可靠的请求-应答模式

第三章中我们使用实例介绍了高级请求-应答模式，本章我们会讲述请求-应答模式的可靠性问题，并使用ZMQ提供的套接字类型组建起可靠的请求-应答消息系统。

本章将介绍的内容有：

- 客户端请求-应答
- 最近最少使用队列
- 心跳机制
- 面向服务的队列
- 基于磁盘（脱机）队列
- 主从备份服务
- 无中间件的请求-应答

什么是可靠性？

要给可靠性下定义，我们可以先界定它的相反面——故障。如果我们可以处理某些类型的故障，那么我们的模型对于这些故障就是可靠的。下面我们就来列举分布式ZMQ应用程序中可能发生的问题，从可能性高的故障开始：

- 应用程序代码是最大的故障来源。程序会崩溃或中止，停止对数据来源的响应，或是响应得太慢，耗尽内存等。
- 系统代码，如使用ZMQ编写的中间件，也会意外中止。系统代码应该要比应用程序代码更为可靠，但毕竟也有可能崩溃。特别是当系统代码与速度过慢的客户端交互时，很容易耗尽内存。
- 消息队列溢出，典型的情况是系统代码中没有对慢客户端做积极的处理，任由消息队列溢出。
- 网络临时中断，造成消息丢失。这类错误ZMQ应用程序是无法及时发现的，因为ZMQ会自动进行重连。
- 硬件系统崩溃，导致所有进程中止。
- 网络会出现特殊情形的中断，如交换机的某个端口发生故障，导致部分网络无法访问。
- 数据中心可能遭受雷击、地震、火灾、电压过载、冷却系统失效等。

想要让软件系统规避上述所有的风险，需要大量的人力物力，故不在本指南的讨论范围之内。

由于前五个故障类型涵盖了99.9%的情形（这一数据源自我近期进行的一项研究），所以我们会深入探讨。如果你的公司大到足以考虑最后两种情形，那请及时联系我，因为我正愁没钱将我家后院的大坑建成游泳池。

可靠性设计

简单地来说，可靠性就是当程序发生故障时也能顺利地运行下去，这要比搭建一个消息系统来得困难得多。我们会根据ZMQ提供的每一种核心消息模式，来看看如何保障代码的持续运行。

- 请求-应答模式：当服务端在处理请求是中断，客户端能够得知这一信息，并停止接收消息，转而选择等待重试、请求另一服务端等操作。这里我们暂不讨论客户端发生问题的情形。

- 发布-订阅模式：如果客户端收到一些消息后意外中止，服务端是不知道这一情况的。发布-订阅模式中的订阅者不会返回任何消息给发布者。但是，订阅者可以通过其他方式联系服务端，如请求-应答模式，要求服务端重发消息。这里我们暂不讨论服务端发生问题的情形。此外，订阅者可以通过某些方式检查自身是否运行得过慢，并采取相应措施（向操作者发出警告、中止等）。

- 管道模式：如果worker意外终止，任务分发器将无从得知。管道模式和发布-订阅模式类似，只朝一个方向发送消息。但是，下游的结果收集器可以检测哪项任务没有完成，并告诉任务分发器重新分配该任务。如果任务分发器或结果收集器意外中止了，那客户端发出的请求只能另作处理。所以说，系统代码真的要减少出错的几率，因为这很难处理。

本章主要讲解请求-应答模式中的可靠性设计，其他模式将在后续章节中讲解。

最基本的请求应答模式是REQ客户端发送一个同步的请求至REP服务端，这种模式的可靠性很低。如果服务端在处理请求时中止，那客户端会永远处于等待状态。

相比TCP协议，ZMQ提供了自动重连机制、消息分发的负载均衡等。但是，在真实环境中这也是不够的。唯一可以完全信任基本请求-应答模式的应用场景是同一进程的两个线程之间进行通信，没有网络问题或服务器失效的情况。

但是，只要稍加修饰，这种基本的请求-应答模式就能很好地在现实环境中工作了。我喜欢将其称为“海盗”模式。

粗略地讲，客户端连接服务端有三种方式，每种方式都需要不同的可靠性设计：

- 多个客户端直接和单个服务端进行通信。使用场景：只有一个单点服务器，所有客户端都需要和它通信。需处理的故障：服务器崩溃和重启；网络连接中断。

- 多个客户端和单个队列装置通信，该装置将请求分发给多个服务端。使用场景：任务分发。需处理的故障：worker崩溃和重启，死循环，过载；队列装置崩溃和重启；网络中断。

- 多个客户端直接和多个服务端通信，无中间件。使用场景：类似域名解析的分布式服务。需处理的故障：服务端崩溃和重启，死循环，过载；网络连接中断。

以上每种设计都必须有所取舍，很多时候会混合使用。下面我们详细说明。

客户端的可靠性设计（懒惰海盗模式）

我们可以通过在客户端进行简单的设置，来实现可靠的请求-应答模式。我暂且称之为“懒惰的海盗”（Lazy Pirate）模式。

在接收应答时，我们不进行同步等待，而是做以下操作：

- 对REQ套接字进行轮询，当消息抵达时才进行接收；
- 请求超时后重发消息，循环多次；
- 若仍无消息，则结束当前事务。

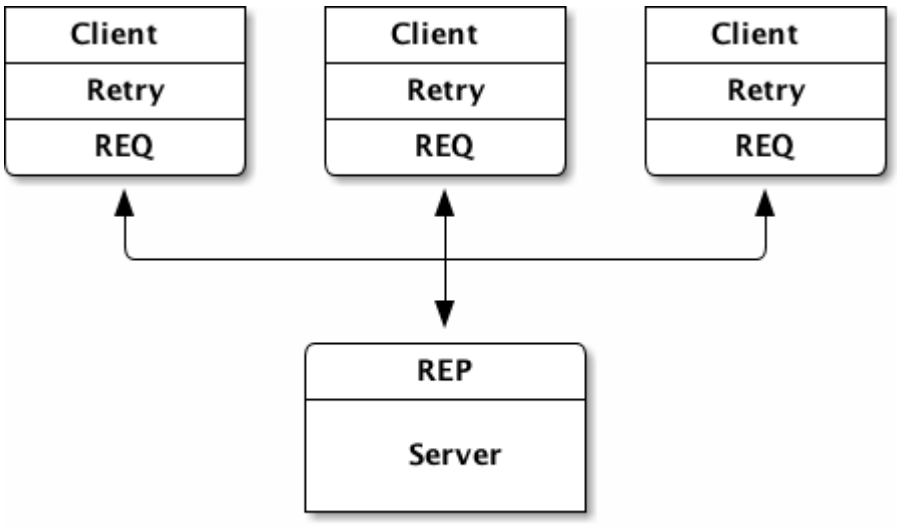

Figure 1 — Lazy Pirate pattern

使用REQ套接字时必须严格遵守发送-接收过程，因为它内部采用了一个有限状态机来限定状态，这一特性会让我们应用“海盗”模式时遇上一些麻烦。最简单的做法是将REQ套接字关闭重启，从而打破这一限定。

lpclient: Lazy Pirate client in C

```c
//
//  Lazy Pirate client
//  使用zmq_poll轮询来实现安全的请求-应答
//  运行时可随机关闭或重启lpserver程序
//
#include "czmq.h"

#define REQUEST_TIMEOUT     2500    //  毫秒，(> 1000!)
#define REQUEST_RETRIES     3       //  尝试次数
#define SERVER_ENDPOINT     "tcp://localhost:5555"

int main (void)
{
    zctx_t *ctx = zctx_new ();
    printf ("I: 正在连接服务器...\n");
    void *client = zsocket_new (ctx, ZMQ_REQ);
    assert (client);
    zsocket_connect (client, SERVER_ENDPOINT);

    int sequence = 0;
    int retries_left = REQUEST_RETRIES;
    while (retries_left && !zctx_interrupted) {
        //  发送一个请求，并开始接收消息
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);
```

```
        int expect_reply = 1;
        while (expect_reply) {
            //  对套接字进行轮询，并设置超时时间
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断

            //  如果接收到回复则进行处理
            if (items [0].revents & ZMQ_POLLIN) {
                //  收到服务器应答，必须和请求时的序号一致
                char *reply = zstr_recv (client);
                if (!reply)
                    break;      //  Interrupted
                if (atoi (reply) == sequence) {
                    printf ("I: 服务器返回正常 (%s)\n", reply);
                    retries_left = REQUEST_RETRIES;
                    expect_reply = 0;
                }
                else
                    printf ("E: 服务器返回异常: %s\n",
                        reply);

                free (reply);
            }
            else
            if (--retries_left == 0) {
                printf ("E: 服务器不可用，取消操作\n");
                break;
            }
            else {
                printf ("W: 服务器没有响应，正在重试...\n");
                //  关闭旧套接字，并建立新套接字
                zsocket_destroy (ctx, client);
                printf ("I: 服务器重连中...\n");
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, SERVER_ENDPOINT);
                //  使用新套接字再次发送请求
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

lpserver: Lazy Pirate server in C

```c
//
//  Lazy Pirate server
//  将REQ套接字连接至 tcp://*:5555
//  和hwserver程序类似，除了以下两点：
//    - 直接输出请求内容
//    - 随机地降慢运行速度，或中止程序，模拟崩溃
//
#include "zhelpers.h"

int main (void)
{
    srandom ((unsigned) time (NULL));

    void *context = zmq_init (1);
    void *server = zmq_socket (context, ZMQ_REP);
    zmq_bind (server, "tcp://*:5555");

    int cycles = 0;
    while (1) {
        char *request = s_recv (server);
        cycles++;

        //  循环几次后开始模拟各种故障
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: 模拟程序崩溃\n");
            break;
        }
        else
        if (cycles > 3 && randof (3) == 0) {
            printf ("I: 模拟CPU过载\n");
            sleep (2);
        }
        printf ("I: 正常请求 (%s)\n", request);
        sleep (1);              //  耗时的处理过程
        s_send (server, request);
        free (request);
    }
    zmq_close (server);
    zmq_term (context);
    return 0;
}
```

运行这个测试用例时，可以打开两个控制台，服务端会随机发生故障，你可以看看客户端的反应。服务端的典型输出如下：

```
I: normal request (1)
I: normal request (2)
I: normal request (3)
I: simulating CPU overload
I: normal request (4)
I: simulating a crash
```

客户端的输出是：

```
I: connecting to server...
I: server replied OK (1)
I: server replied OK (2)
I: server replied OK (3)
W: no response from server, retrying...
I: connecting to server...
W: no response from server, retrying...
I: connecting to server...
E: server seems to be offline, abandoning
```

客户端为每次请求都加上了序列号，并检查收到的应答是否和序列号一致，以保证没有请求或应答丢失，同一个应答收到多次或乱序。多运行几次实例，看看是否真的能够解决问题。现实环境中你不需要使用到序列号，那只是为了证明这一方式是可行的。

客户端使用REQ套接字进行请求，并在发生问题时打开一个新的套接字来，绕过REQ强制的发送/接收过程。可能你会想用DEALER套接字，但这并不是一个好主意。首先，DEALER并不会像REQ那样处理信封（如果你不知道信封是什么，那更不能用DEALER了）。其次，你可能会获得你并不想得到的结果。

这一方案的优劣是：

- 优点：简单明了，容易实施；
- 优点：可以方便地应用到现有的客户端和服务端程序中；
- 优点：ZMQ有自动重连机制；
- 缺点：单点服务发生故障时不能定位到新的可用服务。

基本的可靠队列（简单海盗模式）

在第二种模式中，我们使用一个队列装置来扩展上述的“懒惰的海盗”模式，使客户端能够透明地和多个服务端通信。这里的服务端可以定义为worker。我们可以从最基础的模型开始，分阶段实施这个方案。

在所有的海盗模式中，worker是无状态的，或者说存在着一个我们所不知道的公共状态，如共享数据库。队列装置的存在意味着worker可以在client毫不知情的情况下随意进出。一个worker死亡后，会有另一个worker接替它的工作。这种拓扑结果非常简洁，但唯一的缺点是队列装置本身会难以维护，可能造成单点故障。

在第三章中，队列装置的基本算法是最近最少使用算法。那么，如果worker死亡或阻塞，我们需要做些什么？答案是很少很少。我们已经在client中加入了重试的机制，所以，使用基本的LRU队列就可以运作得很好了。这种做法也符合ZMQ的逻辑，所以我们可以通过在点对点交互中插入一个简单的队列装置来扩展它：

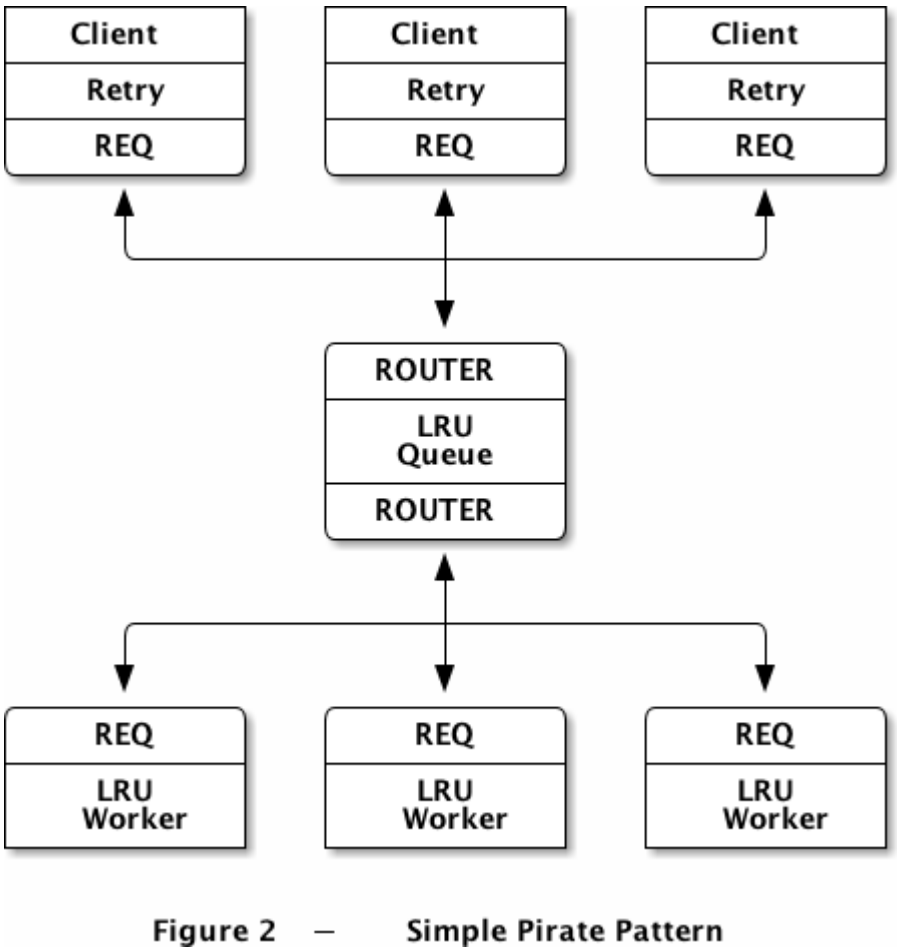

Figure 2 — Simple Pirate Pattern

我们可以直接使用“懒惰的海盗”模式中的client，以下是队列装置的代码：

spqueue: Simple Pirate queue in C

```
//
//  简单海盗队列
//
//  这个装置和LRU队列完全一致，不存在任何可靠性机制，依靠client的重试来保证装置的运行
//
#include "czmq.h"

#define LRU_READY   "\001"      //  消息：worker准备就绪

int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");    //  client端点
    zsocket_bind (backend,  "tcp://*:5556");    //  worker端点

    //  存放可用worker的队列
```

```
    zlist_t *workers = zlist_new ();

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  当有可用的woker时，轮询前端端点
        int rc = zmq_poll (items, zlist_size (workers)? 2: 1, -1);
        if (rc == -1)
            break;              //  中断

        //  处理后端端点的worker消息
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用worker的地址进行LRU排队
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断
            zframe_t *address = zmsg_unwrap (msg);
            zlist_append (workers, address);

            //  如果消息不是READY，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请求，转发给第一个可用的worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (msg) {
                zmsg_wrap (msg, (zframe_t *) zlist_pop (workers));
                zmsg_send (&msg, backend);
            }
        }
    }
    //  程序运行结束，进行清理
    while (zlist_size (workers)) {
        zframe_t *frame = (zframe_t *) zlist_pop (workers);
        zframe_destroy (&frame);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

以下是worker的代码，用到了“懒惰的海盗”服务，并将其调整为LRU模式（使用REQ套接字传递“已就绪”信号）：

spworker: Simple Pirate worker in C

```
//
//  简单海盗模式worker
//
//  使用REQ套接字连接tcp://*:5556，使用LRU算法实现worker
//
#include "czmq.h"
#define LRU_READY   "\001"      //  消息：worker已就绪

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_REQ);

    //  使用随机符号来指定套接字标识，方便追踪
    srandom ((unsigned) time (NULL));
    char identity [10];
    sprintf (identity, "%04X-%04X", randof (0x10000), randof (0x10000));
    zmq_setsockopt (worker, ZMQ_IDENTITY, identity, strlen (identity));
    zsocket_connect (worker, "tcp://localhost:5556");

    //  告诉代理worker已就绪
    printf ("I: (%s) worker准备就绪\n", identity);
    zframe_t *frame = zframe_new (LRU_READY, 1);
    zframe_send (&frame, worker, 0);

    int cycles = 0;
    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        if (!msg)
            break;              //  中断

        //  经过几轮循环后，模拟各种问题
        cycles++;
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) 模拟崩溃\n", identity);
            zmsg_destroy (&msg);
            break;
        }
        else
        if (cycles > 3 && randof (5) == 0) {
            printf ("I: (%s) 模拟CPU过载\n", identity);
```

```
            sleep (3);
            if (zctx_interrupted)
                break;
        }
        printf ("I: (%s) 正常应答\n", identity);
        sleep (1);              //  进行某些处理
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return 0;
}
```

运行上述事例，启动多个worker，一个client，以及一个队列装置，顺序随意。你可以看到worker最终都会崩溃或死亡，client则多次重试并最终放弃。装置从来不会停止，你可以任意重启worker和client，这个模型可以和任意个worker、client交互。

健壮的可靠队列（偏执海盗模式）

“简单海盗队列”模式工作得非常好，主要是因为它只是两个现有模式的结合体。不过，它也有一些缺点：

- 该模式无法处理队列的崩溃或重启。client会进行重试，但worker不会重启。虽然ZMQ会自动重连worker的套接字，但对于新启动的队列装置来说，由于worker并没有发送“已就绪”的消息，所以它相当于是不存在的。为了解决这一问题，我们需要从队列发送心跳给worker，这样worker就能知道队列是否已经死亡。

- 队列没有检测worker是否已经死亡，所以当worker在处于空闲状态时死亡，队列装置只有在发送了某个请求之后才会将该worker从队列中移除。这时，client什么都不能做，只能等待。这不是一个致命的问题，但是依然是不够好的。所以，我们需要从worker发送心跳给队列装置，从而让队列得知worker什么时候消亡。

我们使用一个名为“偏执的海盗模式”来解决上述两个问题。

之前我们使用REQ套接字作为worker的套接字类型，但在偏执海盗模式中我们会改用DEALER套接字，从而使我们能够任意地发送和接受消息，而不是像REQ套接字那样必须完成发送-接受循环。而DEALER的缺点是我们必须自己管理消息信封。如果你不知道信封是什么，那请阅读第三章。

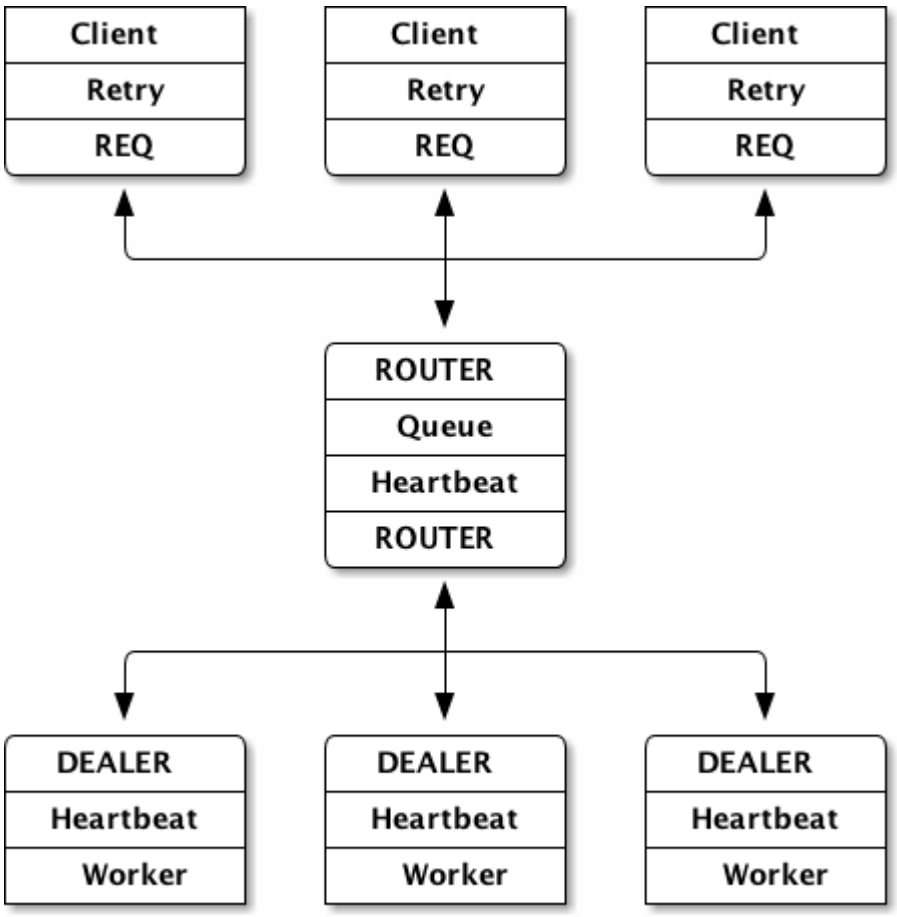

Figure 3 — Paranoid Pirate Pattern

我们仍会使用懒惰海盗模式的client，以下是偏执海盗的队列装置代码：

ppqueue: Paranoid Pirate queue in C

```
//
//  偏执海盗队列
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  心跳健康度，3-5是合理的
#define HEARTBEAT_INTERVAL  1000    //  单位：毫秒

//  偏执海盗协议的消息代码
#define PPP_READY       "\001"      //  worker已就绪
#define PPP_HEARTBEAT   "\002"      //  worker心跳


//  使用以下结构表示worker队列中的一个有效的worker

typedef struct {
    zframe_t *address;          //  worker的地址
    char *identity;             //  可打印的套接字标识
```

```
    int64_t expiry;             //  过期时间
} worker_t;

//  创建新的worker
static worker_t *
s_worker_new (zframe_t *address)
{
    worker_t *self = (worker_t *) zmalloc (sizeof (worker_t));
    self->address = address;
    self->identity = zframe_strdup (address);
    self->expiry = zclock_time () + HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS;
    return self;
}

//  销毁worker结构，包括标识
static void
s_worker_destroy (worker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        worker_t *self = *self_p;
        zframe_destroy (&self->address);
        free (self->identity);
        free (self);
        *self_p = NULL;
    }
}

//  worker已就绪，将其移至列表末尾
static void
s_worker_ready (worker_t *self, zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (streq (self->identity, worker->identity)) {
            zlist_remove (workers, worker);
            s_worker_destroy (&worker);
            break;
        }
        worker = (worker_t *) zlist_next (workers);
    }
    zlist_append (workers, self);
}

//  返回下一个可用的worker地址
static zframe_t *
```

```
s_workers_next (zlist_t *workers)
{
    worker_t *worker = zlist_pop (workers);
    assert (worker);
    zframe_t *frame = worker->address;
    worker->address = NULL;
    s_worker_destroy (&worker);
    return frame;
}

//  寻找并销毁已过期的worker。
//  由于列表中最旧的worker排在最前，所以当找到第一个未过期的worker时就停止。
static void
s_workers_purge (zlist_t *workers)
{
    worker_t *worker = (worker_t *) zlist_first (workers);
    while (worker) {
        if (zclock_time () < worker->expiry)
            break;              //  worker未过期，停止扫描

        zlist_remove (workers, worker);
        s_worker_destroy (&worker);
        worker = (worker_t *) zlist_first (workers);
    }
}


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, "tcp://*:5555");    //  client端点
    zsocket_bind (backend,  "tcp://*:5556");    //  worker端点
    //  List of available workers
    zlist_t *workers = zlist_new ();

    //  规律地发送心跳
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    while (1) {
        zmq_pollitem_t items [] = {
            { backend,  0, ZMQ_POLLIN, 0 },
            { frontend, 0, ZMQ_POLLIN, 0 }
        };
        //  当存在可用worker时轮询前端端点
```

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

        //  处理后端worker请求
        if (items [0].revents & ZMQ_POLLIN) {
            //  使用worker地址进行LRU路由
            zmsg_t *msg = zmsg_recv (backend);
            if (!msg)
                break;          //  中断

            //  worker的任何信号均表示其仍然存活
            zframe_t *address = zmsg_unwrap (msg);
            worker_t *worker = s_worker_new (address);
            s_worker_ready (worker, workers);

            //  处理控制消息，或者将应答转发给client
            if (zmsg_size (msg) == 1) {
                zframe_t *frame = zmsg_first (msg);
                if (memcmp (zframe_data (frame), PPP_READY, 1)
                &&  memcmp (zframe_data (frame), PPP_HEARTBEAT, 1)) {
                    printf ("E: invalid message from worker");
                    zmsg_dump (msg);
                }
                zmsg_destroy (&msg);
            }
            else
                zmsg_send (&msg, frontend);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  获取下一个client请求，交给下一个可用的worker
            zmsg_t *msg = zmsg_recv (frontend);
            if (!msg)
                break;          //  中断
            zmsg_push (msg, s_workers_next (workers));
            zmsg_send (&msg, backend);
        }

        //  发送心跳给空闲的worker
        if (zclock_time () >= heartbeat_at) {
            worker_t *worker = (worker_t *) zlist_first (workers);
            while (worker) {
                zframe_send (&worker->address, backend,
                             ZFRAME_REUSE + ZFRAME_MORE);
                zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
```

```
                zframe_send (&frame, backend, 0);
                worker = (worker_t *) zlist_next (workers);
            }
            heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
        s_workers_purge (workers);
    }

    //  程序结束后进行清理
    while (zlist_size (workers)) {
        worker_t *worker = (worker_t *) zlist_pop (workers);
        s_worker_destroy (&worker);
    }
    zlist_destroy (&workers);
    zctx_destroy (&ctx);
    return 0;
}
```

该队列装置使用心跳机制扩展了LRU模式，看起来很简单，但要想出这个主意还挺难的。下文会更多地介绍心跳机制。

以下是偏执海盗的worker代码：

ppworker: Paranoid Pirate worker in C

```
//
//  偏执海盗worker
//
#include "czmq.h"

#define HEARTBEAT_LIVENESS  3       //  合理值：3-5
#define HEARTBEAT_INTERVAL  1000    //  单位：毫秒
#define INTERVAL_INIT       1000    //  重试间隔
#define INTERVAL_MAX       32000    //  回退算法最大值

//  偏执海盗规范的常量定义
#define PPP_READY       "\001"      //  消息：worker已就绪
#define PPP_HEARTBEAT   "\002"      //  消息：worker心跳

//  返回一个连接至偏执海盗队列装置的套接字

static void *
s_worker_socket (zctx_t *ctx) {
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (worker, "tcp://localhost:5556");
```

```
    //  告知队列worker已准备就绪
    printf ("I: worker已就绪\n");
    zframe_t *frame = zframe_new (PPP_READY, 1);
    zframe_send (&frame, worker, 0);

    return worker;
}

int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *worker = s_worker_socket (ctx);

    //  如果心跳健康度为零，则表示队列装置已死亡
    size_t liveness = HEARTBEAT_LIVENESS;
    size_t interval = INTERVAL_INIT;

    //  规律地发送心跳
    uint64_t heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;

    srandom ((unsigned) time (NULL));
    int cycles = 0;
    while (1) {
        zmq_pollitem_t items [] = { { worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        if (items [0].revents & ZMQ_POLLIN) {
            //  获取消息
            //  - 3段消息，信封+内容，表示一个请求
            //  - 1段消息，表示心跳
            zmsg_t *msg = zmsg_recv (worker);
            if (!msg)
                break;          //  中断

            if (zmsg_size (msg) == 3) {
                //  若干词循环后模拟各种问题
                cycles++;
                if (cycles > 3 && randof (5) == 0) {
                    printf ("I: 模拟崩溃\n");
                    zmsg_destroy (&msg);
                    break;
                }
                else
                if (cycles > 3 && randof (5) == 0) {
```

```
                printf ("I: 模拟CPU过载\n");
                sleep (3);
                if (zctx_interrupted)
                    break;
            }
            printf ("I: 正常应答\n");
            zmsg_send (&msg, worker);
            liveness = HEARTBEAT_LIVENESS;
            sleep (1);              //  做一些处理工作
            if (zctx_interrupted)
                break;
        }
        else
        if (zmsg_size (msg) == 1) {
            zframe_t *frame = zmsg_first (msg);
            if (memcmp (zframe_data (frame), PPP_HEARTBEAT, 1) == 0)
                liveness = HEARTBEAT_LIVENESS;
            else {
                printf ("E: 非法消息\n");
                zmsg_dump (msg);
            }
            zmsg_destroy (&msg);
        }
        else {
            printf ("E: 非法消息\n");
            zmsg_dump (msg);
        }
        interval = INTERVAL_INIT;
    }
    else
    if (--liveness == 0) {
        printf ("W: 心跳失败，无法连接队列装置\n");
        printf ("W: %zd 毫秒后进行重连...\n", interval);
        zclock_sleep (interval);

        if (interval < INTERVAL_MAX)
            interval *= 2;
        zsocket_destroy (ctx, worker);
        worker = s_worker_socket (ctx);
        liveness = HEARTBEAT_LIVENESS;
    }

    //  适时发送心跳给队列
    if (zclock_time () > heartbeat_at) {
        heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        printf ("I: worker心跳\n");
```

```
            zframe_t *frame = zframe_new (PPP_HEARTBEAT, 1);
            zframe_send (&frame, worker, 0);
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 代码中包含了几处失败模拟，和先前一样。这会让代码极难维护，所以当投入使用时，应当移除这些模拟代码。
- 偏执海盗模式中队列的心跳有时会不正常，下文会讲述这一点。
- worker使用了一种类似于懒惰海盗client的重试机制，但有两点不同：1、回退算法设置；2、永不言弃。

尝试运行以下代码，跑通流程：

```
ppqueue &
for i in 1 2 3 4; do
    ppworker &
    sleep 1
done
lpclient &
```

你会看到worker逐个崩溃，client在多次尝试后放弃。你可以停止并重启队列装置，client和worker会相继重连，并正确地发送、处理和接收请求，顺序不会混乱。所以说，整个通信过程只有两种情形：交互成功，或client最终放弃。

心跳

当我在写偏执海盗模式的示例时，大约花了五个小时的时间来协调队列至worker的心跳，剩下的请求-应答链路只花了约10分钟的时间。心跳机制在可靠性上带来的益处有时还不及它所引发的问题。使用过程中很有可能会产生“虚假故障”的情况，即节点误认为他们已失去连接，因为心跳没有正确地发送。

在理解和实施心跳时，需要考虑以下几点：

- 心跳不是一种请求-应答，它们异步地在节点之间传递，任一节点都可以通过它来判断对方已经死亡，并中止通信。

- 如果某个节点使用持久套接字（即设定了套接字标识），意味着发送给它的心跳可能会堆砌，并在重连后一起收到。所以说，worker不应该使用持久套接字。示例代码使用持久套接字是为了便于调试，而且代码中使用了随机的套接字标识，避免重用之前的标识。

- 使用过程中，应先让心跳工作起来，再进行后面的消息处理。你需要保证启动任一节点后，心跳都能正确地执行。停止并重启他们，模拟冻结、崩溃等情况来进行测试。

- 当你的主循环使用了zmq_poll()，则应该使用另一个计时器来触发心跳。不要使用主循环来控制心跳的发送，这回导致过量地发送心跳（阻塞网络），或是发送得太少（导致节点断开）。zhelpers包提供了s_clock()函数返回当前系统时间戳，单位是毫秒，可以用它来控制心跳的发送间隔。C代码如下：

```
// 规律地发送心跳
uint64_t heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
while (1) {
    …
    zmq_poll (items, 1, HEARTBEAT_INTERVAL * 1000);
    …
    // 无论zmq_poll的行为是什么，都使用以下逻辑判断是否发送心跳
    if (s_clock () > heartbeat_at) {
        … 发送心跳给所有节点
        // 设置下一次心跳的时间
        heartbeat_at = s_clock () + HEARTBEAT_INTERVAL;
    }
}
```

- 主循环应该使用心跳间隔作为超时时间。显然不能使用无超时时间的设置，而短于心跳间隔也只是浪费循环次数而已。

- 使用简单的追踪方式来进行追踪，如直接输出至控制台。这里有一些追踪的窍门：使用zmsg()函数打印套接字内容；对消息进行编号，判断是否会有间隔。

- 在真实的应用程序中，心跳必须是可以配置的，并能和节点共同商定。有些节点需要高频心跳，如10毫秒，另一些节点则可能只需要30秒发送一次心跳即可。

- 如果你要对不同的节点发送不同频率的心跳，那么poll的超时时间应设置为最短的心跳间隔。

- 也许你会想要用一个单独的套接字来处理心跳，这看起来很棒，可以将同步的请求-应答和异步的心跳隔离开来。但是，这个主意并不好，原因有几点：首先、发送数据时其实是不需要发送心跳的；其次、套接字可能会因为网络问题而阻塞，你需要设法知道用于发送数据的套接字停止响应的原因是死亡了还是过于繁忙而已，这样你就需要对这个套接字进行心跳。最后，处理两个套接字要比处理一个复杂得多。

- 我们没有设置client至队列的心跳，因为这太过复杂了，而且没有太大价值。

约定和协议

也许你已经注意到，由于心跳机制，偏执海盗模式和简单海盗模式是不兼容的。

其实，这里我们需要写一个协议。也许在试验阶段是不需要协议的，但这在真实的应用程序中是有必要。如果我们想用其他语言来写worker怎么办？我们是否需要通过源代码来查看通信过程？如果我们想改变协议怎么办？规范可能很简单，但并不显然。越是成功的协议，就会越为复杂。

一个缺乏约定的应用程序一定是不可复用的，所以让我们来为这个协议写一个规范，怎么做呢？

- 位于rfc.zeromq.org的wiki页上，我们特地设置了一个用于存放ZMQ协议的页面。
- 要创建一个新的协议，你需要注册并按照指导进行。过程很直接，但并不一定所有人都能撰写技术性文档。

我大约花了15分钟的时间草拟海盗模式规范（PPP），麻雀虽小，但五脏俱全。

要用PPP协议进行真实环境下的编程，你还需要：

- 在READY命令中加入版本号，这样就能再日后安全地新增PPP版本号。
- 目前，READY和HEARTBEAT信号并没有指定其来源于请求还是应答。要区分他们，需要新建一个消息结构，其中包含“消息类型”这一信息。

面向服务的可靠队列（管家模式）

世上的事物往往瞬息万变，正当我们期待有更好的协议来解决上一节的问题时，已经有人制定好了：

- http://rfc.zeromq.org/spec:7

这份协议只有一页，它将PPP协议变得更为坚固。我们在设计复杂架构时应该这样做：首先写下约定，再用软件去实现它。

管家模式协议（MDP）在扩展PPP协议时引入了一个有趣的特性：client发送的每一个请求都有一个“服务名称”，而worker在像队列装置注册时需要告知自己的服务类型。MDP的优势在于它来源于现实编程，协议简单，且容易提升。

引入“服务名称”的机制，是对偏执海盗队列的一个简单补充，而结果是让其成为一个面向服务的代理。

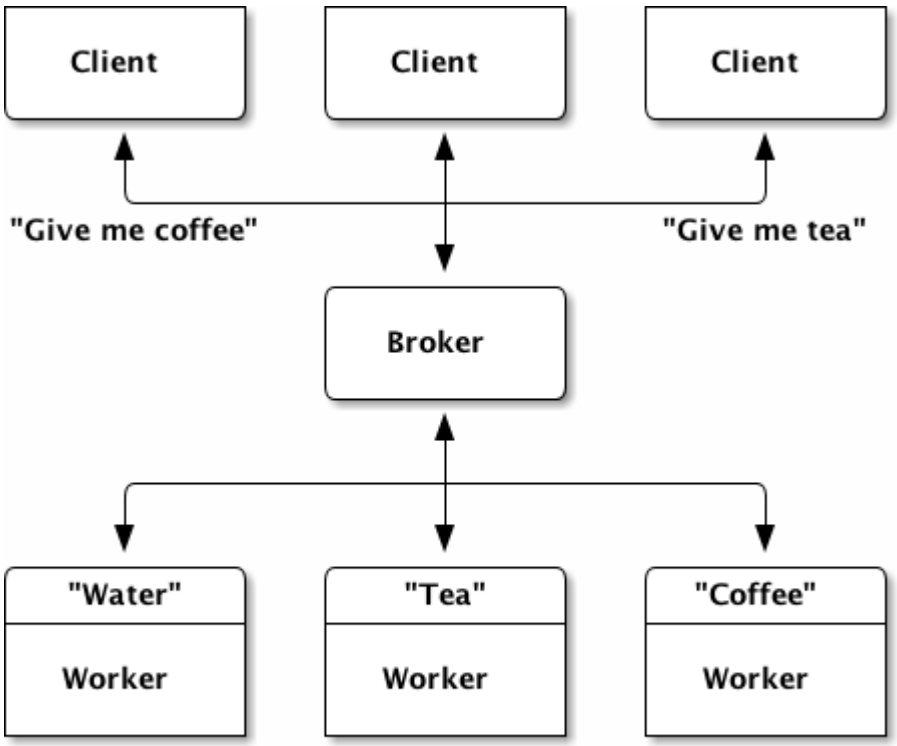

Figure 4 — Majordomo Pattern

在实施管家模式之前，我们需要为client和worker编写一个框架。如果程序员可以通过简单的API来实现这种模式，那就没有必要让他们去了解管家模式的协议内容和实现方法了。
所以，我们第一个协议（即管家模式协议）定义了分布式架构中节点是如何互相交互的，第二个协议则要定义应用程序应该如何通过框架来使用这一协议。
管家模式有两个端点，客户端和服务端。因为我们要为client和worker都撰写框架，所以就需要提供两套API。
以下是用简单的面向对象方法设计的client端API雏形，使用的是C语言的ZFL library。

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
zmsg_t  *mdcli_send    (mdcli_t *self, char *service, zmsg_t **request_p);
```

就这么简单。我们创建了一个会话来和代理通信，发送并接收一个请求，最后关闭连接。以下是worker端API的雏形。

```
mdwrk_t *mdwrk_new     (char *broker,char *service);
void     mdwrk_destroy (mdwrk_t **self_p);
zmsg_t  *mdwrk_recv    (mdwrk_t *self, zmsg_t *reply);
```

上面两段代码看起来差不多，但是worker端API略有不同。worker第一次执行recv()后会传递一个空的应答，之后才传递当前的应答，并获得新的请求。

两段的API都很容易开发，只需在偏执海盗模式代码的基础上修改即可。以下是client API：

mdcliapi: Majordomo client API in C

```c
/*  =====================================================================
    mdcliapi.c

    Majordomo Protocol Client API
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it under
    the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation; either version 3 of the License, or (at
    your option) any later version.

    This software is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/

#include "mdcliapi.h"

//  类结构
//  我们会通过成员方法来访问这些属性

struct _mdcli_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    void *client;               //  连接至代理的套接字
    int verbose;                //  使用标准输出打印当前活动
    int timeout;                //  请求超时时间
    int retries;                //  请求重试次数
};



//  ---------------------------------------------------------------------
//  连接或重连代理
```

```c
void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_REQ);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: 正在连接至代理 %s...", self->broker);
}


//  ---------------------------------------------------------------------
//  构造函数

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  毫秒
    self->retries = 3;              //  尝试次数

    s_mdcli_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  --------------------------------------------------------------------------
//  设定请求超时时间

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}


//  --------------------------------------------------------------------------
//  设定请求重试次数

void
mdcli_set_retries (mdcli_t *self, int retries)
{
    assert (self);
    self->retries = retries;
}


//  --------------------------------------------------------------------------
//  向代理发送请求，并尝试获取应答；
//  对消息保持所有权，发送后销毁；
//  返回应答消息，或NULL。

zmsg_t *
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  用协议前缀包装消息
    //  Frame 1: "MDPCxy" (six bytes, MDP/Client x.y)
    //  Frame 2: 服务名称（可打印字符串）
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    if (self->verbose) {
        zclock_log ("I: 发送请求给 '%s' 服务:", service);
        zmsg_dump (request);
    }
```

```
    int retries_left = self->retries;
    while (retries_left && !zctx_interrupted) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->client);

        while (TRUE) {
            //  轮询套接字以接收应答，有超时时间
            zmq_pollitem_t items [] = {
                { self->client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断

            //  收到应答后进行处理
            if (items [0].revents & ZMQ_POLLIN) {
                zmsg_t *msg = zmsg_recv (self->client);
                if (self->verbose) {
                    zclock_log ("I: received reply:");
                    zmsg_dump (msg);
                }
                //  不要尝试处理错误，直接报错即可
                assert (zmsg_size (msg) >= 3);

                zframe_t *header = zmsg_pop (msg);
                assert (zframe_streq (header, MDPC_CLIENT));
                zframe_destroy (&header);

                zframe_t *reply_service = zmsg_pop (msg);
                assert (zframe_streq (reply_service, service));
                zframe_destroy (&reply_service);

                zmsg_destroy (&request);
                return msg;     //  成功
            }
            else
            if (--retries_left) {
                if (self->verbose)
                    zclock_log ("W: no reply, reconnecting...");
                //  重连并重发消息
                s_mdcli_connect_to_broker (self);
                zmsg_t *msg = zmsg_dup (request);
                zmsg_send (&msg, self->client);
            }
            else {
                if (self->verbose)
                    zclock_log ("W: 发生严重错误，放弃重试。");
```

```
                break;          //  放弃
            }
        }
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，结束client进程...\n");
    zmsg_destroy (&request);
    return NULL;
}
```

以下测试程序会执行10万次请求应答：

mdclient: Majordomo client application in C

```
//
//  管家模式协议 - 客户端示例
//  使用mdcli API隐藏管家模式协议的内部实现
//

//  让我们直接编译这段代码，不生成类库
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        zmsg_t *reply = mdcli_send (session, "echo", &request);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  中断或停止
    }
    printf ("已处理 %d 次请求-应答\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

下面是worker的API：

mdwrkapi: Majordomo worker API in C

```c
/*  =====================================================================
    mdwrkapi.c

    Majordomo Protocol Worker API
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it under
    the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation; either version 3 of the License, or (at
    your option) any later version.

    This software is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/

#include "mdwrkapi.h"

//  可靠性参数
#define HEARTBEAT_LIVENESS  3       //  合理值：3-5

//  类结构
//  使用成员函数访问属性

struct _mdwrk_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    char *service;
    void *worker;               //  连接至代理的套接字
    int verbose;                //  使用标准输出打印活动

    //  心跳设置
    uint64_t heartbeat_at;      //  发送心跳的时间
    size_t liveness;            //  尝试次数
```

```
    int heartbeat;              //  心跳延时，单位：毫秒
    int reconnect;              //  重连延时，单位：毫秒

    //  内部状态
    int expect_reply;           //  初始值为0

    //  应答地址，如果存在的话
    zframe_t *reply_to;
};


//  ---------------------------------------------------------------------
//  发送消息给代理
//  如果没有提供消息，则内部创建一个

static void
s_mdwrk_send_to_broker (mdwrk_t *self, char *command, char *option,
                        zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  将协议信封压入消息顶部
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);
    zmsg_pushstr (msg, "");

    if (self->verbose) {
        zclock_log ("I: sending %s to broker",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->worker);
}


//  ---------------------------------------------------------------------
//  连接或重连代理

void s_mdwrk_connect_to_broker (mdwrk_t *self)
{
    if (self->worker)
        zsocket_destroy (self->ctx, self->worker);
    self->worker = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->worker, self->broker);
```

```
    if (self->verbose)
        zclock_log ("I: 正在连接代理 %s...", self->broker);

    //  向代理注册服务类型
    s_mdwrk_send_to_broker (self, MDPW_READY, self->service, NULL);

    //  当心跳健康度为零，表示代理已断开连接
    self->liveness = HEARTBEAT_LIVENESS;
    self->heartbeat_at = zclock_time () + self->heartbeat;
}


//  ---------------------------------------------------------------------
//  构造函数

mdwrk_t *
mdwrk_new (char *broker,char *service, int verbose)
{
    assert (broker);
    assert (service);

    mdwrk_t *self = (mdwrk_t *) zmalloc (sizeof (mdwrk_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->service = strdup (service);
    self->verbose = verbose;
    self->heartbeat = 2500;     //  毫秒
    self->reconnect = 2500;     //  毫秒

    s_mdwrk_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
mdwrk_destroy (mdwrk_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdwrk_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self->broker);
        free (self->service);
```

```
        free (self);
        *self_p = NULL;
    }
}


//  --------------------------------------------------------------------------
//  设置心跳延迟

void
mdwrk_set_heartbeat (mdwrk_t *self, int heartbeat)
{
    self->heartbeat = heartbeat;
}


//  --------------------------------------------------------------------------
//  设置重连延迟

void
mdwrk_set_reconnect (mdwrk_t *self, int reconnect)
{
    self->reconnect = reconnect;
}


//  --------------------------------------------------------------------------
//  若有应答则发送给代理，并等待新的请求

zmsg_t *
mdwrk_recv (mdwrk_t *self, zmsg_t **reply_p)
{
    //  格式化并发送请求传入的应答
    assert (reply_p);
    zmsg_t *reply = *reply_p;
    assert (reply || !self->expect_reply);
    if (reply) {
        assert (self->reply_to);
        zmsg_wrap (reply, self->reply_to);
        s_mdwrk_send_to_broker (self, MDPW_REPLY, NULL, reply);
        zmsg_destroy (reply_p);
    }
    self->expect_reply = 1;

    while (TRUE) {
        zmq_pollitem_t items [] = {
```

```
            { self->worker,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, self->heartbeat * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->worker);
            if (!msg)
                break;          //  中断
            if (self->verbose) {
                zclock_log ("I: 从代理处获得消息:");
                zmsg_dump (msg);
            }
            self->liveness = HEARTBEAT_LIVENESS;

            //  不要处理错误，直接报错即可
            assert (zmsg_size (msg) >= 3);

            zframe_t *empty = zmsg_pop (msg);
            assert (zframe_streq (empty, ""));
            zframe_destroy (&empty);

            zframe_t *header = zmsg_pop (msg);
            assert (zframe_streq (header, MDPW_WORKER));
            zframe_destroy (&header);

            zframe_t *command = zmsg_pop (msg);
            if (zframe_streq (command, MDPW_REQUEST)) {
                //  这里需要将消息中空帧之前的所有地址都保存起来，
                //  但在这里我们暂时只保存一个
                self->reply_to = zmsg_unwrap (msg);
                zframe_destroy (&command);
                return msg;     //  处理请求
            }
            else
            if (zframe_streq (command, MDPW_HEARTBEAT))
                ;               //  不对心跳做任何处理
            else
            if (zframe_streq (command, MDPW_DISCONNECT))
                s_mdwrk_connect_to_broker (self);
            else {
                zclock_log ("E: 消息不合法");
                zmsg_dump (msg);
            }
            zframe_destroy (&command);
            zmsg_destroy (&msg);
```

```
        }
        else
        if (--self->liveness == 0) {
            if (self->verbose)
                zclock_log ("W: 失去与代理的连接 - 正在重试...");
            zclock_sleep (self->reconnect);
            s_mdwrk_connect_to_broker (self);
        }
        //  适时地发送消息
        if (zclock_time () > self->heartbeat_at) {
            s_mdwrk_send_to_broker (self, MDPW_HEARTBEAT, NULL, NULL);
            self->heartbeat_at = zclock_time () + self->heartbeat;
        }
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，中止worker...\n");
    return NULL;
}
```

以下测试程序实现了名为echo的服务：

mdworker: Majordomo worker application in C

```
//
//  管家模式协议 - worker示例
//  使用mdwrk API隐藏MDP协议的内部实现
//

//  让我们直接编译代码，而不创建类库
#include "mdwrkapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdwrk_t *session = mdwrk_new (
        "tcp://localhost:5555", "echo", verbose);

    zmsg_t *reply = NULL;
    while (1) {
        zmsg_t *request = mdwrk_recv (session, &reply);
        if (request == NULL)
            break;              //  worker被中止
        reply = request;        //  echo服务……其实很复杂:)
    }
    mdwrk_destroy (&session);
```

```
    return 0;
}
```

几点说明：

- API是单线程的，所以说worker不会再后台发送心跳，而这也是我们所期望的：如果worker应用程序停止了，心跳就会跟着中止，代理便会停止向该worker发送新的请求。

- wroker API没有做回退算法的设置，因为这里不值得使用这一复杂的机制。

- API没有提供任何报错机制，如果出现问题，它会直接报断言（或异常，依语言而定）。这一做法对实验性的编程是有用的，这样可以立刻看到执行结果。但在真实编程环境中，API应该足够健壮，合适地处理非法消息。

也许你会问，worker API为什么要关闭它的套接字并新开一个呢？特别是ZMQ是有重连机制的，能够在节点归来后进行重连。我们可以回顾一下简单海盗模式中的worker，以及偏执海盗模式中的worker来加以理解。ZMQ确实会进行自动重连，但如果代理死亡并重连，worker并不会重新进行注册。这个问题有两种解决方案：一是我们这里用到的较为简便的方案，即当worker判断代理已经死亡时，关闭它的套接字并重头来过；另一个方案是当代理收到未知worker的心跳时要求该worker对其提供的服务类型进行注册，这样一来就需要在协议中说明这一规则。

下面让我们设计管家模式的代理，它的核心代码是一组队列，每种服务对应一个队列。我们会在worker出现时创建相应的队列（worker消失时应该销毁对应的队列，不过我们这里暂时不考虑）。额外的，我们会为每种服务维护一个worker的队列。

为了让C语言代码更为易读易写，我使用了ZFL项目提供的哈希和链表容器，并命名为[zhash](https://github.com/imatix/zguide/blob/master/examples/C/zhash.h zhash)和zlist。如果使用现代语言编写，那自然可以使用其内置的容器。

mdbroker: Majordomo broker in C

```
//
//  管家模式协议 - 代理
//  协议 http://rfc.zeromq.org/spec:7 和 spec:8 的最简实现
//
#include "czmq.h"
#include "mdp.h"

//  一般我们会从配置文件中获取以下值

#define HEARTBEAT_LIVENESS  3       //  合理值：3-5
#define HEARTBEAT_INTERVAL  2500    //  单位：毫秒
#define HEARTBEAT_EXPIRY    HEARTBEAT_INTERVAL * HEARTBEAT_LIVENESS

//  定义一个代理
```

```
typedef struct {
    zctx_t *ctx;                //  上下文
    void *socket;               //  用于连接client和worker的套接字
    int verbose;                //  使用标准输出打印活动信息
    char *endpoint;             //  代理绑定到的端点
    zhash_t *services;          //  已知服务的哈希表
    zhash_t *workers;           //  已知worker的哈希表
    zlist_t *waiting;           //  正在等待的worker队列
    uint64_t heartbeat_at;      //  发送心跳的时间
} broker_t;

//  定义一个服务
typedef struct {
    char *name;                 //  服务名称
    zlist_t *requests;          //  客户端请求队列
    zlist_t *waiting;           //  正在等待的worker队列
    size_t workers;             //  可用worker数
} service_t;

//  定义一个worker，状态为空闲或占用
typedef struct {
    char *identity;             //  worker的标识
    zframe_t *address;          //  地址帧
    service_t *service;         //  所属服务
    int64_t expiry;             //  过期时间，从未收到心跳起计时
} worker_t;


//  ---------------------------------------------------------------------
//  代理使用的函数
static broker_t *
    s_broker_new (int verbose);
static void
    s_broker_destroy (broker_t **self_p);
static void
    s_broker_bind (broker_t *self, char *endpoint);
static void
    s_broker_purge_workers (broker_t *self);

//  服务使用的函数
static service_t *
    s_service_require (broker_t *self, zframe_t *service_frame);
static void
    s_service_destroy (void *argument);
static void
    s_service_dispatch (broker_t *self, service_t *service, zmsg_t *msg);
```

```
static void
    s_service_internal (broker_t *self, zframe_t *service_frame, zmsg_t *msg);

//  worker使用的函数
static worker_t *
    s_worker_require (broker_t *self, zframe_t *address);
static void
    s_worker_delete (broker_t *self, worker_t *worker, int disconnect);
static void
    s_worker_destroy (void *argument);
static void
    s_worker_process (broker_t *self, zframe_t *sender, zmsg_t *msg);
static void
    s_worker_send (broker_t *self, worker_t *worker, char *command,
                   char *option, zmsg_t *msg);
static void
    s_worker_waiting (broker_t *self, worker_t *worker);

//  客户端使用的函数
static void
    s_client_process (broker_t *self, zframe_t *sender, zmsg_t *msg);


//  ---------------------------------------------------------------------
//  主程序

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    broker_t *self = s_broker_new (verbose);
    s_broker_bind (self, "tcp://*:5555");

    //  接受并处理消息，直至程序被中止
    while (TRUE) {
        zmq_pollitem_t items [] = {
            { self->socket,  0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, HEARTBEAT_INTERVAL * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断

        //  Process next input message, if any
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (self->socket);
            if (!msg)
                break;          //  中断
```

```
            if (self->verbose) {
                zclock_log ("I: 收到消息:");
                zmsg_dump (msg);
            }
            zframe_t *sender = zmsg_pop (msg);
            zframe_t *empty  = zmsg_pop (msg);
            zframe_t *header = zmsg_pop (msg);

            if (zframe_streq (header, MDPC_CLIENT))
                s_client_process (self, sender, msg);
            else
            if (zframe_streq (header, MDPW_WORKER))
                s_worker_process (self, sender, msg);
            else {
                zclock_log ("E: 非法消息:");
                zmsg_dump (msg);
                zmsg_destroy (&msg);
            }
            zframe_destroy (&sender);
            zframe_destroy (&empty);
            zframe_destroy (&header);
        }
        //  断开并删除过期的worker
        //  适时地发送心跳给worker
        if (zclock_time () > self->heartbeat_at) {
            s_broker_purge_workers (self);
            worker_t *worker = (worker_t *) zlist_first (self->waiting);
            while (worker) {
                s_worker_send (self, worker, MDPW_HEARTBEAT, NULL, NULL);
                worker = (worker_t *) zlist_next (self->waiting);
            }
            self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
        }
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，关闭中...\n");

    s_broker_destroy (&self);
    return 0;
}


//  ---------------------------------------------------------------------
//  代理对象的构造函数

static broker_t *
```

```c
s_broker_new (int verbose)
{
    broker_t *self = (broker_t *) zmalloc (sizeof (broker_t));

    //  初始化代理状态
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->verbose = verbose;
    self->services = zhash_new ();
    self->workers = zhash_new ();
    self->waiting = zlist_new ();
    self->heartbeat_at = zclock_time () + HEARTBEAT_INTERVAL;
    return self;
}

//  ---------------------------------------------------------------------
//  代理对象的析构函数

static void
s_broker_destroy (broker_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        broker_t *self = *self_p;
        zctx_destroy (&self->ctx);
        zhash_destroy (&self->services);
        zhash_destroy (&self->workers);
        zlist_destroy (&self->waiting);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
//  将代理套接字绑定至端点，可以重复调用该函数
//  我们使用一个套接字来同时处理client和worker

void
s_broker_bind (broker_t *self, char *endpoint)
{
    zsocket_bind (self->socket, endpoint);
    zclock_log ("I: MDP broker/0.1.1 is active at %s", endpoint);
}

//  ---------------------------------------------------------------------
//  删除空闲状态中过期的worker
```

```
static void
s_broker_purge_workers (broker_t *self)
{
    worker_t *worker = (worker_t *) zlist_first (self->waiting);
    while (worker) {
        if (zclock_time () < worker->expiry)
            continue;               //  该worker未过期，停止搜索
        if (self->verbose)
            zclock_log ("I: 正在删除过期的worker: %s",
                worker->identity);

        s_worker_delete (self, worker, 0);
        worker = (worker_t *) zlist_first (self->waiting);
    }
}

//  ---------------------------------------------------------------------
//  定位或创建新的服务项

static service_t *
s_service_require (broker_t *self, zframe_t *service_frame)
{
    assert (service_frame);
    char *name = zframe_strdup (service_frame);

    service_t *service =
        (service_t *) zhash_lookup (self->services, name);
    if (service == NULL) {
        service = (service_t *) zmalloc (sizeof (service_t));
        service->name = name;
        service->requests = zlist_new ();
        service->waiting = zlist_new ();
        zhash_insert (self->services, name, service);
        zhash_freefn (self->services, name, s_service_destroy);
        if (self->verbose)
            zclock_log ("I: 收到消息:");
    }
    else
        free (name);

    return service;
}

//  ---------------------------------------------------------------------
//  当服务从broker->services中移除时销毁该服务对象
```

```
static void
s_service_destroy (void *argument)
{
    service_t *service = (service_t *) argument;
    //  销毁请求队列中的所有项目
    while (zlist_size (service->requests)) {
        zmsg_t *msg = zlist_pop (service->requests);
        zmsg_destroy (&msg);
    }
    zlist_destroy (&service->requests);
    zlist_destroy (&service->waiting);
    free (service->name);
    free (service);
}

//  ---------------------------------------------------------------------
//  可能时，分发请求给等待中的worker

static void
s_service_dispatch (broker_t *self, service_t *service, zmsg_t *msg)
{
    assert (service);
    if (msg)                    //  将消息加入队列
        zlist_append (service->requests, msg);

    s_broker_purge_workers (self);
    while (zlist_size (service->waiting)
        && zlist_size (service->requests))
    {
        worker_t *worker = zlist_pop (service->waiting);
        zlist_remove (self->waiting, worker);
        zmsg_t *msg = zlist_pop (service->requests);
        s_worker_send (self, worker, MDPW_REQUEST, NULL, msg);
        zmsg_destroy (&msg);
    }
}

//  ---------------------------------------------------------------------
//  使用8/MMI协定处理内部服务

static void
s_service_internal (broker_t *self, zframe_t *service_frame, zmsg_t *msg)
{
    char *return_code;
    if (zframe_streq (service_frame, "mmi.service")) {
```

```
        char *name = zframe_strdup (zmsg_last (msg));
        service_t *service =
            (service_t *) zhash_lookup (self->services, name);
        return_code = service && service->workers? "200": "404";
        free (name);
    }
    else
        return_code = "501";

    zframe_reset (zmsg_last (msg), return_code, strlen (return_code));

    //  移除并保存返回给client的信封，插入协议头信息和服务名称，并重新包装信封
    zframe_t *client = zmsg_unwrap (msg);
    zmsg_push (msg, zframe_dup (service_frame));
    zmsg_pushstr (msg, MDPC_CLIENT);
    zmsg_wrap (msg, client);
    zmsg_send (&msg, self->socket);
}

//  ---------------------------------------------------------------------
//  按需创建worker

static worker_t *
s_worker_require (broker_t *self, zframe_t *address)
{
    assert (address);

    //  self->workers使用wroker的标识为键
    char *identity = zframe_strhex (address);
    worker_t *worker =
        (worker_t *) zhash_lookup (self->workers, identity);

    if (worker == NULL) {
        worker = (worker_t *) zmalloc (sizeof (worker_t));
        worker->identity = identity;
        worker->address = zframe_dup (address);
        zhash_insert (self->workers, identity, worker);
        zhash_freefn (self->workers, identity, s_worker_destroy);
        if (self->verbose)
            zclock_log ("I: 正在注册新的worker: %s", identity);
    }
    else
        free (identity);
    return worker;
}
```

```
//  --------------------------------------------------------------------------
//  从所有数据结构中删除wroker，并销毁worker对象

static void
s_worker_delete (broker_t *self, worker_t *worker, int disconnect)
{
    assert (worker);
    if (disconnect)
        s_worker_send (self, worker, MDPW_DISCONNECT, NULL, NULL);

    if (worker->service) {
        zlist_remove (worker->service->waiting, worker);
        worker->service->workers--;
    }
    zlist_remove (self->waiting, worker);
    //  以下方法间接调用了s_worker_destroy()方法
    zhash_delete (self->workers, worker->identity);
}

//  --------------------------------------------------------------------------
//  当worker从broker->workers中移除时，销毁worker对象

static void
s_worker_destroy (void *argument)
{
    worker_t *worker = (worker_t *) argument;
    zframe_destroy (&worker->address);
    free (worker->identity);
    free (worker);
}

//  --------------------------------------------------------------------------
//  处理worker发送来的消息

static void
s_worker_process (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 1);     //  消息中至少包含命令帧

    zframe_t *command = zmsg_pop (msg);
    char *identity = zframe_strhex (sender);
    int worker_ready = (zhash_lookup (self->workers, identity) != NULL);
    free (identity);
    worker_t *worker = s_worker_require (self, sender);

    if (zframe_streq (command, MDPW_READY)) {
```

```
        //  若worker队列中已有该worker，但仍收到了它的“已就绪”消息，则删除这个worker。
        if (worker_ready)
            s_worker_delete (self, worker, 1);
        else
        if (zframe_size (sender) >= 4  //  服务名称为保留的服务
        &&  memcmp (zframe_data (sender), "mmi.", 4) == 0)
            s_worker_delete (self, worker, 1);
        else {
            //  将worker对应到服务，并置为空闲状态
            zframe_t *service_frame = zmsg_pop (msg);
            worker->service = s_service_require (self, service_frame);
            worker->service->workers++;
            s_worker_waiting (self, worker);
            zframe_destroy (&service_frame);
        }
    }
    else
    if (zframe_streq (command, MDPW_REPLY)) {
        if (worker_ready) {
            //  移除并保存返回给client的信封，插入协议头信息和服务名称，并重新包装信封
            zframe_t *client = zmsg_unwrap (msg);
            zmsg_pushstr (msg, worker->service->name);
            zmsg_pushstr (msg, MDPC_CLIENT);
            zmsg_wrap (msg, client);
            zmsg_send (&msg, self->socket);
            s_worker_waiting (self, worker);
        }
        else
            s_worker_delete (self, worker, 1);
    }
    else
    if (zframe_streq (command, MDPW_HEARTBEAT)) {
        if (worker_ready)
            worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
        else
            s_worker_delete (self, worker, 1);
    }
    else
    if (zframe_streq (command, MDPW_DISCONNECT))
        s_worker_delete (self, worker, 0);
    else {
        zclock_log ("E: 非法消息");
        zmsg_dump (msg);
    }
    free (command);
    zmsg_destroy (&msg);
```

```
}

//  --------------------------------------------------------------------
//  发送消息给worker
//  如果指针指向了一条消息，发送它，但不销毁它，因为这是调用者的工作

static void
s_worker_send (broker_t *self, worker_t *worker, char *command,
               char *option, zmsg_t *msg)
{
    msg = msg? zmsg_dup (msg): zmsg_new ();

    //  将协议信封压入消息顶部
    if (option)
        zmsg_pushstr (msg, option);
    zmsg_pushstr (msg, command);
    zmsg_pushstr (msg, MDPW_WORKER);

    //  在消息顶部插入路由帧
    zmsg_wrap (msg, zframe_dup (worker->address));

    if (self->verbose) {
        zclock_log ("I: 正在发送消息给worker %s",
            mdps_commands [(int) *command]);
        zmsg_dump (msg);
    }
    zmsg_send (&msg, self->socket);
}

//  --------------------------------------------------------------------
//  正在等待的worker

static void
s_worker_waiting (broker_t *self, worker_t *worker)
{
    //  将worker加入代理和服务的等待队列
    zlist_append (self->waiting, worker);
    zlist_append (worker->service->waiting, worker);
    worker->expiry = zclock_time () + HEARTBEAT_EXPIRY;
    s_service_dispatch (self, worker->service, NULL);
}

//  --------------------------------------------------------------------
//  处理client发来的请求

static void
```

```
s_client_process (broker_t *self, zframe_t *sender, zmsg_t *msg)
{
    assert (zmsg_size (msg) >= 2);      //  服务名称 + 请求内容

    zframe_t *service_frame = zmsg_pop (msg);
    service_t *service = s_service_require (self, service_frame);

    //  为应答内容设置请求方的地址
    zmsg_wrap (msg, zframe_dup (sender));
    if (zframe_size (service_frame) >= 4
    &&  memcmp (zframe_data (service_frame), "mmi.", 4) == 0)
        s_service_internal (self, service_frame, msg);
    else
        s_service_dispatch (self, service, msg);
    zframe_destroy (&service_frame);
}
```

这个例子应该是我们见过最复杂的一个示例了，大约有500行代码。编写这段代码并让其变的健壮，大约花费了两天的时间。但是，这也仅仅是一个完整的面向服务代理的一部分。

几点说明：

- 管家模式协议要求我们在一个套接字中同时处理client和worker，这一点对部署和管理代理很有益处：它只会在一个ZMQ端点上收发请求，而不是两个。
- 代理很好地实现了MDP/0.1协议中规范的内容，包括当代理发送非法命令和心跳时断开的机制。
- 可以将这段代码扩充为多线程，每个线程管理一个套接字、一组client和worker。这种做法在大型架构的拆分中显得很有趣。C语言代码已经是这样的格式了，因此很容易实现。
- 还可以将这段代码扩充为主备模式、双在线模式，进一步提高可靠性。因为从本质上来说，代理是无状态的，只是保存了服务的存在与否，因此client和worker可以自行选择除此之外的代理来进行通信。
- 示例代码中心跳的间隔为5秒，主要是为了减少调试时的输出。现实中的值应该设得低一些，但是，重试的过程应该设置得稍长一些，让服务有足够的时间启动，如10秒钟。

异步管家模式

上文那种实现管家模式的方法比较简单，client还是简单海盗模式中的，仅仅是用API重写了一下。我在测试机上运行了程序，处理10万条请求大约需要14秒的时间，这和代码也有一些关系，因为复制消息帧的时间浪费了CPU处理时间。但真正的问题在于，我们总是逐个循环进行处理（round-trip），即发送-接收-发送-接收.....ZMQ内部禁用了TCP发包优化算法（Nagle's algorithm），但逐个处理循环还是比较浪费。

理论归理论，还是需要由实践来检验。我们用一个简单的测试程序来看看逐个处理循环是否真的耗时。这个测试程序会发送一组消息，第一次它发一条收一条，第二次则一起发送再一起接收。两次结果应该是一样的，但速度截然不同。

tripping: Round-trip demonstrator in C

```
//
//  Round-trip 模拟
//
//  本示例程序使用多线程的方式启动client、worker、以及代理，
//  当client处理完毕时会发送信号给主程序。
//
#include "czmq.h"

static void
client_task (void *args, zctx_t *ctx, void *pipe)
{
    void *client = zsocket_new (ctx, ZMQ_DEALER);
    zmq_setsockopt (client, ZMQ_IDENTITY, "C", 1);
    zsocket_connect (client, "tcp://localhost:5555");

    printf ("开始测试...\n");
    zclock_sleep (100);

    int requests;
    int64_t start;

    printf ("同步 round-trip 测试...\n");
    start = zclock_time ();
    for (requests = 0; requests < 10000; requests++) {
        zstr_send (client, "hello");
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d 次/秒\n",
        (1000 * 10000) / (int) (zclock_time () - start));

    printf ("异步 round-trip 测试...\n");
    start = zclock_time ();
    for (requests = 0; requests < 100000; requests++)
        zstr_send (client, "hello");
    for (requests = 0; requests < 100000; requests++) {
        char *reply = zstr_recv (client);
        free (reply);
    }
    printf (" %d 次/秒\n",
```

```
        (1000 * 100000) / (int) (zclock_time () - start));

    zstr_send (pipe, "完成");
}

static void *
worker_task (void *args)
{
    zctx_t *ctx = zctx_new ();
    void *worker = zsocket_new (ctx, ZMQ_DEALER);
    zmq_setsockopt (worker, ZMQ_IDENTITY, "W", 1);
    zsocket_connect (worker, "tcp://localhost:5556");

    while (1) {
        zmsg_t *msg = zmsg_recv (worker);
        zmsg_send (&msg, worker);
    }
    zctx_destroy (&ctx);
    return NULL;
}

static void *
broker_task (void *args)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    void *backend = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (frontend, "tcp://*:5555");
    zsocket_bind (backend,  "tcp://*:5556");

    //  初始化轮询对象
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    while (1) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1)
            break;              //  中断
        if (items [0].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (frontend);
            zframe_t *address = zmsg_pop (msg);
            zframe_destroy (&address);
            zmsg_pushstr (msg, "W");
            zmsg_send (&msg, backend);
```

```
        }
        if (items [1].revents & ZMQ_POLLIN) {
            zmsg_t *msg = zmsg_recv (backend);
            zframe_t *address = zmsg_pop (msg);
            zframe_destroy (&address);
            zmsg_pushstr (msg, "C");
            zmsg_send (&msg, frontend);
        }
    }
    zctx_destroy (&ctx);
    return NULL;
}

int main (void)
{
    //  创建线程
    zctx_t *ctx = zctx_new ();
    void *client = zthread_fork (ctx, client_task, NULL);
    zthread_new (ctx, worker_task, NULL);
    zthread_new (ctx, broker_task, NULL);

    //  等待client端管道的信号
    char *signal = zstr_recv (client);
    free (signal);

    zctx_destroy (&ctx);
    return 0;
}
```

在我的开发环境中运行结果如下：

```
Setting up test...
Synchronous round-trip test...
 9057 calls/second
Asynchronous round-trip test...
 173010 calls/second
```

需要注意的是client在运行开始会暂停一段时间，这是因为在向ROUTER套接字发送消息时，若指定标识的套接字没有连接，那么ROUTER会直接丢弃该消息。这个示例中我们没有使用LRU算法，所以当worker连接速度稍慢时就有可能丢失数据，影响测试结果。

我们可以看到，逐个处理循环比异步处理要慢将近20倍，让我们把它应用到管家模式中去。

首先，让我们修改client的API，添加独立的发送和接收方法：

```
mdcli_t *mdcli_new     (char *broker);
void     mdcli_destroy (mdcli_t **self_p);
```

```
int      mdcli_send    (mdcli_t *self, char *service, zmsg_t **request_p);
zmsg_t  *mdcli_recv    (mdcli_t *self);
```

然后花很短的时间就能将同步的client API改造成异步的API：

mdcliapi2: Majordomo asynchronous client API in C

```
/*  =====================================================================
    mdcliapi2.c

    Majordomo Protocol Client API (async version)
    Implements the MDP/Worker spec at http://rfc.zeromq.org/spec:7.

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it under
    the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation; either version 3 of the License, or (at
    your option) any later version.

    This software is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/

#include "mdcliapi2.h"

//  类结构
//  使用成员函数访问属性

struct _mdcli_t {
    zctx_t *ctx;                //  上下文
    char *broker;
    void *client;               //  连接至代理的套接字
    int verbose;                //  在标准输出打印运行状态
    int timeout;                //  请求超时时间
```

```
};


//  ---------------------------------------------------------------------
//  连接或重连代理

void s_mdcli_connect_to_broker (mdcli_t *self)
{
    if (self->client)
        zsocket_destroy (self->ctx, self->client);
    self->client = zsocket_new (self->ctx, ZMQ_DEALER);
    zmq_connect (self->client, self->broker);
    if (self->verbose)
        zclock_log ("I: 正在连接代理 %s...", self->broker);
}


//  ---------------------------------------------------------------------
//  构造函数

mdcli_t *
mdcli_new (char *broker, int verbose)
{
    assert (broker);

    mdcli_t *self = (mdcli_t *) zmalloc (sizeof (mdcli_t));
    self->ctx = zctx_new ();
    self->broker = strdup (broker);
    self->verbose = verbose;
    self->timeout = 2500;           //  毫秒

    s_mdcli_connect_to_broker (self);
    return self;
}


//  ---------------------------------------------------------------------
//  析构函数

void
mdcli_destroy (mdcli_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        mdcli_t *self = *self_p;
        zctx_destroy (&self->ctx);
```

```
        free (self->broker);
        free (self);
        *self_p = NULL;
    }
}


//  --------------------------------------------------------------------
//  设置请求超时时间

void
mdcli_set_timeout (mdcli_t *self, int timeout)
{
    assert (self);
    self->timeout = timeout;
}


//  --------------------------------------------------------------------
//  发送请求给代理
//  取得请求消息的所有权，发送后销毁

int
mdcli_send (mdcli_t *self, char *service, zmsg_t **request_p)
{
    assert (self);
    assert (request_p);
    zmsg_t *request = *request_p;

    //  在消息顶部加入协议规定的帧
    //  Frame 0: empty (模拟REQ套接字的行为)
    //  Frame 1: "MDPCxy" (6个字节, MDP/Client x.y)
    //  Frame 2: Service name (看打印字符串)
    zmsg_pushstr (request, service);
    zmsg_pushstr (request, MDPC_CLIENT);
    zmsg_pushstr (request, "");
    if (self->verbose) {
        zclock_log ("I: 发送请求给 '%s' 服务:", service);
        zmsg_dump (request);
    }
    zmsg_send (&request, self->client);
    return 0;
}


//  --------------------------------------------------------------------
```

```c
//  获取应答消息，若无则返回NULL；
//  该函数不会尝试从代理的崩溃中恢复，
//  因为我们没有记录那些未收到应答的请求，所以也无法重发。

zmsg_t *
mdcli_recv (mdcli_t *self)
{
    assert (self);

    //  轮询套接字以获取应答
    zmq_pollitem_t items [] = { { self->client, 0, ZMQ_POLLIN, 0 } };
    int rc = zmq_poll (items, 1, self->timeout * ZMQ_POLL_MSEC);
    if (rc == -1)
        return NULL;            //  中断

    //  收到应答后进行处理
    if (items [0].revents & ZMQ_POLLIN) {
        zmsg_t *msg = zmsg_recv (self->client);
        if (self->verbose) {
            zclock_log ("I: received reply:");
            zmsg_dump (msg);
        }
        //  不要处理错误，直接报出
        assert (zmsg_size (msg) >= 4);

        zframe_t *empty = zmsg_pop (msg);
        assert (zframe_streq (empty, ""));
        zframe_destroy (&empty);

        zframe_t *header = zmsg_pop (msg);
        assert (zframe_streq (header, MDPC_CLIENT));
        zframe_destroy (&header);

        zframe_t *service = zmsg_pop (msg);
        zframe_destroy (&service);

        return msg;     //  Success
    }
    if (zctx_interrupted)
        printf ("W: 收到中断消息，正在中止client...\n");
    else
    if (self->verbose)
        zclock_log ("W: 严重错误，放弃请求");

    return NULL;
}
```

下面是对应的测试代码：

mdclient2: Majordomo client application in C

```
//
//  异步管家模式 - client示例程序
//  使用mdcli API隐藏MDP协议的具体实现
//
//  直接编译源码，而不创建类库
#include "mdcliapi2.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    int count;
    for (count = 0; count < 100000; count++) {
        zmsg_t *request = zmsg_new ();
        zmsg_pushstr (request, "Hello world");
        mdcli_send (session, "echo", &request);
    }
    for (count = 0; count < 100000; count++) {
        zmsg_t *reply = mdcli_recv (session);
        if (reply)
            zmsg_destroy (&reply);
        else
            break;              //  使用Ctrl-C中断
    }
    printf ("收到 %d 个应答\n", count);
    mdcli_destroy (&session);
    return 0;
}
```

代理和worker的代码没有变，因为我们并没有改变MDP协议。经过对client的改造，我们可以明显看到速度的提升。如以下是同步状况下处理10万条请求的时间：

```
$ time mdclient
100000 requests/replies processed

real    0m14.088s
user    0m1.310s
sys     0m2.670s
```

以下是异步请求的情况：

```
$ time mdclient2
100000 replies received

real    0m8.730s
user    0m0.920s
sys     0m1.550s
```

让我们建立10个worker，看看效果如何：

```
$ time mdclient2
100000 replies received

real    0m3.863s
user    0m0.730s
sys     0m0.470s
```

由于worker获得消息需要通过LRU队列机制，所以并不能做到完全的异步。但是，worker越多其效果也会越好。在我的测试机上，当worker的数量达到8个时，速度就不再提升了——四核处理器只能做这么多。但是，我们仍然获得了近四倍的速度提升，而改造过程只有几分钟而已。此外，代理其实还没有进行优化，它仍会复制消息，而没有实现零拷贝。不过，我们已经做到每秒处理2.5万次请求-应答，已经很不错了。

当然，异步的管家模式也并不完美，有一个显著的缺点：它无法从代理的崩溃中恢复。可以看到mdcliapi2的代码中并没有恢复连接的代码，重新连接需要有以下几点作为前提：

- 每个请求都做了编号，每次应答也含有相应的编号，这就需要修改协议，明确定义；
- client的API需要保留并跟踪所有已发送、但仍未收到应答的请求；
- 如果代理发生崩溃，client会重发所有消息。

可以看到，高可靠性往往和复杂度成正比，值得在管家模式中应用这一机制吗？这就要看应用场景了。如果是一个名称查询服务，每次会话会调用一次，那不需要应用这一机制；如果是一个位于前端的网页服务，有数千个客户端相连，那可能就需要了。

服务查询

现在，我们已经有了一个面向服务的代理了，但是我们无法得知代理是否提供了某项特定服务。如果请求失败，那当然就表示该项服务目前不可用，但具体原因是什么呢？所以，如果能够询问代理“echo服务正在运行吗？”，那将会很有用处。最明显的方法是在MDP/Client协议中添加一种命令，客户端可以询问代理某项服务是否可用。但是，MDP/Client最大的优点在于简单，如果添加了服务查询的功能就太过复杂了。

另一种方案是学电子邮件的处理方式，将失败的请求重新返回。但是这同样会增加复杂度，因为我们需要鉴别收到的消息是一个应答还是被退回的请求。

让我们用之前的方式，在MDP的基础上建立新的机制，而不是改变它。服务定位本身也是一项服务，我们还可以提供类似于“禁用某服务”、“提供服务数据”等其他服务。我们需要的是一个能够扩展协议但又不会影响协议本身的机制。

这样就诞生了一个小巧的RFC - MMI（管家接口）的应用层，建立在MDP协议之上：
http://rfc.zeromq.org/spec:8 。我们在代理中其实已经加以实现了，不知你是否已经注意到。下面的代码演

示了如何使用这项服务查询功能：

mmiecho: Service discovery over Majordomo in C

```c
//
//  MMI echo 服务查询示例程序
//

//  让我们直接编译，不生成类库
#include "mdcliapi.c"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  我们需要查询的服务名称
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");

    //  发送给“服务查询”服务的消息
    zmsg_t *reply = mdcli_send (session, "mmi.service", &request);

    if (reply) {
        char *reply_code = zframe_strdup (zmsg_first (reply));
        printf ("查询 echo 服务的结果: %s\n", reply_code);
        free (reply_code);
        zmsg_destroy (&reply);
    }
    else
        printf ("E: 代理无响应，请确认它正在工作\n");

    mdcli_destroy (&session);
    return 0;
}
```

代理在运行时会检查请求的服务名称，自行处理那些mmi.开头的服务，而不转发给worker。你可以在不开启worker的情况下运行以上代码，可以看到程序是报告200还是404。MMI在示例程序代理中的实现是很简单的，比如，当某个worker消亡时，该服务仍然标记为可用。实践中，代理应该在一定间隔后清除那些没有worker的服务。

幂等服务

幂等是指能够安全地重复执行某项操作。如，看钟是幂等的，但借钱给别人老婆就不是了。有些客户端至服务端的通信是幂等的，但有些则不是。幂等的通信示例有：

- 无状态的任务分配，即管道模式中服务端是无状态的worker，它的处理结果是根据客户端的请求状态生成的，因此可以重复处理相同的请求；
- 命名服务中将逻辑地址转化成实际绑定或连接的端点，可以重复查询多次，因此也是幂等的。

非幂等的通信示例有：

- 日志服务，我们不会希望相同的日志内容被记录多次；
- 任何会对下游节点有影响的服务，如该服务会向下游节点发送信息，若收到相同的请求，那下游节点收到的信息就是重复的；
- 当服务修改了某些共享的数据，且没有进行幂等方面的设置。如某项服务对银行账户进行了借操作（debit），这一定是非幂等的。

如果应用程序提供的服务是非幂等的，那就需要考虑它究竟是在哪个阶段崩溃的。如果程序在空闲或处理请求的过程中崩溃，那不会有什么问题。我们可以使用数据库中的事务机制来保证借贷操作是同时发生的。如果应用程序在发送请求的时候崩溃了，那就会有问题，因为对于该程序来说，它已经完成了工作。

如果在返回应答的过程中网络阻塞了，客户端会认为请求发送失败，并进行重发，这样服务端会再一次执行相同的请求。这不是我们想要的结果。

常用的解决方法是在服务端检测并拒绝重复的请求，这就需要：

- 客户端为每个请求加注唯一的标识，包括客户端标识和消息标识；
- 服务端在发送应答时使用客户端标识和消息标识作为键，保存应答内容；
- 当服务端发现收到的请求已在应答哈希表中存在，它会跳过该次请求，直接返回应答内容。

脱机可靠性（巨人模式）

当你意识到管家模式是一种非常可靠的消息代理时，你可能会想要使用磁盘做一下消息中转，从而进一步提升可靠性。这种方式虽然在很多企业级消息系统中应用，但我还是有些反对的，原因有：

- 我们可以看到，懒惰海盗模式的client可以工作得非常好，能够在多种架构中运行。唯一的问题是它会假设worker是无状态的，且提供的服务是幂等的。但这个问题我们可以通过其他方式解决，而不是添加磁盘。

- 添加磁盘会带来新的问题，需要额外的管理和维护费用。海盗模式的最大优点就是简单明了，不会崩溃。如果你还是担心硬件会出问题，可以改用点对点的通信模式，这会在本章最后一节讲到。

虽然有以上原因，但还是有一个合理的场景可以用到磁盘中转的——异步脱机网络。海盗模式有一个问题，那就是client发送请求后会一直等待应答。如果client和worker并不是长连接（可以拿电子邮箱做个类比），我们就无法在client和worker之间建立一个无状态的网络，因此需要将这种状态保存起来。

于是我们就有了巨人模式，该模式下会将消息写到磁盘中，确保不会丢失。当我们进行服务查询时，会转向巨人这一层进行。巨人是建立在管家之上的，而不是改写了MDP协议。这样做的好处是我们可以在一个特定的worker中实现这种可靠性，而不用去增加代理的逻辑。

- 实现更为简单；
    - 代理用一种语言编写，worker使用另一种语言编写；
    - 可以自由升级这种模式。

唯一的缺点是，代理和磁盘之间会有一层额外的联系，不过这也是值得的。

我们有很多方法来实现一种持久化的请求-应答架构，而目标当然是越简单越好。我能想到的最简单的方式是提供一种成为“巨人”的代理服务，它不会影响现有worker的工作，若client想要立即得到应答，它可以和代理进行通信；如果它不是那么着急，那就可以和巨人通信：“嗨，巨人，麻烦帮我处理下这个请求，我去买些菜。”

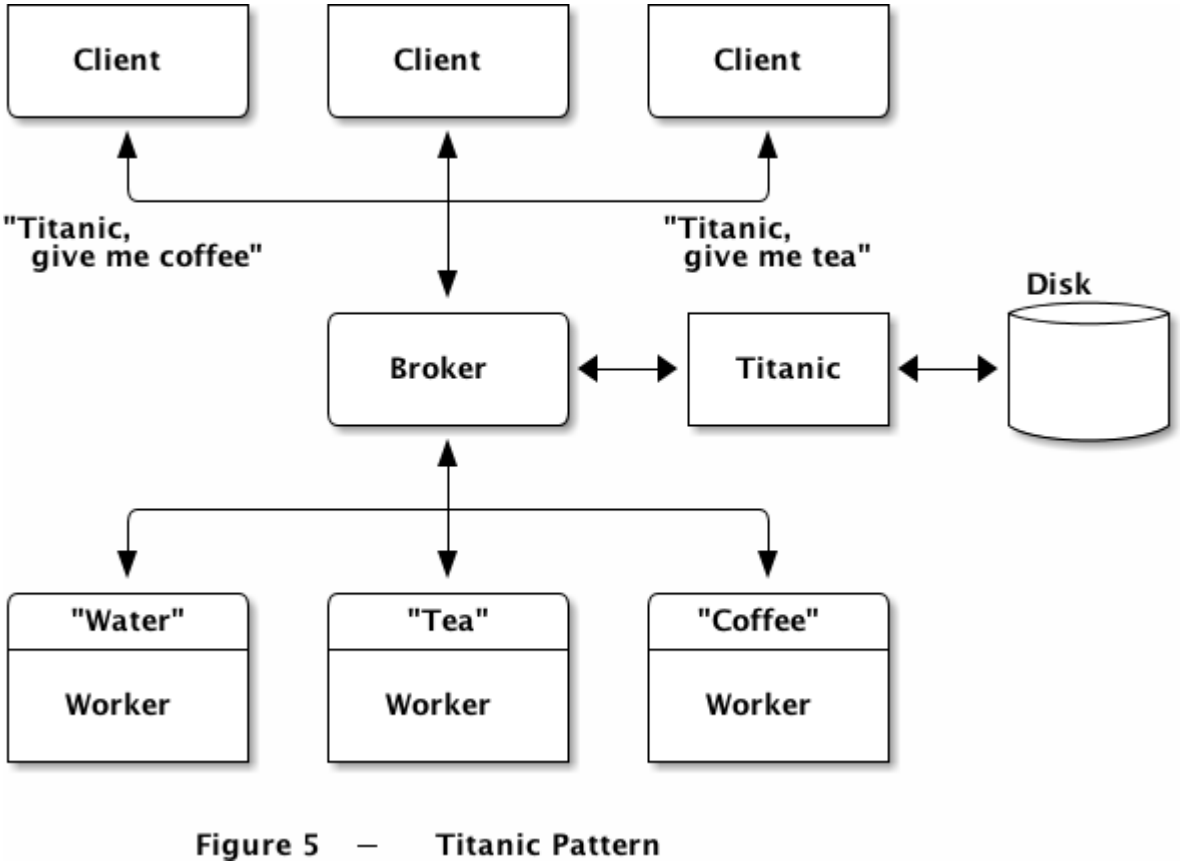

Figure 5 – Titanic Pattern

这样一来，巨人就既是worker又是client。client和巨人之间的对话一般是：

- Client: 请帮我处理这个请求。巨人：好的。
- Client: 有要给我的应答吗？巨人：有的。（或者没有）
- Client: OK，你可以释放那个请求了，工作已经完成。巨人：好的。

巨人和代理之间的对话一般是：

- 巨人：嗨，代理程序，你这里有个叫echo的服务吗？代理：恩，好像有。
- 巨人：嗨，echo服务，请帮我处理一下这个请求。Echo: 好了，这是应答。
- 巨人：谢谢！

你可以想象一些发生故障的情形，看看上述模式是否能解决？worker在处理请求的时候崩溃，巨人会不断地重新发送请求；应答在传输过程中丢失了，巨人也会重试；如果请求已经处理，但client没有得到应答，那它会再次询问巨人；如果巨人在处理请求或进行应答的时候崩溃了，客户端会进行重试；只要请求是被保存在磁盘上的，那它就不会丢失。

这个机制中，握手的过程是比较漫长的，但client可以使用异步的管家模式，一次发送多个请求，并一起等待应答。

我们需要一种方法，让client会去请求应答内容。不同的client会访问到相同的服务，且client是来去自由的，有着不同的标识。一个简单、合理、安全的解决方案是：

- 当巨人收到请求时，它会为每个请求生成唯一的编号（UUID），并将这个编号返回给client；
- client在请求应答内容时需要提供这个编号。

这样一来client就需要负责将UUID安全地保存起来，不过这就省去了验证的过程。有其他方案吗？我们可以使用持久化的套接字，即显式声明客户端的套接字标识。然而，这会造成管理上的麻烦，而且万一两个client的套接字标识相同，那会引来无穷的麻烦。

在我们开始制定一个新的协议之前，我们先思考一下client如何和巨人通信。一种方案是提供一种服务，配合三个不同的命令；另一种方案则更为简单，提供三种独立的服务：

- **titanic.request** - 保存一个请求，并返回UUID
- **titanic.reply** - 根据UUID获取应答内容
- **titanic.close** - 确认某个请求已被正确地处理

我们需要创建一个多线程的worker，正如我们之前用ZMQ进行多线程编程一样，很简单。但是，在我们开始编写代码之前，先讲巨人模式的一些定义写下来：http://rfc.zeromq.org/spec:9 。我们称之为“巨人服务协议”，或TSP。

使用TSP协议自然会让client多出额外的工作，下面是一个简单但足够健壮的client：

ticlient: Titanic client example in C

```
//
//  巨人模式client示例
//  实现 http://rfc.zeromq.org/spec:9 协议中的client端

//  让我们直接编译，不创建类库
#include "mdcliapi.c"

//  请求TSP协议下的服务
//  如果成功则返回应答（状态码：200），否则返回NULL
//
static zmsg_t *
s_service_call (mdcli_t *session, char *service, zmsg_t **request_p)
{
    zmsg_t *reply = mdcli_send (session, service, request_p);
    if (reply) {
        zframe_t *status = zmsg_pop (reply);
        if (zframe_streq (status, "200")) {
            zframe_destroy (&status);
            return reply;
        }
        else
        if (zframe_streq (status, "400")) {
            printf ("E: 客户端发生严重错误，取消请求\n");
```

```
            exit (EXIT_FAILURE);
        }
        else
        if (zframe_streq (status, "500")) {
            printf ("E: 服务端发生严重错误，取消请求\n");
            exit (EXIT_FAILURE);
        }
    }
    else
        exit (EXIT_SUCCESS);    //  中断或发生错误

    zmsg_destroy (&reply);
    return NULL;        //  请求不成功，但不返回失败原因
}

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    mdcli_t *session = mdcli_new ("tcp://localhost:5555", verbose);

    //  1. 发送echo服务的请求给巨人
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "echo");
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = s_service_call (
        session, "titanic.request", &request);

    zframe_t *uuid = NULL;
    if (reply) {
        uuid = zmsg_pop (reply);
        zmsg_destroy (&reply);
        zframe_print (uuid, "I: request UUID ");
    }

    //  2. 等待应答
    while (!zctx_interrupted) {
        zclock_sleep (100);
        request = zmsg_new ();
        zmsg_add (request, zframe_dup (uuid));
        zmsg_t *reply = s_service_call (
            session, "titanic.reply", &request);

        if (reply) {
            char *reply_string = zframe_strdup (zmsg_last (reply));
            printf ("Reply: %s\n", reply_string);
            free (reply_string);
```

```
            zmsg_destroy (&reply);

            //  3. 关闭请求
            request = zmsg_new ();
            zmsg_add (request, zframe_dup (uuid));
            reply = s_service_call (session, "titanic.close", &request);
            zmsg_destroy (&reply);
            break;
        }
        else {
            printf ("I: 尚未收到应答，准备稍后重试...\n");
            zclock_sleep (5000);      //  5秒后重试
        }
    }
    zframe_destroy (&uuid);
    mdcli_destroy (&session);
    return 0;
}
```

当然，上面的代码可以整合到一个框架中，程序员不需要了解其中的细节。如果我有时间的话，我会尝试写一个这样的API的，让应用程序又变回短短的几行。这种理念和MDP中的一致：不要做重复的事。

下面是巨人的实现。这个服务端会使用三个线程来处理三种服务。它使用最原始的持久化方法来保存请求：为每个请求创建一个磁盘文件。虽然简单，但也挺恐怖的。比较复杂的部分是，巨人会维护一个队列来保存这些请求，从而避免重复地扫描目录。

titanic: Titanic broker example in C

```
//
//  巨人模式 - 服务
//
//  实现 http://rfc.zeromq.org/spec:9 协议的服务端

//  让我们直接编译，不创建类库
#include "mdwrkapi.c"
#include "mdcliapi.c"

#include "zfile.h"
#include <uuid/uuid.h>

//  返回一个可打印的唯一编号（UUID）
//  调用者负责释放UUID字符串的内存

static char *
s_generate_uuid (void)
{
```

```
    char hex_char [] = "0123456789ABCDEF";
    char *uuidstr = zmalloc (sizeof (uuid_t) * 2 + 1);
    uuid_t uuid;
    uuid_generate (uuid);
    int byte_nbr;
    for (byte_nbr = 0; byte_nbr < sizeof (uuid_t); byte_nbr++) {
        uuidstr [byte_nbr * 2 + 0] = hex_char [uuid [byte_nbr] >> 4];
        uuidstr [byte_nbr * 2 + 1] = hex_char [uuid [byte_nbr] & 15];
    }
    return uuidstr;
}

//  根据UUID生成用于保存请求内容的文件名，并返回

#define TITANIC_DIR ".titanic"

static char *
s_request_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.req", uuid);
    return filename;
}

//  根据UUID生成用于保存应答内容的文件名，并返回

static char *
s_reply_filename (char *uuid) {
    char *filename = malloc (256);
    snprintf (filename, 256, TITANIC_DIR "/%s.rep", uuid);
    return filename;
}


//  ---------------------------------------------------------------------
//  巨人模式 - 请求服务

static void
titanic_request (void *args, zctx_t *ctx, void *pipe)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.request", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        //  若应答非空则发送，再从代理处获得新的请求
        zmsg_t *request = mdwrk_recv (worker, &reply);
```

```
        if (!request)
            break;      //  中断并退出

        //  确保消息目录是存在的
        file_mkdir (TITANIC_DIR);

        //  生成UUID，并将消息保存至磁盘
        char *uuid = s_generate_uuid ();
        char *filename = s_request_filename (uuid);
        FILE *file = fopen (filename, "w");
        assert (file);
        zmsg_save (request, file);
        fclose (file);
        free (filename);
        zmsg_destroy (&request);

        //  将UUID加入队列
        reply = zmsg_new ();
        zmsg_addstr (reply, uuid);
        zmsg_send (&reply, pipe);

        //  将UUID返回给客户端
        //  将由循环顶部的mdwrk_recv()函数完成
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
        zmsg_addstr (reply, uuid);
        free (uuid);
    }
    mdwrk_destroy (&worker);
}


//  ---------------------------------------------------------------------
//  巨人模式 - 应答服务

static void *
titanic_reply (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.reply", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断并退出
```

```
        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
        if (file_exists (rep_filename)) {
            FILE *file = fopen (rep_filename, "r");
            assert (file);
            reply = zmsg_load (file);
            zmsg_pushstr (reply, "200");
            fclose (file);
        }
        else {
            reply = zmsg_new ();
            if (file_exists (req_filename))
                zmsg_pushstr (reply, "300"); //挂起
            else
                zmsg_pushstr (reply, "400"); //未知
        }
        zmsg_destroy (&request);
        free (uuid);
        free (req_filename);
        free (rep_filename);
    }
    mdwrk_destroy (&worker);
    return 0;
}


//  --------------------------------------------------------------------
//  巨人模式 - 关闭请求

static void *
titanic_close (void *context)
{
    mdwrk_t *worker = mdwrk_new (
        "tcp://localhost:5555", "titanic.close", 0);
    zmsg_t *reply = NULL;

    while (TRUE) {
        zmsg_t *request = mdwrk_recv (worker, &reply);
        if (!request)
            break;      //  中断并退出

        char *uuid = zmsg_popstr (request);
        char *req_filename = s_request_filename (uuid);
        char *rep_filename = s_reply_filename (uuid);
```

```
        file_delete (req_filename);
        file_delete (rep_filename);
        free (uuid);
        free (req_filename);
        free (rep_filename);

        zmsg_destroy (&request);
        reply = zmsg_new ();
        zmsg_addstr (reply, "200");
    }
    mdwrk_destroy (&worker);
    return 0;
}

//  处理某个请求，成功则返回1

static int
s_service_success (mdcli_t *client, char *uuid)
{
    //  读取请求内容，第一帧为服务名称
    char *filename = s_request_filename (uuid);
    FILE *file = fopen (filename, "r");
    free (filename);

    //  如果client已经关闭了该请求，则返回1
    if (!file)
        return 1;

    zmsg_t *request = zmsg_load (file);
    fclose (file);
    zframe_t *service = zmsg_pop (request);
    char *service_name = zframe_strdup (service);

    //  使用MMI协议检查服务是否可用
    zmsg_t *mmi_request = zmsg_new ();
    zmsg_add (mmi_request, service);
    zmsg_t *mmi_reply = mdcli_send (client, "mmi.service", &mmi_request);
    int service_ok = (mmi_reply
        && zframe_streq (zmsg_first (mmi_reply), "200"));
    zmsg_destroy (&mmi_reply);

    if (service_ok) {
        zmsg_t *reply = mdcli_send (client, service_name, &request);
        if (reply) {
            filename = s_reply_filename (uuid);
            FILE *file = fopen (filename, "w");
```

```
            assert (file);
            zmsg_save (reply, file);
            fclose (file);
            free (filename);
            return 1;
        }
        zmsg_destroy (&reply);
    }
    else
        zmsg_destroy (&request);

    free (service_name);
    return 0;
}


int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));
    zctx_t *ctx = zctx_new ();

    //  创建MDP客户端会话
    mdcli_t *client = mdcli_new ("tcp://localhost:5555", verbose);
    mdcli_set_timeout (client, 1000);  //  1 秒
    mdcli_set_retries (client, 1);     //  只尝试一次

    void *request_pipe = zthread_fork (ctx, titanic_request, NULL);
    zthread_new (ctx, titanic_reply, NULL);
    zthread_new (ctx, titanic_close, NULL);

    //  主循环
    while (TRUE) {
        //  如果没有活动，我们将每秒循环一次
        zmq_pollitem_t items [] = { { request_pipe, 0, ZMQ_POLLIN, 0 } };
        int rc = zmq_poll (items, 1, 1000 * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  中断
        if (items [0].revents & ZMQ_POLLIN) {
            //  确保消息目录是存在的
            file_mkdir (TITANIC_DIR);

            //  将UUID添加到队列中，使用“-”号标识等待中的请求
            zmsg_t *msg = zmsg_recv (request_pipe);
            if (!msg)
                break;          //  中断
            FILE *file = fopen (TITANIC_DIR "/queue", "a");
```

```
            char *uuid = zmsg_popstr (msg);
            fprintf (file, "-%s\n", uuid);
            fclose (file);
            free (uuid);
            zmsg_destroy (&msg);
        }
        //  分派
        //
        char entry [] = "?.....:.....:.....:.....:";
        FILE *file = fopen (TITANIC_DIR "/queue", "r+");
        while (file && fread (entry, 33, 1, file) == 1) {
            //  处理UUID前缀为“-”的请求
            if (entry [0] == '-') {
                if (verbose)
                    printf ("I: 开始处理请求 %s\n", entry + 1);
                if (s_service_success (client, entry + 1)) {
                    //  标记为已处理
                    fseek (file, -33, SEEK_CUR);
                    fwrite ("+", 1, 1, file);
                    fseek (file, 32, SEEK_CUR);
                }
            }
            //  跳过最后一行
            if (fgetc (file) == '\r')
                fgetc (file);
            if (zctx_interrupted)
                break;
        }
        if (file)
            fclose (file);
    }
    mdcli_destroy (&client);
    return 0;
}
```

测试时，打开mdbroker和titanic，再运行ticlient，然后开启任意个mdworker，就可以看到client获得了应答。

几点说明：

- 我们使用MMI协议去向代理询问某项服务是否可用，这一点和MDP中的逻辑一致；
- 我们使用inproc（进程内）协议建立主循环和titanic.request服务间的联系，保存新的请求信息。这样可以避免主循环不断扫描磁盘目录，读取所有请求文件，并按照时间日期排序。

这个示例程序不应关注它的性能（一定会非常糟糕，虽然我没有测试过），而是应该看到它是如何提供一种可靠的通信模式的。你可以测试一下，打开代理、巨人、worker和client，使用-v参数显示跟踪信息，然后随意地开关代理、巨人、或worker（client不能关闭），可以看到所有的请求都能获得应答。

如果你想在真实环境中使用巨人模式，你肯定会问怎样才能让速度快起来。以下是我的做法：

- 使用一个磁盘文件保存所有数据。操作系统处理大文件的效率要比处理许多小文件来的高。
- 使用一种循环的机制来组织该磁盘文件的结构，这样新的请求可以被连续地写入这个文件。单个线程在全速写入磁盘时的效率是比较高的。
- 将索引保存在内存中，可以在启动程序时重建这个索引。这样做可以节省磁盘缓存，让索引安全地保存在磁盘上。你需要用到fsync的机制来保存每一条数据；或者可以等待几毫秒，如果不怕丢失上千条数据的话。
- 如果条件允许，应选择使用固态硬盘；
- 提前分配该磁盘文件的空间，或者将每次分配的空间调大一些，这样可以避免磁盘碎片的产生，并保证读写是连续的。

另外，我不建议将消息保存在数据库中，甚至不建议交给那些所谓的高速键值缓存，它们比起一个磁盘文件要来得昂贵。

如果你想让巨人模式变得更为可靠，你可以将请求复制到另一台服务器上，这样就不需要担心主程序遭到核武器袭击了。

如果你想让巨人模式变得更为快速，但可以牺牲一些可靠性，那你可以将请求和应答都保存在内存中。这样做可以让该服务作为脱机网络运行，不过若巨人服务本身崩溃了，我也无能为力。

高可靠对称节点（双子星模式）

概览

双子星模式是一对具有主从机制的高可靠节点。任一时间，某个节点会充当主机，接收所有客户端的请求；另一个则作为一种备机存在。两个节点会互相监控对方，当主机从网络中消失时，备机会替代主机的位置。

双子星模式由Pieter Hintjens和Martin Sustrik设计，应用在iMatix的OpenAMQ服务器中。它的设计理念是：

- 提供一种简明的高可靠性解决方案；
- 易于理解和使用；
- 能够进行可靠的故障切换。

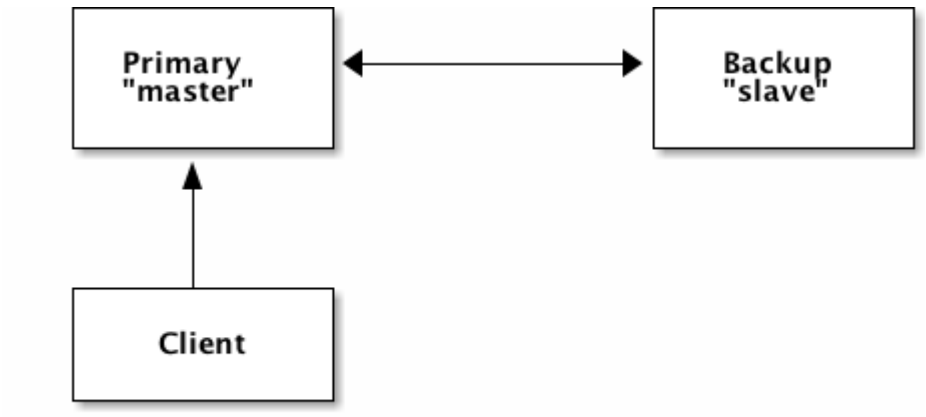

Figure 6 — —High availability pair, normal operation

假设我们有一组双子星模式的服务器，以下是可能发生的故障：

1. 主机发生硬件故障（断电、失火等），应用程序发送后立刻使用备机进行连接；
2. 主机的网络环境发生故障，可能某个路由器被雷击了，立刻使用备机；
3. 主机上的服务被维护人员误杀，无法自动恢复。

恢复步骤如下：

1. 维护人员排查主机故障；
2. 将备机关闭，造成短时间的服务不可用；
3. 待应用程序都连接到主机后，维护人员重启备机。

恢复过程是人工进行的，惨痛的经验告诉我们自动恢复是很可怕的：

- 故障的发生会造成10-30秒之间的服务暂停，如果这是一个真正的突发状况，那最好还是让主机暂停服务的好，因为立刻重启服务可能造成另一个10-30秒的暂停，不如让用户停止使用。
- 当有紧急状况发生时，可以在修复的过程中记录故障发生原因，而不是让系统自动恢复，管理员因此无法用其经验抵御下一次突发状况。
- 最后，如果自动恢复确实成功了，管理员将无从得知故障的发生原因，因而无法进行分析。

双子星模式的故障恢复过程是：在修复了主机的问题后，将备机做关闭处理，稍后再重新开启：

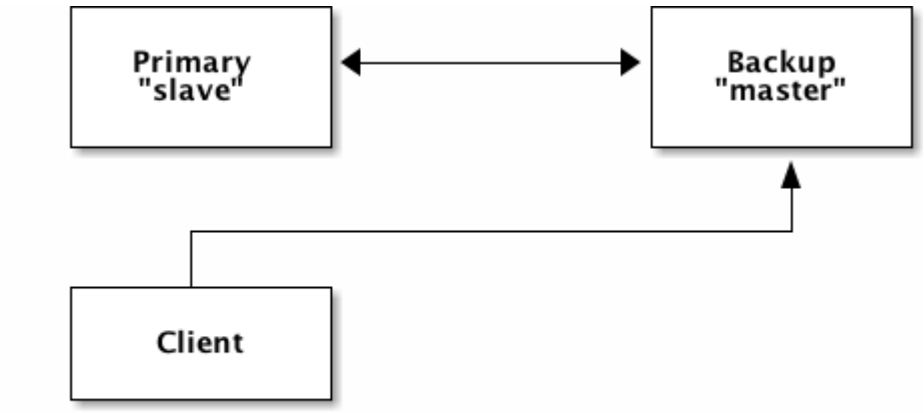

Figure 7 — —High availability pair, during failover

双子星模式的关闭过程有两种：

1. 先关闭备机，等待一段时间后再关闭主机；
2. 同时关闭主机和备机，间隔时间不超过几秒。

关闭时，间隔时间要比故障切换时间短，否则会导致应用程序失去连接、重新连接、并再次失去连接，导致用户投诉。

详细要求

双子星模式可以非常简单，但能工作得很出色。事实上，这里的实现方法已经历经三个版本了，之前的版本都过于复杂，想要做太多的事情，因而被我们抛弃。我们需要的只是最基本的功能，能够提供易理解、易开发、

高可靠的解决方法就可以了。

以下是该架构的详细需求：

- 需要用到双子星模式的故障是：系统遭受灾难性的打击，如硬件崩溃、火灾、意外等。对于其他常规的服务器故障，可以用更简单的方法。
- 故障恢复时间应该在60秒以内，理想情况下应该在10秒以内；
- 故障恢复(failover)应该是自动完成的，而系统还原(recover)则是由人工完成的。我们希望应用程序能够在发生故障时自动从主机切换到备机，但不希望在问题解决之前自动切换回主机，因为这很有可能让主机再次崩溃。
- 程序的逻辑应该尽量简单，易于使用，最好能封装在API中；
- 需要提供一个明确的指示，哪台主机正在提供服务，以避免“精神分裂”的症状，即两台服务器都认为自己是主机；
- 两台服务器的启动顺序不应该有限制；
- 启动或关闭主从机时不需要更改客户端的配置，但有可能会中断连接；
- 管理员需要能够同时监控两台机器；
- 两台机器之间必须有专用的高速网络连接，必须能使用特定IP进行路由。

我们做如下架假设：

- 单台备机能够提供足够的保障，不需要再进行其他备份机制；
- 主从机应该都能够提供完整的服务，承载相同的压力，不需要进行负载均衡；
- 预算中允许有这样一台长时间闲置的备机。

双子星模式不会用到：

- 多台备机，或在主从机之间进行负载均衡。该模式中的备机将一直处于空闲状态，只有主机发生问题时才会工作；
- 处理持久化的消息或事务。我们假设所连接的网络是不可靠的（或不可信的）。
- 自动搜索网络。双子星模式是手工配置的，他们知道对方的存在，应用程序则知道双子星的存在。
- 主从机之间状态的同步。所有服务端的状态必须能由应用程序进行重建。

以下是双子星模式中的几个术语：

- **主机** - 通常情况下作为master的机器；
- **备机** - 通常情况下作为slave的机器，只有当主机从网络中消失时，备机才会切换成master状态，接收所有的应用程序请求；
- **master** - 双子星模式中接收应用程序请求的机器；同一时刻只有一台master；
- **slave** - 当master消失时用以顶替的机器。

配置双子星模式的步骤：

1. 让主机知道备机的位置；
2. 让备机知道主机的位置；
3. 调整故障恢复时间，两台机器的配置必须相同。

比较重要的配置是应让两台机器间隔多久检查一次对方的状态，以及多长时间后采取行动。在我们的示例中，故障恢复时间设置为2000毫秒，超过这个时间备机就会代替主机的位置。但若你将主机的服务包裹在一个shell脚本中进行重启，就需要延长这个时间，否则备机可能在主机恢复连接的过程中转换成master。

要让客户端应用程序和双子星模式配合，你需要做的是：

1. 知道两台服务器的地址；
2. 尝试连接主机，若失败则连接备机；
3. 检测失效的连接，一般使用心跳机制；
4. 尝试重连主机，然后再连接备机，其间的间隔应比服务器故障恢复时间长；
5. 重建服务器端需要的所有状态数据；
6. 如果要保证可靠性，应重发故障期间的消息。

这不是件容易的事，所以我们一般会将其封装成一个API，供程序员使用。

双子星模式的主要限制有：

- 服务端进程不能涉及到一个以上的双子星对称节点；
- 主机只能有一个备机；
- 当备机处于slave状态时，它不会处理任何请求；
- 备机必须能够承受所有的应用程序请求；
- 故障恢复时间不能在运行时调整；
- 客户端应用程序需要做一些重连的工作。

防止精神分裂

“精神分裂”症状指的是一个集群中的不同部分同时认为自己是master，从而停止对对方的检测。双子星模式中的算法会降低这种症状的发生几率：主备机在决定自己是否为master时会检测自身是否收到了应用程序的请求，以及对方是否已经从网络中消失。

但在某些情况下，双子星模式也会发生精神分裂。比如说，主备机被配置在两幢大楼里，每幢大楼的局域网中又分布了一些应用程序。这样，当两幢大楼的网络通信被阻断，双子星模式的主备机就会分别在两幢大楼里接受和处理请求。

为了防止精神分裂，我们必须让主备机使用专用的网络进行连接，最简单的方法当然是用一根双绞线将他们相连。

我们不能将双子星部署在两个不同的岛屿上，为各自岛屿的应用程序服务。这种情况下，我们会使用诸如联邦模式的机制进行可靠性设计。

最好但最夸张的做法是，将两台机器之间的连接和应用程序的连接完全隔离开来，甚至是使用不同的网卡，而不仅仅是不同的端口。这样做也是为了日后排查错误时更为明确。

实现双子星模式

闲话少说，下面是双子星模式的服务端代码：

bstarsrv: Binary Star server in C

```
//
//  双子星模式 - 服务端
//
#include "czmq.h"
```

```
//  发送状态信息的间隔时间
//  如果对方在两次心跳过后都没有应答，则视为断开
#define HEARTBEAT 1000          //  In msecs

//  服务器状态枚举
typedef enum {
    STATE_PRIMARY = 1,          //  主机，等待同伴连接
    STATE_BACKUP = 2,           //  备机，等待同伴连接
    STATE_ACTIVE = 3,           //  激活态，处理应用程序请求
    STATE_PASSIVE = 4           //  被动态，不接收请求
} state_t;

//  对话节点事件
typedef enum {
    PEER_PRIMARY = 1,           //  主机
    PEER_BACKUP = 2,            //  备机
    PEER_ACTIVE = 3,            //  激活态
    PEER_PASSIVE = 4,           //  被动态
    CLIENT_REQUEST = 5          //  客户端请求
} event_t;

//  有限状态机
typedef struct {
    state_t state;              //  当前状态
    event_t event;              //  当前事件
    int64_t peer_expiry;        //  判定节点死亡的时限
} bstar_t;


//  执行有限状态机（将事件绑定至状态）；
//  发生异常时返回TRUE。

static Bool
s_state_machine (bstar_t *fsm)
{
    Bool exception = FALSE;
    //  主机等待同伴连接
    //  该状态下接收CLIENT_REQUEST事件
    if (fsm->state == STATE_PRIMARY) {
        if (fsm->event == PEER_BACKUP) {
            printf ("I: 已连接至备机（slave），可以作为master运行。\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_ACTIVE) {
```

```
            printf ("I: 已连接至备机（master），可以作为slave运行。\n");
            fsm->state = STATE_PASSIVE;
        }
    }
    else
    //  备机等待同伴连接
    //  该状态下拒绝CLIENT_REQUEST事件
    if (fsm->state == STATE_BACKUP) {
        if (fsm->event == PEER_ACTIVE) {
            printf ("I: 已连接至主机（master），可以作为slave运行。\n");
            fsm->state = STATE_PASSIVE;
        }
        else
        if (fsm->event == CLIENT_REQUEST)
            exception = TRUE;
    }
    else
    //  服务器处于激活态
    //  该状态下接受CLIENT_REQUEST事件
    if (fsm->state == STATE_ACTIVE) {
        if (fsm->event == PEER_ACTIVE) {
            //  若出现两台master，则抛出异常
            printf ("E: 严重错误：双master。正在退出。\n");
            exception = TRUE;
        }
    }
    else
    //  服务器处于被动态
    //  若同伴已死，CLIENT_REQUEST事件将触发故障恢复
    if (fsm->state == STATE_PASSIVE) {
        if (fsm->event == PEER_PRIMARY) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            printf ("I: 主机（slave）正在重启，可作为master运行。\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_BACKUP) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            printf ("I: 备机（slave）正在重启，可作为master运行。\n");
            fsm->state = STATE_ACTIVE;
        }
        else
        if (fsm->event == PEER_PASSIVE) {
            //  若出现两台slave，集群将无响应
            printf ("E: 严重错误：双slave。正在退出\n");
            exception = TRUE;
```

```
        }
        else
        if (fsm->event == CLIENT_REQUEST) {
            //  若心跳超时，同伴将成为master；
            //  此行为由客户端请求触发。
            assert (fsm->peer_expiry > 0);
            if (zclock_time () >= fsm->peer_expiry) {
                //  同伴已死，转为激活态。
                printf ("I: 故障恢复，可作为master运行。\n");
                fsm->state = STATE_ACTIVE;
            }
            else
                //  同伴还在，拒绝请求。
                exception = TRUE;
        }
    }
    return exception;
}


int main (int argc, char *argv [])
{
    //  命令行参数可以为：
    //      -p  作为主机启动，at tcp://localhost:5001
    //      -b  作为备机启动，at tcp://localhost:5002
    zctx_t *ctx = zctx_new ();
    void *statepub = zsocket_new (ctx, ZMQ_PUB);
    void *statesub = zsocket_new (ctx, ZMQ_SUB);
    void *frontend = zsocket_new (ctx, ZMQ_ROUTER);
    bstar_t fsm = { 0 };

    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: 主机master，等待备机（slave）连接。\n");
        zsocket_bind (frontend, "tcp://*:5001");
        zsocket_bind (statepub, "tcp://*:5003");
        zsocket_connect (statesub, "tcp://localhost:5004");
        fsm.state = STATE_PRIMARY;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: 备机slave，等待主机（master）连接。\n");
        zsocket_bind (frontend, "tcp://*:5002");
        zsocket_bind (statepub, "tcp://*:5004");
        zsocket_connect (statesub, "tcp://localhost:5003");
        fsm.state = STATE_BACKUP;
    }
```

```
    else {
        printf ("Usage: bstarsrv { -p | -b }\n");
        zctx_destroy (&ctx);
        exit (0);
    }
    //  设定下一次发送状态的时间
    int64_t send_state_at = zclock_time () + HEARTBEAT;

    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = {
            { frontend, 0, ZMQ_POLLIN, 0 },
            { statesub, 0, ZMQ_POLLIN, 0 }
        };
        int time_left = (int) ((send_state_at - zclock_time ()));
        if (time_left < 0)
            time_left = 0;
        int rc = zmq_poll (items, 2, time_left * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文对象被关闭

        if (items [0].revents & ZMQ_POLLIN) {
            //  收到客户端请求
            zmsg_t *msg = zmsg_recv (frontend);
            fsm.event = CLIENT_REQUEST;
            if (s_state_machine (&fsm) == FALSE)
                //  返回应答
                zmsg_send (&msg, frontend);
            else
                zmsg_destroy (&msg);
        }
        if (items [1].revents & ZMQ_POLLIN) {
            //  收到状态消息，作为事件处理
            char *message = zstr_recv (statesub);
            fsm.event = atoi (message);
            free (message);
            if (s_state_machine (&fsm))
                break;          //  错误，退出。
            fsm.peer_expiry = zclock_time () + 2 * HEARTBEAT;
        }
        //  定时发送状态信息
        if (zclock_time () >= send_state_at) {
            char message [2];
            sprintf (message, "%d", fsm.state);
            zstr_send (statepub, message);
            send_state_at = zclock_time () + HEARTBEAT;
        }
```

```
    }
    if (zctx_interrupted)
        printf ("W: 中断\n");

    //  关闭套接字和上下文
    zctx_destroy (&ctx);
    return 0;
}
```

下面是客户端代码：

bstarcli: Binary Star client in C

```
//
//  双子星模式 - 客户端
//
#include "czmq.h"

#define REQUEST_TIMEOUT     1000    //  毫秒
#define SETTLE_DELAY        2000    //  超时时间

int main (void)
{
    zctx_t *ctx = zctx_new ();

    char *server [] = { "tcp://localhost:5001", "tcp://localhost:5002" };
    uint server_nbr = 0;

    printf ("I: 正在连接服务器 %s...\n", server [server_nbr]);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, server [server_nbr]);

    int sequence = 0;
    while (!zctx_interrupted) {
        //  发送请求并等待应答
        char request [10];
        sprintf (request, "%d", ++sequence);
        zstr_send (client, request);

        int expect_reply = 1;
        while (expect_reply) {
            //  轮询套接字
            zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
            int rc = zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
            if (rc == -1)
                break;          //  中断
```

```
            //  处理应答
            if (items [0].revents & ZMQ_POLLIN) {
                //  审核应答编号
                char *reply = zstr_recv (client);
                if (atoi (reply) == sequence) {
                    printf ("I: 服务端应答正常 (%s)\n", reply);
                    expect_reply = 0;
                    sleep (1);  //  每秒发送一个请求
                }
                else {
                    printf ("E: 错误的应答内容: %s\n",
                        reply);
                }
                free (reply);
            }
            else {
                printf ("W: 服务器无响应，正在重试\n");
                //  重开套接字
                zsocket_destroy (ctx, client);
                server_nbr = (server_nbr + 1) % 2;
                zclock_sleep (SETTLE_DELAY);
                printf ("I: 正在连接服务端 %s...\n",
                        server [server_nbr]);
                client = zsocket_new (ctx, ZMQ_REQ);
                zsocket_connect (client, server [server_nbr]);

                //  使用新套接字重发请求
                zstr_send (client, request);
            }
        }
    }
    zctx_destroy (&ctx);
    return 0;
}
```

运行以下命令进行测试，顺序随意：

```
bstarsrv -p     # Start primary
bstarsrv -b     # Start backup
bstarcli
```

可以将主机进程杀掉，测试故障恢复机制；再开启主机，杀掉备机，查看还原机制。要注意是由客户端触发这两个事件的。

下图展现了服务进程的状态图。绿色状态下会接收客户端请求，粉色状态会拒绝请求。事件指的是同伴的状态，所以“同伴激活态”指的是同伴机器告知我们它处于激活态。“客户请求”表示我们从客户端获得了请求，“客户投票”则指我们从客户端获得了请求并且同伴已经超时死亡。

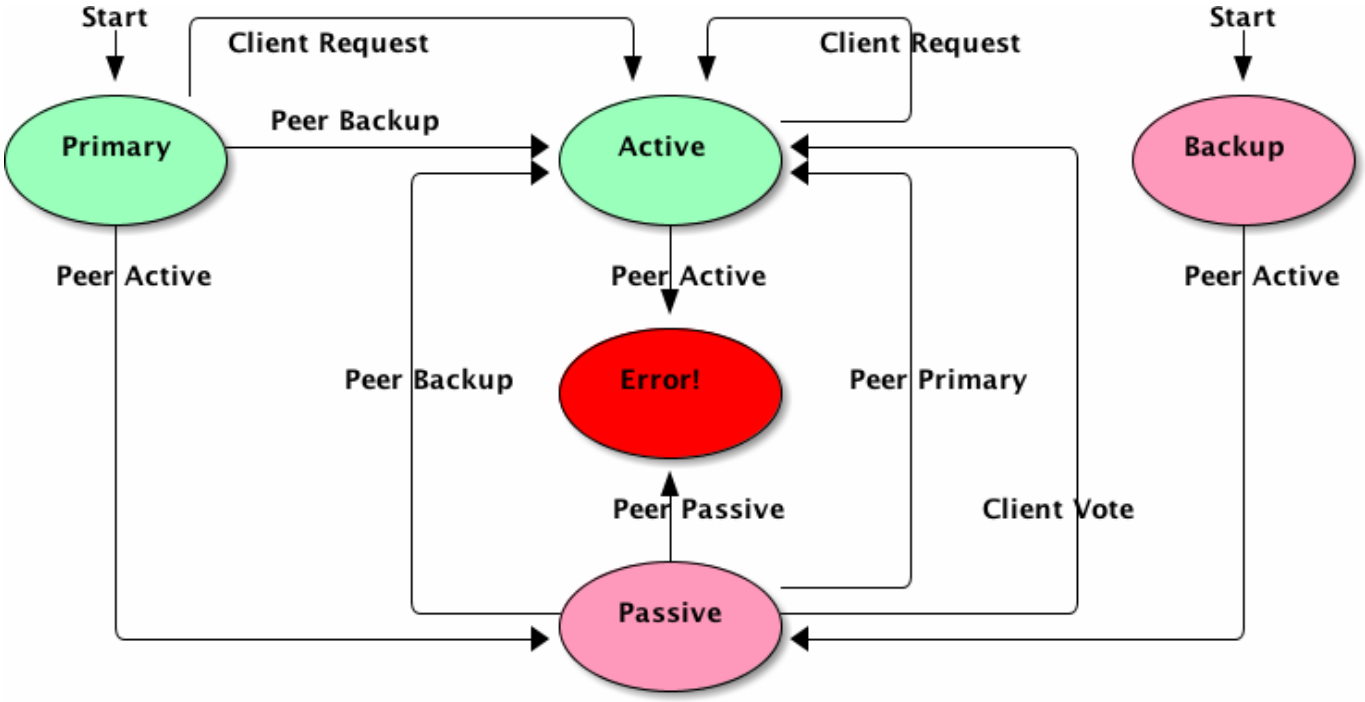

Figure 8 — Binary Star finite state machine

需要注意的是，服务进程使用PUB-SUB套接字进行状态交换，其它类型的套接字在这里不适用。比如，PUSH和DEALER套接字在没有节点相连的时候会发生阻塞；PAIR套接字不会在节点断开后进行重连；ROUTER套接字需要地址才能发送消息。

These are the main limitations of the Binary Star pattern:

- A server process cannot be part of more than one Binary Star pair.
- A primary server can have a single backup server, no more.
- The backup server cannot do useful work while in slave mode.
- The backup server must be capable of handling full application loads.
- Failover configuration cannot be modified at runtime.
- Client applications must do some work to benefit from failover.

双子星反应堆

我们可以将双子星模式打包成一个类似反应堆的类，供以后复用。在C语言中，我们使用czmq的zloop类，其他语言应该会有相应的实现。以下是C语言版的bstar接口：

```
// 创建双子星模式实例，使用本地（绑定）和远程（连接）端点来设置节点对。
bstar_t *bstar_new (int primary, char *local, char *remote);

// 销毁实例
void bstar_destroy (bstar_t **self_p);
```

```
// 返回底层的zloop反应堆，用以添加定时器、读取器、注册和取消等功能。
zloop_t *bstar_zloop (bstar_t *self);

// 注册投票读取器
int bstar_voter (bstar_t *self, char *endpoint, int type,
zloop_fn handler, void *arg);

// 注册状态机处理器
void bstar_new_master (bstar_t *self, zloop_fn handler, void *arg);
void bstar_new_slave (bstar_t *self, zloop_fn handler, void *arg);

// 开启反应堆，当回调函数返回-1，或进程收到SIGINT、SIGTERM信号时中止。
int bstar_start (bstar_t *self);
```

以下是类的实现：

bstar: Binary Star core class in C

```
/*  =====================================================================
    bstar - Binary Star reactor

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it under
    the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation; either version 3 of the License, or (at
    your option) any later version.

    This software is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =====================================================================
*/

#include "bstar.h"
```

```
//  服务器状态枚举
typedef enum {
    STATE_PRIMARY = 1,          //  主机，等待同伴连接
    STATE_BACKUP = 2,           //  备机，等待同伴连接
    STATE_ACTIVE = 3,           //  激活态，处理应用程序请求
    STATE_PASSIVE = 4           //  被动态，不接收请求
} state_t;

//  对话节点事件
typedef enum {
    PEER_PRIMARY = 1,           //  主机
    PEER_BACKUP = 2,            //  备机
    PEER_ACTIVE = 3,            //  激活态
    PEER_PASSIVE = 4,           //  被动态
    CLIENT_REQUEST = 5          //  客户端请求
} event_t;


//  发送状态信息的间隔时间
//  如果对方在两次心跳过后都没有应答，则视为断开
#define BSTAR_HEARTBEAT     1000        //  In msecs

//  类结构

struct _bstar_t {
    zctx_t *ctx;                //  私有上下文
    zloop_t *loop;              //  反应堆循环
    void *statepub;             //  状态发布者
    void *statesub;             //  状态订阅者
    state_t state;              //  当前状态
    event_t event;              //  当前事件
    int64_t peer_expiry;        //  判定节点死亡的时限
    zloop_fn *voter_fn;         //  投票套接字处理器
    void *voter_arg;            //  投票处理程序的参数
    zloop_fn *master_fn;        //  成为master时回调
    void *master_arg;           //  参数
    zloop_fn *slave_fn;         //  成为slave时回调
    void *slave_arg;            //  参数
};


//  ---------------------------------------------------------------------
//  执行有限状态机（将事件绑定至状态）；
//  发生异常时返回-1，正确时返回0。

static int
```

```
s_execute_fsm (bstar_t *self)
{
    int rc = 0;
    //  主机等待同伴连接
    //  该状态下接收CLIENT_REQUEST事件
    if (self->state == STATE_PRIMARY) {
        if (self->event == PEER_BACKUP) {
            zclock_log ("I: 已连接至备机（slave），可以作为master运行。");
            self->state = STATE_ACTIVE;
            if (self->master_fn)
                (self->master_fn) (self->loop, NULL, self->master_arg);
        }
        else
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: 已连接至备机（master），可以作为slave运行。");
            self->state = STATE_PASSIVE;
            if (self->slave_fn)
                (self->slave_fn) (self->loop, NULL, self->slave_arg);
        }
        else
        if (self->event == CLIENT_REQUEST) {
            zclock_log ("I: 收到客户端请求，可作为master运行。");
            self->state = STATE_ACTIVE;
            if (self->master_fn)
                (self->master_fn) (self->loop, NULL, self->master_arg);
        }
    }
    else
    //  备机等待同伴连接
    //  该状态下拒绝CLIENT_REQUEST事件
    if (self->state == STATE_BACKUP) {
        if (self->event == PEER_ACTIVE) {
            zclock_log ("I: 已连接至主机（master），可以作为slave运行。");
            self->state = STATE_PASSIVE;
            if (self->slave_fn)
                (self->slave_fn) (self->loop, NULL, self->slave_arg);
        }
        else
        if (self->event == CLIENT_REQUEST)
            rc = -1;
    }
    else
    //  服务器处于激活态
    //  该状态下接受CLIENT_REQUEST事件
    //  只有服务器死亡才会离开激活态
    if (self->state == STATE_ACTIVE) {
```

```
        if (self->event == PEER_ACTIVE) {
            //  若出现两台master，则抛出异常
            zclock_log ("E: 严重错误：双master。正在退出。");
            rc = -1;
        }
    }
    else
    //  服务器处于被动态
    //  若同伴已死，CLIENT_REQUEST事件将触发故障恢复
    if (self->state == STATE_PASSIVE) {
        if (self->event == PEER_PRIMARY) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            zclock_log ("I: 主机（slave）正在重启，可作为master运行。");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_BACKUP) {
            //  同伴正在重启 - 转为激活态，同伴将转为被动态。
            zclock_log ("I: 备机（slave）正在重启，可作为master运行。");
            self->state = STATE_ACTIVE;
        }
        else
        if (self->event == PEER_PASSIVE) {
            //  若出现两台slave，集群将无响应
            zclock_log ("E: 严重错误：双slave。正在退出");
            rc = -1;
        }
        else
        if (self->event == CLIENT_REQUEST) {
            //  若心跳超时，同伴将成为master；
            //  此行为由客户端请求触发。
            assert (self->peer_expiry > 0);
            if (zclock_time () >= self->peer_expiry) {
                //  同伴已死，转为激活态。
                zclock_log ("I: 故障恢复，可作为master运行。");
                self->state = STATE_ACTIVE;
            }
            else
                //  同伴还在，拒绝请求。
                rc = -1;
        }
        //  触发状态更改事件处理函数
        if (self->state == STATE_ACTIVE && self->master_fn)
            (self->master_fn) (self->loop, NULL, self->master_arg);
    }
    return rc;
```

```c
}


//  --------------------------------------------------------------------
//  反应堆事件处理程序

//  发送状态信息
int s_send_state (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    zstr_sendf (self->statepub, "%d", self->state);
    return 0;
}

//  接收状态信息，启动有限状态机
int s_recv_state (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    char *state = zstr_recv (socket);
    if (state) {
        self->event = atoi (state);
        self->peer_expiry = zclock_time () + 2 * BSTAR_HEARTBEAT;
        free (state);
    }
    return s_execute_fsm (self);
}

//  收到应用程序请求，判断是否接收
int s_voter_ready (zloop_t *loop, void *socket, void *arg)
{
    bstar_t *self = (bstar_t *) arg;
    //  如果能够处理请求，则调用函数
    self->event = CLIENT_REQUEST;
    if (s_execute_fsm (self) == 0) {
        puts ("CLIENT REQUEST");
        (self->voter_fn) (self->loop, socket, self->voter_arg);
    }
    else {
        //  销毁等待中的消息
        zmsg_t *msg = zmsg_recv (socket);
        zmsg_destroy (&msg);
    }
    return 0;
}
```

```
//  --------------------------------------------------------------------------
//  构造函数

bstar_t *
bstar_new (int primary, char *local, char *remote)
{
    bstar_t
        *self;

    self = (bstar_t *) zmalloc (sizeof (bstar_t));

    //  初始化双子星
    self->ctx = zctx_new ();
    self->loop = zloop_new ();
    self->state = primary? STATE_PRIMARY: STATE_BACKUP;

    //  创建状态PUB套接字
    self->statepub = zsocket_new (self->ctx, ZMQ_PUB);
    zsocket_bind (self->statepub, local);

    //  创建状态SUB套接字
    self->statesub = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_connect (self->statesub, remote);

    //  设置基本的反应堆事件处理器
    zloop_timer (self->loop, BSTAR_HEARTBEAT, 0, s_send_state, self);
    zloop_reader (self->loop, self->statesub, s_recv_state, self);
    return self;
}


//  --------------------------------------------------------------------------
//  析构函数

void
bstar_destroy (bstar_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        bstar_t *self = *self_p;
        zloop_destroy (&self->loop);
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}
```

```
//  --------------------------------------------------------------------------
//  返回底层zloop对象，用以添加额外的定时器、阅读器等。

zloop_t *
bstar_zloop (bstar_t *self)
{
    return self->loop;
}


//  --------------------------------------------------------------------------
//  创建套接字，连接至本地端点，注册成为阅读器；
//  只有当有限状态机允许时才会读取该套接字；
//  从该套接字获得的消息将作为一次“投票”；
//  我们要求双子星模式中只有一个“投票”套接字。

int
bstar_voter (bstar_t *self, char *endpoint, int type, zloop_fn handler,
             void *arg)
{
    //  保存原始的回调函数和参数，稍后使用
    void *socket = zsocket_new (self->ctx, type);
    zsocket_bind (socket, endpoint);
    assert (!self->voter_fn);
    self->voter_fn = handler;
    self->voter_arg = arg;
    return zloop_reader (self->loop, socket, s_voter_ready, self);
}

//  --------------------------------------------------------------------------
//  注册状态变化事件处理器

void
bstar_new_master (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->master_fn);
    self->master_fn = handler;
    self->master_arg = arg;
}

void
bstar_new_slave (bstar_t *self, zloop_fn handler, void *arg)
{
    assert (!self->slave_fn);
```

```
    self->slave_fn = handler;
    self->slave_arg = arg;
}


//  --------------------------------------------------------------------------
//  启用或禁止跟踪信息
void bstar_set_verbose (bstar_t *self, Bool verbose)
{
    zloop_set_verbose (self->loop, verbose);
}


//  --------------------------------------------------------------------------
//  开启反应堆，当回调函数返回-1，或进程收到SIGINT、SIGTERM信号时中止。

int
bstar_start (bstar_t *self)
{
    assert (self->voter_fn);
    return zloop_start (self->loop);
}
```

这样一来，我们的服务端代码会变得非常简短：

bstarsrv2: Binary Star server, using core class in C

```
//
//  双子星模式服务端，使用bstar反应堆
//

//  直接编译，不建类库
#include "bstar.c"

//  Echo service
int s_echo (zloop_t *loop, void *socket, void *arg)
{
    zmsg_t *msg = zmsg_recv (socket);
    zmsg_send (&msg, socket);
    return 0;
}

int main (int argc, char *argv [])
{
    //  命令行参数可以为：
    //      -p  作为主机启动，at tcp://localhost:5001
```

```
    //      -b  作为备机启动，at tcp://localhost:5002
    bstar_t *bstar;
    if (argc == 2 && streq (argv [1], "-p")) {
        printf ("I: 主机master，等待备机（slave）连接。\n");
        bstar = bstar_new (BSTAR_PRIMARY,
            "tcp://*:5003", "tcp://localhost:5004");
        bstar_voter (bstar, "tcp://*:5001", ZMQ_ROUTER, s_echo, NULL);
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        printf ("I: 备机slave，等待主机（master）连接。\n");
        bstar = bstar_new (BSTAR_BACKUP,
            "tcp://*:5004", "tcp://localhost:5003");
        bstar_voter (bstar, "tcp://*:5002", ZMQ_ROUTER, s_echo, NULL);
    }
    else {
        printf ("Usage: bstarsrvs { -p | -b }\n");
        exit (0);
    }
    bstar_start (bstar);
    bstar_destroy (&bstar);
    return 0;
}
```

无中间件的可靠性（自由者模式）

我们讲了那么多关于中间件的示例，好像有些违背“ZMQ是无中间件”的说法。但要知道在现实生活中，中间件一直是让人又爱又恨的东西。实践中的很多消息架构能都在使用中间件进行分布式架构的搭建，所以说最终的决定还是需要你自己去权衡的。这也是为什么虽然我能驾车10分钟到一个大型商场里购买五箱音量，但我还是会选择走10分钟到楼下的便利店里去买。这种出于经济方面的考虑（时间、精力、成本等）不仅在日常生活中很常见，在软件架构中也很重要。

这就是为什么ZMQ不会强制使用带有中间件的架构，但仍提供了像内置装置这样的中间件供编程人员自由选用。

这一节我们会打破以往使用中间件进行可靠性设计的架构，转而使用点对点架构，即自由者模式，来进行可靠的消息传输。我们的示例程序会是一个名称解析服务。ZMQ中的一个常见问题是：我们如何得知需要连接的端点？在代码中直接写入TCP/IP地址肯定是不合适的；使用配置文件会造成管理上的不便。试想一下，你要在上百台计算机中进行配置，只是为了让它们知道google.com的IP地址是74.125.230.82。

一个ZMQ的名称解析服务需要实现的功能有：

- 将逻辑名称解析为一个或多个端点地址，包括绑定端和连接端。实际使用时，名称服务会提供一组端点。
- 允许我们在不同的环境下，即开发环境和生产环境，进行解析；
- 该服务必须是可靠的，否则应用程序将无法连接到网络。

为管家模式提供名称解析服务会很有用，虽然将代理程序的端点对外暴露也很简单，但是如果用好名称解析服务，那它将成为唯一一个对外暴露的接口，将更便于管理。

我们需要处理的故障类型有：服务崩溃或重启、服务过载、网络因素等。为获取可靠性，我们必须建立一个服务群，当某个服务端崩溃后，客户端可以连接其他的服务端。实践中，两个服务端就已经足够了，但事实上服务端的数量可以是任意个。

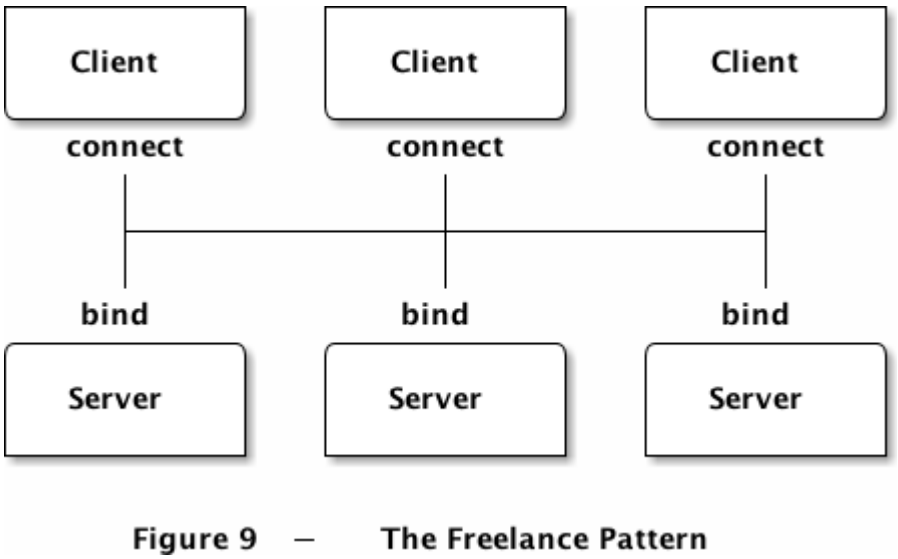

Figure 9 – The Freelance Pattern

在这个架构中，大量客户端和少量服务端进行通信，服务端将套接字绑定至单独的端口，这和管家模式中的代理有很大不同。对于客户端来说，它有这样几种选择：

- 客户端可以使用REQ套接字和懒惰海盗模式，但需要有一个机制防止客户端不断地请求已停止的服务端。
- 客户端可以使用DEALER套接字，向所有的服务端发送请求。很简单，但并不太妙；
- 客户端使用ROUTER套接字，连接特定的服务端。但客户端如何得知服务端的套接字标识呢？一种方式是让服务端主动连接客户端（很复杂），或者将服务端标识写入代码进行固化（很混乱）。

模型一：简单重试

让我们先尝试简单的方案，重写懒惰海盗模式，让其能够和多个服务端进行通信。启动服务端时用命令行参数指定端口。然后启动多个服务端。

flserver1: Freelance server, Model One in C

```
//
//  自由者模式 - 服务端 - 模型1
//  提供echo服务
//
#include "czmq.h"
```

```c
int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: echo服务端点： %s\n", argv [1]);
    while (TRUE) {
        zmsg_t *msg = zmsg_recv (server);
        if (!msg)
            break;          //  中断
        zmsg_send (&msg, server);
    }
    if (zctx_interrupted)
        printf ("W: 中断\n");

    zctx_destroy (&ctx);
    return 0;
}
```

启动客户端，指定一个或多个端点：

flclient1: Freelance client, Model One in C

```c
//
//  自由者模式 - 客户端 - 模型1
//  使用REQ套接字请求一个或多个服务端
//
#include "czmq.h"

#define REQUEST_TIMEOUT     1000
#define MAX_RETRIES         3       //  尝试次数


static zmsg_t *
s_try_request (zctx_t *ctx, char *endpoint, zmsg_t *request)
{
    printf ("I: 在端点 %s 上尝试请求echo服务...\n", endpoint);
    void *client = zsocket_new (ctx, ZMQ_REQ);
    zsocket_connect (client, endpoint);

    //  发送请求，并等待应答
```

```
    zmsg_t *msg = zmsg_dup (request);
    zmsg_send (&msg, client);
    zmq_pollitem_t items [] = { { client, 0, ZMQ_POLLIN, 0 } };
    zmq_poll (items, 1, REQUEST_TIMEOUT * ZMQ_POLL_MSEC);
    zmsg_t *reply = NULL;
    if (items [0].revents & ZMQ_POLLIN)
        reply = zmsg_recv (client);

    //  关闭套接字
    zsocket_destroy (ctx, client);
    return reply;
}


int main (int argc, char *argv [])
{
    zctx_t *ctx = zctx_new ();
    zmsg_t *request = zmsg_new ();
    zmsg_addstr (request, "Hello world");
    zmsg_t *reply = NULL;

    int endpoints = argc - 1;
    if (endpoints == 0)
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
    else
    if (endpoints == 1) {
        //  若只有一个端点，则尝试N次
        int retries;
        for (retries = 0; retries < MAX_RETRIES; retries++) {
            char *endpoint = argv [1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  成功
            printf ("W: 没有收到 %s 的应答，准备重试...\n", endpoint);
        }
    }
    else {
        //  若有多个端点，则每个尝试一次
        int endpoint_nbr;
        for (endpoint_nbr = 0; endpoint_nbr < endpoints; endpoint_nbr++) {
            char *endpoint = argv [endpoint_nbr + 1];
            reply = s_try_request (ctx, endpoint, request);
            if (reply)
                break;          //  Successful
            printf ("W: 没有收到 %s 的应答\n", endpoint);
        }
```

```
    }
    if (reply)
        printf ("服务运作正常\n");

    zmsg_destroy (&request);
    zmsg_destroy (&reply);
    zctx_destroy (&ctx);
    return 0;
}
```

可用如下命令运行：

```
flserver1 tcp://*:5555 &
flserver1 tcp://*:5556 &
flclient1 tcp://localhost:5555 tcp://localhost:5556
```

客户端的核心机制是懒惰海盗模式，即获得一次成功的应答后就结束。会有两种情况：

- 如果只有一个服务端，客户端会再尝试N次后停止，这和懒惰海盗模式的逻辑一致；
- 如果有多个服务端，客户端会每个尝试一次，收到应答后停止。

这种机制补充了海盗模式，使其能够克服只有一个服务端的情况。

但是，这种设计无法在现实程序中使用：当有很多客户端连接了服务端，而主服务端崩溃了，那所有客户端都需要在超时后才能继续执行。

模型二：批量发送

下面让我们使用DEALER套接字。我们的目标是能再最短的时间里收到一个应答，不能受主服务端崩溃的影响。可以采取以下措施：

- 连接所有的服务端；
- 当有请求时，一次性发送给所有的服务端；
- 等待第一个应答；
- 忽略其他应答。

这样设计客户端时，当发送请求后，所有的服务端都会收到这个请求，并返回应答。如果某个服务端断开连接了，ZMQ可能会将请求发给其他服务端，导致某些服务端会收到两次请求。

更麻烦的是客户端无法得知应答的数量，容易发生混乱。

我们可以为请求进行编号，忽略不匹配的应答。我们要对服务端进行改造，返回的消息中需要包含请求编号：
flserver2: Freelance server, Model Two in C

```
//
//  自由者模式 - 服务端 - 模型2
//  返回带有请求编号的OK信息
//
```

```c
#include "czmq.h"

int main (int argc, char *argv [])
{
    if (argc < 2) {
        printf ("I: syntax: %s <endpoint>\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    zctx_t *ctx = zctx_new ();
    void *server = zsocket_new (ctx, ZMQ_REP);
    zsocket_bind (server, argv [1]);

    printf ("I: 服务已就绪 %s\n", argv [1]);
    while (TRUE) {
        zmsg_t *request = zmsg_recv (server);
        if (!request)
            break;          //  中断
        //  判断请求内容是否正确
        assert (zmsg_size (request) == 2);

        zframe_t *address = zmsg_pop (request);
        zmsg_destroy (&request);

        zmsg_t *reply = zmsg_new ();
        zmsg_add (reply, address);
        zmsg_addstr (reply, "OK");
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: interrupted\n");

    zctx_destroy (&ctx);
    return 0;
}
```

客户端代码：

flclient2: Freelance client, Model Two in C

```c
//
//  自由者模式 - 客户端 - 模型2
//  使用DEALER套接字发送批量消息
//
#include "czmq.h"

//  超时时间
```

```c
#define GLOBAL_TIMEOUT 2500

//  将客户端API封装成一个类

#ifdef __cplusplus
extern "C" {
#endif

//  声明类结构
typedef struct _flclient_t flclient_t;

flclient_t *
    flclient_new (void);
void
    flclient_destroy (flclient_t **self_p);
void
    flclient_connect (flclient_t *self, char *endpoint);
zmsg_t *
    flclient_request (flclient_t *self, zmsg_t **request_p);

#ifdef __cplusplus
}
#endif


int main (int argc, char *argv [])
{
    if (argc == 1) {
        printf ("I: syntax: %s <endpoint> ...\n", argv [0]);
        exit (EXIT_SUCCESS);
    }
    //  创建自由者模式客户端
    flclient_t *client = flclient_new ();

    //  连接至各个端点
    int argn;
    for (argn = 1; argn < argc; argn++)
        flclient_connect (client, argv [argn]);

    //  发送一组请求，并记录时间
    int requests = 10000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flclient_request (client, &request);
```

```
        if (!reply) {
            printf ("E: 名称解析服务不可用，正在退出\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("平均请求时间: %d 微秒\n",
        (int) (zclock_time () - start) / 10);

    flclient_destroy (&client);
    return 0;
}



//  ---------------------------------------------------------------------
//  类结构

struct _flclient_t {
    zctx_t *ctx;        //  上下文
    void *socket;       //  用于和服务端通信的DEALER套接字
    size_t servers;     //  以连接的服务端数量
    uint sequence;      //  已发送的请求数
};


//  ---------------------------------------------------------------------
//  Constructor

flclient_t *
flclient_new (void)
{
    flclient_t
        *self;

    self = (flclient_t *) zmalloc (sizeof (flclient_t));
    self->ctx = zctx_new ();
    self->socket = zsocket_new (self->ctx, ZMQ_DEALER);
    return self;
}

//  ---------------------------------------------------------------------
//  析构函数

void
flclient_destroy (flclient_t **self_p)
```

```
{
    assert (self_p);
    if (*self_p) {
        flclient_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  --------------------------------------------------------------------
//  连接至新的服务端端点

void
flclient_connect (flclient_t *self, char *endpoint)
{
    assert (self);
    zsocket_connect (self->socket, endpoint);
    self->servers++;
}

//  --------------------------------------------------------------------
//  发送请求，接收应答
//  发送后销毁请求

zmsg_t *
flclient_request (flclient_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);
    zmsg_t *request = *request_p;

    //  向消息添加编号和空帧
    char sequence_text [10];
    sprintf (sequence_text, "%u", ++self->sequence);
    zmsg_pushstr (request, sequence_text);
    zmsg_pushstr (request, "");

    //  向所有已连接的服务端发送请求
    int server;
    for (server = 0; server < self->servers; server++) {
        zmsg_t *msg = zmsg_dup (request);
        zmsg_send (&msg, self->socket);
    }
    //  接收来自任何服务端的应答
    //  因为我们可能poll多次，所以每次都进行计算
```

```
    zmsg_t *reply = NULL;
    uint64_t endtime = zclock_time () + GLOBAL_TIMEOUT;
    while (zclock_time () < endtime) {
        zmq_pollitem_t items [] = { { self->socket, 0, ZMQ_POLLIN, 0 } };
        zmq_poll (items, 1, (endtime - zclock_time ()) * ZMQ_POLL_MSEC);
        if (items [0].revents & ZMQ_POLLIN) {
            //  应答内容是 [empty][sequence][OK]
            reply = zmsg_recv (self->socket);
            assert (zmsg_size (reply) == 3);
            free (zmsg_popstr (reply));
            char *sequence = zmsg_popstr (reply);
            int sequence_nbr = atoi (sequence);
            free (sequence);
            if (sequence_nbr == self->sequence)
                break;
        }
    }
    zmsg_destroy (request_p);
    return reply;
}
```

几点说明：

- 客户端被封装成了一个API类，将复杂的代码都包装了起来。
- 客户端会在几秒之后放弃寻找可用的服务端；
- 客户端需要创建一个合法的REP信封，所以需要添加一个空帧。

程序中，客户端发出了1万次名称解析请求（虽然是假的），并计算平均耗费时间。在我的测试机上，有一个服务端时，耗时60微妙；三个时80微妙。

该模型的优缺点是：

- 优点：简单，容易理解和编写；
- 优点：它工作迅速，有重试机制；
- 缺点：占用了额外的网络带宽；
- 缺点：我们不能为服务端设置优先级，如主服务、次服务等；
- 缺点：服务端不能同时处理多个请求。

Model Three - Complex and Nasty

批量发送模型看起来不太真实，那就让我们来探索最后这个极度复杂的模型。很有可能在编写完之后我们又会转而使用批量发送，哈哈，这就是我的作风。

我们可以将客户端使用的套接字更换为ROUTER，让我们能够向特定的服务端发送请求，停止向已死亡的服务端发送请求，从而做得尽可能地智能。我们还可以将服务端的套接字更换为ROUTER，从而突破单线程的瓶颈。

但是，使用ROUTER-ROUTER套接字连接两个瞬时套接字是不可行的，节点只有在收到第一条消息时才会为对方生成套接字标识。唯一的方法是让其中一个节点使用持久化的套接字，比较好的方式是让客户端知道服务端的标识，即服务端作为持久化的套接字。

为了避免产生新的配置项，我们直接使用服务端的端点作为套接字标识。

回想一下ZMQ套接字标识是如何工作的。服务端的ROUTER套接字为自己设置一个标识（在绑定之前），当客户端连接时，通过一个握手的过程来交换双方的标识。客户端的ROUTER套接字会先发送一条空消息，服务端为客户端生成一个随机的UUID。然后，服务端会向客户端发送自己的标识。

这样一来，客户端就可以将消息发送给特定的服务端了。不过还有一个问题：我们不知道服务端会在什么时候完成这个握手的过程。如果服务端是在线的，那可能几毫秒就能完成。如果不在线，那可能需要很久很久。

这里有一个矛盾：我们需要知道服务端何时连接成功且能够开始工作。自由者模式不像中间件模式，它的服务端必须要先发送请求后才能的应答。所以在服务端发送消息给客户端之前，客户端必须要先请求服务端，这看似是不可能的。

我有一个解决方法，那就是批量发送。这里发送的不是真正的请求，而是一个试探性的心跳（PING-PONG）。当收到应答时，就说明对方是在线的。

下面让我们制定一个协议，来定义自由者模式是如何传递这种心跳的：

- http://rfc.zeromq.org/spec:10

实现这个协议的服务端很方便，下面就是经过改造的echo服务：

flserver3: Freelance server, Model Three in C

```
//
//  自由者模式 - 服务端 - 模型3
//  使用ROUTER-ROUTER套接字进行通信；单线程。
//
#include "czmq.h"

int main (int argc, char *argv [])
{
    int verbose = (argc > 1 && streq (argv [1], "-v"));

    zctx_t *ctx = zctx_new ();

    //  准备服务端套接字，其标识和端点名相同
    char *bind_endpoint = "tcp://*:5555";
    char *connect_endpoint = "tcp://localhost:5555";
    void *server = zsocket_new (ctx, ZMQ_ROUTER);
    zmq_setsockopt (server,
        ZMQ_IDENTITY, connect_endpoint, strlen (connect_endpoint));
    zsocket_bind (server, bind_endpoint);
    printf ("I: 服务端已准备就绪 %s\n", bind_endpoint);
```

```
    while (!zctx_interrupted) {
        zmsg_t *request = zmsg_recv (server);
        if (verbose && request)
            zmsg_dump (request);
        if (!request)
            break;          //  中断

        //  Frame 0: 客户端标识
        //  Frame 1: 心跳，或客户端控制信息帧
        //  Frame 2: 请求内容
        zframe_t *address = zmsg_pop (request);
        zframe_t *control = zmsg_pop (request);
        zmsg_t *reply = zmsg_new ();
        if (zframe_streq (control, "PONG"))
            zmsg_addstr (reply, "PONG");
        else {
            zmsg_add (reply, control);
            zmsg_addstr (reply, "OK");
        }
        zmsg_destroy (&request);
        zmsg_push (reply, address);
        if (verbose && reply)
            zmsg_dump (reply);
        zmsg_send (&reply, server);
    }
    if (zctx_interrupted)
        printf ("W: 中断\n");

    zctx_destroy (&ctx);
    return 0;
}
```

但是，自由者模式的客户端会变得大一写。为了清晰期间，我们将其拆分为两个类来实现。首先是在上层使用的程序：

flclient3: Freelance client, Model Three in C

```
//
//  自由者模式 - 客户端 - 模型3
//  使用flcliapi类来封装自由者模式
//
//  直接编译，不建类库
#include "flcliapi.c"

int main (void)
{
```

```
    //  创建自由者模式实例
    flcliapi_t *client = flcliapi_new ();

    //  链接至服务器端点
    flcliapi_connect (client, "tcp://localhost:5555");
    flcliapi_connect (client, "tcp://localhost:5556");
    flcliapi_connect (client, "tcp://localhost:5557");

    //  发送随机请求，计算时间
    int requests = 1000;
    uint64_t start = zclock_time ();
    while (requests--) {
        zmsg_t *request = zmsg_new ();
        zmsg_addstr (request, "random name");
        zmsg_t *reply = flcliapi_request (client, &request);
        if (!reply) {
            printf ("E: 名称解析服务不可用，正在退出\n");
            break;
        }
        zmsg_destroy (&reply);
    }
    printf ("平均执行时间： %d usec\n",
        (int) (zclock_time () - start) / 10);

    flcliapi_destroy (&client);
    return 0;
}
```

下面是该模式复杂的实现过程：

flcliapi: Freelance client API in C

```
/*  =====================================================================
    flcliapi - Freelance Pattern agent class
    Model 3: uses ROUTER socket to address specific services

    ---------------------------------------------------------------------
    Copyright (c) 1991-2011 iMatix Corporation <www.imatix.com>
    Copyright other contributors as noted in the AUTHORS file.

    This file is part of the ZeroMQ Guide: http://zguide.zeromq.org

    This is free software; you can redistribute it and/or modify it under
    the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation; either version 3 of the License, or (at
    your option) any later version.
```

```
    This software is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this program. If not, see
    <http://www.gnu.org/licenses/>.
    =========================================================================
*/

#include "flcliapi.h"

//  请求超时时间
#define GLOBAL_TIMEOUT  3000    //  msecs
//  心跳间隔
#define PING_INTERVAL   2000    //  msecs
//  判定服务死亡的时间
#define SERVER_TTL      6000    //  msecs


//  ---------------------------------------------------------------------
//  同步部分，在应用程序层面运行

//  ---------------------------------------------------------------------
//  类结构

struct _flcliapi_t {
    zctx_t *ctx;        //  上下文
    void *pipe;         //  用于和主线程通信的套接字
};

//  这是运行后台代理程序的线程
static void flcliapi_agent (void *args, zctx_t *ctx, void *pipe);


//  ---------------------------------------------------------------------
//  构造函数

flcliapi_t *
flcliapi_new (void)
{
    flcliapi_t
        *self;
```

```
    self = (flcliapi_t *) zmalloc (sizeof (flcliapi_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, flcliapi_agent, NULL);
    return self;
}

//  --------------------------------------------------------------------------
//  析构函数

void
flcliapi_destroy (flcliapi_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        flcliapi_t *self = *self_p;
        zctx_destroy (&self->ctx);
        free (self);
        *self_p = NULL;
    }
}

//  --------------------------------------------------------------------------
//  连接至新服务器端点
//  消息内容：[CONNECT][endpoint]

void
flcliapi_connect (flcliapi_t *self, char *endpoint)
{
    assert (self);
    assert (endpoint);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, endpoint);
    zmsg_send (&msg, self->pipe);
    zclock_sleep (100);      //  等待连接
}

//  --------------------------------------------------------------------------
//  发送并销毁请求，接收应答

zmsg_t *
flcliapi_request (flcliapi_t *self, zmsg_t **request_p)
{
    assert (self);
    assert (*request_p);
```

```
    zmsg_pushstr (*request_p, "REQUEST");
    zmsg_send (request_p, self->pipe);
    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *status = zmsg_popstr (reply);
        if (streq (status, "FAILED"))
            zmsg_destroy (&reply);
        free (status);
    }
    return reply;
}


//  ========================================================================
//  异步部分，在后台运行

//  ------------------------------------------------------------------------
//  单个服务端信息

typedef struct {
    char *endpoint;             //  服务端端点/套接字标识
    uint alive;                 //  是否在线
    int64_t ping_at;            //  下一次心跳时间
    int64_t expires;            //  过期时间
} server_t;

server_t *
server_new (char *endpoint)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));
    self->endpoint = strdup (endpoint);
    self->alive = 0;
    self->ping_at = zclock_time () + PING_INTERVAL;
    self->expires = zclock_time () + SERVER_TTL;
    return self;
}

void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->endpoint);
        free (self);
        *self_p = NULL;
```

```
    }
}

int
server_ping (char *key, void *server, void *socket)
{
    server_t *self = (server_t *) server;
    if (zclock_time () >= self->ping_at) {
        zmsg_t *ping = zmsg_new ();
        zmsg_addstr (ping, self->endpoint);
        zmsg_addstr (ping, "PING");
        zmsg_send (&ping, socket);
        self->ping_at = zclock_time () + PING_INTERVAL;
    }
    return 0;
}

int
server_tickless (char *key, void *server, void *arg)
{
    server_t *self = (server_t *) server;
    uint64_t *tickless = (uint64_t *) arg;
    if (*tickless > self->ping_at)
        *tickless = self->ping_at;
    return 0;
}


//  ---------------------------------------------------------------------
//  后台处理程序信息

typedef struct {
    zctx_t *ctx;                //  上下文
    void *pipe;                 //  用于应用程序通信的套接字
    void *router;               //  用于服务端通信的套接字
    zhash_t *servers;           //  已连接的服务端
    zlist_t *actives;           //  在线的服务端
    uint sequence;              //  请求编号
    zmsg_t *request;            //  当前请求
    zmsg_t *reply;              //  当前应答
    int64_t expires;            //  请求过期时间
} agent_t;

agent_t *
agent_new (zctx_t *ctx, void *pipe)
{
```

```
    agent_t *self = (agent_t *) zmalloc (sizeof (agent_t));
    self->ctx = ctx;
    self->pipe = pipe;
    self->router = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->servers = zhash_new ();
    self->actives = zlist_new ();
    return self;
}

void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        zhash_destroy (&self->servers);
        zlist_destroy (&self->actives);
        zmsg_destroy (&self->request);
        zmsg_destroy (&self->reply);
        free (self);
        *self_p = NULL;
    }
}

//  当服务端从列表中移除时，回调该函数。

static void
s_server_free (void *argument)
{
    server_t *server = (server_t *) argument;
    server_destroy (&server);
}

void
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);

    if (streq (command, "CONNECT")) {
        char *endpoint = zmsg_popstr (msg);
        printf ("I: connecting to %s...\n", endpoint);
        int rc = zmq_connect (self->router, endpoint);
        assert (rc == 0);
        server_t *server = server_new (endpoint);
        zhash_insert (self->servers, endpoint, server);
```

```
        zhash_freefn (self->servers, endpoint, s_server_free);
        zlist_append (self->actives, server);
        server->ping_at = zclock_time () + PING_INTERVAL;
        server->expires = zclock_time () + SERVER_TTL;
        free (endpoint);
    }
    else
    if (streq (command, "REQUEST")) {
        assert (!self->request);    //  遵循请求-应答循环
        //  将请求编号和空帧加入消息顶部
        char sequence_text [10];
        sprintf (sequence_text, "%u", ++self->sequence);
        zmsg_pushstr (msg, sequence_text);
        //  获取请求消息的所有权
        self->request = msg;
        msg = NULL;
        //  设置请求过期时间
        self->expires = zclock_time () + GLOBAL_TIMEOUT;
    }
    free (command);
    zmsg_destroy (&msg);
}

void
agent_router_message (agent_t *self)
{
    zmsg_t *reply = zmsg_recv (self->router);

    //  第一帧是应答的服务端标识
    char *endpoint = zmsg_popstr (reply);
    server_t *server =
        (server_t *) zhash_lookup (self->servers, endpoint);
    assert (server);
    free (endpoint);
    if (!server->alive) {
        zlist_append (self->actives, server);
        server->alive = 1;
    }
    server->ping_at = zclock_time () + PING_INTERVAL;
    server->expires = zclock_time () + SERVER_TTL;

    //  第二帧是应答的编号
    char *sequence = zmsg_popstr (reply);
    if (atoi (sequence) == self->sequence) {
        zmsg_pushstr (reply, "OK");
        zmsg_send (&reply, self->pipe);
```

```
        zmsg_destroy (&self->request);
    }
    else
        zmsg_destroy (&reply);
}


//  --------------------------------------------------------------------
//  异步的后台代理会维护一个服务端池，处理请求和应答。

static void
flcliapi_agent (void *args, zctx_t *ctx, void *pipe)
{
    agent_t *self = agent_new (ctx, pipe);

    zmq_pollitem_t items [] = {
        { self->pipe, 0, ZMQ_POLLIN, 0 },
        { self->router, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        //  计算超时时间
        uint64_t tickless = zclock_time () + 1000 * 3600;
        if (self->request
        &&  tickless > self->expires)
            tickless = self->expires;
        zhash_foreach (self->servers, server_tickless, &tickless);

        int rc = zmq_poll (items, 2,
            (tickless - zclock_time ()) * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文对象被关闭

        if (items [0].revents & ZMQ_POLLIN)
            agent_control_message (self);

        if (items [1].revents & ZMQ_POLLIN)
            agent_router_message (self);

        //  如果我们需要处理一项请求，将其发送给下一个可用的服务端
        if (self->request) {
            if (zclock_time () >= self->expires) {
                //  请求超时
                zstr_send (self->pipe, "FAILED");
                zmsg_destroy (&self->request);
            }
            else {
```

```
                //  寻找可用的服务端
                while (zlist_size (self->actives)) {
                    server_t *server =
                        (server_t *) zlist_first (self->actives);
                    if (zclock_time () >= server->expires) {
                        zlist_pop (self->actives);
                        server->alive = 0;
                    }
                    else {
                        zmsg_t *request = zmsg_dup (self->request);
                        zmsg_pushstr (request, server->endpoint);
                        zmsg_send (&request, self->router);
                        break;
                    }
                }
            }
        }
        //  断开并删除已过期的服务端
        //  发送心跳给空闲服务器
        zhash_foreach (self->servers, server_ping, self->router);
    }
    agent_destroy (&self);
}
```

这组API使用了较为复杂的机制，我们之前也有用到过：

异步后台代理

客户端API由两部分组成：同步的flcliapi类，运行于应用程序线程；异步的agent类，运行于后台线程。flcliapi和agent类通过一个inproc套接字互相通信。所有和ZMQ相关的内容都封装在API中。agent类实质上是作为一个迷你的代理程序在运行，负责在后台与服务端进行通信，只要我们发送请求，它就会设法连接一个服务器来处理请求。

连接等待机制

ROUTER套接字的特点之一是会直接丢弃无法路由的消息，这就意味着当与服务器建立了ROUTER-ROUTER连接后，如果立刻发送一条消息，该消息是会丢失的。flcliapi类则延迟了一会儿后再发送消息。之后的通信中，由于服务端套接字是持久的，客户端就不再丢弃消息了。

Ping silence

0MQ will queue messages for a dead server indefinitely. So if a client repeatedly PINGs a dead server, when that server comes back to life it'll get a whole bunch of PING messages all at once. Rather than continuing to ping a server we know is offline, we count on 0MQ's handling of durable sockets to deliver the old PING messages when the server comes back online. As soon as a server reconnects, it'll get PINGs from all clients that were connected to it, it'll PONG back, and those clients will recognize it as alive again.

调整轮询时间

在之前的示例程序中，我们一般会为轮询设置固定的超时时间（如1秒），这种做法虽然简单，但是对于用电较为敏感的设备来说（如笔记本电脑或手机）唤醒CPU是需要额外的电力的。所以，为了完美也好，好玩也好，我们这里调整了轮询时间，将其设置为到达过期时间时才超时，这样就能节省一部分轮询次数了。我们可以将过期时间放入一个列表中存储，方便查询。

总结

这一章中我们看到了很多可靠的请求-应答机制，每种机制都有其优劣性。大部分示例代码是可以直接用于生产环境的，不过还可以进一步优化。有两个模式会比较典型：使用了中间件的管家模式，以及未使用中间件的自由者模式。