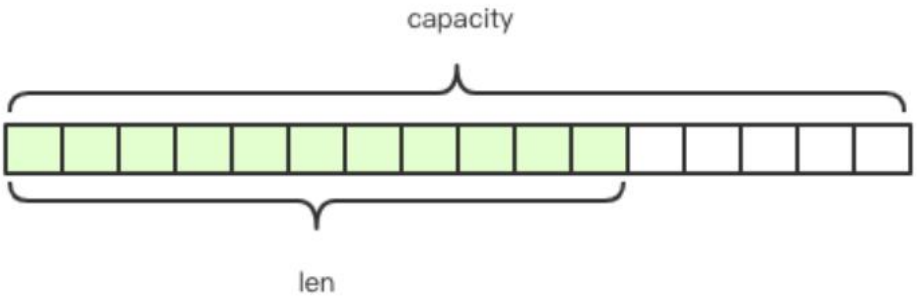
Redis 阅读笔记

数据结构

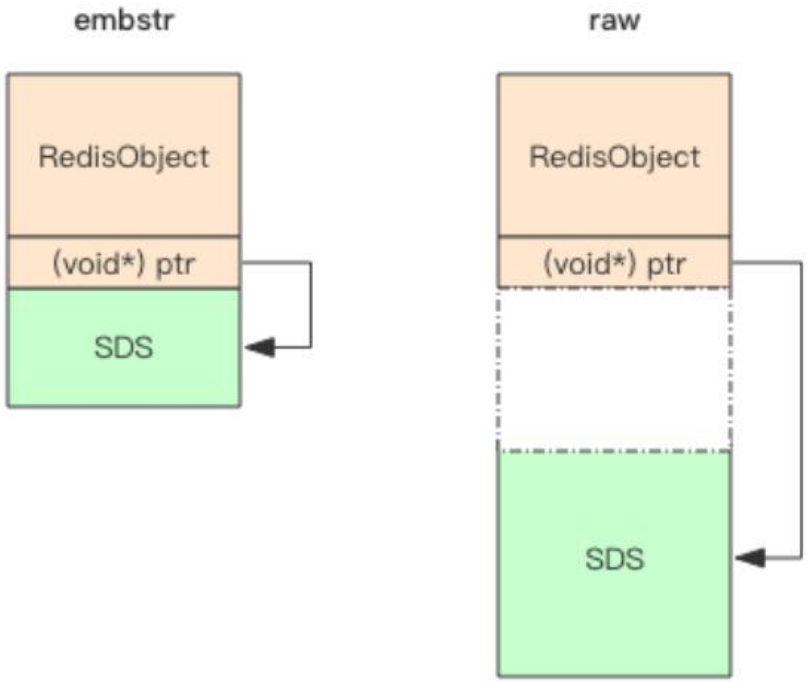
String

Redis 的字符串叫「SDS」,它的结构是一个带长度信息的字节数组。

```
struct SDS<T> {
    T capacity; // 数组容量  最小 1byte
    T len; // 数组长度  最小 1byte
    byte flags; // 特殊标识位，不理睬它 最小 1byte
    byte[] content; // 数组内容
}
```



Redis 的字符串有两种存储方式，在长度特别短时，使用 emb 形式存储(embeded)，当长度超过 44 时，使用 raw 形式存储。



所有的 Redis 对象都有下面的这个结构头(key)

```
struct RedisObject {
    int4 type; // 4bits
    int4 encoding; // 4bits
    int24 lru; // 24bits
    int32 refcount; // 4bytes
    void *ptr; // 8bytes，64-bit system
} robj;
```

内存分配器 jemalloc/tcmalloc 等分配内存大小的单位都是 2、4、8、16、32、64 等, emb 结构本身占用了（4+4+24）/8+4+8+1+1+1=19 content 以字节\0 结尾占用一个字节 所以只剩余 44 字节

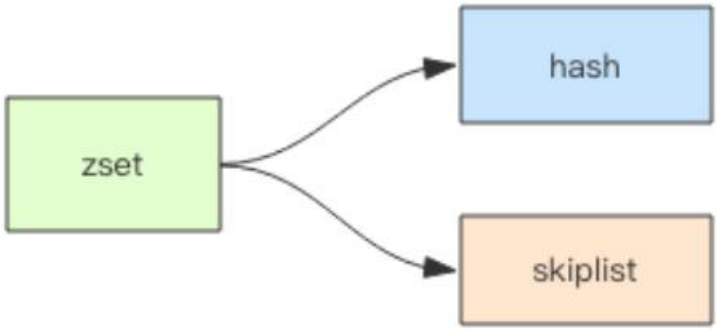
Set

当数据量少时使用 intset(整数数组)实现，当数据量过多时使用 hash 表实现

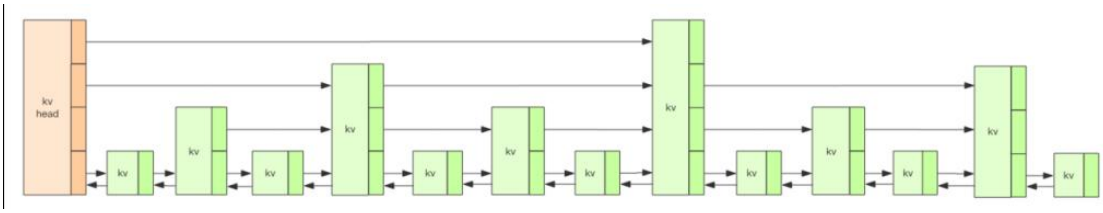
```
struct intset<T> {
    int32 encoding; // 决定整数位宽是 16 位、32 位还是 64 位
    int32 length; // 元素个数
    int<T> contents; // 整数数组，可以是 16 位、32 位和 64 位
}
```

Zset

Redis 的 zset 是一个复合结构，一方面它需要一个 hash 结构来存储 value 和 score 的 对应关系，另一方面需要 zskiplist 来按照 score 来排序




```
struct zset {
    dict *dict; // all values value=>score
    zskiplist *zsl;
}
```



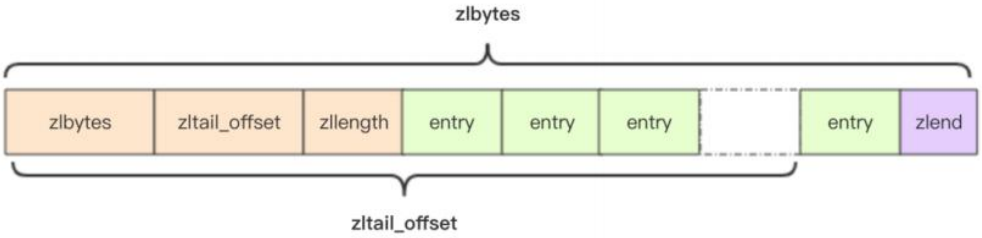

```
struct zsl {
    zslnode* header; // 跳跃列表头指针
    int maxLevel; // 跳跃列表当前的最高层
    map<string, zslnode*> ht; // hash 结构的所有键值对
}
struct zslnode {
    string value;
    double score;
    zslnode*[] forwards; // 多层连接指针
    zslnode* backward; // 回溯指针
}
```

List

当数据量较少时使用 ziplist 实现，当数据量大时使用 quicklist 存储

```
struct ziplist<T> {
    int32 zlbytes; // 整个压缩列表占用字节数
    int32 zltail_offset; // 最后一个元素距离压缩列表起始位置的偏移量，用于快速定位到
    最后一个节点
```

```
    int16 zllength; // 元素个数
    T[] entries; // 元素内容列表，挨个挨个紧凑存储
    int8 zlend; // 标志压缩列表的结束，值恒为 0xFF
}
```



压缩列表为了支持双向遍历，所以才会有 ztail_offset 这个字段，用来快速定位到最后一 个元素，然后倒着遍历。

entry 块随着容纳的元素类型不同，也会有不一样的结构。

```
struct entry {
    int<var> prevlen; // 前一个 entry 的字节长度
    int<var> encoding; // 元素类型编码
    optional byte[] content; // 元素内容
}

struct quicklist {
    quicklistNode* head;
    quicklistNode* tail;
    long count; // 元素总数
    int nodes; // ziplist 节点的个数
    int compressDepth; // LZF 算法压缩深度
    ...
}

struct quicklistNode {
    quicklistNode* prev;
    quicklistNode* next;
    ziplist* zl; // 指向压缩列表
    int32 size; // ziplist 的字节总数
    int16 count; // ziplist 中的元素数量
    int2 encoding; // 存储形式 2bit，原生字节数组还是 LZF 压缩存储
    ...
}
```

Hash

当数据量较少时使用 ziplist 实现，当数据量大时使用哈希表存储

扩容条件：hash 表中元素的个数等于数组的长度时扩容，如果 Redis 正在做 bgsave 则元素的个数已经达到了数组长度的 5 倍时扩容

缩容条件：元素个数低于数组长度的 10%缩容

```
struct dict {
    ...
    dictht ht[2];
}
struct dictht {
    dictEntry** table; // 二维
    long size; // 第一维数组的长度
    long used; // hash 表中的元素个数
    ...
}
struct dictEntry {
    void* key;
    void* val;
    dictEntry* next; // 链接下一个 entry
}
```

位图

- **统计用户一年的签到次数**
  每个用户一个 bitmap,通过 bitcount 统计出某用户本年度签到次数

- **统计上亿用户本周连续签到用户数**
  用户一天用同一个 bitmap，7 个 bitmap 通过 BITOP operation destkey key [key ...] 得到一个 bitmap，opration 可以是 and、OR、NOT、XOR 再通过 bitcount 获取连续签到人数

- **布隆过滤器**
  将所有数据通过几种 hash 函数然后对数组长度取模映射到 bitmap 上 ，存在的数据一定能找到，部分不存在的数据因为 hash 冲突过滤不掉

HyperLogLog

统计网页每天的 UV 数据
通过 pfadd codehole user1 添加计数 pfcount codehole 获取计数 Pfmerge 合并页面计数

GeoHash

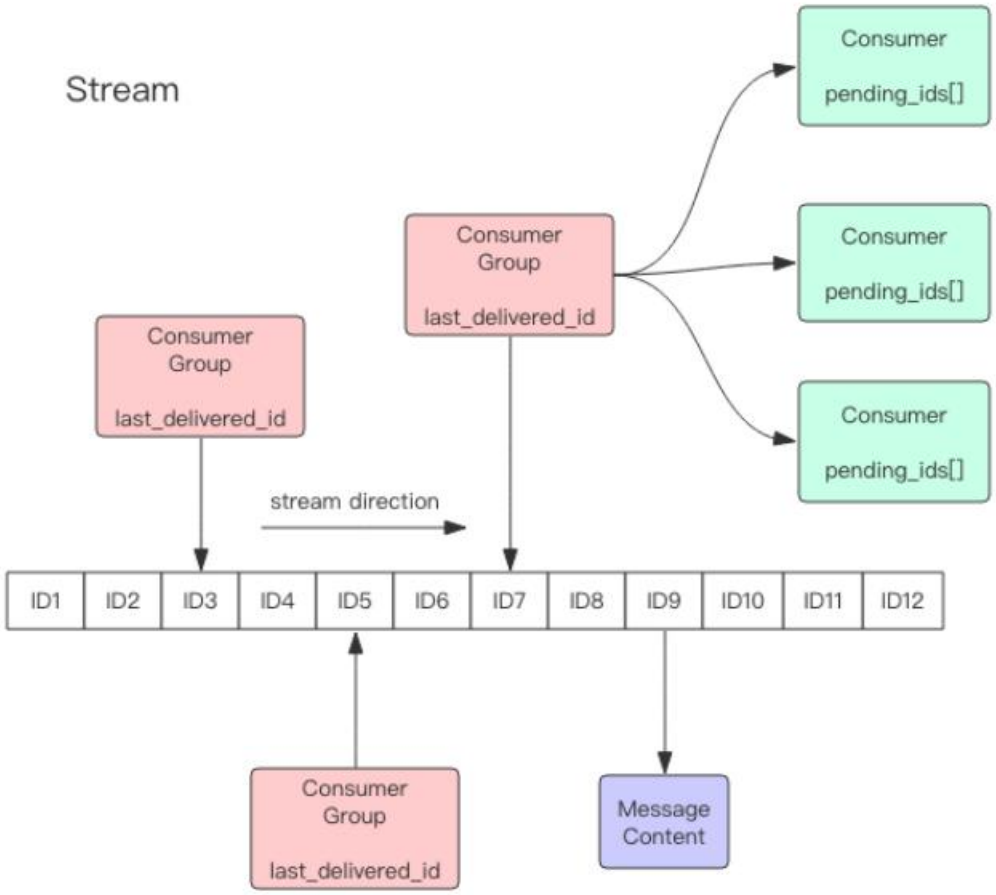
Redis 在 3.2 版本以后增加了地理位置 GEO 模块，意味着我们可以使用 Redis 来实现 摩拜单车「附近的 Mobike」、美团和饿了么「附近的餐馆」这样的功能了。

```
#添加位置
127.0.0.1:6379> geoadd company 116.48105 39.996794 juejin
(integer) 1
127.0.0.1:6379> geoadd company 116.514203 39.905409 ireader
(integer) 1
127.0.0.1:6379> geoadd company 116.489033 40.007669 meituan
#计算距离
127.0.0.1:6379> geodist company juejin ireader km
"10.5501"
127.0.0.1:6379> geodist company juejin meituan km

# 范围 20 公里以内最多 3 个元素按距离正排，它不会排除自身（按地名）
127.0.0.1:6379> georadiusbymember company ireader 20 km count 3 asc
1) "ireader"
2) "juejin"
3) "meituan"
# 范围 20 公里以内最多 3 个元素按距离正排，它不会排除自身（按坐标）
127.0.0.1:6379> georadius company 116.514202 39.905409 20 km withdist count 3 asc
1) 1) "ireader"
2) "0.0000
```

消息

- List
  通过 LPUSH、BRPOP、或者 RPUSH、BLPOP 实现
  没有 ack 机制
- PubSub
  消费者可以订阅一个或多个 channel
  没有数据持久化，无消费者或消费者下线消息丢弃
- Stream



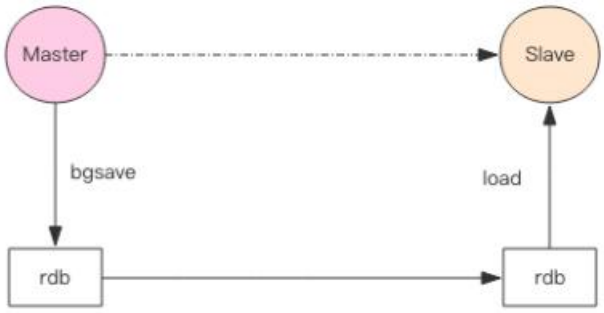
1、xadd 追加消息
2、xdel 删除消息，这里的删除仅仅是设置了标志位，不影响消息总长度
3、xrange 获取消息列表，会自动过滤已经删除的消息
4、xlen 消息长度
5、del 删除 Stream

每个 Stream 都可以挂多个消费组，每个消费组会有个游标 last_delivered_id 在 Stream 数组之上往前移动，表示当前消费组已经消费到哪条消息了。每个消费组都有一个 Stream 内唯一的名称，消费组不会自动创建，它需要单独的指令 xgroup create 进行创建，需要指定 从 Stream 的某个消息 ID 开始消费，这个 ID 用来初始化 last_delivered_id 变量。

每个消费组 (Consumer Group) 的状态都是独立的，相互不受影响。也就是说同一份 Stream 内部的消息会被每个消费组都消费到。

主从哨兵集群

主从



- **全量复制**

slave 服务器第一次连接到 master 服务器，开始进行数据同步，发送 psync 命令。

master 服务器收到 psync 命令之后，返回给从库主库 runID 和主库目前的复制进度 offset，同时开始执行 bgsave 命令生成 RDB 快照文件并使用缓存区记录此后执行的所有写命令。

master 服务器 bgsave 执行完之后，就会向 Slava 服务器发送快照文件，并在发送期间继续在缓冲区内记录被执行的写命令。

slave 服务器收到 RDB 快照文件后，会将接收到的数据写入磁盘，然后清空所有旧数据。

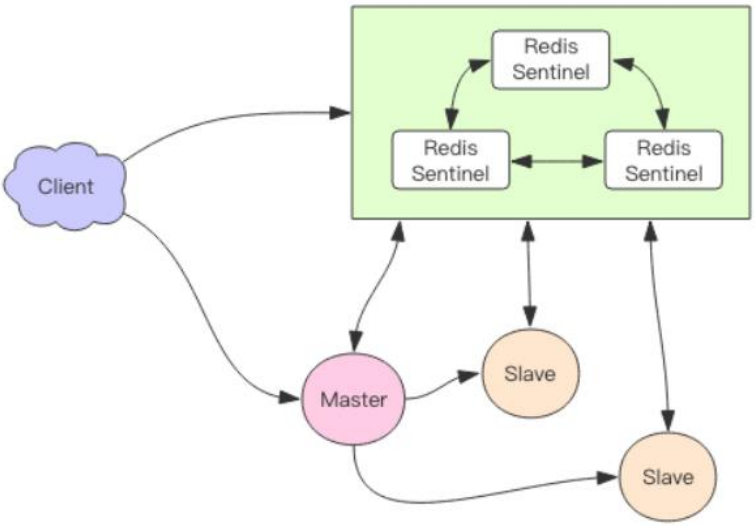
master 服务器发送完 RDB 快照文件之后，便开始向 slave 服务器发送缓冲区中的写命令。 slave 服务器完成对快照的载入，开始接受命令请求，并执行来自主服务器缓冲区的写命令，如果 offset 不在接收的写命令 offset 范围内会重新进行主从复制；

如果 slave node 开启了 AOF，那么会立即执行 BGREWRITEAOF，重写 AOF。

- **增量复制**

网络波动主从断开连接重连时会基于 repl_backlog_buffer 同步数据，它是一个环形数组数据，满了之后会从头开始覆盖之前的数据，如果从库 offset 在主库 repl_backlog_buffer 范围内就只需要同步缓冲区数据即可，可以根据网络断开时间和主库写命令数据量设置 repl_backlog_buffer 缓冲区大小

哨兵



客户端来连接集群时，会首先连接 sentinel，通过 sentinel 来查询主节点的地址， 然后再去连接主节点进行数据交互。当主节点发生故障时，客户端会重新向 sentinel 要地址，sentinel 会将最新的主节点地址告诉客户端。如此应用程序将无需重启即可自动完成节点切换

sentinel 向所有服务器发送 PING 命令，如果 slave 没有在在规定时间内响应「哨兵」的 PING 命令，就会将他记录为「主观下线状态」（PING 命令有效回复：返回 +PONG、-LOADING、-MASTERDOWN 任何一种；无效回复：有效回复之外的回复，或者指定时间内返回任何回复。）

过半的哨兵（quorum 参数）判断 master 已经「主观下线」，这时候才能将 master 标记为「客观下线」,选举成功后由哨兵通知 slave 节点连接到新的 master

主节点选举策略：首先排除掉下线的经常网络不好的，然后再按以下优先级判断 ：slave-priority 配置项优先级，slave_repl_offset 与 master_repl_offset 进度差距，slave runID 小的

哨兵之间通过 pub/sub 发布/订阅机制通信，master 有个 __sentinel__:hello 的专用通道用于哨兵之间发布和订阅消息，哨兵向 master 发送 INFO 命令来获取从节点信息

Cluster 集群

Redis Cluster 将所有数据划分为 16384 的 slots，每个节点负责其中一部分槽位，节点通过 Gossip 协议相互交互集群信息，最后每个节点都保存着其他节点的 slots 分配情况，通过 CLUSTER MEET <ip> <port> 加入新的节点

Cluster 默认会对 key 值使用 crc32 算法进行 hash 得到一个整数值，然后用这个整数 值对 16384 进行取模来得到具体槽位。

如一个节点发现某个节点失联了 (PFail)，它会将这条信息向整个集群广播，其它节点也就可 以收到这点失联信息。如果一个节点收到了某个节点失联的数量 (PFail Count) 已经达到了集 群的大多数，就可以标记该节点为确定下线状态 (Fail)，然后向整个集群广播，强迫其它节 点也接收该节点已经下线的事实，并立即对该失联节点进行主从切换

Cluster 有两个特殊的 error 指令，一个是 moved，一个是 asking

moved ：客户端指令发送到了错误的节点，该节点会将目标节点的地址随 moved 指令回复给客户端通知 ，客户端去目标节点去访问并刷新自己的槽位关系表

asking ：当前槽位正处于迁移中，如果当前旧节点存在数据，那就直接返回结果 如果不存在，就会给客户端返回一个 asking error 携带上目标节点的地址。客户端会去目标节点尝试获取数据。客户端不会刷新槽位映射关系表

分布式锁

1. setnx 成功返回 1，否则就返回 0
   应用崩溃锁不能解除
2. set(key,value,NX,EX,timeout)
   锁过期程序未执行完其它程序获取锁
   删除了其它线程获取的锁（用 lua 判断是当前线程再删除）
3. Redission

Lua 脚本获取锁

```
local key = KEYS[1]; -- 锁的 key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断是否存在
if(redis.call('exists', key) == 0) then
    -- 不存在, 获取锁
    redis.call('hset', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
-- 锁已经存在，判断 threadId 是否是自己
if(redis.call('hexists', key, threadId) == 1) then
    -- 不存在, 获取锁，重入次数+1
    redis.call('hincrby', key, threadId, '1');
```

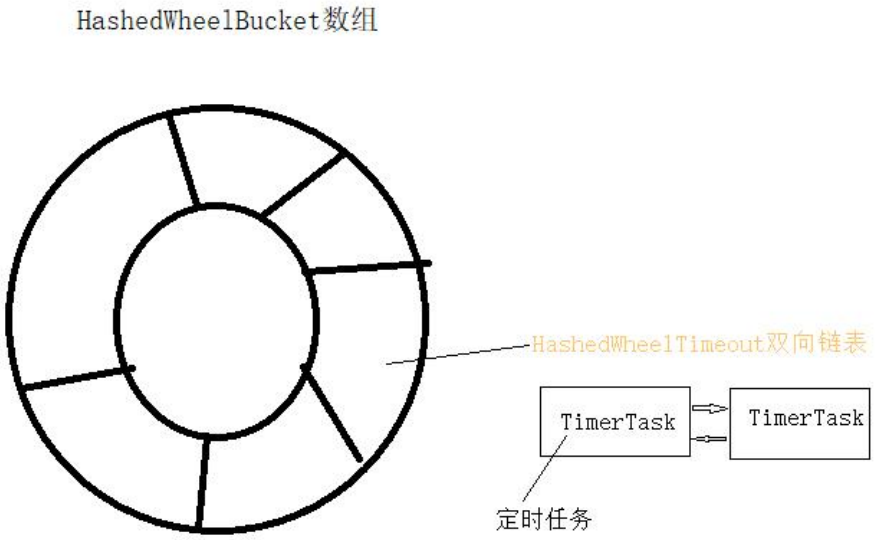
```
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
return 0;
```

Lua 脚本释放锁

```
local key = KEYS[1]; -- 锁的 key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间-- 判断当前锁是否还是被自己持有
if (redis.call('HEXISTS', key, threadId) == 0) then
    return nil; -- 如果已经不是自己，则直接返回
end;
-- 是自己的锁，则重入次数-1
local count = redis.call('HINCRBY', key, threadId, -1);
-- 判断是否重入次数是否已经为 0
if (count > 0) then
    -- 大于 0 说明不能释放锁，重置有效期然后返回
    redis.call('EXPIRE', key, releaseTime);
    return nil;
else  -- 等于 0 说明可以释放锁，直接删除
    redis.call('DEL', key);
    return nil;
end;
```

续期

每隔 10 秒（默认过期时间的 1/3）续期到过期时间 30 秒
//基于 Netty HashedWheelTimer 时间轮来执行延时任务



Timeout task = this.commandExecutor.getConnectionManager().newTimeout（TimerTask var1, long var2, TimeUnit var4)

通过 newTimeout 方法将任务添加到阻塞队列中，然后定时任务不断从队列中获取放入 HashWheelBucket 中

每次时钟转动的时候都会遍历所在数组位置的 HashedWheelTimeout 链表判断 remainingRounds 是否小于等于 0（比如一个数组位是一秒，数组大小 8 添加一个 21 秒后的任务 remainingRounds 为 2，每隔一秒执行），remainingRounds 大于 0 则 remainingRounds 减一 ，否则执行任务再重新通过 newTimeout 添加任务

持久化

- RDB

  二进制日志恢复较快，通过以下两个指令生成

  save：主线程执行，会阻塞；

  bgsave：调用 glibc 的函数 fork 产生一个子进程，基于 COW 机制写入 RDB 文件

- AOF

  AOF 日志存储的是 Redis 服务器的顺序指令序列，文件过大时可以通过 bgrewriteaof 指令对 AOF 日志进行瘦身

  AOF 写入策略：always：同步写，everysec：每秒写，no： 操作系统控制

缓存淘汰及过期删除

缓存淘汰策略

- **noeviction** 不会继续服务写请求 (DEL 请求可以继续服务)，读请求可以继续进行。
- **volatile-lru** 尝试淘汰设置了过期时间的 key，最少使用的 key 优先被淘汰。
- **volatile-ttl** 跟上面一样，除了淘汰的策略不是 LRU，而是 key 的剩余寿命 ttl 的值，ttl 越小越优先被淘汰。
- **volatile-random** 跟上面一样，不过淘汰的 key 是过期 key 集合中随机的 key
- **allkeys-lru** 区别于 volatile-lru，这个策略要淘汰的 key 对象是全体的 key 集合，而不只是过期的 key 集合。
- **allkeys-random** 跟上面一样，不过淘汰的策略是随机的 key。
- **volatile-xxx** 策略只会针对带过期时间的 key 进行淘汰，allkeys-xxx 策略会对所有的 key 进行淘汰。如果你只是拿 Redis 做缓存，那应该使用 allkeys-xxx，客户端写缓存时 不必携带过期时间。如果你还想同时使用 Redis 的持久化功能，那就使用 volatile-xxx 策略，这样可以保留没有设置过期时间的 key，它们是永久的 key 不会被 LRU 算法淘汰

主动删除

1、从过期字典中随机 20 个 key；

2、删除这 20 个 key 中已经过期的 key；

3、如果过期的 key 比率超过 1/4，那就重复步骤 1；同时，为了保证过期扫描不会出现循环过度，导致线程卡死现象，算法还增加了扫描时 间的上限，默认不会超过 25ms。

因为主动删除是主线程执行，所以尽量避免 key 在同一时间过期

懒惰删除

删除指令 del 会直接释放对象的内存，大部分情况下，这个指令非常快，没有明显延 迟。不过如果删除的 key 是一个非常大的对象，比如一个包含了千万元素的 hash，那么删除操作就会导致单线程卡顿。所以需要懒惰删除（查询时发现 key 过期时直接删除）

通信

- **通信协议**

1、单行字符串 以 + 符号开头。

2、多行字符串 以 $ 符号开头，后跟字符串长度。

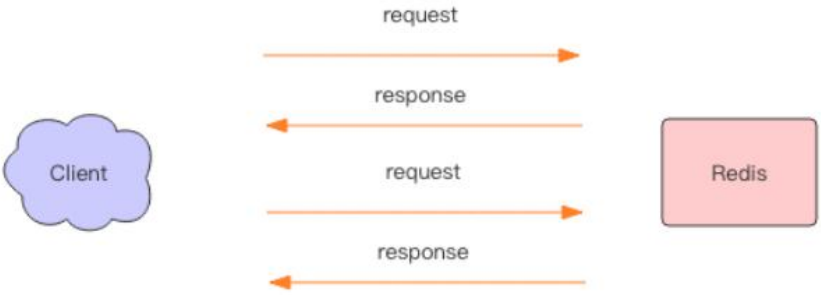
3、整数值 以 : 符号开头，后跟整数的字符串形式。

4、错误消息 以 - 符号开头。

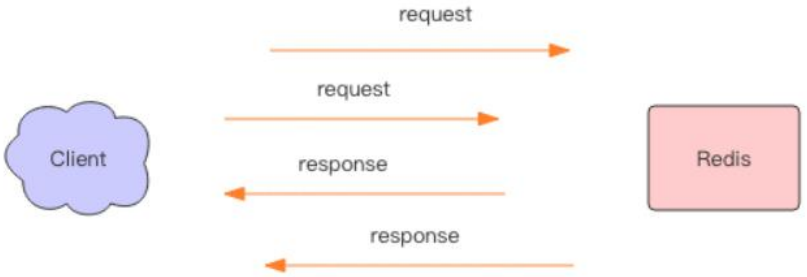
5、数组 以 * 号开头，后跟数组的长度。

- **管道（客户端）**

正常网络通信

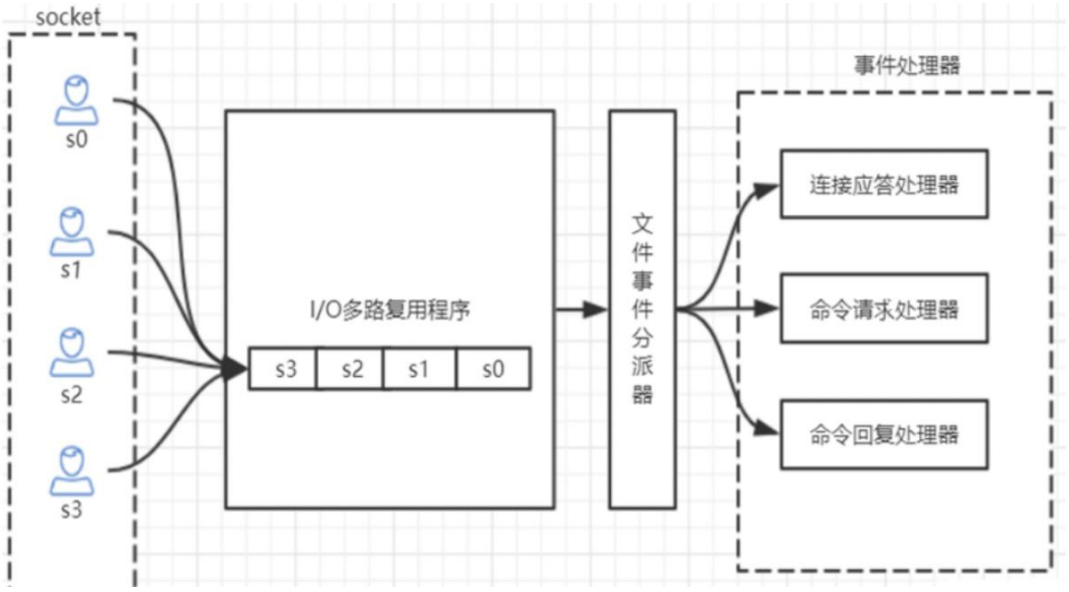


管道网络通信



管道通过改变指令读写顺序提高性能，正常一条指令发送后要一直等待结果返回后在发送下一条指令，管道先发送多条指令（write 指令基本不占时间，发送至本地缓冲区后就返回了），第一条指令等待一个网络开销的时间（read 指令占时长，需要等待结果返回），后续的指令差不多也返回结果了

- **io 模型**



redis 网络 IO 模型通过 reactor 模式实现，首先基于 epoll 的 io 多路复用程序监控 socket 文件描述符，通过监听服务端连接、socket 的读取、写入事件，将事件丢到事件队列，由事件分发器将事件分发分发给对应的事件处理器。

Redis6.0 引入了多线程，主要处理缓冲区指令的读取和解析，执行指令后的结果放入缓冲区，指令的执行还是由主线程（单线程）执行

Redis 变慢

- **使用 keys 指令**
  keys 会遍历所有的 key，导致其它指令阻塞，使用 scan
- **大 key**
  通过 redis-cli -h 127.0.0.1 -p6379 -a "password" -- bigkeys 命令查找大 key，只能找到每种类型中最大的一个

  使用 RdbTools 工具查找大 key 如：rdb dump.rdb -c memory --bytes 10240 -f redis.csv

  拆分大 key

- **大批 key 操作**
  大量 key 过期，删除 key 有主线程执行（设置 key 不一起过期）

  可能有定时任务不断读写

- **慢日志**
  首先通过 redis-cli --intrinsic-latency 100（s） 查看 redis 正常最大响应时间，配置命令执行多久写入慢日志：redis-cli CONFIG SET slowlog-log-slower-than 6000，通过 SLOWLOG get 5（数量） 获取慢指令
- **cpu 飙升**
  查看客户端连接数：redis-cli info clients
  内存使用情况：info memory
  连接数过多拒绝连接数量：redis-cli info stats |grep reject
  查看指令执行情况：redis-cli info commandstats 返回值 calls: 次数 usec: 总时间 usec_per_call:平均时间
  每秒发送指令：redis-cli info stats | grep ops
  复制积压缓冲区大小：redis-cli info replication |grep backlog
  输出执行的指令到文件：ssredis-cli -h 127.0.0.1 -p 6379 -a password monitor > a.txt
- **网络不好**