netty-04-心跳

1.何为心跳

顾名思义, 所谓 心跳, 即在 TCP 长连接中, 客户端和服务器之间定期发送的一种特殊的数据包, 通知对方自己还在线, 以确保 TCP 连接的有效性.

为什么需要心跳
因为网络的不可靠性, 有可能在 TCP 保持长连接的过程中, 由于某些突发情况, 例如网线被拔出, 突然掉电等, 会造成服务器和客户端的连接中断. 在这些突发情况下, 如果恰好服务器和客户端之间没有交互的话, 那么它们是不能在短时间内发现对方已经掉线的. 为了解决这个问题, 我们就需要引入 心跳 机制. 心跳机制的工作原理是: 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.

2.如何实现心跳

我们可以通过两种方式实现心跳机制:

使用 TCP 协议层面的 keepalive 机制.
在应用层上实现自定义的心跳机制.
虽然在 TCP 协议层面上, 提供了 keepalive 保活机制, 但是使用它有几个缺点:

它不是 TCP 的标准协议, 并且是默认关闭的.
TCP keepalive 机制依赖于操作系统的实现, 默认的 keepalive 心跳时间是 两个小时, 并且对 keepalive 的修改需要系统调用(或者修改系统配置), 灵活性不够.
TCP keepalive 与 TCP 协议绑定, 因此如果需要更换为 UDP 协议时, keepalive 机制就失效了.
虽然使用 TCP 层面的 keepalive 机制比自定义的应用层心跳机制节省流量, 但是基于上面的几点缺点, 一般的实践中, 人们大多数都是选择在应用层上实现自定义的心跳.
既然如此, 那么我们就来大致看看在在 Netty 中是怎么实现心跳的吧. 在 Netty 中, 实现心跳机制的关键是 IdleStateHandler, 它可以对一个 Channel 的 读/写设置定时器, 当 Channel 在一定事件间隔内没有数据交互时(即处于 idle 状态), 就会触发指定的事件.

3.Netty 超时机制及实现心跳

Netty 超时机制的介绍

Netty 的超时类型 IdleState 主要分为：

ALL_IDLE : 一段时间内没有数据接收或者发送
READER_IDLE： 一段时间内没有数据接收
WRITER_IDLE： 一段时间内没有数据发送
在 Netty 的 timeout 包下，主要类有：

IdleStateEvent： 超时的事件
IdleStateHandler： 超时状态处理
ReadTimeoutHandler： 读超时状态处理
WriteTimeoutHandler： 写超时状态处理
其中 IdleStateHandler 包含了读\写超时状态处理，比如

```
1 private static final int READ_IDEL_TIME_OUT = 4; // 读超时
2 private static final int WRITE_IDEL_TIME_OUT = 5;// 写超时
3 private static final int ALL_IDEL_TIME_OUT = 7; // 所有超时
4
5 new IdleStateHandler(READ_IDEL_TIME_OUT,
6             WRITE_IDEL_TIME_OUT, ALL_IDEL_TIME_OUT, TimeUnit.SECONDS));
```

上述例子，在 IdleStateHandler 中定义了读超时的时间是 4 秒， 写超时的时间是 5 秒，其他所有的超时时间是 7 秒。

应用 IdleStateHandler

既然 IdleStateHandler 包括了读\写超时状态处理，那么很多时候 ReadTimeoutHandler 、 WriteTimeoutHandler 都可以不用使用。定义另一个名为 HeartbeatHandlerInitializer 的 ChannelInitializer：

```
1  public class HeartbeatHandlerInitializer extends ChannelInitializer<Channel> {
2
3      private static final int READ_IDEL_TIME_OUT = 4; // 读超时
4      private static final int WRITE_IDEL_TIME_OUT = 5;// 写超时
5      private static final int ALL_IDEL_TIME_OUT = 7; // 所有超时
6
7      @Override
8      protected void initChannel(Channel ch) throws Exception {
9          ChannelPipeline pipeline = ch.pipeline();
10         pipeline.addLast(new IdleStateHandler(READ_IDEL_TIME_OUT,
11                 WRITE_IDEL_TIME_OUT, ALL_IDEL_TIME_OUT, TimeUnit.SECONDS)); // 1
12         pipeline.addLast(new HeartbeatServerHandler()); // 2
13
```

```
    }
}
```

使用了 IdleStateHandler ，分别设置了读、写超时的时间
定义了一个 HeartbeatServerHandler 处理器，用来处理超时时，发送心跳

定义了一个心跳处理器

```
public class HeartbeatServerHandler extends ChannelInboundHandlerAdapter {

    // Return a unreleasable view on the given ByteBuf
    // which will just ignore release and retain calls.
    private static final ByteBuf HEARTBEAT_SEQUENCE = Unpooled
            .unreleasableBuffer(Unpooled.copiedBuffer("Heartbeat",
                    CharsetUtil.UTF_8));  // 1

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

        if (evt instanceof IdleStateEvent) {  // 2
            IdleStateEvent event = (IdleStateEvent) evt;
            String type = "";
            if (event.state() == IdleState.READER_IDLE) {
                type = "read idle";
            } else if (event.state() == IdleState.WRITER_IDLE) {
                type = "write idle";
            } else if (event.state() == IdleState.ALL_IDLE) {
                type = "all idle";
            }

            ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate()).addListener(
                    ChannelFutureListener.CLOSE_ON_FAILURE);  // 3

            System.out.println( ctx.channel().remoteAddress()+"超时类型：" + type);
        } else {
            super.userEventTriggered(ctx, evt);
        }
    }
}
```

定义了心跳时，要发送的内容
判断是否是 IdleStateEvent 事件，是则处理
将心跳内容发送给客户端

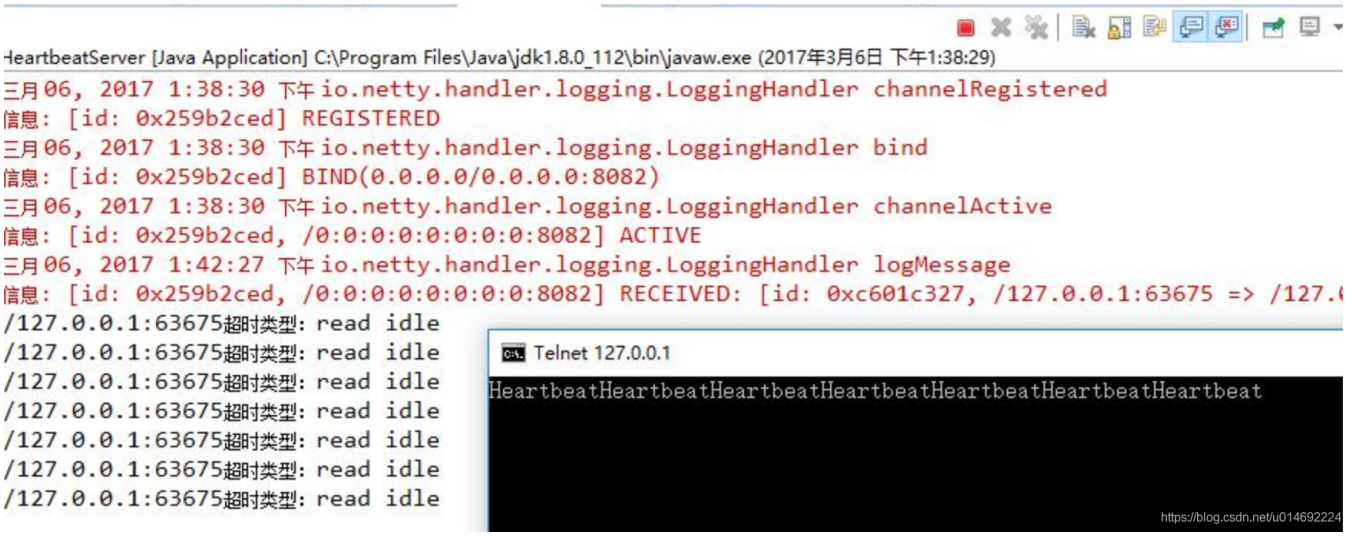
服务器

```
public final class HeartbeatServer {

    static final int PORT = 8082;

    public static void main(String[] args) throws Exception {

        // Configure the server.
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
             .channel(NioServerSocketChannel.class)
             .option(ChannelOption.SO_BACKLOG, 100)
             .handler(new LoggingHandler(LogLevel.INFO))
             .childHandler(new HeartbeatHandlerInitializer());

            // Start the server.
            ChannelFuture f = b.bind(PORT).sync();

            // Wait until the server socket is closed.
            f.channel().closeFuture().sync();
        } finally {
            // Shut down all event loops to terminate all threads.
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

客户端测试
客户端用操作系统自带的 Telnet 程序即可：

telnet 127.0.0.1 8082

效果



案例

使用 Netty 实现心跳

上面我们提到了, 在 Netty 中, 实现心跳机制的关键是 IdleStateHandler, 那么这个 Handler 如何使用呢? 我们来看看它的构造器:

```
1 public IdleStateHandler(int readerIdleTimeSeconds, int writerIdleTimeSeconds, int allIdleTimeSeconds) {
2     this((long)readerIdleTimeSeconds, (long)writerIdleTimeSeconds, (long)allIdleTimeSeconds, TimeUnit.SECONDS);
3 }
```

实例化一个 IdleStateHandler 需要提供三个参数:

readerIdleTimeSeconds, 读超时. 即当在指定的事件间隔内没有从 Channel 读取到数据时, 会触发一个 READER_IDLE 的 IdleStateEvent 事件.
writerIdleTimeSeconds, 写超时. 即当在指定的事件间隔内没有数据写入到 Channel 时, 会触发一个 WRITER_IDLE 的 IdleStateEvent 事件.
allIdleTimeSeconds, 读/写超时. 即当在指定的事件间隔内没有读或写操作时, 会触发一个 ALL_IDLE 的 IdleStateEvent 事件.
为了展示具体的 IdleStateHandler 实现的心跳机制, 下面我们来构造一个具体的EchoServer 的例子, 这个例子的行为如下:

在这个例子中, 客户端和服务器通过 TCP 长连接进行通信.
TCP 通信的报文格式是:
±-------±----±--------------+|
Length |Type | Content |
| 17 | 1 |“HELLO, WORLD” |
±-------±----±--------------+

1.客户端每隔一个随机的时间后, 向服务器发送消息, 服务器收到消息后, 立即将收到的消息原封不动地回复给客户端.
2.若客户端在指定的时间间隔内没有读/写操作, 则客户端会自动向服务器发送一个 PING 心跳, 服务器收到 PING 心跳消息时, 需要回复一个 PONG 消息.

通用部分

```
1 public abstract class CustomHeartbeatHandler extends SimpleChannelInboundHandler<ByteBuf> {
2
3     public static final byte PING_MSG = 1;
```

```
4
5       public static final byte PONG_MSG = 2;
6
7       public static final byte CUSTOM_MSG = 3;
8
9       protected String name;
10
11      private int heartbeatCount = 0;
12
13      public CustomHeartbeatHandler(String name) {
14          this.name = name;
15      }
16
17      @Override
18      protected void channelRead0(ChannelHandlerContext context, ByteBuf byteBuf) throws Exception {
19
20          if (byteBuf.getByte(4) == PING_MSG) {
21              sendPongMsg(context);
22          } else if (byteBuf.getByte(4) == PONG_MSG){
23              System.out.println(name + " get pong msg from " + context.channel().remoteAddress());
24          } else {
25              handleData(context, byteBuf);
26          }
27
28      }
29
30
31
32      protected void sendPingMsg(ChannelHandlerContext context) {
33
34          ByteBuf buf = context.alloc().buffer(5);
35
36          buf.writeInt(5);
37
38          buf.writeByte(PING_MSG);
39
40          buf.retain();
41
42          context.writeAndFlush(buf);
43
44          heartbeatCount++;
45
46          System.out.println(name + " sent ping msg to " + context.channel().remoteAddress() + ", count: " + heartbeatCount);
47
48      }
49
50
51
52      private void sendPongMsg(ChannelHandlerContext context) {
53
54          ByteBuf buf = context.alloc().buffer(5);
55
56          buf.writeInt(5);
57
58          buf.writeByte(PONG_MSG);
59
60          context.channel().writeAndFlush(buf);
61
62          heartbeatCount++;
63
64          System.out.println(name + " sent pong msg to " + context.channel().remoteAddress() + ", count: " + heartbeatCount);
65
66      }
67
68
69      protected abstract void handleData(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf);
70
71      @Override
72      public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
73          // IdleStateHandler 所产生的 IdleStateEvent 的处理逻辑.
74          if (evt instanceof IdleStateEvent) {
75              IdleStateEvent e = (IdleStateEvent) evt;
76
77              switch (e.state()) {
78                  case READER_IDLE:
79                      handleReaderIdle(ctx);
80                      break;
81                  case WRITER_IDLE:
82                      handleWriterIdle(ctx);
83                      break;
84                  case ALL_IDLE:
85                      handleAllIdle(ctx);
```

```
                    break;
                default:
                    break;
            }
        }
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
       System.err.println("---" + ctx.channel().remoteAddress() + " is active---");
    }



    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
       System.err.println("---" + ctx.channel().remoteAddress() + " is inactive---");
    }



    protected void handleReaderIdle(ChannelHandlerContext ctx) {
        System.err.println("---READER_IDLE---");
    }

    protected void handleWriterIdle(ChannelHandlerContext ctx) {
        System.err.println("---WRITER_IDLE---");
    }

    protected void handleAllIdle(ChannelHandlerContext ctx) {
        System.err.println("---ALL_IDLE---");
    }
}
```

类 CustomHeartbeatHandler 负责心跳的发送和接收, 我们接下来详细地分析一下它的作用. 我们在前面提到, IdleStateHandler 是实现心跳的关键, 它会根据不同的 IO idle 类型来产生不同的 IdleStateEvent 事件, 而这个事件的捕获, 其实就是在 userEventTriggered 方法中实现的.
我们来看看 CustomHeartbeatHandler.userEventTriggered 的具体实现:

```
@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
    if (evt instanceof IdleStateEvent) {
        IdleStateEvent e = (IdleStateEvent) evt;
        switch (e.state()) {
            case READER_IDLE:
                handleReaderIdle(ctx);
                break;
            case WRITER_IDLE:
                handleWriterIdle(ctx);
                break;
            case ALL_IDLE:
                handleAllIdle(ctx);
                break;
            default:
                break;
        }
    }
}
```

在 userEventTriggered 中, 根据 IdleStateEvent 的 state() 的不同, 而进行不同的处理. 例如如果是读取数据 idle, 则 e.state() == READER_IDLE, 因此就调用 handleReaderIdle 来处理它. CustomHeartbeatHandler 提供了三个 idle 处理方法: handleReaderIdle, handleWriterIdle, handleAllIdle, 这三个方法目前只有默认的实现, 它需要在子类中进行重写, 现在我们暂时略过它们, 在具体的客户端和服务器的实现部分时再来看它们.
知道了这一点后, 我们接下来看看数据处理部分:

```
@Override
protected void channelRead0(ChannelHandlerContext context, ByteBuf byteBuf) throws Exception {
    if (byteBuf.getByte(4) == PING_MSG) {
        sendPongMsg(context);
    } else if (byteBuf.getByte(4) == PONG_MSG){
        System.out.println(name + " get pong msg from " + context.channel().remoteAddress());
    } else {
        handleData(context, byteBuf);
    }

}
```

在 CustomHeartbeatHandler.channelRead0 中, 我们首先根据报文协议:
+------+---+----------+

| Length |Type | Content |
| 17 | 1 |“HELLO, WORLD” |
+-------+----+-------------+

来判断当前的报文类型, 如果是 PING_MSG 则表示是服务器收到客户端的 PING 消息, 此时服务器需要回复一个 PONG 消息, 其消息类型是 PONG_MSG.扔报文类型是 PONG_MSG, 则表示是客户端收到服务器发送的 PONG 消息, 此时打印一个 log 即可.

客户端部分

客户端初始化

```
public class Client {
    public static void main(String[] args) {
        NioEventLoopGroup workGroup = new NioEventLoopGroup(4);
        Random random = new Random(System.currentTimeMillis());
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap
                    .group(workGroup)
                    .channel(NioSocketChannel.class)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        protected void initChannel(SocketChannel socketChannel) throws Exception {

                            ChannelPipeline p = socketChannel.pipeline();
                            p.addLast(new IdleStateHandler(0, 0, 5));
                            p.addLast(new LengthFieldBasedFrameDecoder(1024, 0, 4, -4, 0));
                            p.addLast(new ClientHandler());
                        }

                    });


            Channel ch = bootstrap.remoteAddress("127.0.0.1", 12345).connect().sync().channel();

            for (int i = 0; i < 10; i++) {
                String content = "client msg " + i;

                ByteBuf buf = ch.alloc().buffer();

                buf.writeInt(5 + content.getBytes().length);

                buf.writeByte(CustomHeartbeatHandler.CUSTOM_MSG);

                buf.writeBytes(content.getBytes());

                ch.writeAndFlush(buf);

                Thread.sleep(random.nextInt(20000));
            }

        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            workGroup.shutdownGracefully();
        }
    }
}
```

上面的代码是 Netty 的客户端端的初始化代码, 使用过 Netty 的朋友对这个代码应该不会陌生. 别的部分我们就不再赘述, 我们来看看 ChannelInitializer.initChannel 部分即可:

```
.handler(new ChannelInitializer<SocketChannel>() {
    protected void initChannel(SocketChannel socketChannel) throws Exception {
        ChannelPipeline p = socketChannel.pipeline();
        p.addLast(new IdleStateHandler(0, 0, 5));
        p.addLast(new LengthFieldBasedFrameDecoder(1024, 0, 4, -4, 0));
        p.addLast(new ClientHandler());
    }
});
```

我们给 pipeline 添加了三个 Handler, IdleStateHandler 这个 handler 是心跳机制的核心, 我们为客户端设置了读写 idle 超时, 时间间隔是5s, 即如果客户端在间隔 5s 后都没有收到服务器的消息或向服务器发送消息, 则产生 ALL_IDLE 事件.
接下来我们添加了 LengthFieldBasedFrameDecoder, 它是负责解析我们的 TCP 报文, 因为和本文的目的无关, 因此这里不详细展开.
最后一个 Handler 是 ClientHandler, 它继承于 CustomHeartbeatHandler, 是我们处理业务逻辑部分.

客户端 Handler

```
public class ClientHandler extends CustomHeartbeatHandler {

    public ClientHandler() {
```

```
        super("client");
    }

    @Override
    protected void handleData(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf) {
        byte[] data = new byte[byteBuf.readableBytes() - 5];

        byteBuf.skipBytes(5);

        byteBuf.readBytes(data);

        String content = new String(data);

        System.out.println(name + " get content: " + content);
    }


    @Override
    protected void handleAllIdle(ChannelHandlerContext ctx) {
        super.handleAllIdle(ctx);

        sendPingMsg(ctx);

    }

}
```

ClientHandler 继承于 CustomHeartbeatHandler, 它重写了两个方法, 一个是 handleData, 在这里面实现 仅仅打印收到的消息.
第二个重写的方法是 handleAllIdle. 我们在前面提到, 客户端负责发送心跳的 PING 消息, 当客户端产生一个 ALL_IDLE 事件后, 会导致父类的 CustomHeartbeatHandler.userEventTriggered 调用, 而 userEventTriggered 中会根据 e.state() 来调用不同的方法, 因此最后调用的是 ClientHandler.handleAllIdle, 在这个方法中, 客户端调用 sendPingMsg 向服务器发送一个 PING 消息.

服务器部分
服务器初始化

```
public class Server {
    public static void main(String[] args) {

        NioEventLoopGroup bossGroup = new NioEventLoopGroup(1);
        NioEventLoopGroup workGroup = new NioEventLoopGroup(4);

        try {
            ServerBootstrap bootstrap = new ServerBootstrap();
            bootstrap
                    .group(bossGroup, workGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        protected void initChannel(SocketChannel socketChannel) throws Exception {
                            ChannelPipeline p = socketChannel.pipeline();
                            p.addLast(new IdleStateHandler(10, 0, 0));
                            p.addLast(new LengthFieldBasedFrameDecoder(1024, 0, 4, -4, 0));
                            p.addLast(new ServerHandler());
                        }

                    });



            Channel ch = bootstrap.bind(12345).sync().channel();
            ch.closeFuture().sync();

        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {

            bossGroup.shutdownGracefully();
            workGroup.shutdownGracefully();
        }
    }
}
```

服务器的初始化部分也没有什么好说的, 它也和客户端的初始化一样, 为 pipeline 添加了三个 Handler.

服务器 Handler

```
public class ServerHandler extends CustomHeartbeatHandler {

    public ServerHandler() {
        super("server");
    }
```

```
    @Override
    protected void handleData(ChannelHandlerContext channelHandlerContext, ByteBuf buf) {
        byte[] data = new byte[buf.readableBytes() - 5];

        ByteBuf responseBuf = Unpooled.copiedBuffer(buf);

        buf.skipBytes(5);

        buf.readBytes(data);

        String content = new String(data);

        System.out.println(name + " get content: " + content);

        channelHandlerContext.write(responseBuf);
    }

    @Override
    protected void handleReaderIdle(ChannelHandlerContext ctx) {
        super.handleReaderIdle(ctx);
        System.err.println("---client " + ctx.channel().remoteAddress().toString() + " reader timeout, close it---");
        ctx.close();
    }
}
```

ServerHandler 继承于 CustomHeartbeatHandler, 它重写了两个方法, 一个是 handleData, 在这里面实现 EchoServer 的功能: 即收到客户端的消息后, 立即原封不动地将消息回复给客户端.
第二个重写的方法是 handleReaderIdle, 因为服务器仅仅对客户端的读 idle 感兴趣, 因此只重新了这个方法. 若服务器在指定时间后没有收到客户端的消息, 则会触发 READER_IDLE 消息, 进而会调用 handleReaderIdle 这个方法. 我们在前面提到, 客户端负责发送心跳的 PING 消息, 并且服务器的 READER_IDLE 的超时时间是客户端发送 PING 消息的间隔的两倍, 因此当服务器 READER_IDLE 触发时, 就可以确定是客户端已经掉线了, 因此服务器直接关闭客户端连接即可.

总结
使用 Netty 实现心跳机制的关键就是利用 IdleStateHandler 来产生对应的 idle 事件.
一般是客户端负责发送心跳的 PING 消息, 因此客户端注意关注 ALL_IDLE 事件, 在这个事件触发后, 客户端需要向服务器发送 PING 消息, 告诉服务器"我还存活着".
服务器是接收客户端的 PING 消息的, 因此服务器关注的是 READER_IDLE 事件, 并且服务器的 READER_IDLE 间隔需要比客户端的 ALL_IDLE 事件间隔大(例如客户端ALL_IDLE 是5s 没有读写时触发, 因此服务器的 READER_IDLE 可以设置为10s)
当服务器收到客户端的 PING 消息时, 会发送一个 PONG 消息作为回复. 一个 PING-PONG 消息对就是一个心跳交互.

实现客户端的断线重连

```
public class Client {

    private NioEventLoopGroup workGroup = new NioEventLoopGroup(4);
    private Channel channel;
    private Bootstrap bootstrap;


    public static void main(String[] args) throws Exception {

        Client client = new Client();
        client.start();
        client.sendData();

    }


    public void sendData() throws Exception {

        Random random = new Random(System.currentTimeMillis());

        for (int i = 0; i < 10000; i++) {

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

                String content = "client msg " + i;

                ByteBuf buf = channel.alloc().buffer(5 + content.getBytes().length);

                buf.writeInt(5 + content.getBytes().length);

                buf.writeByte(CustomHeartbeatHandler.CUSTOM_MSG);

                buf.writeBytes(content.getBytes());

                channel.writeAndFlush(buf);

```

```
            }

            Thread.sleep(random.nextInt(20000));
        }
    }



    public void start() {
        try {
            bootstrap = new Bootstrap();
            bootstrap
                    .group(workGroup)
                    .channel(NioSocketChannel.class)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        protected void initChannel(SocketChannel socketChannel) throws Exception {
                            ChannelPipeline p = socketChannel.pipeline();

                            p.addLast(new IdleStateHandler(0, 0, 5));
                            p.addLast(new LengthFieldBasedFrameDecoder(1024, 0, 4, -4, 0));
                            p.addLast(new ClientHandler(Client.this));
                        }
                    });

            doConnect();

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }



    protected void doConnect() {
        if (channel != null && channel.isActive()) {
            return;
        }

        ChannelFuture future = bootstrap.connect("127.0.0.1", 12345);

        future.addListener(new ChannelFutureListener() {
            public void operationComplete(ChannelFuture futureListener) throws Exception {
                if (futureListener.isSuccess()) {
                    channel = futureListener.channel();

                    System.out.println("Connect to server successfully!");
                } else {
                    System.out.println("Failed to connect to server, try connect after 10s");
                    futureListener.channel().eventLoop().schedule(new Runnable() {

                        @Override
                        public void run() {
                            doConnect();
                        }

                    }, 10, TimeUnit.SECONDS);
                }
            }
        });
    }
}
```

上面的代码中, 我们抽象出 doConnect 方法, 它负责客户端和服务器的 TCP 连接的建立, 并且当 TCP 连接失败时, doConnect 会 通过 “channel().eventLoop().schedule” 来延时10s 后尝试重新连接.

客户端 Handler

```
public class ClientHandler extends CustomHeartbeatHandler {
    private Client client;

    public ClientHandler(Client client) {
        super("client");
        this.client = client;
    }



    @Override
    protected void handleData(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf) {

        byte[] data = new byte[byteBuf.readableBytes() - 5];
```

```
15
16          byteBuf.skipBytes(5);
17
18          byteBuf.readBytes(data);
19
20          String content = new String(data);
21
22          System.out.println(name + " get content: " + content);
23      }
24
25
26      @Override
27      protected void handleAllIdle(ChannelHandlerContext ctx) {
28          super.handleAllIdle(ctx);
29          sendPingMsg(ctx);
30      }
31
32
33
34      @Override
35
36      public void channelInactive(ChannelHandlerContext ctx) throws Exception {
37          super.channelInactive(ctx);
38          client.doConnect();
39      }
40
41  }
```

断线重连的关键一点是检测连接是否已经断开. 因此我们改写了 ClientHandler, 重写了 channelInactive 方法. 当 TCP 连接断开时, 会回调 channelInactive 方法, 因此我们在这个方法中调用 client.doConnect() 来进行重连.