

db4o | オープンソースオブジェクトデータベース | Java and .NET

ブレーンを支えるデータベース

By Rick Grehan

モバイル機器、情報端末、インテリジェントコントロールシステム。コンピュータアプリケーションの広大な分野を知るために、それほど込み入った調査は必要ありません。これらのシステムは通常、“組み込みシステム”と呼ばれますが、それらのアプリケーションが複雑化していることを表してはいません。

CPUのクロック数は上がり、メモリ密度は大きくなり、ハードディスクのデータは圧縮され、価格は下がり、これらのシステムに要求されるタスクはますます複雑になっています。組み込みシステムはこれまで以上に、スマートになることが要求されています。別の言い方で言うと、より多くのデータを格納、取得、操作すること、しかもユーザーが期待するCPUとネットワーク帯域の成長速度に見合うような高速性能が要求されています。

見方を変えると、インテリジェントデバイスがインテリジェントであるのは、そのアルゴリズムだけでなく、それらのアルゴリズムで使うデータのおかげでもあります。つまりデバイスが精巧になるにつれ、それに伴って精密なデータ管理、格納、取得をすることができるソフトウェアが要求されることになります。

何が必要か

“何が必要か”に対する答えは分かっています。一言で言うと、「小さくて速く、なおかつパワフルで使いやすいデータベースライブラリ」です。

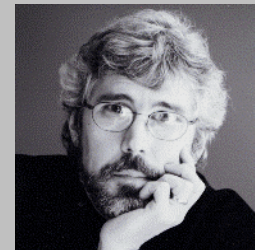
それらが意味するところは明らかですが、詳細に検討してみましょう。

・**最小リソース消費**: 技術的な進歩のおかげで、システムにメモリが不足したら追加すれば済みますが、アプリケーションのメモリを最適化するというは常に必要なことです。トレードオフは単純です。データベースライブラリのメモリ消費量が少なければ、その分アプリケーションがメモリを多く使えるので、設計者はより多くの機能を追加することができます(またはスピードアップ)。上記で“最適化”という用語を使ったことに注意してください。単純にデータベースライブラリのサイズを減らすということではありません。それなら機能を削減すればすぐに実現できます。性能をバイトと交換するのは危険です。小さな製品が出来上がりますが、利用価値の低いものとなるでしょう。

・**高スループット**: これは自明です。データベースアクセスが遅くてもいいのは非常に限られたアプリケーションだけです。違う言い方をすれば、データベースのレスポンスが非常に速いからと言って、不平を言う人はいません。

・**実装が容易**: 開発環境をどんなに改善しても、オブジェクト指向プログラミングが優れていても、山ほどアルゴリズムやデータ構造のライブラリがフレームワークに用意されていても、開発者は、ビジネスロジックをデザインし、コードを書くという面倒な仕事をしなければなりません。一方データベースライブラリのAPIには、このような学習の壁がありません。よくできた表現を借りると、“より簡単にではなく、できるだけ簡単に”ということです。これによって学習が容易になり、データベースライブラリ自体を勉強するのではなく、アプリケーションにより時間を割くことができるようになります。

さらに言うと、データベースライブラリを開発プロジェクトに統合するのはワンステップでやれるべきで



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、InfoWorld誌の執筆も行っていて、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が専門です。

かつてRick氏は雑誌BYTEのSenior Editorをしていました。そこではLab Directorであり、JavaTalkコラムを執筆していました。その後Metrowerks社でDiscoverer DPSプロジェクトに参加した後、現職に就任しています。

す。理想的には、ライブラリは単体のファイル、JavaではJARファイル、.NETではDLLでしょう。スクリプトベースのプロジェクトでは、スクリプトに1行追加するだけ、IDE内のプロジェクトではドラッグアンドドロップするだけでライブラリを追加することができるのが理想です。

・**ポータブル**: データベースライブラリは、実行環境を最大化するため、プラットフォームを問わないことが求められます。それによってアプリケーションの潜在顧客を最大化します。また、プログラマは主要なOSで開発ができるようになるので、最先端のツールを利用することができ、OSに制限されない結果を保証することができます。

・**信頼性**: 信頼性はもう一つの自明な必要事項です。おそらく最重要でしょう。信頼性の無いライブラリは、単純に無用です。たいいていの組込みアプリケーションにとって、特にリアルタイムシステムでは、信頼性は全てのコンポーネントにとって議論の余地の無い条件です。さらに、データベースは業界標準の基準をクリアしなければなりません。特にデータベースライブラリは、ACIDを満たさなければなりません(右記ACID参照)。

リレーショナルデータベースの可能性

リレーショナルデータベースは、確かに最もよく知られていて、恐らく最もよく使われているデータベースパラダイムです。有名であるがゆえに、どんなアプリケーションに対してもまず選択肢としてあがるでしょう。しかしながら、数多くのメリットにもかかわらず、リレーショナルデータベースをオブジェクト指向アプリケーションに組み込むのはスムーズではありません。この事を指して、評論家はよく“インピーダンスミスマッチ”と呼びます。この電気用語を借りて、リレーショナルモデルとオブジェクトモデルの不一致を表しています。このミスマッチは、リレーショナルデータベース(RDBMS)が、情報をテーブルの中の行に格納するために発生します。それ故に、オブジェクトを格納するには、オブジェクトを分解し、その断片をテーブル内のフィールドにはめていくことが必要です。そしてオブジェクトを取得するには、断片を集め、組み立て直さなければなりません。

例 (Java): インテリジェント自動販売機

このインピーダンスミスマッチがいかに面倒くさいか、簡単な例でお見せしましょう。

例えば、インテリジェントな自動販売機を開発するとしましょう。商品の種類とその数量を記録し、お釣りが利用できるかモニターします。さらに売り上げを記録し、そして無線によってインターネットに接続し、商品やつり銭の不足をEメールで連絡するとしましょう。アメリカなどでは信じ難いかもしれませんが、日本などではかなり現実的な話です。

ACID

ACIDとは、データベースが有用であると考えられるために、必ず満たさなければならない4つの条件の頭文字をとったものです。

Atomicity - データベースのトランザクションに参加するコンポーネントは、“all-or-nothing”で実行しなければなりません。例えば、あるトランザクションが4つのオブジェクトを削除するとしたら、それらは1つのオブジェクトであるように削除されなければなりません。3つは削除されたが、1つは残るというようなことがあってはいけません。

Consistency - データベースの操作は、ある状態から次の状態へ、途中の状態無しに、移動します。例えば、オブジェクトAをデータベースに追加する時、そのユーザー（他のいかなるユーザーも同様）が、部分的なオブジェクトAを取得してはいけないということです。データベースは操作が完了しない状態のデータをさらしてはなりません。

Isolation - データベースに実行中の複数のトランザクションは、お互いを意識していません。そのため、もし2人のユーザーが同時に同じオブジェクトを変更しようとした場合、お互いの操作を妨害、または見ることもないように、データベースはオブジェクトへのアクセス順序を並べるようなメカニズムを持っていないければなりません。

Durability - トランザクションが一旦コミットされたら、それは必ず完了されなければなりません。それはたとえ、ハードウェアやソフトウェアの障害があったとしてもです。従って、ユーザーが3つのオブジェクトを削除するトランザクションをコミットした場合、2番目のオブジェクトを削除中にシステムがリポートされたとしたら、システムがリポートされた後、データベース自身を復旧するだけでなく、不完全なトランザクションを完了させなければなりません。つまり2つ目と3つ目のオブジェクトが削除されなければなりません。





RDBMSからSnackオブジェクトを取得するには、次のようなコードが必要です：

```
ResultSet results = statement.executeQuery("SELECT ID, Name, Cost, " +
    " Retail, Supplier FROM Snack WHERE ID = " + searchID);
if(results.next())
{
    snack = new Snack();
    snack.ID = results.getLong("ID");
    snack.name = results.getString("Name");
    snack.cost = results.getLong("Cost");
    snack.retail = results.getLong("Retail");
    snack.manufacturer = result.getString("Supplier");
    // ... Do something with the Snack object ...
}
}
```

Listing 1. RDBMSからsnackオブジェクトを取得します。ここでは与えられたsearchIDに合うIDをもつsnackオブジェクトを検索しています。

クエリの結果は、最初にresultsというコレクションに格納されます。それから、results内のオブジェクトから、Snackオブジェクトへ各フィールドの値をコピーします。さらに、このコピーの際、データベースで利用可能な型からJavaで利用可能な型への変換が行われます(この変換はメソッドの内部で行われるため、ユーザーからは見えません)。この反対の操作が、今度はオブジェクトをデータベースへ格納する際に発生します。大規模で、複雑なオブジェクト構造の場合、コードがどうなるか想像がつくでしょう。

文字列で構成されるSQL文を実行することによってプロセスが始まることにも注意が必要です。SQL文がプリコンパイルされていないと、データベースがそのSQL文を解析、実行することによってCPUリソースが余分に消費されます。

インピーダンスミスマッチは、オブジェクトリレーショナルデータベースではいくらか緩和されています。オブジェクトリレーショナルデータベース(ORDBMS)は、自動でオブジェクトとリレーショナルの変換を行うものです(リレーショナルデータベースは依然として使われていて、単に隠れているだけ)。結果として、上記のコードのような分解、組み立ては不要になります。

しかしながら、コードが不要になったとしても、どこかで行われています。一般的には、オブジェクトリレーショナルデータベースライブラリのメソッドやクラスのコレクションが格納されている、マッピングレイヤーに隠されています。このマッピングレイヤーが、アプリケーション内のオブジェクトと、テーブル、行、フィールドとの変換を行います。

そのため、プログラマはオブジェクトをオブジェクトとして扱うことができるので、コードが少なく済みますが、CPUリソースは、依然としてリレーショナルデータベースとオブジェクトとの変換に消費されています。そして、データベースライブラリ内のどこかで、データベーステーブルの行を取得し、行に格納するためにSQLが実行されなければなりません。

つまり、RDBMSはスペースと時間のオーバーヘッドをアプリケーションに付加してしまいます。スペースはオブジェクトとRDBMSを変換するコードに消費され、時間はそのコードの実行に消費されます。ORDBMSはいくつかの問題を解決しています。少なくとも、プログラマは変換コードを書くことから開放されます。しかし隠されているだけで、変換コードは存在していますので、メモリを消費し、CPUを消費するのです。

ソリューション

以上の取り上げてきた問題に対する解決策が、オブジェクト指向データベースです。そして実用的なオブジェクトデータベースは、db4oです(db4objects社提供)。db4oは最初のセクションで取り上げた問題と、2番目のセクションで取り上げた問題を同時に解決することができます。それではこれまで取り上げてきた課題をそれぞれ検討し、db4oの強みを見てみましょう。

- **最小リソース消費**: db4oライブラリはわずか400Kバイトです。しかしながら、この先すぐにわかりますが、この控えめなサイズであっても、機能を減らされてはいません。
- **高スループット**: db4oの実行速度は最高のデータベースと同等です。次の表は、典型的なSQLデータベースとパフォーマンスを比較したベンチマーク結果です。

barcelona benchmarks	read	write	query	delete
db4o/4.5.200	1.0	1.0	1.0	1.0
Hibernate / MySQL	20.8	32.2	6.7	17.3
Hibernate / HSQLDB	10.4	5.4	536.0	3.9
JDBC / MySQL	10.8	14.6	1.7	6.5
JDBC / HSQLDB	0.4	1.7	677.8	0.7
JDBC / Derby	3,696.0	12.9	1,299.7	7.1
JDO/VOA/MySQL	4.4	14.8	3.0	2.4

fastest

slowest

db4oのベンチマーク結果 上記は読み取り、書き込み、クエリ、そして削除の各操作について、典型的なSQLデータベースとdb4oの性能を比較したテーブルです。これらは5階層の継承のあるツリー構造からなる、複雑なオブジェクトに対する操作のベンチマークです¹。

- **実装が容易**: db4oのJavaバージョンは、単一のJARファイルです。一方.NETバージョンはDLLファイルです。コマンドラインツールでJava開発を行っているなら、CLASSPATHにファイルを追加するだけですし、javaまたは.NET開発でIDEを使っているなら、プロジェクトにファイルをドロップするだけです。

db4oのAPIには、やっかいで複雑なクラスやメソッドはありません。例えば、db4oプログラマは、主にdb4oのObjectContainerオブジェクトで作業をします。ObjectContainerオブジェクトはデータベースそのものを表しています。ObjectContainerインターフェースは、たった10のメソッドしか定義していません。しかしその10のメソッドで、ほとんどのデータベース操作を行うことができます。追加、検索、そして削除などです。

vendingmachineDBという名前のObjectContainerが既に開かれているとすると、snackオブジェクトをデータベースに格納するために必要なJavaのコードは、次のように非常にシンプルです。

```
vendingmachineDB.set (snack);
```

このコードは、すでにデータベースに格納されているSnackオブジェクトを更新したい場合にも使うことができます。もし上記のコードがアプリケーションの操作の最後であれば、データをコミットしてデータベースを閉じるコード、commit()とclose()メソッドをこの後に呼ぶ必要がありますが、同様のコードはRDBMSやORDBMSにも必要なものです。

- **ポータブル**: 前述したように、db4oのバージョンはJava版と.NET版が存在します。Java版は、Java 1.1以降(PersonalJavaとJ2ME CDCコンフィギュレーションを含む)の全てのプラットフォームで動作します。一方.NETバージョンは、.NET 1.0以降とCompact Frameworkで動作します。さらに、.NETバージョンは、全ての.NET言語を使って動作可能で、オープンソースのMonoフレームワークもサポートしています。

¹ ベンチマークに関するより詳しい情報はこちら: <http://www.db4o.com/about/productinformation/benchmarks/>

インピーダンスミスマッチのない世界へ

db4oは純粋なオブジェクトデータベースです。オブジェクトはそのまま格納されます。OR変換レイヤーは、見えるところにも見えないところにもありません。db4oはどんな複雑なオブジェクトでも格納することができます。オブジェクト構造そのものがデータベーススキーマなのです。



インピーダンスミスマッチのない世界 ドメインクラスやそのオブジェクト構造そのものがデータベースのスキーマに

一連のサンプルコードによって、db4oがいかに簡単かをお見せしましょう。前に取り上げた自動販売機のサンプルに戻って、db4oで同様のことを実行してみましょう。データベースを開き、新しいSnackオブジェクトを追加するコードは以下のようになります。

```
// Open an ObjectContainer
// (openFile creates it if it does not exist)
ObjectContainer vendingmachiningDB = Db4o.openFile("vmachine.YAP");

// Create a new Snack object and populate
// its fields.
// Constructor fields are:
// ID code
// Product name
// Cost in pennies
// Retail in pennies
// Supplier's Name
snack = new Snack(100,
    "Cheeze Zaps",
    1500, // Cost is 0.15
    5000, // Retail is 0.50
    "Sooper Cheeze Inc.");

// Put the snack into the database
vendingmachineDB.set(snack);

// A transaction is automatically started when
// the ObjectContainer is opened. Before closing,
// we should commit() the transaction that included
// the set() operation
vendingmachineDB.commit();
vendingmachineDB.close();
```

Listing 2. db4oデータベースにSnackオブジェクトを追加

オブジェクトを格納する操作は、ObjectContainerのset()メソッドに格納したいオブジェクトの参照を渡して実行するだけの簡単な操作です。commit()メソッドは、データベースをオープンしてから、又は最後にcommit()が呼ばれてから、更新されてset()されたオブジェクトをデータベースに書き込みます。この操作は、システム障害があっても失われることはありません。

注目して欲しいのはこの簡単さです。プログラマはデータをデータベースに格納するために、オブジェクトをいじる必要はありません。オブジェクトがあるがままに扱われているのです。プログラマがdb4oにすべきことは、“このオブジェクトを格納して”ということだけです。

格納したSnackオブジェクトを取得するのは簡単です。db4oは斬新な、Query By Example(QBE)を使ってオブジェクトを特定します。取得したいオブジェクトのテンプレートをdb4oに渡せばいいのです。テンプレートとは、検索したい項目をセットした同一クラスのオブジェクトのことです。

vendingmachineDBというObjectContainerがすでにオープンされているとすると、Snackオブジェクトを取得するコードは以下のようになります。

```
// Create a template object.
// Retrieve the snack by ID number.
// Other fields are null or 0, and will be
// ignored by the query
snackTemplate = new Snack(100,null,0,0,null);

// Issue the query
ObjectSet result =
    vendingmachineDB.get(snackTemplate);

// Fetch the retrieved snack
if(result.hasNext())
{
    Snack snack = (Snack)result.next();
    // Do something with the snack.
    . . .
}
```

Listing 3. db4oデータベースからSnackオブジェクトを取得。

先に紹介したRDBMSのコードと比較してください。snackオブジェクトは一回で取得できます。つまりオブジェクトのフィールドにデータを当てはめるコードをプログラマは書く必要がありません。これはどんなにフィールドが多くても同じです。そしてexecuteQuery()メソッドに渡されるSQL文も不要です。

オブジェクトの削除も同様にシンプルです。オブジェクトが一旦データベースから取り出されたら、ObjectContainerのdelete()メソッドに、削除したいオブジェクトの参照を渡すだけです。

```
// Create a template object
// This time, query the snack by name
snackTemplate = new Snack(
    0, "Cheeze Zaps", 0, 0, null);

// Issue the query
ObjectSet result =
    vendingmachineDB.get(snackTemplate);

// Get the retrieved object
if(result.hasNext())
{
    Snack snack = (Snack)result.next();
    // Delete the snack
    vendingmachineDB.delete(snack);
}
```

Listing 4. データベースからSnackオブジェクトを削除する

ここでも、オブジェクトへの操作は一回です。オブジェクトはあるがままに取り扱われます。

これまで取り扱ってきた全てのサンプルは、次のような非常にシンプルなSnackオブジェクトでしたので、もっと複雑なオブジェクトだったらと思われるかもしれません。



```

public class Snack {
    private int id;
    private String name;
    private long cost;
    private long retail;
    private String supplier;
    // Constructors and accessors follow
    ...
}

```

Listing 5. Snackクラス

Snackクラスの2つ目と最後のメンバーは、Stringオブジェクトです。JavaではStringはプリミティブではなくオブジェクトなので、Snackオブジェクトを格納するということは、そのメンバーである2つのStringオブジェクトも格納していることとなります。同様にSnackオブジェクトを削除するということは、同時にそれら2つのStringオブジェクトも削除することとなります。こういった作業をdb4oは背後で自動的に行っているわけです。こうしてわかるように、プリミティブとStringオブジェクトは、アイデンティティを持たないために、それらが所属するオブジェクトと一体のものとして扱われます。

それではString以外のオブジェクトの場合はどうでしょうか？問題はありません。どんな複雑なオブジェクト構造でも、db4oは同様に簡単に扱うことができます。それでは、SnackSlotという自動販売機の中で商品を格納しているスロットを表すオブジェクトを導入してみましょう。ここでいうスロットとは、複数の商品を格納できる、商品ごとの棚のことです。

```

public class SnackSlot {
    int slotnum; // Slot number
    Snack thisSnack; // Snack in this slot
    int original; // Original number stocked
    int current; // Number of snacks left
    // Constructors and accessors follow
    ...
}

```

Listing 6. SnackSlotクラス

このように、SnackSlotは、このスロットに入れられ、消費されることになるSnackオブジェクトへの参照を持っています。また、このスロットに入れられたSnackのオリジナルの数量と、消費された後に残っている数量を記録しています。こうすることでいくつのSnackがそのスロットで売られたかを自動販売機が判断することができるようになります。

商品を追加したり、スロットを追加したりする機能などは既実装したと仮定しましょう。その場合その内部では個々のオブジェクトに対してset()を呼び出していると思うかもしれませんが。しかしその必要はありません。

あるオブジェクトが初めてデータベースに追加される時は、db4oは全ての参照を辿ってそれらも同時に、自動で、格納します。ですからSnackSlotとその中のSnackを格納するには、以下のコードのようになります：

```

// Create the Snack objects
snack = new Snack(101,
    "Nacho Novas",
    1200, // Cost is 0.12
    7500, // Retail is 0.75
    "Sooper Cheeze Inc.");

```

```

// Create the SnackSlot object
snackslot = new SnackSlot(
    10,      // Slot number 10
    snack,  // Connect the snack to the slot
    10,     // 10 bags on this slot...
    10);    // ...none sold yet

// Put the new SnackSlot object in the database
// The Snack is stored as well.
vendingmachineDB.set(snackslot);

```

Listing 7. 複雑なオブジェクトをdb4oに追加する

SnackオブジェクトとSnackSlotオブジェクトを取得、削除するのは少し複雑になりますが、それはdb4o内に永続化されたオブジェクトをより細かくコントロール、つまりより繊細なライフサイクルマネジメントを可能にするためです。

オブジェクト参照を持つ複合オブジェクトを取得するには、どれだけ深く参照階層を辿ってオブジェクトを取り出したいかをdb4oに指示する必要があります。この深さのことを、“activation depth”と呼んでいます。例えばactivation depthが1のときは、オブジェクトのメンバーを辿らずに、そのオブジェクトのメンバーだけを取得します。activation depthを2に設定すると、オブジェクトのメンバーも取得します。ただしそのオブジェクトのオブジェクトメンバーは取得されません。厳密に言うと、activation depthが0の時に取得されています。この時オブジェクトはアイデンティティを持っていますが、まだメンバーフィールドに値が設定されていません。ここでは、値が設定された状態のことを“取得する”として説明しています。

db4oのactivation depthのデフォルト値は5に設定されているので、**listing 3**で使用したコードを使ってSnackSlotとその中のSnackオブジェクトを取得することができます。

一方削除については、参照されているオブジェクトをまとめて一度に削除することが可能です。しかしながら、そのようにdb4oに指示する必要があります。もし指示をしないと、そのオブジェクトだけ削除しますので、参照されているオブジェクトはデータベースに残ることになります。もしSnackSlotオブジェクトが削除された時に、同時に参照されているSnackオブジェクトも削除したい場合には、SnackSlotオブジェクトの“cascaded delete”フラグをオンにします。これはdb4oのグローバル設定を使用して以下のように設定します。

```

Configuration config = Db4o.configure();
ObjectClass oc = config.objectClass("<package>.SnackSlot");
oc.cascadeOnDelete(true);
...

```

Listing 8. cascaded deleteフラグをオンにする

このコードは、db4oにSnackSlotオブジェクトが削除される時はいつも、全てのメンバーを削除するように指示しています(この場合は参照されているSnackオブジェクトメンバー)。**listing 8**のように、わずかな設定を最初にするだけで、**listing 4**のコードでSnackSlotオブジェクトと関連するSnackオブジェクトを削除することができます。

明らかに、cascaded deleteはいつも適切だとは限りません。そのためにデフォルトではこの機能はオフとなっており、設定でオンにするようになっています。この仮想自動販売機の例では、SnackSlotは削除されることがあるかもしれませんが、その中のSnackはまた別のSnackSlotで利用されるかもしれないので、cascaded deleteはオフにしておきましょう。こうすることで、SnackSlotが削除されたとしても、そのスロットに入っている商品を再利用できるように、SnackSlotに参照されていたSnackはデータベースに残ることになります。



ネイティブクエリ(Native Queries)

db4oのQBEは、簡単に言ってしまえば、SQLのWHERE句で完全一致を使っているようなものです。これは構造化されたオブジェクトのルートを特定するには十分な機能でしょう。そうして一旦ルートを取得したら、そこから参照を辿って必要なオブジェクトにアクセスできます。オブジェクトツリー自体がdb4oデータベース内にあるので、このように構造化されたオブジェクトの参照を簡単に辿ることができます。この特徴を使うと、様々なクエリを実行しなくては取得できないオブジェクトに、極めてシンプルにアクセスできるわけです。

しかしながら、より高機能なクエリが必要なことがよくあります。例えば、データベース内にあるオブジェクトを、完全一致ではない条件で取得したい場合です。より複雑なクエリを実行するには、より複雑なクエリ文が必要だと思うでしょう。通常のデータベースではその通りなのですが、db4oは違います。

db4oのネイティブクエリ(NQ)APIは、QBEが対応できないクエリを簡単に行うことができます。そしてこのネイティブクエリも、db4oの“使いやすさ”というコンセプトでデザインされているため、クエリ言語を学習する必要がありません。もちろんdb4o独自のへんてこなAPIを学習する必要もありません。どうということかという、ネイティブクエリを使えば、アプリケーションを書くために必要な言語以外必要ではないということです。

listing 3でお見せしたクエリを思い出してください。そこでは、IDが100のSnackオブジェクトを取得しました。同じことをネイティブクエリで実装してみると、次のようになります。

```
List<Snack> snacks =
    vendingmachineDB.query<Snack>(new Predicate()
    {
        public boolean match(Snack snack)
        { return( snack.id == 100); }
    })

if(snacks.hasNext())
{
    Snack snack = snacks.next();
    // Do something with snack
    ...
}
```

Listing 9. listing 3をネイティブクエリで実装.

このコードでは、db4oのネイティブクエリAPIだけでなく、JavaのGenericsも活用したタイプセーフなクエリも活用しています。GenericsはJDK5から追加された機能ですが、ネイティブクエリそのものは遥か昔のJDK1.1でも利用可能です。

また、上記クエリの例から離れて純粋にネイティブクエリを見てみると、その驚くべき特徴が明らかになります。まず、NQはPredicateクラスを定義しています。このクラスはクエリのエンジンそのものと言えます。このクラスは1つだけ抽象メソッド、match()を持っています。Predicateクラスのサブクラスは、このメソッドを実装しなければなりません。このメソッドは1つ引数を持っているのですが、それは検索の対象になるオブジェクトになります。**listing 9**の例で言うと、対象オブジェクトはSnackオブジェクトになります。

このmatch()メソッドの戻り値はbooleanです。つまり、このメソッドの基本的な意味は、引数のオブジェクトにフィルターをかけることであると言えます。分かりやすく言うと、引数のオブジェクトが条件に合う場合にmatch()メソッドはtrueを返し、そうでない場合はfalseを返します。こうし



てクエリが実行される時、対象オブジェクト、この例の場合はSnackオブジェクトが、match()メソッドに渡されて評価され、そのオブジェクトが検索条件に合っていて結果Listに含まれるべきものかどうかチェックされることとなります。

ネイティブクエリには数多くの特徴があり、アプリケーションと同一のプログラミング言語でクエリが実行できるということだけではありません。例えば、どの商品が最もよく売れているか知りたいとしましょう。ここでは5個以上売れているものをNQ APIを使って実装してみます。

```
// Fetch list of snacks that have sold
// 5 items or more.
// The list is returned in
// popularSnacks ArrayList

List<SnackSlot> slots =
    vendingmachineDB.query<SnackSlot>(new Predicate()
    {
        public boolean match(SnackSlot snackslot)
        { if(snackslot.original == 0)
            return false;
          return((snackslot.original -
            snackslot.current) > 4);
        }
    })
while(slots.hasNext())
{
    SnackSlot goodSlot = slots.next();

    // Read the snack object in, too
    vendingmachineDB.activate(goodSlot,2);
    popularSnacks.add(goodSlot.thisSnack);
}
... process popularSnacks...
```

Listing 10. NQを作ったより複雑なクエリ

ネイティブクエリを使うと、対象オブジェクトのフィールドを使って計算をすることができるので、5個以上売れたSnackSlotオブジェクトを検索することができます。whileループ内では、該当する各SnackSlotオブジェクトをactivate()してそのSnackオブジェクトをデータベースから取り出しています。先に説明したように、このactivate()メソッドは、db4oにどれだけ深く参照を辿ってオブジェクトをデータベースから取り出すかを指示して取り出すメソッドです。この深さ2というのは、SnackSlotオブジェクトだけでなく、Snackオブジェクトも取得するという指示になります。

このクエリを実行した後、アプリケーションはpopulateSnacks ArrayListを処理しています。この後、例えば補充が必要なスロットをオーナーに連絡するといった高度な処理が実行されるでしょう。

もちろん、クエリの比較部分をよりアドホックな複雑なものを実装することができたでしょうし、activate()メソッドを使用して他のオブジェクトメンバーを取得するようなこともできます。NQの素晴らしい点は、クエリがプログラミング言語、この場合はJava、で表現されている限り、どのようなクエリも実装できるということです。もちろん実際にはまだ数多くのデメリットが発生する場合がありますので、詳細はdb4objects社のウェブサイトまたはdb4oのチュートリアルを参照してください。

これらの特徴を通して最も素晴らしい特徴は、完全にタイプセーフであり、リファクタリング可能であると言う点です。もしSQLのような文字列のクエリを使用していると、オブジェクトのフィールドの参照で型や名称などを間違ってもそれらは実行時までエラーが発生しません。一方ネイティブクエリを使うと、コンパイラが指摘してくれます。もし開発が進行してクラスに変更が発生した場合には、SQLのような文字列だと手作業になりますが、NQならIDEによって自動で修正されるでしょう。



最後に

ここではざっと、db4oの主要な機能の一部を紹介しただけですが、少なくとも省スペース、信頼性のある、高レスポンスというデータベースに必要な要件を満たしていることをご紹介できたと思います。さらに、RDBMSやORDBMSがインピーダンスミスマッチを解消しようとするマッピングレイヤーが、db4oでは完全に無くなっていることをご紹介できました。さらに、db4oのネイティブクエリの機能によって、複雑なクエリでも簡単に、メンテナンスし易く、RDBMSのクエリによくあるプログラマが陥りやすい数多くの問題が解消されます。

ここで取り上げることができなかったdb4oの主要な特徴:

- ・**オブジェクトバージョンの変更**: まだご紹介していない機能がたくさんあります。例えばdb4oを使うと、いかに簡単にオブジェクトのバージョン変更に対処できるかということです。Snackクラスを変更してSnackTypeメンバーを追加したとすると、古いSnackクラスで生成され格納されたオブジェクトにその新しい変更を適用することができるのです。データベースをそのまま使い続けることができます。db4oはたった一つのメソッドを実行するだけで、データベースに格納されているオブジェクトのクラス名を変更することができます。例えばSnackクラスをOldSnackと変更することができるわけです。一方RDBMSでは、テーブルに変更を加え、クエリやマッピングコードも変更しなければなりません。
- ・**オブジェクトの同期化**: インテリジェントなデバイスで、常時接続でないデバイスは、マスターデータベースからデータのサブセットを受け取って一時的に保存し、接続していない時にそれらのデータで作業する必要があります。その後再接続して、更新されたデータをマスターデータベースに対して同期化を行います。db4oはこの機能を持っていて、これをdb4o's Replication System (dRS)²と呼んでいます。
- ・**ゼロメンテナンス**: さらに面白いことに、これまでdb4oの管理については何も話をしなかったことには素晴らしい理由があります。db4oは仮想的にゼロメンテナンスです。すべてと言っても過言ではありません。従って、モバイルデバイスアプリケーションや、情報端末、インテリジェントメディカルシステムなど、その他データベースがユーザーから見えない製品に最適です。
- ・**GPLライセンス(無料)でオープンソース利用可**: 最後に、最もdb4oが魅力的であろう特徴をご紹介します。Javaと.NETバージョン両方とも、オープンソースです。もし商用アプリケーションにdb4oを利用したい場合も、低コストで組み込むことが可能です。

しかしこれらの特徴は、db4oのエッセンスがあって初めて意味を持ちます。つまり、db4oは、軽量、パワフル、そして実用的な、Java/.NET向けオブジェクト指向データベースであるということです。

² db4oレプリケーションシステムの詳細情報: <http://www.db4o.com/about/productinformation/features/drs.aspx>