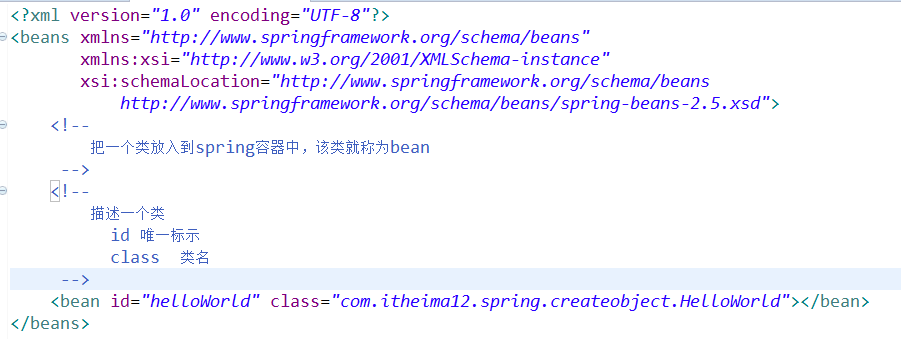
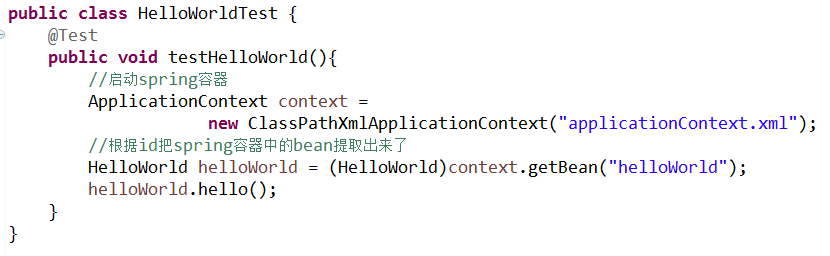
# SpringIOC

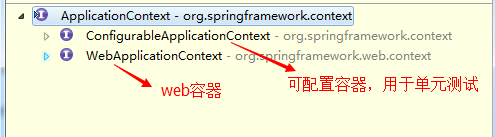
## SpringIOC

Spring的控制反转：把对象的创建、初始化、销毁等工作交给spring容器来做。由spring容器控制对象的生命周期。

SpringIOC的作用主要是让程序员的工作重点转移了，不用自己创建对象了，spring容器为你创建对象了







## 创建对象的方式

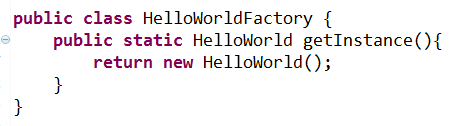
### 无参构造函数



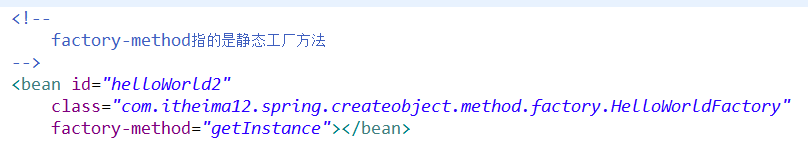
在spring容器中，默认情况下调用了一个类的默认的构造函数创建对象

### 静态工厂

静态工厂:



配置文件：

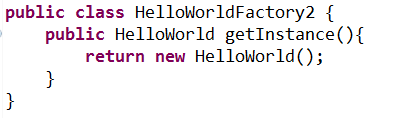


客户端：

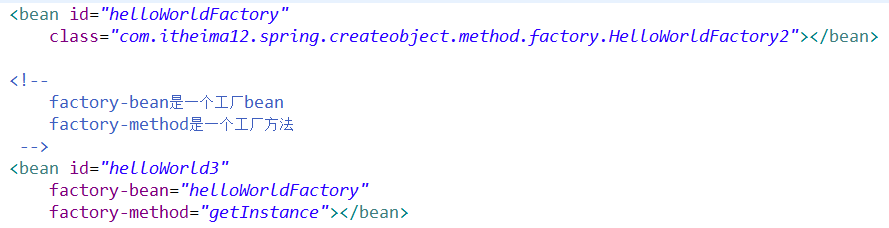


### 实例工厂

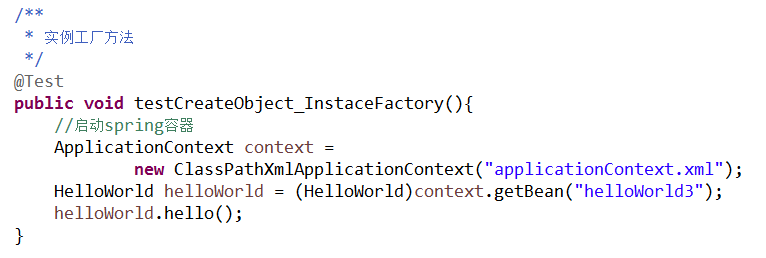
实例工厂：



配置文件：

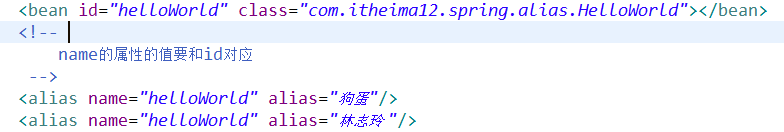


客户端：

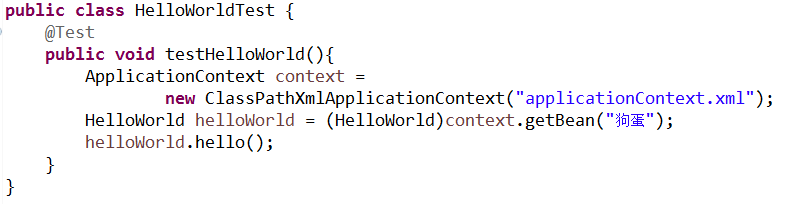


## 别名

配置:



客户端:



## 初始化bean时机

默认：



/\*\*

\* 在默认的情况下，启动spring容器的时候创建对象

\* 因为是在spring容器启动的时候就创建对象，所以只要配置文件书写错误，在一开始的时候(web容器启动)就能发现错误了

\*/

lazy-init:



<bean id="helloWorld2"

class="com.itheima12.spring.createobject.when.HelloWorld"

lazy-init="true"></bean>

在context.getBean时才要创建该对象

## 多实例和单实例

默认：



/\*\*

\* 在spring容器中的对象，默认情况下是单例的

\* 因为对象是单例的，所以只要在类上声明一个属性，该属性中含有数据，那么该属性是全局的,要注意线程安全问题（一般service，dao都是单例的，所以service，dao中不要用实例属性存储数据）

\*/

多例：



/\*\*

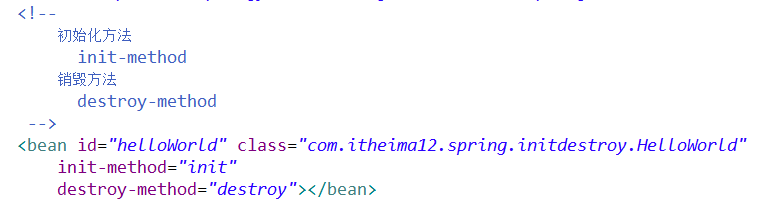
\* 如果说scope为"prototype"的时候，spring容器产生的对象就是多实例

\* 无论lazy-init为什么值，都是在context.getBean时才要创建对象

\*/

一般service和dao都放在spring容器中，是单例的，因为使用单例的话，对于service层和dao层中的某一个类来说，整个应用中只有一个对象，如果使用多例，那么请求多少次就会有产生多少个service层和dao层的对象

## init、destroy



/\*\*

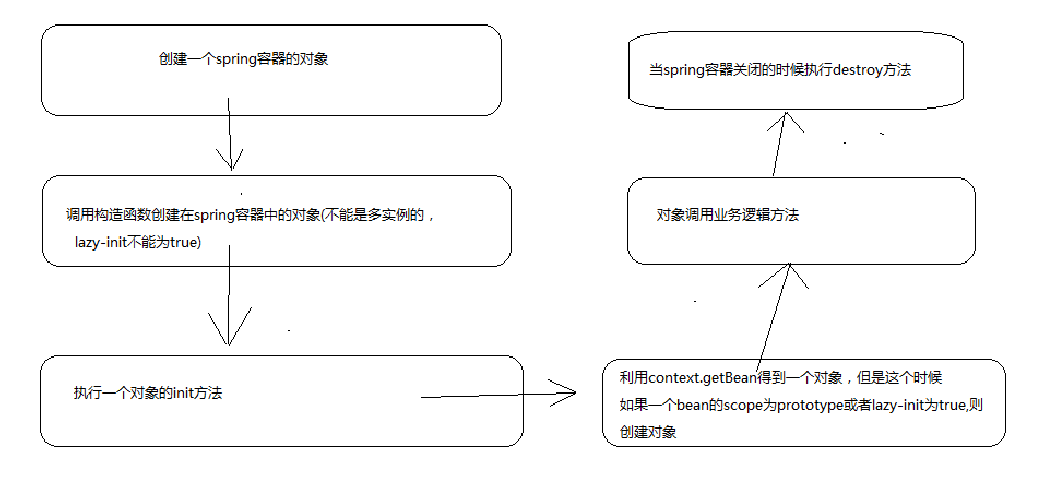
\* 在构造函数之后，立刻执行init方法

\* 如果spring容器没有执行close方法，则不执行销毁方法

\* 如果spring容器执行了close方法，在执行该方法之前要执行销毁方法

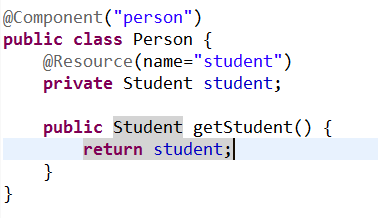
\*/

执行流程：



## 类扫描

Person:



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"*

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

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-2.5.xsd"*>

<!--

component:把一个类放入到spring容器中,该类就是一个component

在base-package指定的包及子包下扫描所有的类

-->

<context:component-scan base-package=*"com.itheima12.spring.scan"*>

</context:component-scan>

</beans>

/\*\*

\* 原理

\* 1、启动spring容器，spring容器解析配置文件

\* 2、当解析到<context:component-scan base-package="com.itheima12.spring.scan">

</context:component-scan>

就会在上面指定的包及子包中扫描所有的类，看哪些类上面有@Component注解

3、如果有该注解，则有如下的规则：

**@Component**

public class PersonDaoImpl{

}

==

<bean id="personDaoImpl" class"..."/> id的值：把类的第一个字母变成小写，其他字母不变

**@Component**("personDao")

public class PersonDaoImpl{

}

==

<bean id="personDao" class=".."/>

4、按照@Resource注解的规则进行赋值

\*/

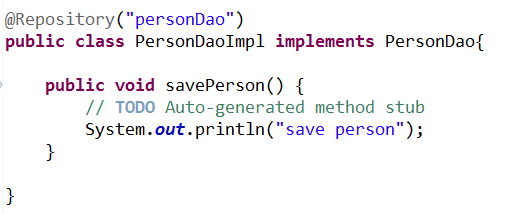
@Service用于标注业务层组件、

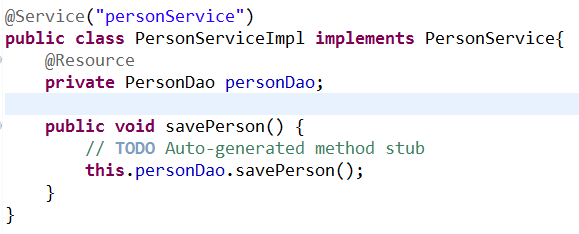
@Controller用于标注控制层组件（如struts中的action）、

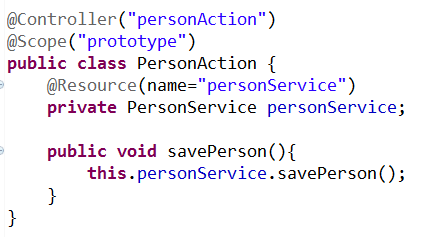
@Repository用于标注数据访问组件，即DAO组件。

而@Component泛指组件

项目中实际应用：







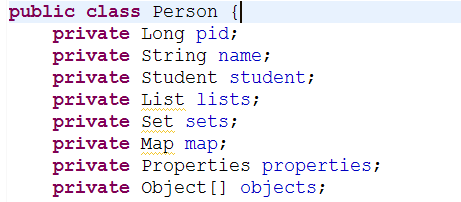
# 依赖注入(DI)



## xml形式

### 使用属性setter方法注入

**这种方式主要应用与于:在spring文件中，有一些配置相关的信息(就是给bean的属性做装配)**



<bean id=*"person"* class=*"com.itheima12.spring.di.xml.setter.Person"*>

<!--

property就是一个bean的属性

name就是用来描述属性的名称

value就是值，如果是一般类型(基本类型和String)

-->

<property name=*"pid"* value=*"1"*></property>

<property name=*"name"* value=*"狗蛋"*></property>

<!--

spring容器内部创建的student对象给Person的student对象赋值了

-->

<property name=*"student"*>

<ref bean=*"student"*/>

</property>

<property name=*"lists"*>

<list>

<value>list1</value>

<value>list2</value>

<ref bean=*"student"*/>

</list>

</property>

<property name=*"sets"*>

<set>

<value>set1</value>

<value>set2</value>

<ref bean=*"student"*/>

</set>

</property>

<property name=*"map"*>

<map>

<entry key=*"m1"*>

<value>map1</value>

</entry>

<entry key=*"m2"*>

<ref bean=*"student"*/>

</entry>

</map>

</property>

<property name=*"properties"*>

<props>

<prop key=*"p1"*>prop1</prop>

<prop key=*"p2"*>prop2</prop>

</props>

</property>

<property name=*"objects"*>

<list>

<value>obj1</value>

<ref bean=*"student"*/>

</list>

</property>

</bean>

<bean id=*"student"* class=*"com.itheima12.spring.di.xml.setter.Student"*></bean>

#### 装配一般类型(基本类型和String)

<property name=*"pid"* value=*"1"*></property>

<property name=*"name"* value=*"狗蛋"*></property>

#### 装配引用类型

<property name=*"student"*>

<ref bean=*"student"*/>

</property>

#### 装配list集合

<property name=*"lists"*>

<list>

<value>list1</value>

<value>list2</value>

<ref bean=*"student"*/>

</list>

</property>

#### 装配set集合

<property name=*"sets"*>

<set>

<value>set1</value>

<value>set2</value>

<ref bean=*"student"*/>

</set>

</property>

#### 装配map集合

<property name=*"map"*>

<map>

<entry key=*"m1"*>

<value>map1</value>

</entry>

<entry key=*"m2"*>

<ref bean=*"student"*/>

</entry>

</map>

</property>

#### 装配properties集合

<property name=*"properties"*>

<props>

<prop key=*"p1"*>prop1</prop>

<prop key=*"p2"*>prop2</prop>

</props>

</property>

#### 装配数组

<property name=*"objects"*>

<list>

<value>obj1</value>

<ref bean=*"student"*/>

</list>

</property>

### 使用构造器注入

#### 通过参数的顺序

<bean id=*"person"* class=*"com.itheima12.spring.di.xml.constructor.Person"*>

<!--

constructor-arg指的是构造器中的参数

index 角标 从0开始

value 如果一般类型,用value赋值

ref 引用类型赋值

-->

<constructor-arg index=*"0"* value=*"asdfsafd"*></constructor-arg>

<constructor-arg index=*"1"* ref=*"student"*></constructor-arg>

</bean>

<bean id=*"student"* class=*"com.itheima12.spring.di.xml.constructor.Student"*></bean>

#### 通过参数的类型

<bean id=*"person"* class=*"com.itheima12.spring.di.xml.constructor.Person"*>

<!--

type 指的是构造器中的参数

-->

<constructor-arg type=*"java.lang.String"* value=*"asdfsafd"*></constructor-arg>

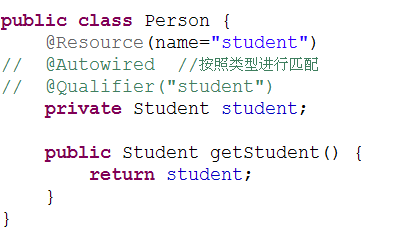
<constructor-arg type=*"com.itheima12.spring.di.xml.constructor.Student"* ref=*"student"*></constructor-arg>

</bean>

<bean id=*"student"* class=*"com.itheima12.spring.di.xml.constructor.Student"*></bean>

## 注解形式

Person：



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"*

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

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-2.5.xsd"*>

<bean id=*"person"* class=*"com.itheima12.spring.di.annotation.Person"*></bean>

<bean id=*"student"* class=*"com.itheima12.spring.di.annotation.Student"*></bean>

<!--

启动依赖注入的注解解析器

-->

<context:annotation-config></context:annotation-config>

</beans>

### @Resource

/\*\*

\* 原理

\* 1、当启动spring容器的时候，创建两个对象

\* 2、当spring容器解析到

\* <context:annotation-config></context:annotation-config>

\* spring容器会在spring容器管理的bean的范围内查找这些类的属性上面是否加了@Resource注解

\* 3、spring解析@Resource注解的name属性

\* 如果name属性为""

\* 说明该注解根本没有写name属性

\* spring容器会得到该注解所在的属性的名称和spring容器中的id做匹配，如果匹配成功，则赋值

\* 如果匹配不成功,则按照类型进行匹配

\* 如果name属性的值不为""

\* 则按照name属性的值和spring的id做匹配，如果匹配成功，则赋值，不成功，则报错

\* 说明:

\* 注解只能用于引用类型

\* 注解写法比较简单,但是效率比较低

\* xml写法比较复杂，但是效率比较高

\*/

@Resource注解是java扩展包中的类

### @Autowired

这两个注解的区别是：@Autowired 默认按类型装配，@Resource默认按名称装配，当找不到与名称匹配的bean才会按类型装配。

@Autowired注解是按类型装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它required属性为false

@Autowired是spring自己包中的类

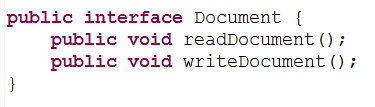
### @Qualifier

@Qualifier注解是按名称装配, 是spring自己包中的类。

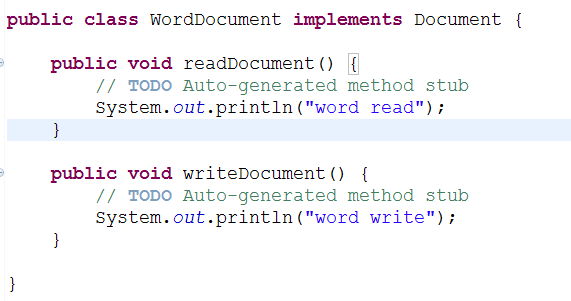
# Spring IOC和DI的意义

IOC和DI结合的真正的意义：java代码端完全的面向接口编程

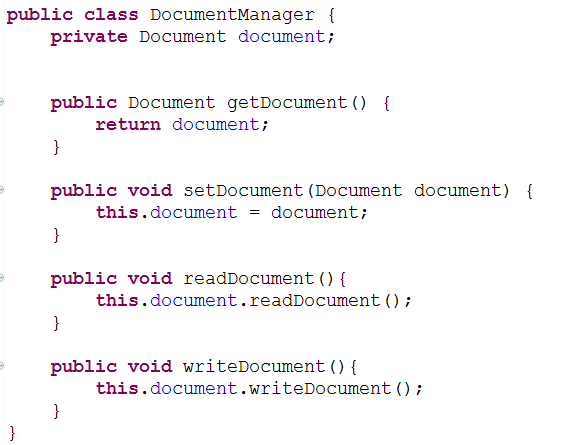
Document:



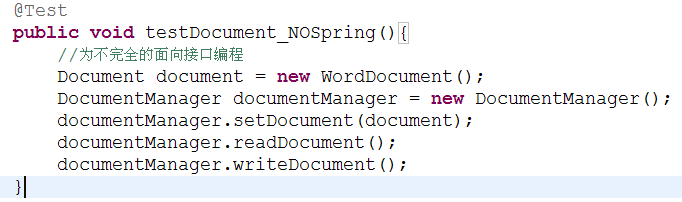
WordDocument:



DocumentManager:



## 不使用spring（为不完全的面向接口编程）：



## 使用spring（完全的面向接口编程）：

<!--

把documentManager,wordDocument,excelDocument,pdfDocument放入到spring容器中

-->

<bean id=*"documentManager"* class=*"com.itheima12.spring.iocdi.document.DocumentManager"*>

<!--

该属性是一个接口

-->

<property name=*"document"*>

<ref bean=*"wordDocument"*/>

</property>

</bean>

<bean id=*"wordDocument"* class=*"com.itheima12.spring.iocdi.document.WordDocument"*></bean>

<bean id=*"excelDocument"* class=*"com.itheima12.spring.iocdi.document.ExcelDocument"*></bean>

<bean id=*"pdfDocument"* class=*"com.itheima12.spring.iocdi.document.PDFDocument"*></bean>



## 在项目中的实际应用（面向接口编程）：

<?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-2.5.xsd"*>

<bean id=*"personDao"* class=*"com.itheima12.spring.dao.impl.PersonDaoImpl"*></bean>

<bean id=*"personService"* class=*"com.itheima12.spring.service.impl.PersonServiceImpl"*>

<property name=*"personDao"*>

<ref bean=*"personDao"*/>

</property>

</bean>

<bean id=*"personAction"* class=*"com.itheima12.spring.action.PersonAction"*>

<property name=*"personService"*>

<ref bean=*"personService"*/>

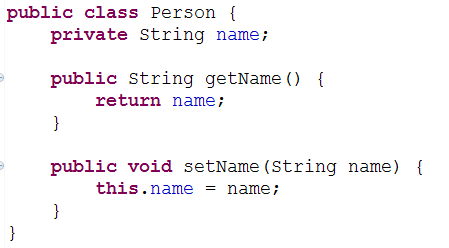
</property>

</bean>

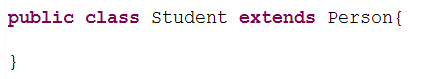
</beans>

# spring中的继承

Person:



Student:



配置文件：

## 在父类中赋值

<?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-2.5.xsd"*>

<!--

在父类中进行赋值,子类通过parent属性继承父类的内容

-->

<bean id=*"person"* class=*"com.itheima12.spring.extend.Person"*>

<property name=*"name"* value=*"aaa"*></property>

</bean>

<!--

parent 实现了spring容器内部的继承关系

-->

<bean id=*"student"* class=*"com.itheima12.spring.extend.Student"* parent=*"person"*></bean>

</beans>

## 在子类中赋值

<?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-2.5.xsd"*>

<!--

因为java的继承机制，子类继承了父类的setXxx方法，所以子类可以利用setXxx方法注入值

-->

<bean id=*"person2"* class=*"com.itheima12.spring.extend.Person"*>

</bean>

<bean id=*"student2"* class=*"com.itheima12.spring.extend.Student"*>

<property name=*"name"* value=*"bbb"*></property>

</bean>

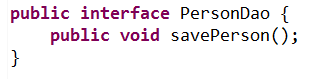
</beans>

# Spring AOP:

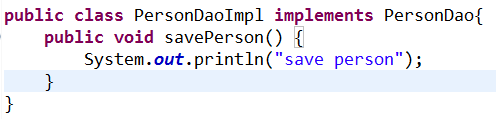
## 代理模式

### 静态代理

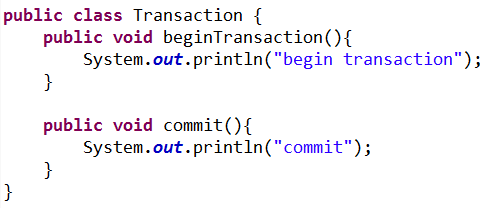
PersonDao:



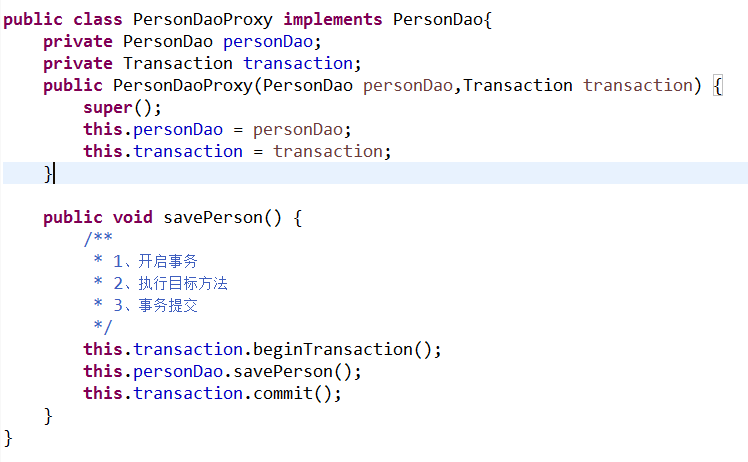
PersonDaoImpl:



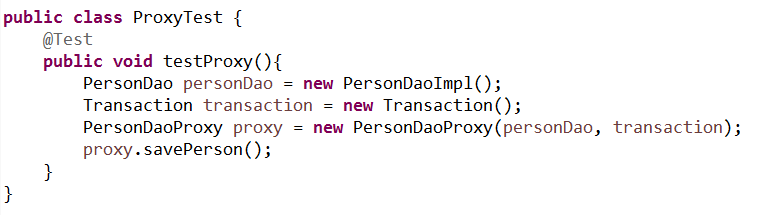
事务类:



代理类:



客户端调用：



**静态代理的缺点:**

**1、静态代理模式并没有做到事务的重用（每个方法中还要自己手动写事务的开启和提交）**

**2、假设dao层有100个类，就需要有100个proxy，接口中有多少方法，在proxy层就得实现多少方法，有多少方法，就要开启和提交多少事务**

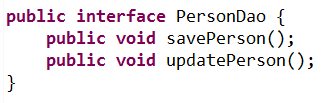
**3、如果一个proxy实现了多个接口，如果其中的一个接口发生变化(添加了一个方法),那么proxy也要做相应的改变**

观察代码可以发现每一个代理类只能为一个接口服务，这样一来程序开发中必然会产生过多的代理，而且，所有的代理操作除了调用的方法不一样之外，其他的操作都一样，则此时肯定是重复代码。

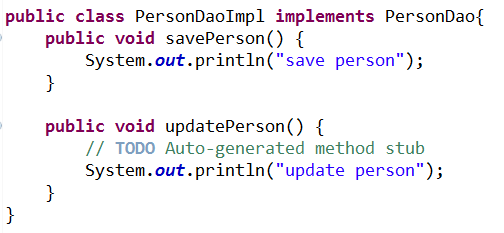
解决这一问题最好的做法是可以通过一个代理类完成全部的代理功能，那么此时就必须使用动态代理完成。

### JDK动态代理

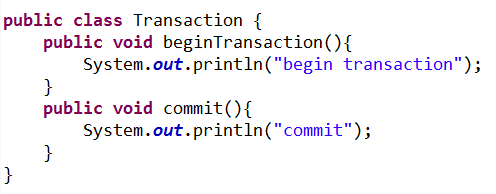
PersonDao:



PersonDaoImpl:



Transaction:



拦截器类：

/\*\*

\* 拦截器

\* 1、目标类导入进来

\* 2、事务导入进来

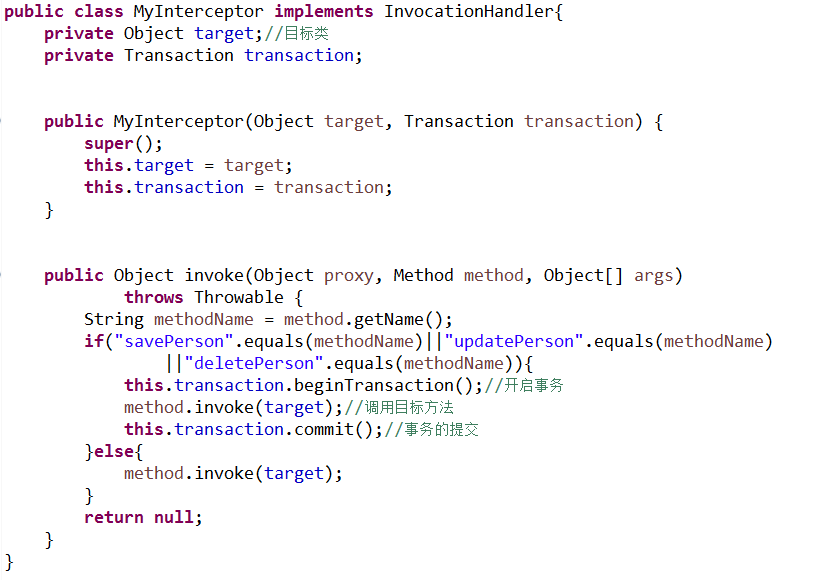
\* 3、invoke完成

\* 1、开启事务

\* 2、调用目标对象的方法

\* 3、事务的提交

\*/



客户端：



/\*\*

\* 1、拦截器的invoke方法是在什么时候执行的？

\* 当在客户端，代理对象调用方法的时候，进入到了拦截器的invoke方法

\* 2、代理对象的方法体的内容是什么？

\* 拦截器的invoke方法的内容就是代理对象的方法的内容

\* 3、拦截器中的invoke方法中的参数method是谁在什么时候传递过来的？

\* 代理对象调用方法的时候，进入了拦截器中的invoke方法，所以invoke方法中的参数method就是代理对象调用的方法

\*/

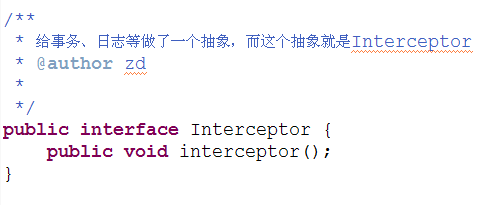
动态代理的缺点:

1、在拦截器中除了能调用目标对象的目标方法以外，功能是比较单一的，在这个例子中只能处理事务

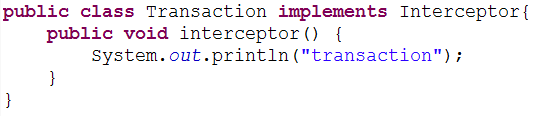
2、拦截器中的invoke方法的if判断语句在真实的开发环境下是不靠谱的，因为一旦方法很多if语句需要写很多。

针对第一个缺点，可以做一些改进：

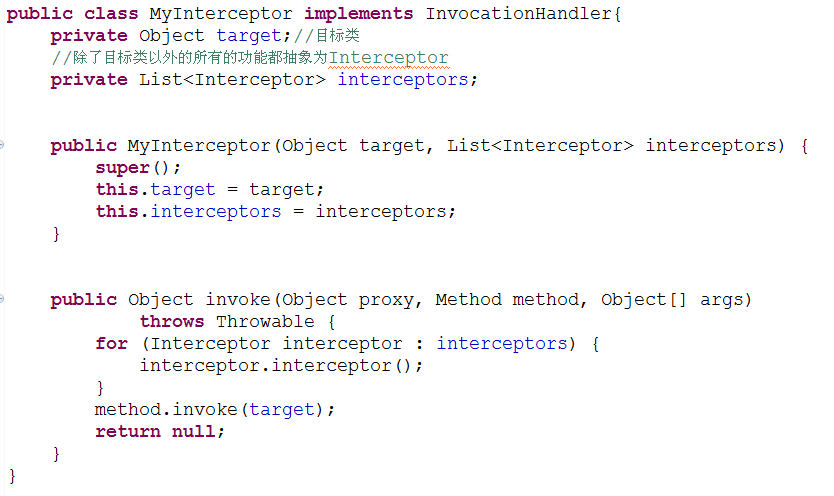
Interceptor:



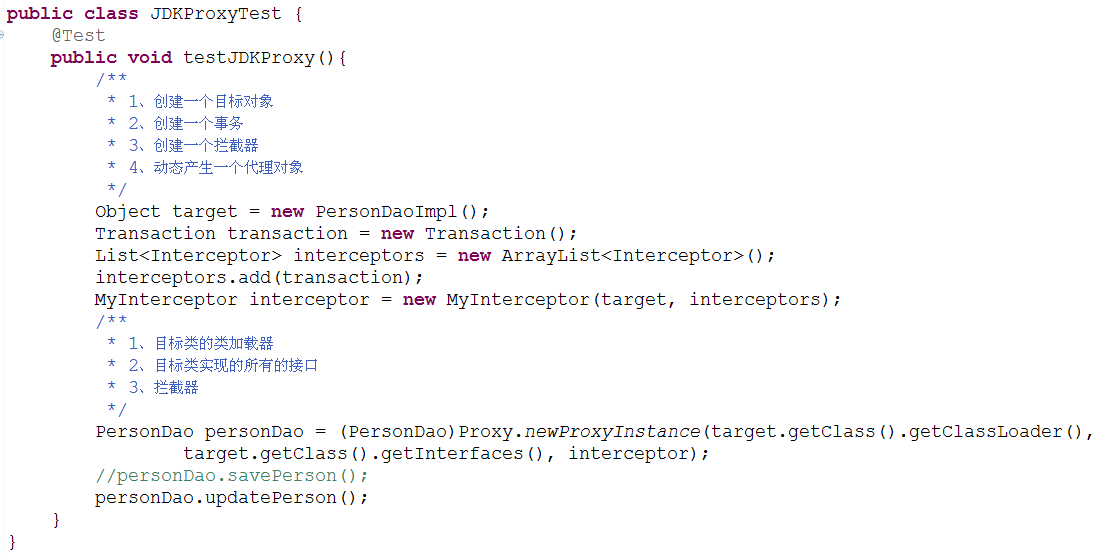
Transaction:



拦截器：



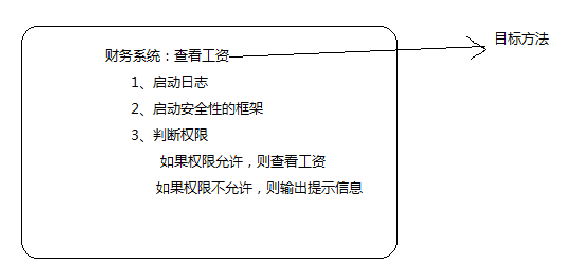
客户端：



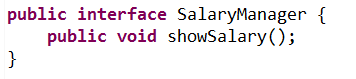
但是这种改进还是没有彻底解决问题，因为有的Interceptor是在目标方法执行之前执行，有的Interceptor是在目标方法执行之后执行，还有时可能只执行Interceptor不执行目标方法，所以还需要进一步的优化，完善

针对第二个缺点，只能使用SpringAOP来解决了

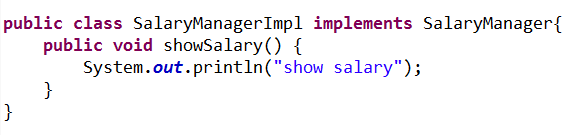
JDK动态代理实例：查看工资



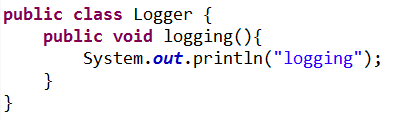
接口：



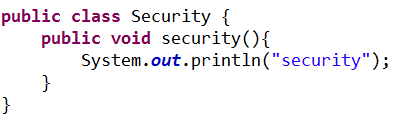
目标方法：



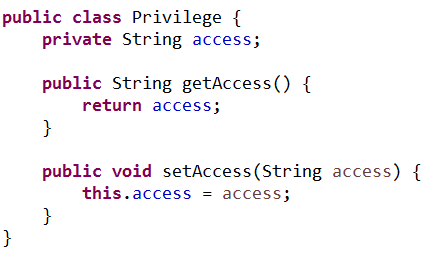
日志：



安全性类：



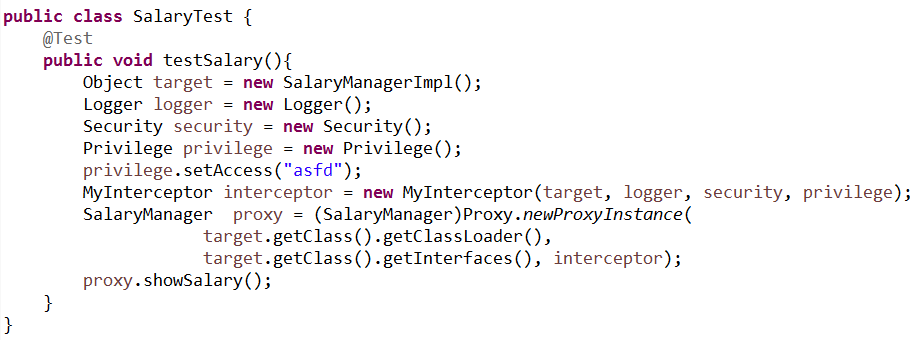
权限：



拦截器类：

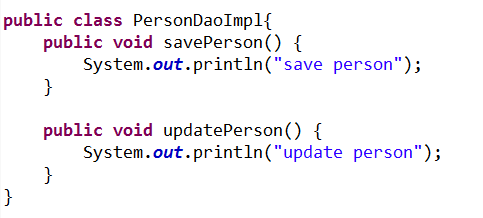


客户端：

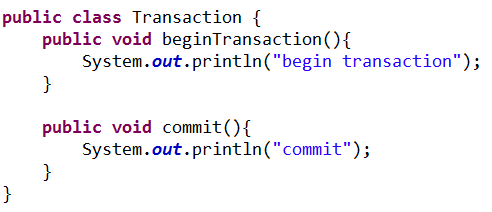
****

### CGLIB代理

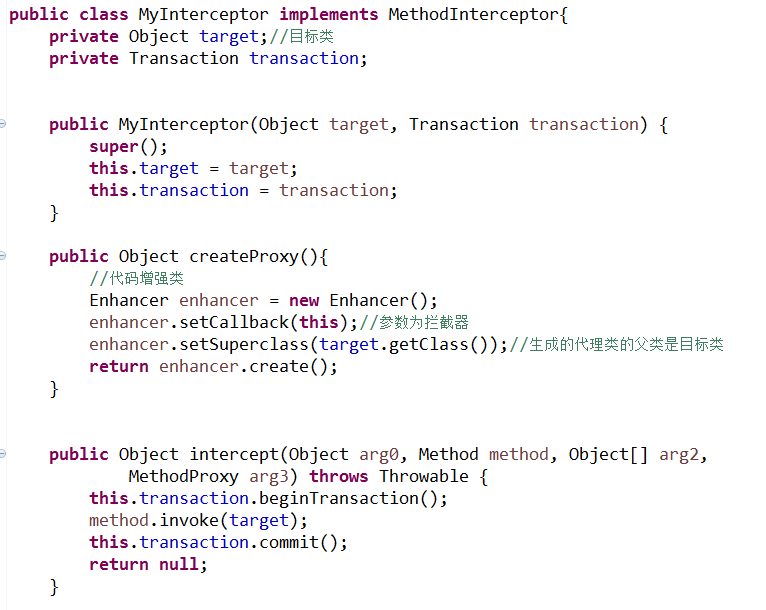
目标类：



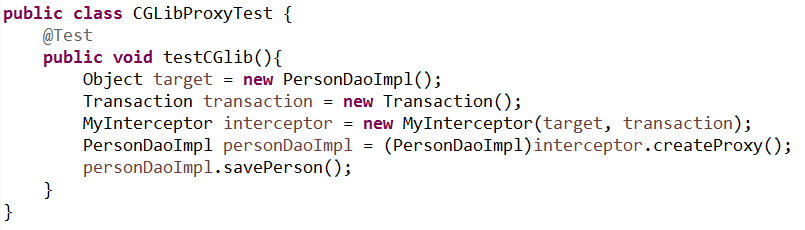
事务：



拦截器类：



客户端：



/\*\*

\* 通过cglib产生的代理对象，代理类是目标类的子类

\*/

总结:

1. CGlib是一个强大的,高性能,高质量的Code生成类库。它可以在运行期扩展Java类与实现Java接口。
2. 用CGlib生成的代理类是目标类的子类。
3. 用CGlib生成 代理类不需要接口
4. 用CGLib生成的代理类重写了父类的各个方法。
5. 拦截器中的intercept方法内容正好就是代理类中的方法体

## AOP

### 概念

切面：除了目标类以外的类都是切面

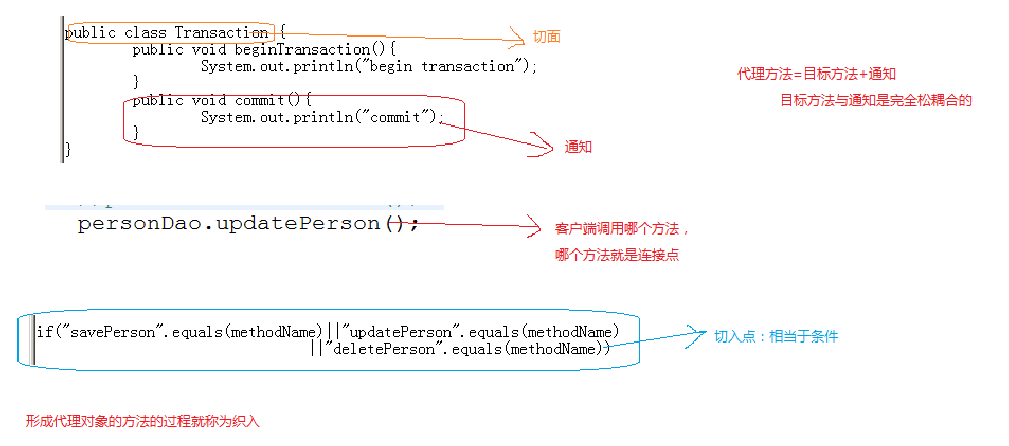
通知：切面中的方法

连接点：客户端调用哪个方法，哪个方法就是连接点

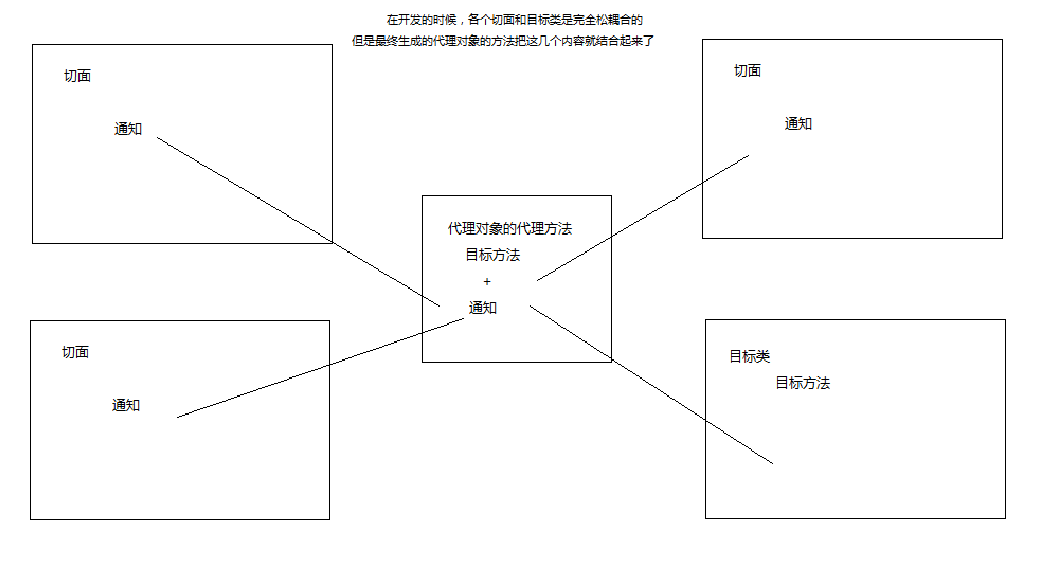
切入点：要对些拦截的方法的定义

目标对象：代理的目标对象

织入：形成代理对象的方法的过程



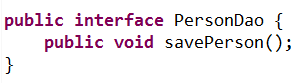
AOP的意义：



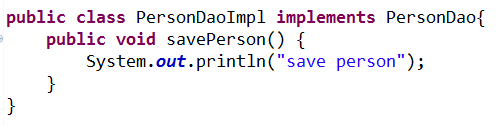
### AOP实现的两种模式

#### xml形式

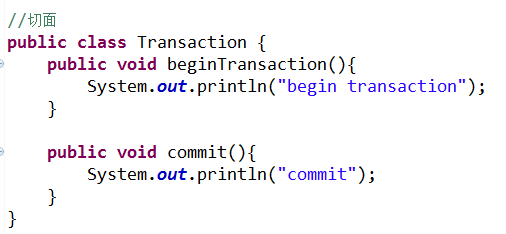
接口：



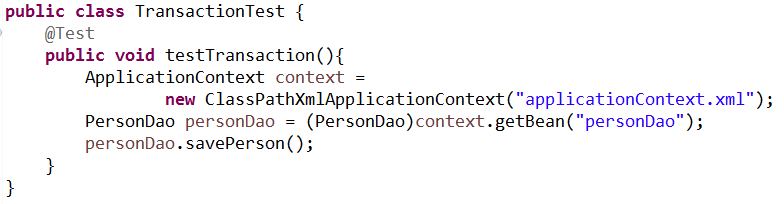
目标类：



切面：



客户端：



配置文件：

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

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

xmlns:aop=*"http://www.springframework.org/schema/aop"*

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-2.5.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"*>

<bean id=*"personDao"* class=*"com.itheima12.spring.aop.xml.transaction.PersonDaoImpl"*></bean>

<bean id=*"transaction"* class=*"com.itheima12.spring.aop.xml.transaction.Transaction"*></bean>

<aop:config>

<!--

切入点表达式 确定目标类

-->

<aop:pointcut

expression=*"execution(\* com.itheima12.spring.aop.xml.transaction.PersonDaoImpl.\*(..))"*

id=*"perform"*/>

<!--

ref指向的对象就是切面

-->

<aop:aspect ref=*"transaction"*>

<aop:before method=*"beginTransaction"* pointcut-ref=*"perform"*/>

<aop:after-returning method=*"commit"* pointcut-ref=*"perform"*/>

</aop:aspect>

</aop:config>

</beans>

/\*\*

\* 原理：

\* 1、当spring容器启动的时候，加载两个bean,对两个bean进行实例化

\* 2、当spring容器对配置文件解析到<aop:config>的时候

\* 3、把切入点表达式解析出来，按照切入点表达式匹配spring容器内容的bean

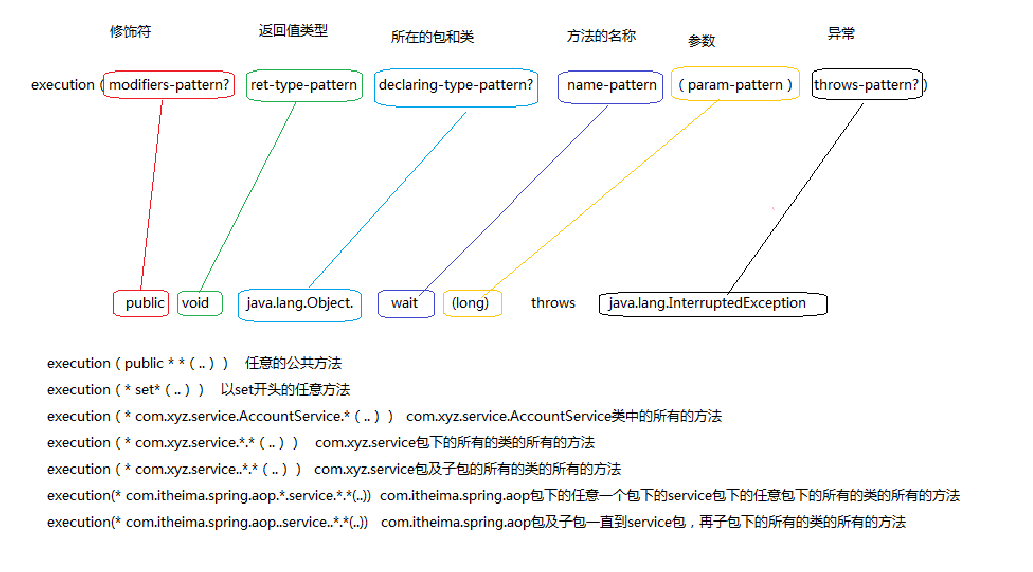
\* 4、如果匹配成功，则为该bean创建代理对象

\* 5、当客户端利用context.getBean获取一个对象时，如果该对象有代理对象，则返回代理对象

\* 如果没有代理对象，则返回对象本身

\*/

切入点表达式:



各种通知：

1. 前置通知：

<!--

前置通知

1、在目标方法执行之前

2、获取不到目标方法的返回值

-->

<aop:before method=*"beginTransaction"* pointcut-ref=*"perform"*/>



参数：JoinPoint point （连接点） 可以获得目标类、目标方法和参数值

1. 后置通知：

<!--

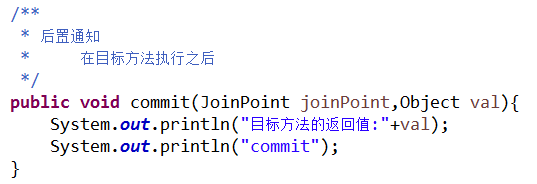
后置通知

1、后置通知可以获取到目标方法的返回值

2、当目标方法抛出异常，后置通知将不再执行

-->

<aop:after-returning method=*"commit"* pointcut-ref=*"perform"* returning=*"val"*/>



有两个参数： JoinPoint point 可以获得目标方法和参数值

Object val 这里的名字要和returning=”val”中保持一致，指的是目标方法的返回值。

1. 最终通知：

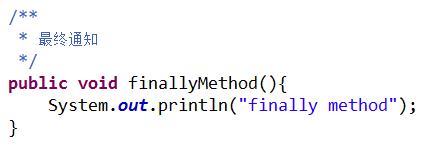
<!--

最终通知

无论目标方法是否抛出异常都将执行

-->

<aop:after method=*"finallyMethod"* pointcut-ref=*"perform"*/>



参数：JoinPoint point （连接点） 可以获得目标类、目标方法和参数值

1. 异常通知：

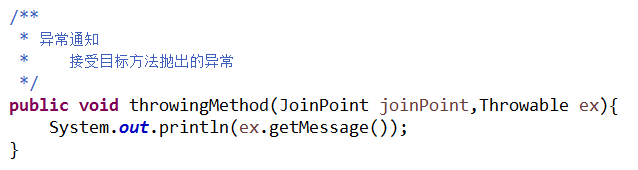
<!--

异常通知

接受目标方法抛出的异常

-->

<aop:after-throwing method=*"throwingMethod"* throwing=*"ex"* pointcut-ref=*"perform"*/>



方法中有两个参数： JoinPoint point 可以获得方法的名称、参数

Throwable ex 利用ex.getMessage()可以获得异常信息

1. 环绕通知：

<!--

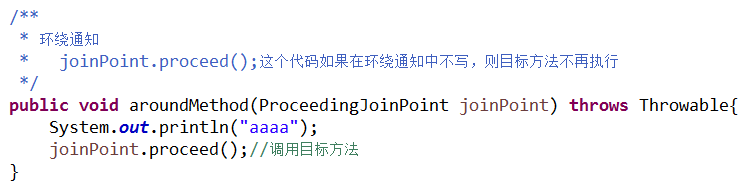
环绕通知

能控制目标方法的执行

前置通知和后置通知能在目标方法的前面和后面加一些代码，但是不能控制目标方法的执行

-->

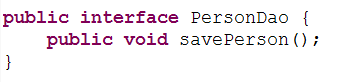
<aop:around method=*"aroundMethod"* pointcut-ref=*"perform"*/>



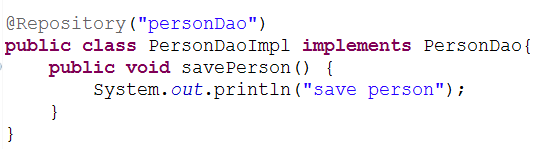
参数类型为ProceedingJoinPoint。ProceedingJoinPoint的proceed方法相当于invoke方法，调用目标类的目标方法。ProceedingJoinPoint继承了JoinPoint类

#### 注解形式

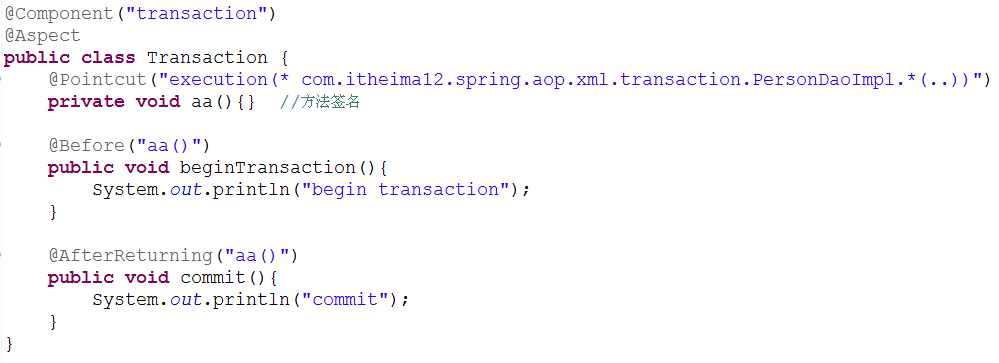
接口：



目标类：



切面：



配置文件：

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

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

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

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-2.5.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-2.5.xsd"*>

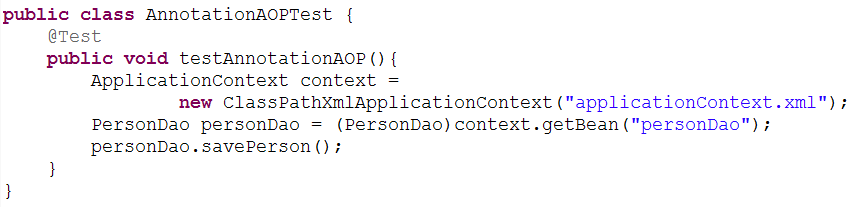
<context:component-scan base-package=*"com.itheima12.spring.aop.xml.transaction"*>

</context:component-scan>

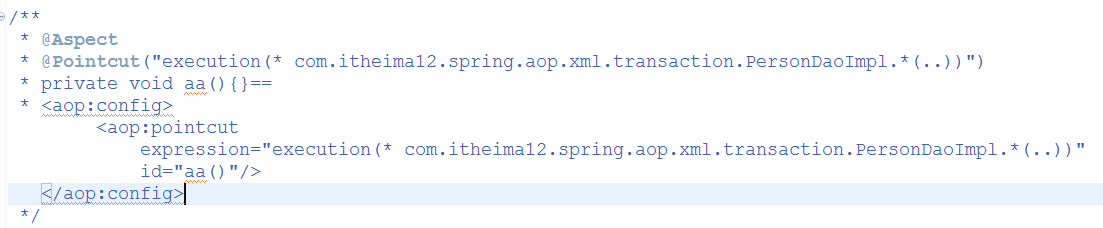
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

</beans>

客户端：

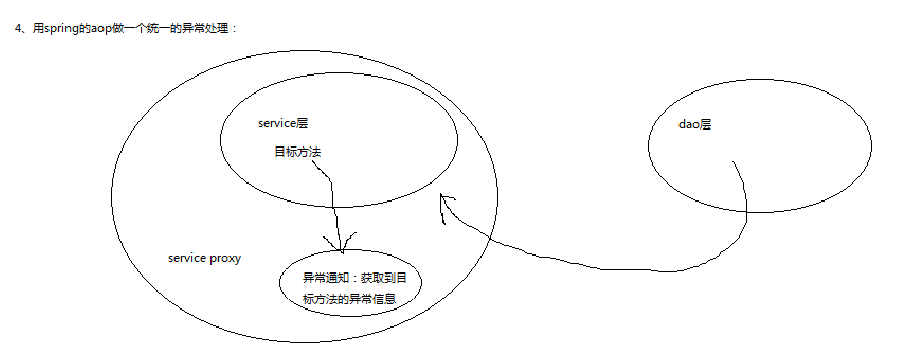


原理：

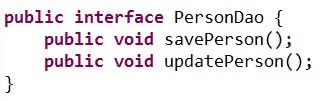


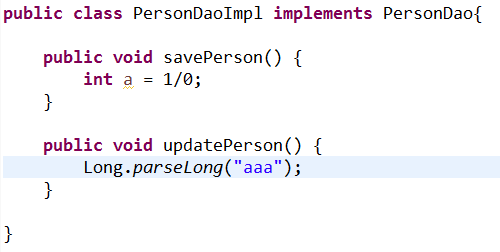
### 实例

#### 统一异常处理

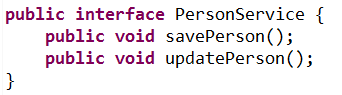


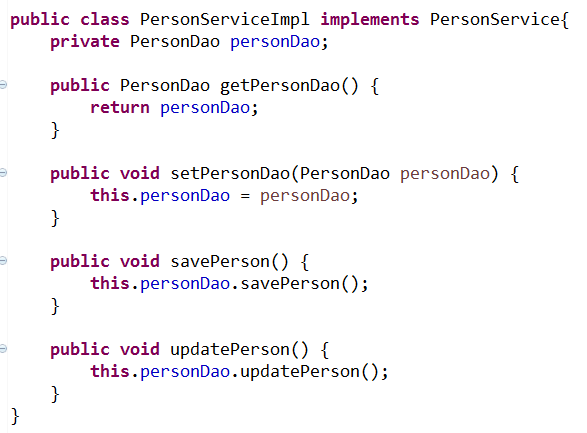
Dao层：





Service层：





异常处理（切面）：



配置文件：

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

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

xmlns:aop=*"http://www.springframework.org/schema/aop"*

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-2.5.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"*>

<bean id=*"personDao"* class=*"com.itheima12.spring.aop.xml.exception.dao.impl.PersonDaoImpl"*></bean>

<bean id=*"personService"* class=*"com.itheima12.spring.aop.xml.exception.service.impl.PersonServiceImpl"*>

<property name=*"personDao"*>

<ref bean=*"personDao"*/>

</property>

</bean>

<bean id=*"studentDao"* class=*"com.itheima12.spring.aop.xml.exception.dao.impl.StudentDaoImpl"*></bean>

<bean id=*"studentService"* class=*"com.itheima12.spring.aop.xml.exception.service.impl.StudentServiceImpl"*>

<property name=*"studentDao"*>

<ref bean=*"studentDao"*/>

</property>

</bean>

<bean id=*"exceptionAspect"*

class=*"com.itheima12.spring.aop.xml.exception.aspect.ExceptionAspect"*></bean>

<aop:config>

<aop:pointcut

expression=*"execution(\* com.itheima12.spring.aop.xml.exception.service.impl.\*.\*(..))"*

id=*"perform"*/>

<aop:aspect ref=*"exceptionAspect"*>

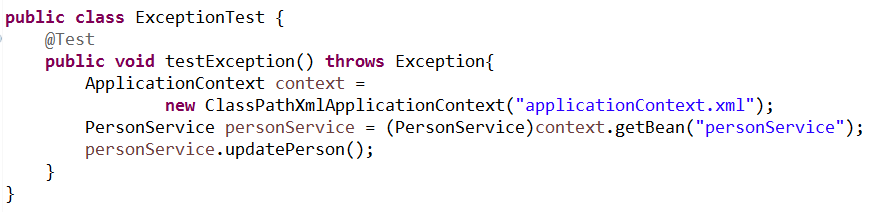
<aop:after-throwing method=*"throwingExcetion"* pointcut-ref=*"perform"* throwing=*"ex"*/>

</aop:aspect>

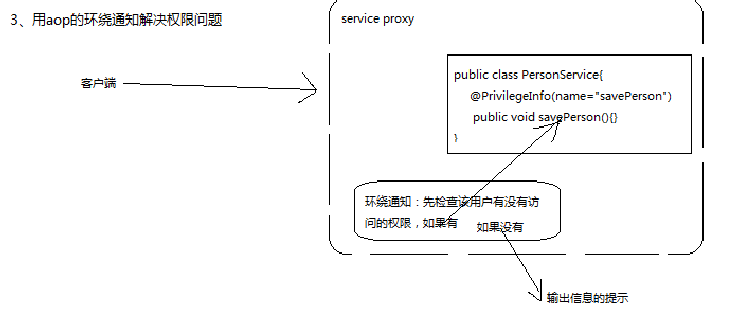
</aop:config>

</beans>

客户端：



#### 用aop解决权限问题



步骤：1、创建service层的类和接口

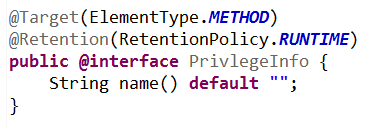
2、自定义一个注解

3、自定义一个注解解析器

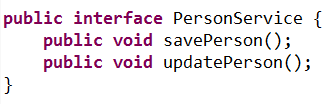
4、写一个权限类Privilege

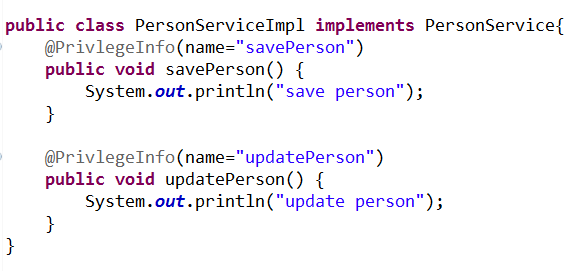
5、写一个切面，写一个环绕通知

自定义注解：



Service层：

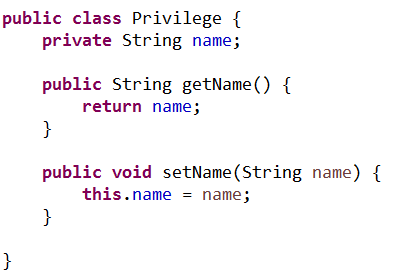


、

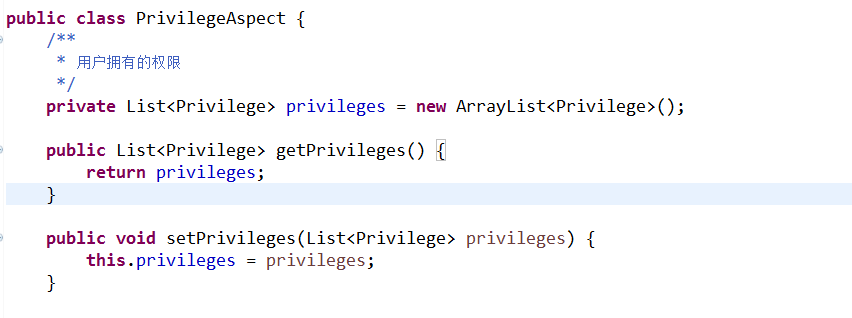
自定义注解解析器：



权限类：



切面：





配置文件：

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

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

xmlns:aop=*"http://www.springframework.org/schema/aop"*

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-2.5.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"*>

<bean id=*"personService"* class=*"com.itheima12.spring.aop.xml.privilege.service.impl.PersonServiceImpl"*></bean>

<bean id=*"privilegeAspect"* class=*"com.itheima12.spring.aop.xml.privilege.aspect.PrivilegeAspect"*></bean>

<aop:config>

<aop:pointcut

expression=*"execution(\* com.itheima12.spring.aop.xml.privilege.service.impl.\*.\*(..))"*

id=*"perform"*/>

<aop:aspect ref=*"privilegeAspect"*>

<aop:around method=*"isAccessMethod"* pointcut-ref=*"perform"*/>

</aop:aspect>

</aop:config>

</beans>

客户端:



#### 计算每一层方法的调用时间



步骤：计算action,servce,dao层每一个类的每一个方法的调用时间

目标类：

action层，service层，dao层的所有的类都是目标类

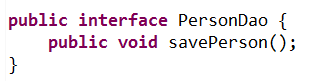
切面：

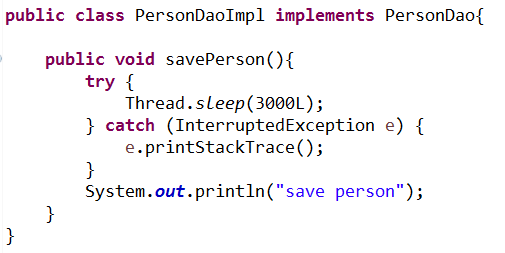
用环绕通知：

在方法调用之前，计算时间

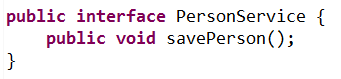
在方法调用之后，计算时间，两个时间做差

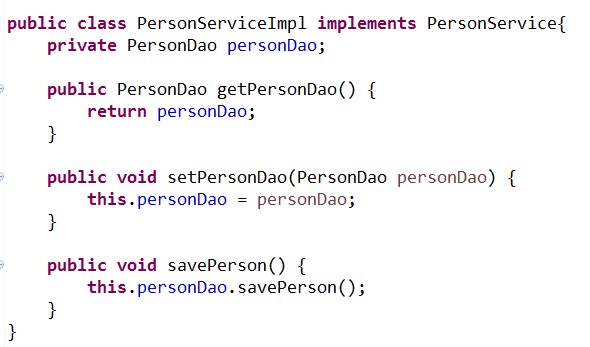
Dao：



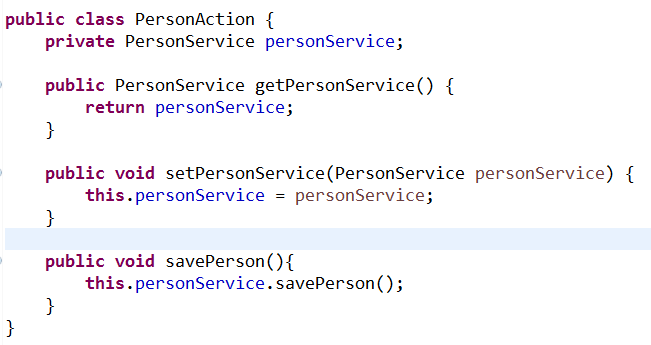


Service：

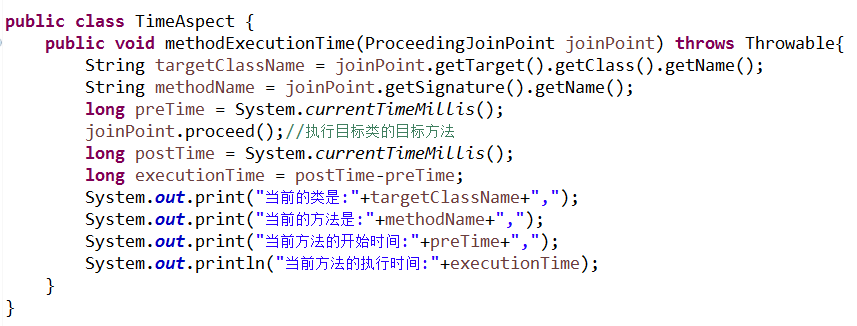




Action：



切面：



配置文件：

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

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

xmlns:aop=*"http://www.springframework.org/schema/aop"*

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-2.5.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"*>

<bean id=*"personDao"* class=*"com.itheima12.spring.aop.xml.time.dao.impl.PersonDaoImpl"*></bean>

<bean id=*"personService"* class=*"com.itheima12.spring.aop.xml.time.service.impl.PersonServiceImpl"*>

<property name=*"personDao"*>

<ref bean=*"personDao"*/>

</property>

</bean>

<bean id=*"personAction"* class=*"com.itheima12.spring.aop.xml.time.action.PersonAction"*>

<property name=*"personService"*>

<ref bean=*"personService"*/>

</property>

</bean>

<bean id=*"timeAspect"* class=*"com.itheima12.spring.aop.xml.time.aspect.TimeAspect"*></bean>

<aop:config>

<aop:pointcut

expression=*"execution(\* com.itheima12.spring.aop.xml.time..\*.\*(..))"*

id=*"perform"*/>

<aop:aspect ref=*"timeAspect"*>

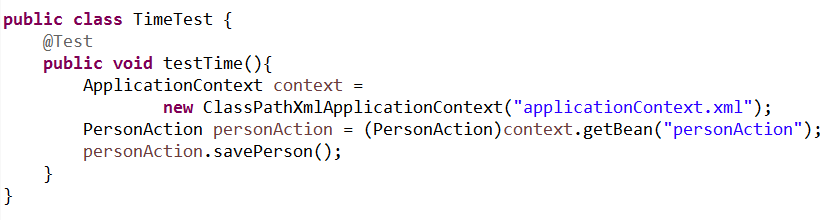
<aop:around method=*"methodExecutionTime"* pointcut-ref=*"perform"*/>

</aop:aspect>

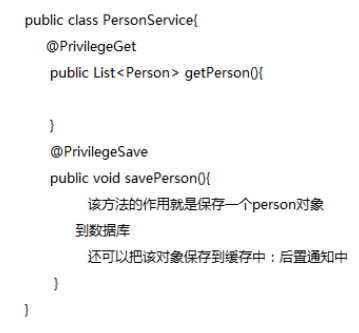
</aop:config>

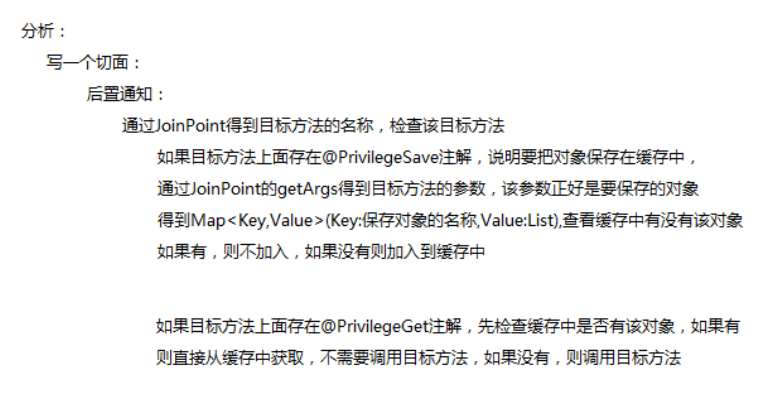
</beans>

客户端：



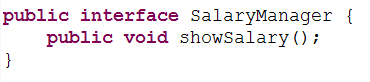
#### 缓存的应用



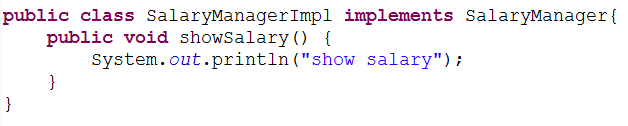


### 配置多个切面的情况

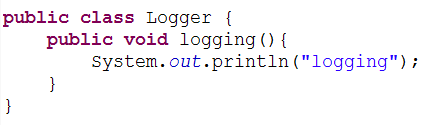
接口：

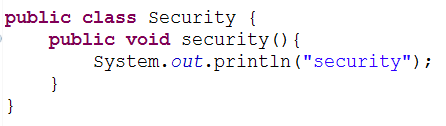


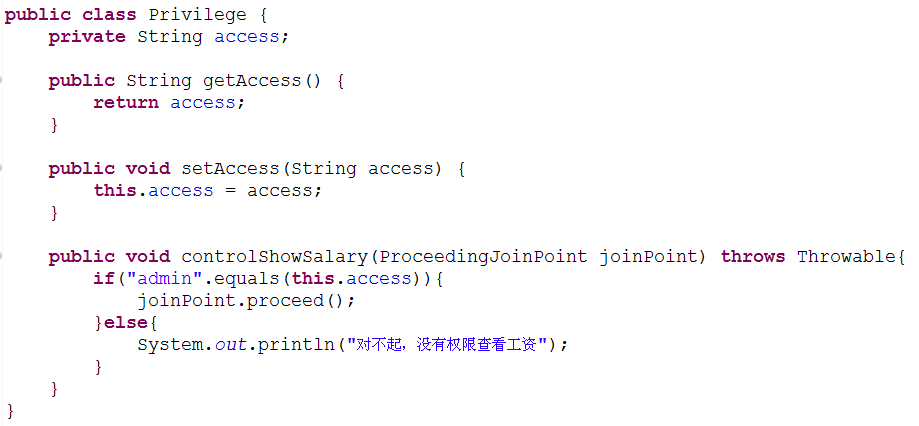
目标类：



切面：







配置文件：

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

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

xmlns:aop=*"http://www.springframework.org/schema/aop"*

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-2.5.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"*>

<bean id=*"salaryManager"* class=*"com.itheima12.aop.xml.aspects.SalaryManagerImpl"*></bean>

<bean id=*"logger"* class=*"com.itheima12.aop.xml.aspects.Logger"*></bean>

<bean id=*"security"* class=*"com.itheima12.aop.xml.aspects.Security"*></bean>

<bean id=*"privilege"* class=*"com.itheima12.aop.xml.aspects.Privilege"*>

</bean>

<aop:config>

<aop:pointcut

expression=*"execution(\* com.itheima12.aop.xml.aspects.SalaryManagerImpl.\*(..))"*

id=*"perform"*/>

<aop:aspect ref=*"logger"*>

<aop:before method=*"logging"* pointcut-ref=*"perform"*/>

</aop:aspect>

<aop:aspect ref=*"security"*>

<aop:before method=*"security"* pointcut-ref=*"perform"*/>

</aop:aspect>

<aop:aspect ref=*"privilege"*>

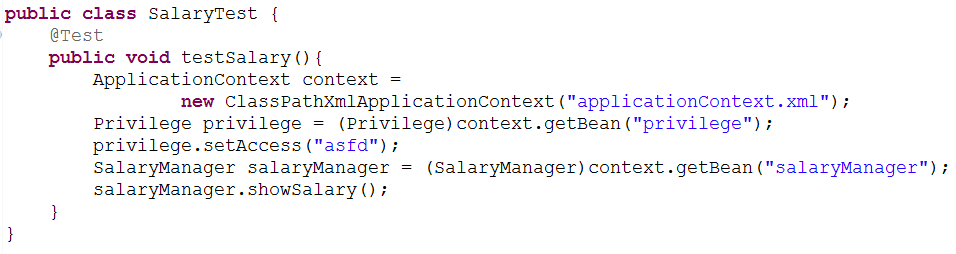
<aop:around method=*"controlShowSalary"* pointcut-ref=*"perform"*/>

</aop:aspect>

</aop:config>

</beans>

客户端：



### 注意事项

1、如果目标类实现了接口，则采用jdkproxy,如果目标类没有实现接口，则采用cglibproxy

而这些都是spring内部完成的

2、如果想要强制使用cglibproxy，则在aop:config下有一个属性proxy-target-class

该属性的默认值为false,如果改成true,则目标类的代理类肯定是由cglib产生的

<aop:config proxy-target-class=*"true"*>

有时候会遇到需要强制使用cglibproxy的情况，比如说目标类实现了对象序列化接口，这时会使用jdkproxy生成代理，但是对象序列化接口中没有方法，这时就会有问题了

### SpringAOP的意义

在Struts2中，就使用了AOP，Action Proxy就是Action的代理，Action就是目标类，Action中的方法就是目标方法，拦截器就是切面，拦截器中的方法就是通知。Struts2中的拦截器的意义就在于，它将很多项目中都经常会用到的功能都抽取出来了，成为一个拦截器，这样就不需要程序员来做了，只需要写一个配置文件就可以完成了

那么如何抽取切面呢？系统中除了核心业务逻辑之外的，被大部分模块使用的，经常使用到的不变的部分就可以抽取出来做成一个切面，里面的功能就是通知

在struts2中的拦截器正是这样做的，拦截器就是将大部分项目都会用到的功能抽取出来做成了一个拦截器，每个拦截器之间都是松耦合的，这样程序员不需要自己写了，只需要配置一下就可以使用了

# Spring数据库

## 引入DataSource

配置文件：

<!--

引入prperties配置文件

-->

<bean

class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<property name=*"locations"*>

<value>classpath:jdbc.properties</value>

</property>

</bean>

<bean id=*"dataSource"* destroy-method=*"close"*

class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

<property name=*"url"* value=*"${jdbc.url}"* />

<property name=*"username"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

</bean>

jdbc.properties:

jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc\:mysql\://localhost\:3306/itheima12\_hibernate

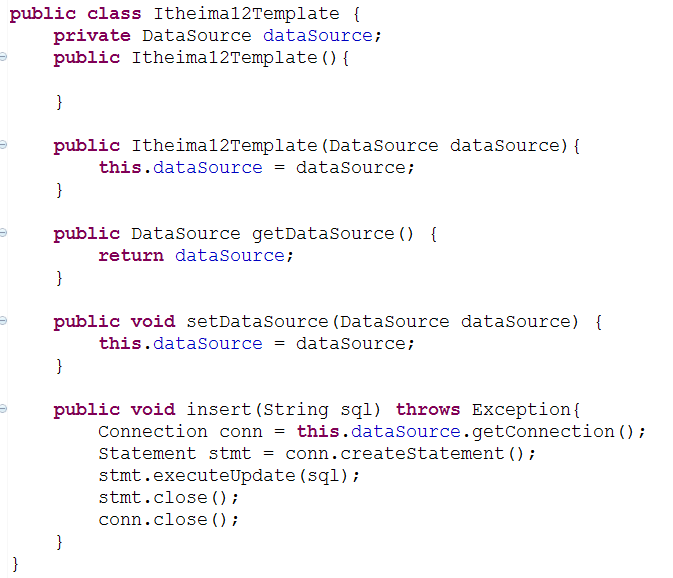
jdbc.username=root

jdbc.password=root

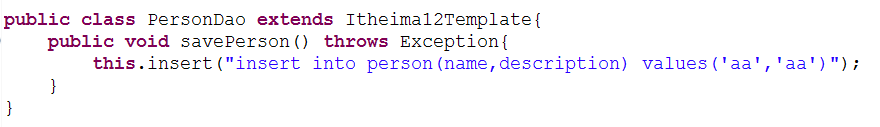
org.springframework.beans.factory.config.PropertyPlaceholderConfigurer是一个解析properties配置文件的类，只需要配置文件路径就可以读取了. Spring加载完这个bean之后，配置文件中的key和value就全部都读取出来了

## 自己模拟一个数据库框架与spring整合

Itheima12Template：



Dao：



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-2.5.xsd"*>

<!--

引入prperties配置文件

-->

<bean

class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<property name=*"locations"*>

<value>classpath:jdbc.properties</value>

</property>

</bean>

<bean id=*"dataSource"* destroy-method=*"close"*

class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

<property name=*"url"* value=*"${jdbc.url}"* />

<property name=*"username"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

</bean>

<bean id=*"personDao"* class=*"com.itheima12.spring.itheima12.PersonDao"*>

<property name=*"dataSource"*>

<ref bean=*"dataSource"*/>

</property>

</bean>

<bean id=*"itheima12Template"* class=*"com.itheima12.spring.itheima12.Itheima12Template"*>

<property name=*"dataSource"*>

<ref bean=*"dataSource"*/>

</property>

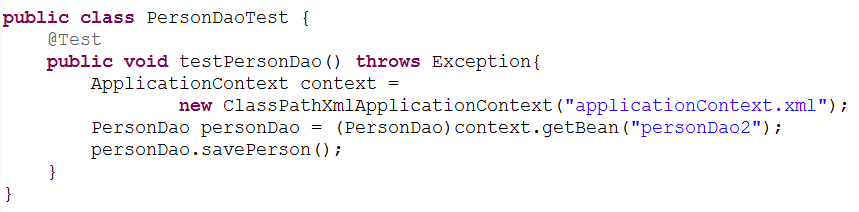
</bean>

<bean id=*"personDao2"* class=*"com.itheima12.spring.itheima12.PersonDao"*

parent=*"itheima12Template"*></bean>

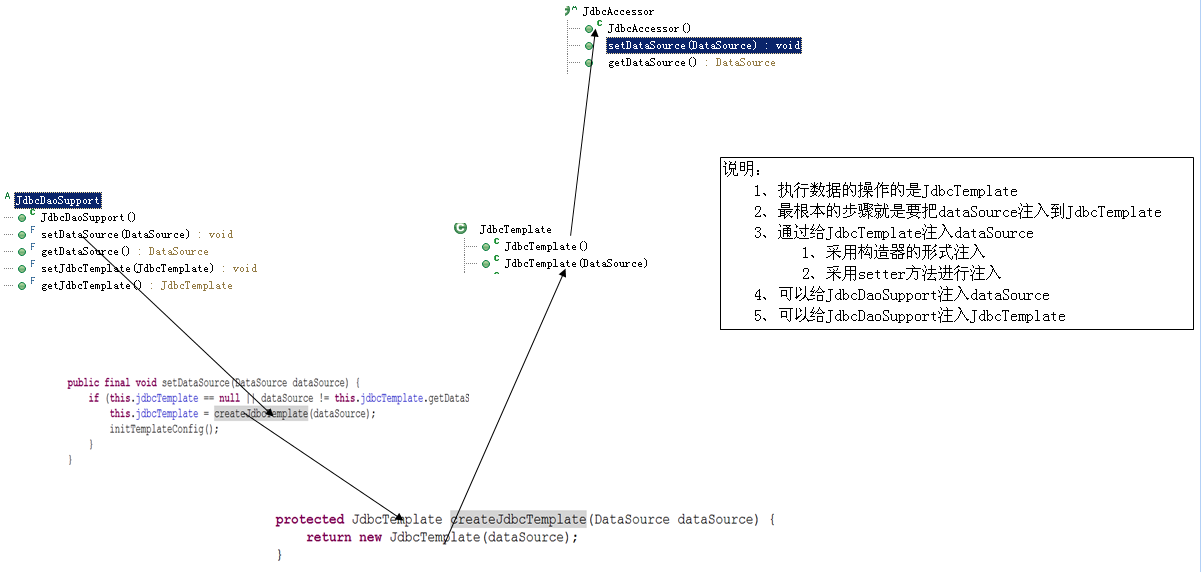
</beans>

客户端：

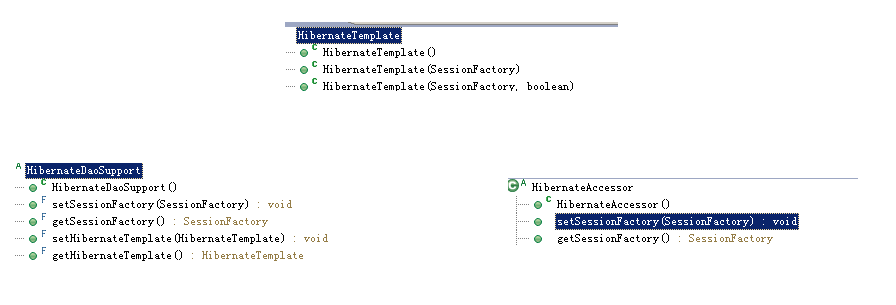


## Spring与数据库结合的架构

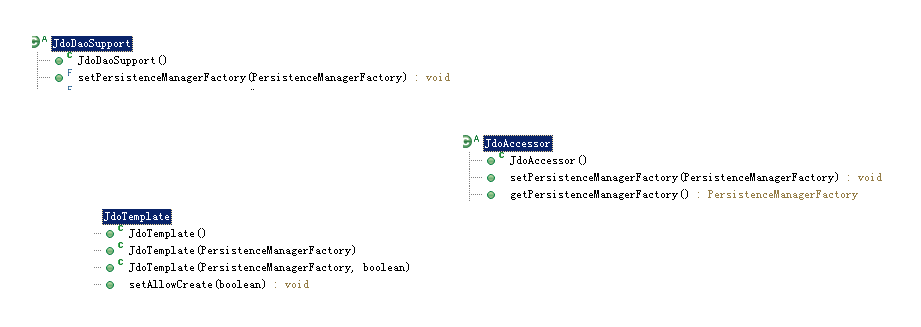
jdbc:



Hibernate:



Jdo:



## Spring+JDBC

### Jdbc编程特点

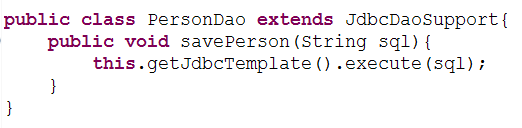
静态代码+动态变量 = jdbc编程。在spring中动态变量可以用注入的形式给予。这样的编程方式适合包装成模板。静态代码构成了模板，而动态变量则是需要传入的参数。

### 使用JdbcTemplate

在Dao类中，用JdbcTemplate作为属性，用spring对JdbcTemplate进行注入。再对JdbcTemplate进行DataSource注入。

### 继承JdbcDaoSupport

在Dao类中，继承JdbcDaoSupport。因为JdbcDaoSupport已经有了JdbcTemplate的引用，所以只要继承JdbcDaoSupport就相当于有了JdbcTemplate属性。



配置文件：

<bean id=*"personDao3"* class=*"com.itheima12.spring.jdbc.dao.PersonDao"*>

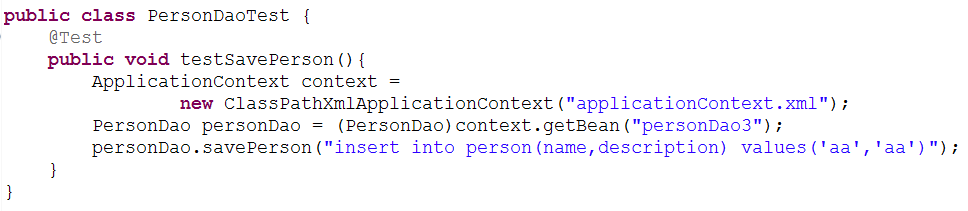
<property name=*"dataSource"*>

<ref bean=*"dataSource"*/>

</property>

</bean>

客户端：



## Spring+Hibernate

### 配置sessionFactory两种方式

第一种：设置hibernate配置文件的路径，通过hibernate配置文件的配置来处理

<bean id=*"sessionFactory1"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"configLocation"*>

<value>classpath:hibernate.cfg.xml</value>

</property>

</bean>

第二种:将hibernate配置文件中的内容在spring的配置文件中设置

<bean id=*"sessionFactory2"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<!--

映射文件所在的路径

-->

<property name=*"mappingDirectoryLocations"*>

<list>

<!--

spring容器会去该包及子包下搜索所有的映射文件

-->

<value>com/itheima12/spring/hibernate/transaction/domain</value>

</list>

</property>

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQL5InnoDBDialect</prop>

<prop key=*"hibernate.show\_sql"*>true</prop>

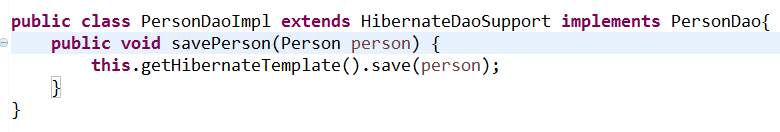
<prop key=*"hibernate.hbm2ddl.auto"*>update</prop>

</props>

</property>

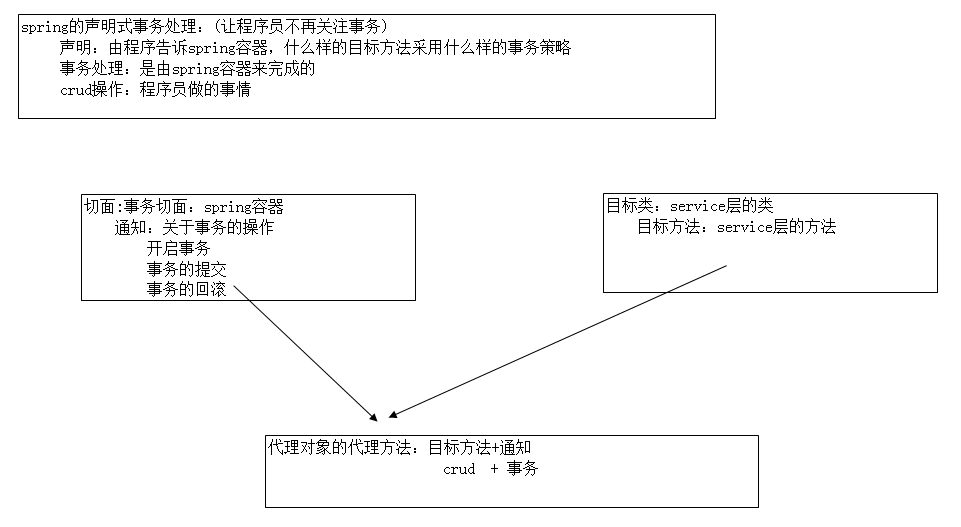
</bean>

### 继承HibernateDaoSupport



# 声明式事务处理

## 原理



## 配置：

<!--

事务管理器

-->

<bean id=*"transactionManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>

<property name=*"sessionFactory"*>

<ref bean=*"sessionFactory2"*/>

</property>

</bean>

<!--

事务的声明

-->

<tx:advice transaction-manager=*"transactionManager"* id=*"tx"*>

<tx:attributes>

<!--

read-only

true 只读事务

false 读写事务

-->

<tx:method name=*"save\*"* isolation=*"DEFAULT"* propagation=*"REQUIRED"*

read-only=*"false"*/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut

expression=*"execution(\* com.itheima12.spring.hibernate.transaction.service.impl.\*.\*(..))"*

id=*"perform"*/>

<aop:advisor advice-ref=*"tx"* pointcut-ref=*"perform"*/>

</aop:config>