第12章

有效修改对象

本章内容

- 传播性状态变化
- 批量和大批量处理
- 持久化生命周期的拦截

本章介绍如何使数据操作更有效率。我们优化和减少保存对象所必需的代码量，并讨论最有效的处理选项。你应该熟悉基础的对象状态和持久化接口；要理解本章必须先阅读前面的章节。

首先，介绍传播性持久化如何使你更容易地使用复杂的对象网络。你可以在Hibernate和Java Persistence应用程序中启用的级联选项，明显减少了在没有这些级联选项时同时插入、更新或者删除几个对象所必需的代码量。

然后讨论如何最好地处理大的数据集，是通过应用程序中的批量操作，还是通过直接在数据库中执行的大批量操作。

最后，介绍数据过滤和拦截，这两者都提供透明地钩入Hibernate引擎内部加载和保存的过程。这些特性让你影响或者参与对象的生命周期，而不用编写复杂的应用程序代码，不用绑定领域模型到持久化机制。

让我们从传播性持久化开始，同时保存多个对象。

12.1 传播性持久化

真正重要的应用程序不仅仅使用单独的对象，还使用对象网络。当应用程序操作持久化的对象网络时，结果可能会是一张由持久化的、脱管的和瞬时的实例组成的对象图。传播性持久化是一种允许你把持久化自动传播到瞬时的和脱管的子图的方法。

例如，如果你把一个刚被实例化的`Category`添加到categories的已经持久化的层次结构，它应该自动变成持久化，而不必调用`save()`或者`persist()`。6.4节映射`Bid`和`Item`之间的父/子关系时，介绍了一个稍微不同的例子。在这种情况下，当出价被添加到一件货品时，它们不仅自动变成持久化，而且当胜出货品被删除时它们也自动被删除。你有效地使`Bid`变成了完全依赖另一个实体`Item`的实体（`Bid`实体不是值类型，它仍然支持共享引用）。

传播性持久化有不止一个模型。最著名的是按可到达性持久化（persistence by reachability），我们先来讨论它。虽然有些基本原理是一样的，但Hibernate还是用了它自己的更为强大的模型，稍后你会看到。对于Java Persistence也一样，它也有传播性持久化的概念，并且具有Hibernate原生提供的几乎所有选项。

12.1.1　按可到达性持久化

每当应用程序从已经持久化的实例中创建对其中一个实例的引用时，如果任何实例变成了持久化，对象持久层便是在实现按可到达性持久化。这种行为如图12-1中的对象图（注意这不是一张类图）所示。

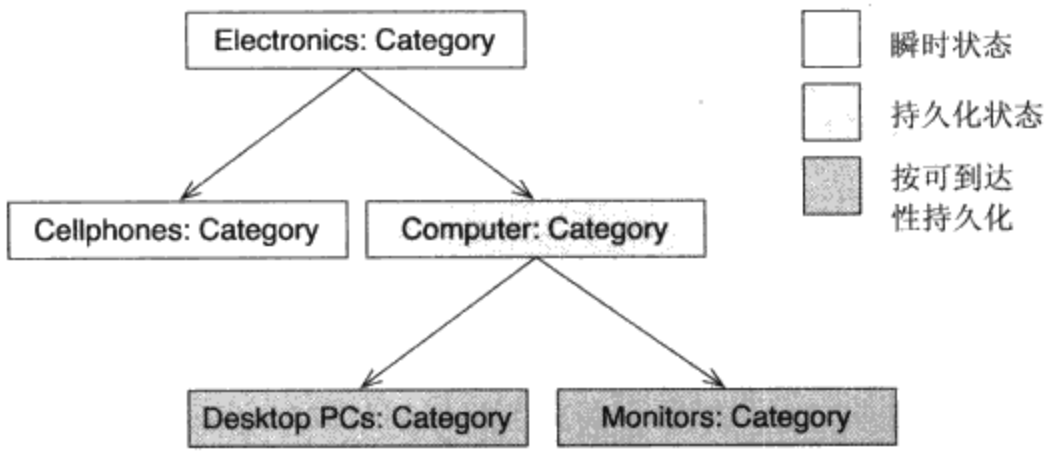

图12-1　包含根持久化对象的按可到达性持久化

在这个例子中，Computer是一个持久化对象。对象Desktop PCs和Monitors也是持久化对象：它们可以从Computer `Category`实例到达。Electronics和Cellphones是瞬时状态。注意我们假设导航只可能对子类进行，对父类则不行——例如，可以调用`computer.getChildCategories()`。按可到达性持久化是一种递归的算法。所有可以从持久化实例到达的对象都在原始实例被变成持久化时、或者就在内存状态与数据库同步之前变成持久化。

按可到达性持久化保证了参照完整性；任何对象图都可以通过加载持久化根对象被完全重新创建。应用程序可以逐个关联地穿过对象图，从来不必担心实例的持久化状态。（SQL数据库有一种不同的参照完整性的方法，它依赖声明和过程的约束来侦测行为异常的应用程序。）

在最纯粹的按可到达性持久化的形式中，数据库有一些顶级对象或者根对象，所有持久化对象都可以从那里到达。理想情况下，如果它不能通过引用从根持久化对象到达，实例应该变成瞬时状态并且就要从数据库中被删除。

Hibernate和其他的ORM解决方案都没有实现这一点——事实上，在SQL数据库中没有类似的根持久化对象，也没有持久化垃圾收集器可以侦测没有被引用的实例。面向对象的数据存储可以实现垃圾收集算法，类似于JVM给内存对象实现的那种。但是这个选项在ORM领域中不可用；扫描所有的表来查找没有被引用的行，执行起来并不令人满意。

因此，按可到达性持久化最多是个不彻底的解决方案。它帮助你使瞬时对象变成持久化，并把它们的状态传播到数据库，而不需要对持久化管理器的诸多调用。然而，至少在SQL数据库和

ORM的上下文中，它并不是使持久化对象变成瞬时（把它们的状态从数据库中移除）这个问题的完整解决方案。这样就出现了更大的难题。当你移除一个对象时，却无法移除所有可到达的实例——其他的持久化实例可能仍然持有对它们的引用（记住实体可以被共享）。你甚至无法安全地移除那些没有被内存中的任何持久化对象引用的实例；内存中的实例只在数据库中表示所有对象的一个小子集。

来看一下Hibernate更灵活的传播性持久化模型。

12.1.2 把级联应用到关联

Hibernate的传播性持久化模型使用与按可到达性持久化相同的基本概念：检查对象关联来确定传播性状态。此外，Hibernate还允许你给每个关联映射指定一种级联样式（cascade style），它为所有的状态转变提供了更多灵活性和更细粒度的控制。Hibernate读取被声明的样式，并自动把操作级联到被关联的对象。

默认情况下，当搜索瞬时的或者脱管的对象时，Hibernate并不导航关联，因此保存、删除、重附、合并等，`Category`对于被父类别的`childCategories`集合引用的任何子类别都没有影响。这与按可到达性持久化的默认行为相反。对于特定的关联而言，如果你希望启用传播性持久化，就必须在映射元数据中覆盖这个默认。

这些设置被称作级联的选项。它们使用XML和注解语法，可用于每一种实体关联映射（一对一、一对多、多对多）。请见表12-1中所有设置的清单和每种选项的描述。

在XML映射元数据中，把`cascade="..."`属性放在`<one-to-one>`或者`<many-to-one>`映射元素中，来启用传播性状态变化。所有的集合映射（`<set>`、`<bag>`、`<list>`和`<map>`）都支持`cascade`属性。然而，`delete-orphan`设置只适用于集合。很显然，你永远不必给引用值类型类的集合启用传播性持久化——这里被关联对象的生命周期是依赖和隐式的。依赖型生命周期的细粒度控制只与实体之间的关联相关，且只对它们可用。

表12-1 Hibernate和Java Persistence实体关联级联选项

XML属性	注解描述
None	（默认） Hibernate忽略关联
save-update	`Org.hibernate.annotations.CascadeType.SAVE_UPDATE` 当`Session`被清除，且对象被传递到`save()`或者`update()`时，Hibernate就导航关联，并保存刚刚被实例化的瞬时实例和把变化持久化到脱管状态的实例
persist	`javax.persistence.CascadeType.PERSIST` 当对象传递到`persist()`时，Hibernate使任何被关联的瞬时实例都变成持久化。如果使用原生的Hibernate，级联则只在调用时发生。如果使用`EntityManager`模块，这项操作则在持久化上下文清除时被级联
merge	`javax.persistence.CascadeType.MERGE` Hibernate导航关联，并在对象传递到`merge()`时，通过相当的持久化实例合并被关联的脱管实例。可到达的瞬时实例被变成持久化
delete	`org.hibernate.annotations.CascadeType.DELETE` 当对象被传递到`delete()`或者`remove()`时，Hibernate就导航关联，并删除被关联的持久化实例

（续）

XML属性	注解描述
remove	javax.persistence.CascadeType.REMOVE 当对象被传递到remove()或者delete()时，这个选项就把级联删除启用到被关联的持久化实例
lock	org.hibernate.annotations.CascadeType.LOCK 这个选项把lock()操作级联到被关联的实例，如果对象是脱管的，就把它们重附到持久化上下文。注意LockMode没有被级联；Hibernate假设你在被关联对象上不想要悲观锁——例如，因为根对象上的悲观锁已经足以避免并发修改
replicate	org.hibernate.annotations.CascadeType.REPLICATE Hibernate导航该关联，并把replicate()操作级联到被关联的对象
evict	org.hibernate.annotations.CascadeType.EVICT 当对象被传递到Hibernate Session中的evict()时，Hibernate从持久化上下文清除被关联的对象
refresh	javax.persistence.CascadeType.REFRESH 当对象被传递到refresh()时，Hibernate从数据库中重新读取被关联对象的状态
all	javax.persistence.CascadeType.ALL 这项设置包括并启用前面列出的所有级联选项
deleteorphan	org.hibernate.annotations.CascadeType.DELETE_ORPHAN 当被关联的对象从关联（即集合中）被移除时，这项额外的特殊设置启用了它们的删除。如果你在一个实体集合中启用这项设置，就等于告诉Hibernate：被关联的对象没有共享的引用，且当引用从集合中被移除时，它可以被安全地删除

常见问题　cascade和inverse之间是什么关系？没有关系；两者是不同的概念。关联的非反向端被用来在数据库中生成管理关联的SQL语句（若干外键列的插入和更新）。级联启用了跨实体类关联的传播性对象状态变化。

这里有几个在XML映射文件中的级联选项示例。注意这段代码不是来自单个实体映射或者单个类，只是用来举例说明：

```
<many-to-one name="parent"
             column="PARENT_CATEGORY_ID"
             class="Category"
             cascade="save-update, persist, merge"/>

...

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

...

<set name="bids" cascade="all, delete-orphan"
     inverse="true">
    <key column ="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

如你所见，几个级联选项可以合并，并作为一个以逗号分隔的列表应用到特定的关联。还要注意delete-orphan没有包括在all中。

级联选项通过注解以两种可能的方式进行声明。第一，所有的关联映射注解（@ManyToOne、@OneToOne、@OneToMany和@ManyToMany）都支持cascade属性。这项属性的值是单个或者一列javax.persistence.CascadeType值。例如，通过注解来完成的XML示例映射看起来像下面这样：

```
@ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
@JoinColumn(name = "PARENT_CATEGORY_ID", nullable = true)
private Category parent;

...

@OneToMany(cascade = CascadeType.ALL)
private Set<Bid> bids = new HashSet<Bid>();
```

很显然，并非所有的级联类型在标准的javax.persistence包中都可用。只有与EntityManager操作相关的级联选项，如persist()和merge()被标准化。必须使用一个Hibernate扩展注解来应用任何只适用于Hibernate的级联选项：

```
@ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
@org.hibernate.annotations.Cascade(
    org.hibernate.annotations.CascadeType.SAVE_UPDATE
)
@JoinColumn(name = "PARENT_CATEGORY_ID", nullable = true)
private Category parent;

...

@OneToOne
@org.hibernate.annotations.Cascade({
    org.hibernate.annotations.CascadeType.SAVE_UPDATE,
    org.hibernate.annotations.CascadeType.LOCK
})
@PrimaryKeyJoinColumn
private Address shippingAddress;

...

@OneToMany(cascade = CascadeType.ALL)
@org.hibernate.annotations.Cascade(
    org.hibernate.annotations.CascadeType.DELETE_ORPHAN
)
private Set<Bid> bids = new HashSet<Bid>();
```

Hibernate扩展级联选项可以用作已经在关联注解中设置的选项的补充（第一个和最后一个例子），如果没有应用标准的选项，也可以用作一项独立的设置（第二个例子）。

Hibernate的关联级别级联样式模型比按可到达性持久化更丰富且更不安全。Hibernate没有像按可到达性持久化一样对参照完整性提供强大的保证。反之，Hibernate委托了部分参照完整性关注点给底层SQL数据库的外键约束。

这一设计决策有一个很好的解释：它允许Hibernate应用程序有效地使用脱管对象，因为你可以在关联级别控制脱管对象图的重附和合并。但是级联选项不是只可用来避免不必要的重附或者

合并：每当需要同时处理多个对象时，它们也很有用。

让我们用一些示例关联映射详细阐述传播性状态概念。我们建议你按顺序阅读接下来的小节，因为每个示例都是建立在前一个的基础上。

12.1.3　使用传播性状态

CaveatEmptor管理员能够创建新的类别，重新命名类别，并在分类层次结构中四处移动子类别。这个结构如图12-2所示。

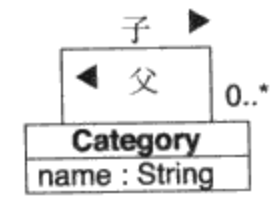

图12-2　Category类包含对自身的关联

现在，用XML映射这个类和关联：

```
<class name="Category" table="CATEGORY">
    ...
    <property name="name" column="CATEGORY_NAME"/>

    <many-to-one name="parentCategory"
                 class="Category"
                 column="PARENT_CATEGORY_ID"
                 cascade="none"/>

    <set name="childCategories"
         table="CATEGORY"
         cascade="save-update"
         inverse="true">
        <key column="PARENT_CATEGORY_ID"/>
        <one-to-many class="Category"/>
    </set>
    ...
</class>
```

这是一个递归的双向一对多关联。一值端通过<many-to-one>元素和包含<set>的Set类型属性被映射。两者都指向相同的外键列PARENT_CATEGORY_ID。所有的列都处在同一张表CATEGORY中。

1．创建新类别

假设你创建新的Category，作为Computer的子类别；请见图12-3。

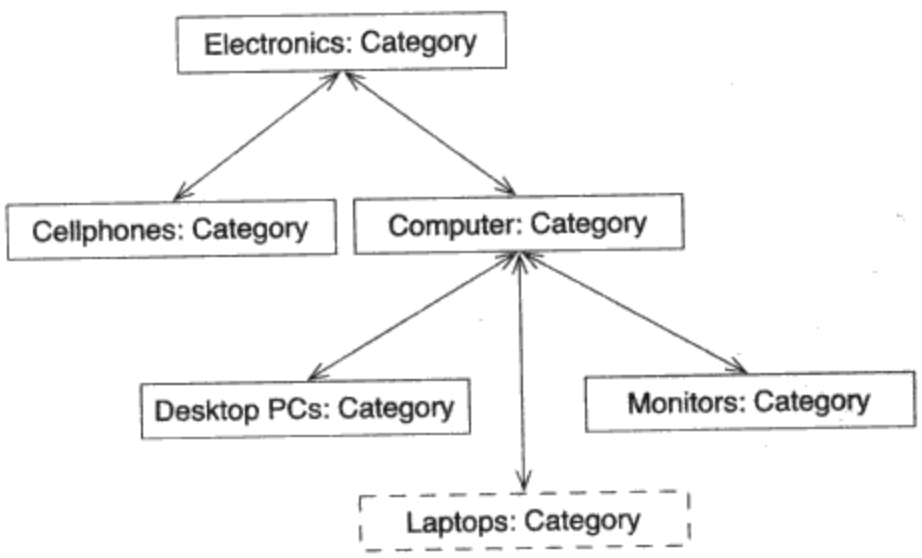

图12-3　把新的Category添加到对象图

创建这个新的Laptops对象并把它保存在数据库中有好几种方法。你可以回到数据库并获取新Laptops类别，将要所属的Computer类别添加这个新类别，并提交事务：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Category computer =
            (Category) session.load(Category.class, computerId);

Category laptops = new Category("Laptops");

computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);

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

computer实例是持久化的（注意如何将load()与代理共用，并避免数据库命中），且childCategories关联启用了级联保存。因而，这段代码在tx.commit()被调用时导致新的laptops类别变成持久化，就像Hibernate把持久化状态级联到computer的childCategories集合元素一样。当持久化上下文被清除，且排列INSERT语句时，Hibernate就会检查对象的状态和它们的关系。

2. 以脱管方式创建新类别

再次完成同样的事，但这次是在持久化上下文范围之外创建Computer和Laptops之间的连接：

```
Category computer =
            (Category) session.get() // Loaded in previous Session

Category laptops = new Category("Laptops");

computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
```

你现在有了脱管的、完全实例化（没有代理）的computer对象，它在前一个Session中加载，并与新的瞬时laptops对象关联（反之亦然）。你通过把这个新对象保存在第二个Hibernate Session——一个新的持久化上下文中，使在对象上所做的变化持久化：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

// Persist one new category and the link to its parent category
session.save(laptops);

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

Hibernate检查laptops.parentCategory对象的数据库标识符属性，并在数据库中正确创建对Computer类别的引用。Hibernate把父标识符值插入到CATEGORY中新Laptops行的外键字段。

在这个例子中，你无法给computer获得脱管的代理，因为computer.getChildCategories()将触发代理的初始化，你会看到LazyInitializationException: Session已经关闭。你无法在脱管状态下跨过未被初始化的边界来导航对象图。

由于你给parentCategory关联定义了cascade="none"，Hibernate忽略对层次结构

（Computer，Electronics）中任何其他类所做的改变！它不把对save()的调用级联到被这个关联引用的实体。如果在parentCategory的<many-to-one>映射中启用了cascade="save-update"，Hibernate就会在内存中导航整个对象图，使所有的实例都与数据库同步。这是你想要避免的一项明显的过载。

在这个例子中，对于parentCategory关联，你既不需要也不想要传播性持久化。

3．用传播性持久化保存几个新实例

为什么要有级联操作？如前一个例子所示，你可以不用任何级联映射保存laptop对象。那么，考虑下列案例：

```
Category computer = ... // Loaded in a previous Session

Category laptops = new Category("Laptops");
Category laptopUltraPortable = new Category("Ultra-Portable");
Category laptopTabletPCs = new Category("Tablet PCs");

laptops.addChildCategory(laptopUltraPortable);
laptops.addChildCategory(laptopTabletPCs);

computer.addChildCategory(laptops);
```

（注意，便利方法addChildCategory()在一个调用中设置了关联连接的两端，如本书前面所述。）

如果必须把这三个新类别的每一个保存在单独的新Session中，这可不是我们想要的。幸运的是，因为你用cascade="save-update"映射了childCategories关联（集合），因此不必那么辛苦。与前面所示的代码一样，它保存了单个Laptops类别，将在一个新Session中保存所有这三个新类别：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

// Persist all three new Category instances
session.save(laptops);

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

你或许想知道为什么级联样式被称作cascade="save-update"，而不仅仅是cascade="save"。前面刚刚使所有三个类别变成持久化，假设在接下来的事件中，你在Session外部对类别层次结构进行下面的改变（你正再一次使用脱管对象）：

```
laptops.setName("Laptop Computers");                    // Modify
laptopUltraPortable.setName("Ultra-Portable Notebooks"); // Modify
laptopTabletPCs.setName("Tablet Computers");            // Modify

Category laptopBags = new Category("Laptop Bags");
laptops.addChildCategory(laptopBags);                   // Add
```

你添加一个新类别（laptopBags）作为Laptops类别的一个子类别，并修改所有三个现有的类别。下列代码把所有这些变化传播到数据库：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

```
// Update three old Category instances and insert the new one
session.saveOrUpdate(laptops);

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

由于你在childCategories集合中指定了cascade="save-update"，Hibernate确定需要用什么把对象持久化到数据库。在这个例子中，它排列三条SQL UPDATE语句（用于laptops、laptopUltraPortable和laptopTablePCs）和一条INSERT（用于laptopBags）。saveOrUpdate()方法告诉Hibernate把实例的状态传播到数据库：如果该实例是一个新的瞬时实例，就通过创建一个新的数据库行；如果该实例是一个脱管实例，就通过更新现有的行。

更有经验的Hibernate用户专门使用saveOrUpdate()；它更容易让Hibernate决定什么是新的以及什么是旧的，尤其在混合状态的更复杂对象图中的时候。这个专门的saveOrUpdate()的唯一缺点在于，它在数据库中没有触发SELECT的情况下，有时无法猜测一个实例是旧的还是新的——例如，当类用一个自然的复合键映射，并且没有版本或者时间戳属性的时候。

Hibernate如何发现哪个实例是旧的以及哪个实例是新的呢？有许多选项可用。Hibernate假设实例是未被保存的瞬时实例，如果：

- 标识符属性为null。
- 版本或者时间戳属性（如果存在的话）为null。
- 同一个持久化类的新实例（由Hibernate内部创建）有着与指定实例相同的数据库标识符值。
- 你在映射文档中给类提供一个unsaved-value，并且标识符属性的值匹配。unsaved-value属性对于版本和时间戳映射元素也可用。
- 包含相同标识符值的实体数据不在二级高速缓存中。
- 你提供org.hibernate.Interceptor的一个实现，并在检查完你自己代码中的实例之后，从Interceptor.isUnsaved()返回Boolean.TRUE。

在CaveatEmptor领域模型中，你随处使用可为空的类型java.lang.LONG作为标识符属性类型。因为你正在使用生成的合成标识符，这样就解决了问题。新实例有一个null标识符属性值，因此Hibernate把它们当作瞬时实例对待。脱管实例有一个非空的标识符值，因此Hibernate对它们做相应的处理。

几乎没有必要定制创建到Hibernate中去的自动侦测子程序。saveOrUpdate()方法始终知道如何处理指定的对象（或者任何可到达的对象，如果给关联启用save-update的级联的话）。然而，如果你使用自然的复合键，并且实体中没有版本或者时间戳属性，Hibernate就必须用SELECT命中数据库，查看是否已经存在包含相同复合标识符的行。换句话说，建议你通常使用saveOrUpdate()而不是单独的save()或者update()方法，Hibernate很聪明，足以正确地处理，并使传播性的“无论新旧，所有这些都应该处于持久化状态”更易于处理。

现在已经讨论了Hibernate中基本的传播性持久化选项，通过尽可能少的代码行保存实例和重附脱管实例。大多数的其他级联选项都相当容易理解：persist、lock、replicate和evict完成你预期的工作——它们使特定的Session操作具有传播性。merge级联选项实际上有着与save-update一样的效果。

事实表明，对象删除更难以掌握；delete-orphan设置尤其对刚接触Hibernate的用户造成混淆。这不是因为它复杂，而是因为许多Java开发人员往往忘了他们正在使用一个指针网络。

4．考虑传播性删除

想象你想要删除Category对象。必须把这个对象传递到Session中的delete()方法；现在它处于移除状态，并且当清除或者提交持久化上下文时，将离开数据库。然而，如果任何其他的Category当时持有对被删除行的引用（可能由于它仍然作为其他类别的父类别被引用），就会得到一个外键约束违例（foreign key constraint violation）。

你有责任在删除实例之前删除对Category的所有链接。这是支持共享引用的实体的一般行为。当自己的实体实例被删除时，实体实例的任何值类型属性（或者组件）值也自动被删除。如果你从自己的集合中移除引用，值类型集合元素（例如，Item的Image对象集合）也会被删除。

在某些情况下，你想要通过从集合中移除引用来删除实体实例。换句话说，你可以保证，一旦从集合中移除对这个实体的引用，就不会存在其他的引用。因此，Hibernate可以在已经移除了最后一个引用之后安全地删除实体。Hibernate假设没有引用的孤儿实体应当被删除。在示例的领域模型中，你在Item的映射中给bids的集合启用了这个特殊的级联样式（它只对集合可用）：

```
<set name="bids"
     cascade="all, delete-orphan"
     inverse="true">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

现在可以通过从这个集合中移除它们来删除Bid对象了——例如，在脱管状态下：

```
Item anItem = ... // Loaded in previous Session

anItem.getBids().remove(aBid);
anItem.getBids().remove(anotherBid);

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.saveOrUpdate(anItem);

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

如果没有启用delete-orphan选项，就必须在从集合中移除最后一个对它们的引用之后，显式地删除Bid实例：

```
Item anItem = ... // Loaded in previous Session

anItem.getBids().remove(aBid);
anItem.getBids().remove(anotherBid);

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.delete(aBid);
session.delete(anotherBid);

session.saveOrUpdate(anItem);
```

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

孤儿的自动删除为你节省了两行代码——不合时宜的两行代码。如果没有孤儿删除，则你必须记住想要删除的所有Bid对象——从集合中移除元素的代码，经常与执行delete()操作的代码处在不同的层中。通过启用孤儿删除，可以从集合中移除孤儿，并且Hibernate将假设它们不再被任何其他的实体引用。再次提醒，如果映射一个组件的集合，孤儿删除就是隐式的；额外的选项只与实体引用的集合相关（通常是<one-to-many>）。

Java Persistence和EJB 3.0也支持跨实体关联的传播性状态变化。标准的级联选项与Hibernate的类似，因此可以很容易地学会它们。

12.1.4 利用 JPA 的传播性关联

Java Persistence规范对于启用了级联对象操作的实体关联支持注解。就像在原生的Hibernate中一样，每个EntityManager操作都有一个相当的级联样式。例如，考虑通过注解映射的Category树（父/子关联）：

```
@Entity
public class Category {

    private String name;

    @ManyToOne
    public Category parentCategory;

    @OneToMany(mappedBy = "parentCategory",
               cascade = { CascadeType.PERSIST,
                           CascadeType.MERGE }
               )
    public Set<Category> childCategories = new HashSet<Category>();
    ...
}
```

你给persist()和merge()操作启用了标准的级联选项。现在可以在持久化和脱管状态下创建和修改Category实例，就像之前通过原生的Hibernate所做的那样：

```
Category computer = ... // Loaded in a previous persistence context

Category laptops = new Category("Laptops");
Category laptopUltraPortable = new Category("Ultra-Portable");
Category laptopTabletPCs = new Category("Tablet PCs");

laptops.addChildCategory(laptopUltraPortable);
laptops.addChildCategory(laptopTabletPCs);

computer.setName("Desktops and Laptops");
computer.addChildCategory(laptops);
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

computer = em.merge(computer);
```

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

对merge()的单个调用，使得所有修改和新增都变成了持久化。记住，merge()与重附不同：在合并之后，它返回一个新值（它应该被绑定到作为句柄的当前变量）到当前状态。

有些级联选项没有被标准化，而是Hibernate特有的。如果你正在使用Session API，或者如果你想要一个额外的设置用于EntityManager（例如org.hibernate.annotations.CascadeType.DELETE_ORPHAN），就映射这些注解（它们全部都在org.hibernate.annotations包中）。但是要注意，在其他的纯JPA应用程序中的定制级联选项引入了隐式的对象状态改变（它可能难以与不想要它们的人们进行通信）。

在前面几节中，我们已经探讨了通过Hibernate XML映射文件和Java Persistence注解的实体关联级联选项。利用传播性状态改变，你通过让Hibernate把修改导航和级联到被关联的对象，节省了代码行。建议你把领域模型所处的关联当作传播性状态改变的备选，然后通过级联选项实现它。在实践中，如果你也在领域模型［带有版型（stereotype）］的UML图中，或者在开发人员之间共享的任何其他类似文档中记下了级联选项，它是相当有帮助的。这么做改善了开发团队内部的交流，因为每个人都知道哪些操作和关联意味着级联状态的改变。

传播性持久化不是你可以通过单个操作来操作许多对象的唯一方法。许多应用程序都必须修改大对象集：例如，想象你必须在50 000个Item对象中设置标记。这是在数据库中直接执行得最好的大批量操作。

12.2　大批量和批量操作

你用ORM把数据移到应用层中，以便可以用一种面向对象的编程语言来处理这一数据。如果你正在实现一个多用户的在线事务处理应用程序，每个工作单元中都包含从少量至中等容量的数据集，那么这是一种好策略。

另一方面，需要大量数据的操作最好不要在应用层中执行。你应该把操作移近数据的位置，而不是其他地方。在SQL系统中，DML语句UPDATE和DELETE直接在数据库中执行，如果必须实现一个涉及上千行的操作，这些通常就足够了。更复杂的操作可能需要更复杂的过程在数据库内部运行；因此，你应该把存储过程当作一种可能的策略。

在Hibernate和Java Persistence应用程序中，你始终可以退回到JDBC和SQL。本节将介绍如何避免这一点，以及如何通过Hibernate和JPA执行大批量和批量操作。

12.2.1　使用 HQL 和 JPA QL 的大批量语句

HQL类似于SQL。这两者之间的区别在于HQL使用类名称而不是表名称，使用属性名称而不是列名称。它也理解继承——也就是说，无论你正在使用超类还是接口进行查询。

JPA查询语言，就像JPA和EJB 3.0定义的一样，是HQL的一个子集。因此，所有有效的JPA QL查询和语句对于HQL也是有效的。现在我们就要介绍这些语句，用于在数据库中直接执行的大批量操作，它们在JPA QL和HQL中均可用。（Hibernate采用了来自JPA的标准大批量操作。）

这些可用的语句支持在数据库中直接更新和删除对象，不需要把对象获取到内存中。也提供可以选择数据并把它当作新实体对象插入的语句。

1．直接在数据库中更新对象

前面的章节已经重申过你应该考虑对象的状态管理，而不是SQL语句如何管理。这种策略假设你正在引用的对象在内存中可用。如果执行一个直接在数据库中的行上操作的SQL语句，那么你所做的任何改变都不影响内存对象（无论它们可能处于什么状态）。换句话说，任何直接的DML语句都绕过了Hibernate持久化上下文（和所有高速缓存）。

避免这个问题的务实解决方案是一个简单的会话：首先在一个新的持久化上下文中执行任何直接的DML操作。然后，用Hibernate `Session`或者`EntityManager`加载和保存对象。这个会话保证持久化上下文不受任何之前执行的语句影响。另一种方法是，如果你知道它已经在持久化上下文的背后被修改了的话，可以选择性地使用`refresh()`操作从数据库中重载持久化对象的状态。

Hibernate和JPA提供DML操作，比简单的SQL更强大一点。让我们看看HQL和JPA QL中的第一项操作`UPDATE`：

```
Query q =
   session.createQuery("update Item i set i.isActive = :isActive");
q.setBoolean("isActive", true);
int updatedItems = q.executeUpdate();
```

这条HQL语句（或者JPA QL语句，如果通过`EntityManager`执行的话）看起来就像一条SQL语句。但是，它使用实体名称（类名称）和属性名称。它也被整合到Hibernate的参数绑定API中。返回被更新实体对象的数目——而不是被更新行的数目。另一个好处在于HQL（JPA QL）`UPDATE`语句适用于继承层次结构：

```
Query q = session.createQuery(
    "update CreditCard set stolenOn <= :now where type = 'Visa'"
);
q.setTimestamp("now", new Date() );
int updatedCreditCards = q.executeUpdate();
```

持久化引擎知道如何执行这项更新，即使必须生成几条SQL语句；它更新几张基表（因为`CreditCard`被映射到几个超类和子类表）。这个例子也没有给实体类包含别名——它是可选的。但是，如果你使用了别名，所有的属性则都必须加上别名作为前缀。还要注意，HQL（和JPA QL）`UPDATE`语句可以只引用单个实体类；例如，你无法编写单个语句来同时更新`Item`和`CreditCard`对象。`WHERE`子句中允许子查询；只有在这些子查询中才允许任何联结。

默认情况下，直接的DML操作对于被影响实体的任何版本或者时间戳值（这在Java Persistence中被标准化了）没有影响。然而，通过HQL，你可以增加直接被修改的实体实例的版本号：

```
Query q =
   session.createQuery(
        "update versioned Item i set i.isActive = :isActive"
   );
q.setBoolean("isActive", true);
int updatedItems = q.executeUpdate();
```

（如果你的版本或者时间戳属性依赖定制的org.hibernate.usertype.UserVersionType，则不允许用versioned关键字。）

引入的第二个HQL（JPA QL）大批量操作是DELETE：

```
Query q = session.createQuery(
    "delete CreditCard c where c.stolenOn is not null"
);
int updatedCreditCards = q.executeUpdate();
```

应用与UPDATE语句相同的规则：没有联结，只有单个实体类、可选的别名，WHERE子句中允许子查询。

就像SQL大批量操作一样，HQL（和JPA QL）大批量操作不影响持久化上下文，它们绕过任何高速缓存。如果你执行这其中一个例子，内存中的信用卡或者货品则不会被更新。

最后一个HQL大批量操作可以直接在数据库中创建对象。

2．在数据库中直接创建新对象

假设所有客户的Visa卡都被偷了。你编写两个大批量操作来标记被偷的日期（即发现被偷的日期），并从记录中移除其安全受到危及的信用卡的数据。因为你为一家有责任感的公司工作，必须向主管和受影响的客户报告被偷的信用卡。因此，在删除这些记录之前，提取了被偷的一切信息，并创建几百（或者几千）个StolenCreditCard对象。这是专为这一用途而编写的一个新类：

```
public class StolenCreditCard {

    private Long id;
    private String type;
    private String number;
    private String expMonth;
    private String expYear;
    private String ownerFirstname;
    private String ownerLastname;
    private String ownerLogin;
    private String ownerEmailAddress;
    private Address ownerHomeAddress;

    ... // Constructors, getter and setter methods
}
```

现在通过XML文件或者JPA注解（你自己完成这些不应该有任何问题了），把这个类映射到它自己的STOLEN_CREDIT_CARD表中。接下来，你需要一条直接在数据库中执行的语句，获取所有安全受到危及的信用卡，并创建新的StolenCreditCard对象：

```
Query q = session.createQuery(
    "insert into StolenCreditCard
           (type, number, expMonth, expYear,
            ownerFirstname, onwerLastname, ownerLogin,
            ownerEmailAddress, ownerHomeAddress)
     select
            c.type, c.number, c.expMonth, c.expYear,
            u.firstname, u.lastname, u.username,
            u.email, u.homeAddress
       from CreditCard c join c.user u
```

```
        where c.stolenOn is not null"
);

int createdObjects = q.executeUpdate();
```

这项操作完成了两件事：首先，选择CreditCard记录和对应的所有者（User）的详细信息。然后结果被直接插入到被映射到StolenCreditCard类上的表中。

注意下面几点：

- INSERT ... SELECT的目标属性（在这个例子中，是你列出的StolenCreditCard属性）必须针对一个特定的子类，而不是（抽象的）超类。由于StolenCreditCard不是继承层次结构的一部分，因此这不成问题。
- SELECT返回的类型必须与INSERT要求的类型相匹配——在这个例子中，指大量的string类型和一个组件（与用于选择和插入相同的组件类型）。
- 用于每个StolenCreditCard对象的数据库标识符，将通过标识符生成器（你使用它来映射该数据库标识符）自动生成。另一种方法是，你可以把标识符属性添加到一个被插入的属性的清单中，并提供通过选择的值。注意，标识符值的自动生成仅适用于直接在数据库内部操作的标识符生成器，如序列或者同一性字段。
- 如果生成的对象具有版本控制的类（通过version或者timestamp属性），也会生成新的版本（0或者当天的时间戳）。另一种方法是，可以选择一个版本（或者时间戳）值，并把版本（或者时间戳）属性添加到被插入的属性列表中。

最后，注意INSERT ... SELECT只可用于HQL；JPA QL没有标准化这种语句——因此，你的语句是不可移植的。

HQL和JPA QL大批量操作涵盖了许多情形，在这些情形之下你通常求助于简单的SQL。另一方面是，有时你无法在大数据操作中排除应用层。

12.2.2 利用批量处理

想象你必须操作所有的Item对象，并且必须做的改变不像设置标记这么繁琐（这个你之前已经用单条语句完成了）。也假设你无法创建SQL存储过程，不管出于什么原因（也许因为应用程序必须在不支持存储过程的数据库管理系统中运行）。你唯一的选择是在Java中编写这个过程，并把大量的数据获取到内存，通过这个过程来运行它。

你应该通过把工作批量化（batching）来执行这个过程。这意味着创建许多更小的数据集，代替不适合内存的单个数据集。

1. 利用批量更新编写过程

下列代码同时加载了100个Item对象用于处理：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults itemCursor =
    session.createQuery("from Item").scroll();

int count=0;
```

```
while ( itemCursor.next() ) {
    Item item = (Item) itemCursor.get(0);
    modifyItem(item);
    if ( ++count % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}

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

你用（简单的）HQL查询从数据库中加载所有的Item对象。但不是完全把查询的结果获取到内存，而是打开了一个在线的游标（cursor）。游标是一个指针，指向处在数据库中的结果集。可以通过ScrollableResults对象控制游标，并让它随着结果移动。get(int i)调用把单个对象获取到内存中，即游标当前所指的对象。对next()的每一次调用，都使游标朝着下一个对象前进。为了避免内存耗尽，你在下一个100个对象加载到内存中之前，flush()和clear()一次持久化上下文。

持久化上下文的清除把你对最新的100个Item对象所做的改变都写到数据库中。为了实现最好的性能，你应该把Hibernate（和JDBC）配置属性hibernate.jdbc.batch_size的大小设置为与过程批量的大小相同：100。然后在清除期间执行的所有UPDATE语句也都在JDBC级被批量化。

（注意，你应该给任何批量操作禁用二级高速缓存；否则，批量过程期间对象的每一次修改都必须为这个持久化类传播到二级高速缓存。这是一项没有必要的过载。第13章将介绍如何控制二级高速缓存。）

遗憾的是，JPA不支持基于游标的查询结果。你必须调用org.hibernate.Session和org.hibernate.Query来访问这个特性。

创建和持久化大量的对象也可以使用相同的方法。

2．批量插入许多对象

如果必须在一个工作单元中创建几百或者几千个对象，就可能遇到内存耗尽的问题。被传递到insert()或者persist()的每一个对象都被添加到这个持久化上下文高速缓存中。

一种简单的解决方案是，在一定数量的对象之后清洗和清除持久化上下文。你有效地批量插入：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Item item = new Item(...);
    session.save(item);
    if ( i % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}

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

这里同时创建并持久化了100 000个对象，每次100个对象。还是要记住把hibernate.jdbc.batch_size配置属性设置为一个相等的值，并给持久化类禁用二级高速缓存。警告：如果你的实体通过identity标识符生成器被映射，Hibernate就会悄悄地禁用JDBC批量插入；许多JDBC驱动程序在这种情况下都不支持批量。

另一种完全避免持久化上下文的内存消耗的选项(通过有效地禁用它)是StatelessSession接口。

12.2.3 使用无状态的会话

持久化上下文是Hibernate和Java Persistence引擎的一项基本特性。没有持久化上下文，你将无法操作对象状态，也无法让Hibernate自动侦测到变化。许多其他的事情也将不可能。

然而，如果你喜欢通过执行语句使用数据库的话，Hibernate给你提供了另一个接口。这个面向语句的接口org.hibernate.StatelessSession，感觉和表现都像简单的JDBC，除了你从被映射的持久化类和Hibernate的数据库可移植性中受益之外。

想象你想通过这个接口执行在前面例子中编写的相同的“更新所有的货品对象”过程：

```
Session session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults itemCursor =
    session.createQuery("from Item").scroll();
while ( itemCursor.next() ) {
    Item item = (Item) itemCursor.get(0);
    modifyItem(item);
    session.update(item);
}

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

在这个例子中批量不见了——你打开了StatelessSession。不再在持久化状态下使用对象，从数据库返回的一切都处于脱管状态。因此，修改Item对象之后，必须调用update()使变化变成持久化。注意，这个调用不再重附脱管的和被修改的item。它执行一个立即的SQL UPDATE；执行完命令之后，item再次处于脱管状态。

禁用持久化上下文及使用StatelessSession接口有一些其他的严重后果和概念上的限制(至少，如果你与一般的Session比较的话)：

- StatelessSession没有持久化上下文高速缓存，没有与任何其他的二级或者查询高速缓存交互。你进行的每一件事都导致立即的SQL操作。
- 对于对象的修改不会被自动侦测（没有脏检查），并且SQL操作也不会被尽可能迟地执行(没有迟写)。
- 你调用的所有对象的修改和操作都没有被级联到任何被关联的实例。你正在使用单个实体类的实例。
- 对于被映射为实体关联（一对多，多对多）的集合的任何修改都被忽略。只考虑值类型

的集合。因此，你不应该把包含集合的实体关联（而只应该把包括外键的非反向端）映射为多对一；只通过一端处理关系。编写一个查询来获得你要的数据，否则要通过迭代一个被映射的集合来获取。

- StatelessSession绕过任何被启用的org.hibernate.Interceptor，并且无法通过事件系统被拦截（本章稍后会讨论到这两项特性）。
- 你没有受保护的对象同一性范围。同一个查询生成两个不同的内存脱管实例。如果没有小心地实现持久化类的euqals()方法，则可能导致数据别名。

StetelessSession好的使用案例很少；如果通过一般的Session手工批量操作变得麻烦，你可能会更喜欢它。记住，insert()、update()和delete()操作与一般Session中相当的save()、update()和delete()操作有着天然不同的语义。（它们或许也应该有不同的名称；StetelessSession API被特别添加到Hibernate，不需要过多的计划。Hibernate开发人员团队讨论过要在Hibernate的未来版本中重新命名这个接口；你可能在正在使用的Hibernate版本中发现它们有着不同的名称。）

本章到目前为止，已经介绍了如何通过级联、大批量和批量操作，利用最有效的策略来保存和操作许多对象。现在要考虑拦截和数据过滤，以及如何以透明的方式钩入Hibernate的过程。

12.3　数据过滤和拦截

想象你不想看到数据库中的所有数据。例如，当前登录到应用程序的用户可能无权看到所有内容。通常，你把一个条件添加到查询，动态地限制结果。如果你必须处理如安全性或者临时数据（例如“只显示最近一周的数据”）这样的关注点，这就有点难度了。更难的是集合中的限制；如果你通过Category中的Item对象迭代，就会看到它们全部。

针对这个问题的一种可能的解决方案是使用数据库视图。SQL没有标准化动态的视图——可以被限制，并在运行时通过一些参数（当前登录的用户、时间周期等）移动的视图。很少有数据库提供更加灵活的视图选项，并且即使可用，也会很贵且（或）很复杂［例如，Oracle提供了一种虚拟专有数据库（Virtual Private Database，VPD）加法］。

Hibernate给动态的数据库视图提供了另一种可选的方法：运行时通过动态参数化的数据过滤器（data filter）。我们会在接下来的小节中看看数据过滤的使用案例和应用程序。

数据库应用程序中另一个常见的问题是，需要了解被保存或者被加载的数据的横切关注点。例如，想象你必须给应用程序中的每一个数据修改编写一个审计日志。Hibernate提供org.hibernate.Interceptor接口，允许你钩入Hibernate内部处理，并执行如审计日志这样的附带作用。可以利用拦截完成更多的工作，讨论完对数据过滤后，我们会给你介绍一些技巧。

Hibernate内核基于一个事件/监听器（event/listener）模型（内部最后重构的结果）。例如，如果对象必须被加载，就触发LoadEvent。Hibernate内核被实现为这些事件默认的监听器，如果你喜欢，这个系统还让你插入自己的监听器的公共接口。事件系统提供发生在Hibernate内部的任何可想象操作的完整定制，应该被当作拦截的另一种更强大的方法——我们将介绍如何编写定制的监听器，并且让你亲自处理事件。

先在一个工作单元中应用动态的数据过滤。

12.3.1 动态数据过滤

动态数据过滤的第一个使用案例与数据安全性相关。CaveatEmptor中的User有一个ranking属性。现在假设用户只可以对由同级或者更低等级的其他用户提供的货品进行出价。在业务方面，你有几组由任意等级（一个数字）定义的用户，并且用户们只能在它们的组内部进行交易。

可以用复杂的查询实现这一点。例如，假设你想要显示Category中的所有Item对象，但只是那些被同组（与登录的用户同一等级或者更低）的用户出售的货品。你得编写一个HQL或者Criteria查询获取这些货品。然而，如果用aCategory.getItems()并导航到这些对象，则所有的Item实例都将可见。

你用动态过滤解决这个问题。

1. 定义数据过滤器

动态数据过滤器用一个全局唯一的名称在映射元数据中定义。你可以在喜欢的任何XML映射文件中添加这个全局的过滤器定义，只要它是在<hibernate-mapping>元素内部：

```
<filter-def name="limitItemsByUserRank">
    <filter-param name="currentUserRank" type="int"/>
</filter-def>
```

这个过滤器被命名为limitItemByUserRank，并接受类型int的一个运行时参数。可以把相当的@org.hibernate.annotations.FilterDef注解放在你喜欢的任何类中（或者包元数据中）；它对该类的行为没有影响：

```
@org.hibernate.annotations.FilterDef(
    name="limitItemsByUserRank",
    parameters = {
        @org.hibernate.annotations.ParamDef(
            name = "currentUserRank", type = "int"
        )
    }
)
```

过滤器现在仍未激活；没有东西（可能这个名称要除外）表明它应该应用到Item对象。你必须在想要过滤的类或者集合中应用和实现过滤器。

2. 应用和实现过滤器

你想要在Item类中应用定义好的过滤器，以便如果登录用户不具备必要的等级，就看不到Item类：

```
<class name="Item" table="ITEM">
    ...
    <filter name="limitItemsByUserRank"
            condition=":currentUserRank >=
                       (select u.RANK from USER u
                        where u.USER_ID = SELLER_ID)"/>
</class>
```

<filter>元素可以对类映射设置。它把一个具名的过滤器应用到该类的实例。condition

是一个SQL表达式，直接被传递到数据库系统，因此你可以使用任何SQL操作符或者函数。如果一条记录应该通过过滤器，它的值就必须为true。在这个例子中，你使用一个子查询来获得货品卖主的等级。非限定列（如SELLER_ID）指向实体类被映射到的表。如果当前登录用户的级别没有大于或者等于由该子查询返回的等级，Item实例就被过滤掉。

下面代码与Item实体注解中的一样：

```
@Entity
@Table(name = "ITEM")
@org.hibernate.annotations.Filter(
    name = "limitItemsByUserRank",
    condition=":currentUserRank >= " +
              "(select u.RANK from USER u" +
              " where u.USER_ID = SELLER_ID)"
)
public class Item implements { ... }
```

可以应用几个过滤器，通过把它们组合在@org.hibernate.annotations.Filters注解中。一个定义好的且得到应用的过滤器，如果为特定的工作单元而被启用，就会过滤掉没有通过这个条件的任何Item实例。我们来启用它。

3. 启用过滤器

你已经定义了一个数据过滤器，并把它应用到了一个持久化类。它仍然没有过滤任何东西；它必须对特定的Session在应用程序中被启用并参数化（EntityManager不支持这个API——你必须退回到Hibernate接口来实现这项功能）：

```
Filter filter = session.enableFilter("limitItemsByUserRank");
filter.setParameter("currentUserRank", loggedInUser.getRanking());
```

你通过名称启用过滤器；这个方法返回一个Filter实例。这个对象接受运行时的参数。必须设置已经定义好的参数。其他有用的Filter方法是getFilterDefinition()（它允许遍历参数名称和类型）和validate()（如果忘了设置参数，它会抛出HibernateException）。也可以用setParameterList()设置一系列参数，如果你的SQL条件包含一个带有量词操作符（如IN操作符）的表达式，它最有用。

现在，在被过滤的Session中执行的每一个HQL或者Criteria查询都限制了返回的Item实例：

```
List<Item> filteredItems =
                session.createQuery("from Item").list();
List<Item> filteredItems =
                session.createCriteria(Item.class).list();
```

两种检索对象的方法没有被过滤：通过标识符检索和通过对Item实例的导航访问（例如通过aCategory.getItems()从Category中）检索。

通过标识符检索无法通过动态的数据过滤器限制。这在概念上来说也是错误的：如果你知道一个Item的标识符，为什么不允许看到它？解决方案是过滤标识符——也就是说，不公开首先被限制的标识符。对于多对一或者一对一关联的过滤也一样。如果多对一关联被过滤（例如，如果你调用anItem.getSeller()，就通过返回null），关联的多样性也会改变！这在概念上也是错误的，并且不是过滤器的本意。

可以通过在一个集合中应用相同的过滤器解决第二个问题：导航访问。

4. 过滤集合

目前为止，调用aCategory.getItems()返回被该Category引用的所有Item实例。这可以通过应用到集合的一个过滤器进行限制：

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

    ...

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

            <filter name="limitItemsByUserRank"
                    condition=":currentUserRank >=
                               (select u.RANK from USERS u where
➥ u.USER_ID = SELLER_ID)"/>
        </many-to-many>
    </set>

</class>
```

在这个示例中，你没有把过滤器应用到集合元素，而是应用到<many-to-many>。现在，子查询中的非限定SELLER_ID列引用关联的目标ITEM表，而不是关联的CATEGORY_ITEM联结表。通过注解，你可以在@ManyToMany字段或者获取方法中使用@org.hibernate.annotations.FilterJoinTable(s)，在多对多关联上应用过滤器。

如果Category和Item之间是一对多关联，你就已经创建了下列映射：

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

    ...

    <set name="items">
        <key column="CATEGORY_ID"/>
        <one-to-many class="Item"/>

        <filter name="limitItemsByUserRank"
                condition=":currentUserRank >=
                           (select u.RANK from USERS u
➥ where u.USER_ID = SELLER_ID)"/>
    </set>

</class>
```

通过注解，你只是把@org.hibernate.annotations.Filter(s)放在正确的字段或者获取方法中，与@OneToMany或者@ManyToMany注解并排在一起。

如果现在在Session中启用过滤器，对Category的items集合的所有遍历都会被过滤。

如果有一个默认的过滤条件应用到多个实体，就用过滤器定义对它进行声明：

```
<filter-def name="limitByRegion"
            condition="REGION >= :showRegion">
    <filter-param name="showRegion" type="int"/>
</filter-def>
```

如果把这个过滤器（无论是否使用额外的条件）应用到实体或者集合，并在Session中启用

它，这个过滤器就始终把实体表中的REGION列与运行时showRegion参数进行比较。

对于动态的数据过滤，还有许多其他的优秀用例。

5. 动态数据过滤的用例

Hibernate的动态过滤器在许多情况下都很有用。唯一受限于你的想象力和使用SQL表达式的能力。典型的用例如下：

- 安全性限制（security limit）——常见的问题是对被给定的一些任意的安全性相关条件的数据访问的限制。这可以是一个用户的等级、是用户必须属于的一个特定的组，或者是用户已经被分配到的一个角色。
- 区域数据（regional data）——数据经常通过一个区域代码进行保存（例如，一个销售团队的所有业务合同）。每个销售人员都只在覆盖他们区域的一个数据集上工作。
- 临时数据（temporal data）——许多企业应用程序需要在数据上应用基于时间的视图（例如，看看上周的一个数据集）。Hibernate的数据过滤器可以提供帮助你实现这种功能的基本的临时限制。

另一个有用的概念是Hibernate内部的拦截，实现正交的关注点。

12.3.2　拦截 Hibernate 事件

假设你想要给所有的对象修改编写一个审计日志。这个审计日志保存在包含有关对其他数据所做改变的数据库表中——特别是，有关导致改变的事件。例如，你可以给拍卖Item记录有关创建和更新事件的信息。通常记录的信息有用户、事件的日期和时间、发生了什么事件类型，以及被改变的货品。

审计日志经常利用数据库触发器处理。另一方面，应用程序承担责任有时更好，尤其当不同的数据库之间需要可移植性的时候。

实现审计日志需要几个元素。第一，必须给你想对其启用审计日志的持久化类做上标记。第二，定义什么信息应该被记入日志，例如用户、日期、时间和修改的类型。第三，用自动创建审计轨迹的org.hibernate.Interceptor把所有的东西连在一起。

1. 创建标记接口

首先，创建标记接口Auditable。用这个接口给所有应该被自动审计的持久化类做上标记：

```
package auction.model;

public interface Auditable {
    public Long getId();
}
```

这个接口要求持久化实体类用一个获取方法公开它的标识符；你需要这个属性来把审计轨迹记入日志。然后给特定的持久化类启用审计日志就是小事了。把它添加到类声明中——例如对Item：

```
public class Item implements Auditable { ... }
```

当然，如果Item类没有公开一个公有的getId()方法，就需要添加它。

2. 创建和映射日志记录

现在创建新的持久化类AuditLogRecord。这个类表示你想要在审计数据库表中记入日志的

信息：

```
public class AuditLogRecord {

    public String message;
    public Long entityId;
    public Class entityClass;
    public Long userId;
    public Date created;

    AuditLogRecord() {}

    public AuditLogRecord(String message,
                          Long entityId,
                          Class entityClass,
                          Long userId) {
        this.message = message;
        this.entityId = entityId;
        this.entityClass = entityClass;
        this.userId = userId;
        this.created = new Date();
    }
}
```

不应该把这个类当成领域模型的一部分！因此，你公开所有属性作为公有的；你将不可能需要重构应用程序的这个部分。AuditLogRecord是持久层的一部分，并可能与其他持久化相关的类共用相同的包，如HibernateUtil或者定制的UserType扩展。

接下来，把这个类映射到AUDIT_LOG数据库表：

```
<hibernate-mapping default-access="field">

<class name="persistence.audit.AuditLogRecord"
       table="AUDIT_LOG" mutable="false">

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

    <property   name="message"
                type="string"
                column="MESSAGE"
                length="255"
                not-null="true"/>

    <property   name="entityId"
                type="long"
                column="ENTITY_ID"
                not-null="true"/>

    <property   name="entityClass"
                type="class"
                column="ENTITY_CLASS"
                not-null="true"/>

    <property   name="userId"
                type="long"
                column="USER_ID"
```

```
                not-null="true"/>
    <property   name="created"
                column="CREATED"
                type="java.util.Date"
                update="false"
                not-null="true"/>
</class>
</hibernate-mapping>
```

你映射对field策略的默认访问（这个类中没有获取方法），并且，由于AuditLogRecord对象从来不更新，映射类为mutable="false"。注意你没有声明一个标识符属性名称（类也没有这样的属性）；因此Hibernate内部管理AuditLogRecord的代理键。你不准备以脱管方式使用AuditLogRecord，因此它不必包含标识符属性。然而，如果用注解映射了这个类为Java Persistence实体，就需要标识符属性。相信你在自己的代码中创建这个实体映射已经没有任何问题了。

审计日志是对引起可记入日志事件的业务逻辑的某种正交关注点。用于审计日志的逻辑可能与业务逻辑混淆，但是在许多应用程序中，审计日志最好在一段重要的代码中进行处理，对业务逻辑透明（尤其当你依赖级联选项的时候）。每当Item被修改时创建新的AuditLogRecord并保存，这必定不是你愿意手工完成的事情。Hibernate提供了Interceptor扩展接口。

3. 编写拦截器

当调用save()时，应该自动调用logEvent()方法。利用Hibernate完成这件事的最好方法是实现Interceptor接口。代码清单12-1展现了用于审计日志的拦截器。

代码清单12-1　实现用于审计日志的拦截器

```
public class AuditLogInterceptor extends EmptyInterceptor {

    private Session session;
    private Long userId;

    private Set inserts = new HashSet();
    private Set updates = new HashSet();

    public void setSession(Session session) {
        this.session=session;
    }

    public void setUserId(Long userId) {
        this.userId=userId;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types)
            throws CallbackException {

        if (entity instanceof Auditable)
            inserts.add(entity);
```

```
        return false;
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types)
            throws CallbackException {
        if (entity instanceof Auditable)
            updates.add(entity);

        return false;
    }

    public void postFlush(Iterator iterator)
                    throws CallbackException {
        try {
            for (Iterator it = inserts.iterator(); it.hasNext();) {
                Auditable entity = (Auditable) it.next();
                AuditLog.logEvent("create",
                                  entity,
                                  userId,
                                  session.connection());
            }
            for (Iterator it = updates.iterator(); it.hasNext();) {
                Auditable entity = (Auditable) it.next();
                AuditLog.logEvent("update",
                                  entity,
                                  userId,
                                  session.connection());
            }
        } finally {
            inserts.clear();
            updates.clear();
        }
    }
}
```

Hibernate `Interceptor` API的方法远远不止于这个例子中所展现的。由于你正在扩展`EntityManager`而不是直接实现这个接口，因此可以依赖你没有覆盖的所有方法的默认语义。拦截器有两个值得关注的方面。

这个拦截器需要`session`和`userId`属性进行工作；使用这个拦截器的客户端必须设置这两个属性。另一个值得关注的方面是`onSave()`和`onFlushDirty()`中的审计日志子程序：把新的、更新过的实体添加到`inserts`和`updates`集合。每当Hibernate保存实体时，就调用`onSave()`拦截器方法；每当Hibernate侦测到脏对象时，就调用`onFlushDirty()`方法。

审计轨迹的实际日志在`postFlush()`方法中完成，Hibernate在执行了把持久化上下文实现为与数据库同步的SQL之后调用它。使用静态的调用`AuditLog.logEvent()`（我们接下来要讨论的

这个类和方法）把事件记入日志。注意你无法在onSave()中把事件记入日志，因为瞬时实体的标识符值不可能在这个时候被知道。Hibernate保证在清除期间设置实体标识符，因此postFlush()是把这个信息记入日志的最好地方。

还要注意如何使用session：你把给定的Session的JDBC连接传递到对AuditLog.logEvent()的静态调用。这么做自有它的道理，我们将会进行深入的讨论。

先把它全部连在一起，看看如何启用这个新的拦截器。

4．启用拦截器

第一次打开会话时，你需要把Interceptor分配到Hibernate Session：

```
AuditLogInterceptor interceptor = new AuditLogInterceptor();
Session session = getSessionFactory().openSession(interceptor);
Transaction tx = session.beginTransaction();

interceptor.setSession(session);
interceptor.setUserId( currentUser.getId() );

session.save(newItem); // Triggers onSave() of the Interceptor

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

对于用其打开的Session来说，拦截器是活动的。

如果使用sessionFactory.getCurrentSession()，就不要控制Session的打开；它由CurrentSessionContext的其中一个Hibernate内建的实现透明地进行处理。可以编写自己的（或者扩展现有的）CurrentSessionContext实现，为打开当前的Session并给它分配一个拦截器提供自己的子程序。

另一种启用拦截器的方法是，在创建SessionFactory之前，通过setInterceptor()在Configuration上全局地设置它。然而，在Configuration中设置的并且所有Session都可以启用的任何拦截器都必须被线程安全地实现！单个Interceptor实例被并发运行的Session共享。AuditLogInterceptor实现不是线程安全的：它使用了成员变量（inserts和updates队列）。

也可以通过persistence.xml中的下列配置选项，设置一个共享的线程安全的拦截器，该拦截器有一个无参构造器，用于JPA中所有的EntityManager实例：

```
<persistence-unit name="...">
  <properties>
      <property name="hibernate.ejb.interceptor"
                value="my.ThreadSafeInterceptorImpl"/>
      ...
  </properties>
</persistence-unit>
```

我们回到拦截器中那段值得关注的Session处理代码，找出为什么要把当前Session的connection()传递到AuditLog.logEvent()。

5．使用临时的会话

AuditLogInterceptor内部为什么需要Session，这应该很清楚了。拦截器必须创建和持久化AuditLogRecord对象，因此onSave()方法的第一个尝试可能就是下列子程序：

```
if (entity instanceof Auditable) {

    AuditLogRecord logRecord = new AuditLogRecord(...);
    // set the log information

    session.save(logRecord);
}
```

这似乎很简单：利用当前运行的Session，创建新的AuditLogRecord实例并保存它。但这样并不起作用。

从Interceptor回调中调用原始的Hibernate Session是非法的。Session在拦截器调用期间处于易碎状态。在其他对象的保存期间，无法save()新对象！避免这个问题的一个好技巧是，只为保存单个AuditLogRecord对象打开新的Session。从原始的Session中复用JDBC连接。

这个临时的Session处理被封装在AuditLog类中，如代码清单12-2所示。

代码清单12-2 AuditLog辅助类使用了临时Session

```
public class AuditLog {

    public static void logEvent(
        String message,
        Auditable entity,
        Long userId,
        Connection connection) {

        Session tempSession =
                    getSessionFactory().openSession(connection);
        try {
            AuditLogRecord record =
                new AuditLogRecord(message,
                                   entity.getId(),
                                   entity.getClass(),
                                   userId );

            tempSession.save(record);
            tempSession.flush();

        } finally {
            tempSession.close();
        }
    }
}
```

logEvent()方法在同一个JDBC连接中使用了新的Session，但它从不启动或者提交任何数据库事务。它所做的就是在清除期间执行单个的SQL语句。

在同一个JDBC连接和事务中给某些操作使用临时Session的这一技巧，有时候在其他情况下也有帮助。你要记住的就是，Session只是持久化对象（持久化上下文）的高速缓存，以及使这个高速缓存与数据库同步的一系列SQL操作。

我们鼓励你去体验和尝试不同的拦截器设计模式。例如，可以重新设计审计机制，把任何实体都记入日志，而不仅仅是Auditable。Hibernate网站也有使用内嵌拦截器、甚至把一个实体的

完整历史（包括被更新的属性和集合信息）记入日志的示例。

org.hibernate.Interceptor接口也有更多可以用来钩入Hibernate过程的方法。我们认为拦截通常足以实现任何正交的关注点。

话虽这么说，Hibernate还是允许你通过它所基于的可扩展事件系统，更深地钩入到它的内核。

12.3.3　内核事件系统

Hibernate 3.x与Hibernate 2.x相比，是内核持久化引擎实现的一项重要的重新设计。新内核引擎基于一个事件/监听器的模型。例如，如果Hibernate需要保存对象，就会触发一个事件。任何监听器听到这种事件都可以捕捉它，并处理对象的保存。所有Hibernate内核功能都因此被实现为一组默认的监听器，它们可以处理所有的Hibernate事件。

这已经被设计为开放的系统：可以给Hibernate事件编写和启用自己的监听器。可以替换现有的默认监听器，或者扩展它们并执行一个附带作用或者额外的过程。替换事件监听器很少见；这么做意味着你自己的监听器实现可以负责一些Hibernate内核功能。

本质上而言，Session接口的所有方法都与一个事件关联。load()方法触发LoadEvent，并且这个事件默认通过DefaultLoadEventListener处理。

定制的监听器应该给它想要处理的事件实现适当的接口，并且（或者）扩展Hibernate提供的其中一个便利的基类，或者任何默认的事件监听器。下面是定制加载事件监听器的一个例子：

```
public class SecurityLoadListener extends DefaultLoadEventListener {

    public void onLoad(LoadEvent event,
                       LoadEventListener.LoadType loadType)
            throws HibernateException {

        if ( !MySecurity.isAuthorized(
                event.getEntityClassName(), event.getEntityId()
             )
           ) {
            throw MySecurityException("Unauthorized access");
        }

        super.onLoad(event, loadType);

    }
}
```

这个监听器调用静态的方法isAuthorized()，通过必须加载的实例的实体名称，以及该实例的数据库标识符。如果对该实例的访问被拒绝，就会抛出一个定制的运行时异常。如果没有异常抛出，过程就会被传递到超类中默认的实现。

监听器实际上应该被认为是单例（singleton），意味着它们在请求之间被共享，因而不应该把任何事务相关的状态保存为实例变量。对于原生Hibernate中所有事件和监听器接口的一个清单，请见org.hibernate.event包的API Javadoc。一个监听器实现也可以实现多个事件—监听器接口。

定制监听器可以通过Hibernate Configuration对象被编程式地注册，或者在Hibernate配置

XML中指定（不支持通过属性文件的声明性配置）。你还需要一个配置入口，告诉Hibernate除了默认的监听器之外还要使用这个监听器：

```
<session-factory>
  ...
  <event type="load">
    <listener class="auction.persistence.MyLoadListener"/>
  </event>
</session-factory>
```

监听器按照它们在你配置文件中排列的相同顺序进行注册。可以创建一个监听器的堆栈。在这个例子中，由于你正在扩展内建的DefaultLoadEventListener，因此只有一个监听器。如果没有扩展DefaultLoadEventListener，则必须把内建的DefaultLoadEventListener指定为堆栈中的第一个监听器，否则要在Hibernate中禁用加载！

另一种方法是，编程式地注册监听器堆栈：

```
Configuration cfg = new Configuration();

LoadEventListener[] listenerStack =
    { new MyLoadListener(), ...};

cfg.getEventListeners().setLoadEventListeners(listenerStack);
```

声明注册的监听器无法共享实例。如果同一个类名被用在多个<listener/>元素中，则每个引用都会产生该类的一个单独实例。如果需要在监听器类型之间共享监听器实例的能力，就必须使用编程式的注册方法。

Hibernate EntityManager也支持监听器的定制。可以在persistence.xml配置中配置共享的事件监听器，就像下面这样：

```
<persistence-unit name="...">
  <properties>
      <property name="hibernate.ejb.event.load"
                value="auction.persistence.MyLoadListener, ..."/>
      ...
  </properties>
</persistence-unit>
```

配置选项的属性名称随着你想要监听的每个事件类型（在前一个例子中是load）的改变而改变。

如果替换内建的监听器，就像MyLoadListener所做的那样，就需要扩展正确的默认监听器。在编写本书之时，Hibernate EntityManager没有绑定它自己的LoadEventListener，因此扩展org.hibernate.event.DefaultLoadEventListener的监听器仍然运行得很好。可以在参考文档以及org.hibernate.ejb.event包的Javadoc中找到完整的、最新的Hibernate EntityManager默认监听器的清单。如果想要保留Hibernate EntityManager引擎的基本行为，就扩展这些监听器中的任何一个。

几乎不必用自己的功能去扩展Hibernate内核事件系统。大多数时候，org.hibernate.Interceptor就足够灵活了。它帮助拥有更多选项，帮助以模块的方式替换任何Hibernate内核引擎。

EJB 3.0标准包括几个拦截选项，用于会话bean和实体。可以把会话bean方法调用包装在任何定制拦截器中，拦截对一个实体实例的任何修改，或者让Java Persistence服务在特定生命周期事件的bean中调用方法。

12.3.4　实体监听器和回调

EJB 3.0实体监听器是拦截实体回调事件（例如实体实例的加载和保存）的类。这类似于原生的Hibernate拦截器。可以编写定制监听器，并通过注解或者XML部署描述符中的一个绑定，把它们附加到实体。

看看下面这个简易的监听器：

```
import javax.persistence.*;

public class MailNotifyListener {

    @PostPersist
    @PostLoad
    public void notifyAdmin(Object entity) {
        mail.send("Somebody saved or loaded: " + entity);
    }

}
```

实体监听器不实现任何特定的接口；它需要一个无参构造器（在前一个例子中，是默认的构造器）。你把回调注解应用到特定事件中需要被通知的任何方法；可以在单个方法中合并几个回调。不允许在几个方法中重复相同的回调。

监听器类通过注解绑定到一个特定的实体类：

```
import javax.persistence.*;

@Entity
@EntityListeners(MailNotifyListener.class)
public class Item {
    ...

    @PreRemove
    private void cleanup() {
        ...
    }
}
```

如果需要绑定几个监听器，@EntityListeners注解可以带上一大批类。也可以把回调注解放在实体类自身中，但还是不能在单个类的方法中重复回调。然而，可以在几个监听器类中或者在监听器和实体类中实现相同的回调。

也可以为整个层次结构把监听器应用到超类，并在orm.xml配置文件中定义默认的监听器。最后，可以通过@ExcludeSuperclassListeners和@ExcludeDefaultListeners注解，给特定的实体排除超类监听器或者默认的监听器。

所有的回调方法都可以有任意的可见性，必须返回void，并且不允许抛出任何checked exception。如果抛出unchecked exception，并且有一个JTA事务正在处理，这个事务就会被

回滚。

可用的JPA回调的清单如表12-2所示。

表12-2 JPA事件回调和注解

回调注解	描 述
@PostLoad	在实体实例通过find()或者getReference()加载之后，或者在执行Java Persistence查询时触发。也在调用refresh()方法之后被调用
@PrePersist, @PostPersist	当persist()在实体中调用时，且在数据库插入之后，立即发生
@PreUpdate, @PostUpdate	在持久化上下文与数据库同步之前和之后执行——也就是说，在清除之前和之后。只在实体状态需要同步时才触发（例如，由于被认为是脏的）
@PreRemove, @PostRemove	当调用remove()时，或者当通过级联移除实体实例时，并且是在数据库删除之后触发

不像Hibernate拦截器，实体监听器是无状态的类。因此无法用实体监听器重写前一个Hibernate审计日志示例，因为你需要在本地队列中持有被修改对象的状态。另一个问题是，不允许实体监听器类使用EntityManager。确定的JPA实现，如Hibernate，让你用一个临时的另一个持久化上下文再次应用这个技巧，但你应该看看EJB 3.0拦截器中的会话bean，或许在应用程序堆栈中更高的层中编写这个审计日志的代码。

12.4 小结

本章介绍了如何有效地利用大型的复杂数据集。我们首先探讨了Hibernate的级联选项，以及如何通过Java Persistence和注解启用传播性持久化。然后，涵盖HQL和JPA QL中的大批量操作，以及如何编写在数据的子集上工作的批量过程，避免内存耗尽。

最后一节介绍了如何启用Hibernate数据过滤器，以及如何在应用程序的级别上创建动态的数据视图。最后，我们介绍了Hibernate Interceptor扩展点、Hibernate内核事件系统和标准Java Persistence实体回调机制。

表12-3展现了可以用来比较原生Hibernate特性和Java Persistence的一个概括。

第13章将转换视角，讨论如何利用最好的抓取和高速缓存策略，从数据库中获取对象。

表12-3 第12章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate通过级联选项，对所有操作支持传播性持久化	包含级联选项的传播性持久化模型相当于Hibernate。对于特殊案例请使用Hibernate注解
Hibernate在多态的HQL中支持大批量UPDATE、DELETE和INSERT...SELECT操作，它们直接在数据库中执行	JPA QL支持直接的大批量UPDATE和DELETE
Hibernate对批量更新支持查询结果游标	Java Persistence没有标准化带有游标的查询，需要退回到Hibernate API
强大的数据过滤可用于动态数据视图的创建	对于数据过滤器的映射请使用Hibernate扩展注解
扩展点可用于拦截和事件监听器	提供标准的实体生命周期回调处理器