# 实现原理

动态代理

# 使用场景

场景一： 记录日志

场景二： 监控方法运行时间 （监控性能）

场景三： 权限控制

场景四： 缓存优化 （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需 要查询数据库 ）

场景五： 事务管理 （调用方法前开启事务， 调用方法后提交关闭事务 ）

新的场景：动态数据源切换。自定义注解+AOP实现。

# 源码分析

## 分析AOP源码之前

1. 在使用applicationContext相关实现类加载bean的时候，会针对所有单例并非懒加载的bean，在构造ApplicationContext的时候就会创建好这些bean，而不会等到使用的时候才会创建。
2. BeanPostProcessor的使用，在初始化bean的时候都会调用这些类的方法，一般用于在bean初始化前后对bean做一些修改。而AOP的功能实现就是基于此，在bean初始化后创建针对该bean的proxy，然后返回给用户该proxy。

## 基本过程

1）通过AspectJAutoProxyBeanDefinitionParser类将AnnotationAwareAspectJAutoProxyCreator注册到Spring容器中

    2）AnnotationAwareAspectJAutoProxyCreator类的postProcessAfterInitialization()方法将所有有advice的bean重新包装成proxy

    3）调用bean方法时通过proxy来调用，proxy依次调用增强器的相关方法，来实现方法切入

## 拦截器

如果是AOP，则生成代理对象，当代理对象调用代理方法时，增强行为也就是拦截器是如何发挥作用的呢？

当没有拦截器的时候，也就是没有代理对象的时候，通过反射的方式直接调用目标对象方法。

当有拦截器时，进入到invocation.proceed方法。AOP的核心部分。这些实现封装在AOP拦截器链中，有一个个具体的拦截器实现的。

## invocation.proceed()--AOP核心

从索引为-1的拦截器开始调用，按顺序递增，直到没有拦截器了，然后开始调用目标对象的方法。

如果和定义的pointcut匹配，那么这个advice将得到执行。如果匹配不成功，匹配下一个拦截器。

## XML中定义的通知时如何配置到代理对象中的拦截器链中的？

都在interceptorsAndDynamicMethodMatchers这个list集合中。

这个集合从哪个地方获取的呢？

我们可以进入JdkDynamicAopProxy类的invoke方法，有这么一行代码：List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);用于获取这个代理方法的拦截器链，进入此方法：

# 事务回滚

## 事务控制的3中方式

* 编程式事务：就是直接在代码里手动开启事务，手动提交，手动回滚。优点就是可以灵活控制，缺点就是太麻烦了，太多重复的代码了。
* 声明式事务：就是使用SpringAop配置事务，这种方式大大的简化了编码。需要注意的是切入点表达式一定要写正确。
* 注解事务：直接在Service层的方法上面加上@Transactional注解，个人比较喜欢用这种方式。

## 事务不回滚的情况

声明式事务和注解事务回滚的原理：当被切面切中或者加了注解的方法中抛出了RuntimeException异常时，spring会进行事务的回滚。默认情况下捕获到方法的RuntimeException异常，也就是说抛出运行时异常(即RuntimeException及其子类)都能回滚，但抛出一个不属于运行时异常时，事务就不会回滚。

不回滚的原因：

1. 声明式事务配置切入点表达式写错了，没切中service中的方法。
2. Service方法中，把异常try catch了，但catch中只是打印了异常信息，没有手动抛出RuntimeException异常。
3. Service方法中，抛出的异常不属于运行时异常。

## 如何保证事务回滚

1. 如果service层抛出不属于运行时异常也要回滚，那么可以将spring默认的回滚时异常修改为Exception，这样保证无论碰到什么异常都可以回滚
2. 声明式事务，在配置里面添加一个rollback-for

<tx:method name="update\*" propagation="REQUIRED" rollback-for="java.lang.Exception"/>

1. 注解事务，直接在注解上面指定

@Transactional(rollbackFor=Exception.class)

1. 只有非只读事务才能回滚，只读事务是不会回滚的
2. 如果在service层用了try catch，在catch里面再抛出一个RuntimeException异常才会回滚
3. 还可以再catch后面加一句回滚代码TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();）这样就可以再抛出异常后也能return返回值。

# 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制

反射的三种方式：

实例.getClass();

类.class();

Class.forName();

一般采用第三种方式：像数据库加载驱动就是用的此种方式。

# 代理

## 静态代理

静态代理是在程序运行前生成的，动态代理是在程序运行过程中，利用反射机制动态实现的，无需程序员手工编写代码。

## 动态代理

JDK动态代理只能对实现了接口的类生成代理，而不能针对类。

CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，

并覆盖其中方法实现增强，但是因为采用的是继承，所以该类或方法最好不要声明成 final，对于final类或方法，是无法继承的。

使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，

在jdk6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，

因为CGLib原理是动态生成被代理类的子类。在jdk6、jdk7、jdk8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLIB代理效率，

只有当进行大量调用的时候，jdk6和jdk7比CGLIB代理效率低一点，但是到jdk8的时候，jdk代理效率高于CGLIB代理，总之，每一次jdk版本升级，jdk代理效率都得到提升，而CGLIB代理消息确有点跟不上步伐。

当Bean实现接口时，Spring就会用JDK的动态代理。

当Bean没有实现接口时，Spring使用CGlib是实现。

可以强制使用CGlib（在spring配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。