Spring

# Spring介绍

## 1Spring基础概念

### 1.1 什么是Spring

Spring是分层的JavaSE/EE full-stack 轻量级开源框架

JavaEE中程序是被分为三层 WEB层【表现层】 、业务逻辑层、数据访问层【持久层】

\*\*\*\*Struts2:表现层 MVC的框架

\*\*\*\*Hibernate是数据访问层 完全的ORM框架

\*\*\*\*Spring框架 **包含了JavaEE三层的每一层解决方案 (一站式)**

\*\* Web层: SpringMVC

\*\*业务层 Spring对Bean管理、AOP、事务

\*\*持久层 SpringJdbcTempleate /ORM模块

### 1.2 Spring的核心

* 以IoC（Inverse of Control 反转控制）和AOP（Aspect Oriented Programming 面向切面编程为内核）
* IOC就是把创建对象的权限反转到Spring容器 由Spring进行管理对象的创建。

User user =new User();

User user =applicationContext.getBean(“xxx.xml”)； 通过配置文件进行反射的方式进行创对象

* AOP是面向切面的编程 OOP

DI：依赖注入

### 1.3Spring的由来

Rod Johson

* Spring的出现是为了取代EJB的臃肿、低效、脱离现实
  + *Expert One-to-One J2EE Design and Development （分析J2EE开发使用技术 EJB） 2002*
  + *Expert One-to-One J2EE Development without EJB (客观分析J2EE开发需要什么 推出了一个全新的框架 就是现在的Spring框架) 2004*

Spring 的出现就是解决JavaEE开发实际问题，轻量级的 相比EJB比较繁重 复杂的解决方案。

### 1.4Spring的优势

* 方便解耦，简化开发
  + Spring就是一个大工厂，可以将所有对象创建和依赖关系维护，交给Spring管理
* AOP编程的支持
  + Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能
* 声明式事务的支持
  + 只需要通过配置就可以完成对事务的管理，而无需手动编程
* 方便程序的测试
  + Spring对Junit4支持，可以通过注解方便的测试Spring程序
* 方便集成各种优秀框架
  + Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的直接支持
* 降低JavaEE API的使用难度
  + Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低

## 2.第一个Spring入门程序

### 2.1准备开发环境

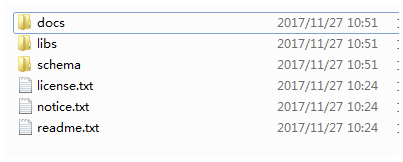
#### 2.1.1添加相关的开发jar包

**1）下载最新的jar包**

http://repo.spring.io/simple/libs-release-local/org/springframework/spring/

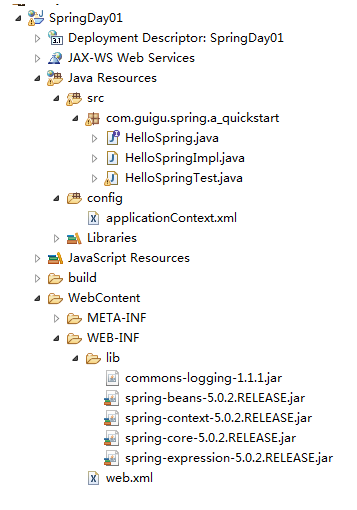
目前最新版本是2017.11.27发布的 5.0.2.RELEASE

2）Spring的目录结构

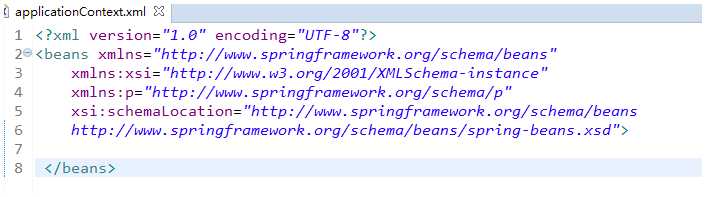


3）将Spring开发的基本jar包导入到工程内

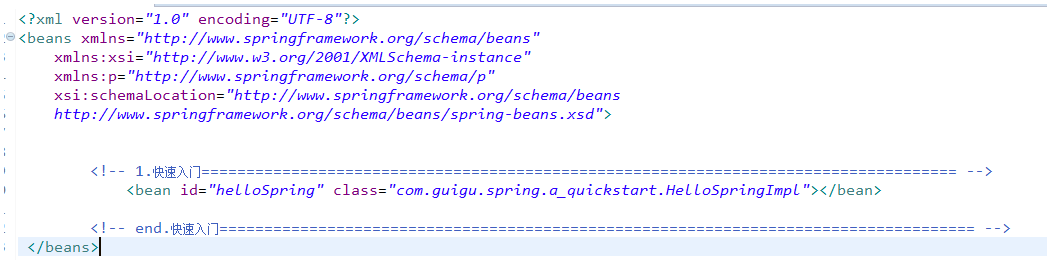
Beans/Core /context/ el /logging



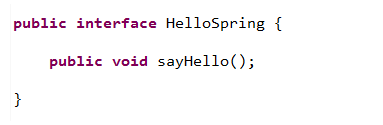
### 2.2准备配置文件

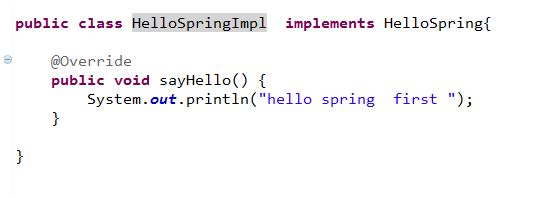


### 2.3编写配置文件信息



### 2.4编写类和接口

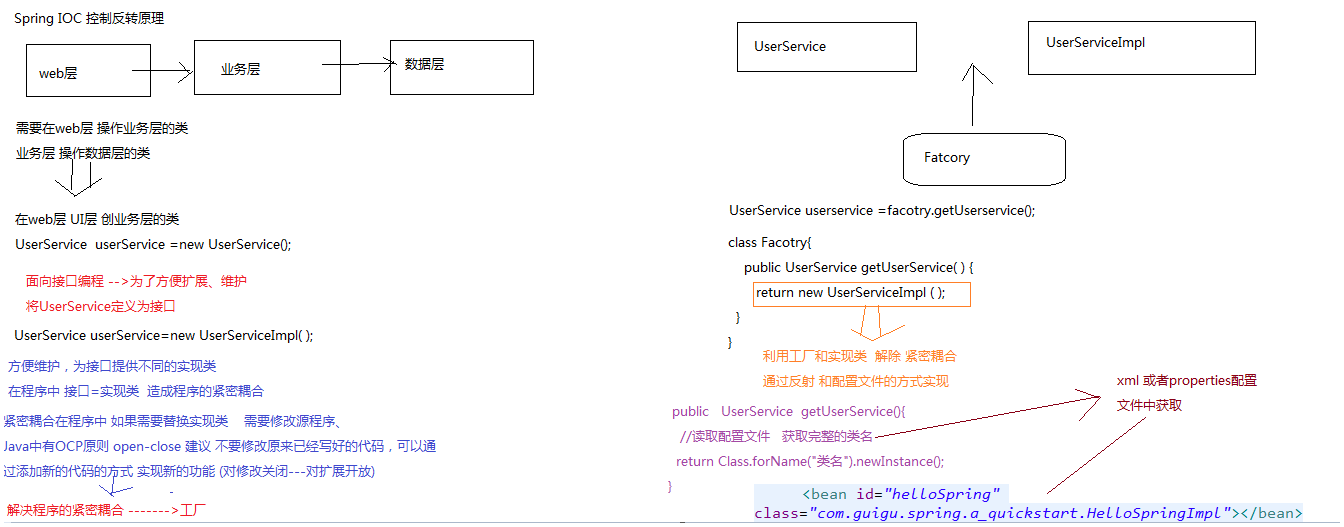




### 2.5编写测试类



### 2.6IOC的原理解释



# 二、IOC容器

IOC容器装配Bean 两种形式 第一种是基于xml配置文件的形式 第二种是基于注解的形式。

## 2.1 IOC装配Bean (XML方式)

Sring装配Bean的三种形式

1. 使用构造函数(有无参的构造函数)

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

2）使用静态工厂方法

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

3）使用实例工厂方法

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

<!-- inject any dependencies required by this locator bean -->

</bean>

<!-- the bean to be created via the factory bean -->

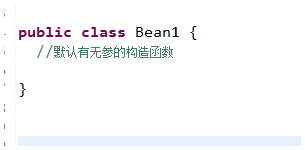
<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

### 2.1.1使用构造函数装配Bean

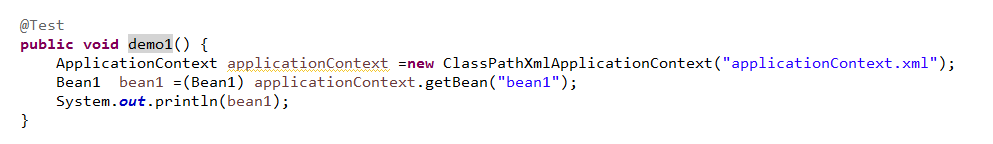
**类**



**配置文件**

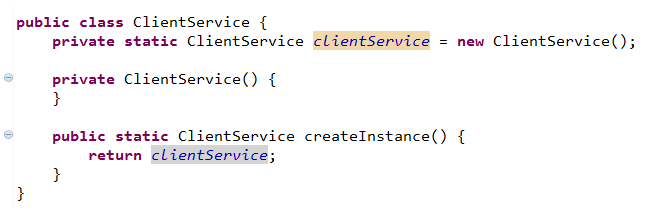


**测试类**



### 2.1.2使用静态工厂方法装配Bean

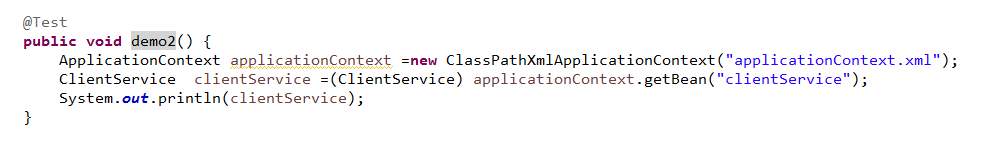
类



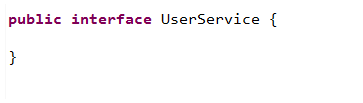
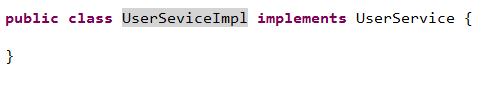
配置文件

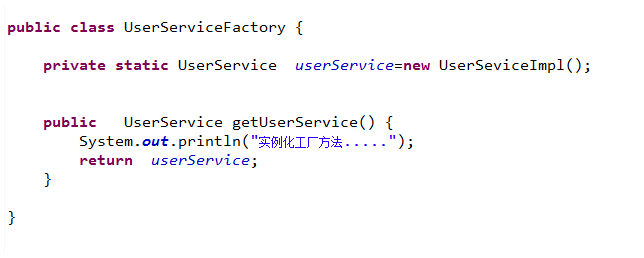


测试类

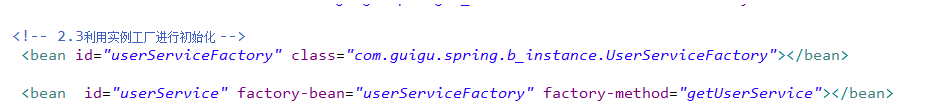


### 2.1.3使用实例化工厂方法

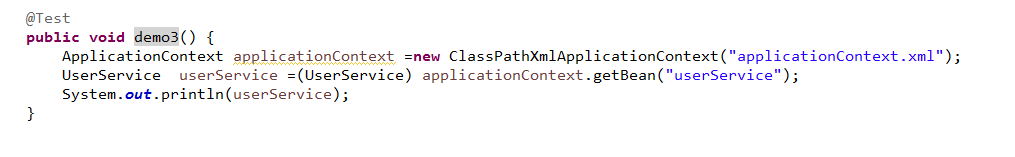
 



配置文件

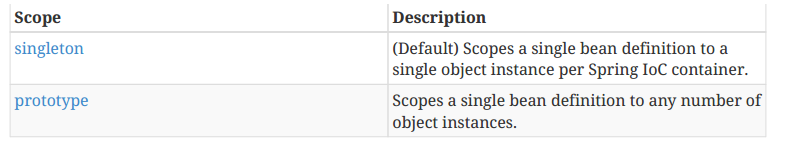


测试代码

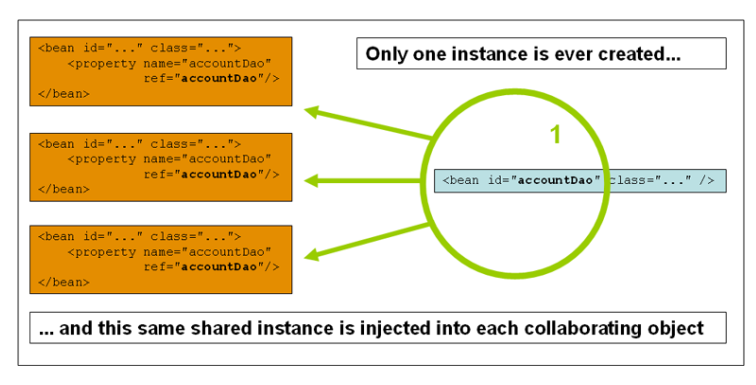


## 2.2 Bean的其他属性的设置

### 2.2.1 Bean的作用域



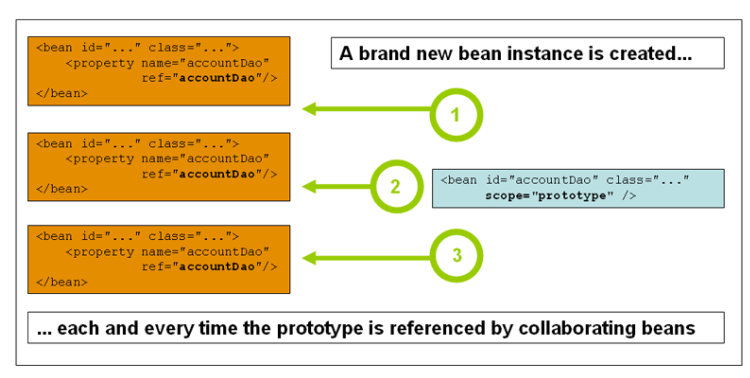
#### 2.2.1.1单例模式



**<bean id="accountService" class="com.foo.DefaultAccountService"/>**

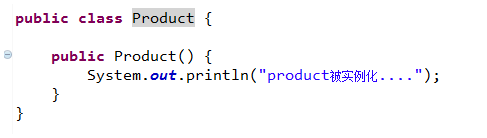
**<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>**

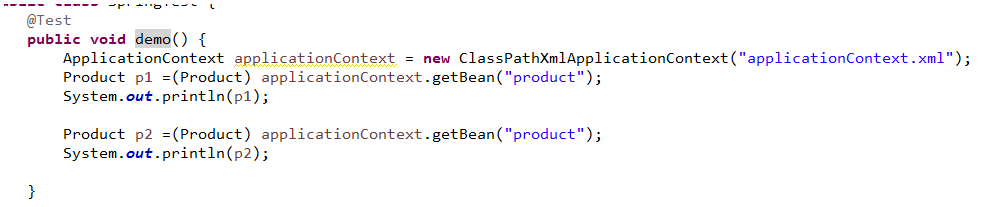
#### 2.2.1.2 多例模式



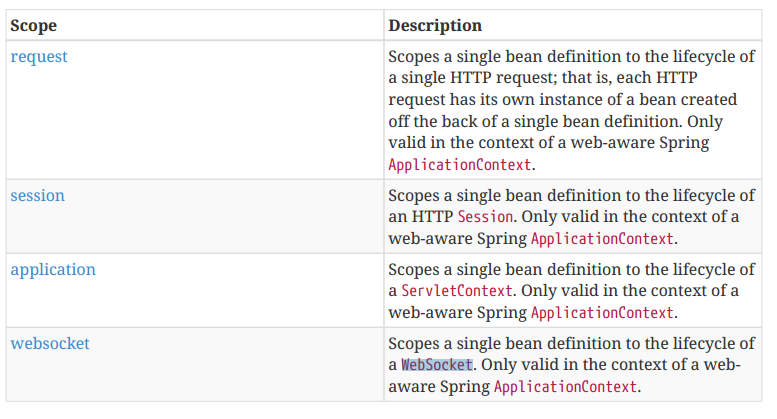
**<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>**

**具体案例**





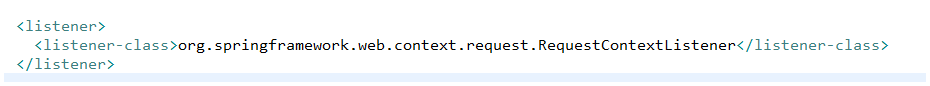
#### **2.2.1.3Web相关的作用域**

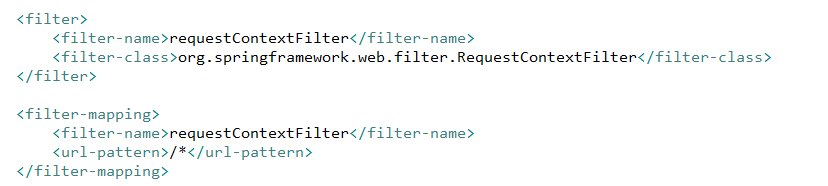


**Request作用域的设置**

首先配置 监听 然后在指定作用域

**在web.xml中配置RequestContextListener**

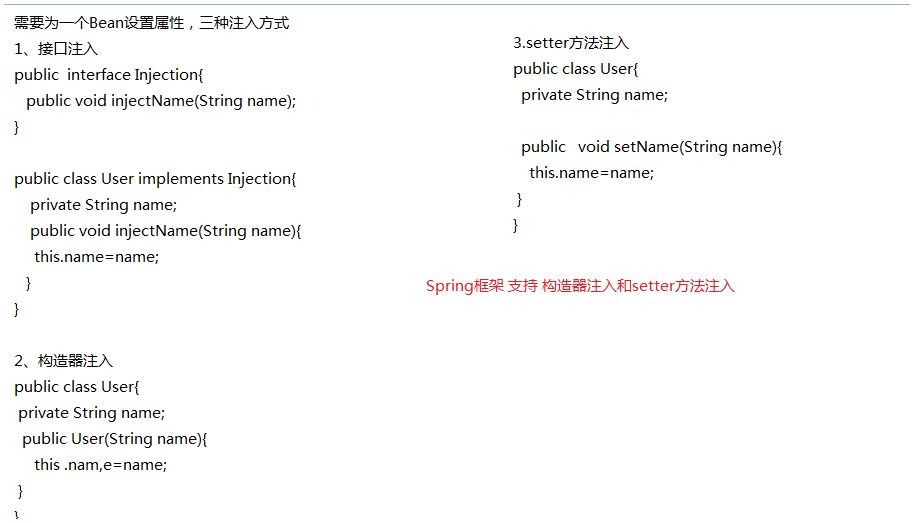




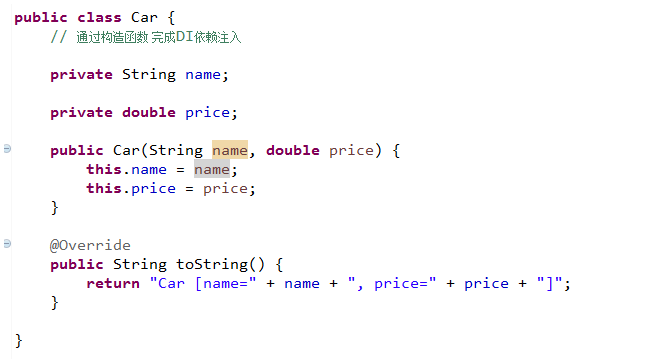
**在代码中指定作用域**



## 2.3 Bean属性的依赖注入



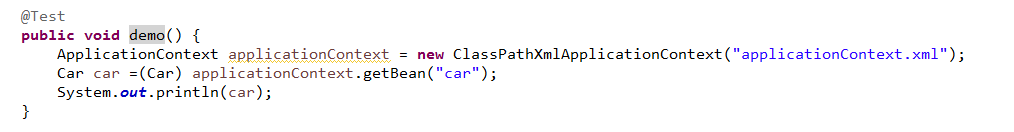
### 2.3.1构造器注入



**配置文件**

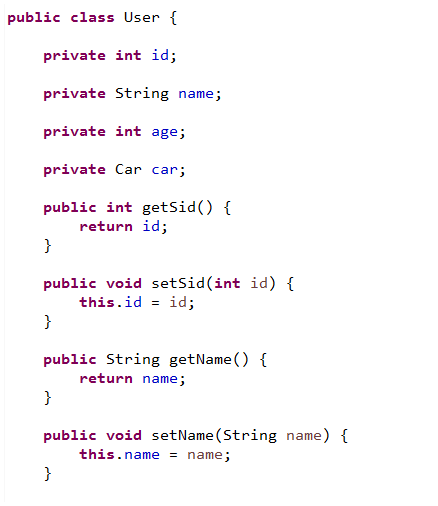


**测试代码**

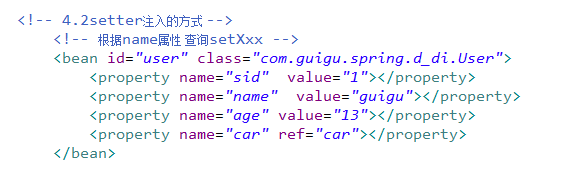


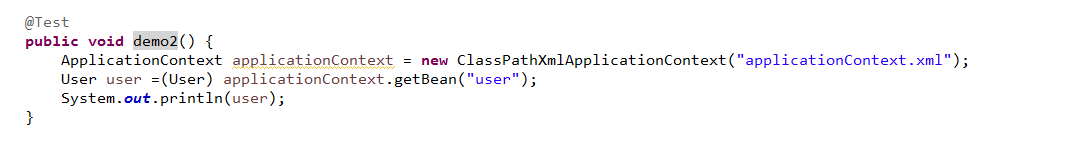
### 2.3.2Setter方法注入

类



配置文件

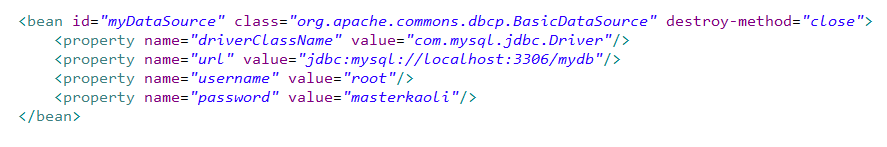




### 2.3.3依赖注入的细节

#### 2.3.3.1配置DBCP数据库连接池

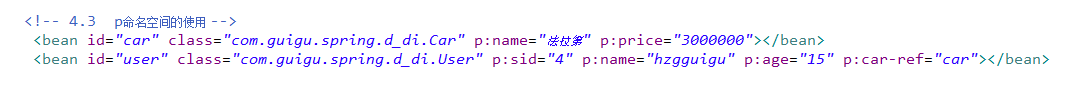
案例一 ：利用Spring配置DBCP数据库连接池



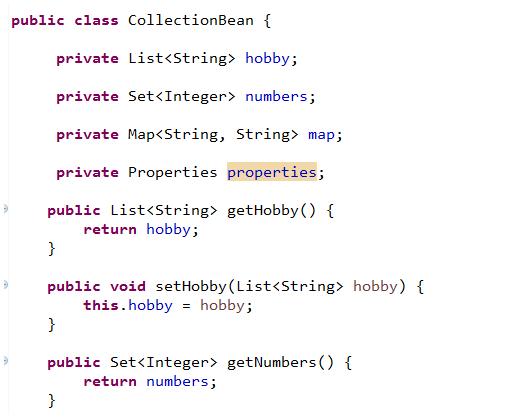
#### 2.3.3.2 p命名空间

把上方的配置DBCP数据库连接池 改为p命名空间的使用。  
**主要是针对setter方法注入的方式进行简化**





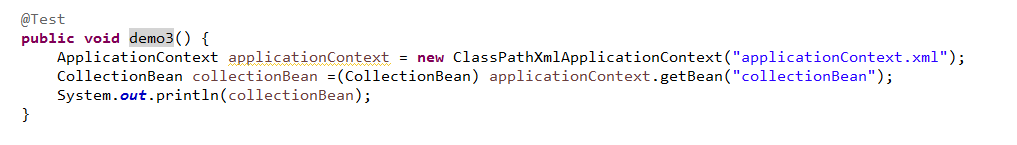
#### 2.3.3.3 集合属性的注入



配置文件



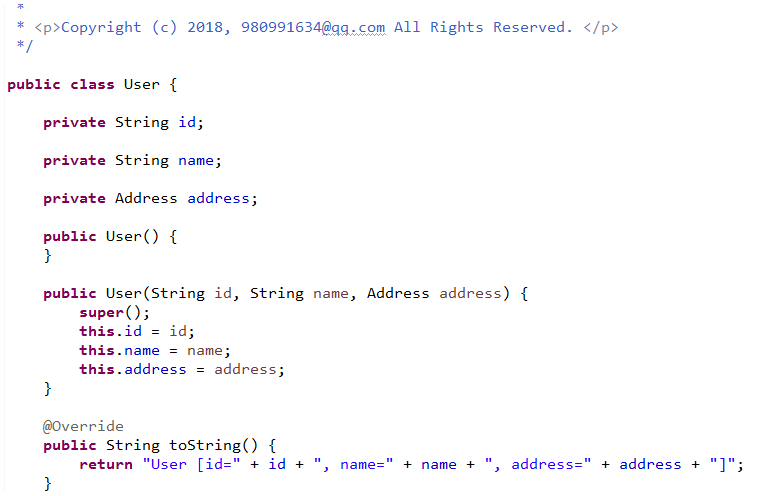
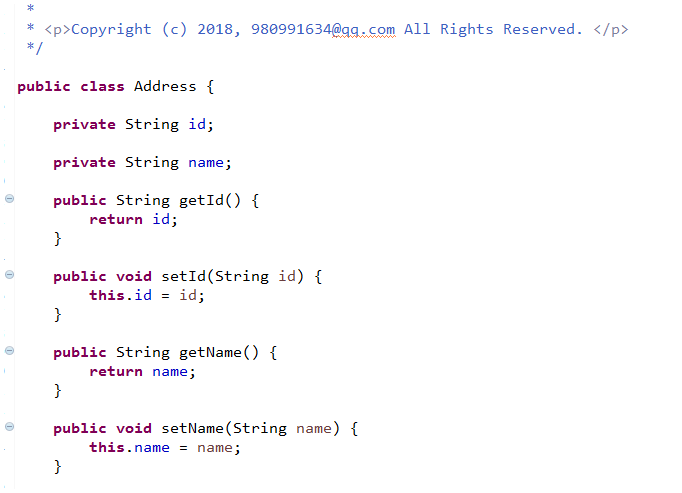
测试代码



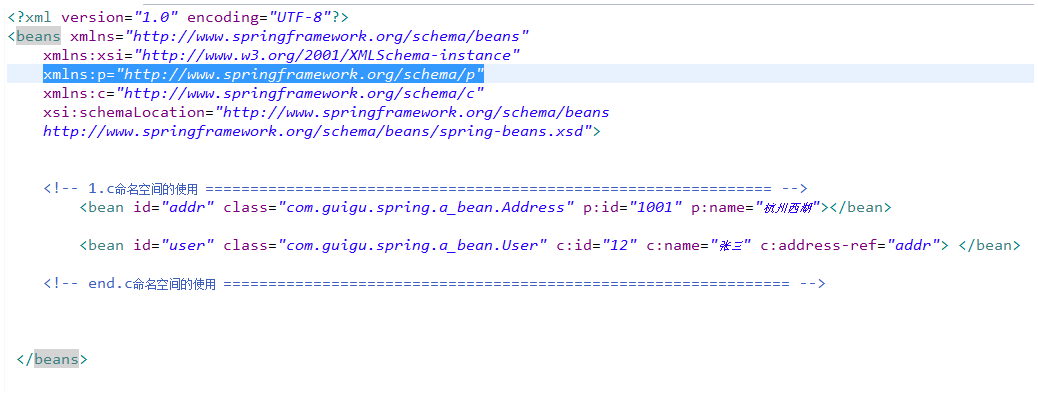
#### 2.3.3.4 属性的合并



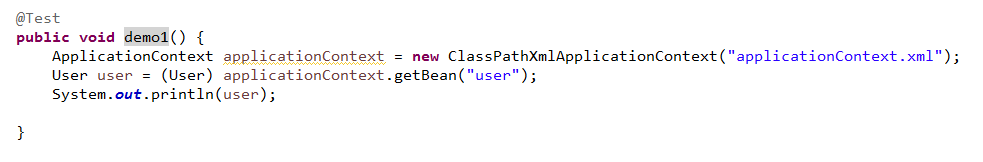
#### 2.3.3.5 C命名空间



**配置文件**



**测试代码**



## 2.4 Ioc容器装配Bean（注解方式）

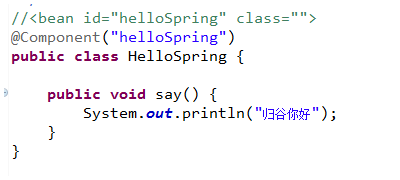
* Spring2.5 引入使用注解去定义Bean
  + @Component 描述Spring框架中Bean

### 2.4.1 注解开发的步骤

**1.搭建开发环境**

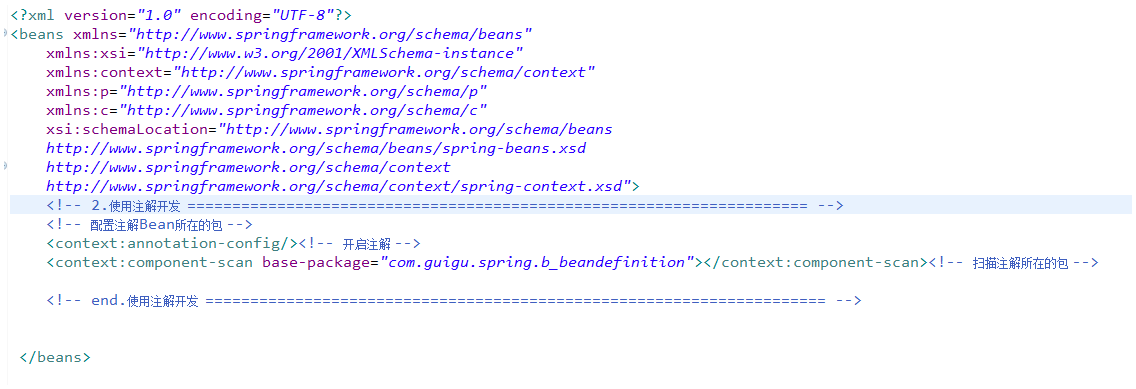
除了基本jar以外 还需要额外导入spring-aop.jar包

**2.使用注解的方式 进行Bean的注册**

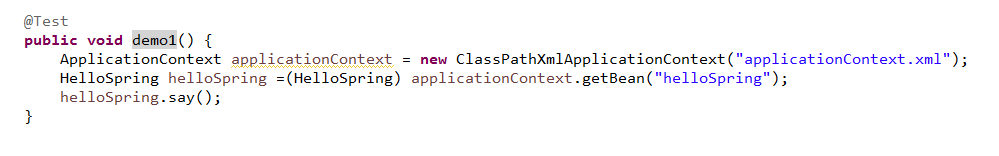


**3.在配置文件开启注解开发 同时配置注解所在的包**

导入context的命名空间



**4.测试代码**

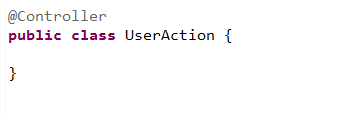
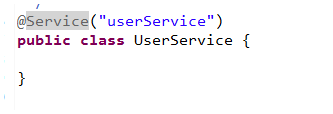


### 2.4.2 常用的基本注解

* @Component 描述Spring框架中Bean 描述某个类
* 除了@Component外，Spring提供了3个功能基本和@Component等效的注解
  + @Repository 用于对DAO实现类进行标注
  + @Service 用于对Service实现类进行标注
  + @Controller 用于对Controller实现类进行标注

这三个注解是为了让标注类本身的用途清晰，Spring已对其增强

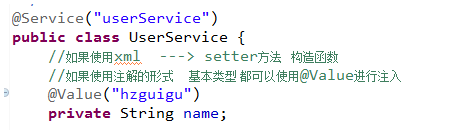




### 2.4.3注解属性

* 使用@Value 注入普通属性
* 使用@Autowired 进行自动注入
* 使用@Qualifier指定注入Bean的名称

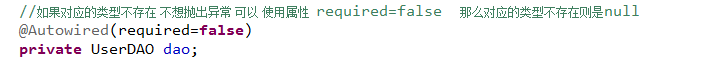
1. **普通属性的注入**



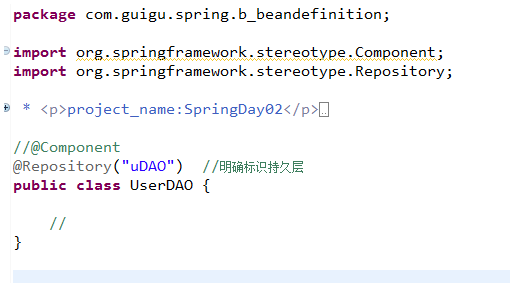
1. **引用类型的注入**



使用@Autowired默认是使用类型注入 如果对应的类型不存在则报错 如果不想报错

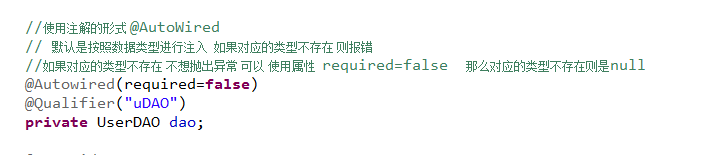


如果存在两个相同的类型



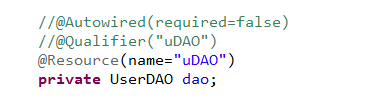


此时程序将报错 可以继续按照 指定的名称进行注入

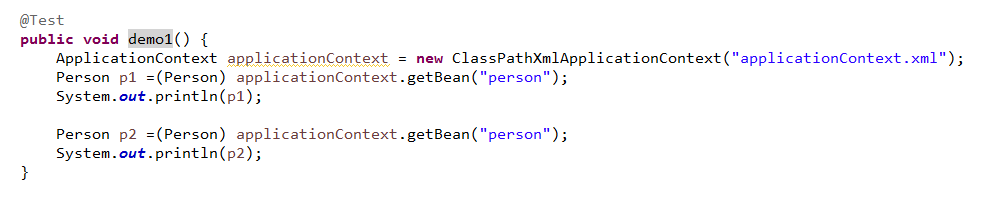
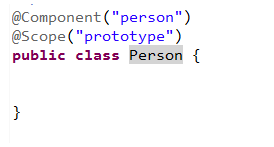


**3) @Resource**

@Resource 等价于 @Autowired



### 2.4.4注解定义作用范围

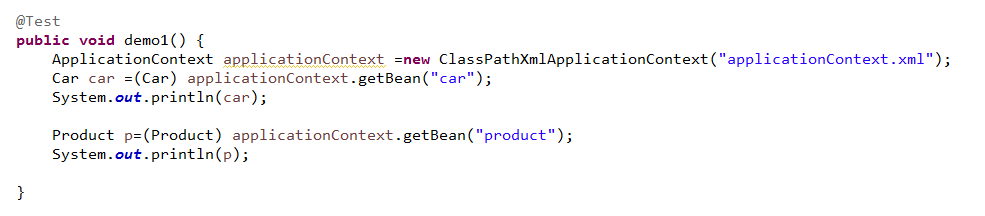


### 2.4.5 Spring3.0提供的注册Bean的注解

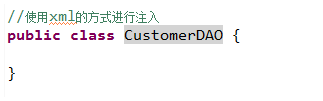
@Configuration 指定POJO类为spring提供bean定义信息

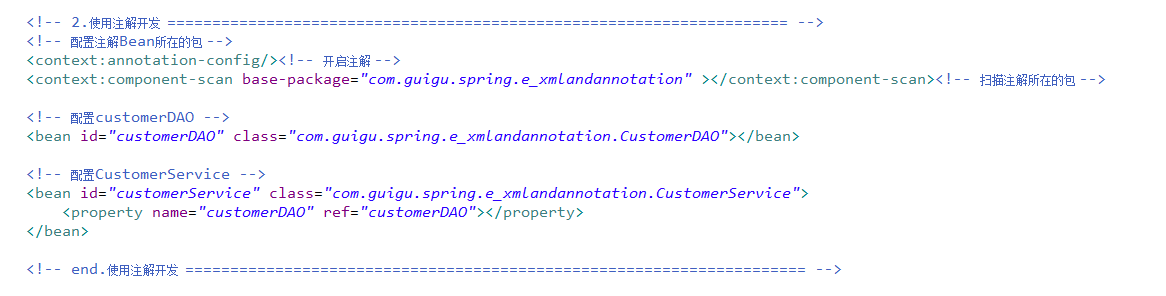
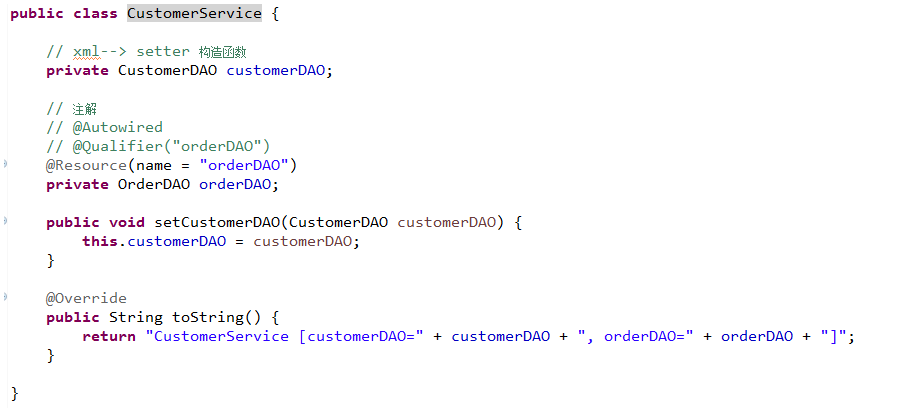
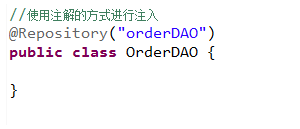
@Bean 提供一个Bean定义信息

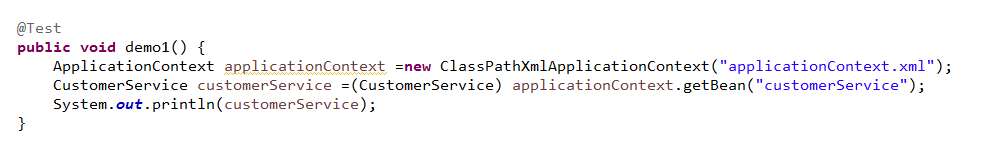
@Primary



### 2.4.6 xml和注解的综合案例

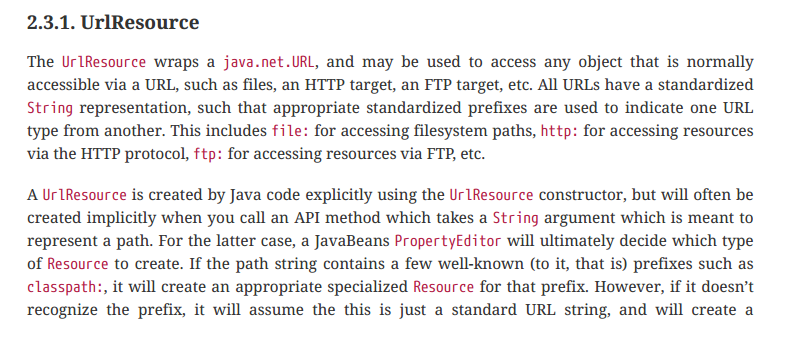


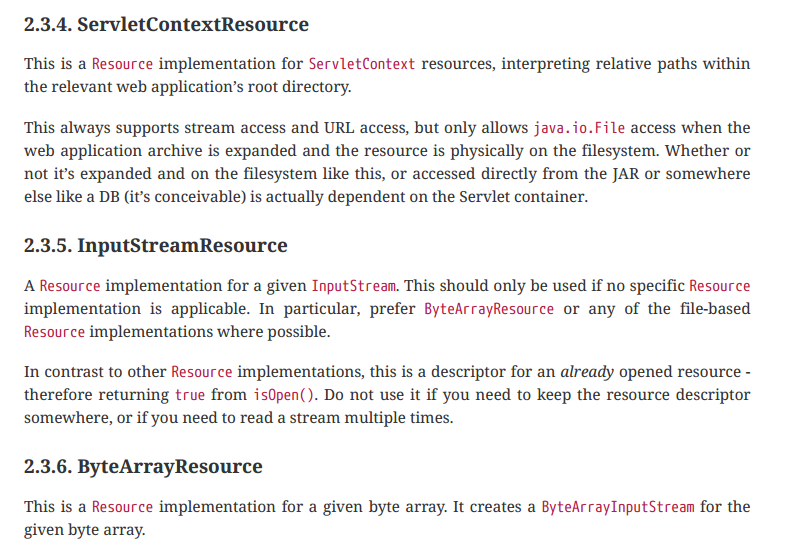
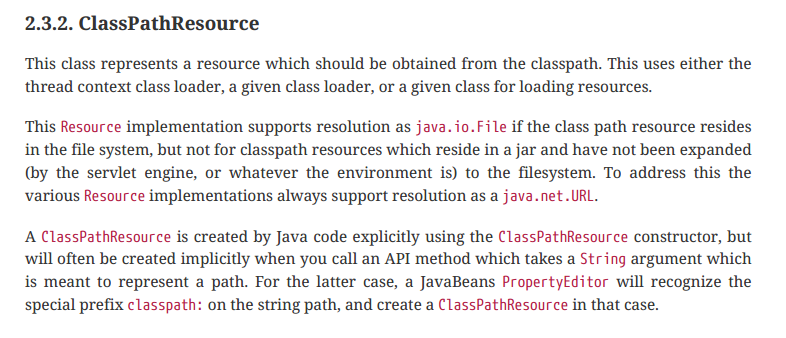




## 2.5资源路径

### 2.5.1在spring中加载某个具体的文件的方式

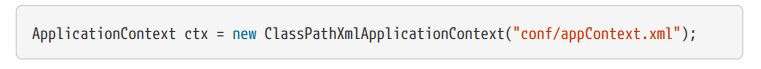


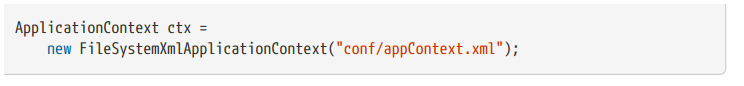




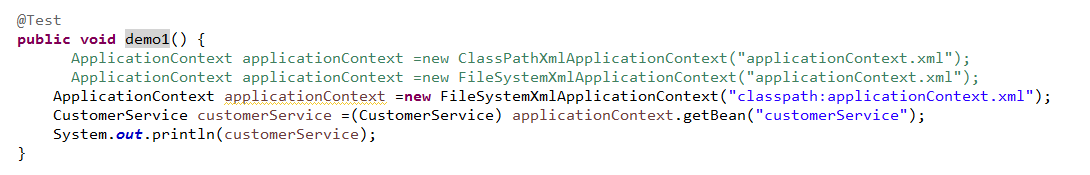


### 2.5.2 Application contexts and Resource paths





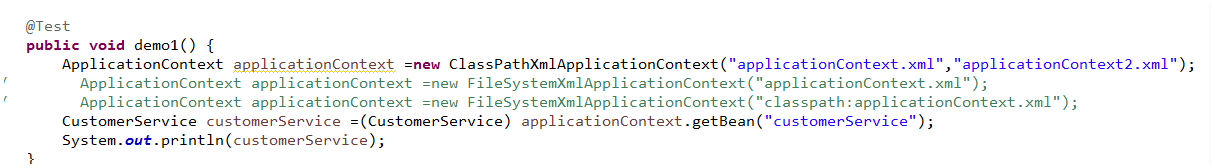




### 2.5.3加载多个配置文件

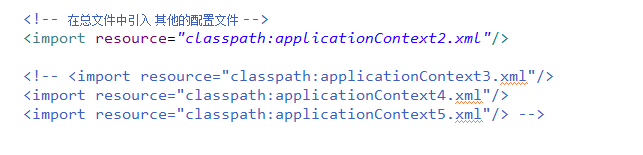
**方式一 ：**

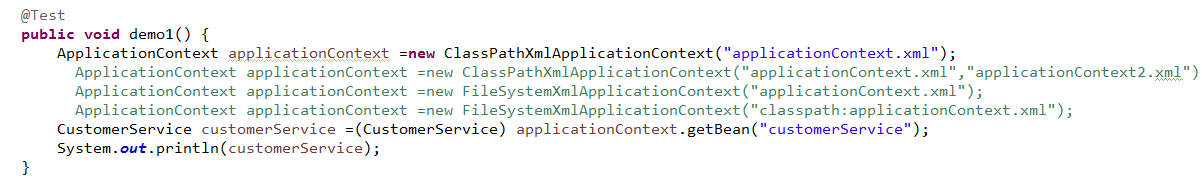
加载多个配置文件可以在定义的使用的时候同时定义多个文件。



**方式二：**

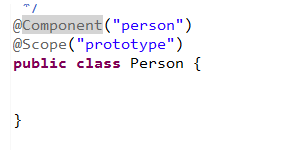
可以在一个总的配置文件中 引入其他的配置文件的信息

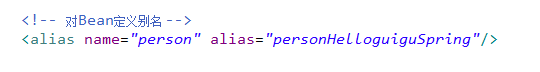


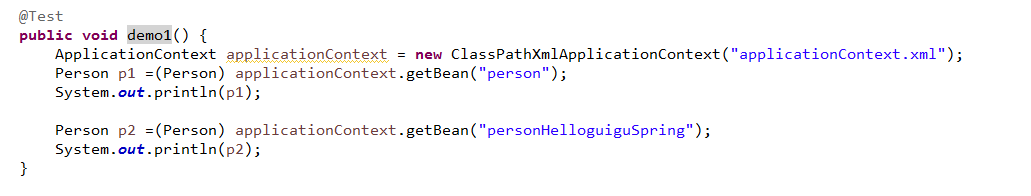


## 2.6Bean的其他信息

### 2.6.1 别名的定义



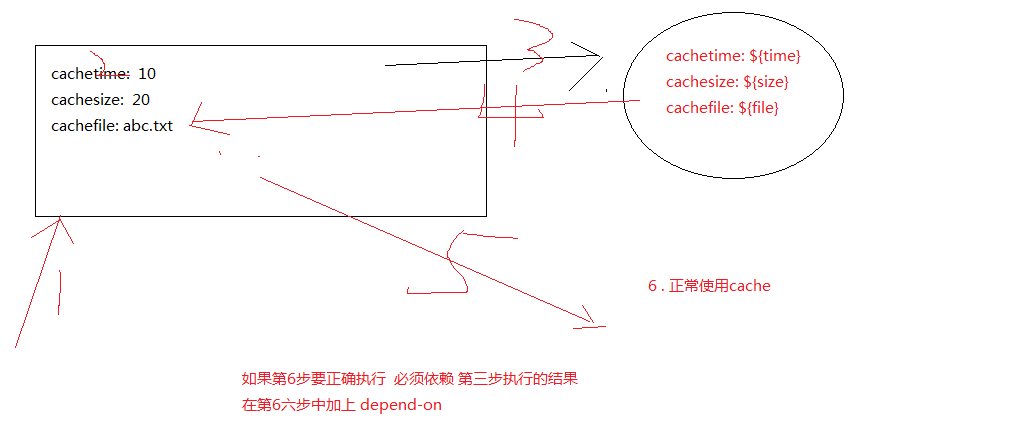




### 2.6.2 depends-on

系统在初始化的时候 需要加载某些配置文件信息。 比如需要加载缓存的信息。

在最初设计系统的时候就考虑到扩展性，最初定义了一个 缓存的信息(缓存的大小，缓存的时间，缓存的位置….) ,在最初由一个默认值 ，每次系统启动的时候 先加载外部的配置文件， 利用配置文件的信息 修改原来默认的信息。



spring的IOC容器负责bean的管理，当实例化一个bean是，spring保证该Bean所依赖的其他bean已经初始化。一般情况下，用<ref>元素建立对其他bean的依赖关系。

比如：

**Xml代码  [收藏代码](javascript:void())**

1. **<bean** id="a"   class="com.yanln.spring.bean.A"**>**   **</bean>**
3. **<bean** id="b" class="com.yanln.spring.bean.B" p:a-ref="a"**>** **</bean>**

 但在某些情况下，这种bean之间的依赖关系并不那么明显。

举个例子：某个论坛系统拥有很多系统参数（如会话过期时间、缓存更新时间等），这些系统参数用于控制系统的运行逻辑。比如：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** SystemSettings{
2. //缓存更新时间
3. **public** **static** **int** REFRESH\_CYCLE = 60;
4. ......
5. }

在SystemSettings中为每一个系统参数提供了默认值，但对于一个灵活的论坛是需要提供一个管理后台的，在管理后台中可以调整这些系统参数并保存到后台数据库中，而在系统启动时，初始化程序从数据库后台加载这些系统参数的值来覆盖系统默认值。比如：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** SystemInit{
2. **public** SystemInit(){
3. //模拟从数据库中加载的系统参数配置值
4. SystemSettings.REFRESH\_CYCLE=100;
5. ......
6. }
7. }

假设论坛系统有一个缓存刷新管理器，它需要根据系统参数SystemSettings.REFRESH\_CYCLE创建缓存刷新定时任务：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** CacheManager{
2. **public** CacheManager(){
3. Timer t = **new** Timer();
4. TimerTask cacheTask = **new** CacheTask();
5. //缓存刷新定时处理
6. t.schedule(cacheTask,0,SystemSettings.REFRESH\_CYCLE);
7. }
8. ......
9. }

在以上的实例中，CacheManager类需要用到SystemSetting类的值,而SystemSettings的值有SystemInit负责初始化，虽然CacheManager不直接依赖SystemInit，但从逻辑上看，CacheManager希望在SystemInit加载并完成系统参数设置后再启动，以获得最新的系统参数值。如果这三个bean都在spring配置文件中定义，我们如何保证SystemInit在CacheManager之前进行初始化呢？

**spring允许用户通过depends-on属性指定bean前置依赖的bean,前置依赖的bean会在本bean实例化之前创建好：**

**Xml代码  [收藏代码](javascript:void())**

1. **<bean** id="sysinit" class="SystemInit"**>**
3. **<bean** id="manager" class="CacheManager"  depends-on="sysinit"**/>**

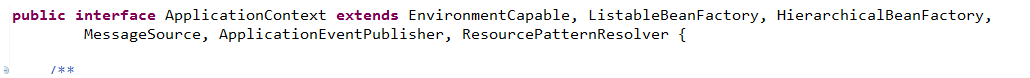
### 2.6.3 BeanFactory 和ApplicationContext的区别

BeanFactory接口和ApplicationContext接口的区别：

他们的主要区别在于BeanFactory采用延迟加载。只有当第一次getBean(“Xxx”)某个对象的时候才初始化， 而ApplicationContext会在加载配置文件的时候就初始化Bean。

同时ApplicationContext是对BeanFactory的扩展， 拥有了 国际化处理、事件传递、Bean的自动装配、各种不同应用层的Conetext的实现。

ApplicationContext是BeanFactory接口的子接口。

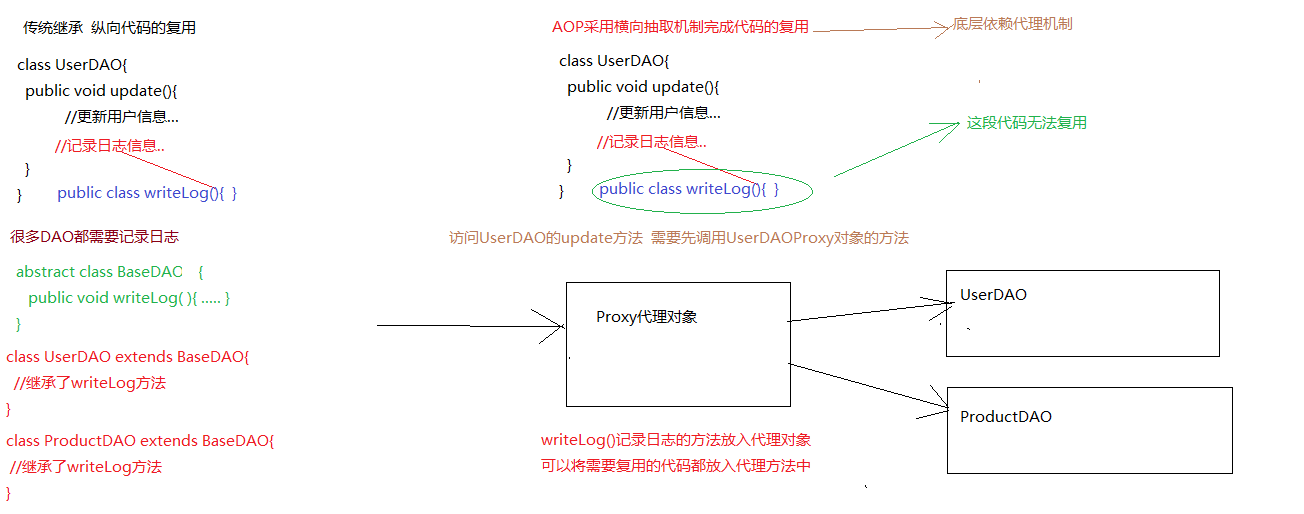


# 三、AOP面向切面的编程

## 3.1AOP基础知识

### 3.1.1 AOP的概念

* 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**（性能监视、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织入增强代码
* AspecJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入



**AOP面向对象底层是代理机制**

### 3.1.2AOP相关的术语

**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,因为spring只支持方法类型的连接点.

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

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

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

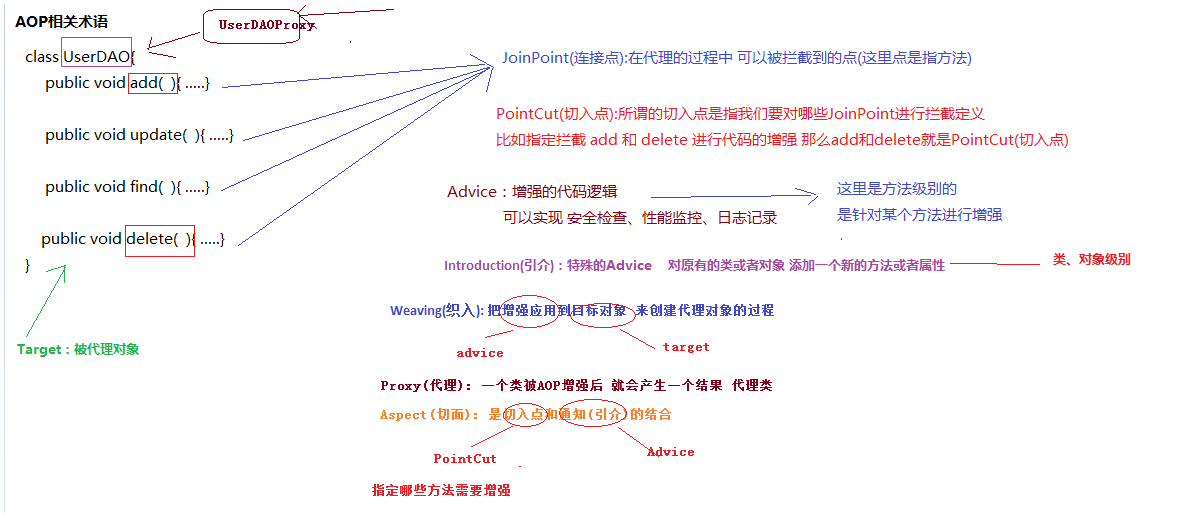
**Target(目标对象):**代理的目标对象

**Weaving(织入):**是指把增强应用到目标对象来创建新的代理对象的过程.

spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入

**Proxy（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):** 是切入点和通知（引介）的结合



### 3.1.3Spring提供了五种Advice

* AOP联盟为通知Advice定义了org.aopalliance.aop.Interface.Advice
* Spring按照通知Advice在目标类方法的连接点位置，可以分为5类
  + 前置通知 org.springframework.aop.MethodBeforeAdvice
    - 在目标方法执行前实施增强
  + 后置通知 org.springframework.aop.AfterReturningAdvice
    - 在目标方法执行后实施增强
  + 环绕通知 org.aopalliance.intercept.MethodInterceptor
    - 在目标方法执行前后实施增强
  + 异常抛出通知 org.springframework.aop.ThrowsAdvice
    - 在方法抛出异常后实施增强
  + 引介通知 org.springframework.aop.IntroductionInterceptor
    - 在目标类中添加一些新的方法和属性

## 3.2 Spring的切面(advisor)

Advisor就是Aspcet 就是对PointCut应用Advice( 通常所说的Advisor是指只有一个PointCut和一个Advice )

Advisor 代表一般的切面。

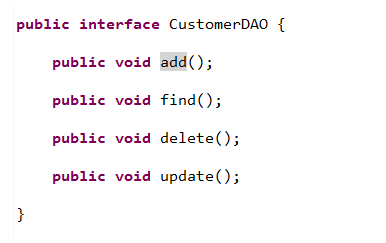
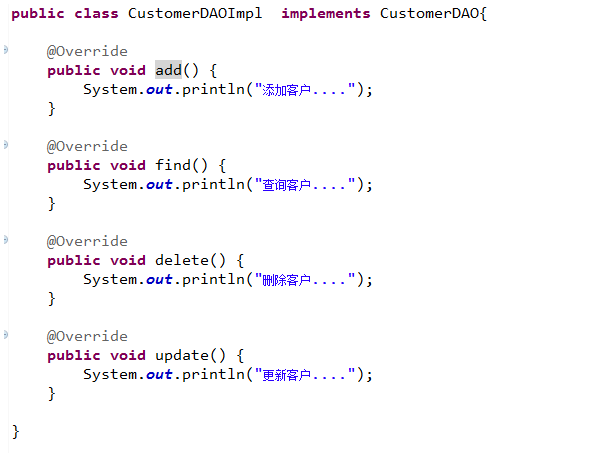
Advice 增强 假如没有任何切点 Advice就是一个切面 ，这个切面就是对目标所有的方法进行拦截。

* Advisor : 代表一般切面，Advice本身就是一个切面，对目标类所有方法进行拦截（没有切点的切面）
* PointcutAdvisor : 代表具有切点的切面，可以指定拦截目标类哪些方法（有切点的切面）
* IntroductionAdvisor : 代表引介切面，针对引介通知而使用切面（不要求掌握）

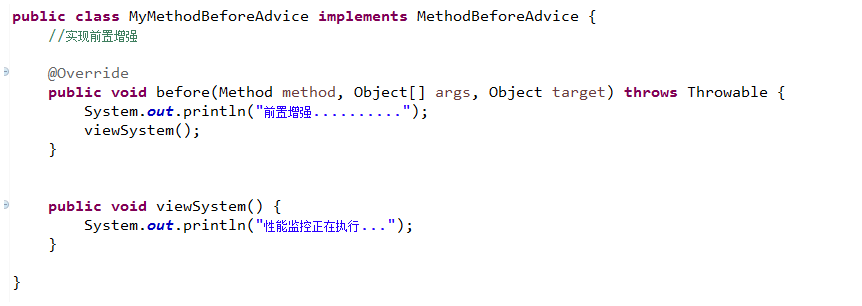
### 3.2.1 Advisor 不带切点的切面(Advice)

--针对目标类中所有的方法都拦截 ---然后都进行增强

**1)编写被代理对象 和实现类**

**2）增强--->前置增强通知advice**



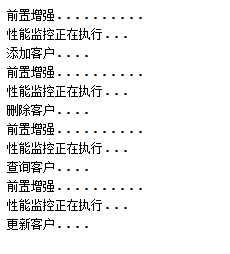
**3）编写配置 文件 创建代理对象**



**4）编写测试代码**

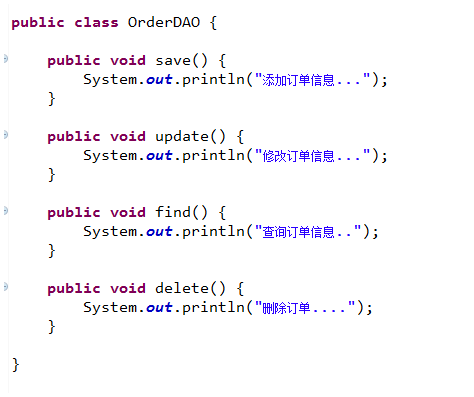


**5）运行结果**

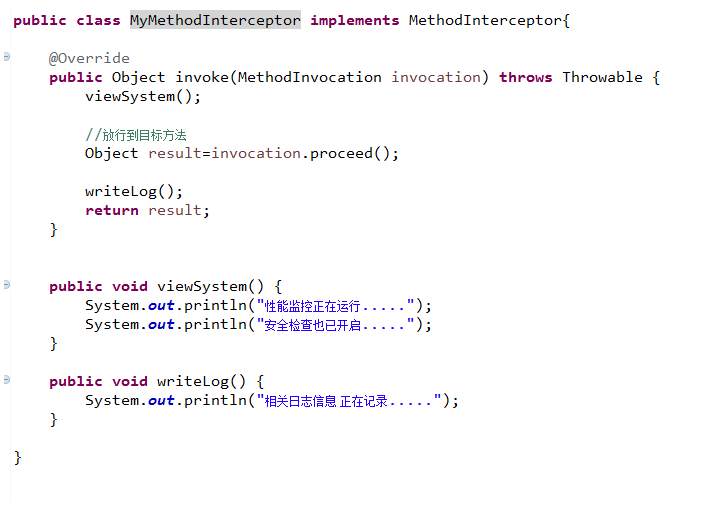


### 3.2.2 PointcutAdvisor带有切点的切面

**1)编写被代理对象 和实现类**



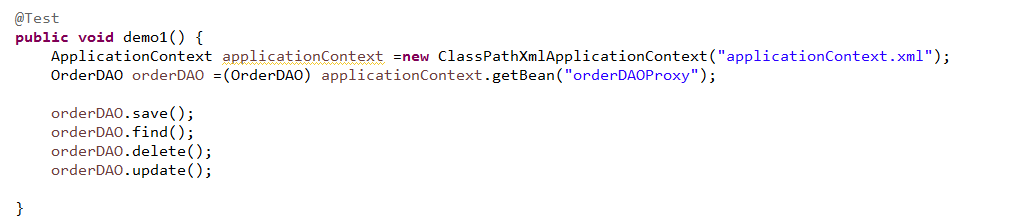
**2）增强--->环绕增强通知advice**



**3）编写配置 文件 创建代理对象**

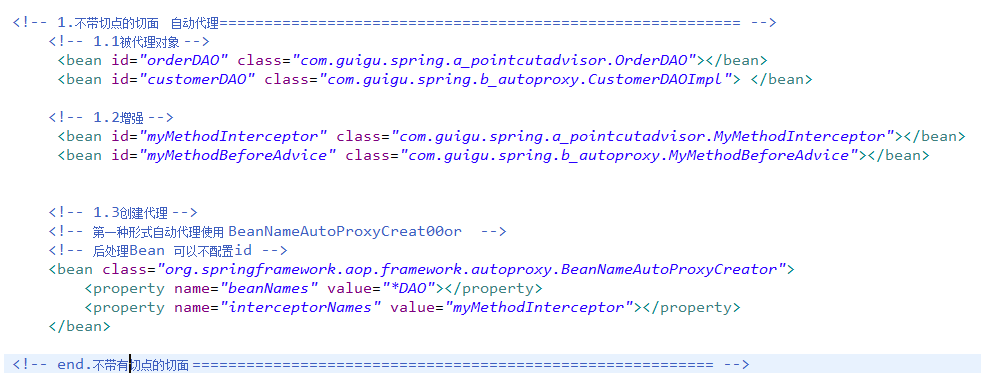


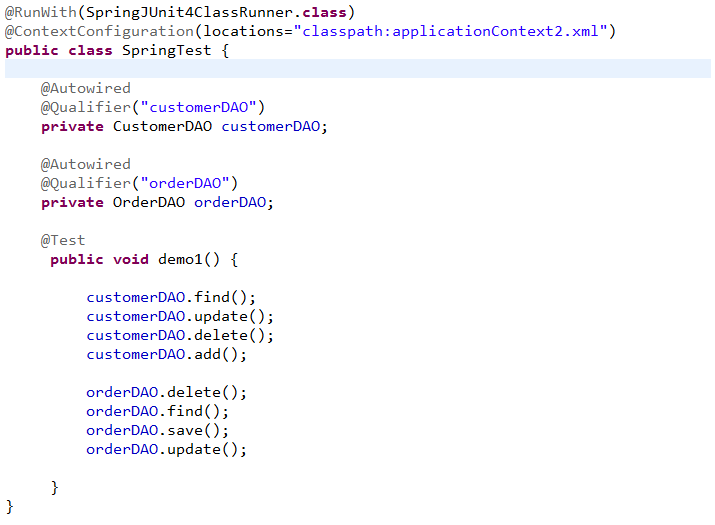
**4)测试代码**



### 3.2.3自动代理实现增强

#### 3.2.3.1 BeanNameAutoProxyCreator





#### 3.2.3.2 DefaultAdvisorAutoProxyCreator



## 3.3使用AspectJ实现AOP(基于注解的方式)

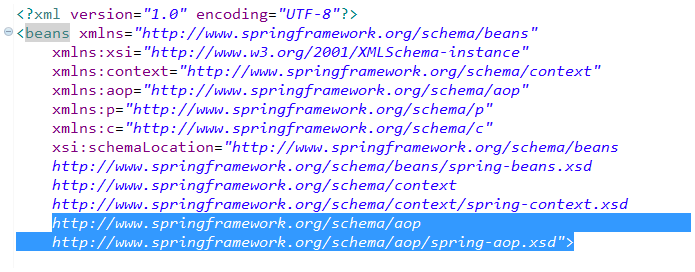
### 3.3.1 基础概念

* AspectJ是一个基于Java语言的AOP框架
* Spring2.0以后新增了对AspectJ切点表达式支持
* @AspectJ 是AspectJ1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面
* 新版本Spring框架，建议使用AspectJ方式来开发AOP
* 使用AspectJ 需要导入Spring AOP和 AspectJ相关jar包
  + spring-aop-4.2.0.RELEASE.jar
  + com.springsource.org.aopalliance-1.0.0.jar
  + spring-aspects-4.2.0.RELEASE.jar

### 3.3.2实现步骤

**1）导入aspectjweaver.jar 包**

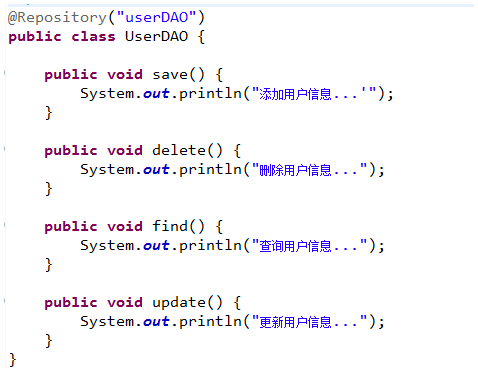
**2）导入spring的aop的命名空间**



**3）开启aspectJ自动代理**



**4）编写代理对象**



**5）编写切面**

**5.1）定义增强**

**@Before**

**@AfterReturning**

**@AfterThrowing**

**@After**

**@Around**

**5.2)定义切点**

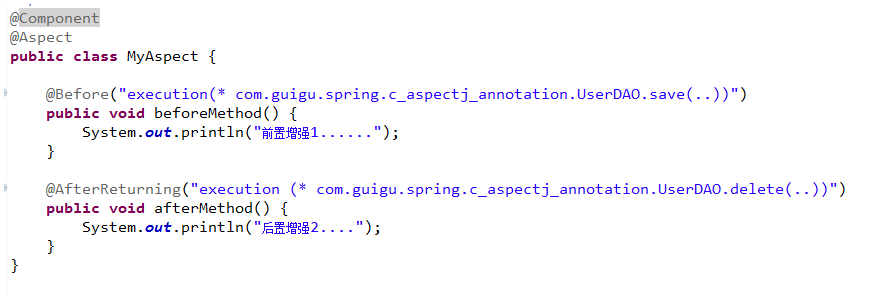
* 通过execution函数，可以定义切点的方法切入
* 语法：
  + execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>)
* 例如
  + 匹配所有类public方法 execution(public \* \*(..))
  + 匹配指定包下所有类方法 execution(\* cn.guigu.dao..\*(..)) 不包含子包
  + execution(\* cn.guigu.dao..\*(..)) ..\*表示包、子孙包下所有类
  + 匹配指定类所有方法 execution(\* cn.guigu.service.UserService.\*(..))
  + 匹配实现特定接口所有类方法

execution(\* cn.guigu.dao.GenericDAO+.\*(..))

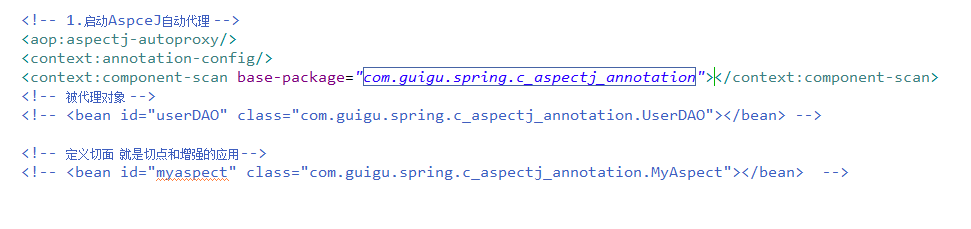
* + 匹配所有save开头的方法 execution(\* save\*(..))

5.3）定义切面对象

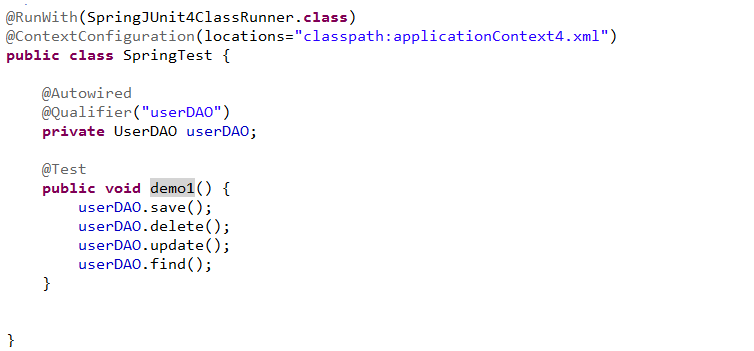
@Aspect



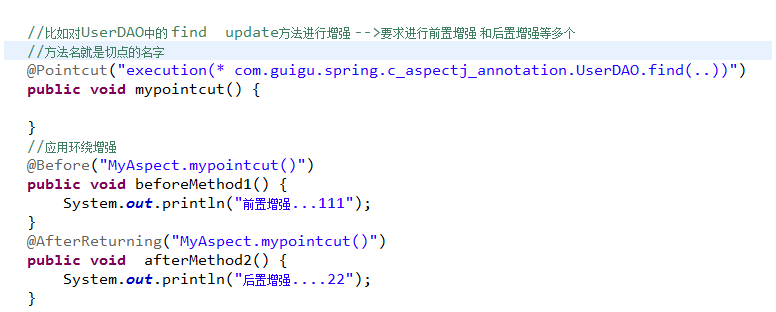
6）编写配置文件



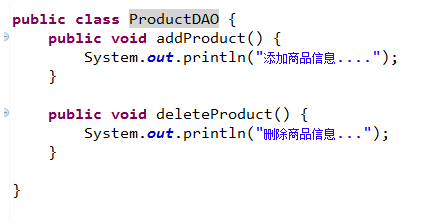
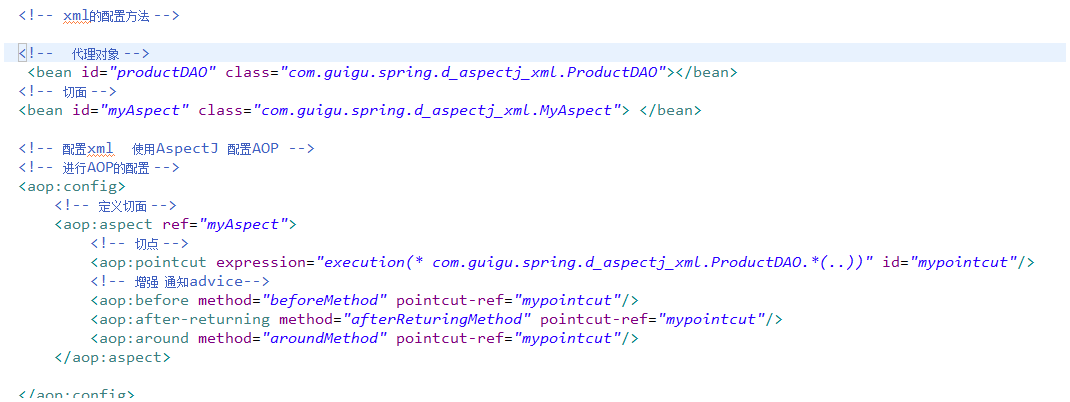
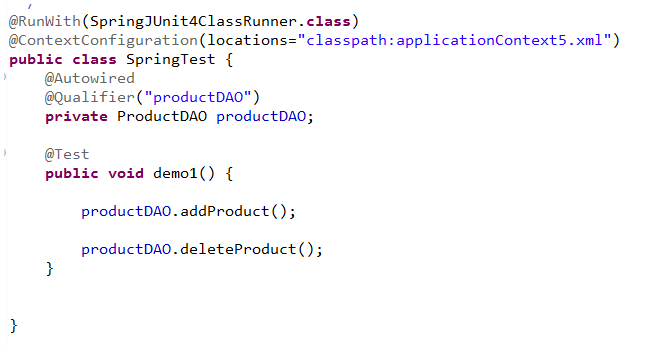
7）测试代码



### 3.3.3 切点的定义



## 3.4使用AspectJ实现AOP(基于XML的方式)

# 四、SpringJDBCTemplate

## 4.1 使用步骤

**1)导入相关的jar包**

spring-jdbc-5.0.2.RELEASE

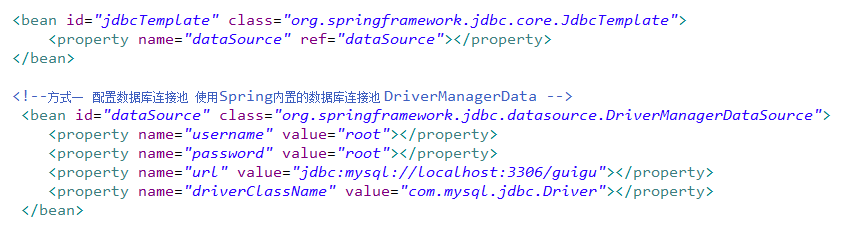
spring-tx-5.0.2.RELEASE

mysql-connector-java-5.0.8-bin

**2）编写配置文件**

配置数据库连接池

**2.1）方式一 使用Spring内置的数据库连接池**

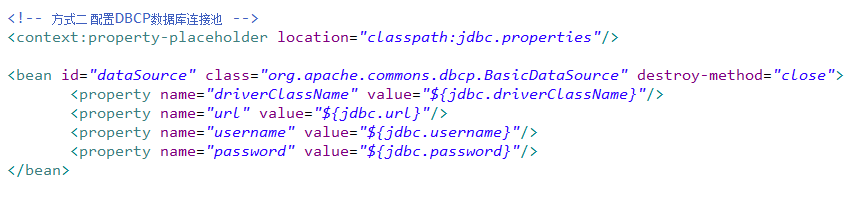


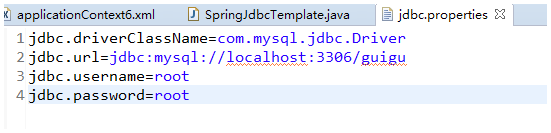
**2.2）方式二 使用DBCP数据库连接池**

commons-dbcp-1.4

commons-pool-1.6

commons-logging-1.1.1

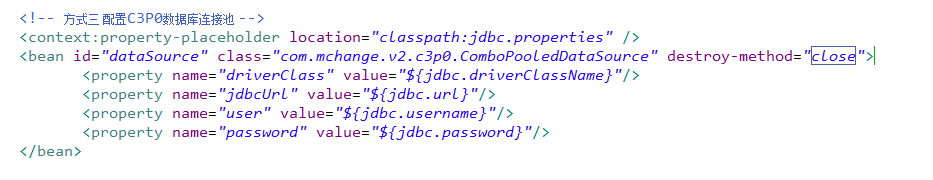




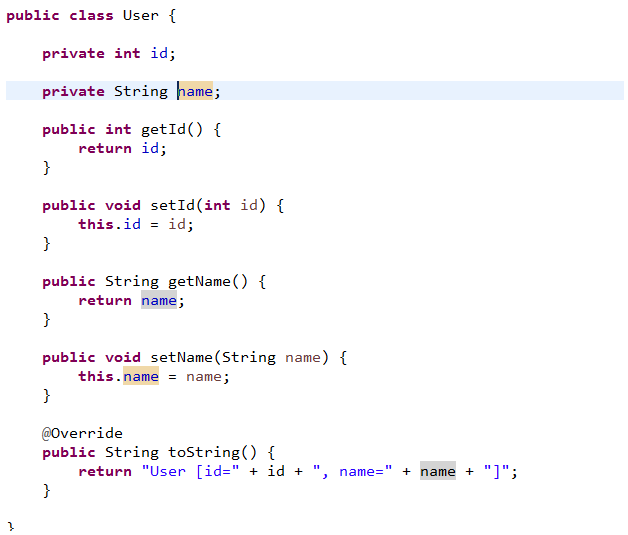
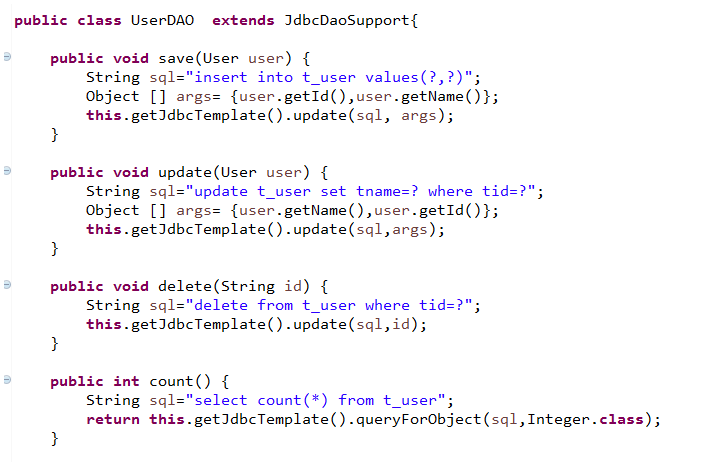
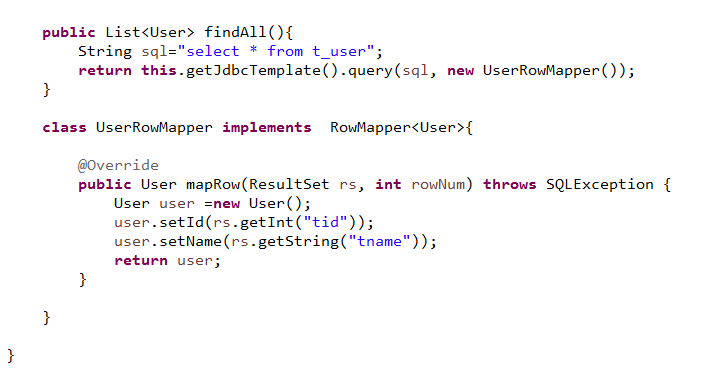
**2.3）使用C3P0数据库连接池**

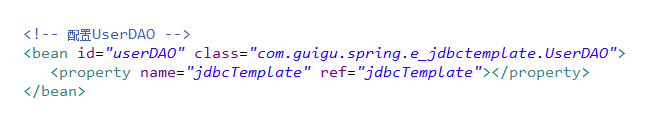
**c3p0-0.9.5.2**

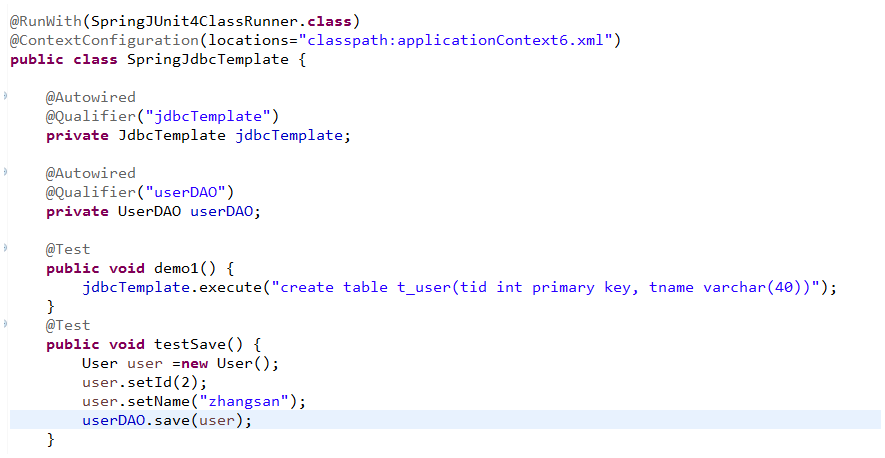
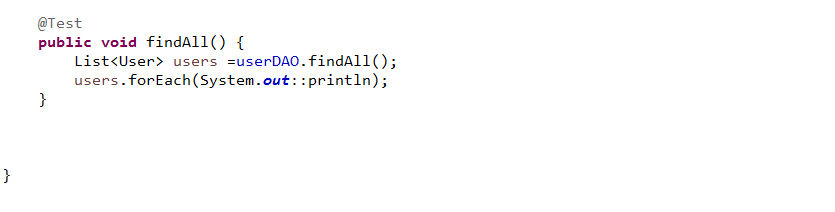
**mchange-commons-java-0.2.14**



## 4.2CRUD



# 五、Spring事务管理

## 5.1 基础概念

### 5.1.1事务管理相关的API

Spring事务管理高层抽象主要包括3个接口

* PlatformTransactionManager 平台事务管理器

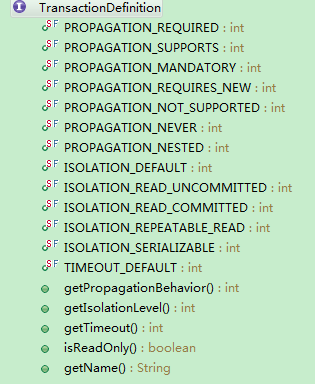
Rollback 回滚事务

Commit 提交事务

getTransaction根据 事务定义的信息 获取当前状态

* TransactionDefinition

事务定义信息(隔离、传播、超时、只读)

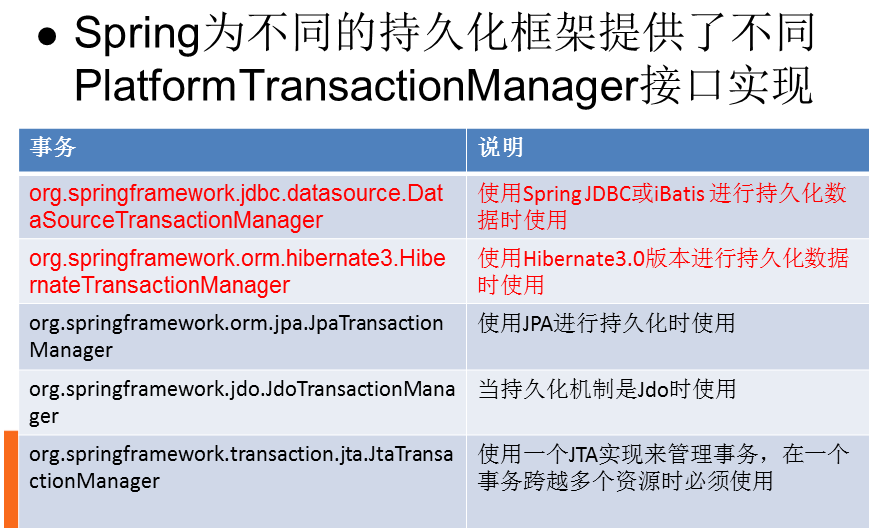


* TransactionStatus

事务具体运行状态

* 关系：  PlatformTransactionManager 根据TransactionDefinition 进行事务管理 ，管理的过程中 ，事务存在多种状态，每个状态的信息 通过TransactionStatus进行表示

### 5.1.2事务管理器PlatformTransactionManager



### 5.1.3事务相关的内容

1）事务的四大特性 ACID 原子性 一致性 隔离性 持久性

2）由于隔离性引发的问题：

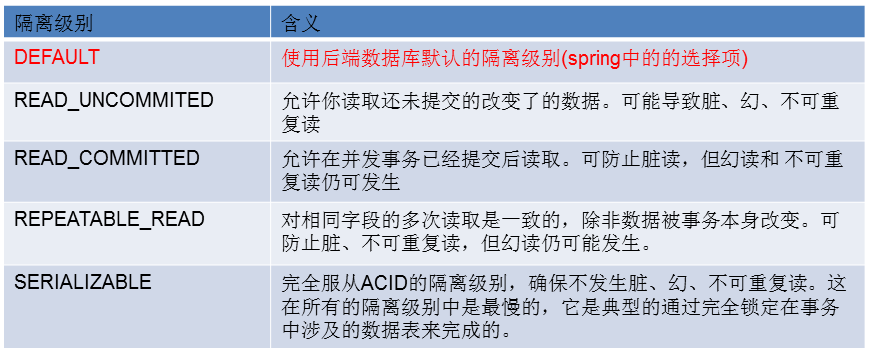
**脏读**:一个事务读取了另一个事务改写但还未提交的数据,如果这些数据被回滚，则读到的数据是无效的。

**不可重复读**：在同一事务中，多次读取同一数据返回的结果有所不同。换句话说就是，后续读取可以读到另一事务已提交的更新数据。相反，“可重复读”在同一事务中多次读取数据时，能够保证所读数据一样，也就是，后续读取不能读到另一事务已提交的更新数据。

**幻读**：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。

1. 为了解决隔离性引发的问题

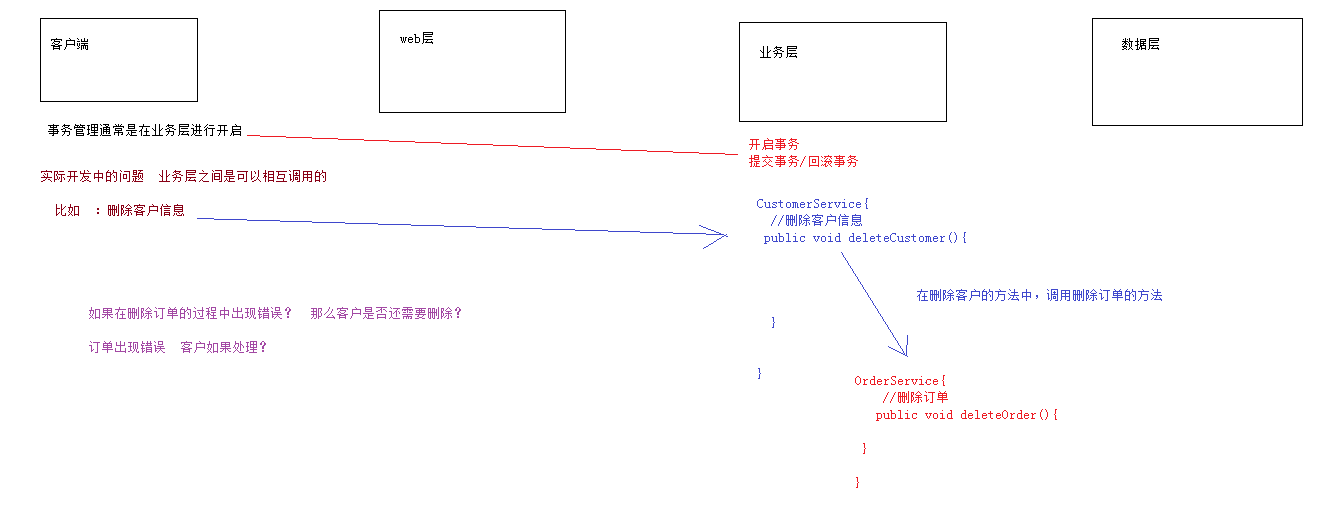
定义了四种隔离级别



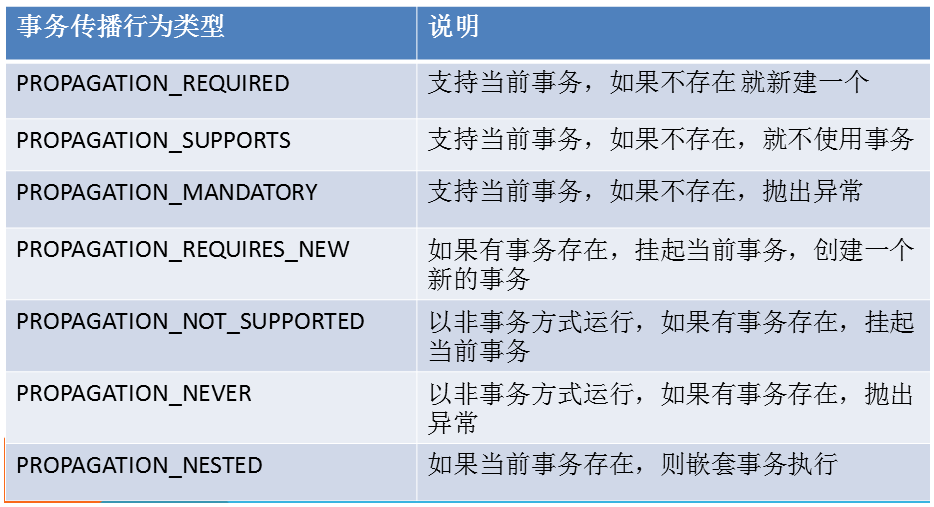
**4）事务的传播行为**

不是JDBC的规范定义

传播行为是针对实际开发中的问题



传播行为就是解决 一个业务层事务调用另外一个业务层事务 ， 那么事务之间的关系如果处理。



PROPAGATION\_REQUIRES\_NEW：如果有事务存在，挂起当前事务，创建一个新的事务。

\*\*\*\*生成订单， 发送邮件到客户邮箱。 通知邮件是创建一个新的事务，如果发送邮件失败，不影响订单的生成。

PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果有事务存在，挂起当前事务

\*\*\*\*不需要事务

PROPAGATION\_NEVER：以非事务方式运行，如果有事务存在，抛出异常

PROPAGATION\_REQUIRED：支持当前事务，如果不存在 就新建一个

\*\*\*\* 一个事务 要么都成功 要么都失败。

PROPAGATION\_NESTED：如果当前事务存在，则嵌套事务执行

\*\*\*一个事务 在A事务调用B事务过程中，如果B事务失败，回滚到之前的savepoint点。 用户也可以选择进行提交事务 commit

## 5.2Spring事务管理的方式

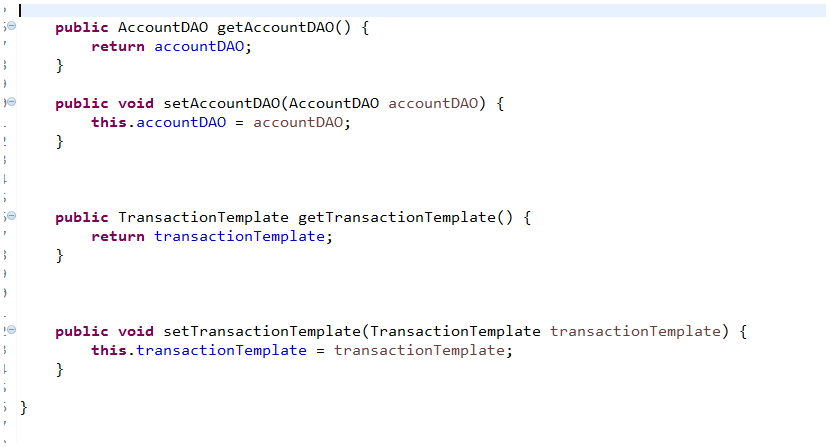
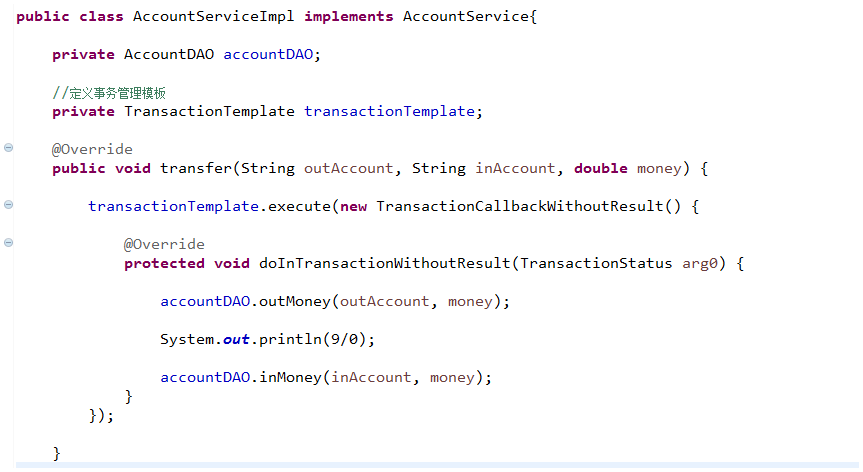
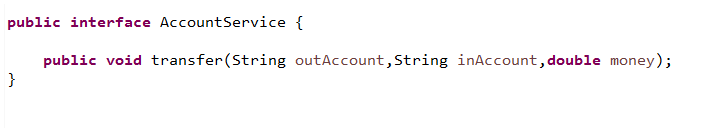
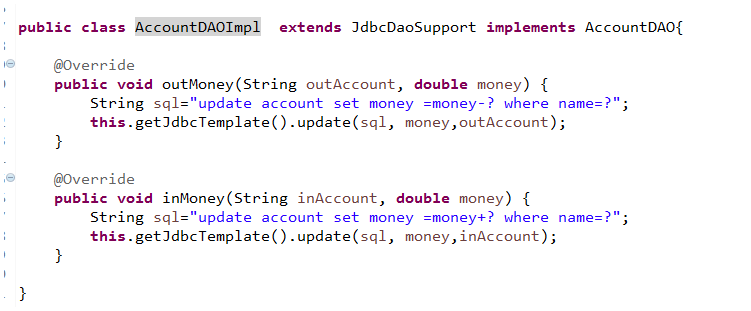
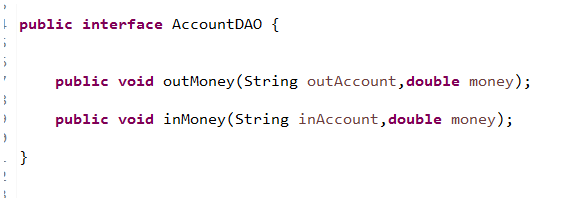
1)编程式事务管理 (了解)

在代码中通过TransactionTemplate.手动进行事务管理 在实际开发中使用比较少

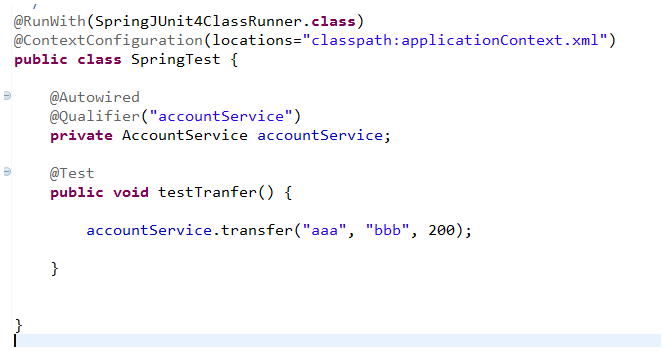
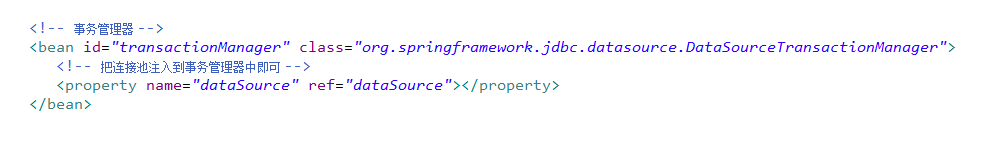
2）声明式事务管理

在配置文件中 对Bean的方法进行事务管理 基于AOP的思想 无需编写太多代码 。TransactionTemplate.

### 5.2.1 Spring编程式 事务管理TransactionTemplate

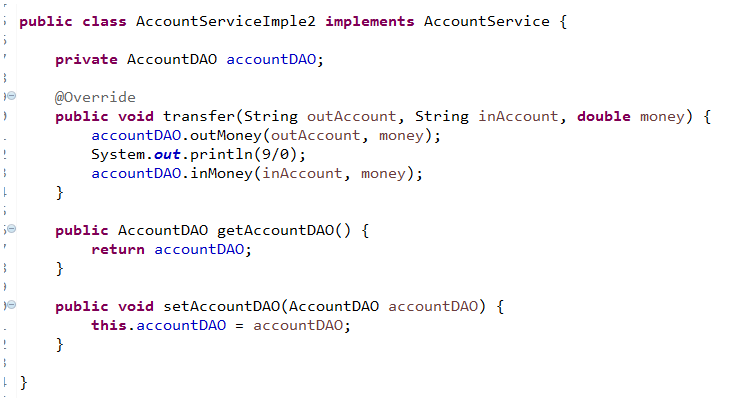


配置文件



在实际开发中 因为编程式事务 容易存在代码的侵入，会有不安全问题，所以不常使用。通常使用声明式事务管理

### 5.2.2Spring声明式事务管理(TransactionProxyFactoryBean)



配置文件





### 5.2.3基于注解的声明式事务管理



