CANNON Manual

目录

CANNON

通用服务端引擎 CANNON 由几个部分组成：

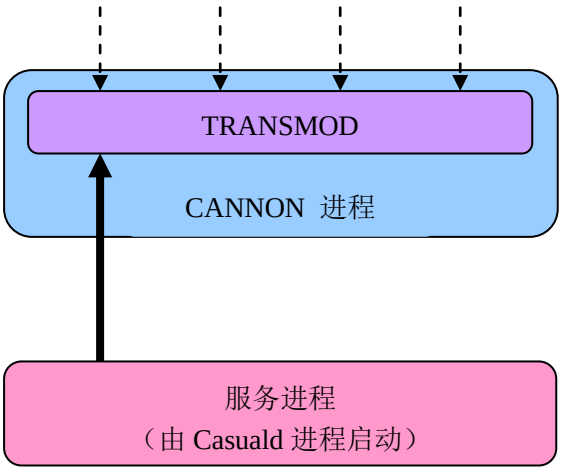
1. cannon.py – 独立进程，加载 transmod.so 启动服务，并运行 channel 下指定的服务进程。
2. transmod.so – 由 cannon.py 启动
3. cclib.so – 服务进程用来连接 transmod.so 用
4. 服务进程 – 不同于 Cannon 进程，是实现具体服务的主进程，由 Cannon 启动。

启动服务

自 SVN 上取得 cannon 目录，进入 bin，执行 cannon.py 或者 cannon 来启动不同的服务。

```
# cd cannon/bin
# ./cannon start echoserver
```

这里以自带的 echoserver 为例，echoserver 的配置信息放在 conf/echoserver.ini 有了它，就可以使用 cannon 来启动了。里面描述了 echoserver 程序执行的路径，在 channel/echoserver 下面的 echosvr 文件，修改其中的代码，就可以改服务逻辑了。



由命令：
cannon.py start xxx
启动该进程

cannon 进程启动后，自动启动服务进程，cannon 停止后，自动停止服务进程，该进程的启动方式及监听端口等由 conf/xxx.ini 描述。

如上图所示其中 echoserver 就是那个服务进程。用 cannon stop echoserver 就可以同时停止两个进程。

查看日志

到 logs/echoserver 下面有几个文件，分别是（主要看 m20110318.log）：

* 服务日志：m20110318.log 由 echoserver 产生，服务自己具体的日志

* 网络日志：d20110318.log 由 transmod.so 产生，描述网络连接情况，断开原因等。
* 额外日志：cannon.log 由 bin/cannon.py 产生，描述服务启停情况。
* 进程编号：cannon.pid 由 bin/cannon.py 产生，记录服务的进程编号，用来停止服务

配置服务

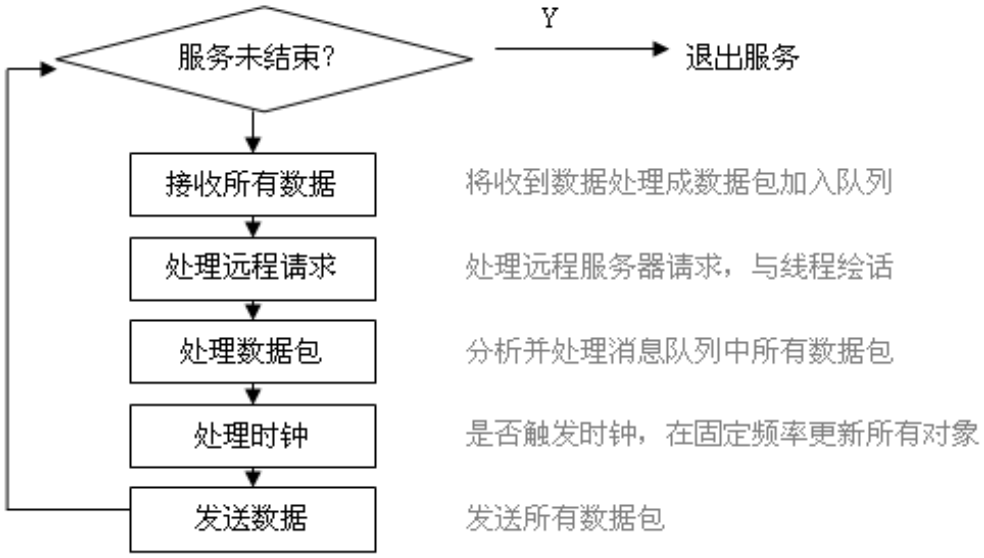
到 conf 目录下有 echoserver.ini，照葫芦画瓢就可以写一个 testsvr.ini，然后就可以用

```
# ./cannon start testsvr
```

启动了，注意配置成不同的端口，PORTU，PORTC。

编写代码

服务端通用的程序主循环如下：



可以用下面的伪代码来简单表示：

```
while not gameover:
        event, wparam, lparam, data = read_event()
        if event == ITMC_NEW:
```

```
        处理玩家进入
    elif event == ITMC_LEAVE:
        处理玩家离开
    elif event == ITMC_DATA:
        处理玩家数据
    elif event == ITMC_TIMER:
        处理时钟
```

参考 echoserver 的代码：channel/echoserver/echosvr.py
在 conf/echoserver.ini 里面指定了 transmod 的端口以及 echoserver 的启动程序后，即可使用 cannon.py 来启动 echoserver。

Echosvr 是一个简单的服务，它将用户发送过来的数据一模一样的返回回去。

```
import sys, time
import os
import cclib
import ccinit
from instruction import *

ccinit.attach()       # attach to transmod
cclib.settimer(1000)   # set timer event (1000ms interval)

while 1:
    # receive event from transmod
    event, wparam, lparam, data = cclib.read()
    if event < 0: break  # transmod disconnected

    if event == ITMT_DATA: # receive data sent by remote user
        cclib.send(wparam, data)
    elif event == ITMT_NEW: # remote user connected to transmod
        remote = ccinit.parseip(data)
        ccinit.plog('new user hid=%d from %s'%(wparam, remote))
    elif event == ITMT_LEAVE: # remote user disconnect from transmod
        ccinit.plog('user disconnect hid=%d'%wparam)
    elif event == ITMT_TIMER: # timer event
        print 'TIMER'
```

基本消息：

服务器逻辑通过 cclib 连接到 transmod 以后，可以收到 transmod 发送过来的以下消息

消 息	说 明	WPARAM	LPARAM	DATA
ITMT_NEW	新用户连接	用户标识 HID	TAG	IP 地址
ITMT_DATA	收到用户数据	用户标识 HID	TAG	数据内容
ITMT_LEAVE	用户断开连接	用户标识 HID	TAG	错误代码
ITMT_TIME	时钟消息	-	-	-
ITMT_CHANNEL	频道通信	频道编号		

基本操作：

服务器逻辑通过调用 cclib 的接口，可以给 transmod 发送消息，完成特定操作

消 息	说 明	参 数
cclib.read	读取消息	cclib.read(noblock = 0)
cclib.send	发送数据	cclib.send(hid, data)
cclib.close	断开连接	cclib.close(hid, code)
cclib.settag	设置 TAG	cclib.settag(hid, tag)
cclib.movec	移动用户	cclib.movec(channel_id, hid, extern_data)
cclib.channel	频道通信	cclib.channel(channel_id, data)
cclib.settimer	设置时钟	cclib.settimer(millisec)
cclib.attach	连接 transmod	cclib.attach(transmod_ip, transmod_port, channel_id)

注意：
用户断开，不管是用户自己断开，还是调用了 cclib.close 由 transmod 来断开该用户，在用户最终断开的时候，都会收到 ITMT_LEAVE 消息。

通常编写逻辑时，收到 ITMT_NEW 消息时肯定会在内存中建立一个对象来保存该用户的数据。应该只有再收到 ITMT_LEAVE 消息的时候才能去清空该用户的数据（比如从字典里面删除），而不应该调用完 cclib.close 以后马上就清空该用户的数据（否则后面 ITMT_LEAVE 消息来的时候如果也做了删除操作，则有可能导致混乱）。

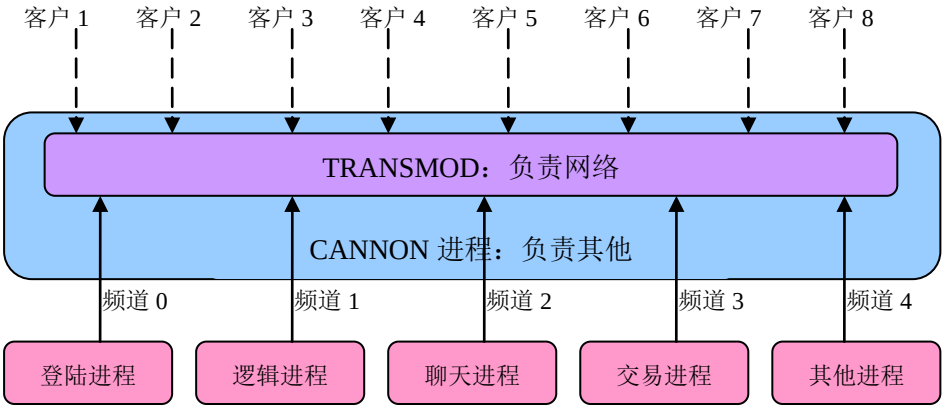
启动 echoserver 以后，可以通过 bin/terminal.py 来调试，右下方填写地址“127.0.0.1:6000”。然后点“GO”，连接上以后，在左下方输入命令：“1,2,3,4,5”，然后点发送，就可以看到发送到服务端的情况，并且可以接收到服务端的返回，详细见后面 TERMINAL 使用说明。

TRANSMOD

基础概念

传输模块（TRANSMOD）监听两个端口：对外端口供众多用户来连接；对内端口供服务逻

辑来连接。对内端口一般监听的是内网 IP 或者 127.0.0.1，只有内网连接才能被允许。



所谓频道就是服务端一个个提供服务的逻辑进程，他们都连接着 TRANSMOD，通过 TCP 连接与 TRANSMOD 交互为用户提供不同的服务，他们可以在一台机器上或者分散到不同的机器上运行。（注意：多频道模式并非强制，初期可以只用频道 0 提供所有服务）

频道控制权：

用户登录后，默认对其控制的频道（频道）是 0 号频道。用户产生的所有消息（发送数据，断开）都会发送到对其拥有控制权的频道。如果 0 号频道不存在（没有逻辑进程 attach 到 transmod 的 0 号频道），transmod 将认为不能为用户提供服务。那么所有连接到 transmod 的用户都会被自动断开。

用户控制权可以在不同频道之间转移，比如在从频道 0 转移到频道 1 以后，频道 0 就不能再接收到该用户的任何消息，这些消息将被发送给频道 1。而频道 1 也有权发送数据给该用户，或者断开该用户。

比如让频道 0 来负责用户登录，而其他频道负责不同的游戏场景，完成登录的用户会被转移到不同的游戏场景，而另外一个频道负责和数据库打交道。但是这并不是必须的。

而频道 0 可以将该用户的控制权转移给频道 1，此后频道 0 不再接收来自该用户的任何消息。而频道 1 才可以接收到该用户的消息，并且可以给该用户发送数据，或者断开该用户。

频道间通信：

频道之间是可以通信的，把消息发送给 transmod，并由 transmod 中转。

数据格式：

所有对外的数据包为两字节（可以配置）的包长和数据组成，Little Endian 格式为标志，及低位在前（可以配置）。

SIZE	DATA(长度为 size-2)

传输模块客户数据包格式

比如客户端给 transmod 发送 100 个字节的数据时，需要先发送两个字节的包头，使用 Little Endian 格式，值为 102（100 字节的数据+2 字节包头），再发送后面的内容。

而包头的格式是可以在 conf/*.ini 里面配置的，一共支持下面几种：

```
0     (WORDLSB)   用两字节表示数据包长度，低位在前 （默认格式）
1     (WORDMSB)   用两字节表示数据包长度，高位在前
2     (DWORDLSB)  用四字节表示数据包长度，低位在前
3     (DWORDMSB)  用四字节表示数据包长度，高位在前
4     (BYTELSB)   用单字节表示数据包长度，低位在前
5     (BYTEMSB)   用单字节表示数据包长度，高位在前
6     (EWORDLSB)  用两字节表示数据包长度，低位在前，不包括自己
7     (EWORDMSB)  用两字节表示数据包长度，高位在前，不包括自己
8     (EDWORDLSB) 用四字节表示数据包长度，低位在前，不包括自己
9     (EDWORDMSB) 用四字节表示数据包长度，高位在前，不包括自己
10    (EBYTELSB)  用单字节表示数据包长度，低位在前，不包括自己
11    (EBYTEMSB)  用单字节表示数据包长度，高位在前，不包括自己
12    (DWORDMASK) 用四字节表示包长度，低位在前，高 8 位表示消息分类（category）
13    (RAWDATA)   原始数据模式，对外无包头，对内 4 字节 LSB（含自己），低位在前
```

具体客户端发送数据给 transmod 的例子可以查看 lib/netstream.py 里面的 netstream 类的实现。或者参考 tools/csimpletcp.cpp（不过里面只提供了 WORDLSB 模式的实现）。如果其他语言做客户端，需要仿照上面所述的模式进行开发。

拥塞控制

重点内容，不管是外部用户链接，还是内部频道连接，他们在 TRANSMOD 中都有一个发送和接收缓存：

- 发送缓存：要发送给该连接的消息先放到缓存里，能发出去的时候再从里面发。
- 接收缓存：接收到的东西先放在接收缓存里，要处理时再从接收缓存里拿。

而同时 TRANSMOD 拥塞控制策略对于外部连接和内部连接的两个缓存都有大小限制，外部连接限制叫做 MEMLIMITU，内部叫做 MEMLIMITC，在下面几种情况中发生作用：

1. 任何频道向 TRANSMOD 发送的数据再 TRANSMOD 处理前都会放到频道缓存，然后再从频道缓存得到处理，也就是说，如果频道一次性向 TRANSMOD 发送的数据量超过了 MEMLIMITC，则该频道连接会被断开，因为频道产生的数据量远远大于 TRANSMOD 的处理速度，所以必须被断开。一般这个值可以设置大点，比如 64M。

2. 频道通过 TRANSMOD 持续向某用户发送消息，虽然没有超过 MEMLIMITC，但是这些

数据被转移到了用户的发送缓存中，当某个用户接收的较慢，它的发送缓存队列里面的数据量超过了 MEMLIMITU，则该用户需要被断开，因为用户的优先级比较低，某用户其实可能网络阻塞很长时间了，或者没有足够的能力去处理发给他的消息，那么它需要被断开。

3. 用户向 TRANSMOD 发送数据，如果一次性发送超过了 MEMLIMITU，则会被断开，同第一条。

4. 用户通过 TRANSMOD 持续向某频道发送数据，如果导致某频道的发送缓存超过 MEMLIMITC 了（频道处理不过来），那么该连接的读事件将会被静止掉，此后所有试图向该频道发送数据的外部连接都会被禁止读事件，并被放到一个等待链表里，直到该频道的逻辑进程又反应过来了，又多处理了几个包，发送缓存小于 MEMLIMITC 时，才会从等待队列里陆续取出外部连接，允许他们的读事件，允许他们向该频道发送数据。这表现出来的就是如果某频道太忙，则向该频道发送数据的外部连接会被挂起，从外部连接的客户端上看起来就是数据发送不出去了。

消息订阅模式

如果启动 transmod 的时候头部模式被设置为 12（DWORDMASK）的话，将启用消息订阅模式。在这个模式下，客户端发送的消息头为“四字节低位在前的整数”，但是该 32 位消息头的高 8 位代表消息分类：

HEADER = (DATASIZE + 4) | (CATEGORY << 24)

客户端在发送消息的时候已经指明了一个>0 并且<255 的消息分类，同时连接 transmod 的频道可以选择订阅特定分类的消息。比如，如果一个频道已经订阅过 1 号分类的消息的话，所有客户端发送上来的分类为 1 的消息都会抄送一份到该频道。同时客户所属的频道如果没订阅过该分类消息，它将不会接收到，除非消息分类为 0 时则和以前一样转发到所属频道。

分类 0 和 255 是特殊分类，如果客户端发送了 category=0 或者 255 的消息过来，将不会被抄送到任何频道，只会被转发到连接所属的频道去。其中分类 0 代表连接情况消息，就是说新连接进来（ITMT_NEW），或者连接断开（ITMT_LEAVE），这两条消息。如果订阅过分类 0 的频道，那么任何用户连接加入或者断开的消息都会被抄送一份到该频道，而不是把客户端发送的 category=0 的消息抄送给它。

而分类 255 代表时钟消息，因为只有 channel0 可以设置时钟，而其他频道不能设置时钟，但是如果其他频道订阅过分类 255 的话，将可以收到 ITMT_TIMER，它和 channel0 用的是同一个时钟，只是系统出发时钟消息的时候，会同时将 ITMT_TIMER 抄送给订阅过 255 分类的频道。

接口说明：

要订阅一个分类的消息需要调用：
cclib.bookadd(category)

要取消一个分类的消息需要调用：
cclib.bookdel(category)

要清空一个频道的所有分类订阅：
cclib.bookrst()

由于 cclib 的限制，一个频道只有订阅过分类 0 的消息（即连接情况），其他分类的订阅才会生效（历史原因，当时 cclib 被设计成本地有一份客户表，只有在 ITMT_NEW 时会在表内添加客户连接信息，ITMT_LEAVE 时删除，如果客户表中没有该客户，那么针对该客户的所有消息都会被屏蔽）。

应用举例：

比如，用 channel0 实现登录等通用逻辑，而 channel1 实现游戏聊天，channel2 实现交易，后面的 channel 实现游戏地图。各个不同的应用订阅不同的分类，而客户端可以在发送的时候指明分类编号，那就可以让对应的 channel 接收到了。

开始时 cannon.py 会自动启动 channel 0 的逻辑，而 channel 0 的逻辑则根据需要启动其他 channel 的进程。而如果结束 cannon.py 的进程，那么因为 transmod 的停止，其下所有连接到 transmod 的频道进程都会在接受消息时返回-1，这时就可以自行退出了。

用户连接后，channel0 判断如果登录成功，可以用频道间通信告诉其他频道，而当客户端发送的带分类的消息发送给订阅过的频道时，频道可以判断该消息是否非法。

而时钟消息，只有 channel0 才可以设置，但是其他频道可以通过订阅 255 分类获得。

客户端发送缓存：

在消息订阅模式下（DWORDMASK），客户端如果要发送一个 100 字节的消息，分类是 5 的话，可以像下面例子代码那样组织缓存：

```
void send_message(const void *data, int size, int category)
{
	unsigned char buffer[0x100000];
	int total = total + 4; // 长度包含头部本身
	buffer [0] = total & 0xff;
	buffer [1] = (total >> 8) & 0xff;
	buffer [2] = (total >> 16) & 0xff;
	buffer [3] = (unsigned char) (category & 0xff);
```

```
    memcpy (buffer + 4, data, size);
    socket_buffer_and_send_all_data (buffer, total)
}
```

技术特点

多频道通信

才用多频道通信模式，完全以多进程的模式进行协作，将网络以及不同类型的逻辑完全隔离。

订阅模式

参考了 ZeroMQ 模式，并根据本身特点实现了类似的消息订阅，不同类型的频道可以根据自己的需要订阅自己关心的消息。由于使用了高位字节作为消息分类，这样就可以兼容老的客户端了（没有分类，为 0）。

多平台支持

同时支持 FreeBSD、Linux、Win32 等多个平台，支持 64 位系统，能在不同平台下使用不同的异步 I/O 模型。

新缓存机制

传统收发缓冲是用的环缓存实现，从 socket 接收下来的东西先放到环状缓存，凑够一个完整包的时候再一次性取出。向 socket 发送的数据也是先放到环状缓存，等到有发送事件时再发送出去，然后将成功发送出去的数据清除缓存，没发完的下次接着发。这里有个问题就是环状缓存要开辟多大？大了浪费内存，小了又会出问题。我见过有的项目是用了一个可变长度的环状缓存，但是每次改变长度的开销也是非常大的，而且也会有频繁的内存分配释放问题，如数据变小时要环缓存也跟着变小？再分配一次？还是浪费掉？

故此我设计的新的缓存系统是由一系列 4KB 的页面构成。缓存维护首指针和尾指针，当数据不断的放入缓存，尾指针不断向后移动超过一个页面时，就在这个页面后面新分配一个 4KB 的页面挂上链表；而读取的时候首指针不断的向后移动，当一页读完以后就马上归还。所有连接的缓存默认是 0KB，有数据才会分配新页面。由于页面大小固定，所以分配释放不会产生碎片，运行时间长了不会出问题。

流量控制机制

当后面的频道来不及处理用户发送的数据，而用户又持续发送数据怎么办？很多人的回答是“断开它”，我觉得“断开它”应该是服务端逻辑来做的事情，而作为一个负责任的网络层，我用了新的机制：

如果频道来不及处理，那么在网络层会表现出频道的发送缓存里的数据越来越多，当超过一定限制的时候，网络层将该频道置为“繁忙”状态，在该状态下，所有试图向该频道发送数据的用户连接都会被禁止读事件，并挪动到一个“等待列表”里面。直到该频道的发送缓存降下来（该频道反应过来处理了一定多的请求）。那么会从“等待列表”里面拿出 1 个连接来允许他们的读事件，这样他们又可以顺利的向该频道发送数据了。每次向频道成功发送一次数据（没有超过缓存限制）就这样做一次，这样如果该频道处理能力不下降的话，“等待列表”很快就会被自动清空。

服务端做这种处理，在客户端看来就是：当禁止掉客户端读事件时，server 断 socket 读缓存被填满，不会接受新的包，客户端发送失败，发送缓存被填满，客户端表现出 BLOCK 状态，直到该连接从“等待队列”里拿出来以后，发送开始生效，客户端又从 BLOCK 转换为非 BLOCK。

这样的表现和 WEB SERVER 类似，避免了粗暴的“断开它”，即使要断开，这个工作也可以由后面的频道逻辑来做。

时钟控制

并不是所有平台都能很好的支持 TCP KEEPALIVE 参数配置，因此我选择将活动性检测放在了网络层，一般的检测是一定时间扫描一下所有连接上次发送数据的时间，或者用传统 TIMER 太久没有数据的就断开。传统几个 TIMER 实现都不是 O1 的，因此我搞了个队列，每次只检测队列头，有数据的话就把该连接的节点从队列中取出，重新放到队列尾，保证队列头的节点时间最久，后面的时间最新，这样避免了循环。

内存管理

整个程序只会申请和释放两种大小的内存：连接控制符、缓存 4K 页面。因此使用一个简单的固定内存分配器就可以实现 O1 的内存分配。同时我优化了以往固定内存分配器只向系统申请内存不向系统归还内存的问题，使之能够有效的归还内存，而且尽量分配地址空间连续的内存块，这样有效避免了缓存失效。

双通道协议

每个连接默认使用 TCP 通道，但是根据需要，可以开辟 UDP 通道。即时性高的东西可以走 UDP 通道。

TAG 功能

每当某连接发送数据到某频道，频道都会知道该连接的标识，但是一般逻辑都会维护一个结构体来表述该用户的各种数据，如果每次一个数据来就要查找一次 map 的话开销太大了。这里的做法是可以让逻辑给连接设置一个整数 tag，可以是一个数组下标，或者是一个转成整数的指针，有了这个 tag 以后每次逻辑收到消息就可以根据它快速索引到对应的结构体了。

O（1）的服务器

最终整个服务端网络部分代码，没有一次搜索，没有一次循环，没有一次分配，不管运行多长时间，所有操作都可以在 O(1)的时间内完成，所有多余的可以去掉的操作全被优化掉了。

性能测试

六万人的连接，每秒钟每人发送一条信息，网络层收到后转给频道 0，频道 0 打上时间戳返回：
在一台普通 Xeon 3.0GHz 四核机器上：（kernel2.6.32 lenny）占用很低的 CPU：

```
top - 16:48:09 up 5 days,  2:45,  1 user,  load average: 0.96, 0.54, 0.26
Tasks:  91 total,   2 running,  89 sleeping,   0 stopped,   0 zombie
```

```
Cpu0  :  3.3%us,  4.0%sy,  0.0%ni, 83.4%id,  0.0%wa,  0.3%hi,  9.0%si,  0.0%st
Cpu1  :  3.7%us,  6.0%sy,  0.0%ni, 79.3%id,  0.0%wa,  0.0%hi, 11.0%si,  0.0%st
Cpu2  :  4.0%us,  5.6%sy,  0.0%ni, 81.5%id,  0.0%wa,  0.3%hi,  8.6%si,  0.0%st
Cpu3  :  4.3%us,  4.0%sy,  0.0%ni, 82.7%id,  0.0%wa,  0.0%hi,  9.0%si,  0.0%st
Cpu4  :  3.0%us,  5.0%sy,  0.0%ni, 83.9%id,  0.0%wa,  0.3%hi,  7.7%si,  0.0%st
Cpu5  :  3.6%us,  4.6%sy,  0.0%ni, 83.1%id,  0.0%wa,  0.3%hi,  8.3%si,  0.0%st
Cpu6  :  3.7%us,  4.7%sy,  0.0%ni, 85.7%id,  0.0%wa,  0.0%hi,  6.0%si,  0.0%st
Cpu7  :  6.6%us,  6.3%sy,  0.0%ni, 77.8%id,  0.0%wa,  0.3%hi,  8.9%si,  0.0%st
Mem:   8307360k total,  1250872k used,  7056488k free,   103476k buffers
Swap:  2104472k total,        0k used,  2104472k free,   225104k cached

procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 1  0      0 7056644 103492 225104    0    0     0     0    4    4  0  0 100  0
 2  0      0 7055820 103492 225108    0    0     0     0 15361 15218  4 13 83  0
 2  0      0 7055572 103492 225108    0    0     0     0 15733 15632  4 13 83  0
```

大部分 CPU 基本都是系统调用，特别是 epoll wait 和 ctrl（红黑树操作）。其次就是 send。

由于网络层基本就设计成最大 6 万个连接（内部编号用了 16 位的编号）。因此测试人数加不上去了，此时整个 transmod 每秒接收 6 万 x2=120k 个数据，发送 120k 个数据，这个 packets per seconds 基本达到了一般系统的上限（经测试，该机器上限是 160k packets/sec，）。而每个请求的处理时间加起来小于 0.1ms。

CCLIB

下面仅以 Python 接口为例，C 接口类似参考。线程不安全，只能一个线程调用 cclib。

headmode (mode)

设置头部格式（0-12），该格式需要和 conf 目录中对应服务的 transmod 头部格式设置相同，并且在 attach 之前设置（默认是 0，如果设置不对，attach 将不成功）。

attach (ip_str, port, channel)

登录到 transmod，ip_str 是字符串形式的 IP 地址，port 是 transmod 对 channel 的端口编号 PORTC，而 channel 是需要登录的频道编号。

如果频道编号为 0xffff 时，transmod 将会自动为该频道分配一个 100-2000 的频道编号，频道进程可以使用 cclib.getchid () 获得自己的频道编号。

登录成功无返回值，否则就退出进程，或者等待。注：登录前需要设置 headmode，可以用 lib/ccinit.py 里面的 attach 接口从环境变量中取出相应参数，一次性完成 headmode, attach 两个操作。

read (nowait = 0)

读取消息，返回一个四元组：event, wparam, lparam, data，比如可以用：

```
event, wparam, lparam, data = cclib.read()
```

参数 nowait 默认值是 0，代表阻塞，直到有消息返回，表或者 transmod 连接断开（此时频道进程最好自己退出，因为已经不能提供服务）。

如果 event < 0 代表 transmod 断开。

参数 nowait 如果被设置成 1 的话，每次调用 read() 时就会马上返回，可以根据 event 来判断是否有消息，如果 event == 99，那么说明没有接收到消息。如果 event >= 0 && event < 99 那么代表成功接收到消息。

send (hid, data)

向对应 hid 的客户发送数据。

close (hid, code)

断开 hid 标志的连接，code 代表断开原因（会被记录到 transmod 的日志中去）。不过断开最好是 server 端发送一条消息给客户端，客户端主动断开。如果超过时间客户端还不断开，那么再 close。

settag (hid, tag)

设置对应 hid 连接的 tag。

movec (channel, hid, data)

将客户的控制权移交给某频道，data 是附着的数据。

channel (channel, data)

频道间通信，像编号为 channel 的频道发送数据，该频道回收到 ITMT_CHANNEL 消息。

settimer (millisec)

设置时钟消息，只有 channel0 可以设置并收到 ITMT_TIMER 消息，其他频道无法收到，如果其他频道也需要，可以由 channel0 调用 cclib.channel () 来转发。

bookadd (category)

关注分类为 category 的消息，必须在 transmod 启动前将头部模式设置为 12。特殊分类 0：新连接和断开消息；特殊分类 255：时钟消息。

bookdel (category)

取消关注分类为 category 的消息，必须在 transmod 启动前将头部模式设置为 12。

bookrst ()

清空该频道的所有分类，必须在 transmod 启动前将头部模式设置为 12。

groupcase (hid_list, data)

组播，hid_list 是一个有 iterator 的容器，比如 list, dict, tuple 等，代表要广播的人群的 hid，data 是要广播的数据。

rc4_set_rkey (hid, key_str)

设置某客户端接收数据的 RC4 加密密钥，空字符串代表取消加密

rc4_set_skey (hid, key_str)

设置某客户端发送数据的 RC4 加密密钥，空字符串代表取消加密

其他语言调用 CCLIB

服务端逻辑因为使用 cclib 同 transmod 交互，而 cclib 是一个 python 的模块，如果使用 C/C++，Java 等写服务端逻辑，可以直接载入 cclib.dll / cclib.so / cclib.pyd 并且导入 cclib.h 下面定义的函数，在 unix 下面，非 python 环境不能直接载入 cclib.so，而要载入没有 python 功能的 cclib.cc（src/cclib/makeme.sh 编译时会自动生成）。

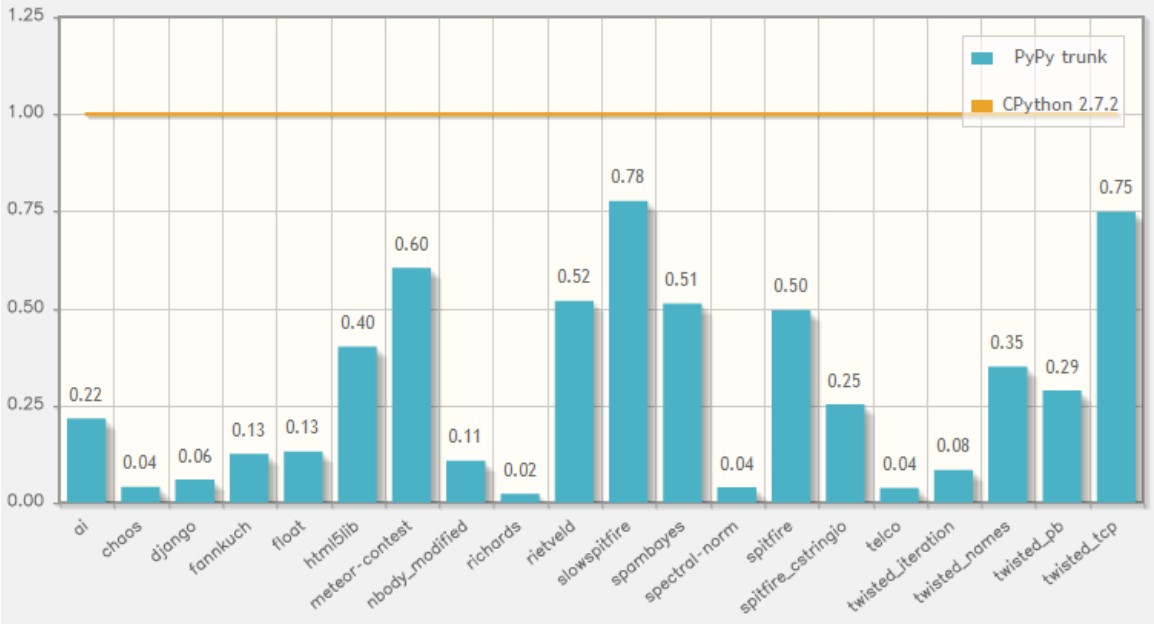
使用 PYPY 调用 CCLIB

如果用 Pypy 来代替 Python 载入 cclib 需要改两个地方：

1. 代码中要先 import ccinit，然后再 import cclib（这样会做一些 pypy 初始化）。

2. 改写频道的启动脚本，例如将 channel/echoserver/echosvr（或者 echosvr.bat）从：
   exec python echosvr.py 改为：exec pypy echosvr.py

pypy 不能载入为 python 编译的 cclib，只能以 ctypes 方式载入，该判断工作在 ccinit.py 开始就会做，故需要先 import ccinit 再 import cclib，当判断为 pypy 就会使用 cport.py。



服务配置

打开 conf/echoserver.ini 可以看到 echoserver 这个服务的配置情况：

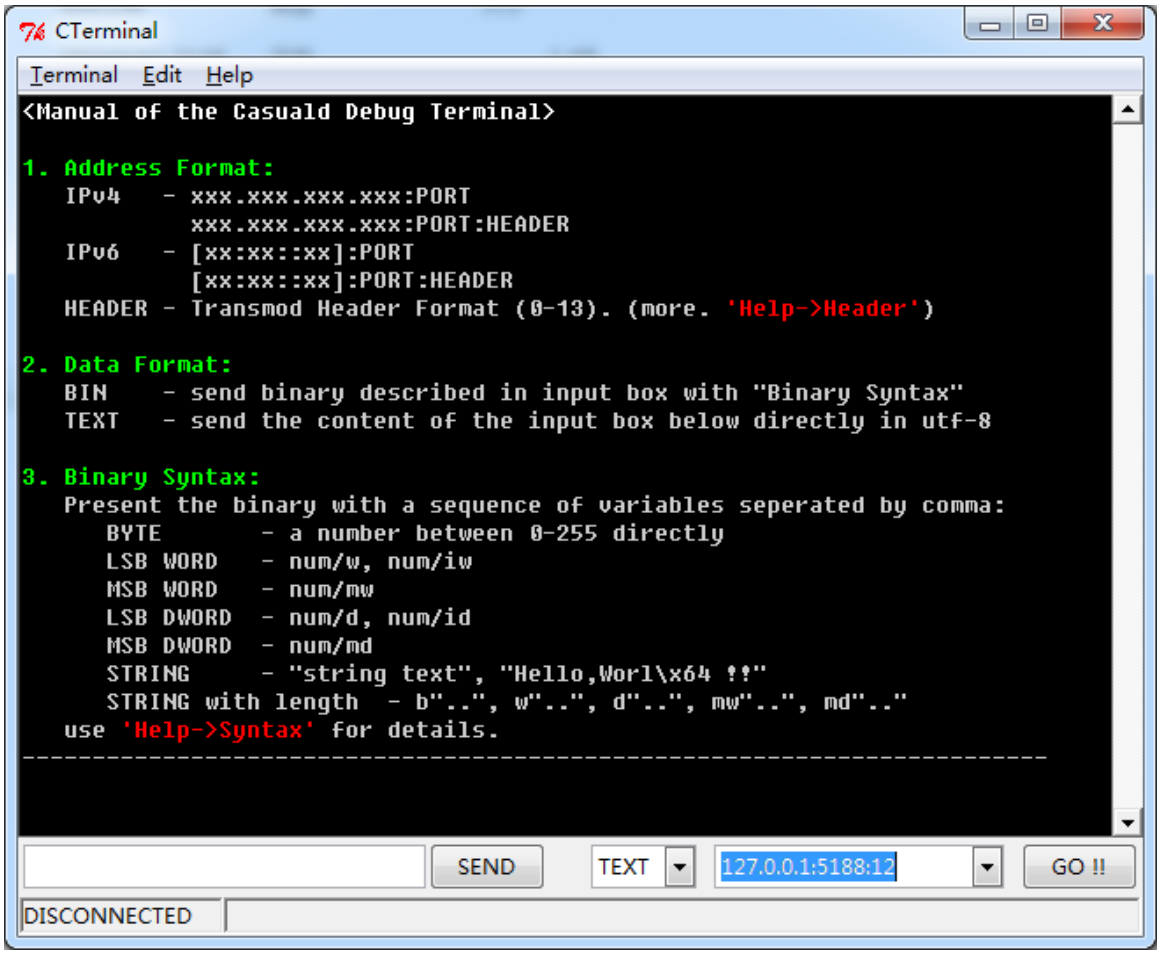
```
[TRANSMOD]
; 头部格式：头部标识长度的格式
;          0     (WORDLSB)   用两字节表示数据包长度，低位在前
;          1     (WORDMSB)   用两字节表示数据包长度，高位在前
;          2     (DWORDLSB)  用四字节表示数据包长度，低位在前
;          3     (DWORDMSB)  用四字节表示数据包长度，高位在前
;          4     (BYTELSB)   用单字节表示数据包长度，低位在前
;          5     (BYTEMSB)   用单字节表示数据包长度，高位在前
;          6     (EWORDLSB)  用两字节表示数据包长度，低位在前，不包括自己
;          7     (EWORDMSB)  用两字节表示数据包长度，高位在前，不包括自己
;          8     (EDWORDLSB) 用四字节表示数据包长度，低位在前，不包括自己
;          9     (EDWORDMSB) 用四字节表示数据包长度，高位在前，不包括自己
;          10    (EBYTELSB)  用单字节表示数据包长度，低位在前，不包括自己
;          11    (EBYTEMSB)  用单字节表示数据包长度，高位在前，不包括自己
;          12    (DWORDMASK) 用四字节表示数据包长度，低位在前，高 8 位代表分类
;          13    (RAWDATA)   对外无包头，对内 4 字节 LSB（含自己），低位在前
HEADER=0

; 用户监听端口：
; 用户新进连接将准对此端口进行，游戏客户端连接的也就是这个端口
PORTU=6000

; 频道监听端口：
; 各个频道所连接用的端口，一般是本机的进程
PORTC=6008

; IPv6 的用户端口、频道端口、数据报端口
PORTU6=6000
PORTC6=6008
PORTD6=6000

; 用户连接超时：单位秒
; 如果该时间内用户没有发送数据也没有频道向该用户发送，则断开对应连接
TIMEOUTU=180

; 频道连接超时，单位秒
; 如果该时间内频道没有发送数据也没有用户向该频道发送，则断开对应连接
TIMEOUTC=60000
```

```
; 用户缓存大小 -- 用于断开不处理数据的用户
; 如果频道向用户发送而用户有没有处理的数据，将会堆放在用户发送缓存，
; 一旦超过这个大小，用户连接将会被断开
MEMLIMITU=1M

; 频道缓存大小 -- 用于阻塞不停发送数据的用户
; 频道缓存，如果用户向频道发送而频道又没有及时处理的数据，在缓存中
; 堆放超过了该大小，对应用户的写事件会被静止并放如等待队列，直到
; 频道把那些超过的数据接收下来，缓存小于该值时才会被允许
MEMLIMITC=10M

; 最大用户连接：
; 超过该数字则不能再有用户来连接
MAXUSER=20000

; 最大频道连接：
; 超过该数字则不能再有频道来连接
MAXCHANNEL=2000

; 日志掩码设置：-- 规定要输出的日志
; 根据下面的代码定制需要显示的日志，定制多种日志显示则将他们做或运算：
;     0x01   日志代码：基本（包括服务情况：启动，关闭等）
;     0x02   日志代码：信息（包括连接情况：用户连接，断开，用户数据或连接错误）
;     0x04   日志代码：错误（包括错误情况：事件错误，频道错误）
;     0x08   日志代码：警告（包括警告情况：频道操作不正确，比如操作不存在用户）
;     0x10   日志代码：数据（包括数据情况：用户和频道间的数据收发情况）
;     0x20   日志代码：频道（包括频道情况：频道之间数据收发，如时钟数据）
;     0x40   日志代码：事件（包括事件情况：显示所有网络事件，审用，很频繁）
;     0x80   日志代码：丢包（数据报乱序情况丢包记录）
; 比如我既要显示'基本'和'信息'，又要显示'错误'和'警告'，那么日志掩码就是：0x0F
LOGMASK=0xF

; 频道监听的地址：
; 频道连接针对的监听地址，比如如果各频道程序保证和传输模块在一台机器内
; 则可以直接是 127.0.0.1，那么其他外部机器则无法连接，如果是双网卡的机器
; 则可以直接是内网 IP 以限制外网的连接
;CHADDR=127.0.0.1
CHADDR=0.0.0.0

; UDP 监听端口
; 数据报监听地址，如果填写 0 的话系统会自动指定，不过最好不要这么做
PORTD=6000

DGRAM=0
```

```
SOCKUDPB=32M

; 是否跳过客户端数据的 HTTP 头部
HTTPSKIP=0

[SERVICE]
EXEC=channel/echoserver/echosvr

[FLASH]
POLICYFILE=conf/flashpolicy.xml
POLICYPORT=843
```

TERMINAL

启动 bin/terminal.pyw 来模拟客户端调试消息，在右下角写入：IP:端口:头部格式(默认 0)



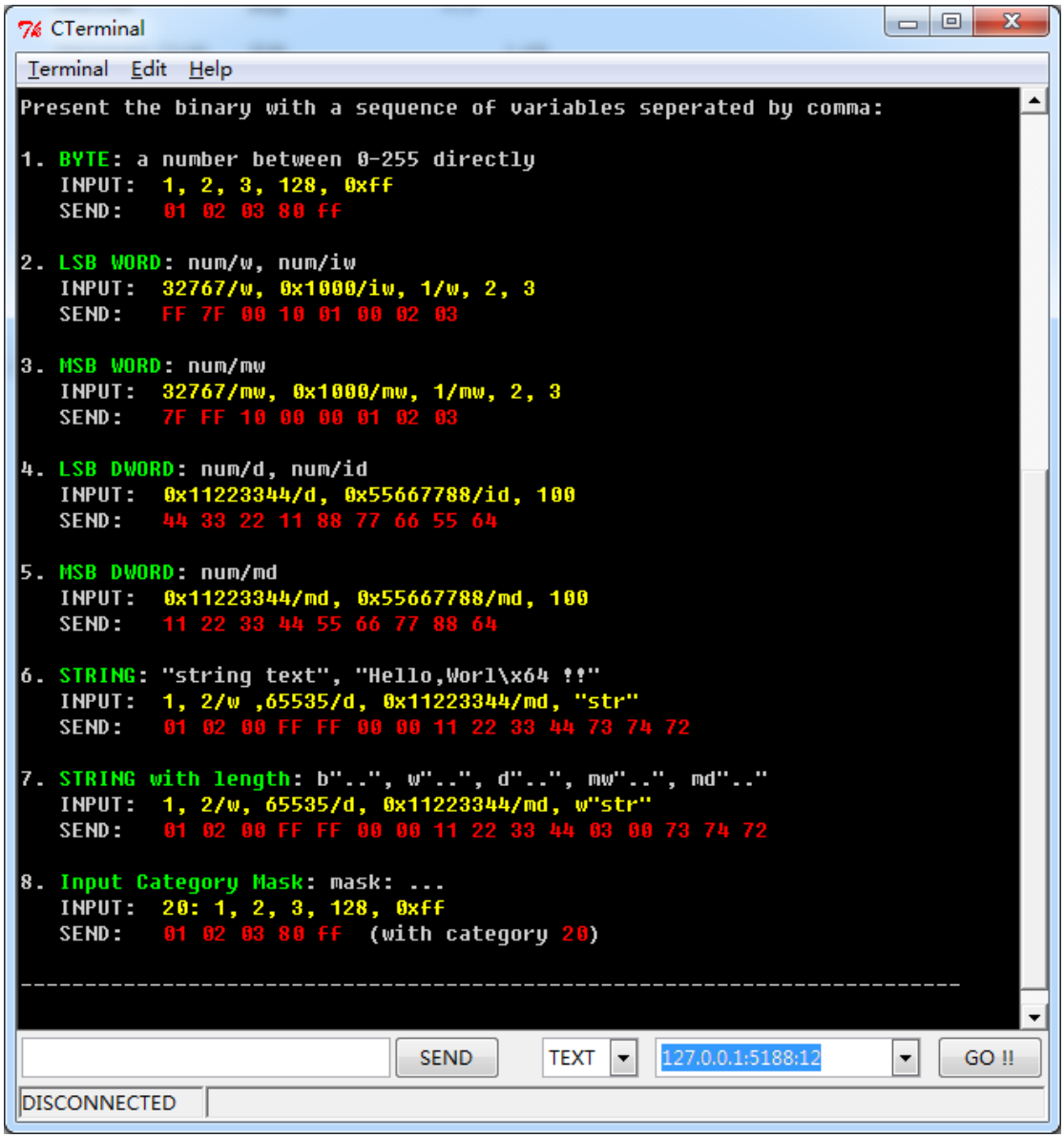
点“GO!!”连接后，如果连接成功，左下角会显示 CONNECTED 状态，中间可以控制消息格式“TEXT”和“BIN”两种格式。

文本模式下，在左下角输入框输入任何文本都会以 utf-8 的格式发送给服务器，接受到的数据也会在上面终端区以灰色字体显示出来。

二进制格式，在左下角输入框按特定的编码规则输入二进制数据，点 SEND 将会发送出去。二进制数据使用一连串以逗号间隔的数字进行描述，十六进制前面加 0x。默认直接写数字代表字节，后面加/w 代表 16 字节数字（LSB 字节序），/d 代表 32 字节数字（LSB 字节序）。二进制格式的描述方式，点击 Help->Syntax 将会出现二进制格式的帮助：

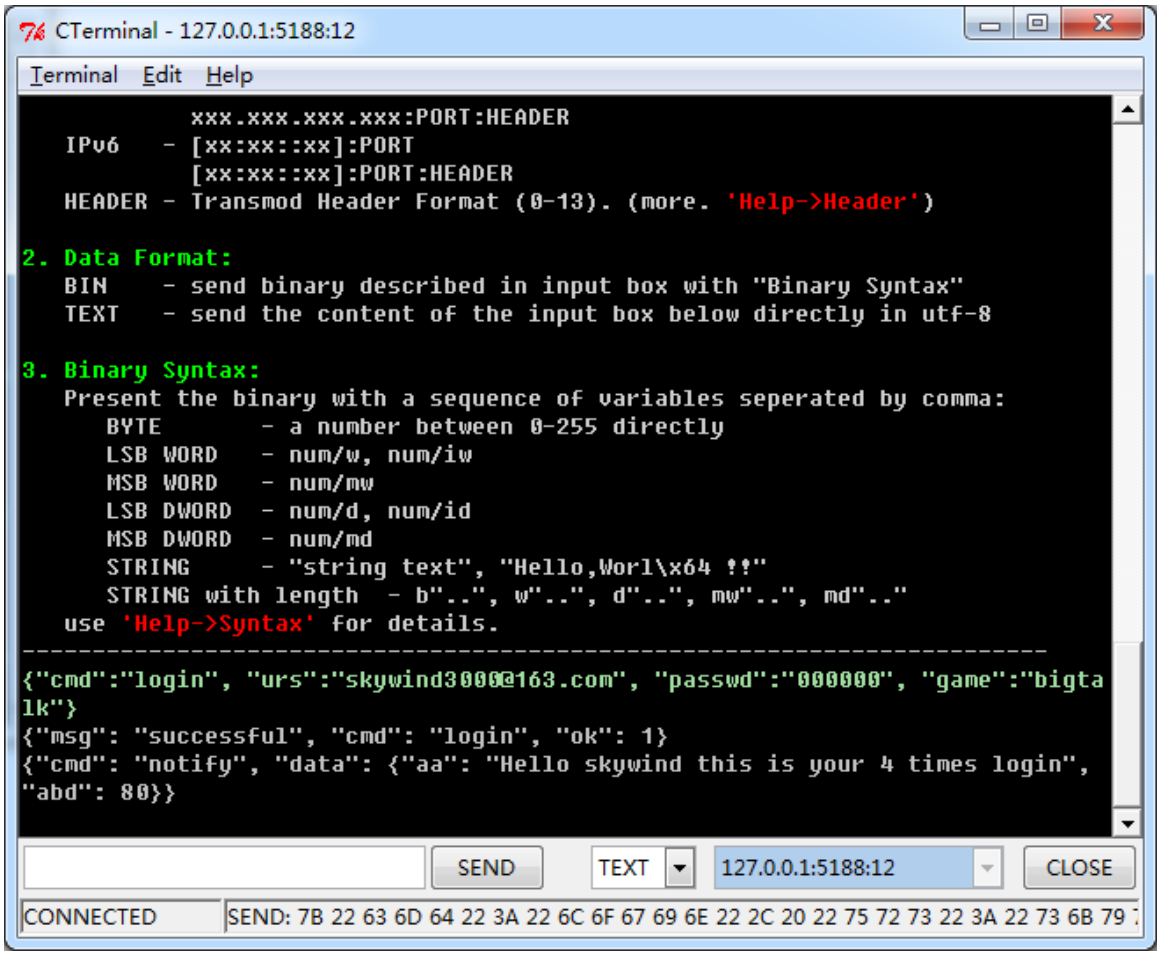
用户输入：1，2，3，4，5
数据输出：01 02 03 04 05 五个字节

用户输入：32767/w, 0x1000/iw, 1/w, 2, 3
数据输出：FF 7F 00 10 01 00 02 03 八个字节



按照这个格式可以与服务器进行二进制数据收发。

下图为文本模式和服务端收发信息的演示，自己发出去的数据是青色字体：
{“cmd”:”login”…..}
接收到的服务端数据是灰色字体：
{“msg”:”notify”,…}



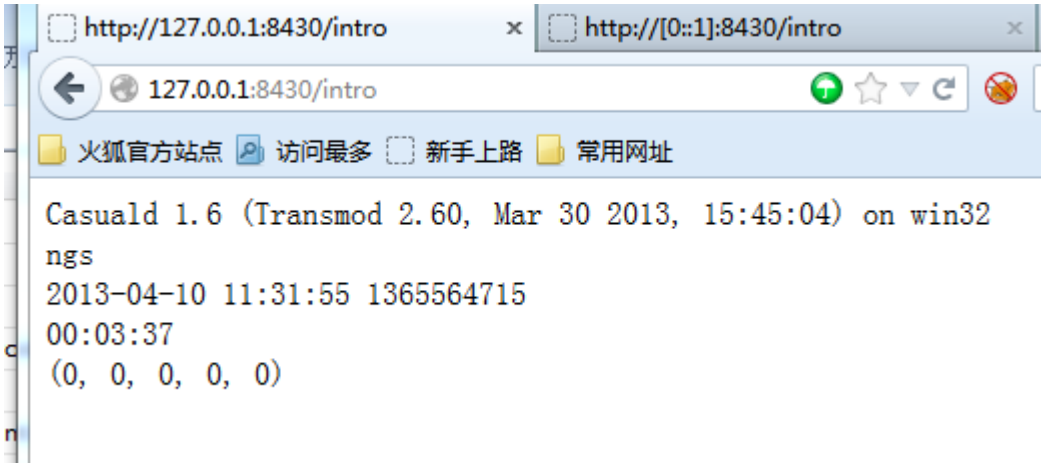
最下面状态按照二进制方式显示了你上一条发出去的数据内容。

查看状态

支持 flash policy，写在 conf/*.ini 文件里面即可，同时 POLICYPORT 端口除了提供 flash policy 服务以外，还支持 HTTP 查询 cannon 运行状态：

```
[FLASH]
POLICYFILE=conf/flashpolicy.xml
POLICYPORT=8430
```

比如浏览器打开 http://127.0.0.1:8430/intro 可以得到 cannon 版本号，正在运行的 service 名称，开始的时间戳，以及运行了多久，当前多少个用户，多少个 channel，一共发送多少个包，接收多少个包，丢弃多少个包：



注意事项

不要在多线程里面调用 cclib
即便调用了 cclib.close 关闭一个连接，也会收到 ITMT_LEAVE 消息，将 ITMT_LEAVE 作为唯一删除对象的入口。