首先使用zmq需要安装；zmq是运行在os的；zmq是在Linux进程通信的基本方式（内存共享，内核媒介传递等等）之上进一步封装的库，是需要安装的第三方库，编译的时候需要include和链接。不同于dbus有好多个守护进程的协助，zmq做的只是链接它的库，至于底层实现不用管。

自己写的代码都是作为一个进程运行在操作系统里，不是运行在裸机上的！这个意识一定要建立起来：代码是以进程的形式运行在OS上面，受到OS的管理和调度 & OS上面运行着各种各样的进程（区别于写单片机代码时的思维）。之前理解的：写好代码，编译成可执行文件，然后load进来，在CPU上跑，在这段时间内接管CPU；这是错的，程序是运行在操作系统之上的，受到OS的调度和管理（赶紧学完Linux高级编程和Linux驱动开发系列视频）。

The ØMQ lightweight messaging kernel is a library which extends the standard socket interfaces with features traditionally provided by specialised messaging middleware products. ØMQ sockets provide an abstraction of asynchronous message queues, multiple messaging patterns, message filtering (subscriptions), seamless access to multiple transport protocols and more.

context是一种结构（暂时将其想为没有函数指针的结构），结构变量实例便有内存区，然后这里可涉及到线程共享，多个线程依据同一个context，执行zmq函数。context是线程安全的，它可以被多个线程共享而不需要去lock和unlock。而socket是非线程安全的，最好是线程间各自独立创建并使用自己的socket。

ØMQ applications always start by creating a context, and then using that for creating sockets. In C, it’s the zmq\_ctx\_new() call. You should create and use exactly one context in your process. Technically, the context is the container for all sockets in a single process, and acts as the transport for inproc sockets, which are the fastest way to connect threads in one process. If at runtime a process has two contexts, these are like separate ØMQ instances. If that’s explicitly what you want, OK, but otherwise remember:

Do one zmq\_ctx\_new() at the start of your main line code, and one zmq\_ctx\_destroy() at the end.

If you’re using the fork() system call, each process needs its own context. If you do zmq\_ctx\_new() in the main process before calling fork(), the child processes get their own contexts. In general, you want to do the interesting stuff in the child processes and just manage these from the parent process.

**multithreaded ØMQ application：**

First, do not try to use the same socket from multiple threads. Please don’t explain why you think this would be excellent fun, just please don’t do it. Next, you need to shut down each socket that has ongoing requests. The proper way is to set a low LINGER value (1 second), and then close the socket. If your language binding doesn’t do this for you automatically when you destroy a context, I’d suggest sending a patch.

Finally, destroy the context. This will cause any blocking receives or polls or sends in attached threads (i.e., which share the same context) to return with an error. Catch that error, and then set linger on, and close sockets in that thread, and exit. Do not destroy the same context twice. The zmq\_ctx\_destroy in the main thread will block until all sockets it knows about are safely closed.

It has a subversive effect on how you develop network-capable applications. Superficially, it’s a socket-inspired API on which you do zmq\_recv() and zmq\_send(). But message processing rapidly becomes the central loop, and your application soon breaks down into a set of message processing tasks. It is elegant and natural. And it scales: each of these tasks maps to a node, and the nodes talk to each other across arbitrary transports. Two nodes in one process (node is a thread), two nodes on one box (node is a process), or two boxes on one network (node is a box)— it’ s all the same, with no application code changes.

Traditional network programming is built on the general assumption that one socket talks to one connection, one peer. There are multicast protocols, but these are exotic. When we assume “one socket = one connection”, we scale our architectures in certain ways. We create threads of logic where each thread work with one socket, one peer. We place intelligence and state in these threads.

In the ØMQ universe, sockets are doorways to fast little background communications engines that manage a whole set of connections automagically for you. You can’t see, work with, open, close, or attach state to these connections. Whether you use blocking send or receive, or poll, all you can talk to is the socket, not the connections it manages for you. The connections are private and invisible, and this is the key to ØMQ’s scalability.

This is because your code, talking to a socket, can then handle any number of connections across whatever network protocols are around, without change. A messaging pattern sitting in ØMQ scales more cheaply than a messaging pattern sitting in your application code.

So the general assumption no longer applies. As you read the code examples, your brain will try to map them to what you know. You will read “socket” and think “ah, that represents a connection to another node”. That is wrong. You will read “thread” and your brain will again think, “ah, a thread represents a connection to another node”, and again your brain will be wrong.

If you’re reading this Guide for the first time, realize that until you actually write ØMQ code for a day or two (and maybe three or four days), you may feel confused, especially by how simple ØMQ makes things for you, and you may try to impose that general assumption on ØMQ, and it won’t work. And then you will experience your moment of enlightenment and trust, that zap-pow-kaboom satori paradigm-shift moment when it all becomes clear.

###### zmq的socket

消息队列？bind是等待别人连接，connect是主动连接别人。

To create a connection between two nodes, you use zmq\_bind() in one node and zmq\_connect() in the other. As a general rule of thumb, the node that does zmq\_bind() is a “server”, sitting on a well-known network address, and the node which does zmq\_connect() is a “client”, with unknown or arbitrary network addresses. Thus we say that we “bind a socket to an endpoint” and “connect a socket to an endpoint”, the endpoint being that well-known network address.

ØMQ connections are somewhat different from classic TCP connections. The main notable differences are:

• They go across an arbitrary transport (inproc, ipc, tcp, pgm, or epgm). See zmq\_inproc(), zmq\_ipc(), zmq\_tcp(), zmq\_pgm(), and zmq\_epgm().

• One socket may have many outgoing and many incoming connections.

• There is no zmq\_accept() method. When a socket is bound to an endpoint it automatically starts accepting connections.

• The network connection itself happens in the background, and ØMQ will automatically reconnect if the network connection is broken (e.g., if the peer disappears and then comes back).

• Your application code cannot work with these connections directly; they are encapsulated under the socket.

socket背后的queuing system可认为是由一个线程维护的（事实上也应该如此），zmq所有的信息交互都是通过socket实现，而不能直接操作（比如断线重连，不在线也可以send）。

#### zmq简介

以它自己封装的socket的形式，完成进程间，线程间，tcp以及多播（multicast）等通讯；它支持多种通信模型：多对多、一对多、一对一（client\_&\_server）。

“可以通过zmq写分布式软件，实现软件之间的协作”。

Many architectures follow some kind of client/server model, where the server is the component that is most static, and the clients are the components that are most dynamic, i.e., they come and go the most. There are sometimes issues of addressing: servers will be visible to clients, but not necessarily vice versa. So mostly it’s obvious which node should be doing zmq\_bind() (the server) and which should be doing zmq\_connect() (the client). It also depends on the kind of sockets you’re using, with some exceptions for unusual network architectures. We’ll look at socket types later.

Now, imagine we start the client before we start the server. In traditional networking, we get a big red Fail flag. But ØMQ lets us start and stop pieces arbitrarily. As soon as the client node does zmq\_connect(), the connection exists and that node can start to write messages to the socket. At some stage (hopefully before messages queue up so much that they start to get discarded, or the client blocks), the server comes alive, does a zmq\_bind(), and ØMQ starts to deliver messages.

A server node can bind to many endpoints (that is, a combination of protocol and address) and it can do this using a single socket. This means it will accept connections across different transports:

zmq\_bind (socket, "tcp://\*:5555");

zmq\_bind (socket, "tcp://\*:9999");

zmq\_bind (socket, "inproc://somename");

With most transports, you cannot bind to the same endpoint twice, unlike for example in UDP. The ipc transport does, however, let one process bind to an endpoint already used by a first process. It’s meant to allow a process to recover after a crash.

#### 学习目标

**两种通讯方式**：zmq\_tcp和zmq\_ipc；前者是tcp方式通讯（要符合相关的网络通讯要求，比如路由转发规则，不同网段之间的交流），后者是进程之间的交流。

**两种通讯模型**：一对一（client\_&\_server）和一对多（publisher\_&\_subscriber）。

以上两种通讯模型可以通过其中一种通讯方式实现，这样能有4种组合。

“**理解核心概念，严格遵循官方架构（不钻牛角尖），并对各个架构列出一个reference（学习文档以便将来在实际中参考并使用）**”

###### 需要了解的函数

zmq\_ctx\_new / zmq\_ctx\_destroy

zmq\_socket / zmq\_close

zmq\_setsockopt

zmq\_getsockopt

zmq\_bind

zmq\_connect

zmq\_send

zmq\_recv

###### zmq\_socket

void \*zmq\_socket(void \*context, int type);

Generally speaking, conventional sockets present a synchronous interface to either connection-oriented reliable byte streams (SOCK\_STREAM), or connection-less unreliable datagrams (SOCK\_DGRAM). In comparison, ØMQ sockets present an abstraction of an asynchronous message queue, with the exact queueing semantics depending on the socket type in use.

zmq的socket可以通过zmq\_connect同时连接多个server，也可以通过zmq\_bind绑定多个endpoint，即多对多的通信模型，但是本学习笔记不涉及这方面，只要求熟练掌握3种模型（编程时根据这3种模型严格的bind和connect，不要纠结其他场景）。

###### zmq\_connect

（阻塞）

###### zmq\_send

int zmq\_send(void \*socket, void \*buf, size\_t len, int flags);

功能是把由buf指针和len长度reference的字节加入到socket的消息队列当中（如何判断消息是否发送成功？是接收到了然后反馈回来就发送成功了？这个是由下面的协议控制的吗？或者说是发送完就不管了？应该是看通信模型吧，具体情况具体分析，比如说tcp方式的publish\_&\_subscribe模型应该是发完就不管了，而tcp方式的request\_&\_reply模型发送成功应该是得到对方的接收确认后才接收成功，send函数返回成功值）。

flags参数如果是ZMQ\_DOWNWAIT则send行为是立即返回（非阻塞方式）：能够把字节串加入到消息队列当中便返回成功，否则返回失败（-1）；flags参数是ZMQ\_SNDMORE表示发送multi-part messages，当flag是0的时候表示这个消息的最后一部分。

假如我发送一个multi-part消息的一部分，然后再发送其他的消息，接着再返回发送其他部分，那么发送的行为是怎么样？经过测试发现在publish-subscribe模型（对应的socket类型是PUB和SUB·）下，publisher是发送就不管有没有接收到的形式。

A successful invocation of zmq\_send() does not indicate that the message has been transmitted to the network, only that（仅仅是） it has been queued on the socket and ØMQ has assumed responsibility for the message.

send做的只是把消息放到socket的队列中，暂时的观点：Socket怎么处理这个队列由socket类型确定，处理方式包括比如拒绝新消息加入（send要么阻塞要么fail），等待发送，立即发送等等；消息的发送是由socket背后的一个线程维护的，它决定什么时候发送，查看是不是到了HWM等等，它保证了消息的异步发送，它对程序员是透明的；

#### 通信模型

zmq支持的通信模型包括···

###### request-reply

Reply用的socket类型时ZMQ\_REP，Request用的socket类型时ZMQ\_REQ。在这个模型下（REQ-REP socket对），client只能依顺序“zmq\_send()···zmq\_recv()”运作：要想recv前面必须要有send，而不能有连续的send和连续的recv。同样的server也是只能依顺序“zmq\_recv()···zmq\_send()”。

1、创建一个context，创建socket（REQ或REP）

2、bind或者connect

###### publish-subscribe（一对多）

1、创建一个context，创建socket（SUB或PUB）

2、bind或者“用setsockopt加消息过滤器以及connect”

该模式下，通信是单向的，即zmq\_recv不能以PUB类型socket作为输入参数，而zmq\_send不能以SUB类型socket作为输入参数。网络通信不是单片机那种硬件上的多个设备挂同一条线上，在这条线上发送一次消息，所有的终端都能接收到，网络通信是数据包的形式，一个数据包被一个终端接收到了，那么其他终端就接收不到（除非复制再转发），所以对于同一条消息，publisher对每一个subscriber都要发一次（fan out），publisher面向每一个connect都有一个queue，然后每一个queue都由high water mark，超过其中一条HWM后publisher就把准备加入该队列的message丢弃，直到降到HWM以下，这也说明该类型socket要工作在非阻塞模式下；

HWM是因为送消息到queue和消息从queue（有发送的queue和接收的queue）中发到网络上是两个过程，每个queue都有一个HWM，socket的主要任务便是维护这些queue（一个socket可以用多个queue，这不同于网络编程中只有一个发送缓冲和一个接收缓冲）；

#### 临时

如果以tcp方式通信，那分配的IP也是要实际存在的，不是随便设置一个就行的，同时那些网段路由转发规则等也要遵守；zmq的tcp方式是在TCP/IP之上的一个协议，属于应用层；

不同主机之间的进程通信不能用ipc方式；ipc通信方式会产生一个文件，如weather.ipc，文件名符合所在OS（Windows上暂时不能使用ipc方式）要求即可，一般都加后缀.ipc以防与其他的文件冲突；创建完之后还有注意不同用户账户登录访问权限的问题，其他进程访问路径正确问题（暂时先不管ipc方式，先掌握tcp方式下的两种成熟的通信模型）；相比于tcp和ipc，inproc方式更快（线程间通信），但是有一个缺陷就是server必须先启动，然后client才能连接得上（这是未来zmq版本要解决得问题，但是当前用inproc方式的socket需要注意的一点，但是正如前面所说的，只关注tcp方式）；

###### socket的消息队列（message queue）

这是理解zmq通信机制的核心概念；所有的socket由一个background线程维护各自的接收队列和发送队列，该线程对程序员是透明的；该线程负责了主要的I/O工作，zmq\_send做的只是把message加入到queue，具体的什么时候发送，什么时候接收由这个线程负责（一般是一个context对应一个这样的background线程，但可以通过zmq\_context\_set去增加这样的线程，进一步压榨CPU，提高传输效率，一个这样的background线程通信速率平均是1Gbyte/s，但如果是inproc方式，那么这个线程可以设置为0）；endpoint不需要存在也可以“建立连接”；线程同时监听endpoint的上线情况，总之zmq封装好多东西，是背后的线程帮助程序员维护这些（比如connecting，queuing等等）；建议在多线程编程中一个thread用一个socket；

The libzmq core library has in fact two APIs to send and receive messages. The zmq\_send() and zmq\_recv() methods that we’ve already seen and used are simple one-liners. We will use these often, but zmq\_recv() is bad at dealing with arbitrary message sizes: it truncates messages to whatever buffer size you provide. So there’s a second API that works with zmq\_msg\_t structures, with a richer but more difficult API:

•Initialise a message: zmq\_msg\_init(), zmq\_msg\_init\_size(), zmq\_msg\_init\_data().

•Sending and receiving a message: zmq\_msg\_send(), zmq\_msg\_recv().

•Release a message: zmq\_msg\_close().

•Access message content: zmq\_msg\_data(), zmq\_msg\_size(), zmq\_msg\_more().

•Work with message properties: zmq\_msg\_get(), zmq\_msg\_set().

•Message manipulation: zmq\_msg\_copy(), zmq\_msg\_move().

#### Q\_&\_A

###### tcp通信方式

**场景：**通过tcp的方式实现一对多的通信，然后publisher要bind的IP\_&\_port和subscriber要connect的IP\_&\_port需要一样：

publisher是zmq\_bind(publisher, "tcp://localhost:5556");

subscriber是zmq\_connect(subscriber, "tcp://localhost:5556");

结果发现不能正常接收消息。

**原因：**通过查看官方文档zmq\_tcp、zmq\_connect以及zmq\_bind发现在TCP通信方式下，zmq\_bind第二个参数限制为以下情况：

An interface may be specified by either of the following:

The wild-card \*, meaning all available interfaces.（绑定所有还是绑定其中一个？）

The primary IPv4 or IPv6 address assigned to the interface, in its numeric representation.

The non-portable interface name as defined by the operating system.

The TCP port number may be specified by:

A numeric value, usually above 1024 on POSIX systems.

The wild-card \*, meaning a system-assigned ephemeral port.

localhost是一个DNS名字，DNS是一个域名系统，比如www.baidu.com对应ip是202.108.22.5，而localhost对应的ip一般是127.0.0.1，localhost不是一个interface name as defined by the operating system，通过ifconfig命令查看到的那些网络接口名字才是，比如eth0，lo等。

而在TCP通信方式下，zmq\_connect第二个参数要求是：

A peer address may be specified by either of the following:

The DNS name of the peer.

The IPv4 or IPv6 address of the peer, in its numeric representation.

###### 多连接疑惑

一个socket可以连接多个endpoint，那么如何指定发送或者接收的endpoint呢？zmq\_send和zmq\_recv都美哟这个选项，难道是zmq\_setsockopt？？？比如一个subscriber订阅多个publisher的消息，那么这个subscriber是怎么recv的？是调用多次zmq\_recv，然后像转盘一样依次接收每个publisher的消息？