

05 | 动态代理：面向接口编程，屏蔽RPC处理流程

2020-02-28 何小锋

RPC实战与核心原理

[进入课程 >](#)




讲述：张浩

时长 09:11 大小 7.37M



你好，我是何小锋。上一讲我分享了网络通信，其实要理解起来也很简单，RPC 是用来解决两个应用之间的通信，而网络则是两台机器之间的“桥梁”，只有架好了桥梁，我们才能把请求数据从一端传输另外一端。其实关于网络通信，你只要记住一个关键字就行了——可靠的传输。

那么接着上一讲的内容，我们再来聊聊动态代理在 RPC 里面的应用。

如果我问你，你知道动态代理吗？你可能会如数家珍般地告诉我动态代理的作用以及  处。那我现在接着问你，你在项目中用过动态代理吗？这时候可能有些人就会犹豫了。那我再换一个方式问你，你在项目中有实现过统一拦截的功能吗？比如授权认证、性能统计等等。你可能立马就会想到，我实现过呀，并且我知道可以用 Spring 的 AOP 功能来实现。

没错，进一步再想，在 Spring AOP 里面我们是怎么实现统一拦截的效果呢？并且是在我们不需要改动原有代码的前提下，还能实现非业务逻辑跟业务逻辑的解耦。这里的核心就是采用动态代理技术，通过对字节码进行增强，在方法调用的时候进行拦截，以便于在方法调用前后，增加我们需要的额外处理逻辑。

那话说回来，动态代理跟 RPC 又有什么关系呢？

远程调用的魔法

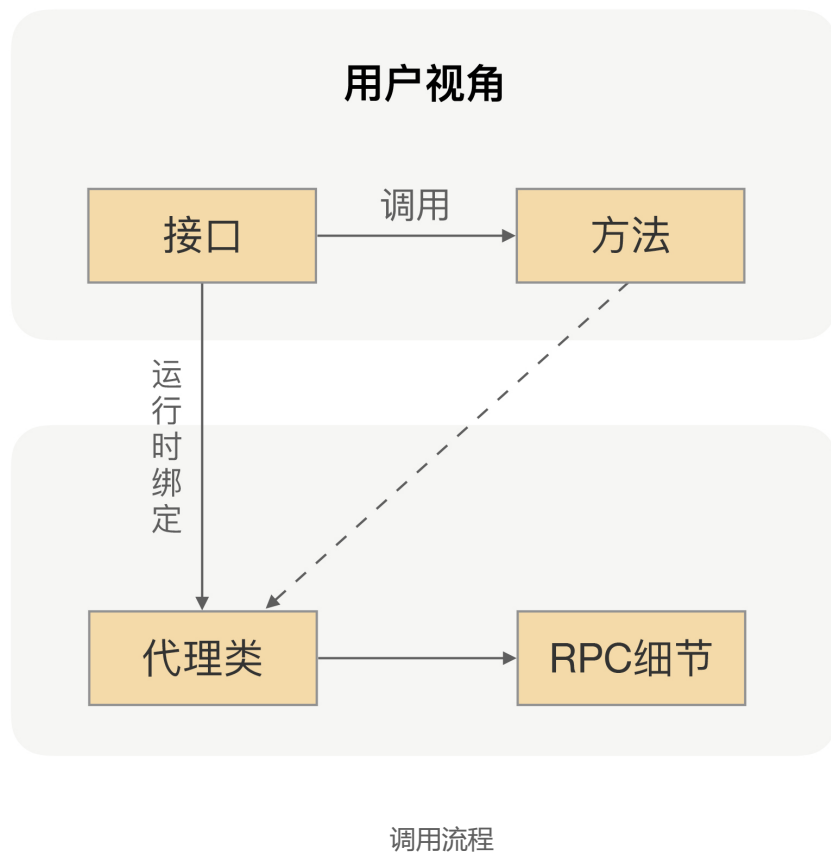
我说个具体的场景，你可能就明白了。

在项目中，当我们要使用 RPC 的时候，我们一般的做法是先找服务提供方要接口，通过 Maven 或者其他工具把接口依赖到我们项目中。我们在编写业务逻辑的时候，如果要调用提供方的接口，我们就只需要通过依赖注入的方式把接口注入到项目中就行了，然后在代码里面直接调用接口的方法。

我们都知道，接口里并不会包含真实的业务逻辑，业务逻辑都在服务提供方应用里，但我们通过调用接口方法，确实拿到了想要的结果，是不是感觉有点神奇呢？想一下，在 RPC 里面，我们是怎么完成这个魔术的。

这里面用到的核心技术就是前面说的动态代理。RPC 会自动给接口生成一个代理类，当我们在项目中注入接口的时候，运行过程中实际绑定的是这个接口生成的代理类。这样在接口方法被调用的时候，它实际上是被生成代理类拦截到了，这样我们就可以在生成的代理类里面，加入远程调用逻辑。

通过这种“偷梁换柱”的手法，就可以帮用户屏蔽远程调用的细节，实现像调用本地一样地调用远程的体验，整体流程如下图所示：



实现原理

动态代理在 RPC 里面的作用，就像是个魔术。现在我不妨给你揭秘一下，我们一起看看这是怎么实现的。之后，学以致用自然就不难了。

我们以 Java 为例，看一个具体例子，代码如下所示：

 复制代码

```
1  /**
2   * 要代理的接口
3   */
4  public interface Hello {
5      String say();
6  }
7
8  /**
9   * 真实调用对象
10  */
11 public class RealHello {
12
13     public String invoke(){
14         return "i'm proxy";
15     }
16 }
17
```

```

18  /**
19   * JDK代理类生成
20   */
21  public class JDKProxy implements InvocationHandler {
22      private Object target;
23
24      JDKProxy(Object target) {
25          this.target = target;
26      }
27
28      @Override
29      public Object invoke(Object proxy, Method method, Object[] paramValues) {
30          return ((RealHello)target).invoke();
31      }
32  }
33
34  /**
35   * 测试例子
36   */
37  public class TestProxy {
38
39      public static void main(String[] args){
40          // 构建代理器
41          JDKProxy proxy = new JDKProxy(new RealHello());
42          ClassLoader classLoader = ClassLoaderUtils.getCurrentClassLoader();
43          // 把生成的代理类保存到文件
44          System.setProperty("sun.misc.ProxyGenerator.saveGeneratedFiles","true");
45          // 生成代理类
46          Hello test = (Hello) Proxy.newProxyInstance(classLoader, new Class[]{Hi
47          // 方法调用
48          System.out.println(test.say());
49      }
50  }

```

这段代码想表达的意思就是：给 Hello 接口生成一个动态代理类，并调用接口 say() 方法，但真实返回的值居然是来自 RealHello 里面的 invoke() 方法返回值。你看，短短 50 行的代码，就完成了这个功能，是不是还挺有意思的？

那既然重点是代理类的生成，那我们就去看下 Proxy.newProxyInstance 里面究竟发生了什么？

一起看下下面的流程图，具体代码细节你可以对照着 JDK 的源码看（上文中有关类和方法，可以直接定位），我是按照 1.7.X 版本梳理的。



代理类生成流程

在生成字节码的那个地方，也就是 `ProxyGenerator.generateProxyClass()` 方法里面，通过代码我们可以看到，里面是用参数 `saveGeneratedFiles` 来控制是否把生成的字节码保存到本地磁盘。同时为了更直观地了解代理的本质，我们需要把参数 `saveGeneratedFiles` 设置成 `true`，但这个参数的值是由 key 为 “`sun.misc.ProxyGenerator.saveGeneratedFiles`” 的 `Property` 来控制的，动态生成的类会保存在工程根目录下的 `com/sun/proxy` 目录里面。现在我们找到刚才生成的 `$Proxy0.class`，通过反编译工具打开 `class` 文件，你会看到这样的代码：

复制代码

```
1 package com.sun.proxy;
2
3 import com.proxy.Hello;
4 import java.lang.reflect.InvocationHandler;
5 import java.lang.reflect.Method;
6 import java.lang.reflect.Proxy;
7 import java.lang.reflect.UndeclaredThrowableException;
8
9 public final class $Proxy0 extends Proxy implements Hello {
10     private static Method m3;
11
12     private static Method m1;
13
14     private static Method m0;
15
16     private static Method m2;
17
18     public $Proxy0(InvocationHandler paramInvocationHandler) {
19         super(paramInvocationHandler);
20     }
21
22     public final String say() {
23         try {
24             return (String)this.h.invoke(this, m3, null);
25         } catch (Error|RuntimeException error) {
26             throw null;
27         } catch (Throwable throwable) {
28             throw new UndeclaredThrowableException(throwable);
29         }
30     }
31 }
```

```
30     }
31 }
32
33 public final boolean equals(Object paramObject) {
34     try {
35         return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject })).l
36     } catch (Error|RuntimeException error) {
37         throw null;
38     } catch (Throwable throwable) {
39         throw new UndeclaredThrowableException(throwable);
40     }
41 }
42
43 public final int hashCode() {
44     try {
45         return ((Integer)this.h.invoke(this, m0, null)).intValue();
46     } catch (Error|RuntimeException error) {
47         throw null;
48     } catch (Throwable throwable) {
49         throw new UndeclaredThrowableException(throwable);
50     }
51 }
52
53 public final String toString() {
54     try {
55         return (String)this.h.invoke(this, m2, null);
56     } catch (Error|RuntimeException error) {
57         throw null;
58     } catch (Throwable throwable) {
59         throw new UndeclaredThrowableException(throwable);
60     }
61 }
62
63 static {
64     try {
65         m3 = Class.forName("com.proxy.Hello").getMethod("say", new Class[0]);
66         m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] {
67         m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]
68         m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]
69         return;
70     } catch (NoSuchMethodException noSuchMethodException) {
71         throw new NoSuchMethodError(noSuchMethodException.getMessage());
72     } catch (ClassNotFoundException classNotFoundException) {
73         throw new NoClassDefFoundError(classNotFoundException.getMessage());
74     }
75 }
76 }
```

我们可以看到 `$Proxy0` 类里面有一个跟 `Hello` 一样签名的 `say()` 方法，其中 `this.h` 绑定的是刚才传入的 `JDKProxy` 对象，所以当我们调用 `Hello.say()` 的时候，其实它是被转发到了 `JDKProxy.invoke()`。到这儿，整个魔术过程就透明了。

实现方法

其实在 Java 领域，除了 JDK 默认的 `InvocationHandler` 能完成代理功能，我们还有很多其他的第三方框架也可以，比如像 `Javassist`、`Byte Buddy` 这样的框架。

单纯从代理功能上来看，JDK 默认的代理功能是有一定的局限性的，它要求被代理的类只能是接口。原因是因为生成的代理类会继承 `Proxy` 类，但 Java 是不支持多重继承的。

这个限制在 RPC 应用场景里面还是挺要紧的，因为对于服务调用方来说，在使用 RPC 的时候本来就是面向接口来编程的，这个我们刚才在前面已经讨论过了。使用 JDK 默认的代理功能，最大的问题就是性能问题。它生成后的代理类是使用反射来完成方法调用的，而这种方式相对直接用编码调用来说，性能会降低，但好在 JDK8 及以上版本对反射调用的性能有很大的提升，所以还是可以期待一下的。

相对 JDK 自带的代理功能，`Javassist` 的定位是能够操纵底层字节码，所以使用起来并不简单，要生成动态代理类恐怕是有点复杂了。但好的方面是，通过 `Javassist` 生成字节码，不需要通过反射完成方法调用，所以性能肯定是更胜一筹的。在使用中，我们要注意一个问题，通过 `Javassist` 生成一个代理类后，此 `CtClass` 对象会被冻结起来，不允许再修改；否则，再次生成时会报错。

`Byte Buddy` 则属于后起之秀，在很多优秀的项目中，像 `Spring`、`Jackson` 都用到了 `Byte Buddy` 来完成底层代理。相比 `Javassist`，`Byte Buddy` 提供了更容易操作的 API，编写的代码可读性更高。更重要的是，生成的代理类执行速度比 `Javassist` 更快。

虽然以上这三种框架使用的方式相差很大，但核心原理却是差不多的，区别就只是通过什么方式生成的代理类以及在生成的代理类里面是怎么完成的方法调用。同时呢，也正是因为这些细小的差异，才导致了不同的代理框架在性能方面的表现不同。因此，我们在设计 RPC 框架的时候，还是需要进行一些比较的，具体你可以综合它们的优劣以及你的场景需求进行选择。

总结

今天我们介绍了动态代理在 RPC 里面的应用，虽然它只是一种具体实现的技术，但我觉得只有理解了方法调用是怎么被拦截的，才能厘清在 RPC 里面我们是怎么做到面向接口编程，帮助用户屏蔽 RPC 调用细节的，最终呈现给用户一个像调用本地一样去调用远程的编程体验。

既然动态代理是一种具体的技术框架，那就会涉及到选型。我们可以从这样三个角度去考虑：

因为代理类是在运行中生成的，那么代理框架生成代理类的速度、生成代理类的字节码大小等等，都会影响到其性能——生成的字节码越小，运行所占资源就越小。

还有就是我们生成的代理类，是用于接口方法请求拦截的，所以每次调用接口方法的时候，都会执行生成的代理类，这时生成的代理类的执行效率就需要很高效。

最后一个是从我们的使用角度出发的，我们肯定希望选择一个使用起来很方便的代理类框架，比如我们可以考虑：API 设计是否好理解、社区活跃度、还有就是依赖复杂度等等。

最后，我想再强调一下。动态代理在 RPC 里面，虽然看起来只是一个很小的技术点，但就是这个创新使得用户可以不用关注细节了。其实，我们在日常设计接口的时候也是一样的，我们会想尽一切办法把细节对调用方屏蔽，让调用方的接入尽可能的简单。这就好比，让你去设计一个商品发布的接口，你并不需要暴露给用户一些细节，比如，告诉他们商品数据是怎么存储的。

课后思考

请你设想一下，如果没有动态代理帮我们完成方法调用拦截，用户该怎么完成 RPC 调用？

欢迎留言和我分享你的答案，也欢迎你把文章分享给你的朋友，邀请他加入学习。我们下节课再见！

更多课程推荐

RPC 实战与核心原理

高效解决分布式系统的通信难题

何小锋

京东技术架构部首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 网络通信：RPC框架在网络通信上更倾向于哪种网络IO模型？

下一篇 06 | RPC实战：剖析gRPC源码，动手实现一个完整的RPC

精选留言 (12)

 写留言



李玥

2020-02-29

我在工作中有幸看了不少小锋老师的代码，京东很多的基础类库中，都有小峰老师的代码。从中学到了很多。包括文章中提到的对调用方尽量屏蔽实现细节的思想，以及通过定义扩展点来剥离实现等等。



👍 9



邵邵

2020-03-01

这一节RPC讲解重度依赖java知识，希望老师在后续的文章中多提炼一些跨语言层面的原理，或者用一些伪代码的方式讲解，谢谢

展开 ∨



忆水寒

2020-03-01

其实也是看需求的。如果没有动态代理，那么调用双方可以通过定义一套消息id和消息结构（才有protobuf定义），也是可以完成远程调用的。

展开 ∨



翌

2020-03-01

如果没有动态代理帮我们完成方法调用拦截，那么就需要使用静态代理来实现，就需要用户对原始类中所有的方法都重新实现一遍，并且为每个方法附加相似的代码逻辑，如果在RPC中，这种需要代理的类有很多个，就需要针对每个类都创建一个代理类。

展开 ∨



每天晒白牙

2020-02-28

如果没有动态代理类帮我们完成方法调用拦截，需要用户自己加入远程调用的逻辑，这样就麻烦，且使用不方便了

展开 ∨

作者回复: 是的，动态代理让用户API调用透明



Json Dumps

2020-03-02

看到过，GRPC，通过proto接口，生成了客户端静态代理代码。
与动态代理相比，接口变化除了更新接口文件，还要重新生成静态代理代码，并更新。
为什么非要动态代理呢？优点是什么？

展开 ∨



Jackey

2020-03-01

动态代理真是个神奇的技术，没有它的话用户就得自己写远程调用的实现了。
ps：SpringAOP也无从谈起了

展开 ∨



cricket1981

2020-03-01

没有动态代理的话就只能用静态代理了

展开 ∨



wusiration

2020-02-28

如果没有动态代理完成方法拦截，那么被调用方需要有调用方的接口实现，就失去了面向接口编程的意义



高源

2020-02-28

还有老师我有个问题请教，对于程序里已经加了必要日志来排出问题，今天发现日志没有异常情况，但是出现了现象就是服务端一直开着，客户端大概10个左右当天关闭了，第二天连接服务端时候发现就1个客户端发生底层连接上了服务端之间socket通信，但是没有正确进入业务逻辑处理，出现问题1个客户端是随机发生的。重启服务端就好了。这个问题怎么排查，客户端嵌入式Linux，服务端win双方tcp socket通讯

展开 ∨



高源

2020-02-28

请教老师，例如算法和开源框架源代码，你是如何学习深入理解的，毕竟我的想法自己能够灵活运用，个人能力有限的

展开 ∨



hello

2020-02-28

Byte Buddy，老师这个后起之秀，是怎么完成动态代理的，能否剖析下，多谢！

作者回复: 相比于其他框架，其API还是很简单的。举个例子：

```
Class<? extends T> clazz = BYTE_BUDDY.subclass(clz)
    .method(ElementMatchers.isDeclaredBy(clazz))
    .intercept(MethodDelegation.to(new ByteBuddyInvocationHandler(invoker)))
    .make()
```

```
        .load(classLoader, ClassLoadingStrategy.Default.INJECTION)
        .getLoaded();
    try {
        return clazz.newInstance();
    } catch (Exception e) {
        throw new ProxyException("Error occurred while creating bytebuddy proxy of " +
            clz.getName(), e);
    }
}
```

