第5章 继承和定制类型

本章内容

- 继承映射策略
- Hibernate映射类型系统
- 映射类型的定制

我们此前有意不过多讨论继承映射。把类的层次结构映射到表可能是一个复杂的问题，本章将介绍各种策略。你将学习在一个特定的场景中要选择哪种策略。

Hibernate类型系统，包含把Java值类型的属性转换为SQL数据类型的所有内建的转换器，是本章要讨论的第二大主题。

让我们从实体继承的映射开始。

5.1 映射类继承

把类映射到数据库表的一种简单的策略可能是“每个实体持久化类对应一张表”。这种方法听起来够简单，在我们遇到继承之前确实一直运行得很好。

继承是面向对象和关系世界之间如此明显的一种结构的不匹配，因为面向对象的系统模型既是is a又是has a关系。基于SQL的模型只提供实体之间的has a关系；SQL数据库管理系统不支持类型继承——甚至即使可用，通常也是私有的或者不完全的。

表示一个继承层次结构有4种不同的方法：

- 每个带有隐式多态的具体类一张表——使用非显式的继承映射和默认的运行时多态行为。
- 每个具体类一张表——完全放弃来自SQL模式的多态和继承关系。
- 每个类层次结构一张表——通过反规范化SQL模式启用多态，并利用保存类型信息的一个类型辨别标志列。
- 每个子类一张表——把is a（继承）关系表示为has a（外键）关系。

本节采用了一种自顶向下的方法；假设你正从一个领域模型开始，并试图获取一个新的SQL Schema。然而，如果你自底向上从现有的数据库表开始时，所描述的映射策略正好有关。我们将介绍一些帮助你处理不理想的表设计的技巧。

5.1.1　每个带有隐式多态的具体类一张表

假设我们坚持用建议过的最简单的方法。可以准确地给每个（非抽象的）类使用一张表。类的所有属性，包括被继承的属性，都可以被映射到这张表的列，如图5-1所示。

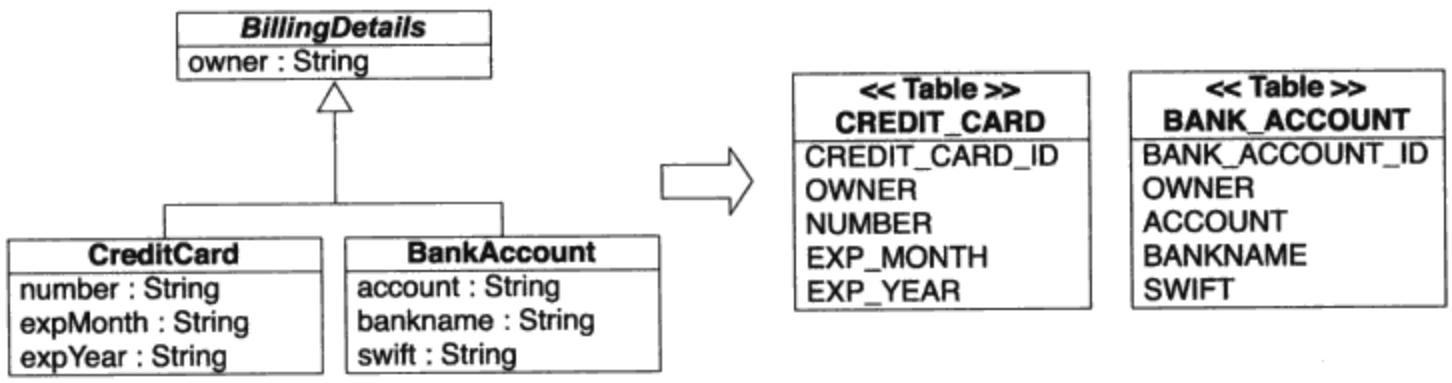


图5-1　把所有具体类映射到一张独立的表

在Hibernate中你不必做任何特别的事来启用多态行为。对CreditCard和BankAccount的映射很简单，各自都在自己的实体<class>元素中，就像我们已经对没有基类（或者持久化接口）的类所做的那样。Hibernate仍然知道这个基类（或者任何接口），因为它在启动时扫描持久化类。

这种方法的主要问题在于，它不太支持多态关联。在数据库中，关联通常被表示为外键关系。在图5-1中，如果子类全部被映射到不同的表，对它们基类（本例中为抽象的BillingDetails）的多态关联就无法被表示为一个简单的外键关系。这在我们的领域模型中会有问题，因为BillingDetails与User关联；两张子类表都需要一个对USERS表的外键引用。或者，如果User与BillingDetails有个多对一的关系，USERS表就需要单个的外键列，它必须同时引用两个具体的子类表。这用一般的外键约束是不可能的。

多态查询（它返回与被查询类的接口相匹配的所有类的对象）也有问题。针对基类的查询必须作为几个SQL SELECT执行，每个具体的子类一个。对于针对BillingDetails类的查询，Hibernate使用下列SQL：

```
select CREDIT_CARD_ID, OWNER, NUMBER, EXP_MONTH, EXP_YEAR ...
from CREDIT_CARD

select BANK_ACCOUNT_ID, OWNER, ACCOUNT, BANKNAME, ...
from BANK_ACCOUNT
```

注意每个具体的子类都需要一个单独的查询。另一方面，针对具体类的查询则十分繁琐，并且执行得很好——只需要一条语句。

（还要注意本书此处以及其他地方，都是介绍概念上与Hibernate执行的SQL一致的SQL。实际的SQL看起来可能稍有不同。）

这种映射策略进一步的概念问题在于，几张不同表的几个不同的列具有完全相同的语义。这使得模式演变更为复杂。例如，对一个基类属性的改变导致了多个列的变化。它也使得把应用到所有子类的数据库完整性约束实现起来变得更加困难。

我们推荐这种方法（仅仅）用于类层次结构的最顶层，那里通常不需要多态，且未来也不太可能修改基类。

Java Persistence接口也不支持完全的多态查询；只有被映射的实体（@Entity）可以正式成为Java Persistence查询的一部分（注意Hibernate查询接口是多态的，即使你用注解映射）。

如果依赖这个隐式多态，就像通常一样用@Entity映射具体的类。但是也必须复制基类的属性，把它们映射到所有具体的类表。默认情况下，基类的属性被忽略并且不是持久化的！你要在具体的子类表中注解超类来启用它属性的嵌入：

```
@MappedSuperclass
public abstract class BillingDetails {

    @Column(name = "OWNER", nullable = false)
    private String owner;

        ...
}
```

现在映射具体的子类：

```
@Entity
@AttributeOverride(name = "owner", column =
  @Column(name = "CC_OWNER", nullable = false)
)
public class CreditCard extends BillingDetails {

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

    @Column(name = "NUMBER", nullable = false)
    private String number;

    ...
}
```

可以用@AttributeOverride注解在一个子类中覆盖来自超类的列映射。在CREDIT_CARD表中重新命名OWNER列为CC_OWNER。数据库标识符也可以在超类中声明，给所有的子类使用一个共用的列名和生成器策略。

在JPA XML描述符中重复相同的映射：

```
<entity-mappings>

    <mapped-superclass class="auction.model.BillingDetails"
                       access="FIELD">
        <attributes>
            ...
        </attributes>
    </mapped-superclass>

    <entity class="auction.model.CreditCard" access="FIELD">
        <attribute-override name="owner">
            <column name="CC_OWNER" nullable="false"/>
        </attribute-override>
        <attributes>
```

```
            ...
        </attributes>
    </entity>
    ...
</entity-mappings>
```

说明　组件是值类型；因此，本章介绍的一般实体继承规则不适用。但是可以把子类映射为组件，通过把超类（或者接口）的所有属性都包括在组件映射中。有了注解，可以在正映射的可嵌入组件的超类上使用@MappedSuperclass注解，就像对实体使用时一样。注意该特性仅在Hibernate Annotations中可用，不是标准的或者可移植的。

在SQL UNION操作的帮助下，可以解决多态查询和关联的大部分问题，这些问题是这个映射策略所带来的。

5.1.2　每个带有联合的具体类一张表

首先，让我们考虑用BillingDetails把联合子类映射为抽象类（或者接口），就像前一节一样。在这种情况下，我们再次有了两张表，并在这两张表中都复制超类列：CREDIT_CARD和BANK_ACCOUNT。这里新出现的东西是包括超类的一个特殊的Hibernate映射，如代码清单5-1所示。

代码清单5-1　使用<union-subclass>继承策略

```
<hibernate-mapping>
    <class
        name="BillingDetails"                    ❶
        abstract="true">
        <id                                      ❷
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
        </id>
        <property                                ❸
            name="name"
            column="OWNER"
            type="string"/>
            ...
        <union-subclass                          ❹
            name="CreditCard" table="CREDIT_CARD">
            <property name="number" column="NUMBER"/>
            <property name="expMonth" column="EXP_MONTH"/>
            <property name="expYear" column="EXP_YEAR"/>
        </union-subclass>
```

```
        <union-subclass
            name="BankAccount" table="BANK_ACCOUNT">
        ...
    </class>
</hibernate-mapping>
```

❶ 抽象的超类或者接口必须被声明为abstract="true"；否则超类的实例就需要一张单独的表。

❷ 数据库标识符映射被该层中所有的具体类共用。CREDIT_CARD和BANK_ACCOUNT表都有一个BILLING_DETAILS_ID主键列。现在数据库标识符属性必须被所有的子类共享；因而你必须把它移到BillingDetails里面，并且从CreditCard和BankAccount中把它移除。

❸ 超类（或者接口）的属性在此处声明，并被所有具体的类映射继承。这样避免了相同映射的复制。

❹ 具体的子类被映射到一张表；表继承超类（或者接口）标识符和其他的属性映射。

你可能注意到了，这种策略的第一个好处是超类（或者接口）属性的共享声明。你不再非得给所有具体的类复制这些映射了——Hibernate会替你完成。记住，SQL模式仍然不知道继承；实际上，我们已经把两个不相关的表映射到了一个更富有表现力的类结构。除了不同的主键列名之外，这两张表看起来完全相同，如图5-1所示。

在JPA注解中，这个策略称作TABLE_PER_CLASS：

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {

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

    @Column(name = "OWNER", nullable = false)
    private String owner;

    ...
}
```

数据库标识符和它的映射必须出现在超类中，在所有子类和它们的表中共享。每个子类中的@Entity注解便是所需要的全部：

```
@Entity
@Table(name = "CREDIT_CARD")
public class CreditCard extends BillingDetails {

    @Column(name = "NUMBER", nullable = false)
    private String number;

    ...
}
```

注意，TABLE_PER_CLASS在JPA标准中被指定为可选，因此并非所有的JPA实现都可以支持它。实际的实现也是由供应商决定的——在Hibernate中，它相当于XML文件中的<union-subclass>映射。

同样的映射在JPA XML描述符中看起来像这样：

```
<entity-mappings>

    <entity class="auction.model.BillingDetails" access="FIELD">
        <inheritance strategy="TABLE_PER_CLASS"/>
        ...
    </entity>

    <entity class="auction.model.CreditCard" access="FIELD"/>
    <entity class="auction.model.BankAccount" access="FIELD"/>

</entity-mappings>
```

如果超类是具体的，就需要另外一张表来存放这个类的实例。必须再次强调，数据库表之间仍然没有关系，除了它们共享一些类似的列之外。如果检验多态查询，这个映射策略的好处会更加明显。例如，对BillingDetails的查询执行下列SQL语句：

```
select
    BILLING_DETAILS_ID, OWNER,
    NUMBER, EXP_MONTH, EXP_YEAR,
    ACCOUNT, BANKNAME, SWIFT
    CLAZZ_
from
  ( select
      BILLING_DETAILS_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, OWNER,
        null as NUMBER, null as EXP_MONTH, null as EXP_YEAR, ...
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
      from
        BANK_ACCOUNT
  )
```

这个SELECT使用一个FROM子句的子查询，从所有具体的类表中获取BillingDetails的所有实例。这些表与一个UNION操作符结合起来，并把文字（在这个例子中是1和2）插入到中间结果中；Hibernate读取结果，通过特定行中的数据实例化正确的类。联合（union）要求被合并的查询要在相同的列上方投影；因而，我们必须用NULL填满不存在的列。你可能会问，这个查询是否真的比两个单独的声明执行得更好？这里我们可以让数据库优化器找到最好的执行计划，合并来自几张表的行，而不是在内存中合并两个结果集，就像Hibernate的多态加载器引擎所做的那样。

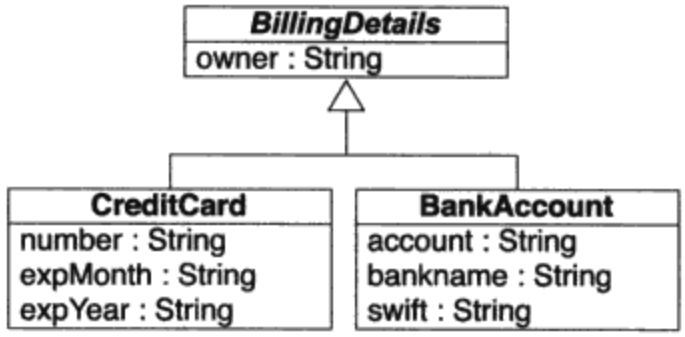
另一个更为重要的好处是处理多态关联的能力；例如，从User到BillingDetails的关联映射现在就有可能了。Hibernate可以用一个UNION查询把单张表模拟成为关联映射的目标。7.3节将

深入讨论这一主题。

目前为止，我们所讨论的继承映射策略都不需要额外考虑SQL Schema。不需要外键，并且关系也被适当地标准化。这种情况随着下一个策略而改变。

5.1.3 每个类层次结构一张表

整个类层次结构可以被映射到单张表。这张表把所有类的所有属性的列都包括在层次结构中。由特定行表示的具体子类通过一个类型辨别标志列的值进行识别。这种方法如图5-2所示。



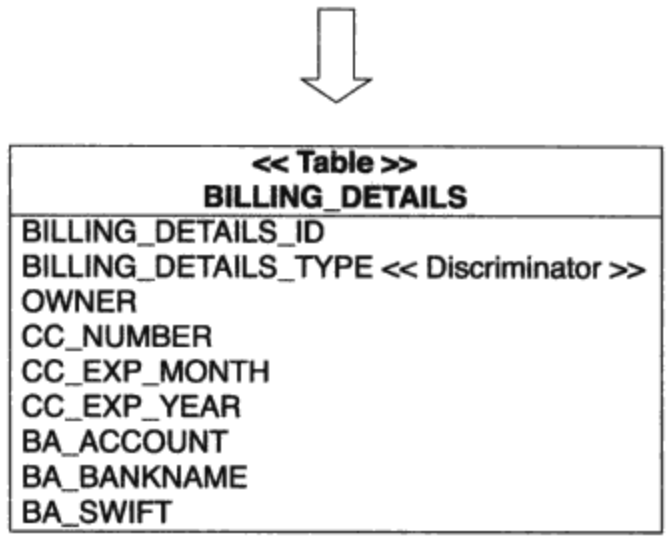


图5-2 把整个类层次结构映射到单张表

这个映射策略在性能和简单性方面都胜出一筹。它是表示多态的最佳方法——多态和非多态的查询都执行得很好——并且更易于手工实现。不用复杂的联结或者联合也有可能生成特殊的报表。Schema演变很简单。

有一个重大的问题：子类声明的属性的列必须声明为可为空。如果每个子类都定义几个不可为空的属性，从数据库完整性的角度来说，没有NOT NULL约束可能会是一个严重的问题。另一个重要的问题是标准化。我们已经创建了非键列之间的功能依赖，违背了第三范式。跟往常一样，性能的反规范化可能会令人误解，因为它为了也能通过SQL执行计划的适当优化而也可能实现的即时获取（换句话说，问问你的数据库管理员），却牺牲了数据的长期稳定性、可维护性和数据完整性。

在Hibernate中，用<subclass>元素给每个类层次结构映射创建一张表，如代码清单5-2所示。

代码清单5-2　Hibernate <subclass>映射

```
<hibernate-mapping>
    <class                                    ❶
        name="BillingDetails"
        table="BILLING_DETAILS">

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

        <discriminator                        ❷
            column="BILLING_DETAILS_TYPE"
            type="string"/>

        <property                             ❸
            name="owner"
            column="OWNER"
            type="string"/>

            ...

        <subclass                             ❹
            name="CreditCard"
            discriminator-value="CC">

            <property name="number" column="CC_NUMBER"/>
            <property name="expMonth" column="CC_EXP_MONTH"/>
            <property name="expYear" column="CC_EXP_YEAR"/>

        </subclass>

        <subclass
            name="BankAccount"
            discriminator-value="BA">
              ...

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

❶ 继承层次结构的根类BillingDetails被映射到表BILLING_DETAILS。

❷ 必须添加一个特殊的列以便在持久化类之间进行区分：辨别标志（discriminator）。这不是持久化类的属性，由Hibernate内部使用。列名为BILLING_DETAILS_TYPE，值为字符串——在这个例子中，为“CC”或者“BA”。Hibernate自动设置和获取辨别标志的值。

❸ 同往常一样，用一个简单的<property>元素映射超类的属性。

❹ 每个子类都有自己的<subclass>元素。子类的属性被映射到BILLING_DETAILS表中的列。记住不允许NOT NULL约束，因为BankAccount实例不会有expMonth属性，且该行的CC_EXP_MONTH字段必须为NULL。

<subclass>元素可以依次包含其他被嵌套的<subclass>元素，直到整个层次结构被映射到表中。

Hibernate在查询BillingDetails类时生成下列SQL：

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC_NUMBER, CC_EXP_MONTH, ..., BA_ACCOUNT, BA_BANKNAME, ...
from BILLING_DETAILS
```

为了查询CreditCard子类，Hibernate在辨别标志列中添加一个限制：

```
select BILLING_DETAILS_ID, OWNER, CC_NUMBER, CC_EXP_MONTH, ...
from BILLING_DETAILS
where BILLING_DETAILS_TYPE='CC'
```

JPA中也有这个映射策略，即SINGLE_TABLE：

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name = "BILLING_DETAILS_TYPE",
    discriminatorType = DiscriminatorType.STRING
)
public abstract class BillingDetails {

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

    @Column(name = "OWNER", nullable = false)
    private String owner;

    ...
}
```

如果没有在超类中指定辨别标志列，它的名称就默认为DTYPE，且它的类型默认为字符串。继承层次结构中所有具体的类都可以有辨别标志值；在这个例子中，BillingDetails是抽象类，CreditCard则是具体的类：

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {

    @Column(name = "CC_NUMBER")
    private String number;

    ...
}
```

没有显式的辨别标志值时，如果使用Hibernate XML文件，Hibernate就默认为完全限定的类名；如果使用注解或者JPA XML文件，则默认为实体名称。注意，在Java Persistence中，没有给非字符串辨别标志类型指定默认值，每个持久化提供程序都可以有不同的默认值。

以下是JPA XML描述符中与之相当的映射：

```
<entity-mappings>
    <entity class="auction.model.BillingDetails" access="FIELD">
        <inheritance strategy="SINGLE_TABLE"/>
        <discriminator-column name="BILLING_DETAILS_TYPE"
            discriminator-type="STRING"/>
```

```
        ...
    </entity>

    <entity class="auction.model.CreditCard" access="FIELD">
        <discriminator-value>CC</discriminator-value>
        ...
    </entity>

</entity-mappings>
```

有时候，尤其在遗留的Schema中，你无权在实体表中包括一个额外的辨别标志列。此时，可以应用一个formula来计算每一行的辨别标志值：

```
<discriminator
  formula="case when CC_NUMBER is not null then 'CC' else 'BA' end"
  type="string"/>

  ...

<subclass
    name="CreditCard"
    discriminator-value="CC">
  ...
```

这个映射依赖于SQL CASE/WHEN表达式，来决定一个特定的行是表示信用卡还是银行账号（许多开发人员从来不用这种SQL表达式；如果你不熟悉它的话请查阅ANSI标准）。表达式的结果是文字，CC或者BA，依次在<subclass>映射中声明。用于辨别的公式不是JPA规范的一部分，但是可以应用Hibernate注解：

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@org.hibernate.annotations.DiscriminatorFormula(
    "case when CC_NUMBER is not null then 'CC' else 'BA' end"
)
public abstract class BillingDetails {
    ...
}
```

每个类层次结构一张表的策略的缺点对于你的设计来说可能太严重了——毕竟，反规范化的Schema长期而言会变成一个重大的负担。你的数据库管理员根本不可能喜欢它。下一个继承映射策略则不会给你造成这个问题。

5.1.4　每个子类一张表

第四种方案是把继承关系表示为相关的外键关联。声明持久化属性的每个类/子类（包括抽象类甚至接口）都有它自己的表。

不同于我们最先映射的每个具体类一张表的策略，此处的表仅仅包含了每个非继承的属性（由子类本身声明的每个属性）以及也是超类表的外键的主键的列。这种方法如图5-3所示。

如果CreditCard子类的一个实例变成持久化，由BillingDetails超类声明的属性值就被持久化到BILLING_DETAILS表的一个新行。只有子类声明的属性值被持久化到CREDIT_CARD表的一个新行。这两行通过它们的共享主键值链接在一起。随后，通过联结子类表与超类表，可以从数

据库中获取子类实例。

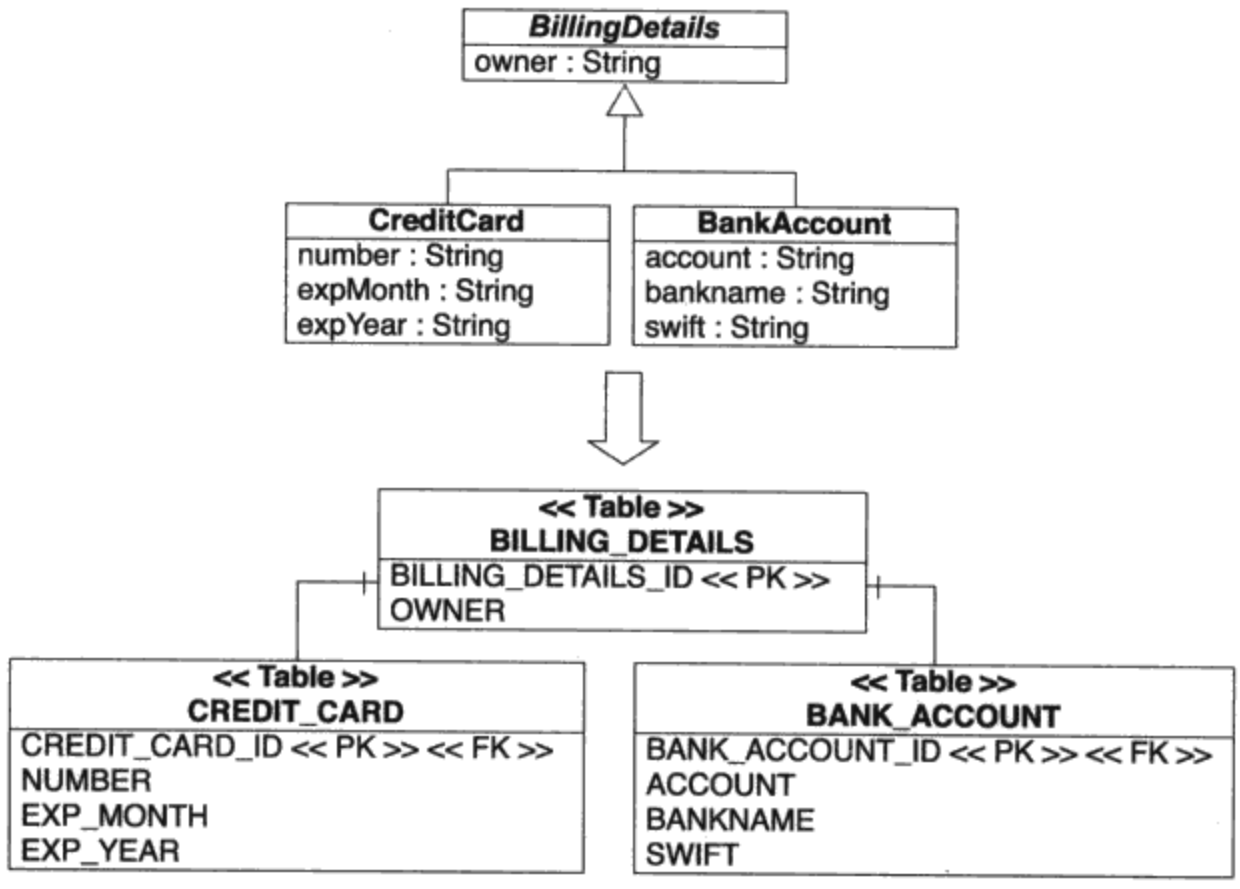


图5-3 把层次结构的所有类映射到它们自己的表

这一策略的主要好处在于，SQL Schema被标准化了。Schema演变和完整性约束定义很简单。对一个特定子类的多态关联可能被表示为引用这个特定子类的表的一个外键。

在Hibernate中，用<joined-subclass>元素给每个子类映射创建一张表。请见代码清单5-3。

代码清单5-3 Hibernate <joined-subclass>映射

```
<hibernate-mapping>
    <class                    ❶
        name="BillingDetails"
        table="BILLING_DETAILS">

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

                  <property
                      name="owner"
                      column="OWNER"
                      type="string"/>

                      ...

                  <joined-subclass    ❷
```

```
            name="CreditCard"
            table="CREDIT_CARD">

            <key column="CREDIT_CARD_ID"/>

            <property name="number" column="NUMBER"/>
            <property name="expMonth" column="EXP_MONTH"/>
            <property name="expYear" column="EXP_YEAR"/>

        </joined-subclass>

        <joined-subclass
            name="BankAccount"
            table="BANK_ACCOUNT">
        ...

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

❶ 根类BillingDetails被映射到表BILLING_DETAILS。注意这个策略不需要辨别标志。

❷ 新的<joined-subclass>元素把子类映射到新的表——在这个例子中为CREDIT_CARD。在被联结的子类中声明的所有属性都被映射到这张表。

❸ CREDIT_CARD表需要主键。这个列也有一个对BILLING_DEAILS表的主键的外键约束。CreditCard对象查找需要这两张表的联结。<joined-subclass>元素可能包含其他嵌套的<joined-subclass>元素，直到整个层次结构已经被映射。

当查询BillingDetails类时，Hibernate依赖一个外部联结：

```
select BD.BILLING_DETAILS_ID, BD.OWNER,
       CC.NUMBER, CC.EXP_MONTH, ..., BA.ACCOUNT, BA.BANKNAME, ...
    case
        when CC.CREDIT_CARD_ID is not null then 1
        when BA.BANK_ACCOUNT_ID is not null then 2
        when BD.BILLING_DETAILS_ID is not null then 0
    end as CLAZZ_
from BILLING_DETAILS BD
    left join CREDIT_CARD CC
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
    left join BANK_ACCOUNT BA
      on BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
```

SQL CASE语句在子类表CREDIT_CARD和BANK_ACCOUNT中侦测行的存在（或者不存在），以便Hibernate可以给BILLING_DETAILS表的一个特定行决定具体的子类。

为了缩小对子类的查询，Hibernate使用了内部联结：

```
select BD.BILLING_DETAILS_ID, BD.OWNER, CC.NUMBER, ...
from CREDIT_CARD CC
    inner join BILLING_DETAILS BD
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```

如你所见，这个映射策略更难以手工实现——甚至生成特殊的报告也更加复杂。如果计划把Hibernate代码和手写的SQL混合，这是一个需要考虑的重要事项。

此外，虽然这个映射策略似乎很简单，但依我们的经验，该性能无法为复杂的类层次结构所

接受。查询始终需要跨多张表的联结，或者多个连续读取。

用相同的策略和注解映射层次结构，此处称作JOINED策略：

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {

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

    ...
}
```

在子类中，如果子类表的主键列有着（或者将有）与超类表的主键列相同的名称时，就不需要指定联结列：

```
@Entity
public class BankAccount extends BillingDetails {
    ...
}
```

这个实体没有标识符属性；它自动从超类继承BILLING_DETAILS_ID属性和列，如果想要获取BankAccount的实例，Hibernate就知道如何把表联结到一起。当然，可以显式地指定列名：

```
@Entity
@PrimaryKeyJoinColumn(name = "CREDIT_CARD_ID")
public class CreditCard extends BillingDetails{
    ...
}
```

最后，下面是在JPA XML描述符中与之相当的映射：

```
<entity-mappings>

    <entity class="auction.model.BillingDetails" access="FIELD">
        <inheritance strategy="JOINED"/>
        ...
    </entity>

    <entity class="auction.model.BankAccount" access="FIELD"/>
    <entity class="auction.model.CreditCard" access="FIELD">
        <primary-key-join-column name="CREDIT_CARD_ID"/>
    </entity>

</entity-mappings>
```

在介绍何时选择何种策略之前，先思考在单个类层次结构中混合继承映射策略。

5.1.5 混合继承策略

通过嵌套<union-subclass>、<subclass>和<joined-subclass>映射元素可以映射整个继承层次结构。你无法把它们混合起来——例如，用一个辨别标志从“每个类层次结构一张表”策略转换到标准的“每个子类一张表”策略。一旦决定使用某种继承策略，就必须坚持使用它。

但这并不完全正确。使用一些Hibernate技巧，可以给一个特定的子类切换映射策略。例如，

可以把一个类层次结构映射到单张表，但是对于特定的子类，则通过外键映射策略切换到单独的表，就像使用每个子类一张表一样。使用<join>映射元素，这种转换是可能的：

```
<hibernate-mapping>
<class name="BillingDetails"
       table="BILLING_DETAILS">

    <id>...</id>

    <discriminator
        column="BILLING_DETAILS_TYPE"
        type="string"/>
        ...

    <subclass
        name="CreditCard"
        discriminator-value="CC">

        <join table="CREDIT_CARD">
            <key column="CREDIT_CARD_ID"/>

            <property name="number" column="CC_NUMBER"/>
            <property name="expMonth" column="CC_EXP_MONTH"/>
            <property name="expYear" column="CC_EXP_YEAR"/>
            ...
        </join>

    </subclass>

    <subclass
        name="BankAccount"
        discriminator-value="BA">

        <property name=account" column="BA_ACCOUNT"/>
        ...
    </subclass>

...

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

<join>元素集合了一些属性，并告诉Hibernate到一个二级表中获取它们。这个映射元素有多种用途，你会在本书稍后再次见到它。在这个例子中，它把CreditCard属性从每个层次结构一张表中分离出来，放到CREDIT_CARD表中。这张表的CREDIT_CARD_ID列同时也是主键，并且它有引用该层次结构表的BILLING_DETAILS_ID的外键约束。BankAccount子类被映射到层次结构表。看看图5-4中的Schema。

运行时，Hibernate执行一个外部联结，多态地抓取BillingDetails和所有的子类实例：

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC.CC_NUMBER, CC.CC_EXP_MONTH, CC.CC_EXP_YEAR,
    BA_ACCOUNT, BA_BANKNAME, BA_SWIFT

from
    BILLING_DETAILS
left outer join
```

```
CREDIT_CARD CC
    on BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```

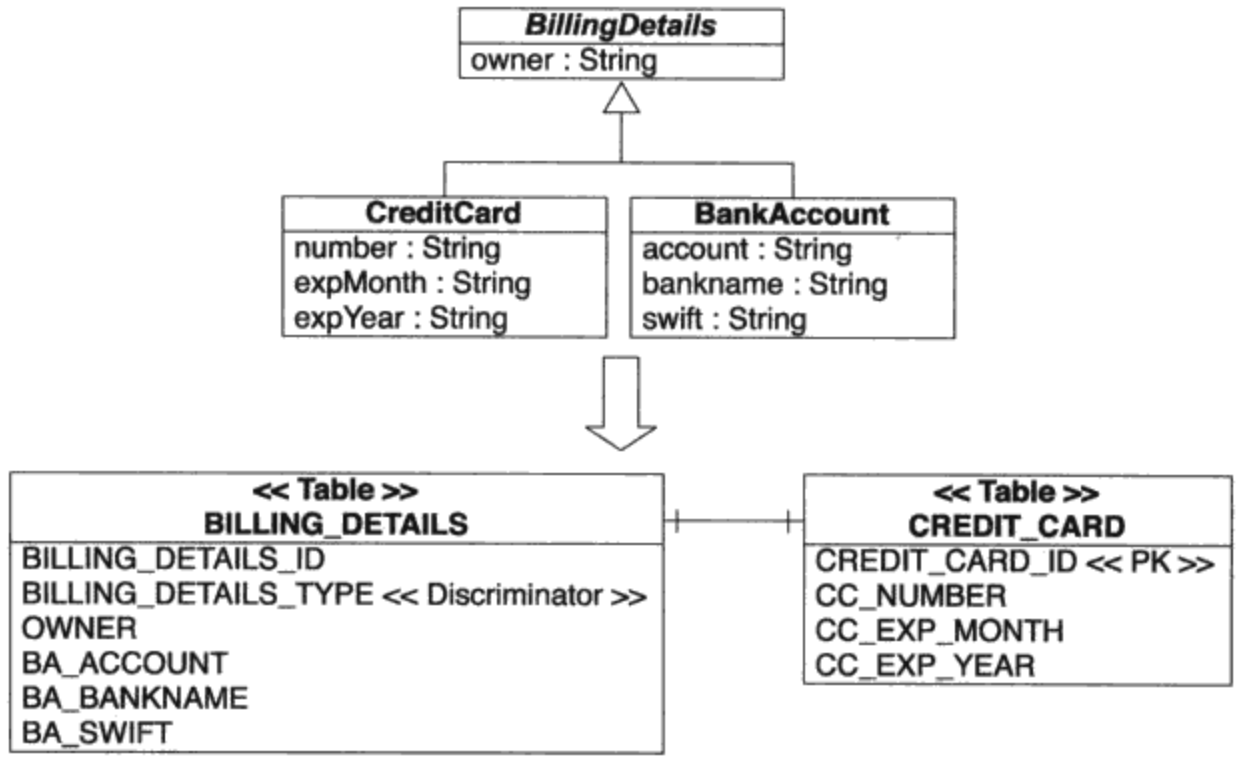


图5-4 把一个子类分到它自己的二级表中

也可以在类层次结构中对其他的子类使用<join>技巧。然而，如果你有一个宽到出乎意料的类层次结构，外部联结就可能成问题。有些数据库系统（例如Oracle）限制外部联结操作中的表数目。对于宽层次结构来说，你可能想要切换到立即执行第二选择而不是外部联结的不同抓取策略：

```
<subclass
    name="CreditCard"
    discriminator-value="CC">

    <join table="CREDIT_CARD" fetch="select">
        <key column="CREDIT_CARD_ID"/>
        ...
    </join>

</subclass>
```

Java Persistence也通过注解支持这种混合的继承映射策略。就像前面所做的那样，用InheritanceType.SINGLE_TABLE映射超类BillingDetails。现在映射你想要从单张表分到一张二级表的子类。

```
@Entity
@DiscriminatorValue("CC")
@SecondaryTable(
  name = "CREDIT_CARD",
  pkJoinColumns =  @PrimaryKeyJoinColumn(name = "CREDIT_CARD_ID")
)
public class CreditCard extends BillingDetails {

    @Column(table = "CREDIT_CARD",
```

```
                name = "CC_NUMBER",
                nullable = false)
    private String number;
    ...
}
```

如果没有给二级表指定主键联结列，就使用单张继承表的主键名称——在这个例子中为BILLING_DETAILS_ID。还要注意，你要用二级表的名称映射被移到二级表里面的所有属性。

你还想要更多关于如何给应用程序的类层次结构选择合适的映射策略组合的技巧。

5.1.6　选择策略

可以把所有的映射策略应用到抽象类和接口。接口可能没有状态，但是可能包含访问方法声明，因此可以像抽象类一样地处理它们。可以用<class>、<union-subclass>、<subclass>或者<joined-subclass>映射接口，并可以用<property>映射任何被声明或者被继承的属性。Hibernate不会试图实例化抽象类，即使你查询或者加载了它。

说明　注意JPA规范不在接口上支持任何映射注解！这在规范的未来版本中会得到解决；当你读到本书时，或许可能在使用Hibernate Annotations了。

以下是一些经验法则：

- 如果你不需要多态关联或者查询，就倾向于每个具体类一张表——换句话说，如果你从不或者很少查询BillingDetails，并且没有关联BillingDetails的类（我们的模型有）。基于UNION的显式映射应该是首选，因为随后（最优化的）多态查询和关联将成为可能。隐式多态对于利用非持久化相关的接口的查询最有用。
- 如果你一定要多态关联（对超类的关联，及由此在运行时通过具体类的动态解析对层次结构中的所有类的关联）或者查询，并且子类相对地声明几种属性（尤其当子类之间的主要区别在于它们的行为中）时，则倾向于每个类层次结构一张表。你的目标是把可为空的列数减到最少，并让你自己（和数据库管理员）确信反规范化的Schema在长期运行中不会产生问题。
- 如果你一定需要多态的关联或者查询，并且子类声明多个属性（子类的不同主要在于它们所持有的数据），则倾向每个子类一张表。或者，根据继承层次结构的宽度和深度，以及相对于联合的可能联结成本，使用每个具体类一张表。

默认情况下，仅对简单的问题选择每个类层次结构一张表。对于更复杂的案例（或者当你被一位坚持认为可为空的能力约束和标准化很重要的数据建模者否决的时候），就应该考虑每个子类一张表的策略。但这时候要问问自己：在对象模型中，重新把继承建模为委托是否未必更好。对于无关持久化或者ORM的各种原因，通常最好避免复杂的继承。Hibernate充当着领域和关系模型之间的缓冲区，但这并不意味着在设计类的时候，可以忽略持久化关注点。

当你开始考虑混合的继承策略时，记住：Hibernate中的隐式多态很聪明，足以处理更多异乎

寻常的情况。例如，考虑在应用程序中增加一个接口ElectronicPaymentOption。这是一个没有持久化方面的业务接口——除了在我们的应用程序中，例如CreditCard这样的持久化类将可能实现这个接口之外。无论你如何映射BillingDetails层次结构，Hibernate都可以正确地给from ElectronicPaymentOption回复查询。甚至不是BillingDetails层次结构一部分的其他类也可以，它们被映射为持久化，并实现这个接口。Hibernate始终知道要查询哪些表、构造哪些实例，以及如何返回一个多态的结果。

最后，也可以在单个的映射文件中使用<union-subclass>、<subclass>和<joined-subclass>映射元素（作为一级元素，而不是<class>）。然后必须声明被扩展的类，例如<subclassname="CreditCard" extends="BillingDetails">，且必须在子类映射文件之前程序化地加载基类映射（在XML配置文件中列出映射资源清单时，不必担心这个顺序）。这种技术允许扩展类层次结构，而不用修改超类的映射文件。

现在知道了有关实体、属性和继承层次结构的映射所需要知道的一切。已经可以映射复杂的领域模型了。本章后半部分讨论Hibernate用户应该熟记的另一项重要的特性：Hibernate映射类型系统。

5.2 Hibernate 类型系统

第4章首先区分了实体和值类型——Java中ORM的一个中心概念。必须详细阐述这种区别，以便你完全理解Hibernate中实体、值类型和映射类型的类型系统。

5.2.1 概述实体和值类型

实体是系统中粗粒度的类。通常按照涉及的实体定义系统的特性。用户给货品出价（the user places a bid for an item）是一个典型的特性定义；它提到了三个实体。值类型的类甚至经常不出现在业务需求中——它们通常是表示字符串、数字和货币金额的细粒度类。值类型也偶尔出现在特性定义中：用户改变账单地址（the user changes billing address）是一个例子，假设Address是一个值类型。

更正式一点的定义：实体是指其实例具有自己的持久化同一性的任何类。值类型是指没有定义某种持久化同一性的类。在实际应用中，这意味着实体类型是包含标识符属性的类，值类型的类则取决于实体。

运行时，你有一个实体实例的网络与值类型实例交错。实体实例可能处于这三种持久化生命周期状态之一：瞬时（transient）、脱管（detached）或者持久化（persistent）。我们不考虑把这些生命周期状态应用到值类型实例。（第9章再回到对象状态的讨论。）

因此，实体有自己的生命周期。Hibernate Session接口的Save()和delete()方法应用到实体类的实例，而从不应用到值类型实例。值类型实例的持久化生命周期完全由自己的实体实例的生命周期所决定。例如，保存用户时，用户名变成持久化；它从来不会独立于用户而变成持久化。

在Hibernate中，值类型可以定义关联；可能从一个值类型实例导航到一些其他的实体。但是永远不可能从其他实体导航回到值类型实例。关联始终指向实体。这意味着，当从数据库中获取值类型实例时，它完全为一个实体所有；从不共享。

在数据库级中，任何表都被当作实体。然而Hibernate提供某些构造，从Java代码中隐藏数据库级实体的存在。例如，多对多（many-to-many）的关联映射从应用程序中隐藏了中间关联表。从应用程序的观点来看，字符串的集合（更准确地说，是值类型实例的集合）表现得就像一个值类型；但它被映射到它自己的表。虽然这些特性乍看起来不错（它们简化了Java代码），但随着时间的推移，我们对此产生了怀疑。不可避免地，这些被隐藏的实体最终要随着业务需求的演变而公开给应用程序。例如，随着应用程序的成熟，多对多的关联表经常会添加额外的表。我们几乎准备推荐让每个数据库级的实体都被当作实体类公开给应用程序。例如，我们会倾向于把多对多的关联建模为对中间实体类的两个一对多的关联。但是，我们把最终的决定权留给你，并在后面的章节中回到多对多的实体关联的话题。

实体类始终利用<class>、<union-subclass>、<subclass>和<joined-subclass>映射元素被映射到数据库。值类型又如何被映射呢？

你已经见过了两种不同的值类型映射：<property>和<component>。组件（component）的值类型很明显：它是被映射为可嵌入的类。然而，属性（property）的类型是个更为一般的概念。考虑CaveatEmptor User和email地址的这个映射：

```
<property name="email"
          column="EMAIL"
          type="string"/>
```

让我们关注type="string"属性。你知道，在ORM中必须处理Java类型和SQL数据类型。这两个不同的类型系统必须被桥接起来。这是Hibernate映射类型的任务，string是内建的Hibernate映射类型的名称。

string不是Hibernate中内建的唯一映射类型。Hibernate带来了各种映射类型，它们给基本的Java类型和某些JDK类定义了默认的持久化策略。

5.2.2　内建的映射类型

Hibernate内建的映射类型通常共享它们映射的Java类型的名称。然而，一个特定的Java类型可能有不止一个Hibernate映射类型。

内建的类型不能用于执行任意的转化，例如把一个VARCHAR数据库值映射到一个Java的Integer属性值。可以给这种转化定义自己的定制值类型，如本章稍后所述。

现在讨论基本的要素：日期和时间、定位对象，以及各种其他内建的映射类型，并介绍它们处理哪些Java和SQL数据类型。

1．Java基本的映射类型

表5-1中基础的映射类型把Java基本的类型（或者它们的包装类型）映射到适当的内建SQL标准类型。

表5-1 基本的类型

映射类型	Java类型	标准的SQL内建类型
integer	int或java.lang.Integer	INTEGER
long	long或java.lang.Long	BIGINT
short	short或java.lang.Short	SMALLINT
float	float或java.lang.Float	FLOAT
double	double或java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte或java.lang.Byte	TINYINT
boolean	boolean或java.lang.Boolean	BIT
yes_no	boolean或java.lang.Boolean	CHAR(1) ('Y' or 'N')
true_false	boolean或java.lang.Boolean	CHAR(1) ('T' or 'F')

你可能已经注意到你的数据库不支持表5-1中提到的有些SQL类型。上面所列的名称为ANSI标准数据类型的名称。大部分数据库供应商忽略这部分SQL标准（因为他们的遗留类型系统经常把标准上的日期填早了）。然而，JDBC驱动程序提供特定于供应商的SQL数据类型的部分抽象，允许Hibernate在执行DML时使用ANSI标准类型。对于数据库专用的DDL生成，Hibernate利用对特定的SQL方言的内建支持，把ANSI标准类型转变为适当的特定于供应商的类型。（这意味着如果你正在使用Hibernate访问数据和定义SQL Schema，通常不必担心SQL数据类型。）

此外，Hibernate类型系统很聪明，可以根据值定义的长度（length）转换SQL数据类型。最明显的案例是string：如果用length属性声明字符串属性映射，Hibernate会根据选中的方言挑选正确的SQL数据类型。例如，对于MySQL，当Hibernate导出Schema时，长度达到65535时产生一个一般的VARCHAR（length）列。长度达到16777215时，使用MEDIUMTEXT的数据类型。较大的字符串映射则导致使用LONGTEXT。如果想知道这种或者其他映射类型的范围，请检查你的SQL方言（与Hibernate一起带来的源代码）。可以通过把方言作为子类，并覆盖这些设置来定制这种行为。

大部分方言也支持设置十进制SQL数据类型的比例和精度。例如，在BigDecimal映射中设置precision或者scale，给MySQL创建了一个NUMERIC（precision，scale）的数据类型。

最后，yes_no和true_false映射类型是遗留的Schema和Oracle用户最有用的转换器；Oracle DBMS产品没有内建的布尔（boolean）或者真值（truth-valued）类型（关系数据模型唯一真正需要的内建数据类型）。

2. 日期和时间映射类型

表5-2列出了与日期、时间和时间戳关联的Hibernate类型。在领域模型中，可以选择使用java.util.Date、java.util.Calendar，或者在java.sql包中定义的java.util.Date的子类，来表示日期和时间数据。这是个人偏爱的问题，我们留给你自己去决定——但是你要确保一致。（在实际应用中，绑定领域模型到来自JDBC包的类型，并非是最好的办法。）

表5-2　日期和时间类型

映射类型	Java类型	标准的SQL内建类型
date	java.util.Date或java.sql.Date	DATE
time	java.util.Date或java.sql.Time	TIME
timestamp	java.util.Date或java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

警告：如果用timestamp（最常见的情况）映射java.util.Date属性，Hibernate就在从数据库中加载属性时返回java.sql.Timestamp。Hibernate必须使用JDBC子类，因为它包括可能出现在数据库中的十亿分之一秒的信息。Hibernate无法只切断这个信息。如果试图把java.util.Date属性与equals()方法进行比较，就会出现问题，因为它与java.sql.Timestamp子类equals()方法不同步。首先，比较两个java.util.Date对象的正确方法（无论如何），对于任何子类也适用的，是aDate.getTime() > bDate.getTime()（用于大于的比较）。其次，可以编写一个切断数据库十亿分之一秒信息的定制映射类型，并在任何情形下都返回java.util.Date。目前（虽然这在未来可能改变），Hibernate中还没有内建这样的映射类型。

3．二进制和大值映射类型

表5-3列出了用来处理十进制数据和大值的Hibernate类型。注意只有binary作为标识符属性的类型得到支持。

表5-3　二进制和大值类型

映射类型	Java类型	标准的SQL内建类型
binary	byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	任何实现java.io.Serializable的Java类	VARBINARY

如果持久化的Java类中的属性是byte[]类型，Hibernate就可以用二进制映射类型把它映射到一个VARBINARY列。（注意真正的SQL类型取决于方言。例如，在PostgreSQL中，SQL类型是BYTEA；在Oracle中，SQL类型是RAW。）如果持久化的Java类中的属性是java.lang.String类型，Hibernate就会用text映射类型把它映射到SQL CLOB列。

注意在这两个案例中，当加载保存着属性变量的实体实例时，Hibernate立即就初始化属性值。当你必须处理潜在的大值时，这样很不方便。

一种解决方案是按需要通过字段访问的拦截延迟加载。但是对于额外代码的注入，这种方法需要持久化类的字节码基础设施（ByteCode Instrumentation，BCI）。13.1.6节将讨论通过BCI和拦截的延迟加载。

第二种解决方案是Java类中一种不同的属性。JDBC直接支持定位器对象（Locator Object，

LOB)[①]。如果你的Java属性是java.sql.Clob或者java.sql.Blob类型，就可以用clob或者blob映射类型映射它，以便不通过BCI而获得大值的延迟加载。当加载属性的所有者时，属性值就是一个定位对象——实际上，它是一个指向了还没有被物化的真实值的指针。一旦访问了属性，值就被物化了。这种按需加载只在数据库事务打开时才起作用，因此当自己的实体实例处于持久化和事务状态，而不是处于脱管状态时，就可以访问这种类型的任何属性。现在领域模型也被绑定到JDBC了，因为需要导入java.sql包。虽然领域模型类在单独的单元测试中是可执行的，但是没有数据库连接则无法访问LOB属性。

如果依赖Java Persistence注解，包含潜在大值的映射属性则稍有不同。默认情况下，类型java.lang.String的一个属性被映射到SQL VARCHAR列（或者相当的列，取决于SQL方言）。如果想要把java.lang.String、char[]、Character[]，或者甚至java.sql.Clob类型属性映射到CLOB列，就要用@Lob注解映射它：

```
@Lob
@Column(name = "ITEM_DESCRIPTION")
private String description;
@Lob
@Column(name = "ITEM_IMAGE")
private byte[] image;
```

对于byte[]、Byte[]或者java.sql.Blob类型的任何属性也一样。注意在任何情况下，除了java.sql.Clob或者java.sql.Blob类型的属性之外，该值都会立即由Hibernate再次加载，而不是按需延迟加载。包含拦截代码的字节码再次成为透明地启用单独属性的延迟加载的一种选项。

如果领域模型中有这些属性类型，为了创建和设置java.sql.Blob或者java.sql.Clob值，就用静态的Hibernate.createBlob()和Hibernate.createClob()方法，并提供字节数组、输入流或者字符串。

最后，注意Hibernate和JPA都对Serializable（可序列化）的任何属性类型提供序列回滚。这种映射类型把属性的值转换为随后保存在VARBINARY（或者相当的）列中的字节流。当属性的所有者被加载时，属性值是反序列化的。自然地，使用这种策略要非常谨慎（数据存在得比应用程序更长久），并且它可能只对临时数据（用户偏爱、登录会话数据等）有用。

4．JDK映射类型

表5-4为JDK的其他Java类型列出了Hibernate类型，它们可以被表示为数据库中的VARCHAR。你可能已经注意到<property>不是唯一有type属性的Hibernate映射元素。

表5-4 其他JDK相关的类型

映射类型	Java类型	标准的SQL内建类型
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

① 据Jim Starkey（提出LOB概念的人）所说，术语BLOB和CLOB并没有任何意义，只是由营销部门创造出来的。可以用喜欢的任何方式来解释它们。我们更喜欢定位器对象（locator object），暗指它们的作用就像指针一样。

5.2.3　使用映射类型

所有的基础映射类型都可能出现在Hibernate映射文档、一般的属性、标识符属性和其他映射元素中的几乎任何地方。<id>、<property>、<version>、<discriminator>、<index>和<element>元素都定义了具名type的属性。

你会看到，在这个映射中，内建的映射类型对于BillingDetails类是多么有用：

```
<class name="BillingDetails" table="BILLING_DETAILS">
    <id name="id" type="long" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>
    <discriminator type="character" column="BILLING_DETAILS_TYPE"/>
    <property name="number" type="string"/>
    ....
</class>
```

BillingDetails类被映射为实体。它的discriminator、identifier和name属性都为值类型，并且我们使用内建的Hibernate映射类型指定变换策略。

在XML映射文档中，通常不需要显式地指定内建的映射类型。例如，如果你有Java类型java.lang.String的一个属性，Hibernate就通过反射发现它，并默认地选择string。可以轻松地简化前一个映射示例：

```
<class name="BillingDetails" table="BILLING_DETAILS">
    <id name="id" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>
    <discriminator type="character" column="BILLING_DETAILS_TYPE"/>
    <property name="number"/>
    ....
</class>
```

Hibernate也理解type="java.lang.String"，因此它不必使用反射。这种方法不太适用的最重要案例是java.util.Date属性。默认情况下，Hibernate认为java.util.Date是一个timestamp映射。如果不希望日期和时间信息都持久化，就要显式地指定type="time"或者type="date"。

利用JPA注解，自动侦测属性的映射类型，就像在Hibernate中一样。对于java.util.Date或者java.util.Calendar属性，Java Persistence标准要求用@Temporal注解选择精确度：

```
@Temporal(TemporalType.TIMESTAMP)
@Column(nullable = false, updatable = false)
private Date startDate;
```

另一方面，Hibernate Annotations放松了标准的规则，默认为TemporalType.TIMESTAMP——选项是TemporalType.TIME和TemporalType.DATE。

在其他极少数情况下，可能要添加@org.hibernate.annotations.Type注解到一个属性，并显式声明一个内建或者定制的Hibernate映射类型的名称。一旦开始编写自己的定制映射类型，这就立即成为一个更为常见的扩展了，本章稍后就会实践到。

与之相当的JPA XML描述符如下：

```
<entity class="auction.model.Item" access="FIELD">
    <attributes>
        ...
        <basic name="startDate">
            <column nullable="false" updatable="false"/>
            <temporal>TIMESTAMP</temporal>
        </basic>
    </attributes>
</entity>
```

对于每个内建的映射类型，都由类org.hibernate.Hibernate定义常量。例如，Hibernate.STRING表示string映射类型。这些常量对于查询参数绑定很有用，如第14章和第15章中所深入讨论的：

```
session.createQuery("from Item i where i.description like :desc")
    .setParameter("desc", d, Hibernate.STRING)
    .list();
```

注意，也可以在这个例子中用setString()实参绑定方法。类型常量对于Hibernate映射元模型的编程式操作也很有用，如第3章所述。

Hibernate不受限于内建的映射类型。把可扩展的映射类型系统当作使Hibernate变得如此灵活的核心特性之一和重要方面。

5.3 创建定制的映射类型

面向对象的语言比如Java，使得通过编写新类来定义新的类型变得很容易。这是面向对象的定义的一个基础部分。当声明持久化类的属性时，如果受限于预设的内建Hibernate映射类型，就会失去一些Java的表现力。此外，我们的领域模型实现就会与物理的数据模型紧藕合了，因为新的类型转化是不可能的。

我们已经见过的大部分ORM解决方案都支持以用户自定义的策略来执行类型转化。这些通常称作转换器。例如，用户可以创建一种新的策略，用来把JDK类型Integer的一个属性持久化到VARCHAR列。Hibernate提供一种类似的、更强大的、称作定制映射类型的特性。

你首先要理解什么时候适合编写自己的定制映射类型，以及哪种Hibernate扩展点（extension point）与你相关。然后编写一些定制的映射类型并探讨选项。

5.3.1 考虑定制的映射类型

拿前面章节的Address类的映射作为组件来举例：

```
<component name="homeAddress" class="Address">

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

</component>
```

这个值类型映射很简单；用户自定义的新Java类型的所有属性都被映射到内建的SQL数据类型的单独的列。然而，也可以选择利用一个定制的映射类型把它当作一个简单的属性来映射：

```
<property name="homeAddress"
          type="auction.persistence.CustomAddressType">

    <column name="HOME_STREET"/>
    <column name="HOME_CITY"/>
    <column name="HOME_ZIPCODE"/>

</property>
```

这也可能是你第一次见到单个<property>元素内部嵌有几个<column>元素。我们正把Address值类型（它甚至被随处命名）和三个指定列之间转变和转换的任务转移给一个单独的类auction.persistence.CustomAddressType。现在这个类负责加载和保存这个属性。注意在领域模型实现中，Java代码都没有改变——homeAddress属性是Address类型。

当然，在这个例子中，使用定制的映射类型替换组件映射的好处是令人置疑的。只要在加载和保存这个对象时不需要特殊的转化，现在必须编写的CustomAddressType就是额外的工作。然而，你可能已经看到定制的映射类型提供了一个额外的缓冲区——当在长期运行中需要额外的转化时，可能派得上用场的东西。当然，定制的映射类型还有更好的使用案例，你很快就会看到。（在Hibernate社区网站上可以找到许多有用的Hibernate映射类型的例子。）

来看看用于创建定制的映射类型的Hibernate扩展点。

5.3.2　扩展点

Hibernate提供定义定制的映射类型时应用程序可能使用的几个接口。这些接口减少了创建新映射类型所涉及的工作，并使定制的类型免受Hibernate核心变化的影响。这帮助你轻松地升级Hibernate，并保留现有的定制映射类型。

扩展点如下：

- org.hibernate.usertype.UserType——基础的扩展点，用于多种情况。它为定制值类型实例的加载和存储提供基础的方法。
- org.hibernate.usertype.CompositeUserType——包含比基础的UserType更多方法的一个接口，通常把有关值类型类的内部信息公开给Hibernate，例如单独的属性。然后可以在Hibernate查询中引用这些属性。
- org.hibernate.usertype.UserCollectionType——很少被用来实现定制集合的接口。实现这个接口的定制映射类型不是在属性映射中声明，而是只对定制的集合映射有用。如果想要持久化一个非JDK的集合，并持久保持额外的语义时，就必须实现这个类型。第6章将讨论集合映射和这个扩展点。
- org.hibernate.usertype.EnhanceUserType——扩展了UserType并提供额外方法的接口，这些方法用来把值类型封送到XML表示法（或者从XML表示法中封送值类型），或者启用一个定制的映射类型，在标识符和辨别标志映射中使用。

- org.hibernate.usertype.UserVersionType——扩展了UserType并提供额外方法的接口，这些方法启用用于实体版本映射的定制映射类型。
- org.hibernate.usertype.ParameterizedType——一个有用的接口，可以与所有其他的接口合并，来提供配置设置——也就是说，元数据中定义的参数。例如，可以根据映射中的一个参数，编写一个知道如何把值转变为欧元或者美元的单个MoneyConverter。

现在要创建一些定制的映射类型。不应该认为这是一个没有必要的练习，即使你对内建的Hibernate映射类型很满意。依我们的经验，每一个复杂的应用程序都会有定制映射类型的许多好的使用案例。

5.3.3 定制映射类型的案例

Bid类定义了一个amount属性，Item类定义了一个initialPrice属性；这两者都是货币值。目前为止，我们只用了一个简单的BigDecimal来表示值，通过big_decimal映射到单个NUMERIC列。

假设你要在拍卖应用程序中支持多种货币，并且必须对这个（客户驱动的）变化重构现有的领域模型。实现这一变化的一种方法是把新的属性添加给Bid、Item:amountCurrency以及initialPriceCurrency。然后可以用内建的currency映射类型，把这些新属性映射到额外的VARCHAR列。我们希望你永远不要使用这种方法！

相反，应该创建一个封装了货币和金额的新MonetaryAmount类。注意这是领域模型的一个类，不依赖任何Hibernate接口：

```
public class MonetaryAmount implements Serializable {

    private final BigDecimal amount;
    private final Currency currency;

    public MonetaryAmount(BigDecimal amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public BigDecimal getAmount() { return amount; }

    public Currency getCurrency() { return currency; }
    public boolean equals(Object o) { ... }
    public int hashCode() { ...}

}
```

我们已经使MonetaryAmount成为了一个不可变的类。这在Java中是一个好的实践，因为它简化了代码。注意，必须实现equals()和hashCode()来完成类（此处没有什么特别的东西要考虑）。用这个新的MonetaryAmount取代Item中initialPrice属性的BigDecimal。可以并且应该把它用在任何持久化类中以及在业务逻辑中（例如，在账单系统中）所有其他的BigDecimal价格（例如Bid.amount）。

让我们通过这个新的MonetaryAmount类型，把Item的被重构的initialPrice属性映射到数据库。

5.3.4　创建 UserType

想象你正在使用一个用USD表示所有货币金额的遗留数据库。这个应用程序不再受限于单种货币(这是重构的重点)，但是它要花费数据库团队一些时间进行改变。当持久化MonetaryAmount对象时，需要把金额转化为USD。当从数据库加载时，把它转化回用户根据其偏爱所选择的货币类型。

创建一个实现Hibernate接口UserType的新MonetaryAmountUserType类。这是你的定制映射类型，如代码清单5-4所示。

代码清单5-4　给用USD为单位的货币金额定制的映射类型

```
public class MonetaryAmountUserType
        implements UserType {

    public int[] sqlTypes() {                                    ❶
        return new int[]{ Hibernate.BIG_DECIMAL.sqlType() };
    }

    public Class returnedClass() { return MonetaryAmount.class; }    ❷

    public boolean isMutable() { return false; }                 ❸

    public Object deepCopy(Object value) {  return value; }      ❹

    public Serializable disassemble(Object value)                ❺
        { return (Serializable) value; }

    public Object assemble(Serializable cached, Object owner)    ❻
        { return cached; }

    public Object replace(Object original,                       ❼
                          Object target,
                              Object owner)
        { return original; }

    public boolean equals(Object x, Object y) {                  ❽
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public int hashCode(Object x) {
        return x.hashCode();
    }

    public Object nullSafeGet(ResultSet resultSet,               ❾
                              String[] names,
                              Object owner)
            throws SQLException {

        BigDecimal valueInUSD = resultSet.getBigDecimal(names[0]);
        // Deferred check after first read
        if (resultSet.wasNull()) return null;
        Currency userCurrency = User.getPreferences().getCurrency();
        MonetaryAmount amount = new MonetaryAmount(valueInUSD, "USD");
```

```
        return amount.convertTo(userCurrency);
    }

    public void nullSafeSet(PreparedStatement statement,                    ⑩
                            Object value,
                            int index)
            throws HibernateException, SQLException {

        if (value == null) {
          statement.setNull(index, Hibernate.BIG_DECIMAL.sqlType());
        } else {
            MonetaryAmount anyCurrency = (MonetaryAmount)value;
            MonetaryAmount amountInUSD =
              MonetaryAmount.convert( anyCurrency,
                                      Currency.getInstance("USD") );
            statement.setBigDecimal(index, amountInUSD.getAmount ());
        }
    }
}
```

❶ sqlTypes()方法告诉Hibernate要使用什么SQL列类型生成DDL模式。注意，这个方法返回类型代码的一个数组。UserType可以把单个属性映射到多个列，但是这个遗留的数据模型只有单个数字化的列。通过使用Hibernate.BIG_DECIMAL.sqlType()方法，你让Hibernate为指定的数据库方言确定正确的SQL数据类型。另一种方法是从java.sql.Types中返回常量。

❷ returnedClass()方法告诉Hibernate这个UserType映射哪些Java值类型。

❸ Hibernate可以对像这样的不可变类型进行一些微小的性能优化，例如，在脏检查期间比较快照的时候。isMutable()方法告诉Hibernate这个类型是不可变的。

❹ UserType首先也部分负责创建值的快照。因为MonetaryAmount是不可变的类，deepCopy()方法返回它的实参。至于可变的类型，则需要返回实参的一个副本，当作快照值使用。

❺ 当Hibernate把MonetaryAmount放进一个二级高速缓存时，调用disassemble()方法。就像稍后要介绍的，这是一种以序列化的形式保存信息的数据的高速缓存。

❻ assemble()方法与分解正好相反：它把被高速缓存的数据转变为MonetaryAmount的一个实例。如你所见，这两个子程序的实现对于不可变的类型都很容易。

❼ 实现replace()来处理脱管对象状态的合并。如本书稍后要介绍的，合并的过程涉及源对象和目标对象，它们的状态必须合并。对于不可变的值类型，再次返回第一个实参。对于可变的类型，至少返回第一个实参的深复制（deep copy）。对于带有组件字段的可变类型，你或许想要应用一个递归的合并子程序。

❽ UserType负责脏检查属性值。equals()方法把当前的属性值与之前的快照进行比较，确定属性是否为脏，并且必须被保存到数据库。两个相等的值类型实例的hashCode()必须相同。我们通常把这个方法委托给实际的值类型类——在这个例子中，是给定的MonetaryAmount对象的hashCode()方法。

❾ nullSafeGet()方法从JDBC的ResultSet获取属性值。如果要它用于转化，你也可以访问组件的所有者。所有数据库值都以USD为单位，因此把它转化为用户目前已经依其偏爱设置的

货币类型。（注意，实现这种转化和偏爱的处理取决于你。）

❿ nullSafeSet()方法把属性值写到JDBC的PreparedStatement。这个方法采用所设置的任何货币类型，并在保存之前把它转化为一个简单的BigDecimal USD金额。

现在映射Item的initialPrice属性如下：

```
<property name="initialPrice"
          column="INITIAL_PRICE"
          type="persistence.MonetaryAmountUserType"/>
```

注意你把定制的用户类型放进了persistence包；它是应用程序的持久层的一部分，而不是领域模型或者业务层的一部分。

为了在注解中使用定制类型，必须添加一个Hibernate扩展：

```
@org.hibernate.annotations.Type(
    type = " persistence.MonetaryAmountUserType"
)
@Column(name = "INITIAL_PRICE")
private MonetaryAmount initialPrice;
```

这是UserType可以执行的一种最简单的转化。更加复杂的事情也有可能。定制的映射类型可以执行验证；它可以把数据读取和写入到一个LDAP目录；它甚至可以从不同的数据库获取持久化对象。你主要受限于你的想象力。

在现实应用中，我们更喜欢在数据库中表示货币金额的金额和货币类型，尤其当Schema不是遗留的，却又可以被定义（或者快速更新）的时候。假设现在有两个列可用，并且不用多少转化就可以保存MonetaryAmount。第一种方法可能还是简单的<component>映射。然而，让我们试着用定制的映射类型来解决。

（不编写新的定制类型，而是试着把前一个例子改成两个列。这一点你可以不改变Java领域模型类就能完成——只有转换器需要为这个新需求和在映射中具名的新增列而更新。）

简单的UserType实现的缺点在于，Hibernate不知道有关MonetaryAmount内部单独属性的任何信息。它所知的只是定制类型类和列名称。Hibernate查询引擎（稍后将更深入讨论）不知道如何查询amount或者特定的currency。

如果需要Hibernate查询的全部功能，就编写CompositeUserType。这个（稍微有点复杂的）接口把MonetaryAmount的属性公开给Hibernate查询。现在再次用这个更灵活的定制接口把它映射到两个列，实际上生成了组件映射的等价物。

5.3.5　创建 CompositeUserType

为了示范定制映射类型的灵活性，不要对MonetaryAmount类（和其他持久化类）作任何改变——只改变定制映射类型，如代码清单5-5所示。

代码清单5-5　给新的数据库Schema中的货币金额定制映射类型

```
public class MonetaryAmountCompositeUserType
        implements CompositeUserType {

    // public int[] sqlTypes()...          ❶
```

```
public Class returnedClass...
public boolean isMutable...
public Object deepCopy...
public Serializable disassemble...
public Object assemble...
public Object replace...
public boolean equals...
public int hashCode...

public Object nullSafeGet(ResultSet resultSet,                    ❷
                          String[] names,
                          SessionImplementor session,
                          Object owner)
        throws SQLException {

    BigDecimal value = resultSet.getBigDecimal( names[0] );
    if (resultSet.wasNull()) return null;
    Currency currency =
        Currency.getInstance(resultSet.getString( names[1] ) );
    return new MonetaryAmount(value, currency);
}

public void nullSafeSet(PreparedStatement statement,              ❸
                        Object value,
                        int index,
                        SessionImplementor session)
        throws SQLException {
    if (value==null) {
      statement.setNull(index, Hibernate.BIG_DECIMAL.sqlType());
      statement.setNull(index+1, Hibernate.CURRENCY.sqlType());
    } else {
        MonetaryAmount amount = (MonetaryAmount) value;
        String currencyCode =
                    amount.getCurrency().getCurrencyCode();
        statement.setBigDecimal( index, amount.getAmount() );
        statement.setString( index+1, currencyCode );
    }
}

public String[] getPropertyNames() {                              ❹
    return new String[] { "amount", "currency" };
}

public Type[] getPropertyTypes() {                                ❺
    return new Type[] { Hibernate.BIG_DECIMAL,
                        Hibernate.CURRENCY };
}

public Object getPropertyValue(Object component, int property) {  ❻
    MonetaryAmount monetaryAmount = (MonetaryAmount) component;
    if (property == 0)
        return monetaryAmount.getAmount();
    else
        return monetaryAmount.getCurrency();
```

```
    }
    public void setPropertyValue(Object component,          ❼
                                 int property,
                                 Object value) {
        throw new
          UnsupportedOperationException("Immutable MonetaryAmount!");
    }
}
```

❶ CompositeUserType接口需要与你前面创建的UserType相同的内部处理方法。但是不再需要sqlTypes()方法。

❷ 现在加载一个值很简单：把结果集中的两个列值转化为新MonetaryAmount实例中的两个属性值。

❸ 保存值涉及在预编译的语句中设置两个参数。

❹ CompositeUserType通过getPropertyNames()公开值类型的属性。

❺ 每个属性都有它们自己的类型，就像getPropertyTypes()定义的那样。现在SQL列的类型对于这个方法是隐式的。

❻ getPropertyValue()方法返回MonetaryAmount的单个属性的值。

❼ setPropertyValue()方法设置MonetaryAmount的单个属性的值。

initialPrice属性现在映射到两个列，因此在映射文件中要对两者都进行声明。第一列保存值；第二列保存MonetaryAmount的货币类型：

```
<property name="initialPrice"
          type="persistence.MonetaryAmountCompositeUserType">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CURRENCY"/>
</property>
```

如果通过注解映射Item，就必须给这个属性声明几个列。无法多次使用javax.persistence.Column注解，因此需要一个特定于Hibernate的新注解：

```
@org.hibernate.annotations.Type(
    type = "persistence.MonetaryAmountUserType"
)
@org.hibernate.annotations.Columns(columns = {
    @Column(name="INITIAL_PRICE"),
    @Column(name="INITIAL_PRICE_CURRENCY", length = 2)
})
private MonetaryAmount initialPrice;
```

在Hibernate查询中，现在可以引用定制类型的amount和currency属性，即使它们没有作为单独的属性出现在映射文件的任何地方：

```
from Item i
where i.initialPrice.amount > 100.0
  and i.initialPrice.currency = 'AUD'
```

我们已经用这个新的定制复合类型扩展了Java对象模型和SQL数据库模式之间的缓冲区。这两种表示法现在都更经得起变化。注意列的数目与选择UserType或者CompositeUserType无

关——只要愿意把值类型属性公开给Hibernate查询。

对于所有定制的映射类型来说，参数化是一项很有帮助的特性。

5.3.6 参数化定制类型

假设你再次面临最初的问题：当把金额保存到数据库中时把它转化为一种不同的货币类型。这些问题经常比一般的转化更为微妙；例如，可以在一些表中保存美元，而在其他表中保存欧元。你仍然想要给它编写单个定制的映射类型，使它可以进行任意的转化。如果把ParameterizedType接口添加到UserType或者CompositeUserType类，这是可能的：

```
public class MonetaryAmountConversionType
    implements UserType, ParameterizedType {

    // Configuration parameter
    private Currency convertTo;

    public void setParameterValues(Properties parameters) {
       this.convertTo = Currency.getInstance(
                            parameters.getProperty("convertTo")
                        );
    }

    // ... Housekeeping methods

    public Object nullSafeGet(ResultSet resultSet,
                              String[] names,
                              SessionImplementor session,
                              Object owner)
            throws SQLException {

       BigDecimal value = resultSet.getBigDecimal( names[0] );
       if (resultSet.wasNull()) return null;
       // When loading, take the currency from the database
       Currency currency = Currency.getInstance(
                            resultSet.getString( names[1] )
                            );
       return new MonetaryAmount(value, currency);
    }

    public void nullSafeSet(PreparedStatement statement,
                            Object value,
                            int index,
                            SessionImplementor session)
            throws SQLException {

       if (value==null) {
           statement.setNull(index, Types.NUMERIC);
       } else {
           MonetaryAmount amount = (MonetaryAmount) value;
           // When storing, convert the amount to the
           // currency this converter was parameterized with
           MonetaryAmount dbAmount =
                MonetaryAmount.convert(amount, convertTo);
           statement.setBigDecimal( index, dbAmount.getAmount() );
```

```
                statement.setString( index+1,
                                     dbAmount.getCurrencyCode() );
            }
        }
    }
```

我们在这个例子中省略了通常强制的内部处理方法。另一种重要的方法是ParameterizedType接口的setParameterValues()。Hibernate在启动时用convertTo参数调用这个方法来初始化这个类。保存MonetaryAmount时，nullSafeSet()方法就使用这个设置转化为目标货币类型。nullSafeGet()方法采用数据库中出现的这种货币，并把它留给客户去处理被加载的MonetaryAmount的货币类型（这种不对称的实现自然不是最佳办法。）

应用定制映射类型时，现在必须在映射文件中设置配置参数。一种简单的解决方案是属性中嵌套着的<type>映射：

```
<property name="initialPrice">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CUR"/>
    <type name="persistence.MonetaryAmountConversionType">
        <param name="convertTo">USD</param>
    </type>
</property>
```

然而，如果在你的领域模型中有许多个货币金额时，这样很不方便，并且需要复制。一种更好的策略是使用类型的单个定义，包括所有参数，用一个可以在所有映射中重用的唯一名称。通过单个的<typedef>元素（你也可以不通过参数使用它）完成这一点：

```
<typedef class="persistence.MonetaryAmountConversionType"
         name="monetary_amount_usd">
    <param name="convertTo">USD</param>
</typedef>

<typedef class="persistence.MonetaryAmountConversionType"
         name="monetary_amount_eur">
    <param name="convertTo">EUR</param>
</typedef>
```

这里介绍的是，把带有一些参数的定制映射类型绑定到名称monetary_amount_usd和monetary_amount_eur上。这个定义可以放在映射文件中的任何位置；它是<hibernate-mapping>的一个子元素（如本书前面提到过的，较大的应用程序经常有一个或者几个没有类映射的MyCustomTypes.hbm.xml文件）。利用Hibernate扩展，可以在注解中定义带有参数的具名定制类型：

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_usd",
        typeClass = persistence.MonetaryAmountConversionType.class,
        parameters = { @Parameter(name="convertTo", value="USD") }
    ),
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_eur",
        typeClass = persistence.MonetaryAmountConversionType.class,
```

```
            parameters = { @Parameter(name="convertTo", value="EUR") }
        )
    })
```

这个注解元数据也是全局的，因此可以放在任何Java类声明之外（就在import语句之后），或者在一个单独的文件package-info.java中，如2.2.1节中所述。这个系统中一个很好的位置则是persistence包中的package-info.java文件。

在XML映射文件和注解映射中，现在引用被定义的类型名称，而不是定制类型完全限定的类名：

```
<property name="initialPrice"
          type="monetary_amount_usd">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CUR"/>
</property>
```

```
@org.hibernate.annotations.Type(type = "monetary_amount_eur")
@org.hibernate.annotations.Columns({
  @Column(name = "BID_AMOUNT"),
  @Column(name = "BID_AMOUNT_CUR")
})
private MonetaryAmount bidAmount;
```

来看定制映射类型的一个不同的、极为重要的应用程序。这个类型安全的枚举（enumeration）设计模式几乎在所有的应用程序中都可以见到。

5.3.7 映射枚举

枚举类型是一个常见的Java惯用语，其中类有着（小）数量不变的不可变实例。在CaveatEmptor中，这可以应用到信用卡上：例如，要表达用户可以输入及应用程序提供的可能的类型（Mastercard、Visa等）。或者，可以枚举用户会在Comment中提交的有关一次特定拍卖的可能等级。

在更早版本的JDK中，你必须自己实现这些类（我们称它们为CreditCardType和Rating），遵循类型安全的枚举模式。如果没有JDK 5.0，这也仍然是实现它的一种好办法；在Hibernate社区网站上可以找到这种模式和兼容的定制映射类型。

1. 在JDK 5.0中使用枚举

如果使用JDK 5.0，可以给类型安全的枚举使用内建的语言支持。例如，Rating类看起来如下：

```
package auction.model;

public enum Rating {
    EXCELLENT, OK, BAD
}
```

Comment类有这个类型的一个属性：

```
public class Comment {
    ...
    private Rating rating;
    private Item auction;
    ...
}
```

下面是在应用程序代码中使用枚举的方式：

```
Comment goodComment =
    new Comment(Rating.EXCELLENT, thisAuction);
```

现在必须持久化这个Comment实例和它的Rating。一种方法是使用枚举的实际名称，并把它保存到COMMENTS表中的一个VARCHAR列。这个RATING列将包含EXCELLENT、OK或者BAD，具体情况取决于给定的Rating。

我们来编写Hibernate UserType，它可以加载和保存VARCHAR支持的枚举，例如Rating。

2. 编写定制的枚举处理程序

现在要介绍的不是最基础的UserType接口，而是EnhanceUserType接口。这个接口允许在XML表示模式中使用Comment实体，而不仅仅作为一个POJO（请见3.4节中对数据表示法的讨论）。此外，由于额外的ParameterizedType接口，你将要编写的实现可以支持任何VARCHAR支持的枚举，而不仅仅是Rating。

看看代码清单5-6中的代码。

代码清单5-6　给string支持的枚举定制映射类型

```
public class StringEnumUserType
        implements EnhancedUserType, ParameterizedType {

    private Class<Enum> enumClass;

    public void setParameterValues(Properties parameters) {   ❶
        String enumClassName =
            parameters.getProperty("enumClassname");
        try {
            enumClass = ReflectHelper.classForName(enumClassName);
        } catch (ClassNotFoundException cnfe) {
            throw new
              HibernateException("Enum class not found", cnfe);
        }
    }

    public Class returnedClass() {   ❷
        return enumClass;
    }

    public int[] sqlTypes() {   ❸
        return new int[] { Hibernate.STRING.sqlType() };
    }

    public boolean isMutable...   ❹
    public Object deepCopy...
    public Serializable disassemble...
    public Object replace...
    public Object assemble...
    public boolean equals...
    public int hashCode...

    public Object fromXMLString(String xmlValue) {   ❺
        return Enum.valueOf(enumClass, xmlValue);
```

```
}

public String objectToSQLString(Object value) {
    return '\'' + ( (Enum) value ).name() + '\'';
}

public String toXMLString(Object value) {
    return ( (Enum) value ).name();
}

public Object nullSafeGet(ResultSet rs,                    ❻
                          String[] names,
                          Object owner)
        throws SQLException {
    String name = rs.getString( names[0] );
    return rs.wasNull() ? null : Enum.valueOf(enumClass, name);   ❼
}

public void nullSafeSet(PreparedStatement st,
                        Object value,
                        int index)
        throws SQLException {
    if (value == null) {
        st.setNull(index, Hibernate.STRING.sqlType());
    } else {
        st.setString( index, ( (Enum) value ).name() );
    }
}
```

❶ 这个定制映射类型的配置参数是用于枚举类的名称，如Rating。

❷ 它也是从这个方法中返回的类。

❸ 数据库表中需要的单个VARCHAR列。通过让Hibernate决定SQL数据类型使它保持可移植。

❹ 这些是不可变类型一般的内部管理方法。

❺ 下列三种方法是EnchanceUserType的一部分，并用于XML封送。

❻ 加载枚举时，从数据库中获取它的名称，并创建一个实例。

❼ 保存枚举时，保存它的名称。

接下来，用这个新定制的类型映射rating属性。

3. 用XML和注解映射枚举

在XML映射中，首先创建一个定制的类型定义：

```
<typedef class="persistence.StringEnumUserType"
         name="rating">
    <param name="enumClassname">auction.model.Rating</param>
</typedef>
```

现在可以在Comment类映射中使用名为rating的类型：

```
<property  name="rating"
           column="RATING"
           type="rating"
           not-null="true"
```

```
            update="false"
            access="field"/>
```

因为等级是不可变的，你把它映射为update="false"，并启用直接的字段访问（不可变的属性没有设置方法）。如果除了Comment之外的其他类有Rating属性，就再次使用定义好的定制映射类型。

注解中这个定制映射类型的定义和声明看起来与你在前一节中所做的一样。

另一方面，可以依赖Java Persistence提供程序持久化枚举。如果在类型java.lang.Enum中的一个被注解的实体类中有一个属性（例如Comment中的rating），并且它没有标记为@Transient或者transient（Java关键字），Hibernate JPA实现必须毫无怨言地保持该属性为开箱即用；它有一个处理这个的内建类型。这个内建的映射类型在数据库中必须默认为枚举的一种表示法。两种最常见的选择是字符串表示法，就像通过定制类型或者有序表示法与原生的Hibernate实现一样。有序表示法保存选中的枚举选项的位置：例如，1为EXCELLENT，2为OK，3为BAD。数据库列也默认为一个数字列。可以在属性上通过Enumerated注解改变这个默认的枚举映射：

```
public class Comment {
    ...

    @Enumerated(EnumType.STRING)
    @Column(name = "RATING", nullable = false, updatable = false)
    private Rating rating;
    ...
}
```

现在已经切换到了基于字符串的表示法，实际上是与你的定制类型可以读写的相同的表示法。也可以使用JPA XML描述符：

```
<entity class="auction.model.Item" access="PROPERTY">
  <attributes>
    ...
    <basic name="rating">
        <column name="RATING" nullable="false" updatable="false"/>
        <enumerated>STRING</enumerated>
    </basic>
  </attributes>
</entity>
```

你可能会（理直气壮地）问，当Hibernate作为Java Persistence提供程序，可以显然开箱即用地持久化和加载枚举时，为什么必须给枚举编写自己的定制映射类型。这个秘密在于Hibernate Annotations包括了几个定制映射类型，该定制映射类型实现由Java Persistence定义的行为。可以在XML映射中使用这些定制类型；但是它们不是对用户友好的（需要许多参数），并且不是为这一目的而编写的。可以检查源代码（例如Hibernate Annotations中的org.hibernate.type.EnumType）来了解它们的参数，并决定是否要在XML中直接使用它们。

4．用定制映射类型查询

你可能遇到的进一步问题是在Hibernate查询中使用被枚举的类型。例如，考虑获取所有等级为“bad”的评语的HQL格式的下列查询：

```
Query q =
    session.createQuery(
      "from Comment c where c.rating = auction.model.Rating.BAD"
    );
```

如果把枚举持久化为字符串，虽然这个查询也有效（查询解析器用枚举值作为常量），但是如果选择了有序表示法，它就不起作用了。必须使用一个绑定参数，并对比较式通过编程来设置等级值：

```
Query q =
    session.createQuery("from Comment c where c.rating = :rating");

Properties params = new Properties();
params.put("enumClassname",
           "auction.model.Rating");

q.setParameter("rating", Rating.BAD,
               Hibernate.custom(StringEnumUserType.class, params)
              );
```

本例的最后一行使用静态的辅助方法Hibernate.custom()，来把定制的映射类型转化为Hibernate Type；这是告诉Hibernate有关枚举映射，以及如何处理Rating.BAD值的一种简单方法。注意，你还必须告诉Hibernate有关被参数化的类型可能需要的任何初始化属性。

不幸的是，Java Persistence中没有用于任意的和定制的查询参数的API，因此必须退回到Hibernate Session API，并创建一个Hibernate Query对象。

建议你要非常熟悉Hibernate类型系统，并把定制映射类型的创建当作一项基本技能——它在用Hibernate或者JPA开发的每个应用程序中都将非常有用。

5.4 小结

本章介绍了如何利用4种基础的继承映射策略，来把实体的继承层次结构映射到数据库：带有隐式多态的每个具体类一张表、带有联合的每个具体类一张表、每个类层次结构一张表和每个子类一张标准化表的策略。你已经见到了如何给一个特定的层次结构混合这些策略，以及每一种策略什么时候最适合。

我们也详细阐述了有关Hibernate实体和值类型的区别，以及Hibernate映射类型系统如何工作。你用了各种内建的类型，并利用如UserType和ParameterizedType的Hibernate扩展点编写自己的定制类型。

表5-5概括了原生的Hibernate特性和Java Persistence的对比。

表5-5 第5章的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
支持4种继承映射策略。混合继承策略成为可能	4种继承映射策略被标准化；在一个层次结构中混合策略被认为是不可移植的。JPA兼容的提供程序只需要每个类层次结构一张表和每个子类一张表
持久化的基类型可以是抽象类或者接口（只包含属性访问方法）	持久化的基类型可以是抽象类；被映射的接口被认为是不可移植的

（续）

Hibernate Core	Java Persistence和EJB 3.0
为值类型属性提供灵活的内建映射类型和转换器	有映射类型的自动侦测，包含对暂时的和枚举映射类型的标准覆盖。Hibernate扩展注解用于任何定制映射类型声明
强大的可扩展的类型系统	标准需要用于枚举、LOB和许多其他值类型的内建类型，必须在原生的Hibernate中为它们编写或者应用一个定制的映射类型

第6章将介绍集合映射，并讨论如何处理值类型对象的集合（例如，`String`的集合）和包含对实体实例引用的集合。