zmq使用多种协议，inproc（进程内）、ipc（进程间）、tcp、pgm（广播）、epgm；

在设计架构时，应遵循”服务端是稳定的，客户端是灵活的”

TCP套接字和ZMQ套接字之间在传输数据方面的区别：

1. ZMQ套接字传输的是消息，而不是字节（TCP）或帧（UDP）。消息指的是一段指定长度的二进制数据块，我们下文会讲到消息，这种设计是为了性能优化而考虑的，所以可能会比较难以理解。
2. ZMQ套接字在后台进行I/O操作，也就是说无论是接收还是发送消息，它都会先传送到一个本地的缓冲队列，这个内存队列的大小是可以配置的。
3. ZMQ套接字可以和多个套接字进行连接（如果套接字类型允许的话）。TCP协议只能进行点对点的连接，而ZMQ则可以进行一对多（类似于无线广播）、多对多（类似于邮局）、多对一（类似于信箱），当然也包括一对一的情况。
4. ZMQ套接字可以发送消息给多个端点（扇出模型），或从多个端点中接收消息（扇入模型）

zmq\_recv()方法使用了公平队列的算法来决定接收哪个连接的消息

调用zmq\_send()方法时其实并没有真正将消息发送给套接字连接。消息会在一个内存队列中保存下来，并由后台的I/O线程异步地进行发送。如果不出意外情况，这一行为是非阻塞的。所以说，即便zmq\_send()有返回值，并不能代表消息已经发送。当你在用zmq\_msg\_init\_data()初始化消息后，你不能重用或是释放这条消息，否则ZMQ的I/O线程会认为它在传输垃圾数据。

ZMQ应用程序和传统应用程序的区别之一就是你不需要为每个套接字都创建一个连接。传统应用程序每个进程或线程会有一个远程连接，它又只能处理一个套接字。

让我们回顾一下ZMQ会为你做些什么：它会将消息快速高效地发送给其他节点，这里的节点可以是线程、进程、或是其他计算机；ZMQ为应用程序提供了一套简单的套接字API，不用考虑实际使用的协议类型（进程内、进程间、TPC、或广播）；当节点调动时，ZMQ会自动进行连接或重连；无论是发送消息还是接收消息，ZMQ都会先将消息放入队列中，并保证进程不会因为内存溢出而崩溃，适时地将消息写入磁盘；ZMQ会处理套接字异常；所有的I/O操作都在后台进行；ZMQ不会产生死锁。

ZMQ的核心消息模式有：

1. **请求-应答模式** 将一组服务端和一组客户端相连，用于远程过程调用或任务分发。
2. **发布-订阅模式** 将一组发布者和一组订阅者相连，用于数据分发。
3. **管道模式** 使用扇入或扇出的形式组装多个节点，可以产生多个步骤或循环，用于构建并行处理架构。我们在第一章中已经讲述了这些模式，不过还有一种模式是为那些仍然认为ZMQ是类似TCP那样点对点连接的人们准备的：
4. **排他对接模式** 将两个套接字一对一地连接起来，这种模式应用场景很少，我们会在本章最末尾看到一个示例。

zmq\_socket()函数的说明页中有对所有消息模式的说明，比较清楚，因此值得研读几次。

以下是合法的套接字连接-绑定对（一端绑定、一端连接即可）：

* **PUB - SUB**
* **REQ - REP**
* **REQ - ROUTER**
* **DEALER - REP**
* **DEALER - ROUTER**
* **DEALER - DEALER**
* **ROUTER - ROUTER**
* **PUSH - PULL**
* **PAIR – PAIR**

ZMQ的消息是作为一个整体来收发的，你不会只收到消息的一部分；

ZMQ不会立即发送消息，而是有一定的延迟；

你可以发送0字节长度的消息，作为一种信号；

消息必须能够在内存中保存，如果你想发送文件或超长的消息，就需要将他们切割成小块，在独立的消息中进行发送；

必须使用zmq\_msg\_close()函数来关闭消息，但在一些会在变量超出作用域时自动释放消息对象的语言中除外

不要贸然使用zmq\_msg\_init\_data()函数。它是用于零拷贝，而且可能会造成麻烦。

如果我们想要读取多个套接字中的消息呢？最简单的方法是将套接字连接到多个端点上，让ZMQ使用公平队列的机制来接受消息。如果不同端点上的套接字类型是一致的，那可以使用这种方法。但是，如果一个套接字的类型是PULL，另一个是PUB怎么办？如果现在开始混用套接字类型，那将来就没有可靠性可言了。

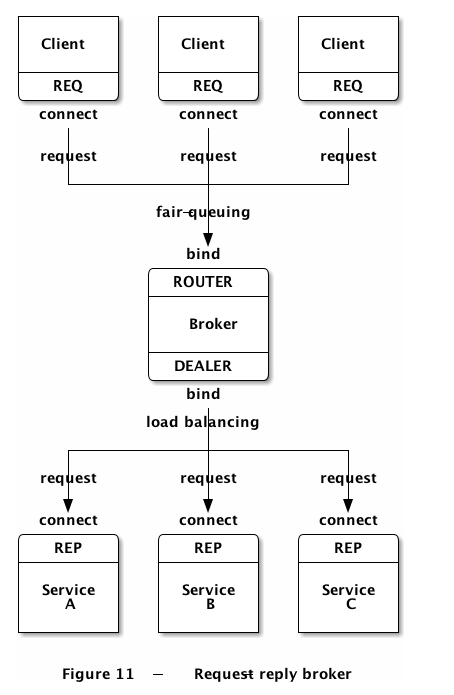
现实中的代码应该对每一次的ZMQ函数调用作错误处理。

ZMQ的原则是：如果需要解决一个新的问题，就该使用新的套接字。

* 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；
* 如果使用了zmq\_poll()函数，当收到了消息的第一帧时，其它帧其实也已经收到了；
* 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。

连接多个客户端和多个服务端有两种方式：

1. 让客户端直接和多个服务端进行连接。客户端套接字可以连接至多个服务端套接字，它所发送的请求会通过负载均衡的方式分发给服务端。这种方式的缺点在于添加服务端时，需要修改客户端的设置。
2. 使用一个模块，如下图请求-应答代理，这个模块能够让C/S网络结构更易于扩展，网络中唯一稳定的组件是中间的代理装置



QUEUE装置应使用ROUTER/DEALER套接字、FORWARDER应使用SUB/PUB、STREAMER应使用PULL/PUSH。

使用ZMQ进行多线程编程时，**不需要考虑互斥、锁、或其他并发程序中要考虑的因素，你唯一要关心的仅仅是线程之间的消息**。

ZMQ进行多线程编程，以下是一些规则：

1. 不要在不同的线程之间访问同一份数据，如果要用到传统编程中的互斥机制，那就有违ZMQ的思想了。唯一的例外是ZMQ上下文对象，它是线程安全的。
2. 必须为进程创建ZMQ上下文，并将其传递给所有你需要使用inproc协议进行通信的线程；
3. 你可以将线程作为单独的任务来对待，使用自己的上下文，但是这些线程之间就不能使用inproc协议进行通信了。这样做的好处是可以在日后方便地将程序拆分为不同的进程来运行。
4. 不要在不同的线程之间传递套接字对象，这些对象不是线程安全的。从技术上来说，你是可以这样做的，但是会用到互斥和锁的机制，这会让你的应用程序变得缓慢和脆弱。唯一合理的情形是，在某些语言的ZMQ类库内部，需要使用垃圾回收机制，这时可能会进行套接字对象的传递。

使用PAIR套接字进行线程间的协调是最合适的。

使用PAIR套接字时，若远程节点断开连接后又进行重连，PAIR不会予以理会。

s\_recv(argument1, sizeof())

是否有办法在程序崩溃时让这些套接字缓存得以保留，稍后能够恢复？

如果接收方（SUB、PULL、REQ）指定了套接字标识，当它们断开网络时，发送方（PUB、PUSH、REP）会为它们缓存信息，直至达到阈值（HWM）。这里发送方不需要有套接字标识。

要将瞬时套接字转化为持久套接字，需要为其设定一个套接字标识。所有的ZMQ套接字都会有一个标识，不过是由ZMQ自动生成的UUID。在ZMQ内部，两个套接字相连时会先交换各自的标识。如果发生对方没有ID，则会自行生成一个用以标识对方。但套接字也可以告知对方自己的标识，那当它们第二次连接时，就能知道对方的身份

zmq\_setsockopt (socket, ZMQ\_IDENTITY, "Lucy", 4);

关于套接字标识还有几点说明：

* 如果要为套接字设置标识，必须在连接或绑定至端点之前设置；
* 接收方会选择使用套接字标识，正如cookie在HTTP网页应用中的性质，是由服务器去选择要使用哪个cookie的；
* 套接字标识是二进制字符串；以字节0开头的套接字标识为ZMQ保留标识；
* 不用为多个套接字指定相同的标识，若套接字使用的标识已被占用，它将无法连接至其他套接字；
* 不要使用随机的套接字标识，这样会生成很多持久化套接字，最终让节点崩溃；
* 如果你想获取对方套接字的标识，只有ROUTER套接字会帮你自动完成这件事，使用其他套接字类型时，需要将标识作为消息的一帧发送过来；
* 说了以上这些，使用持久化套接字其实并不明智，因为它会让发送者越来越混乱，让架构变得脆弱。如果我们能重新设计ZMQ，很可能会去掉这种显式声明套接字标识的功能。

s\_sendmore消息阻塞函数，如果没发送成功会报错

所有的套接字类型都可以使用标识。如果你在使用PUB和SUB套接字，其中SUB套接字为自己声明了标识，那么，当SUB断开连接时，PUB会保留要发送给SUB的消息。**如果你在使用持久化的SUB套接字（即为SUB设置了套接字标识），那么你必须设法避免消息在发布者队列中堆砌并溢出，应该使用阈值（HWM）来保护发布者套接字。**发布者的阈值会分别影响所有的订阅者。

这里有一些典型的方法用以处理不同的订阅者：

* 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。
* 如果你不希望慢速或崩溃的订阅者丢失消息，可以设置一个交换区，在高峰期的时候存放这些消息。交换区的大小可以根据订阅者数、高峰消息比率、消息平均大小、暂存时间等来衡量。比如，你预计有5000个订阅者，消息大小为200个字节左右，每秒会有10万条消息。这样，你每秒就需要100MB的磁盘空间来存放消息。加总起来，你会需要6GB的磁盘空间，而且必须足够的快

关于持久化订阅者：

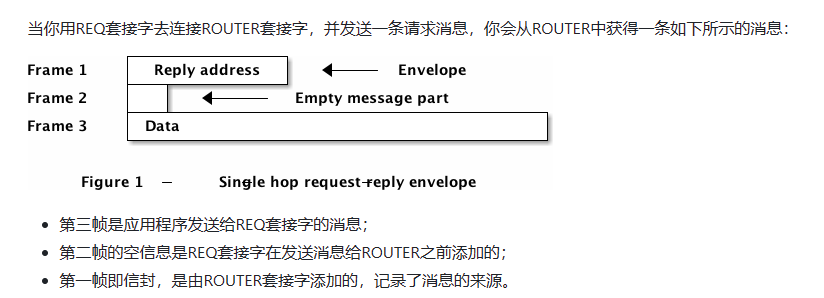
* 数据可能会丢失，这要看消息发布的频率、网络缓存大小、通信协议等。持久化的订阅者比起瞬时套接字要可靠一些，但也并不是完美的。
* 交换区文件是无法恢复的，所以当发布者或代理消亡时，交换区中的数据仍然会丢失。

关于阈值：

* 这个选项会同时影响套接字的发送和接收队列。当然，PUB、PUSH不会有接收队列，SUB、PULL、REQ、REP不会有发送队列。而像DEALER、ROUTER、PAIR套接字时，他们既有发送队列，又有接收队列。
* 当套接字达到阈值时，ZMQ会发生阻塞，或直接丢弃消息。
* 使用inproc协议时，发送者和接受者共享同一个队列缓存，所以说，真正的阈值是两个套接字阈值之和。如果一方套接字没有设置阈值，那么它就不会有缓存方面的限制。

信封机制在ROUTER中的工作原理：

1. 从ROUTER中读取一条消息时，ØMQ会包上一层信封，上面注明了消息的来源。
2. 向ROUTER写入一条消息时（包含信封），ØMQ会将信封拆开，并将消息递送给相应的对象。



请求-应答模式中使用到的四种套接字类型：

* DEALER是一种负载均衡，它会将消息分发给已连接的节点，并使用公平队列的机制处理接受到的消息。DEALER的作用就像是PUSH和PULL的结合。
* REQ发送消息时会在消息顶部插入一个空帧，接受时会将空帧移去。其实REQ是建立在DEALER之上的，但REQ只有当消息发送并接受到回应后才能继续运行。
* ROUTER在收到消息时会在顶部添加一个信封，标记消息来源。发送时会通过该信封决定哪个节点可以获取到该条消息。
* REP在收到消息时会将第一个空帧之前的所有信息保存起来，将原始信息传送给应用程序。在发送消息时，REP会用刚才保存的信息包裹应答消息。REP其实是建立在ROUTER之上的，但和REQ一样，必须完成接受和发送这两个动作后才能继续。

我们对请求-应答模式下的路由做一个小结：

* 对于瞬时的套接字，ROUTER会动态生成一个UUID来标识它，因此从ROUTER中获取到的消息里会包含这个标识；
* 对于持久的套接字，可以自定义标识，ROUTER会直接将该标识放入消息之中；
* 具有显式声明标识的节点可以连接到其他类型的套接字；
* 节点可以通过配置文件等机制提前获知对方节点的标识，作出相应的处理。

黑洞(只负责处理消息，不给任何返回)、代理(将消息转发给其他结点)、服务(会发送返回信息)。

最近最少使用算法路由（LRU模式）

和DEALER相同，REQ只能和一个ROUTER连接，除非你想做类似多路冗余路由这样的事（我甚至不想在这里解释），太复杂。

不要在现实环境中使用随机标识的持久套接字，这样做会将节点消耗殆尽

在经典的请求-应答模式中，ROUTER一般不会和REP套接字通信，而是由DEALER去和REP通信。DEALER会将消息随机分发给多个REP，并获得结果。ROUTER更适合和REQ套接字通信。