## 1.1 什么是Spring

* Spring 是分层的JavaSE/EE full-stack 轻量级开源框架
* 以Ioc (Inverse of Control 反转控制)和AOP(Aspect Oriented Programming 面向切面编程为内核)
* 服务器端: 表示层(web), 业务逻辑层(service), 持久层(dao)
* 全站式开源框架: 为每一个层都提供技术支持

web: struts, Spring MVC

service: Spring, 常用于service管理事务,记录日志等,不仅仅是在service 层.

dao: hibernate, JdbcTemplate

* 轻量级: 相对于 EJB(企业级JavaBean)

AOP

    面向切面编程，AOP让开发人员创建非行为性的关注点，并将它们 可以分解成方面并应用到域对象上，同时不会增加域对象的对象模型 的复杂性。

IoC

IoC，用白话来讲，就是由容器控制程序之间的关系，而非传统实现中， 由程序代码直接操控。这也就是所谓“控制反转”的概念所在：控制权 由应用代码中转到了外部容器，控制权的转移，是所谓反转。

应用代码控制程序与程序之间的关系,但控制权转移到了Spring外部容 器,这就是反转控制(Inversion of Control)

## 用途

     对于Spring的用途很多，但是Spring的核心和经典我们可以简化为三部分：

**1、IoC容器可以将对象之间的依赖关系交由Spring管理，进行控制。**

2、AOP方便进行面向切面编程，是oop的扩展，想加什么功能直接加。

3、能够集成各种优秀的框架，struts、Hibernate等等，还有很多，不再一 一罗列。

## Spring框架是一种非侵入式的轻量级框架

1.非侵入式的技术体现

允许在应用系统中自由选择和组装Spring框架的各个功能模块，并且不强制要求应用系统的类必须从Spring框架的系统API的某个类来继承或者实现某个接口。

2.如何实现非侵入式的设计目标的

 1）应用反射机制，通过动态调用的方式来提供各方面的功能，建立核心组间BeanFactory

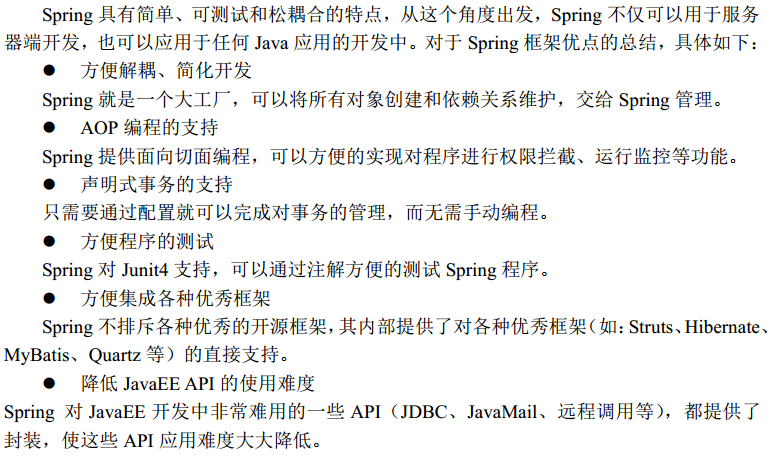
 2）配合使用Spring框架中的BeanWrapper和BeanFactory组件类最终达到对象的实例创建和属性注入

 3）优点：允许所开发出来的应用系统能够在不用的环境中自由移植，不需要修改应用系统中的核心功能实现的代码

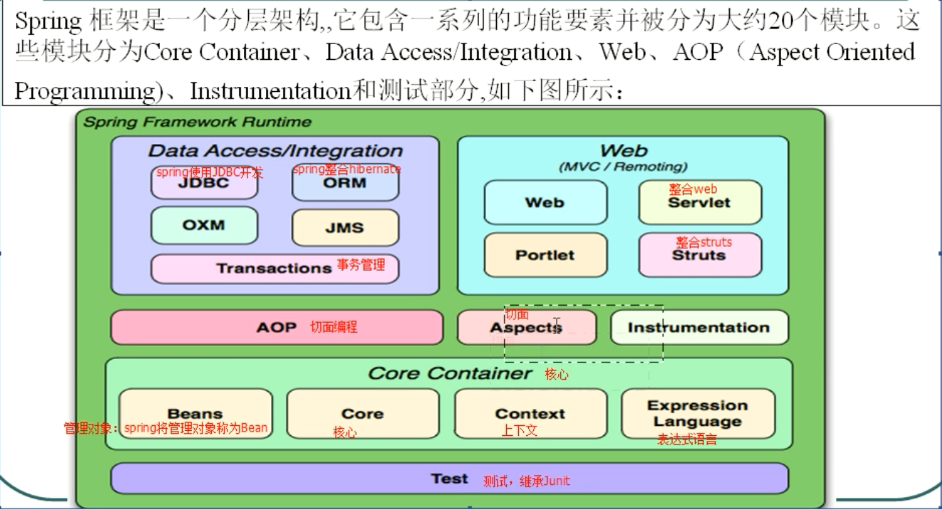
**4**.**轻量级的框架**，轻量级的意思：轻量级框架是相对于重量级框架的一种设计模式：轻量级框架不带有侵略性API,对容器也没有依赖性,易于进行配置,易于通用,启动时间较短。这是轻量级框架相对于重量级框架的优势。

所谓轻量级就是[spring框架](https://www.baidu.com/s?wd=spring%E6%A1%86%E6%9E%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWnHK9P1nkPvDdrjcYuyRY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1bsrHDYrHcY)在[系统初始化](https://www.baidu.com/s?wd=%E7%B3%BB%E7%BB%9F%E5%88%9D%E5%A7%8B%E5%8C%96&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWnHK9P1nkPvDdrjcYuyRY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1bsrHDYrHcY)的时候不用加载所有的服务，为系统节约了资源！  
而EJB框架就是重量级的，每次初始化都必须加载所有的服务！

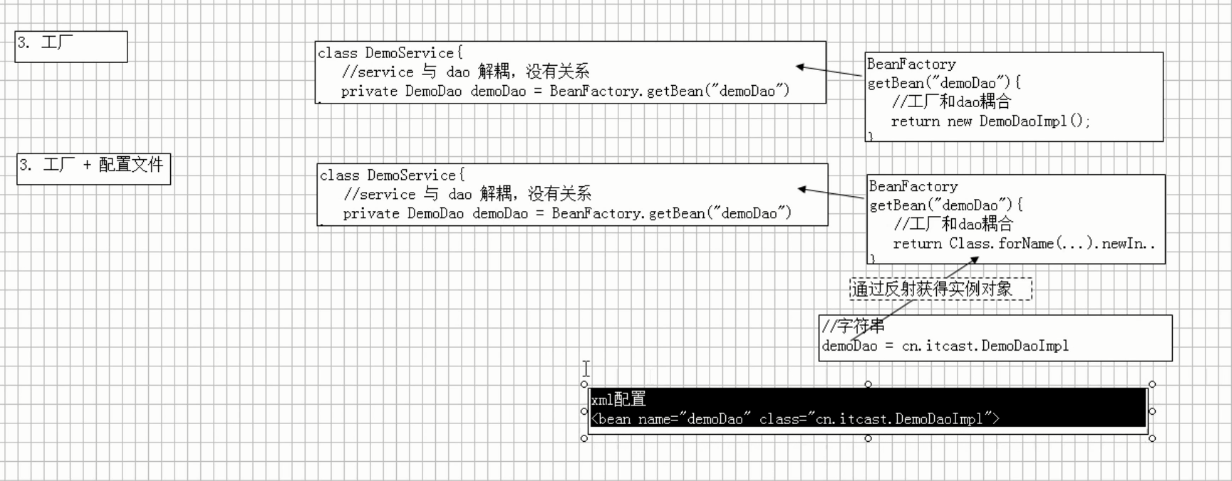
## 1.2 Spring的好处



## 1.3



## 1.4 IOC思想(Inverse of Control)



## 3.4 api使用

public static void main(String[] args){

String xmlPath = “applicationContext.xml”;

//解析配置文件

//AppliationContext在加载配置文件时,对Bean进行实例化

ApplicationContext applicationContext = new ClassPathXmlApplicationCo ntext(xmlPath);

/\*\*其中getBean()方法是单例的,如果new多个相同xmlId,获得的对象其 实是同一个对象,其对象指向的内存地址是相同的,要想getBean()变成多 例获得不同的对象,需要在xml文件的<Bean>标签里写入scope=”prototyp e”

\*/

//获取配置文件中,配置对象的实例

ClassName(类名) name = applicationContext.getBean(“xml的ID”);

//这样就获取ClassName的对象了

name.getMethod();

}



## BeanFactory 和 ApplicationContext



* BeanFactory 采取延迟加载,第一次getBean时才会初始化Bean
* ApplicationContext是对BeanFactory扩展,提供了更多功能,

国际化处理

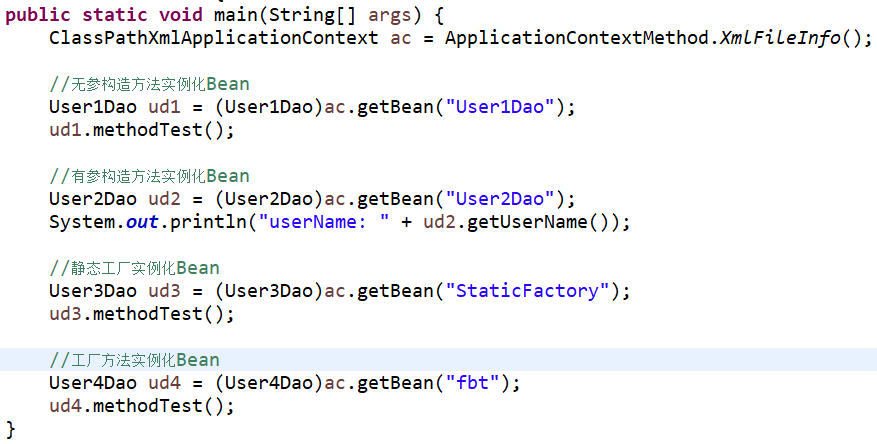
事件传递

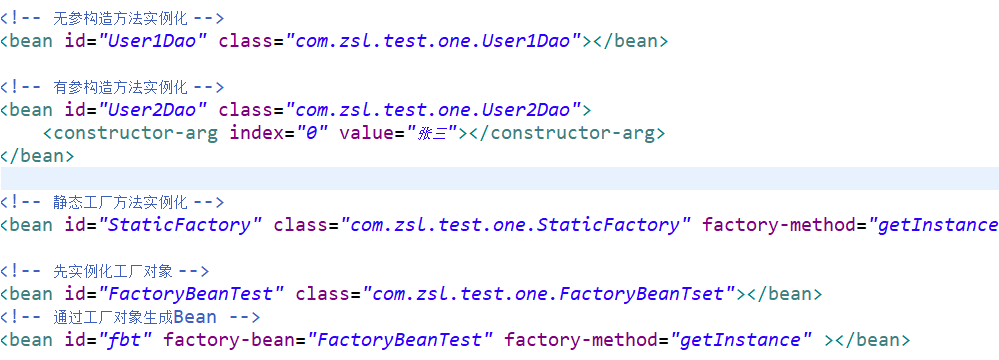
Bean自动装配

各种不同应用层的Context实现



## Spring获取bean的四种方式





<!-- 配置工厂 -->

<bean id=”” class=””></bean>

## 依赖注入

spring有多种依赖注入的形式，下面仅介绍spring通过xml进行IOC配置的方式：

* Set注入

这是最简单的注入方式，假设有一个SpringAction，类中需要实例化一个SpringDao对象，那么就可以定义一个private的SpringDao成员变量，然后创建SpringDao的set方法（这是ioc的注入入口）：

**Java代码  [IMG_256](http://blessht.iteye.com/blog/javascript:void())**

1. **package** com.bless.springdemo.action;
2. **public** **class** SpringAction {
3. //注入对象springDao
4. **private** SpringDao springDao;
5. //一定要写被注入对象的set方法
6. **public** **void** setSpringDao(SpringDao springDao) {
7. **this**.springDao = springDao;
8. }
10. **public** **void** ok(){
11. springDao.ok();
12. }
13. }

随后编写spring的xml文件，<bean>中的name属性是class属性的一个标识名，class属性指类的全名，因为在SpringAction中有一个公共属性Springdao，所以要在<bean>标签中创建一个<property>标签指定SpringDao。<property>标签中的name就是SpringAction类中的SpringDao属性名，ref指下面<bean name="springDao"...>，这样其实是spring将SpringDaoImpl对象实例化并且调用SpringAction的setSpringDao方法将SpringDao注入：

**Java代码  [IMG_257](http://blessht.iteye.com/blog/javascript:void())**

1. <!--配置bean,配置后该类由spring管理-->
2. <bean name="springAction" **class**="com.bless.springdemo.action.SpringAction">
3. <!--(1)依赖注入,配置当前类中相应的属性-->
4. <property name="springDao" ref="springDao"></property>
5. </bean>
6. <bean name="springDao" **class**="com.bless.springdemo.dao.impl.SpringDaoImpl"></bean>

构造器注入

这种方式的注入是指带有参数的构造函数注入，看下面的例子，我创建了两个成员变量SpringDao和User，但是并未设置对象的set方法，所以就不能支持第一种注入方式，这里的注入方式是在SpringAction的构造函数中注入，也就是说在创建SpringAction对象时要将SpringDao和User两个参数值传进来：

**Java代码  [IMG_258](http://blessht.iteye.com/blog/javascript:void())**

1. **public** **class** SpringAction {
2. //注入对象springDao
3. **private** SpringDao springDao;
4. **private** User user;
6. **public** SpringAction(SpringDao springDao,User user){
7. **this**.springDao = springDao;
8. **this**.user = user;
9. System.out.println("构造方法调用springDao和user");
10. }
12. **public** **void** save(){
13. user.setName("卡卡");
14. springDao.save(user);
15. }
16. }

在XML文件中同样不用<property>的形式，而是使用<constructor-arg>标签，ref属性同样指向其它<bean>标签的name属性：

**Xml代码  [IMG_259](http://blessht.iteye.com/blog/javascript:void())**

1. <!--配置bean,配置后该类由spring管理-->
2. **<bean** name="springAction" class="com.bless.springdemo.action.SpringAction"**>**
3. <!--(2)创建构造器注入,如果主类有带参的构造方法则需添加此配置-->
4. **<constructor-arg** ref="springDao"**></constructor-arg>**
5. **<constructor-arg** ref="user"**></constructor-arg>**
6. **</bean>**
7. **<bean** name="springDao" class="com.bless.springdemo.dao.impl.SpringDaoImpl"**></bean>**
8. **<bean** name="user" class="com.bless.springdemo.vo.User"**></bean>**

  解决构造方法参数的不确定性，你可能会遇到构造方法传入的两参数都是同类型的，为了分清哪个该赋对应值，则需要进行一些小处理：

下面是设置index，就是参数位置：

**Xml代码  [IMG_260](http://blessht.iteye.com/blog/javascript:void())**

1. **<bean** name="springAction" class="com.bless.springdemo.action.SpringAction"**>**
2. **<constructor-arg** index="0" ref="springDao"**></constructor-arg>**
3. **<constructor-arg** index="1" ref="user"**></constructor-arg>**
4. **</bean>**

  另一种是设置参数类型：

**Xml代码  [IMG_261](http://blessht.iteye.com/blog/javascript:void())**

1. **<constructor-arg** type="java.lang.String" ref=""**/>**

一、Bean的定义

        <beans…/>元素是Spring配置文件的根元素，<bean…/>元素师<beans../>元素的子元素，<beans…/>元素可以包含多个<bean…/>子元素，每个<bean…/>元素可以定义一个Bean实例，每一个Bean对应Spring容器里的一个Java实例定义Bean时通常需要指定两个属性。

         Id：确定该Bean的唯一标识符，容器对Bean管理、访问、以及该Bean的依赖关系，都通过该属性完成。Bean的id属性在Spring容器中是唯一的。

        Class：指定该Bean的具体实现类。注意这里不能使接口。通常情况下，Spring会直接使用new关键字创建该Bean的实例，因此，这里必须提供Bean实现类的类名。

        下面是定义一个Bean的简单配置

**[html]** [view plain](http://blog.csdn.net/chenssy/article/details/8222744) [copy](http://blog.csdn.net/chenssy/article/details/8222744)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns="http://www.springframework.org/schema/beans"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"**>**
6. <!-- 定义第一个Bean实例：bean1 -->
7. **<bean** id="bean1" class="com.Bean1" **/>**
9. <!-- 定义第二个Bean实例：bean2 -->
10. **<bean** id="bean2" class="com.Bean2" **/>**
12. **</bean>**

        Spring容器集中管理Bean的实例化，Bean实例可以通过BeanFactory的getBean(Stringbeanid)方法得到。BeanFactory是一个工厂，程序只需要获取BeanFactory引用，即可获得Spring容器管理全部实例的引用。程序不需要与具体实例的实现过程耦合。大部分Java EE应用里，应用在启动时，会自动创建Spring容器，组件之间直接以依赖注入的方式耦合，甚至无须主动访问Spring容器本身。

Bean指定特定的作用域。

        Spring支持5种作用域：

           Singleton：单例模式。在整个SpringIoC容器中，使用singleton定义的Bean将只有一个实例。

           Prototype：原型模式。每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例。

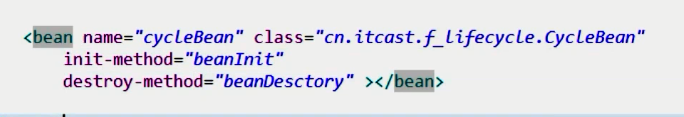
           request：对于每次HTTP请求，使用request定义的Bean都将产生一个新的实例，即每次HTTP请求都会产生不同的Bean实例。当然只有在WEB应用中使用Spring时，该作用域才真正有效。

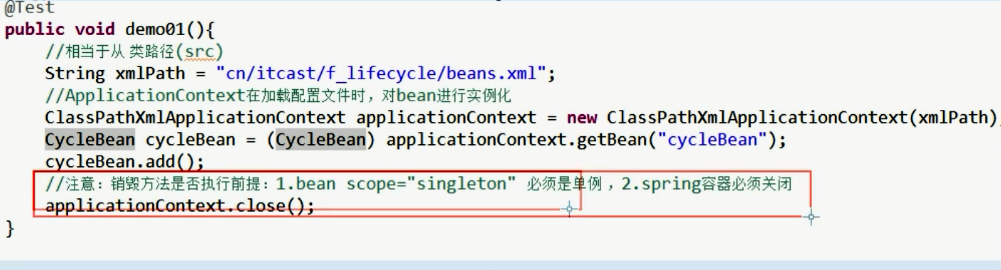
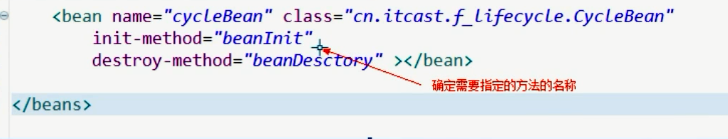
        session：对于每次HTTPSession，使用session定义的Bean都将产生一个新的实例时，即每次HTTP Session都将产生不同的Bean实例。同HTTP一样，只有在WEB应用才会有效。

        global session：每个全局的HTTPSession对应一个Bean实例。仅在portlet Context的时候才有效。

## Bean Control LifeCycle(Bean生命周期)







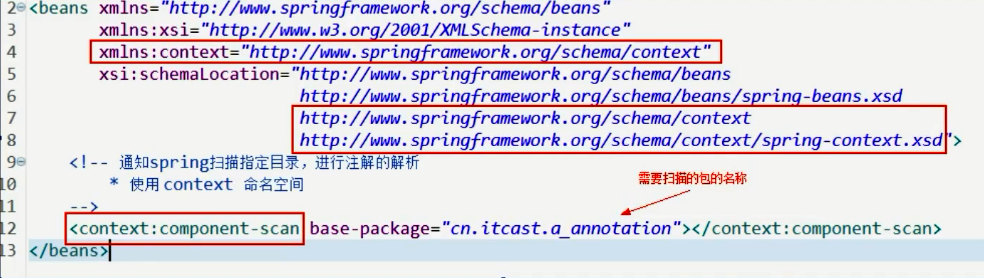
## 装备Bean基于注解,注解(annotation)

传统的Spring做法是使用.xml文件来对bean进行注入或者是配置aop、事物，这么做有两个缺点：  
1、如果所有的内容都配置在.xml文件中，那么.xml文件将会十分庞大；如果按需求分开.xml文件，那么.xml文件又会非常多。总之这将导致配置文件的可读性与可维护性变得很低。  
2、在开发中在.java文件和.xml文件之间不断切换，是一件麻烦的事，同时这种思维上的不连贯也会降低开发的效率。  
为了解决这两个问题，Spring引入了注解，通过"@XXX"的方式，让注解与Java Bean紧密结合，既大大减少了配置文件的体积，又增加了Java Bean的可读性与内聚性。

使用注解代替xml配置文件,简化操作.

常见开发使用方式: 注解 + xml文件配置

Spring提供@component 用于配置Bean,取代了<bean id=”” class=””>



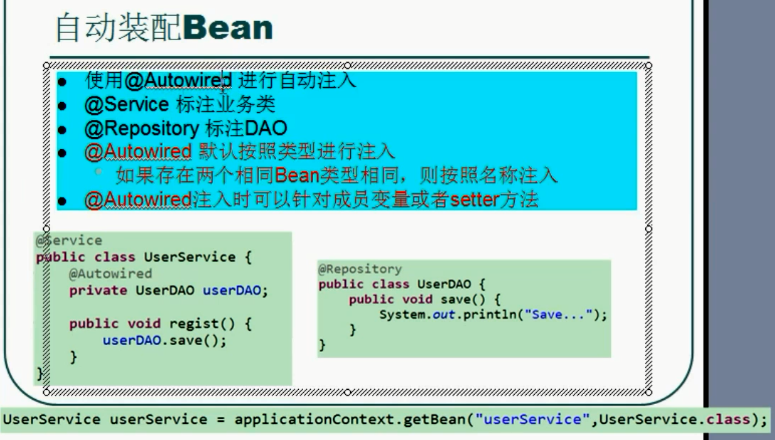
除了@Component外, Spring提供了3个功能基本和@Component等效的注解

* Repository 持久层(用于对DAO实现类进行注标)
* Controller web层(用于对Controller实现类进行注标)
* Service 业务层(用于对Service实现类进行注标)



没有名字的话,Bean名称默认为首字母小写的类名

@Repository, @Controller, @Service 都是@Component的子类,具有Component的所有功能,这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



1. **@Autowired**

**按照类型自动注入**

@Autowired顾名思义，就是自动装配，其作用是为了消除代码Java代码里面的getter/setter与bean属性中的property。当然，getter看个人需求，如果私有属性需要对外提供的话，应当予以保留。

@Autowired默认按类型匹配的方式，在容器查找匹配的Bean，当**有且仅有一个匹配的Bean**时，Spring将其注入@Autowired标注的变量中。

@Autowired注解的意思就是，当Spring发现@Autowired注解时，将自动在代码上下文中找到和其匹配（默认是类型匹配）的Bean(@Component及另外三个功能基本一样的注解)，并自动注入到相应的地方去。

有一个细节性的问题是，假如bean里面有两个property，Zoo.java里面又去掉了属性的getter/setter并使用@Autowired注解标注这两个属性那会怎么样？答案是Spring会按照xml优先的原则去Zoo.java中寻找这两个属性的getter/setter，导致的结果就是初始化bean报错。

如果@AutoWired自动注解根据类型匹配找不到Bean,我不想让Spring容器抛出异常，而就是显示null，可以吗？可以的，其实异常信息里面也给出了提示了，就是将@Autowired注解的required属性设置为false即可

public class Zoo {

@Autowired(required=false)

private Tiger tiger;

@Autowired(required=false)

private Monkey monkey;

public String toString(){

return tiger + "\n" + monkey;

}

1. **Qualifier（指定注入Bean的名称）**

按照指定的名称注入,必须与@AutoWired注解搭配使用

如果容器中有一个以上匹配的Bean，则可以通过@Qualifier注解限定Bean的名称;



public class CarFactory {

@Autowired

@Qualifier("bmwCar")

private ICar car;

public String toString(){

return car.getCarName();

}

}

**3. @Resource**

@Resource注解和@Autowired注解作用非常相似;

public class Zoo1 {

@Resource(name="tiger")

private Tiger tiger;

@Resource(type=Monkey.class)

private Monkey monkey;

public String toString(){

return tiger + "\n" + monkey;

}

}

这是详细一些的用法，说一下@Resource的装配顺序：  
 (1)、@Resource后面没有任何内容，默认通过name属性去匹配bean，找不到再按type去匹配  
 (2)、指定了name或者type则根据指定的类型去匹配bean  
 (3)、指定了name和type则根据指定的name和type去匹配bean，任何一个不匹配都将报错

然后，区分一下@Autowired和@Resource两个注解的区别：  
 (1)、@Autowired默认按照byType方式进行bean匹配，@Resource默认按照byName方式进行bean匹配  
 (2)、@Autowired是Spring的注解，@Resource是J2EE的注解，这个看一下导入注解的时候这两个注解的包名就一清二楚了  
 Spring属于第三方的，J2EE是Java自己的东西，因此，建议使用@Resource注解，以减少代码和Spring之间的耦合。

使用注解之前要开启自动扫描功能，其中base-package为需要扫描的包(含子包)。

<context:component-scan base-package="cn.test"/>

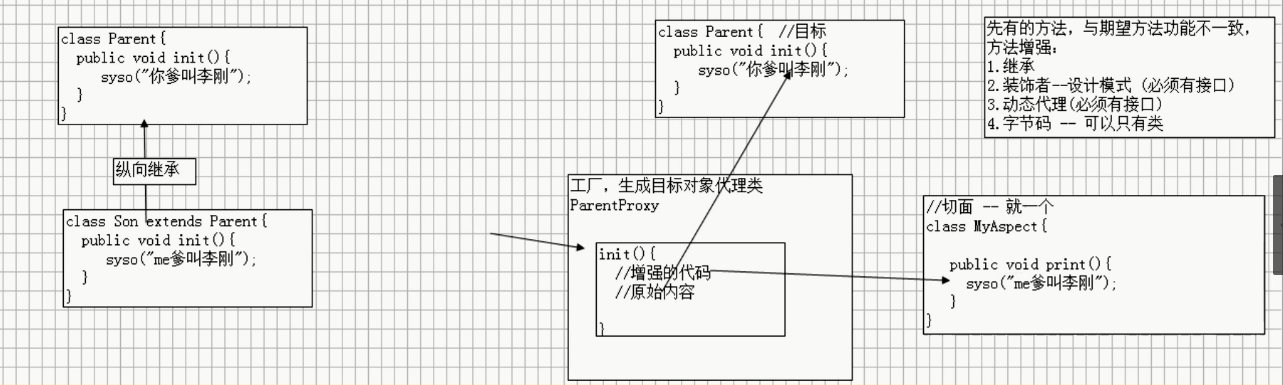
这两种方式没有什么所谓的优劣之分，主要看使用情况，一般来说是这样：

* 涉及到全局配置的，例如数据库相关配置、MVC相关配置等，就用JAVA配置的方式
* 涉及到业务配置的，就使用注解方式

## AOP(Aspect Oriented Programing)面向切面编程

* AOP Aspect Oriented Programing 面向切面编程
* 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(性能监听, 事务管理, 安全检查, 缓存)
* Spring AOP使用纯Java实现,不需要专门的编译过程和类加载器,在运行期通过代理方式向目标类植入增强代码
* 使用AOP常见应用: 性能监听, 事务管理, 日志文件, 缓存

将采用代理方法,将增强代码进行注入



装饰者-设计模式: 解决一种固定的问题的一种固定的代码

动态代理: 出现一个问题,就解决一个问题



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

1. 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

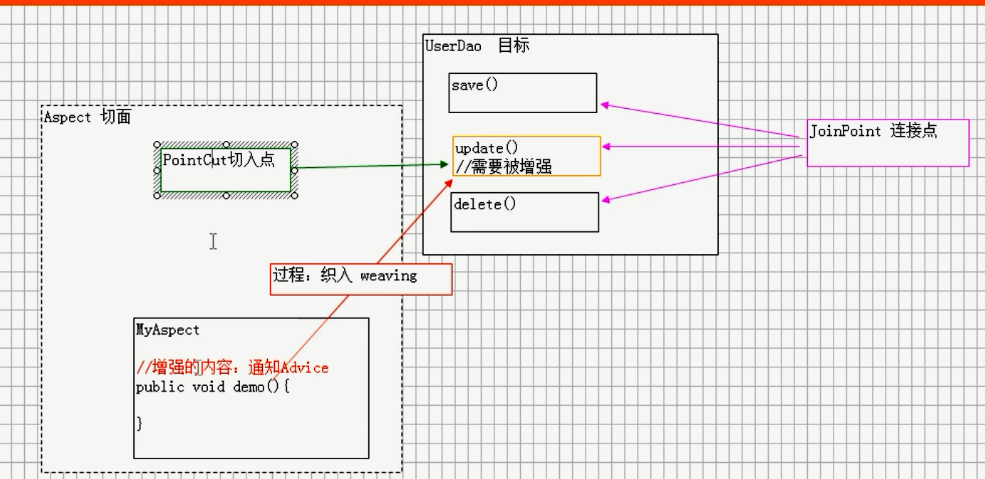
重点: 主要内容就是使用AspectJ进行aop编程.aop的框架

## AOP术语

* Target : 目标,被代理进行增强的.

例如: UserDao类, 功能: seve()/update()/find()等

* JoinPoint 连接点,交接点: 目标类需要被增强的方法.例如: UserDao中所有的方法(可以被增强的方法)
* PointCut 切入点: 需要被拦截的连接点JoinPoint. 例如: UserDao只有 save方法被增强.（拦截某个JoinPoint进行增强,实际增强的方法）
* Advice 通知/增强环节: 增强的代码.例如: MyAspect类中的print()方法(增强的业务逻辑代码)
* Introduction(引介) : 特殊的通知,一般通过增强方法,引介增强类
* Weaving 织入 : 将Advice通知(增强代码)应用到切入点PointCut过程.
* Porxy 代理 :一个类被AOP织入增强后，就产生一个结果代理类
* Aspect 切面 : 通知和切入点PointCut的结合(Adrice增强代码应用到PointCut的过程叫切面)



## 基于XML的声明式

|  |
| --- |
| <!-- 目标类 -->  <bean id=*"UserDaoImpl"* class=*"com.zsl.service.UserDaoImpl"*></bean>  <!-- Adrice类,增强方法的实例 -->  <bean id=*"LogRecord"* class=*"com.zsl.proxy.LogRecord"*></bean>    <!-- 进行aop配置 -->  <aop:config>  <!-- 配置切入点(要被增强的方法) -->  <aop:pointcut expression=*"execution(\*com.zsl.service.UserDaoImpl.addUser(..))"*  id=*"pointcutTest"*/>    <!-- 配置切面,把增强逻辑代码应用到要被增强的方法上面(Weaving织入) -->  <aop:aspect ref=*"LogRecord"*>  <!-- method指增强方法实例里的某个方法,pointcut-ref指要增强哪个方法 -->  <!-- before指在执行原方法代码之前执行 -->  <aop:before method=*"beforeAdvice"* pointcut-ref=*"pointcutTest"*/>  <!-- after指在执行原方法代码之后执行 -->  <aop:after-returning method=*"afterAdvice"* pointcut-ref=*"pointcutTest"*/>  <!-- around指环绕执行,增强的方法代码代替原有方法代码 -->  <aop:around method=*"jjcc"* pointcut-ref=*"pointcutTest"* />  </aop:aspect>  </aop:config> |

|  |
| --- |
| <!-- 配置aop -->  <aop:config>  <!-- 配置pointcut(实际被增强的方法), expression属性指具体的路径地址 -->  <aop:pointcut expression=*"execution(\* com.zsl.\*.\*.\*(..))"*  id=*"pointCutTest"*/>    <!-- 配置Aspect(切面指Adrice的增强逻辑代码实现到PointCut的过程叫切面) -->  <!-- aspect有一个属性,ref属性指向切面发生在哪个Adrice中 -->  <aop:aspect ref=*"operationImpl"*>  <!-- before指在运行原有方法逻辑代码之前运行 -->  <!-- <aop:before method="OperationTime" pointcut-ref="pointCutTest"/> -->    <!-- after值在运行原有方法逻辑代码之后运行 -->  <aop:after-returning method=*"OperationTime"* pointcut-ref=*"pointCutTest"*/>  <!-- around指环绕执行,增强的方法代码代替原有方法代码 -->  <aop:around method=*"jjcc"* pointcut-ref=*"pointcutTest"* />  </aop:aspect>  </aop:config> |

## AopAnnotation(aop注解方式)

* @Aspect 声明切面通知类,使spring可以获得所有通知
* 使切面开启自动代理

**<aop:aspectj-autoproxy></aop:aspect-autoproxy>**

**@ASpectJ提供不同的通知类型**

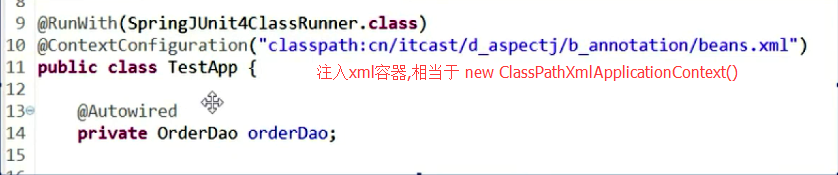
@before 前置通知,相当于beforeAdvice;

@afterReturning 后置通知,相当于AfterReturningAdvice;

@Around 环绕通知, 相当于MethodInterceptor;

@AfterThrowing 抛出通知,相当于ThrowingAdvice

@After 最终final通知,不管是否异常,该通知都会执行.



**Xml配置:**



**切面配置:**



**一定要加@Component,相当于Aop的xml配置切面时的实例Adrice类**



使用@AfterReturning注解时可指定如下两个常用属性：

☞ pointcut/value：这两个属性的作用是一样的，他们都用于指定该切入点对应的切入表达式。当指定了pointcut属性值后，value属性值将会被覆盖

☞ returning：指定一个返回值形参名，增强处理定义的方法可通过该形参名来访问目标方法的返回值

@Component

@Aspect

public class AfterReturningAdviceTest {

@AfterReturning(returning = "rvt", pointcut = "execution(\* main.java.service.\*.\*(..))")

public void log(Object rvt) {

System.out.println("获取目标方法返回值：" + rvt);

System.out.println("模拟记录日志功能...");

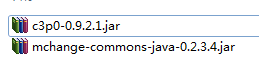
}

}

正如上面的程序中看到的，程序中使用@AfterReturning时，指定了一个returning属性，该属性值为rvt，这表明允许在增强处理方法（log方法）中使用名为rvt的形参，该形参代表目标方法的返回值。

## Spring JdbcTemplate(Jdbc模板)

**所需Jar包:**



SpringJdbcTemplate应用流程:

|  |
| --- |
| //配置数据库连接信息  DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("java\_jjcc");  dataSource.setPassword("123456"); |
| //创建JdbcTemplate对象  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);  String sql = "select \* from user2 where id=?";  List<UserTwoInstance> list = jdbcTemplate.query(sql,  **new** RowMapper<UserTwoInstance>(){  **public** UserTwoInstance mapRow(ResultSet rs, **int** arg1) **throws** SQLException {  UserTwoInstance user = **new** UserTwoInstance();  user.setId(rs.getInt(1));  user.setUserName(rs.getString(2));  user.setUserPassword(rs.getString(3));  user.setSex(rs.getString(4));  **return** user;  }  }, 4); |

其中JdbcTemplate的查询功能又大体分为两种:

query:

new JdbcTemplate(dataSource).query(sql, RowMapper<>{...}, “”);

该query方法放回的是一个List,返回的List具体类型是通过RowMapper<>{...}来实现的

queryForObject:

new JdbcTemplate(dataSource).queryForObject(String sql, Integer.class)

|  |
| --- |
| 1. String sql = "select ID from Structures(nolock) where id=?"; 2. Integer ret = jdbcTemplate.queryForObject(sql, Integer.**class**); |

queryForObject此重装第2个参数Class<T> requiredType只能是基本数据类型的封装类，如Integer、String

所以queryForObject其实支持的是标量子查询,只能传入一个基本类型的包装类的class,并返回一个基本类型对应包装类型的对象.

如果要实现使用queryForObject获得自定义的类的对象,需要使用和方法query相似的重载方法queryForObject(sql,Object[],int[],RowMapper<T>),通过RowMapper填充自定义的类,通过Object[]和int[]设定sql语句的参数参数和参数类型.

但是这样还会存在一些问题,就是如果sql语句查询出0条或者多条数据的话,queryForObject会抛出EmptyResultDataAccessException或IncorrectResultSetColumnCountException:的异常,而如果干脆使用方法query,或者queryForList则可以在编码中处理掉这种问题而无需try-catch

RowMapper:

RowMapper可以将数据中的每一行数据封装成用户定义的类;

可以通过建立内部类实现RowMapper接口,RowMapper中有一个mapRow方法,所以实现RowMapper接口一定要实现mapRow方法,而对自定义类的包装就在mapRow方法中实现.

|  |
| --- |
| 1. /\*\* 2. \* 定义内部类实现RowMapper接口 3. \*/ 4. **public** **class** NpcRowMapper **implements** RowMapper<Java实例Bean的实例类>{ 5. //实现mapRow方法 6. **public** Object mapRow(ResultSet rs, **int** num) **throws** SQLException { 7. //对类进行封装 8. TNpc npc = **new** TNpc(); 9. npc.setId(rs.getLong("id")); 10. npc.setName(rs.getString("name")); 11. **return** npc; 12. } 13. } 14. } |

### SpringXml配置文件配置JdbcTemplate:

1. 配置数据源信息

|  |
| --- |
| <!-- 配置数据源 -->  <bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <!-- 连接数据库基本四项 -->  <property name=*"driverClassName"* value=*"oracle.jdbc.driver.OracleDriver"*></property>  <property name=*"url"* value=*"jdbc:oracle:thin:@localhost:1521:orcl"*></property>  <property name=*"username"* value=*"java\_jjcc"*></property>  <property name=*"password"* value=*"123456"*></property>  </bean> |

1. 配置JdbcTemplate信息

|  |
| --- |
| <!-- 配置JdbcTemplate -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean> |

1. set方法实例化jdbcTemplate

|  |
| --- |
| <!-- set方法实例化jdbcTemplate -->  <bean id=*"operationDao"* class=*"com.zsl.dao.OperationDao"*>  <property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>  </bean> |



添加角色:

|  |
| --- |
| //添加角色  **public** **void** addUser(String name, String password, String sex){  String sql = "insert into user2 values(user2\_id.nextval, ?, ?, ?)";  **this**.jdbcTemplate.update(sql, name, password,sex);  } |

删除角色:

|  |
| --- |
| //删除角色  **public** **void** deleteUser(**int** index){  String sql = "delete from user2 where id=?";  **this**.jdbcTemplate.update(sql, index);  } |

查询角色:

|  |
| --- |
| //查询角色信息  **public** List<UserTwoInfo> queryUser() {  String sql = "select \* from user2";  List<UserTwoInfo> list = **this**.jdbcTemplate.query(sql, **new** RowMapper<UserTwoInfo>(){  **public** UserTwoInfo mapRow(ResultSet rs, **int** arg1) **throws** SQLException {  UserTwoInfo user = **new** UserTwoInfo();  user.setId(rs.getInt(1));  user.setUserName(rs.getString(2));  user.setUserPassword(rs.getString(3));  user.setSex(rs.getString(4));  **return** user;  }  });  **return** list;  } |

修改角色:

|  |
| --- |
| //修改密码  **public** **void** updateUser(String userName, String password, String sex){  String sql = "update user2 set userpassword=?, sex=? where username=?";  **this**.jdbcTemplate.update(sql, password, sex, userName);  } |

## 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 |

### 2. 在xml配置文件中读取db.properties文件

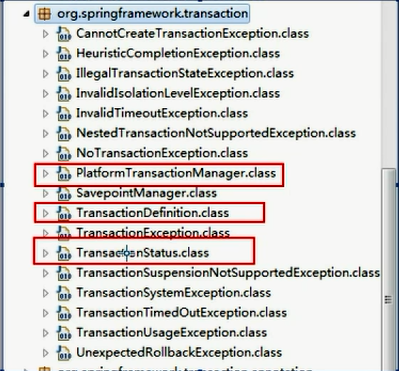
|  |
| --- |
| **<context:property-placeholder** location="classpath:db.properties" **/>**  **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>**  **<property** name="user" value="${jdbc.user}"**></property>**  **<property** name="password" value="${jdbc.password}"**></property>**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

## Spring事务管理

### 1. 介绍

spring提供jar用于事务管理 : spring-tx...jar;

IMG_256



PlatformTransactionManager: spring平台提供平台事务管理器,用于管理事务

TransactionDefinition: 事务定义(描述)对象,对事物进行近一步详细划分.及在xml对 事务进行配置信息.

详细描述事务: 传播行为, 隔离级别, 超时等

TransactionStatus: 事务状态,操作过程中某一时刻事务的状态

### 2.1 事务管理器



API:

TransactionStatus getTransaction(TransactionDefinition definition)

通过xml配置设施事务详细信息,spring将其封装到对象Transaction,通过事务管理器的getTransaction方法,可以获得事务的状态TransactionStatus,就可以对事务进行进一步操作.

保证事务的完整性和唯一性;

spring事务操作:

1. 配置事务详情(增强, 通知)
2. 配置事务管理器
3. 使用AOP设置切入点

### 2.2 事务详细定义对象TransactionDefinition



对事物详细设置: 传播行为, 隔离级别, 超时, 是否只读.

#### 2.2.1 传播行为:

传播行为: 两个被事务管理的方法相互调用的信息

种类:

propagation\_required: required(默认), 支持当前事务,如果A方法已经在事务中,B将直接使用.如果没有,将创建新事务.

propagation\_supports: supports, 支持当前事务, 如果A方法已经在事务中,B将直接使用.如果没有将以非实物状态执行.

propagation\_mandatory: mandatory, 支持当前事务, 如果A方法没有事务,将抛异常

propagation\_requires\_new: requires\_new, 将创建新的事务,如果A方法已经在事务中,将A挂起

propagation\_not\_supported: not\_supported, 不支持当前事务,总是以非事务状态执行, 如果A已经在事务中,将挂起.

propagation\_never: never, 不支持当前事务, 如果A方法在事务中,将抛出异常.

propagation\_nested: nested, 嵌套事务, 底层使用Savepoint形成嵌套事务.

常用:

propagation\_required

propagation\_requires\_new

propagation\_nested

#### 2.2.2 TransactionStatus(事务状态)



#### 2.2.3 Spring事务编程

编程式事务管理, 使用事务管理模板TransactionTemplate,手动的管理实务一般不使用.

声明式事务管理, 在Xml进行配置, 使用AOP进行事务管理.

事务管理学习思路:

编程式

工厂Bean生成代理

注解

### 2.3 案例

#### 2.3.1 搭建环境

##### 2.3.1.1 导入Jar包

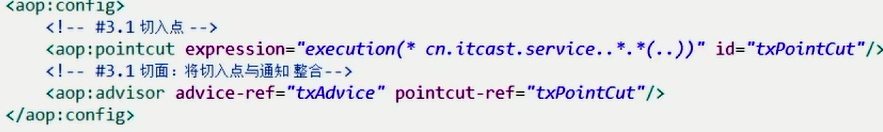


##### 2.3.1.2 声明式--tx xml

tx 提供<tx:advice>对事务管理器 提供事务进行详细配置, 及就是代理方式 transactionAttribute的相同配置信息,就是为了增强.spring aop 增强也成为通知.

tx 编程: 1, 事务管理器 2, 编写通知(增强) 3, 使用aop切面进行配置(通知和切入点的结合)





|  |
| --- |
| <!-- 生成事务管理实例 -->  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!-- 获得连接池 -->  <property name="dataSource" ref="dataSource"></property>  </bean>  <!-- 配置增强Adrice,对事务进行增强 -->  <!-- 配置事务传播特性 -->  <tx:adrice id="txAdrice" transaction-manager="txManager">  <!-- 对切入点增强的具体详情,name为要增强的方法 -->  <tx:attributes>  <tx:method name="method" propagation="required" isolation="default" read-only="false" />  </tx:attributes>  </tx:adrice>  <!-- 配置AOP,通过Spring自动对目标生成代理 -->  <!-- 配置参与事务的类 -->  <!-- 通过aop定义事务增强切面 -->  <aop:config>  <!-- 设置切入点 -->  <aop:pointcut expression="execution(\* com.zsl..\*(..))" id="txPointCut"></aop:pointcut>  <!-- 通知与切入的结合 -->  <aop:advisor pointcout-ref="txPointCut" adrice-ref="txAdrice"></aop:advisor>  </aop:config> |

##### 2.3.1.3 声明式--注解

xml配置将事务管理器注册给spring<tx:annotation-driven transaction-manager=＂transactionManager＂>

在需要使用事务的类或者方法上使用@Transactional();

1. 获取事务管理器实例

<bean id=”transactionManager”class=”org.springframework.jdbc.dataSo ure.dataSourceTransactionManager”>

<property name=”dataSource” ref=”dataSource”></property>

</bean>

2. 打开事务注解

<tx:annotation-driven transaction-manager=”transactionManager”/>



