

2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——



OpenDoc Series'



Spring 开发指南

V0.6 预览版

作者：夏昕 [xiaxin\(at\)gmail.com](mailto:xiaxin(at)gmail.com)

So many open source projects. Why not **Open** your **Documents**? 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。

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。

另外，笔者近来试图就日本、印度的软件开发模式进行一些调研。如果诸位可以赠阅日本、印度软件研发过程中的需求、设计文档以供研究，感激不尽！

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。

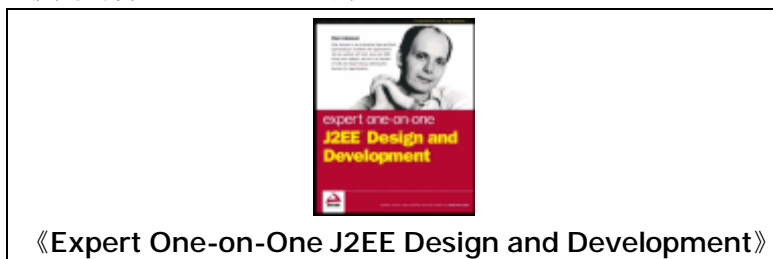
工作间隙，时常见到他摩娑此书，摇头不止（印度人习惯和中国人相反，摇头代表肯定、赞同，相当于与中国人点头。笔者刚开始与印度同僚共事之时，每每组织项目会议，一屋子人频频摇头，让笔者倍感压力.....J）。

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：



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操中的经验所得。

Spring 初探	5
准备工作	5
构建 Spring 基础代码	6
Spring 基础语义	12
Dependency Injection.....	12
依赖注入的几种实现类型	14
Type1 接口注入.....	15
Type2 构造子注入.....	15
Type3 设值注入.....	15
几种依赖注入模式的对比总结	16
Spring Bean 封装机制.....	17
Bean Wrapper	17
Bean Factory	18
ApplicationContext	21
Web Context.....	26
Spring 高级特性	27
Web 应用与 MVC.....	27
Spring MVC	28
Spring MVC 指南.....	28
基于模板的 Web 表示层技术	42
Web 应用中模板技术与 JSP 技术的对比.....	47
输入验证与数据绑定	49
异常处理	60
国际化支持	62
数据持久层	66
事务管理	66
持久层封装	70
JDBC.....	70
Hibernate in Spring.....	78
ibatis in Spring	85
以下内容待整理后发布.....	88
远程调用	88
AOP	88

Spring 初探

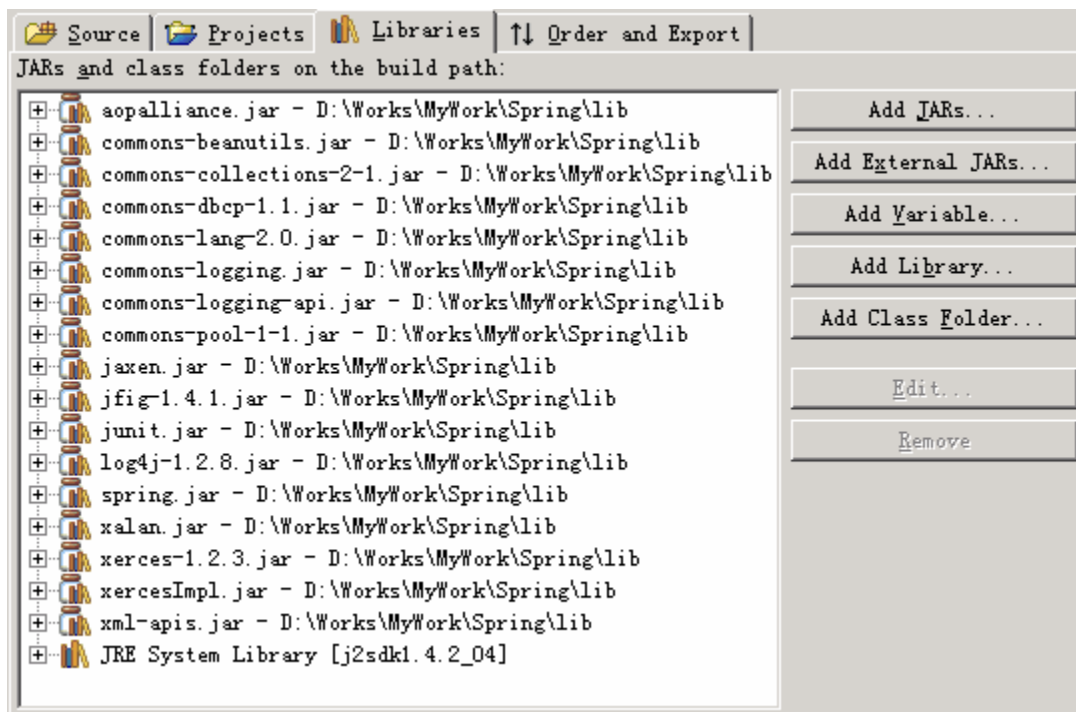
开始 **Spring** 研究之前，先让我们来看一个 1 分钟上手教程。

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：



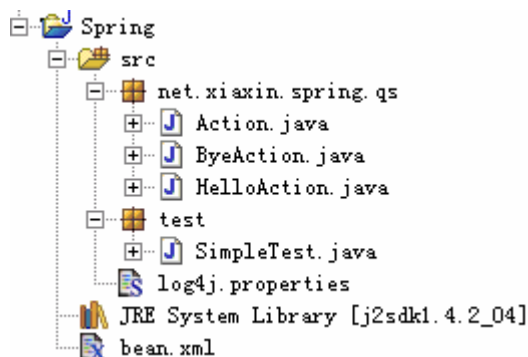
- Ø **Spring** 采用 **Apache common_logging**，并结合 **Apache log4j** 作为日志输出组件。为了在调试过程中能观察到 **Spring** 的日志输出，在 **CLASSPATH** 中新建 **log4j.properties** 配置文件，内容如下：

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>HeLlLo</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。

Ø 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移到了外部容器，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”.....，相信大家对于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到依赖关系的存在。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，而不用担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 init() {
        Object obj =
            Class.forName(Config.BImplementation).newInstance();
        clzB = (InterfaceB)obj;
    }
    .....
}
```

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，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J2EE开发中常用的Context.lookup（ServletContext.getXXX），都是Type1型IOC的表现形式。

Apache Avalon是一个典型的Type1型IOC容器。

Type2 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Type2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Type2类型的依赖注入模式。

Type3 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。

几种依赖注入模式的对比总结

接口注入模式因为具备侵入性，它要求组件必须与特定的接口相关联，因此并不被看好，实际使用有限。

Type2和**Type3**的依赖注入实现模式均具备无侵入性的特点。在笔者看来，这两种实现方式各有特点，也各具优势（一句经典废话**J**）。

Type2 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，**Type2**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。

Type3 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**Type2**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**Type3**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**Type2**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。

可见，**Type2**和**Type3**模式各有千秋，而**Spring**、**PicoContainer**都对**Type2**和**Type3**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**Type2**类型为主，辅之以**Type3**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，**Type3**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，如果感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>HeLlLo</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 ==> |Ï;À?;ã|Ï???“®??;ì: [Erica] |Ï???“°;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。

LoginAction.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 MVC¹ 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。

如何选择一个好的框架? 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

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：

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.InternalResourceViewResolver">
    <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.SimpleUrlHandlerMapping">
    <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>
  </bean>
</beans>
```



```

    <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：

Ø **`org.springframework.web.servlet.view.freemarker.FreeMarkerView`**（用于基于 FreeMarker 模板的表现层实现）

Ø **`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`，`BaseCommandController` 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。

(2) 将输入的请求数据对象强制转型为预定义请求对象类型。

(3) 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:

Ø `${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}
```

Ø `${1+2}`

结果为表达式计算结果，即整数值 3。

Ø `${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:

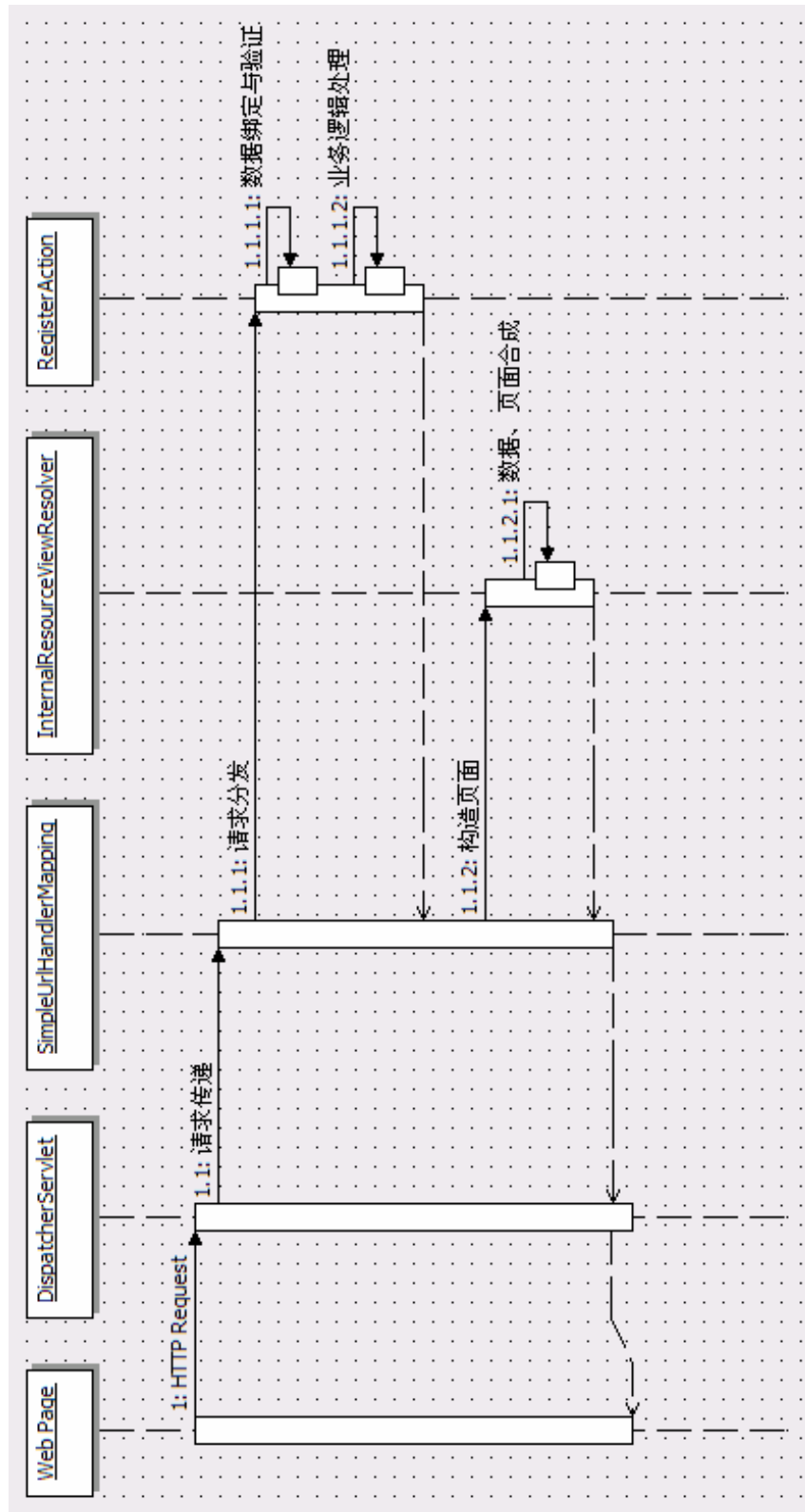
```
Login Success!!!

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:



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 XSLT 支持，现有的 XSLT 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，XSLT 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>
  </bean>
</beans>
```

```
<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

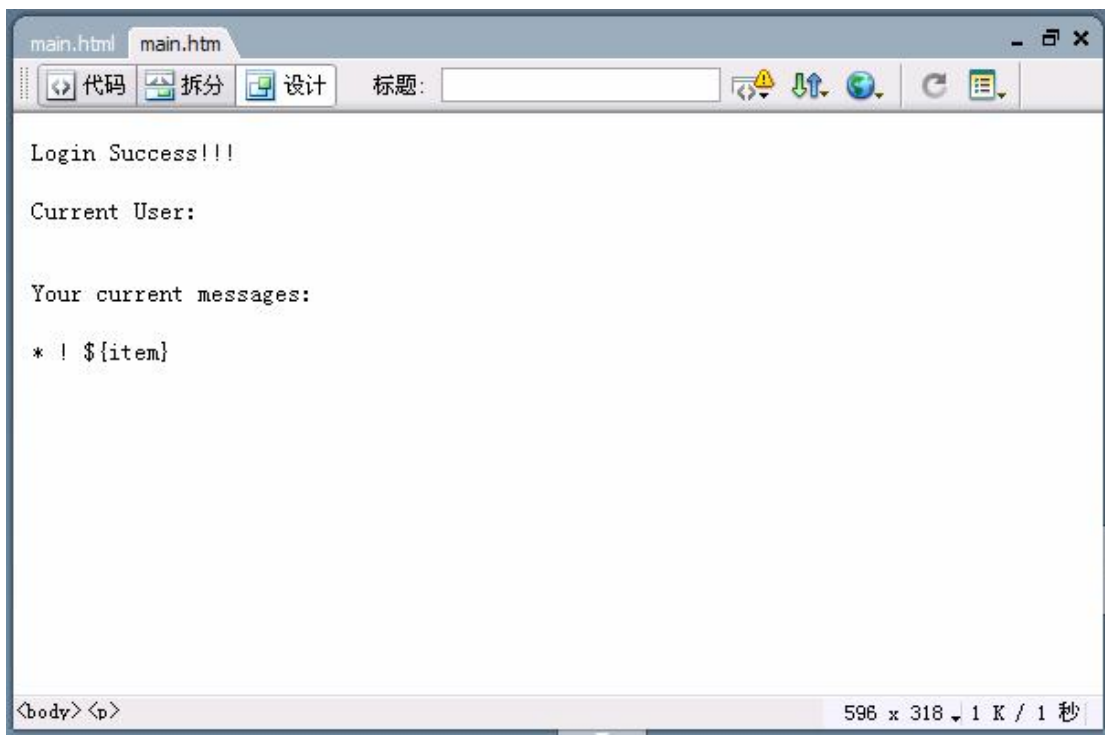
通过上面的例子可以看到，FreeMarker模板文件与之前使用JSTL Core Taglib实现的JSP页面非常相似。

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:

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: 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!

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
sourceViewResolver">
        <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>

</beans>
```

```
</bean>

<!--Request Mapping -->
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <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：

- (1) 配置了一个数据验证Bean: RegisterValidator
net.xiaxin.validator.RegisterValidator
- (2) 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。
- (3) 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。
- (4) 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:

Ø 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。

Ø 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></p>
                    </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）：

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：

密码：

重复密码：

提交 重置

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、种类繁多，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.SimpleMappingExceptionResolver">
    <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SimpleMappingExceptionHandler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SimpleMappingExceptionHandler无法满足异常处理的需要，我们可以针对HandlerExceptionHandlerResolver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.ResourceBundleMessageSource">
    <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）：

Ø 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）。

Ø SessionLocaleResolver

²Windows 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>
```

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

这几个属性并非必须配置，其默认值为分别为：

Ø **cookieName**

`org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE`

Ø **cookiePath**

`/`

Ø **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。

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.TransactionProxyFactoryBean">
  <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** 规范编写 **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PreparedStatement执行update操作以避免SQL Injection 漏洞³：

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate
    .update(
        "UPDATE user SET age = ? WHERE id = ?",
        new PreparedStatementSetter() {
            public void setValues(PreparedStatement 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(
```

³ 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>


```
"SELECT name, sex, address FROM user WHERE age > 18",
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:

Ø **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。

Ø **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。

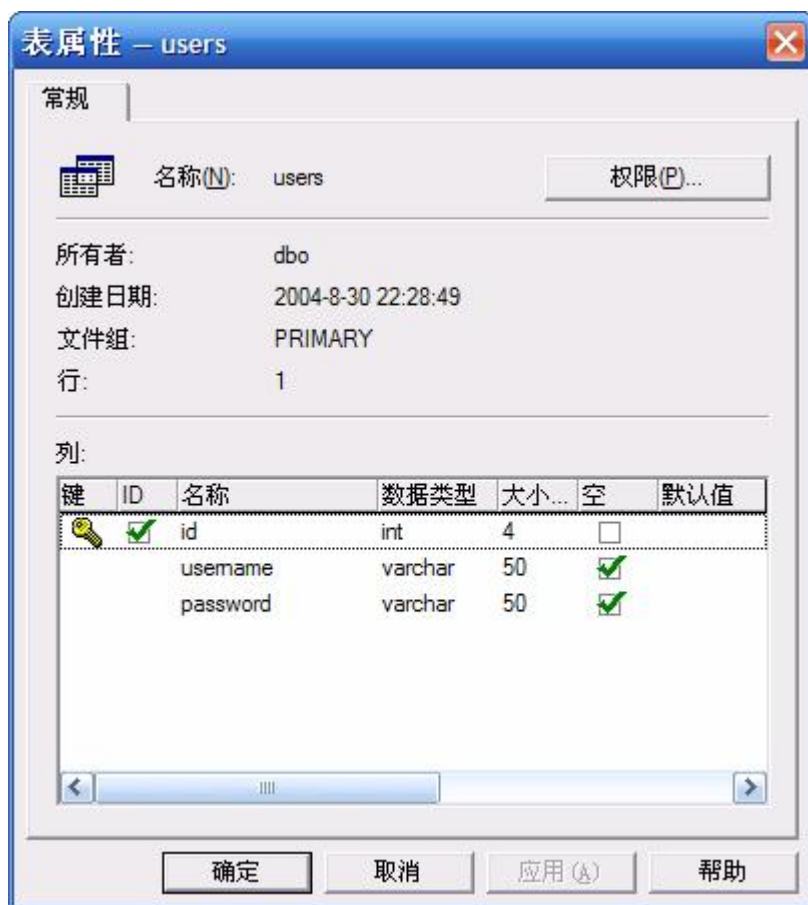
如果需要使用基于容器的数据源（JNDI），我们可以采用如下配置：

```
<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"
/>
```

Ø 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 getDataSource() {
    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(getDataSource());
            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Test Case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。

U 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.TransactionProxyFactoryBean">
    <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(getDataSource());  
    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，相信大家会有一些豁然开朗的感觉。

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作为一个开放性的应用开发平台。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>
  </bean>
</beans>
```

```
</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.HibernateTransactionManager">
  <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.TransactionProxyFactoryBean">

  <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>
```



```
</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，如果对上述基于**JdbcTemplate**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**⁴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**返回的仍然是一个**UserDAO**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**方法创建的经过动态代理的**Bean**实例，这个实例的类结构已经不同于原始的**UserDAO**类，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。

⁴ 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.DataSourceTransactionManager">
    <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.TransactionProxyFactoryBean">

    <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);
```

以下内容待整理后发布

远程调用

AOP