

创建和测试分层的应用程序

本章内容

- 创建分层的应用程序
- 托管的组件和服务
- 集成测试的策略

Hibernate准备只用于任何可以想象的架构场景。Hibernate可以在servlet容器内部运行，可以把它与Web应用程序框架（如Struts、WebWork或者Tapestry）一起使用，或者用在EJB容器内部，或者用在Java Swing应用程序中管理持久化数据。

甚至——尤其可能——由于所有这些选项，理解Hibernate应该如何整合到一个特定的基于Java的架构经常是有困难的。你将不可避免地需要编写基础结构代码来支持自己的应用程序的设计。本章阐述常用的Java架构，并介绍Hibernate如何整合到每一种场景中。

我们讨论如何在一个基于Web应用程序的典型的请求/响应中设计和创建分层，以及如何按功能分离代码。之后，介绍Java EE服务和EJB，并介绍托管组件如何使你的生活更加轻松，减少了没有托管组件时必需的基础结构编码。

最后，假设你也关注测试分层应用程序（无论是否包含托管组件）。如今，在开发人员的工作中，测试已经成了最重要的活动之一，并且应用正确的工具和策略是提高工作效率的根本（更不用说软件的质量了）。我们讨论并利用当前最受欢迎的测试框架TestNG进行单元、功能和集成测试。

从一个典型的Web应用程序示例开始。

16.1 Web 应用程序中的 Hibernate

第1章已经强调过规则的应用程序分层的重要性。分层有助于实现关注点的分离，通过把完成类似工作的代码分组，使得代码更易读。然而，分层要付出一些代价。每增加一层，都要增加由于实现一块简单功能而花费的代码量——而代码越多就越难以改变功能。

本节介绍如何在一个典型的分层应用程序中整合Hibernate。假设你想要用Java servlet编写一个简单的Web应用程序。我们需要CaveatEmptor应用程序的一个简单用例，来示范这些思想。

16.1.1 用例简介

当用户对一件货品进行出价时，CaveatEmptor必须在单个请求中执行下列任务：

- (1) 查找大于该货品现有出价的最大金额的用户输入金额。
- (2) 查找还没有结束的拍卖。
- (3) 给该货品创建一个出价。
- (4) 通知任务结果的用户。

如果其中任何一项查找失败，应该告知客户原因；如果两项查找都成功，则应该通知用户出价已经成功。这些查找是业务规则。如果访问数据库的时候出现失败，应该通知用户，告知目前系统不可用（一个基础结构关注点）。

下面来看如何在Web应用程序中实现这一点。

16.1.2 编写控制器

大多数Java Web应用程序使用某种MVC（Model/View/Controller，模型/视图/控制器）应用程序框架；甚至使用简单servlet的许多应用程序也遵循MVC模式，通过利用模板来实现展现代码，把应用程序控制逻辑分离到一个servlet或者多个servlet。

你现在就要编写一个这样的控制器servlet，它实现前面介绍过的用例。利用MVC方法，编写在一个具名PlaceBidAction的动作的execute()方法中实现“出价”用例的代码。假设某种Web框架，我们不探讨如何读取请求参数或者如何转到下一页。所显示的代码甚至可能是一个简单servlet的doPost()方法的实现。

如代码清单16-1所示，编写这种控制器的第一步是在一个地方混合所有的关注点——没有分层。

代码清单16-1 在一个execute()方法中实现一个用例

```
public void execute() {
    Long itemId = ...           // Get value from request
    Long userId = ...           // Get value from request
    BigDecimal bidAmount = ...   // Get value from request
    Transaction tx = null;

    try {
        Session session = ①
            HibernateUtil.getSessionFactory().getCurrentSession();
        tx = session.beginTransaction();

        // Load requested Item
        Item item = (Item) session.load(Item.class, itemId); ②

        // Check auction still valid
        if (item.getEndDate().before(new Date()) ) { ③
            ... // Forward to error page
        }
    }
}
```

```

// Check amount of Bid ④
Query q = session.createQuery("select max(b.amount) " +
    " from Bid b where b.item = :item");
q.setEntity("item", item);
BigDecimal maxBidAmount = (BigDecimal) q.uniqueResult();
if (maxBidAmount.compareTo(bidAmount) > 0) {
    ... // Forward to error page
}

// Add new Bid to Item
User bidder = (User) session.load(User.class, userId); ⑤
Bid newBid = new Bid(bidAmount, item, bidder);
item.addBid(newBid);

... // Place new Bid into request context ⑥
tx.commit(); ⑦
... // Forward to success page

} catch (RuntimeException ex) { ⑧
    if (tx != null) tx.rollback();
    throw ex;
}
}

```

① 利用当前的持久化上下文得到一个Session，然后启动一个数据库事务。2.1.3节中已介绍过HibernateUtil类，并在11.1节中讨论过持久化上下文范围。在当前的会话中启动一个新的数据库事务。

② 利用Item的标识符值从数据库加载它。

③ 如果拍卖的终止日期比当前的日期早，就转到一个错误页面。通常，对于这种带有限定错误消息的异常你会想要一个更为成熟的错误处理。

④ 利用HQL查询，查找数据库中是否有对当前货品所做的更高出价。如果有，就转到一个错误页面。

⑤ 如果所有的查找都成功，就将它添加到该货品，对其出一个新价。不必手工保存它——利用传播性持久化保存它（从Item级联到Bid）。

⑥ 新的Bid实例需要被保存在可以被后面的页面访问的某些变量中，这样可以把它显示给用户。可以为此在servlet请求上下文中使用一个属性。

⑦ 提交数据库事务，将Session的当前状态清除到数据库，并自动关闭当前的Session。

⑧ 如果抛出任何RuntimeException，就通过Hibernate或者其他服务来回滚事务，并重新抛出要在控制器外部进行适当处理的异常。

这段代码的第一个问题在于由所有事务和异常处理代码造成的混乱。因为这段代码对所有动作都是典型地一致，你想要将它集中在某个地方。一种方法是把它放在某些动作的抽象超类的execute()方法里面。延迟初始化(lazy initialization)也有问题，如果在success页面中访问新的出价，把它拖出请求上下文进行渲染：关闭Hibernate持久化上下文，你再也无法延迟加载集合或者

代理。

让我们开始整理这个设计，并引入分层。第一步是通过实现OSIV（Open Session In View）模式，在success页面中启用延迟加载。

16.1.3 OSIV 模式

OSIV模式背后的动机在于，视图通过导航从某个脱管对象开始的对象图，从业务对象中拖出信息——例如，放在请求上下文中的刚由动作创建的Bid实例。视图（即，必须被渲染和显示的页面）访问这个脱管对象，给页面获得内容数据。

在Hibernate应用程序中，可能有未被初始化的关联（代理或者集合），它们必须在渲染视图时被驳回。在这个例子中，视图可以通过调用newBid.getBidder().getItems().iterator()，列出被出价人出售的所有货品（作为摘要屏的一部分）。这种情况很罕见，但确定是一个有效的访问。因为User的items集合只按需加载（Hibernate的延迟关联和集合默认行为），它在此时没有被初始化。无法加载未被初始化的代理和处于脱管状态的实体实例的集合。

如果Hibernate Session和相应的持久化上下文始终在动作的execute()方法结束时关闭，Hibernate就会在这个未加载的关联（或者集合）被访问时，抛出LazyInitializationException。持久化上下文不再可用，因此Hibernate无法在访问中加载延迟集合。

常见问题 如果Hibernate必须延迟加载对象，为什么它不能打开新的会话？Hibernate Session是持久化上下文，是对象同一性的范围。Hibernate保证在持久化上下文中，一个特定的数据库行最多只有一种内存表示法。按需打开Session，幕后也创建了一个新的持久化上下文，并且在这个同一性范围内加载的所有对象，都将可能与在原始的持久化上下文中加载的对象产生冲突。当对象处于受保护的对象同一性范围之外（即被脱管的时候），你将无法按需加载数据。另一方面，只要对象处于持久化状态中，且由Session管理，那么甚至在原始的事务已经被提交的时候，你都可以加载数据。在这样的场景中，必须启用自动提交模式，如10.3节中所述。建议你不要在Web应用程序中使用自动提交模式，把原始的Session和事务扩展为跨越整个请求更为容易。当对象必须在一个Session内部按需加载时，在你无法轻易启动和终止事务的系统中（例如使用Hibernate的Swing桌面应用程序），自动提交模式就很有用。

最初的解决方案是确保所有必需的关联和集合在转到视图之前被完全初始化（我们稍后讨论它），但是，在一个包含协同定位的表现层和持久层的两层架构中，一种更为方便的方法是保持持久化上下文开着，直到视图完全呈现。

OSIV模式允许你每一个请求都有一个的Hibernate持久化上下文，跨越视图的呈现，且可能跨越多个动作执行execute()。它也可以被轻松地实现——例如，通过servlet过滤器：

```
public class HibernateSessionRequestFilter implements Filter {
    private SessionFactory sf;
    private static Log log = ...;
    public void doFilter(ServletRequest request,
```

```

        ServletResponse response,
        FilterChain chain)
    throws IOException, ServletException {

try {
    // Starting a database transaction
    sf.getCurrentSession().beginTransaction();

    // Call the next filter (continue request processing)
    chain.doFilter(request, response);

    // Commit the database transaction
    sf.getCurrentSession().getTransaction().commit();

} catch (Throwable ex) {
    // Rollback only
    try {
        if (sf.getCurrentSession().getTransaction().isActive())
            sf.getCurrentSession().getTransaction().rollback();
    } catch (Throwable rbEx) {
        log.error("Could not rollback after exception!", rbEx);
        rbEx.printStackTrace();
    }

    // Let others handle it...
    throw new ServletException(ex);
}

}

public void init(FilterConfig filterConfig)
    throws ServletException {
    sf = HibernateUtil.getSessionFactory();
}

public void destroy() {}
}

```

这个过滤器就像servlet请求的一个拦截器。每次请求命中服务器且必须被处理时它都运行，它在启动时需要SessionFactory，并从HibernateUtil辅助类中得到它。当请求到达时，你启动一个数据库事务，并打开新的持久化上下文。你在控制器已经执行且视图已经呈现之后，提交数据库事务。由于Hibernate的自动Session绑定和传播，这也自动成为持久化上下文的范围。

异常处理也被集中和封装在这个拦截器中。至于要给数据库事务的回滚捕捉什么异常，就取决于你了； Throwable是捕捉全部（catch-all）的变形，它意味着不仅Exception和RuntimeException，甚至抛出的Error也会触发回滚。注意实际的回滚也可能抛出错误或者异常——始终确保（例如，通过打印出堆栈轨迹）这个次要的异常不会隐藏或者淹没导致回滚的原始问题。

控制器代码现在脱离了事务和异常处理，看起来已经好多了：

```

public void execute() {

    // Get values from request

```

```

Session session =
    HibernateUtil.getSessionFactory().getCurrentSession();
// Load requested Item
// Check auction still valid
// Check amount of Bid
// Add new Bid to Item
// Place new Bid in scope for next page
// Forward to success page
}

```

由SessionFactory返回的当前Session，与现在界定到包住这个方法（以及结果页面的呈现）的拦截器的持久化上下文一样。

参考你Web容器的文档，看看如何对特定的URL将这个过滤器类启用为拦截器；建议你只把它应用到执行期间需要数据库访问的URL。否则，就会在服务器中给每一个HTTP请求启动数据库事务和Hibernate Session。这可能会耗尽你的数据库连接池，即使没有把任何SQL语句发送到数据库服务器。

可以用你喜欢的任何方法来实现这个模式，只要有能力拦截请求，并把控制器包装在代码中。许多Web框架提供原生的拦截器，应该使用你所发现的最具吸引力的拦截器。此处介绍利用servlet过滤器的实现并非没有问题。

对Session中的对象所做的改变被不定期地清除到数据库，并且在事务提交时被最终清除。事务提交可能发生在视图呈现之后。问题是servlet引擎的缓冲区大小：如果视图的内容超出缓冲区大小，缓冲区就可能被清除，并且内容被发送到客户端。当内容呈现时，缓冲区可能被多次清除，但是第一次清除也发送HTTP协议状况代码。如果Hibernate清除/提交时的SQL语句在数据库中触发了约束违例，用户可能已经看到成功的输出！无法改变状况代码（例如，用500 Internal Server Error），它已经被发送到客户端（如200 OK）。

有几种方法可以防止这种罕见的异常：调整servlet的缓冲区大小，或者在转发（forward）/重定向（redirect）到视图之前清除Session。有些Web框架没有立即用呈现的内容填充响应缓冲区——它们使用自己的缓冲区，并且只通过视图完全呈现之后的响应清除它，因此我们认为这是简单的Java servlet编程的问题。

让我们继续整理控制器并把业务逻辑抽取到业务层。

16.1.4 设计巧妙的领域模型

MVC模式背后的思想就是控制逻辑（在示例应用程序中，这是一个页面流逻辑）、视图定义，以及业务逻辑应该完全分离。目前，控制器包含一些业务逻辑——在应用程序获得新用户界面这样一个公认不可能的事件中你可能重用的代码——并且领域模型由哑数据持有（dumb data-holding）对象组成。持久化类定义状态，但是不定义行为。

建议你把业务逻辑移到领域模型中，创建一个业务层。这个层的API是领域模型API。这样增加了几行代码，但是它也增加了以后重用的可能，并且更加面向对象，因此提供各种扩展业务逻辑的方法[例如，如果你突然需要实现“最低的出价胜出”，就可以对不同的出价策略使用一个

策略模式 (strategy pattern)]。也可以独立于页面流或者任何其他关注点来测试业务逻辑。

首先，把新方法placeBid()添加到Item类：

```
public class Item {
    ...

    public Bid placeBid(User bidder, BigDecimal bidAmount,
                        Bid currentMaxBid, Bid currentMinBid)
    throws BusinessException {
        // Check highest bid (TODO:Strategy pattern?)
        if (currentMaxBid != null &&
            currentMaxBid.getAmount().compareTo(bidAmount) > 0) {
            throw new BusinessException("Bid too low.");
        }

        // Auction still valid
        if (this.getEndDate().before(new Date()) )
            throw new BusinessException("Auction already ended");

        // Create new Bid
        Bid newBid = new Bid(bidAmount, this, bidder);
        // Place bid for this Item
        this.addBid(newBid);

        return newBid;
    }
}
```

这段代码基本上执行了所有需要业务对象的状态、但不执行数据访问代码的查找。目的在于把业务逻辑封装在领域模型的类中，而不用依赖任何持久化数据访问或者其他的基础结构。记住，这些类不应该知道关于持久化的任何东西，因为你可能在持久化上下文外部需要它们（例如，在表现层或者逻辑单元测试中）。

把代码从控制器移到领域模型，有一个值得注意的异常。来自旧控制器的代码不能像这样移动：

```
// Check amount of Bid
Query q = session.createQuery("select max(b.amount) " +
                             " from Bid b where b.item = :item");
q.setEntity("item", item);
BigDecimal maxBidAmount = (BigDecimal) q.uniqueResult();
if (maxBidAmount.compareTo(bidAmount) > 0) {
    ... // Forward to error page
}
```

在实际的应用程序中，你经常会面对同样的情形：业务逻辑与数据访问代码甚至页面流逻辑混合在一起。有时候难以只抽取业务逻辑而不用任何依赖。如果你现在看一下解决方案，在Item.placeBid()方法上引入的currentMaxBid和currentMinBid参数，就会明白如何解决这种问题。页面流和数据访问代码保留在控制器中，但是给业务逻辑提供必要的数据：

```
public void execute() {
    Long itemId = ...           // Get value from request
    Long userId = ...           // Get value from request
```

```

BigDecimal bidAmount = ... // Get value from request

Session session =
    HibernateUtil.getSessionFactory().getCurrentSession();

// Load requested Item
Item item = (Item) session.load(Item.class, itemId);

// Get maximum and minimum bids for this Item
Query q = session.getNamedQuery(QUERY_MAXBID);
q.setParameter("itemid", itemId);
Bid currentMaxBid = (Bid) q.uniqueResult();

q = session.getNamedQuery(QUERY_MINBID);
q.setParameter("itemid", itemId);
Bid currentMinBid = (Bid) q.uniqueResult();

// Load bidder
User bidder = (User) session.load(User.class, userId);

try {
    Bid newBid = item.placeBid(bidder,
                                bidAmount,
                                currentMaxBid,
                                currentMinBid);

    ... // Place new Bid into request context
    ... // Forward to success page
} catch (BusinessException e) {
    ... // Forward to appropriate error page
}
}

```

控制器现在完全不知道任何业务逻辑——它甚至不知道新的出价必须比上一个价格更高还是更低。你已经把所有的业务逻辑都封装在领域模型中，现在可以把业务逻辑作为一个孤立的单元进行测试了，它不依赖于任何动作、页面流、持久化，或者其他的基础结构代码（通过在单元测试中调用 `Item.placeBid()`）。

甚至可以通过捕捉和转发特定的异常设计一个不同的页面流。`BusinessException`是一个已声明的和已检查的异常，因此你必须以某种方式在控制器中对它进行处理。在这种情况下是否要回滚事务，或者是否有机会以某种方式恢复，就取决于你了。然而，处理异常时始终要考虑你持久化上下文的状态：当你在应用程序异常之后重用同一个 `Session` 时，可能有来自之前的未被清除的修改试图出现。（当然，可以永远不重用已经抛出致命的运行时异常的 `Session`。）安全的做法是始终回滚任何异常中的数据库事务，并用新的 `Session` 重试。

这个动作代码看起来已经很好了。你应该努力保持架构简单；隔离异常和事务处理并且抽取业务逻辑，这么做会有很大的区别。然而，动作代码现在被绑定到了 `Hibernate`，因为它用 `Session` API 访问数据库。`MVC` 模型没有提到 `Persistence` 的 `P` 应该何去何从。

16.2 创建持久层

将数据访问代码和应用程序逻辑混合，这违背了关注点分离的重点。之所以你应该考虑把Hibernate调用隐藏在facade（即所谓的持久层）之后，有几个原因：

- 持久层可以给数据访问操作提供更高级别的抽象。不用基础的CRUD和查询操作，而是公开更高级别的操作，例如getMaximumBid()方法。这个抽象就是你为什么想要在更大的应用程序中创建持久层的主要原因：支持重用相同的非CRUD操作。
- 持久层可以有泛型的接口，不公开实际的实现细节。换句话说，可以隐瞒你正在使用Hibernate（或者Java Persistence）从持久层的任何客户端实现数据访问操作的事实。我们认为持久层的可移植性是一个不重要的关注点，因为完全的ORM解决方案（如Hibernate）已经提供了数据库的可移植性。你不太可能在未来使用不同的软件重新编写持久层，而仍然不想改变任何客户端代码。此外，把Java Persistence当作标准的、可以完全移植的API。
- 持久层可以统一数据访问操作。这个关注点与可移植性相关，但是从一个稍微不同的角度来看。想象你必须处理混合的数据访问代码，例如Hibernate和JDBC操作。通过统一客户端看到和使用的facade，你可以从客户端隐藏这个实现细节。

如果把可移植性和统一当作创建持久层的附带作用，主要目的在于实现更高级别的抽象和改进后的可维护性，以及数据访问代码的重用。这些都是很好的理由，我们鼓励创建这样的持久层：除了最简单的应用程序之外，全部用一般的facade。系统不要过于细致（overengineer），并且首先考虑直接使用Hibernate（或者JPA）而不用任何其他分层，这一点仍然很重要。假设你想要创建一个持久层，并设计一个客户端将调用的facade。

设计持久层Facade的方法不止一种——有些小应用程序可以使用单个PersistenceManager对象；有些可以使用某种面向命令的设计，其他的则把数据访问操作混合到领域类（活动的记录）——但我们更喜欢DAO模式。

16.2.1 泛型的数据访问对象模式

DAO设计模式源于Sun公司的Java Blueprint。它甚至用在臭名昭著的Java Petstore演示应用程序中。DAO为与特定的持久化实体相关的持久化操作（CRUD和finder方法）定义了接口，它建议你把与该实体的持久化相关的代码都组合到一起。

使用JDK 5.0特性如泛型（generic）和变量参数，可以轻松地设计一个很好的DAO持久层。这里建议的模式基本结构如图16-1所示。

我们设计了包含两个平行层次结构的持久层：接口在一侧，实现在另一侧。基本的对象存储和对象获取操作被分在泛型的超接口（generic superinterface），以及由于利用特定的持久化解决方案（我们用Hibernate）而实现这些操作的超类中。泛型接口通过接口进行扩展，用于需要额外的业务相关的数据访问操作的特定实体。你可能再次拥有实体DAO接口的一个或者几个实现。

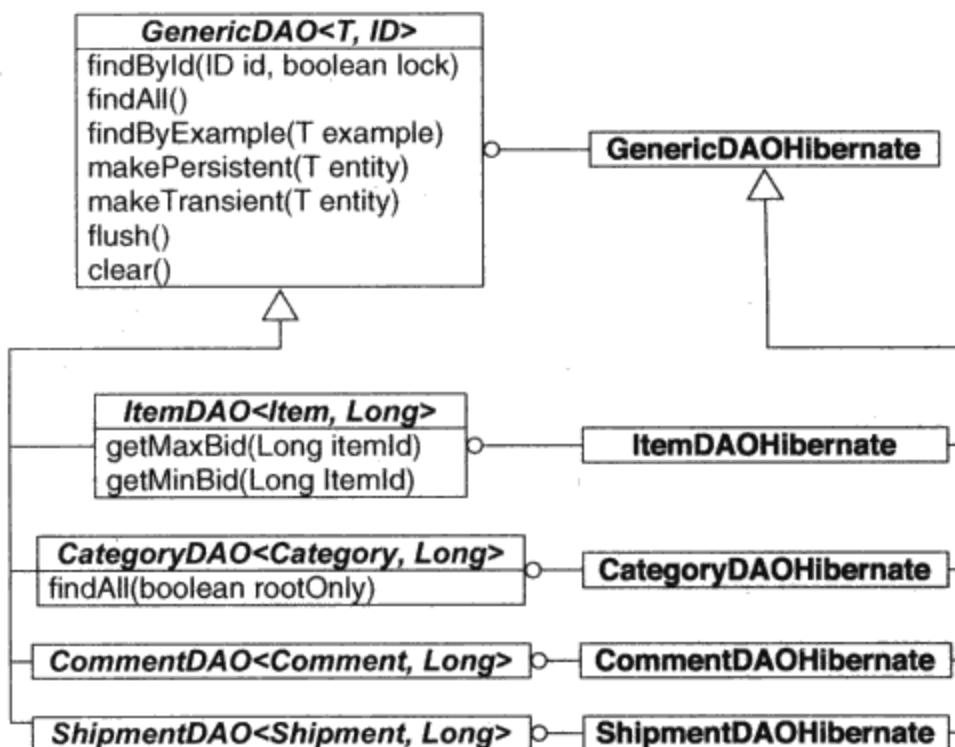


图16-1 泛型DAO接口支持任意的实现

首先考虑每个实体共享和需要的基础CRUD操作，你把这些操作组合在泛型超接口中：

```

public interface GenericDAO<T, ID extends Serializable> {

    T findById(ID id, boolean lock);
    List<T> findAll();
    List<T> findByExample(T exampleInstance,
                           String... excludeProperty);

    T makePersistent(T entity);
    void makeTransient(T entity);
    void flush();
    void clear();
}

```

GenericDAO是一个需要类型参数的接口，如果想要实现它的话。第一个参数T，是你正在为其实现DAO的实体实例。其中许多DAO方法以类型安全的方式使用这个参数来返回对象。第二个参数定义数据库标识符的类型——并非所有的实体都可以给它们的标识符属性使用相同的类型。这里值得关注的第二个东西是`findByExample()`方法中的变量参数，你很快就会看到它是如何为客户端改善API的。

最后，这无疑是持久层面向状态(state-oriented)工作的基础。一些方法如`makePersistent()`和`makeTransient()`改变一个对象（或者同时改变启用了级联的许多对象）的状态。`flush()`和`clear()`操作可以被客户端用来管理持久化上下文。如果你的持久层是面向语句的(statement-oriented)，就得编写一个完全不同的DAO接口。例如，如果你没有用Hibernate而只用了简单的JDBC实现它。

这里介绍的持久层facade不公开任何Hibernate或者Java Persistence接口给客户端，因此理论上可以通过任何软件实现它，而不用对客户端代码做任何改变。你可能不想或者不需要持久层的可移植性，如前所述。在这种情况下，应该考虑公开Hibernate或者Java Persistence接口——例如，客户端可以用来执行任意的Hibernate Criteria查询的`findByCriteria(DetachedCriteria)`方法。这个由你决定。你可能觉得，公开Java Persistence接口相对于公开Hibernate接口来说，是一种更安全的选择。然而，你应该知道，虽然可能把持久层的实现从Hibernate变到Java Persistence，或者变到任何其他全特性的面向状态的ORM软件，但是几乎不可能用简单的JDBC语句重写面向状态的持久化层。

接下来，实现DAO接口。

16.2.2 实现泛型 CRUD 接口

用Hibernate API继续探讨泛型接口的可能实现：

```
public abstract class
    GenericHibernateDAO<T, ID extends Serializable>
        implements GenericDAO<T, ID> {

    private Class<T> persistentClass;
    private Session session;

    public GenericHibernateDAO() {
        this.persistentClass = (Class<T>)
            (ParameterizedType) getClass().getGenericSuperclass()
                .getActualTypeArguments()[0];
    }

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

    protected Session getSession() {
        if (session == null)
            session = HibernateUtil.getSessionFactory()
                .getCurrentSession();
        return session;
    }

    public Class<T> getPersistentClass() {
        return persistentClass;
    }

    ...
}
```

目前为止，这是使用Hibernate实现的内部管路系统。在这个实现中，你需要访问一个Hibernate Session，因此需要DAO的客户端通过设置方法来注入当前它想要使用的Session。这在集成测试中最有用。如果客户端在使用DAO之前没有设置一个Session，那么当DAO代码需要时，你就得查找当前的Session。

DAO实现也必须知道它用于哪些持久化实体类。在构造器中要使用Java反射（Java Reflection），来查找泛型参数T的类，并把它保存在一个本地的成员中。

如果用Java Persistence编写一个泛型DAO实现，那么代码看起来几乎一样。唯一的改变是DAO需要EntityManager，而不是Session。

现在可以实现实际的CRUD操作，还是用Hibernate：

```
@SuppressWarnings("unchecked")
public T findById(ID id, boolean lock) {
    T entity;
    if (lock)
        entity = (T) getSession()
            .load(getPersistentClass(), id, LockMode.UPGRADE);
    else
        entity = (T) getSession()
            .load(getPersistentClass(), id);

    return entity;
}

@SuppressWarnings("unchecked")
public List<T> findAll() {
    return findByCriteria();
}

@SuppressWarnings("unchecked")
public List<T> findByExample(T exampleInstance,
                           String... excludeProperty) {
    Criteria crit =
        getSession().createCriteria(getPersistentClass());
    Example example = Example.create(exampleInstance);
    for (String exclude : excludeProperty) {
        example.excludeProperty(exclude);
    }
    crit.add(example);
    return crit.list();
}

@SuppressWarnings("unchecked")
public T makePersistent(T entity) {
    getSession().saveOrUpdate(entity);
    return entity;
}

public void makeTransient(T entity) {
    getSession().delete(entity);
}

public void flush() {
    getSession().flush();
}

public void clear() {
    getSession().clear();
}

/**
 * Use this inside subclasses as a convenience method.
 */
```

```

@SuppressWarnings("unchecked")
protected List<T> findByCriteria(Criterion... criterion) {
    Criteria crit =
        getSession().createCriteria(getPersistentClass());
    for (Criterion c : criterion) {
        crit.add(c);
    }
    return crit.list();
}

```

所有的数据访问操作都用getSession()获得被分配到这个DAO的Session。这些方法中大多数都很简单，在读了本书前面的章节之后，理解它们应该不成问题了。@SurpressWarning注解是可选的——Hibernate接口是给JDK 5.0之前的版本编写的，因此所有的转换都是未经检查的，否则JDK 5.0编译器就会给每一个转换都生成警告。看一下受保护的findByCriteria()方法：我们认为这是一种使其他数据访问操作的实现变得更容易的便利方法。它采用0个或者更多个Criterion参数，并把它们添加到随后要被执行的Criteria。这是JDK 5.0变量参数的一个示例。注意，我们决定不在公共的泛型DAO接口上公开这个方法；它是一个实现细节（你可能得出一个不同的结论）。

利用Java Persistence实现很简单，虽然它不支持Criteria API。你不用saveOrUpdate()，而是用merge()使任何瞬时的或者脱管的对象变成持久化，并返回合并后的结果。

你现在已经完成了持久层的基础部件和它公开给系统上层的泛型接口。下一步，你要创建实体相关的DAO接口，并通过扩展泛型的接口和实现来实现它们。

16.2.3 实现实体 DAO

假设你想要给Item业务实体实现非CRUD的数据访问操作。首先，编写一个接口：

```

public interface ItemDAO extends GenericDAO<Item, Long> {
    Bid getMaxBid(Long itemId);
    Bid getMinBid(Long itemId);
}

```

ItemDAO接口扩展了泛型的超接口，并用一个Item实体类型和一个作为数据库标识符类的Long，把它参数化。两个数据访问操作与Item实体相关：getMaxBid()和getMinBid()。

这个接口利用Hibernate的实现，扩展泛型的CRUD实现：

```

public class ItemDAOHibernate
    extends GenericHibernateDAO<Item, Long>
    implements ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Query q = getSession().getNamedQuery("getItemMaxBid");
        q.setParameter("itemid", itemId);
        return (Bid) q.uniqueResult();
    }
}

```

```

        public Bid getMinBid(Long itemId) {
            Query q = getSession().getNamedQuery("getItemMinBid");
            q.setParameter("itemid", itemId);
            return (Bid) q.uniqueResult();
        }
    }
}

```

你可以看到，有了超类提供的功能，这个实现是多么容易。查询已经被外部化到映射元数据，并且按名称被调用，这避免了弄乱代码。

建议你甚至为没有任何非CRUD数据访问操作的实体创建接口：

```

public interface CommentDAO extends GenericDAO<Comment, Long> {
    // Empty
}

```

这个实现相当简单：

```

public static class CommentDAOHibernate
    extends GenericHibernateDAO<Comment, Long>
    implements CommentDAO {}

```

我们推荐这个空接口和实现，因为你无法实例化泛型的抽象实现。此外，客户端应该依赖专用于特定实体的接口，这样避免了如果引入其他的数据访问操作而导致未来高成本的重构。但是，你可能不会接受我们的建议，从而使GenericHibernateDAO变成非抽象的。这个决定取决于你正在编写的应用程序，以及你希望未来会做的哪些改变。

把所有这些都放在一起，看看客户端如何实例化并使用DAO。

16.2.4 利用数据访问对象

如果客户端希望利用持久层，它必须实例化它需要的DAO，然后在这些DAO中调用方法。在前面引入的Hibernate Web应用程序的用例中，控制器和动作代码看起来像这样：

```

public void execute() {
    Long itemId = ...           // Get value from request
    Long userId = ...           // Get value from request
    BigDecimal bidAmount = ...  // Get value from request

    // Prepare DAOs
    ItemDAO itemDAO = new ItemDAOHibernate();
    UserDAO userDAO = new UserDAOHibernate();

    // Load requested Item
    Item item = itemDAO.findById(itemId, true);

    // Get maximum and minimum bids for this Item
    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

    // Load bidder
    User bidder = userDAO.findById(userId, false);

    try {
        Bid newBid = item.placeBid(bidder,

```

```

        bidAmount,
        currentMaxBid,
        currentMinBid);

    ... // Place new Bid into request context
    ... // Forward to success page

} catch (BusinessException e) {
    ... // Forward to appropriate error page
}

}

```

几乎设法避免在控制器中依赖任何的Hibernate代码了，除了这一件事之外：你仍然需要在控制器中实例化一个特定的DAO实现。避免这种依赖的一种（并不复杂的）方法是传统的抽象工厂模式。

首先，给数据访问对象创建一个抽象的工厂：

```

public abstract class DAOFactory {

    /**
     * Factory method for instantiation of concrete factories.
     */
    public static DAOFactory instance(Class factory) {
        try {
            return (DAOFactory) factory.newInstance();
        } catch (Exception ex) {
            throw new RuntimeException(
                "Couldn't create DAOFactory: " + factory
            );
        }
    }

    // Add your DAO interfaces here
    public abstract ItemDAO getItemDAO();
    public abstract CategoryDAO getCategoryDAO();
    public abstract CommentDAO getCommentDAO();
    public abstract UserDAO getUserDAO();
    public abstract BillingDetailsDAO getBillingDetailsDAO();
    public abstract ShipmentDAO getShipmentDAO();
}

```

这个抽象的工厂可以构建和返回任何DAO。现在给你的Hibernate DAO实现这个工厂：

```

public class HibernateDAOFactory extends DAOFactory {

    public ItemDAO getItemDAO() {
        return (ItemDAO) instantiateDAO(ItemDAOHibernate.class);
    }

    ...

    private GenericHibernateDAO instantiateDAO(Class daoClass) {
        try {
            GenericHibernateDAO dao = (GenericHibernateDAO)
                daoClass.newInstance();

```

```

        return dao;
    } catch (Exception ex) {
        throw new RuntimeException(
            "Can not instantiate DAO: " + daoClass, ex
        );
    }
}

// Inline all empty DAO implementations

public static class CommentDAOHibernate
    extends GenericHibernateDAO<Comment, Long>
    implements CommentDAO {}

public static class ShipmentDAOHibernate
    extends GenericHibernateDAO<Shipment, Long>
    implements ShipmentDAO {}

...
}

```

这里发生了几件值得关注的事情。第一，工厂的实现封装了实例化DAO的方式。可以定制这个方法，并在返回DAO实例之前手工设置一个Session。

其次，把CommentDAOHibernate的实现作为公共的静态类移到工厂里面。记住，你需要这个实现（即使它是空的）来让客户端使用与实体相关的接口。然而，没有人强迫你在单独的文件中创建几十个空的实现类，可以在工厂中分组所有的空实现。如果未来必须给Comment实体引入更多的数据访问操作，就把这个实现从工厂移到它自己的文件中去。不需要改变其他的代码——客户端只依赖CommentDAO接口。

利用这个工厂模式，可以进一步简化DAO在Web应用程序控制器中的使用方法：

```

public void execute() {

    Long itemId = ...           // Get value from request
    Long userId = ...           // Get value from request
    BigDecimal bidAmount = ...  // Get value from request

    // Prepare DAOs
    DAOFactory factory = DAOFactory.instance(DAOFactory.HIBERNATE);
    ItemDAO itemDAO = factory.getItemDAO();
    UserDAO userDAO = factory.getUserDAO();

    // Load requested Item
    Item item = itemDAO.findById(itemId, true);

    // Get maximum and minimum bids for this Item
    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

    // Load bidder
    User bidder = userDAO.findById(userId, false);

    try {
        ...
    }
}

```

唯一依赖Hibernate，并把持久层的真正实现公开给客户端代码的唯一一行代码，是DAOFactory的获取。你可能想要考虑把这个参数移到应用程序的外部配置中，以便可能转换DAOFactory实现，而不用改变任何代码。

提示 在一个DAO中混用Hibernate和JDBC代码——当你可以使用Hibernate时，很少需要使用简单的JDBC。记住，如果你需要JDBC Connection来执行Hibernate无法自动生成的语句，始终可以退回来使用session.connection()。因此，我们认为你不需要给少数JDBC调用使用不同的和单独的DAO。混合使用Hibernate和简单的JDBC的问题，不在于你有时候可能不得不这么做（不应该指望Hibernate会解决你的全部问题），而在于开发人员经常试图隐藏他们做过的事情。只要形成适当的文档，混合的数据访问代也没有问题。还要记住，Hibernate支持几乎所有利用原生API的SQL操作，因此你不必退回到简单的JDBC。

你现在已经创建了一个整洁、灵活且强大的持久层，它从任何客户端代码中隐藏了数据访问的细节。你可能仍然存有下列疑虑：

- 必须编写工厂吗？工厂模式是传统的，用在主要依赖无状态服务查找的应用程序中。另一种可选的（或者有时是补充的）策略是依赖注入（dependency injection）。EJB 3.0规范给托管组件标准化了依赖注入，因此本章稍后将讨论另一种DAO编写策略。
- 必须给每个领域实体创建DAO接口吗？我们的建议并不涵盖所有可能的情况。在更大的应用程序中，你可能想要按领域包来分组DAO，或者创建更深的DAO层次结构（它给特定的几个子实体提供更细粒度的特殊化）。DAO模式有许多种变形，你不应该受限于我们所建议的泛型解决方案。放心地去体验吧，并把这个模式当作一个好的起点。

你现在知道如何在传统的Web应用程序中整合Hibernate，以及如何根据最佳的实践模式创建持久层。如果必须设计和编写一个三层的应用程序，则需要考虑一个完全不一样的架构了。

16.3 命令模式简介

如果必须利用Hibernate和Java Persistence编写中小型的Web应用程序，前几节中介绍过的模式和策略就非常完美。OSIV模式适用于两层的架构，其中表现层、业务层和持久层都协同分布在一台虚拟机上。

然而，一旦你引入第三层，并把表现层移到一台单独的虚拟机上，当前的持久化上下文就无法再保持打开状态，直到视图完全呈现。这是三层的EJB应用程序或者单独的进程中包含富客户端的架构中的典型案例。

如果表现层在一个不同的进程中运行，就要把这个进程与运行应用程序的业务层和持久层的层之间的请求减到最少。这意味着你不能使用前面的延迟方法（必要时它允许视图从领域模型对象中拖取数据）。反之，业务层必须接受抓取后面呈现视图所需的所有数据的责任。

虽然某些模式可以把远程通信减到最少[如会话外观（session facade）和数据传输对象（data transfer object, DTO）模式]，并在Java开发社区中已经得到广泛应用，但我们还是想讨论一种稍

微不同的方法。命令（Command）模式（经常被称作EJB命令）是一种复杂的解决方案，它结合了其他策略的优势。

我们来编写一个利用这种模式的三层应用程序。

16.3.1 基础接口

命令模式以命令类的层次结构思想为基础，所有命令类都实现一个简单的Command接口。看一下图16-2中的这个层次结构。

特定的Command是动作、事件或者符合类似描述的任何东西的实现。客户端代码创建命令对象，并准备执行它们。CommandHandler是一个可以执行Command对象的接口。客户端把Command对象传递到服务器层中的一个处理器，这个处理器执行它。然后Command对象被返回到客户端。

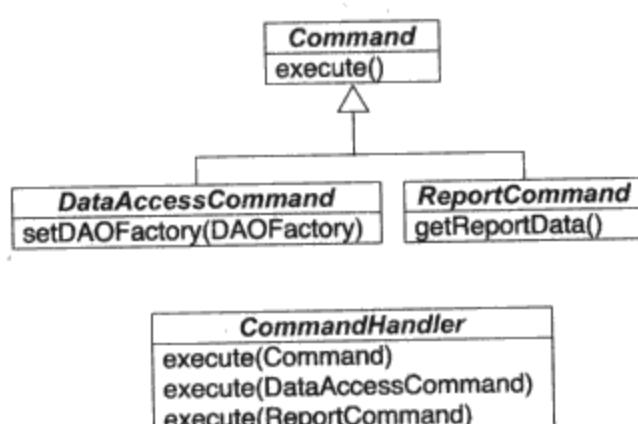


图16-2 命令模式的接口

Command接口有一个execute()方法；任何具体的命令都必须实现这个方法。任何子接口都可以添加在执行Command之前（设置方法）或者之后（获取方法）而被调用的额外方法。因此Command给特定的事件组合了输入（input）、控制器（controller）和输出（output）。

执行Command对象（即，调用它们的execute()方法）是CommandHandler实现的工作。命令的执行被多态地分派。

这些接口（和抽象类）的实现可以看作如下：

```

public interface Command {
    public void execute() throws CommandException;
}

```

Command也封装异常处理，以便执行期间抛出的任何异常都被包装在CommandException（之后可以被客户端相应处理）中。

DataAccessCommand是一个抽象类：

```

public abstract class DataAccessCommand implements Command {
    protected DAOFactory daoFactory;

    public void setDAOFactory(DAOFactory daoFactory) {
        this.daoFactory = daoFactory;
    }
}

```

任何需要访问数据库的Command都必须使用一个数据访问对象，因此DAOFactory必须在DataAccessCommand可以被执行之前进行设置。这通常是CommandHandler实现的工作，因为持久层处在服务器层中。

命令处理器的远程接口相当简单：

```
public interface CommandHandler {
    public Command executeCommand(Command c)
        throws CommandException;
    public DataAccessCommand executeCommand(DataAccessCommand c)
        throws CommandException;
    public ReportCommand executeCommand(ReportCommand c)
        throws CommandException;
}
```

下面编写一些具体的实现并使用命令。

16.3.2 执行命令对象

希望执行命令的客户端需要实例化并准备一个Command对象。例如，给一次拍卖出价需要客户端上的BidForItemCommand：

```
BidForItemCommand bidFormItem =
    new BidForItemCommand(userId, itemId, bidAmount);

try {
    CommandHandler handler = getCommandHandler();
    bidFormItem = (BidForItemCommand) handler.executeCommand(bidFormItem);
    // Extract new bid for rendering
    newBid = bidFormItem.getNewBid();

    // Forward to success page
} catch (CommandException ex) {
    // Forward to error page
    // ex.getCause();
}
```

对于这个动作，BidForItemCommand需要所有的输入值作为构造器参数。然后客户端查找一个命令处理器，并传递BidForItemCommand对象用于执行。处理器在执行之后返回实例，客户端从返回的对象中提取任何输出值。（如果你使用JDK 5.0，就用泛型来避免不安全的类型转换。）

命令处理器如何被查找或者实例化，这取决于命令处理器的实现，以及远程的通信如何发生。你甚至不必调用远程的命令处理器——它可以是一个本地的对象。

来看一下命令和命令处理器的实现。

1. 实现业务命令

BidForItemCommand扩展抽象类DataAccessCommand，并实现execute()方法：

```

public class BidForItemCommand extends DataAccessCommand
    implements Serializable {

    // Input
    private Long userId;
    private Long itemId;
    private BigDecimal bidAmount;

    // Output
    private Bid newBid;

    public BidForItemCommand(Long userId,
                           Long itemId,
                           BigDecimal bidAmount) {
        this.userId = userId;
        this.itemId = itemId;
        this.bidAmount = bidAmount;
    }

    public Bid getNewBid() {
        return newBid;
    }

    public void execute() throws CommandException {
        ItemDAO itemDAO = daoFactory.getItemDAO();
        UserDAO userDAO = daoFactory.getUserDAO();

        try {
            Bid currentMaxBid = itemDAO.getMaxBid(itemId);
            Bid currentMinBid = itemDAO.getMinBid(itemId);

            Item item = itemDAO.findById(itemId, false);
            newBid = item.placeBid(userDAO.findById(userId, false),
                                   bidAmount,
                                   currentMaxBid,
                                   currentMinBid);

        } catch (BusinessException ex) {
            throw new CommandException(ex);
        }
    }
}

```

这基本上与你在本章前面修饰Web应用程序的最后阶段中所写的代码一样。然而，利用这种方法，你对于一次拍卖所需的输入和返回的输出有着清晰的约定。

因为Command实例是跨越网络进行传送，所以你必须实现Serializable（这个标识应该处在具体类中，而不是在超类或者接口中）。

我们来实现命令处理器。

2. 实现命令处理器

命令处理器可以用你喜欢的任何方式实现；它的责任很简单。许多系统只需要单个命令处理器，例如：

```

@Stateless
public class CommandHandlerBean implements CommandHandler {

    // The persistence layer we want to call
    DAOFactory daoFactory =
        DAOFactory.instance(DAOFactory.HIBERNATE);

    @TransactionAttribute(TransactionAttributeType.NEVER)
    public Command executeCommand(Command c)
        throws CommandException {
        c.execute();
        return c;
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Command executeCommand(DataAccessCommand c)
        throws CommandException {
        c.setDAOFactory(daoFactory);
        c.execute();
        return c;
    }
}

```

这是一个实现为无状态的EJB 3.0会话bean的命令处理器。你在客户端用一个EJB查找获得对这个（本地或者远程）bean的引用，然后将Command对象传递给它用于执行。处理器知道如何准备一个特定的命令类型——例如，通过在执行之前设置对持久层的一个引用。

由于容器托管的事务和声明性事务，这个命令处理器没有包含任何Hibernate代码。当然，你也可以将这个命令处理器实现为没有EJB 3.0注解的一个POJO，并编程式地管理事务范围。另一方面，因为EJB支持开箱即用的远程通信，它们是三层架构中最佳的命令处理器选择。

这个基本的命令模式还有更多的变形。

16.3.3 命令模式的变形

首先，命令模式也并非十全十美。这种模式可能最重要的问题是，在客户端类路径中需要非表现接口。由于BidForItemCommand需要DAO，你必须在客户端的类路径中包括持久层接口（即使命令只在中间层执行）。没有真正的解决方案，因此这个问题的严重性取决于你的部署场景，以及你可以多轻松地给应用程序相应地打包。注意，客户端只需要DAO接口来实例化DataAccessCommand，因此，你或许可以在实现持久层之前稳定接口。

而且，由于你只有一个命令，命令模式似乎比传统的会话外观模式喜欢更多的工作。然而，随着系统的成长，因为横切关注点如异常处理和验证检查可以在命令处理器中实现，因此增加新命令变得更加简单。命令很容易实现，并且极易重用。你不应该受限于我们所建议的命令接口层次结构，放心地去设计更复杂且更成熟的命令接口和抽象的命令吧。你也可以利用委托把命令分组在一起——例如，DataAccessCommand可以实例化并调用ReportCommand。

对于呈现特定的视图所需的数据来说，命令是很好的装配器。不用让视图从延迟加载的业务对象中拖取信息（这需要表现层和持久层的协同定位，因此你可以留在同一个持久化上下文中），而是客户端可以准备和执行呈现特定屏幕所需的命令——每一个命令都把数据传输到它输出属

性中的表现层。命令有点像是一种带有内建装配子程序的数据迁移对象（data-transfer object）。

此外，命令模式让你能够轻松地实现任何撤销（Undo）功能。每一个命令都可以有一个`undo()`方法，它可以禁用`execute()`方法已经完成的任何永久的改变。或者，你可以在客户端排列几个命令对象，并且只在特定的对话完成时，才把它们发送到命令处理器。

如果必须实现一个桌面应用程序，命令模式也很好。例如，可以实现在数据改变时触发事件的一个命令。通过在命令处理器中注册一个监听器，所有需要被刷新的对话框都可以监听到这个事件。

可以用EJB 3.0拦截器把命令包装起来。例如，对于能够在特定类型的命令对象上透明地注入特定服务的命令处理器会话bean，可以为它编写拦截器。可以在命令处理器中组合和堆叠这些拦截器。甚至可以实现一个客户端-本地命令处理器，由于EJB拦截器，它可以透明地决定命令是否需要被发送到服务器（到另一个命令处理器），或者命令是否可以在客户端断开连接的情况下执行。

无状态会话bean不仅仅需要命令处理器。实现一个异步执行命令的基于JMS的命令处理器很容易。甚至可以在数据库中给计划好的执行保存命令。命令可以在服务器环境的外部使用——比如在一个批量处理或者单元测试案例中。

在实践中，依赖命令模式的架构运行得相当好。

下一节将讨论EJB 3.0组件如何进一步简化分层的应用程序架构。

16.4 利用EJB 3.0设计应用程序

本书已经关注过Java Persistence标准，并只讨论过几个其他EJB 3.0编程构造的例子。我们编写了一些EJB会话bean，启用了容器托管的事务，并使用容器注入获得EntityManager。

在EJB 3.0编程模型中，还有更多的东西等着去发现。接下来的几节要介绍如何简化前面利用EJB 3.0组件的一些模式。然而，我们仍然只讨论数据库应用程序相关的特性，因此如果你想要了解更多有关定时器、EJB拦截器或者消息驱动的EJB的信息，要参考其他的文档。

首先，你将在一个包含有状态会话bean（对话控制器）的Web应用程序中实现一个动作。然后简化数据访问对象（通过把它们变成EJB）来获得容器托管的事务和依赖注入。你也将从任何Hibernate接口转换到Java Persistence，以保持与EJB 3.0的完全兼容。

通过在Web应用程序中利用EJB 3.0组件实现对话来开始下面的内容。

16.4.1 利用有状态的bean实现会话

对于应用程序和用户之间可能长运行的对话来说，有状态会话bean（SFSB）是最完美的控制器。可以编写一个在一个对话（如PlaceItem对话）中实现所有步骤的SFSB：

- (1) 用户输入货品信息；
- (2) 用户可以给一件货品添加图片；
- (3) 用户提交完成的表格。

如果必须添加的图片不止一张，可以重复执行这个对话的第2步。让我们通过一个直接使用Java Persistence和EntityManager的SFSB来实现它：

单一的SFSB实例负责整个对话。首先，下面是业务接口：

```
public interface PlaceItem {
    public Item createItem(Long userId, Map itemData);
    public void addImage(String filename);
    public void submit();
}
```

在对话的第一步中，用户输入基础的货品细节并提供用户标识符。从这里，创建一个Item实例，并保存在对话中。然后用户可以多次执行addImage()事件。最后，用户完成了表格，并调用submit()方法来终止对话。注意你如何能够阅读这个接口就像阅读对话的历史一样。

这是一种可能的实现：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NEVER)
public class PlaceItemBean implements PlaceItem {
    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    private EntityManager em;

    private Item item;
    private User seller;

    public Item createItem(Long userId, Map itemData) {
        // Load seller into conversation
        seller = em.find(User.class, userId);

        // Create item for conversation
        item = new Item(itemData, seller);
        seller.addItem(item);

        return item;
    }

    public void addImage(String filename) {
        item.getImages().add(filename);
    }

    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void submit() {
        em.persist(item);
    }
}
```

这个有状态会话bean的实例被绑定到一个特定的EJB客户端，因此它在对话期间也表现得像一个高速缓存。你使用一个只在submit()返回时被清除的扩展持久化上下文，因为这是在事务内部执行的唯一方法。其他方法中的所有数据访问都在自动提交模式下运行。因此em.find(User.class,userId)执行非事务的，反之em.persist(item)则是事务的。由于submit()方法也用@Remove作标识，因此当这个方法返回时，持久化上下文自动关闭，有状态会话bean也被销毁了。

这个实现的一种变形不直接调用EntityManager，而是调用数据访问对象。

16.4.2 利用EJB编写DAO

数据访问对象是完美的无状态会话bean。每一个数据访问方法都不需要任何状态，它只需要EntityManager。因此，当用Java Persistence实现GenericDAO时，就需要设置EntityManager：

```
public abstract class GenericEJB3DAO<T, ID extends Serializable>
    implements GenericDAO<T, ID> {

    private Class<T> entityBeanType;

    private EntityManager em;

    public GenericEJB3DAO() {
        this.entityBeanType = (Class<T>)
            ( (ParameterizedType) getClass().getGenericSuperclass() )
            .getActualTypeArguments()[0];
    }

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    protected EntityManager getEntityManager() {
        return em;
    }

    public Class<T> getEntityBeanType() {
        return entityBeanType;
    }

    ...
}
```

这实际上与前面你在16.2.2节中给Hibernate创建的实现是一样的。然而，你用@PersistenceContext给setEntityManager()方法作了标识，因此当这个bean在一个容器内部执行时，就得到正确的EntityManager的自动注入。如果它在EJB 3.0运行时容器之外执行，你可以手工设置EntityManager。

我们不介绍利用JPA的所有CRUD操作的实现，你应该能够自己实现findById()等。

下面是一个带有业务数据访问方法的具体DAO的实现：

```
@Stateless
@TransactionalAttribute(TransactionAttributeType.REQUIRED)
public class ItemDAOBean extends GenericEJB3DAO<Item, Long>
    implements ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Query q = getEntityManager()
            .createNamedQuery("getItemMaxBid");
        q.setParameter("itemid", itemId);
        return (Bid) q.getSingleResult();
    }
}
```

```

public Bid getMinBid(Long itemId) {
    Query q = getEntityManager()
        .createNamedQuery("getItemMinBid");
    q.setParameter("itemid", itemId);
    return (Bid) q.getSingleResult();
}

}

```

这个具体的子类是个无状态EJB会话bean，被调用的所有方法（包括了从GenericDAO超类继承来的那些方法）都需要一个事务上下文。如果这个DAO的客户端调用没有活动事务的方法，就会给这个DAO方法启动一个事务。

不再需要任何DAO工厂。前面编写过的对话控制器通过依赖注入自动与DAO连接。

16.4.3 利用依赖注入

现在重构PlaceItem对话控制器，并增加一个持久层。不直接访问JPA，而是调用在运行时被容器注入到对话控制器中的几个DAO：

```

@Stateful
public class PlaceItemWithDAOsBean implements PlaceItem {

    @PersistenceContext(
        type = PersistenceContextType.EXTENDED,
        properties =
            @PersistenceProperty(
                name="org.hibernate.flushMode",
                value="MANUAL"
            )
    )
    private EntityManager em;

    @EJB ItemDAO itemDAO;
    @EJB UserDAO userDAO;

    private Item item;
    private User seller;

    public Item createItem(Long userId, Map itemData) {
        // Load seller into conversation
        seller = userDAO.findById(userId);

        // Create item for conversation
        item = new Item(itemData, seller);

        return item;
    }

    public void addImage(String filename) {
        item.getImages().add(filename);
    }

    @Remove
    public void submit() {

```

```

        itemDAO.makePersistent(item);
        em.flush();
    }
}

```

@EJB注解给自动的依赖注入标识了itemDAO和userDAO字段。容器查找指定接口的一个实现（这个实现是依赖供应商的，但是在这个例子中，每个接口只有一个），并在字段中对它进行设置。

你还没有在这个实现中禁用事务，只是通过Hibernate org.hibernate.flushmode扩展属性禁用了自动清除。然后当SFSB中标有@Remove的方法结束，并在这个方法的事务提交之前清除持久化上下文一次。

这么做有下面两个原因：

- 你正在调用的所有DAO方法都需要一个事务上下文。如果没有在对话控制器中给每个方法启动一个事务，事务范围就是其中一个数据访问对象中的调用。然而，你想要createItem()、addImages()和submit()方法成为事务的范围，以防万一执行几个DAO操作。
- 你有一个被扩展的持久化上下文，它被自动界定和绑定到有状态会话bean。因为DAO是无状态的会话bean，所以只有当事务上下文是活动的并且也被传播时，这个单独的持久化上下文才可以被传播到所有的DAO里面。如果DAO是有状态的会话bean，你可以通过实例化（甚至在没有事务上下文用于DAO调用的时候）传播当前的持久化上下文，但是这也意味着对话控制器必须手工销毁任何有状态的DAO。

如果没有Hibernate扩展属性，你必须使DAO变成有状态会话bean，以允许在非事务的方法调用之间传播持久化上下文。那么在控制器的@Remove方法中调用每个DAO的@Remove方法，就是它自己的责任了——这两种情况都不是你想要的。你想要禁用清除，又不用编写任何非事务的方法。

EJB 3.0包括更多的注入特性，并且它们扩展到其他的Java EE 5.0规范。例如，可以在一个Java servlet容器中使用@EJB注入，或者@Resource，自动从被注入的JNDI中获得任何指定的资源。但是这些特性都不是本书讨论的范围。

既然已经创建了应用层，就需要一种方法来测试它们的正确性。

16.5 测试

测试可能是Java开发人员一天的工作当中最重要的一项活动。测试从功能以及性能和可伸缩性方面确定系统的正确性。测试执行成功意味着所有的应用程序组件和各层都正确地进行交互，并且如指定的那样一起顺利地运行。

可以用多种不同的方法测试和证明一套软件系统。在持久化上下文和数据管理中，一般最关注自动化测试。在接下来的几节中，你将创建许多种测试，你可以重复运行它们来检查应用程序的正确行为。

先来看一下不同的测试种类。功能测试、集成测试和独立的单元测试，它们全部都有着不同的目标和用途，你要知道每种策略什么时候最适当。然后我们编写测试并引入TestNG框架(<http://www.testng.org>)。最后，考虑压力和负载测试，以及如何查明系统是否要伸缩为大量的并发事务。

16.5.1 理解不同种类的测试

划分软件测试的种类如下：

- 验收测试 (acceptance testing) ——这种测试不是自动化测试必需的，并且通常不是应用程序开发人员和系统设计人员的工作。验收测试是系统测试的最后阶段，由确定系统是否满足项目需求的客户（或者任何其他方）来执行。这些测试可以包括任何准则，从功能到性能到可用性。
- 性能测试 (performance testing) ——压力测试或负载测试使用大量并发的用户对系统进行测试，理想情况下是等于或者高于一旦软件正式运行时所期待的负载。因为对于任何使用在线事务数据处理的应用程序来说，这个测试是如此重要的一个方面，因此我们稍后会深入讨论性能测试。
- 逻辑单元测试 (logic unit testing) ——这些测试考虑单项功能，经常只是一个业务方法（例如，最高的出价是否真的在拍卖系统中胜出）。如果一个组件作为单个单元进行测试，它就是独立于任何其他的组件进行测试。逻辑单元测试不涉及任何像数据库这样的子系统。
- 集成单元测试 (integration unit testing) ——集成测试确定软件组件、服务和子系统之间的交互是否如预期的那样工作。在事务处理和数据管理的上下文中，这可能意味着你想要测试应用程序是否正确地利用数据库进行工作（例如，刚对一件拍卖货品所做的出价是否正确地被保存在数据库中）。
- 功能单元测试 (functional unit testing) ——功能测试检验整个用例，以及完成这个特定的用例所需的所有应用程序组件中的公共接口。功能测试可以包括应用程序的工作流和用户界面（例如，通过模拟用户在对一件拍卖货品给出新价格之前必须如何登录）。

接下来的几节关注集成单元测试，因为当持久化数据和事务处理成为你的主要关注点时，它是最相关的一种测试。这并不意味着其他种类的测试没有它重要，我们会随时提供提示。如果你想要全面地了解单元测试，我们推荐*JUnit in Action* (Massol, 2003)。

不用JUnit，而是用TestNG。这不应该对你产生什么困扰，因为这里介绍的基本原理可适用于任何测试框架。TestNG比JUnit更容易进行集成和功能单元测试，我们尤其喜欢JDK 5.0特性和基于注解的测试集合配置。

先来编写一个简单、单独的逻辑单元测试，以便你可以了解TestNG是如何工作的。

16.5.2 TestNG 简介

TestNG是一个测试框架，它有一些独特的功能，这使得它对于涉及复杂的测试（如集成或者功能测试等）的单元测试特别有用。其中有些TestNG特性是用于测试集合声明的JDK 5.0注解，支持配置参数，以及把测试灵活地分组到测试单元中去的分组，支持用于IDE和Ant的各种插件，以及通过遵循依赖以特定顺序执行测试的能力。

我们想要一步一步地接近这些特性，因此你首先要编写一个简单的逻辑单元测试，没有任何子系统的集成。

1. TestNG中的单元测试

逻辑单元测试验证单项功能，并检查特定的组件或者方法是否遵循所有的业务规则。如果理解本章前面关于巧妙的领域模型的讨论（详见16.1.4节），就会知道我们更喜欢将可以进行单元测试的业务逻辑封装在领域模型实现中。逻辑单元测试在业务层和领域模型中执行方法的测试：

```
public class AuctionLogic {

    @org.testng.annotations.Test(groups = "logic")
    public void highestBidWins() {

        // A user is needed
        User user = new User(...);

        // Create an Item instance
        Item auction = new Item(...);

        // Place a bid
        BigDecimal bidAmount = new BigDecimal("100.00");
        auction.placeBid(user, bidAmount,
                          new BigDecimal(0), new BigDecimal(0) );

        // Place another higher bid
        BigDecimal higherBidAmount = new BigDecimal("101.00");
        auction.placeBid(user, higherBidAmount,
                          bidAmount, bidAmount );

        // Assert state
        assert auction.getBids().size() == 2;
    }
}
```

AuctionLogic类是一个包含所谓测试方法的任意类。测试方法（test method）是用@Test注解作了标识的任何方法。可以选择把分组的名称分配给测试方法，以便以后可以通过合并分组来动态地装配测试单元。

测试方法highestBidWins()给“出价”用例执行部分逻辑。首先，出价需要User的实例——是否为同一个用户并非这项测试的关注点。

这项测试可能以几种方式失败，表明已经违背了业务规则。第一个出价启动了拍卖（当前最高和最低出价均为0），因此这里不会有任何失败。出第二个价格是必须成功且不抛出BusinessException的步骤，因为新的出价金额比前一个更高。最后，用Java assert关键字和一个比较操作决定拍卖状态。

你会经常想要测试失败的业务逻辑，并期望等到异常。

2. 期待测试中的失败

拍卖系统有一个相当严重的bug。如果看一下16.1.4节中Item.placeaBid()的实现，就可以看到你检查所给定的新出价金额是否比任何现有的出价金额更高。然而，你从来不把它与拍卖的初始底价进行对比。这意味着用户可以出任何价格，即使比初始价格更低。

通过测试失败来检验这一点。下列过程会产生异常：

```

public class AuctionLogic {

    @Test(groups = "logic")
    public void highestBidWins() { ... }

    @Test(groups = "logic")
    @ExpectedExceptions(BusinessException.class)
    public void initialPriceConsidered() {

        // A user is needed
        User user = new User(...);

        // Create an Item instance
        Item auction = new Item(..., new BigDecimal("200.00"));
        // Place a bid
        BigDecimal bidAmount = new BigDecimal("100.00");
        auction.placeBid(user, bidAmount,
            new BigDecimal(0), new BigDecimal(0));
    }
}

```

现在，值为100的出价不得不失败，因为拍卖的初始底价是200。TestNG要求这个方法抛出BusinessException——否则测试就失败。更细粒度的业务异常类型让你对业务逻辑的核心部分更精确地测试失败。

最后，考虑领域模型的执行路径的数量多少，决定了全部的业务逻辑测试覆盖度。可以使用各种工具[如cenqua clover (<http://www.cenqua.com/clover/>)，它可以算出测试套件的代码覆盖率]，并提供关于系统质量的许多其他值得关注的细节。

让我们用TestNG和Ant来执行之前的这些测试方法。

3. 创建和运行测试套件

通过TestNG，可以用多种方法创建测试套件，并启动测试。可以通过IDE中的按钮单击来直接调用测试方法（在安装了TestNG插件之后），或者可以把常规构建中的单元测试与测试套件的Ant任务和XML描述集成在一起。

前几节中单元测试的XML测试单元描述，看起来如下：

```

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="CaveatEmptor" verbose="2">

    <test name="BusinessLogic">
        <run><include name="logic.*"/></run>

        <packages>
            <package name="auction.test"/>
        </packages>

        <!-- Or just the class...
        <classes>
            <class name="auction.test.AuctionLogic"/>
        </classes>
        -->
    </test>

</suite>

```

测试套件是几个逻辑测试的集合——不要把这个与测试方法混为一谈。逻辑测试由TestNG在运行时确定。例如，名为BusinessLogic的逻辑测试将所有的测试方法（也就是说，用@Test作了标识的方法）都包括在auction.test包的类中。这些测试方法必须属于一个名称以logic开头的分组，注意.*是个正则表达式，意思是“任意数量的任意字符”。另一种方法是，可以列出你想要显式地考虑这个逻辑测试的一部分的测试类，而不是整个包（或者几个包）。

可以编写一些测试类和方法，以任何方便的方式安排它们，然后通过混合和匹配类、包和具名的组合来创建任意的测试集合。来自任意类和包的这个逻辑测试集合，以及利用通配符匹配分离到分组中，使得TestNG比许多其他的测试框架更为强大。

在项目的根目录下，将单元描述XML文件保存为test-logic.xml。现在，通过Ant和build.xml中的下列目标来运行这个测试套件：

```

<taskdef resource="testngtasks" classpathref="project.classpath"/>
<target name="unittest.logic" depends="compile, copymetafiles">
    description="Run logic unit tests with TestNG">
        <delete dir="${basedir}/test-output"/>
        <mkdir dir="${basedir}/test-output"/>
        <testng outputDir="${basedir}/test-output"
            classpathref="project.classpath">
            <xmlfileset dir="${basedir}">
                <include name="test-logic.xml"/>
            </xmlfileset>
        </testng>
    </target>

```

首先，TestNG Ant任务被引入到构建中。然后，unittest.logic目标通过项目根目录下的单元描述文件textlogic.xml启动一个TestNG运行。TestNG在outputDir中创建了一个HTML报告，因此你每次在运行测试之前都要清理这个目录。

调用这个Ant目标，并体验你的第一个TestNG集合。接下来讨论集成测试，以及TestNG如何通过运行时环境的灵活配置来支持你。

16.5.3 测试持久层

测试持久层意味着必须测试和检验几个组件，看看它们是否正确地进行交互。这意味着：

- 测试映射（testing mapping）——你想要测试映射的语法正确性（被映射的所有列和表是否都与属性和类相匹配）。
- 测试对象状态转变（testing object state transition）——你想要测试一个对象转变是否正确地从瞬时到持久化再到脱管状态。换句话说，你想要确保数据被正确地保存在数据库中，确保它可以被正确加载，确保对于传播性状态变化的所有潜在的级联规则都如预期般有效。
- 测试查询（testing query）——任何重要的HQL、Criteria和（可能的）SQL查询都应该测试被返回数据的正确性。

所有这些测试都要求持久层不要单独测试，而是与一个在运行的数据库管理系统集成在一

起。此外，所有其他的基础结构，诸如Hibernate SessionFactory或者JPA EntityManagerFactory，都必须可用。你需要一个运行时环境，启用你想要包含在集成测试中的任何服务。

考虑你想要在其上面运行这些测试的数据库管理系统。理想情况下，这应该与你要在产品中给应用程序部署的DBMS产品相同。另一方面，在有些情况下，开发时你却可能在一个不同的系统中运行集成测试——例如，轻量级的HSQL DB。注意，由于Hibernate的数据库可移植性，对象状态转变可以透明地进行测试。任何复杂的应用程序都有经常为特定的数据库管理系统进行量身定做的映射和查询（通过公式和原生的SQL语句），因此与非产品的数据库产品的任何集成测试都没有意义。许多DBMS供应商提供免费许可，甚至提供他们主要数据库产品的轻量级版本用于开发。在开发期间，在转换到不同的数据库管理系统之前先考虑这些。

必须先准备测试环境，并在你编写任何集成单元测试之前启用运行时基础设施。

1. 编写DBUnit超类

持久层的集成测试环境要求安装并激活数据库管理系统——我们希望你的案例中考虑到这一点。接下来，要考虑集成测试集合，以及如何以正确的顺序执行配置和测试。

首先，为了使用数据访问对象，你必须启动Hibernate——构建SessionFactory是最容易的部分。更难的是定义运行测试前后所需要的配置操作的顺序。一般的顺序是这样的：

- (1) 重置数据库内容为一种大家熟知的状态。这么做最容易的方法是利用Hibernate工具箱，通过数据库Schema自动导出。然后用空的、干净的数据库开始测试。
- (2) 通过将数据导入到数据库中，给测试创建任何基础数据。这可以通过各种方式来完成，比如用Java代码通过编程完成，或者利用工具如DBUnit (<http://www.dbunit.org>)。
- (3) 创建对象，并执行你想要测试的任何状态转变，比如通过调用几个DAO，在TestNG测试方法中保存或者加载对象。
- (4) 状态转变之后，通过检查Java代码中的对象，以及（或）通过执行SQL语句并验证数据库的状态，来声明状态。

考虑几个这样的集成测试。你应该始终从第1步开始，并在每一个测试方法之后都导出新的数据库Schema，然后再次导入所有基础数据吗？如果执行大量的测试，这可能很费时间。另一方面，这种方法比在每一个测试方法之后的删除和清理更为容易，这应该成为一个额外的步骤。

有一个可以帮助你给每个测试进行这些配置和准备步骤的工具是DBUnit。可以轻松地导入和管理数据集——例如，必须为每一次测试运行重置为大家熟知的状态的数据集。

即使TestNG允许你以任何可以想象的方式组合和装配测试单元，但封装所有配置和DBUnit设置操作的超类还是很方便。看一下代码清单16-2中适合于Hibernate数据访问对象的集成测试的超类。

代码清单16-2 Hibernate集成测试的超类

```
public abstract class HibernateIntegrationTest {
    protected SessionFactory sessionFactory; ①
    protected String dataSetLocation; ②
}
```

```

protected List<DatabaseOperation> beforeTestOperations
    = new ArrayList<DatabaseOperation>();
protected List<DatabaseOperation> afterTestOperations
    = new ArrayList<DatabaseOperation>();

private ReplacementDataSet dataSet;           ←③

@BeforeTest(groups = "integration-hibernate") ←④
void startHibernate() throws Exception {
    sessionFactory = HibernateUtil.getSessionFactory();
}

@BeforeClass(groups = "integration-hibernate") ←⑤
void prepareDataSet() throws Exception {

    // Check if subclass has prepared everything
    prepareSettings();
    if (dataSetLocation == null)
        throw new RuntimeException(
            "Test subclass needs to prepare a dataset location"
        );

    // Load the base dataset file
    InputStream input =
        Thread.currentThread().getContextClassLoader()
            .getResourceAsStream(dataSetLocation);

    dataSet = new ReplacementDataSet(
        new FlatXmlDataSet(input)
    );
    dataSet.addReplacementObject("[NULL]", null);
}

@BeforeMethod(groups = "integration-hibernate") ←⑥
void beforeTestMethod() throws Exception {
    for (DatabaseOperation op : beforeTestOperations) {
        op.execute(getConnection(), dataSet);
    }
}

@AfterMethod(groups = "integration-hibernate") ←⑦
void afterTestMethod() throws Exception {
    for (DatabaseOperation op : afterTestOperations) {
        op.execute(getConnection(), dataSet);
    }
}

// Subclasses can/have to override the following methods
protected IDatabaseConnection getConnection() throws Exception { ←⑧

    // Get a JDBC connection from Hibernate
    Connection con =
        ((SessionFactoryImpl)sessionFactory).getSettings()
            .getConnectionProvider().getConnection();

    // Disable foreign key constraint checking
    con.prepareStatement("set referential_integrity FALSE")
}

```

```

        .execute();
        return new DatabaseConnection( con );
    }

    protected abstract void prepareSettings(); ⑨
}

}

```

- ① 特定单元中的所有测试都使用相同的Hibernate SessionFactory。
- ② 子类可以定制在每种测试方法之前和之后执行的DBUnit数据库操作。
- ③ 子类可以定制哪个DBUnit数据集应该被用于它所有的测试方法。
- ④ Hibernate在测试集合的逻辑测试运行之前启动——注意，@BeforeTest仍然并非意味着是在每个测试方法之前。
- ⑤ 对于每个测试（子）类，DBUnit数据集都必须从XML文件中加载，并且所有的空标识都必须用真正的NULL替换。
- ⑥ 在每一种测试方法之前，用DBUnit执行必要的数据库操作。
- ⑦ 在每一个测试方法之后，用DBUnit执行所需的数据库操作。
- ⑧ 默认情况下，从Hibernate的ConnectionProvider中获得一个简单的JDBC连接，并把它包装在DBUnit DatabaseConnection中。你还禁用对这个连接的外键约束检查。
- ⑨ 子类必须覆盖这个方法，并准备数据集文件位置，以及假定要在每种测试方案之前和之后运行的操作。

这个超类同时兼顾到了许多事情，并且将集成测试写成子类很容易。每个子类都可以定制它要使用哪一个DBUnit数据集（我们很快会讨论这些数据集），以及该数据集上的哪些操作（例如INSERT和DELETE）必须在特定的测试方法执行之前和之后运行。

注意，这个超类假设数据库是活动的，并且已经创建了一个有效的Schema。如果想要重新创建并自动给每个测试单元导出数据库Schema，就通过设置hibernate.hbm2ddl.auto配置选项为create来启用它。然后当创建SessionFactory时，Hibernate就删除旧的并导出新的数据库Schema。

接下来看一下DBUnit数据集。

2. 准备数据集

根据所建议的测试策略，每个测试（子）类都使用一个特定的数据集。这只不过是我们要简化超类的一个决定。你可以给每种测试方法使用一个数据集，或者为整个逻辑测试使用单个数据集，如果喜欢的话。

数据集是DBUnit可以替你维护的一个数据集合。在DBUnit中创建和使用数据集有许多种方法。我们想要介绍最容易的其中一种场景，通常这就足够了。首先，将数据集写入XML文件，用DBUnit要求的语法：

```

<?xml version="1.0"?>

<dataset>
    <USERS      USER_ID          ="1"
                  OBJ_VERSION     ="0"

```

```

        FIRSTNAME      = "John"
        LASTNAME       = "Doe"
        USERNAME       = "johndoe"
        PASSWORD       = "secret"
        EMAIL          = "jd@mail.tld"
        RANK           = "0"
        IS_ADMIN       = "false"
        CREATED        = "2006-09-23 13:45:00"
        HOME_STREET    = "[NULL]"
        HOME_ZIPCODE   = "[NULL]"
        HOME_CITY      = "[NULL]"
        DEFAULT_BILLING_DETAILS_ID = "[NULL]"
    />

<ITEM />
</dataset>

```

对于这个文件你不需要DTD，虽然指定DTD可以让你验证数据集的语法正确性（它也意味着必须把部分数据库Schema转变为DTD）。每个数据行都有它自己的元素，包含表名称。例如，一个`<USERS>`元素声明`USERS`表中的一个行数据。注意，你用`[NULL]`作为被集成测试超类用真正的SQL `NULL`替换掉的标记。还要注意，可以给想让DBUnit维护的每一张表都增加一个空行。在此处介绍的这个数据集中，`ITEM`表是数据集的一部分，并且DBUnit可以删除该表中的任何数据（这一点稍后会派得上用场）。

假设这个数据集保存在`auction.test.dbunit`包的XML文件`basedata.xml`中。接下来，你要编写一个利用这个数据集的测试类。

3. 编写测试类

测试类（test class）把依赖于特定数据集的测试方法进行分组。看一下以下示例：

```

public class PersistentStateTransitions
    extends HibernateIntegrationTest {

    protected void prepareSettings() {
        dataSetLocation = "auction/test/dbunit/basedata.xml";
        beforeTestOperations.add(DatabaseOperation.CLEAN_INSERT);
    }

    ...
}

```

这是`HibernateIntegrationTest`的一个子类，它准备所需的数据集位置。它还要求`CLEAN_INSERT`操作要在任何测试方法之前运行。这个DBUnit数据库操作删除所有的行（实际上清理了`USERS`和`ITEM`表），然后插入数据集中定义的行。因此每个测试方法都有了一个干净的数据库状态。

DBUnit包括许多内建的`DatabaseOperation`，如`INSERT`、`DELETE`、`DELETE_ALL`甚至`REFRESH`。查看DBUnit参考文档中完整的清单，我们不在此重复。注意你可以堆叠这些操作：

```

public class PersistentStateTransitions
    extends HibernateIntegrationTest {

    protected void prepareSettings() {

```

```

        dataSetLocation = "auction/test/dbunit/basedata.xml";
        beforeTestOperations.add(DatabaseOperation.DELETE_ALL);
        beforeTestOperations.add(DatabaseOperation.INSERT);
        afterTestOperations.add(DatabaseOperation.DELETE_ALL);
    }

    ...
}

```

在每种测试方法之前，数据集表中的所有内容都被删除，然后插入。在每个测试方法之后，数据集表中的所有内部都被再次删除。这个堆叠保证每个测试方法之前和之后都有干净的数据库状态。

现在可以在这个测试类中编写实际的测试方法了。类的名称PersistentStateTransaction，暗示着你想要做的事：

```

@Test(groups = "integration-hibernate")
public void storeAndLoadItem() {

    // Start a unit of work
    sessionFactory.getCurrentSession().beginTransaction();

    // Prepare the DAOs
    ItemDAOHibernate itemDAO = new ItemDAOHibernate();
    itemDAO.setSession( sessionFactory.getCurrentSession() );

    UserDaoHibernate userDao = new UserDaoHibernate();
    userDao.setSession( sessionFactory.getCurrentSession() );

    // Prepare a user object
    User user = userDao.findById(11, false);

    // Make a new auction item persistent
    Calendar startDate = GregorianCalendar.getInstance();
    Calendar endDate = GregorianCalendar.getInstance();
    endDate.add(Calendar.DAY_OF_YEAR, 3);

    Item newItem =
        new Item( "Testitem", "Test Description", user,
                  new BigDecimal(123), new BigDecimal(333),
                  startDate.getTime(), endDate.getTime() );
    itemDAO.makePersistent(newItem);

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

    // Direct SQL query for database state in auto-commit mode
    StatelessSession s = sessionFactory.openStatelessSession();
    Object[] result = (Object[])
        s.createQuery("select INITIAL_PRICE ip, "
                    + "SELLER_ID sid from ITEM")
        .addScalar("ip", Hibernate.BIG_DECIMAL)
        .addScalar("sid", Hibernate.LONG)
        .uniqueResult();
    s.close();
}

```

```

    // Assert correctness of state
    assert result[0].getClass() == BigDecimal.class;
    assert result[0].equals( newItem.getInitialPrice().getValue() );
    assert result[1].equals( 11 );

}

```

这个测试方法使Item实例变成了持久化。虽然这看起来好像有很多代码，但是只有几个部分值得关注。

这个状态转变需要User实例，因此你在数据集中定义的用户数据通过Hibernate加载。必须提供与你写入数据集作为主键的相同标识符值（在这个例子中为11）。

当工作单元提交时，所有的状态转变都完成，并且Session的状态与数据库同步。最后一步是真正的测试，断言数据库内容处于预期的状态。

可以用多种方法测试数据库状态。很显然，你不会出于这个目的而使用Hibernate查询或者Session操作，因为Hibernate是你的测试和实际的数据库内容之间一个额外的层。为了确保你真的命中数据库，并看到该状态，建议用SQL查询。

Hibernate使得执行SQL查询和检查返回的值变得很容易。在这个例子中，打开Hibernate StatelessSession来创建这个SQL查询。用在这个查询中的数据库连接处于自动提交模式（hibernate.connection.autocommit设置为true），因为你没有启动事务。这是StatelessSession的理想用例，因为它禁用了任何高速缓存、任何级联、任何拦截器，或者可能在数据库中干扰你看法的任何东西。

把所有这些都放到TestNG测试套件和Ant目标中。

4. 运行集成测试

这是XML测试套件描述符：

```

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="CaveatEmptor" verbose="2">

    <test name="PersistenceLayer">
        <groups>
            <run><include name="integration-hibernate.*"/></run>
        </groups>

        <packages>
            <package name="auction.test.dbunit"/>
        </packages>
    </test>

</suite>

```

逻辑测试PersistenceLayer包括在auction.test.dbunit包中设立的所有测试类和测试方法，如果它们的分组名称以integration-hibernate开头的话。对于任何TestNG配置方法（用@BeforeClass等作了标识的那些方法）也一样，因此你需要把任何包含配置方法的类（超类也是）放在同一个包中，并把它们添加到相同的分组。

为了用Ant运行这个测试套件，要替换掉你在16.5.2节中写在Ant目标中的XML单元描述符。在前面的示例中，我们只是接触到了TestNG和DBUnit的一点皮毛。它们还有更多有用的选择。

项，例如，可以在TestNG中用套件描述符中的任意设置将测试方法参数化。可以创建一个启动EJB 3.0容器服务器的测试集合（请见2.2.3节和下载的CaveatEmptor中的EJB3IntegrationTest超类），然后测试EJB层。当你利用介绍过的策略，从基类开始构建测试环境的时候，我们推荐相应的TestNG和DBUnit文档。

你可能想知道如何测试映射和查询，因为我们只讨论了对象—状态转变的测试。首先，可以通过设置`hibernate.hbm2ddl.auto`为`validate`来轻松地测试映射。然后当构建SessionFactory时，Hibernate通过把它们与数据库目录元数据进行对比，来验证映射。第二，测试查询与测试对象状态转变一样：编写集成测试方法，并断言所返回的数据的状态。

最后，考虑负载和压力测试，以及如果你想要测试系统的性能，还必须关注的一些方面。

16.5.4 考虑性能基准

企业应用程序开发中最难的事情之一是保证应用程序的性能和可伸缩性。先来定义这些术语。

性能（performance）通常被当作是基于应用程序的请求/响应的反应时间。如果单击一个按钮，希望半秒钟之内得到响应。或者，根据用例，你希望特定的事件（或者批量操作）可以在合理的时限之内得到执行。合理的响应时间一般取决于具体情况和某些应用程序功能的使用模式。

可伸缩性（scalability）是系统在更高的负载下合理执行的能力。想象一下，假设不是1个人单击1个按钮，而是5000人单击大量的按钮。系统的可伸缩性越好，就可以在它上面捆绑越多的并发用户，而不会降低性能。

关于性能已经讨论过很多了。在我们看来，创建一个执行得很好的系统，与创建一个没有明显性能瓶颈的Hibernate / 数据库应用程序同义。性能瓶颈可以是你认为的编程错误或者设计不当的任何东西——例如，错误的抓取策略、错误的查询，或者是Session和持久化上下文的错误处理。测试系统性能的合理与否通常是验收测试的一部分。实际上，性能测试经常由实验环境中一组专门的终端用户测试员来完成，或者通过现实世界条件下的少数用户群。纯粹的自动化性能测试很少。

也可以通过自动化的可伸缩性测试发现性能瓶颈，这是终极目标。然而，我们已经在自己的职业生涯中见过许多压力和负载测试，并且它们中大多数都没有考虑下列规则中的一个或者几个：

- 用现实世界的数据集测试可伸缩性。不要用可以完全适合数据库服务器中硬盘高速缓存的数据集进行测试。使用已经存在的数据，或者使用测试数据生成器来生成测试数据（例如，TurboData:<http://www.turbodata.ca/>）。确保测试数据尽可能地接近系统将在投产中运行的数据，包含相同的数量、分类和选择性。
- 用并发测试可伸缩性。测量单个活动用户进行单个查询所花费时间的自动化性能测试，并非在告诉你关于投产中系统可伸缩性的任何信息。持久化服务如Hibernate在设计时考虑了高并发，因此没有并发的测试更有可能显示出乎意料的过载！一旦启用更多的并发工作单元和事务，立即就会看见如二级高速缓存这样的特性如何帮助你保持性能。
- 用真实的用例测试可伸缩性。如果你的应用程序必须处理复杂的事务（例如，根据复杂的统计模型计算股票的市值），就应该通过执行这些用例来测试系统的可伸缩性。分析你

的用例，并选择普遍的场景——许多应用程序都只有几个最关键的用例。避免编写随机保存和加载几千个对象的微基准，从这种测试中得出的数字没有任何意义。

为可伸缩性测试的自动执行创建测试环境是其中一项工作。如果遵循我们所有的规则，就要先花点时间分析数据、用例，以及你期望的系统负载。一旦有了这些信息，就可以建立自动化测试了。

典型的客户端/服务器端应用程序的可伸缩性测试，一般需要并发运行客户端的模拟，以及每个被执行操作的统计集合。你应该考虑现有的测试解决方案，商业的（如LoadRunner，<http://www.mercury.com/>）或者开源的〔如Grinder（<http://grinder.sourceforge.net/>）或者JMeter（<http://jakarta.apache.org/jmeter/>）〕。创建测试通常涉及为模拟客户端编写控制脚本，以及配置在服务器进程（例如，为了特定的事务或者统计集合的直接执行）上运行的代理。

最后，测试系统的性能和（特别是）可伸缩性一般是软件应用程序生命周期中一个独立的阶段。你不应该在开发前期测试系统的可伸缩性。也不应该启用Hibernate的二级高速缓存，直到你具备根据我们提到过的规则而创建的测试环境。

在项目的后期，可以把自动化的可伸缩性测试增加到每日的集成测试中。应该在投产之前测试系统的可伸缩性，作为常规测试周期的一部分。另一方面，我们不主张将任何性能和可伸缩性测试延迟到最后一刻。不要试图在投产之前，通过调整Hibernate二级高速缓存，一天就修复好性能瓶颈。你可能不会成功。

把性能和负载测试当作开发过程中的必需部分，包含定义明确的阶段、准则和需求。

16.6 小结

本章探讨了分层应用程序和一些重要的模式和最佳实践。我们讨论了如何用Hibernate设计Web应用程序，并实现OSIV模式。你现在知道如何创建巧妙的领域模型，以及如何从控制器代码中分离业务逻辑，灵活的命令模式是软件设计工具库中最重要的工具。我们讨论了EJB 3.0组件，以及如何通过添加一些注解，来进一步简化POJO应用程序。

最后广泛地讨论了持久层，你用检验持久层的TestNG编写了数据访问对象和集成测试。

JBoss Seam简介

本章内容

- 利用JSF和EJB 3.0开发Web应用程序
- 利用Seam改善Web应用程序
- 利用Hibernate Validator整合Seam
- 利用Seam管理持久化上下文

在这最后一章中，我们介绍JBoss Seam框架。Seam是一个创新的新框架，用于通过Java EE 5.0平台进行的Web应用程序开发。Seam带来了两个新的标准，即JavaServer Faces（JSF）和EJB 3.0，通过统一它们的组件和编程模型，使二者关系更加密切。最吸引那些依赖Hibernate（或者EJB 3.0中的任何Java Persistence提供程序）的开发人员的是，Seam的自动化持久化上下文管理，以及它在应用程序流中给对话的定义所提供的一级构造。如果你已经在Hibernate应用程序中见过LazyInitializationException，Seam也有合适的解决方案。

关于Seam有更多的内容要讲，我们鼓励你阅读本章，即使你已经决定使用一种不同的框架，或者正在编写一个Web应用程序。虽然Seam目前的目标是Web应用程序，并且也依赖JSF作为表现框架，但是其他的选项在未来应该可用（例如，你可能已经使用Ajax调用来访问Seam组件了）。此外，Seam的许多重要概念目前正被标准化，并通过Web Beans JSR 299 (<http://www.jcp.org/en/jsr/detail?id=299>) 带回到Java EE 5.0平台。

阐述Seam与学习Seam一样，都有许多种方法。本章先介绍一下Seam承诺要解决哪些问题；然后讨论各种解决方案，并强调最吸引Hibernate用户的（比如你）是哪些特性。

17.1 Java EE 5.0 编程模型

Java EE 5.0明显比它之前的版本都更容易使用，也更加强大。与Web应用程序开发人员最相关的Java EE 5.0平台的两种规范是JSF和EJB 3.0。

JSF和EJB 3.0究竟好在哪里？我们先强调每种规范中的主要概念和特性，然后用JSF和EJB 3.0编写一个小示例，并将它与用Java（如Struts和EJB 2.x）编写Web应用程序的旧方法进行比较。最后关注依然存在的问题，以及Seam如何使JSF和EJB 3.0变成一个更强大且更方便的组合。

注意，本章不可能涵盖JSF和EJB 3.0的全部内容。建议把本章与Sun Java EE 5.0教程 (<http://java.sun.com/javaee/5/docs/tutorial/doc/>) 一起读，并且如果你想了解更多关于某个特定主题的内容，就浏览这个教程。另一方面，如果你已经接触过JSF或者EJB 3.0（甚至Hibernate），很可能发现学习Seam不过是小菜一碟。

17.1.1 JSF 详解

JSF简化了在Java中对Web用户界面的构建。作为表现框架，JSF提供了以下高级特性：

- JSF给可视的组件定义了一个可扩展的组件模型，经常称作小部件（widget）。
- JSF给backing bean或者被托管的bean（它们包含应用程序逻辑）定义了组件编程模型。
- JSF定义了用户界面和应用程序逻辑之间的交互，并允许你以灵活的方式把两者绑定在一起。
- JSF允许你用XML声明性地定义导航规则——也就是说，哪一页显示应用程序逻辑中的特定结果。

让我们多花一点时间讨论这每一种特性，以及是什么使它们变得大有用处。

JSF定义了一组内建的可视组件，这是每个JSF实现都必须支持的（例如按钮和输入文本字段）。这些可视组件在页面上呈现为HTML（和Javascript）。在编写本书之时，有几个高质量的开源和商业JSF小部件库可用。对于开发人员，现成的可视组件是很棒的。不必手工对它们编码，并且最重要的是，不必维护它们或者使它们在不同的浏览器上都有效（如果需要更多使用Javascript的复杂可视组件，这尤其痛苦）。

页面可以通过任何理解JSF小部件的HTML模板引擎来创建。虽然JSP似乎是一种显而易见的选择，但就经验来看，它并不是最好的。我们发现JavaServer Facelets (<http://facelets.dev.java.net/>) 完全适合于构建JSF视图和创建包含JSF小部件的HTML模板。（使用Facelets的另一个好处在于免费获得新的统一表达语言，甚至不用有JSP 2.1功能的servlet容器。）本章我们将在所有的JSF示例中使用Facelets。

JSF托管的应用程序组件（称作backing bean）使你的Web应用程序界面生效，它们包含应用程序代码。这些是一般的POJO，它们在JSF XML配置文件中定义并连接在一起。这种连接支持基本的依赖注入，以及backing bean实例的生命周期管理。backing bean的可用范围（它所处的位置）是当前的HTTP请求上下文、当前HTTP会话上下文和全局应用程序上下文。通过创建bean，并让JSF在其中一个上下文中管理它们的生命周期，来编写应用程序逻辑。

可以用一种表达式语言，将模型值从backing bean绑定到可视组件。例如，创建一个包含文本输入字段的页面，并把它绑定到具名的backing bean字段或者获取方法/设置方法对。然后这个backing bean名称就在JSF配置中被映射到实际的backing bean类，并声明该类的一个实例应该如何被JSF处理（在请求中，在HTTP会话中，或者在应用程序上下文中）。JSF引擎自动保持backing bean字段（或者属性）与小部件的状态同步，就像用户看到（或者操作）的一样。

JSF是事件驱动的表现框架。如果单击一个按钮，就会触发JSF ActionEvent，并传递到已注册的监听器。动作事件的监听器还是你在JSF配置中命名的backing bean。然后backing bean可以

对该事件做出响应——例如，通过将backing bean字段的当前值（它被绑定到一个文本输入小部件）保存到数据库中。这是对JSF用处的一种简化的阐述。在内部，来自Web浏览器的每个请求都跨过几个处理阶段。

当在JSF页面上单击一个按钮时，服务器上一个典型的请求处理顺序如下（这个过程如图17-7所示）：

- (1) 所有小部件的恢复视图 (Restore View) (JSF可以将小部件状态保存在服务器或者客户端上)。
- (2) 应用请求参数 (Apply Request Parameters)，更新小部件的状态。
- (3) 处理验证 (Process Validations)，这对于验证用户输入是必需的。
- (4) 更新模型值 (Update Model Values)，通过调用backing bean的绑定字段和设置方法，退回小部件。
- (5) 调用应用程序 (Invoke Application)，并将动作事件传递到监听器。
- (6) 呈现响应 (Render Response)，客户会看到该页面。

很显然，请求可以采用不同的子程序；例如，如果验证失败，呈现响应可能发生在处理验证之后。

在已经提过的Sun Java EE 5教程的第9章中可以找到JSF生命周期和处理阶段的一个很好的图例。本章稍后也将回到JSF处理模型的话题。

哪个响应得到呈现，以及哪个页面显示给用户，这取决于所定义的导航规则，以及动作事件的结果是什么。JSF中的结果是简单的字符串，如“success”或者“failure”。这些字符串通过backing bean产生，然后在JSF XML配置文件中被映射到页面。这也被称作自由导航流 (free navigation flow)，例如，可以在浏览器中单击后退按钮，或者通过输入页面的URL直接跳到该页面。

如果正在寻找一种Web框架，JSF与Facelets的组合就是一种很好的解决方案。另一方面，Web应用程序的backing bean——实现应用程序逻辑的组件——通常需要访问事务的资源（大多数时候为数据库）。这就是EJB 3.0应运而生的地方。

17.1.2 EJB 3.0 详解

EJB 3.0是一个给事务的组件定义编程模型的Java EE 5.0标准。对于Web应用程序开发人员来说，EJB 3.0的下列特性最值得关注：

- EJB 3.0定义了在简单的Java类中主要基于注解的组件编程模型。
- EJB 3.0定义了无状态、有状态和消息驱动的组件，以及运行时环境如何管理组件实例的生命周期。
- EJB 3.0定义了组件如何被连接在一起，如何获得对组件的引用，以及组件如何互相调用。
- EJB 3.0定义横切关注点如何被处理，例如事务和安全性。也可以编写定制拦截器，并用它们把组件包装起来。
- EJB 3.0标准化了Java Persistence，以及如何通过自动的和透明的ORM来访问SQL数据库。如果想要访问SQL数据库，就创建领域模型实体类（例如Item、User、Category），并通过

注解将它们从Java Persistence规范映射到数据库Schema。EJB 3.0持久化管理器API（EntityManager），现在成了数据库操作的通道。

在EJB 3.0组件中执行数据库操作——例如，有状态或者无状态会话bean。这些bean都是简单的Java类，你用一些注解将它们启用为EJB。然后在组件方法中获得容器的服务，例如自动的依赖注入（当你需要的时候就得到EntityManager）和声明性事务划分。例如，如果用户必须在应用程序的一个对话中浏览几个页面，有状态会话bean就会帮助你为特定的客户端保持状态。

可以用EJB 3.0组件和实体作为JSF动作和小部件的backing bean吗？可以将JSF文本字段小部件绑定到Item实体类中的字段吗？JSF按钮单击可以被直接发送到会话bean方法吗？

让我们用一个例子来试试。

17.1.3 用JSF和EJB 3.0编写Web应用程序

你要创建的Web应用程序很简单：它有一个搜索屏，用户可以在那里给特定的货品输入标识符，还有当在数据库中找到货品时所显示的详细信息屏幕。在这个详细信息屏幕中，用户可以编辑货品的数据，并将所做的改变保存到数据库。

（在阅读这些示例的时候不一定非要编写这个应用程序，稍后将通过引入Seam，进行重大的改进。那时候就可以开始编码了。）

从实体Item的数据模型开始。

1. 创建实体类和映射

Item实体类来自CaveatEmptor。它也已经被注解和映射到了SQL数据库（见代码清单17-1）。

代码清单17-1 一个被注解和映射的实体类

```
package auction.model;
import ...;

@Entity
@Table(name = "ITEM")
public class Item implements Serializable {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id = null;

    @Column(name = "ITEM_NAME", length = 255,
             nullable = false, updatable = false)
    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name="SELLER_ID",
                nullable = false, updatable = false)
    private User seller;

    @Column(name = "DESCRIPTION", length = 4000, nullable = false)
    private String description;

    @Column( name="INITIAL_PRICE", nullable = false)
```

```
private BigDecimal initialPrice;  
Item() {}  
// Getter and setter methods...  
}
```

这是CaveatEmptor Item实体的一个简化版，没有任何集合。接下来是允许用户搜索货品对象的搜索页面。

2. 用Facelets和JSF编写搜索页面

应用程序的搜索页面是用作为模板引擎的Facelets写成的一个页面，它是有效的XML。JSF小部件被嵌入在该页面中，通过它的输入字段和按钮来创建搜索表单（见代码清单17-2）。

代码清单17-2 利用Facelets编写XHTML格式的search.xhtml页面

```
<!DOCTYPE html PUBLIC ←①  
        "-//W3C//DTD XHTML 1.0 Transitional//EN"  
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml" ←②  
      xmlns:ui="http://java.sun.com/jsf/facelets"  
      xmlns:h="http://java.sun.com/jsf/html"  
      xmlns:f="http://java.sun.com/jsf/core">  
  
<head>  
    <title>CaveatEmptor - Search items</title>  
    <link href="screen.css" rel="stylesheet" type="text/css"/> ←③  
</head>  
  
<body>  
  <ui:include src="header.xhtml"/> ←④  
  
  <h:form> ←⑤  
    <span class="errors"><h:message for="itemSearchField"/></span> ←⑥  
    <div class="entry">  
      <div class="label">Enter item identifier:</div> ←⑦  
      <div class="input">  
        <h:inputText id="itemSearchField" ←⑧  
                      size="3" required="true"  
                      value="#{itemEditor.itemId}">  
          <f:validateLongRange minimum="0"/> ←⑨  
        </h:inputText>  
      </div>  
    </div>  
  
    <div class="entry">  
      <div class="label">←⑩</div>  
      <div class="input">  
        <h:commandButton value="Search" styleClass="button"  
                      action="#{itemEditor.doSearch}">  
      </div>
```

```

</div>
</h:form>
</body>
</html>

```

- ① 每一个有效的XHTML文件都需要正确的文档类型声明。
- ② 除了常规的XHTML命名空间之外，你还给可视的HTML组件和核心的JSF组件（例如对输入验证）导入Facelets和两个JSF命名空间。
- ③ 页面布局通过外部化到独立文件的CSS进行处理。
- ④ 一般的页面头模板通过<ui:import>从Facelets导入。
- ⑤ JSF表单（注意h命名空间）是一个HTML表单，如果被提交的话，则通过JSF servlet进行处理。
- ⑥ JSF可以输出消息，例如验证错误。
- ⑦ 每一个<div>都是一个标签或者表单字段，通过CSS类label或者input定义样式。
- ⑧ 这是呈现为HTML输入字段的JSF输入文本组件。标识符对于将它绑定到错误消息输出很有用处，其大小定义了输入字段的可视大小，并且提交这张表单时，需要用户输入。最值得关注的部分是将输入字段的值绑定到一个backing bean（具名itemEditor）和该backing bean中的一个获取方法/设置方法对（具名getItemId()/setItemId()）。这个输入字段被绑定到该数据模型，并且JSF自动地将变化同步。
- ⑨ JSF也支持输入验证，并提供全套的内建验证器。此处你声明了用户输入不能为负数（货品标识符是正整数）。
- ⑩ 表单的提交按钮有一个动作绑定(action binding)，绑定到具名itemEditor的backing bean的方法doSearch()。动作执行之后发生什么，取决于该方法的结果。

以下是页面在浏览器中呈现后的样子（见图17-1）。

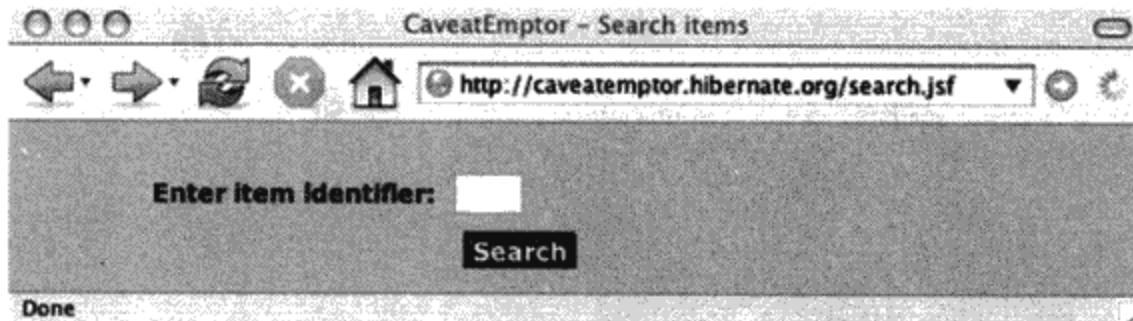


图17-1 利用JSF小部件的搜索页面

如果看一下这个URL，就会发现页面名称已经加上了.jsf的后缀，你或许原以为是search.xhtml。.jsf后缀是一个servlet映射；每当调用以.jsf结尾的URL时都会运行JSF servlet，并在安装完Facelets之后，在web.xml中将它配置为内部使用.xhtml。换句话说，search.xhtml页面通过JSF servlet得到呈现。

如果单击Search（搜索）按钮而没有输入搜索值，页面上就会出现错误消息。如果试图输入

一个非整数或者非正整数的值时，也会出现错误消息，这些全部由JSF自动进行处理。

如果输入一个有效的货品标识符值，并且backing bean在数据库中找到了该货品，就会把你转到货品编辑屏幕。（让我们在关注backing bean中的应用程序逻辑之前完成用户界面。）

3. 编写编辑页面

编辑页面显示了已经搜索到的货品详细信息，并允许用户编辑这些信息。当用户决定保存他的修改时，并在所有的验证都成功之后，应用程序再次显示搜索页面。

编辑页面的源代码如代码清单17-3所示。

代码清单17-3 包含详细表单信息的edit.xhtml页面

```

<!DOCTYPE html PUBLIC
...
<html xmlns=
...
<head>
...
<body>
...
<h2>Editing item: #{itemEditor.itemId}</h2> ①
<h:form>
    <span class="errors"><h:messages /></span>
    <div class="entry">
        <div class="label">Name:</div>
        <div class="input">
            <h:inputText required="true" size="25" ②
                value="#{itemEditor.itemName}">
                <f:validateLength minimum="5" maximum="255"/>
            </h:inputText>
        </div>
    </div>
    <div class="entry">
        <div class="label">Description:</div>
        <div class="input">
            <h:inputTextarea cols="40" rows="4" required="true"
                value="#{itemEditor.itemDescription}">
                <f:validateLength minimum="10" maximum="4000"/>
            </h:inputTextarea>
        </div>
    </div>
    <div class="entry">
        <div class="label">Initial price (USD):</div>
        <div class="input">
            <h:inputText size="6" required="true"
                value="#{itemEditor.itemInitialPrice}">
                <f:converter converterId="javax.faces.BigDecimal"/>
            </h:inputText>
        </div>
    </div>
    <div class="entry">

```

```

<div class="label">&#160;</div>
<div class="input">
    <h:commandButton value="Save" styleClass="button"
                      action="#{itemEditor.doSave}"/>
</div>
</div>

</h:form>
</body>
</html>

```

3

- ① 可以把值绑定的表达式放在任何组件之外。在这个例子中，调用了itemEditor backing bean中的getItemId()方法，且返回值最终在HTML页面中。
- ② 值绑定再次被用来将输入文本字段绑定到backing bean中的获取方法/设置方法对（或者字段）。
- ③ 这个动作绑定引用了itemEditor backing bean中的doSave()方法。根据该方法的结果，要么再次显示页面（包含错误消息），要么将用户转到搜索页面。

图17-2展现了所呈现的页面。

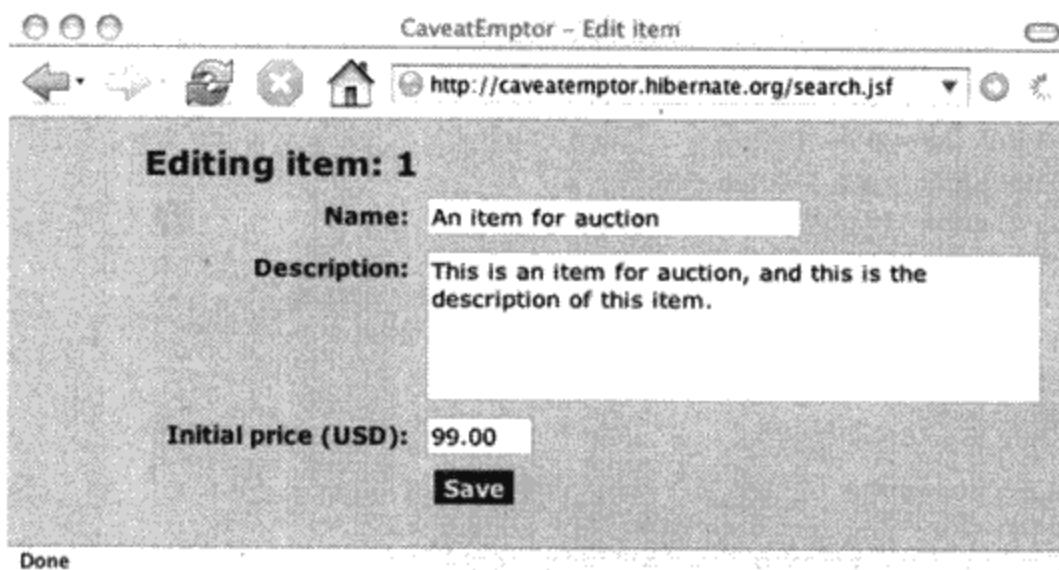


图17-2 加载有货品详细信息的编辑页面

为什么URL显示为search.jsf？它不是应该为edit.jsf吗？考虑JSF servlet的请求处理。如果用户在search.jsf页面中单击Search（搜索）按钮，backing bean的doSearch()方法就会在输入验证之后运行。如果该方法的结果触发而转到edit.xhtml页面，这个文档就通过JSF servlet得到呈现，并且HTML被发送到浏览器。URL没有改变！用户无法给编辑页面制作书签，它在这个简单的应用程序中可是很有必要的。

既然已经完成了应用程序的最顶层（视图），再来考虑访问数据库的层（你可能称它为业务层）。由于访问SQL数据库是一个事务的操作，因此要编写一个EJB。

4. 在EJB中访问数据库

如果之前已经使用过EJB 2.x（和Struts），那么在无状态会话bean中，访问数据库的代码最可

能是过程代码。让我们在EJB 3.0中来完成它（代码清单17-4）。

代码清单17-4 包含数据访问facade的无状态会话bean

```
package auction.beans;

import ...;
①
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EditItemBean implements EditItem {
    @PersistenceContext
    EntityManager em; ④
    ③
    public Item findById(Long itemId) { ⑤
        return em.find(Item.class, itemId);
    }
    public Item save(Item item) { ⑥
        return em.merge(item);
    }
}
```

① @Stateless注解把这个简单的Java类变成了一个无状态的会话bean。在运行时，会准备好一个实例池，且请求会话bean的每个客户端都从该池中获得实例来执行方法。

② 在这个会话bean中调用的所有方法都包装在一个系统事务中，它汇集了该过程期间可能用到的所有事务资源。即使你没有注解这个bean，它也是默认的。

③ 会话bean需要一个接口。你通常直接实现这个接口。EditItem接口有两个方法。

④ 当运行时容器从池中分发会话bean实例时，它就注入了EntityManager，并且把（新的）持久化上下文绑定到事务。

⑤ 如果客户端调用findById()，就启动系统事务。EntityManager操作在该事务中执行一个SQL查询；当事务提交（方法返回）时，持久化上下文被清除并关闭。返回的Item实体实例处于脱管状态。

⑥ 如果客户端调用save()，就启动系统事务。给定的脱管实例被合并到一个（新的）持久化上下文。在这个脱管的Item实例上所做的任何改变都被清除并提交到数据库。返回指向目前最新Item实例的新句柄。当方法返回时，这个新Item实例再次处于脱管状态，且持久化上下文也关闭了。

你可以称代码清单17-4中所示的会话bean为数据访问对象（data access object, DAO）。它也可以是会话外观。这个应用程序还不够复杂，难以进行清楚的区分；如果更多的非数据访问方法被添加到它的接口，会话bean就将作为业务层的一部分与传统的（主要是过程的）会话外观进行交互。

仍然漏掉了一个疑点：JSF输入小部件和按钮都有值和动作绑定到backing bean。这个backing bean与会话bean一样，还是你必须另外编写一个类呢？

5. 利用backing bean连接各个层

如果没有Seam，就必须编写一个将JSF 小部件状态和动作连接到事务的无状态会话bean的backing bean。这个backing bean有着在页面中通过表达式引用的获取方法和设置方法。它也可以与会话bean进行对话，并执行事务的操作。backing bean的代码如代码清单17-5所示。

代码清单17-5 一个JSF backing bean组件连接多个层

```
package auction.backingbeans;
import ...
public class ItemEditor {
    private Long itemId;
    private Item item;

    public Long getItemId() {
        return itemId;
    }

    public void setId(Long itemId) {
        this.itemId = itemId;
    }

    public String getName() {
        return item.getName();
    }

    public void setName(String itemName) {
        this.item.setName(itemName);
    }

    public String getDescription() {
        return item.getDescription();
    }

    public void setDescription(String itemDescription) {
        this.item.setDescription(itemDescription);
    }

    public BigDecimal getInitialPrice() {
        return item.getInitialPrice();
    }

    public void setInitialPrice(BigDecimal itemInitialPrice) {
        this.item.setInitialPrice(itemInitialPrice);
    }

    public String doSearch() {
        item = getEditItemEJB().findById(itemId);
        return item != null ? "found" : null;
    }

    public String doSave() {
        item = getEditItemEJB().save(item);
        return "success";
    }
}
```

The diagram illustrates the connections between the `ItemEditor` backing bean and various layers. Numbered arrows point from the code annotations to specific parts of the code:

- Annotation 1:** Points to the `itemId` field.
- Annotation 2:** Points to the `item` field.
- Annotation 3:** Points to the `getInitialPrice()` method.
- Annotation 4:** Points to the `setInitialPrice()` method.
- Annotation 5:** Points to the `doSearch()` method.
- Annotation 6:** Points to the `doSave()` method.

```

private EditItem getEditItemEJB() {
    try {
        return (EditItem)
            new InitialContext()
                .lookup("caveatemptor/EditItemBean/local");
    } catch (NamingException ex) {
        throw new RuntimeException(ex);
    }
}

```

7

- ① 你没有实现任何接口，这是一个简单的Java类。
- ② backing bean内部利用一个字段维护货品标识符。
- ③ backing bean也保存着正被用户编辑的Item实例。
- ④ 这些是绑定在search.xhtml和edit.xhtml中的所有值的获取方法和设置方法。它们都是被JSF用来使backing bean内部模型状态与UI小部件状态同步的方法。
- ⑤ doSearch()方法被绑定到一个JSF按钮的动作。它在backing bean中用EJB会话bean组件查找Item实例中当前的itemId。它的结果为字符串found或者null。
- ⑥ doSave()方法被绑定到一个JSF按钮的动作。它用EJB会话bean组件来保存item字段的状态。[由于这是一个合并，你必须用返回的值（合并之后的状态）更新item字段。]它的结果为字符串success或者异常。
- ⑦ 辅助方法getEditItemEJB()在EJB会话bean中获得一个句柄。如果运行时环境支持Java Servlet 2.5规范，JNDI中的这个查找可以用自动的依赖注入来替换。（在编写本书之时，Tomcat 5.5还只实现Java Servlets 2.4，Tomcat 6还处在alpha阶段。）

backing bean是一个由JSF运行时托管的组件。你在这些页面中使用的表达式通过名称itemEditor指向一个backing bean。在JSF XML配置文件中（通常是WEB-INF/faces-config.xml），你将这个名称映射到backing bean类（见代码清单17-6）。

代码清单17-6 一个JSF配置描述backing bean和导航流

```

<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd" >

<faces-config>
    <managed-bean>
        <managed-bean-name>itemEditor</managed-bean-name>
        <managed-bean-class>
            auction.backingbeans.ItemEditor
        </managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
    <navigation-rule>

```

```

<from-view-id>/search.xhtml</from-view-id>
<navigation-case>
    <from-outcome>found</from-outcome>
    <to-view-id>/edit.xhtml</to-view-id>
</navigation-case>
</navigation-rule>

<navigation-rule>
    <from-view-id>/edit.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/search.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>

</faces-config>

```

这个应用程序有1个backing bean和2个导航规则。backing bean通过itemEditor名称进行声明，并由auction.backingBeans.ItemEditor实现。JSF页面中的表达式现在可以在backing bean中以一种松散耦合的方式按名称引用方法和字段。JSF servlet管理backing bean的实例，每个HTTP会话一个实例。

更进一步：JSF页面中的表达式是一个字符串，例如#{itemEditor.itemId}。这个表达式基本上导致对具名itemEditor变量的搜索。以这样的顺序进行搜索：先是当前的请求，然后是当前的HTTP会话，最后是当前的应用程序上下文。如果JSF页面呈现且这个表达式必须取值，那么或者在HTTP会话上下文中找到包含该名称的变量，或者JSF servlet创建新backing bean实例，并将它绑定到HTTP会话上下文中。

导航规则声明哪个页面在一个动作结果之后进行呈现。这是一个从字符串（由动作返回）到页面的映射。

现在应用程序完成了，下面对它进行深入的分析。

17.1.4 分析应用程序

可能你会看着前几节中的代码想：“有这么多代码要写，要把四个表单字段放到Web页面上，并把它们连接到数据库中的四个列上。”或者，如果你已经在EJB 2.x和Struts上花了很多时间，可能会说：“太好了，我不必再自己管理HTTP会话了，并且所有的EJB样板代码也都不见了。”

这两种说法都对。Java EE 5.0，尤其JSF和EJB 3.0都是朝着Java Web应用程序前进的一大步，但是并非一切都完美无瑕。现在来看一下Java EE 5.0的优势，并把它与J2EE 1.4和JSF之前的Web框架进行比较。但我们也努力发现可以改善的东西、可以避免的代码，以及可以被简化的策略。这就是后来Seam出现的契机。

1. 将代码与J2EE进行对比

如果你有J2EE 1.4/Struts的背景，这个JSF和EJB 3.0应用程序看起来已经比较吸引人了。要创建的东西比传统Java Web应用程序中的更少——例如，可以从会话bean外观中脱管一个Item实

例，并把它转换到JSF backing bean里面。利用EJB 2.x实体bean时，你还需要数据迁移对象(DTO)来完成这个。

代码更加简洁。利用EJB 2.x时，会话bean必须实现SessionBean接口，并实现所有的维护方法。在EJB 3.0中，这将通过简单的@Stateless注解进行解析。也没有手工将HTML表单字段的状态绑定到动作监听器中实例变量的Struts ActionForm代码。

整体而言，应用程序是透明的，没有在HTTP会话或者HTTP请求中维持值的模糊调用。JSF透明地在这些上下文中放置和查找值。

如果你考虑Item类的对象/关系映射，可能会认同说：POJO中的一些注解比EJB 2.x实体bean的部署描述符更简单。甚至，Java Persistence定义的对象/关系映射不仅比EJB 2.x实体bean更强大、更具有丰富的特性，而且还更容易使用（即使与原生的Hibernate比较）。

无法在简单的应用程序中展现JSF和EJB 3.0背后的威力。JSF是相当的灵活且可扩展，可以编写自己的HTML小部件，可以钩入JSF请求的处理阶段，甚至可以轻松地创建自己的视图层（如果你不想要Facelets的话）。EJB 3.0比EJB 2.x更容易掌控，它还具有之前标准的Java编程模型中从未有过的一些特性（例如拦截器和依赖注入）。

应用程序可以轻松地通过测试框架（如JUnit或者TestNG）进行测试。所有的类都是简单的Java类，可以实例化它们，手工设置（模拟）依赖，并运行一个测试过程。

无论如何，还是有改善的空间。

2. 改善应用程序

在这个JSF/EJB 3.0应用程序中突出的第一个东西是JSF backing bean。这个类做什么用呢？它是JSF需要的，因为你要将值和动作绑定到它的字段和方法上。但是代码没有完成任何有意义的事：它一个接一个地将任何动作传递到EJB。更糟糕的是，它是个包含最多代码行的工件。

你可能认为它从业务层解耦了视图。如果和一个不同的视图层（比如富客户端）一起使用EJB的话，这似乎有道理。但是，如果应用程序是一个简单的Web应用程序，backing bean则导致各层之间的耦合更加紧密。你对视图或者业务层所做的任何改变，也要求改变包含最多代码行的组件。如果想要改善代码，就去掉工件层，并移除backing bean。EJB没有理由不应该是backing bean。编程模型不应该强制你给应用程序分层（它也不应该限制你）。为了改善应用程序，你需要禁用工件层。

应用程序不会在几个浏览器窗口中工作。想象你在两个浏览器窗口中打开搜索页面，在第一个页面上搜索货品1，在第二个页面上搜索货品2。这两个浏览器窗口都给你显示一个编辑屏幕，包含货品1和货品2的详细信息。如果你改变了货品1并单击Save（保存），会发生什么呢？这些变化却改变了货品2！如果在第一个浏览器窗口中单击Save，你就是在HTTP会话中所呈现的状态下工作，那里是backing bean所处的地方。然而，backing bean不再保存货品1——现在的当前状态是第二个浏览器窗口（编辑货品2）的状态。换句话说，你启动了应用程序的两个会话，但是会话之间并非彼此隔离。HTTP会话不是并发对话状态的正确上下文，它在浏览器窗口之间共享。你无法轻易修正这一点。使这个（繁琐的）应用程序在几个浏览器窗口中工作，需要进行重大的架构改变。如今，用户们希望Web应用程序可以在几个浏览器窗口中进行工作。

应用程序泄露内存。当一张JSF页面首先试图解析itemEditor变量时，ItemEditor的一个新实例就被绑定到了HTTP会话上下文中的变量中。这个值从来不会被清除。即使用户在编辑屏幕上单击Save，backing bean实例也会保留在HTTP会话中，直到用户注销或者HTTP会话超时。想象一个更为复杂的应用程序，它有许多表单和许多backing bean。HTTP会话随着用户通过应用程序单击而增长，并且将HTTP会话复制到集群中的其他节点的成本，随着每一次单击而变得越来越昂贵。如果用户在用完应用程序的另一个模块之后回到货品搜索屏幕，那么表单中显示的还是旧数据。针对这个问题的一种解决方案是，在对话结束时，手工清除HTTP会话，但是完成这项工作没有很容易的方法。使用JSF和EJB 3.0时，你必须手工编写这个代码。依据我们的经验，手工处理HTTP会话中的变量和值，是那些难以追踪到的问题的一般根源。

应用程序流很难形象化和控制。如何知道单击Search按钮后会把你引向哪里？至少，你必须观察两个文件：backing bean（它返回字符串结果）和JSF XML配置文件（它定义为特定的结果显示的页面）。在浏览器中也有“后退”按钮曾经出现过的问题，这是有两个以上屏幕的任何对话中一个讨厌的问题。

还要考虑业务流程（business process）。如何将页面流定义为一个更大业务流程的一部分？想象搜索和编辑一件货品只是涉及更多步骤的业务流程中的唯一任务——例如，作为预览流程的一部分。在应用程序中，没有工具或者策略可以帮助你整合业务流程管理。如今，你可以不再忽略以应用程序作为其中一部分的业务流程了，你需要一个支持业务流程管理的编程模型。

最后，这个应用程序包括了太多的XML。应用程序中的元数据没什么可说的，但是它并非要全部都在一个XML文件中。如果XML文件中的元数据独立于代码而改变，将会很棒。对于导航规则可能是这样，但是对于backing bean的声明和它们所处的上下文来说，或许不可能。这种元数据随着应用程序的大小呈线性增长——每一个backing bean都必须在XML中进行声明。反之，应该将一个注解放在类中，表明：“我是JSF的一个backing bean，我处在HTTP会话（或者任何其他的）上下文里面。”你的类将不可能突然改变它的角色，而不改变类代码。

如果你同意这种分析，就会喜欢上Seam。

17.2 用 Seam 改善应用程序

如果使用Seam，前面编写好用来搜索和编辑Web项目的Web应用程序还可以改进。从基础的Seam特性开始说起：

- Seam使JSF backing bean变得没有必要。可以直接将JSF小部件值和动作绑定到EJB有状态和无状态会话bean。Seam引入了一个统一的组件模型：所有的类都可以通过注解转变成Seam组件。组件通过字符串表达式以松散耦合的方式被连接在一起。
- Seam自动引入新的上下文，并管理组件范围。这个富上下文模型包括对应用程序有意义的逻辑上下文，例如对话或者业务流程上下文。
- Seam引入一个有状态的编程模型，它对于对话很棒。包含Seam托管对话的有状态应用程序可以在多个浏览器窗口中工作，无需额外的工作。

这是对Seam功能的一个简短罗列，稍后你还会用到更多。让我们先创建一个基本对话的、

有状态的、简单的Seam应用程序。第一步是建立和配置Seam。

如果想要沿用示例的代码，就从<http://caveatemptor.hibernate.org>为Seam下载CaveatEmptor包，并在IDE中打开它。如果以后想要编写自己的Seam工程，这也是一个好的起点。

17.2.1 配置 Seam

图17-3展现了在接下来的几节中，你对Web应用程序进行改变之前和之后的文件。

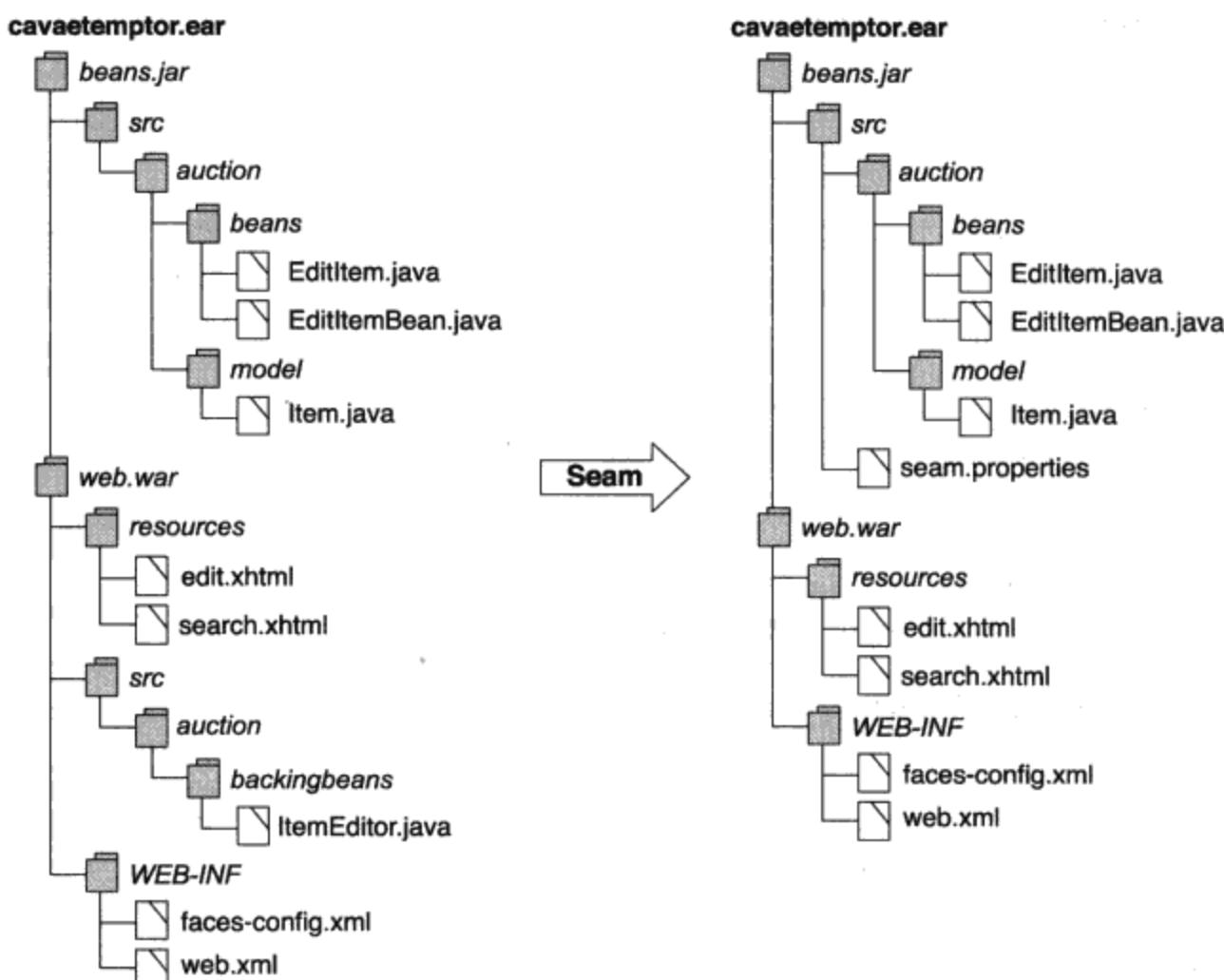


图17-3 引入Seam之前和之后的应用程序文档

发生了两处重大的变化：不再需要JSF backing bean，并且beans.jar文档有了一个新文件：seam.properties。这个文件给这个简单的应用程序包含了两个Seam配置选项（代码清单17-7）。

代码清单17-7 一个简单的seam.properties配置文件

```
org.jboss.seam.core.init.jndiPattern = caveatEmptor/#{ejbName}/local
org.jboss.seam.core.manager.conversationTimeout = 600000
```

第一个设置对于Seam与EJB 3.0容器的整合是必要的。因为现在Seam负责在运行时连接组件实例，它需要知道如何通过查找获得EJB。这里介绍的JNDI模式是用于JBoss应用程序服务器的。（Seam在任何Java EE 5.0服务器中运行，甚至可以在常规的Tomcat中运行（无论是否使用EJB 3.0）。我们认为从JBoss应用程序服务器开始最为方便，因为你不需要安装任何额外的服务。）

为了使Seam与EJB 3.0完全整合，Seam还需要拦截对EJB的所有调用。这很容易做到，因为EJB 3.0支持定制的拦截器。你不会在类的代码中看到任何拦截器，因为它们通常通过一个全局的通配符得到定义，该通配符匹配META-INF/ejb-jar.xml中的所有EJB（这里未显示出来）。如果下载了Seam实例，它就会有这个文件。

seam.properties中的第二个设置定义了Seam可以在600 000毫秒（10分钟）之后销毁非活动的用户对话。这个设置在用户决定去用餐时，释放HTTP会话中的内存。

seam.properties文件不仅是Seam的一个配置文件——它还是一个标识。当Seam启动时，它给Seam组件（包含正确注解的类）扫描类路径和所有的文档。但是，扫描所有JAR的成本太过昂贵，因此Seam只在根路径下递归地扫描包含seam.properties文件的JAR文件和目录。即使没有任何配置设置，在包含Seam组件类的文档中也要有一个空的seam.properties文件。

可以在web.xml和faces-config.xml中找到更多的Seam配置选项，以及与JSF和servlet容器的整合。稍后会回来讨论faces-config.xml，web.xml则没有什么值得关注的（请见CaveatEmptor包中注释的文件）。

Seam也可以通过components.xml文件在WAR包中的WEB-INF目录下进行配置。当你稍后需要更复杂的组件配置时会用到它。（许多Seam被写成Seam组件。字符串org.jboss.seam.core.manager是一个组件名称，conversationTimeout是你可以访问的属性，就像访问任何其他组件属性一样。）

下一步是用Seam组件替换JSF backing bean。

17.2.2 将页面绑定到有状态的Seam组件

search.xhtml页面根本没有改变；回顾代码清单17-2中的代码。这个页面有一个绑定到itemEditor.itemId的值和一个绑定到itemEditor.doSearch的动作。当页面由JSF servlet呈现时，这些表达式被取值，并且小部件被绑定到itemEditor bean中相应的方法上。

1. EJB组件接口

itemEditor bean现在是一个EJB。这个EJB的接口是EditItem.java（见代码清单17-8）。

代码清单17-8 一个有状态组件的接口

```
package auction.beans;

import ...

public interface EditItem {
    // Value binding methods
    public Long getItemId();
    public void setItemId(Long itemId);

    public Item getItem();

    // Action binding methods
    public String doSearch();
```

```

public String doSave();

// Cleanup routine
public void destroy();

}

```

前两个方法是用于页面的搜索输入文本字段的值绑定的获取方法和设置方法。getItem()方法（此处你不需要设置方法）稍后将为编辑页面所用。doSearch()方法被绑定到Search按钮，doSave()将被绑定到编辑页面中的一个按钮。

这是用于有状态组件的一个接口。当有状态组件第一次被请求时就被实例化——例如，因为页面第一次呈现。每一个有状态的组件都需要一个方法，当组件被销毁时，运行时环境可以调用它。可以使用doSave()方法，并假设组件的生命周期在这个方法完成时终止，但是你很快会明白为什么独立的方法更整洁。

接下来看一下这个接口的实现。

2. EJB组件实现

EJB 3.0中标准的有状态组件是一个有状态会话bean。EditItemBean.java中的实现是一个包含几个额外注解的POJO类。在代码清单17-9中，所有的Seam注解都以粗体显示。

代码清单17-9 一个有状态组件的实现

```

package auction.beans;

import ...

@Name("itemEditor")          ①
@Scope(ScopeType.CONVERSATION) ②

@Stateful           ③
public class EditItemBean implements EditItem {

    @PersistenceContext      ④
    EntityManager em;

    private Long itemId;        ⑤

    public Long getItemId() { return itemId; }
    public void setItemId(Long itemId) { this.itemId = itemId; }

    private Item item;
    public Item getItem() { return item; }

@Begin
public String doSearch() {
    item = em.find(Item.class, itemId);

    if (item == null)
        FacesMessages.instance().add(
            "itemSearchField",
            new FacesMessage("No item found."))
}

return item != null ? "found" : null;

```

```

    }
    7
@End
public String doSave() {
    item = em.merge(item);
    return "success";
}
}
8
@Destroy
@Remove
public void destroy() {
}

```

① Seam @Name注解给这个Seam组件声明名称。它也把这个EJB转变成一个Seam组件。现在可以用这个名称随处引用这个组件。

② 当需要这个组件的一个实例时，Seam就替你把它实例化。Seam把该实例放到一个以它为名的上下文中。这是正规的描述：EditItem的实例由Seam在对话上下文中托管，作为上下文变量itemEditor的一个值。

③ POJO需要EJB 3.0 @Stateful注解，来变成一个有状态的会话bean。

④ EJB 3.0容器用一个新的持久化上下文，在bean的任何客户端调用方法之前，将EntityManager注入到这个bean里面。当方法返回时关闭持久化上下文（假设这个方法调用也是系统事务的范围，它在这里是默认的）。

⑤ 这个有状态的组件在内部的字段itemId和item中保存着状态。状态通过属性访问方法公开。

⑥ Seam @Begin注解标识一个启动长运行对话的方法。如果一个JSF动作触发对这个方法的调用，Seam就跨越HTTP请求维持这个组件的状态。doSearch()方法返回字符串结果（或者null），并生成一个可以在页面上呈现的JSF消息。Seam FacesMessages辅助工具使得这个消息传递变得很容易。

⑦ Seam @End注解标识一个终止长运行对话的方法。如果一个JSF动作触发对这个方法的调用，并且方法返回，Seam将销毁组件的状态，并且不再跨越HTTP请求来维持它。

⑧ Seam @Destroy注解标识当组件状态必须被销毁（对话已经终止）时，由Seam调用的方法。这对于内部清除很有用（在这个例子中，没有东西可清除）。EJB 3.0 @Remove注解标识一个客户端（在这个例子中为Seam）必须调用来移除有状态会话bean实例的方法。这两个注解通常出现在同一个方法中。

为什么不用@End、@Destroy和@Remove标识doSave()呢？doSave()方法可能抛出异常，并且这个异常必须回滚任何系统事务。然而，Seam记录并吞没由它的@Destroy方法抛出的任何异常，因此你经常在有状态的Seam组件中看到空的销毁方法。此外，保存了一件货品之后，还需要组件实例一阵子，来呈现响应。

这个EJB实现封装了所有的应用程序逻辑，其他任何地方再没有Java代码（哦，还有Item实体类）。如果忽略繁琐的代码，应用程序逻辑在两个动作方法中就只有四行了。

还需要再做两处改变以使应用程序生效。edit.xhtml中的一些值绑定需要进行修改，并且定义旧JSF backing bean的XML块可以从faces-config.xml中移除。

3. 绑定值和动作

打开edit.xhtml，并改变JSF输入小部件的值绑定，如代码清单17-10中所示。

代码清单17-10 edit.xhtml页面被绑定到一个Seam组件

```

<h2>Editing item: #{itemEditor.itemId}</h2>
<h:form>
    <span class="errors"><h:messages/></span>
    <div class="entry">
        <div class="label">Name:</div>
        <div class="input">
            <h:inputText required="true" size="25"
                value="#{itemEditor.item.name}">
                <f:validateLength minimum="5" maximum="255"/>
            </h:inputText>
        </div>
    </div>
    <div class="entry">
        <div class="label">Description:</div>
        <div class="input">
            <h:inputTextarea cols="40" rows="4" required="true"
                value="#{itemEditor.item.description}">
                <f:validateLength minimum="10" maximum="4000"/>
            </h:inputTextarea>
        </div>
    </div>
    <div class="entry">
        <div class="label">Initial price (USD):</div>
        <div class="input">
            <h:inputText size="6" required="true"
                value="#{itemEditor.item.initialPrice}">
                <f:converter converterId="javax.faces.BigDecimal"/>
            </h:inputText>
        </div>
    </div>
    <div class="entry">
        <div class="label">&#160;</div>
        <div class="input">
            <h:commandButton value="Save" styleClass="button"
                action="#{itemEditor.doSave}">
        </div>
    </div>
</h:form>
...

```

改变了的绑定是关于名称、描述和初始价格输入字段的表达式。它们现在引用itemEditor.

item，后者可以被解析到Seam组件的getItem()方法。JSF在返回的Item实体中调用getName()和setName()，来使小部件的状态同步。同样的技术用于绑定和同步该货品的描述和初始价格。当用户输入新价格时，由itemEditor组件保存的Item实例的initialPrice被自动更新。

对Save按钮的动作绑定没有变——itemEditor组件的doSave()方法仍然是正确的监听器。可以看到逻辑组件名称和表达语言如何允许你轻松地耦合视图和业务层，并且不会太紧。

最后，更新faces-config.xml，如代码清单17-11所示。

代码清单17-11 没有backing bean的JSF配置文件

```
<faces-config>
    <navigation-rule>
        <from-view-id>/search.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>found</from-outcome>
            <to-view-id>/edit.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/edit.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/search.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <!-- Integrate Seam with the JSF request processing model -->
    <lifecycle>
        <phase-listener>
            org.jboss.seam.jsf.SeamPhaseListener
        </phase-listener>
    </lifecycle>
</faces-config>
```

将这段代码与代码清单17-7中的前一个JSF配置进行对比。backing bean声明不见了（移到了EJB中的两个Seam注解）。阶段监听器是新的：Seam必须钩入JSF servlet，并监听每个HTTP请求的处理。定制的JSF阶段监听器将Seam与JSF进行整合。

上面介绍了一些新的概念，如果这是你初次接触Seam，这些概念你可能还从未见过。下面来更深入地分析一下应用程序，并看一看之前对简单的JSF和EJB 3.0应用程序所确定的问题是否已经得到解决。

17.2.3 分析 Seam 应用程序

Web应用程序的接口还没有改变，它看起来一模一样。用户唯一可能注意到的东西是他们可以在几个浏览器窗口中搜索和编辑货品，而不会重叠状态和数据修改。

Seam推出了一个强大且确定的有状态应用程序编程模型。让我们顺着应用程序流（见图17-4），看看其内部的工作原理。

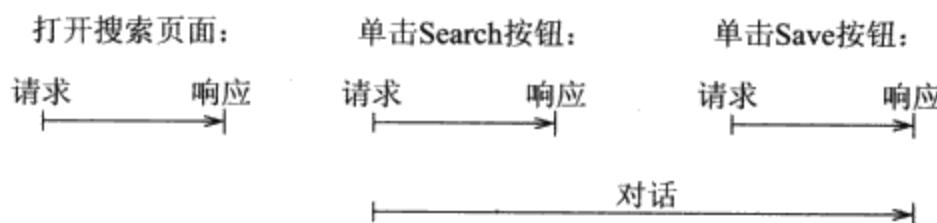


图17-4 应用程序的请求/响应流

1. 打开搜索页面

打开浏览器窗口，并输入/search.jsf URL时，一个HTTP GET请求就被发送到了JSF servlet。JSF处理生命周期开始了（见图17-5）。

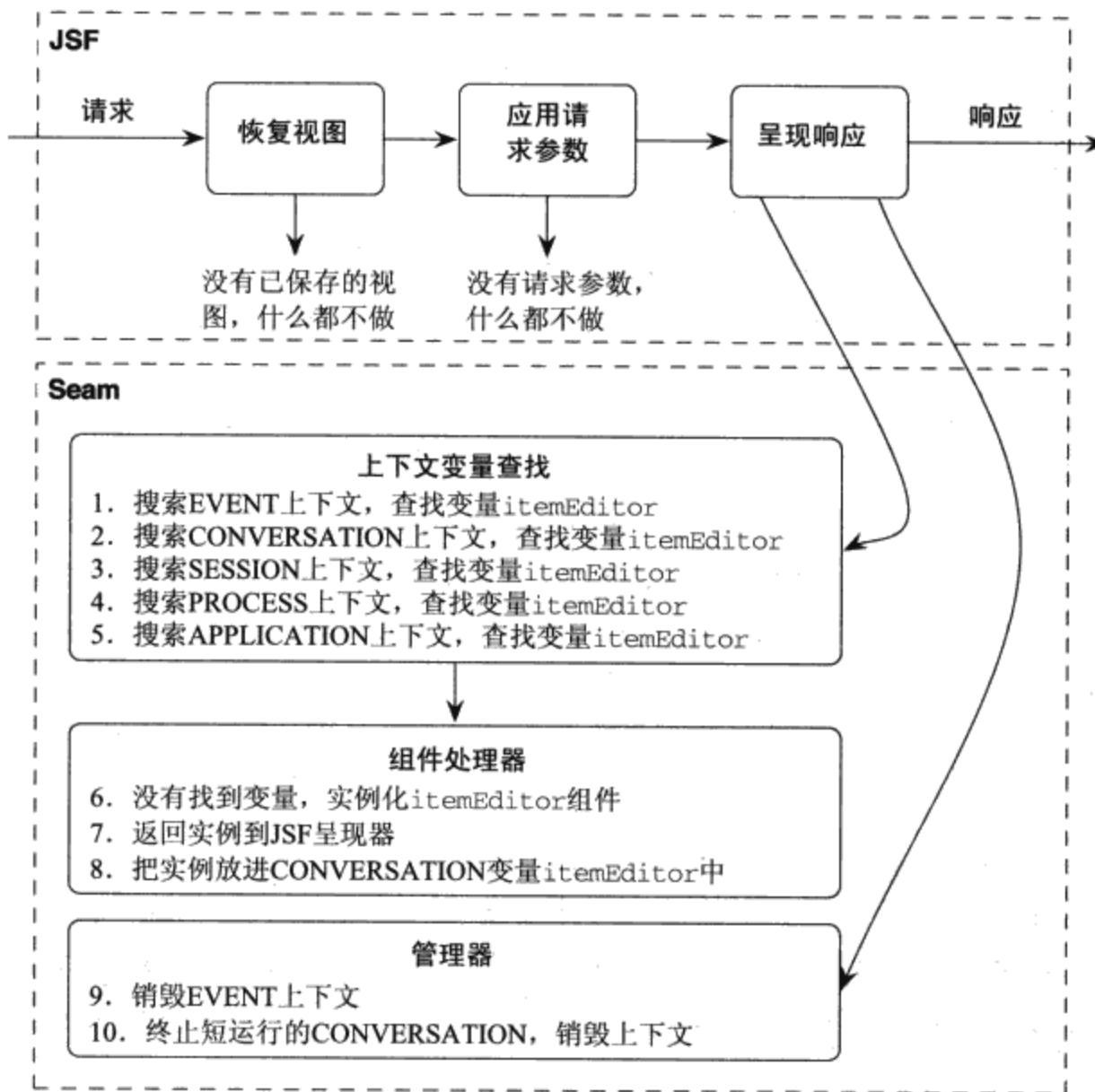


图17-5 当JSF呈现响应（搜索页面）时，Seam是活动的

直到JSF servlet进入请求处理的Render Response（呈现响应）阶段时，才会发生真正值得关注的事情。没有视图要恢复，并且没有必要应用到视图组件（小部件）的HTTP请求参数。

当响应（即search.xhtml文件）呈现时，JSF使用了一个变量解析器来给#{itemEditor.itemId}值绑定取值。Seam变量解析器比标准的JSF变量解析器更智能。它不在HTTP请求、HTTP会话和全局的应用程序上下文中查找itemEditor，而是在Seam逻辑上下文中查找。如果你认为这些逻

辑上下文是一样的就对了——我们很快会讨论关于这个话题的更多内容。至于现在，可以把Seam上下文当作被分层搜索〔从范围最窄的上下文（当前的事件），到范围最宽的上下文（当前的应用程序）〕的变量持有者。

无法找到变量itemEditor。因此，Seam的组件处理器开始寻找以该名字命名的Seam组件。它找到你编写过的有状态会话bean，并创建一个实例。然后这个EJB实例就被提供给JSF页面呈现器，呈现器把getItemId()的返回值放到搜索输入文本字段中去。方法返回null，因此第一次打开页面时字段为空。

Seam组件处理器也意识到有状态会话bean拥有@Scope(CONVERSATION)注解。因此实例被放入到对话上下文中，作为上下文变量itemEditor（组件的名称）的一个值。

当页面完全呈现时，Seam被再次调用（通过Seam阶段监听器）。Seam事件上下文有一个小范围：它只能在单个HTTP请求期间才能够保存变量。Seam销毁了这个上下文和其内部的一切（目前你不需要任何东西）。

Seam也销毁当前的会话上下文，以及随之刚刚创建的itemEditor变量。这可能让你感到很惊讶——你或许以为有状态会话bean对于几个请求会很好。然而，如果沒有人在该请求期间把它提升为长运行对话，对话上下文的范围就是单个的HTTP请求。通过调用一个已经用@Begin做了标识的组件方法，将一个短的单请求对话提升为长运行对话。在这个请求中没有发生这一点。

搜索页面现在通过浏览器显示，应用程序等待用户输入和单击Search按钮。

2. 搜索货品

当用户单击Search按钮时，一个HTTP POST请求就被发送到服务器，并由JSF（见图17-6）进行处理。必须看一下search.xhtml的源代码和EditItemBean，以便理解这个图例。

保存在前一个请求（通常在服务器的HTTP会话中）中的JSF，现在找到了一个表示视图（search.xhtml）的小部件树，并在内部对它进行重新创建。这个小部件树很小：它有一个表单、一个输入文本字段和一个提交按钮。在应用请求参数中，所有的用户输入都取自HTTP请求，并与小部件的状态同步。输入文本字段小部件现在保存着用户输入的搜索字符串。

提示 调试JSF小部件树——Facelets可以给你显示JSF 小部件树。把<ui:debug hotkey="D"/>放在你页面中的任何位置，并在浏览器中打开页面（当然是作为一个JSF URL）。现在按下Ctrl+Shift+d，打开一个包含JSF小部件/组件树的弹出式窗口。如果单击Scoped Variables，就会看到Seam把它的上下文和管理器保存在内部的什么地方（如果你不是Seam开发人员，这可能没有什么值得关注的）。

在处理验证期间，JSF验证器确保由用户输入的搜索字符串为非负整数，并且呈现输入值。如果验证失败，JSF servlet跳入呈现响应阶段，并再次利用错误消息呈现search.xhtml页面（这个阶段的处理看起来如图17-5）。

验证之后，JSF同步已经被绑定到小部件的模型对象的值。它调用itemEditor.setItemId()。这个变量由Seam解析，并在所有的Seam上下文中查找。由于沒有在任何上下文中找到itemEditor变量，所以创建新的EditItemBean实例，并放入对话上下文中。setItemId()

方法在这个有状态会话bean实例中被调用。

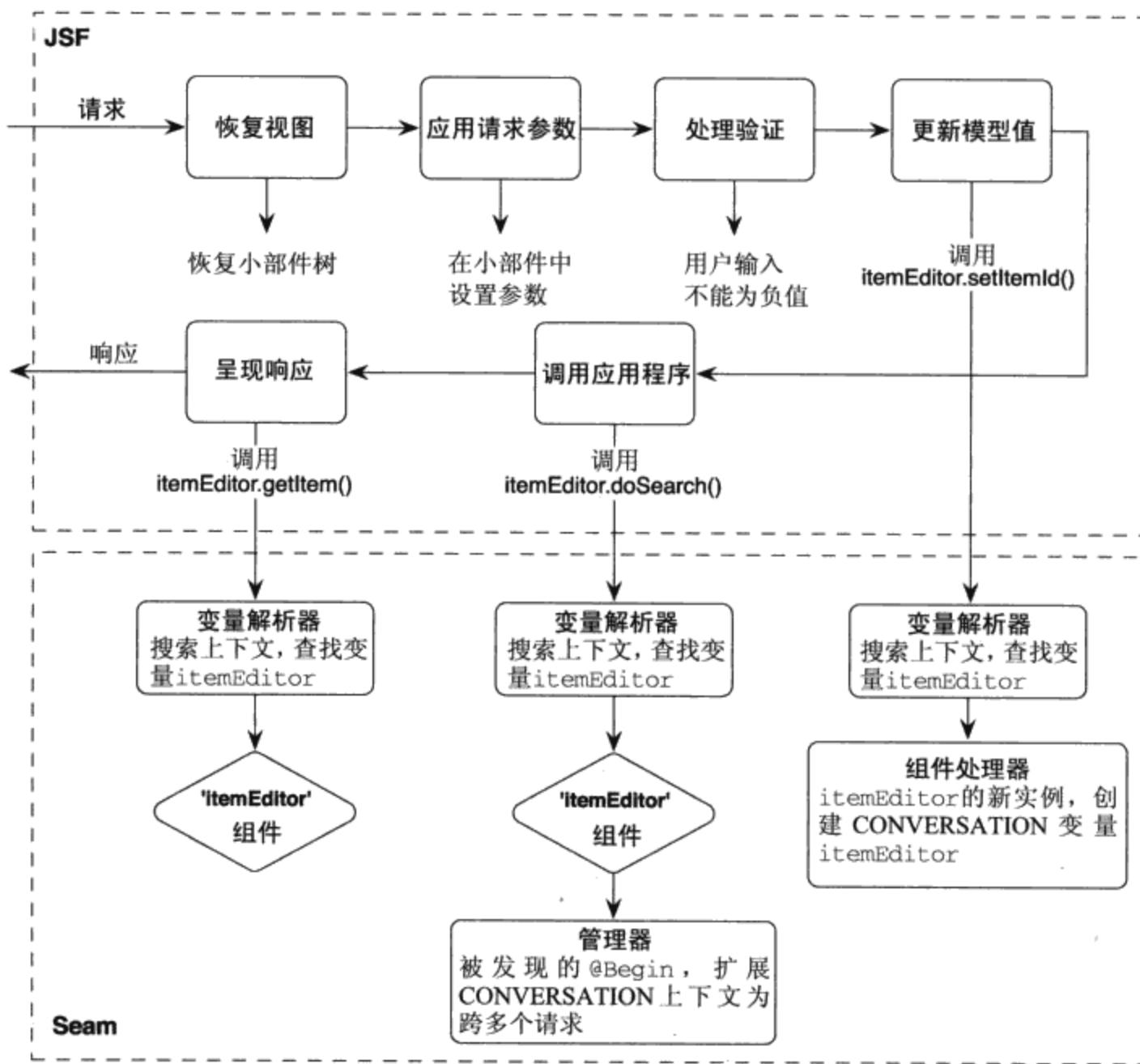


图17-6 Seam参与搜索动作的处理过程

JSF现在通过调用被绑定的方法 itemEditor.doSearch 执行请求的动作。Seam解析 itemEditor 变量，并在对话上下文中找到它。doSearch() 方法在 EditItemBean 实例中被调用，并且 EJB 3.0 容器在该调用期间处理事务和持久化上下文。调用期间发生了两件事：itemEditor 的 item 成员变量现在保存着在数据库中找到的一个 Item 实例（或者如果没有找到任何东西，就为 null），并且 @Begin 注解将当前的对话提升为一个长运行对话。对话上下文由 Seam 保存，直到调用一个标有 @End 注解的方法。

doSearch() 方法返回字符串 found，或者 null。这个结果由 JSF 取值，并应用来自 faces-config.xml 的导航规则。如果结果为 null，search.xhtml 页面就利用 Item not found（货品没有找到）的错误消息进行呈现。如果结果为 found，导航规则就声明 edit.xhtml 页面得到了呈现。

在 edit.xhtml 页面的呈现期间，变量 itemEditor 必须再次由 JSF 进行解析。Seam 在对话上下文中找到 itemEditor 上下文变量，且 JSF 将页面上小部件的值（文本输出，文本输入）绑定到由

`itemEditor.getItem()`返回的货品实例的属性中。

提示 浏览Seam上下文——如果使用Seam调试屏幕，就可以更轻松地调试Seam应用程序。必须启用这个屏幕。要这么做，得编辑你的`seam.properties`文件，并添加`org.jboss.seam.core.init.debug = true`。现在，访问URL `/debug.jsf`，来浏览这个浏览器窗口的Seam上下文。可以看到当前对话中的所有变量和值：会话、过程和应用程序上下文。

请求结束时，Seam销毁它的事件上下文。对话上下文没有被销毁；应用程序的用户通过执行搜索启动长运行对话。当显示编辑页面时，应用程序等待用户输入。如果用户再一次在另一个浏览器窗口中搜索，很快就会启动并发运行的对话，并把它提升为长运行对话。这两个对话和它们的上下文自动被Seam隔离。

3. 编辑货品

当用户单击Save时，编辑表单通过一个HTTP POST请求被提交到服务器（见图17-7）。

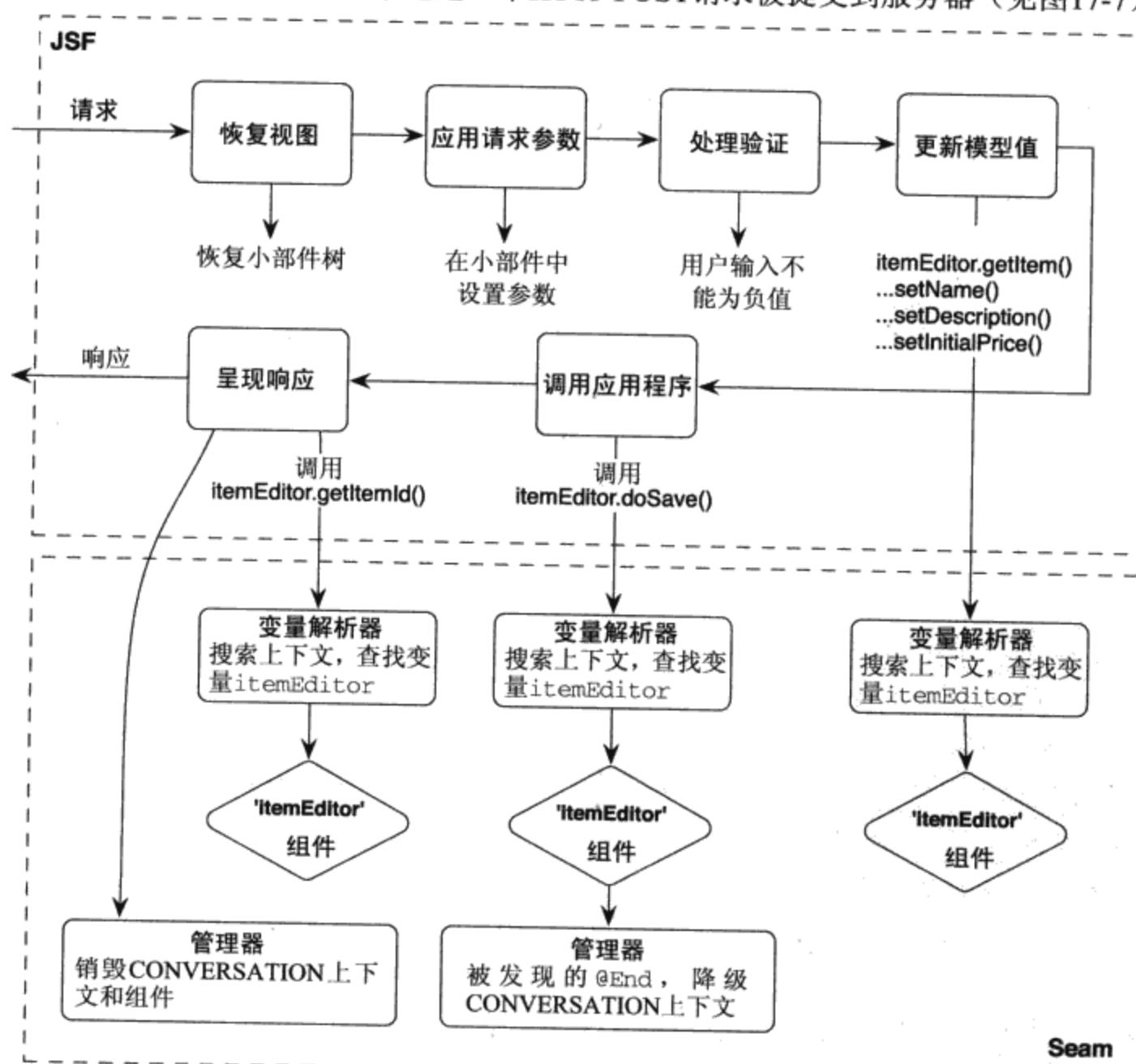


图17-7 Seam参与编辑动作的处理过程

在这个请求中恢复的视图是edit.xhtml，JSF重新创建了表单内部的一个小部件树和它所有的字段，并应用HTTP请求值。验证稍微复杂一些，你已经在edit.xhtml页面中定义了更多的JSF验证器。

验证成功之后，通过调用由itemEditor.getItem()返回的Item实例上的设置方法，JSF更新绑定的模型值。itemEditor绑定解析（通过Seam）到当前对话上下文中的一个上下文变量。Seam将会话上下文扩展到当前的请求中，因为它在前一个请求中被提升成为长运行对话了。

接下来，调用itemEditor.doSave()，变量再一次在会话上下文中被解析。EditItemBean中的代码抛出异常（如果EJB 3.0容器或者EntityManager抛出异常），或者返回字符串结果success。这个方法被标识为@End，因此Seam管理器标识在呈现响应阶段之后要清除的当前对话。

字符串结果success被映射到JSF导航规则中的/search.xhtml。在呈现响应期间，search.xhtml页面中的值绑定必须被解析。唯一的值绑定是#{itemEditor.itemId}，因此Seam再次试图在所有的上下文中找到itemEditor。使用来自（降了级但仍然活动着的）对话上下文的itemEditor，且getItemId()返回一个值。因此用户看到输入字段不是空的，而是显示了与对话开始时的输入相同的搜索值。

当呈现响应完成时，Seam移除了降了级的对话上下文，并销毁处在该上下文中的所有有状态的组件实例。destroy()方法在EditItemBean中被调用。因为它通过@Remove被标识，EJB 3.0容器也内部清除有状态会话bean。现在用户看到搜索页面，可以开始另一个对话了。

如果还未使用过JSF，这就有很多信息需要消化了。另一方面，如果你熟悉JSF，就会看到Seam基本上监听JSF servlet的处理阶段，并用一种更强大的变形给值和动作绑定替换变量解析器。

通过这个微不足道的应用程序，我们几乎无法触及Seam的九牛一毛。下面讨论Seam的一些更值得关注和更高级的特性，它们使得通过数据库后端创建复杂的Web应用程序变得很容易。

17.3 理解上下文组件

在前几节中，你已经把基础的JSF和EJB 3.0 Web应用程序变成了一个有状态的、对话的Seam应用程序。这么做减少了代码，并改善了应用程序的功能。你不应该到此为止——Seam还提供了更多的功能。

可以以上下文的方式把Seam组件组装在一起。这是一个强大的概念，可以对你如何设计有状态的应用程序有着深刻的影响。依据我们的经验，这就是Seam应用程序具有很少的几行简洁代码的主要原因之一。为了进行示范，我们来讨论如何创建新的应用程序功能。

几乎所有的Web应用程序都有一个登录/注销特性，以及登录用户的概念。假设应用程序的第一个页面（将强制为登录屏幕）一出现，用户必须立即登录到CaveatEmptor。支持它的登录屏幕和应用程序逻辑，是学习Seam组件如何在上下文中联结在一起的绝佳场景。

17.3.1 编写登录页面

用户看到如图17-8所示的登录屏幕。

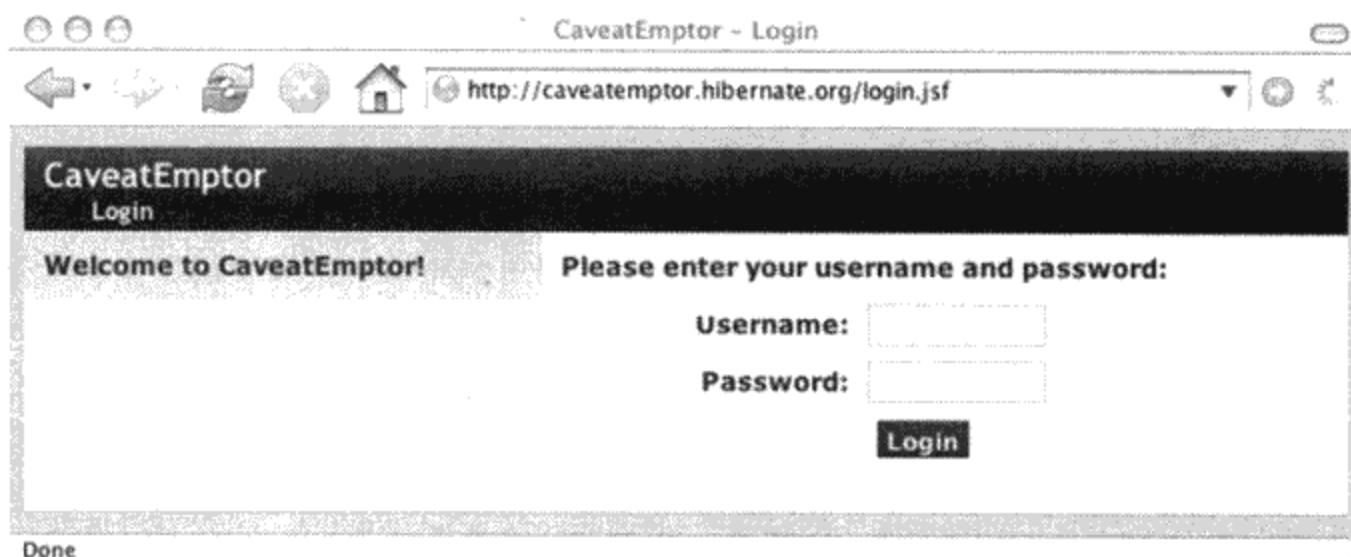


图17-8 CaveatEmptor的登录屏幕

这是一个称作login.xhtml的JSF页面，用Facelets编写的（代码清单17-12）。

代码清单17-12 login.xhtml页面源代码

```

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    template="template.xhtml">

    <ui:define name="screen">Login</ui:define>

    <ui:define name="sidebar">
        <h1>Welcome to CaveatEmptor!</h1>
    </ui:define>

    <ui:define name="content">
        <div class="section">
            <h1>Please enter your username and password:</h1>
            <h:form>
                <span class="errors"><h:messages/></span>
                <div class="entry">
                    <div class="label">Username:</div>
                    <div class="input">
                        <h:inputText required="true" size="10"
                            value="#{currentUser.username}">
                            <f:validateLength minimum="3" maximum="255"/>
                        </h:inputText>
                    </div>
                </div>
                <div class="entry">
                    <div class="label">Password:</div>
                    <div class="input">
                        <h:inputSecret required="true" size="10"

```

```

        value="#{currentUser.password}"/>
    </div>
</div>

<div class="entry">
    <div class="label">&#160;</div>
    <div class="input">
        <h:commandButton value="Login" styleClass="button"
            action="#{login.doLogin}"/>
    </div>
</div>

</h:form>

</div>
</ui:define>

</ui:composition>
```

在这段源代码中，你可以看到更多一点的Facelets：一个全局的页面模板是如何呈现的（在`<ui:composition>`标签中），以及这个模板（screen、sidebar、content）支持的代码片段是如何定义的。

页面的内容是常规的JSF表单，包含到具名组件的值绑定和动作绑定。登录表单的输入字段被绑定到`currentUser`的属性，且Login按钮被绑定到`login`组件的`doLogin()`方法。

当登录页面第一次呈现时，JSF试图解析值和动作绑定。它用Seam变量解析器查找被引用的对象。Seam变量解析器无法在任何Seam上下文中找到对象，因此创建`currentUser`和`login`的实例。来看一下这些组件的源程序。

17.3.2 创建组件

第一个组件是`currentUser`。这是在CaveatEmptor中已经具有的类：`User`实体类。可以把它转变成一个可以由Seam通过注解进行处理的组件：

```

package auction.model;

import ...

@Name("user")
@Role(name = "currentUser", scope = ScopeType.SESSION)

@Entity
public class User implements Serializable {

    @Id @GeneratedValue
    private Long id = null;

    private String firstname;
    private String lastname;
    private String username;
    private String password;

    ...

    public User() {}

}
```

第一个Seam注解@Name，将这个POJO转变成一个Seam组件。现在每当Seam查找一个名为user的组件，并且Seam上下文都未保存具有该名称的变量时，就会由Seam创建User的一个新的空实例，并将它放到以变量名称user为名的事件上下文中。事件上下文是实体Seam组件的默认上下文（有状态和无状态的bean有着不同的默认上下文）。

对于登录功能来说，在事件上下文中并不需要User实例，因此你给这个组件定义一个额外的角色。每当Seam查找名为currentUser的组件，并且Seam上下文没有包含这个名称的变量时，Seam就实例化一个User，并将它放到会话上下文中（HTTP会话）。可以在代码和元数据中的任何位置轻松地指向currentUser，并从Seam取回一个对当前User对象的引用。

当login.xhtml呈现时，Seam创建了一个新的User对象，并将它绑定到会话上下文中。这个用户还没有登录，你需要输入用户名和密码，并单击Login按钮。

这么做就在另一个请求中执行了login组件中的doLogin()方法。这个组件实现为无状态的会话bean，如代码清单17-13所示。

代码清单17-13 无状态的Seam组件实现登录和注销过程

```
package auction.beans;
import ...
@Name("login")
@Stateless
public class LoginBean implements Login {
    @In @Out
    private User currentUser;
    @PersistenceContext
    EntityManager em;
    @In
    private Context sessionContext;
    public String doLogin() {
        User validatedUser = null;
        Query loginQuery = em.createQuery(
            "select u from User u where" +
            " u.username = :uname" +
            " and u.password = :pword"
        );
        loginQuery.setParameter("uname", currentUser.getUsername());
        loginQuery.setParameter("pword", currentUser.getPassword());
        List result = loginQuery.getResultList();
        if (result.size() == 1) validatedUser = (User) result.get(0);
        if (validatedUser == null) {
            FacesMessages.instance().add(
                new FacesMessage("Invalid username or password!")
            );
            return null;
        } else {
```

```

        currentUser = validatedUser;
        sessionContext.set(LoggedIn.LOGIN_TOKEN, true);
        // or:
        Contexts.getSessionContext()
            .set(LoggedIn.LOGIN_TOKEN, true);
        return "start";
    }
}

public String doLogout() {
    Seam.invalidateSession();
    return "login";
}
}

```

这段代码有两个新的Seam注解：@In和@Out。这些都是变量别名，提示你用于组件联结。让我们在关注这些标签之前，先讨论剩下的代码。

doLogin()方法接受成员变量currentUser的用户名和密码，并试图在数据库中找到这个用户。如果找不到任何用户，就会等待JSF错误消息，并且结果null导致重新显示登录页面。如果找到了用户，就被分配到成员变量currentUser，替换该变量的旧值（目前为止，还没有改变HTTP会话中的任何东西）。

你还把一个令牌（一个简单的布尔）放进了会话上下文中，表明当前的User登录了。以后当需要测试当前的用户是否登录时，这个令牌很有用。稍后还将讨论doLogout()方法，它使当前的HTTP会话无效。

来看一下@In和@Out都有什么作用。

17.3.3 给上下文变量起别名

给上下文变量起别名听起来很复杂。然而，这是当你在Seam组件的代码中使用@In和@out时，对于正在进行的情况的正规描述。看一下图17-9，它显示了当用户单击Login之后，在调用应用程序请求处理阶段中所发生的事。

1. 在所有上下文中查找组件
 2. 没有找到，创建一个指向无状态组件的句柄
- ```

@Name("login")
@Stateless
public class LoginBean {
 @In
 @Out
 private User currentUser; 3. 从上下文变量注入值

 public String doLogin() {
 currentUser =
 validUser;
 ...
 }
 ...
}

```
4. 把新值分配给成员变量
  5. 把值从成员变量推进上下文变量

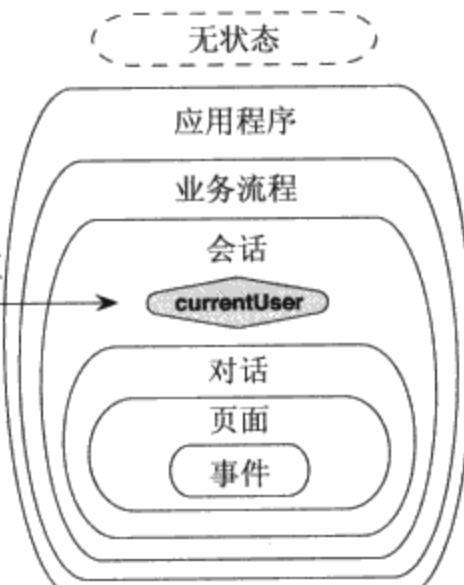


图17-9 Seam使一个成员别名与上下文变量同步

@In注解告诉Seam，你想要让一个值分配到这个组件的一个成员变量。这个值由Seam在该组件的方法被调用之前进行分配（Seam拦截每一个调用）。

这个值从哪里来的呢？Seam读取成员变量的名称，如前一个例子中的字段名称currentUser，并在它所有的上下文中查找同名的上下文变量。当doLogin()被调用时，Seam在会话上下文中找到currentUser变量。它采用这个值，并将它分配给组件的成员变量。这是对会话上下文中相同User实例的一个引用，你在组件的范围内创建了一个别名（alias）。然后可以在currentUser成员变量如getUsername()和getPassword()中调用方法。

@Out注解告诉Seam，你想要当组件的（任何）方法返回时，将一个值分配给上下文变量。上下文变量的名称是currentUser，与字段的名称相同。变量的上下文是currentUser Seam组件的默认上下文（在前一个例子中是会话上下文）。（还记得你在User类中分配的任务吗？）Seam采用成员变量的值，并将它放进上下文变量中。

再一次读取doLogin()方法。

在方法执行之前，Seam将上下文变量currentUser的值（在会话上下文中找到的）注入到同名的成员变量中去。然后方法执行并使用成员变量。成功登录之后（数据库查询），成员变量的值被替换了。这个新值必须被推回到上下文变量中去。方法执行之后，Seam将成员变量currentUser的值推进给这个组件定义的默认上下文（会话）。

不用字段，也可以用获取方法和设置方法对来起别名。例如，@In可以在setCurrentUser()中，@Out在getCurrentUser()中。在这两种情况下，被定义别名上下文变量的名称将是currentUser。

@In和@Out注解极为强大。本章后面会介绍更多的示例，但是我们需要更多的篇幅来描述你可以利用这些注解去做的所有事情。也请阅读Seam参考文档中的教程。

也可以直接利用上下文变量，而不用给它们起别名使之成为成员变量。在代码清单17-13中，doLogin()方法直接调用Contexts来设置变量值。

最后，Seam上下文构成了一个层次结构（除了伪上下文无状态之外），每当需要查找一个上下文变量时（以及当你没有显式地声明应该搜索哪个上下文时），就从该层次结构的最窄范围搜索到最宽范围。Seam参考文档中有一个上下文清单，在第2章中有它们的范围，这里不再重复。

让我们来完成登录/注销特性，并添加配置和代码中漏掉的部分。

#### 17.3.4 完成登录/注销特性

登录/注销特性的导航规则漏掉了。这些是faces-config.xml中用于JSF的规则：

```
<navigation-rule>
 <navigation-case>
 <from-outcome>login</from-outcome>
 <to-view-id>/login.xhtml</to-view-id>
 <redirect/>
 </navigation-case>
 <navigation-case>
```

```

<from-outcome>start</from-outcome>
<to-view-id>/catalog.xhtml</to-view-id>
</navigation-case>
</navigation-rule>

```

当用户成功登录时，动作的start结果把浏览器带到了应用程序的起始页面，这在CaveatEmptor中是拍卖货品的目录。

当用户注销时（通过单击一个被绑定到login.doLogout()方法的按钮），返回login结果，并且/login.xhtml得以呈现。你在这里定义的规则也等于说，这已经通过浏览器转向完成了。这个方法有两个结果：第一，用户看到/login.jsf作为浏览器的URL；第二，执行完doLogout()之后，在调用应用程序阶段之后重定向立即完成。需要用这个重定向在接下来的呈现响应阶段中启动一个新的HTTP会话。旧的HTTP会话被doLogout()标识为无效，并且在调用应用程序阶段之后被丢弃。

这个应用程序并非真正安全。虽然用户在打开应用程序时最终停止在登录页面上，但是他们可以将其他的页面做成书签（就像拍卖货品目录一样），并直接跳到一个URL。如果没有出现登录令牌，你需要保护页面，并将用户重定向到登录页面。

你还需要直接保护组件bean方法，防止用户找到一种不先呈现页面而执行动作的方法。（利用通过JavaScript公开的Seam组件，这是可能的。）用EJB 3.0拦截器保护组件方法（见代码清单17-14）。

#### 代码清单17-14 检查登录令牌的EJB 3.0拦截器

```

package auction.interceptors;

import ...

@Name("loginInterceptor")
@Interceptor(around={BijectionInterceptor.class,
 ValidationInterceptor.class,
 ConversationInterceptor.class,
 BusinessProcessInterceptor.class},
 within= RemoveInterceptor.class)
public class LoggedInInterceptor {

 @AroundInvoke
 public Object checkLoggedIn(InvocationContext invocation)
 throws Exception {

 String loggedInOutcome = checkLoggedIn();

 if (loggedInOutcome == null) {
 return invocation.proceed();
 } else {
 return loggedInOutcome;
 }
 }

 public String checkLoggedIn() {
 boolean isLoggedIn =

```

```

 Contexts.getServletContext()
 .get(LoggedIn.LOGIN_TOKEN) != null;
 if (isLoggedIn) {
 return null;
 } else {
 return "login";
 }
}

```

这个拦截器有两个用处。第一，它是一个EJB 3.0 @Interceptor，在其他EJB 3.0拦截器的中间执行。这些其他的拦截器全部来自Seam，你需要把自己的拦截器放在堆栈的正确位置上。EJB 3.0注解@AroundInvoke标识受保护的组件上的任何方法在调用之前和之后而被调用的方法。如果checkLoggedIn()方法没有返回任何东西（结果为空），被拦截组件的调用就可以继续。如果结果不为空，这个结果就被传递到JSF导航处理器，并且被拦截的组件调用不能继续进行。

拦截器类也是Seam简单的Java组件（Seam组件不一定要是EJB），名为loginInterceptor。JavaBean组件默认的上下文是event（事件）。你现在可以在表达式中使用这个组件名称——例如，用表达式#{loginInterceptor.checkLoggedIn}——不通过EJB拦截。这用来保护页面不被直接访问很有用。在Seam中，可以定义在页面呈现之前运行的动作。这些声明在WEB-INF/pages.xml中：

```

<pages>
 <page view-id="/catalog.xhtml"
 action="#{loginInterceptor.checkLoggedIn}" />
</pages>

```

当用户直接命中/catalog.jsf URL时，loginInterceptor.checkLoggedIn()动作运行。如果这个动作有一个非空的结果，Seam就把结果作为常规的JSF结果进行处理，并应用导航规则。

最后，通过将拦截器应用到一个EJB类，来保护组件方法。这可以用XML（META-INF/ejb-jar.xml）完成，如果想要使用通配符并保护特定包中的所有bean的话，这个很棒。或者可以编写一个封装拦截器的辅助注解：

```

package auction.interceptors;

import ...

@Target(TYPE)
@Retention(RUNTIME)
@Documented
@Interceptors(LoggedInInterceptor.class)
public @interface LoggedIn {
 public static final String LOGIN_TOKEN = "loggedIn";
}

```

这个注解也包含所有其他代码所引用的字符串常量，这很方便。现在，在EJB类中应用这个注解：

```

package auction.beans;

import ...

@Name("catalog")
@LoggedIn
@Stateful
public class CatalogBean implements Catalog { ... }

```

每当这个EJB的任何方法被调用时，`LoggedInInterceptor`就运行，并验证用户登录。如果用户没有登录，拦截器就把`login`结果返回给JSF。

也可以在页面上检查登录令牌——例如，如果你必须决定`Logout`按钮是否应该呈现：

```

<h:form>
 <h:panelGroup rendered="#{loggedIn}">
 Current user: #{currentUser.username}
 (<h:commandLink value="Logout" action="#{login.doLogout}" />)
 </h:panelGroup>
</h:form>

```

表达式`#{loggedIn}`解析到布尔上下文变量`loggedIn`，它要么出现在会话上下文中，要么就不出现。

应用程序的登录/注销功能现在完成了。页面和组件方法受到了保护，只有一个登录用户可以打开和调用它们。

如何获得用户账号呢？必须填写一张申请表。必须验证表单数据，并在数据库中创建账号。

## 17.4 验证用户输入

在前一个示例（登录屏幕和登录/注销代码）中，你依赖标准的JSF验证器和`doLogin()`方法中自己的代码来验证用户输入。当用户提交登录表单时，JSF就在处理验证阶段中运行被声明的验证器（在`login.xhtml`中）。如果用户输入用户名和密码，且验证成功，就执行`login.doLogin()`方法。给定的用户名和密码被绑定到数据库查询。用户输入被验证两次：

- JSF在它将每个输入字段的值与绑定模型（Seam会话上下文中的`currentUser`）同步之前，验证HTML表单输入。如果稍后在动作方法中访问`currentUser`，就会保证页面的验证规则已经得到检查。
- JDBC驱动程序在你将用户名和密码绑定到JPA QL查询时验证用户输入。这是到常规的 JDBC `PreparedStatement`的一个内部绑定，因此JDBC驱动程序转义用户可能已经输入的任何危险字符。

在表现层中验证用户输入，并确保不可能有任何SQL注入攻击，这对于一个简单的登录屏幕来说够好了。但是如果需要在User对象保存在数据库中之前（例如，在账号注册过程期间）对它进行验证，这时该怎么办呢？

你需要更复杂的验证：必须检查所输入的用户名的长度，看看是否使用了任何非法字符，还要验证密码的质量。所有这些都可以用更多和可能定制的验证在JSF表现层中得以解决，但是数据库Schema也必须验证所保存数据的完整性。例如，创建数据库约束，限制保存在USERNAME列

中的值的长度，或者要求一个成功的字符串模式匹配。

在任何复杂的应用程序中，输入验证都是不仅要在表现层还要在几个层甚至不同的层中进行处理的一个关注点。它是可以影响你所有代码的横切关注点。利用Hibernate Validator，可以轻松地为所有的应用层隔离和封装验证和数据完整性规则。

### 17.4.1 Hibernate Validator简介

Hibernate Validator是Hibernate Annotations的一个模块。甚至可以不通过Hibernate和Seam，而只通过类路径中的hibernate3.jar和hibernate-annotations.jar，在任何Java应用程序中使用Hibernate Validator。（未来，Hibernate Validator可能分支到它自己的独立模块。这取决于在JSR 303“Bean Validation”中所做的工作，请见<http://jcp.org/en/jsr/detail?id=303>。）

Hibernate Validator是一组注解，可以将它应用到领域模型来声明性地定义数据验证和完整性规则。可以通过编写自己的注解，用自己的约束来扩展Hibernate Validator。

被应用的这些完整性规则和验证规则可以用于以下对象：

- **简单的Java**——可以在Java代码中的任何位置调用ClassValidator API，并提供需要检查的对象。验证器完成验证，或者返回InvalidValue对象的一个数组。每个InvalidValue都包含关于验证失败的详细信息，例如属性名称和错误消息。
- **Hibernate**——在原生的Hibernate中，可以注册钩入Hibernate持久化操作内部处理的Hibernate Validator事件。通过这些事件，Hibernate可以验证你正在数据库中自动且透明地插入或者更新的任何对象。包含详细信息的InvalidStateException在验证失败时被抛出。
- **Hibernate EntityManager**——如果通过Hibernate EntityManager使用JPA，Hibernate Validator事件就被默认为激活状态，并且当你在数据库中插入或者更新对象时，所有的实体实例都要根据验证注解进行检查。
- **SchemaExport**——Hibernate的数据库模式生成特性可以创建数据库约束，它影响着SQL DDL中的完整性规则。SchemaExport (hbm2ddl) 工具在领域模型中读取验证注解，并在SQL DDL中呈现它们。每个注解都知道SQL应该是什么样子（或者如果SQL中不存在相当的约束）。如果基于定制的过程SQL约束（触发器等）编写自己的验证注解，这个就特别强大。可以将一个定制的数据库完整性规则封装在单个Java注解中，并用Hibernate Validator在运行时检查被注解类的实例。
- **Seam**——通过Seam，可以将Hibernate Validator与应用程序的表现层和应用程序逻辑整合起来。Seam可以在JSF表单提交时自动调用验证API，并用任何验证错误消息装饰表单。

你已经在前几节中使用过Seam和Hibernate EntityManager。一旦将Hibernate Validator注解添加到实体类，这些完整性规则立即就在持久化上下文被清除到数据库时得到验证。

让我们将Hibernate Validator连到JSF用户界面，并给CaveatEmptor实现一项账号注册特性。

### 17.4.2 创建注册页面

我们将从用户界面开始。你需要一张新的页面register.xhtml，包含一个JSF表单。为了到达该

页面，必须在login.xhtml页面中提供一个链接，以便用户们知道他们可以注册：

```
<ui:define name="sidebar">
 <h1>Welcome to CaveatEmptor!</h1>
 <div>
 <h:form>
 If you don't have an account, please
 <h:commandLink action="register" immediate="true">
 register...
 </h:commandLink>
 </h:form>
 </div>
</ui:define>
```

提交这个表单，立即跳到请求过程中的呈现响应阶段（无需验证、模型绑定或者动作执行）。register字符串是一个简单的导航结果，在faces-config.xml中的导航规则中定义：

```
<navigation-rule>
 <navigation-case>
 <from-outcome>login</from-outcome>
 <to-view-id>/login.xhtml</to-view-id>
 <redirect/>
 </navigation-case>

 <navigation-case>
 <from-outcome>register</from-outcome>
 <to-view-id>/register.xhtml</to-view-id>
 <redirect/> <!-- Make this bookmarkable -->
 </navigation-case>
 ...
</navigation-rule>
```

看一下图17-10中注册页面的屏幕快照。

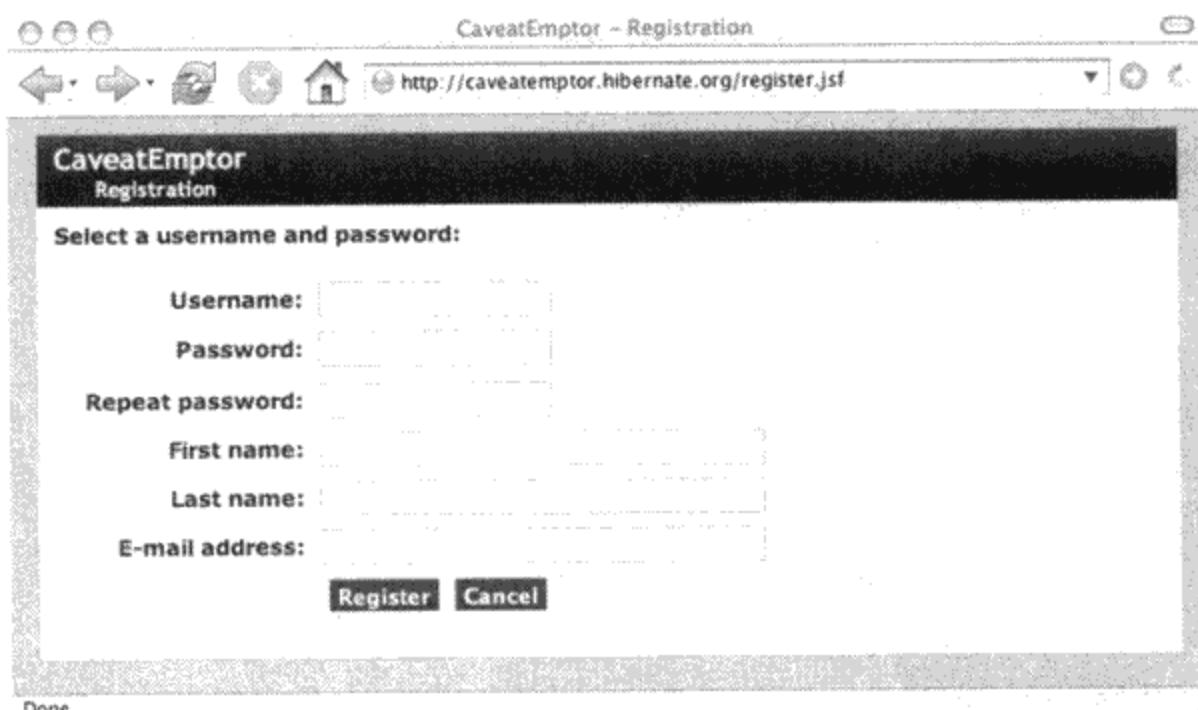


图17-10 register.xhtml页面

register.xhtml中JSF表单的代码给JSF使用了一些可视的Seam组件（可以在jboss-seam-ui.jar文件中找到这些）。

### 1. 用Seam标签装饰页面

现在用Seam组件把页面与Hibernate Validator整合起来（见代码清单17-15）。我们已经省略了页面的基础HTML；唯一值得关注的部分是表单，以及该表单的验证如何生效。还需要给Seam标签库(taglib)声明命名空间，来使用Facelets模板中的组件；在接下来的所有示例中使用的前缀都是s。

#### 代码清单17-15 包含验证的registration.xhtml源代码

```

<ui:composition ...>
 xmlns:s="http://jboss.com/products/seam/taglib"
 ...>

 <h:form>
 <f:facet name="beforeInvalidField"> ①
 <h:graphicImage value="/img/attention.gif"
 width="18" height="18"
 styleClass="attentionImage"/>
 </f:facet>
 <f:facet name="afterInvalidField"> ②
 <s:message/>
 </f:facet>

 <div class="errors" align="center"> ③
 <h:messages globalOnly="true"/>
 </div>
 <s:validateAll> ④
 <div class="entry">
 <div class="label">Username:</div>
 <div class="input"><s:decorate> ⑤
 <h:inputText size="16" required="true"
 value="#{currentUser.username}" />
 </s:decorate></div>
 </div>

 <div class="entry">
 <div class="label">Password:</div>
 <div class="input"><s:decorate>
 <h:inputSecret size="16" required="true"
 value="#{currentUser.password}" />
 </s:decorate></div>
 </div>

 <div class="entry">
 <div class="label">Repeat password:</div>
 <div class="input"><s:decorate>
 <h:inputSecret size="16" required="true"
 value="#{register.verifyPassword}" />
 </s:decorate></div>
 </div>

 <div class="entry">

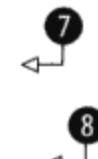
```

```

<div class="label">Firstname:</div>
<div class="input"><s:decorate>
 <h:inputText size="32" required="true"
 value="#{currentUser.firstname}" />
 </s:decorate></div>
</div>
...
</s:validateAll>

<div class="entry">
 <div class="label"> </div>
 <div class="input">
 <h:commandButton value="Register" styleClass="button"
 action="#{register.doRegister}" />
 <h:commandButton value="Cancel" styleClass="button"
 action="login" immediate="true" />
 </div>
</div>
</h:form>

```



**①** 这个组件facet被Seam装饰器（decorator）用于错误显示。你会在具有无效值的任何输入字段之前看见它。

**②** Seam装饰器把错误消息放在无效的字段之后。

**③** 没有被分配到任何字段的全局错误消息显示在表单的顶部。

**④** <s:validateAll> Seam标签给所有的子标签启用Hibernate Validator——也就是封装在这个表单中的所有输入字段。也可以只给单个字段启用Hibernate Validator，通过用<s:validate/>将输入字段包起来。

**⑤** <s:decorate> Seam标签处理验证错误消息。如果出现错误，它就把输入字段包在beforeInvalidField和afterInvalidField facet里面。

**⑥** JSF输入小部件的可见大小为16个字符。注意，JSF并不限制用户可以输入的字符串大小，但它要求用户输入一个值。这个“非空”验证仍然是JSF的工作，而不是Hibernate Validator的工作。

**⑦** Register按钮有一个动作绑定到register.doRegister（一个Seam组件）。

**⑧** 需要Cancel按钮，把用户调整到登录页面。再次用immediate="true"跳过表单的处理。

当注册表单被提交时，Seam就参与到了JSF 处理验证阶段中，并为每个实体对象（你绑定了输入字段到它上面）调用Hibernate Validator。在这个例子中，只有单个实体实例currentUser必须被验证，Seam在它的上下文中查找它。

如果处理验证阶段完成，register.doRegister就在调用应用程序中执行。这是处在事件上下文中的一个有状态会话bean。

## 2. 注册Seam组件

注册表单有两个绑定到register Seam组件。第一个绑定是值绑定，通过register.verify-

Password。JSF和Seam现在把来自这个字段的用户输入与register.setVerifyPassword()和register.getVerifyPassword()方法同步。

第二个绑定是Register按钮到register.doRegister()方法的动作绑定。这个方法必须在JSF和Hibernate Validator输入验证之后，并在currentUser可以作为新账号保存在数据库中之前，实现额外的检查。请见代码清单17-16中的代码。

### 代码清单17-16 有状态会话bean实现注册逻辑

```

package auction.beans;

import ...;

@Name("register") ①
@Scope(ScopeType.EVENT)

@Stateful
public class RegisterBean implements Register {
 @In
 private User currentUser; ②

 @PersistenceContext
 private EntityManager em; ③

 @In(create=true)
 private transient FacesMessages facesMessages; ④

 private String verifyPassword;
 public String getVerifyPassword() {
 return verifyPassword;
 }
 public void setVerifyPassword(String verifyPassword) {
 this.verifyPassword = verifyPassword;
 }
 public String doRegister() { ⑤
 if (!currentUser.getPassword().equals(verifyPassword)) {
 facesMessages.add("Passwords didn't match!");
 verifyPassword = null;
 return null;
 }
 List existing = ⑦
 em.createQuery("select u.username from User u" +
 " where u.username = :uname")
 .setParameter("uname", currentUser.getUsername())
 .getResultList();

 if (existing.size() != 0) {
 facesMessages.add("User exists!");
 return null;
 } else { ⑧
 em.persist(currentUser);
 facesMessages.add("Registration complete.");
 return "login";
 }
 }
}

```

```

 }
}

@Remove @Destroy
public void destroy() {
}

```

① register Seam组件由Seam创建，并且当事件上下文被销毁时它也被销毁，它是单个JSF请求的范围。

② Seam注入currentUser，别名取自会话上下文中的上下文变量。

③ Seam注入（或者创建，如果无法在任何上下文中找到变量的话）FacesMessages的一个实例。如果需要发送消息给一个JSF页面，这就是一个很方便的辅助方法；你在之前没有通过注入但是通过手工查找使用过它。

④ 这个组件的verifyPassword字段与JSF表单同步。

⑤ 这个方法给新账号的注册实现了主要逻辑。它在Hibernate Validator检查完currentUser之后调用。

⑥ 用户输入的两个密码必须相符，否则表单上就会显示一条错误消息。null结果利用错误消息触发登录表单的重新显示。

⑦ 用户名在数据库中是唯一的。这个多行约束不能通过Hibernate Validator在内存中进行检查。需要执行数据库查询，并验证用户名。

⑧ 如果所有的验证都通过，就persist()该currentUser对象，清除持久化上下文，并且当doRegister()方法返回时提交事务。结果login把用户调整回到登录页面，在这里“注册完成”的消息呈现在登录表单上。

⑨ 当事件上下文被销毁的时候，Seam在JSF请求结束时调用组件的destroy()方法。EJB 3.0容器移除有状态会话bean，因为该方法用@Remove作了标识。

用户输入验证经常比在单个对象中核查单个值更为复杂。Seam给注册表单中的所有被绑定的实体实例调用Hibernate Validator。然而，所输入用户名的重复检查需要数据库访问。可以出于这个目的编写自己的Hibernate Validator扩展，但是当User对象必须进行验证时，总是在数据库中检查重复的用户名似乎是不可能的。另一方面，业务逻辑一般通过过程代码实现，而不是完全声明性地实现。

到目前为止，Hibernate Validator还没有做任何事情。如果你提交注册表单而没有输入任何值，就只有required="true"的内建JSF验证器运行。你在每个声明需要值的输入字段中得到一个内建的JSF错误消息。

### 3. 注解实体类

Hibernate Validator不是活动的，因为User实体类中没有完整性规则，因此所有的对象都通过验证测试。可以在实体类的字段或者获取方法中添加验证注解：

```

package auction.model;

import ...

@Name("user")
@Role(name = "currentUser", scope = ScopeType.SESSION)

@Entity
@Table(name = "USERS")
public class User implements Serializable {

 @Id @GeneratedValue
 @Column(name = "USER_ID")
 private Long id = null;

 @Column(name = "USERNAME", nullable = false, unique = true)
 @org.hibernate.validator.Length(
 min = 3, max = 16,
 message = "Minimum {min}, maximum {max} characters."
)
 @org.hibernate.validator.Pattern(
 regex = "^\w+$",
 message = "Invalid username!"
)
 private String username;

 @Column(name = "PASSWORD", length = 12, nullable = false)
 private String password;

 @Column(name = "FIRSTNAME", length = 255, nullable = false)
 private String firstname;

 @Column(name = "LASTNAME", length = 255, nullable = false)
 private String lastname;

 ...
}

```

你只应用两个Hibernate Validator注解：@Length和@Pattern验证器。这些验证器有着如最大长度和最小长度这样的属性，或者正则表达式模式（请见java.util.regex.Pattern）。所有内建的验证注解的清单，请见Hibernate Annotations包中Hibernate Validator参考文档。也可以轻松地编写自己的注解。

所有的验证注解都有message属性。如果出现验证失败，这个消息就与表单字段显示在一起。可以添加更多的验证注解，它们还核查密码、User的名和姓。注意，USERNAME @Column注解的length属性已经被移除了。由于长度验证注解，Hibernate的Schema导出工具现在知道VARCHAR(16)必须在数据库Schema中创建。另一方面，nullable=false属性还在，用于NOT NULL（非空）数据库列约束的生成。（可以使用一个来自Hibernate Validator的@NotNull验证注解，但是JSF已经替你检查了该字段：表单字段为required="true"。）

在将验证注解添加到User之后，提交包含不完整的值的注册表单时会显示错误消息，如图17-11所示。

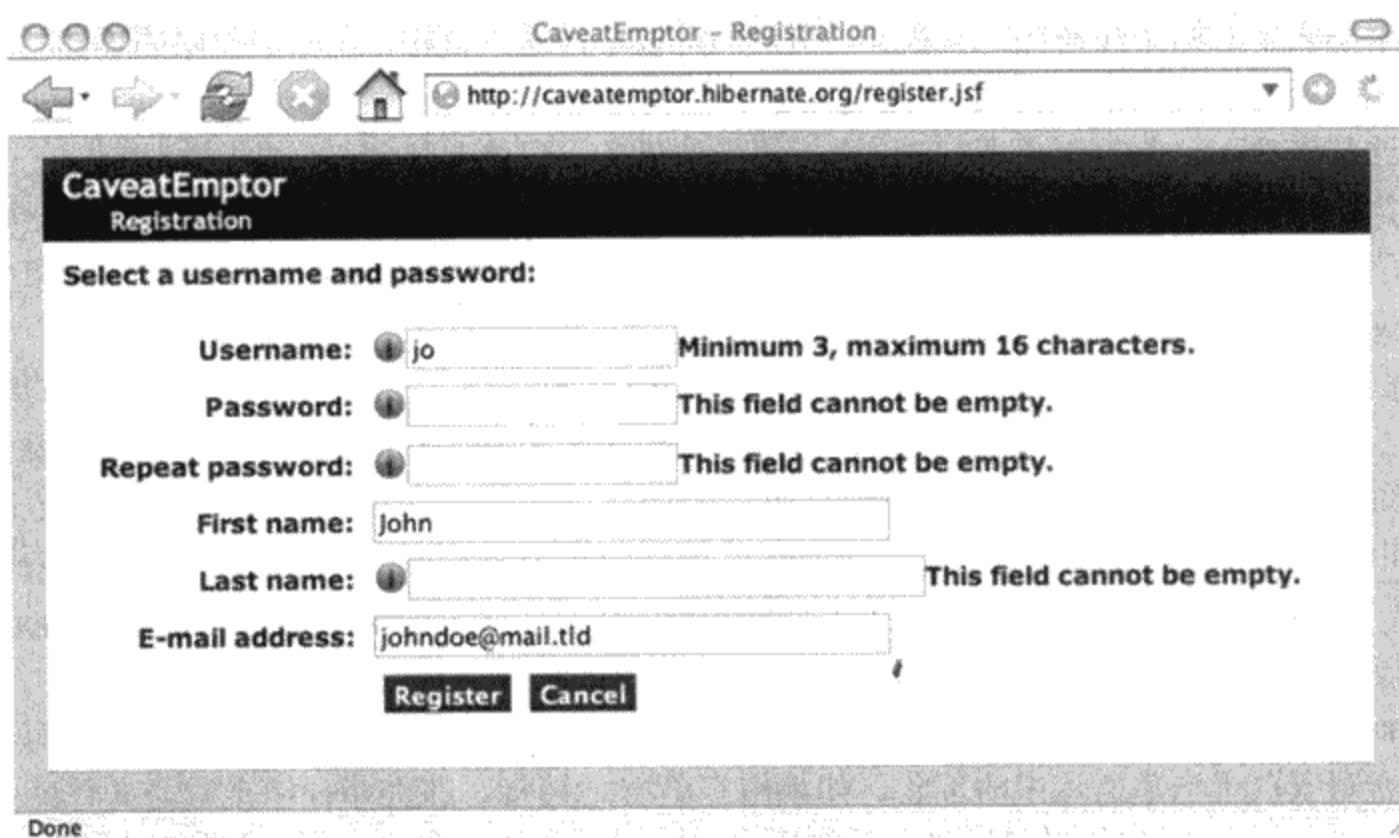


图17-11 Seam用验证错误消息装饰输入字段

注册特性现在完成了，用户们可以创建新的账号。似乎还不完美的方面在于错误消息。如果你试试这段代码，就会发现错误消息并不像图17-11所示的那么好。相反，需要输入的字段中有一个很难看的\_id23: Field input is required消息。而且，把英文的错误消息放到你的实体类中（即使它们是在注解元数据中）是一种好办法吗？

不是只替换默认的JSF错误消息（它包括自动生成的小部件标识符），而是把所有的用户界面消息都隔离，并且还允许用户切换语言。

### 17.4.3 用 Seam 实现国际化

朝着多国语言应用程序迈出的第一步是语言切换器——假设是用户可以在应用程序的最上层菜单中单击的一个链接。Seam有一个localeSelector组件（在会话上下文中），使得这一步变得容易起来：

```
<h:form>
 <h:panelGroup>
 <h:outputText value="#{messages['SelectLanguage']}"/>
 <h:commandLink
 value="EN"
 action="#{localeSelector.selectLanguage('en')}" />
 |
 <h:commandLink
 value="DE"
 action="#{localeSelector.selectLanguage('de')}" />
 </h:panelGroup>
</h:form>
```

这个小表单有两个超链接：EN和DE。用户们可以单击这些链接，在应用程序的界面上进行英语版和德语版之间的切换。链接动作通过文字参数被绑定到`localeSelector.selectLanguage()`方法。这些参数（en和de）都是ISO语言代码，请见Javadoc中的`java.util.Locale`。

但是，这还不是这里所发生的全部。呈现表单时，`# {messages['SelectLanguage']}`表达式被取值，并且该表达式的输出在命令链接之前被呈现为文本。这个表达式的输出有点像“Select your language:”。它是哪里来的？

很清楚，`messages`是一个Seam组件，它在会话上下文中。它表示被外部化消息的映射；`SelectLanguage`是搜索这个映射的一个键。如果该映射包含了该键的一个值，这个值就被打印出来。否则，`SelectLanguage`就被逐字打印。

可以在你能够编写解析Seam组件的表达式的任何地方使用`messages`组件（几乎是任何地方）。这个组件是指向Java资源包的一个方便的句柄，这是一个复杂的术语，意味着.property文件中的键/值对。

Seam自动地从类路径的根目录下将`messages.properties`读取到`messages`组件中。然而，实际的文件名则取决于当前所选中的位置。如果用户单击DE链接，在类路径中搜索的文件就命名为`messages_de.properties`。如果英语是活动的语言（它是默认的，取决于JSF配置和浏览器），被加载的文件就是`messages_en.properties`。

下面是`messages_en.properties`的一个片段：

```
SelectLanguage = Select language:
PleaseRegisterHint = Create a new account...
SelectUsernameAndPassword = Select a username and password
PasswordVerify = Repéat password
PasswordVerifyField = Controlpassword
Firstname = First name
Lastname = Last name
Email = E-mail address
TooShortOrLongUsername = Minimum 3, maximum 16 characters.
NotValidUsername = Invalid name! {TooShortOrLongUsername}
PasswordVerifyFailed = Passwords didn't match, try again.
UserAlreadyExists = A user with this name already exists.
SuccessfulRegistration = Registration complete, please log in:
DoRegister = Register
Cancel = Cancel

Override JSF defaults
javax.faces.component.UIInput.REQUIRED = This field cannot be empty.
```

最后一行给输入字段小部件覆盖默认的JSF验证错误消息。如果想要合并消息，语法{Key}就很有帮助；`TooShortOrLongUsername`消息被附加到`NotValidUsername`消息。

现在可以用在`messages` Seam组件中查找键的表达式，来替换XHTML文件中的所有字符串。也可以在Java代码中使用`RegistrationBean`组件中的资源包的键：

```
public String doRegister() {
 if (!currentUser.getPassword().equals(verifyPassword)) {
 facesMessages
 .addFromResourceBundle("PasswordVerifyFailed");
 }
}
```

```

 verifyPassword = null;
 return null;
}

List existing =
 em.createQuery("select u.username from User u" +
 " where u.username = :uname")
 .setParameter("uname", currentUser.getUsername())
 .getResultList();

if (existing.size() != 0) {
 facesMessages
 .addFromResourceBundle("UserAlreadyExists");
 return null;
} else {
 em.persist(currentUser);
 facesMessages
 .addFromResourceBundle("SuccessfulRegistration");
 return "login";
}
}

```

最后，可以在Hibernate Validator的消息中使用资源包的键（这不是一项Seam特性——它没有Seam也可以进行）：

```

@Entity
public class User implements Serializable {

 ...

 @Column(name = "USERNAME", nullable = false, unique = true)
 @org.hibernate.validator.Length(
 min = 3, max = 16,
 message = "{TooShortOrLongUsername}"
)
 @org.hibernate.validator.Pattern(
 regex="^\\w*$/,
 message = "{NotValidUsername}"
)
 private String username;
...
}

```

让我们翻译一下该资源包，并把它保存为message\_de.properties：

```

SelectLanguage = Sprache:
PleaseRegisterHint = Neuen Account anlegen...
SelectUsernameAndPassword = Benutzername und Passwort w\u00fclhlen
PasswordVerify = Passwort (Wiederholung)
PasswordVerifyField = Kontrollpasswort
Firstname = Vorname
Lastname = Nachname
Email = E-mail Adresse
TooShortOrLongUsername = Minimum 3, maximal 16 Zeichen.
NotValidUsername = Ung\u00fcltiger Name! {TooShortOrLongUsername}
PasswordVerifyFailed = Passworte nicht gleich, bitte wiederholen.
UserAlreadyExists = Ein Benutzer mit diesem Namen existiert bereits.
SuccessfulRegistration = Registrierung komplett, bitte einloggen:

```

```

DoRegister = Registrieren
Cancel = Abbrechen

Override JSF defaults
javax.faces.component.UIInput.REQUIRED = Eingabe erforderlich.

```

注意，你用UTF序列来表达非ASCII的字符。如果用户在应用程序中选择德语，并试图在不完成表单的情况下进行注册，那么所有的消息都以德语显示（见图17-12）。

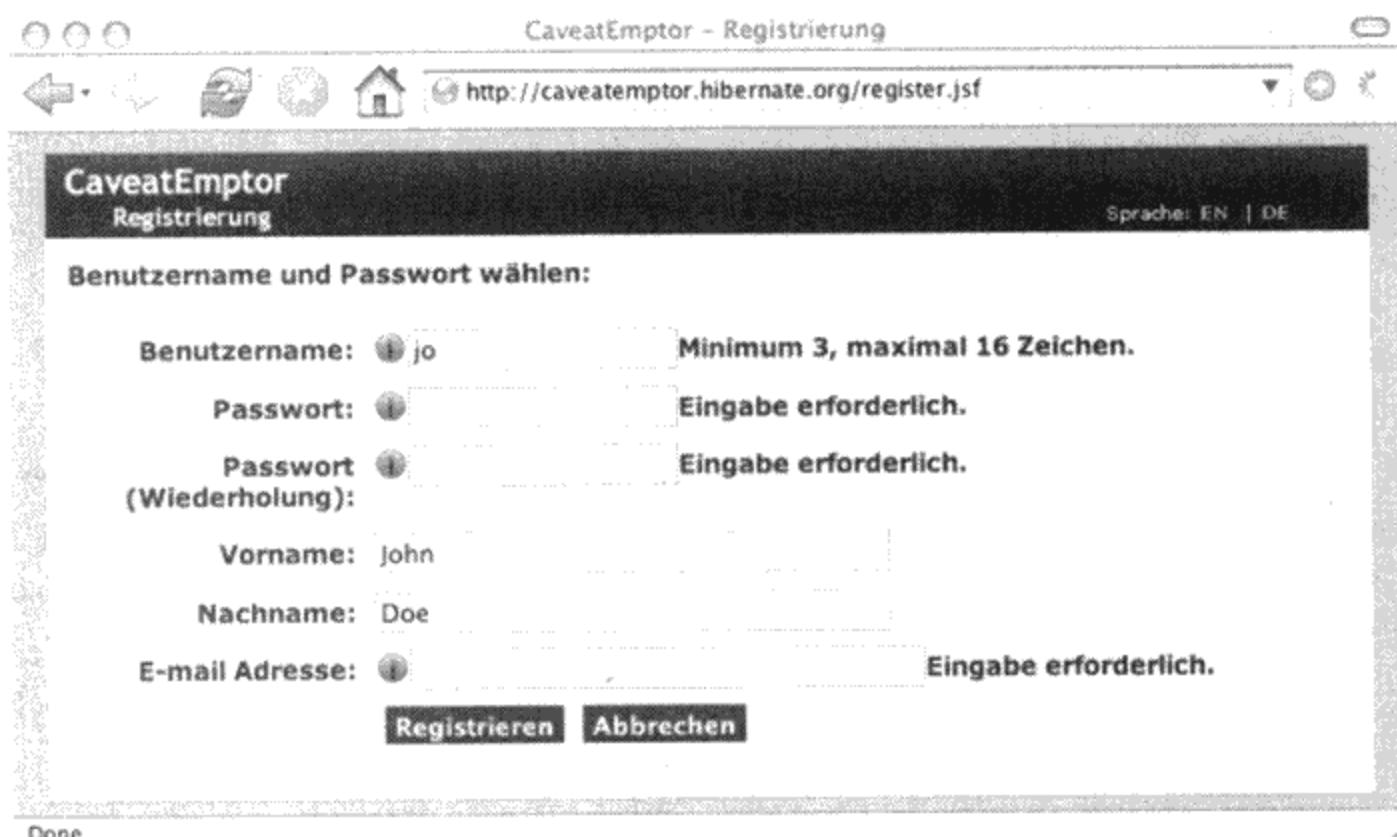


图17-12 用户界面已经被转化为德语

被选中的语言是一个会话范围的设置。它现在是活动的，直到用户注销（使HTTP会话无效）。如果你也在seam.properties中设置localeSelector.cookieEnabled=true开关，那么用户语言选择就会被作为一个cookie保存在Web浏览器中。

我们要示范的最后一项但并非最不重要的Seam特性是，通过Seam的自动持久化上下文处理。如果已经在Hibernate应用程序中见过LazyInitializationException（谁没见过？），这就是最完美的解决方案。

## 17.5 利用 Seam 简化持久化

本章前面的所有示例都使用EntityManager，由EJB 3.0容器注入。EJB中的成员字段利用@PersistenceContext进行注解，并且持久化上下文的范围始终是为特定的动作方法启动和提交的事务。对于Hibernate而言，Hibernate Session为在会话bean中调用的每一个方法而打开、清除和关闭。

当会话bean方法返回，并且持久化上下文关闭时，那么从位于那个bean方法的数据库中加载

的所有实体实例都处于脱管状态。可以在JSF页面中通过访问这些实例已初始化的属性和集合来呈现它们，但是当你试图访问一个未被初始化的关联或者集合时，就会得到LazyInitializationException。如果想要让它再次处于持久化状态，还必须重附（或者合并，通过JPA）脱管实例。此外，必须小心编写实体类的equals()和hashCode()方法代码，因为受保护的同一性范围只有事务，与（相对短的）持久化上下文范围一样。

本书前面已经几次讨论脱管对象状态的后果。通常我们断定，通过将持久化上下文和同一性范围扩展到超过事务来避免脱管状态，这是一种更为可取的解决方案。你已经见过OSIV模式，它将持久化上下文扩展为跨越整个请求。虽然这个模式是一种务实的解决方案，用于以无状态方式（其最重要的范围是请求）构建的应用程序，但是如果你编写一个包含对话的有状态Seam应用程序，就需要一种更加强大的变形。

如果让Seam将EntityManager注入到会话bean中去，并且如果你让Seam管理持久化上下文，将会得到下列结果：

- 被扩展的持久化上下文自动绑定和界定到对话——你有一个跨越对话的受保护的同一性范围。一个特定的对话至少有一种特定数据库行的内存表示法。没有脱管对象，你可以轻松地通过双等号比较实体实例（`a==b`）。不必实现equals()和hashCode()，并通过业务键比较实体实例。
- 在一个对话中访问未被初始化的代理或者集合时，再也没有LazyInitializationExceptions了——持久化上下文对于整个对话都是活动的，且持久化引擎始终可以按需抓取数据。Seam提供了一种更加强大且方便的OSIV模式的实现，它不仅在单个请求期间而且在整个对话期间，避免了游离对象。
- 自动把JSF请求包在几个系统事务中——Seam使用几个事务来封装JSF请求生命周期中的几个阶段。稍后会讨论这个事务集合，它的其中一个好处在于，你有一个保持数据库锁定时间尽可能短的优化集合，而不用编写任何代码。

让我们用一个示例进行示范，通过把前一节的注册过程重新编写为包含被扩展的持久化上下文的对话。前一个实现基本上是无状态的：RegisterBean只被界定到单个事件。

### 17.5.1 实现对话

回来看一下代码清单17-16中所示的代码。这个有状态会话bean是用于CaveatEmptor中账户注册页面的backing bean。当用户打开或者提交注册页面时，就创建该bean的一个实例，并且在处理事件时处于活动状态。JSF将表单值绑定到bean中（通过verifyPassword和Seam注入的currentUser），并在必要时调用动作监听器方法。

这是一个无状态的设计。虽然你使用了一个有状态会话bean，但它的范围是单个请求（Seam中的事件上下文）。这种方法效果很好，因为用户经过的对话很是繁琐——只有包含单个表单的单张页面必须填写和提交。图17-13展现了一个更为复杂的注册过程。

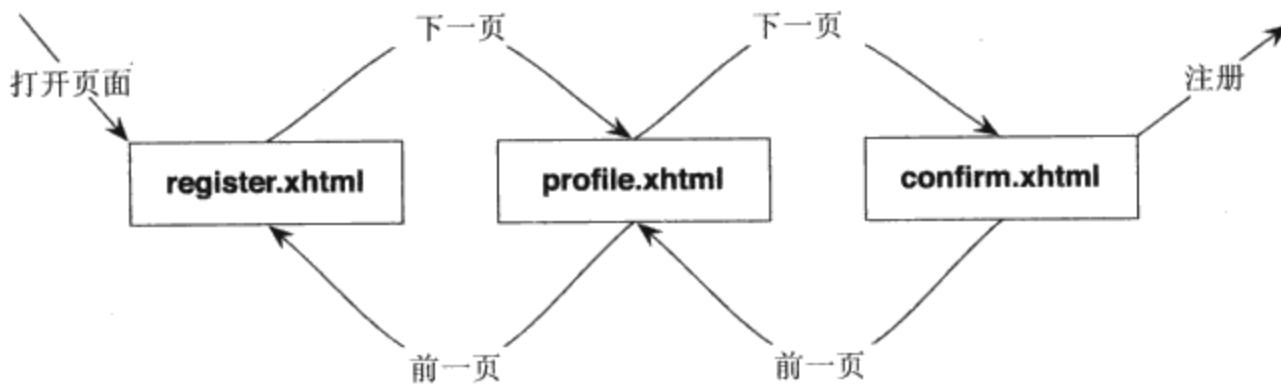


图17-13 CaveatEmptor注册向导

用户打开register.xhtml，并输入想要的用户名和密码。用户单击Next Page之后，第二个包含个性化数据（名字、电子邮件地址等）的表单出现，并且必须填写。最后一页再一次显示所有的账号和个性化数据，因此用户可以确认（或者退回并纠正）。

这个注册过程是一个向导风格的对话，包含一般的Next Page和Previous Page按钮，允许用户在对话中前进或者后退。许多应用程序都需要这种对话框。如果没有Seam，实现多页对话对于Web应用程序开发人员来说，仍然很困难。（注意，有许多其他好的会话用例，向导对话框是很常见的。）

给这个对话编写一些页面和Seam组件。

### 1. 注册页面

register.xhtml页面看起来几乎与代码清单17-15中所示的如出一辙。移除了个性化表单字段（名、姓、电子邮件地址），并用Next Page按钮替换Register按钮：

```

...
<s:validateAll>

<div class="entry">
 <div class="label">Username:</div>
 <div class="input">
 <s:decorate>
 <h:inputText size="16" required="true"
 value="#{register.user.username}" />
 </s:decorate>
 </div>
</div>

<div class="entry">
 <div class="label">Password:</div>
 <div class="input">
 <s:decorate>
 <h:inputSecret size="16" required="true"
 value="#{register.user.password}" />
 </s:decorate>
 </div>
</div>

<div class="entry">
 <div class="label">Repeat password:</div>
 <div class="input">
 <s:decorate>

```

```

 <h:inputSecret size="16" required="true"
 value="#{register.verifyPassword}" />
 </s:decorate>
</div>
</div>

</s:validateAll>

<div class="entry">
 <div class="label"> </div>
 <div class="input">
 <h:commandButton value="Next Page"
 styleClass="button"
 action="#{register.enterAccount}" />
 </div>
</div>

```

你仍然正在引用register组件来绑定值和动作，很快会见到这个类。将表单值绑定到register.getUser()返回的User对象。currentUser不见了。你现在有了一个对话上下文，并且不再需要使用HTTP会话上下文了（如果用户试图同时在两个浏览器窗口中注册两个账户，前一个实现就不会生效）。register组件现在对话期间保存绑定到所有表单字段的User状态。

enterAccount()方法的结果把用户转到了下一页，即个性化表单。注意，你仍然依赖Hibernate Validator [由Seam在请求的处理验证阶段中进行调用(<s:validateAll/>)]进行输入验证。如果输入验证失败，页面就重新显示。

## 2. 个性化页面

profile.xhtml页面几乎与register.xhtml页面一样。个性化表单包括个性化字段，并且页面底部的按钮允许用户在对话中后退或者前进：

```

...
<div class="entry">
 <div class="label">E-mail address:</div>
 <div class="input">
 <s:decorate>
 <h:inputText size="32" required="true"
 value="#{register.user.email}" />
 </s:decorate>
 </div>
</div>

<div class="entry">
 <div class="label"> </div>
 <div class="input">
 <h:commandButton value="Previous Page"
 styleClass="button"
 action="register" />
 <h:commandButton value="Next Page"
 styleClass="button"
 action="#{register.enterProfile}" />
 </div>
</div>

```

当表单提交时，用户填写的任何表单字段都被应用到register.user模型。Previous Page按钮跳过调用应用程序阶段，并产生register结果——显示前一页。注意这个表单没有用<s:validateAll/>围绕起来，当用户单击Previous Page按钮时，你并不想要进行处理验证。调用Hibernate Validator现在被委托给了register.enterProfile动作。应该只在用户单击Next Page的时候验证表单输入。然而，你保留表单字段上的装饰，来显示任何验证错误的消息。

下一页概述了账户和个性化。

### 3. 概述页面

在confirm.xhtml中，所有的输入都呈现在概述中，允许用户在最终提交账户和个性化的详细信息用于注册之前，预览它们：

```
...
<div class="entry">
 <div class="label">Last name:</div>
 <div class="output">#{register.user.lastname}</div>
</div>

<div class="entry">
 <div class="label">E-mail address:</div>
 <div class="output">#{register.user.email}</div>
</div>

<div class="entry">
 <div class="label"> </div>
 <div class="input">
 <h:commandButton value="Previous Page"
 styleClass="button"
 action="profile"/>
 <h:commandButton value="Register"
 styleClass="button"
 action="#{register.confirm}"/>
 </div>
</div>
```

Previous Page按钮呈现了profile结果定义的响应，它就是前一个页面。当用户单击Register时，调用register.confirm方法。这个动作方法终止了对话。

最后，编写退回这个对话的Seam组件。

### 4. 编写对话的Seam组件

代码清单17-16中所示的RegisterBean必须被界定到对话。首先，这是该接口：

```
public interface Register {
 // Value binding methods
 public User getUser();
 public void setUser(User user);

 public String getVerifyPassword();
 public void setVerifyPassword(String verifyPassword);

 // Action binding methods
 public String enterAccount();
```

```

public String enterProfile();
public String confirm();

// Cleanup routine
public void destroy();
}

```

Seam对话模型的优势之一在于，可以像阅读会话的历史一样阅读你的接口。用户输入账户数据，之后是个性化数据。最后，确认并保存输入。

bean的实现如代码清单17-17所示。

### 代码清单17-17 对话范围的Seam组件

```

package auction.beans;

import ...

@Name("register")
@Scope(ScopeType.CONVERSATION) ①
@Stateful
public class RegisterBean implements Register {
 @PersistenceContext ②
 private EntityManager em;

 @In(create=true)
 private transient FacesMessages facesMessages; ③

 private User user;
 public User getUser() {
 if (user == null) user = new User();
 return user;
 }
 public void setUser(User user) {
 this.user = user;
 }

 private String verifyPassword; ④
 public String getVerifyPassword() {
 return verifyPassword;
 }
 public void setVerifyPassword(String verifyPassword) {
 this.verifyPassword = verifyPassword;
 }

 @Begin(join = true) ⑤
 public String enterAccount() {
 if (verifyPasswordMismatch() || usernameExists()) {
 return null; // Redisplay page
 } else {
 return "profile";
 }
 }

 @IfInvalid(outcome = Outcome.REDISPLAY) ⑥
 public String enterProfile() {

```

```

 return "confirm";
 }

@End(ifOutcome = "login") ⑦
public String confirm() {
 if (usernameExists()) return "register"; // Safety check
 em.persist(user);
 facesMessages.add("Registration successful!");
 return "login";
}

@Remove @Destroy
public void destroy() {}

private boolean usernameExists() {
 List existing =
 em.createQuery("select u.username from User u" +
 " where u.username = :uname")
 .setParameter("uname",
 user.getUsername())
 .getResultList();

 if (existing.size() != 0) {
 facesMessages.add("Username exists");
 return true;
 }
 return false;
}

private boolean verifyPasswordMismatch() {
 if (!user.getPassword().equals(verifyPassword)) {
 facesMessages.add("Passwords do not match");
 verifyPassword = null;
 return true;
 }
 return false;
}
}

```

- ① 当Seam实例化这个组件时，一个实例就被绑定到了以变量register为名的会话上下文中。
- ② EJB 3.0容器注入一个事务范围的持久化上下文。稍后你会在这里用Seam注入一个会话范围的持久化上下文。
- ③ user成员变量通过访问方法公开，因此JSF输入小部件可以被绑定到独立的User属性。对话期间user的状态由register组件保存。
- ④ verifyPassword成员变量也通过访问方法公开给表单中的值绑定，并且在对话期间的状态被保存。
- ⑤ 当用户在第一个屏幕中单击Next Page时，调用enterAccount()方法。当这个方法返回时，当前的对话通过@Begin被提升为长运行的对话，因此它跨越未来的请求，直到用@End进行标记的方法返回。由于用户可以退回到第一个页面，并重新提交表单，因此如果现存的对话已经在进

行，那么就需要联结它。

⑥ 当用户在第二个屏幕中单击Next Page时，调用enterProfile()方法。由于它通过@IfInvalid作标记，Seam给输入验证执行Hibernate Validator。如果出现错误，就用Hibernate Validator中的错误消息重新显示页面（Outcome.REDISPLAY是一个方便的常量快捷方式）。如果没有错误，结果就是对话的最后一张页面。

⑦ 当用户在最后一个屏幕中单击Register时，调用confirm()方法。当方法返回login结果时，Seam终止长运行的对话，并通过调用以@Destory作标记的方法来销毁组件。同时，如果有其他人选择了相同的用户名，你就把用户转向到对话的第一个页面，对话上下文保持完整和活动。

本章前面已经介绍过了大多数的注解。唯一新出现的是@IfInvalid，它在调用enterProfile()方法时触发Hibernate Validator。注册对话现在完成了，一切都如预期般进行着。持久化上下文由EJB容器处理，当调用方法时，一个全新的持久化上下文就被分配到了每一个动作。

你还没有遇到任何问题，因为代码和页面没有通过从脱管的领域对象中拖取视图中的数据来按需加载数据。然而，几乎任何比注册过程更复杂的对话都将触发LazyInitializationException。

### 17.5.2 让 Seam 管理持久化上下文

让我们来激发LazyInitializationException。当用户输入对话的最后一个屏幕（确认对话框）时，你呈现了一个拍卖分类的清单。用户可以给他们的账户选择默认的分类：他们想要浏览和从中出售项目的默认分类。分类的清单从数据库中加载，并通过获取方法公开。

#### 1. 触发LazyInitializationException

编辑RegisterBean组件，并公开一个从数据库中加载的拍卖分类的清单：

```
public class RegisterBean implements Register {
 ...
 private List<Category> categories;
 public List<Category> getCategories() {
 return categories;
 }
 ...
 @IfInvalid(outcome = Outcome.REDISPLAY)
 public String enterProfile() {
 categories =
 em.createQuery("select c from Category c" +
 " where c.parentCategory is null")
 .getResultList();
 return "confirm";
 }
}
```

你还将getCategories()方法添加到组件的接口。在confirm.xhtml视图中，现在可以绑定到这个获取方法来显示分类：

```

...
<div class="entry">
 <div class="label">E-mail address:</div>
 <div class="output">#{register.user.email}</div>
</div>

<div class="entry">
 <div class="label">Default category:</div>
 <div class="input">
 <tr:tree var="cat"
 value="#{registrationCategoryAdapter.treeModel}">
 <f:facet name="nodeStamp">
 <h:outputText value="#{cat.name}" />
 </f:facet>
 </tr:tree>
 </div>
</div>
...

```

为了显示分类，你用了一个不同的小部件（它不在标准的JSF集中）。它是Apache Myfaces Trinidad项目中的一个可视化的树数据组件。它也需要一个将分类清单转换为树数据模型的适配器。但是这并不重要（可以在CaveatEmptor下载中为它找到那些库和配置）。

重要的是，如果分类树呈现，那么在调用enterProfile()之后，持久化上下文就已经在呈现响应阶段中被关闭。现在，哪些分类在脱管状态下完全可用？只有根类别（没有父类别的类别）已经从数据库中得以加载。如果用户单击树显示，并且想要看看一种分类是否包含任何子分类，应用程序便通过LazyInitializationException而失败。

通过Seam，可以轻松地扩展持久化上下文为跨越整个对话，而不仅仅是单个方法或者单个事件。然后，数据的按需加载在对话和任何JSF过程阶段中的任何位置便都成为可能。

## 2. 注入Seam持久化上下文

首先，配置Seam托管的持久化上下文。编辑（或者创建）WEB-INF目录下的文件components.xml：

```

<components>

 <component name="org.jboss.seam.core.init">
 <!-- Enable seam.debug page -->
 <property name="debug">false</property>

 <!-- How does Seam lookup EJBs in JNDI -->
 <property name="jndiPattern">
 caveatemptor/#{ejbName}/local
 </property>
 </component>

 <component name="org.jboss.seam.core.manager">
 <!-- 10 minute inactive conversation timeout -->
 <property name="conversationTimeout">600000</property>
 </component>

 <component

```

```

name="caveatEmptorEM"
class="org.jboss.seam.core.ManagedPersistenceContext">
<property name="persistenceUnitJndiName">
 java:/EntityManagerFactories/caveatEmptorEMF
</property>
</component>
</components>

```

你也将所有其他的Seam配置选项都移到了这个文件，因此seam.properties现在是空的（但是对于组件扫描器来说，作为标记它仍然是必需的）。

当Seam启动时，它将类ManagePersistenceContext配置为Seam组件。这就像把Seam注解放到该类中（在这个Seam绑定的类中也有注解）。组件的名称为caveatEmptorEM，它实现EntityManager接口。现在每当需要EntityManager时，就让Seam注入caveatEmptorEM。

(ManagePersistenceContext类需要知道如何获得真正的EntityManager，因此你必须在JNDI中提供EntityManagerFactory的名称。如何获取EntityManagerFactory到JNDI中，这取决于Java Persistence提供程序。在Hibernate中，可以通过在persistence.xml中的jboss.entity.manager.factory.jndi.name配置这个绑定。)

再一次修改RegisterBean，并使用Seam持久化上下文：

```

@Name("register")
@Scope(ScopeType.CONVERSATION)

@Stateful
public class RegisterBean implements Register {

 @In(create = true, value = "caveatEmptorEM")
 private EntityManager em;

 ...
}

```

当这个组件中的方法第一次调用时，Seam就会创建ManagedPersistenceContext的一个实例，将它绑定到对话上下文中的变量caveatEmptorEM中，并且在执行方法之前，将它注入到成员字段em里面。当对话上下文被销毁时，Seam也销毁ManagedPersistenceContext实例，这样就销毁了持久化上下文。

持久化上下文什么时候被清除呢？

### 3. 整合持久化上下文生命周期

每当事务提交时，Seam托管的持久化上下文就被清除。不用把你的动作方法包在事务（通过注解）中，而是让Seam也管理事务。这是JSF的不同Seam阶段监听器的任务，替换掉faces-config.xml中那个基本的任务：

```

<lifecycle>
 <phase-listener>
 org.jboss.seam.jsf.TransactionalSeamPhaseListener
 </phase-listener>
</lifecycle>

```

这个监听器使用了两个系统事务来处理一个JSF请求。一个事务在恢复视图阶段启动，并在调用应用程序阶段之后提交。这些阶段中的任何系统异常都触发事务的自动回滚。可以用一个异

常处理器准备一个不同的响应（这是JSF的弱点——必须在web.xml中用servlet异常处理器来实现这一点）。通过在动作方法执行完成之后提交第一个事务，使得动作方法中由SQL DML创建的任何数据库锁定都尽可能地短。

第二个事务跨越JSF请求的呈现响应阶段。按需拖动数据（并触发延迟加载的关联和集合的初始化）的任何视图都在这第二个事务中运行。这是一个其数据为只读的事务，因此在该阶段期间没有创建任何数据库锁定（如果你的数据库没有以可重复的读取模式运行，或者如果它有一个多版本的并发控制系统的话）。

最后，注意持久化上下文跨越对话，但是清除和提交可能发生在对话期间。因此，整个对话不具有原子性。当对话被提升为长运行时，可以用@Begin(flushMode=FlushModeType.MANUAL)禁用自动的清除；然后必须在对话终止时手工调用flush()（通常在以@End标识的方法中）。

持久化上下文现在可以通过Seam注入在任何组件（无状态或者有状态）中使用。对话中的持久化上下文始终相同，它表现得就像一个高速缓存和已经从数据库中加载的所有实体对象的同一性映射。

跨整个对话的被扩展持久化上下文具有一些乍看起来可能不太明显的其他好处。例如，持久化上下文不仅是同一性映射，还是对话期间已经从数据库加载的所有实体对象的高速缓存。

想象你没有保存请求之间的对话状态，而是在每个请求结束时，将每一块信息都推到数据库或者HTTP会话中（或者放到被隐藏的表单字段，或者cookie或者请求参数等中去）。当下一个请求命中服务器时，通过访问数据库、HTTP会话等再次装配状态。因为没有其他的有用上下文，也没有对话的编程模型，因此你必须给每个请求重组和分解应用程序状态。这个无状态的应用程序设计不可伸缩——无法给每一个客户端请求都命中数据库（对于伸缩最昂贵的那个层）！

开发人员试图通过启用Hibernate的二级高速缓存来解决这个问题。然而，用对话高速缓存伸缩一个应用程序，比用一个哑二级数据高速缓存伸缩它更值得关注。尤其在集群中，每当任何数据片段被任何节点修改时，二级高速缓存就强制在所有集群节点上的高速缓存更新。通过对话高速缓存，只有这个特定对话的负载均衡或者故障切换所需的节点必须参与当前对话数据的复制（在这个例子中是有状态会话bean的复制）。复制可以明显减少，因为没有全局共享的高速缓存需要同步。

我们很想讨论更多关于Seam的内容，并介绍一些其他的示例，但可惜篇幅有限。

## 17.6 小结

本章介绍了JSF、EJB 3.0，以及如何通过JBoss Seam框架改进利用这些标准的Web应用程序。我们讨论了Seam的上下文，以及组件如何以上下文的方式联结在一起。我们讨论了Seam与Hibernate Validator的整合，并且你了解了为什么Seam托管的持久化上下文是LazyInitializationException的首选解决方案。

如果你发现Seam的世界很精彩，还有更多的内容有待探索：

- Seam组件模型也支持事件/监听器概念，它允许组件通过松散耦合的观察者/可观察模式互相调用。

- 可以通过一个页面流描述符给对话启用有状态的导航流，替换无状态的JSF导航模型。这样解决了对话期间用户在浏览器中单击Back按钮时可能遇到的问题。
- Seam有一个复杂的并发模型用于服务器中的异步处理（与JMS整合），以及在对话中的并发性处理（Seam防止对话两次提交）。
- Seam允许你将对话和业务流程管理任务轻松地连在一起。它整合了工作流和JBoss jBPM的业务流程上下文（<http://www.jboss.com/products/jbpm>）。
- Seam整合了JBoss Rules（<http://www.jboss.com/products/rules>）。可以访问Seam组件中的策略，和来自规则的Seam组件。
- Seam带有JavaScript库。通过这个Remoting框架，可以轻松地从客户端代码调用Seam组件。Seam可以处理任何到你服务器的Ajax请求。
- Seam的Application Framework提供开箱即用的组件，让你能够轻捷地编写易伸缩的CRUD数据库应用程序。
- Seam组件很容易测试，无论是否使用了（可嵌入的）容器。Seam通过SeamTest超类，使得为TestNG进行集成测试和功能测试变得极为轻松；这个超类允许你编写模拟Web浏览器交互的脚本。

如果想要继续学习Seam，并探索上列没有的其他特性，请继续学习Seam参考文档中的教程。

## 附录 A

# SQL基础知识

包含着行和列的表，对于任何已经使用过SQL数据库的人来说是再熟悉不过的了。有时候你会看到表被称作关系（relation），行称作字节组（tuple），列称作属性（attribute）。这是关系数据模型（relational data model）的语言，是SQL数据库（不完全）实现的数学模型。

关系模型允许你定义数据结构和保证数据完整性的约束（例如，通过不接受不符合业务规则的值）。关系模型还定义限制、投影、笛卡儿积和关系联结[Codd, 1970]的关系操作。这些操作让你利用数据去做那些有意义的事情，比如对它进行概述或者导航。

每一个操作都从一张指定的表或者表的组合中产生一张新表。SQL是用来在应用程序中表达这些操作〔因此称作数据语言（data language）〕，以及用来定义在其上面执行操作的基本表的一种语言。

编写SQL数据定义语言（Data Definition Language, DDL）语句来创建和管理表。我们称DDL定义了数据库模式。语句如CREATE TABLE、ALTER TABLE和CREATE SEQUENCE都属于DDL。

编写SQL数据操作语言（Data Manipulation Language, DML）语句以在运行时使用数据。让我们在CaveatEmptor应用程序的一些表的上下文中描述这些DML操作。

在CaveatEmptor中，你一般有像货品（item）、用户（user）和出价（bid）这样的实体。假设这个应用程序的SQL数据库模式包括一个ITEM表和一个BID表，如图A-1所示。这个模式的数据类型、表和约束都通过SQL DDL来创建（CREATE和ALTER操作）。

插入（insertion）是从旧表中通过添加一个行来创建一条新记录的操作。SQL数据库在某处执行这个操作，因此新行被添加到现有的表中：

```
insert into ITEM values (4, 'Fum', 45.0)
```

ITEM		
ITEM_ID	DESCRIPTION	...
1	Item Nr. One	...
2	Item Nr. Two	...
3	Item Nr. Three	...

BID		
BID_ID	ITEM_ID	AMOUNT
1	1	99.00
2	1	100.00
3	1	101.00
4	2	4.99

图A-1 包含示例数据的示例表

SQL update (更新) 修改一个现有的行:

```
update ITEM set PRICE = 47.0 where ITEM_ID = 4
```

deletion (删除) 移除一行:

```
delete from ITEM where ITEM_ID = 4
```

SQL的真正威力在于查询数据。单个查询可以在几张表中执行多个关系操作。来看一下基本的操作。

限制 (restriction) 是选择符合特定条件的表行的操作。在SQL中, 这个条件是发生在where子句中的表达式:

```
select * from ITEM where NAME like 'F%'
```

投影 (projection) 是选择一张表的列、并从结果中消除重复行的操作。在SQL中, 要包括的列都列在select子句中。可以通过指定distinct关键字消除重复的行:

```
select distinct NAME from ITEM
```

笛卡儿积 [Cartesian product, 也称作交叉联结 (cross join)] 生成一张新表, 它由两张现有表的行的所有可能的组合构成。在SQL中, 通过在from子句中列出表来表达笛卡儿积:

```
select * from ITEM i, BID b
```

关系的联结通过组合两张表的行生成新的表。对于联结条件 (join condition) 为真的每一对行, 这张新表都包括一个行, 包含来自两个被联结行的所有字段值。在ANSI SQL中, join子句指定一个表联结, 联结条件在on关键字后。例如, 为了获取所有包含出价的货品, 可以在它们共有的ITEM\_ID属性中联结ITEM和BID表:

```
select * from ITEM i inner join BID b on i.ITEM_ID = b.ITEM_ID
```

联结相当于一个有限制的笛卡儿积。因此, 联结经常改为以theta风格来表达, 通过from子句中的一个乘积和where子句中的联结条件。这个SQL theta风格的联结相当于前一个ANSI风格的联结:

```
select * from ITEM i, BID b where i.ITEM_ID = b.ITEM_ID
```

除了这些基本的操作之外, 关系数据库还为聚合行 (GROUP BY) 和排序行 (ORDER BY) 定义操作:

```
select b.ITEM_ID, max(b.AMOUNT)
from BID b
group by b.ITEM_ID
having max(b.AMOUNT) > 15
order by b.ITEM_ID asc
```

SQL曾被称作结构查询语言, 是关于所谓的子查询 (subselect) 的一项特性。因为每个关系操作都从一张现有的表或者几张表中生成新表, 所以SQL查询可以在前一个查询的结果表中操作。SQL让你用单个查询表达这一点, 通过把第一个查询嵌套在第二个里面:

```
select *
from (
 select b.ITEM_ID as ITEM, max(b.AMOUNT) as AMOUNT
 from BID b
 group by b.ITEM_ID
)
where AMOUNT > 15
order by ITEM asc
```

这个查询的结果相当于前一个结果。

子查询可以出现在SQL语句中的任何地方。在where子句中的子查询，是最值得关注的：

```
select * from BID b
 where b.AMOUNT >= (select max(c.AMOUNT) from BID c)
```

这个查询返回数据库中最大的出价。where子句子查询经常与量词（quantification）结合。

下列查询是等价的：

```
select * from BID b
 where b.AMOUNT >= all(select c.AMOUNT from BID c)
```

SQL限制条件用一种复杂的表达语言表达，它支持数学表达式、函数调用、字符串匹配，甚至更复杂的特性如全文搜索：

```
select * from ITEM i
 where lower(i.DESCRIPTION) like '%gc%'
 or lower(i.DESCRIPTION) like '%excellent%'
```

## 附录 B

# 映射快速参考

许多Hibernate书都在附录中罗列了所有可能的XML映射元素和映射注解。这么做有多大用处值得质疑。首先，这些信息已经能以一种很方便的形式得到，你只需要知道如何得到它。其次，我们在这里可能增加的任何参考在几个月甚至几个星期就会过时了。核心的Hibernate映射策略不会那么经常变动，但是有一些小细节、选项和属性，则在改善Hibernate的过程中始终被修改着。

主要的原因难道不是你想要一个映射参考——所以你有所有选项的最新清单？

- 可以在Hibernate中绑定的**hibernate-mapping-3.0.dtd**中找到所有XML映射元素和属性的清单。在任何文本编辑器中打开这个文件，你会看到它被很好地制成了文档，非常容易阅读。如果使用XML映射文件的话，你可以把它打印出来作为一个快速参考。如果DTD的语法让你觉得困扰，就在这个文件的副本上做几个快速的搜索/替换操作，用你在打印输出中喜欢的东西替换DTD标签。
- 通过阅读Javadoc，可以找到**javax.persistence**和**org.hibernate.annotations**包所有映射注解的一个清单。Javadoc与Hibernate Annotations包捆绑在一起。例如，为了给所有的Hibernate扩展注解找到一个可单击的最新参考，请打开[api.org/hibernate/annotations/package-summary.html](http://api.org/hibernate/annotations/package-summary.html)。