目录

[一、背景 3](#_Toc485392895)

[二、目的与要求： 3](#_Toc485392896)

[三、开发环境 3](#_Toc485392897)

[四、框架介绍 4](#_Toc485392898)

[1. Netty 5.X 框架 4](#_Toc485392899)

[2. WebSocket 协议 4](#_Toc485392900)

[3. Bootstrap 框架 5](#_Toc485392901)

[五、功能流程图 6](#_Toc485392902)

[六、功能分析 7](#_Toc485392903)

[七、项目技术分析 9](#_Toc485392904)

[1. 握手和安全认证 9](#_Toc485392905)

[2. 心跳检测机制 9](#_Toc485392906)

[3. WebSocket协议整合 11](#_Toc485392907)

[4. 线程安全使用和并发控制 16](#_Toc485392908)

[八、私有协议开发 20](#_Toc485392909)

[1 私有协议定义 20](#_Toc485392910)

[2 私有协议属性 20](#_Toc485392911)

[九、总结 21](#_Toc485392912)

[十、参考资料 21](#_Toc485392913)

# 一、背景

伴随着Internet的发展与宽带技术的普及，人们可以通过Internet交换动态数据，展示新产品，与人进行沟通并进行电子商务贸易。作为构成网站的重要组成部分，留言管理系统为人们的交流提供了一个崭新的平台。同时，聊天室作为一个新型的Web应用程序，为互联网用户提供了一个实时信息交流的场所。

聊天室在早期的网络时代已经非常流行，例如BBS、IRC这些类似的机制。它为互联网用户提供了实时对话的功能，并因此成为了非常流行的网络服务。网络会议和网上聊天均可以通过聊天室来实现。聊天室为互联网用户提供了一个更好的交友环境，这种交友形式类似于互联网化的笔友，但是大大节省了信件传送时间。对于网站留言管理而言，目前非常受欢迎的做法是基于JAVA WEB和脚本语言，并结合动态网页和数据库，然后通过应用程序来处理信息。

网络聊天系统利用了现代的网络资源和技术，为人们的交流和联系提供了一个平台，用以加快信息化建设，促进人和人之间的交流和沟通。Internet存在于全球范围，它将世界各地大小的网络连接成了一个整体，万维网目前已经成为了世界上最大的信息资源宝库，它是一种更容易被人们接受的信息检索方式。根据估算，目前在Internet上存在数以万计的网站，内容包括文化、金融、教育科研、新闻出版、商业、娱乐等。它的用户群是非常庞大的，所以建立一个好的网站非常重要。

以前旧的联系方法已经不能满足现代人的生活。网上聊天系统因其方便的沟通方式而成为了重要且实用的计算机应用程序。系统管理者通过提供完整的网上聊天系统管理，来促进人们之间相互沟通与交流。

实时显示聊天者的谈话内容是聊天室最重要的特点之一。所谓的实时性与常的留言板和讨论区有很大的不同，它是指同一个聊天室内的用户可以在很短的时间内立即看到其他用户的留言。随着计算机技术的快速发展，现在可以使用Java Web+HTML方便快速地开发出一个典型的聊天室程序。但是还需要花费更多的心思，获得更强大的聊天功能来吸引更多的网络用户。

# 二、目的与要求：

本程序实现一个基于Web的多人聊天室程序，访客可以自由加入聊天室，并设定自己的昵称。

开发要点：采用浏览器端和服务器端(B/S)的开发技术。利用浏览器解析HTML语言达到即时聊天作用，无需考虑操作系统环境等外部因素。服务器开发使用JAVA面向对象的开发方法进行开发与设计，通过采用高性能的Netty框架+WebSocket协议搭建即时聊天服务器，可以支持起高并发稳定交互。

# 三、开发环境

**软件：**

操作系统：Windows 10

Java开发IDE：Intellij IDEA 2016.2.4

HTML/JS/CSS开发IDE：Sublime Text 3

测试浏览器：Google Chrome 版本 58.0.3029.110 (64-bit)

测试服务器：Tomcat 8.0.22

**硬件：**

处理器：Intel® Core(TM) i5-5200U CPU @ 2.20GHz 2.19GHz

控制器：Intel® 9 Series Chipset Family SATA AHCI Controller

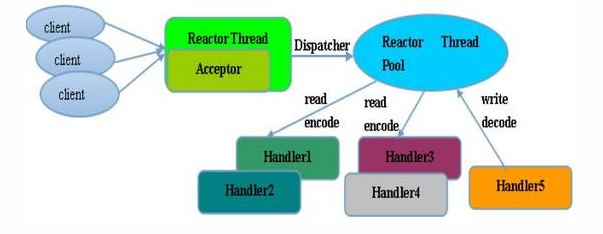
内存(RAM)：8.00 G

# 四、框架介绍

## 1. Netty 5.X 框架

Netty是一个高性能、异步事件驱动的NIO框架，它提供了对TCP、UDP和文件传输的支持，作为一个异步NIO框架，Netty的所有IO操作都是异步非阻塞的，通过Future-Listener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。作为当前最流行的NIO框架，Netty在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，一些业界著名的开源组件也基于Netty的NIO框架构建。

Netty，为了尽可能提升性能，Netty采用了串行无锁化设计，在I/O线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。——摘取自《Netty高性能之道》



Netty多线程模式

## 2. WebSocket 协议

Websocket是html5提出的一个协议规范，是为解决客户端与服务端实时通信而产生的技术。websocket协议本质上是一个基于tcp的协议，是先通过HTTP/HTTPS协议发起一条特殊的http请求进行握手后创建一个用于交换数据的TCP连接，此后服务端与客户端通过此TCP连接进行实时通信。

WebSocket API最伟大之处在于服务器和客户端可以在给定的时间范围内的任意时刻，相互推送信息。 浏览器和服务器只需要要做一个握手的动作，在建立连接之后，服务器可以主动传送数据给客户端，客户端也可随时向服务器发送数据。

**Client**

**(浏览器)**

**WebSocket**

**服务器**

HTTP Request

HTTP Reponse

TCP

TCP

TCP

## 3. Bootstrap 框架

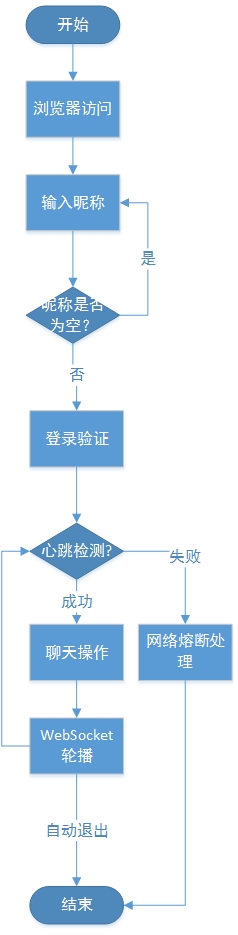
Bootstrap，来自 Twitter，是目前最受欢迎的前端框架。Bootstrap 是基于 HTML、CSS、JAVASCRIPT 的，它简洁灵活，使得 Web 开发更加快捷。

Bootstrap框架的优点:

1. **移动设备优先**：自 Bootstrap 3 起，框架包含了贯穿于整个库的移动设备优先的样式。
2. **浏览器支持**：所有的主流浏览器都支持 Bootstrap。
3. **容易上手**：只要您具备 HTML 和 CSS 的基础知识，您就可以开始学习 Bootstrap。
4. **响应式设计**：Bootstrap 的响应式 CSS 能够自适应于台式机、平板电脑和手机。
5. **其他优点**：为开发人员创建接口提供了一个简洁统一的解决方案；包含了功能强大的内置组件，易于定制；还提供了基于 Web 的定制，并且是开源。

# 五、功能流程图

功能流程：



# 六、功能分析

1. **支持昵称登录**

用户通过浏览器访问服务器时，需要确定自己的昵称，便于交流。



其次调用JS脚本代码来检查用户输入是否为空，代码如下：

function userLogin() {

if (!userNick) {

userNick = $('#nick').val().trim();

}

if (userNick) {

//其他逻辑事务处理

…………………………

} else {

$('#tipMsg').text("连接没有成功，请重新登录");

$('#tipModal').modal('show');

}

若游客昵称验证成功，则进行是否支持WebSocket判断：

if (window.WebSocket) {

window.socket = new WebSocket("ws://localhost:9090/websocket");

window.socket.onmessage = function (event) {

};

window.socket.onclose = function (event) {

console.log("connection close!!!");

closeInvake(event);

};

window.socket.onopen = function (event) {

console.log("connection success!!");

openInvake(event);

};

} else {

alert("您的浏览器不支持WebSocket！！！");

}

1. 支持多人同时在线
2. 同步显示在线人数和成员列表
3. 支持文字和表情的内容
4. 浏览器与服务器保持长连接，定时心跳检测

# 七、项目技术分析

## 1. 握手和安全认证

系统在浏览器和服务器TCP链路建立成功通道激活时，握手信息会由浏览器自动发起构造空消息体的协议内容(详情看“私有协议开发”)。握手请求发送后，按照协议规范，服务端需要返回握手应答消息。

switch (data.extend.code) {

case 20001: // user count

//业务处理

break;

case 20002: // auth

console.log("auth result: " + data.extend.mess);

isAuth = data.extend.mess;

if (isAuth) { //认证成功

$("#menuModal").modal('hide');

$('#chatWin').show();

$('#content').append('欢迎来到聊天室！！<hr/>');

}

## 2. 心跳检测机制

1. 在握手成功后，有服务器端轮询机制发送心跳消息，浏览器收到心跳消息之后，返回应答消息。由于心跳消息的目的是为了检测链路的可用性，因此不需要携带消息体。
2. 服务器端轮询机制：

//定时任务:扫描所有的Channel，关闭失效的Channel

executorService.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

logger.info("scanNotActiveChannel --------");

UserInfoManager.scanNotActiveChannel();

}

}, 3, 60, TimeUnit.SECONDS);

//定时任务:向所有客户端发送Ping消息

executorService.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

UserInfoManager.broadCastPing();

}

}, 3, 50, TimeUnit.SECONDS);

1. 服务器端调用扫描方法scanNotActiveChannel()确保每个TCP链路的激活：

public static void scanNotActiveChannel() {

Set<Channel> keySet = userInfos.keySet();

for (Channel channel : keySet) {

UserInfo userInfo = userInfos.get(channel);

if (userInfo == null) continue;

if (!channel.isOpen() || !channel.isActive()

|| (!userInfo.isAuth() &&

(System.currentTimeMillis() - userInfo.getTime()) > 10000)) {

removeChannel(channel);

}

}

}

1. 服务器端调用广播方法broadCastPing()向每个连接发生Ping消息：

try {

rwLock.readLock().lock(); //加锁

Set<Channel> keySet = userInfos.keySet();

for (Channel channel : keySet) {

UserInfo userInfo = userInfos.get(channel);

if (userInfo == null || !userInfo.isAuth()) continue;

channel.writeAndFlush(

new TextWebSocketFrame(ChatProto.buildPingProto()));

}

} finally {

rwLock.readLock().unlock(); //解锁

}

1. 浏览器在WebSocket连接处理器收到Ping消息后，调用chat.js文件中的pingInvake()返回Pong消息给服务器：

//浏览器处理ping消息

function pingInvake(data) {

//发送pong消息响应

send(isAuth, "{'code':10012}");

};

1. 服务器收到来自浏览器的Pong消息后，调用updateUserTime()方法来更新标记和用户登录时间：

public static void updateUserTime(Channel channel) {

UserInfo userInfo = getUserInfo(channel);

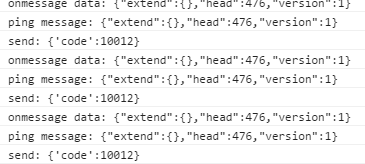
if (userInfo != null) {

userInfo.setTime(System.currentTimeMillis());

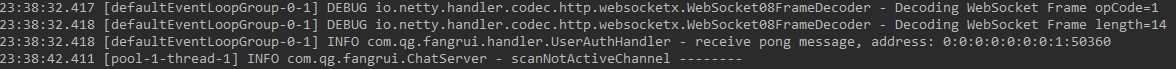
}

}

浏览器端收到Ping消息请求，并发起Pong消息应答：



服务器收到Pong消息后更改标志：



## 3. WebSocket协议整合

Netty基于HTTP协议栈开发了WebSocket协议栈，利用Netty的WebSocket协议栈可以很方便地开发出WebSocket客户端和服务器端。

1. **WebSocket服务器端开发**(以UserAuthHandler类为例)：

public class UserAuthHandler extends SimpleChannelInboundHandler<Object> {

private static final Logger logger = LoggerFactory.getLogger(UserAuthHandler.class);

private WebSocketServerHandshaker handshaker;

@Override

protected void messageReceived(ChannelHandlerContext ctx, Object o) throws Exception {

//传统HTTP接入

if (o instanceof FullHttpRequest) {

handleHttpRequest(ctx, (FullHttpRequest) o);

}

//WebSocket接入

else if (o instanceof WebSocketFrame) {

handleWebSocket(ctx, (WebSocketFrame) o);

}

}

/\*\*

\* 重写心跳检测机制

\* @param ctx

\* @param evt

\* @throws Exception

\*/

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

//判断evt事件是不是IdleStateEvent事件

if (evt instanceof IdleStateEvent) {

IdleStateEvent evnet = (IdleStateEvent) evt;

//判断是读空闲事件还是写空闲事件还是读写空闲事件

if (evnet.state().equals(IdleState.READER\_IDLE)) {

//读操作发起操作

final String remoteAddress = NettyUtil.parseChannelRemoteAddr(ctx.channel());

logger.warn("NETTY SERVER PIPELINE: IDLE exception [{}]", remoteAddress);

//移除用户并更新数量

UserInfo userInfo = UserInfoManager.getUserInfo(ctx.channel());

UserInfoManager.removeChannel(ctx.channel());

UserInfoManager.broadCastInfo(Constants.SYSTEM\_USER\_COUNT,UserInfoManager.getAuthUserCount());

UserInfoManager.broadCastInfo(Constants.SYSTEM\_USER\_LIST, UserInfoManager.getUserInfoList());

if (null != userInfo){

UserInfoManager.broadCastInfo(Constants.SYSTEM\_OTHER\_INFO, "心跳检测发生异常，用户 "+userInfo.getNick()+"("

+userInfo.getUserId()+") 已经强制下线!<hr/>");

}

}

}

ctx.fireUserEventTriggered(evt);

}

private void handleHttpRequest(ChannelHandlerContext ctx, FullHttpRequest request) {

//对URL进行判断，如果HTTP解码失败，返回HTTP异常

if(!request.decoderResult().isSuccess() || !"websocket".equals(request.headers().get("Upgrade"))){

logger.warn("protobuf don't support websocket");

ctx.channel().close();

return;

}

WebSocketServerHandshakerFactory handshakerFactory = new WebSocketServerHandshakerFactory(

Constants.WEBSOCKET\_URL,null, false);

handshaker = handshakerFactory.newHandshaker(request);

if (handshaker == null){

WebSocketServerHandshakerFactory.sendUnsupportedVersionResponse(ctx.channel());

} else {

// 动态加入websocket的编解码处理

handshaker.handshake(ctx.channel(), request);

UserInfo userInfo = new UserInfo();

userInfo.setAddr(NettyUtil.parseChannelRemoteAddr(ctx.channel()));

// 存储已经连接的Channel

UserInfoManager.addChannel(ctx.channel());

}

}

private void handleWebSocket(ChannelHandlerContext ctx, WebSocketFrame frame) {

// 判断是否关闭链路命令

if (frame instanceof CloseWebSocketFrame) {

handshaker.close(ctx.channel(), (CloseWebSocketFrame) frame.retain());

UserInfo userInfo = UserInfoManager.getUserInfo(ctx.channel());

System.out.println(userInfo.toString());

UserInfoManager.removeChannel(ctx.channel());

if (null != userInfo) {

UserInfoManager.broadCastInfo(Constants.SYSTEM\_OTHER\_INFO, "用户 "+userInfo.getNick()+"("

+userInfo.getUserId()+") 退出聊天室!<hr/>");

}

return;

}

// 判断是否Ping消息

if (frame instanceof PingWebSocketFrame) {

logger.info("ping message:{}", frame.content().retain());

ctx.writeAndFlush(new PongWebSocketFrame(frame.content().retain()));

return;

}

// 判断是否Pong消息

if (frame instanceof PongWebSocketFrame) {

logger.info("pong message:{}", frame.content().retain());

ctx.writeAndFlush(new PongWebSocketFrame(frame.content().retain()));

return;

}

//非文本(二进制)信息

if (!(frame instanceof TextWebSocketFrame)) {

throw new UnsupportedOperationException(frame.getClass().getName() + " frame type not supported");

}

String message = ((TextWebSocketFrame) frame).text();

JSONObject json = JSONObject.parseObject(message);

int code = json.getInteger("code");

Channel channel = ctx.channel();

switch (code){

case Constants.PING\_CODE:

case Constants.PONG\_CODE:{

UserInfoManager.updateUserTime(channel);

logger.info("receive pong message, address: {}",NettyUtil.parseChannelRemoteAddr(channel));

return;

}

case Constants.AUTH\_CODe:{

boolean isSuccess = UserInfoManager.saveUser(channel, json.getString("nick"));

UserInfoManager.sendInfo(channel,Constants.SYSTEM\_AUTH\_STATE,isSuccess);

if (isSuccess) {

UserInfo userInfo = UserInfoManager.getUserInfo(channel);

//更新在线人数

UserInfoManager.broadCastInfo(Constants.SYSTEM\_USER\_COUNT,UserInfoManager.getAuthUserCount());

UserInfoManager.broadCastInfo(Constants.SYSTEM\_USER\_LIST, UserInfoManager.getUserInfoList());

if (null != userInfo) {

//增加人数

UserInfoManager.broadCastInfo(Constants.SYSTEM\_OTHER\_INFO, "用户 "+ userInfo.getNick() + "(" + userInfo.getUserId()

+ ") 进入网络聊天室，大家热烈欢迎~<hr/>");

}

}

return;

}

case Constants.MESS\_CODE:

//普通的消息留给MessageHandler处理

break;

default:

logger.warn("The code [{}] can't be auth!!!", code);

return;

}

//后续消息交给MessageHandler处理

ctx.fireChannelRead(frame.retain());

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception{

logger.warn("NETTY SERVER PIPELINE: Unknown exception [{}]", cause.getMessage());

UserInfoManager.removeChannel(ctx.channel());

UserInfoManager.broadCastInfo(Constants.SYSTEM\_USER\_COUNT, UserInfoManager.getAuthUserCount());

UserInfoManager.broadCastInfo(Constants.SYSTEM\_USER\_LIST, UserInfoManager.getUserInfoList());

UserInfoManager.broadCastInfo(Constants.SYSTEM\_OTHER\_INFO, "网络发生未知错误，与部分用户断开连接，请确保网络正常!<hr/>");

}

}

服务器端代码解析：WebSocket第一次握手请求消息由HTTP协议承载，所以他是一个HTTP消息，执行handleHttpRequest方法来处理WebSocket握手请求。当对握手请求消息进行判断，如果消息头中没有包含Upgrade字段或者它的值不是websocket(协议定义)，则返回HTTP400响应。

握手请求简单校验通过之后，开始构造握手工厂，创建握手处理类WebSocketServerHandshaker，通过它构造握手响应消息返回给客户端，同时将WebSocket相关的编码和解码类动态添加到ChannelPipeline中，用于WebSocket消息的编码与解码。

添加了WebSocketEncoder和WebSocketDecoder之后，服务器就可以自动对于WebSocket消息进行编码解码，后面的业务handler可以直接对WebSocket对象进行操作。这样子，浏览器端与服务器端的TCP连接通道就可以建立，一旦浏览器发起对于服务器WebSocket接口的请求，就可以自动激活通道进行通信。

1. **WebSocket客户端开发：**

if (window.WebSocket) {

window.socket = new WebSocket("ws://localhost:9090/websocket");

window.socket.onmessage = function (event) {

var data = eval("(" + event.data + ")");

console.log("onmessage data: " + JSON.stringify(data));

switch (data.head) {

// 私有协议栈开发内容

}

};

window.socket.onclose = function (event) {

// WebSocket断开触发的方法

console.log("connection close!!!");

closeInvake(event);

};

window.socket.onopen = function (event) {

// WebSocket开启时触发的方法

console.log("connection success!!");

openInvake(event);

};

关于WebSocket API的调用较为简单。创建一个Socket实例，参数为URL，ws表示WebSocket协议。onopen、onclose和onmessage方法把事件连接到Socket实例上。每个方法都提供了一个事件，以表示Socket的状态。

onmessage事件提供了一个data属性，它可以包含消息的Body部分。消息的Body部分必须是一个字符串，可以进行序列化/反序列化操作，以便传递更多的数据。

WebSocket对于大多数客户机-服务器的异步通信是理想的，在浏览器内聊天是最突出的应用。WebSocket由于其高效率，被广泛应用。

## 4. 线程安全使用和并发控制

为了实现高并发下的线程安全，该项目中应用了许多线程安全的数据结构与高并发控制。

1. **线程安全的自增/自减类 AtomicInteger**

AtomicInteger，一个提供原子操作的Integer的类。在Java语言中，++i和i++操作并不是线程安全的，在使用的时候，不可避免的会用到synchronized关键字。而AtomicInteger则通过一种线程安全的加减操作接口。

AtomicInteger提供原子操作来进行Integer的使用，因此十分适合高并发情况下的使用。例如，在构建用户实例时：

//线程安全自动递增，产生UID

private static AtomicInteger uidGener = new AtomicInteger(1000);

通过调用 uidGener 来构造用户唯一标识符(uid)，确保能够识别每个接入服务器的用户。

1. **多线程并发调用调优**

在Java多线程编程中，会在JVM进程结束前调用系统方法线程，通过重写线程方法类来确保进程释放所有资源再结束。

Runtime.getRuntime().addShutdownHook()：在jvm中增加一个关闭的钩子，当jvm关闭的时候，会执行系统中已经设置的所有通过方法addShutdownHook添加的钩子，当系统执行完这些钩子后，jvm才会关闭。所以这些钩子可以在jvm关闭的时候进行内存清理、对象销毁等操作。

// 注册进程钩子，在JVM进程关闭前释放资源

Runtime.getRuntime().addShutdownHook(new Thread(){

@Override

public void run(){

server.shutdown();

logger.warn(">>>>>>>>>> jvm shutdown");

System.exit(0);

}

});

1. **优雅退出线程**

在多线程编程中，很经常开启/结束线程，对于结束线程时，除了调用JVM强制性的清理方法外，经常需要重写线程退出方法，以降低对于系统资源的消耗。

以ChatServer类为例：

public void shutdown() {

if (executorService != null) {

executorService.shutdown();

}

super.shutdown();

}

1. **线程池调度与轮询调度**

Java通过Executors提供四种线程池，这里主要调用了newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

线程池的主要作用就是限制系统中执行线程的数量。根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排 队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池 中有等待的工作线程，就可以开始运行了；否则进入等待队列。通过调用线程池减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。同时可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

在初始化服务器实例时，懒加载线程池：

public ChatServer(int port){

this.port = port;

//创建一个定长线程池，支持定时及周期性任务执行

executorService = Executors.newScheduledThreadPool(2);

}

通过线程池的定时及周期性任务调度，执行项目系统的轮询机制：

//定时任务:扫描所有的Channel，关闭失效的Channel

executorService.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

logger.info("scanNotActiveChannel --------");

UserInfoManager.scanNotActiveChannel();

}

}, 3, 60, TimeUnit.SECONDS);

//定时任务:向所有客户端发送Ping消息

executorService.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

UserInfoManager.broadCastPing();

}

}, 3, 50, TimeUnit.SECONDS);

轮询机制主要有两个，一个是每隔60s轮询所有Channel连接，确保网络链路连通；同时，每隔50s调度一次心跳检测，确保线程连接的活跃。

1. **读写锁的调用**

为了提高性能，Java提供了读写锁，在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的,在一定程度上提高了程序的执行效率。

Java中读写锁有个接口java.util.concurrent.locks.ReadWriteLock来获得读写锁。但使用Java读写锁也是有条件的：重入方面其内部的WriteLock可以获取ReadLock,但是反过来ReadLock想要获得WriteLock则永远都不要想；WriteLock可以降级为ReadLock,顺序是:先获得WriteLock再获得ReadLock,然后释放WriteLock,这时候线程将保持Readlock的持有.反过来ReadLock想要升级为WriteLock则不可能；ReadLock可以被多个线程持有并且在作用时排斥任何的WriteLock,而WriteLock则是完全的互斥.这一特性最为重要,因为对于高读取频率而相对较低写入的数据结构,使用此类锁同步机制则可以提高并发量；不管是ReadLock还是WriteLock都支持Interrupt；WriteLock支持Condition并且与ReentrantLock语义一致,而ReadLock则不能使用Condition,否则抛出UnsupportedOperationException异常。

例如，在UserInfoManager处理类中启用读写锁：

//设置读写锁

private static ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock(true);

public static void removeChannel(Channel channel){

try {

//加写锁

rwLock.writeLock().lock();

//业务处理

} finally {

rwLock.writeLock().unlock(); //解锁

}

}

1. **安全的数据结构**

项目系统中调用了许多安全的数据结构，比如ConcurrentHashMap类。

concurrentHashmap是为了高并发而实现，内部采用分离锁的设计，有效地避开了热点访问。而对于每个分段，ConcurrentHashmap采用final和内存可见修饰符volatile关键字（内存立即可见：Java 的内存模型可以保证：某个写线程对 value 域的写入马上可以被后续的某个读线程“看”到。注：并不能保证对volatile变量状态有依赖的其他操作的原子性）。

例如，在UserInfoManager处理类中启用安全的数据结构ConcurrentHashMap类。

//构建安全的数据结构

private static ConcurrentMap<Channel, UserInfo> userInfos = new ConcurrentHashMap<Channel, UserInfo>();

# 八、私有协议开发

## 1 私有协议定义

私有的所有的消息都一个Json字符串，格式如下：{head | body | extend}

1. head：作为头部，用int类型储存，长度为4个字节；
2. body：消息的有效载体，用String类型储存，长度无限制；
3. extend：协议的扩展字段，用Map类型储存，value值为Object对象；

## 2 私有协议属性

**协议头定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 数值 | 描述 | 备注 |
| PING\_PROTO | 1 << 8 | 220 | ping消息(476) |  |
| PONG\_PROTO | 2 << 8 | 220 | pong消息(732) |  |
| SYST\_PROTO | 3 << 8 | 220 | 系统消息(988) |  |
| EROR\_PROTO | 4 << 8 | 220 | 错误消息(1244) |  |
| AUTH\_PROTO | 5 << 8 | 220 | 认证消息(1500) |  |
| MESS\_PROTO | 6 << 8 | 220 | 普通消息(1756) |  |
| PRIV\_PROTO | 7 << 8 | 220 | 私聊消息(2012) | 暂未使用 |

**消息类型定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 数值 | 描述 | 备注 |
| AUTH\_CODE | 10001 | 认证消息类型 |  |
| MESS\_CODE | 10002 | 普通消息类型 |  |
| PING\_CODE | 10011 | Ping消息类型 |  |
| PONG\_CODE | 10012 | Pong消息类型 |  |
| SYSTEM\_USER\_COUNT | 20001 | 在线用户数 |  |
| SYSTEM\_AUTH\_STATE | 20002 | 认证结果 |  |
| SYSTEM\_OTHER\_INFO | 20003 | 系统通知类型 |  |
| SYSTEM\_USER\_LIST | 20004 | 在线用户列表 |  |

**HTTP响应消息：**

|  |  |  |
| --- | --- | --- |
| 数值 | 描述 | 备注 |
| 1xx | 指示消息 | 请求已接受收，继续处理 |
| 2xx | 成功消息 | 请求已被接收、理解、接受 |
| 3xx | 重定向消息 | 要完成请求需要更进一步的操作 |
| 4xx | 客户端错误消息 | 请求有语法错误或者请求无法实现 |
| 5xx | 服务器端错误消息 | 服务器未能处理请求 |

# 九、总结

……………………………………

# 十、参考资料

* 《Netty权威指南》第二版(李林峰 著)
* 《HTML5 WebSocket权威指南》(姚军 译)
* CSDN博客
* Github Netty项目