

零声教育 Mark 老师 QQ: 2548898954

## redis源码学习

### wsl2安装

<https://docs.microsoft.com/zh-cn/windows/wsl/install-win10>

### vscode

<https://docs.microsoft.com/zh-cn/windows/wsl/tutorials/wsl-vscode>

### c/c++基本配置

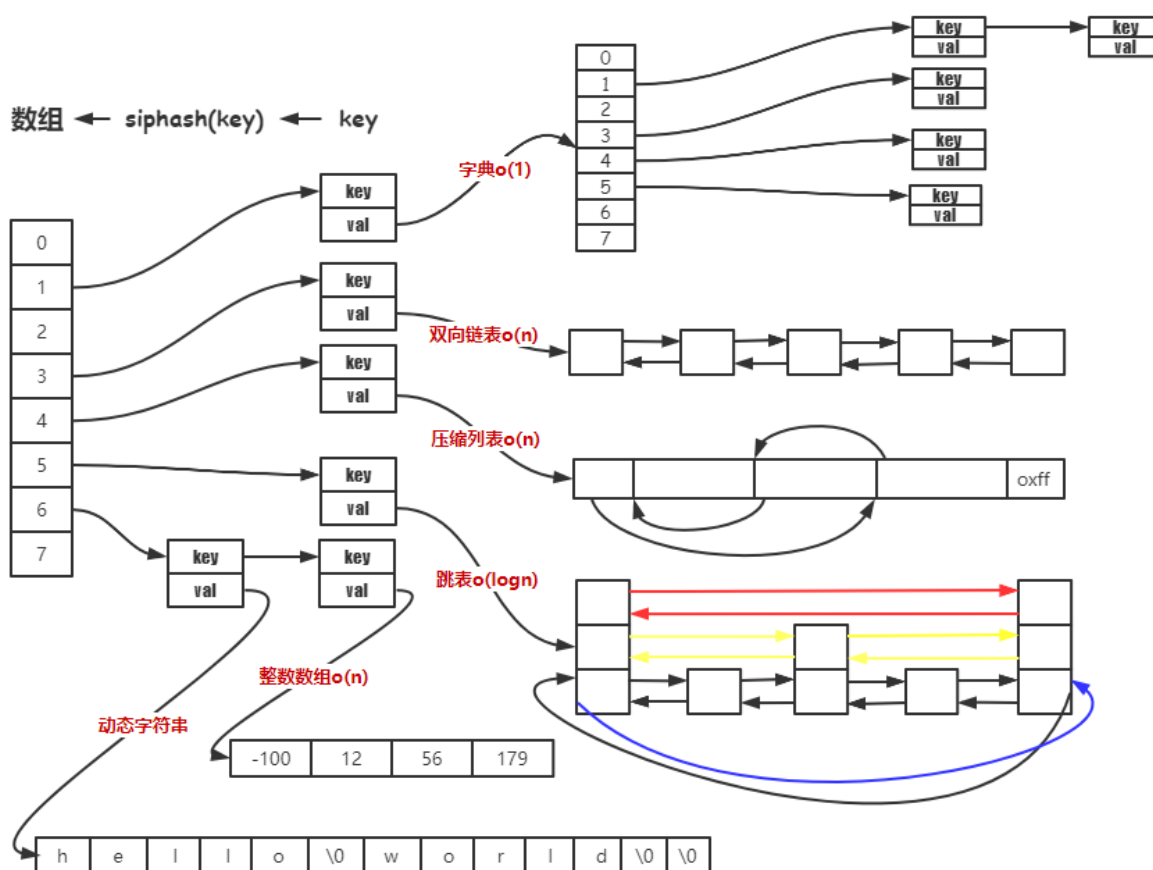
<https://code.visualstudio.com/docs/cpp/config-wsl>

### 建议学习方法

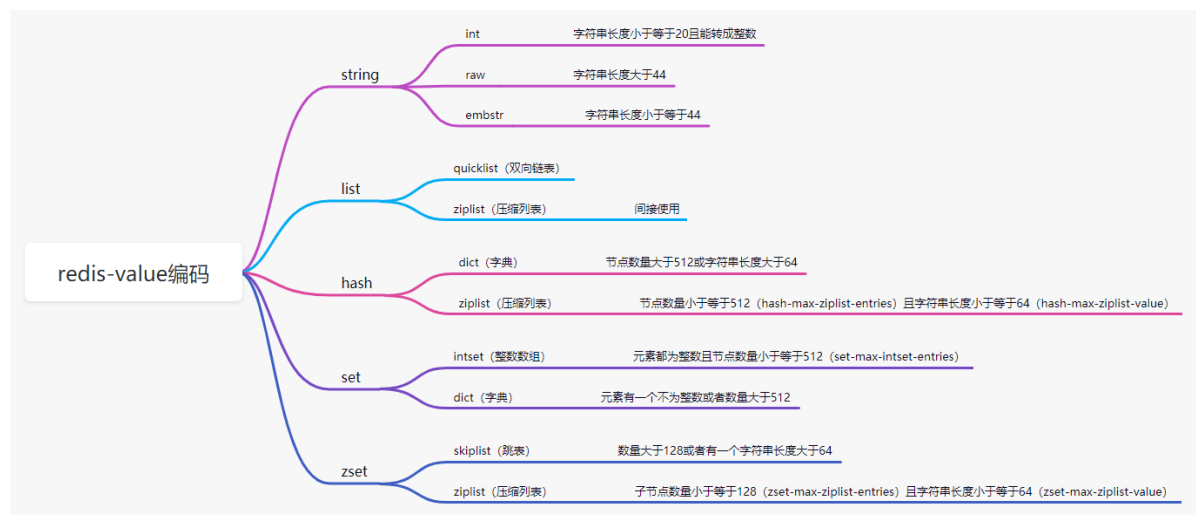
1. 首先定一个小的主题，预期要得到的效果；
2. 准备测试数据以及调试环境；
3. 查看流程，把每一个细支流程拷贝出来；并在旁边写上注释；
4. 得出结论；

## redis 存储结构

### 存储结构



## 存储转换



## 字典实现

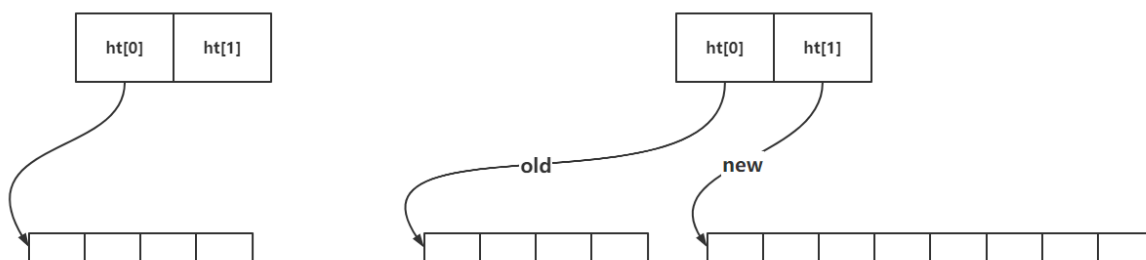
redis DB KV组织是通过字典来实现的；hash结构当节点超过 512 个或者单个字符串长度大于 64 时，hash结构采用字典实现；

## 数据结构

```
1  typedef struct dictEntry {
2      void *key;
3      union {
4          void *val;
5          uint64_t u64;
6          int64_t s64;
7          double d;
8      } v;
9      struct dictEntry *next;
10 } dictEntry;
11
12 typedef struct dictht {
13     dictEntry **table;
14     unsigned long size; // 数组长度
15     unsigned long sizemask; // size-1
16     unsigned long used; // 当前数组当中包含的元素
17 } dictht;
18
19 typedef struct dict {
20     dictType *type;
21     void *privdata;
22     dictht ht[2];
23     long rehashidx; /* rehashing not in progress if rehashidx == -1 */
24     int16_t pauserehash; /* If >0 rehashing is paused (<0 indicates coding
25                          error) 用于安全遍历*/
26 } dict;
```

1. 字符串经过hash函数运算得到64位整数；
2. 相同字符串多次通过hash函数得到相同的64位整数；
3. 整数对2的n次幂取余可以转化为位运算；
4. 抽屉原理 n+1个苹果放在n个抽屉中，苹果最多的那个抽屉至少有2个苹果；64位整数远大于数组的长度，比如数组长度为4，那么1、5、9、1+4n都是映射到1号位数组；所以大概率会

发生冲突;



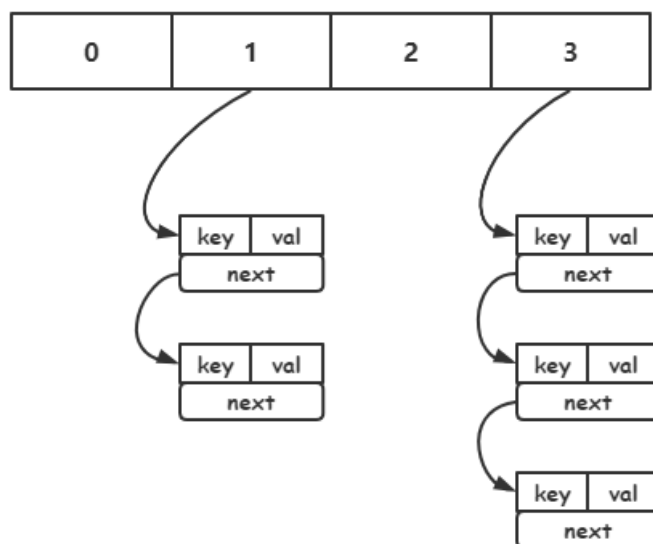
## 冲突

### 负载因子

负载因子 = `used / size`; `used` 是数组存储元素的个数, `size` 是数组的长度;

负载因子越小, 冲突越小; 负载因子越大, 冲突越大;

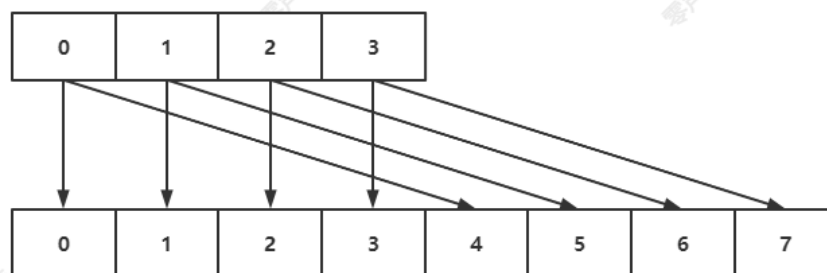
redis的负载因子是 1;



## 扩容

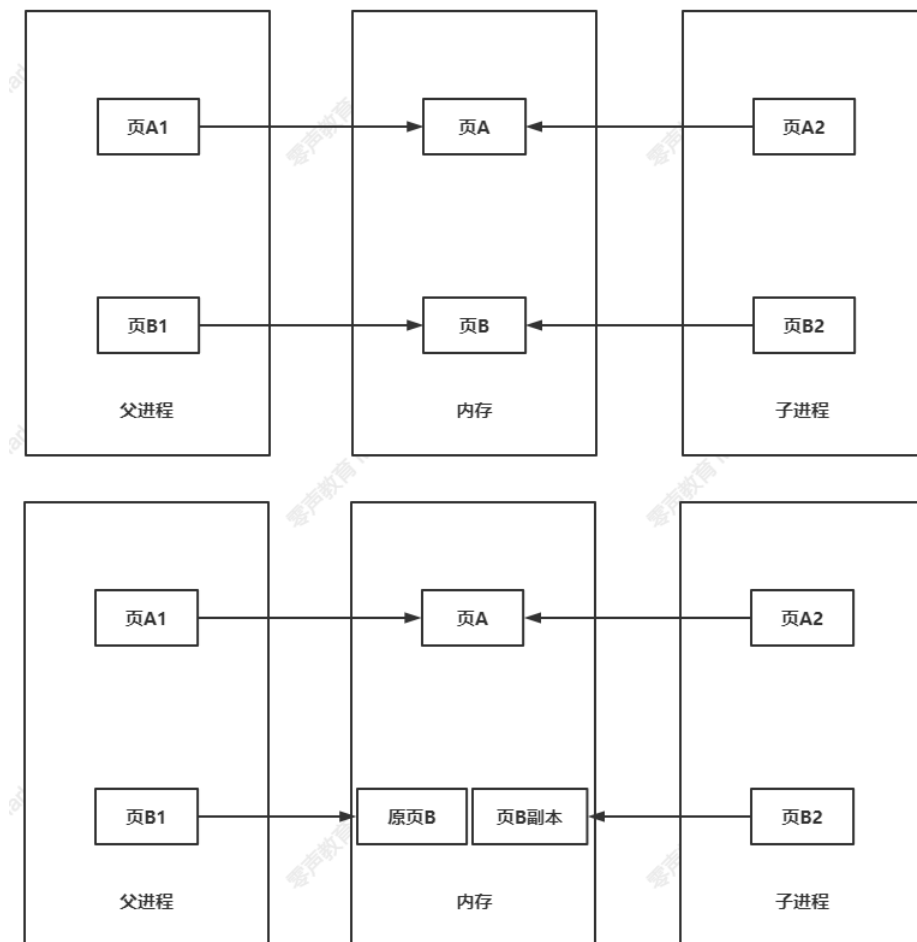
如果负载因子  $> 1$ , 则会发生扩容; 扩容的规则是翻倍;

如果正在 `fork` (在 `rdb`、`aof` 复写以及 `rdb-aof` 混用情况下) 时, 会阻止扩容; 但是此时若负载因子  $> 5$ , 索引效率大大降低, 则马上扩容; 这里涉及到写时复制原理;



## 写时复制

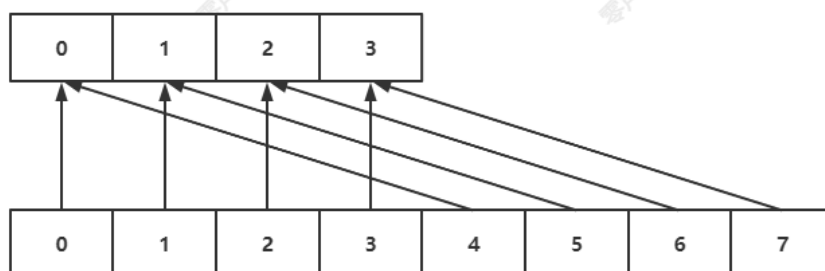
写时复制核心思想：只有在不得不复制数据内容时才去复制数据内容；



## 扩容

如果负载因子  $< 0.1$ ，则会发生扩容；扩容的规则是恰好包含 `used` 的  $2^n$ ；

恰好的理解：假如此时数组存储元素个数为 9，恰好包含该元素的就是  $2^4$ ，也就是 16；



## 渐进式rehash

当 `hashtable` 中的元素过多的时候，不能一次性 `rehash` 到 `ht[1]`；这样会长期占用 `redis`，其他命令得不到响应；所以需要使用渐进式 `rehash`；

**rehash步骤：**

将 `ht[0]` 中的元素重新经过hash函数生成64位整数，再对 `ht[1]` 长度进行取余，从而映射到 `ht[1]`；

**渐进式规则：**

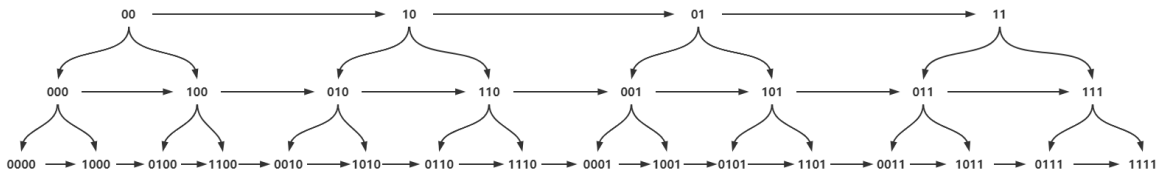
1. 分治的思想，将 `rehash` 分到之后的每步增删改查的操作当中；
2. 在定时器中，最大执行一毫秒 `rehash`；每次步长 100 个数组槽位；

面试：

处于渐进式rehash阶段时，是否会发生扩容缩容？不会！

## scan

```
1 | scan cursor [MATCH pattern] [COUNT count] [TYPE type]
```



采用高位进位加法的遍历顺序，`rehash` 后的槽位在遍历顺序上是相邻的；

遍历目标是：不重复，不遗漏；

## expire机制

```
1 | # 只支持对最外层key过期；
2 | expire key seconds
3 | pexpire key milliseconds
4 | ttl key
5 | pttl key
```

## 惰性删除

分布在每一个命令操作时检查 `key` 是否过期；若过期删除 `key`，再进行命令操作；

## 定时删除

在定时器中检查库中指定个数（25）个 `key`；

```
1 | #define ACTIVE_EXPIRE_CYCLE_KEYS_PER_LOOP 20 /* Keys for each DB loop. */
2 | /*The default effort is 1, and the maximum configurable effort
3 |    * is 10. */
4 | config_keys_per_loop = ACTIVE_EXPIRE_CYCLE_KEYS_PER_LOOP +
5 |                        ACTIVE_EXPIRE_CYCLE_KEYS_PER_LOOP/4*effort,
6 | int activeExpireCycleTryExpire(redisDb *db, dictEntry *de, long long now);
```

## 大KEY

在 `redis` 实例中形成了很大的对象，比如一个很大的 `hash` 或很大的 `zset`，这样的对象在扩容的时候，会一次性申请更大的一块内存，这会导致卡顿；如果这个大 `key` 被删除，内存会一次性回收，卡顿现象会再次产生；

如果观察到 `redis` 的内存大起大落，极有可能因为大 `key` 导致的；

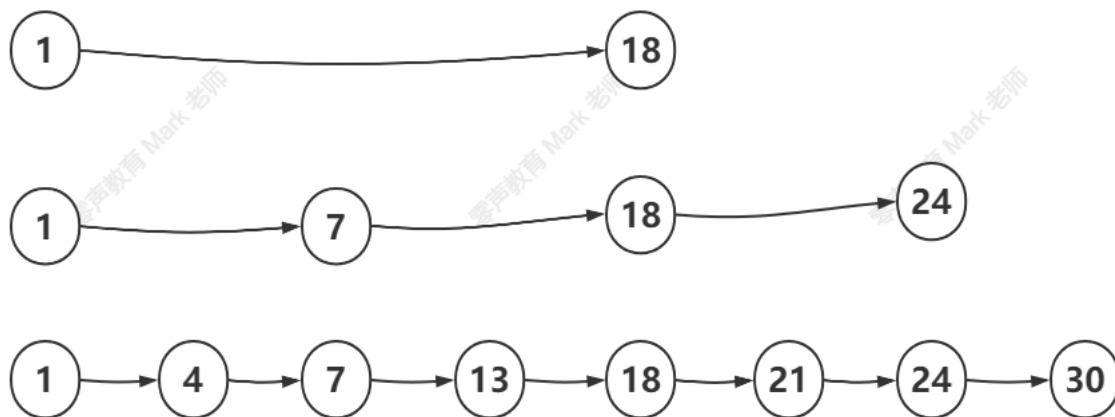
```
1 | # 每隔0.1秒 执行100条scan命令
2 | redis-cli -h 127.0.0.1 --bigkeys -i 0.1
```

## 跳表实现

跳表（多层级有序链表）结构用来实现有序集合；鉴于redis需要实现 `zrange` 以及 `zrevrange` 功能；需要节点间最好能直接相连并且增删改操作后结构依然有序；B+树时间复杂度为  $h * O(\log_2 n)$ ；鉴于B+复杂的节点分裂操作；考虑其他数据结构； $o(1)$   $o(\log_2 n)$

有序数组通过二分查找能获得  $o(\log_2 n)$  时间复杂度；平衡二叉树也能获得  $o(\log_2 n)$  时间复杂度；

### 理想跳表



每隔一个节点生成一个层级节点；模拟二叉树结构，以此达到搜索时间复杂度为  $O(\log_2 n)$ ；

但是如果对理想跳表结构进行删除增加操作，很有可能改变跳表结构；如果重构理想结构，将是巨大的运算；考虑用概率的方法来进行优化；从每一个节点出发，每增加一个节点都有  $1/2$  的概率增加一个层级， $1/4$  的概率增加两个层级， $1/8$  的概率增加三个层级，以此类推；经过证明，当数据量足够大（256）时，通过概率构造的跳表趋向于理想跳表，并且此时如果删除节点，无需重构跳表结构，此时依然趋向于理想跳表；此时时间复杂度为  $(1 - \frac{1}{n^c}) * O(\log_2 n)$ ；

### redis跳表

从节约内存出发，redis 考虑牺牲一点时间复杂度让跳表结构更加变扁平，就像二叉堆改成四叉堆结构；并且redis 还限制了跳表的最高层级为 32；

节点数量大于 128 或者有一个字符串长度大于 64，则使用跳表（skiplist）；

### 数据结构

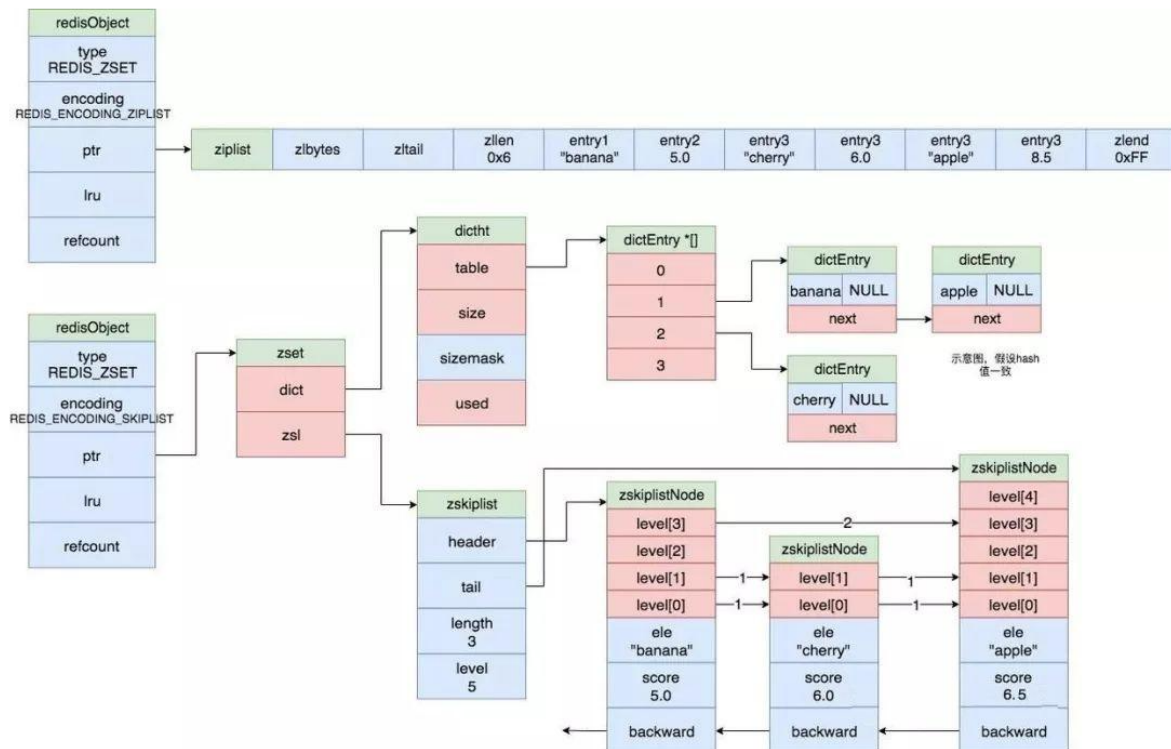
```
1 #define ZSKIPLIST_MAXLEVEL 32 /* Should be enough for 2^64 elements */
2 #define ZSKIPLIST_P 0.25 /* Skiplist P = 1/4 */
3 /* ZSETs use a specialized version of Skiplists */
4 typedef struct zskiplistNode {
5     sds ele;
6     double score; // WRN: score 只能是浮点数
7     struct zskiplistNode *backward;
8     struct zskiplistLevel {
9         struct zskiplistNode *forward;
10        unsigned long span; // 用于 zrank
11    } level[];
12 } zskiplistNode;
13
14 typedef struct zskiplist {
15     struct zskiplistNode *header, *tail;
```

```

16     unsigned long length; // zcard
17     int level;           // 最高层
18 } zskiplist;
19
20 typedef struct zset {
21     dict *dict;           // 帮助快速索引到节点
22     zskiplist *zsl;
23 } zset;

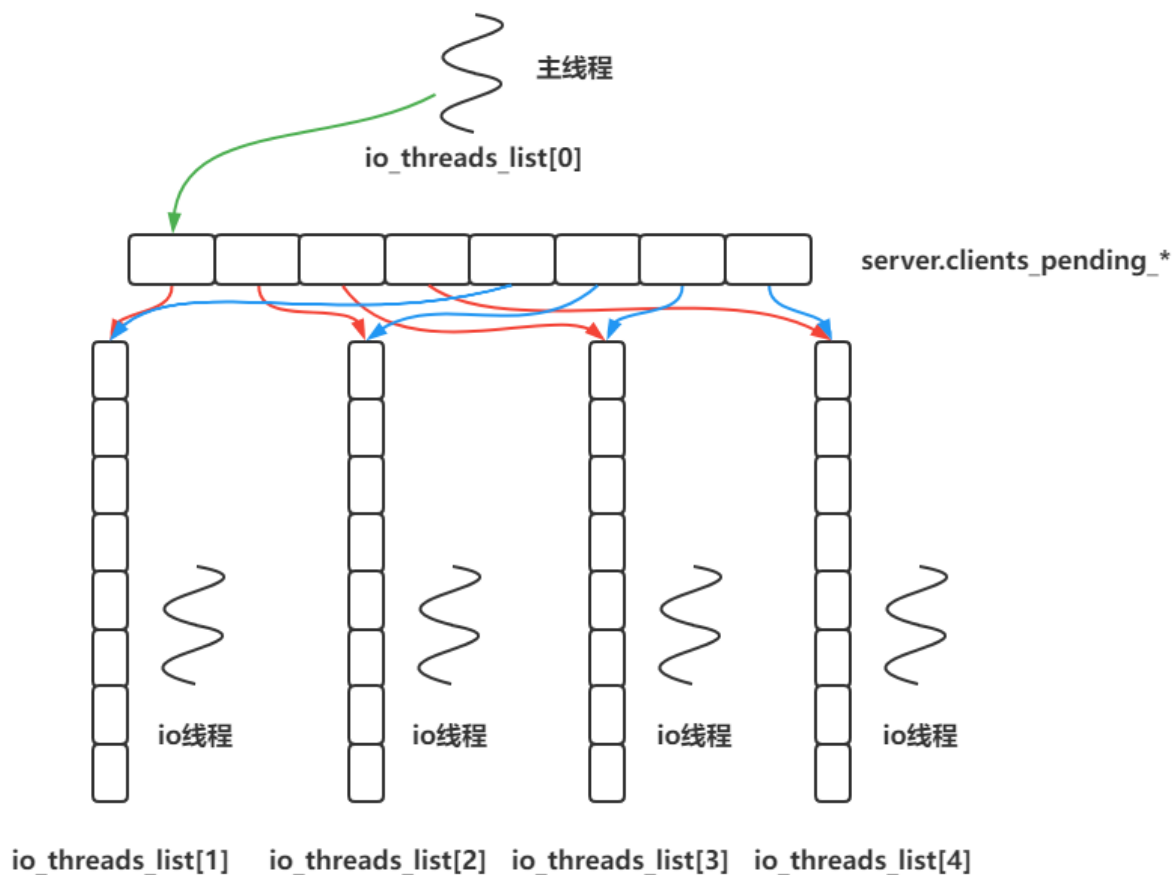
```

## 结构图



## 补充

### redis io多线程



## 线程池总结

