J-IM 开发文档

作者：王超(J-IM 作者)

官网：http://www.j-im.cn

联系方式(QQ)：1241503759

日期：2018 年 05 月 10 日

版本：V1.0.0

目录

第一章 j-im 简介

1.1 初识 j-im

j-im 是一个 IM(网络即时通讯)中间件，它是基于高性能的网络通讯框架 t-io 来开发的轻量级、高性能、易扩展、支持百万在线用户的 IM 中间件，它不仅可以作为 IM 服务无缝接入应用系统,同时也可以为系统提供可靠的消息推送、数据转发等服务,它的主要目标是降低即时通讯门槛，快速打造低成本接入在线 IM 系统，同时通过极简洁的消息格式(JSON)就可以实现多端不同协议间的消息发送如内置(Http、Websocket、Socket 自定义 IM 协议)等，并提供通过 Http 协议的 API 接口进行消息发送,发送方无需关心接收端属于什么协议！

第二章 j-im 入门

1.1 工程结构

首先来看下 j-im 的工程目录结构:分为五个子工程,分别为 jim-client、jim-common、jim-parent、jim-server、jim-server-demo ;工程截图如下：

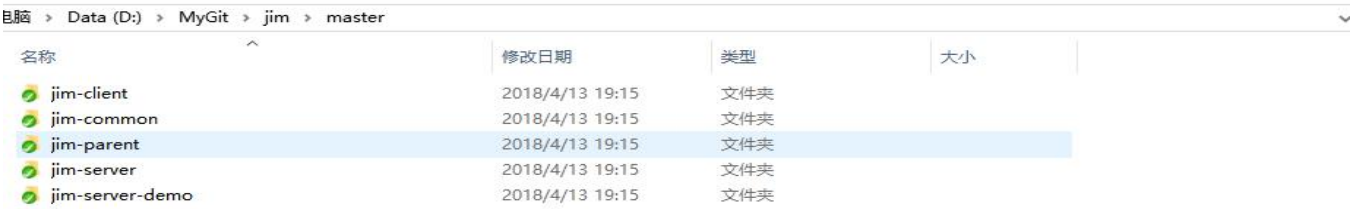

图 1 工程截图

各个项目用途:

1、jim-client(客户端入门工程，供开发者通过 socket 编写自己的 IM 客户端参考使用)
2、jim-common(j-im 客户端及服务端公用包，开发 server 端与 client 端所依赖包)
3、jim-parent(管理所有子工程及配置,包括版本号及 maven 库依赖等)
4、jim-server(j-im 服务端开发包,开发者可以基于该包开发属于自己的 IM 服务端程序)
5、jim-server-demo(笔者基于 jim-server 开发的 IM 服务端 demo,供开发者参考使用)

1.2 如何开发自己 IM 服务器

接下来我们来看下怎么基于 jim-server 开发一个属于自己的 im 服务端程序,本教程中以笔者开发的 jim-server-demo 来作为例子讲解如何快速开发一个 im 服务器。

在开发之前我们先来了解一下 j-im 的消息结构是什么样子,因为 j-im 通讯都是基于这些个消息结构及命令 cmd 来处理用户消息的,也可以帮助更好的理解后面的 jim-server 端是如何处理消息逻辑以及如何扩展实现自己的 cmd 命令及消息结构。

目前消息结构作者只列举了 8 个,因为作者只用到了其中这些,但是还有别的一些消息结构如：加入群组、退出群组等消息结构就不一一列举了，消息命令结构如下：

消息格式

1.聊天请求消息结构

```
{
   "from": "来源ID",
   "to": "目标ID",
   "cmd":"命令码(11)int类型",
   "createTime": "消息创建时间long类型",
   "msgType": "消息类型int类型(0:text、1:image、2:voice、3:vedio、4:music、5:news)",
   "chatType":"聊天类型int类型(0:未知,1:公聊,2:私聊)",
   "group_id":"群组id仅在chatType为(1)时需要,String类型",
   "content": "内容"
}
```

请求:COMMAND_CHAT_REQ(11) 响应:COMMAND_CHAT_RESP(12)

2.鉴权请求消息结构

```
{
    "cmd":"命令码(3)int类型",
    "token": "校验码"
}
```

请求:COMMAND_AUTH_REQ(3) 响应:COMMAND_AUTH_RESP(4)

3.握手请求消息结构

```
{
    "cmd":"命令码(1)int类型",
    "hbyte":"握手1个字节"
}
```

说明:请求:COMMAND_HANDSHAKE_REQ(1) 响应:COMMAND_HANDSHAKE_RESP(2)

4.登录请求消息结构

```
{
    "cmd":"命令码(5)int类型",
    "loginname": "用户名",
    "password": "密码",
    "token": "校验码(此字段可与logingname、password共存,也可只选一种方式)"
}
```

请求:COMMAND_LOGIN_REQ(5) 响应:COMMAND_LOGIN_RESP(6)

5.心跳请求消息结构

```
{
    "cmd":"命令码(13)int类型",
    "hbbyte":"心跳1个字节"
}
```

请求:COMMAND_HEARTBEAT_REQ(13) 响应:COMMAND_HEARTBEAT_REQ(13)

6.关闭、退出请求消息结构

```
{
    "cmd":"命令码(14)int类型",
    "userid":"用户id"
}
```

请求:COMMAND_CLOSE_REQ(14) 响应:无

7.获取用户信息请求消息结构

```
{
     "cmd":"命令码(17)int类型",
     "userid":"用户id(只在type为0或是无的时候需要)",
     "type":"获取类型(0:指定用户,1:所有在线用户,2:所有用户[在线+离线])"
}
```

请求:COMMAND_GET_USER_REQ(17) 响应:COMMAND_GET_USER_RESP(18)

8.获取用户消息请求结构

```
{
     "cmd":"命令码(19)int类型",
     "fromUserId":"消息发送用户id(此字段必须与userId一起使用,获取双方聊天消息),非必填",
     "userId":"当前用户id(必填字段),当只有此字段时,type必须为0, 意思是获取当前用户所有离线消息(好友+群组)",
     "groupId":"群组id(此字段必须与userId一起使用,获取当前用户指定群组聊天消息),非必填",
     "beginTime":"消息区间开始时间Date毫秒数double类型,非必填",
     "endTime":"消息区间结束时间Date毫秒数double类型,非必填",
     "offset":"分页偏移量int类型, 类似Limit 0,10 中的0,非必填",
     "count":"显示消息数量,类似Limit 0,10 中的10,非必填",
     "type":"消息类型(0:离线消息,1:历史消息)"
}
```

请求:COMMAND_GET_MESSAGE_REQ(19) 响应:COMMAND_GET_MESSAGE_RESP(20)

图 2 消息格式

具体每个消息结构及 cmd 命令码是做什么用的，上面都写的很详细，这里先简单认识下，后面也会有详细介绍，我

们继续往下看。

1. 引入 jim-server 开发包

在 pom.xml 文件中引入 jim-server(目前 maven 最新版 1.0.0.v20180413-RELEASE)

```
<dependency>
    <groupId>org.j-im</groupId>
    <artifactId>jim-server</artifactId>
    <version>1.0.0.v20180413-RELEASE</version>
</dependency>
```

图 3 pom.xml 中引入 jim-server

2. 定义启动类

```
public class ImServerDemoStart {

    public static void main(String[] args)throws Exception{
        PropKit.use("app.properties");
        int port = PropKit.getInt("port");//启动端口
        ImConfig.isStore =  PropKit.get("isStore");//是否开启持久化;(on:开启,off:不开启)
        ImConfig imConfig = new ImConfig(null, port);
        HttpServerInit.init(imConfig);
        //ImgMnService.start();//启动爬虫爬取模拟在线人头像;
        ImServerStarter imServerStarter = new ImServerStarter(imConfig,new ImDemoAioListener());
        /***************start 以下处理器根据业务需要自行添加与扩展，每个Command都可以添加扩展,此处为demo中处理*****************************/
        HandshakeReqHandler handshakeReqHandler = CommandManager.getCommand(Command.COMMAND_HANDSHAKE_REQ,HandshakeReqHandler.class);
        handshakeReqHandler.addProcessor(new DemoWsHandshakeProcessor());//添加自定义握手处理器;
        LoginReqHandler loginReqHandler = CommandManager.getCommand(Command.COMMAND_LOGIN_REQ,LoginReqHandler.class);
        loginReqHandler.addProcessor(new LoginServiceProcessor());//添加登录业务处理器;
        /***************end ***********************************************************************************/
        imServerStarter.start();
    }
}
```

图 4 启动类

3. 定义 cmd 命令业务处理器

根据业务需要定义相关 cmd 处理器，如果不需要额外的业务处理可以不需要添加，jim-server-demo 中定义了 DemoWsHandshakeProcessor、LoginServiceProcessor 两个业务处理器，分别应用在握手环节和登录环节其代码分别如下：

DemoWsHandshakeProcessor 代码：

```java
public class DemoWsHandshakeProcessor extends WsHandshakeProcessor{

    /**
     * WS握手方法，返回Null则业务层不同意握手，断开连接！
     */
    @Override
    public ImPacket handshake(ImPacket packet, ChannelContext channelContext) throws Exception {
        WsRequestPacket wsRequestPacket = (WsRequestPacket) packet;
        WsSessionContext wsSessionContext = (WsSessionContext) channelContext.getAttribute();
        if (wsRequestPacket.isHandShake()) {
            LoginReqHandler loginHandler = (LoginReqHandler)CommandManager.getCommand(Command.COMMAND_LOGIN_REQ);
            HttpRequest request = wsSessionContext.getHandshakeRequestPacket();
            String username = request.getParams().get("username") == null ? null : (String)request.getParams().get("username")[0];
            String password = request.getParams().get("password") == null ? null : (String)request.getParams().get("password")[0];
            String token = request.getParams().get("token") == null ? null : (String)request.getParams().get("token")[0];
            LoginReqBody loginBody = new LoginReqBody(username,password,token);
            byte[] loginBytes = JsonKit.toJsonBytes(loginBody);
            request.setBody(loginBytes);
            request.setBodyString(new String(loginBytes,HttpConst.CHARSET_NAME));
            Object loginResponse = loginHandler.handler(request, channelContext);
            if(loginResponse == null)
                return null;
            WsResponsePacket wsResponsePacket = new WsResponsePacket();
            wsResponsePacket.setHandShake(true);
            wsResponsePacket.setCommand(Command.COMMAND_HANDSHAKE_RESP);
            wsSessionContext.setHandshaked(true);
            return wsResponsePacket;
        }
        return null;
    }
}
```

图 5 DemoWsHandshakeProcessor 类

LoginServiceProcessor 代码：

```java
public class LoginServiceProcessor implements LoginProcessorIntf{

    public static final Map<String, User> tokenMap = new HashMap<>();

    private static String[] familyName = new String[] { "谭", "刘", "张", "李", "胡", "沈", "朱", "钱", "王", "伍", "赵", "孙", "吕", "马", "秦", "毛", "成", "梅", "黄", "郭", "杨", "季", "董", "习", "郑",

    private static String[] secondName = new String[] { "艺昕", "红薯", "明远", "天蓬", "三丰", "德华", "歌", "佳", "乐", "天", "燕子", "子牛", "海", "燕", "花", "娟", "冰冰", "丽娅", "大为", "无为", "渔民", "大赋"

     * 根据用户名和密码获取用户
    public User getUser(String loginname, String password) {
     * 根据token获取用户信息
    public User getUser(String token) {

    public List<Group> initGroups(User user){
    public List<Group> initFriends(User user){

    public String nextImg() {

    public String newToken() {

    public User getUser(LoginReqBody loginReqBody , ChannelContext channelContext) {

    public boolean isProtocol(ChannelContext channelContext) {

    public String name() {
}
```

图 6 LoginServiceProcessor 类

具体怎么扩展使用，下面会有详细介绍，这里先简单认识下！

4. 定义服务端用户通道监听器

这个用户根据需要自己来定义，不需要的话也可以不用定义，j-im 会用默认的监听器，下面是 jim-server-demo 中的定义实现：

```java
public class ImDemoAioListener extends ImServerAioListener{

    private Logger log = LoggerFactory.getLogger(ImDemoAioListener.class);

    @Override
    public void onAfterSent(ChannelContext channelContext, Packet packet, boolean isSentSuccess) {
        ImPacket imPacket = (ImPacket)packet;
        if(imPacket.getCommand() == Command.COMMAND_LOGIN_RESP || imPacket.getCommand() == Command.COMMAND_HANDSHAKE_RESP){//首次登陆;
            ImSessionContext imSessionContext = (ImSessionContext)channelContext.getAttribute();
            User user = imSessionContext.getClient().getUser();
            if(user.getGroups() != null){
                for(Group group : user.getGroups()){//绑定群组并发送加入群组通知
                    ImPacket groupPacket = new ImPacket(Command.COMMAND_JOIN_GROUP_REQ,JsonKit.toJsonBytes(group));
                    try {
                        new JoinGroupReqHandler().handler(groupPacket, channelContext);
                    } catch (Exception e) {
                        log.error(e.toString(),e);
                    }
                }
            }
        }
    }
}
```

图 7 ImDemoAilListener 类

demo 中只用到了其中一个方法 onAfterSent，但是 ImServerAioListener 中远不止这些，还有其它一些接口方法如下：

```java
public interface AioListener {
    /**
     * 连接关闭前后触发本方法
     */
    void onAfterClose(ChannelContext channelContext, Throwable throwable, String remark, boolean isRemove) throws Exception;

    /**
     * 建链后触发本方法，注：建链不一定成功，需要关注参数isConnected
     */
    void onAfterConnected(ChannelContext channelContext, boolean isConnected, boolean isReconnect) throws Exception;

    /**
     * 解码成功后触发本方法
     */
    void onAfterReceived(ChannelContext channelContext, Packet packet, int packetSize) throws Exception;

    /**
     * 消息包发送之后触发本方法
     */
    void onAfterSent(ChannelContext channelContext, Packet packet, boolean isSentSuccess) throws Exception;

    /**
     * 连接关闭前触发本方法
     */
    void onBeforeClose(ChannelContext channelContext, Throwable throwable, String remark, boolean isRemove);
}
```

图 8 AioListener 接口

读者可以根据自己需要选择性重写其中的方法，进行业务操作，详细使用，下面也会细说。

5. 注册添加 cmd 业务处理器及配置通道监听器

如果没有定义 3、4 步的话不需要注册，这里 demo 中有用到代码如下：

```java
public class ImServerDemoStart {

    public static void main(String[] args)throws Exception{
        PropKit.use("app.properties");
        int port = PropKit.getInt("port");//启动端口
        ImConfig.isStore =  PropKit.get("isStore");//是否开启持久化;(on:开启,off:不开启)
        ImConfig imConfig = new ImConfig(null, port);
        HttpServerInit.init(imConfig);
        //ImgMnService.start();//启动爬虫爬取模拟在线人头像;
        ImServerStarter imServerStarter = new ImServerStarter(imConfig,new ImDemoAioListener());
        /****************start 以下处理器根据业务需要自行添加与扩展，每个Command都可以添加扩展,此处为demo中处理*****************************/
        HandshakeReqHandler handshakeReqHandler = CommandManager.getCommand(Command.COMMAND_HANDSHAKE_REQ,HandshakeReqHandler.class);
        handshakeReqHandler.addProcessor(new DemoWsHandshakeProcessor());//添加自定义握手处理器;
        LoginReqHandler loginReqHandler = CommandManager.getCommand(Command.COMMAND_LOGIN_REQ,LoginReqHandler.class);
        loginReqHandler.addProcessor(new LoginServiceProcessor());//添加登录业务处理器;
        /****************end ***********************************************************************************/
        imServerStarter.start();
    }
}
```

图 9

6. 启动

程序可以右键直接运行启动类的 main 方法即可启动看到运行效果，如果是打包发布的话,作者打包了 2 个启动脚本，分别是 windows、linux 下的启动脚本，一键运行。

startup.bat	2018/3/30 17:39	Windows 批处理文件	1 KB
startup.sh	2018/3/30 17:41	Shell Script	1 KB

图 10

脚本可在 J-IM 群里下载。

1.3 如何开发自己的 IM 客户端

接下来我们来看下怎么基于 j-im 来开发一个自己的 im 客户端程序，本教程以笔者开发的 jim-client 来作为列子来讲解仅供参考，这里的客户端指的是普通 socket 客户端，在讲解之前我们先来了解一下 j-im 的自定义 im 协议是什么，知道这个以后，开发其它语言的客户端如 C++、android、ios、php 等，就可以自己基于 j-im 的 socket 协议进行消息发送、接收的编解码操作了，

1. 详解 j-im 自定义 socket 协议结构

协议结构如下：

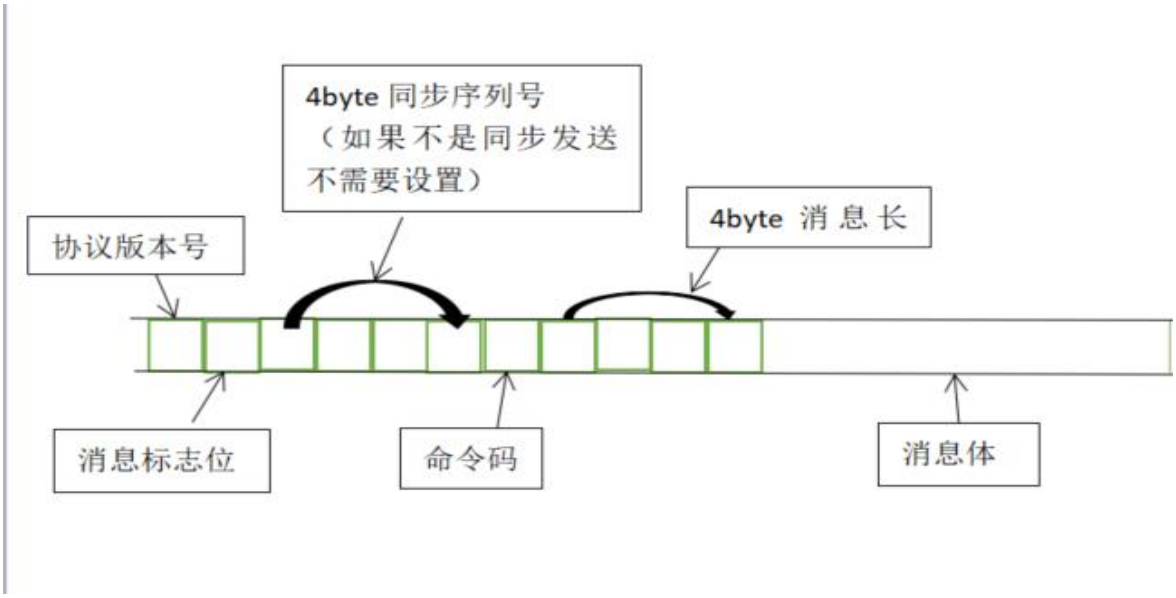

图 11

第 1 个字节

版本号：`byte version = 0x01`,用于描述当前协议版本号，因为后面协议随着版本升级，这个版本号也会发生变化，目前是 1,十六进制表示就是 0x01;

第 2 个字节

消息标志位 mask：用于描述消息是否加密、压缩、同步发送、4 字节长度等,可以通过以下 java 代码来获取：

```
byte maskByte = ImPacket.encodeEncrypt(version, isEncrypt);//是否加密;
maskByte = ImPacket.encodeCompress(maskByte, isCompress);//是否压缩;
maskByte = ImPacket.encodeHasSynSeq(maskByte, isHasSynSeq);//是否同步发送;
maskByte = ImPacket.encode4ByteLength(maskByte, is4ByteLength);//是否 4 字节表示消息体长度;
```

定义的每一位常量字节如下：

```
/**
 * 加密标识位mask，1为加密，否则不加密
 */
public static final byte FIRST_BYTE_MASK_ENCRYPT = -128;

/**
 * 压缩标识位mask，1为压缩，否则不压缩
 */
public static final byte FIRST_BYTE_MASK_COMPRESS = 0B01000000;

/**
 * 是否有同步序列号标识位mask，如果有同步序列号，则消息头会带有同步序列号，否则不带
 */
public static final byte FIRST_BYTE_MASK_HAS_SYNSEQ = 0B00100000;

/**
 * 是否是用4字节来表示消息体的长度
 */
public static final byte FIRST_BYTE_MASK_4_BYTE_LENGTH = 0B00010000;
```

图 12

第一位是否加密的获取方法如下：

```
public static byte encodeEncrypt(byte bs,boolean isEncrypt){
    if(isEncrypt){
        return (byte) (bs | Protocol.FIRST_BYTE_MASK_ENCRYPT);
    }else{
        return (byte)(Protocol.FIRST_BYTE_MASK_ENCRYPT & 0b01111111);
    }
}
```

图 13

第二位是否压缩的获取方法如下：

```
public static byte encodeCompress(byte bs, boolean isCompress)
{
    if (isCompress)
    {
        return (byte) (bs | Protocol.FIRST_BYTE_MASK_COMPRESS);
    } else
    {
        return (byte) (bs & (Protocol.FIRST_BYTE_MASK_COMPRESS ^ 0b01111111));
    }
}
```

图 14

第三位是否同步发送的获取方法如下：

```
public static byte encodeHasSynSeq(byte bs, boolean hasSynSeq)
{
    if (hasSynSeq)
    {
        return (byte) (bs | Protocol.FIRST_BYTE_MASK_HAS_SYNSEQ);
    } else
    {
        return (byte) (bs & (Protocol.FIRST_BYTE_MASK_HAS_SYNSEQ ^ 0b01111111));
    }
}
```

图 15

第四位是否 4 字节表示消息体长度获取方法如下：

```
public static byte encode4ByteLength(byte bs, boolean is4ByteLength)
{
    if (is4ByteLength)
    {
        return (byte) (bs | Protocol.FIRST_BYTE_MASK_4_BYTE_LENGTH);
    } else
    {
        return (byte) (bs & (Protocol.FIRST_BYTE_MASK_4_BYTE_LENGTH ^ 0b01111111));
    }
}
```

图 16

其它语言可根据这个位运算自己来获取即可。

第 3-6 这 4 个字节

同步发送序列号：它占用 4 个字节，存放同步序列号如 1200、1201、1203、…等的整型数字，这里注意，如果消息非同步发送也就是第二个字节 mask 中的第三位为 0，那这里这 4 个 byte 是不需要设置的,为 1 的话才需要设置;

第 7 个字节

cmd 命令码：占用 1 个字节，比如聊天请求的消息命令码为 11，登录请求的消息命令码为 5，其它支持的 cmd 命令码在下面会有详细介绍及如何使用，它的获取方式如下：

```
byte cmdByte = 0x00;
cmdByte = (byte) (cmdByte|你的消息命令码);//消息类型;
```

第 8-11 个字节

消息体长度：占用 4 个字节，这个好理解，直接获取消息包的 body 的字节数 length 就可以获取到。

最后就是接上要发送的消息数据了。
了解了 j-im 的自定义协议以后就好理解了,我们继续往下看；

2. 引入 jim-common 公共开发包

```
<dependency>
  <groupId>org.j-im</groupId>
  <artifactId>jim-common</artifactId>
  <version>1.0.0.v20180413-RELEASE</version>
</dependency>
```

图 17

3. 定义 Packet

如果需要有自己的消息包接口可以自定义 Packet 继承父类 TcpPacket 即可，比如：MyPacket **extends** TcpPacket，如果不需要自己的消息包，直接用 TcpPacket 即可,代码如下：

```
public class TcpPacket extends ImPacket{

    private static final long serialVersionUID = -4283971967100935982L;

    private byte version = 0;
    private byte mask = 0;

    public TcpPacket(){}

    public TcpPacket(Command command, byte[] body){
        super(command, body);
    }
    public TcpPacket( byte[] body){
        super(body);
    }
    public byte getVersion() {
        return version;
    }
    public void setVersion(byte version) {
        this.version = version;
    }
    public byte getMask() {
        return mask;
    }
    public void setMask(byte mask) {
        this.mask = mask;
    }

}
```

图 18

4. 定义客户端消息处理器

这个处理器比较重要，它负责所有消息的发送(编码)、接收(解码)、业务处理，代码如下：

```java
public class HelloClientAioHandler  implements AioHandler,ClientAioHandler
{
    /**
     * 处理消息
     */
    @Override
    public void handler(Packet packet, ChannelContext channelContext) throws Exception
    {
        TcpPacket helloPacket = (TcpPacket)packet;
        byte[] body = helloPacket.getBody();
        if (body != null)
        {
            String str = new String(body, Const.CHARSET);
            System.out.println("收到消息：" + str);
        }

        return;
    }
    /**
     * 编码：把业务消息包编码为可以发送的ByteBuffer
     * 总的消息结构：消息头 + 消息体
     * 消息头结构：    4个字节，存储消息体的长度
     * 消息体结构：   对象的json串的byte[]
     */
    @Override
    public ByteBuffer encode(Packet packet, GroupContext groupContext, ChannelContext channelContext)
    {
        TcpPacket tcpPacket = (TcpPacket)packet;
        return TcpServerEncoder.encode(tcpPacket, groupContext, channelContext);
    }

    @Override
    public TcpPacket decode(ByteBuffer buffer, ChannelContext channelContext) throws AioDecodeException {
        TcpPacket tcpPacket = TcpServerDecoder.decode(buffer, channelContext);
        return tcpPacket;
    }

    private static TcpPacket heartbeatPacket = new TcpPacket(Command.COMMAND_HEARTBEAT_REQ,new byte[]{Protocol.HEARTBEAT_BYTE});

    /**
     * 此方法如果返回null，框架层面则不会发心跳；如果返回非null，框架层面会定时发本方法返回的消息包
     */
    @Override
    public TcpPacket heartbeatPacket()
    {
        return heartbeatPacket;
    }
}
```

图 19

如果你是用 java 语言并且是基于 j-im 来开发客户端的话，那非常简单，编解码自己不需要来编写了，j-im 已经帮我们处理好了，分别在 TcpServerEncode.encode 和 TcpServerDecoder.decode 中，调用即可，如果不是基于 j-im 来开发客户端的话，自己按照上面介绍的协议结构封装好编解码就可以了，其它语言客户端同理。

5. 定义启动类

客户端程序入口，代码如下：

```java
public class HelloClientStarter {
    //服务器节点
    public static Node serverNode = new Node("127.0.0.1", Const.SERVER_PORT);

    //handler, 包括编码、解码、消息处理
    public static ClientAioHandler aioClientHandler = new HelloClientAioHandler();

    //事件监听器，可以为null，但建议自己实现该接口，可以参考showcase了解些接口
    public static ClientAioListener aioListener = null;

    //断链后自动连接的，不想自动连接请设为null
    private static ReconnConf reconnConf = new ReconnConf(5000L);

    //一组连接共用的上下文对象
    public static ClientGroupContext clientGroupContext = new ClientGroupContext(aioClientHandler, aioListener, reconnConf);

    public static AioClient aioClient = null;
    public static ClientChannelContext clientChannelContext = null;

    /**
     * 启动程序入口
     */
    public static void main(String[] args) throws Exception {
        //clientGroupContext.setHeartbeatTimeout(org.tio.examples.helloworld.common.Const.TIMEOUT);
        clientGroupContext.setHeartbeatTimeout(0);
        aioClient = new AioClient(clientGroupContext);
        clientChannelContext = aioClient.connect(serverNode);
        //连上后，发条消息玩玩
        send();
    }

    private static void send() throws Exception {
        byte[] loginBody = new LoginReqBody("hello_client","123").toByte();
        TcpPacket loginPacket = new TcpPacket(Command.COMMAND_LOGIN_REQ,loginBody);
        Aio.send(clientChannelContext, loginPacket);//先登录;
        ChatBody chatBody = new ChatBody()
                .setFrom("hello_client")
                .setTo("admin")
                .setMsgType(0)
                .setChatType(1)
                .setGroup_id("100")
                .setContent("Socket普通客户端消息测试!");
        TcpPacket chatPacket = new TcpPacket(Command.COMMAND_CHAT_REQ,chatBody.toByte());
        Aio.send(clientChannelContext, chatPacket);
    }
}
```

图 20

6. 运行客户端

运行客户端程序 `org.jim.client.HelloClientStarter.main(String[] args)`即可启动。

1.4 常用类介绍及如何使用

这里说明下其中几个重要的类：

1. ImConfig 类

ImConfig 这个类是 j-im 配置启动参数所需要的，比如绑定 IP、端口、持久化控制等。

```java
public class ImConfig {

    private String bindIp = null;

    /**
     * 监听端口
     */
    private Integer bindPort = 80;
    /**
     * 心跳包发送时长heartbeatTimeout/2
     */
    private long heartbeatTimeout = 0;
    /**
     * http相关配置;
     */
    private HttpConfig httpConfig;
    /**
     * websocket相关配置;
     */
    private WsServerConfig wsServerConfig;
    /**
     * 全局群组上下文;
     */
    public static GroupContext groupContext;
    /**
     * 用户消息持久化助手;
     */
    private static IMesssageHelper messageHelper;
    /**
     * 是否开启持久化;
     */
    public static String isStore;
    /**
     *  默认的接收数据的buffer size
     */
    private long readBufferSize = 1024 * 1024;
```

图 21 ImConfig 类几个重要属性

这里着重说下以下几个属性的含义及作用：

(1) heartbeatTimeout：设置服务端检测心跳超时时长参数，保证服务端可以感知到客户端是否还在线，单位是毫秒，如果用户不希望服务端做心跳检查相关工作，请把此值设为 0 或负数，比如这里设置了 12000，那服务端会每隔 12000/2=6000 毫秒，也就是每 6 秒钟服务端会检测一下客户端消息收发状态，如果这个时间内客户端没有发送心跳，那么服务器端会断开客户端连接并释放所有连接资源。

(2) httpConfig：因为 j-im 是支持多协议(Http、Websockt、Socket)的，所以这里便是 Http 协议的配置，如本例 jim-server-demo 中在 app.properties 文件中配置的访问资源文件(html/css/js 等)根目录为 http.page = classpath:page，以及自带 mvc 支持所需要扫描的根目录包等。启动类中配置 Http 协议代码如下:

```java
public class HttpServerInit {

    public static void init(ImConfig imConfig) throws Exception {
        PropKit.use("app.properties");
        String pageRoot = PropKit.get("http.page");//html/css/js等的根目录，支持classpath:，也支持绝对路径
        String[] scanPackages = new String[] { ImServerDemoStart.class.getPackage().getName() };//j-im mvc需要扫描的根目录包
        HttpConfig httpConfig = new HttpConfig();
        httpConfig.setPageRoot(pageRoot);//设置web访问路径;
        httpConfig.setMaxLiveTimeOfStaticRes(0);//不缓存资源;
        httpConfig.setScanPackages(scanPackages);//设置j-im mvc扫描目录;
        imConfig.setHttpConfig(httpConfig);
    }
```

图 22

(3) **wsConfig**：跟 httpConfig 一样道理，不过 WsConfig 中多了一个属性 IWsMsgHandler，它的接口属性接口方法如下：

```java
public class WsMsgHandler implements IWsMsgHandler{
    private static Logger log = LoggerFactory.getLogger(WsMsgHandler.class);

    private WsServerConfig wsServerConfig = null;

    /**
     *
     * @param websocketPacket
     * @param text
     * @param channelContext
     * @return 可以是WsResponsePacket、String、null
     * @author: WChao
     */
    public Object onText(WsRequestPacket wsRequestPacket, String text, ChannelContext channelContext) throws Exception {

    /**
     *
     * @param websocketPacket
     * @param bytes
     * @param channelContext
     * @return 可以是WsResponsePacket、byte[]、ByteBuffer、null
     * @author: WChao
     */
    public Object onBytes(WsRequestPacket websocketPacket, byte[] bytes, ChannelContext channelContext) throws Exception {

    /**
     * @param packet
     * @param channelContext
     * @return
     * @throws Exception
     * @author: WChao
     */
    public WsResponsePacket handler(ImPacket imPacket, ChannelContext channelContext)throws Exception {

    public WsResponsePacket h(WsRequestPacket wsRequest, byte[] bytes, Opcode opcode, ChannelContext channelContext) throws Exception {

    private WsResponsePacket processRetObj(Object obj, String methodName, ChannelContext channelContext) throws Exception {
    @Override
    public Object onClose(WsRequestPacket websocketPacket, byte[] bytes, ChannelContext channelContext) throws Exception {
        Aio.remove(channelContext, "receive close flag");
        return null;
    }
```

```java
public interface IWsMsgHandler
{
     * @param packet
    public ImPacket handler(ImPacket packet, ChannelContext channelContext)  throws Exception;
    /**
     * @param websocketPacket
     * @param text
     * @param channelContext
     * @return 可以是WsResponsePacket、byte[]、ByteBuffer、String或null，如果是null，框架不会回消息
     * @throws Exception
     * @author: WChao
     */
    Object onText(WsRequestPacket wsPacket, String text, ChannelContext channelContext) throws Exception;

    /**
     *
     * @param websocketPacket
     * @param bytes
     * @param channelContext
     * @return 可以是WsResponsePacket、byte[]、ByteBuffer、String或null，如果是null，框架不会回消息
     * @throws Exception
     * @author: WChao
     */
    Object onClose(WsRequestPacket websocketPacket, byte[] bytes, ChannelContext channelContext) throws Exception;

    /**
     *
     * @param websocketPacket
     * @param bytes
     * @param channelContext
     * @return 可以是WsResponsePacket、byte[]、ByteBuffer、String或null，如果是null，框架不会回消息
     * @throws Exception
     * @author: WChao
     */
    Object onBytes(WsRequestPacket websocketPacket, byte[] bytes, ChannelContext channelContext) throws Exception;
}
```

图 23

用户可以通过实现这个接口来扩展实现自己的 WsMsgHandler 处理，如 j-im 默认实现的 WsMsgHandler 如下:

```java
public class WsMsgHandler implements IWsMsgHandler{
    private static Logger log = LoggerFactory.getLogger(WsMsgHandler.class);

    private WsServerConfig wsServerConfig = null;

    /**
     *
     * @param websocketPacket
     * @param text
     * @param channelContext
     * @return 可以是WsResponsePacket、String、null
     * @author: WChao
     */
    public Object onText(WsRequestPacket wsRequestPacket, String text, ChannelContext channelContext) throws Exception {

    /**
     *
     * @param websocketPacket
     * @param bytes
     * @param channelContext
     * @return 可以是WsResponsePacket、byte[]、ByteBuffer、null
     * @author: WChao
     */
    public Object onBytes(WsRequestPacket websocketPacket, byte[] bytes, ChannelContext channelContext) throws Exception {

    /**
     * @param packet
     * @param channelContext
     * @return
     * @throws Exception
     * @author: WChao
     */
    public WsResponsePacket handler(ImPacket imPacket, ChannelContext channelContext)throws Exception {

    public WsResponsePacket h(WsRequestPacket wsRequest, byte[] bytes, Opcode opcode, ChannelContext channelContext) throws Exception {

    private WsResponsePacket processRetObj(Object obj, String methodName, ChannelContext channelContext) throws Exception {
    @Override
    public Object onClose(WsRequestPacket websocketPacket, byte[] bytes, ChannelContext channelContext) throws Exception {
        Aio.remove(channelContext, "receive close flag");
        return null;
    }
```

图 24

然后通过 ImConfig.setWsServerConfig(wsServerConfig)设置进去即可。

(4) isStore：控制是否开启持久化存储,没什么好说的(on:开启,off:不开启);

(5) readBufferSize：设置 t-io 默认接收数据的缓冲区大小，j-im 默认设置的是 1M，这个参数使用过 t-io 的用户应该知道，t-io 会自动帮助我们进行半包、粘包处理，有时候传输的数据比较多或者比较大时，可以看到控制台有日志信息输出提示你解码失败，本次参与解码长度是多少等信息，其实这个只是个日志信息而已，并不影响数据传输的，这正是 t-io 在给我们进行组包、拆包的过程,提示信息可以帮助我们更好监测数据传输的准确性，同时 t-io 默认的缓冲区大小是 2KB,有点抠，跟 t-io 作者性格有点像，哈哈，开玩笑啦，所以会有比较多的日志信息输出，所以 j-im 默认设置的是 1M，可以稍微减少一些频繁组包的过程。

2. ImServerGroupContext 类

它是用来负责服务配置与维护,配置线程池、确定监听端口，维护客户端各种数据等的。

我们在写 TCP Server 时，都会先选好一个端口以监听客户端连接，再创建 N 组线程池来执行相关的任务，譬如发送消息、解码数据包、处理数据包等任务，还要维护客户端连接的各种数据，为了和业务互动，还要把这些客户端连接和各种业务数据绑定起来，譬如把某个客户端绑定到一个群组，绑定到一个 userid，绑定到一个 token 等。GroupContext 就是用来配置线程池、确定监听端口，维护客户端各种数据等的。

GroupContext 是个抽象类，如果你是用 tio 作 tcp 客户端，那么你需要创建 ClientGroupContext，如果你是用 tio 作 tcp 服务器，那么你需要创建 ServerGroupContext,而这里我们就是用 tio 作为服务器端，所以这里存储的就是 ServerGroupContext，只不过 j-im 自己封装了一下叫做 ImServerGroupContext，这个用户是不需要配置

的,jim-server 默认会帮我们初始化好，在后面自己开发的过程中如果需要用到这个参数，那我们直接从 ImConfig 中拿就好了。

GroupContext 对象包含的信息非常多，主要对象见下图

图 25 GroupContext 主要对象

① ServerGroupContext

GroupContext 的子类，当用 tio 作 tcp 服务器时，业务层接触的是这个类的实例。

② ClientGroupContext

GroupContext 的子类，当用 tio 作 tcp 客户端时，业务层接触的是这个类的实例。

3. IMessageHelper 接口类

持久化消息助手,用于 j-im 消息持久化存储，接口方法如下：

```
public interface IMesssageHelper {
     * 获取im群组、人员绑定监听器;
    public ImBindListener getBindListener();
     * 添加群组成员
    public void addGroupUser(String userid,String group_id);
     * 获取群组所有成员;
    public List<String> getGroupUsers(String group_id);
     * 消息持久化写入
    public void writeMessage(String timelineTable , String timelineId , ChatBody chatBody);
     * 移除群组用户
    public void removeGroupUser(String userid,String group_id);
     * 获取与指定用户离线消息;
    public UserMessageData getFriendsOfflineMessage(String userid,String fromUserId);
     * 获取与所有用户离线消息;
    public UserMessageData getFriendsOfflineMessage(String userid);
     * 获取用户指定群组离线消息;
    public UserMessageData getGroupOfflineMessage(String userid,String groupid);
     * 获取与指定用户历史消息;
    public UserMessageData getFriendHistoryMessage(String userid, String fromUerId,Double beginTime,Double endTime,Integer offset,Integer count);

     * 获取与指定群组历史消息;
    public UserMessageData getGroupHistoryMessage(String userid, String groupid,Double beginTime,Double endTime,Integer offset,Integer count);
}
```

图 26 IMessageHelper 接口

默认 j-im 消息持久化采用的是 Redis 进行存储实现代码如下：

```
/**
 * Redis获取持久化+同步消息助手;
 * @author WChao
 * @date 2018年4月9日 下午4:39:30
 */
public class RedisMessageHelper implements IMesssageHelper,Const {

    private RedisCache groupCache = null;
    private RedisCache pushCache = null;
    private RedisCache storeCache = null;
    private RedisCache userCache = null;

    private final String SUBFIX = ":";
    private Logger log = LoggerFactory.getLogger(RedisMessageHelper.class);

    public RedisMessageHelper(){
    static{

    public ImBindListener getBindListener() {

    public List<String> getGroupUsers(String group_id) {


    public void writeMessage(String timelineTable, String timelineId, ChatBody chatBody) {


    public void addGroupUser(String userid, String group_id) {

    public void removeGroupUser(String userid, String group_id) {

    public UserMessageData getFriendsOfflineMessage(String userid, String from_userid) {

    public UserMessageData getFriendsOfflineMessage(String userid) {

    public UserMessageData getGroupOfflineMessage(String userid, String groupid) {

    public UserMessageData getFriendHistoryMessage(String userid, String from_userid,Double beginTime,Double endTime,Integer offset,Integer count) {

    public UserMessageData getGroupHistoryMessage(String userid, String groupid,Double beginTime,Double endTime,Integer offset,Integer count) {

     * 放入用户群组消息;
    public UserMessageData putGroupMessage(UserMessageData userMessage,List<ChatBody> messages){
    /**
```

图 27 RedisMessageHelper 消息持久化

如果用户需要自己实现一个持久化并应用到 j-im 中,可以这样比如 MongoDbMessageHelper 实现 IMessageHelper 接口实现其中的接口方法，然后通过 `ImConfig.setMessageHelper(helper)`方法设置进去，就可以实现自己的消息持久化存储了，非常简单！

4. ImServerStarter 类

`ImServerStarter` 服务端启动类，它有两个构造方法

```
public ImServerStarter(ImConfig imConfig){
public ImServerStarter(ImConfig imConfig,ImServerAioListener imAioListener){
```

图 28 ImServerStarter 服务端启动类

```
ImServerStarter imServerStarter = new ImServerStarter(imConfig);
ImServerStarter imServerStarter = new ImServerStarter(imConfig,new ImDemoAioListener());
```

如果，第二个参数 ImServerAioListener 没有传递，那默认采用 j-im 默认的客户端监听器，如果用户需要自己定义自己的监听器以监听客户端消息状态，则通过继承 ImServerAioListener 来实现其中的监听方法设置即可，如 jim-server-demo 中的 ImDemoAioListener 便实现了自己的监听逻辑代码如下：

```
public class ImDemoAioListener extends ImServerAioListener{

    private Logger log = LoggerFactory.getLogger(ImDemoAioListener.class);

    @Override
    public void onAfterSent(ChannelContext channelContext, Packet packet, boolean isSentSuccess) {
        ImPacket imPacket = (ImPacket)packet;
        if(imPacket.getCommand() == Command.COMMAND_LOGIN_RESP || imPacket.getCommand() == Command.COMMAND_HANDSHAKE_RESP){//首次登陆;
            ImSessionContext imSessionContext = (ImSessionContext)channelContext.getAttribute();
            User user = imSessionContext.getClient().getUser();
            if(user.getGroups() != null){
                for(Group group : user.getGroups()){//绑定群组并发送加入群组通知
                    ImPacket groupPacket = new ImPacket(Command.COMMAND_JOIN_GROUP_REQ,JsonKit.toJsonBytes(group));
                    try {
                        new JoinGroupReqHandler().handler(groupPacket, channelContext);
                    } catch (Exception e) {
                        log.error(e.toString(),e);
                    }
                }
            }
        }
    }
}
```

图 29 ImDemoAioListener 类

在这里顺带介绍一下 AioListener 接口：

AioListener 是处理消息的核心接口，它有两个子接口，ClientAioListener 和 ServerAioListener，当用 tio 作 tcp 客户端时需要实现 ClientAioListener，当用 tio 作 tcp 服务器时需要实现 ServerAioListener，它主要定义了如下方法：

```java
public interface AioListener {
    /**
     * 连接关闭前后触发本方法
     * @param channelContext the channelcontext
     * @param throwable the throwable 有可能为空
     * @param remark the remark 有可能为空
     * @param isRemove 是否是删除
     * @throws Exception
     * @author: tanyaowu
     */
    void onAfterClose(ChannelContext channelContext, Throwable throwable, String remark, boolean isRemove) throws Exception;

    /**
     * 建链后触发本方法，注：建链不一定成功，需要关注参数isConnected
     * @param channelContext
     * @param isConnected 是否连接成功,true:表示连接成功，false:表示连接失败
     * @param isReconnect 是否是重连，true: 表示这是重新连接，false: 表示这是第一次连接
     * @throws Exception
     * @author: tanyaowu
     */
    void onAfterConnected(ChannelContext channelContext, boolean isConnected, boolean isReconnect) throws Exception;

    /**
     * 解码成功后触发本方法
     * @param channelContext
     * @param packet
     * @param packetSize
     * @throws Exception
     * @author: tanyaowu
     */
    void onAfterReceived(ChannelContext channelContext, Packet packet, int packetSize) throws Exception;

    /**
     * 消息包发送之后触发本方法
     * @param channelContext
     * @param packet
     * @param isSentSuccess true:发送成功，false:发送失败
     * @throws Exception
     * @author tanyaowu
     */
    void onAfterSent(ChannelContext channelContext, Packet packet, boolean isSentSuccess) throws Exception;

    /**
     * 连接关闭前触发本方法
     * @param channelContext the channelcontext
     * @param throwable the throwable 有可能为空
     * @param remark the remark 有可能为空
     * @param isRemove
     * @author tanyaowu
     */
    void onBeforeClose(ChannelContext channelContext, Throwable throwable, String remark, boolean isRemove);
}
```

图 30 AioListener 接口

5. CommandManager 类

CommandManager **类比较重要**：cmd 命令处理器管理中心，它管理 j-im 中所有 cmd 命令的注册、删除、获取命令处理器等。

目前 j-im 中主要包含 9 大命令处理器分别如下：

每个请求处理器接收消息结构分别对应上面我们介绍的 j-im 各个消息结构，读者可以自行参考;

AuthReqHandler：鉴权请求处理器,COMMAND_AUTH_REQ(3) 响应:COMMAND_AUTH_RESP(4);

ChatReqHandler：聊天请求处理器,请求:COMMAND_CHAT_REQ(11) 响应:COMMAND_CHAT_RESP(12);

CloseReqHandler：关闭、退出请求处理器,请求:COMMAND_CLOSE_REQ(14) 响应:无;

HandshakeReqHandler：握手请求处理器,请求:COMMAND_HANDSHAKE_REQ(1) 响应:COMMAND_HANDSHAKE_RESP(2);

HeartbeatReqHandler：心跳请求处理器,请求:COMMAND_HEARTBEAT_REQ(13) 响应:COMMAND_HEARTBEAT_REQ(13);

JoinGroupReqHandler：加入群组请求处理器,请求:COMMAND_JOIN_GROUP_REQ(7) 响应:COMMAND_JOIN_GROUP_RESP(8);

LoginReqHandler：登录请求处理器,请求:COMMAND_LOGIN_REQ(5) 响应:COMMAND_LOGIN_RESP(6);

MessageReqHandler：获取用户消息请求处理器,请求:COMMAND_GET_MESSAGE_REQ(19) 响应:COMMAND_GET_MESSAGE_RESP(20);

UserReqHandler：获取用户信息请求处理器,请求:COMMAND_GET_USER_REQ(17) 响应:COMMAND_GET_USER_RESP(18);

这些 cmd 处理器类都是继承了同一个抽象类 AbCmdHandler，也就是说如果你自己需要扩展实现一个 cmd 处理器类，也需要继承这个抽象类然后实现其中的抽象方法就可以了，比如我们拿其中一个握手环节 HandshakeReqHandler 来举列子代码如下：

```
public class HandshakeReqHandler extends AbCmdHandler {

    @Override
    public ImPacket handler(ImPacket packet, ChannelContext channelContext) throws Exception {
        ProcessorIntf proCmdHandler = this.getProcessor(channelContext);
        if(proCmdHandler == null){
            Aio.remove(channelContext, "没有对应的握手协议处理器HandshakeProCmd...");
            return null;
        }
        HandshakeProcessorIntf handShakeProCmdHandler = (HandshakeProcessorIntf)proCmdHandler;
        ImPacket handShakePacket = handShakeProCmdHandler.handshake(packet, channelContext);
        if (handShakePacket == null) {
            Aio.remove(channelContext, "业务层不同意握手");
        }
        return handShakePacket;
    }

    @Override
    public Command command() {
        return Command.COMMAND_HANDSHAKE_REQ;
    }
}
```

图 31 HandshakeReqHandler 类

加入你觉得你想替换掉这个握手处理器，OK，那你写一个比如 MyHandshakeReqHandler 继承抽象类 AbCmdHandler，实现其中的抽象方法就可以了，然后怎么设置进去呢，这就用到了我们上面提到的 CommandManager 类了，他就是负责管理你的 cmd 处理器类的，通过它 CommandManager.registerCommand(imCommandHandler),来注册进去我们定义的这个 MyHandshakeReqHandler 就可以了！代码如下：

MyHandshakeReqHandler myHandshakeReqHandler = new MyHandshakeReqHandler();

CommandManager.registerCommand(myHandshakeReqHandler);

这样就可以使用我们自己定义的处理器类了！这里，细心的人会发现上面握手环节代码里面有这么几行代码，它是做什么用的？

```
public class HandshakeReqHandler extends AbCmdHandler {

    @Override
    public ImPacket handler(ImPacket packet, ChannelContext channelContext) throws Exception {
        ProcessorIntf proCmdHandler = this.getProcessor(channelContext);
        if(proCmdHandler == null){
            Aio.remove(channelContext, "没有对应的握手协议处理器HandshakeProCmd...");
            return null;
        }
        HandshakeProcessorIntf handShakeProCmdHandler = (HandshakeProcessorIntf)proCmdHandler;
        ImPacket handShakePacket = handShakeProCmdHandler.handshake(packet, channelContext);
        if (handShakePacket == null) {
            Aio.remove(channelContext, "业务层不同意握手");
        }
        return handShakePacket;
    }

    @Override
    public Command command() {
        return Command.COMMAND_HANDSHAKE_REQ;
    }
}
```

图 32

其实不难理解，这里就是定义各个 cmd 处理器中的业务逻辑处理的，通过它你可以扩展实现自己的命令处理器中的

业务逻辑，还是拿握手环节这个处理器命令来说，我们知道 j-im 是支持多种协议(Http、Websocket、Socket)的,但是各个协议的握手环节及业务逻辑可能都不一样，那我们怎么办？拿 Websocekt 来说，在我们的 jim-server-demo 中，我们是根据我们的业务登录操作是在握手环节校验的，所以 demo 中自己实现了一个 ws 协议握手环节的业务处理器 DemoWsHandshakeProcessor 具体代码如下：

```
public class DemoWsHandshakeProcessor extends WsHandshakeProcessor{

    /**
     * WS握手方法，返回Null则业务层不同意握手，断开连接!
     */
    @Override
    public ImPacket handshake(ImPacket packet, ChannelContext channelContext) throws Exception {
        WsRequestPacket wsRequestPacket = (WsRequestPacket) packet;
        WsSessionContext wsSessionContext = (WsSessionContext) channelContext.getAttribute();
        if (wsRequestPacket.isHandShake()) {
            LoginReqHandler loginHandler = (LoginReqHandler)CommandManager.getCommand(Command.COMMAND_LOGIN_REQ);
            HttpRequest request = wsSessionContext.getHandshakeRequestPacket();
            String username = request.getParams().get("username") == null ? null : (String)request.getParams().get("username")[0];
            String password = request.getParams().get("password") == null ? null : (String)request.getParams().get("password")[0];
            String token = request.getParams().get("token") == null ? null : (String)request.getParams().get("token")[0];
            LoginReqBody loginBody = new LoginReqBody(username,password,token);
            byte[] loginBytes = JsonKit.toJsonBytes(loginBody);
            request.setBody(loginBytes);
            request.setBodyString(new String(loginBytes,HttpConst.CHARSET_NAME));
            Object loginResponse = loginHandler.handler(request, channelContext);
            if(loginResponse == null)
                return null;
            WsResponsePacket wsResponsePacket = new WsResponsePacket();
            wsResponsePacket.setHandShake(true);
            wsResponsePacket.setCommand(Command.COMMAND_HANDSHAKE_RESP);
            wsSessionContext.setHandshaked(true);
            return wsResponsePacket;
        }
        return null;
    }
}
```

图 33 DemoWsHandshakeProcessor 类

这里发现它继承了 WsHandshakeProcessor 类，这是 j-im 默认的 WS 握手环节处理器，它实现了 HandshakeProcessorIntf 接口，这个接口也是自定义的，假如你自己定义的 cmd 命令处理器也需要在其中添加自己的业务逻辑处理，也可以自定义接口但是必须继承 `ProcessorIntf` 接口，它其实就是规定了你这个处理器属于哪种协议的，如果属于三种协议那在接口中的 `isProtocol` 方法直接返回 `true` 即可。看下 `ProcessorIntf` 接口如下：

```
/**
 * 不同协议CMD命令处理接口
 * @author WChao
 *
 */
public interface ProcessorIntf {
    /**
     * 不同协议判断方法
     * @param channelContext
     * @return
     */
    public boolean isProtocol(ChannelContext channelContext);
    /**
     * 该proCmd处理器名称(自定义)
     * @return
     */
    public String name();

}
```

图 34 ProccessorInntf 接口

HandshakeProcessorIntf 接口如下：

```
public interface HandshakeProcessorIntf extends ProcessorIntf{

    public ImPacket handshake(ImPacket packet,ChannelContext channelContext)  throws Exception;
}
```

图 35

j-im 默认 ws 握手环节实现类 WsHandshakeProcessor 代码如下：

```
public class WsHandshakeProcessor implements HandshakeProcessorIntf {

    /**
     * 对httpResponsePacket参数进行补充并返回，如果返回null表示不想和对方建立连接，框架会断开连接，如果返回非null，框架会把这个对象发送给对方
     * @param httpRequestPacket
     * @param httpResponsePacket
     * @param channelContext
     * @return
     * @throws Exception
     * @author: Wchao
     */
    public ImPacket handshake(ImPacket packet, ChannelContext channelContext) throws Exception {
        WsRequestPacket wsRequestPacket = (WsRequestPacket) packet;
        WsSessionContext wsSessionContext = (WsSessionContext) channelContext.getAttribute();
        if (wsRequestPacket.isHandShake()) {
            WsResponsePacket wsResponsePacket = new WsResponsePacket();
            wsResponsePacket.setHandShake(true);
            wsResponsePacket.setCommand(Command.COMMAND_HANDSHAKE_RESP);
            wsSessionContext.setHandshaked(true);
            return wsResponsePacket;
        }
        return null;
    }


    @Override
    public boolean isProtocol(ChannelContext channelContext){
        Object sessionContext = channelContext.getAttribute();
        if(sessionContext == null){
            return false;
        }else if(sessionContext instanceof WsSessionContext){
            return true;
        }
        return false;
    }


    @Override
    public String name() {

        return Protocol.WEBSOCKET;
    }

}
```

图 36 WsHandshakeProcessor 类

而在 jim-server-demo 中我们根据我们自己的业务重写了上面这个类的 handshake 方法所以继承了 WsHandshakeProcessor 类，那我们写完这个 cmd 命令业务逻辑处理器以后怎么将它添加到 cmd 命令处理器中呢？这里我们又用到了上面的 CommandManager 管理器了，我们首先通过它拿到我自己写的 cmd 处理器，然后添加到这个 cmd 处理器即可，代码如下：

```
HandshakeReqHandler handshakeReqHandler = CommandManager.getCommand(Command.COMMAND_HANDSHAKE_REQ,HandshakeReqHandler.class);

handshakeReqHandler.addProcessor(new DemoWsHandshakeProcessor());//添加自定义握手处理器;
```

这样就将我们刚才编写的 cmd 业务处理器添加到了这个 cmd 处理器中了。

作者提供的 jim-server-demo 中也是这么做的，看最开始我们定义的启动类中代码如下：

```
public class ImServerDemoStart {

    public static void main(String[] args)throws Exception{
        PropKit.use("app.properties");
        int port = PropKit.getInt("port");//启动端口
        ImConfig.isStore =  PropKit.get("isStore");//是否开启持久化;(on:开启,off:不开启)
        ImConfig imConfig = new ImConfig(null, port);
        HttpServerInit.init(imConfig);
        //ImgMnService.start();//启动爬虫爬取模拟在线人头像;
        ImServerStarter imServerStarter = new ImServerStarter(imConfig,new ImDemoAioListener());
        /****************start 以下处理器根据业务需要自行添加与扩展，每个Command都可以添加扩展,此处为demo中处理*********************************/
        HandshakeReqHandler handshakeReqHandler = CommandManager.getCommand(Command.COMMAND_HANDSHAKE_REQ,HandshakeReqHandler.class);
        handshakeReqHandler.addProcessor(new DemoWsHandshakeProcessor());//添加自定义握手处理器;
        LoginReqHandler loginReqHandler = CommandManager.getCommand(Command.COMMAND_LOGIN_REQ,LoginReqHandler.class);
        loginReqHandler.addProcessor(new LoginServiceProcessor());//添加登录业务处理器;
        /****************end ********************************************************************************************/
        imServerStarter.start();
    }
}
```

图 37 ImServerDemoStart 类

至此读者应该可以知道我们如何扩展实现自己的 cmd 命令处理器类，以及如何在这个 cmd 处理器类中添加我们自己的业务逻辑处理器了。

6. ServerHandlerManager 类

ServerHandlerManager 类是服务端协议处理器管理中心，因为 j-im 支持多种协议，前面已经说过这里不再多说，假如我们想扩展增加一种协议的支持如何来做？我们拿 j-im 内置的 Websocket 协议 WsServerHandler 支持来说，代码如下：

```
public class WsServerHandler extends AbServerHandler{

    private Logger logger = LoggerFactory.getLogger(WsServerHandler.class);

    private WsServerConfig wsServerConfig;

    private IWsMsgHandler wsMsgHandler;

    public WsServerHandler() {}

    public WsServerHandler(WsServerConfig wsServerConfig, IWsMsgHandler wsMsgHandler) {
    public void init(ImConfig imConfig) {

    public boolean isProtocol(ByteBuffer buffer,ChannelContext channelContext)throws Throwable{

    public ByteBuffer encode(Packet packet, GroupContext groupContext, ChannelContext channelContext) {

    public void handler(Packet packet, ChannelContext channelContext) throws Exception {

    public ImPacket decode(ByteBuffer buffer, ChannelContext channelContext) throws AioDecodeException {
    public WsServerConfig getWsServerConfig() {

    public void setWsServerConfig(WsServerConfig wsServerConfig) {

    public IWsMsgHandler getWsMsgHandler() {

    public void setWsMsgHandler(IWsMsgHandler wsMsgHandler) {

    public String name() {

}
```

图 38 WsServerHandler 类

只需要继承 AbServerHandler 这个抽象类，实现里面的抽象方法就可以了，然后通过 ServerHandlerManager 管理器 ServerHandlerManager.addServerHandler(serverHandler)，添加上就可以扩展协议支持了,移除也一样，假如我们现在只想让 j-im 支持普通 Socket 协议，而不需要 Http、WebSocket 协议、或其他协议支持，那我们通过管理器移除即可。ServerHandlerManager.removeServerHandler(name)，其中 name 是要移除的协议名称。

7. AbstractChatProcessor 类

如果用户想获取所有聊天数据进行业务存储及处理的话，只需要继承这个抽象类（AbstractChatProcessor），实现其中的抽象方法 doHandler 方法，在这里就可以拿到所有聊天数据，抽象类代码如下：

```java
public abstract class AbstractChatProcessor implements ChatProcessorIntf,Const {

    public static final String BASE_CHAT_PROCESSOR = "base_chat_processor";
    private IMesssageHelper messsageHelper = ImConfig.getMessageHelper();

    public abstract void doHandler(ChatBody chatBody, ChannelContext channelContext);
    @Override
    public boolean isProtocol(ChannelContext channelContext) {
        return true;
    }
    public String name() {

    public void handler(ImPacket chatPacket, ChannelContext channelContext) throws Exception {
    /**
     * 推送持久化群组消息
     * @param pushTable
     * @param storeTable
     * @param group_id
     */
    private void pushGroupMessages(String pushTable, String storeTable , ChatBody chatBody){

    private void writeMessage(String timelineTable , String timelineId , ChatBody chatBody){
        messsageHelper.writeMessage(timelineTable, timelineId, chatBody);
    }
}
```

图 39

比如默认的实现方法如下：

```java
public class DefaultChatProcessor extends AbstractChatProcessor{

    Logger log = LoggerFactory.getLogger(DefaultChatProcessor.class);

    @Override
    public void doHandler(ChatBody chatBody, ChannelContext channelContext){
        log.info("聊天消息处理...");
        log.info("入业务库等...");
    }
}
```

图 40

至此，基本读者应该可以基于 j-im 来开发自己的 IM 服务器了，目前基于 j-im 的案例已经很多，后续官网 http://www.j-im.cn 上线以后，会统计展示所有在线案例，后续 j-im 会持续升级，开发文档也会随着不断更新完善，请持续关注！谢谢。。

J-IM 官方群