

特集1 Part.1

システム構築の現場では、J2EE 準拠のアプリケーションサーバーの導入が進む一方、JDBC を直接使用した DB アクセスが未だに一般的だ。そこには、複雑な Java オブジェクトを単純な RDB データとして表わすという難題がある。これを打開するには、オブジェクト指向言語とRDBの表現力のギャップを埋める技術＝永続化が不可欠だ。永続化の導入は、DBエンジニアには大きな発想の転換をもたらす。本パートでは、この永続化とは何かを解説する。

オブジェクトの 永続化とは何か

📁 DB はメモリ上のオブジェクトの
退避場所

📁 O-R マッピングの苦勞からの脱
却と SQL 抜きの RDB アクセス



オブジェクト指向言語を 使わざるを得ない状況

本稿のテーマは「永続化」とは何かを理解することにあります。オブジェクト指向の世界では特に新しくはないけれど、筆者を含めた DB エンジニア、DB アプリケーションプログラマにはあまり縁がなかったこの言葉が、最近よく聞かれるようになりました。本パートでは、企業システムを構築するための環境がどう変化し、なぜ永続化を理解する必要があるのかを見ていくことにします。

数年前、Java は使いものになるのか否

注1: 最近ではWebサービスや非同期メッセージ配信機能などを統合し、適用範囲が広がっているために、本稿ではアプリケーションサーバーを“Web”アプリケーションサーバーと限定した呼び方をしないことにします。アプリケーションサーバーが利用されるシステムの多くがWebアプリケーションであることに間違いはありません。

かという議論が盛んに行なわれました。これに関連して、オブジェクト指向言語のオーバーヘッドによるパフォーマンスの低下を補えるほど、オブジェクト指向には“ご利益”があるのかという話題もありました。オブジェクト指向推進派、懐疑派、どちらの言い分にも一理あると思われましたが、今や結果は明らかです。

エンタープライズレベルのシステムを開発するには、好むと好まざるとに関係なく、オブジェクト指向言語を使わざるを得ません (Java しか .NET しか)。さらに、一部の例外を除き、アプリケーションサーバー＝J2EE (Java2 Platform, Enterprise Edition) サーバーを使うという現実があります^{注1}。

一方、DB エンジニアに「あなたが知っている RDBMS は？」と問えば、多くの人が最初に「Oracle」と答えるのではないのでしょうか。最近では、IBM の DB2 UDB もシェアを伸ばしているようですが、RDBMS という製品の市場を作り、牽引してきたのが Oracle であることに異論はないと思います。

J2EE アプリケーションサーバーで同様のポジションにあるのが BEA システムズの「WebLogic Server」です。WebLogic Server はこの市場でシェア、実績ともに優位に立っています。BEA システムズは、ほぼアプリケーションサーバー関連製品のための専業ベンダですが、これを追い、DB エンジニアにも馴染みの RDBMS 製品も持つソフトウェアベンダが続きます。IBM のアプリケーションサーバー「WebSphere Application Server」も J2EE 準拠の製品です。その他の RDBMS ベンダが出荷している製品も、多くが J2EE 準拠となっています (表1)。

Oracle のアプリケーションサーバーも、WAS (Web Application Server) と呼ばれていたころには独自アーキテクチャを採用していましたが、OAS (Oracle Application Server) で J2EE をサポートしまし

た。そして、現行の9iAS (Oracle9i Application Server) では、完全に J2EE にフォーカスしています。過去の資産継承のために従来のアーキテクチャもサポートしていますが、プロモーション的にも J2EE 準拠が前面に打ち出されています。日本オラクルの雑誌広告にある「本物 J2EE 600,000 円」という印象的なコピーは記憶に新しいところです。

いずれにせよ、今やごく一部の製品を除いて、アプリケーションサーバーは J2EE のインフラストラクチャ上に構成されています。J2EE 以外の独自アーキテクチャを採用しているのは、マイクロソフトの .NET と「PHP」^{注2}や「ZOEPE」^{注3}などオープンソース系の製品に限られます。一部で根強いシェアを持つマクロメディアの「Cold-Fusion」も、最新バージョンで J2EE ベースに移行しつつあるなど、独自アーキテクチャの製品は今後も減り続けると予想されます。

では、現在主流の 3 層システムの中核に位置するアプリケーションサーバーがオブジェクト指向ベースへと移行していくのに伴い、データストア^{注4}も OODB (Object Oriented DataBase: オブジェクト指向データベース) に移行していくのかというと、今のところ、その動きは見られません。ここ数年間は引き続き RDB が主流であると思われる (理由は次節)。

企業システムを取り巻くこのような状況をまとめると、次のような結論になります。

- アプリケーションサーバーは J2EE 準拠のものが主流となっている。
- DBMS としては、多くのシステムで相変わらず RDBMS が使われている。

なぜ OODB ではなく RDBなのか

プログラミング言語が Java になり、アプリケーションサーバーが J2EE 準拠のもの

表 1 ● 主要 J2EE 準拠アプリケーションサーバー製品

| ベンダ | アプリケーションサーバー(上段) RDBMS(下段) | Web サイト | J2EE 対応バージョン |
|--------------|--|---|--------------|
| BEAシステムズ | WebLogic Server 7.0J — | http://www.beasys.co.jp/ | 1.3 |
| IBM | WebSphere Application Server V4.0 DB2 UDB V7.2 | http://www.ibm.com/jp/software/ | 1.2 + 1.3の一部 |
| サン・マイクロシステムズ | iPlanet Application Server 6.5 — | http://www.sun.co.jp/ipplanet/ | 1.2 |
| オラクル | Oracle9i Application Server R2 Oracle9i Database R2 | http://www.oracle.co.jp/ | 1.2 + 1.3の一部 |
| サイベース | Enterprise Application Server 3.6 Adaptive Server Enterprise 12.5 | http://www.sybase.co.jp/ | 1.2 + 1.3の一部 |
| ボーランド | Borland Enterprise Server InterBase 6.0 | http://www.borland.co.jp/ | 1.3 |

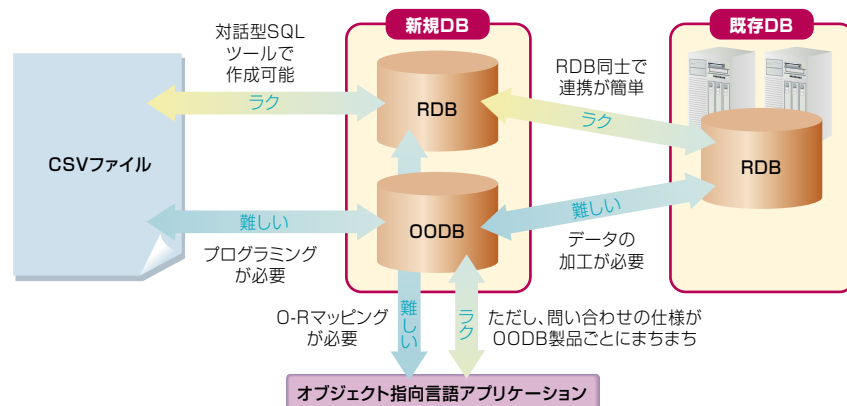


図 1 ● RDB と OODB

になっても、DBMSとしてOODBが使われないのはなぜでしょう (図1)。

最大の理由は、レガシーシステムと連携する上での都合だと思われます。どの企業も膨大なデータをRDBに格納しており、それらの活用を考えた場合、新システムを構築する場合もRDBにしておいたほうが連携しやすいことは間違いありません。

分散データベース機能で直接つなぎ込まないとしても、「データをCSV形式で出力したい。また、CSV形式のデータを取り込めるようにして欲しい」という要件は、非常に多くあります。このようなときに、OracleのSQL*PlusやSQL*Loaderのような対話的にSQLを発行するツールで簡単に対応できるRDBに対し、OODBではCSVファイルを読み書きするプログラムを書かなければなりません。

また、OODB製品に依存しないオブジェクト検索を目的に、ODMG^{注5}が定めている規格「OQL」(Object Query Language)が製品に実装されないのも、OODB普及の足かせとなっています。OODBは製品間の互換性が非常に低く、製品ごとに異なったAPIを学ばなければなりません。

OODBも決して新しいものではありませんし、特定の分野で非常によく使われている

注2: HTML内に記述するタイプのスクリプト言語。HTTPサーバーと連携して、処理結果をHTMLページ中に出力する。

注3: スクリプト言語「Python」で記述されたアプリケーションサーバー。Pythonで動的なWebページを記述できる。

注4: データソース、EIS (Enterprise Information Systems)、永続記憶などとも呼ばれます。要は、システムが使用するデータの格納場所のことです。通常はデータベースですが、OSのファイルシステムやLDAPなどのディレクトリサービスの場合もあります。

注5: Object Data Management Group (<http://www.odmg.org/>)。OODBベンダによる標準化団体。

特集1 “永続化” 徹底入門

ます。しかし、ここに来て急にOODBに追い風が吹くような要素ありません。

こうした状況から、1年以内にOODBがRDBに取って代わることはあり得ないで

しょう。

オブジェクト指向言語に非オブジェクト指向なデータベースという“ねじれ状態”が、実は最も有力な現実解であるわけです。

O-R マッピングとは？

システムがオブジェクト指向アーキテクチャに移行しつつある中でも、データの保管場所（データストア）には、2層クライアント／サーバーシステムやCGI Webアプリケーション時代から変わらず、RDBが利用

されています。このため、プログラム中ではオブジェクトとして存在しているデータを、RDBに格納するための変換作業が必要となっています。これをオブジェクト・リレーショナルマッピング（O-Rマッピング）と言います。

オブジェクトをRDBに格納する場合、O-Rマッピングは避けて通ることはできません。これは、永続化の概念を導入しても本質的には変わりません。永続化というオペラートに包まれて、O-Rマッピングのコードが隠蔽されているのか否かの違いがあるだけです。

インピーダンス不整合

複雑なデータ構造を表現することのできるオブジェクトに対し、RDBのリレーショナルモデルは、すべてのデータを2次元の表形式で表わします。現在、多くのRDBMSが準拠している規格「SQL92」のレベルでは、扱えるデータ型も文字列、数値、日付など、極めてプリミティブな型に限られます。

このように、オブジェクト指向言語で表現できるデータ（オブジェクト）と、RDBで扱えるデータには、表現力や自由度に大きなギャップがあります。これをO-Rマッピングにおける“インピーダンス不整合”と言います（図2）。

複雑な構造を持つデータを、簡単な構造のデータしか扱うことのできない箱にしまわなければならないのですから、O-Rマッピングはなかなか困難な作業となります。筆者は、O-Rマッピングのためのコードがプログラム全体の30%を占めているという統計結果を聞いたことがあります。30%という数字の信憑性はともかく、インピーダンス不整合を埋めるためには、単純作業ながら骨の折れる、高コストな処理を強いられることは間違いありません。

例として、図3のクラス図で表わされる顧客データを、O-Rマッピングを行ない、図

データ型の違い
表現力、自由度の違い

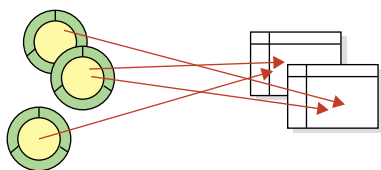


図2 ●インピーダンス不整合

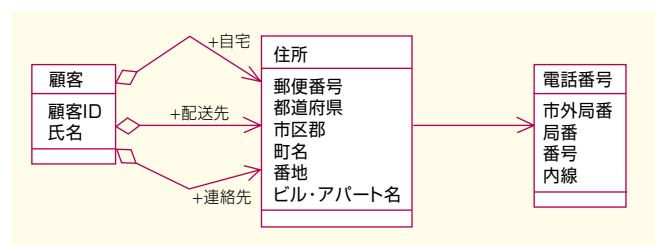


図3 ●顧客クラスのクラス図

顧客

| 顧客ID |
|-------------|
| 氏名 |
| 自宅-郵便番号 |
| 自宅-都道府県 |
| 自宅-市区郡 |
| 自宅-町名 |
| 自宅-番地 |
| 自宅-ビルアパート名 |
| 自宅-市外局番 |
| 自宅-局番 |
| 自宅-番号 |
| 自宅-内線 |
| 配送先-郵便番号 |
| 配送先-都道府県 |
| 配送先-市区郡 |
| 配送先-町名 |
| 配送先-番地 |
| 配送先-ビルアパート名 |
| 配送先-市外局番 |
| 配送先-局番 |
| 配送先-番号 |
| 配送先-内線 |
| 連絡先-郵便番号 |
| 連絡先-都道府県 |
| 連絡先-市区郡 |
| 連絡先-町名 |
| 連絡先-番地 |
| 連絡先-ビルアパート名 |
| 連絡先-市外局番 |
| 連絡先-局番 |
| 連絡先-番号 |
| 連絡先-内線 |

図4 ●顧客データを保存するテーブル

LIST1 ● Customer クラスの定義

```
package sample1;

public class Customer {
    long id;
    String name;
    Address house;
    Address destination;
    Address contact;
}
```

LIST2 ● Address クラスの定義

```
package sample1;

public class Address {
    String zip;
    String prefecture;
    String city;
    String street;
    String houseNumber;
    String buildingName;
    Phone phone;
}
```

LIST3 ● Phone クラスの定義

```
package sample1;

public class Phone {
    String area;
    String exchange;
    String number;
    String extension;
}
```

LIST4 ● 図3のテーブルを作成するSQL文

```
CREATE TABLE CUSTOMER (
  ID                INTEGER    PRIMARY KEY,
  NAME              VARCHAR(32),
  HOUSE_ZIP         CHAR(8),
  HOUSE_PREFECTURE  VARCHAR(10),
  HOUSE_CITY        VARCHAR(24),
  HOUSE_STREET      VARCHAR(32),
  HOUSE_HOUSE_NUMBER VARCHAR(32),
  HOUSE_TEL_AREA    VARCHAR(4),
  HOUSE_TEL_EXCHANGE VARCHAR(6),
  HOUSE_TEL_NUMBER  VARCHAR(4),
  HOUSE_TEL_EXTENTION VARCHAR(6),
  DEST_ZIP          CHAR(8),
  DEST_PREFECTURE   VARCHAR(10),
  DEST_CITY         VARCHAR(24),
  DEST_STREET       VARCHAR(32),
  DEST_HOUSE_NUMBER VARCHAR(32),
  DEST_TEL_AREA     VARCHAR(4),
  DEST_TEL_EXCHANGE VARCHAR(6),
  DEST_TEL_NUMBER   VARCHAR(4),
  DEST_TEL_EXTENTION VARCHAR(6),
  CONTACT_ZIP       CHAR(8),
  CONTACT_PREFECTURE VARCHAR(10),
  CONTACT_CITY      VARCHAR(24),
  CONTACT_STREET    VARCHAR(32),
  CONTACT_HOUSE_NUMBER VARCHAR(32),
  CONTACT_TEL_AREA  VARCHAR(4),
  CONTACT_TEL_EXCHANGE VARCHAR(6),
  CONTACT_TEL_NUMBER VARCHAR(4),
  CONTACT_TEL_EXTENTION VARCHAR(6)
);
```

LIST5 ● 顧客クラスのオブジェクトを顧客テーブルに格納するO-Rマッピングの実装

```
package sample1;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class ORMappingSample {

    public void store(Customer cust) throws SQLException {
        Connection conn = DriverManager.getConnection("...");
        PreparedStatement stmt = conn.prepareStatement(
            "INSERT INTO CUSTOMER" +
            " (ID, NAME, HOUSE_ZIP, HOUSE_PREFECTURE, HOUSE_CITY, HOUSE_STREET," +
            " HOUSE_HOUSE_NUMBER, HOUSE_TEL_AREA, HOUSE_TEL_EXCHANGE," +
            " HOUSE_TEL_NUMBER, HOUSE_TEL_EXTENTION, DEST_ZIP," +
            " DEST_PREFECTURE, DEST_CITY, DEST_STREET, DEST_HOUSE_NUMBER," +
            " DEST_TEL_AREA, DEST_TEL_EXCHANGE, DEST_TEL_NUMBER, " +
            " DEST_TEL_EXTENTION, CONTACT_ZIP, CONTACT_PREFECTURE," +
            " CONTACT_CITY, CONTACT_STREET, CONTACT_HOUSE_NUMBER," +
            " CONTACT_TEL_AREA, CONTACT_TEL_EXCHANGE, CONTACT_TEL_NUMBER," +
            " CONTACT_TEL_EXTENTION)" +
            " VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?" +
            " ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"
        );
        stmt.setLong(1, cust.id);
        stmt.setString(2, cust.name);
        stmt.setString(3, cust.house.zip);
        // ...
        stmt.setString(28, cust.contact.phone.number);
        stmt.setString(29, cust.contact.phone.extention);
        stmt.executeUpdate();
        stmt.close();
        conn.close();
    }
}
```

4のテーブルに保存するものとします。

DBエンジニアが見ると、図3の顧客テーブルは正規化されているのか議論の余地があるかと思いますが、筆者は自宅、配送先、連絡先を繰り返し項目とは捉えていません。仮に論理設計で正規化の結果、住所の項目を別テーブルに切り出すべきと主張する人でも、物理設計では非正規化を行なって、筆者と同様の実装をするのではないのでしょうか。

LIST1～LIST3は、図2のクラス図を実装したものです。LIST4は、図3のテーブルを作成するSQL文です。LIST5は、O-Rマッピングを行なって顧客クラスのオブジェクトを顧客テーブルに格納するクラスの実装となります。本来なら、顧客テーブルの主キーである「顧客ID」を採番する処理も必要となりますが、説明を簡単にするために割愛しています。

また、クラスの各属性の可視性は

private（隠蔽）にし、public（公開）なgetter/setterメソッド^{注6}を用意するのがセオリーですが、こちらも簡単にするためにパッケージスコープとし、属性に直接アクセスできるようにしています。

O-R マッピングの問題点

O-R マッピングの問題点としては、次のものが挙げられます。

① 関連のある複数のクラスで表わされていたオブジェクトを、1つのテーブルに格納するため、フラットな形への変換が必要になっている

この変換はツールなどを使って自動化することはできないため、プログラマがクラスごとに実装しなければなりません。格納のみならず、更新や検索結果を取得するときにも、同じ変換のコードを繰り返し書く必要があります。

② プログラマは、クラスの定義もテーブルの定義も正確に知っていなければならない

オブジェクト指向言語の知識とSQLの知識の両方が必要です。

③ テーブル名や列名など、格納先テーブルの情報が一元管理されていない

更新や検索のメソッドを追加するときにも、テーブル名や列名を正確に繰り返し書かなければなりません。

④ オブジェクトでは参照として表わすものをRDBでは値で表わさなければならないため、「○○ID」といった新たな列を導入して関連を表わさなければならない場合がある

ユニークなIDを採番するための仕掛け

注6: オブジェクトの属性を読み (getter)、書き (setter) するメソッドのことを総じて言う。

特集1 “永続化” 徹底入門

をシステムごとに用意しなければなりません^{注7}。ただし、先の例では、複数テーブルにマッピングされていないので、新たなIDの導入は発生しません。

永続化の概念の導入

システム構築の現場では、J2EE 準拠のアプリケーションサーバーの導入が進んでいますが、多くのシステムでは、JDK 1.1 のころから変わらず、JDBC の API を直接使用した DB アクセスが行なわれていると思います。そこでは、先に挙げたような O-R マッピングが行なわれており、プログラマはインピーダンス不整合との格闘を日々強いられています。

この状況を打開するには、オブジェクト指向言語と RDB の表現力のギャップを埋める技術を導入する必要があります。それが、オブジェクト指向パラダイムで「永続化」と呼ばれるものです(オブジェクトに持続性を持たせるとも言います)。

永続化ではデータストアを抽象化して考えるため、データストアが RDB であることに依存しない方法、RDB であることが十分に隠蔽されている方法が提供されます。もちろん、永続化の方法によっては RDB を意識しなければいけないものもありますし、現状では多くの方法で完全にデータストアの種類を隠蔽できてはいません。

以降では、まず一般論としての永続化を説明し、そのあとで本誌読者の関心の中心であると思われる、RDB へのオブジェクト永続化を考察していきます。

永続化とは

一般に永続化とは、メモリ上のデータを

^{注7}：JDBC 3.0 でキー値を自動生成する標準的な方法が提供されるようになりました。しかし、JDBC 3.0 対応のドライバの普及には、まだまだ時間がかかりそうです。

^{注8}：1970 年代に米ゼロックス社パロアルト研究所のラン・ケイ氏らによって開発されたオブジェクト指向言語。

ハードディスクなど、不揮発性(電源を切っても消えない)の外部記憶装置に格納することです。オブジェクト指向の場合は、「データ=オブジェクト」となります。

メモリ上のオブジェクトは、アプリケーションが終了した時点で失われます。次にアプリケーションを起動したときに、前回の状態を引き継ぐことはできません。もちろん、アプリケーションが予期せず異常終了した場合には、すべてが水泡に帰してしまいます。

このようなことがないように、通常アプリケーションは、ドキュメントや入力したデータを「保存」する機能を持っています。オブジェクト指向で開発されたアプリケーションであれば、「ドキュメント」やその他「データ」と呼ばれるものはオブジェクトとしてメモリ上にあるので、「保存」とはオブジェクトを永続化することにほかなりません。

最も素朴な永続化は、オブジェクトの保持するデータ(属性値)をファイルに保存することです。この方法は、プレーンテキストのように単純なデータ構造であればうまくいきますが、顧客情報など業務アプリケーションの複雑なデータをフラットファイルに格納するのはかなり困難な作業となります(詳細はパート2を参照)。

もう1つのシンプルな永続化の例として、ノートパソコンのハイパネーション(Windows での「休止状態」)のように、アプリケーションが使用しているメモリイメージをそのままディスクに保存してしまう方法もあります。次回、そのアプリケーションを起動するときに、メモリダンプを読み込んで前回の状態に戻すわけですが、かなり乱暴な方法に思われるかもしれませんが、Smalltalk^{注8}では古くから使われてきた手法で、ほとんどの処理系ではメモリダンプをファイルに落とす機能を持っています。

オブジェクト指向的発想への転換

Visual Basic (以下、VB) などを使った2層のクライアント/サーバーシステムや、CGI による Web アプリケーションなど、どんなシステムにおいてもデータベースにデータを格納することは一般的に行なわれます。データが消えないように2次記憶装置に格納するという点では、立派な「永続化」です。

しかし、オブジェクト指向パラダイムでの永続化となると、手続き型プログラミングでのデータ保存とは若干、発想を転換しなければなりません。また、永続化を行なうためのシステムの仕掛けや、ミドルウェアあるいはデータマネジメントシステムに求められるものも変わってきます。

ここでは、O-R マッピングを使った RDB へのデータ格納と対比しつつ、オブジェクトの永続化を見ていきます。

背景の違い

DB アプリケーションプログラマが書くコードと、オブジェクト指向に慣れたプログラマが書くコードは違ったものになります。根本的に、分析や設計のフェーズから、考え方や方法論が異なっているのです。

一般的にプログラマは、開発言語やミドルウェアが変わっても従来のスタイルをできるだけ踏襲したいと考えます。DB アプリケーションの開発に慣れたプログラマならば、世の流れで開発言語が VB や Perl から Java に変わったとしても、RDB へアクセスする API が ODBC や DBI (Database Interface) から JDBC に変わるだけのことで、SQL には変わりがないと考えるでしょう。

しかし、最近幅をきかせているオブジェクト指向に精通したエンジニア(筆者にとっても彼らは未だに向こう岸の人です