第4章 映射持久化类

本章内容

- 理解实体和值类型概念
- 用XML和注解映射类
- 细粒度的属性和组件映射

本章介绍基础的映射选项，阐述类和属性如何被映射到表和列。我们介绍和讨论如何处理数据库同一性和主键，以及各种其他元数据设置如何被用来定制Hibernate加载和存储对象的方式。所有映射示例都以Hibernate的原生XML格式完成，且并列使用JPA注解和XML描述符。我们还深入探讨细粒度领域模型的映射，以及如何映射属性和嵌入的组件。

首先要定义实体和值类型之间的本质区别，并阐述应该如何处理领域模型的ORM。

4.1 理解实体和值类型

实体是表述一级业务对象的持久化类型（术语“对象”在这里指它本来的含义）。换句话说，你在一个应用程序中必须处理的有些类和类型更为重要，它一般使得其他的类和类型变得比较不重要。你可能会认同这种说法：在CaveatEmptor中，`Item`是比`String`更重要的类。`User`可能比`Address`更重要。是什么使得有些东西变得重要呢？让我们换个角度来看这个问题。

4.1.1 细粒度的领域模型

Hibernate的一个主要目标是支持细粒度的领域模型，过去我们把它分离出来作为富领域模型最重要的必要条件。这是我们使用POJO的一个原因。粗略地说，细粒度意味着类比表更多。

例如，用户可能既有付款地址又有家庭住址。在数据库中，你可能有单个`USERS`表，包含列`BILLING_STREET`、`BILLING_CITY`和`BILLING_ZIPCODE`，以及`HOME_STREET`、`HOME_CITY`和`HOME_ZIPCODE`。（还记得我们在第1章讨论过的SQL类型的问题吗？）

在领域模型中，可以使用相同的方法，把两个地址表示为`User`类的6个字符串值属性。但是用`Address`类给它建立模型会更好，在这个模型中`User`有`billingAddress`和`homeAddress`属性，从而给一个表使用三个类。

这个领域模型实现了更高的凝聚力和更好的代码重用，并且它比使用固定类型系统的SQL系统更容易理解。过去，许多ORM解决方案都没有对这种映射提供特别好的支持。

Hibernate强调实现类型安全和行为的细粒度类的有效性。例如，许多人把电子邮件地址作为User的一个字符串值的属性建立模型。一种更成熟的方法是定义EmailAddress类，它添加更高级的语义和行为——它可以提供sendEmail()方法。

这个粒度问题把我们引向了ORM中极其重要的一个特征。在Java中，所有的类都是平等的——所有的对象都有它们自己的同一性和生命周期。

我们来看一个例子。

4.1.2 定义概念

两个人住在同一幢公寓里，他们都在CaveatEmptor注册了用户账号。每个账号一般都由User的一个实例表示，因此你有两个实体实例。在CaveatEmptor模型中，User类与Address类有一个homeAddress关联。两个User实例都在运行时引用同一个Address实例，还是每个User实例都引用它自己的Address？如果Address假定要支持共享的运行时引用，它就是个实体类型。如果不支持，它就可能是个值类型，因此通过一个自己的实体实例（它也提供同一性）依赖于单个引用。

我们提倡类比表多的设计：一行表示多个实例。由于数据库同一性由主键值实现，有些持久化对象将没有自己的同一性。实际上，持久化机制对有些类实现按值传递（pass-by-value）的语义！行中表示的其中一个对象有它自己的同一性，其他对象则依赖于它。在前一个例子中，包含地址信息的USERS表中的列依赖于用户的标识符，表的主键。Address的实例依赖于User的实例。

Hibernate产生了下列本质的特征：

- 实体类型的对象有它自己的数据库同一性（主键值）。对实体实例的对象引用被持久化为数据库中的引用（一个外键值）。实体有它自己的生命周期；它可以独立于任何其他的实体而存在。CaveatEmptor中的例子是User、Item和Category。
- 值类型的对象没有数据库同一性；它属于一个实体实例，并且它的持久化状态被嵌入到自身实体的表行中。值类型没有标识符或者标识符属性。值类型实例的寿命受它自己的实体实例的寿命限制。值类型不支持共享引用：如果两个用户住在同一幢公寓里，每个用户都有一个对他们自身homeAddress实例的引用。最明显的值类型是类，比如String和Integer，但是所有的JDK类都被当作值类型。用户自定义的类也可以被映射为值类型；例如，CaveatEmptor有Address和MonetaryAmount。

领域模型中实体和值类型的识别不是一项特别的任务，但是遵循一个确定的过程。

4.1.3 识别实体和值类型

你可能发现给UML类图添加固定的信息很有帮助，以便可以立即看到和区分实体和值类型。这个实践也迫使你对所有的类考虑这种区别，这是可选映射和执行良好的持久层的第一步。请见图4-1中的例子。

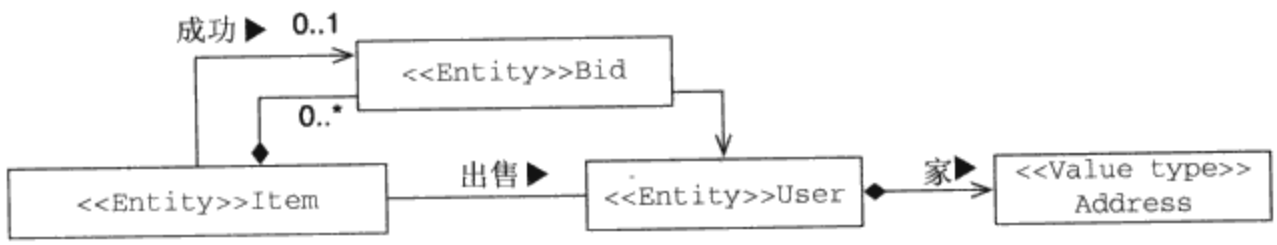

图4-1　已经把实体和值类型的固定信息添加到图中

Item和User类是明显的实体。它们各自都有自己的同一性，它们的实例有来自许多其他实例的引用（共享引用），并有独立的生命周期。

把Address视同一个值类型也很容易：一个特定的Address实例只被单个User实例引用。你知道，因为关联已经作为一个复合来创建，其中的User实例已经被设计成完全负责被引用Address实例的生命周期。因此，Address对象不能被任何其他的实例引用，也不需要它们自己的同一性。

Bid类是个问题。在面向对象的模型中，你表达一个组合（带有钻石符号的Item和Bid之间的关联），且Item管理所有Bid对象生命周期，并且Item有着对所有Bid对象的一个引用（一个引用集合）。这看起来似乎有道理，因为如果Item不再存在了，出价也就没用了。但是同时，还有另一个对Bid的关联：Item可能持有对它sucessfulBid的一个引用。这个成功的出价也必须是集合引用的其中一个出价，但是这在UML中没有表达出来。不管怎样，你必须处理对Bid实例可能的共享引用，因此Bid类必须是一个实体。它有一个依赖的生命周期，但必须有自己的同一性来支持共享引用。

你会经常发现这种混合行为；然而，你的第一反应应该是使所有的东西都成为值类型的类，并且只有当绝对必要时才把它提升为实体。试着简化关联：例如，集合有时候增加了无益的复杂性。不要映射Bid引用的一个持久化集合，而是可以编写一个查询，来获得Item的所有出价（第7章将再次回到这个话题）。

随着下一个步骤，把领域模型图拿走，并对所有实体和值类型实现POJO。你必须小心3件东西：

- *共享引用*——编写POJO类，以一种避免对值类型实例共享引用的方式。例如，确保Address对象只可以被一个User引用。例如，使它不可变，并用Address构造函数强制这种关系。
- *生命周期依赖*——如前所述，一个值类型实例的生命周期受它自己的实体实例限制。如果User对象被删除了，它的Address依赖对象也必须被删除。在Java中没有这种概念或者关键字，但是你的应用程序工作流和用户接口必须被设计成尊重和期待生命周期依赖。持久化元数据包括了所有依赖的级联规则。
- *同一性*——实体类几乎在任何情况下都需要标识符属性。用户自定义的值类型类（和JDK类）没有标识符属性，因为实例通过自己的实体被标识。

当本书稍后讨论到更高级的映射时，会回过头来讲类关联和生命周期规则。然而对象同一性是你目前必须理解的一个主题。

4.2 映射带有同一性的实体

在讨论术语（比如数据库同一性和Hibernate管理同一性的方式）之前，理解对象同一性和对象等同性之间的区别很重要。接下来探讨对象同一性和等同性如何与数据库（主键）的同一性相关联。

4.2.1 理解 Java 同一性和等同性

Java开发人员理解Java对象同一性和等同性之间的区别。对象同一性（==）是通过Java虚拟机定义的一个概念。如果两个对象引用指向相同的内存位置，它们就是同一的。

另一方面，对象等同性则是通过实现equals()方法的类定义的一个概念，有时候也被认为等值。等值意味着两个不同（非同一）的对象有着相同的值。如果String的两个不同实例表示顺序相同的字符，它们就是等值的，即使它们各自在虚拟机的内存中都有自己的位置。（如果你是Java专家，我们承认String是个特例。假设我们用了不同的类来说明同一个问题。）

持久化使事情变得复杂起来。利用对象/关系持久化，持久化对象是数据库表的一个特定行的内存表示。连同Java同一性（内存位置）和对象等同性一起，你还要选择数据库同一性（在持久化数据仓库中的位置）。现在你有3种识别对象的方法：

- 如果对象在JVM中占据着相同的内存位置，它们就是同一的。这可以通过使用 == 操作符进行检查。这个概念称作对象同一性。
- 如果对象有着相同的值，它们就是相等的，如equals(object o)方法定义的一样。不显式覆盖这个方法的类，继承了由java.lang.Object定义的实现，它比较对象同一性。这个概念称作等同性。
- 如果存储在一个关系数据库中的对象表示相同的行，或者它们共享相同的表和主键值，它们就是同一的。这个概念称作数据库同一性。

现在要看看数据库同一性如何与Hibernate中的对象同一性相关联，以及数据库同一性在映射元数据中如何表达。

4.2.2 处理数据库同一性

Hibernate以两种方式把数据库同一性公开给应用程序：

- 持久化实例的标识符属性值。
- Session.getIdentifier(Object entity)返回的值。

1. 给实体添加标识符属性

标识符属性很特殊——它的值是由持久化实例表示的数据库行的主键值。通常在领域模型图中不显示标识符属性。在示例中，标识符属性总是被命名为id。如果myCategory是Category的一个实例，调用myCategory.getId()就返回数据库中由myCategory表示的行的主键值。

来给Category类实现一个标识符属性：

```
public class Category {
    private Long id;
```

```
    ...
    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

你应该使标识符属性的访问方法为私有（private）还是公有（public）范围呢？数据库标识符经常被应用程序当作对特定实例的一个方便的句柄使用，甚至在持久层之外。例如，把一个搜索屏幕的结果给用户显示为一个摘要信息的清单，这对于Web应用程序来说很常见。当用户选择一个特定的元素时，应用程序可能需要获取选中的对象，并且通常为此通过标识符来使用一个查找——你可能已经以这种方式使用过标识符，甚至在依赖JDBC的应用程序中。用一个公有标识符属性访问器来完全公开数据库同一性通常是适当的。

另一方面，通常声明`setId()`方法为私有，并让Hibernate生成和设置标识符值。或者，用直接的字段访问映射，并仅实现一个获取方法。（这个规则的例外是带有自然键的类，在这些类中，标识符的值在对象变成持久化之前由应用分配，而不是由Hibernate生成。我们将在第8章讨论自然键。）Hibernate不允许持久化实例的标识符值被第一次分配之后对它进行改变。主键值永远不变——否则该属性将不是个适当的主键备选对象！

标识符属性的Java类型，如前一个例子中的`java.lang.Long`，取决于`CATEGORY`表的主键类型，以及它在Hibernate元数据中如何被映射。

2. 映射标识符属性

用`<id>`元素在Hibernate XML文件中映射一个普通的（非复合的）标识符属性：

```
<class name="Category" table="CATEGORY">
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="native"/>
</id>

...
</class>
```

标识符属性被映射到表`CATEGORY`的主键列`CATEGORY_ID`。这个属性的Hibernate类型是`long`，它在大部分数据库中映射到`BIGINT`列类型，并且已经被选择用来匹配由`native`标识符生成器生成的同一性值的类型。（下一节讨论标识符生成策略。）

对于JPA实体类，你在Java源代码中用注解映射标识符属性：

```
@Entity
@Table(name="CATEGORY")
public class Category {
    private Long id;
    ...

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "CATEGORY_ID")
    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

获取方法上的@Id注解，标志着它是标识符属性，而包含GenerationType.AUTO选项的@GenerateValue变成了一个原生的标识符生成策略，就像Hibernate XML映射中的native选项一样。注意，即使你没有定义strategy，默认也是GenerationType.AUTO，因此完全可以省略这个属性。你还指定了一个数据库列——否则Hibernate会使用属性名称。Java属性类型暗指的映射类型是java.lang.Long。

当然，也可以对所有属性使用直接的字段访问，包括数据库标识符：

```
@Entity
@Table(name="CATEGORY")
public class Category {

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

    public Long getId() {
        return this.id;
    }
    ...

}
```

就像标准所定义的，当直接的字段访问被启用时，映射注解就被放在字段声明中。

给实体启用字段还是属性访问，这取决于强制的@Id注解的位置。在前面的例子中，它出现在字段中，因此这个类的所有属性都由Hibernate通过字段访问。在那之前的例子，在getId()方法上被注解的，使得能通过获取方法和设置方法访问所有的属性。

另一种方法，可以使用JPA XML描述符创建标识符映射：

```
<entity class="auction.model.Category" access="FIELD">
    <table name="CATEGORY"/>
    <attributes>
        <id name="id">
            <generated-value strategy="AUTO"/>
        </id>
        ...
    </attributes>
</entity>
```

除了用来测试Java对象同一性（a==b）和对象等同性（a.equals(b)）的操作之外，现在可以用a.getId().equals(b.getId())测试数据库同一性。这些概念有什么相似之处？在哪些情况下它们全部返回true？当所有都为真的时候，称作受保证的对象同一性的范围；9.2节将回到这个主题。

在Hibernate中使用数据库同一性很简单且容易理解。选择一个好的主键（和键生成策略）可能就比较困难了。接下来讨论这个问题。

4.2.3　数据库主键

Hibernate需要知道你生成主键的首选策略。不过，先要定义主键。

1．选择主键

备选的键是能够用来识别表中一个特定行的一列或者一组列。要变成主键，备选键必须满足下列属性：

- 它的值（对于备选键的任意列而言）永远不为空。
- 每一行都有唯一的值。
- 一个特定行的值永远不变。

如果一张表只有一个识别属性，根据定义，它就是主键。然而，几个列或者列的组合可能满足一张特定表的这些属性；你在备选键之间选择，给表决定最佳主键。没有被选作主键的备选键应该在数据库中声明为唯一键。

许多遗留的SQL数据模型使用自然主键。自然键是带有业务意义的键：属性或者属性的组合因其业务语义而唯一。自然键的例子有美国社会保障号码（U.S. Social Security Number）和澳大利亚税务档案号码（Australian Tax File Number）。区分自然键很简单：如果一个备选键属性在数据库上下文之外有意义，它就是自然键，无论它是否自动生成。考虑应用程序的用户：当他们谈论和使用应用程序时提到的键属性，它就是自然键。

经验表明，长远来看自然键通常会产生问题。好的主键必须是唯一、不变和必要的（从不为空或者未知）。很少有实体属性满足这些条件，并且有些确实无法被SQL数据库有效地索引（虽然这是一个实现细节，且不应该成为支持或者反对一个特定键的主要动机）。此外，你应该确定备选键定义可以在它成为主键之前的整个数据库生命周期中始终不变。改变主键的值（或者甚至定义）和所有引用它的外键，是一项令人沮丧的工作。此外，自然的备选键经常只有在复合的自然键中合并几个列才看得到。这些复合键，虽然必定适合于某些关系（比如多对多的关系中的链接表），但通常使得维护、特别查询和Schema演变变得更加困难。

基于这些原因，强烈建议你考虑合成标识符（synthetic identifier），也称代理键（surrogate key）。代理键没有业务意义——它们是由数据库或者应用程序生成的唯一值。应用程序的用户最好不要看到或者引用这些键值；它们是系统内部的一部分。一般情况下引入代理键列也合适：如果没有备选键，按定义，表不是一种由关系模型定义的关系——它允许完全相同的行——因此必须添加一个代理键列。生成代理键值有许多众所周知的方法。

2. 选择键生成器

Hibernate有几个内建的标识符生成策略。表4-1中列出了最有用的一些选项。

表4-1 Hibernate内建的标识符生成器模块

生成器名称	JPA GenerationType	选 项	描 述
native	AUTO	—	native同一性生成器挑选其他同一性生成器如identity、sequence或者hilo，取决于底层数据库的能力。使用这个生成器保证映射元数据可以移植到不同的数据库管理系统
identity	IDENTITY	—	这个生成器支持DB2、MySQL、MS SQL Server、Sybase和HypersonicSQL中的同一性列。返回的标识符类型为long、short或者int
sequence	SEQUENCE	sequence，parameters	这个生成器在DB2、PostgreSQL、Oracle、SAP DB或者Mckoi中创建一个序列；或者使用InterBase中的一个生成器。返回的标识符类型为long、short或者int。如果创建序列的其他设置要被添加到DDL，就使用sequence选项给序列定义一个目录名称（默认是hibernate_sequence）和parameters
increment	不可用	—	Hibernate启动时，这个生成器读取表的最大（数字）主键列值，并且每次插入一个新行时值就增加1。生成的标识符类型为long、short或者int。如果单服务器的Hibernate应用程序对数据库具有排它性访问时，这个生成器特别有效，但不应该被用于任何其他场合
hilo	不可用	table，column，max_lo	高/低位算法是生成long类型标识符的一种高效的方法，给定表和列（默认时，分别为hibernate_unique_key和next）作为高值的源。高/低位算法生成仅对特定数据库唯一的标识符。高值从全局的源中获取，并通过添加本地的低值被变成唯一。这种算法在标识符值的单个源必须被多个插入访问时避免拥堵。更多有关高/低位方法生成唯一标识符的信息请见“数据建模101”（Ambler，2002）。这个生成器每次获取高值时总要使用一个单独的数据库连接，因此它不受用户提供的数据库连接支持。换句话说，别把它与sessionFactory.openSession(myConnection)共用。max_lo选项定义了直到获取一个新的高值时添加了多少个低值。只有大于1的设置才有意义；默认为32767（Short.MAX_VALUE）
seqhilo	不可用	Sequence，parameters，max_lo	这个生成器的作用就像一般的hilo生成器，除了它使用一个具名的数据库序列来生成高值之外
（只适用于JPA）	TABLE	Table，catalog，Schema，PkColumnName，ValueColumnName，PkColumnValue，allocationSize	很像Hibernate的hilo策略，TABLE依赖持有最新生成的整数主键值的数据库表，每个生成器都被映射到这张表中的一行。每一行有两个列：pkColumnName和valueColumnName。pkColumnValue把每一行分配到特定的生成器，并且值列持有最新获取的主键。持久化提供程序每次都分配到allocationSize为整数为止

（续）

生成器名称	JPA GenerationType	选　项	描　述
uuid.hex	不可用	separator	这个生成器是一个128位的UUID（生成String类型的标识符的一种算法，在网络内部唯一）。IP地址与一个唯一的时间戳结合使用。UUID被编码成为长度32的十六进制数字的一个字符串，UUID表示法的每个组件之间带有一个可选的separator字符串。仅当你全局地需要唯一标识符时才使用这个生成器策略，例如当你必须规则地合并两个数据库的时候
guid	不可用	—	这个生成器在MySQL和SQL服务器中提供一个数据库生成的全局唯一的标识符字符串
select	不可用	key	这个生成器获取由数据库触发器分配的一个主键，它按某些唯一键选择行并获取主键值。这个策略需要另一个唯一的备选键列，且key选项必须被设置成唯一键列的名称

有些内建的标识符生成器可以通过选项得到配置。在原生的Hibernate XML映射中，把选项定义为键/值对：

```
<id column="MY_ID">
    <generator class="sequence">
        <para name="sequence">MY_SEQUENCE</parameter>
        <para name="parameters">
            INCREMENT BY 1 START WITH 1
        </para>
    </generator>
</id>
```

可以通过注解使用Hibernate标识符生成器，即使没有直接的注解可用：

```
@Entity
@org.hibernate.annotations.GenericGenerator(
    name = "hibernate-uuid",
    strategy = "uuid"
)
class name MyEntity {

    @Id
    @GeneratedValue(generator = "hibernate-uuid")
    @Column(name = "MY_ID")
    String id;
}
```

@GenericGenerator Hibernate扩展可以用来给Hibernate标识符生成器提供一个名称，在这个例子中为hibernate-uuid。然后这个名称被标准的generator属性引用。

生成器的这个声明和它按名称的分配，也必须通过注解应用到基于序列或基于表的标识符生成中。想象你要在所有的实体类中使用一个定制的序列生成器。因为这个标识符生成器必须是全局的，所以在orm.xml中声明：

```
<sequence-generator name="mySequenceGenerator"
    sequence-name="MY_SEQUENCE"
    initial-value="123"
    allocation-size="20"/>
```

这声明了具名MY_SEQUENCE的数据库序列，带有初始值123，可以用作数据库标识符生成的来源，并且每当持久化引擎需要标识符时，它就应该获得20个值。（但是注意，在编写本书之时，Hibernate Annotations忽略initial-Value设置。）

为了把这个标识符生成器应用到特定的实体，就利用它的名称：

```
@Entity
class name MyEntity {

    @Id @GeneratedValue(generator = "mySequenceGenerator")
    String id;
}
```

如果在实体级声明了另一个同名的生成器，且在class关键字之前，它就会覆盖全局的标识符生成器。可以用相同的方法声明和应用@TableGenerator。

你不受限于内建的策略，还可以通过实现Hibernate的IdentifierGenerator接口创建自己的标识符生成器。跟往常一样，查看现有标识符生成器的Hibernate源代码来寻求灵感是一种好策略。

甚至可能在单个领域模型中给持久化类混合标识符生成器，但是对于非遗留的数据，建议给所有实体使用相同的标识符生成策略。

对于遗留的数据和应用程序分配的标识符，情况就更复杂些。此时，应该经常坚持使用自然键，特别是复合键。复合键是由多个表列组成的自然键。因为复合标识符可能更难以使用一些，并且经常只出现在遗留的Schema中，因此我们仅在8.1节的上下文中讨论它。

从现在起，假设你已经把标识符属性添加到领域模型的实体类，并且在完成了每个实体和它的标识符属性的基础映射之后，继续映射实体的值类型属性。然而，有些特殊的选项可以简化或者增强你的类映射。

4.3 类映射选项

如果查看DTD（或者参考文档）中的<hibernate-mapping>和<class>元素，将会发现一些目前还没有讨论到的选项：

- CRUD SQL语句的动态生成；
- 实体易变性控制；
- 给查询命名实体；
- 映射包名称；
- 引用关键字和被保存的数据库标识符；
- 实现数据库命名约定。

4.3.1 动态的 SQL 生成

默认情况下，Hibernate在启动时给每个持久化类创建SQL语句。这些语句是用来读取单个行、

删除一行等的简单创建、读取、更新和删除。

Hibernate在启动时如何创建UPDATE语句？毕竟，这时候还不知道要更新的列。答案是生成的SQL语句更新所有的列，并且如果特定列的值没有被修改，这个语句就会把它设置为它的旧值。

在有些情况下，例如包含几百列的一个遗留表，在该表中，即使最简单的操作（假设只有一个列需要更新）的SQL语句也将很大，必须关闭这个启动时的SQL生成，并切换到运行时生成的动态语句。极为大量的实体也会影响启动时间，因为Hibernate必须为CRUD提前生成所有SQL语句。如果必须为数千个实体高速缓存许多语句的话，这个查询语句高速缓存的内存消耗也将很高（这通常不成问题）。

<class>映射元素中有两个属性可以禁用启动时CRUD SQL的生成：

```
<class name="Item"
   dynamic-insert="true"
   dynamic-update="true">
...
</class>
```

dynamic-insert属性告诉Hibernate是否在SQL INSERT中包括空的属性值，dynamic-update属性告诉Hibernate是否在SQL UPDATE中包括未被修改的属性。

如果使用JDK 5.0注解映射，就需要一个原生的Hibernate注解来启用动态的SQL生成：

```
@Entity
@org.hibernate.annotations.Entity(
    dynamicInsert = true, dynamicUpdate = true
)
public class Item { ...
```

来自Hibernate包的第二个@Entity注解使用额外的选项扩展了JPA注解，包括dynamicInsert和dynamicUpdate。

有时候可以避免生成任何UPDATE语句，如果持久化类被映射为不可变的话。

4.3.2　使实体不可变

一个特定类的实例可以是不可变的。例如，在CaveatEmptor中，对货品所做的Bid是不可变的。因此，在BID表中永远不需要执行UPDATE语句。Hibernate也可以进行一些其他优化，例如，如果通过把mutable属性设置为false来映射一个不可变的类，就可以避免脏检查：

```
<hibernate-mapping default-access="field">
    <class name="Bid" mutable="false">
    ...
    </class>
</hibernate-mapping>
```

如果没有公开类的任何属性的公有设置方法，POJO就是不可变的——所有值都在构造函数中设置。不用私有设置方法，而是经常更喜欢通过Hibernate对不可变的持久化进行直接的字段访问，因此不必编写没用的访问方法。可以用注解映射一个不可变的实体：

```
@Entity
@org.hibernate.annotations.Entity(mutable = false)
```

```
@org.hibernate.annotations.AccessType("field")
public class Bid { ...
```

原生的Hibernate @Entity注解再次用额外的选项扩展了JPA注解。此处我们还展现了Hibernate扩展注解@AccessType——这是个很少使用的注解。如前所述，一个特定实体类的默认访问策略，对于强制的@Id属性的位置来说是隐式的。然而，可以用@AccessType强制一个更细粒度的策略；它可以被放在类声明中（如前面的例子），或者甚至放在特定字段或者访问方法中。

我们来快速看看另一个问题：给查询命名实体。

4.3.3 给查询命名实体

默认情况下，所有类名都自动地“导入”到Hibernate查询语言（HQL）的命名空间。换句话说，可以在HQL中使用没有包前缀的短类名，这很方便。然而，如果给定的SessionFactory存在两个同名的类，这两个类可能在领域模型的不同包中，这个自动导入就可以关闭。

如果存在这种冲突，而又不改变默认的设置，Hibernate将不知道你正在HQL中引用哪个类。可以在<hibernate-mapping>根元素中设置autoimport="false"，对特定的映射文件把名称的自动导入关闭到HQL命名空间。

实体名称也可以被显式地导入到HQL命名空间。甚至可以导入非显式映射的类和接口，因此短名称可以被用在多态的HQL查询中：

```
<hibernate-mapping>
    <import class="auction.model.Auditable" rename="IAuditable"/>
</hibernate-mapping>
```

现在可以用HQL查询（如from IAuditable）来获取实现auction.model.Auditable接口的类的所有持久化实例。（如果此时你不知道这个特性是否与你有关，别着急，本书稍后会回到查询的话题。）注意<import>元素，就像<hibernate-mapping>所有其他直接的子元素一样，是一个适用于整个应用程序的声明，因此你不必（并且不能）在其他映射文件中重复它。

可以利用注解给实体一个显式的名称，如果短名称会在JPA QL或者HQL命名空间中导致冲突的话：

```
@Entity(name="AuctionItem")
public class Item { ... }
```

现在来考虑命名的另一个方面：包的声明。

4.3.4 声明包名称

CaveatEmptor应用程序的所有持久化类都在Java包auction.model中声明。然而，你并不想每当在关联、子类或者组件映射中命名这个类或者任何其他类时都重复整个包名称。替换做法是，指定一个package属性：

```
<hibernate-mapping package="auction.model">
    <class name="Item" table="ITEM">
      ...
    </class>
</hibernate-mapping>
```

现在出现在这个映射文件中的所有未限定的类名都将用被声明的包名作为前缀。假设在本书的所有映射示例中都有这个设置，并给CaveatEmptor模型类使用未限定的类名。

类和表的名称必须谨慎选择。然而，已经选择的名称可能被SQL数据库系统保存，因此这个名称必须用引号把它括起来。

4.3.5　用引号把 SQL 标识符括起来

默认情况下，Hibernate没有在生成的SQL中用引号把表和列名括起来。这使得SQL稍微更容易阅读，也允许你利用这样的事实：大部分SQL数据库在比较没有用引号括起来的标识符时都是不区分大小写的。有时候，特别是在遗留数据库中，你会遇到包含奇怪字符或者空格的标识符，或者你希望强制区分大小写。或者，如果依赖Hibernate的默认设置，Java中的类名称或者属性名称可能被自动转化为数据库管理系统中不允许的表名称或者列名称。例如，User类被映射到USER表，而它通常是SQL数据库中保留的一个关键字。Hibernate不认识任何DBMS产品的SQL关键字，因此数据库系统在启动或者运行时抛出异常。

如果在映射文档中用反引号（backtick）把表名或者列名括起来，Hibernate就会始终在生成的SQL中把这个标识符用引号括起来。下列属性声明强制Hibernate用括起来的列名"DESCRIPTION"生成SQL。Hibernate也知道Microsoft SQL Server需要变量[DESCRIPTION]，以及MySQL需要`DESCRIPTION`。

```
<property name="description"
          column="`DESCRIPTION`"/>
```

除了把所有表名和列名用反引号括起来之外，再没有其他办法可以强制Hibernate在任何地方使用括起来的标识符了。任何可能的时候，都应该考虑用保留的关键字名称重新命名表或者列。用反引号括起来对注解映射有效，但它是Hibernate的一个实现细节，而不是JPA规范的一部分。

4.3.6　实现命名约定

我们经常遇到对数据库表名和列名有严格约定的组织。Hibernate提供一种允许自动强制执行命名标准的特性。

假设CaveatEmptor中的所有表名都应该遵循模式CE_<table name>。一种解决方案是手工在映射文件中的所有<class>和集合元素上指定table属性。然而，这种方法既费时又容易遗忘。替代做法是，实现Hibernate的NamingStrategy接口，如代码清单4-1所示。

代表清单4-1　NamingStrategy实现

```
public class CENamingStrategy extends ImprovedNamingStrategy {

      public String classToTableName(String className) {
            return StringHelper.unqualify(className);
      }

      public String propertyToColumnName(String propertyName) {
          return propertyName;
    }
```

```
    public String tableName(String tableName) {
          return "CE_" + tableName;
    }

    public String columnName(String columnName) {
        return columnName;
    }

    public String propertyToTableName(String className,
                                      String propertyName) {
          return "CE_"
                      + classToTableName(className)
                      + '_'
                      + propertyToColumnName(propertyName);
    }
}
```

你扩展了ImproveNamingStrategy，它对你不想从头开始实现的NamingStrategy的所有方法提供默认的实现（请查看API文档和源代码）。仅当<class>映射没有指定显式的table名称时才调用classToTableName()方法。如果属性没有显式的column名称，就调用propertyToColumnName()方法。当声明了显式的名称时，则调用tableName()和columnName()方法。

如果启用这个CENamingStrategy，类映射声明

```
<class name="BankAccount">
```

将导致CE_BANKACCOUNT作为表的名称。

然而，如果指定了表名，像这样：

```
<class name="BankAccount" table="BANK_ACCOUNT">
```

那么CE_BANK_ACCOUNT就是表的名称。此时，BANK_ACCOUNT就被传递到tableName()方法。

NamingStrategy接口的最好特性是动态行为的潜能。为了启用一个特定的命名策略，可以在启动时把一个实例传递到Hibernate的Configuration：

```
Configuration cfg = new Configuration();
cfg.setNamingStrategy( new CENamingStrategy() );
SessionFactory sessionFactory sf =
                    cfg.configure().buildSessionFactory();
```

这允许你有多个基于相同映射文件的SessionFactory实例，每一个都使用不同的NamingStrategy。这在一个多客户端的安装中极为有用，在这里每个客户端都需要唯一的表名（但要相同的数据模型）。然而，处理这种需求的更好方法则是使用一个SQL Schema（一种命名空间），就如3.3.4节中已经讨论过的那样。

可以在Java Persistence中，用hibernate.ejb.naming_strategy选项在persistence.xml文件中设置一种命名策略实现。

既然已经讨论了实体的概念和最重要的映射，接下来就映射值类型。

4.4 细粒度的模型和映射

在花费了本章前半部分篇幅专门讨论实体和各自基本的持久化类映射选项之后，现在我们把

重点放在它们各种形式的值类型上。马上想起来的两种不同的值类型是：JDK带来的值类型的类（例如String或者基本类型），以及由应用程序的开发人员定义的值类型的类（例如Address和MonetaryAmount）。

首先，映射使用JDK类型的持久化类属性，并学习基础的映射元素和属性。然后着手定制的值类型类，并把它们映射为可嵌入的组件。

4.4.1　映射基础属性

如果映射持久化类，无论它是实体还是值类型，所有持久化属性都必须在XML映射文件中被显式地映射。另一方面，如果用注解映射类，它的所有属性都被默认为是持久化的。可以用@javax.persistence.Transient注解给属性进行标识来把它们排除，或者使用transient的Java关键字（通常只给Java序列化排除字段）。

在一个JPA XML描述符中，可以排除一个特定的字段或者属性：

```
<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <transient name="age"/>
    </attributes>
</entity>
```

一个典型的Hibernate属性映射定义了一个POJO的属性名称、一个数据库列名和一个Hibernate类型的名称，并且它可能经常省略类型。因此，如果description是（Java）类型java.lang.String的一个属性，Hibernate就默认使用Hibernate类型string（第5章将回到Hibernate类型系统的话题）。

Hibernate用反射决定属性的Java类型。因而，下列映射是等价的：

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" column="DESCRIPTION"/>
```

如果列名与属性名相同（忽略大小写），那么列名甚至可以省略。（这就是前面提到过的有意义的默认特性之一。）

对于一些更不寻常的情况，稍后会见到更多，可能要用<column>元素代替XML映射中的column属性。<column>元素提供更多的灵活性：它有更多可选择的属性，且可能不止一次地出现。（单个属性可以映射到不止一个列，这是第5章要讨论的一项技术。）下列两个属性是等价的：

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" type="string">
    <column name="DESCRIPTION"/>
</property>
```

<property>元素（尤其<column>元素）也定义主要应用到自动的数据库Schema生成的某些属性。如果没有正在使用hbm2ddl工具（请见2.1.4节）生成数据库Schema，就可以放心地省略它们。但是最好至少包括not-null属性，因为还没有提交给数据库时，Hibernate就能够报告非法的空属性：

```
<property name="initialPrice" column="INITIAL_PRICE" not-null="true"/>
```

JPA是基于异常模型的一个配置，因此可以依赖默认。如果持久化类的一个属性没有被注解，就应用下列规则：

- 如果属性是JDK类型，它自动就是持久化的。换句话说，它在Hibernate XML映射文件中像<property name="propertyName"/>一样处理。
- 如果属性的类被注解为@Embeddable，它就被映射为自己的类的组件。本章稍后将讨论组件的嵌入。
- 如果属性的类型是Serializable，它的值以序列化的形式保存。这通常不是你想要的，应该始终映射Java类，而不是在数据库中保存一堆字节。想象一下当几年之后应用程序消失时，维护包含这些二进制信息的数据库的情景。

如果不想依赖这些默认，就在一个特定的属性上应用@Basic注解。@Column注解相当于XML的<column>元素。以下是如何按要求声明一个属性值的例子：

```
@Basic(optional = false)
@Column(nullable = false)
public BigDecimal getInitialPrice { return initialPrice; }
```

@Basic注解把属性标识为在Java对象级上不可选。第二个设置，列映射上的nullable = false，只负责NOT NULL（非空）数据库约束的生成。Hibernate JPA实现在任何情况下对待这两个选项都一视同仁，因此你也可以只用其中一个注解达到这一目的。

在JPA XML描述符中，这个映射看起来是相同的：

```
<entity class="auction.model.Item" access="PROPERTY">
    <attributes>
        ...
        <basic name="initialPrice" optional="false">
            <column nullable="false"/>
        </basic>
    </attributes>
</entity>
```

Hibernate元数据中有许多选项可以用来声明Schema约束，例如列上的NOT NULL。然而，除了简单的可为空的能力之外，它们只用于在Hibernate从映射元数据中导出数据库Schema时生成DDL。我们将在8.3节中讨论SQL（包括DDL）的定制。另一方面，Hibernate Annotations包包括一个更高级且更复杂的数据验证框架，不仅可以用它在DDL中定义数据库Schema约束，还可以用于运行时的数据验证。第17章将讨论它。

用于属性的注解永远都在访问方法中吗？

1．定制属性访问

类的属性由持久化引擎直接（通过字段）或者间接（通过获取方法和设置方法属性访问方法）访问。在XML映射文件中，用hibernate-mapping root element的default-access="field|property|noop|custom.Class"属性给类控制默认的访问策略。被注解的实体从强制的@Id注解的位置继承默认的访问策略。例如，如果@Id已经在字段中（而不是获取方法中）声明，那么所有其他的属性映射注解（如货品description属性的列名）也在字段中声明：

```
@Column(name = "ITEM_DESCR")
private String description;

public String getDescription() { return description; }
```

这是JPA规范定义的默认行为。然而，Hibernate允许利用@org.hibernate.annotations.AccessType(<strategy>)注解灵活地定制访问策略：

- 如果在类/实体级中设置AccessType，类的所有属性都根据选中的策略访问。属性级注解是在字段还是在获取方法上，这取决于策略。这个设置覆盖来自标准的@Id注解位置的任何默认值。
- 如果给字段访问默认或者显式地设置一个实体，字段中的AccessType（"property"）注解就把这个特定的属性转换为通过属性获取方法/设置方法的运行时访问。AccessType注解的位置仍然为字段。
- 如果给属性访问默认或者显式地设置一个实体，获取方法中的AccessType（"field"）注解就把这个特定的属性转换为通过同名字段的运行时访问。AccessType注解的位置仍然为获取方法。
- 任何@Embedded类都继承默认的或者显式声明的自己根实体类的访问策略。
- 任何@MappedSuperclass属性都通过被映射实体类的默认或者显式声明的访问策略而被访问。

也可以用access属性在Hibernate XML映射中控制属性级上的访问策略：

```
<property name="description"
          column="DESCR"
          access="field"/>
```

或者，可以用default-access属性给一个根<hibernate-mapping>元素内部的所有类映射设置访问策略。

除了字段和属性访问之外，另一个可能有用的策略是noop。它映射Java持久化类中不存在的属性。这听起来有点奇怪，但是它让你在HQL查询中指向这个“虚拟”属性（换句话说，只在HQL查询中使用数据库列）。

如果没有合适的内建访问策略，也可以通过实现接口org.hibernate.property.PropertyAccessor定义自己的定制属性访问策略。在access映射属性或者@AccessType注解中设置（完全匹配的）类名。看看Hibernate源代码寻求灵感；这是个很简单的练习。

有些属性根本不映射到列。特别是衍生属性（derived property），它从SQL表达式中取值。

2. 使用衍生属性

衍生属性的值在运行时计算，通过对利用formula属性定义的表达式求值。例如，可以把totalIncludingTax属性映射到SQL表达式：

```
<property name="totalIncludingTax"
          formula="TOTAL + TAX_RATE * TOTAL"
          type="big_decimal"/>
```

这个给定的SQL公式在每次从数据库获取实体时求值（并且在任何其他时间不求值，因此如果其他的属性被修改，这个结果就可能过时）。属性没有列属性（或者子元素），并且永远不会出

现在SQL的INSERT或者UPDATE中，而只在SELECT中。公式可能指向数据库表的列，它们可以调用SQL函数，甚至包括SQL子查询。SQL表达式实际上被传递到底层的数据库；如果你不够细心，并依赖特定于供应商的操作符或者关键字，这就是把映射文件绑定到一个特定数据库产品的好机会。

公式也可使用Hibernate注解：

```
@org.hibernate.annotations.Formula("TOTAL + TAX_RATE * TOTAL")
public BigDecimal getTotalIncludingTax() {
    return totalIncludingTax;
}
```

下列示例使用一个关联的子查询来计算一件货品所有出价的平均值：

```
<property
   name="averageBidAmount"
   type="big_decimal"
   formula=
    "( select AVG(b.AMOUNT) from
       BID b where b.ITEM_ID = ITEM_ID )"/>
```

注意未限定的列名指向衍生属性所属的这个类的表列。

另一种特殊的属性依赖数据库生成的值。

3. 生成的和默认的属性值

想象类的一个特定属性有着数据库生成的值，通常在第一次插入实体行的时候。典型的数据库生成的值是创建的时间戳、对一件货品的默认价格，以及每次修改时运行的触发器。

典型的Hibernate应用程序需要刷新包含由数据库为其生成值的任何属性的对象。然而，标识属性为已生成（generated），让应用程序把这个责任委托给了Hibernate。本质上，每当Hibernate给定义了已生成属性的实体执行SQL INSERT或者UPDATE时，它在获取已生成的值之后立即执行SELECT。使用property映射中的generated开关启用这个自动刷新：

```
<property name="lastModified"
          column="LAST_MODIFIED"
          update="false"
          insert="false"
          generated="always"/>
```

标记为数据库生成的属性还必须是非可插入和非可更新的，用insert和update属性进行控制它们。如果两者都设置为false，属性的列就永远不会出现在INSERT或者UPDATE语句中——属性值是只读的。而且，通常不在类中给不可变的属性添加公有的设置方法（并切换到字段访问）。

利用注解，通过@Generated的Hibernate注解声明不可变性（和自动刷新）：

```
@Column(updatable = false, insertable = false)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
private Date lastModified;
```

可用的设置为GenerationTime.ALWAYS和GenerationTime.INSERT，并且XML映射中与之相当的选项是generated="always"和generated="insert"。

数据库生成的属性值的一个特殊情况是默认值。例如，你可能想要实现一种规则，即每件拍卖货品的成本至少为$1。首先，得把它作为INITIAL_PRICE列的默认值添加到数据库目录：

```
create table ITEM (
    ...
    INITIAL_PRICE number(10,2) default '1',
    ...
);
```

如果使用Hibernate的Schema导出工具hbm2ddl，则可以通过把default属性添加到属性映射来启用这项输出：

```
<class name="Item" table="ITEM"
       dynamic-insert="true" dynamic-update="true">
    ...
    <property name="initialPrice" type="big_decimal">
        <column name="INITIAL_PRICE"
                default="'1'"
                generated="insert"/>
    </property>
    ...
 </class>
```

注意，你还必须启用动态的插入并更新语句的生成，以便包含默认值的列不会被包括在每个语句中，如果它的值为null（否则将插入NULL，而不是默认值）。此外，已经被变成持久化但尚未清除到数据库，且尚未再次刷新的Item的一个实例，将不在对象属性上设置默认值。换句话说，你需要执行一个显式的清除：

```
Item newItem = new Item(...);
session.save(newItem);

newItem.getInitialPrice();    // is null

session.flush();              // Trigger an INSERT
// Hibernate does a SELECT automatically

newItem.getInitialPrice();    // is $1
```

因为设置了generated="insert"，Hibernate知道读取数据库生成的属性值立即需要一个额外的SELECT。

可以用注解把默认的列值映射为列的DDL定义的一部分：

```
@Column(name = "INITIAL_PRICE",
        columnDefinition = "number(10,2) default '1'")
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
private BigDecimal initalPrice;
```

columnDefinition属性包括列DDL的完整属性，包含数据类型和所有约束。记住，一个实际上不可移植的SQL数据类型可能把注解映射绑定到一个特定的数据库管理系统。

8.3节将回到约束和DDL定制的主题。

接下来，将映射用户自定义的值类型的类。如果在两个类之间搜索一种复合关系的话，可以

很轻松地在UML类图中认出它们。它们其中一个是依赖的类，即组件。

4.4.2 映射组件

目前为止，对象模型的类已经全部成为实体类，每一个都有自己的生命周期和同一性。然而，User类与Address类有着一种特殊的关联，如图4-2所示。

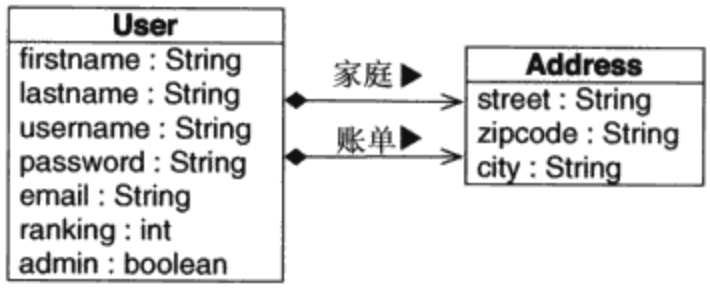

图4-2 使用复合的User和Address之间的关系

从对象建模的角度来看，这个关联是一种聚集——是整体-部分（part-of）的关系。聚集是一种强健的关联形式；它对于对象的生命周期有一些额外的语义。在这种情况下，我们有一种更强健的形式：复合（composition），在这里部分的生命周期完全依赖于整体的生命周期。

对象建模专家和UML设计师声称，就实际的Java实现而言，在这个复合和其他更弱形式的关联之间是没有区别的。但是在ORM的上下文中，则有一个很大的区别：复合的类经常是一个备选的值类型。

把Address映射为值类型，把User映射为实体。这影响POJO类的实现吗？

Java没有复合的概念——类或者属性无法被标记为组件或者复合。唯一的区别是对象标识符：组件没有独立的同一性，因此持久化组件类不需要标识符属性或者标识符映射。它是一个简单的POJO：

```
public class Address {

    private String street;
    private String zipcode;
    private String city;

    public Address() {}

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    public String getZipcode() { return zipcode; }
    public void setZipcode(String zipcode) {
        this.zipcode = zipcode; }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

}
```

User和Address之间的复合是一个元数据级的概念；你只需要告诉Hibernate，Address是映射文档中的值类型，或者是包含注解的值类型。

1. XML格式的组件映射

Hibernate对用户自定义的类使用术语component（组件），该类被持久化到与自己的实体相同的表，代码清单4-2中显示了这样的一个例子。（此处单词component的用法，与建筑级的概念没

有任何关系，而是同软件组件中的概念。）

代码清单4-2　包含组件Address的User类的映射

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

    <id name="id" column="USER_ID" type="long">
        <generator class="native"/>
    </id>

    <property name="loginName" column="LOGIN" type="string"/>

    <component name="homeAddress" class="Address">          ❶
        <property name="street" type="string"
                  column="HOME_STREET" not-null="true"/>
        <property name="city" type="string"
                  column="HOME_CITY" not-null="true"/>
        <property name="zipcode" type="string"
                  column="HOME_ZIPCODE" not-null="true"/>
    </component>

    <component name="billingAddress" class="Address">       ❷
        <property name="street" type="string"
                  column="BILLING_STREET" not-null="true"/>
        <property name="city" type="string"
                  column="BILLING_CITY" not-null="true"/>
        <property name="zipcode" type="string"
                  column="BILLING_ZIPCODE" not-null="true"/>
    </component>

    ...

</class>
```

❶ 在<component>元素的内部声明Address的持久化属性。User类的属性被命名为homeAddress。

❷ 重用同一个组件类，把这个类型的另一个属性映射到同一张表。

图4-3显示了Address类的属性如何被持久化到与User实体相同的表中。

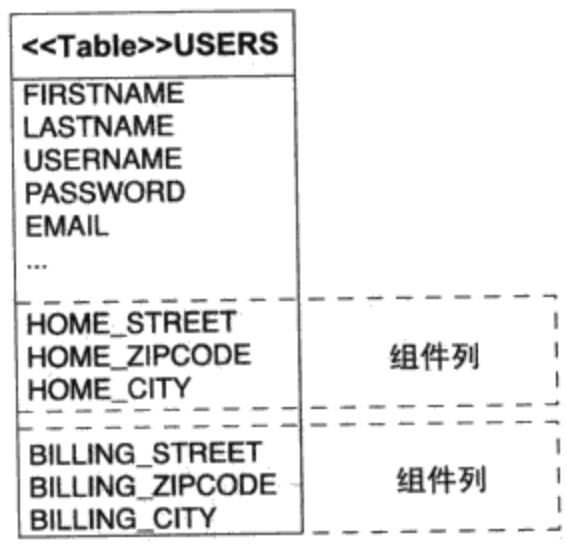

图4-3　包含Address组件的User的表属性

注意，在这个例子中，你把复合关联建模为单向的。你无法从Address导航到User。Hibernate支持单向的和双向的复合，但是单向复合要常见得多。双向映射的一个例子如代码清单4-3所示。

代码清单4-3 把后退指针添加到复合

```
<component name="homeAddress" class="Address">
    <parent name="user"/>
    <property name="street" type="string"
              column="HOME_STREET" not-null="true"/>
    <property name="city" type="string"
              column="HOME_CITY" not-null="true"/>
    <property name="zipcode" type="stringshort"
              column="HOME_ZIPCODE" not-null="true"/>
</component>
```

在代码清单4-3中，<parent>元素把类型User的属性映射到自己的实体，在这个例子中，它是具名user的那个属性。然后可以调用Address.getUser()在另一个方向进行导航。这真是一个简单的后退指针。

Hibernate组件可以拥有其他的组件，甚至关联到其他实体。这种灵活性是Hibernate支持细粒度的对象模型的基础。例如，可以创建Location类，包含关于Address所有者的家庭住址的详细信息：

```
<component name="homeAddress" class="Address">
    <parent name="user"/>

    <component name="location" class="Location">
        <property name="streetname" column="HOME_STREETNAME"/>
        <property name="streetside" column="HOME_STREETSIDE"/>
        <property name="housenumber" column="HOME_HOUSENR"/>
        <property name="floor" column="HOME_FLOOR"/>
    </component>

    <property name="city" type="string" column="HOME_CITY"/>
    <property name="zipcode" type="string" column="HOME_ZIPCODE"/>

</component>
```

Location类的设计相当于Address类。你现在有3个类、1个实体和2个值类型，全部被映射到同一张表。

现在用JPA注解映射组件。

2. 注解被嵌入的类

Java Persistence规范把组件称为被嵌入的类。为了用注解映射被嵌入的类，可以在自己的实体类中声明一个特定的属性为@Embedded，在这个例子中是User的homeAddress：

```
@Entity
@Table(name = "USERS")
public class User {

    ...
```

```
    @Embedded
    private Address homeAddress;

    ...
}
```

如果没有把一个属性声明为@Embedded，并且它不是JDK类型，Hibernate就在被关联的类中查找@Embedded注解。如果它存在，属性就自动地被映射为一个依赖的组件。

这就是被嵌入的类的样子：

```
@Embeddable
public class Address {

    @Column(name = "ADDRESS_STREET", nullable = false)
    private String street;

    @Column(name = "ADDRESS_ZIPCODE", nullable = false)
    private String zipcode;

    @Column(name = "ADDRESS_CITY", nullable = false)
    private String city;

    ...
}
```

可以在被嵌入的类中进一步定制独立的属性映射，例如用@Column注解。现在USERS表除了包含其他的之外，还包含列ADDRESS_STREET、ADDRESS_ZIPCODE和ADDRESS_CITY。任何包含组件字段的其他实体表（假设，Order类也有Address）都使用相同的列选项。也可以添加后退指针属性到Address被嵌入的类，并用@org.hibernate.annotations.Parent映射它。

有时候，你会想要从外部对一个特定的实体覆盖在被嵌入的类内部所做的设置。例如，下面是如何重命名列的例子：

```
@Entity
@Table(name = "USERS")
public class User {

    ...

    @Embedded
    @AttributeOverrides( {
        @AttributeOverride(name   = "street",
                           column = @Column(name="HOME_STREET") ),
        @AttributeOverride(name   = "zipcode",
                           column = @Column(name="HOME_ZIPCODE") ),
        @AttributeOverride(name   = "city",
                           column = @Column(name="HOME_CITY") )
    })
    private Address homeAddress;

    ...
}
```

User类中的这个新的@Column声明覆盖了被嵌入的类的设置。注意，被嵌入的@Column注解

上的所有属性都被替换，因此它们不再nullable = false。

在JPA XML描述符中，被嵌入的类和复合的映射看起来像下面这样：

```
<embeddable class="auction.model.Address access-type="FIELD"/>

<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <embedded name="homeAddress">
            <attribute-override name="street">
                <column name="HOME_STREET"/>
            </attribute-override>
            <attribute-override name="zipcode">
                <column name="HOME_ZIPCODE"/>
            </attribute-override>
            <attribute-override name="city">
                <column name="HOME_CITY"/>
            </attribute-override>
        </embedded>
    </attributes>
</entity>
```

对于类被映射为组件有2条重要的限制。第一，就像对于所有的值类型一样，共享引用是不可能的。组件homeAddress没有它自己的数据库同一性（主键），因此无法被除了User的包含实例之外的任何对象引用。

第二，没有优雅的方式表示对Address的空引用。作为任何优雅的方法的替代，Hibernate在组件的所有被映射列中都把空组件表示为空值。这意味着如果保存一个包含全部空属性值的组件对象，Hibernate就会在从数据库中获取自己的实体对象时返回一个空组件。

通读本书，你将发现更多的组件映射（甚至它们的集合）。

4.5 小结

本章介绍了实体和值类型之间的本质区别，以及这些概念如何影响领域模型实现为持久化的Java类。

实体是系统更粗粒度的类。它们的实例有一个独立的生命周期和它们自己的同一性，可以被许多其他的实例引用。另一方面，值类型依赖于特定的实体类。值类型的实例有着受它自己的实体实例限制的生命周期，并且它只能被一个实体引用——没有独立的同一性。

我们探讨了Java同一性、对象等同性和数据库同一性，以及是什么创造了好主键。你学习了Hibernate给主键值内建了哪些生成器，以及如何使用和扩展这个标识符系统。

你还学习了各种（主要可选的）类映射选项，最后学习了基础的属性和值类型组件在XML映射和注解中如何被映射。

为了方便起见，表4-2概括了本章讨论过的Hibernate和Java Persistence相关概念之间的区别。

表4-2　第4章的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
实体类型和值类型的类是支持富且细粒度的领域模型的本质概念	JPA规范也是如此，但是称值类型为“可嵌入的类”。然而，嵌套的可嵌入类被认为是一项不可移植的特性
Hibernate支持10个开箱即用的标识符生成策略	JPA标准化了4个标识符生成器的一个子集，但是允许供应商进行扩展
Hibernate可以通过字段、访问方法或者用任何定制的PropertyAccessor实现访问属性。可以对一个特定的类混合使用这些策略	JPA标准化了通过字段或者访问方法的属性访问，并且不能对没有Hibernate扩展注解的特定类混合使用这些策略
Hibernate支持公式属性和数据库生成的值	JPA不包括这些特性，需要Hibernate扩展

第5章将探讨继承，以及实体类的层次结构如何通过各种策略映射。我们也将讨论Hibernate映射类型系统，在几个实例中展现值类型的转换器。