第 10 章

事务和并发

本章内容

- 数据库事务
- 使用Hibernate和Java Persistence的事务
- 非事务的数据访问

本章终于讨论事务以及如何在应用程序中创建和控制工作单元了。我们将介绍事务如何在最低级别（数据库）工作，以及在基于原生Hibernate、Java Persistence和用或者不用Enterprise JavaBeans的应用程序中如何处理事务。

事务允许你设置工作单元的范围：一个原子的操作组。它们还帮助你在多用户的应用程序中把一个工作单元从另一个中隔离出来。我们讨论并发，以及如何在应用程序中利用悲观和乐观的策略控制并发的数据访问。

最后，看一下非事务的数据访问，以及什么时候应该在自动提交模式下使用数据库。

10.1 事务本质

从一些背景信息开始。应用程序功能要求同时完成几件事情。例如，当一次拍卖结束时，CaveatEmptor应用程序必须执行3项不同的任务：

(1) 给胜出（金额最高）的出价做标记。

(2) 向卖主收取拍卖费用。

(3) 通知卖主和成功的出价人。

如果你由于外部信用卡系统的失败而无法收取拍卖费用，会发生什么事？业务需求可能规定，列出的所有动作必须要么都成功，要么都不成功。如果这样，你可以把这些步骤全部称为事务（transaction）或者工作单元。哪怕只有一个步骤失败，则整个工作单元都必定失败。这就是大家所知的原子性（atomicity），即所有操作都作为一个原子单元来执行。

此外，事务允许多个用户同时使用相同的数据，而不破坏数据的完整性和正确性；特定的事务不应该对其他同时运行的事务可见。为了完全理解这个隔离性（isolation）行为，有几个策略很重要，我们将在本章中给予探讨。

事务具有其他重要的属性，例如一致性（consistency）和持久性（durability）。一致性意味着事务始终工作在同一组数据上：从其他同时运行的事务中隐藏起来的、并在事务完成之后留在一个清洁和一致的状态中的一组数据。数据库完整性规则保证一致性。你也想要事务的正确性（correctness）。例如，业务规则规定向卖主收取一次费用，而不是两次。这是个合理的假设，但是你可能无法用数据库约束把它表达出来。因而，事务的正确性是应用程序的责任，而一致性则是数据库的责任。持久性意味着一旦事务完成，所有变化都在该事务变成持久化期间进行，即使系统后来失败了，这些变化也不会丢失。

把这些事务属性归结起来，就是大家所知的ACID标准。

数据库事务必须简短。单个的事务通常只涉及单批数据库操作。在实践中，还需要一个概念，允许你有长期运行的对话，在这里一个原子组合的数据库操作不是在一批而是几批中发生。对话允许应用程序的用户有思考时间，而仍然保证原子、隔离且一致的行为。

既然已经定义了术语，就可以讨论事务划分（demarcation）以及如何定义一个工作单元的范围了。

10.1.1 数据库和系统事务

数据库把工作单元的概念实现为一个数据库事务（database transaction）。数据库事务组合了数据库访问操作——也就是SQL操作。所有SQL语句都在一个事务内部执行；无法把SQL语句发送到数据库事务之外的数据库。事务被确保以这两种方式之一终止：要么完全被提交（commit），要么完全被回滚（roll back）。因而，我们说数据库事务是原子的。在图10-1中，你可以看到这个图示。

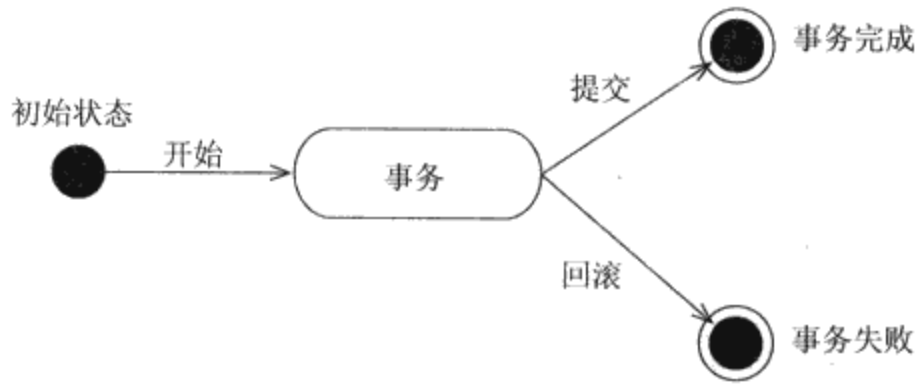

图10-1 一个原子工作单元（事务）的生命周期

为了在事务内部执行所有的数据库操作，必须给这个工作单元的范围做标记。必须启动事务，并在某个时间点提交变化。如果出现错误（在执行操作或者提交事务的时候），就必须回滚事务，把数据留在一致的状态中。这就是大家所知的事务划分，取决于你所使用的技术，它涉及更多或者更少的手工干涉。

一般来说，启动和终止事务的事务范围可以在应用程序代码中编程式地设置，或者声明式地设置。

1. 编程式的事务划分

在非托管环境中，JDBC API被用来给事务范围做标记。通过在JDBC `Connection`中调用

setAutoCommit(false)启动事务，并通过调用commit()终止它。可以在任何时候通过调用rollback()强制立即回滚。

在一个于几个数据库中操作数据的系统中，特定的工作单元涉及对不止一个资源的访问。既然如此，你就无法单独通过JDBC实现原子性。你需要可以在系统事务（system transaction）中处理几个资源的事务管理器（transaction manager）。这样的事务处理系统公开了与开发人员进行交互的Java Transaction API（JTA）。JTA中的主API是UserTransaction接口，包含begin()和commit()系统事务的方法。

此外，Hibernate应用程序中的编程式事务管理通过Hibernate Transaction接口公开给应用程序开发人员。你并没有被强制使用这个API——Hibernate也让你直接启动和终止JDBC事务，但不鼓励这种用法，因为它把代码绑定到了直接的JDBC。在Java EE环境中（或者如果你把它和Java SE应用程序一起安装了），就可以使用JTA兼容的事务管理器，因此你应该调用JTA UserTransaction接口来编程式地启动和终止事务。然而，就像你可能已经猜到的，Hibernate Transaction接口在JTA的顶层也有效。我们将介绍所有这些方法，并更详细地讨论可移植性的关注点。

利用Java Persistence的编程式事务划分，还必须在Java EE应用程序服务器的内部和外部使用。在应用程序服务器外部，利用简单的Java SE，处理本地资源事务；这是EntityTransaction接口的作用——你已经在前面几章中见过它。在应用程序服务器内部，调用JTA UserTransaction接口来启动和终止事务。

让我们概括一下这些接口，以及什么时候使用它们：

- java.sql.Connection——利用setAutoCommit(false)、commit()和rollback()进行简单的JDBC事务划分。它可以但不应该被用在Hibernate应用程序中，因为它把应用程序绑定到了一个简单的JDBC环境。
- org.hibernate.Transaction——Hibernate应用程序中统一的事务划分。它适用于非托管的简单JDBC环境，也适用于以JTA作为底层系统事务服务的应用程序服务器。但是，它最主要的好处在于与持久化上下文管理的紧密整合——例如，你提交时Session被自动清除。持久化上下文也可以拥有这个事务的范围（对会话有用，请见第11章）。如果你无法具备JTA兼容的事务服务，就使用Java SE中的这个API。
- javax.transaction.UserTransaction——Java中编程式事务控制的标准接口，它是JTA的一部分。每当你具备JTA兼容的事务服务，并想编程式地控制事务时，它就应该成为你的首选。
- javax.persistence.EntityTransaction——在使用Java Persistence的Java SE应用程序中，编程式事务控制的标准接口。

另一方面，声明式事务划分不需要额外的代码；并且从定义上来说，它解决了可移植性的问题。

2. 声明式事务划分

在应用程序中，当你希望在一个事务内部进行工作的时候要进行声明（例如，在方法中使用注解）。然后处理这个关注点就是应用部署程序和运行时环境的责任了。Java中提供声明式事务

服务的标准容器是EJB容器，这项服务也称作容器托管事务。我们将再次编写EJB会话bean，介绍Hibernate和Java Persistence如何从这项服务中受益。

在你决定一种特定的API之前，或者为了声明式事务划分，让我们一步步来探讨这些选项。首先，假设你正要在一个简单的Java SE应用程序（一个客户端/服务器端Web应用程序、桌面应用程序或者任何两层系统）中使用原生的Hibernate。之后，要把代码重构到一个托管的Java EE环境下运行（并且首先看看如何避免这种重构）。我们也顺便讨论Java Persistence。

10.1.2 Hibernate 应用程序中的事务

想象你正在编写一个必须在简单的Java中运行的Hibernate应用程序；没有容器、没有托管的数据库资源可以使用。

1. Java SE中的编程式事务

配置Hibernate为你创建一个JDBC连接池，就像在2.1.3节中所做的那样。如果你正在利用Transaction API编写Java SE Hibernate应用程序，除了连接池之外，不需要其他的配置设置：

- hibernate.transaction.factory_class选项默认为org.hibernate.transaction.JDBCTransactionFactory，这是Java SE中Transaction API以及直接的JDBC的正确工厂。
- 可以用自己的TransactionFactory实现扩展和定制Transaction接口。这几乎没有必要，但是有一些值得关注的使用案例。例如，每当启动事务时，如果你必须编写审计日志，就可以把这个日志添加到一个定制的Transaction实现。

Hibernate为你正要使用的每个Session获得一个JDBC连接：

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();

    concludeAuction(session);

    tx.commit();
} catch (RuntimeException ex) {
    tx.rollback();
} finally {
    session.close();
}
```

Hibernate Session是延迟的。这是件好事——意味着它不消费任何资源，除非绝对需要。只有当数据库事务启动时，才从连接池中获得JDBC Connection。对beginTransaction()的调用在新的JDBC Connection中转变成setAutoCommit(false)。现在Session被绑定到了这个数据库连接，并且所有的SQL语句（在这个例子中，终止拍卖所需要的所有SQL）都在该连接上发送。所有的数据库语句都在同一个数据库事务内部执行。（假设concludeAuction()方法调用指定的Session来访问数据库。）

我们已经谈到过迟写行为，因此你知道当Session的持久化上下文被清除时，大批量的SQL语句被尽可能迟地执行。默认情况下，这发生在当你在Transaction上调用commit()的时候。提交事务（或者把它回滚）之后，数据库连接被释放，并且从Session中解除绑定。用相同的Session启动一个新的事务，从连接池中获取另一个连接。

关闭Session释放出所有其他的资源（例如，持久化上下文）；所有托管的持久化实例现在都被认为是脱管的。

常见问题　回滚只读事务更快吗？如果事务中的代码读取数据但没有修改它，你应该回滚事务而不是提交它吗？这样更快吗？很显然，有些开发人员发现这在某些特殊环境下会更快，并且这种信念已经传播到了整个社区。我们用更为普及的数据库系统进行过测试，发现并没有区别。我们也没有发现显示性能差别的真实数据的任何来源。对于数据库系统为什么应该有一个并非最佳的实现——为什么它不应该内部使用最快的事务清除算法，也没有很好的解释。始终提交事务，如果提交失败就回滚。话虽这么说，但SQL标准还是包括了一个SET TRANSACTION READ ONLY（设置事务只读）的语句。Hibernate不支持启用这个设置的API，虽然你可以实现自己定制的Transaction和TransactionFactory来添加这个操作。我们建议你先查一下数据库是否支持这一点，以及可能的性能受益有什么，如果有，就可以了。

我们现在要讨论异常处理。

2. 处理异常

如果上一个例子中介绍过的concludeAuction()（或者提交期间持久化上下文的清除）抛出异常，你就必须调用tx.rollback()强制事务回滚。它会立即回滚事务，因此你发送到数据库的SQL操作对性能没有任何影响。

这似乎很简单，虽然或许你可能已经看到，每当你想要访问数据库的时候，捕捉RuntimeException都不会产生好代码。

说明　异常的历史——异常以及异常应该如何处理，始终是Java开发人员之间争论的热点。Hibernate也有一些值得瞩目的历史，这并不奇怪。在Hibernate 3.x之前，Hibernate抛出的所有异常均为checked exception（已检查异常），因此每个Hibernate API都强制开发人员捕捉和处理异常。这个策略受到JDBC影响，它也只抛出checked exception。但是很快就清楚了，这样没有意义，因为Hibernate抛出的所有异常都是致命的。在许多情况下，开发人员在这种情形之下所能做的就是清除，显示错误消息，并退出应用程序。因此，从Hibernate 3.x开始，Hibernate抛出的所有异常都是unchecked（未检查）的RuntimeException的子类型，它通常在应用程序中单个的位置进行处理。这也使得所有Hibernate模板或者包装API都变得没用了。

首先，即使我们承认你不用许多（或者成百个）try/catch块编写应用程序代码，我们介绍过的例子也是不完整的。这是一个Hibernate工作单元标准惯用语的例子，带有包含真实的异常处

理的数据库事务：

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();

    tx.setTimeout(5);

    concludeAuction(session);

    tx.commit();
} catch (RuntimeException ex) {
    try {
        tx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    session.close();
}
```

任何Hibernate操作，包括清除持久化上下文，都可能抛出RuntimeException。甚至回滚事务也可能抛出异常！你想要捕捉这个异常并记录到日志中；否则，导致回滚的原始异常就被淹没了。

这个例子中可选的方法调用是setTimeout()，它获得允许事务运行的秒数。然而，在Java SE环境中并没有真正被监测的事务。如果在应用程序服务器外部运行这段代码（也就是说，没有事务管理器），Hibernate所能做的最好的就是给驱动器等待PreparedStatement执行的时间设置秒数（Hibernate专用的预编译语句）。如果超出限制，就抛出SQLException。

你不想用这个例子作为自己应用程序中的一个模板，因为你应该通过一般的基础结构代码来隐藏异常处理。例如，可以给RuntimeException编写单个错误处理程序，知道什么时候以及如何回滚事务。对于开启和关闭Session也一样。稍后第11章会用更真实的例子讨论它，16.1.3节也会再次讨论它。

Hibernate抛出类型（typed）异常，即帮助你辨别错误的RuntimeException的所有子类型：

- 最常见的HibernateException是个一般的错误。你必须检查异常消息，或者在通过异常中调用getCause()找出更多原因。
- JDBCException是被Hibernate的内部JDBC层抛出的任何异常。这种异常总是由一个特定的SQL语句产生，可以用getSQL()获得这个引起麻烦的语句。JDBC连接（实际上是JDBC驱动器）抛出的内部异常可以通过getSQLException()或者getCause()获得，并且通过getErrorCode()可以得到特定于数据库和特定于供应商的错误代码。
- Hibernate包括JDBCException的子类型和一个内部转换器，该转换器试图把数据库驱动抛出器的特定于供应商的错误代码变成一些更有意义的东西。内建的转换器可以给Hibernate支持的最重要的数据库方言生成JDBCConnectionException、SQLGrammarException、LockAquisitionException、DataException和ConstraintViolationException。

可以对数据库操作或者增强方言，或者插入SQLExceptionConverterFactory定制这种变换。

- Hibernate抛出的其他RuntimeException也应该终止事务。应该始终确保捕捉RuntimeException，无论你计划利用任何细粒度的异常处理策略去做什么。

你现在知道了应该捕捉什么异常，以及它们什么时候会出现。但是你心里可能有一个问题：捕捉到异常之后应该怎么办？

Hibernate抛出的所有异常都是致命的。这意味着你必须回滚数据库事务，并关闭当前的Session。不允许你继续使用抛出异常的Session。

通常，你还必须在关闭异常后面的Session之后退出应用程序，虽然有一些异常（例如StaleObjectStateException）一般导致在新Session中一个新的尝试（可能在再次与应用程序的用户交互之后）。由于这些与会话以及并发控制密切相关，我们将在稍后讨论它们。

常见问题　可以把异常用于验证吗？有些开发人员一旦知道Hibernate可以抛出许多细粒度的异常类型后就很兴奋。这样可能把你引向歧途。例如，你可能试图捕捉ConstrainViolationException用于验证。如果一个特定的操作抛出异常，为什么不显示一条（根据错误代码和文本而定制的）失败消息给应用程序的用户，让他们纠正错误呢？这个策略有两个明显的缺点。第一，依靠数据库抛出unchecked值来查看问题，对于可伸缩的应用程序而言，这并不是一种好策略。你想要至少在应用层实现一些数据完整性的验证。第二，所有的异常对于当前的工作单元而言都是致命的。然而，这并不是应用程序的用户对验证错误的理解——他们希望仍然处在工作单元内部。围绕这种不匹配编写的代码很笨拙也很困难。我们的建议是，使用细粒度的异常类型显示更好看的（致命）错误消息。这么做在开发期间有帮助（理想情况下，不应该有致命的异常出现在产品中），也帮助任何客户支持必须迅速做出决定的工程师，如果它是个应用程序错误（违背约束，执行了错误的SQL）或者如果数据库系统正在加载（无法获得锁）。

利用Hibernate Transaction接口在Java SE中编程式的事务划分，使代码保持可移植。它也可以在托管环境内部运行，当事务管理器处理数据库资源的时候。

3．使用JTA的编程式事务

能与Java EE兼容的托管运行时环境，能够为你管理资源。在大多数情况下，被管理的资源都是数据库连接，但是任何带有适配器（adaptor）的资源都可以与Java EE系统整合（如消息或者遗留系统）。在那些资源上的编程式事务划分，如果它们是事务的，就被统一并公开给使用JTA的开发人员；javax.transaction.UserTransaction是启动和终止事务的主要接口。

常见的托管运行时环境是Java EE应用程序服务器。当然，应用程序服务器提供更多的服务，而不仅仅是资源管理。许多Java EE服务是模块化的——安装应用程序服务器不是获得它们的唯一方法。如果你只需要托管的资源，可以获得一个独立的JTA提供程序。开源独立的JTA提供程序包括JBoss Transactions（http://www.jboss.com/products/transactions）、ObjectWeb JOTM（http://jotm.objectweb.org）和其他的。可以和Hibernate应用程序一起安装这样一个JTA服务（例如在Tomcat中）。它将替你管

理数据库连接池，给事务划分提供JTA接口，并通过JNDI注册提供托管的数据库连接。

下列是使用JTA的托管资源的好处，以及使用这个Java EE服务的理由：

- 事务管理服务可以统一所有的资源，无论什么类型，并用单个标准的API把事务控制公开给你。这意味着你可以替换Hibernate的`Transaction` API，并直接在任何地方使用JTA。然后，在（或者用）JTA兼容的运行时环境中安装应用程序就是应用程序部署人员的责任了。这个策略把可移植性关注点归属的位置转移了；应用程序依赖标准的Java EE接口，并且运行时环境必须提供实现。
- Java EE事务管理器可以在单个事务中获取多个资源。如果你使用几个数据库（或者不止一个资源），就可能想要一个两阶段提交（two-phase commit）协议来保证跨资源范围的事务原子性。在这样的场景中，Hibernate通过配置几个`SessionFactory`（每个数据库一个），他们的`Session`获得所有参与同一个系统事务的托管数据库连接。
- 与简单的JDBC连接池相比，JTA实现的质量通常更高。应用程序服务器和作为应用程序服务器模块的独立JTA提供程序，通常在包含大量事务的高端系统中已经进行过更多的测试。
- JTA提供程序在运行时不增加不必要的过载（一种常见的错误概念）。简单案例（单个JDBC数据库）的处理与使用简单的JDBC事务一样有效。比起与简单的JDBC一起使用的随机连接池库，JTA服务背后托管的连接池可能是更好的软件。

假设你不喜欢JTA，并想要继续使用Hibernate `Transaction` API来保证代码在Java EE中和使用托管Java EE服务时可以运行，而不用改变任何代码。为了部署前面的代码实例（它们全部在Java EE应用程序服务器上调用Hibernate `Transaction` API），你需要把Hibernate配置转换到JTA：

- `hibernate.transaction.factory_class`选项必须设置为`org.hibernate.transaction.JTATransactionFactory`。
- Hibernate需要知道你正在哪个JTA实现中部署，这出于两个原因：第一，不同的实现可能公开JTA `UserTransaction`，Hibernate现在必须以不同的名称内部调用它。第二，Hibernate必须钩进JTA事务管理器的同步过程来处理它的高速缓存。你必须设置`hibernate.transaction.manager_lookup_class`选项来配置这两项：例如，配置为`org.hibernate.transaction.JBossTransactionManagerLookup`。在类中查找最常见的JTA实现，并且应用程序服务器中附有Hibernate（需要时可以定制）。在Javadoc中检查是否有包。
- Hibernate不再负责管理JDBC连接池；它从运行时容器获得托管的数据库连接。这些连接被JTA提供程序通过JNDI这个全局的注册而被公开。你必须在JNDI上使用正确的名称对数据库资源配置Hibernate，就像在2.4.1节中所做的那样。

现在，你之前直接在JDBC顶层给Java SE编写的同一段代码，在包含托管数据源的JTA环境中也有效：

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();
```

```
        tx.setTimeout(5);

        concludeAuction(session);

        tx.commit();
    } catch (RuntimeException ex) {
        try {
            tx.rollback();
        } catch (RuntimeException rbEx) {
            log.error("Couldn't roll back transaction", rbEx);
        }
        throw ex;
    } finally {
        session.close();
    }
```

然而，数据库连接处理稍微不同。Hibernate给你正在使用的每个Session获得一个托管的数据库连接，并且还是努力尽可能地延迟。没有JTA，Hibernate将从一开始就停在一个特定的数据库连接上，直接事务终止。有了JTA配置，Hibernate甚至更为积极：获得一个连接，并只用于单个SQL语句，然后立即被返回到托管的连接池。应用程序服务器保证当另一个SQL语句再次需要连接时，它将在同一个事务期间分发出同一个连接。这个积极的连接-释放模式是Hibernate的内部行为，对于应用程序以及如何编写代码，都没有任何影响。（因而，这个代码实例每一行都与原来的一样。）

JTA系统支持全局的事务超时；它可以监控事务。因此，现在setTimeout()控制全局的JTA超时设置——相当于调用UserTransaction.setTransactionTimeout()。

Hibernate Transaction API用Hibernate配置一个简单的变化来保证可移植性。如果你想要把这个责任转移到应用部署程序上，就应该针对标准的JTA接口编写代码。为使下列代码更值得关注，你还将在同一个系统事务内部使用两个数据库（两个SessionFactory）：

```
UserTransaction utx = (UserTransaction) new InitialContext()
                        .lookup("java:comp/UserTransaction");

Session session1 = null;
Session session2 = null;

try {
    utx.begin();

    session1 = auctionDatabase.openSession();
    session2 = billingDatabase.openSession();

    concludeAuction(session1);
    billAuction(session2);

    session1.flush();
    session2.flush();

    utx.commit();
} catch (RuntimeException ex) {
    try {
        utx.rollback();
    } catch (RuntimeException rbEx) {
```

```
            log.error("Couldn't roll back transaction", rbEx);
        }
        throw ex;
    } finally {
        session1.close();
        session2.close();
    }
```

（注意，这个代码片段可以抛出一些其他的checked exception，就像从JNDI查找中抛出的NamingException。你需要对这些做相应的处理。）

首先，必须从JNDI注册上获得JTA UserTransaction上的句柄。然后，开启和关闭事务，并且所有的Hibernate Session所用的（容器提供）数据库连接都在这个事务中被自动获取。即使你没有使用Transaction API，也仍然应该给JTA和环境配置hibernate.transaction.factory_class和hibernate.transaction.manager_lookup_class，以便Hibernate可以与事务系统内部进行交互。

利用默认的设置，手工flush()每个Session，使它与数据库同步（来执行所有SQL DML）也是你的责任了。Hibernate Transaction API以前自动为你完成。你还必须手工关闭所有的Session。另一方面，可以启用hibernate.transaction.flush_before_completion和（或）hibernate.transaction.auto_close_session配置选项，并再次让Hibernate替你负责这项工作——然后，清除和关闭就成为事务管理器内部同步过程的一部分，并且发生在JTA事务终止之前（相应地，或者之后）。启用这两项设置的代码可以简化为如下：

```
UserTransaction utx = (UserTransaction) new InitialContext()
                        .lookup("java:comp/UserTransaction");

Session session1 = null;
Session session2 = null;

try {
    utx.begin();

    session1 = auctionDatabase.openSession();
    session2 = billingDatabase.openSession();

    concludeAuction(session1);
    billAuction(session2);

    utx.commit();
} catch (RuntimeException ex) {
    try {
        utx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
}
```

session1和session2持久化上下文现在在UserTransaction的提交期间被自动清除，并且两者都在事务完成之后关闭。

建议尽可能地直接使用JTA。你应该始终努力把可移植性的责任转移到应用程序之外，如果

可以，要求部署在一个提供JTA的环境中。

编程式的事务划分需要针对事务划分接口而编写的应用程序代码。避免任何不可移植的代码传播到整个应用程序的一种更好的方法是，*声明式*（declarative）事务划分。

4．容器托管事务

声明式事务划分意味着容器替你负责这个关注点。你*声明*想要代码在一个事务中是否参与以及如何参与。通过应用程序部署人员，提供支持声明式事务划分容器的责任再次落到了它所属的地方。

CMT是Java EE尤其JEB的标准特性，接下来要介绍的代码是基于EJB 3.0会话bean的（只适用于Java EE），你利用注解定义事务范围。注意，如果你必须使用更早版本的EJB 2.1会话bean，实际的数据访问代码并没有改变；然而，必须用XML编写EJB部署描述符来创建事务装配——这在EJB 3.0中是可选的。

（独立的JTA实现不提供容器托管和声明式事务。但是JBoss应用程序服务器可以当作模块服务器使用，包含最少的模块，如有必要，它可以只提供JTA和EJB 3.0容器。）

假设EJB 3.0会话bean实现了一个终止拍卖的动作。你之前用编程式的JTA事务划分编写的代码被移到了一个无状态会话bean里面：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item) {
        Session session1 = auctionDatabase.openSession();
        Session session2 = billingDatabase.openSession();

        concludeAuction(session1, item);
        billAuction(session2, item);
    }
    ...
}
```

容器注意到了你的TransactionAttribute声明，并把它应用到endAuction()方法。如果调用方法时没有系统事务在运行，就会启动一个新的事务（这是REQUIRED）。一旦方法返回，并且如果调用这个方法（不是任何其他方法）时启动了事务，这个事务就提交。如果方法内部的代码抛出RuntimeException，系统事务就自动回滚。

为了方便举例，我们再给两个数据库引入两个SessionFactory。它们可以通过JNDI查找（Hibernate可以在启动时绑定它们）或者从HibernateUtil的增强版中进行分配。这两者都获得通过相同的容器托管事务而获取的数据库连接。如果容器的事务系统和资源支持它，你就再次获得一个两阶段提交协议，确保跨数据库的事务原子性。

必须用Hibernate设置一些配置选项来启用CMT：

- hibernate.transaction.factory_class选项必须被设置为org.hibernate.transaction.CMTTransactionFactory。
- 你需要给应用程序服务器设置hibernate.transaction.manager_lookup_class为正确的查找类。

还要注意，所有EJB会话bean都默认为CMT，因此如果你想要禁用CMT并在任何会话bean方法中直接调用JTA UserTransaction，就用@TransactionManagement（TransactionManagementType.BEAN）注解EJB类。然后你就是在使用bean托管事务（bean-managed transaction，BMT）了。即使它可能适用于大多数的应用程序服务器，但是Java EE规范还是不允许在单个bean中混合CMT和BMT。

CMT代码看起来已经比编程式的事务划分好多了。如果配置Hibernate来使用CMT，它就知道应该自动清除和关闭参与系统事务的Session。此外，你将很快改进这段代码，甚至移除打开Hibernate Session的那两行。

来看看Java Persistence应用程序中的事务处理。

10.1.3 使用 Java Persistence 的事务

使用Java Persistence也要进行设计抉择：是在应用程序代码中编程式地事务划分，还是由运行时容器自动处理声明式事务划分。先通过简单的Java SE探讨第一种方法，然后用JTA和EJB组件重复例子。

描述本地资源（resource-local）的事务应用到由应用程序（编程式地）控制的、且不参与全局系统事务的所有事务。它们直接变为你正在处理的资源的本地事务系统。由于你正在使用JDBC数据库，这意味着本地资源的事务变成了JDBC数据库事务。

JPA中本地资源的事务通过EntityTransaction API控制。这个接口不是为可移植性而存在，而是用来启用Java Persistence的特定特性——例如，当提交事务时，底层持久化上下文的清除。

你已经多次在Java SE中见过Java Persistence的标准习惯用语。以下代码仍然带有异常处理：

```
EntityManager em = null;
EntityTransaction tx = null;

try {
    em = emf.createEntityManager();
    tx = em.getTransaction();
    tx.begin();

    concludeAuction(em);

    tx.commit();
} catch (RuntimeException ex) {
    try {
        tx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    em.close();
}
```

这个模式接近于它的Hibernate等价物，隐含着相同的含义：必须手工启动和终止数据库事务，并且必须保证应用程序托管的EntityManager在finaly块中关闭。（虽然我们经常介绍不处理异

常或者包在try/catch块中的代码示例，但这不是可选的。）

JPA抛出的异常是RuntimeException的子类型。任何异常都使得当前的持久化上下文无效，并且一旦抛出异常，就不允许你继续使用EntityManager。因此，我们对Hibernate异常处理所讨论的所有策略也适用于Java Persistence异常处理。此外，下列规则也适用：

- 由EntityManager接口的任何方法抛出的任何异常，都会触发当前事务的自动回滚。
- 由javax.persistence.Query接口的任何方法抛出的任何异常，都会触发当前事务的自动回滚，除了NoResultException和NonUniqueResultException之外。因此，前一个捕捉所有异常的代码示例也为这些异常执行回滚。

注意，JPA不提供细粒度的SQL异常类型。最常见的异常是javax.peristence.PersistenceException。被抛出的所有其他异常都是PersistenceException的子类型，除了NoResultException和NonUniqueResultException之外，应该把它们全部都当作是致命的。然而，可以在JPA抛出的任何异常中调用getCause()，并找出被包装的原生Hibernate异常，包括细粒度的SQL异常类型。

如果你在应用程序服务器内部或者至少提供JTA（对于Hibernate请见我们前面的讨论）的环境中使用Java Persistence，就对编程式的事务划分调用JTA接口。EntityTransaction接口只对本地资源的事务可用。

1．使用Java Persistence的JTA事务

如果Java Persistence代码是在一个可用的JTA环境中部署，并且你想要使用JTA系统事务，就需要调用JTA UserTransaction接口来编程式地控制事务范围：

```
UserTransaction utx = (UserTransaction) new InitialContext()
                        .lookup("java:comp/UserTransaction");
EntityManager em = null;

try {
    utx.begin();

    em = emf.createEntityManager();

    concludeAuction(em);

    utx.commit();
} catch (RuntimeException ex) {
    try {
        utx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    em.close();
}
```

EntityManager的持久化上下文被界定到JTA事务。被这个EntityManager清除的所有SQL语句，都在通过事务而获取的数据库连接中的一个JTA事务内部执行。当JTA事务提交时，持久化上下文被自动清除和关闭。可以用几个EntityManager在同一个系统事务中访问几个数据库，

就像在原生的Hibernate应用程序中使用几个Session一样。

注意持久化上下文的范围改变了！现在它被界定到了JTA事务，并且一旦提交事务，事务期间原来处于持久化状态的所有对象就立即被视为脱管。

异常处理的规则相当于那些用于本地资源的事务的规则。如果在EJB中使用JTA，别忘了在类上设置@TransactionManagement（TransactionManagementType.BEAN）来启用BMT。

你不会经常通过JTA使用Java Persistence，也没有可用的EJB容器。如果你没有部署一个独立的JTA实现，Java EE 5.0应用程序服务器就将提供这两样东西。你不用编程式的事务划分，而是可能使用EJB的声明特性。

2. Java Persistence和CMT

让我们从前面只用Hibernate的示例中，把ManageAuction EJB会话bean重构到Java Persistence接口。你还让容器注入EntityManager：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

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

    @PersistenceContext(unitName = "billingDB")
    private EntityManager billingEM;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item)
                throws AuctionNotValidException {
        concludeAuction(auctionEM, item);
        billAuction(billingEM, item);
    }
    ...
}
```

在concludeAuction()和billAuction()方法内部发生的一切与这个例子仍不相关；假设它们需要用EntityManager来访问数据库。用于endAuction()方法的TransactionAttribute要求所有的数据库访问都发生在一个事务内部。如果endAuction()调用时，没有系统事务是活动的，就会给这个方法启动一个新事务。如果方法返回，且如果事务是为这个方法启动的，它就被提交。每个EntityManager都有一个跨越事务范围、并且当事务结束时被自动清除的持久化上下文。如果方法被调用时没有活动的事务，持久化上下文就有与endAuction()方法相同的范围。

这两个持久化单元都被配置为在JTA上部署，因此两个托管的数据库连接（每个数据库一个），都在同一个事务内部获得，原子性受到应用程序服务器的事务管理器的保护。

你声明endAuction()方法可以抛出AuctionNotValidException。这是你编写的一个定制异常；在终止拍卖之前，检查是否一切都准确无误（已经到达拍卖终止时间、有一个出价等）。这是个checked exception，是java.lang.Exception的子类型。EJB容器把它当成应用程序异常（application exception）处理，如果EJB方法抛出这个异常，则不会触发任何动作。然而，容器认识系统异常（system exception），它默认为所有可能被EJB方法抛出的未检查RuntimeException。

由EJB方法抛出的系统异常强制系统事务的自动回滚。

换句话说，你不需要从Java Persistence操作中捕捉和重新抛出任何系统异常——让容器来处理它们。如果抛出应用程序异常，那么对于如何回滚事务，你有两种选择：第一，可以捕捉它，并手工调用JTA `UserTransaction`，设置它回滚。或者把`@ApplicationException(rollback=true)`注解添加到`AuctionNotValidException`的类——然后容器就知道，你希望每当EJB方法抛出这个应用程序异常时就自动回滚。

现在，你准备在应用程序服务器的内部和外部、用或者不用JTA，以及在EJB和容器管理事务的组合中使用Java Persistence和Hibernate。我们已经讨论了事务原子性的（几乎）所有方面。对于并发运行的事务之间的隔离性，你自然可能仍存疑虑。

10.2　控制并发访问

数据库（和其他的事务系统）试图确保事务隔离性（transaction isolation），这意味着，从每个并发事务的观点来看，似乎没有其他的事务在运行。传统上而言，这已经通过锁（locking）实现了。事务可以在数据库中一个特定的数据项目上放置一把锁，暂时防止通过其他事务访问这个项目。一些现代的数据库（如Oracle和PostgreSQL）通过多版本并发控制（multiversion concurrency control，MVCC）实现事务隔离性，这种多版本并发控制通常被认为是更可伸缩的。我们将讨论假设了锁模型的隔离性；但是我们的大部分论述也适用于多版本并发控制。

数据库如何实现并发控制，这在Hibernate和Java Persistence应用程序中是至关重要的。应用程序继承由数据库管理系统提供的隔离性保证。例如，Hibernate从不锁上内存中的任何东西。如果你认为数据库供应商有多年实现并发控制的经验，就会发现这种方法的好处。另一方面，Hibernate和Java Persistence中的一些特性（要么因为你使用它们，要么按照设计）可以改进远甚于数据库所提供的隔离性保证。

以几个步骤讨论并发控制。我们探讨最底层，研究数据库提供的事务隔离性保证。然后，看一下Hibernate和Java Persistence特性对于应用程序级的悲观和乐观并发控制，以及Hibernate可以提供哪些其他的隔离性保证。

10.2.1　理解数据库级并发

作为Hibernate应用程序的开发人员，你的任务是理解数据库的能力，以及如果特定的场景（或者按照数据完整性需求）需要，如何改变数据库的隔离性行为。让我们退一步来看。如果我们正在讨论隔离性，你可能假设两种情况，即隔离或者不隔离；现实世界中没有灰色区域。说到数据库事务，完全隔离性的代价就很高了。有几个隔离性级别（isolation level），它们一般削弱完全的隔离性，但提升了系统的性能和可伸缩性。

1．事务隔离性问题

首先，来看一下削弱完全事务隔离性时可能出现的几种现象。ANSI SQL标准根据数据库管理系统允许的现象定义了标准的事务隔离性级别：

如果两个事务都更新一个行，然后第二个事务异常终止，就会发生丢失更新（lost update），

导致两处变化都丢失。这发生在没有实现锁的系统中。此时没有隔离并发事务。如图10-2所示。

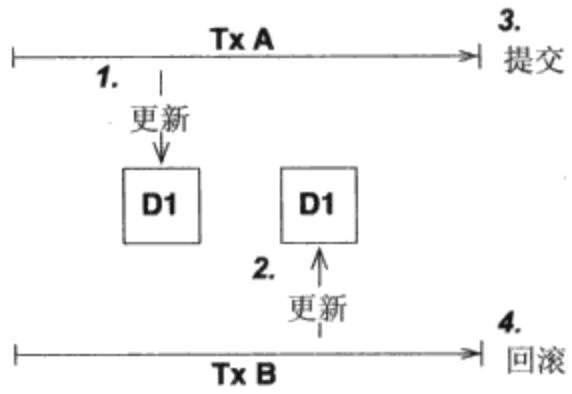

图10-2 丢失更新：两个事务更新没有加锁的同一数据

如果一个事务读取由另一个还没有被提交的事务进行的改变，就发生脏读取（dirty read）。这很危险，因为由其他事务进行的改变随后可能回滚，并且第一个事务可能编写无效的数据，如图10-3所示。

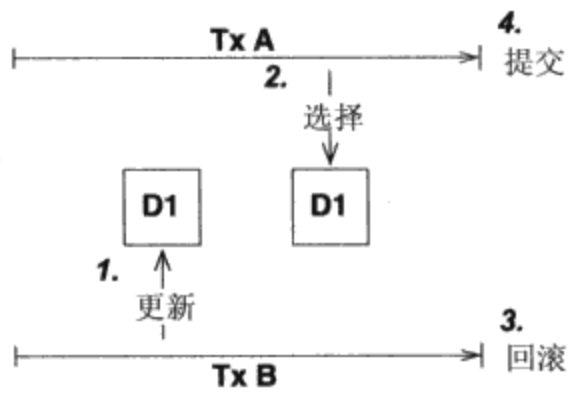

图10-3 脏读取：事务A读取没有被提交的数据

如果一个事务读取一个行两次，并且每次读取不同的状态，就会发生不可重复读取（unrepeatable read）。例如，另一个事务可能已经写到这个行，并已在两次读取之间提交，如图10-4所示。

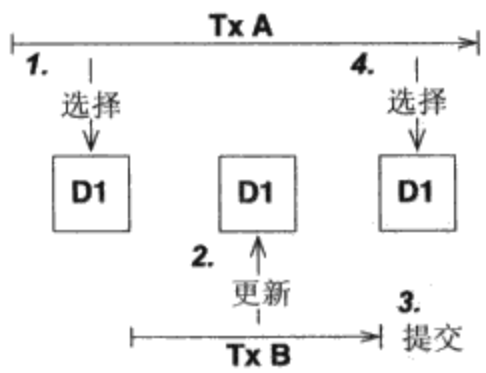

图10-4 不可重复读取：事务A执行不可重复读取两次

不可重复读取的一个特殊案例是二次丢失更新问题（second lost updates problem）。想象两个并发事务都读取一个行：一个写到行并提交，然后第二个也写到行并提交。由第一个事务所做的改变丢失了。如果考虑需要几个数据库事务来完成的应用程序对话，这个问题就特别值得关注。

我们将在稍后更深入地探讨这种情况。

幻读（phantom read）发生在一个事务执行一个查询两次，并且第二个结果集包括第一个结果集中不可见的行，或者包括已经删除的行时。（不需要是完全相同的查询。）这种情形是由另一个事务在两次查询执行之间插入或者删除行造成的，如图10-5所示。

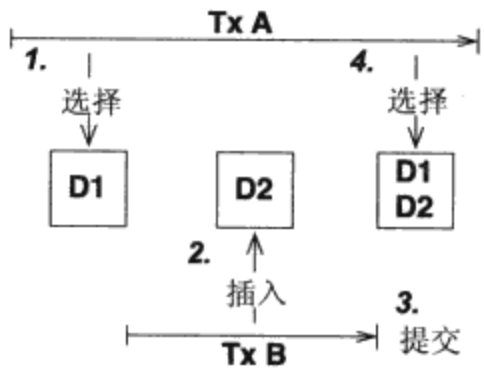

图10-5　幻读：事务A在第二次选择中读取新数据

既然已经了解所有可能发生的不好的情况，我们就可以定义事务隔离性级别了，并看看它们阻止了哪些问题。

2. ANSI事务隔离性级别

标准的隔离性级别由ANSI SQL标准定义，但是它们不是SQL数据库特有的。JTA也定义了完全相同的隔离性级别，稍后你将用这些级别声明想要的事务隔离性。隔离性级别的增加带来了更高成本以及严重的性能退化和可伸缩性：

- 允许脏读取但不允许丢失更新的系统，据说要在读取未提交（read uncommitted）的隔离性中操作。如果一个未提交事务已经写到一个行，另一个事务就不可能再写到这个行。但任何事务都可以读取任何行。这个隔离性级别可以在数据库管理系统中通过专门的写锁来实现。
- 允许不可重复读取但不允许脏读取的系统，据说要实现读取提交（read committed）的事务隔离性。这可以用共享的读锁和专门的写锁来实现。读取事务不会阻塞其他事务访问行。但是未提交的写事务阻塞了所有其他的事务访问该行。
- 在可重复读取（repeatable read）隔离性模式中操作的系统既不允许不可重复读取，也不允许脏读取。幻读可能发生。读取事务阻塞写事务（但不阻塞其他的读取事务），并且写事务阻塞所有其他的事务。
- 可序列化（serializable）提供最严格的事务隔离性。这个隔离性级别模拟连续的事务执行，好像事务是连续地一个接一个地执行，而不是并发地执行。序列化不可能只用低级锁实现。一定有一些其他的机制，防止新插入的行变成对于已经执行会返回行的查询的事务可见。

锁系统在DBMS中具体如何实现很不相同；每个供应商都有不同的策略。你应该查阅DBMS文档，找出更多有关锁系统的信息，如何逐步加强锁（例如从低级别到页面，到整张表），以及每个隔离性级别对于系统性能和可伸缩性有什么影响。

知道所有这些技术术语如何定义，这很好，但是它如何帮助你给应用程序选择隔离性级别呢？

3. 选择隔离性级别

开发人员（包括我们自己）经常不确定要在一个产品应用程序中使用哪种事务隔离性级别。隔离性太强会损害高并发应用程序的可伸缩性。隔离性不足则可能在应用程序中导致费解的、不可重现的bug，直到系统过载运行时才会发现。

注意，我们在接下来的阐述中所指的乐观锁（optimistic locking）（利用版本），是本章稍后要解释的一个概念。你可能想要跳过这一节，并且当要在应用程序中决定隔离性级别时再回来。毕竟，选择正确的隔离性级别很大程度上取决于特定的场景。把以下讨论当作建议来读，不要把它们当作金科玉律。

在数据库的事务语义方面，Hibernate努力尽可能地透明。不过，高速缓存和乐观锁影响着这些语义。在Hibernate应用程序中要选择什么有意义的数据库隔离性级别呢？

首先，消除读取未提交隔离性级别。在不同的事务中使用一个未提交的事务变化是很危险的。一个事务的回滚或者失败将影响其他的并发事务。第一个事务的回滚可能战胜其他的事务，或者甚至可能导致它们使数据库处于一种错误的状态中。甚至由一个终止回滚的事务所做的改变也可能在任何地方被提交，因为它们可以读取，然后由另一个成功的事务传播！

其次，大多数应用程序不需要可序列化隔离性（幻读通常不成问题），并且这个隔离性级别往往难以伸缩。现有的应用程序很少在产品中使用序列化隔离性，但在某些情况下，有效地强制一个操作序列化地执行相当依赖于悲观锁（请见接下来的几节内容）。

这样就把选择读取提交还是可重复读取留给你来决定了。我们先考虑可重复读取。如果所有的数据访问都在单个原子的数据库事务中执行，这个隔离性级别就消除了一个事务可能覆盖由另一个并发事务所做变化（第二个丢失更新问题）的可能性。事务持有的读锁防止了并发事务可能希望获得的任何写锁。这是一个重要的问题，但是启用可重复读取并不是唯一的解决办法。

假设你正使用版本化（versioned）的数据，这是Hibernate可以自动完成的东西。（必需的）持久化上下文高速缓存和版本控制的组合已经提供了可重复读取隔离性的大部分优良特性。特别是，版本控制防止了二次丢失更新问题，并且持久化上下文高速缓存也确保了由一个事务加载的持久化实例状态与由其他事务所做的变化隔离开来。因此，如果你使用版本化的数据，那么对于所有数据库事务来说，读取提交的隔离性是可以接受的。

可重复读取给查询结果集（只针对数据库事务的持续期间）提供了更多的可复制性；但是因为幻读仍然可能，这似乎没有多大价值。可以在Hibernate中给一个特定的事务和数据块显式地获得可重复读取的保证（通过悲观锁）。

设置事务隔离性级别允许你给所有的数据库事务选择一个好的默认锁策略。如何设置隔离性级别呢？

4. 设置隔离性级别

与数据库的每一个JDBC连接都处于DBMS的默认隔离性级别——通常是读取提交或者可重复读取。可以在DBMS配置中改变这个默认。还可以在应用程序端给JDBC连接设置事务隔离性，

通过一个Hibernate配置选项：

```
hibernate.connection.isolation = 4
```

Hibernate在启动事务之前，给每一个从连接池中获得的JDBC连接设置这个隔离性级别。对于这个选项有意义的值如下（你也可能发现它们为java.sql.Connection中的常量）：

- 1——读取未提交隔离性。
- 2——读取提交隔离性。
- 3——可重复读取隔离性。
- 4——可序列化隔离性。

注意，Hibernate永远不会改变在托管环境中从应用程序服务器提供的数据库连接中获得的连接隔离性级别！可以利用应用程序服务器的配置改变默认的隔离性级别。（如果使用独立的JTA实现也一样。）

如你所见，设置隔离性级别是影响所有连接和事务的一个全局选项。给特定的事务指定一个更加限制的锁经常很有用。Hibernate和Java Persistence依赖乐观的并发控制，并且两者都允许你通过版本检查和悲观锁，获得额外的锁保证。

10.2.2　乐观并发控制

乐观的方法始终假设一切都会很好，并且很少有冲突的数据修改。在编写数据时，乐观并发控制只在工作单元结束时才出现错误。多用户的应用程序通常默认为使用读取提交隔离性级别的乐观并发控制和数据库连接。只有适当的时候（例如，当需要可重复读取的时候）才获得额外的隔离性保证；这种方法保证了最佳的性能和可伸缩性。

1．理解乐观策略

为了理解乐观并发控制，想象两个事务从数据库中读取一个特定的对象，并且两者都对它进行修改。由于数据库连接的读取提交隔离性级别，因此没有任何一个事务会遇到任何脏读取。然而，读取仍然是不可重复的，并且更新还是可能丢失。这是当你在考虑对话的时候要面对的问题，从用户的观点来看，这些是原子的事务。请见图10-6。

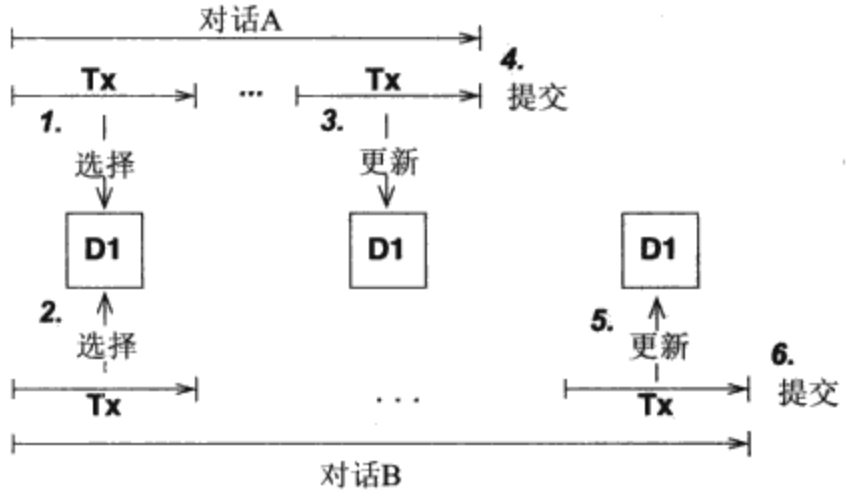

图10-6　对话B覆盖对对话A所做的改变

假设两个用户同时选择同一块代码。对话A中的用户先提交了变化，并且对话终止于第二个事务的成功提交。过了一会儿（可能只是一秒钟），对话B中的用户提交了变化。第二个事务也成功提交。在对话A中所做的改变已经丢失，并且（可能更糟的是）对话B中提交的数据修改可能已经基于失效的信息。

对于如何处理对话中这些第二个事务中的丢失更新，你有3种选择：

- 最晚提交生效（last commit wins）——两个事务提交都成功，且第二次提交覆盖第一个的变化。没有显示错误消息。
- 最先提交生效（first commit wins）——对话A的事务被提交，并且在对话B中提交事务的用户得到一条错误消息。用户必须获取新数据来重启对话，并再次利用没有失效的数据完成对话的所有步骤。
- 合并冲突更新（merge conflicting updates）——第一个修改被提交，并且对话B中的事务在提交时终止，带有一条错误消息。但是失败的对话B用户可以选择性地应用变化，而不是再次在对话中完成所有工作。

如果你没有启用乐观并发控制（默认情况为未启用），应用程序就会用最晚提交生效策略运行。在实践中，丢失更新的这个问题使得许多应用程序的用户很沮丧，因为他们可能发现他们的所有工作都丢失了，而没有收到任何错误消息。

很显然，最先提交生效更有吸引力。如果对话B的应用程序的用户提交，他就获得这样一条错误消息：有人已经对你要提交的数据提交了修改。你已经使用了失效数据。请用新数据重启对话。（Somebody already committed modifications to the data you're about to commit. You've been working with stale data. Please restart the conversation with fresh data。）设计和编写生成这条错误消息的应用程序，并引导用户重新开始对话，这就是你的责任了。Hibernate和Java Persistence用自动乐观锁协助你，以便每当事务试图提交在数据库中带有冲突的被更新状态的对象时，就会得到一个异常。

合并冲突的变化，是最先提交生效的一种变形。不显示始终强制用户返回的错误消息，而是提供一个对话框，允许用户手工合并冲突的变化。这是最好的策略，因为没有工作丢失，应用程序的用户也不会因为乐观并发失败而受挫。然而，对于开发人员来说，提供一个对话框来合并变化比显示一条错误消息并强制用户重复所有的工作来得更加费时。是否使用这一策略，由你自己决定。

乐观并发控制可以用多种方法实现。Hibernate使用自动的版本控制。

2. 在Hibernate中启用版本控制

Hibernate提供自动的版本控制。每个实体实例都有一个版本，它可以是一个数字或者一个时间戳。当对象被修改时，Hibernate就增加它的版本号，自动比较版本，如果侦测到冲突就抛出异常。因此，你给所有持久化的实体类都添加这个版本属性，来启用乐观锁：

```
public class Item {
    ...
    private int version;
    ...
}
```

也可以添加获取方法；但是不许应用程序修改版本号。XML格式的<version>属性映射必须立即放在标识符属性映射之后：

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

    <version name="version" access="field" column="OBJ_VERSION"/>

    ...
</class>
```

版本号只是一个计数值——它没有任何有用的语义值。实体表上额外的列为Hibernate应用程序所用。记住，所有访问相同数据库的其他应用程序也可以（并且或许应该）实现乐观版本控制，并利用相同的版本列。有时候时间戳是首选（或者已经存在）：

```
public class Item {
    ...
    private Date lastUpdated;
    ...
}

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

    <id .../>

    <timestamp name="lastUpdated"
               access="field"
               column="LAST_UPDATED"/>
    ...
</class>
```

理论上来说，时间戳更不安全一点，因为两个并发的事务可能都在同一毫秒点上加载和更新同一件货品；但在实践中不会发生这种情况，因为JVM通常没有精确到毫秒（你应该查阅JVM和操作系统文档所确保的精确度）。

此外，从JVM处获取的当前时间在集群环境（clustered environment）下并不一定安全，该环境中的节点可能不与时间同步。可以转换为在<timestamp>映射中利用source="db"属性从数据库机器中获取当前的时间。并非所有的Hibernate SQL方言都支持这个属性（检查所配置的方言的源代码），每一次增加版本都始终会有命中数据库的过载。

我们建议新项目依赖包含版本号的版本，而不是时间戳。

一旦你把<version>或者<timestamp>属性添加到持久化类映射，就启用了包含版本的乐观锁。没有其他的转换。

Hibernate如何利用版本发现冲突？

3.* 版本控制的自动管理

涉及目前被版本控制的Item对象的每一个DML操作都包括版本检查。例如，假设在一个工作单元中，你从版本为1的数据库中加载一个Item。然后修改它的其中一个值类型属性，例如Item的价格。当持久化上下文被清除时，Hibernate侦测到修改，并把Item的版本增加到2。然后执行SQL UPDATE使这一修改在数据库中永久化：

```
update ITEM set INITIAL_PRICE='12.99', OBJ_VERSION=2
    where ITEM_ID=123 and OBJ_VERSION=1
```

如果另一个并发的工作单元更新和提交了同一个行，OBJ_VERSION列就不再包含值1，行也不会被更新。Hibernate检查由JDBC驱动器返回这个语句所更新的行数——在这个例子中，被更新的行数为0——并抛出StaleObjectStateException。加载Item时呈现的状态，清除时不再在数据库中呈现；因而，你正在使用失效的数据，必须通知应用程序的用户。可以捕捉这个异常，并显示一条错误消息，或者显示帮助用户给应用程序重启对话的一个对话框。

什么样的修改触发实体版本的增加？每当实体实例脏时，Hibernate就增加版本号（或者时间戳）。这包括实体的所有脏的值类型属性，无论它们是单值、组件还是集合。考虑User和BillingDetails之间的关系，这是个一对多的实体关联：如果CreditCard修改了，相关的User版本并没有增加。如果你从账单细节的集合中添加或者删除CreditCard（或者BankAccount），User的版本就增加了。

如果你想要禁用对特定值类型属性或者集合的自动增加，就用optimistic-lock="false"属性映射它。inverse属性在这里没有什么区别。甚至如果元素从反向集合中被添加或者移除，反向集合的所有者的版本也会被更新。

如你所见，Hibernate使得对于乐观并发控制管理版本变得难以置信地轻松。如果你正在使用遗留数据库Schema或者现有的Java类，也许不可能引入版本或者时间戳和列。Hibernate提供了另一种可选的策略。

4．没有版本号或者时间戳的版本控制

如果你没有版本或者时间戳列，Hibernate仍然能够执行自动的版本控制，但是只对在同一个持久化上下文中获取和修改的对象（即相同的Session）。如果你需要乐观锁用于通过脱管对象实现的对话，则必须使用通过脱管对象传输的版本号或者时间戳。

这种可以选择的版本控制实现方法，在获取对象（或者最后一次清除持久化上下文）时，把当前的数据库状态与没有被修改的持久化属性值进行核对。可以在类映射中通过设置optimistic-lock属性来启用这项功能：

```
<class name="Item" table="ITEM" optimistic-lock="all">
    <id .../>
    ...
</class>
```

下列SQL现在被执行，用来清除Item实例的修改：

```
update ITEM set ITEM_PRICE='12.99'
where ITEM_ID=123
  and ITEM_PRICE='9.99'
  and ITEM_DESCRIPTION="An Item"
  and ...
  and SELLER_ID=45
```

Hibernate在SQL语句的WHERE子句中，列出了所有列和它们最后知道的非失效值。如果任何并发的事务已经修改了这些值中的任何一个，或者甚至删除了行，这个语句就会再次返回被更新的行数为0。然后Hibernate抛出一个StaleObjectStateException。

另一种方法是，如果设置optimistic-lock="dirty"，Hibernate只包括限制中被修改的属性（在这个例子中，只有ITEM_PRICE）。这意味着两个工作单元可以同时修改同一个对象，并且只有当两者修改同一个值类型属性（或者外键值）时才会侦测到冲突。在大多数情况下，这对于业务实体来说并不是一种好策略。想象有两个人同时修改一件拍卖货品：一个改变价格，另一个改变描述。即使这些修改在最低级别（数据库行）没有冲突，从业务逻辑观点看它们也可能发生冲突。如果货品的描述完全改变了，还可以改变它的价格吗？如果你想要使用这个策略，还必须在实体的类映射上启用dynamic-update="true"，Hibernate无法在启动时给这些动态的UPDATE语句生成SQL。

不建议在新应用程序中定义没有版本或者时间戳列的版本控制；它更慢、更复杂，如果你正在使用脱管对象，则它不会生效。

Java Persistence应用程序中的乐观并发控制与Hibernate中的几乎如出一辙。

5. 用Java Persistence版本控制

Java Persistence规范假设并发数据访问通过版本控制被乐观处理。为了给一个特定的实体启用自动版本控制，需要添加一个版本属性或者字段：

```
@Entity
public class Item {
    ...
    @Version
    @Column(name = "OBJ_VERSION")
    private int version;
    ...
}
```

同样地，可以公开一个获取方法，但不能允许应用程序修改版本值。在Hibernate中，实体的版本属性可以是任何数字类型，包括基本类型，或者Date或者Calendar类型。JPA规范只把int、Integer、short、Short、long、Long和java.sql.Timestamp当作可移植的版本类型。

由于JPA标准没有涵盖无版本属性的乐观版本控制，因此需要Hibernate扩展，通过对比新旧状态来启用版本控制：

```
@Entity
@org.hibernate.annotations.Entity(
  optimisticLock = org.hibernate.annotations.OptimisticLockType.ALL
)
public class Item {
    ...
}
```

如果只是希望在版本检查期间比较被修改的属性，也可以转换到OptimisticLockType.DIRTY。然后你还需要设置dynamicUpdate属性为true。

Java Persistence没有对哪个实体实例修改应该触发版本增加标准化。如果你用Hibernate作为JPA提供程序，默认是一样的——每一个值类型的属性修改（包括集合元素的添加和移除）都触发版本增加。在编写本书之时，还没有在特定的属性和集合上禁用版本增加的Hibernate注解，但是已经存在一项对@OptimisticLock(excluded=true)的特性请求。你的Hibernate Annotations

版本或许包括了这个选项。

Hibernate EntityManager，像任何其他Java Persistence提供程序一样，当侦测到冲突版本时，就抛出javax.persistence.OptimisticLockException。这相当于Hibernate中原生的StaleObjectStateException，因此应该进行相应处理。

我们现在已经涵盖了数据库连接的基础隔离性级别，结论是你通常应该依赖来自数据库的读取提交保证。Hibernate和Java Persistence中的自动版本控制，在两个并发事务试图在同一块代码中提交修改时，防止了丢失更新。为了处理非可重复读取，你需要额外的隔离性保证。

10.2.3 获得额外的隔离性保证

有几种方法防止不可重复读取，并升级到一个更高的隔离性级别。

1. 显式的悲观锁

已经讨论了把所有的数据库连接转换到一个比读取提交更高的隔离性级别，但我们的结论是，当关注应用程序的可伸缩性时，这则是一项糟糕的默认。你需要更好、仅用于一个特定的工作单元的隔离性保证。还要记住，持久化上下文高速缓存为处于持久化状态的实体实例提供可重复读取。然而，这并非永远都是足够的。

例如，对标量查询（scalar query）可能需要可重复读取：

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

Item i = (Item) session.get(Item.class, 123);

String description = (String)
        session.createQuery("select i.description from Item i" +
                            " where i.id = :itemid")
               .setParameter("itemid", i.getId() )
               .uniqueResult();

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

这个工作单元执行两次读取。第一次通过标识符获取实体实例。第二次读取标量查询，再次加载已经加载的Item实体的描述。在这个工作单元中有一个小窗口，在那里，并发运行的事务可以在两次读取之间提供一个更新过的货品描述。然后第二次读取返回这个提交数据，且变量description有一个与属性i.getDescription()不同的值。

这个例子进行过简化，但仍然足以说明：如果数据库事务隔离性级别是读取提交，那么混有实体和标量读取的工作单元有多么容易受到非可重复读取的影响。

不是把所有的数据库事务转换为一个更高的、不可伸缩的隔离性级别，而是在必要时，在Hibernate Session中使用lock()方法获得更强的隔离性保证：

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

Item i = (Item) session.get(Item.class, 123);

session.lock(i, LockMode.UPGRADE);
```

```
String description = (String)
        session.createQuery("select i.description from Item i" +
                            " where i.id = :itemid")
                .setParameter("itemid", i.getId() )
                .uniqueResult();

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

使用LockMode.UPGRADE，给表示Item实例的（多）行，促成了在数据库中保存的悲观锁。现在没有并发事务可以在相同数据中获得锁——也就是说，没有并发事务可以在你的两次读取之间修改数据。这段代码可以被缩短成如下：

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

Item i = (Item) session.get(Item.class, 123, LockMode.UPGRADE);
...
```

LockMode.UPGRADE导致一个SQL SELECT ... FOR UPDATE或者类似的东西，具体取决于数据库方言。一种变形LockMode.UPGRADE_NOWAIT，添加了一个允许查询立即失败的子句。如果没有这个子句，当无法获得锁时（可能由于一个并发事务已经有锁），数据库通常会等待。等待的持续时间取决于数据库，就像实际的SQL子句一样。

常见问题　可以使用长悲观锁吗？在Hibernate中，悲观锁的持续时间是单个数据库事务。这意味着你无法使用专门的锁，来阻塞比单个数据库事务更长的并发访问。我们认为这是好的一面，因为对于比如整个会话的持续时间来说，唯一的解决方案将是在内存（或者数据库中所谓的锁定表，lock table）中保存一个非常昂贵的锁。这种锁有时也称作离线（offline）锁。这通常是个性能瓶颈；每个数据访问都要对一个同步锁管理器进行额外的锁检查。然而，乐观锁是最完美的并发控制策略，并且在长运行对话中执行得很好。根据你的冲突解析（conflict-resolution）选项（即如果你有足够的时间实现合并变化），你应用程序的用户对此将会像对被阻塞的并发访问一样满意。他们也可能感激当其他人在看相同数据时，自己没有被锁在特定的屏幕之外。

Java Persistence出于同样的目的定义了LockModeType.READ，且EntityManager也有一个lock()方法。规范没有要求未被版本控制的实体支持这种锁模式；但Hibernate在所有的实体中都支持它，因为它在数据库中默认为悲观锁。

2. Hibernate锁模式

Hibernate支持下列其他LockMode：

- LockMode.NONE——别到数据库中去，除非对象不处于任何高速缓存中。
- LockMode.READ——绕过所有高速缓存，并执行版本检查，来验证内存中的对象是否与当前数据库中存在的版本相同。
- LockMode.UPGRADE——绕过所有高速缓存，做一个版本检查（如果适用），如果支持的话，就获得数据库级的悲观升级锁。相当于Java Persistence中的LockModeType.READ。如

果数据库方言不支持SELECT ... FOR UPDATE选项，这个模式就透明地退回到LockMode.READ。

- LockMode.UPGRADE_NOWAIT——与UPGRADE相同，但如果支持的话，就使用SELECT ... FOR UPDATE NOWAIT。它禁用了等待并发锁释放，因而如果无法获得锁，就立即抛出锁异常。如果数据库SQL方言不支持NOWAIT选项，这个模式就透明地退回到LockMode.UPGRADE。
- LockMode.FORCE——在数据库中强制增加对象的版本，来表明它已经被当前事务修改。相当于Java Persistence中的LockModeType.WRITE。
- LockMode.WRITE——当Hibernate已经在当前事务中写到一个行时，就自动获得它。（这是一种内部模式；你不能在应用程序中指定它。）

默认情况下，load()和get()使用LockMode.NONE。LockMode.READ对session.lock()和脱管对象最有用。这里有个例子：

```
Item item = ... ;
Bid bid = new Bid();
item.addBid(bid);
...
Transaction tx = session.beginTransaction();
session.lock(item, LockMode.READ);
tx.commit();
```

这段代码在通过级联（假设从Item到Bid的关联启用了级联）保存新Bid之前，在脱管的Item实例上执行版本检查，验证该数据库行在获取之后没有被另一个事务更新。

（注意，EntityManager.lock()不重附指定的实体实例——它只对已经处于托管持久化状态的实例有效。）

Hibernate LockMode.FORCE和Java Persistence中的LockModeType.WRITE有着不同的用途。如果默认不增加版本，就利用它们强制版本更新。

3. 强制增加版本

如果通过版本控制启用乐观锁，Hibernate会自动增加被修改实体实例的版本。然而，有时你想手工增加实体实例的版本，因为Hibernate不会把你的改变当成一个应该触发版本增加的修改。

想象你修改了CreditCard所有者的名称：

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

User u = (User) session.get(User.class, 123);

u.getDefaultBillingDetails().setOwner("John Doe");

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

当这个Session被清除时，被修改的BillingDetails实例（我们假设是一张信用卡）的版本通过Hibernate自动增加了。这可能并不是你想要的东西——你可能也想增加所有者（User实例）的版本。

用LockMode.FORCE调用lock()，增加一个实体实例的版本：

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

User u = (User) session.get(User.class, 123);

session.lock(u, LockMode.FORCE);

u.getDefaultBillingDetails().setOwner("John Doe");

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

现在，任何使用相同User行的并发工作单元都知道这个数据被修改了，即使只有被你认为是整个聚合的一部分的其中一个值被修改了。这种技术在许多情况下都有用，例如当你修改一个对象，并且想要增加聚合的根对象版本时。另一个例子是对一件拍卖货品出价金额的修改（如果这些金额是不可变的）：利用一个显式的版本增加，可以指出这件货品已经被修改，即使它的值类型属性或者集合都没有发生改变。利用Java Persistence的同等调用是em.lock(o,LockModeType.WRITE)。

现在，你具备了编写更复杂工作单元和创建对话的所有知识。但是，我们需要提及事务的最后一个方面，因为它在使用JPA的更复杂对话中变得必不可少。你必须理解自动提交如何工作，以及在实践中非事务数据访问意味着什么。

10.3　非事务数据访问

许多数据库管理员在每个新的数据库连接上默认启用所谓的自动提交模式。自动提交模式对于SQL的特殊执行很有用。

想象你用SQL控制台连接到数据库，并且运行几个查询，甚至也可能更新和删除行。这种交互式的数据访问是特殊的；大多数时候，你并不具备考虑工作单元的计划或者一系列语句。数据库连接中默认的自动提交模式对于这种数据访问是至臻完美的——毕竟，你并不想给所编写和执行的每个SQL语句都输入begin a transaction和end a transaction。在自动提交模式中，一个（短）数据库事务给对于发送到数据库的每个SQL语句进行启动和终止。你正在有效地进行非事务地工作，因为对于使用SQL控制台的会话来说，并没有原子性或者隔离性保证。（唯一的保证是单个SQL语句具有原子性。）

按照定义，应用程序始终执行计划好的一系列语句。因而你始终创建事务范围，把语句组合到原子单元中去，这似乎很合理。因此，自动提交模式在应用程序中没有用武之地。

10.3.1　揭开自动提交的神秘面纱

许多开发人员经常出于一些含糊的、不确切的理由，仍然喜欢使用自动提交模式。如果你想（或者必须）了解其中的原因的话，就让我们在介绍如何非事务地访问数据之前先揭开这其中的一部分理由吧：

- 许多应用程序的开发人员认为，他们可以在事务外部与数据库对话。这显然是不可能的；没有任何SQL语句可以被发送到数据库事务外部的数据库。术语非事务的数据访问意味着

没有显式的事务范围，没有系统事务，并且数据访问的行为处于自动提交模式。这并不意味着没有涉及实质性的数据库事务。

- 如果你的目标是利用自动提交模式改进应用程序的性能，就应该重新考虑许多小事务的含义。给每一个SQL语句启动和终止数据库事务涉及了重大的过载，它可能降低应用程序的性能。
- 如果你的目标是利用自动提交模式改进应用程序的可伸缩性，就重新考虑：对于每一个SQL语句来说，用更长运行的数据库事务代替许多小事务，可以保持更长时间的数据库锁，也可能不进行伸缩。但是，由于Hibernate持久化上下文和DML的迟写，数据库中所有写锁已经持续了很短的时间。根据你启用的隔离性级别，读锁的成本可能可以忽略。或者，你可以使用一个不需要读锁（Oracle、PostgreSQL、Informix、Firebird）的包含多版本并发的DBMS，因为默认情况下读取操作永远不会被阻塞。
- 因为你正在进行非事务地工作，不仅真的放弃了一组SQL语句的任何事务原子性，而且如果数据被同时修改，还减弱了隔离性保证。基于读锁不可能有自动提交模式的可重复读取。（持久化上下文高速缓存在此处自然很有帮助。）

在应用程序中引入非事务的数据访问时，要考虑更多的问题。我们已经注意到，引入一种新的事务类型，称作只读事务（read-only transaction），可以明显使应用程序的任何未来修改变得复杂起来。如果你引入非事务的操作也一样。

然后你在应用程序中就有3种不同的数据访问了：普通事务、只读事务以及现在又有不包含保证的非事务。想象你必须引入一项操作，把数据写到假定只读取数据的工作单元中。想象你必须把非事务的操作识别为事务操作。

我们的建议是不在应用程序中使用自动提交模式，并且只在有明显的性能好处，或者很可能要在未来改变代码时，才应用只读的事务。始终更喜欢一般的ACID事务，把数据访问操作组合起来，无论你是读还是写数据。

话虽这么说，Hibernate和Java Persistence还是允许非事务的数据访问。事实上，如果你想实现原子的长运行对话，EJB 3.0规范强制你非事务地访问数据。第11章将探讨这个主题。现在要深入到简单的Hibernate应用程序中自动提交模式所带来的结果。（注意，不要管我们负面的评论，自动提交模式还是有一些好的使用案例。依据我们的经验，自动提交经常由于错误的原因而被启用，我们本想先清除这些错误的观念。）

10.3.2 使用 Hibernate 非事务地工作

看看下列代码，它访问没有事务范围的数据库：

```
Session session = sessionFactory.openSession();

session.get(Item.class, 1231);

session.close();
```

默认情况下，在包含JDBC配置的Java SE环境中，如果执行这个片段，将发生：

(1) 打开一个新Session。此时它没有获得数据库连接。

(2) 调用get()触发一个SQL SELECT。现在Session从连接池获得了JDBC Connection。Hibernate立即默认在这个连接中用setAutoCommit(false)关闭自动提交的模式。这样有效地启动了一个JDBC事务！

(3) SELECT在这个JDBC事务内部执行。Session关闭，且连接被返回到池，并由Hibernate释放——Hibernate在JDBC Connection中调用close()。对于没有被提交的事务，发生了什么事？

这个问题的答案是：不一定！JDBC规范没有提到在一个连接上调用close()时任何未处理的事务的内容。会发生什么取决于供应商如何实现规范。例如，利用Oracle JDBC驱动器，调用close()提交了事务！大多数其他的JDBC供应商走健全的路线，并在JDBC Connection对象被关闭，且资源返回到池中时，回滚任何未处理的事务。

显然，这对于你已经执行的SELECT来说不成问题，但是看看这个变形：

```
Session session = getSessionFactory().openSession();

Long generatedId = session.save(item);

session.close();
```

这段代码促成了INSERT语句，该语句在一个从不被提交或者回滚的事务内部执行。在Oracle中，这段代码永久地插入数据；在其他数据库中，则可能不是。(这种情形稍微更复杂一些：INSERT只有在标识符生成器需要它时才执行。例如，标识符值可以从一个没有INSERT的sequence中获得。然后，持久化实体被列队等待，直到清除时插入——这在这段代码中永远不会发生。identity策略对于要生成的值，需要一个立即的INSERT。)

我们甚至还没有接触到自动提交模式，但是已经强调了一个问题，它可能出现在你试图不设置显式的事务范围进行工作时。假设你仍然认为不用事务划分进行工作是个好主意，并且想要一般的自动提交行为。首先，必须在Hibernate配置中告诉Hibernate允许自动提交的JDBC连接：

```
<property name="connection.autocommit">true</property>
```

利用这个设置，当从连接池获得JDBC连接时，Hibernate不再关闭自动提交——如果连接还没有处于该模式，就启用自动提交。现在前一个代码示例可以预期地生效了，并且JDBC驱动器把被发送到数据库的每一个SQL语句（包含我们前面列出过的那些含义）都包在一个简短的事务中。

在Hibernate中，你在什么情况下要启用自动提交模式，以便可以使用不用手工启动和终止事务的Session呢？受益于自动提交模式的系统是那些要求按需（延迟）加载数据的系统，在一个特定的Session和持久化上下文中，但是在这些系统中很难把所有可能触发按需数据获取的代码都包在事务范围中。这通常不是遵循我们在第16章中讨论的设计模式的Web应用程序中的案例。另一方面，通过Hibernate访问数据库层的桌面应用程序经常要求按需加载，没有显式的事务范围。例如，如果你在Java Swing树视图上双击一个节点，这个节点的所有子节点都必须从数据库中被加载。你最好必须把这个事件手工包在一个事务中；自动提交模式是一种更为方便的解决方案。(注意我们不是在建议按需打开和关闭Session!)

10.3.3 使用 JTA 的可选事务

前面着重讨论了自动提交模式，以及利用非托管的JDBC连接（在这里，Hibernate管理连接池）的应用程序中的非事务数据访问。现在想象你想要在Java EE环境中使用Hibernate，包含JTA，可能也有CMT。connection.autocommit配置选项在这个环境中不起作用。是否使用自动提交，取决于你的事务程序集。

想象你有一个EJB会话bean，它把特定的方法标记为非事务：

```
@Stateless
public class ItemFinder {

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public Item findItemById(Long id) {
        Session s = getSessionFactory().openSession();
        Item item = (Item) s.get(Item.class, id);
        s.close();
        return item;
    }
}
```

findItemById()方法生成一个返回Item实例的立即的SQL SELECT。因为该方法被标记为不支持事务上下文，所以没有为这项操作启动任何事务，并且任何现有的事务上下文在这个方法的持续时间内被暂停。SELECT在自动提交模式中被有效地执行。（自动提交的JDBC连接被内部分配用来服务于这个Session。）

最后，你需要知道当没有事务在处理时，Session的默认FlushMode改变了。默认的行为FlushMode.AUTO，导致在每个HQL、SQL或者Criteria查询之前同步。当然，这样不好，因为除了为查询执行SELECT外，DML UPDATE、INSERT和DELETE操作也执行。因为你正在自动提交模式下工作，这些修改是永久的。当你在事务范围外部使用Session时，Hibernate通过禁用自动清除防止了这一点。然后你不得不期待查询可以返回失效数据或者与当前Session中数据状态冲突的数据——当选择FlushMode.MANUAL时，必须有效地处理相同的问题。

我们将在第11章的对话讨论中，回到非事务数据访问的话题。应该把自动提交行为当作一项特性（你可能在利用Java Persistence或者EJB的对话中使用它），并且把所有数据访问事件都包在编程式的事务范围中会很困难（例如在一个桌面应用程序中）。在大多数的其他案例中，自动提交导致系统难以维护，并且没有任何性能或者可伸缩性方面的好处。（依我们之见，RDBMS供应商不应该默认启用自动提交。必要时，SQL查询控制台和工具应该在连接上启用自动提交模式。）

10.4 小结

在本章中，你学习了事务、并发、隔离性和锁。你现在知道Hibernate依赖数据库并发控制机制，但是由于自动版本控制和持久化上下文高速缓存，却在事务中提供了更好的隔离性保证。你学习了如何利用Hibernate API、JTA UserTransaction和JPA EntityTransaction接口编程式地设置事务范围。我们还探讨了包含EJB 3.0组件的事务程序集，以及你如何利用自动提交模式非事务地进行工作。

表10-1显示了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表10-1　第10章中Hibernate和JPA的对照表

Hibernate Core	Java Persistence和EJB 3.0
可以给JDBC和JTA配置`Transaction` API	`EntityTransaction` API只对本地资源的事务有用
Hibernate可以在EJB中被配置为与JTA和容器托管事务整合	利用Java Persistence，在Java SE和Java EE之间，只有数据库连接名称这一配置变化
Hibernate默认通过自动版本控制，为最佳的可伸缩性提供乐观并发控制	Java Persistence通过自动版本控制，标准化了乐观并发控制

现在已经结束了以事务的方式对保存和加载对象所涉及的基础知识的讨论和探索。接下来将通过在用户和应用程序之间创建更为现实的对话，来把所有的知识点串联起来。