Hibernateで継承マッピングを単純化する

クラス階層構造のマップ方法として、実装が容易な3つの戦略を学ぶ

Xavier Coulon
E-business IT Specialist
IBM Business Consulting Services

Christian Brousseau
J2EE Consultant

2004年 12月 14日

Hibernateはオブジェクト・リレーショナルな、マッピングとパーシスタンスのフレームワークであり、イントロスペクションからpolymorphismや継承マッピングまで、様々な高度機能を提供します。ところがクラスの階層構造をリレーショナル・データベース・モデルにマップするのは少し難しいかも知れません。この記事では、複雑なオブジェクト・モデルをリレーショナル・データベース・モデルにマップするために、日々のプログラミング作業の中で使用することのできる3つの戦略について解説します。

概要

Hibernateは純Javaでオブジェクト・リレーショナルな、マッピングとパーシスタンスのフレームワークです。Hibernateを使うことによって、ごく単純なJavaオブジェクトを、XMLコンフィギュレーション・ファイルでリレーショナル・データベース・テーブルにマップできるようになります。Hibernateを使うと、JDBC階層全体がこのフレームワークで管理できるので、開発時間を大いに節約することができます。つまりアプリケーションのデータ・アクセス階層がHibernateの上に位置するようになり、その下にあるデータ・モデルから完全に抽象化できることになります。

Hibernateには、類似のオブジェクト・リレーショナルなマッピング手法（JDO、エンティティbeans、組織内の開発、など）に比較して幾つかの利点があります。無料でオープン・ソースであり、充分成熟しており、広く使用されており、また非常に活発なコミュニティー・フォーラムがあります。

既存のJavaプロジェクトにHibernateを統合するには、次のような手順を踏む必要があります。

1. HibernateのWebサイトから、Hibernateフレームワークの最新リリースをダウンロードします（ 参考文献にリンクがあります ）。
2. 必要なHibernateライブラリー（JARファイル）を、アプリケーションのCLASSPATHにコピーします。

3. Javaオブジェクトをデータベース・テーブルにマップするために使用する、XMLコンフィギュレーション・ファイルを作ります（この手順については、この記事で説明します）。
4. そのXMLコンフィギュレーション・ファイルを、アプリケーションのCLASSPATHにコピーします。

Hibernateフレームワークをサポートするために Javaオブジェクトを何ら変更しなくても良いことに注目してください。例えば何らかの理由から、Javaアプリケーションが使用したデータベース・テーブルを、カラム名をリネームするなどして変更する必要があるとします。一旦テーブルを変更してしまえば、そのJavaアプリケーションを更新するために必要なのは、対応するXMLコンフィギュレーション・ファイルを更新することだけになるのです。どのJavaコードも再コンパイルする必要はありません。

Hibernate Query Language (HQL)

HibernateはHibernate Query Language (HQL) と呼ばれる、SQLと非常によく似たクエリー言語を提供しています。昔ながらのSQLクエリーを好む人達のために、HibernateでもSQLクエリーが使えるようになっています。しかしこの記事の例では、すべてHQLを使います。

HQLを使うのは非常に簡単です。 `SELECT` や `FROM` それに `WHERE` など、SQLでお馴染みのキーワードがHQLにもあります。HQLがSQLと違うのは、HQLではクエリーをデータ・モデルに対して（つまりテーブルやカラムなどに対して）直接書くことはせず、Javaオブジェクトに対して、そのプロパティーやリレーションシップを使って書くのです。

リスト1は基本的な例を示します。このHQLコードは、 `firstName` が「John」である `Individual`（個人）を検索します。

リスト1. 基本的なHQLクエリー

```
SELECT * FROM
eg.hibernate.mapping.dataobject.Individual WHERE
firstName = "John"
```

HQLの構文に関して詳しく知りたい場合には、HibernateのWebサイトにあるHQLの資料を参照してください（ 参考文献 にリンクがあります）。

XMLコンフィギュレーション・ファイル

Hibernateの機能の中心は、XMLコンフィギュレーション・ファイルの内部にあります。こうしたファイルは、アプリケーションのCLASSPATHの中に置かれている必要があります。ここでは、サンプル・コード・パッケージ（ 参考文献 からダウンロードすることができます）のconfigディレクトリーに置きました。

私達が最初に調べるファイルはhibernate.cfg.xmlです。このファイルにはデータ・ソース（データベースURL、スキーマ名、ユーザー名、パスワードなど）に関する情報と、マッピング情報を含んだ他のコンフィギュレーション・ファイルへの参照を含んでいます。

その他のXMLファイルを使うと、Javaクラスをデータベース・テーブルに対してマップすることができます。こうしたファイルに関しては後ほど詳しく見ますが、これらのファイルの名前はClassName.hbm.xmlというパターンに従っていることは重要ですので覚えておいてください。

アプリケーション例

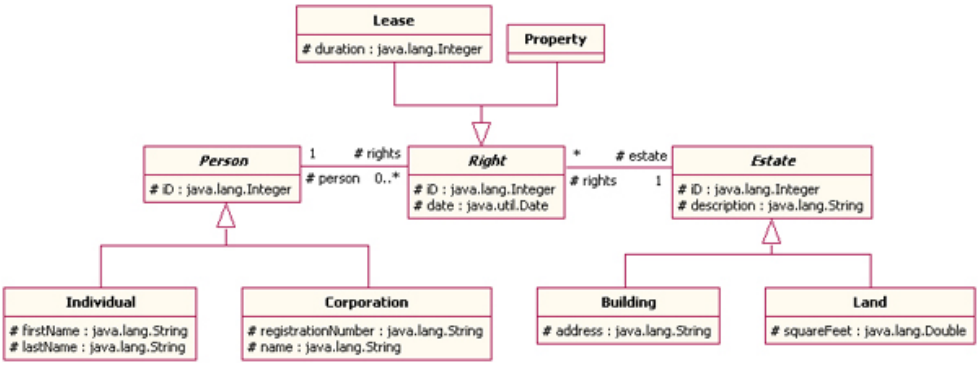
この記事では、Hibernateがどのように動作するのか、また、下でHibernateがオブジェクト・リレーショナルなマッピングを行う、異なった3つの戦略をどのように利用するかを示す基本的な例を調べます。このアプリケーション例は保険会社が使うものとします。この保険会社は、顧客が保険の対象としている財産権の全てに関して、法的な記録を保持する必要があります。このアプリケーションの完全なソースコードは 参考文献 にあります。このコードには、WebアプリケーションやSwingアプリケーションなど、完全機能のアプリケーションを構築できるだけの基本的な機能が用意されています。

この例では、こうしたアプリケーションの古典的な使い方を想定しています。つまり任意の顧客タイプ（個人、法人、政府機関など）に対して、ユーザーが検索の基準を提供します。そうすると、（例え実際の顧客タイプとは異なっていても）その基準に一致する全顧客のリストが表示されるのです。ユーザーはその同じリストから、特定な顧客に関する、より詳細なビューにアクセスすることができます。

このアプリケーションでは、財産権は `Right` クラスで表現されています。 `Right` は `Lease`（リース物件）であるか、あるいは `Property`（財産）です。 `Right` は顧客が所有します。顧客を表現するためには、汎用クラス `Person` を使います。 `Person` は `Individual`（個人）であるか、あるいは `Corporation`（法人）です。当然ですが、保険会社はこうした `Right` が設定されている `Estate`（物件）を知っている必要があります。ご存じの通り `Estate` は非常に一般的な用語です。そこで開発者がもう少し扱いやすいオブジェクトにするために、 `Land`（土地）クラスと `Building`（建物）クラスを使います。

この抽象化から、図1に示すクラス・モデルができます。

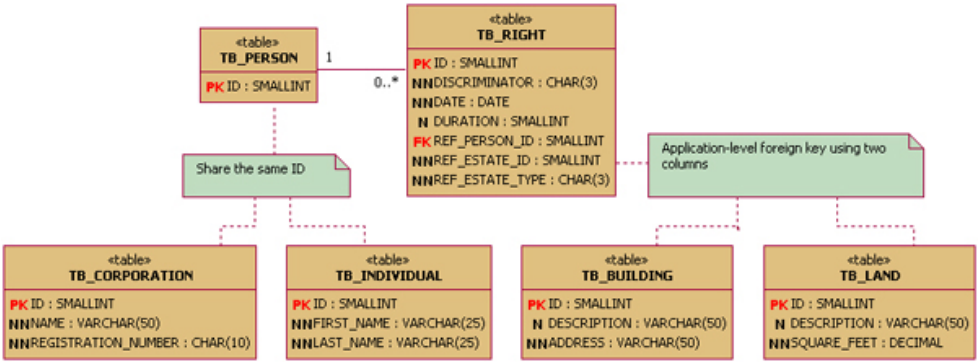
図1. 完全なクラス・モデル



このデータベース・モデルは、この記事で議論する3つの戦略を網羅するように設計されています。 `Right` 階層構造に対しては単一のテーブル（ `TB_RIGHT` ）を使用し、 `DISCRIMINATOR` カラムを使って正しいクラスにマップします。 `Person` 階層構造に対しては、同じ `ID` を他の2つのテーブル（ `TB_CORPORATION` と `TB_INDIVIDUAL` ）と共有する、スーパー・テーブル（ `TB_PERSON` ）と呼ばれるものを使います。3番目の階層（ `Estate` ）は、2つのカラム（ `REF_ESTATE_ID` と `REF_ESTATE_TYPE` ）の組み合わせとして定義される外部キーでリンクされる、別々の2つのテーブル（ `TB_BUILDING` と `TB_LAND` ）を使います。

図2はこのデータ・モデルを示します。

図2. 完全なデータ・モデル



データベースを設定する

Hibernateは広範な種類のRDBMSをサポートしています。この記事のサンプルは、そのどれとでも動作するはずですが、サンプル・コードとこの記事での用語は、全てJavaで書かれた完全機能のリレーショナル・データベースであるHSQLDB（ 参考文献 にリンクがあります ）に合わせています。サンプル・コード・パッケージのsqlディレクトリーに、datamodel.sqlという名前のファイルがあります。このSQLスクリプトが、この記事の例で使っているデータ・モデルを作ります。

Javaプロジェクトを設定する

サンプル・コードをビルドして実行するにはコマンドラインを使えばよいのですが、より良い統合を図るためには、IDEで設定することを考えても良いかも知れません。サンプル・コード・パッケージの中に、次のようなディレクトリーがあります。

- config -- サンプルの、全XMLコンフィギュレーション・ファイル（ マッピングやLog4Jなど ）を含んでいます。
- data -- HSQLDBが使用するコンフィギュレーション・ファイルを含んでいます。またデータベースを起動するために使用する、startHSQLDB.batという名前のバッチファイルもあります。
- src -- サンプルの、全ソースコードを含んでいます。

必要なJavaライブラリーとXMLコンフィギュレーション・ファイルを、アプリケーションのCLASSPATHに確実にコピーします。コードをコンパイルして正常に実行するために必要なのはHibernateとHSQLDBライブラリーのみです。こうしたパッケージは 参考文献 からダウンロードすることができます。

戦略1： サブクラス毎に一つのテーブル（ Persons ）

最初の戦略では、 `Person` 階層構造をどのようにマップするかを見て行きます。データ・モデルとクラス・モデルが非常に似ていることに注意してください。ですから階層構造の中にある各クラスに対して別々のテーブルを使用しますが、こうしたテーブルは全て同じプライマリー・キーを共有する必要があります（ これについてはすぐ後に説明します ）。そうするとHibernateはこのプライマリー・キーを使って、データベースに新しいレコードを挿入します。Hibernateはまたデータベースをアクセスする時にも、この同じプライマリー・キーを使って `JOIN` 操作を行います。

今度はオブジェクト階層構造をテーブル・モデルにマップする必要があります。ここでは3つのテーブル（ `TB_PERSON` と `TB_INDIVIDUAL` そして `TB_CORPORATION` ）があります。先に書いた通り、3つのどれにもプライマリー・キーとして、 `ID` という名前のカラムがあります。このように共通なカラム名を持つことは必須ではありませんが良い習慣であり、生成されるSQLクエリーを読むことがずっと容易になります。

リスト2に示すXMLマッピング・ファイルでは、 `Person` マッピング定義の中で2つの具象クラスが `<joined-subclass>` として宣言されていることに注意してください。XML要素 `<id>` は最上位レベルのテーブル `TB_PERSON` に対するプライマリー・キーにマップされていますが、（各サブクラスからの）`<key>` 要素は、 `TB_INDIVIDUAL` テーブルと `TB_CORPORATION` テーブルの、対応するプライマリー・キーにマップされています。

リスト2. Person.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="eg.hibernate.mapping.dataobject.Person" table="TB_PERSON" polymorphism="implicit">
<id name="id" column="ID">
<generator class="assigned"/>
</id>
<set name="rights" lazy="false">
<key column="REF_PERSON_ID"/>
<one-to-many class="eg.hibernate.mapping.dataobject.Right" />
</set>
<joined-subclass name="eg.hibernate.mapping.dataobject.Individual" table="TB_INDIVIDUAL">
<key column="id"/>
<property name="firstName" column="FIRST_NAME" type="java.lang.String" />
<property name="lastName" column="LAST_NAME" type="java.lang.String" />
</joined-subclass>
<joined-subclass name="eg.hibernate.mapping.dataobject.Corporation" table="TB_CORPORATION">
<key column="id"/>
<property name="name" column="NAME" type="string" />
<property name="registrationNumber" column="REGISTRATION_NUMBER" type="string" />
</joined-subclass>
</class>
</hibernate-mapping>
```

Javaコードでの `Individual` フォームの新しいインスタンスをHibernateで保存するのは、ごく単純です（リスト3）。

リスト3. Individualの新しいインスタンスを保存する

```
public Object create(Object object) {
Session session = null;
try {
session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(object);
session.flush();
tx.commit();
...
}
```

そうするとHibernateは、リスト4に示す2つのSQL `INSERT` リクエストを生成します。これは一つの `save()` に対して、2つのリクエストです。

リスト4. SQL挿入クエリー

```
insert into TB_PERSON (ID) values (?)
insert into TB_INDIVIDUAL (FIRST_NAME, LAST_NAME, id) values (?, ?, ?)
```

データベースから `Individual` にアクセスするには、リスト5に示すように、単純にHQLでクラス名を規定します。

リスト5. HQLクエリーを呼び出す

```
public Person findIndividual(Integer id) {
...
session.find("select p from " + Individual.class.getName() + " as p where p.id = ?",
new Object[] { id },
new Type[] { Hibernate.INTEGER });
...
}
```

そうするとHibernateは自動的にSQL `JOIN` を実行して、両方のテーブルから必要な全情報を検索します（ リスト6 ）。

リスト6. Individual に対するSQL SELECTクエリー

```
select individual0_.id as ID, individual0_.FIRST_NAME as FIRST_NAME55_,
individual0_.LAST_NAME as LAST_NAME55_
from TB_INDIVIDUAL individual0_
inner join TB_PERSON individual0__1_ on individual0_.id=individual0__1_.ID
where (individual0_.id=? )
```

抽象クラスをクエリーする

Hibernateは抽象クラスがクエリーされると、それに一致する、様々な具象サブクラスのコレクションを返します。例えばデータベースから全ての `Person` をクエリーすると、Hibernateは `Individual` オブジェクトと `Corporation` オブジェクトのリストを返します。

ところが具象クラスが何も規定されない場合には、Hibernateはどちらのテーブルを相手にすべきかが分からないので、SQL `JOIN` を実行する必要があります。HQLクエリーで返される、検索されたテーブル・カラムの中には、追加の動的カラムもあります。 `clazz` カラムは、Hibernateが返されたオブジェクトをインスタンス化し、中身を入れるために使用します。このクラス判別を、2番目の戦略で使おうとしているメソッドに対抗して、動的と呼ぶことにします。

リスト7は、 `id` プロパティを与えて抽象 `Person` をクエリーする方法を示し、リスト8はテーブル接合（ table junction ）を含めてHibernateによって自動生成されるSQLクエリーを示しています。

抽象クラスをクエリーする

```
public Person find(Integer id) {
...
session.find("select p from " + Person.class.getName() + " as p where p.id = ?",
new Object[] { id },
new Type[] { Hibernate.INTEGER });
...
}
```

リスト8. 任意タイプのPersonに対するSQL SELECTクエリー

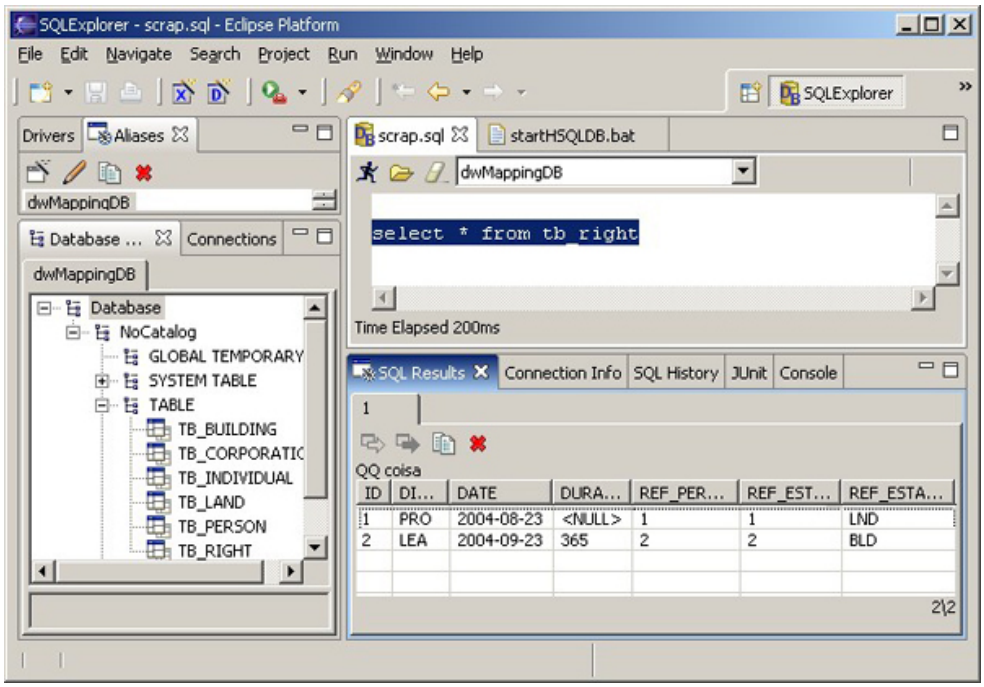
```
select person0_.ID as ID0_,
casewhen(person0__1_.id is not null, 1,
casewhen(person0__2_.id is not null, 2,
casewhen(person0_.ID is not null, 0, -1))) as clazz_0_,
person0__1_.FIRST_NAME as FIRST_NAME61_0_,
person0__1_.LAST_NAME as LAST_NAME61_0_,
person0__2_.NAME as NAME62_0_,
person0__2_.REGISTRATION_NUMBER as REGISTRA3_62_0_
from TB_PERSON person0_
left outer join TB_INDIVIDUAL person0__1_ on person0_.ID=person0__1_.id
left outer join TB_CORPORATION person0__2_ on person0_.ID=person0__2_.id
where person0_.ID=?
```

戦略2：クラス階層構造に対して一つのテーブル（ Rights ）

`Right` 階層構造に対しては、単一のテーブル（ `TB_RIGHT` ）を使って全クラスの階層構造を保存します。 `TB_RIGHT` テーブルは、 `Right` クラス階層構造の全属性を保存するのに必要な、全てのカラムを持っていることに注意してください。保存されたインスタンスの値は次にテーブルに保存され、使われていないカラムにはどれもNULL値が入れられます。（ このテーブルには「穴」がたくさんあるので、よくスイス・チーズ・テーブルと呼ばれます。 ）

図3では、 `TB_RIGHT` テーブルには追加のカラム `DISCRIMINATOR` があることに注意してください。Hibernateはこのカラムを使って対象のクラスを自動的にインスタンス化し、中身を入れます。このカラムは `<discriminator>` XML要素を使って、マッピング・ファイルからマップされます。

図3. TB_RIGHTテーブルの内容



単純化のためのヒント

大きなプロジェクトでは常に、幾つかのレベルの抽象クラスを持つ、複雑なクラス階層構造に直面させられます。幸い抽象クラスに対して `discriminator-value` を規定する必要はなく、Hibernateが使用する、実際の具象クラスに対して規定すれば良いだけです。

リスト2の `Person` マッピング・ファイルと同様、リスト9では、抽象クラス（ `Right` ）とその全属性をマップします。2つの具象クラス（ `Lease` と `Property` ）をマップするために、 `<subclass>` XMLタグを使います。このタグは非常に単純で、 `class` タグが `discriminator-value` 属性を必要とするのと同様、name属性を必要とします。Hibernateは最後の属性を使って、対象とすべきクラスを識別します。

図1 のクラス図から分かるかも知れませんが、 `discriminator` は、どのJavaクラスの属性でもありません。実際、 `discriminator` はマップさえもされておらず、単にHibernateとデータベース間で共有される特別なカラムにすぎません。

リスト9. Right.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="eg.hibernate.mapping.dataobject.Right" table="TB_RIGHT" polymorphism="implicit">
<id name="id" column="ID">
<generator class="assigned"/>
</id>
<discriminator>
<column name="DISCRIMINATOR"/>
</discriminator>
<property name="date" column="DATE" type="java.sql.Date" />
<many-to-one name="person" class="eg.hibernate.mapping.dataobject.Person" column="REF_PERSON_ID"/>
<any name="estate"
meta-type="string"
id-type="java.lang.Integer">
<meta-value value="LND" class="eg.hibernate.mapping.dataobject.Land"/>
<meta-value value="BLD" class="eg.hibernate.mapping.dataobject.Building"/>
<column name="REF_ESTATE_TYPE"/>
<column name="REF_ESTATE_ID"/>          </any>

<subclass name="eg.hibernate.mapping.dataobject.Property" discriminator-value="PRO"/>
<subclass name="eg.hibernate.mapping.dataobject.Lease" discriminator-value="LEA">
<property name="duration" column="DURATION" type="java.lang.Integer" />
</subclass>
</class>
</hibernate-mapping>
```

リスト9のマッピング・ファイルを見ると、 `Right` 階層構造と `Person` 階層構造の間に多対一の関係があり、これは（当然ですが） `Person` 階層構造（一対多）にある関係と反対であることに注意してください。また `Right` 階層構造と `Estate` 階層構造の間の関係にも注意してください（これについては後ほど触れます）。

最初の戦略と同様、Hibernateがデータベースをアクセスする時には、非常に効率的なSQLステートメントを生成します。リスト10に示すように具象クラスをクエリーする時には、Hibernateは `discriminator` の値に対して自動的にフィルターをかけます。つまりHibernateは指定されたクラスの適当なカラムのみを読む、ということなので、これは良いことです。

リスト10. 具象クラスに対するSQLクエリー

```
select property0_.ID as ID, property0_.DATE as DATE,
property0_.REF_PERSON_ID as REF_PERS4_, property0_.REF_ESTATE_TYPE as REF_ESTA5_,
property0_.REF_ESTATE_ID as REF_ESTA6_
from TB_RIGHT property0_ where property0_.DISCRIMINATOR='PRO'
```

抽象クラスをクエリーする場合には、ちょっと面倒になります。Hibernateには具体的にどのクラスを対象にすべきかが分からないので、（discriminatorカラムを含めて）一つ一つ全カラムを読んでからどのクラスをインスタンス化するかを判別し、そして最後にそこに中身を入れる必要があります。ですからdiscriminatorは、最初の戦略での `clazz` カラムと同じ役割を果たします。ところがクラス名はdiscriminatorの値から直接来ているので、この手法は、より静的なものであることが分かります。

リスト11.（抽象）RightクラスのSQLクエリー

```
select right0_.ID as ID,
right0_.DISCRIMINATOR as DISCRIMI2_,
right0_.DATE as DATE, right0_.REF_PERSON_ID as REF_PERS4_,
right0_.REF_ESTATE_TYPE as REF_ESTA5_, right0_.REF_ESTATE_ID as REF_ESTA6_,
right0_.DURATION as DURATION from TB_RIGHT right0_
```

戦略間の互換性

HibernateマッピングのDTDで定義されているように、この記事で取り上げている最初の2つの戦略は相互排他的です。ですから単一の階層構造をマップするために組み合わせて使う、ということはできません。

データベース・モデルの整合性

この2番目の戦略には一つ、大きな懸念があります。これが動作するためには、非共有カラムの全てを `NULLABLE` に設定する必要があるということです。開発者は普通、データベースの制約に頼るものなので、そのように設定した結果できるテーブルは非常に扱いにくいものになります（例えばリース期間を `NULL` に設定した `Lease` では意味がありません！）。

一つの解決方法は、データベース・レベルでのチェック制約（database-level checking constraints）を使うことです。 `DISCRIMINATOR` の値によって、リスト12に示すような実装ルールのセットを定義することができます。もちろん、データベース・エンジンがこの機能をサポートしている必要があります。さらにこうした制約は、全具象クラスに対して単一の有効表現によって、しかも同時に表現する必要があります。ですから階層構造が複雑になると維持管理が困難になります。

リスト12. データ整合性の制約

```
alter table TB_RIGHT
add constraint CHK_RIGHT check(
(discriminant ='DPP' and date is null and duration is null)
or (discriminant ='DLM' and date is not null and duration is not null));
```

戦略3： 具象クラス毎に一つのテーブル（Estates）

3番目の、そして最後の戦略は、最も創造的なものです。具象クラス毎に一つのテーブルを持ち、`Estate` 抽象クラスに対してはテーブルを持ちません。Hibernateが提供するpolymorphismのサ

ポートに依存するのです。リスト13に示すXMLマッピング・ファイルでは、2つの具象クラス（ `Building` と `Land` ）しかマップされていないことに注意してください。

リスト13. Estate.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="eg.hibernate.mapping.dataobject.Land" table="TB_LAND" polymorphism="implicit">
<id name="id" column="ID">
<generator class="assigned"/>
</id>
<property name="description" column="DESCRIPTION" type="java.lang.String" />
<property name="squareFeet" column="SQUARE_FEET" type="java.lang.Double"/>
</class>
<class name="eg.hibernate.mapping.dataobject.Building" table="TB_BUILDING" polymorphism="implicit">
<id name="id" column="ID">
<generator class="assigned"/>
</id>
<property name="description" column="DESCRIPTION" type="java.lang.String" />
<property name="address" column="ADDRESS" type="java.lang.String"/>
</class>
</hibernate-mapping>
```

テーブル間でID値を共有する

同じクラス階層構造内部でマップされる2つのテーブル間では、同じ `ID` 値を共有しないようにすることが重要です。同じ `ID` 値を共有すると、Hibernateは同じ `ID` に対して幾つか別々のオブジェクトを返してしまいます。これではHibernateが、そして皆さんも、混乱してしまいます。

リスト13のマッピング・ファイルを見て皆さんは恐らく、「これは私が毎日使っているマッピングと何も違わないではないか！」と思われるかも知れません。そして、そう考えるのも正しいと言えるのです。実際、この3番目の戦略で必要な条件は一つしかありません。つまり `polymorphism` 属性を明確に、 `implicit` に設定する必要がある、ということです。

`Estate` クラスはマッピング・ファイルのどこにもありませんが、クラス階層構造の中には、やはり存在しています。そして、マップされた2つのクラス（ `Building` と `Land` ）は `Estate` から継承しているので、リスト14に示すように、この抽象スーパークラスをHQLクエリーに使うことができるのです。そうするとHibernateは、サブクラスのそれぞれに対して次々に適当なSQLクエリーを行えるように、この抽象クラスを継承するイントロスペクション（ introspection ）を使ってクラスを識別します。

リスト14. find() コールでのHQLクエリー

```
public Estate find(Integer id) {
...
List objects =
session.find(
"select e from " + Estate.class.getName() + " as e where e.id = ?",
new Object[] { id },
new Type[] { Hibernate.INTEGER });
...
}
```

Hibernateは与えられた `ID` に一致する `Estate` を見つけるために、リスト15に示す2つのクエリーをデータベースに対して送信する必要があります。

リスト15. SQLクエリー

```
select land0_.ID as ID, land0_.DESCRIPTION as DESCRIPT2_, land0_.SQUARE_FEET as SQUARE_F3_
from TB_LAND land0_ where (land0_.ID=? )
select building0_.ID as ID, building0_.DESCRIPTION as DESCRIPT2_, building0_.ADDRESS as ADDRESS
from TB_BUILDING building0_ where (building0_.ID=? )
```

2番目の戦略で見た通り、 `Right` クラスと `Estate` クラスの間には多対一の関係があります。単純に言えば、例えば次のようになります。「 `Estate` は多くの `Rights` を参照している可能性があります。しかし、それぞれの `Rights` は一つの `Estate` を参照しているのみ、という可能性があります。」しかしここでのデータ・モデルの観点から見ると、(`TB_RIGHT` と `TB_PERSON` 間の制約のような) 外部キー制約を作るために使えるような独自のテーブルはありません。これでは実質的に外部キーを作ることが不可能になります。幸いHibernateは `<any>` タグという、非常に強力なXMLマッピング要素を提供しています。このタグの使い方をリスト16に示します。

リスト16. anyリレーションシップのXMLマッピング

```
<any name="estate"
meta-type="string"
id-type="java.lang.Integer">
<meta-value value="LND" class="eg.hibernate.mapping.dataobject.Land"/>
<meta-value value="BLD" class="eg.hibernate.mapping.dataobject.Building"/>
<column name="REF_ESTATE_TYPE"/>
<column name="REF_ESTATE_ID"/>
</any>
```

polymorphismを使用不可にする

polymorphismのサポートが使用不可になっているクラス (`<class...polymorphism="explicit"...>`) は、そのスーパークラスのどれをターゲットにしたクエリーからも除外されます。

この新しいマッピングを詳しく見てみましょう。仮想外部キーは `TB_RIGHT` テーブルからの2つのカラムに基づいています。最初の方 (`REF_ESTATE_TYPE`) は、対象となるクラス名をマップするために使用する判別文字列 (discriminator string) を含んでいます。2番目の方 (`REF_ESTATE_ID`) は、もう一方のテーブルのプライマリー・キーからのカラム名です。Hibernateはデフォルトの設定を使って、マップされたクラス名を最初のカラムに保存しようとしますが、これは (特にコードのリファクタリング中にクラス名が変更される時には) スペースを浪費し非効率なことが分かります。幸いHibernateには、XML要素 `<meta-value>` を使ってクラス名を文字列定数に関連付けるための方法が用意されています。こうした文字列定数は2番目の戦略で説明したdiscriminatorsと同じ役割を果たします。ここでも、この機能に関係するのはHibernateとデータベースのみなので、クラスの階層構造を変更することはありません。

データベース・モデルの整合性

標準的なSQLでは、あるカラムに対して同時に複数のテーブルによる参照制約 (referential constraints) を許していませんが、それでもターゲット・テーブルにデータが存在することをチェックするトリガーを追加することはできます (トリガーが読み取るdiscriminator値がありさえすれば)。しかし、こうした整合性強制 (integrity enforcement) の方法を維持するのは困難かも知れず、データベースの全体的なパフォーマンスも低下させる可能性があります。

Polymorphismを、極限まで使う

Hibernateに組み込まれているpolymorphismを使う場合には、一つ念頭に置くべきことがあります。 `implicit` に設定された `polymorphism` 属性で全てのクラスがマップされているとすると、注意しないと想定以上の情報を検索してしまうことになります。リスト17は、単語2つのHQLクエリーを使ってデータベース全体を検索する方法を示しています。

リスト17. HQLクエリー

```
public List all() {
...
List objects = session.find("from Object");
...
}
```

非常に強力だと思いませんか。もちろん、一つのHQLクエリーでデータベース全体を検索する必要のある人は（あるいは検索したいと思う人は）稀でしょう。この（ほとんど無意味な）例は、暗黙的polymorphismの強力さを示すためのものです。この強力さを活用すれば、データベースに対して送信される、無駄でリソースを浪費するSQLクエリーを避けることができるのです。

まとめ

この記事では、Hibernateが提供している3つのマッピング戦略に関して、非常に単純な実装例を説明しました。まとめると、各戦略には下記のような利点と欠点があります。

- 最初の戦略（サブクラス毎に一つのテーブル）では、オブジェクトがインスタンス化され中身が入れられる度に、Hibernateは複数のテーブルを読みます。インデックスが適切に定義されており、階層構造が深すぎなければ、この操作によって良い結果が得られます。ただし、そうでない場合には、全体的なパフォーマンスに問題が起きるかも知れません。
- 2番目の戦略（クラス階層構造毎に一つのテーブル）では、チェック制約（check constraints）を使って整合性を定義する必要があります。この戦略は、カラム数が時間と共に増えてくると維持が困難になる可能性があります。逆にそうした制約を全く使わず、アプリケーションのコードが自分でデータ整合性を管理するようにする、という選択をすることもできます。
- 3番目の戦略（具象クラス毎に一つのテーブル）には一部マッピングの制限があり、下にあるデータ・モデルは参照整合性（referential integrity）を使うことができません。これではリレーショナル・データベース・エンジンを最大限に使っていないことになります。その代わり良い点として、この戦略は他の2つの戦略と容易に組み合わせることができます。

どの戦略を選択するにせよ、Javaクラスを変更する必要はないことを常に忘れないでください。つまり、ビジネス・オブジェクトとパーシスタンス・フレームワークの間には、全く何のリンクもありません。オブジェクト・リレーショナルなJavaプロジェクトでHibernateが非常に人気があるのは、こうした柔軟性によるものなのです。

ダウンロード

内容	ファイル名	サイズ
Source code	j-hibernate-source.zip	24KB

著者について

Xavier Coulon

Xavier Coulonは6年前にITスペシャリストとしてIBM Franceに入社し、様々なプラットフォームでのERPコンサルティングを開始しました。この2年間は、StrutsやHibernateなどのオープン・ソース・フレームワークを含めた、大規模なJ2EEプロジェクトに従事しています。連絡先はxavier.coulon@fr.ibm.comです。

Christian Brousseau

Christian Brousseau は誇り高きカナダ人であり、10年以上もソフトウェア開発を行っています。当初は大量のWindows開発（C++, Visual Basic, Microsoft Foundation Classes, ActiveX）を行っていましたが、Javaの1.0がリリースされて以降はJavaプロジェクトに移行しています。専門的なJ2EEコンサルタントとして、フランスで働くという素晴らしい機会を得ており、大規模な、企業レベルJ2EEプロジェクトの、設計から開発、展開にまで従事しています。連絡先はcbrous@fr.ibm.comです。