表2-1　Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
与所有东西在任何地方整合。灵活，但有时配置复杂	适用于Java EE和Java SE。简单和标准的配置：在Java EE环境中无需额外的整合或者特殊的配置
配置需要一系列XML映射文件或者被注解的类	JPA提供程序自动扫描XML映射文件和被注解的类
私有但是强大。持续改善原生的编程接口和查询语言	标准和稳定的接口，包含一个充分的Hibernate功能的子集。有可能轻松地退回到Hibernate API

第3章将介绍一个更复杂的示例应用程序，本书后续部分将始终使用这个应用程序。你将学习如何设计和实现一个领域模型，以及哪些映射元数据选项在大项目中是最好的选择。

第3章

领域模型和元数据

本章内容

- CaveatEmptor示例应用程序
- 富领域模型的POJO设计
- ORM元数据选项

前一章的“Hello World”示例介绍了Hibernate；然而，它对理解包含复杂数据模型的现实应用程序的需求并没有帮助。本书后续内容将用更复杂的示例应用程序——CaveatEmptor，一个在线拍卖系统——来演示Hibernate和Java Persistence。

我们对应用程序的讨论从给持久化类引入编程模型开始。设计和实现持久化类是一个多步骤的过程，我们将详细阐述。

首先，将学习如何识别问题领域的业务实体。创建这些实体及其属性的一个概念模型，称为领域模型，通过创建持久化类在Java中实现它。我们花点时间探讨这些Java类看起来到底应该是什么样子，并看一下这些类的持久化能力，以及这个方面如何影响设计和实现。

然后探讨映射元数据选项——可以告诉Hibernate你的持久化类及其属性如何与数据库表和列联系起来的方法。这可能涉及编写XML文档，该文档最终与编译好的Java类一起部署，并在运行时被Hibernate读取。另一种选项基于EJB 3.0标准，直接在持久化类的Java源代码中使用JDK 5.0元数据注解。读完本章之后，你将知道如何在复杂的现实项目中设计领域模型的持久化部分，以及你主要喜欢和使用哪种映射元数据选项。

最后，在本章的最后一节（可能是可选的），看看Hibernate独立表示的能力。Hibernate中一项相对较新的特性允许你在Java中创建完全动态的领域模型，例如一个没有任何具体类而只有`HashMap`的模型。Hibernate也支持使用XML文档的领域模型表示法。

让我们从示例应用程序开始。

3.1 CaveatEmptor 应用程序

CaveatEmptor在线拍卖应用程序演示了ORM技术和Hibernate功能；可以从http://caveatemptor.hibernate.org中下载应用程序的源代码。本书将不太关注用户界面（它可能基于Web或者富客户端

程序）；而是集中在数据访问代码上。然而，当必须做出有关影响用户界面的数据访问代码的设计决策时，我们一般两者都考虑。

为了理解ORM中涉及的设计问题，先假设这个CaveatEmptor应用程序还不存在，你正从头开始构建。第一个任务就是分析。

3.1.1　分析业务领域

软件开发的工作从问题领域的分析开始（假设没有已经存在的遗留代码或者遗留数据库）。

在这个阶段，你在问题领域专家的帮助下，辨别与软件系统有关的主要实体。实体通常是为系统用户所理解的概念：付款、客户、订单、货品、出价等。有些实体可能是用户考虑的更不具体的事物的抽象概念，例如定价算法，但是即使这些通常对于用户也是可以理解的。所有这些实体都在业务的概念视图中，我们有时候称它为业务模型。开发人员和面向对象的软件架构师分析业务模型，并创建面向对象的模型，这些工作仍然处于概念阶段（没有Java代码）。这个模型可能像仅存在于开发人员心中的意象（mental image）一样简单，或者可能像通过计算机辅助软件工程（Computer-Aided Software Engineering，CASE）工具如ArgoUML或者TogetherJ创建的UML类图一样精细。用UML表达的一个简单模型如图3-1所示。

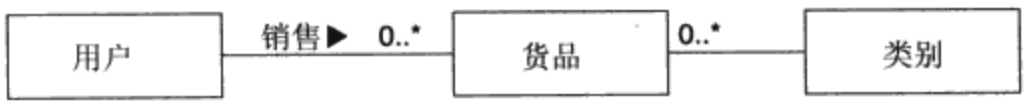

图3-1　典型在线拍卖模型的一个类图

这个模型包含了你一定要在任何典型的拍卖系统中找到的实体：类别、货品和用户。这些实体和它们的关系（可能还有它们的属性）都由问题领域的这个模型表示。我们称这种来自问题领域的面向对象的实体模型（仅包含用户关注的那些实体）为领域模型。它是真实世界的一个抽象视图。

领域模型分析和设计的目标是，为应用程序捕捉业务信息的本质。开发人员和架构师们可能不是从面向对象的模型开始，而是从一个数据模型开始应用程序设计（可能用一个实体关系图表达）。我们通常说，关于持久化，在这两者之间稍有不同；它们不过是不同的起点。最终，我们最关注业务实体的结构和关系，以及必须用来保证数据完整性的规则（例如，关系的多样性），以及用来操作数据的逻辑。

在对象模型中，有个关注点是多态的业务逻辑。对于我们以及自顶向下的开发方法而言，如果能够在多态的Java中实现逻辑模型会很有帮助；因此，初稿是一个面向对象的模型。然后衍生为逻辑关系数据模型（通常没有额外的图），并实现真正的物理数据库Schema。

来看看CaveatEmptor应用程序的问题领域的分析结果。

3.1.2　CaveatEmptor 领域模型

CaveatEmptor网站拍卖许多不同种类的货品，从电子设备到机票，应有尽有。拍卖根据英国的拍卖策略进行：用户连续在一件货品上出价，直到那件货品的出价期终止，最高的出价者胜出。

在任何商店中，货物都是按照类别分类，并把类似的货物集中到一个分区或者架子上。拍卖目录需要某种货品类别的层次，以便买家能够浏览这些类别或者按类别和货品属性任意搜索。货品清单显示在目录浏览器中，并搜索结果屏幕。从清单中选择一件货品，把买家带到一个货品细节的视图中。

一次拍卖由一连串的出价组成，但只有一个胜出。用户细节包括姓名、注册ID、地址、电子邮件地址和账单信息。

信任网（Web of Trust）是一个在线拍卖网站的基本特征。信任网允许用户对可信度（或者不可信度）建立信誉。买家可以对卖家进行评价（卖家也可以对买家进行评价），并且所有其他用户都可以看见评价。

领域模型的高级概览如图3-2所示。我们来简要地讨论这个模型中值得关注的一些特性。

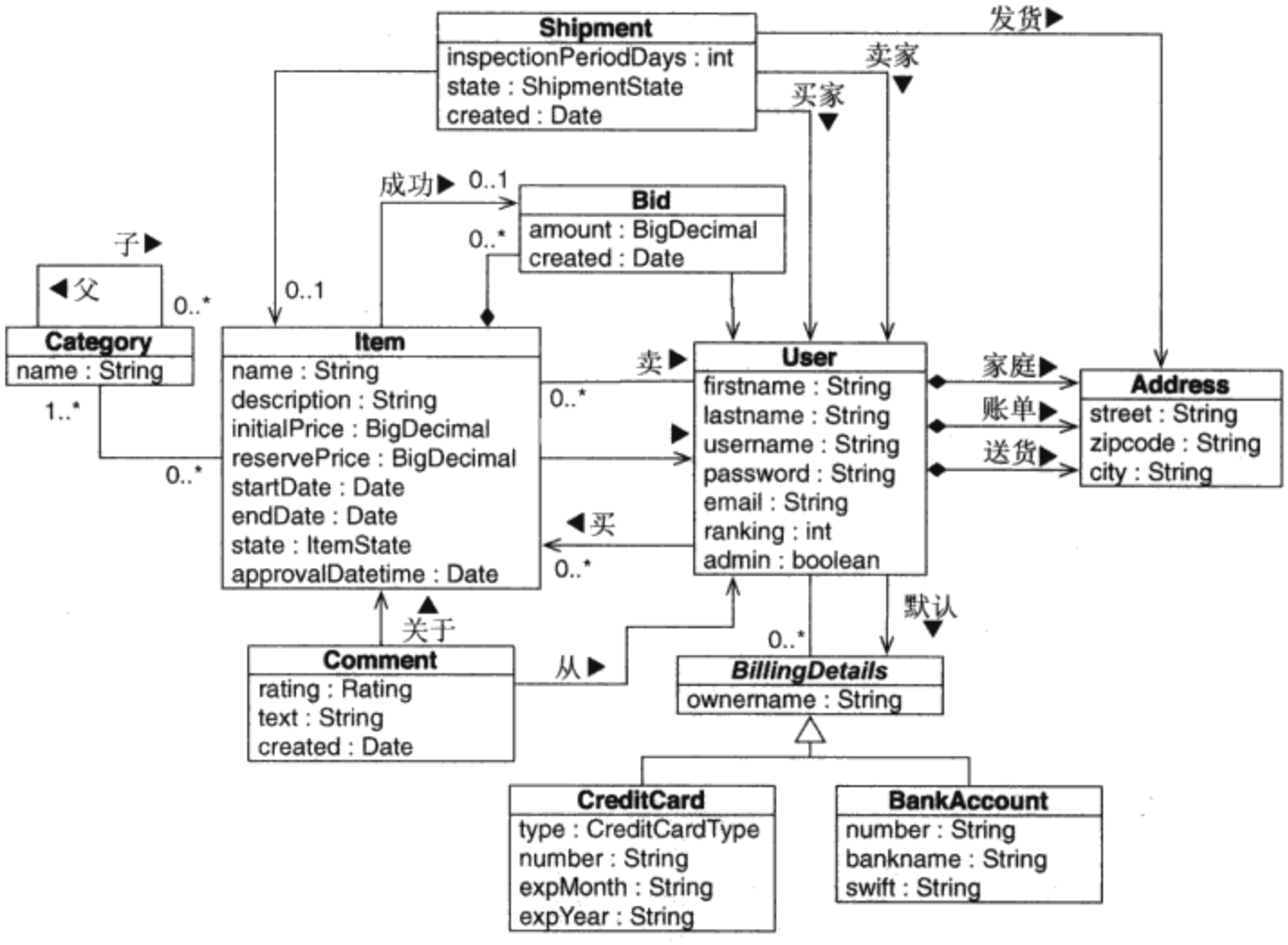

图3-2 CaveatEmptor领域模型及其关系的持久化类

每件货品只能被拍卖一次，因此你不需要把Item与任何拍卖实体区分开来。反之，你有单个拍卖货品实体命名为Item。这样，Bid直接与Item关联。Users只可以在一个拍卖上下文中写有关其他用户的Comments；因此Item和Comment之间有了关联。User的Address信息被当作一个单独的类建模，即使User可能只有一个Address；它们也可能有3个地址，分别为家庭住址、付账地址和收货地址。你一定要允许用户有多个BillingDetails。各种付款策略被表示为一个抽

象类的子类（允许未来进行扩展）。

Category可以被嵌套在另一个Category内部。这通过一个递归关联表达，从Category实体到它自身。注意单个Category可以有多个子类，但是最多只有一个父类。每个Item至少属于一个Category。

领域模型中的实体应该封装状态和行为。例如，User实体应该定义客户的姓名和地址，以及计算货品运输成本（对于这个特定的客户）所需的逻辑。领域模型是一个富对象模型，包含复杂的关联、交互和继承关系。关于使用领域模型的面向对象方法的一个有趣且详细的讨论，请参阅这两本书：*Patterns of Enterprise Application Architecture*（Fowler，2003）[①]或者*Domain-Driven Design*（Evans，2003）[②]。

本书将不过多讨论业务规则或者领域模型的行为。这不是因为我们认为它不重要，而是因为这个关注点与持久化问题最正交（orthogonal）。它是我们持久化实体的状态，因此我们把讨论的重点放在如何在领域模型中最好地表示状态，而不是关于如何表示行为上。例如，本书不关注售出的物品如何计算税，或者系统如何确认新的用户账号。而是更关注用户和他们所卖物品之间的关系如何表示，以及如何使这个关系持久化。后面的几章会在深入讨论分层应用程序设计和逻辑分离以及数据访问时，重新讨论这个问题。

说明　没有领域模型的ORM——我们强调具有完全ORM的对象持久化最适合基于富领域模型的应用程序。如果你的应用没有实现复杂的业务规则或者实体之间的复杂事务（或者如果你的实体不多），可能不需要领域模型。许多简单的和有些不那么简单的问题特别适合面向表的解决方案，其中的应用程序是围绕数据库数据模型，而不是围绕一个面向对象的领域模型而设计的，通常在数据库（存储过程）中执行逻辑。然而，领域模型越复杂、越富有表现力，你就越能从利用Hibernate中受益；在处理对象/关系持久化的全复杂性时，它更大放异彩。

既然有了一个（初步的）包含领域模型的应用程序，下一步就要在Java中实现它。来看一些需要考虑的东西。

3.2　实现领域模型

在Java中实现领域模型时，必须处理几个典型的问题。例如，如何把业务关注点与横切关注点（例如事务和持久化）分开？你需要自动的还是透明的持久化？必须使用一个特定的编程模型来实现这一点吗？本节将探讨这类问题，以及在典型的Hibernate应用程序中如何处理这些问题。

我们从任何实现都必须处理的一个问题开始：关注点的分离。领域模型实现通常是一个中心的组织组件；每当实现新的应用程序功能时，都要大量重用它。基于这个原因，你应该准备想尽办法确保除了业务方面的关注点之外，其余都不要渗透进领域模型实现。

① 中文版《企业应用架构模式》由机械工业出版社出版，影印版已由中国电力出版社出版。——编者注

② 中文版《领域驱动设计》已由清华大学出版社出版，英文影印版已由人民邮电出版社出版。——编者注

3.2.1 处理关注点渗漏

领域模型实现是如此重要的一块代码，它不应该依赖于正交的Java API。例如，领域模型中的代码不应该执行JNDI查找或者通过JDBC API调用数据库。这事实上允许你在任何地方重用领域模型实现。最重要的是，它使得对领域模型进行单元测试变得更加容易，而不需要特定的运行时环境或者容器（或者需要虚拟任何服务依赖）。这种分离强调了逻辑单元测试和集成单元测试之间的区别。

假设领域模型应该只注重给业务领域建模。但是还有其他的关注点，例如持久化、事务管理和授权。你不应该把处理这些横切关注点的代码放在实现领域模型的那些类中。当这些关注点开始出现在领域模型类中时，就是一个关注点渗透的例子。

EJB标准解决了渗透关注点的问题。如果用实体编程模型实现领域类，容器就会替你顾及一些关注点（或者至少让你把那些关注点具体化到元数据中，例如注解或者XML描述符）。EJB容器利用拦截（interception）防止某些横切关注点的渗透。EJB是一个托管的组件，在EJB容器内部执行；容器拦截对bean的调用，并执行它自己的功能。这种方法允许容器以一种普通的方式实现预设的横切关注点——安全性、并发性、持久化、事务和远程。

不幸的是，EJB 2.1规范在有关你必须如何实现领域模型方面强加了许多规则和约束。这本身就是一种关注点的渗透——本例中，容器实现者的关注点就已经渗透了！这一点在EJB 3.0规范中已经得到了解决，它是非侵入性的，并且更接近传统的JavaBean编程模型。

Hibernate不是一个应用程序服务器，它不会试图实现全部EJB规范的所有横切关注点。Hibernate是只针对持久化这个关注点的一种解决方案。如果你需要声明安全性和事务管理，应该通过会话bean访问实体实例，利用这些关注点的EJB容器实现。EJB容器中的Hibernate取代了（EJB 2.1，使用CMP的实体bean）或者实现了（EJB 3.0，Java Persistence实体）持久化方面。

Hibernate持久化类和EJB 3.0实体编程模型提供透明的持久化。Hibernate和Java Persistence还提供自动的持久化。

我们来更详细地探讨这两个术语，并找出一种准确的定义。

3.2.2 透明和自动持久化

我们使用透明（transparent）一词，意味着领域模型的持久化类和持久化逻辑之间一个完整的关注点分离，在这里持久化类不知道且不依赖于持久化机制。我们使用自动（automatic）一词，是指让你不用处理低级机械化细节的一种持久化解决方案，例如编写大部分SQL语句和利用JDBC API工作。

例如，`Item`类就没有任何代码依赖于任何Hibernate API。此外：

- Hibernate不需要任何特殊的超类或者接口被持久化类继承或者实现。也没有任何特殊的类被用来实现属性或者关联。（当然，你始终有选择使用这两种技术的权利。）透明的持久化改善了代码的可读性和维护性，就如你很快要见到的。
- 持久化类可以在持久化上下文之外被重用，例如在单元测试或者在用户界面（UI）层中。

可测试性是使用富领域模型应用程序的一个必要条件。

- 在一个包含透明持久化的系统中，对象不知道底层的数据仓库；它们甚至不需要知道它们正被持久化或者获取。持久化关注点被具体化到一个普通的持久化管理器接口——在Hibernate中，是Session和Query。在JPA中，EntityManager和Query（有相同的名称，但是有着不同的包，以及略微不同的API）起着同样的作用。

透明的持久化产生了一定程度的可移植性：不用特殊的接口，持久化类从任何特定的持久化解决方案中分离出来。我们的业务逻辑在任何其他的应用程序上下文中都是可以重用的。可以轻松地切换到另一种透明的持久化机制。由于JPA遵循同样的基础原则，Hibernate持久化类和JPA实体类之间没有区别。

根据透明持久化的这个定义，某些非自动的持久层也成了透明的（例如DAO模式），因为它们用抽象的编程接口分离与持久化相关的代码。只有没有依赖的简单Java类公开给业务逻辑或者包含业务逻辑。反之，有些自动的持久层（包括EJB 2.1实体实例和一些ORM解决方案）是非透明的，因此它们需要特殊的接口或者侵入的编程模型。

我们认为透明是必需的。透明的持久化应该成为任何ORM解决方案的主要目标。然而，没有任何自动的持久化解决方案是完全透明的：每个自动的持久层（包括Hibernate）都在持久化类上强加了一些要求。例如，Hibernate要求把集合值（collection-valued）属性定义为接口（如java.util.Set或者java.util.List），而不是定义为如java..util.HashSet这样的实际实现（不管怎么说，这是一种好实践）。或者，JPA实体类必须有一个特殊的属性，称作数据库标识符。

现在你知道为什么持久化机制应该尽可能不影响你如何实现领域模型了，并且知道透明和自动的持久化是必需的。应该使用哪种编程模型呢？要遵守哪些具体的要求和约定？真的需要特别的编程模型吗？理论上而言，不必；但是实际应用中，应该采用一种被Java社区广为接受的遵守规则的、一致的编程模型。

3.2.3　编写 POJO 和持久化实体类

作为对EJB 2.1实体实例的反应，许多开发人员开始谈论POJO（Plain Old Java Object，简单Java对象）[①]，这是一种本质上复兴JavaBean的基本（back-to-basics）的方法，是一种用于UI开发的组件模型，并把它重新应用给业务层。（现在大部分开发人员几乎把术语POJO和JavaBean同义地使用。）EJB规范的修订给我们带来了新的轻量级实体，称它们为“能够持久化的JavaBean”是很恰当的。Java开发人员不久将把所有这三个术语作为同义词用给同一种基础的设计方法。

在本书中，我们给任何能持久化实例的类实现使用持久化类；如果有些Java最佳实践是相关的，就用POJO；当Java实现遵循EJB 3.0和JPA规范时则用实体类。再次强调，你不应该太关注这些区别，因为最终的目标是尽可能透明地应用持久化方面。几乎每一个Java类都可以是持久化类，或者POJO，或者如果遵循一些好的实践的话，也可以是实体类。

① POJO有时候也写成Plain Ordinary Java Objects。这个术语于2002年由Martin Fowler、Rebecca Parsons和Josh Mackenzie提出。

Hibernate与用POJO实现的领域模型合作得最好。Hibernate强加给领域模型实现的少数必备条件，对于POJO实现也是最佳实践，因此大部分POJO不用任何改变就可以与Hibernate兼容。Hibernate的必备条件几乎与EJB 3.0实体类的相同，因此POJO实现可以轻松地用注解标识，并创造一个EJB 3.0兼容的实体。

POJO声明了定义行为的业务方法和表示状态的属性。有些属性表示与其他用户自定义的POJO的关联。

一个简单的POJO类如代码清单3-1所示。这是领域模型User实体的一个实现。

代码清单3-1 User类的POJO实现

```
public class User
        implements Serializable {          <-- Serializable的声明

    private String username;
    private Address address;

    public User() {}                       <-- 无参类构造函数

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public MonetaryAmount calcShippingCosts(Address fromLocation) {
        ...
    }                                      <-- 业务方法

}
```


Hibernate不要求持久化类实现Serializable（可序列化）。然而，当对象被存储在一个HttpSession中，或者用RMI按值传递时，就需要序列化。（这可能发生在Hibernate应用程序中。）类可以是抽象的，必要时，可以扩展非持久化的类。

不同于JavaBeans规范，它不需要特定的构造函数，而Hibernate（和JPA）则要求每个持久化类都有个无参构造函数。Hibernate在这个构造函数上使用Java Reflection API调用持久化类来实例化对象。构造函数可以是非公共的，但必须至少是包可见（package-visible）的，如果运行时生成的代理要用于性能优化的话。代理生成也要求这个类不作final声明（也没有final方法）！（13.1节将回到代理的主题。）

POJO的属性实现了业务实体的属性——例如，User的username。属性通常被实现为私有的或者受保护的实例变量，以及公共的属性访问方法：一种获取实例变量的值的方法，以及改变它

的值的方法。这些方法相应称作获取方法和设置方法。代码清单3-1中的示例POJO给username和address属性声明了获取方法和设置方法。

JavaBean规范定义了命名这些方法的指导方针，允许普通的工具（如Hibernate）轻松地发现和操作属性值。获取方法的名称以get开头，接着是属性名称（首字母大写）；设置方法的名称以set开头，并且类似地也跟着属性名称。用于Boolean属性的获取方法可以用is而不是get开头。

可以选择持久化类的实例状态应该如何被Hibernate持久化，通过直接访问它的字段，或者访问方法。你的类设计不受这些考虑因素的干扰。可以使一些访问方法为非公共的，或者把它们完全移除。有些获取方法和设置方法完成一些比访问实例变量更复杂的事（例如验证），但是琐碎的访问方法很普遍。它们的主要优势是在类的内部表示法和公共接口之间提供一个额外的缓冲，允许两者的独立重构。

代码清单3-1中的示例也定义了一种把一件货品运输到一个特定用户的成本计算的方法（我们省略了这种方法的实现）。

JPA实体类的必备条件是什么？好消息是，目前为止对POJO所讨论的所有约定，也都是JPA实体的必备条件。你必须应用一些额外的规则，但是它们同样简单；稍后会回到这些话题。

既然已经涵盖了用POJO持久化类作为编程模型的基础知识，现在来看看如何处理那些类之间的关联。

3.2.4　实现 POJO 关联

你用属性表达POJO类之间的关联，并用访问方法在运行时从一个对象到一个对象进行导航。考虑由Category类定义的那些关联，如图3-3所示。

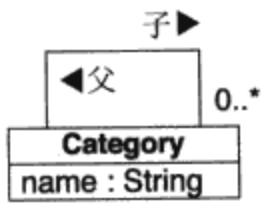

图3-3　包含关联的Category类图

和所有的图一样，上图省略了与关联相关的属性（我们称之为parentCategory和childCategories），因为它们会把图弄乱。操作它们值的这些属性和方法也称作脚手架代码（scaffolding code）。

这就是Category类一对多自关联的脚手架代码：

```
public class Category {
    private String name;
    private Category parentCategory;
    private Set childCategories = new HashSet();

    public Category() { }
    ...
}
```

为了允许关联的双向导航，你需要两个属性。parentCategory字段实现关联的单值端，并且被声明为Category类型。多值端，由childCategories字段实现，必须是集合类型。由于不允许重复，你选择一个Set（集），并把实例变量初始化为HashSet的一个新实例。

Hibernate需要用于集合类型属性的接口，因此必须使用java.util.Set或者java.util.List，而不是HashSet。这与实体中集合的JPA规范的要求相一致。运行时，Hibernate用它自己的一个类的实例把HashSet实例包起来。（这个特殊的类对于应用程序代码是不可见的。）不管怎样，这是集合接口编程的一个好实践，而不是具体的实现，因此这个限制不应该烦扰你。

现在你有了一些私有的实例变量，但是Hibernate没有公共的接口允许从业务代码或者属性管理中访问（如果它不应该直接访问字段的话）。来给这个类添加一些访问方法：

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Set getChildCategories() {
    return childCategories;
}

public void setChildCategories(Set childCategories) {
    this.childCategories = childCategories;
}

public Category getParentCategory() {
    return parentCategory;
}

public void setParentCategory(Category parentCategory) {
    this.parentCategory = parentCategory;
}
```

只有当这些访问方法是持久化类——被应用逻辑用来在两个对象之间创建一种关系——的外部接口的一部分时，它们需要再次被声明为public。然而，管理两个Category实例之间的链接，比在数据库字段中设置外键值更难。依据我们的经验，开发人员经常不知道从一个包含双向引用的网络对象模型中所产生的这种复杂性。我们将一步一步地探讨这个问题。

把一个子Category添加给一个父Category的基本过程看起来像这样：

```
Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategories().add(aChild);
```

每当在父Category和子Category之间创建一个链接时，需要两个动作：

- 必须设置子类的parentCategory，有效地破坏子类和它的旧父类之间的关联（任何子类都只能有一个父类）。
- 子类必须被添加到这个新的父Category的childCategories集合中。

说明　Hibernate中的托管关系——Hibernate不管理持久化关联。如果你要操作一个关联，必须编写与没有Hibernate时要编写完全相同的代码。如果关联是双向的，则关系的两侧都必须考虑。编程模型（如EJB 2.1实体bean）通过引入容器托管的关系扰乱了这个行为——如果一侧被应用程序修改，容器就会自动改变关系的另一侧。这就是为什么使用EJB 2.1实体bean的代码不能在容器之外被重用的原因之一。EJB 3.0实体关联是透明的，就像在Hibernate中一样。如果你不理解Hibernate中关联的这种行为，就问问你自己："没有Hibernate时我会怎么做？"Hibernate不会改变一般的Java语义。

给这些操作进行分组的Category类添加一种方便的方法是个好主意，这样允许重用，帮助确保正确性，并且最终保证数据的完整性：

```
public void addChildCategory(Category childCategory) {
     if (childCategory == null)
      throw new IllegalArgumentException("Null child category!");
     if (childCategory.getParentCategory() != null)
      childCategory.getParentCategory().getChildCategories()
                                        .remove(childCategory);
    childCategory.setParentCategory(this);
    childCategories.add(childCategory);
}
```

addChildCategory()方法不仅在处理Category对象时减少了代码行，而且加强了关联的基数性（cardinality）。它避免了漏掉两个必需动作中的其中一个时产生的错误。如果可能的话，应该始终对关联提供这种操作的分组。如果你把它与关系数据库中外键的关系模型相比较，就很容易明白网络和指针模型如何使一个简单的操作变得复杂：不用声明约束，而是需要过程性代码（procedural code）来保证数据的完整性。

由于想要addChildCategory()成为子类中唯一外部可见的存储器方法（mutator method）（可能还要加上removeChildCategory()方法），你可以让setChildCategories()方法为私有，或者删除它，并使用直接的字段访问进行持久化。获取方法仍然返回一个可以修改的集合，因此客户端可以用它进行在另一侧不受影响的变化。应该考虑静态的方法Collection.unmodifiable-Collection(c)和Collections.unmodifiableSet(s)，如果你更喜欢在返回它们之前把内部集合包在获取方法里面的话。然后，如果客户端试图修改集合，它就会获得一个异常；每次修改都被强制通过关系管理的方法。

在Category和Item类之间存在着一种不同的关系：一个双向的多对多关联，如图3-4所示。

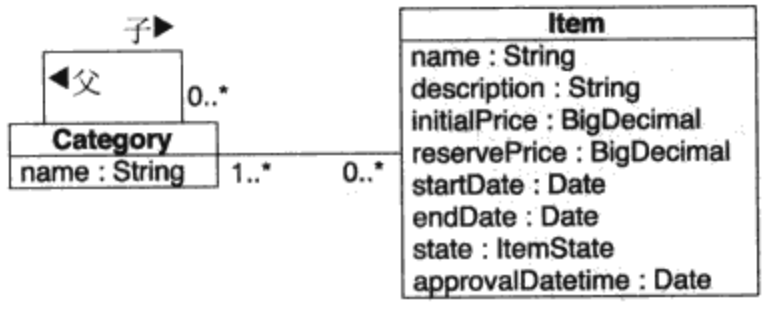

图3-4　Category和关联的Item类

对于多对多的关联，两侧都用集合值属性实现。添加访问Item关系的新属性和方法到Category类，如代码清单3-2所示。

代码清单3-2 Category到Item脚手架代码

```
public class Category {
    ...
    private Set items = new HashSet();
    ...

    public Set getItems() {
        return items;
    }

    public void setItems(Set items) {
        this.items = items;
    }
}
```

Item类的代码（多对多关联的另一侧）类似于Category类的代码。添加集合属性，标准的访问方法，以及简化关系管理的一种方法，如代码清单3-3所示。

代码清单3-3 Item到Category脚手架代码

```
public class Item {

    private String name;
    private String description;
    ...
    private Set categories = new HashSet();
    ...

    public Set getCategories() {
        return categories;
    }

    private void setCategories(Set categories) {
        this.categories = categories;
    }

    public void addCategory(Category category) {
        if (category == null)
            throw new IllegalArgumentException("Null category");
        category.getItems().add(this);
        categories.add(category);
    }
}
```

addCategory()方法类似于Category类的addChildCategory()便利方法。它被客户端用来操作Item和Category之间的链接。为了增加可读性，我们在后面的代码样例中将不显示便利方法了，并假设你会根据自己的偏好来添加它们。

给关联处理使用便利方法并不是改善领域模型实现的唯一途径。也可以把逻辑添加到你的访问方法中。

3.2.5　把逻辑添加到访问方法

我们喜欢使用JavaBean风格的访问方法的原因之一在于它们提供封装：一个属性的内部隐藏实现可以不做任何改变地变换为公共接口。这让你能够从数据库的设计中抽象类的内部数据结构——实例变量，如果Hibernate在运行时通过访问方法访问属性的话。它也促使公共API和类的内部表述的独立重构更加容易。

例如，如果数据库把用户名称保存为单个NAME列，但是User类有firstname和lastname属性，你可以把下列持久化的name属性添加到这个类：

```
public class User {
    private String firstname;
    private String lastname;
    ...

    public String getName() {
        return firstname + ' ' + lastname;
    }

    public void setName(String name) {
        StringTokenizer t = new StringTokenizer(name);
        firstname = t.nextToken();
        lastname = t.nextToken();
    }
    ....

}
```

稍后，你会明白Hibernate定制类型是处理多个这种情况的一种更好的方法。然而，有多种选择的话也自有好处。

访问方法也可以执行验证。例如，在下列例子中，setFirstName()方法验证大写的名称：

```
public class User {
    private String firstname;
    ...

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname)
        throws InvalidNameException {

        if ( !StringUtil.isCapitalizedName(firstname) )
            throw new InvalidNameException(firstname);
        this.firstname = firstname;
    }
    ....

}
```

当从数据库加载对象时，Hibernate可以使用访问方法来填充一个实例的状态，有时候你宁愿在Hibernate初始化新加载的对象时不发生这种验证。在这种情况下，告诉Hibernate直接访问实例变量就有意义了。

另一个要考虑的问题是脏检查。Hibernate自动侦测对象状态的改变，以便使更新过的状态与数据库同步。从获取方法返回一个不同的对象，通常比由Hibernate传递到设置方法的对象来得安全。Hibernate按值比较对象——不是按对象同一性——来确定一个属性的持久化状态是否需要被更新。例如，下列获取方法就不会导致不必要的SQL UPDATE（更新）：

```
public String getFirstname() {
    return new String(firstname);
}
```

这有个重要的例外：集合是按同一性比较的！对于一个被映射为持久化集合的属性，你应该从获取方法中返回与Hibernate传递到设置方法中完全相同的集合实例。如果没有，Hibernate将更新数据库，即使不需要更新，保存在内存中的状态每次也都会与数据库同步。在访问方法中通常应该避免这种代码：

```
public void setNames(List namesList) {
    names = (String[]) namesList.toArray();
}

public List getNames() {
    return Arrays.asList(names);
}
```

最后，必须知道访问方法中的异常如何处理，如果你在加载和存储实例时配置Hibernate来使用这些方法的话。如果抛出RuntimeException，当前的事务就被回滚，你要自己处理这个异常。如果抛出已检查应用异常，Hibernate就会把这个异常包在一个RuntimeException里面。

可以看到Hibernate没有用POJO编程模型对你进行不必要的限制。可以在访问方法中实现所需的任何逻辑（只要在获取方法和设置方法中保持相同的集合实例）。Hibernate如何访问属性是完全可以配置的。这种透明度保证了一个独立的和可重用的领域模型实现。并且目前为止我们所阐述的一切，对于Hibernate持久化类和JPA实体都是相当正确的。

现在给持久化类定义ORM。

3.3 ORM 元数据

ORM工具要求元数据指定类和表、属性和列、关联和外键、Java类型和SQL类型等之间的映射。这种信息称作ORM元数据。元数据是关于数据的数据，映射元数据定义和支配在面向对象和SQL系统中不同的类型系统和关系表示法之间的转化。

作为开发人员，编写和维护这个元数据是你的工作。本节将讨论各种方法，包括在XML文件和JDK 5.0源码注解中的元数据。通常在一个特定的项目中决定使用一种策略，读完这几节内容后，你将具有做出明智决定的背景信息。

3.3.1 XML 中的元数据

任何ORM解决方案都应该提供一个人类可读的、易于手工编辑的映射格式，而不只是一个GUI映射工具。目前，最受欢迎的对象/关系元数据格式是XML。以XML编写或者包含XML的映

射文档都是轻量级的、人类可读的、易于通过版本控制系统和文本编辑器操作的，并且它们可以在部署时被定制（或者甚至在运行时用编程式的XML生成）。

但是基于XML的元数据真的是最好的方法吗？在Java社区中可以看到相当一部分人还是反对过度使用XML的。每一个框架和应用程序服务器似乎都需要它自己的XML描述符。

在我们看来，这种反对有三大原因：

- 基于元数据的解决方案经常被不恰当地应用。元数据生来就没有简单的Java代码那么灵活或者易于维护。
- 许多现有的元数据格式不是被设计成易读和易于用手工编辑的。造成痛苦的一个主要原因尤其在于属性和元素值缺乏有意义的默认值，明显要求比应该需要的输入多得多。更糟糕的是，有些元数据模式只使用XML元素和文本值，没有任何属性。另一个问题是模式太过一般，其中的每个声明都被包在meta元素的普通的扩展属性中。
- 好的XML编辑器，特别是在IDE中，不像好的Java编码环境一样普遍。最糟糕且最容易确定的是，经常不提供文档类型声明（DTD），阻止了自动完成和验证。

在ORM中无法避免对元数据的需求。然而，Hibernate在设计时充分认识到了典型的元数据问题。Hibernate的XML元数据格式非常易读，并定义了有用的默认值。如果没有属性值，就在被映射的类上用反射来确定默认值。Hibernate还包含提供文档化的完整DTD。最终，IDE对XML的支持近来已经得到了改善，现代的IDE提供动态的XML验证，甚至提供一项自动完成的特性。

来看一下可以在Hibernate中使用XML元数据的方法。你在前一节中创建了`Category`类；现在要把它映射到数据库中的`CATEGORY`表。要这么做，得编写代码清单3-4中的XML映射文件。

代码清单3-4　Category类的Hibernate XML映射

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC                    ❶
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>                                   ❷
    <class                                            ❸
        name="auction.model.Category"
        table="CATEGORY">

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

        <property                                     ❺
            name="name"
            column="NAME"
            type="string"/>

    </class>
</hibernate-mapping>
```

❶ Hibernate映射DTD应该在每个映射文件中声明——XML的语法验证需要这么做。

❷ 映射在<hibernate-mapping>元素内部声明。你喜欢包括多少个类映射都可以，包括本书稍后会提到的其他特殊的声明。

❸ Category类（在auction.model包中）被映射到CATEGORY表。这张表中的每一行都表示类型Category的一个实例。

❹ 我们还没有讨论对象同一性的概念，因此你可能对这个映射元素感到好奇。这个复杂的话题将在下一章中讨论。要理解这个映射，了解CATEGORY表中的每一行都有一个与内存中实例的对象同一性相匹配的主键值就足够了。<id>映射元素被用来定义对象同一性的细节。

❺ 类型java.lang.String的属性名称被映射到一个数据库的NAME列。注意映射中声明的类型是一个内建的Hibernate类型（string），而不是Java属性的类型，或者SQL列类型。把它当作表示其他两个类型系统之间桥梁的一个转换器。

我们已经在这个例子中有意省略了集合和关联映射。关联尤其是集合映射更加复杂，因此本书的第二部分将再回到这个话题。

虽然可能通过使用多个<class>元素在一个映射文件中给多个类声明映射，但我们建议的实践（也是一些Hibernate工具所期待的实践）是给每个持久化类使用一个映射文件。习惯上给文件与被映射的类相同的名称，并附加一个后缀（例如Category.hbm.xml），并把它放在与Category类相同的包中。

就像已经提到过的，XML映射文件并不是在Hibernate应用程序中定义映射元数据的唯一方法。如果使用JDK 5.0，最好的选择是基于EJB 3.0和Java Persistence标准的Hibernate Annotations。

3.3.2 基于注解的元数据

基本思想是把元数据与它所描述的信息放在一起，而不是把它分离到一个不同的文件中去。Java在JDK 5.0之前并不具备这项功能，因此开发了一种可供选择的方案。XDoclet项目使用支持键/值对的特殊Javadoc标签，引入了包含元信息的Java源代码的注解。通过标签的嵌套，非常复杂的结构也得到了支持，但是只有一些IDE允许为自动完成和验证定制Javadoc模板。

Java规范请求（JSR）175在Java语言中引入了注解概念，给注解的定义使用类型安全和声明的接口。自动完成和编译时检查不再是问题。我们发现，与XDoclet相比，注解元数据并不冗长，并且它有更好的默认值。然而，JDK 5.0注解有时候比XDoclet注解更难于阅读，因为它们不在普通的注释块里面；你应该使用一个支持注解的可配置语法的突出显示的IDE。除此之外，我们在过去几年的日常工作中，并没有发现使用注解的严重缺点，并且我们认为注解-元数据支持将成为JDK 5.0最重要的特性之一。

现在我们要介绍映射注解并使用JDK 5.0。如果你必须使用JDK 1.4，但是又喜欢使用基于注解的元数据，那就考虑我们后面讨论的XDoclet。

1. 定义和使用注解

在注解第一个持久化类之前，我们来看看注解是如何创建的。自然地，你通常使用预设的注解。然而，知道如何扩展现有的元数据格式，或者知道如何编写你自己的注解是一项非常有用的

技能。下列代码示例显示了一个Entity注解的定义：

```
package javax.persistence;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}
```

第一行定义了包，像往常一样。这个注解在javax.persistence包中，它是EJB 3.0所定义的JPA。它是规范中最重要的注解之一——可以在POJO上应用它，使它成为一个持久化的实体类。接下来一行是一个注解，该注解把元信息添加到@Entity注解（有关元数据的元数据）。它指定@Entity注解只可以放在类型声明中；换句话说，只可以用@Entity注解标识类，而不标识字段或者方法。为这个注解所选择的保持策略是RUNTIME；其他方案（对于其他使用案例）包括编译期间移除注解元数据，或者只有字节码中包含的内容，没有可能的运行时反射。你甚至想要在运行时保存所有实体元信息，以便Hibernate可以在启动时通过Java Reflection读取。这个例子遵循的是注解的实际声明，包括它的接口名称和属性（在这个例子中只有一个属性name和一个空的字符串默认）。

使用这个注解来使POJO持久化类成为Java Persistence实体：

```
package auction.model;

import javax.persistence.*;

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

这个公共的类Item，已经被声明为持久化实体。它的所有属性现在都用默认策略自动持久化。还显示了另一个注解，它声明这个持久化类被映射到的那个数据库Schema中的表名。如果省略这个信息，JPA提供程序默认为未限定类名（unqualified class name），就像如果你在一个XML映射文件中省略表名时Hibernate所做的处理那样。

所有这些都是类型安全的，因此当Hibernate启动时，被声明的注解通过Java Reflection读取。你不需要编写任何XML映射文件，Hibernate无需解析任何XML，并且启动更快。IDE也可以轻松地验证和强调注解——毕竟它们是普通的Java类型。

注解明显的好处之一是它们对于敏捷开发的灵活性，如果你经常重构代码、重新命名、删除或者移动类和属性的话。大部分开发工具和编辑器都无法重构XML元素和属性值，但是注解是Java语言的一部分，包括在所有重构操作中。

应该应用哪些注解呢？有几个标准的且特定于供应商的包可供你选择。

2．考虑标准

基于注解的元数据对于你如何编写Java应用程序有着重大的影响。其他编程环境，比如C#和.NET，早就具备这种支持，开发人员很快就采用了元数据属性。在Java世界里，注解的广泛应

用是在Java EE 5.0中。所有被认为是Java EE一部分的规范，如EJB、JMS、JMX甚至servlet规范，都将被更新，并因为元数据需要而使用JDK 5.0注解。例如，J2EE 1.4中的Web服务通常需要XML文件中的重要元数据，因此我们期待看到通过注解真正提高生产力。或者，你可以通过在一个字段上添加注解，让Web容器把一个EJB句柄注入到servlet中。Sun公司引进了一项规范成果（JSR 250）来处理不同规范的注解，给整个Java平台定义一般的注解。然而，对于你在持久层上的工作，最重要的规范则是EJB 3.0和JPA。

一旦你已经在classpath中包括了JPA接口，来自Java Persistence包的注解在javax.persistence中就可以使用了。可以用这些注解声明持久化的实体类、可嵌入的类（下一章将讨论这些）、属性、字段、键等。JPA规范覆盖了基础的和最相关的高级映射——编写一个可移植的应用程序所需要的一切，包含一个可插拔的、标准的持久化层，在任何运行时容器的内部和外部都适用。

Java Persistence中没有指定哪些注解和映射特性？特定的JPA引擎和产品一般可以提供这些优势——所谓的供应商扩展。

3. 利用供应商扩展

即使你用javax.persistence包中JPA兼容的注解映射了应用程序的大部分模型，有时候还是必须使用供应商扩展。例如，你希望在高质量的持久化软件中可用的几乎所有的性能调优选项，例如抓取和高速缓存设置，都只作为特定于Hibernate的注解。

我们在一个例子中看看这会是什么样。再次注解Item实体源代码：

```
package auction.model;

import javax.persistence.*;

@Entity
@Table(name = "ITEM")
@org.hibernate.annotations.BatchSize(size = 10)
@org.hibernate.annotations.DiscriminatorFormula(
    "case when ITEM_IS_SPECIAL is not null then A else B end"
)
public class Item {
    ...
}
```

这个例子包含了两个Hibernate注解。第一个注解@BatchSize是一个抓取选项，可以在本书稍后要验证的几种情况下提升性能。第二个注解@DiscriminatorFormula是一个Hibernate映射注解，当类继承无法用简单的文字值（literal value）决定的时候（此处它映射了一个遗留的列ITEM_IS_SPECIAL——或许某种标记——到一个文字值），对于遗留的模式特别有用。这两个注解都用org.hibernate.annotations作为包名的前缀。把这当作一个好的实践，因为你现在可以很容易地看到这个实体的什么元数据是来自JPA规范，以及哪些标签是特定于供应商的。你还可以轻松地搜索源代码里的“org.hibernate.annotations”，并在单个搜索结果中得到你应用程序中所有非标准注解的一个全面概览。

如果转换Java Persistence的提供程序，你只需要替换特定于供应商的扩展，并且可以期待最复杂的解决方案中会有一个类似的特性设置。当然，我们希望你永远不要这么做，并且它在实际

应用中很少发生——只是备用着。

类中的注解仅仅涵盖适用于该特定类的元数据。然而，在更高的级别上经常需要元数据，用于整个包，甚至整个应用程序。在讨论这些方法之前，要介绍另一种映射元数据格式。

4．JPA和EJB 3.0中的XML描述符

EJB 3.0和Java Persistence标准都争先包含了注解。然而，专家组已经注意到了XML部署描述符在某些情况下的优势，特别对于随着每次部署而改变的配置元数据。结果，EJB 3.0和JPA中的每一个注解都可以用一个XML描述符元素替换。换句话说，如果你不想用，就不必使用注解（虽然我们强烈鼓励你重新考虑，并试试注解，如果这是你对注解的第一反应的话）。

来看一下特定持久化单元的JPA XML描述符示例：

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

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/001/MLSchema-instance"
  xsi:schemaLocation=
      "http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

    <persistence-unit-metadata>
        <xml-mapping-metadata-complete/>
        <persistence-unit-defaults>
            <schema>MY_SCHEMA</schema>
            <catalog>MY_CATALOG</catalog>
            <cascade-persist/>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <package>auction.model</package>

    <entity class="Item" access="PROPERTY"
            metadata-complete="true">
        <attributes>
            <id name="id">
                <generated-value strategy="AUTO"/>
            </id>
        </attributes>
    </entity>

</entity-mappings>
```

这个XML由JPA提供程序自动挑选，如果把它放在classpath中持久化单元的META-INF目录下的一个名为orm.xml的文件中的话。你会看到只需要给一个类命名一个标识符属性；就像在注解中一样，实体类的所有其他属性也通过一个明显的默认映射自动地被认为是持久化的。

也可以给整个持久化单元设置默认映射，例如Schema名称和默认的级联选项。如果包括<xml-mapping-metadata-complete>元素，JPA提供程序则完全忽略这个持久化单元中实体类的全部注解，并且仅仅依赖如同在orm.xml文件中定义的映射。你可以（在这个例子中是多余的）在一个实体级别上用metadata-complete="ture"启用它。一旦启用，JPA提供程序假设实体的所有属性都以XML格式映射，因此这个实体的所有注解都应该被忽略。

如果不想忽略而是要覆盖注解元数据，就先从orm.xml文件中移除全局的<xml-mapping-metadata-complete>元素。还要从任何应该覆盖（而不是取代）注解的实体映射中移除metadata-complete="true"属性：

```
<entity-mappings ...>

    <package>auction.model</package>

    <entity class="Item">
        <attributes>
            <basic name="initialPrice" optional="false">
                <column name="INIT_PRICE"/>
            </basic>
        </attributes>
    </entity>

</entity-mappings>
```

此处把initialPrice属性映射到了INIT_PRICE列，并指定它不可为空。Item类的initialPrice属性上的任何注解都被忽略，但Item类中的所有其他注解仍然得到应用。还要注意你并没有在这个映射中指定一种访问策略，因此字段或者访问方法访问根据Item中@Id注解的位置来使用。（第4章会讨论这个细节。）

在Java Persistence中使用XML部署描述符的一个明显的问题是，它们与原生的Hibernate XML映射文件的兼容性。这两种格式根本不兼容，你应该决定使用一种或者另一种。JPA XML描述符的语法比原生的Hibernate XML映射文件更接近于实际的JPA注解。

当决定一种XML元数据格式时，还要考虑供应商扩展。Hibernate XML格式支持所有可能的Hibernate映射，因此如果有些东西无法在JPA/Hibernate注解中被映射，则可以用原生的Hibernate XML文件映射。同样的事情对于JPA XML描述符则不是如此——它们只提供覆盖规范的方便且具体化的元数据。Sun公司不允许供应商扩展使用另外的命名空间。

另一方面，你无法用Hibernate XML映射文件覆盖注解；必须用XML格式定义一个完整的实体类映射。

基于这些原因，我们不介绍所有三种格式的所有可能的映射；我们把焦点放在原生的Hibernate XML元数据和JPA/Hibernate注解上。然而，如果你想学，将会学到有关JPA XML描述符的足够知识来使用它。

如果正使用JDK 5.0，就考虑JPA/Hibernate注解为首选。如果想把一个特定的类映射具体化，或者利用一个不能作为注解使用的Hibernate扩展，就回到原生的Hibernate XML映射文件。如果不打算使用任何供应商扩展（实际上，这是不可能的），或者如果只想覆盖几个注解，或者如果需要甚至包括部署描述符的完整可移植性，就考虑用JPA XML描述符。

但是如果你习惯使用JDK 1.4（或者甚至1.3），却又仍然想要享受行内元数据的更好重构能力和减少代码行的好处时，该怎么办？

3.3.3 使用 XDoclet

XDoclet项目给Java带来了面向属性的编程概念。XDoclet利用Javadoc标签格式（@attribute）

来指定类、字段，或者方法级的元数据属性。Manning出版社还有一本关于XDoclet的书*XDoclet in Action*（Walls和Richards，2004）。

XDoclet被实现为生成Hibernate XML元数据（或者其他东西，取决于插件）的一个Ant目标，作为构建过程的一部分。用XDoclet创建Hibernate XML映射文档很简单；不用手工编写，而是用定制的Javadoc标签给持久化类的Java源代码做上标记，如代码清单3-5所示。

代码清单3-5　用XDoclet标签给包含映射元数据的Java类做标记

```
/**
 * The Category class of the CaveatEmptor auction site domain model.
 *
 * @hibernate.class
 *  table="CATEGORY"
 */
public class Category {
    ...

    /**
     * @hibernate.id
     *  generator-class="native"
     *  column="CATEGORY_ID"
     */
    public Long getId() {
        return id;
    }

    ...

    /**
     * @hibernate.property
     */
    public String getName() {
        return name;
    }

    ...
}
```

当被注解的类到位，并且准备好了Ant任务时，你就可以自动生成和前一节（见代码清单3-4）所示一样的XML文档了。

XDoclet的缺点在于需要另一个构建步骤。大部分大Java项目已经使用Ant，因此这通常不成问题。有争议的是，XDoclet映射在部署时更不易配置；但是没有东西阻止你在部署之前用手工编辑生成的XML，因此这可能不是重要的拒绝理由。最终，对XDoclet标签验证的支持在你的开发环境中可能不可用。然而，最新的IDE至少支持标签名称的自动完成。本书不涵盖XDoclet的内容，但你可以在Hibernate网站上找到示例。

无论使用XML文件、JDK 5.0注解，还是XDoclet，都将经常发现你必须在几个地方重复元数据。换句话说，你需要添加不只一个属性、不只一个持久化类、或者甚至整个应用程序都可以应用的全局信息。

3.3.4 处理全局的元数据

考虑以下情形：你所有的领域模型持久化类都在同一个包中。然而，你必须在每个XML映射文件中指定完全限定（fully qualified）的类名，包括包名。声明包名一次，然后仅使用短的持久化类名，这样会容易得多。或者，不要通过access="field"映射属性给每个单独的属性启用直接的字段访问，而使用单个的转换为所有属性启用字段访问。类或者包范围的元数据会更方便。

有些元数据对整个应用程序有效。例如，查询字符串可以被具体化到元数据，并且在应用程序代码中通过一个全局唯一的名称调用。类似地，查询通常与特定的类无关，有时，甚至与特定的包无关。其他应用程序范围的元数据包括用户自定义的映射类型（转换器）和数据过滤器（动态视图）定义。

来快速看一些在Hibernate XML映射和JDK 5.0注解中全局元数据的例子。

1. 全局的XML映射元数据

如果检查XML映射DTD，会看到<hibernate-mapping>根元素有全局的选项，被应用到它内部的（多个）类映射——这些选项中有一些如下列例子所示：

```
<hibernate-mapping
    schema="AUCTION"
    default-lazy="false"
    default-access="field"
    auto-import="false">

<class ...>
    ...
</class>

</hibernate-mapping>
```

schema属性使得一个数据库Schema前缀AUCTION能够被Hibernate用给已映射类所生成的所有SQL声明。通过设置default-lazy为false，给有些类关联启用默认的外联结抓取，在13.1节中有这个主题。（这个default-lazy="true"切换有一个值得关注的副作用：它切换到了Hibernate 2.x默认的抓取行为——如果你移到Hibernate 3.x但不想更新所有抓取设置时可以使用它。）利用default-access，通过Hibernate为这个文件中所有被映射的类的所有持久化属性启用直接的字段访问。最后，auto-import设置对这个文件中的所有类关闭。4.3节将讨论实体的引入和命名。

提示 没有类声明的映射文件——任何复杂的应用都需要和呈现全局的元数据。例如，可以轻松地导入许多接口，或者具体化几百个查询字符串。在大型的应用程序中，通常创建没有实际类映射的映射文件，并且只有导入、外部查询，或者全局过滤器和类型定义。如果看看DTD，就会看到<class>映射在<hibernate-mapping>根元素内部是可选的。把全局的元数据分开，并把它组织到单独的文件中，例如AutionTypes.hbm.xml、AuctionQueries.hbm.xml等，并在Hibernate配置中加载它们，就像一般的映射文件一样。然而，确保在把所有定制的类型和过滤器应用到类映射的任何其他映射元数据之前，就已经加载了这些类型和过滤器。

来看看包含JDK 5.0注解的全局的元数据。

2. 全局的注解元数据

注解天生就被织入一个特定类的Java源代码中。虽然可能把全局的注解放在类的源文件中（在顶部），我们还是喜欢把全局的元数据放在一个单独的文件中。这称作包元数据（package metadata），它被特定包目录中具名package-info.java的文件所启用：

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_usd",
        typeClass = MonetaryAmountType.class,
        parameters = { @Parameter(name="convertTo", value="USD") }
    ),
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_eur",
        typeClass = MonetaryAmountType.class,
        parameters = { @Parameter(name="convertTo", value="EUR") }
    )
})
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByPrice",
        query = "select i from Item i order by i.initialPrice)"
    )
})
package auction.persistence.types;
```

包元数据文件的这个例子，在包auction.persistence.types中，它声明了两个Hibernate类型转换器。5.2节将讨论Hibernate类型系统。现在你可以通过它们的名称在类映射中引用用户自定义的类型。可以用同样的机制来具体化查询和定义全局的标识符生成器（最后这个例子中没有显示）。

这就是前一个代码示例中只包括来自Hibernate包的注解，而没有Java Persistence注解的原因之一。JPA规范（在最后一刻）所做的一个改变是移除了JPA注解的包可见性。结果，没有任何Java Persistence注解可以被放在package-info.java文件中。如果需要可移植的全局的Java Persistence元数据，就把它放在orm.xml文件中。

注意，如果没有使用自动侦测，就必须在Hibernate或者JPA持久化单元配置中指定一个包含元数据文件的包——详情请见2.2.1节。

全局的注解（Hibernate和JPA）也可以被放在特定类的源代码中，就在import部分之后。注解的语法与package-info.java文件中的相同，因此这里不再重复。

现在你知道了如何编写本地的和全局的映射元数据。大型应用程序中的另一个问题是元数据的可移植性。

3. 使用占位符

在任何较大的Hibernate应用程序中，都要在映射元数据中面对本地代码的问题——有效地把映射绑定到一个特定数据库产品的代码。例如，公式、约束或者过滤器映射中的SQL语句，都不

被Hibernate解析，而是直接传递到数据库管理系统。这样的好处是灵活性——可以调用任何原生的SQL函数或者数据库系统支持的关键字。把原生的SQL放在映射元数据中的缺点是损失数据库的可移植性，因为映射及你的应用程序将只适用于特定的DBMS（或者甚至DBMS版本）。

甚至简单的东西，例如主键生成策略，通常不能跨越所有的数据库系统进行移植。第4章将讨论一种特殊的标识符生成器，叫做native，这是个内建的巧妙的主键生成器。在Oracle中，它用一个数据库序列在表中给行生成主键值；在IBM DB2中，它默认使用一个特殊的同一性主键列。以下是如何用XML映射它的方法：

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

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

    ...
</class>
```

稍后将讨论这个映射的细节。值得关注的部分是声明class="native"作为标识符生成器。假设这个生成器提供的可移植性不是你所要的，可能因为你用了一个定制的标识符生成器，你编写的一个类实现了Hibernate的IdentifierGenerator接口：

```
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="auction.custom.MyOracleGenerator"/>
</id>
```

XML映射文件现在被绑定到了一个特定的数据库产品，你损失了Hibernate应用程序的数据库可移植性。处理这个问题的一种方法是在你的XML文件中使用一个占位符，在映射文件被复制到目标目录下的构建期间，这个XML文件被替换（Ant支持这个）。只有当你有使用Ant的经验，或者应用程序的其他部分已经需要构建时的替换时，才建议使用这种机制。

一种更优雅的变更方法是使用定制的XML实体（与应用的业务实体无关）。假设你要在XML文件中具体化一个元素或者属性值，来使它保持可移植：

```
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="&idgenerator;"/>
</id>
```

&idgenerator;值称作实体占位符。可以在XML文件的顶部定义它的值为实体声明，作为文档类型定义的一部分：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY idgenerator    "auction.custom.MyOracleGenerator">
]>
```

当映射文件被读取的时候，XML解析器现在将在Hibernate启动时替换占位符。

可以把这个更推进一步，在单个的文件中把这个新增物具体化到DTD，并在所有其他的映射文件中包括全局的选项：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY % globals SYSTEM "classpath://persistence/globals.dtd">
%globals;
]>
```

这个例子显示了作为DTD一部分的一个外部文件所包含的内容。所用的语法，就像经常采用XML时一样，相当粗糙，但是每一行的用途应该很清楚。所有全局的设置都被添加到classpath中persistence包里的globals.dtd文件中：

```
<!ENTITY idgenerator    "auction.custom.MyOracleGenerator">
<!-- Add more options if needed... -->
```

要从Oracle切换到一个不同的数据库系统，只要部署一个不同的globals.dtd文件。

你经常不仅要替换一个XML元素或者属性值，还要在所有的文件中包括映射元数据的整个块，例如当许多类共享一些常用的属性，并且你无法在单个位置使用继承来捕捉它们的时候。利用XML实体替换，可以把一个XML片段具体化到一个单独的文件，并把它包括在其他XML文件中。

假设所有的持久化类都有一个dateModified属性。第一步是把这个映射放在它自己的文件中，比如DateModified.hbm.xml：

```
<property name="dateModified"
          column="DATE_MOD"
          type="timestamp"/>
```

这个文件不需要XML首部或者任何其他标签。现在为一个持久化类把它包括在映射文件中：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY datemodified   SYSTEM "classpath://model/DateModified.hbm.xml">
]>

<hibernate-mapping>

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

    &datemodified;

    ...
</class>
```

DateModified.hbm.xml的内容将被&datemodified;占位符包括和替换。当然，这也适用于较大的XML片段。

当Hibernate启动和读取映射文件时，XML DTD必须由XML解析器解析。内建的Hibernate实体解析器在classpath中查找Hibernate-mapping-3.0.dtd；它应该在试图从因特网上查找之前先在hibernate3.jar文件中查找DTD，每当一个实体URL被加上 http://hibernate.sourceforge.net/前缀时这都会自动发生。Hibernate实体解析器也可以侦测classpath://前缀，然后在classpath中搜索资

源，部署时可以从中复制资源。必须重复这个常见的问题：如果在你的映射中有正确的DTD引用，并且在classpath中有正确的JAR时，Hibernate就永远不会在因特网上查找DTD。

目前为止我们已经阐述过的方法——XML、JDK 5.0注解和XDoclet属性——假设所有的映射信息在开发（或者部署）时都是已知的。然而，假设有些信息在应用程序启动前未知。能够在运行时通过编程操作映射元数据吗？

3.3.5 运行时操作元数据

在运行时浏览、操作或者构建新映射，对于应用程序来说有时候有用。XML API（如DOM、dom4j和JDOM）都允许XML文档的直接运行时操作，因此你能够在把XML文档传递给Configuration对象之前，在运行时对它进行创建或者操作。

另一方面，Hibernate也公开了一个配置时的元模型，它包含了在静态映射元数据中声明的所有信息。这个元模型直接的编程式操作有时候有用，尤其对于允许通过用户编写的代码进行扩展的应用程序。一种更强大的方法是映射元数据完全纲领性的和动态的定义，没有任何静态的映射。但这很怪异，应该为完全动态的应用程序的一个特定类或者应用程序构建工具包所保留。

下列代码将一个新的属性motto添加到了User类：

```
// Get the existing mapping for User from Configuration
PersistentClass userMapping =
    cfg.getClassMapping(User.class.getName());

// Define a new column for the USER table
Column column = new Column();
column.setName("MOTTO");
column.setNullable(false);
column.setUnique(true);
userMapping.getTable().addColumn(column);

// Wrap the column in a Value
SimpleValue value = new SimpleValue();
value.setTable( userMapping.getTable() );
value.setTypeName("string");
value.addColumn(column);

// Define a new property of the User class
Property prop = new Property();
prop.setValue(value);
prop.setName("motto");
prop.setNodeName(prop.getName());
userMapping.addProperty(prop);

// Build a new session factory, using the new mapping
SessionFactory sf = cfg.buildSessionFactory();
```

PersistentClass对象表示了单个持久化类的元模型，你从Configuration对象中获取它。Column、SimpleValue和Property是Hibernate元模型的所有类，可以在org.hibernate.mapping包中使用它们。

提示　记住，把一种属性添加到一个现有的持久化类映射，如此处所示，相当容易，但是给之前未被映射的类编程式地创建一个新的映射就比较麻烦了。

一旦创建了SessionFactory，它的映射就是不可变的。SessionFactory内部使用一种与配置时所用的不同的元模型。没有办法从SessionFactory或者Session中退回到原始的Configuration。（注意，你可以从Session中获得SessionFactory，如果希望访问一个全局设置的话。）然而，应用程序可以通过调用getClassMetadata()或者getCollectionMetadata()来读取SessionFactory的元模型。这里有个例子：

```
Item item = ...;
ClassMetadata meta = sessionFactory.getClassMetadata(Item.class);
String[] metaPropertyNames =
            meta.getPropertyNames();
Object[] propertyValues =
            meta.getPropertyValues(item, EntityMode.POJO);
```

这个代码片段为一个特定的实例获取Item类的持久化属性名称和那些属性的值。它帮助你编写一般的代码。例如，可以使用这个特性给UI组件贴标签，或者改进日志的输出。

虽然你已经在前几节中见过一些映射的构造，但目前为止我们尚未介绍任何更复杂的类和属性映射。你现在应该决定要在项目中使用哪种映射元数据选项，然后在下一章学习更多有关类和属性映射的内容。

或者，如果你已经是一个经验丰富的Hibernate用户，可以阅读最新的Hibernate版本，看看它允许你如何表示一个没有Java类的领域模型。

3.4　其他实体表示法

本书目前为止讨论的始终是基于Java类的领域模型实现——我们称它们为POJO、持久化类、JavaBean或者实体。基于包含一般属性、集合等的Java类的领域模型实现，是类型安全的。如果访问类的一个属性，IDE就提供基于模型的强类型的自动完成，且编译器检查你的源代码是否正确。然而，你在领域模型实现中对这种安全性花费了更多的时间——而时间就是金钱。

在接下来的几节中，我们介绍Hibernate处理不通过Java类实现的领域模型的能力。我们基本上用类型安全换取了其他的好处，因为天下没有免费的午餐，每当犯下一个错误时，运行时就会有更多的错误。在Hibernate中，可以给应用程序选择一种实体模式（entity mode），或者甚至混合多种实体模式得到一个的模型。甚至可以在单个Session中的实体模式之间进行切换。

Hibernate中有3种内建的实体模式：

- POJO——基于POJO、持久化类的一种领域模型实现。这是目前为止你所见到的，是默认的实体模式。
- MAP——不需要Java类；用HashMap在Java应用程序中表示实体。这个模式允许完全动态应用程序程序的快速原型。
- DOM4J——不需要Java类；实体被表示为XML元素，基于dom4j API。这种模式对于导出

或者导入数据，或者通过XSLT处理来渲染和转换数据时特别有用。

为什么可能要跳过下一节，稍后再回来呢？有两个原因：第一，包含POJO的静态领域模型实现是一个常见的案例，而动态的或者XML表示法可能不是你立即需要的特性。第二，我们正要展现一些映射、查询和其他可能你目前为止还没有见过的操作，即使利用默认的POJO实体模式也没有见过。然而，如果对使用Hibernate有足够的信心，就继续往下读。

我们从MAP模式开始，探讨Hibernate应用程序如何被完全动态地输入。

3.4.1 创建动态的应用程序

动态的领域模型是一个被动态地输入的模型。例如，不是用表示一件拍卖货品的Java类，而是在一个Java Map中使用许多值。拍卖货品的每个属性都由一个键（属性名称）及其值来表示。

1. 映射实体名称

首先，要通过命名业务实体启用这个策略。在一个Hibernate XML映射文件中，使用entity-name属性：

```
<hibernate-mapping>

<class entity-name="ItemEntity" table="ITEM_ENTITY">
    <id name="id" type="long" column="ITEM_ID">
        <generator class="native"/>
    </id>

    <property name="initialPrice"
              type="big_decimal"
              column="INIT_PRICE"/>

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

    <many-to-one name="seller"
                 entity-name="UserEntity"
                 column="USER_ID"/>
</class>

<class entity-name="UserEntity" table="USER_ENTITY">
    <id name="id" type="long" column="USER_ID">
        <generator class="native"/>
    </id>

    <property name="username"
              type="string"
              column="USERNAME"/>

    <bag name="itemsForSale" inverse="true" cascade="all">
        <key column="USER_ID"/>
        <one-to-many entity-name="ItemEntity"/>
    </bag>

</class>

</hibernate-mapping>
```

在这个映射文件中有三个值得关注的东西要看。

第一，把几个类映射混合到一个里面了，这是之前我们不建议的做法。这一次，并没有真正映射Java类，而是映射实体的逻辑名称。你没有把Java源文件和同名的XML映射文件放在一起，因此可以用你喜欢的任何方式组织元数据。

第二，<class name="...">属性已经被<class entity-name="..."取代。为了清楚起见，也附加了...Entity给这些逻辑名称，并把它们与你之前用一般的POJO创建的其他非动态映射区别开来。

第三，所有的实体关联，例如<many-to-one>和<one-to-many>，现在也引用逻辑的实体名称。关联映射中的class属性现在是entity-name。这并非严格需要——Hibernate会知道你正在引用一个逻辑的实体名称，即使你使用class属性。然而，当你稍后混合几种表示法时它可以避免混淆。

来看看使用动态的实体看起来是什么样子。

2. 使用动态的映射

为了创建其中一个实体的实例，要在Java Map中设置所有的属性值：

```
Map user = new HashMap();
user.put("username", "johndoe");

Map item1 = new HashMap();
item1.put("description", "An item for auction");
item1.put("initialPrice", new BigDecimal(99));
item1.put("seller", user);
Map item2 = new HashMap();
item2.put("description", "Another item for auction");
item2.put("initialPrice", new BigDecimal(123));
item2.put("seller", user);

Collection itemsForSale = new ArrayList();
itemsForSale.add(item1);
itemsForSale.add(item2);
user.put("itemsForSale", itemsForSale);

session.save("UserEntity", user);
```

第一个映射是UserEntity，你设置username属性为一个键/值对。接下来的两个映射都是ItemEntity，这里你通过把user映射放进item1和item2映射里面，设置了到每件货品的seller的链接。你正有效地链接映射——这就是为什么这种表示法策略有时也被称作“带有映射的映射的表示法。”

一对多关联反面的集合用一个ArrayList初始化，因为你用包（bag）[①]语义映射了它（Java没有包实现，但是Collection接口有包语义）。最后，Session中的save()方法被给定了一个逻辑的实体名称，其中user映射作为一个输入参数。

Hibernate知道UserEntity引用被动态映射的实体，并知道它应该把这个输入当作一个必须

① 即允许重复元素的无序集合。——译者注

被相应保存的映射来处理。Hibernate也级联到itemsForSale集合中的所有元素；因此，所有货品的映射也都成了持久化的。一个UserEntity和两个ItemEntity被插入到它们各自的表中。

常见问题 可以在动态模式中映射一个Set（集）吗？基于set的集合不适用于动态的实体模式。在前一个代码示例中，想象itemsForSale是一个Set。Set在它的元素中检查重复元素，因此当你调用add(item1)和add(item2)时，这些对象中的equals()方法也被调用。然而，item1和item2是Java Map实例，一个映射的equals()实现基于映射的键组。因此，由于item1和item2是带有相同键的映射，因此当它们被添加给一个Set时，并没有明显的区别。只有当你在动态的实体模式中需要集合时才使用bag或者list。

Hibernate处理映射就像处理POJO实例一样。例如，使一个映射持久化触发了标识符分配；处于持久化状态的每个映射都有一个包含生成值的标识符属性集。此外，持久化的映射被自动检查在一个工作单元内部的任何修改。例如，要在一件货品上设置新的价格，可以加载它，然后让Hibernate完成所有的工作：

```
Long storedItemId = (Long) item1.get("id");

Session session = getSessionFactory().openSession();
session.beginTransaction();

Map loadedItemMap = (Map) session.load("ItemEntity", storedItemId);
loadedItemMap.put("initialPrice", new BigDecimal(100));

session.getTransaction().commit();
session.close();
```

具有类参数如load()的所有Session方法也以一种接受多个实体名称的重载变体出现。在加载了一件货品的映射之后，设置新的价格，并通过提交事务使得所做的修改持久化，默认情况下，它会触发Session的脏检查和清除。

也可以在HQL查询中引用实体名称：

```
List queriedItemMaps =
    session.createQuery("from ItemEntity where initialPrice >= :p")
            .setParameter("p", new BigDecimal(100))
            .list();
```

这个查询返回ItemEntity映射的集合。它们处于持久化状态。

把这更推进一步，把POJO模型与动态的Map混合。为什么要把领域模型的静态实现与动态的映射表示法混合起来，有两个原因：

- 你要默认使用基于POJO类的静态模型，但有时候要把数据轻松地表示为映射的映射。这在生成报表时，或者必须实现一个可以动态地表示不同实体的普通用户界面时，特别有用。
- 要把模型的单个POJO类映射到几张表，然后通过指定一个逻辑的实体名称在运行时选择表。

可以找到混合实体模式的使用案例，但这种案例非常罕见，因此我们要把精力集中在更常见的案例上。

因此，首先我们将混合静态的POJO模型，有时还要对有些实体启用动态的映射表示法。

3. 混合动态的和静态的实体模式

要启用混合的模型表示法，得编辑XML映射元数据，并声明POJO类名和逻辑的实体名称：

```
<hibernate-mapping>

<class name="model.ItemPojo"
       entity-name="ItemEntity"
       table="ITEM_ENTITY">
    ...
    <many-to-one name="seller"
                 entity-name="UserEntity"
                 column="USER_ID"/>

</class>

<class name="model.UserPojo"
       entity-name="UserEntity"
       table="USER_ENTITY">
    ...
    <bag name="itemsForSale" inverse="true" cascade="all">
        <key column="USER_ID"/>
        <one-to-many entity-name="ItemEntity"/>
    </bag>
</class>

</hibernate-mapping>
```

显然，你还需要两个类model.ItemPojo和model.UserPojo，它们实现了这些实体的属性。两个实体之间的多对一和一对多关联仍然基于逻辑名称。

从现在开始Hibernate将主要使用逻辑名称。例如，下列代码就不起作用：

```
UserPojo user = new UserPojo();
...
ItemPojo item1 = new ItemPojo();
...
ItemPojo item2 = new ItemPojo();
...
Collection itemsForSale = new ArrayList();
...

session.save(user);
```

前面的例子创建了几个对象，设置了它们的属性，并链接它们，然后通过把user实例传递给save()，试图通过级联保存对象。Hibernate检查这个对象的类型，试图查出它是什么实体，并且因为Hibernate现在专门依赖逻辑的实体名称，因此它无法给model.UserPojo找到映射。当使用一个混合的表示法映射时，你要告诉Hibernate逻辑名称：

```
...
session.save("UserEntity", user);
```

一旦你改了这一行，前一个代码示例就起作用了。接下来，考虑加载，以及查询返回什么。默认情况下，特定的SessionFactory处于POJO实体模式，因此下列操作返回model.ItemPojo的实例：

```
Long storedItemId = item1.getId();
ItemPojo loadedItemPojo =
    (ItemPojo) session.load("ItemEntity", storedItemId);

List queriedItemPojos =
    session.createQuery("from ItemEntity where initialPrice >= :p")
            .setParameter("p", new BigDecimal(100))
            .list();
```

可以全局地或者临时地切换到一个动态的映射表示法，但是实体模式的全局转换会带来严重的后果。要全局地转换，得把下列代码添加到Hibernate配置中；例如，在hibernate.cfg.xml中：

```
<property name="default_entity_mode">dynamic-map</property>
```

现在所有的Session操作都期待或者返回动态输入的映射！存储、加载和查询POJO实例的前一个代码实例不再有效；你要存储和加载映射。

你更有可能想要临时切换到另一种实体模式，因此假设你把SessionFactory置于默认的POJO模式中。要在一个特定的Session中切换到动态映射，可以在现有会话的上面打开新的临时Session。下列代码使用了这样的一个临时的Session来为一位现有的卖家保存一件新的拍卖货品：

```
Session dynamicSession = session.getSession(EntityMode.MAP);

Map seller = (Map) dynamicSession.load("UserEntity", user.getId() );

Map newItemMap = new HashMap();
newItemMap.put("description", "An item for auction");
newItemMap.put("initialPrice", new BigDecimal(99));
newItemMap.put("seller", seller);

dynamicSession.save("ItemEntity", newItemMap);

Long storedItemId = (Long) newItemMap.get("id");

Map loadedItemMap =
    (Map) dynamicSession.load("ItemEntity", storedItemId);

List queriedItemMaps =
    dynamicSession
        .createQuery("from ItemEntity where initialPrice >= :p")
         .setParameter("p", new BigDecimal(100))
         .list();
```

用getSession()打开的临时dynamicSession不需要被清除或关闭；它继承原始Session的上下文。你只用它在选择的表示法中加载、查询或者保存数据，即前一个例子中的EntityMode.Map。注意，你不能用POJO实例链接一个映射；seller引用的必须是一个HashMap，而不是UserPojo的一个实例。

我们说过，逻辑实体名称的另一个好的用例是把一个POJO映射到几张表，因此我们来看看这个。

4．映射一个类多次

想象你有一些表，它们有一些共同的列。例如，你可能有ITEM_AUCTION和ITEM_SALE表。通常映射每张表给一个实体持久化类，分别为ItemAuction和ItemSale。在实体名称的帮助下，你可以保存工作，并实现单个持久化类。

为了把两张表都映射给单个持久化类，要使用不同的实体名称（通常是不同的属性映射）：

```
<hibernate-mapping>

<class name="model.Item"
       entity-name="ItemAuction"
       table="ITEM_AUCTION">

   <id name="id" column="ITEM_AUCTION_ID">...</id>
   <property name="description" column="DESCRIPTION"/>
   <property name="initialPrice" column="INIT_PRICE"/>

</class>

<class name="model.Item"
       entity-name="ItemSale"
       table="ITEM_SALE">

   <id name="id" column="ITEM_SALE_ID">...</id>
   <property name="description" column="DESCRIPTION"/>
   <property name="salesPrice" column="SALES_PRICE"/>

</class>

</hibernate-mapping>
```

model.Item持久化类有着你映射过的所有属性：id、description、initialPrice和salesPrice。根据在运行时使用的实体名称，有些属性被认为是持久化的，其他则为瞬时的：

```
Item itemForAuction = new Item();
itemForAuction.setDescription("An item for auction");
itemForAuction.setInitialPrice( new BigDecimal(99) );
session.save("ItemAuction", itemForAuction);

Item itemForSale = new Item();
itemForSale.setDescription("An item for sale");
itemForSale.setSalesPrice( new BigDecimal(123) );
session.save("ItemSale", itemForSale);
```

由于逻辑实体名称，Hibernate知道它应该把数据插入到哪张表中。根据用于加载和查询实体的实体名称，Hiberante从适当的表中选择。

需要这项功能的场合很少，你可能和我们一样认为前一个使用案例并不好或者不普遍。

下一节介绍第三种内建的Hibernate实体模式，领域实体用XML文档表示。

3.4.2　表示 XML 中的数据

XML只是一种文本文件格式；它没有成为数据存储或者数据管理的媒介的继承能力。XML

数据模型是脆弱的，它的类型系统复杂且低能，它的数据完整性几乎完全程序化，并且它还引入了几十年前就淘汰了的分层数据结构。然而，XML格式的数据对于在Java中使用却很有吸引力；因为我们有一些好工具。例如，可以用XSLT转换XML数据，这是我们认为最佳的使用案例之一。

Hibernate没有内建的功能来存储XML格式的数据；它依赖于关系表示和SQL，这种策略的好处应该很清楚。另一方面，Hibernate能够以XML格式把数据加载和呈现给应用程序的开发人员。它允许你使用一组复杂的工具，而无需任何额外的转换步骤。

假设你在默认的POJO模式下工作，并且很快想要获得一些以XML表示的数据。用EntityMode.DOM4J打开一个临时的Session：

```
Session dom4jSession = session.getSession(EntityMode.DOM4J);

Element userXML =
    (Element) dom4jSession.load(User.class, storedUserId);
```

此处返回的是一个dom4j Element，可以用这个dom4j API来读取和操作它。例如，可以用下列片段把它很漂亮地打印到控制台：

```
try {
    OutputFormat format = OutputFormat.createPrettyPrint();
    XMLWriter writer = new XMLWriter( System.out, format);
    writer.write( userXML );
} catch (IOException ex) {
    throw new RuntimeException(ex);
}
```

如果假设你重用来自前面那些例子中的POJO类和数据，你会看到一个User实例和两个Item实例（为了清楚起见，我们不再命名它们为UserPojo和ItemPojo）：

```
<User>
  <id>1</id>
  <username>johndoe</username>
  <itemsForSale>
    <Item>
      <id>2</id>
      <initialPrice>99</initialPrice>
      <description>An item for auction</description>
      <seller>1</seller>
    </Item>
    <Item>
      <id>3</id>
      <initialPrice>123</initialPrice>
      <description>Another item for auction</description>
      <seller>1</seller>
    </Item>
  </itemsForSale>
</User>
```

Hibernate假设默认的XML元素名称——实体和属性名称。你也会看到集合元素被嵌入，和通过标识符（<seller>元素）解析循环引用。

可以通过把node属性添加到Hibernate映射元数据来改变这个默认的XML表示法：

```
<hibernate-mapping>

<class name="Item" table="ITEM_ENTITY" node="item">

    <id name="id" type="long" column="ITEM_ID" node="@id">
        <generator class="native"/>
    </id>

    <property name="initialPrice"
              type="big_decimal"
              column="INIT_PRICE"
              node="item-details/@initial-price"/>

    <property name="description"
              type="string"
              column="DESCRIPTION"
              node="item-details/@description"/>

    <many-to-one name="seller"
                 class="User"
                 column="USER_ID"
                 embed-xml="false"
                 node="@seller-id"/>

</class>

<class name="User" table="USERS" node="user">

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

    <property name="username"
              type="string"
              column="USERNAME"
              node="@username"/>

    <bag name="itemsForSale" inverse="true" cascade="all"
         embed-xml="true" node="items-for-sale">
        <key column="USER_ID"/>
        <one-to-many class="Item"/>
    </bag>

</class>

</hibernate-mapping>
```

每一个node属性都定义了XML表示法：

- <class>映射中的node="name"属性，给该实体定义了XML元素的名称。
- 任何属性映射中的node="name"属性，指定属性内容应该表示为给定名称的XML元素文本。
- 任何属性映射中的node="@name"属性，指定属性内容应该表示为给定名称的XML属性值。
- 任何属性映射中的node="name/@attname"属性，指定属性内容应该在给定名称的一个子元素中表示为给定名称的XML属性值。

embed-xml选项被用于触发被关联实体数据的嵌入或者引用。更新过的映射产生下列你之前已经见过的相同数据的XML表示法：

```
<user id="1" username="johndoe">
  <items-for-sale>
    <item id="2" seller-id="1">
      <item-details initial-price="99"
          description="An item for auction"/>
    </item>
    <item id="3" seller-id="1">
      <item-details initial-price="123"
          description="Another item for auction"/>
    </item>
  </items-for-sale>
</user>
```

小心这个embed-xml选项——你可能很容易创建导致无限循环的循环引用！

最后，以XML表示的数据是事务的和持久化的，因此你可以修改被查询的XML元素，并让Hibernate负责更新底层的表：

```
Element itemXML =
    (Element) dom4jSession.get(Item.class, storedItemId);

itemXML.element("item-details")
        .attribute("initial-price")
         .setValue("100");
session.flush(); // Hibernate executes UPDATEs

Element userXML =
    (Element) dom4jSession.get(User.class, storedUserId);

Element newItem = DocumentHelper.createElement("item");
Element newItemDetails = newItem.addElement("item-details");
newItem.addAttribute("seller-id",
                     userXml.attribute("id").getValue() );
newItemDetails.addAttribute("initial-price", "123");
newItemDetails.addAttribute("description", "A third item");

dom4jSession.save(Item.class.getName(), newItem);

dom4jSession.flush(); // Hibernate executes INSERTs
```

对于如何处理Hiberante返回的XML，没有限制。可以用喜欢的任何方式显示、导出和转换它。更多信息请见dom4j文档。

最后，注意如果你喜欢，可以同时使用所有三种内建的实体模式。可以映射领域模型的一个静态的POJO实现，为一般的用户界面切换到动态的映射，并把数据导到XML里面。或者，可以编写一个没有任何领域类的应用，只有动态的映射和XML。可是，必须提醒你，软件行业中的原型经常意味着，客户以这个没有人想要抛弃的原型而告终——你愿意买一辆样板车吗？强烈建议你依赖静态的领域模型，如果要创建可维护的系统的话。

本书不再考虑动态的模型或者XML表示法。反之，我们把焦点放在静态的持久化类以及它

们的映射方式上。

3.5　小结

本章着重讨论了Java中富领域模型的设计和实现。

现在你了解了领域模型中的持久化类应该摆脱横切关注点（例如事务和安全性）。甚至与持久化相关的关注点也不应该渗透到领域模型实现中去。你也知道了如果要独立、轻松地执行和测试业务对象，透明的持久化是多么重要。

你已经学习了POJO、JPA实体编程模型的最佳实践和必要条件，以及它们有哪些概念与旧JavaBean规范的相同。我们深入探讨了持久化类的实现，以及如何最好地表示属性和关系。

为了给本书的后续部分做准备，并学习所有ORM选项，你要做一个有根据的决定：使用XML映射文件，还是JDK 5.0注解，或者可能是两者的结合。现在你准备好以这两种格式编写更复杂的映射了。

为了方便起见，表3-1概括了本章所讨论的Hibernate和Java Persistence有关概念之间的区别。

表3-1　第3章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
持久化类需要一个无参构造函数，包含public或protected的可见性，如果使用基于代理的延迟加载的话	JPA规范强制所有的实体类都有一个无参构造函数，包含public或protected的可见性
持久化集合必须被输入到接口。Hibernate支持所有JDK接口	持久化集合必须被输入到接口。只有所有接口的一个子集（例如没有排序的集合）被认为是完全可移植的
持久化属性可以在运行时通过字段或者访问方法被访问，或者可以应用一种完全可以定制的策略	实体类的持久化属性通过字段或者访问方法被访问，但是如果需要完全的可移植性，则并非这两种都行
XML元数据格式支持所有可能的Hibernate映射选项	JPA注解涵盖了所有基础的和最高级的映射选项。怪异的映射和调优需要Hibernate Annotations
XML映射元数据可以被全局地定义，并且XML占位符被用来使元数据免于依赖	全局的元数据只有在标准的orm.xml元数据文件中声明了，才是完全可移植的

本书的后续部分将介绍所有可能的基础的和一些高级的映射技术，用于类、属性、继承、集合和关联。你将学习如何解决结构化的对象/关系不匹配。

Part 2 第二部分

映射概念和策略

这部分关于实际的ORM，从类和属性到表和列。第4章从普通的类和属性映射开始，阐述了如何映射细粒度的Java领域模型。第5章介绍如何映射更复杂的类继承层次结构，以及如何利用强大的定制映射类型系统来扩展Hibernate的功能。第6章和第7章通过许多复杂的示例介绍如何映射Java集合和类之间的关联。最后，如果需要在现有的应用程序中引入Hibernate，或者如果必须使用遗留的数据库模式和手写的SQL，就会发现第8章最值得关注。我们还在这一章讨论为模式生成定制的SQL DDL。

读完这部分，你将可以开始快速创建甚至是最复杂的映射，并且使用正确的策略。你将理解如何解决继承映射的问题，以及集合和关联如何映射。你将还能够调优和定制Hibernate，以便与任何现有的数据库模式或者应用程序整合。

本部分内容