**Netty学习（五）—IdleStateHandler心跳机制**

Netty的IdleStateHandler心跳机制主要是用来检测远端是否存活，如果不存活或活跃则对空闲Socket连接进行处理避免资源的浪费；

个人主页：[tuzhenyu’s page](http://www.tuzhenyu.top/)   
原文地址：[Netty学习（五）—IdleStateHandler心跳机制](http://www.tuzhenyu.top/article/48)

**(0)基础**

* 心跳机制
  + ﻿心跳是在TCP长连接中，客户端和服务端定时向对方发送数据包通知对方自己还在线，保证连接的有效性的一种机制
  + 在服务器和客户端之间一定时间内没有数据交互时, 即处于 idle 状态时, 客户端或服务器会发送一个特殊的数据包给对方, 当接收方收到这个数据报文后, 也立即发送一个特殊的数据报文, 回应发送方, 此即一个 PING-PONG 交互. 自然地, 当某一端收到心跳消息后, 就知道了对方仍然在线, 这就确保 TCP 连接的有效性.
* 心跳实现
  + 使用TCP协议层的Keeplive机制，但是该机制默认的心跳时间是2小时，依赖操作系统实现不够灵活；
  + 应用层实现自定义心跳机制，比如Netty实现心跳机制；

**(1) IdleStateHandler心跳检测实例**

**服务端**

* 服务端添加IdleStateHandler心跳检测处理器，并添加自定义处理Handler类实现userEventTriggered()方法作为超时事件的逻辑处理；
* 设定IdleStateHandler心跳检测每五秒进行一次读检测，如果五秒内ChannelRead()方法未被调用则触发一次userEventTrigger()方法

ServerBootstrap b= new ServerBootstrap();

b.group(bossGroup,workerGroup).channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG,1024)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel socketChannel) throws Exception {

　socketChannel.pipeline().addLast(new IdleStateHandler(5, 0, 0, TimeUnit.SECONDS));

socketChannel.pipeline().addLast(new StringDecoder());

socketChannel.pipeline().addLast(new HeartBeatServerHandler())；

}

});

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 自定义处理类Handler继承ChannlInboundHandlerAdapter，实现其userEventTriggered()方法，在出现超时事件时会被触发，包括读空闲超时或者写空闲超时；

class HeartBeatServerHandler extends ChannelInboundHandlerAdapter {

private int lossConnectCount = 0;

@Override

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

System.out.println("已经5秒未收到客户端的消息了！");

if (evt instanceof IdleStateEvent){

IdleStateEvent event = (IdleStateEvent)evt;

if (event.state()== IdleState.READER\_IDLE){

lossConnectCount++;

if (lossConnectCount>2){

System.out.println("关闭这个不活跃通道！");

ctx.channel().close();

}

}

}else {

super.userEventTriggered(ctx,evt);

}

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

lossConnectCount = 0;

System.out.println("client says: "+msg.toString());

}

@Override

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

ctx.close();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

**客户端**

* 客户端添加IdleStateHandler心跳检测处理器，并添加自定义处理Handler类实现userEventTriggered()方法作为超时事件的逻辑处理；
* 设定IdleStateHandler心跳检测每四秒进行一次写检测，如果四秒内write()方法未被调用则触发一次userEventTrigger()方法，实现客户端每四秒向服务端发送一次消息；

Bootstrap b = new Bootstrap();

b.group(group).channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel socketChannel) throws Exception {

socketChannel.pipeline().addLast(new IdleStateHandler(0,4,0, TimeUnit.SECONDS));

socketChannel.pipeline().addLast(new StringEncoder());

socketChannel.pipeline().addLast(new HeartBeatClientHandler());

}

});

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 自定义处理类Handler继承ChannlInboundHandlerAdapter，实现自定义userEventTrigger()方法，如果出现超时时间就会被触发，包括读空闲超时或者写空闲超时；

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

System.out.println("客户端循环心跳监测发送: "+new Date());

if (evt instanceof IdleStateEvent){

IdleStateEvent event = (IdleStateEvent)evt;

if (event.state()== IdleState.WRITER\_IDLE){

if (curTime<beatTime){

curTime++;

ctx.writeAndFlush("biubiu");

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

**(2) IdleStateHandler源码分析**

* IdleStateHandler构造器
  + readerIdleTime读空闲超时时间设定，如果channelRead()方法超过readerIdleTime时间未被调用则会触发超时事件调用userEventTrigger()方法；
  + writerIdleTime写空闲超时时间设定，如果write()方法超过writerIdleTime时间未被调用则会触发超时事件调用userEventTrigger()方法；
  + allIdleTime所有类型的空闲超时时间设定，包括读空闲和写空闲；
  + unit时间单位，包括时分秒等；

public IdleStateHandler(

long readerIdleTime, long writerIdleTime, long allIdleTime,

TimeUnit unit) {

this(false, readerIdleTime, writerIdleTime, allIdleTime, unit);

}

* 1
* 2
* 3
* 4
* 5
* 心跳检测也是一种Handler，在启动时添加到ChannelPipeline管道中，当有读写操作时消息在其中传递；

socketChannel.pipeline().addLast(new IdleStateHandler(5, 0, 0, TimeUnit.SECONDS));

* 1
* IdleStateHandler的channelActive()方法在socket通道建立时被触发

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

initialize(ctx);

super.channelActive(ctx);

}

* 1
* 2
* 3
* 4
* 5
* channelActive()方法调用Initialize()方法,根据配置的readerIdleTime，WriteIdleTIme等超时事件参数往任务队列taskQueue中添加定时任务task ；

private void initialize(ChannelHandlerContext ctx) {

switch (state) {

case 1:

case 2:

return;

}

state = 1;

initOutputChanged(ctx);

lastReadTime = lastWriteTime = ticksInNanos();

if (readerIdleTimeNanos > 0) {

readerIdleTimeout = schedule(ctx, new ReaderIdleTimeoutTask(ctx),

readerIdleTimeNanos, TimeUnit.NANOSECONDS);

}

if (writerIdleTimeNanos > 0) {

writerIdleTimeout = schedule(ctx, new WriterIdleTimeoutTask(ctx),

writerIdleTimeNanos, TimeUnit.NANOSECONDS);

}

if (allIdleTimeNanos > 0) {

allIdleTimeout = schedule(ctx, new AllIdleTimeoutTask(ctx),

allIdleTimeNanos, TimeUnit.NANOSECONDS);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 定时任务添加到对应线程EventLoopExecutor对应的任务队列taskQueue中，在对应线程的run()方法中循环执行
  + 用当前时间减去最后一次channelRead方法调用的时间判断是否空闲超时；
  + 如果空闲超时则创建空闲超时事件并传递到channelPipeline中；

protected void run(ChannelHandlerContext ctx) {

long nextDelay = readerIdleTimeNanos;

if (!reading) {

nextDelay -= ticksInNanos() - lastReadTime;

}

if (nextDelay <= 0) {

// Reader is idle - set a new timeout and notify the callback.

readerIdleTimeout = schedule(ctx, this, readerIdleTimeNanos, TimeUnit.NANOSECONDS);

boolean first = firstReaderIdleEvent;

firstReaderIdleEvent = false;

try {

IdleStateEvent event = newIdleStateEvent(IdleState.READER\_IDLE, first);

channelIdle(ctx, event);

} catch (Throwable t) {

ctx.fireExceptionCaught(t);

}

} else {

// Read occurred before the timeout - set a new timeout with shorter delay.

readerIdleTimeout = schedule(ctx, this, nextDelay, TimeUnit.NANOSECONDS);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 在管道中传递调用自定义的userEventTrigger()方法

protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt) throws Exception {

ctx.fireUserEventTriggered(evt);

}

* 1
* 2
* 3

**总结**

* IdleStateHandler心跳检测主要是通过向线程任务队列中添加定时任务，判断channelRead()方法或write()方法是否调用空闲超时，如果超时则触发超时事件执行自定义userEventTrigger()方法；
* Netty通过IdleStateHandler实现最常见的心跳机制不是一种双向心跳的PING-PONG模式，而是客户端发送心跳数据包，服务端接收心跳但不回复，因为如果服务端同时有上千个连接，心跳的回复需要消耗大量网络资源；如果服务端一段时间内内有收到客户端的心跳数据包则认为客户端已经下线，将通道关闭避免资源的浪费；在这种心跳模式下服务端可以感知客户端的存活情况，无论是宕机的正常下线还是网络问题的非正常下线，服务端都能感知到，而客户端不能感知到服务端的非正常下线；
* 要想实现客户端感知服务端的存活情况，需要进行双向的心跳；Netty中的channelInactive()方法是通过Socket连接关闭时挥手数据包触发的，因此可以通过channelInactive()方法感知正常的下线情况，但是因为网络异常等非正常下线则无法感知；