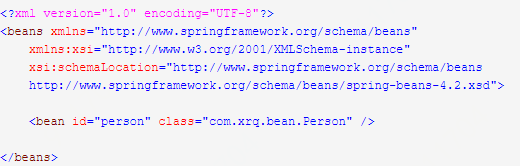
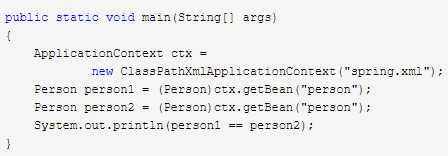
# Ssm技术

## Spring

### Bean使用



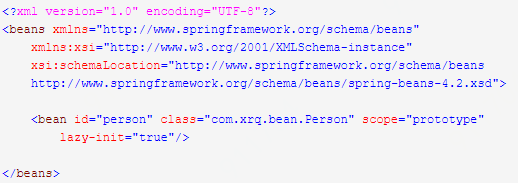


运行结果为true，也就是说**Spring默认以单例的形式给开发者构造出指定的bean**。另外有两点要注意：

1、同一个spring.xml中不可以定义两个id相同的bean

2、ClassPathXmlApplicationContext中有一个可变长度的构造函数，用于加载多个.xml中的bean，如果bean中有id相同，那么id相同的bean，**后加载的会覆盖先加载的**。

#### 作用域及生命周期



1、scope表示的是bean的作用域，有prototype、request、session、singleton四种，其中singleton是默认的，表示单例。prototype表示每次创建都会产生一个bean实例。request和session只在web项目中才会用，其作用域就和web中的request和session一样

2、lazy-init表示的是延迟实例化，默认为false。当scope=singleton时，bean会在装在配置文件时实例化，如果希望bean在产生时才实例化，可以把lazy-init设置为true。当scope=prototype时，在产生bean时才会实例化它。补充一点，如果希望该配置文件中所有的bean都延迟初始化，则应该**在beans根节点中使用lazy-init="true"。**

#### **三种注入方式**

所谓注入即注入bean中的属性，Spring为用户提供了三种注入方式，settter注入、构造方法注入、注解注入。

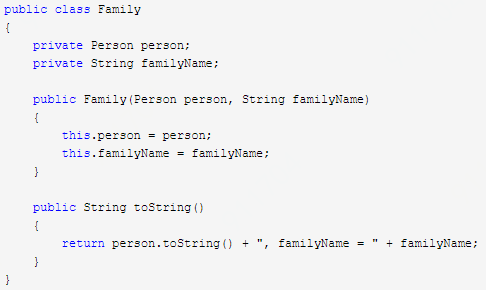
**Setter注入**



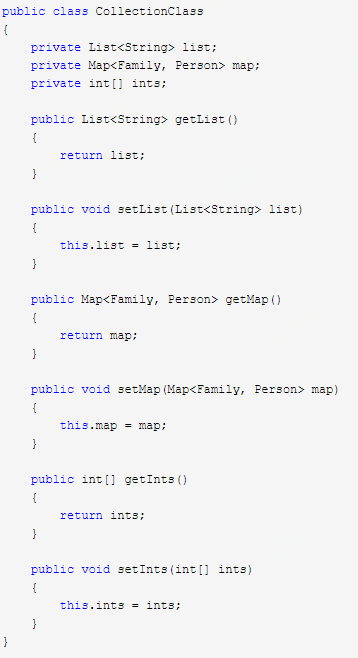
**构造方法注入**

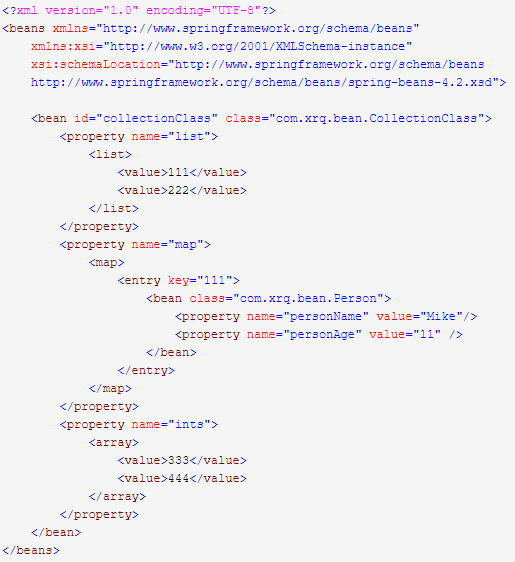


其中family中有person的引用：



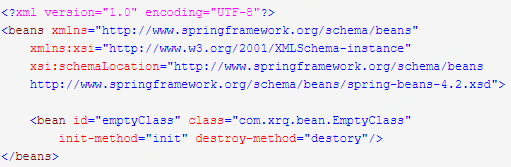
**集合注入：**





#### **init-method和destory-method**

有时候我们希望，在某个bean加载的时候做一些事情，销毁的时候做一些事情

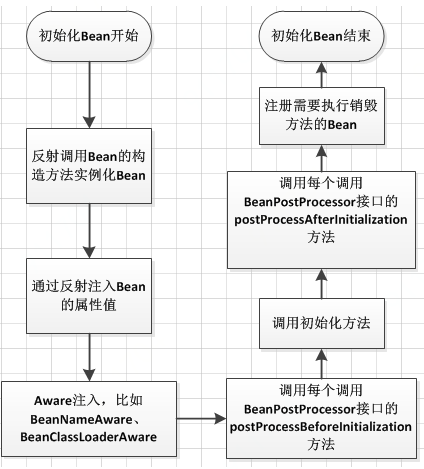


注意两点：

1、实例化类的时候，几个方法的加载顺序为**静态资源->构造方法->init方法**

2、触发destory()方法的调用可以使用"((ClassPathXmlApplicationContext)ctx).close();"，注意**scope="prototype"是不会触发destory()的**，没有为什么，设计就是这样。

### Bean加载



实际上，还有一些附加阶段：

Spring容器创建BeanFactoryPostProcessor实例

调用BeanFactoryPostProcessor的postProcessBeanFactory方法

Spring容器创建BeanPostProcessor实例

1.首先容器启动后，会对scope为singleton且非懒加载的bean进行实例化（反射）。

2.按照Bean定义信息配置信息，注入所有的属性。

3.各种Aware注入BeanNameAware，BeanFactoryAware，ApplicationContextAware。

4. 如果有Bean实现了BeanPostProcessor接口，则会回调该接口的postProcessBeforeInitialzation()方法

这里少了一步

调用InitializingBean的afterPropertiesSet()方法;   
8. 调用定制init-method方法；

5.init-method

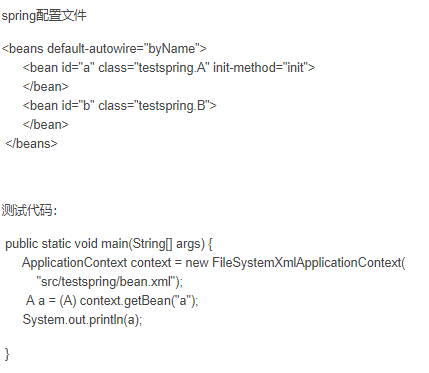
6. 如果有Bean实现了BeanPostProcessor接口，则会回调该接口的postProcessAfterInitialization()方法

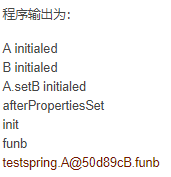
7.使用bean

8.销毁实例destroy-method

#### 例子







从这里看到A的name属性在bean加载完成的时候也被成功设置为B的funB方法的返回值了，要点就是用init-method来实现。

加载顺序也可以看到为：

先构造函数——>然后是b的set方法注入——>InitializingBean   的afterPropertiesSet方法——>init-method方法

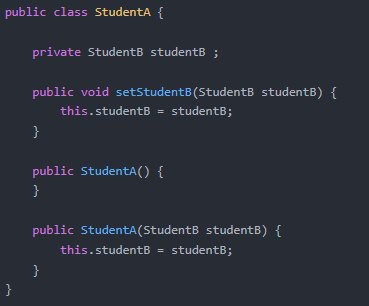
### 循环依赖注入

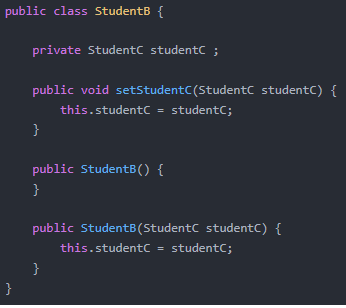
分三种情况：

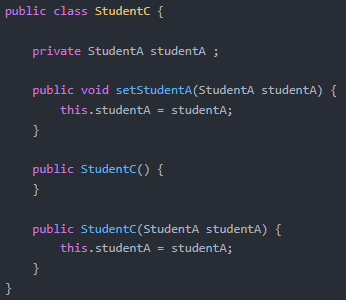
#### 第一种：构造器参数循环依赖

Spring容器会将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持  
 在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出  
BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

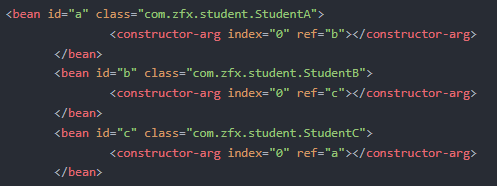
**看一个例子**







样就产生了一个循环依赖的情况。





测试后报错：

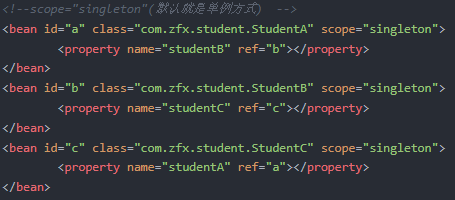
Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException:

Error creating bean with name 'a': Requested bean is currently in creation: Is there an unresolvable circular reference?

Spring容器先创建单例StudentA，StudentA依赖StudentB，然后将A放在“当前创建Bean池”中，此时创建StudentB,StudentB依赖StudentC ,然后将B放在“当前创建Bean池”中,此时创建StudentC，StudentC又依赖StudentA， 但是，此时Student已经在池中，所以会报错，，因为在池中的Bean都是未初始化完的，所以会依赖错误 ，（初始化完的Bean会从池中移除）

#### ****第二种：setter方式单例，默认方式****

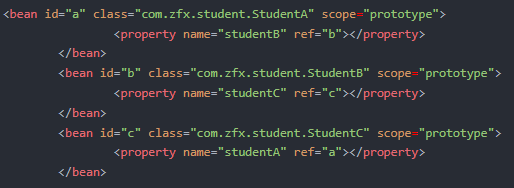
回顾上文的bean加载过程，这种方式就是用的上文的bean加载方式。**Spring是先将Bean对象实例化之后再设置对象属性的**



测试后正常。

**Spring先是用构造实例化Bean对象 ，此时Spring会将这个实例化结束的对象放到一个Map中，并且Spring提供了获取这个未设置属性的实例化对象引用的方法。   结合我们的实例来看，，当Spring实例化了StudentA、StudentB、StudentC后，紧接着会去设置对象的属性，此时StudentA依赖StudentB，就会去Map中取出存在里面的单例StudentB对象，以此类推，不会出来循环的问题。**

#### ****第三种：setter方式原型，prototype****



scope="prototype" 意思是 每次请求都会创建一个实例对象。

测试后会报错，**对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。**

总结一下就是：

Spring为了解决单例的循环依赖问题，使用了**三级缓存**。这三级缓存分别指：

singletonFactories ： 单例对象工厂的cache   
earlySingletonObjects ：提前暴光的单例对象的Cache   
singletonObjects：单例对象的cache

在创建bean的时候，首先想到的是从singletonObjects中获取这个单例的bean，如果获取不到，并且对象正在创建中就再从二级缓存earlySingletonObjects中获取，如果还是获取不到且允许singletonFactories通过getObject()获取，就从三级缓存singletonFactory.getObject()(三级缓存)获取。如果获取到了则从singletonFactories中移除，并放入earlySingletonObjects中。其实也就是从三级缓存移动到了二级缓存。

解决循环依赖的关键就是三级缓存singletonFactories，这个缓存中存储的是经过实例化但并未赋值的bean，Spring此时将这个对象提前曝光。完全初始化的bean会放在以及缓存中。

为啥Spring不能解决“A的构造方法中依赖了B的实例对象，同时B的构造方法中依赖了A的实例对象”呢？因为加入singletonFactories三级缓存的前提是执行了构造器，所以构造器的循环依赖没法解决。

### BeanPostProcessor和BeanFactoryProcessor浅析

BeanPostProcessor 接口定义了一个你可以自己实现的回调方法，来实现你自己的实例化逻辑、依赖解决逻辑等，如果你想要在Spring完成对象实例化、配置、初始化之后实现自己的业务逻辑，你可以补充实现一个或多个BeanPostProcessor的实现。

BeanFactoryPostProcessor的定义和BeanPostProcessor相似，有一个最主要的不同是：BeanFactoryPostProcessor可以对bean的配置信息进行操作；更确切的说Spring IOC容器允许BeanFactoryPostProcessor读取配置信息并且能够在容器实例化任何其他bean（所有的实现了BeanFactoryPostProcessor接口的类）之前改变配置信息。

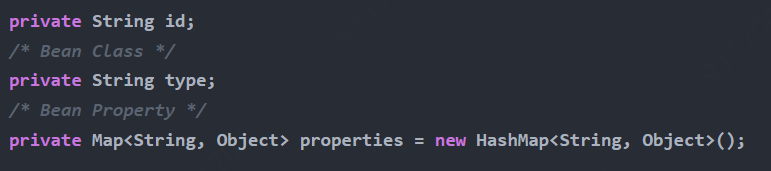
### IOC

IOC的思想是：Spring容器来实现相互依赖对象的创建、协调工作。对象只需要关系业务逻辑本身就可以了。

IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。比如对象A需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的。那么DI是如何实现的呢？ Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring就是通过反射来实现注入的。

#### 例子

**首先，我们定义一个Bean类，这个类用来存放一个Bean拥有的属性**



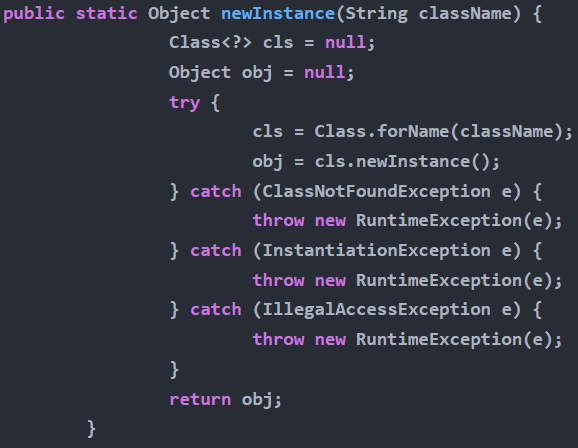
**接下来Spring 就开始加载我们的配置文件了，将我们配置的信息保存在一个HashMap中，HashMap的key就是Bean 的 Id ，HasMap 的value是这个Bean，只有这样我们才能通过context.getBean()这个方法获得这个类。**

**Map配置**



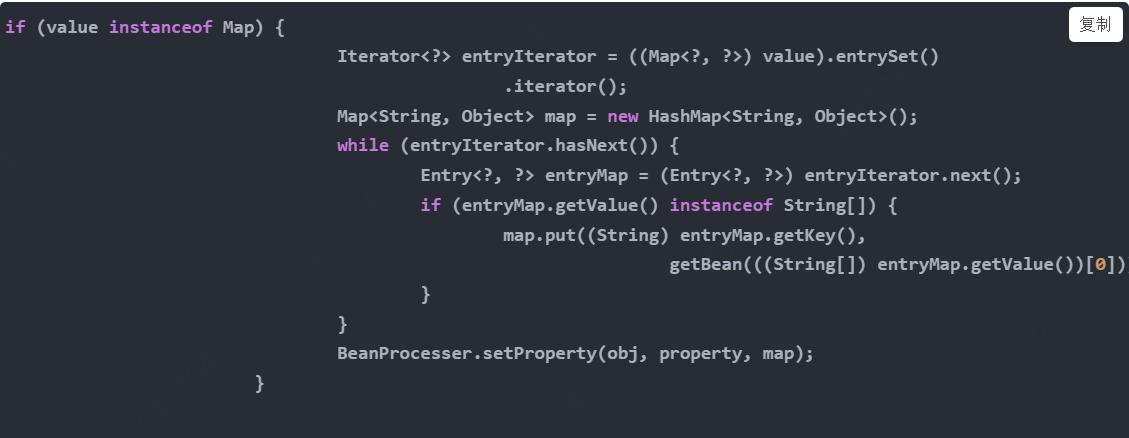
**接下来就是依赖注入，其实依赖注入的思想也很简单，它是通过反射机制实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中。**

**首先实例化一个类：**



**接着它将这个类的依赖注入进去，像这样**



**最后它将这个类的实例返回给我们，我们就可以用了。**

### BeanFactory 和 ApplicationContext

BeanFacotry是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。    
   ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。

除此之外，ApplicationContext还有其他功能。

1. 利用MessageSource进行国际化
2. 资源访问，如URL和文件

ApplicationContext扩展了ResourceLoader(资源加载器)接口，从而可以用来加载多个Resource，而BeanFactory是没有扩展ResourceLoader

1. 强大的事件机制(Event)

ApplicationContext的事件机制主要通过ApplicationEvent和ApplicationListener这两个接口来提供的，和java swing中的事件机制一样。即当ApplicationContext中发布一个事件的时，所有扩展了ApplicationListener的Bean都将会接受到这个事件，并进行相应的处理。

1. 其他

1).BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，这样，我们就不能发现一些存在的Spring的配置问题。而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误。    
  
  2).BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册

### Aop

AOP（Aspect Oriented Programming），即面向切面编程，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

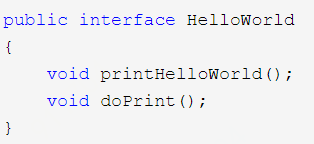
使用"横切"技术，AOP把软件系统分为两个部分：**核心关注点**和**横切关注点**。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

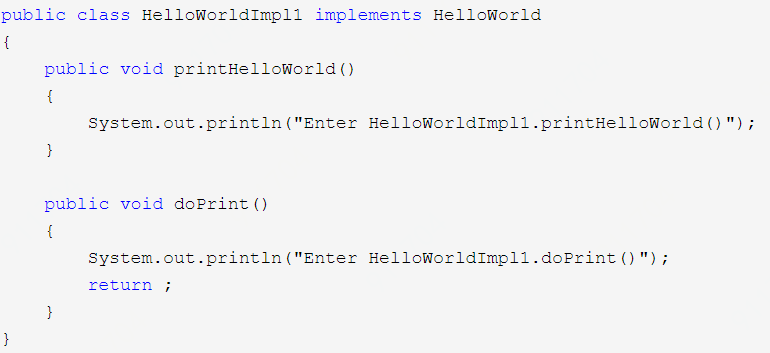
**Spring对AOP的支持：**

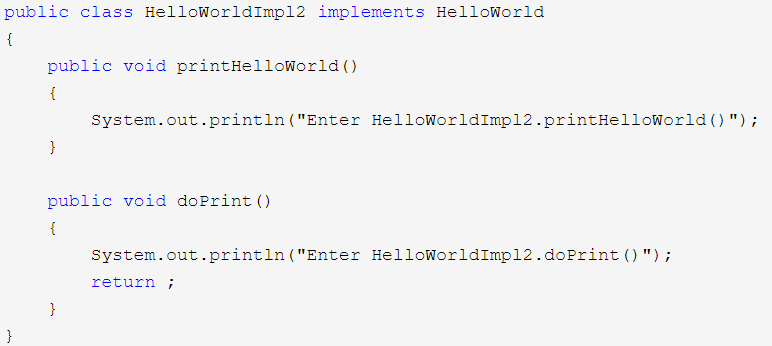
1、**默认使用Java动态代理来创建AOP代理**，这样就可以为任何接口实例创建代理了

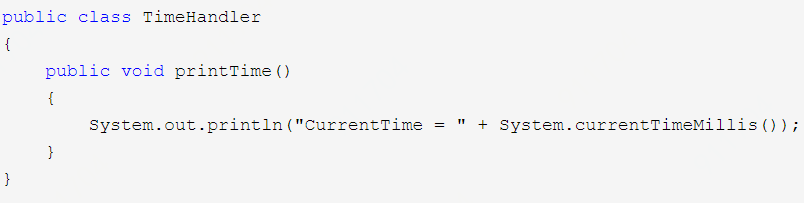
2、**当需要代理的类不是代理接口的时候，Spring会切换为使用CGLIB代理**，也可强制使用CGLIB。

#### 例子











1、横切关注点

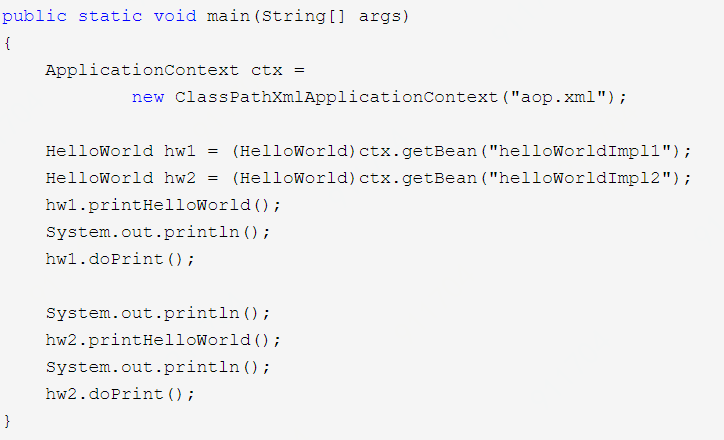
对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

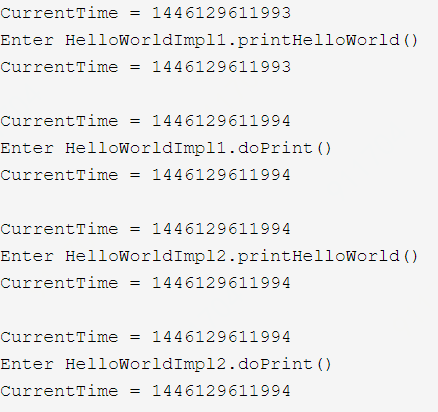
2、切面（aspect）

类是对物体特征的抽象，切面就是对横切关注点的抽象

3、切入点（pointcut）

对连接点进行拦截的定义



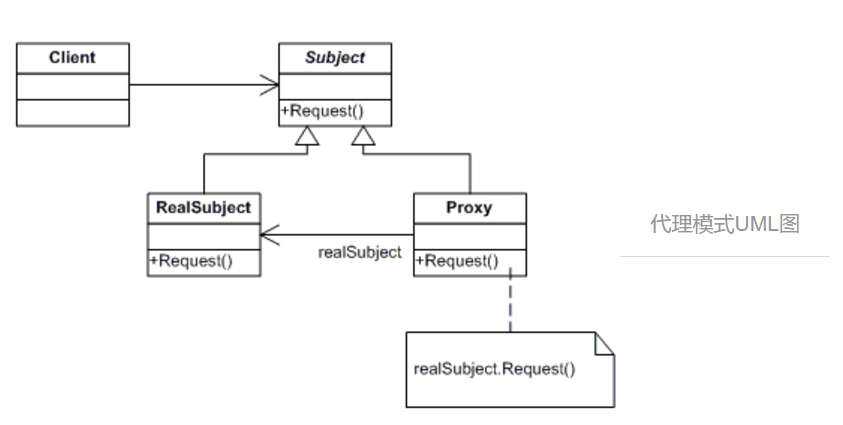


aspect里面有一个order属性，order属性的数字就是横切关注点的顺序

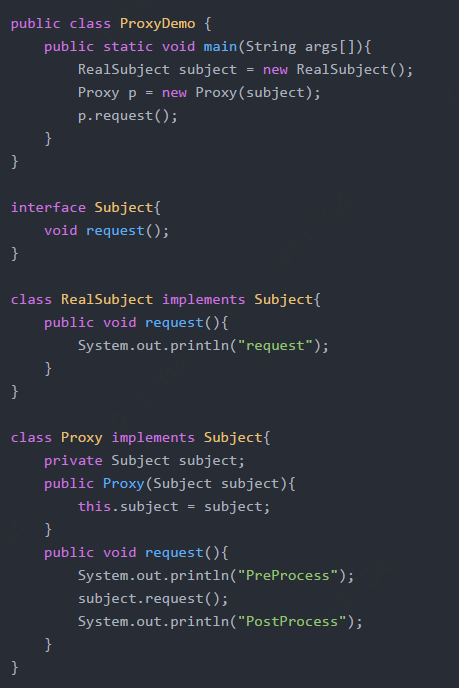
默认情况下，以定义顺序为准。

#### 动态代理

##### 代理模式



定义：给某个对象提供一个代理对象，并由代理对象控制对于原对象的访问，即客户不直接操控原对象，而是通过代理对象间接地操控原对象。



代理的实现分为：

* 静态代理：代理类是在编译时就实现好的。也就是说 Java 编译完成后代理类是一个实际的 class 文件。
* 动态代理：代理类是在运行时生成的。也就是说 Java 编译完之后并没有实际的 class 文件，而是在运行时动态生成的类字节码，并加载到JVM中。

##### Java 实现动态代理

1.定义一个委托类和公共接口。

2.自己定义一个类（调用处理器类，即实现 InvocationHandler 接口），这个类的目的是指定运行时将生成的代理类需要完成的具体任务（包括Preprocess和Postprocess），即代理类调用任何方法都会经过这个调用处理器类。  
3.生成代理对象（当然也会生成代理类），需要为他指定(1)委托对象(2)实现的一系列接口(3)调用处理器类的实例。因此可以看出一个代理对象对应一个委托对象，对应一个调用处理器实例。

**看一个例子**



  
  
newProxyInstance这个静态函数的第一个参数是类加载器对象（即哪个类加载器来加载这个代理类到 JVM 的方法区），第二个参数是接口（表明你这个代理类需要实现哪些接口），第三个参数是调用处理器类实例（指定代理类中具体要干什么）。

Proxy 类中有一个映射表，映射关系为：(<ClassLoader>,(<Interfaces>,<ProxyClass>) )，可以看出一级key为类加载器，根据这个一级key获得二级映射表，二级key为接口数组，因此可以看出：一个类加载器对象和一个接口数组确定了一个代理类。

Invoke方法是代理对象调用任何一个方法时都会调用的，方法不同会导致第二个参数method不同，第一个参数是代理对象（表示哪个代理对象调用了method方法），第二个参数是 Method 对象（表示哪个方法被调用了）。

因为 Java 的单继承特性（每个代理类都继承了 Proxy 类），只能针对接口创建代理类，不能针对类创建代理类。

##### cglib动态代理

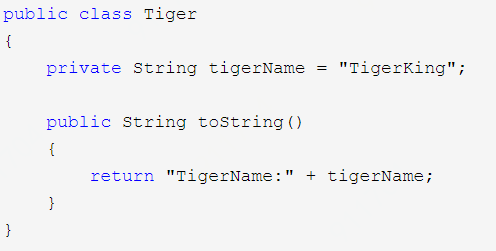
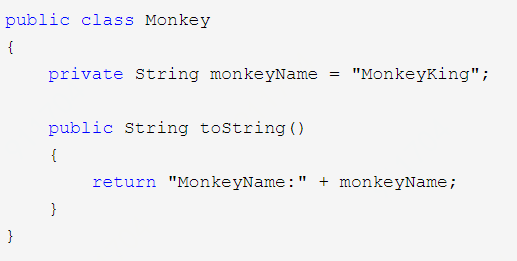
JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理，cglib是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。cglib实现动态代理的方法和JDK动态代理类似。cglib实现了MethodInterceptor接口，重写了intercept方法。

## Springmvc

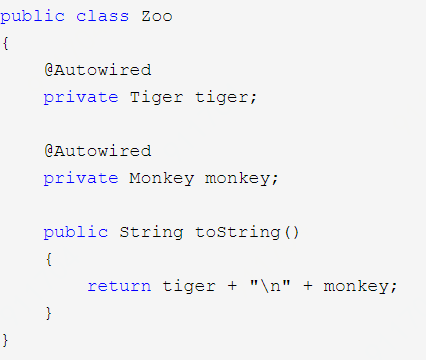
### @Autowired、@Resource和@Service

#### **@Autowired**

@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。





@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，并自动注入到相应的地方去。

将@Autowired注解的required属性设置为false，找不到bean时不会报错，会将属性默认为null。

**那么如果有一个接口，有多个实现，Bean里引用的是接口名，又该怎么做呢**

**通常直接用**@Qualifier注解，@Qualifier（“类名”），**括号里面的应当是接口实现类的类名，而不是bean名。**

#### **@Resource**

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

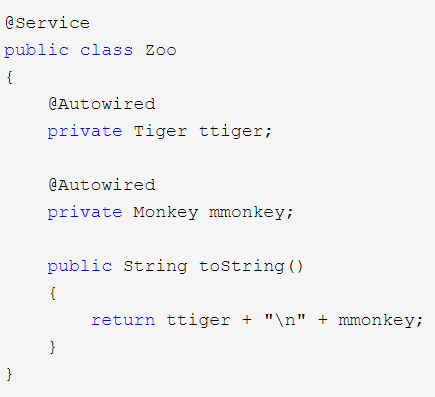
3、指定了name和type则根据指定的name和type去匹配bean，任何一个不匹配都将报错

@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

#### **@Service**

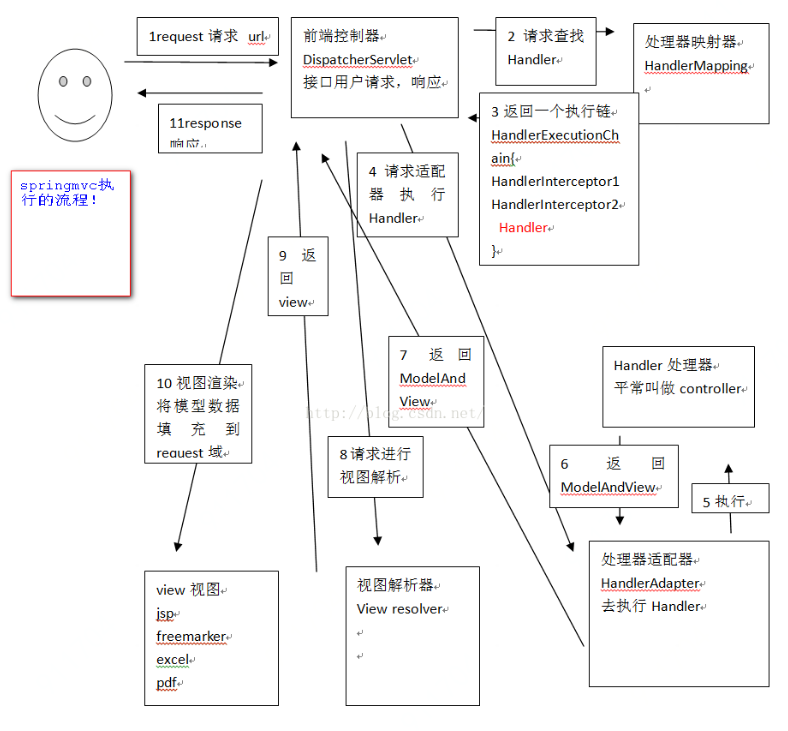


@Service注解，其实做了两件事情：

1、声明Zoo.java是一个bean，这点很重要，因为Zoo.java是一个bean，其他的类才可以使用@Autowired将Zoo作为一个成员变量自动注入

2、Zoo.java在bean中的id是"zoo"，即类名且首字母小写

### Springmvc执行流程



第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求HandlerMapping查找 Handler，可以根据xml配置、注解进行查找

第三步：处理器映射器HandlerMapping向前端控制器返回Handler

第四步：前端控制器调用处理器适配器去执行Handler

第五步：处理器适配器去执行Handler

第六步：Handler执行完成给适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView

第八步：前端控制器请求视图解析器去进行视图解析

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染

         视图渲染将模型数据(在ModelAndView对象中)填充到request域

第十一步：前端控制器向用户响应结果

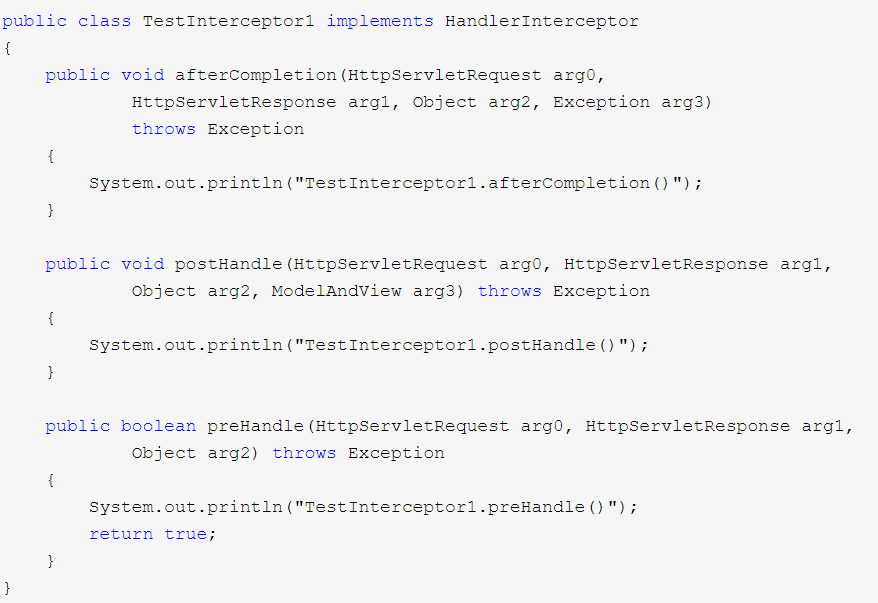
### Model

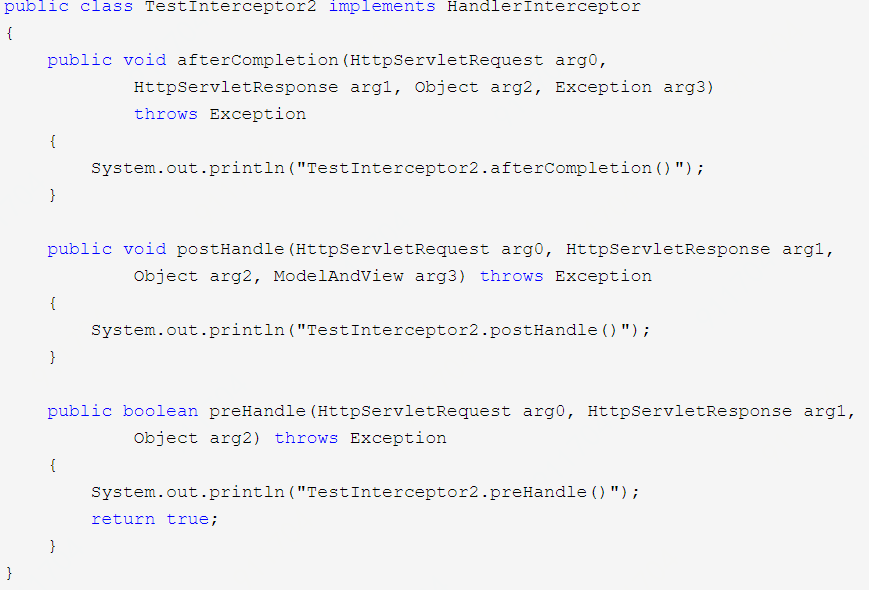
1. Model本身是一个接口，其实现类为ExtendedModelMap，除了使用Model之外还可以使用ModelAndView、ModelMap这些，不过要是没有特殊需求，使用Model比较简单。
2. **Model的生命周期是Request**，也就是说要通过Model传值只能使用转发而不能使用重定向。

### **拦截器（Interceptor）**

SpringMVC中的拦截器相当于J2EE中的过滤器，是非常重要和相当有用的，它的主要作用就是拦截用户的请求并进行相应的处理的，比如通过它来进行权限验证，或者是来判断用户是否登陆。

在SpringMVC中使用拦截器的方法比较简单，首先实现HandlerInterceptor接口，实现afterCompletion、postHandle、preHandle三个抽象方法。





三个方法的作用：

1. afterCompletion：在整个视图渲染完毕之后执行方法里面的内容，主要用于释放一些资源。
2. postHandle：在Controller执行之后，视图渲染之前执行方法里面的内容，也就是说postHandle方法可以对Model进行操作。
3. preHandle：在Controller执行之前，执行方法里面的内容，注意该方法是有返回值的，**当方法返回false时整个请求就结束了。**



## Mybatis