第6章

映射集合和实体关联

本章内容

- 基本的集合映射策略
- 映射值类型的集合
- 映射父/子实体关系

有两个重要（且有时难以理解）的主题——集合的映射和实体类之间关联的映射并没有出现在前面的章节中。

大多数刚接触Hibernate的开发人员在第一次尝试映射典型的父/子关系（parent/child relationship）时，都要处理集合和实体关联。本章不会立即切入正题，而是从基本的集合映射概念和简单的例子开始，之后给出实体关联中的第一个集合，当然我们在第7章会讲到更复杂的实体关联映射。为了对这部分知识获得全面的了解，建议你阅读这两章。

6.1 值类型的 set、bag、list 和 map

值类型（value type）的对象不具备数据库同一性，它属于一个实体实例，其持久化状态被嵌入到所拥有实体的表行中——至少，在实体有一个对值类型的单个实例的引用的情况下。如果实体类有一个值类型的集合（或者对值类型实例的引用的集合），就需要一张额外的表，即所谓的集合表。

在将值类型的集合映射到集合表之前，要记住，值类型的类没有标识符或者标识符属性。值类型实例的生命期限由所拥有的实体实例的生命期限决定。值类型不支持共享的引用。

Java具有一个丰富的集合API，因此可以选择最适合领域模型设计的集合接口和实现。我们来看一些最常见的集合映射。

假设CaveatEmptor的卖主能够给`Item`添加图片。图片只有通过包含的货品才能访问；它不需要支持来自系统中任何其他实体的关联。应用程序通过`Item`类管理图片集合，增加和移除元素。图片对象在集合之外没有生命，它依赖`Item`实体。

在这种情况下，把图片建模为值类型就不无道理了。接下来，需要决定使用什么集合。

6.1.1　选择集合接口

在Java领域模型中，集合属性的惯用语始终相同：

```
private <<Interface>> images = new <<Implementation>>();
...
// Getter and setter methods
```

使用接口来声明属性的类型，而不是实现。选择一种匹配的实现并立即初始化集合，这么做避免了未被初始化的集合（我们不建议在构造函数或者设置方法中太迟初始化集合）。

如果使用JDK 5.0，可能用JDK集合的一般版本进行编码。注意这不是必需的，你也可以在映射元数据中显式地指定集合的内容。以下是一个典型的一般的Set，包含类型参数：

```
private Set<String> images = new HashSet<String>();
...
// Getter and setter methods
```

开箱即用，Hibernate支持最重要的JDK集合接口。换句话说，它知道如何以持久化的方式保存JDK集合、映射和数组的语义。每个接口都有一个Hibernate支持的匹配实现，并且使用正确的组合很重要。Hibernate只包装已经在字段的声明中初始化的集合对象（或者如果不是正确的对象，有时就替换它）。

不扩展Hibernate，而是从下列集合中选择：

- 使用<set>元素映射java.util.Set。使用java.util.HashSet初始化集合。它的元素顺序没有保存，并且不允许重复元素。这在典型的Hibernate应用程序中是最常见的持久化集合。
- 可以使用<set>映射java.util.SortedSet，且sort属性可以设置成比较器或者用于内存排序的自然顺序。使用java.util.TreeSet实例初始化集合。
- 可以使用<list>映射java.util.List，在集合表中用一个额外的索引列保存每个元素的位置。使用java.util.ArrayList初始化。
- 可以使用<bag>或者<idbag>映射java.util.Collection。Java没有Bag接口或者实现；然而，java.util.Collection允许包语义（可能的重复，不保存元素顺序）。Hibernate支持持久化的包（它内部使用列表，但是忽略元素的索引）。使用java.util.ArrayList初始化包集合。
- 可以使用<map>映射java.util.Map，保存键/值对。使用java.util.HashMap初始化属性。
- 可以使用<map>元素映射java.util.SortedMap，且sort属性可以设置为比较器或者用于内存排序的自然顺序。使用java.util.TreeMap实例初始化该集合。
- Hibernate使用<primitive-array>（对于Java基本的值类型）和<array>（对于其他的一切）支持数组（array）。但是它们很少用在领域模型中，因为Hibernate无法包装数组属性。没有字节码基础设施（BCI），就失去了延迟加载，以及为持久化集合优化过的脏检查、基本的便利和性能特性。

JPA标准没有列出所有这些选项。可能的标准集合属性类型是Set、List、Collection和Map。不考虑数组。

此外，JPA规范仅仅指出集合属性持有对实体对象的引用。值类型的集合（例如简单的String实例）没有被标准化。然而，规范文件已经提及JPA的未来版本将支持可嵌入类的集合元素（换句话说，是值类型）。如果想要通过注解映射值类型的集合，将需要特定于供应商的支持。Hibernate Annotations包括了这种支持，我们期待许多其他的JPA供应商也同样支持。

如果想要映射Hibernate不直接支持的集合接口和实现，就要告诉Hibernate定制集合的语义。Hibernate中的扩展点称作PersistentCollection；通常扩展其中现有的PersistentSet、PersistentBag或者PersistentList类中的一个。定制持久化集合不太容易编写，如果你不是个经验丰富的Hibernate用户，我们也不建议这么做。示例作为Hibernate下载包的一部分，可以在Hibernate测试套件（test suite）源代码中找到。

现在快速看一下几种始终实现货品图片集合的场景。首先以XML格式映射它，然后利用Hibernate对集合注解的支持。现在，假设图片被保存在文件系统中的某个地方，并且在数据库中只保存了文件名。我们不讨论如何用这种方法保存和加载图片，而是关注映射。

6.1.2 映射 set

最简单的实现是String图片文件名的Set。首先，把集合属性添加到Item类：

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}
public void setImages(Set images) {
    this.images = images;
}
```

现在，在Item的XML元数据中创建下列映射：

```
<set name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

图片的文件名保存在具名ITEM_IMAGE的集合表中。从数据库的观点来看，这张表是单个的实体，是一张单独的表，但是Hibernate为你隐藏了这一点。<key>元素在引用自己的实体的主键ITEM_ID的集合表中，声明了外键列。<element>标签把这个集合声明为值类型实例的一个集合——在这个例子中，是字符串的集合。

集无法包含重复元素，因此ITEM_IMAGE集合表的主键是<set>声明中这两个列的复合：ITEM_ID和FILENAME。可以在图6-1中见到这个Schema。

允许用户不止一次地添加同一张图片似乎不可能，但是假设你允许这么做，那么在这种情况下，哪种映射最合适呢？

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

图6-1　字符串集合的表结构和示例数据

6.1.3　映射标识符 bag

允许重复元素的无序集合被称作包（bag）。奇怪的是，Java Collections框架没有包括包实现。然而，java.util.Collection接口有包语义，因此只需要一种相匹配的实现。你有两种选择：

- 用java.util.Collection接口编写集合属性，并在声明中用JDK的一个ArrayList对它进行初始化。在Hibernate中用标准的<bag>或者<idbag>元素映射集合。Hibernate有一个内建的PersistentBag可以处理列表；但与包的约定一致，它忽略元素在ArrayList中的位置。换句话说，你得到了一个持久化的Collection。
- 用java.util.List接口编写集合属性，并在声明中用JDK的一个ArrayList把它初始化。像前一个选项一样映射它，但是在领域模型类中公开了一个不同的集合接口。这种方法有效，但不建议使用，因为使用这个集合属性的客户端可能认为元素的顺序会始终被保存着，其实如果把它作为<bag>或者<idbag>映射，就并非如此了。

建议使用第一个选项。把Item类中images的类型由Set改为Collection，并用ArrayList把它初始化：

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

注意，设置方法接受Collection，它可以是JDK集合接口层次结构中的任何东西。然而，Hibernate聪明到足以在持久化集合的时候替换它。（它内部也依赖ArrayList，就像你在字段的声明中所做的那样。）

还必须修改集合表以允许重复FILENAME；表需要一个不同的主键。<idbag>映射添加了一个代理键列到集合表，很像用于实体类的合成标识符：

```
<idbag name="images" table="ITEM_IMAGE">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

在这个例子中，主键是生成的ITEM_IMAGE_ID，就如在图6-2中所见。注意，主键的native生成器不支持<idbag>映射，必须指定一种具体的策略。这通常不成问题，因为无论如何，现实世界的应用程序都经常使用定制的标识符生成器。也可以用占位符隔离标识符生成策略；请见3.3.4节的第3小节。

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	1	fooimage1.jpg
2	1	fooimage1.jpg
3	3	barimage1.jpg

图6-2 允许重复包元素的一个代理主键

还要注意，ITEM_IMAGE_ID列没有以任何方式公开给应用程序。Hibernate在内部管理它。

一种更为可能的场景是，你希望把添加图片到Item的顺序保存在那里。实现这个有许多种好方法，其中一种是使用真实的列表，而不是包。

6.1.4 映射 list

首先，更新Item类：

```
private List images = new ArrayList();
...
public List getImages() {
    return this.images;
}

public void setImages(List images) {
    this.images = images;
}
```

<list>映射需要把一个索引列（index column）新增到集合表。索引列定义元素在集合中的位置。因而，Hibernate能够保存集合元素的顺序。映射集合为<list>：

```
<list name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <list-index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

（XML DTD中也有index元素，用于与Hibernate 2.x兼容。建议使用这个新的list-index；它比较不容易引起混淆，并且作用相同。）

集合表的主键是ITEM_ID和POSITION的复合。注意，现在允许重复元素（FILENAME）了，这与列表的语义一致，请见图6-3。

持久化列表的索引从0开始。可以改变它，例如在映射中使用<list-index base="1".../>。注意，如果数据库中的索引数字不连续，Hibernate就会把空元素添加到Java列表中。

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	POSITION	FILENAME
1	0	fooimage1.jpg
1	1	fooimage2.jpg
1	2	fooimage3.jpg

图6-3　集合表保存每个元素的位置

另一种方法，可以映射一个Java数组而不是列表。Hibernate支持这个；数组映射事实上与前一个例子是等同的，除了使用不同的元素和属性名称之外（<array>和<array-index>）。然而，基于前面阐述过的原因，Hibernate应用程序很少使用数组。

现在，假设一件货品的图片除了文件名之外还有用户提供的名称。在Java中对这个进行建模的一种方法是映射，使用与键相同的名称以及与映射的值一样文件名。

6.1.5　映射 map

再一次对这个Java类做小小的变化：

```
private Map images = new HashMap();
...
public Map getImages() {
    return this.images;
}
public void setImages(Map images) {
    this.images = images;
}
```

再次强调，映射<map>类似于映射列表。

```
<map name="images" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

集合表的主键是ITEM_ID和IMAGENAME的复合。IMAGENAME列保存映射的键。还是允许重复元素；请见图6-4中表的图表形式。

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME
1	Image One	fooimage1.jpg
1	Image Two	fooimage2.jpg
1	Image Three	fooimage3.jpg

图6-4　把字符串作为索引和元素的映射表

这个映射是无序的。如果想要始终按图片的名称进行排序，该怎么办？

6.1.6 排序集合和有序集合

虽然英语单词遭到惊人的滥用，单词sorted和ordered对于Hibernate持久化集合却是指不同的东西。排序集合（sorted collection）是指用一个Java比较器在内存中进行排序。有序集合（ordered collection）则指用一个包含order by子句的SQL查询在数据库级中排列。

来把图片的映射变成一个排了序的映射。首先，要改变Java属性的初始化为java.util.TreeMap，并转换到java.util.SortedMap接口：

```
private SortedMap images = new TreeMap();
...
public SortedMap getImages() {
    return this.images;
}

public void setImages(SortedMap images) {
    this.images = images;
}
```

如果把它映射为排序，Hibernate会相应地处理这个集合：

```
<map name="images"
        table="ITEM_IMAGE"
        sort="natural">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

通过指定sort="natural"，告诉Hibernate使用SortedMap，并根据java.lang.String的compareTo()方法对图片名称进行排序。如果需要一些其他的排序算法（例如反向字母顺序），可以在sort属性中指定实现java.util.Comparator的类名称。例如：

```
<map name="images"
     table="ITEM_IMAGE"
     sort="auction.util.comparator.ReverseStringComparator">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

像下面这样映射java.util.SortedSet（包含一个java.util.TreeSet实现）：

```
<set name="images"
     table="ITEM_IMAGE"
     sort="natural">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

包不可能被排序（可惜没有TreeBag），列表（list）也一样；列表元素的顺序由列表索引定义。

另一种方法，不转换到Sorted*接口（和Tree*实现），你或许想要使用一个链接映射（Linked Map），并在数据库端而不是内存中给元素排序。在Java类中保留Map/HashMap声明，并创建下列映射：

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

order-by属性中的表达式是SQL order by子句的一个片段。在这个例子中，在集合的加载期间，Hibernate按IMAGENAME列的升序排列集合元素。甚至可以在order-by属性中包括SQL函数调用：

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="lower(FILENAME) asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

可以按集合表的任何列进行排列。Hibernate内部使用LinkedHashMap，它是保存关键元素插入顺序的一个映射的变形。换句话说，就是在集合的加载期间，Hibernate把元素添加到集合的顺序，就是你在应用程序中见到的迭代顺序。用Set也可以完成同样的工作：Hibernate内部使用LinkedHashSet。在Java类中，属性是一般的Set/HashSet，但是Hibernate内部包含LinkedHashSet的包装再次通过order-by属性启用了排列：

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="FILENAME asc">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

也可以让Hibernate在集合的加载期间为你排列包的元素。Java集合属性是Collection/ArrayList或者List/ArrayList。Hibernate内部使用ArrayList来实现一个保存了插入-迭代顺序的包：

```
<idbag name="images"
       table="ITEM_IMAGE"
       order-by="ITEM_IMAGE_ID desc">
```

```
        <collection-id type="long" column="ITEM_IMAGE_ID">
            <generator class="sequence"/>
        </collection-id>

        <key column="ITEM_ID"/>

        <element type="string" column="FILENAME" not-null="true"/>
    </idbag>
```

Hibernate内部用于集和映射的链接集合只在JDK 1.4或者更高的版本中可用；更早的JDK没有LinkedHashMap和LinkedHashSet。有序包在所有JDK版本中都可用；内部使用ArrayList。

在真实的系统中，你可能需要保存图片名称和文件名之外的更多东西。你或许想要给这些额外的信息创建一个Image类。对于组件的集合来说，这是一个完美的使用案例。

6.2 组件的集合

可以把Image映射为实体类，并创建一个从Item到Image的一对多关系。但是这没有必要，因为Image可以建模为一个值类型：这个类的实例有一个依赖的生命周期，不需要它们自己的同一性，并且不必支持共享的引用。

作为一个值类型，Image类定义了属性name、filename、sizeX和sizeY。它与它的所有者Item实体有着单个关联，如图6-5所示。

就如你可以从复合关联风格（图中黑色菱形处）中所见，Image是Item的一个组件，Item则是负责Image实例的生命周期的实体。关联的多样性进一步把这个关联声明为多值（many-valued）——也就是说，同一个Item实例有多个（或者0个）Image实例。

来快速看一下Java中的这个实现，以及XML格式的映射。

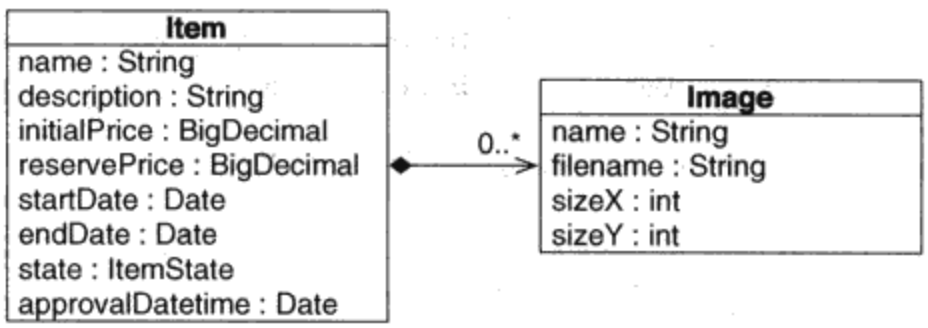

图6-5 Item中Image组件的集合

6.2.1 编写组件类

首先，把Image类实现为一般的POJO。如第4章所述，组件类没有标识符属性。你必须实现equals()（和hashCode()），并比较name、filename、sizeX和sizeY属性。Hibernate依赖这个等同性子程序检查实例中的修改。并非所有的组件类都需要equals()和hashCode()的定制实现（前面已经提到过这一点）。但是，我们建议把它用给任何组件类，因为这种实现很简单，并且“小心不出大错”可是至理名言啊。

Item类可以有图片的Set，不允许重复。我们来把这一思想映射到数据库。

6.2.2　映射集合

组件的集合被类似地映射到JDK值类型的集合。唯一的区别是用<composite-element>代替<element>标签。有序的图片集（内部使用LinkedHashSet）可以像这样映射：

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">

    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <property name="name" column="IMAGENAME" not-null="true"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX" not-null="true"/>
       <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

图6-6展现了包含示例数据的表。

ITEM

ITEM_ID	ITEM_NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME	SIZEX	SIZEY
1	Foo	Foo.jpg	123	123
1	Bar	Bar.jpg	420	80
2	Baz	Baz.jpg	50	60

图6-6　组件映射的集合的示例数据表

这是一个集，因此集合表的主键是键列和所有元素列的一个复合：ITEM_ID、IMAGENAME、FILENAME、SIZEX和SIXEY。因为这些列都出现在主键中，需要用not-null="true"声明它们（或者确保它们在任何现有的Schema中都为NOT NULL）。复合主键中任何列都不可以为空——因为你无法辨别不知道的东西。这可能是这个特定映射的一个缺点。在改善这一点之前（可能你猜到了，用标识符bag），我们来启用双向导航。

6.2.3　启用双向导航

从Item到Image的关联是单向的。可以通过Item实例访问集合并迭代anItem.getImages().iterator()导航到该图片。这是可以获得这些图片对象的唯一方法；没有其他的实体保存对它们（还是值类型）的引用。

另一方面，从一张图片导航回到一件货品没有意义。但是，有时候访问后退指针就像访问anImage.getItem()一样可能会比较方便。如果把<parent>元素添加到映射，Hibernate就会为

你填入这个属性：

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="IMAGE_NAME asc">

    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <parent name="item"/>
       <property name="name" column="IMAGENAME" not-null="true"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX" not-null="true"/>
       <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

但是，真正的双向导航是不可能的。你无法单独获取一个Image，然后导航回到它的父Item。这是一个重要的问题：你可以通过查询它们来加载Image实例。但是当你在HQL中查询或者使用Criteria查询时，这些Image对象都没有引用它们的所有者（属性为null）。它们作为标量值（scalar value）获取。

最后，声明所有属性为not-null可能并不是你所希望的。如果任何属性列都是可为空的，IMAGE集合表就需要一个不同的主键。

6.2.4 避免非空列

类似于<idbag>提供的那个额外的代理标识符属性，现在代理键列要派上用场了。作为一种附带作用，<idset>也允许重复——明显与集的概念冲突。基于各种各样的原因（包括事实上也没有人曾经要过这一特性），Hibernate没有提供<idset>或者除了<idbag>之外的任何代理标识符集合。因而，要通过包语义把Java属性改为Collection：

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

这个集合现在也允许重复Image元素了——这是用户接口或者任何其他应用程序代码的责任，如果需要集语义，就要避免这些重复元素。映射添加了代理标识符列到集合表：

```
<idbag name="images"
       table="ITEM_IMAGE"
       order-by="IMAGE_NAME asc">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
```

```
        <composite-element class="Image">
            <property name="name" column="IMAGENAME"/>
            <property name="filename" column="FILENAME" not-null="true"/>
            <property name="sizeX" column="SIZEX"/>
            <property name="sizeY" column="SIZEY"/>
        </composite-element>
    </idbag>
```

集合表的主键现在是ITEM_IMAGE_ID列，在Image类上实现equals()和hashCode()并不重要（至少Hibernate不需要）。你也不必用not-null="true"来声明属性。它们可以为空，如图6-7所示。

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGENAME	FILENAME	SIZEX	SIZEY
1	1	Foo	Foo.jpg	123	123
2	1	Bar	Bar.jpg	420	80
3	2	Baz	Baz.jpg	NULL	NULL

图6-7　使用带有代理键的包的Image组件的集合

应该指出，在这个包映射和标准的父/子实体关系（如本章前面映射过的那个）之间并没有太大的区别。表是一致的。这种选择主要是个人偏爱的问题。父/子关系支持对子实体和真正的双向导航的共享引用。为此付出的代价是更复杂的对象生命周期。值类型的实例可以通过添加新元素到集合而创建，并与持久化的Item关联。也可以通过从集合中移除元素来解除关联和永远删除它们。如果Image要成为一个支持共享引用的实体类，同样的操作在应用程序中就需要更多的代码，如稍后所见。

转换到不同主键的另一种方法是map。可以从Image类中移除name属性，并用图片名称作为map的键：

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">

    <key column="ITEM_ID"/>

    <map-key type="string" column="IMAGENAME"/>

    <composite-element class="Image">
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZEX"/>
        <property name="sizeY" column="SIZEY"/>
    </composite-element>
</map>
```

集合表的主键现在是ITEM_ID和IMAGENAME的复合。

像Image这样的复合元素类不是只限于像filename这样基本类型的简单属性。它可以包含其他的组件，通过<nested-composite-element>映射，甚至对实体的<many-to-one>关联。但是它无法拥有集合。包含多对一关联的复合元素很有用，第7章会回到这种映射的话题。

这样就结束了有关XML格式的基本集合映射的讨论。就像本节开头提到的，用注解映射值

类型的集合，与以XML格式映射相比有所不同；在编写本书之时，它还不是Java Persistence标准的一部分，但是在Hibernate中它还是可用的。

6.3 用注解映射集合

Hibernate Annotations包对包含值类型元素的集合映射支持非标准的注解，主要是org.hibernate.annotations.CollectionOfElements。再来快速看一些最常用的场景。

6.3.1 基本的集合映射

下列代码映射了String元素的一个简单集合：

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
private Set<String> images = new HashSet<String>();
```

集合表ITEM_IMAGE有两个列；它们一起形成了复合主键。如果使用一般的集合，Hibernate会自动侦测元素的类型。如果没有用泛型的集合编写代码，就要用targetElement属性指定元素类型——因此在前一个例子中，它是可选的。

为了映射持久化的List，要添加@org.hibernate.annotations.IndexColumn，该索引列带有索引的可选基数（默认为0）：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.IndexColumn(
    name="POSITION", base = 1
)
@Column(name = "FILENAME")
private List<String> images = new ArrayList<String>();
```

如果忘记了索引列，这个列表就会被当作一个包集合处理，相当于XML格式的<bag>。

对于值类型的集合，通常用<idbag>在集合表中获得一个代理主键。值类型元素的<bag>没有真正起作用；在Java级中允许重复，但在数据库中不行。另一方面，纯粹的包对于一对多的实体关联非常棒，如你将在第7章所见。

为了映射持久化的映射，就用@org.hibernate.annotations.MapKey：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
```

```
)
@org.hibernate.annotations.MapKey(
    columns = @Column(name="IMAGENAME")
)
@Column(name = "FILENAME")
private Map<String, String> images = new HashMap<String, String>();
```

如果忘记了映射键，这个映射的键就会自动映射到列MAPKEY。

如果该映射的键不是简单的字符串，而是可嵌入的类，就可以指定多个保存可嵌入组件的单独属性的映射键列。注意，@org.hibernate.annotations.MapKey是@javax.persistence.MapKey更强大的替代，后者不太有用（请见7.2.4节）。

6.3.2　排序集合和有序集合

集合也可以通过Hibernate注解排序或者按顺序排列：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.Sort(
    type = org.hibernate.annotations.SortType.NATURAL
)
private SortedSet<String> images = new TreeSet<String>();
```

（注意，没有@JoinColumn和/或@Column，Hibernate就给Schema应用一般的命名约定和默认。）@Sort注解支持各种SortType属性，语义与XML映射选项相同。上述映射使用了一个java.util.SortedSet（带有一个java.util.TreeSet实现）和自然的排序顺序。如果启用SortType.COMPARATOR，也要把comparator属性设置为实现了比较子程序的类。映射也可以被排序；但是，就像以XML格式映射一样，没有排序的Java包或者排序列表（按定义，它有一个持久化的元素顺序）。

映射、集，甚至包，都可以在加载时由数据库排序，通过ORDER BY子句中一个SQL片段进行：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.OrderBy(
    clause = "FILENAME asc"
)
private Set<String> images = new HashSet<String>();
```

特定于Hibernate的@OrderBy注解的clause属性，是被直接传递到数据库的一个SQL片段；它甚至可以包含函数调用或者任何其他原生的SQL关键字。有关排序和有序的内部实现细节请见前面的阐述；注解相当于XML映射。

6.3.3 映射嵌入式对象的集合

终于可以映射组件、用户自定义的值类型元素的集合了。假设你想要映射本章前面见过的同一个Image组件类，包含图片名称、大小等。

要在这个类上添加@Embeddable组件注解来启用嵌入：

```
@Embeddable
public class Image {

    @org.hibernate.annotations.Parent
    Item item;

    @Column(length = 255, nullable = false)
    private String name;
    @Column(length = 255, nullable = false)
    private String filename;

    @Column(nullable = false)
    private int sizeX;

    @Column(nullable = false)
    private int sizeY;

    ... // Constructor, accessor methods, equals()/hashCode()
}
```

注意，再次用Hibernate注解映射了一个后退指针，anImage.getItem()可能有用。如果不需要这个引用，可以省略这个属性。因为集合表需要所有组件列都为复合主键，把这些列映射为NOT NULL很重要。现在可以把这个组件嵌入到一个集合映射中了，甚至覆盖列定义（在下面的例子中，你覆盖了组件集合表的单个列的名称；所有其他的列都被指定为默认的策略）：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@AttributeOverride(
    name = "element.name",
    column = @Column(name = "IMAGENAME",
                     length = 255,
                     nullable = false)
)
private Set<Image> images = new HashSet<Image>();
```

为了避免不可为空的组件列，在集合表中需要一个代理主键，就像<idbag>在XML映射中提供的那样。通过注解，使用@CollectionId Hibernate扩展：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@CollectionId(
```

```
        columns = @Column(name = "ITEM_IMAGE_ID"),
        type = @org.hibernate.annotations.Type(type = "long"),
        generator = "sequence"
    )
    private Collection<Image> images = new ArrayList<Image>();
```

你现在已经使用XML映射元数据和注解，映射了所有基本的和一些更复杂的集合。转换关注点，现在思考包含非值类型、但引用其他实体实例的元素的集合。许多Hibernate用户试图映射一个典型的父/子实体关系，这涉及实体引用的集合。

6.4　映射父/子关系

在Hibernate用户社区的经验告诉我们，许多开发人员在开始使用Hibernate时要尝试的第一件事就是父/子关系的映射。这通常是你第一次遇到集合，也是你第一次必须思考实体和值类型之间的区别，否则会在ORM的复杂性中迷失。

管理类之间的关联和表之间的关系，是ORM的核心。在实现ORM解决方案中涉及的大部分难题都与关联管理有关。

前一节和本书的前面已经通过关系端的各种多样性，映射过值类型的类之间的关系。用一个简单的<property>映射一个（one）多样性，或者把它映射为<component>。多个（many）关联多样性需要值类型的集合，利用<element>或者<composite-element>映射。

现在想要映射实体类之间一值或者多值的关系。很显然，实体方面（如共享的引用和独立的生命周期）使得这个关系映射复杂化了。我们会逐步处理这些问题；而且，万一你不熟悉术语多样性（multiplicity），我们也将讨论到。

接下来的几节中要介绍的关系始终一样，即Item和Bid实体类之间的关系，如图6-8所示。

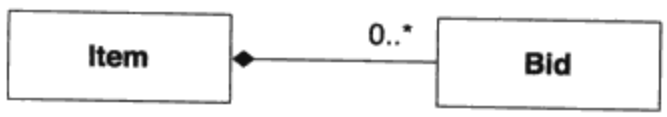

图6-8　Item和Bid之间的关系

记住这个类图。但是首先，我们要先行解释一些东西。

如果你已经用过EJB CMP 2.0，就会熟悉托管关联（或者托管关系）的概念。CMP关联被称作容器托管的关系（container managed relationship，CMR）是有理由的。CMP中的关联天生就是双向的。对关联的一侧所做的改变，会立即影响到另一侧。例如，如果调用aBid.setItem(anItem)，容器就会自动调用anItem.getBids().add(aBid)。

面向POJO的持久化引擎（如Hibernate）不实现托管关联，且POJO标准（如EJB 3.0和Java Persistence）不需要托管关联。与EJB 2.0 CMR相反，Hibernate和JPA关联天生都是单向的（unidirectional）。到Hibernate受到关注为止，从Bid到Item的关联与从Item到Bid的关联都是不同的关联！这是件好事——否则你的实体类就不可能在一个运行时容器之外使用（CMR是EJB 2.1实体之所以被认为有问题的主要原因）。

由于关联是如此重要，你需要一种精确的语言来对它们进行分类。

6.4.1 多样性

在描述和分类关联时，我们通常使用术语多样性（multiplicity）。在我们的例子中，多样性只是两小块信息：

- 一个特定的Item可以有不止一个Bid吗？
- 一个特定的Bid可以有不止一个Item吗？

看一眼领域模型（见图6-8）之后，就可以推断：从Bid到Item的关联是一个多对一的关联。回忆一下，关联是有方向性的，把从Item到Bid的反向关联归为一对多关联。

另外只有两种可能：多对多和一对一。我们将在第7章回到这个话题。

在对象持久化的上下文中，我们不关注多个（many）是否意味着两个或者最大五个或者没有限制；只关注大多数关联的选择性（optionality）；不特别关心是否需要关联实例，或者关联的另一侧是否可以为空（意味着0对多和0对0的关联）。但是，这些是影响你在关系数据Schema中选择完整性规则和在SQL DDL中定义约束的重要方面（请见8.3节）。

6.4.2 最简单的可能关联

从Bid到Item的关联（反之亦然）是一种最简单的可能实体关联的例子。两个类中有两个属性。一个是引用的集合，另一个是单个引用。

首先，这是Bid的Java类实现：

```
public class Bid {
    ...

    private Item item;

    public void setItem(Item item) {
        this.item = item;
    }

    public Item getItem() {
        return item;
    }

    ...
}
```

接下来是这个关联的Hibernate映射：

```
<class
    name="Bid"
    table="BID">
    ...
    <many-to-one
        name="item"
        column="ITEM_ID"
        class="Item"
        not-null="true"/>

</class>
```

这个映射被称作单向的多对一关联。(实际上，因为它是单向的，你不知道另一侧是什么，因此也可以称这个映射为单向的对一关联映射。)BID表中的ITEM_ID列是ITEM表主键的一个外键。

把类命名为Item，它显然是这个关联的目标。这通常是可选的，因为Hibernate可以通过反射在Java属性中确定目标类型。

添加了not-null属性，因为没有货品就不可能有出价——在SQL DDL中生成一项约束来体现这一点。BID中的外键列ITEM_ID永远不能为NULL，关联不是对0或者对一。这个关联映射的表结构如图6-9所示。

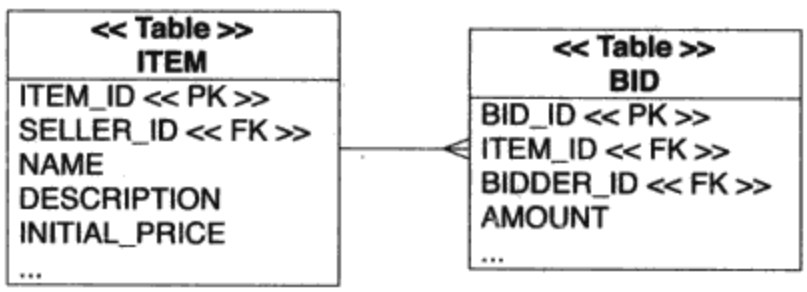

图6-9　一对多映射的表关系和主键

在JPA中，用@ManyToOne注解来映射这个关联，是在字段还是在获取方法中，这取决于实体的访问策略（由@Id注解的位置决定）：

```
public class Bid {
    ...
    @ManyToOne( targetEntity = auction.model.Item.class )
    @JoinColumn(name = "ITEM_ID", nullable = false)
    private Item item;

    ...
}
```

在这个映射中有两个可选的元素。首先，不一定要包括关联的targetEntity；它对于字段的类型来说是隐式的。显式的targetEntity属性在更复杂的领域模型中很有用。例如，在返回委托类（delegate class）——它模仿一个特定的目标实体接口——的一个获取方法中映射@ManyToOne时。

第二个可选的元素是@JoinColumn。如果没有声明外键列的名称，Hibernate会自动使用目标实体名称和目标实体的数据库标识符属性名称的一个组合。换句话说，如果没有添加@JoinColumn注解，外键列的默认名称则为item加id，用一条下划线隔开。然而，因为要使外键列为NOT NULL，所以无论如何都始终要用注解设置nullable = false。如果用Hibernate Tools生成Schema，@ManyToOne中的optional="false"属性也会在生成的列中导致一个NOT NULL约束。

这在以前也很容易。最重要的是，要意识到你可以不用任何其他东西而编写完整的应用程序。(可能经常用共享的主键一对一映射，如第7章所述。)你无需映射这个类关联的另一侧，并且可能已经映射了出现在SQL Schema（外键列）中的任何东西。如果需要为其生成特定Bid的Item实例，就利用创建过的实体关联调用aBid.getItem()。另一方面，如果需要已经对一件货品所做的所有出价，可以编写查询（用Hibernate支持的任何一种语言）。

使用像Hibernate这样的完全的ORM工具，原因之一当然在于不想编写这种查询。

6.4.3 使关联双向

为了不通过显式查询就能够为一件特定的货品轻松地抓取所有的出价，那么就要通过持久化对象的网络图进行导航和迭代。这么做最方便的方法是在Item：anItem.getBids().iterator()中使用集合属性。（注意，映射实体引用的集合还有其他很好的理由，但是不多。始终努力把这种集合映射当作一项特性，而不是必要条件。如果太难，就不要做。）

现在通过使Item和Bid之间的关系变成双向，来映射实体引用的集合。

首先把属性和脚手架代码添加到Item类：

```
public class Item {
    ...

    private Set bids = new HashSet();

    public void setBids(Set bids) {
        this.bids = bids;
    }

    public Set getBids() {
        return bids;
    }

    public void addBid(Bid bid) {
        bid.setItem(this);
        bids.add(bid);
    }
    ...
}
```

可以考虑用addBid()中的代码在对象模型中实现托管关联！（关于这些方法，我们已经在3.2节中讨论过很多。可以复习一下那里的代码示例。）

这个一对多关联的基本映射看起来像这样：

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

如果拿这个与本章前面的集合映射进行比较，会发现你用一个不同的元素<one-to-many>映射了集合的内容。这表明集合没有包含值类型实例，而是包含了对实体实例的引用。现在Hibernate知道如何处理共享的引用和被关联对象的生命周期（它禁用了值类型实例的所有隐式依赖的生命周期）。Hibernate也知道用给集合的表与目标实体类被映射到的表相同——<set>映射不需要table属性。

由<key>元素定义的列映射是BID表的外键列ITEM_ID，即已经在关系的另一侧映射的同一个列。

注意，表Schema并没有改变；它与之前你映射关联的多侧一样。但是，有一点不同：not null="true"不见了。问题是现在有两个不同的单向关联映射到同一个外键列。哪一侧控制这个列呢？

在运行时，同一个外键值有两个不同的内存表示法：Bid的item属性和由Item保存的bids集合的一个元素。假设应用程序修改了关联，例如通过在addBid()方法的这个片段中对货品添加出价：

```
bid.setItem(item);
bids.add(bid);
```

这段代码很好，但是在这种情况下，Hibernate侦测到内存持久化实例的两处变化。从数据库的观点来看，只有一个值必须更新以体现这些变化：BID表的ITEM_ID列。

Hibernate没有透明地侦测到引用同一个数据库列的两处变化，因为此时还没有做任何事来表明这是一个双向的关联。换句话说，已经映射了同一个列两次（在两个映射文件中这么做没有关系），Hibernate始终需要知道这一点，因为它无法自动侦测到这个重复（没有可以处理的合理的默认方法）。

在关联映射中还需要做的一件事，就是使它成为真正的双向关联映射。inverse属性告诉Hibernate，集合是<many-to-one>关联在另一侧的一个镜像：

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids"
         inverse="true">

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>

    </set>

</class>
```

在操作两个实例之间的链接时，如果没有inverse属性，Hibernate会试图执行两个不同的SQL语句，这两者更新同一个外键列。通过指定inverse="true"，显式地告诉Hibernate链接的哪一端不应该与数据库同步。在这个例子中，告诉Hibernate它应该把在关联的Bid端所做的变化传播到数据库，忽略只对bids集合所做的变化。

如果只调用anItem.getBids().add(bid)，则没有使任何变化成为持久化！只有当正确地设置了另一侧即aBid.setItem(anItem)时，才能得到想要的东西。这与Java中没有Hibernate时的行为一致：如果关联是双向的，就必须创建在两侧（而不是只在一侧）带有指针的链接。这就是我们为什么推荐像addBid()这样的便利方法的主要原因——它们在系统中负责双向引用，不用容器托管的关系。

注意，Hibernate Schema导出工具始终对SQL DDL的生成忽略关联映射的inverse侧。在这种情况下，BID表中的ITEM_ID外键列就获取了一个NOT NULL约束，因为已经对它做了与非反向

的（noninverse）<many-to-one>映射中一样的声明。

（你可以转换反向端吗？<many-to-one>元素没有inverse属性，但是可以用update="false"和insert="false"映射它，以有效地忽略掉任何UPDATE或者INSERT语句。然后集合端就是非反向的，考虑把它用于外键列的插入或者更新。第7章将会介绍这一点。）

再次通过JPA注解映射这个反向集合侧：

```
public class Item {
    ...

    @OneToMany(mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...

}
```

mappedBy属性相当于XML映射中的inverse属性；但是，它必须指定目标实体的反向属性。注意，这里不必再次指定外键列（它通过另一侧映射），因此不像XML那么冗长。

你现在有了一个有效的双向的（bidirectional）多对一关联（也可称为双向的一对多关联）。如果想要使它成为一个真正的父/子关系，还缺少最后一个选项。

6.4.4 级联对象状态

父子的概念意味着一个照顾另一个。实际上，这意味着只需要更少的代码行来管理父子之间的关系，因为有些事情可以自动处理。我们来探讨一下选项。

下列代码创建了一个新的Item（我们认为是父）和新的Bid实例（子）：

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

session.save(newItem);
session.save(newBid);
```

如果正在谈的是真正的父/子关系，对session.save()的第二次调用看起来就是多余的。保留这个想法，并再次回想一下实体和值类型：如果两个类都是实体，它们的实例就有一个完全独立的生命周期。新对象是瞬时的，如果要把它们保存在数据库中，必须使它们变成持久化。如果它们是实体，它们的关系则不影响它们的生命周期。如果Bid是值类型，Bid实例的状态则与它自己的实体的状态相同。然而，在这个例子中，Bid是一个单独的实体，有着它自己完全独立的状态。你有3种选择：

- 你自己负责独立的实例，必要时在Bid对象上执行额外的save()和delete()调用——除了管理关系所需的Java代码之外（从集合等中增加和移除引用）。
- 使Bid类成为值类型（组件）。可以用<composite-element>映射集合，并获得隐式的生命周期。但失去了实体的其他方面，例如对实例可能的共享引用。
- 需要对Bid对象的共享引用吗？目前，特定的Bid实例只被一个Item引用。但是，想象如果User实体也有一个由用户生成的bids集合。为了支持共享引用，必须把Bid映射为实

体。需要共享引用的另一个原因是，完整的CaveatEmptor模型中由Item发出的successfulBid关联。在这种情况下，Hibernate就会提供传播性持久化，它是一个可以启用的特性，用来节省代码行并让Hibernate自动管理关联实体实例的生命周期。

你不想执行超过绝对必要的更多持久化操作，并且不想改变领域模型——需要对Bid实例的共享引用。第三个选项是你要用来简化这个父/子示例的。

1. 传播性持久化

当实例化一个新的Bid并把它添加到Item时，bid应该自动变成持久化。你想要避免的是，使用一个额外的save()操作来显式地使Bid持久化。

为了启用跨关联的这个传播性状态，得把cascade选项添加到XML映射：

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids"
         inverse="true"
         cascade="save-update">

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>

    </set>

</class>
```

如果特定的Bid在集合中被持久化的Item引用，cascade="save-update"属性便为Bid实例启用了传播性持久化。

cascade属性是有方向性的：它只应用到关联的一端。也可以把cascade="save-update"添加到Bid映射中的<many-to-one>关联，但是由于出价是在货品之后创建的，这么做并没有什么意义。

JPA也在关联中支持级联实体实例状态：

```
public class Item {
    ...

    @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE },
               mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...

}
```

级联选项是你想要它变成传播性的每一个操作（per operation）。对于原生的Hibernate，用cascade="save-update"把save和update操作级联到被关联的实体。Hibernate的对象状态管理始终把这两个东西绑在一起，后面的章节中将会介绍。在JPA中，（几乎）相当的操作是persist和merge。

现在可以在原生的Hibernate中简化链接和保存Item和Bid的代码：

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

session.save(newItem);
```

bids集合中的所有实体现在也都变成持久化了，就像在每个Bid中手工调用save()时它们的表现一样。利用JPA EntityManager API（相当于Session），代码如下：

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

entityManager.persist(newItem);
```

目前不必担心update和merge操作；我们会在本书的后面回到这个话题。

常见问题 cascade在inverse中有什么作用？许多Hibernate新用户会问这个问题。答案很简单：cascade属性与inverse属性无关。它们经常出现在同一个集合映射中。如果映射实体的集合为inverse="true"，就为双向关联映射控制了SQL的生成。这个提示告诉Hibernate你映射了同一个外键列两次。另一方面，级联被用作一项便利的特性。如果决定从实体关系的一侧级联操作到被关联的实体，就省去了手工管理另一侧的状态所需的代码行。假设对象状态变成了传播性，你不仅可以级联实体的集合中的状态，还可以级联所有的实体关联映射中的状态。cascade和inverse有相似之处，它们都不出现在值类型的集合或者任何其他值类型映射中。这些规则隐含在值类型的特性中。

现在讲完了吗？嗯，可能还没有。

2. 级联删除

利用前一个映射，Bid和Item之间的关联非常松散的。目前为止，我们仅仅把使东西持久化当作一种传播性状态。那么如何删除呢？

货品的删除意味着这个货品的所有出价都删除了，这似乎有道理。事实上，这就是UML图中复合（图6-5黑色菱形处）的意思。利用当前的级联操作，你必须编写以下代码来使它发生：

```
Item anItem = // Load an item

// Delete all the referenced bids
for ( Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext(); ) {

    Bid bid = it.next();

    it.remove();                // Remove reference from collection
    session.delete(bid);        // Delete it from the database
}

session.delete(anItem);         // Finally, delete the item
```

首先通过迭代集合移除对bids的引用。删除数据库中的所有Bid实例。最后，删除Item。迭

代和移除集合中的引用似乎没有必要；毕竟，无论如何最后都会删除Item。如果能保证没有其他对象（或者任何其他表中的行）保存着对这些出价的引用，就可以使删除变成传播性。

Hibernate（和JPA）为此提供了一个级联选项。可以给delete操作启用级联：

```
<set name="bids"
     inverse="true"
     cascade="save-update, delete">
...
```

在JPA中级联的操作称作remove：

```
public class Item {
    ...
    @OneToMany(cascade = { CascadeType.PERSIST,
                           CascadeType.MERGE,
                           CascadeType.REMOVE },
               mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();
    ...
}
```

删除一个货品和它全部出价的相同代码被缩减如下，在Hibernate中或者用JPA：

```
Item anItem = // Load an item
session.delete(anItem);
entityManager.remove(anItem);
```

现在delete操作被级联到了集合中引用的所有实体。你不再需要担心从集合中移除，以及用手工一个个地删除那些实体了。

来思考进一步的复杂因素。你可能有对Bid实例的共享引用。就像前面建议的，User可以有对他们生成的Bid实例的一个引用集合。不先移除这些引用，就无法删除货品和它的所有出价。如果试图提交这个事务，就可能得到一个异常，因为外键约束可能遭到了破坏。

你必须追踪指针。这个过程可能不好看，如下列代码所示，它在删除出价以及最后删除货品之前，从有引用的所有用户中移除了所有引用：

```
Item anItem = // Load an item

// Delete all the referenced bids
for ( Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext(); ) {
    Bid bid = it.next();

    // Remove references from users who have made this bid
    Query q = session.createQuery(
      "from User u where :bid in elements(u.bids)"
    );
    q.setParameter("bid", bid);
    Collection usersWithThisBid = q.list();

    for (Iterator itUsers = usersWithThisBid.iterator();
         itUsers.hasNext();) {
         User user = (User) itUsers.next();
```

```
            user.getBids().remove(bid);
        }
    }

    session.delete(anItem);
    // Finally, delete the item and the associated bids
```

显然，你并不想要额外的查询（事实上是许多查询）。但是，在网络对象模型中，如果想要正确地设置指针和引用，除了像这样执行代码之外别无选择——没有持久化的垃圾收集器或者其他的自动机制。没有Hibernate级联选项可以帮忙；你必须在最终删除实体之前，追踪对它的所有引用。

（这并非全部的真相：因为表示从User到Bid关联的BIDDER_ID外键列是在BID表中，如果BID表中的一行被删除了，这些引用在数据库级别中也被自动移除。这不影响已经出现在当前工作单元内存中的任何对象，而且如果BIDDER_ID被映射到一张不同的（中间）表，它也不起作用。为了确保所有引用和外键列为空，必须在Java中追踪指针。）

另一方面，如果没有对实体的共享引用，就应该重新考虑映射，并把bids映射为集合组件（Bid映射为<composite-element>）。通过<idbag>映射，甚至表看起来也一样：

```
<class
    name="Item"
    table="ITEM">
    ...

    <idbag name="bids" table="BID">
        <collection-id type="long" column="BID_ID">
            <generator class="sequence"/>
        </collection-id>
        <key column="ITEM_ID" not-null="true"/>

        <composite-element class="Bid">
            <parent name="item"/>
            <property .../>
            ...
        </composite-element>

    </idbag>

</class>
```

Bid不再需要单独的映射。

如果真的想要使它成为一对多的实体关联，那么可以使用Hibernate提供的另一个你可能感兴趣的便利选项。

3. 启用孤儿删除

现在要阐述的级联选项有点难以理解。如果你理解了前一节的讨论，就应该不成问题了。

想象你要从数据库中删除一个Bid。注意在这个例子中没有删除父（Item）。目标是移除BID表中的一个行。看看以下代码：

```
anItem.getBids().remove(aBid);
```

如果集合让Bid映射为组件的集合，就像前一节一样，这段代码就触发了几项操作：

第十二章　西方能力本位职业教育与培训模式研究

能力本位的教育与培训(CBET)是当今世界各国职业教育与培训改革的方向。但是,不同国家对CBET的理解是不同的,这一差异不仅表现在能力观上,而且深入到了课程、教学与评价等各个层面,这样,就形成了不同的CBET模式。本章将从能力观、能力标准、课程、教学、评价几个方面详细剖析CBET几种模式之间的差异。

第一节　CBET的能力观

所谓能力观,就是人们关于"能力是什么"这个问题的基本看法。在心理学上对能力的概念早已有一个基本一致的定义,但在这里我们是把它作为一个教育概念来加以讨论。作为一个教育概念,人们对能力的看法往往殊异。然而在CBET体系中,对于能力的概念是不能不明的,因为能力观是CBET体系中最基本的核心要素,是构建CBET体系的基础。[①]不同的能力观,将产生不同的能力标准;能力标准不同,相应的课程模式、教学方式以及评价机制也就各异。当然,我们这里的探讨目的不在于、也不可能最终获得一个共识的能力概念,但是我们希望通过对观点纷纭的能力观的分析,能够有助于人们根据他们所处的环境与条件,来选择和确定与之相适宜的能力观,进而在此基础上构建一种最优化的CBET体系。

一、能力本位运动中对能力的不同理解

能力本位运动在北美地区兴起之初,由于产业界的代表在教育

---

① Ruth Nickse (Ed.) (1981) Competency-Based Education: Beyond Minimum Competency Testing. New York: Teachers College Press, p5.

```
<entity-mappings>

    <entity class="auction.model.Item" access="FIELD">
        ...
        <one-to-many name="bids" mapped-by="item">
            <cascade>
                <cascade-persist/>
                <cascade-merge/>
                <cascade-remove/>
            </cascade>
         </one-to-many>
    </entity>

    <entity class="auction.model.Bid" access="FIELD">
        ...
        <many-to-one name="item">
            <join-column name="ITEM_ID"/>
        </many-to-one>
    </entity>

</entity-mappings>
```

注意，在这个例子中，Hibernate扩展对级联孤儿删除是不起作用的。

6.5 小结

你可能被本章介绍的所有新概念搞得有点晕头转向了。你可能不得不多读几遍，并且我们鼓励你去尝试代码（并观察SQL日志）。本章介绍的许多策略和技术都是ORM的主要概念。如果你掌握了集合映射，并且一旦已经映射了第一个父/子实体关联，就克服了最大的困难。你将已经能够创建完整的应用程序了！

表6-1概括了本章讨论过的Hibernate和Java Persistence相关概念之间的区别。

表6-1 第6章的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate提供支持对Set、List、Map、Bag、标识符bag和数组的映射。支持所有的JDK集合接口，并且给定制的持久化集合提供了扩展点	支持标准的持久化Set、List、Map和bag
支持值类型和组件的集合	值类型和可嵌入对象的集合需要Hibernate Annotations
支持父/子实体关系，包含每个操作在关联中的传播性状态级联	可以映射实体关联，并启用每个操作在关联中的传播性状态级联
支持孤儿实体实例的自动删除	孤儿实体实例的自动删除需要Hibernate Annotations

本章只涵盖了实体关联选项的一小部分。第7章要深入探讨的剩余选项更为少见，或者说是我们刚刚描述过的这些技术的变形。