実用的なGroovy: GroovyによるJDBCプログラミング

次回のレポート・アプリケーションはGroovySqlを使って構築する

Andrew Glover
Scott Davis

2005年 1月 11日

Groovyの実際的な知識をさらに深めましょう。今回は、Andrew GloverがGroovySqlの使い方として、単純なデータ・レポート・アプリケーションの構築を解説します。GroovySqlは、クロージャー（closure）とイテレーター（iterator）を組み合わせることによってリソース管理の負担を開発者からGroovyフレームワーク自体に移し、JDBC（Java Database Connectivity）プログラミングを楽なものにしてくれます。

このシリーズの他の記事を見る

実用的なGroovyシリーズのこれまでの記事では、Groovyの持つ便利な特徴の幾つかを見ました。第1回の記事では、Groovyを使うことによって、通常のJava™コードのユニット・テストが単純に、高速になることを学びました。第2回の記事では、Groovyの持つ表現力の豊かさがAntビルドに何をもたらすかを学びました。今回はGroovyの実際的な使い方として、もう一つの例、SQLベースのレポート・アプリケーションを迅速に構築する方法を学びます。

このシリーズについて

どのようなツールであれ、開発作業の中に採り入れるためには、どういう場合に使うべきか、または使うべきではないかをよく知る必要があります。スクリプト言語は非常に強力なツールですが、その強力さは、適切なシナリオで適切な使い方をした場合にのみ発揮されます。実用的なGroovyシリーズはそうした点を念頭に置き、Groovyの実用的な使い方に焦点を絞って、どういう場合に、どのように使うのかを解説して行きます。

レポート・アプリケーションを迅速に構築するためには、通常はスクリプト言語が素晴らしいツールとなります。しかしGroovyを使うと、明らかに、もっとずっと楽にできるのです。Groovyの持つ軽量な構文によって、Java言語でのJDBCの持つ冗長性が多少軽減されます。それよりももっと重要なのは、クロージャーによって、リソースの処理の責任がクライアントから、（処理を行いやすい）フレームワーク自体に移るということです。

今回の記事では、まずGroovySqlの機能を簡単に説明し、単純なデータ・レポート・アプリケーションの構築を通してGroovySqlの応用方法を解説します。この記事をよく理解するためには、Javaプラットフォームでの JDBCプログラミングに慣れている必要があります。また、ここではクロージャーが重要な役割を果たすので、前回の記事で説明した、Groovyにおけるのクロージャーの紹介も見る必要があるかも知れません。ただし、Groovyで行われたJDBCの機能強化の中

では、イテレーターが重要な役割を果たしているので、今回の記事で最も重要な概念はイテレーターです。そこで、Groovyでのイテレーター・メソッドの概要から始めることにします。

イテレーター入門

繰り返し（iteration）は、あらゆるプログラミング状況において、最も一般的で便利なものです。イテレーターというのは一種のコード・ヘルパーであり、これを使うことによって、任意のコレクションまたはコンテナーの中のデータに対して、一度に一つ、アクセスできるようになります。Groovyはイテレーターを暗黙的に、また使いやすくしており、Java言語でのイテレーターの概念を改善しています。リスト1を見ると、Java言語を使って`String`集合の各要素を出力するために、どれほどの手間がかかるかが分かります。

リスト1. 通常のJavaコードでのイテレーター

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class JavaIteratorExample {
  public static void main(String[] args) {
     Collection coll = new ArrayList();
     coll.add("JMS");
     coll.add("EJB");
     coll.add("JMX");
     for(Iterator iter = coll.iterator(); iter.hasNext();){
        System.out.println(iter.next());
     }
  }
}
```

リスト2を見ると、Groovyでどれほど簡単になるかが分かると思います。ここでは、`Iterator`インターフェースを迂回し、コレクション自体に対してイテレーター風のメソッドを使っています。さらに、Groovyのイテレーター・メソッドは、繰り返しサイクル毎に呼び出されるクロージャーを受け付けています。リスト2は、先に挙げたJavaベースの例をGroovyで変換したものです。

リスト2. Groovyでのイテレーター

```
class IteratorExample1{
   static void main(args) {
     coll = ["JMS", "EJB", "JMX"]
     coll.each{ item | println item }
   }
}
```

ご覧の通り、典型的なJavaコードとは異なり、Groovyは、私が必要とする振る舞いは渡せるようにしながら、繰り返しに特有のコードを制御しています。Groovyは、この制御によってリソース処理の責任を、私の手からGroovy自体にと、巧みに移しているのです。Groovyにリソース処理の責任を負わせるということは、非常に強力なことです。これによってプログラミングがずっと容易になり、必然的に、早く進むようになります。

GroovySqlを導入する

GroovyのSQL魔術は、GroovySqlと呼ばれる優雅なAPIの中にあります。GroovySqlはクロージャーとイテレーターを使うことによって、JDBCのリソース管理を、開発者からGroovyフレームワークにと移すのです。そのため開発者はJDBCプログラミングの重荷から開放され、クエリーとその結果に専念できるようになるのです。

通常のJDBCプログラミングがいかに面倒なものかを忘れてしまった人のために、私が思い出させて差し上げましょう。リスト3は、Java言語での、単純なJDBCプログラミングの例です。

リスト3. 通常のJavaでのJDBCプログラミング

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class JDBCExample1 {
  public static void main(String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try{
      Class.forName("org.gjt.mm.mysql.Driver");
      con = DriverManager.getConnection("jdbc:mysql://localhost:3306/words",
           "words", "words");
      stmt = con.createStatement();
      rs = stmt.executeQuery("select * from word");
      while (rs.next()) {
        System.out.println("word id: " + rs.getLong(1) +
            " spelling: " + rs.getString(2) +
            " part of speech: " + rs.getString(3));
      }
    }catch(SQLException e){
      e.printStackTrace();
    }catch(ClassNotFoundException e){
      e.printStackTrace();
    }finally{
      try{rs.close();}catch(Exception e){}
      try{stmt.close();}catch(Exception e){}
      try{con.close();}catch(Exception e){}
   }
  }
}
```

いやはや・・・。リスト3では、単にテーブルの内容を見るだけなのに、約40行ものコードがあるのです！ GroovySqlを使うと、どのくらいになると思いますか？ 10行以上だと思ったら、間違いです。Groovyを使うと、いかに優雅に本来の作業、つまり単純なクエリー、に専念できるようになるか、そして、下にあるリソース処理をGroovyが行ってくれることを見てください（リスト4）。

リスト4. GroovySqlへようこそ！

```
import groovy.sql.Sql
class GroovySqlExample1{
  static void main(args) {
    sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
           "words", "org.gjt.mm.mysql.Driver")
    sql.eachRow("select * from word"){ row |
       println row.word_id + " " + row.spelling + " " + row.part_of_speech
    }
  }
}
```

悪くないでしょう。`Connection`終了や`ResultSet`終了、その他、JDBCプログラミング特有の重荷に頼ることなく、たった数行で、リスト3と同じ振る舞いをコード化できたのです。素晴らしいで

しょう。そしてまた、非常に容易でもあるのです。では、具体的にどのようにしたのかを説明しましょう。

単純なクエリーを行う

リスト4の最初の行で、対象とするデータベースへの接続に使われる、Groovyの`Sql`クラスのインスタンスを作りました。この場合は、私のマシンで実行しているMySQLデータベースを指す`Sql`インスタンスを作っています。ここまでは、ごく基本的、ですよね？ 問題は次の部分です。イテレーターとクロージャーがその力を、ワン・ツー・パンチで見せてくれるのです。

`eachRow`メソッドを、渡されたクエリーの結果に対するイテレーターと考えてください。その下で、JDBC`ResultSet`オブジェクトが戻され、その内容が`for`ループに渡されるのを想像してください。結果として、私が渡したクロージャーが、それぞれの繰り返しに対して実行されるのです。データベースにある`word`テーブルに行が3つだけあるとすると、クロージャーは3回実行され、`word_id`と`spelling`、そして`part_of_speech`を出力します。

このコードは、名前つきの変数`row`を式から無くすことで、そしてGroovyの持つ暗黙的変数、itを使うことによって、さらに単純化することができます（`it`は、イテレーターのインスタンスです）。これを行ったとすると、前のコードは、リスト5のように書くことができます。

リスト5. GroovySqlにおける、Groovyの`it`変数

```
import groovy.sql.Sql
class GroovySqlExample1{
  static void main(args) {
    sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
           "words", "org.gjt.mm.mysql.Driver")
    sql.eachRow("select * from word"){ println it.spelling +  " ${it.part_of_speech}"}
  }
}
```

このコードでは、`row`変数を使わず、代わりに`it`を使うことができました。さらに、`${it.part_of_speech}`で行ったように、`String`ステートメントの中で`it`変数を参照することもできました。

より複雑なクエリーを行う

前の例は非常に単純でしたが、`insert`や`update`、`delete`クエリーなどのような、より複雑なデータ操作クエリーに関しても、GroovySqlは同じように良くできているのです。これらに対しては、皆さんは必ずしもイテレーターは使いたいとは思わないかも知れません。そのため、Groovyの`Sql`オブジェクトは代わりに、`execute`や`executeUpdate`メソッドを提供しています。こうしたメソッドは、`execute`や`executeUpdate`メソッドも持つ、通常のJDBC`statement`クラスの名残です。

リスト6を見ると、変数置換を使う単純な`insert`があることが分かります（ここでも`${}`構文を使っています）。このコードは、単純に新しい行を`word`テーブルに挿入します。

リスト6. GroovySqlでのInsert

```
wid = 999
spelling = "Nefarious"
pospeech = "Adjective"
sql.execute("insert into word (word_id, spelling, part_of_speech)
  values (${wid}, ${spelling}, ${pospeech})")
```

Groovyにはまた、クエリーの中にある任意の`?`要素に対応する値のリストを扱う、オーバーライド版の`execute`メソッドもあります。リスト7では単純に、`word`テーブル中の特定な行をクエリーしています。この裏では、GroovySqlが、通常のJava言語での`java.sql.PreparedStatement`のインスタンスを作っています。

リスト7. GroovySqlでのPreparedStatement

```
val = sql.execute("select * from word where word_id = ?", [5])
```

Updatesも、`executeUpdate`メソッドを利用するという点では、ほとんど同じです。ただしリスト8では、`executeUpdate`メソッドが、値のリストを扱うことにも注意してください（この値は、対応する（クエリー中の）`?`要素と比較され、一致するかをチェックされます）。

リスト8. GroovySqlでのUpdate

```
nid = 5
spelling = "Nefarious"
sql.executeUpdate("update word set word_id = ? where spelling = ?", [nid, spelling])
```

Deleteは、（当然ながら）クエリーの構文が異なることを除けば、基本的にinsertと同じです。これをリスト9に示します。

リスト9. GroovySqlでのDelete

```
sql.execute("delete from word where word_id = ?" , [5])
```

データ操作を単純化する

JDBCプログラミングを単純化しようとするAPIやユーティリティーであれば、非常に堅牢なデータ操作機能を持っているはずです。そこでこのセクションでは、そのうちの3つを紹介します。

DataSets

単純さを基礎に構築されたGroovySqlは、`DataSet`型の概念をサポートしています。`DataSet`型は基本的に、データベース・テーブルのオブジェクト表現です。`DataSet`を使うと、行に対して繰り返しを行ったり、新しい行を追加したりすることができます。確かに、テーブルに共通なデータの集合を表現する方法として、DataSetを使うのは便利です。

ただし、現在のGroovySql`DataSet`型の欠点は、関係(relationship)を表現せず、単にデータベース・テーブルとの一対一マッピングでしかない、ということです。リスト10では、`word`テーブルから`DataSet`を作っています。

リスト10. GroovySqlでのDataset

```
import groovy.sql.Sql
class GroovyDatasetsExample1{
  static void main(args) {
    sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "words",
          "words", "org.gjt.mm.mysql.Driver")
    words = sql.dataSet("word")
    words.each{ word |
     println word.word_id + " " + word.spelling
    }
    words.add(word_id:"9999", spelling:"clerisy", part_of_speech:"Noun")
  }
}
```

ご覧の通り、テーブルの内容に対して`each`メソッドで繰り返しを行い、対象データを表現する`map`を扱う`add`メソッドで新しい行を追加することが、GroovySqlの`DataSet`型によって容易にできるようになります。

ストアード・プロシージャーと負のインデックスを使う

ストアード・プロシージャー・コールと、負のインデックス(negative indexing)は、データ処理の重要な要素です。GroovySqlでは、ストアード・プロシージャー・コールが、`Sql`クラスに対して`call`メソッドを使うのと同じくらいに簡単です。また負のインデックスに関しては、GroovySqlは`ResultSet`型を強化しており、これはGroovyでのcollectionsとほとんど同じように動作します。例えば、結果セットの最後のアイテムを取得したいのであれば、リスト11に示すような方法で行うことができます。

リスト11. GroovySqlでの負のインデックス

```
sql.eachRow("select * from word"){ grs |
  println "-1  = " + grs.getAt(-1) //prints spelling
  println "2  = " + grs.getAt(2) //prints spelling
}
```

リスト11を見ると分かる通り、結果セットの最後の要素を取得するには、単に -1としてインデックスするだけでよいのです。場合によっては、同じ要素に対して、2というインデックスでアクセスすることもできます。

こうした例もまた非常に基本的なものですが、GroovySqlの持つ強力さの一端が、よく分かると思います。では次に、これまで説明した機能のすべてを使った実世界の例を説明して、今回の記事を終えたいと思います。

単純なレポート・アプリケーションを書く

レポート・アプリケーションは通常、データベースから情報を引き出します。現在のWebセールスに関するレポート・アプリケーションをセールス・チーム向けに書いたり、システムのある面、例えばデータベースのパフォーマンスを開発チームに毎日チェックさせたりする、ということがビジネス環境ではよくあります。

単純にするために、あなたが今ちょうど、エンタープライズWebアプリケーションを展開し終わった、としましょう。あなたはこれまで充分なユニット・テストを（Groovyで）書いたので、

当然ながら、そのアプリケーションは問題なく動作しています。ただしそれでも、調整のために、データベースの状態に関するレポートを生成する必要があります。パフォーマンスの問題を予見し、対応できるようにするために、顧客がアプリケーションをどのように使っているかを知りたいのです。

通常は時間の制約から、そうしたアプリケーションに適用できる華やかな特別機能は限定されがちです。ところが新たに手にしたGroovySqlに関する知識によって、このアプリケーションを簡単に仕上げ、あなたが欲しいと思っていた特別機能を追加するだけ時間の余裕ができるのです。

詳細

この場合、対象とするデータベースは、正にクエリーでステータス情報を得るという概念をサポートしている、MySQLです。あなたの関心対象となるステータス情報は、次のようなものです。

- 実行時間 (Uptime)
- 処理されるクエリーの合計
- `insert`や`update`、`select`など、特別なクエリーの割合

GroovySqlを使うと、MySQLデータベースからこうした情報を取得するのは、簡単すぎるほど簡単です。これは開発チーム用に構築するので、恐らくあなたは単純なコマンド・ライン・レポートから始めるかも知れませんが、後でそのレポートをWebで使えるようにするのは簡単です。このレポート例のユース・ケースは次のようなものでしょう。

1.	私達のアプリケーションの、動作中のデータベースに接続する
2.	`show status`クエリーを発行し、下記をキャプチャーする
	a. 実行時間
	b. クエリーの合計
	c.`insert`の合計
	d.`update`の合計
	e.`select`の合計
3.	こうしたデータ・ポイントから、下記を計算する
	a. 毎分のクエリー
	b.`insert`クエリーの合計の割合
	c.`update`クエリーの合計の割合
	d.`select`クエリーの合計の割合

リスト12が最終結果、つまり必要なデータベース統計をレポートするアプリケーションです。コードの最初の方で実稼働のデータベースに接続します。それに続いて一連の`show status`クエリーがあり、これを使って毎分のクエリーを計算し、タイプ毎に分類します。`uptime`のような変数が、定義されるとすぐに使えるようになることに注意してください。

リスト12. GroovySqlによるデータベースのステータス・レポート

```
import groovy.sql.Sql
```

```
class DBStatusReport{
  static void main(args) {
     sql = Sql.newInstance("jdbc:mysql://yourserver.anywhere/tiger", "scott",
        "tiger", "org.gjt.mm.mysql.Driver")
     sql.eachRow("show status"){ status |
        if(status.variable_name == "Uptime"){
           uptime =  status[1]
        }else if (status.variable_name == "Questions"){
           questions =  status[1]
        }
     }
     println "Uptime for Database: " + uptime
     println "Number of Queries: " + questions
     println "Queries per Minute = " + Integer.valueOf(questions) / Integer.valueOf(uptime)
     sql.eachRow("show status like 'Com_%'"){ status |
        if(status.variable_name == "Com_insert"){
           insertnum =  Integer.valueOf(status[1])
        }else if (status.variable_name == "Com_select"){
           selectnum =  Integer.valueOf(status[1])
        }else if (status.variable_name == "Com_update"){
           updatenum =  Integer.valueOf(status[1])
       }
    }
    println "% Queries Inserts = " + 100 * (insertnum / Integer.valueOf(uptime))
    println "% Queries Selects = " + 100 * (selectnum / Integer.valueOf(uptime))
    println "% Queries Updates = " + 100 * (updatenum / Integer.valueOf(uptime))
    }
}
```

まとめ

今回の実用的なGroovyでは、GroovySqlを使うことによって、いかにJDBCプログラミングが単純になるかを学びました。この便利なAPIによってクロージャーやイテレーターをGroovyの緩やかな構文を組み合わすことができ、Javaプラットフォーム上でのデータベース開発が迅速に進められるようになります。何よりもGroovySqlで強力なのは、リソース管理の重荷を開発者から取り去り、下にあるGroovyのフレームワークに負わせていることです。これによって開発者は、クエリーやその結果といった、より重要なことに集中することができます。これを、私の口先だけのことだと思わないでください。あなたが今度JDBCの骨折り仕事に関わるように言われたら、ちょっとしたGroovySql魔術を試してみてください。

次回の実用的なGroovyでは、Groovyのテンプレート・フレームワークに関する様々な話題を取り上げます。読んで頂ければ分かりますが、この賢いフレームワークを使うことで、アプリケーションのビュー・コンポーネントが簡単に作れるようになるのです。

ダウンロード可能なリソース

内容	ファイル名	サイズ
Sample code	j-pg01115.zip	4KB

関連トピック

- Andy Gloverによる実用的なGroovyの全記事を読んでください。
- Andrew著による「alt.lang.jre: Feeling Groovy」（developerWorks, 2004年8月）は、Groovyを紹介したシリーズです。
- developerWorksのJava technologyゾーンには、Javaプログラミングのあらゆる面に関する記事が豊富に用意されています。
- developerWorksが提供する、Javaに焦点を絞った無料のチュートリアル・リストがJava technology zone tutorials pageにありますので、これもご覧ください。