1. 描述一下Spring AOP

Spring AOP(Aspect Oriented Programming，面向切面编程)是OOPs(面向对象编程)的补充，它也提供了模块化。在面向对象编程中，关键的单元是对象，AOP的关键单元是切面，或者说关注点（可以简单地理解为你程序中的独立模块）。一些切面可能有集中的代码，但是有些可能被分散或者混杂在一起，例如日志或者事务。**这些分散的切面被称为横切关注点**。一个横切关注点是一个可以影响到整个应用的关注点，而且应该被尽量地集中到代码的一个地方，例如事务管理、权限、日志、安全等。

AOP让你可以使用简单可插拔的配置，在实际逻辑执行之前、之后或周围动态添加横切关注点。这让代码在当下和将来都变得易于维护。如果你是使用XML来使用切面的话，要添加或删除关注点，你不用重新编译完整的源代码，而仅仅需要修改配置文件就可以了。

Spring AOP通过以下两种方式来使用。但是最广泛使用的方式是Spring AspectJ注解风格(Spring AspectJ Annotation Style)

- 使用AspectJ 注解风格
- 使用Spring XML 配置风格

> https://blog.csdn.net/dadiyang/article/details/82920139

2. AOP概念

首先让我们从一些重要的AOP概念和术语开始。这些术语不是Spring特有的。不过AOP术语并不是特别的直观，如果Spring使用自己的术语，将会变得更加令人困惑。

- *切面（Aspect）：一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是J2EE应用中一个关于横切关注点的很好的例子。在Spring AOP中，切面可以使用基于模式）或者基于@Aspect注解的方式来实现。*
- *连接点（Joinpoint）：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。在Spring AOP中，一个连接点总是表示一个方法的执行。*

- *通知（Advice）：在切面的某个特定的连接点上执行的动作。其中包括了“around”、“before”和“after”等不同类型的通知（通知的类型将在后面部分进行讨论）。许多AOP框架（包括Spring）都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。*
- *切入点（Pointcut）：匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。*
- *引入（Introduction）：用来给一个类型声明额外的方法或属性（也被称为连接类型声明（inter-type declaration））。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用引入来使一个bean实现`IsModified`接口，以便简化缓存机制。*
- *目标对象（Target Object）： 被一个或者多个切面所通知的对象。也被称做被通知（advised）对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个被代理（proxied）对象。*
- *AOP代理（AOP Proxy）：AOP框架创建的对象，用来实现切面契约（例如通知方法执行等等）。在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。*
- *织入（Weaving）：把切面连接到其它的应用程序类型或者对象上，并创建一个被通知的对象。这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。*

通知类型：

- *前置通知（Before advice）：在某连接点之前执行的通知，但这个通知不能阻止连接点之前的执行流程（除非它抛出一个异常）。*
- *后置通知（After returning advice）：在某连接点正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。*
- *异常通知（After throwing advice）：在方法抛出异常退出时执行的通知。*
- *最终通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。*
- *环绕通知（Around Advice）：包围一个连接点的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它自己的返回值或抛出异常来结束执行。*

环绕通知是最常用的通知类型。和AspectJ一样，Spring提供所有类型的通知，我们推荐你使用尽可能简单的通知类型来实现需要的功能。例如，如果你只是需要一个方法的返回值来更新缓存，最好使用后置通知而不是环绕通知，尽管环绕通知也能完成同样的事情。用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。比如，你不需要在`JoinPoint`上调用用于环绕通知的`proceed()`方法，就不会有调用的问题。

在Spring 2.0中，所有的通知参数都是静态类型，因此你可以使用合适的类型（例如一个方法执行后的返回值类型）作为通知的参数而不是使用Object数组。

通过切入点匹配连接点的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得通知可以独立对应到面向对象的层次结构中。例如，一个提供声明式事务管理 的环绕通知可以被应用到一组横跨多个对象的方法上（例如服务层的所有业务操作）。

> http://shouce.jb51.net/spring/aop.html

3. 两种动态代理实现

为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。,动态代理则不需要写代理类,直接生成到内存中

Jdk代理：基于接口的代理，一定是基于接口，会生成目标对象的接口的子对象。

Cglib代理：基于类的代理，不需要基于接口，会生成目标对象的子对象。

> jdk代理对象实质上是目标对象接口的实现类
> Cglib代理对象实质上是目标对象的子类
> 对接口创建代理优于对类创建代理，因为会产生更加松耦合的系统，所以spring默认是使用JDK代理。
> 对类代理是让遗留系统或无法实现接口的第三方类库同样可以得到通知，这种方式应该是备用方案。
> 来源: https://www.zhihu.com/question/34301445

3.1 jdk代理

三、jdk中的动态代理

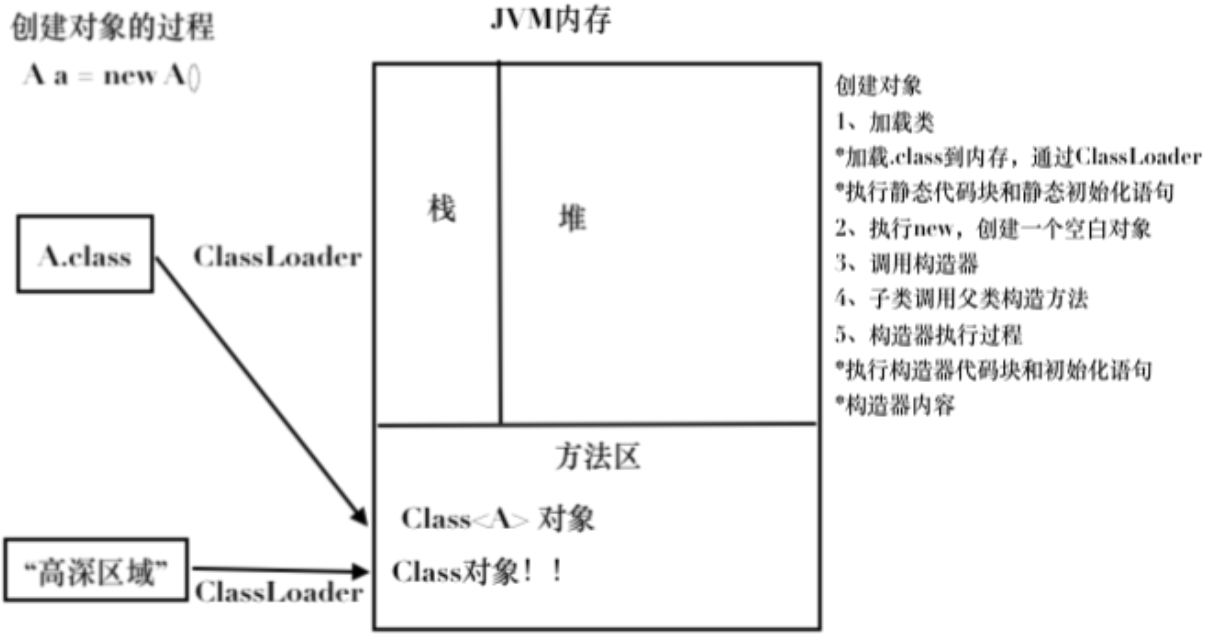
1、Java.lang.reflect.Proxy类可以直接生成一个代理对象

- Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)生成一个代理对象
    - 参数1:ClassLoader loader 代理对象的类加载器 一般使用被代理对象的类加载器
    - 参数2:Class<?>[] interfaces 代理对象的要实现的接口 一般使用的被代理对象实现的接口
    - 参数3:InvocationHandler h (接口)执行处理类

- InvocationHandler中的invoke(Object proxy, Method method, Object[] args)方法：调用代理类的任何方法，此方法都会执行
    - 参数3.1:代理对象(慎用)
    - 参数3.2:当前执行的方法
    - 参数3.3:当前执行的方法运行时传递过来的参数
        - 返回值:当前方法执行的返回值

2、Proxy.newProxyInstance方法参数介绍

- ClassLoader：类加载器!
    - 它是用来加载器的，把.class文件加载到内存，形成Class对象!

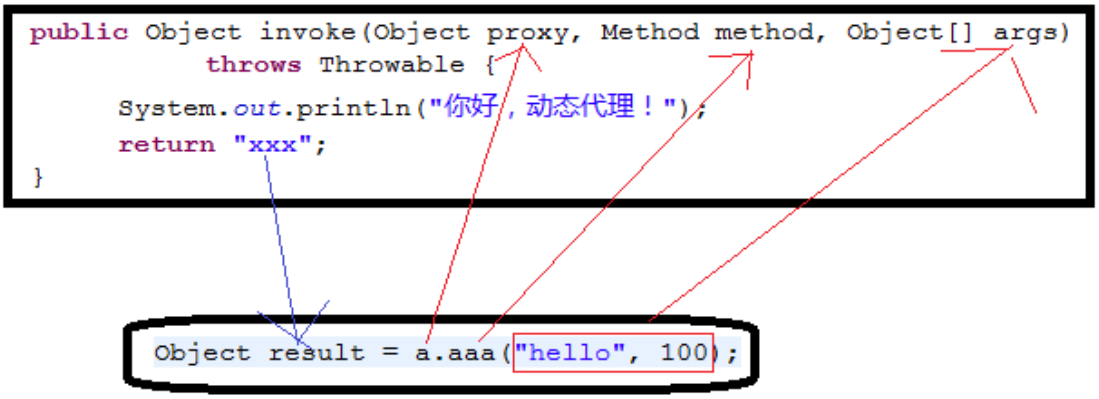


- Class[] interfaces：指定要实现的接口们
- InvocationHandler：代理对象的所有方法(个别不执行，getClass())都会调用InvocationHandler的invoke()方法。

3、InvocationHandler方法参数介绍

```
public Object invoke(Object proxy, Method method, Object[] args);
```

这个invoke()方法在什么时候被调用！在调用代理对象所实现接口中的方法时



- Object proxy：当前对象，即代理对象！在调用谁的方法！
- Method method：当前被调用的方法（目标方法）
- Object[] args：实参！

> AOP都把它用成工厂了,把要代理的类放入工厂,在设置工厂里的增强方法,就可以无限增强任意类,简直强无敌
> https://www.cnblogs.com/gdwkong/p/8035120.html

为什么jdk动态代理必须基于接口

原因如下：

1、生成的代理类继承了Proxy，由于java是单继承，所以只能实现接口，通过接口实现

2、从代理模式的设计来说，充分利用了java的多态特性，也符合基于接口编码的规范

当然，jdk在生成代理的参数中也说明了，需要传入对应接口

3.2 cglib代理

JDK动态代理是基于接口的方式，换句话来说就是代理类和目标类都实现同一个接口，那么代理类和目标类的方法名就一样了，这种方式上一篇说过了；CGLib动态代理是代理类去继承目标类，然后重写其中目标类的方法啊，这样也可以保证代理类拥有目标类的同名方法；

CGLIB底层：使用字节码处理框架ASM，来转换字节码并生成新的类。

CGLIB（CODE GENERLIZE LIBRARY）代理是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的所有方法，所以该类或方法不能声明称final的。

如果目标对象没有实现接口，则默认会采用CGLIB代理；
如果目标对象实现了接口，可以强制使用CGLIB实现代理。

> https://www.cnblogs.com/wyq1995/p/10945034.html
> 原文链接：https://blog.csdn.net/upxiaofeng/article/details/79035493