Grails をマスターする: Grails とレガシー・データベース

規則にはまらないデータベース・スキーマでも、Convention over Configuration の原則を貫くことができるのか？

Scott Davis
Editor in Chief
AboutGroovy.com

2008年 7月 15日

連載「Grails をマスターする」で今回 Scott Davis が取り上げるのは、Grails の命名規則に当てはまらないデータベース・テーブルを Grails で使用するためのさまざまな手段です。Grails では、レガシー・データベースにすでにマッピングされている Java™ クラスがあれば、レガシー・データベースをそのまま変更せずに使用することができます。この記事では、レガシー Java クラスと併せて Hibernate HBM ファイル、そして Enterprise JavaBeans 3 アノテーションを使用する例をそれぞれ紹介します。

このシリーズの他の記事を見る

GORM (Grails Object Relational Mapping) API は、Grails Web フレームワークで最も重要な部分の 1 つです。この連載の記事「GORM: おかしな名前の真面目な技術」では、単純な 1 対多の関係を始めとする GORM の基礎を紹介しました。その後の記事「Ajax をほんの少し加えた多対多の関係」では、GORM を使ってさらに高度なクラス間の関係をモデル化しています。今回は、GORM の「ORM (オブジェクト指向マッピング)」の部分に備わった柔軟性によって、標準の GORM 命名規則に従っていないレガシー・データベースのテーブル名と列名を処理する方法を紹介します。

データのバックアップとリストア

データベースに含まれる既存のデータを処理する際には常に、最新のバックアップを取っておくことが重要です。マーフィーの法則で有名なマーフィーは、私の守護聖人のような存在となっています。彼が言うように、何事にも失敗の可能性があります。そのため、最悪の場合に備えておくのが最善策です。

この連載について

Grails は、Spring や Hibernate などのよく知られた Java 技術に「Convention over Configuration (設定より規約)」といった現代のプラクティスを盛り込んだ最新の Web 開発フレームワークです。Groovy で作成された Grails は既存の Java コードをシームレスに統合するだけでなく、スクリプト言語ならではの柔軟性と動的機能を与えてくれます。Grails を学んだら、Web 開発に対する今までの見方がまったく違ってくるはずです。

データのバックアップ

ターゲットのデータベースを通常のバックアップ・ソフトウェアでバックアップするだけでなく、予備のコピーとしてデータをプレーン・テキストで保持しておくことをお勧めします。そうすれば、同じデータセットでテスト用データベースと開発用データベースを簡単に作成できるだけでなく、データベース・サーバー間でデータを移すのも容易になります (例えば、MySQL から DB2 に移した後、また MySQL に戻すなど)。

今回の記事でも作業の対象となるのは、この連載を通して開発している Trip Planner アプリケーションです。リスト 1 に、`airport` テーブルのレコードをバックアップする backupAirports.groovy という名前の Groovy スクリプトを記載します。3 つの文と 20 行にも満たないコードからなるこのスクリプトが、データベースに接続し、テーブルのあらゆる行を選択し、データを XML としてエクスポートします。

リスト 1. backupAirports.groovy

```
sql = groovy.sql.Sql.newInstance(
   "jdbc:mysql://localhost/trip?autoReconnect=true",
   "grails",
   "server",
   "com.mysql.jdbc.Driver")

x = new groovy.xml.MarkupBuilder()

x.airports{
  sql.eachRow("select * from airport order by id"){ row ->
    airport(id:row.id){
      version(row.version)
      name(row.name)
      city(row.city)
      state(row.state)
      country(row.country)
      iata(row.iata)
      lat(row.lat)
      lng(row.lng)
    }
  }
}
```

リスト 1 の最初の文は、新規 `groovy.sql.Sql` オブジェクトを作成します。このオブジェクトが、`Connection`、`Statement`、`ResultSet` をはじめとする一連の標準 JDBC クラスを覆う薄い Groovy ファサードとなります。`newInstance` ファクトリー・メソッドの 4 つの引数 (JDBC 接続ストリング、ユーザー名、パスワード、そして JDBC ドライバー) はおそらくお馴染みのものだと思います (これらの値は、grails-app/conf/DataSource.groovy にある値と同じです)。

2 番目の文が作成するのは、`groovy.xml.MarkupBuilder` です。このクラスによって、XML 文書をオンザフライで作成できるようになります。

最後の文 (`x.airports` で始まる文) は、XML ツリーを作成します。この XML 文書のルート要素は `airports` です。データベース内のそれぞれの行には、`id` 属性が設定された `airport` 要素が作成されることになります。`airport` 要素のなかには、`version`、`name`、`city` 要素がネストされます (Groovy の `Sql` および `MarkupBuilder` の使用方法についての詳細は、「参考文献」を参照してください)。

リスト 2 に、このスクリプトを実行した結果の XML を記載します。

リスト 2. バックアップ・スクリプトによる XML 出力

```
<airports>
  <airport id='1'>
    <version>2</version>
    <name>Denver International Airport</name>
    <city>Denver</city>
    <state>CO</state>
    <country>US</country>
    <iata>den</iata>
    <lat>39.8583188</lat>
    <lng>-104.6674674</lng>
  </airport>
  <airport id='2'>...</airport>
  <airport id='3'>...</airport>
</airports>
```

バックアップ・スクリプトでは、主キーの列でソートした順にレコードを取り出すことが重要です。このデータをリストアするときに、外部キーの値が同じく対応することを確実にするためには値が同じ順序で挿入されなければなりません (この点については、次のセクションで詳しく説明します)。

このスクリプトは Grails フレームワークからは完全に独立していることに注意してください。スクリプトが機能するには、システムに Groovy がインストールされていなければなりません (ダウンロードおよびインストール手順については、「参考文献」を参照)。さらに、クラスパスに JDBC ドライバー JAR が指定されている必要もあります。JAR は、スクリプトを実行するときに指定できます。UNIX® では、以下のとおりに入力してください。

```
groovy -classpath /path/to/mysql.jar:. backupAirports.groovy
```

もちろん、Windows® ではファイル・パスと JAR 区切り文字が異なります。Windows の場合には、以下のとおりに入力します。

```
groovy -classpath c:\path\to\mysql.jar;. backupAirports.groovy
```

私は MySQL を頻繁に使用するので、ホーム・ディレクトリー (UNIX では /Users/sdavis、Windows では c:\Documents and Settings\sdavis) の .groovy/lib ディレクトリーに JAR のコピーを常備しています。このディレクトリーに配置された JAR は、コマンドラインから Groovy スクリプトを実行すると自動的にクラスパスに組み込まれます。

リスト 1 のスクリプトは、出力を画面に書き出します。このデータをファイルに保存する場合は、スクリプトの実行時に出力をリダイレクトしてください。

```
groovy backupAirports.groovy > airports.xml
```

データのリストア

データベースからデータを取得する作業は、全体の作業の半分でしかありません。データを戻す作業も同じく重要です。リスト 3 に記載する restoreAirports.groovy スクリプトは、Groovy の `XmlParser` によって XML を読み込み、SQL `insert` 文を作成し、この文をGroovy の SQL オブジェ

クトを使って実行します (`XmlParser` についての詳しい情報は、「参考文献」を参照してください)。

リスト 3. XML からデータベース・レコードをリストアする Groovy スクリプト

```
if(args.size()){
   f = new File(args[0])
   println f

   sql = groovy.sql.Sql.newInstance(
      "jdbc:mysql://localhost/aboutgroovy?autoReconnect=true",
      "grails",
      "server",
      "com.mysql.jdbc.Driver")

   items = new groovy.util.XmlParser().parse(f)
   items.item.each{item ->
     println "${item.@id} -- ${item.title.text()}"
      sql.execute(
         "insert into item (version, title, short_description, description,
                 url, type, date_posted, posted_by) values(?,?,?,?,?,?,?,?)",
         [0, item.title.text(), item.shortDescription.text(), item.description.text(),
             item.url.text(), item.type.text(), item.datePosted.text(),
             item.postedBy.text()]
         )
   }
}
else{
   println "USAGE: itemsRestore [filename]"
}
```

上記のスクリプトを実行するには、以下のとおりに入力します。

```
groovy restoreAirports.groovy airports.xml
```

テーブル間の関係が機能するには、1 対多の関係のうち、1 の側の主キー・フィールドが多の側の外部キー・フィールドと対応しなければなりません。例えば、`airport` テーブルの `id` 列に格納された値は、`flight` テーブルの `arrival_airline_id` 列に格納された値と同じである必要があります。

USGS データの変換

USGS は空港データをシェープファイルとして提供しています。シェープファイルは、地理データ交換用として定評のあるファイル・フォーマットです。シェープファイルを構成するのは最小 3つのファイルで、そのうち .shp ファイルには地理データが含まれます (この例では、各空港の緯度と経度による地点)。.shx ファイルは空間インデックス情報を持つファイルです。そして .dbf ファイルはご想像のとおり、昔ながらの dBase ファイルで、ここに空間データ以外のすべてのデータが含まれます (この例では、空港の名前や IATA コードなど)。

シェープファイルをこの記事で使用する GML (Geography Markup Language) ファイルに変換するために使用したのは、GDAL (Geospatial Data Abstraction Library) です。GDAL は、地理データを操作するためのオープンソースのコマンドライン・ツールがセットになったものです。具体的には、以下のコマンドを使用しました。

```
ogr2ogr -f "GML" airports.xml airprtx020.shp
```

GDAL では、CSV (Comma-Separated Value)、GeoJSON、KML (Keyhole Markup Language) など、他にもさまざまなフォーマットにデータを変換できます。

オートナンバー型の `id` フィールドを確実に同じ値にリストアするため、必ずすべてのテーブルを破棄してからリストアを行ってください。こうすれば、Grails が次の起動時にテーブルを再作成する際に、オートナンバーによる番号を 0 から付け始めることができます。

これで、空港データを (おそらく他のテーブルのデータも) 安全にバックアップできたので、新しい「レガシー」データでの実験に取り掛かれます。新しいレガシー・データという言葉に混乱するかもしれませんが、次のセクションを読めば明らかになるはずです。

新しい空港データのインポート

米国地質調査所 (USGS) では、米国内の全空港のリストを公開しています。この包括的リストには、IATA コードや緯度と経度による位置も記載されています (「参考文献」を参照)当然のことながら、USGS のフィールドは、このサンプルで使用している `Airport` クラスには対応していません。Grails クラスを変更して USGS テーブルの名前に合わせるという方法も採れますが、それにはアプリケーションを大々的に書き直す必要が出てきます。そこで、既存の `Airport` クラスを新しい異なるテーブル・スキーマに裏でシームレスにマッピングするための手法をいくつか探っていくことにします。

まず始めに必要なのは、USGS の「レガシー」データをデータベースにインポートすることです。リスト 4 の createUsgsAirports.groovy スクリプトを実行して新しいテーブルを作成してください (このスクリプトでは、MySQL を使用していることを前提とします。テーブルを作成するための構文はデータベースによって多少異なるので、他のデータベースを使用している場合はスクリプトを調整する必要があるかもしれません)。

リスト 4. USGS の空港テーブル (usgs_airports) の作成

```
sql = groovy.sql.Sql.newInstance(
   "jdbc:mysql://localhost/trip?autoReconnect=true",
   "grails",
   "server",
   "com.mysql.jdbc.Driver")

ddl = """
CREATE TABLE usgs_airports (
  airport_id bigint(20) not null,
  locid varchar(4),
  feature varchar(80),
  airport_name varchar(80),
  state varchar(2),
  county varchar(50),
  latitude varchar(30),
  longitude varchar(30),
  primary key(airport_id)
);
"""

sql.execute(ddl)
```

リスト 5 のusgs-airports.xml を見てください。これは、GML フォーマットの一例です。この XML は、バックアップ・スクリプトによって作成されたリスト 2 の単純な XML と比べると少し複雑です。すべての要素は名前空間に含まれ、要素のネスト・レベルはさらに深くなっています。

リスト 5. GML による USGS の空港

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<ogr:FeatureCollection
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://ogr.maptools.org/airports.xsd"
     xmlns:ogr="http://ogr.maptools.org/"
     xmlns:gml="http://www.opengis.net/gml">

  <gml:featureMember>
    <ogr:airprtx020 fid="F0">
      <ogr:geometryProperty>
        <gml:Point>
          <gml:coordinates>-156.042831420898438,19.73573112487793</gml:coordinates>
        </gml:Point>
      </ogr:geometryProperty>
      <ogr:AREA>0.000</ogr:AREA>
      <ogr:PERIMETER>0.000</ogr:PERIMETER>
      <ogr:AIRPRTX020>1</ogr:AIRPRTX020>
      <ogr:LOCID>KOA</ogr:LOCID>
      <ogr:FEATURE>Airport</ogr:FEATURE>
      <ogr:NAME>Kona International At Keahole</ogr:NAME>
      <ogr:TOT_ENP>1271744</ogr:TOT_ENP>
      <ogr:STATE>HI</ogr:STATE>
      <ogr:COUNTY>Hawaii County</ogr:COUNTY>
      <ogr:FIPS>15001</ogr:FIPS>
      <ogr:STATE_FIPS>15</ogr:STATE_FIPS>
    </ogr:airprtx020>
  </gml:featureMember>

  <gml:featureMember>...</gml:featureMember>
  <gml:featureMember>...</gml:featureMember>
</ogr:FeatureCollection>
```

今度は restoreUsgsAirports.groovy スクリプトを作成します (リスト 6 を参照)。名前空間が指定された要素を取得するには、宣言する必要がある `groovy.xml.Namespace` 変数がいくつかあります。この場合、前の restoreAirport.groovy スクリプト (リスト 3) で使用した単純なドット付き表記は使わずに、名前空間が指定された要素を大括弧で囲んでください。

リスト 6. USGS 空港データのデータベースへのリストア

```
if(args.size()){
  f = new File(args[0])
  println f

  sql = groovy.sql.Sql.newInstance(
     "jdbc:mysql://localhost/trip?autoReconnect=true",
     "grails",
     "server",
     "com.mysql.jdbc.Driver")

  FeatureCollection = new groovy.util.XmlParser().parse(f)
  ogr = new groovy.xml.Namespace("http://ogr.maptools.org/")
  gml = new groovy.xml.Namespace("http://www.opengis.net/gml")

  FeatureCollection[gml.featureMember][ogr.airprtx020].each{airprtx020 ->
   println "${airprtx020[ogr.LOCID].text()} -- ${airprtx020[ogr.NAME].text()}"
   points = airprtx020[ogr.geometryProperty][gml.Point][gml.coordinates].text().split(",")

     sql.execute(
        "insert into usgs_airports (airport_id, locid, feature, airport_name, state,
        county, latitude, longitude) values(?,?,?,?,?,?,?,?)",
        [airprtx020[ogr.AIRPRTX020].text(),
        airprtx020[ogr.LOCID].text(),
        airprtx020[ogr.FEATURE].text(),
        airprtx020[ogr.NAME].text(),
        airprtx020[ogr.STATE].text(),
```

```
        airprtx020[ogr.COUNTY].text(),
        points[1],
        points[0]]
        )
  }
}
else{
   println "USAGE: restoreAirports [filename]"
}
```

コマンド・プロンプトに対し、以下のコマンドを入力して、usgs_airports.xml ファイルのデータを新しく作成したテーブルに挿入します。

```
groovy restoreUsgsAirports.groovy
usgs-airports.xml
```

データが正常に挿入されたことを確認するには、コマンドラインから MySQL にログインし、データが存在することを確かめます (リスト 7 を参照)。

リスト 7. データベース内の USGS 空港データの確認

```
$ mysql --user=grails -p --database=trip

mysql> desc usgs_airports;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| airport_id   | bigint(20)  | NO   | PRI |         |       |
| locid        | varchar(4)  | YES  |     | NULL    |       |
| feature      | varchar(80) | YES  |     | NULL    |       |
| airport_name | varchar(80) | YES  |     | NULL    |       |
| state        | varchar(2)  | YES  |     | NULL    |       |
| county       | varchar(50) | YES  |     | NULL    |       |
| latitude     | varchar(30) | YES  |     | NULL    |       |
| longitude    | varchar(30) | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
8 rows in set (0.01 sec)

mysql> select count(*) from usgs_airports;
+----------+
| count(*) |
+----------+
|      901 |
+----------+
1 row in set (0.44 sec)

mysql> select * from usgs_airports limit 1\G
*************************** 1. row ***************************
  airport_id: 1
       locid: KOA
     feature: Airport
airport_name: Kona International At Keahole
       state: HI
      county: Hawaii County
    latitude: 19.73573112487793
   longitude: -156.042831420898438
```

`dbCreate` を無効に設定する

レガシー・テーブルを用意できたら、締めくくりの作業として grails-app/conf/DataSource.groovy 内の dbCreate 変数を無効にします。記事「GORM: おかしな名前の真面目な技術」を思い出してく

ださい。GORM はこの変数の値に従った動作をします。その動作とは、該当するテーブルが存在しなければ裏でそのテーブルを作成し、すでにテーブルがある場合には Grails ドメイン・クラスと一致するように既存のテーブルを変更する動作です。そのため、レガシー・テーブルを扱っている場合には、他のアプリケーションが期待している可能性のあるスキーマを GORM が台無しにしないように、この機能を無効にしなければなりません。

`dbCreate` 変数を特定のテーブルには有効にして、その他のテーブルには無効にすることができれば理想的ですが、残念ながら、これは全体的な「オール・オア・ナッシング」の設定です。新しいテーブルとレガシー・テーブルが混在している場合、私は GORM に新しいテーブルを作成させてから、`dbCreate` 変数を無効に設定して既存のレガシー・テーブルをインポートできるようにします。このような場合に、バックアップしてリストアするという優れたストラテジーが重要になってくることがおわかりでしょう。

静的 mapping ブロック

ドメイン・クラスをレガシー・テーブルにマッピングするためのストラテジーとして最初に説明するのは、静的 `mapping` ブロックを使用したストラテジーです。私はほとんどの場合、最も Grails 的だという理由で、このストラテジーを使っています。私の場合、ドメイン・クラスに静的 `constraints` ブロックを追加するのには慣れているため、静的 `mapping` ブロックを追加するという方法は、フレームワークの他の部分と一貫性があるように思えます。

まず、grails-app/domain/Airport.groovy ファイルを grails-app/domain/AirportMapping.groovy にコピーしてください。この名前は便宜上のものですが、これから使用する 3 つのクラスはすべて同じテーブルにマッピングされるので、それぞれのクラスの名前を固有のものにする必要があります (実際のアプリケーションでは、このようにする必要はないはずです)。

都市 (city) と国 (country) のフィールドは新しいテーブルには存在しないため、コメント・アウトします。これらのフィールドは `constraints` ブロックからも削除してください。次に、静的 `mapping` ブロックを追加して、Grails での名前をデータベースでの名前にリンクさせます (リスト 8 を参照)。

リスト 8. AirportMapping.groovy

```
class AirportMapping{
  static constraints = {
    name()
    iata(maxSize:3)
    state(maxSize:2)
    lat()
    lng()
  }

  static mapping = {
    table "usgs_airports"
    version false
    columns {
      id column: "airport_id"
      name column: "airport_name"
      iata column: "locid"
      state column: "state"
      lat column: "latitude"
      lng column: "longitude"
    }
  }
```

```
  String name
  String iata
  //String city
  String state
  //String country = "US"
  String lat
  String lng

  String toString(){
    "${iata} - ${name}"
  }
}
```

mapping ブロックの最初の文は AirportMapping クラスを usgs_airports テーブルにリンクさせ、次の文が、このテーブルには version 列がないことを Grails に伝えます (GORM は通常、オプティミスティック・ロックを容易にするためにこの列を作成します)。最後に、columns ブロックが Grails での名前をデータベースでの名前にマッピングします。

注目すべき点は、このマッピング手法を使用すると、テーブル内の特定のフィールドを無視できることです。この例では、feature 列と county 列はドメイン・クラスで表されていません。それとは逆に、テーブルに格納されないドメイン・クラスのフィールドを設定するには、静的 transients 行を追加します。この行は 1 対多の関係で使用する belongsTo 変数のようなものです。例えば、Airport クラスにテーブルに格納する必要のない 2 つのフィールドがあるとします。その場合のコードは、以下のようになります。

```
static transients = ["tempField1", "tempField2"]
```

ここに記載した mapping ブロックは、この手法を使って実現できる内容についてほんの少し触れただけに過ぎません。詳細は、「参考文献」を参照してください。

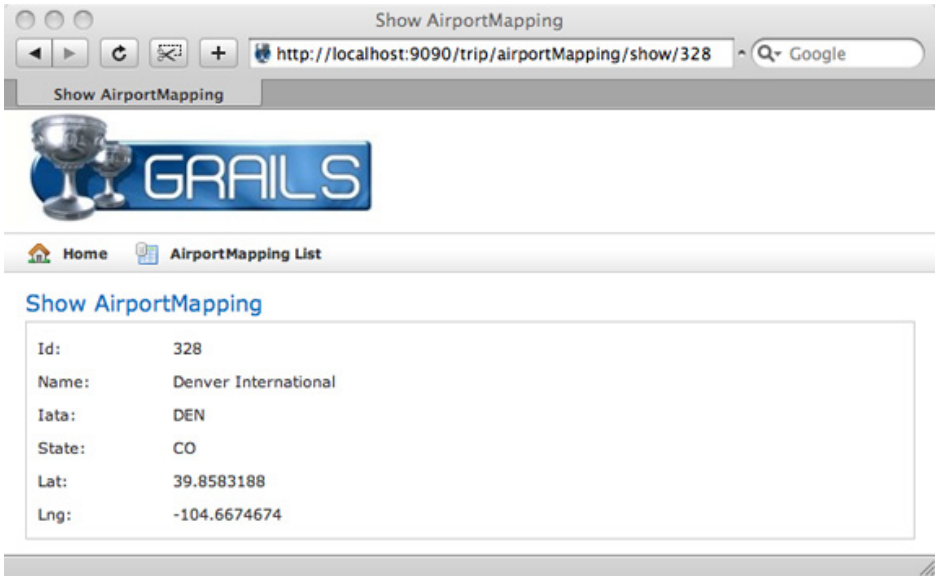
レガシー・テーブルを読み取り専用にする

grails generate-all AirportMapping を実行してコントローラーと GSP ビューを作成します。このテーブルは基本的にルックアップ・テーブルなので、grails-app/controllers/AirportMappingController.groovy 内には list および show クロージャーだけを残します。delete、edit、update、create、save は削除してください (allowedMethods 変数から delete、edit、および save を削除することも忘れないでください)。行全体を削除するのでも、空の大括弧のセットを残しておくのでも構いません。

ビューを読み取り専用にするには、ちょっとした変更をいくつか加えるだけで済みます。まず、grails-app/views/airportMapping/list.gsp の先頭にある New AirportMapping リンクを削除します。grails-app/views/airportMapping/show.gsp でも同様にこのリンクを削除します。そして最後に、show.gsp の終わりにある edit ボタンと delete ボタンを削除します。

mapping ブロックが機能することを確認するには、grails run-app を実行してください。その結果、図 1 に示すページが表示されるはずです。

図 1. `mapping` ブロックの動作確認



Hibernate マッピング・ファイルによってレガシー Java クラスを使用する方法

`mapping` ブロックについて理解できたところで、今度はもう一歩踏み込んだ内容に進みましょう。レガシー・テーブルがあるのなら、レガシー Java クラスがあることも想像に難くありません。ここから紹介する 2 つのマッピング手法では、既存の Java コードを既存のテーブルのデータに適合させて使う必要があることを前提とします。

Java 1.5 がアノテーションを導入する以前は、Hibernate ユーザーは HBM ファイルという名前の XML マッピング・ファイルを作成していました。GORM は Hibernate を覆う薄い Groovy ファサードであることを思い出してください。つまり、当然のことながら、かつての Hibernate の芸当はすべて引き続き有効なのです。

まずは、レガシー Java ソース・ファイルを src/java にコピーします。パッケージを使用している場合は、パッケージ名ごとに 1 つのディレクトリーを作成します。例えば、リスト 9 に記載する AirportHbm.java ファイルは `org.davisworld.trip` パッケージに含まれるため、このファイルの完全パスは src/java/org/davisworld/trip/AirportHbm.java となるはずです。

リスト 9. AirportHbm.java

```
package org.davisworld.trip;

public class AirportHbm {
  private long id;
  private String name;
  private String iata;
  private String state;
  private String lat;
  private String lng;

  public long getId() {
    return id;
  }

  public void setId(long id) {
    this.id = id;
  }

  // all of the other getters/setters go here
}
```

Java ファイルを所定の場所に配置したら、今後はその隣に AirportHbmConstraints.groovy という名前の「シャドー」ファイルを作成します (リスト 10 を参照)。このファイルに、通常はドメイン・クラスに含まれることになる静的 `constraints` ブロックを配置することができます。このファイルは必ず Java クラスと同じパッケージに含まれるようにしてください。

リスト 10. AirportHbmConstraints.groovy

```
package org.davisworld.trip

static constraints = {
  name()
  iata(maxSize:3)
  state(maxSize:2)
  lat()
  lng()
}
```

src ディレクトリーの下に置かれたファイルは、アプリケーションの実行時、またはデプロイメント用の WAR ファイルを作成するときにコンパイルされます。Java コードがすでにコンパイルされている場合は、そのまま JAR ファイルにまとめて lib ディレクトリーに配置するのでも構いません。

次は、コントローラーをセットアップします。Convention over Configuration の原則に従って、コントローラーには AirportHbmController.groovy という名前を指定します。Java クラスはパッケージに含まれていることから、同じパッケージにこのコントローラーも含めることも、あるいは Java クラスをファイルの先頭にインポートすることもできます。ここでは、インポートするという方法を採ります (リスト 11 を参照)。

リスト 11. AirportHbmController.groovy

```
import org.davisworld.trip.AirportHbm

class AirportHbmController {
  def scaffold = AirportHbm
}
```

次に、既存の HBM ファイルを grails-app/conf/hibernate にコピーします。hibernate.cfg.xml という 1 つのファイル (リスト 12 を参照) に、各クラスに使用するそれぞれのマッピング・ファイルを指定します。この例では、AirportHbm.hbm.xml ファイルのエントリーが必要です。

リスト 12. hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <mapping resource="AirportHbm.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

それぞれのクラスには、専用の HBM ファイルがなければなりません。このファイルは、前に使用した静的 `mapping` ブロックに相当する XML です。リスト 13 に、AirportHbm.hbm.xml を記載します。

リスト 13. AirportHbm.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.davisworld.trip.AirportHbm" table="usgs_airports">
        <id name="id" column="airport_id">
            <generator class="native"/>
         </id>
        <property name="name" type="java.lang.String">
            <column name="airport_name" not-null="true" />
        </property>
        <property name="iata" type="java.lang.String">
            <column name="locid" not-null="true" />
        </property>
        <property name="state" />
        <property name="lat" column="latitude" />
        <property name="lng" column="longitude" />
    </class>
</hibernate-mapping>
```

Java クラスへの参照には、完全パッケージ名が指定されていることに注目してください。残りのエントリーは、Java の名前をテーブルの名前にマッピングするものです。`name` および `iata` フィールドのエントリーは長いフォームになっていますが、`state` フィールドは Java コードとテーブルとで共通するので、このエントリーは短くすることができます。最後の 2 つのフィールド、`lat` と `lng` は、ショートカット構文になっています (Hibernate マッピング・ファイルについての詳細は、「参考文献」を参照してください)。

Grails がまだ実行中の場合は、Grails を再始動してください。これで、http://localhost:8080/trip/airportHbm にアクセスすると、Hibernate でマッピングされたデータが表示されるはずです。

UJava クラスで EJB (Enterprise JavaBeans) 3 アノテーションを使用する方法

前述したように、Java 1.5 で Java 言語にはアノテーションが導入されました。アノテーションでは、Java クラスの頭に @ を付けることで、メタデータを直接 Java クラスに追加することができます。2006年12月にリリースされた Groovy 1.0 ではアノテーションといった Java 1.5 言語の機能をサポートしていませんでしたが、1 年後にリリースされた Groovy 1.5 では、すっかり事態は変わっていました。つまり、EJB3 アノテーションを使用した Java ファイルを既存の Grails アプリケーションにも組み込めるようになったのです。

今度は EJB3アノテーション付き Java ファイルを使って、最初から手順を始めます。リスト 14 に記載する AirportAnnotation.java を、src/java/org.davisworld.trip の AirportHbm.java ファイルのすぐ隣に配置してください。

リスト 14. AirportAnnotation.java

```
package org.davisworld.trip;

import javax.persistence.*;

@Entity
@Table(name="usgs_airports")
public class AirportAnnotation {
  private long id;
  private String name;
  private String iata;
  private String state;
  private String lat;
  private String lng;

  @Id
  @Column(name="airport_id", nullable=false)
  public long getId() {
    return id;
  }

  @Column(name="airport_name", nullable=false)
  public String getName() {
    return name;
  }

  @Column(name="locid", nullable=false)
  public String getIata() {
    return iata;
  }

  @Column(name="state", nullable=false)
  public String getState() {
    return state;
  }

  @Column(name="latitude", nullable=false)
  public String getLat() {
    return lat;
  }

  @Column(name="longitude", nullable=false)
  public String getLng() {
    return lng;
  }
```

```
  // The setter methods don't have an annotation on them.
  // They are not shown here, but they should be in the file
  //   if you want to be able to change the values.

}
```

注意する点として、`javax.persistence` パッケージはファイルの先頭でインポートしなければなりません。`@Entity` と `@Table` は、クラス宣言にアノテーションを付けて該当するデータベース・テーブルにマッピングします。残りのアノテーションは、各フィールドのゲッター・メソッドの上にあります。すべてのフィールドには、フィールド名を列名にマッピングする `@Column` アノテーションが必要です。また、主キーにも `@ID` アノテーションを付ける必要があります。

リスト 15 の AirportAnnotationConstraints.groovy ファイルには、リスト 10 に記載した前のサンプル・ファイルと異なる点は 1 つもありません。

リスト 15. AirportAnnotationConstraints.groovy

```
package org.davisworld.trip

static constraints = {
  name()
  iata(maxSize:3)
  state(maxSize:2)
  lat()
  lng()
}
```

リスト 16 の AirportAnnotationController.groovy は、通常の方法で scaffold されています。

リスト 16. AirportAnnotationController.groovy

```
import org.davisworld.trip.AirportAnnotation

class AirportAnnotationController {
  def scaffold = AirportAnnotation
}
```

ここでもまた、hibernate.cfg.xml ファイルが活躍します。しかし今回の構文は少々異なり、このファイルが指すのは HBM ファイルではなく、クラスを直接指しています (リスト 17 を参照)。

リスト 17. hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <mapping resource="AirportHbm.hbm.xml"/>
        <mapping class="org.davisworld.trip.AirportAnnotation"/>
    </session-factory>
</hibernate-configuration>
```

最後に 1 つ、アノテーションを機能させるために必要なステップがあります。Grails はそのままでは、EJB3 アノテーションを検索するように構成されていません。そのため、grails-app/conf/DataSource.groovy の特定のクラスをインポートする必要があります (リスト 18 を参照)。

リスト 18. DataSource.groovy

```
import org.codehaus.groovy.grails.orm.hibernate.cfg.GrailsAnnotationConfiguration
dataSource {
  configClass = GrailsAnnotationConfiguration.class

   pooled = false
   driverClassName = "com.mysql.jdbc.Driver"
   username = "grails"
   password = "server"
}
```

`org.codehaus.groovy.grails.orm.hibernate.cfg.GrailsAnnotationConfiguration` がインポートされて、Spring がこのクラスを `configClass` として `dataSource` ブロックに注入できるようになると、Grails が EJB3 アノテーションをサポートするようになると同時に、HBM ファイルとネイティブ mapping ブロックもサポートするようになります。

最後のステップを忘れると (私が Grails で EJB3 アノテーションを使うときには、ほとんど毎回忘れてしまいますが)、以下のエラー・メッセージを受け取ることになります。

リスト 19. **`configClass`** を DataSource.groovy に注入しないとスローされる例外

```
org.hibernate.MappingException:
An AnnotationConfiguration instance is required to use
<mapping class="org.davisworld.trip.AirportAnnotation"/>
```

まとめ

この記事を読み終わった今、Grails でオブジェクト (Object) をリレーショナル (Relational) データベースにマッピング (Mapping) するのは至って簡単な作業になっているはずです (結局のところ、これが GORM という名前の由来です)。データを難なくバックアップしてリストアできるようになれば、さまざまな方法で、レガシー・データベースの標準とは外れた命名規則に Grails を適合させられるようになります。このタスクを実現するには、最も Grails 的な静的 `mapping` ブロックを使うのが最も簡単な手段です。ただし、レガシー Java クラスがすべにレガシー・データベースにマッピングされている場合には、わざわざ一からやり直す意味はありません。HBM ファイル、あるいはそれより新しい EJB3 アノテーションのどちらを使用するにしても、Grails は今までに行った作業を利用して、開発者が他のタスクに取り掛かれるようにします。

次回の記事では、Grails のイベント・モデルをいろいろと試してみます。アプリケーションのライフサイクルでは、ビルド・スクリプトから個々の Grails 成果物 (ドメイン・クラスやコントローラーなど) に至るまでのあらゆるものが、重要なポイントでイベントをスローします。これらのイベントをキャッチし、カスタムの振る舞いで応答するリスナーをセットアップする方法を学んでください。次回の記事まで、Grails を楽しみながらマスターしてください。

ダウンロード

内容	ファイル名	サイズ
Sample code[1]	j-grails07158.zip	991KB

注

1. コードのダウンロードには、Grails アプリケーション、Groovy スクリプト、そして GML による USGS Airport ファイルが含まれています。

著者について

Scott Davis

Scott Davis は国際的に知られた著者、講演者、そしてソフトウェア開発者です。彼の著書には、『Groovy Recipes: Greasing the Wheels of Java』、『GIS for Web Developers: Adding Where to Your Application』、『The Google Maps API』、『JBoss At Work』などがあります。