2-1-预习资料-ZeroMQ源码分析

- send
- recv
- read
- connect
- zmq::stream_engine_t::stream_engine_t
- zmq::epoll_t::add_fd
- zmq::mailbox_t::send
- zmq::session_base_t::create
- zmq::object_t::send_plug
- zmq::req_session_t::req_session_t
- zmq::req_t::xrecv
- zmq::req_session_t::push_msg
- zmq::pipe_t::read
- zmq::stream_engine_t::out_event
- zmq::stream_engine_t::in_event
- zmq::pipepair
- zmq::pipe_t::pipe_t

关键函数

- zmq_ctx_new
- zmq_socket
- zmq_connect
- zmq_send
- zmq_recv
- zmq::object_t::process_command

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

网页版本：
https://www.yuque.com/docs/share/09e409a7-f004-4bdb-840e-d4cac47a7f4f?#（密码：dzwu）《2-1-预习资料-ZeroMQ源码分析》

ZeroMQ是什么

对传统的网络接口进行再次封装，提供了一个跨语言的库，支持任意语言使用统一的接口，基于这些封装，在分布式环境下，提供了一些常见的消息模型，比如请求-回应（request-reply）、发布订阅（Subscribe-Publish）、推拉（Push-pull）以及基于这些基础的消息模型衍生的更加高级的模型。

ZMQ是网络通信中新的一层，**介于应用层和传输层之间**（按照TCP/IP划分），其是一个可伸缩层，可并行运行，分散在分布式系统间。
ZMQ不是单独的服务，而是一个嵌入式库，它封装了网络通信、消息队列、线程调度等功能，向上层提供简洁的API，应用程序通过加载库文件，调用API函数来实现高性能网络通信。

ZeroMQ能够解决什么

- 调用的socket接口较多；
- TCP是一对一的连接；
- 编程需要关注很多socket细节问题；
- 不支持跨平台编程；
- 需要自行处理分包、组包问题；
- 流式传输时需处理粘包、半包问题；
- 需自行处理网络异常，比如连接异常中断、重连等； 内部去解决
- 服务端和客户端启动有先后；
- 自行处理IO模型；
- 自行实现消息的缓存；
- 自行实现对消息的加密；

ZMQ的特性

1、ZMQ有一些后台IO线程，封装了异步处理
2、ZMQ的连接不上一对一的，是N对N的连接
3、ZMQ封装了message_t的结构，解决了粘包和半包等等问题；
4、ZMQ的messaget_t接口是一个多帧结构，支持任意大小的消息，甚至是2G大小；
5、ZMQ支持零拷贝，使用ZMQ替代原生的套接字后，应用程序的性能并没有下降多少；甚至是提高了应用程序的性能；

6、ZMQ具有消息缓存的功能，且这些缓存是异步的，如果对端断开了连接，依然可以接收这些消息，缓存队列提供HWM的功能（高水位的特性）；
7、ZMQ并不要求对端一定要先启动，启动顺序是无关的；
8、ZMQ能够处理中断重连，并且我们可以调整中断重连的策略；
9、ZMQ是跨平台的；
10、ZMQ提供了任意节点间的通信，这些节点可以是进程、线程和机器；
11、ZMQ支持不同语言开发的程序间通信；
12、在高级消息模式下，具有公平队列的能力；
13、ZMQ可以从公平连接到公平流量；
14、ZMQ的后台异步IO线程可以绑定CPU，提高应用程序的性能；
15、ZMQ可以让网络通信变得很简单（有些高级消息模式却又很复杂）。
那ZMQ是如何一一做到这些还不失去其性能的呢?

不存在zmq_listen()、zmq_accept()方法，当一个套接字被绑定到一个端点的时候，它自定地开始监听、接受连接
网络连接本身是在后台发生的，而如果网络连接断开（例如，对等节点消失后又回来），ZeroMQ会自动地重新连接
应用程序不能与这些连接直接交流，它们是被封装在套接字之下的

名词解析

Mail Box
Main Thread
I/O Thread
Session
Engine
Outbound Pipe
Inbound Pipe
register
callback
Reactor
queue
slot
pipe

关键类

ctx_t

ctx_t ZeroMQ对象的封装，继承了thread_ctx_t

context 干嘛用的?

context 是用于管理全局状态的，例如sockets, io_thread, repear等。下面是zmq内部结构白皮书的解释：

> To prevent this problem libzmq has no global variables. Instead, user of the library is responsible for creating the global state explicitly. Object containing the global state is called 'context'. While from the users perspective context looks more or less like a pool of I/O threads to be used with your sockets, from libzmq's perspective it's just an object to store any global state that we happen to need. For example, the list of available inproc endpoints is stored in the context. The list of sockets that have been closed but still linger in the memory because of unsent messages is held by context. Etc.
> Context is implemented as class ctx_t.

ctx_t内主要有这几类线程：

- Application thread，这是应用主线程。
- 若干 I/O threads，可以没有I/O线程，也可以有多个。
- repear thread，用于关闭和清理 sockets。
  使用回收线程 repear 是为了：

> You can close TCP socket, the call returns immediately, even if there's pending outbound data to be sent to the peer later on.

- term thread，用于销毁。

一个context内可以有多个sockets:

```C++
//  Sockets belonging to this context. We need the list so that
//  we can notify the sockets when zmq_ctx_term() is called.
//  The sockets will return ETERM then.
typedef array_t <socket_base_t> sockets_t;
sockets_t sockets;
```

每个线程都有自己的邮箱，所以用一个数组来管理这些邮箱：

```C++
//  Array of pointers to mailboxes for both application and I/O threads.
uint32_t slot_count;
i_mailbox **slots;
```

总而言之，ctx_t就是负责管理 sockets 和 threads的。

socket_base_t

```C++
void *zmq_socket (void *ctx_, int type_)
{
    ...
    zmq::ctx_t *ctx = (zmq::ctx_t *) ctx_;
    zmq::socket_base_t *s = ctx->create_socket (type_);
    return (void *) s;
}
```

可见，实际是调用了ctx->create_socket的方法，跳到ctx.cpp去看看。
逻辑还是比较简单的，大致分成两步：

1. 创建各路工作线程。I/O threads, repear thread, term thread(应该是自身线程,无须创建），并且注册每个线程的邮箱（上文提到，用一个vector记录每个object的mailbox）。
2. 创建socket，并注册其邮箱。
   `socket_base_t *s = socket_base_t::create (type_, this, slot, sid);`

thread_t

真正对线程的封装
`thread_t`是对pthead的一层封装，封装成C++类。
`thread.hpp`

```C++
#include <pthread.h>

namespace zmq {
    typedef void (thread_fn) (void*);  // 定义个类型‘线程函数’, function

    class thread_t{
    public:
        inline thread_t(): // 没搞懂为什么要显示的使用inline
            tfn(NULL),
            args(NULL) {
        }
        void start(thread_fn *tfn, void *args); //启动线程
        void stop();        // 终止线程

        thread_fn *tfn;     // 函数指针， void (*)(void *)
        void *args;         // 函数参数

    private:
        pthread_t descriptor; // 线程描述符
    };
}
```

thread.cpp

C++ 复制代码

```cpp
#include "thread.h"
extern "C" {
static void *thread_routine (void *arg_) // pthread.h是 C 接口，这里拓展了C特性。
{ // 由于是用pthread，所以要用static void *xxx(void *args)
#if !defined ZMQ_HAVE_OPENVMS && !defined ZMQ_HAVE_ANDROID
    //  Following code will guarantee more predictable latencies as it'll
    //  disallow any signal handling in the I/O thread.
    sigset_t signal_set;
    int rc = sigfillset (&signal_set);
    errno_assert (rc == 0);
    rc = pthread_sigmask (SIG_BLOCK, &signal_set, NULL);
    posix_assert (rc);
#endif
    zmq::thread_t *self = (zmq::thread_t *) arg_;
    self->applySchedulingParameters ();
    self->applyThreadName ();
    self->_tfn (self->_arg);  // 函数调用在此!
    return NULL;
}
}

void zmq::thread_t::start (thread_fn *tfn_, void *arg_, const char *name_)
{
    _tfn = tfn_;
    _arg = arg_;
    if (name_)
        strncpy (_name, name_, sizeof (_name) - 1);
    int rc = pthread_create (&_descriptor, NULL, thread_routine, this);
    posix_assert (rc);
    _started = true;
}
void zmq::thread_t::stop ()
{
    if (_started) {
        int rc = pthread_join (_descriptor, NULL);
        posix_assert (rc);
    }
}
```

那么怎么用呢？举个例子，我们现在想启动一个线程运行`epoll_t`中的`loop()`循环。很有意思，`loop()`是成员函数，为了把他暴露出来传递给thread_t，这里用了一个`worker_routine()`。

`epoll.cpp`

C++ 复制代码

```
void zmq::io_thread_t::start ()
{
    char name[16] = "";
    snprintf (name, sizeof (name), "IO/%u",
              get_tid () - zmq::ctx_t::reaper_tid - 1);
    //  Start the underlying I/O thread.
    _poller->start (name);    // 启动IO线程
}

void zmq::worker_poller_base_t::start (const char *name_)
{
    zmq_assert (get_load () > 0);
    _ctx.start_thread (_worker, worker_routine, this, name_);
}

void zmq::worker_poller_base_t::worker_routine (void *arg_)
{
    (static_cast<worker_poller_base_t *> (arg_))->loop ();
}

void zmq::epoll_t::loop ()
{
    epoll_event ev_buf[max_io_events];

    while (true) {
  .........
        //  Wait for events.
        int n = epoll_wait (_epoll_fd, &ev_buf[0], max_io_events,
                            timeout ? timeout : -1);
    ............
    }
```

msg_t

因为tcp是一种字节流类型的协议，没有边界，所以把该消息边界的制定留给了应用层。

通常有两种方式实现：

1. 在传统的数据中添加分隔符
2. 在每条消息中添加size字段。

而zeromq采用第2种方案。

zmq_msg_t基本的数据结构

C++ 复制代码

```cpp
class msg_t
    {
    public:

        //  Mesage flags. // 表示一些flags
        enum
      {
          more = 1,    //  Followed by more parts
          command = 2, //  Command frame (see ZMTP spec)
          //  Command types, use only bits 2-5 and compare with ==, not bitwise,
          //  a command can never be of more that one type at the same time
          ping = 4,
          pong = 8,
          subscribe = 12,
          cancel = 16,
          credential = 32,
          routing_id = 64,
          shared = 128
      };

        bool check ();
        int init ();
        int init_size (size_t size_);
        int init_data (void *data_, size_t size_, msg_free_fn *ffn_,
            void *hint_);
        int init_delimiter ();
        int close ();
        int move (msg_t &src_);
        int copy (msg_t &src_);
        void *data ();
        size_t size ();
        unsigned char flags ();
        void set_flags (unsigned char flags_);
        void reset_flags (unsigned char flags_);
        bool is_identity () const;
        bool is_delimiter ();
        bool is_vsm ();

        //  After calling this function you can copy the message in POD-style
        //  refs_ times. No need to call copy.
        void add_refs (int refs_);

        //  Removes references previously added by add_refs. If the number of
        //  references drops to 0, the message is closed and false is returned.
        bool rm_refs (int refs_);

```

```
47      private:
48
49          //  Size in bytes of the largest message that is still copied around
50          //  rather than being reference-counted.
51          enum {max_vsm_size = 29}; // 最大的very small message消息的字节大小，小消
    息拷贝，非引用计数
52
53          //  Shared message buffer. Message data are either allocated in one
54          //  continuous block along with this structure - thus avoiding one
55          //  malloc/free pair or they are stored in used-supplied memory.
56          //  In the latter case, ffn member stores pointer to the function to
    be
57          //  used to deallocate the data. If the buffer is actually shared
    (there
58          //  are at least 2 references to it) refcount member contains number
    of
59          //  references.
60          // 共享的消息buffer。
61          struct content_t
62          {
63              void *data; // 指向真正的消息数据
64              size_t size; // 消息数据的字节大小
65              msg_free_fn *ffn; // 指向释放函数
66              void *hint;      // 传递到回到功能的一个提示值
67              zmq::atomic_counter_t refcnt;// 消息的引用计数
68          };
69
70          //  Different message types. // 不同的消息类型
71          enum type_t
72          {
73              type_min = 101,
74              type_vsm = 101,
75              type_lmsg = 102,
76              type_delimiter = 103,
77              type_max = 103
78          };
79
80          //  Note that fields shared between different message types are not
81          //  moved to tha parent class (msg_t). This way we ger tighter
    packing
82          //  of the data. Shared fields can be accessed via 'base' member of
83          //  the union.
84          union {
85              // base
86              struct {
87                  unsigned char unused [max_vsm_size + 1];
88                  unsigned char type;
89                  unsigned char flags;
90              } base;
91              // 针对very small message的小消息做的一些优化，直
```

```
92              接在stack上分配内存了，可以看后面的消息函数的具体操作
93              struct {
94                  unsigned char data [max_vsm_size]; // 小消息的内存区域
95                  unsigned char size; // 小消息的大小
96                  unsigned char type;
97                  unsigned char flags;
98              } vsm;
99             // 针对大消息
100             struct {
101                 content_t *content; // 指向content_t的结构
102                 unsigned char unused [max_vsm_size + 1 - sizeof
    (content_t*)];
103                 unsigned char type;
104                 unsigned char flags;
105             } lmsg;
106            // 定界符
107             struct {
108                 unsigned char unused [max_vsm_size + 1];
109                 unsigned char type;
110                 unsigned char flags;
111             } delimiter;
112         } u;
113     };
```

delimiter类型的消息，类似于终结者的意思，主要在收发的管道中使用。

因为zeromq会将消息先发送到管道中，
然后poller运行在另一个线程，将管道中的数据读出来，发往socket的缓冲区，
所以，可以发送一个delimiter类型的消息去终结管道，并销毁它。

短消息使用copy比malloc高效
长消息使用copy高效
vsm保存短消息
lsm保存长消息

使用zmq消息的基本原则

- 创建并创建zmq_msg_t对象，使用zmq_msg_t来表示消息，而不是使用普通的数据块（char*）来交互数据
- 要读取消息，可使用zmq_msg_int()创建一个空的消息，然后传递给zmq_msg_recv()
- 要写入消息，可以使用zmq_msg_init_size()来创建消息，并分配某个大小的数据块数据，使用memcpy()将数据块的数据拷贝给zmq_msg_t，然后将zmq_msg_t传递给zmq_msg_send()进行发送
- 要释放消息，则调用zmq_msg_close()，这会删除一个引用，当消息引用为0时，ØMQ会最终自动帮你销毁该消息
- 要访问消息内容，可以使用zmq_msg_data()

- 要知道消息包含多少数据，可以使用zmq_msg_size()
- 一般不建议使用zmq_msg_move()、zmq_msg_copy()、zmq_msg_init_data()，除非你的目标很明确就是要用这些函数
- zmq_msg_send()传递一个消息时候，会把该消息清除（把它的大小设置为0），因此消息发送之后需要关闭(zmq_msg_close())并且不再使用。如果你想多次发送相同的数据，可以创建两个zmq_msg_t消息对象发送，或者在调用zmq_msg_init()之前使用zmq_msg_copy()拷贝两份一样的数据并同时发送

pipe_t

内部封装的时候，有两个方向。
upipe_t *_in_pipe;
upipe_t *_out_pipe;

thread_ctx_t

thread_ctx_t 线程上下文的封装

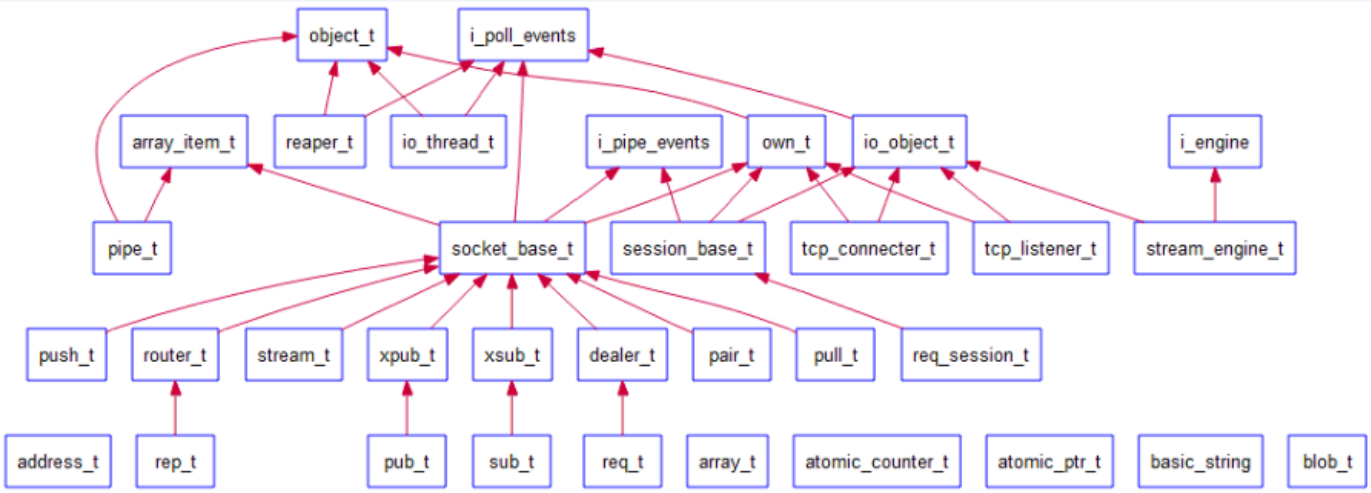
command_t 和pipe_t

zeromq的通信方式有两种：command_t和pipe_t方式。
1 command_t将其它io对象的io操作交给io线程处理，主要有：
plug命令：将一个io对象注册到一个io线程
own命令：将两个对象绑定，
attach命令：绑定engine和session
bind命令：session和socket绑定
2 pipe_t方式实现了一个lock free 的管道，用于线程间的大量数据的传输，如io线程接收完数据后将数据传输给应用线程，pair模式也是直接采用pipe_t实现通信。

poller_base_t
worker_poller_base_t
epoll_t 我们用linux只需要关注epoll

reaper_t 绑定一个线程，并且最终调用了epoll_wait
io_thread_t 绑定一个线程，并且最终调用了epoll_wait
mailbox_t 每个io_thread_t 都对应了一个mailbox_t ，用作数据通信?
ypipe_t 用于mailbox的发送，本质上是使用了yqueue_t

zmq类层次



①、object_t，主要用于发送命令和处理命令，所有继承object_t的子类都具备该类的功能

②、io_thread_t，内含一个poller，可监听句柄的读、写、异常状态，继承自object_t，具有接收命令、处理命令、发送命令的功能

③、io_object_t，可以获取一个io_thread_t的poller，从而具备poller功能，所有继承自该类的子类都具有pollere功能，可监听句柄的读、写、异常状态

④、reaper_t，zmq的回收线程

⑤、own_t，zmq的对象树结点，或者说多叉树的结点，其主要用于对象的销毁，可以想到，对象的销毁就是这棵树的销毁过程，必须要使用深度优先的算法来销毁。关于zmq对象树在Internal Architecture of libzmq有详细讲解

⑥、tcp_connector_t，zmq_socket的连接器，使用她来建立tcp连接

⑦、tcp_listener_t，zmq_socket的监听器

⑧、stream_engine，负责处理io事件中的一种----网络事件，把网络字节流转换成zeromq的msg_t消息传递给session_base_t。另外一些和版本兼容相关的杂务也stream_engine处理的。stream_engine_t处理完杂务，到session_base_t就只看见msg_t了。

⑨、session_base_t，管理zmq_socket的连接和通信，主要与engine进行交换

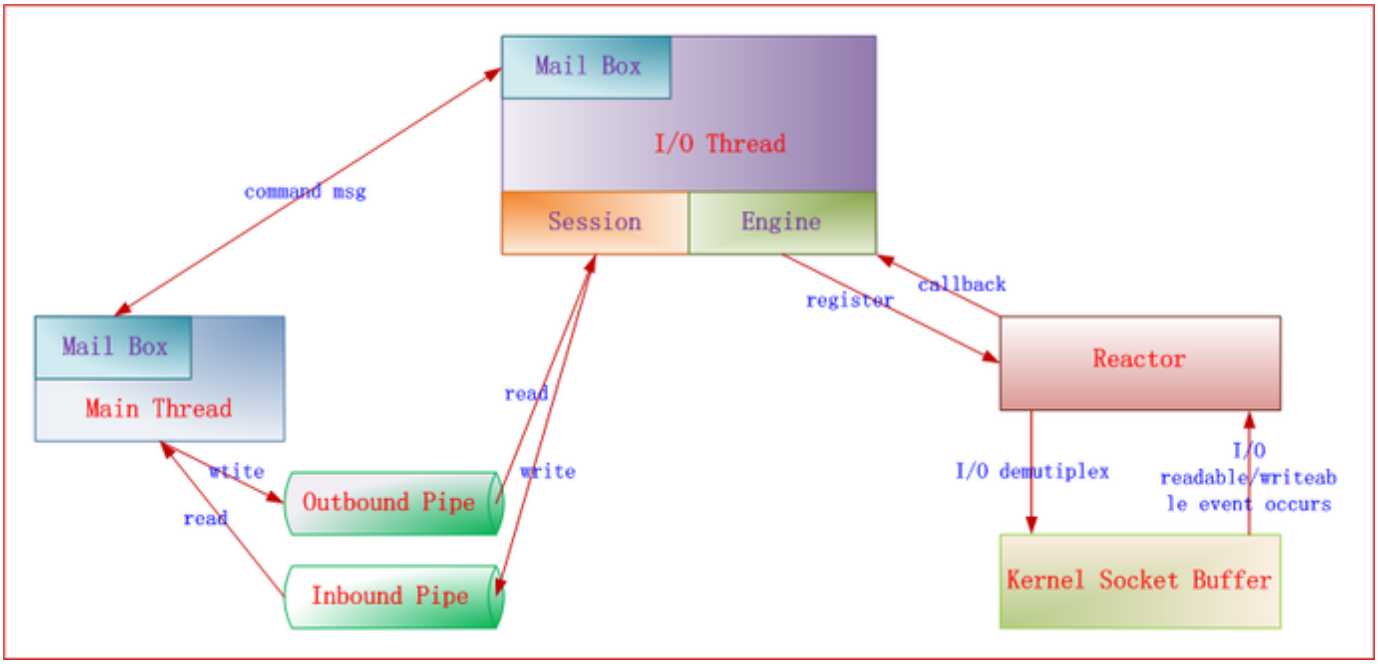
⑩、socket_base_t，zeromq的socket，在zmq中，被当成一种特殊的”线程“，具有收发命令的功能

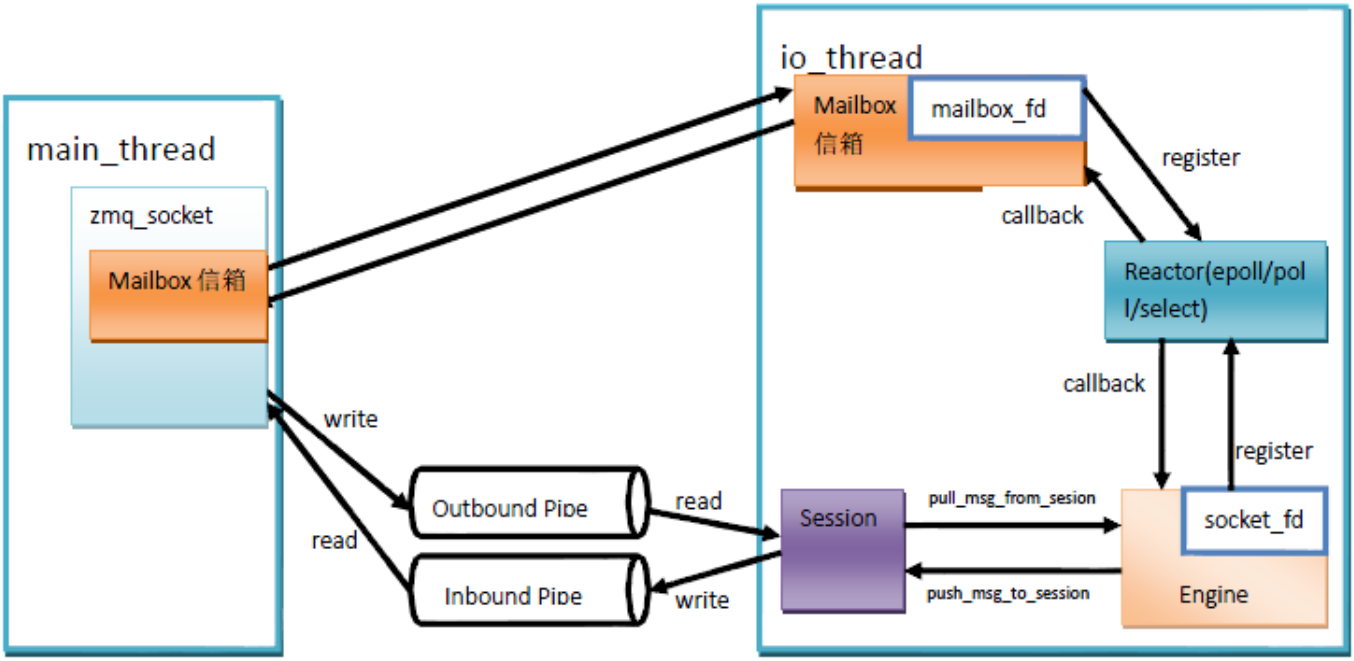
主线程与I/O线程

I/O线程，ZMQ根据用户调用zmq_init函数时传入的参数，创建对应数量的I/O线程。每个I/O线程都有与之绑定的Poller，Poller采用经典的Reactor模式实现。

Poller根据不同操作系统平台使用不同的网络I/O模型（select、poll、epoll、devpoll、kequeue等），所有的I/O操作都是异步的，线程不会被阻塞。。

主线程与I/O线程通过Mail Box传递消息来进行通信。





Server，在主线程创建zmq_listener，通过Mail Box发消息的形式将其绑定到I/O线程，I/O线程把zmq_listener添加到Poller中用以侦听读事件。

Client，在主线程中创建zmq_connecter，通过Mail Box发消息的形式将其绑定到I/O线程，I/O线程把zmq_connecter添加到Poller中用以侦听写事件。

Client与Server第一次通信时，会创建zmq_init来发送identity，用以进行认证。认证结束后，双方会为此次连接创建Session，以后双方就通过Session进行通信。

每个Session**都会关联到相应的读/写管道**， **主线程收发消息只是分别从管道中读/写数据**。Session并不实际跟kernel交换I/O数据，而是通过plugin到Session中的Engine来与kernel交换I/O数据。

Reactor机制(mailbox, event)

zmq在创建的时候会启动两类线程，一是**应用线程**(Application thread)，二是**I/O线程**。那么这些线程之间是如何协作，如何通信的呢?
实际上还有reaper线程。

信号员 signaler

signaler 负责在进程间传递信号。我们课上使用的是Linux系统，所以进程间通信使用`socketpair`来实现。

进程间通信

在UNIX中，socket 一开始是用于网络通信的，后来也应用于**进程间通信**。由于是本地环回，所以就不需要绑定IP地址不需要监听连接了。直接设定一个套接字对儿。

C++ 复制代码

```
int zmq::make_fdpair (fd_t *r_, fd_t *w_)
{
    ..........
    int rc = socketpair (AF_UNIX, type, 0, sv);
    if (rc == -1) {
        errno_assert (errno == ENFILE || errno == EMFILE);
        *w_ = *r_ = -1;
        return -1;
    } else {
        make_socket_noninheritable (sv[0]);
        make_socket_noninheritable (sv[1]);

        *w_ = sv[0];
        *r_ = sv[1];
        return 0;
    }
    ....
}
```

可以认为这是一个**全双工**的管道。在端口`sv[0]`发送，在`sv[1]`就会收到；反之，`sv[1]`发，`sv[0]`收。

signaler 实现

在`signaler_t`中有两个fd成员`w`和`r`，他们代表两个socket。实际并没有用上全双工，而是一个读，一个写。

C++ 复制代码

```
class signaler_t
    {
    public:
      ...
        fd_t get_fd () const;
        void send ();
        int wait (int timeout_);
        void recv ();
        int recv_failable ();

    private:
        //  Creates a pair of file descriptors that will be used
        //  to pass the signals.
        static int make_fdpair (fd_t *r_, fd_t *w_);

        //  Underlying write & read file descriptor
        //  Will be -1 if we exceeded number of available handles
        fd_t w;
        fd_t r;
      ...
    };

// 文件描述符返回的是 r，也就是接收端。
zmq::fd_t zmq::signaler_t::get_fd () const
{
    return r;
}

void zmq::signaler_t::send() {
  ...
    // 用端口 w 发送信号
    ::send(w, ...);
  ...
}

void zmq::signaler_t::recv() {
  ...
    // 用端口 r 接收信号
    ::recv(r, ...);
  ...
}


int zmq::signaler_t::make_fdpair(fd_t *r, fd_t *w){
  ...
  int sv [2];
  int rc = socketpair (AF_UNIX, SOCK_STREAM, 0, sv);
  ...
```

```
49          *w_ = sv [0];
50          *r_ = sv [1];
51      ...
52  }
```

多路复用器poller 监听 socket

在zmq中，端口监听其实是由**I/O多路复用器**poller来实现的。`poller_t`是I/O多路复用的一个抽象类，具体实现的的话，zmq根据系统平台选择poll, select, epoll, kqueue等。在此，我们暂且不管这个I/O多路复用是什么鬼。简而言之，如果把socket fd 注册到poller中，poller 就可以监听该socket 是否有数据进来(poll in)，或者发出(poll out)。

以epoll为例，在loop中监听：

C++ 复制代码

```cpp
void zmq::epoll_t::loop ()
{
    epoll_event ev_buf[max_io_events];

    while (true) {
        //  Execute any due timers.
        int timeout = static_cast<int> (execute_timers ());

        if (get_load () == 0) {
            if (timeout == 0)
                break;

            // TODO sleep for timeout
            continue;
        }

        //  Wait for events.
        int n = epoll_wait (_epoll_fd, &ev_buf[0], max_io_events,
                            timeout ? timeout : -1);
        if (n == -1) {
            errno_assert (errno == EINTR);
            continue;
        }

        for (int i = 0; i < n; i++) {
            poll_entry_t *pe =
              (static_cast<poll_entry_t *> (ev_buf[i].data.ptr));

            if (pe->fd == retired_fd)
                continue;
            if (ev_buf[i].events & (EPOLLERR | EPOLLHUP))
                pe->events->in_event ();
            if (pe->fd == retired_fd)
                continue;
            if (ev_buf[i].events & EPOLLOUT)  // 触发事件
                pe->events->out_event ();
            if (pe->fd == retired_fd)
                continue;
            if (ev_buf[i].events & EPOLLIN) // 触发事件
                pe->events->in_event ();
        }

        //  Destroy retired event sources.
        for (retired_t::iterator it = _retired.begin (), end = _retired.end
();
             it != end; ++it) {
            LIBZMQ_DELETE (*it);
        }
```

```
48          _retired.clear ();
49      }
50  }
```

mailbox

mailbox_t有主要两个成员，一个是管道，另一个是信号员。管道是真正用于收发数据的，而信号员只是负责监视邮箱是否有来信，发信的时候也要发射信号。

C++ 复制代码

```
1  class mailbox_t : public i_mailbox
2  {
3  public:
4    ...
5    // 返回相应文件描述符，用于注册到poller中，监听是否有来信。
6    fd_t get_fd () const;
7    // 发送cmd
8    void send (const command_t &cmd_);
9    // 接收cmd
10   int recv (command_t *cmd_, int timeout_);
11   ...
12 private:
13   ...
14   // 管道，用于收发cmd
15   cpipe_t cpipe;
16   //  Signaler to pass signals from writer thread to reader thread.
17   signaler_t signaler;
18   ...
19 };
```

I/O Thread

`io_thread_t`中有一个邮箱mailbox和一个事件监听poller。

C++ 复制代码

```cpp
class io_thread_t : public object_t, public i_poll_events
{
public:
  ...
  //  Launch the physical thread.
  void start ();
  //  Ask underlying thread to stop.
  void stop ();

  //  Returns mailbox associated with this I/O thread.
  mailbox_t *get_mailbox ();

  //  i_poll_events implementation.
  void in_event ();
  void out_event ();

  //  Used by io_objects to retrieve the associated poller object.
  poller_t *get_poller ();
  ...

private:
  //  I/O thread accesses incoming commands via this mailbox.
  mailbox_t mailbox;
  //  Handle associated with mailbox' file descriptor.
  poller_t::handle_t mailbox_handle;
  //  I/O multiplexing is performed using a poller object.
  poller_t *poller;
  ...
};
```

`io_thread`在创建的时候，需要创建一个`poller`，用于监听事件。
注意到`io_thread_t`继承了`i_poll_events`，所以他自己就可以作为一个**Reactor**。在构造时，需要向`poller`注册自己的mailbox和自己本身(this)：`poller->add_fd (mailbox.get_fd (), this);`。
我觉得这里**非常微妙**，在`poller->loop()`中，可以监听到邮箱mailbox是否有消息进来，然后通过透传数据即可触发`in_event()`。正是因为上述`add_fd`中注册了`this`指针，所以可以很容易地回调自己的成员函数`in_event()`。
构造函数如下：

```C++
zmq::io_thread_t::io_thread_t (ctx_t *ctx_, uint32_t tid_) :
    object_t (ctx_, tid_)
{
    poller = new (std::nothrow) poller_t (*ctx_);
    alloc_assert (poller);

    mailbox_handle = poller->add_fd (mailbox.get_fd (), this);
    poller->set_pollin (mailbox_handle);
}
```

总结

1. `io_thread`创建时向`poller`注册自己的mailbox fd，以及`this`指针，用于事件回调。
2. 在`ctx_t`中，通过`slots [i] = io_thread->get_mailbox ();`注册I/O线程的邮箱。然后通过`slots [tid_]->send (command_);`可以向该邮箱发送指令。
3. 上述`send()`调用的是mailbox 的`send()`，他会向管道`cpipe`写入数据，并发送信号`signaler->send()`。
4. `signaler_t`发送信号其实就是用自己的socket `w`发送。因为`w`和`r`是一对儿，所以`r`会接收到数据。
5. 当有指令进入mailbox时，`poller_t`就会监听到。`poller`监听的是`mailbox->get_fd()`，这个fd正正是上述的`r`。因为`mailbox->get_fd()`返回的是`signaler->get_fd()`，而`signaler->get_fd()`返回的就是`r`。
6. 监听到邮箱有指令进入后，通过`reactor->in_event()`触发事件。因为`io_thread`在创建时向`poller`注册了自己的`this`指针，而`reactor`正是这个`this`指针，于是实现事件触发。
7. 在`io_thread`的`in_event()`中，调用`mailbox->recv()`。于是mailbox向管道cpipe读取指令。至此，io_thread成功收到指令！

以请求响应模型为例

客户端和服务器之间是如何交互的

C++ 复制代码

```cpp
//
//  Hello World 客户端
//  连接REQ套接字至 tcp://localhost:5555
//  发送Hello给服务端，并接收World
//  零声学院: darren QQ 326873713
//  Hello World client
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
//编译: gcc -o hwclient hwclient.c -lzmq
int main (void)
{
    printf ("Connecting to hello world server...\n");
    void *context = zmq_ctx_new ();
    //  连接至服务端的套接字
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        char buffer [10];
        printf ("A 正在发送 Hello %d...\n", request_nbr);
        zmq_send (requester, "Hello", 5, 0);
        zmq_recv (requester, buffer, 5, 0);        // 收到响应才能再发
        //printf ("接收到 Hello World %d\n", request_nbr);
        printf ("B 正在发送 Darren %d...\n", request_nbr);
        //zmq_send (requester, "Darren", 6, 0);   // 无效
        //zmq_send (requester, "king", 4, 0);      // 无效
        zmq_recv (requester, buffer, 6, 0);        // 收到响应才能再发
        printf ("接收到Darren World %d\n", request_nbr);
    }
    zmq_close (requester);
    zmq_ctx_destroy (context);
    return 0;
}
```

C++ 复制代码

```cpp
//
//  Hello World 服务端
//  绑定一个REP套接字至tcp://*:5555
//  从客户端接收Hello，并应答World
//

#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>
//gcc -o hwserver hwserver.c -lzmq
int main (void)
{
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    //  与客户端通信的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5555");  // 服务器要做绑定
    assert (rc == 0);

    while (1) {
        //  等待客户端请求
        char buffer [10];
        int size = zmq_recv (responder, buffer, 10, 0);
        buffer[size] = '\0';
        printf ("收到 %s\n", buffer);
        sleep (1);          //  Do some 'work'
        //  返回应答
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

问题

每个函数调用后内部做了什么?

- void *context = zmq_ctx_new ()
- void *requester = zmq_socket (context, ZMQ_REQ);
- void *responder = zmq_socket (context, ZMQ_REP)的作用
- int rc = zmq_bind (responder, "tcp://*:5555") 的作用
- zmq_connect (requester, "tcp://localhost:5555");
- int size = zmq_recv (responder, buffer, 10, 0);
- zmq_send (responder, "World", 5, 0);

void zmq::stream_engine_t::plug (io_thread_t *io_thread_,
                                 session_base_t *session_)

客户端断点

执行void *requester = zmq_socket (context, ZMQ_REQ);

1.创建回收线程

Breakpoint 2, zmq::thread_t::start (this=this@entry=0x617ed8,
    tfn_=tfn_@entry=0x7ffff7b7cd40 <zmq::worker_poller_base_t::worker_routine(void*)>,
    arg_=arg_@entry=0x617e80, name_=name_@entry=0x7fffffffe2f0 "ZMQbg/Reaper") at src/thread.cpp:234
234   if (name_)
(gdb) bt
#0 zmq::thread_t::start (this=this@entry=0x617ed8,
    tfn_=tfn_@entry=0x7ffff7b7cd40 <zmq::worker_poller_base_t::worker_routine(void*)>,
    arg_=arg_@entry=0x617e80, name_=name_@entry=0x7fffffffe2f0 "ZMQbg/Reaper") at src/thread.cpp:234
#1 0x00007ffff7b54d08 in zmq::thread_ctx_t::start_thread (this=<optimized out>, thread_=...,
    tfn_=0x7ffff7b7cd40 <zmq::worker_poller_base_t::worker_routine(void*)>, arg_=0x617e80,
    name_=0x7ffff7bb4e07 "Reaper") at src/ctx.cpp:433
#2 0x00007ffff7b562c2 in zmq::ctx_t::start (this=this@entry=0x6141b0) at src/**ctx.cpp:311**
#3 0x00007ffff7b56cd0 in zmq::ctx_t::create_socket (this=0x6141b0, type_=3) at src/ctx.cpp:356
#4 0x0000000000400aa1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:17

2.创建IO线程

#0 zmq::thread_t::start (this=this@entry=0x6186c8,
    tfn_=tfn_@entry=0x7ffff7b7cd40 <zmq::worker_poller_base_t::worker_routine(void*)>,
    arg_=arg_@entry=0x618670, name_=name_@entry=0x7fffffffe2b0 "ZMQbg/IO/0") at src/thread.cpp:234
#1 0x00007ffff7b54d08 in zmq::thread_ctx_t::start_thread (this=<optimized out>, thread_=...,
    tfn_=0x7ffff7b7cd40 <zmq::worker_poller_base_t::worker_routine(void*)>, arg_=0x618670,
    name_=0x7fffffffe300 "IO/0") at src/ctx.cpp:433
#2 0x00007ffff7b67e22 in zmq::io_thread_t::start (this=this@entry=0x6180c0) at src/**io_thread.cpp:63**
#3 0x00007ffff7b563a6 in zmq::ctx_t::start (this=this@entry=0x6141b0) at src/ctx.cpp:329
#4 0x00007ffff7b56cd0 in zmq::ctx_t::create_socket (this=0x6141b0, type_=3) at src/ctx.cpp:356

#5 0x0000000000400aa1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:17

zmq::epoll_t::loop

Thread 2 "ZMQbg/Reaper" hit Breakpoint 3, zmq::epoll_t::loop (this=0x617e80) at src/epoll.cpp:168
Thread 3 "ZMQbg/IO/0" hit Breakpoint 3, zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:168

zmq::mailbox_t::send

Thread 1 "hwclient" hit Breakpoint 4, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:60
60 {
(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:60
#1 0x00007ffff7b73c43 in zmq::object_t::send_command (cmd_=..., this=0x618890) at src/object.cpp:533
#2 zmq::object_t::send_plug (this=this@entry=0x618890,
destination_=destination_@entry=0x619510,
    inc_seqnum_=inc_seqnum_@entry=true) at src/object.cpp:233
#3 0x00007ffff7b76ca7 in zmq::own_t::launch_child (this=this@entry=0x618890,
object_=object_@entry=0x619510)
    at src/own.cpp:87
#4 0x00007ffff7b8e3de in zmq::socket_base_t::add_endpoint (this=this@entry=0x618890,
endpoint_pair_=...,
    endpoint_=endpoint_@entry=0x619510, pipe_=pipe_@entry=0x621be0) at src/socket_base.cpp:1020
#5 0x00007ffff7b8fc70 in zmq::socket_base_t::**connect** (this=0x618890,
    endpoint_uri_=0x400c3c "tcp://localhost:5555") at src/socket_base.cpp:984
#6 0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:18

Thread 3 "ZMQbg/IO/0" hit Breakpoint 4, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:60
60 {
(gdb) bt
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:60

#1 0x00007ffff7b73d7f in zmq::object_t::send_command (cmd_=..., this=0x7ffff00008c0) at src/object.cpp:533
#2 zmq::object_t::send_attach (this=this@entry=0x7ffff00008c0, destination_=0x619510, engine_=0x7ffff00017e0,
    inc_seqnum_=inc_seqnum_@entry=true) at src/object.cpp:257
#3 0x00007ffff7b97156 in zmq::stream_connecter_base_t::create_engine (this=this@entry=0x7ffff00008c0,
    fd=fd@entry=10, local_address_="tcp://127.0.0.1:48860") at src/stream_connecter_base.cpp:181
#4 0x00007ffff7b9f8ef in zmq::tcp_connecter_t::out_event (this=0x7ffff00008c0) at src/tcp_connecter.cpp:114
#5 0x00007ffff7b6682a in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:202
#6 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#8 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

Thread 1 "hwclient" hit Breakpoint 4, zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:60
60 {
#0 zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:60
#1 0x00007ffff7b73e9d in zmq::object_t::send_command (cmd_=..., this=<optimized out>) at src/object.cpp:533
#2 zmq::object_t::send_activate_read (this=<optimized out>, destination_=<optimized out>) at src/object.cpp:279
#3 0x00007ffff7b790a3 in zmq::pipe_t::flush (this=<optimized out>) at src/pipe.cpp:269
#4 0x00007ffff7b6b3b0 in zmq::lb_t::sendpipe (this=this@entry=0x618ed8, msg_=msg_@entry=0x7fffffffe380,
    pipe_=pipe_@entry=0x0) at src/lb.cpp:147
#5 0x00007ffff7b6397e in zmq::dealer_t::sendpipe (pipe_=0x0, msg_=msg_@entry=0x7fffffffe380,
    this=this@entry=0x618890) at src/dealer.cpp:138
#6 zmq::dealer_t::xsend (this=this@entry=0x618890, msg_=msg_@entry=0x7fffffffe380) at src/dealer.cpp:102
#7 0x00007ffff7b83565 in zmq::req_t::xsend (this=0x618890, msg_=0x7fffffffe380) at src/req.cpp:118
#8 0x00007ffff7b8b860 in zmq::socket_base_t::send (this=this@entry=0x618890, msg_=msg_@entry=0x7fffffffe380,
    flags_=flags_@entry=0) at src/socket_base.cpp:1123
#9 0x00007ffff7bac819 in s_sendmsg (flags_=0, msg_=0x7fffffffe380, s_=0x618890) at src/zmq.cpp:338

#10 zmq_send (s_=<optimized out>, buf_=0x400c6d, len_=<optimized out>, flags_=0) at src/zmq.cpp:370
#11 0x0000000000400af1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:24

zmq::mailbox_t::recv

send

Thread 3 "ZMQbg/IO/0" hit Breakpoint 6, __libc_send (fd=10, buf=0x7ffff0002710, n=9, flags=flags@entry=0)
    at ../sysdeps/unix/sysv/linux/x86_64/send.c:26
26   ../sysdeps/unix/sysv/linux/x86_64/send.c: No such file or directory.
(gdb) bt
#0 __libc_send (fd=10, buf=0x7ffff0002710, n=9, flags=flags@entry=0)
    at ../sysdeps/unix/sysv/linux/x86_64/send.c:26
#1 0x00007ffff7b9decb in zmq::tcp_write (s_=<optimized out>, data_=<optimized out>, size_=<optimized out>)
    at src/tcp.cpp:241
#2 0x00007ffff7b97e47 in zmq::stream_engine_t::out_event (this=0x7ffff00017d0) at src/stream_engine.cpp:428   // 数据从哪里来
#3 0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#4 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#5 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#6 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#7 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

recv

Thread 3 "ZMQbg/IO/0" hit Breakpoint 7, zmq::tcp_read (s_=10, data_=0x7ffff0004728, size_=8192)
    at src/tcp.cpp:298
298   const ssize_t rc = recv (s_, static_cast<char *> (data_), size_, 0);
(gdb) bt
#0 zmq::tcp_read (s_=10, data_=0x7ffff0004728, size_=8192) at src/tcp.cpp:298
#1 0x00007ffff7b99442 in zmq::stream_engine_t::in_event_internal (this=0x7ffff00017d0)
    at src/stream_engine.cpp:333   // 读到数据后发给谁
#2 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#3 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225

#4 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#5 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

read

Thread 3 "ZMQbg/IO/0" hit Breakpoint 4, read () at ../sysdeps/unix/syscall-template.S:84
84 ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) bt
#0 read () at ../sysdeps/unix/syscall-template.S:84
#1 0x00007ffff7b89b9d in read (__nbytes=8, __buf=0x7ffff62ac0b8, __fd=<optimized out>)
    at /usr/include/x86_64-linux-gnu/bits/unistd.h:44
#2 zmq::signaler_t::recv_failable (this=this@entry=0x618148) at src/signaler.cpp:339
#3 0x00007ffff7b6bc94 in zmq::mailbox_t::recv (this=this@entry=0x6180e0,
cmd_=cmd_@entry=0x7ffff62ac140,
    timeout_=timeout_@entry=0) at src/mailbox.cpp:88
#4 0x00007ffff7b68108 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:87
#5 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#6 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#8 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

Thread 1 "hwclient" hit Breakpoint 4, read () at ../sysdeps/unix/syscall-template.S:84
84 in ../sysdeps/unix/syscall-template.S
(gdb) bt
#0 read () at ../sysdeps/unix/syscall-template.S:84
#1 0x00007ffff7b89b9d in read (__nbytes=8, __buf=0x7fffffffe1b8, __fd=<optimized out>)
    at /usr/include/x86_64-linux-gnu/bits/unistd.h:44
#2 zmq::signaler_t::recv_failable (this=this@entry=0x618f98) at src/signaler.cpp:339
#3 0x00007ffff7b6bc94 in zmq::mailbox_t::recv (this=0x618f30, cmd_=0x7fffffffe240,
timeout_=<optimized out>)
    at src/mailbox.cpp:88
#4 0x00007ffff7b8b677 in zmq::socket_base_t::process_commands
(this=this@entry=0x618890,
    timeout_=timeout_@entry=0, throttle_=throttle_@entry=true) at src/socket_base.cpp:1365
#5 0x00007ffff7b8b7bf in zmq::socket_base_t::send (this=this@entry=0x618890,
msg_=msg_@entry=0x7fffffffe380,
    flags_=flags_@entry=0) at src/socket_base.cpp:1108

#6 0x00007ffff7bac819 in s_sendmsg (flags_=0, msg_=0x7fffffffe380, s_=0x618890) at src/zmq.cpp:338
#7 zmq_send (s_=<optimized out>, buf_=0x400c6d, len_=<optimized out>, flags_=0) at src/zmq.cpp:370
#8 0x0000000000400af1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:24

connect

Thread 3 "ZMQbg/IO/0" hit Breakpoint 12, connect () at ../sysdeps/unix/syscall-template.S:84
84 ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) bt
#0 connect () at ../sysdeps/unix/syscall-template.S:84
#1 0x00007ffff78b476f in open_socket (type=type@entry=GETFDHST, key=key@entry=0x7ffff7904787 "hosts",
    keylen=keylen@entry=6) at nscd_helper.c:183
#2 0x00007ffff78b4e0d in __nscd_get_mapping (type=type@entry=GETFDHST, key=key@entry=0x7ffff7904787 "hosts",
    mappedp=mappedp@entry=0x7ffff7b3e978 <__hst_map_handle+8>) at nscd_helper.c:269
#3 0x00007ffff78b5367 in __nscd_get_map_ref (type=type@entry=GETFDHST, name=name@entry=0x7ffff7904787 "hosts",
    mapptr=mapptr@entry=0x7ffff7b3e970 <__hst_map_handle>, gc_cyclep=gc_cyclep@entry=0x7ffff62ab55c)
    at nscd_helper.c:419
#4 0x00007ffff78b24e6 in nscd_gethst_r (key=key@entry=0x7ffff62abeb0 "localhost", keylen=10,
    type=type@entry=GETHOSTBYNAME, resultbuf=resultbuf@entry=0x7ffff62abbd0,
    buffer=buffer@entry=0x7ffff62ab700 "", buflen=buflen@entry=912, result=0x7ffff62abbc8,
    h_errnop=0x7ffff62abbac) at nscd_gethst_r.c:160
#5 0x00007ffff78b2dca in __nscd_gethostbyname2_r (name=name@entry=0x7ffff62abeb0 "localhost", af=af@entry=2,
    resultbuf=resultbuf@entry=0x7ffff62abbd0, buffer=0x7ffff62ab700 "", buflen=buflen@entry=912,
    result=result@entry=0x7ffff62abbc8, h_errnop=0x7ffff62abbac) at nscd_gethst_r.c:61
#6 0x00007ffff788f36b in __gethostbyname2_r (name=name@entry=0x7ffff62abeb0 "localhost", af=af@entry=2,
    resbuf=resbuf@entry=0x7ffff62abbd0, buffer=buffer@entry=0x7ffff62ab700 "", buflen=buflen@entry=912,
    result=result@entry=0x7ffff62abbc8, h_errnop=0x7ffff62abbac) at ../nss/getXXbyYY_r.c:196

#7 0x00007ffff786167f in gaih_inet (name=name@entry=0x7ffff62abeb0 "localhost", service=
<optimized out>,
    req=req@entry=0x7ffff62abdf0, pai=pai@entry=0x7ffff62abca8,
naddrs=naddrs@entry=0x7ffff62abca4)
---Type <return> to continue, or q <return> to quit---
    at ../sysdeps/posix/getaddrinfo.c:622
#8 0x00007ffff7863e1e in __GI_getaddrinfo (name=<optimized out>,
name@entry=0x7ffff62abeb0 "localhost",
    service=service@entry=0x0, hints=hints@entry=0x7ffff62abdf0,
pai=pai@entry=0x7ffff62abde8)
    at ../sysdeps/posix/getaddrinfo.c:2425
#9 0x00007ffff7b68ae5 in zmq::ip_resolver_t::do_getaddrinfo (res_=0x7ffff62abde8,
hints_=0x7ffff62abdf0,
    service_=0x0, node_=0x7ffff62abeb0 "localhost", this=0x7ffff62abfb0) at
src/ip_resolver.cpp:697
#10 zmq::ip_resolver_t::resolve_getaddrinfo (this=this@entry=0x7ffff62abfb0,
    ip_addr_=ip_addr_@entry=0x7ffff0000dd0, addr_=addr_@entry=0x7ffff62abeb0 "localhost")
    at src/ip_resolver.cpp:333
#11 0x00007ffff7b6929e in zmq::ip_resolver_t::resolve (this=this@entry=0x7ffff62abfb0,
    ip_addr_=ip_addr_@entry=0x7ffff0000dd0, name_=name_@entry=0x6194e0
"localhost:5555")
    at src/ip_resolver.cpp:270
#12 0x00007ffff7b9e604 in zmq::tcp_address_t::resolve (this=this@entry=0x7ffff0000dd0,
    name_=name_@entry=0x6194e0 "localhost:5555", local_=local_@entry=false, ipv6_=
<optimized out>)
    at src/tcp_address.cpp:112
#13 0x00007ffff7b9e0c3 in zmq::tcp_open_socket (address_=0x6194e0 "localhost:5555",
options_=...,
    local_=local_@entry=false, fallback_to_ipv4_=fallback_to_ipv4_@entry=true,
out_tcp_addr_=0x7ffff0000dd0)
    at src/tcp.cpp:394
#14 0x00007ffff7b9f293 in zmq::tcp_connecter_t::open (this=this@entry=0x7ffff00008c0)
    at src/tcp_connecter.cpp:177
#15 0x00007ffff7b9f552 in **zmq::tcp_connecter_t::start_connecting** (this=0x7ffff00008c0)
    at src/tcp_connecter.cpp:131
#16 0x00007ffff7b73a16 in zmq::object_t::process_command (this=0x7ffff00008c0, cmd_=...) at
src/object.cpp:87 （这里是plug命令）
#17 0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#18 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#19 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225

```
#20 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#21 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109


[Switching to Thread 0x7ffff62ad700 (LWP 2683)]
Thread 3 "ZMQbg/IO/0" hit Breakpoint 3, connect () at ../sysdeps/unix/syscall-template.S:84
84    ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) bt
#0  connect () at ../sysdeps/unix/syscall-template.S:84
#1  0x00007ffff78b476f in open_socket (type=type@entry=GETFDHST,
key=key@entry=0x7ffff7904787 "hosts",
    keylen=keylen@entry=6) at nscd_helper.c:183
#2  0x00007ffff78b4e0d in __nscd_get_mapping (type=type@entry=GETFDHST,
key=key@entry=0x7ffff7904787 "hosts",
    mappedp=mappedp@entry=0x7ffff7b3e978 <__hst_map_handle+8>) at nscd_helper.c:269
#3  0x00007ffff78b5367 in __nscd_get_map_ref (type=type@entry=GETFDHST,
name=name@entry=0x7ffff7904787 "hosts",
    mapptr=mapptr@entry=0x7ffff7b3e970 <__hst_map_handle>,
gc_cyclep=gc_cyclep@entry=0x7ffff62ab55c)
    at nscd_helper.c:419
#4  0x00007ffff78b24e6 in nscd_gethst_r (key=key@entry=0x7ffff62abeb0 "localhost",
keylen=10,
    type=type@entry=GETHOSTBYNAME, resultbuf=resultbuf@entry=0x7ffff62abbd0,
    buffer=buffer@entry=0x7ffff62ab700 "", buflen=buflen@entry=912, result=0x7ffff62abbc8,
    h_errnop=0x7ffff62abbac) at nscd_gethst_r.c:160
#5  0x00007ffff78b2dca in __nscd_gethostbyname2_r (name=name@entry=0x7ffff62abeb0
"localhost", af=af@entry=2,
    resultbuf=resultbuf@entry=0x7ffff62abbd0, buffer=0x7ffff62ab700 "",
buflen=buflen@entry=912,
    result=result@entry=0x7ffff62abbc8, h_errnop=0x7ffff62abbac) at nscd_gethst_r.c:61
#6  0x00007ffff788f36b in __gethostbyname2_r (name=name@entry=0x7ffff62abeb0
"localhost", af=af@entry=2,
    resbuf=resbuf@entry=0x7ffff62abbd0, buffer=buffer@entry=0x7ffff62ab700 "",
buflen=buflen@entry=912,
    result=result@entry=0x7ffff62abbc8, h_errnop=0x7ffff62abbac) at ../nss/getXXbyYY_r.c:196
#7  0x00007ffff786167f in gaih_inet (name=name@entry=0x7ffff62abeb0 "localhost", service=
<optimized out>,
    req=req@entry=0x7ffff62abdf0, pai=pai@entry=0x7ffff62abca8,
naddrs=naddrs@entry=0x7ffff62abca4)
---Type <return> to continue, or q <return> to quit---
```

```
    at ../sysdeps/posix/getaddrinfo.c:622
#8  0x00007ffff7863e1e in __GI_getaddrinfo (name=<optimized out>,
name@entry=0x7ffff62abeb0 "localhost",
    service=service@entry=0x0, hints=hints@entry=0x7ffff62abdf0,
pai=pai@entry=0x7ffff62abde8)
    at ../sysdeps/posix/getaddrinfo.c:2425
#9  0x00007ffff7b68ae5 in zmq::ip_resolver_t::do_getaddrinfo (res_=0x7ffff62abde8,
hints_=0x7ffff62abdf0,
    service_=0x0, node_=0x7ffff62abeb0 "localhost", this=0x7ffff62abfb0) at
src/ip_resolver.cpp:697
#10 zmq::ip_resolver_t::resolve_getaddrinfo (this=this@entry=0x7ffff62abfb0,
    ip_addr_=ip_addr_@entry=0x7ffff0000dd0, addr_=addr_@entry=0x7ffff62abeb0 "localhost")
    at src/ip_resolver.cpp:333
#11 0x00007ffff7b6929e in zmq::ip_resolver_t::resolve (this=this@entry=0x7ffff62abfb0,
    ip_addr_=ip_addr_@entry=0x7ffff0000dd0, name_=name_@entry=0x6194e0
"localhost:5555")
    at src/ip_resolver.cpp:270
#12 0x00007ffff7b9e604 in zmq::tcp_address_t::resolve (this=this@entry=0x7ffff0000dd0,
    name_=name_@entry=0x6194e0 "localhost:5555", local_=local_@entry=false, ipv6_=
<optimized out>)
    at src/tcp_address.cpp:112
#13 0x00007ffff7b9e0c3 in zmq::tcp_open_socket (address_=0x6194e0 "localhost:5555",
options_=...,
    local_=local_@entry=false, fallback_to_ipv4_=fallback_to_ipv4_@entry=true,
out_tcp_addr_=0x7ffff0000dd0)
    at src/tcp.cpp:394
#14 0x00007ffff7b9f293 in zmq::tcp_connecter_t::open (this=this@entry=0x7ffff00008c0)
    at src/tcp_connecter.cpp:177
#15 0x00007ffff7b9f552 in zmq::tcp_connecter_t::start_connecting (this=0x7ffff00008c0)
    at src/tcp_connecter.cpp:131
#16 0x00007ffff7b73a16 in zmq::object_t::process_command (this=0x7ffff00008c0, cmd_=...) at
src/object.cpp:87
#17 0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#18 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#19 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#20 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#21 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109


Thread 3 "ZMQbg/IO/0" hit Breakpoint 3, connect () at ../sysdeps/unix/syscall-template.S:84
```

```
84    in ../sysdeps/unix/syscall-template.S
(gdb) bt
#0  connect () at ../sysdeps/unix/syscall-template.S:84
#1  0x00007ffff7b9f357 in zmq::tcp_connecter_t::open (this=this@entry=0x7ffff00008c0)
    at src/tcp_connecter.cpp:226
#2  0x00007ffff7b9f552 in zmq::tcp_connecter_t::start_connecting (this=0x7ffff00008c0)
    at src/tcp_connecter.cpp:131
#3  0x00007ffff7b73a16 in zmq::object_t::process_command (this=0x7ffff00008c0, cmd_=...)
at src/object.cpp:87
#4  0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#5  0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#6  0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7  0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#8  0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
```

zmq::stream_engine_t::stream_engine_t

```
Thread 3 "ZMQbg/IO/0" hit Breakpoint 8, zmq::stream_engine_t::stream_engine_t
(this=0x7ffff0001720, fd_=10,
    options_=..., endpoint_uri_pair_=...) at src/stream_engine.cpp:103
103   zmq::stream_engine_t::stream_engine_t (
(gdb) bt
#0  zmq::stream_engine_t::stream_engine_t (this=0x7ffff0001720, fd_=10, options_=...,
endpoint_uri_pair_=...)
    at src/stream_engine.cpp:103
#1  0x00007ffff7b9713d in zmq::stream_connecter_base_t::create_engine
(this=this@entry=0x7ffff00008c0,
    fd=fd@entry=10, local_address_="tcp://127.0.0.1:48908") at
src/stream_connecter_base.cpp:177
#2  0x00007ffff7b9f8ef in zmq::tcp_connecter_t::out_event (this=0x7ffff00008c0) at
src/tcp_connecter.cpp:114
#3  0x00007ffff7b6682a in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:202
#4  0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#5  0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#6  0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
```

zmq::epoll_t::add_fd

```
Breakpoint 9, zmq::epoll_t::add_fd (this=0x617e80, fd_=5, events_=events_@entry=0x6178d8) at
src/epoll.cpp:85
85    {
```

```
(gdb) bt
#0  zmq::epoll_t::add_fd (this=0x617e80, fd_=5, events_=events_@entry=0x6178d8) at
src/epoll.cpp:85
#1  0x00007ffff7b828bd in zmq::reaper_t::reaper_t (this=0x6178c0, ctx_=<optimized out>, tid_=
<optimized out>)
    at src/reaper.cpp:50
#2  0x00007ffff7b5627c in zmq::ctx_t::start (this=this@entry=0x6141b0) at src/ctx.cpp:303
#3  0x00007ffff7b56cd0 in zmq::ctx_t::create_socket (this=0x6141b0, type_=3) at
src/ctx.cpp:356
#4  0x0000000000400aa1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:17

#0  zmq::epoll_t::add_fd (this=0x618670, fd_=7, events_=events_@entry=0x6180d8) at
src/epoll.cpp:85
#1  0x00007ffff7b67f76 in zmq::io_thread_t::io_thread_t (this=0x6180c0, ctx_=0x6141b0, tid_=
<optimized out>)
    at src/io_thread.cpp:47
#2  0x00007ffff7b56336 in zmq::ctx_t::start (this=this@entry=0x6141b0) at src/ctx.cpp:318
#3  0x00007ffff7b56cd0 in zmq::ctx_t::create_socket (this=0x6141b0, type_=3) at
src/ctx.cpp:356
#4  0x0000000000400aa1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:17
```

zmq::mailbox_t::send

```
#0  zmq::mailbox_t::send (this=0x6180e0, cmd_=...) at src/mailbox.cpp:62
#1  0x00007ffff7b73cdb in zmq::object_t::send_command (cmd_=..., this=0x619510) at
src/object.cpp:533
#2  zmq::object_t::send_own (this=0x619510, destination_=0x619510, object_=0x7ffff00008c0)
at src/object.cpp:243
#3  0x00007ffff7b73a16 in zmq::object_t::process_command (this=0x619510, cmd_=...) at
src/object.cpp:87
#4  0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#5  0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#6  0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7  0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#8  0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
```

zmq::session_base_t::create

```
Thread 1 "hwclient" hit Breakpoint 11, zmq::session_base_t::create
(io_thread_=io_thread_@entry=0x6180c0,
    active_=active_@entry=true, socket_=socket_@entry=0x618890, options_=...,
addr_=addr_@entry=0x6194b0)
    at src/session_base.cpp:58
58    {
(gdb) bt
#0  zmq::session_base_t::create (io_thread_=io_thread_@entry=0x6180c0,
active_=active_@entry=true,
    socket_=socket_@entry=0x618890, options_=..., addr_=addr_@entry=0x6194b0) at
src/session_base.cpp:58
#1  0x00007ffff7b8fae5 in zmq::socket_base_t::connect (this=0x618890,
    endpoint_uri_=0x400c3c "tcp://localhost:5555") at src/socket_base.cpp:949
#2  0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:18


#0  zmq::signaler_t::wait (this=this@entry=0x618f98, timeout_=-1) at src/signaler.cpp:227
#1  0x00007ffff7b6bc85 in zmq::mailbox_t::recv (this=0x618f30, cmd_=0x7fffffffe240,
timeout_=<optimized out>)
    at src/mailbox.cpp:81
#2  0x00007ffff7b8b677 in zmq::socket_base_t::process_commands
(this=this@entry=0x618890, timeout_=-1,
    throttle_=throttle_@entry=false) at src/socket_base.cpp:1365
#3  0x00007ffff7b8c40b in zmq::socket_base_t::recv (this=this@entry=0x618890,
msg_=msg_@entry=0x7fffffffe370,
    flags_=flags_@entry=0) at src/socket_base.cpp:1250
#4  0x00007ffff7bac229 in s_recvmsg (s_=s_@entry=0x618890,
msg_=msg_@entry=0x7fffffffe370,
    flags_=flags_@entry=0) at src/zmq.cpp:454
#5  0x00007ffff7bac5b9 in zmq_recv (s_=<optimized out>, buf_=0x7fffffffe420, len_=5,
flags_=0)
    at src/zmq.cpp:479
#6  0x0000000000400b0b in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:25
```

zmq::object_t::send_plug

zmq::req_session_t::req_session_t

Thread 1 "hwclient" hit Breakpoint 10, zmq::req_session_t::req_session_t (this=0x619510, io_thread_=0x6180c0,
    connect_=true, socket_=0x618890, options_=..., addr_=0x6194b0) at src/req.cpp:259
259 zmq::req_session_t::req_session_t (io_thread_t *io_thread_,
(gdb) bt
#0 zmq::req_session_t::req_session_t (this=0x619510, io_thread_=0x6180c0, connect_=true, socket_=0x618890,
    options_=..., addr_=0x6194b0) at src/req.cpp:259
#1 0x00007ffff7b86f63 in zmq::session_base_t::create (io_thread_=io_thread_@entry=0x6180c0,
    active_=active_@entry=true, socket_=socket_@entry=0x618890, options_=..., addr_=addr_@entry=0x6194b0)
    at src/session_base.cpp:63
#2 0x00007ffff7b8fae5 in zmq::socket_base_t::connect (this=0x618890,
    endpoint_uri_=0x400c3c "tcp://localhost:5555") at src/socket_base.cpp:949
#3 0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:18

zmq::req_t::xrecv

Thread 1 "hwclient" hit Breakpoint 11, zmq::req_t::xrecv (this=0x618890, msg_=0x7fffffffe370) at src/req.cpp:132
132 {
(gdb) bt
#0 zmq::req_t::xrecv (this=0x618890, msg_=0x7fffffffe370) at src/req.cpp:132
#1 0x00007ffff7b8c588 in zmq::socket_base_t::recv (this=this@entry=0x618890, msg_=msg_@entry=0x7fffffffe370,
    flags_=flags_@entry=0) at src/socket_base.cpp:1253
#2 0x00007ffff7bac229 in s_recvmsg (s_=s_@entry=0x618890, msg_=msg_@entry=0x7fffffffe370,
    flags_=flags_@entry=0) at src/zmq.cpp:454
#3 0x00007ffff7bac5b9 in zmq_recv (s_=<optimized out>, buf_=0x7fffffffe420, len_=5, flags_=0)
    at src/zmq.cpp:479
#4 0x0000000000400b0b in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:25

zmq::req_session_t::push_msg

Thread 3 "ZMQbg/IO/0" hit Breakpoint 9, zmq::req_session_t::push_msg (this=0x619510, msg_=0x7ffff0001e98)
    at src/req.cpp:274
#0 zmq::req_session_t::push_msg (this=0x619510, msg_=0x7ffff0001e98) at src/req.cpp:274
#1 0x00007ffff7b9a8aa in zmq::stream_engine_t::decode_and_push (this=this@entry=0x7ffff0001720,
    msg_=msg_@entry=0x7ffff0001e98) at src/stream_engine.cpp:1053
#2 0x00007ffff7b9aa91 in zmq::stream_engine_t::write_credential (this=0x7ffff0001720, msg_=0x7ffff0001e98)
    at src/stream_engine.cpp:1016
#3 0x00007ffff7b99310 in zmq::stream_engine_t::in_event_internal (this=0x7ffff0001720)
    at src/stream_engine.cpp:365
#4 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#5 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#6 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#7 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

zmq::pipe_t::read

Thread 3 "ZMQbg/IO/0" hit Breakpoint 13, zmq::pipe_t::read (this=0x621cf0, msg_=msg_@entry=0x7ffff0001740)
    at src/pipe.cpp:185
185 {
(gdb) bt
#0 zmq::pipe_t::read (this=0x621cf0, msg_=msg_@entry=0x7ffff0001740) at src/pipe.cpp:185
#1 0x00007ffff7b8669d in zmq::session_base_t::pull_msg (this=0x619510, msg_=0x7ffff0001740)
    at src/session_base.cpp:154
#2 0x00007ffff7b97c07 in zmq::stream_engine_t::pull_and_encode (this=0x7ffff0001720, msg_=0x7ffff0001740)
    at src/stream_engine.cpp:1023
#3 0x00007ffff7b97db3 in zmq::stream_engine_t::out_event (this=0x7ffff0001720) at src/stream_engine.cpp:403
#4 0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91
#5 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206
#6 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225
#7 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333
#8 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

zmq::stream_engine_t::out_event

读取主线程发送过来的消息，打包后发送到接收端

#0 zmq::session_base_t::pull_msg (this=0x619510, msg_=0x7ffff0001740) at src/session_base.cpp:159

#1 0x00007ffff7b97c07 in zmq::stream_engine_t::pull_and_encode (this=0x7ffff0001720, msg_=0x7ffff0001740)

    at src/stream_engine.cpp:1023

#2 0x00007ffff7b97db3 in zmq::stream_engine_t::out_event (this=0x7ffff0001720) at src/stream_engine.cpp:403 到这层级查看代码

#3 0x00007ffff7b68127 in zmq::io_thread_t::in_event (this=0x6180c0) at src/io_thread.cpp:91

#4 0x00007ffff7b6684e in zmq::epoll_t::loop (this=0x618670) at src/epoll.cpp:206

#5 0x00007ffff7ba07eb in thread_routine (arg_=0x6186c8) at src/thread.cpp:225

#6 0x00007ffff73576ba in start_thread (arg=0x7ffff62ad700) at pthread_create.c:333

#7 0x00007ffff787c4dd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109

zmq::stream_engine_t::in_event

处理接收端发送过来的消息?

最终调用zmq::stream_engine_t::in_event_internal

主要流程:

调用tcp_read接收数据

调用decode解码

然后通过session通过pipe发送给主线程

Thread 1 "hwclient" hit Breakpoint 14, zmq::pipe_t::write (this=0x621be0, msg_=msg_@entry=0x7fffffffe380)

    at src/pipe.cpp:236

236 {

(gdb) bt

#0 zmq::pipe_t::write (this=0x621be0, msg_=msg_@entry=0x7fffffffe380) at src/pipe.cpp:236

#1 0x00007ffff7b6b201 in zmq::lb_t::sendpipe (this=this@entry=0x618ed8, msg_=msg_@entry=0x7fffffffe380,

    pipe_=pipe_@entry=0x0) at src/lb.cpp:99

#2 0x00007ffff7b6397e in zmq::dealer_t::sendpipe (pipe_=0x0, msg_=msg_@entry=0x7fffffffe380,

    this=this@entry=0x618890) at src/dealer.cpp:138

#3 zmq::dealer_t::xsend (this=this@entry=0x618890, msg_=msg_@entry=0x7fffffffe380) at src/dealer.cpp:102

#4 0x00007ffff7b83565 in zmq::req_t::xsend (this=0x618890, msg_=0x7fffffffe380) at src/req.cpp:118
#5 0x00007ffff7b8b860 in zmq::socket_base_t::send (this=this@entry=0x618890, msg_=msg_@entry=0x7fffffffe380,
    flags_=flags_@entry=0) at src/socket_base.cpp:1123
#6 0x00007ffff7bac819 in s_sendmsg (flags_=0, msg_=0x7fffffffe380, s_=0x618890) at src/zmq.cpp:338
#7 zmq_send (s_=<optimized out>, buf_=0x400c6d, len_=<optimized out>, flags_=0) at src/zmq.cpp:370
#8 0x0000000000400af1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:24

zmq::pipepair

Thread 1 "hwclient" hit Breakpoint 18, zmq::pipepair (parents_=parents_@entry=0x7fffffffdb50, pipes_=pipes_@entry=0x7fffffffdf80,
    hwms_=hwms_@entry=0x7fffffffdb30, conflate_=conflate_@entry=0x7fffffffdb10) at src/pipe.cpp:45
45 {
(gdb) bt
#0 zmq::pipepair (parents_=parents_@entry=0x7fffffffdb50, pipes_=pipes_@entry=0x7fffffffdf80, hwms_=hwms_@entry=0x7fffffffdb30,
    conflate_=conflate_@entry=0x7fffffffdb10) at src/pipe.cpp:45
#1 0x00007ffff7b8fbc0 in zmq::socket_base_t::connect (this=0x618890, endpoint_uri_=0x400c3c "tcp://localhost:5555")
    at src/socket_base.cpp:969
#2 0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:18

zmq::pipe_t::pipe_t

构造pipe
Thread 1 "hwclient" hit Breakpoint 19, zmq::pipe_t::pipe_t (this=0x621be0, parent_=0x618890, inpipe_=0x619a50, outpipe_=0x61dae0,
    inhwm_=1000, outhwm_=1000, conflate_=false) at src/pipe.cpp:91
91 zmq::pipe_t::pipe_t (object_t *parent_,
(gdb) bt
#0 zmq::pipe_t::pipe_t (this=0x621be0, parent_=0x618890, inpipe_=0x619a50, outpipe_=0x61dae0, inhwm_=1000, outhwm_=1000,
    conflate_=false) at src/**pipe.cpp:91**

#1 0x00007ffff7b79f62 in zmq::pipepair (parents_=parents_@entry=0x7fffffffdb50,
pipes_=pipes_@entry=0x7fffffffdf80,
    hwms_=hwms_@entry=0x7fffffffdb30, conflate_=conflate_@entry=0x7fffffffdb10) at
src/pipe.cpp:67
#2 0x00007ffff7b8fbc0 in zmq::socket_base_t::connect (this=0x618890,
endpoint_uri_=0x400c3c "tcp://localhost:5555")
    at src/socket_base.cpp:969
#3 0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:18

Thread 1 "hwclient" hit Breakpoint 19, zmq::pipe_t::pipe_t (this=0x621cf0, parent_=0x619510,
inpipe_=0x61dae0, outpipe_=0x619a50,
    inhwm_=1000, outhwm_=1000, conflate_=false) at src/pipe.cpp:91
91   zmq::pipe_t::pipe_t (object_t *parent_,
(gdb) bt
#0 zmq::pipe_t::pipe_t (this=0x621cf0, parent_=0x619510, inpipe_=0x61dae0,
outpipe_=0x619a50, inhwm_=1000, outhwm_=1000,
    conflate_=false) at src/pipe.cpp:91
#1 0x00007ffff7b79fad in zmq::pipepair (parents_=parents_@entry=0x7fffffffdb50,
pipes_=pipes_@entry=0x7fffffffdf80,
    hwms_=hwms_@entry=0x7fffffffdb30, conflate_=conflate_@entry=0x7fffffffdb10) at
src/pipe.cpp:70
#2 0x00007ffff7b8fbc0 in zmq::socket_base_t::connect (this=0x618890,
endpoint_uri_=0x400c3c "tcp://localhost:5555")
    at src/socket_base.cpp:969
#3 0x0000000000400ab6 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-
test/hwclient.c:18

主线程和io线程的pipe_t不一样，但其inpipe_和outpipe_是一样的，但是要注意的是在不同的pipe里面两者是相反的。

zmq::session_base_t::attach_pipe 绑定pipe
zmq::session_base_t::pull_msg
zmq::session_base_t::push_msg
zmq::pipe_t::write
zmq::pipe_t::read

关键函数

zmq_ctx_new

zmq_ctx_new 创建上下文的时候到底做了什么事情

ctx_t 是什么

在你的主代码开始处执行一个zmq_ctx_new()、在代码最后执行一个zmq_ctx_term()

zmq_socket

创建的是虚拟socket，并不是真正创建系统io的socket

zmq_connect

如何实现和服务器端的连接

zmq_send

如何实现数据的发送

#0 zmq::req_t::xsend (this=0x618890, msg_=0x7fffffffe380) at src/req.cpp:56

#1 0x00007ffff7b8b860 in zmq::socket_base_t::send (this=this@entry=0x618890, msg_=msg_@entry=0x7fffffffe380,

    flags_=flags_@entry=0) at src/socket_base.cpp:1123

#2 0x00007ffff7bac819 in s_sendmsg (flags_=0, msg_=0x7fffffffe380, s_=0x618890) at src/zmq.cpp:338

#3 zmq_send (s_=<optimized out>, buf_=0x400c6d, len_=<optimized out>, flags_=0) at src/zmq.cpp:370

#4 0x0000000000400af1 in main () at /mnt/hgfs/ubuntu/vip/20200811-ZeroMQ/libzmq-test/hwclient.c:24

如果我们是使用了REQ模型，则主线程的收和发都在 req.cpp，

zmq_recv

如何实现数据的接收

ZeroMQ到底会有多少线程

zmq::object_t::process_command

void zmq::object_t::process_command (command_t &cmd_)