

在 JavaEE 企业级开发中，以 SSH2 框架为核心的应用非常广，大象根据项目实践经验，通过一个实例，详细的为大家讲解如何实现全注解式的开发。

开发环境

JDK1.6.0_18

Eclipse3.2.1

MyEclipse5.1.0

Tomcat6.0.10

MySQL5.0.27


Navicat Lite for MySQL 8.1.20

每个人的开发环境可能会有差异，但有一点我需要说明的是，JDK 的版本不得低于 1.5，因为用到了很多 1.5 版才支持的新特性。Tomcat 和 MySQL 请不要低于我所用的版本，因为我在其它的版本上进行测试。Navicat 则是 MySQL 数据库的图形化操作工具。我在这里假定各位目前已经设置好了开发环境，下面就开始详细的说明。

由于要阐述的内容比较多，大象决定将它们划分成个几章节来讲，这一章就主要来说说 jar 包的选择。

第一部分：选择必须的 jar 包

新建一个 web 项目，然后将必要的 jar 包 COPY 到 lib 里面。根据本文实例 demo，大象给出下图中的最少 jar 包配置。



antlr-2.7.6.jar
backport-util-concurrent.jar
commons-collections-3.2.jar
commons-dbc-1.2.2.jar
commons-fileupload-1.2.1.jar
commons-logging-1.1.1.jar
commons-pool-1.3.jar
dom4j-1.6.1.jar
ejb3-persistence-1.0.2.jar
freemarker-2.3.13.jar
hibernate-annotations-3.4.0.jar
hibernate-commons-annotations-3.1.0.jar
hibernate-core-3.3.1.jar
javassist-3.4.GA.jar
jta-1.1.jar
log4j-1.2.15.jar
mysql-connector-java-5.1.6-bin.jar
ognl-2.6.11.jar
slf4j-api-1.5.0.jar
slf4j-log4j12-1.5.0.jar
spring-2.5.6.jar
struts2-convention-plugin-2.1.6.jar
struts2-core-2.1.6.jar
struts2-spring-plugin-2.1.6.jar
xwork-2.1.2.jar
www.blogjava.net/bolo

我对这些 jar 包进行一下说明，方便大家理解。

Struts2

```
commons-fileupload-1.2.1.jar
commons-logging-1.1.1.jar
freemarker-2.3.13.jar
ognl-2.6.11.jar
struts2-convention-plugin-2.1.6.jar
struts2-core-2.1.6.jar
struts2-spring-plugin-2.1.6.jar
xwork-2.1.2.jar
```

解压 Struts2.1.6 的 lib 文件夹，从中选出上面 7 个 jar 包添加到我们的工程库中。commons-logging、freemarker、ognl、struts2-core、xwork 这 5 个还是 struts2 的核心包。但在 Struts2.1.6 这个版本中，还需要加上 commons-fileupload 包。如果没有，则启动就会报错，不过不需要像网上传言的那样还得加上 commons-io 的 jar 包，这些大象都亲自做过测试。在本实例中，我将对 struts2 也采取注解的方式，所以用到了 struts2-convention-plugin-2.1.6.jar 这个插件。因为要与 spring 整合，所以 struts2-spring-plugin-2.1.6.jar 也必不可少。

Spring

```
spring-2.5.6.jar
ackport-util-concurrent.jar
log4j-1.2.15.jar
slf4j-log4j12-1.5.0.jar
```

大象在这里偷个懒，直接将 spring 的完整 jar 包加了进来，如果各位想精简类库的话，就选取它的分类 jar 包吧。比如本例使用 struts2 作为 MVC 框架，所以 spring 的 webmvc 就不可能用到了。有想改的朋友请自己动手改下。另外有点我想说下，如果采取完整 spring 的 jar 包，还需要 Spring2.5.6\lib\concurrent 文件夹中的 backport-util-concurrent.jar，如果不加这个，spring 会报错。但是采取 spring 分类 jar 包的形式，这个可以不用加，至于具体使用什么需要依赖这个包，大象还没去测试过，这个有待验证。还有 lib\slf4j 下的日志包，目前很多都开始采用基于 slf4j 接口的日志器，它的好处就是日志器是根据 slf4j 的接口来进行实现，可以在不改变代码的情况下更换日志器。最后 Spring 的源代码中使用的是 commons-logging 记录日志，因此这个包不能少，不过因为 struts2 也用到了，所以这里就省了。

Hibernate

```
hibernate-core-3.3.1.jar
antlr-2.7.6.jar
commons-collections-3.1.jar
dom4j-1.6.1.jar
javassist-3.4.GA.jar
jta-1.1.jar
slf4j-api-1.5.0.jar
```

Hibernate 从 3.3 版开始，对 jar 包结构做了一次大的调整，我们只需要加入 lib\required 文件夹下面的 6 个 jar 包。请注意这 6 个 jar 包都是使用 Hibernate 所必须的。另外再加上 hibernate 核心包。这里我将 slf4j-api-1.5.2.jar 换成了 1.5.0，这是因为 slf4j 是一个通用日志接口，不提供任何实现，我在 demo 里面使用的是 log4j，而 hibernate 包里面没有 log4j 的 slf4j 实现。而且如果版本不一致，会有异常，因此我就采用 Spring2.5.6\lib\slf4j 里面提供的配套版本。另外我将 commons-collections-3.1.jar 换成了 Struts2.1.6 里面的 3.2 版。

Annotations

ejb3-persistence-1.0.2.jar
hibernate-annotations-3.4.0.jar
hibernate-commons-annotations-3.1.0.jar

例子中使用 Hibernate JPA 来完成实体对象映射，所以上面这些包都必不可少。使用注解的方式，可以不用写繁琐的配置文件，降低了出错机率。而且现在很多人都喜欢这种方式。大家可以去 sourceforge 下载。

下载地址：<http://sourceforge.net/projects/hibernate/files/>

DBCP

commons-dbcp-1.2.2.jar
commons-pool-1.3.jar

本例使用 DBCP 连接池来管理数据源。

MySQL Driver

mysql-connector-java-5.1.6-bin.jar

MySQL 数据库的连接驱动。

可选的jar包

cglib-nodep-2.1_3.jar

这个包的作用是创建动态代理对象。比如在使用 AOP 方式管理 spring 事务时，如果我们的目标对象没有实现接口，而又要使用 AOP 来处理事务，这时就需要用到这个 jar 包。可以在 Spring2.5.6\lib\cglib 里面找到。

jstl-1.1.2.jar

standard-1.1.2.jar

JSTL 标签库，很经典的东东，如果需要可以将它们加入 lib 中。

大象在这里建议大家做开发的时候，不要过多的依赖 MyEclipse 提供的那些功能，多用手动的方式来做。那样方便是方便了，但不利于学习。比如加入上面这些开发所用的类库，这样可以更清楚的了解每个 jar 包的作用，增加知识的积累，方便以后调试。

在上一章中详细分析了 JAR 包的选择，那么这次我将对例子中的一些必须的配置文件进行下说明。虽然这些配置在网上也很容易找到，但是很多都没有讲个因为所以出来，这样根本就得不到提高。在此，大象为各位详细分析一下这些内容。

实例中涉及的配置文件有这么几个

applicationContext.xml
jdbc.properties
log4j.properties
struts.xml
web.xml

我准备在本章中只讲 applicationContext.xml、jdbc.properties 和 web.xml。log4j 的配置大同小异而且也不在本文范围。至于 struts.xml 我准备留到后面与 Action 代码一起来讲，因为用的是 struts2-convention-plugin 插件来实现 struts2 的注解，所以这两个结合起来讲要好一些。

第二部分：分析配置文件

1、jdbc.properties

本例采用 MySQL 数据库，所以我设置了一个属性文件，用来存放一些连接信息和 Hibernate 相关的设置。

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test
jdbc.username=root
jdbc.password=1
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=false
```

因为我们使用的是 Hibernate 来与数据库进行交互，把这些东西写在单独的文件里，是方便修改，如果你想换成 SQL Server 或是 Oracle，只需要更改 driver、url 以及 dialect，而且还可以自由控制 sql 语句的显示的开关，非常方便。至于写在这里怎么用呢？请接着看下面的 applicationContext.xml 说明。

2、applicationContext.xml

这个文件就是 spring 的主配置文件了，当然，本例也只有这么一个 spring 的配置文件，内容不多，但做的工作还是很多的，下面我给大家详细分析一下。

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:jdbc.properties</value>
    </list>
  </property>
</bean>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driver}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

我把这两部分放在一起是因为这两者是相互联系的，而且也比较容易说明。可以这样来理解，PropertyPlaceholderConfigurer 这个类就是读取 jdbc.properties 文件，并将它们设置到这个类的属性中。然后再将下面数据源配置中定义的这些 \${jdbc.driver}、\${jdbc.url} 字符串换成属性文件中相同名称的值。\${} 这种写法，是类里面方法解析用的，网上都说这是叫占位符，我看了源代码的，其实是把它们当成字符串截取前后的特殊字符，再根据里面定义的名称找属性文件中对应的值。所以这个类只能读取 properties 格式的文件，你如果还有其它需要加入的属性文件，可以在 list 之间加入，写在 value 标签里面。

```
<context:component-scan base-package="com.bolo.examples" />
```

根据 base-package 指定的路径，扫描其下所有包含注解的 Bean，并自动注入。比如 @Repository，@Service 这些都是注解，前者表示持久层，后者表示业务层。这可是非常非常好的一个功能，是从 Spring 2.5 开始加入的一个非常棒的特性。有了它，我们将不用再去写那繁琐的 <bean id="" class="" />。本文的主旨就是全注解，就是为了告诉大家不用写配置文件（当然不是绝对不写）来怎样进行开发工作。关于这部分的具体情况，在后面代码章节中会详细讲解。


```

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
            <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
        </props>
    </property>
    <property name="packagesToScan" value="com.bolo.examples.entity.*" />
</bean>

```

www.blogjava.net/bolo

这就是在 Spring 中定义 Hibernate 相关的配置，Spring 已经集成了这部分功能。通过 class 里面定义的类名称我们很容易就能理解，这是使用注解的方式映射实体以及创建 Hibernate SessionFactory。\${hibernate.dialect}、\${hibernate.show_sql} 和上面的数据源配置获取方式一样，当 applicationContext.xml 定义好之后，就不用再对它进行修改，而是将修改对象变成了 jdbc.properties 文件。

另外在 Spring2.5.6 版中，加入了一个很有用的功能，就是 packagesToScan 属性，它是根据 value 中定义的路径来扫描其下所有的注解实体类。大象对这个路径做了多种测试，另外又看了源代码，发现它只能匹配某一类型的路径，而不是所有路径。比如上面的 value 值表示，扫描 entity 包下面的所有包中的注解类，如果你将类直接放在 entity 包下，那么服务器启动和程序运行时都不会报错，但是当你的代码需要用到这个类的时候，就会出现异常，提示你找不到实体。

```

<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>

```

www.blogjava.net/bolo

这是事务定义，而且使用注解方式定义事务（@Transactional），proxy-target-class="true" 表示采用动态代理类来管理事务，如果是 false 表示采用接口代理来管理事务（默认值为 false）。什么意思呢？就是说对于需要加入事务处理的类，如果是实现接口，那么将采用 Spring 的默认事务管理（Spring 默认方式为接口），如果不采用接口，而直接使用类，那么就需要 cglib 类库的支持，它通过动态的创建目标类（就是你需要加入事务的类）的子类，然后对这子类中的方法（当然是从目标类中继承来的）进行事务管理。这其实就是 AOP 切面，而且从中可以看出来，需要加入事务的方法不能为 private、static、final 的方法。这样说也不是很严格，说它不能加入事务，是说它不能主动的启动一个事务，如果某个 private 方法是被某个 public 方法调用的，而 public 方法是可以被动态代理加入事务的，所以这个 private 方法也一样被加入了事务，只是它处在 public 方法的事务之中。但是 static 和 final 这两类方法因为不能被子类覆盖，所以无法加入事务。如果这两类型的方法不被其它的事务方法所调用，那么它们就会以无事务的方式运行，因此很容易造成隐患，这一点请大家特别注意。

```

<aop:config proxy-target-class="true">
  <aop:advisor pointcut="execution(* com.bolo.examples.service..*Manager.*(..))" advice-ref="txAdvice" />
</aop:config>
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true" />
    <tx:method name="find*" read-only="true" />
    <tx:method name="query*" read-only="true" />
    <tx:method name="is*" read-only="true" />
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>

```

www.blogjava.net/bolo

上面这个就是使用配置式来定义事务，两种方式的区分主要是，注解式只用写那么一句话，然后在业务类或方法中加入@Transactional 这个注解标记，就完成事务声明，不过对于每个业务类都需要在类或方法中加入这些标记。而配置式声明，就是不用加这些标记，只要你的方法名称命名比较统一，就可以像上面这样定义事务规范，然后在 aop 标签中定义切入点与执行通知就行了。我感觉如果业务逻辑不是太复杂的情况，配置式会比较简单，而且修改起来也方便，这两种方式我都写出来了，至于用哪一种，由你们自己决定。

3、web.xml

现在使用的 Servlet 容器还是 2.4 版，因此 web.xml 里面还是需要写配置文件的，到了 3.0 版就可以采取注解的方式来实现了。

```

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext*.xml</param-value>
</context-param>

```

www.blogjava.net/bolo

Spring ApplicationContext 配置文件的路径，可使用通配符，applicationContext*.xml 表示所有以 applicationContext 开头的 xml 文件。多个路径用,号分隔。比如可以这样写：

```

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml,classpath:applicationContext-service.xml</param-value>
</context-param>

```

www.blogjava.net/bolo

不过推荐采用通配符的写法，能够简单点，为什么还要弄那么复杂呢？

context-param 是在容器启动后最先被执行的，并且被放入到容器上下文中。在这里引入 spring 的配置文件，是供 Spring 的 ContextLoaderListener 监听器使用。而这个监听器中会有一个 ContextLoader 类用来获取这个配置文件中的信息。从而进行 Spring 容器的初始化工作。因为是采用注解的方式来进行开发，所以 spring 的配置文件其实只有一个，上面那个星号可以去掉。

```

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

```

www.blogjava.net/bolo

这个监听器就是为了读取 Spring 的配置文件，这在上面已经讲到了。

```

<listener>
  <listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>
</listener>

```

www.blogjava.net/bolo

这是 Spring 提供的一个用来防止内存泄漏的监听器。在我们使用 struts2 框架，或其它的某些类

库时，因为它们自身的设计，会用到 Introspector（内省）机制来获取 Bean 对象的信息。但不幸的是，这些框架或类库在分析完一个类之后却没有将它从内存中清除掉，内存中还保留有大量的静态资源，而这些东西又无法进行垃圾回收，因此产生了很严重的内存泄漏问题。直接表现为服务器的内存使用会随着时间而不断上升，最后的结果当然就是服务器当掉。所以在这里加入此监听器，能够帮助我们更好的处理内存资源回收的问题。

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

www.blogjava.net/bolo

这是 Spring 的编码过滤器，我们可以直接拿来用，相信这段配置应该很好理解，不过请大家注意 forceEncoding 这个参数，把它设置为 true 表示不管请求中的编码是什么格式，都将强制采用 encoding 中设置的编码方式。另外对于响应也将按照 encoding 指定的编码进行设置。另外不建议将编码设置成 gb2312 或是 gbk 格式，请采用基于 Unicode 的 UTF-8 编码。

```
<filter>
  <filter-name>hibernateOpenSessionInViewFilter</filter-name>
  <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>
</filter>
```

www.blogjava.net/bolo

这个过滤器是个好东西，有了它，我们在使用 Hibernate 延迟加载的时候，就不会再为因 Session 被关闭，导致延迟加载数据异常而头痛了。网上有很多人说这个不好，其实在使用中，效果还是不错的。

```
<filter>
  <filter-name>struts2CleanupFilter</filter-name>
  <filter-class>org.apache.struts2.dispatcher.ActionContextCleanUp</filter-class>
</filter>
```

www.blogjava.net/bolo

首先我要说这个过滤器的名字很雷，不知道写这类的家伙是不是个变态，或者喜欢恶搞。主要原因就是，这个过滤器的功能是推迟清理值栈中的值，以便在 web 层中进行访问，另外就是为了配合 SiteMesh 装饰器进行工作（官方中的说明）。如果不加这个，那么 Struts2 的默认过滤器就会清空值栈中的值，这样就会导致异常。所以说这类的名字和功能完全不搭边，很容易让人产生误解。

```
<filter>
  <filter-name>struts2Filter</filter-name>
  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>
```

www.blogjava.net/bolo

在 2.1.6 版本里面，已经用这个过滤器取代了以前的 FilterDispatcher，而且在 api 文档中已经标注为 `@deprecated`（不赞成），并说明是从 Struts 2.1.3 版开始就弃用这个过滤器了，改用 StrutsPrepareAndExecuteFilter，除此之外，还可以选择 StrutsPrepareFilter 和 StrutsExecuteFilter。不过大象建议大家还是选择 StrutsPrepareAndExecuteFilter 吧，这也

是官方推荐的。

web.xml 里面的几个重要的配置就这些，不过不要忘了，给这些 filter 加上 filter-mapping 映射。还有一点，请注意这些过滤器的顺序，这个顺序是很重要的，程序运行时，是根据这些 filter-mapping 的排列顺序依次执行过滤操作的。如果不想出现莫名其妙的错误，请控制好这些过滤器映射的顺序。

我会在最后一章附上源码，大家就这样慢慢看吧。看到最后一章的时候，可能这些相关的知识就比较清楚了。到时再对照源码练习下，应该会有一些收获。

在前两章我为大家详细分析了 JAR 包的选择和必须的配置文件，那么这一章，我就对例子的层次结构进行说明，并实现除 WEB 层的功能代码。

第三部分：建立框架代码

工程结构



大家可以看到，本例一共分为：dao、entity、service、web 四层。另外在这些层次下，还以业务功能再进行分包，这样做是为了方便在以后的功能扩展中，能更好的管理和维护代码。如果将所有类都直接集中在这 4 个包下面，随着类的增加，会越来越难以维护，而且查找起来也很费劲。

HibernateDao

在本例中，我是通过继承 Spring 提供的 HibernateDaoSupport 来实现持久层的基类。同时引入泛型参数，封装了一些基本操作方法。


```

/**
 * 扩展HibernateDaoSupport的泛型基类
 * @author 菠萝大象
 * @param <T> 实体类型
 * @version 1.0
 */
public class HibernateDao<T> extends HibernateDaoSupport{

    protected Class<T> entityClass;

    public HibernateDao() {
        entityClass = ReflectUtils.getClassGenericType(getClass());
    }

    public Criteria createCriteria(Class entityClass, Criterion... criterions){
        Criteria criteria = getSession().createCriteria(entityClass);
        for (Criterion c : criterions) {
            criteria.add(c);
        }
        return criteria;
    }

    public Criteria createCriteria(Criterion... criterions){
        return createCriteria(entityClass, criterions);
    }

    public <X> X get(final Class<X> clazz, final Serializable id) {
        return (X) getSession().get(clazz, id);
    }

    public T get(Serializable id){
        return get(entityClass, id);
    }
}

```

bolo.blogjava.net

这是 HibernateDao 的部分代码，引入的这个泛型参数，其实就是实体类（User、Role）。通过传递这个实体类，在构造方法中利用反射特性将它从 JVM 中取出来。

```

/**
 * 反射工具类
 * @author 杜晶
 * @version 1.0
 */
public class ReflectUtils {

    public static <T> Class<T> getClassGenericType(final Class clazz) {
        return (Class<T>) ((ParameterizedType) clazz.getGenericSuperclass()).getActualTypeArguments()[0];
    }
}

```

bolo.blogjava.net

这里的 `getClass()` 方法是获得继承 `HibernateDao` 的类 (`UserDao`、`RoleDao`)

`getGenericSuperclass()` 方法就是通过这些继承了 `HibernateDao` 的类，来得到父类 (父类就是 `HibernateDao`) 的泛型。注意这个方法的返回值为 `Type`，这是一个类型接口。请参考 API。

因为在继承 `HibernateDao` 的时候，会给它加一个泛型参数。比如，`User`、`Role` 实体类。因此超类是参数化类型，所以返回的 `Type` 对象包含所使用的实际类型参数。这里返回的 `Type` 对象是 `ParameterizedType` 接口的实现类 `ParameterizedTypeImpl`，所以要将返回类型转型为 `ParameterizedType`。

`getActualTypeArguments()` 方法是 `ParameterizedType` 接口中的，它的作用就是获得实际类型参数 `Type` 对象的数组，因为我们这里只定义了一个泛型参数，数组里面也只有一个值，所以需要在数组下标处填 0。然后最后一步转型千万别忘记了，因为这个方法返回的可是是一个 `Type` 数组喔。

如果对于这部分的说明还有点不理解的话，请到时候有了代码，设个断点跟踪一下，就会全部清楚了。关于 java 反射，它已经超出本文的范围。大象只对本例中用到的部分进行讲解。

使用这种写法，是方便我们进行类型转换与类型检查。另外还可以简化某些方法的写法。比如：`createCriteria(Criterion... criterions)` 这个方法。参数是 `Criterion` 类型的可变参数，它是用来设置查询条件。如果要进行对象化查询，那么最简单的写法就可以直接写成 `createCriteria()`。另外还有重载的方法，可以根据传入 `class` 类型来创建自定义查询。

dao

持久层的 `Dao` 类是根据实体类定义，一般是一个实体类就会有一个对应的 `Dao` 类。当然这要跟业务需求来设计，不是绝对的。另外你也可以为了简便而去掉 `dao` 层，将持久化操作与业务逻辑全部写在 `service` 层。

```
/**
 * 人员管理
 * @author 菠萝大象
 * @version 1.0
 */
@Repository
public class UserDao extends HibernateDao<User>{

    public List getUsers(){
        return super.createCriteria().list();
    }

    public User getUser(Serializable id){
        return super.get(id);
    }
}
```

bolo.blogjava.net

这些定义的方法是供 `service` 层调用，在业务层，将不会看到一行与持久层有关的代码，降低耦合性是这样做的目的。`@Repository` 注解的作用就是标注这个 `UserDao` 是一个持久层组件。还记得前一章讲到的扫描器吗？`component-scan` 它就是用来将标有 `@Repository`、`@Service` 这样的注解类扫描到 Spring 的容器里，并且同时对标有 `@Autowired` 注解的 Bean 启用了自动注入功能。这就是 Spring 从 2.5 开始提供的新特性。我们使用注解的方法，就可以告别那繁琐的配置文件定义。

entity

关于实体的定义就是使用 JPA 注解，关于这部分，我以前写过一篇文章专门讲这个，如果有不清楚的朋友可以先参考一下。[学习 JPA——Hibernate Annotation 使用实例](#)

本例中，我有两点要讲下。

第一、管理主键的表 `generator_table` 去掉原来单独定义的那个 ID 主键，把 `g_key` 设为主键，

整个表将只有两个字段，g_key 和 g_value。

第二、在 User 实体中，我将角色 ID (role_id) 与角色实体 (Role) 做了一个多对一关联。这一点是原来文章中没有讲过的。

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "role_id")
public Role getRole() {
    return role;
}

public void setRole(Role role) {
    this.role = role;
}
```

请注意 role_id 是 user 表的字段。我在本例中设定的是一个角色可以对应多个人员，所以这个 role_id 存的就是 role 表 id 的值。fetch = FetchType.LAZY 指定采用延迟检索，如果当你取得了 User 对象，但又不想取 Role 中的信息，这时，User 对象中的 role 属性是代理状态。Role 对象中的值都是空的。只有当你使用 role.id 或 role.name 进行取值的时候，hibernate 才会去数据库中查找对应的记录，因此在一定程度上降低了资源消耗。不过这里有点要注意，采用延迟检索的时候，需要加上前一篇讲到的 OpenSessionInViewFilter 过滤器。否则会遇到 session 关闭的异常。

service

```
/**
 * 人员管理
 * @author 菠萝大象
 * @version 1.0
 */
@Service
public class UserManager {

    @Autowired
    private UserDao userDao;

    public List getUsers(){
        return userDao.getUsers();
    }

    public User getUser(Serializable id){
        return userDao.getUser(id);
    }
}
```

@Service 表示这是业务层组件。在业务层需要对 UserDao 加上 @Autowired 注解，大象在这里将业务层的方法名与持久层的方法名定义为一样的，是我的一种习惯，大家可以按自己的想法来做。

测试

既然有了这么多代码，那我们就来测试一下吧，看看有没有问题。

```
11 /**
12  * 测试
13  * @author 菠萝大象
14  * @version 1.0
15  */
16 public class Test {
17
18     public static void main(String[] args) {
19         ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
20         UserManager userManager = (UserManager)context.getBean("userManager");
21         RoleManager roleManager = (RoleManager)context.getBean("roleManager");
22         User user = userManager.getUser(1);
23         System.out.println("ID: "+user.getId()+" 姓名: "+user.getName());
24         Role role = roleManager.getRole(1);
25         System.out.println("ID: "+role.getId()+" 角色名: "+role.getName());
26     }
27 }
```

问题 控制台 SVN 资源库

<已终止> Test (2) [Java 应用程序] D:\jdk1.8.0_18\bin\javaw.exe (2010-4-20 下午10:45:56)

ID: 1 姓名: 张三

ID: 1 角色名: 超级管理员

blogjava.net

好吧，为了照顾那些坚定的 JUnit 拥护者，再写一个 JUnit 测试。本例使用 junit-4.4.jar


```
14 public class TestHibernateDao {
15
16     private static UserManager userManager;
17     private static RoleManager roleManager;
18
19     @BeforeClass
20     public static void init(){
21         ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
22         userManager = (UserManager) context.getBean("userManager");
23         roleManager = (RoleManager) context.getBean("roleManager");
24     }
25
26     @Test
27     public void testUser(){
28         User user = userManager.getUser(1);
29         Assert.assertEquals("张三", user.getName());
30         System.out.println("ID: "+user.getId()+" 姓名: "+user.getName());
31     }
32
33     @Test
34     public void testRole(){
35         Role role = roleManager.getRole(1);
36         Assert.assertEquals("超级管理员", role.getName());
37         System.out.println("ID: "+role.getId()+" 角色名: "+role.getName());
38     }
39 }
```

问题 控制台 SVN 资源库

<已终止> Test (2) [Java 应用程序] D:\jdk1.6.0_18\bin\javaw.exe (2010-4-18 下午09:09:16)

ID: 1 姓名: 张三

ID: 1 角色名: 超级管理员

bolo.blogjava.net



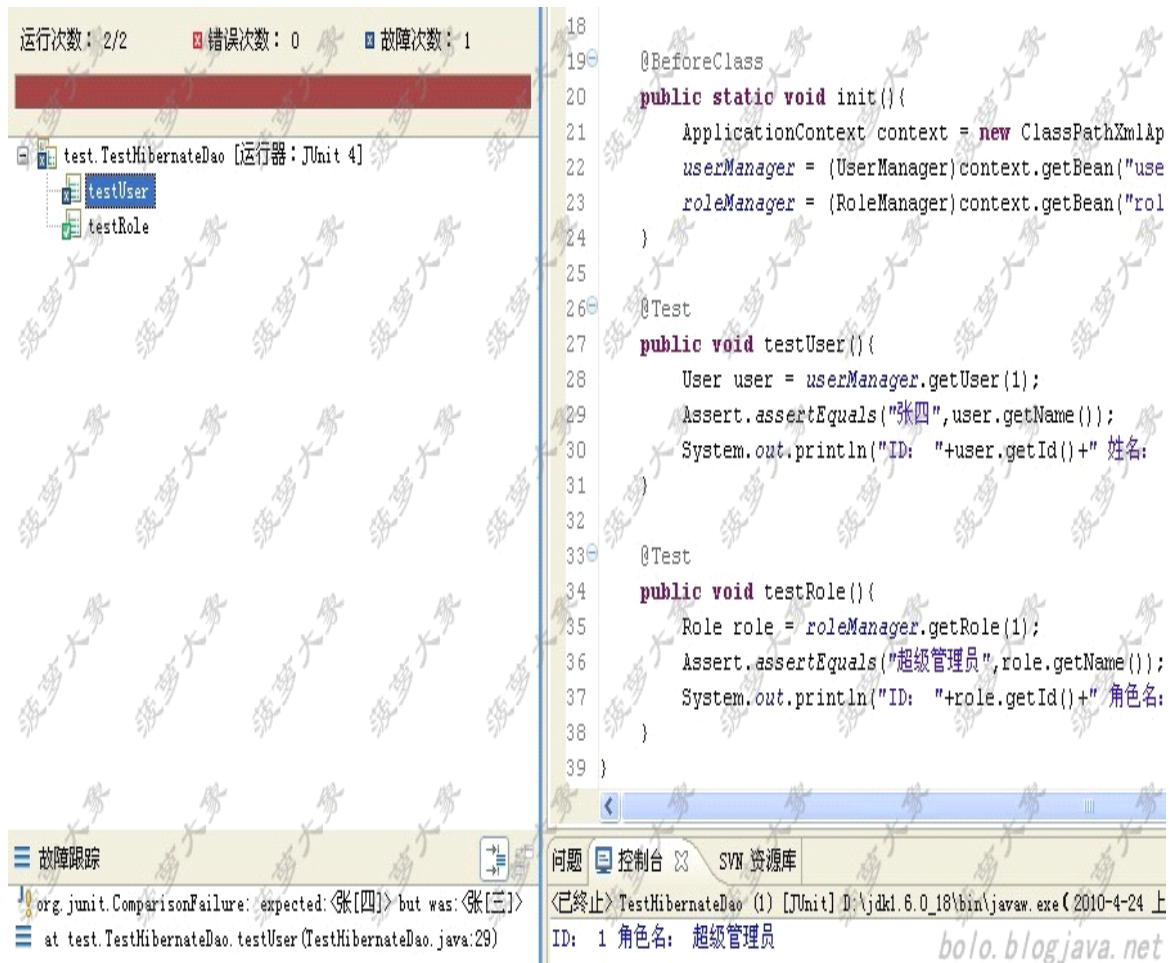
bolo.blogjava.net

`@BeforeClass` 注解的方法表示在类实例化之前，也就是在类的构造方法执行之前就会执行。而且使用这个注解的方法必须是 `static void`

`@Test` 标明这是测试方法，方法名不用像以前那样必须按照规则进行命名，可以自由定义。

上图显示大象使用 JUnit 方式测试也通过了（如果不会通过我写它干嘛？嘿嘿）。

假如我将张三改成张四，再来看看测试结果。



这个截图可以很明显的说明所有东西。

这一篇是给大家讲怎么用代码来实现除 web 层之外的全注解步骤。当然，我主要是讲思路，其实思路比代码重要得多。这一个系列的最后，我会放上所有源码供大家下载。现在这样慢慢分析，是想给大家讲道理。我们应该努力提升自己的境界与层次，而不要只把眼光放在代码上面。下一章将会着重介绍 web 层，以及 struts2 的注解插件 struts2-convention。

这一章，大象将详细分析 web 层代码，以及 struts2 的注解插件——struts2-convention 的用法和其它相关知识。

第四部分：透析控制层

上一章对 dao、entity、service 三层进行了详细的分析，并对代码进行了测试。测试结果表明这部分功能没问题，可以正常使用。本章将对最后一个 web 层进行详细说明，尽可能的讲明白这些知识要点。

数据库

本例使用 MySQL 数据库，只有三张表，一张用于管理表主键的 generator_table，另外两张是人员表与角色表。

```
CREATE TABLE `generator_table` (
  `g_key` varchar(100) NOT NULL,
  `g_value` int(11) default NULL,
  PRIMARY KEY (`g_key`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

bolo.blogjava.net

```
CREATE TABLE `role` (
  `id` int(11) NOT NULL,
  `name` varchar(50) default NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

CREATE TABLE `user` (
  `id` int(11) NOT NULL,
  `name` varchar(20) default NULL,
  `role_id` int(11) default NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

这里我有一点需要说明一下，在[学习 JPA——Hibernate Annotation 使用实例](#)一文中，我将 generator_table 设了一个 id 主键字段，其实这个字段是不需要的，直接将 g_key 设为主键。这样设计更好些，因为表名不可能一样，所以这个存放各个表主键的键名也不会一样。

user 与 role 这两张表只设了一个主键，没有建立外键关联，而且大象也很反对建立表之间的外键关联。因为这样做之后，约束太多，在实际开发中，很容易出问题，这是我亲身体会过的。所以我建议只对表设置一个流水号主键，其它的都可以根据业务关系来设计字段，这样会更灵活。

这里对各个字段都默认将它们设置为 null，因为针对不同的表，你都会实现相应的功能，你当然会知道哪些字段是不能为空的，哪些是可以为空的。而且在做数据库设计的时候，你也不可能在短时间内，面面俱到的把所有问题都考虑进去，根据需求的变化，在开发过程中，也是经常会遇到修改数据库的情况。如果之前过于强调字段的非空设置，在编写代码时，为了减少出错，脑袋里可能会不停的想，啊，这个字段是非空的吗？哪个字段不是非空的吧？然后反复对比数据库进行检查，会使人束手束脚很不舒服。因为这些全部都可以人为来控制，所以除了主键外，将其它字段都设为 null 有利于开发人员更好的进行工作。

有人会说了，进行非空设置是一种约束，当程序出错时，很容易发现问题。当然，这话说得没错。大象只是建议，从没说过一定要这样做，我只是说下自己的一点经验总结，仅此而已！想怎么实现都是你的自由。

struts2-convention

既然说了是全注解开发，而且我们已经实现了 Hibernate 与 Spring 的注解。同样的，Struts2 也能够做到用注解来代替配置文件，struts2-convention 插件可以帮助我们完成这一功能。它是 struts2 提供的一个插件，目前网上相关的中文文档主要是一个叫石太洋的人根据官方文档翻译的，很多网站与博客都有转载。我看了原文与译文，感觉讲的不够清楚，例子也很简单。大象根据自己在项目中的实际使用情况，现将个人对这个插件的经验总结写出来与各位分享，希望与大家多交流，共同提高。

官方文档 <https://cwiki.apache.org/WW/convention-plugin.html>

请不要把地址中的两个大写 W 换成小写，否则是打不开页面滴！这个插件的使用其实非常简单，如果光看文档可能会觉得好像很麻烦。那么大象来告诉你怎样快速学习这个插件。

首先你要搞清楚，这个插件它会默认扫描所有包名为 struts、struts2、action、actions 下面的类。然后它会对实现了 Action 接口以及类名以 Action 结尾的这些类，作为 Action 来进行处理。

你可以重新定义按哪种包名进行扫描。比如本例设定，只扫描 web 包下面的所有类，因为我们将 Action 类都放在这个包下面。

那这个插件是怎么实现原来的配置信息的呢？它的映射规则是这样的，对于以 Action 结尾的类，去掉 Action，剩下的部分，将所有的字母转换为小写，如果有驼峰式的写法，则用“-”连接符来连接不同的单词，这是此插件的默认方式。最终转换之后的就是请求地址，还是用例子说明。

`com.bolo.examples.web.base.UserAction`

按照上面的规则，请求地址就应该是 UserAction 去掉 Action 后缀，将其余部分转换为小写，所以 user 就是我们的请求地址。不过，这还没有完，因为这里面还有一个命名空间的路径，在通常的配置文

件中，一般会将不同的功能进行划分，在 package 标签里加上 namespace 属性。使用这个插件，它会为你自动配上命名空间，默认的就是前面说到的以那四种名称为根目录的命名空间，它们之后的都将成为命名空间的名称。

`com.bolo.examples.struts.UserAction` 映射为 `/user.action`

`com.bolo.examples.struts.base.UserAction` 映射为 `/base/user.action`

要是我们不以 struts 或其它几种默认值为包名，又该怎么办呢？没关系，插件为我们提供了一种自定义根包的配置方式

```
<constant name="struts.convention.package.locators" value="web" />
```

上面这段配置是写在 struts.xml 里面的，它指定 web 为根，作用就相当于那四种默认值。

`com.bolo.examples.web.base.UserAction` 映射为 `/base/user.action`

`com.bolo.examples.web.HelloAction` 映射为 `/hello.action`

`com.bolo.examples.web.HelloWorldAction` 映射为 `/hello-world.action`

请注意驼峰写法的映射方式，假如这里不是 HelloWorld，而是 Helloworld，那就不会再是 hello-world.action，而是 helloworld.action 了。

既然已经知道了它的映射方式，接下来再看看这个插件是如何定义结果页面的。

convention 默认会到 /WEB-INF/content 文件夹下面查找对应的结果页面，这个文件夹的名字可以修改，需要在 struts.xml 中定义

```
<constant name="struts.convention.result.path" value="/WEB-INF/jsp" />
```

文件夹的名字改成了 jsp，这样定义后，convention 就会在这个文件夹下面查找结果页面。它的查找路径与映射的命名空间有关。默认规则是，在请求的命名空间下面，根据请求名称再结合方法返回的字符串生成最终的结果页面名称，再配以后缀名。convention 支持以 jsp、ftl、vm、html、htm 等五种后缀格式的文件。这里有个比较特殊的是如果方法返回 success，那么可以不用将它与请求名称拼接起来，直接使用请求名称作为返回页面的名称。还是举例子说明。

```
package com.bolo.examples.web;
```

```
import com.opensymphony.xwork2.ActionSupport;
```

```
public class HelloAction extends ActionSupport{
```

```
    public String execute(){  
        return SUCCESS;  
    }
```

```
    public String welcome(){  
        return "welcome";  
    }  
}
```

比如上面这段代码，HelloAction 处于我们定义的根包（web）下面，因此，它的 action 请求为 hello.action。这时，会默认执行 execute() 方法，由于返回的是 success 字符串，所以页面的名称可以简写为 hello.jsp，但是当执行 welcome 方法时，由于返回的字符串为 welcome，这时的页面名称则为 hello-welcome.jsp。convention 就是遵循这样的规则来进行命名，当然这只是最基本的，我们再来看看稍微复杂点的东东。


```

package com.bolo.examples.web.base;

import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;

import com.opensymphony.xwork2.ActionSupport;

@Results({@Result(name = "success", location="role-list.jsp"),
    @Result(name = "redirectUser", location="user.action", type="redirectAction")})
public class RoleAction extends ActionSupport{

    public String execute(){
        return SUCCESS;
    }

    public String input(){
        return INPUT;
    }

    public String redirectUser(){
        return "redirectUser";
    }
}

```

这个 RoleAction 类的外部,加了两注解,它们的作用相当于配置文件中的 result 标签.Results 是一个 Result 类型的数组注解,里面可以包含多个 Result 配置。使用 Result 注解来设置返回类型与返回页面,是不准备采取默认的定义方式。比如 HelloAction 就是我们采取的默认方式。另外对于有些特殊的返回类型,也需要显式的进行定义。

因为我对 RoleAction 中的 execute() 方法返回结果进行了显式的定义,所以,它将不再返回默认的 role.jsp,而是 location 指定的 role-list.jsp, Result 注解中的 name 值要与返回值对应。

当请求路径为 role!input.action 时,会执行 input() 方法,对于这个方法来说,由于没有进行显式的定义,所以它会按照默认的命名规则返回 role-input.jsp。

而 redirectUser 方法的返回结果指定了一个 type 为 redirectAction 的值,这表示要对 Action 重定向,在 location 中也说明了是跳转到哪个 Action。请注意这里指定的是 user.action,当程序跳转到 UserAction 时,会默认执行 execute 方法。

假如说,你想执行其它方法该怎么办呢?可以在 location 里面这样定义, location="user!input.action"。请记住,重定向时,如果是跳转到其它 Action 或本 Action 中的其它方法, type 要写成 redirectAction。

更进一步,我还想带些参数过去,又该如何呢?请添加 params 属性,它是一个数组类型。可以这样定义, params={"role_id", "\${role_id}", "role_name", "超级管理员"}。convention 文档中有说明,里面的参数是一个键值对,总是形如 key,value, key,value。所以第一个 role_id 与第三个 role_name 都叫参数名,二和四则是参数值。另外注意下 "\${role_id}" 的含义,这是使用的 OGNL 表达式取出存在于值栈中的名叫 role_id 的值。这是一种动态获取并赋值的方式,在采用配置文件的方式中,也可以这样运用,而 role_name 参数则是一个固定字符串值。需要特别注意的就是,作为参数名的 role_id 与 role_name,一定要在指向的 Action 中有这两个同名的属性,并且还有 set 方法,这是用来给这两个属性赋值。而对于 \${role_id},则要在当前这个 Action 中,有它的 get 方法。用于取值。

补充说明一下,在 Action 类中定义的全局变量,不是非得给它都加上 set、get 方法,这是根据实

际情况来设置的。简单的说 `get()` 是获得值, `set()` 是设置值。比如, 你现在要在页面上显示 `username`, 那么就对这个属性设置 `get` 方法, 如果只是对 `username` 设置值, 从页面传值到 `Action`, 那只需要对它设置 `set` 方法就可以了。除此之外, 我们也可以不采用 `struts2` 提供的值栈方式得到参数值, 而是使用非常熟悉的 `request.getParameter()` 方法来获取参数。至于实际怎么使用, 由各位自己决定, 不知道我这样说, 大家能不能明白?

大象根据实际使用情况, 发现动态参数的传递在 `struts2.1.6` 存在 `BUG`, 如果需要使用这个功能, 请将 `struts2` 升级到 `2.1.8.1` 版。

大象根据实际应用, 建议大家统一在类名上面定义 `Results` 设置, 这样做有利于开发与维护; 不建议单独对方法使用 `@Action` 注解来重新定义它的访问地址与返回结果, 因为这样做有些破坏统一性, 不过可以根据实际情况进行处理, 但不要过多的使用。

struts.xml

```
<struts>
  <!-- 指定默认的父亲包 -->
  <constant name="struts.convention.default.parent.package" value="bolo-default" />
  <!-- 设置convention插件默认的根包 -->
  <constant name="struts.convention.package.locators" value="web" />
  <!-- 搜索此配置下的所有包 -->
  <constant name="struts.convention.package.locators.basePackage" value="com.bolo.examples" />
  <!-- 继承convention-default包, 定义一个默认的拦截器, 根据需要还可扩展 -->
  <package name="bolo-default" extends="convention-default">
    <interceptors>
      <interceptor-stack name="boloStack">
        <interceptor-ref name="paramsPrepareParamsStack" />
      </interceptor-stack>
    </interceptors>
    <default-interceptor-ref name="boloStack" />
  </package>
</struts>
```

整个 `struts.xml` 的配置文件就这么多, 当然你自己还可以扩展, 因为采用了注解, 所以以前的那些配置就再也看不到了。在这个文件中, `package` 是继承 `convention-default`, 而没有继承 `struts-default`, 为什么呢? 查看 `convention` 的 `struts-plugin.xml` 文件, 我们可以发现 `convention-default` 继承了 `struts-default`, 所以这样写是没错的。另外的几个 `constant` 配置就是对 `convention` 的常量设置, 请看注释。

关于 `paramsPrepareParamsStack` 拦截器栈, 我准备在第五篇, 对基础框架进行扩展的时候再详细的说明。大家如果等不急想学习下, 可以在网上查找这方面的资料先看看。

web

大象是这样想的, 如果一次讲的太多太复杂不利于理解和吸收, 所以对于 `web` 层, 大家从前面也看到了, 代码很简单, 因为本篇主要是讲 `convention` 插件的知识, 然后实现一部分功能用于演示它的效果。下面贴上 `web` 和 `WebRoot` 目录结构、`UserAction` 的代码, 以及 `jsp` 代码。





```
public class UserAction extends ActionSupport{

    @Autowired
    private UserManager userManager;

    public String execute(){
        return SUCCESS;
    }

    public List getList(){
        return userManager.getUsers();
    }
}
```

请注意 web 包下面的层次结构，这与你的请求路径相关。content 文件夹是插件默认指定的名字，你可以修改为别的名字。同样请注意在这个目录下面的文件与子文件夹的定义方式是和 web 层相同的。如果还没有理解，请再看下我对 convention 插件的说明。

在 web.xml 文件中，设置了一个<welcome-file-list>标签，定义了一个 index.jsp，这文件里就一句代码 `<% response.sendRedirect("hello.action");%>`它会去执行 HelloAction 的 execute() 方法，这方法里面什么逻辑都没有，直接返回结果页面 hello.jsp

```
<body>
  <h1>这是HelloAction默认的hello.jsp</h1>
  <div>
    <h3><a href="${ctx}/base/user.action">点击链接，执行UserAction</a></h3>
  </div>
  <div>
    <h3><a href="${ctx}/base/role.action">点击链接，执行RoleAction</a></h3>
  </div>
</body>
```

`${ctx}` 是一个 EL 表达式，设置的是当前项目名称。我在文件开头加了一个静态包含，
`<%@includefile="/common/taglibs.jsp"%>`

```

<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="ctx" value="${pageContext.request.contextPath}" />

```

不管是 user.action 还是 role.action, 它们默认的执行方法都是 execute(), 点击这两个链接, 返回指定的结果页面。

```

<body>
  <h1>这是UserAction对应的user.jsp</h1>
  <div>
    <table width="100%" border="0" align="center" cellpadding="3" cellspacing="1">
      <tr align="center">
        <th>序号</th>
        <th>姓名</th>
        <th>角色</th>
      </tr>
      <s:iterator value="list">
        <tr align="center">
          <td>${id}</td>
          <td>${name}</td>
          <td>${role.name}</td>
        </tr>
      </s:iterator>
    </table>
  </div>
  <div>
    <h3><a href="${ctx}/hello.action">返回</a></h3>
  </div>
</body>

```

```

<body>
  <h1>这是RoleAction设置Result属性转向的role-list.jsp</h1>
  <div>
    <div><h4>通过RoleAction的redirectUser方法重定向到user.action</h4></div>
    <h3><a href="${ctx}/base/role!redirectUser.action">点击链接, 执行redirectUser</a></h3>
  </div>
  <div>
    <h3><a href="${ctx}/hello.action">返回</a></h3>
  </div>
</body>

```

在 user.jsp 里面, 用来循环的 list, 是根据 getList() 方法获取的, struts2 会自动的分析出属性名。想一下, list 的 get 方法是不是就是 getList() 呢? 我之前说过, get() 是获得值, set() 是设置值。在这里我只是要在列表页面上得到 list 集合, 没有其它的需求, 所以不用像这样定义 private List list, 再然后给它加上 set()、get() 方法, 因为要得到 list 集合, 所以还要在 execute() 方法里面写上 list = userManager getUsers(), 这样做有必要么? 我一直都在遵循优雅、高效、简洁的代码风格, 并且一直都在朝这方面努力, 也提倡大家这样做。编程是门艺术, 而不是一种工作, 不要把

它当工作看，只想着完成任务，拼命的堆代码。这样做很难有提高。应该换一种心态去对待它，用艺术的眼光来重新审视你的代码，你会发现这很有趣，也会学到很多。自己的一点浅薄之见，让各位见笑了。

这部分的内容就说到这里，下一篇将对 `paramsPrepareParamsStack` 拦截器栈进行详细说明，另外再对框架进行一下扩展，封装 CRUD 功能，只要没有特殊的业务逻辑，在你的实际 Action 中，再不会看到增删改查这些基本功能。

这是本系列的最后一章，大象对示例进行适当的扩充并说明。

其实到第四篇，对于示例的说明就已经全部讲完了，如果按照这样的例子，很难有什么值得学习的地方。大象本着写点有用东西的原则，在这章，对示例进行一下适当的扩充并说明。

第五部分：扩展框架

`paramsPrepareParamsStack` 拦截器栈

`paramsPrepareParamsStack` 这个拦截器栈是在 `struts2-default.xml` 中定义的，里面包含了很多个拦截器，最重要的是这三个：`params`、`prepare`、`modelDriven`。我们只要记住这样几点。

`params`：它负责将请求参数值设置到 Action 中与之同名的属性中。

`prepare`：当 Action 实现了 `Preparable` 接口时，这个拦截器就会调用 `prepare()` 方法。如果你有想在 `execute()` 方法之前执行的逻辑处理，它就可以帮你完成这个功能。

`modelDriven`：如果 Action 实现了 `ModelDriven` 接口，这个拦截器就会把 `getModel()` 方法中的返回结果压入值栈。这就意味着，可以在结果页面上直接使用 `model` 对象的属性。

它的执行顺序是这样的

首先，`params` 拦截器会给 action 中的相关参数赋值，如 `id`，`username`，`password` 等等。

然后，`prepare` 拦截器执行 `prepare()` 方法，`prepare()` 会根据参数，如 `id`，去调用相关的方法，设置 `model` 对象。当然，实现的这个接口方法由你自己来定义，不局限只设置 `model` 之类的功能。

接着，`modelDriven` 拦截器会将 `model` 对象压入值栈，因为它是把 `getModel()` 方法中的返回结果放到值栈中，而这个方法的返回类型是个泛型参数，在实现 `ModelDriven` 接口的时候，可以给它指定一个具体的对象类型，因此返回类型也将是这个指定的对象类型，如 `ModelDriven<User>`

最后，`params` 拦截器会将参数再赋值给 `model` 对象。

思考修改与保存这两种动作。当点击人员修改时，请求为：`user!input.action?id=1`，`params` 拦截器会将 `id` 参数值设置到 Action 中的 `id` 属性，请注意，`id` 属性要有 `set()` 方法，然后 `prepare` 拦截器开始在 `prepare()` 方法中，根据这个 `id` 值取得 `User` 对象，接着 `modelDriven` 会调用 `getModel()` 方法，此时，方法中返回的是 `user` 对象，所以会把 `user` 加入到值栈中，最后再执行一次 `params` 拦截

器，但这时没有其它的参数值需要赋值给 user 对象，所以程序会接着往下走，这里假定没有其它的业务逻辑，执行返回，字符串为 input，根据前面讲的插件知识，结果页面为 user-input.jsp，那么就跳转到修改页面了，而且页面中表单域将显示数据库中的值。如果理解了修改，那么保存也就清楚了。

prepare() 方法虽然不错，但是也有弊端，那就是它会对 Action 中的每个方法都进行拦截，不管你是执行 execute 还是 input，还是其它的自定义方法，它都会对其拦截，这当然不是我们所希望的。那有没有更好的方式？答案是肯定的，请接着往下看。

prepareMethodName

使用 prepare 拦截器的另一种形式，在 prepare 名称后面加上需要拦截的方法名。比如，你要拦截 input 方法，可以写成 prepareInput，需要拦截 save 方法，就写上 prepareSave。采取这样的方式后，将会在执行这些方法之前时，才对它们进行拦截。

例如，请求 role!input.action，会执行 RoleAction 中的 input 方法，如果我们设置了 prepareInput 方法，则会先进入此方法执行，执行完后再回到 input 方法往下执行。

请注意，在使用这种方式时，Preparable 接口定义的 prepare() 方法体内不要含有任何代码，就是说给这个方法一个空实现。这样，它就什么都不做，所有的拦截处理就全部交由相应的 prepareMethodName 来完成。

```
/**
 * Preparable接口的方法，设置为空方法体是屏蔽它去拦截所有的方法
 */
public void prepare() throws Exception {}
```

bolo.blogjava.net

```

public void prepareInput() throws Exception {
    prepareEntity();
}

public void prepareSave() throws Exception {
    prepareEntity();
}

public void prepareView() throws Exception {
    prepareEntity();
}

protected void prepareEntity() throws Exception{
    if (id != null) {
        entity = (T)hibernateDao.get(entityClass,id);
    } else {
        entity = entityClass.newInstance();
    }
}
}

```

blog.blogjava.net

它们在每个对应的方法之前执行。prepareEntity 就是来初始化实体对象，然后由 modelDriven 拦截器将 getModel() 方法中的返回结果放入值栈，当返回页面时，就可以直接取值了。

StrutsAction

重新定义一个基类，里面封装大部分的通用操作，主要依靠泛型来实现，将 hibernateDao 注解进来，通过继承这个基类进行基本的 CRUD 操作。本文末尾提供示例源码下载，里面有详细的注释，这里我只贴出部分重要代码进行说明，为了行文需要，有些注释去掉了，但源码里面都有，请大家放心。

```

/**
 * 扩展ActionSupport的泛型基类
 * <br />封装CRUD基本方法，可在子类进行覆盖重写
 * @author 菠萝大象
 * @param <T> 实体类型
 */
public class StrutsAction<T> extends ActionSupport implements ModelDriven, Preparable{

    protected Integer id; //实体类的主键ID
    protected Class<T> entityClass; //T的反射类型
    protected T entity; //T类型对象
    protected List<T> list; //页面列表list
    protected Logger logger = LoggerFactory.getLogger(StrutsAction.class); //日志记录
    public static final String RELOAD = "reload"; //重定向的返回字符串
    public static final String VIEW = "view"; //查看方法的返回字符串
    public static final String REDIRECT = "redirect"; //重定向，@Result type属性对应的值
    public static final String REDIRECT_ACTION = "redirectAction"; //Action之间重定向，@Result type属性对应的值

    @Autowired
    @Qualifier("hibernateDao")
    protected HibernateDao hibernateDao;

    /**
     * 在构造函数中利用反射机制获得参数T的具体类
     */
    public StrutsAction(){
        this.entityClass = ReflectUtils.getClassGenricType(getClass());
    }
}

```

boto.blogjava.net

通过扩展 ActionSupport，使用泛型参数，构造函数根据反射得到 T 的具体类型。


```
@Override
public String execute() throws Exception{
    doListEntity();
    return SUCCESS;
}
```

```
@Override
public String input() throws Exception{
    doInputEntity();
    return INPUT;
}
```

```
public String view() throws Exception{
    doViewEntity();
    return VIEW;
}
```

```
public String save() throws Exception{
    doSaveEntity();
    return RELOAD;
}
```

```
public String delete() throws Exception{
    doDeleteEntity();
    return RELOAD;
}
```

bolo.blogjava.net

这就是默认的执行方法，基本的操作，在这个超类里面都进行了定义，每个方法里面设置的以 do 开头的方法，是方便让子类进行覆盖，当基本的业务逻辑无法满足我们的需求时，就可以在子类重写这些方法。

```

protected void doListEntity() throws Exception{
    list = hibernateDao.getAll(entityClass);
}

protected void doInputEntity() throws Exception{}

protected void doViewEntity() throws Exception{}

protected void doSaveEntity() throws Exception{
    try {
        hibernateDao.save(entity);
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
}

protected void doDeleteEntity() throws Exception{
    try {
        hibernateDao.delete(entityClass, id);
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
}

```

方法有默认实现，主要是列表显示，保存和删除，新增和修改已经有 getModel() 方法取得实体，在页面上使用 s 标签就可以直接取值，除非有特殊的业务需求，否则不用覆盖 doInputEntity()、doViewEntity() 方法。请注意，当需要实现自己的逻辑时，只需要覆盖上面定义的这些方法，而不用重写 execute、input 之类。

我对 HibernateDao 又进行了适当的扩展与修改，提供了更多的基本封装方法，不过大家还可以继续添加。里面都有详细的注释，这里就不在赘述了。

功能扩展

我对例子做了两个功能，一个是角色表的增加、修改、删除、查看，另一个就是用户表的查询。可以从源码中看到，我在 RoleAction 中没有写一行关于增删改查的代码，因为它属于基本操作，超类中已经封装好了，所以这部分的代码都省了。对于用户表的查询，我覆盖了 doListEntity() 方法，在业务层进行条件封装，执行查询，返回结果。

```

/**
 * 覆盖数据集显示回调函数
 */
@Override
protected void doListEntity() throws Exception {
    list = userManager.queryResult(getRequest());
}

```

这个 list 就是在超类中定义的，因为默认实现中也用到了 list，另外 list 有一个 get 方法，用于在页面中显示。如果不想采取方式取得 list 集合，就重写 doListEntity() 方法。这里说明下，我是没有加分页功能的，大家可以按自己的方式添加分页查询。

在用户查询方法中，我使用的是 QBC 对象查询，因为这种方式很简洁，不过我在 HibernateDao 中也写了 HQL 和 SQL 方式的查询方法，并进行了封装，可以很方便的调用。

```
/**
 * 根据条件查询
 */
public List queryResult(HttpServletRequest request){
    List<Criterion> criterions = new ArrayList<Criterion>();
    String name = request.getParameter("name");
    if(StringUtils.isNotBlank(name))
        criterions.add(Restrictions.like("name",name,MatchMode.ANYWHERE));
    String role_id = request.getParameter("role_id");
    if(StringUtils.isNotBlank(role_id))
        criterions.add(Restrictions.eq("role.id",Integer.valueOf(role_id)));
    return userDao.query(criterions.toArray(new Criterion[criterions.size()]));
}
```

```
/**
 * 根据条件获取数据
 * @param criterions 数量可变的Criterion
 */
public List<T> query(final Criterion... criterions) {
    return createCriteria(criterions).list();
}
```

这个 queryResult 方法的定义，你可以改为传递用户名与角色 ID 的参数，大象在这里就是为了方便，直接使用 Request 请求。这里 userDao 调用的 query 方法是在 HibernateDao 里面封装的，因为继承了 HibernateDao，就直接在 Service 层拿来用了。至于具体的，可以去看源码。

页面部分没有进行大的调整，主要是将 role-list.jsp 重命名为 role.jsp，因为使用的是超类的默认实现。添加了 role-input.jsp 和 role-view.jsp 两个文件，并在 user.jsp 中，加入了查询条件。这些代码都很简单就不再贴了，而且前一篇也贴过一部分。

对于这个例子的完整讲解说明就到此结束了。大象还想补充说明一下，这个例子只适用于学习，不适合商用，想在实际项目中运用，还需要对框架做大量的改造工作。本系列只是基于 SSH2 入门学习之用，源码中不含 jar 包，下图是本例中所需的最少 jar 文件，大家只要下载了 spring、struts、hibernate 三个完整压缩包，那么这些 jar 基本上都包含了。



发布并启动 Tomcat，然后输入访问地址：<http://localhost:8080/ssh2>运行该示例，看看效果。