# X-IM开发文档

## 项目说明

该项目使用的技术栈主要有SpringBoot2.0.x，Netty4，WebSocket。注意改项目纯粹是自己练习使用。

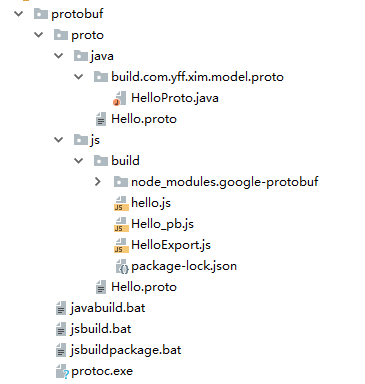
## 项目搭建

为了提高传输效率，前后将才会用google的protobuf。所以项目的第一步先要解决前后端怎么快速正确的生产protobuf格式的model。关于javascript怎么使用protobuf请参考博客<https://blog.csdn.net/arvin_kai/article/details/77532595>。

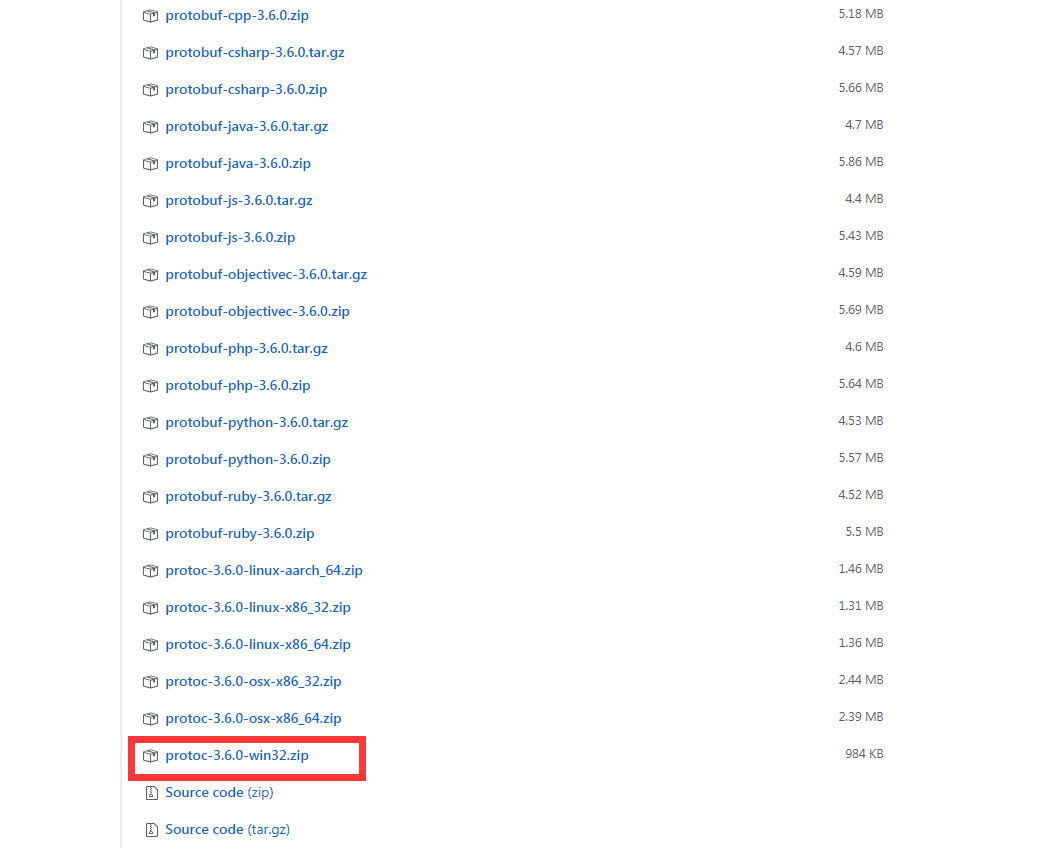
下面是我的实验过程：

### protoBuf在windows的使用

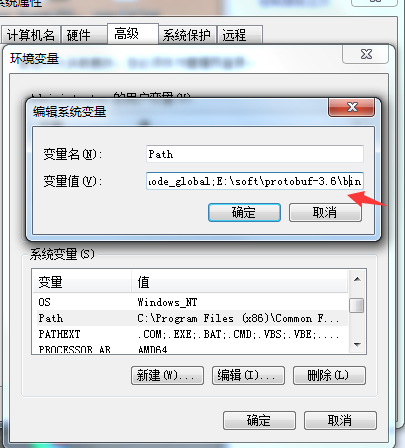
首先在在resource目录下创建protobuf等目录，如下图所示：



Java目录放置的是protoBuf需要编译的java输入和输出文件，js目录放置的是protoBuf需要编译的javascript输入输出文件。那么编译了出我们能够使用的文件了？首先去github上面下载一个protoBuf.exe文件，地址是：<https://github.com/google/protobuf/releases。然后找到xxx_win32.zip>压缩目录即可：



解压该目录，然后将该目录下的bin目录设置到环境变量里面去。比如解压到我的机器目录为E:\soft\protobuf-3.6，那么在环境变量path加上：



这样就能够使用protoc命令了。

如果上面都完成，那么接下来按照protobuf的规范新建一个Hello.proto文件，如下：

syntax = "proto3";

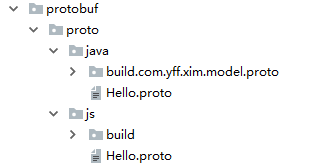
package mypackage;

message myMessage {

int32 my\_value = 1;

}

将Hello.proto分别放到之前新建好的java和js目录。如下图所示：



接下来将一步一步编译出想要的protoBuf在java和js环境都能够使用的文件。

#### Java环境编译

使用 protoc –I=xxx –java\_out=xxx xxx.proto 命令即可编译。下面给出编译Hello.proto的bat。

protoc -I=%~dp0\proto\java --java\_out=%~dp0\proto\java\build %~dp0\proto\java\Hello.proto

pause

注意上面的bat脚本中%~dp0代表了当前目录，执行该脚本后，将在java/build目录下产生如下文件：



这就是我们需要的文件。显然上面的bat脚本还不够灵活，因为每一次只能够编译一个xxx.proto文件，能够不能一次编译多个.proto文件了？显然我们只要变量java目录下所有以.proto的文件即可：

@echo off  
::设置当前目录  
set input\_path=%~dp0proto\java  
::遍历当前目录下的所有以.proto结尾的文件  
for /R %input\_path% %%a in (\*.proto) do (  
 protoc -I=%input\_path% --java\_out=%input\_path%\build %%a  
)

#### Javascript环境编译

怎么在js中使用protoBuf？请先参考<https://blog.csdn.net/arvin_kai/article/details/77532595>。注意请先自行安装npm，然后按照文章即可搭建在js编译.proto文件了。第一步执行jsbuild.bat脚本：

@echo off  
set input\_path=%~dp0proto\js  
for /R %input\_path% %%a in (\*.proto) do (  
 protoc -I=%input\_path% --js\_out=import\_style=commonjs,binary:%input\_path%\build %%a  
)

在js/build目录将生产Hello\_pb.js文件，接下来编写一个HelloExport.js文件，最后执行jsbuildpackage.bat打包脚本即可（在执行打包脚本之前一定要先按照给出的文章安装必要的环境）。接下来需要编写一个HelloExport.js将Hello\_pb.js以模块的方式打包出去。HelloExport.js如下：

var helloObj = require('./Hello\_pb');

module.exports = {

DataProto: helloObj

}

最后执行jsbuildpackage.bat脚本：

@echo off  
set input\_path=%~dp0proto\js\build  
browserify %input\_path%/HelloExport.js > %input\_path%/hello.js

注意该脚本只适用于当个文件的package，我也尝试过使用遍历的方式，当是不知道bat怎么重命名。请读者自行研究。

### Index页面的开发

我们将采用thymeleaf做模板。编写一个Index.html页面如下图：



然后再写一个IndexController：

*/\*\*  
 \* 首页web接入层  
 \*/*@Controller  
**public class** IndexController {  
  
 */\*\*  
 \* 用户访问/index / "" 进入index.html页面  
 \** ***@return*** *\*/* @GetMapping(value = {**""**,**"/"**,**"/index"**})  
 **public** String index() {  
 **return "index"**;  
 }  
}

注意由于使用了静态资源，为了防止SpringMVC拦截静态资源，需要重写MVC的配置。

@Configuration  
**public class** WebMvcConfig **extends** WebMvcConfigurationSupport {  
  
 */\*\*  
 \* 解决静态资源拦截的问题  
 \** ***@param registry*** *\*/* @Override  
 **protected void** addResourceHandlers(ResourceHandlerRegistry registry) {  
 registry.addResourceHandler(**"/static/\*\*"**).addResourceLocations(**"classpath:/static/"**);  
 }  
}

### 单聊页面-前端

在Index页面，我们首先来完成单聊这一个功能模块。这里的需求是，当用户点击单聊按钮，进入chat.html页面。当chat.html第一次渲染的之后，前台会自动连接后台的IMWebsocketServer。当chat.html调用js的createWebSocket方法后会主动链接后端websocket服务。通过websocekt的规范可知，链接成功后回调onopen方法：

*//连接后****socket***.onopen = **function** (event) {  
 *//如果链接成功，那么发送心跳给服务端* **var** message = **new *proto***.Model();  
 **var** browser = ***BrowserUtil***.info();  
 *//版本* message.setVersion(**"1.0"**);  
 *//设备id* message.setDeviceid(**""**);  
 *//请求接口命令字 1绑定 2心跳 3上线 4下线 5消息 6重连* message.setCmd(1);  
 *//发送人* message.setSender(***currentsession***);  
 *//请求1，应答2，通知3，响应4* message.setMsgtype(1);  
 *//1 rsa加密 2aes加密* message.setFlag(1);  
 *//mobile-ios mobile-android pc-windows pc-mac* message.setPlatform(browser.**name**);  
 *//客户端版本* message.setPlatformversion(browser.**version**);  
 *//客户端凭证* message.setToken(***currentsession***);  
 **var** bytes = message.serializeBinary();  
 ***socket***.send(bytes);  
};

此时我们发送一个消息给后台服务器，请求建立连接。注意这里我们没有关注websocekt的握手等过程，当然也可以写一个Handler去处理这写过程。目前已经单独写了一个案例，后期将整合进来。而这里请求链接的意思后端将模拟一条链接去描述这个链接，我们称它为Session，也就是会话。本次会话当然会关联一条物理链接（Channel，或者socket）。

前端发送了两个动作，一个是websocket的握手请求，一个是发送了一个msg（请求建立链接消息）。那么后端怎么处理了？在ImWebsocketServer类中我们详细说明一下几个重要的handler：

### 单聊页面-后端

*// HTTP请求的解码和编码*pipeline.addLast(**new** HttpServerCodec());  
*// 把多个消息转换为一个单一的FullHttpRequest或是FullHttpResponse，  
// 原因是HTTP解码器会在每个HTTP消息中生成多个消息对象HttpRequest/HttpResponse,HttpContent,LastHttpContent*pipeline.addLast(**new** HttpObjectAggregator(GlobalConstant.ImWebsocketServerConfig.***MAX\_AGGREGATED\_CONTENT\_LENGTH***));  
*// 主要用于处理大数据流，比如一个1G大小的文件如果你直接传输肯定会撑暴jvm内存的; 增加之后就不用考虑这个问题了*pipeline.addLast(**new** ChunkedWriteHandler());  
*// WebSocket数据压缩*pipeline.addLast(**new** WebSocketServerCompressionHandler());  
*// 协议包长度限制*pipeline.addLast(**new** WebSocketServerProtocolHandler(**"/ws"**, **null**, **true**,  
 GlobalConstant.ImWebsocketServerConfig.***MAX\_FRAME\_LENGTH***));  
*// 协议包解码*pipeline.addLast(**new** WebSocketDecoder());  
*// 协议包编码*pipeline.addLast(**new** WebsocketEncoder());  
*// 协议包解码时指定ProtoBuf字节数实例化为CommonProtocol类型*pipeline.addLast(**messageProtoBufDecoder**);  
*//心跳处理器*pipeline.addLast(**new** IdleStateHandler(GlobalConstant.ImWebsocketServerConfig.***READ\_IDLE\_TIME***,  
 GlobalConstant.ImWebsocketServerConfig.***WRITE\_IDLE\_TIME***,0));  
*// 业务处理器*pipeline.addLast(**webSocketServerHandler**);

这了包括了处理http请求的处理，也包括了websocket请求的处理器。那么为什么有上面这些处理器？这就要了解websocekt的通信过程。（1）客户端向服务端发送http报文格式的请求，而且是GET方法的请求，不过这里与普通的http请求有稍微不同的地方，那就是头部connection字段是Upgrade，然后又Upgrad字段，值是websocket，其请求格式如下：

1. \* GET /chat HTTP/1.1
2. \* Host: server.example.com
3. \* Upgrade: websocket
4. \* Connection: Upgrade
5. \* Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6. \* Sec-WebSocket-Origin: http:*//example.com*
7. \* Sec-WebSocket-Protocol: chat, superchat
8. \* Sec-WebSocket-Version: 13

（2）由服务器端向客户端返回特定格式的http报文，表示当前websocket建立，报文格式如下：

1. \* HTTP/1.1 101 Switching Protocols
2. \* Upgrade: websocket
3. \* Connection: Upgrade
4. \* Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5. \* Sec-WebSocket-Protocol: chat

（3）当连接建立以后，那么双方就可以通过刚刚建立连接用的socket来发送数据了，这里发送的数据必须经过websocket的帧格式的编码，具体它的信息就去看websocket的协议标准就知道了。

其实知道了整个建立连接以及通信的过程，那么就能够知道如何在channel的pipeline上面安排各种handler的顺序了，如下：

（1）首先要安排http报文的decode，aggregator以及encode的handler，因为最开始还是采用http通信的

（2）接收到websocket的建立连接报文之后，通过一些处理按照规定的格式将http返回发送给客户端，那么这就表示websocket的连接已经建立了

（3）移除前面设置的所有的http报文处理的handler，然后加上websocket的frame的decode以及encodehandler，用于客户端和服务通信。

所以前面这些Handler是处理http请求

*// HTTP请求的解码和编码*pipeline.addLast(**new** HttpServerCodec());  
*// 把多个消息转换为一个单一的FullHttpRequest或是FullHttpResponse，  
// 原因是HTTP解码器会在每个HTTP消息中生成多个消息对象HttpRequest/HttpResponse,HttpContent,LastHttpContent*pipeline.addLast(**new** HttpObjectAggregator(GlobalConstant.ImWebsocketServerConfig.***MAX\_AGGREGATED\_CONTENT\_LENGTH***));  
*// 主要用于处理大数据流，比如一个1G大小的文件如果你直接传输肯定会撑暴jvm内存的; 增加之后就不用考虑这个问题了*pipeline.addLast(**new** ChunkedWriteHandler());

然后接下来是一个比较关键的处理器WebSocketServerProtocolHandler，他的作用就是去除之前处理http请求的Handler，并处理websocket的握手过程。经过WebSocketServerProtocolHandler处理成功后表示websocket通信连接已经建立。注意WebSocketServerProtocolHandler处理成功后的通信格式类型将采用所有规定的WebSocket帧类型。

因为我们这里采用了ProtoBuf这种通信格式，所以显然需要处理ProtoBuf和WebSocketFrame格式之间的编解码。

*// 协议包解码*pipeline.addLast(**new** WebSocketDecoder());  
*// 协议包编码*pipeline.addLast(**new** WebSocketEncoder());  
*// 协议包解码时指定ProtoBuf字节数实例化为CommonProtocol类型*pipeline.addLast(**messageProtoBufDecoder**);

其中WebSocketDecoder将WebsocketFrame格式的数据转换为byte格式数据；然后messageProtoBufDecoder将byte字节格式的数据转为为ProtoBuf格式的模型对象数据，这里指的是MessageProto.Model格式。WebSocketEncoder将byte格式的数据转换为WebsocketFrame格式的数据，

接下来看最后一个Handler，也就是ImWebSocketServerHandler。这个是Handler是我们自己定义的Handler，专门用来处理MessageProto.Model格式的数据。ImWebSocketServerHandler首先我们主要关注userEventTriggered和channelRead0两个方法。

在前面分析前端页面的时候说过，当websocket链接建立完成之后，前端会发送一个消息请求后端绑定，所以在channelRead0方法中我们首先来处理这一种消息的业务。

*//收到消息*String sessionId = **connectorManager**.getChannelSessionId(ctx);  
**if**(msg.getMsgtype() == GlobalConstant.MessageType.***SEND***) {  
 *//更新最后一次客户端发送的心跳时间* ctx.channel().attr(GlobalConstant.SessionConfig.***SERVER\_SESSION\_HEARBEAT***).set(System.*currentTimeMillis*());  
 *//消息处理，注意这里接受到客户端的cmdType类型可能是1绑定 2心跳 3上线 4下线 5消息 6重连* MessageVo messageVo = **messageProcessor**.messageConvert(sessionId,msg);  
 *//设置消息来源* messageVo.setSource(GlobalConstant.ImWebsocketServerConfig.***WEBSOCKET***);  
 *//发送消息给客户端* **if**(messageVo != **null**) {  
 **connectorManager**.processMessage(ctx, messageVo);  
 }  
}

具体的业务逻辑注释已经写的很清楚了，这里我们再一次梳理一下，将整个消息处理的数据流程走一遍，这样有利于读者加深对服务端的核心部分的理解，读者能够按照自己的业务快速开发。

##### 消息分发接口-IMessageProcessor

当客户端发送一个消息过来之后，IMessageProcessor将会拦截住该消息。由于每一个消息的格式我们是知道的，那么根据cmdType将原始消息转为特定格式的消息，这个消息是连接IMessageProcessor和IConnectorManager的载体。为什么会有一个IMessageProcessor接口？原因是为了如果消息cmdType不能够满足当前的业务需求下，通过该接口能够扩展，后续的开发能够看出来这一个特性。

##### 会话管理接口-IConnectorManager，ISessionService

这两个个接口的作用是通过消息驱动Session的会话状态。其核心方法是processMessage，该方法将根据不同的消息cmdType驱动session的状态。比如当前的消息是连接请求，那么：

*//处理客户端和服务连接***if** (messageVo.isConnect()) {  
 *//客户端sessionId* String sessionId = messageVo.getSessionId();  
 *//Channel中的sessionId* String sessionId0 = getChannelSessionId(ctx);  
  
 **if** (StringUtils.*isNotEmpty*(sessionId0) && sessionId0.equals(sessionId)) {  
 *//客户端的同一个连接，更新session即可* Session session = **sessionService**.selectSession(sessionId);  
 session.setUpdateDate(**new** Date());  
 **sessionService**.saveSession(session);  
 ***log***.info(**"connector reconnect sessionId -> "** + sessionId + **", ctx -> "** + ctx.toString());  
 } **else** {  
 *//创建一个新的session,并保存该连接* ***log***.info(**"connector connect sessionId -> "** + sessionId + **", sessionId0 -> "** + sessionId0 + **", ctx -> "** + ctx.toString());  
 Session session = **sessionService**.createSession(ctx, messageVo);  
 setChannelSessionId(ctx,session.getSessionId());  
 }  
}

具体的业务逻辑注释已经说明了，ISessionService将负责创建，销毁，更新session。在ISessionService抽离处理Session的存储ISessionRepositroy，目的是为了后续引入其他的存储方式，因为这连接和并发量比较大的情况，需要调整Session的存储方式。这里为了快速开发采用了内存存储的形式。

#### 心跳处理

在连接成功后，后端会在一定的时间间隔检查心跳。首先服务端后发送一个ping消息给客户端，客户端在onmessage方法里面接受到消息后发送一个pong消息给后端，后端接受到消息在channelRead0方法里面更新最后一次接受到客户端的心跳时间。如果在一定时间内没有接受到客户端的心跳响应，那么服务端认为该客户端已经下线，将关闭该客户端。客户端收到下线通知后，会自动重连。

这是开启当个客户端之后，服务端的日志：



可以看到客户端收到服务端的ping心跳，回复pong消息给服务端，服务端更新接受到心跳的时间。