
Spring快速入门教程

开发第一个Spring程序

目录

概述	1
下载Struts和Spring	3
创建项目目录和Ant Build文件	3
Tomcat和Ant	4
为持久层编写单元测试	7
配置Hibernate和Spring	9
Equinox中Spring是如何配置的	11
用Hibernate实现UserDAO	13
进行单元测试，用DAO验证CRUD操作	14
创建Manager，声明事务处理	15
对Struts Action进行单元测试	19
为web层创建Action和Model(DynaActionForm)	20
运行单元测试，验证Action的CRUD操作	25
填充JSP文件，这样可以通过浏览器来进行CRUD操作	26
通过浏览器验证JSP的功能	27
用Commons Validator添加验证	29
在struts-config.xml中添加ValidatorPlugin	29
创建validation.xml，指定lastName为必填字段	29
把 DynaActionForm 改为 DynaValidatorForm	30
为save()方法设置验证(validation)	30

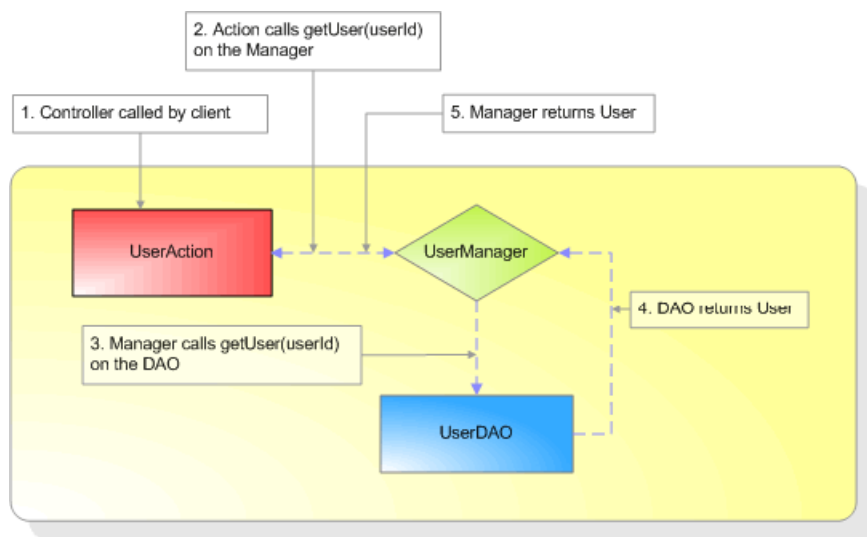
本章学习用Struts MVC框架作前端，Spring作中间层，Hibernate作后端来开发一个简单的Spring应用程序。在第4章将使用Spring MVC框架对它进行重构。

- 编写功能测试。
- 配置Hibernate和Transaction。
- 载入Spring的ApplicationContext.xml文件。
- 设置业务委派(business delegates)和DAO的依赖性。
- 集成Spring和Struts。

概述

你将会创建一个简单的程序完成最基本的CRUD(Create, Retrieve, Update和Delete)操作。这个程序叫MyUsers，作为本书的样例。这是一个三层架构的web程序，通过一个Action来调用业务委派，再通过它来回调DAO类。下面的流程图表示了MyUsers是如何工作的。数字表明了流程的先后顺序，从web层(UserAction)到中间层(UserManager)，再到数据层(UserDAO)，然后返回。

图 1. MyUsers应用程序流程



鉴于大多数读者都比较熟悉Struts，本程序采用它作为MVC框架。Spring的亮点之一就是它声明式的事务处理，依赖性的绑定和持久性的支持(如Hibernate和iBATIS)。第4章中将用Spring框架对它进行重构。

接下来会完成以下几个步骤：

1. 下载Struts和Spring。
2. 创建项目目录和Ant Build文件。
3. 为持久层创建一个单元测试(unit test)。
4. 配置Hibernate和Spring。
5. 编写Hibernate DAO的实现。
6. 进行单元测试，通过DAO验证CRUD。
7. 创建一个Manager来声明事务处理。
8. 为Struts Action 编写测试程序。
9. 为web层创建一个Action和model(DynaActionForm)。
10. 进行单元测试，通过Action验证CRUD。
11. 创建JSP页面，以通过浏览器来进行CRUD操作。
12. 通过浏览器来验证JSP页面的功能。
13. 用Velocity模板替换JSP页面。
14. 使用Commons Validator进行验证。

下载Struts和Spring

1. 下载安装以下组件：
 - JDK 1.4.2(或以上)
 - Tomcat 5.0+
 - Ant 1.6.1+
2. 设置以下环境变量：
 - JAVA_HOME
 - ANT_HOME
 - CATALINA_HOME
3. 把以下路径添加到PATH中：
 - JAVA_HOME/bin
 - ANT_HOME/bin
 - CATALINA_HOME/bin

为了开发基于Java的web项目，开发人员必须事先下载必需的jars，准备好开发目录结构和Ant build文件。对于单一的Struts项目，可以利用Struts包中现成的struts-blank.war。对于基于Spring MVC框架的项目，可以用Spring中自带的webapp-minimal.war。这些都是不错的起点，但两者都没有进行Struts-Spring集成，也没有考虑单元测试。为此，我们为读者准备了Equinox。

Equinox为开发Struts-Spring的程序提供一个基本框架。它已经定义好了目录结构，和Ant build文件(针对compiling,deploying,testing)，并且提供了Struts，Spring，Hibernate开发要用到的jars文件。Equinox中大部分目录结构和Ant build文件来自我的开源项目——AppFuse。可以说，Equinox是一个简化版本的AppFuse，它在最小配置情况下，为快速web开发提供了便利。由于Equinox源于AppFuse，所以在包名，数据库名，及其它地方都找到类似的地方。这样做的目的为了让你从基于Equinox的程序过渡到更为复杂的AppFuse。

从SourceBeat [<http://sourcebeat.com/downloads>]上下载Equinox，解压到一个合适的位置，开始准备MyUsers的开发。

创建项目目录和Ant Build文件

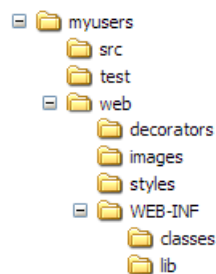
为了设置初始的目录结构，把下载的Equinox解压到硬盘上。建议Windows用户把项目放在C:\Source，UNIX/Linux用户放在~/dev(译注：在当前用户目录建一个dev目录)中。Windows用户可以设置一个HOME环境变量，值为C:\Source。最简单的方法是把Equinox解压到你的喜欢的地方，进入equinox目录，从命令行运行`ant new -Dapp.name=myusers`。

注意

在Windows系统上，我使用了Cygwin(www.cygwin.org)，这样就可以像UNIX/Linux系统一样使用正斜杠，本书所有路径均采用正斜杠。其它使用反斜杠系统(如Windows中命令行窗口)的用户请作相应的调整。

现在MyUsers程序已经有如下的目录结构：

图 2. MyUsers应用程序目录结构



Equinox包含一个简单而功能强大的build.xml，它可以用Ant来进行编译，部署，和测试。要查看所有可用的Ant target,在MyUsers目录下键入ant,回车后将看到如下内容：

```

[echo] Available targets are:
[echo] compile --> Compile all Java files
[echo] war --> Package as WAR file
[echo] deploy --> Deploy application as directory
[echo] deploywar --> Deploy application as a WAR file
[echo] install --> Install application in Tomcat
[echo] remove --> Remove application from Tomcat
[echo] reload --> Reload application in Tomcat
[echo] start --> Start Tomcat application
[echo] stop --> Stop Tomcat application
[echo] list --> List Tomcat applications
[echo] clean --> Deletes compiled classes and WAR
[echo] new --> Creates a new project
  
```

Equinox支持Tomcat的Ant task(任务)。这些task已经集成在Equinox中，解讲一下如何进行集成的有助于理解它们的工作原理。

Tomcat和Ant

Tomcat中定义了一组任务，可以通过Manager程序来安装(install)，删除(remove)，重载(reload)webapps。要使用这些任务，可以把所有的定义写在一个property文件中。在Equinox的根目录下，有一个名为tomcatTasks.properties的文件，其内容如下。

```

deploy=org.apache.catalina.ant.DeployTask
undeploy=org.apache.catalina.ant.UndeployTask
remove=org.apache.catalina.ant.RemoveTask
reload=org.apache.catalina.ant.ReloadTask
  
```

```
start=org.apache.catalina.ant.StartTask
stop=org.apache.catalina.ant.StopTask
list=org.apache.catalina.ant.ListTask
```

在build.xml定义一些task来安装，删除，重新加载应用程序。

```
<!-- Tomcat Ant Tasks -->
<taskdef file="tomcatTasks.properties">
  <classpath>
    <pathelement
      path="${tomcat.home}/server/lib/catalina-ant.jar"/>
    </classpath>
  </taskdef>
<target name="install" description="Install application in Tomcat">
  depends="war">
    <deploy url="${tomcat.manager.url}"
      username="${tomcat.manager.username}"
      password="${tomcat.manager.password}"
      path="/${webapp.name}"
      war="file:${dist.dir}/${webapp.name}.war"/>
  </target>
<target name="remove" description="Remove application from Tomcat">
  <undeploy url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"
    path="/${webapp.name}"/>
</target>
<target name="reload" description="Reload application in Tomcat">
  <reload url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"
    path="/${webapp.name}"/>
</target>
<target name="start" description="Start Tomcat application">
  <start url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"
    path="/${webapp.name}"/>
</target>
<target name="stop" description="Stop Tomcat application">
  <stop url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"
    path="/${webapp.name}"/>
</target>
<target name="list" description="List Tomcat applications">
  <list url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"/>
</target>
```

在上面列出的target中，必须预先定义一些\${tomcat.*}变量。在根目录下有一个build.properties默认定义如下：

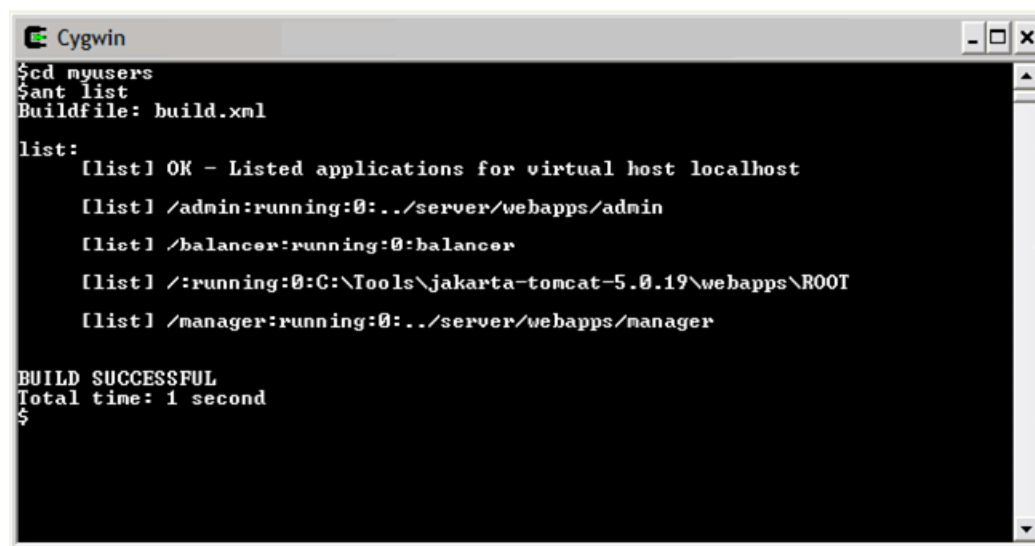
```
# Properties for Tomcat Server
tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=admin
tomcat.manager.password=admin
```

确保admin用户可以访问Manager应用程序，打开\$CATALINA_HOME/conf/tomcat-users.xml中是否存在下面一行。如果不存在，请自己添加。注意，roles属性可能是一个以逗号(",")隔开的系列。

```
<user username="admin" password="admin" roles="manager"/>
```

为了测试所有修改，保存所有文件，启动Tomcat。从命令行中进行MyUsers目录，运行ant list，可以看到Tomcat服务器上运行的应用程序。

图 3. 运行ant list命令的结果



```
Cygwin
$cd myusers
$ant list
Buildfile: build.xml

list:
  [list] OK - Listed applications for virtual host localhost
  [list] /admin:running:0:../server/webapps/admin
  [list] /balancer:running:0:balancer
  [list] /:running:0:C:\Tools\jakarta-tomcat-5.0.19\webapps\ROOT
  [list] /manager:running:0:../server/webapps/manager

BUILD SUCCESSFUL
Total time: 1 second
$
```

好了，现在运行ant deploy来安装MyUsers。打开浏览器，在地址栏中输入http://localhost:8080/myusers，出现如图2.4的“Equinox Welcome”画面。

图 4. Equinox欢迎页面



警告

为了使驻留在内存的HSQL能很好的结合MyUsers程序，从你运行Ant的同一目录中启动Tomcat。在UNIX/Linux下键入\$CATALINA_HOME/bin/startup.sh，Windows下运行%CATALINA_HOME%\bin\startup.bat。你也可以修改数据库设置使用绝对路径。

在接下来的几节中，你将会创建一个User对象和一个维护其持久性的Hibernate DAO对象。用Spring来管理DAO类及其依赖关系。最后，还会创建一个业务委派，来使用AOP和声明式事务处理。

为持久层编写单元测试

在MyUsers程序，使用Hibernate作为持久层。Hibernate是一个O/R映像框架，用来关联Java对象和数据库中的表(tables)。它使得对象的CRUD操作变得非常简单，Spring结合了Hibernate变得更加容易。从Hibernate转向Spring+Hibernate会减少75%的代码。这主要得益于，ServiceLocator和一些DAOFactory类的废弃，Spring的运行时异常代替了Hibernate的检测式异常(checkedException)。

写一个单元测试有助于规范UserDAO接口。为UserDAO写一个JUnit测试程序，需要完成以下步骤：

1. 在test/org/appfuse/dao下新建一个UserDAOTest类。它继承了同一个包中的BaseDAOTestCase，其父类初始化了Spring的ApplicationContext(来自web/WEBINF/ApplicationContext.xml)，以下是Spring测试的代码。

```
package org.appfuse.dao;  
// use your IDE to handle imports
```

```
public class UserDAOTest extends BaseDAOTestCase {
    private User user = null;
    private UserDAO dao = null;

    protected void setUp() throws Exception {
        super.setUp();
        dao = (UserDAO) ctx.getBean("userDAO");
    }

    protected void tearDown() throws Exception {
        super.tearDown();
        dao = null;
    }
}
```

这个类还无法通过编译，因为还没有创建UserDAO接口。在这之前，来写一些来验证User的CRUD操作。

2. 在UserDAOTest类中添加testSave和testAddAndRemove方法，如下所示：

```
public void testSaveUser() throws Exception {
    user = new User ();
    user.setFirstName("Rod");
    user.setLastName("Johnson");
    dao.saveUser(user);
    assertNotNull("primary key assigned", user.getId());
    log.info(user);
    assertNotNull(user.getFirstName());
}

public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Bill");
    user.setLastName("Joy");
    dao.saveUser(user);
    assertNotNull(user.getId());
    assertEquals(user.getFirstName(), "Bill");
    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }
    dao.removeUser(user.getId());
    assertNull(dao.getUser(user.getId()));
}
```

从这些方法中可以看到，你需要在UserDAO创建以下方法：

- saveUser(User)
- removeUser(Long)
- getUser(Long)

- `getUsers()` (返回数据库的所有用户)
3. 在`src/org/appfuse/dao`目录下建一个名为`UserDAO.java`的类，输入以下代码：
注意

如果你使用Eclipse, IntelliJ IDEA之类的IDE，左边会出现在一个灯泡，提示类不存在，可以即时创建。

```
package org.appfuse.dao;
// use your IDE to handle imports

public interface UserDAO extends DAO {
    public List getUsers();
    public User getUser(Long userId);
    public void saveUser(User user);
    public void removeUser(Long userId);
}
```

为了让`UserDAO.java`, `UserDAOTest.java`编译通过，还要建一个`User.java`类。

4. 在`src/org/appfuse/model`下建一个`User.java`文件，添加几个成员变量：`id`, `firstName`, `lastName`，如下所示。

```
package org.appfuse.model;

public class User extends BaseObject {
    private Long id;
    private String firstName;
    private String lastName;
    /*
    Generate your getters and setters using your favorite IDE:
    In Eclipse:
    Right-click -> Source -> Generate Getters and Setters
    */
}
```

注意，你继承了`BaseObject`类，它包含几个有用的方法：`toString()`, `equals()`, `hashCode()`，后两个是Hibernate必需的。

建好`User`对象后，用IDE打开`UserDAO`和`UserDAOTest`两个类，优化导入。

配置Hibernate和Spring

现在已经有了POJO(Plain Old Java Object),写一个映射文件Hibernate就可以维护其持久性。

1. 在`org/appfuse/model`中新建一个名为`User.hbm.xml`文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="org.appfuse.model.User" table="app_user">
  <id name="id" column="id" unsaved-value="0">
    <generator class="increment" />
  </id>
  <property name="firstName" column="first_name"
    not-null="true"/>
  <property name="lastName" column="last_name" not-null="true"/>
</class>
</hibernate-mapping>
```

2. 在web/WEB-INF/目录下的web/WEB-INF/ApplicationContext.xml文件中添加映射关系。打开文件，找到<property name="mappingResources">，修改成如下：

```
<property name="mappingResources">
  <list>
    <value>org/appfuse/model/User.hbm.xml</value>
  </list>
</property>
```

在ApplicationContext.xml文件中，你可以看到数据库是如何设置的，使用Spring时如何配置Hibernate的。Equinox预设置会使用一个名为db/appfuse的HSQL数据库。它将在你的Ant的db目录下创建，详细配置在“[How Spring Is Configured in Equinox](#)”一节中描述。

3. 运行ant deploy reload(Tomcat处于运行状态)，在Tomcat控制台的日志中可以看到数据表的创建过程。

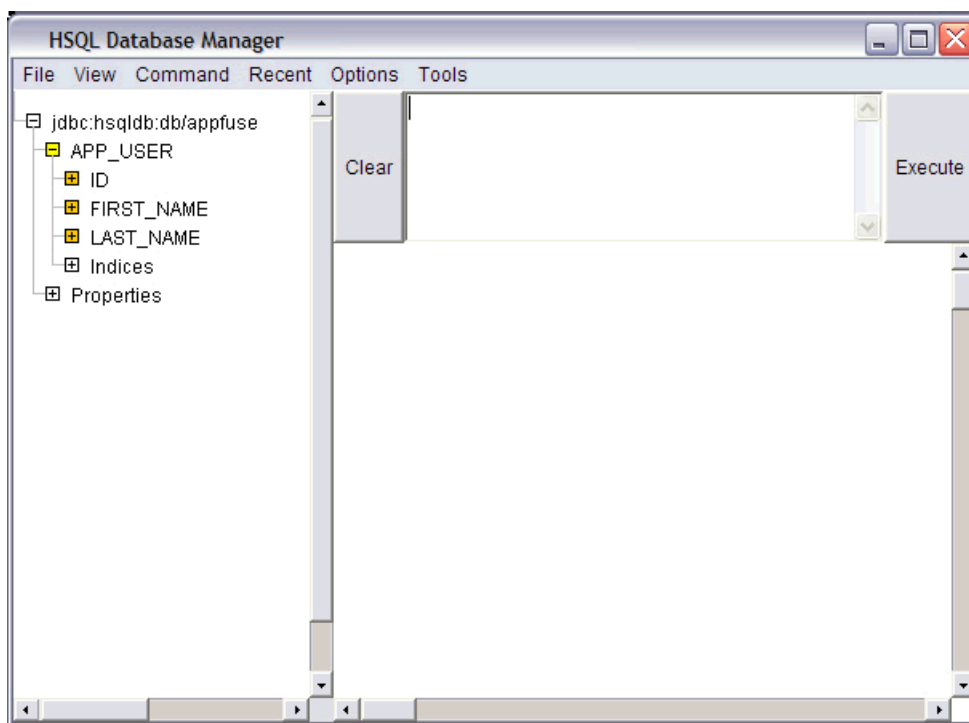
```
INFO - SchemaExport.execute(98) | Running hbm2ddl schema export
INFO - SchemaExport.execute(117) | exporting generated schema to database
INFO - ConnectionProviderFactory.newConnectionProvider(53) | Initializing
connection provider:
org.springframework.orm.hibernate.LocalDataSourceConnectionProvider
INFO - DriverManagerDataSource.getConnectionFromDriverManager(140) | Creating
new JDBC connection to [jdbc:hsql:db/appfuse]
INFO - SchemaExport.execute(160) | schema export complete
```

提示

如果你想看到更多或更少的日志，请修改web/WEB-INF/classes/log4j.xml中log4j的设置。

4. 为了验证数据库是否已经建好，运行 ant browser启动HSQL控制台。你会看到如下的HSQL Database Manager。

图 5. HSQL Database Manager



Equinox中Spring是如何配置的

任何基于J2EE的应用中使用Spring，配置都很简单。在最少情况下，你只要简单的添加Spring的ContextLoaderListener到你的web.xml中。

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

这是一个ServletContextListener，它会在启动web应用过程中进行初始化。默认情况下，它会查找web/WEB-INF/ApplicationContext.xml文件，你可以指定一个名为contextConfigLocation的<context-param>元素来进行更改，例如：

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/sampleContext.xml</param-value>
</context-param>
```

<param-value>元素可以是以空格或是逗号隔开的一系列路径。在Equinox中，Spring的配置使用了这个Listener和其默认的contextConfigLocation。

那么，Spring是如何找到Hibernate的？这就Spring的魅力所在，它让依赖性的绑定变得非常简单。请参阅ApplicationContext.xml的全部内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
            <value>org.hsqldb.jdbcDriver</value>
        </property>
        <property name="url">
            <value>jdbc:hsqldb:db/appfuse</value>
        </property>
        <property name="username"><value>sa</value></property>
        <!-- Make sure <value> tags are on same line - if they're not,
            authentication will fail -->
        <property name="password"><value></value></property>
    </bean>
    <!-- Hibernate SessionFactory -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource">
            <ref local="dataSource"/>
        </property>
        <property name="mappingResources">
            <list>
                <value>org/appfuse/model/User.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    net.sf.hibernate.dialect.HSQLDialect
                </prop>
                <prop key="hibernate.hbm2ddl.auto">create</prop>
            </props>
        </property>
    </bean>
    <!-- Transaction manager for a single Hibernate SessionFactory (alternative
        to JTA) -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref local="sessionFactory"/>
        </property>
    </bean>
</beans>
```

第一个bean(dataSource)代表HSQL数据库, 第2个bean(sessionFactory)依赖它。Spring仅仅是调用LocalSessionFactoryBean的setDataSource(DataSource)使之工作。如果你想用JNDI DataSource替换, 可以bean的定义改成类似下面的几行:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/appfuse</value>
  </property>
</bean>
```

同时请注意sessionFactory定义中的hibernate.hbm2ddl.auto属性。这个属性会在应用启动时自动创建数据表, 可选的值还可以是update或create-drop。

最后一个配置的bean是transactionManager(你也可以使用JTA transaction), 它在处理跨越两个数据库的分布式的事务处理中必不可少。如果你想使用jta transaction manager, 将此bean的class属性改成org.springframework.transaction.jta.JtaTransactionManager。

现在你可以用Hibernate实现UserDAO类。

用Hibernate实现UserDAO

要实现Hibernate UserDAO, 需要完成以下几步:

1. 在目录src/org/appfuse/dao/hibernate下创建一个文件UserDAOHibernate.java, 这个类继承了HibernateDaoSupport类, 并实现了UserDAO接口。

```
package org.appfuse.dao.hibernate;
// use your IDE to handle imports

public class UserDAOHibernate extends HibernateDaoSupport implements UserDAO {
  private Log log = LogFactory.getLog(UserDAOHibernate.class);

  public List getUsers() {
    return getHibernateTemplate().find("from User");
  }

  public User getUser(Long id) {
    return (User) getHibernateTemplate().get(User.class, id);
  }

  public void saveUser(User user) {
    getHibernateTemplate().saveOrUpdate(user);
    if (log.isDebugEnabled()) {
      log.debug("userId set to: " + user.getId());
    }
  }

  public void removeUser(Long id) {
```

```
Object user = getHibernateTemplate().load(User.class, id);
getHibernateTemplate().delete(user);
}
}
```

Spring的HibernateDaoSupport类是一个方便的实现Hibernate DAO接口的超类，你可以利用其一些有用的方法，来获得Hibernate DAO或是SessionFactory。最方便的方法是getHibernateTemplate()，它返回一个HibernateTemplate对象。这个模板把检测式异常(checkedException)包装成运行时异常(runtime exception)，这使得你的DAO接口无需抛出Hibernate异常。

程序还没有把UserDAO绑定到UserDAOHibernate上，必须创建它们之间的关联。

2. 在Spring配置文件(web/WEB-INF/ApplicationContext.xml)中添加以下内容：

```
<bean id="userDAO" class="org.appfuse.dao.hibernate.UserDAOHibernate">
  <property name="sessionFactory">
    <ref local="sessionFactory"/>
  </property>
</bean>
```

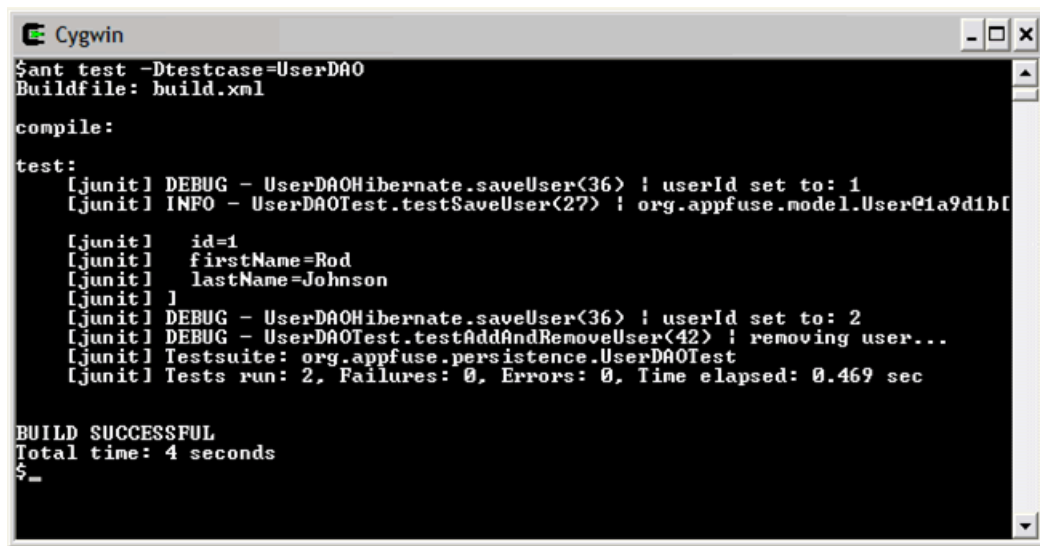
这样就在你的UserDAOHibernate(从HibernateDaoSupport的setSessionFactory继承)中建了一个Hibernate Session Factory。Spring会检测一个Session(也就是，它在web层是开放的)是否已经存在，并且直接使用它，而不是新建一个。这样你可以使用Spring流行的“Open Session in View”模式来延迟载入collection。

进行单元测试，用DAO验证CRUD操作

在进行第一个测试之前，把你的日志级别从“INFO”调到“WARN”。

1. 把log4j.xml(在web/WEB-INF/classes目录下)中<level value="INFO"/>改为<level value="WARN"/>。
2. 键入ant test来运行UserDAOTest。如果你有多个测试，你必须用ant test -Dtestcase=UserDAOTest来指定要运行的测试。运行之后，控制台中会出现一些测试的日志信息，如下所示。

图 6. 运行ant test -Dtestcase=UserDAO命令的结果



```

Cygwin
$ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserDAOHibernate.saveUser(36) : userId set to: 1
[junit] INFO - UserDAOTest.testSaveUser(27) : org.appfuse.model.User@1a9d1b1
[junit] id=1
[junit] firstName=Rod
[junit] lastName=Johnson
[junit] 1
[junit] DEBUG - UserDAOHibernate.saveUser(36) : userId set to: 2
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(42) : removing user...
[junit] Test suite: org.appfuse.persistence.UserDAOTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.469 sec

BUILD SUCCESSFUL
Total time: 4 seconds
$

```

创建Manager，声明事务处理

J2EE开发中强烈建议将各层进行分离。换言之，不要把数据层(DAO)和web层(servlets)混在一起。使用Spring很容易做到这一点，但使用“业务委派”(business delegate)模式，可以对这些层进一步分离。

使用业务委派模式的主要原因是：

- 大多数持久层组件执行一个业务逻辑单元，把逻辑放在一非web类中的最大好处是，web service或是胖客户端(rich platform client)可以像使用servlet一样来用同一API。
- 大多数业务逻辑都在同一方法中完成，当然可能多个DAO。使用业务委派，使得你可以在一个更高的业务委派层(level)使用Spring的声明式业务委派特性。

MyUsers应用中UserManager和UserDAO拥有相同的一个方法。主要不同的是Manager对于web更为友好(web-friendly)，它可以接受String，而UserDAO只能接受Long，并且它可以在saveUser方法中返回一个User对象。这在插入一个新用户比如，要获得主键，是非常方便的。Manager(或称为业务委派)中也可以添加一些应用中所需要的其它业务逻辑。

1. 首先在test/org/appfuse/service(你必须先建好这个目录)中新建一个UserManagerTest类，开始创建“service”，这个类继承了JUnit的TestCase类，代码如下：

```

package org.appfuse.service;
// use your IDE to handle imports

public class UserManagerTest extends TestCase {
    private static Log log = LogFactory.getLog(UserManagerTest.class);
    private ApplicationContext ctx;
    private User user;
    private UserManager mgr;

    protected void setUp() throws Exception {

```

```
String[] paths = {"/WEB-INF/applicationContext.xml"};
ctx = new ClassPathXmlApplicationContext(paths);
mgr = (UserManager) ctx.getBean("userManager");
}

protected void tearDown() throws Exception {
    user = null;
    mgr = null;
}
// add testXXX methods here
}
```

在`setUp()`方法中，使用`ClassPathXmlApplicationContext`把`ApplicationContext.xml`载入变量`ApplicationContext`中。载入`ApplicationContext`有几种途径，从`classpath`中，文件系统，或`web`应用内。这些方法将在第三章(`BeanFactory`及其运行原理)中描述。

2. 编写第一个测试方法，来验证使用`UserManager`成功的完成添加和删除一个`User`对象操作。

```
public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");
    user = mgr.saveUser(user);
    assertNotNull(user.getId());
    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }
    String userId = user.getId().toString();
    mgr.removeUser(userId);
    user = mgr.getUser(userId);
    assertNull("User object found in database", user);
}
```

这个测试实际上是一个集成测试(integration test)，而不是单元测试(unit test)。为了更接近单元测试，可以使用`EasyMock`或是类似工具来“伪装”(fake) DAO。这样，就不必关心`ApplicationContext`和任何依赖Spring API的东西。建议创建这样的测试，因为它可以测试项目所有依赖(Hibernate, Spring自己的类)的内部构件，包括数据库。第9章，讨论重构`UserManagerTest`，使用mock处理DAO的依赖性。

3. 为了编译`UserManagerTest`，在`src/org/appfuse/service`中新建一个接口——`UserManager`。在`org.appfuse.service`包中创建这个类，代码如下：

```
package org.appfuse.service;
// use your IDE to handle imports

public interface UserManager {
    public List getUsers();
    public User getUser(String userId);
}
```



```
public User saveUser(User user);  
public void removeUser(String userId);  
}
```

4. 建一个名为org.appfuse.service.impl的子包, 新建一个类,实现userManager接口。

```
package org.appfuse.service.impl;  
// use your IDE to handle imports  
  
public class UserManagerImpl implements UserManager {  
    private static Log log = LogFactory.getLog(UserManagerImpl.class);  
    private UserDao dao;  
  
    public void setUserDAO(UserDao dao) {  
        this.dao = dao;  
    }  
  
    public List getUsers() {  
        return dao.getUsers();  
    }  
  
    public User getUser(String userId) {  
        User user = dao.getUser(Long.valueOf(userId));  
        if (user == null) {  
            log.warn("UserId '" + userId + "' not found in database.");  
        }  
        return user;  
    }  
  
    public User saveUser(User user) {  
        dao.saveUser(user);  
        return user;  
    }  
  
    public void removeUser(String userId) {  
        dao.removeUser(Long.valueOf(userId));  
    }  
}
```

这个类看不出你在使用Hibernate。当你打算把持久层转向一种不同的技术时, 这样做很重要。

这个类提供一个私有dao成员变量, 还有setUserDAO()方法。这样能够让Spring能够表演“依赖性绑定”魔术(perform “dependency binding” magic), 把这些对象扎在一起。在使用mock重构这个类时, 你必须在userManager接口中添加setUserDAO()方法。

5. 在进行测试之前, 配置Spring, 以便使用getBean("userManager")返回一个UserManagerImpl类。在web/WEB-INF/ApplicationContext.xml文件中, 添加以下几行代码:

```
<bean id="userManager"
```

```

class="org.appfuse.service.UserManagerImpl">
<property name="userDAO"><ref local="userDAO"/></property>
</bean>

```

余下的问题是，你还没有使用Spring的AOP，特别是让声明式的事务处理发挥作用。

6. 为了实现这一点，使用ProxyFactoryBean代替userManager。ProxyFactoryBean可以创建一个类的不同的实现，这样AOP能够解释和覆盖方法调用。在事务处理中，使用TransactionProxyFactoryBean代替userManagerImpl 类。在context文件中添加下面bean的定义：

```

<bean id="userManager"
class="org.springframework.transaction.interceptor.TransactionProxy
FactoryBean">
<property name="transactionManager">
<ref local="transactionManager"/>
</property>
<property name="target">
<ref local="userManagerTarget"/>
</property>
<property name="transactionAttributes">
<props>
<prop key="save*">PROPAGATION_REQUIRED</prop>
<prop key="remove*">PROPAGATION_REQUIRED</prop>
<prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
</props>
</property>
</bean>

```

从这个xml代码片断中可以看出，TransactionProxyFactoryBean必须设置一个transactionManager属性，定义transactionAttributes。

7. 让事务处理代理(Transaction Proxy)知道你要模仿的对象：userManagerTarget。作为新bean的一部分，修改原来的userManager bean，使之拥有一个值为userManagerTarget的id属性。

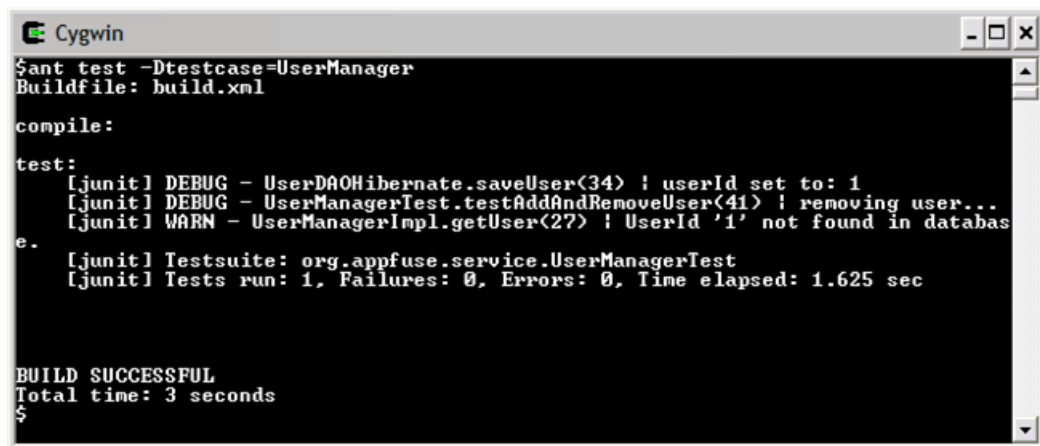
```

<bean id="userManagerTarget"
class="org.appfuse.service.UserManagerImpl">
<property name="userDAO"><ref local="userDAO"/></property>
</bean>

```

编辑applicationContext.xml添加userManager和userManagerTarget的定义后，运行ant test -Dtestcase=UserManager，看看终端输出的结果：

图 7. 运行ant test -Dtestcase=UserManager命令的结果



```

Cygwin
$ant test -Dtestcase=UserManager
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserDaoHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(41) : removing user...
[junit] WARN - UserManagerImpl.getUser(27) : UserId '1' not found in database.
e.
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.625 sec

BUILD SUCCESSFUL
Total time: 3 seconds
$

```

8. 如果你想看看事务处理的执行和提交情况，在log4j.xml中添加：

```

<logger name="org.springframework.transaction">
  <level value="DEBUG"/> <!-- INFO does nothing -->
</logger>

```

重新运行测试，将看到大量日志信息，如它相关的对象，事务的创建和提交等。测试完毕，最好删除上面的日志定义(logger)。

祝贺你！你已经实现了一个web应用的Spring/Hibernate后端解决方案。并且你已经用AOP和声明式业务处理配置好了业务委派。了不起，自我鼓励一下！（This is no small feat; give yourself a pat on the back!）

对Struts Action进行单元测试

现在业务委派和DAO都在起作用，我们看看MVC框架吸盘(sucker)的上部。为了管理用户，创建一个Struts Action，继续进行测试驱动(Test-Driven)开发。

Equinox是为Struts配置的。配置Struts需要在web.xml中进行一些设置，并在web/WEB-INF下定义一个struts-config.xml文件。由于Struts开发人员比较多，这里先使用Struts。第4章用Spring进行处理。如果你想跳过这一节，直接学习Spring MVC方法，请参考第4章：Spring MVC框架。

在test/org/appfuse/web目录下新建一个文件UserActionTest.java，开发你的第一个Struts Action单元测试。这个类继承了MockStrutsTestCase，文件内容如下：

```

package org.appfuse.web;
// use your IDE to handle imports

public class UserActionTest extends MockStrutsTestCase {
  public UserActionTest(String testName) {
    super(testName);
  }
}

```

```
public void testExecute() {
    setRequestPathInfo("/user");
    addRequestParameter("id", "1");
    actionPerform();
    verifyForward("success");
    verifyNoActionErrors();
}
}
```

为web层创建Action和Model(DynaActionForm)

1. 在目录src/org/appfuse/web下新建一个类UserAction.java。这个类继承了DispatchAction，你可以花几分钟，在这个类中，创建CRUD方法。

```
package org.appfuse.web;
// use your IDE to handle imports

public class UserAction extends DispatchAction {
    private static Log log = LogFactory.getLog(UserAction.class);

    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        request.getSession().setAttribute("test", "succeeded!");
        log.debug("looking up userId: " + request.getParameter("id"));
        return mapping.findForward("success");
    }
}
```

2. 配置Struts，使“/user”这个请求路径有意义。在web/WEB-INF/struts-config.xml中加入一个action-mapping。打开文件加入：

```
<action path="/user" type="org.appfuse.web.UserAction">
    <forward name="success" path="/index.jsp"/>
</action>
```

3. 执行命令ant test -Dtestcase=UserAction，你会看到友好的“BUILD SUCCESSFULLY”信息。
4. 在struts-config.xml中添加form-bean定义(在form-beans部分)。对于Struts ActionForm，使用DynaActionForm，这是一个javabean，可以从XML定义中动态的创建。

```
<form-bean name="userForm"
    type="org.apache.struts.action.DynaActionForm">
```

```
<form-property name="user" type="org.appfuse.model.User"/>
</form-bean>
```

这里使用这种表示方式，而没有使用了具体的ActionForm，因为你只需要一个User对象的瘦(thin)包装器。理想情况下，你可以User对象，但会失去Struts环境下的一些特性：验证属性(validate properties)，checkbox复位(reset checkboxes)。后面，将演示用Spring为何会更加简单，它可以让你在web层使用User对象。

5. 修改<action>定义，在request中使用这个form。

```
<action path="/user" type="org.appfuse.web.UserAction"
name="userForm" scope="request">
  <forward name="success" path="/index.jsp"/>
</action>
```

6. 修改UserActionTest类，在Action中测试不同的CRUD方法。如下所示：

```
public class UserActionTest extends MockStrutsTestCase {

    public UserActionTest(String testName) {
        super(testName);
    }

    // Adding a new user is required between tests because HSQL creates
    // an in-memory database that goes away during tests.
    public void addUser() {
        setRequestPathInfo("/user");
        addRequestParameter("method", "save");
        addRequestParameter("user.firstName", "Juergen");
        addRequestParameter("user.lastName", "Hoeller");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
    }

    public void testAddAndEdit() {
        addUser();
        // edit newly added user
        addRequestParameter("method", "edit");
        addRequestParameter("id", "1");
        actionPerform();
        verifyForward("edit");
        verifyNoActionErrors();
    }

    public void testAddAndDelete() {
        addUser();
        // delete new user
        setRequestPathInfo("/user");
    }
}
```

```
        addRequestParameter("method", "delete");
        addRequestParameter("user.id", "1");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
    }

    public void testList() {
        addUser();
        setRequestPathInfo("/user");
        addRequestParameter("method", "list");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
        List users = (List) getRequest().getAttribute("users");
        assertNotNull(users);
        assertTrue(users.size() == 1);
    }
}
```

7. 修改UserAction, 这样测试程序才能通过, 并能处理(客户端)请求。最简单的方法是添加edit, save和delete方法, 请确保你事先已经删除了execute方法。下面是修改过的UserAction.java文件。

```
public class UserAction extends DispatchAction {
    private static Log log = LogFactory.getLog(UserAction.class);
    private UserManager mgr = null;

    public void setUserManager(UserManager userManager) {
        this.mgr = userManager;
    }

    public ActionForward delete(ActionMapping mapping, ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        mgr.removeUser(request.getParameter("user.id"));
        ActionMessages messages = new ActionMessages();
        messages.add(ActionMessages.GLOBAL_MESSAGE,
            new ActionMessage("user.deleted"));
        saveMessages(request, messages);
        return list(mapping, form, request, response);
    }

    public ActionForward edit(ActionMapping mapping, ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
```

```
        if (log.isDebugEnabled()) {
            log.debug("entering 'edit' method...");
        }
        DynaActionForm userForm = (DynaActionForm) form;
        String userId = request.getParameter("id");
        // null userId indicates an add
        if (userId != null) {
            User user = mgr.getUser(userId);
            if (user == null) {
                ActionMessages errors = new ActionMessages();
                errors.add(ActionMessages.GLOBAL_MESSAGE,
                    new ActionMessage("user.missing"));
                saveErrors(request, errors);
                return mapping.findForward("list");
            }
            userForm.set("user", user);
        }
        return mapping.findForward("edit");
    }

    public ActionForward list(ActionMapping mapping, ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'list' method...");
        }
        request.setAttribute("users", mgr.getUsers());
        return mapping.findForward("list");
    }

    public ActionForward save(ActionMapping mapping, ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'save' method...");
        }
        DynaActionForm userForm = (DynaActionForm) form;
        mgr.saveUser((User)userForm.get("user"));
        ActionMessages messages = new ActionMessages();
        messages.add(ActionMessages.GLOBAL_MESSAGE,
            new ActionMessage("user.saved"));
        saveMessages(request, messages);
        return list(mapping, form, request, response);
    }
}
```

现在你已经修改了这个类的CRUD操作，继续以下步骤：

8. 修改struts-config.xml，使用ContextLoaderPlugin来配置Spring的UserManager设置。把下面内容添加到你的struts-config.xml中。

```
<plug-in
  className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
    value="/WEB-INF/applicationContext.xml,
    /WEB-INF/action-servlet.xml"/>
</plug-in>
```

默认情况下这个插件会载入action-servlet.xml文件。要让Test Action能找到你的Manager，你还必须配置这个插件来载入ApplicationContext.xml文件。

注意

使用ContextLoaderPlugin是众多的Struts web层与Spring中间层集成的方法之一。其它的各种选择将在第11章：web框架集成中一一讲解。

9. 对每个使用Spring的Action，定义一个type="org.springframework.web.struts.DelegatingActionProxy"的action-mapping，为每个真实的Struts Action声明一个对应的Spring bean。这样修改一下你的action mapping 就能使用这个新类。
10. 修改Action mapping，使用DispatchAction。

为了让DispatchAction运行，在mapping中添加参数parameter="method"，它表示(在一个URL或是隐藏字段hidden field)要调用的方法，同时转向(forwards)edit和list forward(参考能进行CRUD操作的UserAction类)。

```
<action path="/user"
  type="org.springframework.web.struts.DelegatingActionProxy"
  name="userForm" scope="request" parameter="method">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

确保web目录下已经建好userList.jsp和userForm.jsp两个文件。暂时不必在文件中写入内容。

11. 作为插件的一部分，配置Spring，以便能识别“ /user ” bean并把UserManager设置成它的属性。在web/WEB-INF/action-servlet.xml中添加以下定义。

```
<bean name="/user" class="org.appfuse.web.UserAction"
  singleton="false">
  <property name="userManager">
    <ref bean="userManager"/>
  </property>
</bean>
```

定义中，使用singleton="false"。这样就会为每个请求，新建一个Action，减少对线程安全Action的需求。既然你的Manager和DAO都没有成员变量，没有设置这一属性也不会出问题(默认singleton="true")。

12. 在message.properties资源绑定文件中配置信息(message)。

在UserAction类中，有一些对在完成操作时显示成功或错误信息引用。这些引用是指一个应用资源绑定文件(或messages.properties文件中)中各信息的键。这里是指：

- user.saved
- user.missing
- user.deleted

把这些键存入web/WEB-INF/classes下的messages.properties文件中。例如：

```
user.saved=User has been saved successfully.
user.missing=No user found with this id.
user.deleted=User successfully deleted.
```

这个文件通过struts-config.xml中的<message-resources>元素进行加载。

```
<message-resources parameter="messages" />
```

运行单元测试，验证Action的CRUD操作

运行 `ant test -Dtestcase=UserAction`。输出结果如下：

图 8. 运行ant test -Dtestcase=UserAction命令的输出结果

```
Cygwin
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.edit(48) ! entering 'edit' method...
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.delete(30) ! entering 'delete' method...
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] Testsuite: org.appfuse.web.UserActionTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 3.563 sec

BUILD SUCCESSFUL
Total time: 6 seconds
$
```

填充JSP文件，这样可以通过浏览器来进行CRUD操作

1. 在你的jsp文件(userFrom.jsp和userList.jsp)中添加以下代码，这样它们可以显示action处理的结果。如果还事先准备，在web目录下建一个文件userList.jsp。添加一些代码你就可以看到数据库中所有的用户资料。在下面代码中，第一行包含(include)了一个文件taglibs.jsp。这个文件包含了应用所有JSP Tag Library的声明。大部分是Struts Tag, JSTL和SiteMesh(用来美化JSP页面)。

```
<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User List</title>
<button onclick="location.href='user.do?method=edit'">Add User</button>
<table class="list">
<thead>
<tr>
<th>User Id</th>
<th>First Name</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<c:forEach var="user" items="${users}" varStatus="status">
<c:choose>
<c:when test="${status.count % 2 == 0}"><tr class="even"></c:when>
<c:otherwise><tr class="odd"></c:otherwise>
</c:choose>
<td><a href="user.do?method=edit&id=${user.id}">${user.id}</a></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
</tr>
</c:forEach>
</tbody>
</table>
```

你可以看到有一行“标题头”(headings)(在<thead>中)。JSTL的<c:forEach>进行结果迭代，显示所有的用户。

2. 向数据库添加一些数据，你就会看到一些真实(actual)的用户(users)。你可以选择一种方法，手工添加，使用ant browse，或是在build.xml中添加如下的target:

```
<target name="populate">
<echo message="Loading sample data..." />
<sql driver="org.hsqldb.jdbcDriver"
url="jdbc:hsqldb:db/appfuse"
userid="sa" password="">
<classpath refid="classpath"/>
INSERT INTO app_user (id, first_name, last_name)
values (5, 'Julie', 'Raible');
INSERT INTO app_user (id, first_name, last_name)
```

```

        values (6, 'Abbie', 'Raible');
    </sql>
</target>

```

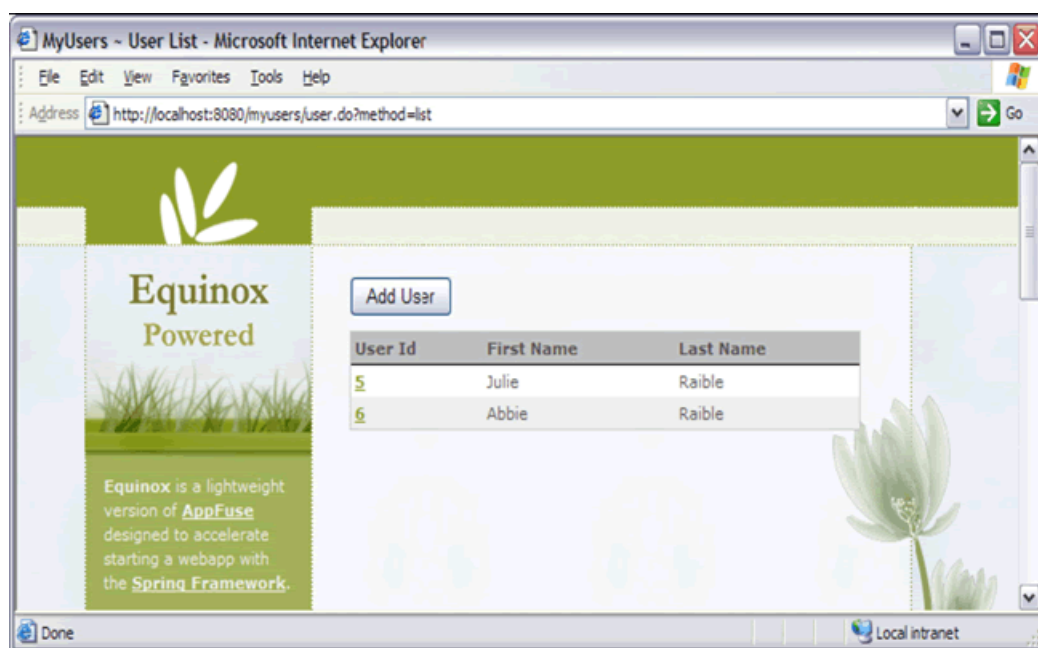
警告

为了使内置的HSQL正常工作，从能运行Ant的目录下启动Tomcat。在UNIX/Linux键入\$CATALINA_HOME/bin/startup.sh，在win上 %CATALINA_HOME%\bin\startup.bat。

通过浏览器验证JSP的功能

1. 有了这个JSP文件和里面的样例数据，就可以通过浏览器来查看这个页面。运行ant deploy reload，转到地址http://localhost:8080/myusers/user.do?method=list。出现以下画面。

图 9. 运行ant deploy reload命令的结果



2. 这个样例中，缺少国际化的页面标题头，和列标题头(column headings)。在 web/WEBINF/classes/messages.properties 中加入一些键。

```

user.id=User Id
user.firstName=First Name
user.lastName=Last Name

```

修改过的国际化的标题头如下：

```
<thead>
```

```

<tr>
<th><bean:message key="user.id" /></th>
<th><bean:message key="user.firstName"/></th>
<th><bean:message key="user.lastName"/></th>
</tr>
</thead>

```

注意同样可以使用JSTL的<fmt:message key="...">标签。如果想为表添加排序和分布功能，可以使用Display Tag(<http://displaytag.sf.net>)。下面是使用这个标签的一个样例：

```

<display:table name="users" pagesize="10" styleClass="list"
requestURI="user.do?method=list">
<display:column property="id" paramId="id" paramProperty="id"
href="user.do?method=edit" sort="true"/>
<display:column property="firstName" sort="true"/>
<display:column property="lastName" sort="true"/>
</display:table>

```

请参考display tag文档中有关的列标题头国际化的部分。

3. 你已经建好了显示(list),创建form就可以添加/编辑(add/edit)数据。如果事先没有准备，可以在web目录下新建一个userForm.jsp文件。向文件中添加以下代码：

```

<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User Details</title>
<p>Please fill in user's information below:</p>
<html:form action="/user" focus="user.firstName">
<input type="hidden" name="method" value="save"/>
<html:hidden property="user.id"/>
<table>
<tr>
<th><bean:message key="user.firstName"/>: </th>
<td><html:text property="user.firstName"/></td>
</tr>
<tr>
<th><bean:message key="user.lastName"/>: </th>
<td><html:text property="user.lastName"/></td>
</tr>
<tr>
<td></td>
<td>
<html:submit styleClass="button">Save</html:submit>
<c:if test="${not empty param.id}">
<html:submit styleClass="button"
onclick="this.form.method.value='delete'">
Delete</html:submit>
</c:if>
</td>
</tr>

```

```
</table>
</html:form>
```

注意

如果你正在开发一个国际化的应用，把上面的信息和按钮标签替换成`<bean:message>`或是`<fmt:message>`标签。这是一个很好的练习。对于信息message，建议把key名称写成`pageName.message`(例如：`userForm.message`)的形式，按钮名字写成“button.name”(例如`button.save`)。

4. 运行`ant deploy`，通过浏览器页面的user form来进行CRUD操作。

最后，大部分web应用都需要验证。下一节中，配置Struts Validator，要求的last name 是必填的。

用Commons Validator添加验证

为了在Struts中使用验证，执行以下几步：

1. 在`struts-config.xml`中添加`ValidatorPlugin`。
2. 创建`validation.xml`，指定`lastName`为必填字段。
3. 把`DynaActionForm`改用`DynaValidatorForm`。
4. 仅为`save()`方法设置验证(validation)。
5. 在`message.properties`中添加validation errors。

在struts-config.xml中添加ValidatorPlugin

配置Validator plugins,添加以下片断到`struts-config.xml`(紧接着Spring plugin):

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
    value="/WEB-INF/validator-rules.xml,
    /WEB-INF/validation.xml"/>
</plug-in>
```

从这里你可以看出，Validator会查找WEB-INF目录下的两个文件`validator-rules.xml`和`validation.xml`。第一个文件，`validator-rules.xml`，是一个标准文件，作为Struts产品的一部分发布，它定义了所有可用的验证器(validators)，功能和客户端的JavaScript类似。第二个文件，包含针对每个form的验证规则。

创建validation.xml，指定lastName为必填字段

`validation.xml`文件中包含很多DTD定义的标准元素。但你只需要如下所示的`<form>`和`<field>`，更多信息请参阅Validator的文档。在`web/WEB-INF/validation.xml`中的`<form-validation>`标签之间添加`<formset>`元素。

```
<formset>
  <form name="userForm">
    <field property="user.lastName" depends="required">
      <arg0 key="user.lastName"/>
    </field>
  </form>
</formset>
```

把 DynaActionForm 改为 DynaValidatorForm

把struts-config.xml中的DynaActionForm改为DynaValidatorForm。

```
<form-bean name="userForm"
type="org.apache.struts.validator.DynaValidatorForm">
...

```

为save()方法设置验证(validation)

使用Struts的DispatchAction弊端是，验证会在映射层(mapping level)激活。为了在list和edit页面关闭验证。你必须单独建一个“validate=false”的映射。例如，AppFuse的UserAction有两个映射：“/editUser”和“/listUser”。然而有一个更简单的方法，可以减少xml，只是多了一些java代码。

1. 在/user映射中，添加validate="false"。
2. 修改UserAction中的save()方法，调用form.validate()方法，如果发现错误，返回编辑页面。

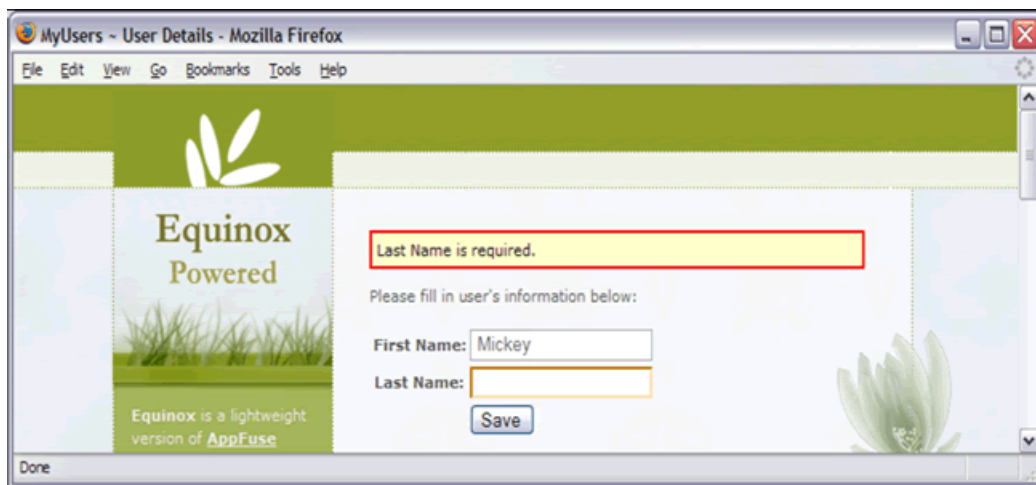
```
if (log.isDebugEnabled()) {
    log.debug("entering 'save' method...");
}
// run validation rules on this form
ActionMessages errors = form.validate(mapping, request);
if (!errors.isEmpty()) {
    saveErrors(request, errors);
    return mapping.findForward("edit");
}
DynaActionForm userForm = (DynaActionForm) form;
```

当DispatchAction运行时，与附带一个属性的两个映射相比，这样更加简洁。但用两个映射也有一些优点：

- 验证失败时，可以指定转向“input”属性。
- 在映射中可以添加“role”属性，可以指定谁有访问权限。例如，任何人都可以看到编辑(edit)页面，但只有管理员可以保存(save)。

3. 运行`ant deploy`重新载入(reload), 尝试添加一个新用户, 不要填写`lastName`。你会看到一个验证错误, 表明`lastName`是必填字段, 如下所示:

图 10. 运行`ant deploy`命令的结果



Struts Validator的另一种比较好的特性是客户端验证(client-side validation)。

4. 在form标签(`web/userForm.jsp`中)中添加`onsubmit` "属性, 在form末尾添加`<html:javascript>`标签。

```
<html:form action="/user" focus="user.firstName"
onsubmit="return validateUserForm(this)">
...
</html:form>
<html:javascript formName="userForm" />
```

现在如果运行`ant deploy`, 试图保存一个`lastName`为空的用户, 会弹出一个JavaScript提示: "Last Name is required"。这里有一个问题, 这个带JavaScript的form把validator的JavaScript功能都载入了页面。更好的方法是, 从外部文件导入JavaScript。参见第5章。

恭喜你! 你已经开发一个web应用程序, 它包含数据库交互, 验证实现, 成功信息和错误信息的显示。第4章, 将会把这个应用转向使用Spring框架。第5章中, 会添加异常处理, 文件上传, 邮件发送等特性。第6章会看一下JSP的替代技术, 在第7章, 会探讨DAO的不同实现, 包括iBATIS, JDO和Spring的JDBC。