第五章 高级发布-订阅模式

第三章和第四章讲述了ZMQ中请求-应答模式的一些高级用法。如果你已经能够彻底理解了，那我要说声恭喜。这一章我们会关注发布-订阅模式，使用上层模式封装，提升ZMQ发布-订阅模式的性能、可靠性、状态同步及安全机制。

本章涉及的内容有：

- 处理慢订阅者（自杀的蜗牛模式）
- 高速订阅者（黑箱模式）
- 构建一个共享键值缓存（克隆模式）

检测慢订阅者（自杀的蜗牛模式）

在使用发布-订阅模式的时候，最常见的问题之一是如何处理响应较慢的订阅者。理想状况下，发布者能以全速发送消息给订阅者，但现实中，订阅者会需要对消息做较长时间的处理，或者写得不够好，无法跟上发布者的脚步。

如何处理慢订阅者？最好的方法当然是让订阅者高效起来，不过这需要额外的工作。以下是一些处理慢订阅者的方法：

- **在发布者中贮存消息**。这是Gmail的做法，如果过去的几小时里没有阅读邮件的话，它会把邮件保存起来。但在高吞吐量的应用中，发布者堆积消息往往会导致内存溢出，最终崩溃。特别是当同是有多个订阅者时，或者无法用磁盘来做一个缓冲，情况就会变得更为复杂。

- **在订阅者中贮存消息**。这种做法要好的多，其实ZMQ默认的行为就是这样的。如果非得有一个人会因为内存溢出而崩溃，那也只会是订阅者，而非发布者，这挺公平的。然而，这种做法只对瞬间消息量很大的应用才合理，订阅者只是一时处理不过来，但最终会赶上进度。但是，这还是没有解决订阅者速度过慢的问题。

- **暂停发送消息**。这也是Gmail的做法，当我的邮箱容量超过7.554GB时，新的邮件就会被Gmail拒收或丢弃。这种做法对发布者来说很有益，ZMQ中若设置了阈值（HWM），其默认行为也就是这样的。但是，我们仍不能解决慢订阅者的问题，我们只是让消息变得断断续续而已。

- **断开与满订阅者的连接**。这是hotmail的做法，如果连续两周没有登录，它就会断开，这也是为什么我正在使用第十五个hotmail邮箱。不过这种方案在ZMQ里是行不通的，因为对于发布者而言，订阅者是不可见的，无法做相应处理。

看来没有一种经典的方式可以满足我们的需求，所以我们就要进行创新了。我们可以让订阅者自杀，而不仅仅是断开连接。这就是“自杀的蜗牛”模式。当订阅者发现自身运行得过慢时（对于慢速的定义应该是一个配置项，当达到这个标准时就大声地喊出来吧，让程序员知道），它会哀嚎一声，然后自杀。

订阅者如何检测自身速度过慢呢？一种方式是为消息进行编号，并在发布者端设置阈值。当订阅者发现消息编号不连续时，它就知道事情不对劲了。这里的阈值就是订阅者自杀的值。

这种方案有两个问题：一、如果我们连接的多个发布者，我们要如何为消息进行编号呢？解决方法是为每一个发布者设定一个唯一的编号，作为消息编号的一部分。二、如果订阅者使用ZMQ_SUBSRIBE选项对消息进行了过滤，那么我们精心设计的消息编号机制就毫无用处了。

有些情形不会进行消息的过滤，所以消息编号还是行得通的。不过更为普遍的解决方案是，发布者为消息标注时间戳，当订阅者收到消息时会检测这个时间戳，如果其差别达到某一个值，就发出警报并自杀。

当订阅者有自身的客户端或服务协议，需要保证最大延迟时间时，自杀的蜗牛模式会很合适。撤销一个订阅者也许并不是最周全的方案，但至少不会引发后续的问题。如果订阅者收到了过时的消息，那可能会对数据造成进一步的破坏，而且很难被发现。

以下是自杀的蜗牛模式的最简实现：

suisnail: Suicidal Snail in C

```c
//
//  自杀的蜗牛模式
//
#include "czmq.h"

//  ---------------------------------------------------------------------
//  该订阅者会连接至发布者，接收所有的消息，
//  运行过程中它会暂停一会儿，模拟复杂的运算过程，
//  当发现收到的消息超过1秒的延迟时，就自杀。

#define MAX_ALLOWED_DELAY   1000    //  毫秒

static void
subscriber (void *args, zctx_t *ctx, void *pipe)
{
    //  订阅所有消息
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5556");

    //  获取并处理消息
    while (1) {
        char *string = zstr_recv (subscriber);
        int64_t clock;
        int terms = sscanf (string, "%" PRId64, &clock);
        assert (terms == 1);
        free (string);

        //  自杀逻辑
        if (zclock_time () - clock > MAX_ALLOWED_DELAY) {
            fprintf (stderr, "E: 订阅者无法跟进, 取消中\n");
            break;
        }
```

```
        //  工作一定时间
        zclock_sleep (1 + randof (2));
    }
    zstr_send (pipe, "订阅者中止");
}


//  ---------------------------------------------------------------------
//  发布者每毫秒发送一条用时间戳标记的消息

static void
publisher (void *args, zctx_t *ctx, void *pipe)
{
    //  准备发布者
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");

    while (1) {
        //  发送当前时间（毫秒）给订阅者
        char string [20];
        sprintf (string, "%" PRId64, zclock_time ());
        zstr_send (publisher, string);
        char *signal = zstr_recv_nowait (pipe);
        if (signal) {
            free (signal);
            break;
        }
        zclock_sleep (1);            //  等待1毫秒
    }
}


//  下面的代码会启动一个订阅者和一个发布者，当订阅者死亡时停止运行
//
int main (void)
{
    zctx_t *ctx = zctx_new ();
    void *pubpipe = zthread_fork (ctx, publisher, NULL);
    void *subpipe = zthread_fork (ctx, subscriber, NULL);
    free (zstr_recv (subpipe));
    zstr_send (pubpipe, "break");
    zclock_sleep (100);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 示例程序中的消息包含了系统当前的时间戳（毫秒）。在现实应用中，你应该使用时间戳作为消息头，并提供消息内容。
- 示例程序中的发布者和订阅者是同一个进程的两个线程。在现实应用中，他们应该是两个不同的进程。示例中这么做只是为了演示的方便

高速订阅者（黑箱模式）

发布-订阅模式的一个典型应用场景是大规模分布式数据处理。如要处理从证券市场上收集到的数据，可以在证券交易系统上设置一个发布者，获取价格信息，并发送给一组订阅者。如果我们有很多订阅者，我们可以使用TCP。如果订阅者到达一定的量，那我们就应该使用可靠的广播协议，如pgm。

假设我们的发布者每秒产生10万条100个字节的消息。在剔除了不需要的市场信息后，这个比率还是比较合理的。现在我们需要记录一天的数据（8小时约有250GB），再将其传入一个模拟网络，即一组订阅者。虽然10万条数据对ZMQ来说很容易处理，但我们需要更高的速度。

假设我们有多台机器，一台做发布者，其他的做订阅者。这些机器都是8核的，发布者那台有12核。

在我们开始发布消息时，有两点需要注意：

1. 即便只是处理很少的数据，订阅者仍有可能跟不上发布者的速度；
2. 当处理到6M/s的数据量时，发布者和订阅者都有可能达到极限。

首先，我们需要将订阅者设计为一种多线程的处理程序，这样我们就能在一个线程中读取消息，使用其他线程来处理消息。一般来说，我们对每种消息的处理方式都是不同的。这样一来，订阅者可以对收到的消息进行一次过滤，如根据头信息来判别。当消息满足某些条件，订阅者会将消息交给worker处理。用ZMQ的语言来说，订阅者会将消息转发给worker来处理。

这样一来，订阅者看上去就像是一个队列装置，我们可以用各种方式去连接队列装置和worker。如我们建立单向的通信，每个worker都是相同的，可以使用PUSH和PULL套接字，分发的工作就交给ZMQ吧。这是最简单也是最快速的方式：

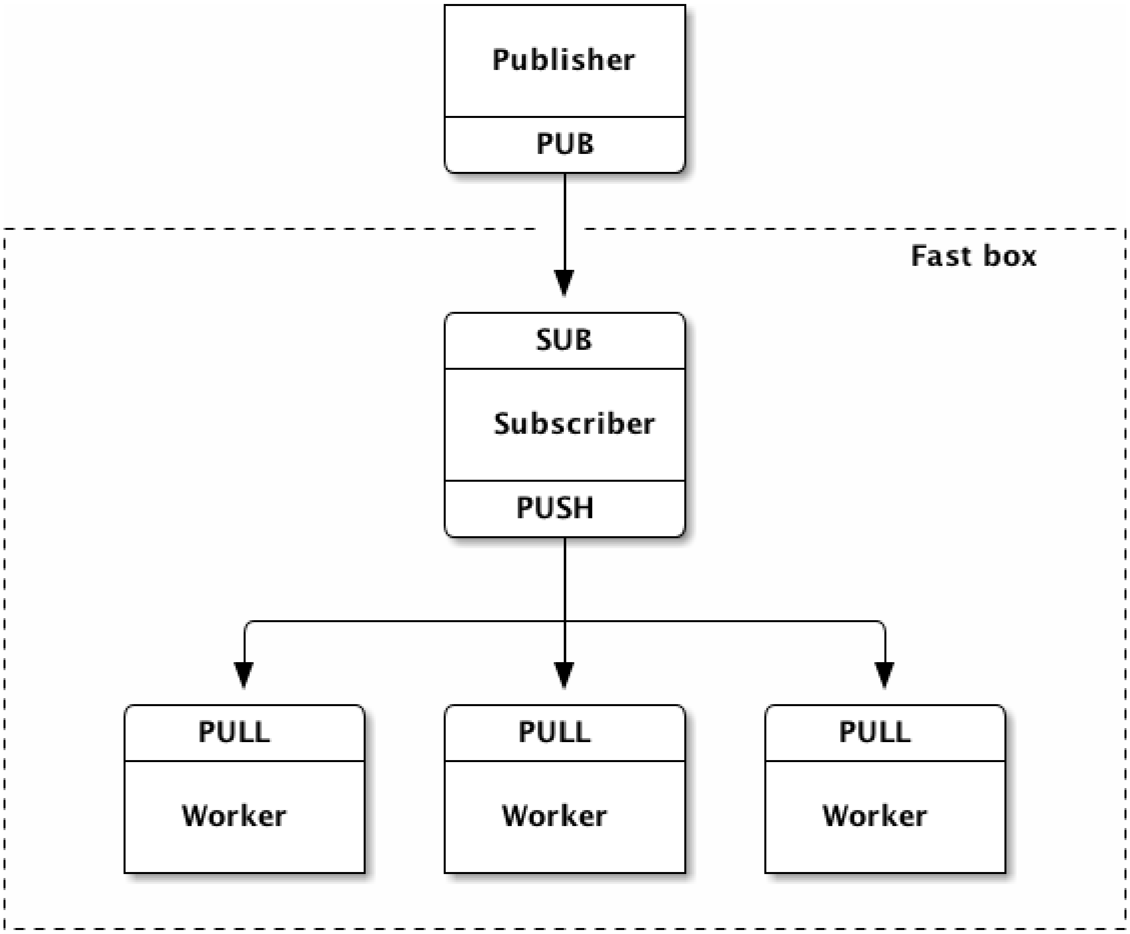

Figure 1 — Simple Black Box Pattern

订阅者和发布者之间的通信使用TCP或PGM协议，订阅者和worker的通信由于是在同一个进程中完成的，所以使用inproc协议。

下面我们看看如何突破瓶颈。由于订阅者是单线程的，当它的CPU占用率达到100%时，它无法使用其他的核心。单线程程序总是会遇到瓶颈的，不管是2M、6M还是更多。我们需要将工作量分配到不同的线程中去，并发地执行。

很多高性能产品使用的方案是分片，就是将工作量拆分成独立并行的流。如，一半的专题数据由一个流媒体传输，另一半由另一个流媒体传输。我们可以建立更多的流媒体，但如果CPU核心数不变，那就没有必要了。
让我们看看如何将工作量分片为两个流：

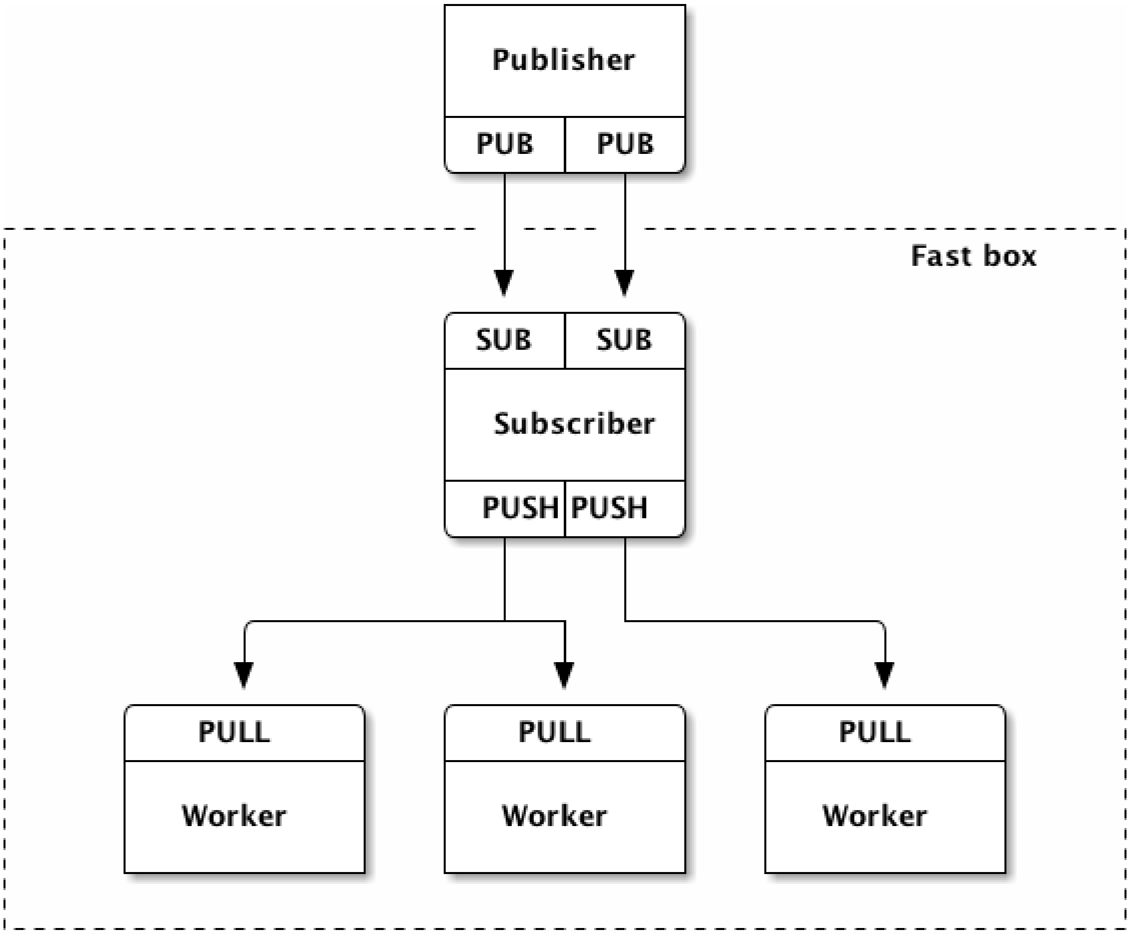

Figure 2 — Mad Black Box Pattern

要让两个流全速工作，需要这样配置ZMQ：

- 使用两个I/O线程，而不是一个；
- 使用两个独立的网络接口；
- 每个I/O线程绑定至一个网络接口；
- 两个订阅者线程，分别绑定至一个核心；
- 使用两个SUB套接字；
- 剩余的核心供worker使用；
- worker线程同时绑定至两个订阅者线程的PUSH套接字。

创建的线程数量应和CPU核心数一致，如果我们建立的线程数量超过核心数，那其处理速度只会减少。另外，开放多个I/O线程也是没有必要的。

共享键值缓存（克隆模式）

发布-订阅模式和无线电广播有些类似，在你收听之前发送的消息你将无从得知，收到消息的多少又会取决于你的接收能力。让人吃惊的是，对于那些追求完美的工程师来说，这种机器恰恰符合他们的需求，且广为传播，成为现实生活中分发消息的最佳机制。想想非死不可、推特、BBS新闻、体育新闻等应用就知道了。

但是，在很多情形下，可靠的发布-订阅模式同样是有价值的。正如我们讨论请求-应答模式一样，我们会根据“故障”来定义“可靠性”，下面几项便是发布-订阅模式中可能发生的故障：

- 订阅者连接太慢，因此没有收到发布者最初发送的消息；

- 订阅者速度太慢，同样会丢失消息；
- 订阅者可能会断开，其间的消息也会丢失。

还有一些情况我们碰到的比较少，但不是没有：

- 订阅者崩溃、重启，从而丢失了所有已收到的消息；
- 订阅者处理消息的速度过慢，导致消息在队列中堆砌并溢出；
- 因网络过载而丢失消息（特别是PGM协议下的连接）；
- 网速过慢，消息在发布者处溢出，从而崩溃。

其实还会有其他出错的情况，只是以上这些在现实应用中是比较典型的。

我们已经有方法解决上面的某些问题了，比如对于慢速订阅者可以使用自杀的蜗牛模式。但是，对于其他的问题，我们最后能有一个可复用的框架来编写可靠的发布-订阅模式。

难点在于，我们并不知道目标应用程序会怎样处理这些数据。它们会进行过滤、只处理一部分消息吗？它们是否会将消息记录起来供日后使用？它们是否会将消息转发给其下的worker进行处理？需要考虑的情况实在太多了，每种情况都有其所谓的可靠性。

所以，我们将问题抽象出来，供多种应用程序使用。这种抽象应用我们称之为共享的键值缓存，它的功能是通过唯一的键名存储二进制数据块。

不要将这个抽象应用和分布式哈希表混淆起来，它是用来解决节点在分布式网络中相连接的问题的；也不要和分布式键值表混淆，它更像是一个NoSQL数据库。我们要建立的应用是将内存中的状态可靠地传递给一组客户端，它要做到的是：

- 客户端可以随时加入网络，并获得服务端当前的状态；
- 任何客户端都可以改变键值缓存（插入、更新、删除）；
- 将这种变化以最短的延迟可靠地传达给所有的客户端；
- 能够处理大量的客户端，成百上千。

克隆模式的要点在于客户端会反过来和服务端进行通信，这在简单的发布-订阅模式中并不常见。所以我这里使用“服务端”、“客户端”而不是“发布者”、“订阅者”这两个词。我们会使用发布-订阅模式作为核心消息模式，不过还需要夹杂其他模式。

分发键值更新事件

我们会分阶段实施克隆模式。首先，我们看看如何从服务器发送键值更新事件给所有的客户端。我们将第一章中使用的天气服务模型进行改造，以键值对的方式发送信息，并让客户端使用哈希表来保存：

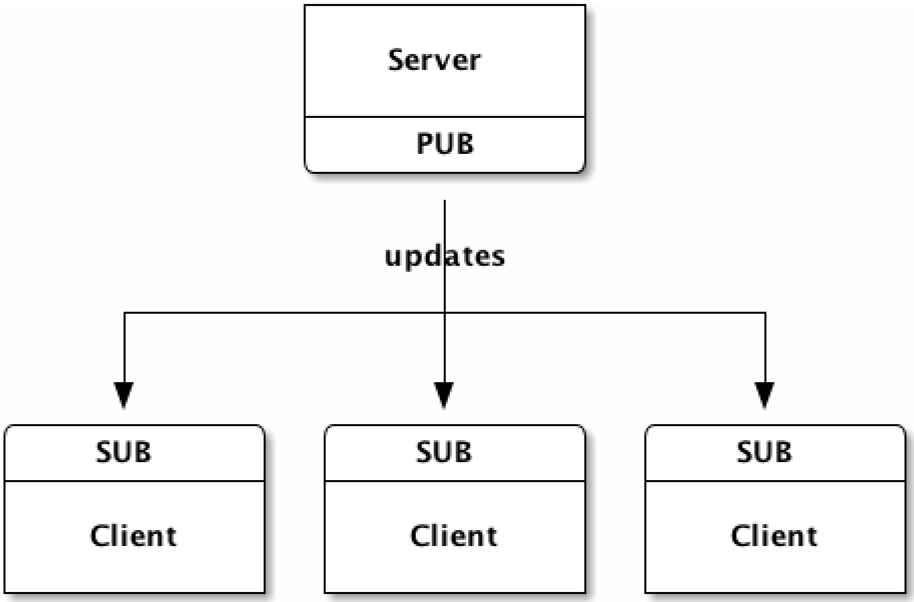

Figure 3 — Simplest Clone Model

以下是服务端代码：

clonesrv1: Clone server, Model One in C

```
//
//  克隆模式服务端模型1
//

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

int main (void)
{
    //  准备上下文和PUB套接字
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5556");
    zclock_sleep (200);

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    while (!zctx_interrupted) {
        //  使用键值对分发消息
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
```

```
        kvmsg_store   (&kvmsg, kvmap);
    }
    printf (" 已中止\n已发送 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

以下是客户端代码：

clonecli1: Clone client, Model One in C

```
//
//  克隆模式客户端模型1
//

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

int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *updates = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (updates, "tcp://localhost:5556");

    zhash_t *kvmap = zhash_new ();
    int64_t sequence = 0;

    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (updates);
        if (!kvmsg)
            break;          //  中断
        kvmsg_store (&kvmsg, kvmap);
        sequence++;
    }
    printf (" 已中断\n收到 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 所有复杂的工作都在kvmsg类中完成了，这个类能够处理键值对类型的消息对象，其实质上是一个ZMQ多帧消息，共有三帧：键（ZMQ字符串）、编号（64位，按字节顺序排列）、二进制体（保存所有附加

信息）。

- 服务端随机生成消息，使用四位数作为键，这样可以模拟大量而不是过量的哈希表（1万个条目）。
- 服务端绑定套接字后会等待200毫秒，以避免订阅者连接延迟而丢失数据的问题。我们会在后面的模型中解决这一点。
- 我们使用“发布者”和“订阅者”来命名程序中使用的套接字，这样可以避免和后续模型中的其他套接字发生混淆。

以下是kvmsg的代码，已经经过了精简：

kvsimple: Key-value message class in C

```
/*  =====================================================================
    kvsimple - simple key-value message class for example applications

    ---------------------------------------------------------------------
    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 "kvsimple.h"
#include "zlist.h"

//  键是一个短字符串
#define KVMSG_KEY_MAX   255

//  消息被格式化成三帧
//  frame 0: 键（ZMQ字符串）
//  frame 1: 编号（8个字节，按顺序排列）
//  frame 2: 内容（二进制数据块）
#define FRAME_KEY       0
```

```
#define FRAME_SEQ       1
#define FRAME_BODY      2
#define KVMSG_FRAMES    3

//  类结构
struct _kvmsg {
    //  消息中某帧是否存在
    int present [KVMSG_FRAMES];
    //  对应的ZMQ消息帧
    zmq_msg_t frame [KVMSG_FRAMES];
    //  将键转换为C语言字符串
    char key [KVMSG_KEY_MAX + 1];
};


//  ---------------------------------------------------------------------
//  构造函数，设置编号

kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    kvmsg_set_sequence (self, sequence);
    return self;
}


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

//  释放消息中的帧，可供zhash_freefn()函数调用
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  销毁消息中的帧
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  释放对象本身
```

```
        free (self);
    }
}

void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  从套接字中读取键值消息，返回kvmsg实例

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  读取所有帧，出错则销毁对象
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_recvmsg (socket, &self->frame [frame_nbr], 0) == -1) {
            kvmsg_destroy (&self);
            break;
        }
        //  验证多帧消息
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsockopt_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    return self;
}
```

```
//  --------------------------------------------------------------------
//  向套接字发送键值对消息，不检验消息帧的内容

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_sendmsg (socket, &copy,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}


//  --------------------------------------------------------------------
//  从消息中获取键值，不存在则返回NULL

char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}


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

```c
//  返回消息的编号

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  返回消息内容，不存在则返回NULL

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息内容的大小

size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
```

```
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  设置消息的键

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}


//  ---------------------------------------------------------------------
//  设置消息的编号

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}
```

```c
// ---------------------------------------------------------------------
// 设置消息内容

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}


// ---------------------------------------------------------------------
// 使用printf()格式设置消息键

void
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}


// ---------------------------------------------------------------------
// 使用springf()格式设置消息内容

void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
```

```
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}


//  ---------------------------------------------------------------------
//  若kvmsg结构的键值均存在，则存入哈希表；
//  如果kvmsg结构已没有引用，则自动销毁和释放。

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (self->present [FRAME_KEY]
        &&  self->present [FRAME_BODY]) {
            zhash_update (hash, kvmsg_key (self), self);
            zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
        }
        *self_p = NULL;
    }
}


//  ---------------------------------------------------------------------
//  将消息内容打印至标准错误输出，用以调试和跟踪

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
```

```
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}


//  ---------------------------------------------------------------------
//  测试用例

int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");

    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *output = zsocket_new (ctx, ZMQ_DEALER);
    int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);
    void *input = zsocket_new (ctx, ZMQ_DEALER);
    rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
    assert (rc == 0);

    zhash_t *kvmap = zhash_new ();

    //  测试简单消息的发送和接受
    kvmsg = kvmsg_new (1);
    kvmsg_set_key  (kvmsg, "key");
    kvmsg_set_body (kvmsg, (byte *) "body", 4);
    if (verbose)
        kvmsg_dump (kvmsg);
    kvmsg_send (kvmsg, output);
    kvmsg_store (&kvmsg, kvmap);

    kvmsg = kvmsg_recv (input);
    if (verbose)
        kvmsg_dump (kvmsg);
    assert (streq (kvmsg_key (kvmsg), "key"));
    kvmsg_store (&kvmsg, kvmap);

    //  关闭并销毁所有对象
    zhash_destroy (&kvmap);
```

```
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

我们会在下文编写一个更为完整的kvmsg类，可以用到现实环境中。

客户端和服务端都会维护一个哈希表，但这个模型需要所有的客户端都比服务端启动得早，而且不能崩溃，这显然不能满足可靠性的要求。

创建快照

为了让后续连接的（或从故障中恢复的）客户端能够获取服务器上的状态信息，需要让它在连接时获取一份快照。正如我们将“消息”的概念简化为“已编号的键值对”，我们也可以将“状态”简化为“一个哈希表”。为获取服务端状态，客户端会打开一个REQ套接字进行请求：

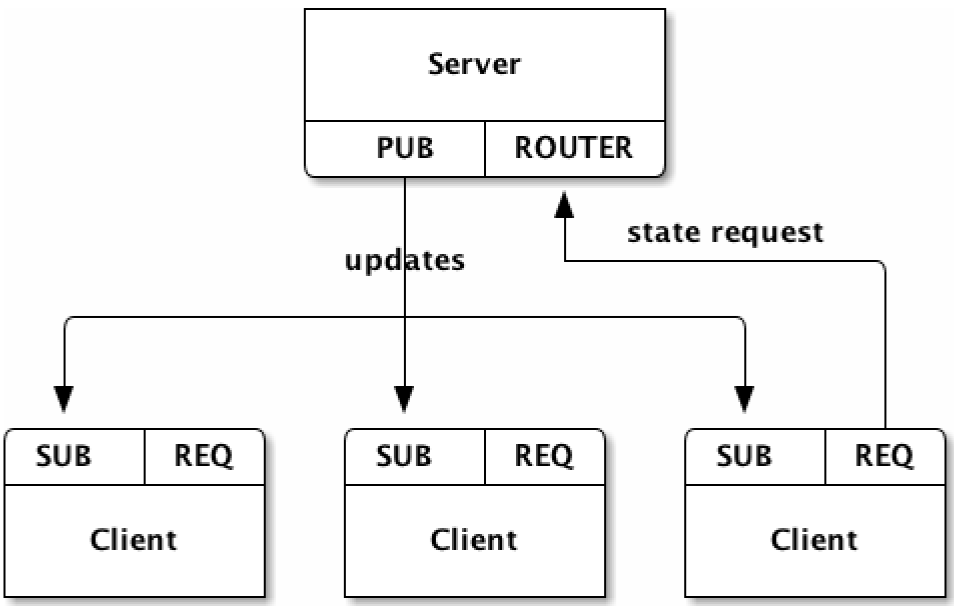

Figure 4 — State Replication

我们需要考虑时间的问题，因为生成快照是需要一定时间的，我们需要知道应从哪个更新事件开始更新快照，服务端是不知道何时有更新事件的。一种方法是先开始订阅消息，收到第一个消息之后向服务端请求“将该条更新之前的所有内容发送给”。这样一来，服务器需要为每一次更新保存一份快照，这显然是不现实的。

所以，我们会在客户端用以下方式进行同步：

- 客户端开始订阅服务器的更新事件，然后请求一份快照。这样就能保证这份快照是在上一次更新事件之后产生的。
- 客户端开始等待服务器的快照，并将更新事件保存在队列中，做法很简单，不要从套接字中读取消息就可以了，ZMQ会自动将这些消息保存起来，这时不应设置阈值（HWM）。

- 当客户端获取到快照后，它将再次开始读取更新事件，但是需要丢弃那些早于快照生成时间的事件。如快照生成时包含了200次更新，那客户端会从第201次更新开始读取。

- 随后，客户端就会用更新事件去更新自身的状态了。

这是一个比较简单的模型，因为它用到了ZMQ消息队列的机制。服务端代码如下：

clonesrv2: Clone server, Model Two in C

```
//
//  克隆模式 - 服务端 - 模型2
//

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

static int s_send_single (char *key, void *data, void *args);
static void state_manager (void *args, zctx_t *ctx, void *pipe);

int main (void)
{
    //  准备套接字和上下文
    zctx_t *ctx = zctx_new ();
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");

    int64_t sequence = 0;
    srandom ((unsigned) time (NULL));

    //  开启状态管理器，并等待同步信号
    void *updates = zthread_fork (ctx, state_manager, NULL);
    free (zstr_recv (updates));

    while (!zctx_interrupted) {
        //  分发键值消息
        kvmsg_t *kvmsg = kvmsg_new (++sequence);
        kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
        kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
        kvmsg_send     (kvmsg, publisher);
        kvmsg_send     (kvmsg, updates);
        kvmsg_destroy (&kvmsg);
    }
    printf (" 已中断\n已发送 %d 条消息\n", (int) sequence);
    zctx_destroy (&ctx);
    return 0;
```

```
}

//  快照请求方信息
typedef struct {
    void *socket;           //  用于发送快照的ROUTER套接字
    zframe_t *identity;     //  请求方的标识
} kvroute_t;

//  发送快照中单个键值对
//  使用kvmsg对象作为载体
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  先发送接收方标识
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}

//  该线程维护服务端状态，并处理快照请求。
//
static void
state_manager (void *args, zctx_t *ctx, void *pipe)
{
    zhash_t *kvmap = zhash_new ();

    zstr_send (pipe, "READY");
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");

    zmq_pollitem_t items [] = {
        { pipe, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    int64_t sequence = 0;       //  当前快照版本
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, -1);
        if (rc == -1 && errno == ETERM)
            break;              //  上下文异常

        //  等待主线程的更新事件
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (pipe);
```

```c
            if (!kvmsg)
                break;          //  中断
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        //  执行快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  中断

            //  请求内容在第二帧中
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: 错误的请求，程序中止\n");
                break;
            }
            //  发送快照给客户端
            kvroute_t routing = { snapshot, identity };

            //  逐项发送
            zhash_foreach (kvmap, s_send_single, &routing);

            //  发送结束标识，内含快照版本号
            printf ("正在发送快照，版本号 %d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    zhash_destroy (&kvmap);
}
```

以下是客户端代码：

clonecli2: Clone client, Model Two in C

```c
//
// 克隆模式 - 客户端 - 模型2
//
```

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

int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");

    zhash_t *kvmap = zhash_new ();

    //  获取快照
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  中断
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("已获取快照，版本号=%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  完成
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    //  应用队列中的更新事件，丢弃过时事件
    while (!zctx_interrupted) {
        kvmsg_t *kvmsg = kvmsg_recv (subscriber);
        if (!kvmsg)
            break;          //  中断
        if (kvmsg_sequence (kvmsg) > sequence) {
            sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, kvmap);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 客户端使用两个线程，一个用来生成随机的更新事件，另一个用来管理状态。两者之间使用PAIR套接字通信。可能你会考虑使用SUB套接字，但是“慢连接”的问题会影响到程序运行。PAIR套接字会让两个线程严格同步的。

- 我们在updates套接字上设置了阈值（HWM），避免更新服务内存溢出。在inproc协议的连接中，阈值是两端套接字阈值的加和，所以要分别设置。

- 客户端比较简单，用C语言编写，大约60行代码。大多数工作都在kvmsg类中完成了，不过总的来说，克隆模式实现起来还是比较简单的。

- 我们没有用特别的方式来序列化状态内容。键值对用kvmsg对象表示，保存在一个哈希表中。在不同的时间请求状态时会得到不同的快照。

- 我们假设客户端只和一个服务进行通信，而且服务必须是正常运行的。我们暂不考虑如何从服务崩溃的情形中恢复过来。

现在，这两段程序都还没有真正地工作起来，但已经能够正确地同步状态了。这是一个多种消息模式的混合体：进程内的PAIR、发布-订阅、ROUTER-DEALER等。

重发键值更新事件

第二个模型中，键值更新事件都来自于服务器，构成了一个中心化的模型。但是我们需要的是一个能够在客户端进行更新的缓存，并能同步到其他客户端中。这时，服务端只是一个无状态的中间件，带来的好处有：

- 我们不用太过关心服务端的可靠性，因为即使它崩溃了，我们仍能从客户端中获取完整的数据。
- 我们可以使用键值缓存在动态节点之间分享数据。

客户端的键值更新事件会通过PUSH-PULL套接字传达给服务端：

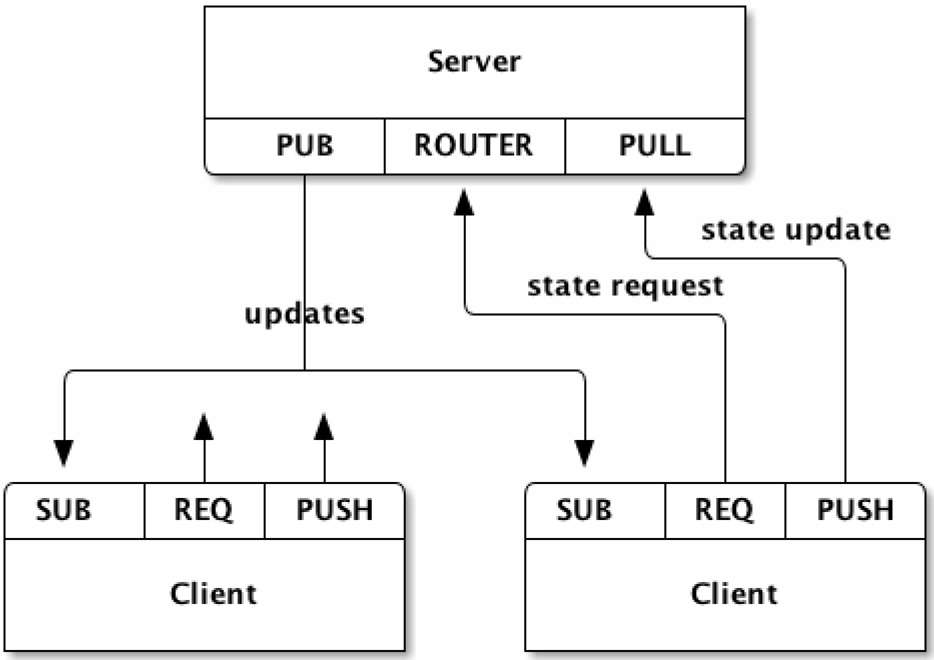

Figure 5 — Republishing Updates

我们为什么不让客户端直接将更新信息发送给其他客户端呢？虽然这样做可以减少延迟，但是就无法为更新事件添加自增的唯一编号了。很多应用程序都需要更新事件以某种方式排序，只有将消息发给服务端，由服务端分发更新消息，才能保证更新事件的顺序。

有了唯一的编号后，客户端还能检测到更多的故障：网络堵塞或队列溢出。如果客户端发现消息输入流有一段空白，它能采取措施。可能你会觉得此时让客户端通知服务端，让它重新发送丢失的信息，可以解决问题。但事实上没有必要这么做。消息流的空挡表示网络状况不好，如果再进行这样的请求，只会让事情变得更糟。所以一般的做法是由客户端发出警告，并停止运行，等到有专人来维护后再继续工作。
我们开始创建在客户端进行状态更新的模型。以下是客户端代码：

clonesrv3: Clone server, Model Three in C

```
//
//  克隆模式 服务端 模型3
//

//  直接编译，不创建类库
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);

//  快照请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求方标识
} kvroute_t;
```

```c
int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  执行来自客户端的更新事件
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  中断
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
            printf ("I: 发布更新事件 %5d\n", (int) sequence);
        }
        //  响应快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  中断

            //  请求内容在消息的第二帧中
            char *request = zstr_recv (snapshot);
            if (streq (request, "ICANHAZ?"))
                free (request);
            else {
                printf ("E: 错误的请求，程序中止\n");
                break;
            }
            //  发送快照
```

```
            kvroute_t routing = { snapshot, identity };

            //  逐条发送
            zhash_foreach (kvmap, s_send_single, &routing);

            //  发送结束标识和编号
            printf ("I: 正在发送快照，版本号：%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) "", 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
        }
    }
    printf (" 已中断\n已处理 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}

//  发送一条键值对状态给套接字，使用kvmsg对象保存键值对
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    //  Send identity of recipient first
    zframe_send (&kvroute->identity,
        kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    kvmsg_send (kvmsg, kvroute->socket);
    return 0;
}
```

以下是客户端代码：

clonecli3: Clone client, Model Three in C

```
//
//  克隆模式 - 客户端 - 模型3
//

//  直接编译，不创建类库
#include "kvsimple.c"
```

```
int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  获取状态快照
    int64_t sequence = 0;
    zstr_send (snapshot, "ICANHAZ?");
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  中断
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: 已收到快照，版本号：%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  完成
        }
        kvmsg_store (&kvmsg, kvmap);
    }
    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 } };
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
            tickless = 0;
        int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文被关闭

        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  中断

            //  丢弃过时消息，包括心跳
            if (kvmsg_sequence (kvmsg) > sequence) {
```

```
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: 收到更新事件：%d\n", (int) sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
        //  创建一个随机的更新事件
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%d", randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
    }
    printf (" 已准备\n收到 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

几点说明：

- 服务端整合为一个线程，负责收集来自客户端的更新事件并转发给其他客户端。它使用PULL套接字获取更新事件，ROUTER套接字处理快照请求，以及PUB套接字发布更新事件。

- 客户端会每隔1秒左右发送随机的更新事件给服务端，现实中这一动作由应用程序触发。

子树克隆

现实中的键值缓存会越变越多，而客户端可能只会需要部分缓存。我们可以使用子树的方式来实现：客户端在发送快照请求时告诉服务端它需要的子树，在订阅更新事件时也指明子树。

关于子树的语法有很多，一种是“分层路径”结构，另一种是“主题树”：

- 分层路径：/some/list/of/paths
    - 主题树：some.list.of.topics

这里我们会使用分层路径结构，以此扩展服务端和客户端，进行子树操作。维护多个子树其实并不太困难，因此我们不在这里演示。

下面是服务端代码，由模型3衍化而来：

clonesrv4: Clone server, Model Four in C

```c
//
//  克隆模式 服务端 模型4
//

//  直接编译，不创建类库
#include "kvsimple.c"

static int s_send_single (char *key, void *data, void *args);

//  快照请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求方标识
    char *subtree;          //  指定的子树
} kvroute_t;


int main (void)
{
    //  准备上下文和套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_ROUTER);
    zsocket_bind (snapshot, "tcp://*:5556");
    void *publisher = zsocket_new (ctx, ZMQ_PUB);
    zsocket_bind (publisher, "tcp://*:5557");
    void *collector = zsocket_new (ctx, ZMQ_PULL);
    zsocket_bind (collector, "tcp://*:5558");

    int64_t sequence = 0;
    zhash_t *kvmap = zhash_new ();

    zmq_pollitem_t items [] = {
        { collector, 0, ZMQ_POLLIN, 0 },
        { snapshot, 0, ZMQ_POLLIN, 0 }
    };
    while (!zctx_interrupted) {
        int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);

        //  执行来自客户端的更新事件
        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (collector);
            if (!kvmsg)
                break;          //  Interrupted
            kvmsg_set_sequence (kvmsg, ++sequence);
            kvmsg_send (kvmsg, publisher);
            kvmsg_store (&kvmsg, kvmap);
```

```
            printf ("I: 发布更新事件 %5d\n", (int) sequence);
        }
        //  响应快照请求
        if (items [1].revents & ZMQ_POLLIN) {
            zframe_t *identity = zframe_recv (snapshot);
            if (!identity)
                break;          //  Interrupted

            //  请求内容在消息的第二帧中
            char *request = zstr_recv (snapshot);
            char *subtree = NULL;
            if (streq (request, "ICANHAZ?")) {
                free (request);
                subtree = zstr_recv (snapshot);
            }
            else {
                printf ("E: 错误的请求，程序中止\n");
                break;
            }
            //  发送快照
            kvroute_t routing = { snapshot, identity, subtree };

            //  逐条发送
            zhash_foreach (kvmap, s_send_single, &routing);

            //  发送结束标识和编号
            printf ("I: 正在发送快照，版本号：%d\n", (int) sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    printf (" 已中断\n已处理 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    return 0;
}

//  发送一条键值对状态给套接字，使用kvmsg对象保存键值对
static int
s_send_single (char *key, void *data, void *args)
```

```
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  先发送接收方的标识
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}
```

下面是客户端代码：

clonecli4: Clone client, Model Four in C

```
//
//  克隆模式 - 客户端 - 模型4
//

//  直接编译，不创建类库
#include "kvsimple.c"

#define SUBTREE "/client/"

int main (void)
{
    //  准备上下文和SUB套接字
    zctx_t *ctx = zctx_new ();
    void *snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (snapshot, "tcp://localhost:5556");
    void *subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (subscriber, "tcp://localhost:5557");
    zsockopt_set_subscribe (subscriber, SUBTREE);
    void *publisher = zsocket_new (ctx, ZMQ_PUSH);
    zsocket_connect (publisher, "tcp://localhost:5558");

    zhash_t *kvmap = zhash_new ();
    srandom ((unsigned) time (NULL));

    //  获取状态快照
    int64_t sequence = 0;
    zstr_sendm (snapshot, "ICANHAZ?");
    zstr_send  (snapshot, SUBTREE);
```

```c
    while (TRUE) {
        kvmsg_t *kvmsg = kvmsg_recv (snapshot);
        if (!kvmsg)
            break;          //  Interrupted
        if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
            sequence = kvmsg_sequence (kvmsg);
            printf ("I: 已收到快照，版本号：%d\n", (int) sequence);
            kvmsg_destroy (&kvmsg);
            break;          //  Done
        }
        kvmsg_store (&kvmsg, kvmap);
    }

    int64_t alarm = zclock_time () + 1000;
    while (!zctx_interrupted) {
        zmq_pollitem_t items [] = { { subscriber, 0, ZMQ_POLLIN, 0 } };
        int tickless = (int) ((alarm - zclock_time ()));
        if (tickless < 0)
            tickless = 0;
        int rc = zmq_poll (items, 1, tickless * ZMQ_POLL_MSEC);
        if (rc == -1)
            break;              //  上下文被关闭

        if (items [0].revents & ZMQ_POLLIN) {
            kvmsg_t *kvmsg = kvmsg_recv (subscriber);
            if (!kvmsg)
                break;          //  中断

            //  丢弃过时消息，包括心跳
            if (kvmsg_sequence (kvmsg) > sequence) {
                sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, kvmap);
                printf ("I: 收到更新事件：%d\n", (int) sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
        //  创建一个随机的更新事件
        if (zclock_time () >= alarm) {
            kvmsg_t *kvmsg = kvmsg_new (0);
            kvmsg_fmt_key  (kvmsg, "%s%d", SUBTREE, randof (10000));
            kvmsg_fmt_body (kvmsg, "%d", randof (1000000));
            kvmsg_send     (kvmsg, publisher);
            kvmsg_destroy (&kvmsg);
            alarm = zclock_time () + 1000;
        }
```

```
    }
    printf (" 已准备\n收到 %d 条消息\n", (int) sequence);
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);
    return 0;
}
```

瞬间值

瞬间值指的是那些会立刻过期的值。如果你用克隆模式搭建一个类似DNS的服务时，就可以用瞬间值来模拟动态DNS解析了。当节点连接网络，对外发布它的地址，并不断地更新地址。如果节点断开连接，则它的地址也会失效。

瞬间值可以和会话（session）联系起来，当会话结束时，瞬间值也就失效了。克隆模式中，会话是由客户端定义的，并会在客户端断开连接时消亡。

更简单的方法是为每一个瞬间值设定一个过期时间，客户端会不断延长这个时间，当断开连接时这个时间将得不到更新，服务器就会自动将其删除。

我们会用这种简单的方法来实现瞬间值，因为太过复杂的方法可能不值当，它们的差别仅在性能上体现。如果客户端有很多瞬间值，那为每个值设定过期时间是恰当的；如果瞬间值到达一定的量，那最好还是将其和会话相关联，统一进行过期处理。

首先，我们需要设法在键值对消息中加入过期时间。我们可以增加一个消息帧，但这样一来每当我们需要增加消息内容时就需要修改kvmsg类库了，这并不合适。所以，我们一次性增加一个“属性”消息帧，用于添加不同的消息属性。

其次，我们需要设法删除这条数据。目前为止服务端和客户端会盲目地增改哈希表中的数据，我们可以这样定义：当消息的值是空的，则表示删除这个键的数据。

下面是一个更为完整的kvmsg类代码，它实现了“属性”帧，以及一个UUID帧，我们后面会用到。该类还会负责处理值为空的消息，达到删除的目的：

kvmsg: Key-value message class - full in C

```
/*  =====================================================================
    kvmsg - key-value message class for example applications

    ---------------------------------------------------------------------
    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 "kvmsg.h"
#include <uuid/uuid.h>
#include "zlist.h"

//  键是短字符串
#define KVMSG_KEY_MAX   255

//  消息包含五帧
//  frame 0: 键(ZMQ字符串)
//  frame 1: 编号(8个字节，按顺序排列)
//  frame 2: UUID(二进制块，16个字节)
//  frame 3: 属性(ZMQ字符串)
//  frame 4: 值(二进制块)
#define FRAME_KEY       0
#define FRAME_SEQ       1
#define FRAME_UUID      2
#define FRAME_PROPS     3
#define FRAME_BODY      4
#define KVMSG_FRAMES    5

//  类结构
struct _kvmsg {
    //  帧是否存在
    int present [KVMSG_FRAMES];
    //  对应消息帧
    zmq_msg_t frame [KVMSG_FRAMES];
    //  键，C语言字符串格式
    char key [KVMSG_KEY_MAX + 1];
    //  属性列表，key=value形式
    zlist_t *props;
    size_t props_size;
};
```

```c
//  将属性列表序列化为字符串
static void
s_encode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    if (self->present [FRAME_PROPS])
        zmq_msg_close (msg);

    zmq_msg_init_size (msg, self->props_size);
    char *prop = zlist_first (self->props);
    char *dest = (char *) zmq_msg_data (msg);
    while (prop) {
        strcpy (dest, prop);
        dest += strlen (prop);
        *dest++ = '\n';
        prop = zlist_next (self->props);
    }
    self->present [FRAME_PROPS] = 1;
}

//  从字符串中解析属性列表
static void
s_decode_props (kvmsg_t *self)
{
    zmq_msg_t *msg = &self->frame [FRAME_PROPS];
    self->props_size = 0;
    while (zlist_size (self->props))
        free (zlist_pop (self->props));

    size_t remainder = zmq_msg_size (msg);
    char *prop = (char *) zmq_msg_data (msg);
    char *eoln = memchr (prop, '\n', remainder);
    while (eoln) {
        *eoln = 0;
        zlist_append (self->props, strdup (prop));
        self->props_size += strlen (prop) + 1;
        remainder -= strlen (prop) + 1;
        prop = eoln + 1;
        eoln = memchr (prop, '\n', remainder);
    }
}


//  ---------------------------------------------------------------------
//  构造函数，指定消息编号
```

```c
kvmsg_t *
kvmsg_new (int64_t sequence)
{
    kvmsg_t
        *self;

    self = (kvmsg_t *) zmalloc (sizeof (kvmsg_t));
    self->props = zlist_new ();
    kvmsg_set_sequence (self, sequence);
    return self;
}


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

//  释放内存函数，供zhash_free_fn()调用
void
kvmsg_free (void *ptr)
{
    if (ptr) {
        kvmsg_t *self = (kvmsg_t *) ptr;
        //  释放所有消息帧
        int frame_nbr;
        for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++)
            if (self->present [frame_nbr])
                zmq_msg_close (&self->frame [frame_nbr]);

        //  释放属性列表
        while (zlist_size (self->props))
            free (zlist_pop (self->props));
        zlist_destroy (&self->props);

        //  释放对象本身
        free (self);
    }
}

void
kvmsg_destroy (kvmsg_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_free (*self_p);
        *self_p = NULL;
    }
```

```
}


//  ---------------------------------------------------------------------
//  复制kvmsg对象

kvmsg_t *
kvmsg_dup (kvmsg_t *self)
{
    kvmsg_t *kvmsg = kvmsg_new (0);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr]) {
            zmq_msg_t *src = &self->frame [frame_nbr];
            zmq_msg_t *dst = &kvmsg->frame [frame_nbr];
            zmq_msg_init_size (dst, zmq_msg_size (src));
            memcpy (zmq_msg_data (dst),
                    zmq_msg_data (src), zmq_msg_size (src));
            kvmsg->present [frame_nbr] = 1;
        }
    }
    kvmsg->props = zlist_copy (self->props);
    return kvmsg;
}


//  ---------------------------------------------------------------------
//  从套接字总读取键值对，返回kvmsg实例

kvmsg_t *
kvmsg_recv (void *socket)
{
    assert (socket);
    kvmsg_t *self = kvmsg_new (0);

    //  读取所有帧，若有异常则直接返回空
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        if (self->present [frame_nbr])
            zmq_msg_close (&self->frame [frame_nbr]);
        zmq_msg_init (&self->frame [frame_nbr]);
        self->present [frame_nbr] = 1;
        if (zmq_recvmsg (socket, &self->frame [frame_nbr], 0) == -1) {
            kvmsg_destroy (&self);
            break;
        }
```

```
        //  验证多帧消息
        int rcvmore = (frame_nbr < KVMSG_FRAMES - 1)? 1: 0;
        if (zsockopt_rcvmore (socket) != rcvmore) {
            kvmsg_destroy (&self);
            break;
        }
    }
    if (self)
        s_decode_props (self);
    return self;
}


//  ---------------------------------------------------------------------
//  向套接字发送键值对消息，空消息也发送

void
kvmsg_send (kvmsg_t *self, void *socket)
{
    assert (self);
    assert (socket);

    s_encode_props (self);
    int frame_nbr;
    for (frame_nbr = 0; frame_nbr < KVMSG_FRAMES; frame_nbr++) {
        zmq_msg_t copy;
        zmq_msg_init (&copy);
        if (self->present [frame_nbr])
            zmq_msg_copy (&copy, &self->frame [frame_nbr]);
        zmq_sendmsg (socket, &copy,
            (frame_nbr < KVMSG_FRAMES - 1)? ZMQ_SNDMORE: 0);
        zmq_msg_close (&copy);
    }
}


//  ---------------------------------------------------------------------
//  返回消息的键

char *
kvmsg_key (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_KEY]) {
        if (!*self->key) {
            size_t size = zmq_msg_size (&self->frame [FRAME_KEY]);
```

```
            if (size > KVMSG_KEY_MAX)
                size = KVMSG_KEY_MAX;
            memcpy (self->key,
                zmq_msg_data (&self->frame [FRAME_KEY]), size);
            self->key [size] = 0;
        }
        return self->key;
    }
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息的编号

int64_t
kvmsg_sequence (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_SEQ]) {
        assert (zmq_msg_size (&self->frame [FRAME_SEQ]) == 8);
        byte *source = zmq_msg_data (&self->frame [FRAME_SEQ]);
        int64_t sequence = ((int64_t) (source [0]) << 56)
                         + ((int64_t) (source [1]) << 48)
                         + ((int64_t) (source [2]) << 40)
                         + ((int64_t) (source [3]) << 32)
                         + ((int64_t) (source [4]) << 24)
                         + ((int64_t) (source [5]) << 16)
                         + ((int64_t) (source [6]) << 8)
                         +  (int64_t) (source [7]);
        return sequence;
    }
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  返回消息的UUID

byte *
kvmsg_uuid (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_UUID]
```

```
    &&  zmq_msg_size (&self->frame [FRAME_UUID]) == sizeof (uuid_t))
        return (byte *) zmq_msg_data (&self->frame [FRAME_UUID]);
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息的内容

byte *
kvmsg_body (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return (byte *) zmq_msg_data (&self->frame [FRAME_BODY]);
    else
        return NULL;
}


//  ---------------------------------------------------------------------
//  返回消息内容的长度

size_t
kvmsg_size (kvmsg_t *self)
{
    assert (self);
    if (self->present [FRAME_BODY])
        return zmq_msg_size (&self->frame [FRAME_BODY]);
    else
        return 0;
}


//  ---------------------------------------------------------------------
//  设置消息的键

void
kvmsg_set_key (kvmsg_t *self, char *key)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_KEY];
    if (self->present [FRAME_KEY])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, strlen (key));
```

```
    memcpy (zmq_msg_data (msg), key, strlen (key));
    self->present [FRAME_KEY] = 1;
}


//  ---------------------------------------------------------------------
//  设置消息的编号

void
kvmsg_set_sequence (kvmsg_t *self, int64_t sequence)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_SEQ];
    if (self->present [FRAME_SEQ])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, 8);

    byte *source = zmq_msg_data (msg);
    source [0] = (byte) ((sequence >> 56) & 255);
    source [1] = (byte) ((sequence >> 48) & 255);
    source [2] = (byte) ((sequence >> 40) & 255);
    source [3] = (byte) ((sequence >> 32) & 255);
    source [4] = (byte) ((sequence >> 24) & 255);
    source [5] = (byte) ((sequence >> 16) & 255);
    source [6] = (byte) ((sequence >> 8)  & 255);
    source [7] = (byte) ((sequence)       & 255);

    self->present [FRAME_SEQ] = 1;
}


//  ---------------------------------------------------------------------
//  生成并设置消息的UUID

void
kvmsg_set_uuid (kvmsg_t *self)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_UUID];
    uuid_t uuid;
    uuid_generate (uuid);
    if (self->present [FRAME_UUID])
        zmq_msg_close (msg);
    zmq_msg_init_size (msg, sizeof (uuid));
    memcpy (zmq_msg_data (msg), uuid, sizeof (uuid));
    self->present [FRAME_UUID] = 1;
```

```
}


// ---------------------------------------------------------------------
// 设置消息的内容

void
kvmsg_set_body (kvmsg_t *self, byte *body, size_t size)
{
    assert (self);
    zmq_msg_t *msg = &self->frame [FRAME_BODY];
    if (self->present [FRAME_BODY])
        zmq_msg_close (msg);
    self->present [FRAME_BODY] = 1;
    zmq_msg_init_size (msg, size);
    memcpy (zmq_msg_data (msg), body, size);
}


// ---------------------------------------------------------------------
// 使用printf()格式设置消息的键
void
kvmsg_fmt_key (kvmsg_t *self, char *format, ...)
{
    char value [KVMSG_KEY_MAX + 1];
    va_list args;

    assert (self);
    va_start (args, format);
    vsnprintf (value, KVMSG_KEY_MAX, format, args);
    va_end (args);
    kvmsg_set_key (self, value);
}


// ---------------------------------------------------------------------
// 使用printf()格式设置消息内容

void
kvmsg_fmt_body (kvmsg_t *self, char *format, ...)
{
    char value [255 + 1];
    va_list args;

    assert (self);
    va_start (args, format);
```

```c
    vsnprintf (value, 255, format, args);
    va_end (args);
    kvmsg_set_body (self, (byte *) value, strlen (value));
}


//  ---------------------------------------------------------------------
//  获取消息属性，无则返回空字符串

char *
kvmsg_get_prop (kvmsg_t *self, char *name)
{
    assert (strchr (name, '=') == NULL);
    char *prop = zlist_first (self->props);
    size_t namelen = strlen (name);
    while (prop) {
        if (strlen (prop) > namelen
        &&  memcmp (prop, name, namelen) == 0
        &&  prop [namelen] == '=')
            return prop + namelen + 1;
        prop = zlist_next (self->props);
    }
    return "";
}


//  ---------------------------------------------------------------------
//  设置消息属性
//  属性名称不能包含=号，值的最大长度是255

void
kvmsg_set_prop (kvmsg_t *self, char *name, char *format, ...)
{
    assert (strchr (name, '=') == NULL);

    char value [255 + 1];
    va_list args;
    assert (self);
    va_start (args, format);
    vsnprintf (value, 255, format, args);
    va_end (args);

    //  分配空间
    char *prop = malloc (strlen (name) + strlen (value) + 2);

    //  删除已存在的属性
```

```c
    sprintf (prop, "%s=", name);
    char *existing = zlist_first (self->props);
    while (existing) {
        if (memcmp (prop, existing, strlen (prop)) == 0) {
            self->props_size -= strlen (existing) + 1;
            zlist_remove (self->props, existing);
            free (existing);
            break;
        }
        existing = zlist_next (self->props);
    }
    //  添加新属性
    strcat (prop, value);
    zlist_append (self->props, prop);
    self->props_size += strlen (prop) + 1;
}


//  ---------------------------------------------------------------------
//  在哈希表中保存kvmsg对象
//  当kvmsg对象不再被使用时进行释放操作；
//  若传入的值为空，则删除该对象。

void
kvmsg_store (kvmsg_t **self_p, zhash_t *hash)
{
    assert (self_p);
    if (*self_p) {
        kvmsg_t *self = *self_p;
        assert (self);
        if (kvmsg_size (self)) {
            if (self->present [FRAME_KEY]
            &&  self->present [FRAME_BODY]) {
                zhash_update (hash, kvmsg_key (self), self);
                zhash_freefn (hash, kvmsg_key (self), kvmsg_free);
            }
        }
        else
            zhash_delete (hash, kvmsg_key (self));

        *self_p = NULL;
    }
}


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

```c
//  将消息内容输出到标准错误输出

void
kvmsg_dump (kvmsg_t *self)
{
    if (self) {
        if (!self) {
            fprintf (stderr, "NULL");
            return;
        }
        size_t size = kvmsg_size (self);
        byte  *body = kvmsg_body (self);
        fprintf (stderr, "[seq:%" PRId64 "]", kvmsg_sequence (self));
        fprintf (stderr, "[key:%s]", kvmsg_key (self));
        fprintf (stderr, "[size:%zd] ", size);
        if (zlist_size (self->props)) {
            fprintf (stderr, "[");
            char *prop = zlist_first (self->props);
            while (prop) {
                fprintf (stderr, "%s;", prop);
                prop = zlist_next (self->props);
            }
            fprintf (stderr, "]");
        }
        int char_nbr;
        for (char_nbr = 0; char_nbr < size; char_nbr++)
            fprintf (stderr, "%02X", body [char_nbr]);
        fprintf (stderr, "\n");
    }
    else
        fprintf (stderr, "NULL message\n");
}


//  ---------------------------------------------------------------------
//  测试用例

int
kvmsg_test (int verbose)
{
    kvmsg_t
        *kvmsg;

    printf (" * kvmsg: ");

    //  准备上下文和套接字
```

```
zctx_t *ctx = zctx_new ();
void *output = zsocket_new (ctx, ZMQ_DEALER);
int rc = zmq_bind (output, "ipc://kvmsg_selftest.ipc");
assert (rc == 0);
void *input = zsocket_new (ctx, ZMQ_DEALER);
rc = zmq_connect (input, "ipc://kvmsg_selftest.ipc");
assert (rc == 0);

zhash_t *kvmap = zhash_new ();

//  测试简单消息的收发
kvmsg = kvmsg_new (1);
kvmsg_set_key  (kvmsg, "key");
kvmsg_set_uuid (kvmsg);
kvmsg_set_body (kvmsg, (byte *) "body", 4);
if (verbose)
    kvmsg_dump (kvmsg);
kvmsg_send (kvmsg, output);
kvmsg_store (&kvmsg, kvmap);

kvmsg = kvmsg_recv (input);
if (verbose)
    kvmsg_dump (kvmsg);
assert (streq (kvmsg_key (kvmsg), "key"));
kvmsg_store (&kvmsg, kvmap);

// 测试带有属性的消息的收发
kvmsg = kvmsg_new (2);
kvmsg_set_prop (kvmsg, "prop1", "value1");
kvmsg_set_prop (kvmsg, "prop2", "value1");
kvmsg_set_prop (kvmsg, "prop2", "value2");
kvmsg_set_key  (kvmsg, "key");
kvmsg_set_uuid (kvmsg);
kvmsg_set_body (kvmsg, (byte *) "body", 4);
assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
if (verbose)
    kvmsg_dump (kvmsg);
kvmsg_send (kvmsg, output);
kvmsg_destroy (&kvmsg);

kvmsg = kvmsg_recv (input);
if (verbose)
    kvmsg_dump (kvmsg);
assert (streq (kvmsg_key (kvmsg), "key"));
assert (streq (kvmsg_get_prop (kvmsg, "prop2"), "value2"));
kvmsg_destroy (&kvmsg);
```

```
    //  关闭并销毁所有对象
    zhash_destroy (&kvmap);
    zctx_destroy (&ctx);

    printf ("OK\n");
    return 0;
}
```

客户端模型5和模型4没有太大区别，只是kvmsg类库变了。在更新消息的时候还需要添加一个过期时间的属性：

```
kvmsg_set_prop (kvmsg, "ttl", "%d", randof (30));
```

服务端模型5有较大的变化，我们会用反应堆来代替轮询，这样就能混合处理定时事件和套接字事件了，只是在C语言中是比较麻烦的。下面是代码：

clonesrv5: Clone server, Model Five in C

```
//
//  克隆模式 - 服务端 - 模型5
//

//  直接编译，不建类库
#include "kvmsg.c"

//  反应堆处理器
static int s_snapshots  (zloop_t *loop, void *socket, void *args);
static int s_collector  (zloop_t *loop, void *socket, void *args);
static int s_flush_ttl  (zloop_t *loop, void *socket, void *args);

//  服务器属性
typedef struct {
    zctx_t *ctx;                //  上下文
    zhash_t *kvmap;             //  键值对存储
    zloop_t *loop;              //  zloop反应堆
    int port;                   //  主端口
    int64_t sequence;           //  更新事件编号
    void *snapshot;             //  处理快照请求
    void *publisher;            //  发布更新事件
    void *collector;            //  从客户端收集接收更新事件
} clonesrv_t;


int main (void)
```

```
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));

    self->port = 5556;
    self->ctx = zctx_new ();
    self->kvmap = zhash_new ();
    self->loop = zloop_new ();
    zloop_set_verbose (self->loop, FALSE);

    //  打开克隆模式服务端套接字
    self->snapshot  = zsocket_new (self->ctx, ZMQ_ROUTER);
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_PULL);
    zsocket_bind (self->snapshot,  "tcp://*:%d", self->port);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  注册反应堆处理程序
    zloop_reader (self->loop, self->snapshot, s_snapshots, self);
    zloop_reader (self->loop, self->collector, s_collector, self);
    zloop_timer  (self->loop, 1000, 0, s_flush_ttl, self);

    //  运行反应堆，直至中断
    zloop_start (self->loop);

    zloop_destroy (&self->loop);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);
    return 0;
}


//  ---------------------------------------------------------------------
//  发送快照内容

static int s_send_single (char *key, void *data, void *args);

//  请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求方标识
    char *subtree;          //  子树信息
} kvroute_t;

static int
```

```c
s_snapshots (zloop_t *loop, void *snapshot, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (snapshot);
    if (identity) {
        //  请求位于消息第二帧
        char *request = zstr_recv (snapshot);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (snapshot);
        }
        else
            printf ("E: 错误的请求，程序中止\n");

        if (subtree) {
            //  发送状态快照
            kvroute_t routing = { snapshot, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  发送结束符和版本号
            zclock_log ("I: 正在发送快照，版本号：%d", (int) self->sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    return 0;
}


//  每次发送一个快照键值对
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  先发送接收方标识
```

```
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}


//  ---------------------------------------------------------------------
//  收集更新事件

static int
s_collector (zloop_t *loop, void *collector, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (collector);
    if (kvmsg) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
        if (ttl)
            kvmsg_set_prop (kvmsg, "ttl",
                "%" PRId64, zclock_time () + ttl * 1000);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 正在发布更新事件 %d", (int) self->sequence);
    }
    return 0;
}


//  ---------------------------------------------------------------------
//  删除过期的瞬间值

static int s_flush_single (char *key, void *data, void *args);

static int
s_flush_ttl (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}

//  删除过期的键值对，并广播该事件
static int
```

```
s_flush_single (char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 发布删除事件 %d", (int) self->sequence);
    }
    return 0;
}
```

克隆服务器的可靠性

克隆模型1至5相对比较简单，下面我们会探讨一个非常复杂的模型。可以发现，为了构建可靠的消息队列，我们需要花费非常多的精力。所以我们经常会问：有必要这么做吗？如果说你能够接受可靠性不够高的、或者说已经足够好的架构，那恭喜你，你在成本和收益之间找到了平衡。虽然我们会偶尔丢失一些消息，但从经济的角度来说还是合理的。不管怎样，下面我们就来介绍这个复杂的模型。

在模型3中，你会关闭和重启服务，这会导致数据的丢失。任何后续加入的客户端只能得到重启之后的那些数据，而非所有的。下面就让我们想办法让克隆模式能够承担服务器重启的故障。

以下列举我们需要处理的问题：

- 克隆服务器进程崩溃并自动或手工重启。进程丢失了所有数据，所以必须从别处进行恢复。
- 克隆服务器硬件故障，长时间不能恢复。客户端需要切换至另一个可用的服务端。
- 克隆服务器从网络上断开，如交换机发生故障等。它会在某个时点重连，但期间的数据就需要替代的服务器负责处理。

第一步我们需要增加一个服务器。我们可以使用第四章中提到的双子星模式，它是一个反应堆，而我们的程序经过整理后也是一个反应堆，因此可以互相协作。

我们需要保证更新事件在主服务器崩溃时仍能保留，最简单的机制就是同时发送给两台服务器。

备机就可以当做一台客户端来运行，像其他客户端一样从主机获取更新事件。同时它又能从客户端获取更新事件——虽然不应该以此更新数据，但可以先暂存起来。

所以，相较于模型5，模型6中引入了以下特性：

- 客户端发送更新事件改用PUB-SUB套接字，而非PUSH-PULL。原因是PUSH套接字会在没有接收方时阻塞，且会进行负载均衡——我们需要两台服务器都接收到消息。我们会在服务器端绑定SUB套接字，在客户端连接PUB套接字。

- 我们在服务器发送给客户端的更新事件中加入心跳，这样客户端可以知道主机是否已死，然后切换至备机。

- 我们使用双子星模式的bstar反应堆类来创建主机和备机。双子星模式中需要有一个“投票”套接字，来协助判定对方节点是否已死。这里我们使用快照请求来作为“投票”。

- 我们将为所有的更新事件添加UUID属性，它由客户端生成，服务端会将其发布给所有客户端。

- 备机将维护一个“待处理列表”，保存来自客户端、尚未由服务端发布的更新事件；或者反过来，来自服务端、尚未从客户端收到的更新事件。这个列表从旧到新排列，这样就能方便地从顶部删除消息。

我们可以为客户端设计一个有限状态机，它有三种状态：

- 客户端打开并连接了套接字，然后向服务端发送快照请求。为了避免消息风暴，它只会请求两次。

- 客户端等待快照应答，如果获得了则保存它；如果没有获得，则向第二个服务器发送请求。

- 客户端收到快照，便开始等待更新事件。如果在一定时间内没有收到服务端响应，则会连接第二个服务端。

客户端会一直循环下去，可能在程序刚启动时，部分客户端会试图连接主机，部分连接备机，相信双子星模式会很好地处理这一情况的。

我们可以将客户端状态图绘制出来：

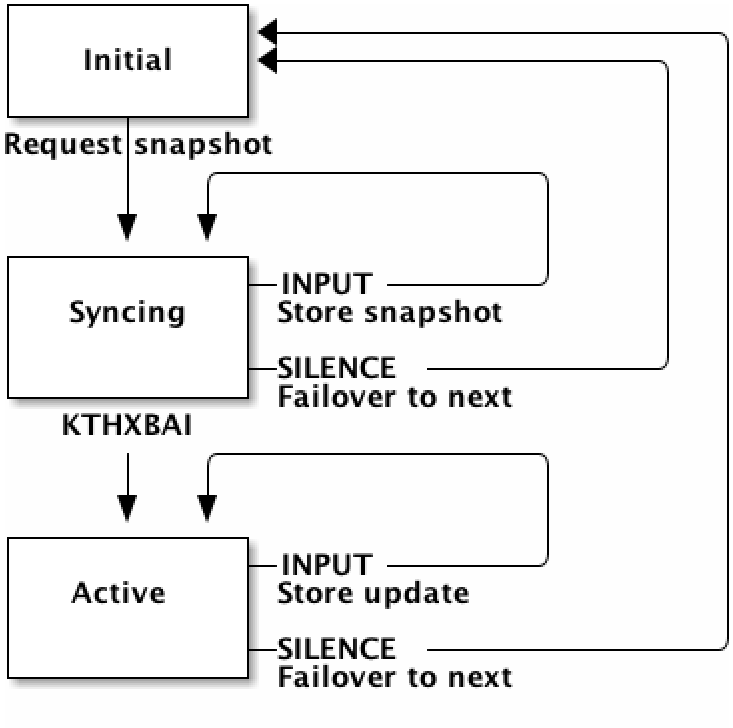

Figure 6 — Clone client FSM

故障恢复的步骤如下：

- 客户端检测到主机不再发送心跳，因此转而连接备机，并请求一份新的快照；
- 备机开始接收快照请求，并检测到主机死亡，于是开始作为主机运行；
- 备机将待处理列表中的更新事件写入自身状态中，然后开始处理快照请求。

当主机恢复连接时：

- 启动为slave状态，并作为克隆模式客户端连接备机；
- 同时，使用SUB套接字从客户端接收更新事件。

我们做两点假设：

- 至少有一台主机会继续运行。如果两台主机都崩溃了，那我们将丢失所有的服务端数据，无法恢复。
- 不同的客户端不会同时更新同一个键值对。客户端的更新事件会先后到达两个服务器，因此更新的顺序可能会不一致。单个客户端的更新事件到达两台服务器的顺序是相同的，所以不用担心。

下面是整体架构图：

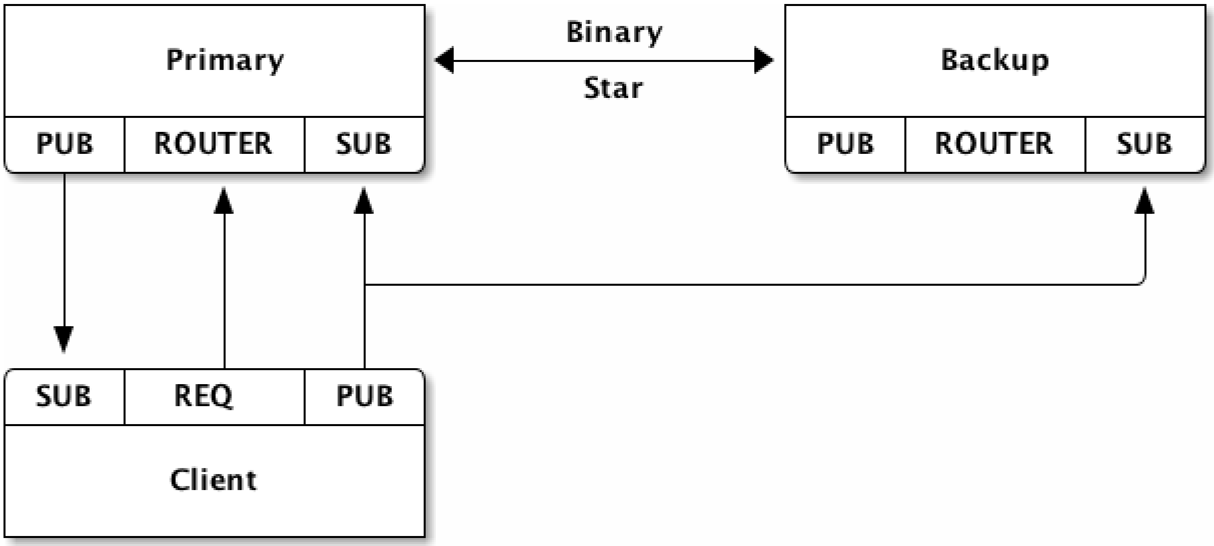

Figure 7 — High availability Clone Server Pair

开始编程之前，我们需要将客户端重构成一个可复用的类。在ZMQ中写异步类有时是为了练习如何写出优雅的代码，但这里我们确实是希望克隆模式可以成为一种易于使用的程序。上述架构的伸缩性来源于客户端的正确行为，因此有必要将其封装成一份API。要在客户端中进行故障恢复还是比较复杂的，试想一下自由者模式和克隆模式结合起来会是什么样的吧。

按照我的习惯，我会先写出一份API的列表，然后加以实现。让我们假想一个名为clone的API，在其基础之上编写克隆模式客户端API。将代码封装为API显然会提升代码的稳定性，就以模型5为例，客户端需要打开三个套接字，端点名称直接写在了代码里。我们可以创建这样一组API：

```
//  为每个套接字指定端点
clone_subscribe (clone, "tcp://localhost:5556");
clone_snapshot  (clone, "tcp://localhost:5557");
clone_updates   (clone, "tcp://localhost:5558");

//  由于有两个服务端，因此再执行一次
clone_subscribe (clone, "tcp://localhost:5566");
clone_snapshot  (clone, "tcp://localhost:5567");
clone_updates   (clone, "tcp://localhost:5568");
```

但这种写法还是比较啰嗦的，因为没有必要将API内部的一些设计暴露给编程人员。现在我们会使用三个套接字，而将来可能就会使用两个，或者四个。我们不可能让所有的应用程序都相应地修改吧？让我们把这些信息包装到API中：

```
//  指定主备服务器端点
clone_connect (clone, "tcp://localhost:5551");
clone_connect (clone, "tcp://localhost:5561");
```

这样一来代码就变得非常简洁，不过也会对现有代码的内部就够造成影响。我们需要从一个端点中推算出三个端点。一种方法是假设客户端和服务端使用三个连续的端点通信，并将这个规则写入协议；另一个方法是向服

务器索取缺少的端点信息。我们使用第一种较为简单的方法：

- 服务器状态ROUTER在端点P；
- 服务器更新事件PUB在端点P + 1；
- 服务器更新事件SUB在端点P + 2。

clone类和第四章的flcliapi类很类似，由两部分组成：

- 一个在后台运行的异步克隆模式代理。该代理处理所有的I/O操作，实时地和服务器进行通信；
- 一个在前台应用程序中同步运行的clone类。当你创建了一个clone对象后，它会自动创建后台的clone线程；当你销毁clone对象，该后台线程也会被销毁。

前台的clone类会使用inproc管道和后台的代理进行通信。C语言中，czmq线程会自动为我们创建这个管道。这也是ZMQ多线程编程的常规方式。

如果没有ZMQ，这种异步的设计将很难处理高压工作，而ZMQ会让其变得简单。编写出来额代码会相对比较复杂。我们可以用反应堆的模式来编写，但这会进一步增加复杂度，且影响应用程序的使用。因此，我们的设计的API将更像是一个能够和服务器进行通信的键值表：

```
clone_t *clone_new (void);
void clone_destroy (clone_t **self_p);
void clone_connect (clone_t *self, char *address, char *service);
void clone_set (clone_t *self, char *key, char *value);
char *clone_get (clone_t *self, char *key);
```

下面就是克隆模式客户端模型6的代码，因为调用了API，所以非常简短：
clonecli6: Clone client, Model Six in C

```
//
//  克隆模式 - 客户端 - 模型6
//

//  直接编译，不建类库
#include "clone.c"

#define SUBTREE "/client/"

int main (void)
{
    //  创建分布式哈希表
    clone_t *clone = clone_new ();

    //  配置
    clone_subtree (clone, SUBTREE);
    clone_connect (clone, "tcp://localhost", "5556");
    clone_connect (clone, "tcp://localhost", "5566");
```

```
    //  插入随机键值
    while (!zctx_interrupted) {
        //  生成随机值
        char key [255];
        char value [10];
        sprintf (key, "%s%d", SUBTREE, randof (10000));
        sprintf (value, "%d", randof (1000000));
        clone_set (clone, key, value, randof (30));
        sleep (1);
    }
    clone_destroy (&clone);
    return 0;
}
```

以下是clone类的实现：

clone: Clone class in C

```
/*  =====================================================================
    clone - client-side Clone Pattern class

    ---------------------------------------------------------------------
    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 "clone.h"

//  请求超时时间
#define GLOBAL_TIMEOUT  4000    //  msecs
```

```c
//  判定服务器死亡的时间
#define SERVER_TTL      5000    //  msecs
//  服务器数量
#define SERVER_MAX      2


//  =====================================================================
//  同步部分，在应用程序线程中工作

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

struct _clone_t {
    zctx_t *ctx;                //  上下文
    void *pipe;                 //  和后台代理间的通信套接字
};

//  该线程用于处理真正的clone类
static void clone_agent (void *args, zctx_t *ctx, void *pipe);


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

clone_t *
clone_new (void)
{
    clone_t
        *self;

    self = (clone_t *) zmalloc (sizeof (clone_t));
    self->ctx = zctx_new ();
    self->pipe = zthread_fork (self->ctx, clone_agent, NULL);
    return self;
}

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

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

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

//  ---------------------------------------------------------------------
//  在链接之前指定快照和更新事件的子树
//  发送给后台代理的消息内容为[SUBTREE][subtree]

void clone_subtree (clone_t *self, char *subtree)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SUBTREE");
    zmsg_addstr (msg, subtree);
    zmsg_send (&msg, self->pipe);
}

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

void
clone_connect (clone_t *self, char *address, char *service)
{
    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "CONNECT");
    zmsg_addstr (msg, address);
    zmsg_addstr (msg, service);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
//  设置新值
//  消息内容：[SET][key][value][ttl]

void
clone_set (clone_t *self, char *key, char *value, int ttl)
{
    char ttlstr [10];
    sprintf (ttlstr, "%d", ttl);

    assert (self);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "SET");
```

```c
    zmsg_addstr (msg, key);
    zmsg_addstr (msg, value);
    zmsg_addstr (msg, ttlstr);
    zmsg_send (&msg, self->pipe);
}

//  ---------------------------------------------------------------------
//  取值
//  消息内容：[GET][key]
//  如果没有clone可用，会返回NULL

char *
clone_get (clone_t *self, char *key)
{
    assert (self);
    assert (key);
    zmsg_t *msg = zmsg_new ();
    zmsg_addstr (msg, "GET");
    zmsg_addstr (msg, key);
    zmsg_send (&msg, self->pipe);

    zmsg_t *reply = zmsg_recv (self->pipe);
    if (reply) {
        char *value = zmsg_popstr (reply);
        zmsg_destroy (&reply);
        return value;
    }
    return NULL;
}


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

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

typedef struct {
    char *address;              //  服务端地址
    int port;                   //  端口
    void *snapshot;             //  快照套接字
    void *subscriber;           //  接收更新事件的套接字
    uint64_t expiry;            //  服务器过期时间
    uint requests;              //  收到的快照请求数
} server_t;
```

```
static server_t *
server_new (zctx_t *ctx, char *address, int port, char *subtree)
{
    server_t *self = (server_t *) zmalloc (sizeof (server_t));

    zclock_log ("I: adding server %s:%d...", address, port);
    self->address = strdup (address);
    self->port = port;

    self->snapshot = zsocket_new (ctx, ZMQ_DEALER);
    zsocket_connect (self->snapshot, "%s:%d", address, port);
    self->subscriber = zsocket_new (ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "%s:%d", address, port + 1);
    zsockopt_set_subscribe (self->subscriber, subtree);
    return self;
}

static void
server_destroy (server_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        server_t *self = *self_p;
        free (self->address);
        free (self);
        *self_p = NULL;
    }
}

//  ---------------------------------------------------------------------
//  后台代理类

//  状态
#define STATE_INITIAL       0   //  连接之前
#define STATE_SYNCING       1   //  正在同步
#define STATE_ACTIVE        2   //  正在更新

typedef struct {
    zctx_t *ctx;                //  上下文
    void *pipe;                 //  与主线程通信的套接字
    zhash_t *kvmap;             //  键值表
    char *subtree;              //  子树
    server_t *server [SERVER_MAX];
    uint nbr_servers;           //  范围：0 - SERVER_MAX
    uint state;                 //  当前状态
    uint cur_server;            //  当前master，0/1
```

```
    int64_t sequence;           //  键值对编号
    void *publisher;            //  发布更新事件的套接字
} agent_t;

static 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->kvmap = zhash_new ();
    self->subtree = strdup ("");
    self->state = STATE_INITIAL;
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    return self;
}

static void
agent_destroy (agent_t **self_p)
{
    assert (self_p);
    if (*self_p) {
        agent_t *self = *self_p;
        int server_nbr;
        for (server_nbr = 0; server_nbr < self->nbr_servers; server_nbr++)
            server_destroy (&self->server [server_nbr]);
        zhash_destroy (&self->kvmap);
        free (self->subtree);
        free (self);
        *self_p = NULL;
    }
}

//  若线程被中断则返回-1
static int
agent_control_message (agent_t *self)
{
    zmsg_t *msg = zmsg_recv (self->pipe);
    char *command = zmsg_popstr (msg);
    if (command == NULL)
        return -1;

    if (streq (command, "SUBTREE")) {
        free (self->subtree);
        self->subtree = zmsg_popstr (msg);
    }
```

```c
    else
    if (streq (command, "CONNECT")) {
        char *address = zmsg_popstr (msg);
        char *service = zmsg_popstr (msg);
        if (self->nbr_servers < SERVER_MAX) {
            self->server [self->nbr_servers++] = server_new (
                self->ctx, address, atoi (service), self->subtree);
            //  广播更新事件
            zsocket_connect (self->publisher, "%s:%d",
                address, atoi (service) + 2);
        }
        else
            zclock_log ("E: too many servers (max. %d)", SERVER_MAX);
        free (address);
        free (service);
    }
    else
    if (streq (command, "SET")) {
        char *key = zmsg_popstr (msg);
        char *value = zmsg_popstr (msg);
        char *ttl = zmsg_popstr (msg);
        zhash_update (self->kvmap, key, (byte *) value);
        zhash_freefn (self->kvmap, key, free);

        //  向服务端发送键值对
        kvmsg_t *kvmsg = kvmsg_new (0);
        kvmsg_set_key  (kvmsg, key);
        kvmsg_set_uuid (kvmsg);
        kvmsg_fmt_body (kvmsg, "%s", value);
        kvmsg_set_prop (kvmsg, "ttl", ttl);
        kvmsg_send     (kvmsg, self->publisher);
        kvmsg_destroy (&kvmsg);
puts (key);
        free (ttl);
        free (key);             //  键值对实际由哈希表对象控制
    }
    else
    if (streq (command, "GET")) {
        char *key = zmsg_popstr (msg);
        char *value = zhash_lookup (self->kvmap, key);
        if (value)
            zstr_send (self->pipe, value);
        else
            zstr_send (self->pipe, "");
        free (key);
        free (value);
```

```
    }
    free (command);
    zmsg_destroy (&msg);
    return 0;
}


//  ---------------------------------------------------------------------
//  异步的后台代理会维护一个服务端池，并处理来自应用程序的请求或应答。

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

    while (TRUE) {
        zmq_pollitem_t poll_set [] = {
            { pipe, 0, ZMQ_POLLIN, 0 },
            { 0,    0, ZMQ_POLLIN, 0 }
        };
        int poll_timer = -1;
        int poll_size = 2;
        server_t *server = self->server [self->cur_server];
        switch (self->state) {
            case STATE_INITIAL:
                //  该状态下，如果有可用服务，会发送快照请求
                if (self->nbr_servers > 0) {
                    zclock_log ("I: 正在等待服务器 %s:%d...",
                        server->address, server->port);
                    if (server->requests < 2) {
                        zstr_sendm (server->snapshot, "ICANHAZ?");
                        zstr_send  (server->snapshot, self->subtree);
                        server->requests++;
                    }
                    server->expiry = zclock_time () + SERVER_TTL;
                    self->state = STATE_SYNCING;
                    poll_set [1].socket = server->snapshot;
                }
                else
                    poll_size = 1;
                break;
            case STATE_SYNCING:
                //  该状态下我们从服务器端接收快照内容，若失败则尝试其他服务器
                poll_set [1].socket = server->snapshot;
                break;
            case STATE_ACTIVE:
```

```
            //  该状态下我们从服务器获取更新事件，失败则尝试其他服务器
            poll_set [1].socket = server->subscriber;
            break;
    }
    if (server) {
        poll_timer = (server->expiry - zclock_time ())
                   * ZMQ_POLL_MSEC;
        if (poll_timer < 0)
            poll_timer = 0;
    }
    //  ------------------------------------------------------------
    //  poll循环
    int rc = zmq_poll (poll_set, poll_size, poll_timer);
    if (rc == -1)
        break;              //  上下文已被关闭

    if (poll_set [0].revents & ZMQ_POLLIN) {
        if (agent_control_message (self))
            break;          //  中断
    }
    else
    if (poll_set [1].revents & ZMQ_POLLIN) {
        kvmsg_t *kvmsg = kvmsg_recv (poll_set [1].socket);
        if (!kvmsg)
            break;          //  中断

        //  任何服务端的消息将重置它的过期时间
        server->expiry = zclock_time () + SERVER_TTL;
        if (self->state == STATE_SYNCING) {
            //  保存快照内容
            server->requests = 0;
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                self->state = STATE_ACTIVE;
                zclock_log ("I: received from %s:%d snapshot=%d",
                    server->address, server->port,
                    (int) self->sequence);
                kvmsg_destroy (&kvmsg);
            }
            else
                kvmsg_store (&kvmsg, self->kvmap);
        }
        else
        if (self->state == STATE_ACTIVE) {
            //  丢弃过期的更新事件
            if (kvmsg_sequence (kvmsg) > self->sequence) {
```

```
                self->sequence = kvmsg_sequence (kvmsg);
                kvmsg_store (&kvmsg, self->kvmap);
                zclock_log ("I: received from %s:%d update=%d",
                    server->address, server->port,
                    (int) self->sequence);
            }
            else
                kvmsg_destroy (&kvmsg);
        }
    }
    else {
        //  服务端已死，尝试其他服务器
        zclock_log ("I: 服务器 %s:%d 无响应",
                server->address, server->port);
        self->cur_server = (self->cur_server + 1) % self->nbr_servers;
        self->state = STATE_INITIAL;
    }
  }
  agent_destroy (&self);
}
```

最后是克隆服务器的模型6代码：

clonesrv6: Clone server, Model Six in C

```
//
// 克隆模式 - 服务端 - 模型6
//

//  直接编译，不建类库
#include "bstar.c"
#include "kvmsg.c"

//  bstar反应堆API
static int s_snapshots  (zloop_t *loop, void *socket, void *args);
static int s_collector  (zloop_t *loop, void *socket, void *args);
static int s_flush_ttl  (zloop_t *loop, void *socket, void *args);
static int s_send_hugz  (zloop_t *loop, void *socket, void *args);
static int s_new_master (zloop_t *loop, void *unused, void *args);
static int s_new_slave  (zloop_t *loop, void *unused, void *args);
static int s_subscriber (zloop_t *loop, void *socket, void *args);

//  服务端属性
typedef struct {
    zctx_t *ctx;                //  上下文
    zhash_t *kvmap;             //  存放键值对
```

```c
    bstar_t *bstar;              //  bstar反应堆核心
    int64_t sequence;            //  更新事件编号
    int port;                    //  主端口
    int peer;                    //  同伴端口
    void *publisher;             //  发布更新事件的端口
    void *collector;             //  接收客户端更新事件的端口
    void *subscriber;            //  接受同伴更新事件的端口
    zlist_t *pending;            //  延迟的更新事件
    Bool primary;                //  是否为主机
    Bool master;                 //  是否为master
    Bool slave;                  //  是否为slave
} clonesrv_t;


int main (int argc, char *argv [])
{
    clonesrv_t *self = (clonesrv_t *) zmalloc (sizeof (clonesrv_t));
    if (argc == 2 && streq (argv [1], "-p")) {
        zclock_log ("I: 作为主机master运行，正在等待备机slave连接。");
        self->bstar = bstar_new (BSTAR_PRIMARY, "tcp://*:5003",
                                 "tcp://localhost:5004");
        bstar_voter (self->bstar, "tcp://*:5556", ZMQ_ROUTER,
                     s_snapshots, self);
        self->port = 5556;
        self->peer = 5566;
        self->primary = TRUE;
    }
    else
    if (argc == 2 && streq (argv [1], "-b")) {
        zclock_log ("I: 作为备机slave运行，正在等待主机master连接。");
        self->bstar = bstar_new (BSTAR_BACKUP, "tcp://*:5004",
                                 "tcp://localhost:5003");
        bstar_voter (self->bstar, "tcp://*:5566", ZMQ_ROUTER,
                     s_snapshots, self);
        self->port = 5566;
        self->peer = 5556;
        self->primary = FALSE;
    }
    else {
        printf ("Usage: clonesrv4 { -p | -b }\n");
        free (self);
        exit (0);
    }
    //  主机将成为master
    if (self->primary)
        self->kvmap = zhash_new ();
```

```
    self->ctx = zctx_new ();
    self->pending = zlist_new ();
    bstar_set_verbose (self->bstar, TRUE);

    //  设置克隆服务端套接字
    self->publisher = zsocket_new (self->ctx, ZMQ_PUB);
    self->collector = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_bind (self->publisher, "tcp://*:%d", self->port + 1);
    zsocket_bind (self->collector, "tcp://*:%d", self->port + 2);

    //  作为克隆客户端连接同伴
    self->subscriber = zsocket_new (self->ctx, ZMQ_SUB);
    zsocket_connect (self->subscriber, "tcp://localhost:%d", self->peer + 1);

    //  注册状态事件处理器
    bstar_new_master (self->bstar, s_new_master, self);
    bstar_new_slave (self->bstar, s_new_slave, self);

    //  注册bstar反应堆其他事件处理器
    zloop_reader (bstar_zloop (self->bstar), self->collector, s_collector, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_flush_ttl, self);
    zloop_timer  (bstar_zloop (self->bstar), 1000, 0, s_send_hugz, self);

    //  开启bstar反应堆
    bstar_start (self->bstar);

    //  中断，终止。
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_destroy (&kvmsg);
    }
    zlist_destroy (&self->pending);
    bstar_destroy (&self->bstar);
    zhash_destroy (&self->kvmap);
    zctx_destroy (&self->ctx);
    free (self);

    return 0;
}


//  ---------------------------------------------------------------------
//  发送快照内容

static int s_send_single (char *key, void *data, void *args);
```

```c
//  请求方信息
typedef struct {
    void *socket;           //  ROUTER套接字
    zframe_t *identity;     //  请求放标识
    char *subtree;          //  子树
} kvroute_t;

static int
s_snapshots (zloop_t *loop, void *snapshot, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zframe_t *identity = zframe_recv (snapshot);
    if (identity) {
        //  请求在消息的第二帧中
        char *request = zstr_recv (snapshot);
        char *subtree = NULL;
        if (streq (request, "ICANHAZ?")) {
            free (request);
            subtree = zstr_recv (snapshot);
        }
        else
            printf ("E: 错误的请求，正在退出……\n");

        if (subtree) {
            //  发送状态快照
            kvroute_t routing = { snapshot, identity, subtree };
            zhash_foreach (self->kvmap, s_send_single, &routing);

            //  发送终止消息，以及消息编号
            zclock_log ("I: 正在发送快照，版本号：%d", (int) self->sequence);
            zframe_send (&identity, snapshot, ZFRAME_MORE);
            kvmsg_t *kvmsg = kvmsg_new (self->sequence);
            kvmsg_set_key  (kvmsg, "KTHXBAI");
            kvmsg_set_body (kvmsg, (byte *) subtree, 0);
            kvmsg_send     (kvmsg, snapshot);
            kvmsg_destroy (&kvmsg);
            free (subtree);
        }
    }
    return 0;
}


//  每次发送一个快照键值对
```

```
static int
s_send_single (char *key, void *data, void *args)
{
    kvroute_t *kvroute = (kvroute_t *) args;
    kvmsg_t *kvmsg = (kvmsg_t *) data;
    if (strlen (kvroute->subtree) <= strlen (kvmsg_key (kvmsg))
    &&  memcmp (kvroute->subtree,
                kvmsg_key (kvmsg), strlen (kvroute->subtree)) == 0) {
        //  先发送接收方的地址
        zframe_send (&kvroute->identity,
            kvroute->socket, ZFRAME_MORE + ZFRAME_REUSE);
        kvmsg_send (kvmsg, kvroute->socket);
    }
    return 0;
}


//  ---------------------------------------------------------------------
//  从客户端收集更新事件
//  如果我们是master，则将该事件写入kvmap对象；
//  如果我们是slave，则将其写入延迟队列

static int s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg);

static int
s_collector (zloop_t *loop, void *collector, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_recv (collector);
    kvmsg_dump (kvmsg);
    if (kvmsg) {
        if (self->master) {
            kvmsg_set_sequence (kvmsg, ++self->sequence);
            kvmsg_send (kvmsg, self->publisher);
            int ttl = atoi (kvmsg_get_prop (kvmsg, "ttl"));
            if (ttl)
                kvmsg_set_prop (kvmsg, "ttl",
                    "%" PRId64, zclock_time () + ttl * 1000);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: 正在发布更新事件：%d", (int) self->sequence);
        }
        else {
            //  如果我们已经从master中获得了该事件，则丢弃该消息
            if (s_was_pending (self, kvmsg))
                kvmsg_destroy (&kvmsg);
```

```c
            else
                zlist_append (self->pending, kvmsg);
        }
    }
    return 0;
}

//  如果消息已在延迟队列中，则删除它并返回TRUE

static int
s_was_pending (clonesrv_t *self, kvmsg_t *kvmsg)
{
    kvmsg_t *held = (kvmsg_t *) zlist_first (self->pending);
    while (held) {
        if (memcmp (kvmsg_uuid (kvmsg),
                    kvmsg_uuid (held), sizeof (uuid_t)) == 0) {
            zlist_remove (self->pending, held);
            return TRUE;
        }
        held = (kvmsg_t *) zlist_next (self->pending);
    }
    return FALSE;
}


//  ---------------------------------------------------------------------
//  删除带有过期时间的瞬间值

static int s_flush_single (char *key, void *data, void *args);

static int
s_flush_ttl (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    zhash_foreach (self->kvmap, s_flush_single, args);
    return 0;
}

//  如果键值对过期，则进行删除操作，并广播该事件
static int
s_flush_single (char *key, void *data, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = (kvmsg_t *) data;
    int64_t ttl;
```

```
    sscanf (kvmsg_get_prop (kvmsg, "ttl"), "%" PRId64, &ttl);
    if (ttl && zclock_time () >= ttl) {
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_set_body (kvmsg, (byte *) "", 0);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 正在发布删除事件：%d", (int) self->sequence);
    }
    return 0;
}


//  ---------------------------------------------------------------------
//  发送心跳

static int
s_send_hugz (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    kvmsg_t *kvmsg = kvmsg_new (self->sequence);
    kvmsg_set_key  (kvmsg, "HUGZ");
    kvmsg_set_body (kvmsg, (byte *) "", 0);
    kvmsg_send     (kvmsg, self->publisher);
    kvmsg_destroy (&kvmsg);

    return 0;
}


//  ---------------------------------------------------------------------
//  状态改变事件处理函数
//  我们将转变为master
//
//  备机先将延迟列表中的事件更新到自己的快照中，
//  并开始接收客户端发来的快照请求。

static int
s_new_master (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    self->master = TRUE;
    self->slave = FALSE;
    zloop_cancel (bstar_zloop (self->bstar), self->subscriber);
```

```
    //  应用延迟列表中的事件
    while (zlist_size (self->pending)) {
        kvmsg_t *kvmsg = (kvmsg_t *) zlist_pop (self->pending);
        kvmsg_set_sequence (kvmsg, ++self->sequence);
        kvmsg_send (kvmsg, self->publisher);
        kvmsg_store (&kvmsg, self->kvmap);
        zclock_log ("I: 正在发布延迟列表中的更新事件：%d", (int) self->sequence);
    }
    return 0;
}

//  ---------------------------------------------------------------------
//  正在切换为slave

static int
s_new_slave (zloop_t *loop, void *unused, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;

    zhash_destroy (&self->kvmap);
    self->master = FALSE;
    self->slave = TRUE;
    zloop_reader (bstar_zloop (self->bstar), self->subscriber,
                  s_subscriber, self);

    return 0;
}

//  ---------------------------------------------------------------------
//  从同伴主机（master）接收更新事件；
//  接收该类更新事件时，我们一定是slave。

static int
s_subscriber (zloop_t *loop, void *subscriber, void *args)
{
    clonesrv_t *self = (clonesrv_t *) args;
    //  获取快照，如果需要的话。
    if (self->kvmap == NULL) {
        self->kvmap = zhash_new ();
        void *snapshot = zsocket_new (self->ctx, ZMQ_DEALER);
        zsocket_connect (snapshot, "tcp://localhost:%d", self->peer);
        zclock_log ("I: 正在请求快照：tcp://localhost:%d",
                    self->peer);
        zstr_send (snapshot, "ICANHAZ?");
        while (TRUE) {
            kvmsg_t *kvmsg = kvmsg_recv (snapshot);
```

```
            if (!kvmsg)
                break;          //  中断
            if (streq (kvmsg_key (kvmsg), "KTHXBAI")) {
                self->sequence = kvmsg_sequence (kvmsg);
                kvmsg_destroy (&kvmsg);
                break;          //  完成
            }
            kvmsg_store (&kvmsg, self->kvmap);
        }
        zclock_log ("I: 收到快照，版本号：%d", (int) self->sequence);
        zsocket_destroy (self->ctx, snapshot);
    }
    //  查找并删除
    kvmsg_t *kvmsg = kvmsg_recv (subscriber);
    if (!kvmsg)
        return 0;

    if (strneq (kvmsg_key (kvmsg), "HUGZ")) {
        if (!s_was_pending (self, kvmsg)) {
            //  如果master的更新事件比客户端的事件早到，则将master的事件存入延迟列表，
            //  当收到客户端更新事件时会将其从列表中清除。
            zlist_append (self->pending, kvmsg_dup (kvmsg));
        }
        //  如果更新事件比kvmap版本高，则应用它
        if (kvmsg_sequence (kvmsg) > self->sequence) {
            self->sequence = kvmsg_sequence (kvmsg);
            kvmsg_store (&kvmsg, self->kvmap);
            zclock_log ("I: 收到更新事件：%d", (int) self->sequence);
        }
        else
            kvmsg_destroy (&kvmsg);
    }
    else
        kvmsg_destroy (&kvmsg);

    return 0;
}
```

这段程序只有几百行，但还是花了一些时间来进行调通的。这个模型中包含了故障恢复，瞬间值，子树等等。虽然我们前期设计得很完备，但要在多个套接字之间进行调试还是很困难的。以下是我的工作方式：

- 由于使用了反应堆（bstar，建立在zloop之上），我们节省了大量的代码，让程序变得简洁明了。整个服务以一个线程运行，因此不会出现跨线程的问题。只需将结构指针（self）传递给所有的处理器即可。此外，使用发应堆后可以让代码更为模块化，易于重用。

- 我们逐个模块进行调试，只有某个模块能够正常运行时才会进入下一步。由于使用了四五个套接字，因此调试的工作量是很大的。我直接将调试信息输出到了屏幕上，因为实在没有必要专门开一个调试器来工作。

- 因为一直在使用valgrind工具进行测试，因此我能确定程序没有内存泄漏的问题。在C语言中，内存泄漏是我们非常关心的问题，因为没有什么垃圾回收机制可以帮你完成。正确地使用像kvmsg、czmq之类的抽象层可以很好地避免内存泄漏。

这段程序肯定还会存在一些BUG，部分读者可能会帮助我调试和修复，我在此表示感谢。

测试模型6时，先开启主机和备机，再打开一组客户端，顺序随意。随机地中止某个服务进程，如果程序设计得是正确的，那客户端获得的数据应该都是一致的。

克隆模式协议

花费了那么多精力来开发一套可靠的发布-订阅模式机制，我们当然希望将来能够方便地在其基础之上进行扩展。较好的方法是将其编写为一个协议，这样就能让各种语言来实现它了。

我们将其称为“集群化哈希表协议”，这是一个能够跨集群地进行键值哈希表管理，提供了多客户端的通信机制；客户端可以只操作一个子树的数据，包括更新和定义瞬间值。

- http://rfc.zeromq.org/spec:12