## Spring框架引言

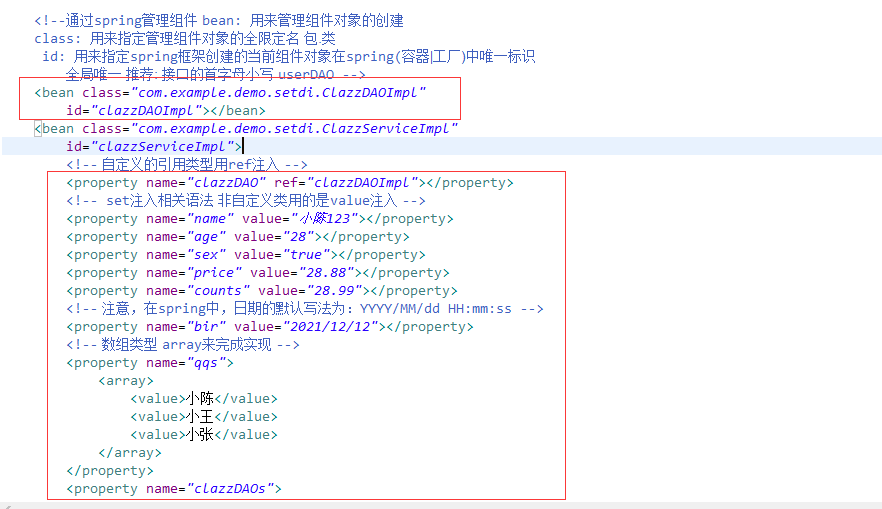
Spring不是替换某项技术，而是对项目中的组件(大部分是类，如service里的类)进行集中管理的工具（创建、使用、销毁）,以spring管理的方式代替了原先new的方式，Spring就是一个管理工具。

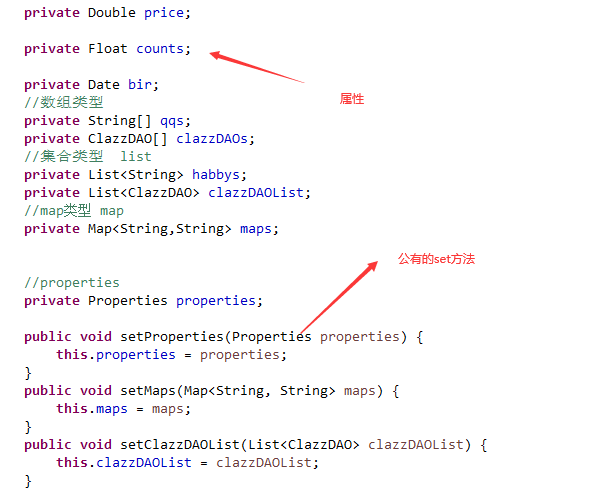
最简单的spring只是简单的几个spring核心jar包和一个xml，xml里通过bean标签里的class参数指定类的全限定名，id为这个类做唯一标识，

Property来完成class之间的调用关系，name为组件中的属性名，然后根据属性类型的不同，进行不同处理。最后声明组件的成员变量并实现公有的set方法。除set（Property）注入外，还有自动注入（autowrite）和构造注入(constructor-arg)（三种方式）。具体见spring文档的图片笔记，或本人的源码。

在jar包作用下我们就把这个类放入了spring管理工具中。

代码中我们用ClassPathXmlApplicationContext加载该配置文件，用ApplicationContext去接受该配置文件中类的容器，用ApplicationContext.getBean( id )去得到容器中的类，这样我们就从spring工厂中获得了需要的类，替代了原先用new的方式来创建。通过非new的方式减少了JVM内存的开支。







### Spring IOC 和 DI

#### IOC 控制反转

其实就是控制权力的反转，因为在没有spring工厂之前，我们都是通过new的方式来创建类，想在项目中何时何地创建就创建，但是有了spring工厂之后，我们将创建的权力交给了spring（即在spring配置文件中加入bean标签），只要加载配置文件，该类就已经被创建了，我们只是按需从工厂中取出来使用，这种创建权力的反转又叫控制反转（IOC）。

#### DI 依赖注入

可以从工厂中拿到各个组件的实例往往有依赖嵌套关系的，例如：service的很多接口都需要调用DAO层的接口，为了维护组件之间的依赖关系，完成组件之间的调用，spring提供依赖注入技术（bean标签中的property标签等），来完成组件之间的调用，这样就用依赖注入的方式代替了组件中用new的方式来调用另一个组件。

总之;IOC就是将各个组件的创建权力交给spring,DI是用来维护组件之间的调用关系。

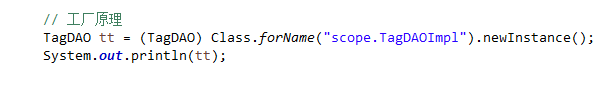
## Spring工厂特性

1. 组件创建模式

* Spring默认管理组件对像（bean标签）是单例创建（singleton）的,也就是说某一个组件无论从spring中取多少次，都是从同一个内存地址取出来的，即用“==”比较时返回true，会产生数据共享的问题。
* Spring也可以改为多例模式（prototype），即scope参数，防止同一内存的数据共享问题。

1. 组件创建原理也是IOC的底层原理。

* Bean标签创建组件的原理：反射+构造参数，即用反射的forName（全限定名）来创建



1. Spring管理组建的声明周期

* 单例对象：

1. spring工厂启动，工厂里的单例对象就会被创建（当含有单例对象bean标签的XML文件被加载到jar后，单例对象就会被创建）
2. spring工厂正常销毁（执行close方法），单例对象也会被销毁。
3. 总结：工厂启动（加载XML后）就被创建，工厂关闭，就被销毁。

* 多例对象：

1. 从Spring工厂中获取的时候才会被创建（即执行getBean的时候）
2. 销毁是在没有对象引用的时候，由JVM垃圾回收。
3. Spring的好处（主要是相对传统的new的方式来说）
4. 解耦和：通过依赖注入的方式来代替组件之间用new相互调用，以后需要更改只 需要更改配置文件而不是一个个更改源代码，降低了组件之间的耦合度。
5. 降低内存使用率：默认的单例共享内存的方式来取代new。

#### Spring AOP

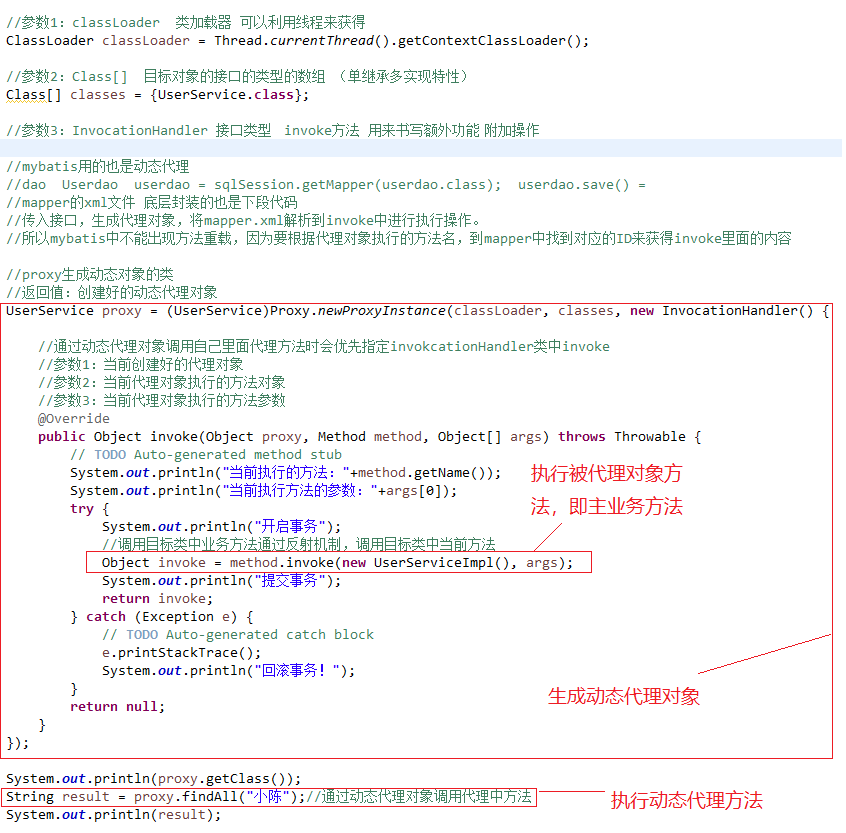
定义：通过在程序运行的过程中动态的为项目中某些组件生成代理对象，通过代理对象执行附加操作或额外功能，减少通用代码的冗余问题。

1. 首先明确什么是代理？

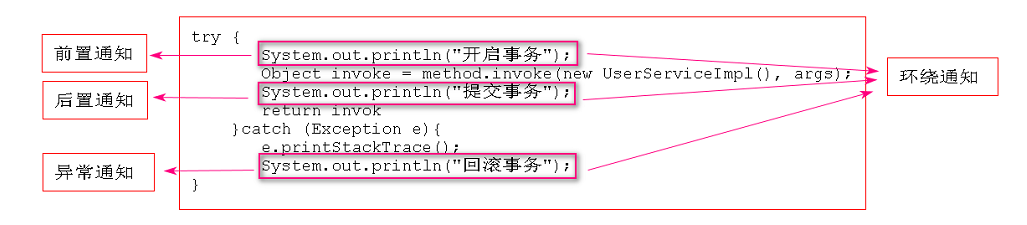
在日常工作中，存在这种情况，例如A类执行了主要的业务逻辑（从数据库按条件A查询listA,条件B查询listB等结果），然后我们又需要在这主要业务逻辑的基础上将结果输出到页面、转为WORD、PDF、XLS等格式将查询到的结果输出到本地，这个时候我们就需要创建专门转word的类B，转PDF的类C，转xls的类D等，我们就把这些手动创建类（B、C、D）叫做A的静态代理。

所以我们可以这么理解代理，依附于主业务类，在主业务类基础上且和主业务类用同一个接口，可以增加附加操作的类，叫做代理，手动创建的叫静态代理，由代码动态创建的叫动态代理。具体demo见本人代码或操作文档。

1. 动态代理的具体理解
2. 通过代码动态的为目标类生成代理对象，生成的对象就叫动态代理对象
3. 动态代理的底层原理就是反射，具体是Proxy.*newProxyInstance（）方法*来传入对应参数生成代理对象， 并实现附加的一系列操作，具体代码如下：



1. Spring 的AOP底层封装了Proxy.*newProxyInstance（）* 生成动态代理对象的方法，利用反射生成代理对象，诞生了很多新名词
2. 把对被代理对象的附加操作，叫做通知。（环绕通知、前置通知、后置通知、异常通知）



1. 把被代理对象的被代理方法叫做切入点
2. 通知（附加操作）+切入点（被代理方法）叫做切面。AOP又叫切面编程
3. 所以AOP的实现只需要传入切入点和通知即可。
4. 好处：在保证原业务不变的情况下，利用代理对象完成附加操作，实现了核心业务和附加业务的解耦，通常用来解决日志打印等冗余重复逻辑的代码。
5. Mybatis底层也是利用的动态代理，传入dao接口，并将mapper.XML文件解析,通过xml文件里面的ID寻找与接口方法对应的sql，将sql作为附加操作，完成动态代理对象的生成，所以mybatis有个不能进行重载的特性，因为xml里面的ID会重复，影响动态代理对像的生成。
6. AOP的具体实现方式

最原始的实现方式：

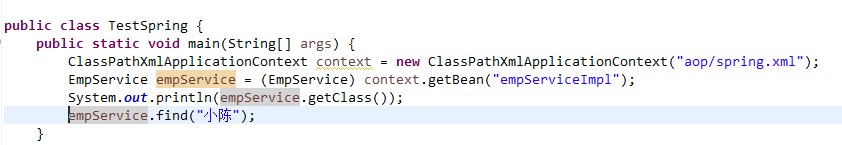
1. 实现通知的接口，完成附加操作的添加



1. 配置文件中，配置通知类，并通过aop标签来完成切入点和切面的配置，切入点表达式常用两种（execution方法级别，within类级别）

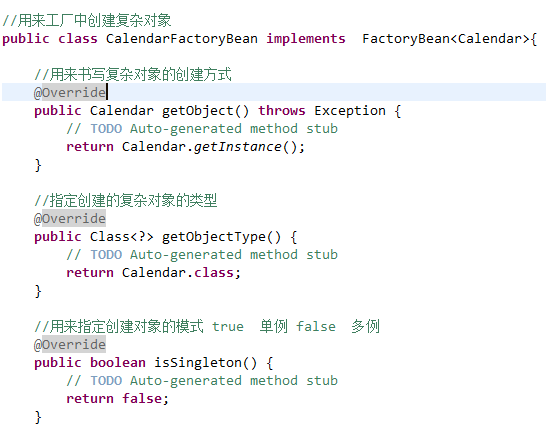


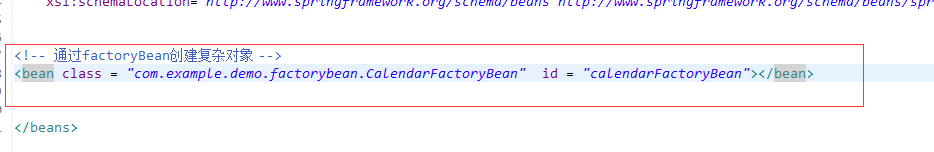
1. 然后对已经配置了AOP切面的方法的调用，其实调用的是底层生成的代理对象，并不是真正调用的原对象。

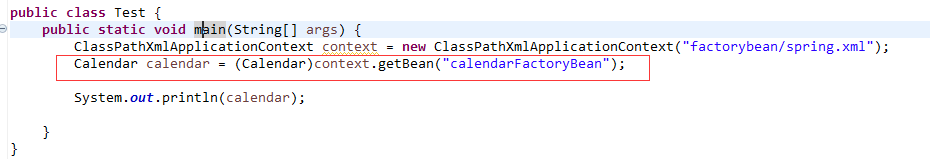




1. 环绕通知，异常通知等通知，见本人demo
2. **Bean标签的补充→复杂对象的创建**
3. 简单对象：可以通过new创建的叫简单对象，如实体，service等，可以直接用bean标签管理
4. 复杂对象：不能通过new创建的叫复杂对象，如接口（Connection等），抽象类（Calendar、MessageDisgest），不可以直接交给Bean标签进行管理。
5. 复杂对象IOC管理：手动创建该类的管理类A，实现FactoryBean<T>，泛型中填入复杂对象，并重写对应的方法（创建方式、复杂对象类型，单例或多例创建），然后A就可以用Bean标签管理，复杂类型就可以和简单类型一样使用了。



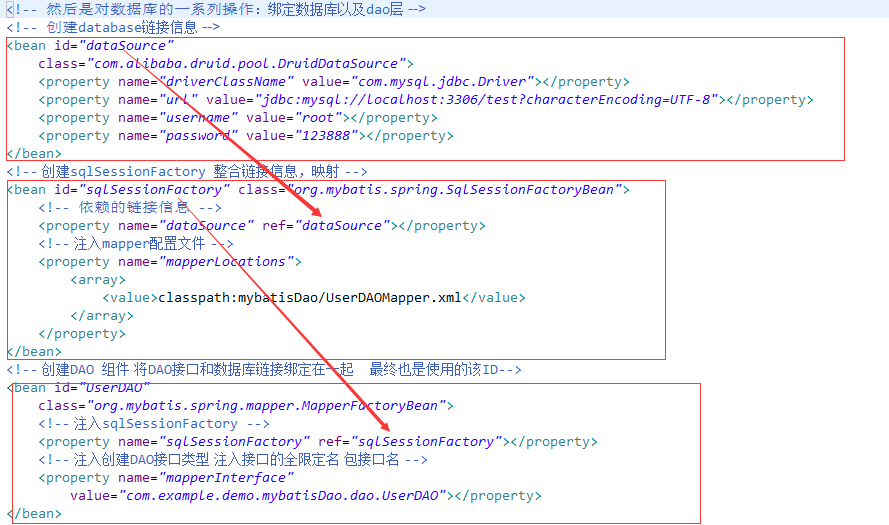


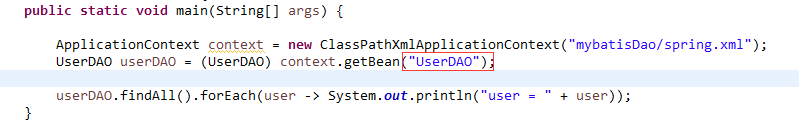


### 至此，IOC和AOP部分完结

## Spring整合mybatis

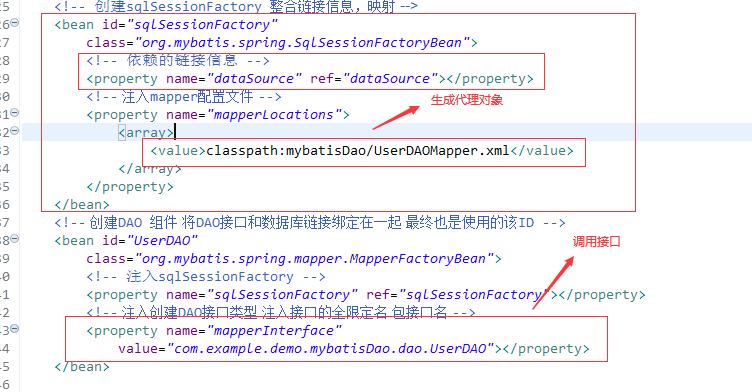
1. spring整合mybatis具体思路就是:将mybatis管理对象的能力交给spring,由spring统一管理，而mybatis所有的组件都来源于SQLSessionFactory，所以接管该对象就接管了mybatis的管理组件权力。
2. SQLSessionFactory是接口（复杂对象），所以不能直接用bean标签管理，要实现factoryBean<T>接口并加载mybatis配置文件，用bean标签来管理，由于上述步骤是重复不变的代码，mybatis公司就整合成新的jar包，提供简单对象SQLSessionFactoryBean,来减少代码量，但是这次数据源文件（填链接用户密码的文件）和mapper文件必须分开加载，这也是springboot加载数据库的雏形。





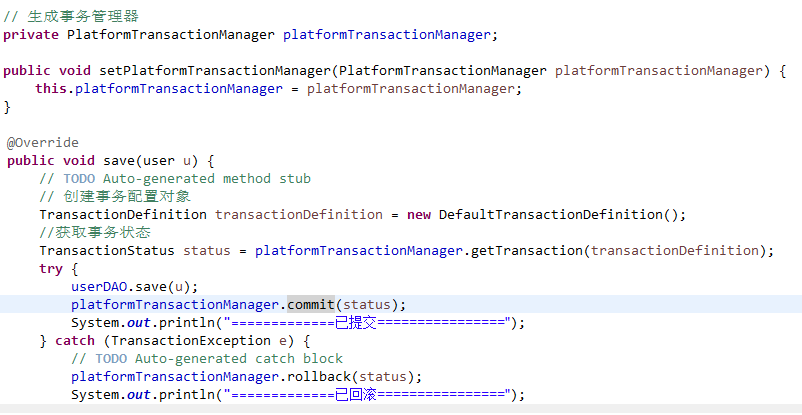
## mybatis和事务的交互

1. 首先事务是JDBC层面的，mybatis对事务的封装最底层的是connection接口,核心就是connection.commit()--------connection.rollback()。
2. 我们通常是在业务层（service层）添加事务，但是在service调用的dao层接口用的connection和mybatis生成的代理对象（mybatis用的也是动态代理）用的connection，必须保证是connection的同一线程，即用的是同一个connection，这样事务的回滚等操作才有意义（其实还是没太搞明白）,也就是说由于单个connection就可以直接用commit和rollback来操作了，但是当多个connection就不可以这样了。所以事务的存在就是为了保证同一线程链接对象（connection）的一致性。



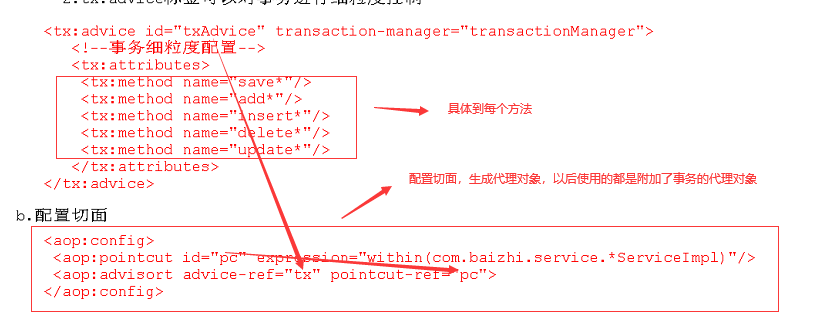
3.mybatis做了一个类*DataSourceTransactionManager，*全局的事务管理器，用来保证业务层当前线程使用连接对象（接口）和DAO层实现连接对象一致，即connection一致。





1. 起初我们需要为每一个方法编写事务，即上图所示，为每一个需要事务的方法手动的进行commit和rollback操作，称为编程式事务处理。由于代码重复冗余问题，我们就交给AOP利用反射生成代理对象来做，如下图，我们把这个事务叫做：声明式事务处理。

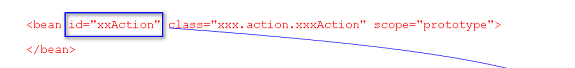
由于通知，即附加操作都一样，都是进行commit和异常rollback操作，spring对此进行进一步封装，tx标签来取代原先手动写环绕通知的情况，并且默认是不加事务的，必须精确到方法名才行。如下图所示



1. 事务的传播属性，本质是将同一个链接（connecttion）传递给其他方法，以此来达到共用链接的目的，且公用同一个事务，所有相互调用的方法，要么全部成功，要么全部回滚。
2. 此外事务还有读写（readonly）和异常性（rollback-for && no-rollback-for=）和超时性（timeout）

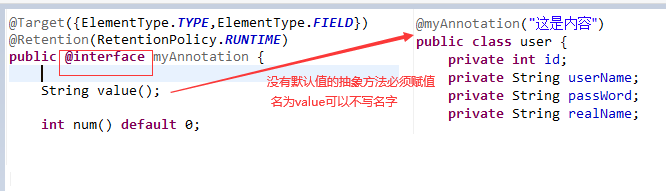
## Spring整合structs2

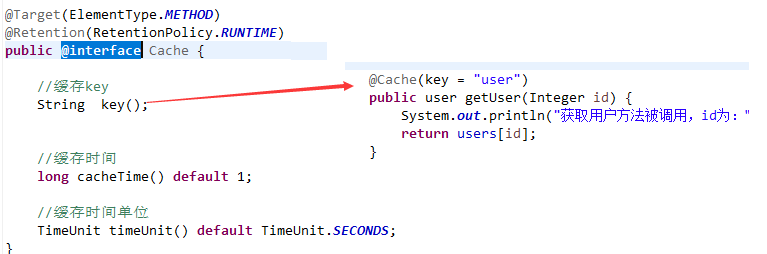
1. Spring整合structs2,同整合mybatis一样，就是将创建核心对象的权力交给spring管理，由spring来创建
2. Structs2的核心对象是action,且是简单对象，所以可以直接用bean标签来管理。



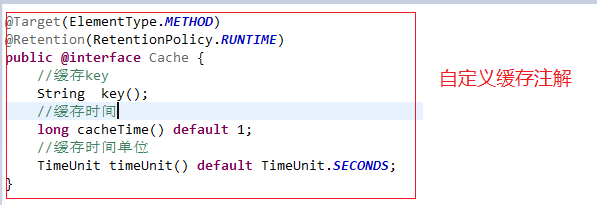
## Spring注解

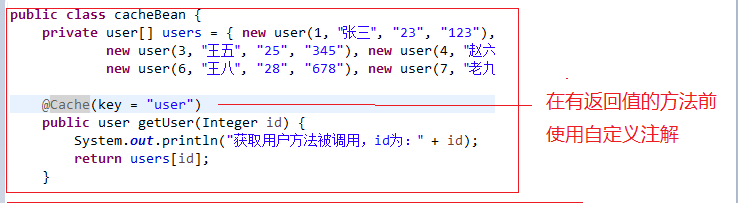
1. 注解是一种特殊的接口**@interface**默认继承Annotation接口，内部的抽象方法作为可赋值的属性，含有默认值的可以不赋值，方法名为value的赋值时可以不写方法名。

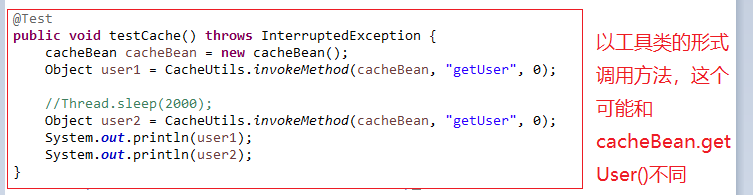




1. 注解只是一个可赋值的标识，必须配合封装好的代码才能起作用。
2. 封装好的代码中，利用反射，通过getAnnotation得到注解的类型和值，注解的作用域，被注解修饰的类或方法，来完成一系列操作。





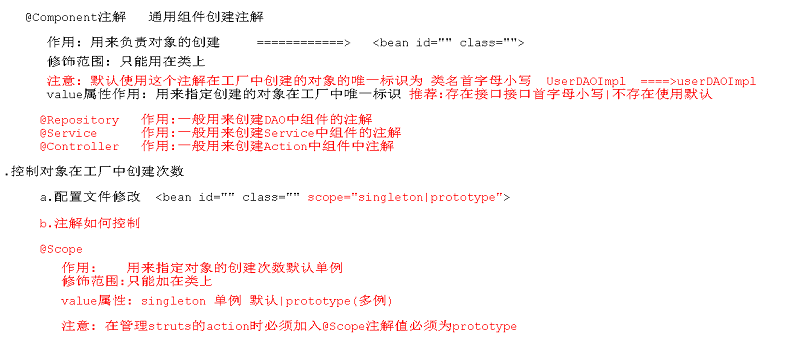




1. 通过在spring配置文件中开启注解扫描，用注解的方式来代替配置文件中繁杂的配置，



如@component来代替bean标签，@autowire来代替property，完成属性注入，并且省去了公开的set方法，@Transactional来代替声明式事务）（AOP）原先的一堆配置，而且遵从局部优先原则（即如果A类上有事务,A类中的方法上也有事务，优先使用方法上的事务）



1. 自定义注解实现，就是实现@interface接口，然后定义它的反射解析类即可，见本人的demo