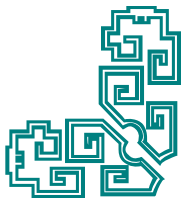
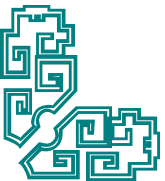




# 第5章 Spring应用

---

- ♥5.1 Spring概述
- ♥5.2 Spring核心机制——依赖注入
- ♥5.3 Spring核心接口及基本配置
- ♥5.4 Spring AOP
- ♥5.5 Spring事务支持
- ♥5.6 Spring与Struts 2整合应用
- ♥5.7 Spring与Hibernate整合应用



# 5.1 Spring概述

Spring框架的主要优势之一是其分层架构，分层架构允许选择使用任何一个组件，同时为 **Java EE** 应用程序开发提供集成的框架。Spring 框架的分层架构，由 7 个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 **Bean** 的方式，如图5.1所示。

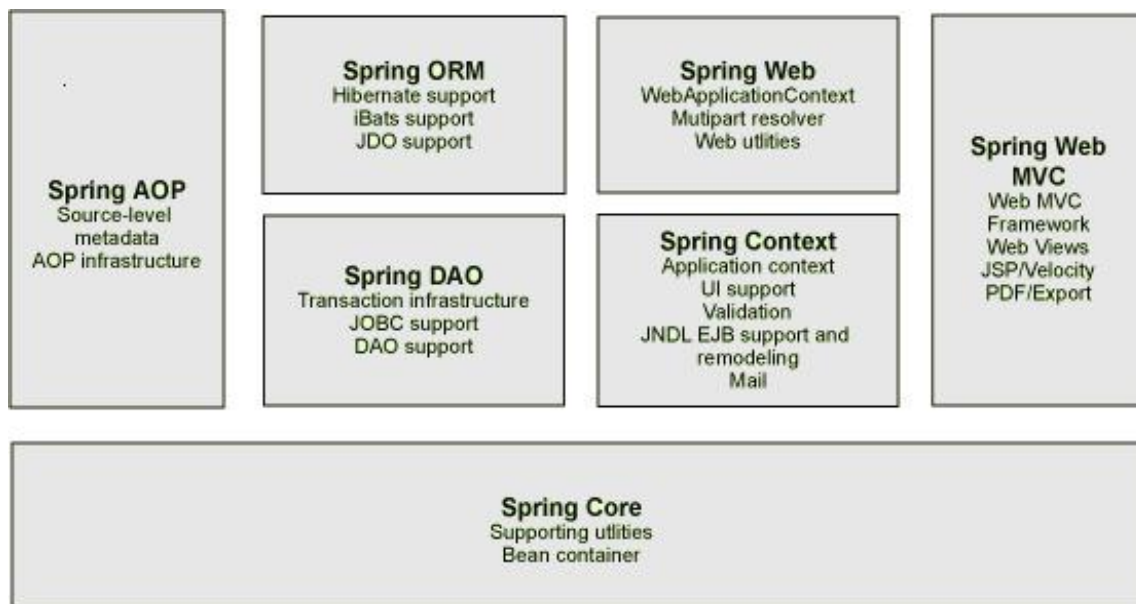
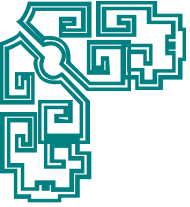


图5.1 Spring框架的组件结构图

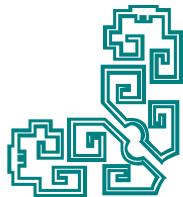
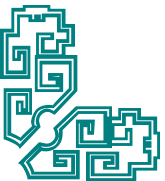


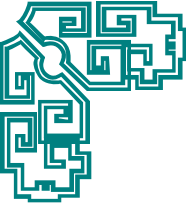
# 5.1 Spring概述

---

组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。各模块的功能如下：

- ① 核心容器。提供Spring框架的基本功能，其主要组件是BeanFactory，是工厂模式的实现。
- ② Spring 上下文。向Spring 框架提供上下文信息，包括企业服务，如 JNDI、EJB、电子邮件、国际化、校验和调度等。
- ③ Spring AOP（**日志**）。通过配置管理特性，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块直接将面向方面编程的功能集成到 Spring 框架中。它为基于Spring 应用程序的对象提供了事务管理服务。
- ④ Spring DAO。JDBC DAO 抽象层提供了有用的异常层次结构，用来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（如打开和关闭连接）。





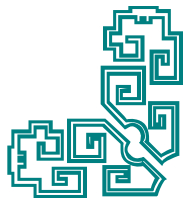
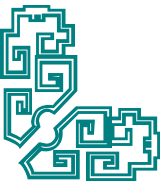
# 5.1 Spring概述

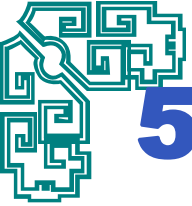
---

⑤ **Spring ORM**。Spring 框架插入了若干ORM框架，提供ORM的对象关系工具，其中包括JDO、Hibernate和iBatis SQL Map，并且都遵从Spring 的通用事务和 DAO 异常层次结构。

⑥ **Spring Web** 模块。为基于 Web 的应用程序提供上下文。它建立在应用程序上下文模块之上，简化了处理多份请求及将请求参数绑定到域对象的工作。Spring 框架支持与 Jakarta Struts 的集成。

⑦ **Spring MVC** 框架，是一个全功能构建Web应用程序的 MVC 实现。通过策略接口实现高度可配置，MVC 容纳了大量视图技术，其中包括JSP、Velocity、Tiles、iText和POI。





## 5.2 Spring核心机制——依赖注入

---

### ✧ 5.2.1 工厂模式（创建对象）

下面举例说明工厂模式的应用。

创建一个Java Project，命名为“FactoryExample”。在src文件夹下建立包face，在该包下建立接口Human，代码如下：

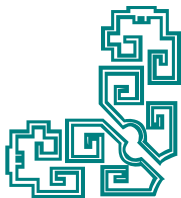
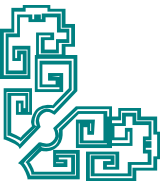
```
package face;  
public interface Human {  
    void eat();  
    void walk();  
}
```

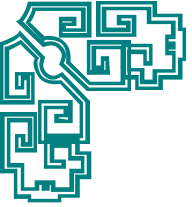
在src文件夹下建立包iface，在该包下建立Chinese类和American类，分别实现Human接口。

Chinese.java代码如下：

```
package iface;  
import face.Human;  
public class Chinese implements Human{  
    public void eat() {  
        System.out.println("中国人很会吃！");  
    }  
    public void walk() {  
        System.out.println("中国人健步如飞！");  
    }  
}
```

---



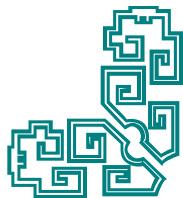
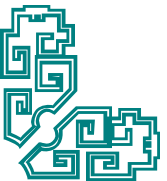


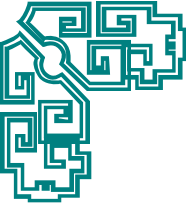
## 5.2.1 工厂模式

---

- American.java代码如下：

```
package iface;
import face.Human;
public class American implements Human{
    public void eat() {
        System.out.println("美国人吃西餐！");
    }
    public void walk() {
        System.out.println("美国人经常坐车！");
    }
}
```



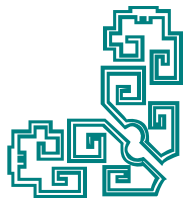
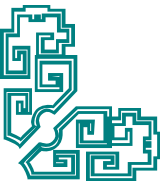


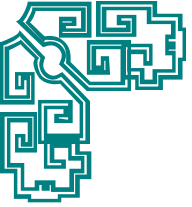
## 5.2.1 工厂模式

---

在src文件夹下建包**factory**，在该包内建立工厂类**Factory**，代码如下：

```
package factory;
import iface.American;
import iface.Chinese;
import face.Human; //(human h =new Chinese())
public class Factory {
    public Human getHuman(String name){
        if(name.equals("Chinese")){
            return new Chinese();
        }else if(name.equals("American")){
            return new American();
        }else{
            throw new IllegalArgumentException("参数不正确");
        }
    }
}
} //用来创建对象
```





## 5.2.1 工厂模式

---

在src文件夹下建包test，在该包内建立测试类Test，代码如下：

package test;好处：模块化管理

import face.Human;

import factory.Factory;

public class Test {

    public static void main(String[] args) {

        Human human=null;

        human=new Factory().getHuman("Chinese");

        human.eat();

        human.walk();

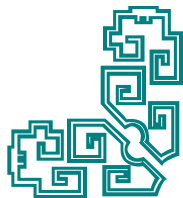
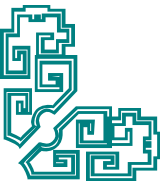
        human=new Factory().getHuman("American");

        human.eat();

        human.walk();

    }

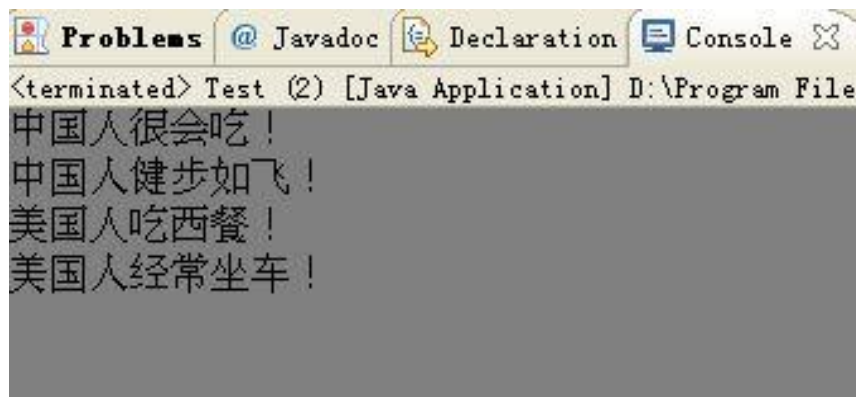
}





## 5.2.1 工厂模式

该程序为Java应用程序，直接运行可看出结果，如图5.2所示。



The screenshot shows a Java IDE window with a console tab selected. The console output displays the following text:

```
<terminated> Test (2) [Java Application] D:\Program File  
中国人很会吃！  
中国人健步如飞！  
美国人吃西餐！  
美国人经常坐车！
```

图5.2 工厂模式运行结果

## 5.2.2 依赖注入应用

### ◆ 1. 为项目添加Spring开发能力

右击项目名，选择【MyEclipse】→【Add Spring Capabilities...】菜单项，将出现如图5.3所示的对话框，选中要应用的Spring的版本及所需的类库文件。

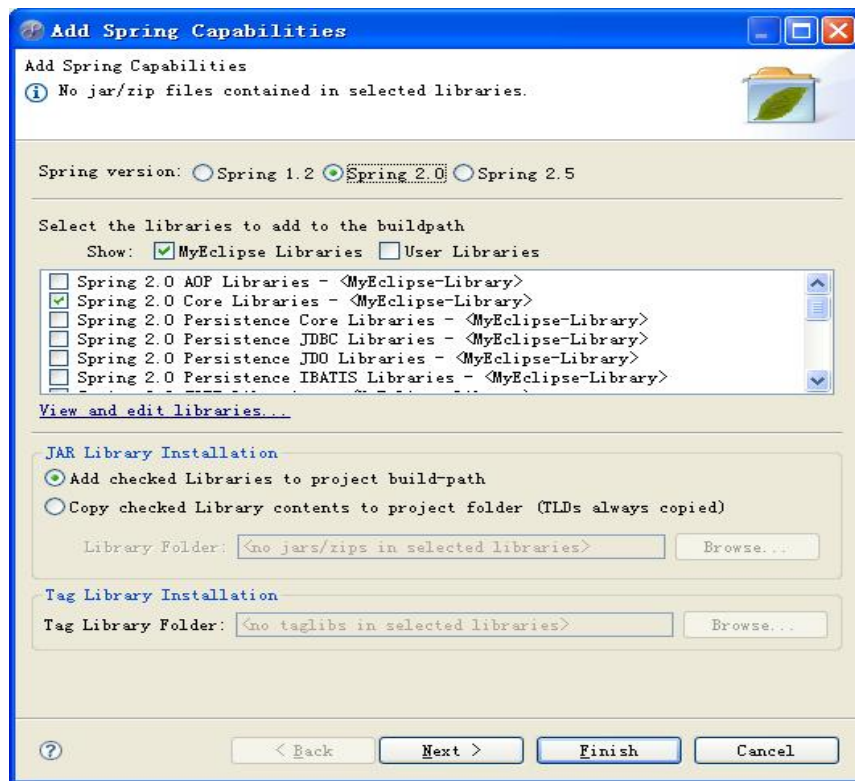


图5.3 选择Spring版本及类库对话框

## 5.2.2 依赖注入应用

选择结束后，单击【Next】按钮，出现如图5.4所示的界面。用于创建Spring的配置文件。

单击【Finish】按钮完成。项目的src文件夹下会出现名为applicationContext.xml的文件，这就是Spring的核心配置文件。

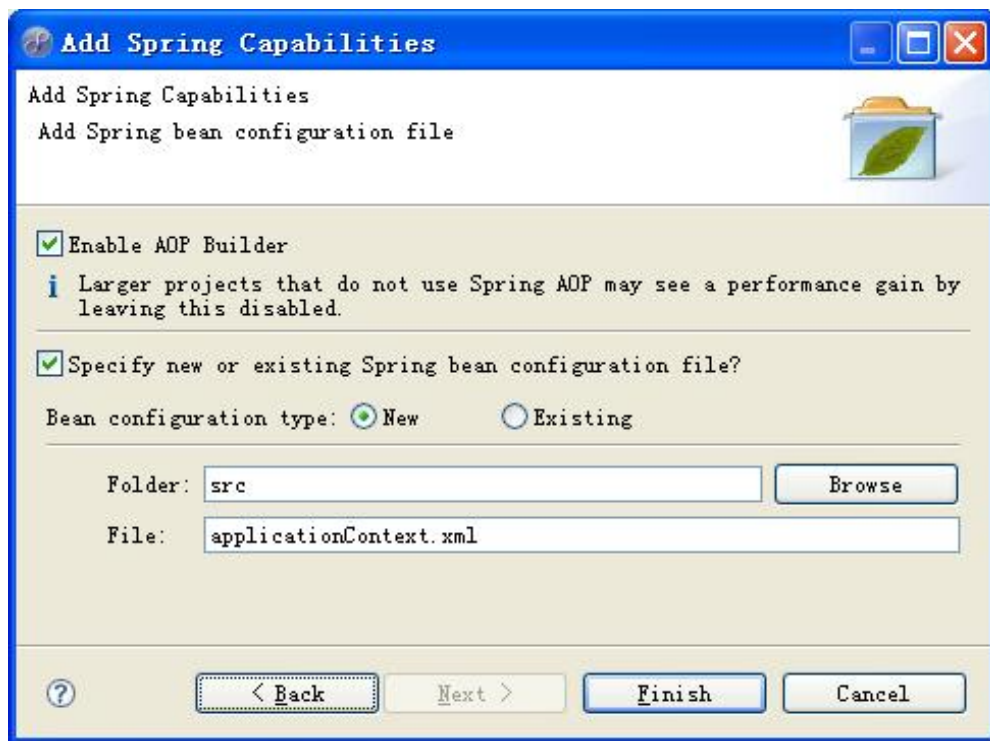
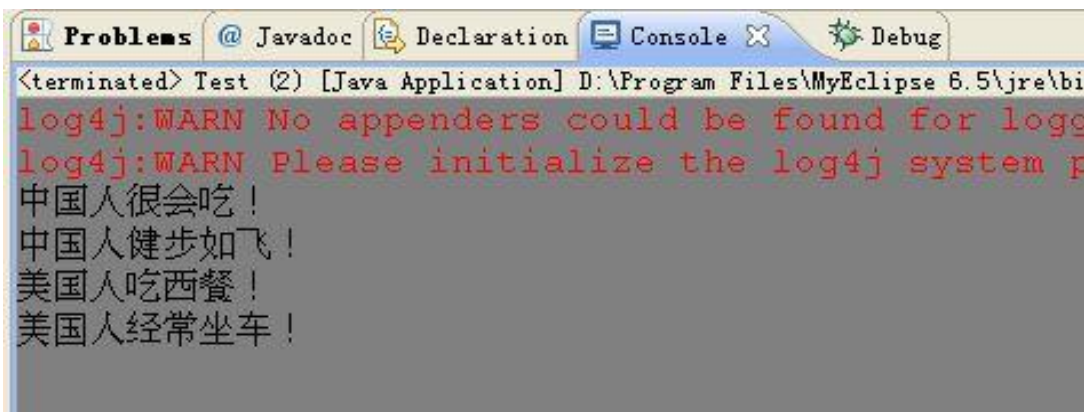


图5.4 创建Spring的配置文件

## 5.2.2 依赖注入应用

- ◇ 3. 修改测试类  
配置完成后，就可以修改Test类，[代码](#)。
- ◇ 4. 运行  
运行该测试类，结果如图5.5所示。



```
<terminated> Test (2) [Java Application] D:\Program Files\MyEclipse 6.5\jre\bin\java.exe
log4j:WARN No appenders could be found for logger org.apache.log4j.Logger.
log4j:WARN Please initialize the log4j system properly.
中国人很会吃！
中国人健步如飞！
美国人吃西餐！
美国人经常坐车！
```

图5.5 运行结果



## 5.2.2 依赖注入应用

---

对象ctx就相当于原来的Factory工厂，原来的Factory可以删除掉了。再回头看原来的applicationContext.xml文件配置：

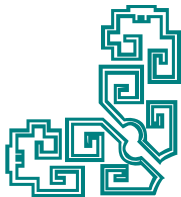
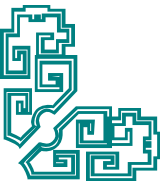
```
<bean id="chinese" class="iface.Chinese"></bean>
```

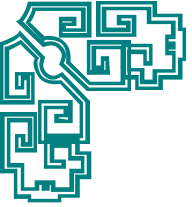
```
<bean id="american" class="iface.American"></bean>
```

id是ctx.getBean的参数值，一个字符串。class是一个类（包名+类名）。然后在Test类里获得Chinese对象及American对象：

```
human = (Human) ctx.getBean("american");
```

```
human = (Human) ctx.getBean("american");
```





## 5.2.3 注入的两种方式

---

### ◆ 1. 设置注入

设置注入是通过setter方法注入被调用者的实例。这种方法简单、直观，很容易理解，因而Spring的依赖注入被大量使用，下面举例说明。

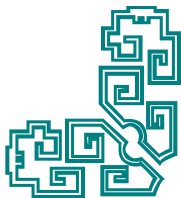
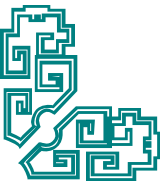
创建一个Java Project，命名为“FactoryExample1”。在项目的src文件夹下建立下面的源文件。

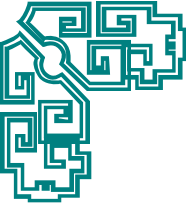
➤ Human的接口，Human.java，代码如下：

```
public interface Human {  
    void speak();  
}
```

➤ Language接口，Language.java，代码如下：

```
public interface Language {  
    public String kind();  
}
```





## 5.2.3 注入的两种方式

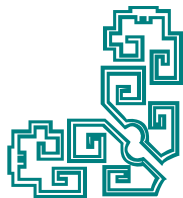
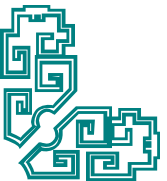
---

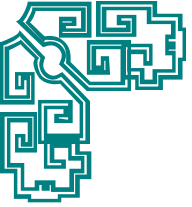
- 下面是Human实现类Chinese.java代码：

```
public class Chinese implements Human{  
    private Language lan;  
    public void speak() {  
        System.out.println(lan.kind());  
    }  
    public void setLan(Language lan) {  
        this.lan = lan;  
    }  
}
```

- 下面是Language实现类English.java代码：

```
public class English implements Language{  
    public String kind() {  
        return "中国人也会说英语！";  
    }  
}
```





## 5.2.3 注入的两种方式

---

- 下面通过Spring的配置文件来完成其对象的注入。代码如下：

```
<?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">
```

```
  <!-- 定义第一个Bean，注入Chinese类对象 -->
```

```
  <bean id="chinese" class="Chinese">
```

```
    <!-- property元素用来指定需要容器注入的属性，lan属性需要容器注入  
         ref就指向lan注入的id -->
```

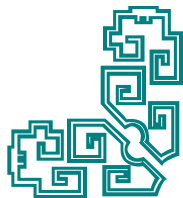
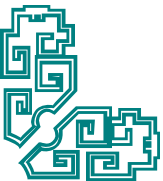
```
    <property name="lan" ref="english"></property>
```

```
  </bean>
```

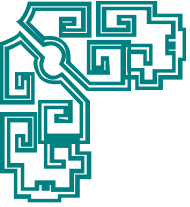
```
  <!-- 注入English -->
```

```
  <bean id="english" class="English"></bean>
```

```
</beans>
```







## 5.2.3 注入的两种方式

➤ 测试代码如下：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Test {
    public static void main(String[] args) {
        ApplicationContext ctx = new
        FileSystemXmlApplicationContext("src/applicationContext.xml");
        Human human = null;
        human = (Human) ctx.getBean("chinese");
        human.speak();
    }
}
```

程序执行结果如图5.6所示。

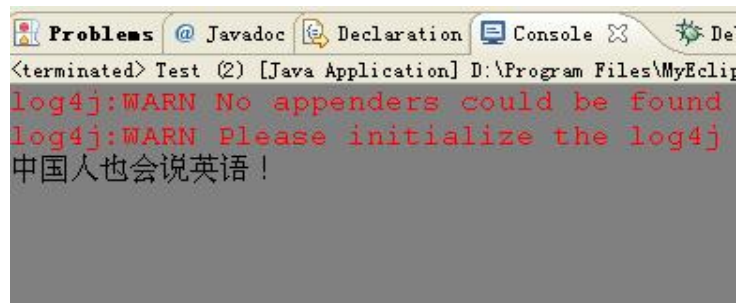
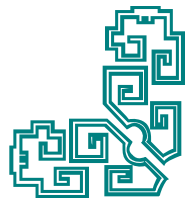
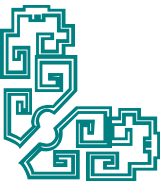
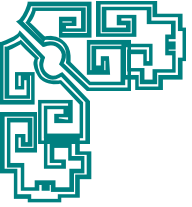


图5.6 程序运行结果





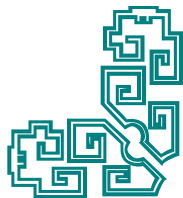
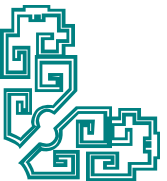
## 5.2.3 注入的两种方式

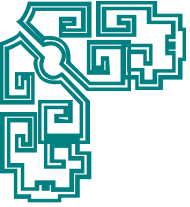
---

### ◇ 2. 构造注入

例如，只要对前面的Chinese类进行简单的修改：

```
public class Chinese implements Human{  
    private Language lan;  
    public Chinese(){};  
    // 构造注入所需要的带参数的构造函数  
    public Chinese(Language lan){  
        this.lan=lan;  
    }  
    public void speak() {  
        System.out.println(lan.kind());  
    }  
}
```





## 5.2.3 注入的两种方式

---

- 配置文件也需要做简单的修改:

```
<?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">
```

```
    <!-- 定义第一个Bean，注入Chinese类对象 -->
```

```
    <bean id="chinese" class="Chinese">
```

```
        <!-- 使用构造注入，为Chinese实例注入Language实例 -->
```

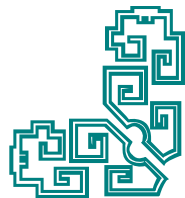
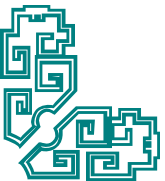
```
            <constructor-arg ref="english"></constructor-arg>
```

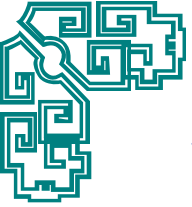
```
    </bean>
```

```
    <!-- 注入English -->
```

```
    <bean id="english" class="English"></bean>
```

```
</beans>
```





## 5.3 Spring核心接口及基本配置

---

### ✧ 5.3.1 Spring核心接口

#### ◆ 1. BeanFactory

在Spring中有几种BeanFactory的实现，其中最常使用的是org.springframework.bean.factory.xml.XmlBeanFactory。它根据XML文件中的定义装载Bean。

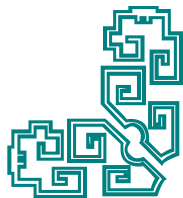
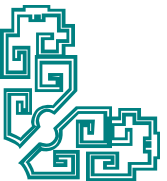
要创建XmlBeanFactory，需要传递一个java.io.InputStream对象给构造函数。InputStream对象提供XML文件给工厂。例如，下面的代码片段使用一个java.io.FileInputStream对象把Bean XML定义文件给XmlBeanFactory：

```
BeanFactory factory = new XmlBeanFactory(new FileInputStream("applicationContext.xml"));
```

这行简单的代码告诉Bean Factory从XML文件中读取Bean的定义信息，但是现在Bean Factory没有实例化Bean，Bean被延迟载入到Bean Factory中，就是说Bean Factory会立即把Bean定义信息载入进来，但是Bean只有在需要的时候才被实例化。

为了从BeanFactory得到Bean，只要简单地调用getBean()方法，把需要的Bean的名字当做参数传递进去就行了。由于得到的是Object类型，所以要进行强制类型转化。

```
MyBean myBean = (MyBean)factory.getBean("myBean");
```





## 5.3.1 Spring核心接口

---

### ◇ 2. ApplicationContext

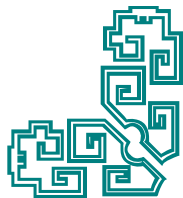
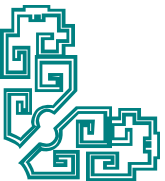
两者都是载入Bean定义信息，装配Bean，根据需要分发Bean。但是ApplicationContext提供了更多功能：

- ① 应用上下文提供了文本信息解析工具，包括对国际化的支持。
- ② 应用上下文提供了载入文本资源的通用方法，如载入图片。
- ③ 应用上下文可以向注册为监听器的Bean发送事件。

由于它提供的附加功能，几乎所有的应用系统都选择ApplicationContext，而不是BeanFactory。

在ApplicationContext的诸多实现中，有三个常用的实现：

- **ClassPathXmlApplicationContext**：从类路径中的XML文件载入上下文定义信息，把上下文定义文件当成类路径资源。
- **FileSystemXmlApplicationContext**：从文件系统（文件系统）中的XML文件载入上下文定义信息。
- **XmlWebApplicationContext**：从Web系统中的XML文件载入上下文定义信息。



## 5.3.1 Spring核心接口

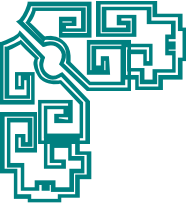
---

例如：

```
ApplicationContext context=new FileSystemXmlApplicationContext ("c:/foo.xml");  
ApplicationContext context=new ClassPathApplicationContext ("foo.xml");  
ApplicationContext context=
```

```
WebApplicationContextUtils.getWebApplicationContext  
(request.getSession().getServletContext ());
```

FileSystemXmlApplicationContext和ClassPathXmlApplicationContext的区别是：FileSystemXmlApplicationContext只能在指定的路径中寻找foo.xml文件，而ClassPathXmlApplicationContext可以在整个类路径中寻找foo.xml。



## 5.3.2 Spring基本配置

---

### ◇ 1. 使用XML装配

理论上，Bean装配可以从任何配置资源获得。但实际上，XML是最常见的Spring应用系统配置源。

如下的XML文件展示了一个简单的Spring上下文定义文件：

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

```
...
```

```
<beans ...>
```

```
// 根元素
```

```
    <bean id="foo" class="com.spring.Foo"/>
```

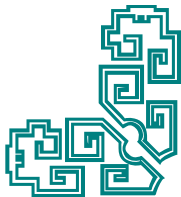
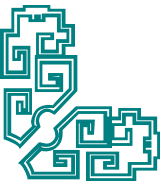
```
// Bean实例
```

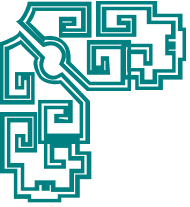
```
    <bean id="bar" class="com.spring.Bar"/>
```

```
// Bean实例
```

```
</beans>
```

在XML文件定义Bean，上下文定义文件的根元素<beans>。<beans>有多个<bean>子元素。每个<bean>元素定义了一个Bean（任何一个Java对象）如何被装配到Spring容器中。





## 5.3.2 Spring基本配置

---

### ◇ 2. 添加一个Bean

向Spring容器中添加一个Bean只需要向XML文件中添加一个<bean>元素。如下面的语句：

```
<bean id="foo" class="com.spring.Foo"/>
```

当通过Spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以为Bean指定特定的作用域。

① **原型模式与单实例模式**：Spring中的Bean默认情况下是单实例模式。在容器分配Bean的时候，它总是返回同一个实例。

<bean>的singleton属性告诉ApplicationContext这个Bean是不是单实例Bean，默认是true，但是把它设置为false的话，就把这个Bean定义成了原型Bean。

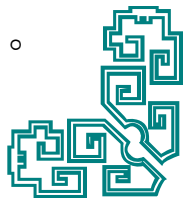
```
<bean id="foo" class="com.spring.Foo" singleton="false"/>
```

//原型模式Bean

② request或session：对于每次HTTP请求或HttpSession，使用request或session定义的Bean都将产生一个新实例，即每次HTTP请求或HttpSession将会产生不同的Bean实例。

③ global session：每个全局的HttpSession对应一个Bean实例。典型情况下，仅在使用portlet context的时候有效。

当一个Bean实例化的时候，有时需要做一些初始化的工作，然后才能使用。因此，Spring可以在创建和拆卸Bean的时候调用Bean的两个生命周期方法。







## 5.3.2 Spring基本配置

---

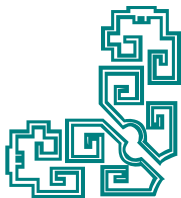
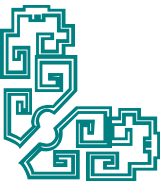
在Bean的定义中设置自己的init-method，这个方法在Bean被实例化时马上被调用。同样，也可以设置自己的destroy-method，这个方法在Bean从容器中删除之前调用。

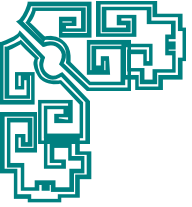
一个典型的例子是连接池Bean。

```
public class MyConnectionPool{  
    ...  
    public void initialize(){//initialize connection pool}  
    public void close(){ //release connection}  
    ...  
}
```

➤ Bean的定义如下：

```
<bean id="connectionPool" class="com.spring.MyConnectionPool"  
    init-method="initialize"           //当Bean被载入容器时调用initialize方法  
    destroy-method="close">          //当Bean从容器中删除时调用close方法  
</bean>
```





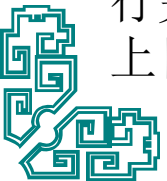
## 5.4 Spring AOP

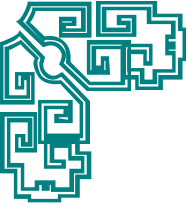
---

- ✧ **5.4.1 从代理机制初探AOP (aspect oriented programming)**  
来看一个简单的例子，当需要在执行某些方法时留下日志信息，可能会这样写：

```
import java.util.logging.*;
public class HelloSpeaker{
    private Logger logger=Logger.getLogger(this.getClass().getName());
    public void hello(String name){
        logger.log(Level.INFO, "hello method starts...");
        // 方法开始执行时留下日志
        System.out.println("hello, "+name);
        // 程序的主要功能
        Logger.log(Level.INFO, "hello method ends...");// 方法执行完毕时留下
        // 日志
    }
}
```

在HelloSpeaker类中，当执行hello()方法时，程序员希望该方法执行开始与执行完毕时都留下日志。最简单的做法是用上面的程序设计，在方法执行的前后加上日志动作。





## 5.4.1 从代理机制初探AOP

---

可以使用代理（**Proxy**）机制来解决这个问题，有两种代理方式：静态代理（**static proxy**）和动态代理（**dynamic proxy**）。

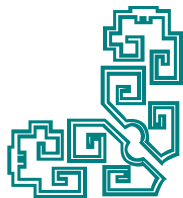
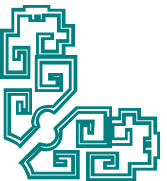
在静态代理的实现中，代理类与被代理的类必须实现同一个接口。在代理类中可以实现记录等相关服务，并在需要的时候再呼叫被代理类。这样被代理类就可以仅仅保留业务相关的职责了。

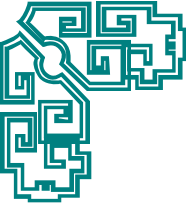
举个简单的例子，首先定义一个IHello接口，IHello.java代码如下：

```
public interface IHello{  
    public void hello(String name);  
}
```

然后让实现业务逻辑的HelloSpeaker类实现IHello接口，HelloSpeaker.java代码如下：

```
public class HelloSpeaker implements IHello{  
    public void hello(String name){  
        System.out.println("hello,"+name);  
    }  
}
```





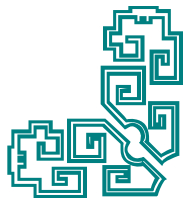
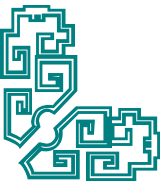
## 5.4.1 从代理机制初探AOP

---

可以看到，在HelloSpeaker类中没有任何日志的代码插入其中，日志服务的实现将被放到代理类中，代理类同样要实现IHello接口。

HelloProxy.java代码如下：

```
public class HelloProxy implements IHello{
    private Logger logger=Logger.getLogger(this.getClass().getName());
    private IHello helloObject;
    public HelloProxy(IHello helloObject){
        this.helloObject=helloObject;
    }
    public void hello(String name){
        log("hello method starts...");           // 日志服务
        helloObject.hello(name);                 // 执行业务逻辑
        log("hello method ends...");             // 日志服务
    }
    private void log(String ms){
        logger.log(Level.INFO,msg);
    }
}
```





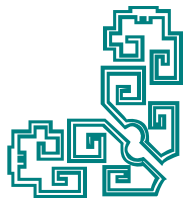
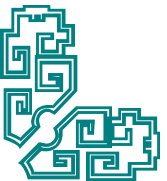
## 5.4.1 从代理机制初探AOP

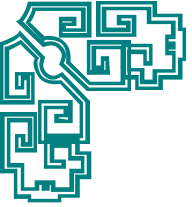
---

在HelloProxy类的hello()方法中，真正实现业务逻辑前后安排记录服务，可以实际撰写一个测试程序来看看如何使用代理类。

```
public class ProxyDemo{  
    public static void main(String[] args){  
        IHello proxy=new HelloProxy(new HelloSpeaker());  
        proxy.hello("Justin");  
    }  
}
```

➤ 程序运行结果：  
hello,Justin





## 5.4.2 动态代理

---

要实现动态代理，同样需要定义所要代理的接口。

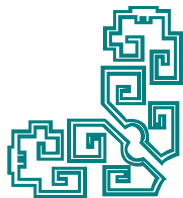
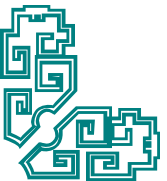
- IHello.java代码如下：

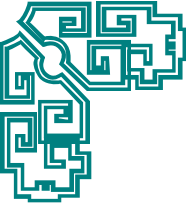
```
public interface IHello{  
    public void hello(String name);  
}
```

然后让实现业务逻辑的HelloSpeaker类实现IHello接口。

- HelloSpeaker.java代码如下：

```
public class HelloSpeaker implements IHello{  
    public void hello(String name){  
        System.out.println("Hello,"+name);  
    }  
}
```



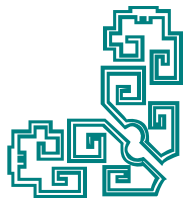
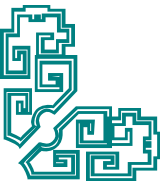


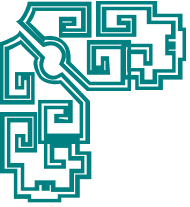
## 5.4.2 动态代理

---

- 与上例不同的是，这里要实现不同的代理类：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
public class LogHandler implements InvocationHandler{
    private Object sub;
    public LogHandler() {
    }
    public LogHandler(Object obj){
        sub = obj;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable{
        System.out.println("before you do thing");
        method.invoke(sub, args);
        System.out.println("after you do thing");
        return null;
    }
}
```





## 5.4.2 动态代理

---

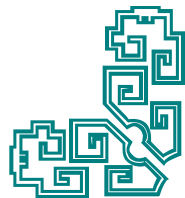
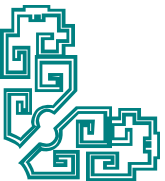
写一个测试程序，使用LogHandler来绑定被代理类。

- ProxyDemo.java代码如下：

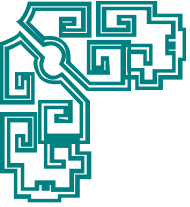
```
import java.lang.reflect.Proxy;
public class ProxyDemo {
    public static void main(String[] args) {
        HelloSpeaker helloSpeaker=new HelloSpeaker();
        LogHandler logHandler=new LogHandler(helloSpeaker);
        Class cls=helloSpeaker.getClass();
        IHello iHello=

        (IHello)Proxy.newProxyInstance(cls.getClassLoader(),cls.getInterfaces(),logHandler);
        iHello.hello("Justin");
    }
}
```

- 程序运行结果：  
before you do thing  
Hello, Justin  
after you do thing







## 5.4.3 AOP术语与概念

### ◇ 1. cross-cutting concerns

在DynamicProxyDemo例子中，记录的动作原先被横切（Cross-cutting）到HelloSpeaker本身所负责的业务流程中。类似于日志这类的动作，如安全检查、事务等服务，在一个应用程序中常被安排到各个类的处理流程之中。这些动作在AOP的术语中称为cross-cutting concerns。如图5.7所示，原来的业务流程是很单纯的。

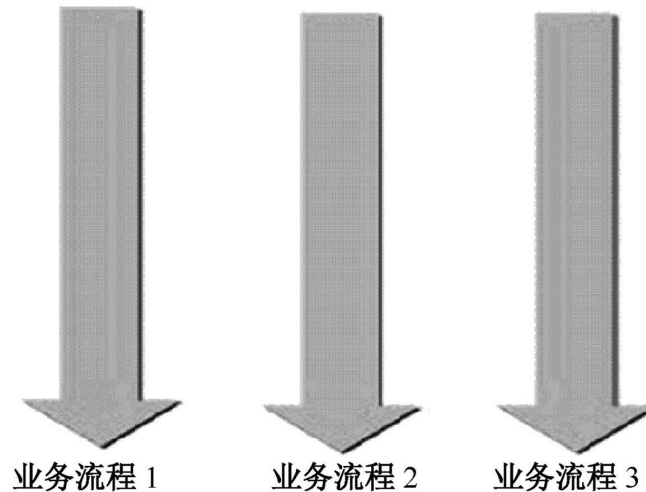
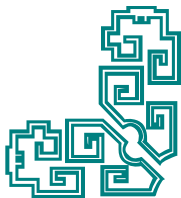
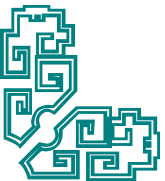


图5.7 原来的业务流程



## 5.4.3 AOP术语与概念

为了加入日志与安全检查等服务，类的程序代码中被硬生生地写入了相关的Logging、Security程序片段，如图5.8所示。

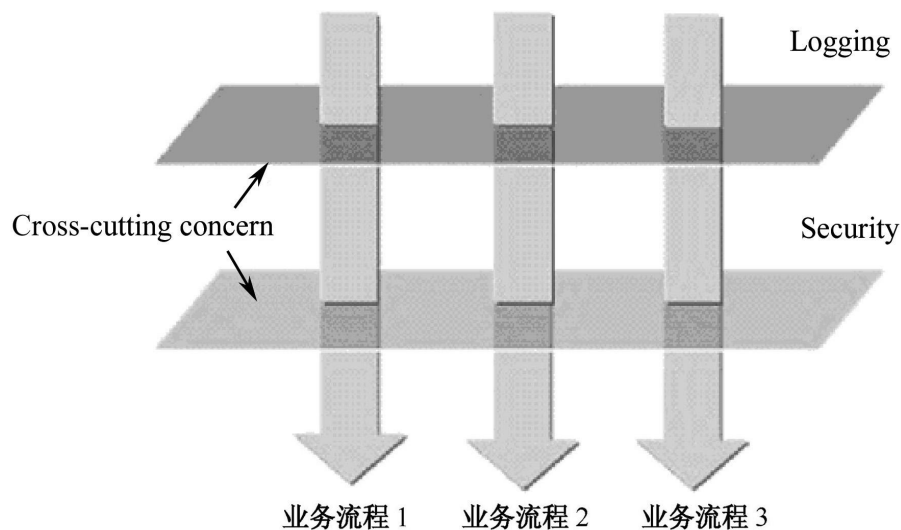
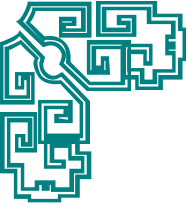


图5.8 加入各种服务的业务流程

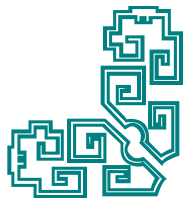
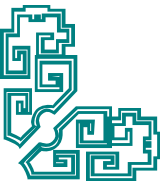


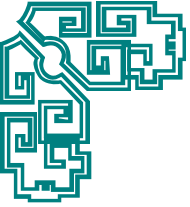
## 5.4.3 AOP术语与概念

---

### ◇ 2. Aspect

将散落在各个业务类中的cross-cutting concerns收集起来，设计各个独立可重用的类，这种类称为**Aspect**。例如，在动态代理中将日志的动作设计为**LogHandler**类，**LogHandler**类在AOP术语中就是**Aspect**的一个具体实例。在需要该服务的时候，缝合到应用程序中；不需要服务的时候，也可以马上从应用程序中脱离。应用程序中的可重用组件不用做任何修改。例如，在动态代理中的**HelloSpeaker**所代表的角色就是应用程序中可重用的组件，在它需要日志服务时并不用修改本身的程序代码。





## 5.4.4 通知Advice

---

- Spring提供了5种通知（Advice）类型：Interception Around、Before、After Returning、Throw 和Introduction。它们分别在以下情况被调用：
- **Interception Around Advice**：在目标对象的方法执行前后被调用。
- **Before Advice**：在目标对象的方法执行前被调用。
- **After Returning Advice**：在目标对象的方法执行后被调用。
- **Throw Advice**：在目标对象的方法抛出异常时被调用。
- **Introduction Advice**：一种特殊类型的拦截通知，只有在目标对象的方法调用完毕后执行。

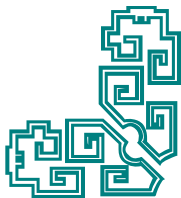
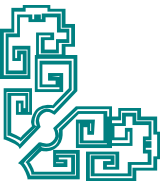
创建一个Before Advice的Web项目，步骤如下。

- ① 创建一个Web项目，命名为“Spring\_Advices”。
- ② 编写Java类。

**Before Advice**会在目标对象的方法执行之前被呼叫。这个接口提供了获取目标方法、参数及目标对象。

**MethodBeforeAdvice**接口的代码如下：

```
import java.lang.ref.*;
import java.lang.reflect.Method;
public interface MethodBeforeAdvice{
    void before(Method method, Object[] args, Object target) throws Exception;
}
```





## 5.4.4 通知Advice

---

用实例来示范如何使用Before Advice。首先要定义目标对象必须实现的接口IHello。

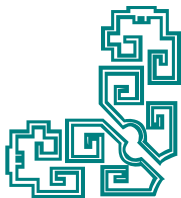
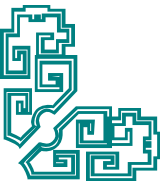
- IHello.java代码如下：

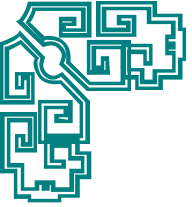
```
public interface IHello{  
    public void hello(String name);  
}
```

接着定义一个HelloSpeaker，实现IHello接口。

- HelloSpeaker.java代码如下：

```
public class HelloSpeaker implements IHello{  
    public void hello(String name){  
        System.out.println("Hello,"+name);  
    }  
}
```





## 5.4.4 通知Advice

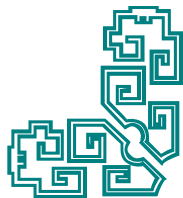
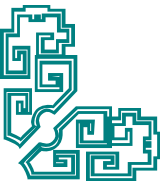
---

在对HelloSpeader不进行任何修改的情况下，想要在hello()方法执行之前可以记录一些信息。有一个组件，但没有源代码，可对它增加一些日志的服务。

➤ LogBeforeAdvice.java代码如下：

```
import java.lang.reflect.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class LogBeforeAdvice implements MethodBeforeAdvice{
    private Logger logger=Logger.getLogger(this.getClass().getName());
    public void before(Method method,Object[] args,Object target) throws
Exception{
        logger.log(Level.INFO, "method starts..." +method);
    }
}
```



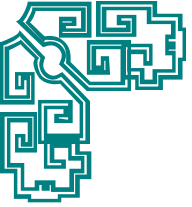
## 5.4.4 通知Advice

③ 添加Spring开发能力。

➤ applicationContext.xml的代码如下：

```
<?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">
  <bean id="logBeforeAdvice" class="LogBeforeAdvice" />
  <bean id="helloSpeaker" class="HelloSpeaker" />
  <bean id="helloProxy"
        class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <value>IHello</value>
    </property>
    <property name="target">
      <ref bean="helloSpeaker" />
    </property>
    <property name="interceptorNames">
      <list>
        <value>logBeforeAdvice</value>
      </list>
    </property>
  </bean>
</beans>
```



## 5.4.4 通知Advice

---

④ 运行程序，测试结果。

写一个程序测试一下Before Advice的运作。

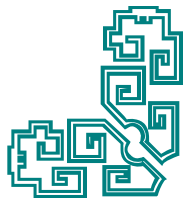
➤ SpringAOPDemo.java代码如下：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class SpringAOPDemo{
    public static void main(String[] args){
        ApplicationContext context=new FileSystemXmlApplicationContext("
        /WebRoot/WEB-INF/classes/applicationContext.xml");
        IHello helloProxy=(IHello)context.getBean("helloProxy");
        helloProxy.hello("Justin");
    }
}
```

➤ 程序运行结果：

Hello,Justin

HelloSpeaker与LogBeforeAdvice是两个独立的类。对于HelloSpeaker来说，它不用知道LogBeforeAdvice的存在；而LogBeforeAdvice也可以运行到其他类之上。HelloSpeaker与LogBeforeAdvice都可以重复使用。





## 5.4.5 切入点Pointcut

创建一个切入点Pointcut项目，步骤如下。

① 创建一个Web项目，命名为“Spring\_Pointcut”。

② 编写Java类。

➤ IHello.java代码如下：

```
public interface IHello{  
    public void helloNewbie(String name);  
    public void helloMaster(String name);  
}
```

HelloSpeaker类实现IHello接口。HelloSpeaker.java代码如下：

```
public class HelloSpeaker implements IHello{  
    public void helloNewbie(String name){  
        System.out.println("Hello, "+name+"newbie! ");  
    }  
    public void helloMaster(String name){  
        System.out.println("Hello, "+name+"master! ");  
    }  
}
```

③ 添加Spring开发能力。

➤ applicationContext.xml的[代码](#)。

## 5.4.5 切入点Pointcut

---

④ 运行程序，测试结果。

➤ SpringAOPDemo.java代码如下：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class SpringAOPDemo {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext(
            "/WebRoot/WEB-INF/classes/applicationContext.xml");
        IHello helloProxy = (IHello) context.getBean("helloProxy");
        helloProxy.helloNewbie("Justin");
        helloProxy.helloMaster("Tom");
    }
}
```

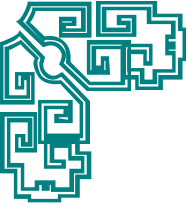
➤ 程序运行结果：

Hello, Justinnewbie!

Hello, Tommaster!

2009-5-21 17:16:34 LogBeforeAdvice before

---

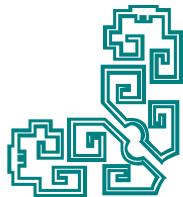
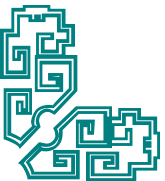


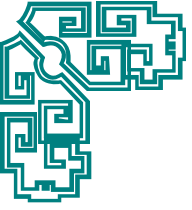
## 5.5 Spring事务支持

---

声明式事务管理的配置方式，通常有以下4种：

- ① 使用**TransactionProxyFactoryBean**为目标Bean生成事务代理的配置。此方式是最传统、配置文件最臃肿、最难以阅读的方式。
- ② 采用Bean继承的事务代理配置方式，比较简洁，但依然是增量式配置。
- ③ 采用**BeanNameAutoProxyCreator**，根据Bean Name自动生成事务代理的方式。这是直接利用Spring的AOP框架配置事务代理的方式，需要对Spring的AOP框架有所理解。但这种方式避免了增量式配置，效果非常不错。
- ④ 采用**DefaultAdvisorAutoProxyCreator**，直接利用Spring的AOP框架配置事务代理的方式，效果非常不错，只是这种配置方式的可读性不如第3种方式。

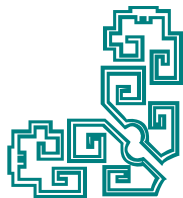
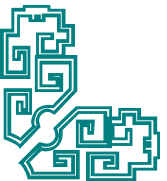




## 5.5.1 使用TransactionProxy FactoryBean生成事务代理

---

采用这种方式的配置，配置文件增加得非常快。每个Bean需要两个Bean配置，一个是目标Bean，另外一个使用TransactionProxyFactoryBean配置一个代理Bean。这是一种最原始的配置方式，其配置[文件](#)。

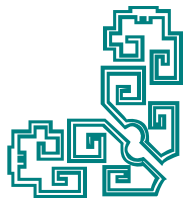
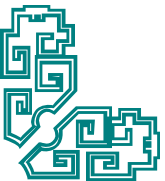




## 5.5.2 利用继承简化配置

---

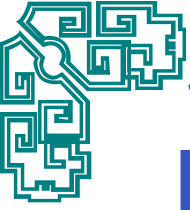
对于这种情况，Spring提供了Bean与Bean之间的继承，可以简化配置。将大部分的通用配置，配置成事务模板。而实际的事务代理Bean，则继承事务模板。这种配置方式可以减少部分配置代码，其配置[文件](#)。



## 5.5.3 用BeanNameAutoProxyCreator自动创建事务代理(事务拦截器)

下面介绍一种优秀的事务代理配置策略：采用这种配置策略，完全可以避免增量式配置，所有的事务代理由系统自动创建。容器中的目标Bean自动消失，避免需要使用嵌套Bean来保证目标Bean不可被访问。下面是采用BeanNameAutoProxyCreator配置事务代理的配置文件。

TranscationInterceptor是一个事务拦截器Bean，需要传入一个TransactionManager的引用。配置中使用Spring依赖注入该属性，事务拦截器的事务属性通过transactionAttributes来指定，该属性有props子元素，配置文件中定义了2个事务传播规则。



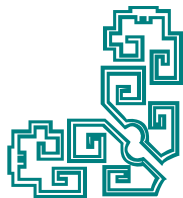
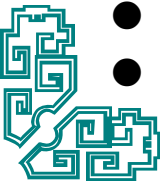
## 5.5.4 用DefaultAdvisorAutoProxyCreator自动创建事务代理

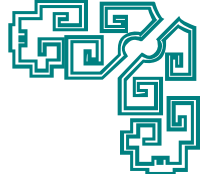
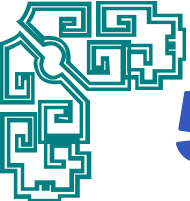
采用DefaultAdvisorAutoProxyCreator的事务代理配置方式更加简单，这个代理生成器自动搜索Spring容器中的Advisor，并为容器中所有的Bean创建代理。相对前一种方式，这种方式的可读性不如前一种直观，所以还是推荐采用前一种配置方式。下面是该方式下的配置[文件](#)。

不管是哪种方式，都定义了事务传播的属性，下面具体介绍事务传播的种类。

Spring在TransactionDefinition接口中规定了7种类型的事务传播行为，它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播。

- PROPAGATION\_REQUIRED: 如果当前没有事务，就新建一个事务；
- PROPAGATION\_SUPPORTS: 支持当前事务。如果当前没有事务，就以非事务方式执行。
- PROPAGATION\_MANDATORY: 使用当前的事务。如果当前没有事务，就抛出异常。
- PROPAGATION\_REQUIRES\_NEW: 新建事务。如果当前存在事务，把当前事务挂起。
- PROPAGATION\_NOT\_SUPPORTED: 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- PROPAGATION\_NEVER: 以非事务方式执行。
- PROPAGATION\_NESTED: 如果当前存在事务，则在嵌套事务内执行。





## 5.6 Spring与Struts 2整合应用

---

开发一个Spring与Struts 2的整合项目的步骤如下。

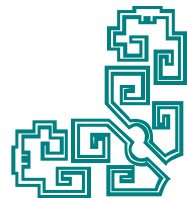
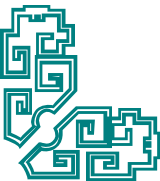
① 创建Web项目Struts\_Spring。

② 添加Struts 2框架。

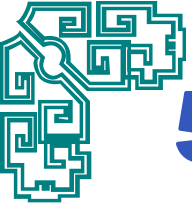
添加5个Jar包：struts 2-core-2.0.14.jar，xwork-2.0.4.jar，ognl-2.6.11.ar，common-logging-1.0.4.jar，freemarker-2.3.8.jar。

➤ 配置web.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```







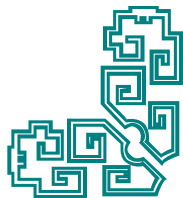
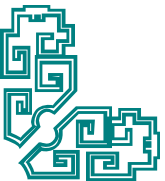
## 5.6 Spring与Struts 2整合应用

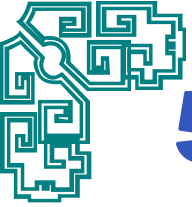
---

③ 创建login.jsp。

➤ login.jsp代码如下：

```
<%@ page language="java" pageEncoding="utf-8"%>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
  <title>登录界面</title>
</head>
<body>
  <s:form action="login.action" method="post">
    <s:textfield name="xh" label="学号"/>
    <s:password name="kl" label="口令"/>
    <s:submit value="登录"/>
  </s:form>
</body>
</html>
```





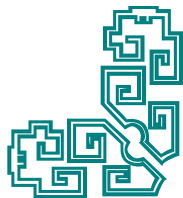
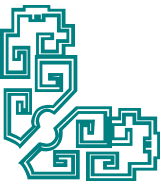
## 5.6 Spring与Struts 2整合应用

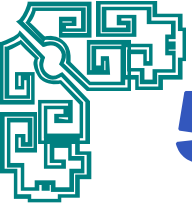
---

### ④ 创建Action。

#### ➤ LoginAction.java代码如下：

```
package org.action;
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport{
    private String xh;
    private String kl;
    public String getXh() {
        return xh;
    }
    public void setXh(String xh) {
        this.xh = xh;
    }
    public String getKl() {
        return kl;
    }
    public void setKl(String kl) {
        this.kl = kl;
    }
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```



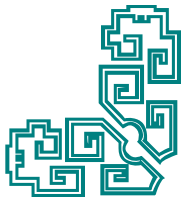
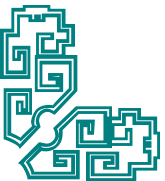


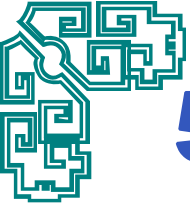
## 5.6 Spring与Struts 2整合应用

---

- 配置struts.xml文件，代码如下：

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <include file="struts-default.xml"/>
    <package name="default" extends="struts-default">
        <action name="login" class="org.action.LoginAction">
            <result name="success">/login_success.jsp</result>
        </action>
    </package>
</struts>
```





## 5.6 Spring与Struts 2整合应用

---

⑤ 创建login\_success.jsp。

➤ 代码如下：

```
<%@ page contentType="text/html;charset=gb2312" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<body >
<h2>您好! <s:property value=" xh"/>欢迎您登录成功 </h2>
</body>
</html>
```

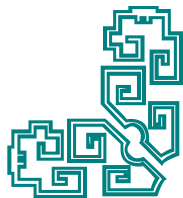
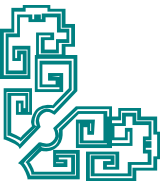
⑥ 部署运行。

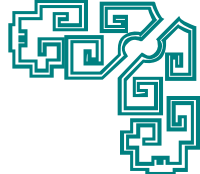
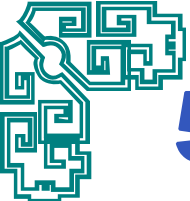
部署，测试Struts 2是否正常。运行

[http://localhost:8080/Struts\\_Spring/login.jsp](http://localhost:8080/Struts_Spring/login.jsp)，在登录框和密码框中任意输入，单击【登录】按钮，转向登录成功界面，并输出登录名。

⑦ 添加Spring框架。

步骤同5.2.2节。





## 5.6 Spring与Struts 2整合应用

⑧ 添加Spring支持包struts2-spring-plugin.jar。

注意，一定要加入该包，该包位于Struts 2的lib目录下。

⑨ 修改web.xml内容。

修改web.xml内容，使得程序增加对Spring的支持。

⑩ 创建消息包文件struts.properties。

在src文件夹下创建struts.properties文件，使得Struts 2的类的生成交给Spring完成。步骤如下：右击项目的src文件夹，在弹出的菜单中选择【New】→【File】菜单项，之后在Enter or select the parent folder中输入struts2\_spring/src，在File name栏中写入struts.properties，单击【确定】按钮。文件内容如下：

```
struts.objectFactory=spring
```

⑪ 修改applicationContext.xml。

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

```
<beans
```

```
    xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

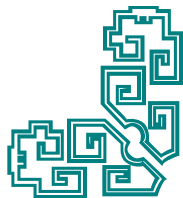
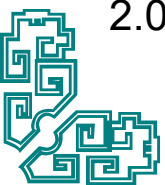
```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

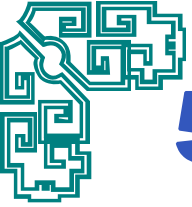
```
        http://www.springframework.org/schema/beans/spring-beans-
```

```
        2.0.xsd">
```

```
        <bean id="loginAction" class="org.action.LoginAction"></bean>
```

```
    </beans>
```





## 5.6 Spring与Struts 2整合应用

---

⑫ 修改struts.xml。

使得struts 2的类的生成交给Spring完成。

```
<!DOCTYPE struts PUBLIC
```

```
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
```

```
    "http://struts.apache.org/dtds/struts-2.0.dtd">
```

```
<struts>
```

```
    <include file="struts-default.xml"/>
```

```
    <package name="default" extends="struts-default">
```

```
<!--使用Spring生成的类对象 -->
```

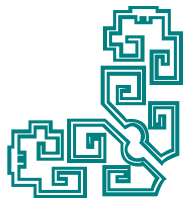
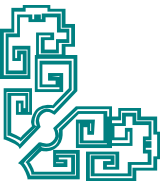
```
    <action name="login" class="loginAction">
```

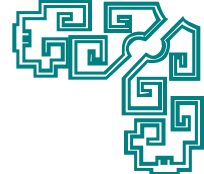
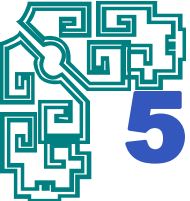
```
        <result name="success">/login_success.jsp</result>
```

```
    </action>
```

```
    </package>
```

```
</struts>
```





## 5.7 Spring与Hibernate整合应用

下面以一个简单的实例说明Spring与Hibernate的整合策略，步骤如下。

- ① 在SQL Server 2005中创建数据库表。  
数据库名为XSCJ，表见附录A 的登录表。
- ② 创建Web项目，命名为“Hibernate\_Spring”。
- ③ 添加Spring的开发能力。

右击项目名，选择

【MyEclipse】→【Add Spring Capabilities...】菜单项，将出现如图5.9所示的对话框，选中要应用的Spring的版本及所需的类库文件。注意，本书用的Spring版本为Spring 2.0。选择Spring的核心类库Spring 2.0 Core Libraries、Spring 2.0 Web Libraries、Spring 2.0 AOP Libraries和Spring 2.0 Persistence JDBC Libraries。

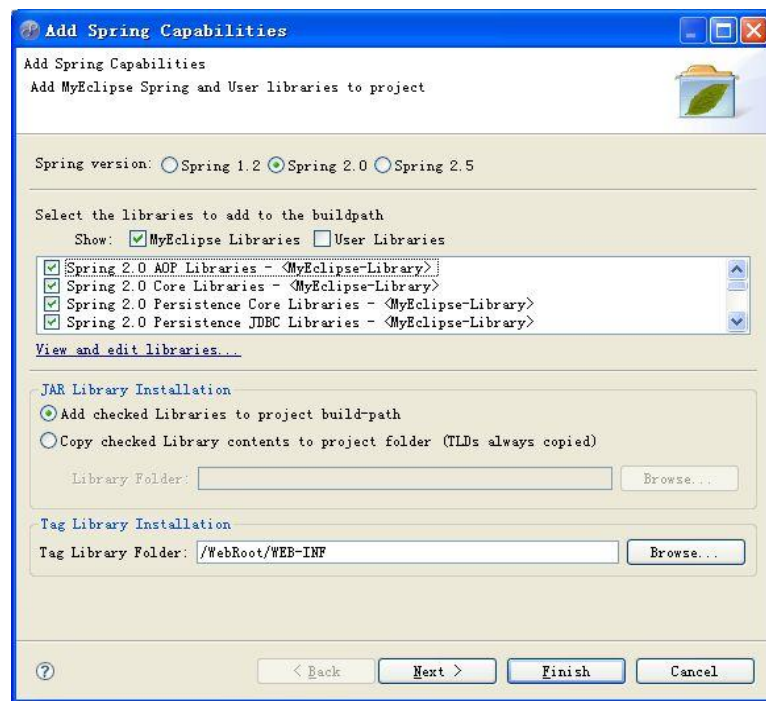
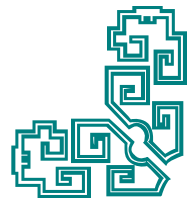
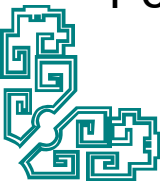
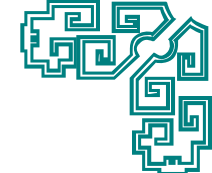
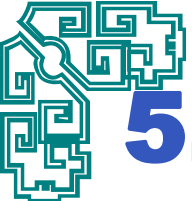


图5.9 添加Spring Capabilities





## 5.7 Spring与Hibernate整合应用

单击【Next】按钮，提示是否建立Spring配置文件，在默认情况下选择application Context.xml文件的存放路径，选择在WEB-INF文件夹下，然后单击【Finish】按钮，如图5.10所示。

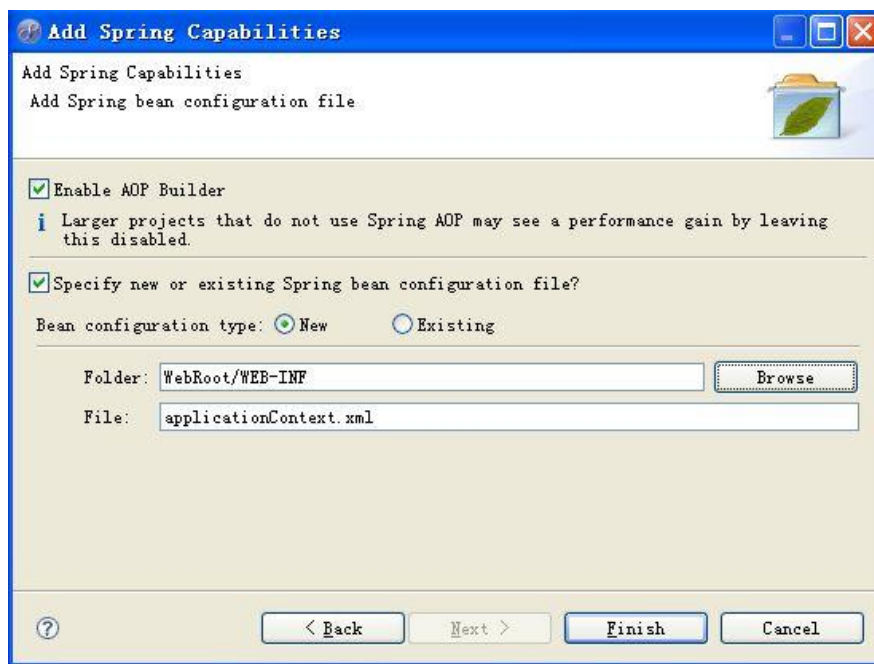
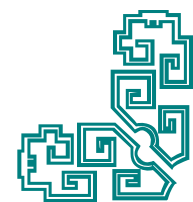
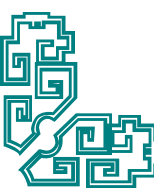


图5.10 创建applicationContext.xml文件





# 5.7 Spring与Hibernate整合应用

④ 加载Hibernate框架。

右击工程文件，选择【MyEclipse】→【Add Hibernate Capabilities...】菜单项，出现如图5.11所示的对话框。选择Hibernate版本及需要的类库。

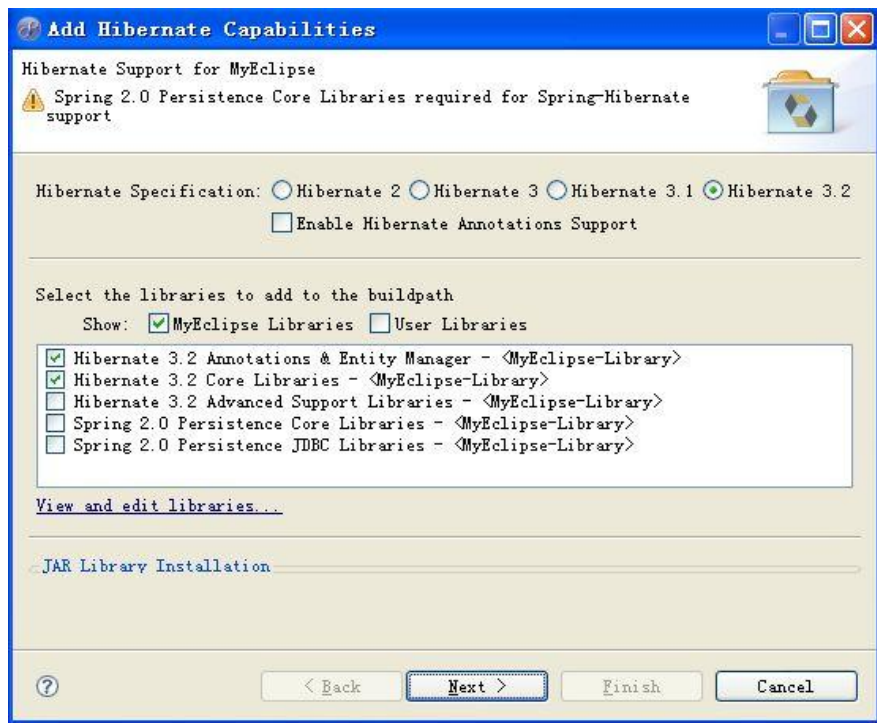


图5.11 添加Hibernate库

# 5.7 Spring与Hibernate整合应用

单击【Next】按钮，出现如图5.12所示对话框，提示是用Hibernate的配置文件还是用Spring的配置文件进行SessionFactory的配置，选择使用Spring来对Hibernate进行管理。这样最后生成的工程中就不包含hibernate.cfg.xml，好处是在一个地方就可以对Hibernate进行管理。

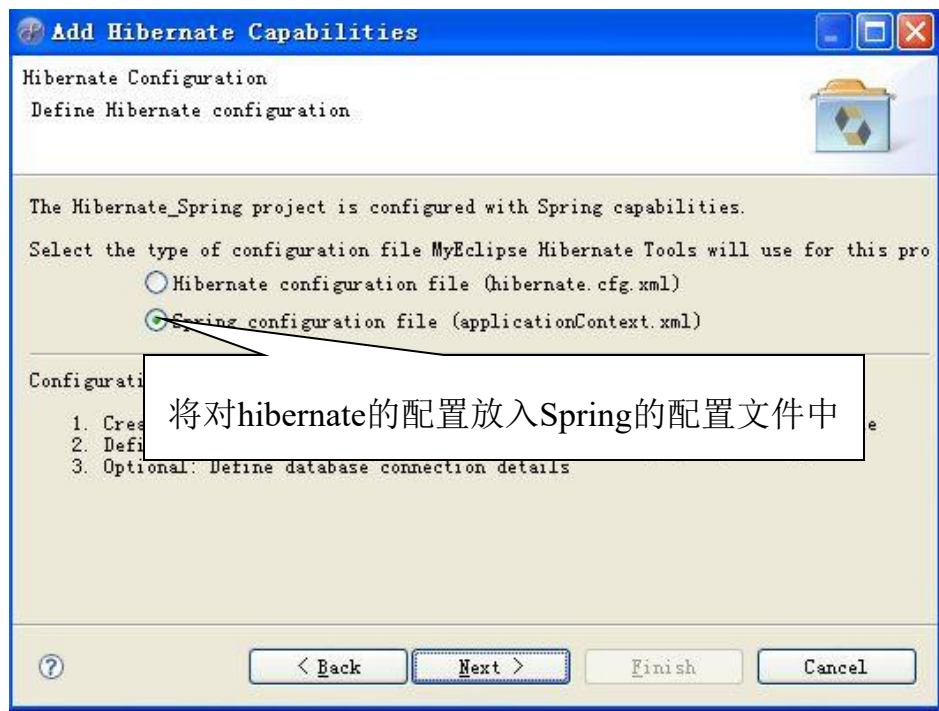


图5.12 定义Hibernate配置

# 5.7 Spring与Hibernate整合应用

单击【Next】按钮，出现如图5.13所示的对话框，提示是创建一个新的Hibernate配置文件还是使用已有的配置文件，由于刚才已经生成了Spring配置文件，并且要在其中进行Hibernate的配置，所以选择复选框“Existing Spring configuration file”。



图5.13 定义Spring-Hibernate配置

# 5.7 Spring与Hibernate整合应用

单击【Next】按钮，出现如图5.14所示的对话框，要求选择数据库连接信息。这里需要注意一点，Bean Id处填写数据源的名称，如“datasource”。数据源的创建请参考4.2.1节中的第2步。

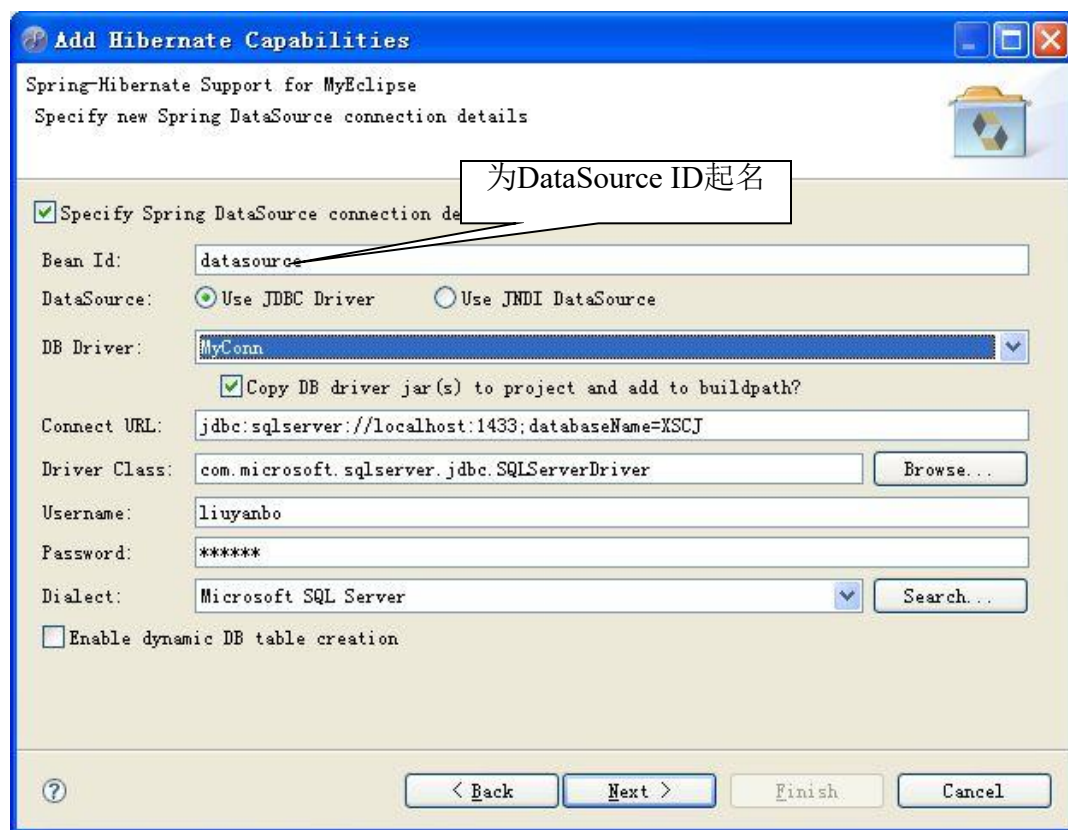
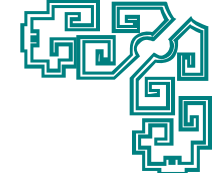
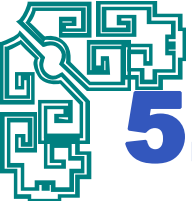


图5.14 指定新的Spring数据源连接信息



## 5.7 Spring与Hibernate整合应用

单击【Next】按钮，出现如图5.15所示的对话框，提示是否创建SessionFactory类，由于本程序Spring为注入sessionFactory，所以不用创建，单击【Finish】按钮。

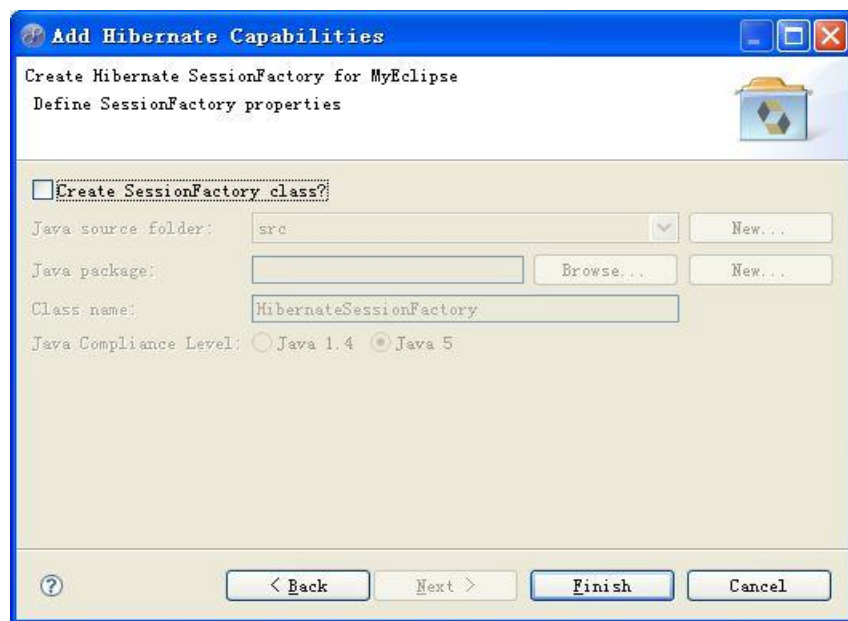
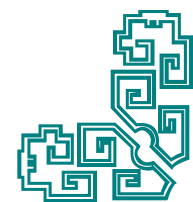
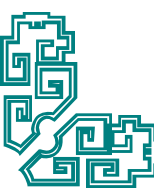


图5.15 定义SessionFactory属性



# 5.7 Spring与Hibernate整合应用

⑤ 生成与数据库表对应的Java数据对象和映射。

打开MyEclipse的Database Explorer Perspective, 右击DLB表, 选择Hibernate Reverse Engineering菜单项, 如图5.16所示设置。

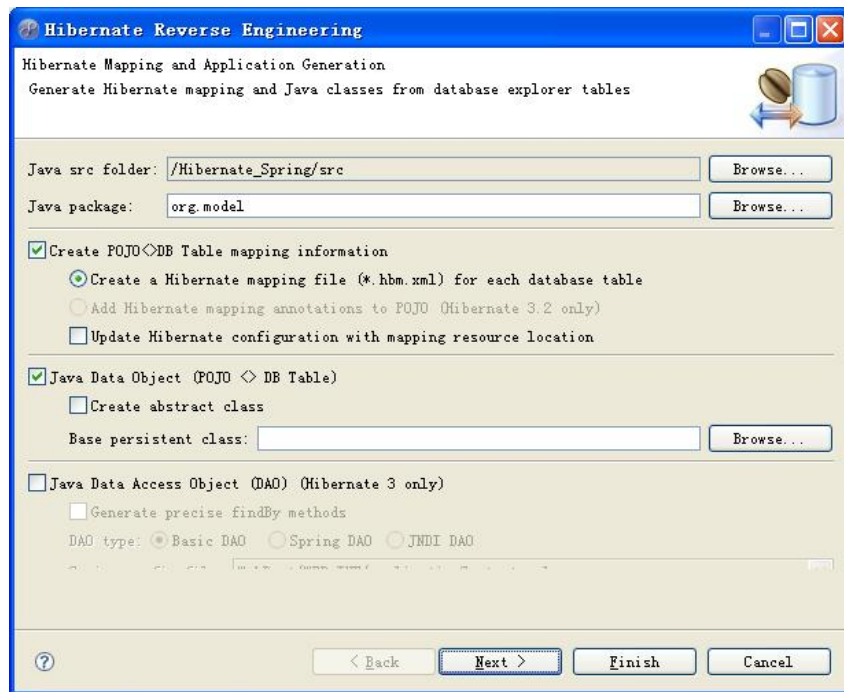
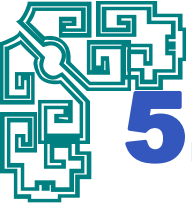


图5.16 Hibernate映射文件和POJO类



## 5.7 Spring与Hibernate整合应用

---

单击【Next】按钮，在ID generator中选择native，直接单击【Finish】按钮完成。

⑥ 编写DlDao.java接口。

在src文件夹下建立包org.dao，在该包先建立接口，命名为“DlDao”，这里主要以添加用户为例，代码如下：

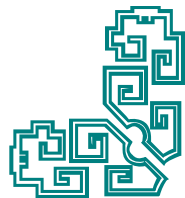
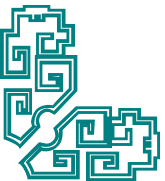
```
package org.dao;  
import org.model.Dlb;  
public interface DlDao {  
    public void save(Dlb dl);  
}
```

⑦ 编写DlDao.java实现类。

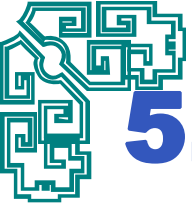
在src文件夹下建立包org.dao.imp，在该包下建立类，命名为“DlDaoImp”，  
代码。

⑧ 修改Spring配置文件applicationContext.xml。

applicationContext.xml文件的代码修改。







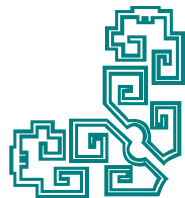
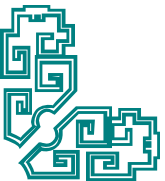
## 5.7 Spring与Hibernate整合应用

---

⑨ 编写测试类。

在src文件夹下建立包test，在该包下建立类Test，代码如下：

```
package test;
import org.dao.DlDao;
import org.model.Dlb;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Test {
    public static void main(String[] args){
        Dlb dlb=new Dlb();
        dlb.setXh("081109");
        dlb.setKl("123456");
        ApplicationContext context=new
            FileSystemXmlApplicationContext("WebRoot/WEB-INF/applicationContext.xml");
        DlDao dlDao=(DlDao) context.getBean("dlDao");
        dlDao.save(dlb);
    }
}
```





## 5.7 Spring与Hibernate整合应用

运行该测试类后，打开数据库，可以发现在DLB表中添加了一项记录，如图5.17所示。

表 - dbo.DLB 摘要			
	ID	XH	KL
	1	081109	123456

图5.17 登录表

Spring的Hibernate ORM 框架带来了方便的HibernateDaoSupport类，该类为Dao类提供了非常方便的方法getHibernateTemplate()，Dao类只要继承HibernateDaoSupport就可以使用该方法，例如上例的Dao实现类可以改成如下的代码：

```
package org.dao.imp;
import org.dao.DIDao;
import org.model.Dlb;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
public class DIDaoImp extends HibernateDaoSupport implements DIDao{
    public void save(Dlb dl) {
        getHibernateTemplate().save(dl);
    }
}
```