使用 Scala、Lift 和 jQuery 构建 Comet 应用程序

创建拍卖站点

实现

Lift 是一个完整的 Web 应用程序库。它提供完整的模型-视图-控制器（Model-View-Controller，MVC）实现，尽管它使用的方法与大多数普通的 MVC 框架有些不同。它大量使用 Maven 来构建项目结构，从而满足依赖性。这就是为何不需要下载或安装 Scala 就可以使用 Lift 的原因 — 这一切都已经为您准备好了！这还解释了为何不需要数据库或 Web 服务器就可以使用 Lift；它使用 Maven 包含一个数据库（Apache Derby）和一个 Web 服务器（Jetty）。事实上，Jetty 非常适合 Comet 风格的应用程序；Lift 利用了这些特点，因此不需要下载任何东西。

创建 Lift 应用程序

如前所述，Lift 几乎使用 Maven 完成所有事情，包括创建应用程序。它使用 Maven 原型（archetype）创建项目结构。您需要使用的命令是 mvn archetype:generate，以及以下参数：

表 1. 样例事件数据字段

参数	值	描述
archetypeGroupId	net.liftweb	这是您需要使用的原型的名称空间。
archetypeArtifactId	lift-archetype-basic	这是原型的 ID。在这里，它指定一个 “基础” 应用程序，见下。
archetypeVersion	0.10-SNAPSHOT	这是原型的版本，它与 Lift 的版本对应。见下。
remoteRepositories	http://scala-tools.org/repo-snapshots	这是包含原型的 Maven 储存库的 URL。
groupId	org.developerworks	您的应用程序的名称空间。您可以更改这个值。所有代码必须位于这个包的子包中。
artifactId	auctionNet	您的应用程序的名称。

有一些地方需要进一步解释。首先，您使用的是*基本* 原型。除此之外，还有一个*空白* 原型。如果您使用空白原型，Maven 将生成最简单的 Lift 应用程序。这是最简洁的应用程序结构，也包含最少的启动代码。不过您将使用基本的原型，它使用 Lift 的 ORM 技术（Mapper），并设置了 Comet 框架。它还将创建一个用户模型和标准用户管理页面所需的所有代码：注册、登录和忘记密码。许多应用程序都需要这些功能，这个应用程序也不例外，因此您需要这种原型。这能为您提供便利！

其次，注意您使用的原型的版本，它必须与您使用的 Lift 的版本对应。在这个例子中，我使用 0.10 版本的快照。在撰写本文时，最新的官方发布版为 0.9。不过，0.10 版本提供一些很好的特性，因此我选择 “舍新求旧”。不利的一面是，虽然可以下载到这里使用的一些代码，但需要经过小修改才能用于官方 0.10 版本。注意，因为我使用了快照，所以我将使用 “repo-snapshots” 储存库。对于官方版本，也有一个 “repo-releases” 储存库。现在，您已经知道应该向 Maven 发出什么命令，那么让我们运行该命令，看看得到什么结果。清单 1 给出了完整的命令及其输出。

清单 1. 使用 Maven 创建一个 Lift 项目命令

```
$ mvn archetype:generate -DarchetypeGroupId=net.liftweb
-DarchetypeArtifactId=lift-archetype-basic -DarchetypeVersion=0.10-SNAPSHOT
 -DremoteRepositories=http://scala-tools.org/repo-snapshots
-DgroupId=org.developerworks.lift -DartifactId=auctionNet
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] Setting property: classpath.resource.loader.class =>
        'org.codehaus.plexus.velocity.ContextClassLoaderResourceLoader'.
```

```
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[net.liftweb:lift-archetype-basic:RELEASE -> http://scala-tools.org/repo-releases]
found in catalog internal
[INFO] snapshot net.liftweb:lift-archetype-basic:0.10-SNAPSHOT:
       checking for updates from lift-archetype-basic-repo
[INFO] snapshot net.liftweb:lift-archetype-basic:0.10-SNAPSHOT:
       checking for updates from scala-tools.org
[INFO] snapshot net.liftweb:lift-archetype-basic:0.10-SNAPSHOT:
       checking for updates from scala-tools.org.snapshots
Downloading: http://scala-tools.org/repo-snapshots/net/liftweb/lift-archetype-basic/
       0.10-SNAPSHOT/lift-archetype-basic-0.10-SNAPSHOT.jar
15K downloaded
Define value for version:  1.0-SNAPSHOT: :
Confirm properties configuration:
groupId: org.developerworks.lift
artifactId: auctionNet
version: 1.0-SNAPSHOT
package: org.developerworks
 Y: : y
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating OldArchetype: lift-archetype-basic:
      0.10-SNAPSHOT
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: org.developerworks.lift
[INFO] Parameter: packageName, Value: org.developerworks.lift
[INFO] Parameter: basedir, Value: /Users/michael/code/lift/auction2
[INFO] Parameter: package, Value: org.developerworks.lift
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: auctionNet
```

这个命令能够完成您需要做的任何事情，包括下载所需的全部库，以及设置项目的结构。通过运行 mvn jetty:run 命令，您可以立即启动应用程序，如清单 2 所示。

清单 2. 启动应用程序

```
$ cd auctionNet/
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] ------------------------------------------------------------------------
[INFO] Building auctionNet
[INFO]    task-segment: [jetty:run]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing jetty:run
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [yuicompressor:compress {execution: default}]
[INFO] nb warnings: 0, nb errors: 0
[INFO] Context path = /
[INFO] Tmp directory =  determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] Webapp directory = /Users/michael/code/lift/auctionNet/src/main/webapp
[INFO] Starting jetty 6.1.10 ...
2008-12-06 18:11:43.621::INFO:  jetty-6.1.10
```

```
2008-12-06 18:11:44.844::INFO:  No Transaction manager found - if your webapp
requires one, please configure one.
INFO - CREATE TABLE users (id BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY , firstname
VARCHAR(32) , lastname VARCHAR(32) , email VARCHAR(48) , locale VARCHAR(16) , timezone
VARCHAR(32) , password_pw VARCHAR(48) , password_slt VARCHAR(20) , textarea
VARCHAR(2048) , superuser SMALLINT , validated SMALLINT , uniqueid VARCHAR(32))
INFO - ALTER TABLE users ADD CONSTRAINT users_PK PRIMARY KEY(id)
INFO - CREATE INDEX users_email ON users ( email )
INFO - CREATE INDEX users_uniqueid ON users ( uniqueid )
2008-12-06 18:11:47.199::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 5 seconds.
```

您将注意到一些被加载的库，并且将编译在上个步骤中生成的所有代码，然后将其打包到一个 WAR 文件。清单 2 删除了这些信息。您需要知道的一点是，Lift 将为您创建一个数据库。还记得吗，使用基本原型将自动获得一个用户管理系统。我刚才已经提到这点。针对该系统的数据库表是在清单 2 中创建的。现在，关于用户的工作已经准备完毕。接下来，您仅需为物品和出价创建一个域模型。

使用 Mapper 进行域建模

Lift 提供自己的对象关系建模（ORM）技术，即 Mapper。正如您在前一个例子中所见，您已经在用户管理中使用这种技术。可以在 <groupId>.model.User 中找到生成的 User 代码，您刚才使用 Maven 生成应用程序时就使用了其中的 groupId。在这个例子中，groupId 为 org.developerworks，因此用户模型保存在 org.developerworks.model.User 中。可以在 /auctionNet/src/main/scala 中找到所有 Scala 代码。这个用户模型如清单 3 所示。

清单 3. 默认的 Lift User 模型

```
/**
 * The singleton that has methods for accessing the database
 */

object User extends User with MetaMegaProtoUser[User] {
  override def dbTableName = "users" // define the DB table name
  override def screenWrap = Full(<lift:surround with="default"
at="content"><lift:bind /></lift:surround>)
  // define the order fields will appear in forms and output
  override def fieldOrder = id :: firstName :: lastName :: email ::
    locale :: timezone ::
  password :: textArea :: Nil
  // comment this line out to require email validations
  override def skipEmailValidation = true
}

/**
 * An O-R mapped "User" class that includes first name, last name, password
 * and we add a "Personal Essay" to it
 */

class User extends MegaProtoUser[User] {
  def getSingleton = User // what's the "meta" server
  // define an additional field for a personal essay
  object textArea extends MappedTextarea(this, 2048) {
    override def textareaRows  = 10
    override def textareaCols = 50
    override def displayName = "Personal Essay"
  }
```

```
}
```

通过这段代码可以很好地理解 Lift 的 Mapper API。我们看看清单 3 底部的 User 类。它扩展了现有的 MegaProtoUser 类。您可以在 Lift 源代码中看看这个类，但是从代码的注释可以了解到，它提供了姓、名和密码。这段代码显示如何定制用户。在这个例子中，它添加一个映射到数据库列 textArea 的 “Personal Essay”。

Mapper 的一个特别之处就是，映射类中的字段（数据库列）不是一般的 Scala 字段，比如 var 或（如果该字段是不可变的）val。相反，它们是 Scala 对象，即单实例对象（singleton）。您可以将它们看作嵌套在封装类内部的单实例对象，因此可以有多个 Users（因为它是一个类），但每个 User 只能有一个 textArea 对象。在 User 类中可以看到使用单个对象的好处。您的对象扩展 Lift 类 net.liftweb.mapper.MappedTextarea。通过将现有的类作为子类，您可以覆盖行为以定制字段。在 User 类中，这样做能够改变将这个字段表示为 HTML TextArea 元素的方式。没错，所有 Mapper 字段类型（MappedString 和 MappedLong 等等）都有一个内置的 HTML 表示。例如清单 4 中的 MappedTextarea 类。

清单 4. Lift 的 MappedTextarea 类

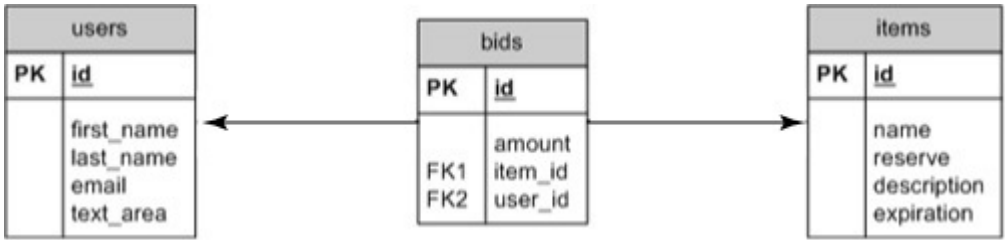
```
class MappedTextarea[T<:Mapper[T]](owner : T, maxLen: Int) extends
      MappedString[T](owner, maxLen) {

  /**
   * Create an input field for the item
   */
  override def _toForm: Can[NodeSeq] = {
    val funcName = S.mapFunc({s: List[String] => this.setFromAny(s)})
    Full(<textarea name={funcName}
      rows={textareaRows.toString}
      cols={textareaCols.toString} id={fieldId}>{is.toString}</textarea>)
  }
  override def toString = {
    val v = is
    if (v == null || v.length < 100) super.toString
    else v.substring(0,40)+" ... "+v.substring(v.length - 40)
  }
  def textareaRows  = 8
  def textareaCols = 20
}
```

如您所见，MappedTextarea 扩展 MappedString，因此，当它映射到数据库列时将被看作是字符串。它有一个 _toForm 方法，该方法使用 Scala 的 XML 语法和 Lift 的帮助函数来创建一个典型的 HTML TextArea。在 User 类中，您将覆盖多个行和列，以及显示的名称。当把映射类型作为子类时，您将获得巨大的威力。在此处添加定制验证逻辑是非常理想的。假如您不希望在评论中显示任何 HTML 字符，那么您可以插入一个能够执行检查的正则表达式。MappedString 类包含用于执行正则表达式的方法，也包含一些能够检查最短和最长长度，甚至能够根据数据库检测字符串的惟一性的方法。当然，将类型进一步分类之后，您可以完成更加复杂的逻辑，以及添加任何代码。

不失所望，User 类是非常有用的。回顾一下清单 3 中的 User 类，就可以看到一个称为 getSingleton 的方法。这个方法返回在以上的 User 类中定义的 User 对象。这个对象表示关于 User 类的元数据，以及如何将它映射到数据库。在这里，需要定义数据库表的名称，以及用于列出 Users 或生成 User 表单的字段。此外，User 对象提供一个可以添加类级别的方法的位置，比如 factory 和 finder 方法。Scala 没有静态的变量和方法，因此不能将它们直接关联到 User 类。

您可能已经注意到，这里的对象和类都称为 User。这种模式在 Scala 中很常见，即配对（companion）对象。在定制模型中，您可能会稍微打破这种约定，让一些代码变得更明显。您将附加一个 “MetaData” 前缀。现在您已经了解 Lift 的 Mapper API，接下来我们将创建一些定制模型。首先看看图 1，它展示了您想创建的数据模型。

图 1. Auction Net 数据模型



这是一个很简单的拍卖模型。每个物品有一个名称和一条描述，以及保留价格（最低价格）和有效期。任何用户都可以对所有物品出价竞购，因此这是一个多对多的关系。在关系数据库模式中，出价就像一个连接表，但它们自身有实际的意义，因为每个出价都有一个数量。现在您已经知道对什么进行建模，接下来我们看看 Item 模型。如清单 5 所示。

清单 5. Item 模型

```
object ItemMetaData extends Item with KeyedMetaMapper[Long, Item]{
     override def dbTableName = "items"
     override def fieldOrder = List(name, description, reserve, expiration) }
class Item extends KeyedMapper[Long, Item] with CRUDify[Long, Item] {
     def getSingleton = ItemMetaData
     def primaryKeyField = id

     object id extends MappedLongIndex(this)
     object reserve extends MappedInt(this)
     object name extends MappedString(this, 100)
     object description extends MappedText(this)
     object expiration extends MappedDateTime(this)

     lazy val detailUrl = "/details.html?itemId=" + id.is
     def bids = BidMetaData.findAll(By(BidMetaData.item, id))

     def tr = {

      <tr>
        <td><a href={detailUrl}>{name}</a></td>
        <td>{highBid.amount}</td>
        <td>{expiration}</td>
      </tr>
  }
  def highBid:Bid = {
    val allBids = bids
    if (allBids.size > 0){
      return allBids.sort(_.amount.is > _.amount.is)(0)
    }
    BidMetaData.create
  }

}
```

我们再次从 Item 类开始。您看到的大部分代码都是非常标准的。将每个字段定义为扩展映射数据类型（long、integer、string 和 date 等）的对象。将另一个属性 detailUrl 定义为 lazy val。这仅表示它是不可变的值，在调用之前不能计算它。这个语法在这里影响不大，但对于计算很复杂并且不经常调用的情况，它就十分有用了。这里有一个能够查询所有出价的 bid 方法，随后我们就介绍 Bid 类。还有一个称为 tr 的方法，它在 HTML 表中将物品表示为一个行。

最后，还有一个 highBid 方法，它对来自数据库的所有出价进行排序，找到最高的出价。这个排序可以由服务器完成，但它展示了 Scala 能够简化常见事务，比如对列表进行排序。先向 sort 方法传递一个闭包，然后使用 Scala 闭包的快捷语法。(_) 中的下划线由传递到闭包的参数填充。对于出价列表，有两个可以比较的物品。您希望根据出价额对出价进行排序，因此使用 _.amount.is。最后的部分 (.is) 调用 amount 对象上的 is 方法。记住，这个对象非常复杂，而不是简单的字段，因此 is 方法获取该字段的值。排序完成之后，要注意 (0) 语法。排序生成一个出价列表。您希望获取列表的第一个元素，而 (0) 帮

助您得到它。这展示了 Scala 的快捷性。实际上您在该列表上使用值 0 调用 apply 方法。这个列表被看作是一个函数，这对 apply 方法而言通常就是语法糖。

我已经解释了 Item 类的大部分内容，但还没有声明它。首先声明 Item 扩展 Lift 特征 KeyedMapper。这是一个参数化特征，其中参数就是映射类的主键和映射类本身的类型。注意，您使用另一个特征 CRUDify 扩展了 KeyedMapper。您正在利用 Scala 的糅合（mix-in）模型来模拟多重继承。您需要获取 KeyedMapper 和 CRUDify 的行为。注意，CRUDify 也是一个参数化特征。这是 Scala 中的另一种常见现象。通过参数化特征来确保它们的类型是安全的。如果您查看其他语言（例如 Python 和 Ruby）的糅合（mix-in），就会发现它们非常繁琐，或者无法表达需求（比如要求使用某些字段或方法）。将糅合添加到自己的类中时，在出现运行时错误之前，您并没有意识到需要修改自己的类才能使用它们。Scala 中的参数化特征避免了这种问题。

您的 Item 模型具有 KeyedMapper 和 CRUDify 特征。KeyedMapper 特征被映射到数据库表，但 CRUDify 呢？根据它的名称可以知道，它为模型提供基本的 CRUD：Create、Read、Update 和 Delete 函数。Lift 将创建用于显示 Items 列表的所有模板代码，创建新的 Item，以及编辑或删除现有的 Item。在函数设计中，您要求能够列出新的物品，CRUDify 能够帮助您轻松实现这个目标。不需要编写额外的代码，但可以使用额外的特性！现在您已经了解 Item 模型，接下来我们看看 Bid 模型。如清单 6 所示。

清单 6. Bid 模型

```
object BidMetaData extends Bid with KeyedMetaMapper[Long, Bid]{
  override def dbTableName = "bids"
  override def fieldOrder = amount :: Nil
  override def dbIndexes = Index(item) :: Index(user) :: super.dbIndexes
}
class Bid extends KeyedMapper[Long, Bid]{
  def getSingleton = BidMetaData
  def primaryKeyField = id
  object id extends MappedLongIndex(this)
  object amount extends MappedLong(this)
  object item extends MappedLongForeignKey(this, ItemMetaData)
  object user extends MappedLongForeignKey(this, User)

}
```

这是一个更加简单的类。它具有一个 Item 对象和一个 User 对象。这些对象扩展 Lift 的参数化类 MappedLongForeignKey。注意如何传入元数据对象（来自清单 5 的 ItemMetaData 对象和来自清单 3 的 User 对象），以表明您将加入到哪里。此外还要注意，在元数据对象中，您在两个外键列上指定了数据库索引，希望根据物品或用户查询出价。现在已经定义好域模型，您可以编写代码来使用它了。

Actor

我使用 Lift 的 CRUDify 特征处理物品管理，并对用户管理使用 Lift 的开箱即用支持，因此您仅需构建一个出价系统。您可以使用一般的控制器/CRUD 代码完成这个系统，但您希望它是一个 Comet 系统，所以您将使用 Scala 的并发性堆栈 Actor。Actor 就像一个轻量级线程，但它没有共享内存，因此不需要进行同步和锁定等。Actor 与消息进行通信。Scala 组合了 case 类（主要是类型数据结构）和模式匹配，从而使它能够更轻松地监听和响应消息。您首先创建一个 Auctioneer actor。它将监听对物品出价的消息，并发出消息表明有新的出价。我们先看看将使用的消息的类型。这些消息如清单 7 所示。

清单 7. 拍卖消息

```
case class AddListener(listener:Actor, itemId:Long)
case class RemoveListener(listener:Actor, itemId:Long)
case class BidOnItem(itemId:Long, amount:Long, user:User)
case class GetHighBid(item:Item)
case class TheCurrentHighBid(amount:Long, user:User)
case class Success(success:Boolean)
```

前两个消息仅用于根据 Item ID 添加或删除监听器。您将向 Auctioneer 发送一条 AddListener 消息，表明您对某款物品感兴趣。如果您想出价，可以发送一条 BidOnItem 消息。如果有新的出价，您希望 Auctioneer 发送一条新的 TheCurrentHighBid 消息。最后，发出 Success 消息表明一个 AddListener 请求已经成功。现在，您可以根据这些强类型对象执行模式匹配。我们看看清单 8 中的 Auctioneer。

清单 8. Auctioneer actor

```
object Auctioneer extends Actor{
  val listeners = new HashMap[Long, ListBuffer[Actor]]
  def notifyListeners(itemId:Long) = {
    if (listeners.contains(itemId)){
      listeners(itemId).foreach((actor) => {
        val item = ItemMetaData.findByKey(itemId).open_!
        actor ! highBid(item)
     })
    }
  }
  def act = {
    loop {
      react {
        case AddListener(listener:Actor, itemId:Long) =>
          if (!listeners.contains(itemId)){
            listeners(itemId) = new ListBuffer[Actor]
          }
          listeners(itemId) += listener
          reply(Success(true))
        case RemoveListener(listener:Actor, itemId:Long) =>
          listeners(itemId) -= listener
        case GetHighBid(item:Item) =>
          reply(highBid(item))
        case BidOnItem(itemId:Long, amount:Long, user:User) =>
          val item =
          ItemMetaData.findAll(By(ItemMetaData.id, itemId)).firstOption.get
          val bid = BidMetaData.create
          bid.amount(amount).item(item).user(user).save
          notifyListeners(item.id)
      }
    }
  }

  def highBid(item:Item):TheCurrentHighBid = {
    val highBid = item.highBid
    val user = highBid.user.obj.open_!
    val amt = highBid.amount.is
    TheCurrentHighBid(amt, user)
  }
  start
}
```

Auctioneer 通过保持一个映射来跟踪谁对那款物品感兴趣。这个映射的关键是物品的 ID，它的值为相关的 Actors 列表。act 方法是所有 Actor 的主要部分。这是您必须实现的 Actor 特征中的抽象方法。loop -> react 是 Actors 的典型构造。函数循环是在 scala.actors.Actor 对象中定义的；它接受一个闭包并反复执行它。此外，还有一个带有谓词的 loopWhile，只要谓词为 true，它就会一直循环下去。这提供了一个钩子，从而能够轻松关闭 Actor。react 方法是在 scala.actors.Actor 特征中定义的。它接收一个消息，并执行传递给它的闭包。您正是在这个闭包的内部使用 Scala 的模式匹配。您将根据进来的消息的类型执行匹配。尤其是收到 BidOnItem 消息时，您将把新的出价保存到数据库，然后通知监听器。

notifyListeners 方法使用 Item ID 的映射获取对某款物品感兴趣的所有 Actors。然后，它向每个 Actor

发送一条新的 TheCurrentBid 消息。这就是 actor ! highBid(item) 代码的工作。（事实上，这可以写成 actor.!(highBid(item))）。换句话说，Actor 类有一个 ! 方法。actor ! highBid(item) 语法更加美观，并且与其他语言（比如 Erlang）实现 actor 的方式保持一致。这些语言有支持 actor 的特定语法，但 Scala 没有。Actor 本质上是一种领域特定语言（Domain Specific Language，DSL），它使用自己的强大语法构建在 Scala 之上。

回过头来看看 Auctioneer，需要注意的地方是最后一行代码。在这里，将调用 start 方法来启动 Actor，从而导致异步地调用 Actor 的 act 方法。您的 Auctioneer 将一直运行下去，向其他 Actors 发送消息，或接收来自它们的消息。如果习惯了 Java 的并发编程，情况就会大不相同，但在很多地方会更简单。现在，需要回答的问题就是谁将向 Auctioneer 发送消息，或接收来自它的消息？毕竟，应用程序中仅有一个 Actor 是比较枯燥的。为了回答这个问题，您需要用到 Lift 的 CometActors。

CometActor

Lift 的 CometActor 特征是 Scala 的 Actor 特征的扩展。它能够轻松地在 Comet 应用程序中使用 Actor。CometActor 可以响应来自其他 Actors 的消息，从而向用户发送 UI 更新。Maven 原型创建一个 Comet 包，放置在其中的任何 CometActors 都可以自动地在应用程序中使用。对于 Auction Net，将创建一个 CometActor，它可以发出竞购价格（出价），以及接收最新的更高出价的更新。我将这个 Actor 称为 AuctionActor，如清单 9 所示。

清单 9. Auction Actor

```
class AuctionActor extends CometActor {
  var highBid : TheCurrentHighBid = null
  def defaultPrefix = "auction"
  val itemId = S.param("itemId").map(Long.parseLong(_)).openOr(0L)
  override def localSetup {
    Auctioneer !? AddListener(this, this.itemId) match {
      case Success(true) => println("Listener added")
      case _ => println("Other ls")
    }
  }
  override def localShutdown {
    Auctioneer ! RemoveListener(this, this.itemId)
  }
  override def lowPriority : PartialFunction[Any, Unit] = {
    case TheCurrentHighBid(a,u) => {
      highBid = TheCurrentHighBid(a,u)
      reRender(false)
    }
    case _ => println("Other lp")
  }
}
```

这个清单显示了 CometActor 的生命周期。调用它时，将导致调用 localSetup 方法。它使用 itemId 属性向 Auctioneer Actor 发送一条 AddListener 消息。注意，我使用 !? 发送这条消息。这是一个同步调用。除非 Auctioneer 知道它对特定物品感兴趣，否则 AuctionActor 是毫无用处的。当收到来自 Auctioneer 的回复之后，您将使用模式匹配来决定应该怎么做，然后记录相关的信息。如果不再使用 CometActor 时，将调用 localShutdown 方法。它将向 Auctioneer 发送一条 RemoveListener 消息。在 Auction Actor 的存活期之内，它将使用 lowPriority 方法（还有 highPriority 方法等）监听消息。在这里，当您收到消息时，Auction Actor 将使用模式匹配。它查找 TheCurrentHighBid 消息，并通过跟踪最高出价和调用 reRender 来响应它。这个方法在 CometActor 上定义。在清单 9 中，为了集中讲解生命周期，我省略了呈现代码。现在，我们看看清单 10 中给出的呈现代码。

清单 10. Auction Actor 的呈现

```
class AuctionActor extends CometActor {
```

```
  var highBid : TheCurrentHighBid = null
  def defaultPrefix = "auction"
  val itemId = S.param("itemId").map(Long.parseLong(_)).openOr(0L)
  def render = {
    def itemView: NodeSeq = {
      val item = if (itemId > 0)
        ItemMetaData.findByKey(itemId).openOr(ItemMetaData.create)
        else ItemMetaData.create
      val currBid = item.highBid
      val bidAmt = if (currBid.user.isEmpty) 0L else currBid.amount.is
      highBid = TheCurrentHighBid(bidAmt,
        currBid.user.obj.openOr(User.currentUser.open_!))
     val minNewBid = highBid.amount + 1L
     val button = <button type="button">{S.?("Bid Now!")}</button> %
        ("onclick" -> ajaxCall(JsRaw("$('#newBid').attr('value')"), bid _))
     (<div>
        <strong>{item.name}</strong>
        <br/>
        <div>
          Current Bid: ${highBid.amount} by {highBid.user.niceName}
        </div>
        <div>
          New Bid (min: ${minNewBid}) :
          <input type="text" id="newBid"/>
          {button}
        </div>
       {item.description}<br/>
     </div>)
   }
   bind("foo" -> <div>{itemView}</div>)
    }
   def bid(s:String): JsCmd = {
     val user = User.currentUser.open_!
     Auctioneer ! BidOnItem(itemId, Long.parseLong(s), user)
     Noop
   }
}
```

render 方法是您必须实现的 CometActor 特征的抽象方法。您将在这个方法中查找物品，然后创建一个 XHTML 片段将它发送给用户。在这里，最有趣的代码部分是按钮的值，它将创建一个基本的 “Bid Now!” 按钮。S.? 函数允许本地化这个字符串，以免给您带来困惑。% 向该元素添加一个属性。在这个例子中，您将创建一个 JavaScript 函数响应来自该按钮的 onclick 事件。ajaxCall 函数在 Lift 的 SHtml 帮助对象中定义。它所需的第一个参数是一个 JsExp（JavaScript 表达式）实例。您将使用 JsRaw 对象打包原始的 JavaScript 字符串，并创建一个 JsExp。

如果您熟悉 jQuery，那么传递的原始 JavaScript 是相当简单的。Lift 默认包含 jQuery 库。jQuery 中的 $ 函数接受一个 CSS 选择器，并返回 DOM 树中满足该选择器的元素。在这个例子中，您将传入 #newBid，它在 CSS 中指定一个 ID 为 newBid 的元素。jQuery 库向 DOM 元素添加一个 attr 函数。这为获取元素的属性值提供一种简易的方法。在这个例子中，您需要 value 属性。如果您仔细查看代码，将发现 newBid 元素是一个文本输入字段，因此 value 属性就是用户向文本输入字段输入的内容。

将计算 jQuery 表达式的值，并将其传递给 Ajax 调用。ajaxCall 函数的第二个参数是一个闭包，它接受一个字符串并返回一个 Lift JsCmd 类型的实例。在这里，您将再一次使用 Scala 的便捷语法，使闭包将字符串传递给您定义的 bid 函数。bid 函数获取最新用户，将字符串解析为 long 类型，并使用这两个值构造一个 BidOnItem 消息。它将这个消息发送给 Auctioneer。然后，它返回 Lift 的 Noop 对象。这是一个 JsCmd，它表明 no op（没做什么事情）。

现在，您可能已经注意到，bid 方法将 itemId 作为 BidOnItem 消息的一部分。您可能想知道当用户对某款物品出价时，这个值是否还在。这正是闭包的奇妙之处；它们在创建封装上下文之后就一直保留它。当执行传递到 ajaxCall 的闭包时，它将 “记住” 它被创建时所获取的所有数据。

现在，您已经知道 Auction Actor 是如何工作的，接下来您将创建一个能够使用它的页面。这个页面就是

关于物品细节的页面，如清单 11 所示。

清单 11. 物品细节视图

```
<lift:surround with="default" at="content">
  <lift:comet type="AuctionActor">
    <auction:foo>Loading...</auction:foo>
  </lift:comet>
</lift:surround>
```

如您所见，这是一个很简单的页面。您使用了 Lift 的默认页面布局，仅为 Auction Actor 包含一小段代码。如果您对 auction:foo 标记感兴趣，请参见清单 10。在清单 10 中，我将 CometActor 的 defaultPrefix 定义为 auction，并使用 bind 命令将用于呈现的 XHTML 绑定到 foo，在清单 11 中是 auction:foo。这将异步加载到页面，因此在加载 CometActor 期间，需要使用 “Loading...” 文本作为占位符。

为了将页面添加到站点，您需要将它添加到应用程序的 SiteMap。这是另一个 Lift 结构，它包含允许访问的页面的白名单，并且允许 Lift 构造导航菜单和痕迹导航（breadcrumb）等。它是在 Lift 的 Boot 类中指定的，如清单 12 所示。

清单 12. Lift 启动代码

```
class Boot {
  def boot {
    if (!DB.jndiJdbcConnAvailable_?)
      DB.defineConnectionManager(DefaultConnectionIdentifier, DBVendor)
    // where to search snippet
    LiftRules.addToPackages("org.developerworks")
    Schemifier.schemify(true, Log.infoF _, User, ItemMetaData, BidMetaData)
    // Build SiteMap
    val entries:List[Menu] = Menu(Loc("Home", List("index"), "Home")) ::
        Menu(Loc("Item Details", List("details"), "Item Details", Hidden)) ::
        User.sitemap ++ ItemMetaData.menus
    LiftRules.setSiteMap(SiteMap(entries:_*))
/*
     * Show the spinny image when an Ajax call starts
     */
    LiftRules.ajaxStart =
      Full(() => LiftRules.jsArtifacts.show("ajax-loader").cmd)
/*
     * Make the spinny image go away when it ends
     */
    LiftRules.ajaxEnd =
      Full(() => LiftRules.jsArtifacts.hide("ajax-loader").cmd)
    LiftRules.appendEarly(makeUtf8)
    S.addAround(DB.buildLoanWrapper)
  }
  /**
   * Force the request to be UTF-8
   */
  private def makeUtf8(req: HttpServletRequest) {
    req.setCharacterEncoding("UTF-8")
  }
}
```

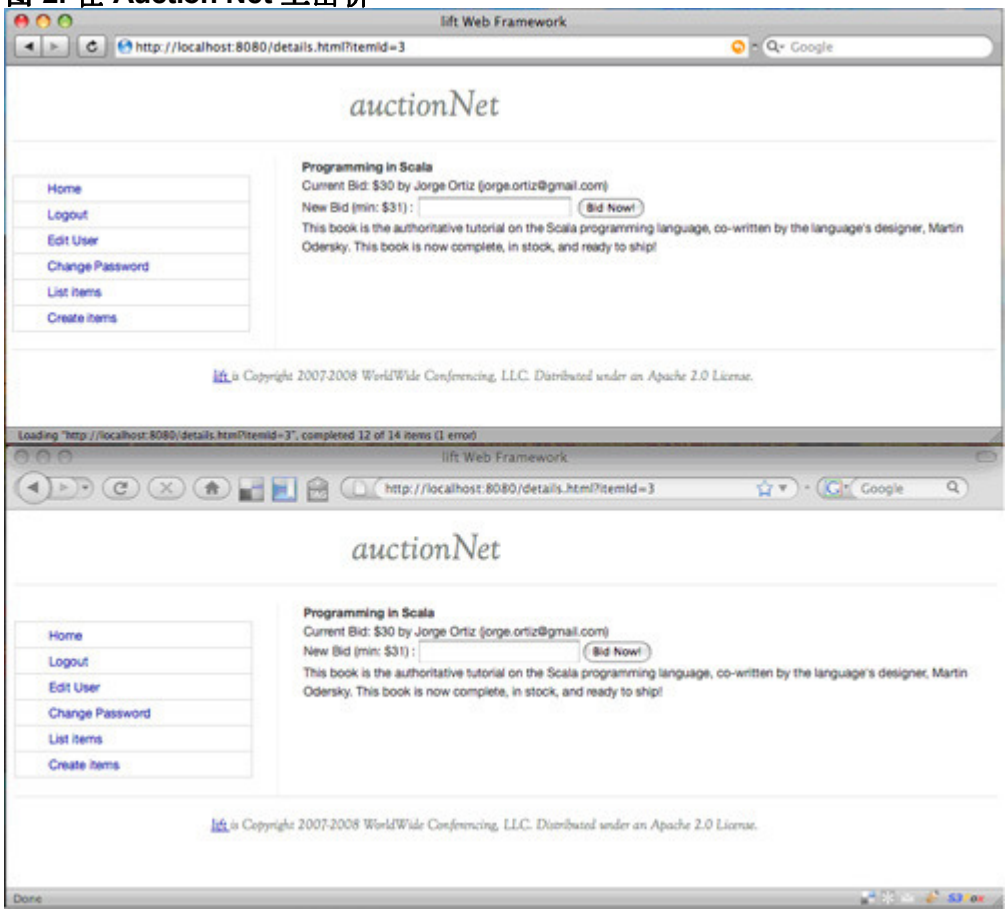
这个文件（Boot.scala）也包含能够更改数据库设置的代码，但在这个应用程序中不进行任何更改。以上代码中，大部分都是作为 Maven 原型的一部分生成的。我们更改了两个主要的地方。首先，Item 和 Bid 元数据对象被添加到对 Schemifier 的调用。这个过程将为您创建数据库。其次，在用于构建 SiteMap 的

条目中，为物品细节页面添加一个新位置，并且为 Item 添加 CRUD 页面。代码已经准备完毕，您可以运行应用程序了！

运行应用程序

如果应用程序已经运行，可以运行 mvn install 重新编译代码。如果应用程序还没有运行，可以再次运行 mvn jetty:run 启动它。您必须能够在 http://localhost:8080 上访问该页面。您可以注册、创建一些物品并开始出价。要查看 Comet 特性的实际效果，需要分别在两个浏览器中作为不同的用户登录，然后查看同一个物品。每个用户都可以针对同一款物品出价，并且其中一个用户出价时，将同时更新这两个用户。图 2 是一个屏幕截图。

图 2. 在 Auction Net 上出价



如果您想知道在出价时将会发生什么事情，那么可以使用 Firefox 的 Firebug 等工具查看 HTTP 流量。清单 13 给出了示例输出。

清单 13. Comet 流量

```
try { destroy_LC2EAICJRLWEKDM4EXIPE2(); } catch (e) {}
try{jQuery('#LC2EAICJRLWEKDM4EXIPE2').each(function(i) {
    this.innerHTML = '\u000a    <div><div>\u000a
 <strong>Programming in Scala</strong>\u000a';});} catch (e) {}
try { /* JSON Func auction $$ F1229133085612927000_NGB */
    function F1229133085612927000_NGB(obj) {
        lift_ajaxHandler('F1229133085612927000_NGB='+
encodeURIComponent(JSON.stringify(obj)),
null,null);} } catch (e) {}
try { destroy_LC2EAICJRLWEKDM4EXIPE2 = function() {}; } catch (e) {}

lift_toWatch['LC2EAICJRLWEKDM4EXIPE2'] = '11';
```

从清单 13 可以看到，Lift 发送回一个 JavaScript 并执行它。该脚本再次使用 jQuery 将原来的呈现内容替换为新的呈现内容。这正是 jQuery('#...').each 表达式需要完成的事情。它使用那个难看的 ID（考虑到安全因素，由 Lift 随机生成；Lift 的另一个出色特性）查找元素，并向找到的每个元素传递一个闭包。这个闭包将 innerHTML 替换为服务器生成的 HTML。为了保持简洁和增强可读性，清单 13 对完整的 HTML 进行了删减。然后，它发起另一个 Comet 会话。这完全是由 Lift 自动创建的，不需要您做任何事情！
