OpenDoc Series'

Spring 开发指南

V0.8 预览版

作者：夏昕 xiaxin(at)gmail.com

So many open source projects. Why not **Open** your **Doc**uments? J

文档说明

参与人员：

作者	联络
夏昕	xiaxin(at)gmail.com

(at) 为 email @ 符号

发布记录

版本	日期	作者	说明
0.5	2004.6.1	夏昕	第一预览版
0.6	2004.9.1	夏昕	补充“持久层”内容。
0.7	2004.9.10	夏昕	追加: Webwork / Struts in Spring 增加 PDF 格式文档书签。 修订“依赖注入实现类型”
0.8	2004.9.20	夏昕	增加 AOP 部分

OpenDoc 版权说明

本文档版权归原作者所有。
在免费、且无任何附加条件的前提下，可在网络媒体中自由传播。

如需部分或者全文引用，请事先征求作者意见。

如果本文对您有些许帮助，表达谢意的最好方式，是将您发现的问题和文档改进意见及时反馈给作者。当然，倘若有时间和能力，能为技术群体无偿贡献自己的所学为最好的回馈。

Open Document，并不是笔者一个人力所能及的事情，欢迎所有读者对文档中的谬误，以及不合理的地方给予指正。

Spring 开发指南

前言

2003 年年初，笔者在国外工作。其时，一位与笔者私交甚好的印度同事 Paradeep 从公司离职去斯坦福深造，临走送给笔者一本他最钟爱的书籍作为纪念。

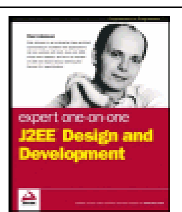
工作间隙，时常见到他摩娑此书，摇头不止（印度人习惯和中国人相反，摇头代表肯定、赞同，相当于与中国人点头。笔者刚开始与印度同僚共事之时，每每组织项目会议，一屋子人频频摇头，让笔者倍感压力......J ）。

下班后，带着好友离职的失落，笔者夹着这本书走在回家的路上，恰巧路过东海岸，天色依然明朗，随意坐上了海边一家酒吧的露天吧台，要了杯啤酒，随手翻弄着书的扉页，不经意看见书中遍布的钢笔勾画的线条。

“呵呵，Paradeep 这家伙，还真把这本书当回事啊”，一边笑着，一边摊开了此书，想看看到底是怎样的书让这样一个聪明老练的同事如此欣赏。

从此开始，这本书伴随笔者度过了整整一个月的业余时间.......

这本书，也就是出自 Rod Johnson 的：



《Expert One-on-One J2EE Design and Development》

此书已经由电子工业出版社出版，译版名为《J2EE 设计开发编程指南》。

半年后，一个新的 Java Framework 发布，同样出自 Rod Johnson 的手笔，这自然引起了笔者极大的兴趣，这就是 SpringFramework。

SpringFramework 实际上是 Expert One-on-One J2EE Design and Development 一书中所阐述的设计思想的具体实现。在 One-on-One 一书中，Rod Johnson 倡导 J2EE 实用主义的设计思想，并随书提供了一个初步的开发框架实现（interface21 开发包）。而 SpringFramework 正是这一思想的更全面和具体的体现。Rod Johnson 在 interface21 开发包的基础之上，进行了进一步的改造和扩充，使其发展为一个更加开放、清晰、全面、高效的开发框架。

本文正是针对 SpringFramework 的开发指南，讲述了 SpringFramework 的设计思想以及在开发中的实际使用。同时穿插了一些笔者在项目实作中的经验所得。

目录

Spring 初探......6
准备工作......6
构建 Spring 基础代码......7
Spring 基础语义......13
Dependency Injection......13
依赖注入的几种实现类型......16
Type1 接口注入......16
Type2 设值注入......17
Type3 构造子注入......17
几种依赖注入模式的对比总结......17
Spring Bean 封装机制......19
Bean Wrapper......19
Bean Factory......20
ApplicationContext......23
Web Context......28
Spring 高级特性......29
Web 应用与 MVC......29
Spring MVC......30
Spring MVC 指南......30
基于模板的 Web 表示层技术......44
Web 应用中模板技术与 JSP 技术的对比......49
输入验证与数据绑定......51
异常处理......62
国际化支持......64
WebWork2 & Spring......68
Quick Start......69
WebWork 高级特性......82
Action 驱动模式......82
XWork 拦截器体系......87
输入校验......93
国际化支持......107
Webwork2 in Spring......110
Struts in Spring......118
数据持久层......127
事务管理......127
持久层封装......131
JDBC......131
Hibernate in Spring......139
ibatis in Spring......146
Aspect Oriented Programming......150
AOP 概念......150
AOP in Spring......153

Dynamic Proxy 与 Spring AOP 153
CGLib 与 Spring AOP 163
AOP 应用 165
DAO Support 169
Remoting 169

Spring 初探

开始 **Spring** 研究之前，先让我们来看一个 **1** 分钟上手教程。

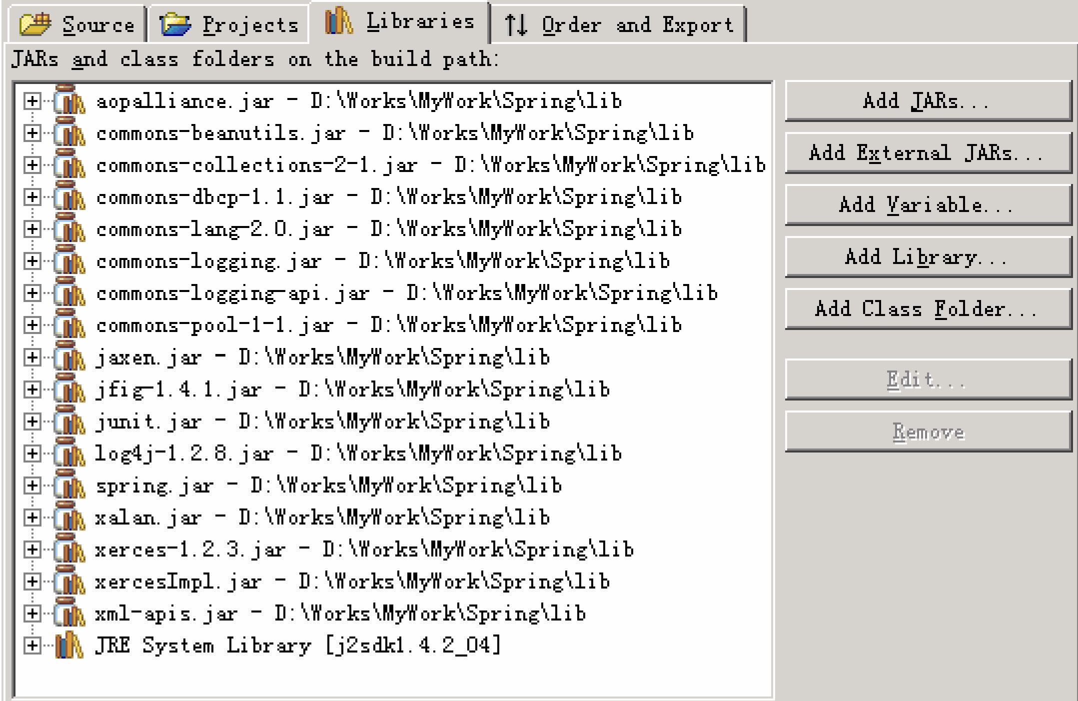
Quick Start!

准备工作

Ø 下载 **SpringFramework** 的最新版本，并解压缩到指定目录。

Ø 在 **IDE** 中新建一个项目，并将 **Spring.jar** 将其相关类库加入项目。

笔者所用 **IDE** 为 **Eclipse**，类库配置如下：

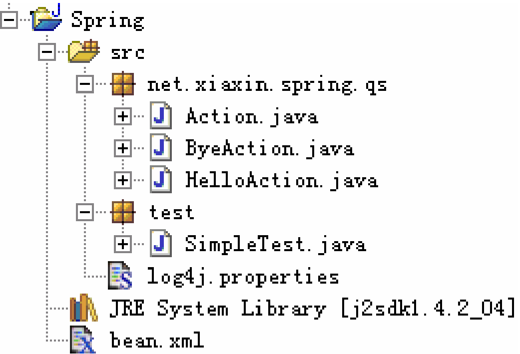


Ø **Spring** 采用 **Apache common_logging**，并结合 **Apache log4j** 作为日志输出组件。为了在调试过程中能观察到 **Spring** 的日志输出，在 **CLASSPATH** 中新建 **log4j.properties** 配置文件，内容如下：

Ø

```
log4j.rootLogger=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%c{1} - %m%n
```

配置完成后，项目结构如下图所示：



构建 Spring 基础代码

示例基础代码包括：

1. Action 接口：

   Action 接口定义了一个 execute 方法，在我们示例中，不同的 Action 实现提供了各自的 execute 方法，以完成目标逻辑。

```
public interface Action {

    public String execute(String str);

}
```

2. Action 接口的两个实现 UpperAction、LowerAction

```
public class UpperAction implements Action {

    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String string) {
        message = string;
    }

    public String execute(String str) {
        return (getMessage() + str).toUpperCase();
    }
}
```

UpperAction将其message属性与输入字符串相连接，并返回其大写形式。

```
public class LowerAction implements Action {

    private String message;

```

```
    public String getMessage() {
        return message;
    }

    public void setMessage(String string) {
        message = string;
    }

    public String execute(String str) {
        return (getMessage()+str).toLowerCase();
    }
}
```

LowerAction将其message属性与输入字符串相连接，并返回其小写形式。

3. Spring 配置文件（bean.xml）

```
<beans>
    <description>Spring Quick Start</description>
    <bean id="TheAction"
          class="net.xiaxin.spring.qs.UpperAction">
       <property name="message">
            <value>HeLLo</value>
       </property>
    </bean>
</beans>
```

（请确保配置bean.xml位于工作路径之下，注意工作路径并不等同于CLASSPATH ，eclipse的默认工作路径为项目根路径，也就是.project文件所在的目录，而默认输出目录/bin是项目CLASSPATH的一部分，并非工作路径。）

4. 测试代码

```
    public void testQuickStart() {

        ApplicationContext ctx=new
                  FileSystemXmlApplicationContext("bean.xml");

        Action action = (Action) ctx.getBean("TheAction");

        System.out.println(action.execute("Rod Johnson"));

    }
```

可以看到，上面的测试代码中，我们根据"bean.xml"创建了一个ApplicationContext实例，并从此实例中获取我们所需的Action实现。

运行测试代码，我们看到控制台输出：

```
......
HELLO ROD JOHNSON
```

我们将bean.xml中的配置稍加修改：

```
<bean id="TheAction"
        class="net.xiaxin.spring.qs.LowerAction"/>
```

再次运行测试代码，看到：

```
......
hello rod johnson
```

示例完成！

很简单的示例，的确很简单，甚至简单到了不够真实。
不过，不知大家从这个最简单的例子中看出了什么？
真的只是打印输出了两行不痛不痒的问候语？

仔细观察一下上面的代码，可以看到：

1. 我们的所有程序代码中（除测试代码之外），并没有出现Spring中的任何组件。

2. UpperAction和LowerAction的Message属性均由Spring通过读取配置文件（bean.xml）动态设置。

3. 客户代码（这里就是我们的测试代码）仅仅面向接口编程，而无需知道实现类的具体名称。同时，我们可以很简单的通过修改配置文件来切换具体的底层实现类。

上面所说的这些，对于我们的实际开发有何帮助？

Ø 首先，我们的组件并不需要实现框架指定的接口，因此可以轻松的将组件从Spring中脱离，甚至不需要任何修改（这在基于EJB框架实现的应用中是难以想象的）。

Ø 其次，组件间的依赖关系减少，极大改善了代码的可重用性。

Spring的依赖注入机制，可以在运行期为组件配置所需资源，而无需在编写组件代码时就加以指定，从而在相当程度上降低了组件之间的耦合。

上面的例子中，我们通过Spring，在运行期动态将字符串 “HeLLo” 注入到Action实现类的Message属性中。
现在假设我们回到传统的实现模式，应该如何处理？
一般的处理办法也就是编写一个Helper类（辅助类），完成配置文件读写功能，然后在各个Action的构造函数中，调用这个Helper类设置message属性值。

此时，我们的组件就与这个Helper类库建立了依赖关系，之后我们需要在其他系统中重用这个组件的话，也必须连同这个Helper类库一并移植。实际开发中，依赖关系往往并非如此简单，组件与项目基层代码之间复杂的关联，使得组件重用性大大下降。

Spring通过依赖注入模式，将依赖关系从编码中脱离出来，从而大大降低了组件之间的耦合，实现了组件真正意义上的即插即用。这也是Spring最具价值的特性之一。

- 面向接口编程。

  诚然，即使没有Spring，实现面向接口的设计也不困难。Spring对于面向接口设计的意义，在于它为面向接口编程提供了一个更加自然的平台。基于Spring开发，程序员会自然而然倾向于使用接口来定义不同层次之间的关联关系，这种自发的倾向性，来自于Spring所提供的简单舒适的依赖注入实现。Spring使得接口的定义和使用不再像传统编码过程中那么繁琐（传统编码过程中，引入一个接口，往往也意味着同时要引入一个Factory类，也许还有一个额外的配置文件及其读写代码）。

既然Spring给我们带来了如此这般的好处，那么，反过来，让我们试想一下，如果不使用Spring框架，回到我们传统的编码模式（也许正是目前的编码模式），情况会是怎样？

对于上例而言，我们需要怎样才能实现相同的功能？

上面的Action接口及其两个实现类UpperAction和LowerAction都与Spring无关，可以保留。而调用Action的测试代码，如果要实现同样的功能，应该如何编写？

首先，我们必须编写一个配置文件读取类，以实现Message属性的可配置化。

其次，得有一个Factory模式的实现，并结合配置文件的读写完成Action的动态加载。

于是，我们实现了一个ActionFactory来实现这个功能：

```
public class ActionFactory{

  public static Action getAction(String actionName){

      Properties pro = new Properties();

      try {

          pro.load(new FileInputStream("config.properties"));
          String actionImplName =
                   (String)pro.get(actionName);
          String actionMessage =
                   (String)pro.get(actionName+"_msg");

          Object obj =
                   Class.forName(actionImplName).newInstance();
          //BeanUtils是Apache Commons BeanUtils提供的辅助类
         BeanUtils.setProperty(obj,"message",actionMessage);
```

```
            return (Action)obj;

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

配置文件则采用最简单的**properties**文件形式：

```
TheAction=net.xiaxin.spring.qs.UpperAction
TheAction_msg=HeLLo
```

测试代码对应更改为：

```
  public void testFactory(){

      Action action = ActionFactory.getAction("TheAction");

      System.out.println(action.execute("Rod Johnson"));

  }
```

且不论实现质量的好坏，总之通过上面新增的**20**来行代码，我们实现了类似的功能（如果不引入**BeanUtils**，而采用手工编写**Reflection**代码完成属性设置的话，显然代码将远远不止**20**行）。

好吧，现在有个新需求，这个**ActionFactory**每次都新建一个类的实例，这对系统性能不利，考虑到我们的两个**Action**都是线程安全的，修改一下**ActionFactory**，保持系统中只有一个**Action**实例供其他线程调用。

另外**Action**对象创建后，需要做一些初始化工作。修改一下**ActionFactory**，使其在创建**Action**实例之后，随即就调用**Action.init**方法执行初始化。

嗯，好像每次创建Action对象的时就做初始化工作消耗了很多无谓资源，来个Lazy Loading吧，只有Action实例被实际调用的时候再做初始化。

差不多了，Action的处理就这样吧。下面我们来看看另外一个Factory。

……

往往这些系统开发中最常见的需求，会导致我们的代码迅速膨胀。纵使苦心经营，往往也未必能得全功。

而Spring的出现，则大大缓解了这样的窘境。通过对编码中常见问题的分解和抽象，Spring提供了一套成熟而全面的基础框架。随着本篇的进展，大家可以看到，上面这些开发中常见的问题在Spring框架中都提供了统一、妥善的处理机制，这为烦杂的应用开发提供了相当有力的支持。

这里暂且抛开Spring Framework在设计上相当出彩的表现不谈。站在应用开发的实际角度来说，其最大的优势在于：**Spring是一个从实际项目开发经验中抽取的，可高度重用的应用框架**。认识到这一点非常重要。

Spring Framework中目前最引人注目的，也就是名为控制反转（IOC ＝Inverse Of Control）或者依赖注入（DI ＝Dependence Injection）的设计思想，这的确是相当优秀的设计理念，但是，光一个单纯的设计模式并不能使得Spring如此成功，而Spring最成功的地方也并不仅仅在于采用了IOC/DI的设计。我们前面示例中的ActionFactory，勉强也可算做是一个IOC/DI设计的实现，但又如何？

可能相关技术媒体和不明就里的技术追随者对于DI/IOC容器的过分炒作，在某种程度上误导了初学者的视线。"控制反转"，这显然不是一个能望文知意的好名称；"依赖注入"，也好不到哪里去，也正因为这样，不少初学者都将Spring和生涩的所谓"控制反转"和"依赖注入"看作一个懵懂的高级概念而供上了神龛。

而实际上，Spring是笔者所见过的，最具实际意义的Java开发框架。它绝非一个高级概念玩具，而是一个切实的，能实实在在帮助我们改善系统设计的好帮手。

首先，Spring涵盖了应用系统开发所涉及的大多数技术范畴，包括MVC、ORM以及Remote Interface等，这些技术往往贯穿了大多数应用系统的开发过程。Spring从开发者的角度对这些技术内容进行了进一步的封装和抽象，使得应用开发更为简便。在笔者的开发工作中，借助Spring提供的丰富类库，相对传统开发模式，大大节省了编码量（平均1/3强，对于ORM和Remote层也许更多）。

其次，Spring并非一个强制性框架，它提供了很多独立的组件可供选择。如笔者在一些项目中，就仅引用了Spring的ORM模板机制对数据存取层进行处理，并取得了相当理想的效果。

评定一个框架是否优良的条件固然有很多种，但是笔者始终认为，对于应用系统开发而言，我们面临着来自诸多方面的压力，此时，最能提高生产力的技术，也就是最有价值的技术。很高兴，Spring让笔者找到了这样的感觉。

笔者对Rod Johnson最为钦佩的，并不是他用了IOC或者DI，而是他对J2EE应用开发的透彻的理解。

他真的明白开发人员需要什么。

Spring 基础语义

Dependency Injection

何谓控制反转（**IoC = Inversion of Control**），何谓依赖注入（**DI = Dependency Injection**）？对于初次接触这些概念的初学者，不免会一头雾水。正如笔者第一次看到这些名词一样，一阵窘迫......

IT 界不亏是哄抢眼球的行业，每个新出现的语汇都如此迷离。好在我们也同时拥有 **Internet** 这个最广博的信息来源。

IoC，用白话来讲，就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。这也就是所谓"控制反转"的概念所在：控制权由应用代码中转到了外部容器，控制权的转移，是所谓反转。

正在业界为 **IoC** 争吵不休时，大师级人物 **Martin Fowler** 也站出来发话，以一篇经典文章《**Inversion of Control Containers and the Dependency Injection pattern**》为 **IoC** 正名，至此，**IoC** 又获得了一个新的名字："依赖注入 （**Dependency Injection**）"。

相对 **IoC** 而言，"依赖注入"的确更加准确的描述了这种古老而又时兴的设计理念。从名字上理解，所谓依赖注入，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

为什么称之为"古老而又时兴"的设计理念？至于"时兴"自然不必多费唇舌，看看国内外大小论坛上当红的讨论主题便知。至于"古老"......，相信大家对下面图片中的设备不会陌生：

这就是笔者的主要工作装备，**IBM T40** 笔记本电脑一台、**USB** 硬盘和 **U** 盘各一只。想必大家在日常工作中也有类似的一套行头。

这与依赖注入有什么关系？

图中三个设备都有一个共同点，都支持 **USB** 接口。当我们需要将数据复制到外围存储设备时，可以根据情况，选择是保存在 **U** 盘还是 **USB** 硬盘，下面的操作大家也都轻车熟路，无非接通 **USB** 接口，然后在资源浏览器中将选定的文件拖放到指定的盘符。

这样的操作在过去几年中每天都在我们身边发生，而这也正是所谓依赖注入的一个典型案例，上面称

之为“古老”想必也不为过分。

再看上例中，笔记本电脑与外围存储设备通过预先指定的一个接口（USB）相连，对于笔记本而言，只是将用户指定的数据发送到 USB 接口，而这些数据何去何从，则由当前接入的 USB 设备决定。在 USB 设备加载之前，笔记本不可能预料用户将在 USB 接口上接入何种设备，只有 USB 设备接入之后，这种设备之间的依赖关系才开始形成。

对应上面关于依赖注入机制的描述，在运行时（系统开机，USB 设备加载）由容器（运行在笔记本中的 Windows 操作系统）将依赖关系（笔记本依赖 USB 设备进行数据存取）注入到组件中（Windows 文件访问组件）。

这就是依赖注入模式在现实世界中的一个版本。

很多初学者常常陷入“依赖注入，何用之有？”的疑惑。想来这个例子可以帮助大家简单的理解其中的含义。依赖注入的目标**并非为软件系统带来更多的功能**，而是为了提升组件重用的概率，并为系统搭建一个灵活、可扩展的平台。将 USB 接口和之前的串/并、PS2 接口对比，想必大家就能明白其中的意味。

回顾 Quick Start 中的示例，UpperAction/LowerAction 在运行前，其 Message 节点为空。运行后由容器将字符串“HeLLo”注入。此时 UpperAction/LowerAction 即与内存中的“HeLLo”字符串对象建立了依赖关系。也许区区一个字符串我们无法感受出依赖关系的存在。如果把这里的 Message 属性换成一个数据源（DataSource），可能更有感觉：

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

    <bean id="SampleDAO" class="net.xiaxin.spring.dao.SampleDAO">
       <property name="dataSource">
          <ref local="dataSource"/>
        </property>
    </bean>
</beans>
```

其中SampleDAO中的dataSource将由容器在运行期动态注入，而DataSource的具体配置和初始化工作也将由容器在运行期完成。

对比传统的实现方式（如通过编码初始化DataSource实例），我们可以看到，基于依赖注入的系统实现相当灵活简洁。

通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定SampleDAO中所需的DataSource实例。SampleDAO只需利用容器注入的DataSource实例，完成自身的业务逻辑，而不用关心具体的资源来自何处、由谁实现。

上面的实例中，我们假设SampleDAO是一个运行在J2EE容器中的组件（如 Weblogic）。在运行期，通过JNDI从容器中获取DataSource实例。

现在假设我们的部署环境发生了变化，系统需要脱离应用服务器独立运行，这样，由于失去了容器的支持，原本通过JNDI获取DataSource的方式不再有效。我们需要如何修改以适应新的系统环境？很简单，我们只需要修改dataSource的配置：

```
<beans>
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close">

        <property name="driverClassName">
             <value>org.gjt.mm.mysql.Driver</value>
         </property>

        <property name="url">
             <value>jdbc:mysql://localhost/sample</value></property>

        <property name="username">
             <value>user</value>
         </property>

        <property name="password">
             <value>mypass</value>
         </property>
    </bean>

    <bean id="SampleDAO" class="net.xiaxin.spring.dao.SampleDAO">
        <property name="dataSource">
          <ref local="dataSource"/>
         </property>
    </bean>
</beans>
```

这里我们的DataSource改为由Apache DBCP组件提供。没有编写任何代码我们即实现了DataSource的切换。回想传统编码模式中，如果要进行同样的修改，我们需要付出多大的努力。

依赖注入机制减轻了组件之间的依赖关系，同时也大大提高了组件的可移植性，这意味着，组件得到重用的机会将会更多。

依赖注入的几种实现类型

Type1 接口注入

我们常常借助接口来将调用者与实现者分离。如:

```
public class ClassA {
      private InterfaceB clzB;
      public doSomething() {
          Ojbect obj =
            Class.forName(Config.BImplementation).newInstance();
          clzB = (InterfaceB)obj;
          clzB.doIt()
      }
  ……
}
```

上面的代码中，ClassA依赖于InterfaceB的实现，如何获得InterfaceB实现类的实例？传统的方法是在代码中创建InterfaceB实现类的实例，并将起赋予clzB。

而这样一来，ClassA在编译期即依赖于InterfaceB的实现。为了将调用者与实现者在编译期分离，于是有了上面的代码，我们根据预先在配置文件中设定的实现类的类名(Config.BImplementation)，动态加载实现类，并通过InterfaceB强制转型后为ClassA所用。这就是接口注入的一个最原始的雏形。

而对于一个Type1型IOC容器而言，加载接口实现并创建其实例的工作由容器完成。

如下面这个类:

```
public class ClassA {
      private InterfaceB clzB;

      public Object doSomething(InterfaceB b) {
          clzB = b;
          return clzB.doIt();
      }
  ……
}
```

在运行期，InterfaceB实例将由容器提供。

Type1型IOC发展较早（有意或无意），在实际中得到了普遍应用，即使在IOC的概念尚未确立时，这样的方法也已经频繁出现在我们的代码中。

下面的代码大家应该非常熟悉：

```
public class MyServlet extends HttpServlet {

  public void doGet(
               HttpServletRequest request,
               HttpServletResponse response)
          throws ServletException, IOException {
      ……
  }
}
```

这也是一个Type1 型注入，HttpServletRequest和HttpServletResponse实例由Servlet Container在运行期动态注入。

另，Apache Avalon是一个较为典型的Type1型IOC容器。

Type2 设值注入

在各种类型的依赖注入模式中，设值注入模式在实际开发中得到了最广泛的应用（其中很大一部分得力于Spring框架的影响）。

在笔者看来，基于设置模式的依赖注入机制更加直观、也更加自然。Quick Start中的示例，就是典型的设置注入，即通过类的setter方法完成依赖关系的设置。

Type3 构造子注入

构造子注入，即通过构造函数完成依赖关系的设定，如：

```
public class DIByConstructor {
      private final DataSource dataSource;
      private final String message;

      public DIByConstructor(DataSource ds, String msg) {
          this.dataSource = ds;
          this.message = msg;
      }
  ……
}
```

可以看到，在Type3类型的依赖注入机制中，依赖关系是通过类构造函数建立，容器通过调用类的构造方法，将其所需的依赖关系注入其中。

PicoContainer（另一种实现了依赖注入模式的轻量级容器）首先实现了Type3类型的依赖注入模式。

几种依赖注入模式的对比总结

接口注入模式因为历史较为悠久，在很多容器中都已经得到应用。但由于其在灵活性、易用性上不如其他两种注入模式，因而在IOC的专题世界内并不被看好。

Type2和Type3型的依赖注入实现则是目前主流的IOC实现模式。这两种实现方式各有特点，也各具优势（一句经典废话J）。

Type2 设值注入的优势

1. 对于习惯了传统JavaBean开发的程序员而言，通过setter方法设定依赖关系显得更加直观，更加自然。
2. 如果依赖关系（或继承关系）较为复杂，那么Type3模式的构造函数也会相当庞大（我们需要在构造函数中设定所有依赖关系），此时Type2模式往往更为简洁。
3. 对于某些第三方类库而言，可能要求我们的组件必须提供一个默认的构造函数（如Struts中的Action），此时Type3类型的依赖注入机制就体现出其局限性，难以完成我们期望的功能。

Type3 构造子注入的优势：

1. "在构造期即创建一个完整、合法的对象"，对于这条Java设计原则，Type3无疑是最好的响应者。
2. 避免了繁琐的setter方法的编写，所有依赖关系均在构造函数中设定，依赖关系集中呈现，更加易读。
3. 由于没有setter方法，依赖关系在构造时由容器一次性设定，因此组件在被创建之后即处于相对"不变"的稳定状态，无需担心上层代码在调用过程中执行setter方法对组件依赖关系产生破坏，特别是对于Singleton模式的组件而言，这可能对整个系统产生重大的影响。
4. 同样，由于关联关系仅在构造函数中表达，只有组件创建者需要关心组件内部的依赖关系。对调用者而言，组件中的依赖关系处于黑盒之中。对上层屏蔽不必要的信息，也为系统的层次清晰性提供了保证。
5. 通过构造子注入，意味着我们可以在构造函数中决定依赖关系的注入顺序，对于一个大量依赖外部服务的组件而言，依赖关系的获得顺序可能非常重要，比如某个依赖关系注入的先决条件是组件的DataSource及相关资源已经被设定。

可见，Type3和Type2模式各有千秋，而Spring、PicoContainer都对Type3和Type2类型的依赖注入机制提供了良好支持。这也就为我们提供了更多的选择余地。理论上，以Type3类型为主，辅之以Type2类型机制作为补充，可以达到最好的依赖注入效果，不过对于基于Spring Framework开发的应用而言，Type2使用更加广泛。

Spring Bean 封装机制

Spring 从核心而言，是一个 DI 容器，其设计哲学是提供一种无侵入式的高扩展性框架。即无需代码中涉及 Spring 专有类，即可将其纳入 Spring 容器进行管理。

作为对比，EJB 则是一种高度侵入性的框架规范，它制定了众多的接口和编码规范，要求实现者必须遵从。侵入性的后果就是，一旦系统基于侵入性框架设计开发，那么之后任何脱离这个框架的企图都将付出极大的代价。

为了避免这种情况，实现无侵入性的目标。Spring 大量引入了 Java 的 Reflection 机制，通过动态调用的方式避免硬编码方式的约束，并在此基础上建立了其核心组件 BeanFactory，以此作为其依赖注入机制的实现基础。

org.springframework.beans 包中包括了这些核心组件的实现类，核心中的核心为 BeanWrapper 和 BeanFactory 类。这两个类从技术角度而言并不复杂，但对于 Spring 框架而言，却是关键所在，如果有时间，建议对其源码进行研读，必有所获。

Bean Wrapper

从Quick Start的例子中可以看到，所谓依赖注入，即在运行期由容器将依赖关系注入到组件之中。讲的通俗点，就是在运行期，由Spring根据配置文件，将其他对象的引用通过组件的提供的setter方法进行设定。

我们知道，如果动态设置一个对象属性，可以借助Java的Reflection机制完成：

```
Class cls =  Class.forName("net.xiaxin.beans.User");
Method mtd = cls.getMethod("setName",new Class[]{String.class});
Object obj = (Object)cls.newInstance();
mtd.invoke(obj,new Object[]{"Erica"});
return obj;
```

上面我们通过动态加载了User类，并通过Reflection调用了User.setName方法设置其name属性。对于这里的例子而言，出于简洁，我们将类名和方法名都以常量的方式硬编码。假设这些常量都是通过配置文件读入，那我们就实现了一个最简单的BeanWrapper。这个BeanWrapper的功能很简单，提供一个设置JavaBean属性的通用方法（Apache BeanUtils 类库中提供了大量针对Bean的辅助工具，如果有兴趣可以下载一份源码加以研读）。

Spring BeanWrapper基于同样的原理，提供了一个更加完善的实现。

看看如何通过Spring BeanWrapper操作一个JavaBean：

```
Object obj = Class.forName("net.xiaxin.beans.User").newInstance();

BeanWrapper bw = new BeanWrapperImpl(obj);
bw.setPropertyValue("name", "Erica");

System.out.println("User name=>"+bw.getPropertyValue("name"));
```

对比之前的代码，相信大家已经知道BeanWrapper的实现原理。

诚然，通过这样的方式设定Java Bean属性实在繁琐，但它却提供了一个通用的属性设定机制，而这样的机制，也正是Spring依赖注入机制所依赖的基础。

通过BeanWrapper，我们可以无需在编码时就指定JavaBean的实现类和属性值，通过在配置文件加以设定，就可以在运行期动态创建对象并设定其属性（依赖关系）。

上面的代码中，我们仅仅指定了需要设置的属性名“name”，运行期，BeanWrapper将根据Java Bean规范，动态调用对象的“setName”方法进行属性设定。属性名可包含层次，如对于属性名

"address.zipcode"，BeanWrapper会调用"getAddress().setZipcode"方法。

Bean Factory

Bean Factory，顾名思义，负责创建并维护Bean实例。

Bean Factory负责根据配置文件创建Bean实例，可以配置的项目有：

1. Bean属性值及依赖关系（对其他Bean的引用）
2. Bean创建模式（是否Singleton模式，即是否只针对指定类维持全局唯一的实例）
3. Bean初始化和销毁方法
4. Bean的依赖关系

下面是一个较为完整的Bean配置示例：

```
<beans>
    <description>Spring Bean Configuration Sample</description>

 <bean
          id="TheAction"                                    (1)
          class="net.xiaxin.spring.qs.UpperAction"          (2)
          singleton="true"                                  (3)
          init-method="init"                                (4)
          destroy-method="cleanup"                          (5)
          depends-on="ActionManager"                        (6)
 >

    <property name="message">
            <value>HeLLo</value>                            (7)
    </property>

    <property name="desc">
            <null/>
    </property>

    <property name="dataSource">
            <ref local="dataSource"/>                       (8)
    </property>
 </bean>

 <bean    id="dataSource"
          class="org.springframework.jndi.JndiObjectFactoryBean">

    <property name="jndiName">
            <value>java:comp/env/jdbc/sample</value>
    </property>
 </bean>
```

```
</beans>
```

(1) id

Java Bean在BeanFactory中的唯一标识，代码中通过BeanFactory获取JavaBean实例时需以此作为索引名称。

(2) class

Java Bean 类名

(3) singleton

指定此Java Bean是否采用单例（Singleton）模式，如果设为“true”，则在BeanFactory作用范围内，只维护此Java Bean的一个实例，代码通过BeanFactory获得此Java Bean实例的引用。反之，如果设为“false”，则通过BeanFactory获取此Java Bean实例时，BeanFactory每次都将创建一个新的实例返回。

(4) init-method

初始化方法，此方法将在BeanFactory创建JavaBean实例之后，在向应用层返回引用之前执行。一般用于一些资源的初始化工作。

(5) destroy-method

销毁方法。此方法将在BeanFactory销毁的时候执行，一般用于资源释放。

(6) depends-on

Bean依赖关系。一般情况下无需设定。Spring会根据情况组织各个依赖关系的构建工作（这里示例中的depends-on属性非必须）。

只有某些特殊情况下，如JavaBean中的某些静态变量需要进行初始化（这是一种Bad Smell，应该在设计上应该避免）。通过depends-on指定其依赖关系可保证在此Bean加载之前，首先对depends-on所指定的资源进行加载。

(7) <value>

通过<value/>节点可指定属性值。BeanFactory将自动根据Java Bean对应的属性类型加以匹配。

下面的”desc”属性提供了一个null值的设定示例。注意<value></value>代表一个空字符串，如果需要将属性值设定为null，必须使用<null/>节点。

(8) <ref>

指定了属性对BeanFactory中其他Bean的引用关系。示例中，TheAction的dataSource属性引用了id为dataSource的Bean。BeanFactory将在运行期创建dataSource bean实例，并将其引用传入TheAction Bean的dataSource属性。

下面的代码演示了如何通过BeanFactory获取Bean实例：

```
InputStream is = new FileInputStream("bean.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

```
Action action = (Action) factory.getBean("TheAction");
```

此时我们获得的Action实例，由BeanFactory进行加载，并根据配置文件进行了初始化和属性设定。

联合上面关于BeanWrapper的内容，我们可以看到，BeanWrapper实现了针对单个Bean的属性设定操作。而BeanFactory则是针对多个Bean的管理容器，根据给定的配置文件，BeanFactory从中读取类名、属性名/值，然后通过Reflection机制进行Bean加载和属性设定。

ApplicationContext

BeanFactory提供了针对Java Bean的管理功能，而ApplicationContext提供了一个更为框架化的实现（从上面的示例中可以看出，BeanFactory的使用方式更加类似一个API，而非Framework style）。

ApplicationContext覆盖了BeanFactory的所有功能，并提供了更多的特性。此外，ApplicationContext为与现有应用框架相整合，提供了更为开放式的实现（如对于Web应用，我们可以在web.xml中对ApplicationContext进行配置）。

相对BeanFactory而言，ApplicationContext提供了以下扩展功能：

1. 国际化支持
   我们可以在Beans.xml文件中，对程序中的语言信息（如提示信息）进行定义，将程序中的提示信息抽取到配置文件中加以定义，为我们进行应用的各语言版本转换提供了极大的灵活性。
2. 资源访问
   支持对文件和URL的访问。
3. 事件传播
   事件传播特性为系统中状态改变时的检测提供了良好支持。
4. 多实例加载
   可以在同一个应用中加载多个Context实例。

下面我们就这些特性逐一进行介绍。

1) 国际化支持
   国际化支持在实际开发中可能是最常用的特性。对于一个需要支持不同语言环境的应用而言，我们所采取的最常用的策略一般是通过一个独立的资源文件（如一个properties文件）完成所有语言信息（如界面上的提示信息）的配置，Spring对这种传统的方式进行了封装，并提供了更加强大的功能，如信息的自动装配以及热部署功能（配置文件修改后自动读取，而无需重新启动应用程序），下面是一个典型的示例：

```
<beans>
    <description>Spring Quick Start</description>
    <bean id="messageSource"
          class="org.springframework.context.support.ResourceB
undleMessageSource">
        <property name="basenames">
            <list>
                <value>messages</value>
            </list>
        </property>
    </bean>
</beans>
```

这里声明了一个名为messageSource的Bean（注意对于Message定义，Bean ID必须为messageSource，这是目前Spring的编码规约），对应类为ResourceBundleMessageSource，目前Spring中提供了两个MessageSource接口的实现，即ResourceBundleMessageSource和ReloadableResourceBundleMessageSource，后者提供了无需重启即可重新加载配置信息的特性。

在配置节点中，我们指定了一个配置名“messages”。Spring会自动在CLASSPATH根路

径中按照如下顺序搜寻配置文件并进行加载（以Locale为zh_CN为例）：

messages_zh_CN.properties

messages_zh.properties

messages.properties

messages_zh_CN.class

messages_zh.class

messages.class

（Spring实际上调用了JDK的ResourceBundle读取配置文件，相关内容请参见JDK文档）

示例中包含了两个配置文件，内容如下：

messages_zh_CN.properties:

```
userinfo=当前登录用户：[{0}] 登录时间:[{1}]
```

messages_en_US.properties:

```
userinfo=Current Login user: [{0}] Login time:[{1}]
```

我们可以通过下面的语句进行测试：

```
ApplicationContext ctx=new
          FileSystemXmlApplicationContext("bean.xml");

Object[] arg = new Object[]{
          "Erica",
          Calendar.getInstance().getTime()
          };
//以系统默认Locale加载信息(对于中文WinXP而言，默认为zh_CN)
String msg = ctx.getMessage("userinfo", arg);

System.out.println("Message is ===> "+msg);
```

代码中，我们将一个Object数组arg作为参数传递给ApplicationContext.getMessage方法，这个参数中包含了出现在最终文字信息中的可变内容，ApplicationContext将根据参数中的Locale信息对其进行处理（如针对不同Locale设定日期输出格式），并用其替换配置文件中的{n}标识（n代表参数数组中的索引，从1开始）。

运行上面的代码，得到以下输出的内容：

```
Message is ===> ¦Ì¡À?¡ã¦Ì???¨®??¡ì: [Erica] ¦Ì???¨°¡À??:[04-7-17 上午3:27]
```

乱码？回忆在传统方式下，针对ResourceBundle的编码过程中发生的问题。这是由于转码过程中产生的编码问题引发的。比较简单的解决办法是通过JDK提供的转码工具native2ascii.exe进行转换。

执行：

```
native2ascii messages_zh_CN.properties msg.txt
```

再用msg.txt文件替换Messages_zh_CN.properties文件。我们可以看到现在的Messages_zh_CN.properties变成了如下形式：

```
userinfo=\u5f53\u524d\u767b\u5f55\u7528\u6237: [{0}]
\u767b\u5f55\u65f6\u95f4:[{1}]
```

（通过在native2ascii命令后追加-reverse参数，可以将文件转回本地格式）

再次运行示例代码，得到正确输出：

```
Message is ===> 当前登录用户: [Erica] 登录时间:[04-7-17 上午3:34]
```

可见，根据当前默认Locale"zh_CN"，getMessage方法自动加载了messages_zh_CN.properties文件。

每次必须运行native2ascii方法比较繁琐，实际开发中，我们可以通过Apache Ant的Native2Ascii任务进行批量转码。如：

```
<native2ascii encoding="GBK" src="${src}" dest="${build}"/>
```

尝试在代码中指定不同的Locale参数：

```
String msg = ctx.getMessage("userinfo", arg, Locale.US);
```

再次运行，可以看到：

```
Message is ===> Current Login user: [Erica] Login time::[7/17/04 3:35 AM]
```

这里，getMessage方法根据指定编码"en_US"加载了messages_en_US.properties文件。同时请注意登录时间部分的变化（Locale不同，时间的输出格式也随之改变）。

getMessage方法还有一个无需Locale参数的版本，JVM会根据当前系统的Locale设定进行相应处理。可以通过在JVM启动参数中追加"-Duser.language=en"来设定当前JVM语言类型，通过JVM级的设定，结合国际化支持功能，我们可以较为简单的实现多国语言系统的自动部署切换。

2) 资源访问

ApplicationContext.getResource方法提供了对资源文件访问支持，如：

```
Resource rs = ctx.getResource("classpath:config.properties");
File file = rs.getFile();
```

上例从CLASSPATH根路径中查找config.properties文件并获取其文件句柄。

getResource方法的参数为一个资源访问地址，如：

```
file:C:/config.properties
/config.properties
classpath:config.properties
```

注意getResource返回的Resource并不一定实际存在，可以通过Resource.exists()方法对其进行判断。

3) 事件传播

ApplicationContext基于Observer模式（java.util包中有对应实现），提供了针对Bean的事件传播功能。通过Application. publishEvent方法，我们可以将事件通知系统内所有的ApplicationListener。

事件传播的一个典型应用是，当Bean中的操作发生异常（如数据库连接失败），则通过事件传播机制通知异常监听器进行处理。在笔者的一个项目中，就曾经借助事件机制，较好的实现了当系统异常时在监视终端上报警，同时发送报警SMS至管理员手机的功能。

在目前版本的Spring中，事件传播部分的设计还有待改进。同时，如果能进一步支持异步事件处理机制，无疑会更具吸引力。

下面是一个简单的示例，当LoginAction执行的时候，激发一个自定义消息"ActionEvent"，此ActionEvent将由ActionListener捕获，并将事件内容打印到控制台。

LoginActoin.java:

```
public class LoginAction implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public void setApplicationContext(
                    ApplicationContext applicationContext
                    )
            throws BeansException {
        this.applicationContext = applicationContext;
    }

    public int login(String username,String password) {
        ActionEvent event = new ActionEvent(username);
        this.applicationContext.publishEvent(event);
        return 0;
    }
}
```

ActionEvent.java:

```
public class ActionEvent extends ApplicationEvent {
  public ActionEvent(Object source) {
     super(source);
  }
}
```

ActionListener.java:

```
public class ActionListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent event) {
```

```
            if (event instanceof ActionEvent) {
                  System.out.println(event.toString());
            }
      }
}
```

配置非常简单：

```
<bean id="loginaction" class="net.xiaxin.beans.LoginAction"/>
<bean id="listener" class="net.xiaxin.beans.ActionListener"/>
```

运行测试代码：

```
ApplicationContext ctx=new
   FileSystemXmlApplicationContext("bean.xml");
LoginAction action = (LoginAction)ctx.getBean("action");
action.login("Erica","mypass");
```

可以看到控制台输出：

```
net.xiaxin.beans.LoginEvent[source=Erica]
```

org.springframework.context.event.ApplicationEventMulticasterImpl实现了事件传播机制，目前还相对简陋。

在运行期，ApplicationContext会自动在当前的所有Bean中寻找ApplicationListener接口的实现，并将其作为事件接收对象。当Application. publishEvent方法调用时，所有的ApplicationListener接口实现都会被激发，每个ApplicationListener可根据事件的类型判断是否是自己需要处理的事件，如上面的ActionListener只处理ActionEvent事件。

Web Context

上面的示例中，ApplicationContext均通过编码加载。对于Web应用，Spring提供了可配置的ApplicationContext加载机制。

加载器目前有两种选择：ContextLoaderListener和ContextLoaderServlet。这两者在功能上完全等同，只是一个是基于Servlet2.3版本中新引入的Listener接口实现，而另一个基于Servlet接口实现。开发中可根据目标Web容器的实际情况进行选择。

配置非常简单，在web.xml中增加：

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

或：

```
<servlet>
    <servlet-name>context</servlet-name>
    <servlet-class>
       org.springframework.web.context.ContextLoaderServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

通过以上配置，Web容器会自动加载/WEB-INF/applicationContext.xml初始化ApplicationContext实例，如果需要指定配置文件位置，可通过context-param加以指定：

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

配置完成之后，即可通过

WebApplicationContextUtils.getWebApplicationContext

方法在Web应用中获取ApplicationContext引用。

Spring 高级特性

Web 应用与 MVC

目前，基于 Web 的 MVC[1]框架在 J2EE 的世界内空前繁荣。TTS 网站上几乎每隔一两个星期就会有新的 MVC 框架发布。

目前比较好的 MVC,老牌的有 Struts、Webwork。新兴的 MVC 框架有 Spring MVC、Tapestry、JSF 等。这些大多是著名团队的作品，另外还有一些边缘团队的作品，也相当出色，如 Dinamica、VRaptor 等。

这些框架都提供了较好的层次分隔能力。在实现良好的 MVC 分隔的基础上，通过提供一些现成的辅助类库，同时也促进了生产效率的提高。

如何选择一个合适的框架？什么是考量一个框架设计是否优秀的标准？很难定夺的问题。旁观各大论坛中铺天盖地的论战，往往在这一点上更加迷茫。

从实际 Web 产品研发的角度而言（而非纯粹设计上，扩展性上，以及支持特性上的比较），目前 Struts 也许是第一选择。作为一个老牌 MVC Framework，Struts 拥有成熟的设计，同时，也拥有最丰富的信息资源和开发群体。换句实在点的话，产品基于 Struts 构建，如果公司发生人事变动，那么，找个熟悉 Struts 的替代程序员成本最低。

从较偏向设计的角度出发，WebWork2 的设计理念更加先进，其代码与 Servlet API 相分离，这使得单元测试更加便利，同时系统从 BS 结构转向 CS 接口也较为简单。另外，对于基于模板的表现层技术（Velocity、Freemarker 和 XSLT）的支持，也为程序员提供了除 JSP 之外的更多的选择（Struts 也支持基于模板的表现层技术，只是实际中不太常用）。

而对于 Spring 而言，首先，它提供了一个相当灵活和可扩展的 MVC 实现，与 WebWork2 相比，它在依赖注入方面、AOP 等方面更加优秀，但在 MVC 框架与底层构架的分离上又与 Webworks 存在着一定差距（Spring 的 MVC 与 Servlet API 相耦合，难于脱离 Servlet 容器独立运行，在这点的扩展性上，比 Webwork2 稍逊一筹）。

我们还要注意到，Spring 对于 Web 应用开发的支持，并非只限于框架中的 MVC 部分。即使不使用其中的 MVC 实现，我们也可以从其他组件，如事务控制、ORM 模板中得益。同时，Spring 也为其他框架提供了良好的支持，如我们很容易就可以将 Struts 与 Spring 甚至 WebWork 与 Spring 搭配使用（与 WebWork 的搭配可能有些尴尬，因为两者相互覆盖的内容较多，如 WebWork 中的依赖注入机制、AOP 机制等与 Spring 中的实现相重叠）。因此，对于 Spring 在 Web 应用中的作用，应该从一个更全面的角度出发。

下面我们就将针对 Spring MVC，以及 Spring+Struts、Spring+Webwork2 的 Web 应用模式进行介绍，通过这几种模式的比较，大家也可以作出自己的判断，寻找更加适合自己的方案组合。

[1] MVC 即 Model-View-Control 的缩写，为架构模式的一种，具体描述可通过 Google 获取J

Spring MVC

上面谈到了 Struts、Webwork 这些现有的 MVC 框架的特色。而 Spring MVC 在这些框架之中，处于怎样一个位置？在我们开始探讨这个新的 MVC Framework 之前，这是个值得思考的问题。

下面，先让我们来看看 Spring MVC 在具体应用中的表现，再有针对性的进行比较，以免形而上学的空谈。

Spring MVC指南

对于现有较成熟的 Model-View-Control(MVC)框架而言，其解决的主要问题无外乎下面几部分：

1. 将 Web 页面中的输入元素封装为一个（请求）数据对象。
2. 根据请求的不同，调度相应的逻辑处理单元，并将（请求）数据对象作为参数传入。
3. 逻辑处理单元完成运算后，返回一个结果数据对象。
4. 将结果数据对象中的数据与预先设计的表现层相融合并展现给用户。

各个 MVC 实现固然存在差异，但其中的关键流程大致如上。结合一个实例，我们来看看这几个关键流程在 Spring MVC 框架中的处理手法。

下面的实例，实现了一个常见的用户登录逻辑，即用户通过用户名和密码登录，系统对用户名和密码进行检测，如果正确，则在页面上显示几条通知信息。如果登录失败，则返回失败界面。

(示例中，表示层以 JSP2.0 实现。)

出于简洁考虑，这里的"用户名/密码"检测以及通知信息的生成均在代码中以硬编码实现。

首先来看登录界面：

登录

用户名：

密 码 ：

提交 重置

对应的 index.html：

```
<html>
<body>

<form method="POST" action="/login.do">
        <p align="center">登录</p>
```

```
            <br>

            用户名：
            <input type="text" name="username" >
            <br>

            密  码 ：
            <input type="password" name="password" >
            <br>

            <p>
                <input type="submit" value="提交" name="B1">
                <input type="reset"  value="重置" name="B2">
            </p>
</form>

</body>
</html>
```

很简单的一个登录界面，其中包含了一个用以输入用户名密码的 form，针对此 form 的提交将被发送到"/login.do"

MVC 关键流程的第一步，即收集页面输入参数，并转换为请求数据对象。这个静态页面提供了一个基本的输入界面，下面这些输入的数据将被发送至何处，将如何被转换为请求数据对象？

现在来看接下来发发生的事情：

当用户输入用户名密码提交之后，此请求被递交给 Web 服务器处理，上面我们设定 form 提交目标为"/login.do"，那么 Web 服务器将如何处理这个请求？

显然，标准 Http 协议中，并没有以.do 为后缀的服务资源，这是我们自己定义的一种请求匹配模式。此模式在 web.xml 中设定：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <servlet>                                                    (1)
        <servlet-name>Dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>

```

```
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/Config.xml</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>                                          (2)
        <servlet-name>Dispatcher</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

</web-app>
```

(1) Servlet 定义

这里我们定义了请求分发 Servlet，即：

org.springframework.web.servlet.DispatcherServlet

DispatcherServlet 是 Spring MVC 中负责请求调度的核心引擎，所有的请求将由此 Servlet 根据配置分发至各个逻辑处理单元。其内部同时也维护了一个 ApplicationContext 实例。

我们在**<init-param>**节点中配置了名为“**contextConfigLocation**”的 Servlet 参数，此参数指定了 Spring 配置文件的位置“**/WEB-INF/Config.xml**”。如果忽略此设定，则默认为“**/WEB-INF/<servlet name>-servlet.xml**”，其中**<servlet name>**以 Servlet 名替换（在当前环境下，默认值也就是“**/WEB-INF/Dispatcher-servlet.xml**）。

(2) **请求映射**

我们将所有以.do 结尾的请求交给 Spring MVC 进行处理。当然，也可以设为其他值，如.action、.action 等。

通过以上设定，Web 服务器将把登录界面提交的请求转交给 Dispatcher 处理，Dispatcher 将提取请求（HttpServletRequest）中的输入数据，分发给对应的处理单元，各单元处理完毕后，将输出页面返回给 Web 服务器，再由 Web 服务器返回给用户浏览器。

Dispatcher 根据什么分发这些请求？显然，我们还需要一个配置文件加以设定。这也就是上面提及的 Config.xml，此文件包含了所有的“请求/处理单元”关系映射设定，以及返回时表现层的一些属性设置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
```

```
    <!--Definition of View Resolver -->
    <bean id="viewResolver"                                              (1)
         class="org.springframework.web.servlet.view.InternalResou
rceViewResolver">
        <property name="viewClass">                                      (2)
            <value>
               org.springframework.web.servlet.view.JstlView
            </value>
        </property>

        <property name="prefix">                                         (3)
            <value>
               /WEB-INF/view/
            </value>
        </property>

        <property name="suffix">                                         (4)
            <value>.jsp</value>
        </property>
    </bean>

    <!--Request Mapping -->
    <bean id="urlMapping"                                                (5)
          class="org.springframework.web.servlet.handler.SimpleUr
lHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/login.do">LoginAction</prop>
            </props>
        </property>
    </bean>

    <!---Action Definition-->
    <bean id="LoginAction"                                               (6)
          class="net.xiaxin.action.LoginAction">
        <property name="commandClass">                                   (7)
            <value>net.xiaxin.action.LoginInfo</value>
        </property>

        <property name="fail_view">                                      (8)
            <value>loginfail</value>
        </property>

        <property name="success_view">
```

```
            <value>main</value>
        </property>
    </bean>

</beans>
```

(1) Resolver 设定
Resolver 将把输出结果与输出界面相融合，为表现层提供呈现资源。

(2) View Resolver 的 viewClass 参数
这里我们使用 JSP 页面作为输出，因此，设定为：
org.springframework.web.servlet.view.JstlView
其余可选的 viewClass 还有：
- **org.springframework.web.servlet.view.freemarker.FreeMarkerView**（用于基于 FreeMarker 模板的表现层实现）
- **org.springframework.web.servlet.view.velocity.VelocityView**（用于基于 velocity 模板的表现层实现）

等。

(3)(4)View Resolver 的 prefix 和 suffix 参数
指定了表现层资源的前缀和后缀，运行时，Spring 将为指定的表现层资源自动追加前缀和后缀，以形成一个完整的资源路径。另参见(8)

(5) "请求/处理单元"关系映射
可以看到，这里我们将"/login.do"请求映射到处理单元 LoginAction。
<props>节点下可以有多个映射关系存在，目前我们只定义了一个。

(6) LoginAction 定义
这里定义了逻辑处理单元 LoginAction 的具体实现，这里，LoginAction 的实现类为 net.xiaxin.action.LoginAction。

(7) LoginAction 的请求数据对象
commandClass 参数源于 LoginAction 的基类 BaseCommandController，BaseCommandControlle 包含了请求数据封装和验证方法（BaseCommandController.bindAndValidate），它将根据传入的 HttpServletRequest 构造请求数据对象。

这里我们指定 commandClass 为 **net.xiaxin.action.LoginInfo**，这是一个非常简单的 Java Bean，它封装了登录请求所需的数据内容：

```
public class LoginInfo {
  private String username;
```

```
  private String password;

  public String getPassword() {
      return password;
  }

  public void setPassword(String password) {
      this.password = password;
  }

  public String getUsername() {
      return username;
  }

  public void setUsername(String username) {
      this.username = username;
  }
}
```

Spring 会根据 LoginAction 的 commandClass 定义自动加载对应的 LoginInfo 实例。
之后，对 Http 请求中的参数进行遍历，并查找 LoginInfo 对象中是否存在与之同名的属性，如果找到，则将此参数值复制到 LoginInfo 对象的同名属性中.

请求数据转换完成之后，我们得到了一个封装了所有请求参数的 Java 对象，并将此对象作为输入参数传递给 LoginAction。

(8) 返回视图定义
对于这里的 LoginAction 而言，有两种返回结果，即登录失败时返回错误界面，登录成功时进入系统主界面。
对应我们配置了 fail_view、success_view 两个自定义参数。
参数值将由 Resolver 进行处理，为其加上前缀后缀，如对于 fail_view 而言，实际的视图路径为 **/WEB-INF/view/loginfail.jsp**。
之后，Resolver 会将 LoginAction 的返回数据与视图相融合，返回最终的显示界面。

业务逻辑处理单元:

LoginAction.java:

```
public class LoginAction extends SimpleFormController {

   private String fail_view;
```

```
    private String success_view;

    protected ModelAndView onSubmit(                                    (1)
            Object cmd,
            BindException ex
        )throws Exception {

        LoginInfo loginInfo = (LoginInfo) cmd;                         (2)

        if (login(loginInfo) == 0) {

            HashMap result_map = new HashMap();
            result_map.put("logininfo", loginInfo);

            List msgList = new LinkedList();
            msgList.add("msg1");
            msgList.add("msg2");
            msgList.add("msg3");
            result_map.put("messages", msgList);

            return new
                ModelAndView(this.getSuccess_view(), result_map); (3)
        } else {
            return new ModelAndView(this.getFail_view());
        }
    }

    private int login(LoginInfo loginInfo) {

        if ("Erica".equalsIgnoreCase(loginInfo.getUsername())
                && "mypass".equals(loginInfo.getPassword())) {
            return 0;
        }
        return 1;
    }

    public String getFail_view() {
        return fail_view;
    }

    public String getSuccess_view() {
        return success_view;
    }
```

```
    public void setFail_view(String string) {
        fail_view = string;
    }

    public void setSuccess_view(String string) {
        success_view = string;
    }

}
```

其中：

(1) onSubmit 方法

我们在子类中覆盖了父类的 onSubmit 方法；而 onSubmit 方法用于处理业务请求。负责数据封装和请求分发的 Dispatcher，将对传入的 HttpServletRequest 进行封装，形成请求数据对象，之后根据配置文件，调用对应业务逻辑类的入口方法（这里就是 LoginAction）的 onSubmit()方法，并将请求数据对象及其他相关资源引用传入。

```
protected ModelAndView onSubmit(
            Object cmd,
            BindException ex
        )
```

onSubmit 方法包含了两个参数：Object cmd 和 BindException ex。
前面曾经多次提到请求数据对象，这个名为 cmd 的 Object 型参数，正是传入的请求数据对象的引用。

BindException ex 参数则提供了数据绑定错误的跟踪机制。它作为错误描述工具，用于向上层反馈错误信息。
在 Spring MVC 中，BindException 将被向上层表现层反馈，以便在表现层统一处理异常情况（如显示对应的错误提示），这一机制稍后在"输入参数合法性校验"部分再具体探讨。

onSubmit 还有另外一个签名版本：

```
protected ModelAndView onSubmit(
            HttpServletRequest request,
            HttpServletResponse response,
            Object cmd,
            BindException ex
        )
```

可以看到，类似 Servlet 的 doGet/doPost 方法，此版本的 onSubmit 方法签名中包含了 Servlet 规范中的 HttpServletRequest、HttpServletResponse 以提供与 Web 服务器的交互功能（如 Session 的访问）。此参数类型的 onSubmit 方法的调用优先级较高。也就是说，如果我们在子类中同时覆盖了这两个不同参数的

onSubmit 方法，那么只有此版本的方法被执行，另一个将被忽略。

⑵ 将输入的请求数据对象强制转型为预定义的请求对象类型。

⑶ 返回处理结果
ModelAndView 类包含了逻辑单元返回的结果数据集和表现层信息。ModelAndView 本身起到关系保存的作用。它将被传递给 Dispatcher，由 Dispatcher 根据其中保存的结果数据集和表现层设定合成最后的界面。
这里我们用到了两种签名版本的 ModelAndView 构造方法：

Ø public ModelAndView(String viewname)
返回界面无需通过结果数据集进行填充。

Ø public ModelAndView(String viewname, Map model)
返回界面由指定的结果数据集加以填充。可以看到，结果数据集采用了 Map 接口实现的数据类型。其中包含了返回结果中的各个数据单元。关于结果数据集在界面中的填充操作，可参见下面关于返回界面的描述。

上面这两个版本的构造子中，通过 viewname 指定了表示层资源。

另外，我们也可以通过传递 View 对象指定表示层资源。
Ø public ModelAndView(View view)
Ø public ModelAndView(View view, Map model)
我们可以结合 RedirectView 完成转向功能，如：

```
return new ModelAndView(
                new RedirectView("/redirected.jsp"
                ));
```

当然，我们也可以在带有 HttpServletRequest 参数的 onSubmit 方法实现中，通过 HttpServletRequest/HttpServletResponse 完成 forward/redirect 功能，这两种途径可以达到同样的效果。

最后，来看返回界面：
错误返回界面 **loginfail.jsp** 只是个纯 html 文件（为了与 View Resolver 中设定的后缀相匹配，因此以.jsp 作为文件后缀），这里就不再浪费篇幅。

再看成功登录后的页面 main.jsp：
界面显示效果如下：

```
Login Success!!!

Current User: erica

Your current messages:

* msg1
msg2
* msg3
```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>

<html>
  <body>
      <p>Login Success!!!</p>

      <p>Current User:
          <c:out value="${logininfo.username}"/><br>
      </p>

      <p>Your current messages:</p>
      <c:forEach items="${messages}"
                  var="item"
                  begin="0"
                  end="9"
                  step="1"
                  varStatus="var">

          <c:if test="${var.index % 2 == 0}">
              *
          </c:if>

          ${item}<br>
      </c:forEach>

  </body>
</html>
```

页面逻辑非常简单，首先显示当前登录用户的用户名。然后循环显示当前用户的通知消息“messages”。如果当前循环索引为奇数，则在消息前追加一个“*”号（这个小特性在这里似乎有些多余，但却为不熟悉 JSTL 的读者提供了如何使用 JSTL Core taglib 进行循环和逻辑判断的样例）。

实际上这只是个普通 JSTL/JSP 页面，并没有任何特殊之处，如果说有些值得研究的技术，也就是其中引用的 JSTL Core Taglib

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
```

上面这句话申明了页面中所引用的 taglib，指定其前缀为“c”，也就是说，在页面中，所有以“c”为前缀，形同<c:xxxx>的节点都表明是此 taglib 的引用，在这里，也就是对 JSTL Core Lib 的引用。

这里需要注意的是，笔者所采用的 Web 容器为 Tomcat 5(支持 Servlet 2.4/JSP2.0 规范)以及 Apache JSTL 2.0(http://jakarta.apache.org/taglibs/index.html)。

```
<c:out value="${logininfo.username}"/>
```

<c:out>将 value 中的内容输出到当前位置，这里也就是把 logininfo 对象的 username 属性值输出到页面当前位置。

${……}是 JSP2.0 中的 Expression Language（EL）的语法。它定义了一个表达式，其中的表达式可以是一个常量（如上），也可以是一个具体的表达语句（如 **forEach** 循环体中的情况）。典型案例如下：

- **${logininfo.username}**

  这表明引用 logininfo 对象的 username 属性。我们可以通过“.”操作符引用对象的属性，也可以用“[]”引用对象属性，如**${logininfo[username]}**与**${logininfo.username}**达到了同样的效果。

  “[]”引用方式的意义在于，如果属性名中出现了特殊字符，如“.”或者“-”，此时就必须使用“[]”获取属性值以避免语法上的冲突（系统开发时应尽量避免这一现象的出现）。

  与之等同的 JSP Script 大致如下：

  ```
  LoginInfo logininfo =
              (LoginInfo)session.getAttribute("logininfo");
  String username = logininfo.getUsername();
  ```

  可以看到，EL 大大节省了编码量。

  这里引出的另外一个问题就是，EL 将从哪里找到 logininfo 对象，对于**${logininfo.username}**这样的表达式而言，首先会从当前页面中寻找之前是否定义了变量 logininfo，如果没有找到则依次到 Request、Session、Application 范围内寻找，直到找到为止。如果直到最后依然没有找到匹配的变量，则返回 null.

  如果我们需要指定变量的寻找范围，可以在 EL 表达式中指定搜寻范围：

  ${pageScope.logininfo.username}
  ${requestScope.logininfo.username}
  ${sessionScope.logininfo.username}

${applicationScope.logininfo.username}
在 Spring 中，所有逻辑处理单元返回的结果数据，都将作为 Attribute 被放置到 HttpServletRequest 对象中返回（具体实现可参见 Spring 源码中 org.springframework.web.servlet.view.InternalResourceView.exposeModelAsRequestAttributes 方法的实现代码），也就是说 Spring MVC 中，结果数据对象默认都是 **requestScope**。因此，在 Spring MVC 中，以下寻址方法应慎用：
${sessionScope.logininfo.username}
${applicationScope.logininfo.username}

- ${1+2}
  结果为表达式计算结果，即整数值 3。

- ${i>1}
  如果变量值 i>1 的话，将返回 bool 类型 true。与上例比较，可以发现 EL 会自动根据表达式计算结果返回不同的数据类型。

  表达式的写法与 java 代码中的表达式编写方式大致相同。

```
<c:forEach items="${messages}"
                var="item"
                begin="0"
                end="9"
                step="1"
                varStatus="var">
        ……
</c:forEach>
```

上面这段代码的意思是针对 messages 对象进行循环，循环中的每个循环项的引用变量为 **item**，循环范围是从 0 到 9，每次步长为 1。而 **varStatus** 则定义了一个循环状态变量 **var**，循环状态变量中保存了循环进行时的状态信息，包括以下几个属性：

属性	类型	说明
index	int	当前循环索引号
count	int	成员总数
first	boolean	当前成员是否首位成员
last	boolean	当前成员是否末尾成员

再看：

```
<c:if test="${var.index % 2 == 0}">
 *
</c:if>
```

这段代码演示了判定 Tag **<c:if>**的使用方法。可以看到，其 test 属性指明了判定条件，

判定条件一般为一个 EL 表达式。

<c:if>并没有提供else子句,使用的时候可能有些不便,此时我们可以通过<c:choose> tag 来达到类似的目的:

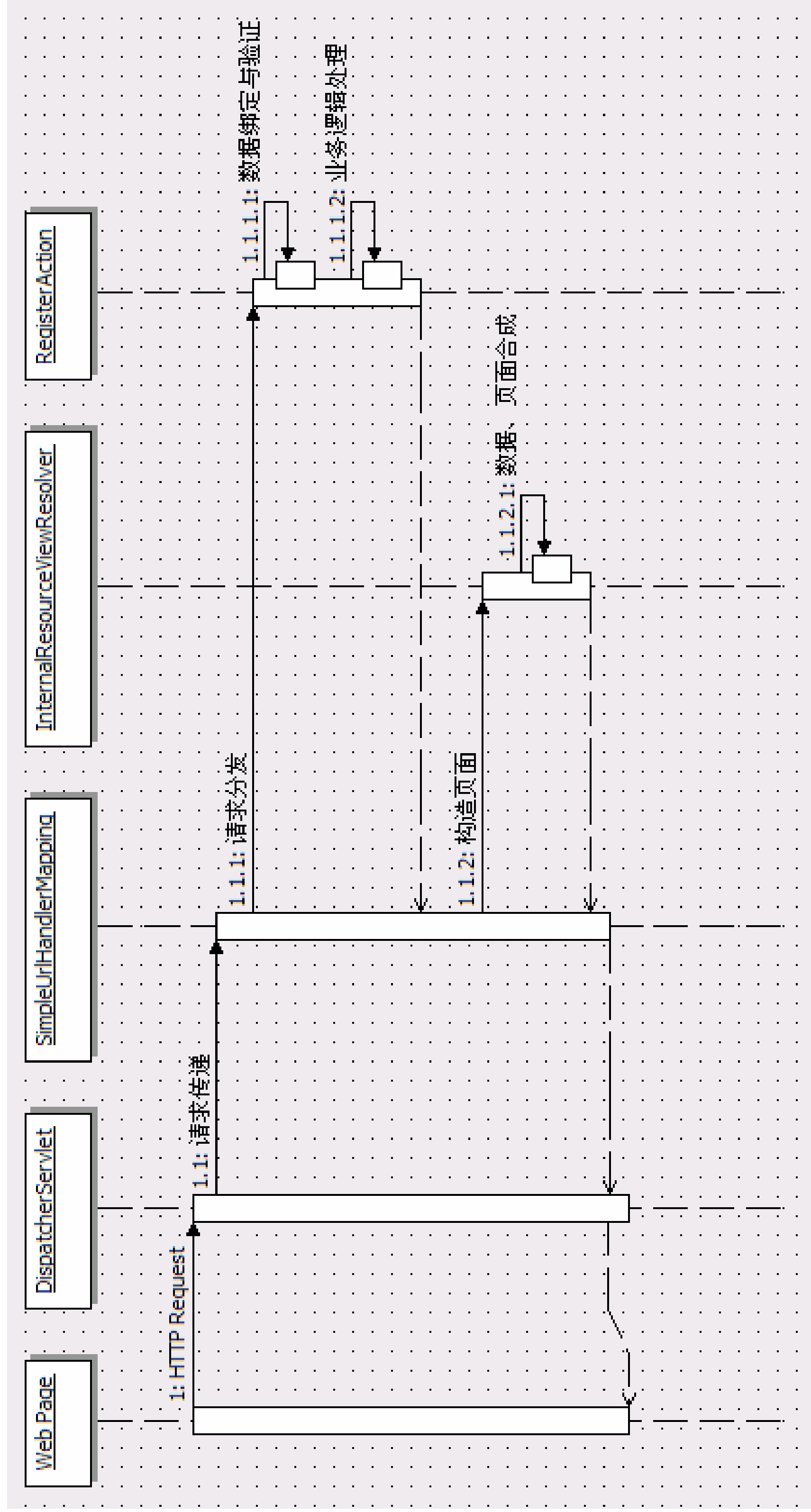
```
<c:choose>
    <c:when test="${var.index % 2 == 0}">
    *
    </c:when>
    <c:otherwise>
    !
    </c:otherwise>
</c:choose>
```

类似 Java 中的 switch 语句，<c:choose>提供了复杂判定条件下的简化处理手法。其中 <c:when>子句类似 case 子句，可以出现多次。上面的代码，在奇数行时输出“*”号，而偶数行时输出“!”。

通过<c:choose>改造后的输出页面:

```
Login Success!!!

Current User: erica

Your current messages:

* msg1
! msg2
* msg3
```

至此，一个典型的请求/响应过程结束。通过这个过程，我们也了解了 Spring MVC 的核心实现机制。对其进行总结，得到以下 UML 序列图:


Web Page
DispatcherServlet
SimpleUrlHandlerMapping
InternalResourceViewResolver
RegisterAction
1: HTTP Request
1.1: 请求传递
1.1.1: 请求分发
1.1.1.1: 数据绑定与验证
1.1.1.2: 业务逻辑处理
1.1.2: 构造页面
1.1.2.1: 数据、页面合成

基于模板的Web表示层技术

传统的 JSP 技术为 Web 表现层技术提供了灵活、丰富的功能支持。然而，站在工程的角度而言，过于凌乱的 JSP Script 也成为系统维护的头号大敌。

JSP 代码几乎等同于 Java 代码，在提供了最丰富的特性支持的同时，也为系统的开发带来一些隐患，程序员往往天马行空，不为羁束，在 JSP 中将业务逻辑、数据逻辑、表现逻辑代码相混杂，代码重用性、系统可维护性极低。特别是在参与开发人员众多，技术水平良莠不齐的情况下，纵使技术经理一再强调设计规范的约束，但人本的约束总是难以控制，随着开发过程进展和产品上线压力的增大，规范约束逐渐薄弱，于是难以避免的造成代码的混乱，可维护性的下降。

面对这个问题，众多组织和厂商开始研发自己的表现层框架，试图通过一个隔离的表现层框架，强行将表现层和逻辑层相剥离。时间似乎退回到了最初 Web 端只支持 Servlet 技术的时代（那时候或多或少各个公司都有自己的模板实现）。不过，现在的模板技术经过长时间的发展，已经将表现层的能力发挥得淋漓尽致，不失为 JSP 技术之外的一个明智选择。

模板技术相对传统 JSP 技术有以下三个主要优势：

1. 在技术层面，将表现逻辑与业务逻辑相分离。
2. 为人员之间的分工提供了一个良好的分界点。页面美工只需专著关心模板的设计，而程序员则专注于业务逻辑的实现。二者重合点明显减少。
3. 如果需要，模板引擎可脱离 Web 容器单独运行，这为系统可能的移植需求提供了更多的弹性空间（这一特性在应用中也许并不会有太大的实际意义，只是提供了一种附加选择）。

目前 Spring 支持一下几种模板技术：

1. XSLT

XSLT 是基于 XML 的表现层模板技术，伴随着 XML 的大量使用。XSLT 也日渐成熟，并迅速成为主流表现层技术之一。XSLT 作为一个通用表现层框架，拥有最好的平台适应性，几乎所有的主流程序设计语言都提供了 XLST 支持，现有的 XLST 模板可以简单的移植到不同的语言平台，如将 J2EE 应用中的 XSLT 移植到.net 平台，这样的可移植性是其他专用模板技术，如 Velocity 和 Freemarker 难以达到的。

笔者在 2001 年在一个原型项目中采用了 XSLT 作为表现层实现，由于当时 XSLT 尚不成熟，XSLT 解析器效率低下，因此在正式产品开发中使用其他技术作为替代。在 2003 年中，经过技术探讨，决定再次在项目实施中引入 XSLT 技术，相对两年前，此时的 XSLT 技术已经相当成熟，解析器的效率也大大改善。经过半年时间的项目研发，产品上线，并取得了令人满意的表现。不过，在之后的项目回顾过程中，笔者认为，目前在项目中大量采用 XSLT 技术尚不可取，上述项目开发过程中，XSLT 技术提供了极佳的扩展性和重用性，也保证了业务逻辑和表示逻辑的清晰划分，然而，最大的问题是，XSLT 缺乏强有力的编辑器支持。虽然通过 XML/XSLT 技术成全了设计上近乎完美的表现，但却为界面开发带来了极大难度，以至后期复杂界面的修改都需要消耗极大的人力，得不偿失。

笔者在项目开发中所用的 XSLT 编辑器为 StylusStudio 和 XmlSpy，目前这两款编辑器可以算是 XSLT 开发的首选，提供了丰富的特性和可视化编辑功能。但即便如此，XLST 繁杂苛刻的语法和调试上的难度也为开发工作带来了极大的障碍。

此外，也许是最重要的一点，xslt 在性能上的表现尚不尽如人意。经过多年的发展，XSLT 解析/合成器的性能相对最初已经大为改观，但依然与其他模板技术存在着较大差距。据实地测试，FreeMarker 和 Velocity 对于同等复杂度的表现层逻辑，平均处理速度是

XSLT 的 10 倍以上，这是一个不得不正视的性能沟壑。同时，XSLT 的内存占用也是 FreeMarker 和 Velocity 的数倍有余（XSLT 中，每个节点都是一个 Java 对象，大量对象的存储对内存占用极大，同时大量对象的频繁创建和销毁也对 JVM 垃圾收集产生了较大负面影响）。在上述项目中，由于硬件上的充分冗余（8G RAM, 4CPU），才使得这些性能上的影响相对微弱。

因此，目前在项目中大量引入 XSLT 技术尚需仔细考量。

2. Velocity

   Velocity 是 Apache Jakarta 项目中的一个子项目，它提供了丰富强大的模板功能。作为目前最为成熟的模板支持实现，Velocity 在诸多项目中得到了广泛应用，不仅限于 Web 开发，在众多代码生成系统中，我们也可以看到 Velocity 的身影（如 Hibernate 中的代码生成工具）。

3. FreeMarker

   FreeMarker 是 Velocity 之外的另一个模板组件。

   与 Velocity 相比，FreeMarker 对表现逻辑和业务逻辑的划分更为严格，Freemarker 在模板中不允许对 Servlet API 进行直接操作（而 Velocity 可以），如 FreeMarker 中禁止对 HttpServletRequest 对象直接访问（但可以访问 HttpServletRequest 对象中的 Attribute）。通过更加严格的隔离机制，牵涉逻辑处理的操作被强制转移到逻辑层。从而完全保证了层次之间的清晰性。

   另外一个 Velocity 无法实现的特性，也是最具备实际意义的特性：FreeMarker 对 JSP Tag 提供了良好支持。这一点可能存在一点争议，JSP 技术的最大问题就是容易在页面中混入逻辑代码。而 FreeMarker 对 JSP Tag 的支持似乎为这个问题又打开了大门。这一点上，我们可以将 FreeMarker 看作是仅允许使用 TAG 的 JSP 页面（实际上，FreeMarker 的表达式语法与 EL 语法也非常类似）。

   从开发角度而言，只允许使用 TAG 的 JSP 页面，已经在很大程度上保证了页面表现逻辑与业务逻辑的分离。程序员在 JSP Script 中混杂逻辑代码的原因，大部分是出于慵懒，只要无法在页面模板中直接编写 Java 代码，相信程序员也不会去专门编写一个 JSP TAG 来刻意违反层次划分原则。

   对 JSP TAG 的支持为 FreeMarker 带来了极大的活力，目前开源社区中已经有了为数众多的成熟 Taglib，如 DisplayTag、Struts Menu 等，这些功能丰富，成熟可靠的 Taglib，将为产品开发提供极大的便利。另一方面，这也为代码重用提供了另一个可选途径，避免了大部分模板实现在这一点上的不足。

   就笔者的经验，对于 Web 开发而言，FreeMarker 在生产效率和学习成本上更具优势，而 Velocity 的相对优势在于更多第三方工具的支持和更广泛的开发和用户团体（然而对于一个轻量级模板类库而言，这样的优势并不是十分明显）。

   如果没有 Velocity 的技术储备，而又需要通过技术上的限定解决视图/模型的划分问

题，这里推荐采用 FreeMarker 作为 Spring MVC 中的表现层实现。以获得最好的（学习、开发）成本受益。

下面，我们将对之前的 JSP DEMO 进行改造，以展示基于 FreeMarker 的 Spring MVC 应用技术。至于 XSLT 和 Velocity，Spring 官方文档中已经有了很全面的介绍，这里暂且掠过，如果有兴趣，可以自行参阅 Spring Reference。（笔者完成此文时 Spring 版本为 1.0.2，在 Spring 1.1 之后，官方文档中也增添了 FreeMarker 的相关介绍）

对于上面例子中的几个文件，需要进行改编的是 Config.xml 和登录后的主页面（main.jsp，改编后为 main.ftl，后缀名 ftl 是 freemarker 模板的习惯命名方式）。

改编后的 Config.xml

```
<?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="viewResolver"                                          (1)
          class="org.springframework.web.servlet.view.freemarker.
FreeMarkerViewResolver">
        <property name="viewClass">
            <value>
          org.springframework.web.servlet.view.freemarker.FreeMar
kerView
          </value>
        </property>
        <property name="cache"><value>false</value></property>
        <property name="suffix"><value>.ftl</value></property>
    </bean>

    <bean id="freemarkerConfig"                                      (2)
          class="org.springframework.web.servlet.view.freemarker.
FreeMarkerConfigurer">
        <property name="templateLoaderPath">
            <value>WEB-INF/view/</value>
        </property>
    </bean>

    <!---Action Definition-->
    <bean id="LoginAction" class="net.xiaxin.action.LoginAction">
        <property name="commandClass">
          <value>net.xiaxin.action.LoginInfo</value>
        </property>
        <property name="fail_view">
          <value>loginfail</value></property>
        <property name="success_view">
```

```
            <value>main</value>
        </property>
    </bean>

    <!--Request Mapping -->
    <bean id="urlMapping"
            class="org.springframework.web.servlet.handler.SimpleUr
lHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/login.do">LoginAction</prop>
            </props>
        </property>
    </bean>

</beans>
```

可以看到，配置文件进行了局部修改：

(1) 由于我们采用 FreeMarker 作为表现层技术。原有的 viewResolver 的定义发生了变化。

viewResolver class 被修改为：

org.springframework……FreeMarkerConfigurer

view 属性被修改为：

org.springframework……FreeMarkerView

suffix 属性被修改为 FreeMarker 模板文件默认后缀“.ftl”。

prefix 属性被取消，因为下面的 freemarkerConfig 中已经定义了模板路径。

(2) 增加了 freemarkerConfig 定义，并通过 templateLoaderPath 属性设定了模板文件的存放路径（相对 Web Application 根目录）。

模板文件 main.ftl：

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Current User: ${logininfo.username}</h1>                    (1)

    <#list messages as msg>                                       (2)
        <#if msg_index % 2=0>                                     (3)
            *
        <#else>
            !
        </#if>
        ${msg}<br>
```

```
    </#list>

</body>
</html>
```

(1) 对数据对象logininfo的username属性的引用。

(2) 针对Collection型数据messages的循环，并通过as子句指定迭代变量msg，msg引用了messages中的当前循环项目。

(3) if else语句
这里的msg_index为当前循环索引号。“_index”是FreeMarker中对于循环索引变量的命名约定。通过“迭代变量名_index”即可访问当前循环索引。
类似的循环状态访问约定还有“迭代变量名_has_next”，可通过这个循环状态属性判断是否还有后继循环。

此模板文件最终运行效果与之前的main.jsp页面大致相同：

Current User: erica!

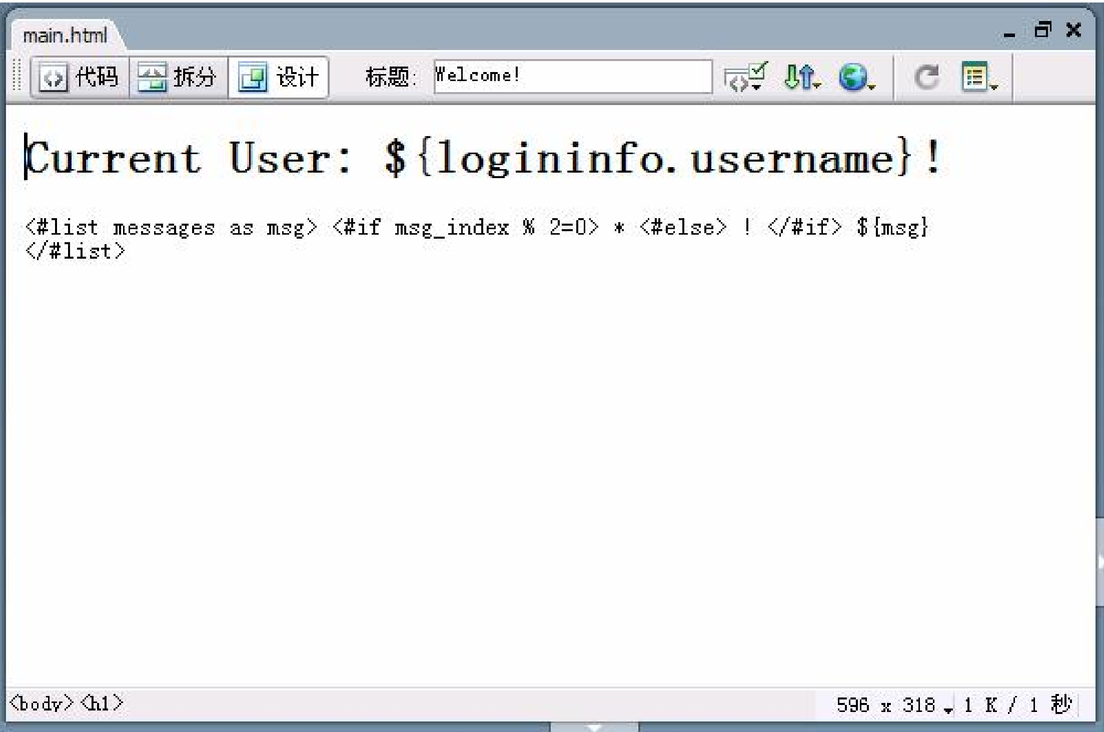
* msg1
! msg2
* msg3

Web应用中模板技术与JSP技术的对比

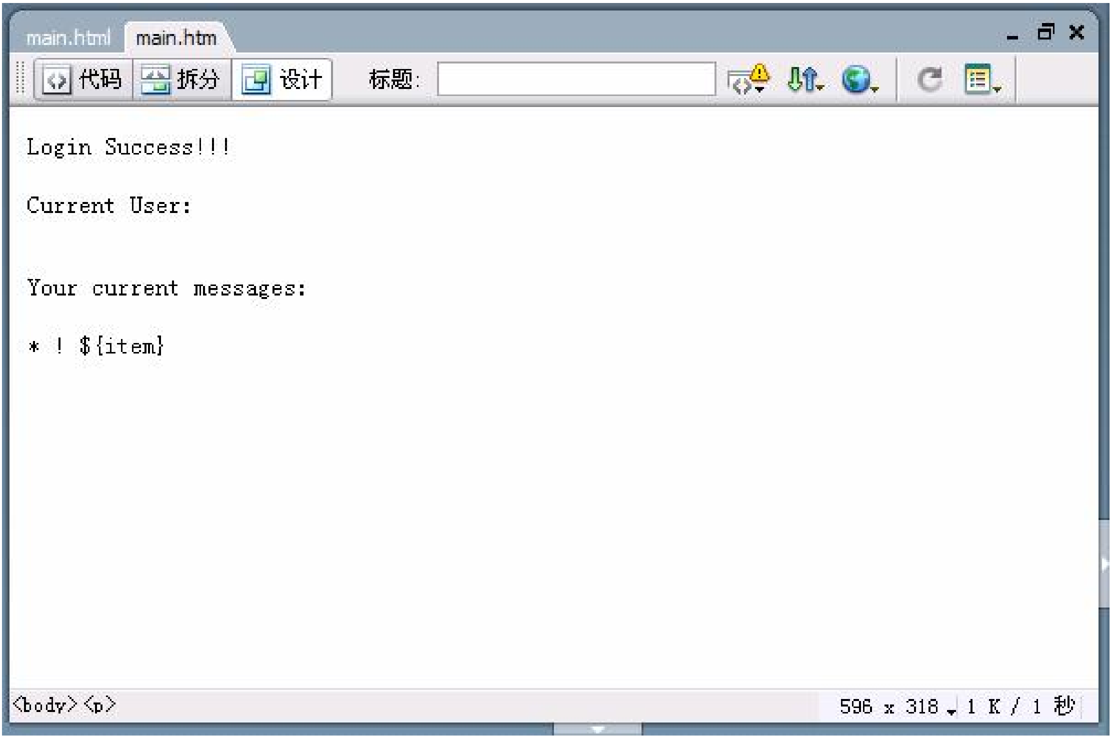
通过上面的例子可以看到，FreeMarker模板文件与之前使用JSTL Core Taglib实现的JSP页面非常相似。

实际上，仅采用JSTL Core Taglib/EL 的JSP页面，也可以理解为基于JSP技术的模板实现，甚至在可视化编辑器中，仅采用JSTL Core Taglib/EL 的JSP页面的可视化编辑效果更好。下面是main.ftl和main.jsp在DreamWeaver可视化编辑环境中的表现：

main.ftl：



main.jsp:



可以看到，main.jsp的可视化编辑效果明显更接近于实际运行效果。这对于表现层的设计而言显得别具意义。

由此，再重申一点，模板技术最大的功用在于强制开发人员将Java代码排除在表现层之外（同时也将页面美工和程序员的工作范围清晰划分），而对于具体表现层设计的帮助倒未必突出。

如果项目参与人员具备良好的技术素养和协同精神，并且在技术上允许（目前很多Web容器还不支持JSP2.0及其EL语法），笔者建议还是采用以JSTL Core为基础，以及有限的、与业务逻辑无关的自定义Tag（或第三方Taglib）的JSP页面作为表现层解决方案。毕竟我们的目标是最低的学习/开发成本，最高的生产率。

输入验证与数据绑定

Spring Framework提供了强大的输入验证和数据绑定功能。结合输入验证器和<spring:bind>tag，传统繁杂混乱的输入校验功能将变得更加清晰简单。

同样，下面先结合一个实例来看Spring输入验证和数据绑定机制的具体验证。

实例目标：实现用户注册功能。

流程：

1. 提供一个界面供用户输入注册信息，下面是一个简化的注册界面，仅提供了用户名和密码的设置

用户注册

用户名：（必须大于等于4个字符）

密码：（必须大于等于6个字符）

重复密码：

提交 重置

2. 如果用户注册信息有误，显示错误界面，要求用户检查输入后重新注册。

错误：用户名长度必须大于等于4个字母！

错误：两次输入的密码不一致！

用户注册

用户名：kkk

错误：用户名长度必须大于等于4个字母！

密码：••••••

重复密码：••••••

错误：两次输入的密码不一致！

提交 重置

3. 注册成功，显示操作成功提示。

Kevin 注册成功！

实例内容

a）配置文件

在这个实例中，我们选用JSTLView作为我们的表现层实现。对应的配置文件如下。

Config.xml:

```
<beans>

  <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalRes
ourceViewResolver">
      <property name="viewClass">
          <value>
              org.springframework.web.servlet.view.JstlView
          </value>
      </property>

      <property name="prefix">
          <value>/WEB-INF/view/</value>
      </property>

      <property name="suffix">
          <value>.jsp</value>
      </property>
  </bean>

  <bean id="RegisterValidator"                                   (1)
        class="net.xiaxin.validator.RegisterValidator"/>

  <bean id="RegisterAction"
        class="net.xiaxin.action.RegisterAction">
      <property name="commandClass">
          <value>net.xiaxin.reqbean.RegisterInfo</value>
      </property>

      <property name="validator">                                (2)
          <ref local="RegisterValidator"/>
      </property>

      <property name="formView">                                 (3)
          <value>register</value>
      </property>

      <property name="successView">                              (4)
          <value>RegisterSuccess</value>
      </property>
```

```
  </bean>


   <!--Request Mapping -->
  <bean id="urlMapping"
         class="org.springframework.web.servlet.handler.SimpleUr
lHandlerMapping">
      <property name="mappings">
          <props>
              <prop key="/register.do">RegisterAction</prop>
          </props>
      </property>
  </bean>

</beans>
```

这个配置文件与篇首MVC介绍中所用实例大同小异。不同之处在于我们在这里引入了数据验证配置节点：

⑴ 配置了一个数据验证Bean：RegisterValidator
  net.xiaxin.validator.RegisterValidator

⑵ 为逻辑处理单元RegisterAction定义输入数据校验Bean
  这里通过一个Bean引用，将RegisterValidator配置为本Action的数据校验类。

⑶ 指定本处理单元的显示界面。
  formView是RegisterAction的父类SimpleFormController中定义的属性，指定了本处理单元的显示界面。
  这里即用户访问register.do时将显示的注册界面。

  要注意的是，完成此界面后，我们必须通过"…/register.do"访问注册界面，而不是"…/register.jsp"，因为我们必须首先借助Spring完成一系列初始化工作（如创建对应的状态对象并与之关联）之后，register.jsp才能顺利执行，否则我们会得到一个应用服务器内部错误。

⑷ 指定成功返回界面。
  successView同样是RegisterAction的父类SimpleFormController中定义的属性，它指向成功返回界面。

b) 数据验证类

在Spring中，所有的数据验证类都必须实现接口：
org.springframework.validation.Validator

Validator接口定义了两个方法：

- boolean supports(Class clazz);
  用于检查当前输入的数据类型是否符合本类的检验范围。Spring调用Validator实现类时，首先会通过这个方法检查数据类型是否与此Validator相匹配。

- void validate(Object obj, Errors errors);
  数据校验方法。Validator实现类通过实现这个方法，完成具体的数据校验逻辑。

RegisterValidator.java:

```
public class RegisterValidator implements Validator {

  public boolean supports(Class clazz) {
      return RegisterInfo.class.isAssignableFrom(clazz);        (1)
  }

  public void validate(Object obj, Errors errors) {
      RegisterInfo regInfo = (RegisterInfo) obj;                 (2)
      //检查注册用户名是否合法
      if (regInfo.getUsername().length() < 4) {
          errors.rejectValue("username",                         (3)
                  "less4chars",
                  null,
                  "用户名长度必须大于等于4个字母！");
      }

      /*检查用户名是否已经存在
      if (UserDAO.getUser(regInfo.getUsername()) != null) {
          errors.rejectValue("username",
                  "existed",
                  null,
                  "用户已存在！");
      }
      */

      if (regInfo.getPassword1().length() < 6) {
          errors.rejectValue("password1",
                  "less6chars",
                  null,
                  "密码长度必须大于等于6个字母");
      }

      if (!regInfo.getPassword2().equals(regInfo.getPassword1()))
```

```
        {
            errors.rejectValue("password2",
                    "notsame",
                    null,
                    "两次输入的密码不一致！");
        }
    }

}
```

(1) RegisterInfo.class.isAssignableFrom方法用于判定参数类别，当传入Class对象与当前类类别相同，或是当前类的父类（或当前类实现的接口）时返回真。这里我们将其用于对校验对象的数据类型进行判定（这里的判定条件为：校验对象必须是RegisterInfo类的实例）。

(2) RegisterInfo regInfo = (RegisterInfo) obj;
将输入的数据对象转换为我们预定的数据类型。

(3) 通过rejectValue方法将错误信息加入Error列表，此错误信息将被页面捕获并显示在错误提示界面上。

rejectVlaue方法有4个参数:

1. Error Code
   显示错误时，将根据错误代码识别错误信息类型。
2. Message Key
   上面关于ApplicationContext的国际化支持时，我们曾经谈及MessageSource的使用，这里我们可以通过引入MessageSource实现提示信息的参数化，此时，本参数将用作.properties文件中的消息索引。
3. Error Argument
   如果提示信息中需要包含动态信息，则可通过此参数传递需要的动态信息对象。具体参见ApplicationContext中关于国际化实现的描述。
4. Default Message
   如果在当前MessageSource中没有发现Message Key对应的信息数据，则以此默认值返回。
   这里我们暂时尚未考虑国际化支持，所有的信息都将通过Default Message返回。关于国际化支持请参见稍后章节。

另外rejectValue还有另外几个简化版本，可根据情况选用。

c) 注册界面

register.jsp提供了注册操作界面。它同时提供了最初的注册界面，当输入参数非法时，同时也会显示错误信息，提示用户检查输入。

register.jsp:

```
<!-- 页面中使用了JSTL Core taglib 和Spring lib-->
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
```

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<!-- 设定页面编译时采用gb2312编码，同时指定浏览器显示时采取gb2312解码-->
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>

<html>

    <head>
        <title>用户注册</title>
    </head>

    <body style="text-align: center">

        <form method="POST" action="/register.do">

            <spring:bind path="command.*">
                <font color="#FF0000">
                    <c:forEach
                        items="${status.errorMessages}"
                        var="error">
                        错误：<c:out value="${error}"/><br>
                    </c:forEach>
                </font>
            </spring:bind>

            <table border="0" width="450" height="101"
                        cellspacing="0" cellpadding="0" >
                <tr>
                    <td height="27" width="408" colspan="2">
                        <p align="center"><b>用户注册</b></td>
                </tr>


                <tr>
                    <td height="23" width="104">用户名：</td>
                    <td height="23" width="450">
                        <spring:bind path="command.username">
                            <input
                            type="text"
                            value="<c:out value="${status.value}"/>"
                            name="<c:out value="${status.expression}"/>"
                            >
                            （必须大于等于4个字符）
                            <br>
```

```
                    <c:if test="${status.error}">
                        <font color="#FF0000">
                            错误:
                            <c:forEach
                                items="${status.errorMessages}"
                                var="error">
                                <c:out value="${error}"/>
                            </c:forEach>
                        </font>
                    </c:if>
                </spring:bind>
            </td>
            </td>

        </tr>

    <tr>
        <td height="23" width="104">密码: </td>
        <td height="23" width="450">
            <spring:bind path="command.password1">
                <input
                type="password"
                value="<c:out value="${status.value}"/>"
                name="<c:out value="${status.expression}"/>"
                >
                (必须大于等于6个字符)
                <br>
                <c:if test="${status.error}">
                    <font color="#FF0000">
                        错误:
                        <c:forEach
                                items="${status.errorMessages}"
                                var="error">
                            <c:out value="${error}"/>
                        </c:forEach>
                    </font>
                </c:if>
            </spring:bind>
        </td>
    </tr>
    <tr>
        <td height="23" width="104">重复密码: </td>
        <td height="23" width="450">
            <spring:bind path="command.password2">
```

```
                    <input
                    type="password"
                    value="<c:out value="${status.value}"/>"
                    name="<c:out value="${status.expression}"/>"
                    >
                    <br>
                    <c:if test="${status.error}">
                        <font color="#FF0000">
                            错误:
                            <c:forEach
                                items="${status.errorMessages}"
                                var="error">
                                <c:out value="${error}"/>
                            </c:forEach>
                        </font>
                    </c:if>
                </spring:bind>
            </td>
        </tr>

        </table>
    <p>
        <input type="submit" value="提交" name="B1">
        <input type="reset" value="重置" name="B2">
    </p>
    </form>
</body>

</html>
```

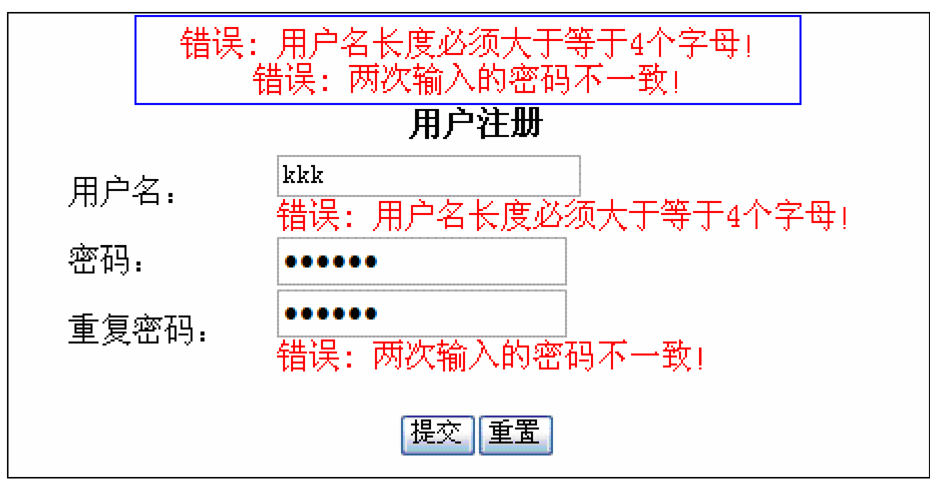
页面起始部分指定了页面中引入的taglib和页面编码方式，实际开发时应该将其独立到一个单独的jsp文件中，并在各个jsp文件中include便于统一维护。

页面中关键所在，也就是<spring:bind> 标记的使用：

```
<spring:bind path="command.*">
    <font color="#FF0000">
        <c:forEach
            items="${status.errorMessages}"
            var="error">
            错误: <c:out value="${error}"/><br>
        </c:forEach>
    </font>
</spring:bind>
```

spring.bind标记通过path参数与CommandClass对象相绑定。之后我们就可以对绑定的

CommandClass对象的状态信息进行访问。上面的片断中，我们通过通配符"*"将当前spring.bind语义与command对象的所有属性相绑定，用于集中的错误信息显示，对应最终提示界面中的（蓝框标注部分）：



而在下面每个输入框下方，我们也提供了对应的错误提示，此时我们绑定到了特定的command属性，如**"command.username"**。

这里的**"command"**是Spring中的默认CommandClass名称，用于引用当前页面对应的CommandClass实例（当前语境下，也就是**net.xiaxin.reqbean.RegisterInfo**）。我们也可以配置CommandClass引用名称，在Config.xml中RegisterAction配置中增加CommandName配置，如下：

```
<bean id="RegisterAction"
        class="net.xiaxin.action.RegisterAction">
      <property name="commandName">
          <value>RegisterInfo</value>
      </property>
      .........
</bean>
```

之后我们就可以在页面中使用"RegisterInfo"替代现在的"command"对数据对象进行引用。

（为了保持前后一致，下面我们仍旧以"command"为例）

绑定到username属性的**<spring:bind>**标记：

```
<spring:bind path="command.username">
  <input
      type="text"
      value="<c:out value="${status.value}"/>"
      name="<c:out value="${status.expression}"/>"
  >
  （必须大于等于4个字符）
  <br>
```

```
      <c:if test="${status.error}">
          <font color="#FF0000">
          错误:
          <c:forEach
              items="${status.errorMessages}"
              var="error">
              <c:out value="${error}"/>
          </c:forEach>
          </font>
      </c:if>
</spring:bind>
```

可以看到，<spring:bind>语义内，可以通过${status.*}访问对应的状态属性。
${status.*} 对应的实际是类

org.springframework.web.servlet.support.BindStatus

BindStatus类提供了与当前CommandClass对象绑定的状态信息，如：

${status.errorMessages}对应绑定对象属性的错误信息。

${status.expression}对应绑定对象属性的名称。

${status.value}对应绑定对象属性当前值。

具体描述可参见BindStatus类的Java Doc 文档。

下面是RegisterAction.java和成功返回界面RegisterSuccess.jsp，出于演示目的，这两个文件都非常简单：

RegisterAction.java:

```
public class RegisterAction extends SimpleFormController {

  protected ModelAndView onSubmit(Object cmd, BindException ex)
          throws Exception {

      Map rsMap = new HashMap();
      rsMap.put("logininfo",cmd);

      return new ModelAndView(this.getSuccessView(),rsMap);
  }
}
```

RegisterSuccess.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>

<html>
  <body>
      <p align="center">
```

```
        <c:out value="${logininfo.username}"/> 注册成功!
      </p>
  </body>
</html>
```

可以看到，结合JSTL Core Taglib和Spring Taglib，我们实现了一个拥有数据校验功能的注册界面。界面显示、数据校验、逻辑处理三大模块被清晰隔离互不干扰，相对传统的jsp解决方案。系统的可维护性得到了大大提升。

不过，我们还必须注意到，spring:bind标记对界面代码的侵入性较大，可以看到页面中混杂了大量的Tag调用，这将对界面的修改和维护带来一定的困难。相对WebWork2而言，Spring在这方面还是显得有些繁琐。

异常处理

Web应用中对于异常的处理方式与其他形式的应用并没有太大的不同——通过try/catch语句针对不同的异常进行相应处理。

但是在具体实现中，由于异常层次、种类繁杂，我们往往很难在Servlet、JSP层妥善的处理好所有异常情况，代码中大量的try/catch代码显得尤为凌乱。

我们通常面对下面两个主要问题：

1. 对异常实现集中式处理

   典型情况：对数据库异常记录错误日志。一般处理方法无外两种，一是在各处数据库访问代码的异常处理中，加上日志记录语句。二是将在数据访问代码中将异常向上抛出，并在上层结构中进行集中的日志记录处理。

   第一种处理方法失之繁琐、并且导致系统难以维护，假设后继需求为“对于数据库异常，需记录日志，并发送通知消息告知系统管理员”。我们不得不对分散在系统中的各处代码进行整改，工作量庞大。

   第二种处理方法实现了统一的异常处理，但如果缺乏设计，往往使得上层异常处理过于复杂。

   这里，我们需要的是一个设计清晰、成熟可靠的集中式异常处理方案。

2. 对未捕获异常的处理

   对于Unchecked Exception而言，由于代码不强制捕获，往往被程序员所忽略，如果运行期产生了Unchecked Exception，而代码中又没有进行相应的捕获和处理，则我们可能不得不面对尴尬的500服务器内部错误提示页面。

   这里，我们需要一个全面而有效的异常处理机制。

上面这两个问题，从技术角度上而言并算不上什么大的难点。套用一些短平快的设计模式，我们也能进行处理并获得不错的效果。同时，目前大多数服务器也都支持在Web.xml中通过<error-page>（Websphere/Weblogic）或者<error-code>(Tomcat)节点配置特定异常情况的显示页面。

Spring MVC中提供了一个通用的异常处理机制，它提供了一个成熟的，简洁清晰的异常处理方案。如果基于Spring MVC开发Web应用，那么利用这套现成的机制进行异常处理也更加自然和有效。

Spring MVC中的异常处理：

以前面的注册系统为例，首先，在Dispatcher配置文件Config.xml中增加id为“exceptionResolver”的bean定义：

```
<bean id="exceptionResolver"
    class="org.springframework.web.servlet.handler.SimpleMappingEx
ceptionResolver">
    <property name="defaultErrorView">
```

```
            <value>failure</value>
        </property>

        <property name="exceptionMappings">
            <props>
                <prop key="java.sql.SQLException">showDBError</prop>
                <prop key="java.lang.RuntimeException">showError</prop>
            </props>
        </property>
</bean>
```

通过SimpleMappingExceptionResolver我们可以将不同的异常映射到不同的jsp页面（通过exceptionMappings属性的配置），同时我们也可以为所有的异常指定一个默认的异常提示页面（通过defaultErrorView属性的配置），如果所抛出的异常在exceptionMappings中没有对应的映射，则Spring将用此默认配置显示异常信息（注意这里配置的异常显示界面均仅包括主文件名，至于文件路径和后缀已经在viewResolver中指定）。

一个典型的异常显示页面如下：

```
<html>
<head><title>Exception!</title></head>
<body>
<% Exception ex = (Exception)request.getAttribute("Exception"); %>
<H2>Exception: <% ex.getMessage();%></H2>
<P/>
<% ex.printStackTrace(new java.io.PrintWriter(out)); %>
</body>
</html>
```

如果SimpleMappingExceptionResolver无法满足异常处理的需要，我们可以针对HandlerExceptionResolver接口实现自己异常处理类，这同样非常简单（只需要实现一个resolveException方法）。

国际化支持

回忆之前章节中关于 ApplicationContext 国际化支持的讨论，可以发现，如果在我们的 Web 应用中结合 ApplicationContext 的国际化支持功能，就可以轻松实现 Web 应用在不同语言环境中的切换，这对项目的可移植性（特别对于涉外项目）带来了极大的提升。

得益于 Spring 良好的整体规划，在 Web 应用中实现国际化支持非常简单。下面我们就围绕这个专题进行一些探讨。

首先，在配置文件 Config.xml 中增加如下节点：

```
<bean id="messageSource"
   class="org.springframework.context.support.ResourceBundleMessageSo
urce">
    <property name="basename"><value>messages</value></property>
</bean>
```

非常简单，上面我们设定了一个 messageSource 资源，并指定资源文件基名为"messages"。关于 messageSource 的介绍，请参见前面 ApplicationContext 国际化支持中的内容。

在/WEB-INF/classes/目录下，新建两个 properties 文件：

1） messages_zh_CN.properties
2） messages_en_US.properties

内容如下：

messages_zh_CN.properties

```
less4chars=用户名长度必须大于4个字符!
less6chars=密码长度必须大于6个字符!
notsame=两次密码输入不一致!
register_title=用户注册
username_label=用户名
password_label=密码
rep_pass_label=重复密码
error_msg=错误:
```

（注意messages_zh_CN.properties文件在部署时候须使用JDK工具native2ascii转码，具体请参见ApplicationContext章节中国际化支持部分内容）

messages_en_US.properties:

```
less4chars=User name length must greater than 4 chars!
less6chars=Password length must greater than 6 chars!
notsame=Tow passwords are not consistent!
register_title=User Register
username_label=Username
password_label=Password
rep_pass_label=Repeat password
error_msg=Error:
```

经过以上配置，“用户注册”实例的输入数据验证类RegisterValidator就可以自动使用message_*.properties中的配置数据，如：

```
errors.rejectValue("username",
                   "less4chars",
                   null,
                   "用户名长度必须大于等于4个字母！");
```

会自动从当前的messageSource中寻找"less4chars"对应的键值。如果当前Locale为“en_US”[2]，则界面上的错误信息将显示为“User name length must greater than 4 chars!”。（rejectValue方法总是先从当前messageSource配置中寻找符合目前Locale配置的信息，如果找不到，才会返回参数中的默认描述信息。具体请参见“输入验证与数据绑定”一节中的描述）

错误信息已经完成了国际化，那么，界面上的提示信息如何处理？

我们需要对原本的register.jsp文件进行一些改造，通过<spring:message>标记将其中硬编码的提示信息替换为动态Tag。如对于页面上方的“用户注册”标题，我们可做如下修改：

```
……
<tr>
  <td height="27" width="408" colspan="2">
        <p align="center">
        <!--用户注册-->
        <spring:message code="register_title"/>
    </td>
</tr>
……
```

<spring:message>标记会自动从当前messageSource中根据code读取符合目前Locale设置的配置数据。此时如果当前Locale为“en_US”，则原界面上方标题“用户注册”将显示为messages_en_US.properties中配置的“User Register”。

Locale的切换

实际操作中，针对不同语言要求进行切换的方式大多有以下几种：

1. 根据语言种类，部署时手动替换*.properties文件
2. 根据当前系统Locale设置，自动匹配（如上例）
3. 根据客户浏览器语言设定，自动切换界面语种。

前两种部署方式依赖于部署时的设定和系统环境。第三种无疑最为灵活，这意味着系统一旦部署，无需更改即可自动根据客户浏览器的语言设定自动切换语言种类。

Spring中目前提供了以下几种语言自动切换机制的实现（均实现了LocaleResolver接口）：

- AcceptHeaderLocaleResolver
  根据浏览器Http Header中的accept-language域判定(accept-language域中一般包含了当前操作系统的语言设定，可通过HttpServletRequest.getLocale方法获得此域的内容)。
- SessionLocaleResolver

[2]Windows 可以通过控制面板中的“区域和语言选项”快速切换系统 Locale 设定，Linux 可通过 export LANG=zh_CN; LC_ALL=zh_CN.GBK 命令修改当前 Locale。

根据用户本次会话过程中的语言设定决定语言种类（如：用户登录时选择语言种类，则此次登录周期内统一使用此语言设定）。

Ø CookieLocaleResolver

根据Cookie判定用户的语言设定（Cookie中保存着用户前一次的语言设定参数）。

使用也非常简单，对于AcceptHeaderLocaleResolver而言，只需在配置文件(Config.xml）中增加如下节点：

```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.AcceptHeaderLocale
Resolver">
</bean>
```

之后，Spring就会根据客户端浏览器的Locale设定决定返回界面所采用的语言种类。可通过AcceptHeaderLocaleResolver.resolveLocale方法获得当前语言设定。

resolveLocale和setLocale方法是LocaleResolver接口定义的方法，用于提供对Locale信息的存取功能。而对于AcceptHeaderLocaleResolver，由于客户机操作系统的Locale为只读，所以仅提供了resolveLocale方法。

SessionLocaleResolver的配置类似与上例类似：

```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.SessionLocaleResol
ver">
</bean>
```

SessionLocaleResolver会自动在Session中维护一个名为“org.springframework.web.servlet.i18n.SessionLocaleResolver.LOCALE”的属性，其中保存了当前会话所用的语言设定信息，同时对外提供了resolveLocale和setLocale两个方法用于当前Locale信息的存取操作。

CookieLocaleResolver配置则包含了三个属性，cookieName、cookiePath 和cookieMaxAge，分别指定了用于保存Locale设定的Cookie的名称、路径和最大保存时间。

```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
>
   <property name="cookieName">
         <value>browserLocale</value>
   </property>

   <property name="cookiePath">
         <value>mypath</value>
   </property>

   <property name="cookieMaxAge">
         <value>999999</value>
```

```
    </property>
</bean>
```

这几个属性并非必须配置，其默认值为分别为：

- cookieName
  org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE
- cookiePath
  /
- cookieMaxAge
  Integer.MAX_VALUE [2147483647]

之后，我们即可通过：CookieLocaleResolver.setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale)方法保存Locale设定，setLocale方法会自动根据设定在客户端浏览器创建Cookie并保存Locale信息。这样，下次客户机浏览器登录的时候，系统就可以自动利用Cookie中保存的Locale信息为用户提供特定语种的操作界面。

另外，Spring还提供了对语言切换事件的捕获机制（LocaleChangeInterceptor），由于实际开发中使用较少，就不多做介绍，具体可参见Spring-Reference。

WebWork2 & Spring

很长一段时间内， OpenSymphony 作为一个开源组织，其光辉始终被 Apache 所掩盖。Java 程序员热衷于 Apache 组织 Struts 项目研讨之后，往往朦朦胧胧的感觉到，似乎还有另外一个框架正在默默的发展。

这种朦胧的感觉，则可能来自曾经在国内流行一时的论坛软件－Jive Forum。

很多软件技术人员不惜从各种渠道得到 Jive 的源代码，甚至是将其全部反编译以探其究竟。作为一个论坛软件能受到技术人员如此垂青，想必作者睡梦中也会乐醒。J

而 WebWork，就是 Jive 中，MVC 实现的核心[3]。

这里我们所谈及的 WebWork，实际上是 Webwork+XWork 的总集，Webwork1.x 版本中，整个框架采用了紧耦合的设计（类似 Struts），而 2.0 之后，Webwork 被拆分为两个部分，即 Webwork 2.x +XWork 1.x，设计上的改良带来了系统灵活性上的极大提升。这一点我们稍后讨论。

下面的章节中，我们将就 Webwork 框架与 Spring 的整合进行探讨。考虑到目前市面上尚无关于 Webwork2 的官方出版物，其间也将 WebWork 应用技术贯穿其间加以介绍（也许应该起草一份新的 OpenDoc 了J）。

另：Webwork 发行包中的文档并不是很全面，如果开发中遇到什么问题，登录 Webwork Wiki 站点查看在线文档是个不错的选择：

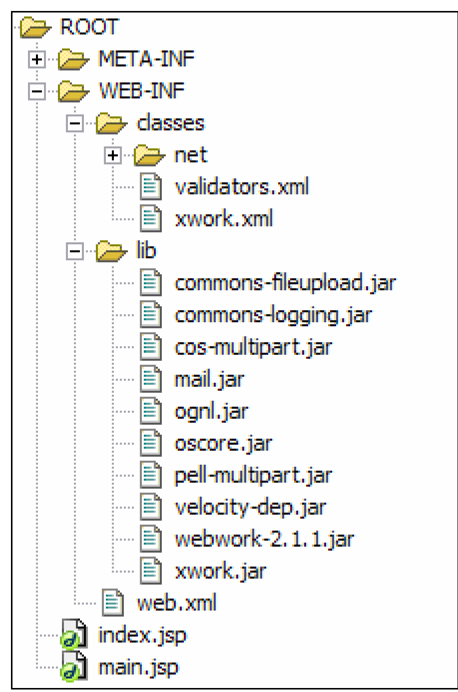
http://www.opensymphony.com/webwork/wikidocs

[3] Jive 对 WebWork 的源代码进行了重新封装，主要是包结构上的变化，如 com.opensymphony.webwork 在 Jive 中被修改为 com.jivesoftware.webwork，核心功能并没有太大改变

Quick Start

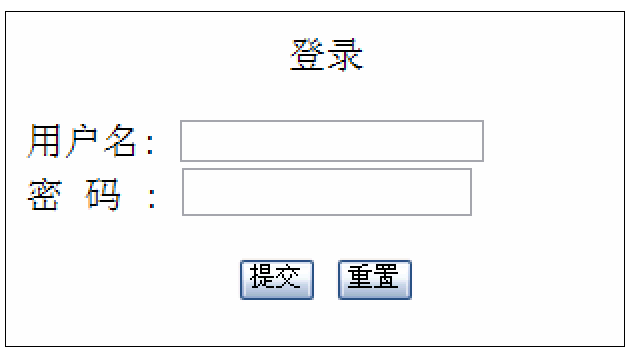
准备工作：首先下载 WebWork2 的最新版本(http://www.opensymphony.com/webwork/)。

WebWork2 发行包中的\lib\core 目录下包含了 WebWork2 用到的核心类库。将\webwork-2.1.1.jar 以及\lib\core*.jar 复制到 Web 应用的 WEB-INF\lib 目录。本例的部署结构如图所示：



这里我们选择了一个最常见的登录流程来演示 Webwork2 的工作流程。虽然简单，但足以明理。

页面流程如下：



登录成功页面，与之前的 Spring MVC 的介绍类似，显示几条通知消息：

Login Success!

Welcome! erica

Messages:
!message1
*message2
!message3

登录失败页面：

只是一个简单不过的登录流程，然而在这个简单的例子里，贯穿了 Webwork 框架的大部分应用技术。在介绍 Spring MVC 的时候，曾经将一个 MVC 框架的运作流程拆分为以下几部分加以讨论：

1. 将 Web 页面中的输入元素封装为一个（请求）数据对象。
2. 根据请求的不同，调度相应的逻辑处理单元，并将（请求）数据对象作为参数传入。
3. 逻辑处理单元完成运算后，返回一个结果数据对象。
4. 将结果数据对象中的数据与预先设计的表现层相融合并展现给用户。

对于 Webwork，我们以同样的顺序加以探讨。
首先来看登录界面：
index.jsp

```
<html>
<body>
<form action="/login.action">

    <p align="center">
        登录<br>
    </p>

    用户名：
```

```
    <input type="text" name="model.username" />
    <br>

    密  码 :
    <input type="password" name="model.password" />
    <br>

    <p align="center">
            <input type="submit" value="提交" name="B1"/>
            <input type="reset" value="重置" name="B2"/>
    </p>

</form>
</body>
</html>
```

这里的 index.jsp 实际上是由纯 html 组成，非常简单，其中包含一个表单：

```
    <form action="/login.action">
```

这表明其提交对象为/login.action

表单中同时包含两个文本输入框，

```
    <input type="text" name="model.username" />
    <input type="password" name="model.password" />
```

可以看到，两个输入框的名称均以"model"开头，这是因为在这里我们采用了 WebWork 中 Model-Driven 的 Action 驱动模式。这一点稍后再做介绍。

当表单被提交之时，浏览器会以两个文本框的值作为参数，向 Web 请求以/login.action 命名的服务。

标准 HTTP 协议中并没有.action 结尾的服务资源。我们需要在 web.xml 中加以设定：

```
……
<servlet>
    <servlet-name>webwork</servlet-name>
        <servlet-class>
          com.opensymphony.webwork.dispatcher.ServletDispatcher
       </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>webwork</servlet-name>
```

```
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>
……
```

此后，所有以.action 结尾的服务请求将由 ServletDispatcher 接管。

ServletDispatcher 接受到 Servlet Container 传递过来的请求，将进行一下几个动作：

1. 从请求的服务名（/login.action）中解析出对应的 Action 名称（login）
2. 遍历 HttpServletRequest、HttpSession、ServletContext 中的数据，并将其复制到 Webwork 的 Map 实现中，至此之后，所有数据操作均在此 Map 结构中进行，从而将内部结构与 Servlet API 相分离。

   至此，Webwork 的工作阶段结束，数据将传递给 XWork 进行下一步处理。从这里也可以看到 Webwork 和 xwork 之间的切分点，Webwork 为 xwork 提供了一个面向 Servlet 的协议转换器，将 Servlet 相关的数据转构转换成 xwork 所需要的通用数据格式，而 xwork 将完成实际的服务调度和功能实现。
   这样一来，以 xwork 为核心，只需替换外围的协议转换组件，即可实现不同技术平台之间的切换（如将面向 Servlet 的 Webwork 替换为面向 JMS 的协议转换器实现，即可在保留应用逻辑实现的情况下，实现不同外部技术平台之间的移植）。

3. 以上述信息作为参数，调用 ActionProxyFactory 创建对应的 ActionProxy 实例。ActionProxyFactory 将根据 Xwork 配置文件（xwork.xml）中的设定，创建 ActionProxy 实例，ActionProxy 中包含了 Action 的配置信息（包括 Action 名称，对应实现类等等）。

4. ActionProxy 创建对应的 Action 实例，并根据配置进行一系列的处理程序。包括执行相应的预处理程序（如通过 Interceptor 将 Map 中的请求数据转换为 Action 所需要的 Java 输入数据对象等），以及对 Action 运行结果进行后处理。ActionInvocation 是这一过程的调度者。而 com.opensymphony.xwork.DefaultActionInvocation 则是 XWork 中对 ActionInvocation 接口的标准实现，如果有精力可以对此类进行仔细研读，掌握了这里面的玄机，相信 XWork 的引擎就不再神秘。

下面我们来看配置文件：

xwork.xml：

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-1.0.dtd">

<xwork>
    <include file="webwork-default.xml" />                    (1)
    <package name="default" extends="webwork-default">        (2)

        <action name="login"                                  (3)
            class="net.xiaxin.webwork.action.LoginAction">
```

```
            <result name="success" type="dispatcher">                    (4)
                <param name="location">/main.jsp</param>
            </result>

            <result name="loginfail" type="dispatcher">
                <param name="location">/index.jsp</param>
            </result>

            <interceptor-ref name="params" />                           (5)
            <interceptor-ref name="model-driven"/>                      (6)

        </action>

    </package>
</xwork>
```

(1) include

通过 include 节点，我们可以将其他配置文件导入到默认配置文件 xwork.xml 中。从而实现良好的配置划分。

这里我们导入了 Webwork 提供的默认配置 webwork-default.xml（位于 webwork.jar 的根路径）。

(2) package

XWork 中，可以通过 package 对 action 进行分组。类似 Java 中 package 和 class 的关系。为可能出现的同名 Action 提供了命名空间上的隔离。

同时，package 还支持继承关系。在这里的定义中，我们可以看到：

```
extends="webwork-default"
```

"webwork-default"是 webwork-default.xml 文件中定义的 package，这里通过继承，"default" package 自动拥有"webwork-default" package 中的所有定义关系。

这个特性为我们的配置带来了极大便利。在实际开发过程中，我们可以根据自身的应用特点，定义相应的 package 模板，并在各个项目中加以重用，无需再在重复繁琐的配置过程中消耗精力和时间。

此外，我们还可以在 Package 节点中指定 namespace，将我们的 action 分为若干个逻辑区间。如：

```
<package name="default" namespace="/user"
                extends="webwork-default">
```

就将此 package 中的 action 定义划归为/user 区间，之后在页面调用 action 的时候，我们需要用/user/login.action 作为 form action 的属性值。其中的/user/就指定了此 action 的 namespace，通过这样的机制，我们可以将系统内的 action 进行逻辑分类，从而使得各模块之间的划分更加清晰。

(3) action

Action 配置节点，这里可以设定 Action 的名称和对应实现类。

⑷ result

通过 result 节点，可以定义 Action 返回语义，即根据返回值，决定处理模式以及响应界面。

这里，返回值"success"（Action 调用返回值为 String 类型）对应的处理模式为"dispatcher"。

可选的处理模式还有：

1. dispatcher
   本系统页面间转向。类似 forward。
2. redirect
   浏览器跳转。可转向其他系统页面。
3. chain
   将处理结果转交给另外一个 Action 处理，以实现 Action 的链式处理。
4. velocity
   将指定的 velocity 模板作为结果呈现界面。
5. xslt
   将指定的 XSLT 作为结果呈现界面。

随后的 param 节点则设定了相匹配的资源名称。

⑷ interceptor-ref

设定了施加于此 Action 的拦截器（interceptor）。关于拦截器，请参见稍后的"XWork 拦截器体系"部分。

interceptor-ref 定义的是一个拦截器的应用，具体的拦截器设定，实际上是继承于 webwork-default package，我们可以在 webwork-default.xml 中找到对应的"params"和"model-driven"拦截器设置：

```
<interceptors>
    ……
    <interceptor name="params"
        class="com.opensymphony.xwork.interceptor.ParametersInterceptor" />
    <interceptor name="model-driven"
        class="com.opensymphony.xwork.interceptor.ModelDrivenInterceptor" />
    ……
</interceptors>
```

"params"大概是 Webwork 中最重要、也最常用的一个 Interceptor。上面曾经将 MVC 工作流程划分为几个步骤，其中的第一步:

"将 Web 页面中的输入元素封装为一个（请求）数据对象"

就是通过"params"拦截器完成。Interceptor 将在 Action 之前被调用，因而，Interceptor 也成为将 Webwork 传来的 MAP 格式的数据转换为强类型 Java 对象的最佳实现场所。

"model-driven"则是针对 Action 的 Model 驱动模式的 interceptor 实现。具体描述请参见稍后的“Action 驱动模式”部分

很可能我们的 Action 都需要对这两个 interceptor 进行引用。我们可以定义一个 interceptor-stack，将其作为一个 interceptor 组合在所有 Action 中引用。如，上面的配置文件可修改为：

```
<xwork>
  <include file="webwork-default.xml" />
  <package name="default" extends="webwork-default">
      <interceptors>
          <interceptor-stack name="modelParamsStack">
              <interceptor-ref name="params" />
              <interceptor-ref name="model-driven" />
          </interceptor-stack>
      </interceptors>

      <action name="login"
          class="net.xiaxin.webwork.action.LoginAction">
          <result name="success" type="dispatcher">
              <param name="location">/main.jsp</param>
          </result>

          <result name="loginfail" type="dispatcher">
              <param name="location">/index.jsp</param>
          </result>

          <interceptor-ref name="modelParamsStack" />

      </action>
  </package>
</xwork>
```

通过引入 interceptor-stack，我们可以减少 interceptor 的重复申明。

下面是我们的 Model 对象：

LoginInfo.java：

```
public class LoginInfo {
  private String password;
  private String username;
  private List messages = new ArrayList();
```

```
  private String errorMessage;

  public List getMessages() {
      return messages;
  }

  public String getErrorMessage() {
      return errorMessage;
  }

  public void setErrorMessage(String errorMessage) {
      this.errorMessage = errorMessage;
  }

  public String getPassword() {
      return password;
  }

  public void setPassword(String password) {
      this.password = password;
  }

  public String getUsername() {
      return username;
  }

  public void setUsername(String username) {
      this.username = username;
  }

}
```

很简单，这只是一个纯粹的值对象（Value-Object）。这里，它扮演着模型（Model）的角色，并与 Action 的输入输出密切相关。

与 Spring MVC 中的 Command 对象不同，Webwork 中的 Model 对象，扮演着承上启下的角色，它既是 Action 的输入参数，又包含了 Action 处理的结果数据。

换句话说，输入的 Http 请求参数，将被存储在 Model 对象传递给 Action 进行处理，Action 处理完毕之后，也将结果数据放置到 Model 对象中，之后，Model 对象与返回界面融合生成最后的反馈页面。

也正由于此，笔者建议在实际开发中采用 Model-Driven 模式，而非 Property-Driven 模式（见稍后“Action 驱动模式”部分），这将使得业务逻辑更加清晰可读。

对应的 Action 代码

```java
public class LoginAction implements Action, ModelDriven {

    private final static String LOGIN_FAIL="loginfail";

    LoginInfo loginInfo = new LoginInfo();

    public String execute() throws Exception {

        if ("erica".equalsIgnoreCase(loginInfo.getUsername())
                && "mypass".equals(loginInfo.getPassword())) {

            //将当前登录的用户名保存到Session
            ActionContext ctx = ActionContext.getContext();
            Map session = ctx.getSession();
            session.put("username",loginInfo.getUsername());

            //出于演示目的，通过硬编码增加通知消息以供显示
            loginInfo.getMessages().add("message1");
            loginInfo.getMessages().add("message2");
            loginInfo.getMessages().add("message3");

            return SUCCESS;
        }else{
            loginInfo.setErrorMessage("Username/Password Error!");
            return LOGIN_FAIL;
        }
    }

    public Object getModel() {
        return loginInfo;
    }
}
```

可以看到，LoginAction 实现了两个接口：

1. Action

   Action 接口非常简单，它指定了 Action 的入口方法（execute），并定义了几个默认的返回值常量：

```java
public interface Action extends Serializable {
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

```

```
    public String execute() throws Exception;
}
```

SUCCESS、NONE、ERROR、INPUT、LOGIN 几个字符串常量定义了常用的几类返回值。我们可以在 Action 实现中定义自己的返回类型，如本例中的 LOGIN_FAIL 定义。

而 execute 方法，则是 Action 的入口方法，XWork 将调用每个 Action 的 execute 方法以完成业务逻辑处理。

2. ModelDriven

ModelDriven 接口更为简洁：

```
public interface ModelDriven {
    Object getModel();
}
```

ModelDriven 仅仅定义了一个 getModel 方法。XWork 在调度 Action 时，将通过此方法获取 Model 对象实例，并根据请求参数为其设定属性值。而此后的页面返回过程中，XWork 也将调用此方法获取 Model 对象实例并将其与设定的返回界面相融合。

注意这里与 Spring MVC 不同，Spring MVC 会自动为逻辑处理单元创建 Command Class 实例，但 Webwork 不会自动为 Action 创建 Model 对象实例，Model 对象实例的创建需要我们在 Action 代码中完成（如 LoginAction 中 LoginInfo 对象实例的创建）。

另外，如代码注释中所描述，登录成功之后，我们随即将 username 保存在 Session 之中，这也是大多数登录操作必不可少的一个操作过程。

这里面牵涉到了 Webwork 中的一个重要组成部分：ActionContext。

ActionContext 为 Action 提供了与容器交互的途径。对于 Web 应用而言，与容器的交互大多集中在 Session、Parameter，通过 ActionContext 我们在代码中实现与 Servlet API 无关的容器交互。

如上面代码中的：

```
ActionContext ctx = ActionContext.getContext();
Map session = ctx.getSession();
session.put("username",loginInfo.getUsername());
```

同样，我们也可以操作 Parameter:

```
ActionContext ctx = ActionContext.getContext();
Map params = ctx.getParameters();
String username = ctx.getParameters("username");
```

上述的操作，将由 XWork 根据当前环境，调用容器相关的访问组件（Web 应用对应的就是 Webwork）完成。上面的 ActionContext.getSession()，XWork 实际上将通过 Webwork

提供的容器访问代码“HttpServletRequest.getSession()”完成。

注意到，ActionContext.getSession 返回的是一个 Map 类型的数据对象，而非 HttpSession。这是由于 WebWork 对 HttpSession 进行了转换，使其转变为与 Servlet API 无关的 Map 对象。通过这样的方式，保证了 Xwork 所面向的是一个通用的开放结构。从而使得逻辑层与表现层无关。增加了代码重用的可能。

此外，为了提供与 Web 容器直接交互的可能。WebWork 还提供了一个 ServletActionContext 实现。它扩展了 ActionContext，提供了直接面向 Servlet API 的容器访问机制。

我们可以直接通过 ServletActionContext.getRequest 得到当前 HttpServletRequest 对象的引用，从而直接与 Web 容器交互。获得如此灵活性的代价就是，我们的代码从此与 ServletAPI 紧密耦合，之后系统在不同平台之间移植就将面临更多的挑战（同时单元测试也难于进行）。

平台移植的需求并不是每个应用都具备。大部分系统在设计阶段就已经确定其运行平台，且无太多变更的可能。不过，如果条件允许，尽量通过 ActionContext 与容器交互，而不是平台相关的 ServletActionContext，这样在顺利实现功能的同时，也获得了平台迁移上的潜在优势，何乐而不为。

登录成功界面：

main.jsp:

```
<%@ taglib prefix="ww" uri="webwork"%>
<html>
<body>
    <p align="center">Login Success!</p>
    <p align="center">Welcome!
        <ww:property value="#session['username']"/>
    </p>
    <p align="center">
        <b>Messages:</b><br>
        <ww:iterator value="messages" status="index">
         <ww:if test="#index.odd == true">
           !<ww:property/><br>
         </ww:if>
         <ww:else>
           *<ww:property/><br>
         </ww:else>
        </ww:iterator>
    </p>
</body>
</html>
```

这里我们引入了 Webwork 的 taglib，如页面代码第一行的申明语句。

下面主要使用了三个 tag:

- `<ww:property value="#session['username']"/>`
  读取 Model 对象的属性填充到当前位置。
  value 指定了需要读取的 Model 对象的属性名。
  这里我们引用了 LoginAction 在 session 中保存的'username'对象。

  由于对应的 Model（LoginInfo）中也保存了 username 属性。下面的语句与之效果相同：
  `<ww:property value="username"/>`

  与 JSP2 中的 EL 类似，对于级联对象，这里我们也可以通过“.”操作符获得其属性值，如 value="user.username"将得到 Model 对象中所引用的 user 对象的 username 属性（假设 LoginInfo 中包含一个 User 对象，并拥有一个名为“username”的属性）。

  关于 EL 的内容比较简单，本文就不再单独开辟章节进行探讨。
  如 value="#session['username']"。Webwork 中包括以下几种特殊的 EL 表达式：
  - parameter['username'] 相当于 request.getParameter("username");
  - request['username'] 相当于 request.getAttribute("username");
  - session['username'] 从 session 中取出以“username”为 key 的值
  - application['username'] 从 ServletContext 中取出以“username”为 key 的值

  注意需要用“#”操作符引用这些特殊表达式。
  另外对于常量，需要用单引号包围，如#session['username'] 中的'username'。

- `<ww:iterator value="messages" status="index">`
  迭代器。用于对 java.util.Collection、java.util.Iterator、java.util.Enumeration,、java.util.Map、Array 类型的数据集进行循环处理。
  其中，value 属性的语义与<ww:property>中一致。
  而 status 属性则指定了循环中的索引变量，在循环中，它将自动递增。
  而在下面的<ww:if>中，我们通过“#”操作符引用这个索引变量的值。
  索引变量提供了以下几个常用判定方法：
  - first 当前是否为首次迭代
  - last 当前是否为最后一次迭代
  - odd 当前迭代次数是否奇数
  - even 当前迭代次数是否偶数

- `<ww:if test="#index.odd == true">`和`<ww:else>`
  用于条件判定。
  test 属性指定了判定表达式。表达式中可通过“#”操作符对变量进行引用。
  表达式的编写语法与 java 表达式类似。
  类似的，还有<ww:elseif test="……">。

登录失败界面

实际上，这个界面即登录界面 index.jsp。只是由于之前出于避免干扰的考虑，隐藏了 index.jsp 中显示错误信息的部分。

完整的 index.jsp 如下：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>
<html>
<body>
<form action="/login.action">

    <p align="center">
        登录<br>
        <ww:if test="errorMessage != null">
            <font color="red">
                <ww:property value="errorMessage"/>
            </font>
        </ww:if>

    </p>

    用户名:
    <input type="text" name="model.username" />
    <br>

    密  码 :
    <input type="password" name="model.password" />
    <br>

    <p align="center">
            <input type="submit" value="提交" name="B1"/>
            <input type="reset" value="重置" name="B2"/>
    </p>

</form>
</body>
</html>
```

这里首先我们进行判断，如果 Model 中的 errorMessage 不为 null，则显示错误信息。这样，在用户第一次登录时，由于 Model 对象尚未创建，errorMessage 自然为 null，错误信息不会显示，即得到了与之前的 index.jsp 同样的效果。

WebWork 高级特性

Action 驱动模式

Webwork 中，提供了两种 Action 驱动模式：

1. Property-Driven
2. Model-Driven

上面的示例中，我们应用了 Model-Driven 的 Action 驱动模式。对于这种模式，相信大家已经比较熟悉。

下面介绍一下 Property-Driven 驱动模式。

Model-Driven 是通过 Model 对象（上例中的 LoginInfo）贯穿整个 MVC 流程，而 Property-Driven，顾名思义， 是由 Property（属性）作为贯穿 MVC 流程的信息携带者。

Property 固然无法独立存在，它必须依附于一个对象。在 Model-Driven 模式中，实际上这些属性被独立封装到了一个值对象，也就是所谓的 Model。而 Property-Driven 中，属性将依附在 Action 对象中。

对上例进行一些小改造：

LoginActoin.java

```
public class LoginAction implements Action{

    private final static String LOGIN_FAIL="loginfail";

    private String password;
    private String username;
    private List messages = new ArrayList();
    private String errorMessage;

    public List getMessages() {
        return messages;
    }

    public String getErrorMessage() {
        return errorMessage;
    }

    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
```

```
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String execute() throws Exception {
        if ("erica".equalsIgnoreCase(getUsername())
                && "mypass".equals(getPassword())) {
            getMessages().add("message1");
            getMessages().add("message2");
            getMessages().add("message3");
            return SUCCESS;
        }else{
            setErrorMessage("Username/Password Error!");
            return LOGIN_FAIL;
        }
    }
}
```

可以发现，LoginAction 无需再实现 ModelDriven 接口，而原本作为 Model 的 LoginInfo 类的属性，亦被转移到 LoginAction 之中。

在 Property-Driven 模式中，不在有单独的 Model 存在，取而代之的是一个 Model 和 Action 融合之后得到的整体类结构。

其他需要修改的还有：

index.jsp

```
    ......
    用户名：
    <input type="text" name="username" />
    <br>

    密　码 :
    <input type="password" name="password" />
    <br>
    ......
```

xwork.xml（删除对 model-driven Interceptor 的引用）：

```
    ......
    <action name="login"
        class="net.xiaxin.webwork.action.LoginAction">
```

```
        <result name="success" type="dispatcher">
            <param name="location">/main.jsp</param>
        </result>

        <result name="loginfail" type="dispatcher">
            <param name="location">/index.jsp</param>
        </result>
        <interceptor-ref name="params" />
    </action>
    ……
```

这样，我们的程序就可以以 Property-Driven 的模式运行。

Property-Driven 和 Model-Driven 模式的比较：

从上面改造后的例子可以感觉到，Property-Driven 驱动模式似乎更加简单，无需再实现 ModelDriven 接口。也减少了一个 Model 类。Jsp 文件和 xwork.xml 也有了些许简化。

出于对框架灵活性的考虑，Webwork 提供了如上两种驱动模式。但对于我们的应用系统而言，这样的灵活性有时反而使得我们有点无所适从——使用 Property-Driven 模式似乎更加简单自由，而使用 Model-Driven 模式又似乎更加清晰……

Webwork 为我们提供了更多的选择和更大的自由度。面对这些选项我们该如何抉择？

就笔者的观点而言。Model-Driven 对于维持软件结构清晰性的贡献，超过了其所带来的微不足道的复杂性。

记得关于面向对象设计法则的著作《Object-Oriented Design Heuristics》[4]中有这样一个有趣的问题（可能与原文在文字细节上有所差异，一时找不到原书，只能凭回忆描述）：

> 当你面对一头奶牛的时候，你会对它说“请给我挤一杯牛奶”。
> 还是对身边的农场工人说“请给我挤一杯牛奶”？

大多程序员对于这样的提问都会不屑一顾，愚蠢的问题，不是么？

不过在软件开发过程中，程序员们却常常不停的对着奶牛大喊“挤牛奶！挤牛奶！”。

回头看看这里的 Property-Driven 模式，是不是也有点这样的味道……

作为贯穿 WebWork MVC 的信息载体，Model 扮演着奶牛的角色，它携带了我们所需要的数据资源（牛奶）。而如何操作这些数据，却不是奶牛的任务，而是农场工人（Action）

[4] 中文版已经由人民邮电出版社出版，名为《OOD 启思录》

的工作。

如此一来，Property-Driven 模式的身份似乎就有点混杂不清。

这也就是笔者所想要表达的意思，Webwork 出于框架灵活性的考虑，提供了 Property-Driven 模式供用户选择，但作为用户的我们，还是需要有着一定取舍原则，这里，笔者推荐将 Model-Driven 驱动模式作为 WebWork 开发的首选。

此外，对于 Property-Driven 模式还有一种应用方法值得一提：即将业务逻辑从 Action 中抽离，而在 Action 之外的逻辑单元中提供对应的实现。

这个改进方案，借用 LoginAction 表示大致可以描述如下：

```
public class LoginAction implements Action{
    …Property getter/setter略…
    public String execute() throws Exception {
        return LoginLogic.doLogin (getUsername(),getPassword());
    }
}
```

LoginLogic 是系统中的逻辑实现类，它提供了一个方法 doLogin 以完成实际的 Login 工作。

如果采用以上方案，那么其中的 Action 就成为实际意义上的 Model Object。这里的 Property-Driven 模式也就实际演变成了 Model-Driven 模式，只是将逻辑层更加向下推进了一层。原本的逻辑层在 Action 中实现，现在转到了框架之外的逻辑实现单元中。

通过这样的 Property-Driven 模式，我们的 LoginAction 无需再实现 ModelDriven 接口，xwork 配置中减少了一行配置代码。

而最为重要的是，应用系统的关键所在——业务逻辑，将与框架本身相分离，这也就意味着代码的可移植性将变得更强。

是不是我们的系统中就应该采用这样的模式？

还是老话，根据实际需求而定。如果产品之后可能需要提供 Web 之外的表现渠道，那么采用上述 Property-Driven 模式是个不错的选择。否则，对于一般的 Web 应用开发而言（如网上论坛），采用 Model-Driven 模式即可。

为什么这里并不建议在所有项目中都采用上述模式？原因有几个方面：首先，Web 系统开发之后，切换框架的可能性微乎其微，而同时，Xwork 的侵入性已经很小，移植的代价并不高昂（这也是 Xwork 相对 Struts 的一个突出优势）。

此外，还有一个最重要的问题。这样的模式（业务逻辑只允许在独立的逻辑单元中实现，而不能混杂在 Action 中）需要较强的开发规范的约束，而人本的约束往往最不可靠。

规则制定人固然能保证自己遵循这个开发模式，但在团队协同开发的过程中，是否真

的能保证每个人都按照这样的模式开发，这还是一个疑问。随之而来的代码复查工作的必然增加，将带来更多的生产率损耗。

当然，具体如何抉择，仁者见仁智者见智，根据客户的真实需求选择适用的开发模式，将为您的团队带来最佳的效益比。

XWork 拦截器体系

Interceptor（拦截器），顾名思义，就是在某个事件发生之前进行拦截，并插入某些处理过程。

Servlet 2.3 规范中引入的 Filter 算是拦截器的一个典型实现，它在 Servlet 执行之前被触发，对输入参数进行处理之后，再将工作流程传递给对应的 Servlet。

而今年来兴起的 AOP（Aspect Oriented Programming），更是将 Interceptor 的作用提升到前所未有的高度。

Xwork 的 Interceptor 概念与之类似。即通过拦截 Action 的调用过程，为其追加预处理和后处理过程。

回到之前讨论过的一个问题：

“将 Web 页面中的输入元素封装为一个（请求）数据对象”

对于 Xwork 而言，前端的 Webwork 组件为其提供的是一个 Map 类型的数据结构。而 Action 面向的却是 Model 对象所提供的数据结构。

在何时、何处对这两种不同的数据结构进行转换？自然，我们可以编写一个辅助类完成这样的工作，并在每次 Action 调用之前由框架代码调用他完成转换工作，就像 Struts 做的那样。

这种模式自然非常简单，不过，如果我们还需要进行其他操作，比如验证数据合法性，那么，我们又需要增加一个辅助类，并修改我们的框架代码，加入这个类的调用代码。显然不是长久之计。

Xwork 通过 Interceptor 实现了这一步骤，从而我们可以根据需要，灵活的配置所需的 Interceptor。从而为 Action 提供可扩展的预处理、后处理过程。

前面曾经提及，ActionInvocation 是 Xworks 中 Action 调度的核心。而 Interceptor 的调度，也正是由 ActionInvocation 负责。

ActionInvocation 是一个接口，而 DefaultActionInvocation 则是 Webwork 对 ActionInvocation 的默认实现。

Interceptor 的调度流程大致如下：

1. ActionInvocation 初始化时，根据配置，加载 Action 相关的所有 Interceptor
   参见 ActionInvocation.init 方法中相关代码：

```
private void init() throws Exception {
    ……
    List interceptorList = new
            ArrayList(proxy.getConfig().getInterceptors());
    interceptors = interceptorList.iterator();
}
```

2. 通过 ActionInvocation.invoke 方法调用 Action 实现时，执行 Interceptor:
   下面是 DefaultActionInvocation 中 Action 调度代码：

```
public String invoke() throws Exception {
    //调用intercaptors
    if (interceptors.hasNext()) {
        Interceptor interceptor =
                    (Interceptor) interceptors.next();
        resultCode = interceptor.intercept(this);
    } else {
        if (proxy.getConfig().getMethodName() == null) {
            resultCode = getAction().execute();
        } else {
            resultCode = invokeAction(
                                getAction(),
                                proxy.getConfig()
                              );
        }
    }
    ……
}
```

可以看到，ActionInvocation 首先会判断当前是否还有未执行的 Interceptor，如果尚有未得到执行的 Interceptor，则执行之，如无，则执行对应的 Action 的 execute 方法（或者配置中指定的方法）。

这里可能有点疑问，按照我们的理解，这里首先应该对 interceptorList 循环遍历，依次执行各 interceptor 之后，再调用 Action.execute 方法。

不过需要注意的是，类似 Servlet Filter，interceptor 之间并非只是独立的顺序关系，而是层级嵌套关系。

也就是说，Interceptor 的调用过程，是首先由外层的（如按定义顺序上的第一个）Inerceptor 调用次级的（定义顺序上的第二个）Interceptor，之后如此类推，直到最终的 Action，再依次返回。

这是设计模式"**Intercepting Filter**[5]"的一个典型应用。

同样，Servlet Filter 也是"Intercepting Filter"模式的一个典型案例，可以参照 ServletFilter 代码，进行一些类比：

```
public void doFilter(ServletRequest srequest,
                     ServletResponse sresponse,
                     FilterChain chain)
    throws IOException, ServletException {

    HttpServletRequest request = (HttpServletRequest)srequest;
    request.setCharacterEncoding(targetEncoding);
    //nest call
    chain.doFilter(srequest,sresponse);
```

[5] 关于 Intercepting Filter Pattern，请参见 http://java.sun.com/blueprints/patterns/InterceptingFilter.html

```
}
```

为了有进一步的感性认识，我们从 Interceptor 的实现机制上入手。

所有的拦截器都必须实现 Interceptor 接口：

```
public interface Interceptor {
    void destroy();
    void init();
    String intercept(ActionInvocation invocation) throws Exception;
}
```

在 Interceptor 实现中，抽象实现 AroundInterceptor 得到了最广泛的应用（扩展），它增加了预处理（before）和后处理（after）方法的定义。

AroundInterceptor.java:

```
public abstract class AroundInterceptor implements Interceptor {
    protected Log log = LogFactory.getLog(this.getClass());
    public void destroy() {
    }
    public void init() {
    }

    public String intercept(ActionInvocation invocation) throws
Exception {
        String result = null;

        before(invocation);
        result = invocation.invoke();
        after(invocation, result);

        return result;
    }

    protected abstract void after(ActionInvocation dispatcher, String
result)
        throws Exception;
    /**
     * Called before the invocation has been executed.
     */
    protected abstract void before(ActionInvocation invocation)
        throws Exception;
}
```

AroundInterceptor.invoke 方法中，调用了参数 invocation 的 invoke 方法。结合

前面 ActionInvocation.invoke 方法的实现，我们即可看出 Interceptor 层级嵌套调用的机制。

这样的嵌套调用方式并非 AroundInterceptor 所独有，通过浏览 Xwork 源代码，我们可以发现，DefaultWorkflowInterceptor 等其他的 Interceptor 实现也同样遵循这一模式。

最后，我们结合最常用的 ParametersInterceptor，看看 Xwork 是如何通过 Interceptor，将 Webwork 传入的 Map 类型数据结构，转换为 Action 所需的 Java 模型对象。

ParametersInterceptor.java:

```
public class ParametersInterceptor extends AroundInterceptor {

    protected void after(ActionInvocation dispatcher, String result)
        throws Exception {
    }

    protected void before(ActionInvocation invocation) throws Exception
{
        if (!(invocation.getAction() instanceof NoParameters)) {
            final Map parameters =
                        ActionContext.getContext().getParameters(); (1)

            if (log.isDebugEnabled()) {
                log.debug("Setting params " + parameters);
            }

            ActionContext invocationContext =
                        invocation.getInvocationContext();

            try {
                invocationContext.put(
                    InstantiatingNullHandler.CREATE_NULL_OBJECTS,
                    Boolean.TRUE);
                invocationContext.put(
                    XWorkMethodAccessor.DENY_METHOD_EXECUTION,
                    Boolean.TRUE);
                invocationContext.put(
                    XWorkConverter.REPORT_CONVERSION_ERRORS,
                    Boolean.TRUE);

                if (parameters != null) {
                    final OgnlValueStack stack =
                        ActionContext.getContext().getValueStack();    (2)

                    for (Iterator iterator =                           (3)
```

```
                        parameters.entrySet().iterator();
                        iterator.hasNext();
                    ) {
                    Map.Entry entry = (Map.Entry) iterator.next();
                    stack.setValue(                                     (4)
                        entry.getKey().toString(),
                        entry.getValue());
                }
            }
        } finally {
            invocationContext.put(
                InstantiatingNullHandler.CREATE_NULL_OBJECTS,
                Boolean.FALSE);
            invocationContext.put(
                XWorkMethodAccessor.DENY_METHOD_EXECUTION,
                Boolean.FALSE);
            invocationContext.put(
                XWorkConverter.REPORT_CONVERSION_ERRORS,
                Boolean.FALSE);
        }
    }
  }
}
```

ParametersInterceptor 扩展了抽象类 AroundInterceptor。并在其预处理方法（before）中实现了数据的转换。

数据转换的过程并不复杂:

⑴ 首先由 ActionContext 获得 Map 型的参数集 parameters。
⑵ 由 ActionContext 获得值栈（OgnlValueStack）。
⑶ 遍历 parameters 中的各项数据。
⑷ 通过 OgnlValueStack，根据数据的键值，为 Model 对象填充属性数据。

OgnlValueStack 是 http://www.ognl.org[6]提供的一套可读写对象属性的类库（功能上有点类似 Jakarta Commons BeanUtils，以及 Spring BeanWrapper）。

OgnlValueStack 使用非常简单，下面这段示例代码将 User 对象的 name 属性设为“erica”。

```
OgnlValueStack stack = new OgnlValueStack();
stack.push(new User());//首先将欲赋值对象压入栈中
stack.setValue("name","erica");//为栈顶对象的指定属性名赋值
```

上面的代码中并没有发现将Model对象入栈的部分，是由于ActionInvocation在初始化的时候已经预先完成了这个工作，如DefaultActionInvocation.init方法中代码所示：

[6] OGNL=Object-Graph Navigation Language。提供基于表达式的对象属性访问功能。在许多项目中得到使用，如 Webwork、Apache Tapestry 等。Spring Framework 1.2 版本中也将引入。

```
    private void init() throws Exception {
        Map contextMap = createContextMap();

        createAction();

        if (pushAction) {
            stack.push(action);
        }
        ......
    }
```

输入校验

Web 应用开发中，我们常常面临如何保证输入数据合法性的头痛问题。实现输入数据校验的方法无外乎两种：

1. 页面 Java Script 校验
2. 服务器端、执行逻辑代码之前进行数据校验

Struts、Spring MVC 均为服务器端的数据校验提供了良好支持。

对于客户端的校验，大多 MVC 框架力所不逮。

而 WebWork 则在提供灵活的服务器端数据校验机制的基础上，进一步提供了对页面 Java Script 数据校验的支持。这也可算是 WebWork 中的一个亮点。

与 Spring MVC 类似，XWork 也提供了一个 Validator 接口，所有数据校验类都必须实现这个接口。

XWork 发行时已经内置了几个常用的数据校验类（位于包 com.opensymphony.xwork.validator.validators），同时也提供了 Validator 接口的几个抽象实现（抽象类），以便于用户在此基础上进行扩展（如 FieldValidatorSupport）。

服务器端数据合法性校验的动作，发生在 Action 被调用之前。回忆之前关于 Xwork 拦截器体系的讨论，我们自然想到，通过 Interceptor 在 Action 运作之前对其输入参数进行校验是一个不错的思路。事实的确如此。WebWork 中提供了一个 ValidationInterceptor，它将调用指定的 Validator 对输入的参数进行合法性校验。

对这一细节感兴趣的读者可参考 Xwork 中的 ActionValidatorManager 和 ValidationInterceptor 的实现代码。

下面我们将首先讨论 Validator 的使用，之后再完成一个针对特定需求的 Validator 实现。

为“用户登录”示例加入数据合法性校验：

1. 首先，为 Action 配置用于数据校验的 Interceptor：

```
<xwork>
  <include file="webwork-default.xml" />
  <package name="default" extends="webwork-default">

    <action name="login"
        class="net.xiaxin.webwork.action.LoginAction">

        <result name="success" type="dispatcher">
            <param name="location">/main.jsp</param>
        </result>

        <result name="loginfail" type="dispatcher">
            <param name="location">/index.jsp</param>
```

```
        </result>

        <interceptor-ref name="params" />
        <interceptor-ref name="model-driven" />
        <interceptor-ref name="validationWorkflowStack" />

    </action>
  </package>
</xwork>
```

这里我们并没有直接引用 ValidationIncepteror。而是引用了 validationWorkflowStack，webwork-default.xml 中对其定义如下：

```
<interceptor-stack name="validationWorkflowStack">
    <interceptor-ref name="defaultStack"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
</interceptor-stack>
```

这表明 validationWorkflowStack 实际上是三个 Interceptor 的顺序组合。

为什么要引用这样一个 Interceptor 组合，而不是直接引用 validation Inceptor？
这与我们的实际需求相关，在 Web 应用中，当输入数据非法时，一般的处理方式是返回到输入界面，并提示数据非法。而这个 Interceptor 组合则通过三个 Interceptor 的协同实现了这一逻辑。由此也可见 Webwork 中拦截器体系设计的精妙所在。

2. 在 Class Path 的根目录（如 WEB-INF/classes）创建一个 validators.xml 文件，此文件中包含当前应用中所有需要使用的 Validator。

   Webwork 发行包中提供了一个 validators.xml 示例（\bin\validators.xml）可供参考，一般而言，我们只需根据实际需要在此文件上进行修改即可。

   对于我们的登录示例而言，需要校验的有两个字段，用户名和密码，校验逻辑分别为：
   1. 用户名不可为空
   2. 密码不可为空，且长度必须为 4 到 6 位之间

   这里，我们通过 Xwork 提供的 RequiredStringValidator 和 StringLengthFieldValidator 来实现这一校验逻辑。

```
<validators>

    <validator name="required"
    class="com.opensymphony.xwork.validator.validators.Requ
iredStringValidator" />

    <validator name="length"
```

```
    class="com.opensymphony.xwork.validator.validators.Stri
ngLengthFieldValidator" />
</validators>
```

配置非常简单，只需指定此 Validator 的实现类和及其名称。下面的数据校验配置文件中，将通过此 Validator 名称，对实际的 Validator 实现类进行引用。

3. 针对页面表单字段配置对应的 Validator。
在我们的登录页面中，共有两个字段 model.username 和 model.password。为其建立一个配置文件：LoginAction-validation.xml：

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork
Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.d
td">

<validators>

    <field name="model.username">
        <field-validator type="required">
            <message>Please enter Username!</message>
        </field-validator>
    </field>

    <field name="model.password">
        <field-validator type="length">
            <param name="minLength">4</param>
            <param name="maxLength">6</param>
            <message>
              Password length must between ${minLength} and
${maxLength} chars!
             </message>
        </field-validator>
    </field>
</validators>
```

配置文件有两种命名约定方式：

1. Action 类名-validation.xml
   如上面的 LoginAction-validation.xml
2. Action 类名-Action 别名-validation.xml
   如 LoginAction-login-validation.xml
   其中 Action 别名就是 xwork.xml 中我们申明 Action 时为其设定的名称。

配置文件必须放置在与对应 Action 实现类相同的目录。

配置文件格式非常简单，结合 validators.xml，我们可以很直观的看出字段的校验关系。
值得注意的是，通过 param 节点，我们可以为对应的 Validator 设置属性值。这一点与 Spring IOC 非常类似，是的，实际上 XWork 也提供了内置的 IOC 实现，不过与 Spring 的 IOC 支持想比还有一些差距，这里就不再深入探讨。有兴趣的读者可参见一下文档：

另外 message 定义中，我们可以通过“${}”操作符对属性进行引用。

4. 修改 LoginAction，使继承 ActionSupport 类。

```
public class LoginAction extends  ActionSupport implements
Action, ModelDriven
{
    ……
}
```

ActionSupport 类实现了数据校验错误信息、Action 运行错误信息的保存传递功能。通过扩展 ActionSupport，LoginAction 即可携带执行过程中的状态信息，这为之后的错误处理，以及面向用户的信息反馈提供了基础数据。

5. 修改页面，增加数据合法性校验错误提示：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>
<html>
    <style type="text/css">
    .errorMessage {
        color: red;
    }
    </style>
<body>
<form action="/login.action">
    <ww:if test="hasFieldErrors()">
        <span class="errorMessage">
            <b>Errors:</b><br>
            <ww:iterator value="fieldErrors">
              <li><ww:property value="value[0]"/></li>
            </ww:iterator>
        </span>
    </ww:if>
    …以下略…
</form>
```

```
</body>
</html>
```

首先，我们通过<ww:if test="hasFieldErrors()">判断是否存在字段验证错误，hasFieldErrors()是 ActionSupport 中提供的方法，用于判定当前是否存在字段验证错误。LoginAction 扩展了 ActionSupport 类，自然继承了这个方法，我们即可在页面中通过 EL 对其进行调用。

如果存在 FieldErrors，则通过一个迭代，在页面上显示错误信息：

```
<ww:iterator value="fieldErrors">
    <li><ww:property value="value[0]"/></li>
</ww:iterator>
```

fieldErrors 是 ActionSupport 中保存字段校验错误信息的 Map 结构。针对这个 Map 进行迭代，得到的每个迭代项都是一个 key-value Entry，key 中保存着字段名，value 则是一个 List 数据结构，里面包含了针对这个 Key 的错误信息列表，错误信息的数量取决于字段验证规则配置中的设定。这里 value="value[0]"，也就是取当前条目 value 中保存的第一条错误信息。

我们在表单中，输入一个符合校验条件的 Password 之后提交，显示结果如下：

Errors:
- Please enter Username!

登录

用户名：
密 码 ：

提交 重置

为了显示一条错误信息，页面中需要增加这么多的代码，似乎繁琐了些。
为了提供简便、统一的页面构成组件。WebWork 提供了一套 UI Tag。UI Tag 对传统的 HTML TAG 进行了封装，并为其增加了更多辅助功能。如服务端校验错误信息的显示，甚至还可以自动为表单元素生成用于客户端校验的 JavaScript 脚本。

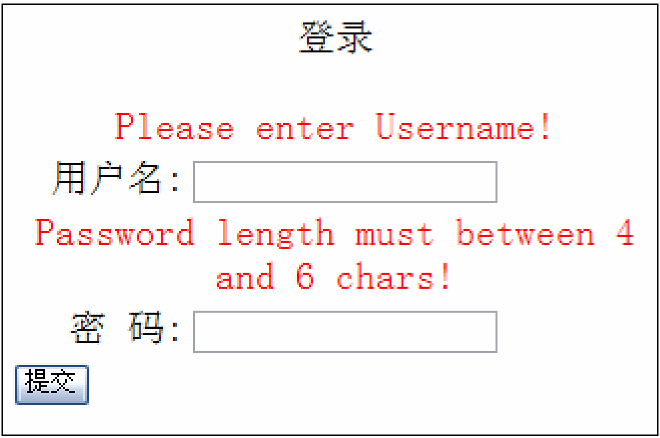
Webwork 提供的 UI_Tag 非常丰富，这里就不逐个介绍。Opensymphony 的 Wiki 站点中提供了丰富的信息，包括功能描述和实例，是开发实践过程中的必备参考资料：
http://wiki.opensymphony.com/display/WW/UI+Tags

下面来看上例的 UI Tag 实现版本：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>

<html>
  <style type="text/css">
  .errorMessage {
      color: red;
  }
  </style>
<body>

<p align="center">登录</p>
<ww:form name="'login'" action="'login'" method="'post'">

  <ww:textfield label="'用户名'" name="'model.username'" />
  <ww:password label="'密  码'" name="'model.password'" />
  <ww:submit value="'提交'"/>

</ww:form>

</body>
</html>
```

（其中的表单属性值均为常量，因此均以单引号包围，以与 Model 中的变量相区分）

运行结果如下：



我们看到，使用 ui-tag 的页面代码简洁了许多，不过，上面的页面布局似乎有点凌乱。

原因一方面在于代码中并没有特意考虑页面布局，毕竟这只是一个用作演示的界面。

另一方面，也就是使用 ui-tag 所需付出的代价——界面灵活性上的牺牲。当然这并不意味着使用了 ui-tag 我们就无法对 tag 内部的 html 结构和风格做出修改，而是作出修改的难度有所增加。

使用 Webwork UI-Tag 构建的 jsp 页面，在编译的时候会从\template\xhtml 目录（默认）中读取 Tag 模板以生成最终的页面 html。

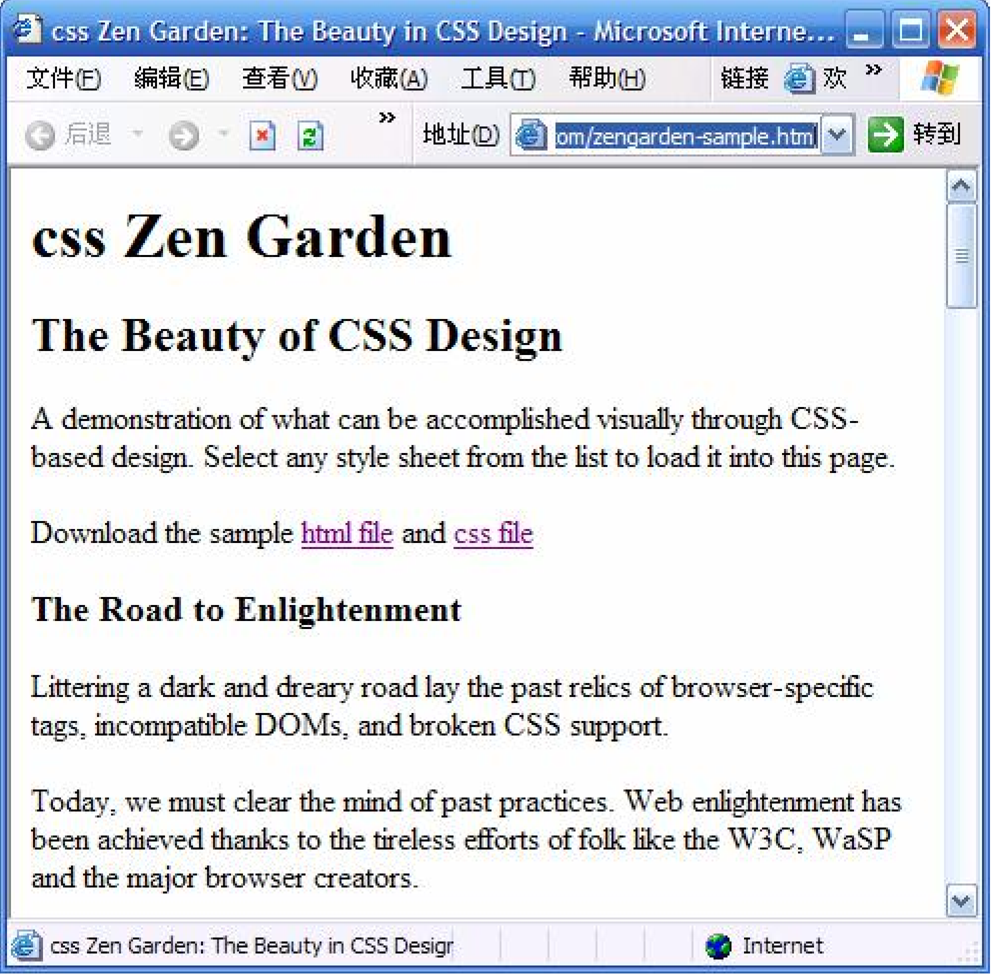
Webwork.jar 中包含了一些预制的模板（\template）。但这些预制模板面对复杂、个性化的页面设计需求可能有点力不从心。这就要求我们具备编写 Webwork 模板文件的能力。

前面曾经提及，这些模板是基于 Velocity 编写，velocity 语法虽然并不复杂，但如果界面设计多样、修改这些琐碎的模板也不是一件轻松的事情。

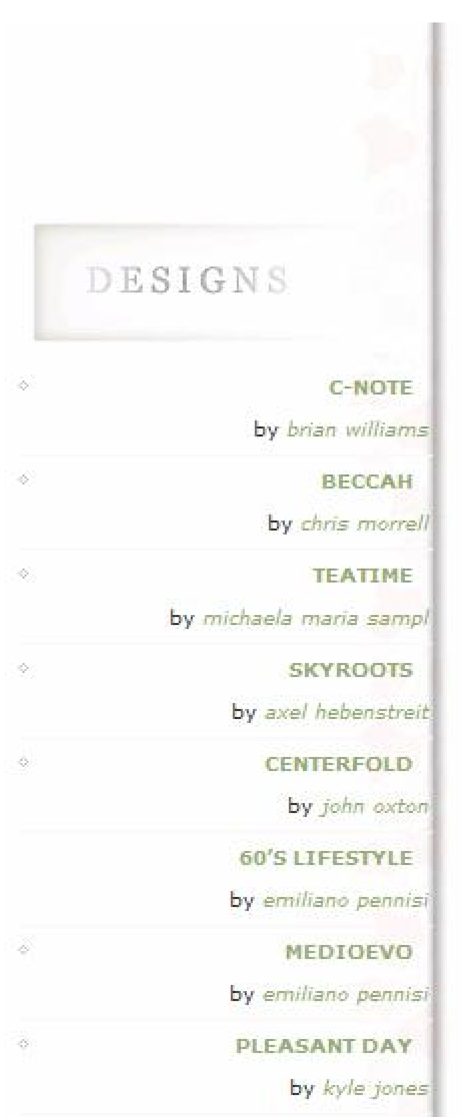
除了通过修改模板改变页面布局，我们还可以通过另外一种手段，实现对页面风格的改造，即 CSS。

通常情况下，CSS 对于大多数 Web 开发者而言，只是用作统一指定页面字体和表格边框属性的一个样式设定工具。不过，CSS 的功能远超于此。我们不但可以指定字体和表格边框这些页面元素属性，也可以通过 CSS 对页面的结构进行调整，同样的一个页面，应用不同的 CSS，可能是完全不同的页面表现效果。

CSS 禅意花园（http://www.csszengarden.com）展示了样式表的神奇之处，效果可能出乎大多数人意料之外：
下面是一个未应用 CSS 的纯 Html 页面样本：



下面是匹配不同 CSS 之后的效果：


DESIGNS
C-NOTE
by brian williams
BECCAH
by chris morrell
TEATIME
by michaela maria sampl
SKYROOTS
by axel hebenstreit
CENTERFOLD
by john oxton
60'S LIFESTYLE
by emiliano pennisi
MEDIOEVO
by emiliano pennisi
PLEASANT DAY
by kyle jones

...the beauty of css design
CSS ZEN GARDEN
DOWNLOAD THE SAMPLE HTML FILE AND CSS FILE
... a demonstration of what can be accomplished visually through CSS based design. Select any style sheet from the list to load it into this page ...
The Road to Enlightenment

想必大家已经感受到冲击性的效果，CSS 原来也可以被发挥得如此淋漓尽致。同样一个 JSP 页面，应用不同的 CSS 设计，完全两样。

不过，这里并没有暗示您应该尽快投入到 CSS 的怀抱，精通 CSS 并能将其发挥到淋漓尽致的人并不多。我们必须根据团队中成员的技术水平和专业分工，在各种选择之间的优劣与代价之前作出权衡。

对于持续的产品开发而言，UI-Tag 结合 CSS 无疑在页面代码简洁性和模板重用上达到了较高水平。一旦开发出一套符合公司产品风格的模板，随后的项目即可对这些模板进行重用。在长期来看，可以带来生产率的提高。另一方面， Web 系统可以轻易的实现换肤功能。

然而对于短期、界面设计多变而人力投入有限的小型项目开发而言。使用 UI-Tag 可能就有点让人疲惫。为了调整一个表单的结构往往要来回修改调试多次，远不如使用传统 jsp 简单高效。建议初次使用 Webwork 进行项目开发的用户不要轻易使用 UI-Tag，除非您已经积累了足够的经验。

关于模板和 Theme，请参见：
http://wiki.opensymphony.com/display/WW/Themes

看过这个主题之后，相信可以对 UI-Tag 的特点和使用成本有所把握。

使用 UI-Tag 的注意点：

请确定/WEB-INF/web.xml 文件中的 ServletDispatcher 设定为自动加载，如下：

```
<servlet>
  <servlet-name>webwork</servlet-name>
     <servlet-class>
        com.opensymphony.webwork.dispatcher.ServletDispatcher
     </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
```

UI tag内部是基于Velocity实现（无论我们选用的表现层技术是JSP还是Velocity，UI tag内部都是通过velocity生成最后的界面元素）。

ServletDispatcher在初始化过程中将完成包括创建Velocity管理器在内的一系列工作。

如果ServletDispatcher没有设置为自动加载模式，则只有当第一个Action 请求被提交时，ServletDispatcher才由Web容器加载。

这也就意味着，当你在应用启动完毕，首次访问含有UI tag的页面（*.jsp）时，由于ServletDispatcher的初始化工作尚未进行（尚未有.Action请求被调用），UI tag所依赖的Velocity管理器尚未创建，页面也就无法正常构建，我们将不得不面对充斥着异常信息的错误提示页面。

最后，来看看客户端数据校验。

Webwork的客户端数据校验功能是一个亮点。它会根据用户的配置，根据预定义的JavaScript模板，自动在页面生成验证脚本。不过使用这一功能的前提条件就是，必须使用UI-Tag构建表单。

下面我们尝试对上面示例进行修改，为其增加客户端校验功能。

非常简单，修改validators.xml和LoginAction-validation.xml，使其包含JavaScriptRequiredStringValidator的定义：

validators.xml

```
<validators>
  <validator name="required"
  class="com.opensymphony.webwork.validators.JavaScriptRequiredStri
ngValidator" />
</validators>
```

LoginAction-validation.xml:

```
<validators>

  <field name="model.username">
      <field-validator type="required">
          <message>Please enter Username!</message>
      </field-validator>
  </field>

</validators>
```

修改jsp页面，为其ww:form加上validate="true"选项：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>

<html>
<body>

<p align="center">登录</p>
<ww:form name="'login'" action="'login'" method="'post'"
    validate="true">

  <ww:textfield label="'用户名'" name="'model.username'"/>
  <ww:password label="'密  码'" name="'model.password'" />
  <ww:submit value="'提交'" align="center"/>

</ww:form>
```

```
</body>
</html>
```

此时，在浏览器中访问此页面，不填用户名提交，则将得到一个JavaScript警告窗口：

查看页面源代码可以看到，Webwork在生成的页面内，自动加入了针对表单域的JavaScript校验代码。

这一切实现是如何实现的？看看Webwork客户端校验器代码也许有助于加深理解。

下面是Webwork内置的客户端/服务器数据校验类JavaScriptRequiredStringValidator：

```
public class JavaScriptRequiredStringValidator extends
RequiredStringValidator implements ScriptValidationAware {

    public String validationScript(Map parameters) {
        String field = (String) parameters.get("name");
        StringBuffer js = new StringBuffer();

        js.append("value = form.elements['" + field + "'].value;\n");
        js.append("if (value == \"\") {\n");
        js.append("\talert('" + getMessage(null) + "');\n");
        js.append("\treturn '" + field + "';\n");
        js.append("}\n");
        js.append("\n");

        return js.toString();
    }
}
```

JavaScriptRequiredStringValidator 扩展了服务器端校验类RequiredStringValidator并实现了ScriptValidationAware接口，从而同时具备了客户端和服务器两端校验的功能，这样即使浏览器的JavaScript功能被禁用，也可以通过服务器端校验防止非法数据的提交。

从代码中我们可以看出，ScriptValidationAware接口定义的validationScript方法提供了JavaScript校验脚本，Webwork进行页面合成时，即可通过调用此方法获得校验脚本并将其插入到页面中。

使用客户端校验的注意点：

1. 必须为ww:form标签指定form name，Webwork生成JavaScript代码时，将通过此Form name作为表单元素的索引。

2. 对于namespace中的action而言，ww:form中必须指定namespace属性。
如对于以下namespace中的login action:

```
<package name="default" namespace="/user"
                extends="webwork-default">
    <action name="login" …>
    ……
</package>
```

对应的ww:form标签申明为：

```
<ww:form name="'login'" namespace="'/user'" action="'login'"
    method="'post'" validate="true">
    ……
</ww:form>
```

3. ww:form中的action无需追加.action后缀，webwork会自动在生成的页面中为其追加.action后缀。

国际化支持

Webwork 的国际化支持主要体现在两个部分：

1. UI-Tag
2. Validator

UI-Tag 中的<ww:i18n/>和<ww:text/>标记为页面显示提供了国际化支持，使用非常简单，下面的例子中，假定登录页面中，针对不同语种（中英文）需要显示欢迎信息“欢迎“或 Welcome”。

```
<ww:i18n name="'messages'">
  <ww:text name="'welcome'"/>
</ww:i18n>
```

上面的脚本，在运行期将调用 JDK ResourceBundle 读取名为 messages 的国际化资源，并从中取出 welcome 对应的键值。

ResourceBundle 会自动在 CLASSPATH 根路径中按照如下顺序搜寻配置文件并进行加载（以当前 Locale 为 zh_CN 为例）：

messages_zh_CN.properties
messages_zh.properties
messages.properties
messages_zh_CN.class
messages_zh.class
messages.class

本示例对应的两个配置文件如下：

messages_zh_CN.properties：

```
welcome=欢迎
```

注意中文资源文件必须通过JDK中内置的native2ascii工具进行转码（转为Unicode）方可正常显示。

转码后的messages_zh_CN.properties：

```
welcome=\u6B22\u8FCE
```

messages_en_US.properties:

```
welcome=Welcome
```

之后，Webwork 会自动根据当前的 Locale 设置，读取对应的资源文件填充页面。在中文系统上，这里即显示“欢迎”，如果是英文系统，则显示“Welcome”。

如果需要在文本中附带参数，则可通过 ww:param 指定参数值，如：

```
<ww:i18n name="'messages'">
  <ww:text name="'welcome'">
    <ww:param>192.168.0.2</ww:param>
```

```
      <ww:param>Erica</ww:param>
    </ww:text>
  </ww:i18n>
```

未转码前的 messages_zh_CN.properties：

```
welcome=欢迎来自{0}的用户{1}
```

运行时，{0}和{1}将被 ww:param 指定的参数值替换，显示结果如下：

欢迎来自 192.168.0.2 的用户 Erica

此外，在 UI Tag 中，我们还可以通过 getText 方法读取国际化资源，如对于刚才的登录界面，如果要求对于输入框的标题栏也实现国际化支持，则可对表单中的内容进行如下改造：

```
  <ww:i18n name="'messages'">
    <ww:text name="'welcome'">
      <ww:param>192.168.0.2</ww:param>
      <ww:param>Erica</ww:param>
    </ww:text>

    <ww:textfield label="getText('username')"
                  name="'model.username'"/>
    <ww:password  label="getText('password')"
                  name="'model.password'" />
    <ww:submit value="getText('submit')" align="center"/>

  </ww:i18n>
```

同时在资源文件添加 username，password 和 submit 对应的键值配置即可。

除了页面上的国际化支持，我们注意到，另外一个可能影响到用户界面语种问题的内容就是 Validator，我们在 Validator 中定义了校验失败时的提示信息，如：

```
<field name="model.username">
    <field-validator type="required">
        <message>Please enter Username!</message>
    </field-validator>
</field>
```

这里的 message 节点，定义了校验失败时的提示信息，对于这样的提示信息我们必须也为其提供国际化支持。

Validator 中国际化支持通过 message 节点的 key 属性完成：

```
<field name="model.username">
    <field-validator type="required">
```

```
            <message key="need_username">
                Please enter Username!
            </message>
        </field-validator>
    </field>
```

这里的 key 指定了资源文件中的键名，显示提示信息时，Webwork 将首先在资源文件中查找对应的键值，如果找不到对应的键值，才以上面配置的节点值作为提示信息。

Validator 对应的资源文件，与 Action 校验配置（这里就是 LoginAction_validation.xml）位于同一路径，命名方式为“<ActionName><_LOCALE>.properties”，这里对应的就是“LoginAction_zh_CN.properties”。

对应上例，只需在资源文件中配置 need_username 键值，Webwork 在运行期就会自动从此资源文件中读取数据，以之作为校验失败提示信息。

Webwork2 in Spring

Spring MVC 在设计时，针对 Webwork 的不足提出了自己的解决方案。不过，设计者一旦脱离实际开发，开始陷入框架本身设计的完美化时，往往容易陷入过度设计的陷阱。

请原谅这里笔者对 Rod Johnson 及其开发团队的一点善意批评。Rod Johnson 开始脱离实际开发，上升到框架设计时，如同大多数框架开发者一样，完美化的思想充斥了设计过程，就 Rod Johnson 针对 Struts 和 Webwork 的评论来看，其注意力过多的集中在设计上的完善，而忽略了实际开发中另外一个重要因素：易用性。

就 Spring MVC 本身而言，设计上的亮点固然无可非议，但在设计灵活性和易用性两者之间，Rod Johnson 的天平倒向了完美的设计。这导致 Spring MVC 使用起来感觉有点生涩，使用者赞叹其功能强大的同时，面对复杂的开发配置过程，难免也有点抓耳挠腮。

随着角色的转变（一个应用开发者到一个框架设计者），思考角度必然产生一些微妙的变化，希望 Rod Johnson 在 Spring MVC 后继版本的开发中，能更多站在使用者的角度出发，针对 Spring MVC 的易用性进行进一步改良。

就笔者在实际项目开发中的感觉来看，Webwork2 虽然也并不是及易上手，但在一定程度上，较好的兼顾了易用性。其代价是设计上不如 Spring MVC 完美，但我们开发中节省的脑细胞，应该可以弥补这一缺陷。

就 MVC 框架设计的角度来看，Webwork2 在目前的主流实现中（Struts、Webwork、Spring MVC）恰恰处于中间位置。其设计比 Struts 更加清晰，比 Spring 更加简练。

而从使用者角度而言，其接受难度也处于中等位置，比 Struts 稍高（Webwork 中也引入了 DI 等新的设计思想导致学习过程中需要更广泛的技术知识），比 Spring MVC 难度稍低。

而正是这一平衡点的把握，为 Webwork2 提供了与其他竞争对手一较高下的资本。

那么，Webwork 和 Spring Framework 的关系是怎样？

笔者曾经参与过一些讨论，很多开发者的观点是“如果选择了 Webwork，就不必再引入 Spring，两种框架同时使用必将导致认识上的冲突和技术感觉的混杂。”

然而，这里，正如之前所说，Spring 是一个高度组件化的框架。如果选择了 Webwork 作为开发基础框架，那么 Spring MVC 自然不必同时使用，但 Spring Framework 中的其他组件还是对项目开发颇有裨益。

站在 Web 应用层开发的角度而言，Spring 中最重要的组件，除了 MVC，还有另外一个令人欣赏的部分：持久层组件。持久层封装机制也许是 Spring 中应用级开发最有价值的部分，就笔者的视野来看，目前无论商业还是开源社区，尚无出其右者。

这里想要表达的意思就是：Webwork+Spring（Core+Persistence+Transaction Management）也许是目前最好的 Web 开发组合。对 Web 应用最重要的技术组成部分（MVC、持久层封装、事务管理）在这个组合中形成强大的合力，可以在很大程度上提高软件产品的质量和产出

效率。

当然，局限于笔者的知识范围，并不能保证这句话就一定正确，不同的出发点，固然有不同的看法。崇尚简单至上的方案（JSP+JavaBean+JDBC），以及皇家正统的企业级策略（JSP+SLSB+CMP），在不同的出发点上，也都是不错的选择。这里是笔者的看法，供大家参考。

下面我们就 Webwork+Spring（WS）组合的应用方式进行探讨。

首先来看，WS 组合中，Webwork 和 Spring 各司何职？

对一个典型的 Web 应用而言，MVC 贯穿了整个开发过程。选用 Webwork 的同时，也就确定了 MVC 框架的实现。

MVC 提供了纵向层面的操控。也就是说，Webwork 将纵向贯穿 Web 应用的设计，它担负了页面请求的接收、请求数据的规格统一、逻辑分发以及处理结果的返回这些纵向流程。

Spring 则在其中提供横向的支持。Model 层的情况比较典型，Webwork 将请求分发到 Action 之后，所能做的事情就是等待 Action 执行完毕然后返回最后的结果。至于 Action 内部如何处理，Webwork 并不关心。而这里，正是 Spring 大展身手的地方。

场景有点类似在 Pizzahut 用餐，伺服人员（Webwork）负责接受客户定餐(请求)，并将用户口头的定餐要求转化为统一的内部数据格式（Pizzahut 订单），然后将订单递交给厨师制作相应的餐点（执行 Action），之后再从厨房将餐点送到客户餐位。而厨师具体如何操作，伺服并不参与。

定单传递到厨师手上之后，厨师即按照烹饪流程（业务逻辑）开始制作餐点，烹饪的过程中，厨具必不可少，厨具可以选用乡间的柴灶、锅、碗、瓢、盆五件套，也可以选择自动化的配套厨具。

相信大家也已经明白我的意思，Spring 就是这里的自动化配套厨具，没有它固然也可以使用传统工具作出一份点心，但是借助这样的工具，你可以做的更好更快。

下面我们通过一个实例来演示 WS 组合的应用技术。

还是上面的用户登录示例。我们使用 Webwork 作为 MVC 框架，而在逻辑层（Action 层面），结合 Spring 提供的事务管理以及 Hibernate 封装，实现一个完整的登录逻辑。

界面部分并没有什么改变，主要的变化发生后台的执行过程。

第一个问题，Webwork 如何与 Spring 相融合？

假设 LoginAction 中调用 UserDAO.isValidUser 方法进行用户名、密码验证。最直接的想法，可能就是在 LoginAction 中通过代码获取 ApplicationContext 实例，然后通过 ApplicationContext 实例获取对应的 UserDAO Bean，再调用此实例的 isValidUser 方法。类似：

```
……
ApplicationContext ctx=new
       FileSystemXmlApplicationContext("bean.xml");
UserDAO userDAO = (UserDAO) ctx.getBean("userDAO");
if (userDAO.isValidUser(username,password)){
    ……
};
```

没问题，这样的确可以达到我们的目的。不过，这样的感觉似乎不是很好，太多的代码，更多的耦合（Action 与 Spring 相耦合）。

事实上，这样的实现方式并不能称之为“融合”，只是通过 API 调用的方式硬生生将两者捆绑。

我们期望能做到以下的效果：

```
……
if (userDAO.isValidUser(username,password)){
    ……
};
……
```

userDAO 自动由容器提供，而无需代码干涉。

这并不难实现，只是需要追加一个类库，以及一些配置上的修改。[7]

首先下载 http://www.ryandaigle.com/pebble/images/webwork2-spring.jar，并将其放入 WEB-INF\lib。
webwork2-spring.jar 中包含了 Spring 与 Webwork 融合所需的类库。

修改 web.xml，为 Web 应用增加相应的 Spring ContextLoaderListener 以及 webwork2-spring.jar 中的 ResolverSetupServletContextListener 配置。如下：

```
……
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<listener>
    <listener-class>
com.atlassian.xwork.ext.ResolverSetupServletContextListener
    </listener-class>
```

[7] 参见 Webwork Wiki Document：WebWork 2 Spring Integration

```
    </listener>
    ……
```

修改 xwork.xml，为 LoginAction 配置外部引用关系：

```
<xwork>
    <include file="webwork-default.xml" />

    <package name="default" extends="webwork-default"
    externalReferenceResolver="com.atlassian.xwork.ext.SpringServl
etContextReferenceResolver">
        <interceptors>
            <interceptor name="reference-resolver"
    class="com.opensymphony.xwork.interceptor.ExternalReferencesIn
terceptor" />

            <interceptor-stack name="WSStack">
                <interceptor-ref name="params" />
                <interceptor-ref name="model-driven" />
                <interceptor-ref name="reference-resolver" />
            </interceptor-stack>
        </interceptors>

        <action name="login" class="net.xiaxin.action.LoginAction">
            <external-ref name="userDAO">
                    userDAOProxy
            </external-ref>

            <result name="success" type="dispatcher">
                <param name="location">/main.jsp</param>
            </result>

            <result name="loginfail" type="dispatcher">
                <param name="location">/index2.jsp</param>
            </result>

            <result name="input" type="dispatcher">
                <param name="location">/index2.jsp</param>
            </result>

            <interceptor-ref name="WSStack" />
        </action>
    </package>
</xwork>
```

可见，interceptors 中增加了一个用于获得外部引用的拦截器 ExternalReferencesInterceptor。我们将其与 params、model-driven 组合为一个拦截器序列用于简化之后 Action 中的拦截器配置。

"login"中增加了一个外部引用"userDAO"，其值为 userDAOProxy，这是 Spring 中 userDAO Transaction Proxy 实例的引用名。

通过以上配置，我们实现了 Webwork、Spring 之间的融合。Webwork 将在运行期从 Spring Context 中获取资源引用，而 Spring 则将对此资源进行管理，并提供相关的调度服务（事务管理等）。

下面，我们在 LoginAction.java 中加入对应的 userDAO 申明：

```
public class LoginAction implements Action, ModelDriven {

    LoginInfo loginInfo = new LoginInfo();

    private UserDAO userDAO;

    public String execute() throws Exception {

        if (userDAO
            .isValidUser(loginInfo.getUsername(),
                         loginInfo.getPassword())
             ) {
            return SUCCESS;
        } else {
            return ERROR;
        }
    }

    public UserDAO getUserDAO() {
        return userDAO;
    }

    public void setUserDAO(UserDAO userDAO) {
        this.userDAO = userDAO;
    }

    public Object getModel() {
        return loginInfo;
    }
}
```

最后，在配置文件 applicationContext.xml 对 userDAO 进行配置，下面的操作与正常情

况下的 Spring 配置方法完全一样。

applicationContext.xml:

```
<beans>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">

        <property name="driverClassName">
            <value>org.gjt.mm.mysql.Driver</value>
        </property>

        <property name="url">
            <value>jdbc:mysql://localhost/sample</value>
        </property>

        <property name="username">
            <value>sysadmin</value>
        </property>

        <property name="password">
            <value>security</value>
        </property>

    </bean>

    <bean id="sessionFactory"
    class="org.springframework.orm.hibernate.LocalSessionFactoryBe
an">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
        <property name="mappingResources">
            <list>
                <value>net\xiaxin\db\entity\User.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    net.sf.hibernate.dialect.MySQLDialect
                </prop>
                <prop key="hibernate.show_sql">
```

```
                    true
                </prop>
            </props>
        </property>
    </bean>

    <bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionM
anager">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAO" class="net.xiaxin.db.dao.UserDAOImp">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAOProxy"
    class="org.springframework.transaction.interceptor.Transaction
ProxyFactoryBean">

        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>

        <property name="target">
            <ref local="userDAO" />
        </property>

        <property name="transactionAttributes">
            <props>
                <prop key="insert*">PROPAGATION_REQUIRED</prop>
                <prop
key="get*">PROPAGATION_REQUIRED,readOnly</prop>
                <prop key="is*">PROPAGATION_REQUIRED,readOnly</prop>

            </props>
        </property>
    </bean>

</beans>
```

UserDAOImp.java：

```
public class UserDAOImp extends HibernateDaoSupport implements
UserDAO {
    private SessionFactory sessionFactory;

    private static String hql = "from User u where u.username=? ";

    public boolean isValidUser(String username, String password) {

        List userList = this.getHibernateTemplate().find(hql,
username);//出于演示的简洁性考虑，这里并没有校验密码

        if (userList.size() > 0) {
            return true;
        }
        return false;
    }
}
```

UserDAO.java：

```
public interface UserDAO {
    public abstract boolean isValidUser(
                            String username,
                            String password
                            );
}
```

至于 Hibernate 的 OR 映射操作这里就不浪费篇幅进行描述。如果有兴趣，可以自行创建一个简单的 User 表（只需一个 username 字段即可）进行测试（可参见笔者的另一篇文档《Hibernate 开发指南》）。

可以看到，在 Webwork 中引入 Spring 组件，同时整合了两大框架的优势技术，这为我们的 Web 应用创造了极佳的框架基础。

Struts in Spring

勿庸置疑，Struts 是目前 Java Web MVC 框架中不争的王者。经过长达五年的发展，Struts 已经逐渐成长为一个稳定、成熟的框架，并且占有了 MVC 框架中最大的市场份额。

长达五年的设计延续性，自然导致其在某些技术特性上已经落后于新兴的 MVC 框架。面对 Spring MVC、Webwork2 这些设计更精密，扩展性更强的框架，Struts 受到了前所未有的挑战。

不过，纵然目前讨伐之声日起，甚至包括 Spring Framework 的作者 Rod Johnson 对 Struts 的评价也不甚高。但站在产品开发的角度而言，Struts 仍然是最稳妥的选择。

何谓"最稳妥"？并非完全意味着技术上的稳定性，而是指社会劳动力供给。感兴趣的读者可以去 51job、chinahr 这些人力资源网站上搜索一下 Java Web 程序员应聘简历。几乎所有 Java Web 开发人员都在简历上注明"精通 Struts"，且不论真正精通的能有多少，但就凭 Struts 这非凡的上镜率，其普及度也可见一斑。

这也就意味着，即使公司发生惨绝人寰的人事大变动，产品经理也不必过于惊惶失措，茫茫人海中，有大批的 Struts 们以供选择，只需好好考虑好新员工的业务培训如何开展即可，而对于技术延续性,则不必太过于担心。

这也就是 Struts 带来的战略性优势（对于公司而言，这一点往往是关键所在），其他 MVC 框架目前还无法在这点上与之并驾齐驱。

考虑到目前市面上已经有了众多的 Struts 书籍[8]。这里也就不再针对 Struts 的技术细节再加赘述，下面主要针对 Struts-Spring 组合应用进行探讨。

采用 Struts+Spring （SS）组合的原因就不再重复，具体可参见 Webwork2 in Spring 章节关于 WS 组合中，Webwork 与 Spring 之间关系的描述，而 SS 组合则与之类似。

与前面类似，这里我们通过 SS 组合，完成一个用户登录逻辑，除了 SS 组合之外，Hibernate 也有涉及，我们暂且称之为 SSH 组合**J**。

[8] 这里推荐笔者最欣赏的两本 Struts 书籍《Programming Struts》和《Jakarta-Struts Live》，前者的译版已经在国内发行，后者可从 theserverside.com 上免费下载。

首先，Struts 与 Spring 如何整合？

为了在 Struts 中加载 Spring Context，在 struts-config.xml 中增加如下部分：

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

Spring 在设计时就充分考虑到了与 Struts 的协同工作，通过内置的 Struts Plug-in 在两者之间提供了良好的结合点，这与 WS 组合不同，WS 组合需要嵌入来自第三方的类库。

通过 plug-in 我们实现了 Spring Context 的加载，不过仅仅加载 Context 并没有什么实际意义，我们还需要修改配置，将 Struts Action 交给 Spring 容器进行管理：

下面的 struts-config.xml 中包含了用户登录示例的相关配置：

```
<struts-config>
    <form-beans>
       <form-bean name="loginForm" type="net.xiaxin.bean.LoginForm" />
    </form-beans>

    <action-mappings>
       <action path="/login"
    type="org.springframework.web.struts.DelegatingActionProxy"
           name="loginForm">
           <forward name="success" path="/main.jsp" />
           <forward name="failure" path="/login.jsp" />
       </action>

       <!-- Struts Action Setting
           <action path="/login" type="net.xiaxin.action.LoginAction"
           name="loginForm">
           <forward name="success" path="/main.jsp" />
           <forward name="failure" path="/login.jsp" />
           </action>
       -->

    </action-mappings>

    <plug-in
    className="org.springframework.web.struts.ContextLoaderPlugIn">
```

```
        <set-property property="contextConfigLocation"
            value="/WEB-INF/applicationContext.xml" />
    </plug-in>

</struts-config>
```

可以看到，其中配置了一个 form bean "LoginForm"，在这点上，配置与传统 Struts 配置并没有什么不同。

而在 action 配置上，则出现了一些变化，上面的 action-mapping 配置中包含了 loginForm 的两种配置形式，第一种是面向 SS 组合改造后的形式，第二种（作为注释）是与之对应的传统 Struts 配置方式。

不难看出，我们试图在 action-mapping 中增加一个名为 loginForm 的 Action，传统方式中，直接将类名作为 action 节点的 type 属性，Struts 将根据 type 中的类名加载对应的 Action 实例。

而在面向 SS 组合的配置方式中，我们用 Spring 提供的 DelegatingActionProxy 作为 Action 的 type 属性。DelegatingActionProxy 同样是 org.apache.struts.action.Action 的一个子类，它将把调用请求转交给真正的 Action 实现。下面是 DelegatingActionProxy 的 execute 方法代码：

```
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    //获得实际的Action实例，并将请求转交
    Action delegateAction = getDelegateAction(mapping);
    return delegateAction.execute(mapping, form, request, response);
}
```

如此一来，Struts 在运行期加载的实际上是 DelegatingActionProxy，而 DelegatingActionProxy 则实现了针对实际 Action 的调用代理，Struts 最终调用的将是由 Spring 管理的 Action 实例。

SS 组合的玄机也正在与此，通过这样的方式，Spring 获得了对 Action 实例的管理权，它将对 Action 进行调度，并为 Struts 提供所需的 Action 实例。既然 Action 已经由 Spring 全权接管，那么我们就可以将此 Action 看作是 Spring 中的一个 Bean，它可享受 Spring 提供的所有服务（依赖注入、实例管理、事务管理等）。

与之对应，Spring Context 配置如下：

applicationContext.xml:

```
<?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.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">

      <property name="driverClassName">
          <value>org.gjt.mm.mysql.Driver</value>
      </property>

      <property name="url">
          <value>jdbc:mysql://localhost/sample</value>
      </property>

      <property name="username">
          <value>sysadmin</value>
      </property>

      <property name="password">
          <value>security</value>
      </property>

  </bean>

  <bean id="sessionFactory"
  class="org.springframework.orm.hibernate.LocalSessionFactoryBean"
>
      <property name="dataSource">
          <ref local="dataSource" />
      </property>
      <property name="mappingResources">
          <list>
              <value>net\xiaxin\db\entity\User.hbm.xml</value>
          </list>
      </property>
      <property name="hibernateProperties">
          <props>
              <prop key="hibernate.dialect">
                  net.sf.hibernate.dialect.MySQLDialect
              </prop>
              <prop key="hibernate.show_sql">true</prop>
          </props>
      </property>
```

```
    </bean>

    <bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionMana
ger">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAO" class="net.xiaxin.db.dao.UserDAOImp">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAOProxy"
    class="org.springframework.transaction.interceptor.TransactionPro
xyFactoryBean">

        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>

        <property name="target">
            <ref local="userDAO" />
        </property>

        <property name="transactionAttributes">
            <props>
                <prop key="insert*">PROPAGATION_REQUIRED</prop>
                <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
                <prop key="is*">PROPAGATION_REQUIRED,readOnly</prop>
            </props>
        </property>
    </bean>

    <bean name="/login" class="net.xiaxin.action.LoginAction"
        singleton="false">
        <property name="userDAO">
            <ref bean="userDAOProxy" />
        </property>
    </bean>
```

```
</beans>
```

最后一个 Bean 的配置是关键，这个名为"/login"的 Bean 与 Struts 中的

```
<action path="/login" ……>
    ……
</action>
```

节点相对应。

这样，Spring Bean Name 与 Struts Action Path 相关联，当 Struts 加载对应的 Action 时，DelegatingActionProxy 就根据传入的 path 属性，在 Spring Context 寻找对应的 bean，并将其实例返回给 Struts（参见 Spring 中 org.springframework.web.struts.DelegatingActionUtils. determineActionBeanName 方法的实现代码，DelegatingActionProxy 将调用此方法获得 Struts Action 对应的 Bean name）。

与此同时，还可以看到，"/login" bean 中包含了一个 userDAO 引用，Spring 在运行期将根据配置为其提供 userDAO 实例，以及围绕 userDAO 的事务管理服务。这样一来，对于 Struts 开发而言，我们既可以延续 Struts 的开发流程，也可以享受 Spring 提供的事务管理服务。

而 bean 的另外一个属性 singleton="false"，指明了 Action 的实例获取方式为每次重新创建。这也解决了 Struts 中令人诟病的线程安全问题（Struts 中，由一个 Action 实例处理所有的请求，这就导致了类公用资源在并发请求中的线程同步问题。）

至此，SS 组合已经将 Struts MVC 以及 Spring 中的 Bean 管理、事务管理融为一体。如果算上 userDAO 中的 Hibernate 部分，我们就获得了一个全面、成熟、高效、自顶而下的 Web 开发框架。

applicationContext.xml 其余部分配置的语义，以及 UserDAO、UserDAOImp 实现代码均与 Webwork2 in Spring 章节中一致。

struts-config.xml 和 applicationContext.xml 上面已经列出，下面是示例中其余新增的部分：

web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

   <servlet>
      <servlet-name>action</servlet-name>
      <servlet-class>
         org.apache.struts.action.ActionServlet
      </servlet-class>
      <init-param>
```

```
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <init-param>
            <param-name>debug</param-name>
            <param-value>2</param-value>
        </init-param>
        <init-param>
            <param-name>detail</param-name>
            <param-value>2</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

login.jsp:

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>

<html:html>

<html:form action="login">
        <p align="center">Login</p>

        Username:
        <html:text property="username" size="25"
maxlength="20"/><br/>

        Password:
        <html:password property="password" /><br/>

        <p align="center">
        <html:submit />
            <html:cancel />
```

```
        </p>
</html:form>

</html:html>
```

LoginAction.java

```
public class LoginAction extends Action {
    private static final String SUCCESS = "success";

    private static final String FAILURE = "failure";

    private UserDAO userDAO;

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

        LoginForm loginForm = (LoginForm) form;

        if (userDAO.isValidUser(loginForm.getUsername(), loginForm
                .getPassword())) {
            return mapping.findForward(SUCCESS);
        } else {
            return mapping.findForward(FAILURE);
        }

    }

    public UserDAO getUserDAO() {
        return userDAO;
    }

    public void setUserDAO(UserDAO userDAO) {
        this.userDAO = userDAO;
    }
}
```

LoginForm.java

```
public class LoginForm extends ActionForm{
    private String username;
    private String password;

```

```
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

至于 UserDAO，UserDAOImp 请参见 Webwork2 in Spring 章节中的描述。

数据持久层

——在数据持久层的杰出贡献，可能是 Spring 最为闪亮的优点。

事务管理

对于 J2EE 应用程序而言，事务的处理一般有两种模式：

1. 依赖特定事务资源的事务处理
   这是应用开发中最常见的模式，即通过特定资源提供的事务机制进行事务管理。如通过 JDBC、JTA 的 rollback、commit 方法；Hibernate Transaction 的 rollback、commit 方法等。这种方法大家已经相当熟悉。

2. 依赖容器的参数化事务管理
   通过容器提供的集约式参数化事务机制，实现事务的外部管理，如 EJB 中的事务管理模式。

   如，下面的 EJB 事务定义中，将 SessionBean MySession 的 doService 方法定义为 Required。

   也就是说，当 MySession.doServer 方法被某个线程调用时，容器将此线程纳入事务管理容器，方法调用过程中如果发生异常，当前事务将被容器自动回滚，如果方法正常结束，则容器将自动提交当前事务。

```
<container-transaction >
   <method >
      <ejb-name>MySession</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>doService</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </method>
   <trans-attribute>Required</trans-attribute>
</container-transaction>
```

容器管理的参数化事务为程序开发提供了相当的灵活性，同时因为将事务委托给容器进行管理，应用逻辑中无需再编写事务代码，大大节省了代码量（特别是针对需要同时操作多个事务资源的应用），从而提高了生产率。

然而，使用 EJB 事务管理的代价相当高昂，撇开 EJB 容器不菲的价格，EJB 的学习成本，部署、迁移、维护难度，以及容器本身带来的性能开销（这往往意味着需要更高的硬件配置）都给我们带来了相当的困惑。此时事务管理所带来的优势往往还不能抵消上面这些负面影响。

Spring 事务管理能给我们带来什么?

对于传统的基于特定事务资源的事务处理而言（如基于 JDBC 的数据库访问），Spring 并不会对其产生什么影响，我们照样可以成功编写并运行这样的代码。同时，

Spring 还提供了一些辅助类可供我们选择使用，这些辅助类简化了传统的数据库操作流程，在一定程度上节省了工作量，提高了编码效率。

对于依赖容器的参数化事务管理而言，Spring 则表现出了极大的价值。Spring 本身也是一个容器，只是相对 EJB 容器而言，Spring 显得更为轻便小巧。我们无需付出其他方面的代价，即可通过 Spring 实现基于容器的事务管理（本质上来讲，Spring 的事务管理是基于动态 AOP）。

下面这段 xml 配置片断展示了 Spring 中的事务设定方式：

```
<beans>
  <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">

      <property name="driverClassName">
          <value>org.gjt.mm.mysql.Driver</value>
      </property>

      <property name="url">
          <value>jdbc:mysql://localhost/sample</value>
      </property>

      <property name="username">
          <value>user</value>
      </property>

      <property name="password">
          <value>mypass</value>
      </property>
  </bean>

  <bean id="transactionManager"
         class="org.springframework.jdbc.datasource.DataSourceTr
ansactionManager">
      <property name="dataSource">
          <ref local="dataSource" />
      </property>
  </bean>

  <bean id="userDAO" class="net.xiaxin.dao.UserDAO">
      <property name="dataSource">
          <ref local="dataSource" />
      </property>
  </bean>
```

```
  <bean id="userDAOProxy"
         class="org.springframework.transaction.interceptor.Tran
sactionProxyFactoryBean">
      <property name="transactionManager">
         <ref bean="transactionManager" />
      </property>

      <property name="target">
         <ref local="userDAO" />
      </property>

      <property name="transactionAttributes">
         <props>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="get*">
                  PROPAGATION_REQUIRED,readOnly
            </prop>
         </props>
      </property>
  </bean>
</beans>
```

配置中包含了 dataSource，transactionManager 等资源定义。这些资源都为一个名为 userDAOProxy 的 TransactionProxyFactoryBean 服务，而 userDAOProxy 则对包含实际数据逻辑的 userDAO 进行了事务性封装。

可以看到，在 userDAOProxy 的"transactionAttributes"属性中，我们定义了针对 userDAO 的事务策略，即将所有名称以 insert 开始的方法（如 UserDAO.insertUser 方法）纳入事务管理范围。如果此方法中抛出异常，则 Spring 将当前事务回滚，如果方法正常结束，则提交事务。

而对所有名称以 get 开始的方法（如 UserDAO.getUser 方法）则以只读的事务处理机制进行处理。（设为只读型事务，可以使持久层尝试对数据操作进行优化，如对于只读事务 Hibernate 将不执行 flush 操作，而某些数据库连接池和 JDBC 驱动也对只读型操作进行了特别优化。）

结合上面这段申明带来的感性认知，看看 Spring 的事务管理机制与 EJB 中事务管理有何不同，或者有何优势。这里自然有许多方面可以比较，不过，笔者认为其中最为关键的两点是：

1. Spring 可以将任意 Java Class 纳入事务管理
   这里的 UserDAO 只是我们编写的一个普通 Java Class，其中包含了一些基本的数据应用逻辑。通过 Spring，我们即可简单的实现事务的可配置化。也就是说，我们可以随意为某个类的某个方法指定事务管理机制。

与之对比，如果使用 EJB 容器提供的事务管理功能，我们不得不按照 EJB 规范编将 UserDAO 进行改造，将其转换为一个标准的 EJB。

2. Spring 事务管理并不依赖特定的事务资源。
EJB 容器必须依赖于 JTA 提供事务支持。而 Spring 的事务管理则支持 JDBC、JTA 等多种事务资源。这为我们提供了更多的选择，从而也使得我们的系统部署更加灵活。

对Spring事务管理机制进行简单分析之后，我们将结合持久层封装的具体事务应用机制，对Spring中的事务管理进行更具实效的探讨。

持久层封装

JDBC

Spring对JDBC进行了良好的封装，通过提供相应的模板和辅助类，在相当程度上降低了JDBC操作的复杂性。并且得益于Spring良好的隔离设计，JDBC封装类库可以脱离Spring Context独立使用，也就是说，即使系统并没有采用Spring作为结构性框架，我们也可以单独使用Spring的JDBC部分（spring-dao.jar）来改善我们的代码。

作为对比，首先让我们来看一段传统的JDBC代码：

```
Connection conn =null;
Statement stmt = null;
try {
  conn = dataSource.getConnection();
  stmt = con.createStatement();
  stmt.executeUpdate("UPDATE user SET age = 18 WHERE id = 'erica'");
} finally {
  if (stmt != null) {
      try {
          stmt.close();
      } catch (SQLException ex) {
          logger.warn("Exception in closing JDBC Statement", ex);
      }
  }
  if (conn != null) {
      try {
          conn.close();
      } catch (SQLException ex) {
          logger.warn("Exception in closing JDBC Connection", ex);
      }
  }
}
```

类似上面的代码非常常见。为了执行一个SQL语句，我们必须编写22行代码，而其中21行与应用逻辑并无关联，并且，这样的代码还会在系统其他地方（也许是每个需要数据库访问的地方）重复出现。

于是，大家开始寻找一些设计模式以改进如此的设计，Template模式的应用是其中一种典型的改进方案。

Spring的JDBC封装，很大一部分就是借助Template模式实现，它提供了一个优秀的JDBC模板库，借助这个工具，我们可以简单有效的对传统的JDBC编码方式加以改进。

下面是借助Spring JDBC Template修改过的代码，这段代码完成了与上面代码相同的功能。

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate.update("UPDATE user SET age = 10 WHERE id = 'erica'");
```

可以看到，两行代码完成了上面需要19行代码实现的功能。所有冗余的代码都通过合理的抽象汇集到了JdbcTemplate中。

无需感叹，借助Template模式，我们大致也能实现这样一个模板，不过，Spring的设计者已经提前完成了这一步骤。org.springframework.jdbc.core.JdbcTemplate中包含了这个模板实现的代码，经过Spring设计小组精心设计，这个实现可以算的上是模板应用的典范。特别是回调（CallBack）的使用，使得整个模板结构清晰高效。值得一读。

Tips:实际开发中，可以将代码中硬编码的SQL语句作为Bean的一个String类型属性，借助DI机制在配置文件中定义，从而实现SQL的参数化配置。

再对上面的例子进行一些改进，通过PrepareStatement执行update操作以避免SQL Injection 漏洞 [9]：

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate
    .update(
        "UPDATE user SET age = ? WHERE id = ?",
        new PreparedStatementSetter() {
            public void setValues(PreparedStatementSetter ps)
                throws SQLException {
                ps.setInt(1, 18);
                ps.setString(2, "erica");
            }
        }
  );
```

可以看到，上面引用了update方法的另一个版本，传入的参数有两个，第一个用于创建PreparedStatement的SQL。第二个参数是为PreparedStatement设定参数的PreparedStatementSetter。

第二个参数的使用方法比较独到，我们动态新建了一个PreparedStatementSetter类，并实现了这个抽象类的setValues方法。之后将这个类的引用作为参数传递给update。update接受参数之后，即可调用第二个参数提供的方法完成PreparedStatement的初始化。

Spring JDBC Template中大量使用了这样的Callback机制，这带来了极强的灵活性和扩展性。

上面演示了update方法的使用（同样的操作适用于update、insert、delete）。下面是一个查询的示例。

```
final List userList = new ArrayList();
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate
    .query(
        "SELECT name, sex, address FROM user WHERE age > 18",
```

[9] SQL Injection： SQL 语句中直接引入参数值而导致的系统漏洞，具体请参见以下论文：
http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenceandWhyItMatters.php

```
        new RowCallbackHandler() {
            public void processRow(ResultSet rs) throws SQLException {
                User user = new User();
                user.setId(rs.getString("name"));
                user.setSex(rs.getString("sex"));
                user.setAddress(rs.getString("address"));
                userList.add(product);
            }
        }
);
```

这里传入query方法的有两个参数，第一个是Select查询语句，第二个是一个RowCallbackHandler实例，我们通过RowCallbackHandler对Select语句得到的每行记录进行解析，并为其创建一个User数据对象。实现了手动的OR映射。

此外，我们还可以通过JdbcTemplate.call方法调用存储过程。

query、update方法还有其他很多不同参数版本的实现，具体调用方法请参见Spring JavaDoc。

JdbcTemplate与事务

上例中的JdbcTemplate操作采用的是JDBC默认的AutoCommit模式，也就是说我们还无法保证数据操作的原子性（要么全部生效，要么全部无效），如：

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate.update("UPDATE user SET age = 10 WHERE id = 'erica'");
jdbcTemplate.update("UPDATE user SET age = age+1 WHERE id = 'erica'");
```

由于采用了AutoCommit模式，第一个update操作完成之后被自动提交，数据库中"erica"对应的记录已经被更新，如果第二个操作失败，我们无法使得整个事务回滚到最初状态。对于这个例子也许无关紧要，但是对于一个金融帐务系统而言，这样的问题将导致致命错误。

为了实现数据操作的原子性，我们需要在程序中引入事务逻辑，在JdbcTemplate中引入事务机制，在Spring中有两种方式：

1. 代码控制的事务管理
2. 参数化配置的事务管理

下面就这两种方式进行介绍。

u 代码控制的事务管理

首先，进行以下配置，假设配置文件为（Application-Context.xml）：

```
<beans>
    <bean id="dataSource"
```

```
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName">
            <value>net.sourceforge.jtds.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>
        </property>
        <property name="username">
            <value>test</value>
        </property>
        <property name="password">
            <value>changeit</value>
        </property>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransac
tionManager">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
    </bean>

    <bean id="userDAO" class="net.xiaxin.dao.UserDAO">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
        <property name="transactionManager">
            <ref local="transactionManager" />
        </property>
    </bean>
</beans>
```

配置中包含了三个节点:

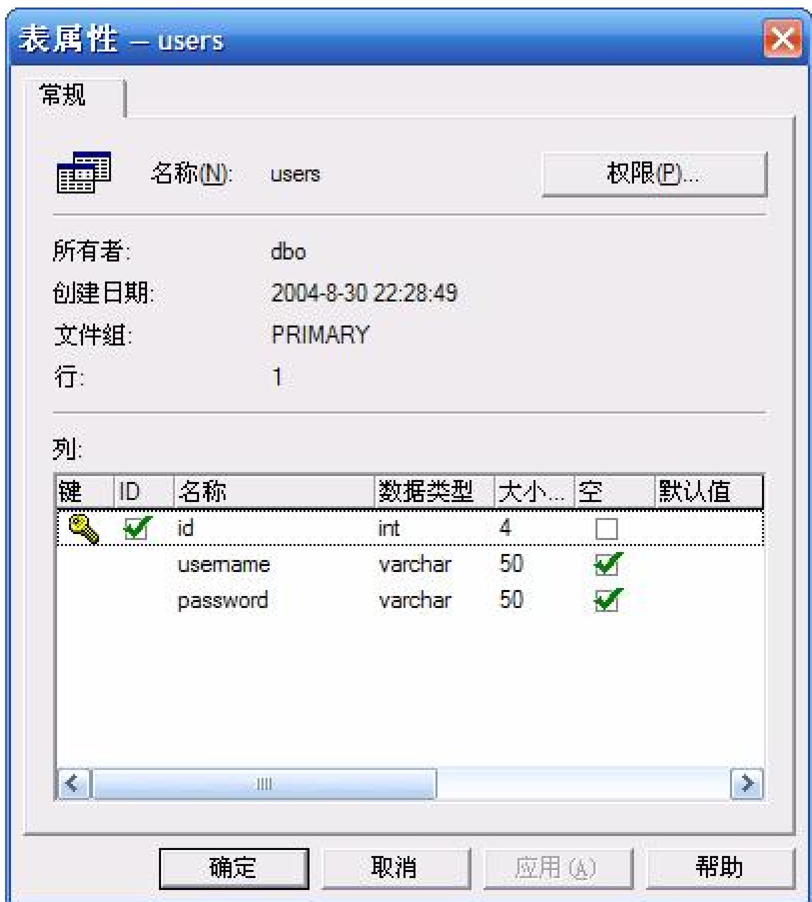
- dataSource
  这里我们采用了apache dhcp组件提供的DataSource实现，并为其配置了JDBC驱动、数据库URL、用户名和密码等参数。
- transactionManager
  针对JDBC DataSource类型的数据源，我们选用了
  DataSourceTransactionManager
  作为事务管理组件。

如果需要使用基于容器的数据源（JNDI），我们可以采用如下配置：

```
<bean id="dataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
      <value>jdbc/sample</value>
  </property>
</bean>
<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTrans
actionManager"
  />
```

- userDAO
  申明了一个UserDAO Bean，并为其指定了dataSource和transactionManger资源。

测试库表非常简单：



UserDAO对应的代码如下：

```
public class UserDAO {

    private DataSource dataSource;
```

```
    private PlatformTransactionManager transactionManager;

    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }

    public void setTransactionManager(PlatformTransactionManager
transactionManager) {
        this.transactionManager = transactionManager;
    }

    public DataSource executeTestSource() {
        return dataSource;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insertUser() {
        TransactionTemplate tt =
            new TransactionTemplate(getTransactionManager());

        tt.execute(new TransactionCallback() {

            public Object doInTransaction(TransactionStatus status) {
                JdbcTemplate jt = new JdbcTemplate(executeTestSource());
                jt.update(
                    "insert into users (username) values ('xiaxin');");
                jt.update(
                    "insert into users (id,username) values(2,
'erica');");
                return null;
            }
        });
    }
}
```

可以看到，在insertUser方法中，我们引入了一个新的模板类：

org.springframework.transaction.support.TransactionTemplate。

TransactionTemplate封装了事务管理的功能，包括异常时的事务回滚，以及操作成功后的事务提交。和JdbcTemplate一样，它使得我们无需在琐碎的try/catch/finally代码中徘徊。

在doInTransaction中进行的操作，如果抛出未捕获异常将被自动回滚，如果成功执行，

则将被自动提交。

这里我们故意制造了一些异常来观察数据库操作是否回滚(通过在第二条语句中更新自增ID字段故意触发一个异常):

编写一个简单的TestCase来观察实际效果:

```
InputStream is = new FileInputStream("Application-Context.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
UserDAO userDAO = (UserDAO) factory.getBean("userDAO");
userDAO.insertUser();
```

相信大家多少觉得上面的代码有点凌乱，Callback类的编写似乎也有悖于日常的编程习惯(虽然笔者觉得这一方法比较有趣，因为它巧妙的解决了笔者在早期自行开发数据访问模板中曾经遇到的问题)。

如何进一步避免上面这些问题? Spring 的容器事务管理机制在这里即体现出其强大的能量。

- **参数化配置的事务管理**

在上面的Application-Context.xml增加一个事务代理(UserDAOProxy)配置，同时，由于事务由容器管理，UserDAO不再需要TransactionManager设定，将其移除:

```
<bean id="UserDAOProxy"
        class="org.springframework.transaction.interceptor.Transac
tionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="target">
        <ref local="userDAO" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>

<bean id="userDAO" class="net.xiaxin.dao.UserDAO">
    <property name="dataSource">
        <ref local="dataSource" />
    </property>
</bean>
```

上面的配置中，UserDAOProxy节点配置了一个针对userDAO bean的事务代理（由target属性指定）。

通过transactionAttributes属性，我们指定了事务的管理策略，即对所有以insert开头的方法进行事务管理，如果被管理方法抛出异常，则自动回滚方法中的事务，如果成功执行，则在方法完成之后进行事务提交。另一方面，对于其他方法（通过通配符*表示），则进行只读事务管理，以获得更好的性能。

与之对应，UserDAO.insertUser的代码修改如下：

```
public void insertUser(RegisterInfo regInfo) {
    JdbcTemplate jt = new JdbcTemplate(executeTestSource());
    jt.update("insert into users (username) values ('xiaxin');");
    jt.update("insert into users (id,username) values (2,'erica');");
}
```

测试代码修改如下：

```
InputStream is = new FileInputStream("Application-Context.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);

//注意这里须通过代理Bean"userDAOProxy"获得引用，而不是直接getBean("userDAO")

//此外这里还存在一个有关强制转型的潜在问题，请参见Hibernate in Spring一节后
//关于强制转型的补充描述。
UserDAO userDAO = (UserDAO) factory.getBean("userDAOProxy");
userDAO.insertUser();
```

可以看到，insertUser变得非常简洁。数据逻辑清晰可见，对比前面代码控制的事务管理，以及传统的JDBC操作，相信大家会有一些霍然开朗的感觉。

细心的读者会说，这只不过将代码转移到了配置文件，并没有减少太多的工作量。这点区别也许并不重要，从应用维护的角度而言，配置化的事务管理显然更具优势。何况，实际开发中，如果前期设计细致，方法的事务特性确定之后一般不会发生大的变动，之后频繁的维护过程中，我们只需面对代码中的数据逻辑即可。

上面我们结合JdbcTemplate介绍了Spring中的模板操作以及事务管理机制。Spring作为一个开放式的应用开发平台。同时也针对其他组件提供了良好的支持。在持久层，Spring提供面向了Hibernate、ibatis和JDO的模板实现，同样，这些实现也为我们的开发提供了强有力的支持。

下面我们就hibernate、ibatis这两种主流持久层框架在Spring中的使用进行介绍。至于JDO，由于实际开发中使用并不广泛（实际上笔者觉得JDO前景堪忧），这里也就不重点介绍，有兴趣的读者可参见Spring-Reference中的相关章节。

Hibernate in Spring

Hibernate在开源的持久层框架中无疑是近期最为鲜亮的角色，其作者甚至被邀请加入新版EJB设计工作之中，足见Hibernate设计的精彩贴切。关于Hibernate的使用，在笔者的另外一篇文档中进行了探讨：

《Hibernate开发指南》 http://www.xiaxin.net/Hibernate_DEV_GUIDE.rar。

下面主要就Hibernate在Spring中的应用加以介绍，关于Hibernate本身就不多加描述。

另外考虑到Spring对容器事务的良好支持，笔者建议在基于Spring Framework的应用开发中，尽量使用容器管理事务，以获得数据逻辑代码的最佳可读性。下面的介绍中，将略过代码控制的事务管理部分，而将重点放在参数化的容器事务管理应用。代码级事务管理实现原理与上面JdbcTemplate中基本一致，感兴趣的读者可以参见Spring-Reference中的相关内容。

出于简洁，我们还是沿用上面的示例。首先，针对Hibernate，我们需要进行如下配置：

Hibernate-Context.xml:

```
<beans>
  <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName">
          <value>net.sourceforge.jtds.jdbc.Driver</value>
      </property>
      <property name="url">
          <value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>
      </property>
      <property name="username">
          <value>test</value>
      </property>
      <property name="password">
          <value>changeit</value>
      </property>
  </bean>

  <bean id="sessionFactory"
  class="org.springframework.orm.hibernate.LocalSessionFactoryBean"
>
      <property name="dataSource">
          <ref local="dataSource" />
      </property>
      <property name="mappingResources">
          <list>
              <value>net/xiaxin/dao/entity/User.hbm.xml</value>
          </list>
```

```
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    net.sf.hibernate.dialect.SQLServerDialect
                </prop>
                <prop key="hibernate.show_sql">
                    true
                </prop>
            </props>
        </property>
    </bean>

    <bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionMana
ger">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAO" class="net.xiaxin.dao.UserDAO">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAOProxy"
    class="org.springframework.transaction.interceptor.TransactionPro
xyFactoryBean">

        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>

        <property name="target">
            <ref local="userDAO" />
        </property>

        <property name="transactionAttributes">
            <props>
                <prop key="insert*">PROPAGATION_REQUIRED</prop>
                <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            </props>
```

```
    </property>
  </bean>
</beans>
```

与上面JDBC中的配置相对比，区别主要在于：

1. SessionFactory的引入

   Hibernate中通过SessionFactory创建和维护Session。Spring对SessionFactory的配置也进行了整合，无需再通过Hibernate.cfg.xml对SessionFactory进行设定。

   SessionFactory节点的mappingResources属性包含了映射文件的路径，list节点下可配置多个映射文件。

   hibernateProperties节点则容纳了所有的属性配置。

   可以对应传统的Hibernate.cfg.xml文件结构对这里的SessionFactory配置进行解读。

2. 采用面向Hibernate的TransactionManager实现：
   org.springframework.orm.hibernate.HibernateTransactionManager

可以看到，对于事务管理配置，基本与上一章节中相同。

对应刚才的Users表，建立如下映射类：

User.java:

```
/**
 * @hibernate.class table="users"
 */
public class User {

    public Integer id;

    public String username;

    public String password;

    /**
     * @hibernate.id
     *     column="id"
     *     type="java.lang.Integer"
     *     generator-class="native"
     */
    public Integer getId() {
        return id;
```

```
    }

    public void setId(Integer id) {
        this.id = id;
    }

    /**
     * @hibernate.property column="password" length="50"
     */
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    /**
     * @hibernate.property column="username" length="50"
     */
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

上面的代码中，通过 xdoclet 指定了类/表；属性/字段的映射关系，通过 xdoclet ant task 我们可以根据代码生成对应的 user.hbm.xml 文件。具体细节请参见《hibernate 开发指南》一文。

下面是生成的 user.hbm.xml：

```
<hibernate-mapping>
    <class
        name="net.xiaxin.dao.entity.User"
        table="users"
        dynamic-update="false"
        dynamic-insert="false"
    >
        <id
            name="id"
            column="id"
```

```
            type="java.lang.Integer"
        >
            <generator class="native">
            </generator>
        </id>

        <property
            name="password"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="password"
            length="50"
        />

        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="50"
        />
    </class>
</hibernate-mapping>
```

UserDAO.java:

```
public class UserDAO extends HibernateDaoSupport implements IUserDAO
{
    public void insertUser(User user) {
        getHibernateTemplate().saveOrUpdate(user);
    }

}
```

看到这段代码想必会有点诧异，似乎太简单了一点……，不过这已经足够。短短一行代码我们已经实现了与上一章中示例相同的功能，这也正体现了Spring+Hibernate的威力所在。

上面的UserDAO实现了自定义的IUserDAO接口（这里的IUserDAO接口仅包含insertUser方法的定义，不过除此之外，它还有另一层含义，见下面的代码测试部分），

并扩展了抽象类：

HibernateDaoSupport

HibernateSupport实现了HibernateTemplate和SessionFactory实例的关联。

与JdbcTemplate类似，HibernateTemplate对Hibernate Session操作进行了封装，而HibernateTemplate.execute方法则是一封装机制的核心，感兴趣的读者可以研究一下其实现机制。

借助HibernateTemplate我们可以脱离每次数据操作必须首先获得Session实例、启动事务、提交/回滚事务以及烦杂的try/catch/finally的繁琐操作。从而获得以上代码中精干集中的逻辑呈现效果。

对比下面这段实现了同样功能的Hibernate原生代码，想必更有体会：

```
Session session
try {
  Configuration config = new Configuration().configure();

  SessionFactory sessionFactory =
      config.buildSessionFactory();
  session = sessionFactory.openSession();

  Transaction tx = session.beginTransaction();

  User user = new User();
  user.setName("erica");
   user.setPassword("mypass");
  session.save(user);

  tx.commit();


} catch (HibernateException e) {
      e.printStackTrace();
      tx.rollback();
}finally{
      session.close();
}
```

测试代码：

```
InputStream is = new FileInputStream("Hibernate-Context.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
IUserDAO userDAO = (IUserDAO)factory.getBean("userDAOProxy");
```

```
User user = new User();
user.setUsername("erica");
user.setPassword("mypass");

userDAO.insertUser(user);
```

这段代码似乎并没有什么特殊，但有一个细微之处：

```
IUserDAO userDAO = (IUserDAO)factory.getBean("userDAOProxy");
```

这里并没有直接用UserDAO对获得的Bean实例进行强制转型。这与上面JdbcTemplate的测试代码不同。并非完全出自设计上的考虑，这里情况有些特殊，我们可以尝试一下用UserDAO类对bean实例进行强制转型，不过将得到一个ClassCastException，程序异常中止。

为什么会出现这样的问题？是不是只有在使用Hibernate才会出现这样的问题？事实并非如此，如果对上面基于JdbcTempate的UserDAO进行改造，使之实现IUserDAO接口，同样的问题也将会出现。IUserDAO接口本身非常简单（仅包含一个insertUser方法的定义），显然也不是导致异常的原因所在。

原因在于Spring的AOP实现机制，前面曾经提及，Spring中的事务管理实际上是基于动态AOP机制实现，为了实现动态AOP，Spring在默认情况下会使用Java Dynamic Proxy，但是，Dynamic Proxy要求其代理的对象必须实现一个接口，该接口定义了准备进行代理的方法。而对于没有实现任何接口的Java Class，需要采用其他方式，Spring通过CGLib[10]实现这一功能。

当UserDAO没有实现任何接口时（如JdbcTemplate示例中）。Spring通过CGLib对UserDAO进行代理，此时getBean返回的是一个继承自UserDAO类的子类实例，可以通过UserDAO对其强制转型。而当UserDAO实现了IUserDAO接口之后，Spring将通过Java Dynamic Proxy机制实现代理功能，此时返回的Bean，是通过java.lang.reflect.Proxy.newProxyInstance方法创建的IUserDAO接口的一个代理实现，这个实例实现了IUserDAO接口，但与UserDAO类已经没有继承关系，因此无法通过UserDAO强制转型。

由于此问题牵涉到较为底层的代理机制实现原理，下面的AOP章节中我们再进行详细探讨。

实际开发中，应该面向接口编程，通过接口来调用Bean提供的服务。

[10] CGLib 可以在运行期对 Class 行为进行修改。由于其功能强大，性能出众，常常被作为 Java Dynamic Proxy 之外的动态 Proxy 模式的实现基础。在 Spring、Hibernate 中都用到了 CGLib 类库。

ibatis in Spring

与Hibernate类似，ibatis也是一个ORM解决方案，不同的是两者各有侧重。

Hibernate提供了Java对象到数据库表之间的直接映射，开发者无需直接涉及数据库操作的实现细节，实现了一站式的ORM解决方案。

而ibatis则采取了另一种方式，即提供Java对象到SQL（面向参数和结果集）的映射实现，实际的数据库操作需要通过手动编写SQL实现。

在Java ORM世界中，很幸运，我们拥有了这两个互补的解决方案，从而使得开发过程更加轻松自如。

关于两者的对比，请参见笔者另一份文档：

《ibatis开发指南》http://www.xiaxin.net/ibatis_Guide.rar

与Hibernate in Spring一节类似，这里我们重点探讨Spring框架下的ibatis应用，特别是在容器事务管理模式下的ibatis应用开发。

针对ibatis，Spring配置文件如下：

Ibatis-Context.xml:

```
<?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.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName">
          <value>net.sourceforge.jtds.jdbc.Driver</value>
      </property>
      <property name="url">
          <value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>
      </property>
      <property name="username">
          <value>test</value>
      </property>
      <property name="password">
          <value>changeit</value>
      </property>
  </bean>

  <bean id="sqlMapClient"
  class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
```

```
        <property name="configLocation">
            <value>SqlMapConfig.xml</value>
        </property>
    </bean>

    <bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactio
nManager">
        <property name="dataSource"><ref
local="dataSource"/></property>
    </bean>

    <bean id="userDAO" class="net.xiaxin.dao.UserDAO">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
        <property name="sqlMapClient">
            <ref local="sqlMapClient" />
        </property>
    </bean>

    <bean id="userDAOProxy"
    class="org.springframework.transaction.interceptor.TransactionPro
xyFactoryBean">

        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>

        <property name="target">
            <ref local="userDAO" />
        </property>

        <property name="transactionAttributes">
            <props>
                <prop key="insert*">PROPAGATION_REQUIRED</prop>
                <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            </props>
        </property>
    </bean>
</beans>
```

对比之前的JDBC和Hibernate配置，可以看到：

1. sqlMapClient节点

类似SessionFactory之与Hibernate，这里我们引入了针对ibatis SqlMap的SqlMapClientFactoryBean配置。SqlMapClient对于ibatis的意义类似于Session与Hibernate以及Connection与JDBC，这里的sqlMapClient节点实际上配置了一个sqlMapClient的创建工厂类。

configLocation属性配置了ibatis映射文件的名称。

2. transactionManager节点

这里我们的transactionManager配置与之前JDBC示例中相同，都采用了DataSourceTransactionManager，这与Hibernate有些差异。

3. userDAO节点

对应的，UserDAO需要配置两个属性，sqlMapClient和DataSource，sqlMapClient将从指定的DataSource中获取数据库连接。

其他配置基本上与JDBC示例中相同，这里就不加赘述。

本例中Ibatis映射文件非常简单：

sqlMapConfig.xml:

```
<sqlMapConfig>
 <sqlMap resource="net/xiaxin/dao/entity/user.xml"/>
</sqlMapConfig>
```

net/xiaxin/dao/entity/user.xml

```
<sqlMap namespace="User">
    <typeAlias alias="user" type="net.xiaxin.dao.entity.User" />

    <insert id="insertUser" parameterClass="user">
        INSERT INTO users ( username, password) VALUES ( #username#,
#password# )
    </insert>

</sqlMap>
```

与Hibernate示例中类似，UserDAO.java同样简洁：

```
public class UserDAO extends SqlMapClientDaoSupport implements
IUserDAO {

    public void insertUser(User user) {
        getSqlMapClientTemplate().update("insertUser", user);
    }
}
```

SqlMapClientDaoSupport（如果使用ibatis 1.x版本，对应支持类是SqlMapDaoSupport）是Spring中面向ibatis的辅助类，它负责调度DataSource、

SqlMapClientTemplate（对应ibatis 1.x版本是SqlMapTemplate）完成ibatis操作，而DAO则通过对此类进行扩展获得上述功能。上面配置文件中针对UserDAO的属性设置部分，其中的属性也是继承自于这个基类。
SqlMapClientTemplate对传统SqlMapClient调用模式进行了封装，简化了上层访问代码。

User.java沿用之前Hibernate示例中的代码。
测试代码也几乎相同：

```
InputStream is = new FileInputStream("Ibatis-Context.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
IUserDAO userdao = (IUserDAO)factory.getBean("userDAOProxy");

User user = new User();
user.setUsername("Sofia");
user.setPassword("mypass");

userdao.insertUser(user);
```

Aspect Oriented Programming

AOP概念

Aspect Oriented Programming（AOP）是近来较为热门的一个话题。AOP，国内大致译作"面向方面编程"。

"面向方面编程",这样的名字并不是非常容易理解，且容易产生一些误导。笔者不止一次听到类似"OOP/OOD[11]即将落伍，AOP是新一代软件开发方式"这样的发言。显然，发言者并没有理解AOP的含义。

Aspect，没错，的确是"方面"的意思。不过，华语传统语义中的"方面"，大多数情况下指的是一件事情的不同维度、或者说不同角度上的特性，比如我们常说："这件事情要从几个方面来看待"，往往意思是：需要从不同的角度来看待同一个事物。这里的"方面"，指的是事务的外在特性在不同观察角度下的体现。

而在AOP中，Aspect的含义，可能更多的理解为"切面"比较合适。所以笔者更倾向于"面向切面编程"的译法。

另外需要提及的是，AOP、OOP在字面上虽然非常类似，但却是面向不同领域的两种设计思想。OOP（面向对象编程）针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。

而AOP则是针对业务处理过程中的切面进行提取，它所面对的是处理过程中的某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。这两种设计思想在目标上有着本质的差异。

上面的陈述可能过于理论化，举个简单的例子，对于"雇员"这样一个业务实体进行封装，自然是OOP/OOD的任务，我们可以为其建立一个"Employee"类，并将"雇员"相关的属性和行为封装其中。而用AOP设计思想对"雇员"进行封装将无从谈起。

同样，对于"权限检查"这一动作片断进行划分，则是AOP的目标领域。而通过OOD/OOP对一个动作进行封装，则有点不伦不类。

换而言之，OOD/OOP面向名词领域，AOP面向动词领域。

AOP和OOD/OOP并不冲突，我们完全可以在一个应用系统中同时应用OOD/OOP和AOP设计思想，通过OOD/OOP对系统中的业务对象进行建模，同时通过AOP对实体处理过程中的阶段进行隔离处理。即使不是OOD/OOP，而是在传统的POP（面向过程编程）中，AOP也能起到同样的作用。

将不同阶段领域加以分隔，这是否就算是AOP呢？

AOP还有另外一个重要特点：源码组成无关性。

[11] OOD ＝ Object Oriented Design OOP = Object Oriented Programming

倘若应用中通过某个具体的业务逻辑类实现了独立的权限检查，而请求调度方法通过预编码调用这个权限模块实现权限管理。那么这也不算是**AOP**。对于**AOP**组件而言，很重要的一点就是源码组成无关性，所谓源码组成无关性，体现在具体设计中就是**AOP**组件必须与应用代码无关，简单来讲，就是应用代码可以脱离**AOP**组件独立编译。

为了实现源码组成无关性，**AOP**往往通过预编译方式（如**AspectJ**）和运行期动态代理模式（如**Spring AOP** 和**JBoss AOP**）实现。

稍后章节中我们会就**Spring Framework**中的**AOP**实现机制进行更加深入的探讨。

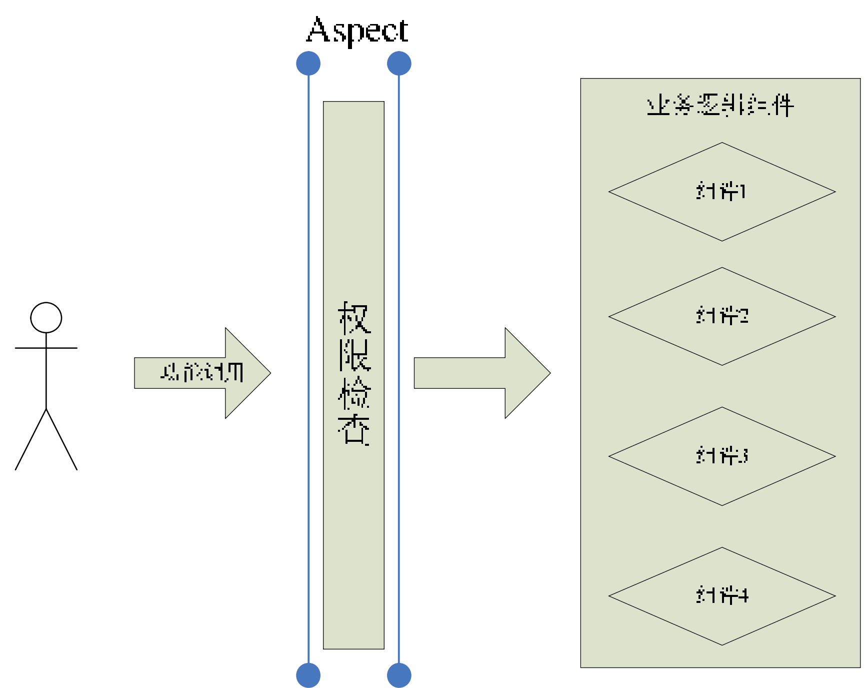
下面先来看**AOP**中几个比较重要的概念：

1. 切面（**Aspect**）

切面，对象操作过程中的截面。这可能是**AOP**中最关键的一个术语。

我们首先来看一个应用开发中常见的切面：用户权限检查。大概只要是完整的应用，都少不了用户权限检查这个模块，不同身份的用户可以做什么，不可以做什么，均由这个模块加以判定。而这个模块调用的位置通常也比较固定：用户发起请求之后，执行业务逻辑之前。

针对权限检查这一模块进行分离，我们就得到了一个切面：



切面意义何在？

首先根据上例，假设我们实现了一个通用的权限检查模块，那么就可以在这层切面上进行统一的集中式权限管理。而业务逻辑组件则无需关心权限方面的问题。也就是说，通过切面，我们可以将系统中各个不同层次上的问题隔离开来，实现统一集约式处理。各切面只需集中于自己领域内的逻辑实现。

这一方面使得开发逻辑更加清晰，专业化分工更加易于进行；另一方面，由于切面的隔离，降低了耦合性，我们就可以在不同的应用中将各个切面组合使用，从而使得代码可重用性大大增强。

2. 连接点（**JoinPoint**）

程序运行过程中的某个阶段点。如某个方法调用，或者某个异常被抛出。

3. 处理逻辑（**Advice**）

在某个连接点所采用的处理逻辑

（这里的**Advice**，国内不少文案中翻译为“通知”，估计是源于金山词霸，与实际含义不符，因而这里采用意译）

处理逻辑的调用模式通常有三种：

i. **Around**

在连接点前后插入预处理过程和后处理过程。

ii. **Before**

仅在连接点之前插入预处理过程。

iii. **Throw**

在连接点抛出异常时进行异常处理。

4. 切点（**PointCut**）

一系列连接点的集合，它指明处理方式（**Advice**）将在何时被触发。

上述几个概念我们将在稍后的“**AOP**应用”一节中结合实际使用进行具体探讨。

AOP in Spring

Spring中提供的内置AOP支持，是基于动态AOP机制实现。从技术角度来讲，所谓动态AOP，即通过动态Proxy模式，在目标对象的方法调用前后插入相应的处理代码。

而Spring AOP中的动态Proxy模式，则是基于Java Dynamic Proxy（面向Interface）和CGLib（面向Class）实现。

前面曾经提及，Spring Framework中的“事务管理”服务，实际上是借助AOP机制完成。我们这里就以“事务管理”为例，对动态AOP的实现加以探讨，一方面对动态AOP的实现原理加以探究，另一方面，也可以加深对Spring中事务管理机制的理解。

首先，我们来看基于Java Dynamic Proxy的AOP实现原理。

Dynamic Proxy 与Spring AOP

Dynamic Proxy是JDK 1.3版本中新引入的一种动态代理机制。它是Proxy模式的一种动态实现版本。

我们先来看传统方式下一个Proxy的实现实例。

假设我们有一个UserDAO接口及其实现类UserDAOImp:

UserDAO.java:

```
public interface UserDAO {
  public void saveUser(User user);
}
```

UserDAOImp.java:

```
public class UserDAOImp implements UserDAO{
  public void saveUser(User user) {
      ……
  }
}
```

UserDAOImp.saveUser方法中实现了针对User对象的数据库持久逻辑。

如果我们希望在UserDAOImp.saveUser方法执行前后追加一些处理过程，如启动/提交事务，而不影响外部代码的调用逻辑，那么，增加一个Proxy类是个不错的选择：

UserDAOProxy.java

```
public class UserDAOProxy implements UserDAO {

    private UserDAO userDAO;

    public UserDAOProxy(UserDAO userDAO) {
```

```
        this.userDAO = userDAO;
    }

    public void saveUser(User user) {
        UserTransaction tx = null;
        try {
            tx = (UserTransaction) (
                        new InitialContext().lookup("java/tx")
                        );

            userDAO.saveUser(user);

            tx.commit();

        } catch (Exception ex) {
            if (null!=tx){
                try {
                    tx.rollback();
                }catch(Exception e) {
                }
            }
        }
    }
}
```

UserDAOProxy同样是UserDAO接口的实现，对于调用者而言，saveUser方法的使用完全相同，不同的是内部实现机制已经发生了一些变化——我们在UserDAOProxy中为UserDAO.saveUser方法套上了一个JTA事务管理的外壳。

上面是静态Proxy模式的一个典型实现。

现在假设系统中有20个类似的接口，针对每个接口实现一个Proxy，实在是个繁琐无味的苦力工程。

Dynamic Proxy的出现，为这个问题提供了一个更加聪明的解决方案。

我们来看看怎样通过Dynamic Proxy解决上面的问题：

```
public class TxHandler implements InvocationHandler {

    private Object originalObject;

    public Object bind(Object obj) {
        this.originalObject = obj;
        return Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
```

```
            this);
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

        Object result = null;
        if (!method.getName().startsWith("save")) {
            UserTransaction tx = null;
            try {
                tx = (UserTransaction) (
                              new InitialContext().lookup("java/tx")
                              );

                result = method.invoke(originalObject, args);

                tx.commit();

            } catch (Exception ex) {
                if (null != tx) {
                    try {
                        tx.rollback();
                    } catch (Exception e) {
                    }
                }
            }

        } else {
            result = method.invoke(originalObject, args);
        }

        return result;
    }

}
```

首先注意到，上面这段代码中，并没有出现与具体应用层相关的接口或者类引用。也就是说，这个代理类适用于所有接口的实现。

其中的关键在两个部分：

1.

```
        return Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
```

```
            this);
```

java.lang.reflect.Proxy.newProxyInstance方法根据传入的接口类型（obj.getClass().getInterfaces()）动态构造一个代理类实例返回，这个代理类是JVM在内存中动态构造的动态类，它实现了传入的接口列表中所包含的所有接口。

这里也可以看出，Dynamic Proxy要求所代理的类必须是某个接口的实现（obj.getClass().getInterfaces()不可为空），否则无法为其构造响应的动态类。这也就是为什么Spring对接口实现类通过Dynamic Proxy实现AOP，而对于没有实现任何接口的类通过CGLIB实现AOP机制的原因，关于CGLIB，请参见稍后章节的讨论。

2.

```
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        ……
        result = method.invoke(originalObject, args);
        ……
        return result;
    }
```

InvocationHandler.invoke方法将在被代理类的方法被调用之前触发。通过这个方法中，我们可以在被代理类方法调用的前后进行一些处理，如代码中所示，InvocationHandler.invoke方法的参数中传递了当前被调用的方法（Method），以及被调用方法的参数。

同时，我们可以通过Method.invoke方法调用被代理类的原始方法实现。这样，我们就可以在被代理类的方法调用前后大做文章。

在示例代码中，我们为所有名称以"save"开头的方法追加了JTA事务管理。

谈到这里，可以回忆一下Spring事务配置中的内容：

```
    <property name="transactionAttributes">
        <props>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
```

想必大家已经猜测到Spring事务管理机制的实现原理。

是的，只需通过一个Dynamic Proxy对所有需要事务管理的Bean进行加载，并根据配置，在invoke方法中对当前调用的方法名进行判定，并为其加上合适的事务管理代码，那么就实现了Spring式的事务管理。

当然，Spring中的AOP实现更为复杂和灵活，不过基本原理一致。

代码胜千言，下面是笔者在客户培训过程中编写的一个Dynamic Proxy based AOP实现示例，非常简单，有兴趣的读者可以看看。

AOPHandler.java:

```
public class AOPHandler implements InvocationHandler {

  private static Log logger = LogFactory.getLog(AOPHandler.class);

  private List interceptors = null;

  private Object originalObject;

  /**
   * 返回动态代理实例
   * @param obj
   * @return
   */
  public Object bind(Object obj) {

      this.originalObject = obj;

      return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
obj
              .getClass().getInterfaces(), this);
  }

  /**
   * 在Invoke方法中，加载对应的Interceptor，并进行
   * 预处理(before)、后处理(after)以及异常处理（exceptionThrow）过程
   */
  public Object invoke(Object proxy, Method method, Object[] args)
          throws Throwable {

      Object result = null;
      Throwable ex = null;

      InvocationInfo invInfo = new InvocationInfo(proxy, method, args,
              result, ex);

      logger.debug("Invoking Before Intercetpors!");

      invokeInterceptorsBefore(invInfo);
```

```
    try {
        logger.debug("Invoking Proxy Method!");

        result = method.invoke(originalObject, args);

        invInfo.setResult(result);

        logger.debug("Invoking After Method!");
        invokeInterceptorsAfter(invInfo);

    } catch (Throwable tr) {
        invInfo.setException(tr);

        logger.debug("Invoking exceptionThrow Method!");
        invokeInterceptorsExceptionThrow(invInfo);

        throw new AOPRuntimeException(tr);
    }

    return result;

}

/**
 * 加载Interceptor
 * @return
 */
private synchronized List getIntercetors() {
    if (null == interceptors) {
        interceptors = new ArrayList();
        //Todo: 读取配置，加载Interceptor实例
        //interceptors.add(new MyInterceptor());
    }
    return interceptors;
}

/**
 * 执行预处理方法
 * @param invInfo
 */
private void invokeInterceptorsBefore(InvocationInfo invInfo) {
    List interceptors = getIntercetors();
    int len = interceptors.size();
```

```
        for (int i = 0; i < len; i++) {
            ((Interceptor) interceptors.get(i)).before(invInfo);
        }

    }

    /**
     * 执行后处理方法
     * @param invInfo
     */
    private void invokeInterceptorsAfter(InvocationInfo invInfo) {
        List interceptors = getIntercetors();
        int len = interceptors.size();

        for (int i = len - 1; i >= 0; i--) {
            ((Interceptor) interceptors.get(i)).after(invInfo);
        }
    }

    /**
     * 执行异常处理方法
     * @param invInfo
     */
    private void invokeInterceptorsExceptionThrow(InvocationInfo
invInfo) {
        List interceptors = getIntercetors();
        int len = interceptors.size();

        for (int i = len - 1; i >= 0; i--) {
            ((Interceptor)
interceptors.get(i)).exceptionThrow(invInfo);
        }
    }

}
```

Interceptor.java:

```
public interface Interceptor {
  public void before(InvocationInfo invInfo);
  public void after(InvocationInfo invInfo);
  public void exceptionThrow(InvocationInfo invInfo);
}
```

InvocationInfo.java:

```
public class InvocationInfo {
  Object proxy;
  Method method;
  Object[] args;
  Object result;
  Throwable Exception;

  public InvocationInfo(Object proxy, Method method, Object[] args,
         Object result, Throwable exception) {
     super();
     this.proxy = proxy;
     this.method = method;
     this.args = args;
     this.result = result;
     Exception = exception;
  }

  public Object getResult() {
     return result;
  }
  public void setResult(Object result) {
     this.result = result;
  }
  public Object[] getArgs() {
     return args;
  }
  public void setArgs(Object[] args) {
     this.args = args;
  }
  public Throwable getException() {
     return Exception;
  }
  public void setException(Throwable exception) {
     Exception = exception;
  }
  public Method getMethod() {
     return method;
  }
  public void setMethod(Method method) {
     this.method = method;
  }
  public Object getProxy() {
     return proxy;
  }
```

```
  public void setProxy(Object proxy) {
      this.proxy = proxy;
  }
}
```

AOPFactory.java:

```
public class AOPFactory {
  private static Log logger = LogFactory.getLog(AOPFactory.class);

  /**
   * 根据类名创建类实例
   * @param clzName
   * @return
   * @throws ClassNotFoundException
   */
  public static Object getClassInstance(String clzName){
      Class cls;

      try {

          cls = Class.forName(clzName);
          return (Object)cls.newInstance();

      } catch (ClassNotFoundException e) {
          logger.debug(e);
          throw new AOPRuntimeException(e);
      } catch (InstantiationException e) {
          logger.debug(e);
          throw new AOPRuntimeException(e);
      } catch (IllegalAccessException e) {
          logger.debug(e);
          throw new AOPRuntimeException(e);
      }

  }

  /**
   * 根据传入的类名，返回AOP代理对象
   * @param clzName
   * @return
   */
  public static Object getAOPProxyedObject(String clzName){
```

```
        AOPHandler txHandler = new AOPHandler();

        Object obj = getClassInstance(clzName);

        return txHandler.bind(obj);

    }

}
```

MyInterceptor.java:

```
public class MyInterceptor implements Interceptor{
  private static Log logger = LogFactory.getLog(MyInterceptor.class);

  public void before(InvocationInfo invInfo) {
      logger.debug("Pre-processing");
  }

  public void after(InvocationInfo invInfo) {
      logger.debug("Post-processing");
  }

  public void exceptionThrow(InvocationInfo invInfo) {
      logger.debug("Exception-processing");
  }
}
```

Spring中Dynamic Proxy AOP实现类为：
org.springframework.aop.framework.JdkDynamicAopProxy

CGLib 与Spring AOP

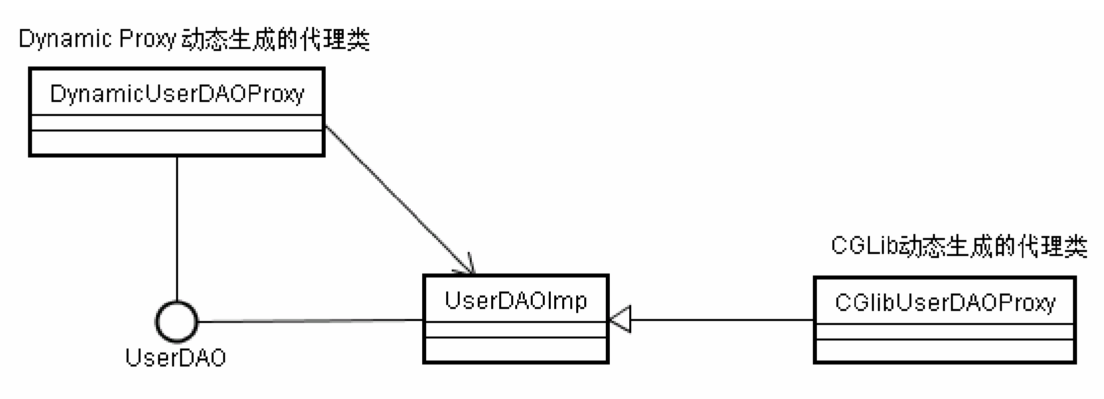
上面曾经提过，Dynamic Proxy是面向接口的动态代理实现，其代理对象必须是某个接口的实现。Dynamic Proxy通过在运行期构建一个此接口的动态实现类完成对目标对象的代理（相当于在运行期动态构造一个UserDAOProxy，完成对UserDAOImp的代理任务）。

而如果目标代理对象并未实现任何接口，那么Dynamic Proxy就失去了创建动态代理类的基础依据。此时我们需要借助一些其他的机制实现动态代理机制。

Spring中，引入了CGLib作为无接口情况下的动态代理实现。

CGLib与Dynamic Proxy的代理机制基本类似，只是其动态生成的代理对象并非某个接口的实现，而是针对目标类扩展的子类。

换句话说，Dynamic Proxy返回的动态代理类，是目标类所实现的接口的另一个实现版本，它实现了对目标类的代理（如同UserDAOProxy与UserDAOImp的关系）。而CGLib返回的动态代理类，则是目标代理类的一个子类（代理类扩展了UserDAOImp类）。



与Dynamic Proxy中的Proxy和InvocationHandler相对应，Enhancer和MethodInterceptor在CGLib中负责完成代理对象创建和方法截获处理。

下面是通过CGLib进行动态代理的示例代码：

AOPInstrumenter.java:

```
public class AOPInstrumenter implements MethodInterceptor {

    private static Log logger =
                        LogFactory.getLog(AOPInstrumenter.class);

    private Enhancer enhancer = new Enhancer();

    public Object getInstrumentedClass(Class clz) {
        enhancer.setSuperclass(clz);
        enhancer.setCallback(this);
        return  enhancer.create();
    }
```

```
    public Object intercept(
        Object o,
        Method method,
        Object[] methodParameters,
        MethodProxy methodProxy)
        throws Throwable {

        logger.debug("Before Method =>"+method.getName());

        Object result = methodProxy.invokeSuper(o, methodParameters);

        logger.debug("After Method =>"+method.getName());

        return result;
    }

}
```

测试代码：

```
AOPInstrumenter aopInst = new AOPInstrumenter();

UserDAOImp userDAO =
    (UserDAOImp) aopInst.getInstrumentedClass(UserDAOImp.class);
User user = new User();
user.setName("Erica");
userDAO.saveUser(user);
```

有兴趣的读者可以利用CGLib对Dynamic Proxy中给出的AOP实现代码进行改造。

Spring中，基于CGLib的AOP实现位于：
org.springframework.aop.framework.Cglib2AopProxy

AOP 应用

前面介绍AOP概念的章节中，曾经以权限检查为例说明AOP切面的概念。

权限检查的确是AOP应用中一个热门话题，假设如果现在出现了一个设计完备的权限管理组件，那么将是一件多么惬意的事情，我们只需要在系统中配置一个AOP组件，即可完成以往需要大费周张才能完成的权限判定功能。

可惜目前还没有这样一个很完善的实现。一方面权限检查过于复杂多变，不同的业务系统中的权限判定逻辑可能多种多样（如对于某些关键系统而言，很可能出现需要同时输入两个人的密码才能访问的需求）。另一方面，就目前的AOP应用粒度而言，“权限管理”作为一个切面尚显得过于庞大，需要进一步切分设计，设计复杂，实现难度较大。

目前最为实用的AOP应用，可能就是Spring中基于AOP实现的事务管理机制，也正是这一点，使得Spring AOP大放异彩。

之前的内容中，我们大多围绕Spring AOP的实现原理进行探讨，这里我们围绕一个简单的AOP Interceptor实例，看看Spring中AOP机制的应用与开发。

在应用系统开发过程中，我们通常需要对系统的运行性能有所把握，特别是对于关键业务逻辑的执行效能，而对于执行效能中的执行时间，则可能是重中之重。

我们这里的实例的实现目标，就是打印出目标Bean中方法的执行时间。

首先，围绕开篇中提到的几个重要概念，我们来看看Spring中对应的实现。

1. 切点（PointCut）

一系列连接点的集合，它指明处理方式（Advice）将在何时被触发。

对于我们引用开发而言，“何时触发”的条件大多是面向Bean的方法进行制定。实际上，只要我们在开发中用到了Spring的配置化事务管理，那么就已经进行了PointCut设置，我们可以指定对所有save开头的方法进行基于AOP的事务管理:

```
<property name="transactionAttributes">
      <props>
          <prop key="save*">PROPAGATION_REQUIRED</prop>
      </props>
</property>
```

同样，对于我们的AOP组件而言，我们也可以以方法名作为触发判定条件。

我们可以通过以下节点，为我们的组件设定触发条件。

```
<bean id="myPointcutAdvisor"
  class="org.springframework.aop.support.RegexpMethodPointcutAdv
isor">
  <property name="advice">
      <ref local="MyInterceptor" />
  </property>
  <property name="patterns">
```

```
        <list>
            <value>.*do.*</value>
            <value>.*execute.*</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor是Spring中提供的，通过逻辑表达式指定方法判定条件的支持类。其中的逻辑表达式解析采用了Apache ORO组件实现，关于逻辑表达式的语法请参见Apache ORO文档。

上面我们针对MyInterceptor设定了一个基于方法名的触发条件，也就是说，当目标类的指定方法运行时，MyInterceptor即被触发。

MyInterceptor是我们对应的AOP逻辑处理单元，也就是所谓的Advice。

2. Advice

Spring中提供了以下几种Advice:

1. Interception around advice

   Spring中最基本的Advice类型，提供了针对PointCut的预处理、后处理过程支持。

   我们将使用Interception around advice完成这里的实例。

2. Before advice

   仅面向了PointCut的预处理。

3. Throws advice

   仅面向PointCut的后处理过程中的异常处理。

4. After Returning advice

   仅面向PointCut返回后的后处理过程。

5. Introduction advice

   Spring中较为特殊的一种Advice，仅面向Class层面（而不像上述Advice面向方法层面）。通过Introduction advice我们可以实现多线程访问中的类锁定。

Spring中采用了AOP联盟（AOP Alliance）[12]的通用AOP接口（接口定义位于aopalliance.jar）。这里我们采用aopalliance.jar中定义的MethodInterceptor作为我们的Advice实现接口:

```
public class MethodTimeCostInterceptor implements
MethodInterceptor,
    Serializable {

  protected static final Log logger = LogFactory
```

[12] http://aopalliance.sourceforge.net/

```
        .getLog(MethodTimeCostInterceptor.class);

  public Object invoke(MethodInvocation invocation) throws
Throwable {

      long time = System.currentTimeMillis();

      Object rval = invocation.proceed();

      time = System.currentTimeMillis() - time;

      logger.info("Method Cost Time => " + time + " ms");

      return rval;
  }

}
```

对应配置如下：

```
  <bean id="MyInterceptor"
      class="net.xiaxin.interceptors.MethodTimeCostInterceptor"
/>
```

除此之外，我们还需要定义一个**Spring AOP ProxyFactory**用以加载执行**AOP**组件。定义如下：

```
  <bean id="myAOPProxy"
  class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
          <value>net.xiaxin.ITest</value>
      </property>

      <!—是否强制使用CGLIB进行动态代理
      <property name="proxyTargetClass">
          <value>true</value>
      </property>
      -->
      <property name="target">
          <ref local="test" />
      </property>

      <property name="interceptorNames">
          <value>myPointcutAdvisor</value>
      </property>
```

```
  </bean>

<bean id="test" class="net.xiaxin.Test"/>
```

其中的 test 是我们用于测试的一个类，它实现了 ITest 接口。

```
public interface ITest {
  public abstract void doTest();

  public abstract void executeTest();
}

public class Test implements ITest {
  public void doTest(){
      for (int i=0;i<10000;i++){}
  }
  public void executeTest(){
      for (int i=0;i<25000;i++){}
  }
}
```

通过以上工作，我们的MyInterceptor即被加载，并将在Test.doTest和Test.executeTest方法调用时被触发，打印出这两个方法的执行时间。

```
    public void testAOP() {

        ApplicationContext ctx=new
                    FileSystemXmlApplicationContext("bean.xml");

        ITest test = (ITest) ctx.getBean("myAOPProxy");

        test.doTest();
        test.executeTest();

    }
```

本章内容待补充...

以下部分待整理后发表

DAO Support

Remoting