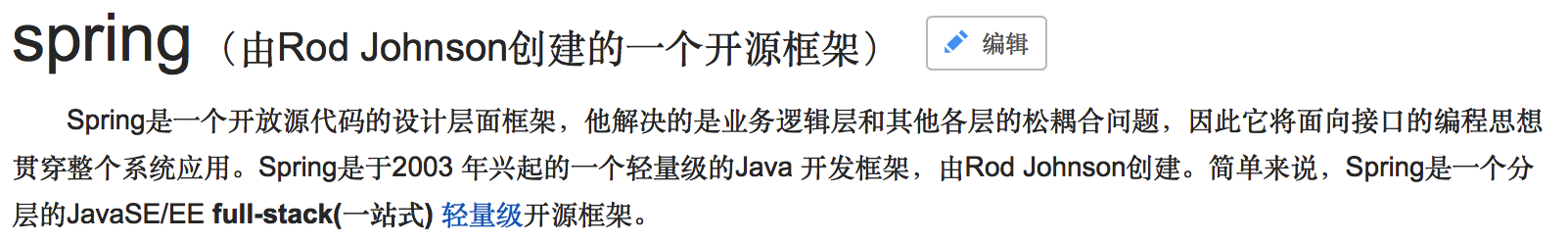
# Spring概述

## 什么是Spring



Spring是一个开源框架，Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson在其著作 Expert One-On-One J2EE Development and Design 中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为J2EE应用程序开发提供集成的框架。Spring使用基本的 JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。 Spring的 核心是控制反转(IoC)和面向切面(AOP) 。简单来说， Spring是一个分层的JavaSE/EEfull-stack(一站式)轻量级开源框架。

**EE开发分成三层结构**

* WEB层: Spring MVC
* 业务层: Bean管理:(IOC)
* 持久层: Spring的JDBC模板。ORM模板用于整合其他的持久层框架

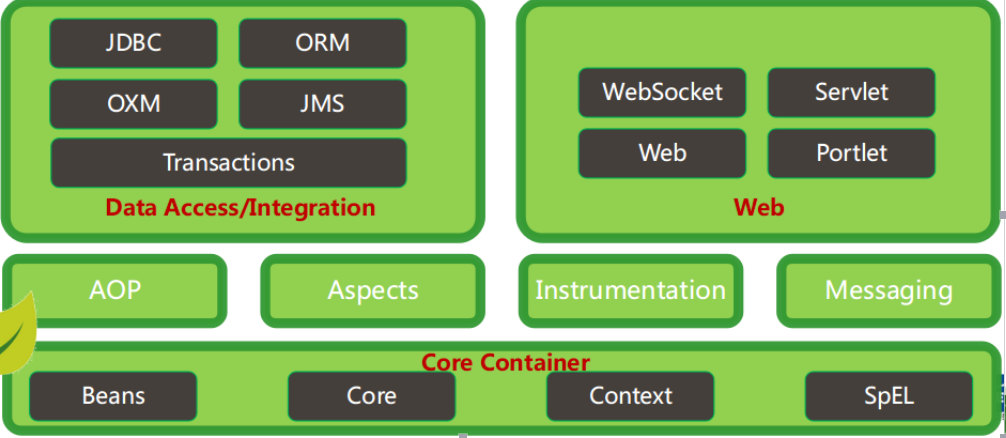
Expert One-to-One J2EE Design and Development : J2EE 的设计和开发:(2002.EJB)

Expert One-to-One J2EE Development without EJB : J2EE 不使用EJB的开发

## Spring的版本

Spring3.X 和 Spring4.X 和 Spring5.X

## spring功能模块划分



# JAVA类的耦合与解耦

## 什么是程序的耦合

在开发中，可能会写很多的类，而有些类之间不可避免的产生依赖关系，这种依赖关系称之为耦合。  
有些依赖关系是必须的，有些依赖关系可以通过优化代码来解除的。

**代码示例**

public class CustomerServiceImpl implements CustomerService {

CustomerDao cusomerDao = new CustomerDaoImpl();

}

以上的代码表示：业务层调用持久层，并且此时业务层在依赖持久层的接口和实现类。如果此时没有持久层实现类，编译将不能通过。这种依赖关系就是我们可以通过优化代码解决的。

**还有如下面的代码：**

我们的类依赖了MySQL的具体驱动类，如果这时候因为某些原因数据库的品牌从MySQL改为Oracle，那么需要通过改源码来修改数据库驱动。这显然不是我们想要的。

public class JdbcDemo01{

public static void main(String[] args) throws Exception {

Class.forName("com.mysql.jdbc.Driver");

//new com.mysql.jdbc.Driver()

}}

## 解决耦合的思路

当是我们学习JDBC时，是通过反射来注册驱动的，代码如下：

Class.forName("com.mysql.jdbc.Driver");

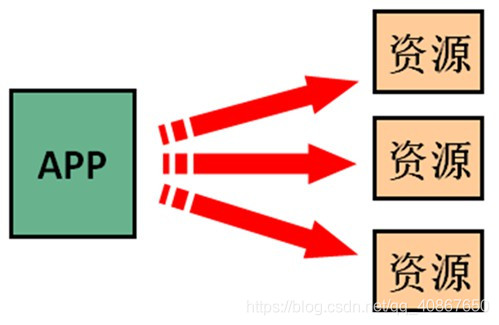
这时的好处是，我们的类中不再依赖具体的驱动类，此时就算删除mysql的驱动jar包，依然可以编译。但是因为没有驱动类，所以不能运行。  
不过，此处也有个问题，就是我们反射类对象的全限定类名字符串是在java类中写死的，一旦要改还是要修改源码。  
解决这个问题也很简单，使用配置文件配置。

### 工厂模式解耦

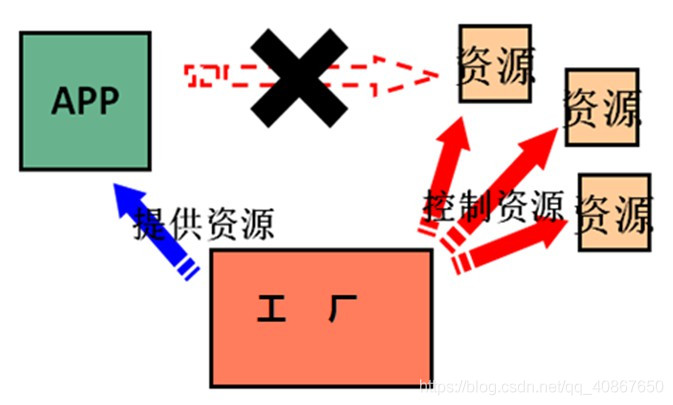
在实际开发中我们可以把所有的dao和service和controller对象使用配置文件配置起来，当启动服务器应用加载的时候，通过读取配置文件，把这些对象创建出来并存起来。在接下来的使用的时候，直接拿过来用就好了。

### 控制反转-Inversion Of Control

上面解耦的思路有2个问题：  
**1、存哪去？**  
分析：由于我们是很多对象，肯定要找个集合来存。这时候有Map和List供选择。  
到底选Map还是List就看我们有没有查找需求。有查找需求，选Map。  
所以我们的答案就是  
在应用加载时，创建一个Map，用于存放controller，Service和dao对象。  
我们把这个map称之为容器。  
**2、还是没解释什么是工厂？**  
工厂就是负责给我们从容器中获取指定对象的类。这时候我们获取对象的方式发生了改变。  
原来：  
我们在获取对象时，都是采用new的方式。**是主动的。**



现在：  
我们获取对象时，同时跟工厂要，有工厂为我们查找或者创建对象。是被动的。



这种被动接收的方式获取对象的思想就是控制反转，它是spring框架的核心之一。  
它的作用只有一个：削减计算机程序的耦合。

# Spring的 IOC

## 什么是IoC

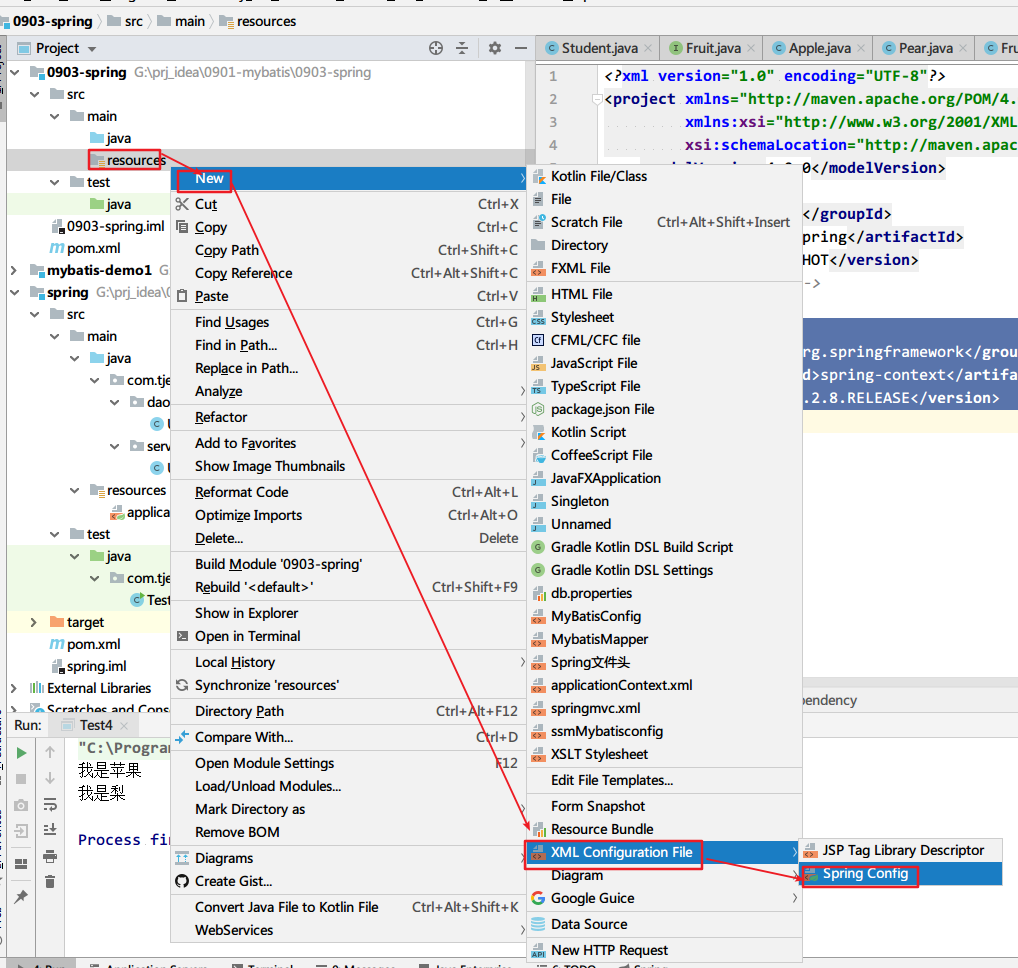
Inversion of Control:控制权的转移,创建对象的权利由应用程序转移到容器称为控制反转

## IoC的作用

削减计算机程序的耦合(解除我们代码中的依赖关系）

## 基于XML的IoC配置入门

创建Maven项目，resources 创建spring配置文件



pom.xml依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>5.3.8</version> </dependency> |

编写UserService类

|  |
| --- |
| public class UserService {  public UserService() {  System.*out*.println("UserService()......");  }   public void add() {  System.*out*.println("UserService.add()......");  } } |

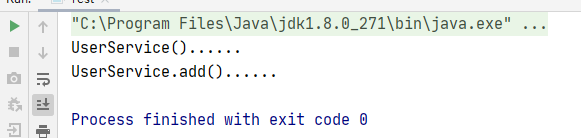
编写applicationContext.xml配置文件

|  |
| --- |
| *<?*xml version="1.0" encoding="UTF-8"*?>* <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  *<!--告诉spring容器 创建UserService实例对象-->* <bean id="userService" class="com.tjetc.service.UserService"></bean> </beans> |

编写测试类Test

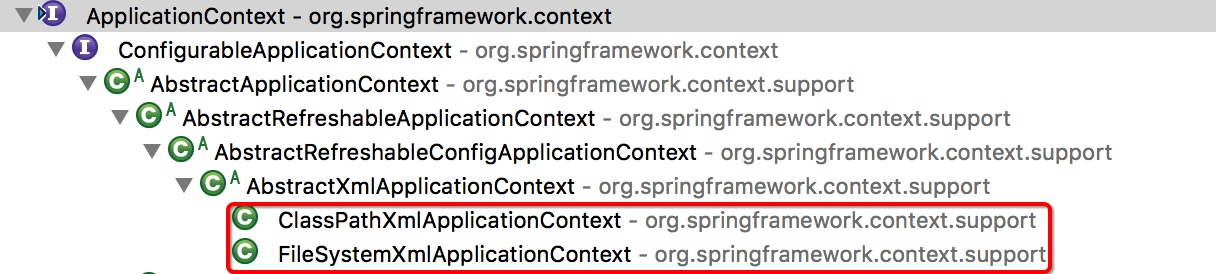
|  |
| --- |
| public class Test {  public static void main(String[] args) {  *//实例化ClassPathXmlApplicationContext对象，配置文件作为参数* ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  *//上线文对象 getBean方法获取配置bean对象UserService* UserService userService = context.getBean(UserService.class);  *//调用UserService的add方法* userService.add();  } } |

结果：



## ApplicationContext

ApplicatioContext接口有两个基本的实现类:



ClassPathXmlApplicationContext : 加载类路径下的Spring配置文件。

FileSystemXmlApplicationContext : 加载本地磁盘下的Spring配置文件。

# Bean的配置

## Spring中的bean元素相关配置

配置文件中的bean元素用于描述需要Spring容器管理的对象。其 class属性 用于指被管理对象的完整类名。

### id属性和name属性

id属性 是为Spring容器管理的对象起个名字。其使用ID约束：

唯一

必须以字母开始

可以使用字母、数字、连字符、下划线、句号、冒号。但 不能出现特殊字符

name属性 也是为被Spring容器管理的对象起个名字。这样后续可以通过该名字从容器中获取对象。

没有ID中的那些约束（可以重复[不推荐]、可以出现特殊字符）

如果<bean>没有id的话，name可以当做id使用

**举例**

<bean id="bookAction">

<bean name="/loginAction" >

|  |  |
| --- | --- |
|  | 此处的名称中由于有特殊字符，只能使用name属性。 |

结论：在定义bean的时候，推荐使用name属性 来为bean指定名称

### scope属性

用于声明bean在Spring容器中的作用范围。其取值包括：

**singleton** : 默认值 单例的；默认在Spring容器启动的时候就会创建该实例。

**prototype** : 多例的;默认在从Spring容器中获取bean时才会创建该实例。

request : WEB项目中，Spring创建一个Bean的对象，将对象存入到request域中。

session : WEB项目中，Spring创建一个Bean的对象，将对象存入到session域中。

globalSession : WEB项目中，应用在Porlet环境。如果没有Porlet环境，那么globalSession相当于session。

### 生命周期属性

在bean被创建的时候，需要执行一些初始化的逻辑。可以指定bean中的一个方法为其初始化方法。这样Spring在创建完该对象之后，立即调用一下该初始化方法。

|  |
| --- |
| public void init() {  System.*out*.println("UserService.init()......"); } |

<bean id="userService" class="com.tjetc.service.UserService" init-method="init" ></bean>

在bean对象销毁的时候，需要执行一些销毁逻辑。可以指定bean中的一个方法为其销毁方法。这样Spring容器在关闭之前并且销毁该对象的时候，会调用一下该销毁方法。

public void destroy() {  
 System.*out*.println("UserService.destroy().......");  
}

<bean id="userService" class="com.tjetc.service.UserService" destroy-method="destroy"></bean>

注意，此处的销毁方法必须是在容器正常关闭（即执行close方法）时，才会被执行。

# Spring注解方式的Bean管理

主要解决的问题，在XML中一个一个对bean进行配置，开发效率也不高。

目的:

* 自动发现哪些对象需要让Spring进行管理，自动将其注册到Spring容器中。
* 自动注入bean依赖的内容。

## Spring Bean管理中的常用注解

### 准备工作

**使用注解要先开启注解扫描的功能**

*<!--配置组件的扫描com.tjetc本包及其子孙包下的所有的在类上标注有@Controller,@Service,@Repository,@Component注解的类,  
spring会把标注了这些注解的类当做你配置bean节点一样纳入spring容器管理-->*

|  |
| --- |
| <**context:component-scan base-package="com.tjetc"**/> |

### @Component

标注在类上，说明这是一个Spring组件。Spring中目前还提供了与 @Component 功能一致的其它三个衍生注解：

* @Controller :WEB 层（控制层）
* @Service :业务层
* @Repository :持久层

|  |  |  |
| --- | --- | --- |
|  |  | 衍生的三个注解是为了让标注类本身的用途清晰，Spring在后续版本会对其增强  作用和XML配置文件的<bean>标签的实现功能一致； |

### 属性注入的注解

使用注解注入的方式，可以不用提供set方法。

**相关注解**

* **@Value : 用于注入普通类型和String类型。**
  + **加在成员变量上——原理是反射赋值（破坏封装）**
  + **加在set方法上——原理是方法调用（推荐使用）**
  + **它可以使用spring中的spEL(也就是spring的el表达式)，SpEl的写法：${表达式}**

@Value使用

|  |
| --- |
| @Component(**"constant"**) **public class** AdminConstant {  *//spEL表达式* @Value(**"${image.baseUrl}"**)  **private** String **imageBaseUrl**;  **private** String **imageTypes**;  @Value(**"测试@Value"**)  **private** String **name**;   @Value(**"${image.types}"**)  **public void** setImageTypes(String imageTypes) {  **this**.**imageTypes** = imageTypes;  }   **public** String getImageTypes() {  **return imageTypes**;  }   **public** String getImageBaseUrl() {  **return imageBaseUrl**;  }   **public void** setImageBaseUrl(String imageBaseUrl) {  **this**.**imageBaseUrl** = imageBaseUrl;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   @Override  **public** String toString() {  **return "AdminConstant{"** +  **"imageBaseUrl='"** + **imageBaseUrl** + **'\''** +  **", imageTypes='"** + **imageTypes** + **'\''** +  **", name='"** + **name** + **'\''** +  **'}'**;  } } |

applicationContext.xml配置

|  |
| --- |
| <**context:property-placeholder location="classpath\*:admin.properties"**></**context:property-placeholder**> |

测试

|  |
| --- |
| **public class** Test2 {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  AdminConstant constant = (AdminConstant) context.getBean(**"constant"**);  System.***out***.println(constant);  } } |

* **@Autowired : 用于注入对象类型（自动装配）**
  + **默认按类型进行装配**
  + **按名称注入: @Qualifier——强制使用名称注入，一般不能单独使用和Autowired配合使用，当注入的对象类型有多个时，@Autowoired必须与@Qualifier一起使用**

@Autowoired和@Qualifier使用

|  |
| --- |
| **public class** User {  **private** Long **id**;  **public void** setId(Long id) {  **this**.**id** = id;  }  @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **'}'**;  } } |

applicationContext.xml配置

|  |
| --- |
| <**bean id="user1" class="com.tjetc.entity.User"**>  <**property name="id" value="1"**></**property**> </**bean**> <**bean id="user2" class="com.tjetc.entity.User"**>  <**property name="id" value="2"**></**property**> </**bean**> |

PrintUser.java

|  |
| --- |
| @Component **public class** PrintUser {  @Autowired  @Qualifier(**"user1"**)  **private** User **user**;  **public void** print(){  System.***out***.println(**user**);  } } |

测试

|  |
| --- |
| **public class** TestPrintUser {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  PrintUser printUser = (PrintUser) context.getBean(**"printUser"**);  printUser.print();  } } |

* **@Resource : 相当于@Autowired和@Qualifier一起使用**

|  |
| --- |
| @Component **public class** PrintUser {  @Resource(name = **"user2"**)  **private** User **user**;   **public void** print(){  System.***out***.println(**user**);  } } |

测试：

|  |
| --- |
| **public class** TestPrintUser {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  PrintUser printUser = (PrintUser) context.getBean(**"printUser"**);  printUser.print();  } } |

### Bean的作用范围的注解

**注解@Scope**

* singleton : 单例
* prototype : 多例

|  |
| --- |
| @Service @Scope(**"prototype"**) **public class** ProductServiceImpl {  **public** ProductServiceImpl(){  System.***out***.println(**"ProductServiceImpl()构造方法....."**);  }  } |

测试

|  |
| --- |
| **public class** TestProductService {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  ProductServiceImpl productService1 = context.getBean(ProductServiceImpl.**class**);  ProductServiceImpl productService2= context.getBean(ProductServiceImpl.**class**);  System.***out***.println(productService1==productService2);  } } |

### Bean的生命周期的配置

**相关注解**

* @PostConstruct : 相当于 init-method
* @PreDestroy : 相当于 destroy-method

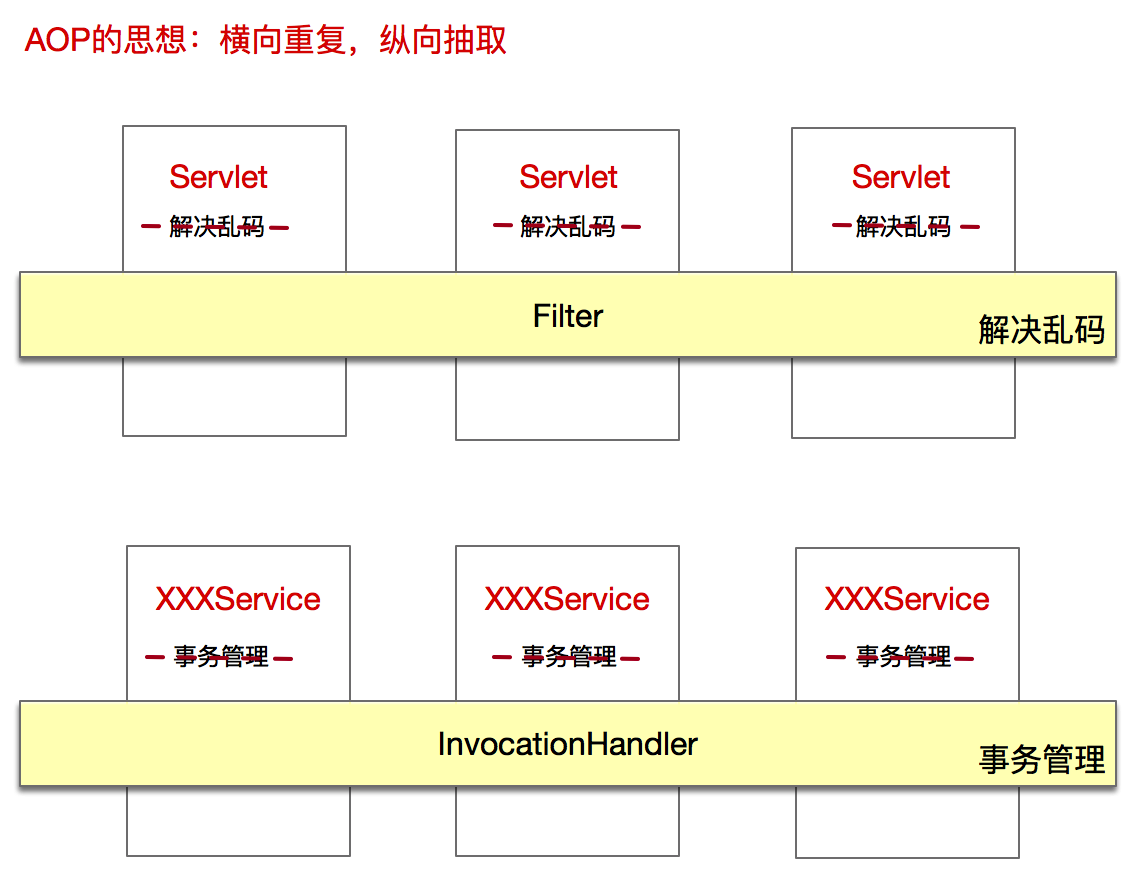
|  |
| --- |
| @Service **public class** ProductServiceImpl {  **public** ProductServiceImpl(){  System.***out***.println(**"ProductServiceImpl()构造方法....."**);  }   @PostConstruct  **public void** init(){  System.***out***.println(**"ProductServiceImpl.init()....."**);  }   @PreDestroy  **public void** destroy(){  System.***out***.println(**"ProductServiceImpl.destroy()......"**);  } } |

测试

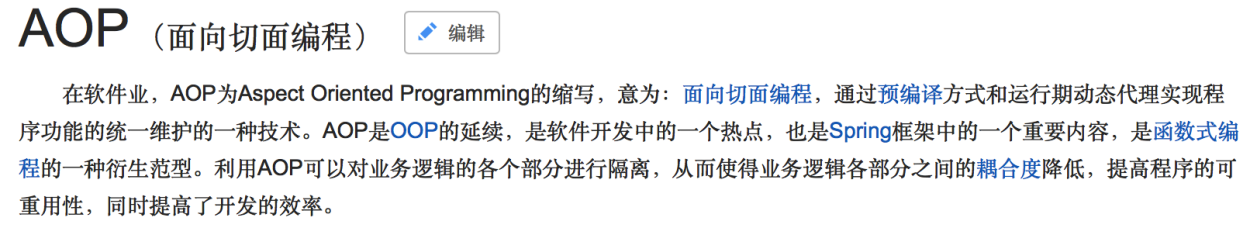
|  |
| --- |
| **public class** TestProductService {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  ProductServiceImpl productService = context.getBean(ProductServiceImpl.**class**);  context.close();} } |

# AOP的概述

Aop思想



## 什么是AOP



**Spring是解决实际开发中的一些问题:**

* AOP解决OOP中遇到的一些问题。是OOP的延续和扩展。
* Spring引入AOP思想来为容器中管理的对象动态生成代理对象。

## 为什么学习AOP

**AOP的常用应用场景**

* 权限校验
* 日志记录
* 性能监控
* 事务控制

# Spring AOP基于Annotation实现

## AOP开发中的相关术语

**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。

**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,即切入点+通知

## 基于注解方式声明切面

### pom依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.aspectj</**groupId**>  <**artifactId**>aspectjweaver</**artifactId**>  <**version**>1.9.7</**version**> </**dependency**> |

### 启动@Aspect注解的支持和注解的扫描机制

applicationContext.xml配置文件用以下配置:

|  |
| --- |
| <!-- 开启对@Aspect注解的支持 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

### 写切面类

@Aspect声明的类，和普通类一样可以添加属性和方法，还可以包含切入点、通知、引入。

 Pointcut声明: 切入点声明包含两个部分：

 1.签名：由一个名字和多个参数组成,必须返回void, 如:**private** **void** anyMethod(){}

 2.切入点表达式  如:@Pointcut("execution(\* com.tjetc.service..\*.\*(..))")

|  |
| --- |
| @Component *//把该类纳入到spring容器中管理* @Aspect *//代表该类是一个切面类* **public class** AnnotationTransactionPrint {  *//声明切入点，切入点表达式（只对OrderServiceImpl这个类所有的方法进行aop）* @Pointcut(**"execution(\* com.tjetc.service.impl.OrderServiceImpl.\*(..))"**)  *//切入点签名：由一个名字和多个参数组成,必须返回void* **private void** anyMethod() {  }   *//前置通知，使用切入点签名* @Before(**"anyMethod()"**)  **public void** doBefore() {  System.***out***.println(**"前置通知...."**);  }   *//后置通知，使用切入点签名* @AfterReturning(**"anyMethod()"**)  **public void** doAfterReturning() {  System.***out***.println(**"后置通知....."**);  }   *//异常通知，使用切入点签名* @AfterThrowing(**"anyMethod()"**)  **public void** doAfterThrowing() {  System.***out***.println(**"异常通知...."**);  }   *//最终通知，使用切入点签名* @After(**"anyMethod()"**)  **public void** doAfter() {  System.***out***.println(**"最终通知......."**);  }   *//环绕通知，使用切入点签名  /\*@Around("anyMethod()")\*/* **public** Object doRound(ProceedingJoinPoint pjp) **throws** Throwable {  **try** {  System.***out***.println(**"环绕前置通知....."**);  Object result = pjp.proceed();  System.***out***.println(**"环绕后置通知....."**);  **return** result;  } **catch** (Exception e) {  System.***out***.println(**"环绕异常通知......"**);  **throw** e;  } **finally** {  System.***out***.println(**"环绕最终通知...."**);  }  } } |

### 写一个业务类(被代理的类)

|  |
| --- |
| @Service *//纳入spring的管理* **public class** OrderServiceImpl {  **public void** add() {  System.***out***.println(**"PersonService.add()..."**);  }  **public void** update() {  System.***out***.println(**"PersonService.update()..."**);  }  **public void** del() {  System.***out***.println(**"PersonService.del()..."**);  **throw new** RuntimeException(**"出错了...."**);  } } |

### 测试类

|  |
| --- |
| **public class** TestAnnotationAop {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  OrderServiceImpl orderService = context.getBean(OrderServiceImpl.**class**);  orderService.add();  orderService.update();  orderService.del();  } } |

开启环绕通知配置，注释前置通知、后置通知、异常通知、最终通知，进行测试