

# Java 静态代理、Java动态代理、CGLIB动态代理

3 2019.03.10 10:29:30 字数 3281 阅读 1550

[TOC]

## 开篇

Java 的代理就是客户类不再直接和委托类打交道,而是通过一个中间层来访问,这个中间层就是代理。为啥要这样呢,是因为使用代理有2个优势：

- 可以隐藏委托类的实现
- 可以实现客户与委托类之间的解耦,在不修改委托类代码的情况下能够做一些额外的处理

我们举个很常见的例子:工厂会生产很多的玩具,但是我们买玩具都是到商店买的,而不是到工厂去买的,工厂怎么生产我们并不关心,我们只知道到商店可以买到自己想要的玩具，并且，如果我们需要送人的话商店可以把这些玩具使用礼品盒包装。这个工厂就是委托类,商店就是代理类,我们就是客户类。

在 Java 中我们有很多场景需要使用代理类,比如远程 RPC 调用的时候我们就是通过代理类去实现的,还有 Spring 的 AOP 切面中我们也是为切面生成了一个代理类等等。  
代理类主要分为静态代理、JDK 动态代理和 CGLIB 动态代理，它们各有优缺点，没有最好的,存在就是有意义的，在不同的场景下它们会有不同的用武之地。

## 1. Java 静态代理

首先,定义接口和接口的实现类,然后定义接口的代理对象,将接口的实例注入到代理对象中,然后通过代理对象去调用真正的实现类，实现过程非常简单也比较容易理解,静态代理的代理关系在编译期间就已经确定了的。它适合于代理类较少且确定的情况。它可实现在怒修改委托类代码的情况下做一些额外的处理，比如包装礼盒，实现客户类与委托类的解耦。缺点是只适用委托方法少的情况下,试想一下如果委托类有几百上千个方法,岂不是很难受,要在代理类中写一堆的代理方法。这个需求动态代理可以搞定

```
1 // 委托接口
2 public interface IHelloService {
3
4     /**
5      * 定义接口方法
6      * @param userName
7      * @return
8      */
9     String sayHello(String userName);
10
11 }
12 // 委托类实现
13 public class HelloService implements IHelloService {
14
15     @Override
16     public String sayHello(String userName) {
17         System.out.println("helloService" + userName);
18         return "HelloService" + userName;
19     }
20 }
```

写下你的评论...

评论2 赞45 ...

华为云

元试用 [4]

数据库、安全、

立即体验

isoleHero

拥有32钻

华为云

元试用 [4]

数据库、安全、

立即体验

```
21
22 // 代理类
23 public class StaticProxyHello implements IHelloService {
24
25     private IHelloService helloService = new HelloService();
26
27     @Override
28     public String sayHello(String userName) {
29         /** 代理对象可以在此处包装一下*/
30         System.out.println("代理对象包装礼盒...");
31         return helloService.sayHello(userName);
32     }
33 }
34 // 测试静态代理类
35 public class MainStatic {
36     public static void main(String[] args) {
37         StaticProxyHello staticProxyHello = new StaticProxyHello();
38         staticProxyHello.sayHello("isole");
39     }
40 }
```

## 2. 动态代理技术

代理类在程序运行时创建的代理方式被成为 **动态代理**。在了解动态代理之前,我们先简回顾一下 JVM 的类加载机制中的加载阶段要做的三件事情 (附 [Java中的类加载器](#))

1. 通过一个类的全名或其它途径来获取这个类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在内存中生成一个代表这个类的 Class 对象,作为方法区中对这个类访问的入口

而我们要说的动态代理，主要就发生在第一个阶段,这个阶段类的二进制字节流的来源可以有很多,比如 zip包、网络、**运行时计算生成**、其它文件生成(JSP)、数据库获取。其中运行时计算生成就是我们所说的动态代理技术，在 Proxy 类中,就是运用了 ProxyGenerator.generateProxyClass 来为特定接口生成形式为 **\*\$Proxy** 的代理类的二进制字节流。所谓的动态代理就是想办法根据接口或者目标对象计算出 **代理类** 的字节码然后加载进 JVM 中。实际计算的情况会很复杂，我们借助一些诸如 JDK 动态代理实现、CGLIB第三方库来完成的

另一方面为了让生成的代理类与目标对象(就是委托类)保持一致,我们有2种做法：通过接口的 JDK动态代理 和通过继承类的 CGLIB动态代理。(还有一个使用了ASM框架的javassist太复杂了，我还没研究过,这里 **TODO** 下)

## 3. JDK动态代理

在 Java 的动态代理中,主要涉及2个类, **java.lang.reflect.Proxy** 和 **java.lang.reflect.InvocationHandler**

我们需要一个实现 InvocationHandler 接口的中间类,这个接口只有一个方法 invoke 方法,方法的每个参数的注释如下代码。我们对处理类中的所有方法的调用都会变成对 invoke 方法的调用，这样我们可以在 invoke 方法中添加统一的处理逻辑（也可以根据 method 参数判断是哪个方法）。中间类(实现了 InvocationHandler 的类)有一个委托类对象引用,在Invoke方法中调用了委托类对象的相应方法，通过这种聚合的方式持有委托类对象引用，把外部对 invoke 的调用最终都转为对委托类对象的调用。实际上，中间类与委托类构成了静态代理关系，在这个关系中，中间类是代理类，委托类是委托类。然后代理类与中间类也构成一个静态代理关系，在这个关系中，中间类是委托类，代理类是代理类。也就是说，动态代理关系由两组静态代理关系组成，这就是动态代理的原理。

```

1 public interface InvocationHandler {
2     /**
3      * 调用处理
4      * @param proxy 代理类对象
5      * @param method 标识具体调用的是代理类的哪个方法
6      * @param args 代理类方法的参数
7      */
8     public Object invoke(Object proxy, Method method, Object[] args)
9         throws Throwable;
10 }

```

Demo如下:

```

1 // 委托类接口
2 public interface IHelloService {
3
4     /**
5      * 方法1
6      * @param userName
7      * @return
8      */
9     String sayHello(String userName);
10
11     /**
12      * 方法2
13      * @param userName
14      * @return
15      */
16     String sayByeBye(String userName);
17 }
18 // 委托类
19 public class HelloService implements IHelloService {
20
21     @Override
22     public String sayHello(String userName) {
23         System.out.println(userName + " hello");
24         return userName + " hello";
25     }
26
27     @Override
28     public String sayByeBye(String userName) {
29         System.out.println(userName + " ByeBye");
30         return userName + " ByeBye";
31     }
32 }
33 // 中间类
34 public class JavaProxyInvocationHandler implements InvocationHandler {
35
36     /**
37      * 中间类持有委托类对象的引用,这里会构成一种静态代理关系
38      */
39     private Object obj ;
40
41     /**
42      * 有参构造器,传入委托类的对象
43      * @param obj 委托类的对象
44      */
45     public JavaProxyInvocationHandler(Object obj){
46         this.obj = obj;
47     }
48
49     /**
50      * 动态生成代理类对象,Proxy.newProxyInstance
51      * @return 返回代理类的实例
52      */
53     public Object newProxyInstance() {
54         return Proxy.newProxyInstance(
55             //指定代理对象的类加载器

```

```
58         obj.getClass().getClassLoader(),
59         //代理对象需要实现的接口，可以同时指定多个接口
60         obj.getClass().getInterfaces(),
61         //方法调用的实际处理器，代理对象的方法调用都会转发到这里
62         this);
63     }
64
65
66     /**
67     *
68     * @param proxy 代理对象
69     * @param method 代理方法
70     * @param args 方法的参数
71     * @return
72     * @throws Throwable
73     */
74     @Override
75     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
76         System.out.println("invoke before");
77         Object result = method.invoke(obj, args);
78         System.out.println("invoke after");
79         return result;
80     }
81 }
82 // 测试动态代理类
83 public class MainJavaProxy {
84     public static void main(String[] args) {
85         JavaProxyInvocationHandler proxyInvocationHandler = new JavaProxyInvocationHandler(new HelloService());
86         IHelloService helloService = (IHelloService) proxyInvocationHandler.newProxyInstance();
87         helloService.sayByeBye("paopao");
88         helloService.sayHello("yupao");
89     }
90 }
91 }
92 }
```

在上面的测试动态代理类中,我们调用 Proxy 类的 newProxyInstance 方法来获取一个代理类实例。这个代理类实现了我们指定的接口并且会把方法调用分发到指定的调用处理器。

首先通过 newProxyInstance 方法获取代理类的实例,之后就可以通过这个代理类的实例调用代理类的方法，对代理类的方法调用都会调用中间类(实现了 invocationHandle的类)的invoke 方法，在 invoke 方法中我们调用委托类的对应方法，然后加上自己的处理逻辑。

java 动态代理最大的特点就是动态生成的代理类和委托类实现同一个接口。java 动态代理其实内部是通过反射机制实现的，也就是已知的一个对象，在运行的时候动态调用它的方法，并且调用的时候还可以加一些自己的逻辑在里面。（附: [Java 反射](#)）

### 3.2 Proxy.newProxyInstance 源码阅读

上面说过,Proxy.newProxyInstance 通过反射机制用来动态生成代理类对象,为接口创建一个代理类，这个代理类实现这个接口。具体源码如下：

```
1 public static Object newProxyInstance(ClassLoader loader,
2                                     Class<?>[] interfaces,
3                                     InvocationHandler h)
4     throws IllegalArgumentException
5 {
6     // 检查空指针
7     Objects.requireNonNull(h);
8     // 用原型实例指定创建对象的种类,并且通过拷贝这些原型创建新的对象
9     final Class<?>[] intfs = interfaces.clone();
10    // 获取系统的安全接口,不为空的话需要验证是否允许访问这种关系的代理访问
11    final SecurityManager sm = System.getSecurityManager();
12    if (sm != null) {
13        checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
14    }
15 }
```



```
16      /*
17      * 生成代理类 Class,通过类加载器和接口
18      */
19      Class<?> cl = getProxyClass0(loader, intfs);
20
21      /*
22      * 通过构造器来创建实例
23      */
24      try {
25          if (sm != null) {
26              checkNewProxyPermission(Reflection.getCallerClass(), cl);
27          }
28          //获取所有的构造器
29          final Constructor<?> cons = cl.getConstructor(constructorParams);
30          final InvocationHandler ih = h;
31          // 构造器不是public的话需要设置可以访问
32          if (!Modifier.isPublic(cl.getModifiers())) {
33              AccessController.doPrivileged(new PrivilegedAction<Void>() {
34                  public Void run() {
35                      cons.setAccessible(true);
36                      return null;
37                  }
38              });
39          }
40          // 返回创建的代理类Class的实例对象
41          return cons.newInstance(new Object[] {h});
42      } catch (IllegalAccessException|InstantiationException e) {
43          throw new InternalError(e.toString(), e);
44      } catch (InvocationTargetException e) {
45          Throwable t = e.getCause();
46          if (t instanceof RuntimeException) {
47              throw (RuntimeException) t;
48          } else {
49              throw new InternalError(t.toString(), t);
50          }
51      } catch (NoSuchMethodException e) {
52          throw new InternalError(e.toString(), e);
53      }
54  }
```

## 4. CGLIB 动态代理

JDK 动态代理依赖接口实现，而当我们只有类没有接口的时候就需要使用另一种动态代理技术 CGLIB 动态代理。首先 CGLIB 动态代理是第三方框架实现的，在 maven 工程中我们需要引入 cglib 的包,如下:

```
1  <dependency>
2      <groupId>cglib</groupId>
3      <artifactId>cglib</artifactId>
4      <version>2.2</version>
5  </dependency>
```

CGLIB 代理是针对类来实现代理的，原理是对指定的委托类生成一个子类并重写其中业务方法来实现代理。代理类对象是由 Enhancer 类创建的。CGLIB 创建动态代理类的模式是:

1. 查找目标类上的所有非final的public类型的方法(final的不能被重写)
2. 将这些方法的定义转成字节码
3. 将组成的字节码转换成相应的代理的Class对象然后通过反射获得代理类的实例对象
4. 实现 MethodInterceptor 接口,用来处理对代理类上所有方法的请求

```
1  // 委托类,是一个简单类
2  public class CglibHelloClass {
3      /**
```

```
4      * 方法1
5      * @param userName
6      * @return
7      */
8      public String sayHello(String userName){
9          System.out.println("目标对象的方法执行了");
10         return userName + " sayHello";
11     }
12
13     public String sayByeBye(String userName){
14         System.out.println("目标对象的方法执行了");
15         return userName + " sayByeBye";
16     }
17
18 }
19 /**
20  * CglibInterceptor 用于对方法调用拦截以及回调
21  *
22  */
23 public class CglibInterceptor implements MethodInterceptor {
24     /**
25      * CGLIB 增强类对象，代理类对象是由 Enhancer 类创建的，
26      * Enhancer 是 CGLIB 的字节码增强器，可以很方便的对类进行拓展
27      */
28     private Enhancer enhancer = new Enhancer();
29
30     /**
31      *
32      * @param obj 被代理的对象
33      * @param method 代理的方法
34      * @param args 方法的参数
35      * @param proxy CGLIB方法代理对象
36      * @return cglib生成用来代替Method对象的一个对象，使用MethodProxy比调用JDK自身的Method直接执行方法
37      * @throws Throwable
38      */
39     @Override
40     public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
41         System.out.println("方法调用之前");
42         Object o = proxy.invokeSuper(obj, args);
43         System.out.println("方法调用之后");
44         return o;
45     }
46
47
48     /**
49      * 使用动态代理创建一个代理对象
50      * @param c
51      * @return
52      */
53     public Object newProxyInstance(Class<?> c) {
54         /**
55          * 设置产生的代理对象的父类,增强类型
56          */
57         enhancer.setSuperclass(c);
58         /**
59          * 定义代理逻辑对象为当前对象，要求当前对象实现 MethodInterceptor 接口
60          */
61         enhancer.setCallback(this);
62         /**
63          * 使用默认无参数的构造函数创建目标对象,这是一个前提,被代理的类要提供无参构造方法
64          */
65         return enhancer.create();
66     }
67 }
68
69 //测试类
70 public class MainCglibProxy {
71     public static void main(String[] args) {
72         CglibProxy cglibProxy = new CglibProxy();
73         CglibHelloClass cglibHelloClass = (CglibHelloClass) cglibProxy.newProxyInstance(CglibHelloClass.class,
74         cglibHelloClass.sayHello("isole");
75         cglibHelloClass.sayByeBye("sss");
76     }
77 }
```

对于需要被代理的类，它只是动态生成一个子类以覆盖非final的方法，同时绑定钩子回调自定义的拦截器。值得说的是，它比JDK动态代理还要快。值得注意的是，我们传入目标类作为代理的父类。不同于JDK动态代理，我们不能使用目标对象来创建代理。目标对象只能被 CGLIB 创建。在例子中，默认的空参构造方法被使用来创建目标对象。

## 总结

1. 静态代理比较容易理解,需要被代理的类和代理类实现自同一个接口,然后在代理类中调用真正实现类,并且静态代理的关系在编译期间就已经确定了。而动态代理的关系是在运行期间确定的。静态代理实现简单，适合于代理类较少且确定的情况，而动态代理则给我们提供了更大的灵活性。
2. JDK动态代理所用到的代理类在程序调用到代理类对象时才由JVM真正创建，JVM根据传进来的 业务实现类对象 以及 方法名 ，动态地创建了一个代理类的class文件并被字节码引擎执行，然后通过该代理类对象进行方法调用。我们需要做的，只需指定代理类的预处理、调用后操作即可。
3. 静态代理和动态代理都是基于接口实现的,而对于那些没有提供接口只是提供了实现类的而言,就只能选择CGLIB动态代理了

### 4. JDK动态代理和CGLIB动态代理的区别

JDK动态代理基于Java反射机制实现,必须要实现了接口的业务类才能用这种方法生成代理对象

CGLIB动态代理基于 ASM 框架通过生成业务类的子类来实现

JDK动态代理的优势是最小化依赖关系，减少依赖意味着简化开发和维护并且有JDK自身支持。还可以平滑进行JDK版本升级，代码实现简单。基于CGLIB框架的优势是无须实现接口，达到代理类无侵入，我们只需操作我们关系的类，不必为其它相关类增加工作量，性能比较高。

### 5. 描述代理的几种实现方式? 分别说出优缺点?

代理可以分为 "静态代理" 和 "动态代理", 动态代理又分为 "JDK动态代理" 和 "CGLIB动态代理" 实现。

静态代理：代理对象和实际对象都继承了同一个接口，在代理对象中指向的是实际对象的实例，这样对外暴露的是代理对象而真正调用的是 Real Object.

优点：可以很好的保护实际对象的业务逻辑对外暴露，从而提高安全性。

缺点：不同的接口要有不同的代理类实现，会很冗余

JDK 动态代理：

为了解决静态代理中，生成大量的代理类造成的冗余；

JDK 动态代理只需要实现 InvocationHandler 接口，重写 invoke 方法便可以完成代理的实现，

jdk的代理是利用反射生成代理类 Proxyxx.class 代理类字节码，并生成对象

jdk动态代理之所以只能代理接口是因为代理类本身已经extends了Proxy，而java是不允许多重继承的，但是允许实现多个接口

优点：解决了静态代理中冗余的代理实现类问题。

缺点：JDK 动态代理是基于接口设计实现的，如果没有接口，会抛异常。

CGLIB 代理：

由于 JDK 动态代理限制了只能基于接口设计，而对于没有接口的情况，JDK方式解决不了；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，来完成动态代理的实现。

实现方式实现 MethodInterceptor 接口，重写 intercept 方法，通过 Enhancer 类的回调方法来实现。

但是CGLib在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方法，无法进行代理。

优点：没有接口也能实现动态代理，而且采用字节码增强技术，性能也不错。

缺点：技术实现相对难理解些。



45人点赞 >



2人踩 >



Java基础







isoleHero

拥有32钻

关注



元试用


[ 华为云4核8G云服务器 ]


数据库、安全、存储0元体验


立即体验


广告

被以下专题收入，发现更多相似内容

 知识点

 spring

 Java


 设计模式

推荐阅读

更多精彩内容 >


### 粉丝安利指南：BIGBANG

粉丝安利指南 BIGBANG 文 \\ 阿期 我喜欢他们的时候 其实并没有那么出众 那会儿他们还被称为韩国最丑男团...

 阿期欧巴


### 下一个起点

成长总不是一路向前的，它的客观规律一定是曲线形的，所以当书宇001开始（2017年），至今2018年6月，走到了第...

 书宇YY

### 朋友是珍贵的财富

曾经桌上划线，桌下论剑，现在吃着火锅聊从前。寒假最开心的事，就是在空闲时突然接到老同学的电话，之后一起出去玩玩逛...

 8云8



## 我内心阴暗，但我逼自己阳光

文/金戈子 图/金戈子 （一） 我，是同学口中的心理变态。我，是老师眼中的城府极深。我，是父母讨厌的病...



金戈子

## 企业文化要做“应该做的事”

企业文化一百篇——第三篇 企业文化建设要做应该做的事。这应该做的事，就是培养习惯——在日常工作中培养员工良好的工作...



欢喜慈悲