33 | 动手实现一个简单的RPC框架（三）：客户端

2019-10-10 李玥

你好，我是李玥。

上节课我们已经一起实现了这个RPC框架中的两个基础组件：序列化和网络传输部分，这节课我们继续来实现这个RPC框架的客户端部分。

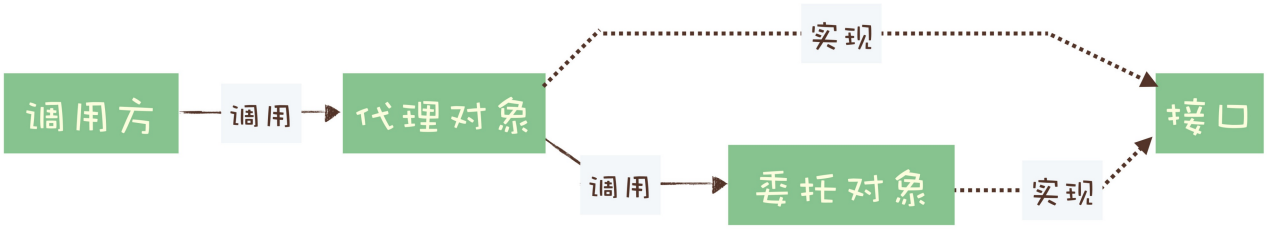
在《31 | 动手实现一个简单的RPC框架（一）：原理和程序的结构》这节课中我们提到过，在RPC框架中，最关键的就是理解“桩”的实现原理，桩是RPC框架在客户端的服务代理，它和远程服务具有相同的方法签名，或者说是实现了相同的接口，客户端在调用RPC框架提供的服务时，实际调用的就是“桩”提供的方法，在桩的实现方法中，它会发请求到服务端获取调用结果并返回给调用方。

在RPC框架的客户端中，最关键的部分，也就是如何来生成和实现这个桩。

如何来动态地生成桩？

RPC框架中的这种桩的设计，它其实采用了一种设计模式：“代理模式”。代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用，被代理的那个对象称为委托对象。

在RPC框架中，代理对象都是由RPC框架的客户端来提供的，也就是我们一直说的“桩”，委托对象就是在服务端，真正实现业务逻辑的服务类的实例。



我们最常用Spring框架，它的核心IOC（依赖注入）和AOP（面向切面）机制，就是这种代理模式的一个实现。我们在日常开发的过程中，可以利用这种代理模式，在调用流程中动态地注入一些非侵入式业务逻辑。

这里的“非侵入”指的是，在现有的调用链中，增加一些业务逻辑，而不用去修改调用链上下游的代码。比如说，我们要监控一个方法A的请求耗时，普通的方式就是在方法的开始和返回这两个地方各加一条记录时间的语句，这种方法就需要修改这个方法的代码，这是一种“侵入式”的方式。

我们还可以给这个方法所在的类创建一个代理类，在这个代理类的A方法中，先记录开始时间，然后调用委托类的A方法，再记录结束时间。把这个代理类加入到调用链中，就可以实现“非侵入式”记录耗时了。同样的方式，我们还可以用在权限验证、风险控制、调用链跟踪等等这些场景中。

下面我们来看下，在我们这个RPC框架的客户端中，怎么来实现的这个代理类，也就是“桩”。首先我们先定一个StubFactory接口，这个接口就只有一个方法：

```
public interface StubFactory {
    <T> T createStub(Transport transport, Class<T> serviceClass);
}
```

这个桩工厂接口只定义了一个方法createStub，它的功能就是创建一个桩的实例，这个桩实现的接口可以是任意类型的，也就是上面代码中的泛型T。这个方法有两个参数，第一个参数是一个Transport对象，这个Transport我们在上节课介绍过，它是用来给服务端发请求的时候使用的。第二个参数是一个Class对象，它用来告诉桩工厂：我需要你给我创建的这个桩，应该是什么类型的。createStub的返回值就是由工厂创建出来的桩。

如何来实现这个工厂方法，创建桩呢？这个桩它是一个由RPC框架生成的类，这个类它要实现给定的接口，里面的逻辑就是把方法名和参数封装成请求，发送给服务端，然后再把服务端返回的调用结果返回给调用方。这里我们已经解决了网络传输和序列化的问题，剩下一个核心问题就是如何来生成这个类了。

我们知道，普通的类它是由我们编写的源代码，通过编译器编译之后生成的。那RPC框架怎么才能根据要实现的接口来生成一个类呢？在这一块儿，不同的RPC框架的实现是不一样的，比

如，gRPC它是在编译IDL的时候就把桩生成好了，这个时候编译出来桩，它是目标语言的源代码文件。比如说，目标语言是Java，编译完成后它们会生成一些Java的源代码文件，其中以Grpc.java结尾的文件就是生成的桩的源代码。这些生成的源代码文件再经过Java编译器编译以后，就成了桩。

而Dubbo是在运行时动态生成的桩，这个实现就更加复杂了，并且它利用了很多Java语言底层的特性。但是它的原理并不复杂，Java源代码编译完成之后，生成的是一些class文件，JVM在运行的时候，读取这些Class文件来创建对应类的实例。

这个Class文件虽然非常复杂，但本质上，它里面记录的内容，就是我们编写的源代码中的内容，包括类的定义，方法定义和业务逻辑等等，并且它也是有固定的格式的。如果说，我们按照这个格式，来生成一个class文件，只要这个文件的格式是符合Java规范的，JVM就可以识别并加载它。这样就不需要经过源代码、编译这些过程，直接动态来创建一个桩。

由于动态生成class文件这部分逻辑和Java语言的特性是紧密关联的，考虑有些同学并不熟悉Java语言，所以在这个RPC的例子中，我们采用一种更通用的方式来动态生成桩。我们采用的方式是：先生成桩的源代码，然后动态地编译这个生成的源代码，然后再加载到JVM中。

为了让这部分代码不会过于复杂，便于你快速理解，我们限定：服务接口只能有一个方法，并且这个方法只能有一个参数，参数和返回值的类型都是String类型。你在学会这部分动态生成桩的原理之后，很容易重构这部分代码来解除这个限定，无非是多遍历几次方法和参数而已。

我之前讲过，我们需要动态生成的这个桩，它每个方法的逻辑都是一样的，都是把类名、方法名和方法的参数封装成请求，然后发给服务端，收到服务端响应之后再把结果作为返回值，返回给调用方。所以，我们定义一个AbstractStub的抽象类，在这个类中实现大部分通用的逻辑，让所有动态生成的桩都继承这个抽象类，这样动态生成桩的代码会更少一些。

下面我们来实现客户端最关键的这部分代码：实现这个StubFactory接口动态生成桩。

```
public class DynamicStubFactory implements StubFactory{
    private final static String STUB_SOURCE_TEMPLATE =
            "package com.github.liyue2008.rpc.client.stubs;\n" +
            "import com.github.liyue2008.rpc.serialize.SerializeSupport;\n" +
            "\n" +
            "public class %s extends AbstractStub implements %s {\n" +
            "    @Override\n" +
            "    public String %s(String arg) {\n" +
            "        return SerializeSupport.parse(\n" +
            "                invokeRemote(\n" +
            "                        new RpcRequest(\n" +
```

```
            "                            \"%s\",\n" +
            "                            \"%s\",\n" +
            "                            SerializeSupport.serialize(arg)\n" +
            "                        )\n" +
            "                )\n" +
            "        );\n" +
            "    }\n" +
            "}";

    @Override
    @SuppressWarnings("unchecked")
    public <T> T createStub(Transport transport, Class<T> serviceClass) {
        try {
            // 填充模板
            String stubSimpleName = serviceClass.getSimpleName() + "Stub";
            String classFullName = serviceClass.getName();
            String stubFullName = "com.github.liyue2008.rpc.client.stubs." + stubSimpleName;
            String methodName = serviceClass.getMethods()[0].getName();

            String source = String.format(STUB_SOURCE_TEMPLATE, stubSimpleName, classFullName, methodNa
            // 编译源代码
            JavaStringCompiler compiler = new JavaStringCompiler();
            Map<String, byte[]> results = compiler.compile(stubSimpleName + ".java", source);
            // 加载编译好的类
            Class<?> clazz = compiler.loadClass(stubFullName, results);
            // 把Transport赋值给桩
            ServiceStub stubInstance = (ServiceStub) clazz.newInstance();
            stubInstance.setTransport(transport);
            // 返回这个桩
            return (T) stubInstance;
        } catch (Throwable t) {
            throw new RuntimeException(t);
        }
    }
}
```

一起来看一下这段代码，静态变量STUB_SOURCE_TEMPLATE是桩的源代码的模板，我们需要做的就是，填充模板中变量，生成桩的源码，然后动态的编译、加载这个桩就可以了。

先来看这个模板，它唯一的这个方法中，就只有一行代码，把接口的类名、方法名和序列化后的参数封装成一个RpcRequest对象，调用父类AbstractStub中的invokeRemote方法，发送给服务端。invokeRemote方法的返回值就是序列化的调用结果，我们在模板中把这个结果反序列化之后，直接作为返回值返回给调用方就可以了。

再来看下面的createStrub方法，从serviceClass这个参数中，可以取到服务接口定义的所有信息，包括接口名、它有哪些方法、每个方法的参数和返回值类型等等。通过这些信息，我们就可以来填充模板，生成桩的源代码。

桩的类名就定义为：“接口名 + Stub”，为了避免类名冲突，我们把这些桩都统一放到固定的包com.github.liyue2008.rpc.client.stubs下面。填充好模板生成的源代码存放在source变量中，然后经过动态编译、动态加载之后，我们就可以拿到这个桩的类clazz，利用反射创建一个桩的实例stubInstance。把用于网络传输的对象transport赋值给桩，这样桩才能与服务端进行通信。到这里，我们就实现了动态创建一个桩。

使用依赖倒置原则解耦调用者和实现

在这个RPC框架的例子中，很多地方我们都采用了同样一种解耦的方法：通过定义一个接口来解耦调用方和实现。在设计上这种方法称为“依赖倒置原则（Dependence Inversion Principle）”，它的核心思想是，调用方不应依赖于具体实现，而是为实现定义一个接口，让调用方和实现都依赖于这个接口。这种方法也称为“面向接口编程”。它的好处我们之前已经反复说过了，可以解耦调用方和具体的实现，不仅实现是可替换的，实现连同定义实现的接口也是可以复用的。

比如，我们上面定义的StubFactory它是一个接口，它的实现类是DynamicStubFactory，调用方是NettyRpcAccessPoint，调用方NettyAccessPoint并不依赖实现类DynamicStubFactory，就可以调用DynamicStubFactory的createStub方法。

要解耦调用方和实现类，还需要解决一个问题：谁来创建实现类的实例？一般来说，都是谁使用谁创建，但这里面我们为了解耦调用方和实现类，调用方就不能来直接创建实现类，因为这样就无法解耦了。那能不能用一个第三方来创建这个实现类呢？也是不行的，即使用一个第三方类来创建实现，那依赖关系就变成了：调用方依赖第三方类，第三方类依赖实现类，调用方还是间接依赖实现类，还是没有解耦。

这个问题怎么来解决？没错，使用Spring的依赖注入是可以解决的。这里再给你介绍一种Java语言内置的，更轻量级的解决方案：SPI（Service Provider Interface）。在SPI中，每个接口在目录META-INF/services/下都有一个配置文件，文件名就是以这个接口的类名，文件的内容就是它的实现类的类名。还是以StubFactory接口为例，我们看一下它的配置文件：

```
$cat rpc-netty/src/main/resources/META-INF/services/com.github.liyue2008.rpc.client.StubFactory
com.github.liyue2008.rpc.client.DynamicStubFactory
```

只要把这个配置文件、接口和实现类都放到CLASSPATH中，就可以通过SPI的方式来进行加载了。加载的参数就是这个接口的class对象，返回值就是这个接口的所有实现类的实例，这样就在“不依赖实现类”的前提下，获得了一个实现类的实例。具体的实现代码在ServiceSupport这个类中。

小结

这节课我们一起实现了这个RPC框架的客户端，在客户端中，最核心的部分就是桩，也就是远程服务的代理类。在桩中，每个方法的逻辑都是一样的，就是把接口名、方法名和请求的参数封装成一个请求发给服务端，由服务端调用真正的业务类获取结果并返回给客户端的桩，桩再把结果返回给调用方。

客户端实现的难点就是，如何来动态地生成桩。像gRPC这类多语言的RPC框架，都是在编译IDL的过程中生成桩的源代码，再和业务代码，使用目标语言的编译器一起编译的。而像Dubbo这类没有编译过程的RPC框架，都是在运行时，利用一些语言动态特性，动态创建的桩。

RPC框架的这种“桩”的设计，其实是一种动态代理设计模式。这种设计模式可以在不修改源码，甚至不需要源码的情况下，在调用链中注入一些业务逻辑。这是一种非常有用的高级技巧，可以用在权限验证、风险控制、调用链跟踪等等很多场景中，希望你能掌握它的实现原理。

最后我们介绍的依赖倒置原则，可以非常有效地降低系统各部分之间的耦合度，并且不会过度增加系统的复杂度，建议你在设计软件的时候广泛的采用。其实你想一下，现在这么流行的微服务思想，其实就是依赖倒置原则的实践。只是在微服务中，它更极端地把调用方和实现分离成了不同的软件项目，实现了完全的解耦。

思考题

今天的课后作业，还是需要动手来写代码。熟悉Java语言的同学，请你扩展一下我们现在这个RPC框架客户端，解除“服务接口只能有一个方法，并且这个方法只能有一个参数，参数和返回值的类型都是String类型”这个限制，让我们的这个RPC框架真正能支持任意接口。

不熟悉Java语言的同学，你可以用你擅长的语言，把我们这节课讲解的RPC客户端实现出来，要求采用和我们这个例子一样的序列化方式，这样，你实现的客户端是可以和我们例子中的服务端正常进行通信，实现跨语言调用的。欢迎你在评论区留言，分享你的代码。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

精选留言

Gred

3

1.改用CGLib动态代理，增加多接口多方法支持。

2.增加Object序列化类以及默认序列化类，增加对多入参的序列化支持。

借花献佛了，麻烦老师指导下

https://github.com/Gred01/simple-rpc-framework

2019-10-24

作者回复

我是第一个Star这个项目的人哦！

2019-10-25

leslie

2

明确的知道自己的问题后再补开发语言那块的漏、、、课程拉了3节：看来一个人的精力还是有限，完全不拉的跟了30节后面还是拉下了课程，动手这块只能利用双休日再争取补上一点了。。。

2019-10-11

wzzJike

2

java的很多框架使用的都是jdk的动态代理吧，能获取到代理类，调用方法，参数信息

2019-10-10

作者回复

是这样的。

2019-10-10

A9

1

解除参数和返回值的限制，意味着序列化模块要支持任意类，这就需要实现一套通用的序列化协议或在serialize模块中实现用到的所有类型。假设依旧使用自定义协议，并且用到的所有类型均实现了序列化接口。需要进行如下修改：

1.DynamicStubFactory#createStub的Method处理，使用getMethods获取method数组，for循环处理method而不是直接去数组下标0

2.把DynamicStubFactory的类代码模板进行拆分，分为类模板和方法模板，先生成所有method的代码，再以此生成整个类的代码

3.定义一个新的协议结构用来存放函数的参数，用此类型代替RpcRequest的serializedArguments变量，并修改RpcRequest的序列化相关函数

4.修改RpcRequestHandler#handle:52行内根据rpcRequest去取得实际method的函数，获取任意参数个数和类型的对应函数

2019-10-10