

# データベース・レベルでの インピーダンス・ミスマッチとの戦い

テクニカル・ホワイト・ペーパー :

Mary A. Finn

プロダクト・マーケティング・マネージャ

InterSystems Corporation

---

---

## はじめに

Java の成熟と浸透によって、アプリケーション開発の世界では、オブジェクト指向プログラミングが主流となっています。今日のアプリケーション開発者が Java、C++、COM などのオブジェクト・テクノロジーを好んで用いるのは、それらが持つリッチなデータ・モデルと、生産性の向上を実現するカプセル化、継承、多態性などの概念のサポートによります。

しかし、その一方で、現実世界の大半のデータは、今日でもリレーショナル・データベースに格納されています。データベース・アプリケーション（格納されているデータにアクセスするアプリケーション）の開発者の多くは、オブジェクト・データ・モデルとリレーショナル・データ・モデルの本質的な相違に起因するインピーダンス・ミスマッチへの対処に多くの労力と時間が費やされている現状に頭を痛めています。リレーショナル・データを適切なオブジェクト構造に "マップ" するための多大な作業は、プログラマの生産性とアプリケーション・パフォーマンスの両方にとって大きな障害となっています。

一方、インピーダンス・ミスマッチは、適切なデータベース・テクノロジーを選択することにより回避できるという事実があります。このホワイト・ペーパーでは、最初にインピーダンス・ミスマッチを定義し、それがアプリケーション開発にどのように影響するかを示す 2 つの簡単な例を紹介します。その後で、リレーショナル・データベース、オブジェクト・データベース、および InterSystems が提供する多次元データベースの Cache という 3 種類のデータベースについて、インピーダンス・ミスマッチの観点からそれぞれの長所と短所を検証します。

## インピーダンス・ミスマッチとは

インピーダンス・ミスマッチとは、本来は電気工学の用語ですが、ソフトウェアの世界では、リレーショナル・データ・モデルとオブジェクト・データ・モデル間の本質的な相違を意味します。

簡単に言うと、リレーショナル・モデルでは、すべてのデータが行と列に編成されます。各行は 1 件のレコードを表し、各列はレコードが持つ様々なデータ項目を表します。データが非常に複雑で 2 次元のグリッドで表現できない場合は、さらにテーブルを作成して "関連付けられた" 情報を格納します。このように、リレーショナル・スキーマの各テーブルには、非常に多くのレコードの一部の（すべてではない）データ項目が格納されます。

一方、オブジェクト・データ・モデルには、データの格納に行と列を使用するという制約がありません。その代わりに、開発者は、情報の格納に使用するクラスを完全に記述した定義、つまりテンプレートを作成します。各レコード（オブジェクト）は、そのクラスの固有のインスタンスとなります。したがって、1 つのオブジェクトに、1 件のレコードの（1 件のレコードのみの）すべてのデータ項目が格納されます。しかし、これだけではありません。クラス定義には、クラスで記述されるデータを操作する、メソッドと呼ばれるコード断片を含めることもできます。リレーショナル・モデルには、これに相当する仕組みはありません。

## 簡単な例

2 つのデータ・モデルの違いを具体的に説明するために、売掛金勘定アプリケーションを開発することを想定します。売掛金勘定アプリケーションでは、多数の請求書を追跡管理できることが絶対要件であり、各請求書は、特定のヘッダ情報（請求書日付や請求書番号など）と 1 つ以上の明細行で構成されます。各明細行には、受注製品とその受注数量が製品別に記載されます。

リレーショナル・データベースで請求書をモデル化する場合は、一般的に 2 つのテーブルを作成します。1 つは Invoice テーブルで、各請求書が持つ固有のヘッダ情報を格納します。もう 1 つは LineItems テ

---

---

ーブルで、これには Invoice\_Parent、Line\_Item\_Product\_Code、および Line\_Item\_Quantity という 3 つの列を作成します。中でも最初の列は特に重要です。というのは、この列の値によって、明細行情報が Invoice テーブルの情報に "関連付けられる" からです。

どちらのテーブルにも、特定の請求書のすべての情報が格納されない点に注意してください。その代わりに、両テーブルには、多数の請求書の一部の情報が格納されます。例えば、アプリケーションに請求書の印刷機能を設計する場合は、Invoice テーブルと Lineltems テーブルの両方にアクセスして、ヘッダ情報と明細情報を別々に取得する必要があります。また、これらのテーブルには、印刷時にデータをフォーマットする方法が組み込まれていないことにも注意してください。これらの手順は、データベース本体の外部に配置されます。

オブジェクト・モデルでは、データを行と列に編成する必要がないため、Invoice クラスの定義は、請求書を構成するすべてのデータ項目の一覧のようになります。Invoice クラスは、InvoiceDate や InvoiceNumber などのヘッダ情報を格納するプロパティと、Lineltem クラスの 1 つ以上のインスタンスのコレクションで構成されます。Lineltem クラスには、ProductCode と LineltemQuantity という 2 つのプロパティがあります。

クラス定義は、データ構造の単なる青写真にすぎません。個々の請求書は Invoice クラスの特定のインスタンスとなり、各インスタンスにはそれに属する特定の Lineltem クラス・インスタンスが含まれます。したがって、各 Invoice オブジェクトには、特定の請求書の (特定の請求書のみ) すべての情報が含まれます。

また、クラス定義には、クラスで記述されるデータを操作するメソッドを含めることもできます。例えば、印刷時に請求書情報のフォーマット方法を指定する Print() メソッドを Invoice クラスに含めることができます。永続化オブジェクトには、データベースへのオブジェクトの格納方法を指定する Save() メソッド (または同等の他のメソッド) が含まれます。Save() メソッドのデフォルトの実装はデータベース・エンジンの構造によって決まり、データベース・ベンダから提供されます。

## データベース操作時のインピーダンス・ミスマッチ

売掛金勘定アプリケーションを使用して、明細行が 1 行の新しい請求書を作成する場合を考えます。リレーショナル・データベース用にプログラミングする場合は、例 1 に示すようなコードになります。このコードには 2 つの Insert 文があり、その 1 つはヘッダ情報を Invoice テーブルに挿入し、もう 1 つは明細情報を Lineltems テーブルに挿入します。Insert は標準の SQL コマンドで、各リレーショナル・データベース・ベンダがその実装を提供しています。

### 例 1 :

#### リレーショナル・モデルでの新規請求書の作成

```
If (flag="New") {
  Insert Into Invoice
  (Invoice_Date, Invoice_Number)
  Values(Today,;NewInvoiceNumber)
  Insert Into Lineltems
  (Invoice_Parent,Line_Item_Quantity,
  Line_Item_Product Code)
  Values(;NewInvoiceNumber,;Quantity,;ProdOrdered)
}
```

---

---

オブジェクト・モデルを使用して明細行が 1 行の請求書を保存するコードを、例 2 に示します。構文の細部を除くと、リレーショナルの例と同じようなコードになります。最も大きな違いは、Save() メソッドの呼び出しが 1 回のみという点です。

## 例 2： オブジェクト・モデルでの新規請求書の作成

```
If (flag="New") {  
  objInv=new Invoice()  
  objInv.InvoiceDate=Today  
  objInv.InvoiceNumber=NewInvoiceNumber  
  objLI=new ObjInv.LineItem()  
  objLI.LineItemQuantity=Quantity  
  objLI.ProductCode=ProdOrdered  
  objInv.Save()  
}
```

ここで、アプリケーションのビジネス・ロジックを Java や C++ などのオブジェクト指向言語で記述し、データをリレーショナル・データベースに格納する場合を想定します。請求書でこれを実現するには、Invoice クラス定義の Save() メソッドの中に SQL Insert 文をプログラミングする必要があります。ここに、1 つのインピーダンス・ミスマッチが生じています。つまり、このコードでは、コレクションを含むオブジェクト・クラスを、リレーショナル・データベース・エンジンの本質的に異なるテーブルに変換することが必要になっています。

## 設計時のインピーダンス・ミスマッチ

アプリケーションの設計プロセスで、別のタイプのインピーダンス・ミスマッチが発生します。オブジェクト・テクノロジーには、よりリッチで直観的なデータ・モデリング手法に加えて、プログラマの生産性を大幅に向上させる概念がいくつか組み込まれています。中でも、継承と多態性は、オブジェクト・テクノロジーでサポートされる有用な概念です。

継承は、特定のクラス定義を別クラスから派生させることができることを意味します。例えば、売掛金勘定アプリケーションでは、汎用的な InvoiceTemplate クラスを作成して、InvoiceTemplate クラスからプロパティとメソッドを継承する、より具体的な SoftwareInvoice クラスと HardwareInvoice クラスを作成できます (これらのクラスには、InvoiceTemplate クラスから継承されない、クラスに固有のプロパティとメソッドを含めることもできます)。アプリケーションの機能強化に伴って InvoiceTemplate が変更された場合は、継承の持つ特性により、これらの変更が自動的に SoftwareInvoice クラス定義と HardwareInvoice クラス定義に反映されます。

多態性は、メソッドの異なる実装で共通のインタフェースを共有できることを意味します。例えば、SoftwareInvoice と HardwareInvoice の Print() メソッドには、異なるフォーマット処理などが記述される場合がありますが、アプリケーションで請求書を印刷する際には、目的のオブジェクトをメモリにロードして、その Print() メソッドを呼び出すだけですみます。多態性のおかげで、そのオブジェクトは、それが属するクラスによって決定される、印刷時の自己のフォーマット方法を "認識" します。

継承も多態性も、リレーショナル・モデルにはない概念です。Oracle、MicroSoft、IBM などの一部の大手リレーショナル・データベース・ベンダは、オブジェクト指向設計の概念を実装しようと試行してきましたが、いずれもオブジェクト・プログラマが納得するだけの機能は実現できていません。

---

---

## インピーダンス・ミスマッチの軽減に向けてのアプローチ

ここまでで説明したインピーダンス・ミスマッチの 2 つの例は極度に単純化されたものですが、問題の背景を知るうえでは有効です。インピーダンス・ミスマッチの "正常化" に必要な対策の重要性は、アプリケーションの複雑度が増すにつれ劇的に高まります。その一方で、インピーダンス・ミスマッチがもたらす影響は、適切なデータベース・テクノロジーを選択することにより大幅に軽減できるという事実があります。ここではデータ・ストレージの 3 種類のオプションとして、リレーショナル・データベース、"純粋な" オブジェクト・データベース、および多次元データベースの Cache についてそれぞれ検証します。

### リレーショナル・データベースの使用

オブジェクト・テクノロジーに立脚したアプリケーションでリレーショナル・データベースを使用すると、必然的に重大なインピーダンス・ミスマッチ問題が生じることは既に説明済みです。しかし、開発者に選択の余地がない場合もあります。現状では、リレーショナル・データベースに既存するデータにアクセスする必要がある場合が頻繁にあります。こうした場合の 1 つのオプションとして "オブジェクト/リレーショナル・マッピング" ツールを使用する方法があります。オブジェクト/リレーショナル・マッピング・ツールには、スタンドアロンのものであれば、"オブジェクト・リレーショナル" データベースと呼ばれる一部のデータベースにマッピング機能として組み込まれたものもあります。

マッピング・ツールの実体は、マップ・ファイルを作成して、オブジェクトとリレーショナル・テーブル間で変換を実行するコードを記述することにあります。開発者は、変換の実行手順、つまり、どのオブジェクト・プロパティをどのテーブルのどの列に対応付けるのかを、正確に指定する必要があります (逆の場合も同様です)。作成されたマップは保存され、アプリケーションがデータベースからデータを読み書きするたびに起動されます。オブジェクト/リレーショナル・マッピング・ツールの中には、オブジェクトとリレーショナル構造間でのデータ変換に伴うパフォーマンス低下を軽減するランタイム・キャッシング・コンポーネントが組み込まれているものもあります。

オブジェクトとリレーショナルをマッピングするアプローチには、実行時のパフォーマンス問題以外に、アプリケーション開発の生産性が大幅に低下するという問題があるほとんどのマッピング・ツールには、継承や多態性などのオブジェクト・モデリング概念がまったく実装されていないか、実装されていても一部に限られるのが現状です。そのため、アプリケーションを拡張したり変更したりするたびに、新しく更新したオブジェクト/リレーショナル・マップ・ファイルを作成する必要が生じます。

オブジェクト指向アプリケーションとリレーショナル・データベース間のインピーダンス・ミスマッチと格闘する開発者は、よりオブジェクト・フレンドリなデータ・ストアにデータを移行できれば効果的と考えられるでしょう。開発者は、進行中のプロジェクトでデータの構造変更および変換作業を一度だけ行うのに必要な労力と、オブジェクト/リレーショナル・マッピングによって生じるパフォーマンス低下を天秤にかけて評価する必要があります。

### オブジェクト・データベースの使用

一見すると、"純粋な" オブジェクト・データベースにデータを格納すると、インピーダンス・ミスマッチは完全になくなると思えるでしょう。これは一部の側面では真実です。一般的に、オブジェクト指向アプリケーションがオブジェクト・データベースとやり取りするときは問題ありません。ただし、このシナリオでは、SQL クエリをオブジェクト・データベースに対して実行すると、インピーダンス・ミスマッチが生じます。クエリ言語としての SQL の浸透度は圧倒的で、この言語はデータがリレーショナル・データベースに格納されていることを前提としていま

---

---

す。一部のオブジェクト・データベース・ベンダは、オブジェクト・クエリ言語 (OQL) を介したデータ・アクセスを実装していますが、この言語は現時点では幅広く受け入れられていません。一般的なデータ解析アプリケーションやレポート・アプリケーションとの互換性を実現するには、オブジェクト・データベース側で ODBC および JDBC をサポートし、データをリレーショナル・データベースとして投影する何らかのメカニズムを実装する必要があります。

典型的なソリューションは、やはりマッピングです。マッピングの欠点であるパフォーマンスの低下とデータ・モデルの拡張に対するサポートの欠如が、ここでも問題となります。有利な点は、マッピングが必要となるのが、オブジェクト・データベースに対して SQL クエリを実行する場合に限られるということです。

## 統一データ構造が備わった多次元 Cache の使用

データ・ストレージの第 3 のオプションは、InterSystems が提供する多次元データベースの Cache です。一般的に、多次元データベースはデータ・ウェアハウス分野での機能とされていますが、Cache ではトランザクション処理アプリケーションの機能として設計されています。また、Cache には、統一データ構造という、インピーダンス・ミスマッチを軽減するユニークなアプローチが実装されています。

統一データ構造のおかげで、オブジェクト・データ・モデルとリレーショナル・データ・モデルが Cache の多次元データを "共有" します。多次元配列はテーブルとしての投影が容易です。なぜなら、テーブルは 2 次元配列そのものだからです。同様に、オブジェクトと多次元配列間の対応付けも容易です。なぜなら、どちらもリレーショナル・テクノロジーの行/列構造に制約されていないためです。データ構造の変換は自動化され、コンパイルされたデータ定義の一部となります。開発者にとっては、事実上、すべてのテーブルが 1 つのオブジェクトで、すべてのオブジェクトが 1 つ以上のテーブルになります。

### Cache の統一データ構造のその他の利点

- **完全な同時性** リレーショナル・インタフェースで作成されたデータに対する更新が、オブジェクト・インタフェースから即座にアクセス可能になります。また、その逆も同様です。
- **データ・モデルの拡張のサポート** データ構造の定義変更は、オブジェクトおよびリレーショナル表現の両方に自動的に反映されます。
- **SQL の完全サポート** SQL DDL、DML、および DCL のコマンドがすべてサポートされます。
- **オブジェクトの完全サポート** 単純継承、多重継承、多態性、高度なデータ型、メソッド・ジェネレータなどのオブジェクト・モデリング概念がすべてサポートされます。
- **各種オブジェクトとしてサービス提供** 統一データ構造で定義されたオブジェクトは、Java、C++、または COM オブジェクトとしてサービスを提供できるため、様々なオブジェクト指向テクノロジーと互換性があります。

統一データ構造によって、インピーダンス・ミスマッチは大幅に減少しますが、完全に取り除かれる訳ではありません。例えば、オブジェクト・メソッドやリレーショナル・トリガなどの一部の概念は、自動共有ができません。それでも、オブジェクトとリレーショナル・データ・アクセスの融合を模索している開発者にとって、Cache が有効な選択であることは変わりません。

---

---

## まとめ

ここ数年の間に、Java、C++、COM などのオブジェクト指向プログラミング言語が、アプリケーション開発テクノロジーの主流となってきました。そのため、データベース・アプリケーションの開発者にとっては、データをオブジェクトとして投影できることが必須となっています。

しかし、その一方で、データ解析やレポーティング・アプリケーションの分野では、SQL テクノロジーが圧倒的に浸透しています。その結果、アプリケーションを機能させるには、データをリレーショナル・テーブルとして投影でき ODBC や JDBC を介してアクセスできることが、あらゆるデータベースに求められます。

オブジェクト・データ・モデルとリレーショナル・データ・モデルの本質的な相違に起因するインピーダンス・ミスマッチは避けることができないものですが、適切なデータベース・テクノロジーを選択することにより大幅に軽減できます。新規アプリケーションを開発する場合は、Cache のような多次元データベースにデータを格納するのが最善です。Cache では、統一データ構造によって、データをオブジェクトとテーブルの両方として同時に投影できます。

進行中のアプリケーション開発において、既存データが既にリレーショナル・データベースに格納されている場合は、開発者はそのデータを多次元構造に変換することを検討したほうがよいでしょう。データ変換作業を一度行うだけで、オブジェクト/リレーショナル・マッピングが原因で生じるパフォーマンス低下と、データ・モデル拡張時の障害を回避できます。

\* この文章は、米国InterSystems社のWhite Paper “Fighting Impedance Mismatch at the Database Level” の翻訳です。ご不明の点は、英文本文をご参照ください。