第7章

高级实体关联映射

本章内容

- 映射一对一和多对一的实体关联
- 映射一对多和多对多的实体关联
- 多态实体关联

使用关联（association）一词时，总是指实体之间的关系。前一章示范了一个单向的多对一关联，使它变成双向，最后把它变成一个父/子关系（包含级联选项的一对多和多对一关联）。

之所以在单独一章里讨论更高级的实体映射，原因之一在于它们中大部分都比较少见，或者至少是可选的。

只使用组件映射和多对一（偶尔为一对一）的实体关联是绝对有可能的。你甚至可以编写复杂的应用程序而不映射任何集合！当然，有效而轻松地访问持久化数据，例如通过迭代集合，就是使用完全的ORM而不是简单的JDBC查询服务的原因之一了。但是有些怪异的映射特性大部分时候应该谨慎使用，甚至尽量不用。

我们会在本章指出推荐使用的和可选的映射技术，就像介绍如何映射包含各种多样性的实体关联（不管是否带有集合）。

7.1 单值的实体关联

让我们从一对一的实体关联开始。

第4章曾讨论过，User和Address之间的关系（user有billingAddress、homeAddress和shippingAddress）最好用<component>映射来表示。这通常是表示一对一关系最简单的方法，因为在这种情况下，生命周期通常是依赖的，在UML中，要么是一个聚合，要么是一个组合。

但是如果想要Address的专用表，并且把User和Address都映射为实体时该怎么办？这个模型的好处之一在于共享引用的可能性——另一个实体类（如Shipment）也可以有一个对特定Address实例的引用。如果User有对该实例的引用，作为它们的shippingAddress，那么Address实例就必须支持共享引用，并且需要有自己的同一性。

在这种情况下，User和Address类就都有一个真正的一对一关联了。看看图7-1中修改过的

类图。

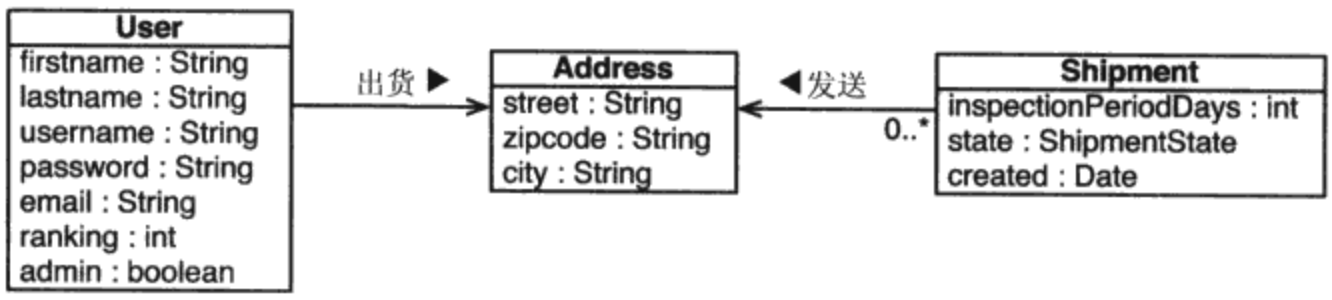

图7-1 作为实体的Address，包含了两个对相同实例引用的关联

第一个变化是Address类作为独立实体的映射：

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator .../>
    </id>
    <property name="street" column="STREET"/>
    <property name="city" column="CITY"/>
    <property name="zipcode" column="ZIPCODE"/>
</class>
```

假设你能够轻松地利用注解创建相同的映射，或者通过标识符属性把Java类变为实体——这是必须做的唯一变化。

现在来创建从其他实体到这个类的关联映射。有几种选择，首要的是主键一对一关联。

7.1.1 共享的主键关联

由主键关联而相关的两张表中的行共享相同的主键值。这种方法的主要困难在于，确保被关联的实例在保存对象时分配到了相同的主键值。在尝试解决这个问题之前，先看看主键关联是如何映射的。

1. 用XML映射主键关联

把实体关联映射到共享主键实体的XML映射元素是<one-to-one>。首先在User类中需要一个新属性：

```
public class User {
    ...
    private Address shippingAddress;
    // Getters and setters
}
```

接下来，在User.hbm.xml中映射关联：

```
<one-to-one name="shippingAddress"
            class="Address"
            cascade="save-update"/>
```

你给这个模型添加了一个固有的级联选项：如果User实例变成持久化，通常它的shippingAddress也要变成持久化。因而，下面是保存这两个对象所需要的所有代码：

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);

session.save(newUser);
```

Hibernate把一行插入到USERS表中，把另一行插入到ADDRESS表中。但是慢着，这不起作用！Hibernate怎么可能知道ADDRESS表中的记录要获取与USERS行相同的主键值？在本节开始时，我们有意不在Address的映射中介绍任何主键生成器。

你需要启用一个特殊的标识符生成器。

2．外标识符生成器

如果Address实例被保存，它就需要获取User对象的主键值。你无法启用一个常规的标识符生成器，假设是数据库序列。用于Address的这个特殊的foreign标识符生成器，必须知道从哪里获取正确的主键值。

创建Address和User之间的这个标识符绑定，其第一步是双向关联。把一个新的user属性添加到Address属性：

```
public class Address {
    ...
     private User user;
    // Getters and setters
}
```

在Address.hbm.xml中映射Address的这个新user属性：

```
<one-to-one name="user"
            class="User"
            constrained="true"/>
```

这个映射不仅使关联变成双向，而且通过constrained="true"，添加了把ADDRESS表的主键链接到USERS表主键的外键约束。换句话说，数据库保证了ADDRESS行的主键引用有效的USERS主键。（作为附带作用，Hibernate现在也可以在加载出货地址时启用用户的延迟加载。外键约束意味着用户必须对于特定的出货地址而存在，因此代理可以不用命中数据库而启用。如果没有这项约束，Hibernate就必须命中数据库去发现是否有对应于这个地址的用户；代理也将成为多余的了。后面的章节会回到这个话题。）

现在可以给Address对象使用特殊的foreign标识符生成器了：

```
<class name="Address" table="ADDRESS">

    <id name="id" column="ADDRESS_ID">
        <generator class="foreign">
            <param name="property">user</param>
        </generator>
    </id>
    ...
    <one-to-one name="user"
                class="User"
                constrained="true"/>

</class>
```

这个映射乍看起来似乎很奇怪。解读如下：当Address被保存时，从user属性中提取主键值。user属性是对User对象的一个引用；因而，插入的主键值与这个实例的主键值相同。看看图7-2中的表结构。

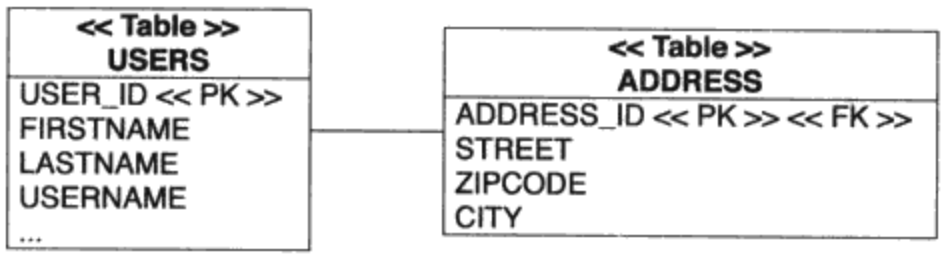

图7-2 USERS和ADDRESS表有相同的主键

保存这两个对象的代码现在必须考虑双向的关系，并且最终生效：

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);
shippingAddress.setUser(newUser);              // Bidirectional

session.save(newUser);
```

让我们利用注解完成同样的事情。

3. 利用注解的共享主键

JPA用@OneToOne注解支持一对一的实体关联。要映射User类中shippingAddress的关联为共享主键关联，还需要@PrimaryKeyJoinColumn注解：

```
@OneToOne
@PrimaryKeyJoinColumn
private Address shippingAddress;
```

在共享主键上创建单向的一对一关联就只需要这些。注意，如果通过复合主键映射，就要用@PrimaryKeyJoinColumns（复数）代替。在JPA XML描述符中，一对一的映射看起来像这样：

```
<entity-mappings>

    <entity class="auction.model.User" access="FIELD">
        ...
        <one-to-one name="shippingAddress">
            <primary-key-join-column/>
        </one-to-one>
    </entity>

</entity-mappings>
```

JPA规范没有包括处理共享主键生成问题的标准方法，这意味着在保存Address实例的标识符值（到被链接的User实例的标识符值）之前，你要负责正确地设置它。Hibernate有一个扩展注解，用于可以通过Address实体使用的定制标识符生成器（就像用XML格式一样）：

```
@Entity
@Table(name = "ADDRESS")
public class Address {
```

```
    @Id @GeneratedValue(generator = "myForeignGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "myForeignGenerator",
        strategy = "foreign",
        parameters = @Parameter(name = "property", value = "user")
    )
    @Column(name = "ADDRESS_ID")
    private Long id;

    ...
    private User user;
}
```

共享的主键一对一关联并不稀奇，但是相对少见。在许多模式中，对一（to-one）的关联都用外键字段和唯一约束来表示。

7.1.2　一对一的外键关联

不共享主键，而是两行可以有一个外键关系。一张表有着引用被关联表的主键的一个外键列。[这个外键约束的源和目标甚至可以是相同的表：称作自引用（self-referencing）关系。]

我们来改变从User到Address的映射。现在不用共享主键，而是在USERS表中添加一个SHIPPING_ADDRESS_ID列：

```
<class name="User" table="USERS">

    <many-to-one name="shippingAddress"
                 class="Address"
                 column="SHIPPING_ADDRESS_ID"
                 cascade="save-update"
                 unique="true"/>

</class>
```

在XML中，这个关联的映射元素是<many-to-one>——而不是你可能期待的<one-to-one>。原因很简单：你不在乎关联的目标端是什么，因此可以像对待没有多端的对一关联一样对待它。你所要的就是表达“这个实体有一个属性，该属性是对另一个实体的实例的引用”，并使用一个外键字段表示这个关系。这个映射的数据库模式如图7-3所示。

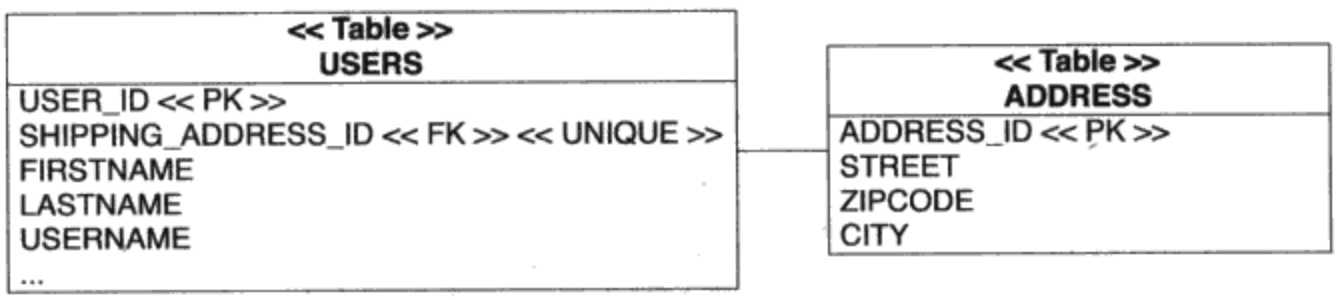

图7-3　USERS和ADDRESS之间的一对一外键关联

额外的约束强制这个关系为真正的一对一。通过使SHIPPING_ID列变成unique，声明一个特定的地址最多只能被一位用户作为出货地址使用。这不像来自共享主键关联（它允许一个特定的地址最多被一位用户引用）的保证那么有力。利用几个外键列（假设你也有唯一的HOME_

ADDRESS_ID和BILLING_ADDRESS_ID），可以多次引用相同的地址目标行。但是无论如何，两位用户无法把同一个地址用于同一目的。

我们来把从User到Address的关联变成双向。

1．反向属性引用

最后一个外键关联是用<many-to-one>从User映射到Address，并且unique约束保证了想要的多样性。可以在Address端添加什么映射元素使这个关联变成双向，以便在Java领域模型中从Address到User的访问成为可能呢？

在XML中，用属性引用（property reference）属性来创建一个<one-to-one>映射：

```
<one-to-one name="user"
            class="User"
            property-ref="shippingAddress"/>
```

你告诉Hibernate，Address类的user属性是关联另一端的一个属性的反向。现在可以调用anAddress.getUser()来访问给定出货地址的用户。没有额外的列或者外键约束；Hibernate替你管理这个指针。

应该使这个关联成为双向吗？跟往常一样，这由你决定，并取决于你是否需要在应用程序代码中的那个方向上导航对象。在这种情况下，我们或许可以推断双向关联没有多大意义。如果调用anAddress.getUser()，就等于在说“请把以这个地址作为出货地址的用户给我”，这不是一个非常合理的请求。我们建议，基于外键的一对一关联，以及在外键列中包含的唯一约束——通常要在另一端没有映射的情况下给予最好表示。

让我们通过注解重复相同的映射。

2．通过注解映射外键

JPA映射注解也支持基于外键列的实体之间的一对一关系。与本章前面的映射相比，主要区别在于用@JoinColumn代替了@PrimaryKeyJoinColumn。

首先，这是从User到Address的对一映射，在SHIPPING_ADDRESS_ID外键列中包含唯一约束。但是，它不用@ManyToOne注解，而是需要@OneToOne注解：

```
public class User {
    ...

    @OneToOne
    @JoinColumn(name="SHIPPING_ADDRESS_ID")
    private Address shippingAddress;

    ...
}
```

Hibernate现在将用唯一约束强制多样性。如果要使这个关联变成双向的，Address中还需要@OneToOne映射：

```
public class Address {
    ...

    @OneToOne(mappedBy = "shippingAddress")
    private User user;
```

```
    ...
}
```

mappedBy属性的作用与XML映射中的property-ref一样：关联的一个简单的反向声明，就在目标实体端指定了一种属性。

JPA XML描述器中与之相当的映射如下：

```
<entity-mappings>

    <entity class="auction.model.User" access="FIELD">
        ...
        <one-to-one name="shippingAddress">
            <join-column name="SHIPPING_ADDRESS_ID"/>
        </one-to-one>
    </entity>

    <entity class="auction.model.Address" access="FIELD">
        ...
        <one-to-one name="user" mapped-by="shippingAddress"/>
    </entity>

</entity-mappings>
```

你现在已经完成了两个基础的单端关联映射：第一个带有共享主键，第二个带有外键引用。我们想要讨论的最后一个选项更怪异一点：在另外一张表的帮助下映射一对一的关联。

7.1.3　用联结表映射

暂时放下复杂的CaveatEmptor模型，考虑一种不同的场景。想象你必须对表示一家公司的办公室配置规划的数据模式进行建模。一般的实体包括在办公桌前工作的人们。这似乎有道理：桌子可能被闲置着，没有分配给任何人员。另一方面，员工可能在家办公，结果也一样。你正在处理的就是Person和Desk之间可选的一对一关联。

如果应用前几节讨论的映射技术，可能会得出如下结论：Person和Desk被映射到两张表，其中一张（假设是PERSON表）有一个外键列，引用另一张带有额外唯一约束（以便两个人不会被分配到同一张桌子）的表（例如ASSIGN_DESK_ID）。如果外键列可为空，这个关系就是可选的。

第二种想法，你意识到人员和桌子之间的分配需用另一张表来表示ASSIGNMENT。在目前的设计中，这张表只有两个列：PERSON_ID和DESK_ID。这些外键列的多样性通过unique约束在这两个列上都得到了强制——特定的人员和桌子只能被分配一次，并且这样的分配只能有一次。

似乎也有可能某一天你会需要扩展这个Schema，并添加列到ASSIGNMENT表，例如当人员被分配到一张桌子时的日期。但是，只要不是这种情况，就可以利用ORM来隐藏中间的表，并创建仅仅两个类之间一对一的Java实体关联。（一旦ASSIGNMENT中引入了额外的列，这种情况就会完全改变。）

在CaveatEmptor中，这种可选的一对一关系在哪里呢?

1. CaveatEmptor用例

再次考虑CaveatEmptor中的Shipment实体，并讨论它的用途。卖主和买主在CaveatEmptor中通过在拍卖中标价和出价进行交互。货物的出货似乎在应用程序的范围之外；卖主和买主在拍卖结束之后认同一种出货和付款的方法。他们可以离线（即在CaveatEmptor之外）进行这些事。另一方面，可以在CaveatEmptor中提供一种额外的*有条件转让契约服务*（escrow service）。一旦拍卖完成，卖主可以利用这项服务创建可追踪的出货。买主可以支付拍卖货品的价钱给托管人（你），并且你要通知卖主钱已到账。一旦货物抵达，买主接受了，你就要把钱转给卖主。

如果你曾经参与过重大价值的在线拍卖，可能已经使用过这么一种有条件转让契约服务。但是在CaveatEmptor中还要更多的服务。不仅要对已完成的拍卖提供信任服务，还要允许用户给他们在拍卖之外（即CaveatEmptor之外）进行的任何交易创建可追踪的信任出货。

这种场景需要Shipment实体，包含可选的关联到Item的一对一关联。请看图7-4中这个领域模型的类图。

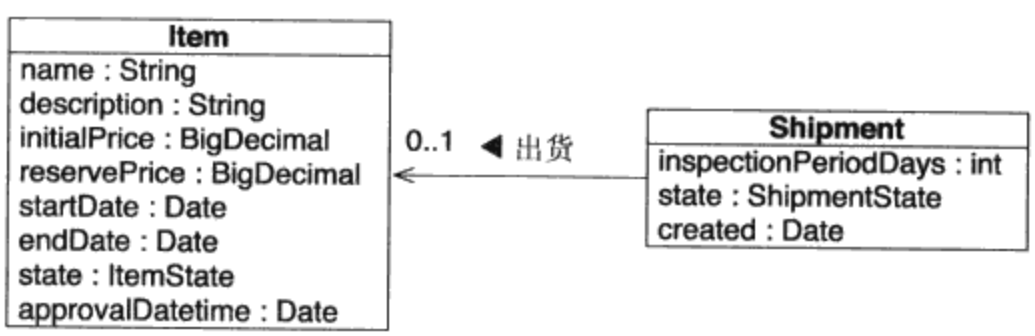

图7-4 出货与单件拍卖货品有可选的链接

在数据库Schema中，添加了一个中间的链接表，称作ITEM_SHIPMENT。这张表中的行表示在一次拍卖的上下文中产生的Shipment。这些表如图7-5所示。

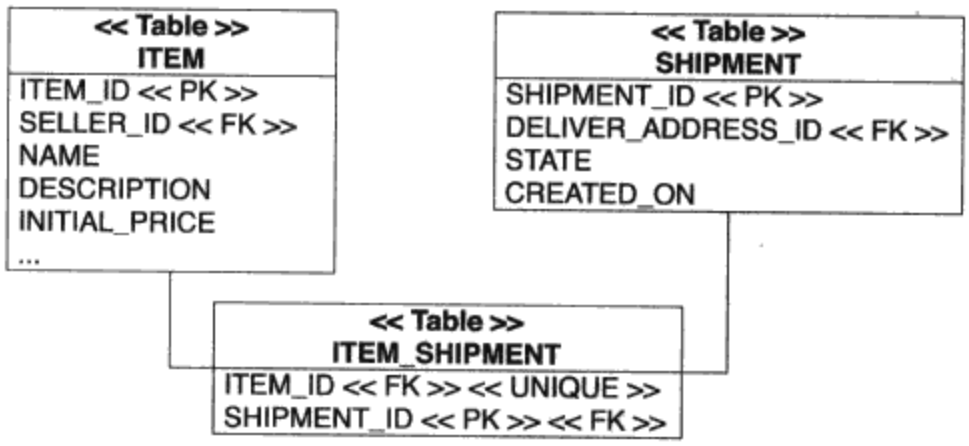

图7-5 被映射到联结表的可选的一对多关系

现在把两个类映射到三张表：先用XML映射，然后通过注解映射。

2. 用XML映射联结表

表示从Shipment到Item关联的属性称作auction：

```
public class Shipment {

    ...
```

```
    private Item auction;
    ...
    // Getter/setter methods
}
```

因为必须用一个外键列映射这个关联，因此在XML中你需要<many-to-one>映射元素。然而，外键列不在SHIPMENT表中，而在ITEM_SHIPMENT联结表中。通过<join>映射元素把它移到那里。

```
<class name="Shipment" table="SHIPMENT">

    <id name="id" column="SHIPMENT_ID">...</id>

    ...

    <join table="ITEM_SHIPMENT" optional="true">
        <key column="SHIPMENT_ID"/>
        <many-to-one name="auction"
                     column="ITEM_ID"
                     not-null="true"
                     unique="true"/>
    </join>

</class>
```

联结表有两个外键列：SHIPMENT_ID（引用SHIPMENT表的主键）和ITEM_ID（引用ITEM表）。ITEM_ID列是唯一的；特定的货品正好可以被分配给一批出货。由于联结表的主键是SHIPMENT_ID，它使得这个列也变成唯一，这样就保证了Shipment和Item之间一对一的多样性。

通过在<join>映射中设置optional="true"，告诉Hibernate它应该只有当这个映射分组的属性为非空时，才把行插入到联结表中。但是如果必须插入行（因为调用了aShipment.setAuction(anItem)），ITEM_ID列中的NOT NULL约束就得到了应用。

可以在另一端使用相同的方法，把这个关联映射为双向。然而，可选的一对一关联大多数时候都是单向的。

JPA也对实体支持作为二级表（secondary table）的关联联结表。

3. 通过注解映射二级联结表

可以通过注解把可选的一对一关联映射到一个中间的联结表：

```
public class Shipment {

    @OneToOne
    @JoinTable(
        name="ITEM_SHIPMENT",
        joinColumns = @JoinColumn(name = "SHIPMENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    private Item auction;
    ...
    // Getter/setter methods
}
```

不必指定SHIPMENT_ID列，因为它自动被认为是联结列；它是SHIPMENT表的主键列。

另一种方法，可以把JPA实体的属性映射到不止一张表，如8.1.3节所述。首先，需要给实体

声明二级表：

```
@Entity
@Table(name = "SHIPMENT")
@SecondaryTable(name = "ITEM_SHIPMENT")
public class Shipment {

    @Id @GeneratedValue
    @Column(name = "SHIPMENT_ID")
    private Long id;

    ...
}
```

注意，@SecondaryTable注解也支持用属性声明外键列名——相当于前面用XML的<key column="..."/>和@JoinTable中的joinColumn(s)。如果没有指定，就使用实体的主键列名——在这个例子中，还是SHIPMENT_ID。

auction属性映射是一个@OneToOne；并且和之前一样，引用ITEM表的外键列被移到中间的二级表：

```
...
public class Shipment {
    ...
    @ManyToOne
    @JoinColumn(table = "ITEM_SHIPMENT", name = "ITEM_ID")
    private Item auction;
}
```

用于目标@JoinColumn的表被显式地指定。为什么要使用这种方法而不是（更简单的）@JoinTable策略呢？如果不止一个属性而是多个属性必须被移到二级表的时候，给实体声明二级表就很有用。我们没有使用Shipment和Item的很好示例，但是如果你的ITEM_SHIPMENT表有额外的列，把这些列映射到Shipment实体的属性可能有用。

这样就结束了我们对一对一关联映射的讨论。总而言之，如果两个实例中有一个似乎更加重要，并且可以表现得像主键源一样，就使用共享的主键关联。在所有其他的情况下都使用外键关联，并且当一对一关联为可选时，就使用被隐藏的中间联结表。

现在来关注多值的实体关联，包括一对多以及最后是多对多映射的更多选项。

7.2 多值的实体关联

多值（many-valued）的实体关联就其定义而言，是指实体引用的一个集合。6.4节中映射过其中一种。父实体实例有一个对多个子对象的引用集合——因而，是一对多。

一对多关联是涉及集合的一种最重要的实体关联。如果简单的双向多对一或一对多能够完成任务时，目前为止我们并不鼓励使用更加怪异的关联方式。多对多关联始终可以表示为对中间类的两个多对一关联。这个模型通常更易于扩展，因此我们趋向于不在应用中使用多对多关联。也要记住：如果不想的话，你不必映射实体的任何集合；你可以始终编写显式查询来代替通过迭代的直接访问。

如果你决定要映射实体引用的集合，我们现在就来讨论几种选项以及更复杂的情况（包括多

对多的关系）。

7.2.1　一对多关联

前面映射过的父/子关系是一个双向的关联，通过<one-to-many>和<many-to-one>映射。这个关联的多（many）端在Java中用Set实现；Item类中有一个bids的集合。

来重新思考这个映射，并关注一些特殊的情况。

1. 考虑bag

对双向的一对多关联使用<bag>映射而不是set（集）是可能的。为什么要这什么做呢？

包（bag）有着可以用于双向一对多实体关联的所有集合的最有效的性能特征（换句话说，如果集合端是is inverse="true"的话）。默认情况下，Hibernate中的集合只有当它们在应用程序中第一次被访问时才被加载。因为bag不必维持其元素的索引（如list），或者检查重复的元素（如set），可以添加新元素给bag，而不触发加载。如果要映射一个可能很大的实体引用的集合，这就是一项重要的特性。另一方面，无法同时即时抓取bag类型的两个集合（例如，如果Item的bids和images都是一对多的bag）。我们会在13.1节中回到抓取策略的主题。一般来说，对于一对多的关联，我们认为bag是它最好的反向集合。

为了把双向的一对多关联映射为bag，必须用Collection和ArrayList实现替换Item持久化类中bids集合的类型。对Item和Bid之间关联的映射本质上保持不变：

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

</class>
<class name="Item"
       table="ITEM">
    ...
    <bag name="bids"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </bag>

</class>
```

重新命名<set>元素为<bag>，没有进行其他的改变。甚至表也是一样的：BID表有ITEM_ID外键列。在JPA中，所有Collection和List属性都被认为具有包语义，因此下列代码相当于XML映射：

```
public class Item {
    ...

    @OneToMany(mappedBy = "item")
```

```
    private Collection<Bid> bids = new ArrayList<Bid>();
    ...
}
```

bag也允许重复元素，这是你之前映射过的set所没有的。结果表明，这与本例无关，因为重复意味着你已经几次把一个特定的引用添加到相同的Bid实例。不要在应用程序代码中这么做。但是即使几次把相同的引用添加到这个集合，Hibernate也会忽略它——它被反向映射。

2. 单向和双向的列表

如果你需要真正的列表来保存元素在集合中的位置，就必须在另一列保存该位置。对于一对多的映射，这也意味着应该把Item中的bids属性改为List，并通过ArrayList初始化变量（或者如果不想把这种行为公开给类的客户端，就保留来自前一节的Collection接口）。

在Item的映射中，保存对Bid实例引用的位置的额外列是BID_POSITION：

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>
</class>
```

目前为止，这些似乎都很简单；你已经把集合映射改为<list>，并添加了<list-index>列BID_POSITION到集合表（在这个例子中是BID表）。用图7-6中所示的表进行检验。

BID

BID_ID	ITEM_ID	BID_POSITION	AMOUNT	CREATED_ON
1	1	0	99.00	19.04.08 23:11
2	1	1	123.00	19.04.08 23:12
3	2	0	433.00	20.04.08 09:30

图7-6 在列表集合中保存每个出价的位置

这个映射并没有真正完成。考虑ITEM_ID外键列：它为NOT NULL（出价必须引用货品）。第一个问题在于没有在映射中指定这个约束。而且，由于这个映射是单向的（集合端为非反向），所以只好假设没有对面端被映射到同一个外键列（可以在那里声明这个约束）。需要把not-null="true"属性添加到集合映射的<key>元素中：

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID" not-null="true"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>
</class>
```

注意，属性必须在<key>映射中，而不是在一个可能嵌套的<column>元素中。每当有实体引用的非反向集合（大多数时候是包含list、map或者array的一对多关联），并且目标表中的外键联结列不可为空时，就需要让Hibernate知道这些。Hibernate需要这个提示，以便正确地处理INSERT和UPDATE语句，避免约束冲突。

用Bid的item属性使这个变成双向。如果理解了前面章节中的例子，可能会想要在ITEM_ID外键列上添加<many-to-one>，使这个关联变成双向的，并在集合中启用inverse="true"。记住，Hibernate忽略反向集合的状态！但是这一次，集合包含了正确地更新数据库所需的信息：它的元素的位置。如果只有每个Bid实例的状态被认为是同步的，集合又是反向的并且被忽略，那么Hibernate就没有值给BID_POSITION列了。

如果通过被索引的集合映射双向的一对多实体关联（映射和数组也是这样），就必须转换反向端。无法使被索引的集合变成inverse="true"。集合变成了负责状态同步，并且一（one）端Bid必须变成反向。然而，多对一映射没有inverse="true"，因此需要在<many-to-one>中模拟这一属性：

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

设置insert和update为false具有预期的效果。如前所述，这两个属性一起使用，实际上使属性变成了只读（read-only）。关联的这一端因此被任何写操作忽略，当内存状态与数据库同步时，集合的状态（包括元素的索引）就是相关的状态。你已经转换了关联的反向／非反向端，如果从set或者bag转换到list（或者任何其他被索引的集合），这是个必要的条件。

JPA中的等价物（双向的一对多映射中的被索引集合）如下：

```
public class Item {
    ...

    @OneToMany
    @JoinColumn(name = "ITEM_ID", nullable = false)
    @org.hibernate.annotations.IndexColumn(name = "BID_POSITION")
    private List<Bid> bids = new ArrayList<Bid>();

    ...

}
```

这个映射为非反向的，因为没有出现mappedBy属性。由于JPA不支持持久化的被索引列表（只有在加载时用@OrderBy进行排序），需要给索引支持添加一个Hibernate扩展注解。以下是Bid中关联的另一端：

```
public class Bid {
    ...

    @ManyToOne
    @JoinColumn(name = "ITEM_ID", nullable = false,
                updatable = false, insertable = false)
    private Item item;

    ...

}
```

现在再讨论一种包含一对多关系的场景：被映射到一个中间的联结表的关联。

3. 利用联结表的可选一对多关联

Item类增加了一个有用的buyer属性。可以调用anItem.getBuyer()访问出价胜出的用户。（当然，anItem.getSuccessfulBid().getBidder()可以通过不同的路径提供相同的访问。）如果变成双向的，这个关联也将有助于呈现屏幕，显示特定用户已经胜出的所有拍卖：调用aUser.getBoughtItems()来代替编写查询。

从User类的观点来看，这个关联是一对多的。类及其关系如图7-7所示。

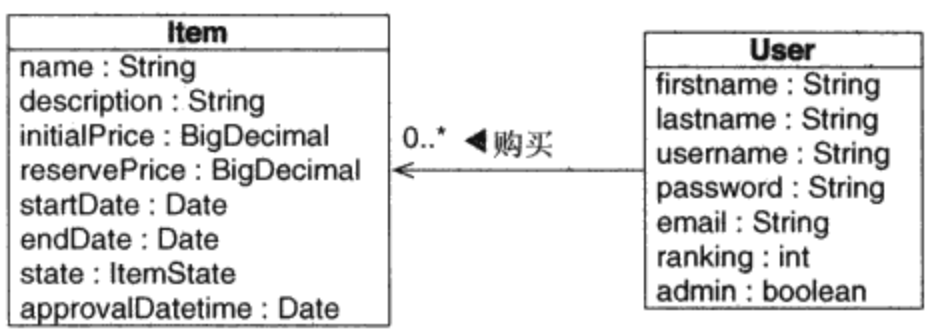

图7-7 用户可以购买货品

为什么这个关联与Item和Bid之间的不同？在UML中，多样性0..*表明引用是可选的。这不太影响Java领域模型，但是对底层的表有影响。你希望在ITEM表中有个BUYER_ID外键列。列必须是可以为空的——不可能已经购买了特定的Item（只要拍卖仍然在进行）。

可以接受外键列为NULL，并应用额外的约束（“仅当还没到拍卖终止时间或者还没有出价时才允许为NULL”）。我们总是努力在关系数据库Schema中避免可为空的列。未知的信息降低了所保存数据的质量。元组（tuple）表示那些为真（true）的命题，我们无法断言不知道的东西。并且，在实际应用程序中，许多开发人员和数据库管理员没有创建正确的约束，并依赖（经常出现bug的）应用程序代码提供数据的完整性。

可选的实体关联（一对一或者一对多）在SQL数据库中用联结表表示得最好。示例Schema请见图7-8。

本章前面给一对一关联添加过一张联结表。为了确保一对一的多样性，在联结表的两个外键列上都应用了unique约束。在目前的情形中，你拥有一对多的多样性，因此只有ITEM_BUYER表的ITEM_ID列是唯一的。一个特定的货品只可以被购买一次。

让我们用XML映射它。首先，是User类的boughtItems集合。

```
<set name="boughtItems" table="ITEM_BUYER">
    <key column="USER_ID"/>
    <many-to-many class="Item"
                  column="ITEM_ID"
                  unique="true"/>
</set>
```

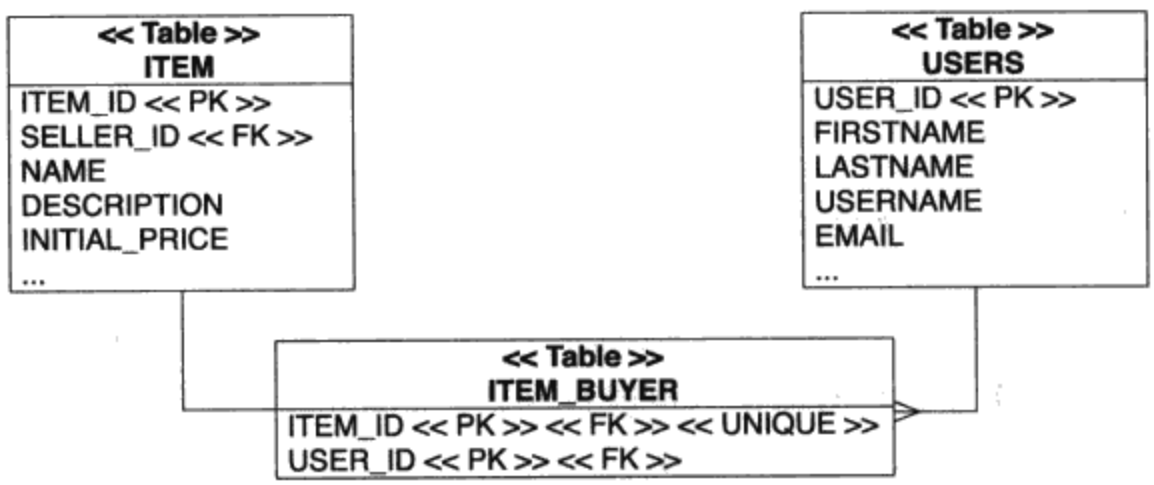

图7-8　使用联结表的可选关系避免了可为空的外键列

用Set作为集合类型。集合表是联结表ITEM_BUYER；它的主键是USER_ID和ITEM_ID的复合。你之前还没见过的新映射元素是<many-to-many>；因为常规的<one-to-many>不知道任何有关联结表的信息，因此需要它。通过在引用目标实体表的外键列上强制unique约束，有效地强制了一对多的多样性。

可以用Item的buyer属性把这个关联映射为双向的。若没有联结表，就要在ITEM表中用BUYER_ID外键列添加<many-to-one>。若有联结表，则必须把这个外键列移到联结表里面。这可能通过<join>映射：

```
<join table="ITEM_BUYER"
      optional="true"
      inverse="true">
    <key column="ITEM_ID" unique="true" not-null="true"/>
    <many-to-one name="buyer" column="USER_ID"/>
</join>
```

这里有两个重要的细节：首先，关联是可选的，你告诉Hibernate如果被分组的属性（这里只有一个属性buyer）为null，就不要把行插入到联结表。第二，这是个双向的实体关联。与往常一样，一端必须为反向的。你已经选择了<join>为反向的；现在Hibernate利用集合状态使数据库同步，并忽略Item.buyer属性的状态。只要集合不是被索引的变量（list、map或者array），就可以通过声明集合inverse="true"来反向这一点。创建购买的货品和用户对象之间链接的Java代码在这两种情况下相同：

```
aUser.getBoughtItems().add(anItem);
anItem.setBuyer(aUser);
```

可以映射JPA中的二级表来创建使用联结表的一对多关联。首先，把@ManyToOne映射到联结表：

```
@Entity
public class Item {
    @ManyToOne
    @JoinTable(
        name = "ITEM_BUYER",
        joinColumns = {@JoinColumn(name = "ITEM_ID")},
        inverseJoinColumns = {@JoinColumn(name = "USER_ID")}
    )
    private User buyer;
    ...
}
```

在编写本书之时，这个映射还有个限制：不能把它设置为optional="true"；因而，USER_ID列是可为空的。如果试图在@JoinColumn上添加nullable="false"，Hibernate Annotations就认为你要整个buyer属性永远不为null。此外，联结表的主键现在只有ITEM_ID列。这样很好，因为你不想在这个表中有重复的货品——它们只能被购买一次。

为了使这个映射变成双向的，要在User类上添加一个集合，并用mappedBy使它变成反向：

```
@OneToMany(mappedBy = "buyer")
private Set<Item> boughtItems = new HashSet<Item>();
```

我们在前一节给联结表中的一对多关联介绍过<many-to-many>XML映射元素。@JoinTable注解是注解中的等价物。来映射一个真正的多对多关联。

7.2.2 多对多关联

Category和Item之间的关联是多对多关联，如图7-9所示。

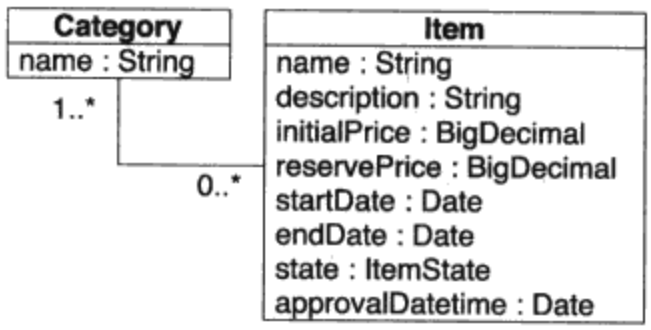

图7-9 Category和Item之间多对多的值关联

在真实的系统中，可能没有多对多关联。依我们的经验，通常有其他信息必须被附加到被关联实例之间的每一个链接（例如货品被添加到目录时的日期和时间，表示这一信息的最好方式是通过中间的关联类（association class）。在Hibernate中，可以把这个关联类映射为实体，并给任何一端映射两个一对多的关联。可能更方便的做法是，也可以映射一个复合的元素类，这是我们后面要介绍的一种方法。实现一个真正的多对多实体关联是本节的目的。让我们从一个单向的示例开始。

1. 简单的单向多对多关联

如果只需要单向导航，映射就很简单。单向的多对多关联本质上不比我们之前讨论过的值类型实例的集合更难。例如，如果Category有一组Items，你就可以创建这个映射：

```
<set name="items"
     table="CATEGORY_ITEM"
     cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
```

联结表（也被有些开发人员称为链接表，link table）有两个列：CATEGORY和ITEM表的外键。主键是这两列的复合。完整的表结构如图7-10所示。

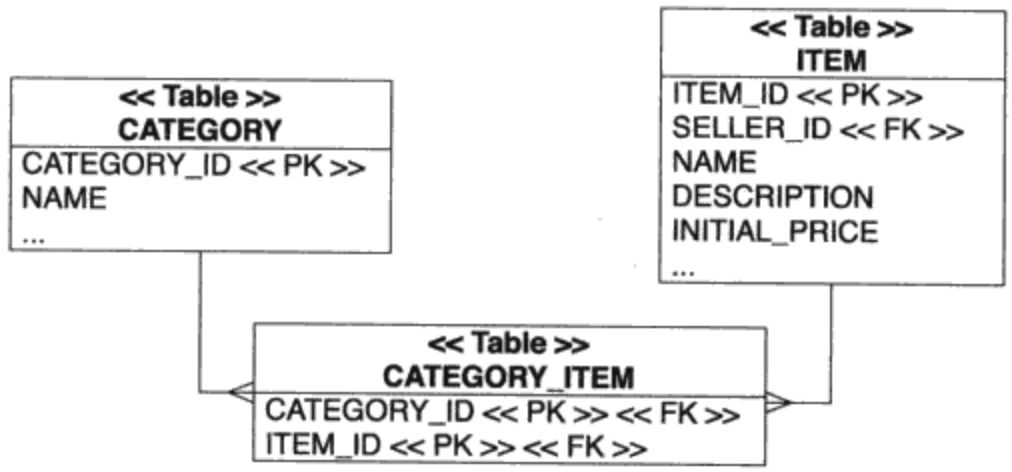

图7-10　被映射到关联表的多对多实体关联

在JPA注解中，多对多的关联通过@ManyToMany属性映射：

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```

在Hibernate XML中，也可以在联结表中通过一个单独的主键列切换到<idbag>：

```
<idbag name="items"
       table="CATEGORY_ITEM"
       cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
```

与通常使用<idbag>映射一样，主键是一个代理键列CATEGORY_ITEM_ID。因此允许重复的链接；同一个Item可以被添加到Category两次。（这似乎不是一项有用的特性。）利用注解，可以通过Hibernate @CollectionId切换到一个标识符bag：

```
@ManyToMany
@CollectionId(
    columns = @Column(name = "CATEGORY_ITEM_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = "sequence"
```

```
)
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Collection<Item> items = new ArrayList<Item>();
```

使用集的一般多对多映射的JPA XML描述符（你不能给标识符bag使用Hibernate扩展）看起来像这样：

```
<entity class="auction.model.Category" access="FIELD">
    ...
    <many-to-many name="items">
        <join-table name="CATEGORY_ITEM">
            <join-column name="CATEGORY_ID"/>
            <inverse-join-column name="ITEM_ID"/>
        </join-table>
    </many-to-many>

</entity>
```

甚至可以在多对多关联中切换到一个被索引的集合（map或者list）。下列例子在Hibernate XML中映射了一个list：

```
<list name="items"
      table="CATEGORY_ITEM"
      cascade="save-update">
    <key column="CATEGORY_ID"/>
    <list-index column="DISPLAY_POSITION"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</list>
```

链接表的主键是CATEGORY_ID和DISPLAY_POSITION列的复合；这个映射保证每个Item在Category中的位置都是持久化的。或者，利用注解：

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
@org.hibernate.annotations.IndexColumn(name = "DISPLAY_POSITION")
private List<Item> items = new ArrayList<Item>();
```

如前所述，JPA只支持有序集合（通过一个可选的@OrderBy注解或者按主键排列），因此必须再次对被索引的集合支持使用Hibernate扩展。如果没有添加@IndexColumn，List就会通过bag语义保存（不保证元素的持久化顺序）。

创建Category和Item之间的链接很容易：

```
aCategory.getItems().add(anItem);
```

双向的多对多关联稍微难一些。

2. 双向的多对多关联

你知道，双向关联的一端必须被映射为反向，因为你已经对（一个或多个）外键列命名了两次。给双向的多对多关联应用同样的原则：链接表的每一行都由两个集合元素表示，关联的两端各一个元素。Item和Category之间的关联，在内存中由Category的items集合中Item实例表示，而且还通过Item的Categories集合中Category实例表示。

在讨论这个双向案例的映射之前，必须知道创建对象关联的代码也变了：

```
aCategory.getItems().add(anItem);
anItem.getCategories().add(aCategory);
```

如往常一样，双向关联（无论什么多样性）要求关联的两端都要设置。

映射双向的多对多关联时，必须用inverse="true"声明关联的一端，来定义哪一端的状态用来更新联结表。可以选择哪一端应该为反向。

回想前一节items集合中的这个映射：

```
<class name="Category" table="CATEGORY">
    ...
    <set name="items"
         table="CATEGORY_ITEM"
         cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
```

可以给双向关联的Category端重用这个映射，并映射另一端如下：

```
<class name="Item" table="ITEM">
    ...
    <set name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

注意inverse="true"。这个设置再次告诉Hibernate忽略对categories集合所做的改变，并且如果在Java代码中链接实例，关联的另一端（items集合）就是应该与数据库同步的表示法。

你已经对集合的两端都启用了cascade="save-update"。我们推想这是不切实际的。另一方面，级联选项all、delete和delete-orphans对于多对多的关联都是没有意义的。（这是测试你是否理解实体和值类型的一个好机会——试着想出合理的答案，为什么这些级联选项对于多对多的关联没有意义。）

在JPA中，使用注解，可以很容易地使多对多的关联变成双向。首先是非反向端：

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```

现在是对面的反向端：

```
@ManyToMany(mappedBy = "items")
private Set<Category> categories = new HashSet<Category>();
```

如你所见，并不一定要在反向端重复联结表声明。

哪些集合类型可以用于双向的多对多关联？需要在每一端使用相同的集合类型吗？例如，给关联的非反向端映射使用<list>、给反向端映射使用<bag>是合理的。

对于反向端，可以接受<set>，下列bag映射也一样：

```
<class name="Item" table="ITEM">
    ...
    <bag name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

在JPA中，bag是没有持久化索引的一个集合：

```
@ManyToMany(mappedBy = "items")
private Collection<Category> categories = new ArrayList<Category>();
```

没有其他的映射可以用于多对多关联的反向端。被索引的集合（list和map）不行，因为如果集合为反向，Hibernate将不会初始化或者维持索引列。换句话说，无法通过被索引的集合在两端映射多对多的关联。

我们已经反对多对多关联的使用，因为联结表中的额外列通常是不可避免的。

7.2.3 把列添加到联结表

本节将讨论Hibernate用户经常问的一个问题：如果联结表有一些额外的列，而不只是两个外键列时，该怎么办？

想象在每次把Item添加到Category时都要记录一些信息。例如，可能要保存日期，以及把该货品添加到这个目录的用户名称。这在联结表中就需要额外的列，如图7-11所示。

可以用两种常用的策略把这样一种结构映射到Java类。第一种策略需要一个用于联结表的中间实体类，并通过一对多的关联而被映射。第二种策略通过给联结表使用一个值类型的类来利用组件的集合。

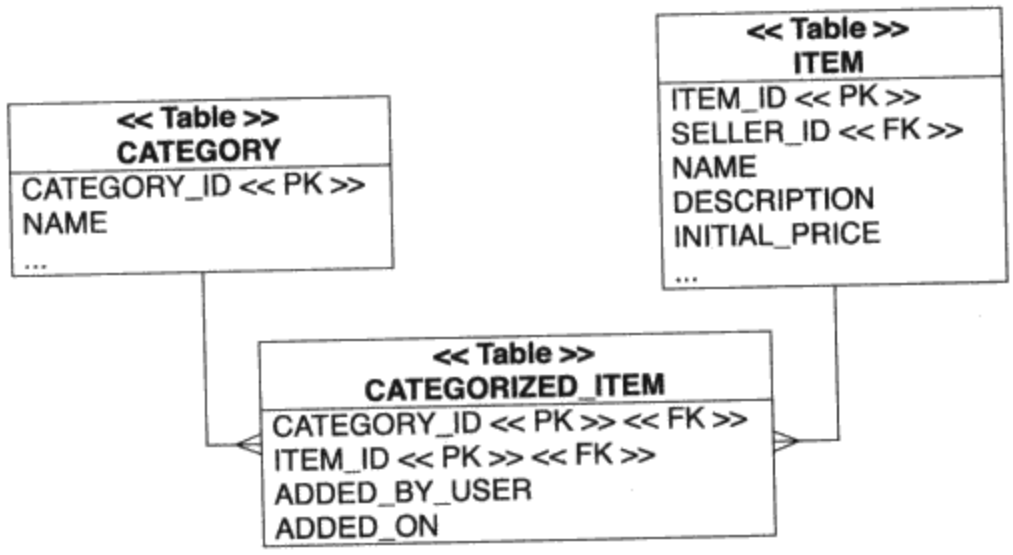

图7-11　多对多关联中联结表上的额外列

1. 把联结表映射到中间实体

我们现在讨论的第一个选项解决了Category和带有中间实体类(CategorizedItem)的Item之间的多对多关系。代码清单7-1展现了这个实体类，它在Java中表示联结表（包括JPA注解）：

代码清单7-1　表示包含额外列的链接表的实体类

```
@Entity
@Table(name = "CATEGORIZED_ITEM")
public class CategorizedItem {

    @Embeddable
    public static class Id implements Serializable {

        @Column(name = "CATEGORY_ID")
        private Long categoryId;

        @Column(name = "ITEM_ID")
        private Long itemId;

        public Id() {}

        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }
        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
                Id that = (Id)o;
                return this.categoryId.equals(that.categoryId) &&
                       this.itemId.equals(that.itemId);
            } else {
                return false;
            }
        }
        public int hashCode() {
            return categoryId.hashCode() + itemId.hashCode();
        }
    }
```

```
@EmbeddedId
private Id id = new Id();

@Column(name = "ADDED_BY_USER")
private String username;

@Column(name = "ADDED_ON")
private Date dateAdded = new Date();

@ManyToOne
@JoinColumn(name="ITEM_ID",
            insertable = false,
            updatable = false)
private Item item;

@ManyToOne
@JoinColumn(name="CATEGORY_ID",
            insertable = false,
            updatable = false)
private Category category;
      public CategorizedItem() {}

      public CategorizedItem(String username,
                             Category category,
                             Item item) {
          // Set fields
          this.username = username;

          this.category = category;
          this.item = item;

          // Set identifier values
          this.id.categoryId = category.getId();
          this.id.itemId = item.getId();

          // Guarantee referential integrity
          category.getCategorizedItems().add(this);
          item.getCategorizedItems().add(this);
      }

      // Getter and setter methods
      ...
}
```

实体类需要一个标识符属性。联结表的主键是CATEGORY_ID和ITEM_ID的复合。因而，实体类也有一个复合键，为了方便起见，把它封装在一个静态的嵌套类中了。你还可以看到构建CategorizedItem涉及了标识符值的设置——复合键值由应用程序分配。要特别注意这个构造器，以及它如何设置字段值，并通过在关联的任意一端管理集合来确保参照完整性。

在XML中把这个类映射到联结表：

```
<class name="CategorizedItem"
       table="CATEGORY_ITEM"
       mutable="false">

    <composite-id name="id" class="CategorizedItem$Id">
        <key-property name="categoryId"
```

```
                    access="field"
                    column="CATEGORY_ID"/>

        <key-property name="itemId"
                    access="field"
                    column="ITEM_ID"/>
    </composite-id>

    <property name="dateAdded"
              column="ADDED_ON"
              type="timestamp"
              not-null="true"/>

    <property name="username"
              column="ADDED_BY_USER"
              type="string"
              not-null="true"/>

    <many-to-one name="category"
                 column="CATEGORY_ID"
                 not-null="true"
                 insert="false"
                 update="false"/>

    <many-to-one name="item"
                 column="ITEM_ID"
                 not-null="true"
                 insert="false"
                 update="false"/>
</class>
```

实体类被映射为不可变——你创建之后永远不用更新任何属性。Hibernate直接访问<composite-id>字段——在这个被嵌套的类中不需要获取方法和设置方法。这两个<many-to-one>映射实际上是只读的：insert和update被设置为false。这是必需的，因为列被映射两次：一次在复合键中（负责值的插入），另一次用于多对一的关联。

Category和Item实体（可以）有对CategorizedItem实体（一个集合）的一对多关联。例如，在Category中：

```
<set name="categorizedItems"
     inverse="true">
    <key column="CATEGORY_ID"/>
    <one-to-many class="CategorizedItem"/>
</set>
```

下面是注解的等价物：

```
@OneToMany(mappedBy = "category")
private Set<CategorizedItem> categorizedItems =
                                    new HashSet<CategorizedItem>();
```

这里没有什么特别的东西要考虑；它是个包含反向集合的一般的双向一对多关联。添加相同的集合和映射到Item来完成关联。创建和保存目录和货品之间链接的代码如下：

```
CategorizedItem newLink =
    new CategorizedItem(aUser.getUsername(), aCategory, anItem);

session.save(newLink);
```

Java对象的参照完整性受到CategorizedItem的构造器的保证，该构造器管理aCategory和anItem中的集合。移除和删除目录和货品之间的链接：

```
aCategory.getCategorizedItems().remove( theLink );
anItem.getCategorizedItems().remove( theLink );

session.delete(theLink);
```

这个策略的主要好处在于双向导航的可能性：可以通过调用aCategory.getCategorizedItems()获取目录中的所有货品，并用anItem.getCategorizedItems()从反向导航。缺点是要用更复杂的代码管理CategorizedItem实体实例，创建和移除关联——它们必须被独立地被保存和删除，并且在CategorizedItem类中需要一些基础结构，例如复合标识符。但是，可以用集合中的级联选项启用传播性持久化，从Category和Item到CategorizedItem，如12.1节中所述。

处理联结表中额外列的第二种策略不需要中间的实体类；它更为简单。

2. 把联结表映射到组件的集合

首先，简化CategorizedItem类，并使它成为值类型，不用标识符或者任何复杂的构造器：

```
public class CategorizedItem {
    private String username;
    private Date dateAdded = new Date();
    private Item item;
    private Category category;

    public CategorizedItem(String username,
                           Category category,
                           Item item) {
        this.username = username;
        this.category = category;
        this.item = item;
    }
    ...

    // Getter and setter methods
    // Don't forget the equals/hashCode methods
}
```

至于所有的值类型，这个类必须为一个实体所有。所有者是Category，它有这些组件的一个集合：

```
<class name="Category" table="CATEGORY">

    ...
    <set name="categorizedItems" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <composite-element class="CategorizedItem">
            <parent name="category"/>

            <many-to-one name="item"
                         column="ITEM_ID"
                         not-null="true"
                         class="Item"/>

            <property name="username" column="ADDED_BY_USER"/>
            <property name="dateAdded" column="ADDED_ON"/>
```

```
        </composite-element>
    </set>

</class>
```

这是对于联结表中包含额外列的多对多关联的完整映射。<many-to-one>元素表示对Item的关联；<property>映射涵盖了联结表中的额外列。数据库表中只有一处变化：现在CATEGORY_ITEM表有一个主键，它是所有列的复合，而不仅仅是CATEGORY_ID和ITEM_ID，就像前一节的一样。因此，所有属性都永远不应该为空的——否则就无法识别联结表中的行了。除了这个变化之外，这些表看起来仍然如图7-11中所示的一样。

你可以通过对User而不只是用户名的引用来增强这个映射。这要求联结表中要有一个额外的USER_ID列，包含USERS的一个外键。这是一个三重关联（ternary association）映射：

```
<set name="categorizedItems" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>

        <many-to-one name="item"
                     column="ITEM_ID"
                     not-null="true"
                     class="Item"/>

        <many-to-one name="user"
                     column="USER_ID"
                     not-null="true"
                     class="User"/>

        <property name="dateAdded" column="ADDED_ON"/>

    </composite-element>
</set>
```

这是一段相当怪异的代码！

组件集合的好处无疑是链接对象的隐式生命周期。为了创建Category和Item之间的关联，就把新的CategorizedItem实例添加到集合。要解除链接，就从集合中移除该元素。不需要额外的级联设置，并且Java代码也被简化了：

```
CategorizedItem aLink =
    new CategorizedItem(aUser.getUserName(), aCategory, anItem);

aCategory.getCategorizedItems().add( aLink );

aCategory.getCategorizedItems().remove( aLink );
```

这种方法的缺点在于无法启用双向导航：根据定义，组件（例如CategorizedItem）不可能有共享的引用。不能从Item导航到CategorizedItem。但是，你可以编写一个查询来获取所需的对象。

让我们用注解进行相同的映射。首先，使组件类变成@Embeddable，并添加组件列和关联映射：

```
@Embeddable
public class CategorizedItem {

    @org.hibernate.annotations.Parent // Optional back-pointer
    private Category category;
```

```
    @ManyToOne
    @JoinColumn(name = "ITEM_ID",
                nullable = false,
                updatable = false)
    private Item item;

    @ManyToOne
    @JoinColumn(name = "USER_ID",
                nullable = false,
                updatable = false)
    private User user;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "ADDED_ON", nullable = false, updatable = false)
    private Date dateAdded;

    ...
    // Constructor
    // Getter and setter methods
    // Don't forget the equals/hashCode methods
}
```

现在把这个CategorizedItem类映射为Category类中组件的集合：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = @JoinColumn(name = "CATEGORY_ID")
)
private Set<CategorizedItem> categorizedItems =
                                    new HashSet<CategorizedItem>();
```

就是这样：你已经通过注解映射了一个三重关联。一开始看起来复杂到难以置信的东西，现在已经被减少成了几行注解元数据，它们中大部分是可选的。

我们要探讨的最后一个集合映射是实体引用的Map。

7.2.4 映射 map

上一章映射了Java Map——Map的键和值都是值类型，它们是简单的字符串。可以创建更复杂的map：不仅键可以是对实体的引用，值也可以。因此结果可以是三重关联。

1. 值对实体的引用

首先，假设只有每个映射项的值才是对另一个实体的引用。键是个值类型：long。想象Item实体有一个Bid实例的map，并且每个映射项都是一对Bid标识符与对Bid实例的引用。如果通过anItem.getBidsByIdentifier()迭代，便通过映射项如(1, <reference to Bid with PK 1>), (2,<reference to Bid with PK 2>)等进行迭代。

这个映射的底层表没有什么特别的：你再次有了ITEM和BID表，BID表中有ITEM_ID外键列。此处的目的是在应用程序中用一个Map对数据进行稍微不同的表示。

Item类中包括一个Map：

```
@MapKey(name="id")
@OneToMany(mappedBy="item")
private Map<Long,Bid> bidsByIdentifier = new HashMap<Long,Bid>();
```

此处新出现的东西是JPA的@MapKey元素——它把目标实体的一个属性映射为该映射的键。如果省略name属性，则默认为目标实体的标识符属性（因此这里的名称是多余的）。由于映射的多个键形成了一个集，因此一个特定映射的值希望是唯一的——这是对于Bid主键，但是对于Bid的任何其他属性则可能不是如此。

在Hibernate XML中，这个映射如下：

```
<map name="bidsByIdentifier"inverse="true">
    <key column="ITEM_ID"/>
    <map-key type="long" formula="BID_ID"/>
    <one-to-many class="Bid"/>
</map>
```

映射的formula键使这个列变成了只读，因此当你修改该映射时它从不更新。一种更常见的情况是三重关联中间的映射。

2. 三重关联

你现在可能有点不耐烦了，但我们保证这是最后一次介绍映射Category和Item之间关联的另一种方法了。下面来概括一下你已经了解的有关多对多关联的知识：

- 它可以利用两个集合在任意一端映射，也可以通过只有两个外键列的联结表映射。这是一个常规的多对多关联映射。
- 它可以通过表示联结表的中间实体类和那里任何额外的列映射。一对多的关联在任意一端（Category和Item）映射，并且双向的多对一相当于在中间的实体类中映射。
- 它可以被映射为单向，通过表示为值类型组件的联结表。Category实体有组件的一个集合。每个组件都有一个对它自己Category的引用以及对Item的多对一实体关联。（也可以在这段解释中切换单词Category和Item。）

你之前通过把另一个多对一实体关联添加到User，把上一个场景变成了一个三重关联。下面来用Map完成同样的事。

Category有Item实例的一个Map——每个映射项的键都是对Item的引用。每个映射项的值都是把Item添加到Category的User。如果联结表上没有额外的列，这个策略就很合适；请见图7-12中的Schema。

这一策略的好处在于不需要任何中间类，不需要实体或者值类型，就可以在Java应用程序中表示联结表的ADDED_BY_USER_ID列。

首先，这是Category中带有Hibernate扩展注解的Map属性。

```
@ManyToMany
@org.hibernate.annotations.MapKeyManyToMany(
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = @JoinColumn(name = "CATEGORY_ID"),
    inverseJoinColumns = @JoinColumn(name = "USER_ID")
)
private Map<Item,User> itemsAndUser = new HashMap<Item,User>();
```

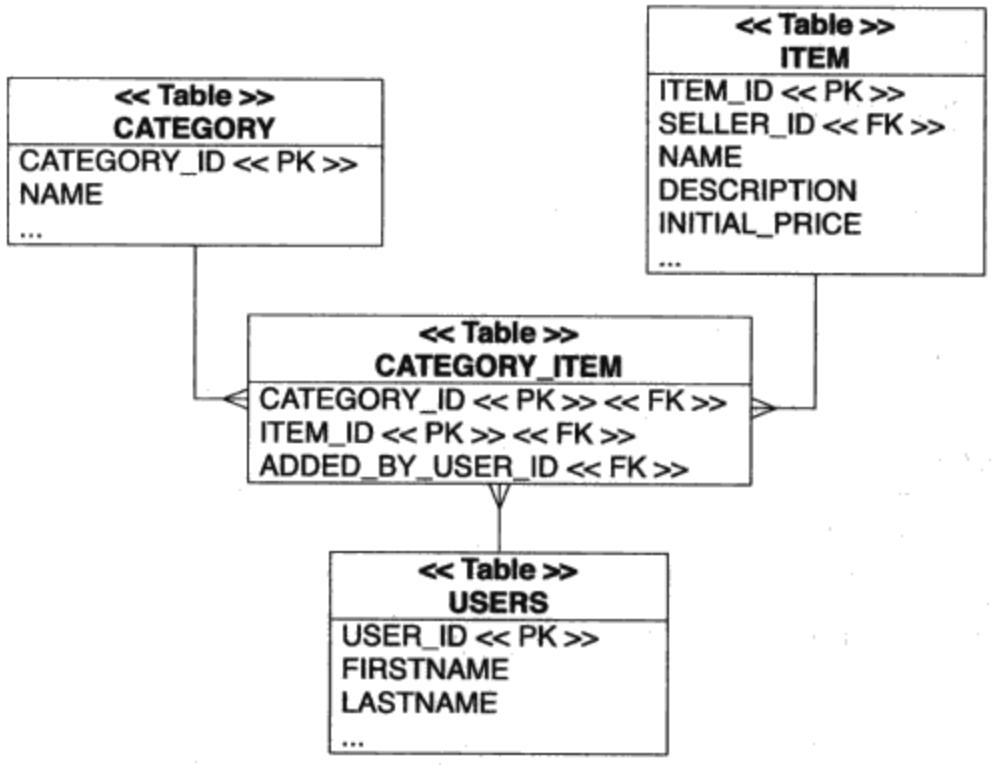

图7-12 三个实体之间使用联结表的三重关联

Hibernate XML映射包括一个新的元素<map-key-many-to-many>:

```
<map name="itemsAndUser" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <map-key-many-to-many column="ITEM_ID" class="Item"/>
    <many-to-many column="ADDED_BY_USER_ID" class="User"/>
</map>
```

为了在所有三个实体之间创建一个链接，如果你的所有实例都已经处于持久化状态，就添加一个新的项到该映射中：

```
aCategory.getItemsAndUser().add( anItem, aUser );
```

要移除链接，就从该映射中移除该项。当作练习，你可以试着通过Item中categories的一个集合，使这个映射变成双向。

记住，这必须是一个反向的集合映射，因此它不支持被索引的集合。

既然你知道了用于一般实体的所有关联映射方法，那么我们仍然必须考虑继承，以及对各种级别的继承层次结构的关联。我们真正要的是多态的（polymorphic）行为。我们来看看Hibernate是如何处理多态的实体关联的。

7.3 多态关联

多态是面向对象语言如Java的一项定义特性。对多态关联和多态查询的支持是ORM解决方案（如Hibernate）的一项绝对基础的特性。令人惊讶的是，我们已经谈了这么多，却不需要讨论多态。甚至更为令人惊讶的是，关于这一主题并没有太多的内容可说——多态在Hibernate中是如此容易使用，以致我们无须花费太多的精力对它进行解释。

为了有一个整体的认识，首先考虑对可能有子类的一个类的多对一关联。在这种情况下，

Hibernate保证你可以创建到任何子类实例的链接，就像创建到超类的实例的链接一样。

7.3.1　多态的多对一关联

多态关联（polymorphic association）可以引用在映射元数据中显式指定的类的子类实例。对于这个例子，考虑User的defaultBillingDetails属性。它引用一个特定的BillingDetails对象，该对象在运行时可以是该类的任何具体实例。这些类如图7-13所示。

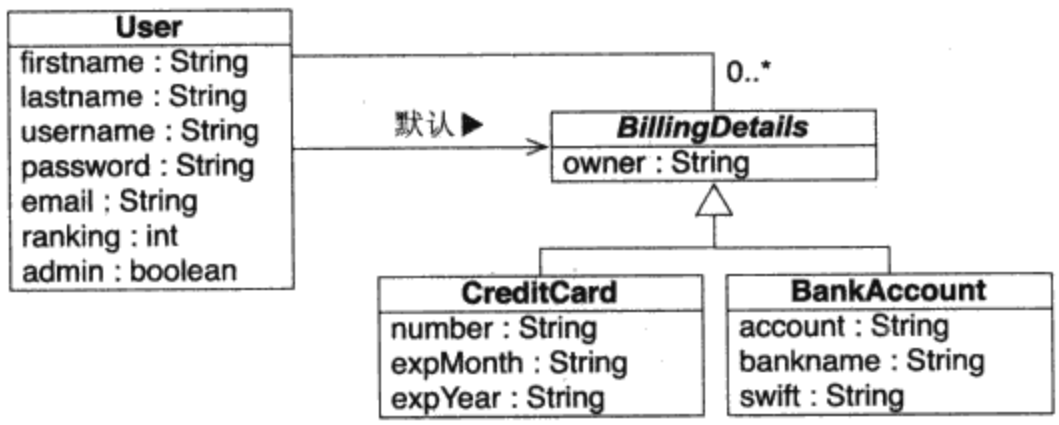

图7-13　user默认为拥有信用卡或者银行账号

在User.hbm.xml中把这个关联映射到抽象类BillingDetails如下：

```
<many-to-one name="defaultBillingDetails"
             class="BillingDetails"
             column="DEFAULT_BILLING_DETAILS_ID"/>
```

但是由于BillingDetails是抽象的，关联必须在运行时引用它其中一个子类的实例——CreditCard或者BankAccount。

不必做任何特别的事情来启用Hibernate中的多态关联；在关联映射中指定任何被映射的持久化类的名称（或者让Hibernate利用反射发现它），然后，如果该类声明了任何<union-subclass>、<subclass>或者<joined-subclass>元素，该关联就自然地成为多态。

下列代码示范了如何创建对CreditCard子类的实例的关联：

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

User user = (User) session.get(User.class, userId);
user.addBillingDetails(cc); // Add it to the one-to-many association

user.setDefaultBillingDetails(cc);

// Complete unit of work
```

现在，当你在第二个工作单元中导航关联时，Hibernate就自动获取CreditCard实例：

```
User user = (User) secondSession.get(User.class, userId);

// Invoke the pay() method on the actual subclass instance
user.getDefaultBillingDetails().pay(amount);
```

只有一件事情必须小心：如果BillingDetails通过lazy="true"映射（这是默认的），Hibernate就会代理defaultBillingDetails关联目标。在这种情况下，就不能在运行时执行类型转换到具体的类CreditCard，甚至instanceof操作符也会表现得很奇怪：

```
User user = (User) session.get(User.class, userid);
BillingDetails bd = user.getDefaultBillingDetails();
System.out.println( bd instanceof CreditCard ); // Prints "false"
CreditCard cc = (CreditCard) bd; // ClassCastException!
```

在这段代码中，类型转换失败了，因为bd是一个代理实例。当一个方法在代理中被调用时，调用被委托给了被延迟抓取的CreditCard的实例（它是运行时生成的子类的一个实例，因此instanceof也失败）。直到这个初始化发生，Hibernate才知道给定实例的子类型是什么——这将需要一个数据库命中，这是你试图首先通过延迟加载要努力避免的。为了执行代理安全的类型转换，要使用load()：

```
User user = (User) session.get(User.class, userId);
BillingDetails bd = user.getDefaultBillingDetails();

// Narrow the proxy to the subclass, doesn't hit the database
CreditCard cc =
    (CreditCard) session.load( CreditCard.class, bd.getId() );
expiryDate = cc.getExpiryDate();
```

调用load()之后，bd和cc指向两个不同的代理实例，这两者都委托给同一个底层的CreditCard实例。然而，第二个代理有不同的接口，可以调用只应用给这个接口的方法（如getExpiryDate()）。

注意，可以通过避免延迟抓取来避免这些问题，就像下列代码中的那样，用一个主动抓取查询：

```
User user = (User)session.createCriteria(User.class)
    .add(Restrictions.eq("id", uid) )
    .setFetchMode("defaultBillingDetails", FetchMode.JOIN)
    .uniqueResult();

// The users defaultBillingDetails have been fetched eagerly
CreditCard cc = (CreditCard) user.getDefaultBillingDetails();
expiryDate = cc.getExpiryDate();
```

真正面向对象的代码不应该使用instanceof或者众多的类型转换。如果你发现自己使用代理时遇到了问题，就应该置疑设计，问问是否有更加多态的方法。Hibernate也提供BCI（ByteCode Instrumentation），作为通过代理延迟加载的另一种选择；我们会在13.1节中回到抓取策略的主题。

一对一的关联以同样的方式进行处理。多值的关联要如何处理呢——例如，每个User的billingDetails集合？

7.3.2 多态集合

User可以有对多个BillingDetails的引用，而不只是单个默认（默认是这多个中的一个）。用一个双向的一对多关联映射它。

BillingDetails中有下列代码：

```
<many-to-one name="user"
             class="User"
             column="USER_ID"/>
```

在Users映射中有：

```
<set name="billingDetails"
     inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

添加CreditCard很容易：

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpMonth(...);
cc.setExpYear(...);

User user = (User) session.get(User.class, userId);

// Call convenience method that sets both sides of the association
user.addBillingDetails(cc);

// Complete unit of work
```

同往常一样，addBillingDetails()调用getBillingDetails().add(cc)和cc.setUser(this)，通过设置两个指针来保证关系的完整性。

可以迭代集合，并多态地处理CreditCard和BankAccount的实例（尽管你可能并不想在最终的系统中向用户开几次账单）：

```
User user = (User) session.get(User.class, userId);

for( BillingDetails bd : user.getBillingDetails() ) {
    // Invoke CreditCard.pay() or BankAccount.pay()
    bd.pay(paymentAmount);
}
```

在目前为止的这些例子中，假设BillingDetails是被显式映射的一个类，继承映射策略是每个类层次结构一张表，或者用每个子类一张表标准化。

然而，如果层次结构通过每个具体的类一张表映射（隐式多态）或者显式地通过每个带有联合的具体类一张表映射，这个场景就需要一个更为复杂的解决方案了。

7.3.3　对联合的多态关联

Hibernate支持多态的多对一和一对多关联，如前几节所示，即使类层次结构是通过每个具体类一张表的策略映射。你可能想知道这是如何进行的，因为你可能没有使用这一策略以用于超类的表；如果真是如此，就无法引用或者添加外键列到BILLING_DETAILS了。

回顾5.1.2节中对每个带有联合的具体类一张表的讨论。特别注意获取BillingDetails的实例时，Hibernate执行的多态查询。现在，考虑给User映射的BillingDetails的集合：

```
<set name="billingDetails"
     inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

如果想要启用多态的联合特性，这个多态关联的必要条件是它为反向：在对面端必须有一个映射。在BillingDetails的映射中，通过<union-subclass>，必须包括<many-to-one>关联：

```
<class name="BillingDetails" abstract="true">

    <id name="id" column="BILLING_DETAILS_ID" .../>

    <property .../>

    <many-to-one name="user"
                 column="USER_ID"
                 class="User"/>

    <union-subclass name="CreditCard" table="CREDIT_CARD">
        <property .../>
    </union-subclass>

    <union-subclass name="BankAccount" table="BANK_ACCOUNT">
        <property .../>
    </union-subclass>

</class>
```

有两张表用于层次结构的两个具体类。每张表都有外键列USER_ID，引用USERS表。该Schema如图7-14所示。

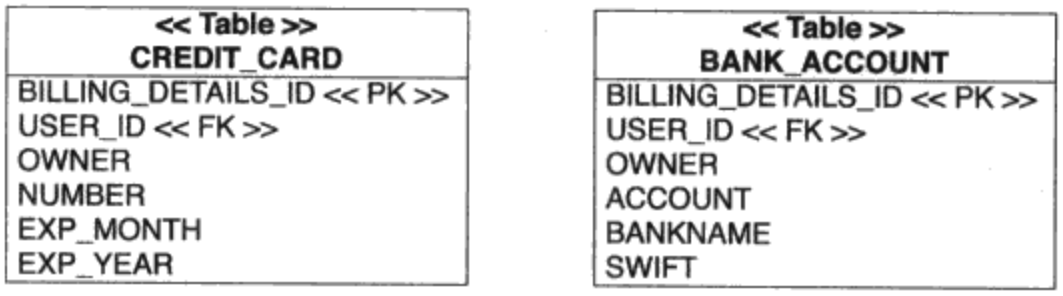

图7-14 被映射到两张单独表的两个具体类

现在，考虑下列数据访问代码：

```
aUser.getBillingDetails().iterator().next();
```

Hibernate执行UNION查询，获取在这个集合中引用的所有实例：

```
select
    BD.*
from
  ( select
      BILLING_DETAILS_ID, USER_ID, OWNER,
      NUMBER, EXP_MONTH, EXP_YEAR,
      null as ACCOUNT, null as BANKNAME, null as SWIFT,
      1 as CLAZZ
    from
      CREDIT_CARD
```

```
    union

    select
      BILLING_DETAILS_ID, USER_ID, OWNER,
      null as NUMBER, null as EXP_MONTH, null as EXP_YEAR
      ACCOUNT, BANKNAME, SWIFT,
      2 as CLAZZ
    from
      BANK_ACCOUNT
  ) BD
where
    BD.USER_ID = ?
```

FROM子句子查询是所有具体类表的一个联合，它给所有实例包括了USER_ID外键值。现在外部的select对引用特定用户的所有行在WHERE子句中包括了一项限制。

这个魔法对于数据的获取非常棒。如果操作集合和关联，非反向端被用来在具体的表中更新USER_ID列。换句话说，反向集合的修改不起作用：采用CreditCard或者BankAccount实例的user属性值。

现在再次考虑多对一的关联defaultBillingDetails，在USERS表中通过DEFAULT_BILLING_DETAILS_ID列映射。如果访问属性，Hibernate就执行一个看起来类似于前一个查询的UNION查询来获取这个实例。然而，不是在WHERE子句中限制特定的用户，而是在特定的BILLING_DETAILS_ID中进行限制。

重要提示：Hibernate不能也不会用这个策略为DEFAULT_BILLING_DETAILS_ID创建外键约束。这个引用的目标表可以是任何具体的表，它们不能被轻易地约束。你应该考虑通过一个数据库触发器为这个列编写一个定制的完整性规则。

还有一种有问题的继承策略：带有隐式多态的每个具体类一张表。

7.3.4　每个具体类一张多态表

在5.1.1节中，我们定义过每个具体类一张表的策略，并且发现这个映射策略使得表示多态关联变得很困难，因为无法把外键关系映射到抽象超类的表。没有使用这一策略时用于超类的表；只有用于具体类的表。也无法创建UNION，因为Hibernate不知道什么东西在统一这些具体的类；该超类（或者接口）不会被随处映射。

如果这个继承映射策略应用在BillingDetails层次结构中，Hibernate就不支持User中多态的billingDetails一对多集合。如果你需要多态的多对一关联来使用这一策略，则必须求助于hack。我们在本节中介绍的这种方法应该成为你最迫不得已时的选择。先试着转换到<union-subclass>映射。

假设你想要表示从User到BillingDetails的多态的多对一关联，其中BillingDetails类层次结构通过每个具体类一张表的策略和Hibernate中的隐式多态行为进行映射。你有CREDIT_CARD表和BANK_ACCOUNT表，但是没有BILLING_DETAILS表。Hibernate在USERS中需要两条信息，以便唯一地辨别被关联的默认CreditCard或者BankAccount：

- 被关联实例所在的表的名称；

- 被关联实例的标识符。

USERS表除了DEFAULT_BILLING_DETAILS_ID之外，还需要DEFAULT_BILLING_DEFAULTS_TYPE列。这个额外的列作用就像额外的辨别标志一样，并且在User.hbm.xml中还需要Hibernate <any>映射：

```
<any name="defaultBillingDetails"
     id-type="long"
     meta-type="string">
    <meta-value value="CREDIT_CARD" class="CreditCard"/>
    <meta-value value="BANK_ACCOUNT" class="BankAccount"/>
    <column name="DEFAULT_BILLING_DETAILS_TYPE"/>
    <column name="DEFAULT_BILLING_DETAILS_ID"/>
</any>
```

meta-type属性指定DEFAULT_BILLING_DEFAULTS_TYPE列的Hibernate类型；id-type属性指定DEFAULT_BILLING_DETAILS_ID列的类型（CreditCard和BankAccount必须有相同的标识符类型）。

<meta-value>元素告诉Hibernate如何解释DEFAULT_BILLING_DETAILS_TYPE列的值。在这里不需要使用完整的表名——可以使用你喜欢的任何值作为类型辨别标志。例如，可以把信息编码成两个字符：

```
<any name="defaultBillingDetails"
     id-type="long"
     meta-type="string">
    <meta-value value="CC" class="CreditCard"/>
    <meta-value value="CA" class="BankAccount"/>
    <column name="DEFAULT_BILLING_DETAILS_TYPE"/>
    <column name="DEFAULT_BILLING_DETAILS_ID"/>
</any>
```

这个表结构的例子如图7-15所示。

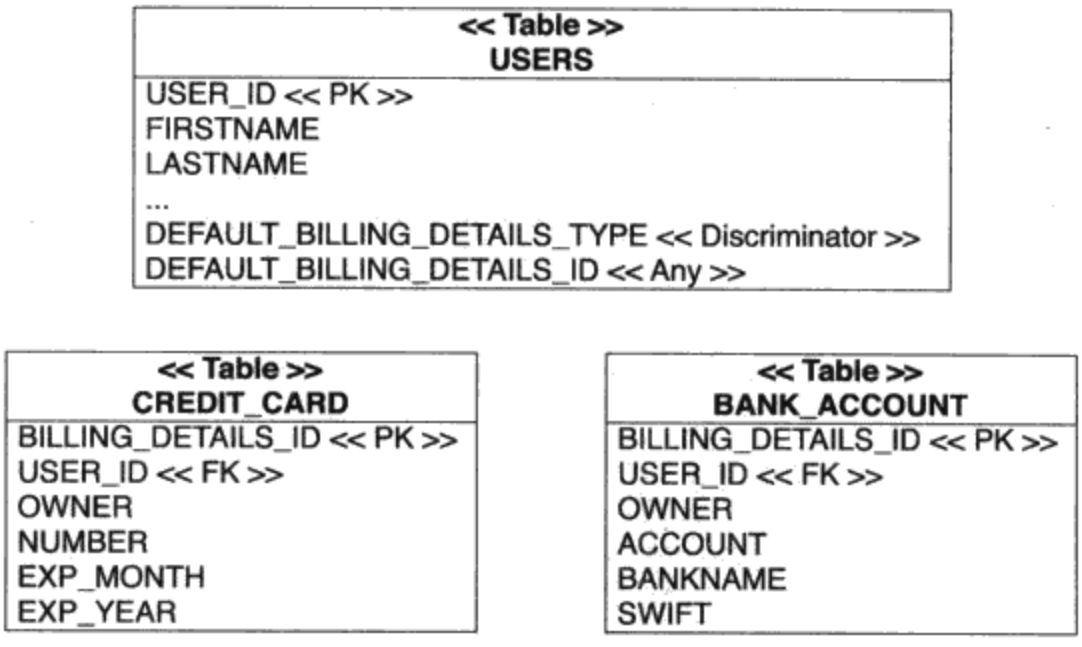

图7-15 通过任一关联使用辨别标志列

下面是使用这种关联时的第一个主要问题：你无法把外键约束添加到DEFAULT_BILLING_DETAILS_ID列，因为有些值引用BANK_ACCOUNT表，其他的则引用CREDIT_CARD表。这样，你就必须找出一些其他的方法来确保完整性（如触发器）。这与你使用<union-subclass>策略时要面对的问题相同。

此外，很难给这个关联编写SQL表联结。尤其是，Hibernate查询工具不支持这种关联映射，这个关联也不能通过外部联结而被抓取。除非最特殊的情况，否则我们不鼓励在任何情况下使用<any>关联。还要注意，这种映射技术不能通过注解或者在Java Persistence中使用（这个映射是如此罕见，因此目前为止还没有人要求注解支持它）。

如你所见，只要你不准备把关联创建到通过隐式多态映射的类层次结构，关联就很简单；通常不需要考虑它。你可能感到奇怪，前几节中竟然没有介绍任何JPA或者注解实例——运行时的行为是一样的，获得它不需要任何额外的映射。

7.4　小结

本章介绍了如何映射更复杂的实体关联。我们介绍的许多方法都是很少需要用到的，如果你可以简化类之间的关系，可能根本不需要用到它们。尤其是，多对多的实体关联通常最好表示为两个到中间实体类的一对多关联，或者利用组件的一个集合。

表7-1概括了原生的Hibernate特性和Java Persistence的比照。

表7-1　第7章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate对共享主键一对一关联映射支持键生成	支持标准的一对一映射。自动的共享主键生成需要通过Hibernate扩展才有可能
Hibernate支持跨联结表的所有实体关联映射	标准的关联映射可以跨二级表使用
Hibernate通过持久化索引支持列表的映射	持久化索引需要一个Hibernate扩展注解
Hibernate支持完全多态的行为。它对任一映射到一个通过隐式多态映射的继承层次结构的关联提供额外的支持	可用完全多态的行为，但是没有注解支持任一映射

下一章将关注遗留数据库的整合，以及如何定制让Hibernate自动为你生成的SQL。无论你是必须使用遗留的Schema，还是比如想要通过定制DDL改善新的Schema，这一章都值得关注。