

db4o | オープンソースデータベース | Java and .NET

複雑なオブジェクト構造、オブジェクトの永続化、そしてdb4o

By Rick Grehan

オブジェクト指向プログラミングの長所の1つは、これは実際長所の中でもコアとなるものですが、データを自然にモデル化できるということです。これはモデリングプロセスが簡単になるという意味ではありません。実世界のたくさんのシステムは、サブシステムとそのサブシステムというように複雑なツリー構造をしています。他のシステムと相互運用をするシステムは、どれも同様に複雑です。そのようなシステムで使われるクラス構造は結果的に、非常にやっかいで、プログラムには細心の注意を必要とします。

ここにデータの永続化を加えると、ますますやっかいになってきます。このような複雑で、相互作用を行う、マルチクラスシステムのデータとメソッドのコードを書くことと、素早くそして簡単にオブジェクトを取得し、検索し、更新し、そして削除できるように、なおかつこの間リレーションをキープできるようにオブジェクトを格納することは全く別のことです。それではこれがどのようにやっかいか見てみましょう。複雑で巨大なオブジェクト構造を永続化、一般的な言い方をすればデータベースへ書き込みをする際に問題に直面します。それではいくつかの例を通して説明しましょう。仮想ではあるものの十分に実世界のプログラミングを代表することができます。

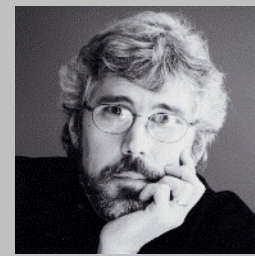
- フラットオブジェクト構造
- 背の高いオブジェクトツリー構造
- 進化するオブジェクト構造

それでは、パワフルなオブジェクト指向データベースであるdb4oを使うと、それらの問題がどのように楽に解決されるかをお見せしましょう。db4oによって、アプリケーションに最適なクラスやオブジェクト構造の作成に専念できることがすぐに分かります。データベースの操作とオブジェクト構造のバランスを考慮する必要はありません。言い方を代えると、db4oは永続化レイヤーに関する開発者の負担を最小にするのです。特別なコードは不要ですし、データベースに合うようにオブジェクト構造の調整も不要です。オブジェクトのまま、格納、取得、削除されます。

残りはdb4oが行います。

Table 1. RDBMSはフラットオブジェクト構造に対してはうまく対処できます。ネイティブでないOODBMSは、さらに大きなオブジェクトツリー構造を扱うことができます。db4oのみが、さらに進化するオブジェクト構造に対応することができます。

ability to easily store...	RDBMS	non-native OODBMS	db4o
flat object structures	X	X	X
tall object trees		X	X
evolving object structures			X



Rick Grehan氏はCompuware/Numega研究所の品質管理エンジニアで、Javaと.NETのプロジェクトに参加しています。また、組み込みシステムプログラミング、EDN、マイクロプロセッサレポート、コンピュータデザインに関して、InfoWorld誌の執筆を行っています。Compuwareの以前は、Metrowerks社でDiscoverer DPSプロジェクトに参加していました。さらにそれ以前には、雑誌BYTEのSenior Editorをしていて、Lab Directorであり、JavaTalkコラムを執筆していました。



db4oとは？

オブジェクトデータベースライブラリであるdb4o(database for objects)は、db4objects社によって提供されている、Javaと.NETにネイティブな、ピュアオブジェクト指向データベースです。最大の特徴は、スピード、操作の容易さ、小メモリ消費です。db4oは普通のオブジェクトを格納することができます。ネイティブでないオブジェクトデータベースは、格納するためのコードを付け足す必要があります。それ以外のものは、別に“スキーマ”ファイルを用意しなければなりません。しかしdb4oを使えば、データベースに格納するためやデータベースのデータを操作するための余分な、つまり本来不要なコードを書く必要がありません。

これからお見せするサンプルはJavaで記述されていますが、前述したように、.NETバージョンも利用可能です。そのため、ここで取り上げるコンセプトは、Javaに適用可能なように.NETにも適用可能です。¹

フラットオブジェクト構造

どれだけたくさんのオブジェクトを含んでいても、単純なオブジェクト構造のことを“フラット”構造といいます。通常大きな配列や、リンクリストを含んでいます。そのような構造を永続化させる場合、まずそれらの配列やリンクリストを全てメモリ内に置くかどうかを決定する必要があります。これらの配列やリストは、数千から数百万のオブジェクトを含んでいるものとします。そうすると、利用できるメモリのサイズと、予想される配列やリストの大きさとのバランスで決定されることとなります。

もし配列のサイズがメモリに収まるなら、丸ごと1つのものとして永続化できるでしょう。そうでない場合は、データベースへの書き込みや読み取りは、ばらばらにされなければなりません。

例1: ウェブアプリケーションのトランザクションを監視する

ある組織が、ウェブサービスを使って顧客にサービスを提供しているとします。クライアントのサイトはこのアプリケーションに接続し、読取や更新などのトランザクションをPOSTします。その組織はこれらのトランザクションを、任意の時刻に任意の間隔で、このアプリケーションへのリクエストの量と応答時間を監視したいとします。

そこで、この組織の開発者は、サーバーへ接続するStubと、管理コンソールからなる監視システムを構築しています。Stubはそのアプリケーションへ接続し、トランザクションの追跡情報を管理コンソールに送ります(これは何らかのIPCメカニズムを利用します)。このコンソールアプリケーションを実行しているユーザーは、セッションを開始することができます。セッションが開始されると、クライアントのリクエストの種類、到着時刻と出発(終了)時刻を監視します。それらの情報はコンソールに転送され、コンソールは情報を受け取り、処理をして、データベースに書き込みます。

¹ db4oでは、データベースはObjectContainerとして参照されます。ObjectContainerは特定のファイルに関連付けられています。ObjectContainerを開くには以下のように行います。

```
ObjectContainer db = Db4o.openFile("<path>");
```

where <path> is the path to the database file. When all operations on the database are completed, you call:

```
db.close();
```

この先取り上げるコードでは、ObjectContainerのインスタンスであるdbは既に開いているものとします。

その後、管理者はデータベースに問い合わせをし、トレンドを見たり、レポートやグラフを生成したりします。レポートやグラフは、ある日の時間帯ごとのトランザクション要求数や、毎秒のトランザクション数に対する平均処理時間などです。これらの情報を元にして、この組織はクライアントからのトランザクション要求を処理するために、十分な能力がこのシステムが持っているかどうかを判断することができます。

実装

この仮想アプリケーションの開発者は、2つのクラスを利用してセッションをモデル化しました。1つは、Sessionクラスで、グローバルなセッション情報である、名前やセッションの開始時刻やセッションの終了時刻と、トランザクションの情報のリストオブジェクトを含んでいます。

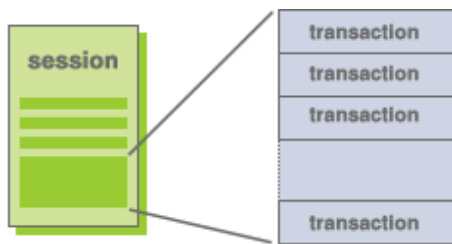


Figure 1. セッションをモデル化する単純なオブジェクト構造。セッションオブジェクトはトランザクションオブジェクトのリストを含んでいます。

セッションクラスは以下ようになります。

```
// Session
public class Session
{ private String sessionName; // Unique name for the session
  private timestamp start; // Start of reading
  private timestamp finish; // End of reading
  private List transactions; // Transactions
  // Constructor
  Session (String sessionName,
           timestamp start)
  {
    this.sessionName = sessionName;
    this.start = start;
    this.transactions = new ArrayList();
  }
  // Add a transaction
  public void addTransaction(Transaction trans)
  {
    transactions.add(trans);
  }
  ... other methods for Session ...
}
```

そしてトランザクションクラスは、Stubから送られてくる情報をモデル化しています。

```
// Transaction objects
public class Transaction
{ private int type; // Transaction type
  private int quantity; // # of bytes of data exchanged
  private timestamp start; // Request arrival
  private timestamp finish; // Response transmission
  ... remainder of Transaction class ...
}
```

² 主に、どのようにしオブジェクトのデータメンバーを永続化したらよいかに関心があり、クラスのメソッドではないので、上記の

セッションが開始される時、セッションオブジェクトがインスタンス化され、ユニークな名前 (sessionName) が与えられます。それから、コンソールはStubにトランザクションデータを収集するように連絡します。Stubから新しいパケットが到着する度に、コンソールは新しいトランザクションオブジェクトをインスタンス化し、そのオブジェクトにデータをセットし、セッションオブジェクトのトランザクションArrayListにオブジェクトを格納します。

```
theSession.addTransaction(theTransaction);3
```

セッションが完了するとき、セッションオブジェクトをdb4oに格納するのは簡単であるはずがありません。しかしここで、プログラマの作業を楽にしまうと、db4oに搭載されたインテリジェント機能をまず知るようになります。たった一行のコードで、オブジェクト全体、つまりListとその全てを格納してしまいます。

```
db.set(theSession);
```

db4oはオブジェクトツリーを自動で調べ、セッションオブジェクトだけでなく、トランザクションArrayListの全てのメンバーを探し出して格納します。

セッションオブジェクトを取得するのも同様に簡単です。db4oは、“query-by-example”パラダイムによる、分かりやすいクエリ機能を持っています。クエリを実行するには、検索したいオブジェクトと同じクラスを使ってテンプレートを作成します。そして条件でフィールドを埋めてから、ObjectContainerのget()メソッドを実行します。

例えば、“FridayCapture:”というセッション名のオブジェクトを取得したいとすると、

```
Session proto = new Session("FridayCapture",
    (timestamp)0);
ObjectSet result = db.get(proto);4
```

クエリの結果はObjectSetです。それを使って条件に合ったオブジェクトを順に取得することができます。上記の例では、結果オブジェクトは1つなので、結果セットから直接対象オブジェクトを取得できます。

```
Session theSession = (Session)result.next();
db.activate(theSession, 2);
```

2行目は、db4oにセッションオブジェクトを2階層目まで’activate’させる指示です。db4oがオブジェクトをactivateする時、データベースからフィールドをロードします。つまり上記の例では、セッションオブジェクトだけではなく、子オブジェクトであるArrayListの中身もロードするように指示しています。3階層目までをロードするように指示すると、子オブジェクトとさらに孫オブジェクトまで、4階層目は子オブジェクト、孫オブジェクト、ひ孫オブジェクトといったようになります。⁵

クラスは省略されています。また、トランザクションタイプのint型だけでなくtimestamp型が利用可能であると仮定しています。

³ SessionオブジェクトであるtheSessionとTransactionオブジェクトであるtheTransaction

⁴ 空かゼロのフィールドはクエリに適用されません

⁵ db4oはデータベース内の全クラスや特定のクラスの全オブジェクトに対し、activateの深さを設定することができます。

ざっと見てきましたが、このように単純です。一回の呼び出しで、全てのオブジェクト構造が格納されます。そして全てのオブジェクト構造が、単純で分かりやすいクエリAPIで取得可能です。

以上のことからわかるのdb4oの特徴は、

*db4oでオブジェクト構造を扱える
あたかもメモリ内にオブジェクト構造があるように
ごくわずかのコードでオブジェクトを永続化することが可能*

これまでの応用

これまでに見てきたサンプルは、分かりやすく機能的でした。しかし、理想的ではありません。なぜならセッションオブジェクトは、全部メモリ内にあるからです。それは結局、メモリ容量という、上限を持っています。ひょっとしたらメモリ容量が十分な場合もあり得ますが、ここではそうではないと仮定して進めて行きます。

Appendix A では、メモリリークを解決するとき、それはどんなデータベースを利用しても存在していますが、db4oは障害とはならなかった、つまり問題解決にぴったり合うことを見せします。具体的には、単純なクエリメカニズムが、問題解決を容易にしました。

削除する

このセクションを終了する前に、まだご紹介していない操作について、触れておかなければなりません。ここでdb4oのもう1つのパワフルな機能をご紹介します。

それでは全てのセッションに関するオブジェクトを削除する必要があるとしましょう。オブジェクト単体を削除するのは簡単です。格納されていたSessionオブジェクトがtheSession変数にあるとすると、次のようにデータベースからオブジェクトを削除することができます。

```
db.delete(theSession);
```

しかし、任意の長さのリンクリストを削除するには、やっかいなループ操作が必要です。なぜなら永続化されたオブジェクトを削除するには、まずオブジェクトを取得し、ループは以下のようになります。

- 1) 先頭のTransactionオブジェクトを取得
- 2) 次のリンクを保存
- 3) Transactionオブジェクトを削除
- 4) 次のTransactionオブジェクトを取得
- 5) 2)に戻り、次のリンクが無くなるまで続けます

機能させることはできますが、実装したくなくなるほど優雅さを欠いています。

幸運にも、db4oを使えば、この”cascaded delete(並列削除)”を優雅に実装できます。データベースを開く前に、ある特定のクラスを”cascaded delete”にして欲しいとdb4oに指示するだけです。サンプルは下記のようになります。

```
Configuration config = Db4o.configure();
ObjectClass sessionObjectClass =
    config.objectClass("<package>.Session");
sessionObjectClass.cascadeOnDelete(true);6
```

一旦上記のコードを実行し、データベースを開いてから削除したいセッションオブジェクトをデータベースから取得すると、そのセッションオブジェクトに関するオブジェクトを削除することができます。つまりSessionオブジェクトと、Transactionオブジェクトのリストを一発で削除できるのです。

```
db.delete(theSession);
```

db4oはオブジェクトの構造を全て調べ、全ての子オブジェクト、孫オブジェクト、ひ孫オブジェクトなどを削除します。この場合は、リンクリストを辿ってどんどん進み、全ての連なるトランザクションオブジェクトを削除します。これらはまるで一回ボタンを押すだけのように行われるのです。⁷

リレーショナルデータベースとの比較

さて、ここで一旦話題を変えて、リレーショナルデータベースを利用するとしたらと考えて見ましょう。このセクションで取り上げたサンプルを見て、どのようにしなければならないか考えてみましょう。どこかでテーブルの構造を定義しなければならないでしょう。セッションテーブルとトランザクションテーブルを作るとしましょう。主キーのカラムを定義し、一対多のリレーションを定義します。1つのセッションに複数のトランザクションです。もちろんトランザクションテーブルには、関連するセッション行を識別するIDの外部キーのカラムが必要なことは言うまでもありません。INSERT、UPDATE、そしてDELETEといったSQLコマンドがあって、オブジェクトのメンバーを各テーブルのカラムにセットしたり、テーブルからオブジェクトにセットしたり、SQL分のパラメータにセットすることも必要でしょう。セッションオブジェクト構造全体を取得するには結合構文が必要でしょう。

簡単に言うと、かなりの量のコードが必要です。オブジェクト指向パラダイムとリレーショナルデータベースパラダイムの見えない壁を越えてデータをやり取りするために…

…db4oなら、もうそれは不要です。

⁶ <package>はSessionクラスのパッケージを表しています

⁷ 並列削除を使うと驚くほどプログラミングが楽になります。しかし強力なので間違えると大変ですから注意してください。

それは必要？

この時点で、私たちがモグラの穴の山から、でっかい山を作っているのではないかと疑っている人もいるかもしれません。Transactionデータは順番で来るので、順番に格納されているのだから、シーケンシャルファイルを作って、順番にTransactionデータを書き込めばいいのではないのでしょうか？なぜわざわざデータベースを使うのでしょうか？

条件によっては、それが最善の方法にこともあるでしょう。しかし将来的に、データマイニングを行うためのリポジトリを作るなら、データベースに搭載された検索機能を利用することが出来ます。さもなければ、シーケンシャルファイルに保存されたデータを取り出すものを自分で書かなければなりません。

このセクションのサンプルを使って、すべてのセッションに対する、ある特定のトランザクションで転送される平均バイトデータ量を調べるとしましょう。もしシーケンシャルファイルを使うなら、情報を集めるためにすべてのファイルの中で条件に合うトランザクションオブジェクトを取り出すコードを書かなければならなかったでしょう。

一方で、db4oは単純なクエリで必要なトランザクションを見つけてくれます。例えば、トランザクションタイプが一定の、“READ_TRANSACTION”というものだとすると、コードは以下ようになります。

```
Transaction theTrans;
double totalbytes = (double)0.0;
int totaltrans = 0;
double average;
...
Transaction proto = new Transaction();
proto.setType(READ_TRANSACTION);
ObjectSet result = db.get(proto);
while(result.hasNext())
{
    theTrans = (Transaction)result.next();
    totalbytes + (double)theTrans.getQuantity();
    totaltrans++;
}
if(totaltrans!=0)
    average = totalbytes/(double)totaltrans;
...
```

Transactionオブジェクトがそのまま格納されているので、このようにできます。それゆえ、Sessionオブジェクトに関わらず、db4oのクエリメカニズムを利用してアクセスすることが可能です。

これはあきらかに、すべてのSessionの中からTransactionを探し出すために書かなければならないコードと比べると、はるかに容易です。もちろんSessionオブジェクトが追加されてもコードの変更は不要です。

背の高いツリー構造

オブジェクト指向言語の美点のひとつに、複雑でやっかいな実世界のリレーションを、構造化言語を使わざるをえない場合のやり方に比べ(メモリポインターエラーの温床になる)、もっと自然なやり方でモデル化できることが挙げられます。そして中でもやっかいなのが、何らかのツリー構造で表現されるものです。

ここで“背の高いツリー構造”というのは、前述のオブジェクトとはその広がりにおいて異なるオブジェクト構造のことを指しています。前述のオブジェクトには、深みはありませんが、広がりがあります。これから検討するのは、深くて、枝分かれがあるものです。つまりルートオブジェクトが複数のオブジェクトを参照し、それぞれの子オブジェクトは、各自が多数の子オブジェクトを参照しています。そしてそのような構造が任意の深さまで続いています。

例2: 発電機用部材データベース

発電機を製造している会社があるとします。そして部材の明細データベースを作成したいと考えています。そのようなデータベースでは、それぞれの製品は、組立て品(assembly)のコレクションとサブ組立て品(sub-assemblies)を指しています。それらは、さらに他のサブ組立て品を指しています。さらに、組立て品とサブ組立て品は、部品(component parts)リストを指しています。うまい具合に、製品の全体構造は、完全にオブジェクト構造を辿ることで、製品を作り上げている関連し合う全ての構成要素が分かるように、組立て品、サブ組立て品、そして部品の構造を映し出します。

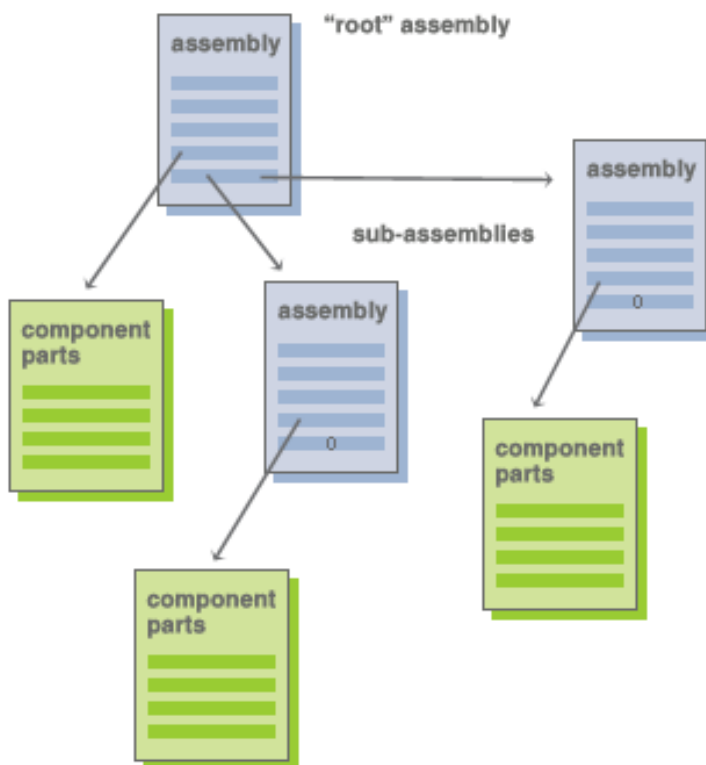


Figure 2. 完成品の構成要素を表すオブジェクト構造。大本の組立て品は、1つ以上のサブ組立て品からできているだけでなく、部品リストからもできています。サブ組立て品はそれ自身が組立て品でもあるので、さらに部品リストと、サブ組立て品から構成されています。

これはオブジェクトデータベースに特に適したデータ構造です。データベース内の製品は自然にツリーとして表されています。従って、この会社の開発者は、Assembly, ComponentParts, RawPartからなるトライアングル構造を定義しています。

Assemblyは任意の数のsubAssembliesと任意の数のcomponentPartsからできています。従ってsubAssembliesとcomponentPartsは配列としてモデル化されています。部品(a component part)は、実際の物質的なアイテム、例えばねじやボルト、ファスナーなどを表しています。この後分かりますが、componentPartsはそれぞれの数量フィールドも持っています。もし組立て品(a assembly)が部品、例えば15本の2番ねじ、を含んでいる場合は、componentParts配列の中では15ではなく1つの要素として扱われます。さらに、構造は再帰構造です。サブ組立て品(a subassembly)はそれ自体が組立て品で、それ故にcomponentPartsと0以上のsubAssembliesから構成されています。

componentParts配列内のそれぞれの要素は、RawPartアイテムを参照しています。それは特定の部材を特定する情報です。それを発注した相手の会社や、最小在庫管理単位、それからコストです。

従って製品の説明は、ツリーになります。トップのルートは製品自体です。ツリーを辿っていくと、最終的には“葉”にたどり着きます。それはcomponentPartsのみからなる、サブ組立て品です。

```
public class Assembly
{
    public String name;        // Name or description
    public Assembly subAssemblies[];
    public ComponentParts componentParts[];
    ... remainder Assembly class ...
}
public class ComponentParts
{
    public int productID;     // RawPart product ID
    public int quantity;
    ...remainder of ComponentParts class...
}
public class RawPart
{
    public int productID;     // RawPart product ID
    public string SKU;        // Raw part's SKU
    public List suppliers;    // List of suppliers
    public BigDecimal cost;   // Cost
    ...remainder of RawPart class...
}
```

この時点で、もしずらりと子オブジェクトを持つ新しいアイテムを追加するのに、複雑なコードを考えているとしたら、それは不要です。db4oは前のセクションで見たようなオブジェクトと同様に簡単に処理することができます。theAssemblyというAssemblyオブジェクトがあったとして、それを作成し、全てのサブ組立て品とcomponentPartsをつなげたら、theAssemblyを次のように保存することができます。

```
db.set(theAssembly);
```

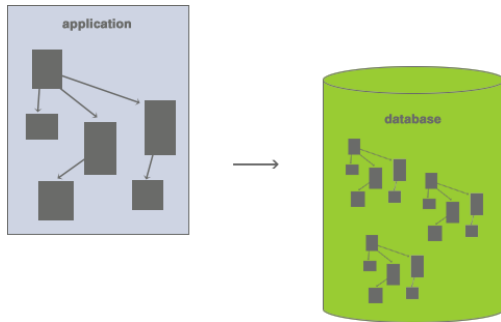


Figure 3. db4oを使えば、複雑なオブジェクト構造でも単純なオブジェクト同様に格納することができます。1行セットが鍵です。

そして前述したように、もし”cascaded delete”をオンにしてAssemblyオブジェクトを、関連するサブ部品とcomponentPartsと一緒に削除したいなら、次のようにします。

```
db.delete(theAssembly);
```

しかしながら注意していただきたいのは、ComponentPartsとRawPartを関連付けるために、データベースにある外部キーという概念を借りています。つまりproductIDフィールドで、両者を結びつけています。従って、例えば特定の部品のコストをもし取得したい時は、次のようなコードになります。

```
int prodID = theAssembly.componentParts[<index>];
RawPart rawPartProto = new RawPart; // Build prototype for query
rawPartProto.productID = prodID;    // Field to match
ObjectSet result = db.get(rawPartProto);
theRawPart = (RawPart) result.next();
```

ComponentPartsとRawPartオブジェクトのリレーションはオブジェクト参照では無いので、Assemblyを”cascaded delete”にしても、一緒に削除されることはありません。

さて、ここで前述の特徴に戻ってみましょう。

*db4oでオブジェクト構造を扱える
あたかもメモリ内にオブジェクト構造があるように
ごくわずかのコードでオブジェクトを永続化することが可能*

これまで見てきたように、深いツリー構造を操作するのは、フラット構造である配列やリストを操作するのと同じように簡単です。従ってもう1つ特徴を追加します。

*永続化要求が複雑になる時は、
db4oのユニークなデザインが増加した複雑さの面倒を見ます
ですから気にせず作業を続けてください
何も無かったかのように*

進化するオブジェクト構造

“おれが複雑にしたんじゃない。自然とそうなるものなんだよ。”

Martin Riggs, Lethal Weapon

この有名なセリフが意味するところは、「好きでも嫌いでも、物事は変化して、複雑な方へ行きやすい」ということです。期間と仕様が修正されたり、捻じ曲げられたり、時にはがらりと変わってしまったプロジェクトで苦い経験をしたことがあるプログラマなら、納得してうなづくことでしょう。データ構造が変化する場合は、それに伴って発生するコードの変更はやっかいで、エラーの温床となります。特に大変なのは、データベースのデータを新しいものに移行しなくてはならない時です。

しかしdb4oなら変更は簡単です。それでは最初の例に戻って見てみましょう。

例1進化版: ウェブアプリケーションのトランザクションを監視する

さて、時は流れ、組織は大きくなり、様々な場所でサーバーが動いています。技術者はこれまで、複数のサーバーから同時にデータを収集できるようにシステムを拡張できないか尋ねられたことがありました。

そこでクラスの定義を以下のように変えました。

```
// Session objects
public class Session
{ private String sessionName; // Unique name for the session
  private List hosts;        // List of hosts involved in the reading
  private timestamp start;   // Start of reading
  private timestamp finish;  // End of reading
  private Transaction head;  // Reference to head of list
  private int length;        // Number of items in the list
  ... remainder of Session class ...
}
// Transaction objects
public class Transaction
{ private String host;        // Origin of transaction
  private int type;           // Transaction type
  private int quantity;      // # of bytes of data exchanged
  private timestamp start;
  private timestamp finish;
  private Transaction next;  // Reference to next transaction
  ... remainder of Transaction class ...
}
```

これらの新しいバージョンでは、Sessionオブジェクトが、データを収集した先のホストリストを持っています。さらに、各TransactionはホストIDを含んでいます。これにより1つのSessionが複数のホストからのTransactionデータを取得できます。⁸

⁸ ホストを特定するためにstring型を利用しています。これは例を分かりやすくするためです。実際のアプリケーションだとしたら、容量を少しでも小さくするためにバイトのフィールドを使い、1から255のIDを割り当ててでしょう。

さて今度は、セッションオブジェクトが開始すると、監視するホスト名がホストリストに格納されます。そしてトランザクションが到着すると、リンクリストに追加される前に、どのホストから来たかによってタグが付けられてからデータベースに格納されます。最初のアプリケーションの例でお見せした、データが格納される方法やどのようにリンクリストがデータベースに書き込まれるのかは、大きく変化していません。

変更されたスキーマ

でもちょっと待ってください。何かに変更されたはずですよ。そうですね。クラス構造です。すでにデータベースに格納されているデータはどうなるのでしょうか？アプリケーションサーバー1台の時に集めたデータはなくなってしまうのでしょうか？アプリケーションが新しいセッションオブジェクトとトランザクションオブジェクトを元のデータベースに書き込み始めたとしましょう。古いデータが新しいクラスでインスタンス化されるとアプリケーションはクラッシュするのでしょうか？新しいデータベースを作成し、古いデータをコピーして新しいデータに変換して書き込む必要があるのでしょうか？

幸いにも、その必要はありません。db4oは進化したクラスをスムーズに同化しています。唯一必要なのは、新しいメソッドをクラスに追加することです。Sessionクラスに以下のメソッドを追加します。

```
public void objectOnActivate(ObjectContainer container)
{
    if(hosts == null)
    {
        hosts = new ArrayList();
        hosts.add("MainServer");
    }
}
```

Transactionクラスには次のメソッドを追加します。

```
public void objectOnActivate(ObjectContainer container)
{
    if(host == null)
        host = "MainServer";
}
```

以上です。手を払っていただいて結構です。変換作業は終わりました。

db4oがオブジェクトをアクティベートする時、クラス定義からオブジェクトをインスタンス化する必要があります。そしてオブジェクトをインスタンス化する時、db4oは、上記の例にあるようなobjectOnActivate()メソッドがあるかチェックします。もしある場合は、アクティベーションの最後にこのメソッドを呼び出します。このようにしてインスタンス化されたオブジェクトの中身を調べることができ、アプリケーションで使われる前にフィールドの値を埋めてオブジェクトを修正することができます。

ですから、古いオブジェクトを新しいオブジェクトのようにすることができるのです。アプリケーションは変更があったことに気づきませんし、気づく必要も無いでしょう。アプリケーションにとっては、データベースから取得された全てのオブジェクトが新しいオブジェクトでしょう。実際には、アプリケーションが一度アクティベートしたら、古いオブジェクトは新しいオブジェクトになっています。もしその新しいオブジェクトをデータベースに書き込むなら、新しいバージョンが古いバージョンを上書きします。

この例では、デフォルトのアプリケーションサーバー名を、“MainServer”と仮定しています。ですから全ての古いSessionオブジェクトとTransactionオブジェクトは、全て1つの“MainServer”から来ているように見えています。

最後に、db4oがオブジェクト構造、実際はオブジェクトスキーマ、の変更に対し、アプリケーションコードを一切必要とせずに吸収して対処していることに注意してください。変更に伴って必要になったのはたった2つの小さなメソッドです。しかも必要な時だけしか呼ばれないもので、db4oが呼ぶので気にする必要さえありません。

もう一度、リレーショナルデータベースと比較

さて、もう一度考えて見ましょう。もしデータがリレーショナルデータベースにあった場合に何をしなければならぬか。次のどちらかをやらなければならなかったでしょう。

- 1) 2つ新しいテーブルを作成します。1つは古いデータ、もう1つは新しいデータを入れます。次々にアクセスされる新しいデータと古いデータを処理する非常につまらないコードを書かなければならぬでしょう。
- 2) 新しいデータベースを作り、新しいスキーマを定義し、古いデータを新しいデータに変換するコードを書き、もう二度と変更が無いようにとお願いする。

実際には、ネイティブでないOODBMSであっても、db4oのような軽いフットワークでオブジェクト構造の変更に対応はできないでしょう。たいていは古いものと新しいものが混在して混乱するでしょう。何らかの変換アプリケーションがたいていは必要になります。

ここで前述した2番目の特徴を振り返って見ましょう。

*永続化要求が複雑になる時は、
db4oのユニークなデザインが増加した複雑さの面倒を見ます
ですから気にせず作業を続けてください
何も無かったかのように*

確かに、db4oはこれ以上ではないかと言いたくなります。例え、全く分からないところでオブジェクト構造が変化してしまったとしても、うまく続けられるように、db4oがそれらの変更に対応してくれるのです。db4oを使えば、メンテナンスが容易なAPIを用いて、そういった変更にきちんと対応できるようになります。従って、変換作業に費やす無駄な時間が無くなり、価値のある作業に専念することができるようになるのです。

導入事例 製品の市場投入を10%速く

これまでには主に、db4oの単純さに着目してきました。それによって最高に複雑なオブジェクトでさえ簡単に永続化処理を行うことができるようになります。しかし、それがどのように新製品の製品化に要する時間に反映されるのでしょうか？

BOSCH Technology Groupは、世界最大のパッケージング技術会社です。BOSCH Siggpack Systems社のエンジニアは、製品化に要する時間を最重要課題だと考えています。そんな彼らが選んだのは、db4oでした。db4oは、“ピックアンドブレース”ロボットパッケージングシステムで、高速で信頼性のあるDelta XR31をコントロールするアプリケーションの永続化エンジンとして利用されています。



そのアプリケーションでは、ロボットシステムを、それぞれがフィーダーオブジェクトの配列に接続されている3000のオブジェクトでモデル化しています。そしてそれぞれのフィーダーオブジェクトは、センサーオブジェクトの配列にリンクしています。合わせると、39000のオブジェクトが同時にメモリ内、データベース内で操作されています。ロボットの製品ラインを特定の商品組立てにするとときはいつも、設定が必要です。別な言い方をすると、全てのオブジェクトを素早く、しかも正確に再設定しなければなりません。製造ラインが停止している時間は全て、時間の無駄になります。

Sebastian Hubrich, BOSCH Siggpack Systems社のチーフエンジニアは、db4oが複雑なオブジェクト構造を容易に扱えることで、どれ位製品化に要する時間を短くするか定量化しました。

“バックエンドにdb4oを使って助かりました

各製品で少なくとも10%は製造時間の節約効果があります

このプロジェクトは非常に需要がありますが時間的余裕はありません

db4oはこのような条件下で最適な選択肢でした”

まとめ

このホワイトペーパーを通して、オブジェクトの永続化処理が、db4oを使えば非常に簡単にできるということを示すことができましたと思います。そしてどんなに複雑な構造のデータを永続化しても変わらないということ。

同時に、時間的、スペース的に複雑なオブジェクトにも、それはつまりフラットや背の高いツリー構造だけでなく進化する構造でも、db4oを使えば臆することはないことをご紹介できたと思います。さらに、次にJavaまたは.NETアプリケーションを開発するときには、JDBCやODBC、SQLやOQLなどに時間を費やすことが不要であるということをお見せしたこともなります。オブジェクト指向の学習には、すでに時間をかけて投資をしてきたと思いますが、db4oならその投資をさらに有効活用することができます。

そしてご紹介したデータベーステクノロジーは、未熟だったり、二流品ではありません。RDBMSを使ったら何行にもなることを、db4oはたったの1行でやってのけることをお見せしました。そしてネイティブでないOODBMSでも、何らかの追加コードの実装か、スキーマファイル等の学習が必要ですが、幸いにもdb4oにはそれさえ不要です。

一言で言えば、db4oは複雑なオブジェクト構造のタスクを、非複雑化することができるのです。

Appendix A – db4oのメモリリークへの対処法

“フラットオブジェクト構造”の中で取り上げた“ウェブアプリケーションを監視する”の例は、分かりやすく機能的ですが、理想的ではありません。なぜならすべてのデータがメモリ内にあるので、セッションオブジェクトのサイズに上限を設けることとなります。それはもちろんメモリの容量で制限されることとなります。現実的であり得ることでありますが、例ではあえて問題ないものとして進めました。

そこで、開発者が任意の大きさのセッションオブジェクトを取り扱わなければならないと決めたことにしましょう。そこで、トランザクションデータを単一のリンクリストで管理できるようにクラスを以下のように変更します。

```
public class Session
{ private String sessionName; // Unique name for the reading
  private timestamp start;    // Start of reading
  private timestamp finish;   // End of reading
  private Transaction head;   // Reference to head of list
  private int length;        // Number of items in the list
  ...
  public void incrementLength()
  { length++; }

  public int getLength()
  { return(length); }

  public void setFinish(timestamp theTime)
  { finish = theTime; }

  public void setHead(Transaction theHead)
  { head = theHead; }
  ... remainder of Session class ...
}

public class Transaction
{ private int type;          // Transaction type
  private int quantity;     // # of bytes of data exchanged
  private timestamp start;  // Request arrival
  private timestamp finish; // Response transmission
  private Transaction next; // Reference to next transaction
  ... remainder of Transaction class ...
}
```

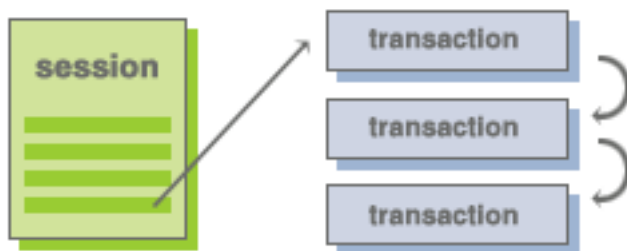


Figure 4. さらにやっかいな構造。Sessionオブジェクトは、Transactionオブジェクトへの参照を持つ単体のリンクリストを持っています。

もちろんこうした目的は、メモリ消費を減らして、トランザクションオブジェクトをデータベースに格納することです。トランザクションデータが到着すると、リストに追加され、データベースに書き込まれます。各Transactionオブジェクトは次のTransactionオブジェクトへの参照を持っていて、最後尾のオブジェクトは最後を表すnull参照を持っています。セッションが終了すると、Sessionオブジェクトに終了時刻が刻まれ、Sessionオブジェクトはデータベースに格納されます。興味深いコードを次に示します。

```
// Globals
Transaction tail;
Transaction newTransaction;
Session theSession;
...
// This code initializes the session
theSession = new Session("<Session Name>",
new timestamp(System.currentTimeMillis()));
tail = null;
...
// This code is executed when new transaction data arrives
newTransaction = new Transaction(<fill in from incoming data>);
if (theSession.getLength() == 0)
    Session.setHead(newTransaction);
else
{
    tail.next = newTransaction;
    db.set(tail); // Persist the Transaction object
}
tail=newTransaction;
theSession.incrementLength();
...
// This code is executed when the session is finished
theSession.setFinish(new timestamp(System.currentTimeMillis()));
if(tail != null) db.set(tail); // Persist last Transaction
db.set(theSession);
...
```

この新しいコードで興味深いのは、開発者がインストールして実行すると・・・

・・・うまくいきません。全てのデータはまだメモリ内にあります。

鋭いプログラマはお気づきだと思いますが、これは典型的なJavaのメモリリークです。なぜならtheSessionオブジェクトはリストの先頭のオブジェクトへの参照を持っているので、単純にリストに追加してもその文メモリを消費するだけです。たとえリストの要素が永続化されても同じです。先頭への参照がメモリ内にある限り、全体がメモリ内に残ってしまいます。

幸運にも、簡単な解決方法があります。これはリークを解決するだけでなく、ディスクに書き込まれるアイテムの“ペース”をチューニングすることもできます。修正するには、

- 1) Sessionオブジェクトをディスクへ書き込みます。この時、参照を削除します。
- 2) 到着したトランザクションオブジェクトをキャッシュします。キャッシュがいっぱいになったら、データベースへの書き込みをまとめて行います。
- 3) Sessionオブジェクトを最後に読み込み、長さで終了フィールドを修正します。

修正したコードは以下のようになります。

```

// Globals
// Cache
Transaction transactionCache[];
int transactionCacheNext;
int numTransactions;
boolean sessionWritten; // True if the session has been written
Session theSession;
String theSessionName;
...
// Initialization
// This code executes at the very start of a Session,
// before any data has arrived.
// (It assumes that theSession has already been
// created and its description and start fields initialized.)
sessionWritten=false;
numTransactions=0;
theSessionName = theSession.getSessionName();
dbTransCacheInit(CACHE_SIZE);
...
// This method initializes the cache to size entries
public void dbTransCacheInit(int size)
{ transactionCache = new Transaction[size];
  transactionCacheNext = 0;
}
...
// This method is a "cached set()". Whenever a transaction
// comes in, the console application calls this routine to
// write the transaction to the database (in place of a .set()).
// Transactions are first placed in the cache. When the
// cache fills, its contents are written to disk (see the
// following method dbCacheFlush())
// Note that the first time the cache is written, the Session
// object is also written. This cuts off the head of the list,
// so the entire list is not kept in memory.
public void dbCachedSet(Transaction obj)
{
  // If the cache is full flush it
  if((transactionCacheNext+1)==transactionCache.length)
    dbCacheFlush();
  // Put the new object in the cache
  transactionCache[transactionCachNext++]=obj;
}
public void dbCacheFlush()
{
  // If the session object has not yet been written,
  // write it
  if(sessionWritten==false)
  { db.set(theSession);
    theSession=null; // Cut off the head
    sessionWritten=true;
  }
  // Flush whatever is still in the cache
  if(transactionCacheNext>0)
  for(int i=0; i<transactionCacheNext; i++)
  { db.set(transactionCache[i];
    numTransactions++;
    transactionCache[i]=null;
  }
  transactionCacheNext=0;
}
...

```

```
// This code is executed when the Session has
// completed.
// Set finish time
finishTime = new timestamp(currentTimeMillis());
// Flush the cache one last time
dbCacheFlush();
// Fetch the session object from the database to
// fix it up.
Session proto = new Session(theSessionName,0);
ObjectSet result = db.get(proto);
theSession = (Session) result.next();
theSession.setFinish(finishTime);
theSession.setLength(numTransactions);
// Write the updated session
db.set(theSession);
...
```

この新しいコードは、永続化されたリンクリスト構造を使っています。それによって任意のサイズに大きくなることができます。さらに、キャッシュのサイズは設定可能ですので、開発者は必要に応じてメモリ消費量を調整することができます。

ここでお見せしたことに注意して下さい。メモリリーク問題を解決しました。どんな永続化エンジンをバックエンドに利用しようとも存在していたものが、最小の努力で解決されています。もう一度言いますが、db4oは障害とならずに、問題解決にスムーズにフィットしました。実際には、単純なクエリメカニズムが、この問題解決をより簡単にしています。