2-2-预习资料-ZeroMQ源码分析

零声学院：Darren QQ 326873713
课程地址：https://ke.qq.com/course/420945?tuin=137bb271
日　　期：20210710

官方文档API
http://api.zeromq.org/

网页版本：https://www.yuque.com/docs/share/6d6d1d2d-2ad0-48fa-8cb1-e4bf3114b31d?#
《2-2-预习资料-ZeroMQ源码分析》

编译：参考《**ZMQ编译安装和测试**》 **编译debug版本时使用 ./configure --enable-debug**

主要内容

msg_t
command_t
mailbox_t
命令交互

pipe_t session engine
tcp_listener_t 和 tcp_connecter_t
高水位标记high-water mark
重连机制
公平队列问题
性能测试

思路

- 主线程和IO线程的命令交互逻辑
  - mailbox
    - io_thread有自己的mailbox
    - socket有自己的mailbox
    - io_thread和socket怎么知道对方的mailbox?
  - 命令类型
  - 触发方式
  - io线程处理mailbox
    - io_thread_t::io_thread_t 构造时将其加入的epoll里面，然后以事件触发的方式回调
- 主线程和IO线程的数据交互逻辑
  - pipe
  - session
  - engine
  - 数据收发方式
  - 触发方式
  - 解答
    - 什么时候创建pipe
    - 什么时候创建session
    - 什么时候创建engine
    - 什么时候active
- 水位控制
  - 为什么有水位控制
- 如何保证不丢消息
  - 如何保证消息不丢失
- 重连机制
- PUB/SUB模式

active的作用

如果不使用active，每写一条命令都必须发送一个信号读读线程，在大并发的情况下，这也是一笔消耗。
而使用active，只需要在读线程睡眠的时候（没有命令可读时，io_thread_t这类线程会睡眠，

socket_base_t实例线程特殊，不会睡眠）发送信号唤醒读线程就可以，可以节省大量的资源。

大体的流程

感觉zeromq的特点是：将所有io处理由多个io线程单独完成，io线程与应用线程独立。每一个io操作的对象都与一个io线程绑定，它的所有 io操作都有该io线程完成。线程间通过一种command_t格式，通过传送命令类型和对象指针，将本该由应用线程调用的函数，交给io线程完成。 app线程发送和接收时通过队列和io线程交换数据。

处理流程

1. zmq初始化(zmq_init())。

主要实现在ctx_t::ctx_t()中，初始化了signalers数组，创建了多个io线程(io_thread_t)，每个io线程和app线程 都有一个signaler,他处理线程间的command_t，在unix中的实现是本地套接。io_thread_t在构造时，根据编译宏决定了使用 select,poll，kqueue,epoll等那种方式监听io,以后所有的io监听也都采用该方式，将io_thread_t线程的signaler也添加到监听事件。之 后调用io_thread_t->start(),开启多线程。假设采用kqueue，则最终会调用kqueue_t::loop()，kqueue_t::loop()是个while循环，当有监听事件发生时，则交给对应的hook进行处 理。对于io_thread_t线程的signaler，如果有读事件发生，则交由io_thread_t::in_event()处理。

2. 创建socket(zmq_socket())。

主要实现在ctx_t::create_socket()中。创建app线程(app_thread_t)，与一个signaler进行绑定。调用 app_thread_t::create_socket， 确定app线程该应用采用哪种连接模式:ZMQ_PAIR、ZMQ_PUB、ZMQ_SUB、ZMQ_REQ、ZMQ_REP、ZMQ_XREQ、 ZMQ_XREP、ZMQ_PULL、ZMQ_PUSH几类。分别对应pair_t、pub_t、sub_t、req_t、rep_t、xreq_t、 xrep_t、pull_t、push_t。它们继承自socket_base_t,其中每一个类实现了以下接口：xattach_pipes()、 xdetach_inpipe()、xdetach_outpipe()、xkill()、xrevive()、xsetsockopt()、 xsend()、xrecv()、xhas_in()、xhas_out()。程序中调用的发送和接收数据会调用到具体某个模式的 xsend(),xrecv()。

3 监听连接(zmq_bind())

解析地址类型inproc，tcp，ipc。这里以tcp为例，创建一个zmq_listener_t对象，调用zmq_listener_t::set_address()完成socket(),bind(),listen()系统调用。调用 send_plug(),向io线程发送command_t,类型为plug，对象指针为刚创建的zmq_listener_t。io线程接到该命令后， 最终会有zmq_listener_t::process_plug()来处理，这里完成在kqueue上添加监听tcp的fd。该fd上收到连接请求时，交由zmq_listener_t::in_event()处理。

4 服务端建立连接

当有新的连接时，zmq_listener_t::in_event()调用tcp_listener::accept()完成accept()操作，获 取新连接的fd。选择一个负载最低的io线程，创建一个 zmq_init_t对象，调用send_plug(),对象指针为zmq_init_t,负载最低io线程接到该命令后，完成以下调用 zmq_init_t::process_plug()->zmq_engine::plug(),zmq_engine::plug()中在 kqueue上添加监听新的fd。

5 客户端建立连接(zmq_connect())

以tcp为例，socket_base_t::connect()中，创建session_t(用于app线程与io线程发送与接收时数据的传送)，构造 zmq_connecter_t，调用zmq_connecter_t::set_address()设置连接地址。调用send_plug(),对象指 针为zmq_connecter_t。io线程接到该命令后，完成以下调用zmq_connecter_t::process_plug()->zmq_connecter_t::start_connecting()->tcp_connecter_t::open ()，tcp_connecter_t::open () 中会完成socket (),connect()操作，在kqueue上添加监听该连接。

6 数据的接收（zmq_recv())

连接建立起来后，收到读事件则由zmq_engine::in_event()处理。调用tcp_socket_t::read()完成recv()系统调用。然后调用zmq_init::flush->zmq_init_t::finalise ()，创建一个session_t,调用send_attach()发送attach命令。io线程收到命令后调用 session_t::process_attach(),process_attach()中创建pipe_t（pipe_t的实现是一个队列），调用send_bind()发送bind命令。io线程收到命令 后调用session_t::process_attach(),将创建的pipe_t与对应的app线程对应的连接模式进行关联,之后app线程的接收 数据都从pipe_t中获取。

命令command_t

命令结构

C++ 复制代码

```
//  This structure defines the commands that can be sent between threads.
    struct command_t
    {
        //  Object to process the command.
        zmq::object_t *destination;

        enum type_t
        {
           ...
        } type;

        union {
           ...
        } args;
    };
```

位置：command.hpp

概述：一个command由三部分组成：**分别是发往的目的地destination，命令的类型type，命令的参数args**。所谓的命令就是一个对象交代另一个对象去做某件事情，说白了就是告诉另一个对象应该调用哪个方法，命令的发出者是一个对象，而接收者是一个线程，线程接收到命令后，根据目的地派发给相应的对象做处理。可以看到命令的destination属性是object_t类型的，在上节介绍类的层次结构图时，说到object_t及其子类都具有发送和处理命令的功能（没有收命令的功能），所以有必要弄清楚一件事，对象、object_t、poller、线程、mailbox_t、命令是什么关系?

- 在zmq中，每个线程都会拥有一个信箱，命令收发功能底层都是由信箱实现的
- zmq提供了object_t类，用于使用线程信箱发送命令的功能（object_t类还有其他的功能），object_t还有处理命令的功能。
- io线程内还有一个poller用于监听激活mailbox_t的信号，线程收到激活信号后，会去mailbox_t中读命令，然后把命令交由object_t处理

简单来说就是，object_t发命令，poller监听命令到来信号告知线程收命令，交给object_t处理。无论是object_t、还是线程本身、还是poller其实都操作mailbox_t，object_t、poller、mailbox_t都绑定在同一个线程上。

种类：command有21种可能的情况

1. stop：发给io thread以终止该线程；
2. plug：发送给io object以使其在所在的io thread注册；
3. own：发送给socket以让它知道新创建的对象，并建立own关系；
4. attach：将引擎engine连接至会话session，如果engine为null的话，告知session失败；
5. bind：从会话session发送到套接字socket以在他们之间建立管道pipe，调用者事先调用了inc_seqnum发送命令（增加接收方中的计数 sent_seqnum）；
6. activate_read：由pipe wirter告知pipe reader管道中有message；
7. activate_write：由pipe reader告知pipe writer目前已读取的message数；
8. hiccup：创建新inpipe之后由pipe reader发送给writer，它的类型应该是pipe_t::upipe_t，然而这个类型的定义是private的，所以我们使用void*定义；
9. pipe_term：由pipe reader发送给pipe writer以使其term其的末端；
10. pipe_term_ack：pipewriter确认pipe_term命令；
11. pipe_hwm：由一个pipe发送到另一部分以修改hwm（高水位线）；
12. term_req：由一个i/o object发送给套接口以请求关闭该i/o object；
13. term：由套接口发送给i/o object以启动；
14. term_ack：由i/o object发送给套接口以确认其已经关闭；
15. term_endpoint：由session_base（I/O thread）发送到套接字（app thread）以请求断开端点。
16. reap：将已closed的socket的所有权转移到reaper thread；
17. reaped：已closed的socket通知reaper thread它已经被解除分配；
18. inproc_connected
19. pipe_peer_stats
20. pipe_stats_publish
21. done：当所有的socket都成功被解除分配之后，由收割者线程发送给term thread。

邮箱mailbox_t

位置：mailbox.hpp, mailbox.cpp, i_mailbox.hpp

简述：是每一个真正的thread中处理命令流的组件。

主要变量列表：

• cpipe_t _cpipe：cpipe即command pipe，是用于存储真正的命令的管道，实现了单一读/写线程的无锁访问。（其中粒度的含义为，每一次内存操作会在底层的pipe中分配出16个command_t的size的内存空间）

• signaler_t _signaler：从writer管道到reader管道用于通知命令到达的信号机。

• mutex_t _sync：互斥量，只有一个线程从信箱接收命令，但是有任意数量的线程向信箱发送命令，由于ypipe_t需要在它的两个端点进行同步访问，所以我们需要同步发送端。

• bool *active：command pipe是否是可用的。*

主要接口：

• fd_t get_fd()：（备注typedef SOCKET fd_t）得到信箱所对应的读文件标识符。

• bool valid()：检测mailbox的有效性，本质上就是检测信号机的有效性。

• void send(const command_t &cmd)：发送命令。在实际使用的过程中其实是首先找到目的mailbox对象，然后调用此对象的send函数，把消息放入到这个对象的queue中。并不是从一个mailbox能够发送消息到另外一个mailbox。send的含义更像是把消息放入到函数所属对象的queue中。

C++ 复制代码

```
void zmq::mailbox_t::send (const command_t &cmd_)
{
    _sync.lock ();      //加锁
    _cpipe.write (cmd_, false); //写消息到管道
    const bool ok = _cpipe.flush ();        // 是消息对读线程可见
    _sync.unlock ();                        // 解锁
    if (!ok)
        _signaler.send ();                  // 当reader不处于alive时则触发其alive读取
}
```

- *int recv(command_t *cmd_, int timeout_)*：接收命令，把command读到参数cmd_里面，第二个参数是延迟时长，表示信号机等待信号的时长上限。

C++ 复制代码

```cpp
int zmq::mailbox_t::recv (command_t *cmd_, int timeout_)
{
    //  Try to get the command straight away 先尝试直接读取命令，如果读到了命令则直接返回
    if (_active) {  //开始的时候，信箱是不活跃状态
        if (_cpipe.read (cmd_))
            return 0;

        //  If there are no more commands available, switch into passive state.
        _active = false; //如果读取失败说明当前mailbox中没有未处理的命令，那么把状态设置为不活跃
    }

    //  Wait for signal from the command sender.
    int rc = _signaler.wait (timeout_);// 等待信号，如果有信号到达说明有命令到达了mailbox。（函数会阻塞在此）
    if (rc == -1) {
        errno_assert (errno == EAGAIN || errno == EINTR);//这里对应wait的前两种情况
        return -1;
    }

    //  Receive the signal.
    rc = _signaler.recv_failable ();    //把socket的数据读取出来，只是为了能够下次继续触发
    if (rc == -1) {
        errno_assert (errno == EAGAIN);
        return -1;
    }

    //  Switch into active state.
    _active = true; //收到信号说明有命令要处理。此时把mailbox状态设置为活跃

    //  Get a command.
    const bool ok = _cpipe.read (cmd_); // 读取命令
    zmq_assert (ok);
    return 0;
}
```

- zmq::mailbox_t::~mailbox_t(): 析构函数

```C++
zmq::mailbox_t::~mailbox_t ()
{
    //  TODO: Retrieve and deallocate commands inside the _cpipe.

    // Work around problem that other threads might still be in our
    // send() method, by waiting on the mutex before disappearing.
    _sync.lock ();
    _sync.unlock ();
}
```

注意 ！：

在析构函数中，同步机执行了一个“加锁”后立即“解锁”的流程，因为其他线程的此时仍有可能正在使用此组件的send()方法，这样可以在消失之前在mutex中等待一段时间。

其实：

其实一条命令的发送的流程可以被表示为：将command压入接收方的pipe（此时接收方不知情） —> 用信号机告知接收方“你有新的command待处理” —> 接收方调用recv取出队列中刚刚被压入的command。

补充 ⚠：

先来想一个问题，既然signaler可作为信号通知，**为何还要active这个属性?**

active和signaler是这样合作的：写命令线程每写一条命令，先去检查读命令线程是否阻塞，如果阻塞，会调用读命令线程mailbox_t中的signaler，发送一个激活读线程mailbox_t的信号，读线程收到这个信号后在recv函数中把activ设置为true，这时，读线程循环调用recv的时候，发现active为true，就会一直读命令，直到没命令可读时，才会把active设置为false，等待下一次信号到来。

现在可以回答上面那个问题了，active是否多余?

先试想一下如果不使用active，每写一条命令都必须发送一个信号到读线程，在大并发的情况下，这也是一笔消耗。而使用active，只需要在读线程睡眠的时候（没有命令可读时，io_thread_t这类线程会睡眠，socket_base_t实例线程特殊，不会睡眠）发送信号唤醒读线程就可以，可以节省大量的资源。

io_thread_t的处理

```
zmq::io_thread_t::io_thread_t (ctx_t *ctx_, uint32_t tid_) :
    object_t (ctx_, tid_),
    _mailbox_handle (static_cast<poller_t::handle_t> (NULL))
{
    _poller = new (std::nothrow) poller_t (*ctx_);
    alloc_assert (_poller);
    if (_mailbox.get_fd () != retired_fd) {
```

```
        _mailbox_handle = _poller->add_fd (_mailbox.get_fd (), this);       // 监听mailbox 命令
        _poller->set_pollin (_mailbox_handle);  //
    }
}
```

当有命令时的触发

```
void zmq::io_thread_t::in_event ()
{
    //  TODO: Do we want to limit number of commands I/O thread can
    //  process in a single go?
    command_t cmd;
    int rc = _mailbox.recv (&cmd, 0);
    while (rc == 0 || errno == EINTR) {
        if (rc == 0)
            cmd.destination->process_command (cmd); // 处理命令
        rc = _mailbox.recv (&cmd, 0);               // 收到命令
    }
    errno_assert (rc != 0 && errno == EAGAIN);
}
```

命令交互

这里是以req-rep为例 debug了部分函数

```
Connecting to hello world server...
[New Thread 0x7ffff6a60700 (LWP 10152)]
[New Thread 0x7ffff625f700 (LWP 10153)]
Thread 1 "hwclient" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at
src/mailbox.cpp:61
61      _sync.lock ();
```

plug

```
(gdb) bt
#0  zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1  0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...)
at src/ctx.cpp:484
#2  0x00007ffff7b38875 in zmq::object_t::send_command (this=0x618890, cmd_=...) at
src/object.cpp:533
```

```
#3  0x00007ffff7b3743c in zmq::object_t::send_plug (this=0x618890, destination_=0x619510,
inc_seqnum_=true)
    at src/object.cpp:233
#4  0x00007ffff7b3c5ff in zmq::own_t::launch_child (this=0x618890, object_=0x619510) at
src/own.cpp:87
#5  0x00007ffff7b60405 in zmq::socket_base_t::add_endpoint (this=0x618890,
endpoint_pair_=...,
    endpoint_=0x619510, pipe_=0x621be0) at src/socket_base.cpp:1020
#6  0x00007ffff7b5ff97 in zmq::socket_base_t::connect (this=0x618890,
    endpoint_uri_=0x400c2c "tcp://localhost:5555") at src/socket_base.cpp:984
#7  0x00007ffff7b8b93f in zmq_connect (s_=0x618890, addr_=0x400c2c
"tcp://localhost:5555") at src/zmq.cpp:313
#8  0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:18
(gdb) c
Continuing.
Thread 1 "hwclient" hit Breakpoint 7, zmq::mailbox_t::send (this=0x618f30, cmd_=...) at
src/mailbox.cpp:61
61      _sync.lock ();
```

own

```
(gdb) bt
#0  zmq::mailbox_t::send (this=0x618f30, cmd_=...) at src/mailbox.cpp:61
#1  0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=3, command_=...)
at src/ctx.cpp:484
#2  0x00007ffff7b38875 in zmq::object_t::send_command (this=0x618890, cmd_=...) at
src/object.cpp:533
#3  0x00007ffff7b374ec in zmq::object_t::send_own (this=0x618890, destination_=0x618890,
object_=0x619510)
    at src/object.cpp:243
#4  0x00007ffff7b3c616 in zmq::own_t::launch_child (this=0x618890, object_=0x619510) at
src/own.cpp:90
#5  0x00007ffff7b60405 in zmq::socket_base_t::add_endpoint (this=0x618890,
endpoint_pair_=...,
    endpoint_=0x619510, pipe_=0x621be0) at src/socket_base.cpp:1020
#6  0x00007ffff7b5ff97 in zmq::socket_base_t::connect (this=0x618890,
    endpoint_uri_=0x400c2c "tcp://localhost:5555") at src/socket_base.cpp:984
#7  0x00007ffff7b8b93f in zmq_connect (s_=0x618890, addr_=0x400c2c
"tcp://localhost:5555") at src/zmq.cpp:313
```

#8 0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:18

(gdb) c
Continuing.
[Switching to Thread 0x7ffff625f700 (LWP 10153)]
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61   _sync.lock ();

process_plug

(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x619510, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b3743c in zmq::object_t::send_plug (this=0x619510,
destination_=0x7ffff00008c0,
    inc_seqnum_=true) at src/object.cpp:233
#4 0x00007ffff7b3c5ff in zmq::own_t::launch_child (this=0x619510, object_=0x7ffff00008c0) at src/own.cpp:87
#5 0x00007ffff7b5797b in zmq::session_base_t::start_connecting (this=0x619510, wait_=false)
    at src/session_base.cpp:623
#6 0x00007ffff7b56772 in zmq::session_base_t::process_plug (this=0x619510) at src/session_base.cpp:331
#7 0x00007ffff7b36e6f in zmq::object_t::process_command (this=0x619510, cmd_=...) at src/object.cpp:87
#8 0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#9 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#10 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#11 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#12 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#13 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c
Continuing.
A 正在发送 Hello 0...
send fnish

Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61 _sync.lock ();

process_plug

(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x619510, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b374ec in zmq::object_t::send_own (this=0x619510, destination_=0x619510, object_=0x7ffff00008c0)
at src/object.cpp:243
#4 0x00007ffff7b3c616 in zmq::own_t::launch_child (this=0x619510, object_=0x7ffff00008c0) at src/own.cpp:90
#5 0x00007ffff7b5797b in zmq::session_base_t::start_connecting (this=0x619510, wait_=false)
at src/session_base.cpp:623
#6 0x00007ffff7b56772 in zmq::session_base_t::process_plug (this=0x619510) at src/session_base.cpp:331
#7 0x00007ffff7b36e6f in zmq::object_t::process_command (this=0x619510, cmd_=...) at src/object.cpp:87
#8 0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#9 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#10 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#11 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#12 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#13 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c
Continuing.
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61 _sync.lock ();

attach

(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484

#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x7ffff00008c0, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b375ad in zmq::object_t::send_attach (this=0x7ffff00008c0, destination_=0x619510,
    engine_=0x7ffff0001730, inc_seqnum_=true) at src/object.cpp:257
#4 0x00007ffff7b6c56a in **zmq::stream_connecter_base_t::create_engine** (this=0x7ffff00008c0, fd=10,
    local_address_="tcp://127.0.0.1:58216") at src/stream_connecter_base.cpp:181
#5 0x00007ffff7b75d3f in **zmq::tcp_connecter_t::out_event** (this=0x7ffff00008c0) at src/tcp_connecter.cpp:114
#6 0x00007ffff7b23511 in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:202
#7 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#8 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#9 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#10 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c
Continuing.
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61   _sync.lock ();

term_req

(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x7ffff00008c0, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b37c41 in zmq::object_t::send_term_req (this=0x7ffff00008c0, destination_=0x619510,
    object_=0x7ffff00008c0) at src/object.cpp:364
#4 0x00007ffff7b3c7af in zmq::own_t::terminate (this=0x7ffff00008c0) at src/own.cpp:147
#5 0x00007ffff7b6c576 in zmq::stream_connecter_base_t::create_engine (this=0x7ffff00008c0, fd=10,
    local_address_="tcp://127.0.0.1:58216") at src/stream_connecter_base.cpp:184
#6 0x00007ffff7b75d3f in zmq::tcp_connecter_t::out_event (this=0x7ffff00008c0) at src/tcp_connecter.cpp:114
#7 0x00007ffff7b23511 in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:202

#8 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#9 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#10 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#11 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c
Continuing.
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61   _sync.lock ();

process_term_req

(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x619510, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b37ce0 in zmq::object_t::send_term (this=0x619510, destination_=0x7ffff00008c0, linger_=-1)
    at src/object.cpp:373
#4 0x00007ffff7b3c6c7 in zmq::own_t::process_term_req (this=0x619510, object_=0x7ffff00008c0)
    at src/own.cpp:115
#5 0x00007ffff7b3705a in zmq::object_t::process_command (this=0x619510, cmd_=...) at src/object.cpp:137
#6 0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#7 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#8 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#9 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#10 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#11 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c
Continuing.
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61   _sync.lock ();

term_ack

```
(gdb) bt
#0  zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1  0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...)
at src/ctx.cpp:484
#2  0x00007ffff7b38875 in zmq::object_t::send_command (this=0x7ffff00008c0, cmd_=...) at
src/object.cpp:533
#3  0x00007ffff7b37d6c in zmq::object_t::send_term_ack (this=0x7ffff00008c0,
destination_=0x619510)
    at src/object.cpp:381
#4  0x00007ffff7b3cb02 in zmq::own_t::check_term_acks (this=0x7ffff00008c0) at
src/own.cpp:202
#5  0x00007ffff7b3c90e in zmq::own_t::process_term (this=0x7ffff00008c0, linger_=-1) at
src/own.cpp:170
#6  0x00007ffff7b6c084 in zmq::stream_connecter_base_t::process_term (this=0x7ffff00008c0,
linger_=-1)
    at src/stream_connecter_base.cpp:97
#7  0x00007ffff7b75c61 in zmq::tcp_connecter_t::process_term (this=0x7ffff00008c0,
linger_=-1)
    at src/tcp_connecter.cpp:90
#8  0x00007ffff7b3707f in zmq::object_t::process_command (this=0x7ffff00008c0, cmd_=...) at
src/object.cpp:141
#9  0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#10 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#11 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at
src/poller_base.cpp:139
#12 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#13 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#14 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c
Continuing.
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x618f30, cmd_=...) at
src/mailbox.cpp:61
61      _sync.lock ();
```

activate_write

```
(gdb) bt
#0  zmq::mailbox_t::send (this=0x618f30, cmd_=...) at src/mailbox.cpp:61
#1  0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=3, command_=...)
at src/ctx.cpp:484
```

#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x621cf0, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b3779d in zmq::object_t::send_activate_write (this=0x621cf0, destination_=0x621be0,
    msgs_read_=0) at src/object.cpp:289
#4 0x00007ffff7b3f73c in zmq::pipe_t::read (this=0x621cf0, msg_=0x7ffff0001740) at src/pipe.cpp:215
#5 0x00007ffff7b55be6 in zmq::session_base_t::pull_msg (this=0x619510, msg_=0x7ffff0001740)
    at src/session_base.cpp:154
#6 0x00007ffff7b71b5b in zmq::stream_engine_t::pull_and_encode (this=0x7ffff0001720, msg_=0x7ffff0001740)
    at src/stream_engine.cpp:1023
#7 0x00007ffff7b6e992 in zmq::stream_engine_t::out_event (this=0x7ffff0001720) at src/stream_engine.cpp:403
#8 0x00007ffff7b6ec64 in zmq::stream_engine_t::restart_output (this=0x7ffff0001720)
    at src/stream_engine.cpp:462
#9 0x00007ffff7b70fd0 in zmq::stream_engine_t::process_handshake_command (this=0x7ffff0001720,
    msg_=0x7ffff0001e98) at src/stream_engine.cpp:888
#10 0x00007ffff7b6e703 in zmq::stream_engine_t::in_event_internal (this=0x7ffff0001720)
    at src/stream_engine.cpp:365
#11 0x00007ffff7b6e2de in zmq::stream_engine_t::in_event (this=0x7ffff0001720) at src/stream_engine.cpp:302
#12 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#13 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
---Type <return> to continue, or q <return> to quit---
#14 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#15 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#16 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
(gdb) c

process_activate_write

Thread 1 "hwclient" hit Breakpoint 9, zmq::pipe_t::process_activate_write (this=0x621be0, msgs_read_=0)
    at src/pipe.cpp:283
283   _peers_msgs_read = msgs_read_;
(gdb) bt
#0 zmq::pipe_t::process_activate_write (this=0x621be0, msgs_read_=0) at src/pipe.cpp:283

#1 0x00007ffff7b36e37 in zmq::object_t::process_command (this=0x621be0, cmd_=...) at src/object.cpp:79
#2 0x00007ffff7b61643 in zmq::socket_base_t::process_commands (this=0x618890, timeout_=-1, throttle_=false)
    at src/socket_base.cpp:1369
#3 0x00007ffff7b610a0 in zmq::socket_base_t::recv (this=0x618890, msg_=0x7fffffffe3a0, flags_=0)
    at src/socket_base.cpp:1250
#4 0x00007ffff7b8c008 in s_recvmsg (s_=0x618890, msg_=0x7fffffffe3a0, flags_=0) at src/zmq.cpp:454
#5 0x00007ffff7b8c164 in zmq_recv (s_=0x618890, buf_=0x7fffffffe420, len_=10, flags_=0) at src/zmq.cpp:479
#6 0x0000000000400b12 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:26

Continuing.
Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::mailbox_t::send (this=0x618f30, cmd_=...) at src/mailbox.cpp:61
61   _sync.lock ();

activate_read

(gdb) bt 通知主线程 可读了
#0 zmq::mailbox_t::send (this=0x618f30, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=3, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x621cf0, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b376fc in zmq::object_t::send_activate_read (this=0x621cf0, destination_=0x621be0)
    at src/object.cpp:279
#4 0x00007ffff7b3fa4a in zmq::pipe_t::flush (this=0x621cf0) at src/pipe.cpp:269
#5 0x00007ffff7b55f35 in zmq::session_base_t::flush (this=0x619510) at src/session_base.cpp:217
#6 0x00007ffff7b6e774 in zmq::stream_engine_t::in_event_internal (this=0x7ffff0001720)
    at src/stream_engine.cpp:381
#7 0x00007ffff7b6e2de in zmq::stream_engine_t::in_event (this=0x7ffff0001720) at src/stream_engine.cpp:302
#8 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#9 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139

#10 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#11 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#12 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

activate_write

(gdb) c
Continuing.
[Switching to Thread 0x7ffff7fd8740 (LWP 10151)]
Thread 1 "hwclient" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61 _sync.lock ();
(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x621be0, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b3779d in zmq::object_t::send_activate_write (this=0x621be0, destination_=0x621cf0,
msgs_read_=0) at src/object.cpp:289
#4 0x00007ffff7b3f73c in zmq::pipe_t::read (this=0x621be0, msg_=0x7fffffffe3a0) at src/pipe.cpp:215
#5 0x00007ffff7b246d1 in zmq::fq_t::recvpipe (this=0x618ea0, msg_=0x7fffffffe3a0, pipe_=0x7fffffffe230)
at src/fq.cpp:93
#6 0x00007ffff7b1d322 in zmq::dealer_t::recvpipe (this=0x618890, msg_=0x7fffffffe3a0, pipe_=0x7fffffffe230)
at src/dealer.cpp:143
#7 0x00007ffff7b50aa6 in zmq::req_t::recv_reply_pipe (this=0x618890, msg_=0x7fffffffe3a0) at src/req.cpp:251
#8 0x00007ffff7b5075f in zmq::req_t::xrecv (this=0x618890, msg_=0x7fffffffe3a0) at src/req.cpp:162
#9 0x00007ffff7b610dc in zmq::socket_base_t::recv (this=0x618890, msg_=0x7fffffffe3a0, flags_=0)
at src/socket_base.cpp:1253
#10 0x00007ffff7b8c008 in s_recvmsg (s_=0x618890, msg_=0x7fffffffe3a0, flags_=0) at src/zmq.cpp:454
#11 0x00007ffff7b8c164 in zmq_recv (s_=0x618890, buf_=0x7fffffffe420, len_=10, flags_=0) at src/zmq.cpp:479

#12 0x0000000000400b12 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:26
(gdb) c
Continuing.
接收到Darren 12345 0
A 正在发送 Hello 1...
Thread 1 "hwclient" hit Breakpoint 7, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
61   _sync.lock ();

activate_read

(gdb) bt 通知对方有数据可以读取了
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:61
#1 0x00007ffff7b08ed3 in zmq::ctx_t::send_command (this=0x6141b0, tid_=2, command_=...) at src/ctx.cpp:484
#2 0x00007ffff7b38875 in zmq::object_t::send_command (this=0x621be0, cmd_=...) at src/object.cpp:533
#3 0x00007ffff7b376fc in zmq::object_t::send_activate_read (this=0x621be0, destination_=0x621cf0)
    at src/object.cpp:279
#4 0x00007ffff7b3fa4a in zmq::pipe_t::flush (this=0x621be0) at src/pipe.cpp:269
#5 0x00007ffff7b2a909 in zmq::lb_t::sendpipe (this=0x618ed8, msg_=0x7fffffffe3a0, pipe_=0x0) at src/lb.cpp:147
#6 0x00007ffff7b1d2ee in zmq::dealer_t::sendpipe (this=0x618890, msg_=0x7fffffffe3a0, pipe_=0x0)
    at src/dealer.cpp:138
#7 0x00007ffff7b1d1ac in zmq::dealer_t::xsend (this=0x618890, msg_=0x7fffffffe3a0) at src/dealer.cpp:102
#8 0x00007ffff7b50538 in zmq::req_t::xsend (this=0x618890, msg_=0x7fffffffe3a0) at src/req.cpp:118
#9 0x00007ffff7b60b17 in zmq::socket_base_t::send (this=0x618890, msg_=0x7fffffffe3a0, flags_=0)
    at src/socket_base.cpp:1123
#10 0x00007ffff7b8ba00 in s_sendmsg (s_=0x618890, msg_=0x7fffffffe3a0, flags_=0) at src/zmq.cpp:338
#11 0x00007ffff7b8bb48 in zmq_send (s_=0x618890, buf_=0x400c5d, len_=5, flags_=0) at src/zmq.cpp:370
#12 0x0000000000400aee in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:24
(gdb)

组件间如何连系?

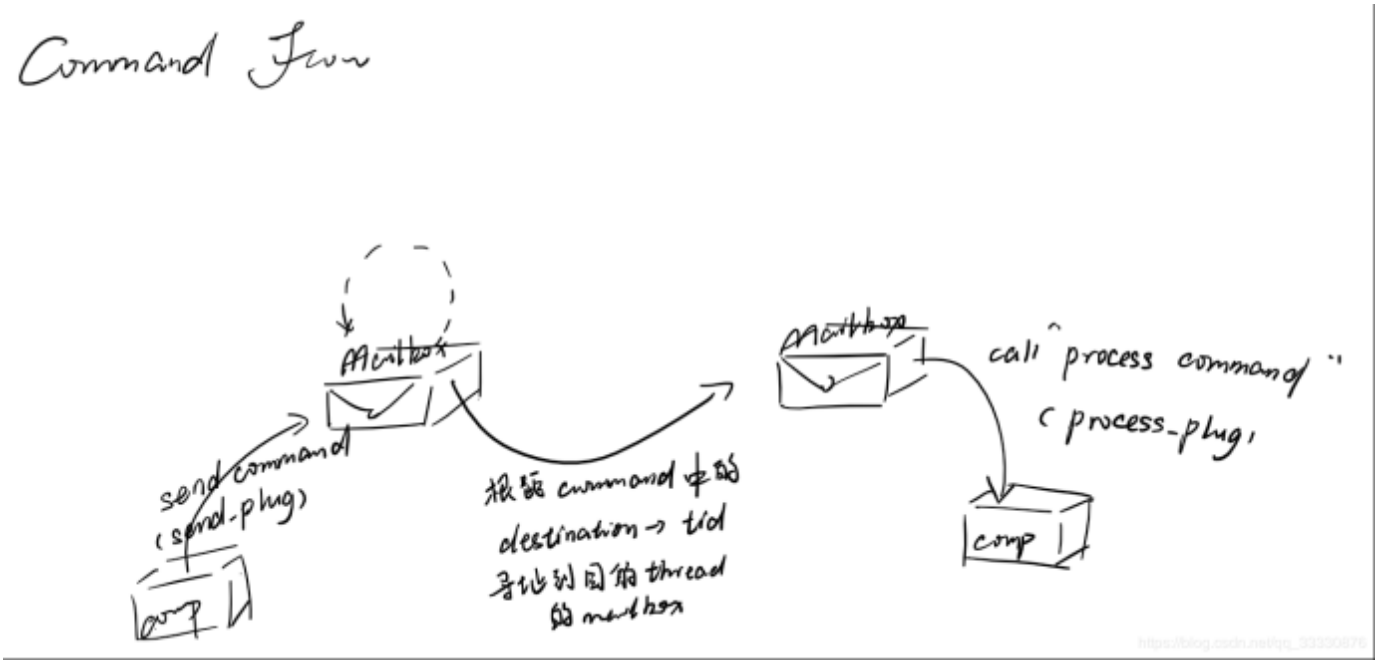
说到object_t及其子类都具有发送和处理命令的功能（没有收命令的功能），所以有必要弄清楚一件事，object_t、poller、线程、mailbox_t、command是什么关系?

- 在ZMQ中，每个线程都会拥有一个信箱，命令收发功能底层都是由信箱实现的
- ZMQ提供了object_t类，用于使用线程信箱发送命令的功能（object_t类还有其他的功能），object_t还有处理命令的功能。
- io线程内还有一个poller用于监听激活mailbox_t的信号，线程收到激活信号后，会去mailbox_t中读命令，然后把命令交由object_t处理

简单来说就是，object_t发命令，poller监听命令到来信号告知线程收命令，交给object_t处理。无论是object_t、还是线程本身、还是poller其实都操作mailbox_t，object_t、poller、mailbox_t都绑定在同一个线程上。

总结 – Command Flow

简单来说，就是对象A将命令发出到目标对象B所在线程绑定的mailbox，然后poller监听收到命令的信号，以通知线程处理命令，线程将命令交给对象B



发出命令:

如果一个类想要使用线程收发命令的功能，那么这个类就必须继承自object_t。源码中可以看到，object_t定义了一个uint32_t变量tid，tid（thread id）表示该object_t对象所在的线程，即应该使用哪个线程的mailbox。关于ctx_t（context），在zmq中被称为上下文，上下文简单来说就是zmq的存活环境，里面存储着的是zmq的全局状态。zmq线程中的mailbox_t指针表（slots）会被zmq维护在ctx_t对象中，表示tid与对应线程绑定的mailbox的对应关系。在zmq中，在context中使用一个容器slots（插槽表）存储线程的mailbox；在新建线程的时候，给线程分配一个线程标识符tid和mailbox，把mailbox放入slots容器的编号为tid的位置，直观来说就是slots[tid]=mailbox。这样，线程A给线程B发命令就只要往slots[B.tid]（B所在的线程绑定的邮箱）写入命令就可以了。

发送一条command时需要经过的路径

```C++
//发送一条command时需要经过的路径
void zmq::object_t::send_command (command_t &cmd_)
{
    _ctx->send_command (cmd_.destination->get_tid (), cmd_);
}

void zmq::ctx_t::send_command (uint32_t tid_, const command_t &command_)
{
    _slots[tid_]->send (command_);
}

void zmq::mailbox_t::send (const command_t &cmd_)
{
    _sync.lock ();      //加锁
    _cpipe.write (cmd_, false); //写消息到管道
    const bool ok = _cpipe.flush ();        // 是消息对读线程可见
    _sync.unlock ();                        // 解锁
    if (!ok)
        _signaler.send ();                  // 当reader不处于alive时则触发其alive读取
}
```

接收命令

io_thread接收命令：每一个io线程中都含有一个poller；在构造函数中，mailbox的句柄被加入poller，则poller可监听mailbox的读事件。所以当有命令进入mailbox的时候，poller会被唤醒，并调用io_thread的in_event()函数。在in_event()函数中，线程调用了mailbox接收信息的recv，然后直接调用destination（目的对象）处理命令的函数去处理命令。

socket_base_t接收命令：**每一个socket_base其实都可以被视为一个线程，但是并没有使用poller**，而是在使用到socket下面几个方法的时候去检查是否有未处理的命令：

```C++
int zmq::socket_base_t::getsockopt(int option_, void *optval_, size_t *optvallen_)
int zmq::socket_base_t::bind(const char *addr_)
int zmq::socket_base_t::connect(const char *addr_)
int zmq::socket_base_t::term_endpoint(const char *addr_)
int zmq::socket_base_t::send(msg_t *msg_, int flags_)
int zmq::socket_base_t::recv(msg_t *msg_, int flags_)
void zmq::socket_base_t::in_event() //这个函数只有在销毁socket的时候会用到
```

socket_base_t使用process_commands方法来检查是否有未处理的命令：

C++ 复制代码

```cpp
int zmq::socket_base_t::process_commands (int timeout_, bool throttle_)
{
    if (timeout_ == 0) {
        //  If we are asked not to wait, check whether we haven't processed
        //  commands recently, so that we can throttle the new commands.

        //  Get the CPU's tick counter. If 0, the counter is not available.
        const uint64_t tsc = zmq::clock_t::rdtsc ();

        //  Optimised version of command processing - it doesn't have to check
        //  for incoming commands each time. It does so only if certain time
        //  elapsed since last command processing. Command delay varies
        //  depending on CPU speed: It's ~1ms on 3GHz CPU, ~2ms on 1.5GHz CPU
        //  etc. The optimisation makes sense only on platforms where getting
        //  a timestamp is a very cheap operation (tens of nanoseconds).
        if (tsc && throttle_) {
            //  Check whether TSC haven't jumped backwards (in case of migration
            //  between CPU cores) and whether certain time have elapsed since
            //  last command processing. If it didn't do nothing.
            if (tsc >= _last_tsc && tsc - _last_tsc <= max_command_delay)
                return 0;
            _last_tsc = tsc;
        }
    }

    //  Check whether there are any commands pending for this thread.
    command_t cmd;
    int rc = _mailbox->recv (&cmd, timeout_);

    //  Process all available commands.
    while (rc == 0) {
        cmd.destination->process_command (cmd);
        rc = _mailbox->recv (&cmd, 0);
    }

    if (errno == EINTR)
        return -1;

    zmq_assert (errno == EAGAIN);

    if (_ctx_terminated) {
        errno = ETERM;
        return -1;
    }

```

```
46        return 0;
47    }
```

可见，最终都是使用mailbox_t的接收命令的功能。
这里有一个值得思考的问题，为什么socket_base_t实例这个线程不使用poller呢？每次使用上面那些方法的时候去检查不是很麻烦吗？
socket_base_t实例之所以被认为是一个特殊的线程，是因为其和io_thread_t一样，都具有收发命令的功能，（关于这点可以看一下io_thread_t的源码，可以发现其主要功能就是收发命令），但是socket_base_t实例是由用户线程创建的，也就是依附于用户线程，而zmq中所有通信都是异步了，所以用户线程是不能被阻塞的，一旦使用poller，线程将被阻塞，也就违背了设计初衷

Message Flow —— ZMQ如何发挥消息中间件的职能

基类：msg_t —— 真正的消息

资料：ZeroMQ源码分析之Message - 夕阳飞鸟
位置：msg_t.hpp, msg_t.cpp
概述：zmq中用于储存信息的类，对于不同大小的消息采用不同的处理，从内存分配的角度优化zmq的效率。

- 对于短信息（vsm：very small），zmq会直接存储在消息的struct内，复制等操作采用直接赋值的方式；
- 对于长消息（lmsg：long message），zmq在初始化的时候会在msg外面另开一块内存，参数中的ffn为销毁信息的时候使用的函数，refcnt是表示该消息被引用次数的计数器。当销毁该消息的时候，需要refcnt为0，即当前没有被引用or复制的时候，才可以被销毁。当复制该消息的时候，只需要将指针指向该内存，并将refcnt计数器加一即可，这就实现了zmq所谓的零拷贝。

结构体定义中的**unused数组的作用**：zmq将每一种类型的消息人为地设置为等大小的，而且使在unused数组后面定义的type和flags变量在每一种struct中的位置是一样的。这样就能做到，无论是什么类型的消息（vsm或者lmsg），只要调用u.base.type就能获取到这个消息的类型了。
在api层（zmq.h中），zmq将一个消息定义为一个长度为64的unsigned char数组（资料中的版本为32字节），大小为64字节，这个大小与定义当中每个结构体的大小恰好相等，因此合理。

核心类：session_t —— 套接字与底层网络engine的纽带

资料：ZMQ源码分析（五） --TCP通讯 - tbyzs
关系：每个session属于一个io线程（一个io线程可以有多个session）；每个session属于一个socket（一个socket也可以同时拥有多个session）；每个session与一个engine连接。（session与engine的关系的一对一的，一个session相当于socket对应的一个endpoint）
简述：每一条tcp连接都需要一对应的session_base（inproc连接不需要，socket_base互相直接连接，通过管道进行消息交换）。
session_base是stream_engine和socket_base之间的纽带，他和socket_base之间有一个pipe_t进行连接，当socket_base需要发出一条数据的时候就把msg写入out管道，之后session_base通过stream_engine发送出去；当stream_engine读取到msg时session_base会把数据写入到session_base

的in管道。session_base_t有一个变量active，它用来标记是否在process_plug中进行connecting操作，start_connecting操作中主要是建立一个tcp_connecter_t 并挂载tcp_connecter_t 作为自己的子对象。之前说过，session_base 和socket_base_t之间有一条传送msg的管道，这个管道是在process_attach的时候建立的，但是如果socket_base进行connect操作，并且制定了option的immediate为非1，则在socket_base_t的connect中直接建立管道。
session_base在attach_pipe 操作中会将自己设置为管道的数据事件监听对象，这样当管道读写状态发生变化时，session_base_t可以通知对应的engine。stream_engine和session_base_t进行msg传递主要通过两个方法，分别是从管道中读数据给engine发送以及收到msg写入管道中。

主要变量：

- _connecter_factories_map：由_connecter_factories[]中的元素构造而成的map，用于维护从协议protocol的名称到创建connecter的函数的映射。
- _start_connecting_map：由_start_connecting_entries[]中的元素构造而成的map，用于维护从协议protocol的名称到启动connecting的函数的映射。

处理命令：

- plug：如果active的话，直接启动start_connecting(false)，启动连接。
- attach：“将引擎engine连接至会话session，如果engine为null的话，告知session失败”；如果与socket间没有pipe的话，就建立一对传输msg的pipes，这之后将engine插到session。

主要接口：

- void start_connecting(bool wait_)：选出负载最小的io线程并在其上建立该session对应的协议的connecter，并在对象树中将connecter视为自己的子节点，然后start该connecter。（这个用法只被用于reconnect时）
- own_t* create_connecter_xxx(io_thread_t *io_thread_, bool wait)：创建并返回一个对应到该session地址的对应类型的connecter；tcp类型的create较为特殊，当存在socks_proxy_address的时候，则创建socks_connecter，否则才创建对应类型的connecer。
- void start_connecting_xxx(io_thread_t *io_thread_)：源文件中共有三种engine：“stream_engine”、“udp_engine”和“norm_engine”；其中的udp引擎和norm引擎用于处理对应协议的通信，而stream引擎在注释中解释为“处理任何具有SOCK_STREAM语义的socket，例如tcp和unix域套接字”。都是创建引擎并接入到session上。

核心类：engine_t —— 真正与网络层通信的组件

于我而言仍然是黑盒。

核心类：socket_base_t —— 底层架构中和应用程序最近的组件

在应用程序中，将会用到各种具有不同功能特性的socket用于进程内、进程间的通信，在后文会介绍两种比较典型且普通的socket种类——router和dealer。

以TCP协议为例：tcp_listener_t 和 tcp_connecter_t

tcp_listener_t

关系：该类在被触发in_event时，将新建一个stream引擎和一个session，并将engine连接到创建的session上。

变量：

- fd_t _s：底层套接字。（其实就是一个文件描述符）
- tcp_address_t _address：监听的tcp地址。
- handle_t _handle：监听套接字对应的句柄。
- zmq::socket_base_t *_socket：此监听组件所属的套接字。
- string _endpoint：需要绑定到的端点的string表示。

处理命令：

- void process_plug()：将该listener插入poller以开始监听。
- void process_term()：关闭该listener。

接口：

- int set_adress(const char *addr_)：设置监听的地址。
- int get_adress(std::string &addr_)：得到被bind的地址以使用通配符。
- void in_event()：处理I/O事件；当有新的连接的时候，in_event被调用，并新建一个stream engine和一个session，并将stream engine连接到创建的session上

tcp_connecter_t

tcp_connecter和tcp_listerner相反，他是由session_base_t创建的并负责管理和销毁的，tcp_connecter_t也不会一直存在，**当连接成功之后就会被销毁**；

当tcp_connecter连接失败时会设定一个定时器以便重新连接，这就使zmq支持在lbind之前进行connect依旧是可以成功的。

以建立一个tcp协议的连接为例

Bind

- 简述：socket新建listener → listener监听到连接connect → listener为此连接建立session和engine。
- 过程：当一个socket_base需要bind到一个endpoint的时候，首先，在socket_base::bind(....)中关于tcp的部分，新建一个tcp_listener，然后将tcp_listener的监听句柄插入poller开始监听，当有新的连接的时候，poller触发tcp_listener::in_event事件，tcp_listener会新建一个session_base和一个stream_engine；并向session_base发送attach命令，将新建的stream_engine插到session上，与此同时，建立socket_base与session_base间交换message的pipe对。（connecter在这个过程中好像始终没派上什么用场）
- 备注：当socket_base要bind到某个endpoint的时候，会建立一个tcp_listener并将句柄插入poller以监听其他socket_base的connect请求；每收到一个connect请求，都会触发一次tcp_listener::in_event并新建一个session_base和一个stream_engine。也就是说，每个tcp_listener对应着一个endpoint，而每个session_base和stream_engine都对应着一个连接着该endpoint的对端peer。

Connect

- 简述：**socket新建session** → session新建connecter → connecter新建engine并插入session → connecter被销毁。
- 过程：当一个socket_base要connect的时候，在socket_base::connect(....)中的line938**新建了一个session**，并在参数immediate非1的时候（不太清楚什么情况下会非1）直接建立socket_base与session_base之间用于交换message的pipe对。然后在add_endpoint方法中向刚建立的session_base发送plug命令，session_base在此进入process_plug方法，即开始secssion_base::start_connecting。在此方法中会launch一个tcp_connecter并向其发送一个plug command，tcp_connecter在成功打开底层socket之后会直接进入out_event方法，在此方法中新建对应的stream_engine并将engine插入到session上；在此之后，tcp_connecter自毁。
- 备注：**每一个connect都对应着一对session_base和stream_engine**。

How Messages Flow

当socket_base需要发出一条数据的时候就把message写入out管道，之后session_base通过stream_engine发送出去；当stream_engine读取到message时session_base会把数据写入到session_base的in管道。

高水位标记high-water mark，HWM

```
//  High watermark for the outbound pipe.
    int _hwm;
    //  Low watermark for the inbound pipe.
    int _lwm;
```

ZMQ使用高水位标记（HWM）的概念来定义其内部管道的容量。每个连接都有其发送或接受数据的管道和对应的HWM。对于管道和HWM，不同的socket有不同的行为：

- PUB，PUSH只有发送管道
- SUB，PULL，REP，REQ只有接收管道
- DEALER, ROUTER, PAIR两者都有

缺省的值
const int default_hwm = 1000;

可以通过setsocketopt函数来设置HWM的值。
当数据填满管道，达到HWM时，不同的socket也有不同的表现：

- 在 PUB 和 ROUTER 类型套接字上，当在传出管道上达到 SNDHWM 时，将会丢弃消息。DEALER 和 PUSH 类型套接字将会堵塞。
- 在传入管道上，到达 RCVHWM 时，SUB 类型套接字将会丢弃消息，而 PULL 和 DEALER 套接字将拒绝新消息并强制消息在上游等待。

On PUB and ROUTER sockets, when the SNDHWM is reached on outgoing pipes, messages are dropped. DEALER and PUSH sockets will block when their outgoing pipes are full.
On incoming pipes, SUB sockets will drop received messages when they reach their RCVHWM. PULL and DEALER sockets will refuse new messages and force messages to wait upstream due to TCP backpressure.

设置范例

C++ 复制代码

```
 void *socket = zmq_socket (context, ZMQ_PUSH);
zmq_connect (requester, "tcp://localhost:5555");
int queue_length = 5000;
zmq_setsockopt(socket, ZMQ_SNDHWM, &queue_length,sizeof(queue_length));
zmq_connect (socket, "tcp://127.0.0.1:5555");
```

水位控制在于pipe_t

必须为PUB套接字设置阈值，具体数字可以通过最大订阅者数、可供队列使用的最大内存区域、以及消息的平均大小来衡量。举例来说，你预计会有5000个订阅者，有1G的内存可供使用，消息大小在200个字节左右，那么，一个合理的阈值是1,000,000,000 / 200 / 5,000 = 1,000。

水位主要是控制写入端

C++ 复制代码

```cpp
bool zmq::pipe_t::read (msg_t *msg_)
{
  ...........
    for (bool payload_read = false; !payload_read;) {
        if (!_in_pipe->read (msg_)) {       // 读取主线程发送过来的消息
            _in_active = false;
            return false;
        }
    ............
    }

    if (!(msg_->flags () & msg_t::more) && !msg_->is_routing_id ())
        _msgs_read++;
  // _lwm默认是500，_hwm默认是1000 ，对于sub模式
    if (_lwm > 0 && _msgs_read % _lwm == 0)     // 每次读取完一次最低水位则通知对方
        send_activate_write (_peer, _msgs_read);    // 告诉对方可以继续写入以及读取了多少的数据

    return true;
}

bool zmq::pipe_t::write (msg_t *msg_)
{
    if (unlikely (!check_write ()))     // 检测是否可以写，水位满时则不能继续写入
        return false;   // 如果满了则退出

    const bool more = (msg_->flags () & msg_t::more) != 0;
    const bool is_routing_id = msg_->is_routing_id ();
    _out_pipe->write (*msg_, more);
    if (!more && !is_routing_id)
        _msgs_written++;

    return true;
}

check_write是实际是检测hwm，_peers_msgs_read是对端send_activate_write时附带的_msgs_read
bool zmq::pipe_t::check_hwm () const
{
    const bool full = // 本质就是写入的消息数量和已经读取的消息数量做对比
      _hwm > 0 && _msgs_written - _peers_msgs_read >= uint64_t (_hwm);
    return !full;
}
```

多帧消息

ZMQ消息可以包含多个帧，这在实际应用中非常常见.
多帧消息的每一帧都是一个zmq_msg结构，也就是说，当你在收发含有五个帧的消息时，你需要处理五个zmq_msg结构。你可以将这些帧放入一个数据结构中，或者直接一个个地处理它们。
下面的代码演示如何发送多帧消息：

C++ 复制代码

```
zmq_msg_send (&message, socket, ZMQ_SNDMORE);
...
zmq_msg_send (&message, socket, ZMQ_SNDMORE);
...
zmq_msg_send (&message, socket, 0);   // 最后一帧以0结束
```

然后我们看看如何接收并处理这些消息，这段代码对单帧消息和多帧消息都适用：

C++ 复制代码

```
while (1)
{
    // 处理一帧消息
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_msg_recv (&message, socket, 0);
    zmq_msg_close (&message);

    // 已到达最后一帧
    int64_t more;
    zmq_getsockopt (socket, ZMQ_RCVMORE, &more, sizeof (more));
    if (!more)
        break;
}
```

关于多帧消息，你需要了解的还有：

- 在发送多帧消息时，只有当最后一帧提交发送了，整个消息才会被发送；
- 多帧消息是整体传输的，不会只收到一部分；
- 多帧消息的每一帧都是一个zmq_msg结构；
- 无论你是否检查套接字的ZMQ_RCVMORE选项，你都会收到所有的消息；
- 发送时，**ZMQ会将开始的消息帧缓存在内存中，直到收到最后一帧才会发送**；
- 我们无法在发送了一部分消息后取消发送，只能关闭该套接字。

重连机制

ZMQ_CONNECT_TIMEOUT

int zmq_setsockopt (void *socket, int option_name, const void *option_value, size_t option_len);

ZMQ_CONNECT_TIMEOUT: Set connect() timeout

Sets how long to wait before timing-out a connect() system call. The connect() system call normally takes a long time before it returns a time out error. Setting this option allows the library to time out the call at an earlier interval.

Option value type	int
Option value unit	milliseconds
Default value	0 (disabled)
Applicable socket types	all, when using TCP transports.

ZMQ_RECONNECT_IVL: Set reconnection interval

The *ZMQ_RECONNECT_IVL* option shall set the initial reconnection interval for the specified *socket*. The reconnection interval is the period ØMQ shall wait between attempts to reconnect disconnected peers when using connection-oriented transports. The value -1 means no reconnection.

The reconnection interval may be randomized by ØMQ to prevent reconnection storms in topologies with a large number of peers per socket.

Option value type	int
Option value unit	milliseconds
Default value	100
Applicable socket types	all, only for connection-oriented transports

ZMQ_RECONNECT_IVL_MAX: Set maximum reconnection interval

The *ZMQ_RECONNECT_IVL_MAX* option shall set the maximum reconnection interval for the specified *socket*. This is the maximum period ØMQ shall wait between attempts to reconnect. On each reconnect attempt, the previous interval shall be doubled untill ZMQ_RECONNECT_IVL_MAX is reached. This allows for exponential backoff strategy. Default value means no exponential backoff is performed and reconnect interval calculations are only based on ZMQ_RECONNECT_IVL.

Values less than ZMQ_RECONNECT_IVL will be ignored.

Option value type	int
Option value unit	milliseconds
Default value	0 (only use ZMQ_RECONNECT_IVL)
Applicable socket types	all, only for connection-oriented transports

由命令plug 触发启动连接

```
#0  zmq::session_base_t::start_connecting (this=0x619510, wait_=false) at
src/session_base.cpp:607
#1  0x00007ffff7b56772 in zmq::session_base_t::process_plug (this=0x619510) at
src/session_base.cpp:331
#2  0x00007ffff7b36e6f in zmq::object_t::process_command (this=0x619510, cmd_=...) at
src/object.cpp:87
#3  0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#4  0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#5  0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at
src/poller_base.cpp:139
#6  0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7  0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#8  0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
```

真正连接的函数

```
zmq::tcp_connecter_t::open ()
#0  zmq::tcp_connecter_t::open (this=0x7ffff00008c0) at src/tcp_connecter.cpp:177
#1  0x00007ffff7b75e1a in zmq::tcp_connecter_t::start_connecting (this=0x7ffff00008c0)
    at src/tcp_connecter.cpp:131
#2  0x00007ffff7b6bff8 in zmq::stream_connecter_base_t::process_plug (this=0x7ffff00008c0)
    at src/stream_connecter_base.cpp:80
#3  0x00007ffff7b36e6f in zmq::object_t::process_command (this=0x7ffff00008c0, cmd_=...) at
src/object.cpp:87
#4  0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#5  0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#6  0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at
src/poller_base.cpp:139
#7  0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#8  0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
```

#9 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

连接不成功时则需要启动重连定时器
#0 zmq::stream_connecter_base_t::add_reconnect_timer (this=0x7ffff00008c0) at src/stream_connecter_base.cpp:101
#1 0x00007ffff7b75d12 in zmq::tcp_connecter_t::out_event (this=0x7ffff00008c0) at src/tcp_connecter.cpp:110
#2 0x00007ffff7b6c442 in zmq::stream_connecter_base_t::in_event (this=0x7ffff00008c0)
    at src/stream_connecter_base.cpp:166
#3 0x00007ffff7b234b2 in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:198
#4 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#5 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#6 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#7 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

重连时
#0 zmq::tcp_connecter_t::start_connecting (this=0x7ffff00008c0) at src/tcp_connecter.cpp:129
#1 0x00007ffff7b6c686 in zmq::stream_connecter_base_t::timer_event (this=0x7ffff00008c0, id_=1)
    at src/stream_connecter_base.cpp:193
#2 0x00007ffff7b75de5 in zmq::tcp_connecter_t::timer_event (this=0x7ffff00008c0, id_=1)
    at src/tcp_connecter.cpp:125
#3 0x00007ffff7b441ab in zmq::poller_base_t::execute_timers (this=0x618670) at src/poller_base.cpp:103
#4 0x00007ffff7b2330e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:173
#5 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x618670) at src/poller_base.cpp:139
#6 0x00007ffff7b773b3 in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#8 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

公平队列问题

比如sub-pub模式，sub可以订阅多路的pub，那此时对应多个pipe，但每次只能从一个pipe里面读取数据

int zmq::xsub_t::xrecv (msg_t *msg_)

不同的连接独有独立的pipe

#0 zmq::xsub_t::xattach_pipe (this=0x619890, pipe_=0x622c40, subscribe_to_all_=false, locally_initiated_=true)
    at src/xsub.cpp:65
#1 0x00007ffff7b5d530 in zmq::socket_base_t::attach_pipe (this=0x619890, pipe_=0x622c40,
    subscribe_to_all_=false, locally_initiated_=true) at src/socket_base.cpp:388
#2 0x00007ffff7b5fee4 in zmq::socket_base_t::connect (this=0x619890,
    endpoint_uri_=0x40196a "tcp://localhost:5556") at src/socket_base.cpp:973
#3 0x00007ffff7b8b93f in zmq_connect (s_=0x619890, addr_=0x40196a "tcp://localhost:5556") at src/zmq.cpp:313
#4 0x00000000004016cb in main (argc=1, argv=0x7fffffffe528)
    at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/wuclient.c:13

读取的时候

io线程有写入数据时会触发 activated

#0 zmq::fq_t::activated (this=0x619ea0, pipe_=0x622c40) at src/fq.cpp:74
#1 0x00007ffff7b8a984 in zmq::xsub_t::xread_activated (this=0x619890, pipe_=0x622c40) at src/xsub.cpp:76
#2 0x00007ffff7b61ff4 in zmq::socket_base_t::read_activated (this=0x619890, pipe_=0x622c40)
    at src/socket_base.cpp:1583
#3 0x00007ffff7b3fabc in zmq::pipe_t::process_activate_read (this=0x622c40) at src/pipe.cpp:276
#4 0x00007ffff7b36e10 in zmq::object_t::process_command (this=0x622c40, cmd_=...) at src/object.cpp:75
#5 0x00007ffff7b61643 in zmq::socket_base_t::process_commands (this=0x619890, timeout_=-1, throttle_=false)
    at src/socket_base.cpp:1369
#6 0x00007ffff7b610a0 in zmq::socket_base_t::recv (this=0x619890, msg_=0x7fffffffe240, flags_=0)
    at src/socket_base.cpp:1250
#7 0x00007ffff7b8c008 in s_recvmsg (s_=0x619890, msg_=0x7fffffffe240, flags_=0) at src/zmq.cpp:454
#8 0x00007ffff7b8c164 in zmq_recv (s_=0x619890, buf_=0x7fffffffe2c0, len_=255, flags_=0) at src/zmq.cpp:479
#9 0x00000000004010f7 in s_recv (socket=0x619890)
    at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/zhelpers.h:54
#10 0x0000000000401769 in main (argc=1, argv=0x7fffffffe528)
    at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/wuclient.c:26

公平读取

#0 zmq::fq_t::recvpipe (this=0x619ea0, msg_=0x7fffffffe240, pipe_=0x0) at src/fq.cpp:86

#1 0x00007ffff7b24604 in zmq::fq_t::recv (this=0x619ea0, msg_=0x7fffffffe240) at src/fq.cpp:80

#2 0x00007ffff7b8adf6 in zmq::xsub_t::xrecv (this=0x619890, msg_=0x7fffffffe240) at src/xsub.cpp:164

#3 0x00007ffff7b610dc in zmq::socket_base_t::recv (this=0x619890, msg_=0x7fffffffe240, flags_=0)

    at src/socket_base.cpp:1253

#4 0x00007ffff7b8c008 in s_recvmsg (s_=0x619890, msg_=0x7fffffffe240, flags_=0) at src/zmq.cpp:454

#5 0x00007ffff7b8c164 in zmq_recv (s_=0x619890, buf_=0x7fffffffe2c0, len_=255, flags_=0) at src/zmq.cpp:479

#6 0x0000000004010f7 in s_recv (socket=0x619890)

    at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/zhelpers.h:54

#7 0x0000000000401769 in main (argc=1, argv=0x7fffffffe528)

    at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/wuclient.c:26

C++ 复制代码

```cpp
int zmq::fq_t::recvpipe (msg_t *msg_, pipe_t **pipe_)
{
    //  Deallocate old content of the message.
    int rc = msg_->close ();
    errno_assert (rc == 0);

    //  Round-robin over the pipes to get the next message.
    while (_active > 0) {
        //  Try to fetch new message. If we've already read part of the message
        //  subsequent part should be immediately available.
        bool fetched = _pipes[_current]->read (msg_);

        //  Note that when message is not fetched, current pipe is deactivated
        //  and replaced by another active pipe. Thus we don't have to increase
        //  the 'current' pointer.
        if (fetched) {
            if (pipe_)
                *pipe_ = _pipes[_current];
            _more = (msg_->flags () & msg_t::more) != 0;
            if (!_more) {
                _last_in = _pipes[_current];
                _current = (_current + 1) % _active;  // 按着顺序读取消息 所谓公平
            }
            return 0;
        }

        //  Check the atomicity of the message.
        //  If we've already received the first part of the message
        //  we should get the remaining parts without blocking.
        zmq_assert (!_more);

        _active--;
        _pipes.swap (_current, _active);
        if (_current == _active)
            _current = 0;
    }

    //  No message is available. Initialise the output parameter
    //  to be a 0-byte message.
    rc = msg_->init ();
    errno_assert (rc == 0);
    errno = EAGAIN;
    return -1;
}
```

stream_engine数据发送触发

触发数据的发送

#0 zmq::io_object_t::set_pollout (this=0x7ffff0001720, handle_=0x7ffff0001680) at src/io_object.cpp:85
#1 0x00007ffff7b6ec42 in zmq::stream_engine_t::restart_output (this=0x7ffff0001720)
    at src/stream_engine.cpp:454
#2 0x00007ffff7b56599 in zmq::session_base_t::read_activated (this=0x61a580, pipe_=0x622d50)
    at src/session_base.cpp:297
#3 0x00007ffff7b3fabc in zmq::pipe_t::process_activate_read (this=0x622d50) at src/pipe.cpp:276
#4 0x00007ffff7b36e10 in zmq::object_t::process_command (this=0x622d50, cmd_=...) at src/object.cpp:75
#5 0x00007ffff7b25a09 in zmq::io_thread_t::in_event (this=0x6190c0) at src/io_thread.cpp:91
#6 0x00007ffff7b2356c in zmq::epoll_t::loop (this=0x619670) at src/epoll.cpp:206
#7 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x619670) at src/poller_base.cpp:139
#8 0x00007ffff7b773b3 in thread_routine (arg_=0x6196c8) at src/thread.cpp:225
#9 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#10 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

没有数据可以发送时

#0 zmq::io_object_t::reset_pollout (this=0x7ffff0001720, handle_=0x7ffff0001680) at src/io_object.cpp:90
#1 0x00007ffff7b6ebdc in zmq::stream_engine_t::out_event (this=0x7ffff0001720) at src/stream_engine.cpp:445
#2 0x00007ffff7b23511 in zmq::epoll_t::loop (this=0x619670) at src/epoll.cpp:202
#3 0x00007ffff7b44365 in zmq::worker_poller_base_t::worker_routine (arg_=0x619670) at src/poller_base.cpp:139
#4 0x00007ffff7b773b3 in thread_routine (arg_=0x6196c8) at src/thread.cpp:225
#5 0x00007ffff73096ba in start_thread (arg=0x7ffff625f700) at pthread_create.c:333
#6 0x00007ffff782e4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

性能测试

官方文档：http://wiki.zeromq.org/whitepapers:measuring-performance

中文翻译：https://blog.csdn.net/bluegreen315/article/details/11873735