

### 本章内容

- 对象的生命周期和状态
- 使用Hibernate API
- 使用JPA

你现在对于Hibernate和ORM如何解决对象/关系不匹配的静态方面有了一定的了解。利用你目前所知的东西，有可能解决结构不匹配的问题，但是这个问题的有效解决方案还需要更多的东西。你必须研究运行时数据访问的策略，因为它们对于应用程序的性能是至关重要的。你基本上已经学会了如何控制对象的状态。

这一章和后面的几章涵盖了对象/关系不匹配的行为方面。我们认为这些问题至少与前面章节中讨论过的结构问题一样重要。依据我们的经验，许多开发人员只是真正知道结构不匹配，但很少关注这种不匹配的更动态的行为方面。

本章讨论对象的生命周期——对象如何变成持久化，以及它如何不再被认为是持久化的——以及触发这些转变的方法调用和其他动作。Hibernate持久化管理器Session，负责管理对象状态，因此我们讨论如何使用这个重要的API。EJB 3.0中主要的Java Persistence接口被称作EntityManager，并且由于它与Hibernate API非常类似，将很容易一起学习。当然，如果你没有使用Java Persistence或者EJB 3.0，可以很快地跳过这部分资料——我们鼓励你这两种选项都读，然后决定哪种选项对你的应用程序更好。

我们从持久化对象、它们的生命周期以及触发持久化状态改变的事件开始。虽然有些资料可能过于正式，但是对持久化生命周期（persistence lifecycle）的扎实理解是必不可少的。

## 9.1 持久化生命周期

由于Hibernate是一种透明的持久化机制——类不知道它们自己的持久化能力——有可能编写这样的应用程序逻辑：它不知道自己所操作的对象是表示持久化状态，还是表示只存在于内存中的临时状态。应用程序没必要在意在调用对象的方法时它是否为持久化。例如，你可以在Item类的一个实例中调用calculateTotalPrice()业务方法，根本不必考虑持久化（例如，在单元

测试中)。

任何包含持久化状态的应用程序都必须与持久化服务交互,每当它需要把保存在内存中的状态传播到数据库的时候(反之亦然)。换句话说,你必须调用Hibernate(或者Java Persistence)接口来保存和加载对象。

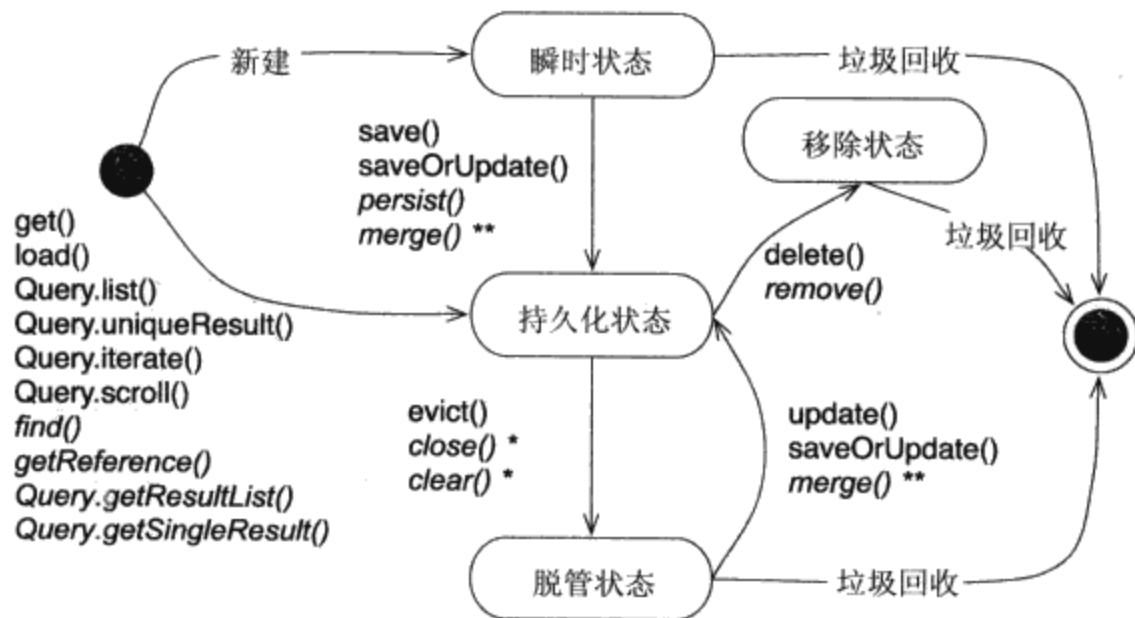
当以这种方式与持久化机制进行交互时,应用程序必须让自己知道有关持久化的对象状态和生命周期。我们这里所谓的持久化生命周期指的是对象在它的生命期间经历的状态。我们也使用术语工作单元(unit of work),指的是把一组操作当作一个(通常是原子的)组。另一个难题是由持久化服务提供的持久化上下文(persistence context)。把持久化上下文当作高速缓存,它记住你在一个特定的工作单元中给对象所做的所有修改和状态改变(这有点简化了,但这是一个好的起点)。

现在来剖析一下所有这些术语:对象和实体状态、持久化上下文和托管范围。你可能更习惯于考虑使内容从数据库进出(经由JDBC和SQL)必须管理哪些语句。然而,成功地使用Hibernate(和Java Persistence)的关键因素之一在于对状态管理的理解,因此你要学习本节内容。

### 9.1.1 对象状态

不同的ORM解决方案使用不同的术语,并给持久化生命周期定义不同的状态和状态转变。甚至,内部使用的对象状态可能与那些公开给客户端应用程序的对象状态不同。Hibernate只定义4种状态,从客户端代码中隐藏了它内部实现的复杂性。

Hibernate定义的对象状态和它们在状态图中的转变如图9-1所示。你还可以看到对触发转变的持久化管理器API的方法调用。Hibernate中的这个API是Session。我们在本章中讨论这幅图,每当你需要整体概况时就可以参考它。



\* Hibernate&JPA, 影响持久化上下文中的所有实例

\*\* 合并返回一个持久化实例, 原始实例没有改变状态

图9-1 触发持久化管理器操作时的对象状态及其转变

我们也在图9-1中包括了Java Persistence实体实例的状态。如你所见, 它们几乎与Hibernate的

一样, Session的大部分方法也类似于EntityManager API (如斜体内容所示)。假设Hibernate是Java Persistence中标准化的子集提供的功能的一个超集。

有些方法在两种API中都有;例如, Session有一个persist()操作, 与EntityManager的对应物有着相同的语义。其他的, 如load()和getReference(), 也有着相同的语义, 只是方法名称不同。

在它的生命期间, 对象可以从瞬时对象转变为持久化对象再到脱管对象。让我们更深入地探讨状态和转变。

### 1. 瞬时对象

利用new操作符实例化的对象并不立即就是持久化的。它们的状态是瞬时的(transient), 意味着它们不与任何数据库表行相关联, 因此一旦不再被其他的对象引用时, 它们的状态立即丢失。这些对象有寿命, 它在那个时候有效地终止, 并且变成不可访问, 等待垃圾回收。Java Persistence不包括这种状态的术语; 刚刚被实例化的实体对象是新的。我们将继续把它们当作是瞬时的, 以强调这些实例被持久化服务变成托管的可能。

Hibernate和Java Persistence认为所有的瞬时实例都要变成非事务的: 持久化上下文不知道瞬时实例的任何修改。这意味着Hibernate不给瞬时对象提供任何回滚功能。

只被其他瞬时实例引用的对象也默认为瞬时。对于从瞬时转变为持久化状态的实例, 要变成托管, 需要调用一个持久化管理器, 或者从已经持久化的实例中创建一个引用。

### 2. 持久化对象

持久化实例是一个包含数据库同一性的实体实例, 如在4.2节中定义的一样。这意味着持久化且被托管的实例具有设置成为其数据库标识符的主键值。(当这个标识符被分配到持久化实例的时候, 有一些变化形式。)

持久化实例可能是被应用程序实例化的对象, 然后通过持久化管理器上调用其中一种方法变成持久化。它们甚至可能是当从另一个已经托管的持久化对象中创建引用时变成持久化的对象。或者, 持久化实例也可能是通过执行查询、标识符查找、或者开始从另一个持久化实例导航对象图, 从数据库中获取的一个实例。

持久化实例始终与持久化上下文(persistence context)关联。Hibernate高速缓存它们, 并且可以侦测到它们是否已经被应用程序修改。

关于这种状态以及实例在持久化上下文中如何得到托管, 还有更多的内容要讨论。本章稍后会回到这个话题。

### 3. 移除对象

可以通过几种方式删除实体实例: 例如, 可以用持久化管理器的一个显式操作把它移除。如果移除所有对它的引用, 它可能也变成可以删除的了, 这项特性只有在包含Hibernate扩展设置(实体的孤儿删除)的Hibernate或者Java Persistence中才可用。

如果一个对象已经被计划在一个工作单元结束时删除, 它就是处于移除状态, 但仍然由持久化上下文托管, 直到工作单元完成。换句话说, 移除对象不应该被重用, 因为一旦工作单元完成, 它就将立即从数据库中被删除。你也应该放弃在应用程序中保存着的任何对它的引用(当然, 是

在你用完它之后——例如，在你已经呈现了用户看到的删除确认屏幕之后）。

#### 4. 脱管对象

要理解脱管（detached）对象，你需要考虑实例的一种典型的转变：它先是瞬时的，因为它刚刚在应用程序中创建。现在通过在持久化管理器中调用一个操作使它变成持久化。所有这些都发生在单个工作单元中，并且这个工作单元的持久化上下文在某个时间点（当产生一个SQL的INSERT时）与数据库同步。

现在工作单元完成了，持久化上下文也关闭了。但是应用程序仍然有一个句柄（handle）：对被保存实例的一个引用。只要持久化上下文是活动的，这个实例的状态就是持久化的。在工作单元结束时，持久化上下文关闭。你现在保存着对其引用的是什么对象状态，以及如何处理呢？

我们把这些对象当作脱管（detached），表示它们的状态不再保证与数据库状态同步，不再被附加到持久化上下文中，并仍然包含持久化数据（可能很快会失效）。可以继续使用脱管对象并修改它。但有时候你或许想要使那些变化变成持久化——换句话说，把脱管实例变回到持久化状态。

Hibernate提供重附（reattachment）和合并（merging）两种操作来处理这种情况。Java Persistence只对合并标准化。这些特性对于如何设计多层应用程序有着深刻的影响。从一个持久化上下文返回对象到表现层，并且随后在一个新的持久化上下文中重用它们的能力是Hibernate和Java Persistence的主要卖点。它让你能够创建跨越用户思考时间的长工作单元。我们称这种长期运行的工作单元为对话（conversation）。我们会很快回到脱管对象和对话的话题。

现在你对于对象状态以及如何发生转变应该有了初步的理解。我们的下一个主题是持久化上下文及其提供的对象管理。

### 9.1.2 持久化上下文

你可能认为持久化上下文会成为托管实体实例的一个高速缓存。持久化上下文不是你在应用程序中所见到的东西；它不是一个可以调用的API。在Hibernate应用程序中，假设一个Session有一个内部的持久化上下文。在Java Persistence应用程序中，EntityManager具有持久化上下文。一个工作单元中所有处于持久化状态和托管状态的实体都被高速缓存在这个上下文中。本章稍后会探讨Session和EntityManager API。现在你要知道这个（内部的）持久化上下文的作用。

持久化上下文之所以有用，基于以下几个原因：

- Hibernate可以进行自动的脏检查和事务迟写。
- Hibernate可以用持久化上下文作为一级高速缓存。
- Hibernate可以保证Java对象同一性的范围。
- Hibernate可以把持久化上下文扩展到跨整个对话。

所有这些要点对Java Persistence提供程序也有效。我们来看看每一种特性。

#### 1. 自动脏检查

持久化实例托管在一个持久化上下文中——它们的状态在工作单元结束时与数据库同步。当一个工作单元结束时，保存在内存中的状态通过SQL INSERT、UPDATE和DELETE语句（DML）的

执行被传播到数据库。这个过程也可能发生在其他时间点。例如，Hibernate可能在查询执行之前与数据库同步。这样确保了查询知道在工作单元期间之前所做的改变。

Hibernate没有于工作单元结束时在内存中更新每一个单独的持久化对象的数据库行。ORM软件必须有一个策略，用来侦测哪个持久化对象已经被应用程序修改。我们称之为自动脏检查（automatic dirty checking）。一个修改过的对象还没被传播到数据库时被认为是脏的（dirty）。这种状态对于应用程序不可见。利用透明的事务级迟写（transparent transaction-level write-behind），Hibernate尽可能迟地把状态变化传播到数据库，但是从应用程序中隐藏这个细节。通过尽可能迟地执行DML（趋向于数据库事务的结束），Hibernate试图保证数据库中的锁时间尽可能短。（DML通常在数据库中创建一直被保存到事务结束的锁。）

Hibernate能够准确地侦测哪些属性已经被修改，以便有可能只包括需要在SQL UPDATE语句中更新的列。这可能带来一些性能上的收获。但是差别通常不明显，理论上来说，这样在某些环境下会损害性能。默认情况下，Hibernate包括SQL UPDATE语句中被映射的表的所有列（因而，Hibernate可以在启动时而不是运行时生成这个基础的SQL）。如果你想要只更新被修改的列，可以通过在类映射中设置dynamic-update="true"启用动态的SQL生成。对新记录的插入实现相同的机制，并且可以用dynamic-insert="true"启用INSERT语句的运行时生成。当一张表中有特别多的列（假设，超过50列）时，我们建议考虑这种设置；有时候，无变化的字段所引起的过载的网络流量也是不容忽视的。

少数情况下，你也可能想要给Hibernate提供自己的脏检查算法。Hibernate默认把一个对象的旧快照与同步时的快照进行比较，侦测任何需要更新数据库状态的修改。可以通过org.hibernate.Interceptor给Session提供一个定制的findDirty()方法来实现自己的子程序。本书稍后将介绍拦截器（interceptor）的实现。

本章后面会回到同步过程（即清除，flushing）。

## 2. 持久化上下文高速缓存

持久化上下文是持久化实体实例的一个高速缓存。这意味着它记住了你已经在特定的工作单元中处理过的所有持久化实体实例。自动脏检查是这个高速缓存的好处之一。另一个好处是对实体的可重复读取（repeatable read），以及工作范围高速缓存单元的性能优势。

例如，如果Hibernate被告知通过主键（标识符的一个查找）加载对象，它就可以先在当前的工作单元的持久化上下文中检查。如果在那里找到了实体，就不会发生数据库命中——这是对应用程序的可重复读取。如果查询通过其中一个Hibernate（或者Java Persistence）接口执行也一样。Hibernate读取查询的结果集，并封送随后返回给应用程序的实体对象。在这个过程期间，Hibernate与当前的持久化上下文交互。它试图解析这个高速缓存中的每一个实体实例（通过标识符）；只有在当前的持久化上下文中无法找到该实体时，Hibernate才会从结果集中读取剩下的数据。

持久化上下文高速缓存带来重大的性能好处，并改进工作单元中的孤立保证（免费得到了实体实例的可重复读取）。由于这个高速缓存只有工作单元的范围，它没有真正的缺点，例如对并发访问的锁管理——工作单元是在单个线程中处理的。

持久化上下文高速缓存有时候帮助避免不必要的数据库流量；但更重要的是，它确保了：



- 持久层在对象图中出现循环引用时，不会受到堆栈溢出（stack overflow）的影响。
- 工作单元结束时永远不能有相同数据库行的冲突表示法。在持久化上下文中，最多一个对象表示任何一个数据库行。对该对象进行的所有变化都可以被安全地写到数据库中。
- 同样地，在特定持久化上下文中进行的改变，也始终立即对在持久化上下文和它的工作单元内部（对实体保证的可重复读取）执行的所有其他代码可见。

不必做任何特别的事情来启用持久化上下文高速缓存。它始终开着，基于上述原因，它不可能被关闭。

本章稍后将介绍对象如何被添加到这个高速缓存（基本上是每当它们变成持久化的时候），以及如何管理这个高速缓存（通过手工从持久化上下文中脱管对象，或者通过清除持久化上下文）。

我们给持久化上下文所列的最后两项好处：受保护的同一性范围和扩展持久化上下文到跨对话的可能性，是密切相关的概念。要理解它们，就要退回一步，并从不同的角度考虑处于脱管状态的对象。

## 9.2 对象同一性和等同性

一个基础的Hibernate客户端/服务器端应用程序可能通过跨单个客户端请求的服务器端工作单元进行设计。当来自应用程序用户的一个请求需要数据访问时，就启动一个新的工作单元。这个工作单元在处理结束时终止，并为客户准备好了响应。这也被称作每次请求一个会话（session-per-request）策略（每当读到类似这样的东西时，可以用持久化上下文代替会话，但有点拗口）。

我们已经提到过，Hibernate可以支持一个可能长期运行的工作单元（称作对话，conversation）的实现。我们在接下来的几节中介绍对话的概念，以及对象同一性的基础和对象什么时候被认为是等同的——这些概念可能影响你如何思考和设计对话。

对话概念为什么会有用？

### 9.2.1 引入对话

例如，在Web应用程序中，通常不维护跨用户交互的数据库事务。用户花了很长的时间思考修改，但是由于可伸缩性的原因，必须保持数据库事务简短，并尽快释放数据库资源。每当需要引导用户通过几个屏幕完成一个工作单元（从用户的角度）时，可能面对这个问题——例如，填写一张在线的表格。在这个一般的场景中，拥有持久化服务的支持非常有用，因此可以用最少的代码和最佳的可伸缩性实现这样一个对话。

有两种策略可以在Hibernate或者Java Persistence应用程序中实现对话：利用脱管对象，或者通过扩展一个持久化上下文。这两者均各有利弊。

脱管对象状态和已经提到过的重附或者合并的特性是实现对话的方法。用户在思考时间期间对象以脱管状态保存，并且这些对象的任何修改都通过重附或者合并被手工变成持久化。这一策略也被称作利用脱管对象每次请求一个会话（session-per-request-with-detached-object）。可以在图

9-2中看到这种会话模式的图示。

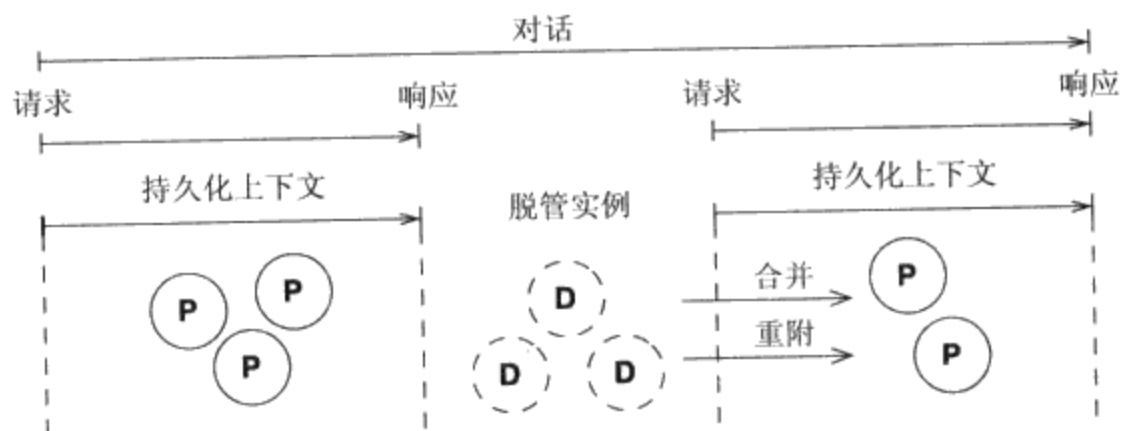


图9-2 利用脱管对象状态的对话实现

持久化上下文只跨一个特定请求的处理，对话期间应用程序手工重附和合并（且有时候脱管）实体实例。

另一种方法不需要手工重附或者合并：利用每次对话一个会话（session-per-conversation）模式，把一个持久化上下文扩展到跨整个工作单元（请见图9-3）

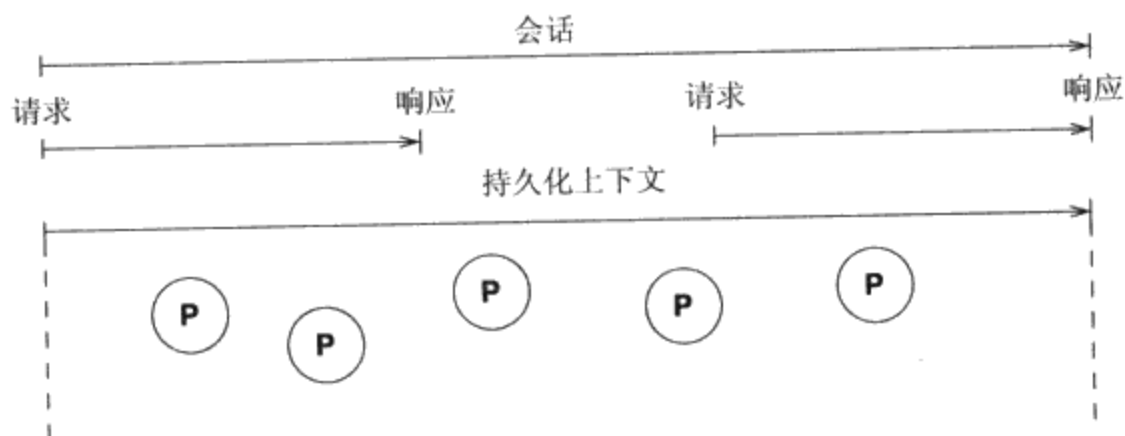


图9-3 利用被扩展持久化上下文的对话实现

首先我们来更深入地看一下脱管对象，以及在利用这种策略实现对话时会遇到的同一性问题。

### 9.2.2 对象同一性的范围

作为应用程序开发人员，我们用Java对象同一性（ $a==b$ ）辨别对象。如果对象改变了状态，Java同一性保证与新状态一样吗？在分层应用程序中，可能不会。

为了探讨这个问题，理解Java同一性 $a==b$ 和数据库同一性 $x.getId().equals(y.getId())$ 之间的关系非常重要。有时候它们是相等的；有时候不等。我们把Java同一性等价于数据库同一性的条件称作对象同一性的范围（scope of object identity）。

对于这一范围，有3种常见的选择：

- 没有同一性范围的基本持久层不保证一个行是否被访问两次，以及是否会把相同的对象实例返回给应用程序。如果应用程序修改了都在单个工作单元中表示同一行的两个不同的实例，就会有问题。（我们应该如何决定哪种状态应该被传播到数据库呢？）

- 持久层利用持久化上下文范围的同一性 (persistence context-scoped identity), 保证在单个持久化上下文的范围中, 只有一个对象实例表示一个特定的数据库行。这样避免了前一个问题, 还允许在上下文级的一些高速缓存。
- 过程范围的同一性 (process-scoped identity) 更进一步, 保证在整个过程 (JVM) 中只有一个对象实例表示该行。

对于典型的Web或者企业应用程序, 持久化上下文范围的同一性是首选。过程范围的同一性, 的确对跨多个工作单元的实例重用, 在高速缓存利用和编程模型方面提供了一些潜在的优势。但是, 在一个遍布多线程的应用程序中, 始终在全局的同一性映射中同步对持久化对象的共享访问成本太高, 难以支付。更简单并且更可伸缩些的办法是, 让每个线程在每个持久化上下文中使用一组独特的持久化实例。

假设Hibernate实现了持久化上下文范围的同一性。因此, Hibernate天生最适合多用户应用程序中高并发的数据访问。然而, 我们已经提到过一些在对象不与持久化上下文关联时会遇到的问题。让我们用一个示例来讨论这个话题。

Hibernate同一性范围是持久化上下文的范围。它在包含Hibernate API的代码中的工作原理: Java Persistence代码相当于EntityManager, 而不是Session。虽然我们还没有介绍更多有关这些接口的内容, 但是下面的例子很简单, 理解我们在Session中调用的方法应该不成问题。

如果你请求两个对象在同一个Session中使用相同的数据库标识符, 结果是对相同内存实例的两个引用。代码清单9-1用几个get()操作在两个Session中进行了示范。

**代码清单9-1** Hibernate中受保护的对象同一性范围

```
Session session1 = sessionFactory.openSession();
Transaction tx1 = session1.beginTransaction();

// Load Item with identifier value "1234"
Object a = session1.get(Item.class, new Long(1234));
Object b = session1.get(Item.class, new Long(1234));

( a==b ) // True, persistent a and b are identical

tx1.commit();
session1.close();

// References a and b are now to an object in detached state

Session session2 = sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();

Object c = session2.get(Item.class, new Long(1234));

( a==c ) // False, detached a and persistent c are not identical

tx2.commit();
session2.close();
```

对象引用a和b, 不仅有着相同的数据库同一性, 而且还有相同的Java同一性, 因为它们是在相同的Session中获得的。它们引用那个工作单元的持久化上下文中所知道的同一个持久化实



例。然而，一旦出了这个范围，Hibernate就不保证Java同一性了，因此a和c不相等。当然，测试数据库同一性的`a.getId().equals(c.getId())`，将仍然返回`true`。

如果你使用处于脱管状态的对象，就是正在处理处于受保护的对象同一性范围之外的对象。

### 9.2.3 脱管对象的同一性

如果对象引用离开了受保护的同一性范围，那么就称它为脱管对象引用（reference to a detached object）。在代码清单9-1中，如果只考虑数据库同一性——它们的主键值，则所有三个对象引用a、b和c都相等。然而，它们不是相等的内存对象实例。如果你在脱管状态下把它们作为相等处理，就会产生问题。例如，考虑代码的下列扩展，在`session2`已经终止之后：

```
...
session2.close();

Set allObjects = new HashSet();
allObjects.add(a);
allObjects.add(b);
allObjects.add(c);
```

所有三个引用都已经被添加到了一个Set。所有都是对脱管对象的引用。现在，如果你查看集合的大小、元素的数量，你希望是什么样的结果呢？

首先你必须了解Java Set的约定：在这种集合中不允许有重复元素。重复通过Set进行侦测；每当添加一个对象时，它的`equals()`方法就会自动被调用。被添加的对象与已经处在集合中的所有其他元素进行核对。如果`equals()`对已经在集合中的任何对象返回`true`，就不发生添加。

如果你知道对象的`equals()`实现，就会知道可以在Set中期待的元素数量。所有Java类都默认继承`java.lang.Object`的`equals()`方法。这个实现使用了双等号（`==`）比较；它在Java堆中检查两个引用是否指向同一个内存实例。

你可能猜测集合中的元素数量为两个。毕竟，a和b引用了相同的内存实例；它们已经被加载在相同的持久化上下文中。引用c在第二个Session中获得；它在堆中引用不同的实例。你有对两个实例的三个引用。然而，你知道这个，仅仅因为你已经见过加载对象的代码。在一个真实的应用程序中，你不可能知道a和b被加载在同一个Session中，c却在另一个之中。

此外，你显然希望这个集合正好只有一个元素，因为a、b和c表示相同的数据库行。

每当使用脱管状态下的对象时，特别在给同一性进行测试时（通常在一个基于散列的集合中），就需要为持久化类提供你自己的`equeals()`和`hashCode()`方法的实现。

#### 1. 理解`equals()`和`hashCode()`

在介绍如何实现你自己的等同性子程序之前，我们必须提醒两个重点。第一，依据我们的经验，许多Java开发人员在使用Hibernate（或者Java Persistence）之前永远不必覆盖`equals()`和`hashCode()`方法。传统上，Java开发人员似乎不关心这样一种实现的复杂细节。在公开的Hibernate论坛上最长的讨论是关于这个等同性问题的，并且经常把“抱怨”对准Hibernate。你应该了解根本的问题：如果默认的约定没有提供想要的语义，每一种带有基于散列的集合的面向对象编程语言都需要一个定制的等同性子程序。Hibernate应用程序中的脱管对象状态让你面对这个问题，这

可能是第一次。

另一方面，你可能不一定要覆盖`equals()`和`hashCode()`。如果你从来不比较脱管实例——也就是说，如果从来不把脱管实例放进相同的`Set`中，**Hibernate**提供的这个同一性范围保证就足够了。你可能决定设计一个不使用脱管对象的应用程序。可以给对话实现应用一个被扩展的持久化上下文策略，并从应用程序中完全消除脱管状态。这个策略也把受保护对象同一性的范围扩展到了跨整个对话。（注意，你仍然需要这条规定，不比较在两个会话中获得的脱管实例！）

假设你想要使用脱管对象，并且必须用自己的子程序测试它们的等同性。可以通过几种方式实现`equals()`和`hashCode()`。记住，在覆盖`eugals()`时，始终也需要覆盖`hashCode()`，以便这两种方法保持一致。如果两个对象相等，它们就必须有相同的散列码（`hashCode`）。

一种更为聪明的方法是实现`equals()`，只比较数据库标识符属性（通常是个代理主键）值：

```
public class User {
    ...

    public boolean equals(Object other) {
        if (this==other) return true;
        if (id==null) return false;
        if ( !(other instanceof User) ) return false;
        final User that = (User) other;
        return this.id.equals( that.getId() );
    }

    public int hashCode() {
        return id==null ?
            System.identityHashCode(this) :
            id.hashCode();
    }
}
```

注意这个`equals()`方法如何为还没有分配数据库标识符值的瞬时实例（如果`id==null`）回到Java同一性。这是有道理的，因为它们不可能等于脱管实例，这个脱管实例有一个标识符值。

不幸的是，这种解决方案有一个大问题：标识符值直到对象变成持久化时才由**Hibernate**分配。如果瞬时对象在保存之前被添加到一个`Set`，它的散列值就可能在它被`Set`包含时改变，与`java.util.Set`的约束相反。特别是，这个问题使得级联保存（本书后面会讨论到）对于集来说变得毫无用处。我们强烈反对这种解决方案（数据库标识符等同性）。

一种更好的方法是包括持久化类的所有持久化属性在`equals()`比较中，远离任何数据库标识符属性。这就是大多数人知道的`equals()`的含义；我们称它为按值（`by value`）等同性。

当我们说到所有属性时，并不想包括集合。集合状态与不同的表关联，因此把它包括在内似乎错了。更重要的是，你并不想强制获取整个对象图而只是用来执行`equals()`。对于`User`而言，这意味着你不应该把`boughtItems`集合包括在比较式中。你可以编写下面的实现：

```
public class User {
    ...

    public boolean equals(Object other) {
        if (this==other) return true;
```

```

        if ( !(other instanceof User) ) return false;
        final User that = (User) other;
        if ( !this.getUsername().equals( that.getUsername() ) )
            return false;
        if ( !this.getPassword().equals( that.getPassword() ) )
            return false;
        return true;
    }

    public int hashCode() {
        int result = 14;
        result = 29 * result + getUsername().hashCode();
        result = 29 * result + getPassword().hashCode();
        return result;
    }
}

```

然而,这种方法还是有两个问题。第一,如果修改了其中一个(例如,如果用户改变了密码),那么来自不同Session的实例便不再相等。第二,包含不同数据库同一性的实例(表示数据库表不同行的实例)可以被认为相等,除非属性的有些组合保证唯一(数据库列有一个唯一约束)。对于这个案例中的用户而言,有一个唯一的属性:username。

这样就把我们引到了等同性检查的首选(并且语义上正确的)实现。你需要一个业务键(business key)。

## 2. 利用业务键实现等同性

为了开始我们推荐的这个解决方案,要先理解业务键的概念。业务键是一种属性,或者一些属性的组合,它对于每个包含相同的数据库同一性的实例来说是唯一的。本质上,它就是你要使用的自然键,如果你没有正在使用代理主键来代替的话。业务键不同于自然键,它的永远不变并非绝对的必备条件——只要它很少改变,就足够了。

我们认为,本来每个实体类都应该有一些业务键,即使它包括了类的所有属性(对于一些不可变的类而言,这很合适)。业务键是用户作为唯一辨别一个特定记录的东西,而代理键则是应用程序和数据库使用的东西。

业务键等同性意味着equals()方法只比较构成业务键的属性。这是个完美的解决方案,它避免了前面描述过的所有问题。唯一的缺点是它需要特别记着先辨别正确的业务键。无论如何,这项工作都是必要的;如果你的数据库必须经由约束检查来确保数据完整性,辨别任何唯一键就很重要了。

对于User类而言,username是一个很好的备用业务键。它永远不为空,通过数据库约束而唯一,并且即使改变也很少:

```

public class User {
    ...

    public boolean equals(Object other) {
        if (this==other) return true;
        if ( !(other instanceof User) ) return false;
        final User that = (User) other;
    }
}

```

```

        return this.username.equals( that.getUsername() );
    }

    public int hashCode() {
        return username.hashCode();
    }
}

```

对于一些其他的类，业务键可能更加复杂，由一个属性的组合组成。以下是一些提示，帮助你在类中辨别业务键：

- 考虑当应用程序的用户必须辨别一个对象（在现实世界中）时，他们会引用什么属性。如果对象显示在屏幕上，用户如何把一个对象和另一个对象区分开来？这可能就是你正在寻找的业务键。
- 每个不可变的属性都可能是一个好的备选业务键。如果可变属性很少更新，或者如果当它们更新时你可以控制局势的话，可变属性也可能是好的备选对象。
- 每个包含UNIQUE数据库约束的属性都是好的备选业务键。记住，业务键的精确度必须好到足以避免重叠。
- 任何基于日期或者时间的属性，例如记录的创建时间，通常都是业务键的一个好组件。然而，`System.currentTimeMillis()`的准确性则取决于虚拟机和操作系统。我们建议过的安全缓冲区是50毫秒，如果基于时间的属性是业务键的单个属性时，它可能还不够精确。
- 可以用数据库标识符作为业务键的一部分。这似乎与我们之前的陈述相矛盾，但是我们不讨论指定类的数据库标识符。可能可以使用被关联对象的数据库标识符。例如，`Bid`类的一个备选业务键是`Item`的标识符，它与出价金额一起产生。你在数据库Schema中甚至可以有一个表示这个复合业务键的唯一约束。可以使用被关联`Item`的标识符值，因为它在`Bid`的生命周期期间从不改变——`Bid`构造器需要设置一个已经持久化的`Item`。

如果遵循了我们的建议，给你所有的业务类找到一个好的业务键就不难了。如果遇到很难的情况，就试着不考虑用Hibernate来解决它——毕竟，这是个纯粹的面向对象的问题。注意，在一个子类上覆盖`equals()`并在比较中包括另一个属性，这种做法通常都是错误的。在这种情况下，满足等同性既是对称性（symmetric）又是传递性（transitive）的要求是一个小技巧；并且更重要的是，业务键可能不符合数据库中任何定义好的备选自然键（子类属性可能被映射到不同的表）。

你可能也已经注意到，`equals()`和`hashCode()`方法始终通过获取方法访问“其他”对象的属性。这非常重要，因为作为`other`传递的对象实例可能是一个代理对象，而不是保存有持久化状态的真正实例。为了初始化这个代理来获得属性值，你需要用一个获取方法访问它。这是Hibernate不完全透明的一点。但是，使用获取方法而不是直接的实例变量访问，这无论如何都是个好实践。

现在转换视角，给不需要脱管对象且不公开任何脱管对象同一性问题的对话考虑一种实现策略。如果在使用脱管对象时可能公开的同一性范围问题似乎不堪重负，那么第二种对话实现策略可能就是你要找的。Hibernate和Java Persistence通过一个被扩展的持久化上下文——每个对话一

个会话（session-per-conversation）策略——支持对话的实现。

### 9.2.4 扩展持久化上下文

一个特定的对话为所有交互重用相同的持久化上下文。对话期间的所有请求处理都由相同的持久化上下文管理。当来自用户的请求被处理之后，持久化上下文没有关闭。在用户思考时间内，它断开与数据库的连接，并保持这种状态。当用户在对话中继续时，持久化上下文被重新连接到数据库，就可以处理下一个请求了。对话结束时，持久化上下文与数据库同步并关闭。下一个对话从一个新的持久化上下文开始，并且不重用来自前一个对话的任何实例；这一模式是可重复的。

注意，这样消除了脱管对象状态！所有实例要么是瞬时的（不为持久化上下文所知），要么是持久化的（附加到一个特定的持久化上下文）。它也不需要手工重附或合并上下文之间的对象状态，这是这个策略的好处之一。（你在对话之间仍然可能有脱管对象，但是我们认为这是你应该努力避免的一种特殊情况。）

在Hibernate中，这个策略为对话的持续使用单个的Session。Java Persistence对被扩展的持久化上下文有着内建的支持，甚至可以为你自动保存请求之间断开连接的上下文（在一个有状态的EJB会话bean中）。

本书后面会回到对话的话题，并介绍有关这两种实现策略的所有细节。你不必现在选择一种合适的策略，但是应该注意这些策略对于对象状态和对象同一性的影响，并且应该理解每个案例中必需的事务。

我们现在来探讨持久化管理器API，以及如何使对象状态背后的理论实际上真的可行。

## 9.3 Hibernate 接口

任何透明的持久化工具都包括一个持久化管理器（persistence manager）API。这个持久化管理器通常为以下内容提供服务：

- 基础的CRUD（创建、获取、更新、删除）操作；
- 查询执行；
- 事务的控制；
- 持久化上下文的管理。

持久化管理器可能通过几个不同的接口而被公开。对于Hibernate而言，它们是Session、Query、Criteria和Transaction。这些接口的实现在幕后是紧密相连的。

在Java Persistence中，你交互的主要接口是EntityManager；它扮演着与Hibernate的Session一样的角色。其他的Java Persistence接口是Query和EntityTransaction（你可能会猜测它们在原生的Hibernate中的等价物是什么）。

现在介绍如何用Hibernate和Java Persistence加载和保存对象。有时候这两者有着完全相同的语义和API，甚至方法名称也相同。因此更为重要的是注意一些微小的差别。为了使本书的这一部分更易于理解，我们决定使用一种与平时不一样的策略，即先解释Hibernate，然后再解释Java Persistence。

让我们从Hibernate开始，假设你编写一个依赖原生API的应用程序。



### 9.3.1 保存和加载对象

在一个Hibernate应用程序中，通过实质上改变它们的状态来保存和加载对象。你在工作单元中进行这项工作。单个工作单元是一组被认为是原子团的操作。如果你现在猜测它与事务密切相关，就对了。但是，没有必要是相同的东西。我们必须逐步解决这个问题；现在，把工作单元当作对已经组合到一起的对象进行状态改变的一个特定序列。

首先你必须开始一个工作单元。

#### 1. 开始工作单元

在工作单元开始时，应用程序从SessionFactory处获得了Session的一个实例：

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

此时，一个新的持久化上下文被初始化了，它将管理在这个Session中使用的所有对象。如果应用程序访问几个数据库，它可能会有多个SessionFactory。SessionFactory如何创建，以及你如何在应用程序代码中访问它依赖于开发环境和配置——如果遵循了2.1.3节中的设置，就应该具备了简单的HibernateUtil启动辅助类。

你永远不应该创建一个新的SessionFactory而只是服务一个特定的请求。SessionFactory的创建非常昂贵。另一方面，Session创建则非常便宜。Session甚至直到需要连接时才获得一个JDBC Connection。

前一段代码中的第二行在另一个Hibernate接口中开始一个Transaction。你在工作单元内部执行的所有操作都在一个事务内部发生，无论你是读取还是写入数据。但是，这个Hibernate API是可选的，你可以用喜欢的任何方式开始事务——我们将在第10章探讨这些选项。如果你使用Hibernate Transaction API，代码在所有的环境下都有效，因此你将在接下来的几节中给所有示例这么做。

打开一个新的Session和持久化上下文之后，就可以用它来加载和保存对象。

#### 2. 使对象变成持久化

你要用Session做的第一件事就是通过save()方法使一个新的瞬时对象变成持久化（见代码清单9-2）。

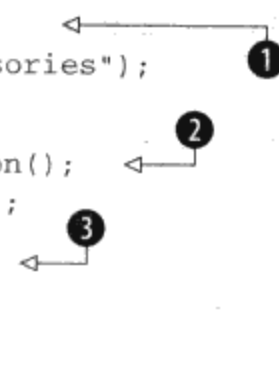
**代码清单9-2 使瞬时实例变成持久化**

```
Item item = new Item();
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );

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

Serializable itemId = session.save(item);

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



新的瞬时对象`item`像往常一样被实例化①。当然，也可以在打开一个`Session`之后再对它进行实例化；它们目前并不相关。新的`Session`用`SessionFactory`②打开。这样就启动了一个新事务。

调用`save()`③使得`Item`的瞬时实例变成了持久化。它现在与当前的`Session`和它的持久化上下文关联。

对持久化对象所做的变化必须与数据库在某个时间点上进行同步。这发生在当你对`Hibernate Transaction`④实施`commit()`的时候。我们说发生了清除（你也可以手工调用`flush()`；后面有关于这个话题的更多内容）。为了使持久化上下文同步，`Hibernate`获得了一个JDBC连接，并发出单个SQL `INSERT`语句。注意，对于插入来说，并非总是这样：`Hibernate`保证`item`对象在保存之后有一个被分配的数据库标识符，因此可能需要更早的`INSERT`，取决于你在映射中启用的标识符生成器。`save()`操作也返回持久化实例的数据库标识符。

`Session`最终可以关闭⑤，并且持久化上下文终止。`item`引用现在是对于处于脱管状态的对象的一个引用。

在图9-4中可以看到相同的工作单元，以及对象如何改变状态。

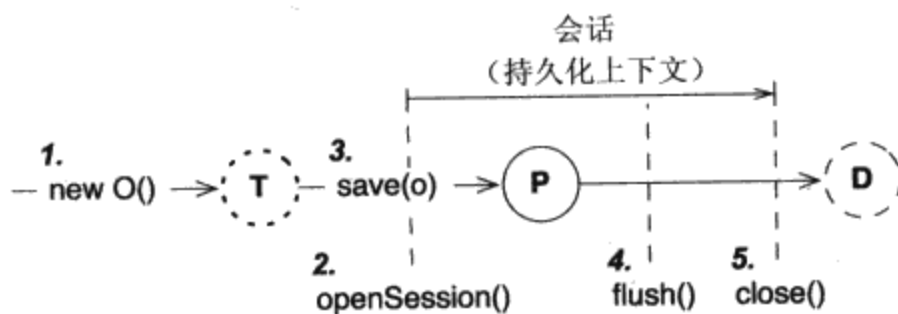


图9-4 在工作单元中使对象变成持久化

在用`Session`管理`Item`实例之前完全初始化它更好（但并非必要）。SQL `INSERT`语句包含调用`save()`时对象保存着的值。可以在调用`save()`之后修改对象，并且这种变化将被作为一个（额外的）SQL `UPDATE`传播到数据库。

`Session.beginTransaction()`和`tx.commit()`之间的一切都在一个事务中发生。现在记住，在事务范围中的所有数据库操作要么完全成功，要么完全失败。如果清除期间在`tx.commit()`上所做的其中一个`UPDATE`或者`INSERT`语句失败了，那么在这个事务中对持久化对象所做的所有变化便在数据库级回滚。然而，`Hibernate`不会把内存变化回滚到持久化对象。这是有道理的，因为事务的失败一般是不可恢复的，你必须立即放弃失败的`Session`。我们稍后将在第10章讨论异常处理。

### 3. 获取持久化对象

`Session`也用于查询数据库，并获取现有的持久化对象。`Hibernate`在这方面的功能尤其强大，你在本书稍后会体会到。它给最简单的查询提供了两种特殊的方法：通过标识符获取。`get()`和`load()`方法如代码清单9-3所示。

## 代码清单9-3 通过标识获取Item

```

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

Item item = (Item) session.load(Item.class, new Long(1234));
// Item item = (Item) session.get(Item.class, new Long(1234));

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

```

可以在图9-5中看到相同的工作单元。

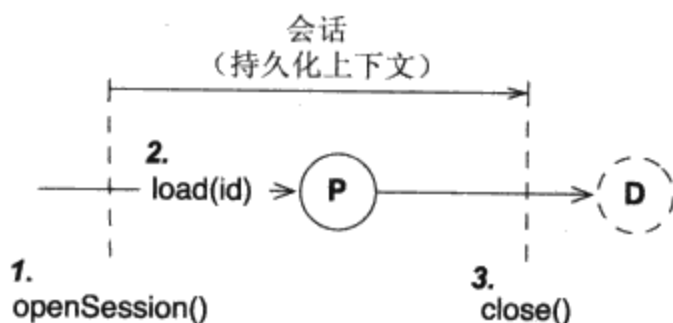


图9-5 通过标识符获取持久化对象

获取到的对象`item`处于持久化状态，并且一旦持久化上下文关闭，它就立即处于脱管状态。

`get()`和`load()`之间的一个区别在于它们如何表明实例无法被找到。如果数据库中不存在包含给定标识符值的行，`get()`就返回`null`。`load()`方法则抛出一个`ObjectNotFoundException`。由你选择喜欢的错误处理方式。

更重要的是，`load()`方法可能返回一个代理（`proxy`），一个占位符，而不命中（`hit`）数据库。这个结果就是稍后你可能得到一个`ObjectNotFoundException`，一旦你试图访问返回的占位符，就立即强制它初始化。（这也称作延迟加载（`lazy loading`）；我们在后面的几章中讨论加载优化。）`load()`方法始终试图返回一个代理，如果它已经由当前的持久化上下文管理，则仅返回一个已被初始化的对象实例。在前面介绍过的例子中，根本没有发生数据库命中！另一方面，`get()`方法从不返回代理，它始终命中数据库。

你可能会问，这种方法为什么有用——毕竟，获取对象是要进行访问的。获得一个持久化实例，并把它作为对另一个实例的引用进行分配，这很常见。例如，假如需要`item`只为了一个目的：用`Comment: aComment.setForAuction(item)`设置一个关联。如果这就是你计划用`item`做的一切，那么代理会做得很好；不需要命中数据库。换句话说，当`Comment`被保存时，你需要`item`的外键值插入到`COMMENT`表中。`item`的代理正好提供包在一个占位符中的标识符值，这个占位符看起来就像真的一样。

#### 4. 修改持久化对象

由`get()`、`load()`返回的任何持久化对象，或者任何被查询的实体，都已经与当前的`Session`和持久化上下文关联。它可以被修改，并且其状态与数据库同步（请见代码清单9-4）。

**代码清单9-4 修改持久化实例**

```

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

Item item = (Item) session.get(Item.class, new Long(1234));

item.setDescription("This Playstation is as good as new!");

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

```

图9-6展现了这个工作单元和对象转变。

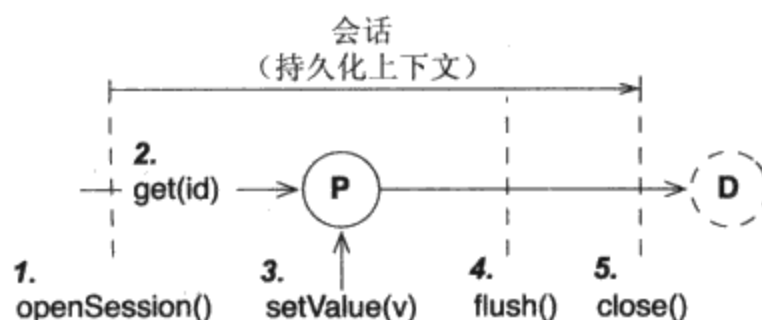


图9-6 修改持久化实例

首先，通过给定的标识符从数据库中获取对象。你修改对象，且当`tx.commit()`被调用时，这些修改在清除期间被传播到数据库。这种机制称作自动脏检查——意味着Hibernate追踪并保存在持久化状态中对一个对象所做的改变。一关闭Session，这个实例就被认为是脱管了。

**5. 使持久化对象变成瞬时**

利用`delete()`方法可以轻松地将一个持久化对象变成瞬时（请见代码清单9-5），从数据库中移除它的持久化状态。

**代码清单9-5 利用delete()使持久化对象变成瞬时**

```

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

Item item = (Item) session.load(Item.class, new Long(1234));

session.delete(item);

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

```

看看图9-7。

调用`delete()`之后，`item`对象处于移除（**removed**）状态；你不应该继续使用它，大多数情况下，应该确保应用程序中对它的任何引用都被移除了。只有当Session的持久化上下文在工作单元结束时与数据库同步，才执行SQL `DELETE`。Session关闭之后，`item`对象被认为是一个普通的瞬时实例。如果这个瞬时实例不再被任何其他对象引用，就会被垃圾收集器破坏。内存对象实例和持久化数据库行都将已经被移除了。

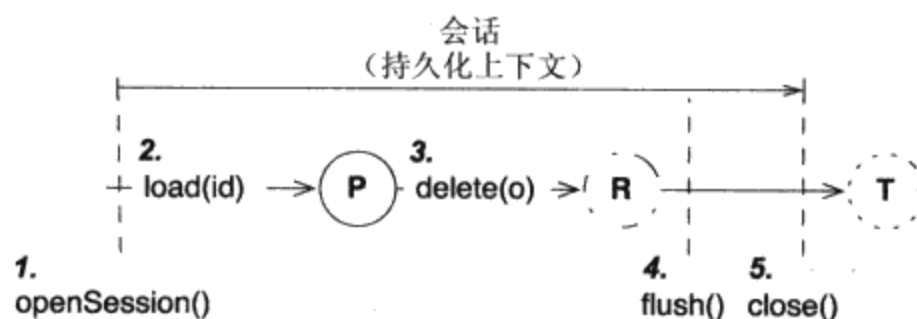


图9-7 使持久化对象变成瞬时

**常见问题** 必须加载要删除的对象吗？是的，对象必须加载到持久化上下文中；实例必须处于要被移除的持久化状态（注意代理就够好了）。原因很简单：你可能已经启用Hibernate拦截器，并且对象必须通过这些拦截器，完成它的生命周期。如果你直接在数据库中删除行，拦截器就不会运行了。话虽这么说，Hibernate（和Java Persistence）还是提供变成直接SQL DELETE语句的大批量操作；12.2节将讨论这些操作。

如果你启用hibernate.use\_identifier\_rollback配置选项，Hibernate也可以回滚已经被删除的任何实体的标识符。在前一个例子中，如果启用这个选项，Hibernate就会在删除和清除之后，设置被删除item的数据库标识符属性为null。然后它就成了一个可以在未来的工作单元中重用的干净的瞬时实例了。

## 6. 复制对象

目前为止，我们介绍在Session中的操作全部都是常见的；每一个Hibernate应用程序都需要它们。但是Hibernate可以用一些特殊的使用案例提供帮助——例如，当你需要从一个数据库获取对象并把它们保存在另一个数据库中的时候。这称作对象的复制（replication）。

复制采用加载在Session中的脱管对象，并在另一个Session中使它们变成持久化。这些Session通常在已经通过映射给同一个持久化类配置的两个不同的SessionFactory中打开。这里有个例子：

```

Session session = sessionFactory1.openSession();
Transaction tx = session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(1234));
tx.commit();
session.close();

Session session2 = sessionFactory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(item, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
  
```

ReplicationMode控制复制过程的细节：

- ❑ ReplicationMode.IGNORE——当现有的数据库行包含与目标数据库中相同的标识符时忽略对象。



- ReplicationMode.OVERWRITE——覆盖任何包含与目标数据库中相同标识符的现有数据库行。
- ReplicationMode.EXCEPTION——如果现有的数据库行包含与目标数据库中相同的标识符时抛出异常。
- ReplicationMode.LATEST\_VERSION——如果目标数据库的版本比对象的版本更早，则覆盖它里面的行，否则忽略对象。你需要启用的Hibernate乐观并发性控制。

当你任由数据进入不同的数据库时，当你在产品升级期间升级系统配置信息时（这经常涉及迁移到一个新的数据库实例），或者当你需要回滚在非ACID事务期间所做的改变时，都可能需要复制。

你现在知道了持久化生命周期和持久化管理器的基础操作。把这些与我们在前面章节中讨论过的持久化类映射一起使用，你就可以创建自己的小Hibernate应用程序了。映射一些简单的实体类和组件，然后在一个独立的应用程序中保存并加载对象。你不需要Web容器或者应用程序服务器：编写一个main()方法，并像我们在前一节讨论过的那样调用Session。

在接下来的几节中，我们讨论脱管对象状态，以及重附和合并持久化上下文之间的脱管对象的方法。这是你实现长工作单元（对话）所需要的基础知识。假设你熟悉本章前面所述的对象同一性范围。

### 9.3.2 使用脱管对象

Session关闭之后修改item，对于它在数据库中的持久化表示法没有影响。持久化上下文一关闭，item就变成了一个脱管实例。

如果想要保存对脱管对象所做的修改，必须重附或者合并它。

#### 1. 重附被修改的脱管实例

脱管实例可以通过在托管对象上调用update()，被重附到新的Session（并由这个新的持久化上下文托管）。依据我们的经验，如果你在心里把update()改名为reattach()可能更容易理解——然而，它被称作更新也是有道理的。

update()方法在数据库中强制更新对象的持久化状态，始终计划一个SQL UPDATE。脱管对象处理的例子请见代码清单9-6。

#### 代码清单9-6 更新脱管实例

```
item.setDescription(...); // Loaded in previous Session

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

sessionTwo.update(item);

item.setEndDate(...);

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

item对象是在传递到update()之前或者之后被修改都没有关系。这里重要的东西是对update()的调用把脱管实例重附到了这个新的Session（和持久化上下文）。Hibernate始终把这个对象当作脏对象来处理，并计划一个SQL UPDATE（将在清除期间执行）。可以在图9-8中看到相同的工作单元。

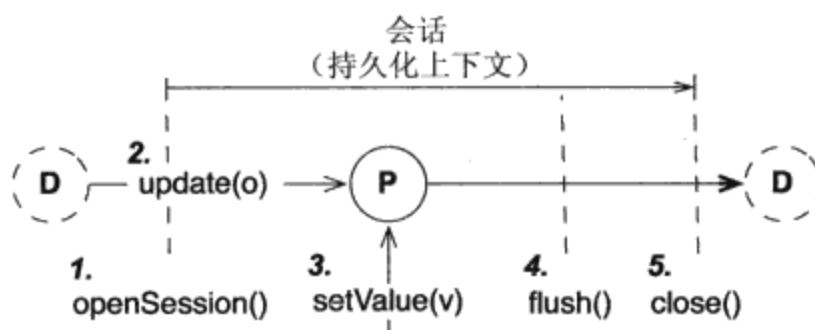


图9-8 重附脱管对象

你可能会很惊讶，或许本以为Hibernate会知道你修改了脱管item的描述（或者Hibernate应该知道你没有修改任何东西）。然而，新的Session和它的新持久化上下文没有这种信息。脱管对象也不包含已经进行的所有修改的一些内部清单。Hibernate只好假设数据库中需要UPDATE。避免这个UPDATE语句的一种方法是，通过select-before-update="true"属性配置Item的类映射。然后Hibernate通过执行一个SELECT语句，并把对象的当前状态与当前的数据库状态进行比较，确定该对象是否为脏对象。

如果确定没有修改脱管实例，你可能会更喜欢另一种重附方法：它不会总是计划一个数据库更新。

## 2. 重附未被修改的脱管实例

对lock()的调用把对象与Session和它的持久化上下文关联起来，不强制更新，如代码清单9-7所示。

### 代码清单9-7 用lock()重附脱管实例

```

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

sessionTwo.lock(item, LockMode.NONE);

item.setDescription(...);
item.setEndDate(...);

tx.commit();
sessionTwo.close();

```

在这个例子中，变化是在对象被重附之前或者之后进行的确很重要。在调用lock()之前进行的改变没有被传播到数据库，只有当你确定脱管实例还没有被修改时才使用它。这种方法只保证对象的状态从脱管变成持久化，并保证Hibernate会再次管理持久化对象。当然，一旦你对处于托管的持久化状态的对象进行任何修改时，就需要更新数据库。

第10章会讨论Hibernate锁模式。这里通过指定LockMode.NONE, 告诉Hibernate不要执行版本检查或在使对象与Session关联时重新获得任何数据库级锁。如果指定了LockMode.READ或者LockMode.UPGRADE, Hibernate就会执行SELECT语句, 以便执行版本检查(并锁定数据库中的行进行更新)。

### 3. 使脱管对象变成瞬时

最后, 可以使脱管实例变成瞬时, 从数据库中删除它的持久化状态, 如代码清单9-8所示。

**代码清单9-8** 利用delete()使脱管对象变成瞬时

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

session.delete(item);

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

这意味着你不一定要重附(利用update()或者lock())脱管实例来把它从数据库中删除。在这个例子中, 调用delete()具有两重功能: 重附对象到Session, 然后计划删除对象(在tx.commit()中执行)。delete()调用之后的对象状态为移除。

脱管对象的重附是在几个Session之间传输数据的唯一可行的方法。可以用另一个选项使对脱管实例的修改与数据库同步, 通过它的状态合并(merging)。

### 4. 合并脱管对象的状态

脱管对象的合并是另一种可以选择的方法。它可以是重附的补充, 也可以取代重附。合并最初引入到Hibernate中, 是用来处理重附无法再满足的特殊案例(merge()在Hibernate 2.x中的旧名称是saveOrUpdateCopy())。看看下列代码, 试着重附一个脱管对象:

```
item.getId(); // The database identity is "1234"
item.setDescription(...);

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

Item item2 = (Item) session.get(Item.class, new Long(1234));

session.update(item); // Throws exception!

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

上述代码给定了一个包含数据库同一性的脱管item对象。修改它之后, 你尝试把它重附到一个新的Session。但在重附之前, 另一个表示相同数据库行的实例已经被加载到了这个Session的持久化上下文中。显然, 通过update()进行的重附与这个已经持久化的实例相抵触, 抛出NonUniqueObjectException。异常的错误信息为: persistent instance with the same database identifier is already associated with the Session! (包含相同数据库标识符的持久化实例已经与会话关联!) Hibernate无法确定哪个对象表示当前状态。

可以先通过重附item解决这种情况; 然后, 由于对象处于持久化状态, 无需获取item2。在

像这个例子一样简单的代码中，这很容易理解，但是它或许不可能在更为复杂的应用程序中进行重构。毕竟，客户端把脱管对象发送到了持久层让它变成托管，并且客户端可能不（且不应该）知道已经处于持久化上下文中的托管实例。

可以让Hibernate自动合并item和item2：

```

item.getId() // The database identity is "1234"
item.setDescription(...);

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

Item item2 = (Item) session.get(Item.class, new Long(1234)); ②
Item item3 = (Item) session.merge(item); ③
(item == item2) // False
(item == item3) // False
(item2 == item3) // True

return item3;

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

```

看看图9-9中的工作单元。

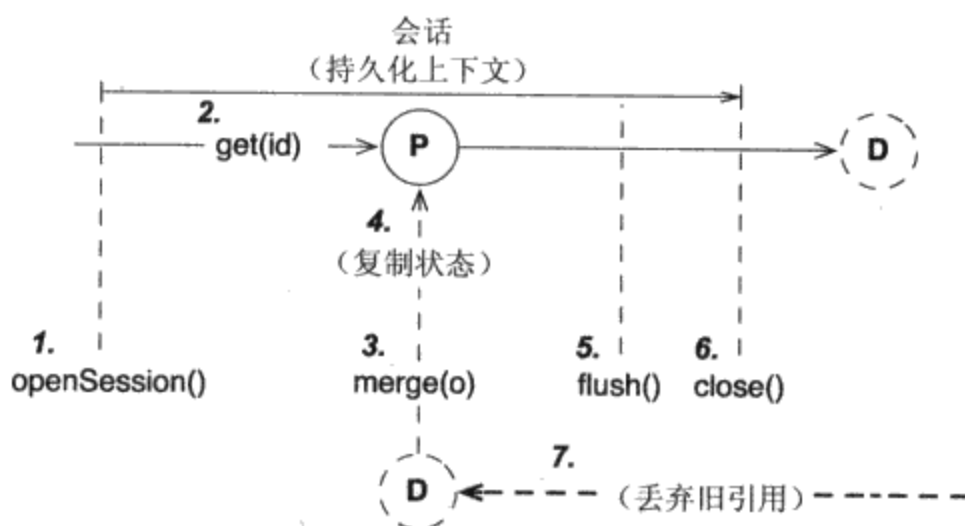


图9-9 把脱管实例合并到持久化实例

merge(item)调用③导致了几个动作。第一，Hibernate检查持久化上下文中的持久化实例是否具有与正在合并的脱管实例相同的数据库标识符。在这个例子中，答案是肯定的：item和item2都用get()②得到加载，它们具有相同的主键值。

如果持久化上下文中有相等的持久化实例，Hibernate把脱管实例的状态复制到持久化实例中去④。换句话说，已经在脱管item上设置的新描述也设置在了持久化item2上。

如果持久化上下文中没有相等的持久化实例，Hibernate就从数据库中加载它（就像你用get()所做的那样，通过标识符有效地执行相同的获取），然后把脱管状态与被获取的对象的状态合并。如图9-10所示。

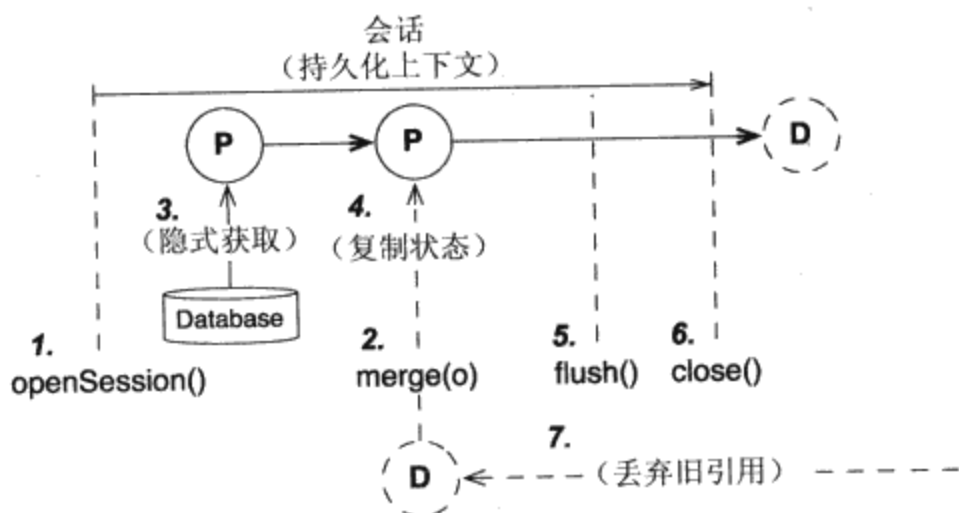


图9-10 把脱管实例合并到一个隐式加载的持久化实例

如果持久化上下文中没有相等的持久化实例，并且在数据库中的查找没有结果，就会创建新的持久化实例，并且把被合并的实例的状态复制到新实例中。然后就计划把这个新对象插入到数据库中，并通过merge()操作返回。

如果传到merge()里面的实例是一个瞬时实例，而不是脱管对象时，也会发生插入。

你可能会有下列疑问：

- 到底从item把什么复制到了item2？合并包括所有值类型的属性，以及任何集合中元素的增加和移除。
- item处于什么状态？与持久化实例合并的任何脱管对象保持脱管状态。它没有改变状态；不受合并操作的影响。因此，item和其他两个引用在Hibernate的同一性范围中是不同的。（前两个同一性在最后一个例子中进行检验）。但是item2和item3是对相同持久化内存实例的相同引用。
- 为什么item3从merge()操作中被返回？merge()操作始终返回一个句柄到它已经与之合并了状态的持久化实例。这对调用了merge()的客户端来说很方便，因为它现在可以继续使用脱管的item对象，并在需要时再次合并它；或者可以放弃这个引用，继续使用item3。区别很明显：Session完成之前，如果后来对item2或者item3所做的修改是在合并之后，客户端就完全不知道这些修改。客户端只有一个对脱管item对象的句柄，现在也过时了。但是，如果客户端决定在合并之后扔掉item，并继续使用返回的item3，在最新的状态上就有了一个新句柄。合并之后item和item2都应该被认为是废弃的。

状态的合并比重附略微复杂一些。我们认为，如果围绕脱管对象设计应用程序逻辑，它就是你有时候可能必须使用的一个基本操作。可以把这个策略作为重附的另一种可选方案，并且每次都合并而不重附。也可以用它使任何瞬时实例变成持久化。如本章稍后所述，这是Java Persistence的标准模型；不支持重附。

目前为止，我们还没有太关注持久化上下文，以及它是如何管理持久化对象的。

### 9.3.3 管理持久化上下文

持久化上下文具有很多功能：自动脏检查、保证对象同一性范围等。了解它管理的一些细节，



和你有时候影响了幕后在进行的东西同等重要。

### 1. 控制持久化上下文高速缓存

持久化上下文是持久化对象的一个高速缓存。持久化状态中的每个对象都为持久化上下文所知，并且每个持久化实例的复制、快照（snapshot）都保存在高速缓存中。这个快照内部用于进行脏检查，侦测对持久化对象所做的任何修改。

忽视这个简单事实的许多Hibernate用户都会遇到OutOfMemoryException。这是当在一个Session中加载上千个对象却永远不想修改它们时的典型案例。Hibernate仍然必须在持久化上下文高速缓存中创建每个对象的快照，并保持对托管对象的引用，它可能导致内存耗尽。（很显然，如果修改上千个对象，就应该执行大批量数据操作（bulk data operation）——12.2节会回到这种工作单元的话题。）

持久化上下文高速缓存从不自动收缩。为了减少或者重新获得被持久化上下文在特定工作单元中消费掉的内存，必须做下列事情：

- ❑ 保持持久化上下文的大小为必需的最小尺寸。Session中的许多持久化实例经常意外出现——例如，因为你只需要几个却查询了许多个。只有当这个状态绝对需要它们时才使对象变成持久化；非常大的图表可能严重影响性能，并需要大量的内存用于状态快照。检查你的查询是否只返回需要的对象。如本书稍后所述，也可以在Hibernate中执行一个查询，返回只读状态的对象，而不用创建持久化上下文快照。
- ❑ 可以调用`session.evict(object)`，从持久化上下文高速缓存中手工脱管一个持久化实例。可以调用`session.clear()`，从持久化上下文中脱管所有持久化实例。脱管对象不用检查脏状态；它们没有被托管。
- ❑ 利用`session.setReadOnly(object, true)`，可以禁用对特定实例的脏检查。如果它是只读的，持久化上下文将不再维持快照。利用`session.setReadOnly(object, false)`，可以给一个实例重新启用脏检查，并强制重新创建快照。注意这些操作并不改变对象的状态。

工作单元结束时，你做过的所有修改都必须通过SQL DML语句与数据库同步。这个过程称作持久化上下文的清除。

### 2. 清除持久化上下文

Hibernate的Session实现了迟写。在持久化上下文的范围中对持久化对象所做的改变并没有立即传播到数据库。这样允许Hibernate把许多改变接合到最少的数据库请求中去，帮助把网络延时的影响减到最小。尽可能迟地（趋向于数据库事务的结束）执行DML的另一个很棒的副作用，在于缩短了数据库内部锁的持续时间。

例如，如果对象的单个属性在同一个持久化上下文中被改变两次，Hibernate就只需要执行一个SQL UPDATE。迟写有用的另一个例子是，当执行多个UPDATE、INSERT或者DELETE语句时，Hibernate能够利用JDBC批量API。

持久化上下文与数据库的同步被称作清除。Hibernate清除发生在以下几个时间点：

- ❑ 当Hibernate API中的Transaction被提交时；
- ❑ 执行查询之前；

□ 应用程序显式地调用`Session.flush()`时。

需要在工作单元结束时把`Session`状态清除到数据库，是为了使变化可以持续，这是常见的案例。注意，事务提交时的自动清除是Hibernate API的一项特性！提交一个包含JDBC API的事务不会触发清除。Hibernate不会在每个查询之前清除。如果变化保存在会影响查询结果的内存中，Hibernate就会默认地先进行同步。

可以通过调用`session.setFlushMode()`，显式地设置Hibernate的`FlushMode`，来控制这个行为。默认的清除模式为`FlushMode.AUTO`，并启用前面所说的行为。如果你选择了`FlushMode.COMMIT`，在查询执行之前，持久化上下文不会被清除（只有在手工调用`Transaction.commit()`或者`Session.flush()`时，它才会被清除）。这个设置可能让你面临废弃的数据：对只处在内存中的托管对象所做的修改，可能与查询的结果冲突。通过选择`FlushMode.MANUAL`，可以指定只有显式地调用`flush()`才导致托管状态与数据库同步。

控制持久化上下文的`FlushMode`，在本书稍后扩展上下文为跨对话时将很有必要。

持久化上下文的重复清除经常成为性能问题的根源，因为持久化上下文中所有的脏对象都必须在清除时间被发现。一种常见的原因在于，特定的工作单元模式多次重复查询—修改—查询—修改的顺序。每一次修改都导致在每个查询之前进行一次清除和所有持久化对象的脏检查。在这种情况下，`FlushMode.COMMIT`可能比较恰当。

永远记着，清除过程的性能部分取决于持久化上下文的大小——它管理的持久化对象的数量。因而，我们在前一节中对管理持久化上下文所提的建议，在这里仍然适用。

现在你已经见过了最重要的策略和在Hibernate应用程序中与对象交互的一些可选的策略，以及在Hibernate Session中可以使用什么方法和操作。如果你准备只利用Hibernate API进行工作，就可以跳过下一节，直接进入第10章学习事务。如果你想利用Java Persistence和（或）EJB 3.0组件处理对象，就接着往下读。

## 9.4 JPA

现在我们用JPA保存和加载对象。这是你可以在Java SE应用程序中或者利用EJB 3.0组件时所用的API，作为Hibernate本地接口一个供应商独立的可选方案。

你已经读完了本章的前面部分，知道JPA定义的对象状态，以及它们如何与Hibernate的对象状态相关。因为这两个非常相似，因此本章第一部分无论选择哪种API都没有关系。它遵循你与对象交互的方法，并且操作数据库的方式也很类似。因此，假设你已经学会了前一节中的Hibernate接口（如果你跳过了前一节，就会没有印象；这里我们不再重复）。这很重要还有另一个原因：JPA提供Hibernate原生API的超集功能的一个子集。换句话说，每当你需要的时候，就有理由退回到原生的Hibernate接口。你可以期待在应用程序中需要的大多数功能都被这个标准涵盖了，这没有太大必要。

就像在Hibernate中一样，你通过JPA操作对象的当前状态来保存和加载对象。并且，就像在Hibernate中一样，你在工作单元中完成它，这被认为是原子的一组操作。（我们仍然还没有涵盖足够的内容来阐述所有关于事务的知识点，但我们很快就会做到。）

为了在一个Java Persistence应用程序中启动工作单元，需要有一个`EntityManager`（相当于

Hibernate的Session)。然而，在Hibernate应用程序中从SessionFactory打开Session的地方，可以用托管和非托管的工作单元编写Java Persistence应用程序。我们要把它保持简单，并假设你首先想要编写一个不从托管环境中的EJB 3.0组件受益的JPA应用程序。

### 9.4.1 保存和加载对象

术语非托管是指用Java Persistence创建持久层的可能性，它不用任何特殊的运行时环境就可以运行和工作。可以不通过应用程序服务器在运行时容器之外、简单的Java SE应用程序中使用JPA。这可以是个servlet应用程序（Web容器不提供你需要用于持久化的东西）或者简单的main()方法。另一个常见的案例是用于桌面应用程序的本地持久化，或者用于两层系统的持久化，在这里桌面应用程序访问远程的数据库层（虽然对于为什么你不能在这样的场景中利用带有EJB支持的、轻量化的模块应用程序服务器，还没有很好的解释。）

#### 1. 在Java SE中启动工作单元

无论如何，由于你没有可以提供EntityManager的容器，因此需要手工创建一个。与Hibernate SessionFactory相当的是JPA EntityManagerFactory：

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("caveatemptorDatabase");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
```

这段代码的第一行是系统配置的一部分。你应该为在Java Persistence应用程序中部署的每个持久化单元都创建一个EntityManagerFactory。我们已经在2.2.2节中涵盖了这一点，因此这里不再重复。接下来的三行相当于在一个独立的Hibernate应用程序中启动工作单元时要做的：首先，创建EntityManager，然后启动事务。为了让自己熟悉EJB 3.0的行话，可以称这个EntityManager为应用程序托管的(application-managed)。这里启动的事务也有一个特殊的描述：它是本地资源(resource-local)的事务。你正在控制着应用程序代码直接涉及的资源（在这个例子中指数据库）；没有运行时容器替你关照这个。

创建EntityManager时，分配了一个新的持久化上下文。你在这个上下文中保存和加载对象。

#### 2. 使实体实例变成持久化

实体类与你其中一个Hibernate持久化类相同。当然，你通常更喜欢用注解映射实体类，取代Hibernate XML映射文件。毕竟，使用Java Persistence的（主要）原因在于标准接口和映射带来的好处。

让我们创建实体的一个新实例，并把它从瞬时状态带到持久化状态：

```
Item item = new Item();
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

em.persist(item);

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

如果你已经理解了本章前面几节的内容，这段代码应该看起来很熟悉了。一旦你在瞬时的item实体实例中调用persist()，它立即就变成持久化；现在它被托管在持久化上下文中。注意，persist()不返回实体实例的数据库标识符值（与Hibernate的save()方法相比的这个差别，在11.2.3节中实现会话时，将会再次变得重要）。

**常见问题** 应该在Session中使用persist()吗？Hibernate的Session接口也特写了persist()方法。它有着与JPA的persist()操作相同的语义。然而关于清除，这两个操作之间有着重要的差别。在同步期间，即使用这个选项映射了一个关联，Hibernate的Session也不会把persist()操作级联到被关联的实体和集合。它只是被级联到调用persist()时可以到达的实体！如果使用Session API，则只有save()（和update()）在清除时被级联。但是在JPA应用程序中则相反：只有persist()在清除时被级联。

被托管的实体实例受到监控。你在持久化状态中对实例所做的修改在某个时间点与数据库同步（除非放弃工作单元）。由于EntityTransaction由应用程序托管，你需要手工进行commit()。应用相同的规则到应用程序控制的EntityManager：你要通过关闭它来释放所有资源。

### 3. 获取实体实例

EntityManager也用来查询数据库，并获取持久化的实体实例。Java Persistence支持复杂的查询特性（我们将在本书稍后讨论到）。最基础的部分和往常一样，是通过标识符获取（retrieval by identifier）：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
tx.commit();
em.close();
```

你不需要转换find()操作的返回值；它是个一般的方法，其返回类型设置为第一个参数的一个附带作用。这是JPA一个很小但很方便的好处——Hibernate原生的方法必须与不支持泛型的更早期的JDK一起工作。

被获取的实体实例处于持久化状态，现在可以在工作单元内部被修改，或者脱管在持久化上下文之外使用。如果找不到包含指定标识符的持久化实例，find()就返回null。find()操作始终命中数据库（或者一个特定于供应商的透明高速缓存），因此实体实例始终在加载期间被初始化。可以期待稍后它所有的值都可以在脱管状态中使用。

如果不想命中数据库，因为你不确定是否需要完全初始化的实例，可以告诉EntityManager尝试获取一个占位符：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.getReference(Item.class, new Long(1234));
```

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

这个操作返回被完全初始化的item（例如，如果当前的持久化上下文中已经有实例）或者一个代理（空的占位符）。

一旦你试图访问不是数据库标识符属性的item的任何属性时，就立即执行一个额外的SELECT来完全初始化这个占位符。这也意味着在这时候（或者甚至更早，执行getReference()的时候）应该会有一个EntityNotFoundException。合乎逻辑的结论是，如果决定脱管item引用，就不保证它会被完全初始化（当然，除非在脱管之前访问它其中一个非标识符属性）。

#### 4. 修改持久化的实体实例

处于持久化状态的实体实例由当前的持久化上下文管理。可以修改它，并期待持久化上下文在同步时清除必要的SQL DML。这是由Hibernate Session提供的相同的自动脏检查特性：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
item.setDescription(...);

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

持久化实体实例通过它的标识符值获取。然后，你修改其中一个被映射的属性（还没有用@Transient或者transient的Java关键字注解的一个属性）。在这个代码实例中，下一次与数据库的同步发生在本地资源事务被提交的时候。Java Persistence引擎执行必要的DML，在这个例子中是一个UPDATE。

#### 5. 使持久化实体实例变成瞬时

如果想要从数据库中移除一个实体实例的状态，就必须使它变成瞬时状态。在EntityManager中使用remove()方法：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));

em.remove(item);

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

Java Persistence remove()方法的语义与Hibernate Session中的delete()方法相同。之前的持久化对象现在处于移除状态，你应该放弃应用程序中存有对它的任何引用。在持久化上下文的下一个同步期间执行一个SQL DELETE。JVM垃圾收集器侦测item不再被任何人引用，并最终删除该对象的最后轨迹。但是注意，不能在脱管状态中在一个实体实例上调用remove()，否则会抛出异常。你必须先合并脱管实例，然后移除被合并的对象（或者，利用相同的标识符获得一个引用，并移除之）。



## 6. 清除持久化上下文

对持久化实体实例所做的所有修改都在某个时间点与数据库同步，这个过程称作清除。这种迟写行为与Hibernate的相同，通过尽可能迟地执行SQL DML来保证最佳的可伸缩性。

每当EntityTransaction中的commit()被调用时，就清除EntityManager的持久化上下文。本章这一节中所有之前的代码示例都已经使用了这一策略。但是JPA实现被允许在其他的时间点上与持久化上下文同步，如果它们希望如此的话。

Hibernate作为一个JPA实现，在下列时间点同步：

- 当EntityTransaction被提交时；
- 执行查询之前；
- 当应用程序显式地调用em.flush()时。

这些规则与我们在前一节中对原生Hibernate阐述的一样。就像在原生的Hibernate中一样，可以用JPA接口FlushModeType控制这个行为：

```
EntityManager em = emf.createEntityManager();
em.setFlushMode(FlushModeType.COMMIT);
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
item.setDescription(...);

List result = em.createQuery(...).getResultList();

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

对于EntityManager把FlushModeType转换为COMMIT，就会在查询之前禁用自动同步；它只在提交事务或者手工清除时才会发生。默认的FlushModeType是AUTO。

就像利用原生的Hibernate一样，控制持久化上下文的同步行为对于会话的实现将是很重要的功能，我们稍后会讨论到。

你现在知道了Java Persistence的基本的操作，可以继续在自己的应用程序中保存和加载一些实体实例。像2.2节中描述的一样，建立系统，并利用注解映射一些类到数据库模式中。编写使用EntityManager和EntityTransaction的main()方法；相信你很快就会明白，使用Java Persistence甚至不用EJB 3.0托管组件或者应用程序服务器有多么容易。

接下来讨论如何使用脱管的实体实例。

### 9.4.2 使用脱管的实体实例

假设你已经知道如何定义脱管对象（如果不知道，再读一遍本章的第一小节）。你没有必要知道如何在Hibernate中使用脱管对象，但如果使用Java Persistence的策略与原生Hibernate中的一样，我们就会让你参考之前的章节。

首先，再来看看实体实例在Java Persistence应用程序中如何变成脱管状态。

#### 1. JPA持久化上下文范围

你已经在Java SE环境中使用过Java Persistence，利用应用程序管理的持久化上下文和事务。

每一个持久化的和托管的实体实例都在持久化上下文关闭时变成脱管状态。但是我们并没有告诉过你持久化上下文什么时候关闭。

如果你熟悉原生的Hibernate，就已经知道了答案：当Session关闭时，持久化上下文终止。EntityManager是JPA中的等价物；默认情况下，如果你创建了EntityManager，持久化上下文就会被界定到这个EntityManager实例的生命周期。

看看下列代码：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
Item item = em.find(Item.class, new Long(1234));
tx.commit();

item.setDescription(...);

tx.begin();
User user = em.find(User.class, new Long(3456));
user.setPassword("secret");
tx.commit();

em.close();
```

在第一个事务中，你获取了Item对象。然后事务完成，但item仍然处于持久化状态。因而，在第二个事务中，你不仅加载了User对象，而且在提交第二个事务时更新了被修改的持久化item（除了对脏user实例的更新之外）。

就像在原生的Hibernate中使用Session一样，持久化上下文始于createEntityManager()，止于close()。

关闭持久化上下文不是脱管实体实例的唯一途径。

## 2. 实体实例的手工脱管

实体实例在它离开持久化上下文时变成脱管状态。EntityManager中的一种方法允许你清除持久化上下文，并脱管所有持久化实例：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
Item item = em.find(Item.class, new Long(1234));

em.clear();

item.setDescription(...); // Detached entity instance!

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

获取item之后，你清除EntityManager的持久化上下文。由这个持久化上下文管理的所有实体实例现在都脱管了。脱管实例的修改不在提交期间与数据库同步。

**常见问题** 单独实例的清除在哪里？Hibernate Session API具备`evict(object)`方法，Java Persistence则没有这个能力。原因可能只有一些专家组成员才知道——我们无法解释。（注意，这种方式很好地说明了专家们不可能对该操作语义达成一致意见。）可以只完全清除持久化上下文，并脱管所有的持久化对象。如果你想要从持久化上下文中清除单独的实例，就必须退回到2.2.4节中所述的Session API。

很显然，你也想要保存在某个时间点对脱管实体所做的修改。

### 3. 合并脱管的实体实例

尽管Hibernate提供了重附和合并两种策略使脱管对象的任何变化与数据库同步，但Java Persistence却只提供后者。假设你已经在前一个持久化上下文中获取了一个`item`实体实例，并且现在想修改它并保存这些修改：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
tx.commit();
em.close();

item.setDescription(...); // Detached entity instance!

EntityManager em2 = emf.createEntityManager();
EntityTransaction tx2 = em2.getTransaction();
tx2.begin();

Item mergedItem = (Item) em2.merge(item);

tx2.commit();
em2.close();
```

`item`在第一个持久化上下文中被获取，并且在脱管状态中的修改后，合并到一个新的持久化上下文。`merge()`操作完成几件事情：

首先，Java Persistence引擎检查持久化上下文中的持久化实例是否与你正在合并的脱管实例有着相同的数据库标识符。因为，在我们的代码示例中，第二个持久化上下文中没有相等的持久化实例，有一个通过标识符进行查找从数据库中被获取。然后，脱管的实体实例被复制到持久化实例。换句话说，已经在脱管的`item`中设置的新描述也在持久化的`mergedItem`上设置了，它从`merge()`操作中返回。

如果在持久化上下文中没有相等的持久化实例，并且在数据库中通过标识符进行的查找是否定的，被合并的实例就被复制到一个新的持久化实例，然后当第二个持久化上下文与数据库同步时就被插入到数据库。

状态的合并是重附的另一种选择（与原生的Hibernate提供的一样）。参考我们前面在9.3.2节中讨论的利用Hibernate的合并；这两个API提供相同的语义，若作必要的修正（*mutatis mutandis*），其中的注释将适用于JPA。

你现在准备扩展Java SE应用程序，并在Java Persistence中体验使用持久化上下文和脱管对象。不是只在单个工作单元中保存和加载实体实例，而是尝试使用几个工作单元，尝试合并脱管对象的修改。别忘了观察SQL日志，看看幕后发生了什么。

一旦你掌握了使用Java SE的基础Java Persistence操作，就可能想要在托管环境中做同样的事。从完全EJB 3.0容器的JPA中得到的好处很有价值。你不再需要自己管理EntityManager和EntityTransaction。可以关注自己想要做的事：加载和保存对象。

## 9.5 在 EJB 组件中使用 Java Persistence

托管的运行时环境仿佛是某种容器。应用程序组件就存在这个容器里面。当今的大部分容器都利用一种拦截技术来实现，拦截对象上的方法调用，并应用需要在这个方法之前（或者之后）执行的任何代码。这对任何横切关注点（cross-cutting concern）来说都是最好的：打开和关闭EntityManager是必然的，因为你要它处在被调用的方法里面。你的业务逻辑不需要关注这个方面。事务划分是容器可以替你负责的另一个关注点。（你可能在任何应用程序中发现其他的方面。）

不像EJB 2.x时代更早的应用程序服务器，支持EJB 3.0和其他Java EE 5.0服务的容器很容易安装和使用——请参考2.2.3节，为下一节准备好系统。此外，EJB 3.0编程模型以简单的Java类为基础。如果你见到我们在本书中编写许多EJB，不应该感到惊讶；大多数时候，与简单的JavaBean的唯一区别仅仅在于一个简单的注解，一项你希望使用的由运行组件的环境所提供的一项服务的声明。如果你无法修改源代码和添加注解，可以用XML部署描述符把一个类变成一个EJB。因而，（几乎）每个类都可以成为EJB 3.0中的托管组件，使你更容易从Java EE 5.0服务中受益。

目前为止你已经创建的实体类还不足以编写一个应用程序。你还要无状态或者有状态的会话bean、可以用来封装应用程序逻辑的组件。在这些组件内部，你需要容器的服务：例如，你通常要容器注入（inject）EntityManager，以便可以加载和保存实体实例。

### 9.5.1 注入 EntityManager

还记得在Java SE中如何创建EntityManager的实例吗？必须手工从EntityManagerFactory中打开和关闭它。还必须用EntityTransaction接口启动和终止本地资源的事务。

在EJB 3.0服务器中，容器管理的EntityManager可以通过依赖注入（dependency injection）使用。考虑在CaveatEmptor应用程序中实现特定动作的下列EJB会话bean：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    // Use field injection:
    @PersistenceContext
    private EntityManager em;

    // or setter injection:
    //
    // @PersistenceContext
    // public void setEntityManager(EntityManager em) {
    //     this.em = em;
    }
```

```
// }

@Transactional(TransactionalAttributeType.REQUIRED)
public Item findAuctionByName(String name) {
    return (Item) em.createQuery()...
    ...
}
}
```

这是个无状态的动作，它实现ManageAuction接口。无状态EJB的这些细节不是我们目前的关注点，值得关注的是可以在这个动作的findAuctionByName()方法中访问EntityManager。容器在动作方法执行之前，自动把EntityManager的一个实例注入到bean的em字段里面。字段的可见性对于容器来说并不重要，但是需要应用@PersistenceContext注解来表明你想要这个容器的服务。也可以给这个字段创建一个公有的设置方法，并在这个方法上应用注解。如果你还计划手工设置EntityManager，建议采用这种方法——例如，在集成测试或者功能测试期间。

被注入的EntityManager由这个容器维护。你不必清除或者关闭它，也不必启动和终止事务——你在前一个例子中告诉容器：会话bean的findAuctionByName()方法需要一个事务。（对于所有EJB会话bean方法而言，这是默认的。）当方法被客户端调用（或者新事务自动启动）时，事务必须是活动的。当方法返回时，事务要么继续，要么被提交，取决于它是否这个方法所启动。

被注入的容器管理的EntityManager的持久化上下文由事务的范围决定。因而，当事务终止时它也自动被清除和关闭。这是个重要的区别，如果把它与前面在Java SE中介绍的例子进行比较的话。那里的持久化上下文没有被界定到事务，而是界定到你显式地关闭的EntityManager实例。事务范围的持久化上下文对于无状态bean是自然的默认，就像当你稍后关注对话实现和事务时，在接下来的章节中所看到的那样。

显然只对使用JBoss EJB 3.0有效的一种很好的技巧是Session（而不是EntityManager对象的自动注入）：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext
    private Session session;
    ...
}
```

如果有一个托管组件要依赖Hibernate API，这最有用。

这是对两个数据库（也就是说，两个持久化单元）都有效的一种变形：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(unitName = "auctionDB")
    private EntityManager auctionEM;

    @PersistenceContext(unitName = "auditDB")
    private EntityManager auditEM;

    @Transactional(TransactionalAttributeType.REQUIRED)
    public void createAuction(String name, BigDecimal price) {
```

```
        Item newItem = new Item(name, price);
        auctionEM.persist(newItem);
        auditEM.persist( new CreateAuctionEvent(newItem) );
        ...
    }
}
```

unitName指向被配置和部署的持久化单元。如果你使用数据库（EntityManager或者SessionFactory），就不需要给注入声明持久化单元的名称。注意，来自两个不同持久化单元的EntityManager实例不共用相同的持久化上下文。一般地，两个都是托管实体对象的独立高速缓存，但这并不意味着它们不能参与相同的系统事务。

如果用Java Persistence编写EJB，你的选择就很清楚了：想要包含正确持久化上下文的EntityManager通过容器注入到托管组件。你很少使用的另一种方法是查找（lookup）容器托管的EntityManager。

### 9.5.2 查找 EntityManager

不是让容器在字段或者设置方法中注入EntityManager，而是当你需要它的时候从JNDI中查找：

```
@Stateless
@PersistenceContext(name = "em/auction", unitName = "auctionDB")
public class ManageAuctionBean implements ManageAuction {

    @Resource
    SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
        EntityManager em = (EntityManager) ctx.lookup("em/auction");

        return (Item) em.createQuery()...
    }
}
```

在这个代码片段中发生了几件事情：第一，你声明了想要bean的组件环境中填充有EntityManager，并且假定绑定引用的名称为em/auction。JNDI中的全称为java:comp/env/em/auction——java:comp/env/部分就是所谓的bean命名上下文（bean-naming context）。在JNDI这个子上下文（subcontext）中的一切都是依赖bean。换句话说，EJB容器读取这个注解，并知道它在这个bean执行的运行时必须只给该bean绑定EntityManager，在为这个bean保存的JNDI中的命名空间下。

利用SessionContext，在bean实现中查找EntityManager。这个上下文的好处在于它自动用java:comp/env/给正在查找的名称加上前缀；因而，它尝试在bean的命名上下文中查找引用，而不是全局的JNDI命名空间。@Resource注解指示EJB容器注入SessionContext。

当EntityManager中的第一个方法被调用时，持久化上下文由容器创建；并且当事务终止（当这个方法返回）时，它被清除和关闭。



如果需要EntityManagerFactory, 也可以使用注入和查找。

### 9.5.3 访问 EntityManagerFactory

EJB容器也允许你直接为持久化单元访问EntityManagerFactory。没有托管环境, 你必须在Persistence引导类 (bootstrap class) 的帮助下创建EntityManagerFactory。在容器中, 可以再次利用自动的依赖注入获得EntityManagerFactory:

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceUnit(unitName = "auctionDB")
    EntityManagerFactory auctionDB;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
        EntityManager em = auctionDB.createEntityManager();
        ...
        Item item = (Item) em.createQuery()...
        ...
        em.flush();
        em.close();
        return item;
    }
}
```

unitName属性是可选的, 只有当你有不止一个被配置的持久化单元 (几个数据库) 时才需要它。你从被注入的工厂中创建的EntityManager再次成为应用程序托管的——容器将不删除这个持久化上下文, 也不会关闭它。你很少把容器托管的工厂与应用程序托管的EntityManager实例混合, 但是如果你需要对EJB组件中EntityManager的生命周期的更多控制时, 这么做就很有帮助。

可以在任何JTA事务范围之外创建EntityManager; 例如, 在一个不需要事务上下文的EJB方法中。然后必要时, 利用joinTransaction()方法通知EntityManager一个JTA事务是活动的, 这就是你的责任了。注意, 这个操作没有把持久化上下文绑定或者界定到JTA事务; 它只是内部转换EntityManager到事务行为的一个提示。

前面的语句还没有完成: 如果close()这个EntityManager, 它不会立即关闭它的持久化上下文, 如果这个持久化上下文已经与一个事务关联的话。当事务结束时, 持久化上下文关闭。但是, 对被关闭的EntityManager的任何调用都会抛出异常 (除了Java SE中的getTransaction()方法和isOpen()方法之外)。可以用hibernate.ejb.discard\_pc\_on\_close配置设置转换这个行为。如果你永远不在事务范围之外调用EntityManager, 就不必担心这一点。

访问EntityManagerFactory的另一个原因, 可能是你想在这个接口中访问一个特定的供应商扩展, 就像我们在2.2.4节中讨论过的一样。

如果先把EntityManagerFactory绑定到EJB的命名上下文中, 也可以查找它:

```
@Stateless
@PersistenceUnit(name= "emf/auction", unitName = "auctionDB")
public class ManageAuctionBean implements ManageAuction {
```

```

@Resource
SessionContext ctx;

@Transactional(TransactionalAttributeType.REQUIRED)
public Item findAuctionByName(String name) {
    EntityManagerFactory auctionDB =
        (EntityManagerFactory) ctx.lookup("emf/auction");

    EntityManager em = auctionDB.createEntityManager();
    ...
    Item item = (Item) em.createQuery()...
    ...

    em.flush();
    em.close();
    return item;
}
}

```

如果把这个查找方法与自动的注入进行比较，会发现它还是没有特别的优势。

## 9.6 小结

本章已经谈到了不少内容。你现在知道了Java Persistence中的基本接口与Hibernate提供的那些没有多大区别。加载和保存对象几乎相同。但是持久化上下文的范围略有不同；在Hibernate中，默认与Session范围相同。在Java Persistence中，持久化上下文的范围很不相同，取决于你是自己创建EntityManager，还是让容器管理并它把绑定到EJB组件中的当前事务范围。

表9-1展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概述。

表9-1 第9章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate定义和依赖4种对象状态：瞬时、持久化、删除和脱管	相当的对象状态在EJB 3.0中被标准化和定义
脱管对象可以被重附到一个新的持久化上下文或者被合并到持久化实例中	Java Persistence管理接口只支持合并
在清除时，save()和update()操作可以被级联到所有被关联的和可获得的实例。persist()操作只能被级联到调用时可获得的实例	在清除时，persist()操作可以被级联到所有被关联的和可获得的实例。如果退回到Session API，save()和update()也只能被级联到调用时可获得的实例
get()命中数据库；load()可能返回代理	find()命中数据库；getReference()可以返回代理
EJB中Session的依赖注入只在JBoss应用程序服务器中有效	EntityManager的依赖注入在所有EJB组件中都有效

我们已经讲到了应用程序中的对话，以及如何利用脱管对象或者利用被扩展的持久化上下文设计它们。虽然我们还没有时间非常深入地进行讨论，但你可能已经看到，使用脱管对象工作需要规定（在受保护的對象同一性范围之外）和手工重附或者合并。在实际应用程序中，依据我们多年的经验，建议把脱管对象当作次要的方法，并且先看一下包含被扩展的持久化上下文的对话实现。

遗憾的是，我们仍然没有利用所有的东西编写一个包含对话的真正复杂的应用程序。你可能特别想知道更多有关事务的信息。第10章涵盖了事务的概念和接口。