



Caché 多次元ストレージ の使用方法

Version 5.1

2006-03-14

Caché 多次元ストレージの使用法

Caché Version 5.1 2006-03-14

Copyright © 2006 InterSystems Corporation.

All rights reserved.

このドキュメントは、Sun Microsystems、RenderX Inc.、アドビ システムズ および ワールドワイド・ウェブ・コンソーシアム (www.w3c.org) のツールと情報を使用して、Adobe Portable Document Format (PDF) で作成およびフォーマットされました。主要ドキュメント開発ツールは、InterSystemsが構築したCaché と Javaを使用した特別目的のXML処理アプリケーションです。



Caché 製品とロゴは InterSystems Corporation の登録商標です。



Ensemble 製品とロゴは InterSystems Corporation の登録商標です。



InterSystems という名前とロゴは InterSystems Corporation の登録商標です

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

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

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

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

Caché および InterSystems Caché、Caché SQL、Caché ObjectScript および Caché Object は、インターシステムズ社の商標です。

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

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

InterSystems ワールドワイド カスタマサポート

Tel: +1 617 621-0700

Fax: +1 617 374-9391

Email: support@InterSystems.com

目次

1 はじめに	1
1.1 機能	1
1.2 例	3
1.3 アプリケーションの使用法	3
2 グローバル構造	5
2.1 グローバルの論理構造	5
2.1.1 グローバルの名前付け規約	5
2.1.2 添え字の名前付け規約と制限	6
2.1.3 グローバル・データ	7
2.1.4 グローバル添え字	7
2.1.5 照合	8
2.2 グローバルの物理構造	8
2.2.1 グローバルの格納法	9
2.3 グローバルの参照	9
2.3.1 グローバル・マッピングの設定	10
2.3.2 拡張グローバル参照	10
3 多次元ストレージの使用法 (グローバル)	13
3.1 グローバルへのデータ格納	13
3.1.1 グローバルの生成	13
3.1.2 グローバル・ノードに格納するデータ	14
3.1.3 グローバル・ノードへの構造化されたデータ格納	14
3.2 グローバル・ノードの削除	16
3.3 グローバル・ノードの存在有無のテスト	17
3.4 グローバル・ノード値の検索	17
3.4.1 \$GET 関数	17
3.4.2 WRITE コマンド、ZWRITE コマンド、ZZDUMP コマンド	18
3.5 グローバル内のデータ検索	18
3.5.1 \$ORDER (Next / Previous) 関数	18
3.5.2 グローバルの反復	20
3.5.3 \$QUERY 関数	20
3.6 グローバル内でのデータのコピー	21
3.7 グローバルでの共有カウンタ保持	21
3.8 一時グローバル	22
3.9 グローバルでのデータのソート	23
3.9.1 グローバル・ノードの照合	23
3.9.2 数値添え字と文字列値添え字	24

3.9.3 \$SORTBEGIN 関数と \$SORTEND 関数	24
3.10 グローバルを使用した間接演算の使用法	25
3.11 トランザクション管理	26
3.11.1 ロックとトランザクション	27
3.11.2 入れ子にされた TSTART の呼び出し	28
3.12 並行処理の管理	29
3.13 最新のグローバル参照	29
3.13.1 ネイキッド・グローバル参照	29
4 多次元ストレージの SQL およびオブジェクトの使用法	33
4.1 データ	33
4.1.1 既定構造	33
4.1.2 IDKEY	34
4.1.3 サブクラス	35
4.1.4 親子リレーションシップ	36
4.1.5 埋め込みオブジェクト	37
4.1.6 ストリーム	37
4.2 インデックス	38
4.2.1 標準インデックスのストレージ構造	38
4.3 ビットマップ・インデックス	39
4.3.1 ビットマップ・インデックスの論理処理	39
4.3.2 ビットマップ・インデックスのストレージ構造	40
4.3.3 ビットマップ・インデックスの直接アクセス	41

テーブル一覧

ビット文字列処理	39
----------------	----

1

はじめに

多次元ストレージ・エンジンは、Cachéの主な機能の1つです。この機能により、アプリケーションはデータをコンパクトで効率的な多次元スパース配列に格納できます。このような配列は **グローバル** と呼ばれます。

ここでは以下の説明を行います。

- ・ グローバルの意味と実行可能なオペレーション
- ・ 分散データベース・アーキテクチャでのグローバルの使用法など、論理的かつ物理的 **グローバルの構造**
- ・ アプリケーションにデータを格納、検索するための **グローバルの使用法**
- ・ SQL とオブジェクト・エンジンでの **Caché によるグローバル使用法**

1.1 機能

グローバルは、永続多次元配列での簡単なデータ格納法を提供します。

例えば、設定 というグローバルを使用して、値 “赤” をキー “色” と関連付けます。

```
SET ^Settings("Color")="Red"  
WRITE !,^Settings("Color")
```

グローバルの多次元性を利用して、さらに複雑な構造の定義が可能となります。

```
^Settings("Auto1","Properties","Color") = "Red"  
^Settings("Auto1","Properties","Model") = "SUV"  
^Settings("Auto2","Owner") = "Mo"  
^Settings("Auto2","Properties","Color") = "Green"
```

グローバルには以下の機能があります。

はじめに

- ・ 使用の容易性 – グローバルは、他のプログラミング言語変数と同様に使用が簡単です。Cache ObjectScript 言語と Basic スクリプト言語のいずれにも包括的なコマンドがあるため、アプリケーションでのグローバルの使用が簡単です。
- ・ 多次元 – 任意の添え字を使用して、グローバルでノードのアドレスを指定できます。例えば `^ (" 2", " ", " ")` の場合、添え字 色 は 設定 グローバルで 3 番目のノードになります。添え字は、整数、数値、あるいは文字列値で、連続している必要はありません。
- ・ スパース – グローバル・ノードのアドレス指定をするための添え字は非常にコンパクトで、連続した値を持つ必要はありません。
- ・ 効率的 – グローバルでの処理 (挿入、更新、削除、走査、検索) はすべて、最高のパフォーマンスと並行処理のために高度に最適化されています。他にも、特定のオペレーションに対応するためのコマンドがあります (データの一括挿入など)。また、記録のソートなど一時的なデータ構造のための特別仕様グローバルもあります。
- ・ 信頼性 – Cache データベースはさまざまな機能を提供し、論理レベルと物理レベル両方のジャーナリングなど、グローバル内に格納されたデータの信頼性を確かなものにします。グローバル内に格納されるデータは、データベース・バックアップ・オペレーションが実行されるときにバックアップされます。
- ・ 分散型 – Cache により、グローバル内に格納されたデータの物理位置を制御することができます。グローバルの格納に使用する物理データベースを定義し、複数のデータベースにグローバルの一部を配布できます。Cache 分散型データベース機能を使用することで、データベース・ネットワークおよびアプリケーション・サーバ・システム間でグローバルを共有することができます。また、データ・シャドウイング・テクノロジーにより、システムのグローバル内に格納されたデータを、他のシステムに自動的に複製することができます。
- ・ 並行処理 – グローバルは、複数プロセス間の同時アクセスをサポートします。個別ノード (配列要素) 内の値の設定と取得は常にアトミックです。信頼性のある同時アクセスを保証するためのロックは必要ありません。また、Cache は強力なロック・オペレーションをサポートしており、複数のノードなど、より複雑な状況で並行処理を行うために使用できます。オブジェクトや SQL アクセス使用の際、この並行処理は自動的に処理されます。
- ・ トランザクション – Cache が提供するトランザクションの範囲を指定するコマンドで、トランザクションの開始、実行、ロールバックが可能です。ロールバックの際、トランザクション内でのグローバルの変更をすべて取り消し、データベースのコンテンツをトランザクション以前の状態にリストアします。Cache のさまざまなロック・オペレーションをトランザクションと併せて使用することで、グローバルを使用した従来の ACID トランザクションを実行できます。オブジェクトや SQL アクセス使用の際、トランザクションは自動的に処理されます。

1.2 例

グローバルの容易性とパフォーマンスは、簡単な例を取り上げるとわかります。以下のプログラムは、10,000 配列ノードを生成(既存ノードは先に削除)し、データベースに保存します。グローバルのパフォーマンスを実感するためにも、試してみましょう。

永続配列の生成

```
Set start = $ZH // get current time

Kill ^Test.Global
For i = 1:1:10000 {
    Set ^Test.Global(i) = i
}

Set elap = $ZH - start // get elapsed time
Write "Time (seconds): ",elap
```

配列内の値を繰り返し処理して読み込むまでに、どのくらいの時間がかかるのか見てみましょう(まず上記例を実行して、配列を構築してください)。

永続配列の読み込み

```
Set start = $ZH // get current time
Set total = 0
Set count = 0

// get key and value for first node
Set i = $Order(^Test.Global(""),1,data)

While (i != "") {
    Set count = count + 1
    Set total = total + data

    // get key and value for next node
    Set i = $Order(^Test.Global(i),1,data)
}

Set elap = $ZH - start // get elapsed time

Write "Nodes:          ",count,!
Write "Total:         ",total,!
Write "Time (seconds): ",elap,!
```

1.3 アプリケーションの使用法

Caché アプリケーション内において、グローバルはさまざまな方法で使用されます。以下はその例です。

- ・ オブジェクト、および SQL エンジンに共有される基本ストレージ・メカニズムとして。

はじめに

- ・ オブジェクト、および SQL データに対するビットマップ・インデックスなど、多様なインデックスを提供するために使用されるメカニズムとして。
- ・ プロセス・メモリに収まらない可能性のあるオペレーションを実行するためのワーク・スペースとして。例えば、用途に合った既存のインデックスがない場合、SQL エンジンではデータをソートするために一時的なグローバルを使用します。
- ・ オブジェクト、および SQL アクセスという点で、表現が困難もしくは非効率的な永続オブジェクト、または SQL テーブルでの特別なオペレーションを実行するため。例えば、メソッド (あるいはストアド・プロシージャや Web メソッド) を指定して、テーブルの特別なデータ分析を行うことが可能です。メソッドを使用することで、そのようなオペレーションは完全にカプセル化され、呼び出し側が単にメソッドを呼び出します。
- ・ アプリケーションに特化したカスタマイズされたストレージ構造を実装するため。多くのアプリケーションでは、リレーショナル形式で表現することが難しいデータを保存する必要がある場合があります。グローバルでカスタム構造を指定し、外部クライアントがその構造をオブジェクト・メソッド経由で利用できるようにします。
- ・ Caché システムで使用される、構成データ、クラス定義、エラー・メッセージ、実行可能なコードなど、特別な目的をもった多様なデータ構造のため。

グローバルは、リレーショナル・モデルの範囲に限定されません。特定のアプリケーションを最大限に利用するカスタマイズされた構造を自由に開発できます。多くのアプリケーションでグローバルを賢明に使用することにより、リレーショナル・アプリケーション開発者達が待ち望んだパフォーマンスを提供することができます。

アプリケーションがグローバルを直接使用するか否かに関わらず、そのオペレーションを理解しておくに役に立ちます。グローバルとその性能を理解することは、アプリケーションに最適な配置構成を決定するのに役立つのはもとより、更に効率的なアプリケーションを設計することにも役立ちます。

2

グローバル構造

この章では、グローバルの論理的 (プログラミングからの観点) な説明と、グローバルがどのようにディスクへ物理的に格納されるかの概要を説明します。

2.1 グローバルの論理構造

グローバルとは名前付きの多次元配列で、物理 Caché データベースに格納されます。アプリケーションで、物理データベースへのグローバルのマッピングは、現在のネーム・スペースを基に行います。ネーム・スペースは、1 つ以上の物理データベースの論理的な統一ビューを提供しています。

2.1.1 グローバルの名前付け規約

- ・ グローバル名はキャレット文字 (^) で開始します。キャレット文字は、ローカル変数とグローバルを区別するために使用します。
- ・ グローバル名の長さは 31 文字までです(接頭のキャレット文字を除く)。それ以上に長い名前を付けることも可能ですが、Caché は 31 文字しかグローバル名として処理しません。
- ・ グローバル名は、大文字と小文字を区別します。
- ・ グローバル名の先頭にあるキャレット文字の直後には、必ず文字かパーセント記号 (%) がきます。“%” 記号で始まるグローバル名は特殊システム・グローバルで、通常は %SYS もしくは %CACHELIB データベースに格納されます。
- ・ 他のグローバル名の文字は、文字、数字、ピリオド記号 (.) になります。パーセント記号 (%) は、グローバル名の最初の文字として以外には使用できません。またピリオド記号 (.) は、グローバル名の最後の文字としては使用できません。

注釈: グローバル名は、上述のように既定で、適切な識別子文字のみ使用できます。しかし、NLS (各国言語サポート) の設定により、異なる識別子文字を定義する場合があります。ただし、グローバル名に Unicode 文字を使用できません。

したがって、以下はすべて有効なグローバル名です。

```
SET ^a="The "  
SET ^A="quick "  
SET ^%A="brown "  
SET ^A7="fox "  
SET ^A.7="jumped over "  
SET ^A7..7="the lazy "  
SET ^A1B2C3="dog's back."  
WRITE ^a,^A,^%A,^A7,^A.7,^A7..7,^A1B2C3
```

2.1.2 添え字の名前付け規約と制限

グローバルの添え字には複数の名前があり、ノード・レベルを識別します。添え字には以下の名前付け規約があります。

- ・ 添え字名は最大 256 文字までです。
- ・ 添え字名は、任意の数値あるいは引用符に囲まれた任意の文字列ですが、NULL 文字列(“)は使用できません。空白、出力不能文字、Unicode 文字など全種類の文字を使用できます。添え字名は、大文字と小文字を区別します。
- ・ 標準的な数値評価は、算術演算、連結演算など数値の添え字名で先頭と末尾のゼロを削除して実行されます。文字列連結演算は、文字列の添え字名で実行されます。
- ・ 添え字名は、ローカル変数あるいはグローバル変数として指定できますが、変数の定義が必要です。

したがって、以下は有効な添え字名です (SET コマンドと WRITE コマンドの組み合わせにより、対応する添え字名を示しています)。

```
SET ^a(1)="The " ; leading and trailing zeros stripped  
WRITE ^a(001.00)  
SET ^a("2")="quick " ; numeric and its string equivalent  
WRITE ^a(2)  
SET ^a(1+2)="brown " ; arithmetic operations performed  
WRITE ^a(3)  
SET ^a(1_2)="fox " ; concatenation performed  
WRITE ^a(12)  
SET ^a(0)="jumped " ; zero valid, extra zeros stripped  
WRITE ^a(0000)  
SET ^b="word",^a(^b)="over " ; name as defined global  
WRITE ^a("word")  
SET ^a("short_"_"word")="the " ; string concatenation  
WRITE ^a("short word")  
SET ^a("!@#$$%^&*")="lazy " ; punctuation characters  
WRITE ^a("!@#_"_"$%^&*")  
SET ^a(" ")="dog's " ; blank space valid  
WRITE ^a(" ")  
SET b=$CHAR(960),^a(b)="back." ; Unicode chars valid  
WRITE ^a(b)
```

多くの添え字レベルを指定できます。添え字レベルの範囲は、グローバル参照の文字総数に基づきます。グローバル参照の合計の長さは、グローバル名と添え字すべての合計で、最長 1,023 文字です (通常は 1,023 文字以下を入力)。したがって、複数の添え字レベルを使用する場合、長い添え字名やグローバル名を避ける方が無難です。

2.1.3 グローバル・データ

グローバル内のデータは、添え字名で識別される 1 つ以上の ノード に格納されます。各ノードのテキストは約 32K の文字が含まれます (正確な最大サイズは 32K マイナス 1 文字もしくは 32,767 文字です)。アプリケーションで、ノードは一般的に以下の構造タイプを含みます。

1. 文字列、または数値データ。Caché の UNICODE バージョンでは、文字列データに UNICODE 本来の文字を含むことができます。
2. 特殊文字で区切られた複数のフィールドを持つ文字列。

```
^Data(10) = "Smith^John^Boston"
```

Caché ObjectScript の \$PIECE 関数を使用して、そのようなデータを切り離すことができます。

3. Caché の \$LIST 構造に含まれた複数のフィールド。\$LIST 構造は、複数の長さエンコード値を含む文字列です。特殊識別文字は必要ありません。
4. NULL 文字列 (""). グローバル添え字名自身がデータとして使用される場合、データは実際のノードに格納されません。
5. ビット文字列。ビットマップ・インデックスの一部を格納するために、グローバルを使用する場合、ノードに格納された値はビット文字列です。ビット文字列は、論理的で、圧縮された一連の 1 と 0 の値を含む文字列です。\$Bit 関数を使用して、ビット文字列を構築できます。
6. 大規模な一連のデータの一部。例えば、オブジェクト、および SQL エンジン、ストリーム (BLOB) を一連の 32 K シーケンシャル・ノードとしてグローバルに格納します。ストリーム・インタフェースにより、ストリームのユーザは、ストリームをこのように格納することを認識していません。

2.1.4 グローバル添え字

グローバルの各ノードは添え字なし、もしくは複数の添え字で指定します。グローバル・ノードは、任意の数の添え字で指定されます (総グローバル参照数の範囲による)。ノードは添え字なしでも構いません (その場合、グローバル名に続く括弧が省略されます)。

添え字は連続している必要はありません。

以下の文はいずれも有効なグローバル参照を含みます。

```
Print ^Data
Print ^Data(1)
Print ^Data(1,2,3)
Print ^Data("Customer", "453-543", 56, "Balance")
```

2.1.5 照合

グローバル内で、ノードは照合（ソート）順序で格納します。

アプリケーションは通常、添え字として使用する値に変換を適用して、ノードを格納する順序を制御します。例えば SQL エンジンが文字列値にインデックスを生成するとき、すべての文字列値を大文字に変換し、スペースを先頭に付けます。これにより、確実にインデックスが大文字と小文字を区別せず、(数値を文字列として格納しても) テキストとして照合するようにします。

2.2 グローバルの物理構造

グローバルは、最適化された構造を使用して、物理ファイルに格納されます。このデータ構造を管理するコードも、Cache が起動するプラットフォームごとに最適化されます。この最適化により、グローバルでのオペレーションは高いスループット（単位時間あたりのオペレーション処理能力）、高水準の並行処理（同時アクセスの総数）、そして Cache メモリの有効利用が可能となり、性能に対するメンテナンス（頻繁に行われてきた再構築や、再度のインデックス付けや圧縮作業）も一切必要ありません。

グローバルの格納に使用する物理構造は完全にカプセル化されるため、アプリケーションはどのような状況でも、物理データ構造に注意する必要はありません。

グローバルは、ディスク上の一連のデータ・ブロックに格納されます。各ブロックのサイズ（通常 8 KB）は、物理データベースを生成するときに決まります。データに効率的なアクセスを提供するため、Cache は、ポインタ・ブロックを使用して関連したデータにリンクする、高性能な B ツリーに類似した構造を保持します。Cache はバッファ・プール（頻繁に参照されるブロックのメモリ内 Cache）を保持し、ディスクからブロックをフェッチするためのコストを削減します。

データ・ストレージに関して、多くのデータベース技術が B ツリーに類似した構造を使用している中、Cache には多くの面で他とは異なるユニークな商品です。

- ・ ストレージのメカニズムは、安全で利用しやすいインタフェースを通して公開されます。
- ・ 添え字とデータは、ディスク・スペースと貴重なメモリ内キャッシュ・スペースを節約するために圧縮されます。
- ・ ストレージ・エンジンは、トランザクション処理操作を最適化し、挿入、更新、削除を高速にします。リレーショナル・システムとは異なり、パフォーマンスのリストアのときのインデックスやデータの再構築は一切必要ありません。
- ・ ストレージ・エンジンは、最適化により、できる限り多くの同時アクセスを可能します。
- ・ 効率よく変換するために、データを自動的にクラスタ化します。

2.2.1 グローバルの格納法

データ・ブロック内で、グローバルは順番に格納されます。添え字とデータの両方は、一緒に格納します。別のブロックに格納する大きなノード値 (長い文字列) には、特例もあります。この別のブロックのポインタは、ノード添え字とともに格納します。

例えば、以下の要領でグローバルが存在するとします。

```
^Data(1999) = 100
^Data(1999,1) = "January"
^Data(1999,2) = "February"
^Data(2000) = 300
^Data(2000,1) = "January"
^Data(2000,2) = "February"
```

おそらくこのデータは、以下の例と類似した連続構造で、単独のデータ・ブロックに格納されます (実際の表現は一連のバイトです)。

```
Data(1999):100|1:January|2:February|2000:300|1:January|2:February|...
```

^Data 処理は最小限のディスク処理で、すべてのコンテンツの検索を行うことができます。

その他にも、挿入、更新、削除の有効利用に使用する方法があります。

2.3 グローバルの参照

グローバルは、特定の Caché データベースに格納します。適切にマッピングされている場合、グローバルの一部を別のデータベースに置くことも可能です。データベースは、現在のシステムあるいは Caché のネットワーク経由でアクセスされるリモート・システムに物理的に格納できます。データセットは、Caché データベースを含むシステムとディレクトリを指します。ネットワークの詳細は、“分散データ管理ガイド” を参照してください。

ネームスペースは、データセットとグローバル・マッピングの論理定義で、共に対応する情報を構成します。

単純グローバル参照は、現在選択されているネームスペースに適用します。ネームスペースを定義することで、ローカル・システムあるいはリモート・システム上のデータベースに物理的にアクセスできるようになります。各グローバルは、それぞれ異なる位置やデータセットにマップされます (データセットは、Caché データベースを含むシステムとディレクトリを指します)。

例えば以下の構文を使用して、現在マップされているネームスペース内のグローバル ORDER へ単純参照を生成します。

```
^ORDER
```

2.3.1 グローバル・マッピングの設定

同じシステムあるいは異なるシステム上に存在するデータベースから別のデータベースにグローバルとルーチンをマップできます。これによって、さまざまな場所に存在するデータを、簡単に参照することができるようになります。グローバル全体あるいはその一部をマップできます。グローバル添え字をマップする機能により、データのディスク間の移動を簡単に行うことができます。

このタイプのマッピングを行うには、“Cache システム管理ガイド”の“Cache の構成”の章の“ネームスペースへのグローバル、ルーチン、およびパッケージ・マッピングの追加”のセクションを参照してください。

一旦グローバルをあるネームスペースから別のネームスペースへマップすると、現在のネームスペースに存在するかのように、マップされたグローバルを参照（単純参照）できます。例えば以下のようになります。

```
^ORDER
```

2.3.2 拡張グローバル参照

現在のネームスペース以外のネームスペースに格納されているグローバルの参照も可能です。これを **拡張グローバル参照**、または省略して **拡張参照** と呼びます。

拡張参照には以下の 2 つの形式があります。

- ・ 明示的なネームスペース参照 – グローバルが、グローバル参照構文の一部を格納しているネームスペース名を指定します。
- ・ 暗黙のネームスペース参照 – ディレクトリを指定します。オプションで、グローバル参照構文の一部としてシステム名を指定できます。この場合、物理データセット（ディレクトリとシステム）がグローバル参照の一部として割り当てられるため、グローバル・マッピングは適応されません。

明示的なネームスペースを優先的に使用します。これにより、必要に応じて、アプリケーション・コードを変更せずに外部的に論理マッピングを再定義できます。

Cache は以下の 2 つの拡張参照形式をサポートします。

- ・ ブラケット構文 – 各括弧 ([]) で拡張参照を囲みます。
- ・ 環境構文 – 垂直バー (|) で拡張参照を囲みます。

2.3.2.1 ブラケット構文

ブラケット構文を使用して、明示的あるいは暗黙のネームスペースで拡張グローバル参照を指定できます。

明示的なネームスペース

```
^[nspace]glob
```

暗黙ネームスペース

```
^[dir,sys]glob
```

上記の明示的なネームスペース参照で、nspace は定義済みのネームスペースで、グローバル glob は現在マップあるいは複製されていません。また、上記の暗黙のネームスペース参照で、dir はディレクトリ、sys はシステム、glob はディレクトリ内のグローバルです。

ディレクトリとシステム名またはネームスペース名は、変数として指定しない限り引用符で囲みます。ディレクトリとシステムは共に暗黙のネームスペースを構成します。また、暗黙のネームスペースは以下のいずれかを参照できます。

- ・ 指定されたシステムの指定ディレクトリ
- ・ 参照でシステム名を指定されていない場合、ローカル・システムで指定されたディレクトリ。システム名を暗黙のネームスペース参照から削除する場合、ディレクトリ参照内に二重キャレット文字 (^) を置き、削除したシステム名であることを示す必要があります。

以下はリモート・システムに暗黙のネームスペースを指定します。

```
["dir", "sys"]
```

以下はローカル・システムに暗黙のネームスペースを指定します。

```
["^^dir"]
```

例えば、以下は SALES と呼ばれるマシンの BUSINESS ディレクトリ内でグローバル ORDER にアクセスします。

```
SET x = ^["BUSINESS", "SALES"]ORDER
```

以下は、ローカル・マシンの BUSINESS ディレクトリ内でグローバル ORDER にアクセスします。

```
SET x = ^["^^BUSINESS"]ORDER
```

MARKETING として定義済みのネームスペースでグローバル ORDER にアクセスします。

```
SET x = ^["MARKETING"]ORDER
```

2.3.2.2 環境構文

環境構文は以下のように定義されます。

```
^|"env" |global
```

“env” は 4 つの形式のうちの 1 つです。

- ・ NULL 文字列 (“”) – ローカル・システムの現在のネームスペース
- ・ "namespace" – グローバル が現在マップされていない定義済みのネームスペース。ネームスペース名は、大文字と小文字を区別しません。
- ・ "^^dir" – 暗黙のネームスペース。既定のディレクトリは、ローカル・システムの指定ディレクトリです。
- ・ "^system^dir" – 暗黙のネームスペース。既定のディレクトリは、指定されたリモート・システムの指定ディレクトリです。

ORDER にマッピングが定義されていない場合、以下の構文を使用し、現システムのネームスペースでグローバル ORDER にアクセスします。

```
SET x = ^|" "|ORDER
```

これは、単純グローバル参照と同じです。

```
SET x = ^ORDER
```

MARKETING として定義済みのネームスペースにマップされたグローバル ORDER にアクセスします。

```
SET x = ^|"MARKETING" |ORDER
```

暗黙のネームスペースを使用して、ローカル・システムの BUSINESS ディレクトリでグローバル ORDER にアクセスします。

```
SET x = ^|"^^BUSINESS" |ORDER
```

暗黙のネームスペースを使用して、SALES というリモート・システムの BUSINESS ディレクトリでグローバル ORDER にアクセスします。

```
SET x = ^|"^SALES^BUSINESS" |ORDER
```

3

多次元ストレージの使用方法（グローバル）

この章では、多次元ストレージ（グローバル変数）を使用して実行できるさまざまな処理について説明します。

3.1 グローバルへのデータ格納

グローバル・ノードへのデータの格納は単純で、グローバルを他の変数のように処理します。違いは、グローバルでの処理が自動的にデータベースに書き込まれる点です。

3.1.1 グローバルの生成

新しいグローバルを生成するための設定作業は一切ありません。グローバルにデータを設定するだけで、自動的に新しいグローバル構造が生成されます。グローバル（またはグローバル添え字）を生成し、シングル演算でデータを置くことができます。あるいは、グローバル（または添え字）を生成し、NULL 文字列を設定して空にします。Cache ObjectScript では、これらの処理は SET コマンドを使用して実行します。

以下の例は、色 という名のグローバルを定義し（存在しない場合）、値 “赤” と関連付けます。色 という名前のグローバルが既に存在する場合、新規の情報を組み込むために以下を実行します。

Cache Basic では、以下の通りです。

```
^Color = "Red"
```

Cache ObjectScript では、以下の通りです。

```
SET ^Color = "Red"
```

注釈: アプリケーション内でダイレクト・グローバル・アクセスを使用するとき、名前付け規約を作成し、アプリケーションの異なる部分同士でお互いの名前が“衝突しない”ように規約に従う必要があります。これは、クラス、メソッド、他の変数に名前付け規約を作成するときと同様です。

3.1.2 グローバル・ノードに格納するデータ

グローバル添え字ノードに値を格納するには、他の変数配列と同様に、グローバル・ノードの値を設定します。指定したノードが存在していない場合は、そのノードを生成します。すでに存在している場合、既存の値を新しい値に置き換えます。

式を使用して、グローバルにノードを指定します (グローバル参照)。グローバル参照は、キャレット文字 (^)、グローバル名、(必要に応じて) 1 つ以上の添え字値で構成されます。添え字 (が存在する場合は括弧 “()” で囲み、コンマで区切ります。各添え字値は、リテラル値、変数、論理式、グローバル参照などの式になります。

グローバル・ノードの値の設定は、アトミック処理です。これは、正常に行われることが保証されており、整合性を確保するためにロックを使用する必要はありません。

以下は、すべて有効なグローバル参照です。

Caché Basic では、以下の通りです。

```
^Data = 2
^Data("Color") = "Red"
^Data(1,1) = 100
^Data(^Data) = 10      ' The value of ^Data is the subscript
^Data(a,b) = 50      ' The values of local variables a and b are subscripts
^Data(a + 10) = 50
```

Caché ObjectScript では、以下の通りです。

```
SET ^Data = 2
SET ^Data("Color")="Red"
SET ^Data(1,1)=100      /* The 2nd level subscript (1,1) is set
                        to the value 100. The 1st level subscript
                        (1) is undefined. */
SET ^Data(^Data)=10    /* The value of global variable ^Data
                        is the name of the subscript */
SET ^Data(a,b)=50      /* The values of local variables a and b
                        are the names of the subscripts */
SET ^Data(a+10)=50
```

Caché ObjectScript を使用している場合、[間接演算](#) を使用して、実行時にグローバル参照を構築できます。

3.1.3 グローバル・ノードへの構造化されたデータ格納

各グローバル・ノードは最長 32K 文字の単独の文字列を含むことが可能です。

データは通常、以下のいずれかの方法でノード内に格納されます。

- ・ 最長 32K 文字の 1 文字列として (正確には 32K マイナス 1 文字)
- ・ 複数のデータ部分を含む、文字で区切られた文字列として

区切り文字を使用してノード内にフィールド一式を格納するには、結合演算子 () を使用して値を連結させます。以下の Caché ObjectScript の例は、区切り文字として # 記号を使用しています。

```
SET ^Data(id)=field(1)_"#"_field(2)_"#"_field(3)
```

データ取得の際、\$PIECE 関数を使用してフィールドを引き離すことができます。

```
SET data = $GET(^Data(id))
FOR i=1:1:3 {
    SET field(i) = $PIECE(data,"#",i)
}
QUIT
```

- ・ 複数のデータを含む \$LIST でコード化された文字列として。

\$LIST 関数は、区切り文字を指定する必要がない特殊な長さエンコード法を使用します (Caché オブジェクトと SQL で使用される既定構造です)。

ノード内にフィールド一式を格納するには、\$LISTBUILD 関数を使用してリストを構築します。

```
SET ^Data(id)=$LISTBUILD(field(1),field(2),field(3))
```

データ取得の際、\$LIST 関数または \$LISTGET 関数を使用してフィールドを切り離すことができます。

```
SET data = $GET(^Data(id))
FOR i = 1:1:3 {
    SET field(i)=$LIST(data,i)
}
QUIT
```

- ・ 大きなデータの一部 (ストリームや “BLOB” など) として。

個々のノードが 32K までのデータに限られているため、例えばストリームなど大規模な構造は、連続したノード内にデータを格納することで実装されます。

```
SET ^Data("Stream1",1) = "First part of stream...."
SET ^Data("Stream1",2) = "Second part of stream...."
SET ^Data("Stream1",3) = "Third part of stream...."
```

ストリームをフェッチするコード (%GlobalCharacterStream クラスで提供されるコードなど) は、連続した文字列としてデータを提供する構造で、継続的にノードをループします。

- ・ 文字列として

ビットマップ・インデックス (ビット文字列のビットが表の行に対応するインデックス) を実装する場合は、インデックス・グローバルのノード値をビット文字列に設定します。Caché は、エンコーディング・ビット文字列に圧縮アルゴリズムを使用します。したがって、ビット文字列は、Caché \$BIT

関数を使用してのみ操作できます。ビット文字列に関する詳細は、“ビット文字列関数の概要”を参照してください。

- ・ 空ノードとして

必要なデータがノード添え字名内にある場合、一般的に実ノード値を NULL 文字列 (“”) に設定します。例えば、ID 値を持つ名前に対応するインデックスであれば、通常は以下の通りです。

```
SET ^Data("APPLE",1) = ""
SET ^Data("ORANGE",2) = ""
SET ^Data("BANANA",3) = ""
```

3.2 グローバル・ノードの削除

Caché ObjectScript の KILL コマンド、ZKILL コマンド、Caché Basic の Erase コマンドを使用して、グローバル・ノード、サブノード・グループ、またはグローバル全体をデータベースから削除することができます。

KILL コマンドは、下位ノードを含む特定のグローバル参照で、すべてのノード (データおよび配列内にあるそのエントリ) を削除します。つまり、指定した添え字で開始するすべてのノードが削除されます。

例えば、以下の Caché ObjectScript 文を試みましょう。

```
KILL ^Data
```

これは ^Data グローバル全体を削除します。このグローバルを引き続き参照すると <UNDEFINED> エラーを返します。

以下にもう 1 つ Caché ObjectScript 文があります。

```
KILL ^Data(100)
```

これは、^Data グローバルのノード 100 のコンテンツを削除します。^Data(100,1)、^Data(100,2)、^Data(100,1,2,3) などの下位ノードがある場合、同様に削除されます。

Caché ObjectScript ZKILL コマンドは、指定されたグローバルやグローバルの添え字ノードを削除します。下位ノードは削除しません。

グローバル変数には New コマンドを使用できません。

3.3 グローバル・ノードの存在有無のテスト

Caché ObjectScript の \$DATA 関数を使用して、特定のグローバル (またはその下位ノード) がデータを含むかどうかをテストできます。

\$DATA は、指定したグローバル参照が存在するか否かを示した値を返します。以下がその値の例です。

状態値	意味
0	グローバル変数が未定義
1	グローバル変数が存在し、データを含みますが、下位ノードはありません。NULL 文字列 ("") もデータと見なされます。
10	グローバル変数に下位ノードがありますが (下位ノードへの下方ポインタを含みます)、そのノード自身はデータを含みません。このような変数への直接参照は、<UNDEFINED> エラーになります。例えば、\$DATA(^y) が 10 を返す場合、SET x=^y は <UNDEFINED> エラーを生成します。
11	データと下位ノードを含むグローバル変数です (下位ノードへの下方ポインタも含みます)。

3.4 グローバル・ノード値の検索

グローバル参照を式として使用することで、特定のグローバル・ノード内に格納された値を取得できます。

Caché Basic の使用法

```
color = ^Data("Color")      ' assign to a local variable
Print ^Data("Color")       ' use as an argument to a command
MyMethod(^Data("Color"))   ' use as a function argument
```

Caché ObjectScript の使用法

```
SET color = ^Data("Color")   ; assign to a local variable
WRITE ^Data("Color")        ; use as a command argument
SET x=$LENGTH(^Data("Color")) ; use as a function parameter
```

3.4.1 \$GET 関数

Caché ObjectScript では、\$GET 関数を使用して、グローバル・ノードの値も取得できます。

```
SET mydata = $GET(^Data("Color"))
```

これは、(存在する場合は) 指定されたノード値を取得します。ノードが未定義の場合は NULL 文字列を返します。オプションで 2 番目の \$GET 引数を使用し、ノードが未定義の場合に指定された既定値を返すこともできます。

3.4.2 WRITE コマンド、ZWRITE コマンド、ZZDUMP コマンド

さまざまな Caché ObjectScript 表示コマンドを使用して、グローバルやグローバル・サブノードのコンテンツを表示できます。WRITE コマンドは、指定されたグローバルの値や文字列としてサブノードを返します。ZWRITE コマンドは、グローバル変数名とその値、また、下位ノードとその値を返します。ZZDUMP コマンドは、指定されたグローバルの値や 16 進数ダンプ形式のサブノードを返します。

3.5 グローバル内のデータ検索

グローバル内に格納されているデータの検索 (反復) には多くの方法があります。

3.5.1 \$ORDER (Next / Previous) 関数

Caché ObjectScript の \$ORDER 関数 (および Caché Basic での Traverse) では、グローバル内の各ノードに順にアクセスできます。

\$ORDER 関数は、与えられたレベル (添え字番号) で次の添え字値を返します。例えば、以下のグローバルを定義するとします。

```
Set ^Data(1) = ""
Set ^Data(1,1) = ""
Set ^Data(1,2) = ""
Set ^Data(2) = ""
Set ^Data(2,1) = ""
Set ^Data(2,2) = ""
Set ^Data(5,1,2) = ""
```

最初のファースト・レベル添え字を検索するには、以下を使用します。

```
SET key = $ORDER(^Data(""))
```

これは、NULL 文字列 (") に続く最初のファースト・レベル添え字を返します (NULL 文字列は、最初のエントリの前の添え字値を示します。返り値として使用する場合、後ろに添え字値がないことを示します)。この例では、key はここで値 1 を含みます。

1、または \$ORDER 式の key を使用して、2 番目のファースト・レベル添え字を検索できます。

```
SET key = $ORDER(^Data(key))
```

key に初期値 1 がある場合、この文が 2 に設定します (^Data(2) が次のファースト・レベル添え字であるため)。この文を再度実行すると、key は次のファースト・レベル添え字の 5 に設定されます。^Data(5) に直接格納されているデータはありませんが、5 を返すという点に注意してください。この文を再実行しても、これ以上ファースト・レベル添え字が存在しないため、key を NULL 文字列(“)に設定します。

追加の添え字を \$ORDER 関数で使用することで、異なる添え字レベルを繰り返すことができます。\$ORDER は、引数リストにある最終添え字の次の値を返します。以下の文は、上記のデータを使用した例です。

```
SET key = $ORDER(^Data(1,“”))
```

これは、^Data(1,1) が次のセカンド・レベルの添え字であるため、key を 1 に設定します。この文を再度実行すると、key は次のセカンド・レベル添え字の 2 に設定されます。この文をもう一度実行しても、ノード ^Data(1) にはこれ以上セカンド・レベル添え字が存在しないため、key は “” に設定されます。

3.5.1.1 \$ORDER によるループ

以下の Caché ObjectScript コードは単純グローバルを定義し、そのファースト・レベル添え字をすべて繰り返します。

```
// clear ^Data in case it has data
Kill ^Data

// fill in ^Data with sample data
For i = 1:1:100 {
    // Set each node to a random person's name
    Set ^Data(i) = ##class(%PopulateUtils).Name()
}

// loop over every node
// Find first node
Set key = $Order(^Data(“”))

While (key != “”) {
    // Write out contents
    Write “#”, key, “ ”, ^Data(key), !

    // Find next node
    Set key = $Order(^Data(key))
}
```

3.5.1.2 追加の \$ORDER 引数

Caché ObjectScript の \$ORDER 関数は、オプションとして 2 番目の引数や 3 番目の引数を持ちます。2 番目の引数は方向フラグで、グローバルを検索する方向を示します。既定値の 1 は、前方検索を指定し、-1 は後方検索を指定します。

3 番目の引数が存在する場合、ローカル変数名を含みます。\$ORDER で見つかったノードがデータを含む場合、このデータはローカル変数で記述されます。グローバルを繰り返し、添え字値とノード値が必要な場合は、更に効率的に処理します。

3.5.2 グローバルの反復

与えられたグローバルがまとまって、連続した数値の添え字を使用することがわかっている場合、単純な For ループを使用して、その値を繰り返します。例えば Caché Basic では以下の通りです。

```
For i = 1 To 100
  Print ^Data(i)
Next
```

また、Caché ObjectScript では以下のようになります。

```
For i = 1:1:100 {
  Write ^Data(i),!
}
```

通常は、上記で説明した \$ORDER 関数を使用することをお勧めします。その方が効率も良く、データ間の欠落部分を心配する必要もありません (削除されたノードなど)。

3.5.3 \$QUERY 関数

サブノード間を移動して、グローバル内の各ノードとサブノードにアクセスする必要がある場合は、Caché ObjectScript の \$QUERY 関数を使用できます (または、入れ子にした \$ORDER ループの使用も可能です)。

\$QUERY 関数はグローバル参照を取り、グローバル (または、後にノードが続かない場合は "") にある次のノードのグローバル参照を含む文字列を返します。\$QUERY に返された値を使用するには、Caché ObjectScript の [間接](#) 演算子を使用する必要があります。

例えば、以下のグローバルを定義するとします。

```
Set ^Data(1) = ""
Set ^Data(1,1) = ""
Set ^Data(1,2) = ""
Set ^Data(2) = ""
Set ^Data(2,1) = ""
Set ^Data(2,2) = ""
Set ^Data(5,1,2) = ""
```

以下は \$QUERY の呼び出しです。

```
SET node = $QUERY(^Data(""))
```

これは、node を文字列 “^Data(1)” に設定します。“^Data(1)” は、グローバルの最初のノードのアドレスです。\$QUERY を再度呼び出し node で間接演算子を使用することで、グローバルの次のノードを取得できます。

```
SET node = $QUERY(@node)
```

この時点で、node は文字列 “^Data(1,1)” を含みます。

以下の例では、グローバル・ノードを設定した後、\$QUERY を使用して検索し、各ノードのアドレスを記述します。

```

Kill ^Data // make sure ^Data is empty

// place some data into ^Data
Set ^Data(1) = ""
Set ^Data(1,1) = ""
Set ^Data(1,2) = ""
Set ^Data(2) = ""
Set ^Data(2,1) = ""
Set ^Data(2,2) = ""
Set ^Data(5,1,2) = ""

// now walk over ^Data
// find first node
Set node = $Query(^Data(""))
While (node != "") {
    Write node,!
    // get next node
    Set node = $Query(@node)
}

```

3.6 グローバル内でのデータのコピー

Caché ObjectScript の MERGE コマンドを使用して、グローバルのコンテンツ（全体または一部）を別のグローバル（またはローカル配列）にコピーできます。

以下の例は、OldData グローバルのコンテンツ全体を NewData グローバルにコピーする MERGE コマンドの使用法を示しています。

```
Merge ^NewData = ^OldData
```

MERGE コマンドのソース引数に添え字がある場合、そのノード内のすべてのデータと派生ノードがコピーされます。方向引数に添え字がある場合、方向アドレスをトップ・レベル・ノードとして使用し、データをコピーします。以下はコードの例です。

```
Merge ^NewData(1,2) = ^OldData(5,6,7)
```

これは、^OldData(5,6,7) と、その下にあるすべてのデータを ^NewData(1,2) にコピーします。

3.7 グローバルでの共有カウンタ保持

大規模なトランザクション処理作業の主な並行処理障害により、一意の識別子の値が生成されることがあります。例えば新規送り状に、それぞれ一意の識別子番号を付ける注文処理作業を考えてみましょう。従来の方法としては、カウンタ・テーブルのようなものを保持します。新規送り状作成の

各過程では、カウンタのロックを取得し、その値をインクリメントし、ロックの解放を行います。その結果、この単独のレコード上で、リソースが頻繁に競合することになります。

この問題を処理するために、Cache ObjectScript の \$INCREMENT 関数があります。\$INCREMENT は、自動的にグローバル・ノードの値をインクリメントします (ノードが未定義の場合は 1 に設定されます)。\$INCREMENT の基本的な性質としてロックは不要です。他のプロセスからの干渉なしで、インクリメントされた値を返す機能が保証されているからです。

以下のようにして \$INCREMENT を使用できます。まず、カウンタを保持するグローバル・ノードを決定します。次に、新規のカウンタ値が必要になるたびに \$INCREMENT を実行します。

```
SET counter = $INCREMENT(^MyCounter)
```

Cache Objects と SQL で使用される既定のストレージ構造は、\$INCREMENT を使用して固有のオブジェクト (行) 識別子の値を割り当てます。

3.8 一時グローバル

特定の処理に対して、永続性を要求せずに、グローバルの強力な性能が必要になる場合もあります。例えば、グローバルを使用して、ディスクに保存する必要のないデータをソートするとします。このような処理には、Cache は一時グローバルを提供します。

一時グローバルには、以下のような特性があります。

- ・ 一時グローバルは、常にローカルとして定義される (つまり非ネットワーク) データベースである CACHETEMP データベース内に格納されます。CACHETEMP データベースにマップされたグローバルは、すべて一時グローバルとして扱います。
- ・ 一時グローバルに対する変更は、ディスクに記述されません。その代わりに、そのような変更はメモリ内のバッファ・プールに保持されます。大規模な一時グローバルは、バッファ・プールに十分なスペースがない場合はディスクに記述されます。
- ・ 最大限に効率を良くするため、一時グローバルへの変更はジャーナル・ファイルにはログされません。
- ・ Cache システムを再起動すると、一時グローバルは自動的に削除されます。(メモ : システムの再起動には長時間を要する場合がありますので、一時グローバルを削除する目的でこの方法を使用するのは避けてください。)

既定では、名前が “CacheTemp” で始まるグローバルはすべて、一時グローバルとして定義されます。Cache 自体が使用する一時グローバルとの衝突を避けるため、使用する一時グローバル名は “CacheTempUser” で始めると良いでしょう。

Caché SQL は、一時グローバルを複雑なクエリの最適化用スラッチ・スペースとして使用します。また、特定のクエリ（ソート、グループ分け、集約の計算用など）の実行中には、臨時インデックスとしても使用されます。

3.9 グローバルでのデータのソート

グローバルに格納されたデータは、添え字値に従って自動的にソートされます。例えば、以下の Caché ObjectScript コードは、グローバル一式を（順不同で）定義し、繰り返すことにより、グローバル・ノードが添え字により自動的にソートされることを示します。

```
// Erase any existing data
Kill ^Data

// Define a set of global nodes
Set ^Data("Cambridge") = ""
Set ^Data("New York") = ""
Set ^Data("Boston") = ""
Set ^Data("London") = ""
Set ^Data("Athens") = ""

// Now iterate and display (in order)
Set key = $Order(^Data(""))
While (key '= "") {
    Write key,!
    Set key = $Order(^Data(key)) // next subscript
}
```

アプリケーションはグローバルが提供する自動ソートを活用し、ソート処理を実行するか、オーダーされた相互参照付インデックスを特定値に保持します。Caché SQL とオブジェクトは、グローバルを使用してそのようなタスクを自動的に実行します。

3.9.1 グローバル・ノードの照合

グローバルのノードがソートされる順序（照合という）は、グローバル自体とそのグローバルを使用しているアプリケーションの二段階で制御されています。

アプリケーション・レベルでは、添え字として使用される値のデータ変換を行うことで、グローバル・ノードの照合法を制御できます（Caché SQL とオブジェクトはユーザ指定の照合機能で行います）。例えば、大文字小文字は関係なくアルファベット順でソートされた名前リストを生成したい場合、一般的にその名前の大文字版を添え字として使用します。

```
// Erase any existing data
Kill ^Data

// Define a set of global nodes for sorting
For name = "Cobra","jackal","zebra","AARDVark" {
  // use UPPERCASE name as subscript
  Set ^Data($ZCONVERT(name,"U")) = name
}

// Now iterate and display (in order)
Set key = $Order(^Data(""))
While (key != "") {
  Write ^Data(key),! // write untransformed name
  Set key = $Order(^Data(key)) // next subscript
}
```

この例は、添え字が大文字小文字を区別せずにソートされるように、名前をそれぞれ大文字に変換します (\$ZCONVERT 関数を使用します)。各ノードは、オリジナルの値が表示されるように、未変換の値を含みます。

3.9.2 数値添え字と文字列値添え字

数字の値は文字列の値の 前に 照合されます。つまり、1 の値が “a” の値よりも先にきます。与えられた添え字に対して数値と文字列値の両方を使用する場合は、この事実は認識しておく必要があります。(値を基にデータをソートするため) インデックスにグローバルを使用する場合、通常、値は数字 (例えば給与) としてソートするか、文字列 (例えば郵便番号) としてソートします。

数値的に照合されたノードに対する一般的な解決法としては、単項演算子 + を使用して、添え字値を強制的に数値にします。例えば、id 値を age でソートするインデックスを構築する場合、age が常に数値になるように、以下の通り強制できます。

```
Set ^Data(+age,id) = ""
```

値を文字列としてソートしたい場合は (例えば “0022”、“0342”、“1584”)、スペース文字 (“ ”) を付けることで、添え字値が常に文字列となるように強制できます。例えば、id 値を zipcode (郵便番号) でソートするインデックスを構築する場合、zipcode が常に文字列になるように、以下の通り強制できます。

```
Set ^Data(" "_zipcode,id) = ""
```

これにより、“0022” など、先頭に 0 が付く値は常に文字列として扱われます。

3.9.3 \$SORTBEGIN 関数と \$SORTEND 関数

通常、Cache 内データのソートに関しては心配する必要はありません。SQL を使用するか直接グローバル・アクセスを使用するかによって、自動的にソート処理されます。

しかし、場合によっては、さらに効率的なソート処理が可能な場合もあります。特に、(1) 順不同 (つまりソートされていない状態) で多数のグローバル・ノードを設定する必要があり、(2) 結果グローバルの合計サイズが Cache バッファ・プールの大部分を占める場合、(データがキャッシュに適合しな

いので) SET 処理の多くはディスク処理に関わるため、パフォーマンスが悪影響を受けることがあります。通常は、一時グローバルの大容量データのロード、インデックスの集合、インデックスなしの値のソートなど、インデックス・グローバルの生成に関わる場合、上記のような状況になります。

これらの状況に効率的に対処するため、Cache ObjectScript は \$SORTBEGIN と \$SORTEND 関数を使用します。\$SORTBEGIN 関数はグローバル (またはその部分) の特別ノードを初期化します。グローバルへのデータ・セットはスクラッチ・バッファに記述され、メモリ (あるいは一時ディスク・ストレージ) にソートされます。\$SORTEND 関数が処理の最後に呼び出されると、データは実際のグローバル・ストレージに順に書き込まれます。ディスク処理を以前ほど要求されないで、書き込みが適切に終了しているため、操作全体がより効率的です。

\$SORTBEGIN 関数の使用法は非常に簡単です。ソートを開始する前にソートしたいグローバル名で起動し、処理が完了した時点で \$SORTEND を呼び出します。

```
// Erase any existing data
Kill ^Data

// Initiate sort mode for ^Data global
Set ret = $SortBegin(^Data)

// Write random data into ^Data
For i = 1:1:10000 {
    Set ^Data($Random(1000000)) = ""
}

Set ret = $SortEnd(^Data)

// ^Data is now set and sorted

// Now iterate and display (in order)
Set key = $Order(^Data(""))
While (key != "") {
    Write key,!
    Set key = $Order(^Data(key)) // next subscript
}
```

\$SORTBEGIN 関数はグローバル作成の特殊なケースに対して設計されており、使用の際には注意が必要です。特に \$SORTBEGIN モードの場合、書き込みをしているグローバルからの読み込みはできません。これはデータが記述されていないと、読み込みが正しく行われなかったためです。

Cache SQL は自動的にこれら関数を使用して、一時インデックス・グローバル (インデックスのついていないフィールドでのソートを行うためなど) を作成します。

3.10 グローバルを使用した間接演算の使用法

間接演算を使用して、Cache ObjectScript は実行時のグローバル参照の作成方法を提供します。これはプログラムの完了時、グローバル構造や名前がわからないアプリケーションにおいて便利です。

間接演算は間接演算子 @ でサポートされ、式を含んだ文字列をデリファレンスします。間接演算には @ 演算子の使用法によって複数のタイプがあります。

以下のコードは、グローバル参照を含む文字列をデリファレンスするときに @ 演算子を使用する、名前間接演算の例を提供します。

```
// Erase any existing data
Kill ^Data

// Set var to an global reference expression
Set var = "^Data(100)"

// Now use indirection to set ^Data(100)
Set @var = "This data was set indirectly."

// Now display the value directly:
Write "Value: ",^Data(100)
```

添え字間接演算を使用して、式 (変数またはリテラル値) を間接演算文内で混合させることもできます。

```
// Erase any existing data
Kill ^Data

// Set var to a subscript value
Set glvn = "^Data"

// Now use indirection to set ^Data(1) to ^Data(10)
For i = 1:1:10 {
    Set @glvn@(i) = "This data was set indirectly."
}

// Now display the values directly:
Set key = $Order(^Data(""))
While (key != "") {
    Write "Value ",key, ": ", ^Data(key),!
    Set key = $Order(^Data(key))
}
```

間接演算は Caché ObjectScript の基本機能で、グローバル参照に制限されません。直接の情報に関する詳細は、“Caché ObjectScript の使用法”の“演算子”の章の“間接演算”を参照してください。間接演算は直接アクセスほど効果的ではないので、その点を考慮して使用してください。

3.11 トランザクション管理

Caché は、グローバルを使用した本格的なトランザクション処理の実装に必要な初期演算を提供します。Caché オブジェクトと SQL は、これらの機能を自動的に利用します。トランザクション用データをグローバルに直接書き込む場合に、これら演算子を使用してください。

トランザクション・コマンドは、トランザクションの開始を定義する TSTART、現在のトランザクションを実行する TCOMMIT、現在のトランザクションを一時停止して、トランザクションの始めからのグローバルへの変更を元に戻す TROLLBACK です。

例えば、以下の Caché ObjectScript コードは、トランザクションの開始を定義し、グローバル・ノード数を設定し、ok 値によってトランザクションを実行、またはロールバックします。

```
TSTART

Set ^Data(1) = "Apple"
Set ^Data(2) = "Berry"

If (ok) {
    TCOMMIT
}
Else {
    TROLLBACK
}
```

TSTART は、トランザクション開始マーカを Caché ジャーナル・ファイルに書き込みます。これは、トランザクション開始の境界線を定義します。変数 ok が上記の例で真 (ゼロ以外) の場合、TCOMMIT コマンドはトランザクションの終わりをマークし、トランザクション終了マーカがジャーナル・ファイルに書き込まれます。ok が偽 (0) の場合、TROLLBACK コマンドはセットすべてを元に戻すか、またはトランザクションの開始時点からの演算を無効にします。この場合、^Data(1) と ^Data(2) は前の値にリストアします。

トランザクションの正常終了の時点で、書き込まれるデータはありません。これは、トランザクション中のデータベースの変更すべてが、正常としてトランザクションの過程で実行されるためです。ロールバックの場合にのみ、データベースのデータに影響します。これは、この例でのトランザクションには限定の分離があることを意味します。つまり、他のプロセスでは修正されたグローバル値がトランザクション終了前に見られます。これは通常、不確定な読み込みであるとみなされます。これが良いか悪いかは、アプリケーションの要件によって異なりますが、多くの場合では問題ありません。アプリケーションに更なる分離が必要な場合は、ロックを使用します。これは、以下のセクションで説明します。

3.11.1 ロックとトランザクション

トランザクションを分離させ、修正されたデータがトランザクションのコミット前に他の処理から見えないようにするには、ロックを使用する必要があります。Caché ObjectScript では、LOCK コマンドを使用して、ロックを直接設定および解除できます。ロックは規約に従って機能します。与えられたデータ構造 (永続オブジェクトに使用されるものなど) に対し、ロックを必要とするすべてのコードは、すべて同じ論理ロック参照 (例えば LOCK コマンドによって同じアドレスが使用される) を使用します。

トランザクション内には、ロックに特別機能があり、トランザクション中に取得されたすべてのロックは、トランザクションが終了するまで解放されません。その理由を、通常のトランザクションで実行されるアクションで考えてみます。

1. TSTART を使用して、トランザクションを開始します。

2. 修正を希望するノード (単数または複数) で、ロック (単数または複数) を取得します。これは通常 “書き込み” ロックと呼ばれます。
3. ノード (1 つ、または複数) を変更します。
4. ロック (1 つ、または複数) を解放します。トランザクション中のため、これらのロックはこの時点では実際には解放されません。
5. TCOMMIT を使用して、トランザクションを終了します。この時点で、前段階で解放されたすべてのロックが実際に解放されます。

他のプロセスがこのトランザクションに関連するノードを見る際に、コミットされていない変更箇所を表示させたくない場合、ノードからデータを読み込む前にロック (“読み込み” ロックと呼ばれます) のテストを行います。書き込みロックはトランザクション終了まで保持されるため、読み取り処理はトランザクションが完了 (終了、またはロールバック) するまでデータを表示しません。

大半のデータベース管理システムが、類似した機能を使用して、トランザクション分離を提供します。Cache は、このメカニズムを開発者が使用できるという点がユニークです。トランザクションをサポートしているときでも、新規アプリケーション・タイプのカスタム・データベース構造の生成を可能にします。当然、Cache オブジェクト、または SQL でデータの管理やトランザクション管理を自動的に行うことができます。

3.11.2 入れ子にされた TSTART の呼び出し

Cache には、特別なシステム変数 \$TLEVEL があります。これは、TSTART コマンドを呼び出した回数をトラッキングします。\$TLEVEL は値 0 から開始し、TSTART を呼び出すたびに \$TLEVEL の値を 1 ずつインクリメントします。また、TCOMMIT を呼び出すたびに値を 1 ずつデクリメントします。TCOMMIT の呼び出しによって \$TLEVEL が 0 に戻る場合、トランザクションは (コミットされて) 終了します。

TROLLBACK コマンドへの呼び出しは常に、現在のトランザクションを終了させ、\$TLEVEL をその値に関係なく元の 0 に設定します。

この動作で、アプリケーションはトランザクションを、それ自身がトランザクションを含むコード (オブジェクト・メソッドなど) を包むことができるようにします。例えば、永続オブジェクトに備わっている %Save メソッドは、常に、トランザクションとしてのオペレーションを実行します。TSTART と TCOMMIT を明示的に呼び出すことで、複数のオブジェクト保存処理を含むさらに大きなトランザクションを生成することができます。

```
TSTART
Set sc = object1.%Save()
If ($$$ISOK(sc)) {
    // first save worked, do the second
    Set sc = object2.%Save()
}

If ($$$ISERR(sc)) {
    // one of the saves failed, rollback
    TROLLBACK
}
Else {
    // everything is ok, commit
    TCOMMIT
}
```

3.12 並行処理の管理

シングル・グローバル・ノードの設定、または検索はアトミックです。必ず、常に一定の結果が得られます。多次元ノードの操作やトランザクション分離制御に対し ([ロックとトランザクション](#) のセクション参照)、Cache はロックを取得して解放する機能を提供します。

ロックは、Cache ロック・マネージャで管理されます。Cache ObjectScript では、LOCK コマンドを使用して、ロックを直接設定および解除できます (Cache オブジェクトと SQL は、ロックの取得と解放を必要に応じて自動的に行います)。

LOCK コマンドの詳細は、リファレンス・ページの “LOCK” コマンドを参照してください。

3.13 最新のグローバル参照

最新式のグローバル参照は、Cache ObjectScript \$ZREFERENCE 特殊変数に記述されています。\$ZREFERENCE には、指定によって添え字と拡張グローバル参照など最新のグローバル参照が組み込まれています。\$ZREFERENCE は、グローバル参照が成功したかどうか、あるいは指定グローバルが存在するかどうかを示すものではありません。Cache は、指定された最新のグローバル参照を単に記録しています。

3.13.1 ネイキッド・グローバル参照

Cache は、添え字付きグローバル参照の後にグローバル名と添え字レベルを示す ネイキッド・インジケータを設定します。その後 ネイキッド・グローバル参照を使用して、同じグローバルと添え字レベルに連続して参照を作成します。グローバル名と上位レベルの添え字は削除します。これにより、同じ (もしくは下位の) 添え字レベルの同じグローバルに対する参照の繰り返しを能率的に行います。

ネイキッド参照に下位の添え字レベルを指定すると、そのレベルに合わせてネイキッド・インジケータをリセットします。したがって、ネイキッド・グローバル参照を使用する場合、常に最新のグローバル参照で構築した添え字レベルで作業することになります。

ネイキッド・インジケータ値は、\$ZREFERENCE 特殊変数に記録されます。この値は NULL 文字列に初期化されます。ネイキッド・インジケータが設定されていないのにネイキッド・グローバル参照を試みると、〈NAKED〉エラーが生じます。ネームスペースを変更すると、ネイキッド・インジケータも再初期化されます。\$ZREFERENCE を NULL 文字列 ("") に設定することで、ネイキッド・インジケータを再初期化できます。

以下の例は、添え字付きのグローバル `^Produce("fruit",1)` が最初の参照に指定されています。Cache は、このグローバル名と添え字をネイキッド・インジケータに保存します。したがって、次のネイキッド・グローバル参照ではグローバル名 "Produce" と上位の添え字レベル "fruit" を省略できます。(3,1) ネイキッド参照が下位の添え字レベルに移動した場合、この新規の添え字レベルが、その後続くネイキッド・グローバル参照の条件となります。

```
SET ^Produce("fruit",1)="Apples" /* Full global reference */
SET ^(2)="Oranges" /* Naked global references */
SET ^(3)="Pears" /* assume subscript level 2 */
SET ^(3,1)="Bartlett pears" /* Go to subscript level 3 */
SET ^(2)="Anjou pears" /* Assume subscript level 3 */
WRITE "latest global reference is: ", $ZREFERENCE, !
ZWRITE ^Produce
KILL ^Produce
```

この例は、グローバル変数 `^Produce("fruit",1)`、`^Produce("fruit",2)`、`^Produce("fruit",3)`、`^Produce("fruit",3,1)`、`^Produce("fruit",3,2)` を設定します。

例外はありますが、ほとんどのグローバル参照 (完全あるいはネイキッド) がネイキッド・インジケータを設定します。\$ZREFERENCE 特殊変数は、ネイキッド・グローバル参照だったとしても最新のグローバル参照の完全なグローバル名と添え字を持ちます。ZWRITE コマンドも、ネイキッド参照を使用して設定したかどうかにかかわらず、各グローバルの完全な名前と添え字を表示します。

ネイキッド・グローバル参照は注意して使用する必要があります。Cache は、ネイキッド・インジケータを常に明確に設定するわけではないからです。以下はその例です。

- ・ 完全グローバル参照は、ネイキッド・インジケータを最初に設定し、グローバル参照が失敗しても、その後の完全グローバル参照あるいはネイキッド・グローバル参照は、ネイキッド・インジケータを変更します。例えば、存在しないグローバルの値を WRITE しようとする、ネイキッド・インジケータを設定します。
- ・ 添え字付きのグローバルを参照するコマンド後置条件は、Cache が後置条件を評価する方法に関係なくネイキッド・インジケータを設定します。
- ・ 添え字付きのグローバルを参照するオプション関数の引数は、Cache がすべての引数を評価するかどうかによって、ネイキッド・インジケータを設定する場合としない場合があります。例えば、含まれる既定値が使用されていない場合でも、\$GET の 2 番目の引数が常にネイキッド・インジケータを設定します。Cache は、左から右の順番で引数を評価します。したがって、最後の引数は、最初の引数で設定されたネイキッド・インジケータをリセットする場合があります。

- ・ トランザクションをロールバックする TROLLBACK コマンドは、ネイキッド・インジケータをトランザクションの最初の値にはロールバックしません。

完全グローバル参照に [拡張グローバル参照](#) を含む場合、その後続くネイキッド・グローバル参照は、同じ拡張グローバル参照と見なされます。つまり、ネイキッド・グローバル参照の一部として拡張参照を指定する必要はありません。

4

多次元ストレージの SQL およびオブジェクトの使用法

この章では、Cache オブジェクトおよび SQL エンジンが、どのように多次元ストレージ (グローバル) を利用して永続オブジェクト、リレーショナル・テーブル、インデックスを格納するかについて説明します。

データ・ストレージ構造の提供と管理は、Cache オブジェクトおよび SQL エンジンが自動的に実行しますが、どのように動作するかを理解しておけば便利でしょう。

データのオブジェクト・ビューとリレーショナル・ビューで使用されるストレージ構造は同じものです。簡潔にするため、ここではオブジェクトの見地からのストレージのみ説明します。

4.1 データ

`%CacheStorage` ストレージ・クラス (既定) を使用する各永続クラスは、それ自体のインスタンスを、多次元ストレージ (グローバル) の 1 つ以上のノードを使用して、Cache データベース内に格納することができます。

各永続クラスには、プロパティのグローバル・ノードへの格納法を指定した、ストレージ定義があります。このストレージ定義 (“既定構造” と呼ばれる) は、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は変更でき、必要に応じて代替バージョンを提供できます。これについてはこのドキュメントでは説明しません)。

4.1.1 既定構造

永続オブジェクトの格納に使用される既定構造は、非常に単純です。

- データは、グローバル名が完全なクラス名 (パッケージ名含む) で始まるグローバル内に格納されます。データ・グローバル名には “D” を、インデックス・グローバルには “I” を付けて、それぞれの名前を作成します。
- 各インスタンスのデータは、\$List 構造内に置かれた、すべての一時的でないプロパティと併せて、データ・グローバルのシングル・ノード内に格納されます。
- データ・グローバル内の各ノードは、オブジェクト ID 値で添え字が付けられています。既定では、オブジェクト ID 値は、データ・グローバルのルート (添え字なし) で格納されているカウンタ・ノードの \$Increment 機能呼び出すことにより提供される整数です。

例えば、2 つのリテラル・プロパティを持つ、単純な永続クラス、**MyApp.Person** を定義するとします。

```
Class MyApp.Person Extends %Persistent [ClassType = 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生成して保存する場合、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(" ",530,"Abraham")
^MyApp.PersonD(2) = $LB(" ",680,"Philip")
```

各ノードに格納されている \$List 構造の最初の部分は空で、クラス名用に確保されます。この **Person** クラスのサブクラスのいずれかを定めると、このスロットはサブクラス名を含みます。(%Persistent クラスが提供する) %OpenId メソッドは、複数のオブジェクトが同じエクステンツ内に保存されている場合、この情報を使用して多様な形態で正しいタイプのオブジェクトを開きます。このスロットはクラス・ストレージ定義に、プロパティ名 “%CLASSNAME” として表示されます。

詳細については、以下の [サブクラス](#) を参照してください。

4.1.2 IDKEY

IDKEY 機能によって、オブジェクト ID として使用する値を明示的に定義できます。これを行うには、IDKEY インデックス定義をクラスに追加し、ID 値を提供するプロパティを指定します。オブジェクトを一旦保存すると、オブジェクト ID 値は変更できません。つまり、IDKEY 機能を使用したオブジェクトを保存した後は、オブジェクト ID が基としているプロパティを変更することはできません。

例えば、IDKEY インデックスを使用するために上記の例で使用した **Person** クラスは変更できます。

```
Class MyApp.Person Extends %Persistent [ClassType = 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生成し保存する場合、結果のグローバルは以下のようになります。

```
^MyApp.PersonD("Abraham") = $LB("",530,"Abraham")
^MyApp.PersonD("Philip") = $LB("",680,"Philip")
```

定義されたカウンタ・ノードは、既に存在していないことに注意してください。また、**Name** プロパティでのオブジェクト ID に基づくと、**Name** の値は各オブジェクトに対して一意である必要があることを意味しています。

IDKEY インデックスが複数のプロパティに基づく場合、メイン・データ・ノードは複数の添え字を持ちます。例えば以下のようになります。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
Index IDKEY On (Name,Age) [ Idkey ];

Property Name As %String;
Property Age As %Integer;
}
```

この場合、結果のグローバルは以下のようになります。

```
^MyApp.PersonD("Abraham",530) = $LB("",530,"Abraham")
^MyApp.PersonD("Philip",680) = $LB("",680,"Philip")
```

4.1.3 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り発生したフィールドは、追加ノードに格納されます。サブクラス名は、追加の添え字値として使用されます。

例えば、2 つのリテラル・プロパティを持つ、単純な永続クラス **MyApp.Person** を定義するとします。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
Property Name As %String;
Property Age As %Integer;
}
```

ここで、2 つの追加リテラル・プロパティを取り込む、永続サブクラス **MyApp.Student** を定義します。

```
Class MyApp.Student Extends Person [ClassType = persistent]
{
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生成し保存する場合、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")
```

Student クラスから取得したプロパティは追加サブノードに格納されますが、**Person** クラスから派生したプロパティはメイン・ノードに格納されます。この構造は、**Student** データが **Person** データとして確実に交互に使用できるようにします。例えば、すべての **Person** オブジェクト名を表示している SQL クエリは、**Person** データと **Student** データの両方を正確に取得します。また、この構造により、プロパティはスーパークラスかサブクラスのいずれかに追加されるため、クラス・コンパイラは、容易にデータ互換性を保持することができるようになります。

メイン・ノードの最初の部分は文字列 “Student” を含みますが、これが、**Student** データが含まれるノードを識別します。

4.1.4 親子リレーションシップ

親子リレーションシップ内で、子オブジェクトのインスタンスは属する親オブジェクトのサブノードとして格納されます。この構造は、子インスタンス・データが親データと物理的にクラスタ化するようにします。

例えば、以下は 2 つの関連したクラスの 1 つ、**Invoice** の定義です。

```
/// An Invoice class
Class MyApp.Invoice Extends %Persistent [ClassType = persistent]
{
Property CustomerName As %String;

/// an Invoice has CHILDREN that are LineItems
Relationship Items As LineItem [inverse = TheInvoice, cardinality = CHILDREN];
}
```

次に、**LineItem** は以下のようになります。

```
/// A LineItem class
Class MyApp.LineItem Extends %Persistent [ClassType = persistent]
{
Property Product As %String;
Property Quantity As %Integer;

/// a LineItem has a PARENT that is an Invoice
Relationship TheInvoice As Invoice [inverse = Items, cardinality = PARENT];
}
```

Invoice オブジェクトの複数のインスタンスを、それぞれを関連した **LineItem** オブジェクトとともに格納すると、結果のグローバルは以下のようになります。

```

^MyApp.InvoiceD = 2 // invoice counter node
^MyApp.InvoiceD(1) = $LB("", "Wiley Coyote")
^MyApp.InvoiceD(1, "Items") = 2 // lineitem counter node
^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 counter node
^MyApp.InvoiceD(2, "Items", 1) = $LB("", "Birdseed", 30)

```

リレーションシップ名は、追加のリテラル添え字として使用されます。これにより、クラスはデータ同士の衝突なしに複数のリレーションシップをサポートできます。また、**Invoice** の各インスタンスは、それ自体のカウンタ・ノードを保持し、**LineItem** オブジェクトに ID 値を割り当てます。

リレーションシップについての詳細は、“Caché オブジェクトの使用法” の “リレーションシップ” の章を参照してください。

4.1.5 埋め込みオブジェクト

埋め込みオブジェクトは、まずシリアル化された状態に変換され (既定では \$List で、オブジェクトのプロパティを含む構造です)、その後他のプロパティと同様に、そのシリアル状態で格納されます。

例えば、2 つのリテラル・プロパティを持つ、単純な連続した (埋め込み) クラスを定義するとします。

```

Class MyApp.MyAddress Extends %SerialObject [ClassType = serial]
{
Property City As %String;
Property State As %String;
}

```

前述の例を変更して、埋め込みの **Home** アドレス・プロパティを追加します。

```

Class MyApp.MyClass Extends %Persistent [ClassType = 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生成して保存する場合、結果のグローバルは以下のようになります。

```

^MyApp.MyClassD = 2 // counter node
^MyApp.MyClassD(1) = $LB(530, "Abraham", $LB("UR", "Mesopotamia"))
^MyApp.MyClassD(2) = $LB(680, "Philip", $LB("Bethsaida", "Israel"))

```

4.1.6 ストリーム

グローバル・ストリームは、それぞれが 32,000 バイトより小さくなるようにデータを一連の塊に分け、その塊をシーケンシャル・ノードに書き込み、グローバル内に格納します (ファイル・ストリームは、外部ファイルに格納します)。

4.2 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。インデックスは、処理（ソートや条件付き検索など）をさらに効率的にするために使用される追加のデータ構造です。Caché SQL は、クエリを実行するときに、このようなインデックスを使用します。Caché オブジェクト、および SQL は、挿入、更新、削除の処理が実行されるときに、インデックス内に自動的に正しい値を保持します。

4.2.1 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、順序付けられた 1 つ以上のプロパティ値を、プロパティを含むオブジェクトのオブジェクト ID 値に関連付けます。

例えば、2 つのリテラル・プロパティ、および **Name** プロパティにインデックスを持つ、単純な永続クラスの **MyApp.Person** を定義するとします。

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
  Index NameIdx On Name;

  Property Name As %String;
  Property Age As %Integer;
}
```

この **Person** クラスの複数のインスタンスを生成し保存する場合、結果のデータおよびインデックス・グローバルは以下ようになります。

```
// data global
^MyApp.PersonD = 3 // counter node
^MyApp.PersonD(1) = $LB(" ", 34, "Jones")
^MyApp.PersonD(2) = $LB(" ", 22, "Smith")
^MyApp.PersonD(3) = $LB(" ", 45, "Jones")

// index global
^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、以下の点に注意してください。

1. 既定では、“I”（インデックス）が付いたクラス名をグローバル名として持つグローバルに配置します。
2. 既定では、最初の添え字文字がインデックス名です。これにより、衝突することなく、同じグローバルへ複数のインデックスを格納することができます。
3. 2 つ目の添え字は、照合された データ値を含みます。この場合、データは既定の SQLUPPER 照合機能を使用して照合されます。これによって、(大文字小文字の区別なしにソートするために) すべての文字を大文字に変換し、(強制的にすべてのデータを文字列として照合するために) 空白文字を付加します。

4. 3つ目の添え字は、インデックス付きのデータ値を含むオブジェクトのオブジェクト ID 値を含みます。
5. ノード自体は空の状態です。必要なデータはすべて添え字内にあります。インデックス定義が、データをインデックスと共に格納すると指定する場合、インデックス定義はインデックス・グローバルのノード内に配置します。

このインデックスは、すべての **Person** クラスを **Name** の順番でリストするなど、多数のクエリを満たすのに十分な情報を含みます。

4.3 ビットマップ・インデックス

ビットマップ・インデックスは標準インデックスと類似していますが、唯一異なる点は、インデックス値に対応したオブジェクト ID 値のセットを、一連のビット文字列を使用して格納することです。

4.3.1 ビットマップ・インデックスの論理処理

ビット文字列は、一連のビット (0 と 1 の値) を特別な圧縮形式で含む文字列です。Caché は、ビット文字列を効率的に生成して使用するための機能を提供します。以下はその説明です。

ビット文字列処理

関数	説明
\$Bit	ビット文字列内のビットを設定または取得します
\$BitCount	ビット文字列内のビット数を数えます
\$BitFind	ビット文字列内でのビットの次の発生箇所を見つけます
\$BitLogic	2 つ以上のビット文字列に対し、論理演算 (AND、OR) を実行します

ビットマップ・インデックス内で、ビット文字列内の順番の位置は、インデックス付きのテーブル内の行 (オブジェクト ID 番号) に対応しています。ビットマップ・インデックスは任意の値に対して、その値が存在する各行には 1 を持ち、その値が存在しない各行には 0 を持った文字列を保持します。ビットマップ・インデックスは、システムが割り当てた数値のオブジェクト ID を持つ、既定のストレージ構造を使用しているオブジェクトに対してのみ作用します。

例えば、以下のような表があるとします。

ID	州	製品
1	MA	帽子
2	NY	帽子
3	NY	椅子
4	MA	椅子
5	MA	帽子

State および Product 列にビットマップ・インデックスがある場合、以下の値を含みます。

State 列のビットマップ・インデックスは以下の値を含みます。

MA	1	0	0	1	1
NY	0	1	1	0	0

州 が “MA” である行に対応した場所 (1、4、5) の “MA” 値には、1 があります。

同様に、Product 列のビットマップ・インデックスは、以下のビット文字列値を含みます (インデックス内では値は大文字に照合されています)。

椅子	0	0	1	1	0
帽子	1	1	0	0	1

Caché SQL エンジンは、これらインデックスにより保持されているビット文字列の繰り返し、文字列内部のビット数のカウント、または論理的な組み合わせ (AND、OR) を行うことにより、さまざまな演算を実行できます。例えば、州 が “MA” で、製品 が “帽子” である行を見つけるには、SQL エンジンは、該当するビット文字列に対し AND 演算を実行します。

システムは、これらのインデックスに加え、“エクステント・インデックス” と呼ばれる別のインデックスを保持します。これは、存在する行すべてに 1 を含み、存在しない行 (削除された行など) すべてに 0 を含みます。これは否定演算子など、特定の演算に対して使用します。

4.3.2 ビットマップ・インデックスのストレージ構造

ビットマップ・インデックスは、順序付けられた 1 つ以上の値の集合を、プロパティ値に対応したオブジェクト ID を持つ 1 つ以上のビット文字列と関連させます。

例えば、2 つのリテラル・プロパティとビットマップ・インデックスを持つ単純な永続クラスの MyApp.Person を、クラスの Age プロパティに定義するとします。

```

Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
Index AgeIdx On Age [Type = bitmap];

Property Name As %String;
Property Age As %Integer;
}

```

この **Person** クラスの複数のインスタンスを生成し保存する場合、結果のデータおよびインデックス・グローバルは以下ようになります。

```

// data global
^MyApp.PersonD = 3 // counter node
^MyApp.PersonD(1) = $LB("",34,"Jones")
^MyApp.PersonD(2) = $LB("",34,"Smith")
^MyApp.PersonD(3) = $LB("",45,"Jones")

// index global
^MyApp.PersonI("AgeIdx",34,1) = 110...
^MyApp.PersonI("AgeIdx",45,1) = 001...

// extent index global
^MyApp.PersonI("$Person",1) = 111...
^MyApp.PersonI("$Person",2) = 111...

```

インデックス・グローバルについて、以下の点に注意してください。

1. 既定では、(インデックスの) “I” が付いたクラス名をグローバル名として持つグローバルに配置します。
2. 既定では、最初の添え字文字がインデックス名です。これにより、衝突することなく、同じグローバルへ複数のインデックスを格納することができます。
3. 2つ目の添え字は、照合されたデータ値を含みます。これは数値データのインデックスのため、照合関数は適用されません。
4. 3つ目の添え字は、いわゆる チャンク 番号を含みます。効率性を上げるため、ビットマップ・インデックスはそれぞれがテーブルのおよそ 64,000 行の情報を含む一連のビット文字列に分けられます。これら各ビット文字列がチャンクと呼ばれます。
5. ノードはビット文字列を含みます。

また、この表にはビットマップ・インデックスが含まれるため、エクステント・インデックスが自動的に保持されます。このエクステント・インデックスは、インデックス・グローバル内に格納され、最初の添え字同様 “\$” 文字の付いたクラス名を使用します。

4.3.3 ビットマップ・インデックスの直接アクセス

以下の例はクラス・エクステント・インデックスを使用して、保存されたオブジェクト・インスタンス (行) の合計数を計算します。\$Order 使用して、エクステント・インデックスのチャンクを繰り返します (各チャンクはおよそ 64,000 行の情報を含みます)。

多次元ストレージの SQL およびオブジェクトの使用法

```
/// Return the number of objects for this class.<BR>
/// Equivalent to SELECT COUNT(*) FROM Person
ClassMethod Count() As %Integer
{
    New total,chunk,data
    Set total = 0

    Set chunk = $Order(^MyApp.PersonI("$Person",""),1,data)
    While (chunk '= "") {
        Set total = total + $bitcount(data,1)
        Set chunk = $Order(^MyApp.PersonI("$Person",chunk),1,data)
    }

    Quit total
}
```