我们知道，Spring 中 AOP 是一大核心技术，也是面试中经常会被问到的问题，最近我在网上也看到很多面试题，其中和 Spring AOP 相关的就有不少，这篇文章主要来总结下相关的技术点，希望对大家有用。

0. 几个常见的问题

针对这一块的东西，一般下面几个问题面试官问的比较多：

  Spring AOP用的是哪种设计模式？

谈谈你对代理模式的理解？

静态代理和动态代理有什么区别？

如何实现动态代理？

Spring AOP中用的是哪种代理技术？

如果这些问题都能回答的很流畅的话，说明对代理这一块的基本知识有一定的了解了。因为我们在实际开发中，写业务代码会更多，所以这一块的东西，大部分人可能知道个一二，但是如果让他们很有条理的表达出来，可能就不那么容易了。

1. 什么是 Spring AOP？

一般面试官问到这个问题，面试者基本上都会回答：AOP 就是面向切面编程。其实这真的是句废话，这么回答真的没有任何意义。

或许你可以给面试官举个例子：歌星都有好多助理，歌星最重要的一件事就是唱歌，其他事他不用关注，比如唱歌前可能需要和其他人谈合作，还要布置场地，唱歌后还要收钱等等，这些统统交给他对应的助理去做。也许哪一天，这个歌星做慈善，免费唱歌了，不收钱了，那么就可以把收钱这个助力给辞退了。这就是 AOP，每个人各司其职，灵活组合，达到一种可配置的、可插拔的程序结构。AOP 的实现原理就是代理模式。

在程序中也是如此，通过代理，可以详细控制访问某个或者某类对象的方法，在调用这个方法前做前置处理，调用这个方法后做后置处理。

2. 什么是代理模式？

代理模式的核心作用就是通过代理，控制对对象的访问。它的设计思路是：定义一个抽象角色，让代理角色和真实角色分别去实现它。

真实角色：实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。它只关注真正的业务逻辑，比如歌星唱歌。

代理角色：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并在前后可以附加自己的操作，比如谈合同，布置场地，收钱等等。

这就是代理模式的设计思路。代理模式分为静态代理和动态代理。静态代理是我们自己创建一个代理类，而动态代理是程序自动帮我们生成一个代理，我们就不用管了。下面我详细介绍一下这两种代理模式。

3. 静态代理模式

就举明星唱歌这个例子，根据上面提供的设计思路，首先我们需要创建明星这个抽象角色，

/\*\*

\* 明星接口类

\* @author shengwu ni

\* @date 2018-12-07

\*/

public interface Star {

  /\*\*

   \* 唱歌方法

   \*/

  void sing();

}

静态代理需要创建真实角色和代理角色，分别实现唱歌这个接口，真实角色很简单，直接实现即可，因为真实角色的主要任务就是唱歌。

/\*\*

\* 真实明星类

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class RealStar implements Star {

  @Override

  public void sing() {

      System.out.println("明星本人开始唱歌……");

  }

}

代理类就需要做点工作了，我们思考一下，代理只是在明星唱歌前后做一些准备和收尾的事，唱歌这件事还得明星亲自上阵，代理做不了。所以代理类里面是肯定要将真实的对象传进来。有了思路，我们将代理类写出来。

/\*\*

\* 明星的静态代理类

\*

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class ProxyStar implements Star {

  /\*\*

   \* 接收真实的明星对象

   \*/

  private Star star;

  /\*\*

   \* 通过构造方法传进来真实的明星对象

   \* @param star star

   \*/

  public ProxyStar(Star star) {

      this.star = star;

  }

  @Override

  public void sing() {

      System.out.println("代理先进行谈判……");

      // 唱歌只能明星自己唱

      this.star.sing();

      System.out.println("演出完代理去收钱……");

  }

}

这样的话，逻辑就非常清晰了。在代理类中，可以看到，维护了一个Star对象，通过构造方法传进来一个真实的Star对象给其赋值，然后在唱歌这个方法里，使用真实对象来唱歌。所以说面谈、收钱都是由代理对象来实现的，唱歌是代理对象让真实对象来做。下面写个客户端测试下。

/\*\*

\* 测试客户端

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class Client {

  /\*\*

   \* 测试静态代理结果

   \* @param args args

   \*/

  public static void main(String[] args) {

      Star realStar = new RealStar();

      Star proxy = new ProxyStar(realStar);

      proxy.sing();

  }

}

读者可以自己运行下结果，静态代理比较简单。动态代理比静态代理使用的更广泛，动态代理在本质上，代理类不用我们来管，我们完全交给工具去生成代理类即可。动态代理一般有两种方式：JDK 动态代理和 CGLIB 动态代理。

4. JDK 动态代理

既然动态代理不需要我们去创建代理类，那我们只需要编写一个动态处理器就可以了。真正的代理对象由 JDK 在运行时为我们动态的来创建。

/\*\*

\* 动态代理处理类

\*

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class JdkProxyHandler {

  /\*\*

   \* 用来接收真实明星对象

   \*/

  private Object realStar;

  /\*\*

   \* 通过构造方法传进来真实的明星对象

   \*

   \* @param star star

   \*/

  public JdkProxyHandler(Star star) {

      super();

      this.realStar = star;

  }

  /\*\*

   \* 给真实对象生成一个代理对象实例

   \*

   \* @return Object

   \*/

  public Object getProxyInstance() {

      return Proxy.newProxyInstance(realStar.getClass().getClassLoader(),

              realStar.getClass().getInterfaces(), (proxy, method, args) -> {

                  System.out.println("代理先进行谈判……");

                  // 唱歌需要明星自己来唱

                  Object object = method.invoke(realStar, args);

                  System.out.println("演出完代理去收钱……");

                  return object;

              });

  }

}

这里说一下 Proxy.newProxyInstance() 方法，该方法接收三个参数：第一个参数指定当前目标对象使用的类加载器,获取加载器的方法是固定的；第二个参数指定目标对象实现的接口的类型；第三个参数指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法。网上针对第三个参数的写法都是 new 一个匿名类来处理，我这直接用的 Java8 里面的 lamda 表达式来写的，都一样。底层原理使用的是反射机制。接下来写一个客户端程序来测试下。

/\*\*

\* 测试客户端

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class Client {

  /\*\*

   \* 测试JDK动态代理结果

   \* @param args args

   \*/

  public static void main(String[] args) {

      Star realStar = new RealStar();

      // 创建一个代理对象实例

      Star proxy = (Star) new JdkProxyHandler(realStar).getProxyInstance();

      proxy.sing();

  }

}

可以看出，创建一个真实的对象，送给 JdkProxyHandler 就可以创建一个代理对象了。

我们对 JDK 动态代理做一个简单的总结：相对于静态代理，JDK 动态代理大大减少了我们的开发任务，同时减少了对业务接口的依赖，降低了耦合度。JDK 动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler 来处理。但是 JDK 动态代理有个缺憾，或者说特点：JDK 实现动态代理需要实现类通过接口定义业务方法。也就是说它始终无法摆脱仅支持 interface 代理的桎梏，因为它的设计就注定了这个遗憾。

5. CGLIB 动态代理

由上面的分析可知，JDK 实现动态代理需要实现类通过接口定义业务方法，那对于没有接口的类，如何实现动态代理呢，这就需要 CGLIB 了。

CGLIB 采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是继承，所以不能对final修饰的类进行代理。我们来写一个 CBLIB 代理类。

/\*\*

\* cglib代理处理类

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class CglibProxyHandler implements MethodInterceptor {

  /\*\*

   \* 维护目标对象

   \*/

  private Object target;

  public Object getProxyInstance(final Object target) {

      this.target = target;

      // Enhancer类是CGLIB中的一个字节码增强器，它可以方便的对你想要处理的类进行扩展

      Enhancer enhancer = new Enhancer();

      // 将被代理的对象设置成父类

      enhancer.setSuperclass(this.target.getClass());

      // 回调方法，设置拦截器

      enhancer.setCallback(this);

      // 动态创建一个代理类

      return enhancer.create();

  }

  @Override

  public Object intercept(Object object, Method method, Object[] args,

          MethodProxy methodProxy) throws Throwable {

      System.out.println("代理先进行谈判……");

      // 唱歌需要明星自己来唱

      Object result = methodProxy.invokeSuper(object, args);

      System.out.println("演出完代理去收钱……");

      return result;

  }

}

使用 CGLIB 需要实现 MethodInterceptor 接口，并重写intercept 方法，在该方法中对原始要执行的方法前后做增强处理。该类的代理对象可以使用代码中的字节码增强器来获取。接下来写个客户端测试程序。

/\*\*

\* 测试客户端

\* @author shengwu ni

\* @date 2018-12-08

\*/

public class Client {

  /\*\*

   \* 测试Cglib动态代理结果

   \* @param args args

   \*/

  public static void main(String[] args) {

      Star realStar = new RealStar();

      Star proxy = (Star) new CglibProxyHandler().getProxyInstance(realStar);

      proxy.sing();

  }

}

这个客户端测试程序和 JDK 动态代理的逻辑一模一样，所以也可以看出，代理模式中的动态代理，其实套路都是相同的，只是使用了不同的技术而已。

我们也对 CGLIB 动态代理做一下总结：CGLIB 创建的动态代理对象比 JDK 创建的动态代理对象的性能更高，但是 CGLIB 创建代理对象时所花费的时间却比 JDK 多得多。所以对于单例的对象，因为无需频繁创建对象，用 CGLIB 合适，反之使用JDK方式要更为合适一些。同时由于 CGLIB 由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。

当然了，不管是哪种动态代理技术，在上面的代码里，要代理的类中可能不止一种方法，有时候我们需要对特定的方法进行增强处理，所以可以对传入的 method 参数进行方法名的判断，再做相应的处理。

6. Spring AOP 采用哪种代理？

JDK 动态代理和 CGLIB 动态代理均是实现 Spring AOP 的基础。对于这一块内容，面试官问的比较多，他们往往更想听听面试者是怎么回答的，有没有看过这一块的源码等等。

针对于这一块内容，我们看一下 Spring 5 中对应的源码是怎么说的。

public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

@Override

public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {

  if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {

    Class<?> targetClass = config.getTargetClass();

    if (targetClass == null) {

      throw new AopConfigException("TargetSource cannot determine target class: " +

          "Either an interface or a target is required for proxy creation.");

    }

      // 判断目标类是否是接口或者目标类是否Proxy类型，若是则使用JDK动态代理

    if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {

      return new JdkDynamicAopProxy(config);

    }

      // 配置了使用CGLIB进行动态代理或者目标类没有接口，那么使用CGLIB的方式创建代理对象

    return new ObjenesisCglibAopProxy(config);

  }

  else {

      // 上面的三个方法没有一个为true，那使用JDK的提供的代理方式生成代理对象

    return new JdkDynamicAopProxy(config);

  }

}

  //其他方法略……

}

从上述源码片段可以看出，是否使用 CGLIB 是在代码中进行判断的，判断条件是 config.isOptimize()、config.isProxyTargetClass() 和 hasNoUserSuppliedProxyInterfaces(config)。

其中，config.isOptimize() 与 config.isProxyTargetClass()默认返回都是 false，这种情况下判断结果就由hasNoUserSuppliedProxyInterfaces(config)的结果决定了。

简单来说，

hasNoUserSuppliedProxyInterfaces(config) 就是在判断代理的对象是否有实现接口，有实现接口的话直接走 JDK 分支，即使用 JDK 的动态代理。

所以基本上可以总结出 Spring AOP 中的代理使用逻辑了：如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实现 AOP；如果目标对象没有实现了接口，则采用 CGLIB 库，Spring 会自动在 JDK 动态代理和 CGLIB 动态代理之间转换。