

Redis缓存数据库



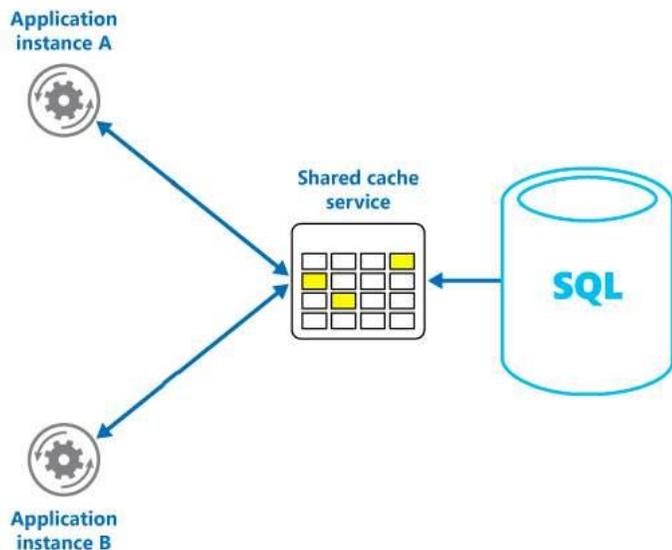
01

缓存通识

缓存通识

Redis在互联网技术存储方面使用如此广泛，几乎所有的后端技术都需要在Redis的使用和原理方面进行熟知。Redis绝大多数情况下都是被当做缓存来使用的。

缓存是高并发场景下提高热点数据访问性能的一个有效手段，在开发项目时会经常使用到。

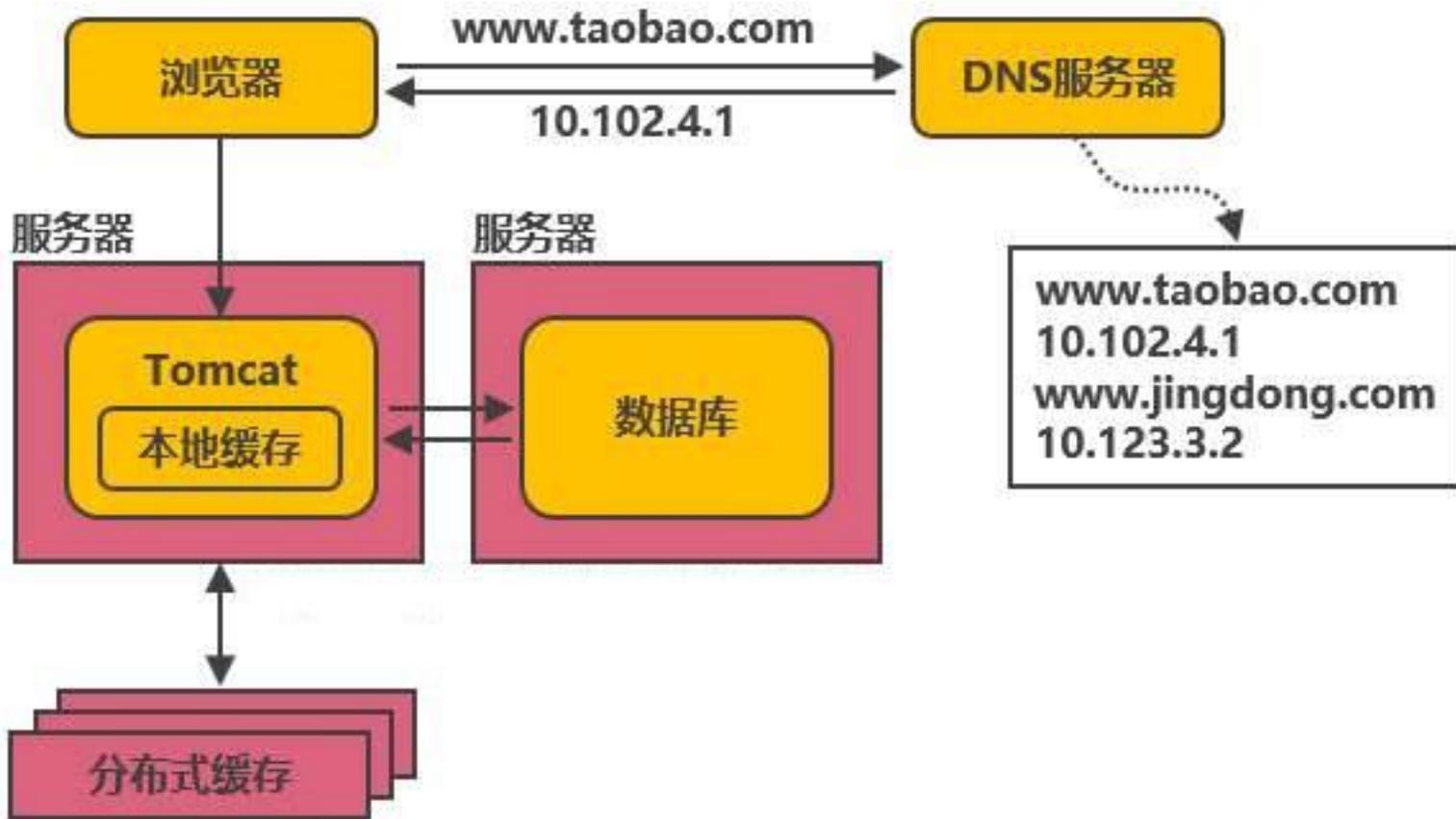


缓存类型

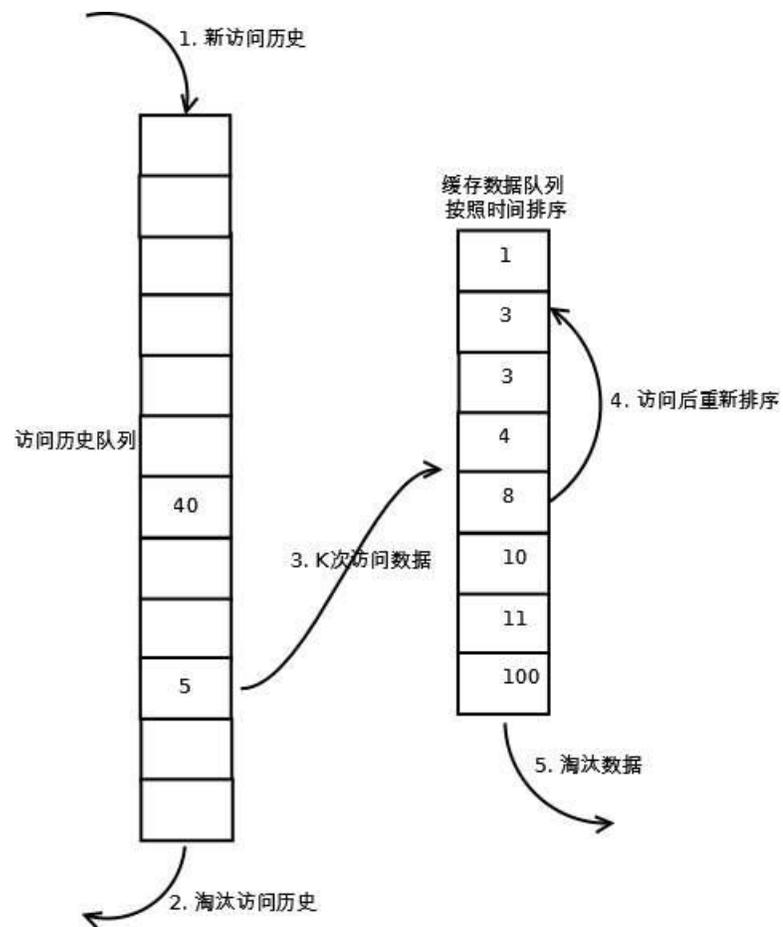
本地缓存 LruMap
Ehcache

分布式缓存

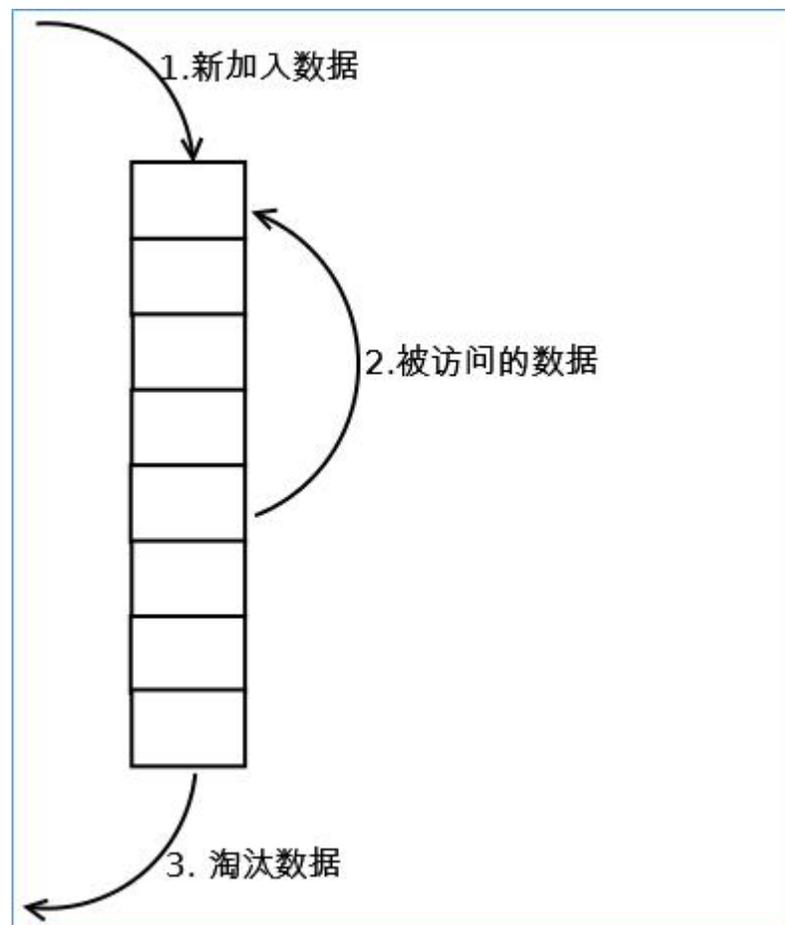
多级缓存



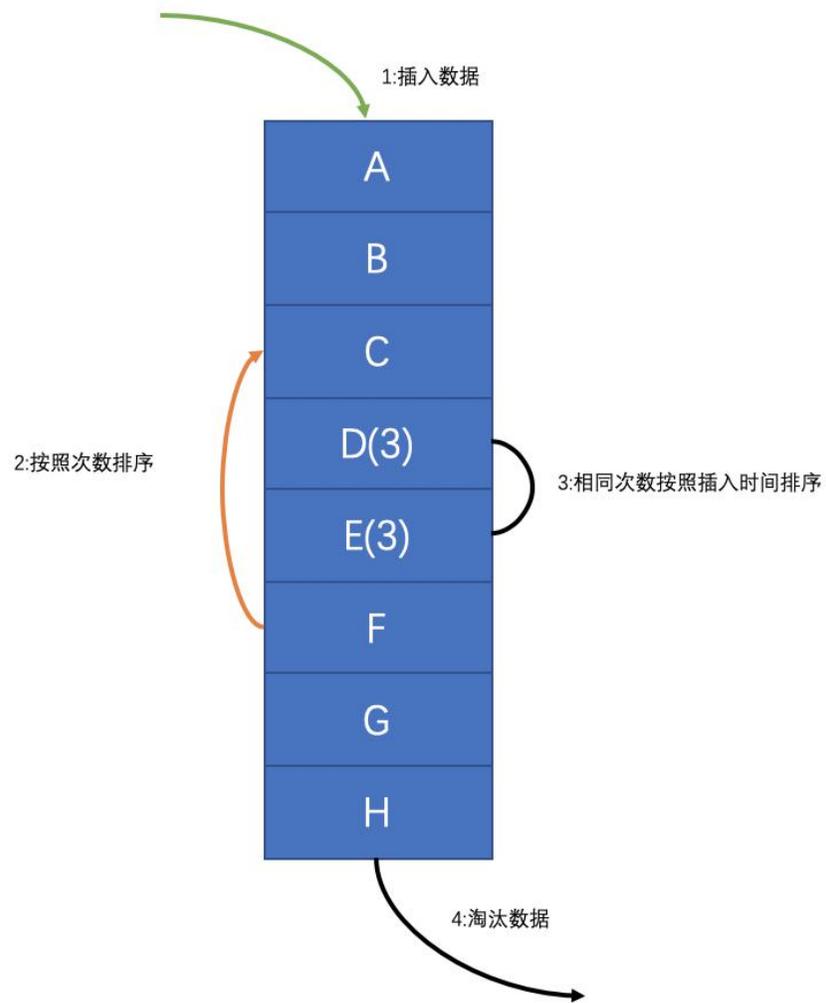
淘汰策略-FIFO



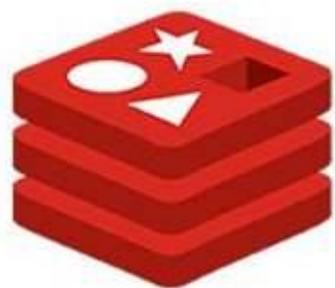
淘汰策略-LRU



淘汰策略-LFU



Memcache 和 Redis

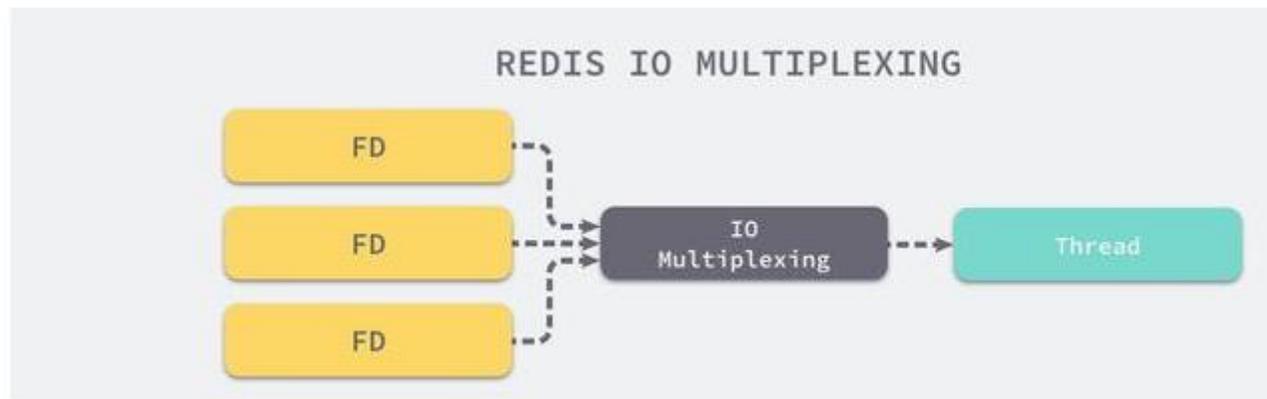


redis



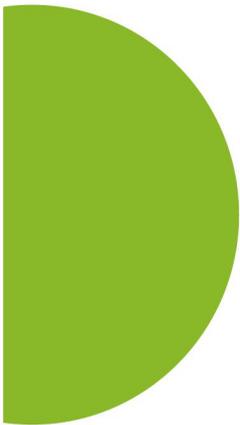
Redis6.0 引入了多线程

1 为什么 Redis 一开始选择单线程模型（单线程的好处）？



2 为什么 Redis 在 6.0 之后加入了多线程？





02

Redis 架构原理

内存模型-查看内存统计

```
127.0.0.1:6379> info memory
# Memory
#Redis分配的内存总量,包括虚拟内存(字节)
used_memory:853464
#占操作系统的内存, 不包括虚拟内存(字节)
used_memory_rss:12247040
#内存碎片比例 如果小于0说明使用了虚拟内存
mem_fragmentation_ratio:15.07
#Redis使用的内存分配器
mem_allocator:jemalloc-5.1.0
```

Redis内存划分

数据

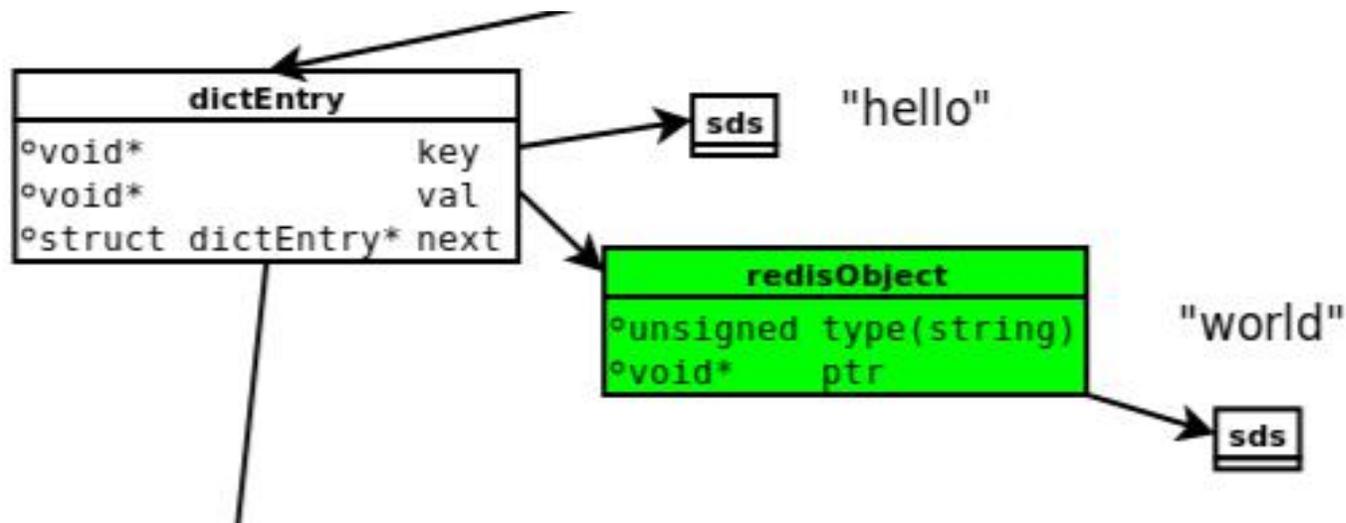
进程

缓冲内存

内存碎片

Redis数据存储的细节

Redis 是一个K-V NOSQL
五种类型 都是针对K-V中的V的



Redis内存分配器-jemalloc

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, ..., 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
	512	[2560, 3072, 3584]
	Large	4 KiB
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]



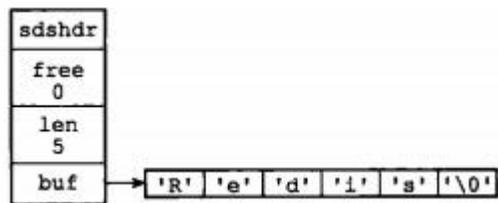
redisObject

```
1 typedef struct redisObject {  
2     unsigned type:4; //类型 五种对象类型  
3     unsigned encoding:4; //编码  
4     void *ptr; //指向底层实现数据结构的指针  
5     //...  
6     int refcount; //引用计数  
7     //...  
8     unsigned lru:24; //记录最后一次被命令程序访问的时间  
9     //...  
10 }robj;
```

SDS

Redis没有直接使用C字符串(即以空字符' \0' 结尾的字符数组)作为默认的字符串表示,而是使用了SDS。SDS是简单动态字符串(Simple Dynamic String)的缩写。

```
1 struct sdshdr{
2     //记录buf数组中已使用字节的数量
3     //等于 SDS 保存字符串的长度
4     int len;
5     //记录 buf 数组中未使用字节的数量
6     int free;
7     //字节数组,用于保存字符串
8     char buf[];
9 }
```



3.2之前SDS所占用空间 = free所占长度+len所占
长度+buf数组长度 = 4+4+free+len+1 = free+len+9

```

1 typedef char *sds;
2
3 struct __attribute__((packed)) sdshdr5 { // 对应的字符串长度小于 1<<5
4     unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
5     char buf[];
6 };
7 struct __attribute__((packed)) sdshdr8 { // 对应的字符串长度小于 1<<8
8     uint8_t len; /* used */ //目前字符创的长度
9     uint8_t alloc; //已经分配的总长度
10    unsigned char flags; //flag用3bit来标明类型, 类型后续解释, 其
    余5bit目前没有使用
11    char buf[]; //柔性数组, 以'\0'结尾
12 };
13 struct __attribute__((packed)) sdshdr16 { // 对应的字符串长度小于 1<<16
14     uint16_t len; /* used */
15     uint16_t alloc; /* excluding the header and null terminator */
16     unsigned char flags; /* 3 lsb of type, 5 unused bits */
17     char buf[];
18 };
19 struct __attribute__((packed)) sdshdr32 { // 对应的字符串长度小于 1<<32
20     uint32_t len; /* used */
21     uint32_t alloc; /* excluding the header and null terminator */
22     unsigned char flags; /* 3 lsb of type, 5 unused bits */
23     char buf[];
24 };
25 struct __attribute__((packed)) sdshdr64 { // 对应的字符串长度小于 1<<64
26     uint64_t len; /* used */ //
27     uint64_t alloc; /* excluding the header and null terminator */
28     unsigned char flags; /* 3 lsb of type, 5 unused bits */
29     char buf[];
30 };
31

```

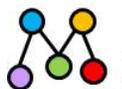
新版带来的好处就是针对长度不同的字符串做了优化，选取不同的数据类型uint8_t或者uint16_t或者uint32_t等来表示长度、一共申请字节的大小等。结构体中的__attribute__((packed))设置是告诉编译器取消字节对齐，则结构体的大小就是按照结构体成员实际大小相加得到的。



SDS:O(1) C:O(n)

Redis的对象类型与内部编码

类型	编码	OBJECT ENCODING命令输出	对象
REDIS_STRING	REDIS_ENCODING_INT	"int"	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	"embstr"	使用embstr编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	"raw"	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	"ziplist"	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	"linkedlist"	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	"ziplist"	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	"hashtable"	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	"intset"	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	"hashtable"	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	"ziplist"	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	"skiplist"	使用跳跃表和字典实现的有序集合对象



字符串长度**不能超过512MB**。

int: 8个字节的长整型 long整数表示

embstr <=44字节的字符串

raw: 大于44个字节的字符串

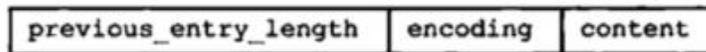
列表

只有同时满足下面两个条件时，才会使用压缩列表：

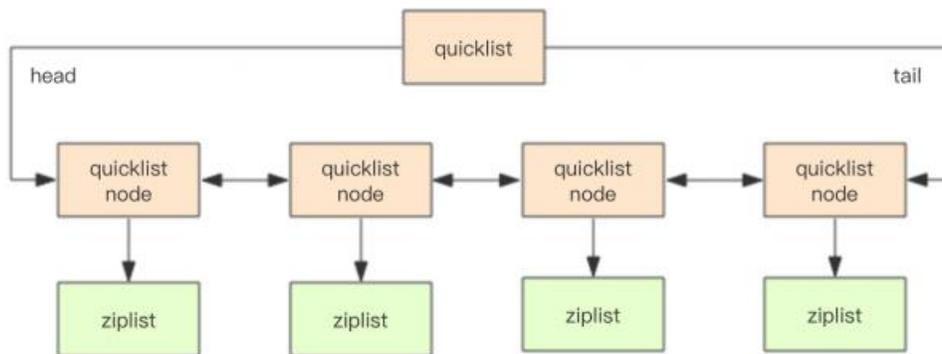
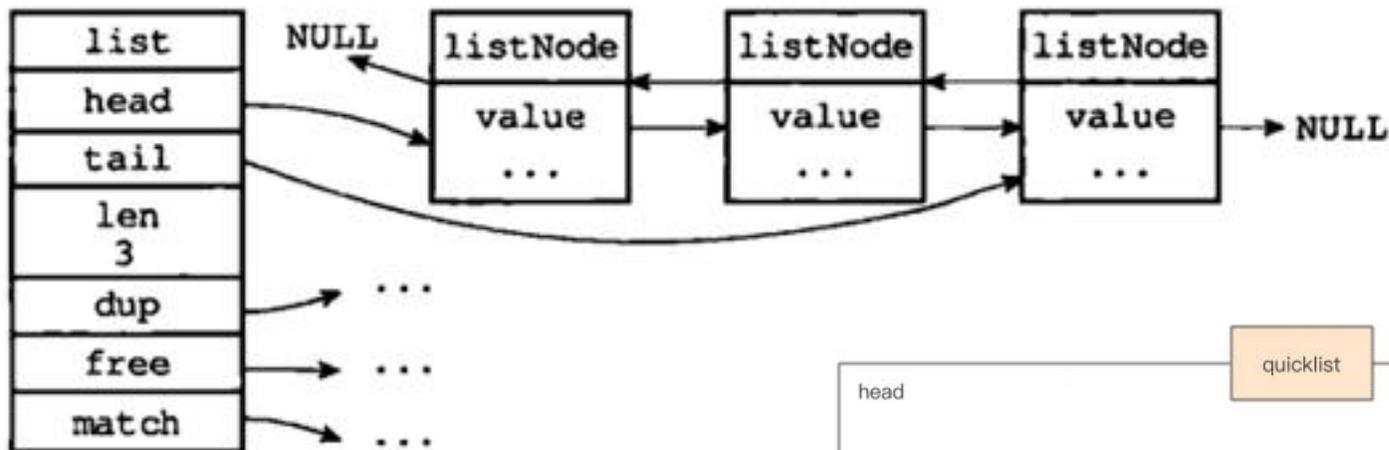
列表中元素数量小于512个；

*列表中所有字符串对象都不足64字节

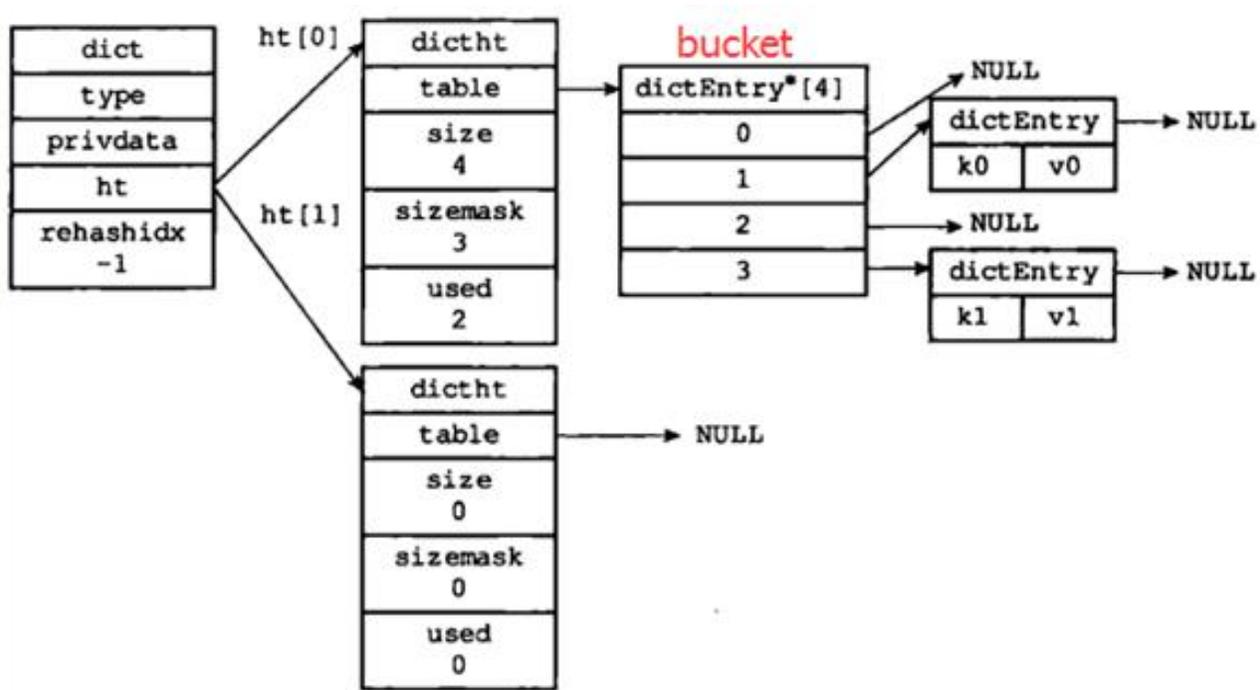
压缩列表的每个节点构成如下：



压缩列表节点各个组成部分



哈希 (压缩列表和哈希表)



哈希 (压缩列表和哈希表)

```
1 typedef struct dictEntry{
2     void *key;
3     union{
4         void *val;
5         uint64_tu64;
6         int64_ts64;
7     }v;
8     struct dictEntry *next;
9 }dictEntry;
```

```
1 typedef struct dictht{
2     dictEntry **table;
3     unsigned long size;
4     unsigned long sizemask;
5     unsigned long used;
6 }dictht;
```

```
1 typedef struct dict{
2     dictType *type;
3     void *privdata;
4     dictht ht[2];
5     int trehashidx; //rehash
6     int iterators;
7 } dict;
```

集合 (整数集合和哈希表)

```
1 typedef struct intset{
2     uint32_t encoding;
3     uint32_t length;
4     int8_t contents[];
5 } intset;
```

只有同时满足下面两个条件时，集合才会使用整数集合：

- * 集合中**元素数量小于512个**
- * 集合中**所有元素都是整数值**

有序集合（压缩列表和跳跃表）

只有同时满足下面两个条件时，才会使用压缩列表：

有序集合中**元素数量小于128个**；

有序集合中**所有成员长度都不足64字节**。

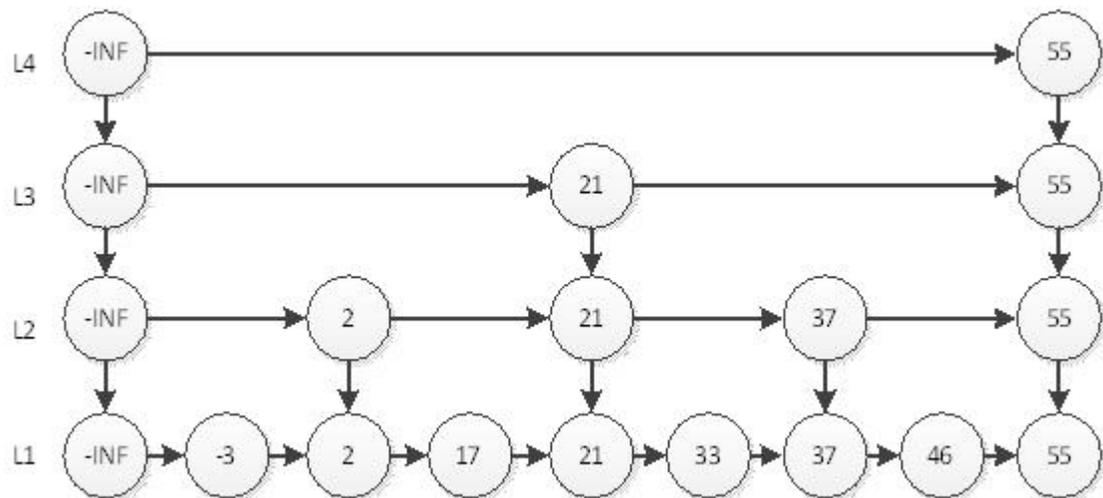
如果有一个条件不满足，则使用跳跃表；且编码只可能由压缩列表转化为跳跃表，反方向则不可能。

跳跃表

普通单向链表图示

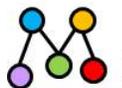


跳跃表图示



跳跃表

```
1 typedef struct zskiplistNode {
2     //层
3     struct zskiplistLevel{
4         //前进指针 后边的节点
5         struct zskiplistNode *forward;
6         //跨度
7         unsigned int span;
8     }level[];
9
10    //后退指针
11    struct zskiplistNode *backward;
12    //分值
13    double score;
14    //成员对象
15    robj *obj;
16
17 } zskiplistNode
18
19 --链表
20 typedef struct zskiplist{
21     //表头节点和表尾节点
22     struct zskiplistNode *header, *tail;
23     //表中节点的数量
24     unsigned long length;
25     //表中层数最大的节点的层数
26     int level;
27
28 }zskiplist;
```



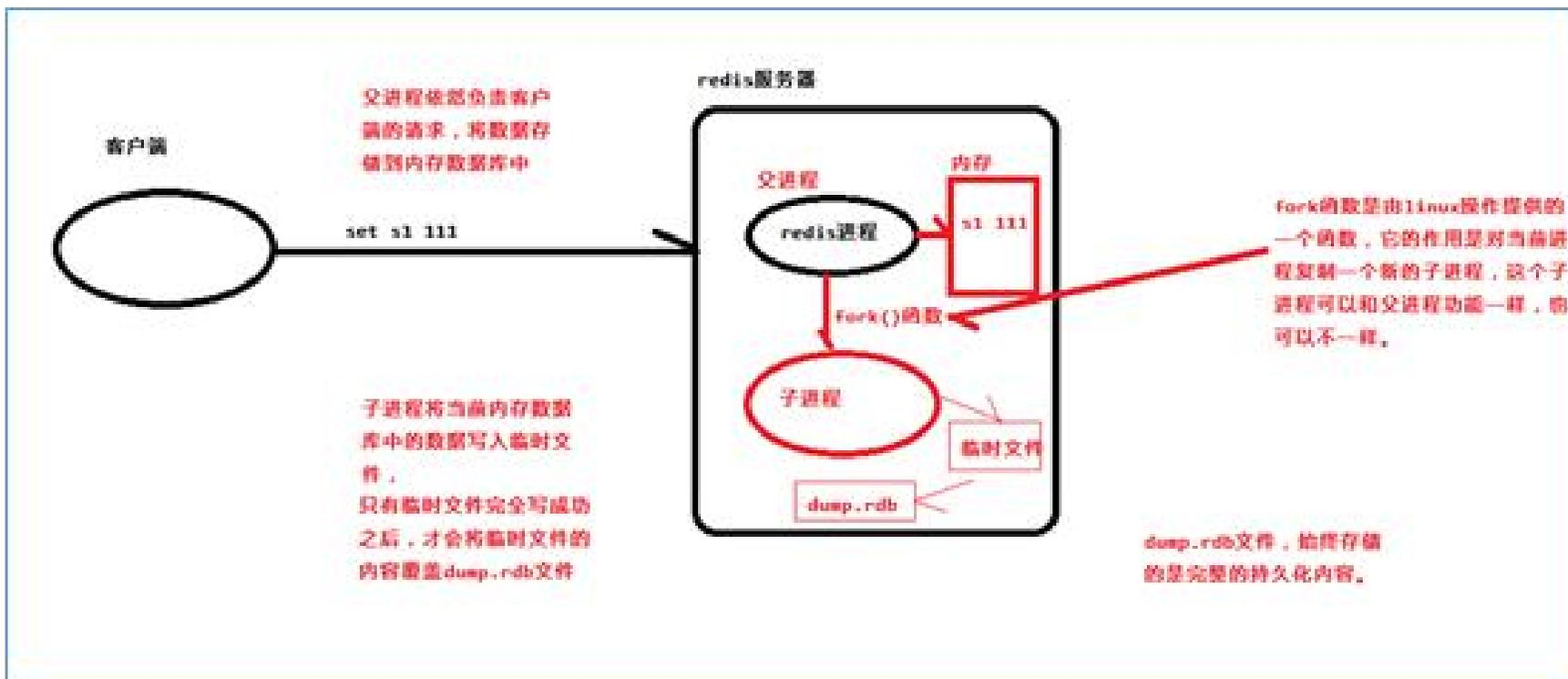
Redis持久化

RDB方式 (默认)

AOF方式

混合持久化模式 (5.0后默认开启)

Redis持久化-RDB



Redis持久化-AOF

redis.conf:

```
1 # 可以通过修改redis.conf配置文件中的appendonly参数开启
2 appendonly yes
3
4 # AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的。
5 dir ./
6
7 # 默认的文件名是appendonly.aof，可以通过appendfilename参数修改
8 appendfilename appendonly.aof
```

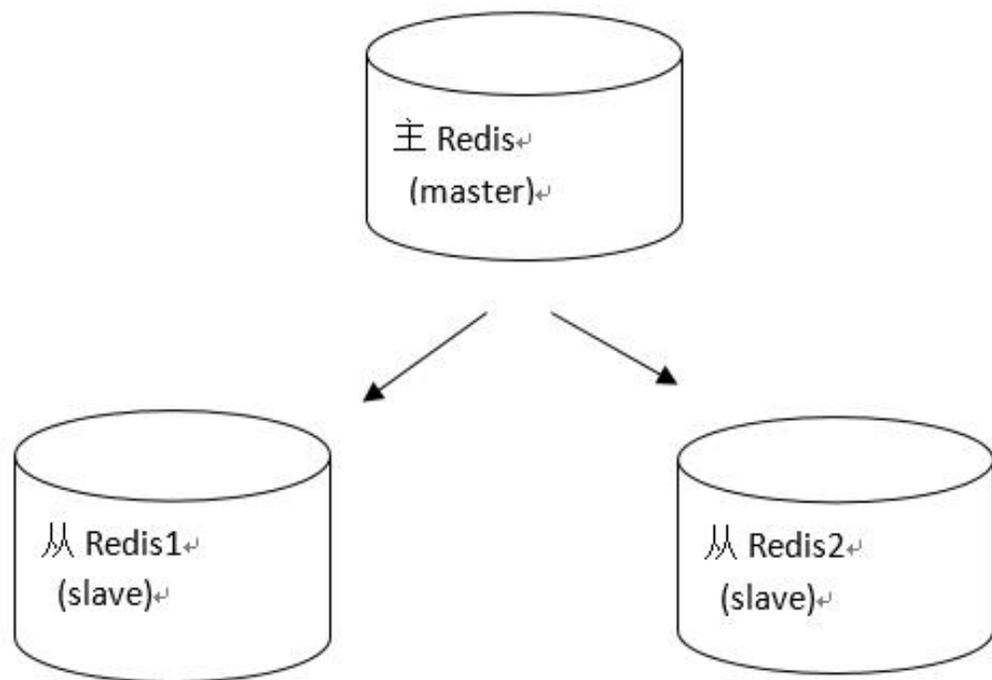


Redis持久化-混合持久化

查询是否开启混合持久化可以使用

```
config get aof-use-rdb-preamble
```

Redis主从复制



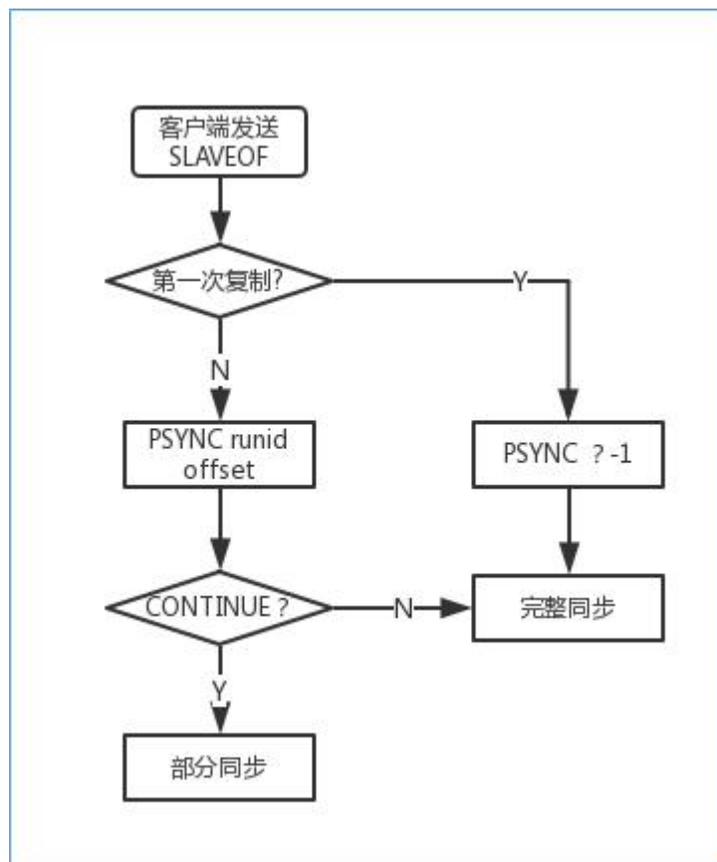
Redis主从复制-从Redis复制

修改从服务器上的redis.conf文件:

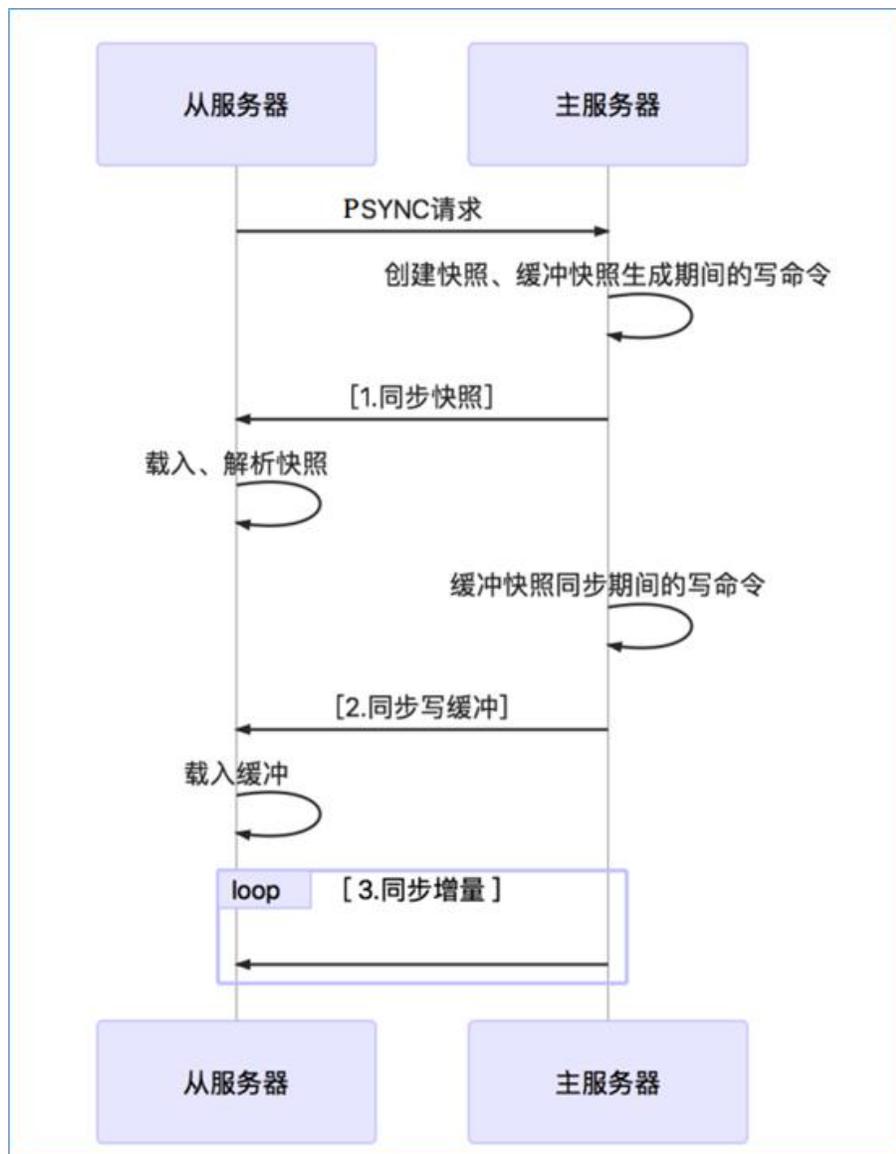
```
1 # replicaof <masterip> <masterport>
2 # 表示当前【从服务器】对应的【主服务器】的IP是192.168.10.135，端口是6379。
3 slaveof 192.168.10.135 6379
4 replicaof 192.168.10.135 6379
```

Redis主从复制原理

- Redis`的主从同步，分为**全量同步**和**增量同步**。
- 只有从机第一次连接上主机是**全量同步**。
- 断线重连有可能触发**全量同步**也有可能是**增量同步**（master判断runid是否一致）。
- 除此之外的情况都是**增量同步**。



Redis主从复制-全量同步



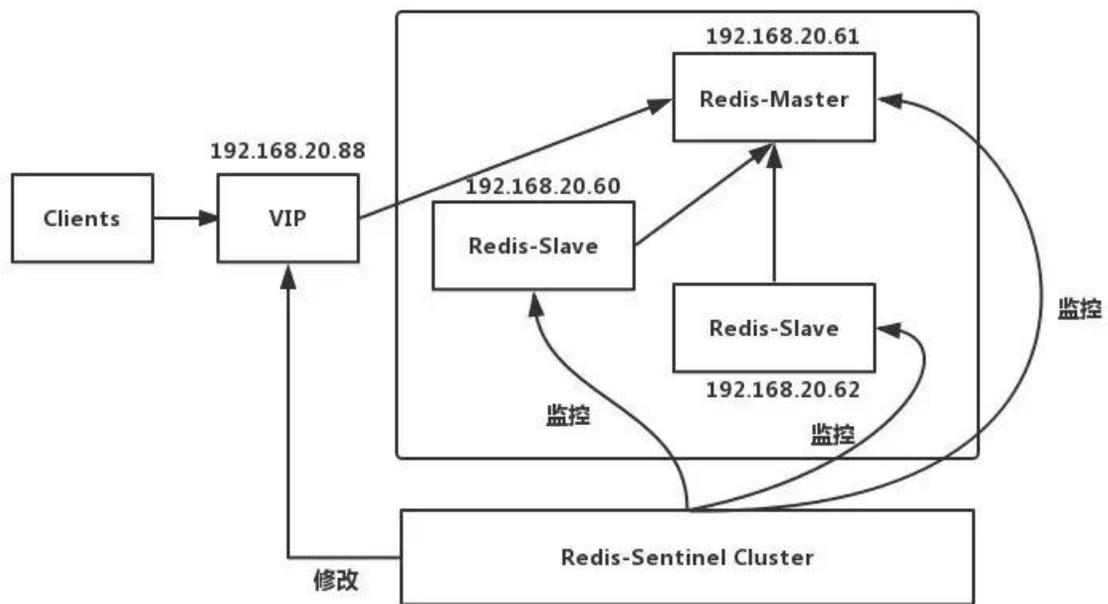
- **监控(Monitoring)**:哨兵(sentinel) 会不断地检查你的 Master 和 Slave 是否运作正常。
- **提醒(Notification)**: 当被监控的某个`Redis`节点出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。
- **自动故障迁移(Automatic failover)**: 当一个Master不能正常工作时, 哨兵(sentinel) 会开始 一次自动故障迁移操作

Redis集群演进

1. Redis主从复制
2. Replication+Sentinel
3. Proxy+Replication+Sentinel
4. Redis Cluster

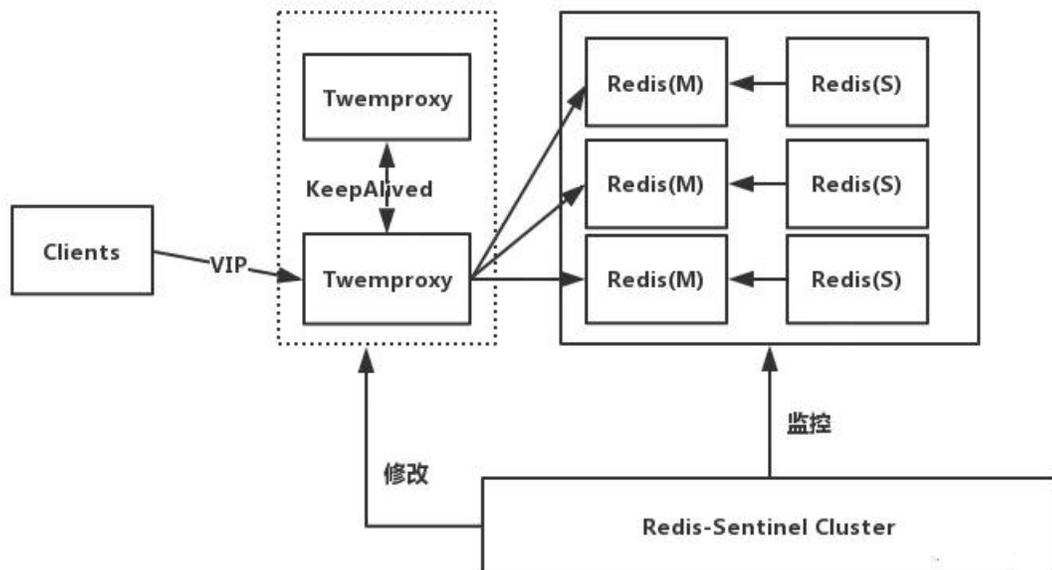
Redis集群演进-Replication+Sentinel

这套架构使用的是社区版本推出的原生高可用解决方案

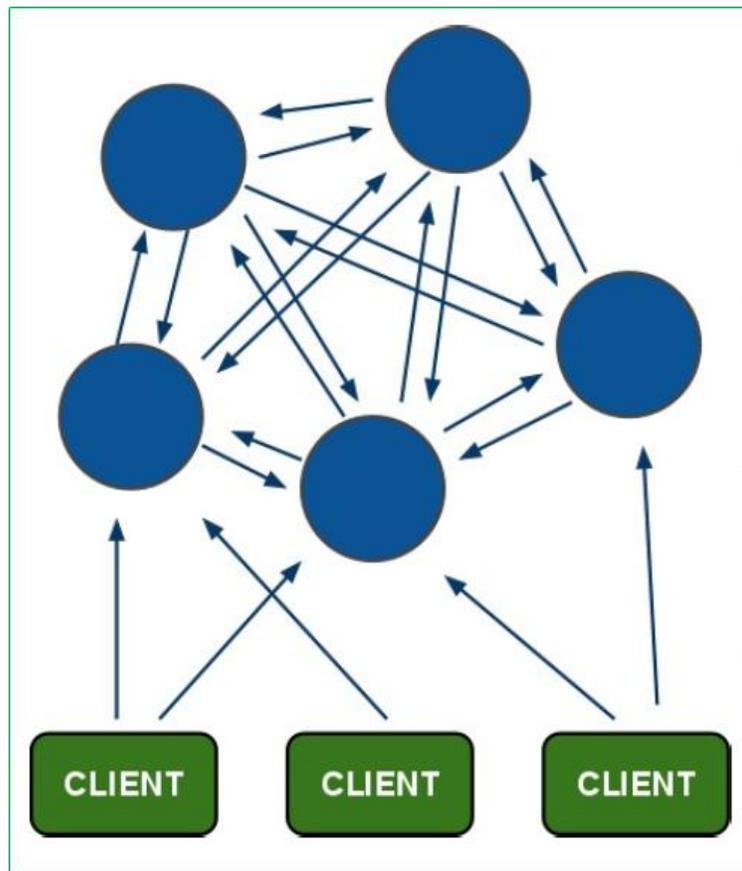


Redis集群演进-Proxy+Replication+Sentinel

这里的Proxy目前有两种选择:Codis（豌豆荚）和Twemproxy（推特）



Redis集群演进-Redis Cluster



JedisCluster类连接redis集群

```
1 @Test
2 public void testJedisCluster() throws Exception {
3     //创建一连接, JedisCluster对象,在系统中是单例存在
4     Set<HostAndPort> nodes = new HashSet<>();
5     nodes.add(new HostAndPort("192.168.10.133", 7001));
6     nodes.add(new HostAndPort("192.168.10.133", 7002));
7     nodes.add(new HostAndPort("192.168.10.133", 7003));
8     nodes.add(new HostAndPort("192.168.10.133", 7004));
9     nodes.add(new HostAndPort("192.168.10.133", 7005));
10    nodes.add(new HostAndPort("192.168.10.133", 7006));
11    JedisCluster cluster = new JedisCluster(nodes);
12    //执行JedisCluster对象中的方法,方法和redis一一对应。
13    cluster.set("cluster-test", "my jedis cluster test");
14    String result = cluster.get("cluster-test");
15    System.out.println(result);
16    //程序结束时需要关闭JedisCluster对象
17    cluster.close();
18 }
19
```

I

