Hibernate In Action

中文版

第四章 操作持久对象

4.1 持久化生命周期

由于 Hibernate 是透明的持久化机制—类不能意识到它们自己的持久能力—编写应用逻辑时不用意识到你所操纵的对象是持久状态还是存在于内存中的临时状态。当应用调用对象的方法时不需要关心它的状态是否是持久的。

然而，在持久状态的应用中，只要应用需要把内存中的状态传到数据库（反之亦然）就必须同持久层打交道。你可以调用 Hibernate 持久化管理器和查询接口来完成这种操作。当使用那种方式同持久层打交道时，应用关心与持久化相关的对象的状态及生命周期是必要的。我们将把它称为持久化生命周期。

对于持久化生命周期，不同的 ORM 实现使用不同的术语，定义不同的状态及状态转换。此外，内部使用的对象状态可能与其暴露给客户端应用的状态不同。Hibernate 仅仅定义了三种状态：瞬时、持久和分离，对客户端代码隐藏了其内部实现的复杂性。这一章，我们解释这三种状态：瞬时、持久和分离。

让我们在状态图中看看这些状态和它们的转换，如图 4.1 所示。你也可以看到调用持久管理器触发转换的方法。在这一节我们讨论这张图；以后你无论什么时候需要一个综述都可以引用它。

在其生命周期中，对象可以从瞬时对象转换到持久对象，再转换到分离对象。让我们仔细看看这些状态中的每一个状态。

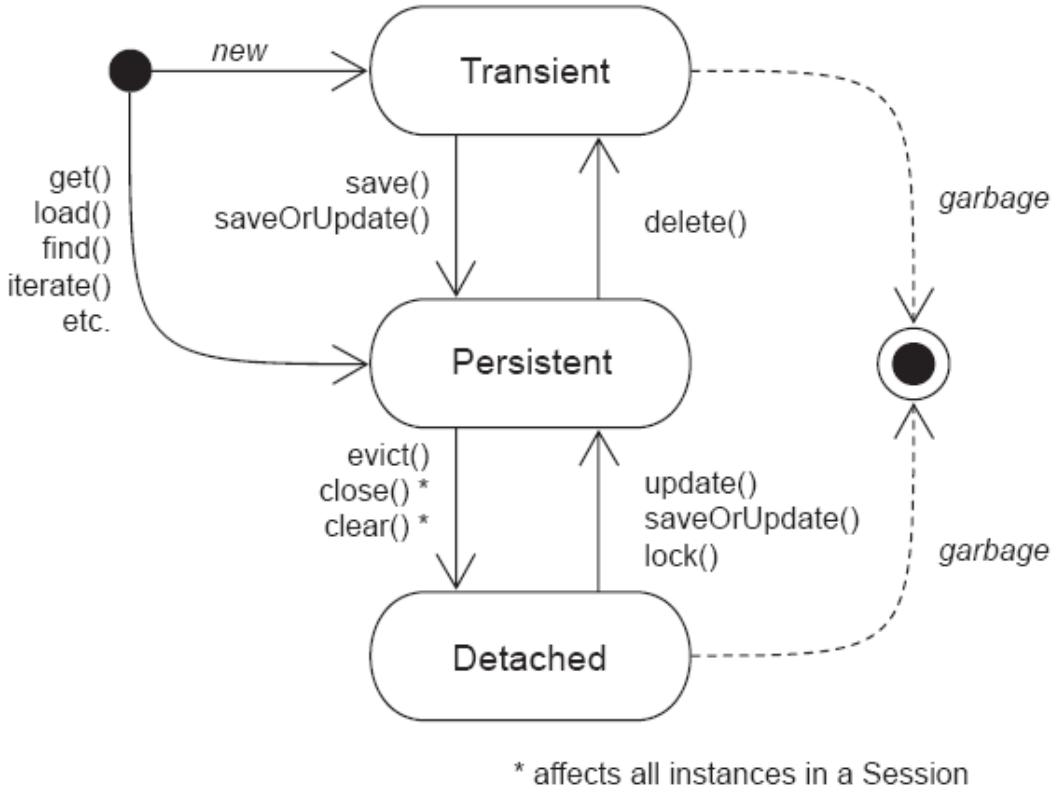

图 4.1Hibernate 中的对象状态及转换

4.1.1 瞬时对象

在 Hibernate 中，使用 new 操作符初始化的对象不是立刻就是持久的。它们的状态是瞬时的，也就是说它们没有跟任何数据库表的行相关联，只要应用不再引用这些对象（不再被任何其它对象所引用），它们的状态将会丢失。这时，那些对象的生命期将会有效地终止，变得不可访问，交给垃圾回收机制来回收。

Hibernate 认为所有的瞬时实例都是非事务的，瞬时实例状态的修改不能在任何事务的上下文中执行。这就意味着 Hibernate 不能对瞬时对象提供任何回滚功能。（实际上 Hibernate 不回滚任何对象所做的修改，以后你就会看到）。

默认地，仅仅被其他瞬时实例引用的对象也是瞬时的。把实例从瞬时状态转换为持久状态有两种方式：调用持久管理器的 save()方法或者从已经存在的持久实例中创建引用。

4.1.2 持久对象

持久实例是任何具有数据库标识的实例，就像第三章第 3.4 节“理解对象标识符”所定义的那样。也就是持久实例有一个主键值设为数据库标识符。

持久实例可能是由应用程序初始化的对象调用持久管理器（Hibernate Session，本章的后面部分将会详细讨论）的 save()方法实现持久化。然后，持久实例就跟持久管理器关联起来。它们甚至可能是引用另一个已经与持久管理器相关联的持久对象来实现持久化。可选的，持久实例可能是通过执行查询，通过标识符查找从数据库中检索出来的实例，或者是从另一个持久实例开始导航对象图。换句话说，持久实例通常是同 Session 相关的，是事务的。

持久实例是在事务中进行操作的—它们的状态在事务结束时同数据库进行同步。当事务提交时，通过执行 SQL 的 INSERT、UPDATE 和 DELETE 语句把内存中的状态同步到数据库中。这个过程也可能在其它时间发生。例如，Hibernate 在执行查询之前可能要同数据库同步。这就确保查询能够意识到在事务早期所做的更改。

我们称已经分配主键值但是还没有插入到数据库中的持久实例是新的。新的持久实例将会仍然保持“新”的状态直到同步发生。

当然，你不必在事务结束时把内存中的每个持久对象的状态更新到数据库中对应的行上。ORM 软件必须有一种机制来检测哪个持久对象已经被应用程序在事务中修改了。我们称其为自动脏数据检查（修改过的对象还没有同步到数据库中被任为是脏的）。这种状态对于应用程序来说是不可见的。我们把这种特性称为 *transparent transaction-level write-behind*，意思是 Hibernate 尽可能晚地把改变的状态同步到数据库中，但是对应用程序隐藏其实现细节。

Hibernate 能够确切地知道哪个属性改变了，因此可以在 SQL UPDATE 语句中仅包含那些需要更新的列。这样做可以提高性能，对于一些特定的数据库尤其如此。然而，这样做并不是总能得到性能提升，理论上，在某些环境中，可能要降低一些性能。因此，Hibernate 在 SQL UPDATE 语句中默认包含所有的列（因此，Hibernate 在启动时能够生成这个基本的 SQL，而不是在运行时）。如果仅仅希望更新修改过的列，你可以在类映射中通过把 dynamic-update 设

为 true 启动动态 SQL 生成功能。（注意，此特性在手工编码的持久层中是非常难以实现的。）下一章，我们详细讨论 Hibernate 的事务概念和同步过程（即 flushing）。

最后，持久实例通过调用持久管理器 API 的 delete()方法使其变成瞬时的，导致删除数据库中相应的行。

4.1.3 分离对象

当事务结束时，同持久管理器相关联的持久实例仍然存在（如果事务成功，它们在内存中的状态将会同数据库同步）。在具有过程范围标识（*process-scoped identity*，看下一节）的 ORM 解决方案中，这些实例保留着同持久管理器的关联并且仍然认为是持久的。

可是在 Hibernate 中，当你关闭 Session 时这些实例就失去了同持久管理器的关联。我们把这些对象称为分离的，表明这些状态不再跟数据库中的状态同步，不再在 Hibernate 的管理下。然而，它们仍然含有持久数据（可能是稳定的）。应用程序可能（通常）含有事务（及持久管理器）之外的分离对象的引用。Hibernate 可以让你同新的持久管理器重新关联这些实例以便在新事务中重用这些实例（重新关联后，这些对象被认为是持久的）。这种特性对怎样设计多层应用有很大影响。从一个事务中返回对象到表示层，以后在新的事务中重用它们的能力是 Hibernate 的一个主要卖点。我们在下一章讨论这种作为对于长时间运行的应用事务（*application transaction*）的一种实现技术的使用方法。我们也在第八章“重新考虑传输对象”这一节告诉你怎样通过使用分离的对象避免 DTO（anti-）模式。

Hibernate 也提供了一个显式的分离操作：Session 的 evict()方法。然而，这个方法只在 cache 管理中使用（考虑性能问题）。通常不显式地执行分离。然而，所有在事务中检索出来的对象在当 Session 关闭或当它们被序列化时（例如，它们被远程传递）就变成分离的。因此，Hibernate 不需要提供控制子图分离的功能。然而，应用程序能够使用查询语言或显式的图表导航来控制抓取子图（当前装载在内存中的实例）的深度。那么，当 Session 关闭时，整个子图（所有与持久管理器关联的对象）就会变成分离的。

让我们再来看看不同的状态，但是这次考虑对象标识的范围。

4.1.4 对象标识的范围

作为应用程序开发者，我们使用 Java 对象标识（a=b）来标识对象。因此，如果对象改变了状态，那么能够保证在新的状态中它的 Java 标识仍然相同么？在分层的应用程序中可能不会这样。

为了研究这个话题，有必要理解 Java 标识 a=b 与数据库标识 a.getId().equals(b.getId())之间的关系。有时它们是相等的，有时却不相等。我们把 Java 标识等于数据库标识的情况称为对象标识范围（scope of object identity）。

在这个范围内，通常有三种选择：

- 没有标识范围（*no identity scope*）的简单持久层不能保证如果某一行被访问两次，应用程序能够返回相同的 Java 对象实例。如果应用程序在一个单独的事务中修改了代表相同行的两个不同的实例将会出问题（你怎样决定哪一个状态将会同数据库同步？）。
- 使用事务范围标识（transaction-scoped identity）的持久层保证在单独的事务上下文中，仅仅有一个对象实例代表数据库中的某一行。这就避免了前面那个问题，并且允许做一些事务级的缓存。
- 过程范围标识（process-scoped identity）更进一步，它能保证在整个过程（JVM）中只有一个对象实例代表某一行。

对于典型的网络或企业应用程序，事务范围标识是首选的。过程范围标识在利用缓存和多个事务重用实例的编程模型方面有一些潜在的优点；然而在普遍的多线程应用程序中，同步共享访问全局标识图中的持久对象要花费很大代价。在每个事务范围内每个线程只同完全不同的一组持久实例工作将会更加简单、更易于升级。

宽松地讲，Hibernate 执行事务范围的标识。实际上，Hibernate 标识范围是 Session 的实例，如果对象在几个操作中使用相同的持久管理器（the Session）就能够保证这些对象是相等的。但是 Session 同（数据库）事务是不一样的—它是一个更复杂的元素。我们将会在下一章探索

这个概念的不同及结果。让我们再次关注持久化生命周期和标识范围。

如果你在同一个 Session 中使用相同的数据库标识符值请求两个对象，结果将会是对同一个内存对象的两个引用。下面的代码示例在两个 Session 中用几个 load()操作演示这种行为：

```
Session session1 = sessions.openSession();
Transaction tx1 = session1.beginTransaction();
// Load Category with identifier value "1234"
Object a = session1.load(Category.class, new Long(1234) );
Object b = session1.load(Category.class, new Long(1234) );
if ( a==b ) {
System.out.println("a and b are identical.");
}
tx1.commit();
session1.close();
Session session2 = sessions.openSession();
Transaction tx2 = session2.beginTransaction();
Object b2 = session2.load(Category.class, new Long(1234) );
if ( a!=b2 ) {
System.out.println("a and b2 are not identical.");
}
tx2.commit();
session2.close();
```

由于对象引用 a 和 b 在相同的 Session 中装载，它们不仅有相同的数据库标识，而且有相同的 Java 标识。然而，一旦超出了这种界限，Hibernate 就不能保证 Java 标识是相等的，因此，a 和 b2 是不相等的，信息打印在控制台上。当然，测试数据库标识--a.getId().equals (b2.getId())—将仍然返回 true。

为了更深入讨论标识范围，我们需要考虑持久层怎样持有对标识范围外的对象的引用。例如，对于像 Hibernate 这样的事务范围标识的持久层，对分离的对象（那就是在以前完成了的 Session 中持久或装载过的实例）进行引用是否是可容忍的？

4.1.5 标识范围之外

如果对象引用离开了能够得到保证的标识范围，我们把它称为对分离对象的引用。为什么这方面内容很有用呢？

在 web 应用程序中，通常不维护跨用户交互的数据库事务。用户花很长时间思考怎样修改，由于升级性的原因，你必须让数据库事务短而且尽可能快地释放数据库资源。在这种环境中，能够重用对分离实例的引用非常有用。例如，你可能希望把在一个工作单元中检索的对象传给表示层，以后，用户修改过它之后在第二个工作单元重用它。

通常你不希望在第二个工作单元重新绑定整个对象图，由于性能（或其它）原因，有选择地重新关联分离的实例很重要。Hibernate 支持选择性地重新关联分离实例（*selective reassociation of detached instances*）。这意味着应用程序能有效地将分离对象图的子图重新绑定到当前（或第二个）Hibernate Session。一旦分离对象重新绑定到新的 Hibernate 持久管理器，这个对象就会被认为是持久实例，它的状态将会在事务结束时同数据库同步（由于 Hibernate 自动检查持久实例的脏数据）。

当创建从分离实例到新的瞬时实例的引用时，重新绑定可能会导致在数据库中创建新的行。例如，新的 Bid 在表示层的时候就可能已经加到分离的 Item。Hibernate 能够察觉到 Bid 是新的，必须插入到数据库中。为了使其工作，Hibernate 必须能够分辨“新”的瞬时实例和“老”的分离实例。瞬时实例（如 Bid）可能需要保存；分离实例（如 Item）可能需要重新绑定（以后会在数据库中更新）。有几种方式区分瞬时实例和分离实例，但是最好的方式是查看标识符属性值。Hibernate 能在重新绑定时检查瞬时对象或分离对象的标识符并能正确处理对象（和关联的对象图）。我们将在第 4.3.4 节“区分瞬时和持久实例”详细讨论这个重要内容。

如果你想要在你自己的应用程序中利用 Hibernate 对重新关联附加实例的支持，你需要在设计应用程序时知道 Hibernate 的标识范围—那就是保证标识实例的 Session 范围。只要离开了那个范围就会有分离实例，就会出现另一个有趣的概念。

我们需要讨论 Java 相等（看第三章第 3.4.1 节“标识和相等”）和数据库标识之间的关系。

相等是一个标识概念，你作为类开发者，可以（有时不得不）控制和使用具有分离实例的类。Java 相等通过在业务模型的持久类中实现 equals()和 hashCode()来定义。

4.1.6 实现 equals()和 hashCode()

equals()方法被应用程序代码调用，更重要的是被 Java 集合调用。例如，Set 集合，把每个对象放进此集合中都会调用 equals()方法来确定（阻止）重复的元素。

首先，让我们考虑由 java.lang.Object 定义的默认的 equals()实现，它是通过 Java 标识进行比较的。Hibernate 为 Session 中数据库的每一行分配一个唯一的实例。因此，如果你从不混合实例，默认的 equals()方法是正确的—那就是，你从不把不同 session 中的分离对象放进相同的 Set。（实际上，如果分离的对象来自相同的 session 但是在不同的区域序列化和反序列化，我们探索的这个话题仍然适用）。然而，只要你有来自多个 session 的实例，就有可能在一个 Set 中包含两个 Item，每个 Item 代表数据库表中相同的行但是没有相同的 Java 标识。这可能被认为是语法错误。然而，只要你按照原则处理不同 session 的分离对象（也要关注序列化和反序列化），建立标识（默认的）相等的复杂应用程序是可能的。这种方法的好处是不用写额外的代码来实现相等。

然而，如果这种相等的内容不是你想要的，你不得不在持久类中重写 equals()方法。记住，当你重写 equals()方法时也要重写 hashCode()方法以便两种方法保持一致（如果两个对象是相等的，他们必须有相同的 hashcode）。让我们看看在持久类中重写 equals()和 hashCode()方法的几种方式。

使用数据库标识符相等

比较聪明的方式是仅仅比较数据库标识符属性（通常是代理主键）值来实现 equals()方法：

```
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()方法对还没有分配数据库标识符值的瞬时实例（如果 id==null）是怎样转而求助 Java 标识符的。这是合理的，因为它们没有同另一个实例相同的持久标识。

不幸的是这种方案有个很大的问题：Hibernate 直到实体被保存了之后才分配标识符值。因此，如果对象在被保存之前加到 Set 中，当对象被包含到 Set 中时 hashcode 也改变了，跟 java.util.Set 的内容正好相反。特别地，这就使集合的级联保存（在本章的后面部分讨论）变得毫无用处。我们不赞同这种解决方案（数据库标识符相等）。

值比较

较好的方法是在 equals()比较中除了包含数据库标识符属性之外还要包含持久类的所有其它持久属性。这就是多数人怎样理解 equals()含义的，我们把它称为值相等。

我们所说的“所有属性”不包括集合。集合的状态与不同的表有关，所以包含它是错误的。更重要的是，你不想仅仅为了执行 equals()而强迫检索整个对象图。在 User 那个例子中，这意味着你在比较时不必包含 items 集合（此用户售出的项目）。因此，下面才是你应该使用的实现代码：

```
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，这种方法也有两个问题：

- 来自不同 session 的实例如果其中的一个修改了（例如，如果用户修改了密码），它们将不再相等。
- 具有不同数据库标识的实例（代表数据库表中不同行的实例）被认为是相等的，除非有一些组合属性被赋成唯一的（数据库列有唯一性约束）。在 User 例子中有一个唯一性属性：username。

为了开始我们推荐的解决方案，你需要理解业务键的概念。

使用业务键相等

业务键是一个属性，或是一些属性的组合，对于具有相同数据库标识的每个实例是唯一的。本质上，如果不使用代理键，选择业务键是很正常的。不像自然主键，对于业务键从不改变的实例来说它不是必须的—只要业务键改变，那就需要。

我们要求每个实体都应该有业务键，即使包含了类的所有属性（这对一些不可变类来说是正确的）。用户认为业务键唯一地标识某条记录，不管应用程序和数据库使用什么样的代理键。

业务键相等意味着 equals()方法仅仅比较形成业务键的属性。这就是避免前面提到的所有问题的最好解决方案。唯一不好的是首先需要额外定义正确的业务键。但是这些付出是必要

的，如果想让数据库通过约束检查来确保数据完整性，定义唯一性键非常重要。

对于 User 类，username 是非常好的候选业务键。username 从不为 null，唯一而且很少改变：

```
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();
}
}
```

其它类业务键可能更复杂，由复合属性组成。例如，Bid 类的候选业务键是项目 ID 与投标量的组合或项目 ID 同投标日期和时间的组合。BillingDetails 抽象类的好的业务键是 number 同帐单细目种类（子类）的组合。注意，在子类中重写 equals()并且在比较中包含另一个属性是不对的。在本例中，既满足对称性的又满足可传递性的相等需求是狡猾的。更重要的是业务键不能同任何数据库中定义好的候选自然键相符（子类属性可能映射到不同的表）。

你可能已经注意到 equals()和 hashCode()方法一直通过 getter 方法访问另一个对象的属性。这很重要，因为以 other 传递的对象实例可能是代理对象，而不是持有持久状态的真正实例。这一点是 Hibernate 不是完全透明的原因之一，但是使用访问器方法而不是访问直接的实例变量是最好的实践。

最后，注意当修改业务键属性的值时，当业务对象在集合中不要改变其值。

本节我们已经讨论过持久管理器。是仔细看看持久管理器及详细地探索 Hibernate Session API 的时候了。我们将在下一章重新详细讨论分离对象。

4.2 持久管理器

任何透明的持久管理器都含有持久管理器 API，通常提供下列服务：

- 基本的 CURD 操作；
- 执行查询；
- 控制事务；
- 事务级的缓存管理；

持久管理器可以暴露给几个不同的接口（就 Hibernate 来说，Session，Query，Criteria 和 Transaction）。这些接口的内部实现连接的很紧密。

应用程序与 Hibernate 之间的核心接口是 Session，它是刚刚列出的所有操作的起点。我们将在这本书余下的大部分章节交替谈到 persistence manager 和 session，这是同 Hibernate 社区中的用法一致的。

那么，怎样开始使用 session 呢？在工作单元的开头，一个线程含有从应用程序的 SessionFactory 返回的 Session 实例。应用程序如果访问多个数据源可能含有多个 SessionFactory。但是仅仅为了服务于某个请求你不要创建一个新的 SessionFactory—创建 SessionFactory 需要耗费大量的资源。另一方面，创建 Session 耗费的资源却很少。Session 甚至只有需要连接时才获得 JDBC 连接。

打开新的 Session 之后，就可以用它装载和保存对象了。

4.2.1 使对象持久化

想用 Session 做的第一件事就是把新创建的瞬时对象持久化。用 save()方法来做这件事情：

```
User user = new User();
user.getName().setFirstname("John");
user.getName().setLastname("Doe");
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
session.save(user);
tx.commit();
```

```
session.close();
```

首先，我们同往常一样初始化一个新的瞬时对象 user。当然，也可以在打开 session 之后初始化它，它们仍然是不相关的。我们使用 SessionFactory 打开一个新的 Session，然后开始一个新的数据库事务。

调用 save()方法使 User 的瞬时实例持久化。现在就跟当前 session 关联起来。然而，还没有执行 SQL 的 INSERT 语句。Hibernate Session 只有完全必要时才执行 SQL 语句。

对持久实例做的更改必须在某一时刻同数据库同步，这在 commit() Hibernate 事务时发生。在这个例子中，Hibernate 获得 JDBC 连接，然后执行 SQL INSERT 语句。最后，Session 关闭，释放 JDBC 连接。

注意在同 Session 关联之前最好（但不是必须的）完全初始化 User 实例。SQL 的 INSERT 语句含有对象调用 save()时持有的值。当然可以在调用完 save()之后修改对象，并且你所做的更改将会通过 SQL 的 UPDATE 语句同数据库同步。

在 session.beginTransaction()和 tx.commit()之间发生的任何事情都在一个数据库事务中。我们还没有详细地讨论过事务，我们将在下一章讨论。但是记住一个事务范围内的所有数据库操作或者完全成功或者完全失败。如果一条 INSERT 或 UPDATE 语句在 tx.commit()时失败，在这个事务中对持久对象所做的所有改变将会回滚到数据库级别。然而，Hibernate 不会回滚在内存中对持久对象所做的更改。这是合理的，因为数据库事务的失败通常是不可恢复的，你不得不立即丢弃失败的 Session。

4.2.2 更新分离实例的持久状态

在 session 关闭之后修改 user 将不会对数据库的持久表示层有影响。当 session 关闭时，user 变成一个分离实例。它可以在新的 session 中通过调用 update()或 lock()重新关联。

update()方法通过调度 SQL 的 UPDATE 语句强制更新数据库中对象的持久状态。下面有一个操作分离对象的例子：

```
user.setPassword("secret");
Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();
sessionTwo.update(user);
user.setUsername("jonny");
tx.commit();
sessionTwo.close();
```

对象是否在被传给 update()之前或之后修改过了都没有关系。重要的是调用 update()方法来重新将分离实例同新的 Session（当前事务）关联并且告诉 Hibernate 把此对象当作脏数据来对待（除非在持久类映射中设置了 select-before-update，这种情况下，Hibernate 就会通过执行一个 SELECT 语句把对象的当前状态同当前数据库的状态相比较来决定对象是否是脏的）。

调用 lock()方法将对象同 Session 相关联而不强制更新，就像下面这样：

```
Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();
sessionTwo.lock(user, LockMode.NONE);
user.setPassword("secret");
user.setLoginName("jonny");
tx.commit();
sessionTwo.close();
```

在这个例子中，是否在对象同 session 相关联之前或之后做了更改很重要。在调用 lock()之前做的更改不会同步到数据库中，只有你能够确信分离对象没有做过更改才能使用 lock()。

我们在下一章讨论 Hibernate 锁定模式。在这里我们通过指定 LockMode.NONE 告诉 Hibernate 当重新将对象同 Session 相关联时不执行版本检查或获得任何数据库级别锁。如果我们指定了 LockMode.READ 或 LockMode.UPDATE，Hibernate 将会执行 SELECT 语句来进行版本检查（并且设置更新锁）。

4.2.3 检索持久对象

Session 也用来查询数据库，检索存在的持久对象。Hibernate 在这个领域尤为强大，你将在本章的后面和第七章看到。然而，Session API 为最简单的查询提供了特定的方法：通过标

识符检索。这些方法其中之一是 get()，如下所示：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
tx.commit();
session.close();
```

检索出来的对象 user 现在可能被传递给表示层在事务之外作为分离对象（session 关闭之后）使用。如果数据库中没有给定标识符值对应的行，get()方法返回 null。

4.2.4 更新持久对象

任何通过 get()或其它种类的查询返回的持久对象已经同当前 Session 和事务上下文相关联了。可以修改对象并把修改后的状态同步到数据库中去。这种机制叫做自动脏数据检查，那意味着 Hibernate 将会跟踪和保存在一个 session 中对对象所做的更改。

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
user.setPassword("secret");
tx.commit();
session.close();
```

首先我们需要根据给定的标识符从数据库中检索对象。修改对象，然后当调用 tx.commit()时把这些修改同步到数据库中去。当然，一旦我们关闭了 Session，这个实例就被认为是分离的。

4.2.5 把持久对象转换为瞬时的

将持久对象转为瞬时对象很容易，使用 delete()方法将它的持久状态删除：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));
```

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

只有在事务结束 Session 同数据库同步时才执行 SQL 的 DELETE 语句。

Session 关闭以后，user 被认为是普通的瞬时实例。瞬时对象如果不再被其它对象引用将由垃圾回收器销毁。无论是内存中的对象实例还是持久的数据库行都将被移除。

4.2.6 把分离对象转换为瞬时的

最后，你可以将分离的实例转为瞬时的，从数据库中删除它的持久状态。这意味着不必重新绑定分离的实例就可以从数据库中删除它，可以直接删除分离的实例：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
session.delete(user);
tx.commit();
session.close();
```

在这个例子中，调用 delete()做两件事：将对象同 Session 关联然后为删除准备对象，在 tx.commit()时执行。

现在你已经知道持久生命期和持久管理器的基本操作。同第三章讨论的持久类映射一起，你可以创建自己的小型 Hibernate 应用程序了。（如果你喜欢，可以跳到第八章阅读 SessionFactory 和 Session 管理器的 Hibernate 帮助类。）记住我们没有给你任何异常处理代码，但是你应该能够自己解决 try/catch 块。映射一些简单的实体类和组件，然后在单独的应用程序中存储和装载对象（你不需要 web 容器或应用服务器，仅仅写一个 main 方法）。然而，只有你存储关联的实体对象—那就是，当你处理更复杂的对象图时—你就会知道在每个对象图中调用 save()或 delete()不是写应用程序的有效方式。

你最好尽可能少地调用 Session。传递的持久化（Transitive persistence）对强迫改变对象状态及控制持久化生命周期提供了更自然的方式。

4.3 在 Hibernate 中使用传递的持久化

实际上，重要的应用程序不是操纵单个对象而是操作对象图。当应用程序操纵持久对象图时，结果可能是由持久、分离和瞬时实例组成的对象图。传递的持久化是一种允许你把持久化自动传递给瞬时和分离子图的技术。

例如，如果我们为已经持久化的分层类添加一个新初始化的 Category，那么它不用调用 Session.save()就能自动持久化。在第三章我们映射 Bid 和 Item 之间的父子关系时给出了一个有点不同的例子。在那个例子中，不仅当 bid 加到 item 时会自动持久化，而且当拥有它们的 item 删除时也会自动地被删除。

传递的持久化模型不只一个。最有名的是可到达性的持久化（persistence by reachability），我们首先讨论它。虽然一些基本的原则是相同的，Hibernate 使用它自己的更强大的模型，以后你就会看到。

4.3.1 可到达性的持久化

如果当应用程序从另一个已经持久化的实例创建对实例的对象引用时所有实例都变成持久的，这样的对象持久层就称为实现了可到达性持久化。这种行为如图 4.2 的对象图（注意不是类图）所示：

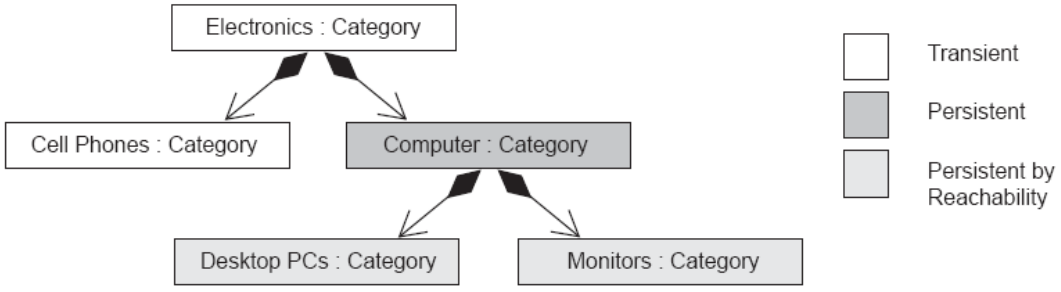

Figure 4.2 Persistence by reachability with a root persistent object

在这个例子中，“Computer”是持久的对象。对象“Desktop PCs”和“Monitors”也是持久的，它们从“Computer”Category 实例可以到达。“Electronic”和“Cell Phones”是瞬时的。注意我们假设导航仅仅对子目录是可以的，对父目录不可以—例如，我们可以调用

computer.getChildCategories()。可到达性的持久化是递归的算法：从一个持久实例开始所有可到达的对象在两种时候变成持久的，或者当原始对象持久化时或者当内存状态同数据存储同步前。

可到达性的持久化保证完整性约束，任何对象图能够通过装载持久的根对象完全重建。应用程序可以遍历对象图，从一个关联到另一个关联而不用担心实例的持久状态。（SQL 数据库使用不同的方法保证完整性约束，依靠外键和其它约束来检测越界（misbehaving）的应用程序。

在可到达性的持久化形式中，数据库有一些顶级的（或根的）对象，从这些对象可以到达所有的持久对象。实际上，如果一个实例不能通过根持久对象引用到达，这个实例就应该变成瞬时的并从数据库中删除。

Hibernate 和其它 ORM 方案都不能实现这种形式，在 SQL 数据库中没有类似于根持久对象的东西，也没有能够检测未引用实例的持久垃圾收集器。面向对象的数据存储实现的垃圾收集算法可能跟 JVM 的内存对象实现的算法相似，但是这种方式不适用于 ORM 世界，为了未引用的列而检索所有的表是不现实的。

因此，可到达性的持久化是最好的折中解决方案。它帮你把瞬时实例持久化并把他们的状态同步到数据库中而不用多次调用持久管理器。但是（至少在 SQL 上下文和 ORM 中）它不是解决把持久对象转为瞬时和从数据库中删除它们状态的问题的完全解决方案。这是更复杂的问题。当你删除某个对象时不能简单的删除所有可到达的对象，其它持久实例可能持有对它们的引用（记住实体是可以共享的）。你甚至不能安全地删除那些内存中不被任何持久对象引用的实例，内存中的实例仅仅是表示数据库的所有对象的一个小的子集。让我们看看 Hibernate 更复杂的传递持久化模型。

4.3.2 Hibernate 的级联持久化

Hibernate 的传递持久化模型使用跟可到达性持久化相同的基本内容—那就是检查对象关系来决定传递状态。然而，Hibernate 允许你为每个关系映射指定级联形式，为所有的状态转

换提供更复杂更精确的控制。Hibernate 读取声明的状态并自动级联操作到相关的对象。

当查找瞬时或分离对象时，Hibernate 默认不导航关联，因此保存、删除或重新绑定 Category 不会影响子目录对象。这跟可到达性的持久化的默认行为正好相反。如果对于某个关联想使用传递的持久化，你必须在映射元数据中重写这个默认的行为。

你可以在元数据中用下面的属性映射实体关系：

- cascade="none"，默认值，告诉 Hibernate 忽略关系。
- cascade="save-update"告诉 Hibernate 在下面这些情况导航关联：当事务提交时，当对象传给 save()或 update()方法并保存新初始化的瞬时实例及把更改持久到分离实例时。
- cascade="delete"告诉 Hibernate 当对象传给 delete()时导航关联并删除持久实例。
- cascade="all"意思是 save-update 和 delete 都级联，就像调用 evict 和 lock。
- cascade="all-delete-orphan"，跟 cascade="all"一样，但是除此之外，Hibernate 删除任何已经从关联（例如，从集合）删除（不再被引用）的持久实体实例。
- cascade="delete-orphan"，Hibernate 将会删除任何已经从关联（例如，从集合）删除（不再被引用）的持久实体实例。

这种关联级的级联形式模型比可到达性的持久化丰富但是缺少安全性。Hibernate 不能提供与可到达性持久化提供的相同的引用完整性。相反，Hibernate 部分代理与默认关系数据库的外键约束有关的引用完整性。当然，这种设计决定有一个好的理由：它允许 Hibernate 应用程序有效地使用分离对象，因为可以在关联级上控制分离对象图的重新绑定。

让我们用一些关联映射的例子详细阐述级联的内容。我们推荐你通读下一节，因为每个例子是建立在以前的例子之上的。我们的第一个例子很简单，有效地保存新增加的目录。

4.3.3 管理拍卖类别

系统管理员可以在目录树中创建新目录，重命名目录及删除目录。这种结构如图 4.3 所示。

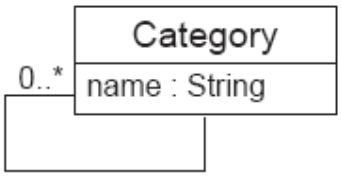

Figure 4.3
Category class with association to itself

现在，我们映射这个类和关系：

```
<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>
```

这是递归双向的一对多关系，就像第三章简单讨论的那样。“一”的这一端映射 <many-to-one> 元素和用 <set> 映射 Set 类型的属性。两者都引用相同的外键列：PARENT_CATEGORY_ID。

假设我们创建一个新的 Category 作为“Computer”的子目录（如图 4.4）。

我们有几种方式创建这个新的“Laptops”对象并把它保存到数据库中。我们回到数据库

中检索新的“Laptops”目录属于的“Computer”目录，增加新的目录并提交事务：

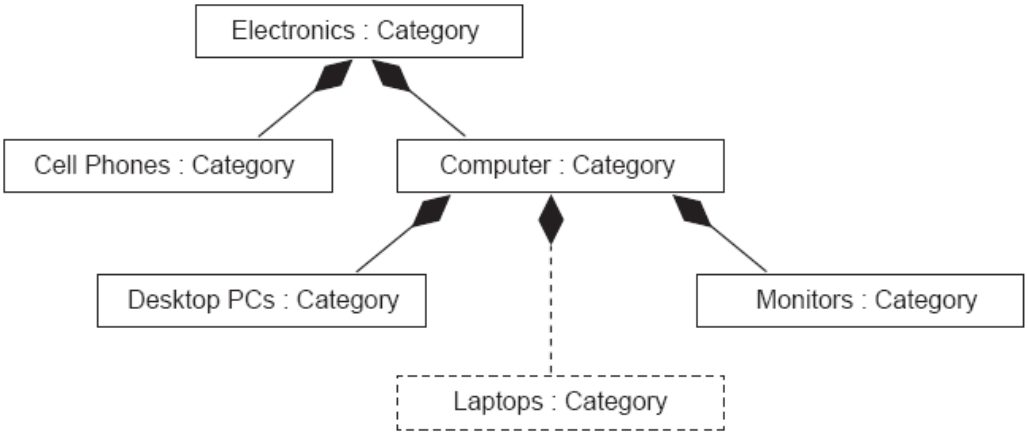

Figure 4.4 Adding a new `Category` to the object graph

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
Category computer = (Category) session.get(Category.class, computerId);
Category laptops = new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
tx.commit();
session.close();
```

computer 实例是持久的（绑定到 session），childCategory 关联已经打开了级联保存。因此，这段代码导致调用 tx.commit()时新的 laptops 目录成为持久的，因为 Hibernate 级联脏数据检查到 computer 的儿子。Hibernate 执行 INSERT 语句。

让我们再一次作相同的事，但是这次在任何事物之外（真正的应用程序中，在表示层操纵对象图很有用—例如，在把图传回持久层之前改变持久状态）创建“Computer”和“Laptops”之间的连接：

```
Category computer = ... // Loaded in a previous session
Category laptops = new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
```

现在分离的 computer 对象及它引用的其它分离对象与新的瞬时 laptops 对象相关联（反之亦然）。我们在第二个 Hibernate session 中保存新的对象把更改持久化到对象图中：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Persist one new category and the link to its parent category
session.save(laptops);
tx.commit();
session.close();
```

Hibernate 将会检查 laptops 父目录的数据库标识符属性，在数据库中正确创建与“Computer”的关系。Hibernate 把父亲的标识符值插入到 CATEGORY 中新的“Laptops”行的外键域。

由于 cascade=”none”是为 parentCategory 关联定义的，Hibernate 忽略层次树（“Computer”，“Electronics”）中任何其它目录的改变。调用 save()方法不会级联到被这种关联引用的实体。如果我们在映射 parentCategory 的<many-to-one>中设置 cascade=”save-update”，Hibernate 将会不得不导航内存中的整个对象图，将所有的实例同数据库同步。这个过程执行起来很糟糕，因为需要很多无用的数据访问。这种情况，我们对 parentCategory 既不需要也不想使用传递的持久化。

为什么有级联操纵？我们可以像前一个例子那样保存 laptop 对象，不使用任何级联映射。那么，考虑下面的例子：

```
Category computer = ... // Loaded in a previous Session
Category laptops = new Category("Laptops");
Category laptopAccessories = new Category("Laptop Accessories");
Category laptopTabletPCs = new Category("Tablet PCs")
laptops.addChildCategory(laptopAccessories);
laptops.addChildCategory(laptopTabletPCs);
computer.addChildCategory(laptops);
```

（注意，我们使用方便的 addChildCategory()方法在一端调用时设置关联连接的两端，就像第三章所描述的那样。）

不得不分别保存三个新的目录是我们所不希望的。幸运的是，因为我们把 childCategory 关联映射为 cascade=”save-update”，我们就不必那么做。我们用与以前相同的代码在新的 session 中保存单个“Laptops”目录就会保存所有三个新的目录了：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Persist all three new Category instances
session.save(laptops);
tx.commit();
session.close();
```

你可能想知道为什么级联形式叫做 cascade=”save-update”而不是 cascade=”save”。我们刚刚在前面把三个目录都持久化了，假设我们在接下来的请求中（session 和事务之外）对目录层次树作一些改变：

```
laptops.setName("Laptop Computers");
laptopAccessories.setName("Accessories & Parts");
laptopTabletPCs.setName("Tablet Computers");
Category laptopBags = new Category("Laptop Bags");
laptops.addChildCategory(laptopBags);
```

我们增加了一个新的目录作为“Laptops”目录的孩子并修改了三个已经存在的目录。下面的代码把这些改变同步到数据库中：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Update three old Category instances and insert the new one
session.update(laptops);
tx.commit();
session.close();
```

在 childCategories 关联中指定 cascade=”save-update”准确地反映了 Hibernate 能够决定需要什么把对象持久化到数据库中。这种情况，Hibernate 将会重新绑定或更新那三个分离的目录（laptops，laptopAccessories 和 laptopTabletPCs）并保存新的子目录（laptopBags）。

注意最后一个代码示例与前面两个在单独方法中调用的 session 示例不同。最后一个示例

使用 update()而不是 save()，因为 laptops 已经持久化了。

我们可以用 saveOrUpdate()方法重写所有的例子。然后那三个代码片断就会一样了：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
// Let Hibernate decide what's new and what's detached
session.saveOrUpdate(laptops);
tx.commit();
session.close();
```

saveOrUpdate()方法告诉 Hibernate 如果实例是新的瞬时实例就创建新的数据库行把实例的状态同步到数据库中，如果实例是分离的实例就更新已经存在的行。换句话说，它对 laptops 目录做的事情跟 cascade=”save-update”对 laptops 的子目录做的事情一样。

最后一个问题：Hibernate 怎样知道哪个孩子是分离的实例哪个是新的瞬时实例？

4.3.4 区分瞬时实例和分离实例

由于 Hibernate 不保留对分离实例的引用，你不得不让 Hibernate 知道怎样区分像 laptops（如果它在前一个 session 中创建）的分离实例和像 laptopBags 的新的瞬时实例。

可以利用一些选项。Hibernate 会把一个实例认为是未保存的瞬时实例，如果：

- 标识符属性（如果存在）为 null。
- 版本属性（如果存在）为 null。
- 在类的映射文档中支持 unsaved-value 及标识符属性匹配的值。
- 在类的映射文档中支持 unsaved-value 及版本属性匹配的值。
- 支持 Hibernate Interceptor 并在代码中检查完实例后从 Interceptor.isUnsaved()返回 Boolean.TRUE。

在我们的业务模型中，已经到处使用过可空的类型 java.lang.Long 作为标识符属性类型。由于我们正在使用生成的复合标识符，这就可以解决问题。新的实例有空的标识符属性值，因此 Hibernate 认为它们是瞬时的。分离的实例有非空的标识符值，Hibernate 也会正确的对待它们。

然而，如果在持久类中使用原始的 long 类型，就需要在所有的类中使用下面的标识符映射：

```
<class name="Category" table="CATEGORY">
<id name="id" unsaved-value="0">
<generator class="native"/>
</id>
....
</class>
```

unsaved-value 属性告诉 Hibernate 把具有标识符值为 0 的 Category 实例当作新初始化的瞬时实例。unsaved-value 属性的默认值为 null，因此，由于我们选择 Long 为标识符属性类型，可以在我们的拍卖应用程序类中忽略 unsaved-value 属性（我们到处使用相同的标识符类型）。

未保存的分配标识符	这种方式很适合复合标识符，但是不适合程序分配的键，包括遗留系统中的联合键。我们将在第八章第 8.3.1 节“遗留模式和联合键”讨论这个问题。在新的应用程序中尽量避免使用程序分配的键。

如果需要保存和删除对象，现在就能优化 Hibernate 应用程序和减少调用持久管理的次数了。检查所有类的 unsaved-value 属性，对分离对象做实验以便获得关于 Hibernate 传递的持久模型的感性知识。

现在我们看看另外一个重要的内容：怎样得到数据库之外的持久对象图（那就是怎样装载对象）。

4.4 检索对象

从数据库中检索对象是使用 Hibernate 最有趣（也是最复杂）的部分。Hibernate 提供下列方式从数据库中提取对象：

- 导航对象图，从一个已经装载的对象开始，通过像 aUser.getAddress().getCity()的属性访问器方法访问相关的对象。如果 Session 是打开的，当你导航图时，Hibernate 会自动装载图的节点。

- 当对象的唯一标识符值是已知的时候，通过标识符检索是最方便最有性能的方法。
- 使用 Hibernate 查询语言（HQL），它是完全面向对象的查询语言。
- 使用 Hibernate 条件 API，它提供了类型安全的面向对象的方式执行查询而不需要操纵字符串。这种便利性包括基于例子对象的查询。
- 使用本地 SQL 查询，这种查询 Hibernate 只关心把 JDBC 结果集映射到持久对象图。

在 Hibernate 应用程序中，将结合使用这几种技术。每一种检索方法可能使用不同的抓取策略—那就是定义持久对象图的哪个部分应该检索的策略。目标是在你的应用程序中为每个使用场合发现最好的检索方法和抓取策略，同时最小化查询语句的数量以获得最好的性能。

在这一节我们不仔细讨论每个检索方法，相反，我们将集中于基本的抓取策略和怎样调整 Hibernate 映射文件以便对所有的方法达到最好的默认抓取性能。在看抓取策略之前，我们将给出检索方法的概述。（我们提到 Hibernate 缓存系统但是将在下一章完全研究它。）

让我们开始最简单的例子，通过给定的标识符值检索对象（导航对象图不加以说明了）。在这一章的前半部分你已经看过一个简单的通过标识符检索的例子，但是还有许多需要知道。

4.4.1 根据标识符检索对象

下面的 Hibernate 代码片断从数据库中检索 User 对象：

```
User user = (User) session.get(User.class, userID);
```

get()方法很特别，因为标识符唯一地标识类的单个实例。因此，应用程序通常使用标识符方便地处理持久对象。当用标识符检索对象时可以使用缓存，如果对象已经缓存了可以避免数据库碰撞（hit）。

Hibernate 也提供了 load()方法：

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

load()方法是旧的，因为用户的请求已经把 get()方法加入到 Hibernate API。不同之处微不足道：

- 如果 load()方法不能在缓存或数据库中找到对象会抛出异常。load()方法从不返回

null。如果对象没找到 get()方法返回 null。

- load()方法返回代理而不是真正的持久实例。代理是一个占位符，当第一次调用它时才装载真正的对象。我们将在本节的后半部分讨论代理。另一方面，get()方法从不返回代理。

在 get()和 load()之间选择很简单：如果你能确定持久实例存在，不存在将会认为是异常，那么 load()是很好的选择。如果你不能确定给定的标识符是否有一个实例，使用 get()测试返回值，看它是否为 null。使用 load()有更深的含义：应用程序可能检索一个对持久实例的引用（代理）而不会强制数据库检索它的持久状态。因此，在缓存或数据库中不能找到持久对象时 load()不能抛出异常。异常会在以后抛出，当代理被访问的时候。

当然，使用标识符检索对象没有使用任意的查询复杂。

4.4.2 介绍 HQL

Hibernate 查询语言是与其相似的关系型查询语言 SQL 的面向对象方言。HQL 与 ODMG OQL 和 EJB-QL 很相像，但是不像 OQL，它是用于 SQL 数据库的，并且比 EJB-QL 更强大更优秀（然而，EJB-QL3.0 将会与 HQL 非常相似。）只要有 SQL 基础 HQL 非常容易学。

HQL 不是像 SQL 这样的数据操纵语言。它只能用来检索对象，不能更新、插入或删除数据。对象状态同步是持久管理器的工作，而不是开发者的工作。

大部分时间你仅仅需要检索特定类的对象，并且受那个类的属性的约束。例如，下面的查询根据姓来检索用户：

```
Query q = session.createQuery("from User u where u.firstname = :fname");
q.setString("fname", "Max");
List result = q.list();
```

准备查询 q 之后，我们把标识符值绑定到命名参数 fname 上。User 对象的 List 作为结果返回。

HQL 功能非常强大，虽然你不会一直使用其高级特征，但是你将会需要它们来解决一些复杂问题。例如，HQL 支持下面这些功能：

- 通过引用或持有集合（使用查询语言导航对象图）把限制条件应用到相关的关联对象的属性上的能力。
- 在事务范围仅仅检索一个或多个实体的属性而不是装载整个实体的能力。有时把它称为 *report query*，更正确的说法是 *projection*。
- 排列查询结果的能力。
- 分页查询的能力。
- 使用 group、having 及统计函数（如 sum、min 和 max）进行统计。
- 当在一行中检索多个对象时使用外联接。
- 调用用户自定义的 SQL 函数的能力。
- 子查询（嵌套查询）。

我们将在第七章把所有这些特性同可选的本地 SQL 查询机制放到一起讨论。

4.4.3 通过条件查询（Query by Criteria）

Hibernate 的通过条件查询（*query by criteria*(QBC)）API 允许你在运行时通过操纵查询对象来建立查询。这种方法允许动态的指定约束而不是直接操纵字符串，但是，它也丢掉了许多 HQL 的复杂性或强大功能。另一方面，以条件表示的查询比以 HQL 表示的查询可读性差。

通过名字检索用户使用查询对象更简单：

```
Criteria criteria = session.createCriteria(User.class);
criteria.add( Expression.like("firstname", "Max") );
List result = criteria.list();
```

Criteria 是一个 Criterion 实例树。Expression 类提供返回 Criterion 实例的静态工厂方法。一旦建立了想要的查询树，就会对数据库执行。

许多开发者喜欢 QBC，把它认为是更复杂的面向对象方法。他们也喜欢查询语法在编译时解释和验证的事实，而 HQL 只有在运行时才解释。

关于 Hibernate Criteria API 最好的方面是 Criterion 框架。这个框架允许用户对其进行扩展，像 HQL 这样的查询语言却很困难。

4.4.4 通过例子查询（Query by example）

作为 QBC 便利性的一部分，Hibernate 支持通过例子查询（QBE）。使用 QBE 的前提条件是应用程序支持具有某种属性值集合（非默认值）的查询类实例。查询返回所有的匹配属性值的持久实例。QBE 不是特别强大的方法，但是对一些应用程序却很方便。下面的代码片断演示 Hibernate 的 QBE：

```
User exampleUser = new User();
exampleUser.setFirstname("Max");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(exampleUser) );
List result = criteria.list();
```

QBE 的典型用例是允许用户指定属性值范围的查找屏幕，指定属性值范围用来匹配返回的结果集。这种功能在查询语言中很难清晰地表达，操纵字符串需要指定动态的约束集。

QBC API 和这种查询机制的例子将在第七章详细讨论。

现在你知道 Hibernate 中基本的检索选项。我们在本节的剩余部分关注对象图的抓取策略。抓取策略定义了用查询或装载操作检索对象图（或子图）的哪一部分。

4.4.5 抓取策略

传统的关系数据访问利用内连接和外连接检索相关的实体，用单个 SQL 查询抓取对某个计算所需要的所有数据。一些原始的 ORM 实现分开抓取数据，多次请求小块的数据，应用程序作为响应也会多次导航持久对象图。这种方法不能有效利用关系数据库的连接能力。实际上，这种数据访问策略将来很难扩展。ORM 中的一个最困难的问题—可能是最困难的—是提供对关系数据库的有效访问，鉴于应用程序喜欢把数据当成对象图看待。

对于我们经常开发的多种应用程序（多用户，分布式，web 和企业应用），检索对象时多次往返于数据库是不可取的。因此，我们讨论的工具比传统的工具更强调 ORM 中的 R（关系）。

有效地抓取对象图的问题已经通过在关联映射的元数据中指定关联级抓取策略解决了。这种方法存在的问题是每段代码使用一个需要不同集合的相关对象的实体。但是这是不够的。

我们需要的是支持细粒度的运行时关联抓取策略。Hibernate 两者都支持，允许在映射文件中指定默认的抓取策略，然后在代码运行时重载。

Hibernate 对于任何关联允许在四种抓取策略中选择，在关联元数据和运行时：

- 立即抓取—立即抓取关联的对象，使用连续的数据库读（或缓存查找）。
- 延迟抓取—当第一次访问时，“延迟”抓取相关的对象或集合。这个结果在对数据库的新请求中（除非缓存了相关的对象）。
- 提前抓取—相关的对象或集合同拥有它们的对象一起抓取，使用 SQL 外连接，不需要额外的数据库请求。
- 批量抓取—在访问延迟关联时，这种方法通过检索一批对象或集合来提高延迟抓取的性能。（批量抓取也用来提高立即抓取的性能。）

让我们仔细看看每种抓取策略。

立即抓取

立即的关联抓取发生在从数据库中检索实体然后立即在下一个对数据库或缓存的请求中检索另一个（或一些）相关的实体的时候。立即抓取通常不是有效的抓取策略除非希望关联的实体一直被缓存。

延迟抓取

当客户请求数据库中的实体及其相关的对象图时，通常不必检索每个（非直接的）关联对象的整个对象图。你不希望立即把整个数据库装载到内存中，例如，装载单个 Category 不应该触发装载这个目录的所有 Item。

延迟抓取能够让你决定第一次访问数据库时装载多少对象图，并且与其关联的对象只有在第一次访问时才装载。延迟抓取是对象持久化中的基本内容，而且是取得可接受性能的第一步。

我们推荐在开始的时候把映射文档中所有的关联映射为延迟（或可能是批量延迟）抓取。这种策略然后被强制提前抓取发生的查询重载。

提前（外连接）抓取

延迟关联抓取能够帮助减少数据库装载，而且通常是一种好的默认策略。然而，这在性能优化发生前有点盲目猜测。

提前抓取让你显式地指定哪些关联的对象应该同引用它们的对象一起装载。Hibernate 然后在单个数据库请求中使用 SQL 的 OUTER JOIN 返回关联的对象。Hibernate 的性能优化通常包括针对某些事务明智地使用提前抓取。因此，即使在映射文件中声明了默认的抓取策略，在运行时对于某个 HQL 或条件查询指定使用这种策略更普遍。

批量抓取

批量抓取不是严格的关联抓取策略，它是帮助提高延迟（或立即）抓取性能的一种技术。通常，当装载对象或集合的时候，SQL 的 WHERE 子句指定对象的标识符或拥有集合的对象。如果开启了批量抓取，Hibernate 看起来知道什么会在当前 session 中引用其它代理实例或未初始化的集合，尽量通过在 WHERE 子句中指定多个标识符值来同时装载这些对象。

我们不是这种方法的热心者，提前抓取几乎一直是更快的。批量抓取对那些希望用 Hibernate 达到可接受的性能而不用想太多关于要执行的 SQL 的经验不足的用户很有用。（注意，你可能很熟悉批量抓取，因为它已经被许多 EJB2 引擎使用）。

现在我们在映射元数据中对一些关联声明抓取策略。

4.4.6 在映射中选择一种抓取策略

Hibernate 允许你在映射元数据中通过指定属性选择默认的关联抓取策略。可以使用 Hibernate 查询方法的特性重载默认的策略，你将会在第七章看到。一点小警告：不比立即明白本节出现的每个参数，我们推荐你先浏览一下，当你在应用程序中优化默认的抓取策略时把本节当作参考。Hibernate 映射形式的一个缺点就是集合映射函数有点同单点关联不同，因此，我们将要分别覆盖两种情况。让我们首先考虑 Bid 和 Item 之间双向关联的两端。

单点关联

对于<many-to-one>或<one-to-one>关联，如果关联的类映射开启代理延迟抓取是可能的。

对于 Item 类，我们通过指定 lazy=”true”开启代理：

```
<class name="Item" lazy="true">
```

现在记得从 Bid 关联到 Item：

```
<many-to-one name="item" class="Item">
```

当我们从数据库中检索 Bid 时，关联属性可能持有 Hibernate 生成的 Item 子类的实例，这个实例代理所有对从数据库中延迟抓取的不同 Item 实例的方法调用（这是 Hibernate 代理更详细的定义）。

Hibernate 使用两种不同的实例以便能够代理更多态的关联—当代理的对象被抓取时，它可能是 Item 映射子类的一个实例（那就是，如果 Item 有子类的话）。我们甚至可以选择被 Item 类实现的任何接口作为代理的类型。要这么做，使用 proxy 属性声明它，而不是指定 lazy=”true”：

```
<class name="Item" proxy="ItemInterface">
```

只要我们在 Item 中声明了 proxy 或 lazy 属性，任何对 Item 的单点关联都被代理和延迟抓取，除非关联通过声明 outer-join 属性重载了抓取策略。

对于 outer-join 有三种可能值：

- outer-join=”auto”—默认值。当没有指定这个属性时，如果开启了代理 Hibernate 延迟抓取关联的对象，或者如果禁止了代理（默认值）提前使用外连接。
- outer-join=”true”—Hibernate 一直使用外连接提前抓取关联，即使开启了代理。这允许你对相同的代理类的不同关联选择不同的抓取策略。
- outer-join=”false”—Hibernate 从不用外连接抓取关联，即使开启了代理。这对于希望相关的对象存在于第二级缓存中很有用（见第五章）。如果在第二级缓存中不可用，使用额外的 SQL SELECT 立即抓取对象。

因此，如果我们希望再一次开启关联的提前抓取，既然开启了代理，我们就要指定

```
<many-to-one name="item" class="Item" outer-join="true">
```

对于一对一关联（在第六章详细讨论），延迟抓取仅仅当关联的对象已经存在时是理论上

可行的。我们通过指定 constrained=”true”来标识。例如，如果 item 仅仅有一个 bid，Bid 的映射应该为：

```
<one-to-one name="item" class="Item" constrained="true">
```

constrained 属性跟<many-to-one>映射的 not-null 属性有点相似。它告诉 Hibernate 关联的对象是必需的，不能为 null。

为了开启批量抓取，在 Item 的映射中指定 batch-size：

```
<class name="Item" lazy="true" batch-size="9">
```

批量大小限制在单个批量中可能检索的 item 数。这里选择一个合理的小的整数。

当我们考虑集合时可能会遇到相同的属性（outer-join，batch-size 和 lazy），但是解释有点不同。

集合

对于集合来说，抓起策略不仅支持实体关联，而且也支持集合值（例如，字符串的集合可以被外连接抓取）。

就像类一样，集合有它们自己的代理，通常叫做*集合包装*。不像类，集合包装一直是那样，即使延迟抓取关闭了（Hibernate 需要包装来检测集合变化）。

集合映射可以既不声明 lazy 属性又不声明 outer-join 属性或两者都声明（两者都指定没有什么意义）。有意义的参数如下：

- 两个属性都不指定—这个参数跟 outer-join=”false” lazy=”false”等价。集合从第二级缓存中抓取或通过立即的额外的 SQL SELECT 抓取。这个参数是默认的并且当为这个集合开启第二级缓存时最有用处。
- outer-join=”true”—Hibernate 使用外连接提前抓取关联。在写这本书的时候，Hibernate 对于每个 SQL SELECT 仅仅能抓取一个集合，因此用 outer-join=”true”不可能声明属于相同持久类的多个集合。
- lazy=”true”—Hibernate 当第一次访问集合时延迟抓取集合。

我们不推荐对集合使用提前抓取，因此用 lazy=”true”映射 bid 的 item 集合。这个参数几乎一直用于集合映射（应该是默认值，我们推荐你在所有的集合映射中把它当作默认值）：

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

我们甚至能够为集合开启批量抓取。这种情况下，批量大小不再指批量抓取中的 bid 数，而是指 bid 的集合数：

```
<set name="bids" lazy="true" batch-size="9">
<key column="ITEM_ID"/>
<one-to-many class="Bid"/>
</set>
```

这个映射告诉 Hibernate 在一次批量抓取中装载九个 bid 集合，取决于多少未初始化的 bid 集合当前出现在与 session 关联的 item 里。换句话说，如果在一个 Session 中有五个具有持久状态的 Item 实例，并且所有的都有未初始化的 bid 集合，如果访问了其中的一个，Hibernate 将在单个 SQL 查询中自动装载所有的五个集合。如果有十一个 item，仅仅会抓取九个。批量抓取会显著地减少请求层次对象的查询数（例如，当装载父子 Category 对象树时）。

让我们讨论一种特殊情况：多对多关联（我们在第六章详细讨论这种映射）。通常使用连接表（一些开发者也把它叫做关系表或关联表）保存两个关联表的键值并且因此允许多对多的多样性。如果你决定使用提前抓取就不得不考虑这个附加表。看看下面这个简单的多对多示例，映射从 Category 到 Item 的关联：

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

在这个例子中，提前抓取策略仅仅引用关联的表 CATEGORY_ITEM。如果我们用这种抓取策略装载 Category，Hibernate 将会用一个外连接 SQL 查询自动抓取所有来自

CATEGORY_ITEM 的连接入口，而不是来自 ITEM 的 item 实例！

多对多关联中包含的实体当然也能用相同的 SQL 查询提前抓取。<many-to-many>元素允许自定义这种行为：

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

现在当装载 Category 时 Hibernate 用单个外连接查询抓取所有 Category 中的 Item。然而，记住我们通常推荐把延迟加载作为默认的抓取策略并且 Hibernate 限制每个映射持久类只能有一个提前抓取的集合。

设置抓取深度

现在我们讨论全局抓取策略设置：最大抓取深度。这个设置控制 Hibernate 在单个 SQL 查询中使用的外连接表数。考虑从 Category 到 Item 和从 Item 到 Bid 的整个关联链。第一个是多对多关联，第二个是一对多关联，因此两种关联都要映射成集合元素。如果我们为两种关联都声明 outer-join=”true”（不要忘记指定<many-to-many>声明）并装载单个 Category，Hibernate 将会执行多少查询？仅仅提前抓取 Item，或者也提前抓取所有 Bid 中的每个 Item？

你可能希望使用包含 CATEGORY，CATEGORY_ITEM，ITEM 和 BID 表的外连接操作的单个查询。然而，这不是默认的情况。

Hibernate 的外连接抓取行为受全局配置参数 hibernate.max_fetch_depth 的控制。如果把它设为 1（也是默认值），Hibernate 就只抓取 Category 和来自 CATEGORY_ITEM 关联表的连接入口。如果把它设为 2，Hibernate 用相同的 SQL 查询执行也包含 Item 的外连接。把这个参数设为 3，在一个 SQL 语句中连接所有的四个表并且装载所有的 Bid。

抓取深度的推荐值依赖于连接性能和数据库表的大小，先用低的值（小于 4）测试应用程序，然后当调整你的应用程序时增加或减少这个值。全局最大抓取深度也可以用于使用提前抓取策略映射的单端关联（many-to-one,<one-to-one>）。

记住在映射元数据中声明的提前抓取策略只有当使用标识符检索、使用条件查询 API 或手动导航对象图时才有效。任何 HQL 查询可以在运行时指定自己的查询策略，这样忽略映射的默认值。

初始化延迟关联

4.4.7 调整检索的对象

让我们看看在应用程序中调整检索的对象需要的几步操作：

1. 像第二章描述的那样开启 Hibernate SQL 日志功能。你也应该准备阅读、理解和评估 SQL 查询及对于特定的关系模型的执行性能：单个的连接操作会比两个选择快么？在所有的索引正确使用的情况下，数据库的缓存命中率将会怎样？让 DBA 帮你评估性能，她只具有能够决定执行哪个 SQL 是最好的知识。
2. 单步跟踪你的应用用例，注意 Hibernate 执行了多少条 SQL 语句及执行了什么样的 SQL 语句。用例可能是 web 应用中的一个页面也可能是一系列用户对话框。这步也包括收集用例中使用的检索对象的方法：遍历图形，通过标识符、HQL 及条件查询检索。你的目标是通过调整元数据中的默认抓取策略降低 SQL 查询数及复杂度。
3. 你可能会遇到下面两个常见的问题：
   - 如果使用连接操作的 SQL 语句太复杂太慢，把<many-to-many>关系的 outer-join 设为 false（默认情况是开启的）。尽量调整全局 hibernate.max_fetch_depth 配置选项，记住最好把值设在 1 和 4 之间。
   - 如果执行太多的 SQL 语句，对所有的集合映射使用 lazy=”true”。Hibernate 对集合元素默认使用立即的附加抓取（那就是，如果它们是实体，可以级联到图形中）。很少情况下，如果你能确信，设置 outer-join=”true”并关闭对特定集合的延迟加载。记住每个持久类只有一个集合属性可能被明确抓取。如果给定的工作单元包括几个“相同的”集合或者访问父子对象树，使用值在 3 和 10 之间的批量抓取来进一步优化集合抓取。

4. 设置完新的抓取策略，返回用例，再一次检查生成的 SQL。注意 SQL 语句，走到下一个用例。
5. 优化完所有的用例，再检查一遍看是否某个优化对其它优化有影响。对于同样的经历，你就能够避免负面影响并及时纠正。

这种优化技术不仅对默认的抓取策略是可行的，也可以用它来调整 HQL 和条件查询，对特定的用例和工作单元忽略及覆盖默认的抓取。我们将在第七章讨论运行时抓取。

本节，我们已经开始考虑性能问题，尤其是与关联抓取相关的问题。当然，抓取对象图最好的方式是从内存缓存中抓取，如下一章所示的那样。

4.5 总结

对象/关系不匹配的动态问题跟更加熟悉和理解结构化不匹配问题一样重要。本章我们主要关心与持久机制相关的对象生命周期。现在你理解了三种 Hibernate 定义的对象状态：持久的，分离的和瞬时的。当你调用 Session 接口的方法或创建和删除对已经持久化的实例的引用时，对象在不同状态之间转换。后面的行为由可配置的级联形式控制，Hibernate 的传递持久化模型。这个模型允许声明以关联为基础的级联操作（像保存或删除），比传统的可到达性持久化模型更强大更灵活。你的目标是为每个关联找到最好的级联形式，因而最小化存储对象时不得不调用持久管理器的次数。

从数据库中检索对象是同样重要的：可以通过访问属性来遍历业务模型图，让 Hibernate 显式地抓取对象。也可以通过标识符装载对象，用 HQL 写任意的查询或者使用条件查询 API 创建面向对象的表示层。除此之外，对于特别的情况可以使用本地 SQL 查询。

大多数哲学对象检索方法使用映射远射击中定义的默认抓取策略（HQL 忽略它们，条件查询能够重载它们）。正确的抓取策略最小化通过延迟、提前或批量抓取对象不得不执行的 SQL 语句数量。你可以通过分析每个用例中执行的 SQL，调整默认的和运行时抓取策略来优化 Hibernate 应用程序。

下面我们讨论与事务和缓存相关的话题。