

34 | 动手实现一个简单的RPC框架（四）：服务端

李玥 · 消息队列高手课



你好，我是李玥。

上节课我们一起学习了如何来构建这个 RPC 框架中最关键的部分，也就是：在客户端，如何根据用户注册的服务接口来动态生成桩的方法。在这里，除了和语言特性相关的一些动态编译小技巧之外，你更应该掌握的是其中动态代理这种设计思想，它的使用场景以及实现方法。

这节课我们一起来实现这个框架的最后一部分：服务端。对于我们这个 RPC 框架来说，服务端可以分为两个部分：注册中心和 RPC 服务。其中，注册中心的作用是帮助客户端来寻址，找到对应 RPC 服务的物理地址，RPC 服务用于接收客户端桩的请求，调用业务服务的方法，并返回结果。

注册中心是如何实现的？

我们先来看看注册中心是如何实现的。一般来说，一个完整的注册中心也是分为客户端和服务端两部分的，客户端给调用方提供 API，并实现与服务端的通信；服务端提供真正的业务功

能，记录每个 RPC 服务发来的注册信息，并保存到它的元数据中。当有客户端来查询服务地址的时候，它会从元数据中获取服务地址，返回给客户端。

由于注册中心并不是这个 RPC 框架的重点内容，所以在这里，我们只实现了一个单机版的注册中心，它只有客户端没有服务端，所有的客户端依靠读写同一个元数据文件来实现元数据共享。所以，我们这个注册中心只能支持单机运行，并不支持跨服务器调用。


但是，我们在这里，同样采用的是“面向接口编程”的设计模式，这样，你可以在不改动一行代码的情况下，就可以通过增加一个 SPI 插件的方式，提供一个可以跨服务器调用的真正的注册中心实现，比如说，一个基于 HTTP 协议实现的注册中心。我们再来复习一下，这种面向接口编程的设计是如何在注册中心中来应用的。

首先，我们在 RPC 服务的接入点，接口 `RpcAccessPoint` 中增加一个获取注册中心实例的方法：

 复制代码

```
1 public interface RpcAccessPoint extends Closeable{
2     /**
3      * 获取注册中心的引用
4      * @param nameServiceUri 注册中心URI
5      * @return 注册中心引用
6      */
7     NameService getNameService(URI nameServiceUri);
8
9     // ...
10 }
```

这个方法的参数就是注册中心的 URI，也就是它的地址，返回值就是访问这个注册中心的实例。然后我们再给 `NameService` 接口增加两个方法：

 复制代码

```
1 public interface NameService {
2
3     /**
4      * 所有支持的协议
5      */
6     Collection<String> supportedSchemes();
```

```
7
8    /**
9     * 连接注册中心
10     * @param nameServiceUri 注册中心地址
11     */
12    void connect(Uri nameServiceUri);
13
14    // ...
15 }
```

其中 supportedSchemes 方法，返回可以支持的所有协议，比如我们在这个例子中的实现，它的协议是“file”。connect 方法就是给定注册中心服务端的 URI，去建立与注册中心服务端的连接。

下面我们来看获取注册中心的方法 getNameService 的实现，它的实现也很简单，就是通过 SPI 机制加载所有的 NameService 的实现类，然后根据给定的 URI 中的协议，去匹配支持这个协议的实现类，然后返回这个实现的引用就可以了。由于这部分实现是通用并且不会改变的，我们直接把实现代码放在 RpcAccessPoint 这个接口中。

这样我们就实现了一个可扩展的注册中心接口，系统可以根据 URI 中的协议，动态地来选择不同的注册中心实现。增加一种注册中心的实现，也不需要修改任何代码，只要按照 SPI 的规范，把协议的实现加入到运行时 CLASSPATH 中就可以了。（这里设置 CLASSPATH 的目的，在于告诉 Java 执行环境，在哪些目录下可以找到你所要执行的 Java 程序所需要的类或者包。）

我们这个例子中注册中心的实现类是 LocalFileNameService，它的实现比较简单，就是去读写一个本地文件，实现注册服务 registerService 方法时，把服务提供者保存到本地文件中；实现查找服务 lookupService 时，就是去本地文件中读出所有的服务提供者，找到对应的服务提供者，然后返回。

这里面有一点需要注意的是，由于这个本地文件它是一个共享资源，它会被 RPC 框架所有的客户端和服务端并发读写。所以，这时你要怎么做呢？对，**必须要加锁！**

由于我们这个文件可能被多个进程读写，所以这里不能使用我们之前讲过的，编程语言提供的那些锁，原因是这些锁只能在进程内起作用，它锁不住其他进程。我们这里面必须使用由操作系统提供的文件锁。这个锁的使用和其他的锁并没有什么区别，同样是在访问共享文件之前先获取锁，访问共享资源结束后必须释放锁。具体的代码你可以去查看 `LocalFileNameService` 这个实现类。


RPC 服务是怎么实现的？

接下来，我们再来看看 RPC 服务是怎么实现的。RPC 服务也就是 RPC 框架的服务端。我们在之前讲解这个 RPC 框架的实现原理时讲到过，RPC 框架的服务端主要需要实现下面这两个功能：

1. 服务端的业务代码把服务的实现类注册到 RPC 框架中；
2. 接收客户端桩发出的请求，调用服务的实现类并返回结果。

把服务的实现类注册到 RPC 框架中，这个逻辑的实现很简单，我们只要使用一个合适的数据结构，记录下所有注册的实例就可以了，后面在处理客户端请求的时候，会用到这个数据结构来查找服务实例。

然后我们来看，RPC 框架的服务端如何来处理客户端发送的 RPC 请求。首先来看服务端中，使用 Netty 接收所有请求数据的处理类 `RequestInvocation` 的 `channelRead0` 方法。

 复制代码

```
1 @Override
2 protected void channelRead0(ChannelHandlerContext channelHandlerContext, Command
3     RequestHandler handler = requestHandlerRegistry.get(request.getHeader().getTy
4     if(null != handler) {
5         Command response = handler.handle(request);
6         if(null != response) {
7             channelHandlerContext.writeAndFlush(response).addListener((ChannelFut
8                 if (!channelFuture.isSuccess()) {
9                     logger.warn("Write response failed!", channelFuture.cause());
10                    channelHandlerContext.channel().close();
11                }
12            });
13        } else {
14            logger.warn("Response is null!");
```

```

15         }
16     } else {
17         throw new Exception(String.format("No handler for request with type: %d!",
18     }
19 }

```

这段代码的处理逻辑就是，根据请求命令的 Header 中的请求类型 type，去 requestHandlerRegistry 中查找对应的请求处理器 RequestHandler，然后调用请求处理器去处理请求，最后把结果发送给客户端。

这种通过“请求中的类型”，把请求分发到对应的处理类或者处理方法的设计，我们在 RocketMQ 和 Kafka 的源代码中都见到过，在服务端处理请求的场景中，这是一个很常用的方法。我们这里使用的也是同样的设计，不同的是，我们使用了一个命令注册机制，让这个路由分发的过程省略了大量的 if-else 或者是 switch 代码。这样做的好处是，可以很方便地扩展命令处理器，而不用修改路由分发的方法，并且代码看起来更加优雅。这个命令注册机制的实现类是 RequestHandlerRegistry，你可以自行去查看。

因为我们这个 RPC 框架中只需要处理一种类型的请求：RPC 请求，所以我们只实现了一个命令处理器：RpcRequestHandler。这部分代码是这个 RPC 框架服务端最核心的部分，你需要重点掌握。另外，为了便于你理解，在这里我只保留了核心业务逻辑，你在充分理解这部分核心业务逻辑之后，可以再去查看项目中完整的源代码，补全错误处理部分。

我们先来看它处理客户端请求，也就是这个 handle 方法的实现。

 复制代码

```


1  @Override
2  public Command handle(Command requestCommand) {
3      Header header = requestCommand.getHeader();
4      // 从payload中反序列化RpcRequest
5      RpcRequest rpcRequest = SerializeSupport.parse(requestCommand.getPayload());
6      // 查找所有已注册的服务提供方，寻找rpcRequest中需要的服务
7      Object serviceProvider = serviceProviders.get(rpcRequest.getInterfaceName());
8      // 找到服务提供者，利用Java反射机制调用服务的对应方法
9      String arg = SerializeSupport.parse(rpcRequest.getSerializedArguments());
10     Method method = serviceProvider.getClass().getMethod(rpcRequest.getMethodName()
11     String result = (String ) method.invoke(serviceProvider, arg);
12     // 把结果封装成响应命令并返回
13     return new Command(new ResponseHeader(type(), header.getVersion(), header.get

```

```
14      // ...
15  }
```

1. 把 requestCommand 的 payload 属性反序列化成为 RpcRequest;
2. 根据 rpcRequest 中的服务名, 去成员变量 serviceProviders 中查找已注册服务实现类的实例;
3. 找到服务提供者之后, 利用 Java 反射机制调用服务的对应方法;
4. 把结果封装成响应命令并返回, 在 RequestInvocation 中, 它会把这个响应命令发送给客户端。

再来看成员变量 serviceProviders, 它的定义是: Map<String/*service name*/, Object/*service provider*/> serviceProviders。它实际上就是一个 Map, Key 就是服务名, Value 就是服务提供方, 也就是服务实现类的实例。这个 Map 的数据从哪儿来的呢? 我们来看一下 RpcRequestHandler 这个类的定义:

 复制代码

```
1  @Singleton
2  public class RpcRequestHandler implements RequestHandler, ServiceProviderRegistry
3      @Override
4      public synchronized <T> void addServiceProvider(Class<? extends T> serviceClass,
5          serviceProviders.put(serviceClass.getCanonicalName(), serviceProvider);
6          logger.info("Add service: {}, provider: {}.",
7              serviceClass.getCanonicalName(),
8              serviceProvider.getClass().getCanonicalName());
9      }
10     // ...
11 }
```

可以看到, 这个类不仅实现了处理客户端请求的 RequestHandler 接口, 同时还实现了注册 RPC 服务 ServiceProviderRegistry 接口, 也就是说, RPC 框架服务端需要实现的两个功能——注册 RPC 服务和处理客户端 RPC 请求, 都是在这一个类 RpcRequestHandler 中实现的, 所以说, 这个类是这个 RPC 框架服务端最核心的部分。成员变量 serviceProviders 这个 Map 中的数据, 也就是在 addServiceProvider 这个方法的实现中添加进去的。

还有一点需要注意的是，我们 `RpcRequestHandler` 上增加了一个注解 `@Singleton`，限定这个类它是一个单例模式，这样确保在进程中任何一个地方，无论通过 `ServiceSupport` 获取 `RequestHandler` 或者 `ServiceProviderRegistry` 这两个接口的实现类，拿到的都是 `RpcRequestHandler` 这个类的唯一的一个实例。这个 `@Singleton` 的注解和获取单例的实现，在 `ServiceSupport` 中，你可以自行查看代码。顺便说一句，在 Spring 中，也提供了单例 Bean 的支持，它的实现原理也是类似的。

小结

以上就是实现这个 RPC 框架服务端的全部核心内容，照例我们来做一下总结。

首先我们一起来实现了一个注册中心，注册中心的接口设计采用了依赖倒置的设计原则（也就是“面向接口编程”的设计），并且还提供了一个“根据 URI 协议，自动加载对应实现类”的机制，使得我们可以通过扩展不同的协议，增加不同的注册中心实现。

这种“通过请求参数中的类型，来动态加载对应实现”的设计，在我们这个 RPC 框架中不止这一处用到，在“处理客户端命令并路由到对应的处理类”这部分代码中，使用的也是这样一种设计。

在 RPC 框架的服务端处理客户端请求的业务逻辑中，我们分两层做了两次请求分发：

1. 在 `RequestInvocation` 类中，根据请求命令中的请求类型（`command.getHeader().getType()`），分发到对应的请求处理器 `RequestHandler` 中；
2. `RpcRequestHandler` 类中，根据 RPC 请求中的服务名，把 RPC 请求分发到对应的服务实现类的实例中去。

这两次分发采用的设计是差不多的，但你需要注意的是，这并不是一种过度设计。原因是，我们这两次分发分别是在不同的业务抽象分层中，第一次分发是在服务端的网络传输层抽象中，它是网络传输的一部分，而第二次分发是 RPC 框架服务端的业务层，是 RPC 框架服务端的一部分。良好的分层设计，目的也是让系统各部分更加的“松耦合，高内聚”。

思考题

这节课的课后作业，我们来继续写代码。需要你实现一个 JDBC 协议的注册中心，并加入到我们的 RPC 框架中。加入后，我们的注册中心就可以使用一个支持 JDBC 协议的数据库（比如 MySQL）作为注册中心的服务端，实现跨服务器的服务注册和查询。要求：

1. 调用 `RpcAccessPoint.getNameService()` 方法，获取注册中心实例时，传入的参数就是 JDBC 的 URL，比如：“jdbc:mysql://127.0.0.1/mydb”；
2. 不能修改 RPC 框架的源代码；
3. 实现必须具有通用性，可以支持任意一种 JDBC 数据库。

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。

© 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。

精选留言 (13)



益军

2019-12-10

您好哈，提个疑问：

服务端业务方法不应该在 `channelRead0` 中执行，会导致 netty `ioEvent` 线程阻塞，应该异步提交到业务线程池执行

```
workers.execute(() -> {  
    Command response = handler.handle(request);  
    if (null != response) {  
        channelHandlerContext.writeAndFlush(response).addListener((ChannelFutureListener) channelFuture -> {  
            if (!channelFuture.isSuccess()) {  
                logger.warn("Write response failed!", channelFuture.cause());  
                channelHandlerContext.channel().close();  
            }  
        });  
    } else {
```



```
        logger.warn("Response is null!");
    }
});
Command response = handler.handle(request);
```

作者回复: 在实际生产过程中, 很多情况是需要这样处理的。避免阻塞IO线程。



👍 12



Gred

2019-10-28

来交作业啦【<https://github.com/Gred01/simple-rpc-framework/tree/namespace>】, 其实本应该在周五就写好了, 现在写的demo支持mysql和oracle, 初始化sql在rpc-netty底下init-sql.md。

作者回复: 交作业的都是好同学, 🍌!

一个建议: 使用PreparedStatement, 不要用字符串拼接SQL, 会有SQL注入漏洞。

共 2 条评论 >

👍 7



东方奇骥

2020-01-21

这里注册中心信息是保存在本地文件中, 如果保存在数据库, 就要用数据库锁, 或者zookeeper、redis的分布式锁。感谢老师, 工作三年了, 没写过框架, 这个专栏收获不小! 实战篇就超值了。

共 1 条评论 >

👍 5



亚洲舞王.尼古拉斯赵...

2019-11-11

看过了这四节rpc的课程之后, 再去看了看KafkaClient及实现的源码, 和本篇讲述思想都是相似的, 玥哥厉害! 还是思想重要, 同时觉得期末测试中的状态转换图也是中间件的基本中的基本, 重要中的重要的点, KafkaClient这个接口中很多方法都是基于状态字段来给予的返回。(为什么提起状态转换图我就想到了线程的状态转换~~☺)



👍 3



任鹏斌

2019-10-13

代码拿下来刚消化了一部分，慢慢消化，希望能做一些扩展，一转眼课程要结束了，老师辛苦！



👍 3



z.l

2019-11-01

请教下老师，如果要实现一个可供生产环境大规模集群使用的注册中心，JDBC协议是不是就不太合适了？这种情况下一个注册中心要满足哪些要求呢？个人盲猜：可多台部署、基于tcp协议最好也是netty实现、彼此之间要保证数据一致性（好像也不用强一致），不知道理解对不对

作者回复: 大致的要求就是：

可用性：不能因为某个节点故障导致注册中心不可以；

数据一致性：能保证顺序一致性一般就可以满足需求了。

性能：server节点状态变更后，能尽快更新。

另外，大规模集群情况下，需要考虑业务系统上下线时，注册中心的性能问题。比如：一个几千个节点的微服务上线，会重启几千个Server节点，这时候注册中心需要更新大量的数据。



👍 2



zero

2019-10-30

如果服务端挂掉了，怎么通知NameService呢？

作者回复: 这个问题在我们的demo中并没有解决。实际在生产环境中，有些注册中心会定期检查服务端的状态，但是即使这样也不能保证注册中心中维护的状态和服务端实际的状态实时同步。

解决这个问题，更多的是依靠rpc框架客户端的自动重试策略，比如，请求某个服务端实例超时后，立即换一个实例自动重试。



👍 2



南山

2019-10-12

抓耳挠腮了两天，还没开始动手，不知道怎么下手～

作者回复: 先把课后思考题完成了

共 2 条评论 >

👍 2



Asha

2021-07-23

老师，有个问题，不知道还能不能到达你那，我们在进阶篇中讲到了应用层协议，其实就是半包处理，为什么在rpc这里没有这部分的处理

共 1 条评论 >

👍



grey927

2020-05-12

老师，单例模式的实现，如果用Effective Java中推荐的枚举单例是否会更好一些。

💬

👍



张小勋

2019-11-08

老师 你好~ 本人是net的 最近刚下了 git 的代码 去看了下~有些写法java和net 还是有区别的 问几个问题 希望老师能够作答一下

首先: helloServiceapi 项目中 建了一个接口 这个接口在服务端去实现 客户端也用到了~ 问下这个接口 在正常的使用中 都是服务端定义好 去给客户端 去使用的么

第二点: 就是在服务端 服务端启动的时候 这个demo 中 是自己去实例化了HelloServiceImp I对象 如果在生产环境中 是不是启动的时候 通过反射 去实例化 那些特定的对外提供的服务 HelloService这样的接口 是不是也要做下标识 是这个思路么

最后 给自己立下个flag 这个月 自己会去 用netcore+netty+zookeeper 去实现一个RPC 多谢老师

作者回复: 准确的说，应该是实现定义好接口，服务端和客户端都需要使用这个接口；

💬

👍



kim118000

2019-10-15

```
Method method = serviceProvider.getClass().getMethod(rpcRequest.getMethodName(), String.class);
```

每次请求都用反射获取method，有没有性能损耗，是不是存起来，请问老师这部分生产级别怎么处理的？

作者回复: JVM对反射的优化已经很好了，这部分的性能损耗并不是很大，所以一般不用特别的优化。

共 2 条评论 >



leslie

2019-10-12

动手要欠账了：大致明白了RPC整个过程需要什么了，啃几遍梳理一下-看看各个是怎么实现的啃明白再去思考Go怎么实现。。。

共 1 条评论 >

