# Spring教程

by李雪林 2017-10-20

# 第一章、Spring概述

## spring是什么东西？

Spring是一套开源框架，是免费使用的，所有的源代码是对外公开的。框架是一套骨架，提供一些基础的功能，具体细节是由开发人员去补充完善。一般在公司都会使用框架去开发项目，第一，开发方便，快捷。第二，开发稳定可靠。第三，性能相对来说都是非常好的。

**Spring主要掌握其两样思想：IOC/AOP**

**IOC: 反转控制。通过spring的配置文件来创建对象和组装对象间依赖关系。**

**AOP:面向切面编程。**

**OOP:面向对象编程。**

## spring能做什么？

Spring框架除了不能帮我们写业务代码，其他的东西都能做。给我们管理对象及其依赖关系, 不管是JAVA SE（C/S架构）应用程序还是JAVA EE（B/S架构）应用程序都可以使用这个平台进行开发。

1. 传统程序开发，创建对象及组装对象间依赖关系由我们在程序内部进行控制，这样会加大各个对象间的耦合。而如果采用Spring，则由Spring根据配置文件来进行创建及组装对象间依赖关系，只需要改配置文件即可，无需重新编译。所以，**Spring能帮我们根据配置文件创建及组装对象之间的依赖关系。这个主要体现在spring的IOC设计思想里面。**
2. 采用Spring，这些日志记录、权限控制、性能统计从业务逻辑中分离出来，通过Spring支持的面向切面编程，在需要这些功能的地方动态添加这些功能，无需渗透到各个需要的方法或对象中。

**Spring 面向切面编程能帮助我们无耦合的实现日志记录，性能统计，安全控制。**

1. **Spring能非常简单的帮我们管理数据库事务。**
2. **Spring还提供与第三方Web（如Struts、JSF）框架无缝集成。**老一点的项目使用SSH,现在大部分的项目使用SSM。

## 为什么需要Spring

**1、应用程序**：是能完成我们所需要功能的成品，比如购物网站、OA系统。

**2、框架**：是能完成一定功能的半成品，比如我们可以使用框架进行购物网站开发；框架做一部分功能，我们自己做一部分功能，这样应用程序就创建出来了。

**3**. **非侵入式设计**：从框架角度可以这样理解，无需继承框架提供的类，这种设计就可以看作是非侵入式设计，如果继承了这些框架类，就是侵入设计，如果以后想更换框架之前写过的代码几乎无法重用，如果非侵入式设计则之前写过的代码仍然可以继续使用。

**4**.**轻量级的框架**，轻量级的意思：就是说在开发程序的过程中，必须大量的使用框架本身的一些特性或者定义在框架里面定义好的类，所以需要修改代码或者更换应用框架的话，程序项目本身不会受到太大的改动，这是轻量级的框。比如现在用的spring就是轻量级的框架，以前的EJB是重量级。

**轻量级与重量级的区别：总结起来就是框架本身对项目的改动影响的大与小。**

**轻量级与非侵入式是相辅相成，是绑定起来的。**

**5.** **控制反转：**即Inversion of Control，缩写为IoC，控制反转还有一个名字叫做依赖注入（Dependency Injection），就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。

## spring的优点

1. **非常轻量级的容器**：以集中的、自动化的方式进行应用程序对象创建和装配，负责对象创建和装配，管理对象生命周期，能组合成复杂的应用程序。
2. **IOC：反转控制。**Inversion of controller。
3. **AOP: 面向切面编程。**Aspect oriented programming
4. **事务控制。**
5. **JDBC模板及ORM框架支持。**ORM框架：Hibernate/Mybatis。
6. **灵活的Web层支持。**SpringMVC
7. **简化各种技术集成**：提供对Java Mail、任务调度、JMX、JMS、JNDI、EJB、动态语言、远程访问、Web Service等的集成。

## 5.如何学好spring

**spring的核心思想：IOC和AOP，所以要重点IOC和AOP，其他的东西都是基于这些点做出来的。**

# 第二章、IOC详解

## 1.IOC:inversion of control 控制反转。

IOC的核心功能就两点：1.创建对象 2.依赖注入（DI）。

有了IoC/DI的容器后，在客户端类中不再主动去创建对象，由IOC容器帮我实现对象的创建和依赖关系的管理，依赖注入DI。

IoC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

## 2.IOC底层原理。

1.使用DOM4J解析XML文件。得到bean的名字，拿到类的全限定名字。

比如：String classValue = “com.csxinhua.spring.HelloWorld”

2.使用java的反射机制创建类的对象

Class clazz = Class.forName (classValue)

3.获取类的实例。

HelloWorld helloWorld = (HelloWorld)clazz. newInstance();

4.返回HelloWorld

return helloWorld

## 3.IOC是如何工作的？

**一、准备配置文件**：就像前边Hello World配置文件一样，在配置文件中声明Bean定义也就是为Bean配置元数据。

**二、由IoC容器进行解析元数据：** IoC容器的Bean Reader读取并解析配置文件，根据定义生成BeanDefinition配置元数据对象，IoC容器根据BeanDefinition进行实例化、配置及组装Bean。

**三、实例化IoC容器：**由客户端实例化容器，获取需要的Bean。

## Spring获取bean的三种方式

### 第一种：无参构造方法实例化bean

Xml文件配置

|  |
| --- |
| <!-- 无参构造方法实例化bean -->  <bean id=*"student"* class=*"com.csxinhua.bean.Student"*></bean> |

实体bean类

|  |
| --- |
| **package** com.csxinhua.bean;  **public** **class** Student {  } |

### 第二种 有参构造方法实例化bean

Xml 文件配置

|  |
| --- |
| <!-- 有参构造方法实例化bean -->  <bean id=*"student1"* class=*"com.csxinhua.bean.Student1"*>  <constructor-arg index=*"0"* value=*"zhangsan"*></constructor-arg>  </bean> |

Bean实体类

|  |
| --- |
| **package** com.csxinhua.bean;  **public** **class** Student1 {  **private** String name;  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getName(){  **return** name;  }  **public** Student1(String name){  **this**.name = name;  }  } |

### 第三种：静态工厂方法实例化bean

|  |
| --- |
| <!-- 静态工厂方法实例化bean -->  <!-- class 属性指定生成bean的工厂类 factory-method指定生成bean的静态方法 -->  <bean id=*"bean"* class=*"com.csxinhua.factory.StaticFactory"* factory-method=*"getInstance"*></bean> |

|  |
| --- |
| **package** com.csxinhua.factory;  **import** com.csxinhua.bean.Bean;  **public** **class** StaticFactory {  **public** **static** Bean getInstance(){  **return** **new** Bean();  }  } |

### 第四种：工厂方法实例化bean

|  |
| --- |
| <!-- 工厂方法实例化bean -->  <!-- 先实例化工厂对象 -->  <bean id=*"factory"* class=*"com.csxinhua.factory.Factory"*></bean>  <!-- 再通过工厂对象生成bean -->  <bean id=*"bean1"* factory-bean=*"factory"* factory-method=*"getInstance"*></bean> |

|  |
| --- |
| **package** com.csxinhua.factory;  **import** com.csxinhua.bean.Bean1;  **public** **class** Factory {  **public** Bean1 getInstance(){  **return** **new** Bean1();  }  } |

## Spring Bean标签常用属性

### （1）id属性：起名称，id属性值名称任意命名

- id属性值，不能包含特殊符号

- 根据id值得到配置对象

### （2）class属性：创建对象所在类的全路径

### （3）name属性：功能和id属性一样的，id属性值不能包含特殊符号，但是在name属性值里面可以包含特殊符号

### （4）scope属性

- singleton：默认值，单例

**- request：创建对象把对象放到request域里面**

**- session：创建对象把对象放到session域里面**

**- globalSession：创建对象把对象放到globalSession里面**

## Spring依赖注入（DI）

### 1 创建对象时候，向类里面属性里面设置值。

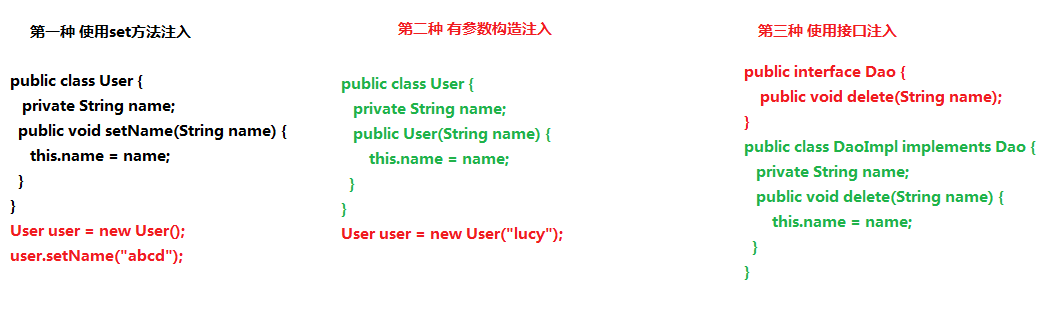
### 2 属性注入的方式介绍（三种方式）

（1）使用set方法注入（重点）

（2）使用有参数构造注入

（3）使用接口注入

Spring框架支持前面两种方式。

****

第一种方式：set方法注入

|  |
| --- |
| <!-- 使用set方法注入属性值 -->  <bean id=*"bean"* class=*"com.spring.bean.Bean"*>  <property name=*"beanName"* value=*"使用set方法注入属性值"*></property>  </bean> |

|  |
| --- |
| **package** com.spring.bean;  **public** **class** Bean {  **private** String beanName;  **public** String getBeanName() {  **return** beanName;  }  **public** **void** setBeanName(String beanName) {  **this**.beanName = beanName;  }    } |

第二种方式：使用有参构造方法注入属性的值

|  |
| --- |
| <!-- 使用有参构造方法注入属性值 -->  <bean id=*"bean1"* class=*"com.spring.bean.Bean1"*>  <constructor-arg index=*"0"* value=*"使用有参构造方法注入属性值"*></constructor-arg>  </bean> |

|  |
| --- |
| **package** com.spring.bean;  **public** **class** Bean {  **private** String beanName;  **public** String getBeanName() {  **return** beanName;  }  **public** **void** setBeanName(String beanName) {  **this**.beanName = beanName;  }    } |

第三种方式：注入对象类型属性（重点）

|  |
| --- |
| <!-- 注入对象属性 -->  <bean id=*"bean2"* class=*"com.spring.bean.Bean2"*></bean>  <bean id=*"bean3"* class=*"com.spring.bean.Bean3"*>  <property name=*"bean2"* ref=*"bean2"*></property>  </bean> |

|  |
| --- |
| //注入对象属性  Bean3 bean3 = (Bean3)ApplicationContextUtil.*getInstance*().getBean("bean3");  System.***out***.println(bean3.getBean2()); |

第四种方式：注入复杂类型属性数据

|  |
| --- |
| <!-- 注入复杂类型属性 -->  <bean id=*"bean4"* class=*"com.spring.bean.Bean4"*>  <!-- 数组 -->  <property name=*"strArr"*>  <list>  <value>a</value>  <value>b</value>  <value>c</value>  </list>  </property>    <!-- list -->  <property name=*"strList"*>  <list>  <value>d</value>  <value>e</value>  <value>f</value>  </list>  </property>    <!-- map -->  <property name=*"strMap"*>  <map>  <entry key=*"aa"* value=*"g"*></entry>  <entry key=*"bb"* value=*"h"*></entry>  <entry key=*"cc"* value=*"i"*></entry>  </map>  </property>    <!-- properties -->  <property name=*"properties"*>  <props>  <prop key=*"driverclass"*>com.mysql.jdbc.Driver</prop>  <prop key=*"username"*>root</prop>  </props>  </property>  </bean> |

|  |
| --- |
| **package** com.spring.bean;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Properties;  **public** **class** Bean4 {  **private** String[] strArr;  **private** List<String> strList;  **private** Map<String,String> strMap;  **private** Properties properties;  **public** String[] getStrArr() {  **return** strArr;  }  **public** **void** setStrArr(String[] strArr) {  **this**.strArr = strArr;  }  **public** List<String> getStrList() {  **return** strList;  }  **public** **void** setStrList(List<String> strList) {  **this**.strList = strList;  }  **public** Map<String, String> getStrMap() {  **return** strMap;  }  **public** **void** setStrMap(Map<String, String> strMap) {  **this**.strMap = strMap;  }  **public** Properties getProperties() {  **return** properties;  }  **public** **void** setProperties(Properties properties) {  **this**.properties = properties;  }    } |

## IOC和DI区别

（1）IOC: 控制反转，把对象创建交给spring进行配置。

（2）DI: 依赖注入，向类里面的属性中设置值。

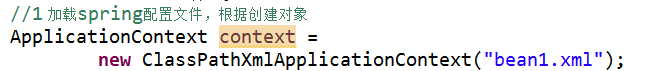
（3）关系：依赖注入不能单独存在，需要在ioc基础之上完成操作

## 8.ClassPathXmlApplicationContext类的效率问题。

在前面讲的例子中发现，想要从IOC容器中获取得到一个bean实例对象，都得先得到ClassPathXmlApplicationContext对象，在复杂的项目中，类的实例个数并不是小数量计算。如果每得到一个实例对象都需要new一下ClassPathXmlApplicationContext对象，那么可想而知，项目的性能和效率低下。

## ClassPathXmlApplicationContext避免使用new对象的解决方法。

### 从根本上来说，我们的目的就是要高效的获取ClassPathXmlApplicationContext实例对象。加载spring核心配置文件。

****

### 怎么高效的获取ClassPathXmlApplicationContext对象？

解决思路：spring主要在web项目中使用，所以将加载配置文件的过程和创建对象在服务器启动的时候完成。

### 实现原理：

3 实现原理

#### （1）ServletContext对象

#### （2）监听器

#### （3）具体使用：

- 在服务器启动时候，为每个项目创建一个ServletContext对象。

- 在ServletContext对象创建时候，使用监听器可以监听到ServletContext对象在什么时候创建。

- 当监听器监听到ServletContext对象创建的时候，加载spring配置文件，把配置文件配置对象创建。

- 把创建出来的对象放到ServletContext域对象里面（setAttribute方法）。

- 获取对象时候，到ServletContext域得到 （getAttribute方法）。

#### 具体配置：

在spring里面不需要我们自己写代码实现，已经帮我们封装好了，所以只需要按照spring的规范去配置。

（1）封装了一个监听器，只需要配置监听器 就可以了

（2）配置监听器之前做事情：导入spring整合web项目jar包。

****

|  |
| --- |
| <!-- 指定spring配置文件的地址 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value>  </context-param>  <!-- 配置spring监听器 -->  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> |

## Spring注解开发。

### 注解介绍：格式如@Component(vaule=”xxxx”)

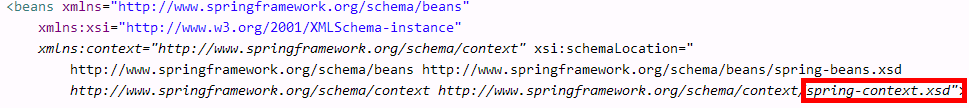
1 代码里面特殊标记，使用注解可以完成功能

2 注解写法 @注解名称(属性名称=属性值)

1. 注解使用在类上面，方法上面 和 属性上面
2. 导入aop的jar包

****

5.创建spring配置文件，配置注解约束

****

## Spring注解applicationContext.xml配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>    <!-- 开启注解扫描，到base-package所指定的包下面扫描类、属性、方法上面是否有注解 -->  <context:component-scan base-package=*"com.spring"*></context:component-scan>  <!-- 扫描属性上面的注解 -->  <!-- <context:annotation-config></context:annotation-config> -->  </beans> |

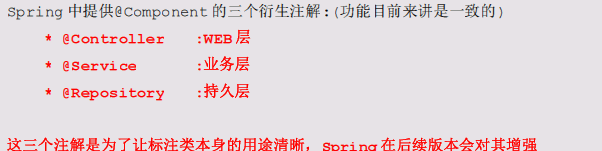
## 注解创建对象@Component

1.类上面加入注解@Component(value=”xxxx”)注解

@Component相当于实例化了一个bean对象。

|  |
| --- |
| **package** com.spring.bean;  **import** org.springframework.stereotype.Component;  @Component(value="bean") //<bean id="bean" class="......."></bean>  **public** **class** Bean {  } |

2 创建对象有四个注解

****

（1）@Component

（2）@Controller

（3）@Service

（4）@Repository

目前这四个注解功能是一样的，都是创建对象。

遵循MVC原则，采用分层的结构使用不同的注解，这是Spring在后续项目为了扩展，因为类的功能不同，所以类的名字不一样，有controller类，service类等等。。。

1. 创建对象单实例还是多实例@Scope

|  |
| --- |
| **package** com.spring.bean;  **import** org.springframework.context.annotation.Scope;  **import** org.springframework.stereotype.Component;  @Component(value="bean")  @Scope(value="prototype")  **public** **class** Bean1 {  } |

## 注解注入属性的值@Value

类中属性上面加入注解@Value(value=”xxxx”)注解

@Component相当于xml文件配置中的<property name=”beanName” value=”xxx”></property>

相当于setter方法

|  |
| --- |
| **package** com.spring.bean;  **import** org.springframework.beans.factory.annotation.Value;  **import** org.springframework.stereotype.Component;  @Component(value="bean2")  **public** **class** Bean2 {  @Value(value="使用@Value注入属性的值")  **private** String beanName;  **public** String getBeanName() {  **return** beanName;  }  } |

## 注入属性对象@Autowired/@Resource(name=”xx”)

1.类中的属性是一个对象，加入注解@Autowired(value=”xxxx”)注解

@Autowired注解相当于xml配置文件中:

<property name=”beanName” ref=”xxx”></property>

|  |
| --- |
| **package** com.spring.bean;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Component;  @Component(value="bean3")  **public** **class** Bean3 {    @Autowired  **private** Bean2 bean2;    **public** **void** showInfo(){  System.***out***.println(bean2.getBeanName());  }  } |

1. @Resource(name=”bean2”)注解也可以注入属性对象。

区别在于@Resource注解需要加上name属性，指定注入的对象。

## 配置文件和注解混合使用

1. 创建对象操作使用配置文件方式实现。<bean id=”bean” class=”xxxxx”></bean>
2. 注入属性的操作使用注解方式实现。使用@Resource(name=”xxx”);

## AOP概念：Aspect Orentied Programming

AOP: Aspect Orentied Programming(面向切面编程)

OOP:Object Orentied Programming(面向对象编程)

* 1. AOP:面向切面（方面）编程，扩展功能不修改源代码实现。
  2. AOP:采取横向抽取机制，取代了传统纵向继承体系重复性代码。
  3. AOP:底层使用动态代理实现

（1）第一种情况，有接口情况，使用动态代理创建接口实现类代理对象

（2）第二种情况，没有接口情况，使用动态代理创建类的子类代理对象

## 17.功能扩展和纵向继承体系

|  |
| --- |
| **package** com.util;  **public** **class** Register {  // 注册功能添加新用户  **public** **void** addUser(){    }  } |

在原本的需求中，添加新用户功能是没有日志记录的，也就是说用户是什么时候注册的并没有记录下来，而此时，我需要在记录新用户的注册时间。

### 第一种方案：直接在类中修改源代码，新增日志记录代码

|  |
| --- |
| **package** com.util;  **public** **class** Register {  // 注册功能添加新用户  **public** **void** addUser(){  // 添加新用户逻辑    // 日志记录逻辑  }  } |

此种方法虽然也能实现最终的目的，但是对原有的代码改动太多，且，如果需要改动的类太多的话，新增的工作量页不少。

### 第二种方案：纵向继承机制。

1.先扩展出一个BaseRegister类。在这个类实现你想要的扩展功能业务逻辑

|  |
| --- |
| **package** com.util;  **public** **class** BaseRegister {  **public** **void** writeLog(){  // 此处为日志添加逻辑  }  } |

2.然后Register类继承BaseRegister类，在类中调用父类的扩展功能方法，实现日志操作。

|  |
| --- |
| **package** com.util;  **public** **class** Register **extends** BaseRegister{  // 注册功能添加新用户  **public** **void** addUser(){  // 添加新用户逻辑    // 调用父类的方法实现日志功能，达到扩展的目的  **super**.writeLog();  }  } |

此种方案比第一种要好的多，至少在代码量上面要少的多，而且如果多个类需要实现此功能，只需要继承就可，但是，也有很明显的缺陷，比如父类的方法名字发生改变，所以子类也得都跟着改动一遍。

## 18.横向抽取机制：采用动态代理机制

### 第一种情况： 有接口。

先定义一个接口：

|  |
| --- |
| **package** com.util;  **public** **interface** RegisterService {  **public** **void** addUser();  } |

然后实现接口，对接口中的方法做好具体的实现

|  |
| --- |
| **package** com.util;  **public** **class** RegisterServiceImpl **implements** RegisterService {  @Override  **public** **void** addUser() {  // 添加新用户逻辑代码  }  } |

使用动态代理机制，创建接口实现类代理对象。这个代理对象中实现扩展功能。而原本的实现类代码不需要改动，同时，由于实现类和代理实现类都是实现同一个接口，所以代理实现类和实现类是平级关系。但是，这个实例对象并不是真正的对象，只是在实现接口的代理类中的方法里面完成了扩展功能。

使用这种接口代理对象最大的好处就是我原本的实现类（这个是真正的类）中代码不需要改动。所以的扩展功能在代理实现类中完成，也就相当于增强了方法的功能。同时对于有接口的情况，使用的是jdk的动态代理。

动态代理的具体实现不需要掌握，重点是这种横向代理的思维，要理解横向代理是什么意思。

### 第二种情况：没有接口

|  |
| --- |
| **package** com.util;  **public** **class** Register **extends** BaseRegister{  // 注册功能添加新用户  **public** **void** addUser(){  // 添加新用户逻辑    }  } |

现在，我要实现addUser方法功能的扩张，那么就使用动态代理的方式创建出Register的动态代理子类，然后在动态代理子类中调用父类的方法，在达到扩张方法功能的目的。这种没有接口的动态代理使用的cglib的动态代理机制。

## 19.AOP操作术语：

以下是AOP的专业术语，不要太在意这些个中文定义，这是咱们国内的软件行业根据国外的文档翻译总结的。

Joinpoint(连接点): 所谓连接点是指那些被拦截到的点。在 spring 中,这些点指的是方法,因为 spring 只支持方法类型的连接点.

Pointcut(切入点):所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

Advice(通知/增强):所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

Aspect(切面): 是切入点和通知（引介）的结合

Introduction(引介):引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.

Target(目标对象):代理的目标对象(要增强的类)

Weaving(织入):是把增强应用到目标的过程.把advice 应用到 target的过程

Proxy（代理）:一个类被AOP织入增强后，就产生一个结果代理类

## 20.AOP术语解释

|  |
| --- |
| **package** com.spring.test;  **public** **class** Register {  **public** **void** addUser(){};  **public** **void** deleteUser(){};  **public** **void** updateUser(){};  **public** **void** selectUser(){};  } |

### Joinpoint连接点：通俗一点的讲就是类里面可以被增强的方法，这些方法称为连接点。

实例类中现在有四个方法，这四个方法都可以被增强，就是类中哪些方法可以被增强，那么这些方法就叫做连接点。

### Pointcut切入点（重点）：在类中可以有很多方法可以被增强，被增强的方法就叫做切入点。

比如说在实践操作中，只是增强了addUser方法和updateUser方法，那么这两个实际增强的方法就都叫做切入点。

### Advice增强（重点）：就是实践操作中，想要添加的功能。

比说要在addUser方法中添加日志记录功能，那么这个日志记录的逻辑代码就叫增强。

增强分为如下几种形式：

前置通知：在方法之前执行。

后置通知：在方法之后执行。

异常通知：方法中出现了异常。

最终通知：在后置之后执行。

环绕通知(切面要完成的功能)。

### Aspect切面（重点）：把增强应用到具体方法上面的过程叫做切面。

比如说，在addUser方法增加日志记录功能，那么把日志功能用在addUser方法上面，那么这个过程就叫做切面。把增强用到切入点的过程就叫做切面。

## Spring的AOP操作。

### 1 在spring里面进行aop操作，使用aspectj实现

（1）aspectj不是spring一部分，和spring一起使用进行aop操作

（2）Spring2.0以后新增了对AspectJ支持

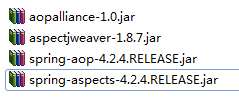
### 2 使用aspectj实现aop有两种方式

（1）基于aspectj的xml配置

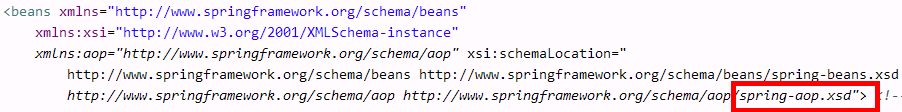
（2）基于aspectj的注解方式

### Aop操作准备

1. 除了导入基本的jar包之外，还需要导入aop相关的jar包。

****

（2）创建spring核心配置文件，导入aop的约束

****

## Spring Aop的xml配置操作。

### 编写连接点和切入点类

|  |
| --- |
| **package** com.spring.aop;  **public** **class** Register {  **public** **void** addUser(){  System.***out***.println("添加新注册的用户，使用aop不修改改方法的源代码来记录日志信息！");  }  **public** **void** deleteUser(){}  **public** **void** updateUser(){}  **public** **void** selectUser(){}  } |

### 编写增强类

|  |
| --- |
| **package** com.spring.aop;  **import** org.aspectj.lang.ProceedingJoinPoint;  **public** **class** LogRecord {  **public** **void** beforeAdvice(){  System.***out***.println("日志记录信息，前置增强");  }  **public** **void** afterAdvice(){  System.***out***.println("日志记录信息，后置增强");  }    **public** **void** aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  System.***out***.println("日志记录信息，环绕前，被增强方法之前执行！");    //此处执行被增强的方法  proceedingJoinPoint.proceed();  System.***out***.println("日志记录信息，环绕后，被增强方法之后执行！");  }  } |

### 编辑xml配置文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>    <bean id=*"register"* class=*"com.spring.aop.Register"*></bean>  <bean id=*"logRecord"* class=*"com.spring.aop.LogRecord"*></bean>    <!-- 进行 aop 的配置 -->  <aop:config>  <!-- 配置切入点表达式:哪些类的哪些方法需要进行增强 -->  <aop:pointcut expression=*"**execution(\* com.spring.aop.Register.addUser(..))"* id=*"pointcutTest"*/>  <!-- 配置切面 把增强用到方法上面 -->  <aop:aspect ref=*"logRecord"*>  <!-- 配置增强类型 -->  <aop:before method=*"beforeAdvice"* pointcut-ref=*"pointcutTest"*/>  <!-- after:最终 after-returning :后置 -->  <aop:after-returning method=*"afterAdvice"* pointcut-ref=*"pointcutTest"*/>  <!-- around 环绕执行 -->  <aop:around method=*"aroundAdvice"* pointcut-ref=*"pointcutTest"*/>  </aop:aspect>  </aop:config>  </beans> |

### 切入点表达式：

1 切入点：实际增强的方法

2 常用的表达式

execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>)

（1）execution(\* com.spring.aop.Register.addUser(..))

（2）execution(\* com.spring.aop.Register.\*(..))

（3）execution(\* \*.\*(..))

（4） 匹配所有save开头的方法 execution(\* save\*(..))

注意：第一个\*号后面添加空格。

## log4j介绍

### 1 通过log4j可以看到程序运行过程中更详细的信息

（1）经常使用log4j查看日志

### 2 使用

（1）导入log4j的jar包

（2）复制log4j的配置文件，复制到src下面



### 3 设置日志级别



（1）info：看到基本信息

（2）debug：看到更详细信息

## Spring Aop注解操作

### 编写连接点和切入点类

|  |
| --- |
| **package** com.spring.aop;  **public** **class** RegisterAnnotation {  **public** **void** addUser(){  System.***out***.println("添加新注册的用户，使用aop不修改改方法的源代码来记录日志信息！");  }  **public** **void** deleteUser(){}  **public** **void** updateUser(){}  **public** **void** selectUser(){}  } |

### 编写增强类

|  |
| --- |
| **package** com.spring.aop;  **import** org.aspectj.lang.ProceedingJoinPoint;  **import** org.aspectj.lang.annotation.AfterReturning;  **import** org.aspectj.lang.annotation.Around;  **import** org.aspectj.lang.annotation.Aspect;  **import** org.aspectj.lang.annotation.Before;  @Aspect  **public** **class** LogRecordAnnotation {    @Before(value="execution(\* com.spring.aop.Register.addUser(..))")  **public** **void** beforeAdvice(){  System.***out***.println("日志记录信息，前置增强");  }    @AfterReturning(value="execution(\* com.spring.aop.Register.addUser(..))")  **public** **void** afterAdvice(){  System.***out***.println("日志记录信息，后置增强");  }    @Around(value="execution(\* com.spring.aop.Register.addUser(..))")  **public** **void** aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  System.***out***.println("日志记录信息，环绕前，被增强方法之前执行！");    //此处执行被增强的方法  proceedingJoinPoint.proceed();  System.***out***.println("日志记录信息，环绕后，被增强方法之后执行！");  }    } |

注意：1.在类上面加上@Aspect注解

2.前置增强@Before

3.后置增强@AfterReturning

4.环绕增强@Around

1. 编写xml配置文件开启AOP注解扫描

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>    <!-- 开启spring的AOP注解功能 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>    <bean id=*"register"* class=*"com.spring.aop.Register"*></bean>  <bean id=*"logRecord"* class=*"com.spring.aop.LogRecord"*></bean>    </beans> |

aop注解代理： <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

## 25.Spring整合web项目演示

Spring整合web项目

### 1 演示问题

（1）action调用service，service调用dao

每次访问action时候，都会加载spring配置文件

### 2 解决方案：

（1）在服务器启动时候，创建对象加载配置文件

（2）底层使用监听器、ServletContext对象

### 3 在spring里面不需要我们自己写代码实现，已经给封装好了

（1）封装了一个监听器，只需要 配置监听器 就可以了

（2）配置监听器之前做事情：导入spring整合web项目jar包

（3）指定加载spring配置文件位置

## 26.Spring的jdbcTemplate操作

### 1 spring框架一站式框架

（1）针对javaee三层，每一层都有解决技术。

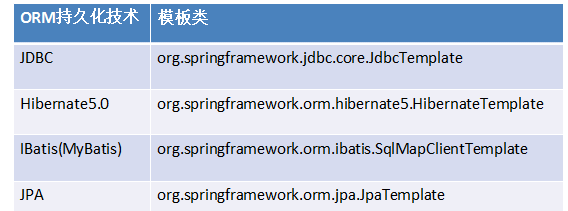
- web层：springMVC

- service层：spring的ioc/aop

- dao层：spring的jdbcTemplate

（2）在dao层，使用 jdbcTemplate。

### spring对不同的持久化层技术都进行封装

****

### 3 jdbcTemplate使用和dbutils使用很相似，对数据库进行crud操作

crud是指在做计算处理时的增加(Create)、读取查询(Retrieve)、更新(Update)和删除(Delete)几个单词的首字母简写。crud主要被用在描述软件系统中数据库或者持久层的基本操作功能。

### jdbcTemplate使用配置流程

1 导入jdbcTemplate使用的jar包





2 创建对象，设置数据库信息

3 创建jdbcTemplate对象，设置数据源

4 调用jdbcTemplate对象里面的方法实现操作。

### jdbcTemplate之增删改查操作

#### 增加操作：

|  |
| --- |
| // 1 添加操作  **public** **void** add() {  // 设置数据库信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.Driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("scott");  dataSource.setPassword("tiger");  // 创建jdbcTemplate对象，设置数据源  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);  // 调用jdbcTemplate对象里面的方法实现操作  // 创建sql语句  String sql = "insert into user values(?,?)";  **int** rows = jdbcTemplate.update(sql, "lucy", "250");  System.***out***.println(rows);  } |

#### 修改操作：

|  |
| --- |
| // 2 修改操作  **public** **void** update() {  // 设置数据库信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.Driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("scott");  dataSource.setPassword("tiger");  // 创建jdbcTemplate对象  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);  // 调用jdbcTemplate里面的方法实现 update方法  String sql = "update user set password=? where username=?";  **int** rows = jdbcTemplate.update(sql, "1314", "lucy");  System.***out***.println(rows);  } |

#### 删除操作：

|  |
| --- |
| **public** **void** delete() {  // 设置数据库信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.Driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("scott");  dataSource.setPassword("tiger");  // 创建jdbcTemplate对象  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);  // 调用update方法实现删除  String sql = "delete from user where username=?";  **int** rows = jdbcTemplate.update(sql, "lucy");  System.***out***.println(rows);  } |

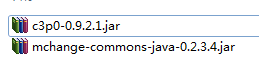
#### 查询操作：

|  |
| --- |
| /\*  \* QueryRunner runner = new QueryRuner(datasource);  \* 返回对象  \* runner.query(sql,new BeanHandler<User>(User.class));  \*  \* 返回list集合  \* runner.query(sql,new BeanListHander<User>(User.class))  \*  \* 1 在dbutils时候，有接口 ResultSetHandler  \* dbutils提供了针对不同的结果实现类  \*  \* 2 jdbcTemplate实现查询，有接口 RowMapper，  \* jdbcTemplate针对这个接口没有提供实现类，得到不同的类型数据需要自己进行数据封装  \*  \* \*/      //2 jdbc实现代码  **public** **void** testJDBC() {  Connection conn = **null**;  PreparedStatement psmt = **null**;  ResultSet rs = **null**;  //加载驱动  **try** {  Class.*forName*("oracle.jdbc.driver.OracleDriver");  //创建连接  conn = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");  //编写sql语句  String sql = "select \* from user where username=?";  //预编译sql  psmt = conn.prepareStatement(sql);  //设置参数值  psmt.setString(1, "lucy");  //执行sql  rs = psmt.executeQuery();  //遍历结果集  **while**(rs.next()) {  //得到返回结果值  String username = rs.getString("username");  String password = rs.getString("password");  //放到user对象里面  User user = **new** User();  user.setUsername(username);  user.setPassword(password);    System.***out***.println(user);  }    } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  rs.close();  psmt.close();  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }    //1 查询表有多少条记录  **public** **void** testCount() {  //设置数据库信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("scott");  dataSource.setPassword("tiger");    //创建jdbcTemplate对象  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);    //调用方法得到记录数  String sql = "select count(\*) from user";  //调用jdbcTemplate的方法  **int** count = jdbcTemplate.queryForObject(sql, Integer.**class**);  System.***out***.println(count);  }    //3 查询返回对象  **public** **void** testObject() {  //设置数据库信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("scott");  dataSource.setPassword("tiger");    //创建jdbcTemplate对象  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);    //写sql语句，根据username查询  String sql = "select \* from user where username=?";  //调用jdbcTemplate的方法实现  //第二个参数是接口 RowMapper，需要自己写类实现接口，自己做数据封装  User user = jdbcTemplate.queryForObject(sql, **new** MyRowMapper(), "mary");  System.***out***.println(user);  }    //4 查询返回对象  **public** **void** testList() {  //设置数据库信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("scott");  dataSource.setPassword("tiger");      //创建jdbcTemplate对象  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);    //写sql语句  String sql = "select \* from user";  //调用jdbcTemplate的方法实现  List<User> list = jdbcTemplate.query(sql,**new** MyRowMapper());    System.***out***.println(list);    }  }  **class** MyRowMapper **implements** RowMapper<User> {  @Override  **public** User mapRow(ResultSet rs, **int** num) **throws** SQLException {  // 1 从结果集里面把数据得到  String username = rs.getString("username");  String password = rs.getString("password");    // 2 把得到数据封装到对象里面  User user = **new** User();  user.setUsername(username);  user.setPassword(password);    **return** user;  } |

## Spring配置连接池C3p0

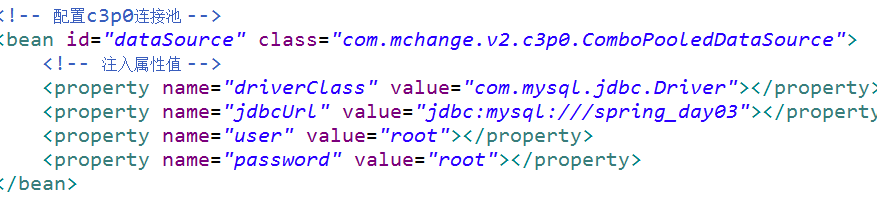
### 1 spring配置c3p0连接池

#### 第一步 导入jar包



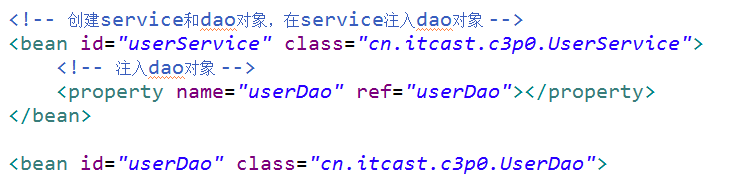
#### 第二步 创建spring配置文件，配置连接池

（1）在配置文件中进行配置

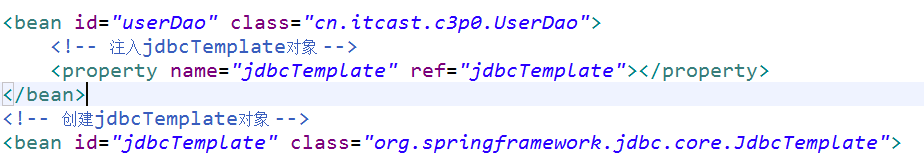


### dao使用jdbcTemplate

#### （1）创建service和dao，配置service和dao对象，在service注入dao对象

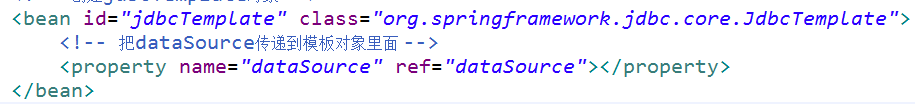


#### 创建jdbcTemplate对象，把模板对象注入到dao里面



#### （3）在jdbcTemplate对象里面注入dataSource

#### 

****

## 使用db.properties文件配置数据源

### 在classpath路径下创建db.properties

|  |
| --- |
| jdbc.driverClass= oracle.jdbc.driver.OracleDriver  jdbc.jdbcUrl= jdbc:oracle:thin:@localhost:1521:orcl  jdbc.user = scott  jdbc.password = tiger |

### 在xml配置文件中读取db.properties文件

|  |
| --- |
| **<context:property-placeholder** location="classpath:db.properties" **/>**  **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**  **<property** name="user" value="${jdbc.user}"**></property>**  **<property** name="password" value="${jdbc.password}"**></property>**  **<property** name="driverClass" value="${jdbc.driverClass}"**></property>**  **<property** name="jdbcUrl" value="${jdbc.jdbcUrl}"**></property>**  **</bean>** |

## Spring 事务管理

### 1.Spring事务管理api

*需求：*

*-- 7499Allen员工和7521Ward转账*

*-- sal本身是工资薪水的意思 假设做为银行账号存款*

*-- 7499 转600到7521账户里面*

*-- 所以最终的结果是Allen 少600 Ward多600*

#### 1 spring事务管理两种方式

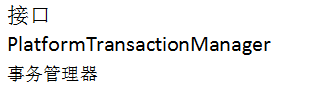
第一种 编程式事务管理（不用）

第二种 声明式事务管理

基于xml配置文件实现

基于注解实现

#### spring事务管理的api介绍

****

1. **spring针对不同的dao层框架，提供接口不同的实现类**

****

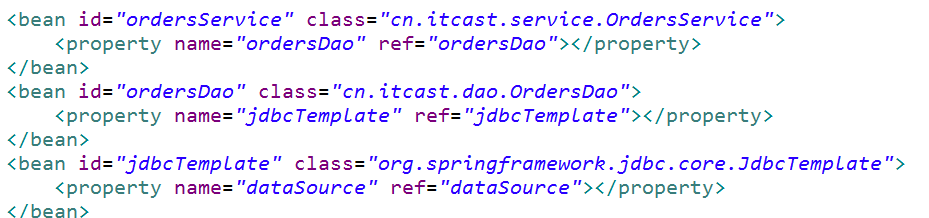
**（2）首先 配置事务管理器**

### 搭建转账环境

1 创建数据库表，添加数据



2 创建service和dao类，完成注入关系



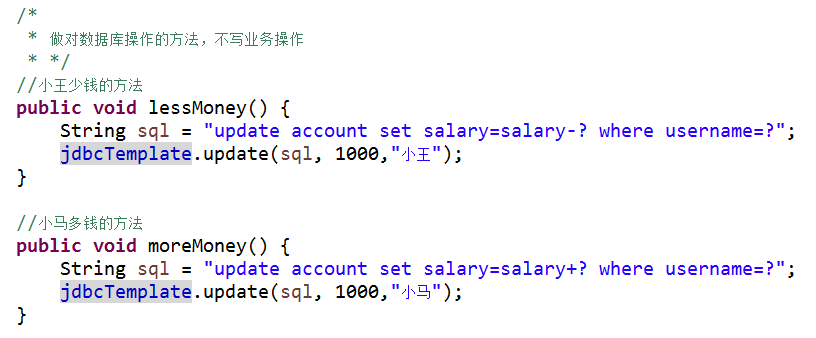
（1）service层又叫业务逻辑层

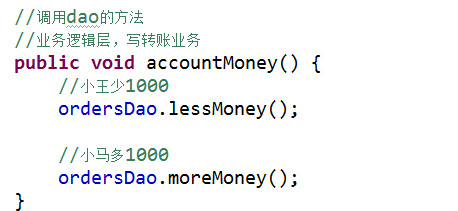
（2）dao层，单纯对数据库操作层，在dao层不添加业务

（3）需求：小王 转账 1000 给 小马

- 小王少1000

- 小马多1000

****

****

3 产生问题：

（1）如果小王少了1000之后，出现异常，小马不会多1000，钱丢失了

4 解决：

（1）添加事务解决，出现异常进行回滚操作

### 声明式事务管理（xml配置）

|  |
| --- |
| **1 配置文件方式使用aop思想配置**  **第一步 配置事务管理器**    **第二步 配置事务增强**    **第三步 配置切面** |

### 声明式事务管理（注解）

|  |
| --- |
| **第一步 配置事务管理器**    **第二步 配置事务注解**    **第三步 在要使用事务的方法所在类上面添加注解** |