第2章 体验控制反转的理念(Spring讲解)

继 Rod Johonson 在 2002 年出版了《Expert one on one J2EE design and development》一书后，Spring 的设计思想被广泛的接受。因为 Spring 为企业级应用提供了一套轻量级的解决方案。在设计上它实现了一些优秀的模式如控制反转(Ioc)模式，或者说依赖注入。与此同时，它也和 struts 一样也实现了 MVC 框架。可以说，Spring 是目前最流行的一种开源框架，越来越多的开发人员正在接触它，学习它，使用它。

2.1 Spring 概述

本节将通过介绍 Spring 的一些发展历史和 Spring 的整体构成给大家带来对 Spring 的一个感性认识，引入大家进入 Spring 的世界，从本节中你将了解到 Spring 是如何而来，它的整体外貌是什么。

2.1.1 Spring 的框架构成

Spring 由 7 个定义良好的架构构成：Spring AOP，Spring ORM，Spring Web，Spring Context，Spring DAO，Spring Web MVC 和 Spring Core。而前六种模块又构建在 Spring Core 基础之上，如图 2-1 所示。

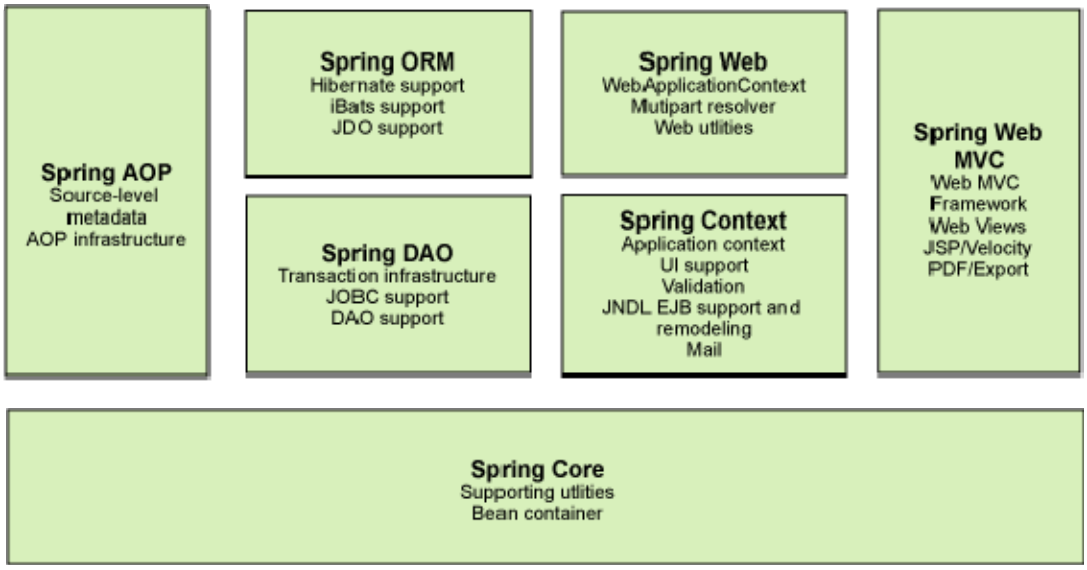

图 2-1 Spring 框架图

下面针对框架图中的 7 个模块逐一进行介绍

- Spring Core：这是 Spring 的核心容器，它包含 Spring 的核心功能。这个容器通过实现工厂模式来管理着各种不严格规范的 Bean.运用控制反转的思想将应用程序代码和依赖性隔离出来，它是整个 Spring 框架的基础和核心所在。
- Spring AOP：AOP 意为面向方面编程（或者说面向切面编程）是 Spring 框架的另一个重要思想。AOP 是相对传统 OOP 编程技术提出来的，一个典型的系统具有很多的关注点，而传统的 OOP 编程技术在跨越多个模块进行编程时显得能力不够，会带来诸如代码混乱之类的问题。AOP 就是针对 OOP 的这一缺陷而提出来

的一种新的编程方法学，它很有可能成为下一个被广泛接受的编程方法学。

- Spring ORM：ORM 是对象关系模型。这一模块包含了与数据库持久化密切相关的技术，其中有将在下一章介绍的 Hibernate 以及 JDO.iBatis SQL Map.Spring 将它们有效的组织在一起为用户提供很好的数据库访问，操作技术支持。
- Spring DAO：这一模块主要是为了实现在上一节提到的尽量减少不必要异常捕获这一目标而设计的。它提供了一个有效的异常层次结构，用来管理来自不同数据库供应商抛出的错误消息。
- Spring Web：这个模块为基于 web 的应用提供了一个上下文。也正因为这个能力使得 Spring 能与 Struts 这样的基于 web 的框架进行友好方便的结合。
- Spring Context：它的原型是一个配置文件，类似于 Struts 中的的 struts-config.xml 文件，它为整个 Spring 框架提供上下文信息.
- Spring Web MVC：MVC 是 Spring 与 Struts 共同具备的能力，不同的是，Spring 的一些思想让它所实现的 MVC 变的高度可配置。

2.2 依赖注入思想

控制反转（IoC）和依赖注入(DI)是接触 Spring 最先需要了解的概念，也是 Spring 的核心思想，它在整个 Spring 框架中起到骨架地位。鉴于这些理由本章将其放在较前位置进行讲解。首先本节将通过实例让读者了解其概念，然后会逐一介绍三种方式的依赖注入。

2.2.1 控制反转与依赖注入思想

控制反转的英文全称是 Inversion of Control 简称为 IoC。依赖注入的全称是 Dependncy Injection 简称 DI。DI 是后面对 Ioc 的另一个提法，所以在本质上它们是同种思想的两个不同的说法。这两个词语顾其名很难思其义，从字面意义上的确很难理解其含义。所以这里打算举个通俗的例子来给大家一个感性的认识。

张三在某家公司任职，老板要求他负责公司职员日常的饮水问题。老板对张三可以有两种形式的要求：第一种是必须只能购买 A 公司的水。第二种是只要有水喝就行不管哪家都行。如果是第一种情况，那么当张三发现不能在和 A 公司合作而想转向购买 B 公司的水的时候，张三必须要做的一件事情是向老板请示并且征得其同意方可更改卖主。如果是第二种情况，则张三完全不用关心老板的意见就可以决定并且控制这个更改。这就体现了控制反转思想。什么控制发生了反转？老板只对张三提出一个接口要求那就是水，至于从什么地方来，这个过程的更改控制权反转给了张三。

在 Spring 框架中起到上面张三作用的是一个 xml 格式的配置文件。矿泉水公司相当于提供服务 Beans 而老板则相当于请求服务的另一个 Bean。XML 是它们之间的桥梁，当两者依赖关系发生变化时候，只需要更改 XML 文件进行依赖注入就可以实现服务变更。所以从某种程度上而言 Spring 框架可以用一个数学表达式进行表述那就是：Spring=Beans+XML。

上面这个例子可以转化成具体代码形式，首先需要创建一矿泉水公司 A，这个类包括了一个提供矿泉水服务的基本属性,代码如下：

```
public class WaterCompanyA {
 //水是一家矿泉水公司最基本的服务
private Water water;


   ......
//提供水
```

```
public Water getWater() {
 return water;
}
}
```

同样的，还需要为矿泉水 B 公司创建一个类，它也具有 water 属性，但这并不能代表两家公司就可以抽象成为一个类，因为它们之间必然存在诸多的不同点。就好像 DB2 数据库和 Oracle 数据库一样尽管同样提供数据库的查询等基本功能，但却不可将二者等同起来。

```
public class WaterCompanyB {
 //水是一家矿泉水公司最基本的服务
private Water water;

   ......
//提供水
public Water getWater() {
 return water;
}
}
```

现在老板应该出场了，用更准确的话来说老板是一个客户端程序，他需要矿泉水公司的服务。所以做为老板如果出于第一种选择那么这个类看上去应该是这样的：

```
public class Boss {
//实例化一个 A 公司对象就好像和 A 公司签订了一份合同。
private WaterCompanyA wca=new WaterCompanyA();
public void needWater(){
 //如愿以偿的得到了水服务。
 wca.getWater();
}

}
```

是不是这样就万事大吉了呢，还有张三去哪了呢，先别急，现在还没有轮到他出场。某一天，A 公司的某位高层突然心血来潮说 A 公司不再提供矿泉水而是改成饮料了。这时候张三的老板是不是要为此做出点反映呢？当然这也不是什么灾难性的后果稍微改改像这样就可以将 A 公司换成 B 公司了:

```
public class Boss {
//实例化一个 B 公司对象就好像和 B 公司签订了一份合同。
private WaterCompanyB wcb=new WaterCompanyB();
public void needWater(){
 //如愿以偿的得到了水服务。
 wcb.getWater();
}

}
```

张三的老板以为这次可以高枕无忧了。可万万没有想到，A 公司改成经营饮料没多久，就遇到了重重阻力，通过董事会决议后，又只好重操旧业了，可这时候 B 公司已经远远超过了自己。于是为了在原本属于自己的行业中重新找回地位，A 公司决定薄利多销，价格卖的比 B 公司便宜。于是这让张三的老板又蠢蠢欲动了。他又想改回去买 A 公司的水。为了避免如此折腾，他终于想到一个办法就是把张三叫了过来“以后水的事情，就你负责了，买哪家公司的水你来决定”。这样张三的老板和矿泉水公司之间就关系由直接变成了间接。中间多了一个张三。下面针对这种情况重新设计上面的几个类。

在设计矿泉水公司类之前，需要先设计一个接口，只所以需要一个接口是为了实现 Boss 类不直接通过调用类来取得服务，而是通过接口隔离。这个接口只包含一个方法如下：

```
//接口的命名一般习惯上以 I 开头
public interface IWaterCompany {
   //提供水服务
 public Water getWater();


}
```

然后 A，B 两家公司都来实现这个接口，因为它们都可以并非常想为 Boss 提供服务。A 公司实现接口后的类如下：

```
//实现 IwaterCompany 接口
public class WaterCompanyA    implements IWaterCompany{

private Water water;

   //这个方法是接口中定义的方法必须实现
public Water getWater() {
 return water;
}
}
```

实现 IwaterCompany 接口则必须实现 getWater()方法。同样的 B 公司也应该实现 IwaterCompany 接口，实现接口后的 B 公司类如下：

```
//实现 IwaterCompany 接口
public class WaterCompanyB    implements IWaterCompany{

private Water water;

   //这个方法是接口中定义的方法必须实现
public Water getWater() {
 return water;
}

}
```

它也必须实现 getWater()这个方法。到现在可以编写客户端程序来调用它们的服务

了，这个客户端是就改写 Boss 类

```
public class Boss {
//以接口为属性
private IWaterCompany wcp;

public void needWater(){
 //如愿以偿的得到了水服务。
 getWcp().getWater();
}

public IWaterCompany getWcp() {
 return wcp;
}
//实现 IwaterCompany 接口的所有类的实例都可以做为参数传递进来
public void setWcp(IWaterCompany wcp) {
 this.wcp = wcp;
}
```

这个类看上去和前两个 Boss 类大有不同，这里再也无法从里面找到 A 或者 B 的字眼。它的属性是一个接口，因此也就没有发现 new 操作。那么 A 公司和 B 公司怎么样才能将它们注入给 Boss 呢。下面为此写个测试类：

```
public class TestClient {

public static void main(String[] args) {

 Boss boss=new Boss();
    //先实例化一个 WaterCompanyA 对象，然后将它注入进去
 boss.setWcp(new WaterCompanyA());
   //的到了 A 公司提供的水服务
 boss.needWater();
}

}
```

可以看到这个测试类首先通过 new 操作实例化一个 WaterCompanyA 对象，然后将其做为方法参数传递给了 boss 类中的 wcp。这时候应该明白为什么 WaterCompanyA 必须实现 IwaterCompany 这个接口了。然后 boss 通过调用 needWater()方法获得了水的服务，而且这个就是 A 公司提供的水，那么现在要改成 B 公司呢.不难，修改下 TestClient 就可以满足这一要求，修改后的代码如下：

```
public class TestClient {

public static void main(String[] args) {

 Boss boss=new Boss();
    //先实例化一个 WaterCompanyB 对象，然后将它注入进去
 boss.setWcp(new WaterCompanyB());
   //的到了 B 公司提供的水服务
 boss.needWater();
```

```
    }

    }
```

到这里发现一个重要的事情了没有？Boss 这个类在这次变更中仍然完好无缺，也就是说这一变更并不需要更改他的内在结构，而只要改下 TestClient 这个类就可以了。TestClient 充当的就是张三这种角色。不过在 Spring 中它不以一个客户端类存在的，而是一个 XML 文件。通过这种接口隔离的方式实现了控制反转。在下面的一节中将介绍依赖注入的几种方式。

2.2.2 三种依赖注入方式

通过上一小节的实例讲解，我们知道了控制反转的思想，这里不区分的也把它称为依赖注入。在上一小节的 TestClient 这个类中，是通过 set 方式进行对象注入的，事实上，这不是唯一的一种注入方式，本小节将针对依赖注入的三种方式逐一进行介绍。

2.2.2.1 接口注入

接口注入就是将要注入的内容置入到一个接口中，然后将其注入到它的实现类中，因为实现一个接口必须实现接口定义的所有方法。比如为可以为上面的例子再定义一个接口：

```
public interface IGetCompany {
//取得一家具体的公司
public IWaterCompany getCompany(IWaterCompany wcp);
}
```

当一个类实现了这个几接口，就意味着它被注入了 getCompany 这个方法，且必须实现它哪怕是空实现，这是强加给它的实现类的。如下面这个实现类：

```
public class GetCompanyImpl implements IGetCompany {
//实现 getCompany 方法，且必须实现，它是 IgetCompany 注入给它的一个方法。
 public IWaterCompany getCompany(IWaterCompany wcp) {

        return wcp;
 }

}
```

这里的 getCompany 只是把从调用者传入的参数原封不动的返回去。

2.2.2.2 set 注入

所谓 set 注入就是注入者通过调用 setter 方法将一个对象注入进去，如上面的 Boss 类。

```
public class Boss {
//以接口为属性
private IWaterCompany wcp;

public void needWater(){
 //如愿以偿的得到了水服务。
 getWcp().getWater();
```

```
}

public IWaterCompany getWcp() {
 return wcp;
}
//实现 IwaterCompany 接口的所有类的实例都可以做为参数传递进来
public void setWcp(IWaterCompany wcp) {
 this.wcp = wcp;
}
```

Boss 类中含有一个 wcp 属性，要想给这个属性注入一个具体的对象值，那么就可以通过 setWcp 这个方法做到。具体办法如前面示例中的 TestClient 类：

```
public class TestClient {

public static void main(String[] args) {

 Boss boss=new Boss();
    //先实例化一个 WaterCompanyA 对象，然后将它注入进去
 boss.setWcp(new WaterCompanyA());
   //的到了 A 公司提供的水服务
 boss.needWater();
}

}
```

注意到 TestClient 首先实例化一个 Boss 对象，然后通过调用它的 setWcp 方法将一个 WaterCompanyA 实例注入给了 boss 的属性 wcp，这个过程就叫 set 注入。

2.2.2.3 构造注入

构造注入是通过一个带参的构造函数将一个对象注入进去。上面的 Boss 类是通过 set 方法进行依赖注入的，那么也很容易改为构造注入方法如下：

```
public class Boss {

private IWaterCompany wcp;
//构造注入
public Boss(IWaterCompany wcp){
 this.wcp=wcp;
}
}
```

可见这个类中不再有 set 方法，而是直接将参数写在了构造函数中。那么在实例化一个 Boss 类的时候就可以将一个 IWaterCompany 实例注入给 Boss。这个过程就叫构造注入。

三种注入方式中，构造注入与 set 注入比较类似，那么什么时候用 set 注入，什么时候用构造注入呢？这就取决于注入的先后是否对其它业务逻辑有影响，如果在一个业务逻辑中，属性的值必须在最先时候就要初始化，那么使用构造注入会是一个较好的选择，否则如果是在一种动态的不确定的环境下需要注入一个对象，但又没有必要调用构造函数而造成实例化出一个多余的对象的情况下，使用 set 注入会比较自然。

2.3Bean 介绍

Spring 可以用一个数学公式表达，那就是 Spring=Beans+XML。尽管这种提法有点极端，但通过后面的学习后会发现 Spring 的确就是在讲述 Bean 和 XML 之间的故事。鉴于此，本节将详细介绍 Bean。

2.3.1Bean 与 Spring 容器

在 Spring 中所有的组件都会被认为是一个 Bean，Bean 是容器管理的一个基本单位。Bean 可以是标准的 JavaBean，但大不部分情况下，它并不是标准的 JavaBean 而是一些如数据源，Hibernate 的 SessionFactory，甚至一个任何 Java 对象都可以看成是一个 Bean。Spring 通过一个叫 BeanFactory 的容器对这些 Bean 进行管理。最典型的管理是实例化一个 Bean.它的原型是一个接口，实现了工厂模式。BeanFactory 包含以下一些基本的方法：

- public Object getBean(String name)：根据名称返回一个 Bean 的实例
- public Object getBean(String name,Class requiredType)：根据名称和类型返回一个 Bean 的实例
- public Class getType(String name)：返回一个名为 name 的 Bean 的类型
- public boolean containsBean(String name)：判断一个名称为 name 的 Bean 是否存在容器中。

可见，客户程序可以方便的调用上面的方法达到比如实例化一个 Bean 的目的，并且不需要关心其过程。不过，BeanFactory 提供的功能是基本的，也是有限的，Spring 为其设计了一个子接口 ApplicationContext。这个接口对外提供更多的功能，在大部分情况下，Spring 推荐使用这个接口。ApplicationContext 的常用实现类是 org.spring-framwork.context.support.FileSystemXmlApplicationContext。这个类名里面包含有 XML 字样，事实上，管理一个 Bean 在大部分情况下是需要在一个 XML 配置文件配置信息的，这也是为什么说 Spring=Beans+XML。下面来看一个一般的配置文件是什么样的的：

```
<?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="DBInfo" class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName"
           value="oracle.jdbc.driver.OracleDriver">
      </property>
      <property name="url"
           value="jdbc:oracle:thin:@XXXX:1521:ora9i">
      </property>
      <property name="username" value="xxxx"></property>
      <property name="password" value="xxxx"></property>
 </bean>
 <bean id="ssmssSessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
      <property name="dataSource">
           <ref bean="DBInfo" />
      </property>
```

```
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.Oracle9Dialect
                </prop>
            </props>
        </property>
        <property name="mappingResources">
            <list>
                <value>xif/dao/Imenus.hbm.xml</value>
                <value>rel/dao/RelTeaspec.hbm.xml</value>
                <value>rel/dao/RelStuselsec.hbm.xml</value>
                <value>view/dao/ViewTeaall.hbm.xml</value></list>
        </property></bean>

    <bean id="hibernateDataFetcher" class="org.starc.commons.datagrid.HibernateDataFetcher">
        <property name="sessionFactory">
            <ref local="ssmssSessionFactory" />
        </property>
    </bean>

    <!-- 服务 Bean :Service Bean -->
        <bean id="CheckUserService"
        class="Service.CheckUserServiceImpl">
        <property name="udao">
            <ref bean="IusersDAO" />
        </property>
    </bean>


    <bean id="ImenusDAO" class="xif.dao.ImenusDAO">
        <property name="sessionFactory">
            <ref bean="ssmssSessionFactory" />
        </property>
    </bean>
    <bean id="CoreSpecDAO" class="core.dao.CoreSpecDAO">
        <property name="sessionFactory">
            <ref bean="ssmssSessionFactory" />
        </property>
    </bean>
    <bean id="CoreStuinfoDAO" class="core.dao.CoreStuinfoDAO">
        <property name="sessionFactory">
            <ref bean="ssmssSessionFactory" />
        </property>
    </bean>
    <bean id="CoreTeainfoDAO" class="core.dao.CoreTeainfoDAO">
        <property name="sessionFactory">
            <ref bean="ssmssSessionFactory" />
        </property>
    </bean>
    </beans>
```

观察下这个配置文件，细心的读者就应该不难发现它其实就是一个<beans>标签里

面包含很多的<bean>标签。的确，每一个<bean>标签就代表一个 bean 的信息。Spring 总是以这个配置信息为纲然后通过 BeanFactory 或者 ApplicationContext 来管理 Bean。

假如上面这个配置文件名称为 build.xml。那么如何将其与 BeanFactory 联系上呢，或者说如何用 BeanFatory 进行管理呢。下面以 ApplicationContext 的一个实现类 FileSystemXmlApplicationContext 进行装载这个配置文件代码如下：

```
FileSystemXmlApplictionContext appContext=new
FileSystemXmlApplicationContext(“build.xml”);
    BeanFactory factory=(BeanFactory)appContext;//由于 FileSystemXmlApplicationContext 所实
现//的 ApplicationContext 是 BeanFactory 的子接口，所以这种类型转化是允许的
```

这两行代码非常简单的将一个配置文件与一个 BeanFactory 对象联系了起来，它首先通过 FileSystemXmlApplicationContext 将 buil.xml 文件载入到内存并且实例出一个 appContext 对象，然后通过类型转化得到一个 BeanFactory 对象 factory，后面就可以通过这个 factory 进行相关管理操作了。但这并不是唯一的方式，只是这种方式比较常用。

2.3.2 在 XML 文件中配置 Bean

在上一小节中讲到 Spring 的配置文件内容都是由<bean>构成，配置一个基本的 Bean，至少需要设置两个属性:id 和 class。下面是一个最简单的配置示例：

```
<?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="HelloWorld" class="test.HelloWorld"  >

</bean>
</beans>
```

HelloWorld 是一个 JavaClass。它可能是一个标准的 JavaBean，但也可能不是，id 是它在 BeanFactory 容器中存在的标识，可以通过它找到这个 Bean 的引用，class 是 HelloWorld 这个类的类路径，这里不能是一个接口，因为 Spring 容器是通过它来进行实例化一个 Bean 对象的。

上面讲到实例化一个 Bean 对象，那么读者一定会问，是不是每次使用的时候都需要通过 new 操作进行对象实例化？幸运的是，Spring 容器给它所管理的 Bean 提供了两者存在的方式，一种是 singeton 即所谓的单件，一种是 non-singleton 非单件。单件这种设计模式将在第七章中详细介绍。简言之，以单件存在的 Bean 永远都只有一份，而以非单件存在的 Bean 会随时的被创建，销毁，而这是需要系统开销的，但往往又不是必须的，所以在默认情况下 Spring 容器是以单件的形式创建一个 Bean 的。如果用户需要将其改成非单件模式。那么可以通过修改<bean>标签的 singleton 属性，如将上面的 HelloWorld 改成非单件模式：

```
<!—配置一个非单件 Bean -- >
<bean id="HelloWorld" class="test.HelloWorld" singleton=”false” >
</bean>
</beans>
```

于是，如果通过 BeanFactory 的 getBean 方法去申请 HelloWorld 这个 Bean，那么对于单件形式的 Bean 每次返回的是同一个 Bean，而对于非单件的 Bean 每次返回的都不是同一个 Bean.为了验证这个结论可以写个测试程序：

```
package test;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

//测试单件 Bean 与非单件 Bean 的区别
public class TestClient {
public static void main(String[] args) {

      ApplicationContext                                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//两次调用 getBean 方法获得两个 HelloWorld 的引用
      HelloWorld hw=(HelloWorld)appContext.getBean("HelloWorld");
          HelloWorld hw2=(HelloWorld)appContext.getBean("HelloWorld");
//==操作符用以判断两个对象是否为同一个对象
          if(hw==hw2){
            System.out.println("hw 和 hw2 是同一个 Bean");
          }else{

            System.out.println("hw 和 hw2 不是同一个 Bean");
          }

}

}
```

这个程序的行为非常简单，它先通过装载 applicationContext.xml 文件也就是上面配置 HelloWorld 的 XML 文件来实例化一个 ApplicationContext 对象，然后两次通过 getBean 方法取得两个引用，最后通过“==”操作符来判断这两个引用是否为同一对象。控制台输入的结果是我们所预期的：“hw 和 hw2 不是同一个 Bean”，在 applicationContext.xml 中去掉 singleton=”false”这一个属性的配置，那么控制台的结果变为“hw 和 hw2 是同一个 Bean”。

2.3.3 创建一个 Bean

还是以上面的 HelloWorld 为例，来看一看一个 Bean 对象是在什么时候创建的。上面没有给出 HelloWorld 的源代码，因为上面并不关心其具体的实现，这里给出了这个代码：

```
package test;

public class HelloWorld {
    public HelloWorld(){
    System.out.println("HelloWorld 被实例化");
    }


}
```

这个程序简单到只有一个构造方法。在构造方法中打印了一条消息。如果这个构造方法被调用也就是说如果产生了一个 HelloWorld 实例。这里仍然使用前面

ApplcationContext.xml 的配置，稍微修改下 TestClient 如下所示：

```
package test;



import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

//测试一个 Bean 被实例化的时机
public class TestClient {
public static void main(String[] args) {

    ApplicationContext                                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
    System.out.println("产生了一个 ApplicationContext 对象");
    HelloWorld hw=(HelloWorld)appContext.getBean("HelloWorld");
    System.out.println("得到一个 HelloWorld 对象");


}

}
```

这个程序结合上面的 HelloWorld，他们通过打印消息的先后顺序来测试行为的发生时机。运行这个程序结果如下：

```
产生了一个 ApplicationContext 对象
HelloWorld 被实例化
产生了一个 HelloWorld 对象
```

通过这个结果可以发现通过 getBean 方法就可以实例化一个 Bean，而且是通过调用无参构造方法进行的，但这并不是唯一的方式。下面将介绍另外两者方式：

1. 使用静态工厂方法实例化一个 Bean

关于工厂模式将在第七章详细介绍，可以简单的认为工厂是用来生产某一类事务的容器，具体一点就是可以看成它是一个实例化其它类的一个类。为了使用实例说明问题，修改上面的程序，先设计一个接口：

```
package test;

public interface IGreeter {
//唯一的方法：问候
public void greeting();
}
```

这个接口的设计是非常简单的，它只具有一个方法，那就是问候 greeting()。下面用两个类分别实现这个接口，其中一个只需要修改下上面提到的 HelloWorld，修改后的代码如下所示：

```
 package test;

public class HelloWorld  implements IGreeter{

   public void greeting() {

System.out.println("你好，世界");
}
```

```
}
```

它实现了 greeting 方法，如果方法被调用将打印“你好，世界”，然后还需要一个实现类：

```
package test;

public class HelloSpring implements IGreeter {

public void greeting() {
    System.out.println("你好，Spring");

}

}
```

上面这个程序也是非常简单到可以自解释的，本节为了得到这两个类的实例需要写一个静态工厂：

```
package test;

public class GreetFactory {
/**
 * 获取 IGreeter 实例的静态方法
 * @param name 通过这个名称决定实例化哪个类
 * @return 一个 IGreeter 实例
 */
public static IGreeter getGreeter(String name){
    if(name.equals("world")){
        return new HelloWorld();
    }
    else{
        return new HelloSpring();
    }
}
}
```

这个工厂的核心逻辑是通过不同的名称取得对应的对象，但是调用者不用关心 new 操作过程，静态工厂封闭了这个细节，同时由于它是静态的所以也不需要实例化这个工厂才能调用 getGreeter 方法。那么 Spring 如何通过调用这个 GreetFactory 方法来实例化一个 Igreeter 呢，是通过配置 XML。在 2.3.2 中讲述了如何在 Spring 配置文件中配置 Bean，不过所涉及的属性都是最基本的，通过这种最基本的配置也就是说具有 id 和 class 属性就可以将一个 Bean 注册到 Spring 环境中，并由 Spring 容器所管理，class 属性指定了 Bean 这个类的类路径，Spring 容器的确是通过它进行实例化一个 Bean 的，但现实情况总是复杂多变的，这一节讲述的是将通过静态工厂去获得一个 Bean 的实例，class 属性值则应该是这个工厂类的路径，但这还不够，还需要明确指定静态的工厂方法，这一点是通过 factory-mothod 进行设置的。因此上面示例的配置文件应该是这样的：

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

```
<beans>
<!-- 通过静态工厂配置 Bean -->
<bean          id="greeter"          class="test.GreetFactory"
factory-method="getGreeter">
<!-- 配置参数 -->
<constructor-arg><value>world</value></constructor-arg>
</bean>
</beans>
```

可以看到这时候的 class 是静态工厂的类路径，而且明确指定了 getGreeter 方法，但由于这个方法是需要提供参数的，所以还需要通过<constructor-arg>为其注入一个参数，这里是 world.写好配置文件后，就可以其编写测试类了：

```
package test;



import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试通过静态工厂获得 bean 的实例
public class TestClient {
public static void main(String[] args) {
//取得配置信息
    ApplicationContext                                  appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
    //获得 IGreeter Bean
    IGreeter greeter=(IGreeter)appContext.getBean("greeter");
    //调用 greeting 方法
    greeter.greeting();


}

}
```

本程序先通过一般的方法获得一个容器对象即这里的 appContext 然后通过它的 getBean 方法获得一个 IGreeter 对象，这里只能通过 IGreeter 接口进行强制类型转化.然后调用其提供的静态方法，也就是在配置文件中 factory-mothod 属性的值。运行本程序后的结果如下：

```
你好，世界
```

的确如愿以偿，工厂返回了 HelloWorld 的实例。下面把 world 修改成 Spring。修改后的配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 通过静态工厂配置 Bean -->
<bean          id="greeter"          class="test.GreetFactory"
```

```
factory-method="getGreeter">
<!-- 配置参数 -->
<constructor-arg><value>spring</value></constructor-arg>
</bean>
</beans>
```

上面的配置文件只是将原来的“世界”改成了 spring，再运行 TestClient 结果如下：

```
你好，Spring
```

对，结果就应该是这样的，当然只要不是“世界”就会打印“你好，Spring”.这里讲述了通过静态方法实例化一个 Bean。下面将讲述通过实例工厂方法实例化一个 Bean。

2. 使用实例工厂方法实例化一个 Bean

实例工厂和静态工厂最基本的不同点在于，使用静态工厂的静态方法不需要产生一个工厂实例，可以通过 class.function 方式直接调用，而实例工厂为客户提供的方法不是静态的，在使用其方法前必须先通过 new 操作产生一个工厂实例。也因此导致它们的存在方式和生命周期不一样。修改上面的 GreetFactory 将其改成一个非静态的工厂：

```
package test;

public class GreetFactory {
/**
 * 获取 IGreeter 实例的方法
 * @param name 通过这个名称决定实例化哪个类
 * @return 一个 IGreeter 实例
 */
public  IGreeter getGreeter(String name){
    if(name.equals("world")){
        return new HelloWorld();
    }
    else{
        return new HelloSpring();
    }
}
}
```

其它类都保持不变，包括 TestClient。但是由于上面提到的两者的差异，在 Spring 中配置这个工厂 Bean 时，与静态工厂是有所不同的，配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置非静态工厂 Bean -->
<bean id="GreetFactory" class="test.GreetFactory"></bean>
<!-- factory-bean 必须是一个已经存在了的 Bean-->
<bean      id="greeter"           factory-bean="GreetFactory"
factory-method="getGreeter">
<!-- 配置参数 -->
<constructor-arg><value>world</value></constructor-arg>
</bean>
```

```
</beans>
```

下面来分析下这个配置文件：首先像配置普通 Bean 一样用最基本的属性即 id 和 class 配置非静态工厂，将其注册到 Spring 环境中，使得能够被 Spring 容器所管理。与静态工厂不同的是在配置 greeter 这个 bean 的时候，不再是使用 class 属性指定一个具体 Bean 的类路径，而是通过 facotry-bean 这个属性指定一个在前面已经配置好了的 Bean 的 id.这种方式似乎没那么直接，不过却显得简洁，但 Spring 绝对不是因为这个理由而作出这一改变的，真实的原因在前面讲述静态工厂和非静态工厂的区别时候已经说出了其本质。运行上面的测试客户端也就是 TestClient，注意不用修改这个测试类。运行结果当然是我们期望的：

```
你好，世界
```

2.4Bean 的生命周期

本节将着重讲解 Bean 的生命周期。这里提到的 Bean 都只是针对单件 Bean，因为对于一个非单件 Bean，Spring 容器是不可能知道它什么时候不再使用，什么时候被销毁，这个工作交给了客户端，Spring 容器负责诞生一个非单件 Bean.然后就不再管它的死活了。而单件 Bean 由于在整个过程，都是一个实例，客户端始终调用的是一个实例，也就是说它们是共享这个 Bean 的，因此容器需要管理它的创建和销毁。

2.2.0 创建一个单件 Bean

单件和非单件 Bean 在代码级上没有任何区别，比如创建一个名为 HelloWorld 的 Bean。

（注意在 2.4 节中所有的示例都是基于本小节创建的）。

```
package test;
//一个简单的 Bean
public class HelloWorld  {
//消息:问候的话
private String greetMsg;
//构造函数
public HelloWorld(){
System.out.println("你好,世界");
}
//----set 和 get 方法区--------------

public String getGreetMsg() {
return greetMsg;
}

public void setGreetMsg(String greetMsg) {
this.greetMsg = greetMsg;
}

}
```

这个代码只具有一个属性，就是问候的语句。然后就是它的 getter 和 setter 方法，以及它的构造函数，构造函数中具有一条打印语句，用来测试是否创建了一个实例。是否为单件从代码上并看不出什么特征。但却可以通过在 Spring 的配置文件中配置它

使得这个类只会产生一个实例。配置文件如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean -->
<bean id="hello" class="test.HelloWorld" >
</bean>
</beans>
```

是的，hello 这个 Bean 的配置是再普通不过了的，它只具有最基本的 id 和 class 属性，事实上，在默认情况下它已经被配置成一个单件了。到此一个 Bean 注册完毕。那么注册并不代表这个单件实例就被创建了，只有当配置文件被容器所加载的时候，单件实例才会被创建。如下面的测试代码：

```
package test;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;


public class TestClient {
public static void main(String[] args) {
//取得配置信息
new
FileSystemXmlApplicationContext("src/applicationContext.xml");


}

}
```

这个类很简单，它简单到只有一个 new 语句，而正是这个 new 语句导致了一个行为的发生，那就是产生了一个容器对象，它装载了 applicationContext.xml 这个配置文件，也就是上面配置 hello 的文件。运行这个程序在控制台中打印了结果：

```
你好，世界
```

到现在这个 Bean 才算是创建完毕。

2.2.2 Bean 的属性值注入

上面的配置告诉容器 hello 这个 Bean 就交托给它所管理了，通过更加详细的配置就可以完成一些依赖注入。下面介绍如何将注入一个属性值。

hello 这个 Bean 具有唯一的属性 `greetMsg`。同时它具有标准的 `setter` 方法，具备了这个条件后，就可以通过<property>元素进行属性值注入。配置文件如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean -->
```

```
<bean id="hello" class="test.HelloWorld" >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
</bean>
</beans>
```

配置文件在原有的基础上在<bean>的标签体里面添加了<property>元素，它具有一个 name 属性，这个 name 属性的值必须是在 HelloWorld 这个类中定义了的属性，关是这点不够，它还必须具备 public setter 方法，如下：

```
......
public void setGreetMsg(String greetMsg) {
this.greetMsg = greetMsg;
}
......
```

<value>元素的值就是注入的值。为了测试它需要写个测试类代码如下：

```
package test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {
public static void main(String[] args) {
//取得配置信息
ApplicationContext                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean
HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//输出消息
System.out.println(hw.getGreetMsg());


}

}
```

这个测试类首先取得一个工厂实例即 appContext 然后通过这个工厂获得 hello 这个 Bean，为了验证 appContext 工厂是否已经通过 setter 方法将 greetMsg 注入了进去，则需要调用 greetMsg 的 getter 方法即 getGreetMsg 取得 greetMsg 的值，最后将其打印出来。运行此程序结果如下：

```
你好，世界
你好，这个一个测试
```

由于用得是和前一节同样的例子也就是没有修改 HelloWorld 所以也同时打印出了“你好，世界”.

2.2.3 Bean 的属性值注入后行为

如果需要在上面完成注入后再做一些初始化工作，那么该如何配置呢？首先需要 HelloWorld 这个类，为其添加一个方法，这里称其为 init。当然这个名称可以随意。

修改后的代码如下：

```
package test;
//一个简单的 Bean
public class HelloWorld  {
//消息:问候的话
private String greetMsg;
//----set 和 get 方法区--------------
public HelloWorld(){
System.out.println("创建了一个实例...");
}
public String getGreetMsg() {
return greetMsg;
}

public void setGreetMsg(String greetMsg) {
//属性在通过 setter 方法进行注入的时候会输出这条语句。
System.out.println("进行属性注入...");
this.greetMsg = greetMsg;
}
//属性注入后行为
public void init(){
System.out.println("在这里可以做一些初始化工作...");
}

}
```

代码主要修改了两个地方，第一个地方就是在 setGreetMsg 中添加了输出语句，当这个 setter 方法被调用的时候好跟踪。第二个地方就是新增加了一个方法 init.它也是输出一条语句。通过这两条语句的打印先后顺序就可以判断两个方法的调用顺序，从而了解 Spring 容器对 Bean 的管理行为。init 方法要被调用还需要在配置文件中进行配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean 通过指定 init-method 值来指定一个初始方法 -->
<bean id="hello" class="test.HelloWorld" init-method="init" >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
</bean>
</beans>
```

分析下这个配置文件，通过和前一小节的对比，它只是在<bean>元素中多配置了一个称为 init-method 的属性，也就是粗体部分内容。这个属性的值就是刚才在 HelloWorld 中新增的方法 init.不需要修改 TestClientl 来测试这一行为：

```
package test;
```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {
public static void main(String[] args) {
//取得配置信息
ApplicationContext                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean
HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//输出消息
System.out.println(hw.getGreetMsg());


}

}
```

这个测试类相比前一小节没有任何的改变，运行它可以看到结果：

```
创建了一个实例...
进行属性注入...
在这里可以做一些初始化工作...
你好，这个一个测试
```

这个结果很清楚的告诉了我们 init 行为的确发生了，而且是在 setter 方法之后。

通过配置 init-method 属性可以达到调用某个初始化方法的目的，但 Spring 还提供了另一种方法已达到同样的效果：即实现 org.springframework.beans.factory.InitializingBean 这个接口。这个接口中定义了一个 afterPropertiesSet 方法。它相当于 init 方法。实现这个方法就可以达到同样的效果。修改 HelloWorld 代码后如下所示：

```
package test;

import org.springframework.beans.factory.InitializingBean;

//一个简单的 Bean
public class HelloWorld  implements InitializingBean{
//消息:问候的话
private String greetMsg;
//----set 和 get 方法区--------------
public HelloWorld(){
System.out.println("创建了一个实例...");
}
public String getGreetMsg() {
return greetMsg;
}
```

```
public void setGreetMsg(String greetMsg) {
//属性在通过 setter 方法进行注入的时候会输出这条语句。
System.out.println("进行属性注入...");
this.greetMsg = greetMsg;
}
//属性注入后行为
public void afterPropertiesSet() throws Exception {
System.out.println("在这里可以做一些初始化工作...");

}


}
```

修改配置，现在已经不再需要配置 init-method 方法了。修改后的配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean -->
<bean id="hello" class="test.HelloWorld"  >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
</bean>
</beans>
```

这个配置文件去掉了 init-method 属性。因为在 HelloWorld 这个类中已经实现了 org.springframework.beans.factory.InitializingBean 这个接口。客户端 TestClient 还是不变，运行后结果如下：

```
创建了一个实例...
进行属性注入...
在这里可以做一些初始化工作...
你好，这个一个测试
```

2.2.4 Bean 的销毁前行为

既然有初始化行为，那也一定会有销毁行为，上一小节中提到初始化具有两种方法，同样销毁也具有两种方式

- 在配置文件中通过配置 destory-method 属性指定一个销毁前方法
- 实现 org.springframework.beans.factory.DisposableBean 这个接口

先来看第一种方式，读者不难举一反三，在一切工作的开始必须得为 HelloWorld 添加一个方法，这里命名为 destory 如下：

```
package test;


import org.springframework.beans.factory.InitializingBean;
```

```
//一个简单的 Bean
public class HelloWorld  implements InitializingBean{
//消息:问候的话
private String greetMsg;
//----set 和 get 方法区--------------
public HelloWorld(){
System.out.println("创建了一个实例...");
}
public String getGreetMsg() {
return greetMsg;
}

public void setGreetMsg(String greetMsg) {
//属性在通过 setter 方法进行注入的时候会输出这条语句。
System.out.println("进行属性注入...");
this.greetMsg = greetMsg;
}
//属性注入后行为
public void afterPropertiesSet() throws Exception {
System.out.println("在这里可以做一些初始化工作...");

}
//销毁方法
public void destroy(){
System.out.println("在销毁前还可以在这里做一些事情...");
}


}
```

粗体部分是新添加的方法，它打印了一条消息当这个方法被调用的时候。在现实情况下，可以使用相关的逻辑代码代替这条打印语句，这里仅为了测试使用。下面就是配置它了：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean 通过指定 destroy-method 值来指定一个初始方法 -->
<bean            id="hello"            class="test.HelloWorld"
destroy-method="destroy">
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
</bean>
</beans>
```

配置 destory 和 init 的区别仅在于两个属性名称不同，配置方法都一样，destory-method 的值为刚才新添加的 destory 方法。测试程序不变。运行结果如下：

```
创建了一个实例...
```

```
进行属性注入...
在这里可以做一些初始化工作...
你好，这个一个测试
在销毁前还可以在这里做一些事情...
```

很遗憾的是，如果在你的应用中做同样的测试可能并不会打印最后一条语句，的确，我们要承认这是 Spring 不完美的地方之一。不过，它值得我们学习的地方是多数的，再来看第二种方式，它是依靠实现 org.springframework.beans.factory.InitializingBean 这个接口实现上面同样的效果。修改 HelloWorld 代码如下：

```
package test;


import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

//一个简单的 Bean
public class HelloWorld implements
InitializingBean,DisposableBean{
//消息:问候的话
private String greetMsg;
//----set 和 get 方法区--------------
public HelloWorld(){
System.out.println("创建了一个实例...");
}
public String getGreetMsg() {
return greetMsg;
}

public void setGreetMsg(String greetMsg) {
//属性在通过 setter 方法进行注入的时候会输出这条语句。
System.out.println("进行属性注入...");
this.greetMsg = greetMsg;
}
//属性注入后行为
public void afterPropertiesSet() throws Exception {
System.out.println("在这里可以做一些初始化工作...");

}
//销毁行为
public void destroy() throws Exception {
System.out.println("使用 DisposaleBean 方式的销毁方法被调用");

}

}
```

这个代码中的 destory 方法是 DisposableBean 这个接口定义的，上一例中的 destory 方法是自定义的。修改配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
```

```
<beans>
<!-- 配置一个单件 Bean ->
<bean id="hello" class="test.HelloWorld"   >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
</bean>
</beans>
```

通过和前面的比较可以发现这里的修改只是去掉了 destory-method 这个属性的设定。读者一定会问如果不去掉呢？当然，这也是可以的，而且两者定义的方法都会被调用。不过容器会先执行 DisposableBean 接口中定义的方法，然后执行 destory-method 属性指定的方法。对于 init 行为也是同样的规则。

2.5 管理依赖关系

在前面讲述的关于 bean 的管理几乎只是一个单一的 Bean。事实上，事物是不可能与外界毫无关联的单独存在。所以存在一个容器中的 Bean 彼此之间都可以会有各种联系。在 Spring 中更喜欢把各种联系称为依赖。而把通过容器与配置文件管理依赖称为依赖注入，或者控制反转。

2.5.1 使用 lazy-load

在默认情况下，如果是使用 BeanFactory 管理 Bean 的时候只有当客户访问这个 Bean 的时候才实例化，而如果是使用 BeanFactory 的子接口 ApplicationContext 容器进行管理 Bean 的时候，会在加载完配置文件后就完成了对所有的单件 Bean 的实例化。ApplicationContext 的这一行为是可以通过修改 bean 元素的 lazy-load 属性进行改变的方法是设置其为 true。ApplicationContext 就会对 Bean 进行惰性实例化，也就是当 Bean 使用时才实例化。下面为此举个具体的实例。首先编写 HelloWorld，其外貌基本是类似于 2.4 节中使用的 HelloWorld：

```
package test;
//一个简单的 Bean
public class HelloWorld  {
//消息:问候的话
private String greetMsg;
public HelloWorld(){
System.out.println("创建了一个实例...");
}
//----set 和 get 方法区--------------
public String getGreetMsg() {
return greetMsg;
}

public void setGreetMsg(String greetMsg) {
//属性在通过 setter 方法进行注入的时候会输出这条语句。
System.out.println("进行属性注入...");
this.greetMsg = greetMsg;
```

```
    }


    }
```

通过前面所学，读者应该不难为其编写配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean  通过配置 lazy-init 属性使得 hello 这个 Bean 被惰性实例化-->
<bean id="hello" class="test.HelloWorld"  lazy-init="true" >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
</bean>
</beans>
```

可见 hello 这个 Bean 被配置成了 lazy-init。为了验证这一配置的行为，需要为其编写一个测试程序，这个测试如下所示：

```
package test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {
public static void main(String[] args) {
//取得配置信息
ApplicationContext                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean
System.out.println("appContext 被创建");
HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//输出消息
System.out.println(hw.getGreetMsg());




}

}
```

与前面的测试程序不同的地方是，这里新增加了一条输出语句，它的位置在 appContext 创建之后，也就是 applicationContext.xml 文件加载之后，在 getBean 方法之前，也就是在 hello 这个 Bean 被使用之前。运行这个程序得出如下结果：

```
appContext 被创建
```

```
创建了一个实例...
进行属性注入...
你好，这个一个测试
```

通过消息的打印顺序可以知道实例的创建是延迟发生的。否则去掉 lazy-init 属性重新运行客户端程序发现结果变为：

```
创建了一个实例...
进行属性注入...
appContext 被创建
你好，这个一个测试
```

这个差别是明显的。尽管 Spring 具有高配置的特性，但并不建议对它的默认设置进行修改，如这里的 lazy-init 最好保持默认，原因在于当实例化动作被延迟时就会减少一些对 bean 的依赖检查的验证工作，这会导致在后期出现一些不确定的后果，也不符合软件设计的尽量验证尽早验证的经验规则。

2.5.2 再谈 bean 的属性注入

在讲述 Bean 的生命周期的 2.2.2 节中介绍了使用 value 方式进行属性注入，这是最直接和最简单的一种方式，但 value 只能接受字符串。然后 Spring 容器将 java.lang.String 类型转化成其它类型，对于基本类型这个转化都不会有什么问题，那么如果是一个日期类型，情况会如何呢？修改一下 HelloWorld 这个程序后：

```
package test;

import java.util.Date;

//一个简单的 Bean
public class HelloWorld  {
//消息:问候的话
private String greetMsg;
//当天日期
private Date date;
//构造函数
public HelloWorld(){
System.out.println("HelloWorld 被创建...");
}

//setter 和 getter 方法区
public String getGreetMsg() {
return greetMsg;
}

public void setGreetMsg(String greetMsg) {

this.greetMsg = greetMsg;
}
public Date getDate() {
return date;
}
public void setDate(Date date) {
this.date = date;
}
```

```
}
```

和以往的 HelloWorld 不同的是，这个 HelloWorld 具有一个非 String 类型的属性，事实上，在实际应用中这是非常正常的。既然是这样，那么如何为其在配置文件中注入属性值呢。可能你会这样做：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!-- 配置一个单件 Bean 通过配置 lazy-init 属性使得 hello 这个 Bean 被惰性实例化-->
<bean id="hello" class="test.HelloWorld" >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
<!-- 一种自厢情愿的做法 -->
<property name="date">
<value>2007.10.10</value>
</property>
</bean>
</beans>
```

Value 值直接使用一个我们自己认识的日期进行注入，看上去很自然，那么为其编写一个测试程序：

```
package test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {
public static void main(String[] args) {
//取得配置信息
ApplicationContext appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean

HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//输出消息
System.out.println(hw.getGreetMsg());
//输出当天时间
   System.out.println("今天是:"+hw.getDate());



}

}
```

这个测试程序没什么和前面有特别的地方，仅仅是加了一最后一条测试代码。我们的期望输出应该

```
……
今天是:2007.10.10
```

然而当运行这个程序的时候发现结果是这样的：

```
……
Caused by: org.springframework.beans.TypeMismatchException:
Failed to convert property value of type [java.lang.String] to
required type [java.util.Date] for property 'date';
……
```

是的出现了一堆的异常，从异常的关键词可以看出是 date 的类型不匹配。当然你不要期望将 2007.10.10 改成 2007-10-10 就会侥幸通过。那么 Spring 是如何为 Date 注入一个可接受的值呢？在这种情况下，hello 这个 Bean 不再能够单独的存在一个容器中了，它需要别的 Bean 的支持。修改一下配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>

<!-- 配置一个单件 Bean 通过配置 lazy-init 属性使得 hello 这个 Bean 被惰性实例化-->
<bean id="hello" class="test.HelloWorld" >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
<!-- 一种自厢情愿的做法 -->
<property name="date">
<!-- 通过 value 元素进行属性注入 now 是前面出现了的一个 bean 的 id 值 -->
<!-- 再定义一个 Date 类型的 ben -->
<bean class="java.util.Date"> </bean>
</property>
</bean>
</beans>
```

粗体部分原本是<value>元素所在的领地，现在被一个<bean>属性占据了。再仔细点会发现这个<bean>元素并不具备 id 值。这需要解释下了：这个 Bean 是存在 hello 这个 Bean 的标签体内部的，也就是说它是一个嵌套的 Bean。笔者更喜欢称其为局部 bean. 与普通的 bean 比较它的特别之处就是只对所在的 bean 可见，对外部容器是不可见的。所以不要企图在客户端程序中通过 getBean 方法获得这个 Bean 的实例。也不要企图在别的地方使用<ref>元素（接下来会介绍）来引用这个 Bean。言归正传，现在再运行下测试程序，终于得到了我们期望已久的答案：

```
HelloWorld 被创建...
你好，这个一个测试
今天是:Wed Oct 10 15:57:42 CST 2007
```

那么，如果这个局部 bean 具有一些共享价值呢？换句话说就是希望它能被其它成员所访问，对容器可见，能被客户端访问。对于 Spring 来说，这不是一个过分的要求，它可以很轻松的做到这一点。只需要修改下配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!--配置一个日期 Bean -->
<bean id="now" class="java.util.Date"> </bean>
<!-- 配置一个单件 Bean 通过配置 lazy-init 属性使得 hello 这个 Bean 被惰性实例化-->
<bean id="hello" class="test.HelloWorld" >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>
<property name="date">
<!-- 通过 ref 元素进行属性注入 now 是前面出现了的一个 bean 的 id 值 -->
<ref local="now"/>
</property>
</bean>
</beans>
```

<ref>这个元素具有两个属性 local 和 bean. 由于 Spring 的配置文件可以是多个，为什么？对于一个非常大的应用如果只维护一个配置文件，那会显得混乱难以维护。Local 的值指的是本文件中定义了的 bean 的 id 值。而 bean 则指外部文件定义的 bean 的 id 值。

本着循序渐进的学习精神，下面继续修改 HelloWorld 程序，使得它复杂化：

```
package test;

import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

//一个简单的 Bean
public class HelloWorld {
//消息:问候的话
private String greetMsg;
//当天日期
private Date date;
//集合属性区
private List<String> alist;
private Map<String, String> aMap;
private Set<String> aSet;
private Properties aProperty;
//构造函数
public HelloWorld(){
System.out.println("HelloWorld 被创建...");
}

//setter 和 getter 方法区
public String getGreetMsg() {
```

```
    return greetMsg;
    }

    public void setGreetMsg(String greetMsg) {

    this.greetMsg = greetMsg;
    }
    public Date getDate() {
    return date;
    }
    public void setDate(Date date) {
    this.date = date;
    }

    public List<String> getAlist() {
    return alist;
    }

    public void setAlist(List<String> alist) {
    this.alist = alist;
    }

    public Map<String, String> getAMap() {
    return aMap;
    }

    public void setAMap(Map<String, String> map) {
    aMap = map;
    }

    public Properties getAProperty() {
    return aProperty;
    }

    public void setAProperty(Properties property) {
    aProperty = property;
    }

    public Set<String> getASet() {
    return aSet;
    }

    public void setASet(Set<String> set) {
    aSet = set;
    }


    }
```

这个 HelloWorld 相比以前的 HelloWorld 其复杂之处在于属性的类型。粗体部分是几个集合类型的属性。那么如何给它们注入值呢？对于其配置信息要稍微复杂点：

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

```
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 唯一的根元素 -->
<beans>
<!--配置一个日期 Bean  -->
<bean  id="now" class="java.util.Date"> </bean>
<!-- 配置一个单件 Bean  -->
<bean id="hello" class="test.HelloWorld"  >
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好，这个一个测试</value>
</property>

<property name="date">
<ref local="now"/>
</property>
<!-- 配置 list 属性注入 -->
<property name="alist">
<list>
<value>list 元素 1</value>
<value>list 元素 2</value>
<value>list 元素 3</value>
</list>
</property>
<!-- Map 属性注入 -->
<property name="AMap">
<map>
<entry key="map_key1">
<value>map 值 1</value>
</entry>
<entry key="map_key2">
<value>map 值 2</value>
</entry>
<entry key="map_key3">
<value>map 值 3</value>
</entry>
</map>
</property>
<!-- Set（集合）属性注入 -->
<property name="ASet">
<set>
<value>set 的值 1</value>
<value>set 的值 2</value>
<value>set 的值 3</value>
</set>
</property>
<!-- props 注入 -->
<property name="AProperty">
<props>
<prop key="pkey1">prop 值 1</prop>
<prop key="pkey2">prop 值 2</prop>
<prop key="pkey3">prop 值 3</prop>
```

```
    </props>
    </property>

    </bean>
    </beans>
```

下面解释下上面的配置内容：一个 collection 类型的属性直接使用<collection><value></value></collection>这种方式就可以为这个 collection 添加一个对象，当然如前面所说一般使用<value>注入的都是 java.lang.String 类型的对象，或者可以从这个对象转化的对象。如果要添加的是一不能由 java.lang.String 类型转化，则可以使用<bean>或者<ref>元素进行对象注入。对于 Map 类型的值注入，也是一样的，

为了测试这些注入，需要为它编写一个测试程序，这个测试程序可能也要比以往复杂点：

```
package test;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;

import java.util.Set;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {

//  输出 collection 里面的元素
private static void listCollection(Collection<String> collection){
    for (Iterator iter = collection.iterator(); iter.hasNext();) {
        String element = (String) iter.next();
        System.out.println(element);

    }
}

//输出 Map 类型的值
private static void listMap(Map map){
Set keys=map.keySet();
for (Iterator iter = keys.iterator(); iter.hasNext();) {
    String key = (String) iter.next();
    String value=(String)map.get(key);
    System.out.println("键："+key+" 值："+value);

}
```

```
}


public static void main(String[] args) {
//取得配置信息
ApplicationContext                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean

HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//取得各种集合并输出其结果
//取得 List 对象
System.out.println("list 的值：");
listCollection(hw.getAlist());
//取得 Set 对象
System.out.println("Set 的值：");
listCollection(hw.getASet());
//取得 Map 对象
System.out.println("Map 的值:");
    listMap(hw.getAMap());
   //取得 Properties 对象
    System.out.println("Prop 的值：");
    //Properties 这个类是实现了 Map 接口的，所以可以向上映射。
    listMap(hw.getAProperty());


}

}
```

这个类新添加了两个静态方法一个是 listCollection，它接受 List，Set 对象并将它们的内容打印出来，另一个是 listMap 它接受 Map 和 Properties 对象。它们都体现了向上映射(upcasting)的思想。运行这个程序得到如下的结果：

```
HelloWorld 被创建...
list 的值：
list 元素 1
list 元素 2
list 元素 3
Set 的值：
set 的值 1
set 的值 2
set 的值 3
Map 的值:
键：map_key1 值：map 值 1
键：map_key2 值：map 值 2
键：map_key3 值：map 值 3
Prop 的值：
键：p_key3 值：prop 值 3
键：p_key2 值：prop 值 2
键：p_key1 值：prop 值 1
```

需要注意的是<property>元素的 name 值大小写是敏感的，如将上面的 AMap 改成

aMap.则在客户程序运行时出现下面这样的错误提示。

```
……
Bean property 'aMap' is not writable or has an invalid setter method.
Did you mean 'AMap'?
……
```

2.5.3 依赖检查

初学者在实际开发时在进行属性配置时总是会导致一些错误的出现。Spring 容器为此做出了一些努力，及时的对配置信息进行检查。Spring 是通过 bean 元素的 dependcy-check 属性进行依赖检查。共有四种模式：simple，object，all，none。在默认情况下不进行依赖检查。

- Simple 模式

Simple 模式意为简单检查，它仅对于基本数据类型，字符串类型和集合类型进行依赖检查。下面使用一个实例来说明其行为。首先编写一个类，仍然称为 HelloWorld：

```
package test;

//一个简单的 Bean
public class HelloWorld {
// 一个 String 类型的属性
private String greetMsg;

// --------setter 方法和 getter 方法区---------
public String getGreetMsg() {
    return greetMsg;
}

public void setGreetMsg(String greetMsg) {
    this.greetMsg = greetMsg;
}

}
```

这个类里面只包含一个 java.lang.String 类型的属性，所以它是 simple 模式的检查的范围之内。同时可以知道 greetMsg 如果没有被注入值，那么就应该是 null.下面为其编写配置文件：

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

<!-- 配置一个单件 Bean  -->
<bean id="hello" class="test.HelloWorld"  >
</bean>
</beans>
```

这个配置文件关于 HelloWorld 的配置是非常简单的。它没有为其进行属性注入。下面为其编写测试程序 TestClient:

```
package test;
```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {




public static void main(String[] args) {
//取得配置信息
ApplicationContext                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean

HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//输出消息
   System.out.println(“消息：”+hw.getGreetMsg());

}

}
```

这个测试程序的行为是简单的取得 hello 这个 bean，然后通过 getter 方法将 greetMsg 的值输出。结果是：

```
消息：null
```

这一切都是在我们的预料之中，但，如果这时候修改下配置文件，修改后的配置文件如下：

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

<!-- 配置一个单件 Bean  运用 simple 模式的依赖检查-->
<bean            id="hello"            class="test.HelloWorld"
dependency-check="simple">
</bean>
</beans>
```

与上面配置文件不同的是，修改后的配置文件在<bean>标签中新添加了 dependcy-check 的设置。这里使用 simple 模式的依赖检查。测试程序不变，并运行后发现出现下面异常信息：

```
……
Unsatisfied dependency expressed through bean property 'greetMsg':
Set this property value or disable dependency checking for this
bean.
……
```

异常的意思是告诉开发人员 greetMsg 这个属性需要进行属性设置。于是你可能会这样做。不修改配置文件而是修改 HelloWorld.修改成：

```
package test;

//一个简单的 Bean
public class HelloWorld {
// 一个 String 类型的属性
private String greetMsg="hello";

// --------setter 方法和 getter 方法区---------
public String getGreetMsg() {
    return greetMsg;
}

public void setGreetMsg(String greetMsg) {
    this.greetMsg = greetMsg;
}

}
```

这里对 greetMsg 给了一个初值，但遗憾的是运行 TestClient 仍然是上面的异常信息。要让不报异常必须修改配置文件:

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

<!-- 配置一个单件 Bean  运用 simple 模式的依赖检查-->
<bean            id="hello"            class="test.HelloWorld"
dependency-check="simple">
<!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
<property name="greetMsg">
<value>你好,Spring</value>
</property>
</bean>
</beans>
```

现在的配置信息添加上了对 hello 这个 bean 的属性注入。运行测试程序得出结果：

```
消息：你好,Spring
```

为了验证 simple 模式仅对基本数据类型，String 类型，和集合类型进行依赖检查。修改 HelloWorld 成如下代码：

```
package test;

//一个简单的 Bean
public class HelloWorld {
// 一个 String 类型的属性
private String greetMsg;
//一个非基本类型属性
   private MyObject mobj;
```

```
    // --------setter 方法和 getter 方法区---------
    public String getGreetMsg() {
        return greetMsg;
    }

    public void setGreetMsg(String greetMsg) {
        this.greetMsg = greetMsg;
    }

    public MyObject getMobj() {
        return mobj;
    }

    public void setMobj(MyObject mobj) {
        this.mobj = mobj;
    }

    }
```

这个类不同的是，它新增了一个非基本类型的属性 Myobject。这个 Myobject 的默认值是为 null 的。MyObject 代码如下所示：

```
    package test;

    public class MyObject {

      private  String message="这个世界真奇妙！";

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    }
```

它是一个简单的 JavaBean.有一条消息属性。HelloWorld 配置文件如下：

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

    <!-- 配置一个单件 Bean  运用 simple 模式的依赖检查-->
    <bean            id="hello"            class="test.HelloWorld"
    dependency-check="simple">
    <!-- greetMsg 必须是已经定义的属性，且具有 public setter 方法 -->
    <property name="greetMsg">
    <value>你好,Spring</value>
    </property>
```

```
</bean>
</beans>
```

它和上面最近提到的配置文件没有什么区别，也就是说并没有为 hello 这个 bean 注入 mobj 的属性。并且这时候的依赖检查是 simple 模式。修改下测试程序：

```
package test;


import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试属性值的注入
public class TestClient {




public static void main(String[] args) {
//取得配置信息
ApplicationContext                                    appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean

HelloWorld hw=(HelloWorld)appContext.getBean("hello");
//输出消息
   System.out.println("消息："+hw.getGreetMsg());
   //输出 myobject 的消息属性
   System.out.println("消息："+hw.getMobj().getMessage());

}

}
```

这个测试程序相对前面最近提到的 TestClient，区别在于它多添加了一条消息输出语句。也就是访问了 MyObject 对象。运行它得到结果：

```
消息：你好,Spring
Exception in thread "main" java.lang.NullPointerException
at test.TestClient.main(TestClient.java:22)
```

通过结果可以发现，程序在运行到打印第二个消息的时候打出了异常。不过有些 Java 基本知识的读者应该知道，这个异常与 Spring 并无关系。它并不是由于出现依赖异常而发生的。因为在 HelloWorld 程序中 MyObject 本来就没有实例化的。所以可以通过修改 HelloWorld 这个程序而不是在配置文件中为其添加信息就可以避免这个异常的出现，修改后的 HelloWorld 如下：

```
package test;

//一个简单的 Bean
public class HelloWorld {
// 一个 String 类型的属性
private String greetMsg;
```

```
    //一个非基本类型属性
       private MyObject mobj=new MyObject();

    // --------setter 方法和 getter 方法区---------
    public String getGreetMsg() {
        return greetMsg;
    }

    public void setGreetMsg(String greetMsg) {
        this.greetMsg = greetMsg;
    }

    public MyObject getMobj() {
        return mobj;
    }

    public void setMobj(MyObject mobj) {
        this.mobj = mobj;
    }

    }
```

再次运行测试程序终于得出了正确的结果：

```
消息：你好,Spring
消息：这个世界真奇妙!
```

这个过程验证了 simple 模式并不检查非基本类型的属性是否进行了依赖注入。

- Object 模式

那么如果要对上面的 MyObject 属性也进行依赖检查呢。很容易，使用 Object 模式。Object 模式是针对非基本类型的一般对象进行依赖检查的。为了验证其行为，使用下面这个 HelloWorld 程序：

```
    package test;

    //一个简单的 Bean
    public class HelloWorld {

    //一个非基本类型属性
       private MyObject mobj;

    // --------setter 方法和 getter 方法区---------


    public MyObject getMobj() {
        return mobj;
    }

    public void setMobj(MyObject mobj) {
        this.mobj = mobj;
    }

    }
```

这个 HelloWorld 只含有一个非基本类型的属性 MyObject.这个 MyObject 类在上面已经提到过，这里不在累述。为其编写配置文件：

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

<!-- 配置一个单件 Bean 运用 object 模式的依赖检查-->
<bean id="hello" class="test.HelloWorld"
dependency-check="objects">

</bean>
</beans>
```

这里的 dependency-check 值改成了 objects 它表示对 hello 这个 bean 运用 Object 模式依赖检查。

下面编写测试程序：

```
package test;


import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationCont
ext;

//测试依赖检查
public class TestClient {




public static void main(String[] args) {
//取得配置信息
ApplicationContext appContext=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
//取得 Bean

HelloWorld hw=(HelloWorld)appContext.getBean("hello");

   //输出 myobject 的消息属性
   System.out.println("消息："+hw.getMobj().getMessage());

}

}
```

测试程序是很简单的。在没有对 Myobject 进行依赖注入的情况下调用了 getter 方法取得其属性。那么运行这个测试程序后结果如下：

```
……
Unsatisfied dependency expressed through bean property 'mobj':
```

```
Set this property value or disable dependency checking for this
bean.
……
```

这是我们第二次见到这样的异常了，它表示 mobj 这个属性没有进行依赖注入。因此，需要修改配置文件，修改后的配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Spring 配置文件的 dtd -->
<!DOCTYPE    beans    PUBLIC    "-//SPRING//DTD    BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- 根元素 -->
<beans>
<!-- 配置 MyObject -->
<bean id="myobj" class="test.MyObject"></bean>
<!-- 配置一个单件 Bean  运用 object 模式的依赖检查-->
<bean            id="hello"            class="test.HelloWorld"
dependency-check="objects">
<property name="mobj">
<!-- 属性注入 -->
<ref local="myobj"/>
</property>
</bean>
</beans>
```

这里运用了前面所学的属性注入，对 mobj 这个非基本类型的属性进行了引用方式的依赖注入。这时候再运行测试程序就可以得出正确的结果：

```
消息：这个世界真奇妙!
```

- all 模式和 none 模式

通过将 dependency-check 修改成 all。则表示对所有类型的属性进行检查。它的效果相当于同时运用了 simple 和 Object 模式。如果改成 none 则不对任何类型进行依赖检查。

2.6 Spring 中的 MVC 框架

Spring 的 web 框架是 Spring 的七大模块之一，核心概念是 MVC，MVC 是一种设计概念上的模式，Struts 实现了这种模式，当然这不意味着 Spring 就不能这么做。Spring 对 MVC 的实现在细节上相对 Struts 而言会有所差异，但更多的是相似，所以有了前一章对 Struts 的了解再来学习 Spring 的 MVC 就不会感到困难。

本节将使用一个用户登录的例子演示如何在 MyEclipse 这个 IDE 中开发一个基于 MVC 框架的 Spring Web 应用程序。用户登录的基本逻辑是这样的，用户从登录页面输入用户名和密码，尝试登录系统，如果登录失败比如用户名不存在将跳转到失败页面并提示失败信息，如果登录成功将跳转到成功页面显示欢迎信息。

2.6.1 建立一个具有 Spring 能力的 Web Project

在 MyEclipse 工作台中选择 File|New|Project 命令，将弹出如下对话框，如图 2-1 所示：

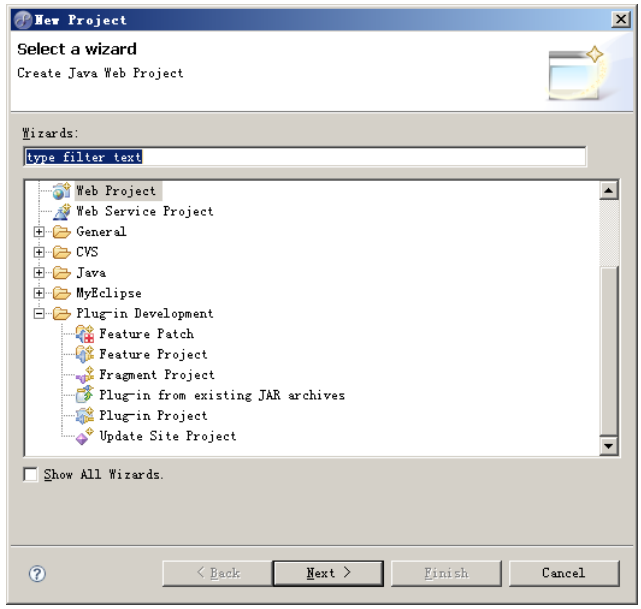

图 2-2 新建工程向导

选择 Web Project，单击 Next 按钮，进入下一步，如图 2-2 所示：

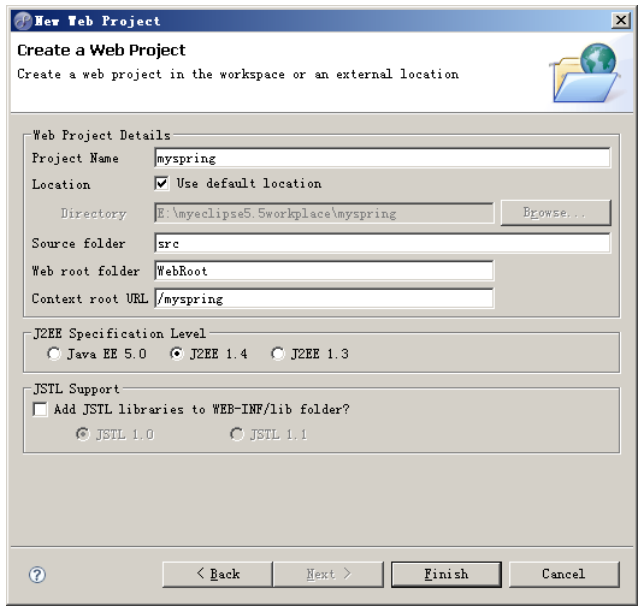

图 2-3 新建工程向导之填充信息对话框

这里只需要填写工程名也就是 Project Name，填充好后单击 Finish 按钮，一个 web Project 就创建好了，可以在 Package Explorer 视图中看到这个新建的工程如图 2-3 所示：

图 2-4 myspring 工程

现在创建好的工程还不具备 Spring 能力，Spring 所需要的 jar 都没有导入到工程，有两个办法，一个是在 Spring 的官方网站上下载到所有的 jar 包然后将其粘贴到 webRoot/WEB-INF/lib 目录下，第二种办法也是推荐的办法，就是使用 MyEclipse 自带

的功能自动添加所需要的 jar 包，办法是选中 myspring 这个工程，图 2-3 就是选中状态，然后在菜单栏中选择 MyEclipse|Add Spring Capabilities…命令如图 2-4 所示：

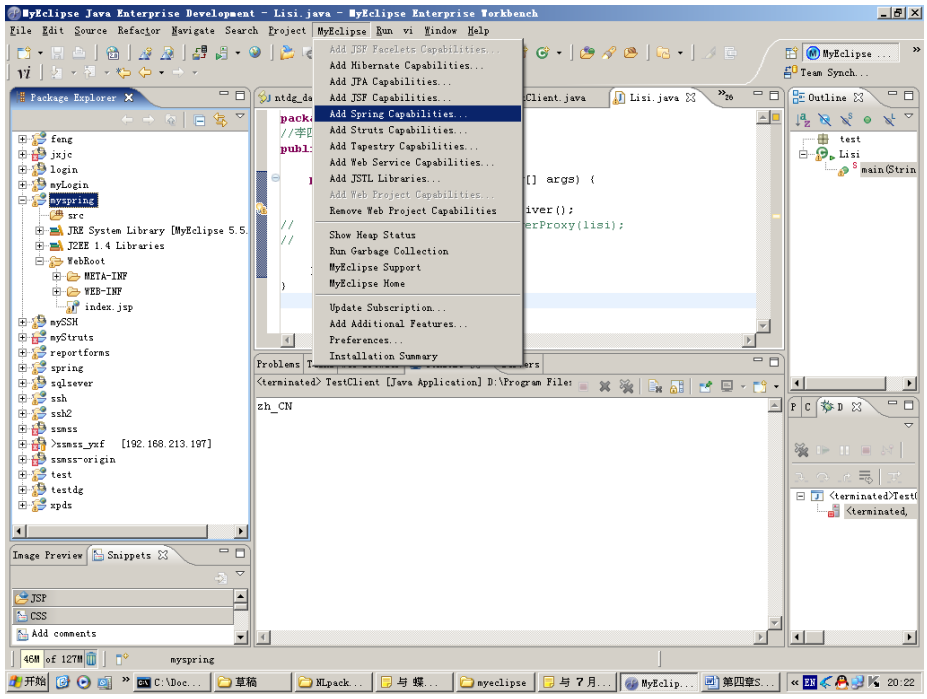

图 2-5 添加 Spring 能力

选择这个命令后将弹出一个添加向导如图 2-5 所示：

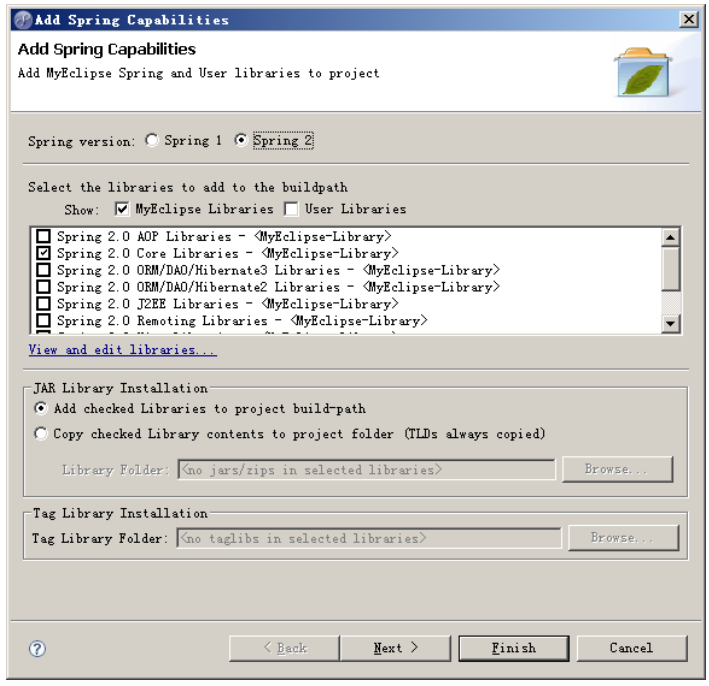

图 2-6 添加 Spring 能力向导

在这个选择向导中选中 Spring 2.0 Core Libraries 和 Spring 2.0 web Libraries，然后单击 Finish 按钮，当然也可以单击 Next 按钮去修改 Spring 配置文件的路径，但这里保持默认所以直接点击 Finish 完成 Spring 能力的添加。具有 Spring 能力的 myspring 工程会多出一些文件，具体的如图 2-6 所示：

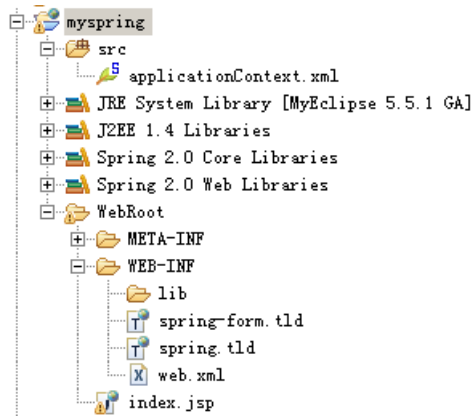

图 2-7 具有 spring 能力的 myspring 工程

到此一个具有 spring 能力的 web 工程就创建好了，从下一节中将讲述如何建立视图层

2.6.2 创建 Spring 框架中的视图层

Spring 和 Struts 的一个显著区别在于 Spring 的视图层技术更丰富，它包括传统的 JSP 技术和基于标签库技术的 Jstl。以及 velocity,xslt,tiles,freemaker,jasperreports 等。

这里使用大家所熟悉的 JSP 做为本节登录例子的视图层。选中 WebRoot 目录右击选择 new|JSP 命令如图 2-7 所示：

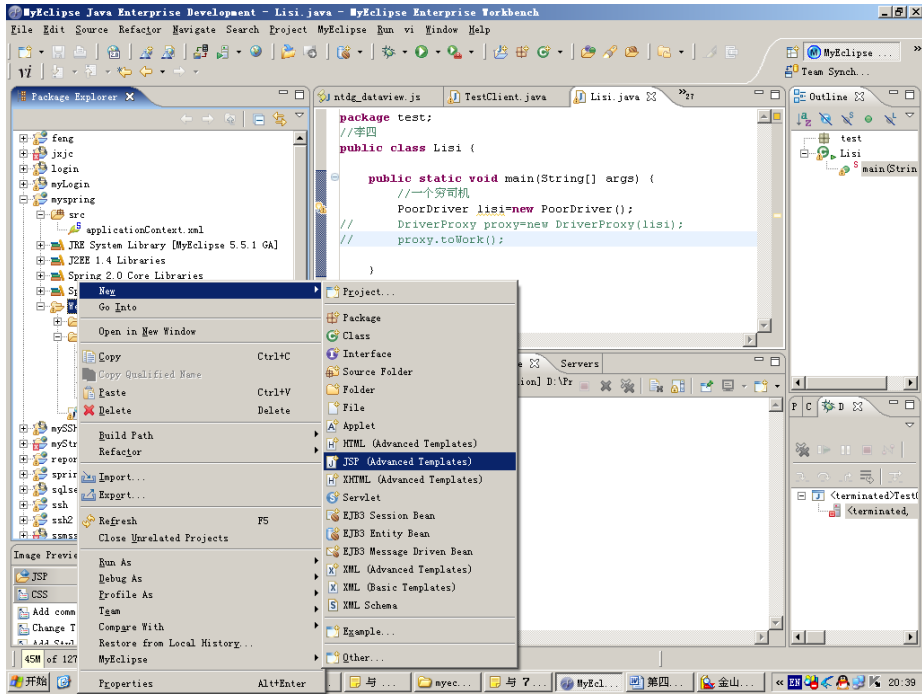

图 2-8 新建一个 JSP 页面

然后会出现一个新建向导对话框如图 2-8 所示：

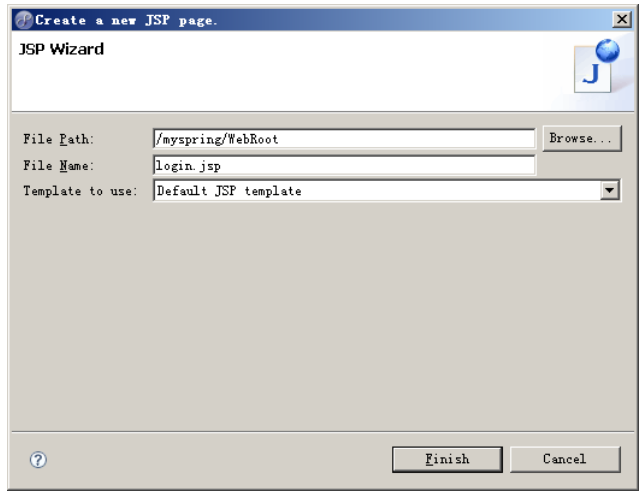

图 2-9 新建 JSP 页面向导

修改默认的文件名为 login.jsp。这是用户登录的首页。单击 Finish 按钮，用户首页的 JSP 页面创建完毕。用户首页的登录页面需要一个表单包含用户名和密码。完整的 login.jsp 内容如下所示：

```
<%@ page language="java" import="java.util.*"
pageEncoding="GBK"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+
request.getServerName()+":"+request.getServerPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>My JSP 'login.jsp' starting page</title>

<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta http-equiv="keywords"
content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="This is my page">
<!--
<link rel="stylesheet" type="text/css" href="styles.css">
-->

  </head>

  <body>

    <form action="login.do" method="post">
        用户名：<input type=text name="username"/><br>
        密码：<input type=text  name="password"/><br>
       <input type=submit value="登录" />
```

```
    </form>



  </body>
</html>
```

这里唯一需要解释的是 form 的 action 属性，这一点非常类似于 struts。Spring 的控制器也能通过截取*.do 的 URL 进行请求响应的。下面一节将讲述如何创建 login.do 所对应的控制层。

当用户登录失败后需要跳转到一个失败页面 error.jsp，这个失败页面输出失败原因，代码如下：

```
<%@ page language="java" import="java.util.*"
pageEncoding="GB18030"%>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+
request.getServerPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>My JSP 'error.jsp' starting page</title>

<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta http-equiv="keywords"
content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="This is my page">
<!--
<link rel="stylesheet" type="text/css" href="styles.css">
-->

  </head>

  <body>
 十分对不起,登录失败，由于:<%=request.getAttribute("msg")%>
  </body>
</html>
```

关键代码是加粗部分，当这个页面被请求时，会从请求对象 request 中取得 msg 的属性值，这个 msg 值必须是请求者已定义好的一个值，这会在控制层中看到。

用户登录成功后，也会跳转到一个成功页面并提示成功信息，这个页面命名为 success.jsp 代码内容如下：

```
<%@ page language="java" import="java.util.*"
```

```
pageEncoding="GB18030"%>
<%
String path = request.getContextPath();
String                          basePath                          =
request.getScheme()+"://"+request.getServerName()+":"
+request.getServerPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>My JSP 'success.jsp' starting page</title>

<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta                                        http-equiv="keywords"
content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="This is my page">
<!--
<link rel="stylesheet" type="text/css" href="styles.css">
-->

  </head>

  <body>
  恭喜你：<%=request.getAttribute("username") %>，登录成功
  </body>
</html>
```

username 会在控制层中定义。下一节将讲述用户登录的控制层。

2.6.3 创建 Spring 框架中的控制层

在 myspring 的 src 目录下新建一个包取名为 controller，然后在这个包下新建一个类，名称为 LoginController,这个类可以通过实现 Spring web Libararies 里面所提供的 Controller 接口成为一个 Spring 控制器，这个接口仅包含一个方法

- public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception 这个方法用来处理请求，非常类似于普通的 servlet 中的 doGet 和 doPost 方法，不同的是返回了一个 ModelAndView 对象，ModelAndView 有点类似 Struts 中的 ActionMapping 。通过它可以进行页面跳转，并将信息封装在了 request 对象中一并传递。LoginController 代码清单如下所示：

```
package controller;

import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import model.UserInfoBean;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

/**
 * 用户登录控制器
 *
 * @author Administrator
 *
 */
public class LoginController implements Controller {
// 成功页面，它的值在 applicationContext.xml 中通过配置被注入
private String successPage;

// 失败页面 ，它的值在 applicationContext.xml 中通过配置被注入
private String errorPage;

// 从 Controller 接口实现的方法
public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
    // 取得用户名和密码
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    // 消息
    String message = null;


    // 简单的空值验证
    if (username == null || password == null ||
username.trim().equals("")
            || password.trim().equals("")) {
        // 本质上是通过 request.setAttribute 将 msg 消息封装在 request
对象中
        message = "用户名或者密码为空";
        Map<String, String> model = new HashMap<String, String>();
        model.put("msg", message);
        // 跳转到错误页面
        return new ModelAndView(getErrorPage(), model);
    }



    // UserInfoBean 是模型层，它模拟了数据库，exisitUser 方法判断用户
名是否存在
    if (!UserInfoBean.exisitUser(username)) {
        // 用户名不存在
        message = username + "不存在！";
        Map<String, String> model = new HashMap<String, String>();
        model.put("msg", message);
        // 跳转到错误页面
        return new ModelAndView(getErrorPage(), model);
```

```
        }



        // UserInfoBean 是模型层，它模拟了数据库，confirmPassword 方法判
        断//密码是否正确，
        //前提是用户名已经存在
        if (!UserInfoBean.confirmPassword(username, password)) {
            // 密码不正确
            message = username + "的密码不正确";
            Map<String, String> model = new HashMap<String, String>();
            model.put("msg", message);
            // 跳转到错误页面
            return new ModelAndView(getErrorPage(), model);
        }



        else {
            // 用户名存在，且密码正确
            Map<String, String> model = new HashMap<String, String>();
            model.put("username", username);
            // 跳转到成功页面
            return new ModelAndView(getSuccessPage(), model);

        }



    }

    // -------------getter 和 setter 方法区----------------
    public String getErrorPage() {
        return errorPage;
    }

    public void setErrorPage(String errorPage) {
        this.errorPage = errorPage;
    }

    public String getSuccessPage() {
        return successPage;
    }

    public void setSuccessPage(String successPage) {
        this.successPage = successPage;
    }

    }
```

下面对上面的这段代码进行简单评论，这个类首先从 request 对象中获得用户名和密码信息，对于这一点，如果视图层的表单中需要提交的字段比较多的话，这种方法

就有点麻烦了，解决办法是使用 Spring 提供的专门的表单处理控制器 SimpleFormController

具体可以查看专门的 Spring 书籍。还需要解释一下的是 ModelAndView，它只是一个存储器，用 Map 做为容器，将信息存储在里面供高层调用者读取，这个使用者一般为一个后台 Servlet，这个 Servlet 负责将里面的内容读取出来并封装在 request 对象中，最后跳转到指定的页面。下一节将介绍做为模型层的 UserInfoBean。

2.6.4 创建 Spring 框架中的模型层

Struts 事实上是没有实现模型层的，一般使用 JavaBean 等技术做为模型层。Spring 框架在模型层上也没有做出过多的工作，这是有原因的，一般认为模型层灵活比较大，很难为其设计一个通用的框架。在上面这个例子中使用的模型层是一个普通的类。代码如下：

```
package model;

import java.util.HashMap;
import java.util.Map;
//用户信息模型层，由于没有使用到数据库，所以使用一个类进行模拟数据
public class UserInfoBean {
//存储用户信息容器，key 为用户名 value 为密码
     private   static     Map<String   ,String>   userinfo=new
HashMap<String,String>();

     //初始化数据
     static{
     String numberOneUser="zhangsan";
     String numberOnePassword="123";
     String numberTwoUser="lisi";
     String numberTwoPassword="456";
     userinfo.put(numberOneUser, numberOnePassword);
     userinfo.put(numberTwoUser, numberTwoPassword);
     }

     //判断一个用户名是否存在
    public static boolean  exisitUser(String username){
        return userinfo.containsKey(username);
    }

    //判断一个已经存在的用户名的密码时候正确
    public static boolean confirmPassword(String username,String
password){
    return userinfo.get(username).equals(password);
    }


}
```

代码比较简单，它模拟的实现一个数据库，手动的办法输入了一些测试数据，这个用户 Bean 中仅包含 zhansan 和 lisi 两个用户，仅包含判断用户名时候存在，密码时候正确两个方法。在实际开发中，它应该被与数据库连接相关的代码所替代。

2.6.5 编写配置文件

对于上面这个示例而言，最后一件要做的一件事情是编写配置文件，有两个配置文件需要编写，一个是 web.xml 一个是 applicationContext.xml。前者是大家熟悉的一个 web project 必须的配置文件，后者是 Spring 所需要的配置文件，这个名称可以更改，在 myspring 这个工程中，它默认为 applicationContext 且存放在 src 目录下。下面是 web.xml 文件的内容：

```
<?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_2.xsd">


 <!-- 配置 Spring 的后台 servlet -->
  <servlet>
  <servlet-name>dispatcherServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServl
et
  </servlet-class>
  <!-- 指定 Spring 配置文件的路径 -->
  <init-param>
  <param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/classes/applicationContext.xml</param-va
lue>
  </init-param>
  <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- 拦截所有以.do 结尾的请求，可以修改 -->
  <servlet-mapping>
   <servlet-name>dispatcherServlet</servlet-name>
   <url-pattern>*.do</url-pattern>
  </servlet-mapping>



</web-app>
```

这个 web.xml 文件中没有与 Spring 无关的其它配置，dispatcherServlet 是一个监听器，这个监听器里面包含了 Spring 的容器，它负责拦截所有的请求并管理 Spring 中的 Bean.它在应用服务器启动时运行，其角色类似于 Struts 中的 ActionServlet。下面是 applicationContext.xml 文件的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
```

```
    ">

    <!-- 映射处理器 -->
    <bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <!-- login.do 由 id 为 login 的 bean 处理 -->
                <prop key="login.do">login</prop>
            </props>
        </property>
    </bean>

    <!-- 配置控制器 -->
    <bean id="login" class="controller.LoginController">
        <!-- 注入错误页面属性 -->
        <property name="errorPage">
            <value>error.jsp</value>
        </property>
        <!-- 注入成功页面属性 -->
        <property name="successPage">
            <value>success.jsp</value>
        </property>
    </bean>
    </beans>
```

配置文件中主要包含两部分，一部分是配置映射关系，这是为什么在 login.jsp 中的表单的 action 属性使用 login.do 就可以找到对应的控制器的原因。另一部分是配置控制器的配置，这个控制器有两个属性，它们分别代表失败页面的绝对路径和成功页面的绝对路径 这里的 success.jsp 和 error.jsp 都是在 webRoot 根目录下。

2.6.6 运行程序

将这个 myspring 部署到应用服务器上，这里使用 tomcat。在地址栏上敲入地址：http://localhost:9090/myspring/login.jsp，可能你的端口是 8080，将出现如图 2-9 所示的登录界面：

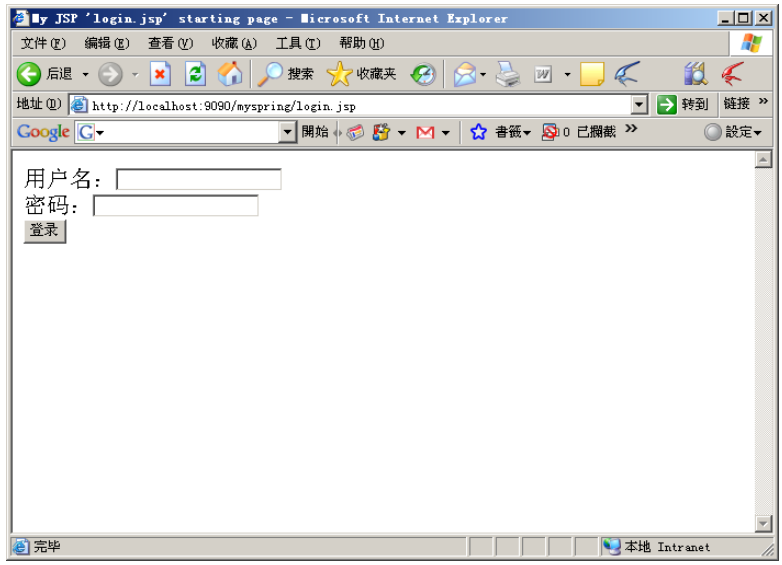

图 2-9 myspring 用户登录界面

第一次什么都不输入，单击“登录”按钮，页面跳转到 error.jsp，但在地址上是看不到 error.jsp 的。如图 2-10 所示：

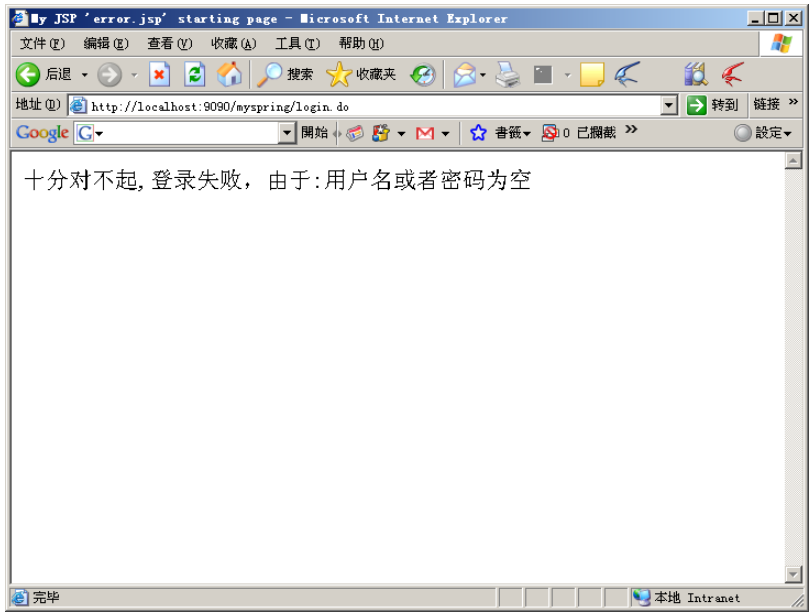

图 2-10 用户或者密码为空的失败页面

再后退到登录页面，输入一个正确的用户名和随便输入一个错误密码。如图 2-11 所示：

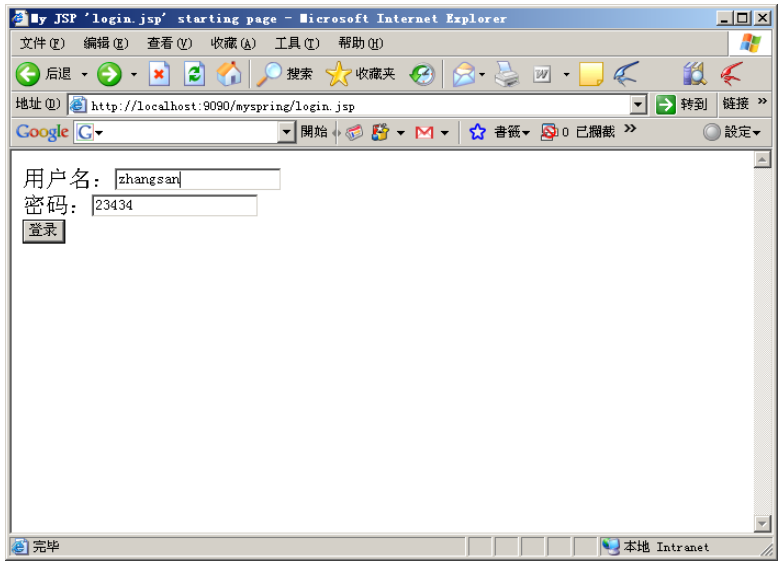

图 2-11 输入一个正确的用户名和错误的密码

为了说明问题，这里的密码输入框是可见的，在实际开发中不要这样做。单击“登录”按钮将出现如图 2-12 的错误提示信息：

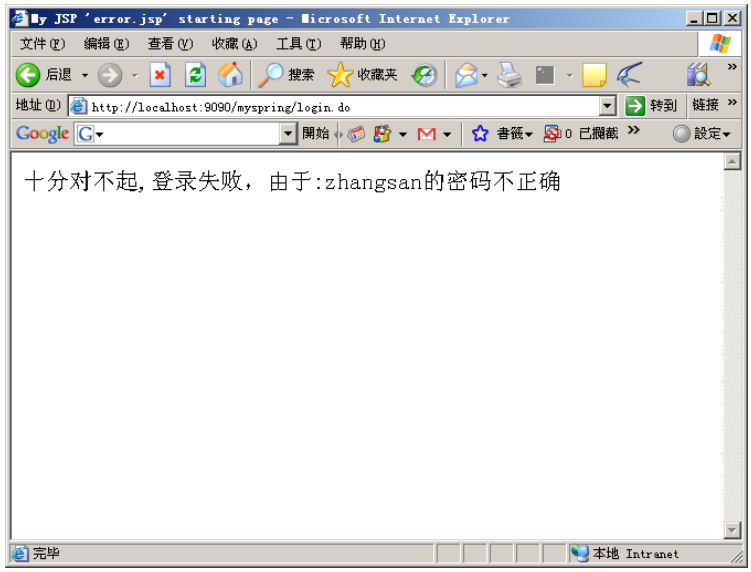

图 2-12 密码不正确的提示页面

最后再回到用户登录页面，输入正确的用户名和密码，单击“登录”按钮后将得到如图 2-13 所示的成功页面：

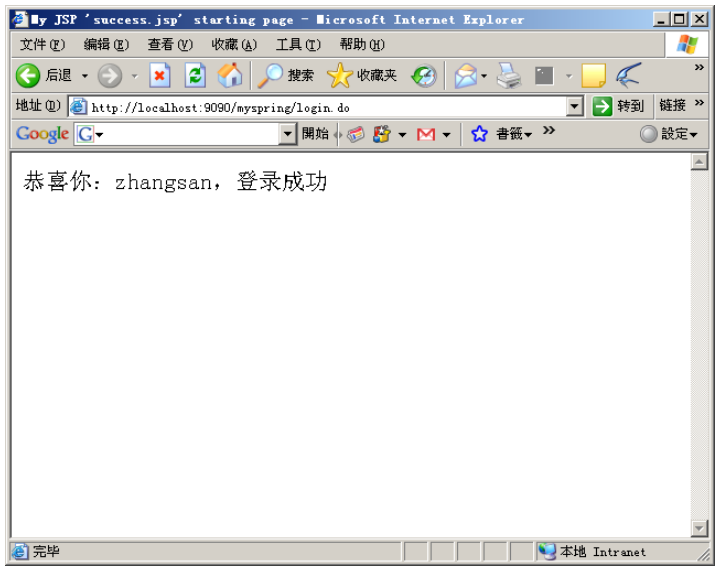

图 2-13 登录成功