1. 基本用法：

1.zmq_init()创建一个 zmq context 也就是上下文

2.zmq_socket()建立个 socket

3.zmqbind()/zmq_connect()把 socket 连接到某个信道上

4.zmq_setsockopt()/zmq_getsockopt()设置 socket 参数（filter，HWM 等）

5.s_send()（非阻塞）/s_recv()（默认阻塞）发和收数据

6.zmq_close()关闭 socket，zmq_term()关闭上下文

2. 支持的地址类型：

1.inproc　　线程间

2.ipc　进程间

3.tcp　网络间

4.pgm　广播

5.epgm

3. 目前支持的 zmq_socket 类型，和几种 connect-bind 的组合：

PUB and SUB

REQ and REP

REQ and ROUTER (take care, REQ inserts an extra null frame)

DEALER and REP (take care, REP assumes a null frame)

DEALER and ROUTER

DEALER and DEALER

ROUTER and ROUTER

PUSH and PULL

PAIR and PAIR

4. 几个常用的网络模型和对应的图示

4.1.request-reply 模型：（涉及 ZMQ_REQ/ZMQ_REP）

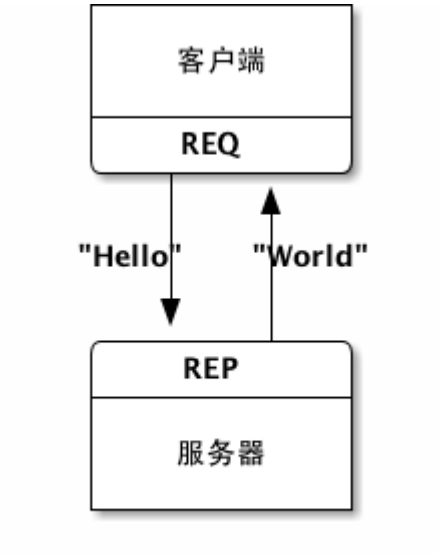

Figure 1 — Request-Reply

```
//
//  Hello World 服务端
//  绑定一个 REP 套接字至 tcp://*:5555
//  从客户端接收 Hello，并应答 World
//
#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

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

    //  与客户端通信的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_bind (responder, "tcp://*:5555");

    while (1) {
        //  等待客户端请求
        zmq_msg_t request;
        zmq_msg_init (&request);
        zmq_recv (responder, &request, 0);
        printf ("收到 Hello\n");
        zmq_msg_close (&request);

        //  做些“处理”
        sleep (1);
```

```
        //  返回应答
        zmq_msg_t reply;
        zmq_msg_init_size (&reply, 5);
        memcpy (zmq_msg_data (&reply), "World", 5);
        zmq_send (responder, &reply, 0);
        zmq_msg_close (&reply);
    }
    //  程序不会运行到这里，以下只是演示我们应该如何结束
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
//
//  Hello World 客户端
//  连接 REQ 套接字至 tcp://localhost:5555
//  发送 Hello 给服务端，并接收 World
//
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

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

    //  连接至服务端的套接字
    printf ("正在连接至 hello world 服务端...\n");
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        zmq_msg_t request;
        zmq_msg_init_size (&request, 5);
        memcpy (zmq_msg_data (&request), "Hello", 5);
        printf ("正在发送 Hello %d...\n", request_nbr);
        zmq_send (requester, &request, 0);
        zmq_msg_close (&request);

        zmq_msg_t reply;
        zmq_msg_init (&reply);
        zmq_recv (requester, &reply, 0);
        printf ("接收到 World %d\n", request_nbr);
        zmq_msg_close (&reply);
    }
    zmq_close (requester);
    zmq_term (context);
    return 0;
}
```

4.2.publish-subscribe 模型：（涉及 ZMQ_PUB/ZMQ_SUB）

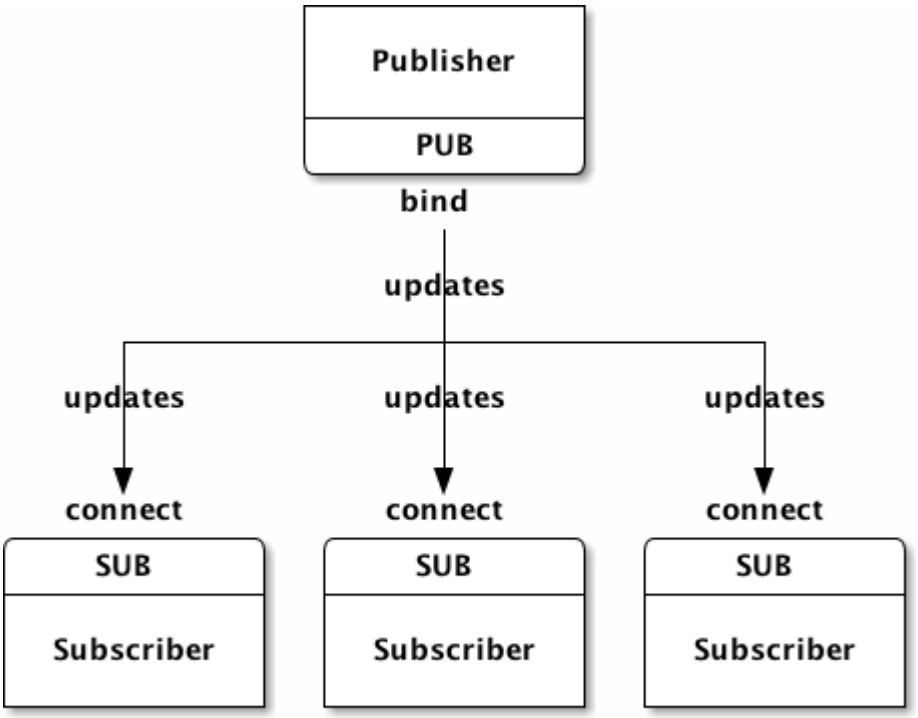

Figure 4 – Publish-Subscribe

下面是服务端的代码，使用 5556 端口：

wuserver: Weather update server in C

```
//
//  气象信息更新服务
//  绑定 PUB 套接字至 tcp://*:5556 端点
//  发布随机气象信息
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和 PUB 套接字
    void *context = zmq_init (1);
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5556");
    zmq_bind (publisher, "ipc://weather.ipc");

    //  初始化随机数生成器
    srandom ((unsigned) time (NULL));
    while (1) {
        //  生成数据
```

```
        int zipcode, temperature, relhumidity;
        zipcode     = randof (100000);
        temperature = randof (215) - 80;
        relhumidity = randof (50) + 10;

        //  向所有订阅者发送消息
        char update [20];
        sprintf (update, "%05d %d %d", zipcode, temperature,
relhumidity);
        s_send (publisher, update);
    }
    zmq_close (publisher);
    zmq_term (context);
    return 0;
}
```

下面是客户端程序，它会接受发布者的消息，只处理特定邮编标注的信息，如纽约的邮编是 10001:

wuclient: Weather update client in C

```
//
//  气象信息客户端
//  连接 SUB 套接字至 tcp://*:5556 端点
//  收集指定邮编的气象信息，并计算平均温度
//
#include "zhelpers.h"

int main (int argc, char *argv [])
{
    void *context = zmq_init (1);

    //  创建连接至服务端的套接字
    printf ("正在收集气象信息...\n");
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");

    //  设置订阅信息，默认为纽约，邮编 10001
    char *filter = (argc > 1)? argv [1]: "10001 ";
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, filter, strlen (filter));

    //  处理 100 条更新信息
    int update_nbr;
    long total_temp = 0;
    for (update_nbr = 0; update_nbr < 100; update_nbr++) {
        char *string = s_recv (subscriber);
        int zipcode, temperature, relhumidity;
        sscanf (string, "%d %d %d",
```

```
            &zipcode, &temperature, &relhumidity);
        total_temp += temperature;
        free (string);
    }
    printf ("地区邮编 '%s' 的平均温度为 %dF\n",
        filter, (int) (total_temp / update_nbr));

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

注意 publish-subscribe 之间，不止可以 1:1 的关系，也可以是 N:M 的关系，即多对多的关系。

pub 要用 zmq_bind

sub 要用 zmq_connect

4.3.分布式处理任务所需的 push-pull 模型：（涉及 ZMQ_PUSH/ZMQ_PULL）

下面一个示例程序中，我们将使用 ZMQ 进行超级计算，也就是并行处理模型：

- 任务分发器会生成大量可以并行计算的任务；
- 有一组 worker 会处理这些任务；
- 结果收集器会在末端接收所有 worker 的处理结果，进行汇总。

现实中，worker 可能散落在不同的计算机中，利用 GPU（图像处理单元）进行复杂计算。下面是任务分发器的代码，它会生成 100 个任务，任务内容是让收到的 worker 延迟若干毫秒。

taskvent: Parallel task ventilator in C

```
//
//  任务分发器
//  绑定 PUSH 套接字至 tcp://localhost:5557 端点
//  发送一组任务给已建立连接的 worker
//
#include "zhelpers.h"

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

    //  用于发送消息的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_bind (sender, "tcp://*:5557");

    //  用于发送开始信号的套接字
```

```
    void *sink = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sink, "tcp://localhost:5558");

    printf ("准备好 worker 后按任意键开始: ");
    getchar ();
    printf ("正在向 worker 分配任务...\n");

    //  发送开始信号
    s_send (sink, "0");

    //  初始化随机数生成器
    srandom ((unsigned) time (NULL));

    //  发送 100 个任务
    int task_nbr;
    int total_msec = 0;     //  预计执行时间(毫秒)
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        int workload;
        //  随机产生 1-100 毫秒的工作量
        workload = randof (100) + 1;
        total_msec += workload;
        char string [10];
        sprintf (string, "%d", workload);
        s_send (sender, string);
    }
    printf ("预计执行时间: %d 毫秒\n", total_msec);
    sleep (1);              //  延迟一段时间,让任务分发完成

    zmq_close (sink);
    zmq_close (sender);
    zmq_term (context);
    return 0;
}
```

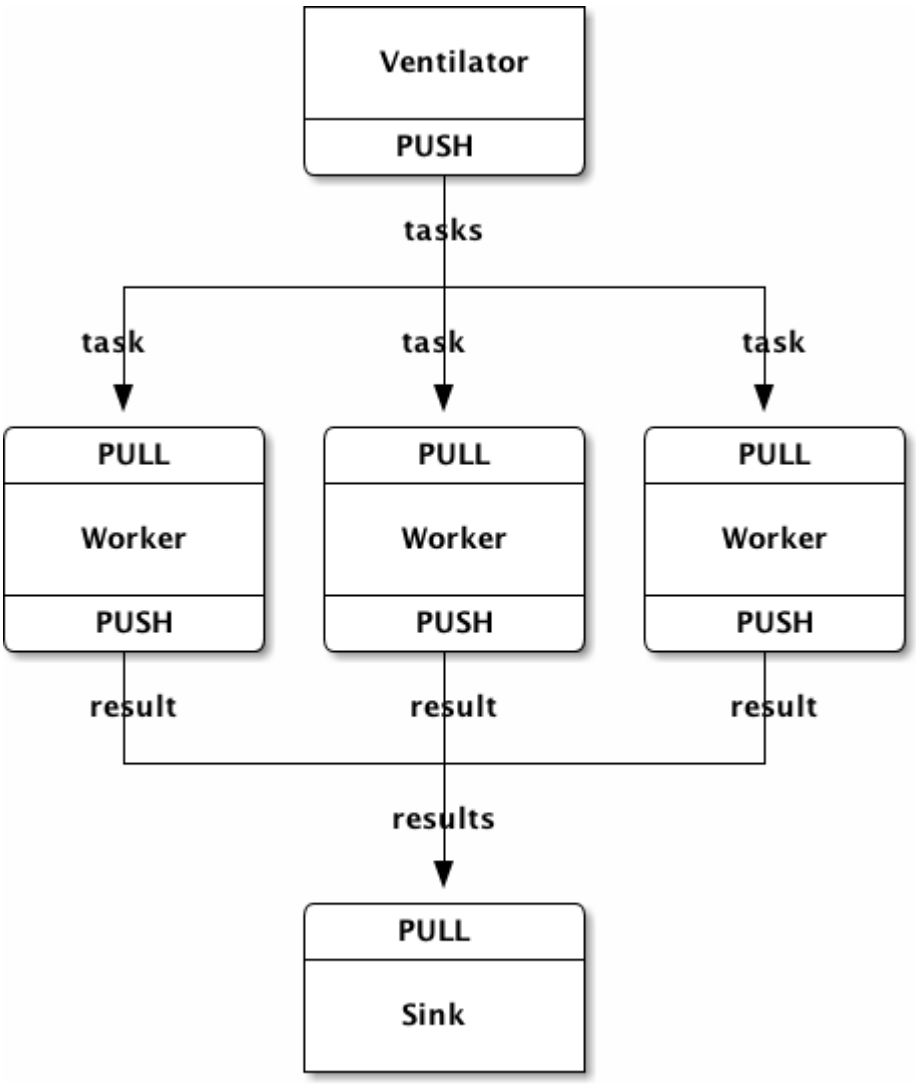

Figure 5 – Parallel Pipeline

下面是 worker 的代码，它接受信息并延迟指定的毫秒数，并发送执行完毕的信号：

taskwork: Parallel task worker in C

```
//
//  任务执行器
//  连接 PULL 套接字至 tcp://localhost:5557 端点
//  从任务分发器处获取任务
//  连接 PUSH 套接字至 tcp://localhost:5558 端点
//  向结果采集器发送结果
//
#include "zhelpers.h"
```

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

    //  获取任务的套接字
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_connect (receiver, "tcp://localhost:5557");

    //  发送结果的套接字
    void *sender = zmq_socket (context, ZMQ_PUSH);
    zmq_connect (sender, "tcp://localhost:5558");

    //  循环处理任务
    while (1) {
        char *string = s_recv (receiver);
        //  输出处理进度
        fflush (stdout);
        printf ("%s.", string);

        //  开始处理
        s_sleep (atoi (string));
        free (string);

        //  发送结果
        s_send (sender, "");
    }
    zmq_close (receiver);
    zmq_close (sender);
    zmq_term (context);
    return 0;
}
```

下面是结果收集器的代码。它会收集 100 个处理结果，并计算总的执行时间，让我们由此判别任务是否是并行计算的。

tasksink: Parallel task sink in C

```
//
//  任务收集器
//  绑定 PULL 套接字至 tcp://localhost:5558 端点
//  从 worker 处收集处理结果
//
#include "zhelpers.h"

int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_init (1);
    void *receiver = zmq_socket (context, ZMQ_PULL);
    zmq_bind (receiver, "tcp://*:5558");
```

```
    //  等待开始信号
    char *string = s_recv (receiver);
    free (string);

    //  开始计时
    int64_t start_time = s_clock ();

    //  确定 100 个任务均已处理
    int task_nbr;
    for (task_nbr = 0; task_nbr < 100; task_nbr++) {
        char *string = s_recv (receiver);
        free (string);
        if ((task_nbr / 10) * 10 == task_nbr)
            printf (":");
        else
            printf (".");
        fflush (stdout);
    }
    //  计算并输出总执行时间
    printf ("执行时间: %d 毫秒\n",
        (int) (s_clock () - start_time));

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

一组任务的平均执行时间在 5 秒左右，以下是分别开始 1 个、2 个、4 个 worker 时的执行结果：

```
#   1 worker
Total elapsed time: 5034 msec
#   2 workers
Total elapsed time: 2421 msec
#   4 workers
Total elapsed time: 1018 msec
```

4.4.在内网 subscribe 到的消息，publish 给外网（即 publish-subscribe 代理，涉及 ZMQ_PUB/ZMQ_SUB）

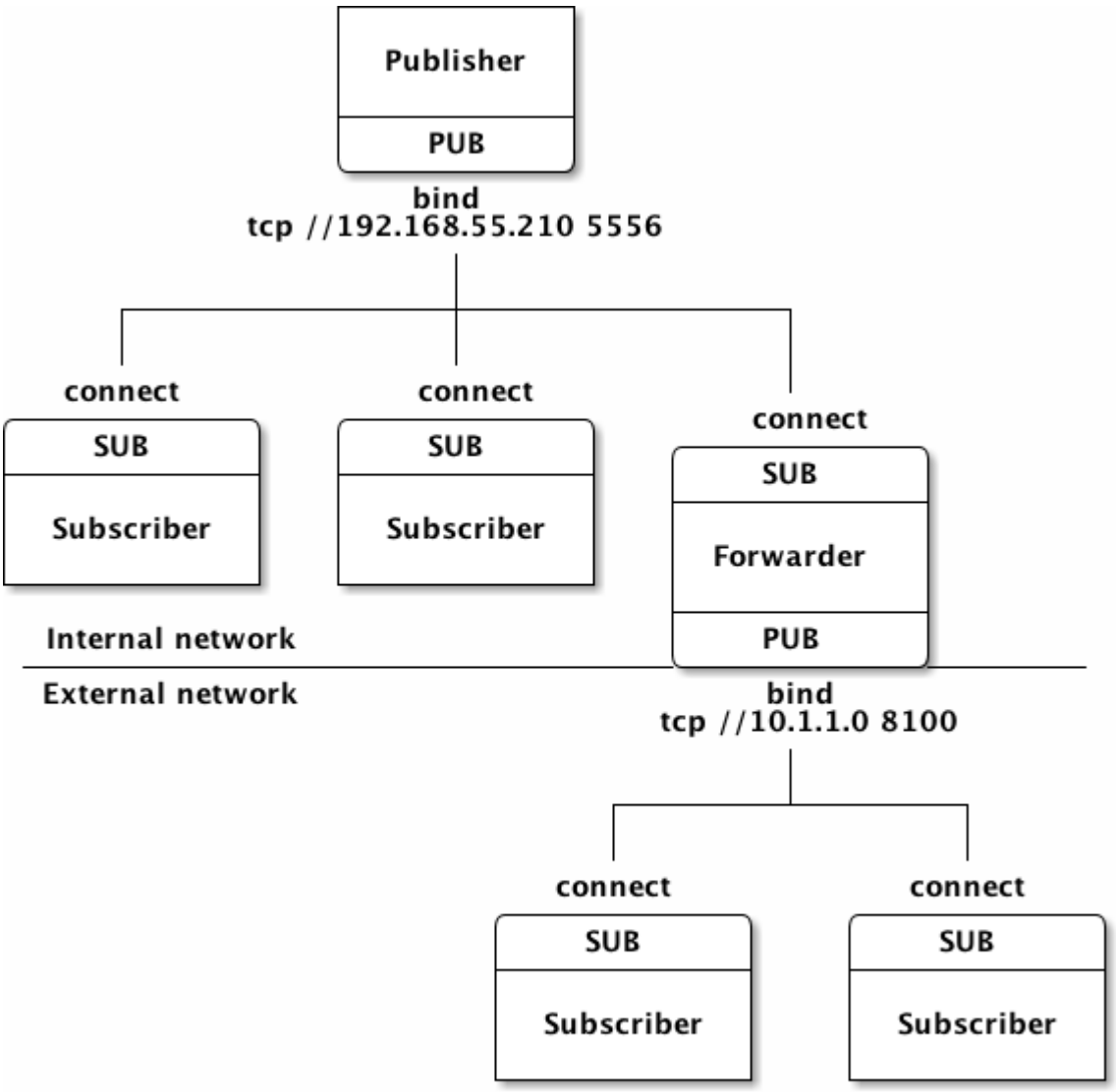

Figure 8 — Forwarder proxy device

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

wuproxy: Weather update proxy in C

```
//
//  气象信息代理服务装置
//
#include "zhelpers.h"

int main (void)
```

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

    //  订阅气象信息
    void *frontend = zmq_socket (context, ZMQ_SUB);
    zmq_connect (frontend, "tcp://192.168.55.210:5556");

    //  转发气象信息
    void *backend = zmq_socket (context, ZMQ_PUB);
    zmq_bind (backend, "tcp://10.1.1.0:8100");

    //  订阅所有消息
    zmq_setsockopt (frontend, ZMQ_SUBSCRIBE, "", 0);

    //  转发消息
    while (1) {
        while (1) {
            zmq_msg_t message;
            int64_t more;

            //  处理所有的消息帧
            zmq_msg_init (&message);
            zmq_recv (frontend, &message, 0);
            size_t more_size = sizeof (more);
            zmq_getsockopt (frontend, ZMQ_RCVMORE, &more, &more_size);
            zmq_send (backend, &message, more? ZMQ_SNDMORE: 0);
            zmq_msg_close (&message);
            if (!more)
                break;      //  到达最后一帧
        }
    }
    //  程序不会运行到这里，但依然要正确地退出
    zmq_close (frontend);
    zmq_close (backend);
    zmq_term (context);
    return 0;
}
```

这种由内网往外网的转发代理，有一个专门的节点（被称为“装置”）完成，这个节点，是内网的 sub，又是外网的 pub。内部实现的逻辑就是把从内网 sub 到的信息，转手 pub 给外网。

代理中需要实现：

新建两个 socket，一个 sub 一个 pub，分别对应内网外网。sub connect 到内网的 pub 地址，pub 则 bind 到外网的某个网络端口。sub 收到的东西，转手发给 pub

信息的传递在 publish-subscribe 代理转发的时候，是单向的，总是 pub->sub。

zmq 对此有一个内置的装置，专门做 publish-subscribe 代理，即 ZMQ_FORWARDER。

4.5.多个 request 需要连接多个 reply（N:M 多对多连接），中间用 broker 代理，使用 router-dealer 机制来提高扩展性的模型：（即 req-rep 代理，涉及 ZMQ_REQ/ZMQ_REP 和 ZMQ_DEALER/ZMQ_ROUTER）

上面讲了 publish-subscribe 代理，其原因是遇到类似内网外网的情况，需要用这个代理。

那为什么要用 req-rep 代理？

假设有 N 个 req，需要连接 M 个 rep。那如果要添加 rep 服务端，要修改所有相关的 req 客户端的配置，非常复杂

如果要做到，任意增减 N 和 M，而不需要因此改动其相关的节点的配置，这时候就需要一个中间的代理，即 req-rep 代理

由 router 连接 request 客户端们，由 dealer 连接 reply 服务端们。

因为 req-rep 这种形式，信息传递是双向的，也就是需要同时监听 router 和 dealer，所以代理中需要用 zmq_poll()来实现轮询。也就是要求有消息就要转发，有消息时，不能因为别的 socket 而阻塞。

代理中需要实现：

建立两个 socket，一个 ZMQ_ROUTER 一个 ZMQ_DEALER，并分别都 bind 到各自的网络端口上。

使用 zmq_poll()来实现轮询，如果 router 收到东西就转给 dealer，如果 dealer 收到东西就转给 router

可以看到其实这种模型中，用到了 3 种 socket 之间的组合：

DEALER and ROUTER (代理内部)

REQ and ROUTER (连接 req 的方向)

DEALER and REP (连接 rep 的方向)

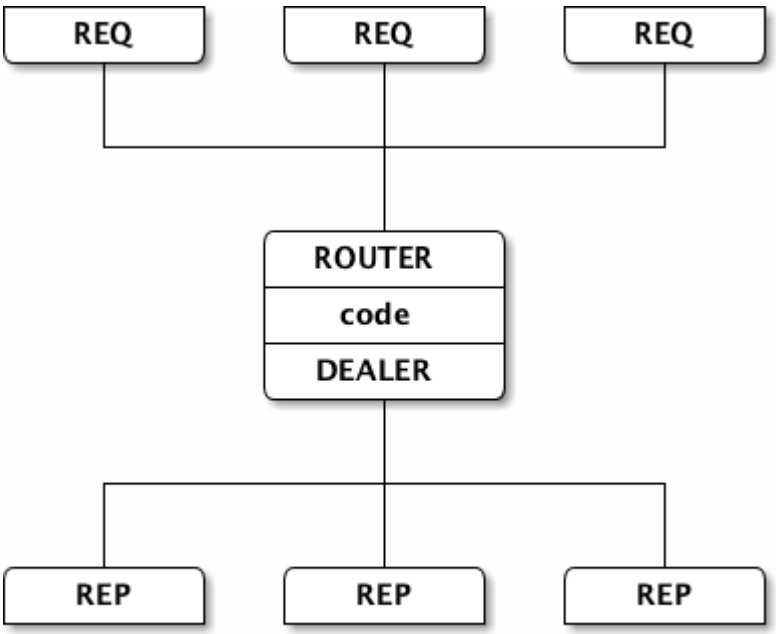

Figure 10 – Extended request reply

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

** rrclient: Request-reply client in C **

```
//
//  Hello world 客户端
//  连接 REQ 套接字至 tcp://localhost:5559 端点
//  发送 Hello 给服务端，等待 World 应答
//
#include "zhelpers.h"

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

    //  用于和服务端通信的套接字
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5559");

    int request_nbr;
    for (request_nbr = 0; request_nbr != 10; request_nbr++) {
        s_send (requester, "Hello");
        char *string = s_recv (requester);
        printf ("收到应答 %d [%s]\n", request_nbr, string);
        free (string);
    }
    zmq_close (requester);
```

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

下面是服务代码：

rrserver: Request-reply service in C

```
//
//  Hello World 服务端
//  连接 REP 套接字至 tcp://*:5560 端点
//  接收 Hello 请求，返回 World 应答
//
#include "zhelpers.h"

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

    //  用于何客户端通信的套接字
    void *responder = zmq_socket (context, ZMQ_REP);
    zmq_connect (responder, "tcp://localhost:5560");

    while (1) {
        //  等待下一个请求
        char *string = s_recv (responder);
        printf ("Received request: [%s]\n", string);
        free (string);

        //  做一些“工作”
        sleep (1);

        //  返回应答信息
        s_send (responder, "World");
    }
    //  程序不会运行到这里，不过还是做好清理工作
    zmq_close (responder);
    zmq_term (context);
    return 0;
}
```

最后是代理程序，可以看到它是能够处理多帧消息的：

rrbroker: Request-reply broker in C

```
//
//  简易请求-应答代理
//
#include "zhelpers.h"
```

```
int main (void)
{
    //  准备上下文和套接字
    void *context = zmq_init (1);
    void *frontend = zmq_socket (context, ZMQ_ROUTER);
    void *backend  = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (frontend, "tcp://*:5559");
    zmq_bind (backend,  "tcp://*:5560");

    //  初始化轮询集合
    zmq_pollitem_t items [] = {
        { frontend, 0, ZMQ_POLLIN, 0 },
        { backend,  0, ZMQ_POLLIN, 0 }
    };
    //  在套接字间转发消息
    while (1) {
        zmq_msg_t message;
        int64_t more;           //  检测多帧消息

        zmq_poll (items, 2, -1);
        if (items [0].revents & ZMQ_POLLIN) {
            while (1) {
                //  处理所有消息帧
                zmq_msg_init (&message);
                zmq_recv (frontend, &message, 0);
                size_t more_size = sizeof (more);
                zmq_getsockopt (frontend, ZMQ_RCVMORE, &more,
&more_size);
                zmq_send (backend, &message, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  最后一帧
            }
        }
        if (items [1].revents & ZMQ_POLLIN) {
            while (1) {
                //  处理所有消息帧
                zmq_msg_init (&message);
                zmq_recv (backend, &message, 0);
                size_t more_size = sizeof (more);
                zmq_getsockopt (backend, ZMQ_RCVMORE, &more,
&more_size);
                zmq_send (frontend, &message, more? ZMQ_SNDMORE: 0);
                zmq_msg_close (&message);
                if (!more)
                    break;      //  最后一帧
            }
        }
    }
    //  程序不会运行到这里，不过还是做好清理工作
    zmq_close (frontend);
    zmq_close (backend);
```

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

zmq 对此有一个内置的装置，专门做 req-rep 代理，即 ZMQ_QUEUE。

```
zmq_device (ZMQ_QUEUE, frontend, backend);
```

4.6.多个线程之间需要做同步，用 zmq 实现（涉及 ZMQ_PAIR）

假设有两个两个线程，叫 step1 和 step2。而主进程为 step3。

从执行顺序来说，必须先执行完 step1，才能做 step2。做完 step2 才能做 step3，也就是有线程间同步的需要。

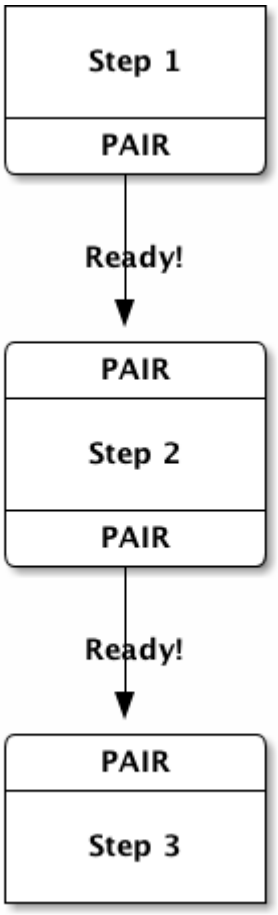

Figure 13 — The Relay Race

我们使用 PAIR 套接字和 inproc 协议。

mtrelay: Multithreaded relay in C

```
//
```

```
// 多线程同步
//
#include "zhelpers.h"
#include <pthread.h>

static void *
step1 (void *context) {
    //  连接至步骤 2，告知我已就绪
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step2");
    printf ("步骤 1 就绪，正在通知步骤 2……\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

static void *
step2 (void *context) {
    //  启动步骤 1 前线绑定至 inproc 套接字
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step2");
    pthread_t thread;
    pthread_create (&thread, NULL, step1, context);

    //  等待信号
    char *string = s_recv (receiver);
    free (string);
    zmq_close (receiver);

    //  连接至步骤 3，告知我已就绪
    void *xmitter = zmq_socket (context, ZMQ_PAIR);
    zmq_connect (xmitter, "inproc://step3");
    printf ("步骤 2 就绪，正在通知步骤 3……\n");
    s_send (xmitter, "READY");
    zmq_close (xmitter);

    return NULL;
}

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

    //  启动步骤 2 前线绑定至 inproc 套接字
    void *receiver = zmq_socket (context, ZMQ_PAIR);
    zmq_bind (receiver, "inproc://step3");
    pthread_t thread;
    pthread_create (&thread, NULL, step2, context);

    //  等待信号
    char *string = s_recv (receiver);
```

```
    free (string);
    zmq_close (receiver);

    printf ("测试成功！\n");
    zmq_term (context);
    return 0;
}
```

代码的基本思想是，由父进程建个 context，然后建个 ZMQ_PAIR 的 socket。

首先每个 step，都建 ZMQ_PAIR 的 socket

做法就是 step1 和 step2 之间用他们之间专有的 inproc 通信来同步（假设名字是 inproc://step2）

在 step2 完成后，再用 step2 和 step3 之间专有的 inproc 通信来同步（假设名字是 inproc://step3）

先执行完 step1，用 inproc://step2 通知 step2。做完 step2 用 inproc://step3 通知做 step3.

在例子中，step3 进程创建了 step2 线程，step2 线程又创建了 step1 线程。

但我感觉在 step3 中一起先后创建 step2 和 step1 线程其实应该也是可以的。只要保证按照同步的反方向来创建线程，最后创建第一个干活的线程就可以了。

在这个例子中，需要解释下为什么要用 ZMQ_PAIR 这种 socket 类型？

1.如果用 ZMQ_PULL/ZMQ_PUSH，因为这个是专用于并发多任务处理的类型，所以当接收方>1 的时候，发送方会自动做负载均衡。如果你不小心开启了两个接收方，就会“丢失”一半的信号。

2.如果用 ZMQ_DEALER/ZMQ_ROUTER，有两个原因。

一是 ROUTER 在接收数据的时候，会在原有的信息上加上个“头”，包含来源地址的信息。如果这个数据被转到 DEALER 发出去，没问题（也就是通常 req-rep 所使用的 dealer-router 代理的情况）。

但如果直接被拿来使用，则会发现，收到的数据比想象中多个“头”。这对于上面例子中只有接收方的主进程来说是个问题。

二是 DEALER 这边，和 PULL/PUSH 一样，有负载均衡的特性，接收方>1，数据就会被均衡。

3.如果用 ZMQ_SUB/ZMQ_PUB，这种情况其实乍一看基本没问题。同步的通知是单向传输的，没啥毛病。唯一的问题是，需要确认 pub-sub 之间建立好连接再发同步通知，要不然会丢消息。

但既然是 inproc 方式的，建立连接总比 tcp 要快吧。所以没啥问题，也行。只是麻烦点，需要定义谁是 sub 谁是 pub，sub 要设置订阅消息的类型。不如 ZMQ_PAIR 方便。

另外，PAIR 不会自动重连（大部分别的 zmq socket 会），所以无法做网络节点间的同步，因为网络节点经常上线下线，需要自动重连。

5. zmq 和 tcp 连接有什么区别？

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

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

连接是异步的，并由一组消息队列做缓冲；

连接会表现出某种消息模式，这是由创建连接的套接字类型决定的；

一个套接字可以有多个输入和输出连接；

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

应用程序无法直接和这些连接打交道，因为它们是被封装在 ZMQ 底层的。

6. 如何定位消息丢失的问题？

参见《zeromq 丢失数据解决办法.png》

7. pub-sub 同步问题：

如果 sub 先准备好。此时 pub 上线，在 zmq_bind 后直接 s_send。因为 zmq_bind 实质是 pub 通过 tcp 和 sub 三次握手。而 zmq 的消息发送/接收又是异步 IO 的，所以可能握手没完成，s_send 的内容就出去了，导致 sub 丢失数据。

又或者 pub 已上线，sub 还没上线，这时候 pub 发消息，sub 再上线，自然就丢失数据。

解决的方法：

1.pub 在 zmq_bind 后睡几秒（差劲的方法）

2.假如 pub 发出的消息无穷无尽，则 sub 丢几个也无妨，比如天气数据。

3.真正等 pub-sub 同步后再通信，也就是 pub 要等所有的 sub 都上线准备好后，再开始发消息，也就是网络节点间需要做同步。（但有限制，就是 pub 需要提前知道总共有多少 sub 会连上来）

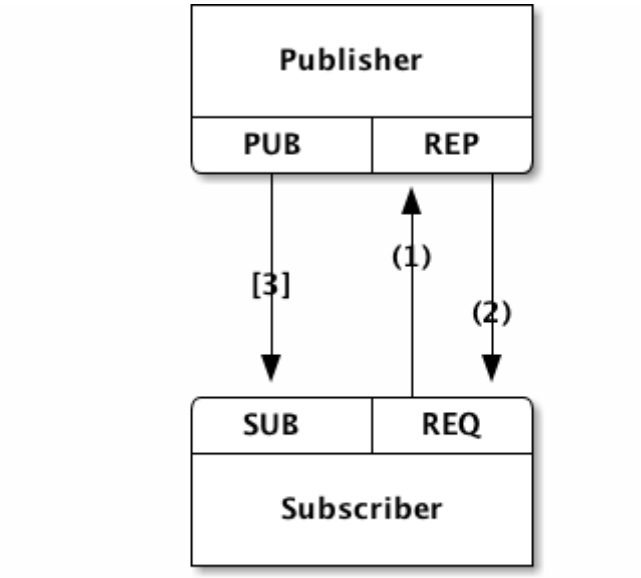

Figure 14 — Pub Sub Synchronization

syncpub: Synchronized publisher in C

```
//
//  发布者 - 同步版
//
#include "zhelpers.h"

//  等待 10 个订阅者连接
#define SUBSCRIBERS_EXPECTED  10

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

    //  用于和客户端通信的套接字
    void *publisher = zmq_socket (context, ZMQ_PUB);
    zmq_bind (publisher, "tcp://*:5561");

    //  用于接收信号的套接字
    void *syncservice = zmq_socket (context, ZMQ_REP);
    zmq_bind (syncservice, "tcp://*:5562");

    //  接收订阅者的就绪信号
    printf ("正在等待订阅者就绪\n");
    int subscribers = 0;
    while (subscribers < SUBSCRIBERS_EXPECTED) {
        //  - 等待就绪信息
        char *string = s_recv (syncservice);
        free (string);
```

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

    s_send (publisher, "END");

    zmq_close (publisher);
    zmq_close (syncservice);
    zmq_term (context);
    return 0;
}
```

以下是订阅者的代码：

syncsub: Synchronized subscriber in C

```
//
//  订阅者 - 同步版
//
#include "zhelpers.h"

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

    //  一、连接 SUB 套接字
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5561");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);

    //  ZMQ 太快了，我们延迟一会儿……
    sleep (1);

    //  二、与发布者进行同步
    void *syncclient = zmq_socket (context, ZMQ_REQ);
    zmq_connect (syncclient, "tcp://localhost:5562");

    //  - 发送请求
    s_send (syncclient, "");

    //  - 等待应答
    char *string = s_recv (syncclient);
    free (string);
```

```
    //  三、处理消息
    int update_nbr = 0;
    while (1) {
        char *string = s_recv (subscriber);
        if (strcmp (string, "END") == 0) {
            free (string);
            break;
        }
        free (string);
        update_nbr++;
    }
    printf ("收到 %d 条消息\n", update_nbr);

    zmq_close (subscriber);
    zmq_close (syncclient);
    zmq_term (context);
    return 0;
}
```

以下这段 shell 脚本会启动 10 个订阅者、1 个发布者：

```
echo "正在启动订阅者..."
for a in 1 2 3 4 5 6 7 8 9 10; do
    syncsub &
done
echo "正在启动发布者..."
syncpub
```

结果如下：

```
正在启动订阅者...
正在启动发布者...
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
收到 1000000 条消息
```

比较推荐的方法是：

发布者打开 PUB 套接字，开始发送 Hello 消息（非数据）；这样做的意义在于，保证发布者在收到 sub 的同步信号之前，就已经处于就绪的状态。防止订阅者过早发布 REQ 同步消息，而发布者没收到。可以发多次 hello。

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

当发布者获得所有订阅者的同步消息后，针对每个订阅者的情况做好统计，等所有订阅者至少发过一次 REQ-REP 套接字同步后，才开始发送真正的数据。

8. 何时用 zmq_bind，何时用 zmq_connect？

首先 zmq 的 bind 和 connect 和原始的 socket 的 bind 和 connect 含义不太一样。原始的 socket 要求先 bind 后 connect，而 zmq 无此要求，且对谁选择 bind 谁选择 connect 也毫不关心

假设某个 zmq socket 先选择了 bind 到 xxx 端口，那后续想连上来的 socket，就只能选择 connect 到这个端口。

总的来说有以下几个考虑，来决定谁应该 bind，谁应该 connect：

1.生命周期更长的 socket 应该选择 bind 而不是 connect。有些 socket（例如 push-pull 模型中那些干活的 worker 进程里的 socket）会中途进来，做完事情就走人，这种短命的就应该用 connect。

2.有多个实例的 socket 应该选择 connect（例如 push-pull 模型中那些干活的 worker 进程里的 socket，有很多个 worker），只有一个实例的 socket 通常应该选择 bind（例如 push-pull 模型中任务分发者和结果汇总者）

3.有较为固定的 ip 地址的服务端程序，适合用 bind，而 ip 地址不固定的客户端程序适合用 connect。

合理的选择谁来 bind，可以最大限度的节约端口的资源，减少维护多个端口的成本。

详见：

《when to use zmq_bind or zmq_connect - Stack Overflow.pdf》

另外多说一句：

在传统的 C/S（Client/Server）模型中，如果先启动客户端，再启动服务端，客户端肯定会先报错的。

但是 zmq 无此限制，客户端先 connect，服务端后来再 bind，这种情况，客户端其实不会报错，connect 不会失败。

zmq 的服务端，可以用同一个 socket 对不同地址多次 bind。（而传统的网络程序，针对一个连接，需要一个 socket，如果有 1000 个连接，就要 1000 个 socket）

同样，zmq 的客户端，可以用同一个 socket 对不同地址多次 connect。

9. 如何公平的读取多个 socket 上的信息？

首先最容易想到的是利用同一个 socket 可以 bind/connect 不同的地址的特定，用一个 socket 去连接不同的地址，让 zmq 内部用其公平队列的机制类收数据。

但这有个限制，就是这些不同的地址，必须是同一种 socket 类型的（例如都是 SUB）

如果有多个不同类型的 socket，怎么办？（同一个进程/线程内）

因为 recv()的时候会阻塞，所以如果顺序的写代码的话（先 recv()第一个 socket，然后再 recv 第二个 socket，如此循环），会导致如果前面的 socket 阻塞住的话，就会不能及时处理后面的 socket。

这时候就应该用 zmq_poll()

基本原理是，zmq_poll()会帮助发现哪个 socket 有数据，并且公平的给予每个 socket 机会。

代码详见《第二章 ZeroMQ 进阶.pdf》

10. 一次性传送多帧 zmq_msg_t 的消息：

发送多帧消息：

```
zmq_send (socket, &message, ZMQ_SNDMORE);
zmq_send (socket, &message, ZMQ_SNDMORE);
zmq_send (socket, &message, 0);                    最后一帧
```

接收多帧消息：

```
while (1) {
        zmq_msg_t message;
        zmq_msg_init (&message);
        zmq_recv (socket, &message, 0);
        // 处理一帧消息
```

```
        zmq_msg_close (&message);
        int64_t more;
        size_t more_size = sizeof (more);
        zmq_getsockopt (socket, ZMQ_RCVMORE, &more, &more_size);
        if (!more)
                break; // 已到达最后一帧
}
```

多帧消息传送的时候，是作为一个整体的，体现在：

在发送多帧消息时，只有当最后一帧提交发送了，整个消息才会被发送；

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

多帧消息是整体传输的，不会只收到一部分；

多帧消息的每一帧都是一个 zmq_msg 结构；

无论你是否检查套接字的 ZMQ_RCVMORE 选项，你都会收到所有的消息；

发送时，ZMQ 会将开始的消息帧缓存在内存中，直到收到最后一帧才会发送；

我们无法在发送了一部分消息后取消发送，只能关闭该套接字。

这种多帧的传输，除了把数据分块以外，还有什么用？

其实 zmq 如果在发送的时候用的 ZMQ_SNDMORE 进行了分块。虽然传输本身是整体的，但是在接收方，ZMQ_RCVMORE 的时候，其实是严格按照发送方的分块方式接收的。

如果发送方发送了 1k 字节+5k 字节，接收方得到的就是 1k 字节+5k 字节。

这种特性有时候可以被用来给消息加个 header。

例如 pub-sub 模型里面，如果 pub 需要发一个带特定 header 的消息，让下面的 sub 们有所鉴别，这是发给其中一个特定的 sub 的，就可以把这个 header 用 ZMQ_SNDMORE 和消息的主题分开。

sub 们接收的时候，就能根据 header 判断是不是给自己的消息。当然 sub 方其实不用 ZMQ_RCVMORE 多次。只要用 zmq_setsockopt 设置下 ZMQ_SUBSCRIBE 项的值，就能起到过滤消息的效果。

但作为 pub 方来说，仍然需要用 ZMQ_SNDMORE，先发送 header 再发送消息体。

这种多帧传输的另外一个用例是：

如果 sub 订阅了多个不同的 pub 的消息，那么都是从 sub 的同一个 socket 收进来的。

怎么区别是哪个 pub 发过来的？这时候就需要 pub 在发送多帧消息的时候，除了刚刚说了的 header，还需要加一个帧，说明发送者是谁。

Figure 18 – Pub sub envelope with sender address

11. 使用 zeromq 做多线程编程有以下这些注意点：

1.要使用 inproc 协议做线程间的通信，必须在父进程中创建 zmq 上下文。这个上下文可以被子线程们用来建立 socket。换句话说 zmq 上下文是线程安全的。

2.zmq socket 并非线程安全，不要在线程之间传递 zmq socket。

12. 一个多线程同时处理 request 的服务端例子程序：

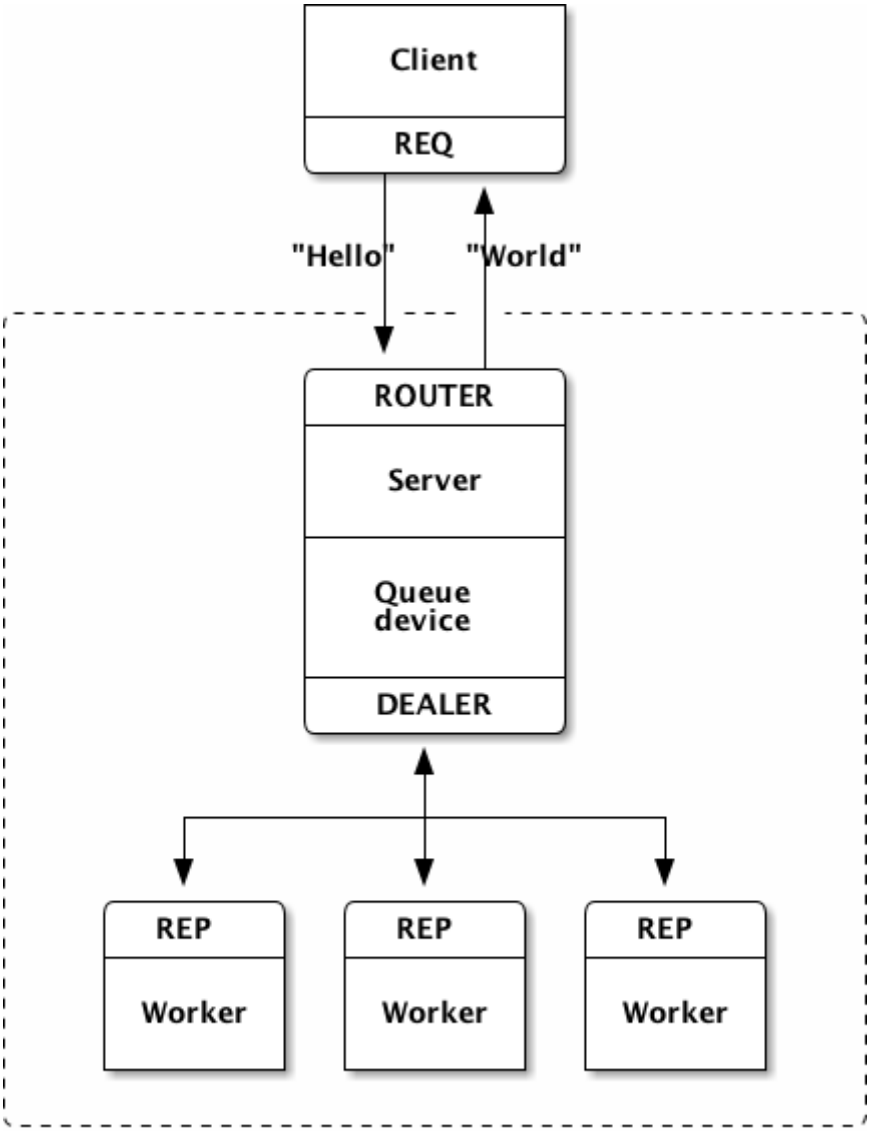

Figure 12 — Multithreaded server

mtserver: Multithreaded service in C

```
//
//  多线程版 Hello World 服务
//
#include "zhelpers.h"
#include <pthread.h>

static void *
worker_routine (void *context) {
    //  连接至代理的套接字
    void *receiver = zmq_socket (context, ZMQ_REP);
    zmq_connect (receiver, "inproc://workers");
```

```
    while (1) {
        char *string = s_recv (receiver);
        printf ("Received request: [%s]\n", string);
        free (string);
        //  工作
        sleep (1);
        //  返回应答
        s_send (receiver, "World");
    }
    zmq_close (receiver);
    return NULL;
}

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

    //  用于和 client 进行通信的套接字
    void *clients = zmq_socket (context, ZMQ_ROUTER);
    zmq_bind (clients, "tcp://*:5555");

    //  用于和 worker 进行通信的套接字
    void *workers = zmq_socket (context, ZMQ_DEALER);
    zmq_bind (workers, "inproc://workers");

    //  启动一个 worker 池
    int thread_nbr;
    for (thread_nbr = 0; thread_nbr < 5; thread_nbr++) {
        pthread_t worker;
        pthread_create (&worker, NULL, worker_routine, context);
    }
    //  启动队列装置
    zmq_device (ZMQ_QUEUE, clients, workers);

    //  程序不会运行到这里，但仍进行清理工作
    zmq_close (clients);
    zmq_close (workers);
    zmq_term (context);
    return 0;
}
```

首先模型上是类似 Figure 10 的，属于有代理的 req-rep 模型

只不过在 Figure 12 里，rep 的节点，换成了一个个线程。

rep 的节点们，用 inproc 方式和代理通信。

数据由 req 的节点们发出，被 router 收到。通过 ZMQ_QUEQUE 装置，传递给 dealer。而 rep 的线程们，通过 inproc 方式连接到 dealer，处理请求并回复。

可以看到，代码中，context 由进程创建。而线程则需要使用进程的 context，来创建自己的 socket。

13. 零拷贝：

zmq 的零拷贝的概念其实和 linux 本身的零拷贝相去甚远

linux 零拷贝追求的是去掉用户空间和内核空间的上下文切换，去掉用户空间和内核空间的内存拷贝动作。

这点 zmq 的零拷贝都不涉及

唯一涉及的是，如果已经有个现成的分配好的空间，那可以直接把这个空间的指针扔给 zmq，而不用 memcpy 到一个新的 zmq_msg_t 再发送。所以能起到的作用也很有限。

要能实现这样的效果，唯一需要做的就是把如何释放现成的分配好的空间的方法，告诉 zmq。

见下面的例子代码：

```
void my_free (void *data, void *hint) {
  free (data);      //注意此处需要把 free 的方法告知 zmq。后续就由 zmq 来 free 了。
}
//  Send message from buffer, which we allocate and 0MQ will free for us
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
zmq_send (socket, &message, 0);
```

14. 持久套接字：

这个概念是针对断线重连而产生的。

假设有一个 pub-sub 结构。这时候其中一个 sub 突然掉线，而 pub 仍只管往外发消息。当掉线的 sub 重新上线后，虽然能自动重连，但掉线期间 pub 发出的数据，sub 都丢失了。

解决的办法是：

针对这个 sub，设置一个 zmq_setsockopt 的 ZMQ_IDENTITY 属性值（例如"John"），pub 端无需任何身份设置。

这样 pub 就会记下来有一个身份是 John 的人连上来过。如果 John 下线，则 pub 会把未能发送给 John 的消息暂时缓存在内存里。

注意，这里的 ZMQ_IDENTITY 必须是唯一的。实际应用中，最好使用类似 uuid 的东西来作为标记，而不是字串。

另外，pub 因为要缓存 sub 未收到的消息，时间长了会导致内存用尽。所以 pub 端必须设置一个阈值。当内存的使用量超过阈值的时候，就开始丢弃消息。

例如：

uint64_t hwm = 2;//意味着 pub 会为每个 sub 保留 2 个未收到的消息。

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

阈值的计算方法需要注意，用到的内存是需要乘以 sub 订阅者的总数的。

例如，每个 zmq_msg_t 是 x 字节，阈值 hwm=y，而订阅者总数=z

那用到的物理内存总数就是：x * y * z。当订阅者很多的时候，这个数量是相当大的。

针对这个情况，还有一个办法就是在硬盘上设置交换区，类似内存的 swap：

uint64_t swap = 25000000;

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

这个办法可以把缓冲区设的很大，缺点是访问速度较慢。