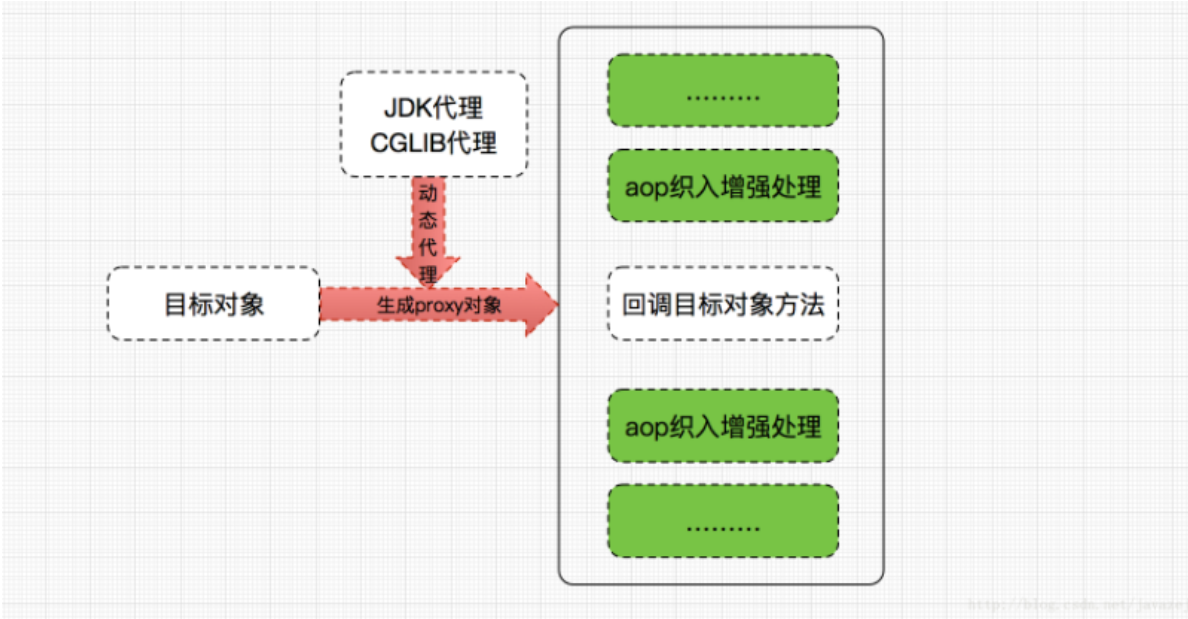
Spring AOP动态代理

- Spring AOP的实现原理是基于动态代理植入的方式。动态代理技术又分为**Java JDK动态代理**和**CGLIB动态代理**，前者是基于反射技术的实现，后者是基于继承的机制实现。
- 代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。代理类负责为委托类预处理消息，过滤消息并转发消息，以及进行消息被委托类执行后的后续处理。



1 Java JDK动态代理

优势

- 代理类和委托类通常会实现相同的接口，所以在访问者看来两者没有丝毫的区别。
- 通过代理类这中间一层，能有效控制对委托类对象的直接访问
- 可以很好地隐藏和保护委托类对象，同时也为实施不同控制策略预留了空间，从而在设计上获得了更大的灵活性

涉及的方法

- Interface InvocationHandler：该接口中仅定义了一个方法
  - public object invoke(Object obj,Method method, Object[] args)
    - 第一个参数obj一般是指代理类
    - method是被代理的方法
    - args为该方法的参数数组

- Proxy：该类即为动态代理类
  - `protected Proxy(InvocationHandler h)`：构造函数，用于给内部的h赋值。
  - `static Class getProxyClass (ClassLoaderloader, Class[] interfaces)`：获得一个代理类，其中loader是类装载器，interfaces是真实类所拥有的全部接口的数组。
  - `static Object newProxyInstance(ClassLoaderloader, Class[] interfaces, InvocationHandler h)`：返回代理类的一个实例，返回后的代理类可以当作被代理类使用

步骤

- 创建一个实现接口InvocationHandler的类，它必须实现invoke方法
- 创建被代理的类以及接口
- 通过Proxy的静态方法newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)创建一个代理
  - loader: 用哪个类加载器去加载代理对象
  - interfaces:动态代理类需要实现的接口
  - h:动态代理方法在执行时，会调用h里面的invoke方法去执行
- 通过代理调用方法

代码实现

- 创建需要动态代理的接口

```
public interface Star {
    /**
     * 明星唱歌
     * @param name 歌名
     */
    void sing(String name);
}
```

- 创建被代理的实际对象

```
public class RealStar implements Star{

    @Override
    public void sing(String name) {
        System.out.println("Star will sing " + name);
    }
}
```

- 调用处理器实现类

```
public class JDKProxy implements InvocationHandler {

    private Object obj;//需要使用被代理类的对象进行赋值

    public JDKProxy(Object obj){
        this.obj = obj;
    }

    public Object createProxy() {
        return Proxy.newProxyInstance(this.obj.getClass().getClassLoader(),
                this.obj.getClass().getInterfaces(), this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        Report.beforeLog();
```

```
        long start = System.currentTimeMillis();
        //method:被代理的该方法
        //obj：被代理的对象
        Object returnValue = method.invoke(obj, args);
        //上述方法的返回值就作为当前类中的invoke()的返回值。
        Report.afterLog();
        System.out.println("该方法花费时长为：" + (System.currentTimeMillis() -
start));
        return returnValue;
    }

}
```

- 测试

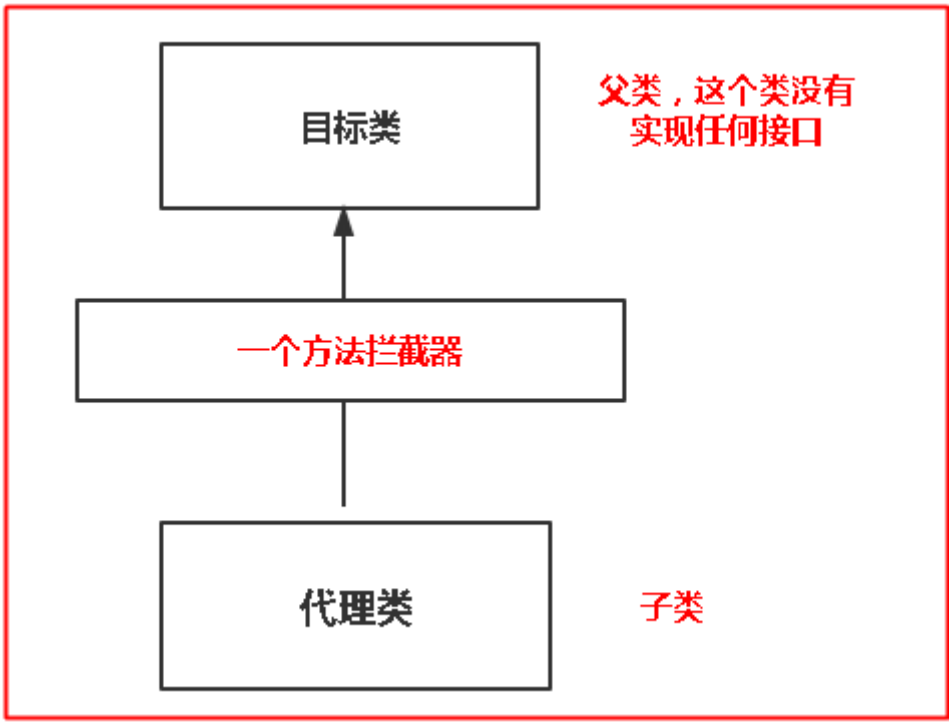
```
@Test
public void JDKProxyTest() {
    //被代理的对象
    RealStar realStar = new RealStar();
    //创建JDK代理，并将 被代理对象 放进JDK代理中
    JDKProxy jdkProxy = new JDKProxy(realStar);
    //通过jdk代理，生成代理对象
    Star proxy = (Star) jdkProxy.createProxy();
    //执行增强之后的方法
    proxy.sing("烟花易冷");
}
```

- 输出

```
[INFO ] 2021-01-06 16:35:06,674 method:proxy.Report.beforeLog(Report.java:10)
在之前调用
Star will sing 烟花易冷
[INFO ] 2021-01-06 16:35:06,676 method:proxy.Report.afterLog(Report.java:14)
在之后调用
该方法花费时长为：1
```

2 CGLIB动态代理

- 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
- 代理类去继承目标类，每次调用代理类的方法都会被方法拦截器拦截，在拦截器中才是调用目标类的该方法的逻辑，结构还是一目了然的；



使用步骤

- 导入依赖

```
<dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2.2</version>
</dependency>
```

- 创建被代理类

```
public class Human {
    public void eat(){
        System.out.println("Human eat");
    }

    public void sleep(){
        System.out.println("Human sleep");
    }
}
```

- 实现MethodInterceptor接口生成方法拦截器

```
public class CGLIBProxy implements MethodInterceptor {
    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        //此处是对方法进行增强
        Report.beforeLog();
        long start = System.currentTimeMillis();
        //这里调用的是invokeSuper()这个方法，并不是反射
        Object object = methodProxy.invokeSuper(o, objects);
```

```
        Report.afterLog();
        System.out.println("该方法花费时长为：" + (System.currentTimeMillis() -
start));
        return object;
    }

    public Human createProxy(){
        //创建Enhancer对象，类似于JDK动态代理的Proxy类，下一步就是设置几个参数
        Enhancer enhancer = new Enhancer();
        //设置目标类的字节码文件
        enhancer.setSuperclass(Human.class);
        //设置回调函数
        enhancer.setCallback(this);
        //这里的create方法就是正式创建代理类
        return (Human) enhancer.create();
    }
}
```

- 测试

```
@Test
public void CGLIBProxyTest() {
    //创建 cglib代理
    CGLIBProxy cglibProxy = new CGLIBProxy();
    //通过代理创建代理对象
    Human proxy = cglibProxy.createProxy();
    //调用增强的代理方法
    proxy.eat();
}
```

- 结果

```
[INFO ] 2021-01-06 17:38:44,523 method:proxy.Report.beforeLog(Report.java:10)
在之前调用
Human eat
[INFO ] 2021-01-06 17:38:44,534 method:proxy.Report.afterLog(Report.java:14)
在之后调用
该方法花费时长为：10
```