# Javaの理論と実践: データベース無しにデータベース・クエリーを行う

## データ・モデルを借りることで、開発が簡単に、そしてパフォーマンスが向上

Brian Goetz
Principal Consultant
Quiotix

2005年 5月 31日

昔から『金槌を手にすると、何でも釘に見える』と言われています。しかし、金槌を持っていなかったらどうでしょう。そう、場合によっては金槌を借りればよいのです。そうすれば、借用した金槌を使って仮想的な釘を打ち込み、金槌を返却し、それで全てが済むのです。今回のJavaの理論と実践では、一時的なデータに対して、SQLやXQueryなど操作用の金槌をどのように適用するかについて、Brian Goetzが解説します。

このシリーズの他の記事を見る

私は最近、Webを次々に訪れて回る、Webクローリング（Web crawling）プロジェクトに対して提言を行いました。そのプログラムは、様々なWebサイトをクロールし、それらのサイトや各ページ、各ページに含まれるリンク、各ページに対する分析結果などのデータベースを構築するのです。最終的な結果として、どのサイトやページを訪ねたか、どのリンクはつながっているか、どのリンクは切れているか、どのページにエラーがあるか、計算したページ・メトリックスなどの詳細を集めた、一連のレポートができ上がります。最初は、どのようなレポートが必要なのか、どのようなフォーマットのレポートにすべきかが、誰にも分かりません。単にレポートが必要なことが分かっているだけなのです。ということから、このレポート開発のフェーズは繰り返し作業であることが分かります。結果を見て微調整を行い、場合によっては別の構成でやり直す、といったフィードバック作業を何回か繰り返す必要があります。レポートの要求事項として唯一規定されているのは、XML（場合によってはHTML）で表示されなければならない、ということだけです。つまり、レポートの要求事項は最初から規定されているわけではなく、「動的に発見される」のです。ですからレポートの開発、修正プロセスは、軽量でなければなりません。

## データベースは必要ありません

この問題に対して「明白な」手法は、全てをSQLデータベースに入れてしまうことです。つまり、ページやリンク、メトリックス、HTTP結果コード、タイミング結果、その他のメタデータを全てSQLデータベースに入れるのです。訪ねたページの内容は保存する必要がなく、ページの構造やメ

タデータのみを保存する必要があるため、これは結局、リレーショナル表現の問題ということになります。

ここまでは、典型的なデータベース・アプリケーションのように思え、パーシスタンス戦略の候補には事欠きませんでした。しかし、パーシスタンスのためにデータベースを使う、という複雑さは避けられそうなのです。クローラー（crawler）は、せいぜい数万ページを訪ねるだけです。この数字はデータベース全体をメモリー内に保持できる程度に小さな値であり、もしパーシスタンスが必要であれば、シリアル化によって行うこともできます。（確かに、ロードやセーブには長い時間がかかりますが、ロードやセーブは頻繁に行うものではありません。）『無精を決めこめば儲かる』。パーシスタンスを扱わなくて済むことによって、アプリケーション開発に必要な時間を大幅に削減でき、開発努力という面で大きな節約ができます。メモリー内データ構造を構築、操作する方が、データを追加し、フェッチし、分析する操作の度にデータベースにまで行くよりは、ずっと容易です。（パーシスタンスを扱うと）どのようなパーシスタンス・モデルを選んだとしても、データに触れるコードの構成方法が必ず制約されてしまうのです。

メモリー内データ構造は、クロールの対象である様々なサイトのホームページをルートにした、一種のツリー構造を持っています（リスト1）。ですから、そこで検索を行ったり、そこからデータを抽出したりするには、Visitorパターンが理想的でした。（リンクのサイクル（AがBにリンクされ、BがCにリンクされ、CがAにリンクして戻る）に巻き込まれないベースVisitorクラスを作るのは、それほど難しくありませんでした。）

## リスト1. 単純化した、Webクローラー用のスキーマ

```
public class Site {
    Page homepage;
    Collection<Page> pages;
    Collection<Link> links;
}
public class Page {
    String url;
    Site site;
    PageMetrics metrics;
}
public class Link {
    Page linkFrom;
    Page linkTo;
    String anchorText;
}
```

クローラー・アプリケーションには、1ダースほどのVisitorがあり、さらなる分析のためにページを選択したり、切れているリンクを持つページを選択したり、エラーのあるページを選択したり、「最も頻繁にリンクされる」ページを一覧表にしたり、といったことをしています。これらの操作はどれも非常に単純なので、このVisitorパターンは非常にうまく動作しました（リスト2）。また、データ構造はメモリー内に収まるため、徹底的に行われる検索であっても、十分安上がりに済むのです。

## リスト2. Webクローラー・データベース用のVisitorパターン

```
public interface Visitor {
    public void visitSite(Site site);
    public void visitLink(Link link);
}
```

## おっと、レポートを忘れた

Visitorパターンは、データ・アクセスに関しては非常にうまく動作しましたが、レポートの実行では話が別でした。パーシスタンスのためにデータベースを使う利点の一つは、レポート生成となるとSQLの真価が発揮されるということです。データベースに全てを任せられる、と言っても良いほどなのです。さらに良いことは、SQLを使うと、レポートのプロトタイプ化が容易に行えることです。プロトタイプ・レポートを実行し、その結果が望むものではなかった場合には、SQLクエリーをいじるか、または新しいクエリーを書き、再度試せばよいのです。変更がSQLクエリーのみである場合には、編集、コンパイル、実行というサイクルは非常に迅速です。SQLがプログラムに保存されていない場合には、このサイクルのコンパイル部分をスキップすることさえでき、そのためレポートを非常に速くプロトタイプ化できます。いったん必要なレポートが見つかれば、そのレポートをアプリケーションの中に組み込むのは非常に容易です。

ですから、メモリー内データ構造は、新しい結果の追加や特定な結果の検索、様々な種類の一時的トラバースなどには最適なのですが、レポートとなると、とたんに不利になるのです。データベース構造と類似していない構造を持つレポートの場合は、Visitorはレポート・データを保持する、全く新しいデータ構造を作らなければなりません。従って各タイプのレポートには、結果を保持するための、そのレポート独自の中間データ構造や、中間データ構造に中身を入れるビジター、中間データ構造を最終的なレポートにするための後処理コードなどが必要になります。プロトタイプ化されたレポートの大部分は捨てられることを考えると、この手法では大仕事のようです。例えば、次のような例を考えてみてください。対象サイトにリンクした他のサイトの全ページと、こうした外部ページそれぞれに対して、（そのサイト上にある）対象サイトにリンクしたページのリストをリストアップしたレポートを作り、その後で、リンク数でレポートをソートし、最も頻繁にリンクされるページが最初になるようにするのです。この例は基本的に、データ構造を裏返しにするものです。この種のデータ変換をVisitorで行うためには、対象のサイトから到達できる外部ページへのリンクのリストを使い、リンクされるページ数によるソートを行うのです。これをリスト3に示します。

## リスト3. 最も頻繁にリンクされるページと、それらにリンクしたページとをVisitorが一覧表にする

```
public class InvertLinksVisitor {
    public Map<Page, Set<Page>> map = ...;

    public void visitLink(Link link) {
        if (link.linkFrom.site.equals(targetSite)
            && !link.linkTo.site.equals(targetSite)) {
            if (!map.containsKey(link.linkTo))
                map.put(link.linkTo, new HashSet<Page>());
            map.get(link.linkTo).add(link.linkFrom);
        }
    }
}
```

リスト3のVisitorは、外部ページそれぞれを、それにリンクされた内部ページのリストで関連付けたマップを作成します。レポートを作るには、関連付けられたページ・セットのサイズでエントリーをソートするのです。これらのステップはどれも難しくはありませんが、レポートそれぞれに対する、そのレポート特有のコードの量は膨大です。（レポートに関する要求事項が明示されているわけではないため）レポートのプロトタイプ化を迅速に行えることが重要な目標であることを考えると、新しいレポートを試すためのオーバーヘッドが大きくなりすぎます。レポート数

が多いということは、データを選択、集約、ソートするために、何度もデータにアクセスする必要があるということです。

## 私のデータ・モデル王国

収集されるデータを記述し、また選択や集約のクエリーを表現する対象となる正式なデータ・モデルが無いことが、この時点で不利に見え始めます。どうやら、無精を決めこむのは、最初期待したほど効率的ではなかったようです。しかし、このアプリケーションには正式なデータ・モデルがありませんが、データをメモリー内データベースにストリーミングし、それに対してクエリーすることによって、「データ・モデルを借用」できるかも知れません。そうだとすると、すぐに2つの可能性が頭に浮かんできます。オープンソースのメモリー内SQLデータベースであるHSQLDBと、XQueryです。私にはデータベースが提供するパーシスタンスは必要ありませんでしたが、クエリー言語は必要だったのです。

HSQLDBは、埋め込み可能なデータベース・エンジンであり、Java™言語で書かれています。HSQLDBは、メモリー内テーブルとディスク・ベースのテーブルの両方に対するテーブル型を持っており、アプリケーション内に完全に埋め込まれるように作られているため、大部分の本物のデータベースにまつわる、管理オーバーヘッドが必要ありません。HSQLDBにデータをロードするためには、メモリー内データ構造をトラバースする、そして保存すべき各実体に対して適切なINSERTステートメントを生成するVisitorを書けばよいだけです。そして、レポートを行うためにメモリー内データベース・テーブルに対してSQLクエリーを実行し、終わったら「データベース」を捨てるのです。

### おっと、リレーショナル・データベースが面倒なことを忘れていました

HSQLDBの手法は実行可能なものだったのですが、すぐに、オブジェクト・リレーショナル(object-relational)のインピーダンス・ミスマッチによる罰金を、一度ならず二度も払う羽目になることが明らかになりました。一度はツリー構造のデータベースをリレーショナル・データモデルに変換する時、もう一度は、フラットなリレーショナル・クエリーの結果を、構造化XML結果セットまたはHTML結果セットにと変換する時です。しかもJDBC ResultSetをXML文書またはHTML文書のDOM表現に後処理することは、相変わらず大仕事であり、レポート毎に何らかのカスタム化が必要になります。ですから、メモリー内SQLデータベースによって確かにクエリーは単純になったのですが、データベースにデータを出し入れするための余分なコーディングによって、節約した分が食いつぶされてしまうのです。

## そこへXQueryが救いに

手軽に使用できるデータ・クエリーの代替手段のもう一つは、XQueryです。XQueryには、クエリーを行った結果、XMLまたはHTML文書を作るようにできている、という利点があります。ですからクエリー結果に対して、後処理が何も必要ないのです。この点は魅力的です。各レポートに対して、1レイヤーのコーディングが必要なだけで、2レイヤーやそれ以上のコーディングは必要ありません。ですから最初の課題は、全データ・セットを表すXML文書を作ることです。単純なXMLデータ・モデルを設計し、データ構造をトラバースして各要素をDOM文書に付加するVisitorを書くのは簡単でした。( この文書を書き出す必要はありませんでした。この文書はクエリー用にメモリー内に保持され、終了すると捨てられます。基礎となるデータが変更されたら、単純に再生成されます。 ) その後、必要なこととしては、XQueryクエリーを書くことだけです。このXQueryクエリーが、レポート用にデータ選択と集約を行い、最終的に必要な形式 ( XMLまたは

HTML ）にフォーマットするのです。このクエリーは、迅速なプロトタイプが行えるように、そして複数のレポート・フォーマットがサポートできるように、別ファイルに保存することができます。Saxonを使ってクエリーを評価するためのコードをリスト4に示します。

## リスト4. XQueryクエリーを実行し、結果をXMLまたはHTML文書にシリアル化するためのコード

```
String query = readFile(queryFile + ".xq");
Configuration c = new Configuration();
StaticQueryContext qp = new StaticQueryContext(c);
XQueryExpression xe = qp.compileQuery(query);
DynamicQueryContext dqc = new DynamicQueryContext(c);
dqc.setContextNode(new DocumentWrapper(document, z.getName(), c));
List result = xe.evaluate(dqc);
FileOutputStream os = new FileOutputStream(fileName);
XMLSerializer serializer = new XMLSerializer (os, format);
serializer.asDOMSerializer();
for(Iterator i = result.iterator(); i.hasNext(); ) {
    Object o = i.next();
    if (o instanceof Element)
        serializer.serialize((Element) o);
    else if (o instanceof Attr) {
        Element e = document.createElement("scalar");
        e.setTextContent(((Attr) o).getNodeValue());
        serializer.serialize(e);
    }
    else {
        Element e = document.createElement("scalar");
        e.setTextContent(o.toString());
        serializer.serialize(e);
    }
}
os.close();
```

データベースを表現するXML文書の構造は、メモリー内のデータ構造とは少し異なります。各<site>要素にはネストした<page>要素があり、各<page>要素にはネストした<link>要素があり、各<link>要素には<link-to>要素と<link-from>要素があります。この表現が、大部分のレポートでは十分便利に使えることが分かりました。

リスト5はXQueryの一例であり、リンクの選択、ソート、プレゼンテーションなどを処理します。これにはVisitorによる手法に比べて幾つかの利点があります。コード量が少ない（クエリー言語が選択や集約、ソートをサポートしているため）だけではなく、レポート用の全コード、つまり選択や集約、ソート、プレゼンテーションなどのコードの全てが、一ヶ所にあるのです。

### リスト5. 最も頻繁にリンクされるページのレポート全体を作成するXQueryコード

```
<html>
<head><title>Most-linked pages</title></head>
<body>
<ul>
{
  let $links := //link[link-to/@siteUrl ne $targetSite
                       and link-from/@siteUrl eq $targetSite]
  for $page in distinct-values($links/link-to/@url)
  let $linkingPages := $links[link-to/@url eq $page]/link-from/@url
  order by count($linkingPages)
  return <li>Page {$page}, {count($linkingPages)} links <ul> {
      for $p in $linkingPages return <li>Linked from {$p/@url}</li>
    }
    </ul></li>
}
</ul> </body> </html>
```

## まとめ

開発コストという観点から見ると、XQueryを使った手法で大きな節約ができることが分かりました。ツリー構造は、データ構築や検索には理想的ですが、レポートに理想的とは言えません。XMLの手法は（XQueryの力を利用できるため）レポートには適していますが、不便であり、アプリケーション全体の実装する上でのパフォーマンスは悪くなります。このデータ・セットは扱いやすい程度のサイズ（数十メガバイト以内）だったので、開発面から見て一番便利なように、データを一つの形式から別の形式に変換することができました。全体を手軽にメモリーに保存するには大きすぎるような、より大きなデータ・セットであれば、データベースを基にアプリケーションを構築することが必要だったでしょう。パーシスタント・データ(persistent data)を扱うための良いツールは数多くありますが、それらを使うためには、単純にメモリー内データ構造を操作する以上の面倒な作業が必要になってきます。データ・セットが適切なサイズであれば、両者の良いところだけを組み合わせることができるのです。

# 著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。