**t-io使用手册**

作者：谭耀武(t-io框架作者)

官网：<http://www.t-io.org>

说明：文档尚未完工

|  |  |
| --- | --- |
| ali_300px-1 | wechat_300px-1 |



**文档不错，可以捐赠**

**文档不错，购买纸质**

**文档不错，购买纸质**

|  |
| --- |
| <https://item.taobao.com/item.htm?spm=a1z38n.10677092.0.0.42051debpD7w0H&id=564922752040> |

目录

[第一章 tio简介 4](#_Toc3264)

[1.1. tio是啥 4](#_Toc31575)

[1.2. tio历史 4](#_Toc24236)

[1.3. tio适用场景 4](#_Toc30673)

[1.4. tio案例 5](#_Toc12709)

[1.5. tio性能 6](#_Toc18510)

[1.6. tio稳定性 7](#_Toc31768)

[1.7. tio生态 8](#_Toc27866)

[1.8. tio荣誉 10](#_Toc23530)

[1． 首批码云最有价值开源项目 11](#_Toc22947)

[2． 2017年最受欢迎开源软件上榜 11](#_Toc4840)

[3． 2017年热门开源项目Star数第3，Fork数第5 12](#_Toc22077)

[第二章 预备知识 13](#_Toc27523)

[1.1. TCP/IP协议分层模型 13](#_Toc18384)

[1.2. 应用层和传输层的数据传递 13](#_Toc28861)

[1． 应用层数据是个什么鬼 13](#_Toc17898)

[2． 应用层数据解码 15](#_Toc31283)

[3． 应用层数据编码 16](#_Toc5848)

[1.3. 认识java中的bytebuffer 17](#_Toc21916)

[1． 初识ByteBuffer 17](#_Toc22436)

[2． 创建ByteBuffer 19](#_Toc24900)

[3． 往ByteBuffer中写入数据 20](#_Toc29998)

[4． 从ByteBuffer读取数据 20](#_Toc18242)

[第三章 开启tio之旅 23](#_Toc1794)

[1.1. Hello Tio 23](#_Toc18299)

[1． hello world业务简介 23](#_Toc25042)

[2． 公共模块 23](#_Toc11959)

[3． 服务端代码 25](#_Toc9914)

[4． 客户端代码 28](#_Toc19275)

[5． 运行hello tio 32](#_Toc32189)

[1.2. Tio常见类介绍 34](#_Toc29734)

[1． ChannelContext(通道上下文) 34](#_Toc23838)

[2． GroupContext(服务配置与维护) 35](#_Toc7497)

[3． AioHandler(消息处理接口) 36](#_Toc3199)

[4． AioListener(通道监听者) 37](#_Toc3308)

[5． Packet(应用层数据包) 38](#_Toc18919)

[6． AioServer（tio服务端入口类） 39](#_Toc7904)

[7． AioClient（tio客户端入口类） 39](#_Toc26663)

[8． ObjWithLock（自带读写锁的对象） 40](#_Toc12271)

# tio简介

## tio是啥

t-io是一个网络框架，从这一点来说是有点像netty的，但t-io为常见和网络相关的业务（如IM、消息推送、RPC、监控）提供了近乎于现成的解决方案，即丰富的编程API，极大减少业务层的编程难度。

## tio历史



图 1 tio历史

## tio适用场景

如果你的应用是基于TCP的，那么tio就是支持的；tio也有提供udp开发，但笔者并没有过多使用，所以暂时不建议大家使用tio-core的udp api。

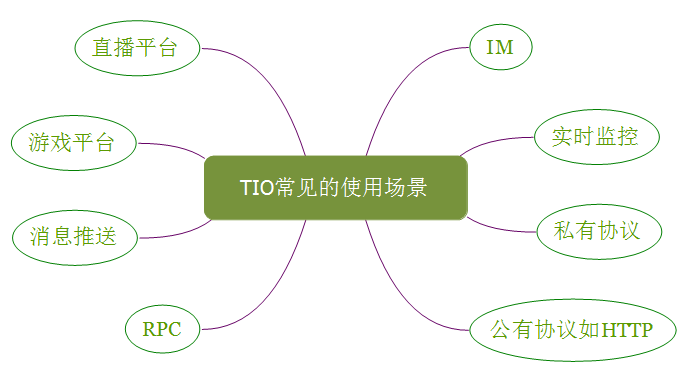


图 2 tio常见的使用场景

## tio案例

目前在册案例22个，还有两个已知的游戏案例没有记录（用户出于安全，不愿意被记录），还有至少5个以上的案例被漏记。本人不知道的案例应该更多，所以在册案例数只供参考，并没有太多意义。

图 3 tio案例展示

## tio性能

性能不是评价框架唯一甚至不是最重要的指标，就像性能不是万能的，但没有性能是万万不能的。

1. 曾有用户用tio提供的im测出每秒收发500万条聊天消息
2. t-io 30W长连接并发压力测试报告(https://my.oschina.net/u/2369298/blog/915435)



图 4 t-io 30W长连接并发压力测试报告

## tio稳定性

事实胜于雄辩，关于tio的稳定性，笔者就列举一些事实吧

1. 参看tio历史，tio前身的前身，也就是中兴的EMF，代码仍然在使用（截止2018年3月3号，已经使用了7年），不稳定的代码早就会中兴这种大公司抛弃了（当年emf上线前，被拷机测试过3个月，而且是专门配了个C++工程师写的测试客户端）。
2. 热波直播平台是用tio的前身talent-nio开发的，聊天模块非常稳定，上线两年多零故障（是聊天模块零故障，并不是指整个直播平台零故障----因为流媒体模块故障挺多）。
3. 基于tio-http-server的tio官网不间断运行1610+小时（67天），该时间仍在延长，用JVirsualVM监控出来的内存和CPU仍十分漂亮。

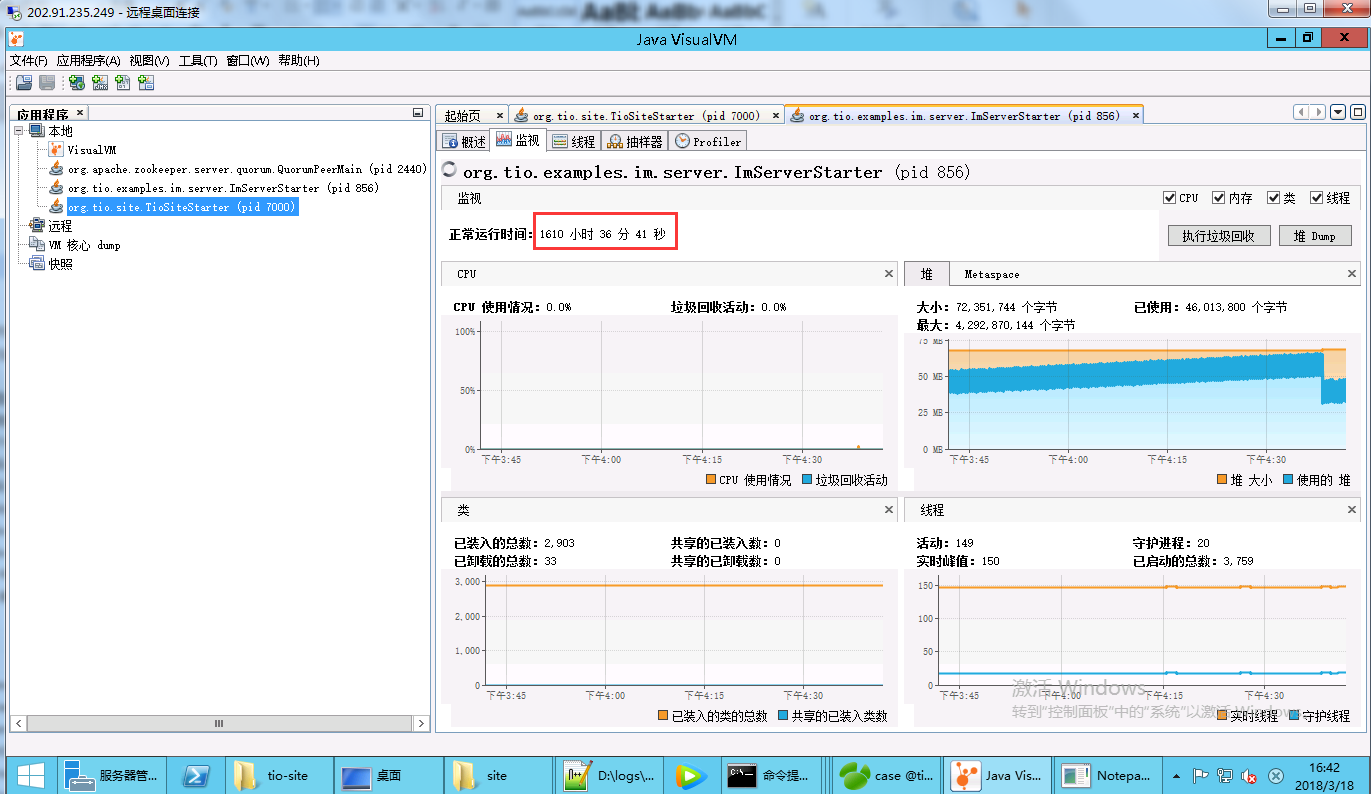


图 5 tio官网运行时间

1. 如果以上几条不能让你放心，您可以来牛吧云播体验一下tio全家桶开发的作品(<https://www.nb350.com)---->未使用spring、未使用任何servlet容器，从http到websocket到socket到视力模板，都是tio实现。

## tio生态

tio生态已经自动形成，可以参看下面三张截图



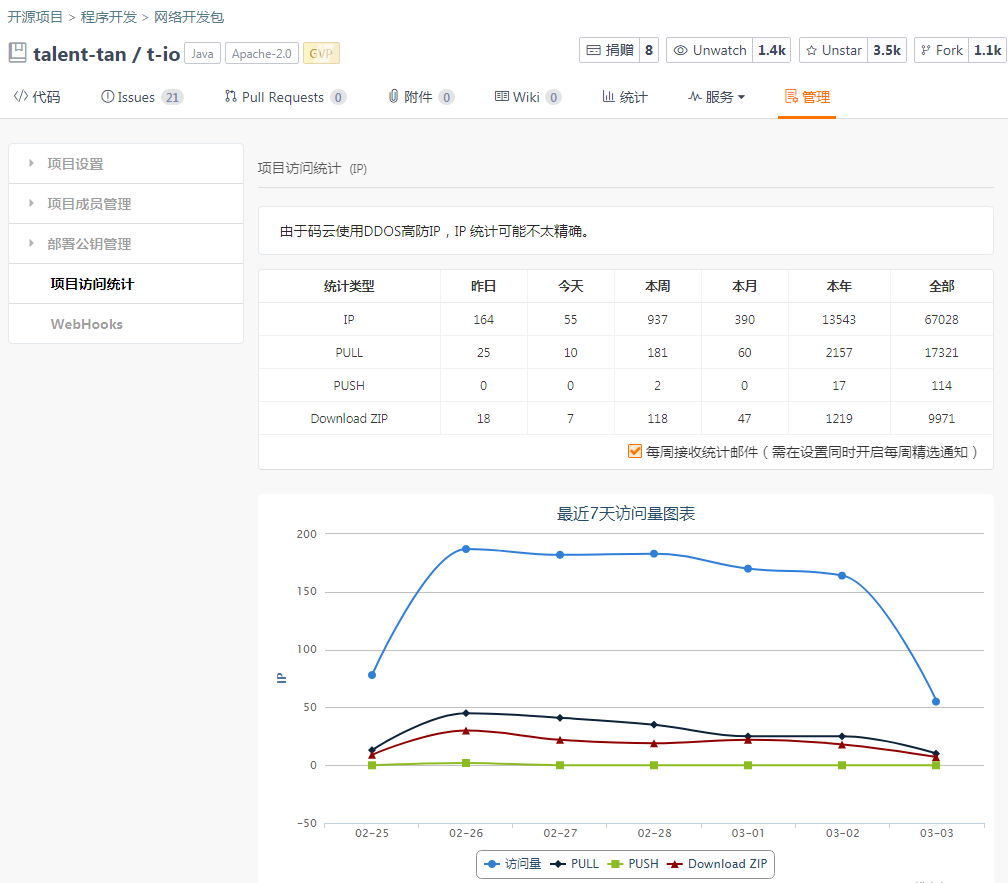
图 6 tio在OSC上的收藏数、评论数、

图 7 tio在码云上的后台统计截图



图 8 tio受捐次数

## tio荣誉

荣誉与作者而言，大抵是浮云，但之所以提起这些荣誉，只是想让用户们确信，他们的选择没毛病！

### 首批码云最有价值开源项目

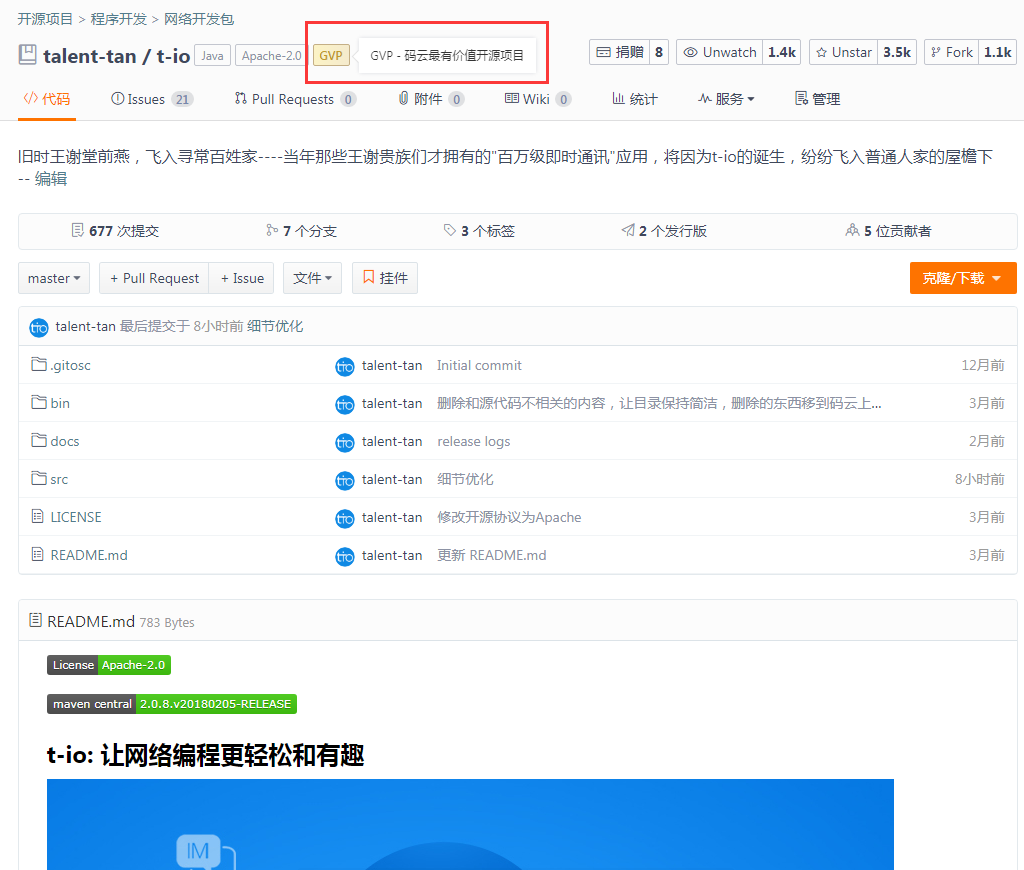


图 9 首批码云最有价值开源项目

### 2017年最受欢迎开源软件上榜

考虑到tio的用户群体特点，能进入TOP20，已属不易



图 10 2017年最受欢迎开源软件上榜

### 2017年热门开源项目Star数第3，Fork数第5



图 11 2017年热门开源项目Star数第3，Fork数第5

# 预备知识

## TCP/IP协议分层模型

大学教科书中有说分成7层，也有说分成4层的，我个人觉得4层更合适一些，像七层中的第5、6层完全不是必须的，就算有也是各自制定协议，而制定协议的人基本不会去考虑第5层叫会话层第6层叫表示层，在私有应用层协议中，更多的是会私定一个握手互信协议，以表示通讯双方是靠谱的队友，而不是敌人（常见于各种网络攻击）。

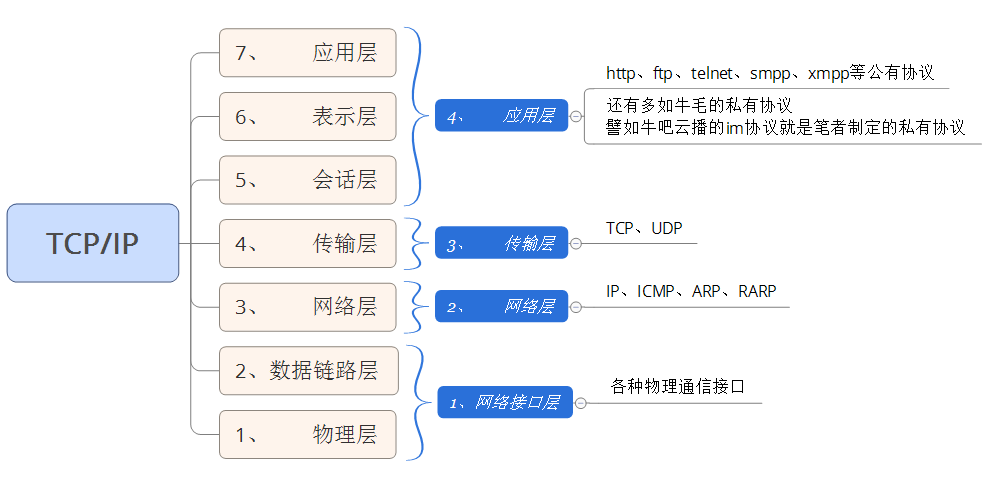


图 12 TCP/IP协议分层模型

本教程都按4层模型来讲解。Tcp/ip协议不在本教程讲解范围，但本教程会讲和应用层相关的部分tcp/ip知识。

## 应用层和传输层的数据传递

### 应用层数据是个什么鬼

以http协议为例，我们在访问一个网站时，浏览器会通过TCP协议发送如下字符串到服务器的应用层。

|  |
| --- |
| GET /test/abtest HTTP/1.1  Host: 127.0.0.1  Connection: keep-alive  Cache-Control: max-age=0  Upgrade-Insecure-Requests: 1  User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8  Accept-Encoding: gzip, deflate, br  Accept-Language: zh-CN,zh;q=0.9,en;q=0.8  Cookie: PHPSESSID=970260278652571648 |

程序调试截图(tio的HttpRequest.toString())：

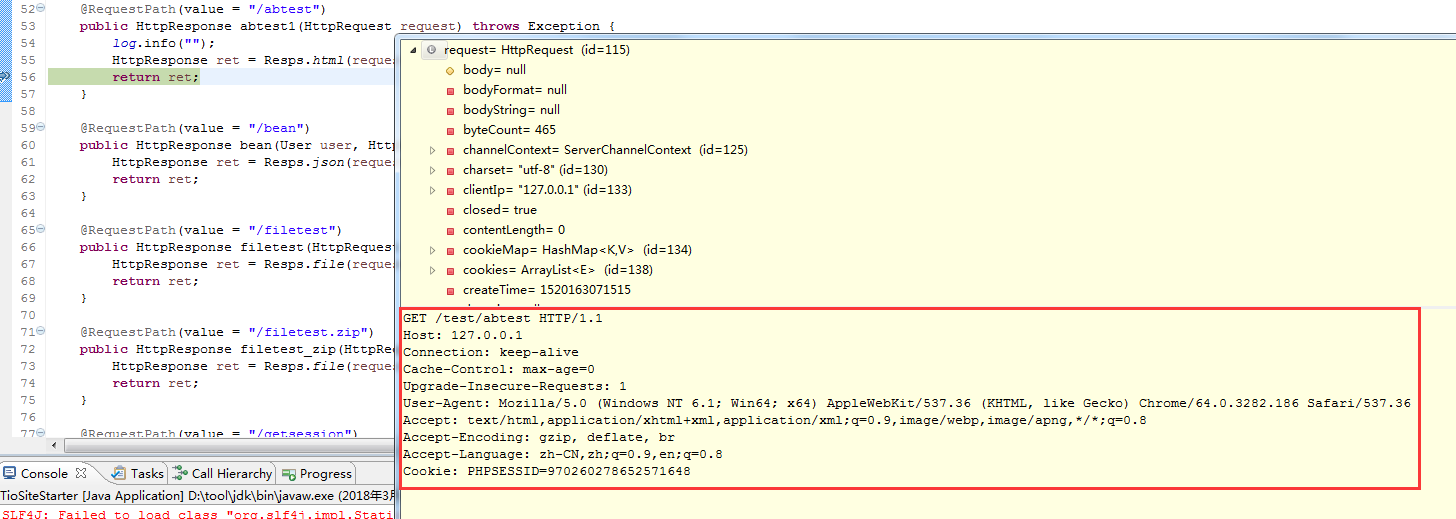


图 13 http请求数据演示

这些字符串就是应用层数据，应用层数据是按照一定格式来组织的，这个格式就是应用层协议，譬如http协议。

传输层在往应用层传递数据时，并不保证每次传递的数据是一个完整的应用层数据包（以http协议为例，就是并不保证应用层收到的数据刚好可以组成一个http包），这就是我们经常提到的半包和粘包。传输层只负责传递byte[]数据，应用层需要自己对byte[]数据进行解码，以http协议为例，就是把byte[]解码成http协议格式的字符串。



图 14 半包演示



图 15 粘包演示

### 应用层数据解码

应用层数据解码就是把传输层传递过来的byte[]数据解析成应用层协议指定的格式，以http协议为例，如下图所示

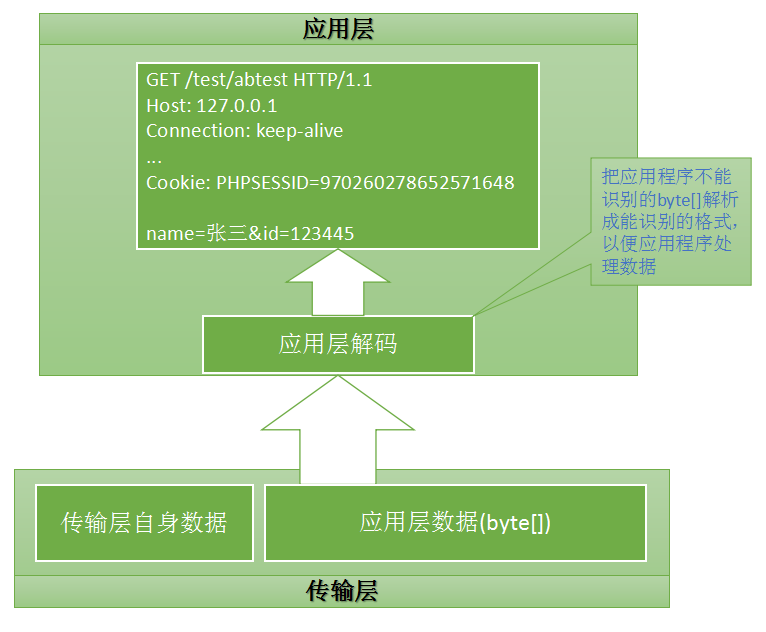


图 16 应用层解码

### 应用层数据编码

应用层数据编码与解码过程相反，是把已经的数据变成传输层可以使用的byte[]，以http协议为例，如下图所示

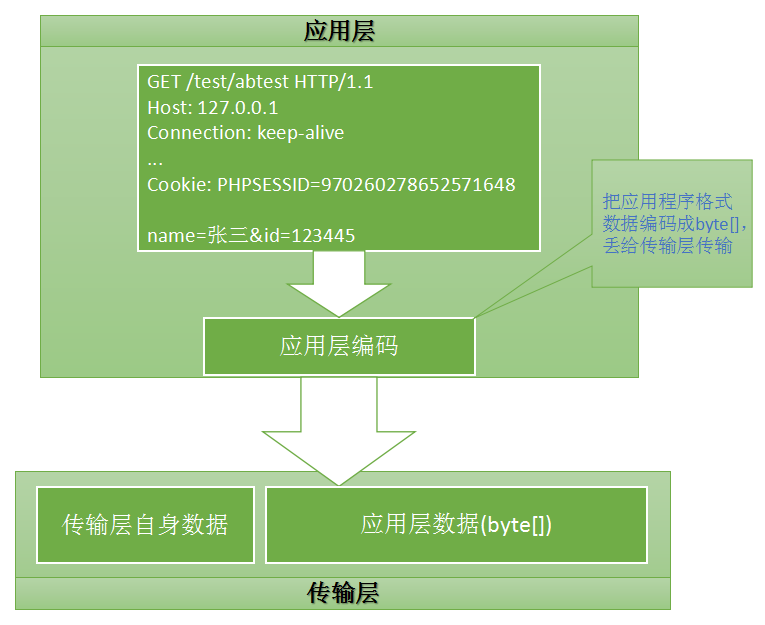


图 17 应用层编码

## 认识java中的bytebuffer

ByteBuffer是java nio和aio编程所必须掌握的一个数据结构，也是掌握tio所必须要学会的基础知识。

### 初识ByteBuffer

我们可以把bytebuffer理解成如下几个属性组成的一个数据结构

1、byte[] **bytes**: 用来存储数据

2、 int **capacity**: 用来表示bytes的容量，那么可以想像capacity就等于bytes.size()，此值在初始化bytes后，是不可变的。

3、int **limit**: 用来表示bytes实际装了多少数据，可以容易想像得到limit <= capacity，此值是可灵活变动的

4、int **position**: 用来表示在哪个位置开始往bytes写数据或是读数据，此值是可灵活变动的

通过下图，对bytebuffer形成一个感观认识吧

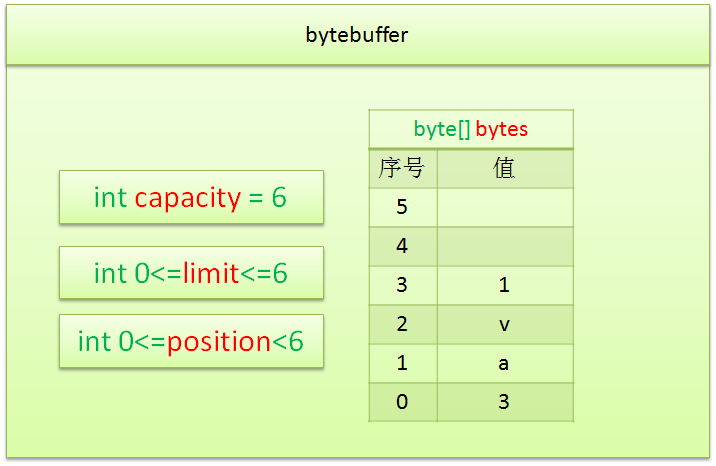


图 18 初识ByteBuffer

### 创建ByteBuffer

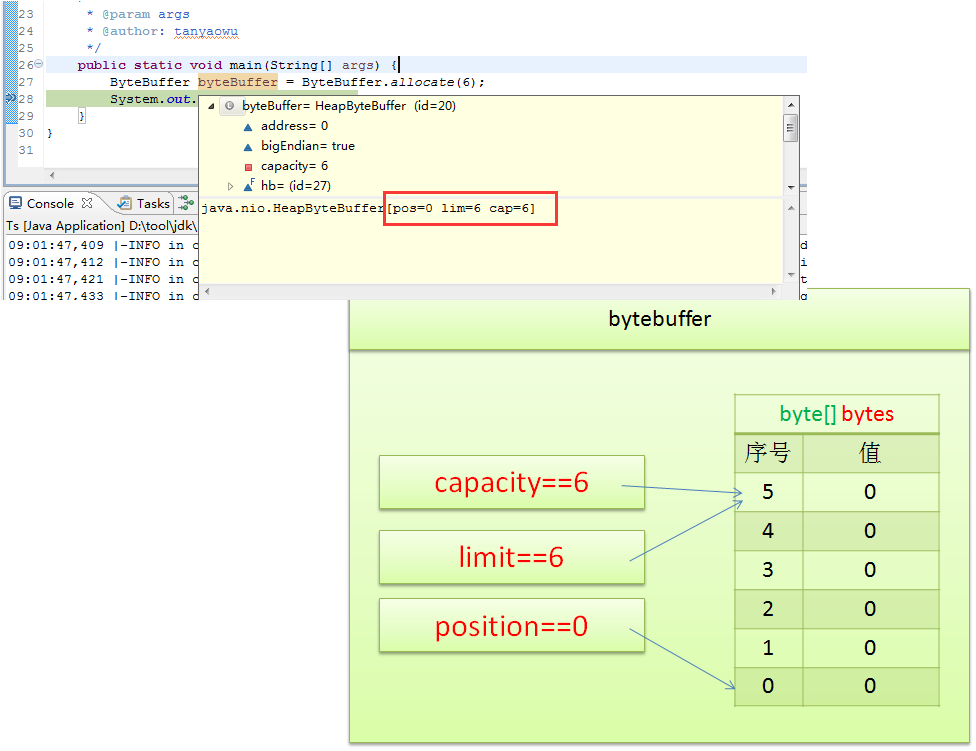


图 19 创建ByteBuffer

### 往ByteBuffer中写入数据

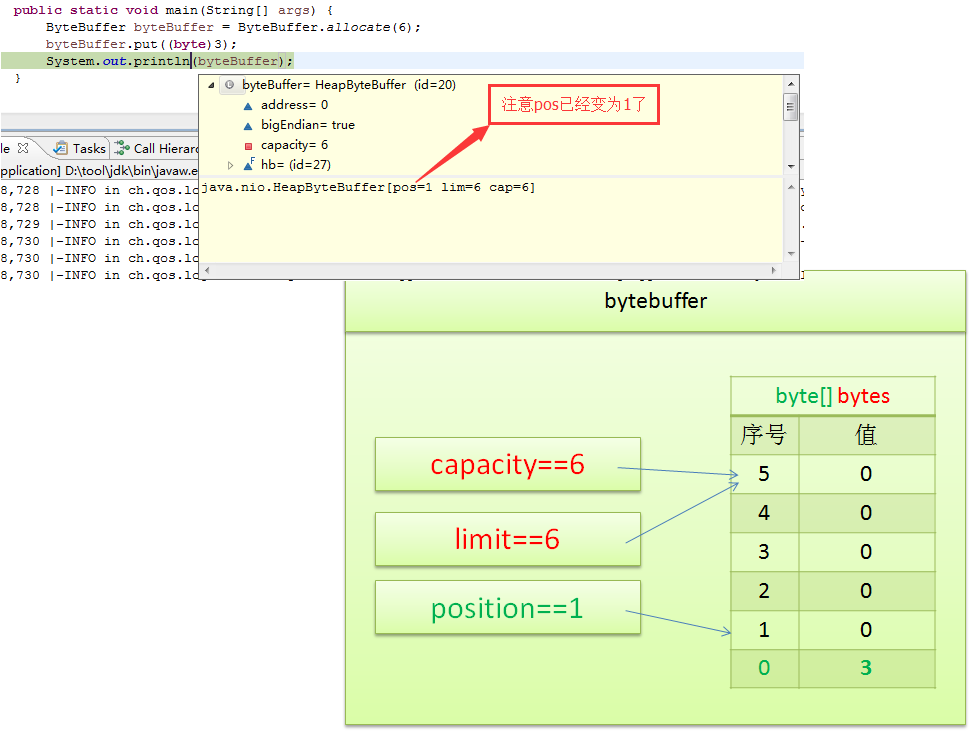


图 20 往ByteBuffer中写入数据

### 从ByteBuffer读取数据

对于刚刚写好的bytebuffer，我们要读取它的内容，需要先设置一下position和limit，否则读的位置就不对

|  |
| --- |
| byteBuffer.position(0); //设置position到0位置，这样读数据时就从这个位置开始读  byteBuffer.limit(1); //设置limit为1，表示当前bytebuffer的有效数据长度是1 |

我们看一下，设置position和limit后，bytebuffer的内部变化

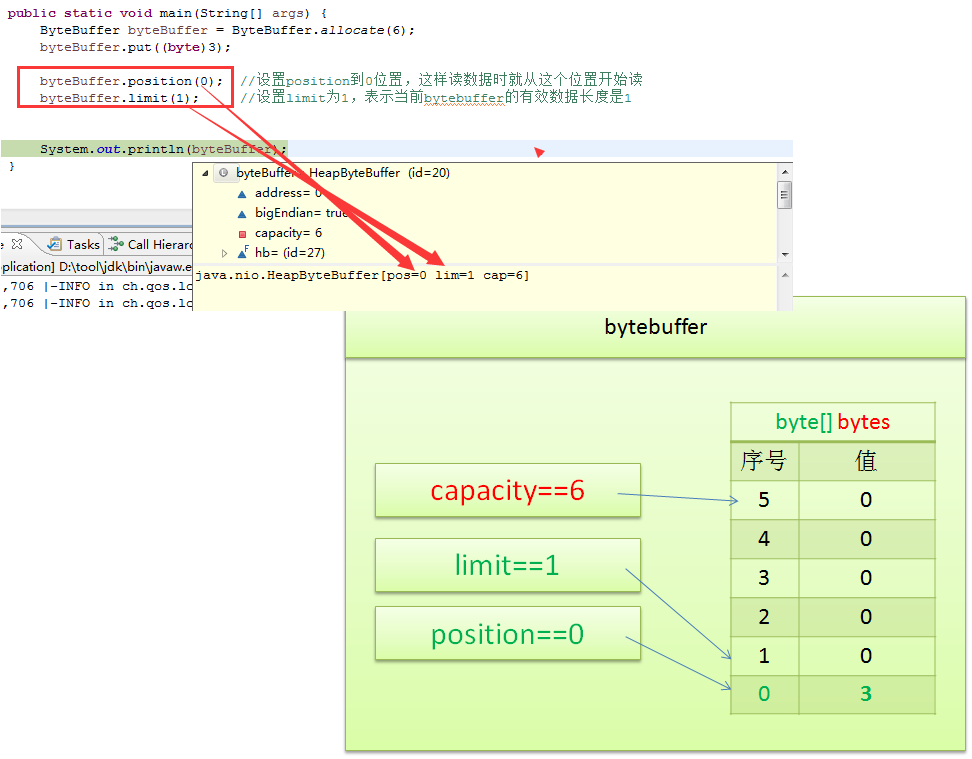


图 21 设置position和limit后，bytebuffer的内部变化

接下来，我们就可以读取刚才写入的数据了

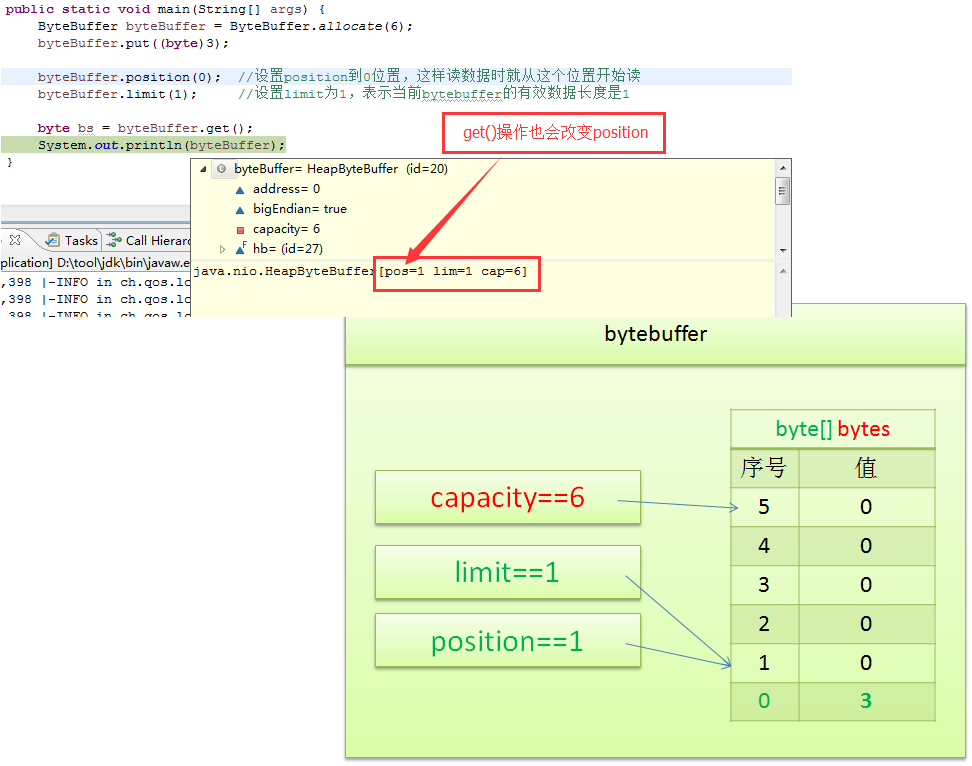


图 22 从ByteBuffer中读取数据

# 开启tio之旅

## Hello Tio

Hello world是个好东西，tio很早就有。

Tio在码云上很早就提供的hello world工程，很多人就是靠这个入门tio，它在/t-io/src/example/helloworld目录下：

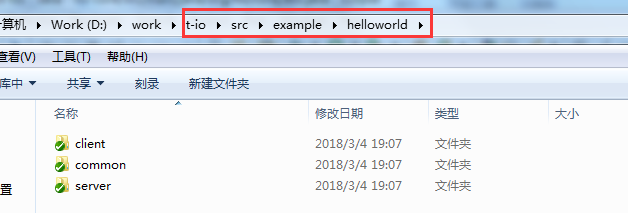


图 23 helloworld工程所在目录

虽然是个极简的helloworld，但笔者也分了三个子工程，这样便于一些用户把它当成一个简便的脚手架，套 于自己项目。

### hello world业务简介

本例子演示的是一个典型的TCP长连接应用，大体业务简介如下。

* 分为server和client工程，server和client共用common工程
* 服务端和客户端的消息协议比较简单，消息头为4个字节，用以表示消息体的长度，消息体为一个字符串的byte[]
* 服务端先启动，监听6789端口
* 客户端连接到服务端后，会主动向服务器发送一条消息
* 服务器收到消息后会回应一条消息
* 之后，框架层会自动从客户端发心跳到服务器，服务器也会检测心跳有没有超时（这些事都是框架做的，业务层只需要配一个心跳超时参数即可）
* 框架层会在断链后自动重连（这些事都是框架做的，业务层只需要配一个重连配置对象即可）

### 公共模块

#### 在pom.xml文件中引入tio-core

|  |
| --- |
| <dependency>  <groupId>org.t-io</groupId>  <artifactId>tio-core</artifactId>  <version>2.0.8.v20180205-RELEASE</version>  </dependency> |

#### 定义Packet

注：有时候服务器和客户端的业务消息包结构不一样，这种情况下，消息包的定义就不要放在公共模块中，而是在服务端和客户端分别定义

|  |
| --- |
| **package** org.tio.examples.helloworld.common;  **import** org.tio.core.intf.Packet;  /\*\*  \* **@author** tanyaowu  \*/  **public** **class** HelloPacket **extends** Packet {  **private** **static** **final** **long** ***serialVersionUID*** = -172060606924066412L;  **public** **static** **final** **int** ***HEADER\_LENGHT*** = 4;//消息头的长度  **public** **static** **final** String ***CHARSET*** = "utf-8";  **private** **byte**[] body;  /\*\*  \* **@return** the body  \*/  **public** **byte**[] getBody() {  **return** body;  }  /\*\*  \* **@param** body the body to set  \*/  **public** **void** setBody(**byte**[] body) {  **this**.body = body;  }  } |

#### 定义服务器端和客户端都用得到的常量

|  |
| --- |
| **package** org.tio.examples.helloworld.common;  /\*\*  \* **@author** tanyaowu  \* 2017年3月30日 下午7:05:54  \*/  **public** **interface** Const {  /\*\*  \* 服务器地址  \*/  **public** **static** **final** String ***SERVER*** = "127.0.0.1";    /\*\*  \* 监听端口  \*/  **public** **static** **final** **int** ***PORT*** = 6789;  /\*\*  \* 心跳超时时间  \*/  **public** **static** **final** **int** ***TIMEOUT*** = 5000;  } |

### 服务端代码

#### 实现org.tio.server.intf.ServerAioHandler

|  |
| --- |
| **package** org.tio.examples.helloworld.server;  **import** java.nio.ByteBuffer;  **import** org.tio.core.Aio;  **import** org.tio.core.ChannelContext;  **import** org.tio.core.GroupContext;  **import** org.tio.core.exception.AioDecodeException;  **import** org.tio.core.intf.Packet;  **import** org.tio.examples.helloworld.common.HelloPacket;  **import** org.tio.server.intf.ServerAioHandler;  /\*\*  \* **@author** tanyaowu  \*/  **public** **class** HelloServerAioHandler **implements** ServerAioHandler {  /\*\*  \* 解码：把接收到的ByteBuffer，解码成应用可以识别的业务消息包  \* 总的消息结构：消息头 + 消息体  \* 消息头结构： 4个字节，存储消息体的长度  \* 消息体结构： 对象的json串的byte[]  \*/  @Override  **public** HelloPacket decode(ByteBuffer buffer, ChannelContext channelContext) **throws** AioDecodeException {  **int** readableLength = buffer.limit() - buffer.position();  //收到的数据组不了业务包，则返回null以告诉框架数据不够  **if** (readableLength < HelloPacket.***HEADER\_LENGHT***) {  **return** **null**;  }  //读取消息体的长度  **int** bodyLength = buffer.getInt();  //数据不正确，则抛出AioDecodeException异常  **if** (bodyLength < 0) {  **throw** **new** AioDecodeException("bodyLength [" + bodyLength + "] is not right, remote:" + channelContext.getClientNode());  }  //计算本次需要的数据长度  **int** neededLength = HelloPacket.***HEADER\_LENGHT*** + bodyLength;  //收到的数据是否足够组包  **int** isDataEnough = readableLength - neededLength;  // 不够消息体长度(剩下的buffe组不了消息体)  **if** (isDataEnough < 0) {  **return** **null**;  } **else** //组包成功  {  HelloPacket imPacket = **new** HelloPacket();  **if** (bodyLength > 0) {  **byte**[] dst = **new** **byte**[bodyLength];  buffer.get(dst);  imPacket.setBody(dst);  }  **return** imPacket;  }  }  /\*\*  \* 编码：把业务消息包编码为可以发送的ByteBuffer  \* 总的消息结构：消息头 + 消息体  \* 消息头结构： 4个字节，存储消息体的长度  \* 消息体结构： 对象的json串的byte[]  \*/  @Override  **public** ByteBuffer encode(Packet packet, GroupContext groupContext, ChannelContext channelContext) {  HelloPacket helloPacket = (HelloPacket) packet;  **byte**[] body = helloPacket.getBody();  **int** bodyLen = 0;  **if** (body != **null**) {  bodyLen = body.length;  }  //bytebuffer的总长度是 = 消息头的长度 + 消息体的长度  **int** allLen = HelloPacket.***HEADER\_LENGHT*** + bodyLen;  //创建一个新的bytebuffer  ByteBuffer buffer = ByteBuffer.*allocate*(allLen);  //设置字节序  buffer.order(groupContext.getByteOrder());  //写入消息头----消息头的内容就是消息体的长度  buffer.putInt(bodyLen);  //写入消息体  **if** (body != **null**) {  buffer.put(body);  }  **return** buffer;  }    /\*\*  \* 处理消息  \*/  @Override  **public** **void** handler(Packet packet, ChannelContext channelContext) **throws** Exception {  HelloPacket helloPacket = (HelloPacket) packet;  **byte**[] body = helloPacket.getBody();  **if** (body != **null**) {  String str = **new** String(body, HelloPacket.***CHARSET***);  System.***out***.println("收到消息：" + str);  HelloPacket resppacket = **new** HelloPacket();  resppacket.setBody(("收到了你的消息，你的消息是:" + str).getBytes(HelloPacket.***CHARSET***));  Aio.*send*(channelContext, resppacket);  }  **return**;  }  } |

#### 启动类

|  |
| --- |
| **package** org.tio.examples.helloworld.server;  **import** java.io.IOException;  **import** org.tio.examples.helloworld.common.Const;  **import** org.tio.server.AioServer;  **import** org.tio.server.ServerGroupContext;  **import** org.tio.server.intf.ServerAioHandler;  **import** org.tio.server.intf.ServerAioListener;  /\*\*  \*  \* **@author** tanyaowu  \* 2017年4月4日 下午12:22:58  \*/  **public** **class** HelloServerStarter {  //handler, 包括编码、解码、消息处理  **public** **static** ServerAioHandler *aioHandler* = **new** HelloServerAioHandler();  //事件监听器，可以为null，但建议自己实现该接口，可以参考showcase了解些接口  **public** **static** ServerAioListener *aioListener* = **null**;  //一组连接共用的上下文对象  **public** **static** ServerGroupContext *serverGroupContext* = **new** ServerGroupContext("hello-tio-server", *aioHandler*, *aioListener*);  //aioServer对象  **public** **static** AioServer *aioServer* = **new** AioServer(*serverGroupContext*);  //有时候需要绑定ip，不需要则null  **public** **static** String *serverIp* = **null**;  //监听的端口  **public** **static** **int** *serverPort* = Const.***PORT***;  /\*\*  \* 启动程序入口  \*/  **public** **static** **void** main(String[] args) **throws** IOException {  *serverGroupContext*.setHeartbeatTimeout(org.tio.examples.helloworld.common.Const.***TIMEOUT***);  *aioServer*.start(*serverIp*, *serverPort*);  }  } |

### 客户端代码

#### 实现org.tio.client.intf.ClientAioHandler

|  |
| --- |
| **package** org.tio.examples.helloworld.client;  **import** java.nio.ByteBuffer;  **import** org.tio.client.intf.ClientAioHandler;  **import** org.tio.core.ChannelContext;  **import** org.tio.core.GroupContext;  **import** org.tio.core.exception.AioDecodeException;  **import** org.tio.core.intf.Packet;  **import** org.tio.examples.helloworld.common.HelloPacket;  /\*\*  \*  \* **@author** tanyaowu  \*/  **public** **class** HelloClientAioHandler **implements** ClientAioHandler {  **private** **static** HelloPacket *heartbeatPacket* = **new** HelloPacket();  /\*\*  \* 解码：把接收到的ByteBuffer，解码成应用可以识别的业务消息包  \* 总的消息结构：消息头 + 消息体  \* 消息头结构： 4个字节，存储消息体的长度  \* 消息体结构： 对象的json串的byte[]  \*/  @Override  **public** HelloPacket decode(ByteBuffer buffer, ChannelContext channelContext) **throws** AioDecodeException {  **int** readableLength = buffer.limit() - buffer.position();  //收到的数据组不了业务包，则返回null以告诉框架数据不够  **if** (readableLength < HelloPacket.***HEADER\_LENGHT***) {  **return** **null**;  }  //读取消息体的长度  **int** bodyLength = buffer.getInt();  //数据不正确，则抛出AioDecodeException异常  **if** (bodyLength < 0) {  **throw** **new** AioDecodeException("bodyLength [" + bodyLength + "] is not right, remote:" + channelContext.getClientNode());  }  //计算本次需要的数据长度  **int** neededLength = HelloPacket.***HEADER\_LENGHT*** + bodyLength;  //收到的数据是否足够组包  **int** isDataEnough = readableLength - neededLength;  // 不够消息体长度(剩下的buffe组不了消息体)  **if** (isDataEnough < 0) {  **return** **null**;  } **else** //组包成功  {  HelloPacket imPacket = **new** HelloPacket();  **if** (bodyLength > 0) {  **byte**[] dst = **new** **byte**[bodyLength];  buffer.get(dst);  imPacket.setBody(dst);  }  **return** imPacket;  }  }  /\*\*  \* 编码：把业务消息包编码为可以发送的ByteBuffer  \* 总的消息结构：消息头 + 消息体  \* 消息头结构： 4个字节，存储消息体的长度  \* 消息体结构： 对象的json串的byte[]  \*/  @Override  **public** ByteBuffer encode(Packet packet, GroupContext groupContext, ChannelContext channelContext) {  HelloPacket helloPacket = (HelloPacket) packet;  **byte**[] body = helloPacket.getBody();  **int** bodyLen = 0;  **if** (body != **null**) {  bodyLen = body.length;  }  //bytebuffer的总长度是 = 消息头的长度 + 消息体的长度  **int** allLen = HelloPacket.***HEADER\_LENGHT*** + bodyLen;  //创建一个新的bytebuffer  ByteBuffer buffer = ByteBuffer.*allocate*(allLen);  //设置字节序  buffer.order(groupContext.getByteOrder());  //写入消息头----消息头的内容就是消息体的长度  buffer.putInt(bodyLen);  //写入消息体  **if** (body != **null**) {  buffer.put(body);  }  **return** buffer;  }    /\*\*  \* 处理消息  \*/  @Override  **public** **void** handler(Packet packet, ChannelContext channelContext) **throws** Exception {  HelloPacket helloPacket = (HelloPacket) packet;  **byte**[] body = helloPacket.getBody();  **if** (body != **null**) {  String str = **new** String(body, HelloPacket.***CHARSET***);  System.***out***.println("收到消息：" + str);  }  **return**;  }  /\*\*  \* 此方法如果返回null，框架层面则不会发心跳；如果返回非null，框架层面会定时发本方法返回的消息包  \*/  @Override  **public** HelloPacket heartbeatPacket() {  **return** *heartbeatPacket*;  }  } |

#### 启动类

|  |
| --- |
| **package** org.tio.examples.helloworld.client;  **import** org.tio.client.AioClient;  **import** org.tio.client.ClientChannelContext;  **import** org.tio.client.ClientGroupContext;  **import** org.tio.client.ReconnConf;  **import** org.tio.client.intf.ClientAioHandler;  **import** org.tio.client.intf.ClientAioListener;  **import** org.tio.core.Aio;  **import** org.tio.core.Node;  **import** org.tio.examples.helloworld.common.Const;  **import** org.tio.examples.helloworld.common.HelloPacket;  /\*\*  \*  \* **@author** tanyaowu  \*  \*/  **public** **class** HelloClientStarter {  //服务器节点  **public** **static** Node *serverNode* = **new** Node(Const.***SERVER***, Const.***PORT***);  //handler, 包括编码、解码、消息处理  **public** **static** ClientAioHandler *aioClientHandler* = **new** HelloClientAioHandler();  //事件监听器，可以为null，但建议自己实现该接口，可以参考showcase了解些接口  **public** **static** ClientAioListener *aioListener* = **null**;  //断链后自动连接的，不想自动连接请设为null  **private** **static** ReconnConf *reconnConf* = **new** ReconnConf(5000L);  //一组连接共用的上下文对象  **public** **static** ClientGroupContext *clientGroupContext* = **new** ClientGroupContext(*aioClientHandler*, *aioListener*, *reconnConf*);  **public** **static** AioClient *aioClient* = **null**;  **public** **static** ClientChannelContext *clientChannelContext* = **null**;  /\*\*  \* 启动程序入口  \*/  **public** **static** **void** main(String[] args) **throws** Exception {  *clientGroupContext*.setHeartbeatTimeout(Const.***TIMEOUT***);  *aioClient* = **new** AioClient(*clientGroupContext*);  *clientChannelContext* = *aioClient*.connect(*serverNode*);  //连上后，发条消息玩玩  *send*();  }  **private** **static** **void** send() **throws** Exception {  HelloPacket packet = **new** HelloPacket();  packet.setBody("hello world".getBytes(HelloPacket.***CHARSET***));  Aio.*send*(*clientChannelContext*, packet);  }  } |

### 运行hello tio

* 运行服务器：org.tio.examples.helloworld.server.HelloServerStarter.main(String[])

控制台应该会打印如下日志：

|  |
| --- |
| 2018-03-18 19:36:25,608 WARN org.tio.server.AioServer[109]: hello-tio-server started, listen on 0.0.0.0:6789 |

* 运行客户端：org.tio.examples.helloworld.client.HelloClientStarter.main(String[])

控制台应该会打印如下日志：

|  |
| --- |
| 2018-03-18 19:36:27 INFO o.t.c.ConnectionCompletionHandler[100]: connected to 127.0.0.1:6789  收到消息：收到了你的消息，你的消息是:hello world  2018-03-18 19:36:28 INFO org.tio.client.AioClient[370]: [1]: curr:1, closed:0, received:(1p)(55b), handled:1, sent:(1p)(15b)  2018-03-18 19:36:30 INFO org.tio.client.AioClient[370]: [1]: curr:1, closed:0, received:(1p)(55b), handled:1, sent:(1p)(15b)  2018-03-18 19:36:31 INFO org.tio.client.AioClient[364]: 0:0:0:0:0:0:0:0:9084发送心跳包  2018-03-18 19:36:31 INFO org.tio.client.AioClient[370]: [1]: curr:1, closed:0, received:(1p)(55b), handled:1, sent:(1p)(15b)  2018-03-18 19:36:32 INFO org.tio.client.AioClient[370]: [1]: curr:1, closed:0, received:(1p)(55b), handled:1, sent:(2p)(19b)  2018-03-18 19:36:33 INFO org.tio.client.AioClient[364]: 0:0:0:0:0:0:0:0:9084发送心跳包 |

同时，服务器端的控制台会出现类似下面的日志

|  |
| --- |
| 2018-03-18 19:36:25,608 WARN org.tio.server.AioServer[109]: hello-tio-server started, listen on 0.0.0.0:6789  收到消息：hello world  2018-03-18 19:38:25,654 INFO o.t.server.ServerGroupContext[219]:  hello-tio-server  ├ 当前时间:1521373105587  ├ 连接统计  │ ├ 共接受过连接数 :1  │ ├ 当前连接数 :1  │ ├ 异IP连接数 :1  │ └ 关闭过的连接数 :0  ├ 消息统计  │ ├ 已处理消息 :44  │ ├ 已接收消息(packet/byte):44/187  │ ├ 已发送消息(packet/byte):1/55b  │ ├ 平均每次TCP包接收的字节数 :4.25  │ └ 平均每次TCP包接收的业务包 :1.0  └ IP统计时段  └ []  ├ 节点统计  │ ├ clientNodes :1  │ ├ 所有连接 :1  │ ├ 活动连接 :1  │ ├ 关闭次数 :0  │ ├ 绑定user数 :0  │ ├ 绑定token数 :0  │ └ 等待同步消息响应 :0  ├ 群组  │ ├ channelmap :0  │ └ groupmap:0  └ 拉黑IP  └ []  2018-03-18 19:38:25,655 INFO o.t.server.ServerGroupContext[273]: hello-tio-server, 检查心跳, 共1个连接, 取锁耗时0ms, 循环耗时70ms, 心跳超时时间:5000ms |

## Tio常见类介绍

### ChannelContext(通道上下文)

每一个tcp连接的建立都会产生一个ChannelContext对象，这是个抽象类，如果你是用tio作tcp客户端，那么就是ClientChannelContext，如果你是用tio作tcp服务器，那么就是ServerChannelContext。

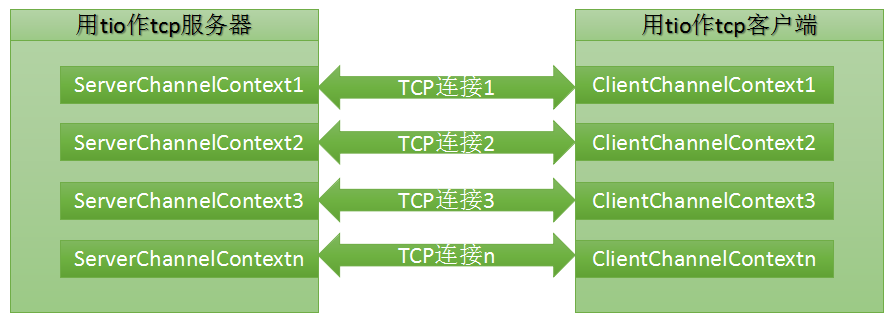


图 24 ChannelContext概念

ChannelContext对象包含的信息非常多，主要对象见下图



图 25 ChannelContext主要对象

#### ServerChannelContext

ChannelContext的子类，当用tio作tcp服务器时，业务层接触的是这个类的实例。

#### ClientChannelContext

ChannelContext的子类，当用tio作tcp客户端时，业务层接触的是这个类的实例。

### GroupContext(服务配置与维护)

我们在写TCP Server时，都会先选好一个端口以监听客户端连接，再创建N组线程池来执行相关的任务，譬如发送消息、解码数据包、处理数据包等任务，还要维护客户端连接的各种数据，为了和业务互动，还要把这些客户端连接和各种业务数据绑定起来，譬如把某个客户端绑定到一个群组，绑定到一个userid，绑定到一个token等。GroupContext就是用来配置线程池、确定监听端口，维护客户端各种数据等的。

GroupContext是个抽象类，如果你是用tio作tcp客户端，那么你需要创建ClientGroupContext，如果你是用tio作tcp服务器，那么你需要创建ServerGroupContext

GroupContext对象包含的信息非常多，主要对象见下图

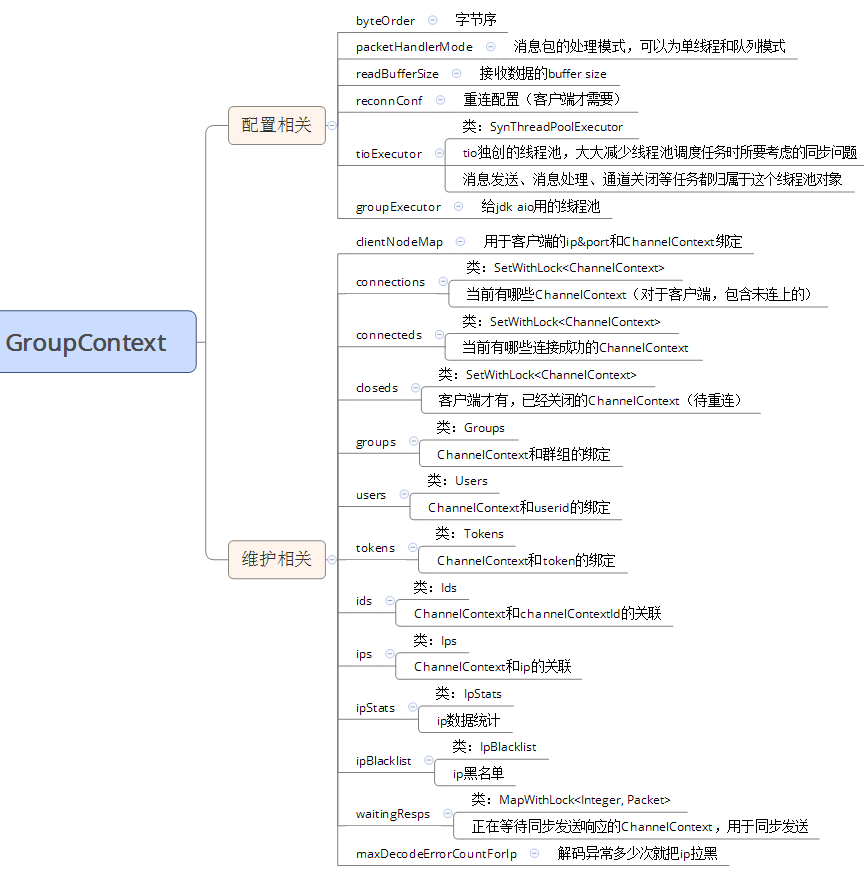


图 26 GroupContext主要对象

#### ServerGroupContext

GroupContext的子类，当用tio作tcp服务器时，业务层接触的是这个类的实例。

#### ClientGroupContext

GroupContext的子类，当用tio作tcp客户端时，业务层接触的是这个类的实例。

### AioHandler(消息处理接口)

AioHandler是处理消息的核心接口，它有两个子接口，ClientAioHandler和ServerAioHandler，当用tio作tcp客户端时需要实现ClientAioHandler，当用tio作tcp服务器时需要实现ServerAioHandler，它主要定义了3个方法，见下图



图 27 AioHandler接口

#### ServerAioHandler

AioHandler的子类，当用tio作tcp服务器时，业务层需要实现该接口。

#### ClientAioHandler

AioHandler的子类，当用tio作tcp客户端时，业务层需要实现该接口。

### AioListener(通道监听者)

AioListener是处理消息的核心接口，它有两个子接口，ClientAioListener和ServerAioListener，当用tio作tcp客户端时需要实现ClientAioListener，当用tio作tcp服务器时需要实现ServerAioListener，它主要定义了如下方法



图 28 AioListener接口

#### ServerAioListener

AioListener的子类，当用tio作tcp服务器时，业务层实现该接口。

#### ClientAioListener

AioListener的子类，当用tio作tcp客户端时，业务层实现该接口。

### Packet(应用层数据包)

TCP层过来的数据，都会被tio要求解码成Packet对象，应用都需要继承这个类，从而实现自己的业务数据包。

请回忆一下前面讲的TCP/IP协议分层模型

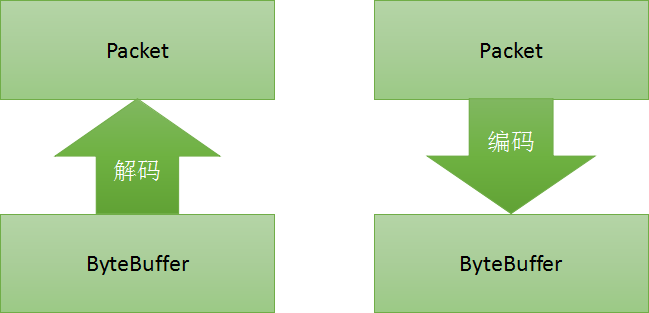


图 29 Packet和ByteBuffer转换

### AioServer（tio服务端入口类）

tio服务端入口类，主要对象见下图

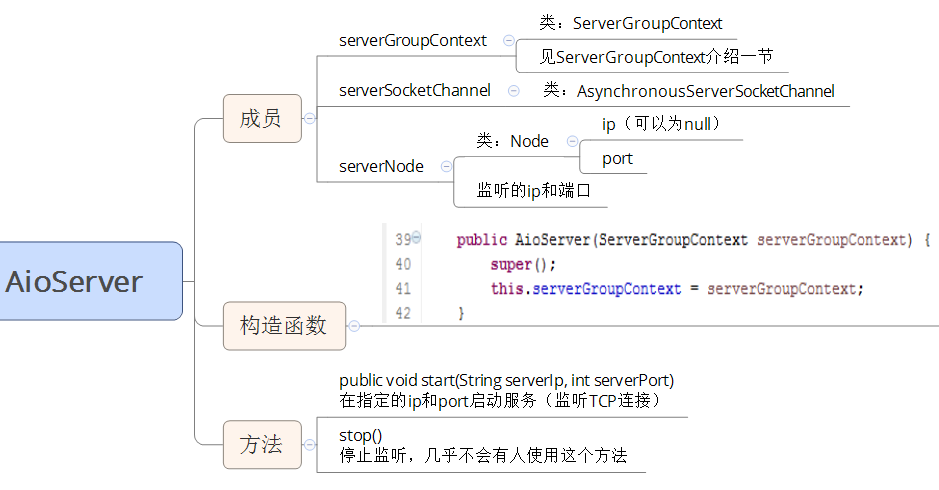


图 30 AioServer对象

### AioClient（tio客户端入口类）

tio客户端入口类，主要对象见下图



图 31 AioClient对象

### ObjWithLock（自带读写锁的对象）

网络编程中会伴随大量并发操作，大家对ConcurrentModificationException一定不会陌生，这个是典型的并发操作集合引发的异常。为了更好的处理并发，tio自创了一个ObjWithLock对象，这个对象很简单，但给并发编程带来了极大的方便，如果您阅读过tio源代码，相信已经体会到这个对象在tio中是无处不在的。ObjWithLock顾名思义，它就是一个自带了一把（读写）锁的普通对象（一般是集合对象），每当要对这个对象进行同步安全操作（并发下对集合进行遍历或对集合对象进行元素修改删除增加）时，就得用这个锁。

ObjWithLock对象部分源代码见下图：



图 32 ObjWithLock

为了更便捷地操作，tio提供了三个ObjWithLock子类，见下图

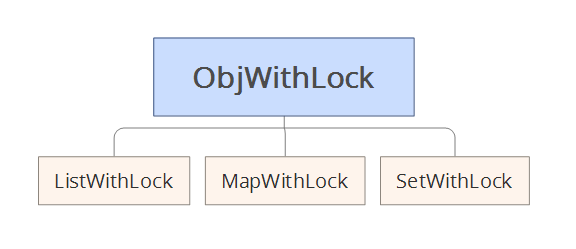


图 33 ObjWithLock的子类

ListWithLock里面有Obj就是List对象，MapWithLock里面有Obj就是Map对象，SetWithLock里面有Obj就是Set对象。

掌握XxxWithLock这些对象，我觉得最好的方法是看个例子，相信读者看了例子能悟出作者的意图，例子如下：



图 34 SetWithLock例子



图 35 MapWithLock例子

上面的例子，都是先拿到相应的锁（根据业务需要获取读锁或写锁，如果只是读取数据，则获取读锁，如果需要对集合进行修改，则获取写锁），然后【 lock()-->业务处理-->unlock() 】，注意一定要在try块进行lock()，在finally块中进行unlock()操作，这样可以保证一个获取锁到释放锁形成一个原子操作。