服务器消息流动

Server类表示一个服务器线程，包括完整的消息流动过程。

该类包括三个线程：

accept\_接受连接线程，

receive\_接收数据线程，线程内开启epoll\_watch，监听EPOLLIN事件

send\_发送数据线程，线程内开启epoll\_watch，监听EPOLLOUT事件

包含两个数据结构，用来处理svc连接

Object\_Po0l<Server\_Svc, Spin\_Lock> svc\_static\_list\_ 存放svc的列表，所有client到server的连接都保存在这里

Svc\_Static\_List<Server\_Svc \*, Spin\_Lock> svc\_pool\_ svc的内存管理类

Socket流程简介：

客户端：

通过socket函数创建socketfd，connect服务器ip，port，连接成功后，可通过socketfd进行read/write与服务器进行通信。

服务端：

通过socket函数创建listenfd，bind相应的端口/ip，然后listen，等待有客户端连接。

当有客户端连接上来时候，通过accept接受连接，返回个connfd，然后可以通过connfd进行read/write，与客户端进行通信。

消息流动过程：

1、接受连接

Accept线程启动后，会在run\_handler函数里面调用server\_listen进行监听连接，有客户端连接到达时候，调用server\_accept接受该连接，

然后调用accept\_svc处理连接后的svc，将svc存到svc\_static\_list\_里面，

然后调用register\_recv\_handler将该svc注册到receiver线程的epoll里面，

调用register\_send\_handler将该svc注册到sender线程的epoll里面，等待接受和发送数据。

流程如下：

Accept::thread\_create->run\_handler->server\_listen->server\_accept->

Server\_Accept::accept\_svc->

Svc\_Static\_List::record\_svc->

Server\_Svc::register\_recv\_handler->register\_send\_handler

2、接收数据

Receive线程里面的epoll\_watch在监听到EPOLLIN事件后，会触发Event\_Hander的handle\_input事件，

因为Svc继承Event\_Handler类，receive的epoll监听了该svc，会调用Svc的handle\_input函数，

从socket缓冲区读取到的数据Block\_Buffer通过push\_recv\_block函数放到Svc类的recv\_block\_list\_里面

然后调用Svc\_Handler的handler\_pack函数进行解包，最后将buffer放到server的block\_list\_

流程如下：

Server\_Svc::register\_recv\_handler->

Receive::register\_svc->(reactor\_->add(svc, Epoll\_Watcher::EVENT\_INPUT))

Epoll\_Watcher::watcher\_loop->Svc::handle\_input->

push\_recv\_block->recv\_block\_list\_(该结构存储接收到的数据BlockBuffer)

Svc\_Hander->handle\_pack->Server::block\_list\_

3、数据处理

以Game\_Gate\_Server为例，在启动时候，会开启一个循环，循环从Server的block\_list\_中取数据，通过

GAME\_MANAGER->push\_game\_gate\_data(buf)放到Game\_Manager的game\_gate\_data\_list\_中。

然后js脚本线程启动后，会不断的从game\_gate\_data\_list\_里面pop消息进行逻辑处理

流程如下：

Game\_Gate\_Server:thread\_create->run\_handler->process\_list->

Game\_Manager::push\_game\_gate\_data

game\_server.js::pop\_game\_gate\_msg\_object->process\_game\_gate\_msg

4、发送数据

当有数据需要发送给客户端时候，调用Gate\_Manger里面的send\_to\_client，然后将数据发送给send\_，

存到append\_list\_里面，当send\_线程里面epoll监听的定时器时间到期时候，会调用send的hanlder\_timeout函数，

该函数先调用append\_send\_block将append\_list\_里面的数据传送到svc的send\_block\_list\_里面，

然后通过svc的handle\_send将数据发送到socket缓冲区，然后通过网络发送给客户端。

流程如下：

Game\_Manager::send\_to\_client->

Send::push\_data\_block\_with\_len->

Svc::append\_list\_(该结构存放待发送的Block\_Buffer)

Send:thread\_create->run\_handler->register\_self\_timer->

Epoll\_Watcher::watcher\_loop->

Send::handle\_timeout->append\_send\_block->

Svc::push\_send\_block->send\_block\_list\_(循环从list中取出数据通过socket发送)

Svc::handle\_send

5、连接关闭流程

Receive里面注册了个io心跳超时事件，当epoll监听到该事件时候，说明已经无法连接到客户端，服务器会主动关闭该连接，

首先调用epoll\_watch里面的handle\_timeout函数，然后调用到svc类里面的handle\_close函数，

然后调用Server\_Svc里面的close\_handler，该函数会通知Receive掉线的cid，放到drop\_list\_里面，

在Receive的epoll处理drop\_list\_时候，又会通过Server\_Receive的drop\_handler函数通知Send处理掉线，

放到Send的drop\_list\_里面，Send超时函数处理时候，在Send的drop\_handler里面，会回收svc，关闭fd，通知Game\_Manger，

通过Game\_Manger的process\_drop\_cid让Game\_Player掉线。

流程如下：

Receive::init->Epoll\_Watcher::WITH\_IO\_HEARTBEAT

Epoll\_Watcher::watch\_loop->handle\_timeout->

svc:handle\_close->

Server\_Svc::close\_handler->

Receive::push\_drop->drop\_list->

Server\_Receive::drop\_handler->

Send::push\_drop\_->

Server\_Send:drop\_handler->

Game\_Manager::push\_drop\_cid->drop\_cid\_list\_->

Server::svc\_static\_list\_-> erase\_svc

关于cid

cid是服务器生成的一个id,每个客户端连接到服务器后，都会生成一个唯一的cid，用来查找客户端的连接Svc,

cid保存在svc类中，在Svc::handle\_input时候，将cid写入到BlockBuffer的头部，然后消息组包，丢到逻辑层处理

当服务器需要向客户端发送消息时候，在Svc::handle\_send中，会将cid去掉，将包的内容发给客户端，所以在客户端

和服务器的通信消息中，是不带cid的