第11章 实现对话

本章内容

- 利用Hibernate实现对话
- 利用JPA实现对话
- 利用EJB 3.0组件实现的对话

在前面章节中你已经尝试过这些例子，并在事务内部保存和加载过对象。很可能你已经注意到了五行的代码范例非常有助于你理解一个特定的问题、学习API以及对象如何改变它们的状态。如果你更进一步，试着在自己的应用程序中应用所学过的知识，可能很快意识到你还少了两个重要的概念。

本章要介绍的第一个概念——持久化上下文传播（persistence context propagation），当你必须在应用程序中调用几个类来完成一个特定的动作时非常有用，它们全都需要数据库访问。目前为止，我们只用过一种内部打开和关闭持久化上下文的方法（`Session`或者`EntityManager`）。不用手工传递类和方法之间的持久化上下文，而是要介绍在Hibernate和Java Persistence中可以自动负责传播的机制。Hibernate可以帮助你创建更为复杂的工作单元。

你接下来会遇到的设计问题是，当应用程序的用户必须通过几个屏幕被引导来完成一个工作单元时的应用程序的流程问题。你必须创建代码，控制从一个屏幕到另一个屏幕的导航——但是，这是在持久化的范围之外，本章不对其做过多讨论。持久化机制的部分责任在于，为跨越可能的用户思考时间的工作单元实现数据访问的原子性和隔离性。我们把在客户端/服务器端中完成几个请求并响应的周期的工作单元称为*对话*。Hibernate和Java Persistence提供几种对话实现的策略，本章要介绍如何把这些知识点与现实的示例结合起来。

从Hibernate开始，然后在本章的后半部分讨论JPA对话。让我们先创建更复杂的数据访问示例，看看几个类如何通过自动传播来复用相同的持久化上下文。

11.1 传播 Hibernate `Session`

回忆一下前一章中介绍过的用例：触发拍卖终止的事件必须得到处理（见10.1节）。对于下面的示例，谁触发这个事件并不重要；可能是当自动定时器到达拍卖货品的终止日期和终止时间

时终止其拍卖。也可能是人工的操作员触发了该事件。

为了处理这个事件，你需要执行一系列操作：为拍卖检查胜出的出价，收取拍卖费用，通知卖主和赢家，等等。可以编写有一个大过程的单个类。更好的一种设计是，把这些步骤中每一步的责任都移到可重用的更小组件上，并按关注点把它们分开。第16章会详细讨论这个话题。现在，假设你听取了我们的建议，需要在同一个工作单元内部调用几个类，来处理拍卖的关闭。

11.1.1　Session传播的用例

看一下代码清单11-1中的代码示例，它控制事件的处理。

代码清单11-1　关闭和终止拍卖的控制器代码

```
public class ManageAuction {

    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    public void endAuction(Item item) {

        // Reattach item
        itemDAO.makePersistent(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setSuccessfulBid(winningBid);
        item.setBuyer( winningBid.getBidder() );

        // Charge seller
        Payment payment = new Payment(item);
        paymentDAO.makePersistent(payment);

        // Notify seller and winner
        ...
    }
    ...
}
```

ManageAuction类被称作控制器（controller）。它的任务是协调所有必需的步骤来处理一个特定的事件。当事件触发时，定时器（或者用户界面）调用endAuction()方法。控制器没有包含完成和关闭拍卖必需的所有代码；它尽可能多地委托给其他的类。首先，它需要两个无状态的服务对象（称作数据访问对象）来完成这项工作——它们直接被控制器的每个实例实例化。当endAuction()方法需要访问数据库时，它使用DAO。例如，ItemDAO用来重附脱管的item，并在数据库中查询最高的出价。PaymentDAO用来使一项瞬时的新付款持久化。甚至不需要明白如何通知拍卖的卖主和赢家——你有足够的代码证明需要上下文传播。

代码清单11-1中的代码不能正常工作。首先，没有事务划分。endAuction()中的所有代码都被当作一个原子的工作单元：它要么全部失败，要么全部成功地完成。因此，要把所有这些操作包装在一个事务中。接下来通过不同的API来完成。

更难的问题是持久化上下文。想象ItemDAO和PaymentDAO在每一个（无状态的）方法中使

用不同的持久化上下文。换句话说，itemDAO.getMaxBid()和paymentDAO.makePersistent()都打开、清除并关闭它们自己的持久化上下文（Session或者EntityManager）。这是一种应该永远避免的反模式！在Hibernate中，这就是众所周知的每个操作一个会话，这是检验应用程序设计的性能瓶颈时，作为一名优秀的Hibernate开发人员应该注意的第一件事。单个持久化上下文不应该被用来处理一个特定的操作，而是处理整个事件（它自然可能需要几项操作）。持久化上下文的范围经常与数据库事务的范围相同。这就是众所周知的每个请求一个会话，请见图11-1。

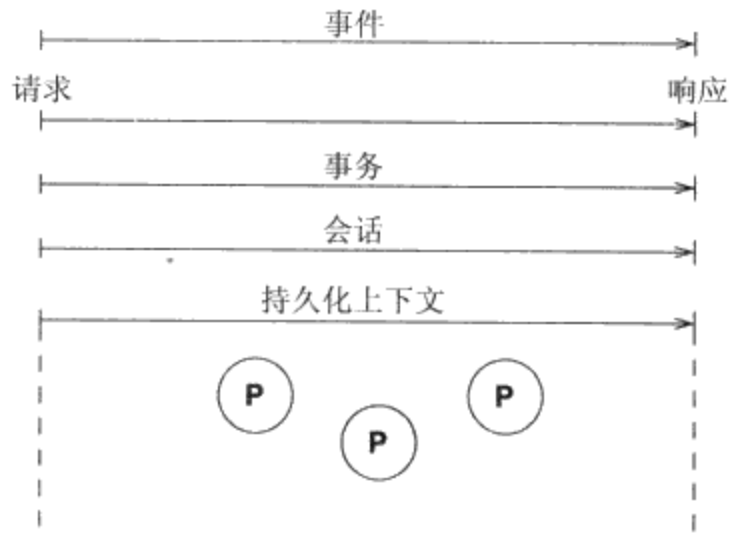

图11-1 单个持久化上下文服务于一个特定的事件

下面来把事务划分添加到ManageAuction控制器，并在数据访问类之间传播持久化上下文。

11.1.2 通过线程局部传播

Hibernate为使用简单Java SE的独立Java应用程序以及任何使用了JTA（不管是否包含EJB）的应用程序，提供了自动的持久化上下文传播。我们极力鼓励你在自己的应用程序中考虑这项特性，因为所有其他的选择方案都没有这个来得完善。

在Hibernate中，可以访问当前的Session来访问数据库。例如，考虑利用Hibernate的ItemDAO实现：

```
public class ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Session s = getSessionFactory().getCurrentSession();
        return (Bid) s.createQuery("...").uniqueResult();
    }
    ...

}
```

方法getSessionFactory()返回全局的SessionFactory。它如何做到这一点则完全取决于你，取决于如何配置和部署应用程序——它可能来自于从JNDI注册中被查找或者当实例化ItemDAO时被手工注入的一个静态变量（HibernateUtil）。这种依赖管理很繁琐；SessionFactory是个线程安全的对象。

SessionFactory中的getCurrentSession()方法是我们要讨论的话题。（PaymentDAO实现

也在所有的方法中使用当前的Session。）当前的Session是什么，当前指的是什么？让我们把事务划分添加到调用ItemDAO和PaymentDAO的控制器中，请见代码清单11-2。

代码清单11-2　把事务划分添加到控制器

```
public class ManageAuction {

    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    public void endAuction(Item item) {
        try {
            // Begin unit of work
            sf.getCurrentSession().beginTransaction();

            // Reattach item
            itemDAO.makePersistent(item);

            // Set winning bid
            Bid winningBid = itemDAO.getMaxBid( item.getId() );
            item.setWinningBid(winningBid);
            item.setBuyer( winningBid.getBidder() );

            // Charge seller
            Payment payment = new Payment(item);
            paymentDAO.makePersistent(payment);

            // Notify seller and winner
            ...

            // End unit of work
            sf.getCurrentSession().getTransaction().commit();

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

工作单元始于调用endAuction()方法的时候。如果在当前的Java线程中第一次调用sessionFactory.getCurrentSession()，就会打开和返回一个新的Session——你得到了新的持久化上下文。你通过Hibernate的Transaction接口（它在Java SE应用程序中转变为JDBC事务），在这个新的Session中立即启动一个数据库事务。

在全局共享的SessionFactory中调用getCurrentSession()的所有数据访问代码，都访问相同的当前Session——如果它在相同的线程中被调用的话。当Transaction被提交（或者回滚）时，工作单元结束。如果提交或者回滚事务，Hibernate也清除和关闭当前的Session以及它的持

久化上下文。这意味着提交或者回滚之后对getCurrentSession()的调用生成了新的Session和新的持久化上下文。

你有效地把相同的范围应用到了数据库事务和持久化上下文。你通常会想要改善这段代码，通过把事务和异常处理移到方法实现之外。一种简单的解决方案是事务拦截器（transaction interceptor），第16章中将编写一个。

Hibernate内部把当前的Session绑定到当前正在运行的Java 线程。[在Hibernate社区中，这也就是大家所知的线程本地会话（ThreadLocal Session）模式]。必须在Hibernate配置中通过把hibernate.current_session_context_class属性设置为thread来启用这个绑定。

如果用JTA部署应用程序，可以启用一种稍微不同的策略，直接把持久化上下文界定和绑定到系统事务。

11.1.3 利用 JTA 传播

在前几节中，我们始终推荐用JTA服务来处理事务，现在重申这个建议。JTA除了提供许多其他的东西之外，还提供用Hibernate接口避免代码污染的标准事务划分接口。代码清单11-3展现了利用JTA重构的ManageAuction控制器。

代码清单11-3 控制器中利用JTA的事务划分

```
public class ManageAuction {
    UserTransaction utx = null;
    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    public ManageAuction() throws NamingException {
        utx = (UserTransaction) new InitialContext()
                .lookup("UserTransaction");
    }

    public void endAuction(Item item) throws Exception {
        try {
            // Begin unit of work
            utx.begin();

            // Reattach item
            itemDAO.makePersistent(item);

            // Set winning bid
            Bid winningBid = itemDAO.getMaxBid( item.getId() );
            item.setWinningBid(winningBid);
            item.setBuyer( winningBid.getBidder() );

            // Charge seller
            Payment payment = new Payment(item);
            paymentDAO.makePersistent(payment);

            // Notify seller and winner
            ...

            // End unit of work
```

```
            utx.commit();

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

这段代码没有任何Hibernate导入。并且更重要的是，内部使用getCurrentSession()的ItemDAO和PaymentDAO类不变。当第一次在其中一个DAO类中调用getCurrentSession()时，新的持久化上下文开始了。当前的Session被自动绑定到当前的JTA系统事务上。事务结束时，通过提交或者回滚，持久化上下文被清除，并且内部绑定的当前Session也关闭了。

与前面没有JTA的示例相比，这段代码中的异常处理略有不同，因为UserTransaction API可能抛出已检查异常（并且在构造器中的JNDI查找也可能失败）。

如果你给JTA配置Hibernate应用程序，就不必启用这个JTA绑定的持久化上下文；getCurrentSession()始终返回一个被界定和绑定到当前JTA系统事务的Session。

（注意，不能把Hibernate的Transaction接口与getCurrentSession()特性和JTA一起使用。你需要Session来调用beginTransaction()，但Session必须被绑定到当前的JTA事务上——这是个鸡和蛋的问题。这再次强调了你应该始终尽可能地使用JTA，而只有在无法使用JTA时才使用Hibernate的Transaction。）

11.1.4　利用 EJB 传播

如果把控制器编写成EJB，并应用容器托管的事务，这段代码（如代码清单11-4所示）甚至会更加简洁。

代码清单11-4　控制器中利用CMT的事务划分

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item) {

        // Reattach item
        itemDAO.makePersistent(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setWinningBid(winningBid);
```

```
            item.setBuyer( winningBid.getBidder() );

            // Charge seller
            Payment payment = new Payment(item);
            paymentDAO.makePersistent(payment);

            // Notify seller and winner
            ...
        }
        ...
    }
```

当前的Session被绑定到为endAuction()方法启动的事务，当这个方法返回时，它被清除和关闭。在这个方法内部运行和调用sessionFactory.getCurrentSession()的所有代码都获得相同的持久化上下文。

如果把这个示例与代码清单11-1中第一个不能工作的示例对比，会发现你只必须添加一些注解使它生效。@TransactionAttribute甚至是可选的——它默认就是REQUIRED。这就是为什么EJB 3.0要提供一个简化的编程模型（simplified programming model）的原因。注意，目前为止，你还没有使用到JPA接口；数据访问类仍然依赖当前的Hibernate Session。稍后你可以轻松地重构它——关注点被清楚地分离了。

你现在知道了如何把持久化上下文界定到事务来服务一个特定的事件，以及如何创建更为复杂的需要在几个对象之间传播和共享持久化上下文的数据访问操作。Hibernate内部使用当前的线程或者当前的JTA系统事务，来绑定当前的Session和持久化上下文。Session和持久化上下文的范围与Hibernate Transaction或者JTA系统事务的范围相同。

我们现在关注第二个设计概念，它可以明显改善设计和创建数据库应用程序的方式。我们将实现长运行的对话，与跨越用户思考时间的应用程序用户的一系列交互。

11.2 利用 Hibernate 的对话

前面的章节已经多次介绍了对话的概念。第一次，我们说对话是跨用户思考时间的工作单元。然后探讨了必须放在一起来创建对话应用程序的基础构建块：脱管对象、重附、合并，以及使用被扩展的持久化上下文的其他可选策略。

现在，该来看看所有这些选项的实战了。我们在前一个示例（拍卖的关闭和完成）上构建，并把它变成对话。

11.2.1 提供对话保证

你已经实现过对话——只是它不长。你实现了最短的可能的对话：跨越来自应用程序用户的单个请求的对话：用户（假设现在指的是人工操作员）在CaveatEmptor管理界面单击Complete Auction（完成拍卖）按钮。然后处理这个被请求的事件，并且把表明事件成功的响应显示给操作人员。

在实践中，短对话很常见。几乎所有的应用程序都有更复杂的对话——更复杂的必须被组合

起来作为一个单元的一系列动作。例如，单击Complete Auction按钮的人工操作员这么做，因为他们确信这次拍卖应该结束。他们根据查看展现在屏幕上的数据做出决定——信息是如何到达那里的？更早的请求被发送到应用程序，并触发拍卖加载用于显示。从应用程序用户的观点来看，这个数据加载是同一个工作单元的一部分。这似乎很合理：应用程序也应该知道这两个事件——用于显示的拍卖项目的加载和拍卖的完成——假定处于同一个工作单元中。我们正在扩大工作单元的概念，并采用应用程序用户的观点。把两个事件都组合到同一个对话中。

应用程序用户希望在经过这个对话时，应用程序能有一些保证：

- 用户将要关闭和终止的拍卖在他们看着它的时候没有被修改。拍卖的完成要求这个决定所基于的数据在完成时仍然不变。否则，操作人员就是在失效的数据上工作，可能会做出错误的决定。
- 对话是原子的：用户可以随时终止对话，并且他们做过的所有变化都被回滚。这在我们当前的场景中不是个大问题，因为只有最后一个事件会进行任何永久的改变；第一个请求只加载用于显示的数据。然而，更为复杂的对话也有可能，且很常见。

作为应用程序的开发人员，你希望通过尽可能少的工作实现这些保证。

现在介绍如何利用Hibernate（无论是否带有EJB）实现长对话。在任何环境下，你必须做的第一项决定都是，要使用利用脱管对象的策略，还是使用扩展持久化上下文的策略。

11.2.2　利用脱管对象的对话

让我们用原生的Hibernate接口和脱管对象策略来创建对话。对话有两个步骤：第一步加载对象，第二步使对被加载的对象所做的变化变成持久化。这两个步骤如图11-2所示。

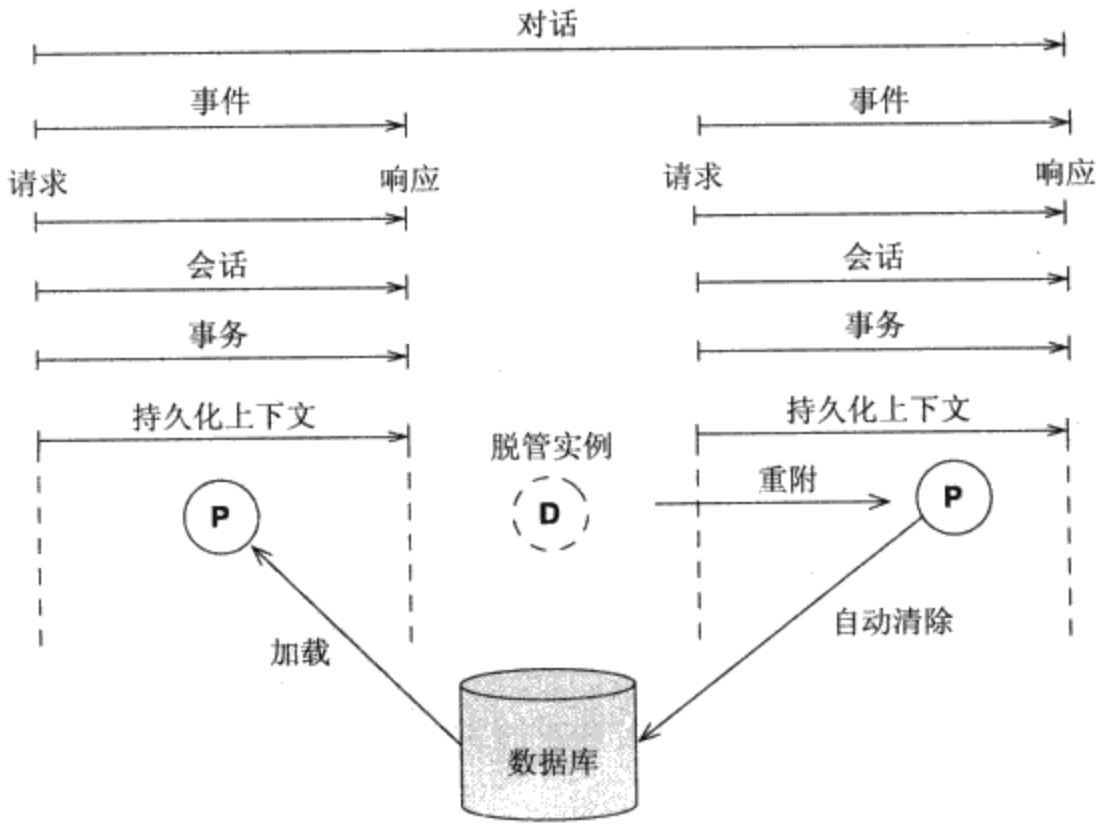

图11-2　利用脱管对象实现的两步对话

第一步需要Hibernate Session来按其标识符获取实例（假设这是个指定的参数）。你将编写另一个可以处理这个事件的ManageAuction控制器：

```
public class ManageAuction {

    public Item getAuction(Long itemId) {
        Session s = sf.getCurrentSession();

        s.beginTransaction();

        Item item = (Item) s.get(Item.class, itemId);

        s.getTransaction().commit();

        return item;
    }
    ...
}
```

对这段代码进行一些简化，避免弄乱例子——你知道异常处理实际上并不是可选的。注意，这比之前介绍的版本更为简单；我们要介绍理解对话所需的最少代码。如果你喜欢，也可以用DAO编写这个控制器。

调用getAuction()的时候，一个新的Session、持久化上下文和数据库事务开始了。对象从数据库中加载，事务提交，并且持久化上下文关闭。现在item对象处于脱管状态，并且返回到调用这个方法的客户端。客户端利用脱管对象，显示它，甚至可能允许用户修改它。

会话中的第二步是完成拍卖。这是ManageAuction控制器中另一种方法的用途。与前面的示例相比，它再次简化了endAuction()方法，避免任何不必要的复杂：

```
public class ManageAuction {

    public Item getAuction(Long itemId) ...
    ...

    public void endAuction(Item item) {
        Session s = sf.getCurrentSession();

        s.beginTransaction();

        // Reattach item
        s.update(item);

        // Set winning bid
        // Charge seller
        // Notify seller and winner
        ...

        s.getTransaction().commit();
    }

}
```

客户端调用endAuction()方法，并传回脱管的item实例——这与第一步中返回的实例相同。Session中的update()操作把脱管对象重附到持久化上下文中，并计划一个SQL UPDATE。Hibernate必须假设客户端在对象被脱管时对它进行了修改。（否则，如果你确定它没有被修改，lock()就足够了。）当对话中的第二个事务提交时，持久化上下文自动被清除，并且对曾经脱管

而现在持久化的对象所做的任何修改都与数据库同步。

在实践中，saveOrUpdate()方法比update()、save()或者lock()更有用：在复杂的对话中，你不知道item是否处于脱管状态，或者它是否为新的、瞬时的、必须保存的。当你不仅在单个实例上工作，而且还想要重附或者持久化被连接的对象图并应用级联选项时，saveOrUpdate()提供的自动状态侦测变得更有用。还要重读merge()操作的定义，以及什么时候使用合并而不是重附：详见9.3.2节。

目前为止，你还只是解决了其中一个对话实现问题：用很少的代码实现对话。然而，应用程序的用户仍然期待工作单元不仅要与并发修改隔离，而且还要有原子性。

你用乐观锁把并发的对话隔离开来。一般来说，不应该应用跨越长运行对话的悲观并发控制策略——这意味着昂贵和不可伸缩的锁定。换句话说，不能防止两个操作人员看见同一件拍卖货品。你希望这种情况很少发生：你是乐观的。但是如果发生了，你也有一种冲突的解决策略。需要给Item持久化类启用Hibernate自动的版本控制，就像在10.2.2节中所做的那样。然后，在对话期间的任何时候，每一个SQL UPDATE或者DELETE都将包括一个针对数据库中所展现的状态的版本检查。如果这个检查失败，就得到StaleObjectStateException，然后必须采取适当的行动。在这个例子中，你给用户展现了一条错误消息（“对不起，有人修改了同一件拍卖品！”），并强制从第一步开始重启对话。

如何使对话变成原子化呢？对话跨越几个持久化上下文和几个数据库事务。但是从应用程序用户的观点来看，这并不是工作单元的范围；他们认为对话会是一个原子的操作组合，要么全部失败，要么全部成功。在当前的对话中，这不成问题。因为你只在最后一步（第二步）中修改和持久化数据。任何只读取数据，并把被修改对象的重附延迟到最后一步的对话，都自动成为原子化，可以随时终止。如果对话在一个中间的步骤中把修改重附和提交到数据库，它就不再是原子的。

一种解决方案是不在提交时清除持久化上下文——也就是说，在一个假定不持久化修改的Session中设置FlushMode.MANUAL（当然，不是针对对话的最后一步）。另一种方法是使用补偿（compensation）动作，撤销（undo）任何进行永久改变的步骤，并在用户终止会话时调用适当的补偿动作。我们不过多讨论编写补偿动作；它们取决于你正在实现的对话。

接下来用不同的策略实现相同的对话，消除脱管的对象状态。你扩展持久化上下文为跨整个对话。

11.2.3　给对话扩展 Session

Hibernate Session有一个内部的持久化上下文。通过扩展持久化上下文来跨整个对话，你可以实现一个不涉及脱管对象的对话。这就是大家所知的每个对话一个会话（session-per-conversation）策略，如图11-3所示。

在对话开始的时候打开新的Session和持久化上下文。第一步，Item对象的加载在第一个数据库事务中实现。一旦提交数据库事务，Session就立即自动地从底层的JDBC Connection中断开。现在你可以在用户思考时间的过程中保持这个断开连接的Session和它内部的持久化上下

文。一旦用户继续对话，并执行下一个步骤，你就立即通过启动第二个数据库事务把Session重新连接到新的JDBC Connection。已经在这个对话中加载的任何对象都处于持久化状态：永远不会脱管。因而，一旦你在Session中调用flush()，对任何持久化对象所做的修改就立即被清除到数据库。必须通过设置FlushMode.MANUAL来禁用Session的自动清除——你应该在对话启动和Session打开的时候完成这项工作。

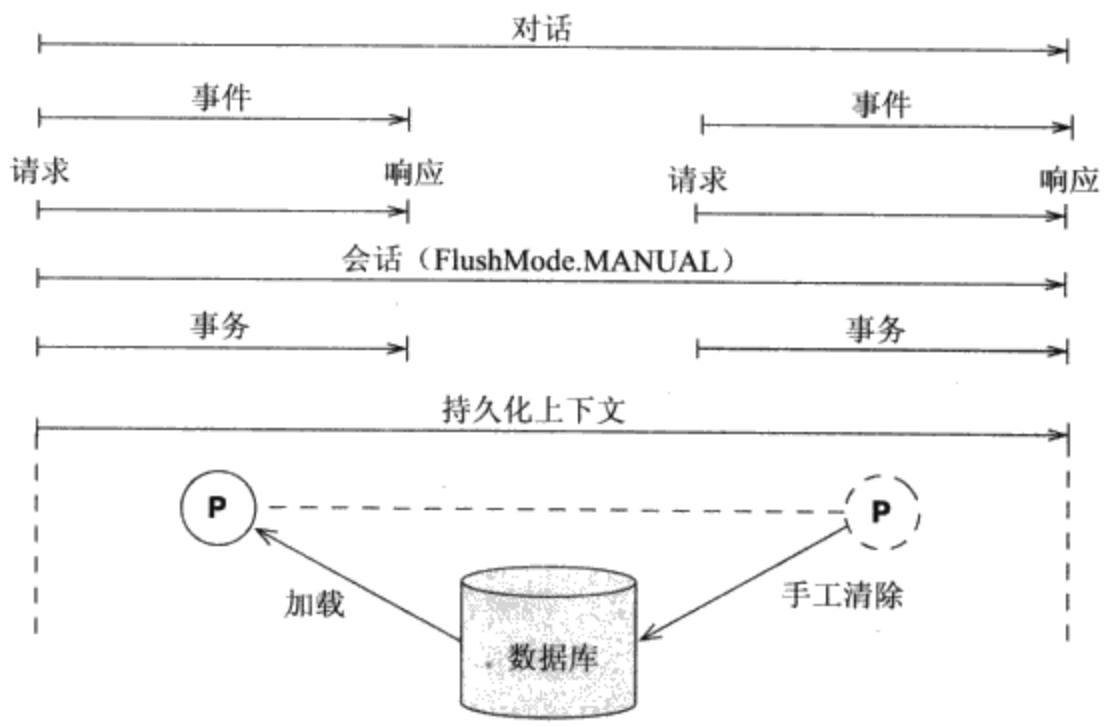

图11-3 断开连接的持久化上下文扩展为跨整个对话

在清除的期间，由于乐观锁和Hibernate的自动版本检查，在并发对话中所做的修改被隔离。如果你直到最后一步即对话结束时才清除Session，对话的原子性就得到了保证——如果关闭未清除的Session，就是有效地终止了对话。

需要详细阐述这种行为的一个例外：新实体实例的插入时间。注意，这在本例中不成问题，但它是在更复杂的对话中必须处理的东西。

1. 延迟到清除时间插入

为了理解这个问题，回想一种保存对象的方法，以及分配它们标识符值的方式。因为你没有在Complete Auction的对话中保存任何新对象，因此还没有遇到过这个问题。但是，在中间步骤中保存对象的任何对话都不可能有原子性。

Session中的save()方法要求必须返回被保存的实例的新数据库标识符。因此，标识符值必须在调用save()方法的时候生成。对于大多数标识符生成器策略来说，这不成问题；例如，Hibernate可以调用sequence，在内存中进行increment，或者向hilo生成器要一个新值。Hibernate不一定要执行SQL INSERT来在save()上返回标识符值，并把它分配到目前持久化的实例。

例外的是INSERT发生之后触发的标识符生成策略。其中一个是identity，另一个是select；两者都要求先插入一个行。如果通过这些标识符生成器映射持久化类，调用save()时就立即执行INSERT！因为你正在对话期间提交数据库事务，这个插入可能有永久的效果。

看看下列稍微不同的对话代码，它示范了这种效果：

```
Session session = getSessionFactory().openSession();
session.setFlushMode(FlushMode.MANUAL);

// First step in the conversation
session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(123) );
session.getTransaction().commit();

// Second step in the conversation
session.beginTransaction();
Item newItem = new Item();
Long newId = (Long) session.save(newItem); // Triggers INSERT!
session.getTransaction().commit();

// Roll back the conversation!
session.close();
```

你可以期待整个对话（两个步骤）可以通过关闭未清除的持久化上下文进行回滚。newItem的插入应该被延迟到你在Session上调用flush()的时候，这在本代码中永远不会发生。这是个只有在没有选择identity或者select作为标识符生成器时才会发生的情况。利用这些生成器，INSERT必须在对话的第二步中执行，并被提交到数据库。

除了关闭未被清除的持久化上下文之外，一种解决方案是使用补偿动作，执行它们用来撤销在被终止的对话期间所做的任何可能的插入。你必须手工删除插入的行。另一种解决方案是用不同的标识符生成器，如sequence，支持新标识符值的生成而不用插入。

persist()操作让你面临了相同的问题。然而，它还提供了另一种可供选择（更好）的解决方案。它可以延迟插入，如果在事务之外调用它，甚至可以用事后插入（post-insert）标识符生成：

```
Session session = getSessionFactory().openSession();
session.setFlushMode(FlushMode.MANUAL);

// First step in the conversation
session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(1));
session.getTransaction().commit();

// Second step in the conversation
Item newItem = new Item();
session.persist(newItem);

// Roll back the conversation!
session.close();
```

persist()方法可以延迟插入，因为它不必返回标识符值。注意，newItem实体在你调用persist()之后处于持久化状态，但是如果你用identity或者select生成器策略映射持久化类，它就没有分配到标识符值。当在清除时发生INSERT时，标识符值就被分配到实例。在事务之外调用persist()时，没有执行任何SQL语句。newItem对象只是在列队中等待插入。

记住，我们已经讨论的这个问题取决于所选择的标识符生成器策略——你可能不会遇上它，或者可能避免它。persist()的非事务行为在本章稍后再次变得重要，当你用JPA而不是Hibernate接口编写对话的时候。

让我们先用一个被扩展的Session完成对话的实现。利用“每个对话一个会话”策略，不再需要在代码中手工脱管和重附（或者合并）对象。必须给整个对话实现可以重用同一个Session的基础代码。

2. 管理当前的Session

前面讨论过的当前Session支持是一种可转换的机制。你已经见过两种可能的内部策略：一种是线程绑定，另一种把当前的Session绑定到JTA事务。但是，这两者都在事务结束时关闭Session。需要一个不同的Session范围用于“每个对话一个会话”的模式，但是你仍然想要能够在应用程序的代码中访问当前的Session。

第三种内建的选项正是你使用“每个对话一个会话”策略需要的东西。必须通过把hibernate.current_session_context_class配置选项设置为managed来启用它。已经讨论过的其他内建选项是thread和jta，如果你给JTA部署配置Hibernate，后者就会被隐式地启用。注意，所有这些内建的选项都是org.hibernate.context.CurrentSessionContext接口的实现；可以编写自己的实现，并在配置中命名该类。这通常没有必要，因为内建的选项涵盖了大多数的情况。

你刚刚启用的Hibernate内建实现称作托管，因为它把管理范围（启动和终止当前的Session）委托给你。你用3种静态的方法管理Session的范围：

```
public class ManagedSessionContext implements CurrentSessionContext {
    public static Session bind(Session session) { ... }
    public static Session unbind(SessionFactory factory) { ... }
    public static boolean hasBind(SessionFactory factory) { ... }
}
```

你可能已经猜到了“每个对话一个会话”策略必须做什么：

- 当对话启动时，必须用ManageSessionContext.bind()打开和绑定一个新的Session，来服务对话中的第一个请求。你还必须在这个新的Session中设置FlushMode.MANUAL，因为你不想在背后发生任何持久化上下文的同步。
- 现在调用SessionFactory.getCurrentSession()的所有数据访问代码都接收到了你所绑定的Session。
- 当对话中的请求完成时，你需要调用ManagedSessionContext.unbind()，并在某处保存现在断开连接的Session，直到对话中有了下一个请求。或者，如果这是对话中的最后一个请求，你需要清除和关闭Session。

所有这些步骤都可以在拦截器（interceptor）中实现。

3. 创建对话拦截器

你需要一个为了响应对话中的每个请求而被自动触发的拦截器。如果使用EJB（像你很快要做的那样），就可以免费获得许多这种基础结构代码。如果你编写非Java EE的应用程序，就必须编写自己的拦截器。编写拦截器有许多种方法；我们介绍一种只示范概念的抽象拦截器。可以在CaveatEmptor下载文件中的org.hibernate.ce.auction.web.filter包中找到Web应用程序有效的、已经通过测试的拦截器实现。

假设每当对话中的事件必须被处理时，拦截器就运行起来。还假设每个事件都必须通过前端控制器和它的execute()动作方法——最容易的场景。现在可以把这个方法包装在拦截器中；也就是说，编写一个在这个方法执行之前和之后调用的拦截器。如图11-4所示；从左到右阅读编了号的项目。

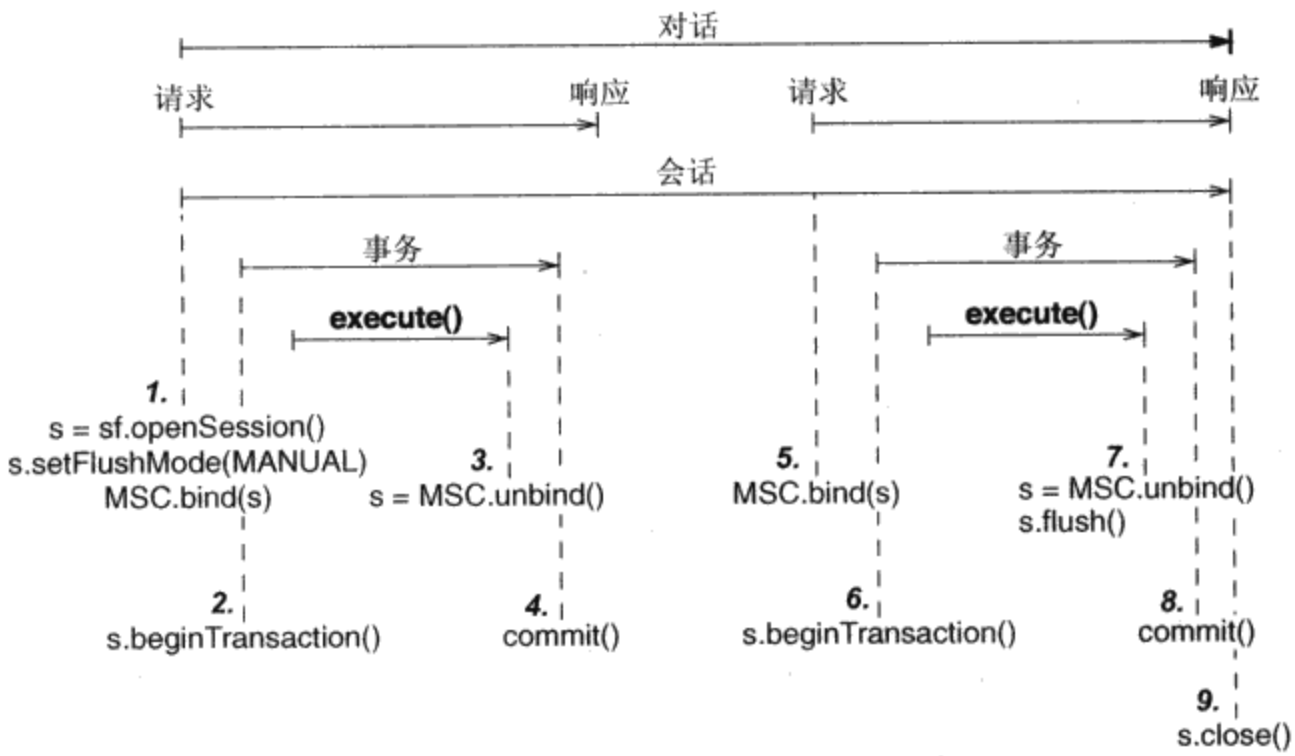

图11-4　拦截事件来管理Session的生命周期

当对话中的第一个请求命中服务器，拦截器运行并打开新的Session❶；这个Session的自动清除立即被禁用。然后这个Session被绑定到Hibernate的ManagedSessionContext。在拦截器让控制器处理事件之前启动事务❷。在这个控制器（或者任何被该控制器调用的DAO）内部运行的所有代码现在都可以调用SessionFactory.getCurrentSession()，并使用Session。当控制器完成它的工作时，拦截器再次运行，并解绑当前的Session❸。事务提交❹之后，Session自动断开连接，并且可以在用户思考时间内保存。

现在，服务器等待对话中的第二个请求。

一旦第二个请求命中服务器，拦截器就立即运行，侦测到有被保存的断开连接的Session，并把它绑定到ManagedSessionContext❺中。控制器在事务被拦截器❻启动之后处理事件。当控制器完成它的工作时，拦截器再次运行，并从Hibernate中解绑当前的Session。然而，不是断开连接并保存它，而是拦截器现在侦测到这是对话的终点，且Session需要在提交事务之前❽被清除❼。最后，对话结束，拦截器关闭Session❾。

这听起来比在代码中更为复杂。代码清单11-5是这种拦截器的一个伪实现：

代码清单11-5　拦截器实现“每个对话一个会话”策略

```
public class ConversationInterceptor {

    public Object invoke(Method method) {
```

```
// Which Session to use?
Session currentSession = null;

if (disconnectedSession == null) {
    // Start of a new conversation
    currentSession = sessionFactory.openSession();
    currentSession.setFlushMode(FlushMode.MANUAL);
} else {
    // In the middle of a conversation
    currentSession = disconnectedSession;
}

// Bind before processing event
ManagedSessionContext.bind(currentSession);

// Begin a database transaction, reconnects Session
currentSession.beginTransaction();

// Process the event by invoking the wrapped execute()
Object returnValue = method.invoke();

// Unbind after processing the event
currentSession =
    ManagedSessionContext.unbind(sessionFactory);

// Decide if this was the last event in the conversation
if ( returnValue.containsEndOfConversationToken() ) {

    // The event was the last event: flush, commit, close
    currentSession.flush();
    currentSession.getTransaction().commit();
    currentSession.close();

    disconnectedSession = null; // Clean up

} else {

    // Event was not the last event, continue conversation
    currentSession.getTransaction().commit(); // Disconnects
    disconnectedSession = currentSession;
}

return returnValue;
    }
}
```

invoke(Method)拦截器把控制器的execute()操作包装起来。每当必须处理来自应用程序的用户的请求时，这段拦截代码就会运行。当它返回时，你检查返回值是否包含一个特殊的标记或者记号。这个标记意味着这是特定对话中必须被处理的最后一个事件。现在清除Session，提交所有变化，并关闭Session。如果这不是对话的最后一个事件，你就提交数据库事务，保存断开连接的Session，并继续等待对话中的下一个事件。

这个拦截器对于调用execute()的任何客户端代码都是透明的。它对于在execute()内部运行的任何代码也是透明的：任何数据访问操作都使用当前的Session；关注点被恰当地分离。我们甚至不必展示数据访问代码，因为它不用任何数据库事务划分或者Session处理。只是利用

getCurrentSession()加载和保存对象。

你心里可能会有下列疑问：

- 当应用程序等待用户发送对话的下一个请求时，disconnectSession保存在哪里？它可以保存在HttpSession中甚至有状态的EJB中。如果没有使用EJB，这个责任就委托给了应用程序的代码。如果使用EJB 3.0和JPA，你可以把持久化上下文的范围（Session的一个等价物）绑定到一个有状态的EJB——简化编程模型的另一个优势。
- 标志对话结束的特殊标记来自哪里？在我们的抽象示例中，这个标记出现在execute()方法的返回值中。把这样的特殊信号实现到拦截器有多种方法，只要你找到一种方法把它传到那里。把它放在事件处理的结果中是一种务实的解决方案。

这样就结束了我们对利用Hibernate的持久化上下文传播和对话实现的讨论。前面的小节缩短和简化了一些示例，让你更容易理解概念。如果想用Hibernate继续实现更加复杂的工作单元，建议也先读一下第16章。

另一方面，如果没有使用Hibernate API，却想要使用Java Persistence和EJB 3.0组件，就接着往下读。

11.3　使用 JPA 的对话

现在看看利用JPA和EJB 3.0进行的持久化上下文传播和对话实现。就像使用原生的Hibernate一样，当你想要利用Java Persistence实现对话时，必须考虑三点：

- 你想要传播持久化上下文，以便持久化上下文可以在特定的请求中用于所有数据访问。在Hibernate中，这项功能内建在getCurrentSession()特性中。如果JPA在Java SE中独立部署，它就没有这项特性。另一方面，由于EJB 3.0编程模型和预设好的范围，以及事务与托管组件的生命周期，与EJB结合的JPA比原生的Hibernate更为强大。
- 如果你决定使用脱管的对象方法作为你的对话实现策略，就要使得对脱管对象的改变成为持久化。Hibernate提供重附和合并；JPA只支持合并。前一章曾深入讨论过其中的区别，但是我们想要通过更为现实的对话示例简要地对它进行重述。
- 如果决定用“每个对话一个会话”方法作为对话实现策略，就要扩展持久化上下文来跨整个对话。我们看一下JPA持久化上下文范围，并探讨如何在Java SE中使用JPA和EJB组件来实现被扩展的持久化上下文。

注意，必须再次在两种不同的环境中处理JPA：在简单的Java SE中，以及通过EJB在Java EE环境中。阅读本节时，可以对其中一个更加关注。我们之前通过先讨论上下文传播，然后讨论长对话，探讨了这个使用Hibernate的对话主题。使用JPA和EJB 3.0我们将同时探讨这两种，但是把Java SE和Java EE放在单独的小节中。

先在一个没有任何托管组件或者容器的Java SE应用程序中，通过JPA实现对话。我们经常会指出原生的Hibernate对话之间的区别，以便确定你理解了本章前面几节的内容。下面讨论之前提出过的3个问题：持久化上下文传播、脱管实例的合并和被扩展的持久化上下文。

11.3.1 Java SE 中的持久化上下文传播

再次考虑代码清单11-1中的控制器。这段代码依赖执行持久化操作的DAO。这里再现了这种包含Hibernate API的数据访问对象的实现：

```
public class ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Session s = getSessionFactory().getCurrentSession();
        return (Bid) s.createQuery("...").uniqueResult();
    }
    ...

}
```

如果试图通过JPA重构，唯一的选择似乎只能是这样：

```
public class ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Bid maxBid;
        EntityManager em = null;
        EntityTransaction tx = null;
        try {
            em  = getEntityManagerFactory().createEntityManager();
            tx = em.getTransaction();
            tx.begin();

            maxBid = (Bid) em.createQuery("...")
                             .getSingleResult();
            tx.commit();
        } finally {
            em.close();
        }
        return maxBid;
    }
    ...

}
```

如果在Java SE中应用程序在它自身上处理EntityManager，JPA中就没有定义任何持久化上下文传播。在Hibernate SessionFactory中，没有getCurrentSession()方法的等价物。

在Java SE中获得EntityManager的唯一方法是在工厂中利用createEntityManager()方法来实例化。也就是说，数据访问的所有方法使用它们自己的EntityManager实例——这就是我们前面指出过的“每个操作一个会话”的反模式！更糟糕的是，对于跨几个数据访问操作的事务划分，并没有明显的位置。

对于这个问题，有3种可能的解决方案：

- 可以在创建DAO时给整个DAO实例化EntityManager。这不会让你获得每个请求一个持久化上下文（persistence-context-per-request）范围，但是比“每个操作一个持久化上下文”略好一点。但是，事务划分仍然是这种策略的一个问题；所有DAO中的所有DAO操作都无法组合成为一个原子的、隔离的工作单元。

- 可以在控制器中实例化单个EntityManager，并在创建DAO（构造器注入）时，把它传递到所有的DAO里面。这样就解决了问题。处理EntityManager的代码可以与单个位置中的事务划分代码（即控制器）配对。
- 可以在拦截器中实例化单个EntityManager，并把它绑定到辅助类中的ThreadLocal变量。DAO从ThreadLocal中获取当前的EntityManager。这种策略模拟Hibernate中的getCurrentSession()功能。拦截器也可以包括事务划分，并且可以把控制器方法包装在拦截器中。不用自己编写这个基础结构代码，可以首先考虑EJB。

我们把在Java SE中要对持久化上下文传播使用哪种策略的决定权留给你。建议考虑Java EE组件、EJB，然后就可以使用强大的上下文传播了。可以轻松地给应用程序部署轻量化的EJB容器，就像在2.2.3节中所做的那样。

让我们进行第二个问题：长对话中脱管实例的修改。

11.3.2　在对话中合并脱管对象

我们已经详细阐述了脱管对象的概念，以及如何把被修改的实例重附到新的持久化上下文，或者另一种方法是，把它们合并到新的持久化上下文。因为JPA只给合并提供持久化操作，所以有必要复习一下有关利用原生的Hibernate代码合并的示例和注释（在9.3.2节中），以及9.4.2节对JPA中脱管对象的讨论。

在这里我们想要关注之前提出过的一个问题，并且从一个稍微不同的视角来看待它。这个问题就是：为什么要从merge()操作中返回持久化的实例？

你之前用Hibernate实现的长对话有两个步骤（两个事件）。在第一个事件中，获取一件拍卖货品用于显示。在第二个事件中，（可能被修改过的）货品被重附到一个新的持久化上下文中，然后关闭拍卖。

代码清单11-6展现了相同的控制器，它可以通过JPA和合并，服务于两个事件：

代码清单11-6　利用JPA合并脱管对象的控制器

```
public class ManageAuction {

    public Item getAuction(Long itemId) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        Item item = em.find(Item.class, itemId);

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

        return item;
    }

    public Item endAuction(Item item) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
```

```
        tx.begin();

        // Merge item
        Item mergedItem = em.merge(item);

        // Set winning bid
        // Charge seller
        // Notify seller and winner
        // ... this code uses mergedItem!

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

        return mergedItem;
    }

}
```

这里应该没有令你感到惊讶的代码——你已经多次见过所有这些操作。考虑调用这个控制器的客户端，它通常是某种表现代码。首先，调用getAuction()方法，获取用于显示的Item实例。不久，触发第二个事件，并调用endAuction()方法。脱管的Item实例被传递到这个方法中；但是该方法也返回一个Item实例。返回的Item、mergeItem是不同的实例！客户端现在有两个Item对象：旧的和新的。

就像我们在9.3.2节中指出的，对旧实例的引用应该被客户端认为是过时的：它不表示最新的状态。只有mergeItem才是对最新状态的引用。使用合并（而不是重附），丢弃对失效对象的过时引用变成是客户端的责任。如果考虑下列客户端代码，这通常不成问题：

```
ManageAuction controller = new ManageAuction();

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
item = controller.endAuction(item);
```

最后一行代码设置了合并结果为item变量值，因此你有效地通过新引用更新了这个变量。记住，这一行只更新这个变量。表现层中仍然有对旧实例引用的任何其他代码也必须刷新变量——要多加小心。这实际上意味着表现代码必须知道重附和合并这两种策略之间的区别。

我们发现，通过被扩展的持久化上下文（extended persistence context）策略构造的应用程序经常比严重依赖脱管对象的应用程序更加容易理解。

11.3.3 在 Java SE 中扩展持久化上下文

我们已经在10.1.3节中讨论过，在Java SE中使用JPA的持久化上下文的范围。现在详细阐述这些基础的知识，并主要关注包含对话实现的一个被扩展的持久化上下文示例。

1. 默认的持久化上下文范围

在没有EJB的JPA中，持久化上下文被绑定到EntityManager实例的生命周期和范围。为了在对

话中给所有的事件重用相同的持久化上下文，只需要重用同一个EntityManager去处理所有事件。

一种简单的方法是把这个责任委托给对话控制器的客户端：

```
public static class ManageAuctionExtended {

    EntityManager em;

    public ManageAuctionExtended(EntityManager em) {
        this.em = em;
    }

    public Item getAuction(Long itemId) {
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        Item item = em.find(Item.class, itemId);

        tx.commit();

        return item;
    }

    public Item endAuction(Item item) {
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        // Merge item
        Item mergedItem = em.merge(item);

        // Set winning bid
        // Charge seller
        // Notify seller and winner
        // ... this code uses mergedItem!

        tx.commit();

        return mergedItem;
    }
}
```

控制器期待在它的构造器中给整个对话设置持久化上下文。客户端现在创建和关闭EntityManager：

```
// Begin persistence context and conversation
EntityManager em = emf.createEntityManager();

ManageAuctionExtended controller = new ManageAuctionExtended(em);

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
controller.endAuction(item);

// End persistence context and conversation
em.close();
```

自然地，包装有getAuction()和endAuction()方法并提供正确EntityManager实例的拦截器可能更加方便。它还避免了关注点向上渗透到表现层。如果把控制器编写成有状态的EJB会话bean，就可以轻松地获得这个拦截器。

当你试图通过跨整个对话的被扩展持久化上下文应用这种策略时，或许会遇到一个可能破坏对话原子性的问题——自动清除。

2. 防止自动清除

考虑下列对话，它添加了一个事件作为中间步骤：

```
// Begin persistence context and conversation
EntityManager em = emf.createEntityManager();

ManageAuctionExtended controller = new ManageAuctionExtended(em);

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
if ( !controller.sellerHasEnoughMoney(seller) )
    throw new RuntimeException("Seller can't afford it!");

// Third event
controller.endAuction(item);

// End persistence context and conversation
em.close();
```

看看这个新的对话客户端代码，你认为更新过的货品描述什么时候被保存在数据库中？这取决于持久化上下文的清除。你知道JPA中默认的FlushMode是AUTO，它在执行查询之前和提交事务时启用同步。对话的原子性取决于sellerHasEnoughMoney()方法的实现，以及它是执行查询还是提交事务。

假设你通过一般的事务块，把在该方法内部执行的操作包装起来：

```
public class ManageAuctionExtended {
    ...

    public boolean sellerHasEnoughMoney(User seller) {
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();

        tx.commit();

        return sellerCanAffordIt;
    }

    ...
}
```

这个代码片段甚至包括两个调用，它们触发EntityManager持久化上下文的清除。首先，FlushMode.AUTO意味着查询的执行触发了一次清除。其次，事务提交触发了另一次清除。这显

然不是你想要的——你想要使整个对话原子化，并在最后一个事件完成之前防止任何清除。

Hibernate提供org.hibernate.FlushMode.MANUAL，它从同步中解耦事务划分。不幸的是，由于JSR-220专家组的成员意见不一，javax.persistence.FlushMode只提供AUTO和COMMIT。在介绍“正式”的解决方案之前，先看看下列代码是如何通过退回到Hibernate API来获得FlushMode.MANUAL的：

```
// Prepare Hibernate-specific EntityManager parameters
Map params = new HashMap();
params.put("org.hibernate.flushMode," "MANUAL");

// Begin persistence context with custom parameters
EntityManager em = emf.createEntityManager(params);

// Alternative: Fall back and disable automatic flushing
((org.hibernate.Session)em.getDelegate())
   .setFlushMode(org.hibernate.FlushMode.MANUAL);

// Begin conversation
ManageAuction controller = new ManageAuction(em);

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
if ( !controller.sellerHasEnoughMoney(seller) )
    throw new RuntimeException("Seller can't afford it!");

// Third event
controller.endAuction(item);

// End persistence context and conversation
em.close();
```

别忘了em.flush()必须在第三个事件的最后一个事务中被手工调用，否则就没有任何修改会变成持久化：

```
public static class ManageAuctionExtended {
    ...
    public Item endAuction(Item item) {
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        // Merge item
        ...
        // Set winning bid
        ...

        em.flush(); // Commit the conversation

        tx.commit();

        return mergedItem;
    }
}
```

正式的架构解决方案依赖非事务的行为。不是用简单的FlushMode设置，而是需要给数据访问操作编写没有事务范围的代码。专家组成员对于漏掉的FlushMode给出的其中一个原因是，“事务提交应该使所有修改永久化”。因此，你只能通过移除事务划分，给对话中的第二步禁用清除：

```
public class ManageAuction {
    ...
    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select ...").getSingleResult();
        return sellerCanAffordIt;
    }
    ...
}
```

这段代码没有触发持久化上下文的清除，因为EntityManager是在任何事务范围之外使用的。执行这个查询的EntityManager现在正在自动提交模式下工作，包含我们先前在10.3节中涵盖的所有值得关注的结果。更糟糕的是，你失去了拥有可重复读取的能力：如果同一个查询执行两次，每次查询都在自动提交模式下在它们自己的数据库连接上执行。它们可能返回不同的结果，因此数据库事务隔离级别可重复读取（repeatable read）和可序列化（serializable）都不起作用。换句话说，利用正式的解决方案，你无法获得可重复读取的数据库事务隔离性，并同时禁用自动清除。持久化上下文高速缓存可以只对实体查询（而不对标量查询）提供可重复读取。

如果用JPA实现对话，强烈建议你考虑Hibernate的FlushMode.MANUAL设置。我们也期待这个问题将在规范的未来版本中得到解决；（几乎）所有的JPA供应商都已经包括一个专有的清除模式设置，效果与org.hibernate.FlushMode.MANUAL的一样。

你现在知道如何通过脱管的实体实例和被扩展的持久化上下文编写JPA对话了。前几节为下一步打下了基础：利用JPA和EJB的对话实现。如果现在觉得JPA比Hibernate更麻烦，那么相信你定会惊讶于一旦引入EJB后的对话实现却变得非常容易。

11.4 使用 EJB 3.0 的对话

必须再看一下问题清单：持久化上下文传播、脱管对象的处理和跨整个对话的被扩展的持久化上下文。这一次，你将在其中增加EJB。

我们将不过多讨论脱管的实体实例，以及如何在对话中合并持久化上下文之间的修改——要用到的概念和API与Java SE中和利用EJB时的完全相同。

另一方面，当你引入EJB，然后依赖标准的上下文传播规则以及JPA与EJB3.0编程模型的整合时，持久化上下文传播和利用JPA的扩展的持久化上下文管理将变得更为容易。

先关注EJB调用中的持久化上下文传播。

11.4.1 使用 EJB 的上下文传播

JPA和EJB 3.0定义持久化上下文在应用程序中如何处理，以及当几个类（或者EJB组件）使用一个EntityManager时所应用的规则。使用EJB的应用程序中最常见的案例是容器托管的且被

注入的EntityManager。你通过注解，把ItemDAO变成托管的无状态EJB组件，并重写代码来使用EntityManager：

```
@Stateless
public class ItemDAOBean implements ItemDAO {

    @PersistenceContext
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Bid getMaxBid(Long itemId) {
        return (Bid) em.createQuery("...").getSingleResult();
    }
    ...

}
```

当这个bean的客户端调用getMaxBid()时，EJB容器就注入EntityManager。这个EntityManager的持久化上下文是当前的持久化上下文（很快会做更详细的讨论）。如果调用getMaxBid()时没有事务在处理，getMaxBid()返回时就会启动新的事务并提交。

说明　许多开发人员过去没有对使用EJB 2.1的DAO类使用EJB会话bean。在EJB 3.0中，所有的组件都是简单的Java对象，你确实应该通过几个简单的注解（如果你不喜欢注解，就用XML部署描述符）来获得容器的服务。

1. 连接EJB组件

既然ItemDAO是EJB组件（如果你沿用前面利用Hibernate实现对话的例子，别忘了还要重构PaymentDAO），那么可以通过依赖注入把它装配到同样被重构的ManageAuction组件，并把整个操作包在单个事务中：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @EJB
    ItemDAO itemDAO;

    @EJB
    PaymentDAO paymentDAO;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item endAuction(Item item) {

        // Merge item
        itemDAO.makePersistent(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setWinningBid(winningBid);
        item.setBuyer( winningBid.getBidder() );

        // Charge seller
        Payment payment = new Payment(item);
        paymentDAO.makePersistent(payment);
```

```
            // Notify seller and winner
            ...
            return item;
        }
        ...
    }
```

EJB容器基于你用@EJB声明的字段，注入了想要的组件，该接口具名ItemDAO和PaymentDAO这一信息足以使容器找到必需的组件。

让我们来关注事务和应用到这个组件装配的持久化上下文传播规则。

2. 传播规则

首先，系统事务是必需的，如果客户端调用ManageAuction.endAuction()，就会启动系统事务。当这个方法返回时，事务因此被容器提交。这是系统事务的范围。被调用的以及需求或者支持事务（比DAO方法）的任何其他无状态的组件方法，都继承相同的事务上下文。如果在任何这些无状态的组件中使用EntityManager，那么你正在使用的持久化上下文就自动相同，它们都被界定到系统事务。注意，这不同于你在Java SE应用程序中使用JPA的情况：EntityManager实例定义持久化上下文的范围（我们前面也详细阐述过这一点）。

当ItemDAO和PaymentDAO，这两个无状态的组件都在系统事务内部被调用时，两者都继承界定到事务的持久化上下文。容器通过当前的持久化上下文，在幕后把EntityManager实例注入到itemDAO和paymentDAO。

（在内部，如果客户端获得ManageAuction控制器，容器就从它的无状态bean池中抓取一个闲置的ManageAuctionBean实例，注入一个闲置的无状态ItemDAOBean和PaymentDAOBean，在所有需要它的组件上设置这个持久化上下文，并把ManageAuction bean句柄返回到客户端用于方法调用。这当然有点被简化了。）

对于持久化上下文的范围和传播，有正式的规则：

- 如果容器提供的（通过注入或者通过查找获得的）EntityManager第一次被调用，持久化上下文就开始了。默认情况下，它是事务范围的，并在系统事务提交或者回滚时关闭。它在事务提交时被自动清除。
- 如果容器提供的（通过注入或者通过查找获得的）EntityManager第一次被调用，持久化上下文就开始了。如果此时没有系统事务是活动的，持久化上下文就很短，并只服务于单独的方法调用。被任何这样一种方法触发的任何SQL都在自动提交模式下在一个数据库连接中执行。在这个EntityManager调用中（可能）获取的所有实体实例都立即变成脱管状态。
- 如果无状态的组件（如ItemDAO）被调用，并且调用者有一个活动的事务并被传播到被调用的组件（因为ItemDAO方法需要或者支持事务），绑定到JTA事务的任何持久化上下文就通过事务传播。
- 如果无状态的组件（如ItemDAO）被调用，且调用者没有活动的事务（例如ManageAuction.endAuction()没有启动事务），或者事务没有传播到被调用的组件（因为ItemDAO方法不需要或者不支持事务），当在无状态的组件内部调用EntityManager时，就创建了新的持久化上下文。换句话说，如果没有事务被传播，就不会发生持久化上下文的传播。

如果只阅读正式的定义，这些规则看起来就很复杂；但是，在实践中，它们变成了一种自然的行为。持久化上下文被自动界定和绑定到JTA系统事务，如果有的话——你只需要学习事务传播的规则，了解持久化上下文如何被传播。如果没有JTA系统事务，持久化上下文就服务于单个的EntityManager调用。

目前为止，你在几乎所有的例子中都使用了TransactionAttributeType.REQUIRED。这是在事务装配中应用的最常见的属性；毕竟，EJB是事务处理的编程模型。我们只介绍过一次TransactionAttributeType.NOT_SUPPORTED，是在10.3.3节讨论利用Hibernate Session的非事务数据访问的时候。

还要记住，在JPA中，你需要用非事务的数据访问来禁用长对话中持久化上下文的自动清除——还是被漏掉的FlushMode.MANUAL的问题。

现在要更深入地探讨事务属性类型，以及如何通过EJB和被扩展的持久化上下文的手工清除来实现对话。

11.4.2　利用 EJB 扩展持久化上下文

在前一节中，你只利用了被界定到JTA系统事务的持久化上下文。容器自动被注入了EntityManager，并且透明地处理持久化上下文的清除和关闭。

如果想要利用EJB和被扩展的持久化上下文实现一个对话，有两种选择：

- 可以编写一个有状态的会话bean作为对话控制器。持久化上下文可以被自动界定到这个有状态bean的生命周期，这是一种方便的方法。当这个有状态的EJB被删除时，持久化上下文就关闭。
- 可以用EntityManagerFactory自己创建EntityManager。这个EntityManager的持久化上下文是应用程序托管的——你必须手工清除和关闭它。如果在JTA事务范围内部调用它，还必须用joinTransaction()操作通知EntityManager。使用有状态的会话bean的第一种策略通常更好。

就像以前在Java SE中实现的对话一样：必须把三个步骤当作一个原子化的工作单元来完成：获取一件拍卖货品用于显示和修改，卖主账号的流动资金检查，以及最终关闭拍卖。

必须再次决定对话期间想要如何禁用被扩展的持久化上下文的自动清除，以保持原子性。可以在这两者中选择：利用FlushMode.MANUAL的Hibernate供应商扩展，以及利用正式的非事务的操作方法。

1．通过Hibernate扩展来禁用清除

先用比较容易的Hibernate扩展编写一个有状态的EJB作为对话控制器：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(
      type = PersistenceContextType.EXTENDED,
      properties = @PersistenceProperty(
                    name="org.hibernate.flushMode",
```

```
                    value="MANUAL")
    )
    EntityManager em;

    public Item getAuction(Long itemId) {
        return em.find(Item.class, itemId);
    }

    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();
        return sellerCanAffordIt;
    }

    @Remove
    public void endAuction(Item item, User buyer) {
        // Set winning bid
        // Charge seller
        // Notify seller and winner
        item.setBuyer(...);

        em.flush();
    }
}
```

这个bean实现了ManageAuction接口（我们没有必要介绍这个接口）的三种方法。首先，这是个有状态的EJB；容器给特定的客户端创建和保存一个实例。当客户端第一次获得这个EJB的句柄时，创建一个新实例，容器注入新的被扩展的持久化上下文。持久化上下文现在被绑定到EJB实例的生命周期，并在标记为@Remove的方法返回时关闭。注意你如何像读对话的历史一样一步一步地读EJB的方法。可以用@Remove注解几个方法；例如，可以添加cancel()方法来撤销所有的对话步骤。对于对话来说，这是个强大的、方便的编程模型，EJB 3.0中全部免费内建了。

接下来是自动清除的问题。ManageAuctionBean的所有方法都需要事务；在类级别上需要声明这一点。sellerHasEnoughMoney()方法（对话中的第二步），在执行查询之前清除持久化上下文，并在这个方法的事务返回时再次清除。为了防止这种情况，需要声明被注入的持久化上下文应该在FlushMode.MANUAL这个Hibernate扩展中。现在，每当你想要把列队中等待的SQL DML编写到数据库时，清除持久化上下文就是你的责任了——你只在对话结束时进行一次。

事务装配现在从持久化引擎的清除行为中解耦出来了。

2. 通过禁用事务来禁用清除

根据EJB 3.0规范，正式的解决方案混合了这两个关注点。通过使对话的所有步骤（除了最后一个之外）变成非事务来防止自动的清除：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    EntityManager em;

    public Item getAuction(Long itemId) {
        return em.find(Item.class, itemId);
```

```
    }

    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();
        return sellerCanAffordIt;
    }

    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item, User buyer) {
        // Set winning bid
        // Charge seller
        // Notify seller and winner
        item.setBuyer(...);
    }
}
```

在这个实现中，你转换到一个用于所有方法的不同默认：TransactionAttributeType.NOT_SUPPORTED，并且只有endAuction()方法需要事务。这最后一个事务也在提交时清除持久化上下文。

现在不用事务调用EntityManager的所有方法都在自动提交模式（第10章讨论过）下有效地运行着。

3. 复杂的事务装配

你现在已经使用过了几个不同的TransactionAttributeType注解，请见表11-1中可用选项的完整清单。

最常用的事务属性类型是REQUIRED，这是所有无状态和有状态EJB方法的默认值。为了对有状态会话bean中的一个方法禁用被扩展的持久化上下文的自动清除，要转换到NOT_SUPPORTED甚至NEVER。

表11-1　EJB 3.0声明式事务属性类型

属性名称	描　述
REQUIRED	必须通过事务上下文调用的一种方法。如果客户端没有事务上下文，容器就会启动事务，并获取这个事务所用的所有资源（数据源等）。如果这个方法调用其他的事务组件，事务就会被传播。方法返回时，容器在结果被发送到客户端之前提交事务
NOT_SUPPORTED	如果方法在从客户端传播的事务上下文内部调用，调用者的事务就被暂停，并在方法返回时重新激活。如果调用者没有事务上下文，就没有给该方法启动任何事务。不是全部所用的资源都通过事务得到获取（发生自动提交）
SUPPORTS	如果方法在从客户端传播的事务上下文内部调用，它就把这个事务上下文与跟REQUIRED相同的结果联结起来。如果调用者没有事务上下文，就没有事务被启动，结果与NOT_SUPPORTED的相同。这个事务属性类型应该只用于可以正确处理这两种情况的方法
REQUIRES_NEW	方法始终在新的事务上下文内部执行，结果和行为与使用REQUIRED的相同。任何被传播的客户端事务都被暂停，并在方法返回且新事务完成时继续
MANDATORY	方法必须通过一个活动的事务上下文被调用。然后它联结这个事务上下文，并且在需要时进一步传播。如果调用时没有出现事务上下文，就抛出异常
NEVER	它与MANDATORY相反。如果方法通过一个活动的事务上下文被调用，就抛出异常

当利用有状态的EJB设计对话时，或者如果你想要混合无状态和有状态的组件时，必须知道事务上下文和持久化上下文的传播规则：

- 如果具有被扩展的持久化上下文的有状态会话bean调用（实际上实例化）了另一个也有持久化上下文的有状态会话bean，第二个有状态会话bean就从调用者处继承持久化上下文。持久化上下文的生命周期由第一个有状态会话bean决定；它在两个会话bean都被移除时关闭。如果涉及更多有状态的会话bean，这个行为就是递归的。这个行为还不依赖于任何事务规则和事务传播。
- 如果EntityManager用于有着绑定的被扩展持久化上下文的有状态会话bean的一个方法中，并且这个方法需要/支持客户端的JTA事务，那么如果方法的调用程序也通过它的事务传播不同的持久化上下文，就会出现异常。（这是一个罕见的设计问题。）

这些规则隐含的意思是：想象有状态的ManageAuction控制器没有直接调用EntityManager，但是它委托给其他的组件（例如数据访问对象）。它仍然有并且负责被扩展的持久化上下文，虽然EntityManager从不被直接使用。这个持久化上下文必须被传播到所有其他被调用的组件（例如，ItemDAO和PaymentDAO）里面。

如果把DAO实现为无状态的会话bean，就像你之前所做的那样，那么如果它们从有状态的控制器的一个非事务的方法中被调用，它们就不继承被扩展的持久化上下文。这又是有状态的调用DAO的控制器：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    EntityManager em;

    @EJB
    PaymentDAO paymentDAO;

    public boolean sellerHasEnoughMoney(User seller) {
        return paymentDAO.checkLiquidity(seller);
    }

    ...
}
```

sellerHashEnoughMoney()方法没有启动事务，以便在对话中间的提交中自动避免持久化上下文的清除。问题是对DAO这个无状态EJB的调用。为了让持久化上下文传播到无状态EJB调用中，你需要传播一个事务上下文。如果paymentDAO.checkLiquidity()使用EntityManager，它就得到了新的持久化上下文！

第二个问题在于PaymentDAO无状态会话bean：

```
@Stateless
public class PaymentDAO {

    @PersistenceContext
    EntityManager em;
    public boolean checkLiquidity(User u) {
```

```
        boolean hasMoney = (Boolean)
            em.createQuery("select...").getSingleResult();
        return hasMoney;
    }
    ...
}
```

由于调用checkLiquidity()方法时，没有持久化上下文被传播到它里面去，因此就会创建新的持久化上下文用来服务于这个单独的操作。这是“每个操作一个会话”的反模式！更糟糕的是，现在你在一个请求和这个对话中有两个（或者更多）持久化上下文，并将遇到数据别名的问题（没有同一性范围保证）。

如果你把DAO实现为有状态的会话bean，它们就从调用有状态会话bean控制器中继承持久化上下文。在这种情况下，持久化上下文就通过实例化（而不是通过事务传播）而被传播。

如果把控制器写成一个有状态的会话bean，就把DAO写成有状态的EJB。这个问题是遗漏的FlushMode.MANUAL另一个讨厌的副作用，它会严重影响到你的设计方式和分层应用程序。建议你在EJB 3.0（或者3.1?）规范修正之前，都依赖Hibernate扩展。通过FlushMode.MANUAL，控制器不一定使用TransactionAttributeType.NOT_SUPPORTED，并且持久化上下文始终与事务一起传播（还可以轻松地混合无状态和有状态的EJB调用）。

我们会在第16章回到这个问题，那时我们将编写更加复杂的应用程序代码和DAO。

11.5　小结

在本章中，你利用Hibernate、JPA和EJB 3.0组件实现了对话。你学习了如何传播当前的Hibernate Session和持久化上下文来创建更加复杂的分层应用程序，而不渗透关注点。你也已经知道持久化上下文传播是EJB 3.0深入整合的一项特性，以及持久化上下文可以被轻松地绑定到JTA（或者CMT）事务范围。你还学习了FlushMode.MANUAL这项Hibernate特性如何独立地从事务装配中禁用持久化上下文的清除。

表11-2展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表11-2　第11章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
持久化上下文传播在Java SE和Java EE中可以通过线程或者JTA事务的绑定而可用。持久化上下文要么被界定到事务，要么由应用程序托管	Java Persistence只对Java EE标准化了持久化上下文传播模型（它与EJB 3.0组件进行了深入整合）。很好地定义了持久化上下文范围到事务或者有状态会话bean
Hibernate支持利用脱管对象实现对话，这些对象可以在对话期间被重附或者合并	Java Persistence标准化了脱管对象的合并，但是不支持重附
Hibernate支持用FlushMode.MANUAL选项对长对话禁用持久化上下文的自动清除	禁用被扩展的持久化上下文的自动清除，需要非事务的事件处理（对应用程序的设计和分层有严格的限制）或者需要Hibernate退回到FlushMode.MANUAL

第12章将探讨每当你要利用更复杂、更大的数据集时应该依赖的各种选项。你会了解到传播性持久化如何利用Hibernate的级联模型、如何有效地执行批量和大批量操作，以及在加载和保存对象时如何钩入和操作Hibernate的默认行为。