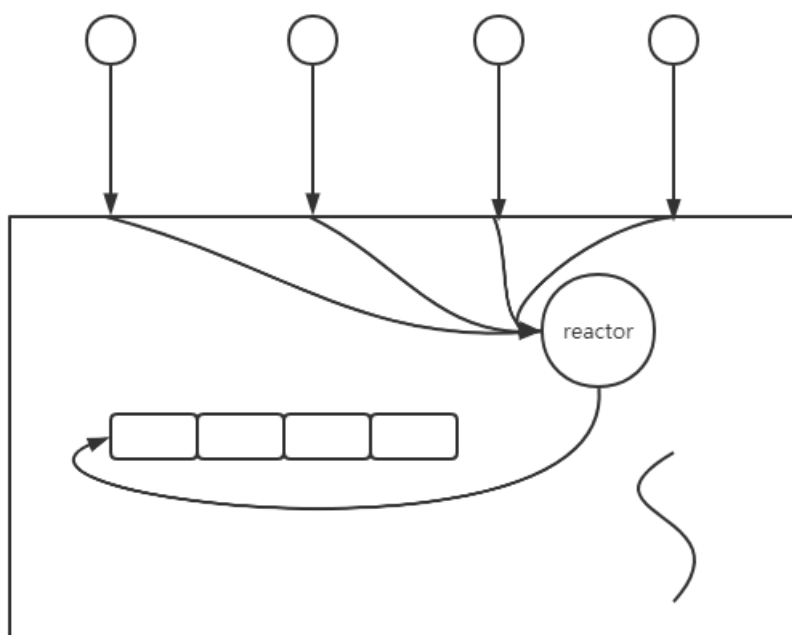


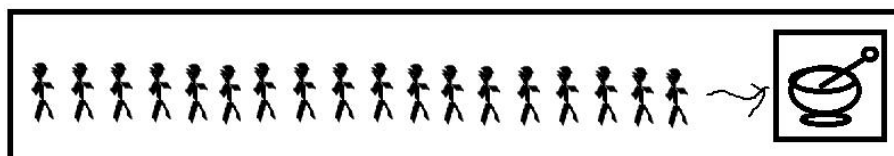
## redis 网络层



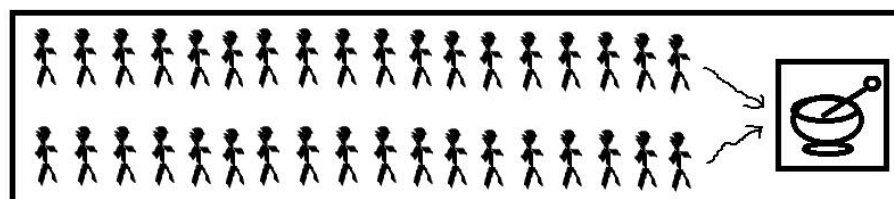
对于所有连接的数据处理，redis 并发执行的；

对于单条连接的数据处理，redis 串行执行的；

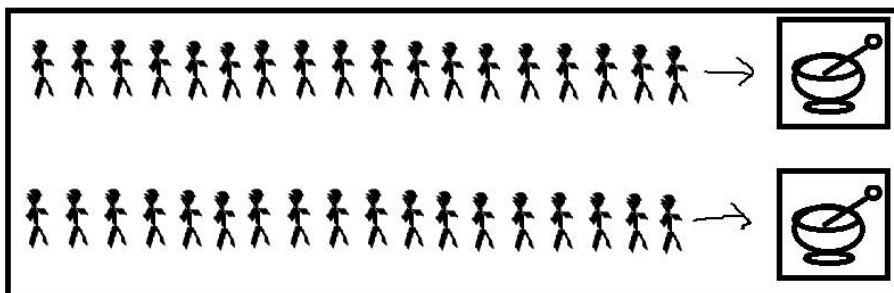
串行



并发



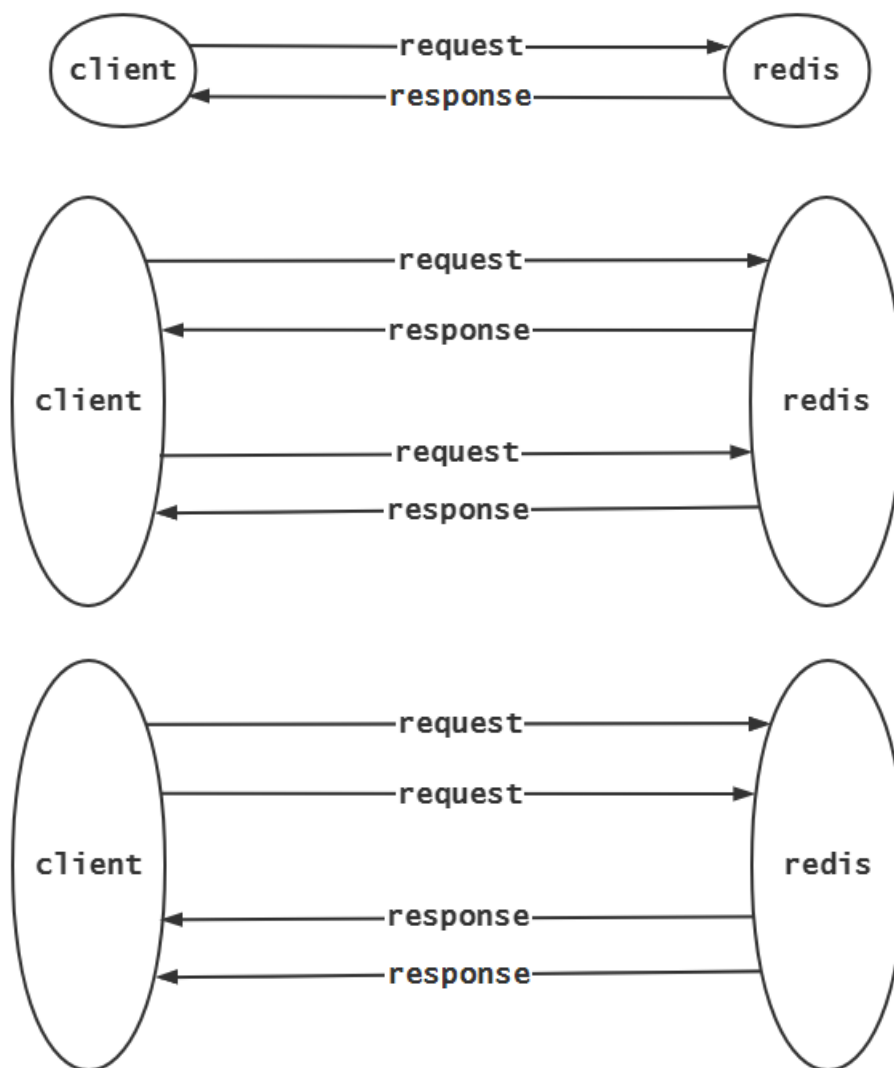
并行



并发：活跃队列的个数大于处理器的个数；

## redis pipeline

redis pipeline 是一个客户端提供的，而不是服务端提供的；



对于request操作，只是将数据写到fd对应的写缓冲区，时间非常快，真正耗时操作在读取response；

## redis事务

`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

```
1 WATCH zset
2 element = ZRANGE zset 0 0
3 MULTI
4 ZREM zset element
5 EXEC
```

### 事务实现 加倍操作

```
1 WATCH score:10001
2 val = GET score:10001
3 MULTI
4 SET score:10001 val*2
5 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；

**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；

```
1 # 从文件中读取 lua脚本内容
2 cat test1.lua | redis-cli script load --pipe
3 # 加载 lua脚本字符串 生成 sha1
4 > script load 'local val = KEYS[1]; return val'
5 "b8059ba43af6ffe8bed3db65bac35d452f8115d8"
6 # 检查脚本缓存中，是否有该 sha1 散列值的lua脚本
7 > script exists "b8059ba43af6ffe8bed3db65bac35d452f8115d8"
8 1) (integer) 1
9 # 清除所有脚本缓存
10 > script flush
11 OK
12 # 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
13 > script kill
14 (error) NOTBUSY No scripts in execution right now.
```

## EVAL

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

## EVALSHA

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

## 应用

```
1 # 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 # 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 # 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；事务使数据库从一个一致性状态到另外一个一致性状态；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；但是并不满足传统意义上的一致性；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；事务的操作不被其他用户操作所打断；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；

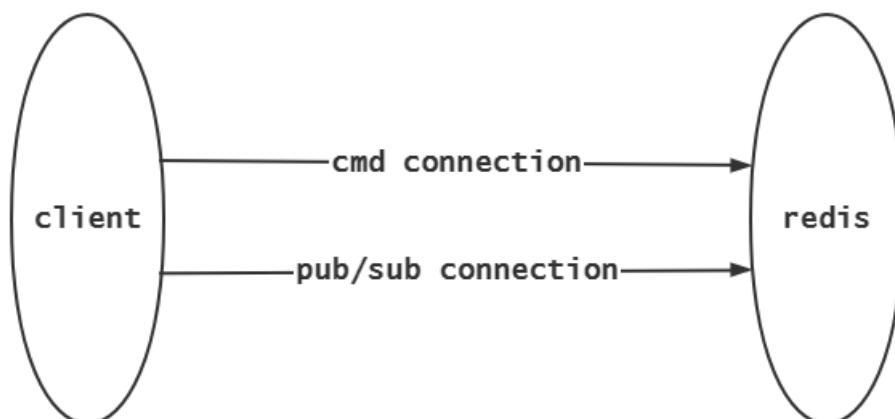
## redis 发布订阅

为了支持消息的多播机制，redis 引入了发布订阅模块；

```
1 # 订阅频道
2 subscribe 频道
3 # 订阅模式频道
4 psubscribe 频道
5 # 取消订阅频道
6 unsubscribe 频道
7 # 取消订阅模式频道
8 punsubscribe 频道
9 # 发布具体频道或模式频道的内容
10 publish 频道 内容
11 # 客户端收到具体频道内容
12 message 具体频道 内容
13 # 客户端收到模式频道内容
14 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；

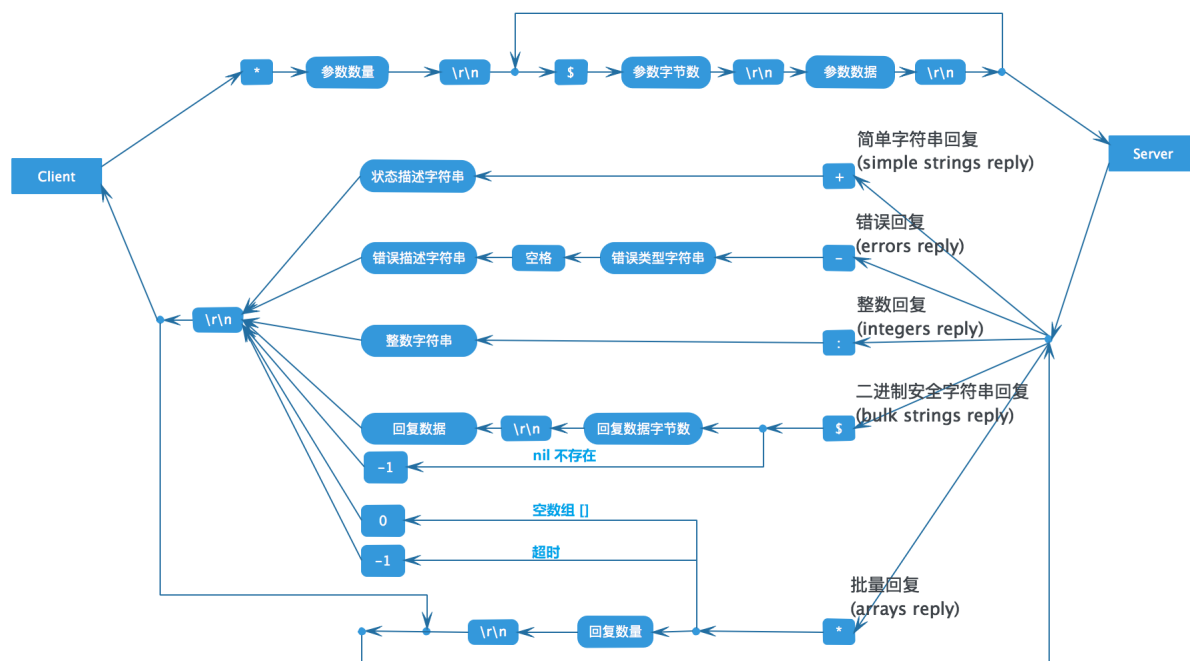
另外，redis停机重启，pubsub的消息是不会持久化的，所有的消息被直接丢弃；

## 应用

```
1 | subscribe news.it news.showbiz news.car
2 | psubscribe news.*
3 | 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；

## redis6.0 io多线程

redis6.0版本后添加的 io多线程主要解决redis协议的压缩以及解压缩的耗时问题；一般项目中不需要开启；如果有大量并发请求，且返回数据包一般比较大的场景才有它的用武之地；

### 原理

```

1  int n = read(fd, buff, size); // redis io-threads
2  msg = decode(buff, size);
3  data = do_command(msg);
4  bin = encode(data, sz); // io-threads
5  send(fd, bin, sz1);

```

### 开启

```

1 # 在 redis.conf 中
2 # if you have a four cores boxes, try to use 2 or 3 I/O threads, if you have
  a 8 cores, try to use 6 threads.
3 io-threads 4
4 # 默认只开启 encode 也就是redis发送给客户端的协议压缩工作; 也可开启io-threads-do-reads
  yes来实现 decode;
5 # 一般发送给redis的命令数据包都比较少, 所以不需要开启 decode 功能;
6 # io-threads-do-reads no

```

## 实现方案

### hiredis + libevent

```

1 /* Context for a connection to Redis */
2 typedef struct redisContext {
3     const redisContextFuncs *funcs; /* Function table */
4
5     int err; /* Error flags, 0 when there is no error */
6     char errstr[128]; /* String representation of error when applicable */
7     redisFD fd;
8     int flags;
9     char *obuf; /* Write buffer */
10    redisReader *reader; /* Protocol reader */
11
12    enum redisConnectionType connection_type;
13    struct timeval *connect_timeout;
14    struct timeval *command_timeout;
15
16    struct {
17        char *host;
18        char *source_addr;
19        int port;
20    } tcp;
21
22    struct {
23        char *path;
24    } unix_sock;
25
26    /* For non-blocking connect */
27    struct sockadr *saddr;
28    size_t addrlen;
29
30    /* 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，提供可以替换网络检测的接口；

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

```

1 static bool
2 readline(u_char *start, u_char *last, int &pos)
3 {
4     for (pos = 0; start+pos <= last-1; pos++) {

```



```

5         if (start[pos] == '\r' && start[pos+1] == '\n') {
6             pos--;
7             return true;
8         }
9     }
10    return false;
11 }
12
13 /*
14 -2 包解析错误
15 -1 未读取完整的包
16 0 正确读取
17 1 是错误信息
18 */
19 static int
20 read_sub_response(u_char *start, u_char *last, svar &s, int &usz)
21 {
22     int pos = 0;
23
24     if (!readline(start, last, pos))
25         return -1;
26     u_char *tail = start+pos+1; //
27     u_char ch = start[0];
28     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;
66         for (int i=0; i<n; i++) {
67             if (pt > last) return -1;
68             int sz = 0;
69             SVar t;
70             int ret = read_sub_response(pt, last, t, sz);
71             if (ret < 0) return -1;
72             s.Insert(t);
73             usz += sz;
74             pt += sz;
75             if (ret == 1) ok = 1;
76         }
77         return ok;
78     }
79 }
80 return -2;
81 }
82
83 static int
84 read_response(SHandle *pHandle, SVar &s, int &size)
85 {
86     int len = pHandle->GetCurBufSize();
87     u_char *start = pHandle->m_pBuffer;
88     u_char *last = pHandle->m_pBuffer+len;
89     return read_sub_response(start, last, s, size);
90 }

```

## 协议压缩

```

1  static void
2  write_header(string &req, size_t n)
3  {
4      char chv[16] = {0};
5      _itoa(n, chv, 10);
6      req.append("\r\n$");
7      req.append(chv);
8      req.append("\r\n");
9  }
10
11 static void
12 write_count(string &req, size_t n)
13 {
14     char chv[16] = {0};
15     _itoa(n, chv, 10);
16     req.append("*");
17     req.append(chv);
18 }
19
20 static void
21 write_command(string &req, const char *cmd)
22 {
23     int n = strlen(cmd);
24     write_header(req, n);
25     req.append(cmd);
26     //req.append("\r\n");

```

```
27 }
28
29 void SRedisClient::RunCommand(const char* cmd, vector<string> &params)
30 {
31     string req;
32     size_t nsize = params.size();
33     write_count(req, nsize+1);
34     write_command(req, cmd);
35     for (size_t i = 0; i < params.size(); i++) {
36         size_t n = params[i].size();
37         write_header(req, n);
38         req.append(params[i]);
39     }
40     req.append("\r\n");
41     Send(req);
42 }
```