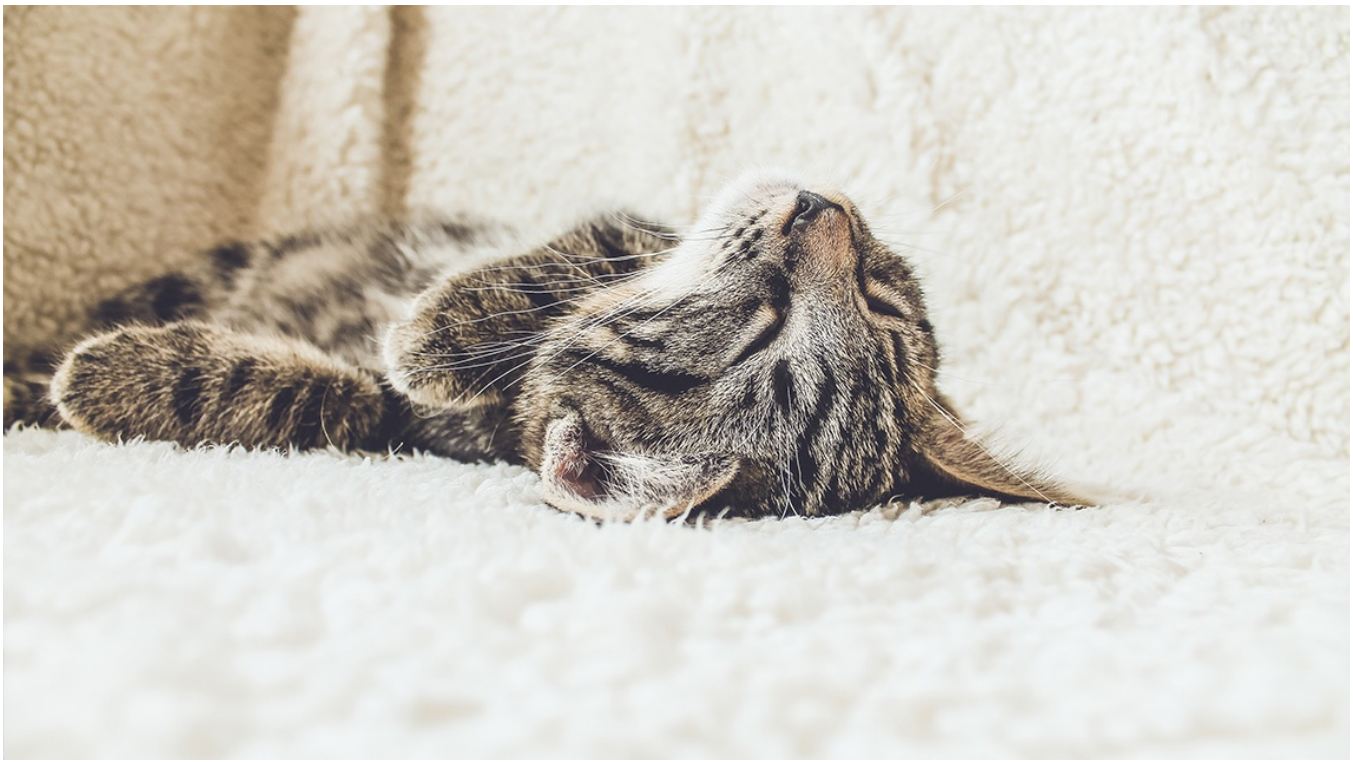
32 | 动手实现一个简单的RPC框架（二）：通信与序列化

2019-10-08 李玥



你好，我是李玥。

继续上节课的内容，这节课我们一起来实现这个RPC框架的通信和序列化部分。如何实现高性能的异步通信、如何来将结构化的数据序列化成字节流，用于网络传输或者存储到文件中，这两部分内容，我在进阶篇中都有在专门的课程分别讲解过。

网络传输和序列化这两部分的功能相对来说是非常通用并且独立的，在设计的时候，只要能做到比较好的抽象，这两部的实现，它的通用性是非常强的。不仅可以用于我们这个例子中的RPC框架中，同样可以直接拿去用于实现消息队列，或者其他需要互相通信的分布式系统中。

我们在实现这两部分的时候，会尽量以开发一个高性能的生产级系统这样的质量要求来设计和实现，但是为了避免代码过于繁杂影响你理解主干流程，我也会做适当的简化，简化的部分我会尽量给出提示。

如何设计一个通用的高性能序列化实现？

我们先来实现序列化和反序列化部分，因为后面讲到的部分会用到序列化和反序列化。

首先我们需要设计一个可扩展的，通用的序列化接口，为了方便使用，我们直接使用静态类的方式来定义这个接口（严格来说这并不是一个接口）。

```
public class SerializeSupport {
    public static  <E> E parse(byte [] buffer) {
        // ...
    }
    public static <E> byte [] serialize(E  entry) {
        // ...
    }
}
```

上面的parse方法用于反序列化，serialize方法用于序列化。如果你对Java语言不是特别的熟悉，可能会看不懂<E>是什么意思，这是Java语言泛型机制，你可以先忽略它。看一下如何来使用这个类就明白了：

```
// 序列化
MyClass myClassObject = new MyClass();
byte [] bytes = SerializeSupport.serialize(myClassObject);
// 反序列化
MyClass myClassObject1 = SerializeSupport.parse(bytes);
```

我在讲解序列化和反序列化的时候说过，可以使用通用的序列化实现，也可以自己来定义专用的序列化实现。专用的序列化性能最好，但缺点是实现起来比较复杂，你要为每一种类型的数据专门编写序列化和反序列化方法。一般的RPC框架采用的都是通用的序列化实现，比如gRPC采用的是Protobuf序列化实现，Dubbo支持hession2等好几种序列化实现。

那为什么这些RPC框架不像消息队列一样，采用性能更好的专用的序列化实现呢？这个原因很简单，消息队列它需要序列化数据的类型是固定的，只是它自己的内部通信的一些命令。但RPC框架，它需要序列化的数据是，用户调用远程方法的参数，这些参数可能是各种数据类型，所以必须使用通用的序列化实现，确保各种类型的数据都能被正确的序列化和反序列化。我们这里还是采用专用的序列化实现，主要的目的是带你一起来实践一下，如何来实现序列化和反序列化。

我们给所有序列化的实现类定义一个Serializer接口，所有的序列化实现类都实现这个接口就可以了：

```
public interface Serializer<T> {
    /**
     * 计算对象序列化后的长度，主要用于申请存放序列化数据的字节数组
     * @param entry 待序列化的对象
     * @return 对象序列化后的长度
     */
    int size(T entry);

    /**
     * 序列化对象。将给定的对象序列化成字节数组
     * @param entry 待序列化的对象
     * @param bytes 存放序列化数据的字节数组
     * @param offset 数组的偏移量，从这个位置开始写入序列化数据
     * @param length 对象序列化后的长度，也就是{@link Serializer#size(java.lang.Object)}方法的返回值。
     */
    void serialize(T entry, byte[] bytes, int offset, int length);

    /**
     * 反序列化对象
     * @param bytes 存放序列化数据的字节数组
     * @param offset 数组的偏移量，从这个位置开始写入序列化数据
     * @param length 对象序列化后的长度
     * @return 反序列化之后生成的对象
     */
    T parse(byte[] bytes, int offset, int length);

    /**
     * 用一个字节标识对象类型，每种类型的数据应该具有不同的类型值
     */
    byte type();

    /**
     * 返回序列化对象类型的Class对象。
     */
    Class<T> getSerializeClass();
}
```

这个接口中，除了serialize和parse这两个序列化和反序列化两个方法以外，还定义了下面这几个方法：size方法计算序列化之后的数据长度，用于事先来申请存放序列化数据的字节数组；type方法定义每种序列化实现的类型，这个类型值也会写入到序列化之后的数据中，主要的作用是在反序列化的时候，能够识别是什么数据类型的，以便找到对应的反序列化实现类；getSerializeClass这个方法返回这个序列化实现类对应的对象类型，目的是，在执行序列化的时候，通过被序列化的对象类型找到对应序列化实现类。

利用这个Serializer接口，我们就可以来实现SerializeSupport这个支持任何对象类型序列化的通用静态类了。首先我们定义两个Map，这两个Map中存放着所有实现Serializer接口的序列化实现类。

```
private static Map<Class<?>/*序列化对象类型*/, Serializer<?>/*序列化实现*/> serializerMap = new HashMap<
private static Map<Byte/*序列化实现类型*/, Class<?>/*序列化对象类型*/> typeMap = new HashMap<>();
```

serializerMap中的key是序列化实现类对应的序列化对象的类型，它的用途是在序列化的时候，通过被序列化的对象类型，找到对应的序列化实现类。typeMap的作用和serializerMap是类似的，它的key是序列化实现类的类型，用于在反序列化的时候，从序列化的数据中读出对象类型，然后找到对应的序列化实现类。

理解了这两个Map的作用，实现序列化和反序列化这两个方法就很容易了。这两个方法的实现思路是一样的，都是通过一个类型在这两个Map中进行查找，查找的结果就是对应的序列化实现类的实例，也就是Serializer接口的实现，然后调用对应的序列化或者反序列化方法就可以了。具体的实现在SerializeSupport中，你可以自行查看。我刚刚讲的这几个类型，听起来可能会感觉有些晕，但其实并不难，你对着代码来自己看一遍，就很容易理解了。

所有的Serializer的实现类是怎么加载到SerializeSupport的那两个Map中的呢？这里面利用了Java的一个SPI类加载机制，我会在后面的课程中专门来讲。

到这里，我们就封装好了一个通用的序列化的接口，对于使用序列化的模块来说，它只要依赖SerializeSupport这个静态类，调用它的序列化和反序列化方法就可以了，不需要依赖任何序列化实现类。对于序列化实现的提供者来说，也只需要依赖并实现Serializer这个接口就可以了。比如，我们的HelloService例子中的参数是一个String类型的数据，我们需要实现一个支持String类型的序列化实现：

```
public class StringSerializer implements Serializer<String> {
    @Override
    public int size(String entry) {
        return entry.getBytes(StandardCharsets.UTF_8).length;
    }

    @Override
    public void serialize(String entry, byte[] bytes, int offset, int length) {
        byte [] strBytes = entry.getBytes(StandardCharsets.UTF_8);
        System.arraycopy(strBytes, 0, bytes, offset, strBytes.length);
    }

    @Override
    public String parse(byte[] bytes, int offset, int length) {
        return new String(bytes, offset, length, StandardCharsets.UTF_8);
    }

    @Override
    public byte type() {
        return Types.TYPE_STRING;
    }

    @Override
    public Class<String> getSerializeClass() {
        return String.class;
    }
}
```

这里面有个初学者容易犯的错误，在把String和byte数组做转换的时候，一定要指定编码方式，确保序列化和反序列化的时候都使用一致的编码，我们这里面统一使用UTF8编码。否则，如果遇到执行序列化和反序列化的两台服务器默认编码不一样，就会出现乱码。我们在开发过程用遇到的很多中文乱码问题，绝大部分都是这个原因。

在我们这个例子中，还有一个更复杂的序列化实现MetadataSerializer，用于将注册中心的数据持久化到文件中，你也可以参考一下。

到这里序列化的部分就实现完成了。我们这个序列化的实现，对外提供服务的就只有一个

SerializeSupport静态类，并且可以通过扩展支持序列化任何类型的数据，这样一个通用的实现，不仅可以用在我们这个RPC框架的例子中，你完全可以把这部分直接拿过去用在你的业务代码中。

使用Netty来实现异步网络通信

接下来我们来说网络通信部分的实现。

同样的思路，我们把通信的部分也封装成接口。在我们这个RPC框架中，对于通信模块的需求是这样的：只需要客户端给服务端发送请求，然后服务返回响应就可以了。所以，我们的通信接口只需要提供一个发送请求方法就可以了：

```
public interface Transport {
    /**
     * 发送请求命令
     * @param request 请求命令
     * @return 返回值是一个Future，Future
     */
    CompletableFuture<Command> send(Command request);
}
```

这个send方法参数request就是需要发送的请求数据，返回值是一个CompletableFuture对象，通过这个CompletableFuture对象可以获得响应结果。这里面使用一个CompletableFuture作为返回值，使用起来就非常灵活，我们可以直接调用它的get方法来获取响应数据，这就相当于同步调用；也可以使用以then开头的一系列异步方法，指定当响应返回的时候，需要执行的操作，就等同于异步调用。等于，这样一个方法既可以同步调用，也可以异步调用。

在这个接口中，请求和响应数据都抽象成了一个Command类，我们来看一下这个类是如何定义的：

```
public class Command {
    protected Header header;
    private byte [] payload;
    //...
}

public class Header {
    private int requestId;
    private int version;
    private int type;
    // ...
}
public class ResponseHeader extends Header {
    private int code;
    private String error;
    // ...
}
```

Command类包含一个命令头Header和一个payload字节数组。payload就是命令中要传输的数据，这里我们要求这个数据已经是被序列化之后生成的字节数组。Header中包含三个属性：requestId用于唯一标识一个请求命令，在我们使用双工方式异步收发数据的时候，这个requestId可以用于请求和响应的配对儿。我们在加餐那节课实现两个大爷对话的例子中，使用的是同样的设计。

version这个属性用于标识这条命令的版本号。type用于标识这条命令的类型，这个类型主要的目的是为了能让接收命令一方来识别收到的是什么命令，以便路由到对应的处理类中去。

另外，在返回的响应Header中，我们还需要包含一个code字段和一个error字段，code字段使用一个数字表示响应状态，0代表成功，其他值分别代表各种错误，这个设计和HTTP协议中的StatueCode是一样的。

在设计通信协议时，让协议具备持续的升级能力，并且保持向下兼容是非常重要的。因为所有的软件，唯一不变的就是变化，由于需求一直变化，你不可能保证传输协议永远不变，一旦传输协议发生变化，为了确保使用这个传输协议的这些程序还能正常工作，或者是向下兼容，协议中必须提供一个版本号，标识收到的这条数据使用的是哪个版本的协议。

发送方在发送命令的时候需要带上这个命令的版本号，接收方在收到命令之后必须先检查命令的版本号，如果接收方可以支持这个版本的命令就正常处理，否则就拒绝接收这个命令，返回响应

告知对方：我不认识这个命令。这样才是一个完备的，可持续的升级的通信协议。

需要注意的是，这个版本号是命令的版本号，或者说是传输协议的版本号，它不等同于程序的版本号。我们这个例子中，并没有检查命令版本号，你在生产系统中需要自己补充这部分逻辑。

然后我们继续来看Transport这个接口的实现NettyTransport类。这个send方法的实现，本质上就是一个异步方法，在把请求数据发出去之后就返回了，并不会阻塞当前这个线程去等待响应返回来。来看一下它的实现：

```
@Override
public  CompletableFuture<Command> send(Command request) {
    // 构建返回值
    CompletableFuture<Command> completableFuture = new CompletableFuture<>();
    try {
        // 将在途请求放到inFlightRequests中
        inFlightRequests.put(new ResponseFuture(request.getHeader().getRequestId(), completableFuture));
        // 发送命令
        channel.writeAndFlush(request).addListener((ChannelFutureListener) channelFuture -> {
            // 处理发送失败的情况
            if (!channelFuture.isSuccess()) {
                completableFuture.completeExceptionally(channelFuture.cause());
                channel.close();
            }
        });
    } catch (Throwable t) {
        // 处理发送异常
        inFlightRequests.remove(request.getHeader().getRequestId());
        completableFuture.completeExceptionally(t);
    }
    return completableFuture;
}
```

这段代码实际上就干了两件事儿，第一件事儿是把请求中的requestId和返回的completableFuture一起，构建了一个ResponseFuture对象，然后把这个对象放到了inFlightRequests这个变量中。inFlightRequests中存放了所有在途的请求，也就是已经发出了请求但还没有收到响应的这些responseFuture对象。

第二件事儿就是调用netty发送数据的方法，把这个request命令发给对方。这里面需要注意的一

点是，已经发出去的请求，有可能会因为网络连接断开或者对方进程崩溃等各种异常情况，永远都收不到响应。那为了确保这些孤儿ResponseFuture不会在内存中越积越多，我们必须要捕获所有的异常情况，结束对应的ResponseFuture。所以，我们在上面代码中，两个地方都做了异常处理，分别应对发送失败和发送异常两种情况。

即使是我们对所有能捕获的异常都做了处理，也不能保证所有ResponseFuture都能正常或者异常结束，比如说，编写对端程序的程序员写的代码有问题，收到了请求就是没给我们返回响应，为了应对这种情况，还必须有一个兜底超时的机制来保证所有情况下ResponseFuture都能结束，无论什么情况，只要超过了超时时间还没有收到响应，我们就认为这个ResponseFuture失败了，结束并删除它。这部分代码在InFlightRequests这个类中。

这里面还有一个很重要的最佳实践分享给你。我们知道，如果是同步发送请求，客户端需要等待服务端返回响应，服务端处理这个请求需要花多长时间，客户端就要等多长时间。这实际上是一个天然的背压机制（Back pressure），服务端处理速度会天然地限制客户端请求的速度。

但是在异步请求中，客户端异步发送请求并不会等待服务端，缺少了这个天然的背压机制，如果服务端的处理速度跟不上客户端的请求速度，客户端的发送速度也不会因此慢下来，就会出现在途的请求越来越多，这些请求堆积在服务端的内存中，内存放不下就会一直请求失败。服务端处理不过来的时候，客户端还一直不停地发请求显然是没有意义的。为了避免这种情况，我们需要增加一个背压机制，在服务端处理不过来的时候限制一下客户端的请求速度。

这个背压机制的实现也在InFlightRequests类中，在这里面我们定义了一个信号量：

```
private final Semaphore semaphore = new Semaphore(10);
```

这个信号量有10个许可，我们每次往inFlightRequest中加入一个ResponseFuture的时候，需要先从信号量中获得一个许可，如果这时候没有许可了，就会阻塞当前这个线程，也就是发送请求的这个线程，直到有人归还了许可，才能继续发送请求。我们每结束一个在途请求，就归还一个许可，这样就可以保证在途请求的数量最多不超过10个请求，积压在服务端正在处理或者待处理的请求也不会超过10个。这样就实现了一个简单有效的背压机制。

我们在ResponseInvocation这个类中异步接收所有服务端返回的响应，处理逻辑比较简单，就是根据响应头中的requestId，去在途请求inFlightRequest中查找对应的ResponseFuture，设置返回值并结束这个ResponseFuture就可以了。

使用Netty来收发数据这部分代码，我都放在了com.github.liyue2008.rpc.transport.netty这个包中，你可以自行查看。

小结

这节课我们一起实现了序列化和异步网络通信这两部分的代码，用到的都是在进阶篇中讲过的知

识。我们在设计序列化和网络传输这两部分实现的时候，都预先定义了对外提供服务的接口。使用服务的使用方只依赖这个接口，而不依赖这个接口的任何实现。

这样做的好处是，让接口的使用者和接口的调用者充分解耦，使得我们可以安全地替换接口的实现。把接口定义的尽量通用，让接口定义与接口的使用方无关，这个接口的实现就很容易被复用，比如我们这个例子中网络传输和序列化这两部分代码，不仅可以用在这个RPC框架中，同样可以不做任何修改就用在其他的系统中。

在设计协议的时候，我们给每个命令都设计了一个固定的头，这样设计的好处是，我们在解析命令的时候可以先把头解析出来，就可以对命令进行版本检查、路由分发等通用的预处理工作，而不必把整个命令都解析出来。那为了应对变化，使协议具有持续升级的能力，命令中需要携带一个协议版本号，我们需要在收到命令后检查这个版本号，确保接收方可以支持这个版本的协议。

在实现异步网络传输的时候，一定要配套实现一个背压的机制，避免客户端请求速度过快，导致大量的请求失败。

思考题

课后给你留一个思考题：来做一个序列化的替换实现。我们这个例子中，使用了自己实现的专有的序列化实现，这些实现类都放在了com.github.liyue2008.rpc.serialize.impl这个包中，你需要换一种序列化的实现方式，来替换掉我们这个序列化实现。具体实现可以使用JSON、Protobuf或者任何一种序列化方式。

你可以删除或者改动com.github.liyue2008.rpc.serialize.impl这个包中的所有代码，但是不要修改其他代码。要求替换后，我们的这个RPC框架仍然可以正常运行。欢迎在留言区分享你的代码。

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

Tim Zhang 0

有一点不是很理解，为啥typeMap不可以是(type, serializer)，通过type获取序列化实现类一步操作，现在看源码是通过type拿到class，再class拿到serializer

2019-10-30

JackJin 0

有点不理解，Server启动时，向NameService注册接口时，序列化对象是MetadataSerializer？

2019-10-18

作者回复

是这样的。

2019-10-21

每天晒白牙 0

今日得到

关键字:背压机制(back pressure)

当客户端同步请求服务端时，客户端需要等待服务端处理完请求并返回响应，在此期间，客户端需要等待，这是一种天然背压机制

那换成异步请求呢？

客户端异步请求服务端时，不会等待服务端处理请求，这样就丧失了同步请求的那种天然背压。

如果服务器处理请求的速度慢于客户端发送请求的速度，就会导致InFlightRequest(在途请求，即还未被服务端处理或响应的请求)越来越多，如果这些在途请求保存在内存中，就可能导致内存飙升，进而引发频繁FGC，之前线上遇到过几次FGC的案例【在公众号:每天晒白牙 中分享过一个vertx-redis-client客户端的案例 https://mp.weixin.qq.com/s/fWsy26VeUvb8yPKON3OTmA，后面也遇到rpc框架中也出现类似的问题】都是因为这些在途请求保存在一个无界队列中，请求得不到处理，导致内存占用过高。

想要避免上面的情况，需要我们主动引入背压机制，即在服务端处理不过来的情况要限制客户端的请求速率，具体的做法应该有很多，我列举几个
1.把无界队列换成有界队列，队列满了就不让添加了
2.在往队列中添加请求的时候获取许可，服务端处理完请求释放许可，如果许可没了就阻塞客户端的请求或返回异常
——每天晒白牙

2019-10-10

陈华应 0

动手动手，一定要动手，netty不熟，看起来吃力，但也收获满满，每明白一点都是进步～～～

2019-10-10

作者回复

是这样的，一定要动手写代码。

2019-10-10

A9 0

使用JSON字符串进行序列化，String的序列化保持原样。MetaData和RpcRequest修改了size parse serialize函数 https://gist.github.com/WangYangA9/210ca898525832cba8ddd57ae1ae3d13

2019-10-09

作者回复

[illegible]

2019-10-10

Better me 0

ResponseInvocation 这个类是相当于一个回调类的形式吧。“就是根据响应头中的 requestId，去在途请求 inFlightRequest 中查找对应的 ResponseFuture，设置返回值并结束这个 ResponseFuture 就可以了。”这里的设置返回值是指code码形式吗？这里的作用是还需要在返回给服务端确认收到吗？老师有空看看

2019-10-09

作者回复

这里面的返回值不是状态码，而是RPC调用的返回值，比如我们这个HelloService中，这个返回值就是调用HelloServiceImpl.hello()方法的返回值。

2019-10-09

A9 0

老师的课程很多地方让人茅塞顿开，感觉从高中毕业之后就再也没有过这种抽丝剥茧学习知识

的感觉了。感谢带来这么好的课程

2019-10-08

作者回复

感谢，希望你能通过学习有收获，有提高。

2019-10-09

lecy_L

0

准点研读了一遍，受益匪浅。明天利用空闲时间尝试完成思考题。

2019-10-08