    }

    write !

    set itemNo = 0
  
```

```

for {
    read "注文したい商品 ID を入力してください ", pid,!

    if pid="" quit

    set result=##class(%SQL.Statement).%ExecDirect("select      Name      from
Sales.Product where id = ?",.customerId)
    set resultset = result.%CurrentResult

    if 'resultset.%Next(.sc) write "商品が存在しません",! continue

    read "数量を入力してください ", amount,!
    if amount'?.N write "正しい数値を入力してください",! continue
    set itemNo = itemNo + 1
    set items(itemNo) = $listbuild(pid,amount)
}

if itemNo = 0 quit '$$$OK

//&sql は、変数スコープのコンテキストが異なるため、簡易的に%をつけて、変
数を PUBLIC 変数にする

set %customerId = customerId
set %purchaseDate = $piece($horolog,",",1)

tstart

&sql(insert into Sales.PurchaseOrder
(Customer,PurchaseDate)
VALUES (:%customerId,:%purchaseDate))

if SQLCODE '=0 {
    write "注文情報の登録に失敗しました",!

```

```
trollback
quit '$$$OK
}

set %orderid = %ROWID

set success = 1

for i = 1:1:itemNo {
    set %pid = $list(items(i),1)
    set %amount = $list(items(i),2)

    &sql(insert into Sales.OrderItem
        (TheOrder,Product,Amount)
        VALUES (:%orderid,:%pid,:%amount))

    if SQLCODE !=0 {
        write "注文明細の登録に失敗しました",!
        trollback
        set success = 0
        quit
    }
}

if success {
    write "注文を受け付けました",!
    tcommit
}

kill %customerId,%orderid,%purchaseDate,%ROWID

quit '$$$OK
}
```

第 6 章 Caché Server Pages による Web アプリケーション構築

Caché Server Pages (CSP) は、IRIS を使ってインタラクティブな Web ベースのアプリケーションを構築するためのアーキテクチャであり、そのためのツールセットです。CSP は、IRIS データベースのデータを使って Web ページを動的に生成します。

さらに CSP は、Web ベースのデータベースアプリケーションを構築するために不可欠ないくつかの機能を提供し、また、2 つの開発スタイルをサポートしています。

本章では、CSP の概要について説明します。

6-1 CSP の機能

CSP を使ってできることは、次のとおりです。

- 分単位に変更がある在庫データを表示する
- 何千ものアクティブユーザが存在する Web コミュニティをサポートする
- IRIS データベースに保存されたユーザ情報に基づくページのパーソナライズ
- 個人の要求、セキュリティ認可などに依存したユーザデータに基づくページのカスタマイズ
- HTML、XML、イメージ、その他のバイナリデータおよびテキストデータを提供する
- IRIS データベースと密接に結び付いていることによる高速パフォーマンスの提供

また、CSP は、Web ベースのデータベースアプリケーションを構築するために、次の機能を提供しています。

- セッション管理
- ページ認証
- Web ページで会話的なデータベース操作の実行

CSP による Web 開発のためのスタイルは、次の 2 種類です。

- クラスを使ったアプリケーション開発に対する、オブジェクトフレームワークの提供
- HTML ファイルを使ったアプリケーション開発のための、Web ページ内にオブジェクトやサーバサイドスクリプトを含めることができる HTML ベースのマークアップ言語の提供

それでは、実際に CSP による開発について説明していきましょう。

6-2 CSP の学習を始める前に

CSP が動作するためには、Web サーバと IRIS がインストールされていなければなりません。

また、CSP アプリケーションを作成するためには、次の 2 つの準備が必要です。

- 実運用 Web サーバと IRIS 提供プライベート Web サーバの準備
- Web サーバと CSP ゲートウェイの構成

実運用 Web サーバと IRIS 提供プライベート Web サーバの準備

IRIS は、システム管理ポータルを動作させるために「プライベート Web サーバ」という最小限の Web サーバを提供しています。

その「プライベート Web サーバ」を使って、CSP サンプルや CSP ページを表示できますが、実運用の環境下で堅牢な CSP アプリケーションを動作させることはできません。実運用用には、Apache やマイクロソフトの Internet Information Services (IIS) などの Web サーバをインストールする必要があります。

「プライベート Web サーバ」は、Apache Web サーバのミニマムビルドをベースとして、非標準の TCP ポートをリッスンするように構成されています（既定値は 52773 です）。

Web サーバと CSP ゲートウェイの構成

IRIS のインストール定義ファイルには、Web サーバと CSP ゲートウェイを構成するスクリプトが含まれています。

指示に従って IRIS をインストールする際に、サポートする Web サーバの一般的な構成をインストールすれば、CSP ゲートウェイがそのシステム上で動作します。

CSP で知っておくべきこと

CSP を使って効率よく開発するには、ある程度は、次のことを知っておいたほうがいいでしょう。

- IRIS オブジェクト、ObjectScript 言語
- HTML
- JavaScript
- SQL

CSP サンプルについて

セットアップ環境には、CSP のサンプルアプリケーションも自動的にインストールされます。

そのサンプルをチェックするには、次の操作を行います。

- ② IRIS を開始する（既に開始しているはずです。）
- ② ブラウザを起動し、CSP サンプルメニュー
（<http://localhost:52780/csp/user/login.htm>）に移動する
- ③ ログインページが表示されるので、ユーザー名: **Taro** パスワード: **Taro** を入力します。
- ④ 注文画面が表示されます。

CSP ドキュメントについて

CSP に関するドキュメントは、IRIS ドキュメント内の、次の箇所を参照してください。

- 「Caché Server Pages を使う」 : CSP ページを作成する方法
- 「CSP HTML タグリファレンス」 : CSP タグについての詳細

CSP に関連するクラスリファレンス情報は、次の箇所を参照してください。

- 「%CSP.Pages」
- 「%CSP.Session」

その他にも、「CSP Web ゲートウェイドキュメント」「CSP ゲートウェイの構成に関するオンラインヘルプ」が、システム管理ポータル「CSP Web ゲートウェイ管理ページ」から利用できます。

また、「CSP ゲートウェイ構成ガイド」は、CSP ゲートウェイをマニュアルで構成する必要がある場合に利用します。

6-3 最初の CSP ページを作成する

本節では、「こんにちは、みなさん」と表示する簡単な CSP ページ作成のための、次の 2 種類の方法を紹介します。

- Web ページオブジェクトを使って、クラスを基本とした CSP ページを作成する
- マークアップ HTML ファイルを使って、HTML を基本とした CSP ページを作成する

クラスによる CSP ページの作成手順

まず、クラスによる CSP ページの作成手順を説明しましょう。

- ① スタジオを起動する
- ② [ファイル] → [プロジェクトの新規作成] で、ローカルデータベース USER ネームスペースに新規プロジェクトを作成する (図 6-1)

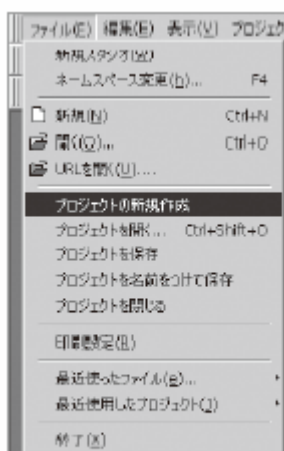


図 6-1 : プロジェクトの新規作成

- ③ [ファイル] → [新規] で表示された [新規作成] ウィンドウのカテゴリから [一般] をクリックし、テンプレートから [Cache クラス定義] を選択する (図 6-2)

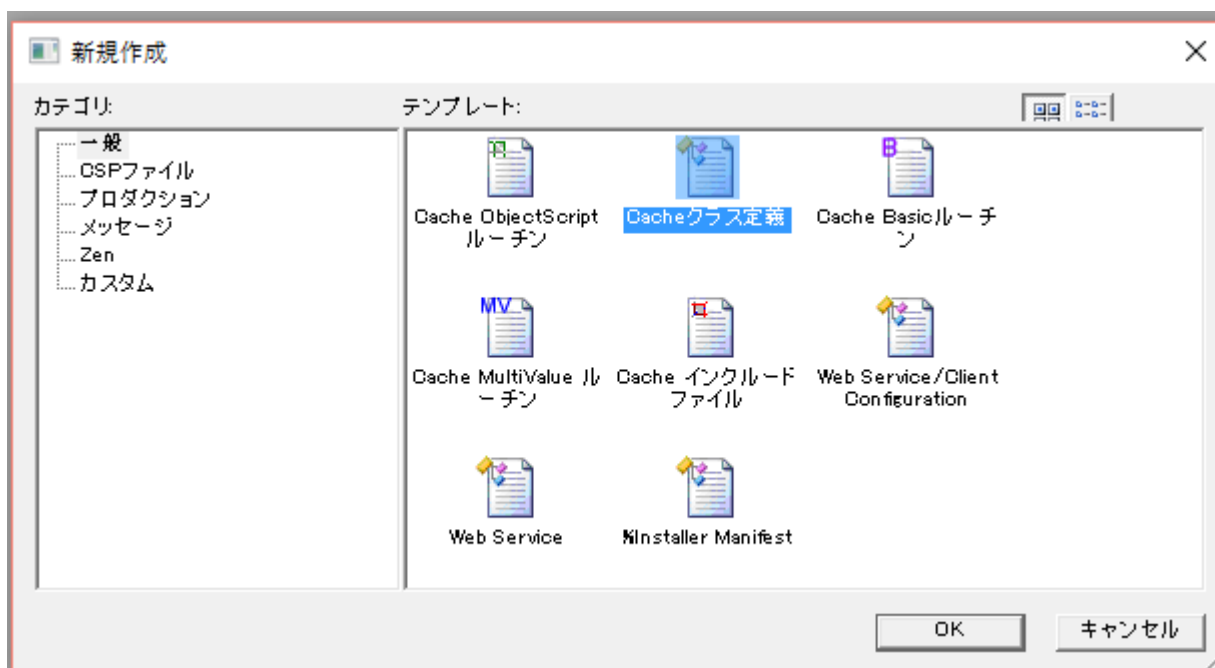


図 6-2 : 「新規作成」 ウィンドウ

- ④ 「新規クラスウィザード」の最初のページで、パッケージ名に「TEST」、クラス名に「Hello」と入力する（図 6-3）

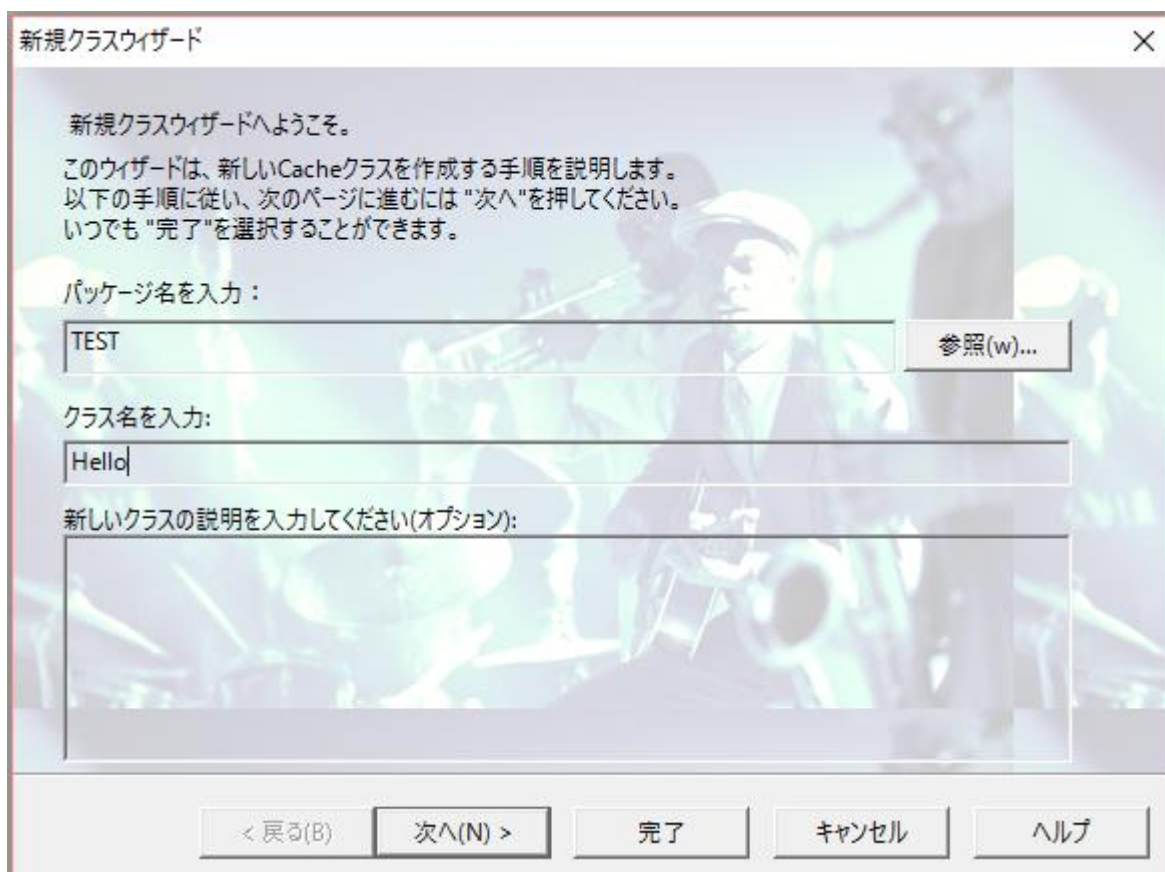


図 6-3 : [新規クラスウィザード] の最初のページ

⑤ ウィザードの 2 番目のページで、クラスタイプとして [CSP (HTTP イベントの処理に使用)] を選択する (図 6-4)

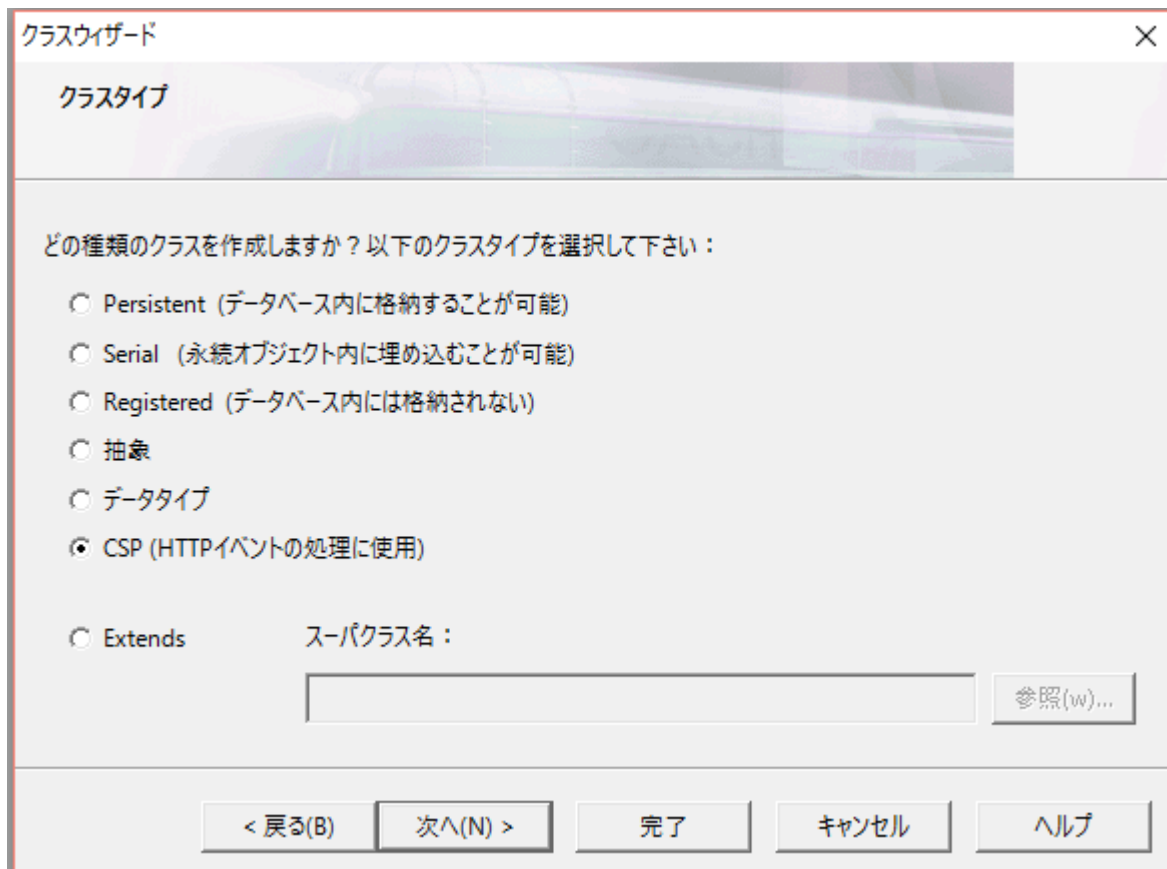


図 6-4 : [新規クラスウィザード] の 2 番目のページ

⑥ [完了] をクリックする。

その後、スタジオのクラスエディタ内に、次のコードのような新しい CSP クラス定義が表示される

```
Class Test.Hello Extends %CSP.Page [ ProcedureBlock ]
{
  ClassMethod OnPage() As %Status
  {
    &html<<html>
    <head>
    </head>
    <body>>
    ; To do...
    &html<</body>
    </html>>
    Quit $$$OK
  }
}
```

- ⑦ 「OnPage」メソッドの中のコメント「; To do...」の行を次のように置き換える

```
Write "<b>こんにちは、みなさん</b>";!
```

- ⑧ [ビルド] → [コンパイル] で、クラス保存後にコンパイルする

これで、[表示] → [ブラウザで表示] をクリックすると、ブラウザに「こんにちは、みなさん」と表示されます（図 6-5）。

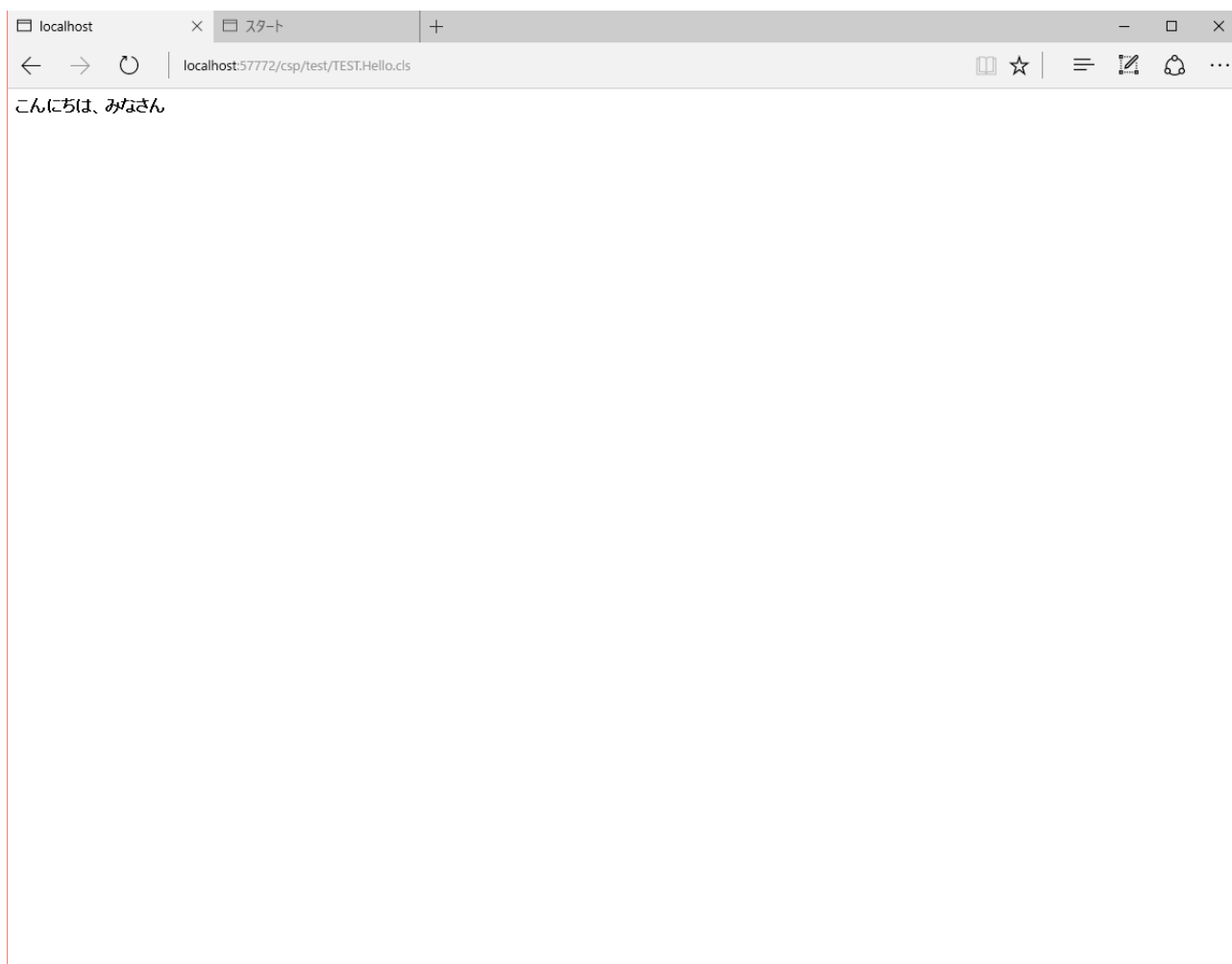


図 6-5 : 「こんにちは、みなさん」のブラウザ表示

この CSP ページは、完成された 1 つの CSP アプリケーションですが、その動作は次のようになっています。

ここでは、ブラウザ、ローカル Web サーバ、IRIS アプリケーションサーバは、すべて同じマシン上で動作している前提で説明します。

実環境では、すべて異なるマシン上で稼働していることが多いでしょう。

- ① ブラウザが、**Test.Hello.cls** に対する要求を、指定したネームスペース内にあるローカル Web サーバに送る
- ② 受け取ったローカル Web サーバは、この要求を CSP ゲートウェイにパスし、ゲートウェイが、その要求を IRIS の CSP サーバに送る
- ③ 受け取った CSP サーバは、**Test.Hello** という名のクラスを探し、その **OnPage** メソッドを起動する

④ OnPage メソッドが主デバイスに書き込む出力が、CSP ゲートウェイとローカル Web サーバを通してブラウザに送り戻される

この簡単な CSP ページとその動作については、CSP の最重要部分です。

その他の CSP の数々の機能は、この振る舞いの上に構築されています。

例えば次のコードは、簡単なロジックを追加した例です。

「こんにちは、みなさん」を含む行のあとに、次のコードを追加してみてください。

```
Write "<ul>",!  
For i = 1:1:10 {  
  Write "<LI> これは、",i,"行目です ",!  
}  
Write "</ul>",!
```

図 6-6 が、コンパイルしてブラウザで表示させた例です。

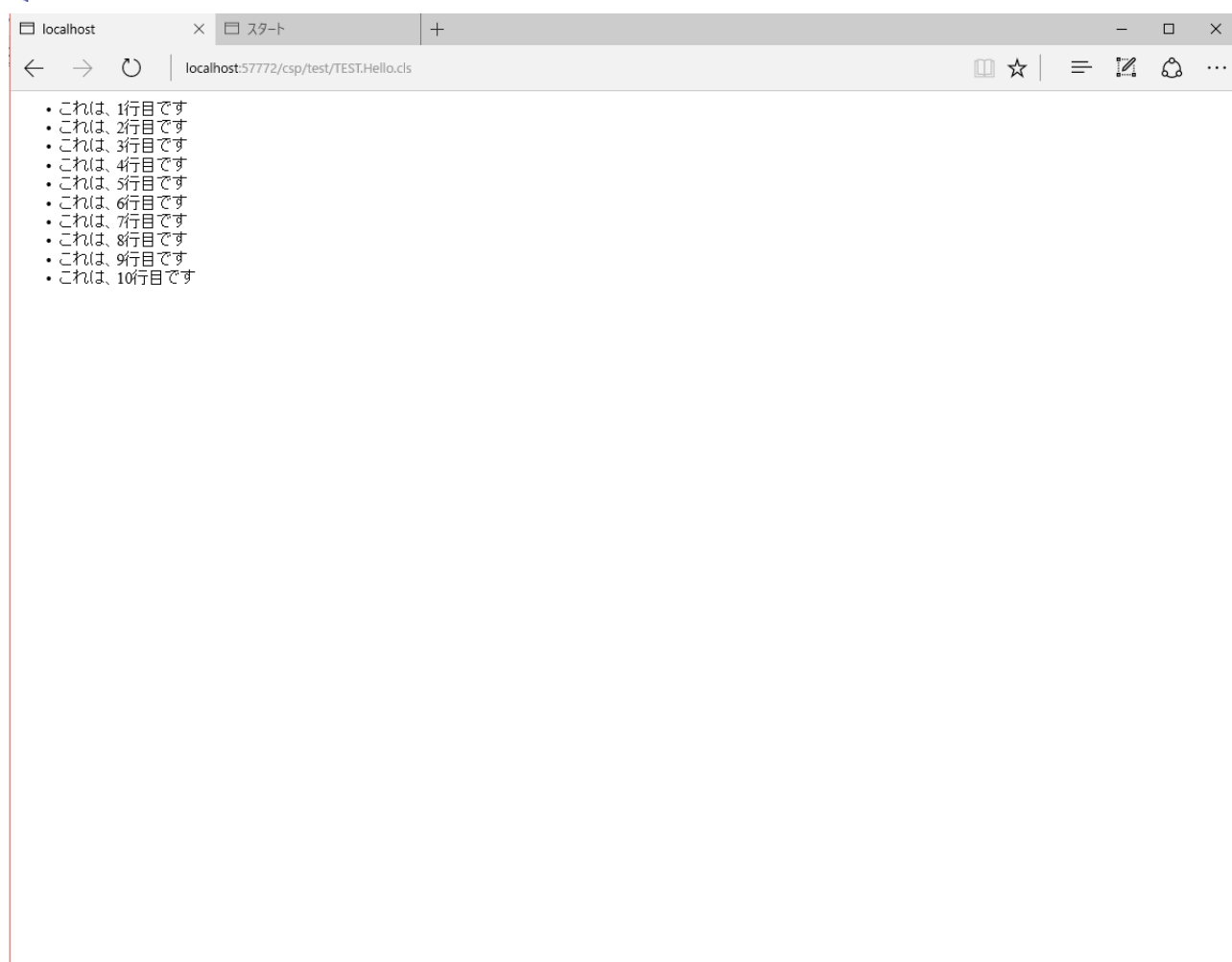


図 6-6 : ロジックを追加した CSP ページ

HTML タグを使った CSP ページの作成

CSP ページを作成するもう 1 つの方法は、HTML ファイルを作成して、CSP コンパイラに IRIS クラスに変換してもらう方法です。

HTML ファイルで「こんにちは、みなさん」ページを作成するには、次のように行います。

スタジオを起動して、[ファイル] → [新規] で表示された [新規作成] ウィンドウのカテゴリから [CSP ファイル] をクリックし、[Cache Server Page] を選択する (図 6-7)

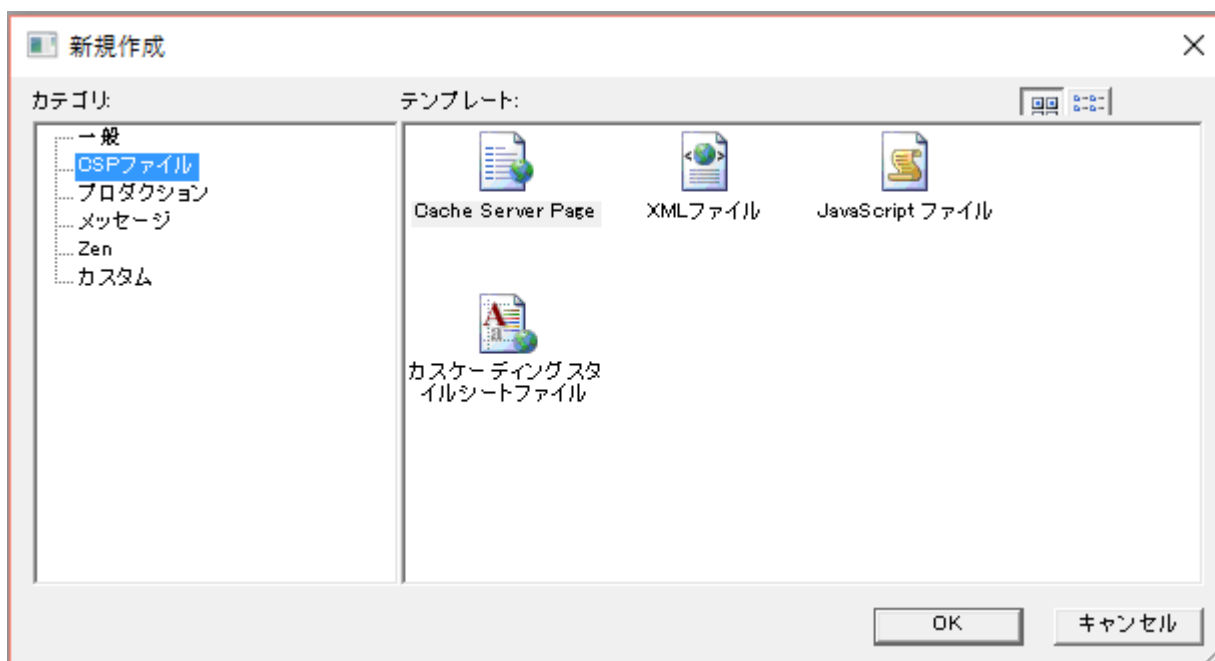


図 6-7 : [新規作成] ウィンドウの [CSP ファイル]

② スタジオの CSP エディタで、表示されているコードを次のコードと入れ換える

```
<html>
<body>
<b>こんにちは、みなさん!</b>
</body>
</html>
```

③ このページを USER ネームスペースの「csp/user」ディレクトリに「Hello.csp」という名前で保存する (図 6-8)

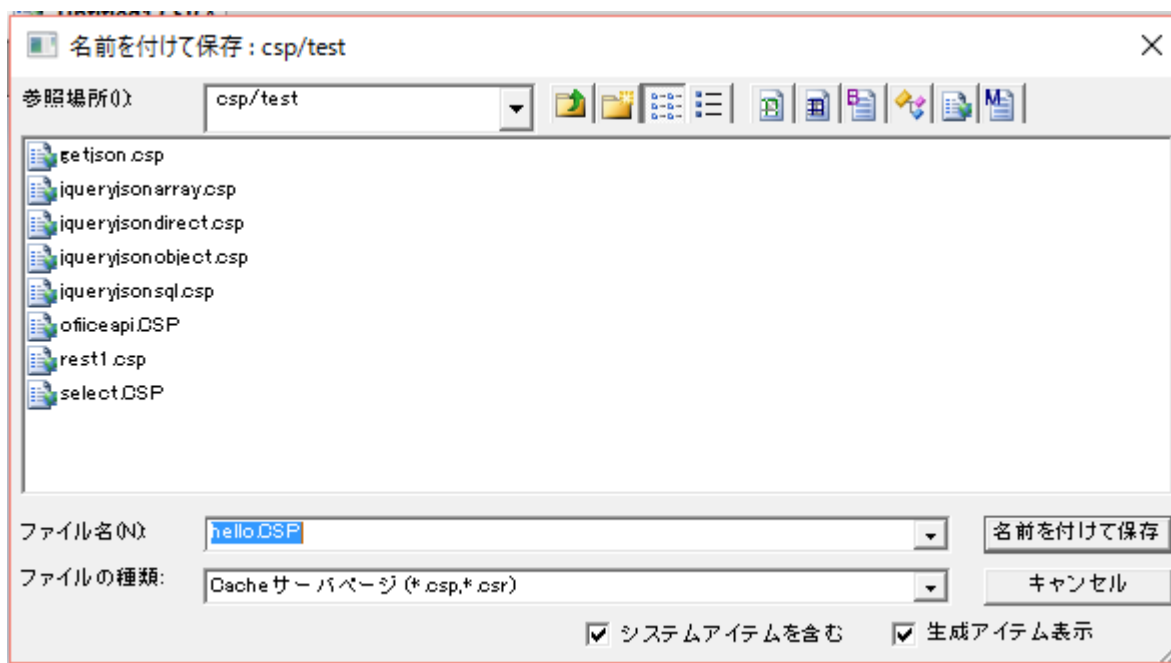


図 6-8 : 「Hello.csp」の保存

これで、[表示] → [ブラウザで表示] をクリックすると、上記の例と同様に、ブラウザに「こんにちは、みなさん」と表示されます（図 6-9）

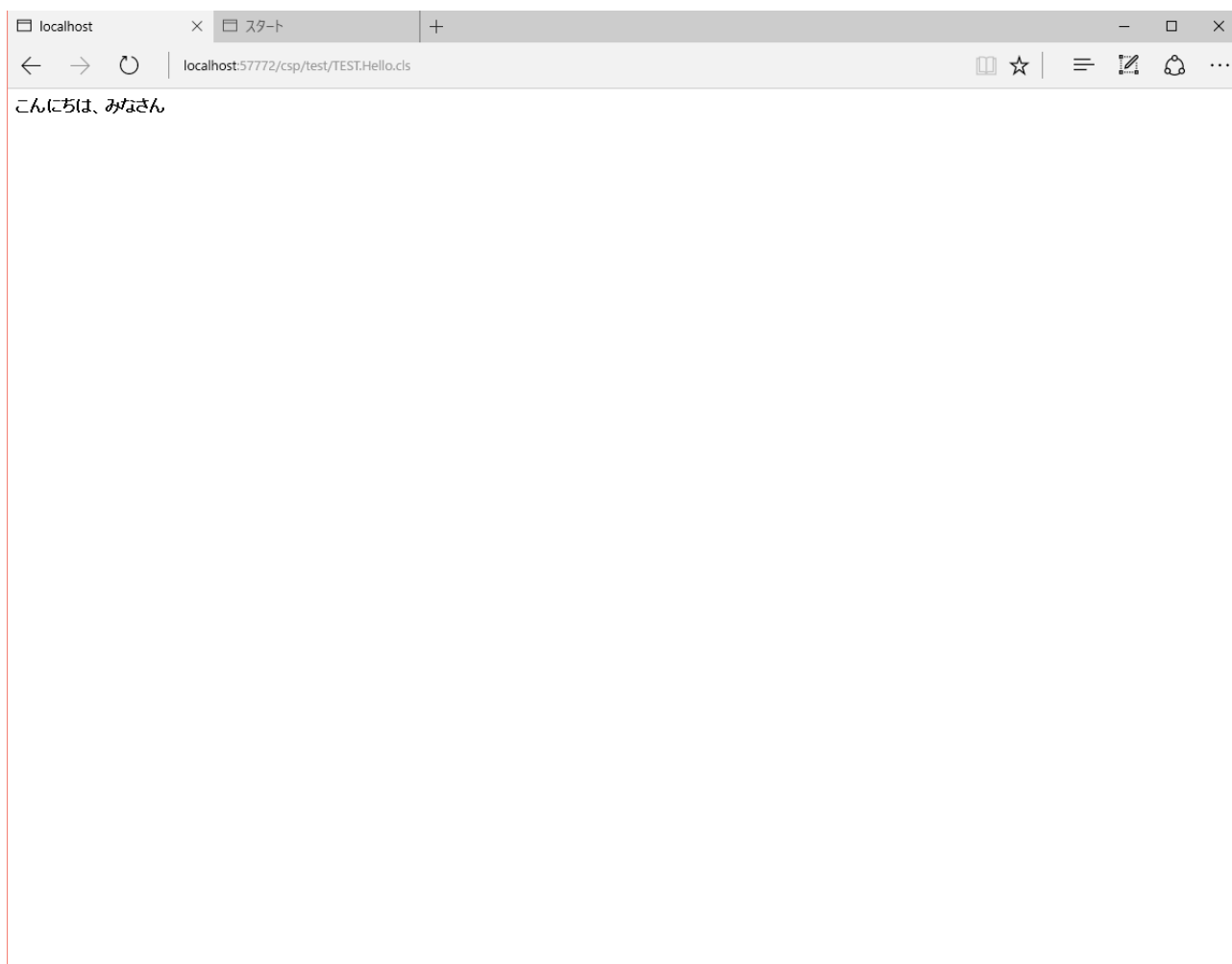


図 6-9 : HTML タグを使った CSP ページの表示

このアプリケーションは、次のように動作します。

- ① ブラウザが、**Hello.csp** に対する要求をローカル Web サーバに送る
- ② 受け取ったローカル Web サーバは、この要求を CSP ゲートウェイに送り、ゲートウェイが、その要求を IRIS の CSP サーバに送る
- ③ 要求を受け取った IRIS の CSP サーバは、「**Hello.csp**」ファイルを探して、それを CSP コンパイラに渡す
- ④ CSP コンパイラは、「**csp.Hello**」という新しいクラスを作成し、その中に **OnPage** メソッドを作る
- ⑤ CSP サーバは、その新しく生成された **OnPage** メソッドを起動して、その出力をブラウザに送る

このコンパイルに関しては次のような特徴があります。

- OnPage メソッドには、Hello.csp ファイルの内容が書き出される（実際には、複数のメソッドが生成され、各々は、OnPage メソッドから呼び出される）
- このコンパイル作業は、その.csp ファイルが生成されたクラスよりも新しいときだけ発生する
- これ以降のリクエストは、直接、その生成されたクラスに送られる

CSP コンパイラは、実際には XML/HTML に特化した処理エンジンであり、次のようなことが可能となっています。

- HTML ページ内で、サーバ上のスクリプトや表現を処理する
- 特定の HTML タグを認識したときに、サーバ上でアクションを実行する

例として、このページにロジック部分を追加してみましょう。

```
<html>
<body>
<b>こんにちは、みなさん!</b>
<script language="Cache" runat="server">
// このコードは、サーバ上で実行
Write "<ul>",!
For i = 1:1:10 {
Write "<li> これは、",i,"行目です。",!
}
Write "</ul>",!
</script>
</body>
</html>
```

6-4 CSP のアーキテクチャ

CSP は、Web サーバ、CSP ゲートウェイ、IRIS サーバという 3 つのソフトウェアコンポーネントを使います。

Web サーバと IRIS サーバは、1 つ以上のサーバによって実装することができます。

開発中は、3 つのコンポーネントをすべて 1 つの PC 内に設置することも可能です。

大規模開発では、一般的に、2 層または 3 層構成による複数 Web サーバと IRIS サーバを構成します。

各コンポーネントの役割

IRIS での Web サーバの役割は次のとおりです。

- HTTP リクエストの（通常はブラウザから）受付
- パーミッションのチェック
- 静的な内容の処理
- CSP リクエスト（URL の最後が `.csp` または `.cls` で終わるもの）の CSP ゲートウェイへの送信
- リクエストを送信する IRIS サーバの決定
- IRIS サーバへの（毎回接続する手間を省くための）接続維持

CSP サーバは、IRIS サーバ上で稼働していて、CSP ゲートウェイからのリクエストの処理を専門にしています。

CSP サーバは、次の処理を行います。

- アプリケーション用 HTTP リクエストの受け取り
- アプリケーションの構成設定のチェック
- リクエストにもとづくクラスの実行

CSP による情報の流れ

CSP リクエストは、標準的な Web サーバと標準的な HTTP プロトコルによって処理されます。

CSP は、Web サーバと IRIS 間の接続を管理し、ページを生成するためのアプリケーションコードを起動します。

そのリクエストと戻りのプロセスは、次のとおりです。

- ① HTTP クライアント（通常は Web ブラウザ）が、HTTP を使って Web サーバからページを要求する
- ② Web サーバは、この要求を CSP リクエストと認識し、高速なサーバ API を使って CSP ゲートウェイに転送する
- ③ CSP ゲートウェイは、対話をする IRIS サーバを決定し、ターゲットシステム上の CSP サーバに要求を転送する
- ④ IRIS 内で稼働する CSP サーバが、その要求を処理して、Web サーバにページを戻す
- ⑤ Web サーバは、そのページを表示のためにブラウザに送る

静的ファイル

CSP は、すべて IRIS サーバによって処理されます。

標準的な Web アプリケーションでの静的コンテンツは、データベースサーバでは処理されずに、Web サーバによって処理されます。

既定では、IRIS は、IRIS サーバがすべての静的コンテンツを処理するように Web サーバを構成しています。

これを、標準的な方法である、Web サーバが静的ページを処理するように変更することもできます。

ただし、既定の設定まま IRIS サーバから静的ファイルを処理するようにしたほうが、CSP ゲートウェイ内に静的ファイルがキャッシュされるため、最適化が可能となります。

6-5 CSP の HTTP リクエスト

CSP の主な仕事は、HTTP リクエストに対する応答の動的なコンテンツを提供することです。

HTTP はステートレス（状態を維持しない）プロトコルであり、クライアントとサーバ間の接続はリクエストを処理する間だけ存続します。

各 HTTP リクエストには、リクエストヘッダがあり、リクエストタイプ（GET や POST）、URL、バージョン番号を指定します。

リクエストには追加の情報を含むこともできます。

CSP は、自動的にどの HTTP リクエストを処理すべきかを判断して、IRIS サーバ上で稼働する適当なクラスに割り当てます。

CSP 実行環境

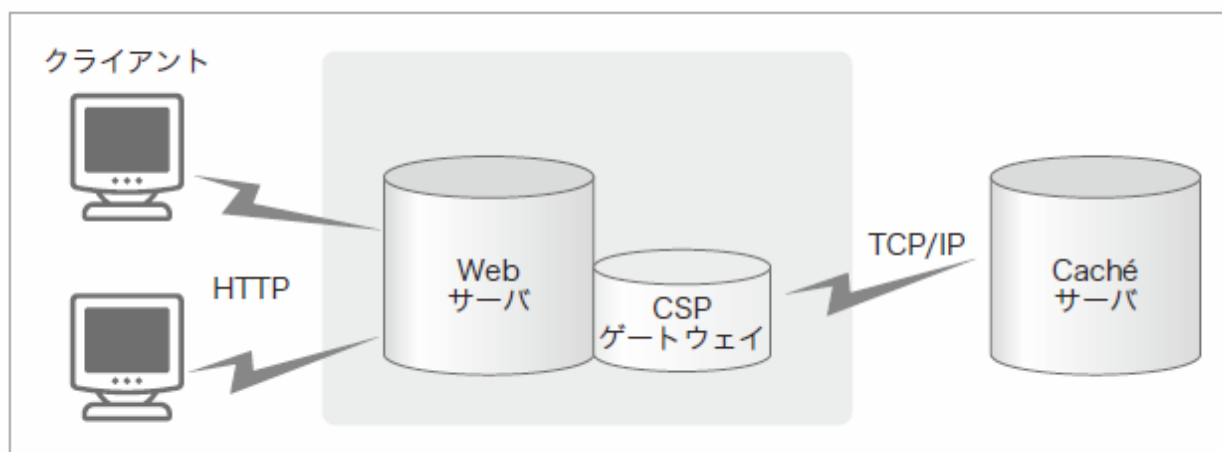


図6-10：CSPとHTTPリクエストのアーキテクチャ

CSP アプリケーションの実行環境は、次のもので構成されます。

- HTTP クライアント（Web ブラウザなど）
- Web サーバ（Apache、IIS など）
- CSP ゲートウェイ
- IRIS サーバ

HTTP リクエスト処理

図 6-11 は、CSP が HTTP リクエストを処理するときのイベントの流れを表しています。

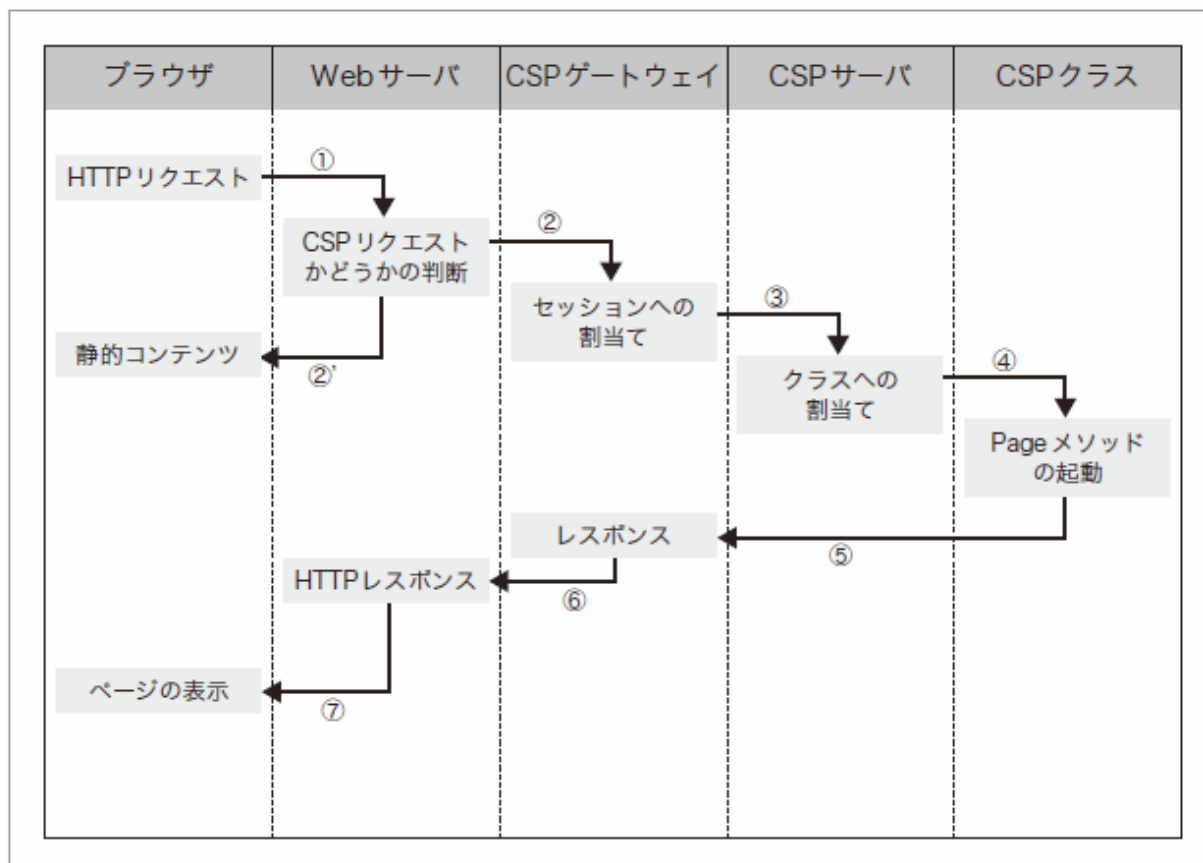


図 6-11 : リクエスト処理

- ① ブラウザが HTTP リクエストを行う
- ② Web サーバは、それが CSP リクエストかどうか判断し、CSP ゲートウェイに割り当てる
- ②' アプリケーション構成によっては、静的コンテンツを処理する
- ③ CSP ゲートウェイは、そのリクエストを再パッケージ化して、正しい IRIS サーバに送る
- ④ IRIS サーバは、メッセージを解析し、その要求が静的ページなのか CSP クラスなのか決定する

その要求が静的ファイル（.html、.jpg など）ならば、IRIS サーバはファイルをローカルファイルシステム内で探して、その内容をクライアントに送信します。

その要求がクラスに対してであれば、イベントを処理するイベントハンドリングクラスを決定し、そのクラスの Page メソッドを起動します。

- ⑤ その Page メソッドの出力（または静的ページ）は、HTTP レスポンスとして CSP ゲートウェイに戻される
- ⑥ CSP ゲートウェイは、HTTP レスポンスを Web サーバに手渡す
- ⑦ Web サーバは、そのレスポンスを Web ブラウザに戻し、ブラウザはレスポンスを処理する

Web サーバと CSP ゲートウェイ

HTTP リクエストが開始されるのは、HTTP クライアントが Web サーバにメッセージを送信したときです。

CSP ゲートウェイとは、特定イベントを処理するために Web サーバによって使用される DLL または共有ライブラリです。

CSP ゲートウェイが HTTP リクエストを処理するのは、URL のディレクトリパスが、Web サーバ内で定義された正しいアクセス権限を持っているときに限ります。

また、CSP ゲートウェイは、次の機能を提供しています。

- 最小限の処理負荷をかけ、ほとんどの作業を IRIS サーバに負わせることにより、Web サーバにより多くのリソースを提供する
- 名前の付いた CSP サーバへのコネクションプールを維持する
- 内部的につながった複数 CSP サーバの使用を許可し、フェイルオーバーオプションを提供する

CSP サーバ

CSP サーバとは、IRIS サーバ上で稼働しているプロセスであり、CSP ゲートウェイからのリクエストを処理しています。

IRIS サーバは、必要な分だけ CSP サーバプロセスを走らせることができます。

各 CSP サーバプロセスは、ステートレスのリクエストを処理する際には、多数の異なるクライアントからのリクエストがサポート可能です。

ステートプリザーブモードでは、1 クライアントからのリクエストのみ処理します。

CSP サーバのイベントフロー

CSP サーバは、CSP ゲートウェイからのリクエストを受け取ると、そのリクエストが静的なページなのか CSP クラスなのかを判断します。

静的なページなら、すぐにそのページを戻します。CSP クラスだった場合は、次のことを行います。

- そのリクエストが所属するセッションを判断する
- どのセッションにも所属していない場合は、新しいセッションを開始する
- そのリクエストが正しい IRIS ネームスペースで処理されているかどうかを確認する
- 正しい%**CSP.Session** オブジェクトが利用可能ならば、**HTTP** リクエスト上に含まれる情報をもとに%**CSP.Request** オブジェクトのインスタンスを作成する
- 暗号化が必要であれば行う
- アプリケーションがレスポンスヘッダを書き換えられるように%**CSP.Response** オブジェクトを作る
- どのクラスがそのリクエストを処理すべきかを判断し、その **Page** メソッドを起動する（そこから **OnPage** コールバックメソッドが呼ばれる）

CSP サーバ URL とクラス名解決

CSP サーバは、URL を解釈して、HTTP リクエストをディスパッチするクラスを決定します。

CSP は URL を表 6-1 のようにコンポーネントに分解します。

URL: `http://webserver:[<port_no>]/csp/user/object.csp?OBJID=2`

表 6-1 : CSP によってコンポーネントに分解された URL

コンポーネント	目的
<code>http://</code>	プロトコル
<code>webserver</code>	Web サーバのアドレス
<code>[<port_no>]</code>	(オプション) Web サーバが動作するポート番号、既定値は 80
<code>/csp/samples/</code>	ディレクトリ
<code>object.csp</code>	ファイル名と拡張子
<code>?OBJID=2</code>	クエリ

このうち、プロトコルと Web サーバのアドレスは、Web サーバによって取り扱われ、CSP サーバとは関係ありません。

ディレクトリは、その URL がどの CSP アプリケーションを参照するかを決めるために使用します。

この URL のディレクトリ部分で識別される CSP アプリケーションは、複数定義することができます。

CSP アプリケーションは、システム管理ポータル の CSP アプリケーションページ で作成、変更が可能です。

リクエストを担当するクラスの名前は、次のアルゴリズムを使って、ファイル名から決定します。

- ファイル拡張子が「.cls」ならば、ファイル名をクラス名として使用する
- ファイル拡張子が「.csp」ならば、csp をパッケージ名とし（既定の動作）、ファイル名をクラス名として使用する
- クラスが存在しない場合または期限切れの場合は、CSP コンパイラが CSP ソースファイルからクラスを生成する

例えば、次のような URL があるとします。

`http://localhost:52773/csp/user/menu.csp`

この場合、「/csp/user」ディレクトリに関連する IRIS ネームスペース内で稼働している「csp」パッケージに含まれている「menu」というクラスに割り当てられます。

%CSP.Page クラス

CSP サーバ上では、すべての HTTP リクエストは、%CSP.Page クラスに定義されたメソッドを起動することによって処理します。

%CSP.Page クラスは、リクエストそれ自身を直接操作することはありません。

それは、単に HTTP リクエストを処理するために必要なインターフェイスを定義するだけです。

実際のイベント操作は、%CSP.Page のサブクラスによって行います。

%CSP.Page のサブクラスは、決してインスタンス化はされません。

つまり、%CSP.Page オブジェクトは生成されません。

%CSP.Page に定義しているメソッドは、すべてクラスメソッドです。

起動するためにオブジェクトは必要ありません。

これらのメソッドが必要とする状態情報は、CSP サーバが管理する他のオブジェクト (%CSP.Request や %CSP.Session オブジェクト) が提供します。

Page メソッド

CSP サーバは、どの %CSP.Page クラスがリクエストを処理するかを決定した後、適切な処理コンテキストをセットアップし、そのクラスの Page メソッドを起動します。

プロセスコンテキストのセットアップでは、次のことを行います。

- 標準出力デバイス (\$IO) をリダイレクトして、Write コマンドを使用するすべての出力が HTTP クライアントに送り戻されるようにする
- 必要なオブジェクト (%request,%response,%session オブジェクトなど) またはローカル変数を作成する

Page メソッドは、HTTP リクエストに対する完全なレスポンスを、コールバックメソッドを「OnPreHTTP」「OnPage」「OnPostHTTP」の順番に起動することで行います。コールバックメソッドとは、サブクラスが個別の振る舞いを提供するためにそれらを上書きするため、そう呼ばれています。

OnPreHTTP メソッドは、HTTP レスポンスのヘッダを記述します。

これには、コンテンツタイプやクッキーなどの情報が含まれます。

既定の振る舞いは、コンテンツタイプを `text/html` にすることです。

レスポンスヘッダを直接操作する必要があるときには、OnPreHTTP メソッドをオーバーライドする必要があります。

OnPage メソッドは、HTTP リクエストへの返答の一連の作業を実行します。

HTML や XML などのリクエストの内容を書き出します。

例えば、次のような OnPage メソッドを含むサンプルの CSP クラスがあるとします。

```
Class MyApp.Page Extends %CSP.Page
ClassMethod OnPage() As %Status
{
Write "<html>",!
Write "<body>",!
Write "My page",!
Write "</body>",!
Write "</html>",!
Quit $$$OK
}
}
```

OnPostHTTP メソッドは、HTTP リクエストの処理が終わった後で、実行したい操作を実行する場所として提供しています。

%CSP.Page クラスは、コードを記述することなくカスタムな振る舞いを提供するために、オーバーライド可能ないくつかのクラスパラメータを含んでいます。

詳しくは、%CSP.Page のドキュメントを参照してください。

CSP エラーの操作

`%CSP.Error` は、既定の CSP エラーページです。これをスーパークラスとしてエラーページを作成します。

そうすることで、`%CSP.Error` が提供する関数を使って、エラーから情報を引き出すことができます。

既存セッションがある場合、ページに移動しようとして、そのページが見つからなければ、そのセッションはライセンスを持っているので、CSP は標準エラーを表示します。CSP アプリケーションがまだライセンスを持っていない場合にエラーが発生すると、既定では、「標準 Web HTTP/1.1 404 ページが見つかりません」というエラーメッセージが表示されます。

エラーページ上のパラメータを設定すれば、発生したエラーに対して、表示されるページが変更できます。以降で、各種パラメータについて説明します。

「LICENSEERRORPAGE」パラメータ

次のようなライセンスがないというエラーが発生したときには、CSP は「LICENSEERRORPAGE」パラメータの値を参照します。

Cannot grant license.

LICENSEERRORPAGE は、次の 2 つの値を持つことができます。

- "" ? 「HTTP/1.1 404 ページが見つかりません」エラーを返す（既定値）
- （静的ページのパス）? : 「/csp/user/static.html」などの指定の静的ページを表示する

「PAGENOTFOUNDEORRORPAGE」パラメータ

次のようなエラーが発生したとき、CSP は「PAGENOTFOUNDEORRORPAGE」パラメータの値を参照します。

Class does not exist
Method does not exist

CSP application does not exist (set parameter on default error page)
CSP page does not exist
File does not exist

CSP namespace does not exist
CSP illegal request
File cannot open
CSP session timeout

PAGENOTFOUNDEORRORPAGE は、次の 3 つの値を持つことができます。

- "" ? : 「HTTP/1.1 404 ページが見つかりません」エラーを返す（既定値）
- 1 ? : ライセンスを取得して、標準エラーメッセージを表示する
- （静的ページのパス） ? : 「/csp/user/static.html」などの指定の静的ページを表示する

「OTHERSTATICERRORPAGE」パラメータ

その他のエラーが発生したとき、CSP は「OTHERSTATICERRORPAGE」パラメータの値を参照します。

OTHERSTATICERRORPAGE は、次の 3 つの値を持つことができます。

- "" ? : ライセンスを取得して、標準エラーメッセージを表示する（既定値）
- 1 ? : 「HTTP/1.1 404 ページが見つかりません」エラーを返す。ライセンスは必要ない
- （静的ページのパス） ? : 「/csp/user/static.html」などの指定の静的ページを表示する

%CSP.Request オブジェクト

CSP サーバが HTTP リクエストに応答するときは、要求に関する情報を %CSP.Request オブジェクトのインスタンスにまとめます。

このオブジェクトは、%request 変数を使って参照できます。

URL プロパティ

HTTP リクエストの URL を見つけるには、%CSP.Request オブジェクトの URL プロパティを使います。

`Write "URL: ", %request.URL`

Data プロパティと URL パラメータ

URL はパラメータのリストを含むことができます（URL クエリと呼ぶこともあります）。

%CSP.Request オブジェクトは、その Data プロパティを通してそれらを利用可能にします。

例えば、入ってくる URL が次の内容を含んでいるとします。


```
/csp/user/MyPage.csp?A=10&a=20&B=30&B=40
```

サーバ側では、これらのパラメータは、次のように取得できます。

```
Write %request.Data("A",1) // これは 10 です  
Write %request.Data("a",1) // これは 20 です  
Write %request.Data("B",1) // これは 30 です  
Write %request.Data("B",2) // これは 40 です
```

Data は、多次元プロパティであり、2つのサブスクリプトの中にパラメータ名とそのパラメータのインデックス番号が格納されます（パラメータは、上記コードの B 例のように URL 内で複数発生可能です）。

なお、パラメータ名は、大文字小文字の区別があります。

HTTP リクエストが GET リクエストでも POST リクエストでも関係ありません。

Data プロパティは、パラメータ値をそのまま表現します。

ObjectScript の \$Data 関数を使って、あるパラメータ値が定義されているかどうかテストすることができます。

```
If ($Data(%request.Data("parm",1))) {  
}
```

パラメータを参照したいけれども定義されているかどうかわからない場合には、ObjectScript の \$Get 関数を使うことができます。

```
Write $Get(%request.Data("parm",1))
```

%CSP.Request オブジェクトの Count メソッドを使って、あるパラメータ名に何個の値が定義されているかを見つけることができます。

```
For i = 1:1:%request.Count("parm") {  
  Write %request.Data("parm",i)  
}
```

CgiEnvs プロパティと CGI 環境変数

Web サーバは、CGI 環境変数と呼ばれる一連の値を提供します。

それは、HTTP クライアントや Web サーバに関する情報を含みます。

CgiEnvs 多次元プロパティを使えば、それらの CGI 環境変数にアクセスできます。

例えば、どのブラウザのタイプが HTTP リクエストを行ったかを調べるためには、CGI 環境変数 HTTP_USER_AGENT の値を参照します。

```
Write %request.CgiEnvs("HTTP_USER_AGENT")
```

Cookies プロパティ

HTTP リクエストにクッキーが含まれるときには、多次元プロパティである Cookies を使って、それらの値を取得することができます。

MIME データ プロパティ

リクエストには、MIME データを含むことができます。

%CSP.Request オブジェクトを使って、MIME データを取得できます。

そうすることで、IRIS ストリームオブジェクトのインスタンスが戻され、それを使って MIME データを読むことができます。

%CSP.Response オブジェクトと OnPreHTTP メソッド

%CSP.Response オブジェクトを使って、HTTP クライアントにどのレスポンスヘッダを戻すかを制御できます。

CSP サーバは、自動的にこのオブジェクトのインスタンスを作成し、変数%response にその参照を設定します。

%response オブジェクトは、HTTP ヘッダを制御するので、そのプロパティは、%CSP.Page クラスの OnPreHTTP メソッド内でセットします。

例えば、HTTP リクエストをリダイレクトするには、次のように OnPreHTTP メソッドを定義します。

```
Class MyApp.Page Extends %CSP.Page
{
ClassMethod OnPreHTTP() As %Boolean
{
Set%response.ServerSideRedirect= "C:¥InterSystems¥IRIS¥csp¥user¥redirect.csp"
Quit 1
}
}
```

SetCookie メソッドでクッキーを処理する

%response オブジェクトの SetCookies メソッドを使って、HTTP クライアントにクッキーを送ることができます。

異なるコンテンツタイプを処理する

通常は、CSP ページは、text/html コンテンツを処理します。いくつかの方法で、異なるコンテンツタイプを指定できます。

%CSP.Page クラスパラメータ CONTENTTYPE の値を設定する

ページの OnPreHTTP メソッド内で、%response オブジェクトの ContentType プロパティの値を設定する

6-6 CSP セッション管理

セッションというのは、一定期間内でのある特定のクライアントから特定のアプリケーションへの一連のリクエストを表します。

CSP.Session によるセッション

CSP は、セッションの追跡を自動的に行います。

CSP アプリケーションは、`%CSP.Session` オブジェクトによって、セッションのさまざまな側面を問い合わせたり、変更したりできます。

CSP サーバは、ObjectScript の `%session` 変数を経由して、オブジェクトを利用可能にします。

セッションの作成

セッションは、HTTP クライアントが CSP アプリケーションに対して最初のリクエストを行ったときに開始されます。

新しいセッションが作成されると、CSP サーバは、次のことを行います。

- 新しいセッション ID 番号の作成
- ライセンスチェックの実行
- `%CSP.Session` オブジェクトの新しいインスタンスの作成（永続的）
- （存在すれば）現セッションイベントクラスの `OnSessionStart` メソッドの起動
- セッション中の HTTP クライアントからのリクエストを追跡するためのセッションクッキーの作成

クライアントブラウザがクッキーを無効にしているときには、セッション追跡用に、CSP は自動的に URL リライティング（URL ごとに特別な値を置く）を使います。

セッションの最初のリクエストでは、`%CSP.Session` オブジェクトの `NewSession` プロパティに 1 がセットされます。

その後のリクエストでは、それは 0 に設定されます。

```
If (%session.NewSession = 1) {  
  // これは新しいセッション  
}
```

セッション ID

セッション ID は、特定のセッションを識別するために使用するユニークな値です。

セッション ID は、永続 `%CSP.Session` オブジェクトのオブジェクト識別子として使います。

CSP サーバは、HTTP リクエストを処理する際に、`%CSP.Session` オブジェクトの正しいインスタンスとその他のプロセスコンテキストが利用可能かどうかを確認するために、そのセッション ID を使います。

CSP アプリケーションは、`%CSP.Session` オブジェクトの `SessionId` プロパティによって、特定のセッション ID を見つけることができます。

```
Write "Session ID is: ", %session.SessionId
```

セッション終了とクリーンアップ

セッションは、次のいずれかの理由によって終了します。

- 指定されたセッションタイムアウトの期間、要求を受け付けなかったため、セッションがタイムアウトした
- セッションが、サーバ側で明示的およびプログラマ的に終了した (`%CSP.Session` オブジェクトの `EndSession` プロパティに 1 を設定する)

セッションが終了するとき、CSP サーバは、永続 `%CSP.Session` オブジェクトを削除し、セッションライセンスカウントを減算します。

タイムアウトやサーバアクションによってセッションが終了すると、セッションイベントクラスの `OnSessionEnd` メソッドを起動します。

%CSP.Session オブジェクト

%CSP.Session オブジェクトは、現セッションに関する情報を含むとともに、セッションをプログラマ的にコントロールする方法も含んでいます。

ユーザセッションデータの Data プロパティ

Data プロパティを使って、%CSP.Session オブジェクト内にアプリケーション特有の情報を格納することができます。

Data は、多次元配列プロパティで、1 つの多次元配列内に特定情報のかたまりを関連付けます。

この配列の内容は、セッションの生存期間中、自動的に維持管理されます。

%CSP.Session オブジェクトの Data プロパティは、他の ObjectScript 多次元配列と同じように使うことができます。

例えば、OnPage メソッド内で次のコードが実行されるとします。

```
Set %session.Data("MyData") = 22
```

その同じセッションへのその後のリクエスト（どのクラスがそのリクエストを処理するかに関わらず）には、%CSP.Session オブジェクト内に次の値があります。

```
Write $Get(%session.Data("MyData"))
```

ユーザセッションデータをセットする Set コマンド

%CSP.Session オブジェクトのデータに格納するには、Set コマンドを使用します。

```
Set %session.Data("MyData") = "hello"  
Set %session.Data("MyData",1) = 42
```

ユーザセッションデータを取得する Write コマンド

ObjectScript の表現の一部として Data プロパティからデータを取得できます。

```
Write %session.Data("MyData")  
Write %session.Data("MyData",1) * 5
```

Data 配列の未定義ノードを参照すると、<UNDEFINED> (未定義) エラーが実行時に発生します。

これを避けるには、ObjectScript の \$Get 関数を使用します。

```
Write $Get(%session.Data(1,1,1)) // 値を返すか""を返す
```

ユーザセッションデータを削除する Kill コマンド

Data プロパティからデータを削除するには、ObjectScript の Kill コマンドを使います。

```
Kill %session.Data("MyData")
```


セッションタイムアウト

CSP セッションは、クライアントからのリクエストを受けてから、どれだけの時間が経過したかを自動的に追跡しています。

その経過時間が、ある閾値を超えると、セッションは自動的にタイムアウトします。

既定では、そのセッションタイムアウトは、900 秒（15 分）です。

システム管理ポータルで CSP アプリケーションごとにこの既定値を変更することができます。

%CSP.Session オブジェクトの AppTimeout プロパティの値をセットすることで、アプリケーション内でその既定値をセットすることもできます。

```
Set %session.AppTimeout = 3600 // timeout を 1 時間に設定する
```

セッションタイムアウトを無効にするには、そのタイムアウト値を 0 にセットします。

タイムアウトの通知 OnTimeout メソッド

CSP アプリケーションのタイムアウトが起こると、その%CSP.Page クラスの OnTimeout メソッドを起動することによって、CSP サーバはそのアプリケーションに通知します。

既定では、イベントクラスは定義されていないので、タイムアウトは単純に現セッションを終了するだけです。

状態管理

HTTP は、ステートレスプロトコルなので、Web 用に書いたアプリケーションは、アプリケーションのコンテキストあるいは状態を管理する特別なテクニックを使わなければなりません。

CSP は、状態管理のためのいくつかのメカニズムを提供しています。

リクエスト間のデータを追跡する

Web アプリケーションの状態管理の基本的な問題は、連続する HTTP リクエスト間の情報の保持です。

そのためには、次のいくつかの方法が利用可能です。

- hidden フォームフィールドまたは URL パラメータのいずれかを使って、各ページにデータを格納する
- クライアントのクッキーにデータを格納する
- サーバの `%CSP.Session` オブジェクトにデータを格納する
- IRIS データベース内にデータを格納する

ページ内にデータを格納する

ページ内に状態情報を格納するためには、そのページの次のリクエストが情報を含むようにページに配置しなければなりません。

そのページがハイパーリンクを使ってリクエストを行った場合、ハイパーリンクの URL 内にそのデータがなければなりません。

例えば、次の例では、CSP ファイルに定義された状態情報を含むハイパーリンクがあります。

```
<a href="page2.csp?DATA=#(data)#">Page 2</A>
```

CSP がこのリンクを含むページを処理するときに、表現「 `#(data) #`」はクライアントに送るテキスト内でサーバ変数データに置き換えられます。

ユーザが `page2.csp` へのリンクを選ぶと、CSP サーバは `%request` オブジェクトを通して `DATA` の値にアクセスします。

必要ならば、CSP は、そのようなデータをエンコードすることができます。

また、ページにフォームがあるときには、**hidden** フィールドに状態情報を置くことができます。

```
<form>
<input type="HIDDEN" name="DATA" value="#(data)#">
<input type="SUBMIT">
</form>
```

ハイパーリンクの例と同様に、このフォームがクライアントに送られると、表現「**#(data)#**」は、その変数データの値で置き換えられます。

ユーザがこのフォームをサブミットすると、**DATA** の値は **%request** オブジェクトを通して利用可能になります。

クッキーにデータを格納する

その他のテクニックは、クッキーの中に状態情報を置くことです。

クッキーは、クライアントの中に格納される「名前- 値」のペアです。

クライアントからのリクエストごとに、以前のページのクッキーの値を含んでいます。

クッキーの値をセットするには、%CSP.Response オブジェクト内のページクッキー値を、次のようにオーバーライドします。

```
Class MyApp.Page Extends %CSP.Page
{
  ClassMethod OnPreHTTP() As %Boolean
  {
    Do %response.SetCookie("UserName",name)
    Quit 1
  }
}
```

サーバは、%CSP.Request オブジェクトの Cookies プロパティを使ってこの情報を取得することができます。

セッションにデータを格納する Data プロパティ

Data プロパティを使って、セッションの%CSP.Session オブジェクトに状態情報を格納できます。

%session オブジェクトに置かれた情報は、現セッションの終了（あるいは明示的に削除する）まで利用できます。

データベースにデータを格納する

ユーザに関係するより複雑な情報を保持したい場合には、おそらく IRIS データベースにデータを持つのが最善でしょう。

その 1 つの方法は、1 つ以上の永続クラスを定義して、そのオブジェクト ID 値を `%session` オブジェクトの中に格納し、その後のアクセスに使用します。

サーバコンテキスト保存 Preserve プロパティ

リクエストから次のリクエストの間で、CSP サーバで保護されるプロセスコンテキストは、通常は `%session` オブジェクトの中だけに保持されます。

CSP サーバは、完全なプロセスコンテキスト変数、インスタンス化したオブジェクト、データベースロック、オープンしたデバイスを、リクエスト間で保持するメカニズムを提供します。

これを、コンテキストプリザービングモードと言います。

CSP アプリケーションの中で、コンテキスト保護をオンまたはオフにすることができます (`%CSP.Session` オブジェクトの **Preserve** プロパティ)。

ただし、プロセスを 1 つのセッションに固定することは、スケーラビリティの低下につながる可能性がある点に注意が必要です。

認証と暗号化

HTTP クライアントに送られたページ上に状態情報を置くことは、かなり一般的に行われています。

それらのページから引き続きリクエストがあるときには、その状態情報はサーバに戻されます。

そこで重要なのは、状態情報は Web ページに次の 2 つの点を考慮しなければならないことです。

- HTTP ソースを参照する人が、その状態情報の値をわからないようにする
- サーバは、戻ってきた情報が同じサーバとセッションから送られてきたことを確かめることができる

暗号化サービスを経由して、CSP は、これを実現する簡便な方法を提供しています。

セッションキー

CSP は、暗号キーを使って、サーバ上のデータの暗号化および復号ができます。

CSP セッションはユニークなセッションキーを持っているため、それをセッション用データの暗号化に使います。

セッションキーは、HTTP クライアントには送られないので、このメカニズムは安全です。

それは、`%CSP.Session` オブジェクトの一部として CSP サーバ上に残ります。

暗号化された URL と CSPToken

次に示すような状況下では、`.csp` ファイルから生成されるクラスは、クライアントに送る URL 値を自動的に暗号化します。

例えば、次のような、`.csp` ファイルが他ページへのリンクを定義しているアンカータグを含む URL が暗号化されます。

```
<a href="page2.csp?PI=314159">Page 2</a>
```

次のようなものがクライアントに送られます。

```
<a href="page2.csp?CSPToken=8762KJH987JLJ">Page 2</a>
```

ユーザがこのリンクを選ぶと、暗号化されたパラメータ `CSPToken` が CSP サーバに送られます。

サーバは、それを復号し、復号された内容を `%request` オブジェクトに置きます。

暗号化された値が修正されたり他セッションから送られたりしたならば、サーバはエラーを返します。

パラメータ値が、最初から暗号化されているかいないかは、`%CSP.Request` クラスの `IsEncrypted` メソッドで決めることができます。

CSP コンパイラは、HTML ドキュメント内で URL を記述できるすべての場所を自動的に検出できるため、必要に応じて暗号化を実施します。

プログラミングによってページを作成しようとするときには、%CSP.Page クラスの Link メソッドを使って、同じ動作を得ることができます。

関数への引数としてリンクをパスしようとしているときには、「#url0#」ディレクティブではなく、いつも %CSP.Page の Link メソッドを使ってください。

```
window.showModalDialog('#(..Link("locks.csp"))#',"windowFeatures);
```

次のように、「#url0#」ディレクティブを使った例は、動作しません。

```
window.showModalDialog('#url(locks.csp)#',"windowFeatures);
```

CSP コンパイラが検知できない場所に暗号化 URL を提供する必要があるときには、「#url0#」ディレクティブを使ってください。

例えば、クライアント側の Javascript 関数内でリンクがパラメータならば、次のように記述します。

```
<script language=JavaScript>
function NextPage()
{
// 次のページへジャンプ
CSPPage.document.location = '#url(nextpage.csp)#';
}
</script>
```

プライベートページ

CSP は、プライベートページという概念を提供します。

プライベートページは、同じ CSP セッション内の他ページからのみナビゲート可能です。

プライベートページは、アプリケーションのあるページへのアクセスを制限したい場合に便利です。

例えば、`private.csp` というプライベートページがあるとする、ユーザは、直接その `private.csp` にナビゲートできず、他の CSP ページの中に含まれるリンクからのみ `private.csp` にナビゲートできます。

なお、参照している CSP ページに含まれるリンクは、`http://` で始まる絶対 URL にはできません。

参照しているページからの相対的なパスだけが、そのプライベートページのメソッドによって暗号化（トークン化）されます。

つまり、最初の 2 つのリンクが同じトークンを、ターゲットのプライベートページ「`test2.csp`」にパスします。

```
<A HREF='test2.csp'>プライベートページへのリンク - 相対パス</A> <BR>  
<A HREF='/csp/user/test2.csp'>
```

プライベートページへのリンク - 完全アプリケーションパス full

しかし、次のリンクは違った形でハッシュされるため、アクセスできません。

```
<A HREF='http://myserver/csp/user/test2.csp'>
```

プライベートページへのリンク - 絶対パス

暗号化されたトークンは、現セッションのみに有効なので、このプライベートページをブックマークすることも不可能です。

暗号化した URL パラメータ

プライベートページと同じような方法で、%CSP.Page クラスパラメータ ENCODED の値を設定して、CSP ページの URL パラメータを暗号化するようにも指定できます。

- ENCODED=0：クエリパラメータは、暗号化されない
- ENCODED=1：クエリパラメータは、暗号化され、CSPToken 内に渡される
- ENCODED=2：暗号化されていないパラメータは、%request オブジェクトから取り除かれる点を除いては ENCODED=1 と同じ

ENCODE=1 の例を挙げてみます。サンプルページの `protected.csp` と `protectedentry.csp` が、ENCODED=1 の例を示しています。

`protected.csp` のソースは、暗号化されていない URL に何かが追加されたかどうかをチェックする様子を明らかにしています。

暗号化されていないものがあると、ページはスクロール可能なマーカを表示して、**HACKER ALERT!**と表示します。

この様子確かめるためには、CSP サンプルページを表示して、次の手順を踏みます。

- ① `protectedentry.csp` をダブルクリックする
- ② [BALANCE] フィールドに 500 を入力する
- ③ [View Balance] をクリックする
- ④ `protected.csp` ページが表示されて、Your Account Balance is: 500 と表示される
- ⑤ URL には暗号化された CSPToken（すべて CSPToken=の後ろに）が表示される（これは暗号化された 500）
- ⑥ この URL の最後に&BALANCE=8000 と入力して、enter を押す
- ⑦ `protected.csp` ページが表示される

その暗号化されていない URL への追加部分を受け付けて、**HACKER ALERT!**マーカが表示されます。

もし、ENCODE が 2 に設定されていると、その暗号化されていない入力は、無視されます。

6-7 CSP によるタグを使った開発

開発者は、標準の HTML ファイルを使って CSP アプリケーションを開発します。

CSP コンパイラは、HTML と XML ドキュメントを %CSP.Page クラスに変換して、HTTP リクエストに応答することができます。

CSP コンパイラが生成したクラスは、自分で作成するクラスと何ら違いはなく、完全に相互運用可能です。

CSP コンパイラが処理した HTML ドキュメントは、タグを含みます。

そのタグは、クラス生成を制御するもの、制御フローを提供するもの、データアクセスを管理するもの、サーバ側の振る舞いを制御するもの、などが考えられます。

これらのタグは、CSP マークアップ言語または CSP タグで、CSP サーバで開発時に解釈されます。

CSP で HTTP クライアントに送られる HTML は、完全な標準的なもので CSP タグを含みません。

CSP ファイル内では、普通の HTML タグに加えて次のものを使うことができます。

- ページ生成時に値を置換する #0# を使った IRIS データ表現
- IRIS CSP タグ「<csp:xxx>」：組み込みカスタム機能
- IRIS スクリプト「<script language=cache runat=server/compiler>」：ページ生成時またはコンパイル時の IRIS コード実行
- IRIS メソッド：ページ内で起動、再利用可能なクラスメソッド
- サーバ側サブルーチンコール「#server0#と#call0#」：クライアント側からサーバ側サブルーチンを起動する（ハイパーイベント）
- カスタムタグ

CSP コンパイラ

CSP コンパイラは、IRIS サーバ上の IRIS クラスとプログラムの集合で、次のことを行います。

- CSP マークアップ言語を使った HTML ドキュメントを読んで、解釈する
- CSP ルールに基づくパターンマッチングロジックを適用する
- IRIS クラスを生成する
- そのクラスを実行可能コードにコンパイルする

例えば、次のような簡単な CSP ドキュメント「hello.csp」がコンパイルされたとします。

```
<html>  
<body>
```

こんにちは!

```
</body>  
</html>
```

CSP コンパイラは、これを次のようなクラスに変換します。

```
Class csp.hello extends %CSP.Page  
{  
  ClassMethod OnPage() As %Status  
  {  
    Write "<html>"  
    Write "<body>"  
    Write "こんにちは!"  
    Write "</body>"  
    Write "</html>"  
    Quit $$$OK  
  }  
}
```

ユーザがブラウザから hello.csp を要求すると、CSP サーバは、その生成された OnPage メソッドを起動し、CSP ドキュメントのオリジナルテキストが表示用にブラウザに送られます。

自動と手動ページコンパイル

CSP サーバは、CSP ソースドキュメントを自動的にまたは手動でコンパイルすることができます。

自動コンパイルがオンのとき（既定値）、CSP サーバは、自動的に CSP コンパイラに CSP ソースドキュメントを必要に応じてクラスに変換するよう依頼します。

CSP サーバは、ソースファイルのタイムスタンプとクラスのタイムスタンプを比較し、ソースのほうがクラスよりも新しかったら、ページを再コンパイルします。

一般的には、アプリケーション運用時には、そのタイムスタンプをチェックするオーバーヘッドを避けるためにこのモードをオフにします。

自動コンパイルをオフにするには、次のようにします。

① システム管理ポータルで [セキュリティ管理] をクリックし、「アプリケーション定義」にある [CSP アプリケーション] をクリック（図 6-12）

図 6-12：システム管理ポータルの「アプリケーション定義」ペイン

② 自動コンパイルをオフにしたいアプリケーションの [編集] をクリック（図 6-13）

図 6-13：CSP アプリケーション一覧画面

③ CSP アプリケーションの編集で、自動コンパイルを「いいえ」にする（図 6-14）

図 6-14：CSP アプリケーション編集画面

CSP マークアップ言語

CSP マークアップ言語は、CSP コンパイラが生成するクラスを制御するために使用可能なディレクティブとタグの集合です。

CSP ドキュメントをコンパイルすると、その結果は 1 つの IRIS クラスとなり、そのクラスは ObjectScript を実行します。

CSP ページ言語

テキスト

CSP ドキュメント (HTML または XML) に含まれるテキストで、CSP ディレクティブや特殊タグでないものは、ページを要求した HTTP クライアントにそのまま変更せずに送られます。

例えば、次の内容を含む CSP ドキュメントがあるとします。

```
<b>こんにちは!</b>
```

生成されたコード内には、次のコードが生成されます。

```
Write "<b>こんにちは!</b>";!
```

その後、次の内容が HTTP クライアントに送られます。

```
<b>こんにちは!</b>
```

コンパイル時表現とコンパイル時コード

実行時ではなく、CSP のコンパイル時に評価される表現を指定することもできます。

そのような表現は、CSP ルール定義によく使われます。

コンパイル時表現は、「**##(expr)##**」ディレクティブを使って区切られます。

ここでいう **expr** は、ObjectScript 表現です。

例えば、次の内容を含む CSP ドキュメントがあるとします。

```
このページは、<b>##($ZDATETIME($H,3))##</b>にコンパイルされました。
```

生成されたクラス内には、次のコードが生成されます。

```
Write "このページは、<b>2000-08-10 10:22:22</b>に生成されました",!
```

次のように、<script>タグの runat 属性を使っても、ページコンパイル時に実行されるコード行を定義することができます。

```
<script language="Cache" runat="compiler">
```

実行時表現

CSP ドキュメントは、ページが処理されたとき（つまり実行時に）に CSP サーバ上で実行される表現を含むことができます。

そのような表現は、「#(expr)#」ディレクティブを使って区切られます。

ここでいう expr は、正しい ObjectScript 表現です。

例えば、次の内容を含む CSP ドキュメントがあるとします。

```
2 足す 2 は、<b>#(2 + 2)#</b>
```

生成されたクラス内には、次のコードが生成されます。

```
Write "2 足す 2 は、<b>", (2 + 2), "</b>",!
```

その後、次の内容が HTTP クライアントに送られます。

```
2 足す 2 は、<b>4</b>
```

実行時表現の例を挙げてみましょう。

- ページ上の以前にセットされた変数の値

答えは、

```
<b>#(answer)#</b>
```

- オブジェクトプロパティやメソッド

お客様の残高は、

```
<b>#(account.Balance)#</b>
```

です。

- %ResultSet オブジェクトのフィールド

```
<table>  
<csp:while condition="result.Next()">  
<tr><td>#(result.Get("BookTitle"))#</td></tr>  
</csp:while>  
</table>
```

● %request オブジェクトを使った URL パラメータ

```
<table bgcolor='#(%request.Data("tablecolor",1))#></table>
```

実行時表現の値が、<や>などの特別な文字を含む場合は、正しいエスケープシーケンスを HTTP クライアントに送るために、それらをエスケープしなければなりません。

%CSP.Page が提供するエスケープメソッドを使います。

次の例は、EscapeHTML クラスメソッドを表しています。

説明:

```
<b>#(..EscapeHTML(object.Description))#</b>
```

実行時表現が HTML 属性値として使われているときには、実行時表現に見つかる任意の HTML エンティティは、実行コードに変換される前にそれらが表現する文字に変換されます。

例えば次のようなコードがあるとします。

```
<font size=#(1 &gt; 0)#>
```

これは、生成コード内に次のコードを生成します。

```
Write "<font size=", (1 > 0), ">", !
```

実行時コードページ内で、CSP サーバ上で実行する簡単な表現以上のものを処理したい場合には、<script runat=server>タグを使えば、CSP サーバ上で稼働するコードを置くことができます。

例えば、次のような CSP ドキュメントがあるとします。

```
<ul>  
<script language="cache" runat=server>  
For i = 1:1:4 {  
Write "<li>項目 ",i,!  
}  
</script>  
</ul>
```

生成クラス内には、次のコードが生成されます。

```
Write "<ul>",!  
For i = 1:1:4 {  
Write "<li>項目 ",i,!  
}  
Write "</ul>",!
```

その結果として、次の内容が HTTP クライアントに送信されます。

```
<ul>  
<li>項目 1  
<li>項目 2  
<li>項目 3  
<li>項目 4  
</ul>
```

実行時コード ObjectScript 単一行

1 行の ObjectScript を実行するためには、次のような構文を使うことができます。

ただし、行はラップすることはできません。

```
#[ set x = a + b write x ]#
```

サーバ側メソッド

CSP ドキュメント内でクラスに属するメソッドを定義できます。

これには、引数付き<script>タグを使います。

メソッド名とその引数および戻り値が指定できます。また、メソッドを実装する言語も指定できます。

例えば、次のコードでは、count 数分の項目を含む順序リストを作成する MakeList というメソッドを定義しています。

```
<script language="Cache" method="MakeList" arguments="count:%Integer"
returntype="%String">
New i
Write "<ol>",!
For i = 1:1:count {
Write "<li> 項目",i,!
}
Write "</ol>",!
Quit ""
</script>
```

このコードは、次のようにして、CSP ドキュメントの他の場所から起動することができます。

```
<hr>  
#(..MakeList(100))#
```

また、次のように<csp:class>タグを使って、以前に定義したメソッドを継承することもできます。

```
<hr>  
#(##class(MyApp.Utilities).MakeList(100))#
```

SQL <script>タグ

<script>タグを使って、IRIS %ResultSet オブジェクトを定義し、SQL を使うことができます。

次の例では、query という名前の動的 SQL %ResultSet を作成し、指定の SQL クエリを準備して実行しています（繰り返し処理の準備を行います）。

```
<script language="SQL" name="query">  
SELECT Name FROM MyApp.Employee ORDER BY Name  
</script>
```

一般的には、その SQL スクリプトタグで作成した %ResultSet オブジェクトと <csp:while>タグを組み合わせて、クエリの結果を表示します。

SQL<script>タグは、ページ実行が終了した時点で、そのインスタンス化された %ResultSet オブジェクトをクローズします。

SQL テキストの中に ? 文字を使って、SQL クエリのパラメータを指定できます。

そして、SQL <script>タグの P1,P2,...Pn 属性を使い、そのパラメータの値を提供できます。

次の例は、現ユーザが行った購入を表示するために SQL `<script>` タグを使ったコードです（%session オブジェクトに現ユーザのユーザ ID が格納されているという前提です）。

```
<script language=SQL name=query P1='%session.Data("UserID")'>
SELECT DateOfPurchase,ItemName,Price
FROM MyApp.Purchases
WHERE UserID = ?
ORDER BY DateOfPurchase
</script>
<hr>
商品購入者: <b>#(%session.Data("UserID"))#</b>
<br>
<table>
<tr><th>日付</th><th>商品名</th><th>価格</th></tr>
<csp:while condition="query.Next()">
<tr>
<td>#(..EscapeHTML(query.GetData(1)))#</td>
<td>#(..EscapeHTML(query.GetData(2)))#</td>
<td>#(..EscapeHTML(query.GetData(3)))#</td>
</tr>
</csp:while>
</table>
```

`<csp:query>` タグを使って、IRIS クラスのクエリを使うこともできます。

```
<csp:query NAME="query" CLASSNAME="Sample.Person"
QUERYNAME="ByName">
```

結果の %ResultSet オブジェクトは、SQL `<script>` タグの結果と同様に使うことができます。

生成クラスを制御する

`<csp:class>`タグを使って、CSP コンパイラが生成するクラスに対していくつかの制御を行うことができます。

これは、クラスのスーパークラスを選択することと`%CSP.Page` クラスパラメータの値を定義することを含みます。

例えば、`%CSP.Page` クラスを継承するとともに、他のクラスを継承したい場合を想定しましょう。

`SUPER` 属性は、カンマ区切りのクラスリストを受け付けて、それらを生成クラスのスーパークラスとして使用します。

```
<csp:class SUPER="%CSP.Page,MyApp.Utilities">
```

次のコードは、クラスパラメータ値を再定義する例です。

```
<csp:class PRIVATE=1>
```

制御フロー

CSP マークアップ言語は、ページ実行の制御をしやすくするさまざまなタグを提供します。

`<csp:if>` タグ

`<csp:if>` タグは、`<csp:else>` と `<csp:elseif>` タグとともに CSP ページの条件付き出力を定義する方法を提供します。

`<csp:if>` タグには、1 つの属性「`condition`」があり、その値は `ObjectScript` の表現です。

実行時の CSP サーバで評価され、その値が真のときにタグの内容が実行されます。

```
<csp:if condition='user="Jack"'>
ようこそ ジャック!
<csp:elseif condition='user="Jill"'>
ようこそ ジル!
<csp:else>
ようこそ!
</csp:if>
```

<csp:while>タグ

<csp:while>タグは、あるサーバ側の条件が真である限り、CSP ドキュメントのある部分を繰り返し処理する方法を提供します。

<csp:while>タグの **condition** 属性は、ObjectScript の表現を含むことができます。

それは、ページが処理されるときに CSP サーバ上で評価されます。**condition** 値が真 (1) である限り、**csp:while** タグの内容が評価されます。

一般的には、<csp:while>タグは、IRIS %SQL.StatementResult オブジェクトとともに HTML 内に SQL クエリの結果を表示するために使われます。

次の例では、<csp:while>タグの内容、つまりクエリの **Name** カラムの値を書き出すことを、%SQL.StatementResult オブジェクトの **Next** メソッドが偽 (0) の値を返すまで繰り返し実行します。

```
<script language=SQL name=query>
SELECT Name
FROM MyApp.Employee
ORDER BY Name
</script>
<csp:while condition="query.Next0">
#(..EscapeHTML(query.Get("Name")))#<BR>
</csp:while>
```

また、<csp:while>タグの **counter** 属性を使って、カウンタ変数を定義できます。

毎回の繰り返しの始めには、その変数は 0 に初期化され、自動的に 1 増分されます。

```
<table>
<csp:while counter="row" condition="row<5">
<tr><td>これは、行 #(row)#です。</td></tr>
</csp:while>
</table>
```

次のコードは、条件の中で **not** 演算子を使う例です。

この条件は、スペースを含まず、開始と終了のクオートを含みません。

条件には、(**mystr'="QUIT"**)のように括弧を使うこともできます。

```
<csp:while condition=mystr'="QUIT">
// ここにコードを追加する
</csp:while>

<csp:loop>タグを使った順番付きリストの例

<csp:loop>タグは、カウンタ変数と開始値、終了値、増分値を定義し、指定した内容を
繰り返し実行する方法を提供します。

例えば、5 つの項目を含むリストを作成するために、次のように<csp:loop>タグを使う
ことができます。

<ul>
<csp:loop counter="x" FROM="1" TO="5">
<li>項目 #(x)#
</csp:loop>
</ul>
```


HTTP 出力のエスケープ

HTML で使う特別な文字のリテラル表示のためには、エスケープシーケンスを使う必要があります。

例えば、>（右矢印括弧）文字を HTML 内で表示するには、HTML の中では特別な意味を持っているため、文字列「:>」を使ってエスケープしなければなりません。

%CSP.Page クラスは、次のような、2 種類のエスケープと引用のメソッドを提供しています。

- エスケープメソッド：文字列を入力として受け取り、すべての特殊文字を適切なエスケープシーケンスに置換した文字列を返す
- 引用メソッド：文字列を入力として受け取り、（適切な引用符で囲まれた）引用された文字列を返す。

特殊文字はエスケープシーケンスで置換される

各エスケープメソッドには、対応するアンエスケープメソッドがあり、エスケープシーケンスを平文に置き換えます。

EscapeHTML で HTML をエスケープする

%CSP.Page クラスは、対応する HTML エスケープシーケンスで文字列を置換することができます。

例えば、CSP ファイルがサーバ側の変数 **x** の値をブラウザに表示する必要があるときは、**x** の任意の文字は、次のような表現でエスケープできます。

`#(..EscapeHTML(x))#`

x の値が<mytag>ならば、次のようなエスケープされたテキストが HTTP クライアントに送られます。

`<mytag>`

同じように、HTML 属性の値を送るときにもエスケープ処理が必要です。

`<input type="BUTTON" value="#(..EscapeHTML(value))#">`

この `value` の値が `<ABC>` ならば、この結果は、次のようなテキストとして HTTP クライアントに送られます。

ここでは、2つの角括弧と、その文字シーケンスと同等な値である「`<`」と「`>`」に置換されます。

```
<input type="BUTTON" value="&lt;ABC&gt;">
```

結果の HTML 属性値が引用されるようにするには、`#()`ディレクティブの回りに`"`（引用符）を置いてください。

データベースからの出力を HTTP クライアントに送るときに、それをエスケープする習慣を身に付けてください。

例えば、ユーザ名を Web ページに表示する表現を考えてみましょう（`user` は、`Name` プロパティを持つオブジェクトへの参照であるという想定です）。

```
ユーザ名: #(user.Name)#
```

アプリケーションによって、ユーザが名前をデータベースに入力するときには、いたずら好きなユーザが、HTML コマンドを含んだ名前を入力するかもしれません。

もし、HTML エスケープシーケンスなしに HTTP クライアントに書き込まれると、ページは意図しない動作を起こす危険性があります。

```
Set user.Name = "<input type=button onclick=alert('Ha!');>"
```

EscapeURL で URL パラメータをエスケープする

URL 文字列のパラメータ値もエスケープ可能です。

URL には、HTML とは異なるエスケープシーケンスセットを使用します。

%CSP.Page クラスの `EscapeURL` メソッドは、すべての特殊 URL パラメータ値を、対応するエスケープシーケンスに置換します。

例えば、CSP ファイルがサーバ側の値 `x` を URL のパラメータ値として使うならば、`x` の任意の文字は、次のような表現でエスケープできます。

```
<a href="page2?ZOOM=#(..EscapeURL(x))#">リンク</A>
```

`x` の値が、100%ならば、次のようなテキストが HTTP クライアントに送られます (% 文字は%25 にエスケープされます)。

```
<a href="page2?ZOOM=100%25">リンク</A>
```

QuoteJS で JavaScript をエスケープする

%CSP.Page クラスは、すべての特殊文字を、対応する JavaScript エスケープシーケンスに置き換える「#(..QuoteJS(x))#」文字を提供します。

例えば、CSP ファイルがサーバ側の変数 *x* の値を指定してメッセージを表示する JavaScript 関数を定義

すると想定します。

x の値は、JavaScript の引用された文字に変換されます。

```
<script language="JavaScript">
function showMessage()
{
  alert(#(..QuoteJS(x))#);
}
</script>
```

x の値が、「Don't press this button!」ならば、次のようなテキストが HTTP クライアントに送られます。

```
<script language="JavaScript">
function showMessage()
{
  alert('Don¥t press this button!');
}
</script>
```

サーバ側メソッド

CSP は、HTTP クライアントからサーバ側のメソッドを起動するための、次の 2 つのテクニックを提供しています。

● HTTP サブミットメカニズム

● #server（同期）または#call（非同期）のどちらかのハイパーイベントを使う

HTTP サブミットを使うメリットは、クライアント側のプログラミングが単純になり、クライアント側のコンポーネントが必要ない点です。

不利な点は、メソッド呼び出しの後にクライアントによるページの再描画とサーバ側のプログラミングが難しくなることです。

ハイパーイベントを使うと、#server と #call は XMLHttpRequest を使って実装されます。

#call は非同期であるため、Web ページ上でユーザが値を入力しても、ページは即座に更新されず、更新されるときには他のページに移動していても問題はありません。

HTTP サブミットによるサーバ側メソッドの呼び出し

HTTP サブミットを使うときには、要求ページは、ユーザが SUBMIT ボタンを押すたびに毎回再表示されます。

それは、次のように操作することができます。

① SUBMIT ボタンを含む HTML フォームを処理する

```
<form name="MyForm" action="MyPage.csp" method="GET">  
ユーザ名: <input type="TEXT" name="USERNAME"><br>  
<input type="SUBMIT" name="BUTTON1" value="OK">  
</form>
```

ここでは、USERNAME と呼ばれるテキストフィールドと BUTTON1 と呼ばれる SUBMIT ボタンのあるフォームを定義します。

フォームの ACTION 属性には、フォームがサブミットされる URL を指定します。

METHOD 属性には、フォームをサブミットするために、POST と GET のどちらの HTTP プロトコルを使うかを指定します。

② コントロールの値を ACTION 属性に指定してある URL に送る

ユーザが **BUTTON1** をクリックすると、ブラウザは、フォーム内のすべてのコントロールの値を集めてフォームの ACTION 属性に指定してある URL に送ります。

POST または GET のどちらによって送信されたに関わらず、CSP がサブミットされた値を URL パラメータのように処理します。

フォームをサブミットすることは、次の URL を要求することと同等です。

MyPage.csp?USERNAME=Elvis&BUTTON1=OK

これには、SUBMIT ボタンの名前と値が含まれます。複数の SUBMIT ボタンがフォーム上にあるときには、実際に押されたデータボタンだけがその要求に含まれます。

これは、サブミットが発生したときの検出の鍵となります。

② サーバコードによるサブミットボタンのテスト

この例では、MyPage.csp は、サブミットが発生したことを検知します。これは、%request オブジェクトの名前「**BUTTON1**」を調べることで行います。

```
<script language="Cache" runat="SERVER">  
// サブミットボタンのテスト  
If ($Data(%request.Data("BUTTON1",1))) {  
// これはサブミットです。メソッドを呼び出します。  
Do ..MyMethod($Get(%request.Data("USERNAME",1)))  
}  
</script>
```

④ サーバコードによって HTML を返す

サーバ側ロジックを呼び出した後、サーバコードは継続し、ブラウザが表示する HTML を返します。

ハイパーイベント「#server」と「#call」を使ったサーバ側メソッドの呼び出し

ハイパーイベントは、Web ブラウザイベントの CSP 拡張機能で、会話型の Web アプリケーションを作成するための開発テクニックでもあります。

ハイパーイベントを使うと、HTML ページをクライアントに再ロードすることなく、クライアント Web ブラウザのイベントに応答する IRIS サーバのクラスメソッドを実行することができます。

これは、データの検証、検索機能などを行いたいデータベースアプリケーションでは、さまざまなケースでとても便利な機能です。

なお、CSP ページは、ハイパーイベントを開始するために Javascript を使います。

#server を使ってサーバ側メソッドを呼び出す

#server ディレクティブを使ってサーバ側のメソッドを呼び出すことができます。JavaScript が書ける場所であれば、どこでもこのディレクティブを使うことができます。

#server ディレクティブの構文は、次のとおりです。

```
#server(classname.methodname(args,...))#
```

ここでいう classname は、サーバ側 IRIS クラス名で、methodname はそのクラスのメソッド名です。

args は、クライアント側 JavaScript 引数のリストです。

それらは、サーバ側のメソッドに渡されます。例えば、IRIS クラス MyPackage の Test という名前のサーバ側メソッドを呼び出すには、次のようにします。

```
<script language="JavaScript">
function test(value)
{
// サーバ側メソッド Test を起動する
#server(MyPackage.Test(value))#;
}
```

```
</script>
```

CSP コンパイラは、**#server** ディレクティブをそのサーバ側メソッドを呼び出す JavaScript コードに置換します。

ある CSP ページからそれに属するメソッドを **..MethodName** シンタックスを使って起動することができます。

例えば、次のようにします。

```
#server(..MyMethod(arg))#
```

#call を使ってサーバ側メソッドを呼び出す

#call ディレクティブは、**#server** ディレクティブの代替の方法です。

次に説明する重要な例外を除いて、**#call** と **#server** は同等です。

#call は非同期であり、サーバ側のメソッドを起動したときに **#call** は戻り値を待ちません。

その代わりに、アプリケーションはクライアントで必要な操作を実行するためにサーバから送られてくる JavaScript に依存します。

その非同期性のため、**#call** を使って複数の連続呼び出しを行うときには注意が必要です。

前のメソッドが終わる前に **#call** でメソッドを呼ぶと、Web サーバが前のメソッド呼び出しをキャンセルする可能性があります。

#call ディレクティブの構文は、次のとおりです。

```
#call(classname.methodname(args,...))#
```

ここでいう **classname** は、サーバ側 IRIS クラス名で、**methodname** はそのクラスのメソッド名です。

args は、クライアント側 JavaScript 引数のリストです。

それらは、サーバ側のメソッドに渡されます。

例えば、IRIS クラス `MyPackage` の `Test` という名前のサーバ側メソッドを呼び出すには、次のようにします。

```
<script language="JavaScript">
function test(value)
{
// サーバ側メソッド Test を起動する
#call(MyPackage.Test(value))#;
}
</script>
```

CSP コンパイラは、`#call` ディレクティブをそのサーバ側メソッドを呼び出す JavaScript コードに置換します。

ある CSP ページからそれに属するメソッドを `..MethodName` シンタックスを使って起動することができます。

例えば、次のようにします。

```
#call(..MyMethod(arg))#
```

ハイパーイベントの例

データベースに新しい顧客を追加するために使うフォームがあります。

アプリケーションは、顧客の名前が入力されるとすぐに、その顧客がデータベースに存在しないことを確かめます。

次のフォーム定義は、入力の内容が変更されたときに、サーバ側 Find メソッドを呼び出す例です。

```
<form name="Customer" method="POST">  
顧客名:  
<input type="Text" name="CName"  
onChange=#server(..Find(document.Customer.CName.value))# >  
</form>
```

この例では、Find メソッドは同じ CSP ファイルの中で次のように定義できます。

```
<script language="Cache" method="Find" arguments="name:%String">  
// 顧客の名前が存在するかテスト  
// 埋め込み SQL 使用  
New id,SQLCODE  
&sql(SELECT ID INTO :id FROM MyApp.Customer WHERE Name = :name)  
If (SQLCODE = 0) {  
// 顧客が見つかった  
// JavaScript をクライアントに送る  
&js<alert('Customer with name: #(name)# already exists.');}  
</script>
```

このメソッドは、クライアントに実行 JavaScript を送り返すことで交信します。

サーバ側メソッドが起動したときは、それによるすべての主デバイスへの出力は、クライアントに送られます。

つまり、それは、JavaScript 関数に変換され、クライアントページのコンテキスト内で実行されるということです。

例えば、サーバ側のメソッドが、次のようなコードを実行するとします。

```
Write "CSPPage.document.title = '新しい肩書き';"
```

すると、次のような JavaScript がクライアントに送られ、実行されます。

```
CSPPage.document.title = '新しい肩書き';
```

このケースでは、ブラウザに表示されている肩書きが新しい肩書きに変わります。

このように、任意の有効な JavaScript を送り返すことが可能です。

JavaScript の各行の終わりには、キャリッジリターン (!文字を使って) を置く必要があります。

そうしないと、ブラウザはそれを実行できません。

サーバメソッドから JavaScript を送り返すのを簡単にするために、ObjectScript は、「&js<>」ディレクティブを使った埋め込み JavaScript をサポートしています。

これを使うと、ObjectScript の中に JavaScript の行を含むことができます。

その埋め込み JavaScript を含んでいるメソッドをコンパイルすると、「&js<>」ディレクティブの内容は、適切な Write コマンド行に変換されます。

埋め込み JavaScript は、ObjectScript の「#0#」ディレクティブを参照することができます。

例えば、次の内容を含む IRIS メソッドがあるとします。

```
Set count = 10
&js<
for (var i = 0; i < #count; i++) {
alert('これは楽しい!');
}
>
```

これは、次のコードと同等です。

```
Set count = 10
Write "for (var i = 0; i < ", count, "; i++) {",!
Write " alert('これは楽しい!');",!
Write "}",!
```

CSP クラスの中で#server を使う

CSP クラスの中で、ハイパーイベントと JavaScript を使うには、明示的にハイパーイベントブローカを呼ばなければなりません。

次の例のように、終了の<head>タグの上に「#(..HyperEventHead())#」を置きます。

```
Class esl.csptest Extends %CSP.Page [ ProcedureBlock ]
{
  ClassMethod OnPage() As %Status
  {
    &html<<html>
    <head>
    <script language=javascript>
    function onServer()
    {
      alert(#server(..ServerMethod())#);
    }
    </script>
    #(..HyperEventHead())#
    </head>
    <body>
    <input type=button value="ここをクリック" onclick='onServer()' />
    </body>
    </html>>
    Quit $$$OK
  }
  ClassMethod ServerMethod()
  {
    quit "サーバから"
  }
}
```

サーバ側メソッドを使うための Tips

Web ページからサーバ側のメソッドを起動できる能力は強力ですが、アプリケーションでそれを使うときには注意する点もあります。

それは、**#server** シンタックスを使う場合のいくつかの要素についての注意です。

これに注意しないと、非常に遅いアプリケーションとなったり、場合によっては動作しなくなったりする場合があります。

#server または **#call** を使う際に注意すべき基本的なルールは、次の 2 点です。

- Web ページの **onload** イベントの中では、**#server** は決して使わない（IRIS の中で Web ページを生成するときにデータを生成するほうが速くて簡単）

- Web ページの **onunload** イベントの中でも、**#server** は使わない。

ブラウザとサーバ間のラウンドトリップを含む呼び出しは、それ自身コストの高いものなので、極力少ない数の **#server** 呼び出しになるようにする

1 番目のルールの理由は、**onload** イベントは **HTML** ページの読み込みだけが終わった時点で起動されるので、ページ上のアプレットがロードされるのを待たないし、参照ファイルのロードも終わっていないためです。

#server 呼び出しは、ブラウザがロードするアプレットを使う可能性もあり、タイミングによっては、その呼び出しの時点でアプレットがロードされていないことが十分ありえます。

遅いネットワーク接続の場合には、現象が顕著に現れます。

2 番目の理由は、**onload** イベント内で実行する必要がある任意のコードは、ページが生成されるときに行うほうがずっと速いのと簡単であるということです。

例えば、JavaScript 変数の初期値をセットアップするときに、次のように行っているとします。

```
<html>
<head>
<script language="JavaScript">
function LoadEvent()
{
var value=#server(..GetValue())#;
}
</script>
</head>
<body onload=LoadEvent();>
</body>
</html>

<script language="Cache" method="GetValue" returntype="%String">
Quit %session.Data("value")
</script>
```

しかし、この JavaScript 変数の値は、ページを生成したときに `%session.Data("value")` の中にあり、簡単にわかるので、`#server` を呼び出す理由がありません。

そのため、次のように書き換えるほうがずっと効率的です。

```
<html>
<head>
<script language="JavaScript">
function LoadEvent()
{
var value='#(%session.Data("value"))#';
}
</script>
</head>
<body onload=LoadEvent();>
</body>
</html>
```

ドキュメントをロードし、フォーム要素の値を変更するときには、どんなときでも同じトリックを使えます。

つまり、ページが生成されたときにその値を設定するように変更します。

```
<input type="text" name="TextBox" value='#(%request.Get("Value"))#>
```

同様に `onunload` イベント内で `#server` または `#call` を呼ぶことも問題を引き起こす可能性があります。

IRIS から戻ってきた JavaScript が実行されるかどうかの保証はありません。また、ユーザがコンピュータの電源を切っても、`onload` イベントを受け取ることは決してありません。

アプリケーションは、`%session` オブジェクトのタイムアウトを使って、この可能性に対処しなければなりません。

可能な限り#server と#call 呼び出しを少なくする

#server と#call は、両方とも、ブラウザが特別に暗号化されたトークンとともに、ページを要求する HTTP リクエストを発行することによって動作します。

そして、IRIS に呼び出すメソッドの名前を知らせます。

IRIS は、このメソッドを実行して、送り返した出力が、ブラウザ側で JavaScript として実行されます。

さらに、#server 呼び出しは、戻り値を返すこともできます。

これらは、HTTP リクエストを使うため、ネットワークパケットやサーバ上の CPU 性能などの観点からすると、通常の CSP ページと同様に非常に高価なものです。

大量の#server 要求を使うと、個々の要求が IRIS サーバから新しい CSP ページを要求するため、アプリケーションのスケーラビリティを大きく損なうことになります。

#server 呼び出し数を少なくする方法は、まずアプリケーションが本当にその#server 呼び出しが必要なのかを確かめることと、もし必要ならば、#server 呼び出しがサーバ側でたくさんの仕事を行っているかどうか確かめることです。

例えば、次の例は、サーバからの新しい値でフォームを更新する JavaScript のブロックです。

```
<script language="JavaScript">
function UpdateForm()
{
  CSPPage.document.form.Name.value = #server(..workGet("Name",objid))#;
  CSPPage.document.form.Address.value = #server(..workGet("Address",objid))#;
  CSPPage.document.form.DOB.value = #server(..workGet("DOB",objid))#;
}
</script>
```


サーバ側のコードは次のとおりです。

```
<script language="Cache" method="workGet" arguments="type:%String,id:%String"
returntype="%String">
Quit $get(^work(id,type))
</script>
```

この単一の更新は、IRIS サーバから新しい Web ページのために 3 回の呼び出しを行います。

これは、すべての値を一度に更新する 1 つの `#server` 呼び出しに変換できます。

```
<script language="JavaScript">
function UpdateForm()
{
#server(..workGet(objid))#;
}
</script>
```

このメソッドの定義は、次のようになります。

```
<script language="Cache" method="workGet" arguments="id:%String"
returntype="%String">
&js<CSPPage.document.form.Name.value = #($get(^work("Name",objid)))#;
CSPPage.document.form.Address.value = #($get(^work("Address",objid)))#;
CSPPage.document.form.DOB.value = #($get(^work("DOB",objid)))#;>
</script>
```

6-8 データベースアプリケーションの構築

CSP の強力な点は、組み込みのオブジェクトデータベースと直接やりとりをして動的な Web ページを作成できる点です。

つまり、次のような特徴があります。

- オブジェクトへの複雑なリレーショナルデータのマッピングを避ける
- 複雑なミドルウェアが必要ない

CSP は柔軟です。

データを HTML フォームに自動的に結び付ける高レベルのタグを使うことや、オブジェクトを使って直接データにアクセスするサーバ側のスクリプトを書くことまで、さまざまなテクニックを使ってデータベースアプリケーションを構築できます。

ページ上のオブジェクトを使う

IRIS では、アプリケーションのデータを表現する永続オブジェクトのデータベースを簡単に構築できます。

Web アプリケーションでも、さまざまな方法でこの永続オブジェクトを使うことができます。

ページ上でオブジェクトデータを表示するもっとも簡単な方法は、オブジェクトをオープンして、その内容を書き出すサーバ側のスクリプトを使うことです。

次の例は、CSP ページを使いますが、このテクニックは `%CSP.Page` クラスをサブクラス化することによって構築するアプリケーションにも当てはまります。

テーブルのオブジェクトデータを表示する

次の CSP ページは、永続オブジェクトのインスタンスをオープンし、そのプロパティのいくつかを HTML テーブルとして表示します。

そして、そのオブジェクトをクローズします。

```
<html>
<body>
<script language="Cache" runat="SERVER">
// Sample.Person のインスタンスをオープンする
Set id = 1
Set person = ##class(Sample.Person).%OpenId(1)
</script>
<table border="1">
<tr><td>Name:</td><td>#(person.Name)#</td></tr>
<tr><td>SSN:</td><td>#(person.SSN)#</td></tr>
<tr><td>City:</td><td>#(person.Home.City)#</td></tr>
<tr><td>State:</td><td>#(person.Home.State)#</td></tr>
<tr><td>Zip:</td><td>#(person.Home.Zip)#</td></tr>
</table>
<script language="Cache" runat="SERVER">
// オブジェクトを閉じる
Set person = ""
</script>
</body>
</html>
```

フォーム内にオブジェクトデータを表示する

HTML フォームにデータを表示することもできます。

次の例は、永続オブジェクトのインスタンスをオープンし、そのプロパティのいくつかを HTML フォーム内で表示し、そのオブジェクトをクローズします。

```
<html>
<body>
<script language="Cache" runat="SERVER">
// Sample.Person のインスタンスをオープンする
Set id = 1
Set person = ##class(Sample.Person).%OpenId(1)
If ($Data(%request.Data("SAVE",1))) {
// "SUBMIT"が定義されている場合は、これはサブミットとなります。
// ポストされたデータをオブジェクトに書き、保存します。
Set person.Name = $Get(%request.Data("Name",1))
Set person.SSN = $Get(%request.Data("SSN",1))
Set person.Home.City = $Get(%request.Data("City",1))
Do person.%Save()
}
</script>
<form method="POST">
<br>Name:
<input type="TEXT" name="Name" value="#(..EscapeHTML(person.Name))#">
<br>SSN:
<input type="TEXT" name="SSN" value="#(..EscapeHTML(person.SSN))#">
<br>City:
<input type="TEXT" name="City" value="#(..EscapeHTML(person.Home.City))#">
<br>
<input type="SUBMIT" name="SAVE" value="SAVE">
</form>
<script language="Cache" runat="SERVER">
// オブジェクトをクローズします
```

```
Set person = ""
```

```
</script>
```

```
</body>
```

```
</html>
```

フォームサブミット要求を処理する

上記の例は、オブジェクトの内容をフォームに表示することに加えて、ユーザが **Save** をクリックしてフォームをサブミットすると、オブジェクトへの変更を保存するものです。フォームがサブミットされたら、その内容の値はサーバに送り返されます。

この例では、フォームは、ページを処理することを依頼した同じ **CSP** ページにサブミットされます。

CSP サーバは、サブミットされた値を `%request` オブジェクトの **Data** プロパティに置きます。

そのページの開始のサーバ側スクリプトは、`request` パラメータの **Save** (サブミットボタンの名前) が定義されているかを調べて、サブミット要求への応答を処理しているかどうかを判断します。

要求がサブミット要求ならば、そのスクリプトは、フォームからサブミットされた値をオブジェクトの適切なプロパティにコピーし、そのオブジェクトを呼び出します。

```
If ($Data(%request.Data("SAVE",1))) {  
  // "SUBMIT"が定義されている場合は、これはサブミットとなります。  
  // オブジェクトにポストされたデータを書き、それを保存する  
  Set person.Name = $Get(%request.Data("Name",1))  
  Set person.SSN = $Get(%request.Data("SSN",1))  
  Set person.Home.City = $Get(%request.Data("City",1))  
  Do person.%Save()  
}  
  
<csp:object>タグ  
  
<csp:object>タグを使うと、上記の例で説明した動作のいくつかが自動的に行われます。<csp:object>
```

タグは、**CSP** ページ上で、オブジェクトのインスタンスを作成、オープン、クローズするために必要なサーバ側のコードを生成します。

例えば、ページに `person` を結び付けるには、次のようにします。

```
<csp:object NAME="person" CLASSNAME="Sample.Person" OBJID="1">
<!-- オブジェクトを使う -->
名前: #(person.Name)# <br>
自宅住所: #(person.Home.Street)#, #(person.Home.City)# <br>
```

この例では、クラス `CLASSNAME` のオブジェクトの Object ID が 1 のオブジェクトをオープンし、`person` という変数にそれを割り当てます。

実際のアプリケーションでは、object ID は `%request` オブジェクトが提供します。

```
<csp:object NAME="person" CLASSNAME="Sample.Person"
OBJID="#($Get(%request.Data("PersonID",1)))#">
名前: #(person.Name)# <br>
自宅住所: #(person.Home.Street)#, #(person.Home.City)# <br>
```

次の表現は、URL パラメータ `PersonID` を参照します。

```
$Get(%request.Data("PersonID",1))
```

`<csp:object>` タグに、次のようにヌル `OBJID` 属性を付けると、その指定クラスの新しいオブジェクトを作成します。

```
<csp:object NAME="person" CLASSNAME="Sample.Person" ObjID="">
```

データをフォームに結び付ける

CSP は、オブジェクトのデータを HTML フォームに結び付けるメカニズムを提供します。`<csp:object>` タグが、オブジェクトインスタンスとそのフォームに追加する属性 `cspbind` と、どのように結び付けられるかを指示する入力コントロールタグを指定します。

CSP コンパイラは、`cspbind` 属性を含んだフォームを認識し、自動的に次のような内容のコードを生成します。

- 適切な入力コントロール内に指定されたオブジェクトのプロパティの値を表示する
- 簡単なデータ検証を実行するクライアント側の JavaScript 関数を生成する
- その結び付けられたオブジェクトを保存するための、生成されたサーバ側メソッドを起動するクライアント

側 JavaScript 関数を生成する

- データ入力を検証し、保存するサーバ側メソッドを生成する。これらのメソッドは、CSP イベントブローカを使って、ページから直接呼び出すことができる
- フォームに結び付けられたフォームのオブジェクト ID 値を含む **hidden** フィールド OBJID を生成する

次のコードは、**Sample.Person** クラスのインスタンスに結び付けられるフォームの簡単な例です。

```
<html>
<head>
</head>
<body>
<csp:object NAME="person" CLASSNAME="Sample.Person" OBJID="1">
<form NAME="MyForm" cspbind="person">
<br>名前:
<input type="TEXT" name="Name" cspbind="Name" csprequired>
<br>社会保障番号:
<input type="TEXT" name="SSN" cspbind="SSN">
<br>都市:
<input type="TEXT" name="City" cspbind="Home.City">
<br>
<input type="BUTTON" name="SAVE" value="保存" onClick="MyForm_save0;">
</form>
</body>
</html>
```

この例では、**Sample.Person** クラスのインスタンスをオープンするために **<csp:object>** タグを使います。

このオブジェクトのインスタンスは、**person** という名前です。

それから、そのオブジェクトインスタンスを、その `form` タグに `cspbind` という名前の属性に `person` 値を指定して HTML フォームに結び付けます。

このフォームは、3つのテキスト入力コントロール「Name」「SSN」「City」を含んでいます。

それらは、それぞれのオブジェクトプロパティ「Name」「SSN」「Home.City」の `input` タグに、`cspbind` と呼ばれる属性を追加することによって結び付けられます。

Name コントロールは、**CSPREQUIRED** という属性を持っています。

これは、必須フィールドであることを指示するためのものです。

CSP コンパイラは、ユーザがこのフィールドの値を提供しているかどうかをテストする、クライアント側の JavaScript を生成します。

最後のコントロールが、それがクリックされたときにクライアント側 JavaScript 関数 `MyForm_save` を呼び出すボタンです。

`MyForm_save` 関数は、CSP コンパイラが自動的に生成します。

この関数は、フォーム上のコントロールの値を集めて、それらをサーバ側メソッドに送ります。

そのサーバメソッドは、オブジェクトのインスタンスを再オープンし、プロパティに行われた変更を適用して、オブジェクトをデータベースに保存します。

そして、変更を反映するための値を保存するために、JavaScript をクライアントに送ります。

このドキュメントに **HEAD** セクションを定義していることに注目してください。

これは、任意のクライアント側 JavaScript の位置決めのために必要です。

バウンドフォームで使われるオブジェクトのオブジェクト ID は、URL パラメータ **OBJID** に指定しなければなりません。

URL パラメータの値をオブジェクト ID として使うには、`csp:object` タグの中に、それを参照する表現を記述します。

```
<csp:object NAME="person"
CLASSNAME="Sample.Person" OBJID=#($G(%request.Data("OBJID",1)))#>
```

プロパティに結び付ける

特定の HTML 入力コントロールをオブジェクトプロパティに結び付けるには、次のことを行います。

- `csp:object` タグを使い、オブジェクトインスタンスを参照するサーバ側変数を定義する
- フォームタグを使って HTML フォームを作成する。

そのフォームタグに `cspbind` 属性を追加して、オブジェクトインスタンスに結び付ける。
`cspbind` 属性の値を `csp:object` タグの名前にする

- フォームに HTML 入力コントロールを作成し、それに `cspbind` 属性を追加する。

結び付けるためには、この `cspbind` 属性の値をオブジェクトプロパティの名前にする

<csp:serach>タグによる CSP 検索ページ

csp:search タグは、汎用的な検索ページを作成します。

検索操作を行うためにバウンドフォームととも使うことができます。

アプリケーションユーザは、バウンドフォームを含むページから CSP 検索ページにアクセスできます。

CSP 検索ページは、データベースから、ある条件に一致するオブジェクトを探すために使います。

そして、その中の 1 つのオブジェクトを選んで修正したりできます。

csp:search タグは、検索ページを表示するクライアント側 JavaScript 関数を生成します。

その検索ページは、%CSP.PageLookup クラスが表示し、検索ページの操作を制御する属性を含みます。

次の例は、MySearch という JavaScript 関数を定義しています。

この関数は、Sample.Person オブジェクトを名前で検索するポップアップ検索ウィンドウを表示します。

```
<csp:search NAME="MySearch" WHERE="Name" CLASSNAME="Sample.Person"
OPTIONS="popup"
STARTVALUES="Name" ONSELECT="MySearchSelect">
```

この検索ページの ONSELECT コールバック関数は、次のようになります。

```
<script language="JavaScript">
function MySearchSelect(id)
{
#server(..MyFormLoad(id))#;
return true;
}
</script>
```

この関数は、サーバ側メソッド MyFormLoad を起動する CSP の「#server0#」ディレクティブを使います（これは自動生成されます）。

このメソッドは、そのフォームの内容をオブジェクトのプロパティ値として集めます（オブジェクト ID は `id`）。

6-9 CSP を使ったサンプルアプリケーション

ここでは、CSP で作成した簡単なオンラインショッピングのシステムを紹介します。
サンプルアプリケーションのソースコードは非常に長いので、ここでは掲載しません。

ページ構成

このアプリケーションは、表 6-2 のようなページで構成されています。

表 6-2: サンプルアプリケーションを構成するページ

ページ	内容
login.htm	ユーザ認証用ページ。ユーザ名とパスワードを入力する
frame.csp	タイトル (title.csp)、商品とリストのページ (main.csp) と、現在のかごの情報を表示するページ (menu.csp) を含むフレームページ
title.csp	タイトル表示のページ。あいさつ文と過去の購入実績を表示する
main.csp	商品の画像イメージと説明、価格などの情報と、注文数の入力テキストなどで構成されるページ
menu.csp	顧客の現時点のかごの状態 (注文前状況) を表示するページ。注文明細と合計金額を表示する
result.csp	顧客が注文を確認したときに処理されるページ。注文オブジェクトを生成し、その内容をページに表示する
errmsgmsg.csp	エラーが発生したときに表示するエラーメッセージ用ページ

...

クラス構成

このアプリケーションを構成するモデルクラスは、表 6-3 のとおりです。

表 6-3: サンプルアプリケーションを構成するモデルクラス

モデルクラス	内容
Shop.Customer	顧客クラス
Shop.POrder	注文クラス
Shop.OrderItem	注文明細クラス
Shop.Product	商品クラス
Shop.ProductImage	商品の画像クラス
Shop.Address	住所クラス (埋め込みクラス)

セットアップ

既にセットアップ環境にインストールされています。

セットアップの構成

- shop.xml : サンプルアプリケーションのクラス
- shop-data.xml : サンプルアプリケーションのデータ
- shop-csp.zip : イメージファイルなど

この圧縮ファイルを、解凍用ソフトウェアを使って展開してください。

動作確認

IRIS をインストールした PC の Web ブラウザを起動して、次の URL を入力してください。

`http://localhost:52780/csp/user/Order/login.htm`

すると、図 6-15 のようなサンプルアプリケーションのログイン画面が表示されます。



図 6-15: サンプルアプリケーションのログイン画面

次のユーザ名とパスワードを入力して、ログインしてください。

- ユーザ名 : Taro
- パスワード : Taro

すると、図 6-16 のようなサンプルアプリケーション画面が表示されます。



図 6-16: サンプルアプリケーションの画面

適当に注文数を入力して、[かごへ] ボタンをクリックしてみましょう。

図 6-17 のように、かごの内容と合計金額が表示されます。



図 6-17 : 注文した明細 (かごの内容) が表示された画面

ここで「注文」ボタンをクリックすると、図 6-18 のような、注文を受け付けた旨を表示する確定画面が表示されます。



図6-18: 注文確定画面

このサンプルアプリケーションは、ここまでの機能しか持っていませんが、オーダリングシステムの基本的な機能は有しています。

7 章 REST

IRIS は、REST アーキテクチャスタイルをサポートしています。

REST の説明は、以下を参照ください。

<https://ja.wikipedia.org/wiki/REST>

7-1 %CSP.REST クラス

このクラスを使って、以下のことを行うことで **REST** サービスを実装できます。

- **IRIS** のメソッドを **REST** の **URL** と **HTTP** メソッドとして実行できるように **URL** マップを定義します。

REST サービスを実装するには、**%CSP.REST** クラスを継承します。

ネームスペース内に複数の**%CSP.REST** のサブクラスを定義することができますが、各々のサブクラスは、対応する別々の **CSP** アプリケーションが必要です。

管理ポータル>セキュリティ>アプリケーション>ウェブ・アプリケーションで、**CSP** アプリケーションの定義ができます。

CSP アプリケーションの定義の所で、その**%CSP.REST** のサブクラスの名前をディスパッチ・クラスとして定義します。

そしてその **REST** 呼出しの **URL** の最初の部分をその **CSP** アプリケーションの名前として指定します。

7-2 REST 用の URL マップを作る

XDATA の URLMap で指定した URL と HTTP 操作の結果として呼ばれるクラスメソッドを指定するルートを定義します。

ルート定義が 3 つの部分で構成します。

- **url** - REST サービスを呼び出す REST URL の最後の部分の形式を指定します。
- **メソッド** - その REST 呼出しの HTTP 操作を指定します。通常は、GET,POST,PUT,DELETE 操作ですが、任意の HTTP 操作を指定できます。
- **コール** - REST サービスを実行するために呼び出すクラスメソッドを指定します。

以下の例では、このサブクラスの **Test** メソッドを呼出し、**GET** メソッドを使うことを指定しています。

```
XData UrlMap
{
  <Routes>
    <Route Url="/test" Method="GET" Call="Test"/>
  </Routes>
}
```

REST の URL は以下の部分で構成されます。

- IRIS サーバーのサーバー名とポート番号

例： http://localhost:52773/

- ウェブ・アプリケーションの名前

- REST URL の残りの部分は、ルート url 要素として定義したものです。

前に : がある時には、それはパラメータであることを意味しています。

以下のルート定義の例では、URL に 2 つのパラメータ、`namespace` と `class` を定義しています。

```
<Route Url="/class/:namespace/:classname" Method="GET" Call="GetClass" />
```

REST 呼出しとして以下の様な呼出しを行います。

```
http://localhost:52780/rest/class/user/Cinema.Review
```

そしてこの呼出しは、`"user"` と `"Cinema.Review"` を引数として `GetClass` メソッドを呼出します。

`GetClass` メソッドは以下の様な定義で始まります。

```
/// このメソッドは、指定した Cache クラスの説明を返します。  
ClassMethod GetClass(  
    pNamespace As %String,  
    pClassname As %String) As %Status  
{
```

同じ URL に対して異なる HTTP 操作の異なるメソッドを定義できます。

```
<Route Route Url="/request" Method="GET" Call="GetRequest" />  
<Route Route Url="/request" Method="POST" Call="PostRequest" />
```

さらに異なる HTTP 操作を同じ 1 つのメソッドで操作することもできます。

この場合には、`CSP` リクエストオブジェクトを調べることで URL のテキストを取得して判別することができます。

7-3 データ形式を指定する

REST サービスを定義する時に様々な形式（JSON、XML、テキスト、CSV）でデータを指定できます。

HTTP リクエストの **ContentType** 要素を指定して、送ることで期待するデータ形式を指定できます。

また HTTP リクエストの **Accept** 要素を指定することで返り値の形式を要求することができます。

以下の例では、JSON データを要求しているかどうかをチェックしています。

```
If $Get(%request.CgiEnvs("HTTP_ACCEPT"))="application/json"
```

8 章 JSON

REST でデータ交換する際のデータ形式は上記でも少し触れた様に様々な形式を使用することが可能ですが、REST の普及とともに JSON をデータ交換の形式として使用するケースが増えてきました。

IRIS は、JSON を容易に使用できるようなサポート機能を含んでいます。

8.1 概要

IRIS は、動的オブジェクトと動的配列という新しいオブジェクトの種類をサポートします。

これらのオブジェクトはデータをシリアル化して JSON に相互変換するメソッドを提供します。

それらのオブジェクトを生成するための便利な方法を提供する新しい ObjectScript コマンドのシンタックス（動的オブジェクト表現と動的配列表現）があります。

8.2 動的オブジェクトと配列

動的オブジェクトは特別な種類の IRIS オブジェクトでスキーマがありません。

つまり割り当て文を使用してプロパティを作っていきます。

動的オブジェクトは JSON 以外にも使用できますが、現時点では JSON サポートに焦点を当てています。

3つのキークラスがこれらの動的オブジェクトを定義します。

%DynamicObject のインスタンスが順番の決まっていない名前付きプロパティとその値を持つことを想定します。

プロパティの値にはリテラル値、オブジェクト、配列が可能です。

%DynamicArray のインスタンスには順番の決まった値のセットを持つことを想定しており、その位置によってアクセスができます。位置の開始は 0 です。

%AbstractObject はこれらのクラスの共通スーパークラスとなり、共通のメソッドを定義しています。

oref という変数が動的オブジェクトのインスタンスを表現しているとすると、以下の表現はそのインスタンスのメソッド%methodname を呼出します。

`do oref.%methodname(arguments)`

8.3 動的オブジェクトの使用

8.3.1 動的オブジェクトの作成

以下のように動的オブジェクトを作成できます。

```
set varname = {"プロパティ 1":123,"プロパティ 2":456}
```

%FromJSON()メソッドを使うこともできます。

```
set jstring="{\"プロパティ 1\":123,\"プロパティ 2\":456}"  
set varname=##class(%DynamicObject).%FromJSON(jstring)
```

8.3.2 動的オブジェクトの参照、設定

プロパティを参照するにはドットシンタックスを使用します。

通常の IRIS プロパティ名として許可されない空白などの文字を含めたい時には、ダブルクォートで囲みます。

```
set o."another property" = 456
```

%Set メソッドを使うこともできます。

```
do varname.%Set("another proprty",456)
```

8.3.3 動的オブジェクトにプロパティがあるかチェックする

動的オブジェクトに該当するプロパティがあるかどうかを確認するには、`$isDefined()` メソッドを使います。

```
USER>set o={"プロパティ 1":123,"プロパティ 2":456}
USER>w o.%IsDefined("プロパティ 1")
1
USER>w o.%IsDefined("プロパティ 3")
0
```

8.3.4 動的オブジェクトからプロパティを削除する

プロパティを削除するには、`%Remove()` メソッドを使います。

```
USER>set varname={"プロパティ 1":123,"プロパティ 2":456}

USER>write varname.%ToJSON()
{"プロパティ 1":123,"プロパティ 2":456}

USER>do varname.%Remove("プロパティ 1")

USER>write varname.%ToJSON()
{"プロパティ 2":456}
```

8.3.5 プロパティの値を得る

プロパティの値を取得するには、以下のいずれかの方法があります。

- ドットシンタックスでプロパティを参照する

```
USER>write varname.プロパティ 2  
456
```

- %Get()メソッドを使う

```
USER>set x=varname.%Get("プロパティ 2")  
USER>write x  
456
```

8.3.6 プロパティの数を得る

プロパティ数を取得するには、%Size()メソッドを使います。

```
USER>set varname={"プロパティ 1":123,"プロパティ 2":[7,8,9],"プロパティ  
3":{"a":1,"b":2}}  
USER>write varname.%Size()  
3
```

8.3.7 プロパティを繰り返し処理する

複数のプロパティを繰り返し処理するには、以下のように操作します。

例:

```
set varname={"プロパティ 1":123,"プロパティ 2":"abc","プロパティ 3":true,"プロパティ 4":[7,8,9],"プロパティ 5":{"a":1,"b":2}}
set iterator=varname.%GetIterator()
while iterator.%GetNext(.key,.value) {
write !, "プロパティ名: ",key
write !, "プロパティ値: ",value
}
```

このサンプルを実行した結果は以下のようになります:

```
プロパティ名: プロパティ 1  
プロパティ値: 123  
プロパティ名: プロパティ 2  
プロパティ値: abc  
プロパティ名: プロパティ 3  
プロパティ値: 1  
プロパティ名: プロパティ 4  
プロパティ値: 3@%Library.Array  
プロパティ名: プロパティ 5  
Property value: 7@%Library.Object
```

8.4 動的配列の使用

8.4.1 動的配列の作成

動的配列の作成は以下のようになります。

```
set array = [123,"A",true]  
  
set array=[123,("ABC"["A"]),true]
```

%FromJSON()メソッドも使えます。

```
set jstring="[123, ""A"",true]"  
set array=##class(%Array).%FromJSON(jstring)
```

8.4.2 動的配列に項目を追加する

項目を追加するには、%Push()メソッドを使います。

```
set array=[]
do array.%Push(42)
do array.%Push("abc")
do array.%Push("def")
set subarray=[]
do subarray.%Push("X")
do subarray.%Push("Y")
do subarray.%Push(0)
do array.%Push(subarray)
```

8.4.3 配列の項目を修正する

```
USER>set array=[123,"A",true]
USER>set array."0"]=456
USER>write array.%ToJSON()
[456,"A",true]
```

8.4.4 動的配列に該当プロパティがあるかどうかチェックする

%IsDefined()メソッドを使います。

引数には配列インデックスを指定します。

```
USER>set a=["a","b","c"]
USER>w a.%IsDefined(0)
1
USER>w a.%IsDefined(4)
0
```

8.4.5 動的配列から項目を削除する

項目を削除するには%Pop()メソッドを使います。

```
USER>set array=[]
USER>do array.%Push("abc")
USER>do array.%Push("def")
USER>write array.%ToJSON()
["abc","def"]
USER>do array.%Pop()
USER>write array.%ToJSON()
["abc"]
```


8.4.6 配列項目の値を得る

値を取得するには、以下のいずれかの方法があります。

- ・ ドットシンタックスを使ってプロパティを参照する

```
USER>write varname."0"  
456
```

- ・ \$get()メソッドを使う

```
USER>set x=varname.%Get(0)  
USER>write x  
456
```

8.4.7 項目数を得る

動的配列の項目数を取得するには、%Size()メソッドを使います。

```
USER>set array=["abc",999]  
USER>do array.%Push("another one")  
USER>write array.%Size()  
3
```

8.4.8 配列項目を繰り返す

動的配列を繰り返し処理するには、以下のようにします。

```
set varname=["abc",999,"def"]
set iterator=varname.%GetIterator()
while iterator.%GetNext(.key,.value) {
write !, "項目インデックス: ",key
write !, "項目値: ",value
}
```

結果は以下のようになります:

```
項目インデックス: 0
項目値: abc
項目インデックス: 1
項目値: 999
項目インデックス: 2
項目値: def
```

8.5 動的オブジェクトまたは動的配列を JSON にシリアルライズする

動的オブジェクトまたは配列から JSON 形式の文字列を生成する方法を説明します。

8.5.1 基本テクニック

%ToJSON() メソッドを使います。

```
USER>set varname={"プロパティ 1":true}
USER>do varname.%Set("プロパティ 2",123)
USER>write varname.%ToJSON()
{"プロパティ 1":true,"プロパティ 2":123}
```

JSON オブジェクトは、順序付けされていないプロパティのセットなので、%ToJSON() はプロパティを適当な順番で返します。

この順番は作成した順番ではないかもしれません。

動的配列も同様にシリアルライズできます。

```
USER>set varname=[]
USER>do varname.%Push("新しい値")
USER>do varname.%Push("新しい値")
USER>do varname.%Push("新しい値")
USER>write varname.%ToJSON()
["新しい値","新しい値","新しい値"]
```

10章 他プログラミング言語との連携

他のプログラミング言語との連携については、**Caché** 開発当初より様々な仕組みを提供してきました。

その結果、他プログラミング言語との連携に関して様々な選択肢があり、どれを選択すれば良いのかわかりずらくなってきました。

また、時代の変遷と共に技術として古くなったもの、普及しなかったものなどいくつかの仕組みについては現在ではその使用を推奨していないものもあります。

ここでは、現時点で推奨している連携の仕組みについてまとめてみます。

10.1 一般的に推奨するインタフェース

10.1.1 REST/JSON

この本にも章を割り当てている通り、昨今 **REST/JSON** が普及してきています。

最近では主要な言語、開発環境はほとんど **REST/JSON** をサポートしているので、**REST/JSON** の使用ができない特別な理由がない限り **REST/JSON** の使用を推奨します。

10.1.2 Web サービス (SOAP)

REST/JSON の普及前には **SOAP** も良く使われていました。

既に **Web サービス** でインタフェースを構築してあるもしくは既存の **Web サービス** の使用が前提であるプロジェクト用の開発などの場合には、**Web サービス** の使用が可能です。

10.2 Java 言語との連携

10.2.1 JDBC

JDBC は、Java 言語とのデータベース連携の標準ですので、他データベース用に書かれたアプリケーションの書き換え等の場合には適切なインタフェースとなります。

10.2.2 Java eXTreme

比較的単純なオブジェクトモデルを大量、高速に処理したい場合には、Java eXTreme の使用を推奨します。

10.2.3 Hibernate

複雑なオブジェクトを取り扱いたい場合には、Hibernate の使用を推奨します。

Hibernate の詳細は、www.hibernate.org を参照してください。

10.2.4 Native API

IRIS のクラスインスタンス、メソッドやグローバルを直接操作する API も用意しています。

10.3 .NET との連携

10.3.1 ADO.NET

SQL 主体の開発の際は、ADO.NET Managed Provider の IRIS による実装の使用を推奨します。

10.3.2 .NET eXTreme

Java と同様に比較的単純なオブジェクトモデルを大量、高速に処理したい場合には、.Net eXTreme の使用を推奨します。

10.3.3 ADO.NET Entity Framework

Java と同様に複雑なオブジェクトモデルを取り扱いたい場合には、ADO.NET Entity Framework の使用を推奨します。

10.3.4 Native API

IRIS のクラスインスタンス、メソッドやグローバルを直接操作する API も用意しています。

10.4 使用推奨しないインタフェース

下位互換性の見地から現在もサポートされているインタフェースには様々な理由で新規開発に使用することを推奨しないインタフェースがあります。以下にそれらのインタフェースを列挙します。

Java バインディング (IRIS には含まれません)

Jalapeño (IRIS には含まれません)

.NET オブジェクト・バインディング

ActiveX バインディング

Direct (VisM)インタフェース

アクティベート

11 章 XML のサポート

IRIS オブジェクトを XML に投影する方法が用意されています。

オブジェクトを定義するクラスのスーパークラスリストに `%XML.Adaptor` を追加することで、XML に投影する方法を利用することができます。

11-1 クラスのオブジェクトを XML ドキュメントにエクスポートする

オブジェクトを XML に投影 (または、そのオブジェクトの XML プロジェクションを定義) するには、以下の手順を実行します。

オブジェクトを定義するクラスのスーパークラス・リストに `%XML.Adaptor` を追加します。

以下はその例です。

```
Class MyApp.MyClass Extends (%Persistent, %XML.Adaptor)
{
//class details
}
```

XML 対応クラスの例

以下は、構造的プロパティの主な種類を含む XML 対応クラスを示します。

```
Class Basics.BasicDemo Extends (%RegisteredObject, %XML.Adaptor)
{

    Parameter XMLTYPENAMESPACE = "mytypes";

    Property SimpleProp As %String;

    Property ObjProp As SimpleObject;

    Property Collection1 As list Of %String;

    Property Collection2 As list Of SimpleObject;

    Property MultiDimProp As %String [ MultiDimensional ];

    Property PrivateProp As %String [ Private ];

}
```

XML ドキュメントの例

以下は、BasicDemo クラスのインスタンスから生成された XML ドキュメントを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<BasicDemo>
  <SimpleProp>abc</SimpleProp>
  <ObjProp>
    <MyProp>12345</MyProp>
    <AnotherProp>67890</AnotherProp>
  </ObjProp>
  <Collection1>
    <Collection1Item>list item 1</Collection1Item>
    <Collection1Item>list item 2</Collection1Item>
  </Collection1>
  <Collection2>
    <SimpleObject>
      <MyProp>12345</MyProp>
      <AnotherProp>67890</AnotherProp>
    </SimpleObject>
    <SimpleObject>
      <MyProp>12345</MyProp>
      <AnotherProp>67890</AnotherProp>
    </SimpleObject>
  </Collection2>
</BasicDemo>
```

XML ライターの作成の概要

IRIS には、IRIS オブジェクトを XML 出力するツールが用意されています。

以下に%XML.Writer クラスを使用して XML 対応オブジェクトにアクセスし、XML を出力する簡単な例を示します。

```
GXML>Set p=##class(GXML.Person).%OpenId(1)
```

```
GXML>Set w=##class(%XML.Writer).%New()
```

```
GXML>Set w.Indent=1
```

```
GXML>Set status=w.RootObject(p)
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Person GroupID="R9685">
```

```
  <Name>Zimmerman,Jeff R.</Name>
```

```
  <DOB>1961-10-03</DOB>
```

```
  <Address>
```

```
    <City>Islip</City>
```

```
    <Zip>15020</Zip>
```

```
  </Address>
```

```
  <Doctors>
```

```
    <Doctor>
```

```
      <Name>Sorenson,Chad A.</Name>
```

```
    </Doctor>
```

```
    <Doctor>
```

```
      <Name>Sorenson,Chad A.</Name>
```

```
    </Doctor>
```

```
    <Doctor>
```

```
      <Name>Uberoth,Roger C.</Name>
```

```
    </Doctor>
```

```
  </Doctors>
```

```
</Person>
```

11-2 XML ドキュメントを IRIS にインポートする

%XML.Reader クラスを使用して、IRIS オブジェクトに XML ドキュメントをインポートすることができます。

XML ドキュメントをインポートするには、%XML.Reader のインスタンスを作成してから、そのインスタンスメソッドを呼び出します。

そこで XML ソースドキュメントを指定し、XML 要素を XML 対応クラスに関連付け、ソースからオブジェクトに要素を読み込みます。

%XML.Reader クラスによって生成されたオブジェクトインスタンスは、そのままではデータベースに格納されません。

データベースにオブジェクトを格納するには、%Save()メソッドを呼び出します。

以下は、XML ドキュメントをインポートしてデータベースに保存する例です。

```
GXML>Set reader = ##class(%XML.Reader).%New()

GXML>Set file="c:\sample-input.xml"

GXML>Set status = reader.OpenFile(file)

GXML>Write status
1
GXML>Do reader.Correlate("Person","GXML.Person")

GXML>Do reader.Next(.object,.status)

GXML>Write status
1
GXML>Write object.Name
Worthington,Jeff R.
GXML>Write object.Doctors.Count()
2
GXML>Do object.%Save()
```

この例では、次のサンプル XML を使用しています。

```
<Person GroupID="90455">
  <Name>Worthington,Jeff R.</Name>
  <DOB>1976-11-03</DOB>
  <Address>
    <City>Elm City</City>
    <Zip>27820</Zip>
  </Address>
  <Doctors>
    <Doctor>
      <Name>Best,Nora A.</Name>
    </Doctor>
    <Doctor>
      <Name>Weaver,Dennis T.</Name>
    </Doctor>
  </Doctors>
</Person>
```


11-3 DOM の生成

DOM として XML ドキュメントを開く

既存の XML ドキュメントを DOM として使用するために開くには、以下の手順を実行します。

%XML.Reader クラスのインスタンスを作成します。

ソースドキュメントを開きます。

ドキュメントを開くには%XML.Reader の次のメソッドのいずれかを使用します。

OpenFile() - ファイルを開きます。

OpenStream() - ストリームを開きます。

OpenString() - 文字列を開きます。

DOM である Document プロパティにアクセスします。

このプロパティは%XML.Document のインスタンスでドキュメント全体についての情報を見つけるために使用できるメソッドを提供します。

XML ドキュメントを格納したストリームが既にある場合には、%XML.Document の GetDocumentFromStream()メソッドを呼び出します。

ファイルの DOM への変換

以下のメソッドで XML ファイルを読み取り、そのドキュメントを表す %XML.Document のインスタンスを返します。

```
ClassMethod GetXMLDocFromFile(file) As %XML.Document
{
    set reader=##class(%XML.Reader).%New()

    set status=reader.OpenFile(file)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit
$$$NULLOREF}

    set document=reader.Document
    quit document
}
```

オブジェクトの DOM への変換

以下のメソッドで OREF を受け入れ、そのオブジェクトを表す %XML.Document のインスタンスを返します。 OREF は XML 対応クラスのインスタンスと想定しています。

```
ClassMethod GetXMLDoc(object) As %XML.Document
{
    //XML 有効クラスのインスタンスであることを確かめてください
    if '$IsObject(object){
        write "引数がオブジェクト参照ではない"
        quit $$$NULLOREF
    }
    set classname=$CLASSNAME(object)
    set isxml=$zobjclassmethod(classname,"%Extends","%XML.Adaptor")
    if 'isxml {
        write "引数が XML 有効クラスのインスタンスではない"
        quit $$$NULLOREF
    }
    //ステップ 1 - オブジェクトを XML としてストリームに書く
    set writer=##class(%XML.Writer).%New()
    set stream=##class(%GlobalCharacterStream).%New()
    set status=writer.OutputToStream(stream)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$ERROR0}
    set status=writer.RootObject(object)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$ERROR0}

    //ステップ 2 - ストリームから %XML.Document を抽出する
    set
    status=##class(%XML.Document).GetDocumentFromStream(stream,.document)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$ERROR0}

    quit document
}
```

DOM ノードのナビゲート

ドキュメントのノードにアクセスするには以下の 2 つの手法を使用できます。

`%XML.Document` のインスタンスの `GetNode()` メソッドを使用します。

`%XML.Document` のインスタンスの `GetDocumentElement()` メソッドを呼び出します。

これらのメソッドは `%XML.Node` のインスタンスを返します。

子ノードまたは兄弟ノードへの移動

子ノードまたは兄弟ノードに移動するには、`%XML.Node` のインスタンスの以下のメソッドを使用します。

`MoveToFirstChild()`

`MoveToLastChild()`

`MoveToNextSibling()`

`MoveToPreviousSibling()`

これらのメソッドは、別のノードへの移動を行います。

親ノードへの移動

現在のノードの親ノードに移動するには `%XML.Node` のインスタンスの `MoveToParent()` メソッドを使用します。

特定のノードへの移動

特定のノードに移動するには、`%XML.Node` のインスタンスの `NodeId` プロパティを設定できます。

```
set saveNode = node.NodeId
//..... lots of processing
//...
// restore position
set node.NodeId=saveNode
```

11-4 %XML.TextReader クラスの使用

%XML.TextReader クラスを使うと、任意の XML ドキュメントを単純、簡単な方法で読み取ることができます。

このクラスを使用し、適格な形式 (Well Formed) の XML ドキュメントをナビゲートし、その中の情報

(要素、属性、コメント、ネームスペース、URI など) を表示できます。

テキストリーダーメソッドの作成

テキストリーダーオブジェクトには、ナビゲート可能なノードのツリーが含まれ、そのそれぞれにソースドキュメントについての情報が含まれています。

そのオブジェクトのメソッドでドキュメントをナビゲートし、ドキュメントに関する情報を検出できます。

以下のメソッドのいずれかを使用して、ドキュメントソースを指定します。

メソッド	最初の引数
ParseFile()	完全なパスを含むファイル名
ParseStream()	ストリーム
ParseString()	文字列
ParseURL()	URL

ソースドキュメントは適格な XML ドキュメント、つまり、XML 構文の基本的な規約に従ったドキュメントである必要があります。

これらのメソッドはそれぞれ、結果が成功だったかどうかを示すステータス (\$\$\$OK または失敗コード) を返します。

\$\$\$OK を返す場合、これらのメソッドの第二引数に参照形式で XML ドキュメント内の情報を含むテキストリーダーオブジェクト

を返します。

以下のインスタンスメソッドのいずれかを使用して、ドキュメントの読み取りを開始します。

ドキュメントの最初のノードへ移動するには、Read()を使用します。

特定のタイプの最初の要素に移動するには ReadStartElement()を使用します。

"chars"の最初のノードへ移動するには MoveToContent()を使用します。

関係するプロパティの値があれば、それを取得します。

利用可能なプロパティには Name,Value,Depth などがあります。

必要に応じてドキュメントのナビゲートおよびプロパティ値の取得を続けます。

現在のノードが要素の場合、`MoveToAttributeIndex()`または `MoveToAttributeName()` メソッドを使用して要素の属性にフォーカスを移動することができます。

要素に戻るには `MoveToElement()` を使用します。

必要に応じて `Rewind()` メソッドを使用してドキュメントの最初に戻ります。

ここでは、任意の XML ファイルを読み取り、各ノードのシーケンス番号、タイプ、名前、および値を表示する単純なメソッドを示します。


```
ClassMethod WriteNodes(myfile As %String)
{
    set status=##class(%XML.TextReader).ParseFile(myfile,.textreader)
    //ステータスをチェック
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}
    //ドキュメントをノード毎に繰り返し走査
    while textreader.Read()
    {
        Write !, "Node ", textreader.seq, " is a(n) "
        Write textreader.NodeType," "
        If textreader.Name=""
        {
            Write "名前: ", textreader.Name
        }
        Else
        {
            Write "名前なし"
        }
        Write !, "  path: ",textreader.Path
        If textreader.Value=""
        {
            Write !, "  値: ", textreader.Value
        }
    }
}
```

この例は、以下を実行します。

ParseFile() クラス・メソッドを呼び出します。

このメソッドはソース・ファイルを読み取り、テキスト・リーダー・オブジェクトを生成して、変数 **doc** 内でそのオブジェクトを参照によって返します。

ParseFile() が成功した場合、このメソッドは Read() メソッドを実行して、ドキュメント内で次のノードをそれぞれ検索します。

各ノードについて、このメソッドは、そのノードのシーケンス番号、ノード・タイプ、ノード名 (存在する場合)、ノード・パス、およびノード値 (存在する場合) が含まれた出力行を記述します。現在のデバイスに出力されます。

以下の例のソース・ドキュメントを考えてみます。

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="mystyles.css"?>
<Root>
  <s01:Person xmlns:s01="http://www.root.org">
    <Name attr="xyz">Willeke, Clint B.</Name>
    <DOB>1925-10-01</DOB>
  </s01:Person>
</Root>
```

このソース・ドキュメントについては、前述のメソッドで以下の出力が生成されます。

Node 1 is a(n) processinginstruction 名前: xml-stylesheet

path:

value: type="text/css" href="mystyles.css"

Node 2 is a(n) element 名前: Root

path: /Root

Node 3 is a(n) startprefixmapping 名前: s01

path: /Root

値: s01 http://www.root.org

Node 4 is a(n) element 名前: s01:Person

path: /Root/s01:Person

Node 5 is a(n) element 名前: Name

path: /Root/s01:Person/Name

Node 6 is a(n) chars and 名前なし

path: /Root/s01:Person/Name

値: Willeke,Clint B.

Node 7 is a(n) endelement 名前: Name

path: /Root/s01:Person/Name

Node 8 is a(n) element 名前: DOB

path: /Root/s01:Person/DOB

Node 9 is a(n) chars and 名前なし

path: /Root/s01:Person/DOB

値: 1925-10-01

Node 10 is a(n) endelement 名前: DOB

path: /Root/s01:Person/DOB

Node 11 is a(n) endelement 名前: s01:Person

path: /Root/s01:Person

Node 12 is a(n) endprefixmapping named: s01

path: /Root

値: s01

Node 13 is a(n) endelement 名前: Root

path: /Root

以下の例では、XML ファイルを読み取り、その中の各要素をリスト表示します。

```
ClassMethod ShowElements(myfile As %String)
{
    set status = ##class(%XML.TextReader).ParseFile(myfile,.textreader)
    //ステータスをチェック
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}
    //ノード毎にドキュメントを繰り返し走査
    while textreader.Read()
    {
        if (textreader.NodeType = "element")
        {
            write textreader.Name,!
        }
    }
}
```

このメソッドは、**NodeType** プロパティを使用して、各ノードのタイプをチェックします。

そのノードが要素の場合、メソッドはその名前を現在のデバイスに出力します。

前述の XML ソース・ドキュメントについては、このメソッドで以下の出力が生成されます。

```
Root
s01:Person
Name
DOB
```

11-5 XPath の使用

XPath (XML Path Language) は、XML ドキュメントからデータを取得するための XML を使用した式言語です。

任意の XML ドキュメントを指定して、`%XML.XPATH.Document` クラスを使用し、簡単に XPath 式を評価できます。

IRIS における XPath 式の評価の概要

IRIS XML サポートを使用し、任意の XML ドキュメントを使用して XPath 式を評価するには、次の手順を実行します。

`%XML.XPATH.Document` のインスタンスを作成します。

そのためには、`CreateFromFile()`、`CreateFromStream()`、または `CreateFromString()` のいずれかのクラス・メソッドを使用します。

これらのどのメソッドを使用する場合でも、入力する XML ドキュメントを最初の引数として指定し、`%XML.XPATH.Document` のインスタンスを出力パラメータとして受け取ります。

この手順によって、組み込み XSLT プロセッサを使用して XML ドキュメントを解析します。

`%XML.XPATH.Document` のインスタンスの `EvaluateExpression()` メソッドを使用します。

このメソッドには、評価するノード・コンテキストと式を指定します。

ノード・コンテキストは、式を評価するコンテキストを指定します。

XPath 構文を使用してその目的のノードへのパスを表します。

以下はその例です。

```
"/staff/doc"
```

評価する式にも XPath 構文を使用します。以下はその例です。

```
"name[@last='Marston']"
```

出力パラメータとして (3 番目の引数として) 結果を受け取ります。

XPath の結果の使用法

このセクションの例は、以下の XML ドキュメントに対して XPath 式を評価します。

```
<?xml version="1.0"?>
<staff>
  <doc type="consultant">
    <name first="David" last="Marston">Mr. Marston</name>
    <name first="David" last="Bertoni">Mr. Bertoni</name>
    <name first="Donald" last="Leslie">Mr. Leslie</name>
    <name first="Emily" last="Farmer">Ms. Farmer</name>
  </doc>
  <doc type="GP">
    <name first="Myriam" last="Midy">Ms. Midy</name>
    <name first="Paul" last="Dick">Mr. Dick</name>
    <name first="Scott" last="Boag">Mr. Boag</name>
    <name first="Shane" last="Curcuru">Mr. Curcuru</name>
    <name first="Joseph" last="Kesselman">Mr. Kesselman</name>
    <name first="Stephen" last="Auriemma">Mr. Auriemma</name>
  </doc>
</staff>
```

サブツリー結果を持つ XPath 式の評価

以下のクラス・メソッドは XML ファイルを読み取り、XML サブツリーを返す XPath 式を評価します。

```
/// DOM を返す XPath 表現を評価する
ClassMethod Example1()
{
    Set tSC=$$OK
    do {

        Set tSC=##class(%XML.XPATH.Document).CreateFromFile(filename,.tDoc)
        If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC) Quit}

        Set context="/staff/doc"
        Set expr="name[@last='Marston']"
        Set tSC=tDoc.EvaluateExpression(context,expr,.tRes)
        If $$$ISERR(tSC) Quit

        Do ##class(%XML.XPATH.Document).ExampleDisplayResults(tRes)

    } while (0)
    If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC)}
    Quit
}
```

この例では、last 属性が Marston である <name> 要素を持つすべてのノードが選択されます。

この式は、<staff> 要素の <doc> ノードで評価されます。

この例は %XML.XPATH.Document の ExampleDisplayResults() クラス・メソッドを使用することに注意してください。

前述の XML ファイルを入力として指定して `Example10` メソッドを実行すると、以下の出力が表示されます。

XPATH DOM

element: name

attribute: first Value: David

attribute: last Value: Marston

chars : #text Value: Mr. Marston

スカラ結果を持つ XPath 式の評価

以下のクラス・メソッドは XML ファイルを読み取り、スカラ結果を返す XPath 式を評価します。

```
/// 値を返す XPath 表現を評価する
ClassMethod Example20
{
    Set tSC=$$$OK
    do {

        Set tSC=##class(%XML.XPATH.Document).CreateFromFile(filename,.tDoc)
        If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC) Quit}

        Set tSC=tDoc.EvaluateExpression("/staff","count(doc)",.tRes)
        If $$$ISERR(tSC) Quit

        Do ##class(%XML.XPATH.Document).ExampleDisplayResults(tRes)

    } while (0)
    If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC)}
    Quit
}
```

この例では、<doc> サブノードがカウントされます。

この式は、<staff> 要素で評価されます。

前述の XML ファイルを入力として指定して **Example20** メソッドを実行すると、以下の出力が表示されます。

XPATH VALUE
2

11-6 XSLT の使用

XSLT (Extensible Stylesheet Language Transformations) は XML 由来の言語であり、これを使用して指定の XML ドキュメントを別の XML ドキュメントまたは別の “人が読める形式の” ドキュメントに変換する方法を記述します。

%XML.XSLT および %XML.XSLT2 パッケージ内のクラスを使用すると、XSLT 1.0 および 2.0 の変換を実行できます。

IRIS における XSLT 変換の実行の概要

IRIS では、XSLT 1.0 および XSLT 2.0 をサポートしています。

XSLT 変換を実行するには、以下の手順に従います。

XSLT 2.0 の場合は、次のセクションで説明するように、XSLT ゲートウェイ・サーバを構成します。

または、既定の構成を使用します。

XSLT 1.0 を使用する変換では、ゲートウェイは必要ありません。

必要に応じて、IRIS は自動的にゲートウェイを起動します。ゲートウェイは手動で起動することもできます。

必要に応じて、コンパイル済みスタイル・シートを作成して、メモリにロードします。

スタジオには、XSLT 変換のテストに使用できるウィザードも用意されています。

例

ここでは、以下のコードを使用する 2 つの変換を紹介します (ただし、入力ファイルは異なります)。

```
Set in="c:\0test\xslt-example-input.xml"
Set xsl="c:\0test\xslt-example-stylesheet.xsl"
Set out="c:\0test\xslt-example-output.xml"
Set tSC=##class(%XML.XSLT.Transformer).TransformFile(in,xsl,.out)
Write tSC
```

例 1 : 単純な置換

この例では、以下の入力 XML から始めます。

```
<?xml version="1.0" ?>
<s1 title="s1 title attr">
  <s2 title="s2 title attr">
    <s3 title="s3 title attr">Content</s3>
  </s2>
</s1>
```

また、以下のスタイル・シートを使用します。

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="xml" indent="yes"/>

<xsl:template match="//@* | //node()">
  <xsl:copy>
    <xsl:apply-templates select="@*" />
    <xsl:apply-templates select="node()" />
  </xsl:copy>
</xsl:template>

<xsl:template match="/s1/s2/s3">
<xsl:apply-templates select="@*" />
<xsl:copy>
Content Replaced
</xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

この場合、出力ファイルは以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<s1 title="s1 title attr">
  <s2 title="s2 title attr">
    <s3>
      Content Replaced
    </s3>
  </s2>
</s1>
```

例 2 : コンテンツの抽出

この例では、以下の入力 XML から始めます。

```
<?xml version="1.0" encoding="UTF-8"?>
<MyRoot>
  <MyElement No="13">Some text</MyElement>
  <MyElement No="14">Some more text</MyElement>
</MyRoot>
```

また、以下のスタイル・シートを使用します。

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xsl:output method="text"
    media-type="text/plain"/>
<xsl:strip-space elements="*" />

<!-- utilities not associated with specific tags -->
<!-- emit a newline -->
<xsl:template name="NL">
    <xsl:text>&#xa;</xsl:text>
</xsl:template>

<!-- beginning of processing -->

<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="MyElement">
    <xsl:value-of select="@No"/>
    <xsl:text>: </xsl:text>
    <xsl:value-of select="."/>
    <xsl:call-template name="NL"/>
</xsl:template>

</xsl:stylesheet>

```

この場合、出力ファイルは以下のようになります。

13: Some text

14: Some more text

11-7 XML スキーマからのクラスの生成

スタジオは、(ファイルまたは URL からの) XML スキーマを読み取り、そのスキーマで定義されたタイプに対応する XML 対応クラスのセットを生成するウィザードを提供します。

すべてのクラスは `%XML.Adaptor` を拡張します。

クラスを収めるパッケージと、クラス定義の詳細を制御するさまざまなオプションを指定します。

ウィザードはクラス・メソッドとしても用意されているので、利用可能です。

内部的には、SOAP ウィザードは、WSDL ドキュメントを読み取って Web クライアントまたは Web サービスを生成するときに、このクラス・メソッドを使用します。

ウィザードの使用法

XML スキーマ・ウィザードを使用するには、以下の手順を実行します。

[ツール]→[アドイン]→[XML スキーマ・ウィザード] をクリックします。

最初の画面で、使用する XML スキーマを指定します。

以下のいずれかを行います。

[スキーマファイル] で、[参照] をクリックして XML スキーマ・ファイルを選択します。

[URL] で、スキーマの URL を指定します。

[次へ] をクリックします。

次の画面に、正しいものを選択したことを確認できるように、スキーマが表示されます。

クラスからの XML スキーマの生成

XML スキーマを構築する手順は次のとおりです。

`%XML.Schema` の新しいインスタンスを作成します。

オプションとして、インスタンスのプロパティを設定します。

ほかに割り当てられていないタイプのネームスペースを指定するには、`DefaultNamespace` プロパティを指定します。

既定は `Null` です。

インスタンスの `AddSchemaType()` メソッドを呼び出します。

スキーマの出力の生成

`%XML.Schema` のインスタンスを作成したら、以下の手順を実行して出力を生成します。

インスタンスの `GetSchema()` メソッドを呼び出し、ドキュメント・オブジェクト・モデル (DOM) のノードとしてスキーマを返します。

このメソッドの引数は、スキーマのターゲットのネームスペースの `URI` のみです。

このメソッドは、`%XML.Node` のインスタンスを返します。

スキーマにネームスペースが指定されていない場合、`GetSchema()` の引数として `""` を使用します。

オプションとして、この `DOM` を変更します。

スキーマを生成する手順は以下のとおりです。

%XML.Writer クラスのインスタンスを作成し、オプションで **Indent** などのプロパティを設定します。

オプションで、ライターの **AddNamespace()** などのメソッドを呼び出して、<schema> 要素にネームスペース宣言を追加します。

多くの場合、スキーマは単純な **XSD** タイプを参照するため、**AddSchemaNamespace()** を呼び出して **XML** スキーマのネームスペースを追加することは有用です。

引数としてスキーマを使用して、ライターの **DocumentNode()** メソッドまたは **Tree()** メソッドを呼び出します。

例

簡単な例

最初の例では、基本的な手順を示しています。

```
Set schemawriter=##class(%XML.Schema).%New()

//add class and package (for example)
Set status=schemawriter.AddSchemaType("Facets.Test")

//retrieve schema by its URI
//which is null in this example
Set schema=schemawriter.GetSchema("")

//create writer
Set writer=##class(%XML.Writer).%New()
Set writer.Indent=1

//use writer
Do writer.DocumentNode(schema)
```

より複雑なスキーマの例

以下の例を考えてみます。Person クラスは以下のとおりです。

```
Class SchemaWriter.Person Extends (%Persistent, %XML.Adaptor)
{

    Parameter NAMESPACE = "http://www.myapp.com";

    Property Name As %Name;

    Property DOB As %Date(FORMAT = 5);

    Property PatientID as %String;

    Property HomeAddress as Address;

    Property OtherAddress as AddressOtherNS ;
}
```

Address クラスは、同じ XML ネームスペース ("http://www.myapp.com") 内に存在するように定義されます。

OtherAddress クラスは、別な XML ネームスペース ("http://www.other.com") 内に存在するように定義されます。

Company クラスも、XML ネームスペース "http://www.myapp.com" 内に存在するように定義されます。

以下のように定義されます。

```
Class SchemaWriter.Company Extends (%Persistent, %XML.Adaptor)
```

```
{  
  
Parameter NAMESPACE = "http://www.myapp.com";  
  
Property Name As %String;  
  
Property CompanyID As %String;  
  
Property HomeOffice As Address;  
  
}
```

Person クラスと Company クラスを結び付けるプロパティ・リレーションシップはありません。

ネームスペース "http://www.myapp.com" に対するスキーマを生成するには、以下の構文を使用できます。

```
ClassMethod Demo()
{
    Set schema=##class(%XML.Schema).%New()
    Set schema.DefaultNamespace="http://www.myapp.com"
    Set status=schema.AddSchemaType("SchemaWriter.Person")
    Set status=schema.AddSchemaType("SchemaWriter.Company")
    Do schema.DefineLocation("http://www.other.com","c:/other-schema.xsd")

    Set schema=schema.GetSchema("http://www.myapp.com")

    //create writer
    Set writer=##class(%XML.Writer).%New()
    Set writer.Indent=1

    Do writer.AddSchemaNamespace()
    Do writer.AddNamespace("http://www.myapp.com")
    Do writer.AddNamespace("http://www.other.com")

    Set status=writer.DocumentNode(schema)
    If $$$ERROR(status) {Do $system.OBJ.DisplayError() Quit }
}
```

出力は、以下のとおりです。

```

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:s01="http://www.myapp.com"
xmlns:s02="http://www.other.com"
elementFormDefault="qualified"
targetNamespace="http://www.myapp.com">
  <import namespace="http://www.other.com" schemaLocation="c:/other-
schema.xsd"/>
  <complexType name="Person">
    <sequence>
      <element minOccurs="0" name="Name" type="s:string"/>
      <element minOccurs="0" name="DOB" type="s:date"/>
      <element minOccurs="0" name="PatientID" type="s:string"/>
      <element minOccurs="0" name="HomeAddress" type="s01:Address"/>
      <element minOccurs="0" name="OtherAddress" type="s02:AddressOtherNS"/>
    </sequence>
  </complexType>
  <complexType name="Address">
    <sequence>
      <element minOccurs="0" name="State">
        <simpleType>
          <restriction base="s:string">
            <maxLength value="2"/>
          </restriction>
        </simpleType>
      </element>
      <element minOccurs="0" name="Zip">
        <simpleType>
          <restriction base="s:string">
            <maxLength value="10"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>

```



```
</sequence>
</complexType>
<complexType name="Company">
  <sequence>
    <element minOccurs="0" name="Name" type="s:string"/>
    <element minOccurs="0" name="CompanyID" type="s:string"/>
    <element minOccurs="0" name="HomeOffice" type="s01:Address"/>
  </sequence>
</complexType>
</schema>
```

12 章 Web サービス

IRIS は、SOAP (Simple Object Access Protocol) の 1.1 および 1.2 をサポートしています。

12-1 IRIS Web サービスの概要

IRIS Web サービスの作成

IRIS では、次のいずれかの方法で Web サービスを作成できます。

既存のクラスに小さな変更をいくつか加えて、Web サービスに変換します。

この場合、引数として使用されているオブジェクト・クラスを変更て、%XML.Adaptor を拡張し、SOAP メッセージにパッケージ化できるようにする必要があります。

新規の Web サービス・クラスを最初から作成します。

IRIS SOAP ウィザードを使用して、既存の WSDL ドキュメントを読み取り、Web サービス・クラスおよびサポートするすべてのタイプ・クラスを生成します。

Web アプリケーションの一部としての Web サービス

IRIS Web サービス・クラスは、%SOAP.WebService クラスから継承され、その継承元クラスは %CSP.Page クラスから継承されます。

WSDL

IRIS のクラス・コンパイラは、Web サービスをコンパイルする際、そのサービス用の WSDL を生成し、利便性を向上するために、その WSDL を Web サーバ経由で公開します。

この WSDL は、WS-I (Web Services Interoperability Organization) によって確立された Basic Profile 1.0 に準拠しています。

IRIS では、WSDL ドキュメントが特定の URL で動的に処理され、(実行時に追加されるヘッダ要素以外にも) ユーザの Web サービス・クラスのインタフェースに対する変更のすべてが自動的に反映されます。

ほとんどの場合は、このドキュメントを使用して、Web サービスと相互運用する Web クライアントを生成できます。

12-2 IRIS Web クライアントの概要

IRIS Web クライアントの作成

IRIS では、IRIS SOAP ウィザードを使用して既存の WSDL ドキュメントを読み取ることで、Web クライアントを作成します。

このウィザードでは、Web クライアント・クラスおよびサポートするすべてのタイプ・クラスが生成されます。

生成された Web クライアント・インタフェースには、Web サービスで定義された各メソッドのプロキシ・メソッドを含むクライアント・クラスがあります。

各プロキシでは、対応する Web サービス・メソッドで使用するものと同じシグニチャが使用されます。

インタフェースには、メソッドの入出力として必要な XML タイプを定義するためのクラスも含まれています。

12-3 IRIS Web サービスの作成

IRIS で Web サービスを作成するには、`%SOAP.WebService` を拡張するクラスを作成します。

このクラスによって、**SOAP** プロトコル経由で呼び出し可能な 1 つ以上のメソッドの作成に必要なすべての機能が提供されます。

さらに、このクラスは **SOAP** に関連するブックキーピング (サービスを記述している **WSDL** ドキュメントの管理など) の管理を自動化します。

このクラスは、`%CSP.Page` クラスから派生します。

したがって、すべての Web サービス・クラスは **HTTP** 要求に応答できます。

`%SOAP.WebService` クラスは、以下を実行するための **HTTP** イベントに応答するメソッドを実装します。

Web サービスの **WSDL** ドキュメントを、**XML** ドキュメントとして発行します。

Web サービスとそのメソッドについて説明するカタログ・ページを、(**HTML** を使用して) 人が読める形式で発行します。

このページの説明では、クラス定義に含まれるコメントが示されます。

基本要件

IRIS で Web サービスを作成および発行するには、次の基本要件を満たす IRIS クラスを作成してコンパイルします。

クラスは `%SOAP.WebService` を拡張したものである必要があります。

クラスによって `SERVICENAME` パラメータを定義する必要があります。

このパラメータが定義されていないクラスは、IRIS でコンパイルされません。

このクラスによって、`WebMethod` キーワードを使用してマークを付けるメソッドまたはクラス・クエリを定義する必要があります。

Web メソッドの場合、メソッド・シグニチャ内の各値に XML プロジェクションがあることを確認してください。

例えば、メソッドに以下のシグニチャがあったとします。

`Method MyWebMethod(myarg as ClassA) as ClassB [WebMethod]`

この場合、`ClassA` と `ClassB` の両方に XML 表現が含まれている必要があります。

つまり、ほとんどの場合、これらのスーパークラス・リストに `%XML.Adaptor` が含まれていなければなりません。

IRIS SOAP サポートは、このリストの後に示すように、コレクションおよびストリームに対して特別な処理を行います。

Web メソッドは、一般メソッドと同じ方法で `ByRef` キーワードと `Output` キーワードを指定できます。

これらの引数および返り値内に含まれる可能性のある値について考えてみましょう。

XML では、印字不能文字、特に **ASCII 32** 未満の文字は許可されません (キャリッジ・リターン、改行、およびタブは **XML** で許可されるので例外です)。

許可されない印字不能文字を含める必要がある場合、タイプを **%Binary**、(同等の) **%xsd.base64Binary**、またはサブクラスとして指定します。

この値は、**XML** にエクスポートされるときに、**Base 64** のエンコードに自動的に変換されます (またはインポートされるときに、**Base 64** のエンコードから自動的に変換されます)。

引数の既定の値を指定する場合に、メソッド・シグニチャを利用しないでください。

メソッド・シグニチャを利用すると、既定値は無視され、代わりに **NULL** 文字列が使用されます。

例えば、以下のメソッドを考えてみます。

```
Method TestDefaults(val As %String = "Default String") As %String [ WebMethod ]
```

このメソッドを **Web** メソッドとして呼び出す場合に、引数を指定しないと **NULL** 文字列が使用され、値 **"Default String"** は無視されます。

その代わりに、メソッドの実装の最初に、値をテストし、必要に応じて目的の既定を使用します。1 つの手法は、以下のとおりです。

```
if arg="" {
    set arg="Default String"
}
```

通常通り、メソッド・シグニチャで既定値を示すことができますが、これは情報を提供することのみを目的とし、SOAP メッセージには影響を与えません。

Web メソッドで必要なすべての引数に対し、メソッド・シグニチャ内で **REQUIRED** プロパティ・パラメータを指定します。

以下はその例です。

`Method MyWebMethod(myarg as ClassA(REQUIRED=1)) as ClassB [WebMethod]`

既定では、継承されたメソッドはすべて、スーパークラスによって Web メソッドとしてマーク付けされていても、一般メソッドとして扱われます。

入力または出力としての結果セットの使用法

結果セットは入力または出力として使用できますが、その方法は対象の Web クライアントによって異なります。

Web サービスと Web クライアントの両方が IRIS を基盤とする場合や、いずれかが .NET を基盤とする場合、特殊な結果セット・クラスである %XML.DataSet を使用できます。

または、クラス・クエリを Web メソッドとして使用することもできます。

XML 表現は自動的に %XML.DataSet と同じになります。

Java ベースの Web クライアントでできるようにクエリの結果を出力するには、%ListOfObjects サブクラスを使用します。USER ネームスペースの SOAP.Demo に例があります。

簡単な例

このセクションでは、Web サービスの例、および Web サービスが認識できる要求メッセージと対応する応答メッセージの例を示します。

まず、Web サービスは以下のとおりです。

```
/// MyApp.StockService
Class MyApp.StockService Extends %SOAP.WebService
{

    /// Name of the WebService.
    Parameter SERVICENAME = "StockService";

    /// TODO: change this to actual SOAP namespace.
    /// SOAP Namespace for the WebService
    Parameter NAMESPACE = "http://tempuri.org";

    /// Namespaces of referenced classes will be used in the WSDL.
    Parameter USECLASSNAMESPACES = 1;

    /// This method returns tomorrow's price for the requested stock
    Method Forecast(StockName As %String) As %Integer [WebMethod]
    {
        // apply patented, nonlinear, heuristic to find new price
        Set price = $Random(1000)
        Quit price
    }
}
```

Web クライアントからこのメソッドを呼び出すと、クライアントによって SOAP メッセージが Web サービスに送信されます。

この SOAP メッセージは以下のようになります (改行とスペースが、読みやすいように追加してあります)。

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:s='http://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
    <ForecastResponse xmlns="http://www.myapp.org">
      <ForecastResult>799</ForecastResult>
    </ForecastResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Web サービスの生成

Web サービス・ウィザードの使用法

Web サービス・ウィザードでは、簡単なスタブが生成されます。

[ファイル]→[新規作成] をクリックします。

[新規作成] ダイアログ・ボックスが表示されます。

[一般] タブをクリックします。

[新規 Web サービス] をクリックしてから [OK] をクリックします。

ウィザードが表示されます。

パッケージ名、クラス名、および Web サービス名の値を入力します。

これらの入力は必須です。

オプションで、ネームスペースの URI を編集します (またはこの初期値を後で変更します)。

これは、IRIS ネームスペースではなく、XML ネームスペースです。

オプションで、別の行にメソッド名のリストを入力します。

[OK] をクリックします。

Web メソッドのスタブを含む新しい Web サービス・クラスが作成されました。

以下はその例です。

```
/// MyApp.StockService
Class MyApp.StockService Extends %SOAP.WebService
{

    /// Name of the WebService.
    Parameter SERVICENAME = "StockService";

    /// TODO: change this to actual SOAP namespace.
    /// SOAP Namespace for the WebService
    Parameter NAMESPACE = "http://tempuri.org";

    /// Namespaces of referenced classes will be used in the WSDL.
    Parameter USECLASSNAMESPACES = 1;

    /// TODO: add arguments and implementation.
    /// Forecast
    Method Forecast() As %String [ WebMethod ]
    {
        ;Quit "Forecast"
    }

}
```

カタログおよびテスト・ページについて

Web サービス・クラスをコンパイルすると、クラス・コンパイラによって便利なカタログ・ページが作成されます。

そのカタログ・ページを使用して、Web サービスを検証することができます。

このカタログ・ページは簡易テストのページにリンクしています。

これらのページにアクセスできるようにウェブ・アプリケーションを構成するには、ターミナルを開いて %SYS ネームスペースに移動し、次のコマンドを入力します。

```
set ^SYS("Security","CSP","AllowClass",webapplicationname,"%SOAP.WebServiceInfo")=1
set ^SYS("Security","CSP","AllowClass",webapplicationname,"%SOAP.WebServiceInvoke")=1
```

webapplicationname は、末尾にスラッシュを付けた Web アプリケーションの名前です。

例えば、"/csp/mynamespace/" のようになります。

これにより、既定で、/csp/user Web アプリケーションにアクセスできます。

また、これらのページは、%Development リソースに対する USE 特権を持つユーザとしてログインした場合にのみ使用できます。

これらの CSP ページを表示するには、以下の操作を実行します。

スタジオで、Web サービス・クラスを表示します。

[ビュー]→[ブラウザで表示] をクリックします。

カタログ・ページがすぐに表示されます。URL は以下のような構造になっています。

`base/csp/app/web_serv.cls`

ここで、`base` は Web サーバのベース URL です (必要に応じてポートを含む)。

`/csp/app` は Web サービスが格納されている Web アプリケーションの名前です。

`web_serv` は Web サービスのクラス名です (通常、`/csp/app` は `/csp/namespace` です)。以下はその例です。

`http://localhost:52780/csp/user/MyApp.StockService.cls`

このページは以下ようになります。

WSDL の表示

Web サービスの WSDL を表示するには、次の URL を使用します。

```
base/csp/app/web_serv.cls?WSDL
```

ここで、base は Web サーバのベース URL です (必要に応じてポートを含む)。

/csp/app は Web サービスが格納されている Web アプリケーションの名前です。

web_serv は Web サービスのクラス名です (通常、/csp/app は /csp/namespace です)。

注意:

クラス名内のパーセント記号 (%) は、この URL ではアンダースコア文字 () に置き換えられます。

以下はその例です。

```
http://localhost:52780/csp/user/MyApp.StockService.cls?WSDL
```

WSDL の生成

WSDL は静的ドキュメントとしても生成できます。%SOAP.WebService クラスには、そのために使用できるインスタンス・メソッドがあります。

12-4 Web クライアントの作成

Web クライアントは、Web サービスにアクセスするソフトウェアです。

Web クライアントには、一連のプロキシ・メソッドが用意されています。

各プロキシ・メソッドは、Web サービスのメソッドに対応しています。

プロキシ・メソッドは、対応する Web サービス・メソッドと同じシグニチャを使用し、必要に応じて Web サービス・メソッドを呼び出します。

この章では、IRIS で Web クライアントを作成し、使用方法を説明します。

SOAP ウィザードの概要

IRIS Web クライアントを作成するには、IRIS スタジオの SOAP ウィザード、または IRIS で提供されている対応するクラス・メソッドを使用します。

どちらの場合も、入力は WSDL ドキュメントです。

このツールでは、Web クライアント・クラスおよびサポートするすべての必要なクラスが生成されます。

SOAP ウィザードの使用法

スタジオに提供される SOAP ウィザードでは、特定の Web サービス、そのサービスの特定の WSDL のクライアントを生成できます。

SOAP ウィザードを使用するには：

[ツール]→[アドイン]→[SOAP ウィザード] をクリックします。

最初の画面で、次の手順を実行します。

WSDL の場所に応じて、[URL] または [FILE] のいずれかをクリックします。

WSDL の場所を入力します。WSDL の完全な URL または完全パスとファイル名のいずれかを入力してください。

生成された Web クライアント・クラスの使用法

IRIS Web クライアント・クラスを生成した後で、そのクラスを編集することはありません。

その代わりに、その Web クライアントのインスタンスを作成するコードと、クライアント側のエラー処理を提供するコードを記述します。

このコードでは、以下の処理を実行します。

Web クライアント・クラスのインスタンスを作成します。

インスタンスのプロパティを設定します。

ここでは、以下のような項目を制御できます。

Web クライアントのエンドポイント (Web クライアントで使用する Web サービスの URL)。制御するには、Location プロパティを設定します。

これは、Web クライアント・クラスの LOCATION パラメータをオーバーライドします。

プロキシ・サーバを指定する設定。

基本の HTTP ユーザ認証を制御する設定。

必要に応じて、Web クライアントのメソッドを呼び出します。

クライアント側のエラー処理を実行します。

オプションで、Web クライアントが受け取る HTTP 応答を検証します。

ターミナルのセッションからの簡単な例を以下に示します。

```
GSOAP>set client=##class(Proxies.CustomerLookupServiceSoap).%New()
```

```
GSOAP>set resp=client.GetCustomerInfo("137")
```

```
GSOAP>w resp
```

```
11@Proxies.CustomerResponse
```

```
GSOAP>w resp.Name
```

```
Smith,Maria
```

13 章 その他

13-1 ファイル IO

IRIS にはファイルから Read/Write する方法に関して、以下の複数の方法を用意しています。

- OPEN, USE, READ, WRITE コマンドを使用する
- %Library.File クラスを使用する
- %Stream.FileCharacter, %Stream.FileBinary クラスを使用する

どのインタフェースを推奨するということは特にありませんが、%Stream クラスの利用が一番簡便な方法だと思います。

ただし、性能が要求される場合には、コマンドを使用する方法が最速です。

以下は%Stream.FileCharacter の使用例です。

```
ClassMethod ExportXML(pFile As %String) As %Status

{
    set rc = 1
    Try {
        Set File = ##class(%Stream.FileCharacter).%New()
        Do File.FilenameSet(pFile)
        Do File.TranslateTableSet("UTF8")
        Do File.WriteLine("<?xml version='\"1.0\"' encoding='\"UTF-8\"' ?>")
        Do File.WriteLine("<Export>")
        For count=1:1:10 {
            Set sc=..%OpenId(count)
            Set rc= sc.XMLExportToStream(.File)
            Set sc=""
        }
        Do File.WriteLine("</Export>")
        Set sc = File.%Save()
        Set File = ""
    }
    Catch error {
        Write error.Name
    }
    Quit rc
}
```

13-2 インターネットライブラリー

IRIS にはインターネット関連の様々なプロトコルに対応した以下のクラスライブラリーが用意されています。

%Net.HttpRequest	HTTP の要求
%Net.MailMessage	電子メールメッセージ
%Net.SMTP	電子メールの送信
%Net.POP3	電子メールの受信
%Net.FtpSession	FTP
%Net.FtpCallback	FTP
%Net.SSH	SSH

詳細は、以下を参照してください。

<http://docs.intersystems.com/irislatestj/csp/docbook/DocBook.UI.Page.cls?KEY=GNET>

13-3 %Populate クラス

IRIS は、テストデータを生成するフレームワークを用意しています。

詳細は、以下を参照してください。

http://docs.intersystems.com/irislatestj/csp/docbook/DocBook.UI.Page.cls?KEY=GORIENT_ch_devtasks#GORIENT_devtasks_populate

13-4 UNITTEST

テストを自動化するためのフレームワークが用意されています。

詳細は以下を参照してください。

<http://docs.intersystems.com/irislatestj/csp/docbook/DocBook.UI.Page.cls?KEY=TUNT>