# 基于Netty的Android系统IM简单实现原理

最近在开发MobIM，实现了消息传输和群等功能的IM功能。SDK功能包小，而功能全面。可以与原来的系统进行无缝整合。

自己抽空也实现了一套IM Server和IMClient的业务通信模式。没有实现复杂的UI界面，实现简单的登录注册，发消息，收消息。服务器端与客户端都使用Netty通信。

Netty基于非阻塞（nio），事件驱动的网络应用程序框架和工具。

通过Netty面对大规模的并发请求可以处理的得心用手。用来替代原来的bio网络应用请求框架。

BIO通信即平时使用的基于Socket，ServerSocket的InputStream和OutStream。

Netty神奇的地方在于是否是阻塞的。

while(true){

//主线程死循环等待新连接到来

Socket socket = serverSocket.accept();

//为新的连接创建新的线程,客户端与服务器上的线程数1:1

executor.submit(new ConnectIOnHandler(socket));

在BIO模型中，服务器通过ServerSocket来开启监听，每当有请求的时候开启一个线程来接受处理和维持状态。这种思想在低并发，小吞吐的应用还可以应付，一旦遇到大并发，大吞吐的请求，必然歇菜。线程和客户端保持着1:1的对应关系，维持着线程。维持那么的多的线程，JVM必然不堪重负，服务器必然崩溃，宕机。

而在非阻塞的Netty中，却可以应付自如。从容应对。Tomcat就是基于BIO的网络通信模式（Tomcat可以通过一定配置，改成非阻塞模式），而JBoss却是基于非阻塞的NIO实现。

NIO的网络通信模式很强劲，但是上手却一点都不容易。其中解决和牵扯到好多网络问题。如：网络延时，TCP的粘包/拆包，网络故障等一堆一堆的问题。而Netty呢，针对nio复杂的编程难题而进行一系列的封装实现，提供给广大开发者一套开源简单，方便使用的API类库，甚至青出于蓝而胜于蓝，甚至几乎完美的解决CPU突然飙升到100%的bug ：http://bugs.sun.com/bugdatabase/view\_bug.do?bug\_id=6403933 （其实也没有真正的解决，只是把复现的概率降到了最低而已）。

用Netty来实现IM实在太合适了。可以在最短的时间里整出一套思路清晰，架构简明的IM通信底层模型。提下需求，底层用JSON 字符串String进行通信，对象通过JSON序列化成JSON String。收到JSON数据后再反序列化成对象。

首先，我们先看服务器是怎么实现的。

**private** **static** **final** StringDecoder ***DECODER*** = **new** StringDecoder();

**private** **static** **final** StringEncoder ***ENCODER*** = **new** StringEncoder();

...

//boss线程监听端口，worker线程负责数据读写

bossGroup = **new** NioEventLoopGroup(1);

workerGroup = **new** NioEventLoopGroup();

//辅助启动类

ServerBootstrap bootstrap = **new** ServerBootstrap();

**try** {

//设置线程池

bootstrap.group(bossGroup, workerGroup);

//设置socket工厂

bootstrap.channel(NioServerSocketChannel.**class**);

bootstrap.handler(**new** LoggingHandler(LogLevel.***INFO***));

//设置管道工厂

bootstrap.childHandler(**new** ChannelInitializer<SocketChannel>() {

@Override

**protected** **void** initChannel(SocketChannel socketChannel) **throws** Exception {

//获取管道

ChannelPipeline pipe = socketChannel.pipeline();

// Add the text line codec combination first,

pipe.addLast(**new** DelimiterBasedFrameDecoder(8192, Delimiters.*lineDelimiter*()));

// the encoder and decoder are static as these are sharable

//字符串编码器

pipe.addLast(***DECODER***);

//字符串解码器

pipe.addLast(***ENCODER***);

//业务处理类

pipe.addLast(**new** IMServerHandle());

}

});

//绑定端口

// Bind and start to accept incoming connections.

ChannelFuture f = bootstrap.bind(port).sync();

**if** (f.isSuccess()) {

Log.*debug*("server start success... port: " + port + ", main work thread: "

+ Thread.*currentThread*().getId());

}

////等待服务端监听端口关闭

// Wait until the server socket is closed.

f.channel().closeFuture().sync();

} **finally** {

//优雅退出，释放线程池资源

bossGroup.shutdownGracefully();

workerGroup.shutdownGracefully();

}

以上是Netty服务器启动的代码。其中需要注意childHandler方法。需要把我们要添加的业务处理handler来添加到这里。通过ChannelPipeline 添加ChannelHandler。而处理字符串的就在IMServerHandle里实现。IMServerHandle继承了SimpleChannelInboundHandler<T>类。其中泛型T就是要转换成的对象。客户端与服务器端通信是本质上通过字节码byte[]通信的，而通过StringDecoder 和StringEncoder工具类对byte[]进行转换，在IMServerHandle中获取到String进行处理即可。

看下IMServerHandle的实现方式。

/\*\*\*

\* 面向IM通信操作的业务类

\* **@author** xhj

\*

\*/

**public** **class** IMServerHandle **extends** SimpleChannelInboundHandler<String> {

/\*\*

\* user操作业务类

\*/

**private** UserBiz userBiz = **new** UserBiz();

/\*\*\*

\* 消息操作的业务类

\*/

**private** IMMessageBiz immessagebiz = **new** IMMessageBiz();

/\*\*\*

\* 处理接受到的String类型的JSON数据

\*/

@Override

**protected** **void** channelRead0(ChannelHandlerContext ctx, String msg) **throws** Exception {

System.***out***.println(" get msg >> "+msg);

//把JSON数据进行反序列化

Request req = JSON.*parseObject*(msg, Request.**class**);

Response respon = **new** Response();

respon.setSendTime(System.*currentTimeMillis*());

//判断是否是合法的请求

**if**(req != **null** ) {

System.***out***.println("the req method >> "+req.getMethod());

//获取操作类型

**if**(req.getMethod() == IMProtocol.***LOGIN***) {

//获取要操作的对象

User user = JSON.*parseObject*(req.getBody(),User.**class**);

//设置返回数据的操作类型

respon.setMethod(IMProtocol.***LOGIN***);

//执行业务操作

**boolean** bl = userBiz.login(user);

**if**(bl) {//检验用户有效

//设置响应数据

respon.setBody("login ok");

//设置状态

respon.setStatus(0);

//登录成功将连接channel保存到Groups里

ChannelGroups.*add*(ctx.channel());

//将用户的uname和channelId进行绑定，服务器向指定用户发送消息的时候需要用到uname和channelId

ChannelGroups.*putUser*(user.getUname(), ctx.channel().id());

//发送广播通知某人登录成功了

userBiz.freshUserLoginStatus(user);

} **else** {//用户密码错误

//设置错误描述

respon.setErrorStr("pwd-error");

//设置状态描述码

respon.setStatus(-1);

}

//将Response序列化为json字符串

msg = JSON.*toJSONString*(respon);

//发送josn字符串数据，注意后面一定要加"\r\n"

ctx.writeAndFlush(msg+"\r\n");

} **else** **if**(req.getMethod() == IMProtocol.***SEND***) {

IMMessage immsg = JSON.*parseObject*(req.getBody(), IMMessage.**class**);

immsg.setSendTime(System.*currentTimeMillis*()); c

通过IMServerHandle可以十分方便的处理获取到的String字符串。处理完后，可以直接通过ChannelHandlerContext的writeAndFlush方法发送数据。

再看下Netty客户端如何实现。

**private** BlockingQueue<Request> **requests** = **new** LinkedBlockingQueue<>();  
  
 */\*\*  
 \* String字符串解码器  
 \*/***private static final** StringDecoder ***DECODER*** = **new** StringDecoder();  
  
 */\*\*\*  
 \* String字符串编码器  
 \*/***private static final** StringEncoder ***ENCODER*** = **new** StringEncoder();  
  
 */\*\*  
 \* 客户端业务处理Handler  
 \*/* **private** IMClientHandler **clientHandler** ;  
  
 */\*\*  
 \* 添加发送请求Request  
 \** ***@param request*** *\*/* **public void** addRequest(Request request) {  
 **try** {  
 **requests**.put(request);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 是否继续进行运行  
 \*/* **private boolean run** = **true**;  
  
 **public void** run() {  
 *//远程IP* String host = **"172.20.10.7"**;  
 *//端口号* **int** port = 10000;  
 *//工作线程* EventLoopGroup workerGroup = **new** NioEventLoopGroup();  
 **try** {  
 *//辅助启动类* Bootstrap b = **new** Bootstrap(); *// (1)  
 //设置线程池* b.group(workerGroup); *// (2)  
 //设置socket工厂 不是ServerSocket而是Socket* b.channel(NioSocketChannel.**class**); *// (3)* b.handler(**new** LoggingHandler(LogLevel.***INFO***));  
 *//设置管道工厂* b.handler(**new** ChannelInitializer<SocketChannel>() {  
 **public void** initChannel(SocketChannel ch) **throws** Exception {  
 ChannelPipeline pipe = ch.pipeline();  
 *// Add the text line codec combination first,* pipe.addLast(**new** DelimiterBasedFrameDecoder(8192, Delimiters.*lineDelimiter*()));  
 *// the encoder and decoder are static as these are sharable  
 //字符串解码器* pipe.addLast(***DECODER***);  
 *//字符串编码器* pipe.addLast(***ENCODER***);  
 **clientHandler** = **new** IMClientHandler();  
 *//IM业务处理类* pipe.addLast(**clientHandler**);  
 }  
 });  
  
 *// Start the client.* ChannelFuture f = b.connect(host, port).sync(); *// (5)* Channel ch = f.channel();  
 ChannelFuture lastWriteFuture = **null**;  
 **while**(**run**) {  
 *//将要发送的Request转化为JSON String类型* String line = JSON.*toJSONString*(**requests**.take());  
 **if**(line != **null** && line.length() > 0) {*//判断非空  
 // Sends the received line to the server.  
 //发送数据到服务器* lastWriteFuture = ch.writeAndFlush(line + **"\r\n"**);  
 }  
 }  
 *// Wait until all messages are flushed before closing the channel.  
 //关闭写的端口* **if** (lastWriteFuture != **null**) {  
 lastWriteFuture.sync();  
 }  
 } **catch**(Exception ex){  
 ex.printStackTrace();  
 } **finally** {  
 *//优雅的关闭工作线程* workerGroup.shutdownGracefully();  
 }  
 }  
  
 */\*\*  
 \* 增加消息监听接受接口  
 \** ***@param messgeReceivedListener*** *\*/* **public void** addMessgeReceivedListener(MessageSender.MessgeReceivedListener messgeReceivedListener) {  
 **clientHandler**.addMessgeReceivedListener(messgeReceivedListener);  
 }  
  
 */\*\*\*  
 \* 移除消息监听接口  
 \** ***@param messgeReceivedListener*** *\*/* **public void** remove(MessageSender.MessgeReceivedListener messgeReceivedListener) {  
 **clientHandler**.remove(messgeReceivedListener);  
 }

Netty的client端实现和Server实现方式大同小异。比Server端要简要些了。少一个NIOEventLoop。在Bootstrap 的handle方法中增加ChannelInitializer初始化监听器，并增加了IMClientHandler的监听操作。其中IMClientHandler具体处理服务器返回的通信信息。

通过ChannelFuture 获取Channel，通过Channel在一个循环里发送请求。如果消息队列BlockingQueue非空的时候，获取Request并发送。以上发送，如何接受数据呢？接受到的json被反序列化直接变成了对象Response，对Response进行处理即可。

定义了一个消息接受到的监听接口。

**public static interface** MessgeReceivedListener {  
 **public void** onMessageReceived(Response msg);  
 **public void** onMessageDisconnect();  
 **public void** onMessageConnect();  
}

在接口onMessageReceived方法里直接对获取成功的响应进行处理。

而服务器端对某个客户端进行发送操作，把Channel添加到ChannelGroup里，将uname和channelid对应起来。需要对某个用户发送消息的时候通过uname获取channelid，通过channelid从ChannelGroup里获取channel，通过channel发送即可。

具体操作如下：

**public** **void** transformMessage(IMMessage message) {

Channel channel = ChannelGroups.*getChannel*(ChannelGroups.*getChannelId*(message.getTo()));

**if**(channel != **null** && channel.isActive()) {

Response response = **new** Response();

response.setBody(JSON.*toJSONString*(message));

response.setStatus(0);

response.setMethod(IMProtocol.***REV***);

response.setSendTime(System.*currentTimeMillis*());

channel.writeAndFlush(JSON.*toJSON*(response)+"\r\n");

}

}

ChannelGroups的代码实现：

**public** **class** ChannelGroups {

**private** **static** **final** Map<String,ChannelId> ***userList*** = **new** ConcurrentHashMap();

**private** **static** **final** ChannelGroup ***CHANNEL\_GROUP*** = **new** DefaultChannelGroup("ChannelGroups",

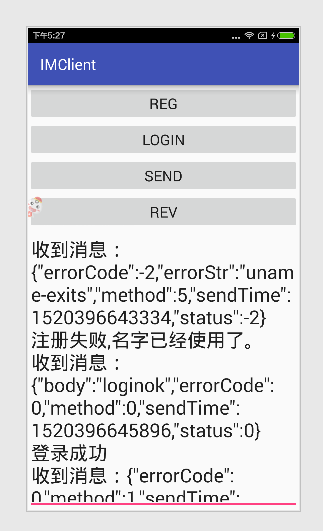
GlobalEventExecutor.***INSTANCE***);

**public** **static** **void** putUser(String uname,ChannelId id) {

***userList***.put(uname,id);

}

通过以上代码解析应该对IM的通信模式有了比较全面的认识。具体实现过程可以下载源代码进行查看。欢迎大家反馈提出问题。



运行效果图。