手写RPC - Protocol

Hello，大家好，欢迎来到有趣程序员的 BoredLife 手写 RPC 视频专栏，本期专栏为大家带来的是Java 与Go 双语言手写RPC框架

今天是视频专栏的第二节，在本节我们会实现 RPC 框架基础的通信协议封装以及编解码器的开发，本节代码会涉及一些 Java 网络相关的知识，以及 Netty 相关的操作，需要伙伴们有网络编程和 Netty 的基础，当然，入门的开发同学也可以跟着我一步步先实现功能，课后再去补齐网络部分知识，查缺补漏；我也会在整个课程的彩蛋节里录制网络基础和 Netty 基础的课程，敬请期待，那让我们开始。

在开始编码前，我们来纵览一下我们今天的任务，在上一节中，我们介绍了 RPC 框架的一个架构图，我们的框架计划由七部分搭建而成，而我们的开发呢，要从 RPC 框架的最基础层，通信协议 Protocol 开始

我们首先初始化一个简单的 Maven 工程

```
<properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <netty.version>4.1.6.Final</netty.version>
        <fastjson.version>1.2.29</fastjson.version>
        <jboss-marshalling-river.version>1.4.11.Final</jboss-marshalling-river.version>
        <jboss-marshalling-serial.version>1.4.11.Final</jboss-marshalling-serial.version>
        <slf4j-api.version>1.7.13</slf4j-api.version>
    </properties>

    <dependencies>

        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>${netty.version}</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j-api.version}</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
```

```
            <version>${fastjson.version}</version>
        </dependency>

        <!--序列化 接收方工具-->
        <dependency>
            <groupId>org.jboss.marshalling</groupId>
            <artifactId>jboss-marshalling-river</artifactId>
            <version>${jboss-marshalling-river.version}</version>
        </dependency>

        <!--序列化 处理工具-->
        <dependency>
            <groupId>org.jboss.marshalling</groupId>
            <artifactId>jboss-marshalling-serial</artifactId>
            <version>${jboss-marshalling-serial.version}</version>
        </dependency>

        <dependency>
            <groupId>com.jaguarliu</groupId>
            <artifactId>jaguarliu-rpc-interface</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
```

对于两个服务之间的通信协议，其实没有多么神秘，它就如同我现在知晓你的手机号之后，给你打电话一样简单；对于RPC 协议也是一样的，我们最终要实现的是类似与本地调用，就像dataservice.sayHi()一样，那么我们要知道你要请求的服务名以及你要请求的目标方法，目标方法需要携带的参数，同时我们要给当前这条消息给一个唯一标识，这里我们使用 UUID，最后我们请求结束后我们还会获得一个返回值，当然这个返回值可能是任何类型，我们定义为 Object。

定义完我们的协议内容之后，我们需要将协议中服务端与客户端交互的部分封装起来，作为消息内容，我们新建一个`RpcProtocol`，这里记笔记，在 RPC 框架中需要在网络中传输的类都需要实现`Serializable`接口，这是因为 `Serializable` 接口是Java提供的一种机制，用于标识一个类的对象是否可以被序列化。通过实现 `Serializable` 接口，我们告诉Java虚拟机，这个类的对象可以被序列化和反序列化。

```
public class RpcInvocation {


    private String targetMethod;

    private String targetServiceName;

    private Object[] args;

    private String uuid;

    private Object response;
```

```
    // .. set / get tostring
}
```

因为通信双方协议的版本有可能会有差异，我们会在消息的头部做一个魔数，是一个固定值，常量，用于标识协议的有效性和版本信息，用来保证通信双方的安全性，随后就是我要发送的消息长度和消息内容，这里可能会有小伙伴问，为什么这里的消息内容是个 byte 数组而不是 object 呢？是因为在RPC协议传输过程中，数据需要进行序列化和反序列化操作。而 `byte` 数组是一种原始的数据类型，可以直接进行序列化和反序列化操作，而无需进行额外的转换。使用 `byte` 数组可以更高效地表示数据，并且在网络传输过程中占用的空间较小。相比之下，使用 `Object` 类型可能会引入更多的开销，因为 `Object` 类型可能包含更多的信息和方法，需要更多的字节来表示。

```
public class RpcProtocol implements Serializable {

    private static final long serialVersionUID = -5842578441934800777L;

    private short magicNumber = MAGIC_NUMBER;

    private int contentLength;

    private byte[] content;

    public RpcProtocol(byte[] content){
        this.contentLength = content.length;
        this.content = content;
    }
// .. set / get tostring
}
```

Ok，我们定义完我们核心的协议组成，

首先来实现我们的编码器，我们的编码器比较简单，继承了 netty 的 `MessageToByteEncoder`，接收参数有 `ChannelHandlerContext` 对象，表示当前的上下文环境； `RpcProtocol` 对象 `msg` ，表示要编码的数据； `ByteBuf` 对象 `out` ，表示编码后的字节数据输出。首先我们要将RpcProtocol对象的魔数写入到 `out` 字节缓冲区中。魔数是一个特殊的标识，用于标识数据包的起始位置。将RpcProtocol对象的内容长度写入到 `out` 字节缓冲区中。内容长度表示实际数据的长度。将RpcProtocol对象的内容写入到 `out` 字节缓冲区中。内容是一个字节数组，表示实际的数据内容。我们依次将RpcProtocol对象的魔数、内容长度和内容数据写入到 out 缓冲区，就完成了编码的过程

```
public class RpcEncoder extends MessageToByteEncoder<RpcProtocol> {
    @Override
```

```
    protected void encode(ChannelHandlerContext ctx, RpcProtocol msg, ByteBuf out) throws Exception {
        out.writeShort(msg.getMagicNumber());
        out.writeShort(msg.getContentLength());
        out.writeBytes(msg.getContent());
    }
}
```

接下来就要来写一个消息编解码器，，这里我们的解码器要实现 netty 的`ByteToMessageDecoder`。首先，我们要知道消息是不断的在传递的，我们并不是每个字节都要解码下，我们首先要知道数据包有没有达到解码的标准，这里我们设置一个常量，还记得我们的消息体的结构么，每条消息的开头都有一个 2 字节的魔数和一个 4 字节的消息长度，如果消息没有达到这个标准，我们就不解码。同时我们也要防止大的数据包导致的内存溢出，这里我们如果可读字节数超过1000，表示收到的数据包体积过大，为了防止数据包过大导致内存溢出，直接跳过该数据包。开始接收消息，首先记录当前的读索引，并使用 `markReaderIndex()` 方法标记当前读索引的位置。读取两个字节，判断是否等于预设的魔数 `MAGIC_NUMBER` 。如果相等，表示找到了魔数，跳出循环；否则，表示收到的数据包不合法，关闭连接并返回。

```
public class RpcDecoder extends ByteToMessageDecoder {

    private final int BASE_LENGTH = 2 + 4;

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        if (in.readableBytes() > BASE_LENGTH) {
            if (in.readableBytes() > 1000){
                in.skipBytes(in.readableBytes());
            }
            int beginReader;
            while (true) {
                beginReader = in.readerIndex();
                in.markReaderIndex();
                if (in.readShort() == MAGIC_NUMBER) {
                    break;
                }else {
                    ctx.close();
                    return;
                }
            }
            int length = in.readInt();
            if (in.readableBytes() < length) {
                in.readerIndex(beginReader);
                return;
            }
            byte[] data = new byte[length];
            in.readBytes(data);

            RpcProtocol rpcProtocol = new RpcProtocol(data);
            out.add(rpcProtocol);
        }
    }
}
```

读取四个字节，表示 `RpcProtocol` 对象的 `contentLength` 字段，即实际数据的长度。判断剩余可读字节数是否小于实际数据长度，如果是，则表示数据包不完整，需要重置读索引并返回。建一个长度为 `length` 的字节数组 `data` ，并从 `byteBuf` 中读取该长度的字节数据。使用 `data` 字节数组创建一个 `RpcProtocol` 对象，并将其添加到 `out` 列表中，以便后续处理。这样我们的解码器就开发完成了。

好啦，以上就是本节的全部内容，我们开发了 RPC 框架的基础协议层，完成了协议和编解码器的开发，为后续开发打好基础～开发过程中对于代码有任何疑问或者你有更好的思路，请在评论区留言，我会在看到的第一时间答复你；如果觉得我做的还可以，那麻烦点个关注，这对我很重要，谢谢大家，我们下期再见！