

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: 文档在编写中, 进度不会太快

纸质版

考虑到本教程并不一定能完成，所以请**慎买**。

购买地址：<http://t-io.org/tb.html>



图 1 《t-io 使用手册》纸质版

捐 赠



t-io特点

- 1 极简洁、清晰、易懂的API**
 原生态bytebuffer减少学习成本，又减少各种中间对象的创建，只需花30分钟学习helloworld，就能较好地掌握并实现。
- 2 极震撼的性能**
 单机轻松支持百万级tcp长连接，彻底用开业界C1000K瓶颈，最高每秒可收发500万条业务消息，约165M。
- 3 对开发人员极体贴的内置功能**
 内置心跳检测
 内置心跳发送
 各种便捷的绑定API
 各种便捷的发送API
 一行代码拥有自动重连功能
 各项消息统计等功能，全部一键内置搞定，省却各种烦恼。

码云最有价值开源项目

[访问项目](#)


商业服务

没人融资的产品不是好产品，没有商业关注的项目不是好项目，t-io 迫于存活现实，目前推出一些互利共赢的商业服务。

t-io 提供此类商业服务，一方面可以让技术选型人员毫无后顾之忧，另一方面可以帮助创业公司快速搭建一流的技术基础平台。

1.1. 服务项目

1. 商业级网络接入层脚手架

内容： t-io 官方提供商业级脚手架服务，帮助您快速打造百万级网络接入服务，并提供长期技术咨询服务，让你无后顾之忧。注：此服务不包括应用层协议的实现。

价格： 2.4 万（相当于聘请月薪 48K 的架构师半个月，项目风险和时间成本却大大减少）。

2. 私有协议实现

内容： 很多朋友并不知道如何实现私有协议，或者说是不知道如何实现安全的私有协议（很多业余开发人员在实现协议时，会有各种漏洞，容易被攻击），t-io 官方可以代劳（视协议难易，有的协议不一定会接）。

价格： 根据协议难易程度而定，这类项目一般是根据工时来定的，算是苦力活，所以有的协议不一定会接。

3. 大型直播平台 and IM 技术咨询

内容： t-io 有丰富的直播平台搭建经验和一些 IM 经验。目前业余时间提供技术咨询服务。

价格： 1 万起步，看需要提供什么样的服务了。

4. 自用版 t-io 代码以及技术支持

内容： t-io 自用版代码，并提供技术支持。自用版 t-io 的代码多了**集群支持**和**拉黑功能**，代码和开源版相差不大，所以本项服务，更多的是技术支持，让你无后顾之忧（就是锅由 t-io 来背）！

价格： 1.5 万

5. 疑难技术问题解决

内容： 各类 java 相关的技术问题，t-io 丰富的开发经验和广泛的技术人脉，也许可以帮助您解决燃眉之急。

价格： 电话沟通或者面议。

1.2. 商业案例

被服务方	服务内容	价格	备注
氩氮云	1、网络接入层专业级脚手架	----	http://www.hekr.me
	2、定制开发的统计数据		由于是第一次商业服务，所以价格很友好，更多的是交
	3、技术支持		个朋友

表 1 商业服务案例

目 录

封 面.....	1
纸质版.....	2
捐 赠.....	3
商业服务.....	4
1.1. 服务项目.....	4
1. 商业级网络接入层脚手架.....	4
2. 私有协议实现.....	4
3. 大型直播平台 IM 技术咨询.....	4
4. 自用版 t-io 代码以及技术支持.....	4
5. 疑难技术问题解决.....	4
1.2. 商业案例.....	5
目 录.....	6
图 表.....	8
第一章 t-io 简介.....	9
1.3. t-io 是啥.....	9
1.4. t-io 历史.....	9
1.5. t-io 适用场景.....	10
1.6. t-io 案例.....	10
1.7. t-io 性能.....	11
1.8. t-io 稳定性.....	12
1.9. t-io 生态.....	13
1.10. t-io 荣誉.....	16
6. 第一批码云最有价值开源项目.....	17
7. 2017 年最受欢迎开源软件上榜.....	17
8. 2017 年热门开源项目 Star 数第 3, Fork 数第 5.....	18
1.11. t-io 分支.....	19
第二章 预备知识.....	20
1.1. TCP/IP 协议分层模型.....	20
1.2. 应用层和传输层的数据传递.....	20
1. 应用层数据是个什么鬼.....	20
2. 应用层数据解码.....	22
3. 应用层数据编码.....	23
1.3. 认识 java 中的 ByteBuffer.....	24
1. 初识 ByteBuffer.....	24
2. 创建 ByteBuffer.....	26
3. 往 ByteBuffer 中写入数据.....	27
4. 从 ByteBuffer 读取数据.....	27
第三章 开启 t-io 之旅.....	30
1.1. Hello Tio.....	30
1. 业务简介.....	30
2. 公共模块.....	31
3. 服务端代码.....	33
4. 客户端代码.....	38

5. 运行 hello tio.....	44
1.2. Tio 常见类介绍.....	46
1. ChannelContext(通道上下文).....	46
2. GroupContext(服务配置与维护).....	47
3. AioHandler(消息处理接口).....	48
4. AioListener(通道监听者).....	49
5. Packet(应用层数据包).....	50
6. AioServer (tio 服务端入口类)	51
7. AioClient (tio 客户端入口类)	51
8. ObjWithLock (自带读写锁的对象)	52

图 表

表 1 商业服务案例.....5

图 1 《tio 使用手册》纸质版.....2

图 2 tio 历史.....9

图 3 tio 常见的使用场景.....10

图 4 tio 案例展示.....11

图 5 t-io 30W 长连接并发压力测试报告.....12

图 6 tio 官网运行时间.....13

图 7 tio 在 OSC 上的收藏数、评论数.....14

图 8 tio 在码云上的后台统计截图.....15

图 9 tio 受捐次数.....16

图 10 首批码云最有价值开源项目.....17

图 11 2017 年最受欢迎开源软件上榜.....18

图 12 2017 年热门开源项目 Star 数第 3，Fork 数第 5.....18

图 13 tio 自用版和开源版的差异.....19

图 14 TCP/IP 协议分层模型.....20

图 15 http 请求数据演示.....21

图 16 半包演示.....22

图 17 粘包演示.....22

图 18 应用层解码.....23

图 19 应用层编码.....24

图 20 初识 ByteBuffer.....25

图 21 创建 ByteBuffer.....26

图 22 往 ByteBuffer 中写入数据.....27

图 23 设置 position 和 limit 后，bytebuffer 的内部变化.....28

图 24 从 ByteBuffer 中读取数据.....29

图 25 helloworld 工程所在目录.....30

图 26 ChannelContext 概念.....46

图 27 ChannelContext 主要对象.....47

图 28 GroupContext 主要对象.....48

图 29 AioHandler 接口.....49

图 30 AioListener 接口.....50

图 31 Packet 和 ByteBuffer 转换.....51

图 32 AioServer 对象.....51

图 33 AioClient 对象.....52

图 34 ObjWithLock.....53

图 35 ObjWithLock 的子类.....54

图 36 SetWithLock 例子.....54

图 37 MapWithLock 例子.....55

图 38 锁操作 4 步曲.....56

表 1 商业服务案例.....5

第一章 tio 简介

1.3. 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。

1.4. tio 历史

笔者参与了中兴网管系统项目，其中有个EMF模块，是负责和各刀片进行网络通讯的，由于各种原因，领导（梅贤昌，目前已入职华为）要求全部重写EMF模块（也就是和刀片进行通讯的那一部分代码），而且不允许用mina（考虑到mina的各种坑，证明领导有多英明）

2010

在此背景下，笔者对java的nio进行了一翻细致的学习，重写了EMF模块。这时候并没有形成nio框架，通讯相关的代码和业务绑定较紧，不具有复用性

2011

今天（2018年3月3号）询问仍在职的前中兴同事李某（由于信息安全不便写名字），中兴仍在使用这版EMF

2012

在离开中兴后，笔者觉得这个通讯模块很有价值，用开发EMF积累的知识，花了2个月时间写了talent-nio框架

2014

热波直播平台，底层用的便是talent-nio。

在进行热波直播平台开发过程中，感觉到一些常用的业务功能，譬如群发、用户绑定等完全可以在框架内进行提供，不必在业务层去实现，于是想进一步改造talent-nio，在改造过程中笔者又学习了java aio，感觉aio提供的api要比nio更友好

2016

放弃了原来的talent-nio，直接使用java aio开发了talent-aio（注意一个是talent-nio一个是talent-aio），并在码云上开源了talent-aio（2016年11月19号）

2017

在和用户交流，每次都要用键盘敲出talent-aio这十个字母挺费事儿，于是把talent-aio改名为t-io（2017年4月1号）

2018

笔者用tio全家桶开发了一个直播平台——牛吧云播（<https://www.nb350.com>）

tio全家桶

- tio-core（客户端的TCP长连接）
- tio-websocket-server（浏览器中的websocket连接）
- tio-http-server（取代tomcat、jetty等传统web容器）
- tio-webpack（用于页面资源的压缩、缓存、渲染等）

图 2 tio 历史

1.5. 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。

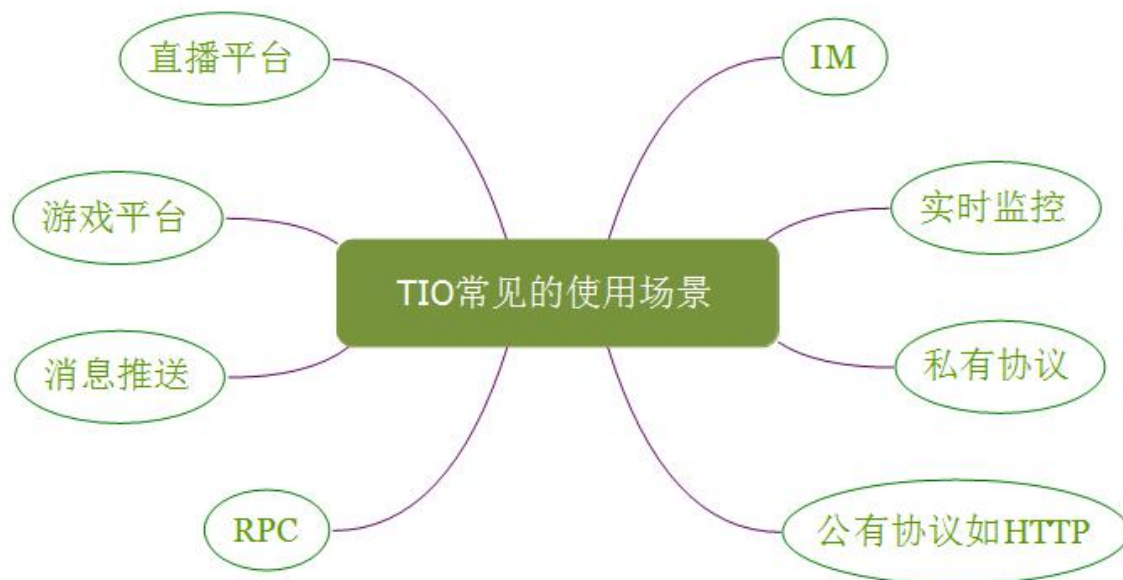




图 3 tio 常见的使用场景

1.6. 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。

 [首页](#) [案例](#) [文档](#) [提问](#) [捐赠](#)




牛吧云播

33745965 2018-02-08 09:14:40

牛吧云播是一个大型直播平台，包括直播、视频、小游戏等业务，该平台的im是基于tio-core和tio-websocket完成的，http api服务是基于tio-http和tio-mvc完成的，http静态页面是基于tio-webpack完成的

[去看看](#)

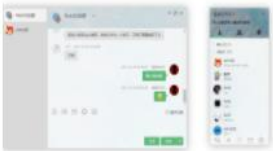


贝密游戏

5622928 2018-04-05 12:22:26

贝密游戏是一系列棋牌游戏的名称，其中包含麻将、斗地主、德州，目前正在进行UI设计以及后台系统（JAVA）开发！

[去看看](#)




陆离Chat

1719411461 2017-12-20 16:46:19

一个Nutz为后台支撑,T-io为通讯支持,LayIM为前台UI交互的纯国产框架开发的一个即时通讯项目

[去看看](#)



中网移动CMPP3

270249250 2018-02-26 16:45:21

tio实现的中网移动CMPP3

[未提供地址](#)

<

1

2

3

4

5

6

7

>

4

跳至

页

图 4 tio 案例展示

1.7. 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>)

```
cpu2 4843 128 7913 648005 214 0 3312 0 0 0
cpu3 5046 198 3151 659423 413 0 22 0 0 0

jvm

root@Rain-ubuntu:/home/ubuntu/software/nlife-1.0.0-release# jstat -gcutil 6203
S0    S1     E      O      M      CCS    YGC    YGCT   FGC    FGCT   GCT
0.00  98.10  30.07  92.71  96.47  88.16   315    7.045   14     7.570  14.616
root@Rain-ubuntu:/home/ubuntu/software/nlife-1.0.0-release# ps auxlgrep java
root      6203  4.5 15.7 5054900 1202688 pts/7  Sl   11:23   2:45 java -jar nlife-1.0.0.jar

root@Rain-ubuntu:/home/ubuntu/software/nlife-1.0.0-release# jstat -gccapacity 6203
NGCMN  NGOMX   NGC    S0C    S1C     EC    DGMN   DGMX   DGC      OC      MCMN   MCMX   MC    CCSMN  CCSMX
39936.0 634880.0 55808.0 26112.0 11776.0 3584.0 80896.0 1269760.0 1073664.0 1073664.0 0.0 1060864.0 13440.0 0.0 104
8576.0 1664.0 315    14
```

2小时后观察内存变化情况如下图

```
ubuntu@Rain-ubuntu:~$ free -m
              total        used         free       shared  buff/cache       available
Mem:           7433          2089          2702           244          2642          3974
Swap:          7632             0          7632
```

结论如下

- 1：0-10万连接。内存变化较大。主要是由于内存初始化时jvm会占用一部分内存。

2：10万之后。每增加10万连接 内存占用率上升300M左右。

3：随着连接数增加可以发现cpu性能对连接数影响不大。

4：新生代内存与老生代内存占用率比较合理。

5：并发30W连接在2小时内。内存变化不明显。处于合理状态

根据以上数据可以推论 --> 以8G内存(可使用内存为7.5G左右)为例:当内存占用率达到5G左右时足以支撑100W并发连接。

© 著作权归作者所有

分类：工作日志 字数：361

标签： t-io

图 5 t-io 30W 长连接并发压力测试报告

1.8. 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 2111 小时（88 天），该时间仍在延长，用 JVisualVM 监控出来的内存和 CPU 仍十分漂亮。

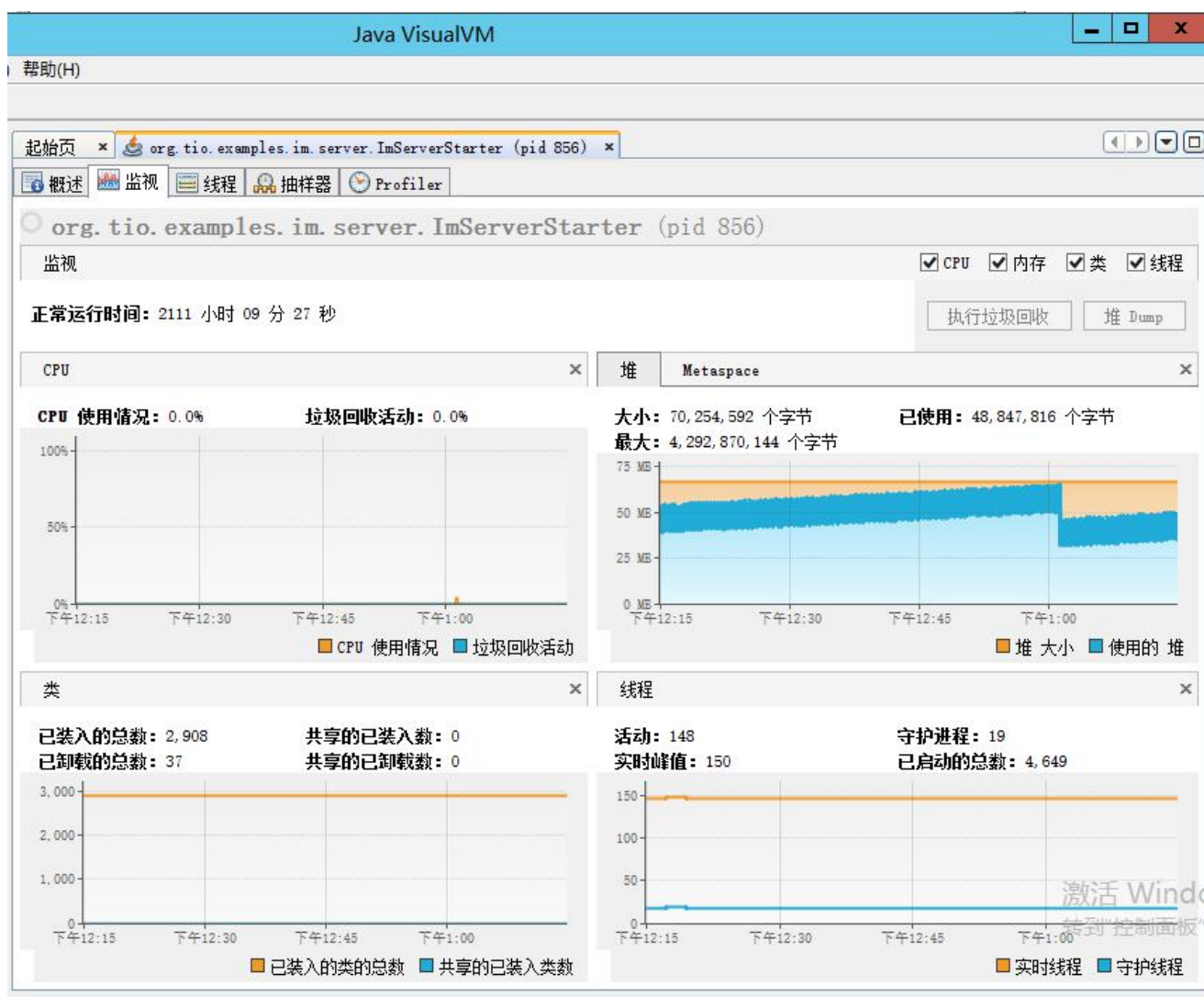


图 6 tio 官网运行时间

4、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。

1.9. tio 生态

tio 生态已经自动形成，可以参看下面三张截图

让网络编程更轻松和有趣 t-io [推荐] [国产]

t-io —— 让网络编程更轻松和有趣 t-io 诞生的意义 旧时王谢堂前燕，飞入寻常百姓家----当年那些王谢贵族们才拥有的"百万级即时通讯"应用，将因为 t-io 的诞生，纷纷飞入普通



上次更新: 2018年03月26日 收藏 1727 评论 126 评分 9.2

图 7 t-io 在 OSC 上的收藏数、评论数

ache-2.0 GVP

📁 捐赠 10

👁 Unwatch 1.5k

☆ Unstar 3.6k

🔗 Fork 1.2k

🔗 Pull Requests 0

📎 附件 0

📖 Wiki 0

📊 统计

⚙ 服务 ▾

👤 管理

项目访问统计 (IP)

由于码云使用DDOS高防IP，IP 统计可能不太精确。

统计类型	昨日	今天	本周	本月	本年	全部
IP	471	818	1285	4698	17851	71336
PULL	96	133	229	1094	3191	18355
PUSH	1	1	2	10	27	124
Download ZIP	65	100	165	547	1719	10471

☒ 每周接收统计邮件（需在设置同时开启每周精选通知）

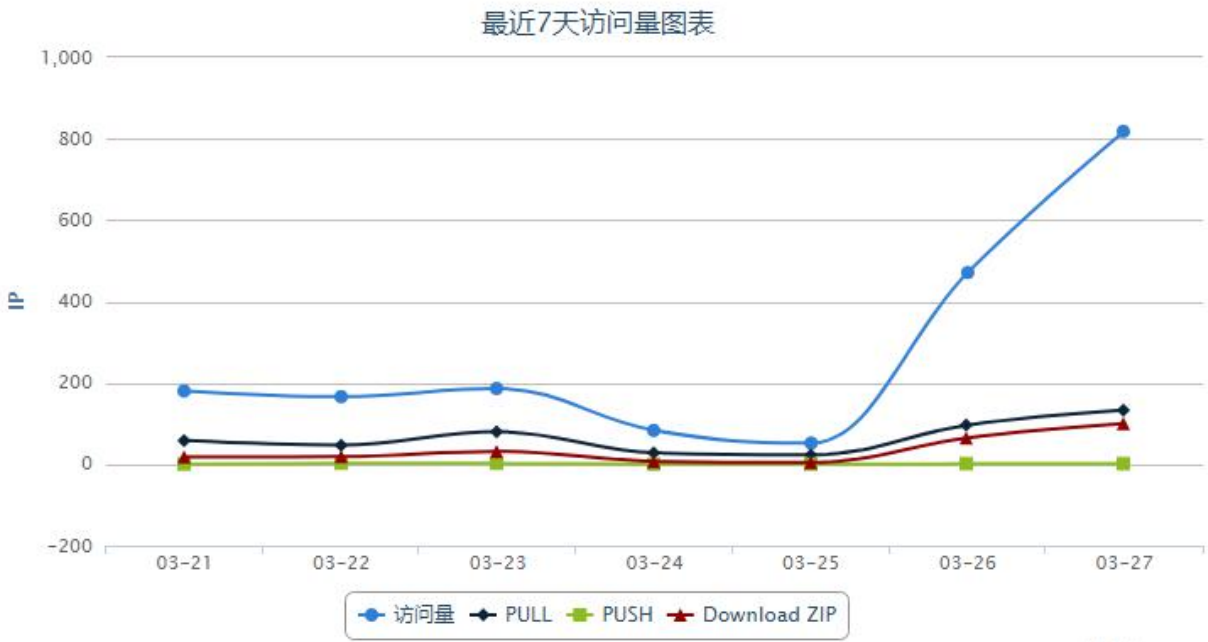


图 8 tio 在码云上的后台统计截图



图 9 t.io 受捐次数

1.10. t.io 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！

6. 第一批码云最有价值开源项目

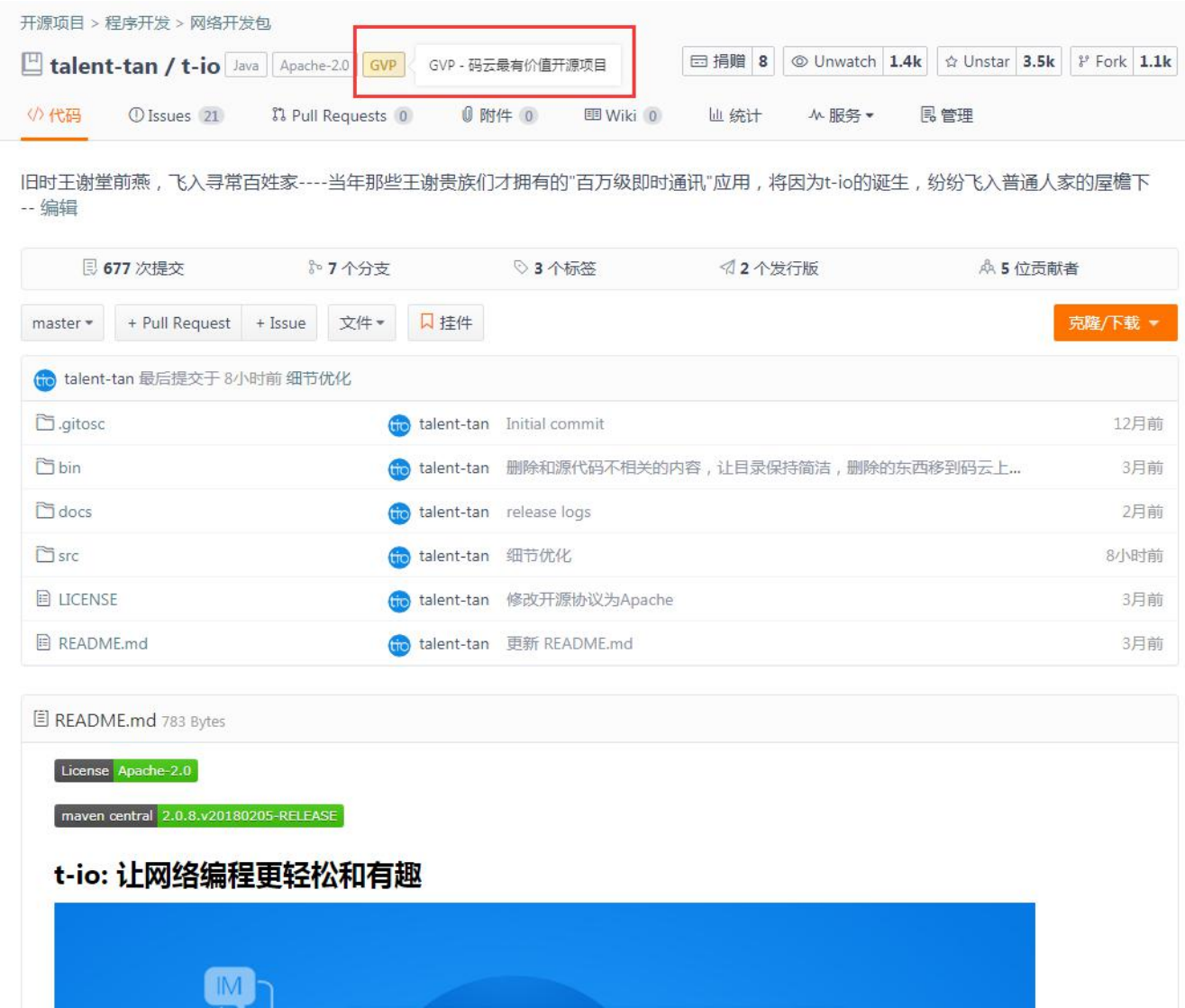


图 10 首批码云最有价值开源项目

7. 2017 年最受欢迎开源软件上榜

考虑到 t-io 的用户群体特点，能进入 TOP20，已属不易



图 11 2017 年最受欢迎开源软件上榜

8. 2017 年热门开源项目 Star 数第 3，Fork 数第 5



图 12 2017 年热门开源项目 Star 数第 3，Fork 数第 5

1.11. tio 分支

tio 分为自用版和开源版 (<https://gitee.com/tywo45/t-io>)，两者差别很小，自用版多了**集群**和**拉黑**功能，其它几乎是一样的。

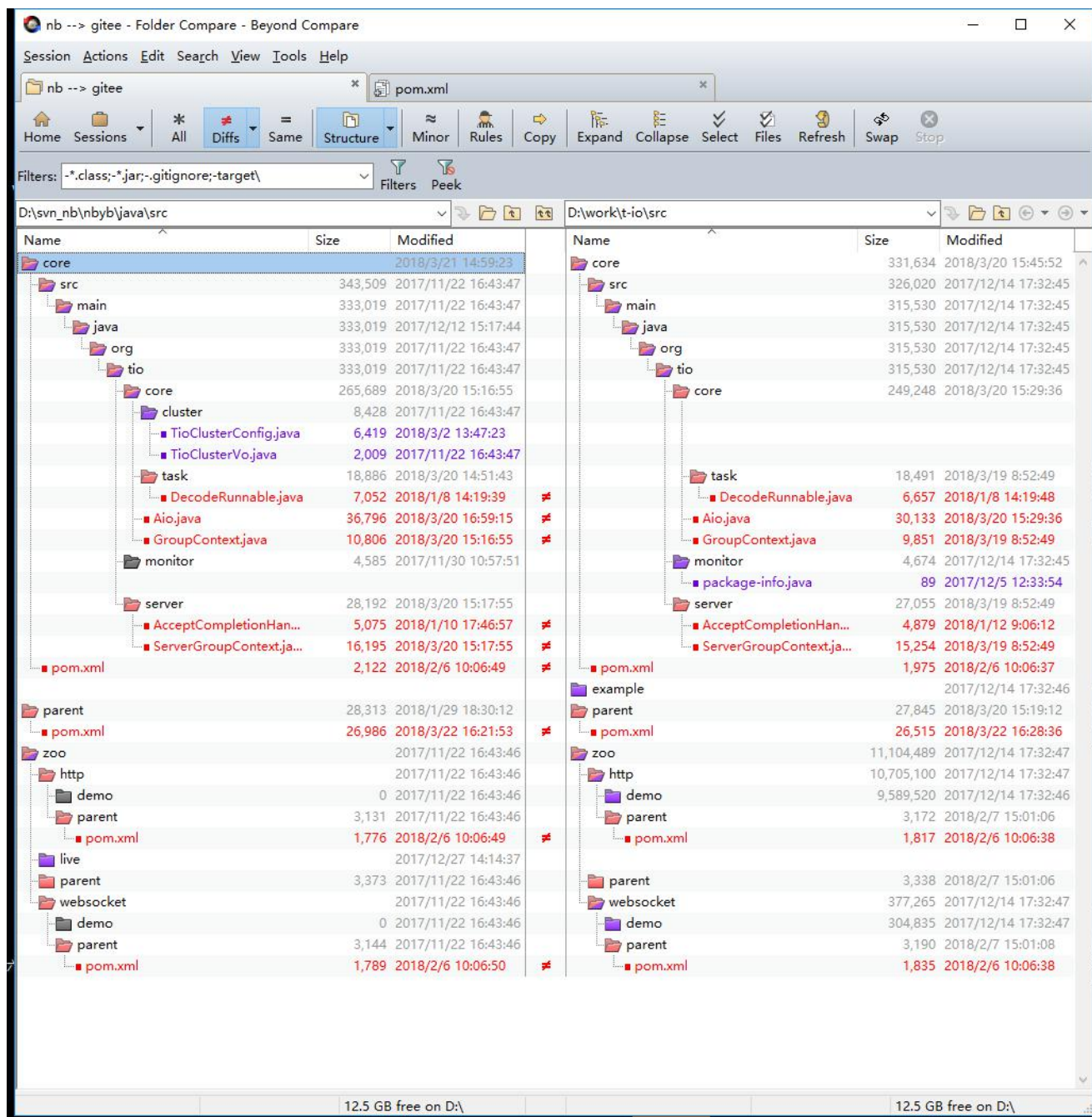


图 13 tio 自用版和开源版的差异

第二章 预备知识

1.1. 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 14 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。

1.2. 应用层和传输层的数据传递

1. 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()`):

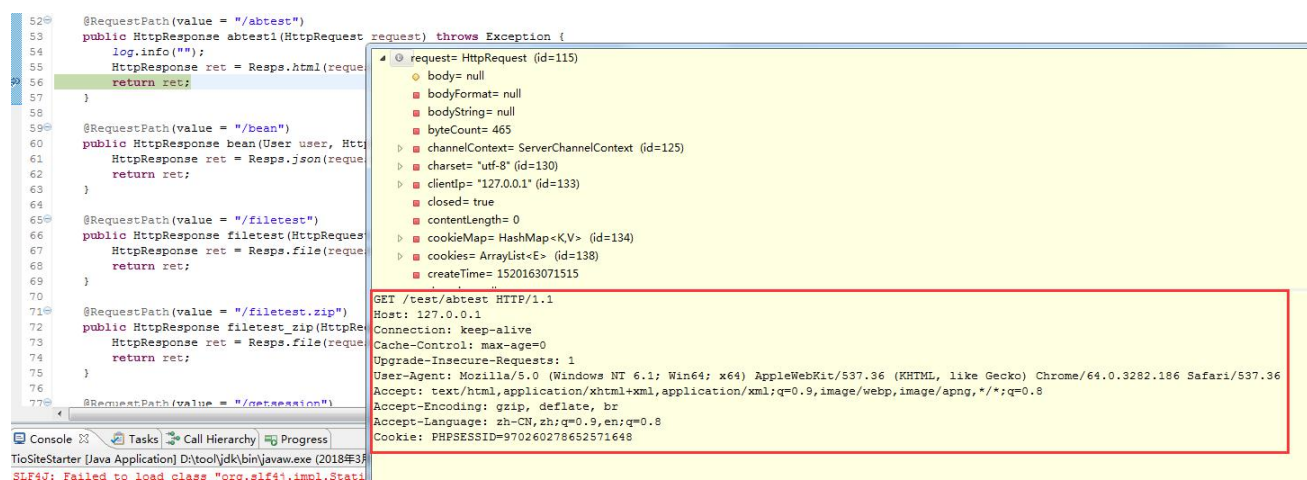


图 15 http 请求数据演示

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 16 半包演示



图 17 粘包演示

2. 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

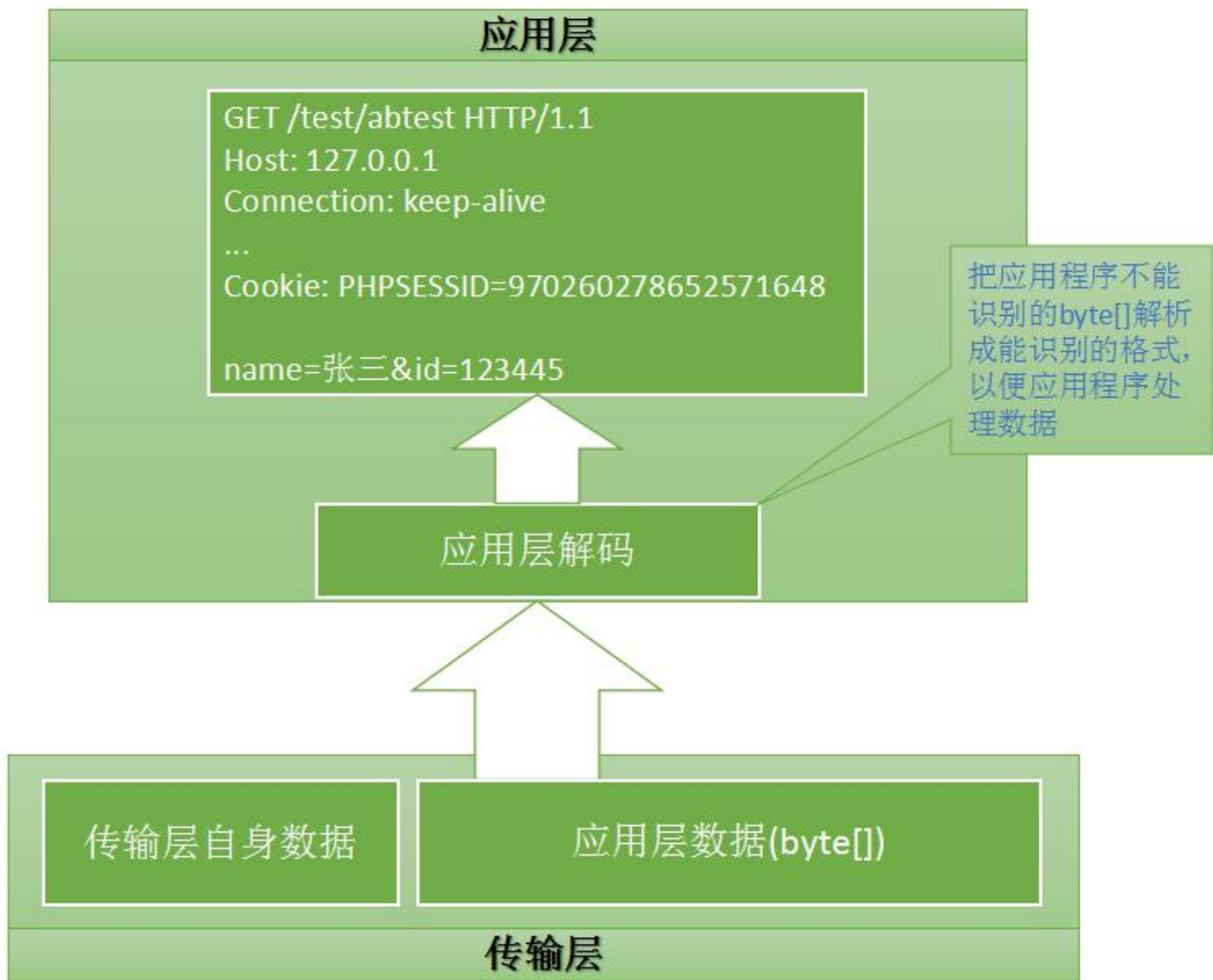


图 18 应用层解码

3. 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

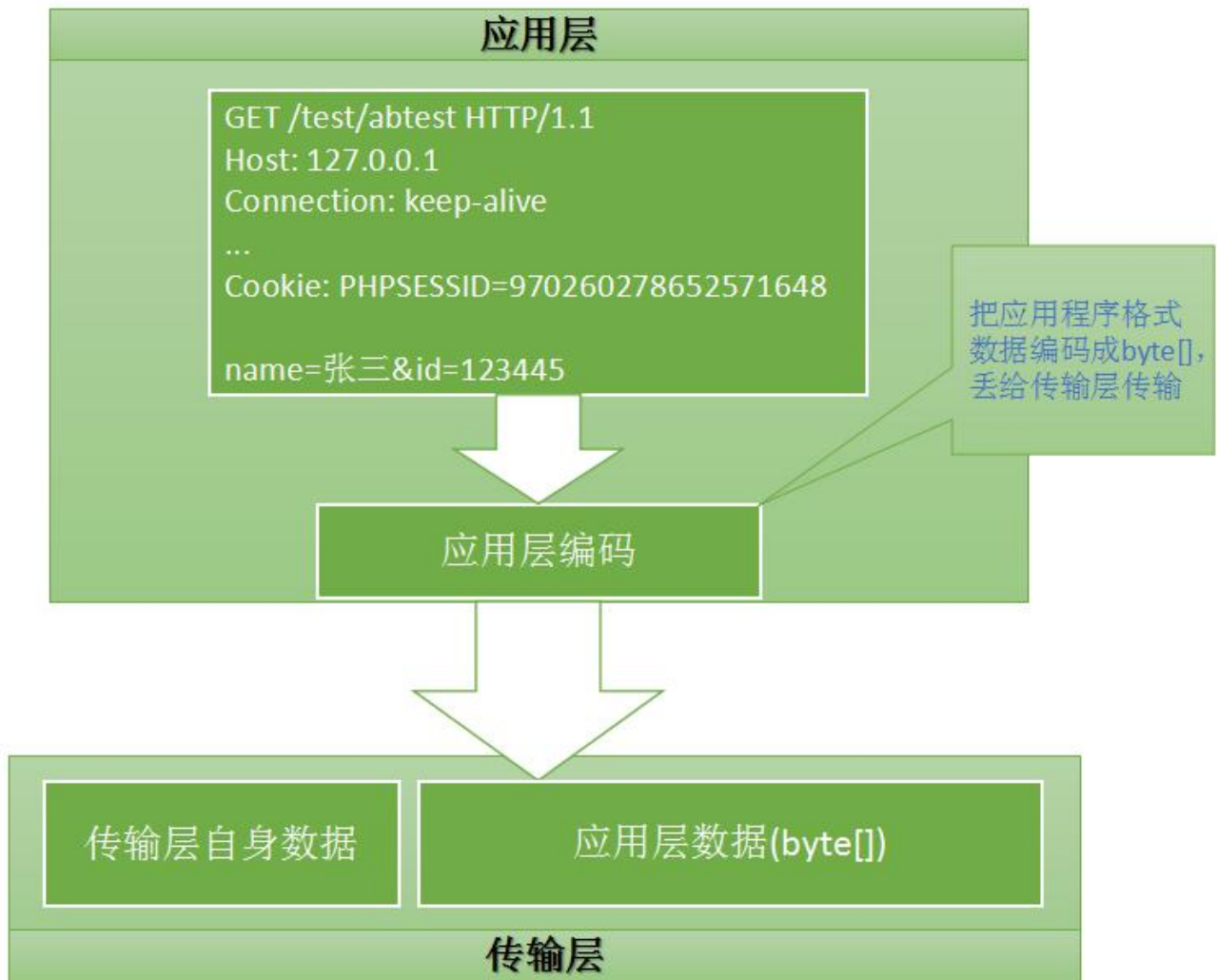


图 19 应用层编码

1.3. 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。

1. 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 bytearray 形成一个感观认识吧

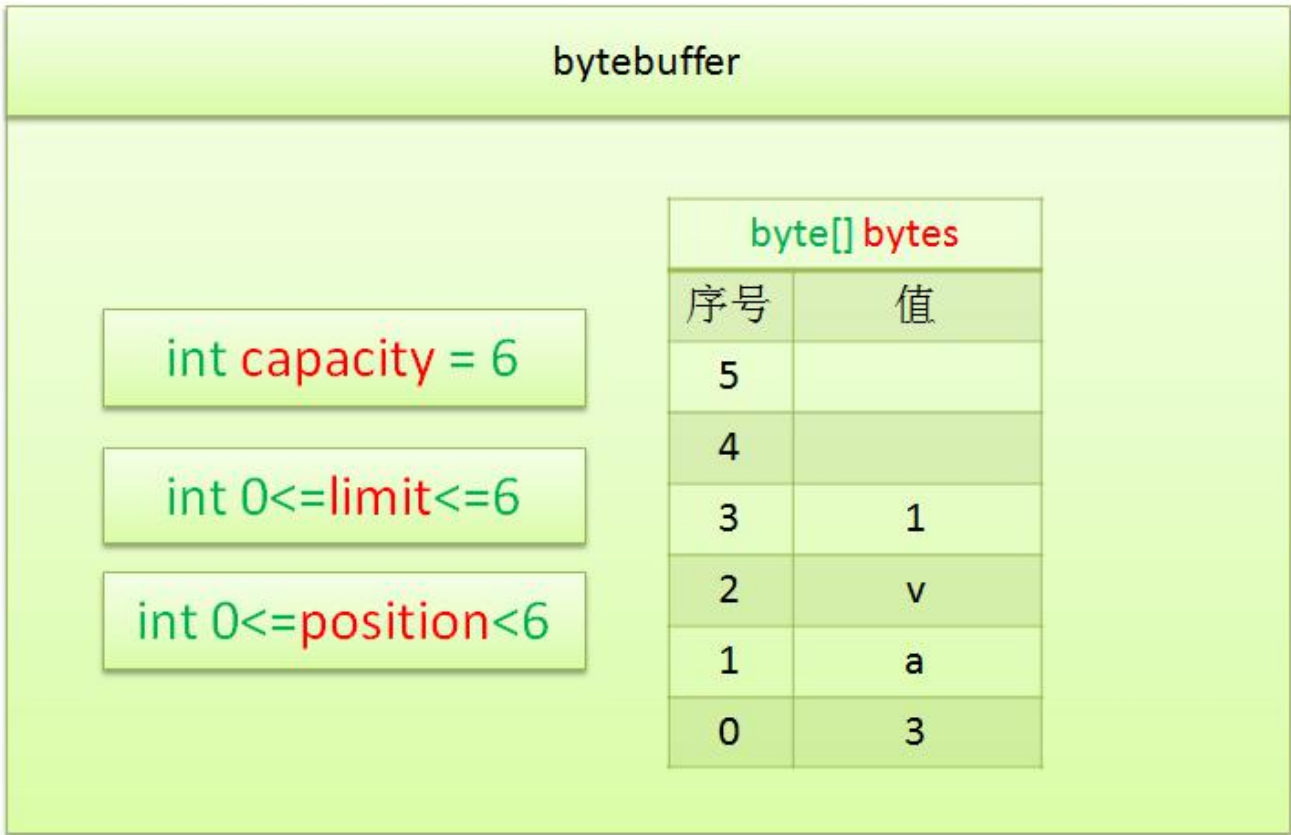


图 20 初识 ByteBuffer

2. 创建 ByteBuffer

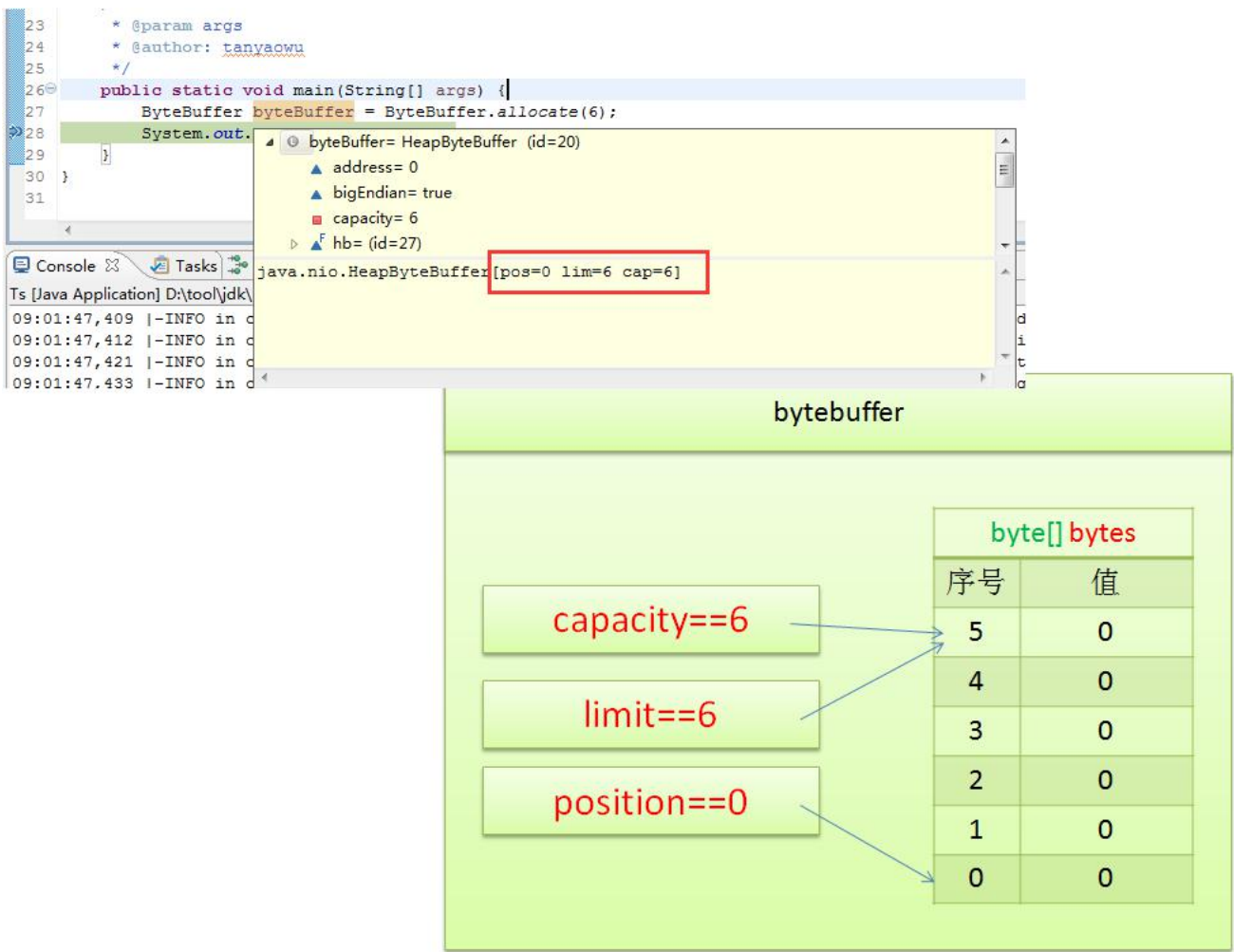


图 21 创建 ByteBuffer

3. 往 ByteBuffer 中写入数据

```
public static void main(String[] args) {
    ByteBuffer byteBuffer = ByteBuffer.allocate(6);
    byteBuffer.put((byte)3);
    System.out.println(byteBuffer);
}
```

byteBuffer= HeapByteBuffer (id=20)
 address= 0
 bigEndian= true
 capacity= 6
 hb= (id=27)
 java.nio.HeapByteBuffer[pos=1 lim=6 cap=6]

注意pos已经变为1了

bytebuffer

byte[] bytes	
序号	值
5	0
4	0
3	0
2	0
1	0
0	3

capacity==6
 limit==6
 position==1

图 22 往 ByteBuffer 中写入数据

4. 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

```
public static void main(String[] args) {  
    ByteBuffer byteBuffer = ByteBuffer.allocate(6);  
    byteBuffer.put((byte) 3);  
  
    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  
}
```

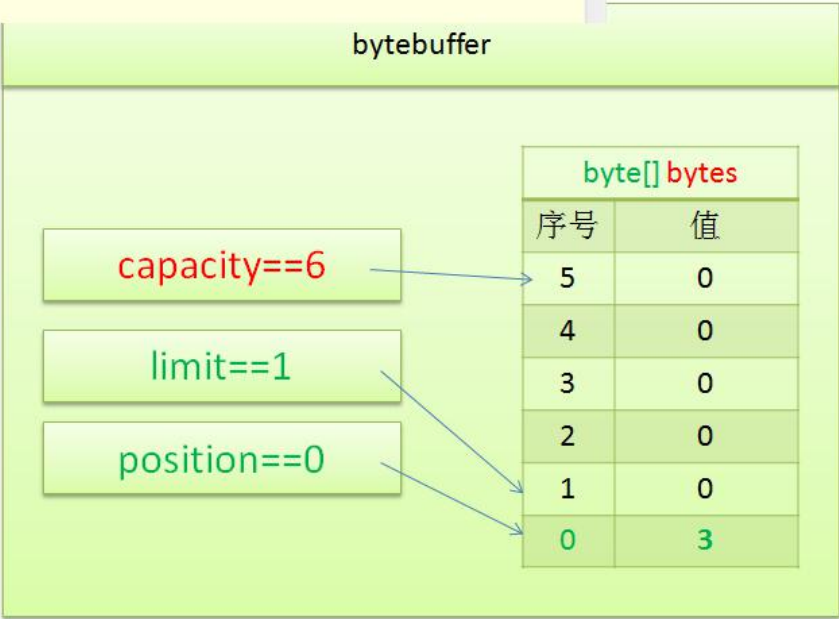
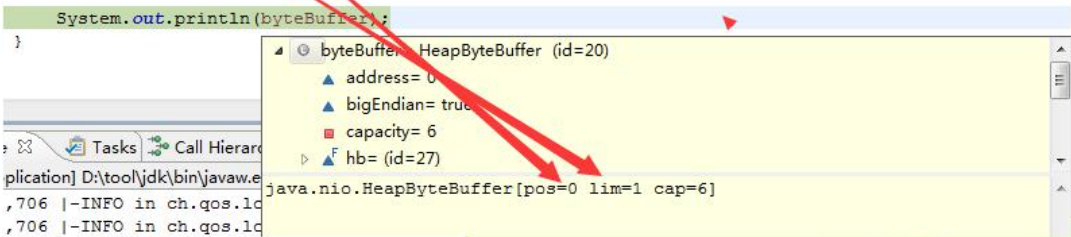


图 23 设置 position 和 limit 后，bytebuffer 的内部变化

接下来，我们就可以读取刚才写入的数据了

```
public static void main(String[] args) {
    ByteBuffer byteBuffer = ByteBuffer.allocate(6);
    byteBuffer.put((byte)3);

    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

    byte bs = byteBuffer.get();
    System.out.println(byteBuffer);
}
```

get()操作也会改变position

byteBuffer= HeapByteBuffer (id=20)
address= 0
bigEndian= true
capacity= 6
hb= (id=27)
java.nio.HeapByteBuffer[pos=1 lim=1 cap=6]

bytebuffer

byte[] bytes	
序号	值
5	0
4	0
3	0
2	0
1	0
0	3

capacity==6

limit==1

position==1

图 24 从 ByteBuffer 中读取数据

第三章 开启 tio 之旅

1.1. 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 25 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，套用于自己项目。

1. 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)

2. 公共模块

(1) 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>
```

(2) 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_LENGTH = 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;
    }
}
```

(3) 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;

}

```

3. 服务端代码

(1) 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_LENGTH) {

        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_LENGTH + 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 buffer 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_LENGTH + 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;

    }

}

```

(2) 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);  
}  
}
```

4. 客户端代码

(1) 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_LENGTH) {

            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_LENGTH + 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 buffer 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_LENGTH + 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;
    }
}

```

(2) 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);
    }
}

```

5. 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
```

1.2. Tio 常见类介绍

1. 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。

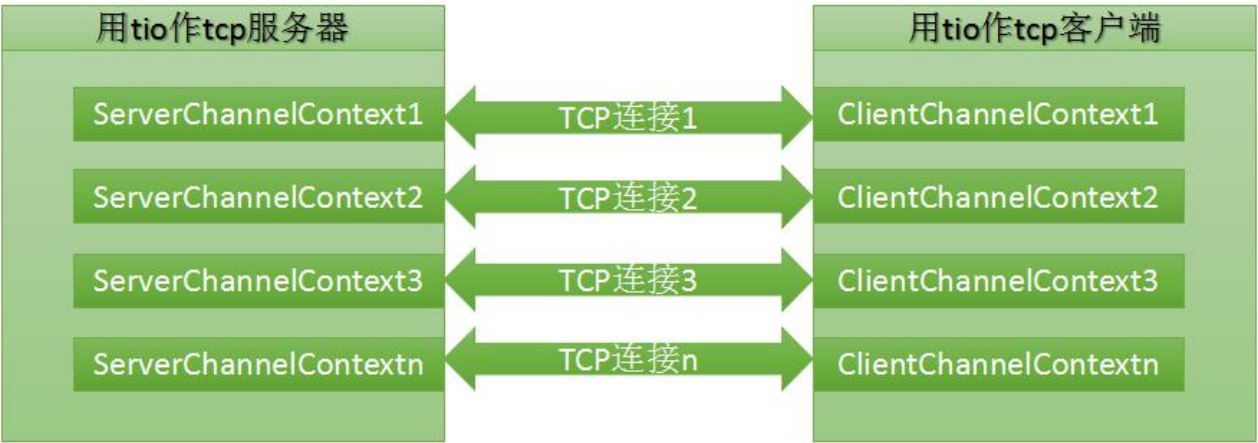


图 26 ChannelContext 概念

ChannelContext 对象包含的信息非常多，主要对象见下图



图 27 ChannelContext 主要对象

(1) 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。

(2) 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。

2. 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

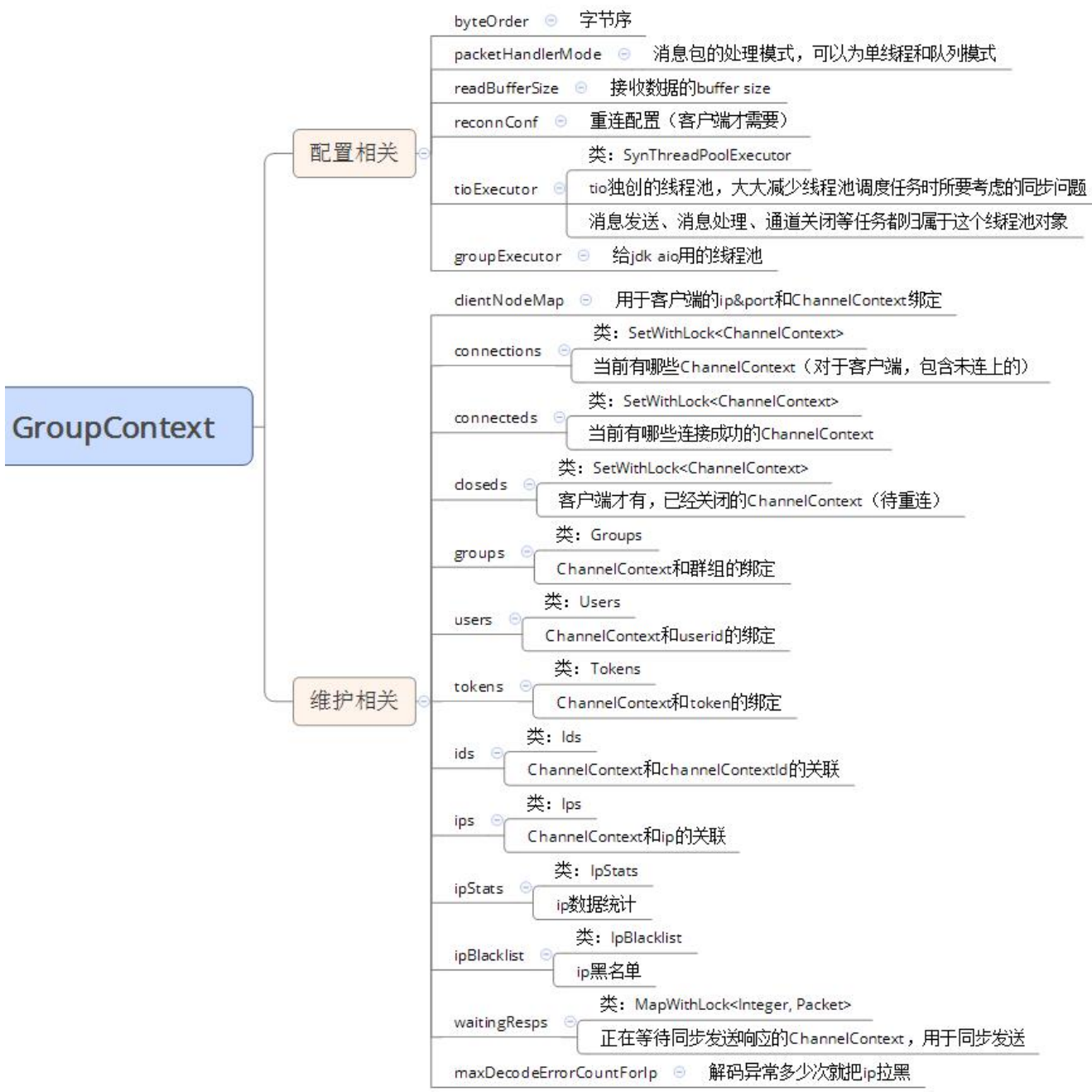


图 28 GroupContext 主要对象

(1) 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。

(2) 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。

3. 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

```

14 public interface AioHandler {
15
16     /**
17      * 根据ByteBuffer解码成业务需要的Packet对象.
18      * 如果收到的数据不全, 导致解码失败, 请返回null, 在下次消息来时框架层会自动续上前面的收到的数据
19      * @param buffer
20      * @param channelContext
21      * @return
22      * @throws AioDecodeException
23      * @author: tanyaowu
24      */
25     Packet decode(ByteBuffer buffer, ChannelContext channelContext) throws AioDecodeException;
26
27     /**
28      * 编码
29      * @param packet
30      * @param groupContext
31      * @param channelContext
32      * @return
33      * @author: tanyaowu
34      */
35     ByteBuffer encode(Packet packet, GroupContext groupContext, ChannelContext channelContext);
36
37     /**
38      * 处理消息包
39      * @param packet
40      * @param channelContext
41      * @throws Exception
42      * @author: tanyaowu
43      */
44     void handler(Packet packet, ChannelContext channelContext) throws Exception;
45
46 }

```

图 29 AioHandler 接口

(1) ServerAioHandler

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

(2) ClientAioHandler

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

4. 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

```

10 public interface AioListener {
11     /**
12      * 连接关闭后触发本方法
13      * @param channelContext the channelContext
14      * @param throwable the throwable 有可能为空
15      * @param remark the remark 有可能为空
16      * @param isRemove 是否是删除
17      * @throws Exception
18      * @author: tanyaowu
19      */
20     void onAfterClose(ChannelContext channelContext, Throwable throwable, String remark, boolean isRemove) throws Exception;
21
22     /**
23      * 建链后触发本方法, 注: 建链不一定成功, 需要关注参数isConnected
24      * @param channelContext
25      * @param isConnected 是否连接成功, true:表示连接成功, false:表示连接失败
26      * @param isReconnect 是否是重连, true: 表示这是重新连接, false: 表示这是第一次连接
27      * @throws Exception
28      * @author: tanyaowu
29      */
30     void onAfterConnected(ChannelContext channelContext, boolean isConnected, boolean isReconnect) throws Exception;
31
32     /**
33      * 解码成功后触发本方法
34      * @param channelContext
35      * @param packet
36      * @param packetSize
37      * @throws Exception
38      * @author: tanyaowu
39      */
40     void onAfterReceived(ChannelContext channelContext, Packet packet, int packetSize) throws Exception;
41
42     /**
43      * 消息包发送之后触发本方法
44      * @param channelContext
45      * @param packet
46      * @param isSentSuccess true:发送成功, false:发送失败
47      * @throws Exception
48      * @author tanyaowu
49      */
50     void onAfterSent(ChannelContext channelContext, Packet packet, boolean isSentSuccess) throws Exception;
51
52     /**
53      * 连接关闭前触发本方法
54      * @param channelContext the channelContext
55      * @param throwable the throwable 有可能为空
56      * @param remark the remark 有可能为空
57      * @param isRemove
58      * @author tanyaowu
59      */
60     void onBeforeClose(ChannelContext channelContext, Throwable throwable, String remark, boolean isRemove);
61 }
62

```

图 30 AioListener 接口

(1) ServerAioListener

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

(2) ClientAioListener

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

5. 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

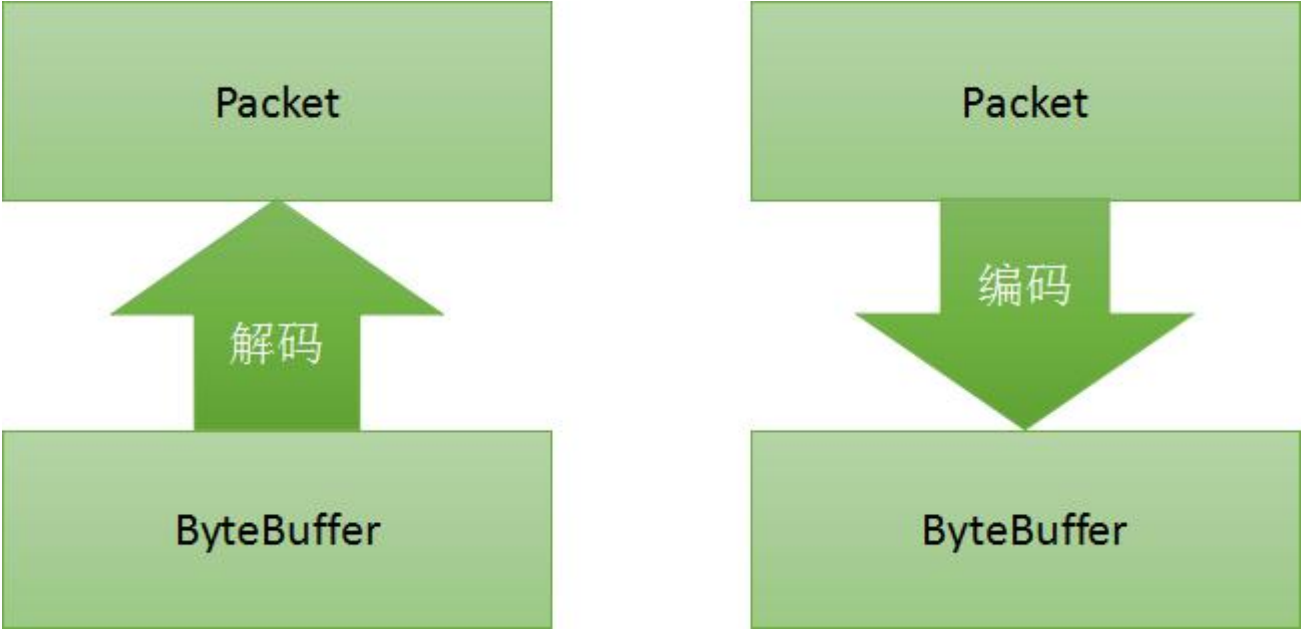


图 31 Packet 和 ByteBuffer 转换

6. AioServer (tio 服务端入口类)

tio 服务端入口类，主要对象见下图

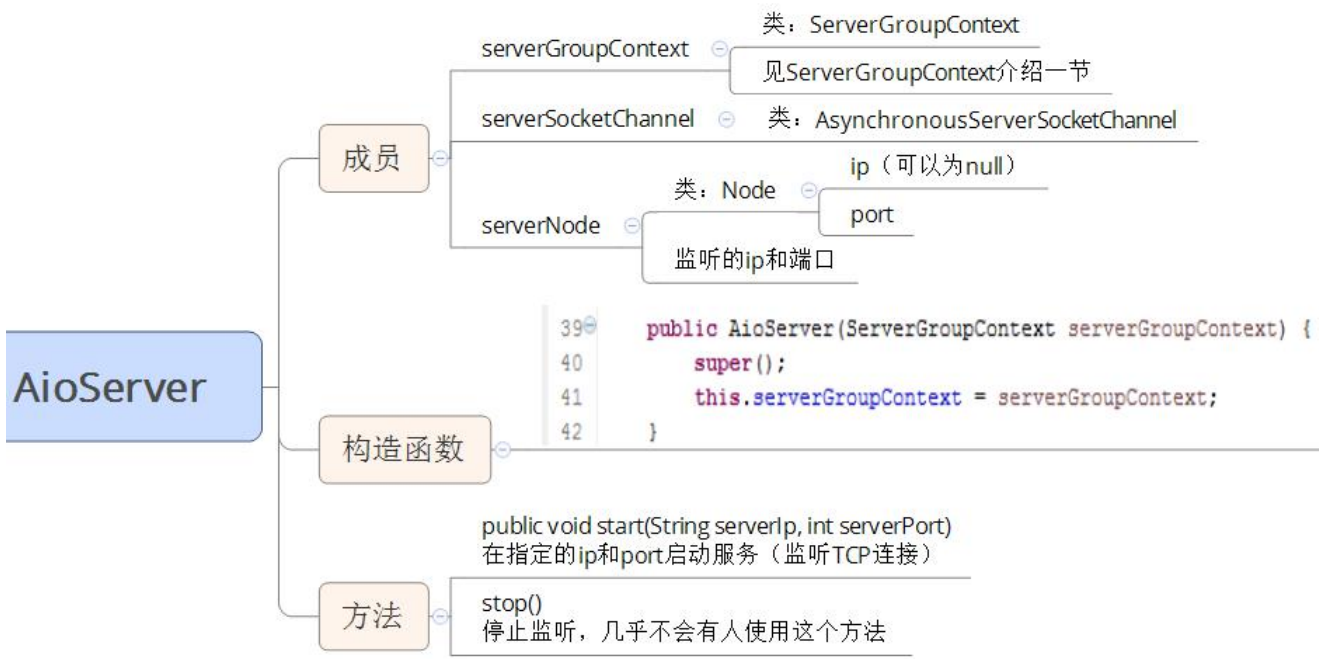


图 32 AioServer 对象

7. AioClient (tio 客户端入口类)

tio 客户端入口类，主要对象见下图

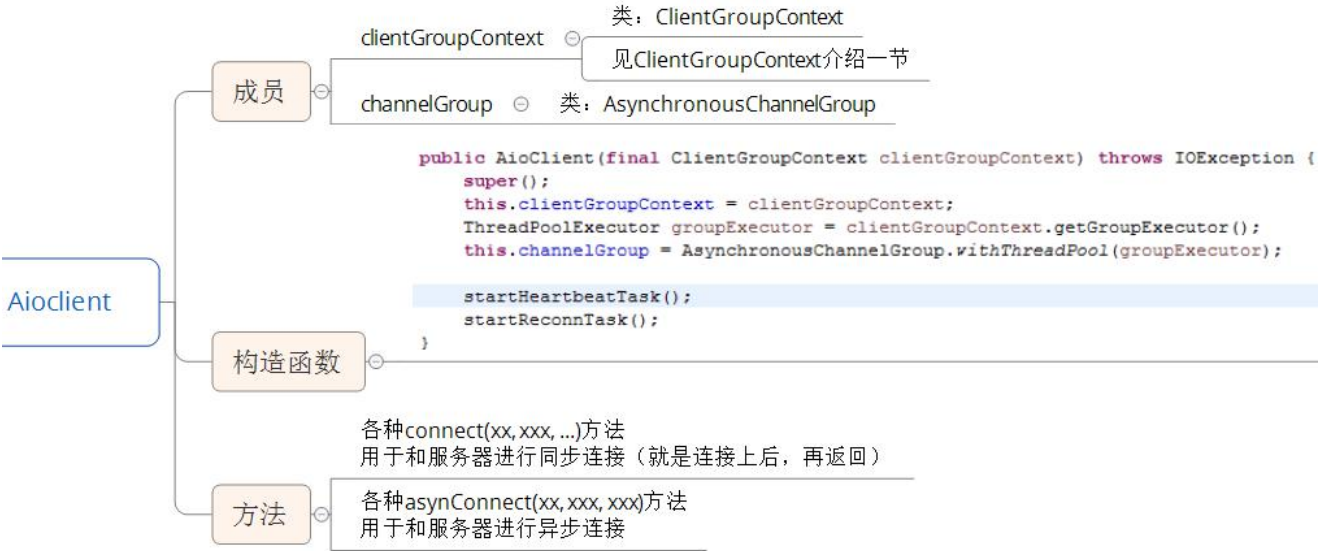


图 33 AioClient 对象

8. 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：

```

6  /**
7   * 自带读写锁的对象.
8   *
9   * @author tanyaowu
10  */
11 public class ObjWithLock<T> implements Serializable {
12     /**
13      *
14      */
15     private T obj = null;
16
17     /**
18      *
19      */
20     private ReentrantReadWriteLock lock = null;
21
22     /**
23      *
24      * @param obj
25      * @author tanyaowu
26      */
27     public ObjWithLock(T obj) {
28         this(obj, new ReentrantReadWriteLock());
29     }
30
31     /**
32      *
33      * @param obj
34      * @param lock
35      * @author tanyaowu
36      */
37     public ObjWithLock(T obj, ReentrantReadWriteLock lock) {
38         super();
39         this.obj = obj;
40         this.lock = lock;
41     }
42
43     /**
44      *
45      * @return
46      * @author tanyaowu
47      */
48     public ReentrantReadWriteLock getLock() {
49         return lock;
50     }
51

```

图 34 ObjWithLock

为了更便捷地操作，tio 提供了三个 ObjWithLock 子类，见下图



图 35 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：

```

457- /**
458-  * 删除client ip为指定值的所有连接
459-  * @param groupContext
460-  * @param ip
461-  * @param remark
462-  * @author: tanyawu
463-  */
464- public static void remove(GroupContext groupContext, String ip, String remark) {
465-     SetWithLock<ChannelContext> setWithLock = Aio.getAllChannelContexts(groupContext);
466-     Lock lock2 = setWithLock.getLock().readLock();
467-     lock2.lock();
468-     try {
469-         Set<ChannelContext> set = setWithLock.getObj();
470-         for (ChannelContext channelContext : set) {
471-             String clientIp = channelContext.getClientNode().getIp();
472-             if (StringUtils.equals(clientIp, ip)) {
473-                 Aio.remove(channelContext, remark);
474-             }
475-         }
476-     } finally {
477-         lock2.unlock();
478-     }
479- }
  
```

图 36 SetWithLock 例子

```

99  /**
100  * 某通道是否在某群组中
101  * @param group
102  * @param channelContext
103  * @return true: 在该群组
104  * @author: tanyaowu
105  */
106  public static boolean isInGroup(String group, ChannelContext channelContext) {
107      MapWithLock<ChannelContext, SetWithLock<String>> mapWithLock =
108          channelContext.getGroupContext().groups.getChannelmap();
109      ReadLock lock = mapWithLock.getLock().readLock();
110      lock.lock();
111      try {
112          Map<ChannelContext, SetWithLock<String>> m = mapWithLock.getObj();
113          if (m == null || m.size() == 0) {
114              return false;
115          }
116          SetWithLock<String> set = m.get(channelContext);
117          if (set == null) {
118              return false;
119          }
120          return set.getObj().contains(group);
121      } catch (Throwable e) {
122          log.error(e.toString(), e);
123          return false;
124      } finally {
125          lock.unlock();
126      }
127  }

```

图 37 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 前面进行 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。

```
99  /**
100  * 某通道是否在某群组中
101  * @param group
102  * @param channelContext
103  * @return true: 在该群组
104  * @author: tanyaowu
105  */
106  public static boolean isInGroup(String group, ChannelContext channelContext)
107      MapWithLock<ChannelContext, SetWithLock<String>> mapWithLock =
108          channelContext.getGroupContext().groups.getChannelmap();
109      1 ReadLock lock = mapWithLock.getLock().readLock();
110      2 lock.lock();
111      try {
112          Map<ChannelContext, SetWithLock<String>> m = mapWithLock.getObj();
113          if (m == null || m.size() == 0) {
114              3 return false;
115          }
116          SetWithLock<String> set = m.get(channelContext);
117          if (set == null) {
118              return false;
119          }
120          return set.getObj().contains(group);
121      } catch (Throwable e) {
122          log.error(e.toString(), e);
123          return false;
124      } finally {
125          4 lock.unlock();
126      }
127  }
128  }
```

图 38 锁操作 4 步曲