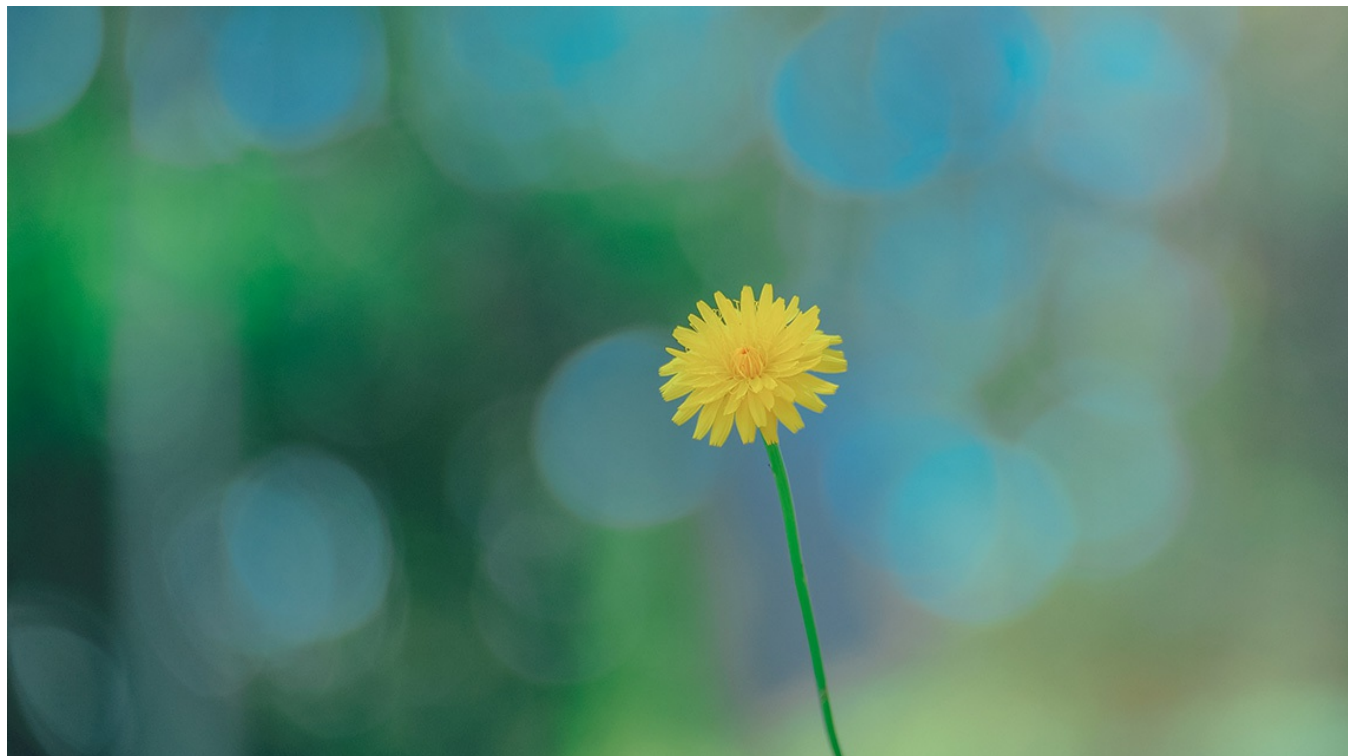


31 | 动手实现一个简单的RPC框架（一）：原理和程序的结构

2019-10-05 李玥



你好，我是李玥。

接下来的四节课，我们会一起实现一个RPC框架。你可能会问，为什么不实现一个消息队列，而要实现一个RPC框架呢？原因很简单，我们课程的目的是希望你能够学以致用举一反三，而不只是照猫画虎。在之前的课程中，我们一直在讲解消息队列的原理和实现消息队列的各种技术，那我们在实践篇如果再实现一个消息队列，不过是把之前课程中的内容重复实现一遍，意义不大。

消息队列和RPC框架是我们最常用的两种通信方式，虽然这两种中间系统的功能不一样，但是，实现这两种中间件系统的过程中，有很多相似之处，比如，它们都是分布式系统，都需要解决应用间通信的问题，都需要解决序列化的问题等等。

实现RPC框架用到的大部分底层技术，是和消息队列一样的，也都是我们在之前的课程中讲过的。所以，我们花四节课的时间来实现一个RPC框架，既可以检验你对进阶篇中学习到的底层技术掌握的是不是扎实，又可以学到RPC框架的实现原理，买一送一，很超值。

接下来的四节课，我们是这样安排的。本节课，我们先来学习RPC框架的实现原理，然后我们一起看一下如何来使用这个RPC框架，顺便给出整个项目的总体结构。第二节课中，一起来实现RPC框架的通信与序列化部分，最后的两节课，分别来实现客户端与服务端这两部分。

下面我们先来一起了解一下，RPC框架的实现原理。

首先需要明确一下RPC框架的范围。我们这里所说的RPC框架，是指类似于Dubbo、gRPC这种

框架，使用这些框架，应用程序可以“在客户端直接调用服务端方法，就像调用本地方法一样。”而一些基于**REST**的远程调用框架，虽然同样可以实现远程调用，但它对使用者并不透明，无论是服务端还是客户端，都需要和**HTTP**协议打交道，解析和封装**HTTP**请求和响应。这类框架并不能算是“**RPC**框架”。

RPC框架是怎么调用远程服务的？

所有的**RPC**框架，它们的总体结构和实现原理都是一样的。接下来，我们以最常使用的**Spring**和**Dubbo**配合的微服务体系为例，一起来看一下，**RPC**框架到底是如何实现调用远程服务的。

一般来说，我们的客户端和服务端分别是这样的：

```
@Component
public class HelloClient {

    @Reference // dubbo注解
    private HelloService helloService;

    public String hello() {
        return helloService.hello("World");
    }
}

@Service // dubbo注解
@Component
public class HelloServiceImpl implements HelloService {

    @Override
    public String hello(String name) {
        return "Hello " + name;
    }
}
```

在客户端，我们可以通过**@Reference**注解，获得一个实现了**HelloService**这个接口的对象，我们的业务代码只要调用这个方法，就可以获得结果。对于客户端代码来说，调用就是**helloService**这个本地对象，但实际上，真正的服务是在远程的服务端进程中实现的。

再来看服务端，在服务端我们的实现类**HelloServiceImpl**，实现了**HelloService**这个接口。然后，我们通过**@Service**这个注解（注意，这个**@Service**是**Dubbo**提供的注解，不是**Spring**提供的同

名注解)，在Dubbo框架中注册了这个实现类HelloServiceImpl。在服务端，我们只是提供了接口HelloService的实现，并没有任何远程调用的实现代码。

对于业务代码来说，无论是客户端还是服务端，除了增加了两个注解以外，和实现一个进程内调用没有任何区别。Dubbo看起来就像把服务端进程中的实现类“映射”到了客户端进程中一样。接下来我们一起来看一下，Dubbo这类RPC框架是如何来实现调用远程服务的。

注意，Dubbo的实现原理，或者说是RPC框架的实现原理，是各大厂面试中最容易问到的问题之一，所以，接下来的这一段非常重要。

在客户端，业务代码得到的HelloService这个接口的实例，并不是我们在服务端提供的真正的实现类HelloServiceImpl的一个实例。它实际上是由RPC框架提供的一个代理类的实例。这个代理类有一个专属的名称，叫“桩（Stub）”。

在不同的RPC框架中，这个桩的生成方式并不一样，有些是在编译阶段生成的，有些是在运行时动态生成的，这个和编程语言的语言特性是密切相关的，所以在不同的编程语言中有不同的实现，这部分很复杂，可以先不用过多关注。我们只需要知道这个桩它做了哪些事儿就可以了。

我们知道，HelloService的桩，同样要实现HelloServer接口，客户端在调用HelloService的hello方法时，实际上调用的是桩的hello方法，在这个桩的hello方法里面，它会构造一个请求，这个请求就是一段数据结构，请求中包含两个重要的信息：

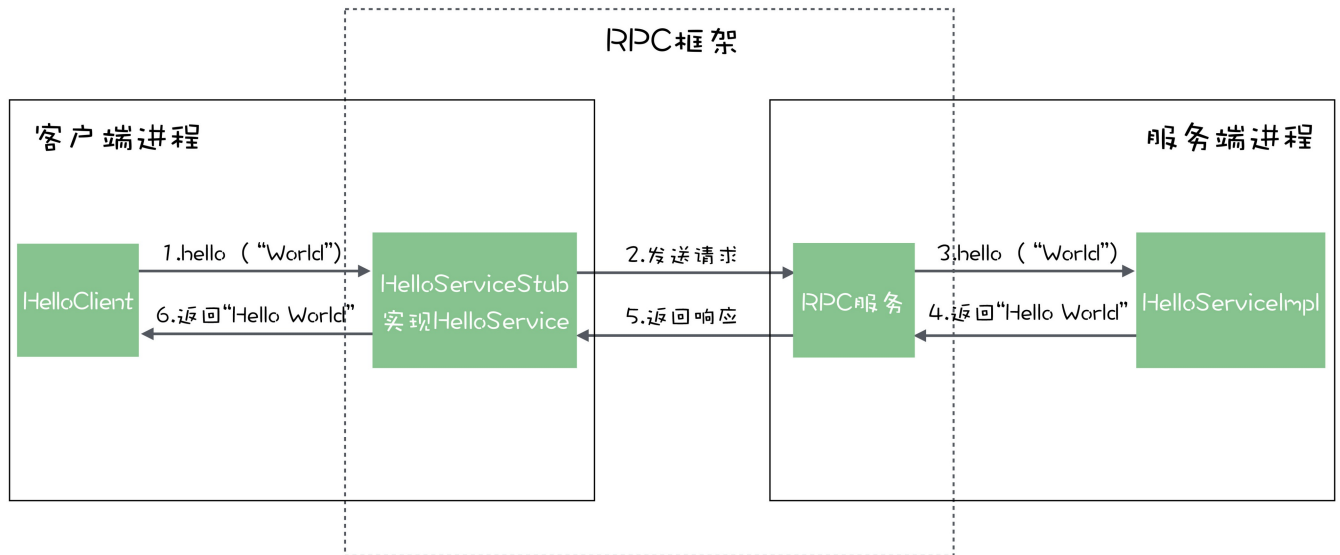
1. 请求的服务名，在我们这个例子中，就是HelloService#hello(String)，也就是说，客户端调用的是HelloService的hello方法；
2. 请求的所有参数，在我们这个例子中，就只有一个参数name，它的值是“World”。

然后，它会把这个请求发送给服务端，等待服务的响应。这个时候，请求到达了服务端，然后我们来看服务端是怎么处理这个请求的。

服务端的RPC框架收到这个请求之后，先把请求中的服务名解析出来，然后，根据这个服务名找一下，在服务端进程中，有没有这个服务名对应的服务提供者。

在这个例子的服务端中，由于我们已经通过@Service注解向RPC框架注册过HelloService的实现类，所以，RPC框架在收到请求后，可以通过请求中的服务名找到HelloService真正的实现类HelloServiceImpl。找到实现类之后，RPC框架会调用这个实现类的hello方法，使用的参数值就是客户端发送过来的参数值。服务端的RPC框架在获得返回结果之后，再将结果封装成响应，返回给客户端。

客户端RPC框架的桩收到服务端的响应之后，从响应中解析出返回值，返回给客户端的调用方。这样就完成了一次远程调用。我把这个调用过程画成一张图放在下面，你可以对着这张图再消化一下上面的流程。



在上面的这个调用流程中，我们忽略了一个问题，那就是客户端是如何找到服务端地址的呢？在RPC框架中，这部分的实现原理其实和消息队列的实现是完全一样的，都是通过一个NamingService来解决的。

在RPC框架中，这个NamingService一般称为注册中心。服务端的业务代码在向RPC框架中注册服务之后，RPC框架就会把这个服务的名称和地址发布到注册中心上。客户端的桩在调用服务端之前，会向注册中心请求服务端的地址，请求的参数就是服务名称，也就是我们上面例子中的方法签名HelloService#hello，注册中心会返回提供这个服务的地址，然后客户端再去请求服务端。

有些RPC框架，比如gRPC，是可以支持跨语言调用的。它的服务提供方和服务调用方是可以使用不同的编程语言来实现的。比如，我们可以用Python编写客户端，用Go语言来编写服务端，这两种语言开发的服务端和客户端仍然可以正常通信。这种支持跨语言调用的RPC框架的实现原理和普通的单语言的RPC框架并没有什么本质的不同。

我们可以再回顾一下上面那张调用的流程图，如果可以实现跨语言的调用，也就是说，图中的客户端进程和服务端进程是由两种不同的编程语言开发的。其实，只要客户端发出去请求能被服务端正确解析，同样，服务端返回的响应，客户端也能正确解析，其他的步骤完全不用做任何改变，不就可以实现跨语言调用了吗？

在客户端和服务端，收发请求响应的工作都是RPC框架来实现的，所以，只要RPC框架保证在不同的编程语言中，使用相同的序列化协议，就可以实现跨语言的通信。另外，为了在不同的语言中能描述相同的服务定义，也就是我们上面例子中的HelloService接口，跨语言的RPC框架还需要提供一套描述服务的语言，称为IDL（Interface description language）。所有的服务都需要用IDL定义，再由RPC框架转换为特定编程语言的接口或者抽象类。这样，就可以实现跨语言调用了。

讲到这里，**RPC**框架的基本实现原理就很清楚了，可以看到，实现一个简单的**RPC**框架并不是很难，这里面用到的绝大部分技术，包括：高性能网络传输、序列化和反序列化、服务路由的发现方法等，都是我们在学习消息队列实现原理过程中讲过的知识。

下面我就一起来实现一个“麻雀虽小但五脏俱全”的**RPC**框架。

RPC框架的总体结构是什么样的？

虽然我们这个**RPC**框架只是一个原型系统，但它仍然有近**50**个源代码文件，**2000**多行源代码。学习这样一个复杂的项目，最好的方式还是先学习它的总体结构，然后再深入到每一部分的实现细节中去，所以我们一起先来看一下这个项目的总体结构。

我们采用**Java**语言来实现这个**RPC**框架。我们把**RPC**框架对外提供的所有服务定义在一个接口**RpcAccessPoint**中：

```

/**
 * RPC框架对外提供的服务接口
 */
public interface RpcAccessPoint extends Closeable{

    /**
     * 客户端获取远程服务的引用
     * @param uri 远程服务地址
     * @param serviceClass 服务的接口类的Class
     * @param <T> 服务接口的类型
     * @return 远程服务引用
     */
    <T> T getRemoteService(URL uri, Class<T> serviceClass);

    /**
     * 服务端注册服务的实现实例
     * @param service 实现实例
     * @param serviceClass 服务的接口类的Class
     * @param <T> 服务接口的类型
     * @return 服务地址
     */
    <T> URL addServiceProvider(T service, Class<T> serviceClass);

    /**
     * 服务端启动RPC框架，监听接口，开始提供远程服务。
     * @return 服务实例，用于程序停止的时候安全关闭服务。
     */
    Closeable startServer() throws Exception;
}

```

这个接口主要的方法就只有两个，第一个方法**getRemoteService**供客户端来使用，这个方法的作用和我们上面例子中Dubbo的**@Reference**注解是一样的，客户端调用这个方法可以获得远程服务的实例。第二个方法**addServiceProvider**供服务端来使用，这个方法的作用和Dubbo的**@Service**注解是一样的，服务端通过调用这个方法来注册服务的实现。方法**startServer**和**close**（在父接口**Closeable**中定义）用于服务端启动和停止服务。

另外，我们还需要定一个注册中心的接口**NameService**:

```

/**
 * 注册中心
 */
public interface NameService {
    /**
     * 注册服务
     * @param serviceName 服务名称
     * @param uri 服务地址
     */
    void registerService(String serviceName, URI uri) throws IOException;

    /**
     * 查询服务地址
     * @param serviceName 服务名称
     * @return 服务地址
     */
    URI lookupService(String serviceName) throws IOException;
}

```

这个注册中心只有两个方法，分别是注册服务地址**registerService**和查询服务地址**lookupService**。

以上，就是我们要实现的这个**RPC**框架的全部功能了。然后，我们通过一个例子看一下这个**RPC**框架如何来使用。同样，需要先定义一个服务接口：

```

public interface HelloService {
    String hello(String name);
}

```

接口定义和本节课开始的例子是一样的。然后我们分别看一下服务端和客户端是如何使用这个**RPC**框架的。

客户端：

```
URI uri = nameService.lookupService(serviceName);
HelloService helloService = rpcAccessPoint.getRemoteService(uri, HelloService.class);
String response = helloService.hello(name);
logger.info("收到响应: {}.", response);
```

客户端首先调用注册中心**NameService**的**lookupService**方法，查询服务地址，然后调用**rpcAccessPoint**的**getRemoteService**方法，获得远程服务的本地实例，也就是我们刚刚讲的“桩”**helloService**。最后，调用**helloService**的**hello**方法，获得返回值并打印出来。

然后来看服务端，首先我们需要有一个**HelloService**的实现：

```
public class HelloServiceImpl implements HelloService {
    @Override
    public String hello(String name) {
        String ret = "Hello, " + name;
        return ret;
    }
}
```

然后，我们将这个实现注册到**RPC**框架上，并启动**RPC**服务：

```
rpcAccessPoint.startServer();
URI uri = rpcAccessPoint.addServiceProvider(helloService, HelloService.class);
nameService.registerService(serviceName, uri);
```

首先启动**RPC**框架的服务，然后调用**rpcAccessPoint.addServiceProvider**方法注册**helloService**服务，然后我们再调用**nameServer.registerService**方法，在注册中心注册服务的地址。

可以看到，我们将要实现的这个**RPC**框架的使用方式，总体上和上面使用**Dubbo**和**Spring**的例子是一样的，唯一的一点区别是，由于我们没有使用**Spring**和注解，所以需要用代码的方式实现同样的功能。

我把这个**RPC**框架的实现代码以及上面如何使用这个**RPC**框架的例子，放在了GitHub的[simple-rpc-framework](#)项目中。整个项目分为如下5个Module:

Module	说明
client	例子：客户端
server	例子：服务端
rpc-api	RPC框架接口
hello-service-api	例子：接口定义
rpc-netty	基于Netty实现的RPC框架

其中，RPC框架提供的服务RpcAccessPoint和注册中心服务NameService，这两个接口的定义在Module rpc-api中。使用框架的例子，HelloService接口定义在Module hello-service-api中，例子中的客户端和服务端分别在client和server这两个Module中。

后面的三节课，我们将一起来实现这个RPC框架，也就是Module rpc-netty。

小结

从这节课开始，我们要用四节课，利用之前学习的、实现消息队列用到的知识来实现一个RPC框架。

我们在实现RPC框架之前，需要先掌握RPC框架的实现原理。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，它会发请求的服务名和参数到服务端，服务端的RPC框架收到请求后，解析出服务名和参数后，调用在RPC框架中注册的“真正的服务提供者”，然后将结果返回给客户端。

思考题

课后你需要从GitHub上把我们即将实现的RPC框架的源代码下载到本地，先分别运行一下例子中的服务端和客户端，对整个项目有一个感性的认识。然后再分别看一下rpc-api、hello-service-api、server和client这四个Module的源代码，理清楚RPC框架的功能，以及如何使用这个RPC框架，为后续三节课的学习做好准备。欢迎在留言区与我分享讨论。

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。

消息队列高手课

从源码角度全面解析 MQ 的设计与实现

李玥

京东零售技术架构部资深架构师



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。

精选留言



姜戈

👍 3

太棒了，这是买一送一呀，这个专栏学到的绝不只是消息中间件知识~，还帮我们打通任督二脉，赞~

2019-10-12



QQ怪

👍 3

牛逼，老师厉害了

2019-10-05



海罗沃德

👍 1

感覺老師講的這個跟EJB很像，name service和JNDI很像，求教是不是這樣？

2019-10-21

作者回复

是的，实质上就是一个东西。

2019-10-22



A:春哥大魔王

👍 1

老师 桩的实现可在编译阶段和运行时阶段，在选择上除了语言本身特点外，性能上有没有差异，比如运行时阶段跑不掉使用代理反射，如果直接在编译时生成代理类，是不是可以减少这部分性能损耗

2019-10-08

作者回复

严格来说，性能肯定是有差异的。

但是在用于生产的RPC框架中，生成的桩的实例都是可以反复调用的，而不是每调用一次就重新生成一次桩，对于已经生成好的桩，调用的时候，就和动态编译还是静态编译没有关系了，所以，考虑反复多次调用的平均性能，几种方式其实是差不多的。

2019-10-09



每天晒白牙

激动，太喜欢这个专栏了

2019-10-05

👍 1



oh..

老师我想问下，这个RPC调用过程是不是所有流程都是同步的，等于从客户端开始调用服务端接口就会一直阻塞到等返回相应结果后，程序才会结束。就像直接调用HTTP请求一样，一来一回，还是会出现几次请求返回的。这个问题我有点疑惑

2019-10-22

👍 0

作者回复

是这样的，虽然我们的网络传输部分使用了异步设计，但是由于rpc调用本身是同步的，所以整个调用过程还是同步的。

2019-10-24



一步

不是做 java 的，对与 RPC 的调用方式 不是很理解

客户端去注册中心 找到对应的需要执行的方法，和通知服务端需要执行的方法，不还是要进行通讯的吗？这个通讯走的不是http了？

2019-10-12

👍 0

作者回复

最后一节课会回答你这个问题

2019-10-13



陈华应

看得困意全无！

2019-10-07

👍 0