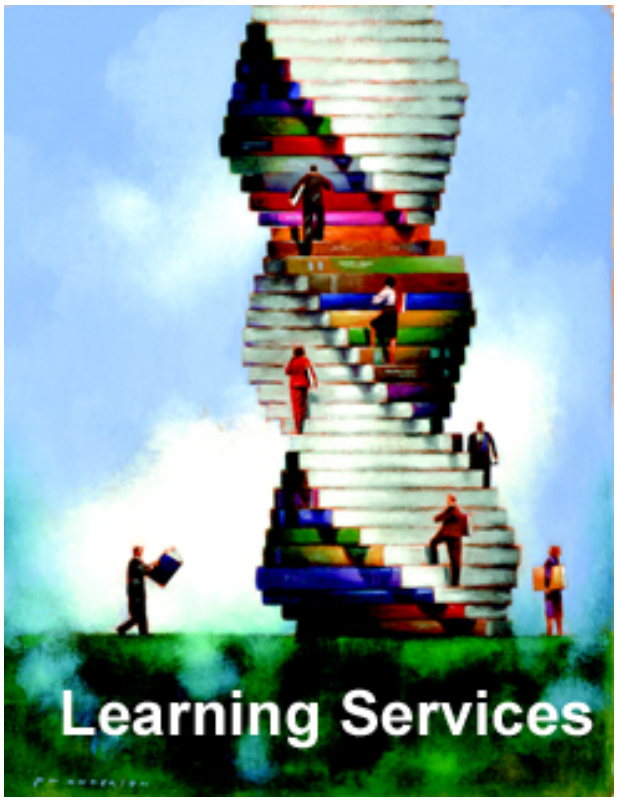


# 統一データアーキテクチャ

（Caché Version 2010.2 ベース）

V1.0

2011 年 3 月

インターシステムズジャパン株式会社

**目次**:



**図表目次**:

統一データアーキテクチャ

Caché Version 2010.2 2011 年 3 月 24 日

## 1. はじめに

Caché の強力でユニークな特徴は、その統一データアーキテクチャにあります。
このアーキテクチャにより、Caché に格納されたデータに対して、オブジェクトとリレーショナルの高性能なアクセスを同時に実現します。この文書の内容を十分に理解するには、Caché の多次元データエンジンを理解している必要があります。多次元データエンジンに関しては、別文書『多次元データエンジンの概念とアーキテクチャ』ご参照下さい。

## 1.1. 統一データ辞書

Caché の中で、アプリケーションコンポーネントをオブジェクトとしてモデル化することができます。オブジェクトは、クラスとして体系付けられ、クラスは、オブジェクトのデータ（プロパティ）と振る舞い（メソッド）を定義します。
各クラスのメタ情報あるいは定義は、Caché クラス辞書と呼ばれる共通のリポジトリ内に格納されます。
クラス辞書自身も Caché の中に格納されるオブジェクトデータベースで、オブジェクトインタフェースを使用してその内容にアクセスできます。
クラス辞書は、クラスコンパイラによって、永続オブジェクトに必要な格納構造を定義します。
そしてクラス定義を、この格納構造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の両方を提供する 2 種類の一連の実行コードに変換します。

このアーキテクチャにより、そのオブジェクトとリレーショナルコードのデータアクセス経路は、効率が良く、自動的にお互いに同期します。

クラス定義は、クラス辞書にいくつかの方法で追加することができます。

- Caché スタジオの開発環境を使用して会話的に。
- DDL を使用し、リレーショナル的に。
  Caché は、標準 SQL DDL 文を受け付け、自動的に対応するクラス定義とテーブル定義を生成します。
- XML を使用し、テキストとして。
  Caché は、クラス定義の外部 XML 表現をサポートしています。これは、ソースコード管理、配備、自動コード生成、他ツールとの連携に使用します。
- オブジェクトインタフェースを使用したプログラミングによって。
  一連の Caché クラス定義オブジェクトを使用して、クラス辞書と直接連携するプログラムを作成することができます。またアプリケーションの実行時に新しいクラスを作成することができます。
- Caché スタジオに含まれる XML スキーマウィザードと使用して。
  大抵の XML スキーマファイルからクラス定義を作成することができます。

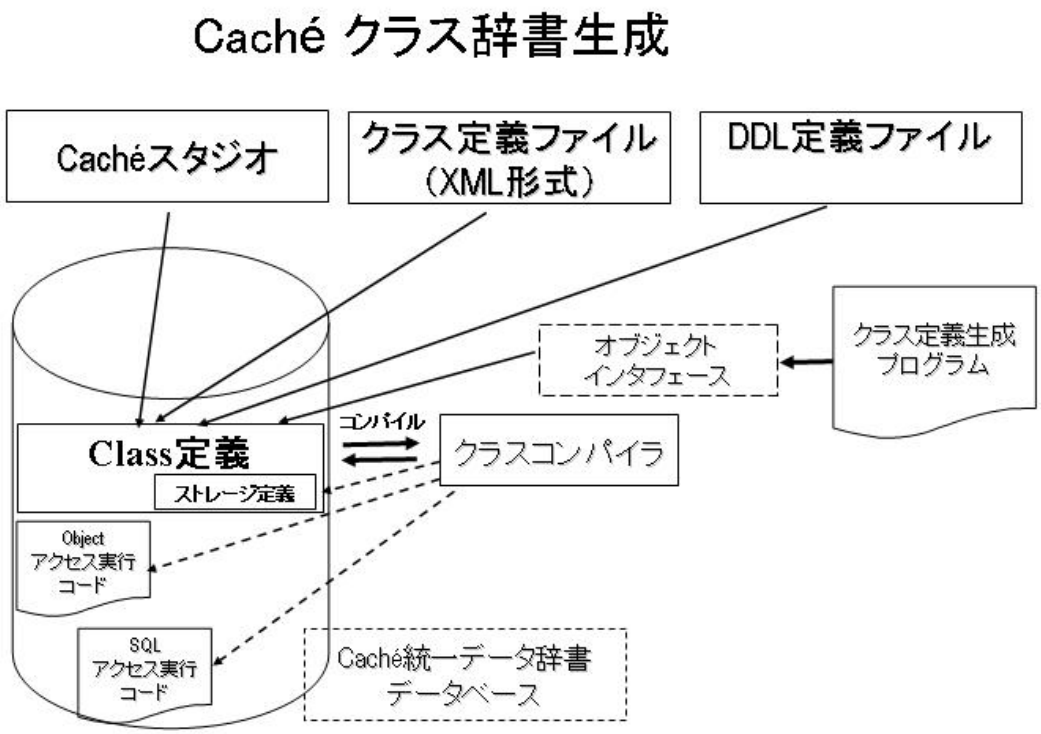


**図 1　Caché クラス辞書イメージ図**

## 1.2. 柔軟なストレージ

Caché のオブジェクトモデルは、プログラミング言語のオブジェクトモデルとは多少異なっており、それはプロパティとメソッドのサポートに加え、ストレージに関連する振る舞い、つまりインデックスや制約、ストレージ構造を定義することができます。
永続オブジェクトが使用するストレージ構造は、クラスの論理定義から独立しており、非常に柔軟性があります。開発者は、通常クラスコンパイラが提供する既定の構造を使うことができます。
または、特定のケースに合わせて、その構造を調整することもできます。Caché SQL ゲートウェイを使用して、外部リレーショナルデータベースにデータを格納するクラスを持つこともできます。

## 2. SQL およびオブジェクトによる多次元ストレージの利用

ここでは、Caché オブジェクトと SQL エンジンが永続オブジェクト、リレーショナルテーブル、インデックスを格納するために、多次元ストレージ（グローバル）を利用する方法について説明します。

Caché オブジェクトと SQL エンジンは、自動的にデータ構造を提供、管理しますが、それがどの様に動作するかの詳細を理解するのは、いろいろと役に立ちます。

オブジェクトとリレーショナルのデータビューが使用するストレージ構造は、全く同じです。
ここでは、話を単純化するため、オブジェクトの観点からストレージを説明します。

%CacheStorage ストレージクラス（既定値）を使用する全ての永続クラスは、Caché データベースの中に 1 つ以上の多次元ストレージ（グローバル）を使用して、それ自身のインスタンスを格納することができます。

全ての永続クラスは、そのプロパティがグローバルノード内にどの様に格納されるのかを定義するストレージ定義を持ちます。このストレージ定義は（既定構造と呼ばれます）クラスコンパイラによって自動的に管理されます。このストレージ定義を変更することも、まったく違う定義を用意することも可能ですが、その内容は、この文書の範囲を超えるので、ここでは、その方法については、説明しません。

## 2.1. 既定の構造

永続オブジェクトの格納に使用する既定の構造は、非常に単純です。
データは、完全なクラス名（パッケージ名も含む）で始まるグローバルに格納します。データグローバルの名前には"D"がその名前の最後に付きます。インデックスグローバルには、"I"がその名前の最後に付きます。

各インスタンスのデータは、そのデータグローバルの1つのノードの中に格納します。そして、全ての永続プロパティは、$List 構造の中に配置します。

データグローバルの中の各ノードは、オブジェクト ID 値の添え字を持ちます。既定では、オブジェクト ID は、そのデータグローバルのルート（添え字なし）に格納しているカウンタノードに対して $Increment 関数を実行することによって得られる整数値です。

例えば、簡単な永続クラス、2 つのリテラルプロパティがある MyApp.Person を定義します。

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
}
```

このクラスの 2 つのインスタンスを作成、保存すると、結果のグローバルは、以下のようになります。

```
^MyApp.PersonD = 2 // カウンタノード
^MyApp.PersonD(1) = $LB("",53,"Abraham")
^MyApp.PersonD(2) = $LB("",68,"Noah")
```

各ノードに格納される$List 構造の最初の部分は、空白であることに着目して下さい。これは、クラス名に予約してあります。この Person クラスのサブクラスを定義すると、このスロットは、そのサブクラス名を含みます。同じエクステントの中に複数種類のオブジェクトが格納されている時に%OpenId()メソッド（%Persistent クラスが提供）は、この情報を使用し、多態的に正しいタイプのオブジェクトをオープンします。
このスロットは、そのクラスストレージ定義の中で"%%CLASSNAME"という名前のプロパティとしてあらわれます。

## 2.2. IDKEY

IDKEY メカニズムによって、オブジェクト ID に使用する値を明示的に定義することができます。
そうするには、単にクラスに IDKEY インデックス定義を追加し、その ID 値を提供する 1 つ以上のプロパティを定義するだけです。
オブジェクトを一度保存すると、そのオブジェクト ID は、変更できない点、注意が必要です。
これは、IDKEY メカニズムを使用したオブジェクトを一度保存すると、そのオブジェクト ID を構成するプロパティを変更できないことを意味します。
例えば、以前の例で使用した Person クラスで IDKEY インデックスを使用するように変更できます。

```
Class MyApp.Person Extends %Persistent
{
Index IDKEY On Name [ Idkey ];
Property Name As %String;
Property Age As %Integer;
}
```

Person クラスの 2 つのインスタンスを作成、保存すると、結果のグローバルは、以下のようになります。

```
^MyApp.PersonD("Abraham") = $LB("",53,"Abraham")
^MyApp.PersonD("Noah") = $LB("",68,"Noah")
```

最早カウンタノードが定義されていないことに注意して下さい。
オブジェクト ID は、Name プロパティを基準にしていますので、各オブジェクトの Name の値は、ユニークであるという仮定を置いていることにも注意して下さい。

IDKEY インデックスが、複数プロパティを基準としていると、主データノードは、複数サブスクリプトを持ちます。

例えば、

```
Class MyApp.Person Extends %Persistent
{
Index IDKEY On (Name,Age) [ Idkey ];
Property Name As %String;
Property Age As %Integer;
}
```

この例では、結果のグローバルは、以下のようになります。

```
^MyApp.PersonD("Abraham",53) = $LB("",53,"Abraham")
^MyApp.PersonD("Noah",68) = $LB("",68,"Noah")
```

## 2.3. サブクラス

既定では、永続オブジェクトのサブクラスによって導入されるフィールドは、追加ノードに格納されます。そのサブクラスの名前がその追加の添え字値として使われます。
例えば、簡単な永続クラス、2 つのリテラルプロパティがある MyApp.Person を定義します。

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
}
```

次に 2 つの追加のリテラルプロパティを持つ永続サブクラス MyApp.Student を定義します。

```
Class MyApp.Student Extends Person
{
Property Major As %String;
Property GPA As %Float;
}
```

この MyApp.Student クラスの 2 つのインスタンスを作成、保存すると、結果のグローバルは、以下のようになります。

```
^MyApp.PersonD = 2 // counter node
^MyApp.PersonD(1) = $LB("Student",19,"Jack")
^MyApp.PersonD(1,"Student") = $LB(3.2,"Physics")
^MyApp.PersonD(2) = $LB("Student",20,"Jill")
^MyApp.PersonD(1,"Student") = $LB(3.8,"Chemistry")
```

Person クラスから継承したプロパティは主ノードに格納します。Student クラスで定義したプロパティは追加のサブノードに格納します。この構造は、Student データは、Person データとして相互に交換可能になることを保証します。例えば、全ての Person オブジェクトをリストする SQL クエリは、正しく Person と Student データの両方を収集します。またこの構造は、プロパティがそのスーパクラスとサブクラスのいずれかに追加されるので、クラスコンパイラがデータ互換性を維持するのも簡単になります。
主ノードの最初の部分は、文字列"Student"を含むことに着目して下さい。これは、これらのノードが、Student データを含んでいることを示唆しています。

## 2.4. 親子リレーションシップ

親子リレーションシップの中では、子オブジェクトのインスタンスは、それが属する親オブジェクトのサブノードとして格納します。この構造は、子のインスタンスデータが親データと伴に物理的にクラスタ化することを保証します。

例えば、2 つの関連するクラス定義があります。

```
/// Invoice クラス
Class MyApp.Invoice Extends %Persistent
{
Property CustomerName As %String;
/// 1 つの Invoice は、 LineItem の CHILDREN を持つ
Relationship Items As LineItem [inverse = TheInvoice, cardinality = CHILDREN];
}

/// LineItem クラス
Class MyApp.LineItem Extends %Persistent
{
Property Product As %String;
Property Quantity As %Integer;
/// 1 つの LineItem は、Invoice の PARENT を持つ
Relationship TheInvoice As Invoice [inverse = Items, cardinality = PARENT];
}
```

Invoice オブジェクトの複数インスタンスを格納すると、関連する LineItem オブジェクトの各グローバルは、以下の様になります。

```
^MyApp.InvoiceD = 2 // invoice カウンタノード
^MyApp.InvoiceD(1) = $LB("","Wiley Coyote")
^MyApp.InvoiceD(1,"Items") = 2 // lineitem カウンタノード
^MyApp.InvoiceD(1,"Items",1) = $LB("","Rocket Roller Skates",2)
^MyApp.InvoiceD(1,"Items",2) = $LB("","Acme Magnet",1)
^MyApp.InvoiceD(2) = $LB("","Road Runner")
^MyApp.InvoiceD(2,"Items") = 1 // lineitem カウンタノード
^MyApp.InvoiceD(2,"Items",1) = $LB("","Birdseed",30)
```

Relationship の名前が追加のリテラル添え字に使われていることに着目して下さい。
こうすることによって、クラスが複数のリレーションシップをデータの衝突なしにサポートできます。
また、Invoice の各インスタンスは、LineItem オブジェクトの ID 値を割り当てるためにその自分自身のカウンタを維持することにも着目してください。

## 2.5. 埋め込みオブジェクト

埋め込みオブジェクトは、最初にシリアライズ（直列）化された状態に変換して格納します。（既定では、そのオブジェクトプロパティを含む$List 構造）それから、この直列な状態を他のプロパティと同じ方法で格納します。

例えば、2 つのリテラルプロパティを持つ簡単なシリアル（埋め込み可能）クラスを定義します。

```
Class MyApp.MyAddress Extends %SerialObject
{
Property City As %String;
Property State As %String;
}
```

以前の例を修正して埋め込み Home address プロパティを追加します。

```
Class MyApp.MyClass Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
Property Home As MyAddress;
}
```

このクラスの 2 つのインスタンスを作成、保存すると、結果のグローバルは、以下のようになります。

```
^MyApp.MyClassD = 2 // カウンタノード
^MyApp.MyClassD(1) = $LB(53,"Abraham",$LB("UR","Mesopotamia"))
^MyApp.MyClassD(2) = $LB(68,"Noah",$LB("Gommorrah","Israel"))
```

## 2.6. Streams

グローバルストリームは、一連のかたまりにデータを分割することによりグローバル内に格納します。そのかたまりの1つは、32K バイト以下です。その複数のかたまりは、連続するノードに書きます。(ファイルストリームは、外部ファイルに格納します)

## 2.7. インデックス

永続クラスは、1 つ以上のインデックスを定義できます。操作をより効率的にするため（ソートや条件検索など）に追加のデータ構造を使用します。Caché SQL は、クエリを実行するときにそれらのインデックスを使用します。Caché オブジェクトと SQL は、挿入、更新、削除処理を実行するときにインデックス内の正しい値を自動的に維持管理します

### 2.7.1. 標準インデックスのストレージ構造

標準インデックスは、該当プロパティを含むオブジェクトのオブジェクト ID 値と 1 つ以上の並べ替えられたプロパティ値を関連付けます。

例えば、2 つのプロパティを持ち、その Name プロパティがインデックスを持つ簡単な永続 MyApp.Person クラスを定義してみましょう。

```
Class MyApp.Person Extends %Persistent
{
Index NameIdx On Name;
Property Name As %String;
Property Age As %Integer;
}
```

この Person クラスのいくつかのインスタンスを作成、保存すると、その結果のデータとインデックスは、以下のようになります。

```
// データグローバル
^MyApp.PersonD = 3 // カウンタノード
^MyApp.PersonD(1) = $LB("",34,"Jones")
^MyApp.PersonD(2) = $LB("",22,"Smith")
^MyApp.PersonD(3) = $LB("",45,"Jones")
// インデックスグローバル
^MyApp.PersonI("NameIdx"," JONES",1) = ""
^MyApp.PersonI("NameIdx"," JONES",3) = ""
^MyApp.PersonI("NameIdx"," SMITH",2) = ""
```

インデックスグローバルについて以下の点も注意して下さい。

1. 既定では、 その名前がクラス名＋最後に"I"（インデックス用）が付く名前のグローバルを配置します。
2. 既定では、 その 1 番目の添え字名が、インデックス名です。このことは、衝突することなしに同じグローバルに複数のインデックスを格納することができます。
3. 2 番目のサブスクリプトは、照合順に並んだデータ値を含みます。この場合、そのデータは、既定の SQLUPPER 照合関数を使って並べ替えられます。これは、全ての文字を大文字に変換し（大文字小文字を意識せずにソートするため）、1 つの空白文字を先頭に追加します。（全てのデータが文字列として並ぶことを強制するため）
4. 3 番目の添え字は、そのオブジェクトのオブジェクト ID を含みます。それは、インデックス化したデータ値を含みます。
5. ノード自身は、空です。全ての必要なデータは、その添え字の中に保持しています。
   インデックスの定義が、そのインデックスとともにデータも格納するように指定する場合、そのインデックスグローバルのノード内に置かれることに注意して下さい。

このインデックスは、Name 順に並んだ全ての Person クラスのリスティングのような様々なクエリを満足させるための十分な情報を含んでいます。

## 3. 統一データアーキテクチャ上での実行コード生成の流れ

以下では、実際に統一データアーキテクチャに基づき、どの様に実行コードが生成されるかを説明します。

### 3.1. 統一データ実行アーキテクチャ

コード生成の流れを図にすると以下の様になります。

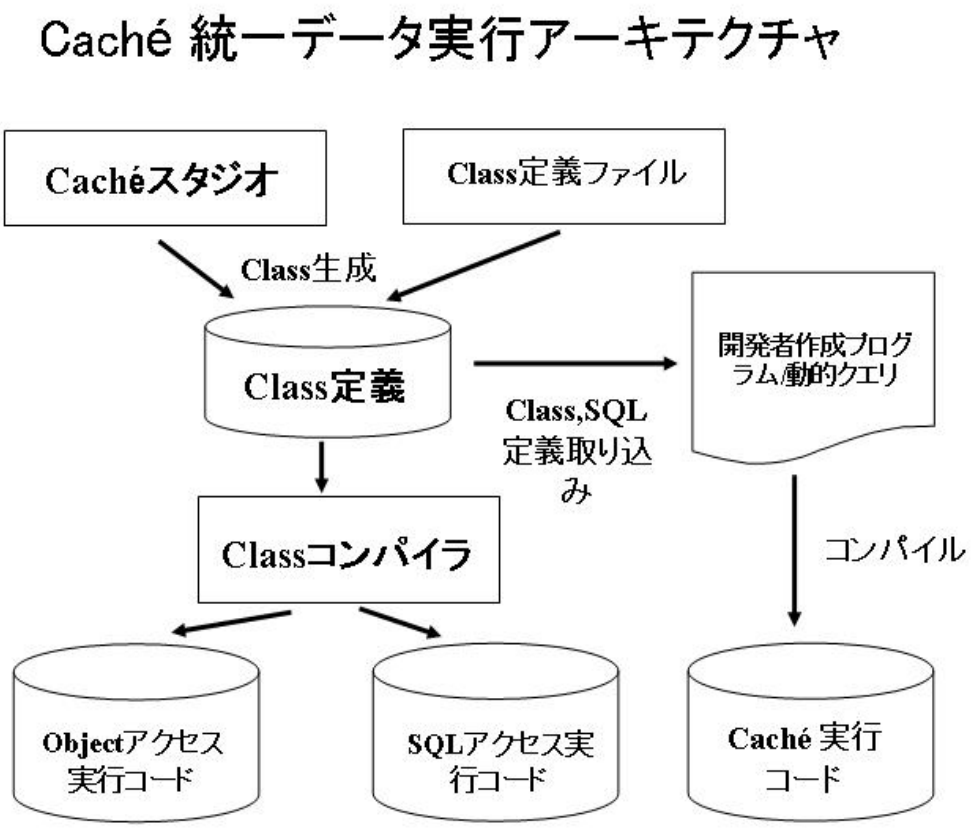


**図 2 統一データ実行アーキテクチャ**

## 3.2. 統一データアーキテクチャ実行コード生成の流れ

- Caché スタジオ上でクラス定義を実行し、保存するまたは、クラス定義ファイル（XML 形式）をインポートすると、クラス生成が行われ、クラス辞書にそのクラスが登録される。

- コンパイルを実行すると、そのクラスをオブジェクトインタフェースでアクセスするための実行コード、および SQL アクセスするための実行コードが生成される。（MAC->INT->OBJ これは、スタジオの表示メニュ->他の表示で内容確認できる）

- 開発者が作成したプログラムの中で、SQL アクセスを行う場合、クラス定義を参照し、そのストレージで定義したグローバル構造にアクセスするコードを自動生成する。また、JDBC、ODBC 経由で発行された SQL クエリも Caché がクエリコードを自動生成し、そのクラス定義を参照し、グローバル構造にアクセスするコードを自動生成する。