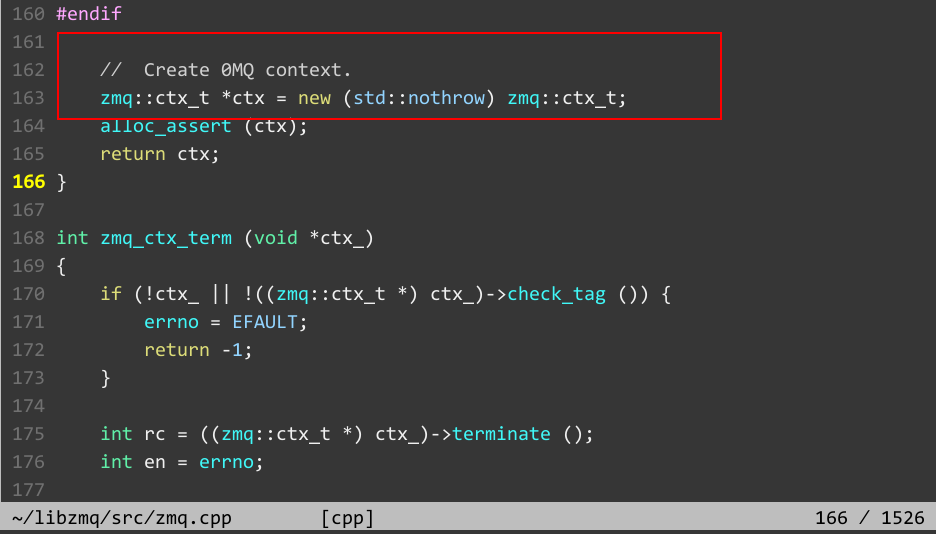
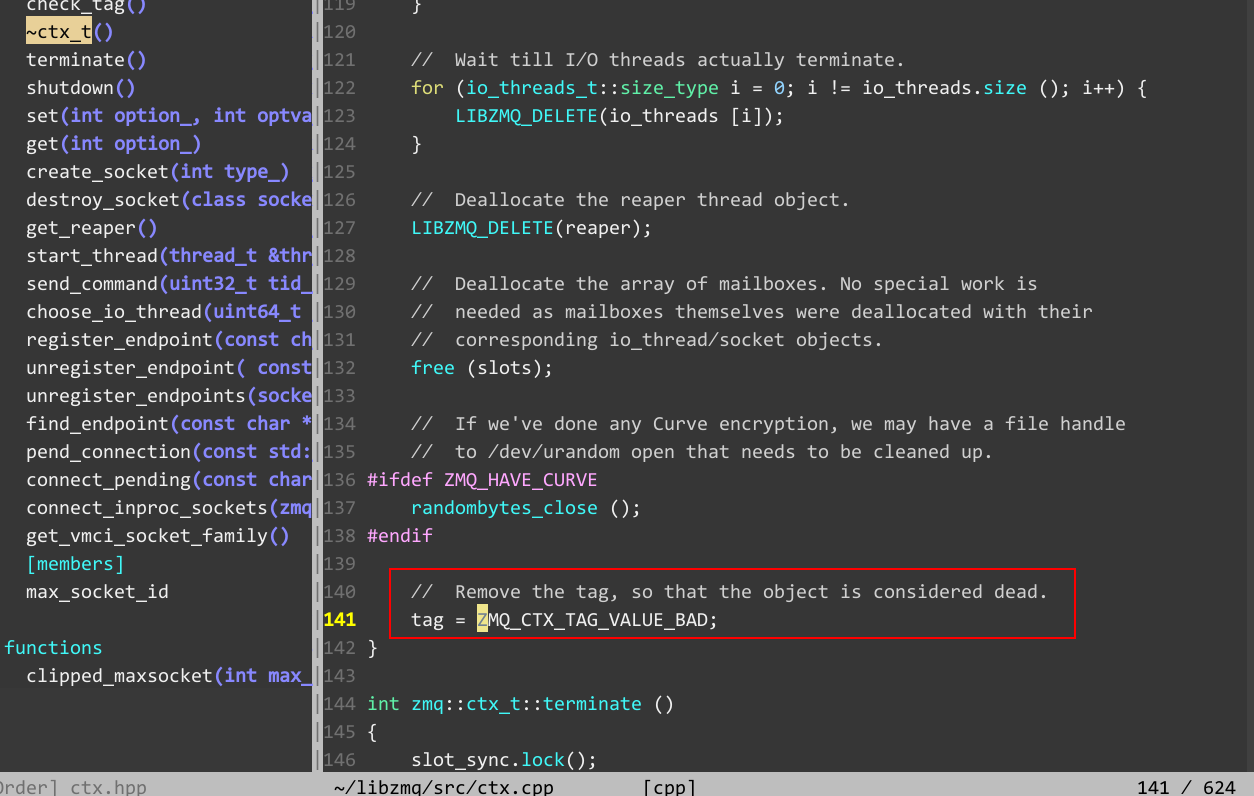
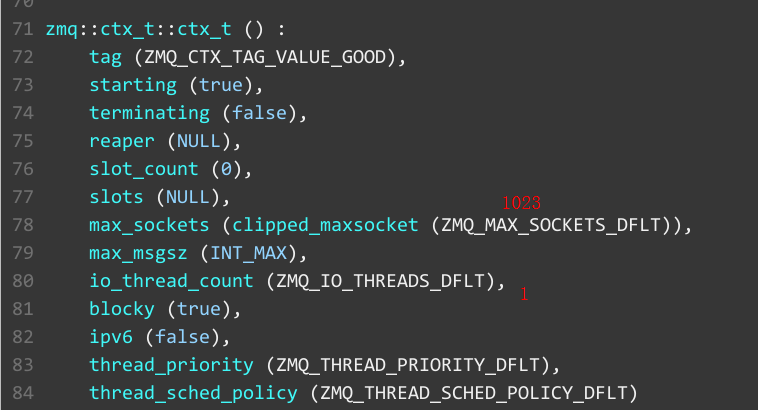
1. 不抛出异常的 new 操作符



1. ctx\_t 类的析构函数中会将 tag 置位为ZMQ\_CTX\_TAG\_VALUE\_BAD

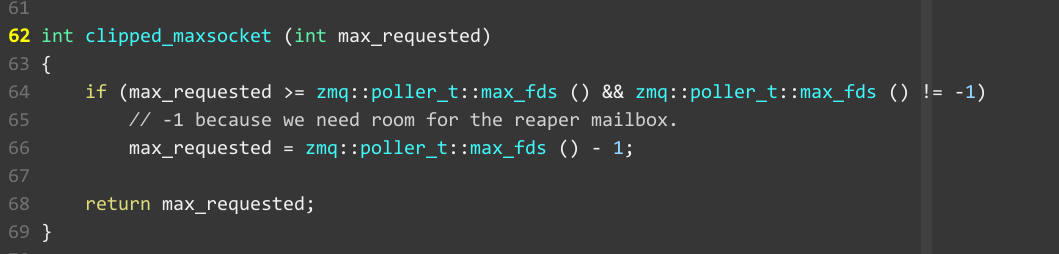


1. ctx\_t 初始化参数



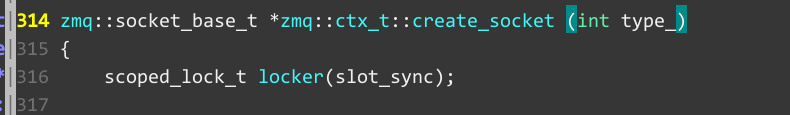
默认 io 线程启动 1 个

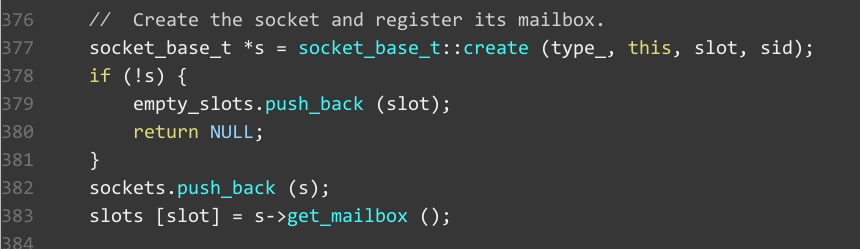
max\_sockets 数目还与系统支持的最大文件描述符有关。



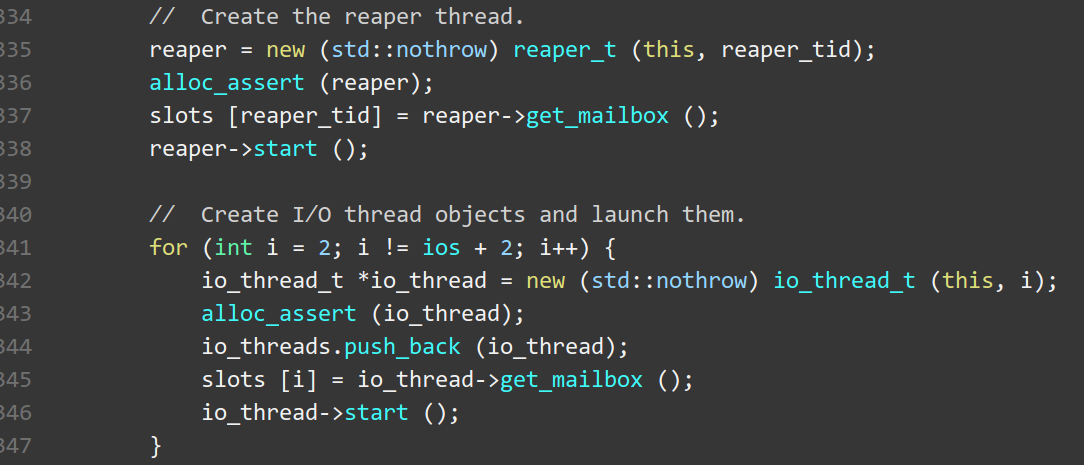
1. zmq\_socket() 创建 socket 过程

实际将调用的是

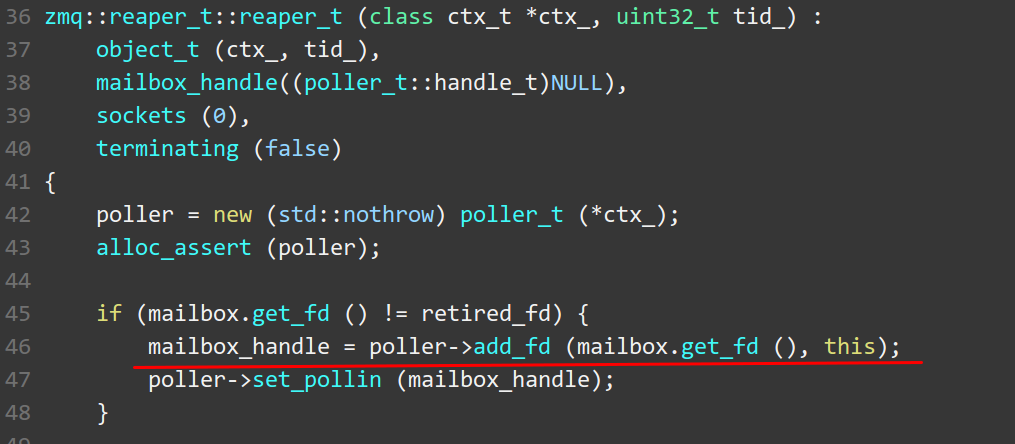




第一次调用会启动两个线程：reaper 和 io

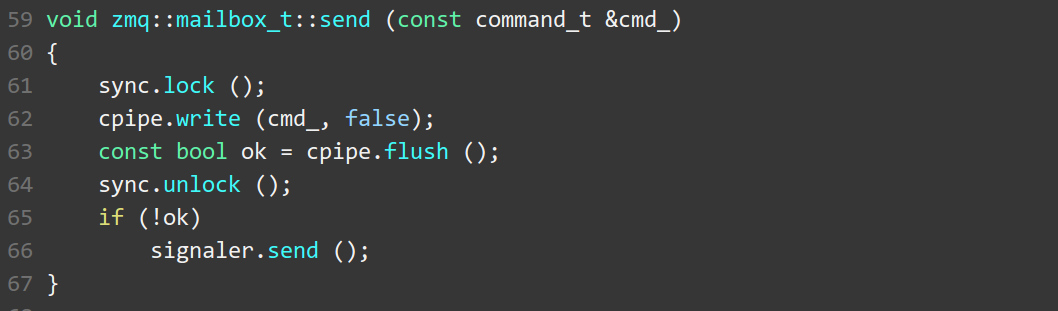


reaper\_t 构造时会创建一个 poller，并将 mailbox 的 fd 加入到 poller 监听列表中，



mailbox 的 fd 是由 eventfd() 创建的，man eventfd 查看详情，写线程通过mailbox 调用 send 发送消息（命令）后，会让 event fd 也发出一个写信号通知读线程。mailbox 被注册到 ctx 中，所有的线程都可以访问，这样就可以方便实现线程间通信。

（zmq的事件回调方式是：需要作为回调类的类都继承 i\_poller\_events 接口，这个类一般会包含一个 fd，这个类就是这个 fd 的回调类，里面实现了各种回调函数。如上面的 reaper\_t 就是 mailbox fd 的回调类。）

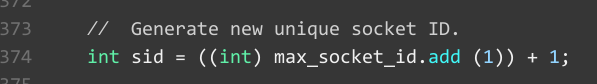


object\_t 类中包含各种消息（命令）的发送和处理函数，是参数进程内（线程间）通信的类必须继承的基类。

mailbox 内的 cpipe 中的 item 类型就是 command，object\_t 中定义了对各个 command 的处理虚函数，子类可以重载对 command 的处理方法。

创建 socket 涉及到下面几点:

创建一个 unique socket ID

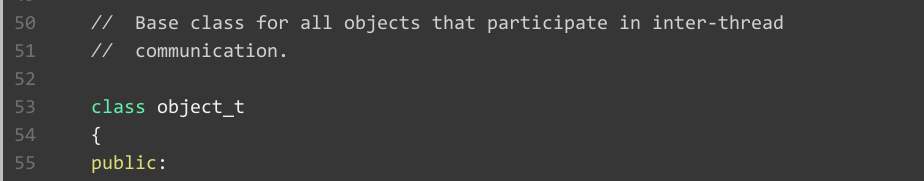


1. 注册 socket 的 mailbox
2. 将 socket 信息添加到 ctx 中（调用 zmq\_ctx\_term 时需要通知 socket）
3. 探究 reaper 线程

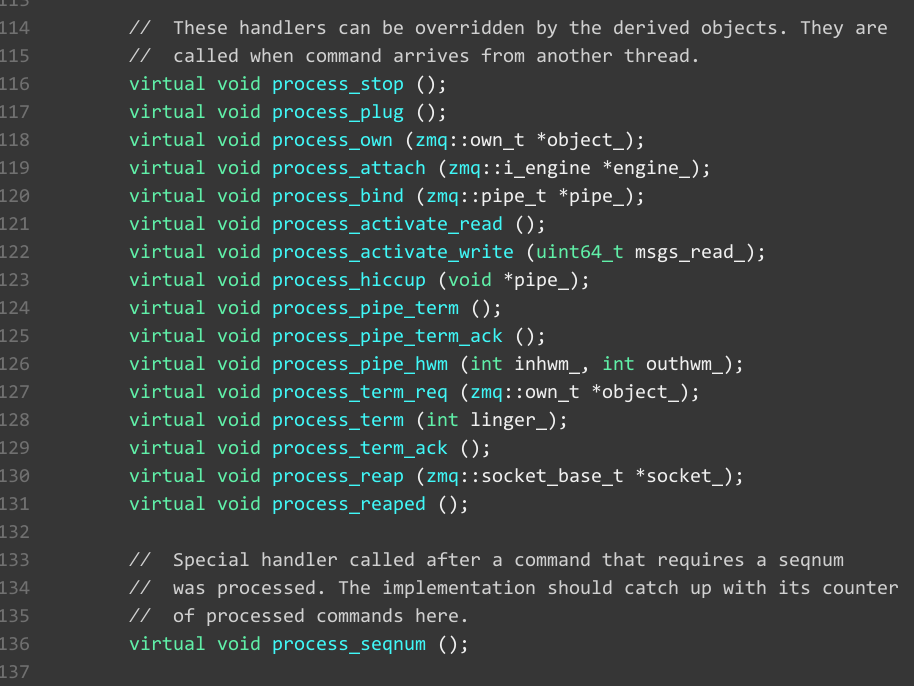
（reaper.hpp/cpp）

继承两个类: object\_t 和 i\_poll\_events

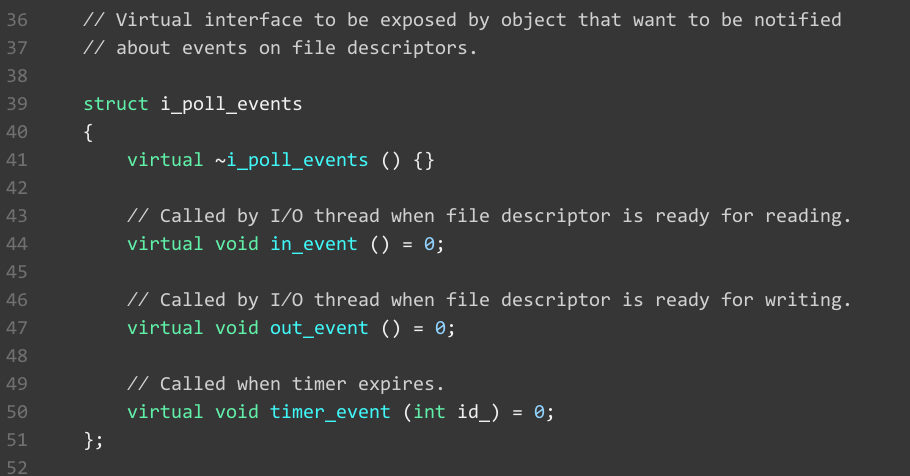
1）object\_t 类



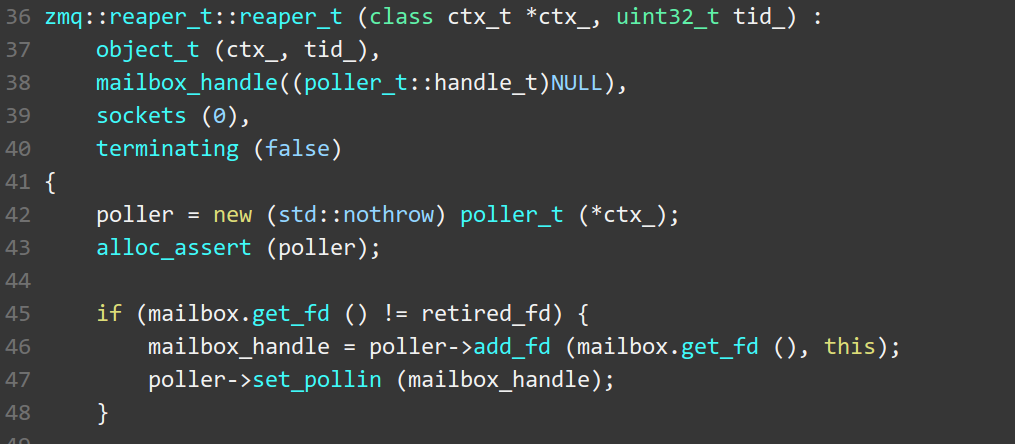
有一组虚函数



2) i\_poll\_events 类



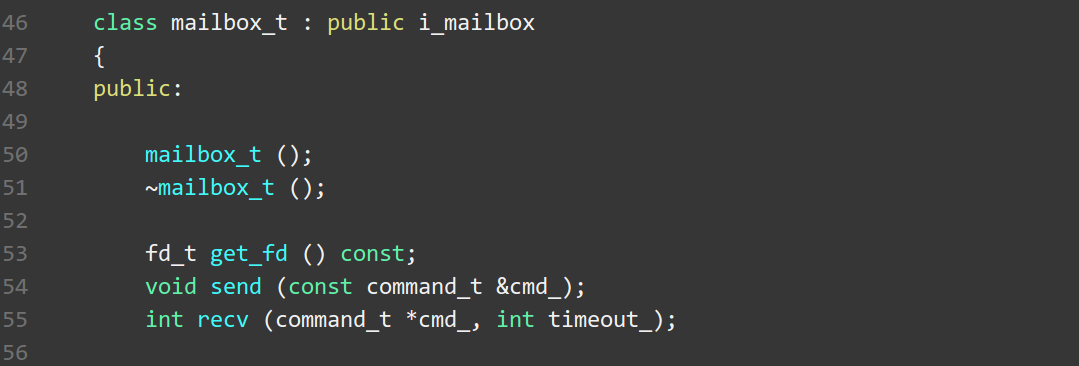
reaper\_t 构造时会将 mailbox 的 fd 添加到 poller 监听列表中



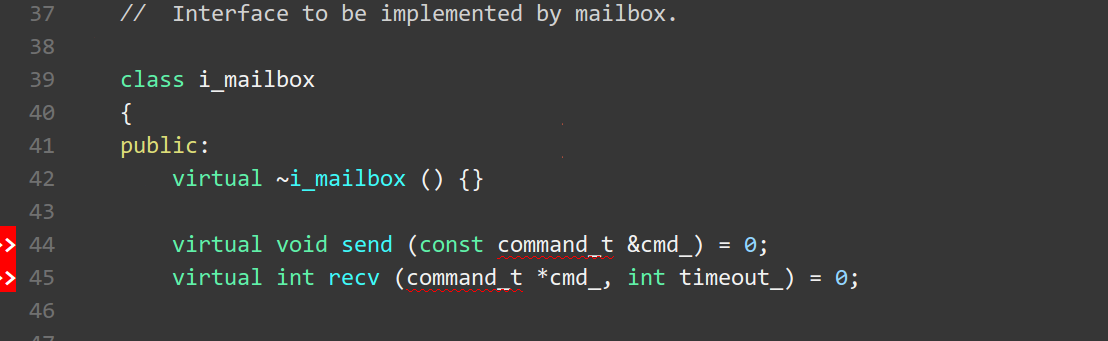
包含 mailbox\_t 类的成员变量

1）mailbox\_t 类

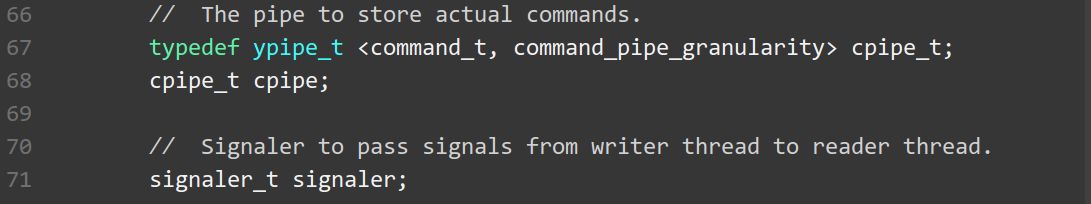
继承自 i\_mailbox\_t 类



i\_mailbox\_t 为接口类，有两个纯虚函数

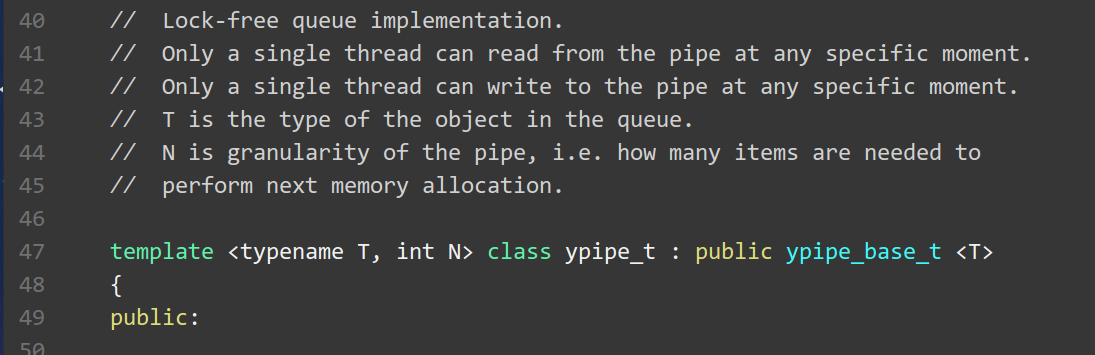


mailbox\_t 类包含有 ypipe\_t 和 signaler 两个类

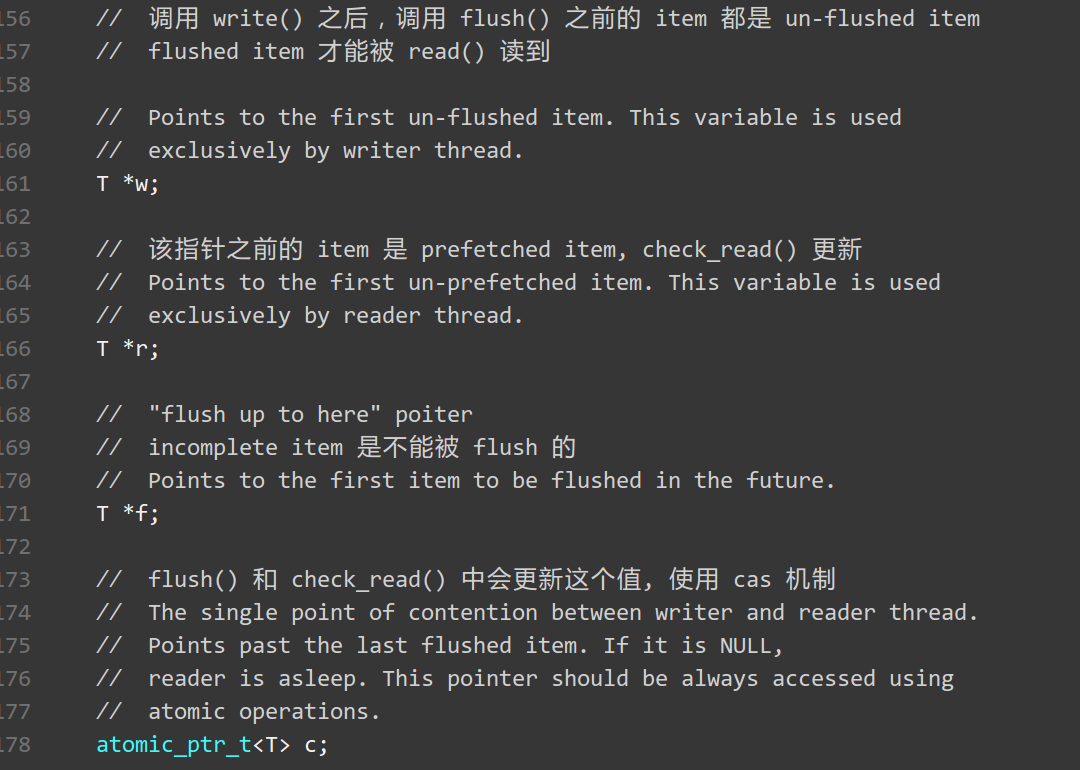


1.1）ypipe\_t 类

无锁队列

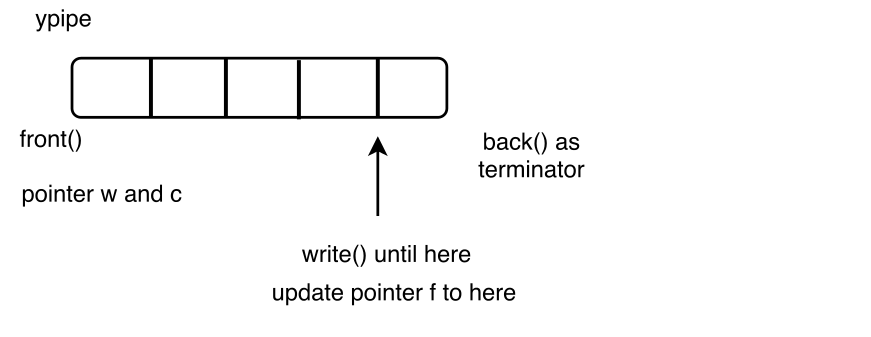


主要有 4 个指针：w, r, f, c

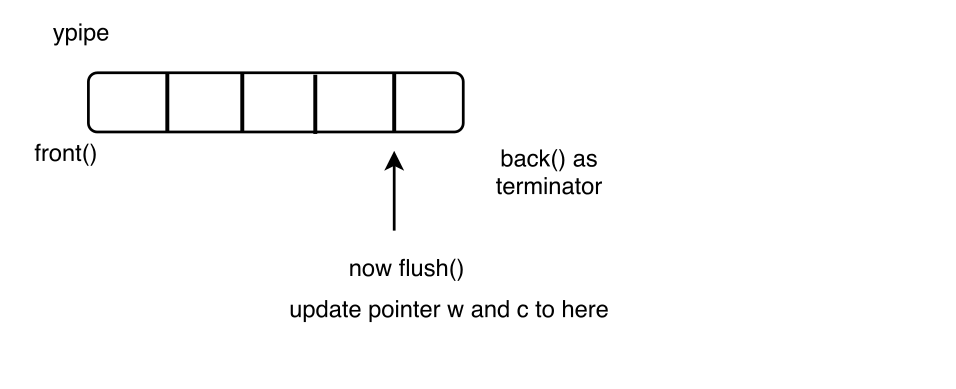


下面展示 ypipe 的工作方式

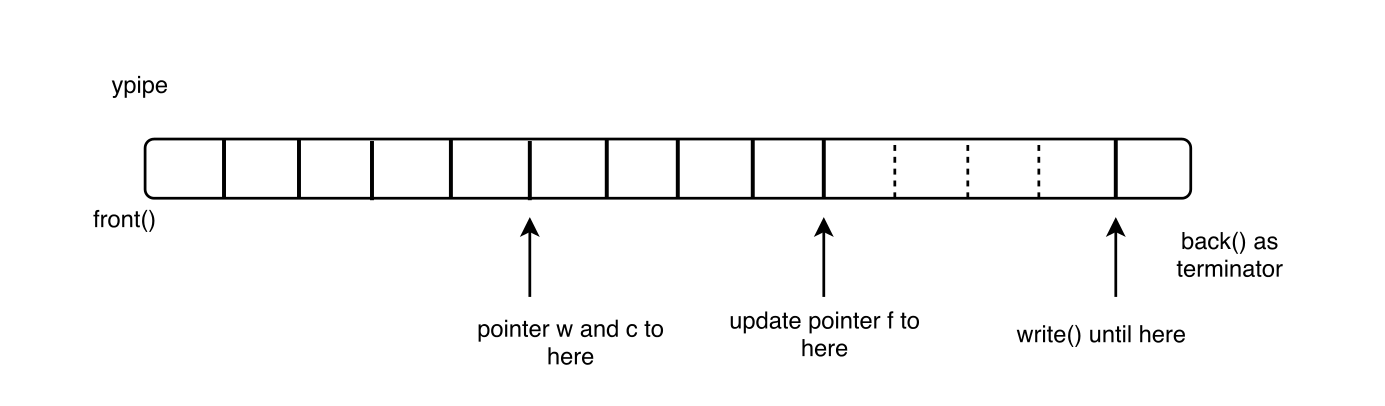
(1) 初始时 4 个指针都指向 front()

(2) 调用 write() 写入4 个元素，此时会更新 指针 f，若 4 个元素都是 complete 类型, 则 f 会指向 back()，即 terminator 元素；指针 w 和 c 不变。如下图：

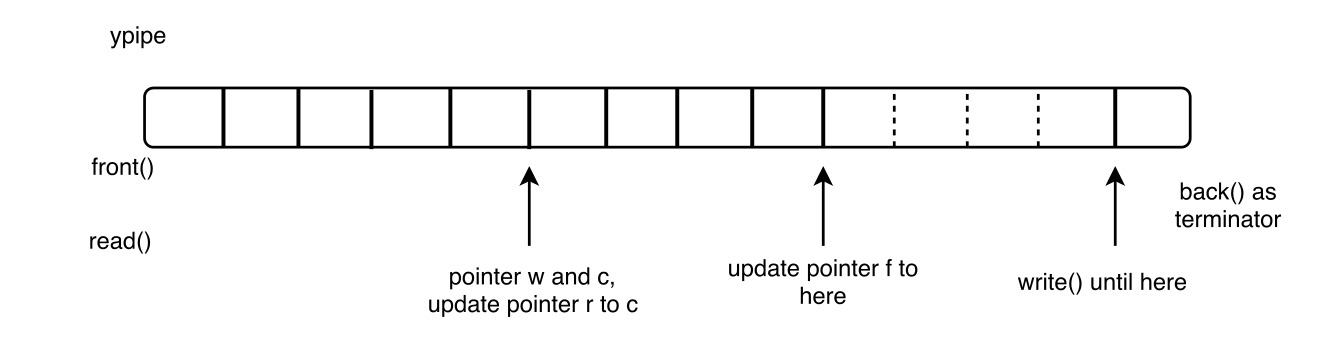
(3) 调用 flush() 刷新，此时指针 w 和 c 会更新为指针 f 的值。（若之前c的值为 null，则flush会返回false，暗示上一次reader没有读到数据，进入了asleep）



(4) 继续调用 write()，但最后一个元素为 incomplete 类型，如下图中虚线框所示，最后一个元素写入了4个部分，并且仍然是 incomplete；此时指针f将指向这个incomplete元素的首部分；



(5) 调用 read(), 更新指针 r 为指针c的值，返回 front() 元素

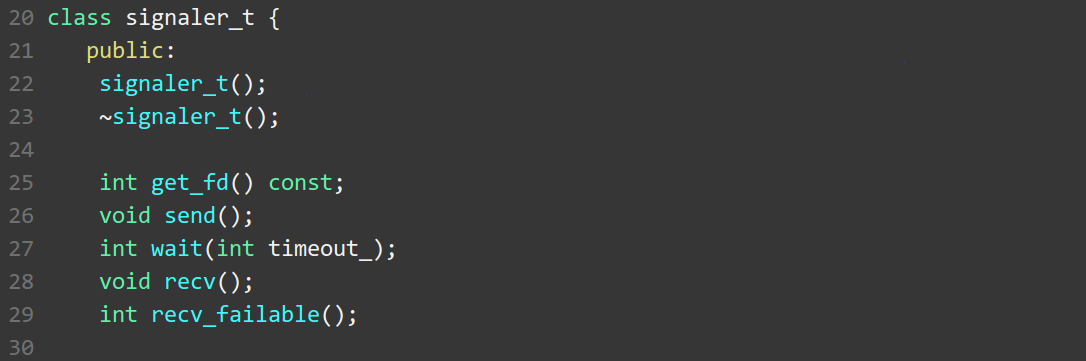


(6) 之后对 read() 的调用将不更新指针 r；直到r 指向 front()，此时又将 r 更新为c值，若 r 还是指向 front()，说明没有元素可读，此时将c的值置为null。

1.2）signaler\_t 类

阅读 man eventfd

封装eventfd()，写线程和读线程间的事件通信 （wait/notify）

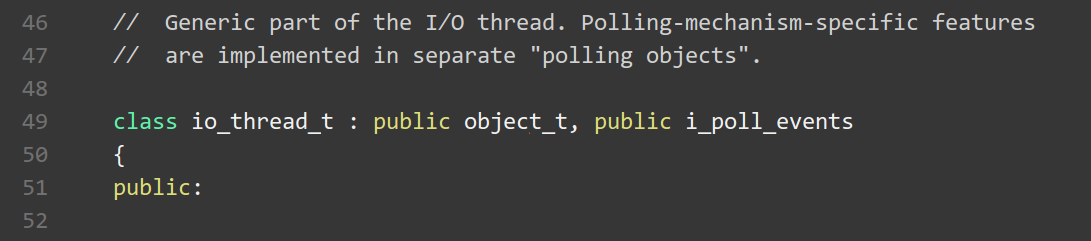


poller\_t 类

封装多路复用 api

reaper\_t 启动的线程执行的就是 poller\_t 的 loop() 方法

1. 探索 io\_thread\_t 类



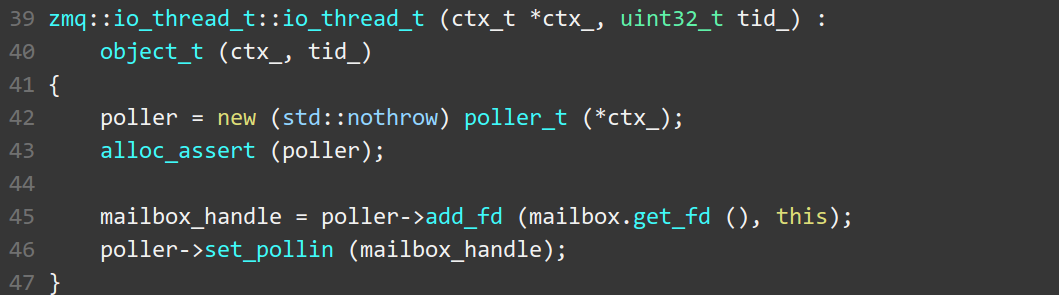
Q: epoll\_wait 为一个独立线程，在另一个线程中 删除监听 fd 会怎样？

A: epoll\_wait 会剔除删除的监听 fd.

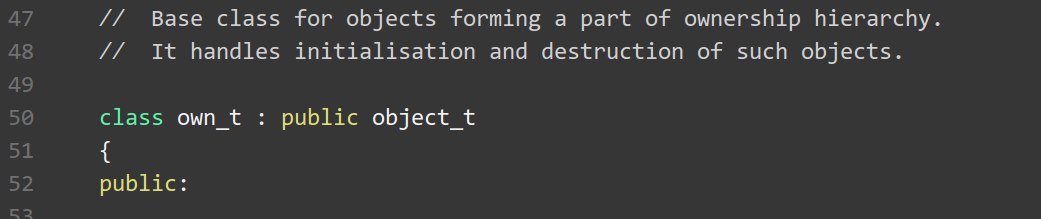
See: <https://stackoverflow.com/questions/7058737/is-epoll-thread-safe>

线程执行的是 poll\_t::loop()

创建时会创建一个poller，并将 mailbox fd 加入到 poller 监听列表



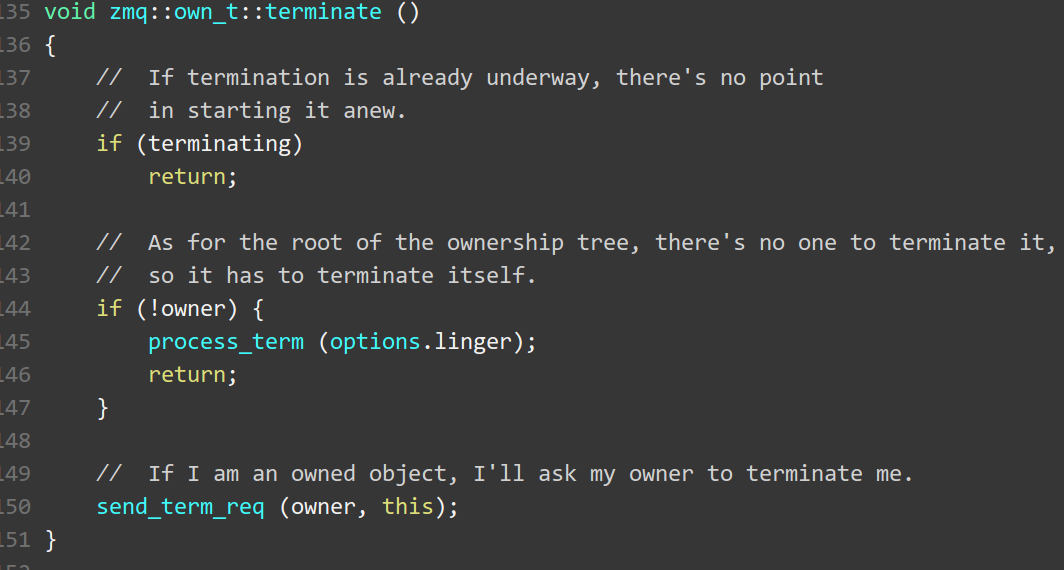
1. own\_t 类



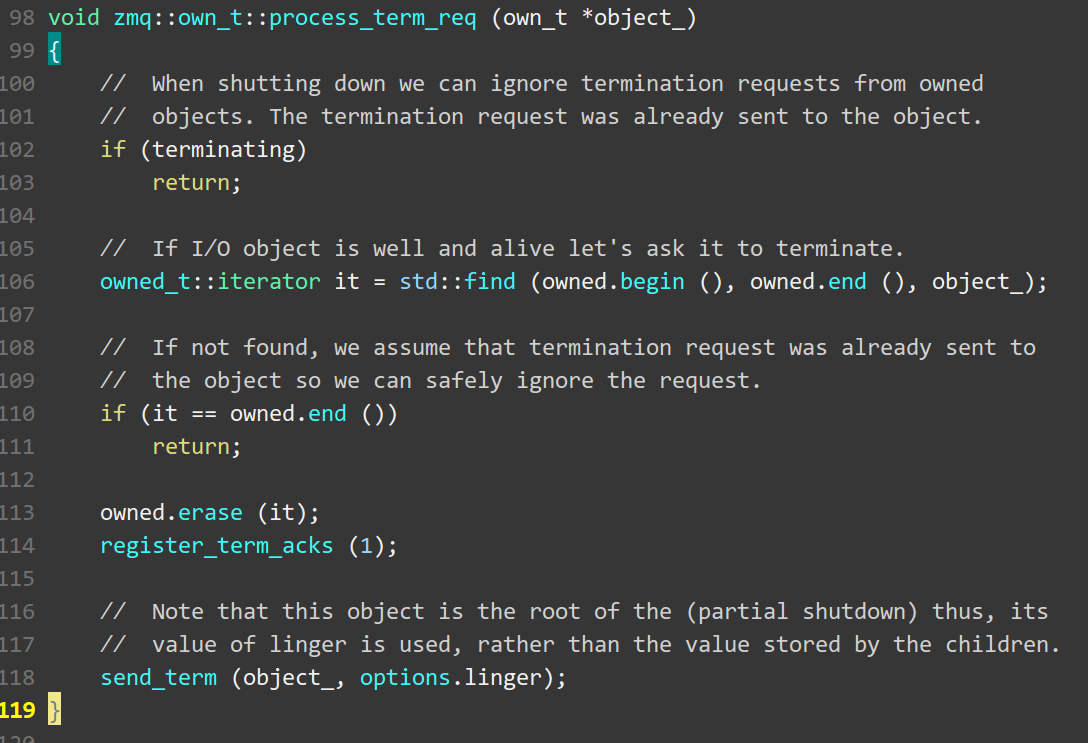
归属关系形成树形结构，节点为线程，节点的终止(terminator)需要其父节点发送 termination 信号。

父节点需要等其所有的子节点都终止后才能终止，父节点发送 term req信号给子节点请求子节点终止，子节点需要发送 term ack 信号给父节点确认其已终止。

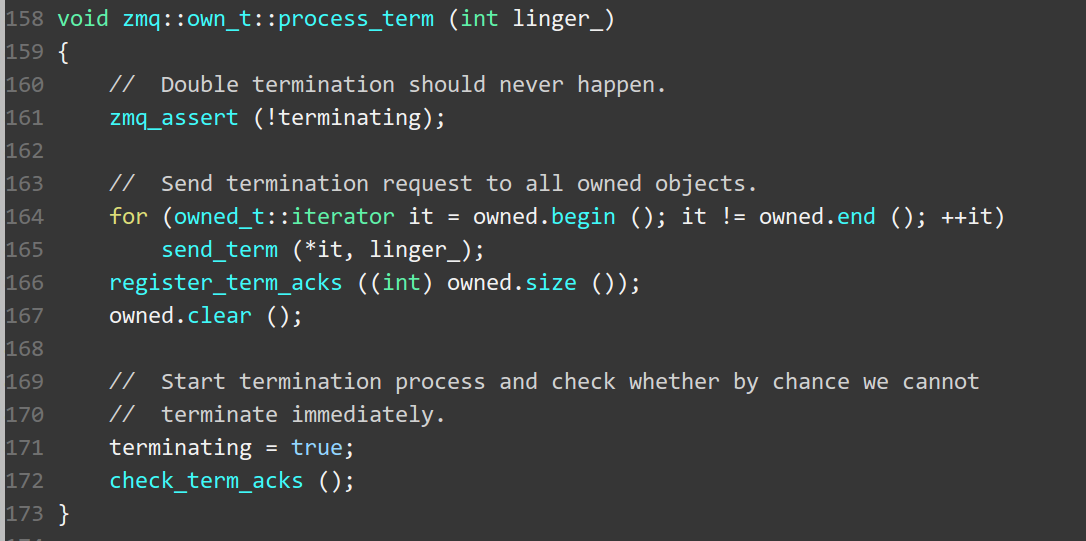
子节点向父节点发送 term req



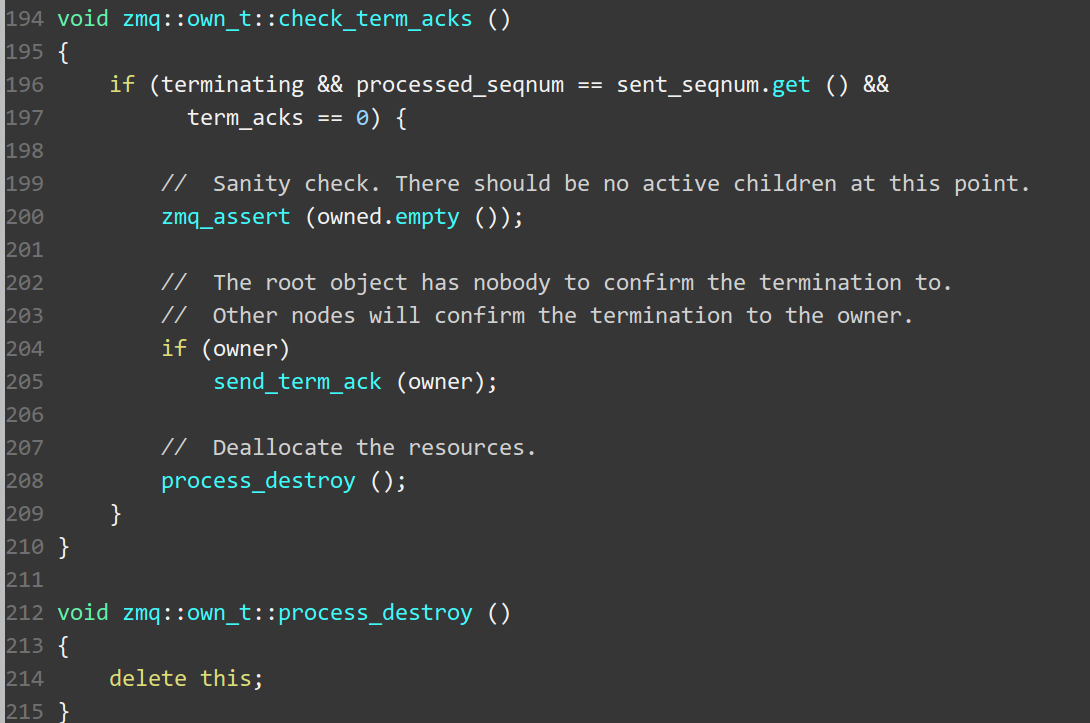
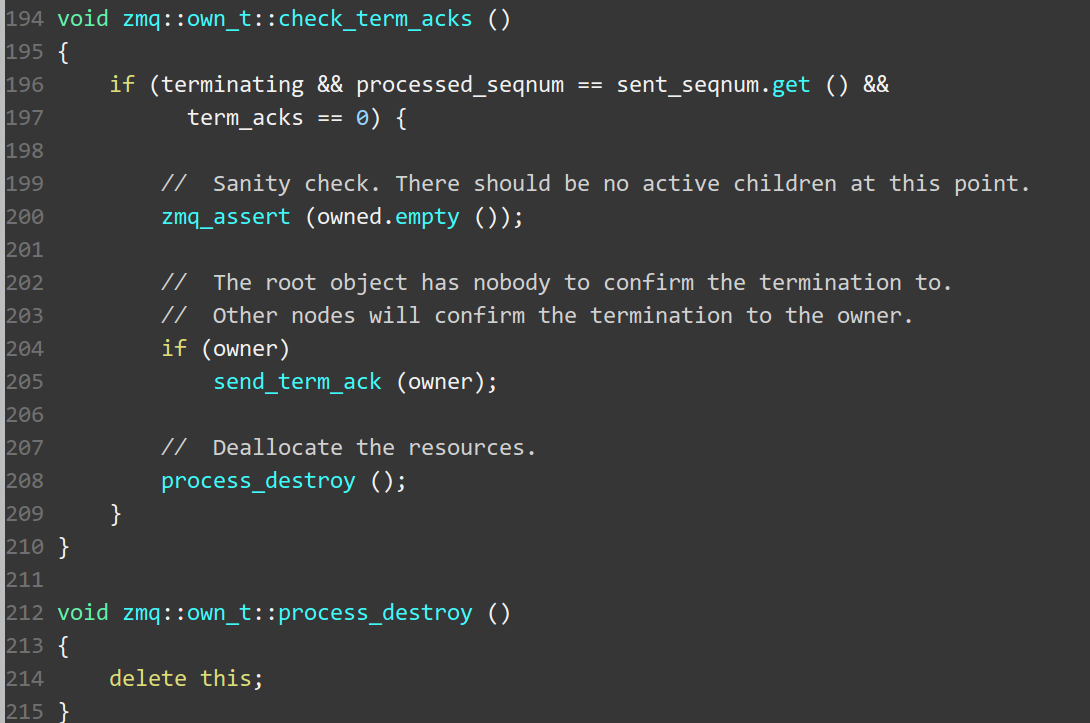
Owner 处理 term req



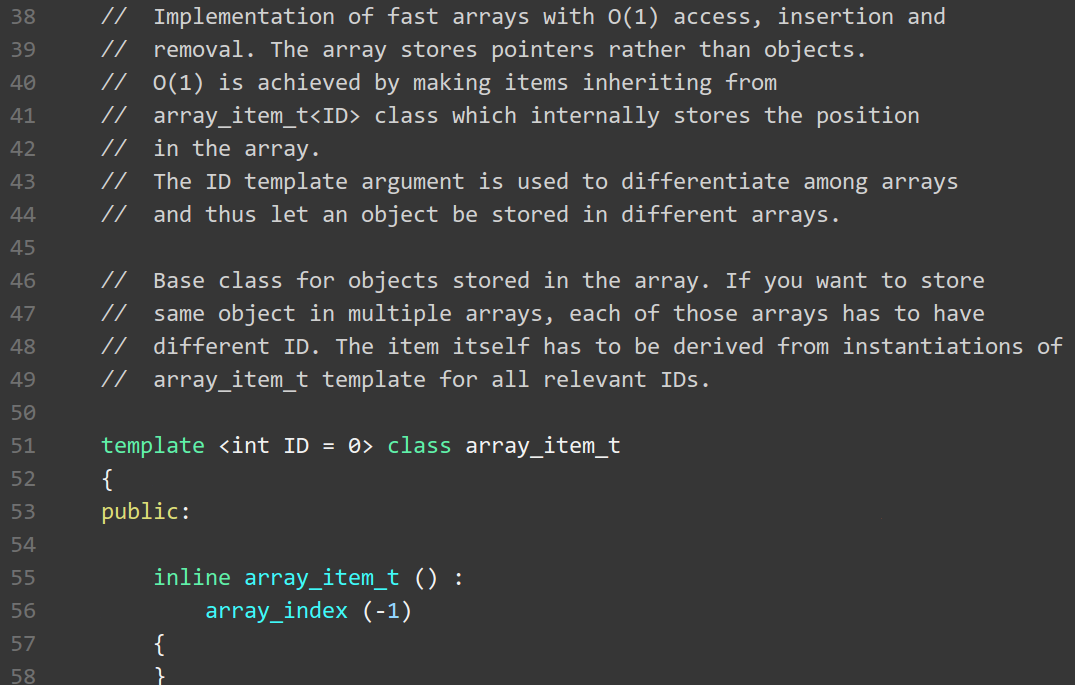
执行 term 命令



父节点收到所有子节点终止信号 ack 后，销毁自身



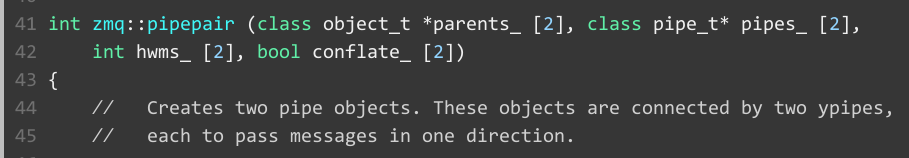
1. array\_item\_t 类



1. pipe\_t 类

底层包含一个读管道和一个写管道 (ypipe)

有一个对端管道 peer (pipe\_t) 类，对端的读写管道和自身正好相反



pipepair() 为两个 object 建立管道连接

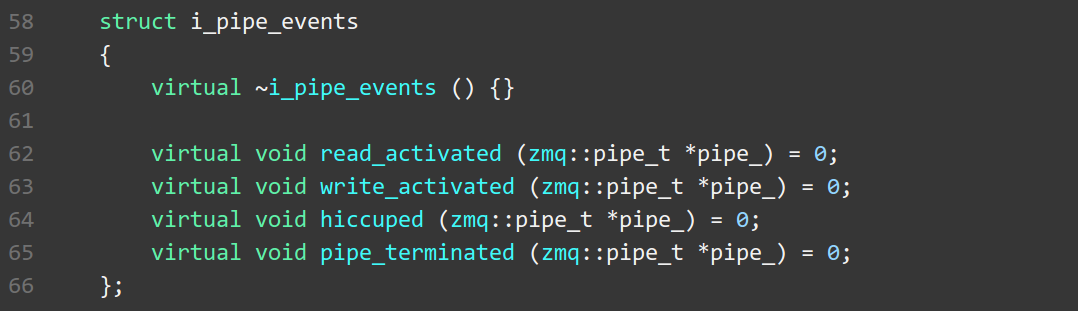
有一个类似 TCP 的状态转换机

TODO 画出状态机

1. i\_pipe\_events 类

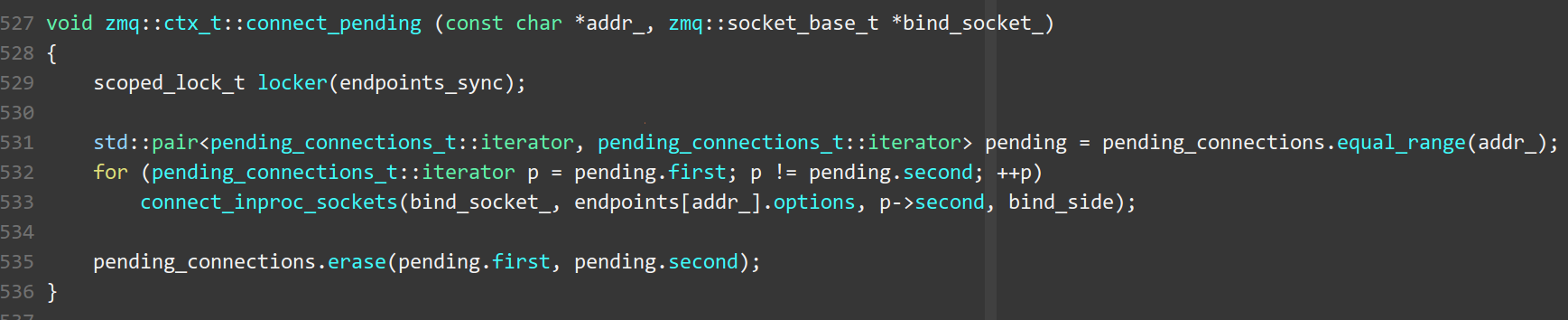
接口类

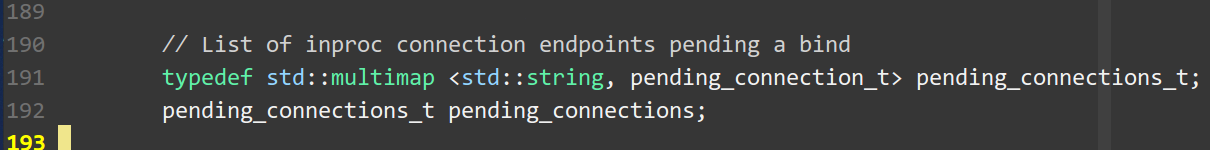
pipe\_t 类产生事件的处理函数接口

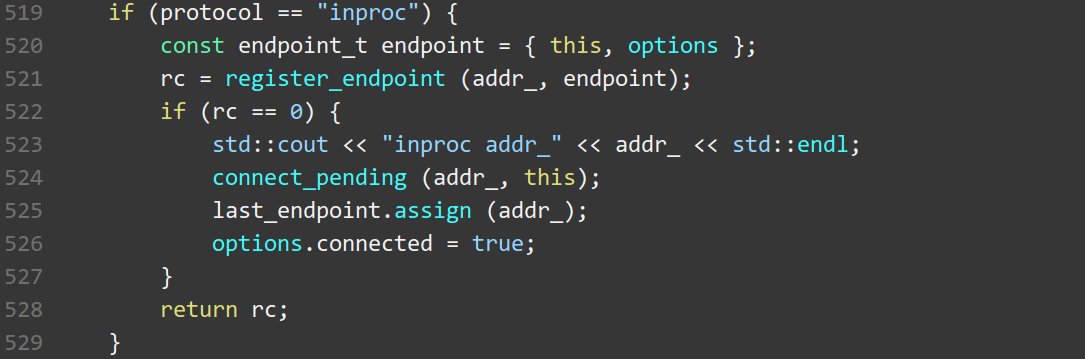


1. socket\_base\_t 类

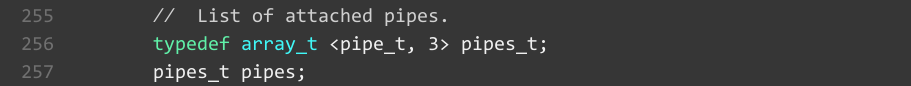
1）inproc 通信方式

connect 和 bind 的顺序无所谓，实现方式：将 pending connect 添加到容器中，当bind调用时，查找该容器，将对应的 pending connect 连接上

进程内的通信是通过 pipe\_t 类来实现的，connect socket 有一个 connect\_pipe，bind socket 端为 bind\_pipe



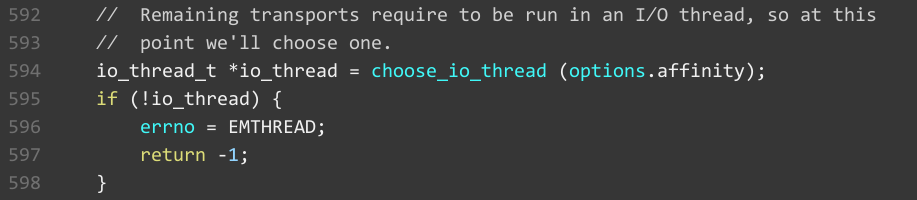
已建立的连接将放入 pipes 容器中

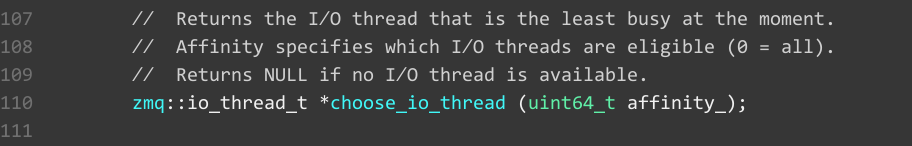


Pipes 容器中的 pipe 同样有来自于与 session\_base 通信的pipe，session\_base 是可以处理 tcp 连接的，所以 zmq\_socket 是通过 pipe 管理与其他 tcp socket来通信的

2）TCP通信方式

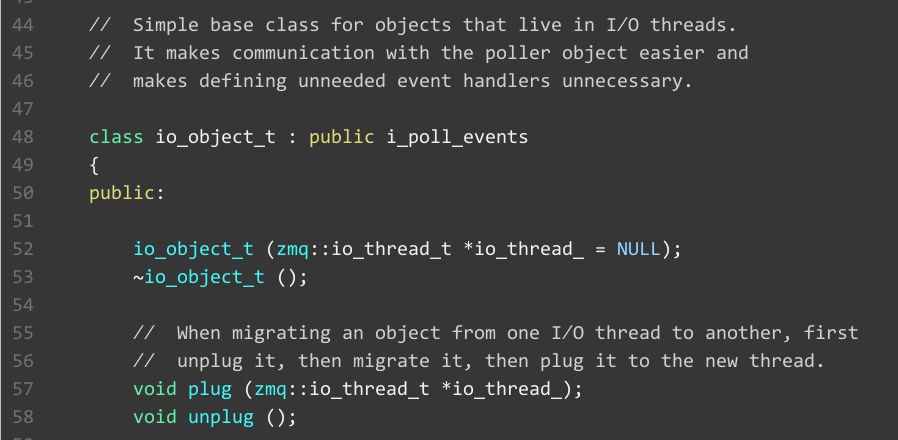
运行在一个 IO 线程中



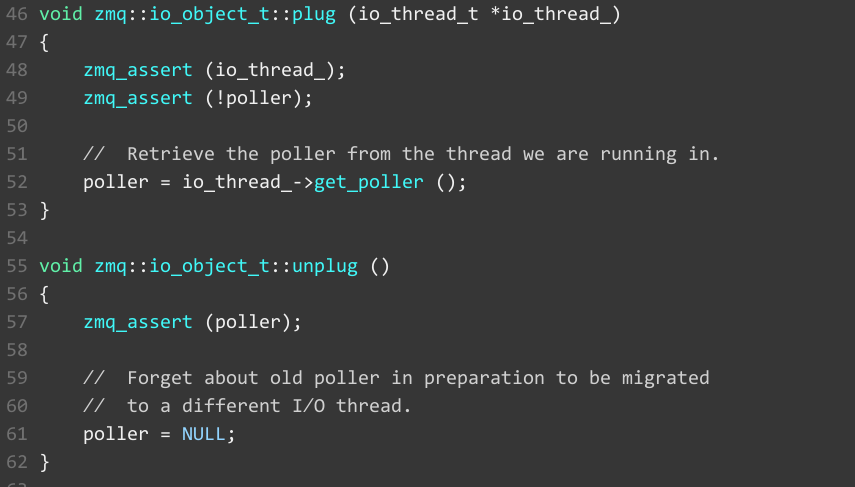


在 IO 线程中运行的含义实际上是将 socket fd 添加到 IO 线程的 epoll 监听列表中

1. io\_object\_t 类

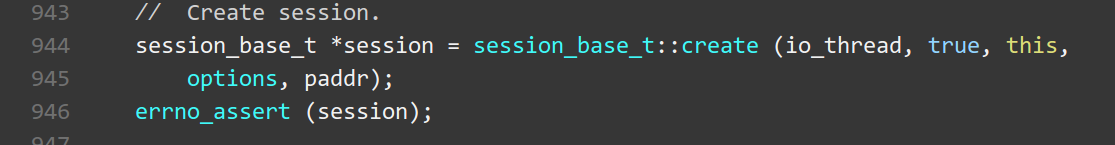


plug / unplug 实际是获取与释放 io\_thread\_t 中的 poller 指针



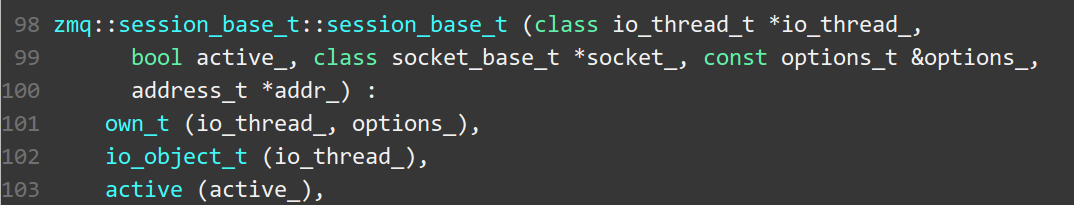
TCP connect()

调用 connect (tcp 协议)时会创建一个对应的 session\_base\_t，并用 bi-pipe 将两者连接起来，其中 socket 可以对应有多个 session 的pipe，session 只有一个对应 socket 的pipe。



inc\_seqnum 的含义是此object 的 owner 给此 object 发送命令的序号，owner在给object发送命令前都

创建 session 时，会传入一个 io\_thread，则 session 的 tid 将会是这个 io\_thread 的，所以 session 的 mailbox 实际上使用的是 io\_thread 的，对这个 session 发送命令会被 io\_thread 的 poller 所监听到，然后调用 process\_commands 来处理。



会先将 object 的 inc\_seqnum 值加一（原子），object 在没有收到序号 inc\_seqnum 的命令前不能关闭。

io\_object\_t 用 io\_thread\_ 初始化时会将其plug 到 io\_thread\_ 上，即获取 io\_thread\_的poller的指针。

lauch\_child() 启动 session时会发送 plug 信号，poler将会响应，于是在 io\_thread 线程中将会处理该命令。

session\_base\_t 对 plug 信号的处理是 start\_connecting()，start\_connecting() 将会 创建 tcp\_connecter 类，tcp\_connecter 通过 lanch\_child 被启动，同样将在一个 io\_thread 中运行，使用的是 timer 定时事件来 reconnect

ZMQ\_REQ 类型的 zmq\_send 只是把数据写到 pipe 中，并不会阻塞。zmq\_recv 会阻塞，等待数据到达，阻塞在 process\_command 的 recv command 上面

1. tcp\_listener\_t类
2. tcp\_conneter 类 ( 非阻塞connect学习的好地方)

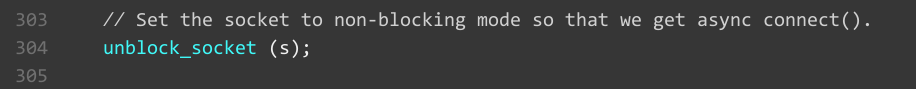
做实际的 connect 操作，会用 socket() 新创建一个 socket

Connect 失败会将添加 reconnet timer 定时重连， 连接成功会未 socket 创建 stream\_engine, 该 stream\_engine 会attach 到一个 session\_base, seeion\_base 会与一个 zmq socket 绑定

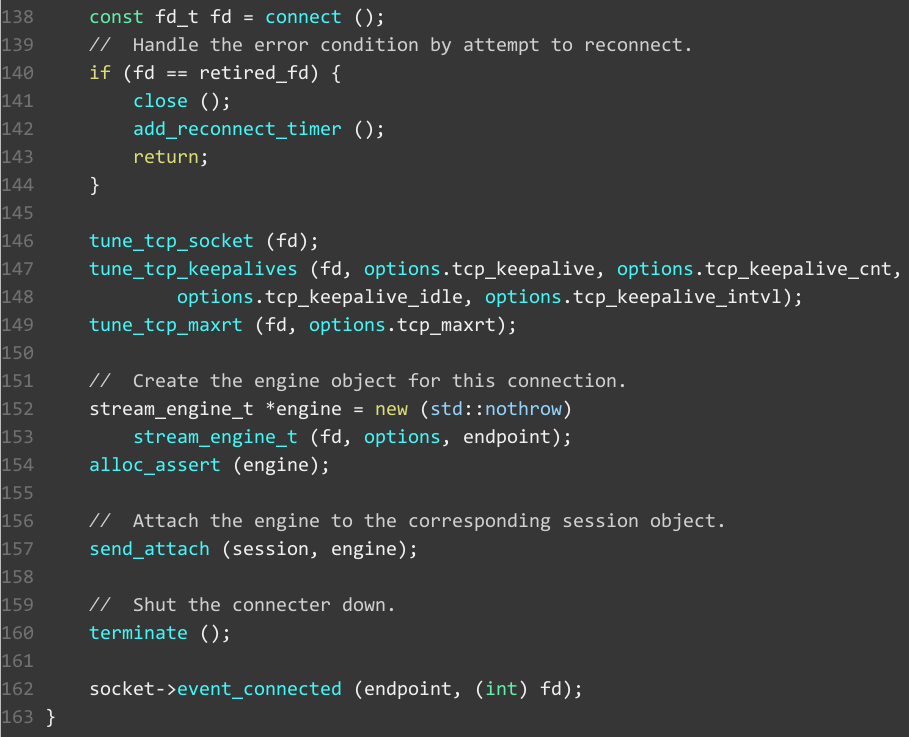
连接成功后会调用 terminate 关闭这个 object

这里使用的是非阻塞式的 connect 方法：1. 先检测是否已经connect完成，2. 没有完成将 connect socket 添加到 poller，设置 connect timer，connect timer 定时到了 将 connect socket 从poller 中删除 3. Poller 提示 connect socket 可写时，需要先取消 connect timer，然后检查是否出错，出错则添加 reconnect timer

重连时间不是每次都固定的，有一个算法来计算

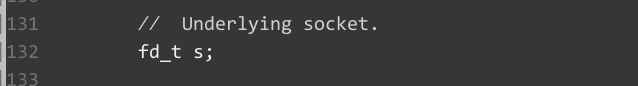


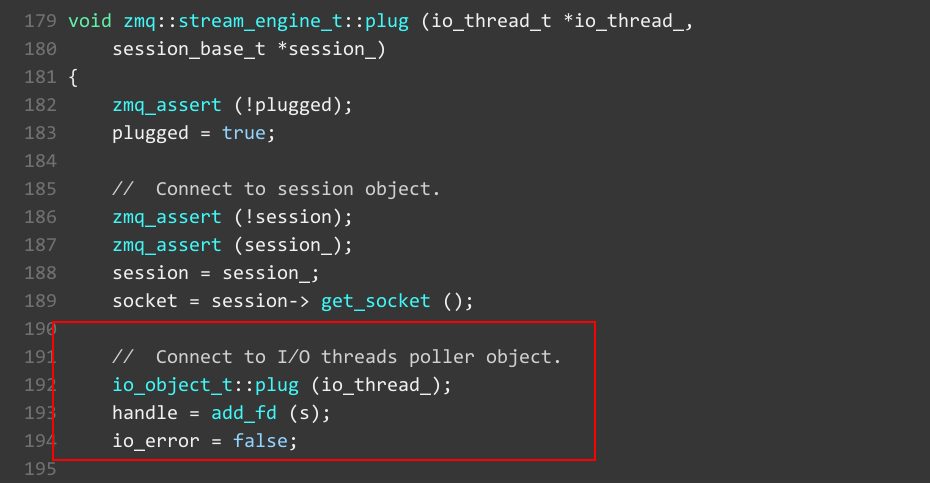
可以学习如何处理非阻塞式 connect() 的错误处理



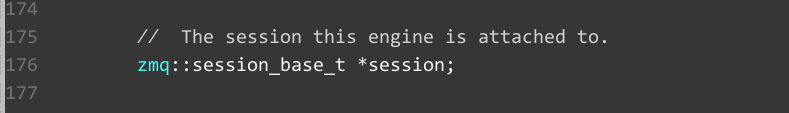
1. stream\_engine 类

管理一个 linux socket



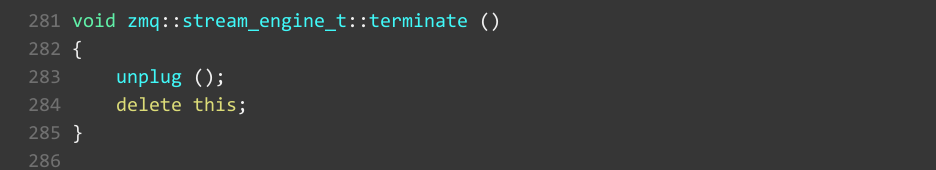
1. 

一个 engine 与一个 session 绑定，plug() 时绑定



**(Epoll 监听的fd是可以动态删除的，即在 epoll\_walit() 阻塞时，在另一个线程中可以执行 epoll\_ctl 删除某个监听 fd )**

简单的 terminate 函数



收到数据后，会将数据发送给 session，然后 session调用 flush。下图是 in\_event()

