第8章 遗留数据库和定制SQL

本章内容

- 遗留数据库整合和灵活映射
- SQL语句的定制
- 定制DDL改善SQL Schema

本章介绍的许多例子都是关于“困难”映射的。可能你第一次创建映射有问题时，是在使用一个不能被修改的遗留数据库Schema的时候。我们讨论在这样的场景中会遇到的典型问题，以及如何扭曲映射元数据，而不是改变应用程序或者数据库Schema。

我们还介绍如何覆盖Hibernate自动生成的SQL。这包括SQL查询、DML（创建、更新、删除）操作，以及Hibernate的自动DDL生成特性。你会看到如何映射存储过程和用户自定义的SQL函数，以及如何在数据库Schema中应用正确的完整性规则。如果你的数据库管理员需要完全的掌控，本节的内容将会特别有用（或者如果你是个数据库管理员，并想在SQL级优化Hibernate的话）。

如你所见，本章的主题很广泛，不必马上把它们全部读完。可以把本章的大部分内容当作参考资料，遇到特定的问题时再回到这里。

8.1 整合遗留数据库

我们希望本节能够涵盖处理现有的遗留数据库或者（这通常是一回事）奇怪的或者损坏的Schema时可能遇到的所有问题。然而，如果你的开发过程是自顶向下的，则可能想跳过本节。此外，建议你在尝试反向工程复杂的遗留Schema之前，先阅读有关类、集合和关联映射的所有章节。

必须提醒你：当应用程序继承现有的遗留数据库Schema时，通常应该尽可能地对现有的Schema少做一些改变。你对Schema所做的每一处改变都可能破坏访问数据库的其他现有应用程序。现有数据的迁移可能很昂贵，这也是你要考虑的东西。一般来说，构建一个新的应用程序不可能不对现有的数据模型做任何改变——新应用程序通常意味着额外的业务需求，自然需要数据库Schema的演变。

因此要考虑两种类型的问题：与改变业务需求相关的问题（不改变Schema一般无法解决）和只与你希望如何在新应用程序中表示相同业务问题相关的问题（这些问题虽不总是但大多数时候可以不用改变数据库Schema而解决）。很显然，第一种问题通常只要看看逻辑数据模型就可以看

见。第二种则更经常与逻辑数据模型实现为物理数据库Schema相关。

如果你接受这种观点，就会看到需要Schema改变的种种问题就是那些需要增加新实体、重构现有的实体、把新属性增加到现有的实体，以及修改实体之间的关联。这些可以不用改变Schema而解决的问题，通常涉及对特定实体不方便的表（或者列）的定义。本节将集中讨论这种问题。

假设你已经试过用Hibernate工具箱对现有的Schema进行反向工程，如2.3节中所述。接下来几节讨论的概念和解决方案假设你已经有基础的ORM，并且需要做些额外的变化使它生效。另一种方法，可以试着完全用手工编写映射，而不用反向工程工具。

从最明显的问题（遗留主键）开始。

8.1.1　处理主键

如前所述，我们认为自然主键可能是个馊主意。自然键经常使得在业务需求改变时很难重构数据模型。在极端的情况下，它们甚至可能影响性能。不幸的是，许多遗留Schema大量使用（自然的）复合键，并且由于我们反对使用复合键的缘故，可能很难改变遗留Schema来使用非复合的自然键或者代理键。

因此，Hibernate支持自然键的使用。如果自然键是一个复合键，支持就是通过`<composite-id>`映射。我们来映射复合的和非复合的这两种自然主键。

1．映射自然键

如果你在遗留Schema中遇到USERS表，USERNAME可能就是真正的主键。在这种情况下，就没有自动生成的代理标识符了。反之，你启用assigned标识符生成器策略，告诉Hibernate标识符是在对象被保存之前由应用分配的一个自然键：

```
<class name="User" table="USERS">
    <id name="username" column="USERNAME" length="16">
        <generator class="assigned"/>
    </id>

    ...
</class>
```

保存新User的代码如下：

```
User user = new User();
user.setUsername("johndoe"); // Assign a primary key value
user.setFirstname("John");
user.setLastname("Doe");
session.saveOrUpdate(user); // Will result in an INSERT
// System.out.println( session.getIdentifier(user) );
session.flush();
```

Hibernate如何知道saveOrUpdate()需要INSERT而不是UPDATE呢？它不知道，因此需要一个技巧：Hibernate在USERS表中查询指定的用户名。如果找到，Hibernate就更新行；如果没找到，就要插入新行。这当然不是最好的解决方案，因为它在数据库上触发了额外的瞬时干扰。

几种策略避免SELECT：

- 把`<version>`或者`<timestamp>`映射和一个属性添加到实体。Hibernate在内部通过乐观并

发控制来管理这两个值（本书稍后会讨论）。作为一项附带作用，空的时间戳（或者0或者NULL）版本表明实例是新的，并且必须插入而不是更新。

- 实现Hibernate Interceptor，并把它钩入到Session里面。这个扩展接口允许你实现方法isTransient()，它包含着你区分旧对象和新对象时可能需要的任何定制过程。

另一方面，如果喜欢显式地使用save()和update()而不是saveOrUpdate()，Hibernate就不必区分瞬时实例和脱管实例——你通过选择要调用的正确方法来进行。（实际上，这就是始终不用saveOrUpdate()的唯一原因。）

用JPA注解映射自然主键很简单：

```
@Id
private String username;
```

如果没有声明标识符生成器，Hibernate就会假设它必须应用一般的select-to-determine-state-unless-versioned[①]策略，并期待应用程序负责分配主键值。可以通过扩展包含拦截器的应用或者添加版本控制属性（版本号或者时间戳），再次避免SELECT。

复合的自然键按同样的思路进行扩展。

2. 映射复合自然键

假设USERS表的主键由USERNAME和DEPARTMENT_NR组成。可以添加一个具名departmentNr的属性到User类，并创建下列映射：

```
<class name="User" table="USERS">
    <composite-id>
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentNr"
                      column="DEPARTMENT_NR"/>
    </composite-id>

    ...
</class>
```

保存新User的代码看起来如下：

```
User user = new User();

// Assign a primary key value
user.setUsername("johndoe");
user.setDepartmentNr(42);

// Set property values
user.setFirstname("John");
user.setLastname("Doe");
```

① 程序分配的标识符。如果需要应用程序分配标识符（而非Hibernate来生成），可使用assigned生成器。该生成器会使用已经分配给对象的标识符属性的标识符值。它使用自然键作为主键。这是没有指定<generator>元素时的默认行为。当选择assigned生成器时，除非有version或timestamp属性，或者你定义了Interceptor.isUnsaved()，否则需要让Hibernate使用unsavedvalue="undefined"，强制Hibernate查询数据库来确定一个实例是瞬时的，还是脱管的。——译者注

```
session.saveOrUpdate(user);
session.flush();
```

同样要记住，Hibernate执行一个SELECT来确定saveOrUpdate()应该做什么——除非你启用版本控制或者定制的Interceptor。但是调用load()或者get()时，可以（应该）使用什么对象作为标识符呢？例如，可能使用User类的一个实例：

```
User user = new User();

// Assign a primary key value
user.setUsername("johndoe");
user.setDepartmentNr(42);

// Load the persistent state into user
session.load(User.class, user);
```

在这个代码片段中，User表现得就像它自己的标识符类。定义一个只声明键属性的单独复合标识符类则更为优雅。调用这个类UserId：

```
public class UserId implements Serializable {
    private String username;
    private Integer departmentNr;

    public UserId(String username, Integer departmentNr) {
        this.username = username;
        this.departmentNr = departmentNr;
    }

    // Getters...

    public int hashCode() {
        int result;
            result = username.hashCode();
            result = 29 * result + departmentNr.hashCode();
            return result;
    }

    public boolean equals(Object other) {
        if (other==null) return false;
        if ( !(other instanceof UserId) ) return false;
        UserId that = (UserId) other;
        return this.username.equals(that.username) &&
            this.departmentNr.equals(that.departmentNr);
    }
}
```

正确地实现equals()和hashCode()很重要，因为Hibernate的高速缓存查找依赖这些方法。标识符类还需要实现Serializable接口。

现在从User和UserId属性中移除username和departmentNr属性。创建下列映射：

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

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentNr"
```

```
                        column="DEPARTMENT_NR"/>
    </composite-id>

    ...
</class>
```

用下列代码保存User的新实例：

```
UserId id = new UserId("johndoe", 42);

User user = new User();

// Assign a primary key value
user.setUserId(id);

// Set property values
user.setFirstname("John");
user.setLastname("Doe");

session.saveOrUpdate(user);
session.flush();
```

要使saveOrUpdate()起作用还是需要SELECT。下列代码展现了如何加载实例：

```
UserId id = new UserId("johndoe", 42);

User user = (User) session.load(User.class, id);
```

现在，假设DEPARTMENT_NR是一个引用DEPARTMENT表的外键，并且你希望在Java领域模型中把这个关联表示为多对一的关联。

3. 复合主键中的外键

建议你映射一个也是复合主键一部分的外键列，通过一般的<many-to-one>元素，并用insert="false" update="false"禁用该列的任何Hibernate插入或者更新，如下：

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

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentId"
                      column="DEPARTMENT_ID"/>
    </composite-id>

    <many-to-one name="department"
                 class="Department"
                 column="DEPARTMENT_ID"
                 insert="false" update="false"/>
    ...
</class>
```

现在Hibernate在更新或者插入User时忽略department属性，但是你当然可以用johndoe.getDepartment()读取它。User和Department之间的关系现在通过UserId复合键类的departmentId属性得到托管：

```
UserId id = new UserId("johndoe", department.getId() );

User user = new User();

// Assign a primary key value
user.setUserId(id);
// Set property values
user.setFirstname("John");
user.setLastname("Doe");
user.setDepartment(department);

session.saveOrUpdate(user);
session.flush();
```

只有Department的标识符值对持久化状态有些影响；为了一致，调用setDepartment(department)；否则，你必须在清除之后从数据库中刷新对象来获取部门集。（在实际应用中，你可以把所有这些细节移到复合标识符类的构造器里面。）

另一种方法是<key-many-to-one>：

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

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-many-to-one name="department"
                         class="Department"
                         column="DEPARTMENT_ID"/>
    </composite-id>
    ...
</class>
```

然而，在复合标识符类中有关联通常并不方便，因此除非特殊情况，否则不推荐这种方法。<key-many-to-one>构造在查询方面也有限制：你无法限制一个跨<key-many-to-one>联结的HQL或者Criteria的查询结果（虽然这些特性可能将在以后的Hibernate版本中得以实现）。

4．复合主键的外键

由于USERS有一个复合主键，任何引用外键也是复合的。例如，从Item到User（卖主）的关联现在通过复合外键映射。

通过下列从Item到User的关联映射，Hibernate可以从Java代码中隐藏这一细节：

```
<many-to-one name="seller" class="User">
    <column name="USERNAME"/>
    <column name="DEPARTMENT_ID"/>
</many-to-one>
```

User类所拥有的任何集合也都有复合外键——例如，反向关联，被这位用户出售的items：

```
<set name="itemsForAuction" inverse="true">
    <key>
        <column name="USERNAME"/>
        <column name="DEPARTMENT_ID"/>
    </key>
    <one-to-many class="Item"/>
</set>
```

注意，列的排列顺序很重要，应该与它们在User主键映射的<composite-id>元素中出现的顺序一致。

这样就完成了对Hibernate中基础复合键映射技术的讨论。利用注解映射复合键也几乎相同，但是同往常一样，微小的差别很重要。

5. 利用注解的复合键

JPA规范涵盖了处理复合键的策略。你有3种选择：

- 把标识符属性封装在一个单独的类里面，并把它标识为@Embeddable，就像一般的组件一样。在实体类中包括这个组件类型的一个属性，并利用@Id给一个应用分配的策略映射它。
- 把标识符属性封装在一个没有任何注解的单独类里面。在实体类中包括这个类型的一个属性，并用@EmbeddedId映射它。
- 把标识符属性封装在单个的类里面。现在——这与你通常在原生的Hibernate中所做的不同——复制实体类中的所有标识符属性。然后，用@IdClass注解实体类，并指定被封装的标识符类的名称。

第一种选择很简单。需要使来自前一节的UserId类变成是可嵌入的：

```
@Embeddable
public class UserId implements Serializable {
    private String username;
    private String departmentNr;
    ...
}
```

至于所有的组件映射，你可以在这个类的字段（或者获取方法）中定义额外的映射属性。为了映射User的复合键，通过省略@GenerateValue注解而对分配的应用设置生成策略：

```
@Id
@AttributeOverrides({
    @AttributeOverride(name   = "username",
                       column = @Column(name="USERNAME") ),
    @AttributeOverride(name   = "departmentNr",
                       column = @Column(name="DEP_NR") )
})
private UserId userId;
```

就像你在本书前面对一般的组件映射所做的那样，如果喜欢，可以覆盖组件类的特定属性映射。

第二种复合键映射策略不要求你给UserId主键类作标记。因此，这个类不需要@Embeddable和其他注解。在自己的实体中，你通过@EmbeddedId映射复合的标识符属性，仍然通过可选的覆盖：

```
@EmbeddedId
@AttributeOverrides({
    @AttributeOverride(name   = "username",
                       column = @Column(name="USERNAME") ),
    @AttributeOverride(name   = "departmentNr",
```

```
                        column = @Column(name="DEP_NR") )
})
private UserId userId;
```

在JPA XML描述符中，这个映射看起来如下：

```
<embeddable class="auction.model.UserId" access ="PROPERTY">
    <attributes>
        <basic name="username">
            <column name="UNAME"/>
        </basic>
        <basic name="departmentNr">
            <column name="DEPARTMENT_NR"/>
        </basic>
    </attributes>
</embeddable>

<entity class="auction.model.User" access="FIELD">
    <attributes>
        <embedded-id name="userId">
            <attribute-override name="username">
                <column name="USERNAME"/>
            </attribute-override>
            <attribute-override name="departmentNr">
                <column name="DEP_NR"/>
            </attribute-override>
        </embedded-id>
        ...
    </attributes>
</entity>
```

第三种复合键映射策略则更难理解一点，尤其对于那些经验丰富的Hibernate用户而言。首先，把所有标识符属性封装在一个单独的类里面——如前一种策略一样，这个类不需要额外的注解。现在复制实体类中的所有标识符属性：

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

    @Id
    private String username;

    @Id
    private String departmentNr;

    // Accessor methods, etc.
    ...
}
```

Hibernate检查@IdClass，并挑出所有重复属性（通过比较名称和类型），作为标识符属性和主键的一部分。所有主键属性都通过@Id注解，并根据这些元素（字段或者获取方法）的位置，实体默认为字段或者属性访问。

注意，Hibernate XML映射中也有这最后一种策略；但是有点含糊：

```
<composite-id class="UserId" mapped="true">
    <key-property name="username"
                  column="USERNAME"/>
    <key-property name="departmentNr"
                  column="DEP_NR"/>
</composite-id>
```

你省略实体的标识符属性名称（因为没有），因此Hibernate内部处理标识符。利用mapped="true"，启用最后一种JPA映射策略，因此所有的键属性现在都要出现在User和UserId类中。

如果使用JPA XML描述符，这个复合标识符映射策略看起来如下：

```
<entity class="auction.model.User" access="FIELD">
    <id-class class="auction.model.UserId"/>
    <attributes>
        <id name="username"/>
        <id name="departmentNr"/>
    </attributes>
</entity>
```

由于我们没有给Java Persistence中定义的这最后一种策略找到令人信服的案例，因此只好假设它被添加到了规范中，来支持一些遗留的行为（EJB 2.x实体bean）。

复合外键也可能使用注解。先映射从Item到User的关联：

```
@ManyToOne
@JoinColumns({
  @JoinColumn(name="USERNAME", referencedColumnName = "USERNAME"),
  @JoinColumn(name="DEP_NR", referencedColumnName = "DEP_NR")
})
private User seller;
```

一般的@ManyToOne和这个映射之间的主要区别在于涉及的列数——这个顺序还是很重要，并且应该与主键列的顺序一致。然而，如果你对每个列声明referencedColumnName，顺序就不重要了，并且外键约束的源表和目标表都可以有不同的列名称。

从User到带有Item集合的反向映射甚至更为简单：

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

这个反向端需要mappedBy属性，就像通常对于双向的关联一样。因为这是反向端，因此它不需要任何列声明。

在遗留Schema中，外键经常不引用主键。

6. 外键引用非主键

通常，外键约束引用主键。外键约束是一个完整性规则，它保证被引用的表有一行所包含的键值与引用表和给定行中的键值相匹配。注意，外键约束可以自引用；换句话说，包含外键约束的列可以引用同一张表的主键列。（CaveatEmptor的CATEGORY表中的PAARAENT_CATEGORY就是一个例子。）

遗留Schema有时候会有不遵循简单的“外键引用主键”规则的外键约束。有时候，外键引用

非主键：一个简单的唯一列，一个自然的非主键。假设在CaveatEmptor中，你需要在USERS表中处理具名CUSTOMER_NR的遗留自然键列：

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

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

    <property name="customerNr"
              column="CUSTOMER_NR"
              not-null="true"
              unique="true"/>

</class>
```

这个映射中对于你来说唯一可能陌生的是unique属性。这是Hibernate中的其中一个SQL定制选项；不在运行时使用它（Hibernate不做任何唯一性验证），而是通过hbm2dbl导出数据库Schema。如果你有个包含自然键的现有Schema，就假设它是唯一的。为了完整起见，你可以并且应该在映射元数据中重复如此重要的约束——或许有一天你会用它导出一个新的Schema。

相当于XML映射，你可以在JPA注解中声明一个列为unique：

```
@Column(name = "CUSTOMER_NR", nullable = false, unique=true)
private int customerNr;
```

在遗留Schema中可能遇到的下一个问题是，ITEM表有一个外键列SELLER_NR。在理想的世界中，你会期待用这个外键去引用USERS表的主键USER_ID。然而，在遗留Schema中，它可以引用自然的唯一键CUSTOMER_NR。需要用一个属性引用映射它：

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

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

    <many-to-one name="seller" column="SELLER_NR"
                 property-ref="customerNr"/>

</class>
```

在更加怪异的Hibernate映射中你会遇到property-ref属性。它用来告诉Hibernate“这是具名属性的一个镜像”。在前一个例子中，Hibernate现在知道外键引用的目标了。你要进一步注意的是，property-ref要求目标属性要唯一，因此如前所述，这个映射需要unique="true"。

如果试图通过JPA注解映射这个关联，可以找一个相当于property-ref属性的等价物。用一个对自然键列CUSTOMER_NR的显式引用映射关联：

```
@ManyToOne
@JoinColumn(name="SELLER_NR", referencedColumnName = "CUSTOMER_NR")
private User seller;
```

现在Hibernate知道被引用的目标列是一个自然键，并相应地管理外键关系。

为了完成这个例子，利用User类中itemsForAuction集合的映射，使这两个类之间的关联映射变成双向。首先，这是以XML格式的：

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

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

    <property name="customerNr" column="CUSTOMER_NR" unique="true"/>
```

```
        <set name="itemsForAuction" inverse="true">
            <key column="SELLER_NR" property-ref="customerNr"/>
            <one-to-many class="Item"/>
        </set>

    </class>
```

ITEM中的外键列再次通过一个对customerNr的属性引用而被映射。在注解中，这更加容易映射为一个反向端：

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

7. 复合外键引用非主键

有些遗留Schema甚至比前面讨论过的更为复杂：外键可能是一个复合键，并且设计为引用一个复合的自然的非主键！

假设USERS有一个自然的复合键，包括FIRSTNAME、LASTNAME和BIRTHDAY列。外键可以引用这个自然键，如图8-1所示。

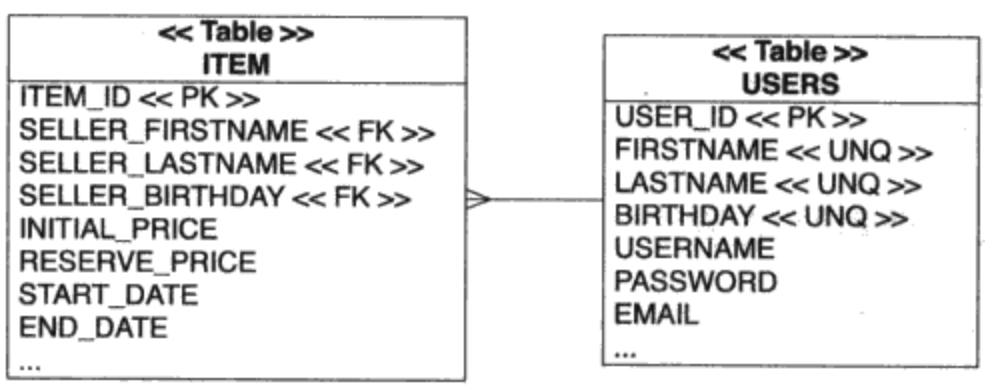

图8-1 复合外键引用复合的唯一键

为了映射这个，你需要按相同的名称给几个属性分组，否则无法在property-ref中给复合命名。应用<properties>元素来给映射分组：

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

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

    <properties name="nameAndBirthday" unique="true" update="false">
        <property name="firstname" column="FIRSTNAME"/>
        <property name="lastname" column="LASTNAME"/>
        <property name="birthday" column="BIRTHDAY" type="date"/>
    </properties>

    <set name="itemsForAuction" inverse="true">
        <key property-ref="nameAndBirthday">
            <column name="SELLER_FIRSTNAME"/>
            <column name="SELLER_LASTNAME"/>
            <column name="SELLER_BIRTHDAY"/>
        </key>
        <one-to-many class="Item"/>
    </set>

</class>
```

如你所见，<properties>元素不仅可以用于给几个属性命名，而且定义了一个多列的unique约束，或者使几个属性变成不可变。对于关联映射，列的顺序仍然很重要：

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

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

    <many-to-one name="seller" property-ref="nameAndBirthday">
        <column name="SELLER_FIRSTNAME"/>
        <column name="SELLER_LASTNAME"/>
        <column name="SELLER_BIRTHDAY"/>
    </many-to-one>

</class>
```

幸好，通过重构外键来引用主键的方式清除这样一个Schema经常很简单——如果你对数据库进行的改变能够不影响其他共享数据的应用。

这样就结束了我们对于在尝试映射遗留Schema时可能必须处理的自然、复合和外键相关的问题的探讨。让我们继续讨论其他值得关注的特殊映射策略。

有时候，你无法对遗留数据库进行任何改变——甚至不能创建表或者视图。Hibernate可以把类、属性，甚至关联的几个部分映射到一个简单的SQL语句或者表达式。我们称这种映射为公式映射（formula mapping）。

8.1.2　带有公式的任意联结条件

把Java工件映射到SQL表达式不仅仅对于整合遗留Schema有用。你已经创建过两个公式映射：第一个是一个简单的衍生只读属性映射，详见4.4.1节。第二个公式在一个继承映射中计算了辨别标志；请见5.1.3节。

现在将把公式应用于更为怪异的目的。记住，你现在将要见到的有些映射很复杂，在读完本书第二部分的所有章节之后可能会更容易理解它们。

1．理解用例

现在在两个实体之间映射一个文字联结条件。这听起来比实际应用程序更为复杂。看看图8-2中所示的两个类。

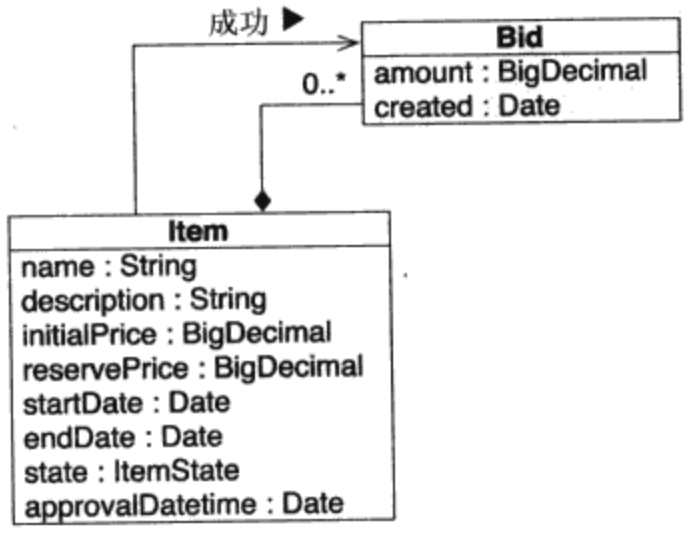

图8-2　引用多关联中一个实例的单个关联

特定的Item可以有几个Bid——这是个一对多的关联。但它不是两个类之间的唯一关联；另一个（单向的一对一关联）可用于挑选出一个特定的Bid实例作为胜出的出价。映射第一个关联，因为你想要能够通过调用anItem.getSuccessfulBid()获取一件拍卖货品的所有出价。逻辑上来说，集合中的其中一个元素也是被getSuccessfulBid()引用的成功的出价对象。

第一个关联很显然是个双向的一对多（多对一）关联，BID表中包含有一个外键ITEM_ID。（如果你之前还没有映射过它，请见6.4节。）

一对一的关联更为困难；可以用几种方式映射它。最自然的是ITEM表中一个被唯一约束的外键，它引用BID表中的一行——胜出的行，例如SUCCESSFUL_BID_ID列。

遗留Schema经常需要一个不是简单的外键关系的映射。

2. 映射公式联结条件

想象BID表中的每一行都有一个标志列，给胜出的出价做上标记，如图8-3所示。一个BID行把标记设置为true，且这件拍卖货品的所有其他行自然为false。在遗留Schema中很可能找不到对这个关系的约束或者完整性规则，但是我们现在不管它，而关注到Java类的映射。

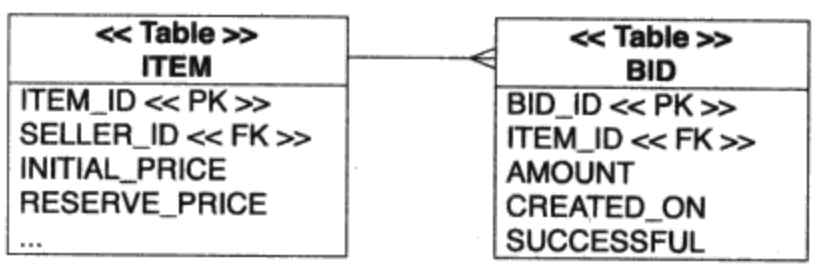

图8-3 胜出的出价用SUCCESSFUL列标志做上标记

为了使这个映射更值得关注，就假设遗留Schema没有使用SQL BOOLEAN数据类型，而是用CHAR(1)字段和值T（指true）和F（指false），模仿布尔的转换。你的目标是把这个标记列映射到Item类的一个successfulBid属性。要把它映射为对象引用，需要一个文字联结条件，因为Hibernate没有可以用于联结的外键。换句话说，对于每个ITEM行，你都需要从把SUCCESSFUL标记设置为T的BID表中联结一个行。如果没有这样的行，anItem.getSuccessfulBid()调用就返回null。

首先把Bid类和一个successful布尔属性映射到SUCCESSFUL数据库列：

```
<class name="Bid" table="BID">

    <id name="id" column="BID_ID"...

    <property name="amount"
              ...

    <properties name="successfulReference">

        <property name="successful"
                  column="SUCCESSFUL"
                  type="true_false"/>
                  ...

        <many-to-one name="item"
```

```
                    class="Item"
                    column="ITEM_ID"/>
                    ...
        </properties>

        <many-to-one name="bidder"
                     class="User"
                     column="BIDDER_ID"/>
                     ...
    </class>
```

type="true_false"属性在Java boolean基本（或者它的包装）属性和包含T/F文字值的一个简单的CHAR(1)列之间创建一个映射——它是个内建的Hibernate映射类型。你可以在其他映射中引用的名称下使用<properties>再次对若干属性进行组合。这里新出现的是，你可以组合<many- to-one>，而不只是那些基础的属性。

真正的技巧发生在另一端，对于Item类successfulBid属性的映射：

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

    <id name="id" column="ITEM_ID"...

    <property name="initialPrice"
        ...

    <one-to-one name="successfulBid"
                property-ref="successfulReference">
        <formula>'T'</formula>
        <formula>ITEM_ID</formula>
    </one-to-one>

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

</class>
```

在这个例子中忽略<set>关联映射；这是Item和Bid之间常规的一对多双向关联，处于BID中的ITEM_ID外键列中。

说明　<one-to-one>不用于主键关联吗？通常，当两个实体表中的行有着相同的主键值时，<one-to-one>映射便是两个实体之间的主键关系。然而，通过使用一个包含property-ref的formula，你可以把它应用到外键关系。在本节所示的例子中，可以用<many-to-one>代替<one-to-one>元素，它仍然有效。

值得关注的部分是<one-to-one>映射，以及当你在使用关联的时候，它如何依赖property-ref和文字型formula值作为联结条件。

3. 使用关联

查询一件拍卖货品的获取和它的成功出价的完整的SQL看起来像这样：

```
select
    i.ITEM_ID,
    i.INITIAL_PRICE,
    ...
    b.BID_ID,
    b.AMOUNT,
    b.SUCCESSFUL,
    b.BIDDER_ID,
    ...
from
    ITEM i
left outer join
    BID b
        on 'T' = b.SUCCESSFUL
        and i.ITEM_ID = b.ITEM_ID
where
    i.ITEM_ID = ?
```

加载Item时，Hibernate现在通过应用一个涉及successfulReference属性列的联结条件，来从BID表联结一个行。因为这是个分组属性，你可以给所涉及的每个列以正确的顺序声明单独的表达式。第一个（'T'）是一个文字，如你从引号中所见到的。当Hibernate试图查出在BID表中是否有成功的行时，它现在把'T'=SUCCESSFUL包括在联结条件中。第二个表达式不是文字，而是一个列名称（没有引号）。因此，附上另一个联结条件i.ITEM_ID = b.ITEM_ID。如果需要额外的限制，可以把它扩展并添加更多的联结条件。

注意生成了一个外部联结，因为所讨论的货品不可能有成功的出价，因此给每个b.*列返回NULL。现在可以调用anItem.getSuccessfulBid()，来获取一个对成功出价的引用（或者如果不存在引用则为null）。

最后，用或者不用数据库约束，都无法只实现仅在Item实例的一个私有字段上设置值的item.setSuccessfulBid()方法。你必须在这个设置方法中实现一个小过程，负责这个特殊的关系和出价中的标记属性：

```
public class Item {
    ...

    private Bid successfulBid;
    private Set<Bid> bids = new HashSet<Bid>();

    public Bid getSuccessfulBid() {
        return successfulBid;
    }

    public void setSuccessfulBid(Bid successfulBid) {
        if (successfulBid != null) {

            for (Bid bid : bids)
                bid.setSuccessful(false);

            successfulBid.setSuccessful(true);
            this.successfulBid = successfulBid;
        }
```

```
        }
    }
```

调用setSuccessfulBid()时，你设置所有出价都不成功。这么做可能触发集合的加载——使用这一策略必须付出的代价。然后，这个新的成功出价被做上标记，并设置成一个实例变量。保存对象时，对标记的设置更新BID表中的SUCCESSFUL列。为了完成这个（并修复遗留Schema），数据库级的约束必须完成与这个方法相同的事。（本章的后面会回到这个话题。）

关于这个文字联结条件映射要记住的其中一点是，它可以被应用在许多其他的情形中，而不仅仅对于成功的或者默认的关系。每当需要把一些任意的联结条件附加到查询时，最好的选择是公式。例如，可以在<many-to-many>映射中用它创建从关联表到实体表的一个文字联结条件。

不幸的是，在编写本书之时，Hibernate Annotations还不支持用公式表达的任意联结条件。在一个引用名称下分组属性也是不可能的。我们希望这些特性一旦可用时，将十分类似于XML映射。

在遗留Schema中可能遇到的另一个问题是，它不能与类粒度很好地整合。通常建议的类要比表更多可能不起作用，并且可能你必须反其道而行之，把任意表联结到一个类里面。

8.1.3　联结任意的表

第5章的一个继承映射中已经介绍过<join>映射元素，请见5.1.5节。它帮助把特定子类的属性分到一张单独的表里面，处在主继承层次结构表之外。这项一般的功能有更多用途——但是我们必须提醒你，<join>也可能是个馊主意。任何设计适当的系统都应该是类比表更多。把单个类分离到几张单独的表中，是只有在你需要把遗留Schema中的几张表合并到单个类中时才应该做的事。

1. 把属性移到二级表中

假设在CaveatEmptor中，你没有把包含用户主要信息的用户地址信息保存在映射为组件的USERS表中，而是在一张单独的表中。如图8-4所示。注意，每个BILLING_ADDRESS都有一个外键USER_ID，反过来USER_ID也是BILLING_ADDRESS表的主键。

为了在XML映射它，需要把Address的属性组合在<join>元素中：

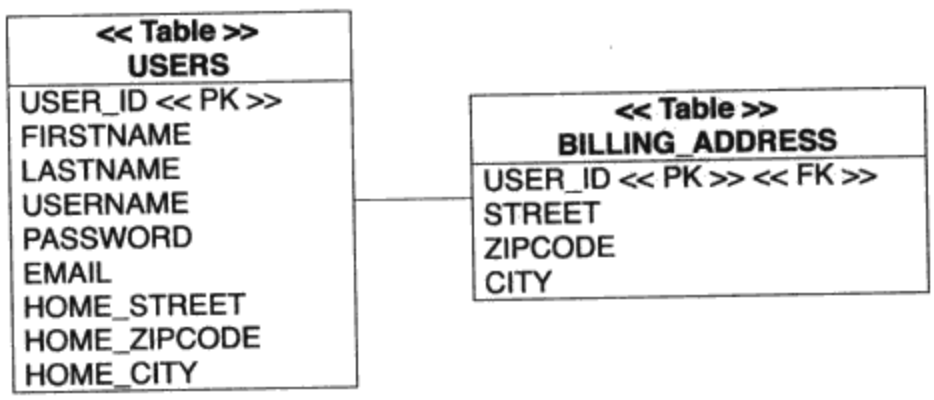

图8-4　把账单地址数据分到二级表

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

    <join table="BILLING_ADDRESS" optional="true">
        <key column="USER_ID"/>
```

```
        <component name="billingAddress" class="Address">
            <property   name="street"
                        type="string"
                        column="STREET"
                        length="255"/>
            <property   name="zipcode"
                        type="string"
                        column="ZIPCODE"
                        length="16"/>
            <property   name="city"
                        type="string"
                        column="CITY"
                        length="255"/>
        </component>
    </join>

</class>
```

你不一定要联结组件；也可以联结单独的属性甚至<many-to-one>（我们在第7章中给可选的实体关联这么实现过）。通过设置optional="true"，表明组件属性对于没有billingAddress的User也可能为null，并且没有行应该被插入到这张二级表中。Hibernate还执行一个外部联结（而不是内部联结）来从二级表中获取行。如果在<join>映射中声明了fetch="select"，二级表就可以用于这一目的。

Java Persistence规范中也包括二级表的概念。首先，你必须对一个特定的实体声明一张（或者几张）二级表：

```
@Entity
@Table(name = "USERS")
@SecondaryTable(
    name = "BILLING_ADDRESS",
    pkJoinColumns = {
        @PrimaryKeyJoinColumn(name="USER_ID")
    }
)
public class User {
    ...
}
```

每张二级表都需要一个名称和一个联结条件。在这个例子中，外键列引用USERS表的主键列，就像前面在XML映射中的一样。（这是个默认的联结条件，因此可以只声明二级表名称，不需要其他东西。）你可能看到注解的语法开始有问题，并且代码也更难以阅读。所幸的是，你不必经常使用二级表。

实际的组件属性billingAddress，被映射为一个常规的@Embedded类，就像常规的组件一样。然而，需要覆盖每个组件属性列，并把它分配到User类中的二级表：

```
@Embedded
@AttributeOverrides( {
    @AttributeOverride(
        name   = "street",
        column = @Column(name="STREET",
                         table = "BILLING_ADDRESS")
```

```
        ),
        @AttributeOverride(
            name   = "zipcode",
            column = @Column(name="ZIPCODE",
                             table = "BILLING_ADDRESS")
        ),
        @AttributeOverride(
            name   = "city",
            column = @Column(name="CITY",
                             table = "BILLING_ADDRESS")
        )
    })
    private Address billingAddress;
```

这样就不再容易阅读了，但这是为在注解中使用声明元数据的映射灵活性所付出的代价。或者，可以使用JPA XML描述符：

```
<entity class="auction.model.User" access="FIELD">
    <table name="USERS"/>
    <secondary-table name="BILLING_ADDRESS">
        <primary-key-join-column
            referenced-column-name="USER_ID"/>
    </secondary-table>
    <attributes>
        ...
        <embedded name="billingAddress">
            <attribute-override name="street">
                <column name="STREET" table="BILLING_ADDRESS"/>
            </attribute-override>
            <attribute-override name="zipcode">
                <column name="ZIPCODE" table="BILLING_ADDRESS"/>
            </attribute-override>
            <attribute-override name="city">
                <column name="CITY" table="BILLING_ADDRESS"/>
            </attribute-override>
        </embedded>
    </attributes>
</entity>
```

另一种<join>元素甚至更为怪异的使用案例是被反向联结的属性或者组件。

2. 被反向联结的属性

假设在CaveatEmptor中，有具名DAILY_BILLING的遗留表。这张表包括所有开放支付，对任何拍卖都以每夜批量的方式执行。表里有一个到ITEM的外键列，如图8-5所示。

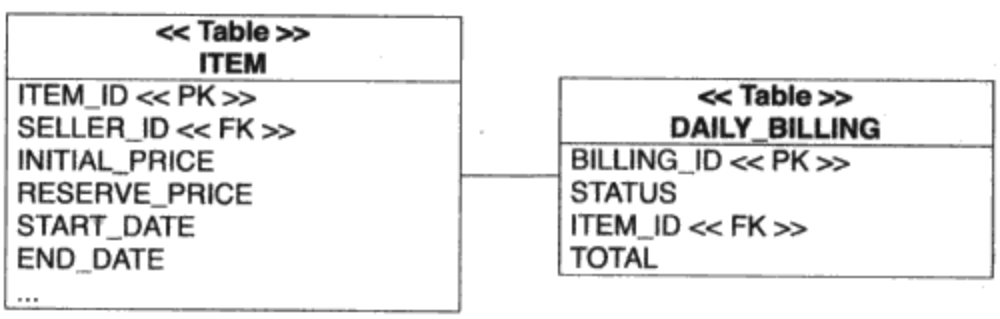

图8-5　每日账单摘要引用货品并包含总金额

每笔支付都包括TOTAL列，包含要收的总金额。在CaveatEmptor中，如果能够通过调用anItem.getBillingTotal()访问一次特定拍卖的价格就会很方便。

可以把这个列从DAILY_BILLING表映射到Item类中。然而，永远不要从这一端插入或者更新它；它是只读的。基于这个原因，把它映射为反向的——负责维持列值的另一端的简单镜像（假设你在这里没有映射它）：

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

    <join table="DAILY_BILLING" optional="true" inverse="true">
        <key column="ITEM_ID"/>
        <property   name="billingTotal"
                    type="big_decimal"
                    column="TOTAL"/>
    </join>

</class>
```

注意，对于这个问题的另一种可以选择的解决方案是一个使用公式表达式和关联子查询的衍生属性：

```
<property name="billingTotal"
          type="big_decimal"
          formula="( select db.TOTAL from DAILY_BILLING db
                     where db.ITEM_ID = ITEM_ID )"/>
```

主要区别在于用来加载ITEM的SQL SELECT：如果启用<join fetch="select">，第一种解决方案则默认为一个外部联结，通过可选的另一个SELECT。这个衍生属性在原始查询的选择子句中产生一个嵌入的子查询。在编写本书之时，注解还不支持反向的联结映射，但是可以给公式使用Hibernate注解。

就像你或许可以从这些例子中所猜到的，<join>映射在多种情况下派上了用场。如果与公式结合，它们甚至更为强大，但是我们希望你不必经常使用这种结合。

在使用遗留数据的上下文中经常出现的进一步问题是数据库触发器。

8.1.4 使用触发器

甚至在全新的数据库中也有理由使用触发器，因此遗留数据不是触发器可能产生问题的唯一场景。使用ORM软件的触发器和对象状态管理通常是个问题，因为触发器可能在不合时机的时候运行，或者可能修改不与内存状态同步的数据。

1. 在插入时运行的触发器

假设ITEM表有一个CREATED列，被映射到类型Date的created属性，它通过一个在插入中自动执行的触发器初始化。下列映射很恰当：

```
<property name="created"
          type="timestamp"
          column="CREATED"
          insert="false"
          update="false"/>
```

注意，你映射这个属性insert="false" update="false"，表明它没有被Hibernate包括在SQL INSERT或者UPDATE中。

保存新Item之后，Hibernate不知道被触发器分配到这个列的值，因为它发生在INSERT该货品行之后。如果需要在应用程序中生成的值，就必须显式地告诉Hibernate用SQL SELECT重新加载该对象。例如：

```
Item item = new Item();
...
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

session.save(item);
session.flush(); // Force the INSERT to occur
session.refresh(item); // Reload the object with a SELECT

System.out.println( item.getCreated() );

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

涉及触发器的大部分问题都可能以这个方式解决，用一个显式的flush()强制立即执行触发器，可能接着调用refresh()来获取触发器的结果。

在把refresh()调用添加到应用程序之前，我们必须告诉你前一节的主要目标在于介绍什么时候使用refresh()。许多Hibernate的初学者不理解它真正的用途，经常不恰当地使用它。refresh()更为正式的定义是“利用数据库中的当前值，刷新处于持久化状态的内存实例。”

对于所介绍的例子，数据库触发器在插入之后填入了一个列值，这是可以使用的一种更为简单的技术：

```
<property name="created"
          type="timestamp"
          column="CREATED"
          generated="insert"
          insert="false"
          update="false"/>
```

通过注解，使用Hibernate扩展：

```
@Temporal(TemporalType.TIMESTAMP)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
@Column(name = "CREATED", insertable = false, updatable = false)
private Date created;
```

4.4.1节的“生成的和默认的属性值”已经深入讨论了generated属性。利用generated="insert"，Hibernate在插入之后自动执行SELECT，来获取更新过的状态。

当数据库执行触发器时，要进一步注意的问题是：脱管对象图的再关联和在每次UPDATE时运行的触发器。

2. 在更新时运行的触发器

在讨论ON UPDATE触发器与对象的重附结合使用之前，需要强调另外一项对generated属性

的设置：

```
<version name="version"
         column="OBJ_VERSION"
         generated="always"/>
...
<timestamp name="lastModified"
           column="LAST_MODIFIED"
           generated="always"/>
...
<property name="lastModified"
          type="timestamp"
          column="LAST_MODIFIED"
          generated="always"
          insert="false"
          update="false"/>
```

利用注解，与之相当的映射如下：

```
@Version
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "OBJ_VERSION")
private int version;

@Version
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "LAST_MODIFIED")
private Date lastModified;

@Temporal(TemporalType.TIMESTAMP)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "LAST_MODIFIED", insertable = false, updatable = false)
private Date lastModified;
```

利用always，启用Hibernate的自动刷新，不仅针对行的插入还针对行的更新。换句话说，每当一个版本、时间戳、或者任何属性值由在UPDATE SQL语句中运行的触发器生成时，都需要启用这个选项。再次参考我们之前在4.4.1节中对生成的属性所进行的讨论。

看看如果你拥有在更新中运行的触发器可能会遇到的第二个问题。由于当脱管对象被重附到新的Session（通过update()或者saveOrUpdate()）时，没有快照可用，Hibernate可能执行不必要的SQL UPDATE语句，确保数据库状态与持久化的上下文状态同步。这可能导致不合时宜地触发一个UPDATE触发器。通过在映射中为被持久化到带有触发器的表的类启用select-before-update，可以避免这种行为。如果ITEM表有一个更新触发器，就把下列属性添加到映射中：

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

```
        select-before-update="true">
    ...
</class>
```

这项设置强制Hibernate利用SQL SELECT来获取当前数据库状态的快照，如果内存Item的状态相同，则避免后面的UPDATE。你用额外的SELECT避免了不合时宜的UPDATE。

Hibernate注解启用同样的行为：

```
@Entity
@org.hibernate.annotations.Entity(selectBeforeUpdate = true)
public class Item { ... }
```

在试图映射遗留Schema之前，要注意更新之前的SELECT仅仅获取所述实体实例的状态。没有主动抓取任何集合或者关联实例，也没有任何预抓取优化是活动的。如果开始给系统中的多个实体启用selectBeforeUpdate，可能会发现通过未被优化的查询引入的性能问题是有问题的。更好的一种策略是使用合并而不是重附。然后Hibernate可以在获取数据库快照时应用一些优化（外部联结）。我们会在本书的后面更深入地讨论重附和合并之间的区别。

我们来概括一下对遗留数据模型的讨论：Hibernate提供几种轻松处理（自然的）复合键和不合时宜的列的策略。在尝试映射遗留Schema之前，我们建议要细心检查，看是否有可能改变Schema。依我们的经验，许多开发人员立即就认为改变数据库Schema太过复杂和费时，转而寻求一种Hibernate解决方案。有时候这样并不正确，应该把Schema演变当作Schema生命周期很自然的一部分。如果表改变了，那么数据导出、一些转换和导入便可能解决问题。从长期来看，一天的工作可能节省几天的工作。

遗留Schema经常也需要由Hibernate生成的SQL定制，用于数据操作（DML）或者Schema定义（DDL）。

8.2　定制 SQL

出现于20世纪70年代的SQL，直到1986年才被（ANSI）标准化。虽然SQL标准的每一次更新都带来（许多有争议的）新特性，每一个支持SQL的DBMS产品都以它自己独特的方式这么做着。可移植性的重担再次落在了数据库应用程序开发人员的肩上。这就是Hibernate有用的地方：它的内建查询机制、HQL和Criteria API，根据所配置的数据库方言生成了SQL。所有其他自动生成的SQL（例如，当集合必须按需获取时）也在方言的帮助下生成。利用一个简单的方言转换，可以在一个不同的DBMS上运行应用。

为了支持这种可移植性，Hibernate必须处理三种操作：

- 每个数据获取操作导致正被执行的SELECT语句。它可能有多种变形。例如，数据库产品可能给联结操作使用一种不同的语法，或者结果如何被限制到特定数目的行。
- 每个数据修改都需要执行DML语句，例如UPDATE、INSERT和DELECT。DML经常不像数据获取那么复杂，但是它仍然具有特定于产品的变形。
- 数据库Schema必须在可以执行DML和数据获取之前创建或者改变。使用DDL在数据库目录上工作；它包括如CREATE、ALTER和DROP这样的语句。DDL几乎完全是特定于供应商的，但是大部分产品至少有一个类似的语法结构。

我们经常使用的另一个术语是CRUD，指create（创建）、read（读取）、update（更新）和delete（删除）。Hibernate为你、为所有CRUD操作和模式定义生成所有这些SQL。这种转化基于`org.hibernate.dialect.Dialect`实现——Hibernate为所有流行的SQL数据库管理系统捆绑了方言。我们鼓励你看看正在使用的方言的源代码；它并不难读。一旦你对使用Hibernate更有经验之后，可能甚至想要扩展一种方言，或者编写自己的方言。例如，想要注册一个定制SQL函数用在HQL查询中，要用一个新的子类扩展现有的方言，并添加注册代码——还是要检查现有的源代码，找出更多有关方言系统的灵活性。

另一方面，当你要在一个更低级的抽象上工作时，有时候需要比Hibernate API（或者HQL）提供的更多控制。通过Hibernate，可以覆盖或者完全替换所有将被执行的CRUD SQL语句。如果依赖Hibernate的自动Schema导出工具（不一定），就可以定制和扩展所有定义Schema的DDL SQL语句。

此外，Hibernate允许你始终通过`session.connection()`获取简单的JDBC `Connection`对象。当没有其他东西可行，或者任何其他东西都比简单的JDBC更难时，就应该把这项特性用作最后的杀手锏。最新的Hibernate版本，是有幸十分罕见的，因为它给典型的无状态JDBC操作（例如大批量的更新和删除）内建了越来越多的特性，并且已经有了用于定制SQL的许多扩展点。

这些定制的SQL（DML和DDL）都是本节的话题。我们从用于创建、读取、更新和删除操作的定制DML开始。随后整合数据库的存储过程去做相同的工作。最后，看看给数据库Schema自动生成的DDL定制，以及如何创建表示数据库管理员优化工作的一个好起点的Schema。

注意，在编写本书之时，注解中还没有自动生成的SQL的这个详细定制；因此，接下来的例子将专门使用XML元数据。我们期待Hibernate Annotations的未来版本将更好地支持SQL定制。

8.2.1 编写定制 CRUD 语句

你要编写的第一个定制SQL用来加载实体和集合。（下列示例中的大部分代码显示了与Hibernate默认执行的几乎相同的SQL，没有多少定制——这有助于你更快地理解映射技术。）

1. 用定制SQL加载实体和集合

对于需要用定制SQL操作来加载实例的每个实体类，定义对具名查询的`<loader>`引用：

```
<class name="User" table="USERS">
    <id name="id" column="USER_ID"...
    <loader query-ref="loadUser"/>
    ...
</class>
```

`loadUser`查询现在可以被定义在映射元数据中的任何地方，从它的用途中分离出来封装。这是一个简单查询的例子，它给`User`实体实例获取数据：

```
<sql-query name="loadUser">
    <return alias="u" class="User"/>
    select
        us.USER_ID      as {u.id},
        us.FIRSTNAME    as {u.firstname},
```

```
        us.LASTNAME     as {u.lastname},
        us.USERNAME     as {u.username},
        us."PASSWORD"   as {u.password},
        us.EMAIL        as {u.email},
        us.RANKING      as {u.ranking},
        us.IS_ADMIN     as {u.admin},
        us.CREATED      as {u.created},
        us.HOME_STREET  as {u.homeAddress.street},
        us.HOME_ZIPCODE as {u.homeAddress.zipcode},
        us.HOME_CITY    as {u.homeAddress.city},
        us.DEFAULT_BILLING_DETAILS_ID as {u.defaultBillingDetails}
    from
        USERS us
    where
        us.USER_ID = ?
</sql-query>
```

如你所见，从列名称到实体属性的映射使用了一个简单的别名。在具名的加载程序中查询实体，必须SELECT以下列和属性：

- 主键列和主键属性或者属性，如果使用复合主键的话。
- 所有标量属性，必须从它们相应的列中被初始化。
- 所有必须被初始化的复合属性。可以利用下列别名语法处理单独的标量元素：{entityalias.componentProperty.scalarProperty}。
- 所有外键列，必须被获取并映射到相应的many-to-one属性。请见前一个片段中的DEFAULT_BILLING_DEFAULTS_ID实例。
- 所有标量属性、复合属性和<join>元素内部的多对一实体引用。如果所有被联结的属性永远不为NULL，就使用一个联结到二级表的内部联结；否则，用外部联结比较恰当。（注意这在本例中并没有体现出来。）
- 如果你通过字节码基础设施给标量属性启用了延迟加载，就不需要加载延迟属性。请见13.1.6节。

前一个例子中所示的{propertyName}别名并非绝对必要。如果结果中的一个列名与被映射列的名称相同，Hibernate就会自动把它们绑到一起。

甚至可以在应用程序中利用session.getNamedQuery("loadUser")，按名称调用被映射的查询。利用定制的SQL查询，更多的事情成为可能，但是我们在本节中关注对CRUD的基础SQL定制。15.2节将回到这个话题。

假设你也想定制用来加载集合的SQL——例如，被一位User售出的items。首先，在集合映射中声明一个装载程序引用：

```
<set name="items" inverse="true">
    <key column="SELLER_ID" not-null="true"/>
    <one-to-many class="Item"/>
    <loader query-ref="loadItemsForUser"/>
</set>
```

具名查询loadItemsForUser看起来几乎与实体装载程序相同：

```
<sql-query name="loadItemsForUser">
    <load-collection alias="i" role="User.items"/>
    select
        {i.*}
    from
        ITEM i
    where
        i.SELLER_ID = :id
</sql-query>
```

有两个重要的区别：一个是从别名映射到集合角色的<load-collection>；它应该是一目了然的。这个查询中新出现的东西是，从SQL表别名ITEM i到所有包含{i.*}的货品属性的自动映射。通过使用相同的别名（符号i），创建了两者之间的一个联结。此外，你现在正在使用具名参数:id，而不是包含问号标记的简单的定位参数。可以使用自己喜欢的任何一种语法。

有时候，通过外部联结，可以在单个查询中更好地加载实体实例和集合（实体可能有一个空的集合，因此无法使用内部联结）。如果想要应用这个主动抓取，就不要为集合声明装载程序引用。实体装载程序负责集合的获取：

```
<sql-query name="loadUser">
    <return alias="u" class="User"/>
    <return-join alias="i" property="u.items"/>
    select
        {u.*}, {i.*}
    from
        USERS u
        left outer join ITEM i
            on u.USER_ID = i.SELLER_ID
    where
        u.USER_ID = ?
</sql-query>
```

注意如何使用<return-join>元素把别名绑定到实体的集合属性，有效地把两个别名链接在一起。还要进一步注意，如果想要在一个原始查询中主动抓取一对一和多对一关联的实体，这个方法也有效。在这种情况下，如果被关联的实体是强制的（外键不能为NULL），你可能想要用内部联结；或者如果目标是可选的，则可能要用外部联结。可以在查询中即时获取许多个单端的关联；然而，如果（外部）联结了不止一个集合，就会创建一个笛卡儿积，有效地乘以所有的集合行。这样会产生可能比两个查询更慢的巨大结果。当我们在第13章中讨论抓取策略时，你会再次见到这一限制。

如前所述，你会在本书的后面见到更多用于对象加载的SQL选项。现在来讨论插入、更新和删除操作的定制，完成CRUD基础的讨论。

2. 定制插入、更新和删除

Hibernate在启动时生成所有琐碎的CRUD SQL。它在内部高速缓存SQL声明以备未来之用，这样避免了对最普通操作的SQL生成的任何运行时成本。你已经知道如何覆盖CRUD中的R（读取），因此让我们对CUD进行同样的讨论。

对于给每个实体或者集合，可以在<sql-insert>、<sql-delete>和<sql-update>内部相

应地定义定制的CUD SQL语句：

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

    <id name="id" column="USER_ID"...

    ...

    <join table="BILLING_ADDRESS" optional="true">
        <key column="USER_ID"/>
        <component name="billingAddress" class="Address">
            <property ...
        </component>

        <sql-insert>
            insert into BILLING_ADDRESS
                            (STREET, ZIPCODE, CITY, USER_ID)
            values (?, ?, ?, ?)
        </sql-insert>

        <sql-update>...</sql-update>

        <sql-delete>...</sql-delete>

    </join>

    <sql-insert>
        insert into USERS (FIRSTNAME, LASTNAME, USERNAME,
                           "PASSWORD", EMAIL, RANKING, IS_ADMIN,
                           CREATED, DEFAULT_BILLING_DETAILS_ID,
                           HOME_STREET, HOME_ZIPCODE, HOME_CITY,
                           USER_ID)
            values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    </sql-insert>

    <sql-update>...</sql-update>

    <sql-delete>...</sql-delete>

</class>
```

这个映射示例可能看起来有点复杂，但实际上很简单。你在单个映射中有两张表：主表用于实体USERS，二级表BILLING_ADDRESS来自本章前面的遗留映射。每当有用于实体的二级表时，就必须把它包括在任何定制SQL里面——因而<sql-insert>、<sql-delete>和<sql-update>元素既在映射的<class>部分，又在它的<join>部分。

下一个问题是给语句绑定参数。对于CUD SQL定制，在编写本书之时还只支持占位参数。但是什么是参数的正确顺序？Hibernate如何把实参绑定到SQL参数有一个内在的顺序。找到正确的SQL语句和参数顺序的最容易方法是让Hibernate为你生成一个。从映射文件中移除定制SQL，给org.hibernate.persister.entity包启用DEBUG日志，并观察（或者搜索）Hibernate启动日志，查找类似于这些的行：

```
AbstractEntityPersister - Insert 0: insert into USERS (FIRSTNAME,
 LASTNAME, USERNAME, "PASSWORD", EMAIL, RANKING, IS_ADMIN,
 CREATED, DEFAULT_BILLING_DETAILS_ID, HOME_STREET, HOME_ZIPCODE,
 HOME_CITY, USER_ID) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

```
AbstractEntityPersister - Update 0: update USERS set
 FIRSTNAME=?, LASTNAME=?, "PASSWORD"=?, EMAIL=?, RANKING=?,
 IS_ADMIN=?, DEFAULT_BILLING_DETAILS_ID=?, HOME_STREET=?,
 HOME_ZIPCODE=?, HOME_CITY=? where USER_ID=?
...
```

你现在可以复制想要定制到映射文件中的语句，并进行必要的改变。更多有关Hibernate中日志的信息，请参考2.1.3节。

你现在已经把CRUD操作映射到了定制的SQL语句。另一方面，动态的SQL并不是获取和操作数据的唯一方法。保存在数据库中预设的和编译好的过程也可以被映射到对实体和集合的CRUD操作。

8.2.2 整合存储过程和函数

存储过程在数据库应用程序开发中很常见。把代码移近数据并在数据库内部执行它有着明显的好处。

首先，你不一定要在访问数据的每个程序中复制功能和逻辑。一种不同的观点是许多业务逻辑不应该被重复，因此它始终可以被应用。这包括保证数据完整性的过程（例如，复杂到难以声明式地实现的约束）。你通常也会在具有过程完整性规则的数据库中发现触发器。

存储过程在大量数据上的所有处理中都有优势，例如生成报表和统计分析。你应该始终努力避免在网络上以及在数据库与应用程序服务器之间设置大数据，因此存储过程是大数据操作自然的选择。或者，你可以实现一个复杂的数据获取操作，在它把最终的结果传递到应用程序客户端之前，通过几个查询把数据装配起来。

另一方面，你将经常看到利用存储过程实现甚至最基础的CRUD操作的（遗留）系统。作为它的一种变形，不允许任何直接的SQL DML但只允许存储过程调用的系统，也（且有时候仍然）有它们的用处。

可以开始给CRUD或者大量数据操作整合现有的存储过程，或者可以先开始编写自己的存储过程。

1. 编写过程

用于存储过程的编程语言通常是专有的。Oracle PL/SQL（SQL的一种过程方言）非常流行（且适用于其他数据库产品的变形）。有些数据库甚至支持用Java编写的存储过程。标准的Java存储过程是SQLJ成果的一部分，可惜还没有成功。

在本节中你将使用最常见的存储过程系统：Oracle数据库和PL/SQL。结果表明Oracle中的存储过程，就像众多的其他东西一样，始终与你期待的结果会有所不同；每当有些东西需要特别的关注时，我们会提醒你。

用PL/SQL编写的存储过程必须在数据库目录中作为源代码创建，然后编译。先来编写一个可以加载所有符合特定标准的User实体的存储过程：

```
<database-object>
    <create>
        create or replace procedure SELECT_USERS_BY_RANK
```

```
(
 OUT_RESULT out SYS_REFCURSOR,
 IN_RANK    in  int
) as
begin
 open OUT_RESULT for
 select
     us.USER_ID          as USER_ID,
     us.FIRSTNAME        as FIRSTNAME,
     us.LASTNAME         as LASTNAME,
     us.USERNAME         as USERNAME,
     us."PASSWORD"       as PASSWD,
     us.EMAIL            as EMAIL,
     us.RANKING          as RANKING,
     us.IS_ADMIN         as IS_ADMIN,
     us.CREATED          as CREATED,
     us.HOME_STREET      as HOME_STREET,
     us.HOME_ZIPCODE     as HOME_ZIPCODE,
     us.HOME_CITY        as HOME_CITY,
     ba.STREET           as BILLING_STREET,
     ba.ZIPCODE          as BILLING_ZIPCODE,
     ba.CITY             as BILLING_CITY,
     us.DEFAULT_BILLING_DETAILS_ID
                         as DEFAULT_BILLING_DETAILS_ID
 from
     USERS us
 left outer join
     BILLING_ADDRESS ba
         on us.USER_ID = ba.USER_ID
    where
        us.RANKING >= IN_RANK;
end;
</create>
<drop>
    drop procedure SELECT_USERS_BY_RANK
</drop>
</database-object>
```

你在<database-object>元素中给存储过程嵌入了DDL，用于创建和移除。这样，在数据库Schema利用hbm2ddl工具被创建和更新时，Hibernate就会自动创建和删除过程。也可以用手工在数据库目录中执行DDL。如果使用一个没有现有存储过程的非遗留系统时，把它留在映射文件（在任何适当的位置，例如在MyStoredProcedures.htm.xml）中是个好办法。我们会在本章稍后回到<database-object>映射的其他选项。

跟以前一样，这个例子中的存储过程代码很简单：一个针对基表（User类的主表和二级表）的联结查询和RANKING限制（一个输入参数到过程）。

对于在Hibernate中映射的存储过程你必须遵守一些规则。存储过程支持IN和OUT参数。如果你通过Oracle自己的JDBC驱动程序使用存储过程，Hibernate要求存储过程的第一个参数是OUT；对于要被假定用于查询的存储过程，查询结果被认为是要在这个参数中返回。在Oracle 9或者更新的版本中，OUT参数的类型必须为SYS_REFCURSOR。在Oracle更早的版本中，你必须首先定义

自己的引用光标类型，称作REF CURSOR——在Oracle产品文档中可以找到许多例子。所有其他主要的数据库管理系统（和用于并非来自Oracle的Oracle DBMS的驱动程序）与JDBC兼容，可以在存储过程中不用OUT参数直接返回结果。例如，Microsoft SQL Server中一个类似的过程看起来如下：

```
create procedure SELECT_USERS_BY_RANK
  @IN_RANK int
  as
    select
       us.USER_ID          as USER_ID,
       us.FIRSTNAME        as FIRSTNAME,
       us.LASTNAME         as LASTNAME,
    ...
from
   USERS us
where us.RANKING >= @IN_RANK
```

把这个存储过程映射到Hibernate中的具名查询。

2. 利用过程查询

用于查询的存储过程被映射为一般的具名查询，但有一些微小的区别：

```
<sql-query name="loadUsersByRank" callable="true">
    <return alias="u" class="User">
        <return-property name="id"           column="USER_ID"/>
        <return-property name="firstname"    column="FIRSTNAME"/>
        <return-property name="lastname"     column="LASTNAME"/>
        <return-property name="username"     column="USERNAME"/>
        <return-property name="password"     column="PASSWD"/>
        <return-property name="email"        column="EMAIL"/>
        <return-property name="ranking"      column="RANKING"/>
        <return-property name="admin"        column="IS_ADMIN"/>
        <return-property name="created"      column="CREATED"/>
        <return-property name="homeAddress">
            <return-column name="HOME_STREET"/>
            <return-column name="HOME_ZIPCODE"/>
            <return-column name="HOME_CITY"/>
        </return-property>
        <return-property name="billingAddress">
            <return-column name="BILLING_STREET"/>
            <return-column name="BILLING_ZIPCODE"/>
            <return-column name="BILLING_CITY"/>
        </return-property>
        <return-property name="defaultBillingDetails"
                         column="DEFAULT_BILLING_DETAILS_ID"/>
    </return>
    { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

与一般的SQL查询映射相比，第一个区别是callable="true"属性。它在Hibernate中启用了对可调用语句的支持，并纠正了存储过程的输出的处理。下列映射把在过程结果中返回的列名称绑定到User对象的属性。需要特别考虑的一个特定案例：如果在类中出现一些多列属性，包括几个组件（homeAddress），那么需要按正确的顺序映射它们的列。例如，homeAddress属性通过

三个属性映射为<component>，每个属性都映射到它自己的列。因此，存储过程映射包括绑定到homeAddress属性的三个列。

存储过程的call准备了OUT（问号标记）和具名的输入参数。如果你没有使用Oracle JDBC驱动（其他驱动或者不同的DBMS），就不需要保存第一个OUT参数；结果可以直接从存储过程中返回。

看看User类的常规类映射。注意，这个例子中被过程返回的列名与你已经映射过的列名相同。可以省略每个属性的绑定，并让Hibernate自动负责映射：

```
<sql-query name="loadUsersByRank" callable="true">
    <return class="User"/>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

给所有属性以及与常规映射中同名的类的外键关联返回正确列的责任，现在转移到了存储过程代码上。因为你在存储过程中已经有别名（select ... us.FIRSTNAME as FIRSTNAME...），这很简单。或者，只要过程结果返回的列中有一些列有着与已经映射为属性的列不同的名称，就只需要声明下面这些代码：

```
<sql-query name="loadUsersByRank" callable="true">
    <return class="User">
        <return-property name="firstname"    column="FNAME"/>
        <return-property name="lastname"     column="LNAME"/>
   </return>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

最后，来看看存储过程的call。这里使用的语法{ call PROCEDURE()}，定义在SQL标准中，是可移植的。对Oracle有效的一种不可移植的语法是begin PROCEDURE(); end;。建议你始终使用可移植的语法。过程有两个参数，如前所述，第一个被保存为输出参数，因此你用一个占位参数符号（?）。如果给Oracle JDBC驱动配置了一种方言，Hibernate就会负责这个参数。第二个是执行调用时必须提供的输入参数。可以只使用占位参数，或者混合使用具名和占位参数。为了方便阅读，我们更喜欢具名参数。

在应用程序中利用这个存储过程的查询看起来就像任何其他具名查询执行一样：

```
Query q = session.getNamedQuery("loadUsersByRank");
q.setParameter("rank", 12);
List result = q.list();
```

在编写本书之时，被映射的存储过程可以被启用为具名查询，如你在本节前面所做的那样，或者启用为用于实体的装载程序，类似于你在前面映射过的loadUser示例。

存储过程不仅可以查询和加载数据，还可以操作数据。第一种使用案例是大数据操作，在数据库层中执行。不应该在Hibernate中映射它，而是应该通过简单的JDBC：session.connection().prepareCallableStatement();等来执行它。在Hibernate中可以映射的数据操作是实体对象的创建、删除和更新。

3. 把CUD映射到过程

你前面为类把<sql-insert>、<sql-delete>和<sql-update>元素映射到定制SQL语句。如果不喜欢对这些操作使用存储过程，就把映射改为可调用的语句：

```
<class name="User">
    ...
    <sql-update callable="true" check="none">
        { call UPDATE_USER(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?) }
    </sql-update>
</class>
```

使用Hibernate的当前版本时，你面临着和以前相同的问题：把值绑定到占位参数。首先，存储过程必须有与Hibernate期待相同的输入参数数量（就像前面介绍的一样，启用SQL日志，获取可以复制和粘贴的生成语句）。这些参数必须再次按照Hibernate期待的相同顺序。

考虑check="none"属性。为了正确的（且如果启用了）乐观锁，Hibernate需要知道这个定制更新操作是否成功。通常，对于动态生成的SQL，Hibernate查看从操作中返回的被更新行的数量。如果操作没有或者无法更新任何行，就出现乐观锁失败。如果你编写自己的定制SQL操作，也可以定制这个行为。

利用check="none"，Hibernate期待你的定制过程内部处理失败的更新（例如，通过给需要被更新的行做一个版本检查），并期待过程在有东西出错时抛出异常。Oracle中的这种过程如下：

```
<database-object>
<create>
   create or replace procedure UPDATE_USER
    (IN_FIRSTNAME  in varchar,
     IN_LASTNAME   in varchar,
     IN_PASSWORD   in varchar,
     ...
    )
  as
    rowcount INTEGER;
  begin

     update USERS set
       FIRSTNAME    = IN_FIRSTNAME,
       LASTNAME     = IN_LASTNAME,
       "PASSWORD"   = IN_PASSWORD,
     where
       OBJ_VERSION = ...;

     rowcount := SQL%ROWCOUNT;
     if rowcount != 1 then
         RAISE_APPLICATION_ERROR( -20001, 'Version check failed');
     end if;

  end;

</create>
<drop>
    drop procedure UPDATE_USER
```

```
        </drop>
    </database-object>
```

SQL错误被Hibernate抓住，并被转换为一个随后可以在应用程序代码中处理的乐观锁异常。检查属性的其他选项如下：

- 如果启用check="count"，Hibernate就会利用简单的JDBC API检查被修改的行数。这是默认的，在你编写动态的SQL而没用存储过程时使用。
- 如果启用check="param"，Hibernate就会保存一个OUT参数，获取存储过程调用的返回值。你需要把一个额外的问号添加到调用，并在存储过程中返回这（第一）个OUT参数中DML操作的行数。然后Hibernate为你验证被修改的行数。

对插入和删除的映射很类似；所有这些都必须声明乐观锁检查如何执行。你可以从Hibernate启动日志中复制一个模板，获得参数的正确顺序和数量。

最后，也可以在Hibernate中映射存储函数。它们有着稍微不同的语义和用例。

4．映射存储函数

存储函数只有输入参数——没有输出参数。但是它可以返回值。例如，存储函数可以返回用户的等级：

```
<database-object>
    <create>
        create or replace function GET_USER_RANK
         (IN_USER_ID int)
        return int is
         RANK int;
        begin
            select
                RANKING
            into
                RANK
            from
                USERS
            where
                USER_ID = IN_USER_ID;

            return RANK;
        end;
    </create>
    <drop>
        drop function GET_USER_RANK
    </drop>
</database-object>
```

这个函数返回一个标量数。返回标量的存储函数的主要用例在常规的SQL或者HQL查询中嵌入调用。例如，可以获取比指定用户等级更高的所有用户：

```
String q = "from User u where u.ranking > get_user_rank(:userId)";
List result = session.createQuery(q)
                .setParameter("userId", 123)
                .list();
```

这是用HQL编写的查询；由于WHERE子句中（而不是在任何其他的子句中）用于函数调用的

传递（pass-through）功能，你可以直接调用数据库中的任何存储函数。函数的返回类型应该与操作相匹配：在这个例子中，包含ranking属性（也是数字）的大于比较。

如果函数返回一个结果集游标，如前几节一样，你甚至可以把它映射为具名查询，并让Hibernate把该结果集封送到对象图。

最后记住，尤其在遗留数据库中，存储过程和函数有时候不能在Hibernate中映射；在这种情况下，必须退回到简单的JDBC。有时候你可以用另一个包含Hibernate期待的参数接口的存储过程把遗留的存储过程包起来。一般的映射工具将涵盖许多变形和特殊案例。但是Hibernate的未来版本将改进映射能力——我们期待在不久的将来，会有更好的参数处理（没有更多的问号需要计算），并能支持任意的输入和输出参数。

你现在已经完成了运行时SQL查询和DML的定制。下面来转换一下视角，给数据库Schema的创建和修改定制SQL：DDL。

8.3 改进 Schema DDL

在Hibernate应用中定制DDL，通常是只有在利用Hibernate工具集生成数据库Schema时才要考虑的东西。如果已经有一个Schema，这样的定制将不会影响Hibernate的运行时行为。

可以把DDL导出到一个文本文件，或者每当运行集成测试时直接在数据库中运行它。因为DDL主要是特定于供应商的，你放到映射元数据中的每一个选项都有可能把元数据绑定到一个特定的数据库产品——当你应用以下特性时要记住这一点。

DDL定制可分成两类：

- 在映射元数据中显式地命名自动生成的数据库对象，例如表、列和约束，而不是依赖由Hibernate从Java类和属性名称中衍生的自动命名。我们已经在4.3.5节中讨论过引用和扩展命名的内建机制和选项。接下来我们看一下可以启用以美化生成的DDL脚本的其他选项。
- 在映射元数据中处理额外的数据库对象，例如索引、约束和存储过程。在本章前面，你用<database-object>元素把任意的CREATE和DROP语句添加到了XML映射文件。也可以利用常规类内部的额外映射元素和属性映射，启用索引和约束的创建。

8.3.1 定制 SQL 名称和数据类型

在代码清单8-1中，把属性和元素添加到Item类的映射中：

代码清单8-1 对hbm2ddl的Item映射中的额外元素

```
<class name="Item" table="ITEMS">

    <id name="id" type="string">
        <column name="ITEM_ID" sql-type="char(32)"/>   ❶
        <generator class="uuid"/>
    </id>

    <property name="initialPrice" type="big_decimal">
        <column name="INIT_PRICE"                      ❷
                not-null="true"
```

```
                precision="10"
                scale="2"/>
        </property>

        <property name="description" type="string"
                  column="ITM_DESCRIPTION" length="4000"/>

        <set name="categories" table="CATEGORY_ITEM" cascade="none">
            <key>
                <column name="ITEM_ID" sql-type="char(32)"/>
            </key>
            <many-to-many class="Category">
                <column name="CATEGORY_ID" sql-type="char(32)"/>
            </many-to-many>
        </set>

        ...

    </class>
```

❶ 如果属性（甚至标识符属性）是映射类型string，hbm2ddl导出器就会生成VARCHAR类型列。你知道标识符生成器uuid永远生成32个字符的字符串；因此你转换到CHAR SQL类型，并设置它的大小固定为32个字符。这个声明需要<column>元素，因为<id>元素中没有支持SQL数据类型的属性。

❷ 对于十进制的类型，你可以声明精确度和范围。这个例子用一种Oracle方言把列创建为INIT_PRICE number(10,2)；但是，对于不支持十进制精确度类型的数据库而言，就会（用HSQL）生成一个简单的INIT_PRICE numeric。

❸ 对于描述字段，是在<property>元素而不是嵌套的<column>元素中添加DDL属性。DESCRIPTION列被生成为VARCHAR(4000)——这是Oracle数据库中变量字符字段的一个限制（在Oracle中，在DDL中是VARCHAR2(4000)，但是方言负责这项工作）。

❹ <column>元素也可以用来在关联映射中声明外键字段。否则，关联表CATEGORY_ITEM的列就会是VARCHAR(32)而不是更精确的CHAR(32)类型。

注解中也可能有同样的定制，请见代码清单8-2。

代码清单8-2　用于定制DDL导出的额外注解

```
@Entity
@Table(name = "ITEMS")
public class Item {

    @Id
    @Column(name = "ITEM_ID", columnDefinition = "char(32)")
    @GeneratedValue(generator = "hibernate-uuid.hex")
    @org.hibernate.annotations.GenericGenerator(
            name = "hibernate-uuid.hex",
            strategy = "uuid.hex"
    )
    Private String id;

    @Column(name = "INIT_PRICE", nullable = false,
```

```
        precision = 10, scale = 2)
BigDecimal initialPrice;

@Column(name = "ITM_DESCRIPTION", length = 4000)
Private String description;

@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns =
        { @JoinColumn(name = "ITEM_ID",
                      columnDefinition = "char(32)")
        },
    inverseJoinColumns =
              { @JoinColumn(name = "CATEGORY_ID",
                            columnDefinition = "char(32)")
              }
      )
      Private Set<Category> categories = new HashSet<Category>();

      ...
  }
```

必须使用一个Hibernate扩展来命名非标准的标识符生成器。生成的SQL DDL的所有其他定制都通过JPA规范的注解完成。有一个属性值得特别关注：columnDefinition与Hibernate映射文件中的sql-type不同。它更加灵活：JPA持久化提供程序在CREATE TABLE语句中的列名称后面添加整个字符串，就像在ITEM_ID char(32)中一样。

名称和数据类型的定制是绝对应该考虑的。我们建议你始终利用适当的完整性规则改进数据库Schema的质量（并最终改进被存储数据的质量）。

8.3.2 确保数据一致性

完整性规则是数据库Schema的重要组成部分。数据库最重要的责任是保护信息，并保证它永远不会处于不一致的状态。这称作一致性，它是一般被应用到事务的数据库管理系统的ACID标准的一部分。

规则是业务逻辑的一部分，因此你通常在应用程序代码和数据库中混合实现业务相关的规则。这样编写应用程序是为了避免违背任何数据库规则。但是，数据库管理系统的任务是从不允许任何错误（从业务逻辑的意义来说）信息被永久保存——例如，如果其中一个访问数据库的应用程序有bug。只在应用程序代码中确保完整性的系统倾向于数据损坏，并经常随着时间的推移而降低数据库的质量。记住，大多数业务应用程序的主要目的从长远来说就是生成有价值的业务数据。

与在过程（或者面向对象的）应用程序代码中确保数据一致性相比，数据库管理系统则允许你把完整性规则声明式地实现为数据Schema的一部分。声明规则的好处包括减少了代码中的可能错误，以及数据库管理系统优化数据访问的机会。

我们来分辨一下四个级别的规则：

- 领域约束——领域是（不严格地说，在数据库范畴中）数据库中的一种数据类型。因此，

领域约束定义了一个特定的数据类型可以处理的可能值的范围。例如，int数据类型可以用于整数值。char数据类型可以保存字符串（例如，在ASCII中定义的所有字符）。因为我们主要使用创建在数据库系统中的数据类型，因此我们依赖由供应商定义的领域约束。如果创建用户自定义的数据类型（UDT），你将必须定义它们的约束。如果它们得到你的SQL数据库的支持，就可以使用对定制领域的（有限）支持，给特定的数据类型添加额外的约束。

- *列约束*——限制一个列保存特定领域的值，相当于增加一个列约束。例如，在DDL中声明INITIAL_PRICE列保存领域MONEY的值，它内部使用数据类型number(10,2)。你大多数时候直接使用数据类型，而不用事先定义领域。SQL数据库中一个特殊的列约束是NOT NULL。
- *表约束*——应用到单行或者几行的完整性规则是表约束。典型的声明表约束是UNIQUE（检查所有行中的重复值）。只影响单行的一个样例规则是“拍卖的终止日期应该晚于开始日期”。
- *数据库约束*——如果一项规则应用给不止一张表，它就具有数据库范围。你应该已经熟悉最常用的数据库约束：外键。这个规则保证了行之间引用的完整性，通常在独立的表中，但并非永远如此（自引用的外键约束并不罕见）。

大多数（如果不是全部）SQL数据库管理系统都支持上面提到的约束级别，以及每一种约束中最重要的选项。除了简单的关键字（例如NOT NULL和UNIQUE）之外，通常也可以声明带有应用了任意SQL表达式的CHECK约束的更复杂规则。但是，完整性约束却是SQL标准的软肋之一，并且来自不同供应商的解决方案有着明显的不同。

此外，对于拦截数据修改操作的数据库触发器来说，非声明约束和过程约束是可能的。然后，触发器可以直接实现约束过程，或者调用一个现有的存储过程。就像用于存储过程的DDL一样，你可以通过包含在生成的DDL中的<database-object>元素，把触发器声明添加到Hibernate映射元数据中。

最后，当数据修改语句被执行时，可以立即检查完整性约束，或者可以延期到事务结束时才进行检查。SQL数据库中的违背响应通常是拒绝，没有任何定制的可能。

现在要深入探讨一下完整性约束的实现。

8.3.3　添加领域约束和列约束

SQL标准包括领域，但是，它不仅相当受限制，而且经常不为DBMS所支持。如果你的系统支持SQL领域，就可以用它们把约束添加到数据类型：

```
create domain EMAILADDRESS as varchar
  constraint DOMAIN_EMAILADDRESS
  check ( IS_EMAILADDRESS(value) );
```

创建表的时候，你现在可以把这个领域标识符用作列类型：

```
create table USERS (
    ...
```

```
    USER_EMAIL EMAILADDRESS(255) not null,
    ...
);
```

SQL中领域（相对次要）的好处在于把一般的约束提取到一个单个位置。当插入或者修改数据时，领域约束始终立即得到检查。为了完成前一个例子，还必须编写存储函数IS_EMAILADDRESS（可以在网上找到许多正则表达式来完成这个）。在Hibernate映射中添加新领域与sql-type一样简单：

```
<property name="email" type="string">
    <column name="USER_EMAIL"
            length="255"
            not-null="true"
            sql-type="EMAILADDRESS"/>
</property>
```

利用注解声明自己的columnDefinition：

```
@Column(name = "USER_EMAIL", length = 255,
        columnDefinition = "EMAILADDRESS(255) not null")
String email;
```

如果想要利用剩下的Schema自动创建和删除领域声明，就把它放到<database-object>映射里面。

SQL支持额外的列约束。例如，业务规则允许用户注册名中只有字母和数字：

```
create table USERS (
    ...
    USERNAME varchar(16) not null
        check(regexp_like(USERNAME,'^[[:alpha:]]+$')),
    ...
);
```

可能在你的DBMS中不能使用这个表达式，除非它支持正则表达式。可以在Hibernate映射的<column>映射元素中声明单列检查约束：

```
<property name="username" type="string">
    <column name="USERNAME"
            length="16"
            not-null="true"
            check="regexp_like(USERNAME,'^[[:alpha:]]+$')"/>
</property>
```

在注解中检查约束只有在Hibernate扩展中才是可能的：

```
@Column(name = "USERNAME", length = 16,
        nullable = false, unique = true)
@org.hibernate.annotations.Check(
    constraints = "regexp_like(USERNAME,'^[[:alpha:]]+$')"
)
private String username;
```

注意，你有一种选择：创建和使用领域，或者添加单个列约束，作用都一样。长期而言，领域通常更易于维护，并且更可能避免重复。

我们来看看下一级规则：单行和多行表约束。

8.3.4　表级约束

想象你要保证CaveatEmptor拍卖不能在它开始之前被终止。你编写应用程序代码，防止用户把Item中的startDate和endDate属性设置为错误的值。可以在用户界面或者属性的设置方法中进行。在数据库Schema中，添加单个行的表约束：

```
create table ITEM (
    ...
    START_DATE timestamp not null,
    END_DATE timestamp not null,
    ...
    check (START_DATE < END_DATE)
);
```

表约束被添加在CREATE TABLE DDL中，并且可以包含任意的SQL表达式。你把约束表达式包括在Hibernate映射文件中的<class>映射元素上：

```
<class name="Item"
       table="ITEM"
       check="START_DATE &lt; END_DATE">
```

注意，< 这个字符必须用XML的<转义。利用注解，你需要添加一个Hibernate扩展注解来声明检查约束：

```
@Entity
@org.hibernate.annotations.Check(
    constraints = "START_DATE < END_DATE"
)
public class Item { ... }
```

多行的表约束可以通过更复杂的表达式实现。表达式中可能需要一个子查询来完成这个，你的DBMS可能不支持它——先查阅产品文档。但是有一些一般的多行表约束，可以在Hibernate映射中直接作为属性添加。例如，把User的注册名称标识为系统中唯一的：

```
<property name="username" type="string">
    <column name="USERNAME"
            length="16"
            not-null="true"
            check="regexp_like(USERNAME,'^[[:alpha:]]+$')"
            unique="true"/>
</property>
```

唯一约束声明也可能在注解元数据中：

```
@Column(name = "USERNAME", length = 16, nullable = false,
        unique = true)
@org.hibernate.annotations.Check(
    constraints = "regexp_like(USERNAME,'^[[:alpha:]]+$')"
)
private String username;
```

当然，还可以在JPA XML描述符中进行（但是没有检查约束）：

```
<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <basic name="username">
            <column name="USERNAME"
                    length="16"
                    nullable="false"
                    unique="true"/>
        </basic>
    </attributes>
</entity>
```

被导出的DDL包括unique约束：

```
create table USERS (
    ...
    USERNAME varchar(16) not null unique
        check(regexp_like(USERNAME,'^[[:alpha:]]+$')),
    ...
);
```

唯一约束也可以跨几个列。例如，CaveatEmptor支持嵌套的Category对象树。其中一个业务规则说，一个特定的分类不能与它的任何同胞同名。因此，你需要一个保证这种唯一性的多列多行约束：

```
<class name="Category" table="CATEGORY">
    ...
    <property name="name">
      <column name="CAT_NAME"
              unique-key="unique_siblings"/>
    </property>

    <many-to-one name="parent" class="Category">
      <column name="PARENT_CATEGORY_ID"
              unique-key="unique_siblings"/>
    </many-to-one>
    ...
</class>
```

使用unique-key属性把一个标识符分配给约束，以便可以在一个类映射中多次引用它，并把相同的约束分组成列。然而，在DDL中不用标识符命名约束：

```
create table CATEGORY (
    ...
    CAT_NAME varchar(255) not null,
    PARENT_CATEGORY_ID integer,
    ...
    unique (CAT_NAME, PARENT_CATEGORY_ID)
);
```

如果想要利用注解创建一个跨几列的唯一约束，就需要在实体而不是单个列中声明它：

```
@Entity
@Table(name = "CATEGORY",
   uniqueConstraints = {
        @UniqueConstraint(columnNames =
```

```
                {"CAT_NAME", "PARENT_CATEGORY_ID"} )
    }
)
public class Category { ... }
```

利用JPA XML描述符，多列约束如下：

```
<entity class="Category" access="FIELD">
    <table name="CATEGORY">
        <unique-constraint>
            <column-name>CAT_NAME</column-name>
            <column-name>PARENT_CATEGORY_ID</column-name>
        </unique-constraint>
    </table>
...
```

完全定制约束，包括一个用于数据库目录的标识符，可以通过<database-object>元素添加到DDL：

```
<database-object>
    <create>
        alter table CATEGORY add constraint UNIQUE_SIBLINGS
            unique (CAT_NAME, PARENT_CATEGORY_ID);
    </create>
    <drop>
        drop constraint UNIQUE_SIBLINGS
    </drop>
</database-object>
```

这项功能在注解中不可用。注意，可以利用你所有的定制数据库DDL对象，在基于注解的应用程序中添加Hibernate XML元数据文件。

最后，约束的最后一个目录包括跨几张表的适用于整个数据库的规则。

8.3.5　数据库约束

可以通过联结在任何check表达式的子查询中创建跨几张表的规则。不是只引用约束在其中被声明的表，你还可以查询不同的表（通常查询特定部分的信息存不存在）。

创建适用于整个数据库的约束的另一种技术是使用定制触发器，它们在特定表中行的插入或者更新中运行。这是一种过程方法，具有已经提到的缺点，但是天生灵活。

目前为止，大部分跨几张表的常用规则都是参照完整性（referential integrity）规则。它们是广为人知的外键，是两个东西的组合：一个从相关的行中复制来的键值，和一个保证引用值存在的约束。Hibernate自动地给关联映射中的所有外键列创建外键约束。如果你查看由Hibernate生成的DDL，可能注意到这些约束也已经自动生成数据库标识符——不容易阅读并且使调试更困难的一些名称：

```
alter table ITEM add constraint FK1FF7F1F09FA3CB90
    foreign key (SELLER_ID) references USERS;
```

这个DDL给ITEM表中的SELLER_ID列声明了外键约束。它引用USERS表的主键列。可以利用foreign-key属性在Item类的<many-to-one>映射中定制约束的名称：

```
<many-to-one name="seller"
             class="User"
             column="SELLER_ID"
             foreign-key="FK_SELLER_ID"/>
```

利用注解，使用Hibernate扩展：

```
@ManyToOne
@JoinColumn(name = "SELLER_ID")
@org.hibernate.annotations.ForeignKey(name = "FK_SELLER_ID")
private User seller;
```

给多对多关联创建的外键需要一种特殊的语法：

```
@ManyToMany
@JoinTable(...)
@org.hibernate.annotations.ForeignKey(
    name = "FK_CATEGORY_ID",
    inverseName = "FK_ITEM_ID"
)
private Set<Category> categories...
```

如果想要自动生成与人类数据库管理员编写的没有区别的DDL，就在映射元数据中定制你所有的外键约束。这不仅是个好实践，而且当你必须阅读异常消息时，它的帮助十分明显。注意，hbm2ddl导出器只对已经在一个双向关联映射的非反向端设置的外键考虑约束名称。

外键约束在SQL中也有遗留Schema可能已经使用过的特性。不要立即拒绝可能违背外键约束的数据修改，SQL数据库可以把这个变化CASCADE到引用行。例如，如果父行被删除了，父行主键上包含外键约束的所有子行也可能被删除。如果必须或者想要使用这些数据库级的级联选项，就在外键映射中启用它们：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids" cascade="save-update, delete">
        <key column="ITEM_ID" on-delete="cascade"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

Hibernate现在创建和依赖外键约束的数据库级ON CASCADE DELETE选项，而不是在Item实例被删除且所有出价必须被移除时，执行多个单独的DELETE语句。注意这个特性绕过了Hibernate的为了版本化数据而常用的乐观锁策略！

最后，与从业务逻辑中转变来的完整性规则不相关，数据库性能优化也成为典型的DDL定制工作的一部分。

8.3.6 创建索引

索引是优化数据库应用程序的性能时的一项关键特性。数据库管理系统中的查询优化器可以利用索引避免过度扫描数据表。因为它们只在数据库的物理实现中相关，索引不是SQL标准的一部分，并且DDL和可用的索引选项是专门用于特定产品的。然而，典型索引的最常用的DDL可以被嵌入在一个Hibernate映射中（也就是说，不用一般的<database-object>元素）。

CaveatEmptor中的许多查询都将可能涉及拍卖Item的endDate属性。可以通过对这个属性的列创建索引来加快这些查询：

```
<property    name="endDate"
             column="END_DATE"
type="timestamp"
index="IDX_END_DATE"/>
```

自动生成的DDL现在包括一条额外的语句：

```
create index IDX_END_DATE on ITEM (END_DATE);
```

注解也有相同的功能，作为Hibernate扩展：

```
@Column(name = "END_DATE", nullable = false, updatable = false)
@org.hibernate.annotations.Index(name = "IDX_END_DATE")
private Date endDate;
```

可以通过在几个属性（或者列）映射中设置相同的标识符来创建一个多列的索引。任何其他的索引选项，例如UNIQUE INDEX（它创建一个额外的多行表级约束）、索引方法（一般为btree、hash和binary）和任何存储子句（例如，在单个的表空间中创建索引）可以通过<database-object>只在完整定制的DDL中设置。

包含注解的多列索引在实体级中定义，通过把额外的属性应用到表映射的定制Hibernate注解：

```
@Entity
@Table(name="ITEMS")
@org.hibernate.annotations.Table(
  appliesTo = "ITEMS",  indexes =
    @org.hibernate.annotations.Index(
     name = "IDX_INITIAL_PRICE",
     columnNames = { "INITIAL_PRICE", "INITIAL_PRICE_CURRENCY" }
    )
)
public class Item { ... }
```

注意，@org.hibernate.annotations.Table不是@javax.persistence.Table的替代物，因此如果需要覆盖默认的表名称，仍然需要常规的@Table。

如果想要学习不同的数据库优化技术，尤其是索引如何让你更接近查询的最佳执行计划，我们建议你买本好书：Dan Tow所著的*SQL Tunning*（Tow，2003）。

本章已经多次介绍的一种映射是<database-object>。它有一些我们还没有讨论到的其他选项。

8.3.7　添加辅助的 DDL

Hibernate自动地给表和约束创建基础的DDL；如果你有特定的标识符生成器，它甚至可以创建序列。但是，存在一些Hibernate无法自动创建的DDL。这包括各种非常特定于供应商的性能选项和任何其他只与数据的物理存储（例如表空间）相关的DDL。

这种DDL没有映射元素或者注解的一个原因在于有太多的变形和可能性——没有人能够或

者想要维护超过25种数据库方言的DDL的所有可能组合。第二个更为重要的原因在于，你应该永远让数据库管理员来完成数据库Schema。例如，你知道外键列上的索引会在某些情况下损害性能，因此不用Hibernate自动生成吗？我们建议数据库管理员早点介入，并验证从Hibernate自动生成的DDL。

如果正从一个新应用程序和新数据库开始，一般的过程是在开发期间由Hibernate自动生成DDL；数据库性能关注点在这方面不应该并且通常也不扮演重要的角色。同时（或者迟一点，在测试期间），专业的数据库管理员验证并优化SQL DDL，并创建最终的数据库Schema。可以把DDL导出到文本文件，并把它交给数据库管理员。

或者可以把定制的DDL语句添加到映射元数据中（这种方法你可能已经见过多次）：

```
<database-object>
    <create>
        [CREATE statement]
    </create>
    <drop>
        [DROP statement]
    </drop>

    <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/>
    <dialect-scope name="org.hibernate.dialect.OracleDialect"/>

</database-object>
```

<dialect-scope>元素把定制的CREATE或者DROP语句限制为一个特定的被配置数据库方言组，如果你正在几个系统中部署并且需要不同的定制时，它很有用。

如果需要更多对于生成的DDL可编程式的控制，就实现AuxiliaryDatabaseObject接口。Hibernate包含一个可以子类化的便利实现；然后你可以有选择地覆盖方法：

```
package auction.persistence;

import org.hibernate.mapping.*;
import org.hibernate.dialect.Dialect;
import org.hibernate.engine.Mapping;

public class CustomDDLExtension
    extends AbstractAuxiliaryDatabaseObject {

    public CustomDDLExtension() {
        addDialectScope("org.hibernate.dialect.Oracle9Dialect");
    }

    public String sqlCreateString(Dialect dialect,
                                  Mapping mapping,
                                  String defaultCatalog,
                                  String defaultSchema) {

        return "[CREATE statement]";
    }

    public String sqlDropString(Dialect dialect,
                                String defaultCatalog,
                                String defaultSchema) {
```

```
            return "[DROP statement]";
        }
    }
```

可以通过编程来添加方言范围，甚至访问sqlCreateString()和sqlDropString()方法中的一些映射信息。这给你提供了许多关于如何创建和编写DDL语句的灵活性。必须在映射元数据中启用这个定制类：

```
<database-object>
    <definition class="auction.persistence.CustomDDLExtension"/>
    <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
</database-object>
```

额外的方言范围是累积的；前面的例子全都应用两种方言。

8.4　小结

本章探讨了当你必须处理遗留数据库Schema时可能遇到的问题。自然键、复合键和外键经常不方便，需要格外小心地映射。Hibernate也提供公式以及映射文件中的一些SQL表达式，它们可以帮助你处理无法改变的遗留Schema。

通常，你也依赖Hibernate在应用程序中为所有创建、读取、更新和删除操作而自动生成的SQL。在本章中，你已经学习了如何用自己的语句定制这个SQL，以及如何把Hibernate与存储过程和存储函数整合。

最后一节探讨了数据库Schema的生成，以及如何定制和扩展映射为包括各种约束、索引和数据库管理员可能建议的任意DDL。

表8-1展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表8-1　第8章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate支持任何种类的自然的和复合的主键，包括自然键的外键、复合主键和复合主键中的外键	为自然键和组合键提供标准支持，相当于Hibernate
Hibernate通过公式映射和属性引用支持任意的关联联结条件	在编写本书之时，还没有标准或者注解支持被分组的属性引用
Hibernate对特定的实体类支持二级表的基本联结	对二级表和基本联结提供标准支持
Hibernate支持触发器整合和生成的属性设置	Hibernate Annotations支持生成的属性和触发器整合
Hibernate让你利用XML映射元数据中的选项定制所有的SQL DML语句	在编写本书之时，还不支持利用注解的SQL DML定制
Hibernate让你给自动的模式生成定制SQL DDL。任意的SQL DDL语句可以被包括在XML映射元数据中	JPA标准化了基本的DDL声明，但是并非XML映射元数据的所有特性都能通过注解得到支持

你现在已经学习了关于把类映射到Schema的一切内容(我们尽可能详细地介绍了在一本书中所能够介绍的内容)。本书的下一部分，将讨论如何使用持久化管理器API来加载和保存对象，事务和会话如何实现，以及如何编写查询。

Part 3

第三部分

会话对象处理

本书的这部分将介绍如何使用持久化对象。第 9 章介绍如何利用 Hibernate 和 Java Persistence 编程接口加载和保存对象。事务和并发性控制是另一个重要的话题，在第 10 章详细讨论。然后在第 11 章实现会话，介绍这个概念如何改进系统设计。第 12 章和第 13 章则关注效率，以及在必须加载和修改大型且复杂的数据集时，Hibernate 特性如何使你变得更加轻松。查询、查询语言和 API 在第 14 章和第 15 章将得到深入的实践。在第 16 章中，我们通过设计和测试一个包含 ORM 持久化的分层应用程序，把它们集中到一起。

在读完这部分之后，你将学会如何使用 Hibernate 和 Java Persistence 编程接口，以及如何有效地加载、修改和保存对象。你将理解事务是如何工作的，以及为什么会话处理可以给应用程序设计打开新路子。你将能够优化任何对象修改场景、编写复杂的查询，并应用最佳的抓取和高速缓存策略来改善性能和增强可伸缩性。

本部分内容