ZMQ　第二章 ZeroMQ 进阶

第一章我们简单试用了 ZMQ 的若干通信模式：**请求-应答模式、发布-订阅模式、管道模式**。这一章我们将学习更多在实际开发中会使用到的东西：

本章涉及的内容有：

* 创建和使用 ZMQ 套接字

* 使用套接字发送和接收消息

* 使用 ZMQ 提供的异步 I/O 套接字构建你的应用程序

* 在单一线程中使用多个套接字

* 恰当地处理致命和非致命错误

* 处理诸如 Ctrl-C 的中断信号

* 正确地关闭 ZMQ 应用程序

* 检查 ZMQ 应用程序的内存泄露

* 发送和接收多帧消息

* 在网络中转发消息

* 建立简单的消息队列代理

* 使用 ZMQ 编写多线程应用程序

* 使用 ZMQ 在线程间传递信号

* 使用 ZMQ 协调网络中的节点

* 使用标识创建持久化套接字

* 在发布-订阅模式中创建和使用消息信封

* 如何让持久化的订阅者能够从崩溃中恢复

* 使用阈值（HWM）防止内存溢出

零的哲学

ØMQ 一词中的Ø让我们纠结了很久。一方面，这个特殊字符会降低 ZMQ 在谷歌和推特中的收入量；另一方面，这会惹恼某些丹麦语种的民族，他们会嚷道Ø并不是一个奇怪的 0。

一开始 ZMQ 代表零中间件、零延迟，同时，它又有了新的含义：零管理、零成本、零浪费。总的来说，零表示最小、最简，这是贯穿于该项目的哲理。我们致力于减少复杂程度，提高易用性。

套接字 API

说实话，ZMQ 有些偷梁换柱的嫌疑。不过我们并不会为此道歉，因为这种概念上的切换绝对不会有坏处。ZMQ 提供了一套类似于 BSD 套接字的 API，但将很多消息处理机制的细节隐藏了起来，你会逐渐适应这种变化，并乐于用它进行编程。

套接字事实上是用于网络编程的标准接口，ZMQ 之所那么吸引人眼球，原因之一就是它是建立在标准套接字 API 之上。因此，ZMQ 的套接字操作非常容易理解，其生命周期主要包含四个部分：

* **创建和销毁套接字：zmq_socket(), zmq_close()**

* **配置和读取套接字选项：zmq_setsockopt(), zmq_getsockopt()**

* **为套接字建立连接：zmq_bind(), zmq_connect()**

* **发送和接收消息：zmq_send(), zmq_recv()**

如以下 C 代码：

```c

void *mousetrap;

// Create socket for catching mice

mousetrap = zmq_socket (context, ZMQ_PULL);
```

```
// Configure the socket

int64_t jawsize = 10000;

zmq_setsockopt (mousetrap, ZMQ_HWM, &jawsize, sizeof jawsize);



// Plug socket into mouse hole

zmq_connect (mousetrap, "tcp://192.168.55.221:5001");



// Wait for juicy mouse to arrive

zmq_msg_t mouse;

zmq_msg_init (&mouse);

zmq_recv (mousetrap, &mouse, 0);

// Destroy the mouse

zmq_msg_close (&mouse);
```

// Destroy the socket

zmq_close (mousetrap);

```

请注意，**套接字永远是空指针类型的，而消息则是一个数据结构**（我们下文会讲述）。所以，在 C 语言中你通过变量传递套接字，而用引用传递消息。记住一点，在 ZMQ 中所有的套接字都是由 ZMQ 管理的，只有消息是由程序员管理的。

创建、销毁、以及配置套接字的工作和处理一个对象差不多，但请记住 ZMQ 是异步的，伸缩性很强，因此在将其应用到网络结构中时，可能会需要多一些时间来理解。

## ### 使用套接字构建拓扑结构

在连接两个节点时，其中一个需要使用 zmq_bind()，另一个则使用 zmq_connect()。通常来讲，使用 zmq_bind()连接的节点称之为服务端，它有着一个较为固定的网络地址；使用 zmq_connect()连接的节点称为客户端，其地址不固定。我们会有这样的说法：绑定套接字至端点；连接套接字至端点。端点指的是某个广为周知网络地址。

ZMQ 连接和传统的 TCP 连接是有区别的，主要有：

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

* 当客户端使用 zmq_connect()时连接就已经建立了，并不要求该端点已有某个服务使用 zmq_bind()进行了绑定；

* 连接是异步的，并由一**组消息队列做缓冲**；
```

* 连接会表现出某种消息模式，这是由创建连接的套接字类型决定的；

* **一个套接字可以有多个输入和输出连接**；

* ZMQ 没有提供类似 zmq_accept()的函数，因为当套接字绑定至端点时它就自动开始接受连接了；

* 应用程序无法直接和这些连接打交道，因为它们是被封装在 ZMQ 底层的。

在很多架构中都使用了类似于 C/S 的架构。服务端组件式比较稳定的，而客户端组件则较为动态，来去自如。所以说，服务端地址对客户端而言往往是可见的，反之则不然。这样一来，架构中应该将哪些组件作为服务端（使用 zmq_bind()），哪些作为客户端（使用 zmq_connect()），就很明显了。同时，这需要和你使用的套接字类型相联系起来，我们下文会详细讲述。

让我们试想一下，如果先打开了客户端，后打开服务端，会发生什么？传统网络连接中，我们打开客户端时一定会收到系统的报错信息，但 ZMQ 让我们能够自由地启动架构中的组件。当客户端使用 zmq_connect()连接至某个端点时，它就已经能够使用该套接字发送消息了。如果这时，服务端启动起来了，并使用 zmq_bind()绑定至该端点，ZMQ 将自动开始转发消息。

服务端节点可以仅使用一个套接字就能绑定至多个端点。也就是说，它能够使用不同的协议来建立连接：

```c
zmq_bind (socket, "tcp://*:5555");

zmq_bind (socket, "tcp://*:9999");
```

```
zmq_bind (socket, "ipc://myserver.ipc");
```

```

当然，你不能多次绑定至同一端点，这样是会报错的。

每当有客户端节点使用 zmq_connect()连接至上述某个端点时，服务端就会自动创建连接。ZMQ 没有对连接数量进行限制。此外，客户端节点也可以使用一个套接字同时建立多个连接。

大多数情况下，哪个节点充当服务端，哪个作为客户端，是网络架构层面的内容，而非消息流问题。不过也有一些特殊情况（如失去连接后的消息重发），同一种套接字使用绑定和连接是会有一些不同的行为的。

所以说，当我们在设计架构时，应该遵循"服务端是稳定的，客户端是灵活的"原则，这样就不太会出错。

套接字是有类型的，套接字类型定义了套接字的行为，它在发送和接收消息时的规则等。你可以将不同种类的套接字进行连接，如 PUB-SUB 组合，这种组合称之为发布-订阅模式，其他组合也会有相应的模式名称，我们会在下文详述。

正是因为套接字可以使用不同的方式进行连接，才构成了 ZMQ 最基本的消息队列系统。我们还可以在此基础之上建立更为复杂的装置、路由机制等，下文会详述。总的来说，ZMQ 为你提供了一套组件，供你在网络架构中拼装和使用。

## ### 使用套接字传递数据

发送和接收消息使用的是 zmq_send()和 zmq_recv()这两个函数。虽然函数名称看起来很直白，但由于 ZMQ 的 I/O 模式和传统的 TCP 协议有很大不同，因此还是需要花点时间去理解的。
```

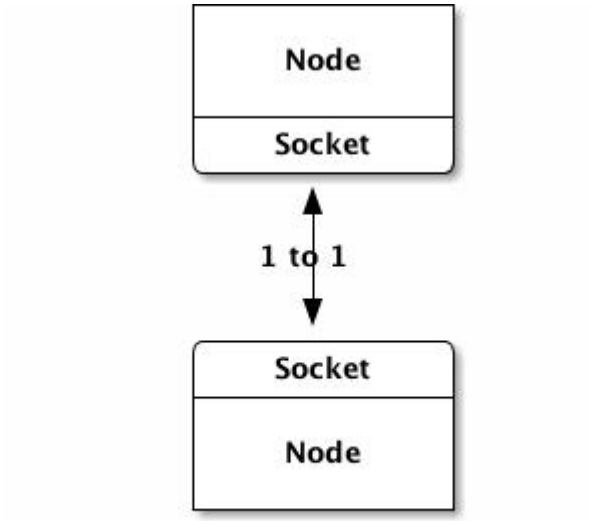

Figure 1 — TCP sockets are 1 to 1

让我们看一看 TCP 套接字和 ZMQ 套接字之间在传输数据方面的区别：

* **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。

* **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。**

* **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。**

* **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）**

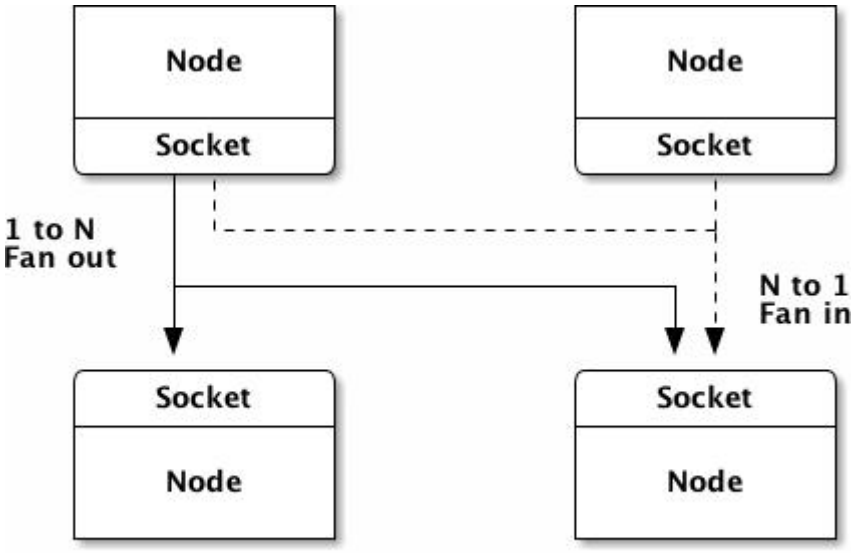

Figure 2 — 0MQ sockets are N to N

所以，**向套接字写入一个消息时可能会将消息发送给很多节点**，相应的，套接字又会从所有已建立的连接中接收消息。**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。这对初学者来讲是一个常犯的错误，下文我们会讲述如何正确地处理消息。

单播传输

ZMQ 提供了一组**单播传输协议（inporc, ipc, tcp），和两个广播协议（epgm, pgm）**。广播协议是比较高级的协议，我们会在以后讲述。如果你不能回答我扇出比例会影响一对多的单播传输时，就先不要去学习广播协议了吧。

TCP 协议:一般而言我们会使用**tcp**作为传输协议，这种 **TCP 连接是可以脱机运作的**，它灵活、便携、且足够快速。为什么称之为脱机，是因为 ZMQ 中的 TCP 连接不需要该端点已经有某个服务进行了绑定，客户端和服务端可以随时进行连接和绑定，这对应用程序而言都是透明的。

进程间协议:，即**ipc**，和 tcp 的行为差不多，但已从网络传输中抽象出来，不需要指定 IP 地址或者域名。这种协议很多时候会很方便，本指南中的很多示例都会使用这种协议。**ZMQ 中的 ipc 协议同样可以是脱机的**，但有一个缺点——无法在 Windows 操作系统上运作，这一点也许会在未来的 ZMQ 版本中修复。我们一般会在端点名称的末尾附上.ipc 的扩展名，在 UNIX 系统上，使用 ipc 协议还需要注意权限问题。你还需要保证所有的程序都能够找到这个 ipc 端点。

进程内协议:，即inproc**，可以在同一个进程的不同线程之间进行消息传输**，它比 ipc 或 tcp 要快得多。这种协议有一个要求，**必须先绑定到端点，才能建立连接，也许未来也会修复。通常的做法是先启动服务端线程，绑定至端点，后启动客户端线程，连接至端点。**

ZMQ 不只是数据传输

经常有新人会问，如何使用 ZMQ 建立一项服务？我能使用 ZMQ 建立一个 HTTP 服务器吗？

他们期望得到的回答是，我们用普通的套接字来传输 HTTP 请求和应答，那用 ZMQ 套接字也能够完成这个任务，且能运行得更快、更好。

只可惜答案并不是这样的。**ZMQ 不只是一个数据传输的工具，而是在现有通信协议之上建立起来的新架构**。它的数据帧和现有的协议并不兼容，如下面是一个 HTTP 请求和 ZMQ 请求的对比，同样使用的是 TCP/IPC 协议：

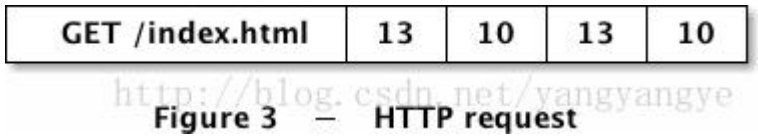

Figure 3 — HTTP request

HTTP 请求**使用 CR-LF（换行符）作为信息帧的间隔，而 ZMQ 则使用指定长度来定义帧**：

Figure 4 — 0MQ request

所以说，你的确是**可以用 ZMQ 来写一个类似于 HTTP 协议的东西，但是这并不是 HTTP**。

不过，如果有人问我如何更好地使用 ZMQ 建立一个新的服务，我会给出一个不错的答案，那就是：你可以自行设计一种通信协议，用 ZMQ 进行连接，使用不同的语言提供服务和扩展，可以在本地，亦可通过远程传输。赛德•肖的 [Mongrel2](http://www.mongrel2.org)网络服务的架构就是一个很好的示例。

I/O 线程

我们提过 ZMQ 是通过后台的 I/O 线程进行消息传输的。**一个 I/O 线程已经足以处理多个套接字的数据传输要求，当然，那些极端的应用程序除外**。这也就是我们在创建上下文时传入的 1 所代表的意思：

```c
void *context = zmq_init (1);           //I/O 线程数
```

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。**单个 ZMQ 套接字可以处理所有的发送和接收任务**。如，当你需要向一千个订阅者发布消息时，使用一个套接字就可以了；当你需要向二十个服务进程分发任务时，使用一个套接字就可以了；当你需要从一千个网页应用程序中获取数据时，也是使用一个套接字就可以了。

这一特性可能会颠覆网络应用程序的编写步骤，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。ZMQ 让你打破这种结构，使用一个线程来完成所有工作，更易于扩展。

核心消息模式

ZMQ 的套接字 API 中提供了多种消息模式。如果你熟悉企业级消息应用，那这些模式会看起来很熟悉。不过对于新手来说，ZMQ 的套接字还是会让人大吃一惊的。

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。

但是，以上种种的前提是用户能够正确地使用消息模式，这种模式往往也体现出了 ZMQ 的智慧。消息模式将我们从实践中获取的经验进行抽象和重组，用于解决之后遇到的所有问题。ZMQ 的消息模式目前是编译在类库中的，不过未来的 ZMQ 版本可能会允许用户自行制定消息模式。

ZMQ 的消息模式是指不同类型套接字的组合。换句话说，要理解 ZMQ 的消息模式，你需要理解 ZMQ 的套接字类型，它们是如何一起工作的。这一部分是需要死记硬背的。

###ZMQ 的核心消息模式有：

* **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。

* **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。

* **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：

* **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。我们会介绍每种消息模式的内容和应用场景。

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**

其他的组合模式会产生不可预知的结果，在将来的 ZMQ 版本中可能会直接返回错误。你也可以通过代码去了解这些套接字类型的行为。

上层消息模式

上文中的四种核心消息模式是内建在 ZMQ 中的，他们是 API 的一部分，在 ZMQ 的 C++核心类库中实现，能够保证正确地运行。如果有朝一日 Linux 内核将 ZMQ 采纳了进来，那这些核心模式也肯定会包含其中。

在这些消息模式之上，我们会建立更为_上层的消息模式_。这种模式可以用任何语言编写，他们不属于核心类型的一部分，不随 ZMQ 发行，只在你自己的应用程序中出现，或者在 ZMQ 社区中维护。

本指南的目的之一就是为你提供一些上层的消息模式，有简单的（如何正确处理消息），也有复杂的（可靠的发布-订阅模式）。

消息的使用方法

ZMQ 的传输单位是消息，即一个二进制块。你可以使用任意的序列化工具，如谷歌的 Protocal Buffers、XDR、JSON 等，将内容转化成 ZMQ 消息。不过这种转化工具最好是便捷和快速的，这个请自己衡量。

在内存中，ZMQ 消息由 zmq_msg_t 结构表示（每种语言有特定的表示）。在 C 语言中使用 ZMQ 消息时需要注意以下几点：

* 你需要创建和传递 zmq_msg_t 对象，而不是一组数据块；

*** 读取消息时，先用 zmq_msg_init()初始化一个空消息，在讲其传递给 zmq_recv()函数；**

*** 写入消息时，先用 zmq_msg_init_size()来创建消息（同时也已初始化了一块内存区域），然后用 memcpy()函数将信息拷贝到该对象中，最后传给 zmq_send()函数；**

* 释放消息（并不是销毁）时，使用 **zmq_msg_close()函数，它会将对消息对象的引用删除，**最终由 ZMQ 将消息销毁；

* 获取消息内容时需使用 **zmq_msg_data()**函数；若想知道消息的长度，可以使用 **zmq_msg_size()**函数；

* 至于 zmq_msg_move()、zmq_msg_copy()、zmq_msg_init_data()函数，在充分理解手册中的说明之前，建议不好贸然使用。

以下是一段处理消息的典型代码，如果之前的代码你有看的话，那应该会感到熟悉。这段代码其实是从 zhelpers.h 文件中抽出的：

```
// 从套接字中获取 ZMQ 字符串，并转换为 C 语言字符串

static char * s_recv (void *socket) {

    zmq_msg_t message;

    zmq_msg_init (&message);

    zmq_recv (socket, &message, 0);

    int size = zmq_msg_size (&message);

    char *string = malloc (size + 1);

    memcpy (string, zmq_msg_data (&message), size);

    zmq_msg_close (&message);

    string [size] = 0;
```

```
    return (string);
}
// 将 C 语言字符串转换为 ZMQ 字符串，并发送给套接字
static int s_send (void *socket, char *string) {
    int rc;
    zmq_msg_t message;
    zmq_msg_init_size (&message, strlen (string));
    memcpy (zmq_msg_data (&message), string, strlen (string));
    rc = zmq_send (socket, &message, 0);
    assert (!rc);
    zmq_msg_close (&message);
    return (rc);
}
```

你可以对以上代码进行扩展，让其支持发送和接受任一长度的数据。

需要注意的是，当你将一个消息对象传递给 zmq_send()函数后，该对象的长度就会被清零，因此你无法发送同一个消息对象两次，也无法获得已发送消息的内容。

如果你想发送同一个消息对象两次，就需要在发送第一次前新建一个对象，使用 **zmq_msg_copy()函数进行拷贝。这个函数不会拷贝消息内容，只是拷贝引用**。然后你就可以再次发送这个消息了（或者任意多次，只要进行了足够的拷贝）。当消息最后一个引用被释放时，消息对象就会被销毁。

ZMQ 支持多帧消息，即在一条消息中保存多个消息帧。这在实际应用中被广泛使用，我们会在第三章进行讲解。

关于消息，还有一些需要注意的地方：

*** 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**。

再重复一句，**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**。关于 ZMQ 还有太多东西需要你去学习，因此现在暂时不用去考虑如何削减几微妙的开销。

处理多个套接字

在之前的示例中，主程序的循环体内会做以下几件事：

1. 等待套接字的消息；

1. 处理消息；

1. 返回第一步。

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。

正确的方法应该是使用 zmq_poll()函数。**更好的方法是将 zmq_poll()包装成一个框架，编写一个事件驱动的反应器**，但这个就比较复杂了，我们这里暂不讨论。

我们先不使用 zmq_poll()，而用 NOBLOCK（非阻塞）的方式来实现从多个套接字读取消息的功能。下面将气象信息服务和并行处理这两个示例结合起来：

msreader: Multiple socket reader in C

```c

//
```

```
// 从多个套接字中获取消息

// 本示例简单地再循环中使用 recv 函数

//

#include "zhelpers.h"

 int main (void)

{

    // 准备上下文和套接字

    void *context = zmq_init (1);

     // 连接至任务分发器

    void *receiver = zmq_socket (context, ZMQ_PULL);

    zmq_connect (receiver, "tcp://localhost:5557");

     // 连接至天气服务

    void *subscriber = zmq_socket (context, ZMQ_SUB);

    zmq_connect (subscriber, "tcp://localhost:5556");
```

```
zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

 // 处理从两个套接字中接收到的消息

// 这里我们会优先处理从任务分发器接收到的消息

while (1) {

    // 处理等待中的任务

    int rc;

    for (rc = 0; !rc; ) {

        zmq_msg_t task;

        zmq_msg_init (&task);

        if ((rc = zmq_recv (receiver, &task, ZMQ_NOBLOCK)) == 0) {

            // 处理任务

        }

        zmq_msg_close (&task);

    }
```

```
    // 处理等待中的气象更新

    for (rc = 0; !rc; ) {

        zmq_msg_t update;

        zmq_msg_init (&update);

        if ((rc = zmq_recv (subscriber, &update, ZMQ_NOBLOCK)) == 0) {

            // 处理气象更新

        }

        zmq_msg_close (&update);

    }

    // 没有消息，等待 1 毫秒

    s_sleep (1);

}

// 程序不会运行到这里，但还是做正确的退出清理工作

zmq_close (receiver);
```

```
    zmq_close (subscriber);

    zmq_term (context);

    return 0;

}
```

这种方式的缺点之一是，在收到第一条消息之前会**有 1 毫秒的延迟**，这在高压力的程序中还是会构成问题的。此外，你还需要翻阅诸如 nanosleep()的函数，不会造成循环次数的激增。

`示例中将任务分发器的优先级提升了，你可以做一个改进，轮流处理消息，正如 ZMQ 内部做的公平队列机制一样。

下面，让我们看看如何用 zmq_poll()来实现同样的功能：

mspoller: Multiple socket poller in C

```c
//

// 从多个套接字中接收消息

// 本例使用 zmq_poll()函数
```

```
//

#include "zhelpers.h"

 int main (void)

{

    void *context = zmq_init (1);

     // 连接任务分发器

    void *receiver = zmq_socket (context, ZMQ_PULL);

    zmq_connect (receiver, "tcp://localhost:5557");

     // 连接气象更新服务

    void *subscriber = zmq_socket (context, ZMQ_SUB);

    zmq_connect (subscriber, "tcp://localhost:5556");

    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "10001 ", 6);

     // 初始化轮询对象

    zmq_pollitem_t items [] = {
```

```
    { receiver, 0, ZMQ_POLLIN, 0 },
    { subscriber, 0, ZMQ_POLLIN, 0 }
};
// 处理来自两个套接字的消息
while (1) {
    zmq_msg_t message;
    zmq_poll (items, 2, -1);
    if (items [0].revents & ZMQ_POLLIN) {
        zmq_msg_init (&message);
        zmq_recv (receiver, &message, 0);
        // 处理任务
        zmq_msg_close (&message);
    }
    if (items [1].revents & ZMQ_POLLIN) {
```

```
            zmq_msg_init (&message);
            zmq_recv (subscriber, &message, 0);
            // 处理气象更新
            zmq_msg_close (&message);
        }
    }
    // 程序不会运行到这儿
    zmq_close (receiver);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

处理错误和 ETERM 信号

ZMQ 的错误处理机制提倡的是快速崩溃。我们认为，**一个进程对于自身内部的错误来说要越脆弱越好，而对外部的攻击和错误要足够健壮**。举个例子，活细胞会因检测到自身问题而瓦解，但对外界的攻击却能极力抵抗。在 **ZMQ 编程中，断言用得是非常多**的，如同细胞膜一样。**如果我们无法确定一个错误是来自于内部还是外部，那这就是一个设计缺陷了，需要修复**。

在 C 语言中，断言失败会让程序立即中止。其他语言中可以使用异常来做到。

当 ZMQ 检测到来自外部的问题时，它会返回一个错误给调用程序。如果 ZMQ 不能从错误中恢复，那它是不会安静地将消息丢弃的。某些情况下，ZMQ 也会去断言外部错误，这些可以被归结为 BUG。

到目前为止，我们很少看到 C 语言的示例中有对错误进行处理。**现实中的代码应该对每一次的 ZMQ 函数调用作错误处理**。如果你不是使用 C 语言进行编程，可能那种语言的 ZMQ 类库已经做了错误处理。但在 C 语言中，你需要自己动手。以下是一些常规的错误处理手段，从 POSIX 规范开始：

* 创建对象的方法如果失败了会返回 NULL；

* 其他方法执行成功时会返回 0，失败时会返回其他值（一般是-1）；

* 错误代码可以从变量 errno 中获得，或者调用 **zmq_errno()**函数；

* 错误消息可以调用 **zmq_strerror()**函数获得。

有两种情况不应该被认为是错误：

* 当线程使用 NOBLOCK 方式调用 zmq_recv()时，若没有接收到消息，该方法会返回-1，并设置 errno 为 EAGAIN；

* 当线程调用 zmq_term()时，若其他线程正在进行阻塞式的处理，该函数会中止所有的处理，关闭套接字，并使得那些阻塞方法的返回值为-1，errno 设置为 ETERM。

遵循以上规则，你就可以在 ZMQ 程序中使用断言了：

```c

void *context = zmq_init (1);

assert (context);

void *socket = zmq_socket (context, ZMQ_REP);

assert (socket);

int rc;

rc = zmq_bind (socket, "tcp://*:5555");

assert (rc == 0);

```

第一版的程序中我将函数调用直接放在了 assert()函数里面，这样做会有问题，**因为一些优化程序会直接将程序中的 assert()函数去除**。

让我们看看如何正确地关闭一个进程，我们用管道模式举例。当我们在后台开启了一组 worker 时，我们需要在任务执行完毕后关闭它们。我们可以向这些 worker 发送自杀的消息，这项工作由结果收集器来完成会比较恰当。

如何将结果收集器和 worker 相连呢？PUSH-PULL 套接字是单向的。**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**。这里我们使用发布-订阅模式来发送自杀的消息：

* 结果收集器创建 PUB 套接字，并连接至一个新的端点；

* worker 将 SUB 套接字连接至这个端点；

* 当结果收集器检测到任务执行完毕时，会通过 PUB 套接字发送自杀信号；

* worker 收到自杀信号后便会中止。

这一过程不会添加太多的代码：

```c
    void *control = zmq_socket (context, ZMQ_PUB);
    zmq_bind (control, "tcp://*:5559");
    ...
```

```
    // Send kill signal to workers

    zmq_msg_init_data (&message, "KILL", 5);

    zmq_send (control, &message, 0);

    zmq_msg_close (&message);
```

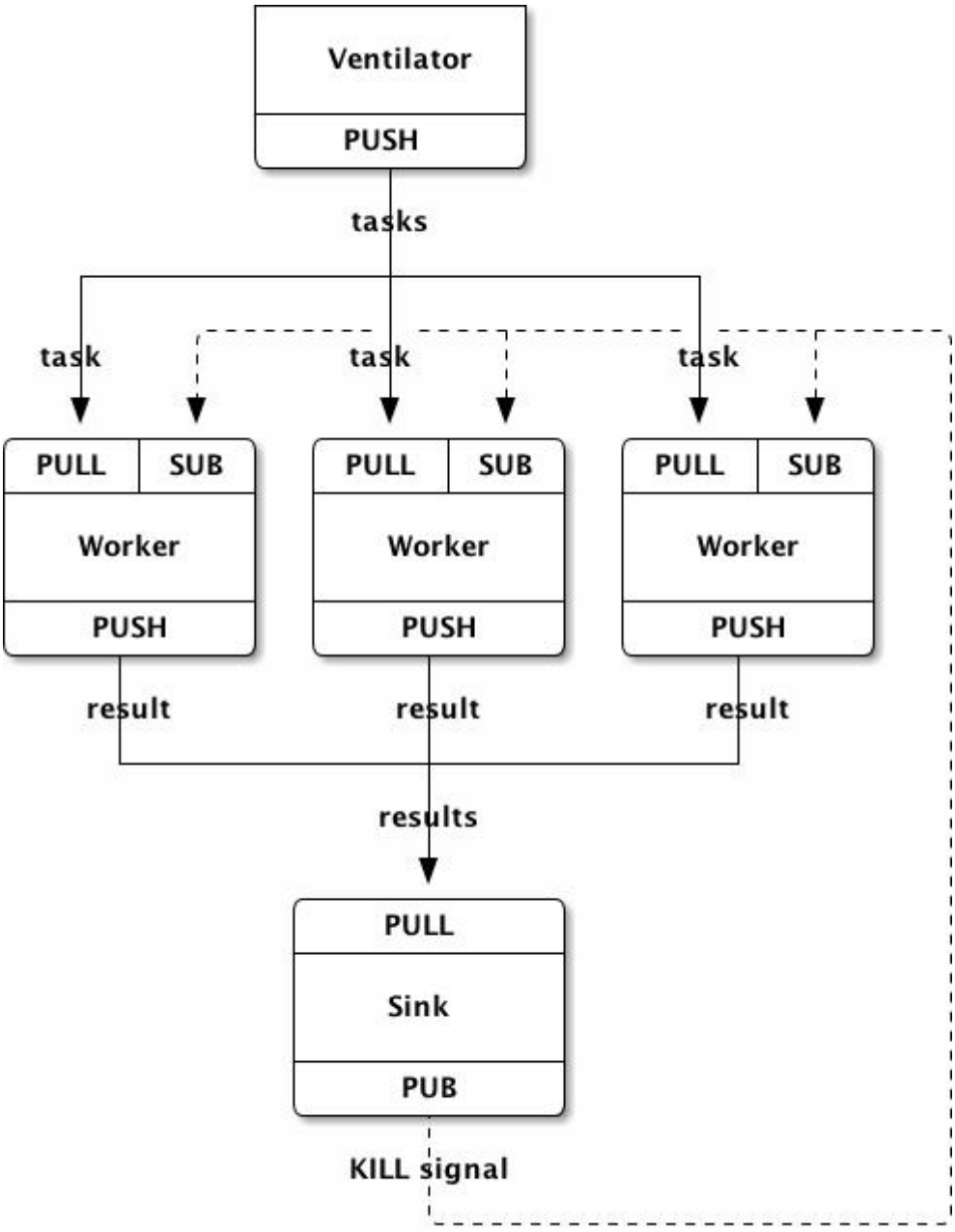

Figure 5 — Parallel Pipeline with Kill signaling

下面是 worker 进程的代码，它会打开三个套接字：用于接收任务的 PULL、用户发送结果的 PUSH、以及用于接收自杀信号的 SUB，使用 zmq_poll()进行轮询：

taskwork2: Parallel task worker with kill signaling in C

```c

//

// 管道模式 - worker 设计 2

// 添加发布-订阅消息流，用以接收自杀消息

//

#include "zhelpers.h"

int main (void)

{

    void *context = zmq_init (1);

    // 用于接收消息的套接字

    void *receiver = zmq_socket (context, ZMQ_PULL);
```

```
zmq_connect (receiver, "tcp://localhost:5557");

 // 用户发送消息的套接字

void *sender = zmq_socket (context, ZMQ_PUSH);

zmq_connect (sender, "tcp://localhost:5558");

 // 用户接收控制消息的套接字

void *controller = zmq_socket (context, ZMQ_SUB);

zmq_connect (controller, "tcp://localhost:5559");

zmq_setsockopt (controller, ZMQ_SUBSCRIBE, "", 0);

 // 处理接收到的任务或控制消息

zmq_pollitem_t items [] = {

   { receiver, 0, ZMQ_POLLIN, 0 },

   { controller, 0, ZMQ_POLLIN, 0 }

};

// 处理消息
```

```
while (1) {

    zmq_msg_t message;

    zmq_poll (items, 2, -1);

    if (items [0].revents & ZMQ_POLLIN) {

        zmq_msg_init (&message);

        zmq_recv (receiver, &message, 0);

        // 工作

        s_sleep (atoi ((char *) zmq_msg_data (&message)));

        // 发送结果

        zmq_msg_init (&message);

        zmq_send (sender, &message, 0);

        // 简单的任务进图指示

        printf (".");

        fflush (stdout);
```

```
            zmq_msg_close (&message);
        }
        // 任何控制命令都表示自杀
        if (items [1].revents & ZMQ_POLLIN)
            break; // 退出循环
    }
    // 结束程序
    zmq_close (receiver);
    zmq_close (sender);
    zmq_close (controller);
    zmq_term (context);
    return 0;
}
```

下面是修改后的结果收集器代码，在收集完结果后向所有 worker 发送自杀消息：

tasksink2: Parallel task sink with kill signaling in C

```c
//
// 管道模式 - 结构收集器 设计 2
// 添加发布-订阅消息流，用以向 worker 发送自杀信号
//
#include "zhelpers.h"
 int main (void)
{
    void *context = zmq_init (1);
     // 用于接收消息的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");
```

```
// 用以发送控制信息的套接字

void *controller = zmq_socket (context, ZMQ_PUB);

zmq_bind (controller, "tcp://*:5559");

// 等待任务开始

char *string = s_recv (receiver);

free (string);

// 开始计时

int64_t start_time = s_clock ();

// 确认 100 个任务处理完毕

int task_nbr;

for (task_nbr = 0; task_nbr < 100; task_nbr++) {

    char *string = s_recv (receiver);

    free (string);

    if ((task_nbr / 10) * 10 == task_nbr)
```

```
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    printf ("总执行时间: %d msec\n",
        (int) (s_clock () - start_time));
    // 发送自杀消息给 worker
    s_send (controller, "KILL");
    // 结束
    sleep (1); // 等待发送完毕
    zmq_close (receiver);
    zmq_close (controller);
    zmq_term (context);
```

```
    return 0;

}
```

处理中断信号

现实环境中，当应用程序收到 Ctrl-C 或其他诸如 ETERM 的信号时需要能够正确地清理和退出。默认情况下，这一信号会杀掉进程，意味着尚未发送的消息就此丢失，文件不能被正确地关闭等。

在 C 语言中我们是这样处理消息的：

interrupt: Handling Ctrl-C cleanly in C

```c
//

// Shows how to handle Ctrl-C

//

#include <zmq.h>

#include <stdio.h>
```

```
#include <signal.h>

 // ---------------------------------------------------------------------

// 消息处理

//

// 程序开始运行时调用 s_catch_signals()函数；

// 在循环中判断 s_interrupted 是否为 1，是则跳出循环；

// 很适用于 zmq_poll()。

 static int s_interrupted = 0;

static void s_signal_handler (int signal_value)

{

    s_interrupted = 1;

}

 //注册信号

static void s_catch_signals (void)
```

```
{
    struct sigaction action;
    action.sa_handler = s_signal_handler;
    action.sa_flags = 0;
    sigemptyset (&action.sa_mask);
    sigaction (SIGINT, &action, NULL);
    sigaction (SIGTERM, &action, NULL);
}

 int main (void)
{
    void *context = zmq_init (1);
    void *socket = zmq_socket (context, ZMQ_REP);
    zmq_bind (socket, "tcp://*:5555");
```

```
s_catch_signals ();

while (1) {

    // 阻塞式的读取会在收到信号时停止

    zmq_msg_t message;

    zmq_msg_init (&message);

    zmq_recv (socket, &message, 0);

     if (s_interrupted) {

        printf ("W: 收到中断消息，程序中止...\n");

        break;

     }

}

zmq_close (socket);

zmq_term (context);

return 0;
```

}

```

这段程序**使用 s_catch_signals()函数来捕捉像 Ctrl-C(SIGINT)和 SIGTERM 这样的信号**。收到任一信号后，该函数会将全局变量 s_interrupted 设置为 1。你的程序并不会自动停止，需要显式地做一些清理和退出工作。

* 在程序开始时调用 s_catch_signals()函数，用来进行信号捕捉的设置；

* 如果程序在 zmq_recv()、zmq_poll()、zmq_send()等函数中阻塞，当有信号传来时，这些函数会返回 EINTR；

* 像 s_recv()这样的函数会将这种中断包装为 NULL 返回；

* 所以，你的应用程序可以检查是否有 EINTR 错误码、或是 NULL 的返回、或者 s_interrupted 变量是否为 1。

如果以下代码就十分典型：

```c

s_catch_signals ();

client = zmq_socket (...);

while (!s_interrupted) {

    char *message = s_recv (client);
```
```

```
    if (!message)

        break; // 按下了 Ctrl-C

}

zmq_close (client);
```

如果你在设置 s_catch_signals()之后没有进行相应的处理，那么你的程序将对 Ctrl-C 和 ETERM 免疫。

检测内存泄露

任何长时间运行的程序都应该妥善的管理内存，否则最终会发生内存溢出，导致程序崩溃。如果你所使用的编程序言会自动帮你完成内存管理，那就要恭喜你了。但若你使用类似 C/C++之类的语言时，就需要自己动手进行内存管理了。下面会介绍一个名为 **valgrind 的工具**，可以用它来报告内存泄露的问题。

* 在 Ubuntu 或 Debian 操作系统上安装 valgrind：**sudo apt-get install valgrind**

* **缺省情况下，ZMQ 会让 valgrind 不停地报错，想要屏蔽警告的话可以在编译 ZMQ 时使用 ZMQ_MAKE_VALGRIND_HAPPY 宏选项**：

```
$ cd zeromq2
```

$ **export CPPFLAGS=-DZMQ_MAKE_VALGRIND_HAPPY**

$ ./configure

$ make clean; make

$ sudo make install

```

* 应用程序应该正确地处理 Ctrl-C，特别是对于长时间运行的程序（如队列装置），如果不这么做，valgrind 会报告所有已分配的内存发生了错误。

* **使用-DDEBUG 选项编译程序，这样可以让 valgrind 告诉你具体是哪段代码发生了内存溢出**。

* 最后，使用如下方法运行 valgrind：

```

valgrind --tool=memcheck --leak-check=full someprog

```

解决完所有的问题后，你会看到以下信息：

```

==30536== ERROR SUMMARY: 0 errors from 0 contexts...

```

### 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，特别是那些有关"信封"的应用，我们下文会谈到。我们这一节要讲的是如何正确地收发多帧消息。

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
zmq_send (socket, &message, ZMQ_SNDMORE);       //发第 1 帧
...
zmq_send (socket, &message, ZMQ_SNDMORE);       //发第 2 帧
...
zmq_send (socket, &message, 0);                  //发最后帧
```
```

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
while (1) {
        zmq_msg_t message;
        zmq_msg_init (&message);
        zmq_recv (socket, &message, 0);
       // 处理一帧消息
        zmq_msg_close (&message);
        int64_t more;
        size_t more_size = sizeof (more);
        zmq_getsockopt (socket, ZMQ_RCVMORE, &more, &more_size);
        if (!more)       //判断是否是最后消息
       {
            break; //  已到达最后一帧
```

```
    }

}

```

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；

* 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。

中间件和装置

当网络组件的数量较少时，所有节点都知道其它节点的存在。但随着节点数量的增加，这种结构的成本也会上升。因此，我们需要将这些组件拆分成更小的模块，使用一个中间件来连接它们。

这种结构在现实世界中是非常常见的，我们的社会和经济体系中充满了中间件的机制，用以降低复杂度，压缩构建大型网络的成本。中间件也会被称为批发商、分包商、管理者等等。

ZMQ 网络也是一样，如果规模不断增长，就一定会需要中间件。ZMQ 中，我们称其为"装置"（Device）。在构建 ZMQ 软件的初期，我们会画出几个节点，然后将它们连接起来，不使用中间件：

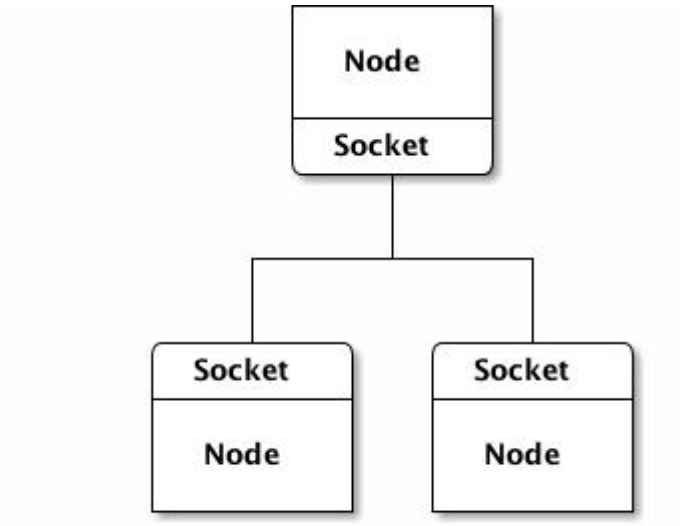

Figure 6 — Small scale 0MQ application

随后，我们对这个结构不断地进行扩充，**将装置放到特定的位置，进一步增加节点数量**：

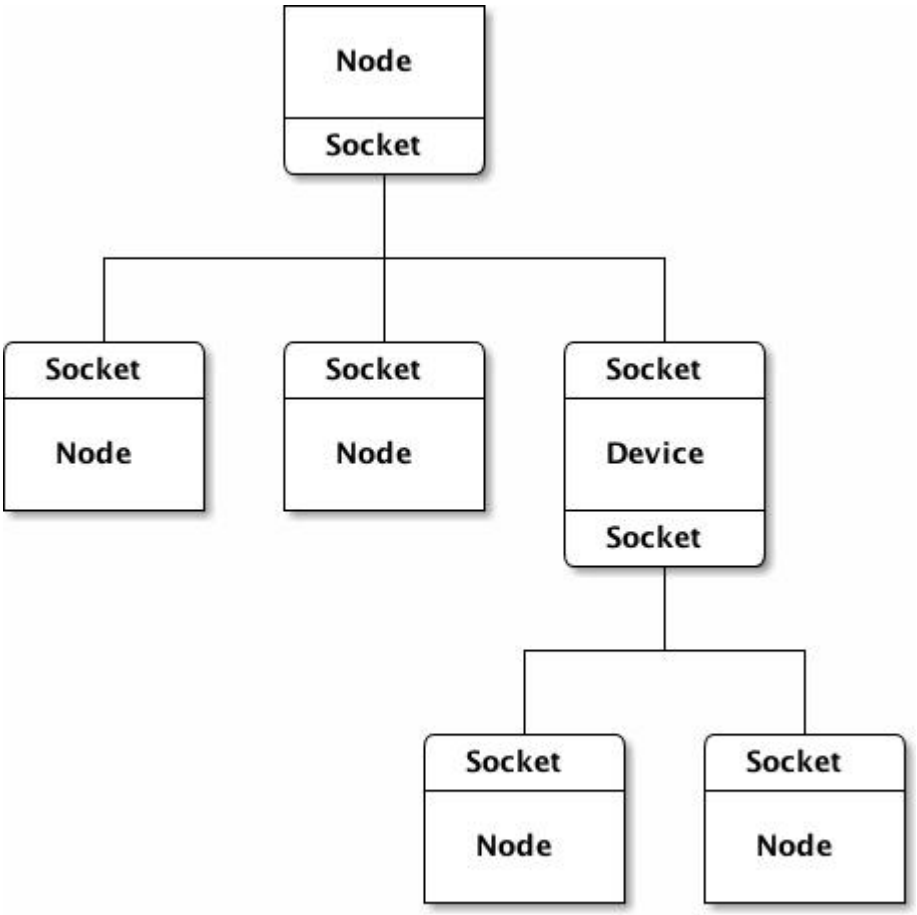

Figure 7 — Larger scale 0MQ application

ZMQ 装置没有具体的设计规则，但一般会有一组"前端"端点和一组"后端"端点。装置是无状态的，因此可以被广泛地部署在网络中。**你可以在进程中启动一个线程来运行装置，或者直接在一个进程中运行装置**。ZMQ 内部也提供了基本的装置实现可供使用。

ZMQ 装置可以用作路由和寻址、提供服务、队列调度、以及其他你所能想到的事情。不同的消息模式需要用到不同类型的装置来构建网络。如，请求-应答模式中可以使用队列装置、抽象服务；发布-订阅模式中则可使用流装置、主题装置等。

ZMQ 装置比起其他中间件的优势在于，你可以将它放在网络中任何一个地方，完成任何你想要的事情。

发布-订阅代理服务

我们经常会需要将发布-订阅模式扩充到不同类型的网络中。比如说，有一组订阅者是在外网上的，我们想用广播的方式发布消息给内网的订阅者，而用 TCP 协议发送给外网订阅者。

我们要做的就是写一个简单的代理服务装置，在发布者和外网订阅者之间搭起桥梁。这个装置有两个端点，一端连接内网上的发布者，另一端连接到外网上。它会从发布者处接收订阅的消息，并转发给外网上的订阅者们。

wuproxy: Weather update proxy in C

```c

//

// 气象信息代理服务装置

//

#include "zhelpers.h"

 int main (void)

{
```

```
void *context = zmq_init (1);

 // 订阅气象信息

void *frontend = zmq_socket (context, ZMQ_SUB);

zmq_connect (frontend, "tcp://192.168.55.210:5556");

 // 转发气象信息

void *backend = zmq_socket (context, ZMQ_PUB);

zmq_bind (backend, "tcp://10.1.1.0:8100");

 // 订阅所有消息

zmq_setsockopt (frontend, ZMQ_SUBSCRIBE, "", 0);

 // 转发消息

while (1) {

    while (1) {

        zmq_msg_t message;

        int64_t more;
```

```
        // 处理所有的消息帧

        zmq_msg_init (&message);

        zmq_recv (frontend, &message, 0);       //接受消息

        size_t more_size = sizeof (more);

        zmq_getsockopt (frontend, ZMQ_RCVMORE, &more,
&more_size);

        zmq_send (backend, &message, more? ZMQ_SNDMORE: 0);       //
转发消息

        zmq_msg_close (&message);

        if (!more)

                break; //  到达最后一帧

    }

}

// 程序不会运行到这里，但依然要正确地退出

zmq_close (frontend);
```

```
    zmq_close (backend);

    zmq_term (context);

    return 0;

}
```

```

我们称这个装置为代理，因为它既是订阅者，又是发布者。这就意味着，添加该装置时不需要更改其他程序的代码，只需让外网订阅者知道新的网络地址即可。
```

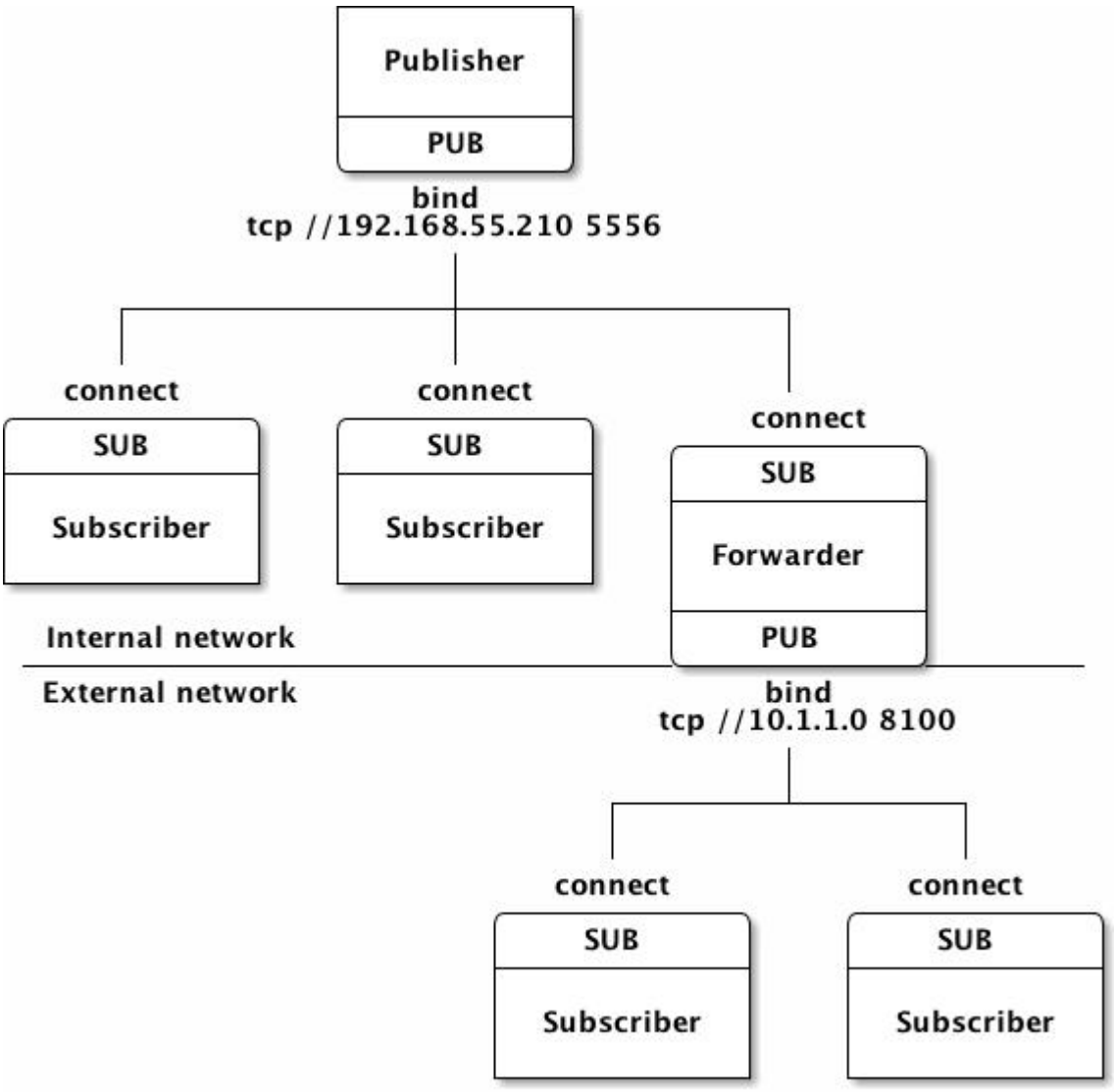

Figure 8 — Forwarder proxy device

可以注意到，这段程序能够正确处理多帧消息，会将它完整的转发给订阅者。**如果我们在发送时不指定 ZMQ_SNDMORE 选项，那么下游节点收到的消息就可能是破损的**。编写装置时应该要保证能够正确地处理多帧消息，否则会造成消息的丢失。

请求-应答 代理

下面让我们在请求-应答模式中编写一个小型的消息队列代理装置。

在 Hello World 客户/服务模型中，一个客户端和一个服务端进行通信。但在真实环境中，我们会需要让多个客户端和多个服务端进行通信。关键问题在于，服务端应该是无状态的，所有的状态都应该包含在一次请求中，或者存放其它介质中，如数据库。

我们有两种方式来连接多个客户端和多个服务端。第一种是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。比如说，有一个客户端连接了三个服务端，A、B、C，客户端产生了 R1、R2、R3、R4 四个请求，那么，R1 和 R4 会由服务 A 处理，R2 由 B 处理，R3 由 C 处理：

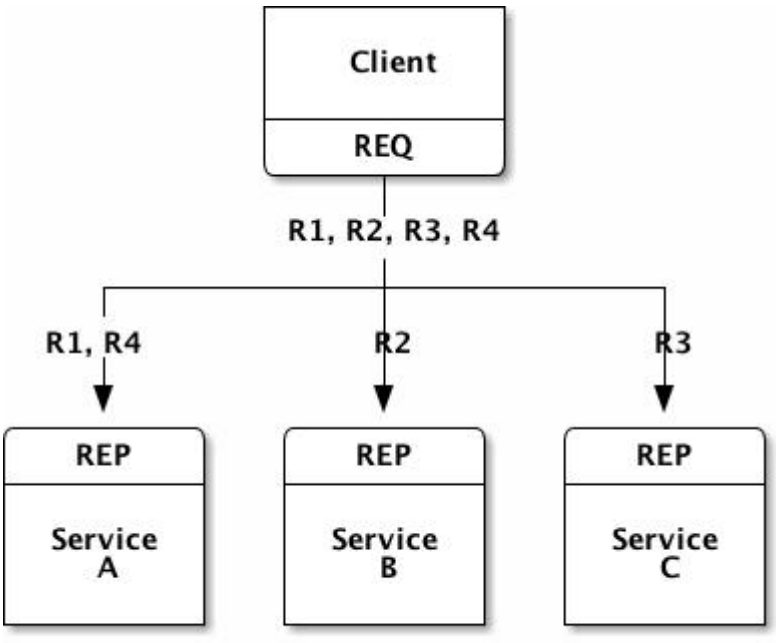

Figure 9 – Load balancing of requests

这种设计的好处在于可以方便地添加客户端，但若要添加服务端，那就得修改每个客户端的配置。如果你有 100 个客户端，需要添加三个服务端，那么这些客户端都需要重新进行配置，让其知道新服务端的存在。

这种方式肯定不是我们想要的。一个网络结构中如果有太多固化的模块就越不容易扩展。因此，**我们需要有一个模块位于客户端和服务端之间，将所有的知识都汇聚到这个网络拓扑结构中**。**理想状态下，我们可以任意地增减客户端或是服务端，不需要更改任何组件的配置**。

下面就让我们编写这样一个组件。这个代**理会绑定到两个端点，前端端点供客户端连接，后端端点供服务端连接**。它会使用 zmq_poll()来轮询这两个套接字，接收消息并进行转发。装置中不会有队列的存在，因为 ZMQ 已经自动在套接字中完成了。

在使用 REQ 和 REP 套接字时，其请求-应答的会话是严格同步。客户端发送请求，服务端接收请求并发送应答，由客户端接收。如果客户端或服务端中的一个发生问题（如连续两次发送请求），程序就会报错。

但是，我们的**代理装置必须要是非阻塞式的**，虽然可以使用 zmq_poll()同时处理两个套接字，但这里显然不能使用 REP 和 REQ 套接字。

幸运的是，我们有 **DEALER 和 ROUTER 套接字可以胜任这项工作，进行非阻塞的消息收发**。DEALER 过去被称为 XREQ，ROUTER 被称为 XREP，但新的代码中应尽量使用 DEALER/ROUTER 这种名称。在第三章中你会看到如何用 DEALER 和 ROUTER 套接字构建不同类型的请求-应答模式。

下面就让我们看看 DEALER 和 ROUTER 套接字是怎样在装置中工作的。

下方的简图描述了一个请求-应答模式，REQ 和 ROUTER 通信，DEALER 再和 REP 通信。ROUTER 和 DEALER 之间我们则需要进行消息转发：

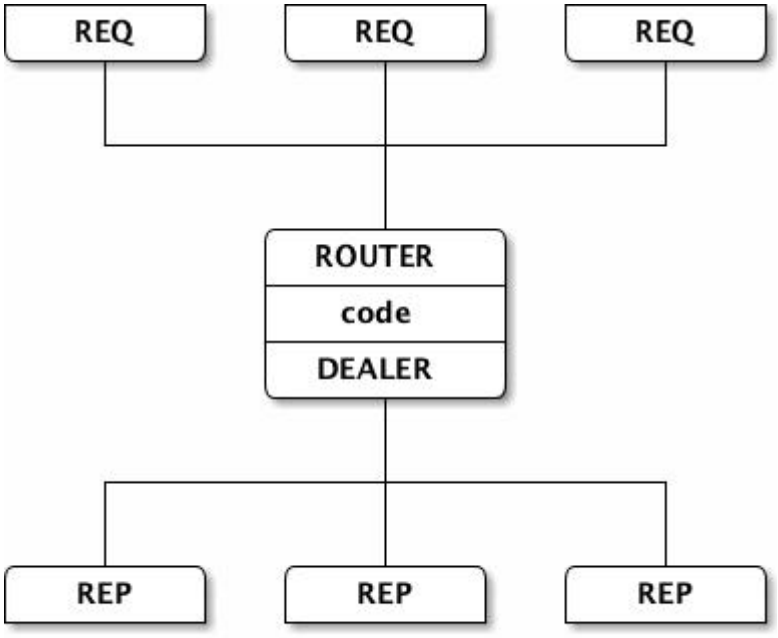

Figure 10 – Extended request reply

请求-应答代理会将两个套接字分别绑定到前端和后端，供客户端和服务端套接字连接。在使用该装置之前，还需要对客户端和服务端的代码进行调整。

客户端：

```
// Hello world 客户端

// 连接 REQ 套接字至 tcp://localhost:5559 端点

// 发送 Hello 给服务端，等待 World 应答

//
```

```
#include "zhelpers.h"

 int main (void)

{

    void *context = zmq_init (1);

     // 用于和服务端通信的套接字

    void *requester = zmq_socket (context, ZMQ_REQ);

    zmq_connect (requester, "tcp://localhost:5559");

     int request_nbr;

    for (request_nbr = 0; request_nbr != 10; request_nbr++) {

        s_send (requester, "Hello");

        char *string = s_recv (requester);

        printf ("收到应答  %d [%s]\n", request_nbr, string);

        free (string);

    }
```

```
    zmq_close (requester);

    zmq_term (context);

    return 0;

}
```

```

## 服务代码：

**rrserver: Request-reply service in C**

```
// Hello World 服务端

// 连接 REP 套接字至 tcp://*:5560 端点

// 接收 Hello 请求，返回 World 应答

//

#include "zhelpers.h"

 int main (void)

{
```
```

```
void *context = zmq_init (1);

 // 用于何客户端通信的套接字

void *responder = zmq_socket (context, ZMQ_REP);

zmq_connect (responder, "tcp://localhost:5560");

 while (1) {

    // 等待下一个请求

    char *string = s_recv (responder);

    printf ("Received request: [%s]\n", string);

    free (string);

     // 做一些“工作”

    sleep (1);

     // 返回应答信息

    s_send (responder, "World");

}
```

```
    // 程序不会运行到这里，不过还是做好清理工作

    zmq_close (responder);

    zmq_term (context);

    return 0;

}
```

代理程序：

可以看到它是能够处理多帧消息的：

rrbroker: Request-reply broker in C

```c
//

// 简易请求-应答代理

//

#include "zhelpers.h"
```

```
int main (void)

{

    // 准备上下文和套接字

    void *context = zmq_init (1);

    void *frontend = zmq_socket (context, ZMQ_ROUTER);

    void *backend = zmq_socket (context, ZMQ_DEALER);

    zmq_bind (frontend, "tcp://*:5559");

    zmq_bind (backend, "tcp://*:5560");

     // 初始化轮询集合

    zmq_pollitem_t items [] = {

        { frontend, 0, ZMQ_POLLIN, 0 },

        { backend, 0, ZMQ_POLLIN, 0 }

    };
```

```
// 在套接字间转发消息

while (1) {

    zmq_msg_t message;

    int64_t more; // 检测多帧消息

     zmq_poll (items, 2, -1);

    if (items [0].revents & ZMQ_POLLIN) {

        while (1) {

            // 处理所有消息帧

            zmq_msg_init (&message);

            zmq_recv (frontend, &message, 0);

            size_t more_size = sizeof (more);

            zmq_getsockopt (frontend, ZMQ_RCVMORE, &more, &more_size);

            zmq_send (backend, &message, more? ZMQ_SNDMORE: 0);

            zmq_msg_close (&message);
```

```
        if (!more)
            break; //  最后一帧
    }
}
if (items [1].revents & ZMQ_POLLIN) {
    while (1) {
        //  处理所有消息帧
        zmq_msg_init (&message);
        zmq_recv (backend, &message, 0);
        size_t more_size = sizeof (more);
        zmq_getsockopt (backend, ZMQ_RCVMORE, &more, &more_size);
        zmq_send (frontend, &message, more? ZMQ_SNDMORE: 0);
        zmq_msg_close (&message);
        if (!more)
```

```
            break; //  最后一帧
        }
    }
}
// 程序不会运行到这里，不过还是做好清理工作
zmq_close (frontend);
zmq_close (backend);
zmq_term (context);
return 0;
}
```

“使用请求-应答代理”可以让你的 C/S 网络结构更易于扩展：客户端不知道服务端的存在，服务端不知道客户端的存在。网络中唯一稳定的组件是中间的代理装置：

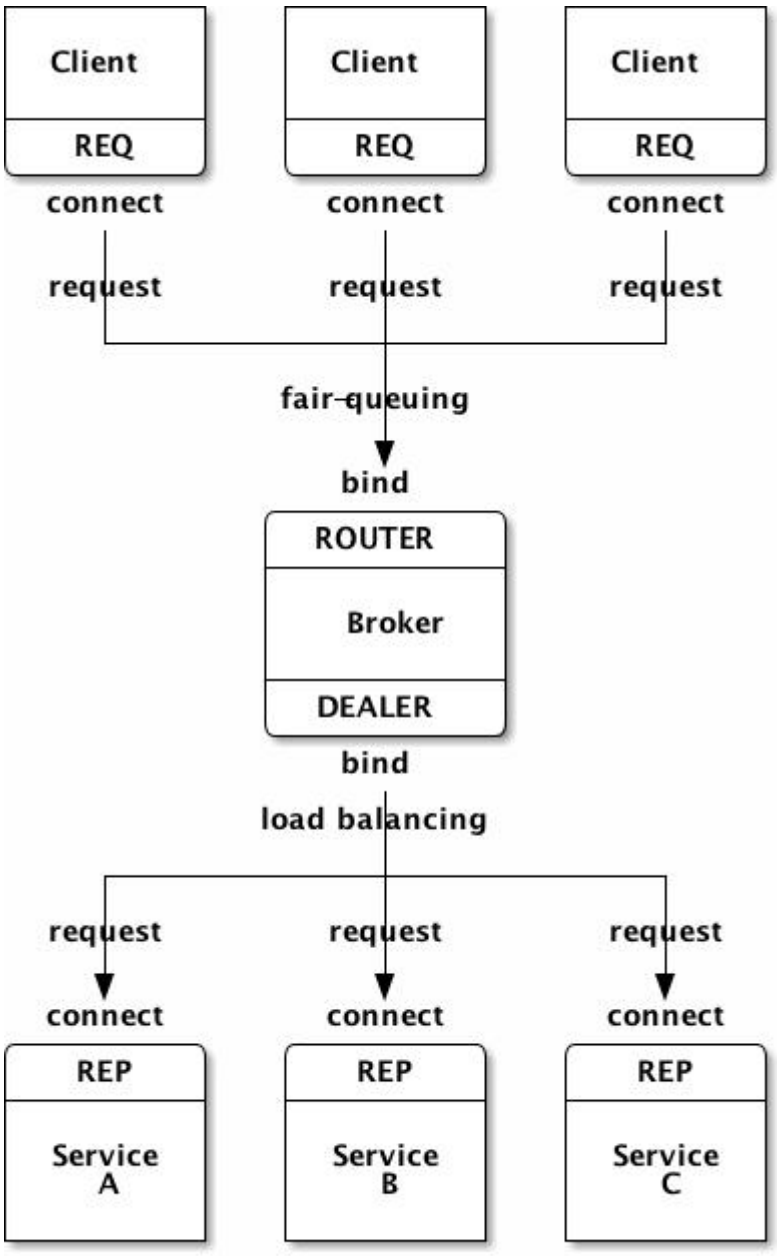

Figure 11 — Request reply broker

内置装置

ZMQ 提供了一些内置的装置，不过大多数人需要自己手动编写这些装置。内置装置有：

*** QUQUE，可用作请求-应答代理；**

*** FORWARDER，可用作发布-订阅代理服务；**

*** STREAMER，可用作管道模式代理。**

可以使用 zmq_device()来启动一个装置，需要传递两个套接字给它：

zmq_device(ZMQ_QUEUE, frontend, backend);

启动了 QUEUE 队列就如同在网络中加入了一个请求-应答代理，只需为其创建已绑定或连接的套接字即可。下面这段代码是使用内置装置的情形：

```
// 简单消息队列代理

// 功能和请求-应答代理相同，但使用了内置的装置

//

#include "zhelpers.h"
```

```
int main (void)

{

    void *context = zmq_init (1);

    // 客户端套接字

    void *frontend = zmq_socket (context, ZMQ_ROUTER);

    zmq_bind (frontend, "tcp://*:5559");

    // 服务端套接字

    void *backend = zmq_socket (context, ZMQ_DEALER);

    zmq_bind (backend, "tcp://*:5560");

    // 启动内置装置

    zmq_device (ZMQ_QUEUE, frontend, backend);

    // 程序不会运行到这里

    zmq_close (frontend);

    zmq_close (backend);
```

```
    zmq_term (context);

    return 0;

}
```

内置装置会恰当地处理错误，而我们手工实现的代理并没有加入错误处理机制。所以说，当你能够在程序中使用内置装置的时候就尽量用吧。

可能你会像某些 ZMQ 开发者一样提出这样一个问题：如果我将其他类型的套接字传入这些装置中会发生什么？答案是：别这么做。你可以随意传入不同类型的套接字，但是执行结果会非常奇怪。

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

当你需要其他的套接字类型进行组合时，那就需要自己编写装置了。

ZMQ 多线程编程

使用 ZMQ 进行多线程编程（MT 编程）将会是一种享受。在多线程中使用 ZMQ 套接字时，你不需要考虑额外的东西，让它们自如地运作就好。

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****。

什么叫"完美"的多线程编程，指的是代码易写易读，可以跨系统、跨语言地使用同一种技术，能够在任意颗核心的计算机上运行，没有状态，没有速度的瓶颈。

如果你有多年的多线程编程经验，知道如何使用锁、信号灯、临界区等机制来使代码运行得正确（尚未考虑快速），那你可能会很沮丧，因为 ZMQ 将改变这一切。**三十多年来，并发式应用程序开发所总结的经验是：不要共享状态**。这就好比两个醉汉想要分享一杯啤酒，如果他们不是铁哥们儿，那他们很快就会打起来。当有更多的醉汉加入时，情况就会更糟。多线程编程有时就像醉汉抢夺啤酒那样糟糕。

进行多线程编程往往是痛苦的，当程序因为压力过大而崩溃时，你会不知所然。有人写过一篇《**多线程代码中的 11 个错误易发点**》的文章，在大公司中广为流传，列举其中的几项：**没有进行同步、错误的粒度、读写分离、无锁排序、锁传递、优先级冲突**等。

假设某一天的下午三点，当证券市场正交易得如火如荼的时候，突然之间，应用程序因为锁的问题崩溃了，那将会是何等的场景？所以，作为程序员的我们，为解决那些复杂的多线程问题，只能用上更复杂的编程机制。

有人曾这样比喻，那些多线程程序原本应作为大型公司的核心支柱，但往往又最容易出错；那些想要通过网络不断进行延伸的产品，最后总以失败告终。

ZMQ 进行多线程编程规则：

* 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。

* **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**；

* 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**。

* 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。

当你需要在应用程序中使用两个装置时，可能会将套接字对象从一个线程传递给另一个线程，这样做一开始可能会成功，但最后一定会随机地发生错误。所以说，应在同一个线程中打开和关闭套接字。

如果你能遵循上面的规则，就会发现多线程程序可以很容易地拆分成多个进程。程序逻辑可以在线程、进程、或是计算机中运行，根据你的需求进行部署即可。

ZMQ 使用的是系统原生的线程机制，而不是某种"绿色线程"。这样做的好处是你不需要学习新的多线程编程 API，而且可以和目标操作系统进行很好的结合。你可以使用类似英特尔的 ThreadChecker 工具来查看线程工作的情况。缺点在于，如果程序创建了太多的线程（如上千个），则可能导致操作系统负载过高。

下面我们举一个实例，让原来的 Hello World 服务变得更为强大。原来的服务是单线程的，如果请求较少，自然没有问题。ZMQ 的线程可以在一个核心上高速地运行，执行大量的工作。但是，如果有一万次请求同时发送过来会怎么样？因此，现实环境中，我们会启动多个 worker 线程，他们会尽可能地接收客户端请求，处理并返回应答。

当然，我们可以使用启动多个 worker 进程的方式来实现，但是启动一个进程总比启动多个进程要来的方便且易于管理。而且，作为线程启动的 worker，所占用的带宽会比较少，延迟也会较低。

以下是多线程版的 Hello World 服务：

mtserver: Multithreaded service in C

```
// 多线程版 Hello World 服务

//

#include "zhelpers.h"

#include <pthread.h>

 //work 线程

static void *worker_routine (void *context) {

    // 连接至代理的套接字

    void *receiver = zmq_socket (context, ZMQ_REP);

    zmq_connect (receiver, "inproc://workers");

     while (1) {

        char *string = s_recv (receiver);

        printf ("Received request: [%s]\n", string);
```

```
        free (string);

        // 工作

        sleep (1);

        // 返回应答

        s_send (receiver, "World");

    }

    zmq_close (receiver);

    return NULL;

}

 int main (void)

{

    void *context = zmq_init (1);

     // 用于和 client 进行通信的套接字

    void *clients = zmq_socket (context, ZMQ_ROUTER);
```

```
zmq_bind (clients, "tcp://*:5555");

 // 用于和 worker 进行通信的套接字

void *workers = zmq_socket (context, ZMQ_DEALER);

zmq_bind (workers, "inproc://workers");

    // 启动一个 worker 池

int thread_nbr;

for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {

    pthread_t worker;

    pthread_create (&worker, NULL, worker_routine, context);

}

// 启动队列装置

zmq_device (ZMQ_QUEUE, clients, workers);

 // 程序不会运行到这里，但仍进行清理工作

zmq_close (clients);
```

```
    zmq_close (workers);

    zmq_term (context);

    return 0;

}
```

所有的代码应该都已经很熟悉了：

* 服务端启动一组 worker 线程，每个 worker 创建一个 REP 套接字，并处理收到的请求。worker 线程就像是一个单线程的服务，唯一的区别是使用了 inproc 而非 tcp 协议，以及绑定-连接的方向调换了。

* 服务端创建 ROUTER 套接字用以和 client 通信，因此提供了一个 TCP 协议的外部接口。

* 服务端创建 DEALER 套接字用以和 worker 通信，使用了内部接口（inproc）。

* 服务端启动了 QUEUE 内部装置，连接两个端点上的套接字。QUEUE 装置会将收到的请求分发给连接上的 worker，并将应答路由给请求方。

需要注意的是，在某些编程语言中，创建线程并不是特别方便，POSIX 提供的类库是 pthreads，但 Windows 中就需要使用不同的 API 了。我们会在第三章中讲述如何包装一个多线程编程的 API。

示例中的“工作”仅仅是 1 秒钟的停留，我们可以在 worker 中进行任意的操作，包括与其他节点进行通信。消息的流向是这样的：REQ-ROUTER-queue-DEALER-REP。

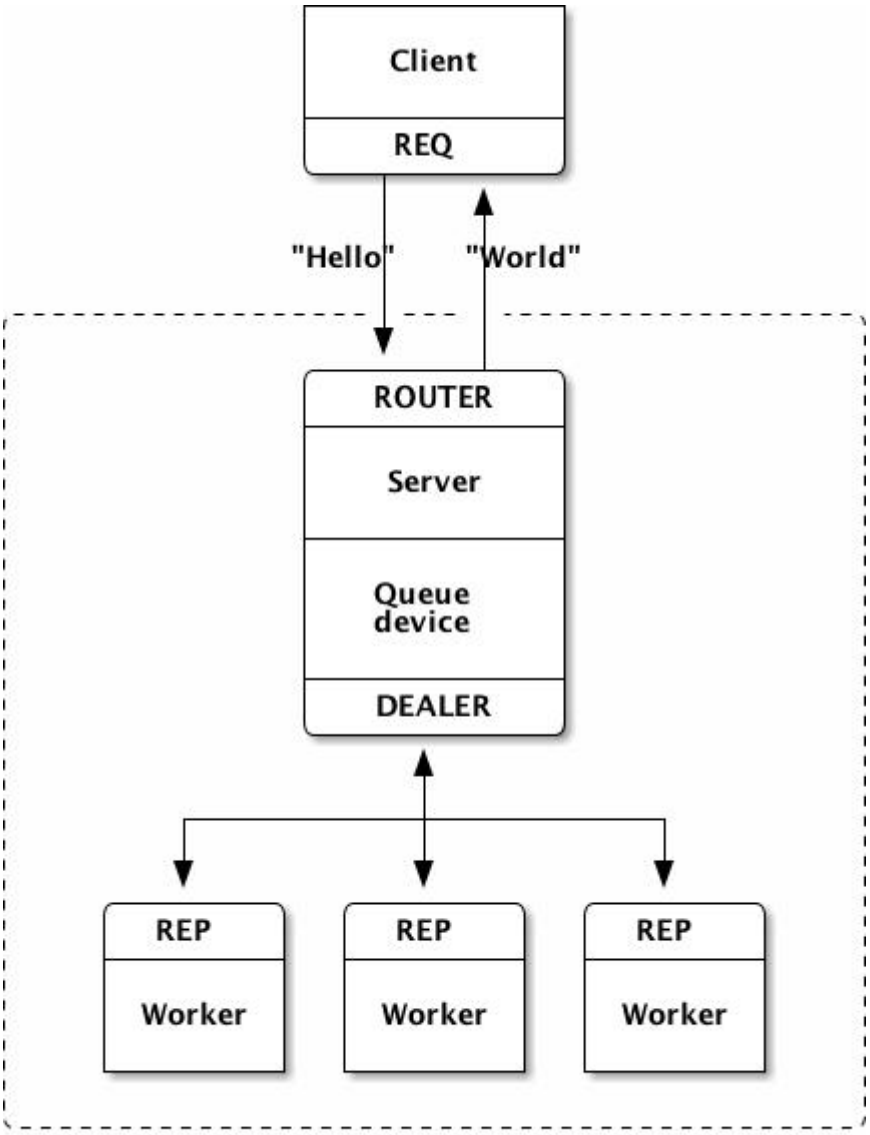

Figure 12 – Multithreaded server

线程间的信号传输

当你刚开始使用 ZMQ 进行多线程编程时，你可能会问：要如何协调两个线程的工作呢？可能会想要使用 sleep()这样的方法，或者使用诸如信号、互斥等机制。事实上，**你唯一要用的就是 ZMQ 本身**。回忆一下那个醉汉抢啤酒的例子吧。

下面的示例演示了三个线程之间需要如何进行同步：

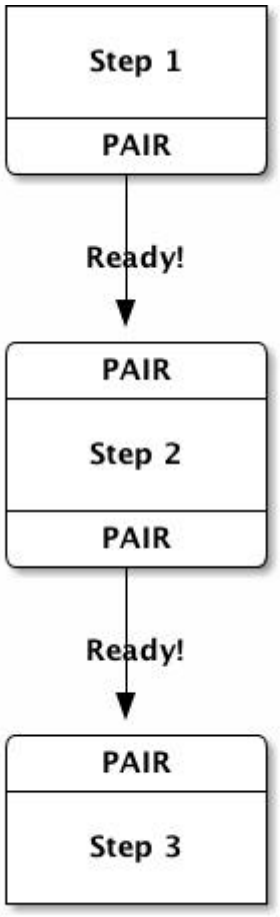

Figure 13 — The Relay Race

我们使用 PAIR 套接字和 inproc 协议。

多线程同步

```
//

#include "zhelpers.h"

#include <pthread.h>

 static void *step1 (void *context) {

    // 连接至步骤 2，告知我已就绪

    void *xmitter = zmq_socket (context, ZMQ_PAIR);

    zmq_connect (xmitter, "inproc://step2");

    printf ("步骤 1 就绪，正在通知步骤 2……\n");

    s_send (xmitter, "READY");

    zmq_close (xmitter);

     return NULL;

}
```

```
static void *step2 (void *context) {

    // 启动步骤 1 前线绑定至 inproc 套接字

    void *receiver = zmq_socket (context, ZMQ_PAIR);

    zmq_bind (receiver, "inproc://step2");

    pthread_t thread;

    pthread_create (&thread, NULL, step1, context);

     // 等待信号

    char *string = s_recv (receiver);

    free (string);

    zmq_close (receiver);

     // 连接至步骤 3，告知我已就绪

    void *xmitter = zmq_socket (context, ZMQ_PAIR);

    zmq_connect (xmitter, "inproc://step3");

    printf ("步骤 2 就绪，正在通知步骤 3……\n");
```

```
    s_send (xmitter, "READY");

    zmq_close (xmitter);

     return NULL;

}

 int main (void)

{

    void *context = zmq_init (1);

     // 启动步骤 2 前线绑定至 inproc 套接字

    void *receiver = zmq_socket (context, ZMQ_PAIR);

    zmq_bind (receiver, "inproc://step3");

    pthread_t thread;

    pthread_create (&thread, NULL, step2, context);

     // 等待信号

      char *string = s_recv (receiver);
```

```
    free (string);

    zmq_close (receiver);

     printf ("测试成功！\n");

    zmq_term (context);

    return 0;

}
```

这是一个 ZMQ 多线程编程的典型示例：

1. **两个线程通过 inproc 协议进行通信，使用同一个上下文**；

1. 父线程创建一个套接字，绑定至 inproc://端点，然后再启动子线程，将上下文对象传递给它；

1. 子线程创建第二个套接字，连接至 inproc://端点，然后发送已就绪信号给父线程。

需要注意的是，这段代码无法扩展到多个进程之间的协调。如果你使用 inproc 协议，只能建立结构非常紧密的应用程序。在延迟时间必须严格控制的情况下可以使用这种方法。对其他应用程序来说，每个线程使用同一个上下文，协议选用 ipc 或 tcp。然后，你就可以自由地将应用程序拆分为多个进程甚至是多台计算机了。

这是我们第一次使用 PAIR 套接字。为什么要使用 PAIR？其他类型的套接字也可以使用，但都有一些缺点会影响到线程间的通信：

* 你可以让信号发送方使用 **PUSH**，接收方使用 **PULL**，这看上去可能可以，但是需要注意的是，**PUSH 套接字发送消息时会进行负载均衡**，如果你不小心开启了两个接收方，就会"丢失"一半的信号。**而 PAIR 套接字建立的是一对一的连接，具有排他性**。

* **可以让发送方使用 DEALER，接收方使用 ROUTER**。但是，**ROUTER 套接字会在消息的外层包裹一个来源地址，这样一来原本零字节的信号就可能要成为一个多段消息了**。如果你不在乎这个问题，并且不会重复读取那个套接字，自然可以使用这种方法。但是，如果你想要使用这个套接字接收真正的数据，你就会发现 ROUTER 提供的消息是错误的。**至于 DEALER 套接字，它同样有负载均衡的机制，和 PUSH 套接字有相同的风险**。

* 可以让发送方使用 PUB，接收方使用 SUB。一来消息可以照原样发送，**二来 PUB 套接字不会进行负载均衡**。但是，**你需要对 SUB 套接字设置一个空的订阅信息（用以接收所有消息）；而且，如果 SUB 套接字没有及时和 PUB 建立连接，消息很有可能会丢失**。

综上，**使用 PAIR 套接字进行线程间的协调是最合适的**。

节点协调

当你想要对节点进行协调时，PAIR 套接字就不怎么合适了，这也是线程和节点之间的不同之处。一般来说，节点是来去自由的，而线程则较为稳定。**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**。

第二个区别在于，线程的数量一般是固定的，而节点数量则会经常变化。让我们以气象信息模型为基础，看看要怎样进行节点的协调，以保证客户端不会丢失最开始的那些消息。

下面是程序运行逻辑：

* 发布者知道预期的订阅者数量，这个数字可以任意指定；

* 发布者启动后会先等待所有订阅者进行连接，也就是节点协调。每个订阅者会使用另一个套接字来告知发布者自己已就绪；

* 当所有订阅者准备就绪后，发布者才开始发送消息。

这里我们会使用 REQ-REP 套接字来同步发布者和订阅者。发布者的代码如下：

发布者 - 同步版

```
//

#include "zhelpers.h"

// 等待 10 个订阅者连接

#define SUBSCRIBERS_EXPECTED 10

int main (void)

{

    void *context = zmq_init (1);
```

```
// 用于和客户端通信的套接字

void *publisher = zmq_socket (context, ZMQ_PUB);

zmq_bind (publisher, "tcp://*:5561");

// 用于接收信号的套接字

void *syncservice = zmq_socket (context, ZMQ_REP);

zmq_bind (syncservice, "tcp://*:5562");

// 接收订阅者的就绪信号

printf ("正在等待订阅者就绪\n");

int subscribers = 0;

while (subscribers < SUBSCRIBERS_EXPECTED) {

    // - 等待就绪信息

    char *string = s_recv (syncservice);

    free (string);

    // - 发送应答
```

```
        s_send (syncservice, "");
        subscribers++;
    }
    // 开始发送 100 万条数据
    printf ("正在广播消息\n");
    int update_nbr;
    for (update_nbr = 0; update_nbr < 1000000; update_nbr++)
        s_send (publisher, "Rhubarb");
     s_send (publisher, "END");
     zmq_close (publisher);
    zmq_close (syncservice);
    zmq_term (context);
    return 0;
}
```

```

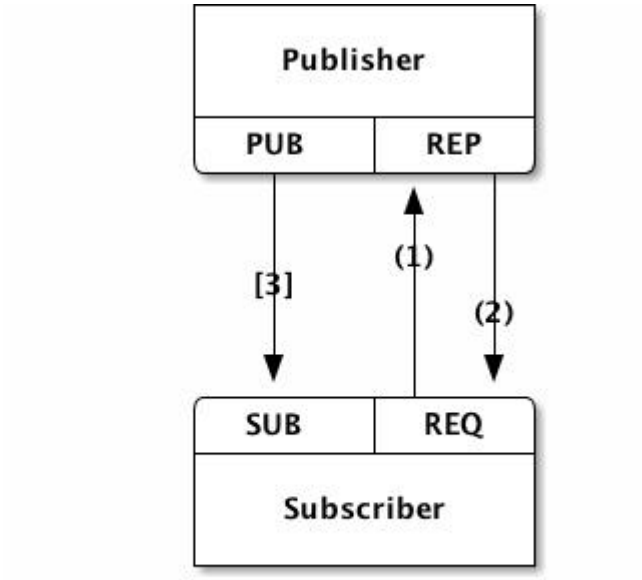


**Figure 14 — Pub Sub Synchronization**

以下是订阅者的代码：

## 订阅者 - 同步版

```
//

#include "zhelpers.h"

 int main (void)

{

    void *context = zmq_init (1);
```
```

```
// 一、连接 SUB 套接字

void *subscriber = zmq_socket (context, ZMQ_SUB);

zmq_connect (subscriber, "tcp://localhost:5561");

zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

// ZMQ 太快了，我们延迟一会儿.....    等待 SUB PUB 建立连接

sleep (1);

// 二、与发布者进行同步

void *syncclient = zmq_socket (context, ZMQ_REQ);

zmq_connect (syncclient, "tcp://localhost:5562");



// - 发送请求

s_send (syncclient, "");



// - 等待应答
```

```
char *string = s_recv (syncclient);

free (string);

 // 三、处理消息

int update_nbr = 0;

while (1) {

    char *string = s_recv (subscriber);

    if (strcmp (string, "END") == 0) {

        free (string);

        break;

    }

    free (string);

    update_nbr++;

}

printf ("收到 %d 条消息\n", update_nbr);
```

```
    zmq_close (subscriber);

    zmq_close (syncclient);

    zmq_term (context);

    return 0;

}
```

以下这段 shell 脚本会启动 10 个订阅者、1 个发布者：

```sh
echo "正在启动订阅者..."

for a in 1 2 3 4 5 6 7 8 9 10; do

    syncsub &

done

echo "正在启动发布者..."

syncpub
```

```

结果如下：

```

正在启动订阅者...

正在启动发布者...

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

收到 1000000 条消息

```

当REQ-REP请求完成时，我们仍无法保证SUB套接字已成功建立连接。除非使用 inproc 协议，否则对外连接的顺序是不一定的。因此，示例程序中使用了 sleep(1)的方式来进行处理，随后再发送同步请求。

更可靠的模型可以是：

* 发布者打开 PUB 套接字，开始发送 Hello 消息（非数据）；

* 订阅者连接 SUB 套接字，当收到 Hello 消息后再使用 REQ-REP 套接字进行同步；

* 当发布者获得所有订阅者的同步消息后，才开始发送真正的数据。

## ### 零拷贝(只能在发送做，不能再接收做)

第一章中我们曾提过零拷贝是很危险的，其实那是吓唬你的。既然你已经读到这里了，说明你已经具备了足够的知识，能够使用零拷贝。但需要记住，条条大路通地狱，过早地对程序进行优化其实是没有必要的。简单的说，如果你用不好零拷贝，那可能会让程序架构变得更糟。

**ZMQ 提供的 API 可以让你直接发送和接收消息，不用考虑缓存的问题。正因为消息是由 ZMQ 在后台收发的，所以使用零拷贝需要一些额外的工作。**

**做零拷贝时，使用 zmq_msg_init_data()函数创建一条消息，其内容指向某个已经分配好的内存区域，然后将该消息传递给 zmq_send()函数。创建消息**
```

时，你还需要提供一个用于释放消息内容的函数，**ZMQ** 会在消息发送完毕时调用。下面是一个简单的例子，我们假设已经分配好的内存区域为 1000 个字节：

```
void my_free (void *data, void *hint) {
    free (data);
}
// Send message from buffer, which we allocate and 0MQ will free for us
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
zmq_send (socket, &message, 0);
```

在接收消息的时候是无法使用零拷贝的：**ZMQ** 会将收到的消息放入一块内存区域供你读取，但不会将消息写入程序指定的内存区域。

ZMQ 的多段消息能够很好地支持零拷贝。在传统消息系统中，你需要将不同缓存中的内容保存到同一个缓存中，然后才能发送。但 **ZMQ** 会将来自不同内存区域的内容作为消息的一个帧进行发送。而且在 ZMQ 内部，一条消息会作为一个整体进行收发，因而非常高效。

瞬时套接字和持久套接字

在传统网络编程中，套接字是一个 API 对象，它们的生命周期不会长过程序的生命周期。但仔细打量一下套接字，它会占用一项特定的资源——缓存，这时 ZMQ 的开发者可能会问：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？

这种特性应该会非常有用，虽然不能应对所有的危险，但至少可以挽回一部分损失，特别是多发布-订阅模式来说。让我们来讨论一下。

这里有两个套接字正在欢快地传送着气象信息：

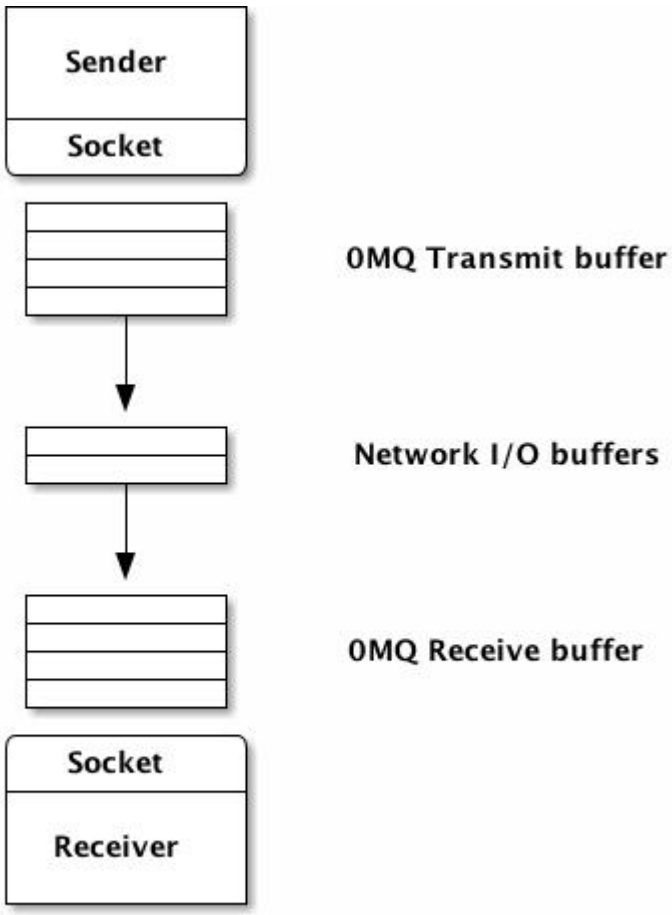

Figure 15 — Sender boring the pants off receiver

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。

需要注意，ZMQ 的套接字缓存对程序原来说是不可见的，正如 TCP 缓存一样。

到目前为止，我们使用的套接字都是瞬时套接字。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：

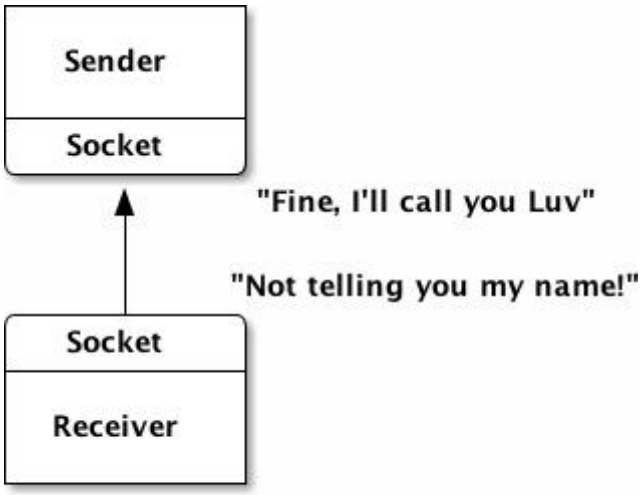

Figure 16 — Transient socket

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：

```

        +-----------+

        |                 |

        | Sender    |

        |                 |

        +-----------+

        | Socket    |

        \-----------/

            ^ "Lucy! Nice to see you again..."

            |

            |

            | "My name's Lucy"

        /-----+-----\
```

| Socket |

+----------+

| |

| Receiver |

| |

+----------+

Figure # - Durable socket

```

下面这行代码就可以为套接字设置标识，从而建立了一个持久套接字：

```c

**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。

其他信息可以查看 zmq_setsockopt()函数的 ZMQ_IDENTITY 一节。注意，该方法只能获取程序中套接字的标识，而不能获得对方套接字的标识。

发布-订阅消息“信封”

我们简单介绍了多帧消息，下面就来看看它的典型用法——**消息信封。信封是指为消息注明来源地址，而不修改消息内容**。

在发布-订阅模式中，信封包含了订阅信息，用以过滤掉不需要接收的消息。

如果你想要使用发布-订阅信封，就需要自行生成和设置。这个动作是可选的，我们在之前的示例中也没有使用到。在发布-订阅模式中使用信封可能会比较麻烦，但在现实应用中还是很有必要的，毕竟信封和消息的确是两块不想干的数据。

这是发布-订阅模式中一个带有信封的消息：

Figure 17 – Pub sub envelope with separate key

我们回忆一下，发布-订阅模式中，消息的接收是根据订阅信息来的，也就是消息的前缀。将这个前缀放入单独的消息帧，可以让匹配变得非常明显。因为不会有一个应用程序恰好只匹配了一部分数据。

下面是一个最简的发布-订阅消息信封示例。发布者会发送两类消息：A 和 B，信封中指明了消息类型：

发布-订阅消息信封 - 发布者：

```
// s_sendmore()函数也是 zhelpers.h 提供的

//
```

```
#include "zhelpers.h"

int main (void)

{

    // 准备上下文和 PUB 套接字

    void *context = zmq_init (1);

    void *publisher = zmq_socket (context, ZMQ_PUB);

    zmq_bind (publisher, "tcp://*:5563");

    while (1) {

        // 发布两条消息，A 类型和 B 类型

        s_sendmore (publisher, "A");

        s_send (publisher, "We don't want to see this");

                s_sendmore (publisher, "B");

        s_send (publisher, "We would like to see this");

        sleep (1);
```

```
    }

    // 正确退出

    zmq_close (publisher);

    zmq_term (context);

    return 0;

}
```

```

假设订阅者只需要 B 类型的消息：

## 发布-订阅消息信封 - 订阅者：

```
//

#include "zhelpers.h"

 int main (void)

{

    // 准备上下文和 SUB 套接字
```
```

```
void *context = zmq_init (1);

void *subscriber = zmq_socket (context, ZMQ_SUB);

zmq_connect (subscriber, "tcp://localhost:5563");

zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "B", 1);

 while (1) {

    // 读取消息信封

    char *address = s_recv (subscriber);

    // 读取消息内容

    char *contents = s_recv (subscriber);

    printf ("[%s] %s\n", address, contents);

    free (address);

    free (contents);

}

// 正确退出
```

```
    zmq_close (subscriber);

    zmq_term (context);

    return 0;

}
```

执行上面的程序时，订阅者会打印如下信息：

```
[B] We would like to see this

[B] We would like to see this

[B] We would like to see this

[B] We would like to see this

...
```

这个示例说明订阅者会丢弃未订阅的消息，且接收完整的多帧消息——你不会只获得消息的一部分。

如果你订阅了多个套接字，又想知道这些套接字的标识，从而通过另一个套接字来发送消息给它们（这个用例很常见），你可以让发布者创建一条含有三帧的消息：

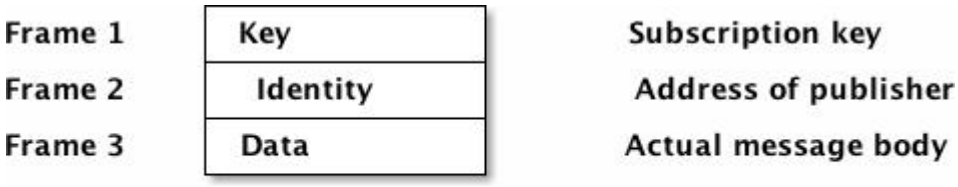

Figure 18 — Pub sub envelope with sender address

（半）持久订阅者和阈值（HWM）

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**。

这种机制有好有坏。好的地方在于发布者会暂存这些消息，当订阅者重连后进行发送；不好的地方在于这样很容易让发布者因内存溢出而崩溃。

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。

我们可以运行一个示例来证明这一点，用第一章中的 wuclient 和 wuserver 具体，在 wuclient 中进行套接字连接前加入这一行：

```
zmq_setsockopt (subscriber, ZMQ_IDENTITY, "Hello", 5);
```

编译并运行这两段程序，一切看起来都很平常。但是观察一下发布者的内存占用情况，可以看到当订阅者逐个退出后，发布者的内存占用会逐渐上升。若此时你重启订阅者，会发现发布者的内存占用不再增长了，一旦订阅者停止，就又会增长。很快地，它就会耗尽系统资源。

我们先来看看如何设置阈值，然后再看如何设置得正确。下面的发布者和订阅者使用了上文提到的"节点协调"机制。发布者会每隔一秒发送一条消息，这时你可以中断订阅者，重新启动它，看看会发生什么。

以下是发布者的代码：

发布者 - 连接持久化的订阅者：

```
//

#include "zhelpers.h"

 int main (void)

{

    void *context = zmq_init (1);

     // 订阅者会发送已就绪的消息

    void *sync = zmq_socket (context, ZMQ_PULL);

    zmq_bind (sync, "tcp://*:5564");
```

```
// 使用该套接字发布消息

void *publisher = zmq_socket (context, ZMQ_PUB);

zmq_bind (publisher, "tcp://*:5565");

// 等待同步消息

char *string = s_recv (sync);

free (string);

// 广播 10 条消息，一秒一条

int update_nbr;

for (update_nbr = 0; update_nbr < 10; update_nbr++) {

    char string [20];

    sprintf (string, "Update %d", update_nbr);

    s_send (publisher, string);

    sleep (1);

}
```

```
    s_send (publisher, "END");      // 发送 END 消息

     zmq_close (sync);

    zmq_close (publisher);

    zmq_term (context);

    return 0;

}
```

订阅者的代码：

```
// 持久化的订阅者

//

#include "zhelpers.h"

 int main (void)

{

    void *context = zmq_init (1);
```

```
// 连接 SUB 套接字

void *subscriber = zmq_socket (context, ZMQ_SUB);

zmq_setsockopt (subscriber, ZMQ_IDENTITY, "Hello", 5);

zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

zmq_connect (subscriber, "tcp://localhost:5565");

// 发送同步消息

void *sync = zmq_socket (context, ZMQ_PUSH);

zmq_connect (sync, "tcp://localhost:5564");

s_send (sync, "");

// 获取更新，并按指令退出

while (1) {

    char *string = s_recv (subscriber);

    printf ("%s\n", string);

    if (strcmp (string, "END") == 0) {
```

```
            free (string);
            break;
        }
        free (string);
    }
    zmq_close (sync);
    zmq_close (subscriber);
    zmq_term (context);
    return 0;
}
```

运行以上代码，在不同的窗口中先后打开发布者和订阅者。当订阅者获取了一至两条消息后按 Ctrl-C 中止，然后重新启动，看看执行结果：

```
$ durasub
Update 0
```

Update 1

Update 2

^C

$ durasub

Update 3

Update 4

Update 5

Update 6

Update 7

^C

$ durasub

Update 8

Update 9

END

```

可以看到订阅者的唯一区别是为套接字设置了标识，发布者就会将消息缓存起来，待重建连接后发送。设置套接字标识可以让瞬时套接字转变为持久套接字。实践中，你需要小心地给套接字起名字，可以从配置文件中获取，或者生成一个 UUID 并保存起来。

当我们为 PUB 套接字**设置了阈值，发布者就会缓存指定数量的消息，转而丢弃溢出的消息**。让我们将阈值设置为 2，看看会发生什么：

```c

uint64_t hwm = 2;

**zmq_setsockopt (publisher, ZMQ_HWM, &hwm, sizeof (hwm));**

```

运行程序，中断订阅者后等待一段时间再重启，可以看到结果如下：

```

$ durasub

Update 0

Update 1

^C
```
```

$ durasub

Update 2

Update 3

Update 7

Update 8

Update 9

END

```

看仔细了，发布者**只为我们保存了两条消息（2 和 3）**。阈值使得 ZMQ 丢弃溢出队列的消息。

简而言之，**如果你要使用持久化的订阅者，就必须在发布者端设置阈值，否则可能造成服务器因内存溢出而崩溃**。但是，还有另一种方法。ZMQ 提供了名为交换区（swap）的机制，它是一个磁盘文件，用于存放从队列中溢出的消息。启动它很简单：

```c

// 指定交换区大小，单位：字节。

uint64_t swap = 25000000;

**zmq_setsockopt (publisher, ZMQ_SWAP, &swap, sizeof (swap));**

```

我们可以将上面的方法综合起来，编写一个既能接受持久化套接字，又不至于内存溢出的发布者：

发布者 - 连接持久化订阅者：

```
//

#include "zhelpers.h"

 int main (void)

{

    void *context = zmq_init (1);

     // 订阅者会告知我们它已就绪

    void *sync = zmq_socket (context, ZMQ_PULL);

    zmq_bind (sync, "tcp://*:5564");

     // 使用该套接字发送消息

    void *publisher = zmq_socket (context, ZMQ_PUB);
```
```

```
// 避免慢持久化订阅者消息溢出的问题

uint64_t hwm = 1;

zmq_setsockopt (publisher, ZMQ_HWM, &hwm, sizeof (hwm));

// 设置交换区大小，供所有订阅者使用

uint64_t swap = 25000000;

zmq_setsockopt (publisher, ZMQ_SWAP, &swap, sizeof (swap));

zmq_bind (publisher, "tcp://*:5565");

// 等待同步消息

char *string = s_recv (sync);

free (string);

// 发布 10 条消息，一秒一条

int update_nbr;

for (update_nbr = 0; update_nbr < 10; update_nbr++) {

    char string [20];
```

```
        sprintf (string, "Update %d", update_nbr);

        s_send (publisher, string);

        sleep (1);

    }

    s_send (publisher, "END");

     zmq_close (sync);

    zmq_close (publisher);

    zmq_term (context);

    return 0;

}
```

若在现实环境中将阈值设置为 1，致使所有待发送的消息都保存到磁盘上，会大大降低处理速度。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（这超出了本指南的讲解范围）。

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 不会有接收队列。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。

这就是你想要的！

ZMQ 就像是一盒积木，只要你有足够的想象力，就可以用它组装出任何造型的网络架构。

这种高可扩、高弹性的架构一定会打开你的眼界。其实这并不是 ZMQ 原创的，早就有像[Erlang](http://www.erlang.org/)这样的[基于流的编程语言](http://en.wikipedia.org/wiki/Flow-based_programming)已经能够做到了，只是 ZMQ 提供了更为友善和易用的接口。

正如[Gonzo Diethelm](http://permalink.gmane.org/gmane.network.zeromq.devel/2145)所言："我想用一句话来总结，'如果 ZMQ 不存在，那它就应该被发明出来。'作为一个有着多年相关工作经验的人，ZMQ 太能引起我的共鸣了。我只能说，'这就是我想要的！'"