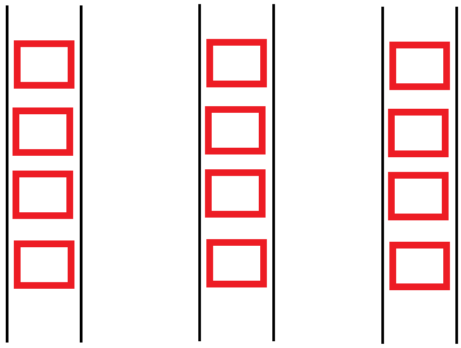
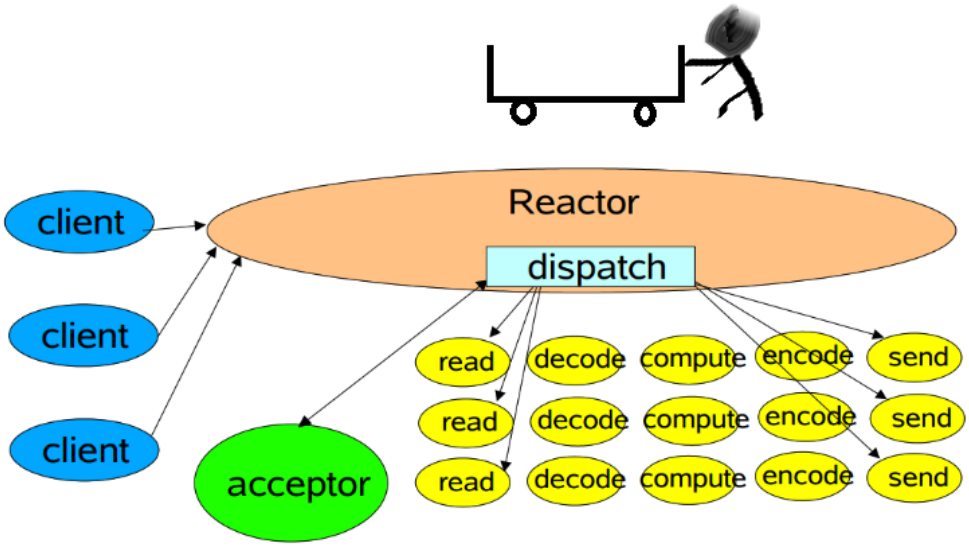
零声教育 Mark 老师 QQ: 2548898954

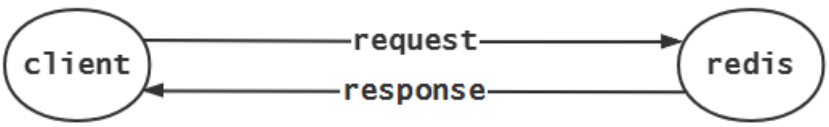
redis 网络层

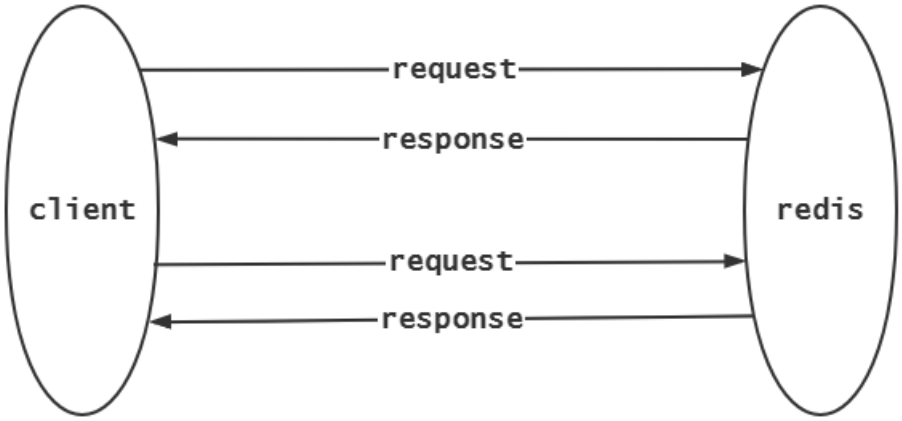


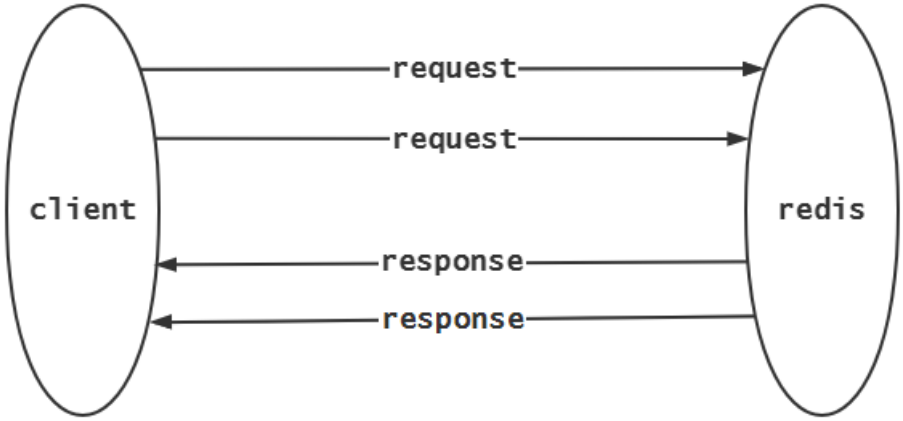


redis pipeline

> *redis pipeline* 是一个客户端提供的机制，而不是服务端提供的;
>
> *pipeline* 不具备事务性







redis事务

> 事务：用户定义一系列数据库操作，这些操作视为一个完整的**逻辑处理工作单元**，要么全部执行，要么全部不执行，是不可分割的工作单元。
>
> `MULTI` 开启事务，事务执行过程中，单个命令是入队列操作，直到调用 `EXEC` 才会一起执行；

MULTI

> 开启事务
>
> begin / start transaction

EXEC

> 提交事务
>
> commit

DISCARD

> 取消事务
>
> rollback

WATCH

> 检测 *key* 的变动，若在事务执行中，*key* 变动则取消事务；在事务开启前调用，乐观锁实现（cas）；
>
> 若被取消则事务返回 `nil`;

应用

事务实现 `zpop`

```
WATCH zset
element = ZRANGE zset 0 0
MULTI
ZREM zset element
EXEC
```

事务实现 加倍操作

```
WATCH score:10001
val = GET score:10001
MULTI
SET score:10001 val*2
EXEC
```

lua 脚本

> *lua* 脚本实现原子性；
>
> *redis* 中加载了一个 *lua* 虚拟机；用来执行 *redis lua* 脚本；*redis lua* 脚本的执行是原子性的；当某个脚本正在执行的时候，不会有其他命令或者脚本被执行；
>
> *lua* 脚本当中的命令会直接修改数据状态；
>
> *lua* 脚本 *mysql* 存储区别：*MySQL*存储过程不具备事务性，所以也不具备原子性；
>
> **注意**：如果项目中使用了 *lua* 脚本，不需要使用上面的事务命令；

```
# 从文件中读取 lua脚本内容
cat test1.lua | redis-cli script load --pipe
# 加载 lua脚本字符串 生成 sha1
> script load 'local val = KEYS[1]; return val'
"b8059ba43af6ffe8bed3db65bac35d452f8115d8"
# 检查脚本缓存中，是否有该 sha1 散列值的lua脚本
> script exists
"b8059ba43af6ffe8bed3db65bac35d452f8115d8"
1) (integer) 1
# 清除所有脚本缓存
> script flush
OK
# 如果当前脚本运行时间过长，可以通过 script kill 杀死当前运行的脚本
> script kill
(error) NOTBUSY No scripts in execution right now.
```

EVAL

```
# 测试使用
EVAL script numkeys key [key ...] arg [arg ...]
```

EVALSHA

```
# 线上使用
EVALSHA sha1 numkeys key [key ...] arg [arg ...]

```

应用

```
# 1: 项目启动时，建立redis连接并验证后，先加载所有项目中使用的lua脚本（script load）；
# 2: 项目中若需要热更新，通过redis-cli script flush；然后可以通过订阅发布功能通知所有服务器重新加载lua脚本；
# 3：若项目中lua脚本发生阻塞，可通过script kill暂停当前阻塞脚本的执行；
```

ACID特性分析

> A 原子性；事务是一个不可分割的工作单位，事务中的操作要么全部成功，要么全部失败；*redis* 不支持回滚；即使事务队列中的某个命令在执行期间出现了错误，整个事务也会继续执行下去，直到将事务队列中的所有命令都执行完毕为止。
>
> C 一致性；事务的前后，所有的数据都保持一个一致的状态，不能违反数据的一致性检测；这里的一致性是指预期的一致性而不是异常后的一致性；所以 *redis* 也不满足；这个争议很大：*redis* 能确保事务执行前后的数据的完整约束；但是并不满足业务功能上的一致性；比如转账功能，一个扣钱一个加钱；可能出现扣钱执行错误，加钱执行正确，那么最终还是会加钱成功；系统凭空多了钱；

> `I` 隔离性；各个事务之间互相影响的程度；*redis* 是单线程执行，天然具备隔离性；
>
> `D` 持久性；*redis* 只有在 `aof` 持久化策略的时候，并且需要在 `redis.conf` 中 `appendfsync=always` 才具备持久性；实际项目中几乎不会使用 `aof` 持久化策略；

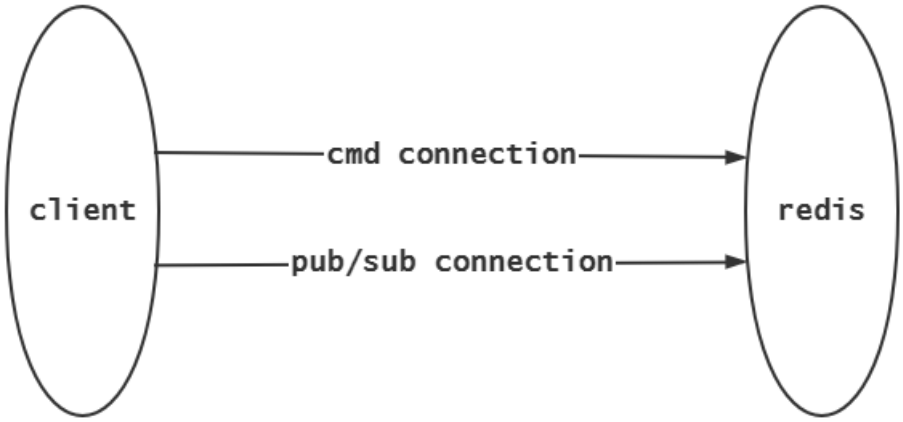
redis 发布订阅

> 为了支持消息的多播机制，*redis* 引入了发布订阅模块；
>
> 消息不一定可达；分布式消息队列； *stream* 的方式确保一定可达；

```
# 订阅频道
subscribe 频道
# 订阅模式频道
psubscribe 频道
# 取消订阅频道
unsubscribe 频道
# 取消订阅模式频道
punsubscribe 频道
# 发布具体频道或模式频道的内容
publish 频道 内容
# 客户端收到具体频道内容
message 具体频道 内容
# 客户端收到模式频道内容
pmessage 模式频道 具体频道 内容
```

应用

> 发布订阅功能一般要区别命令连接重新开启一个连接；因为命令连接严格遵循请求回应模式；而 *pubsub* 能收到 *redis* 主动推送的内容；所以实际项目中如果支持 *pubsub* 的话，需要**另开一条连接**用于处理发布订阅；



缺点

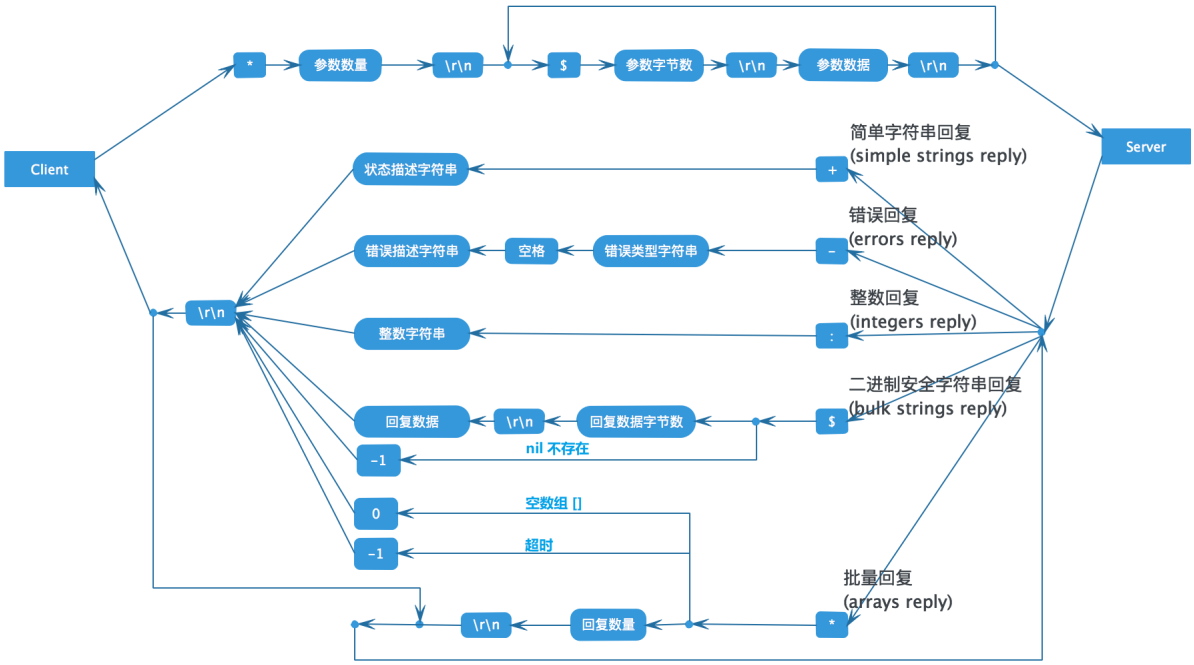
> 发布订阅的生产者传递过来一个消息，redis 会直接找到相应的消费者并传递过去；假如没有消费者，消息直接丢弃；假如开始有2个消费者，一个消费者突然挂掉了，另外一个消费者依然能收到消息，但是如果刚挂掉的消费者重新连上后，在断开连接期间的消息对于该消费者来说彻底丢失了；
>
> 另外，*redis* 停机重启，*pubsub* 的消息是不会持久化的，所有的消息被直接丢弃；

应用

```
subscribe news.it news.showbiz news.car
psubscribe news.*
publish new.showbiz 'king kiss darren'
```

redis异步连接

redis协议图



> 协议实现的第一步需要知道如何界定数据包：
>
> 1. 长度 + 二进制流
> 2. 二进制流 + 特殊分隔符

异步连接

> 同步连接方案采用阻塞 *io* 来实现；优点是代码书写是同步的，业务逻辑没有割裂；缺点是阻塞当前线程，直至 *redis* 返回结果；通常用多个线程来实现线程池来解决效率问题；
>
> 异步连接方案采用非阻塞 *io* 来实现；优点是没有阻塞当前线程，*redis* 没有返回，依然可以往 *redis* 发送命令；缺点是代码书写是异步的（回调函数），业务逻辑割裂，可以通过协程解决（*openresty*，*skynet*）；配合 *redis6.0* 以后的 *io* 多线程（前提是有大量并发请求），异步连接池，能更好解决应用层的数据访问性能；

实现方案

hiredis + libevent

```c
/* Context for a connection to Redis */
typedef struct redisContext {
    const redisContextFuncs *funcs;   /* Function table */

    int err; /* Error flags, 0 when there is no error */
    char errstr[128]; /* String representation of error when applicable */
    redisFD fd;
    int flags;
    char *obuf; /* Write buffer */
    redisReader *reader; /* Protocol reader */

    enum redisConnectionType connection_type;
    struct timeval *connect_timeout;
    struct timeval *command_timeout;

    struct {
        char *host;
        char *source_addr;
        int port;
    } tcp;

    struct {
        char *path;
    } unix_sock;

    /* For non-blocking connect */
    struct sockadr *saddr;
    size_t addrlen;

    /* Optional data and corresponding destructor users can use to provide
```

```
31      * context to a given redisContext.  Not used
by hiredis. */
32     void *privdata;
33     void (*free_privdata)(void *);
34 
35     /* Internal context pointer presently used by
hiredis to manage
36      * SSL connections. */
37     void *privctx;
38 
39     /* An optional RESP3 PUSH handler */
40     redisPushFn *push_cb;
41 } redisContext;
42 
43 static int redisLibeventAttach(redisAsyncContext
*ac, struct event_base *base) {
44     redisContext *c = &(ac->c);
45     redisLibeventEvents *e;
46 
47     /* Nothing should be attached when something
is already attached */
48     if (ac->ev.data != NULL)
49         return REDIS_ERR;
50 
51     /* Create container for context and r/w
events */
52     e = (redisLibeventEvents*)hi_calloc(1,
sizeof(*e));
53     if (e == NULL)
54         return REDIS_ERR;
55 
56     e->context = ac;
57 
58     /* Register functions to start/stop listening
for events */
59     ac->ev.addRead = redisLibeventAddRead;
60     ac->ev.delRead = redisLibeventDelRead;
```

```
61      ac->ev.addWrite = redisLibeventAddWrite;
62      ac->ev.delWrite = redisLibeventDelWrite;
63      ac->ev.cleanup = redisLibeventCleanup;
64      ac->ev.scheduleTimer =
    redisLibeventSetTimeout;
65      ac->ev.data = e;
66
67      /* Initialize and install read/write events
    */
68      e->ev = event_new(base, c->fd, EV_READ |
    EV_WRITE, redisLibeventHandler, e);
69      e->base = base;
70      return REDIS_OK;
71  }
```

原理

> *hiredis* 提供异步连接方式，提供**可以替换 IO 检测**的接口；
>
> 关键替换 `addRead`，`delRead`，`addWrite`，`delWrite`，`cleanup`，`scheduleTimer`，这几个检测接口；其他 io 操作，比如 `connect`，`read`，`write`，`close`等都交由 *hiredis* 来处理；
>
> 同时需要提供连接建立成功以及断开连接的回调；
>
> 用户可以使用当前项目的网络框架来替换相应的操作；从而实现跟项目网络层兼容的异步连接方案；

自定义实现

> 有时候，用户除了需要与项目网络层兼容，同时需要考虑与项目中数据结构契合；这个时候可以考虑自己实现 *redis* 协议，从解析协议开始转换成项目中的数据结构；

> 下面代码是 Mark 老师在之前项目中的实现；之前项目中实现了一个类似 lua 中 *table* 的数据对象（SVar），所以希望操作 *redis* 的时候，希望直接传 `SVar` 对象，然后在协议层进行转换；

协议解压缩

```
static bool
readline(u_char *start, u_char *last, int &pos)
{
    for (pos = 0; start+pos <= last-1; pos++) {
        if (start[pos] == '\r' && start[pos+1] == '\n') {
            pos--;
            return true;
        }
    }
    return false;
}

/*
-2  包解析错误
-1  未读取完整的包
 0  正确读取
 1  是错误信息
*/
static int
read_sub_response(u_char *start, u_char *last, SVar &s, int &usz)
{
    int pos = 0;

    if (!readline(start, last, pos))
        return -1;
    u_char *tail = start+pos+1; //
    u_char ch = start[0];
    usz += pos+2+1; // pos+1 + strlen("\r\n")
```

```
29
30      switch (ch)
31      {
32      case '$':
33          {
34              string str(start+1, tail);
35              int len = atoi(str.c_str());
36              if (len < 0) return 0; // nil
37              if (tail+2+len > last) return -1;
38              s = string(tail+2, tail+2+len);
39              usz += len+2;
40              return 0;
41          }
42      case '+':
43          {
44              s = string(start+1, tail);
45              return 0;
46          }
47      case '-':
48          {
49              s = string(start+1, tail);
50              return 1;
51          }
52      case ':':
53          {
54              string str(start+1, tail);
55              s = atof(str.c_str());
56              return 0;
57          }
58      case '*':
59          {
60              string str(start+1, tail);
61              int n = atoi(str.c_str());
62              if (n == 0) return 0; // 空数组
63              if (n < 0) return 0; // 超时
64              int ok = 0;
65              u_char *pt = tail+2;
```

```
            for (int i=0; i<n; i++) {
                if (pt > last) return -1;
                int sz = 0;
                SVar t;
                int ret = read_sub_response(pt, last, t, sz);
                if (ret < 0) return -1;
                s.Insert(t);
                usz += sz;
                pt += sz;
                if (ret == 1) ok = 1;
            }
            return ok;
        }
    }
    return -2;
}

static int
read_response(SHandle *pHandle, SVar &s, int &size)
{
    int len = pHandle->GetCurBufSize();
    u_char *start = pHandle->m_pBuffer;
    u_char *last = pHandle->m_pBuffer+len;
    return read_sub_response(start, last, s, size);
}
```

协议压缩

```
static void
write_header(string &req, size_t n)
{
    char chv[16] = {0};
    _itoa(n, chv, 10);
    req.append("\r\n$");
```

```
    req.append(chv);
    req.append("\r\n");
}

static void
write_count(string &req, size_t n)
{
    char chv[16] = {0};
    _itoa(n, chv, 10);
    req.append("*");
    req.append(chv);
}

static void
write_command(string &req, const char *cmd)
{
    int n = strlen(cmd);
    write_header(req, n);
    req.append(cmd);
    //req.append("\r\n");
}

void SRedisClient::RunCommand(const char* cmd,
vector<string> &params)
{
    string req;
    size_t nsize = params.size();
    write_count(req, nsize+1);
    write_command(req, cmd);
    for (size_t i = 0; i < params.size(); i++) {
        size_t n = params[i].size();
        write_header(req, n);
        req.append(params[i]);
    }
    req.append("\r\n");
    Send(req);
}
```