第三章 高级请求-应答模式

在第二章中我们通过开发一系列的小应用来熟悉ØMQ的基本使用方法，每个应用会引入一些新的特性。本章会沿用这种方式，来探索更多建立在ØMQ请求-应答模式之上的高级工作模式。

本章涉及的内容有：

- 在请求-应答模式中创建和使用消息信封
- 使用REQ、REP、DEALER和ROUTER套接字
- 使用标识来手工指定应答目标
- 使用自定义离散路由模式
- 使用自定义最近最少使用路由模式
- 构建高层消息封装类
- 构建基本的请求应答代理
- 合理命名套接字
- 模拟client-worker集群
- 构建可扩展的请求-应答集群云
- 使用管道套接字监控线程

Request-Reply Envelopes

在请求-应答模式中，信封里保存了应答目标的位置。这就是为什么ØMQ网络虽然是无状态的，但仍能完成请求-应答的过程。

在一般使用过程中，你并不需要知道请求-应答信封的工作原理。使用REQ、REP时，ØMQ会自动处理消息信封。下一章讲到的装置（device），使用时也只需保证读取和写入所有的信息即可。ØMQ使用多段消息的方式来存储信封，所以在复制消息时也会复制信封。

然而，在使用高级请求-应答模式之前是需要了解信封这一机制的，以下是信封机制在ROUTER中的工作原理：

- 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。
- 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。

如果将从ROUTER A中获取的消息（包含信封）写入ROUTER B（即将消息发送给一个DEALER，该DEALER连接到了ROUTER），那么在从ROUTER B中获取该消息时就会包含两层信封。

信封机制的根本作用是让ROUTER知道如何将消息递送给正确的应答目标，你需要做的就是在程序中保留好该信封。回顾一下REP套接字，它会将收到消息的信封逐个拆开，将消息本身传送给应用程序。而在发送时，又会在消息外层包裹该信封，发送给ROUTER，从而传递给正确的应答目标。

我们可以使用上述原理建立起一个ROUTER-DEALER装置：

```
[REQ] <--> [REP]
[REQ] <--> [ROUTER--DEALER] <--> [REP]
[REQ] <--> [ROUTER--DEALER] <--> [ROUTER--DEALER] <--> [REP]
...etc.
```

当你用REQ套接字去连接ROUTER套接字，并发送一条请求消息，你会从ROUTER中获得一条如下所示的消息：

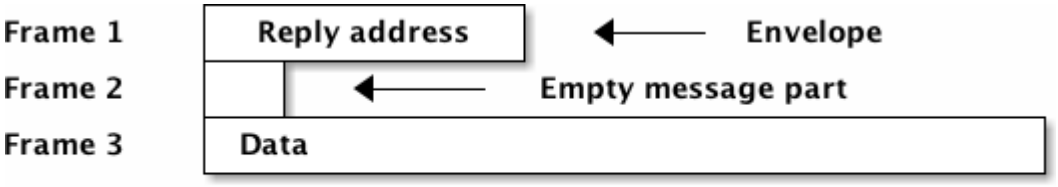

Figure 1 — Single hop request-reply envelope

- 第三帧是应用程序发送给REQ套接字的消息；
- 第二帧的空信息是REQ套接字在发送消息给ROUTER之前添加的；
- 第一帧即信封，是由ROUTER套接字添加的，记录了消息的来源。

如果我们在一条装置链路上传递该消息，最终会得到包含多层信封的消息。最新的信封会在消息的顶部。

Figure 2 — Multihop request-reply envelope

以下将详述我们在请求-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。

REP要求消息中的信封由一个空帧结束，所以如果你没有用REQ发送消息，则需要自己在消息中添加这个空帧。

你肯定会问，ROUTER是怎么标识消息的来源的？答案当然是套接字的标识。我们之前讲过，一个套接字可能是瞬时的，它所连接的套接字（如ROUTER）则会给它生成一个标识，与之相关联。一个套接字也可以显式地给自己定义一个标识，这样其他套接字就可以直接使用了。

这是一个瞬时的套接字，ROUTER会自动生成一个UUID来标识消息的来源。

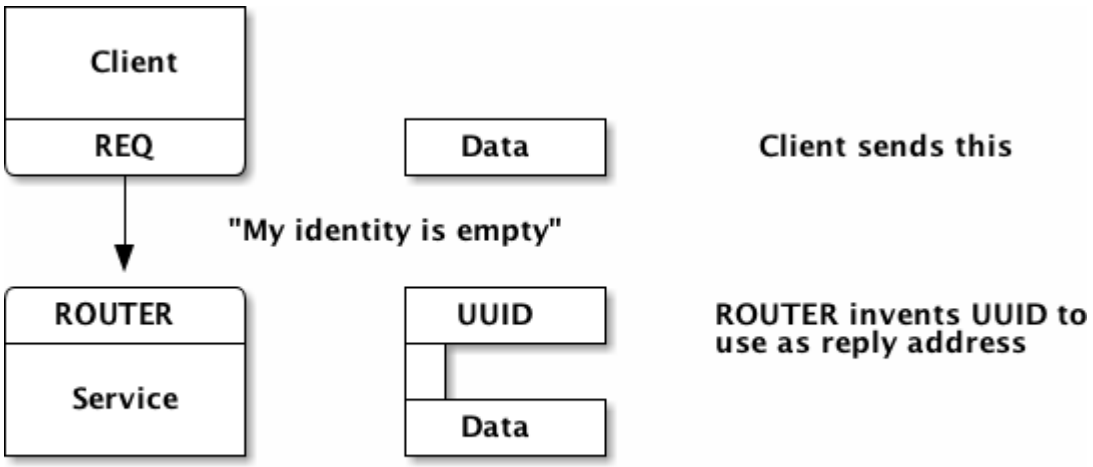

Figure 3 — ROUTER invents a UUID for transient sockets

这是一个持久的套接字，标识由消息来源自己指定。

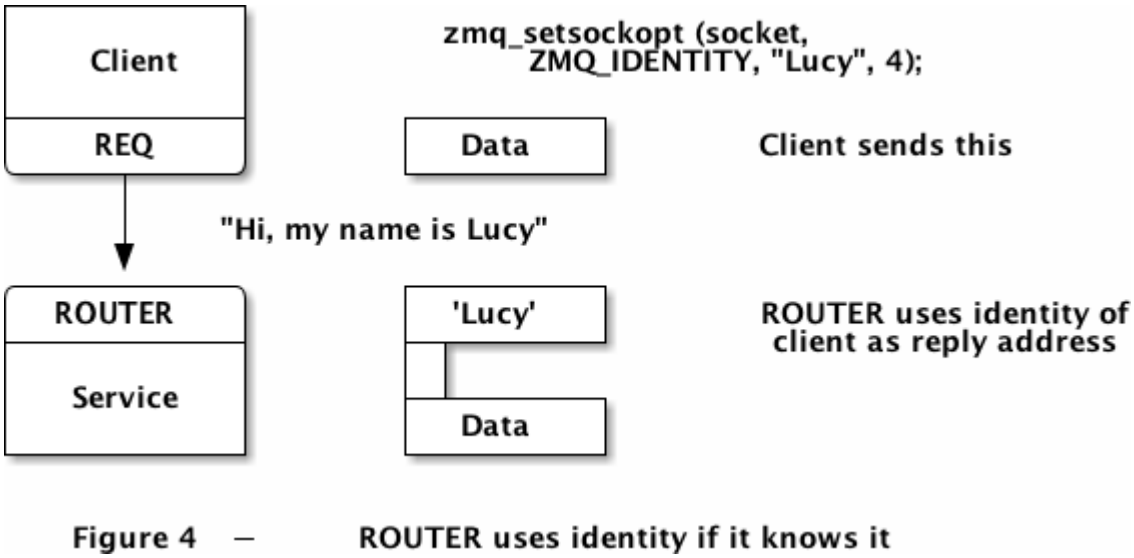

Figure 4 — ROUTER uses identity if it knows it

下面让我们在实例中观察上述两种操作。下列程序会打印出ROUTER从两个REP套接字中获得的消息，其中一个没有指定标识，另一个指定了“Hello”作为标识。

identity.c

```
//
// 以下程序演示了如何在请求-应答模式中使用套接字标识。
// 需要注意的是s_开头的函数是由zhelpers.h提供的。
// 我们没有必要重复编写那些代码。
//
#include "zhelpers.h"
```

```
int main (void)
{
    void *context = zmq_init (1);

    void *sink = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (sink, "inproc://example");

    // 第一个套接字由0MQ自动设置标识
    void *anonymous = zmq_socket (context, ZMQ_REQ);
    zmq_connect (anonymous, "inproc://example");
    s_send (anonymous, "ROUTER uses a generated UUID");
    s_dump (sink);

    // 第二个由自己设置
    void *identified = zmq_socket (context, ZMQ_REQ);
    zmq_setsockopt (identified, ZMQ_IDENTITY, "Hello", 5);
    zmq_connect (identified, "inproc://example");
    s_send (identified, "ROUTER socket uses REQ's socket identity");
    s_dump (sink);

    zmq_close (sink);
    zmq_close (anonymous);
    zmq_close (identified);
    zmq_term (context);
    return 0;
}
```

运行结果：

```
----------------------------------------
[017] 00314F043F46C441E28DD0AC54BE8DA727
[000]
[026] ROUTER uses a generated UUID
----------------------------------------
[005] Hello
[000]
[038] ROUTER socket uses REQ's socket identity
```

自定义请求-应答路由

我们已经看到ROUTER套接字是如何使用信封将消息发送给正确的应答目标的，下面我们从一个角度来定义ROUTER：在发送消息时使用一定格式的信封提供正确的路由目标，ROUTER就能够将该条消息异步地发送给对应的节点。

所以说ROUTER的行为是完全可控的。在深入理解这一特性之前，让我们先近距离观察一下REQ和REP套接字，赋予他们一些鲜活的角色：

- REQ是一个“妈妈”套接字，不会耐心听别人说话，但会不断地抛出问题寻求解答。REQ是严格同步的，它永远位于消息链路的请求端；
- REP则是一个“爸爸”套接字，只会回答问题，不会主动和别人对话。REP也是严格同步的，并一直位于应答端。

关于“妈妈”套接字，正如我们小时候所经历的，只能等她向你开口时你们才能对话。妈妈不像爸爸那么开明，也不会像DEALER套接字一样接受模棱两可的回答。所以，想和REQ套接字对话只有等它主动发出请求后才行，之后它就会一直等待你的回答，不管有多久。

“爸爸”套接字则给人一种强硬、冷漠的感觉，他只做一件事：无论你提出什么问题，都会给出一个精确的回答。不要期望一个REP套接字会主动和你对话或是将你俩的交谈传达给别人，它不会这么做的。

我们通常认为请求-应答模式一定是有来有往、有去有回的过程，但实际上这个过程是可以异步进行的。我们只需获得相应节点的地址，即可通过ROUTER套接字来异步地发送消息。ROUTER是ZMQ中唯一一个可以定位消息来源的套接字。

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会如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。

我们至少有三种模式来实现和ROUTER的连接：

- ROUTER-DEALER
- ROUTER-REQ
- ROUTER-REP

每种模式下我们都可以完全掌控消息的路由方式，但不同的模式会有不一样的应用场景和消息流，下一节开始我们会逐一解释。

自定义路由也有一些注意事项：

- 自定义路由让节点能够控制消息的去向，这一点有悖ØMQ的规则。使用自定义路由的唯一理由是ØMQ缺乏更多的路由算法供我们选择；
- 未来的ØMQ版本可能包含一些我们自定义的路由方式，这意味着我们现在设计的代码可能无法在新版本的ØMQ中运行，或者成为一种多余；
- 内置的路由机制是可扩展的，且对装置友好，但自定义路由就需要自己解决这些问题。

所以说自定义路由的成本是比较高的，更多情况下应当交由ØMQ来完成。不过既然我们已经讲到这儿了，就继续深入下去吧！

ROUTER-DEALER路由

ROUTER-DEALER是一种最简单的路由方式。将ROUTER和多个DEALER相连接，用一种合适的算法来决定如何分发消息给DEALER。DEALER可以是一个黑洞（只负责处理消息，不给任何返回）、代理（将消息转发给其他节点）或是服务（会发送返回信息）。

如果你要求DEALER能够进行回复，那就要保证只有一个ROUTER连接到DEALER，因为DEALER并不知道哪个特定的节点在联系它，如果有多个节点，它会做负载均衡，将消息分发出去。但如果DEALER是一个黑洞，那就可以连接任何数量的节点。

ROUTER-DEALER路由可以用来做什么呢？如果DEALER会将它完成任务的时间回复给ROUTER，那ROUTER就可以知道这个DEALER的处理速度有多快了。因为ROUTER和DEALER都是异步的套接字，所以我们要用zmq_poll()来处理这种情况。

下面例子中的两个DEALER不会返回消息给ROUTER，我们的路由采用加权随机算法：发送两倍多的信息给其中的一个DEALER。

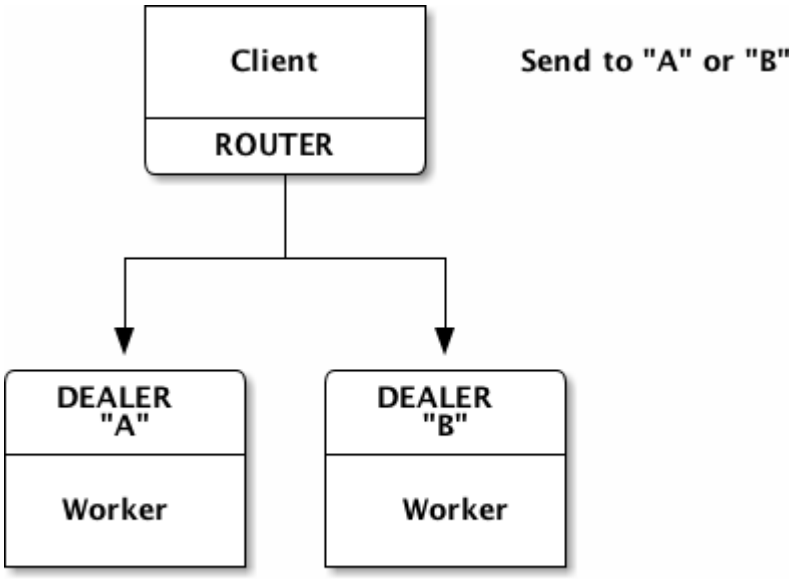

Figure 5 – Router to dealer custom routing

rtdealer.c

```
//
// 自定义ROUTER-DEALER路由
//
// 这个实例是单个进程，这样方便启动。
// 每个线程都有自己的ZMQ上下文，所以可以认为是多个进程在运行。
//
#include "zhelpers.h"
#include <pthread.h>

// 这里定义了两个worker，其代码是一样的。
//
static void *
worker_task_a (void *args)
{
```

```
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "A", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        // 我们只接受到消息的第二部分
        char *request = s_recv (worker);
        int finished = (strcmp (request, "END") == 0);
        free (request);
        if (finished) {
            printf ("A received: %d\n", total);
            break;
        }
        total++;
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

static void *
worker_task_b (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "B", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        // 我们只接受到消息的第二部分
        char *request = s_recv (worker);
        int finished = (strcmp (request, "END") == 0);
        free (request);
        if (finished) {
            printf ("B received: %d\n", total);
            break;
        }
        total++;
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}
```

```c
int main (void)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (client, "ipc://routing.ipc");

    pthread_t worker;
    pthread_create (&worker, NULL, worker_task_a, NULL);
    pthread_create (&worker, NULL, worker_task_b, NULL);

    // 等待线程连接至套接字，否则我们发送的消息将不能被正确路由
    sleep (1);

    // 发送10个任务，给A两倍多的量
    int task_nbr;
    srandom ((unsigned) time (NULL));
    for (task_nbr = 0; task_nbr < 10; task_nbr++) {
        // 发送消息的两个部分：第一部分是目标地址
        if (randof (3) > 0)
            s_sendmore (client, "A");
        else
            s_sendmore (client, "B");

        // 然后是任务
        s_send (client, "This is the workload");
    }
    s_sendmore (client, "A");
    s_send (client, "END");

    s_sendmore (client, "B");
    s_send (client, "END");

    zmq_close (client);
    zmq_term (context);
    return 0;
}
```

对上述代码的两点说明：

- ROUTER并不知道DEALER何时会准备好，我们可以用信号机制来解决，但为了不让这个例子太过复杂，我们就用sleep(1)的方式来处理。如果没有这句话，那ROUTER一开始发出的消息将无法被路由，ØMQ会丢弃这些消息。
- 需要注意的是，除了ROUTER会丢弃无法路由的消息外，PUB套接字当没有SUB连接它时也会丢弃发送出去的消息。其他套接字则会将无法发送的消息存储起来，直到有节点来处理它们。

在将消息路由给DEALER时，我们手工建立了这样一个信封：

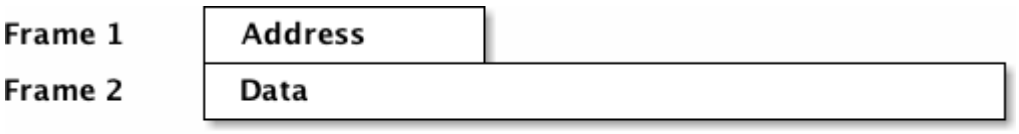

Figure 6 – Routing envelope for dealer

ROUTER套接字会移除第一帧，只将第二帧的内容传递给相应的DEALER。当DEALER发送消息给ROUTER时，只会发送一帧，ROUTER会在外层包裹一个信封（添加第一帧），返回给我们。

如果你定义了一个非法的信封地址，ROUTER会直接丢弃该消息，不作任何提示。对于这一点我们也无能为力，因为出现这种情况只有两种可能，一是要送达的目标节点不复存在了，或是程序中错误地指定了目标地址。如何才能知道消息会被正确地路由？唯一的方法是让路由目标发送一些反馈消息给我们。后面几章会讲述这一点。

DEALER的工作方式就像是PUSH和PULL的结合。但是，我们不能用PULL或PUSH去构建请求-应答模式。

最近最少使用算法路由（LRU模式）

我们之前讲过REQ套接字永远是对话的发起方，然后等待对方回答。这一特性可以让我们能够保持多个REQ套接字等待调配。换句话说，REQ套接字会告诉我们它已经准备好了。

你可以将ROUTER和多个REQ相连，请求-应答的过程如下：

- REQ发送消息给ROUTER
- ROUTER返回消息给REQ
- REQ发送消息给ROUTER
- ROUTER返回消息给REQ
- ...

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），其复杂度会超过你的想象并迫使你放弃的。

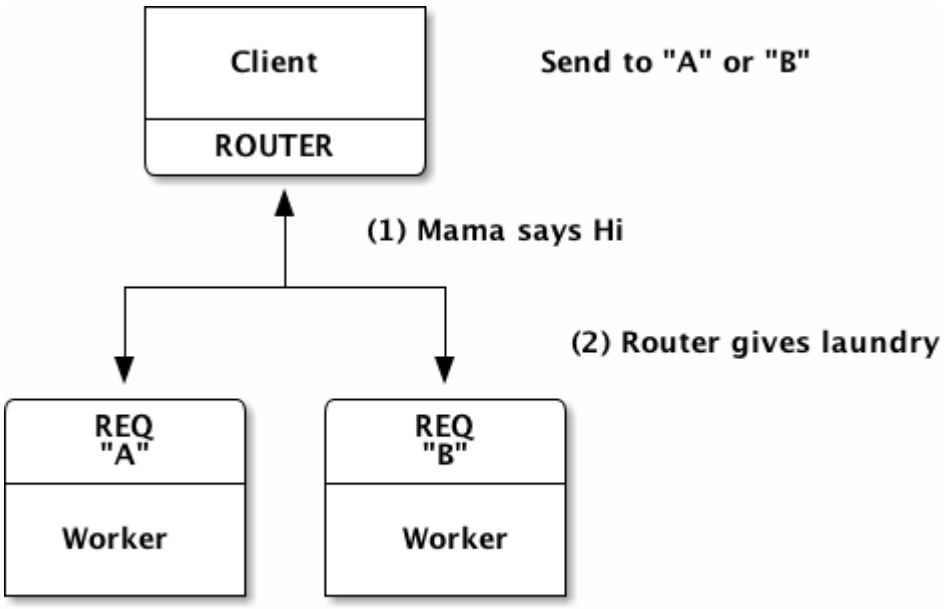

Figure 7 — Router to mama custom routing

ROUTER-REQ模式可以用来做什么？最常用的做法就是最近最少使用算法（LRU）路由了，ROUTER发出的请求会让等待最久的REQ来处理。请看示例：

```
//
// 自定义ROUTER-REQ路由
//
#include "zhelpers.h"
#include <pthread.h>

#define NBR_WORKERS 10

static void *
worker_task(void *args) {
    void *context = zmq_init(1);
    void *worker = zmq_socket(context, ZMQ_REQ);

    // s_set_id()函数会根据套接字生成一个可打印的字符串，
    // 并以此作为该套接字的标识。
    s_set_id(worker);
    zmq_connect(worker, "ipc://routing.ipc");

    int total = 0;
    while (1) {
        // 告诉ROUTER我已经准备好了
        s_send(worker, "ready");

        // 从ROUTER中获取工作，直到收到结束的信息
        char *workload = s_recv(worker);
        int finished = (strcmp(workload, "END") == 0);
```

```
        free(workload);
        if (finished) {
            printf("Processed: %d tasks\n", total);
            break;
        }
        total++;

        // 随机等待一段时间
        s_sleep(randof(1000) + 1);
    }
    zmq_close(worker);
    zmq_term(context);
    return NULL;
}

int main(void) {
    void *context = zmq_init(1);
    void *client = zmq_socket(context, ZMQ_ROUTER);
    zmq_bind(client, "ipc://routing.ipc");
    srandom((unsigned) time(NULL));

    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create(&worker, NULL, worker_task, NULL);
    }
    int task_nbr;
    for (task_nbr = 0; task_nbr < NBR_WORKERS * 10; task_nbr++) {
        // 最近最少使用的worker就在消息队列中
        char *address = s_recv(client);
        char *empty = s_recv(client);
        free(empty);
        char *ready = s_recv(client);
        free(ready);

        s_sendmore(client, address);
        s_sendmore(client, "");
        s_send(client, "This is the workload");
        free(address);
    }
    // 通知所有REQ套接字结束工作
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        char *address = s_recv(client);
        char *empty = s_recv(client);
        free(empty);
        char *ready = s_recv(client);
```

```
        free(ready);

        s_sendmore(client, address);
        s_sendmore(client, "");
        s_send(client, "END");
        free(address);
    }
    zmq_close(client);
    zmq_term(context);
    return 0;
}
```

在这个示例中，实现LRU算法并没有用到特别的数据结构，因为ØMQ的消息队列机制已经提供了等价的实现。一个更为实际的LRU算法应该将已准备好的worker收集起来，保存在一个队列中进行分配。以后我们会讲到这个例子。

程序的运行结果会将每个worker的执行次数打印出来。由于REQ套接字会随机等待一段时间，而我们也没有做负载均衡，所以我们希望看到的是每个worker执行相近的工作量。这也是程序执行的结果。

```
Processed: 8 tasks
Processed: 8 tasks
Processed: 11 tasks
Processed: 7 tasks
Processed: 9 tasks
Processed: 11 tasks
Processed: 14 tasks
Processed: 11 tasks
Processed: 11 tasks
Processed: 10 tasks
```

关于以上代码的几点说明：

- 我们不需要像前一个例子一样等待一段时间，因为REQ套接字会明确告诉ROUTER它已经准备好了。
- 我们使用了zhelpers.h提供的s_set_id()函数来为套接字生成一个可打印的字符串标识，这是为了让例子简单一些。在现实环境中，REQ套接字都是匿名的，你需要直接调用zmq_recv()和zmq_send()来处理消息，因为s_recv()和s_send()只能处理字符串标识的套接字。
- 更糟的是，我们使用了随机的标识，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。
- 如果你只是将上面的代码拷贝过来，没有充分理解，那你就像是看到蜘蛛人从屋顶上飞下来，你也照着做了，后果自负吧。

在将消息路由给REQ套接字时，需要注意一定的格式，即地址-空帧-消息：

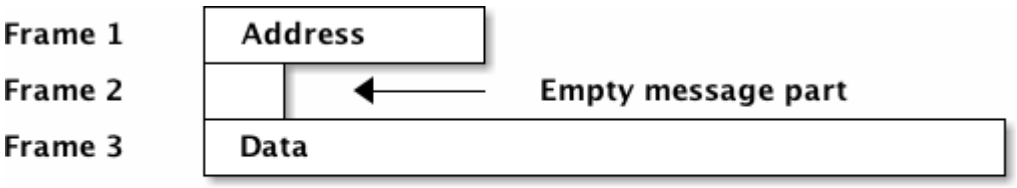

Figure 8 — Routing envelope for mama (REQ)

使用地址进行路由

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。

我们应该记住，ØMQ的经典模型往往是运行得最好的，毕竟人走得多的路往往是条好路，如果不按常理出牌，那很有可能会跌入无敌深潭。下面我们就将ROUTER和REP进行连接，看看会发生什么。

REP套接字有两个特点：

- 它需要完成完整的请求-应答周期；
- 它可以接受任意大小的信封，并能完整地返回该信封。

在一般的请求-应答模式中，REP是匿名的，可以随时替换。因为我们这里在将自定义路由，就要做到将一条消息发送给REP A，而不是REP B。这样才能保证网络的一端是你，另一端是特定的REP。

ØMQ的核心理念之一是周边的节点应该尽可能的智能，且数量众多，而中间件则是固定和简单的。这就意味着周边节点可以向其他特定的节点发送消息，比如可以连接到一个特定的REP。这里我们先不讨论如何在多个节点之间进行路由，只看最后一步中ROUTER如何和特定的REP通信的。

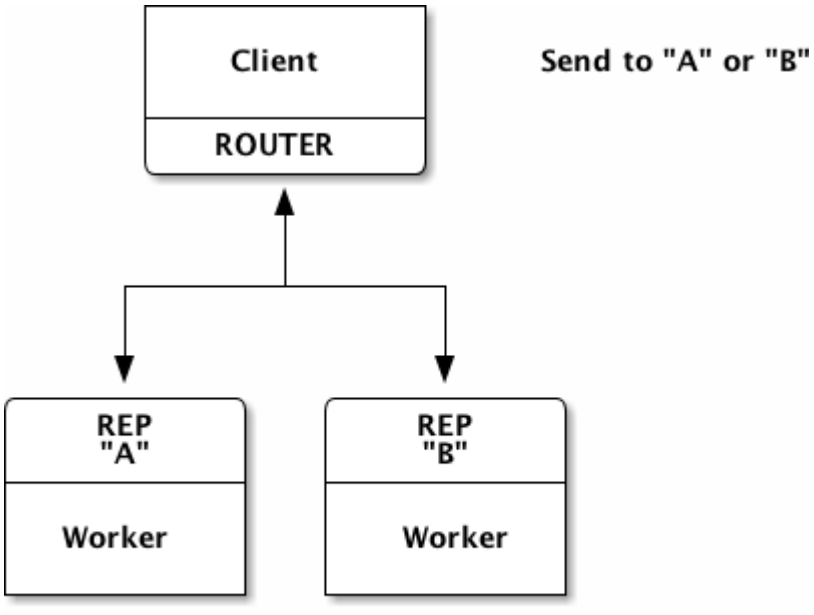

Figure 9 — Router to papa custom routing

这张图描述了以下事件：

- client有一条消息，将来会通过另一个ROUTER将该消息发送回去。这条信息包含了两个地址、一个空帧、以及消息内容；
- client将该条消息发送给了ROUTER，并指定了REP的地址；
- ROUTER将该地址移去，并以此决定其下哪个REP可以获得该消息；
- REP收到该条包含地址、空帧、以及内容的消息；
- REP将空帧之前的所有内容移去，交给worker去处理消息；
- worker处理完成后将回复交给REP；
- REP将之前保存好的信封包裹住该条回复，并发送给ROUTER；
- ROUTER在该条回复上又添加了一个注明REP的地址的帧。

这个过程看起来很复杂，但还是有必要取了解清楚的。只要记住，REP套接字会原封不动地将信封返回回去。

rtpapa.c

```
//
//  自定义ROUTER-REP路由
//
#include "zhelpers.h"

//  这里使用一个进程来强调事件发生的顺序性
int main (void)
{
    void *context = zmq_init (1);

    void *client = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (client, "ipc://routing.ipc");

    void *worker = zmq_socket (context, ZMQ_REP);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "A", 1);
    zmq_connect (worker, "ipc://routing.ipc");

    //  等待worker连接
    sleep (1);

    //  发送REP的标识、地址、空帧、以及消息内容
    s_sendmore (client, "A");
    s_sendmore (client, "address 3");
    s_sendmore (client, "address 2");
    s_sendmore (client, "address 1");
    s_sendmore (client, "");
    s_send     (client, "This is the workload");

    //  worker只会得到消息内容
    s_dump (worker);

    //  worker不需要处理信封
```

```
    s_send (worker, "This is the reply");

    //  看看ROUTER里收到了什么
    s_dump (client);

    zmq_close (client);
    zmq_close (worker);
    zmq_term (context);
    return 0;
}
```

运行结果

```
----------------------------------------
[020] This is the workload
----------------------------------------
[001] A
[009] address 3
[009] address 2
[009] address 1
[000]
[017] This is the reply
```

关于以上代码的几点说明：

- 在现实环境中，ROUTER和REP套接字处于不同的节点。本例没有启用多进程，为的是让事件的发生顺序更为清楚。

- zmq_connect()并不是瞬间完成的，REP和ROUTER连接的时候是会花费一些时间的。在现实环境中，ROUTER无从得知REP是否已经连接成功了，除非得到REP的某些回应。本例中使用sleep(1)来处理这一问题，如果不这样做，那REP将无法获得消息（自己尝试一下吧）。

- 我们使用REP的套接字标识来进行路由，如果你不信，可以将消息发送给B，看看A能不能收到。

- 本例中的s_dump()等函数来自于zhelpers.h文件，可以看到在进行套接字连接时代码都是一样的，所以我们才能在ØMQ API的基础上搭建上层的API。等今后我们讨论到复杂应用程序的时候再详细说明。

要将消息路由给REP，我们需要创建它能辨别的信封：

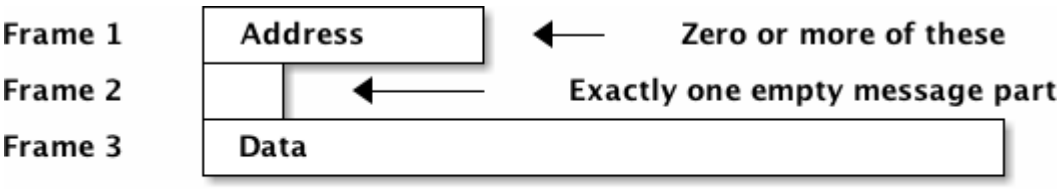

Figure 10 – Routing envelope for papa aka REP

请求-应答模式下的消息代理

这一节我们将对如何使用ØMQ消息信封做一个回顾，并尝试编写一个通用的消息代理装置。我们会建立一个队列装置来连接多个client和worker，装置的路由算法可以由我们自己决定。这里我们选择最近最少使用算法，因为这和负载均衡一样比较实用。

首先让我们回顾一下经典的请求-应答模型，尝试用它建立一个不断增长的巨型服务网络。最基本的请求-应答模型是：

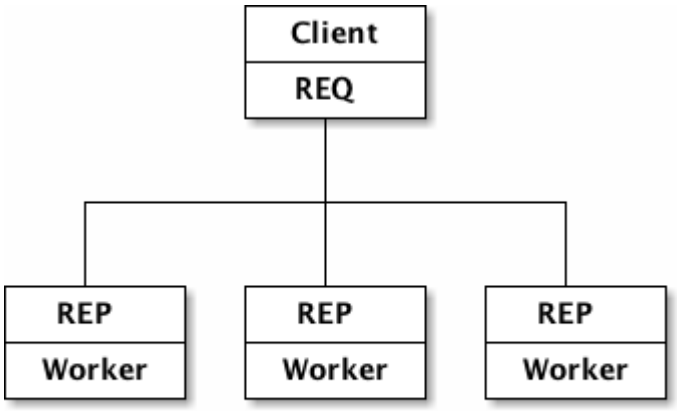

Figure 11 – Basic request reply

这个模型支持多个REP套接字，但如果我们想支持多个REQ套接字，就需要增加一个中间件，它通常是ROUTER和DEALER的结合体，简单将两个套接字之间的信息进行搬运，因此可以用现成的ZMQ_QUEUE装置来实现：

```
+--------+  +--------+  +--------+
| Client |  | Client |  | Client |
+--------+  +--------+  +--------+
|  REQ   |  |  REQ   |  |  REQ   |
+---+----+  +---+----+  +---+----+
    |           |           |
    +-----------+-----------+
                |
            +---+----+
            | ROUTER |
            +--------+
            | Device |
            +--------+
            | DEALER |
```

```
            +---+----+
                |
    +-----------+-----------+
    |           |           |
+---+----+  +---+----+  +---+----+
|  REP   |  |  REP   |  |  REP   |
+--------+  +--------+  +--------+
| Worker |  | Worker |  | Worker |
+--------+  +--------+  +--------+


Figure # - Stretched request-reply
```

这种结构的关键在于，ROUTER会将消息来自哪个REQ记录下来，生成一个信封。DEALER和REP套接字在传输消息的过程中不会丢弃或更改信封的内容，这样当消息返回给ROUTER时，它就知道应该发送给哪个REQ了。这个模型中的REP套接字是匿名的，并没有特定的地址，所以只能提供同一种服务。

上述结构中，对REP的路由我们使用了DEALER自带的负载均衡算法。但是，我们想用LRU算法来进行路由，这就要用到ROUTER-REP模式：

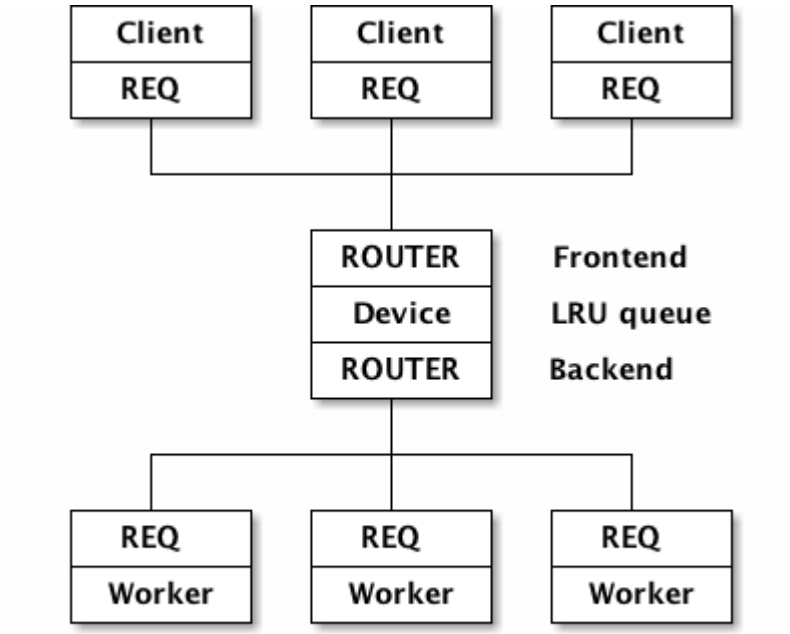

Figure 12 – Stretched request-reply with LRU

这个ROUTER-ROUTER的LRU队列不能简单地在两个套接字间搬运消息，以下代码会比较复杂，不过在请求-应答模式中复用性很高。

lruqueue.c

```
//
//  使用LRU算法的装置
//  client和worker处于不同的线程中
//
```

```c
#include "zhelpers.h"
#include <pthread.h>

#define NBR_CLIENTS 10
#define NBR_WORKERS 3

//  出队操作，使用一个可存储任何类型的数组实现
#define DEQUEUE(q) memmove (&(q)[0], &(q)[1], sizeof (q) - sizeof (q [0]))

//  使用REQ套接字实现基本的请求-应答模式
//  由于s_send()和s_recv()不能处理0MQ的二进制套接字标识，
//  所以这里会生成一个可打印的字符串标识。
//
static void *
client_task (void *args)
{
    void *context = zmq_init (1);
    void *client = zmq_socket (context, ZMQ_REQ);
    s_set_id (client);          //  设置可打印的标识
    zmq_connect (client, "ipc://frontend.ipc");

    //  发送请求并获取应答信息
    s_send (client, "HELLO");
    char *reply = s_recv (client);
    printf ("Client: %s\n", reply);
    free (reply);
    zmq_close (client);
    zmq_term (context);
    return NULL;
}

//  worker使用REQ套接字实现LRU算法
//
static void *
worker_task (void *args)
{
    void *context = zmq_init (1);
    void *worker = zmq_socket (context, ZMQ_REQ);
    s_set_id (worker);          //  设置可打印的标识
    zmq_connect (worker, "ipc://backend.ipc");

    //  告诉代理worker已经准备好
    s_send (worker, "READY");

    while (1) {
        //  将消息中空帧之前的所有内容（信封）保存起来，
```

```
        //  本例中空帧之前只有一帧，但可以有更多。
        char *address = s_recv (worker);
        char *empty = s_recv (worker);
        assert (*empty == 0);
        free (empty);

        //  获取请求，并发送回应
        char *request = s_recv (worker);
        printf ("Worker: %s\n", request);
        free (request);

        s_sendmore (worker, address);
        s_sendmore (worker, "");
        s_send     (worker, "OK");
        free (address);
    }
    zmq_close (worker);
    zmq_term (context);
    return NULL;
}

int main (void)
{
    //  准备0MQ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_ROUTER);
    zmq_bind (frontend, "ipc://frontend.ipc");
    zmq_bind (backend,  "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++) {
        pthread_t client;
        pthread_create (&client, NULL, client_task, NULL);
    }
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_task, NULL);
    }
    //  LRU逻辑
    //  - 一直从backend中获取消息；当有超过一个worker空闲时才从frontend获取消息。
    //  - 当woker回应时，会将该worker标记为已准备好，并转发woker的回应给client
    //  - 如果client发送了请求，就将该请求转发给下一个worker

    //  存放可用worker的队列
```

```
    int available_workers = 0;
    char *worker_queue [10];

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, available_workers? 2: 1, -1);

        //  处理backend中worker的队列
        if (items [0].revents & ZMQ_POLLIN) {
            //  将worker的地址入队
            char *worker_addr = s_recv (backend);
            assert (available_workers < NBR_WORKERS);
            worker_queue [available_workers++] = worker_addr;

            //  跳过空帧

            char *empty = s_recv (backend);
            assert (empty [0] == 0);
            free (empty);

            // 第三帧是“READY”或是一个client的地址
            char *client_addr = s_recv (backend);

            //  如果是一个应答消息，则转发给client
            if (strcmp (client_addr, "READY") != 0) {
                empty = s_recv (backend);
                assert (empty [0] == 0);
                free (empty);
                char *reply = s_recv (backend);
                s_sendmore (frontend, client_addr);
                s_sendmore (frontend, "");
                s_send     (frontend, reply);
                free (reply);
                if (--client_nbr == 0)
                    break;      //  处理N条消息后退出
            }
            free (client_addr);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取下一个client的请求，交给空闲的worker处理
            //  client请求的消息格式是：[client地址][空帧][请求内容]
            char *client_addr = s_recv (frontend);
            char *empty = s_recv (frontend);
```

```
            assert (empty [0] == 0);
            free (empty);
            char *request = s_recv (frontend);

            s_sendmore (backend, worker_queue [0]);
            s_sendmore (backend, "");
            s_sendmore (backend, client_addr);
            s_sendmore (backend, "");
            s_send     (backend, request);

            free (client_addr);
            free (request);

            //  将该worker的地址出队
            free (worker_queue [0]);
            DEQUEUE (worker_queue);
            available_workers--;
        }
    }
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

这段程序有两个关键点：1、各个套接字是如何处理信封的；2、LRU算法。我们先来看信封的格式。

我们知道REQ套接字在发送消息时会向头部添加一个空帧，接收时又会自动移除。我们要做的就是在传输消息时满足REQ的要求，处理好空帧。另外还要注意，ROUTER会在所有收到的消息前添加消息来源的地址。

现在我们就将完整的请求-应答流程走一遍，我们将client套接字的标识设为“CLIENT”，worker的设为“WORKER”。以下是client发送的消息：

Figure 13 — Message that client sends

代理从ROUTER中获取到的消息格式如下：

Figure 14 — Message coming in on frontend

代理会从LRU队列中获取一个空闲woker的地址，作为信封附加在消息之上，传送给ROUTER。注意要添加一个空帧。

Figure 15 — Message sent to backend

REQ（worker）收到消息时，会将信封和空帧移去：

Figure 16 — Message delivered to worker

可以看到，worker收到的消息和client端ROUTER收到的消息是一致的。worker需要将该消息中的信封保存起来，只对消息内容做操作。

在返回的过程中：

- worker通过REQ传输给device消息[client地址][空帧][应答内容]；
- device从worker端的ROUTER中获取到[worker地址][空帧][client地址][空帧][应答内容]；
- device将worker地址保存起来，并发送[client地址][空帧][应答内容]给client端的ROUTER；
- client从REQ中获得到[应答内容]。

然后再看看LRU算法，它要求client和worker都使用REQ套接字，并正确的存储和返回消息信封，具体如下：

- 创建一组poll，不断地从backend（worker端的ROUTER）获取消息；只有当有空闲的worker时才从frontend（client端的ROUTER）获取消息；
- 循环执行poll
- 如果backend有消息，只有两种情况：1）READY消息（该worker已准备好，等待分配）；2）应答消息（需要转发给client）。两种情况下我们都会保存worker的地址，放入LRU队列中，如果有应答内容，则转发给相应的client。
- 如果frontend有消息，我们从LRU队列中取出下一个worker，将该请求发送给它。这就需要发送[worker地址][空帧][client地址][空帧][请求内容]到worker端的ROUTER。

我们可以对该算法进行扩展，如在worker启动时做一个自我测试，计算出自身的处理速度，并随READY消息发送给代理，这样代理在分配工作时就可以做相应的安排。

ØMQ上层API的封装

使用ØMQ提供的API操作多段消息时是很麻烦的，如以下代码：

```
while (1) {
    //  将消息中空帧之前的所有内容（信封）保存起来，
    //  本例中空帧之前只有一帧，但可以有更多。
    char *address = s_recv (worker);
    char *empty = s_recv (worker);
    assert (*empty == 0);
    free (empty);

    //  获取请求，并发送回应
    char *request = s_recv (worker);
    printf ("Worker: %s\n", request);
    free (request);
    s_sendmore (worker, address);
    s_sendmore (worker, "");
    s_send     (worker, "OK");
    free (address);
}
```

这段代码不满足重用的需求，因为它只能处理一个帧的信封。事实上，以上代码已经做了一些封装了，如果调用ØMQ底层的API的话，代码就会更加冗长：

```
while (1) {
    //  将消息中空帧之前的所有内容（信封）保存起来，
    //  本例中空帧之前只有一帧，但可以有更多。
    zmq_msg_t address;
    zmq_msg_init (&address);
    zmq_recv (worker, &address, 0);

    zmq_msg_t empty;
    zmq_msg_init (&empty);
    zmq_recv (worker, &empty, 0);

    //  获取请求，并发送回应
    zmq_msg_t payload;
    zmq_msg_init (&payload);
    zmq_recv (worker, &payload, 0);

    int char_nbr;
```

```
    printf ("Worker: ");
    for (char_nbr = 0; char_nbr < zmq_msg_size (&payload); char_nbr++)
        printf ("%c", *(char *) (zmq_msg_data (&payload) + char_nbr));
    printf ("\n");

    zmq_msg_init_size (&payload, 2);
    memcpy (zmq_msg_data (&payload), "OK", 2);

    zmq_send (worker, &address, ZMQ_SNDMORE);
    zmq_close (&address);
    zmq_send (worker, &empty, ZMQ_SNDMORE);
    zmq_close (&empty);
    zmq_send (worker, &payload, 0);
    zmq_close (&payload);
}
```

我们理想中的API是可以一步接收和处理完整的消息，包括信封。ØMQ底层的API并不是为此而涉及的，但我们可以在它上层做进一步的封装，这也是学习ØMQ的过程中很重要的内容。

想要编写这样一个API还是很有难度的，因为我们要避免过于频繁地复制数据。此外，ØMQ用“消息”来定义多段消息和多段消息中的一部分，同时，消息又可以是字符串消息或者二进制消息，这也给编写API增加的难度。

解决方法之一是使用新的命名方式：字符串（s_send()和s_recv()中已经在用了）、帧（消息的一部分）、消息（一个或多个帧）。以下是用新的API重写的worker：

```
while (1) {
    zmsg_t *zmsg = zmsg_recv (worker);
    zframe_print (zmsg_last (zmsg), "Worker: ");
    zframe_reset (zmsg_last (zmsg), "OK", 2);
    zmsg_send (&zmsg, worker);
}
```

用4行代码代替22行代码是个不错的选择，而且更容易读懂。我们可以用这种理念继续编写其他的API，希望可以实现以下功能：

- 自动处理套接字。每次都要手动关闭套接字是很麻烦的事，手动定义过期时间也不是太有必要，所以，如果能在关闭上下文时自动关闭套接字就太好了。

- 便捷的线程管理。基本上所有的ØMQ应用都会用到多线程，但POSIX的多线程接口用起来并不是太方便，所以也可以封装一下。

- 便捷的时钟管理。想要获取毫秒数、或是暂停运行几毫秒都不太方便，我们的API应该提供这个接口。

- 一个能够替代zmq_poll()的反应器。poll循环很简单，但比较笨拙，会造成重复代码：计算时间、处理套接字中的信息等。若有一个简单的反应器来处理套接字的读写以及时间的控制，将会很方便。

- 恰当地处理Ctrl-C按键。我么已经看到如何处理中断了，最好这一机制可以用到所有的程序里。

我们可以用czmq来实现以上的需求。这个扩展很早就有了，提供了很多ØMQ的上层封装，甚至是数据结构（哈希、链表等）。

以下是用czmq重写的LRU代理：

lruqueue2.c

```
//
//  LRU消息队列装置，使用czmq库实现
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  worker准备就绪的信息

//  使用REQ套接字实现基本的请求-应答模式
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    //  发送请求并接收应答
    while (1) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字，实现LRU路由
//
```

```c
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  告知代理worker已准备就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  接收消息并处理
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  终止
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "ipc://frontend.ipc");
    zsocket_bind (backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  LRU逻辑
    //  - 一直从backend中获取消息；当有超过一个worker空闲时才从frontend获取消息。
    //  - 当woker回应时，会将该worker标记为已准备好，并转发woker的回应给client
    //  - 如果client发送了请求，就将该请求转发给下一个worker

    //  存放可用worker的队列
    zlist_t *workers = zlist_new ();
```

```
    while (1) {
        //  初始化poll
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  当有可用的worker时，从frontend获取消息
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  中断

        //  对backend发来的消息进行处理
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用worker的地址进行LRU路由
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);

            //  如果不是READY消息，则转发给client
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取client发来的请求，转发给worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  如果完成了，则进行一些清理工作
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

czmq提供了一个简单的中断机制，当按下Ctrl-C时程序会终止ØMQ的运行，并返回-1，errno设置为EINTR。程序中断时，czmq的recv方法会返回NULL，所以你可以用下面的代码来作判断：

```
while (1) {
    zstr_send (client, "HELLO");
    char *reply = zstr_recv (client);
    if (!reply)
        break; // 中断
    printf ("Client: %s\n", reply);
    free (reply);
    sleep (1);
}
```

如果使用zmq_poll()函数，则可以这样判断：

```
int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
if (rc == -1)
    break; // 中断
```

上例中还是使用了原生的zmq_poll()方法，也可以使用czmq提供的zloop反应器来实现，它可以做到：

- 从任意套接字上获取消息，也就是说只要套接字有消息就可以触发函数；
- 停止读取套接字上的消息；
- 设置一个时钟，定时地读取消息。

zloop内部当然是使用zmq_poll()实现的，但它可以做到动态地增减套接字上的监听器，重构poll池，并根据poll的超时时间来计算下一个时钟触发事件。

使用这种反应器模式后，我们的代码就更简洁了：

```
zloop_t *reactor = zloop_new ();
zloop_reader (reactor, self->backend, s_handle_backend, self);
zloop_start (reactor);
zloop_destroy (&reactor);
```

对消息的实际处理放在了程序的其他部分，并不是所有人都会喜欢这种风格，但zloop的确是将定时器和套接字的行为融合在了一起。在以后的例子中，我们会用zmq_poll()来处理简单的示例，使用zloop来处理复杂的。

下面我们用zloop来重写LRU队列装置

lruqueue3.c

```
//
//  LRU队列装置，使用czmq及其反应器模式实现
//
#include "czmq.h"
```

```c
#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  woker已准备就绪的消息

//  使用REQ实现基本的请求-应答模式
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://frontend.ipc");

    //  发送请求并接收应答
    while (1) {
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字来实现路由
//
static void *
worker_task (void *arg_ptr)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://backend.ipc");

    //  告诉代理worker已经准备就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  获取消息并处理
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断
```

```
        //zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  LRU队列处理器结构，将要传给反应器
typedef struct {
    void *frontend;             //  监听client
    void *backend;              //  监听worker
    zlist_t *workers;           //  可用的worker列表
} lruqueue_t;


//  处理frontend端的消息
int s_handle_frontend (zloop_t *loop, void *socket, void *arg)
{
    lruqueue_t *self = (lruqueue_t *) arg;
    zmsg_t *msg = zmsg_recv (self->frontend);
    if (msg) {
        zmsg_wrap (msg, (zframe_t *) zlist_pop (self->workers));
        zmsg_send (&msg, self->backend);

        //  如果没有可用的worker，则停止监听frontend
        if (zlist_size (self->workers) == 0)
            zloop_cancel (loop, self->frontend);
    }
    return 0;
}

//  处理backend端的消息
int s_handle_backend (zloop_t *loop, void *socket, void *arg)
{
    //  使用worker的地址进行LRU路由
    lruqueue_t *self = (lruqueue_t *) arg;
    zmsg_t *msg = zmsg_recv (self->backend);
    if (msg) {
        zframe_t *address = zmsg_unwrap (msg);
        zlist_append (self->workers, address);

        //  当有可用worker时增加frontend端的监听
        if (zlist_size (self->workers) == 1)
            zloop_reader (loop, self->frontend, s_handle_frontend, self);
```

```
        //  如果是worker发送来的应答，则转发给client
        zframe_t *frame = zmsg_first (msg);
        if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
            zmsg_destroy (&msg);
        else
            zmsg_send (&msg, self->frontend);
    }
    return 0;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    lruqueue_t *self = (lruqueue_t *) zmalloc (sizeof (lruqueue_t));
    self->frontend = zsocket_new (ctx, ZMQ_ROUTER);
    self->backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (self->frontend, "ipc://frontend.ipc");
    zsocket_bind (self->backend, "ipc://backend.ipc");

    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  可用worker的列表
    self->workers = zlist_new ();

    //  准备并启动反应器
    zloop_t *reactor = zloop_new ();
    zloop_reader (reactor, self->backend, s_handle_backend, self);
    zloop_start (reactor);
    zloop_destroy (&reactor);

    //  结束之后的清理工作
    while (zlist_size (self->workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (self->workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&self->workers);
    zctx_destroy (&ctx);
    free (self);
    return 0;
}
```

要正确处理Ctrl-C还是有点困难的，如果你使用zctx类，那它会自动进行处理，不过也需要代码的配合。若zmq_poll()返回了-1，或者recv方法（zstr_recv, zframe_recv, zmsg_recv）返回了NULL，就必须退出所有的循环。另外，在最外层循环中增加!zctx_interrupted的判断也很有用。

异步C/S结构

在之前的ROUTER-DEALER模型中，我们看到了client是如何异步地和多个worker进行通信的。我们可以将这个结构倒置过来，实现多个client异步地和单个server进行通信：

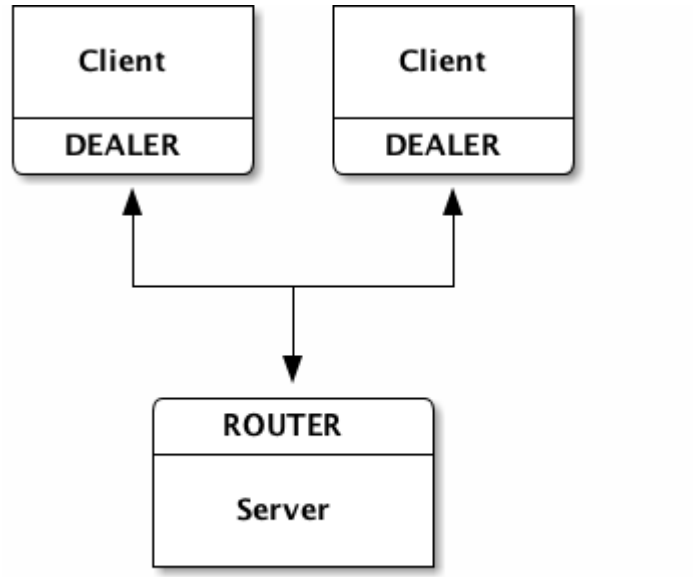

Figure 17 — Asynchronous Client Server

- client连接至server并发送请求；
- 每一次收到请求，server会发送0至N个应答；
- client可以同时发送多个请求而不需要等待应答；
- server可以同时发送多个应答而不需要新的请求。

asyncsrd.c

```
//
//  异步C/S模型（DEALER-ROUTER）
//

#include "czmq.h"

//  ---------------------------------------------------------------------
//  这是client端任务，它会连接至server，每秒发送一次请求，同时收集和打印应答消息。
//  我们会运行多个client端任务，使用随机的标识。

static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
```

```
    void *client = zsocket_new (ctx, ZMQ_DEALER);

    //  设置随机标识，方便跟踪
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zsockopt_set_identity (client, identity);
    zsocket_connect (client, "tcp://localhost:5570");

    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    int request_nbr = 0;
    while (1) {
        //  从poll中获取消息，每秒一次
        int centitick;
        for (centitick = 0; centitick < 100; centitick++) {
            zmq_poll (items, 1, 10 * ZMQ_POLL_MSEC);
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (client);
                zframe_print (zmsg_last (msg), identity);
                zmsg_destroy (&msg);
            }
        }
        zstr_sendf (client, "request #%d", ++request_nbr);
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  ---------------------------------------------------------------------
//  这是server端任务，它使用多线程机制将请求分发给多个worker，并正确返回应答信息。
//  一个worker只能处理一次请求，但client可以同时发送多个请求。

static void server_worker (void *args, zctx_t *ctx, void *pipe);

void *server_task (void *args)
{
    zctx_t *ctx = zctx_new ();

    //  frontend套接字使用TCP和client通信
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5570");

    //  backend套接字使用inproc和worker通信
    void *backend = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_bind (backend, "inproc://backend");

    //  启动一个worker线程池，数量任意
```

```
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++)
        zthread_fork (ctx, server_worker, NULL);

    //  使用队列装置连接backend和frontend，我们本来可以这样做：
    //      zmq_device (ZMQ_QUEUE, frontend, backend);
    //  但这里我们会自己完成这个任务，这样可以方便调试。

    //  在frontend和backend间搬运消息
    while (1) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { backend,  0, ZMQ_POLLIN, 0 }
        };
        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (frontend);
            //puts ("Request from client:");
            //zmsg_dump (msg);
            zmsg_send (&msg, backend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            //puts ("Reply from worker:");
            //zmsg_dump (msg);
            zmsg_send (&msg, frontend);
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

//  接收一个请求，随机返回多条相同的文字，并在应答之间做随机的延迟。
//
static void
server_worker (void *args, zctx_t *ctx, void *pipe)
{
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "inproc://backend");

    while (1) {
        //  DEALER套接字将信封和消息内容一起返回给我们
        zmsg_t *msg = zmsg_recv (worker);
        zframe_t *address = zmsg_pop (msg);
        zframe_t *content = zmsg_pop (msg);
        assert (content);
```

```
        zmsg_destroy (&msg);

        //  随机返回0至4条应答
        int reply, replies = randof (5);
        for (reply = 0; reply < replies; reply++) {
            //  暂停一段时间
            zclock_sleep (randof (1000) + 1);
            zframe_send (&address, worker, ZFRAME_REUSE + ZFRAME_MORE);
            zframe_send (&content, worker, ZFRAME_REUSE);
        }
        zframe_destroy (&address);
        zframe_destroy (&content);
    }
}


//  主程序用来启动多个client和一个server
//
int main (void)
{
    zctx_t *ctx = zctx_new ();
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, client_task, NULL);
    zthread_new (ctx, server_task, NULL);

    //  运行5秒后退出
    zclock_sleep (5 * 1000);
    zctx_destroy (&ctx);
    return 0;
}
```

运行上面的代码，可以看到三个客户端有各自的随机标识，每次请求会获得零到多条回复。

- client每秒会发送一次请求，并获得零到多条应答。这要通过zmq_poll()来实现，但我们不能只每秒poll一次，这样将不能及时处理应答。程序中我们每秒取100次，这样一来server端也可以以此作为一种心跳（heartbeat），用来检测client是否还在线。

- server使用了一个worker池，每一个worker同步处理一条请求。我们可以使用内置的队列来搬运消息，但为了方便调试，在程序中我们自己实现了这一过程。你可以将注释的几行去掉，看看输出结果。

这段代码的整体架构如下图所示：

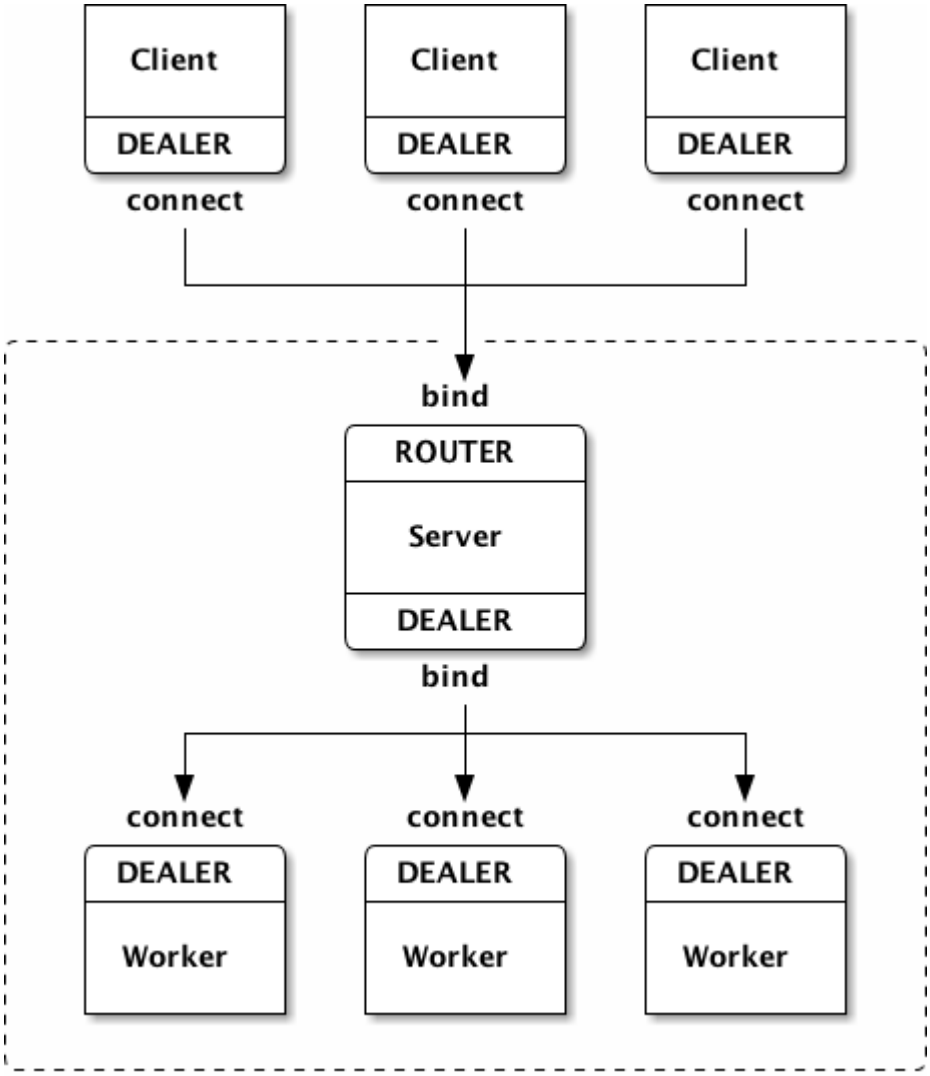

Figure 18 – Detail of asynchronous server

可以看到，client和server之间的连接我们使用的是DEALER-ROUTER，而server和worker的连接则用了DEALER-DEALER。如果worker是一个同步的线程，我们可以用REP。但是本例中worker需要能够发送多个应答，所以就需要使用DEALER这样的异步套接字。这里我们不需要对应答进行路由，因为所有的worker都是连接到一个server上的。

让我们看看路由用的信封，client发送了一条信息，server获取的信息中包含了client的地址，这样一来我们有两种可行的server-worker通信方案：

- worker收到未经标识的信息。我们使用显式声明的标识，配合ROUTER套接字来连接worker和server。这种设计需要worker提前告知ROUTER它的存在，这种LRU算法正是我们之前所讲述的。

- worker收到含有标识的信息，并返回含有标识的应答。这就要求worker能够处理好信封。

第二种涉及较为简单：

```
   client          server       frontend       worker
[ DEALER ]<---->[ ROUTER <----> DEALER <----> DEALER ]
```

```
            1 part         2 parts       2 parts
```

当我们需要在client和server之间维持一个对话时，就会碰到一个经典的问题：client是不固定的，如果给每个client都保存一些消息，那系统资源很快就会耗尽。即使是和同一个client保持连接，因为使用的是瞬时的套接字（没有显式声明标识），那每次连接也相当于是一个新的连接。

想要在异步的请求中保存好client的信息，有以下几点需要注意：

- client需要发送心跳给server。本例中client每秒都会发送一个请求给server，这就是一种很可靠的心跳机制。
- 使用client的套接字标识来存储信息，这对瞬时和持久的套接字都有效；
- 检测停止心跳的client，如两秒内没有收到某个client的心跳，就将保存的状态丢弃。

实战：跨代理路由

让我们把目前所学到的知识综合起来，应用到实战中去。我们的大客户今天打来一个紧急电话，说是要构建一个大型的云计算设施。它要求这个云架构可以跨越多个数据中心，每个数据中心包含一组client和worker，且能共同协作。

我们坚信实践高于理论，所以就提议使用ZMQ搭建这样一个系统。我们的客户同意了，可能是因为他的确也想降低开发的成本，或是在推特上看到了太多ZMQ的好处。

细节详述

喝完几杯特浓咖啡，我们准备着手干了，但脑中有个理智的声音提醒我们应该在事前将问题分析清楚，然后再开始思考解决方案。云到底要做什么？我们如是问，客户这样回答：

- worker在不同的硬件上运作，但可以处理所有类型的任务。每个集群都有成百个worker，再乘以集群的个数，因此数量众多。
- client向worker指派任务，每个任务都是独立的，每个client都希望能找到对应的worker来处理任务，越快越好。client是不固定的，来去频繁。
- 真正的难点在于，这个架构需要能够自如地添加和删除集群，附带着集群中的client和worker。
- 如果集群中没有可用的worker，它便会将任务分派给其他集群中可以用的worker。
- client每次发送一个请求，并等待应答。如果X秒后他们没有获得应答，他们会重新发送请求。这一点我们不需要多做考虑，client端的API已经写好了。
- worker每次处理一个请求，他们的行为非常简单。如果worker崩溃了，会有另外的脚本启动他们。

听了以上的回答，我们又进一步追问：

- 集群之间会有一个更上层的网络来连接他们对吗？客户说是的。

- 我们需要处理多大的吞吐量？客户说，每个集群约有一千个client，单个client每秒会发送10次请求。请求包含的内容很少，应答也很少，每个不超过1KB。

我们进行了简单的计算，2500个client x 10次/秒 x 1000字节 x 双向 = 50MB/秒，或400Mb/秒，这对1Gb网络来说不成问题，可以使用TCP协议。

这样需求就很清晰了，不需要额外的硬件或协议来完成这件事，只要提供一个高效的路由算法，设计得缜密一些。我们首先从一个集群（数据中心）开始，然后思考如何来连接他们。

单个集群的架构

worker和client是同步的，我们使用LRU算法来给worker分配任务。每个worker都是等价的，所以我们不需要考虑服务的问题。worker是匿名的，client不会和某个特定的worker进行通信，因而我们不需要保证消息的送达以及失败后的重试等。

鉴于上文提过的原因，client和worker是不会直接通信的，这样一来就无法动态地添加和删除节点了。所以，我们的基础模型会使用一个请求-应答模式中使用过的代理结构。

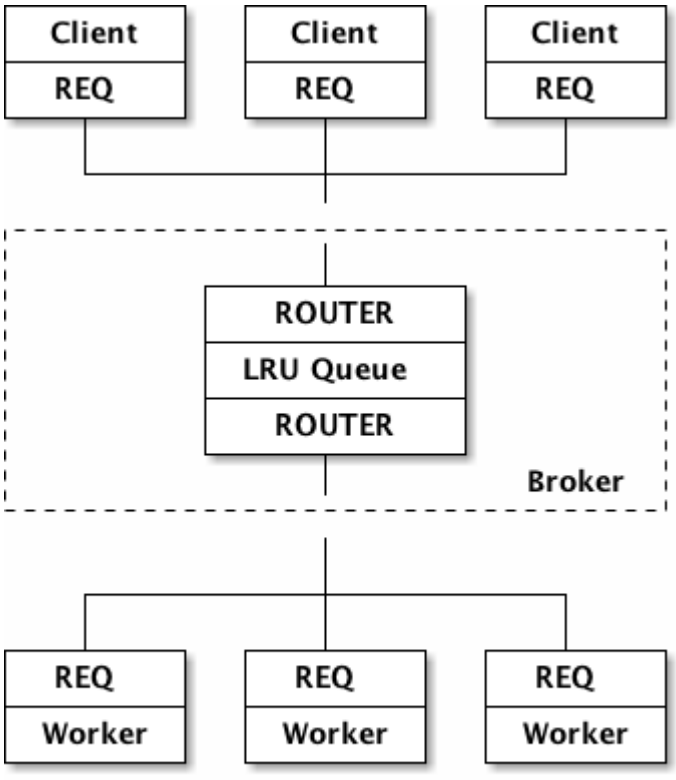

Figure 19 — Cluster architecture

多个集群的架构

下面我们将集群扩充到多个，每个集群有自己的一组client和worker，并使用代理相连接：

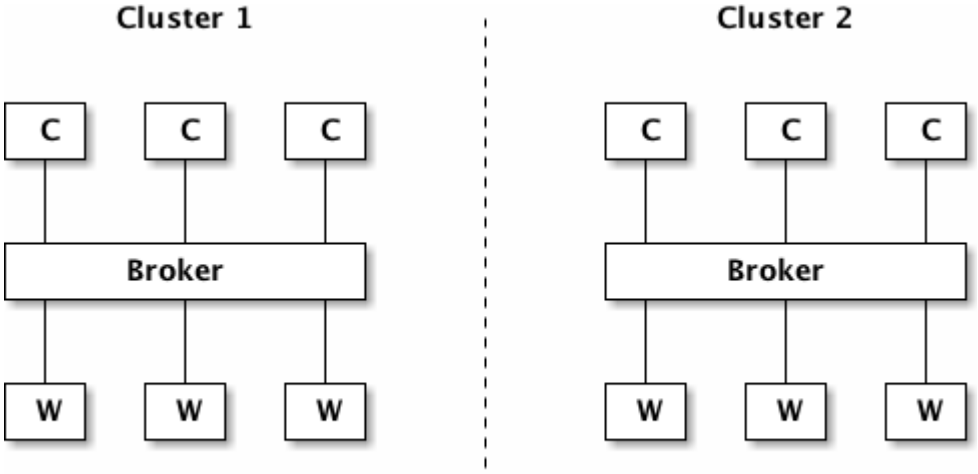

Figure 20 — Multiple clusters

问题在于：我们如何让一个集群的client和另一个集群的worker进行通信呢？有这样几种解决方案，我们来看看他们的优劣：

- client直接和多个代理相连接。优点在于我们可以不对代理和worker做改动，但client会变得复杂，并需要知悉整个架构的情况。如果我们想要添加第三或第四个集群，所有的client都会需要修改。我们相当于是将路由和容错功能写进client了，这并不是个好主意。

- worker直接和多个代理相连接。可是REQ类型的worker不能做到这一点，它只能应答给某一个代理。如果改用REP套接字，这样就不能使用LRU算法的队列代理了。这点肯定不行，在我们的结构中必须用LRU算法来管理worker。还有个方法是使用ROUTER套接字，让我们暂且称之为方案1。

- 代理之间可以互相连接，这看上去不错，因为不需要增加过多的额外连接。虽然我们不能随意地添加代理，但这个问题可以暂不考虑。这种情况下，集群中的worker和client不必理会整体架构，当代理有剩余的工作能力时便会和其他代理通信。这是方案2。

我们首先看看方案1，worker同时和多个代理进行通信：

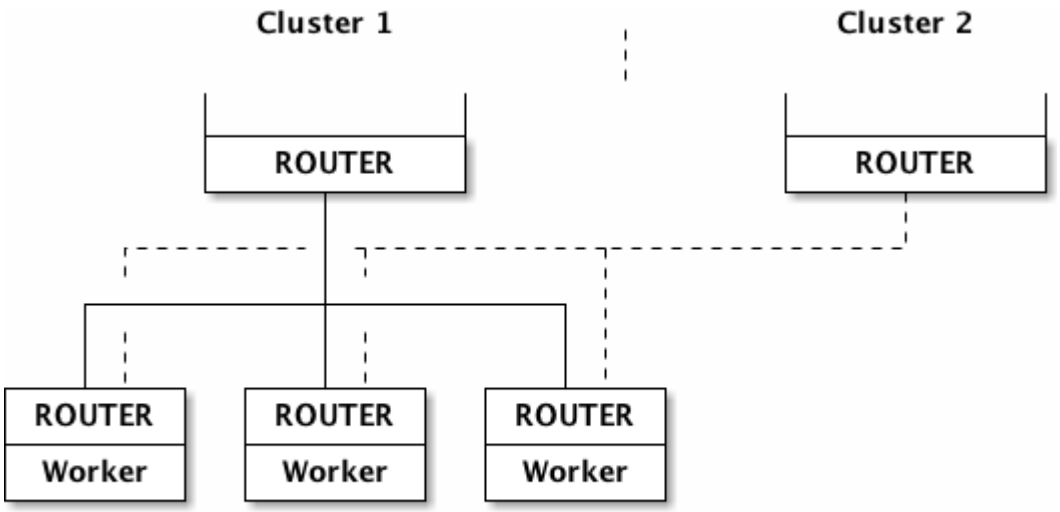

Figure 21 — Idea 1 — cross connected workers

这看上去很灵活，但却没有提供我们所需要的特性：client只有当集群中的worker不可用时才会去请求异地的worker。此外，worker的“已就绪”信号会同时发送给两个代理，这样就有可能同时获得两份任务。这个方案的失败还有一个原因：我们又将路由逻辑放在了边缘地带。

那来看看方案2，我们为各个代理建立连接，不修改worker和client：

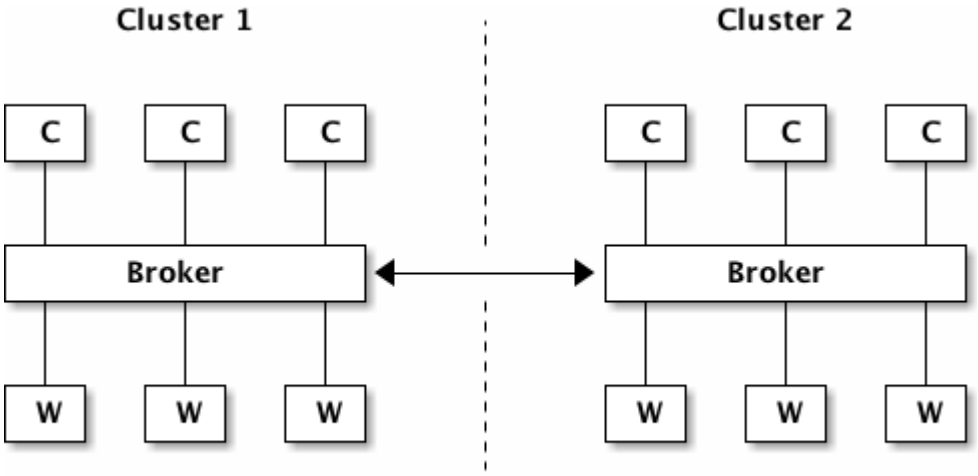

Figure 22 – Idea 2 – brokers talking to each other

这种设计的优势在于，我们只需要在一个地方解决问题就可以了，其他地方不需要修改。这就好像代理之间会秘密通信：伙计，我这儿有一些剩余的劳动力，如果你那儿忙不过来就跟我说，价钱好商量。

事实上，我们只不过是需要设计一种更为复杂的路由算法罢了：代理成为了其他代理的分包商。这种设计还有其他一些好处：

- 在普通情况下（如只存在一个集群），这种设计的处理方式和原来没有区别，当有多个集群时再进行其他动作。
- 对于不同的工作我们可以使用不同的消息流模式，如使用不同的网络链接。
- 架构的扩充看起来也比较容易，如有必要我们还可以添加一个超级代理来完成调度工作。

现在我们就开始编写代码。我们会将完整的集群写入一个进程，这样便于演示，而且稍作修改就能投入实际使用。这也是ZMQ的优美之处，你可以使用最小的开发模块来进行实验，最后方便地迁移到实际工程中。线程变成进程，消息模式和逻辑不需要改变。我们每个“集群”进程都包含client线程、worker线程、以及代理线程。

我们对基础模型应该已经很熟悉了：

- client线程使用REQ套接字，将请求发送给代理线程（ROUTER套接字）；
- worker线程使用REQ套接字，处理并应答从代理线程（ROUTER套接字）收到的请求；
- 代理会使用LRU队列和路由机制来管理请求。

联邦模式和同伴模式

连接代理的方式有很多，我们需要斟酌一番。我们需要的功能是告诉其他代理“我这里还有空闲的worker”，然后开始接收并处理一些任务；我们还需要能够告诉其他代理“够了够了，我这边的工作量也满了”。这个过程不

一定要十分完美，有时我们确实会接收超过承受能力的工作量，但仍能逐步地完成。

最简单的方式称为联邦，即代理充当其他代理的client和worker。我们可以将代理的前端套接字连接至其他代理的后端套接字，反之亦然。提示一下，ZMQ中是可以将一个套接字绑定到一个端点，同时又连接至另一个端点的。

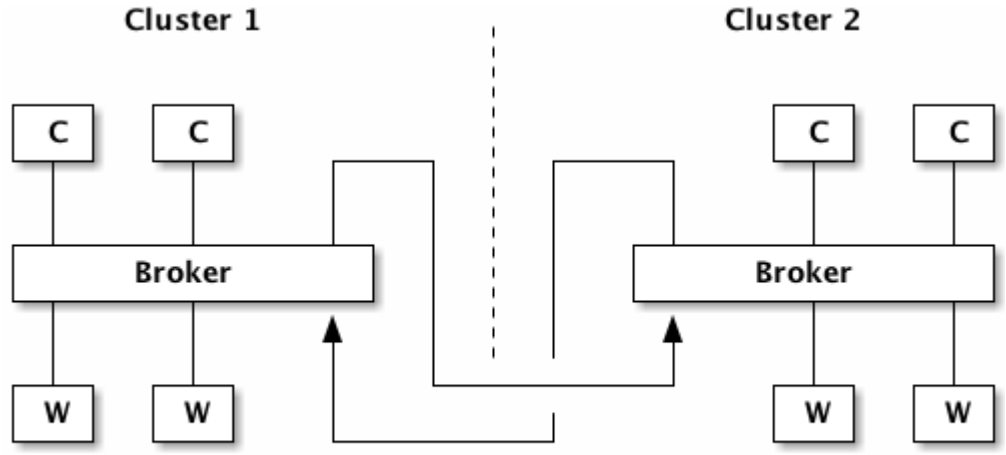

Figure 23 – Cross connected brokers in federation model

这种架构的逻辑会比较简单：当代理没有client时，它会告诉其他代理自己准备好了，并接收一个任务进行处理。但问题在于这种机制太简单了，联邦模式下的代理一次只能处理一个请求。如果client和worker是严格同步的，那么代理中的其他空闲worker将分配不到任务。我们需要的代理应该具备完全异步的特性。

但是，联邦模式对某些应用来说是非常好的，比如面向服务架构（SOA）。所以，先不要急着否定联邦模式，它只是不适用于LRU算法和集群负载均衡而已。

我们还有一种方式来连接代理：同伴模式。代理之间知道彼此的存在，并使用一个特殊的信道进行通信。我们逐步进行分析，假设有N个代理需要连接，每个代理则有N-1个同伴，所有代理都使用相同格式的消息进行通信。关于消息在代理之间的流通有两点需要注意：

- 每个代理需要告知所有同伴自己有多少空闲的worker，这是一则简单的消息，只是一个不断更新的数字，很显然我们会使用PUB-SUB套接字。这样一来，每个代理都会打开一个PUB套接字，不断告知外界自身的信息；同时又会打开一个SUB套接字，获取其他代理的信息。

- 每个代理需要以某种方式将工作任务交给其他代理，并能获取应答，这个过程需要是异步的。我们会使用ROUTER-ROUTER套接字来实现，没有其他选择。每个代理会使用两个这样的ROUTER套接字，一个用于接收任务，另一个用于分发任务。如果不使用两个套接字，那就需要额外的逻辑来判别收到的是请求还是应答，这就需要在消息中加入更多的信息。

另外还需要考虑的是代理和本地client和worker之间的通信。

The Naming Ceremony

代理中有三个消息流，每个消息流使用两个套接字，因此一共需要使用六个套接字。为这些套接字取一组好名字很重要，这样我们就不会在来回切换的时候找不着北。套接字是有一定任务的，他们的所完成的工作可以是命名的一部分。这样，当我们日后再重新阅读这些代码时，就不会显得太过陌生了。

以下是我们使用的三个消息流：

- 本地（local）的请求-应答消息流，实现代理和client、代理和worker之间的通信；
- 云端（cloud）的请求-应答消息流，实现代理和其同伴的通信；
- 状态（state）流，由代理和其同伴互相传递。

能够找到一些有意义的、且长度相同的名字，会让我们的代码对得比较整齐。可能他们并没有太多关联，但久了自然会习惯。

每个消息流会有两个套接字，我们之前一直称为“前端（frontend）”和“后端（backend）”。这两个名字我们已经使用很多次了：前端会负责接受信息或任务；后端会发送信息或任务给同伴。从概念上说，消息流都是从前往后的，应答则是从后往前。

因此，我们决定使用以下的命名方式：

- localfe / localbe
- cloudfe / cloudbe
- statefe / statebe

通信协议方面，我们全部使用ipc。使用这种协议的好处是，它能像tcp协议那样作为一种脱机通信协议来工作，而又不需要使用IP地址或DNS服务。对于ipc协议的端点，我们会命名为xxx-localfe/be、xxx-cloud、xxx-state，其中xxx代表集群的名称。

也许你会觉得这种命名方式太长了，还不如简单的叫s1、s2、s3……事实上，你的大脑并不是机器，阅读代码的时候不能立刻反应出变量的含义。而用上面这种“三个消息流，两个方向”的方式记忆，要比纯粹记忆“六个不同的套接字”来的方便。

以下是代理程序的套接字分布图：

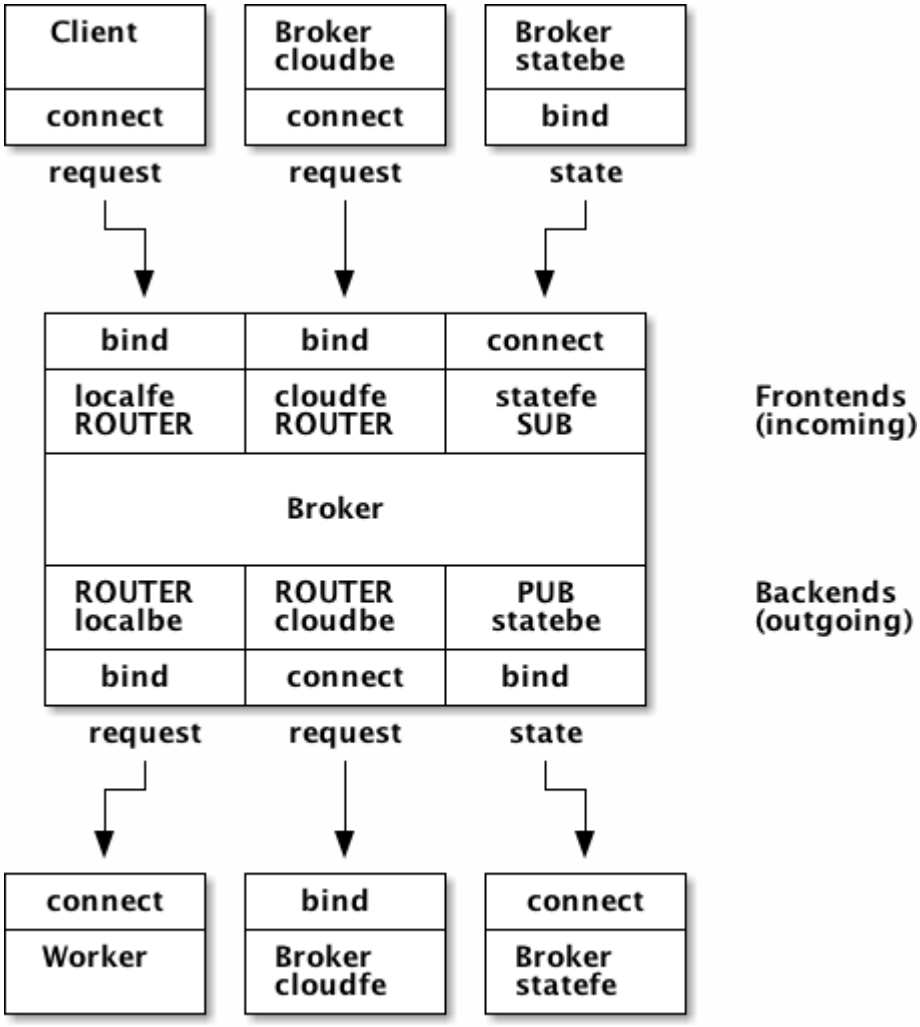

Figure 24 — Broker socket arrangement

请注意，我们会将cloudbe连接至其他代理的cloudfe，也会将statebe连接至其他代理的statefe。

状态流原型

由于每个消息流都有其巧妙之处，所以我们不会直接把所有的代码都写出来，而是分段编写和测试。当每个消息流都能正常工作了，我们再将其拼装成一个完整的应用程序。我们首先从状态流开始：

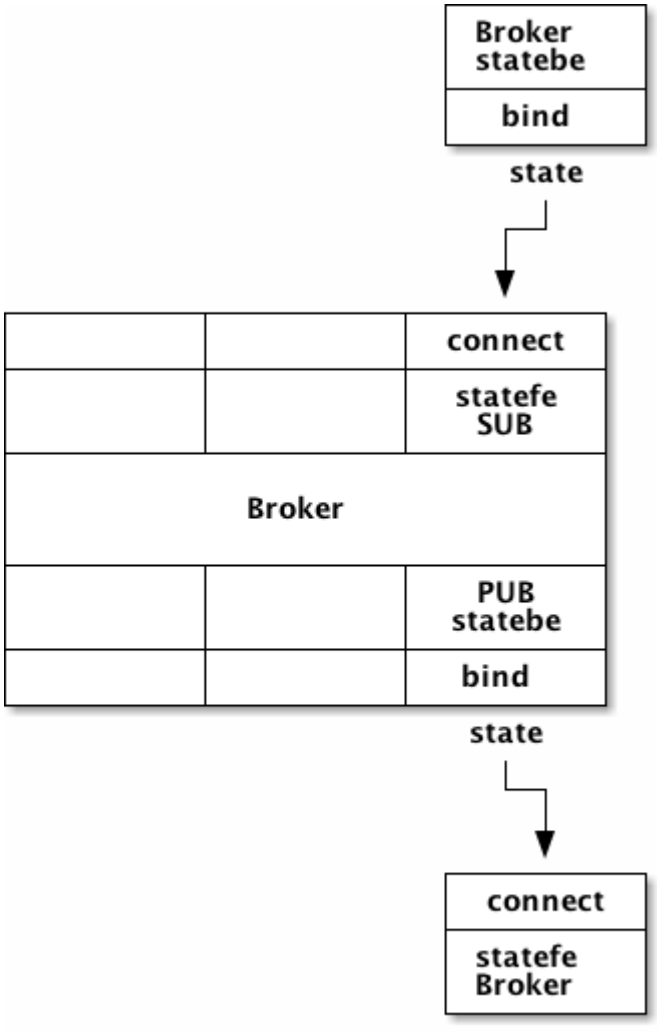

Figure 25 — The state flow

代码如下：

peering1: Prototype state flow in C

```
//
//  代理同伴模拟（第一部分）
//  状态流原型
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是各个同伴的名称
    //
    if (argc < 2) {
        printf ("syntax: peering1 me {you}...\n");
        exit (EXIT_FAILURE);
    }
```

```c
    char *self = argv [1];
    printf ("I: 正在准备代理程序 %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  连接statefe套接字至所有同伴
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的状态流后端\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  发送并接受状态消息
    //  zmq_poll()函数使用的超时时间即心跳时间
    //
    while (1) {
        //  初始化poll对象列表
        zmq_pollitem_t items [] = {
            { statefe, 0, ZMQ_POLLIN, 0 }
        };
        //  轮询套接字活动，超时时间为1秒
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        //  处理接收到的状态消息
        if (items [0].revents & ZMQ_POLLIN) {
            char *peer_name = zstr_recv (statefe);
            char *available = zstr_recv (statefe);
            printf ("同伴代理 %s 有 %s 个worker空闲\n", peer_name, available);
            free (peer_name);
            free (available);
        }
        else {
            //  发送随机数表示空闲的worker数
            zstr_sendm (statebe, self);
            zstr_sendf (statebe, "%d", randof (10));
        }
    }
    zctx_destroy (&ctx);
```

```
    return EXIT_SUCCESS;
}
```

几点说明：

- 每个代理都需要有各自的标识，用以生成相应的ipc端点名称。真实环境中，代理需要使用TCP协议连接，这就需要一个更为完备的配置机制，我们会在以后的章节中谈到。
- 程序的核心是一个zmq_poll()循环，它会处理接收到消息，并发送自身的状态。只有当zmq_poll()因无法获得同伴消息而超时时我们才会发送自身状态，如果我们每次收到消息都去发送自身状态，那消息就会过量了。
- 发送的状态消息包含两帧，第一帧是代理自身的地址，第二帧是空闲的worker数。我们必须要告知同伴代理自身的地址，这样才能接收到请求，唯一的方法就是在消息中显示注明。
- 我们没有在SUB套接字上设置标识，否则就会在连接到同伴代理时获得过期的状态信息。
- 我们没有在PUB套接字上设置阈值（HWM），因为订阅者是瞬时的。我们也可以将阈值设置为1，但其实是没有必要的。

让我们编译这段程序，用它模拟三个集群，DC1、DC2、DC3。我们在不同的窗口中运行以下命令：

```
peering1 DC1 DC2 DC3  #  Start DC1 and connect to DC2 and DC3
peering1 DC2 DC1 DC3  #  Start DC2 and connect to DC1 and DC3
peering1 DC3 DC1 DC2  #  Start DC3 and connect to DC1 and DC2
```

每个集群都会报告同伴代理的状态，之后每隔一秒都会打印出自己的状态。

在现实编程中，我们不会通过定时的方式来发送自身状态，而是在状态发生改变时就发送。这看起来会很占用带宽，但其实状态消息的内容很少，而且集群间的连接是非常快速的。

如果我们想要以较为精确的周期来发送状态信息，可以新建一个线程，将statebe套接字打开，然后由主线程将不规则的状态信息发送给子线程，再由子线程定时发布这些消息。不过这种机制就需要额外的编程了。

本地流和云端流原型

下面让我们建立本地流和云端流的原型。这段代码会从client获取请求，并随机地分派给集群内的worker或其他集群。

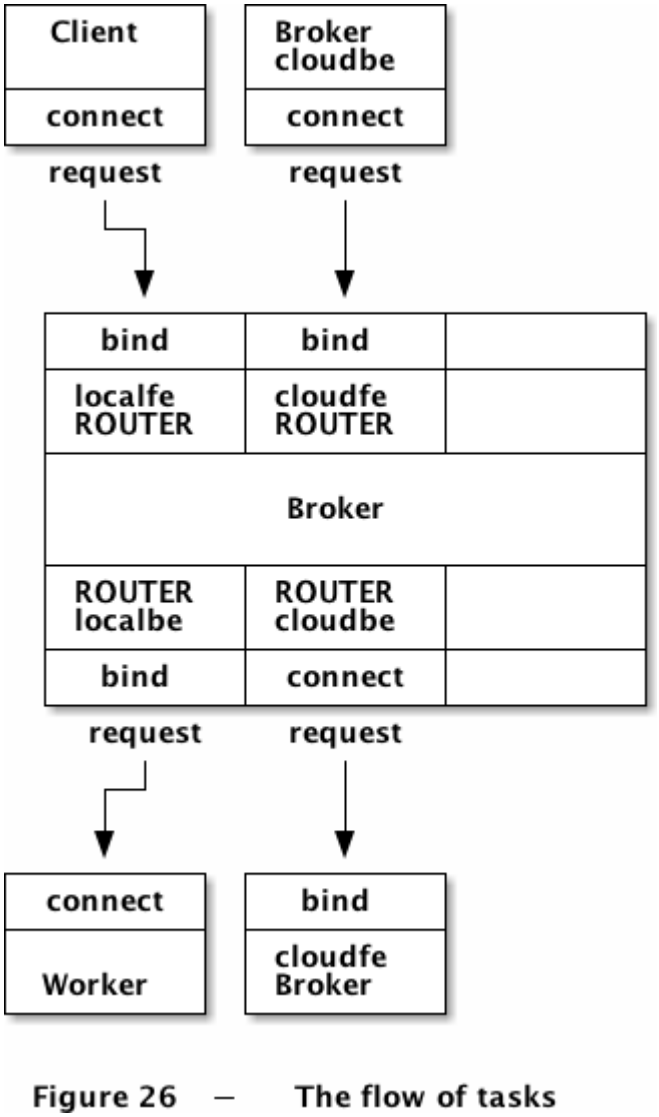

Figure 26 — The flow of tasks

在编写代码之前，让我们先描绘一下核心的路由逻辑，整理出一份简单而健壮的设计。

我们需要两个队列，一个队列用于存放从本地集群client收到的请求，另一个存放其他集群发送来的请求。一种方法是从本地和云端的前端套接字中获取消息，分别存入两个队列。但是这么做似乎是没有必要的，因为ZMQ套接字本身就是队列。所以，我们直接使用ZMQ套接字提供的缓存来作为队列使用。

这项技术我们在LRU队列装置中使用过，且工作得很好。做法是，当代理下有空闲的worker或能接收请求的其他集群时，才从套接字中获取请求。我们可以不断地从后端获取应答，然后路由回去。如果后端没有任何响应，那也就没有必要去接收前端的请求了。

所以，我们的主循环会做以下几件事：

- 轮询后端套接字，会从worker处获得“已就绪”的消息或是一个应答。如果是应答消息，则将其路由回集群client，或是其他集群。
- worker应答后即可标记为可用，放入队列并计数；
- 如果有可用的worker，就获取一个请求，该请求可能来自集群内的client，也可能是其他集群。随后将请求转发给集群内的worker，或是随机转发给其他集群。

这里我们只是随机地将请求发送给其他集群，而不是在代理中模拟出一个worker，进行集群间的任务分发。这看起来挺愚蠢的，不过目前尚可使用。

我们使用代理的标识来进行代理之前的消息路由。每个代理都有自己的名字，是在命令行中指定的。只要这些指定的名字和ZMQ为client自动生成的UUID不重复，那么我们就可以知道应答是要返回给client，还是返回给另一个集群。

下面是代码，有趣的部分已在程序中标注：

peering2: Prototype local and cloud flow in C

```
//
//  代理同伴模拟（第二部分）
//  请求-应答消息流原型
//
//  示例程序使用了一个进程，这样可以让程序变得简单，
//  每个线程都有自己的上下文对象，所以可以认为他们是多个进程。
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 3
#define LRU_READY   "\001"      //  消息：worker已就绪

//  代理名称；现实中，这个名称应该由某种配置完成
static char *self;

//  请求-应答客户端使用REQ套接字
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);

    while (1) {
        //  发送请求，接收应答
        zstr_send (client, "HELLO");
        char *reply = zstr_recv (client);
        if (!reply)
            break;              //  中断
        printf ("Client: %s\n", reply);
        free (reply);
        sleep (1);
    }
```

```
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字，并进行LRU路由
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  告知代理worker已就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    //  处理消息
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断

        zframe_print (zmsg_last (msg), "Worker: ");
        zframe_reset (zmsg_last (msg), "OK", 2);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}


int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是同伴代理的名称
    //
    if (argc < 2) {
        printf ("syntax: peering2 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: 正在准备代理程序 %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  准备上下文和套接字
```

```
    zctx_t *ctx = zctx_new ();
    char endpoint [256];

    //  将cloudfe绑定至端点
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  将cloudbe连接至同伴代理的端点
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的cloudfe端点\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }
    //  准备本地前端和后端
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);
    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  让用户告诉我们何时开始
    printf ("请确认所有代理已经启动，按任意键继续: ");
    getchar ();

    //  启动本地worker
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  启动本地client
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);

    //  有趣的部分
    //  ---------------------------------------------------------------
    //  请求-应答消息流
    //  - 若本地有可用worker，则轮询获取本地或云端的请求；
    //  - 将请求路由给本地worker或其他集群。

    //  可用worker队列
    int capacity = 0;
    zlist_t *workers = zlist_new ();
```

```
    while (1) {
        zmq_pollitem_t backends [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 }
        };
        //  如果没有可用worker，则继续等待
        int rc = zmq_poll (backends, 2,
            capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  中断

        //  处理本地worker的应答
        zmsg_t *msg = NULL;
        if (backends [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);
            capacity++;

            //  如果是“已就绪”的信号，则不再进行路由
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  处理来自同伴代理的应答
        else
        if (backends [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  中断
            //  我们不需要使用同伴代理的地址
            zframe_t *address = zmsg_unwrap (msg);
            zframe_destroy (&address);
        }
        //  如果应答消息中的地址是同伴代理的，则发送给它
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  将应答路由给本地client
```

```
        if (msg)
            zmsg_send (&msg, localfe);

        //  开始处理客户端请求
        //
        while (capacity) {
            zmq_pollitem_t frontends [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            rc = zmq_poll (frontends, 2, 0);
            assert (rc >= 0);
            int reroutable = 0;
            //  优先处理同伴代理的请求，避免资源耗尽
            if (frontends [1].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (cloudfe);
                reroutable = 0;
            }
            else
            if (frontends [0].revents & ZMQ_POLLIN) {
                msg = zmsg_recv (localfe);
                reroutable = 1;
            }
            else
                break;      //  没有请求

            //  将20%的请求发送给其他集群
            //
            if (reroutable && argc > 2 && randof (5) == 0) {
                //  随地地路由给同伴代理
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv [random_peer]));
                zmsg_send (&msg, cloudbe);
            }
            else {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                capacity--;
            }
        }
    }
    //  程序结束后的清理工作
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
```

```
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

在两个窗口中运行以上代码：

```
peering2 me you
peering2 you me
```

几点说明：

- zmsg类库让程序变得简单多了，这类程序显然应该成为我们ZMQ程序员必备的工具；
  由于我们没有在程序中实现获取同伴代理状态的功能，所以先暂且认为他们都是有空闲worker的。现实中，我们不会将请求发送个一个不存在的同伴代理。

- 你可以让这段程序长时间地运行下去，看看会不会出现路由错误的消息，因为一旦错误，client就会阻塞。你可以试着将一个代理关闭，就能看到代理无法将请求路由给云端中的其他代理，client逐个阻塞，程序也停止打印跟踪信息。

组装

让我们将所有这些放到一段代码里。和之前一样，我们会在一个进程中完成所有工作。我们会将上文中的两个示例程序结合起来，编写出一个可以模拟任意多个集群的程序。

代码共有270行，非常适合用来模拟一组完整的集群程序，包括client、worker、代理、以及云端任务分发机制。

peering3: Full cluster simulation in C

```
//
//  同伴代理模拟（第三部分）
//  状态和任务消息流原型
//
//  示例程序使用了一个进程，这样可以让程序变得简单，
//  每个线程都有自己的上下文对象，所以可以认为他们是多个进程。
//
#include "czmq.h"

#define NBR_CLIENTS 10
#define NBR_WORKERS 5
#define LRU_READY   "\001"      //  消息：worker已就绪

//  代理名称；现实中，这个名称应该由某种配置完成
```

```c
static char *self;

//  请求-应答客户端使用REQ套接字
//  为模拟压力测试，客户端会一次性发送大量请求
//
static void *
client_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, "ipc://%s-localfe.ipc", self);
    void *monitor = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (monitor, "ipc://%s-monitor.ipc", self);

    while (1) {
        sleep (randof (5));
        int burst = randof (15);
        while (burst--) {
            char task_id [5];
            sprintf (task_id, "%04X", randof (0x10000));

            //  使用随机的十六进制ID来标注任务
            zstr_send (client, task_id);

            //  最多等待10秒
            zmq_pollitem_t pollset [1] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (pollset, 1, 10 * 1000 * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断

            if (pollset [0].revents & ZMQ_POLLIN) {
                char *reply = zstr_recv (client);
                if (!reply)
                    break;              //  中断
                //  worker的应答中应包含任务ID
                puts (reply);
                assert (streq (reply, task_id));
                free (reply);
            }
            else {
                zstr_sendf (monitor,
                    "E: 客户端退出，丢失的任务为： %s", task_id);
                return NULL;
            }
        }
    }
```

```c
    zctx_destroy (&ctx);
    return NULL;
}

//  worker使用REQ套接字，并进行LRU路由
//
static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (worker, "ipc://%s-localbe.ipc", self);

    //  告知代理worker已就绪
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    while (1) {
        //  worker会随机延迟几秒
        zmsg_t *msg = zmsg_recv (worker);
        sleep (randof (2));
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (int argc, char *argv [])
{
    //  第一个参数是代理的名称
    //  其他参数是同伴代理的名称
    //
    if (argc < 2) {
        printf ("syntax: peering3 me {you}...\n");
        exit (EXIT_FAILURE);
    }
    self = argv [1];
    printf ("I: 正在准备代理程序 %s...\n", self);
    srandom ((unsigned) time (NULL));

    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    char endpoint [256];

    //  将cloudfe绑定至端点
    void *cloudfe = zsocket_new (ctx, ZMQ_ROUTER);
```

```
    zsockopt_set_identity (cloudfe, self);
    zsocket_bind (cloudfe, "ipc://%s-cloud.ipc", self);

    //  将statebe绑定至端点
    void *statebe = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (statebe, "ipc://%s-state.ipc", self);

    //  将cloudbe连接至同伴代理的端点
    void *cloudbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsockopt_set_identity (cloudbe, self);
    int argn;
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的cloudfe端点\n", peer);
        zsocket_connect (cloudbe, "ipc://%s-cloud.ipc", peer);
    }

    //  将statefe连接至同伴代理的端点
    void *statefe = zsocket_new (ctx, ZMQ_SUB);
    for (argn = 2; argn < argc; argn++) {
        char *peer = argv [argn];
        printf ("I: 正在连接至同伴代理 '%s' 的statebe端点\n", peer);
        zsocket_connect (statefe, "ipc://%s-state.ipc", peer);
    }
    //  准备本地前端和后端
    void *localfe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localfe, "ipc://%s-localfe.ipc", self);

    void *localbe = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (localbe, "ipc://%s-localbe.ipc", self);

    //  准备监控套接字
    void *monitor = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (monitor, "ipc://%s-monitor.ipc", self);

    //  启动本地worker
    int worker_nbr;
    for (worker_nbr = 0; worker_nbr < NBR_WORKERS; worker_nbr++)
        zthread_new (ctx, worker_task, NULL);

    //  启动本地client
    int client_nbr;
    for (client_nbr = 0; client_nbr < NBR_CLIENTS; client_nbr++)
        zthread_new (ctx, client_task, NULL);

    //  有趣的部分
```

```
    //  ----------------------------------------------------------------
    //  发布-订阅消息流
    //  - 轮询同伴代理的状态信息；
    //  - 当自身状态改变时，对外广播消息。
    //  请求-应答消息流
    //  - 若本地有可用worker，则轮询获取本地或云端的请求；
    //  - 将请求路由给本地worker或其他集群。

    //  可用worker队列
    int local_capacity = 0;
    int cloud_capacity = 0;
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t primary [] = {
            { localbe, 0, ZMQ_POLLIN, 0 },
            { cloudbe, 0, ZMQ_POLLIN, 0 },
            { statefe, 0, ZMQ_POLLIN, 0 },
            { monitor, 0, ZMQ_POLLIN, 0 }
        };
        //  如果没有可用的worker，则一直等待
        int rc = zmq_poll (primary, 4,
            local_capacity? 1000 * ZMQ_POLL_MSEC: -1);
        if (rc == -1)
            break;              //  中断

        //  跟踪自身状态信息是否改变
        int previous = local_capacity;

        //  处理本地worker的应答
        zmsg_t *msg = NULL;

        if (primary [0].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (localbe);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);
            local_capacity++;

            //  如果是“已就绪”的信号，则不再进行路由
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), LRU_READY, 1) == 0)
                zmsg_destroy (&msg);
        }
        //  处理来自同伴代理的应答
```

```
        else
        if (primary [1].revents & ZMQ_POLLIN) {
            msg = zmsg_recv (cloudbe);
            if (!msg)
                break;          //  Interrupted
            //  我们不需要使用同伴代理的地址
            zframe_t *address = zmsg_unwrap (msg);
            zframe_destroy (&address);
        }
        //  如果应答消息中的地址是同伴代理的，则发送给它
        for (argn = 2; msg && argn < argc; argn++) {
            char *data = (char *) zframe_data (zmsg_first (msg));
            size_t size = zframe_size (zmsg_first (msg));
            if (size == strlen (argv [argn])
            &&  memcmp (data, argv [argn], size) == 0)
                zmsg_send (&msg, cloudfe);
        }
        //  将应答路由给本地client
        if (msg)
            zmsg_send (&msg, localfe);

        //  处理同伴代理的状态更新
        if (primary [2].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (statefe);
            cloud_capacity = atoi (status);
            free (status);
        }
        //  处理监控消息
        if (primary [3].revents & ZMQ_POLLIN) {
            char *status = zstr_recv (monitor);
            printf ("%s\n", status);
            free (status);
        }

        //  开始处理客户端请求
        //  - 如果本地有空闲worker，则总本地client和云端接收请求；
        //  - 如果我们只有空闲的同伴代理，则只轮询本地client的请求；
        //  - 将请求路由给本地worker，或者同伴代理。
        //
        while (local_capacity + cloud_capacity) {
            zmq_pollitem_t secondary [] = {
                { localfe, 0, ZMQ_POLLIN, 0 },
                { cloudfe, 0, ZMQ_POLLIN, 0 }
            };
            if (local_capacity)
                rc = zmq_poll (secondary, 2, 0);
```

```
            else
                rc = zmq_poll (secondary, 1, 0);
            assert (rc >= 0);

            if (secondary [0].revents & ZMQ_POLLIN)
                msg = zmsg_recv (localfe);
            else
            if (secondary [1].revents & ZMQ_POLLIN)
                msg = zmsg_recv (cloudfe);
            else
                break;      //  没有任务

            if (local_capacity) {
                zframe_t *frame = (zframe_t *) zlist_pop (workers);
                zmsg_wrap (msg, frame);
                zmsg_send (&msg, localbe);
                local_capacity--;
            }
            else {
                //  随机路由给同伴代理
                int random_peer = randof (argc - 2) + 2;
                zmsg_pushmem (msg, argv [random_peer], strlen (argv [random_peer]));
                zmsg_send (&msg, cloudbe);
            }
        }
        if (local_capacity != previous) {
            //  将自身代理的地址附加到消息中
            zstr_sendm (statebe, self);
            //  广播新的状态信息
            zstr_sendf (statebe, "%d", local_capacity);
        }
    }
    //  程序结束后的清理工作
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return EXIT_SUCCESS;
}
```

这段代码并不长，但花费了大约一天的时间去调通。以下是一些说明：

- client线程会检测并报告失败的请求，它们会轮询代理套接字，查看是否有应答，超时时间为10秒。

- client线程不会自己打印信息，而是将消息PUSH给一个监控线程，由它打印消息。这是我们第一次使用ZMQ进行监控和记录日志，我们以后会见得更多。

- clinet会模拟多种负载情况，让集群在不同的压力下工作，因此请求可能会在本地处理，也有可能会发送至云端。集群中的client和worker数量、其他集群的数量，以及延迟时间，会左右这个结果。你可以设置不同的参数来测试它们。

- 主循环中有两组轮询集合，事实上我们可以使用三个：信息流、后端、前端。因为在前面的例子中，如果后端没有空闲的worker，就没有必要轮询前端请求了。

以下是几个在编写过程中遇到的问题：

- 如果请求或应答在某处丢失，client会因此阻塞。回忆以下，ROUTER-ROUTER套接字会在消息如法路由的情况下直接丢弃。这里的一个策略就是改变client线程，检测并报告这种错误。此外，我还在每次recv()之后以及send()之前使用zmsg_dump()来打印套接字内容，用来更快地定位消息。

- 主循环会错误地从多个已就绪的套接字中获取消息，造成第一条消息的丢失。解决方法是只从第一个已就绪的套接字中获取消息。

- zmsg类库没有很好地将UUID编码为C语言字符串，导致包含字节0的UUID会崩溃。解决方法是将UUID转换成可打印的十六进制字符串。

这段模拟程序没有检测同伴代理是否存在。如果你开启了某个代理，它已向其他代理发送过状态信息，然后关闭了，那其他代理仍会向它发送请求。这样一来，其他代理的client就会报告很多错误。解决时有两点：一、为状态信息设置有效期，当同伴代理消失一段时间后就不再发送请求；二、提高请求-应答的可靠性，这在下一章中会讲到。