TcpConnection#onConnected方法中调用AsyncPacketReader#startRead()方法；
startRead()方法创建任务线程，并start()启动线程，然后线程调用run()方法；
run()方法：

清空buffer;

检查buffer的capacity空间，如果剩余空间不够每次增加1K；

如果没有可读的数据，跳出循环，进入finally块进行重连，重新创建连接任务；

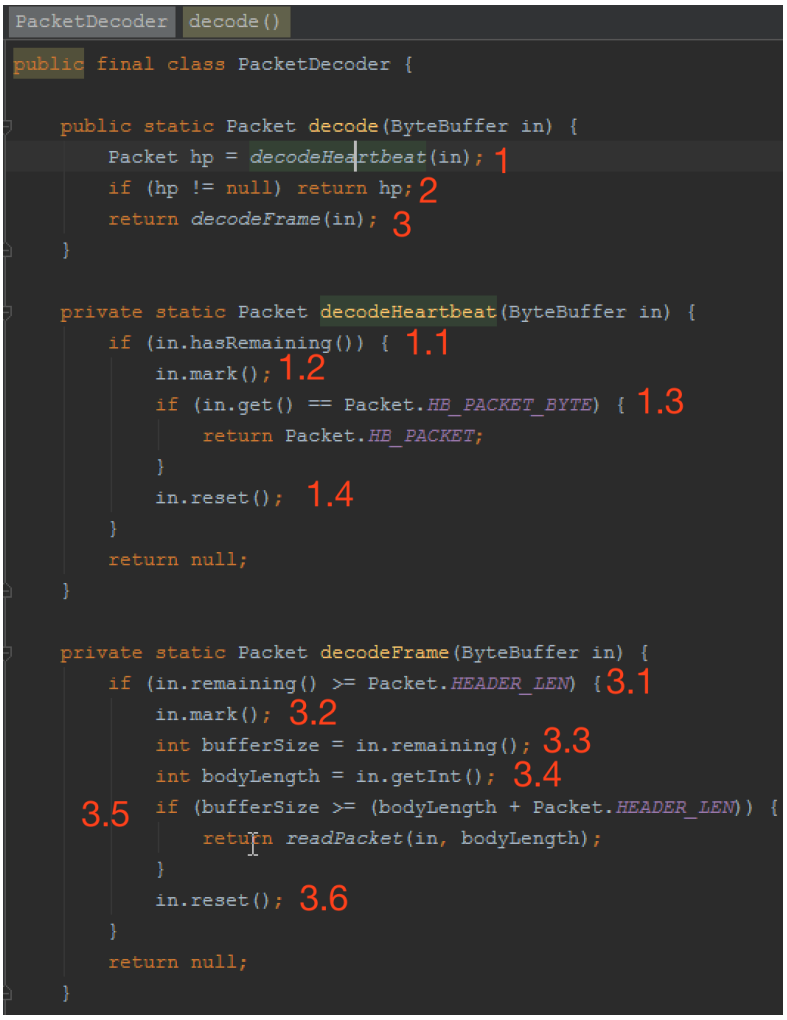
调用in.flip()，写模式切换成读模式，limit=position, position=0

调用decodePacket(in)消息解码，解码完成之后进行消息处理，调用MessageDispatcher#onReceive方法

调用in.compact() 压缩未读完的数据

```java
public final class AsyncPacketReader implements PacketReader, Runnable {
    private final NamedThreadFactory threadFactory = new NamedThreadFactory(ExecutorManager.READ_THREAD_NAME);
    private final Connection connection;
    private final PacketReceiver receiver;
    private final ByteBuf buffer;
    private final Logger logger;

    private Thread thread;

    public AsyncPacketReader(Connection connection, PacketReceiver receiver) {
        this.connection = connection;
        this.receiver = receiver;
        this.buffer = ByteBuf.allocateDirect(Short.MAX_VALUE);//默认读buffer大小为32k
        this.logger = ClientConfig.I.getLogger();
    }

    @Override
    public synchronized void startRead() {
        this.thread = threadFactory.newThread(this);
        this.thread.start();
    }

    @Override
    public synchronized void stopRead() {
        if (thread != null) {
            thread.interrupt();
            thread = null;
        }
    }

    public void run() {
        try {
            this.buffer.clear();
            while (connection.isConnected()) {
                ByteBuffer in = buffer.checkCapacity(1024).nioBuffer();//如果剩余空间不够每次增加1k
                if (!read(connection.getChannel(), in)) break;
                in.flip();
                decodePacket(in);
                in.compact();
            }
        } finally {
            logger.w("read an error, do reconnect!!!");
            connection.reconnect();
        }
    }

    private void decodePacket(ByteBuffer in) {
        Packet packet;
        while ((packet = PacketDecoder.decode(in)) != null) {
            //  logger.d("decode one packet=%s", packet);
            receiver.onReceive(packet, connection);
        }
    }

    private boolean read(SocketChannel channel, ByteBuffer in) {
        int readCount;
        try {
            readCount = channel.read(in);
            connection.setLastReadTime();
        } catch (IOException e) {
            logger.e(e, "read packet ex, do reconnect");
            readCount = -1;
            sleep4Reconnect();
        }
        return readCount > 0;
    }

    private void sleep4Reconnect() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
    }
}
```

消息解码



1、首先尝试心跳消息的解码

1.1 是否有剩余未读数据

1.2 标记现在读取的位置

1.3 调用in.get()获取1byte数据，判断是否是心跳类型，如果是心跳消息直接返回心跳实体对象；

1.4 如果不是心跳消息，则指针重置到1.2标记的位置(刚开始读取的位置)，好方便下一轮读取；

2、如果心跳实体对象不为空，则直接返回心跳实体对象，否则进行下一轮读取；

3、解码除心跳之外的所有消息

3.1 判断BUFFER中剩余的未读数据，是否大于消息头的长度 HEADER_LEN=13

bodyLength(4)+cmd(1)+cc(2)+flags(1)+sessionId(4)+lrc(1)+body(n)

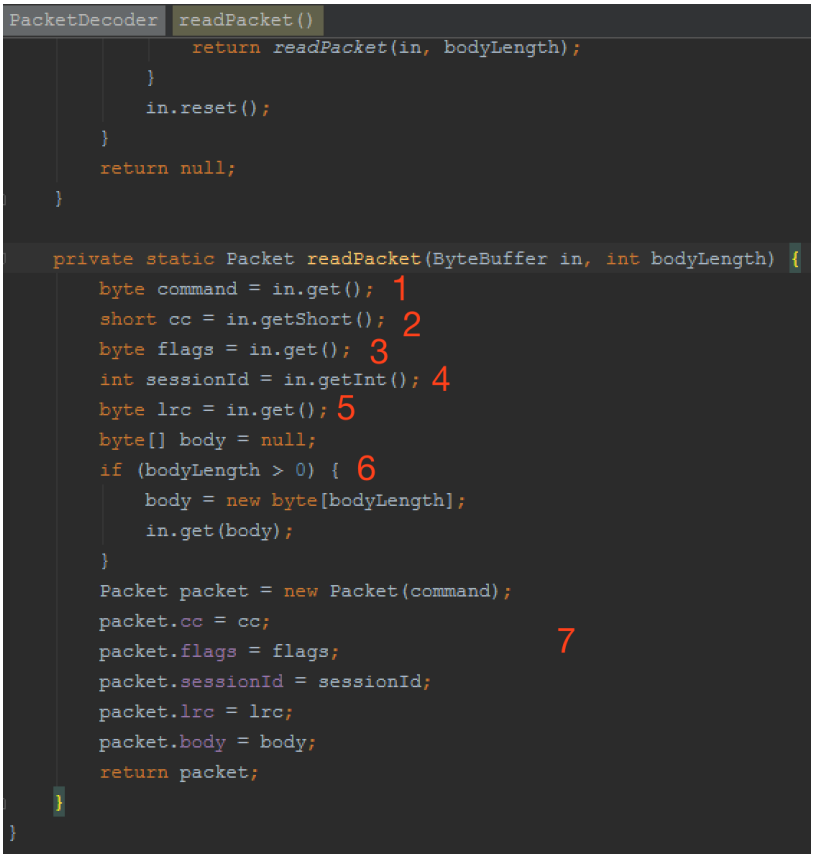
3.2 标记现在读取的位置

3.3 获取剩余未读数据大小

3.4 调用in.getInt()获取4个字节的bodyLength

3.5 如果 剩余未读数据长度 >= （body长度+包头长度），则表示有完整的数据可读，调用readPacket()方法进一步解码；

3.6 如果没有完整可读数据，则指针重置到3.2标记的位置(刚开始读取的位置)



1、获取cmd，调用in.get()获取1byte数据

2、获取cc，调用in.getShort()获取2byte数据

3、获取flags，调用in.get()获取1byte数据

4、获取sesssionId，调用in.getInt()获取4byte数据

5、获取lrc，调用in.get()获取1byte数据

6、如果包体长度bodyLength大于0，则获取body，调用in.get(body)获取数据

7、把获取到的数据封装成Packet对象