34 如何支持Protobuf

更新时间：2020-08-25 09:38:00

> 人不可有傲气，但不可无傲骨。——徐悲鸿

更多一手资源请+V：Andyqclu
qq：3118617541

前言

你好，我是彤哥。

前面的章节，我们使用 Netty 完整实现了一个麻将游戏的实战项目，不过，它还只能算是一个 Demo 项目，要达到生产级，还需要做一些优化，这些优化包括：性能调优、参数调优、安全性、可扩展性、监控等多个方面。

前面，我们为了省事，使用的是 JSON 方式来序列化消息，不过，JSON 虽然容易阅读，但是性能方面确实要低不少，它的性能损耗主要体现在序列化之后的报文太大的问题。

因此，本节，我们将从性能调优的角度出发，学习如何将 JSON 替换成更高效的序列化方式 ——Protobuf。

Protobuf 简介

Protocol Buffers (简称 Protobuf)，是 Google 出品的序列化框架，与开发语言无关，和平台无关，具有良好的可扩展性。Protobuf 和所有的序列化框架一样，都可以用于数据存储、通讯协议。

Protobuf 相对于 JSON、XML、Java 序列化等其它序列化方式的优点主要体现在：

1. 序列化速度更快；
2. 序列化之后的体积更小；
3. 能够自动生成代码；
4. 支持多语言合作开发；

其中，前面两点是对性能的提升，后面两点，对于开发效率的提升也是显而易见的，通过 Protobuf 的自动生成代码框架，两端定义好协议，可以快速地进行开发，非常方便。比如，前端使用 lua，后端使用 Java，后端定义好协议，丢给前端，前端自己生成适合 lua 的协议代码，即可进入开发，无需等待后端定义接口文档等繁琐的过程。

另外，Protobuf 分成两个版本：proto2 和 proto3，如无特殊说明，本节所有内容都是基于 proto3 进行讲解的。

Protobuf 的简单使用

Netty 天然就是支持 Protobuf 的，它提供了两组编解码方式的支持：

1. 一次编解码：ProtobufVarint32LengthFieldPrepender/ProtobufVarint32FrameDecoder
2. 二次编解码：ProtobufEncoder/ProtobufDecoder

对于一次编解码，它使用的是一种变异的“长度 + 内容”法实现的，在经典的“长度 + 内容”法中，长度一旦确定是不能修改的，而 Protoubf 的两个一次编解码器，它们的长度是可变的，会根据内容的大小自动适配，具体怎么实现的呢？有兴趣的同学可以去看看源码。

对于二次编解码，编码过程比较简单，将 Java 对象转换成字节数组即可，解码过程稍微复杂一点，需要传入要解码的类型，根据这个类型去反序列化，这种实现不太友好。

好了，我们以打招呼为例来简单看一下怎么使用吧。

客户端发送 hello 消息，带上当前人的姓名，服务端返回 hello + 姓名，整个过程包含两个协议：HelloRequest 和 HelloResponse。

更多一手资源请+V：Andyqclu
qq：3118617541

使用 Protobuf，第一步就是定义前后端通信的协议，以 .proto 后缀保存：

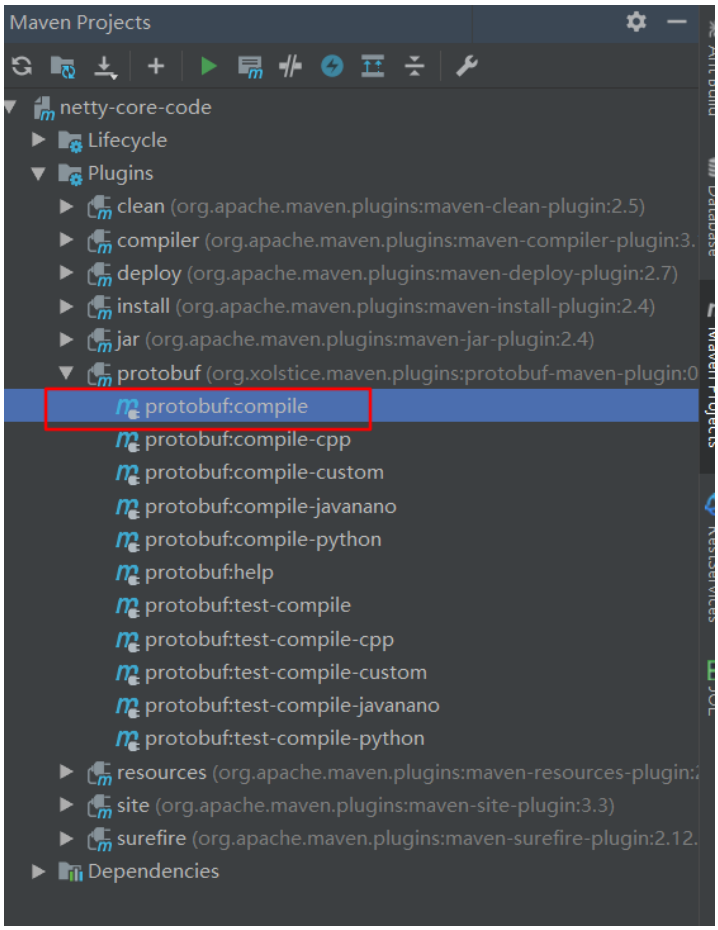
```
// 使用的版本
syntax = "proto3";
// 输出到的包名
option java_package = "com.imooc.netty.core.$34.proto";
// 是否拆分成多个文件
option java_multiple_files = true;

// HelloRequest
message HelloRequest {
    string name = 1;
}

// HelloResponse
message HelloResponse {
    bool result = 1;
    string message = 2;
}
```

定义完协议，第二步，是根据协议生成代码，此时，需要使用到官方提供的工具，或者在 Maven 中引入相关的插件来生成，我这里使用 Maven 插件的形式生成。当然，为了能够支持 Protobuf 协议还需要引入相关的依赖：

```
<dependencies>
  <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.11.4</version>
  </dependency>
</dependencies>
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.4.1.Final</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.5.1</version>
      <configuration>
        <protocArtifact>
          com.google.protobuf:protoc:3.1.0:exe:${os.detected.classifier}
        </protocArtifact>
      </configuration>
    </plugin>
  </plugins>
</build>
```

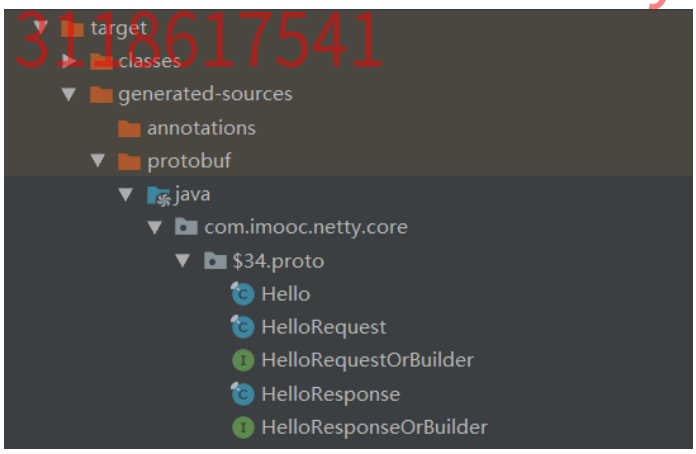
使用 Maven 插件需要把上述定义的文件放到 `src/main/proto` 目录下面，在 IDEA 中双击 probobuf compile，即可生成 Java 文件：

更多一手资源请+V：Andyqclu
qq：3118617541



更多一手资源请+V：Andyqclu
qq：3118617541

生成好的文件在 target 目录下面：



将这些文件“移动”到 `src/main/java` 下对应的目录就可以使用了。

我们先来看看如何实现服务端：

```
public class ProtobufServer {

    public static void main(String[] args) throws Exception {
        // ...省略其他代码
        serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception {
                ChannelPipeline p = ch.pipeline();
                // 一次编解码
                p.addLast(new ProtobufVarint32FrameDecoder());
                p.addLast(new ProtobufVarint32LengthFieldPrepender());
                // 二次编解码，解码器必须传入具体的类型
                p.addLast(new ProtobufDecoder(HelloRequest.getDefaultInstance()));
                p.addLast(new ProtobufEncoder());
                // 服务端处理器
                p.addLast(new DefaultEventLoopGroup(), new ProtobufServerHandler());
            }
        });

    }
}
public class ProtobufServerHandler extends SimpleChannelInboundHandler<HelloRequest> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, HelloRequest msg) throws Exception {
        // 响应
        HelloResponse helloResponse = HelloResponse.newBuilder()
                .setResult(true)
                .setMessage("hello " + msg.getName())
                .build();

        ctx.writeAndFlush(helloResponse);
    }
}
```

更多一手资源请+V：Andyqclu
qq：3118617541

请注意，ProtobufDecoder 需要传入一个 HelloRequest 的实例。

我们再来看看如何实现客户端：

```
public class ProtobufClient {

    public static void main(String[] args) throws Exception {
        // ...省略其他代码
        bootstrap.handler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                ChannelPipeline p = ch.pipeline();
                // 一次编解码
                p.addLast(new ProtobufVarint32FrameDecoder());
                p.addLast(new ProtobufVarint32LengthFieldPrepender());
                // 二次编解码，解码器必须传入具体的类型
                p.addLast(new ProtobufDecoder(HelloResponse.getDefaultInstance()));
                p.addLast(new ProtobufEncoder());
                // 客户端处理器
                p.addLast(new ProtobufClientHandler());
            }
        });
    }
}
public class ProtobufClientHandler extends SimpleChannelInboundHandler<HelloResponse> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, HelloResponse msg) throws Exception {
        if (msg.getResult()) {
            System.out.println(msg.getMessage());
        }
    }
}
```

注意，ProtobufDecoder 需要传入一个 HelloResponse 的实例。

整个实现过程非常简单，我就不演示结果了。

更多，手资源请+V：Andyqclu
qq：3118617541

但是，这里有一个坑就是 ProtobufDecoder 解码器，它必须传入一个参数，指定你要解码成哪个类型，这使得服务端和客户端只能对一种消息类型进行编解码，在我们的麻将实战项目中，使用的是 MahjongProtocol，看似满足条件，实则不满足，还记得 MahjongProtocol 的 body 吗？它是 MahjongProtocolBody 接口类型，如果在 ProtobufDecoder 中使用，它将不知道 body 具体解码成哪个类型，所以，ProtobufDecoder 编码器似乎不太适用于我们的场景，针对这种场景，我们应该怎么处理呢？

我这里提供两种思路：

1. 编写自己的 ProtobufDecoder，从 header 中取出 cmd，再根据 cmd 取出 body 的类型，再使用 protobuf 解码；
2. 把 MahjongProtocol 定义为大消息，里面包含所有的消息类型，而不是现在的 body 形式，MahjongProcessor 中参数修改为 MahjongProtocol，根据 header 中的 cmd 选择对应的 MahjongProcessor，在具体的 MahjongProcessor 中再调用 getXxx () 方法获取真正的消息内容，比如 LoginRequestProcessor 中调用 getLoginRequest ()；

其中，第一种方式对现有代码改动量较少且解析方式保持一致，第二种方式对现有代码改动量较大，故此处我们采用第一种方式对项目进行改造，使其支持 Protobuf。

项目改造

通过上面的简单案例，我们会发现，生成的消息都实现了 MessageLite 接口，而且，我们已经确定了保持现有协议基本不变，即还是分成 header 和 body 两个部分，所以，需要将 body 的类型换成 MessageLite：

```
public final class MahjongProtocol {
    /**
     * 协议头
     */
    private MahjongProtocolHeader header;
    /**
     * 协议体
     */
    private MessageLite body;
}
```

那么，下面我们将如何进行改造呢？

我认为通过下面的步骤来改造要轻松一些：

1. 把 pom.xml 中引用的 JSON 依赖去除，并添加 Protobuf 的依赖和插件；
2. 编写 Protobuf 协议文件，把之前定义的消息全部使用 Protobuf 语法全部写一遍；
3. 删除 MahjongProtocolBody 和 MahjongMessage 这两个接口；
4. 修改所有 MahjongMessage 为 MessageLite；
5. 修改服务启动编解码器；
6. 修改其他报错的地方；

OK，让我们按照这个步骤来走一遍。

去除 JSON 依赖，添加 Protobuf 依赖和插件

这一步比较简单，相关的依赖和插件在上面案例讲过了，这里就不再赘述了。

更多 手资源请+V：Andyqclu
qq：3118617541

编写 Protobuf 协议文件

我们以下面四条协议为例简单讲一下编写 proto 文件应该注意的点：

```
// 使用的版本
syntax = "proto3";
// 输出到的包名
option java_package = "com.imooc.netty.mahjong.common.proto";
// 是否拆分成多个文件
option java_multiple_files = true;

message LoginRequest {
    string username = 1;
    string password = 2;
}

message LoginResponse {
    bool result = 1;
    PlayerMsg player = 2;
    string message = 3;
}

message OperationRequest {
    int32 operation = 1;
    int32 pos = 2;
    int32 card = 3;
}
message PlayerMsg {
    int64 id = 1;
    string username = 2;
    string password = 3;
    int32 score = 4;
    int32 pos = 5;
    bytes cards = 6;
    bytes chuCards = 7;
    bytes pengList = 8;
    bytes gangList = 9;
}
```

更多一手资源请+V：Andyqclu
qq：3118617541

1. 在第一行要定义协议的语法（版本信息），默认为 proto2，使用 proto3 需要显式地指定；
2. 协议文件无法引用外部类，所以需要把房间信息和玩家信息也定义成相应的协议，不过，多个 proto 文件之间是可以相互引用的；
3. 协议文件无法支持 `byte` 类型，不足 4 个字节的数值类型统一使用 int32，还有 uint32、sint32、fixed32、sfixed32 也可以使用，略微有些差别，虽然不支持字节类型，但是在编码的时候会根据实际占用的字节数进行压缩，所以，不怕；
4. 字节数组可以使用 `bytes` 类型，对应到 Java 的 `ByteString` 类，可以支持遍历、按索引取值等操作，不支持修改；
5. 协议文件不支持数组，如果表示多个，可以在前面加上 `repeated`，生成的类中使用的是 `List<T>` 表示的；
6. 生成的类不支持修改，一般只作查询使用，如果要修改会很麻烦，需要重新 build，所以，我们依然保留原来的 Room 和 Player 领域对象，只有发送消息的时候才将他们转换成 RoomMsg 和 PlayerMsg。

OK，编写完协议文件，双击 maven 插件，即可生成相应的类，把这些类移动到 `com.imooc.netty.mahjong.common.proto` 包下面，注意是移动，不是复制，复制后 target 目录下面还有一份源文件，启动项目的时候会报重复的类错误。

删除 MahjongProtocolBody 和 MahjongMessage 接口

选中这两个接口，按下 Delete 即可，不要犹豫。

修改所有 MahjongMessage 为 MessageLite

这个主要是在各个接口中：MessageManager、MahjongProcessorManager、MahjongRenderManager 等，还有 MahjongEventExecutorGroup 中，修改起来还算比较容易。

修改服务启动编解码器

根据前面的描述，对于一次编解码，我们完全可以换成 ProtobufVarint32LengthFieldPrepender 和 ProtobufVarint32FrameDecoder，所以，直接把原来的一次编解码删除。

对于二次编解码，总体逻辑跟原来是一致的，其实编解码类本身并不需要修改，主要逻辑都在 MahjongProtocol 类中，可以参考 ProtobufEncoder 和 ProtobufDecoder 的写法，修改为如下：

```
public final class MahjongProtocol {
    /**
     * 协议头
     */
    private MahjongProtocolHeader header;
    /**
     * 协议体
     */
    private MessageLite body;

    public void decode(ByteBuf msg) throws InvalidProtocolBufferException {
        MahjongProtocolHeader header = new MahjongProtocolHeader();
        // 解码header
        header.decode(msg);
        this.header = header;

        // 命令字
        int cmd = header.getCmd();
        // 根据命令字获取body的真实类型
        MessageLite msgType = getBodyTypeByCmd(cmd);

        // 解码body
        final byte[] array;
        final int offset;
        final int length = msg.readableBytes();
        if (msg.hasArray()) {
            array = msg.array();
            offset = msg.arrayOffset() + msg.readerIndex();
        } else {
            array = ByteBufUtil.getBytes(msg, msg.readerIndex(), length, false);
            offset = 0;
        }

        this.body = msgType.getParserForType().parseFrom(array, offset, length);
    }

    private MessageLite getBodyTypeByCmd(int cmd) {
        return MessageManager.getMsgTypeByCmd(cmd);
    }

    public void encode(ByteBuf buffer) {
        header.encode(buffer);
        buffer.writeBytes(body.toByteArray());
    }
}
```

更多一手资源请+V：Andyqclu
qq：3118617541

可以看到，编码非常简单，解码的过程相对复杂一些，我们根据 cmd 从 MessageManager 中拿到实际的消息类型，再通过一系列操作解析出 body。

当然，不要忘记修改 MahjongClient 和 MahjongServer 中的 Pipeline：

```
// 一次编解码器
p.addLast(new ProtobufVarint32FrameDecoder());
p.addLast(new ProtobufVarint32LengthFieldPrepender());
// 二次编解码器
p.addLast(new MahjongProtocolDecoder());
p.addLast(new MahjongProtocolEncoder());
// 处理器
p.addLast(new MahjongServerHandler());
```

修改其他报错的地方

这个步骤特别繁琐，主要都是消息构造方式的改变带来的锅，以前构造消息是这样的：

```
LoginResponse response = new LoginResponse();
response.setResult(false);
response.setMessage("username error");
MessageUtils.sendResponse(response);
```

现在构造消息变成了这样：

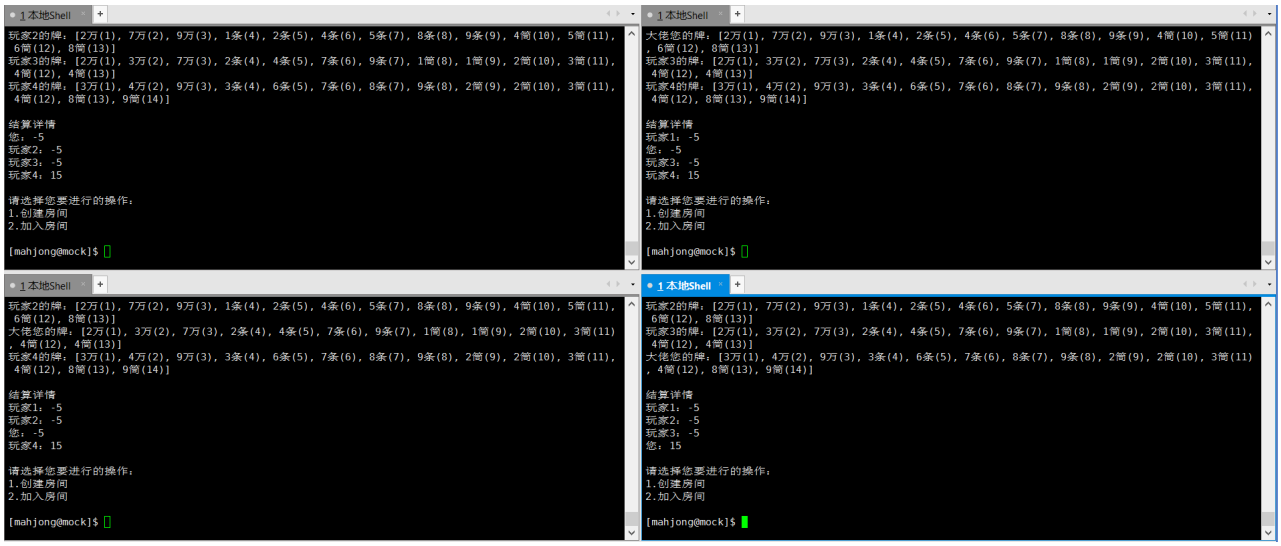
```
LoginResponse response = LoginResponse
        .newBuilder()
        .setResult(false)
        .setMessage("username error")
        .build();
MessageUtils.sendResponse(response);
```

要把所有发送消息的地方都修改一遍，是个体力活。

更多一手资源请+V：Andyqclu
qq：3118617541

这个步骤修改完成后，基本上就差不多了，还有一些报错的地方再针对性的修改即可。

OK，当我们全部修改完成后，启动四个客户端来打一局试试：



注意观察出牌结果、格式化、牌局刷新等情况是否都跟之前保持一致，可以发现，除了直接打印的消息换行有点奇怪，其他的都是没问题的。关于直接打印消息的换行问题，在 Protobuf 中是使用 `\n` 表示换行，在 XSHELL 中使用 `\r\n` 表示换行，所以，这个问题可以忽略。

OK，到此，Protobuf 改造完毕。

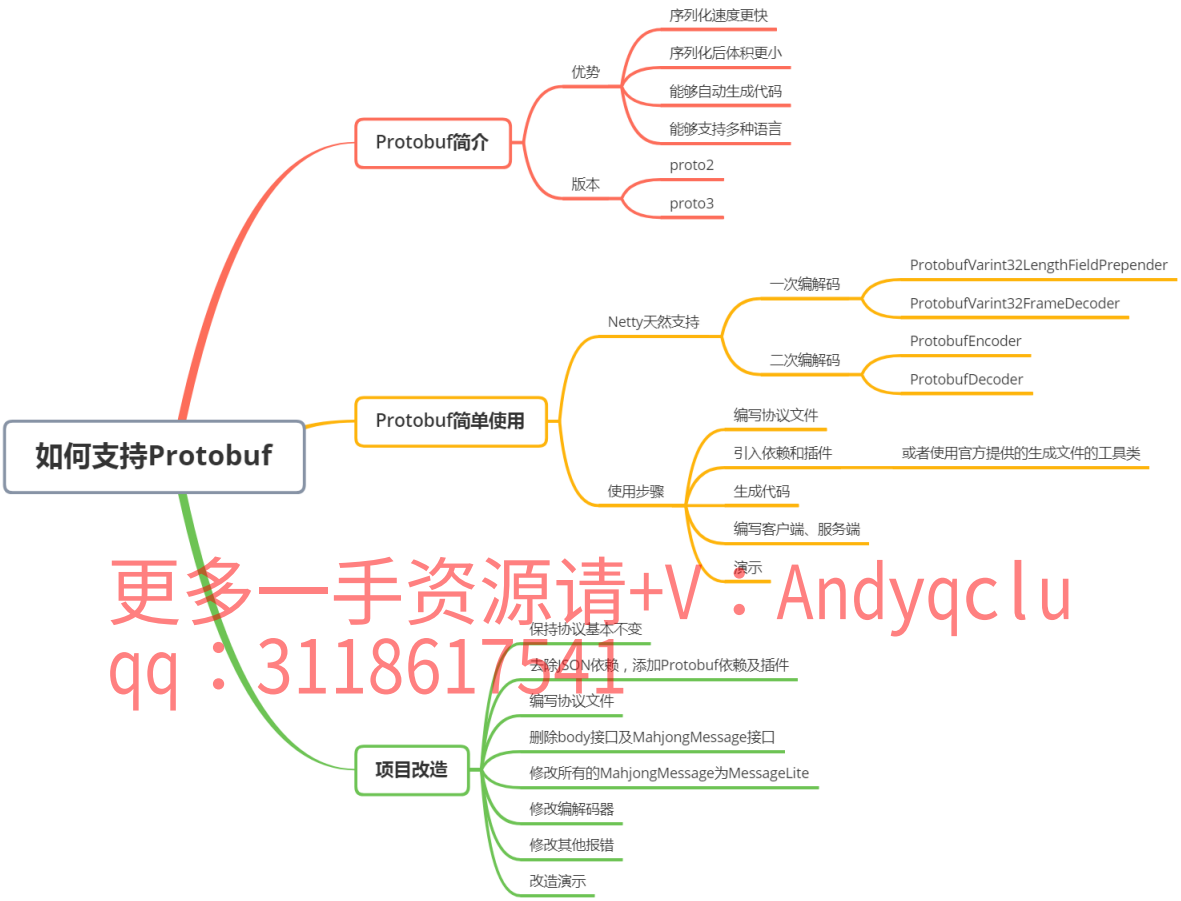
后记

本节，我们一起将 JSON 序列化更改为了 Protobuf 的形式，可以看到，如果前期没有规划好，到后期再去动架构层面的东西是非常痛苦的。

如果老板说，我们现在要做微信小游戏了，此时应该怎么办？

下一节，我们将从可扩展性的角度来分析，如何让服务端支持 WebSocket 协议，敬请期待。

思维导图



更多一手资源请+V：Andyqclu
qq：3118617541

}

33 Mock客户端实现，四个人来一局

35 如何支持WebSocket