课前准备

- 准备redis安装包

课堂主题

缓存淘汰策略、Redis事务、Redis持久化、Redis主从复制、Redis哨兵机制

课堂目标

- 掌握集合和有序集合的对象类型和内部编码
- 掌握Redis内存优化的一些技巧
- 理解缓存淘汰的LRU策略
- 理解Redis事务的应用
- 掌握redis的RDB和AOF的原理和选型
- 理解Redis主从复制原理和哨兵原理
- 能够配置Redis主从+哨兵

课程回顾

RedisObject

type:五种对外的类型

encoding：对应的类型的内部编码

refcount引用次数

共享对象 refcount > 1

0-9999

OBJ_SHARED_INTEGERS

字符串（SDS）int、embstr、raw

int 整数

embstr <=44字节

raw>44字节

列表（压缩列表和双向链表）

列表中存储的都是字符串元素

512个元素

SDS 64字节

特点：每个节点都有指针去指向下一个节点。

单向链表： 只会指向next节点

双向链表： next\pre节点

跳跃表　　： 回退节点、Level（下一个节点）

哈希（压缩列表和哈希表）

集合（整数集合和哈希表）

集合中存储的都是字符串元素

有序集合（压缩列表和跳跃表）

知识要点

Redis的对象类型与内部编码

3、哈希（压缩列表和哈希表）

（1）概况

哈希（作为一种数据结构），不仅是redis对外提供的5种对象类型的一种（与字符串、列表、集合、有序结合并列），也是Redis作为Key-Value数据库所使用的数据结构。为了说明的方便，后面当使用“内层的哈希”时，代表的是redis对外提供的5种对象类型的一种；使用“外层的哈希”代指Redis作为Key-Value数据库所使用的数据结构。

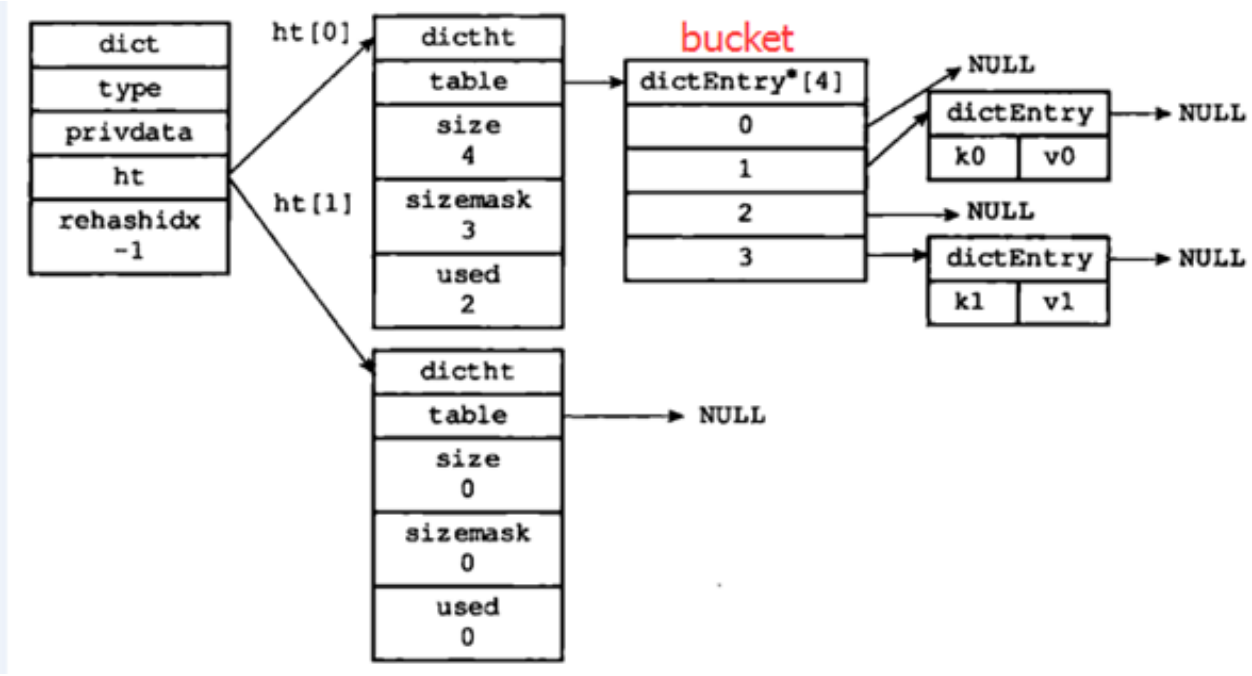
（2）内部编码

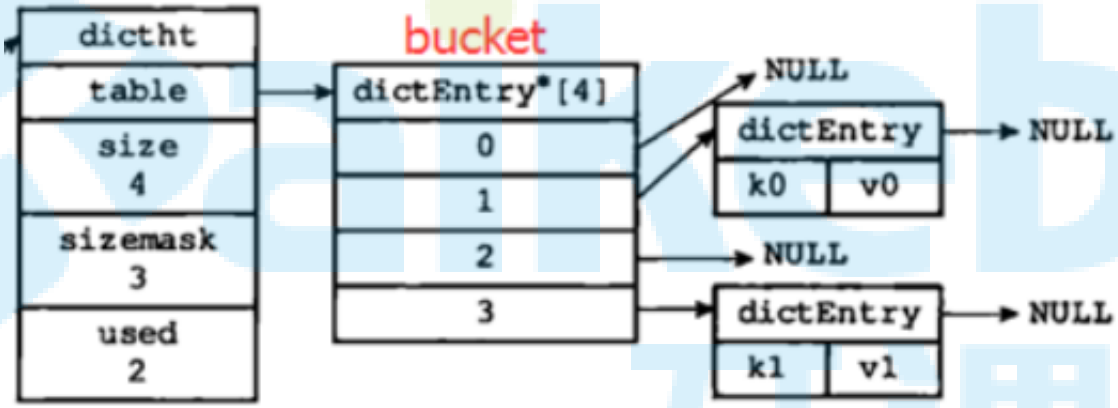
内层的哈希使用的内部编码可以是压缩列表（ziplist）和哈希表（hashtable）两种；Redis的外层的哈希则只使用了hashtable。

> 压缩列表前面已介绍。与哈希表相比，压缩列表用于元素个数少、元素长度小的场景；其优势在于集中存储，节省空间；同时，虽然对于元素的操作复杂度也由O(1)变为了O(n)，但由于哈希中元素数量较少，因此操作的时间并没有明显劣势。

hashtable：一个hashtable由1个dict结构、2个dictht结构、1个dictEntry指针数组（称为bucket）和多个dictEntry结构组成。

正常情况下（即hashtable没有进行rehash时）各部分关系如下图所示：





dictEntry

dictEntry结构用于保存键值对，结构定义如下：

```
typedef struct dictEntry{
    void *key;
    union{
        void *val;
        uint64_tu64;
        int64_ts64;
    }v;
    struct dictEntry *next;
}dictEntry;
```

其中，各个属性的功能如下：

- key：键值对中的键
- val：键值对中的值，使用union(即共用体)实现，存储的内容既可能是一个指向值的指针，也可能是64位整型，或无符号64位整型；

- next：指向下一个dictEntry，用于解决哈希冲突问题

> 在64位系统中，一个dictEntry对象占24字节（key/val/next各占8字节）。

bucket

bucket是一个数组，数组的每个元素都是指向dictEntry结构的指针。redis中bucket数组的大小计算规则如下：大于dictEntry的、最小的2^n；

> 例如，如果有1000个dictEntry，那么bucket大小为1024；如果有1500个dictEntry，则bucket大小为2048。

dictht

dictht结构如下：

```
typedef struct dictht{
    dictEntry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
}dictht;
```

其中，各个属性的功能说明如下：

- table属性是一个指针，指向bucket；
- size属性记录了哈希表的大小，即bucket的大小；
- used记录了已使用的dictEntry的数量；
- sizemask属性的值总是为size-1，这个属性和哈希值一起决定一个键在table中存储的位置。

dict

一般来说，通过使用dictht和dictEntry结构，便可以实现普通哈希表的功能；但是Redis的实现中，在dictht结构的上层，还有一个dict结构。下面说明dict结构的定义及作用。

dict结构如下：

```
typedef struct dict{
    dictType *type;
    void *privdata;
    dictht ht[2];
    int trehashidx;
} dict;
```

其中，type属性和privdata属性是为了适应不同类型的键值对，用于创建多态字典。

ht属性和trehashidx属性则用于rehash，即当哈希表需要扩展或收缩时使用。ht是一个包含两个项的数组，每项都指向一个dictht结构，这也是Redis的哈希会有1个dict、2个dictht结构的原因。通常情况下，所有的数据都是存在放dict的ht[0]中，ht[1]只在rehash的时候使用。dict进行rehash操作的时候，将ht[0]中的所有数据rehash到ht[1]中。然后将ht[1]赋值给ht[0]，并清空ht[1]。

> 因此，Redis中的哈希之所以在dictht和dictEntry结构之外还有一个dict结构，一方面是为了适应不同类型的键值对，另一方面是为了rehash。

（3）编码转换

如前所述，Redis中内层的哈希既可能使用哈希表，也可能使用压缩列表。

只有同时满足下面两个条件时，才会使用压缩列表：

- 哈希中元素数量小于512个；
- 哈希中所有键值对的键和值字符串长度都小于64字节。

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

下图展示了Redis内层的哈希编码转换的特点：

```
127.0.0.1:6379> hset myhash k1 v1
(integer) 1
127.0.0.1:6379> hset myhash k2 v2
(integer) 1
127.0.0.1:6379> hset myhash k3 v3
(integer) 1
127.0.0.1:6379> object encoding myhash
"ziplist"
127.0.0.1:6379> hset myhash k4 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
(integer) 1
127.0.0.1:6379> object encoding myhash
"hashtable"
127.0.0.1:6379> hdel myhash k4
(integer) 1
127.0.0.1:6379> object encoding myhash
"hashtable"
```

4、集合（整数集合和哈希表）

（1）概况

集合（set）与列表类似，都是用来保存多个字符串，但集合与列表有两点不同：集合中的元素是无序的，因此不能通过索引来操作元素；集合中的元素不能有重复。

> 一个集合中最多可以存储2^32-1个元素；除了支持常规的增删改查，Redis还支持多个集合取交集、并集、差集。

（2）内部编码

集合的内部编码可以是整数集合（intset）或哈希表（hashtable）。

整数集合的结构定义如下：

```
typedef struct intset{
    uint32_t encoding;
    uint32_t length;
    int8_t contents[];
} intset;
```

> 其中，encoding代表contents中存储内容的类型，虽然contents（存储集合中的元素）是int8_t类型，但实际上其存储的值是int16_t、int32_t或int64_t，具体的类型便是由encoding决定的；length表示元素个数。

（3）编码转换

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。

如果有一个条件不满足，则使用哈希表；且编码只可能由整数集合转化为哈希表，反方向则不可能。

> 整数集合适用于集合所有元素都是整数且集合元素数量较小的时候，与哈希表相比，整数集合的优势在于集中存储，节省空间；同时，虽然对于元素的操作复杂度也由O(1)变为了O(n)，但由于集合数量较少，因此操作的时间并没有明显劣势。

下图展示了集合编码转换的特点：

```
127.0.0.1:6379> sadd myset 111 222 333
(integer) 3
127.0.0.1:6379> object encoding myset
"intset"
127.0.0.1:6379> sadd myset helloworld
(integer) 1
127.0.0.1:6379> object encoding myset
"hashtable"
127.0.0.1:6379> srem myset helloworld
(integer) 1
127.0.0.1:6379> object encoding myset
"hashtable"
```

5、有序集合（压缩列表和跳跃表）

（1）概况

有序集合与集合一样，元素都不能重复；但与集合不同的是，有序集合中的元素是有顺序的。与列表使用索引下标作为排序依据不同，有序集合为每个元素设置一个分数（score）作为排序依据。

（2）内部编码

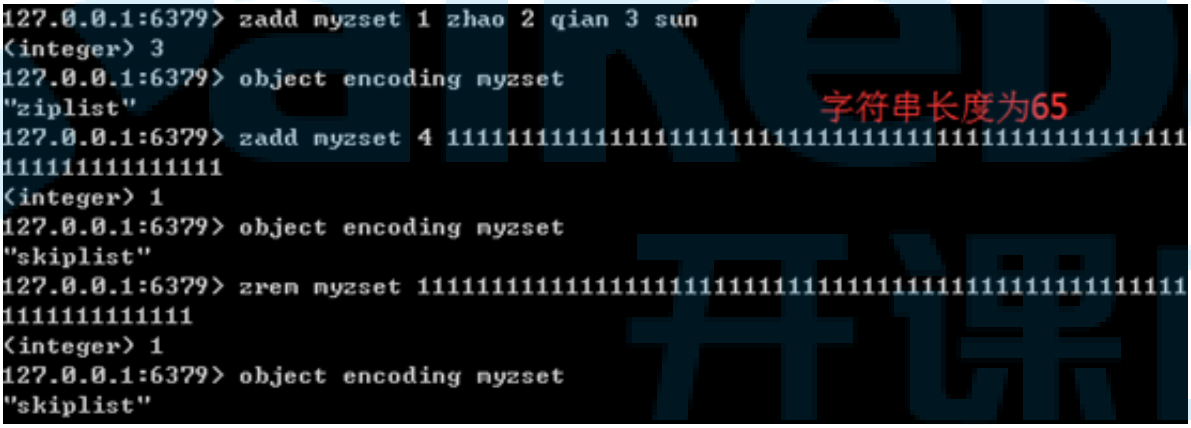
有序集合的内部编码可以是压缩列表（ziplist）或跳跃表（skiplist）。

跳跃表是一种有序数据结构，通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

（3）编码转换

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。

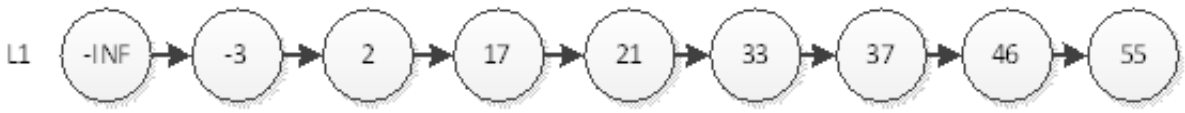
下图展示了有序集合编码转换的特点：



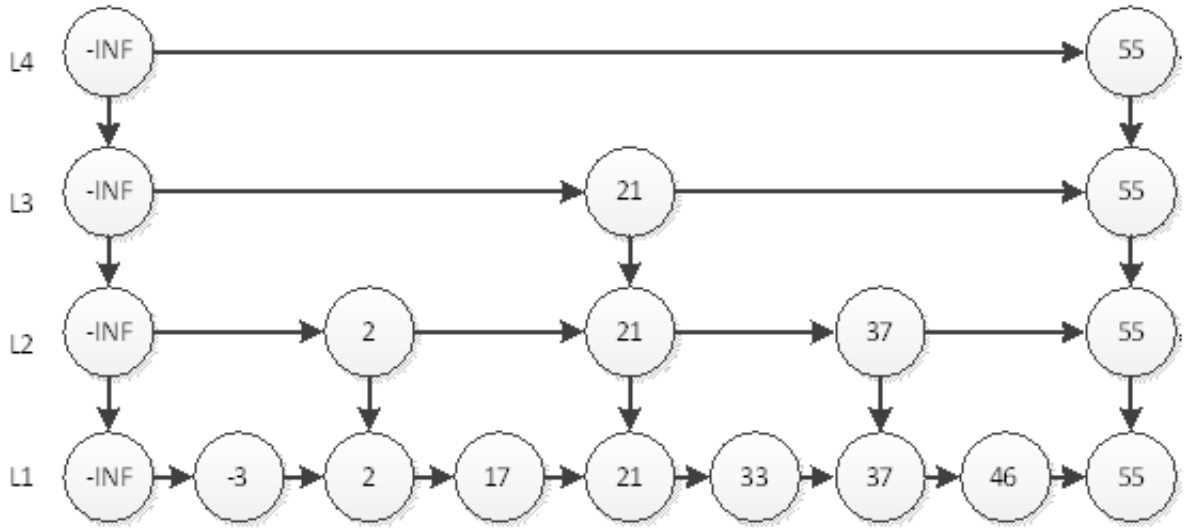
（4）跳跃表

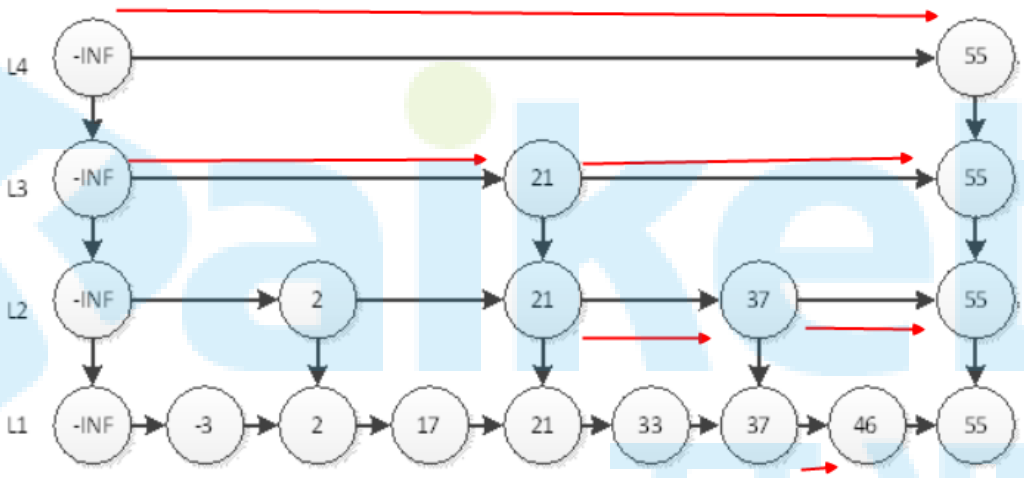
图示

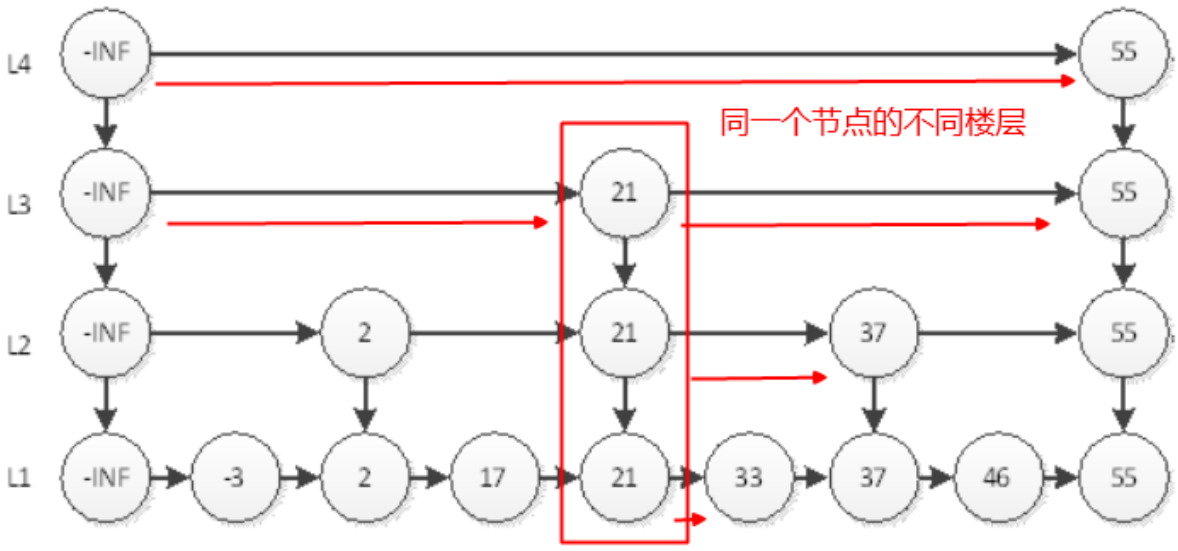
普通单向链表图示：



跳跃表图示：


L4
L3
L2
L1
-INF
-3
2
17
21
33
37
46
55


L4
L3
L2
L1
-INF
-3
2
17
21
33
37
46
55


L4
L3
L2
L1
-INF
-3
2
17
21
33
37
46
55
同一个节点的不同楼层

查询

查找一个节点时，我们只需从高层到低层，一个个链表查找，每次找到该层链表中小于等于目标节点的最大节点，直到找到为止。由于高层的链表迭代时会“跳过”低层的部分节点，所以跳跃表会比正常的链表查找少查部分节点，这也是skiplist名字的由来。

例如：

> 查找46： **55---21---55--37--55--46**

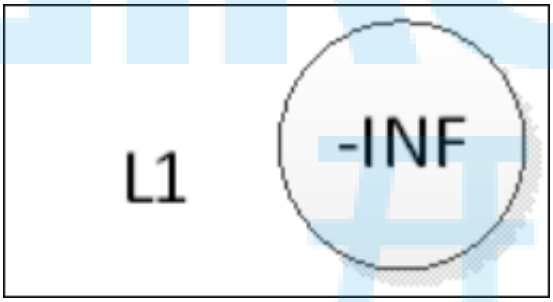
插入

> 跳跃表的数据插入，从楼层的角度来说，是从1层开始插入，至于是否有其他层，需要结合概率算法（抛硬币）。

L1 层

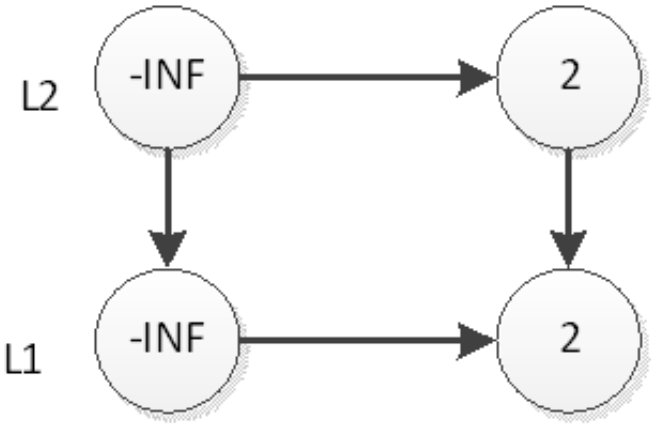
概率算法

在此还是以上图为例：跳跃表的初试状态如下图，表中没有一个元素：

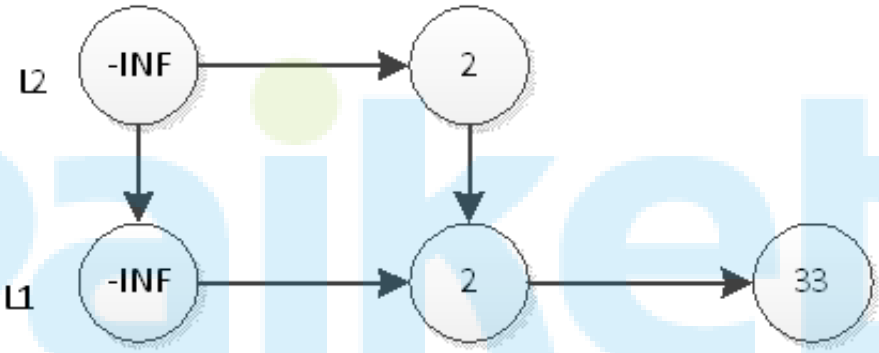


如果我们要插入元素2，首先是在底部插入元素2，如下图：

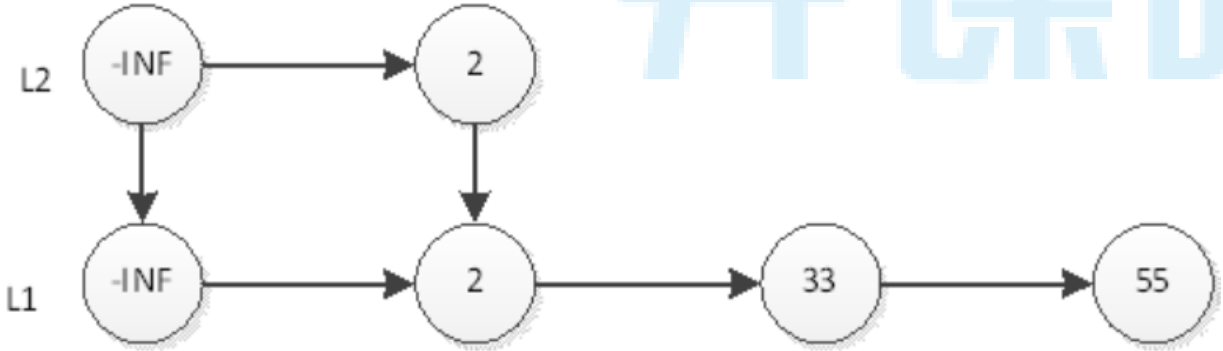
然后我们抛硬币，结果是正面，那么我们要将2插入到L2层，如下图



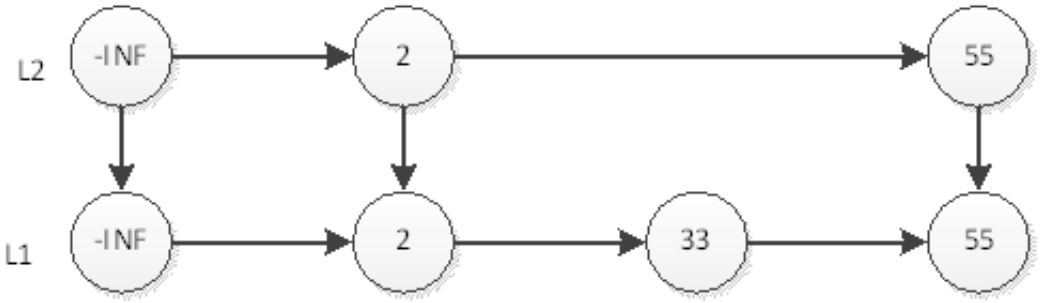
继续抛硬币，结果是反面，那么元素2的插入操作就停止了，插入后的表结构就是上图所示。接下来，我们插入元素33，跟元素2的插入一样，现在L1层插入33，如下图：



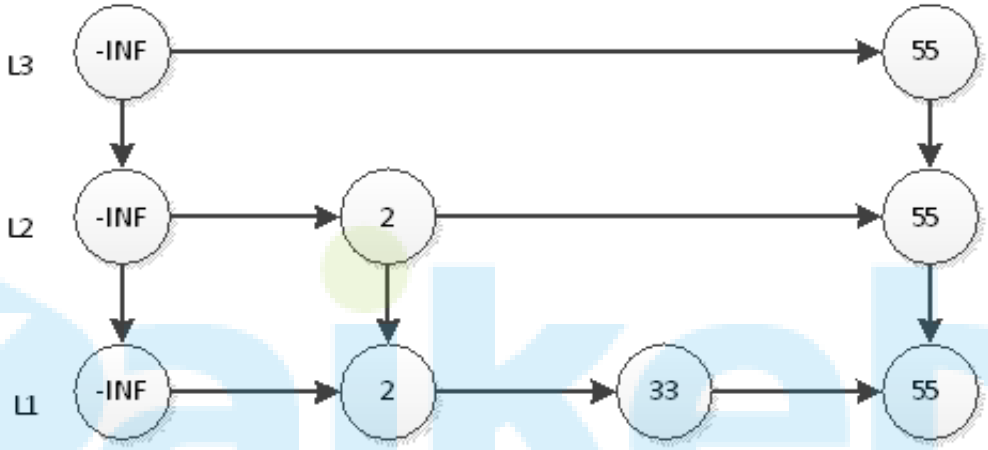
然后抛硬币，结果是反面，那么元素33的插入操作就结束了，插入后的表结构就是上图所示。接下来，我们插入元素55，首先在L1插入55，插入后如下图：



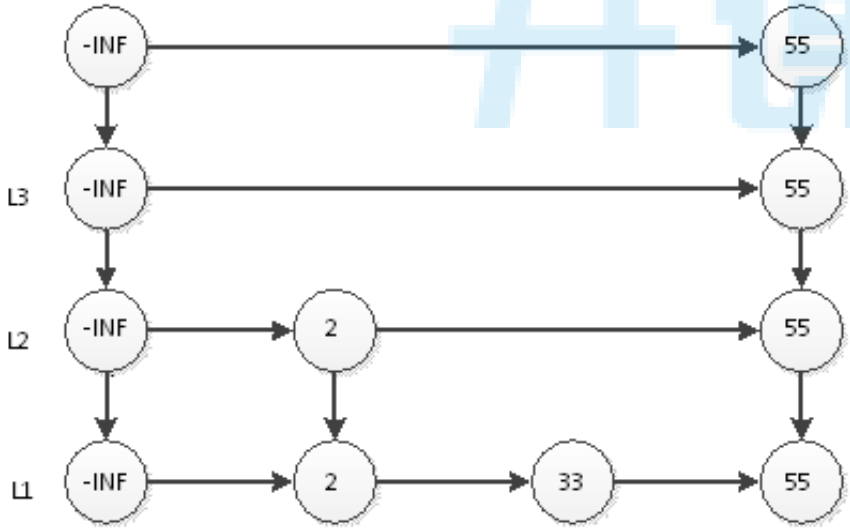
然后抛硬币，结果是正面，那么L2层需要插入55，如下图：



继续抛硬币，结果又是正面，那么L3层需要插入55，如下图：



继续抛硬币，结果又是正面，那么要在L4插入55，结果如下图：



继续抛硬币，结果是反面，那么55的插入结束，表结构就如上图所示。

以此类推，我们插入剩余的元素。当然因为规模小，结果很可能不是一个理想的跳跃表。但是如果元素个数n的规模很大，学过概率论的同学都知道，最终的表结构肯定非常接近于理想跳跃表（隔一个一跳）。

Redis的跳跃表实现

Redis的跳跃表实现由**zskiplist**和**zskiplistNode**两个结构组成：前者用于保存跳跃表信息（如头结点、尾节点、长度等），后者用于表示跳跃表节点。

```
typedef struct zskiplistNode {
      //层
      struct zskiplistLevel{
            //前进指针 后边的节点
            struct zskiplistNode *forward;
            //跨度
            unsigned int span;
      }level[];

      //后退指针
      struct zskiplistNode *backward;
      //分值
      double score;
      //成员对象
      robj *obj;

} zskiplistNode

--链表
typedef struct zskiplist{
      //表头节点和表尾节点
      structz skiplistNode *header, *tail;
      //表中节点的数量
      unsigned long length;
      //表中层数最大的节点的层数
      int level;

}zskiplist;
```

内存优化

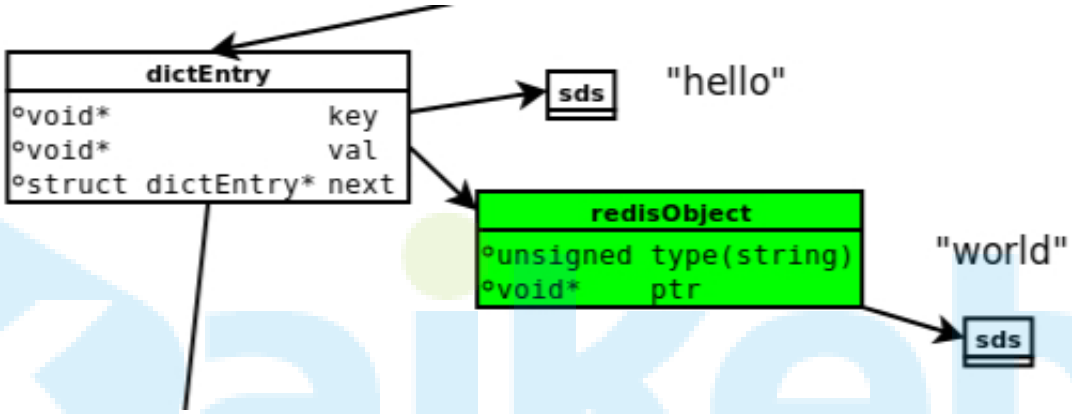
了解Redis的内存模型之后，下面通过几个例子说明其应用。

1、估算Redis内存使用量

下面以最简单的字符串类型来进行说明。

案例1

> 假设有90000个键值对，每个key的长度是7个字节，每个value的长度也是7个字节（且key和value都不是整数）；
>
> set kkk1111 vvv1111
>
> 下面来估算这90000个键值对所占用的空间。在估算占据空间之前，首先可以判定字符串类型使用的编码方式：embstr。
>
> 90000个键值对占据的内存空间主要可以分为两部分：一部分是90000个dictEntry占据的空间；一部分是键值对所需要的bucket空间。



每个dictEntry占据的空间包括：

1) 一个dictEntry结构，24字节，jemalloc会分配32字节的内存块(64位操作系统下，一个指针8字节，一个dictEntry由三个指针组成)

2) 一个key，7字节，所以SDS(key)需要7+9=16个字节（SDS的长度=free+len+9），jemalloc会分配16字节的内存块

3) 一个redisObject，16字节，jemalloc会分配16字节的内存块(4bit+4bit+24bit+4Byte+8Byte=16Byte)

4) 一个value，7字节，所以SDS(value)需要7+9=16个字节（SDS的长度=free+len+9），jemalloc会分配16字节的内存块

5) 综上，一个c所占据的空间需要32+16+16+16=80个字节。

bucket空间：

bucket数组的大小为大于90000的最小的2^n，是131072；每个bucket元素（bucket中存储的都是指针元素）为8字节（因为64位系统中指针大小为8字节）。

因此，可以估算出这90000个键值对占据的内存大小为：90000*80 + 131072*8 + 8+24 = 8248608。

案例2

> 作为对比将key和value的长度由7字节增加到8字节，
>
> set kkkk1111 vvvv1111
>
> set 1000081:comment
>
> 则对应的SDS变为17个字节，jemalloc会分配32个字节，因此每个dictEntry占用的字节数也由80字节变为112字节。此时估算这90000个键值对占据内存大小为：90000*112 + 131072*8 = 11128576。

2、优化内存占用

了解redis的内存模型，对优化redis内存占用有很大帮助。下面介绍几种优化场景。

（1）利用jemalloc特性进行优化

上一小节所讲述的90000个键值便是一个例子。由于jemalloc分配内存时数值是不连续的，因此key/value字符串变化一个字节，可能会引起占用内存很大的变动；在设计时可以利用这一点。

> 例如，如果key的长度如果是8个字节，则SDS为17字节，jemalloc分配32字节；此时将key长度缩减为7个字节，则SDS为16字节，jemalloc分配16字节；则每个key所占用的空间都可以缩小一半。

（2）使用整型/长整型

如果是整型/长整型，Redis会使用int类型（8字节）存储来代替字符串，可以节省更多空间。因此在可以使用长整型/整型代替字符串的场景下，尽量使用长整型/整型。

（3）共享对象

利用共享对象，可以减少对象的创建（同时减少了redisObject的创建），节省内存空间。目前redis中的共享对象只包括10000个整数（0-9999）；可以通过调整REDIS_SHARED_INTEGERS参数提高共享对象的个数；

> 例如将REDIS_SHARED_INTEGERS调整到20000，则0-19999之间的对象都可以共享。

> 考虑这样一种场景：论坛网站在redis中存储了每个帖子的浏览数，而这些浏览数绝大多数分布在0-20000之间，这时候通过适当增大REDIS_SHARED_INTEGERS参数，便可以利用共享对象节省内存空间。

（4）避免过度设计

然而需要注意的是，不论是哪种优化场景，都要考虑内存空间与设计复杂度的权衡；而设计复杂度会影响到代码的复杂度、可维护性。

如果数据量较小，那么为了节省内存而使得代码的开发、维护变得更加困难并不划算；还是以前面讲到的90000个键值对为例，实际上节省的内存空间只有几MB。但是如果数据量有几千万甚至上亿，考虑内存的优化就比较必要了。

3、关注内存碎片率

内存碎片率是一个重要的参数，对redis 内存的优化有重要意义。

如果内存碎片率过高（jemalloc在1.03左右比较正常），说明内存碎片多，内存浪费严重；这时便可以考虑重启redis服务，在内存中对数据进行重排，减少内存碎片。

如果内存碎片率小于1，说明redis内存不足，部分数据使用了虚拟内存（即swap）；由于虚拟内存的存取速度比物理内存差很多（2-3个数量级），此时redis的访问速度可能会变得很慢。因此必须设法增大物理内存（可以增加服务器节点数量，或提高单机内存），或减少redis中的数据。

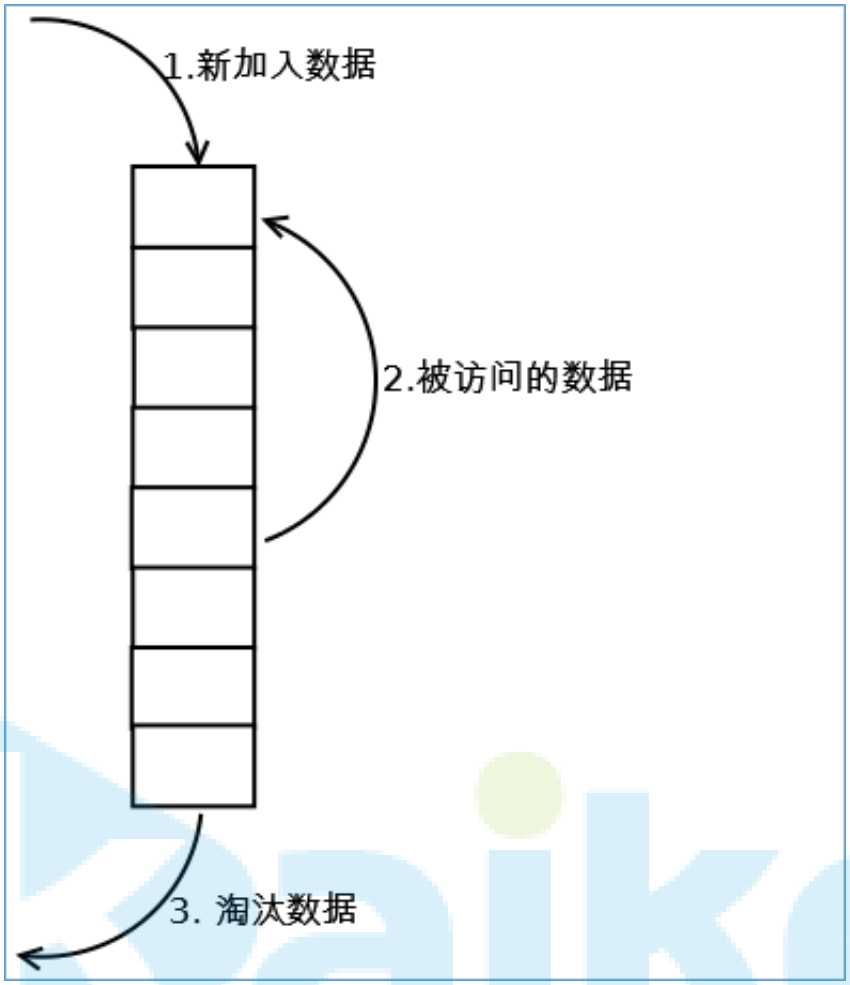
要减少redis中的数据，除了选用合适的数据类型、利用共享对象等，还有一点是要设置合理的数据回收策略（maxmemory-policy），当内存达到一定量后，根据不同的优先级对内存进行回收。

缓存淘汰策略

LRU原理

LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

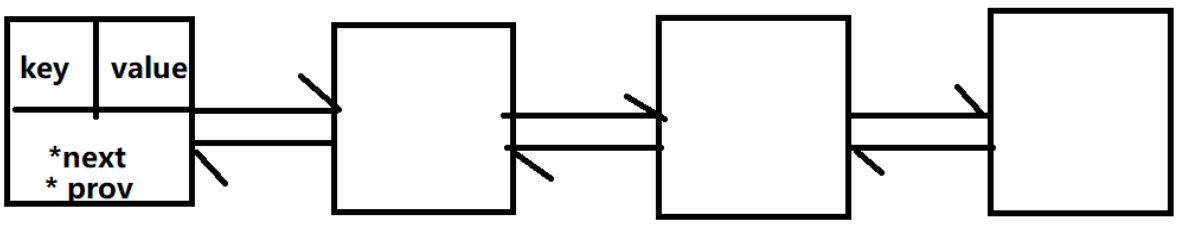
最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：



1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

在Java中可以使用**LinkHashMap**去实现LRU

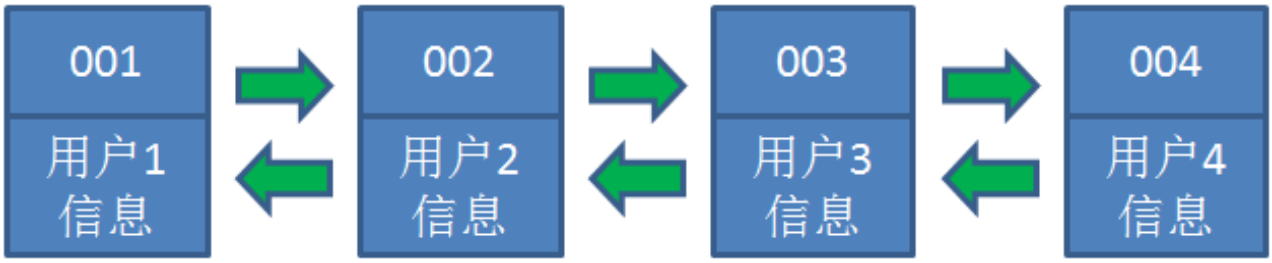
利用哈希链表实现



案例分析

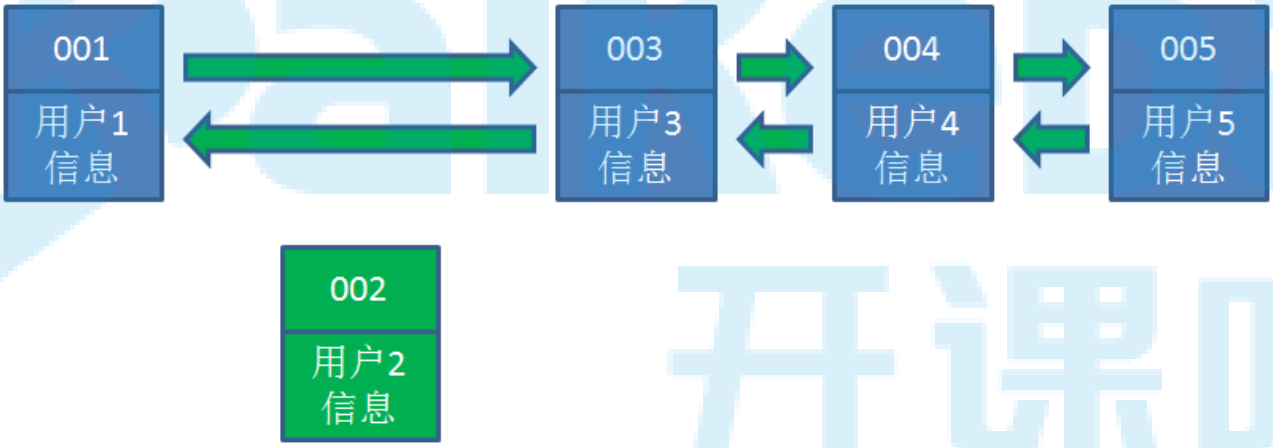
> 让我们以用户信息的需求为例，来演示一下LRU算法的基本思路：

1.假设我们使用哈希链表来缓存用户信息，目前缓存了4个用户，这4个用户是按照时间顺序依次从链表右端插入的。

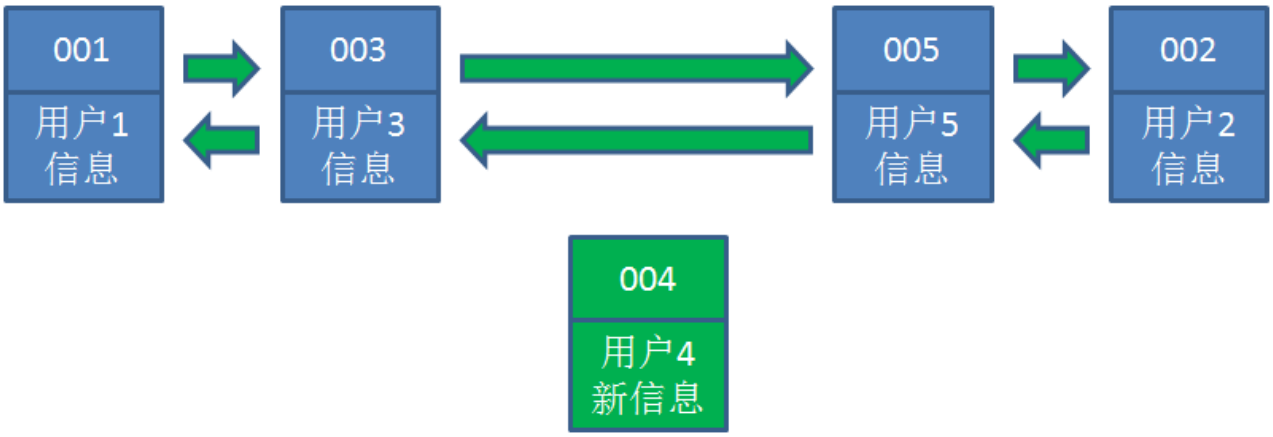


2.此时，业务方访问用户5，由于哈希链表中没有用户5的数据，我们从数据库中读取出来，插入到缓存当中。这时候，链表中最右端是最新访问到的用户5，最左端是最近最少访问的用户1。

3.接下来，业务方访问用户2，哈希链表中存在用户2的数据，我们怎么做呢？我们把用户2从它的前驱节点和后继节点之间移除，重新插入到链表最右端。这时候，链表中最右端变成了最新访问到的用户2，最左端仍然是最近最少访问的用户1。



4.接下来，业务方请求修改用户4的信息。同样道理，我们把用户4从原来的位置移动到链表最右侧，并把用户信息的值更新。这时候，链表中最右端是最新访问到的用户4，最左端仍然是最近最少访问的用户1。



5.后来业务方换口味了，访问用户6，用户6在缓存里没有，需要插入到哈希链表。假设这时候缓存容量已经达到上限，必须先删除最近最少访问的数据，那么位于哈希链表最左端的用户1就会被删除掉，然后再把用户6插入到最右端。

以上，就是LRU算法的基本思路。

https://www.itcodemonkey.com/article/11153.html

Redis缓存淘汰策略

设置最大缓存

> 在 redis 中，允许用户设置最大使用内存大小maxmemory，默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。

redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

淘汰策略

> redis淘汰策略配置：maxmemory-policy voltile-lru，支持热配置

redis 提供 6种数据淘汰策略：

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-enviction（驱逐）：禁止驱逐数据

Redis事务

Redis事务典型应用—Redis乐观锁

在生产环境里，经常会利用redis乐观锁来实现秒杀，Redis乐观锁是Redis事务的经典应用。

秒杀场景描述:

秒杀活动对稀缺或者特价的商品进行定时，定量售卖，吸引成大量的消费者进行抢购，但又只有少部分消费者可以下单成功。因此，秒杀活动将在较短时间内产生比平时大数十倍，上百倍的页面访问流量和下单请求流量。

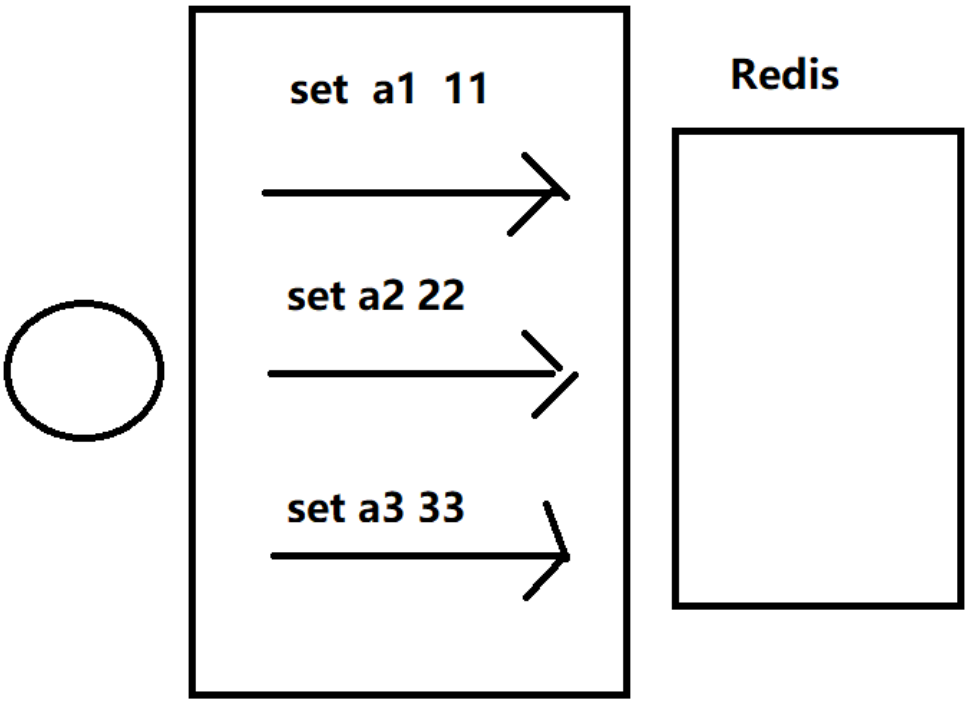
由于秒杀只有少部分请求能够成功，而大量的请求是并发产生的，所以如何确定哪个请求成功了，就是由redis乐观锁来实现。具体思路如下：

监控 锁定量，如果该值被修改成功则表示该请求被通过，反之表示该请求未通过。

从监控到修改到执行都需要在redis里操作，这样就需要用到Redis事务。

Redis事务介绍

- `Redis`的事务是通过`MULTI`、`EXEC`、`DISCARD`和`WATCH`这四个命令来完成的。
- `Redis`的单个命令都是原子性的，所以这里需要确保事务性的对象是**命令集合**。
- `Redis`将命令集合序列化并确保处于同一事务的**命令集合连续且不被打断**的执行
- `Redis`不支持回滚操作。



在一个事务中 处理命令集不会被干扰

Redis是单进程单线程的，所以不会出现线程并发。

事务命令

MULTI

用于标记事务块的开始。
Redis会将后续的命令逐个放入队列中，然后使用EXEC命令原子化地执行这个命令序列。

语法：

```
multi
```

EXEC

在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态

语法：

```
exec
```

DISCARD

> 清除所有先前在一个事务中放入队列的命令，然后恢复正常的连接状态。

语法：

```
discard
```

WATCH

> 当某个[事务需要按条件执行]时，就要使用这个命令将给定的[键设置为受监控]的状态。

语法：

```
watch key [key…]
```

注意事项：使用该命令可以实现`Redis`的**乐观锁**。（后面实现）

UNWATCH

> 清除所有先前为一个事务监控的键。

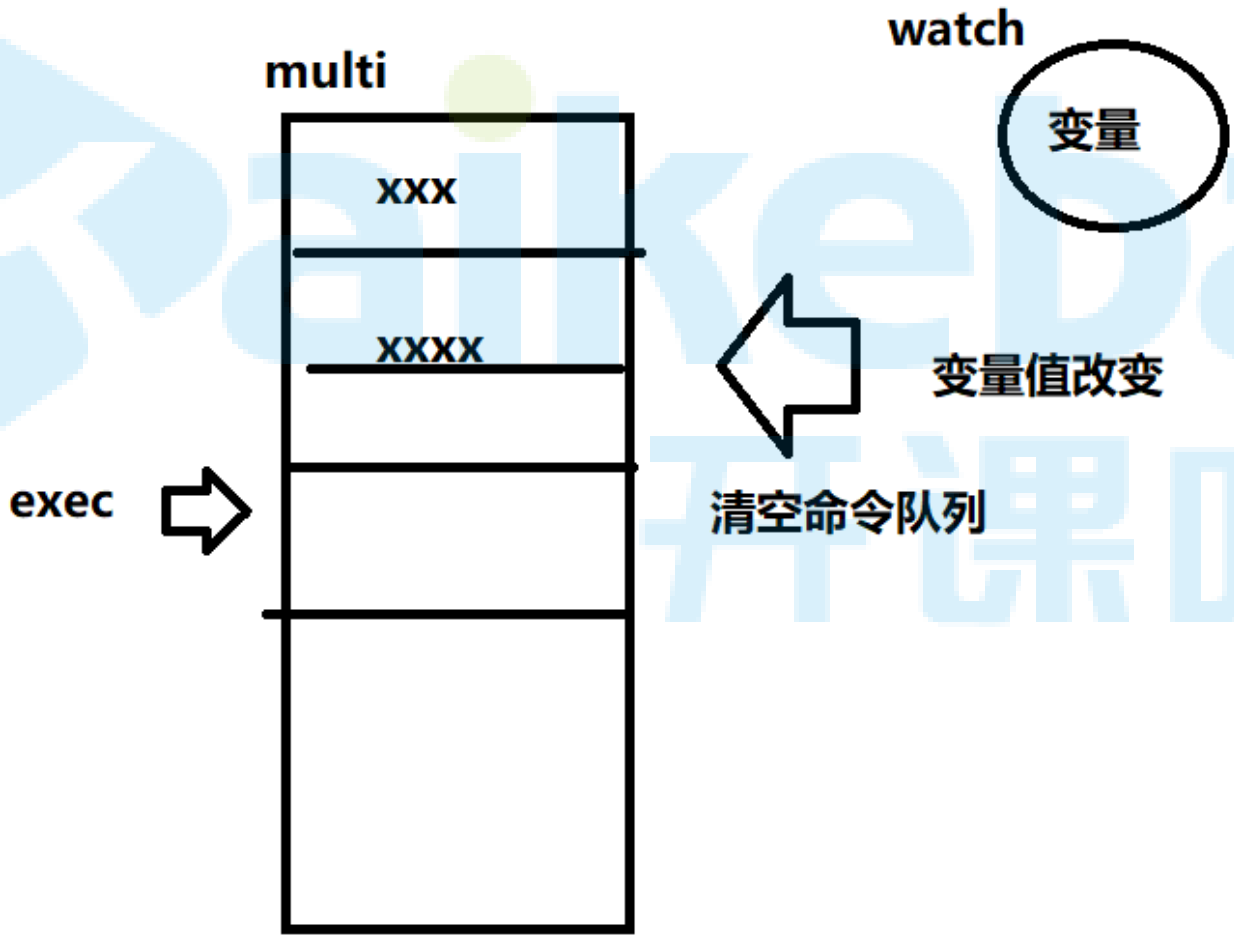
语法：

```
unwatch
```

事务演示

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s1 111
QUEUED
127.0.0.1:6379> hset set1 name zhangsan
QUEUED
127.0.0.1:6379> exec
1) OK
2) (integer) 1
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s2 222
QUEUED
127.0.0.1:6379> hset set2 age 20
QUEUED
```

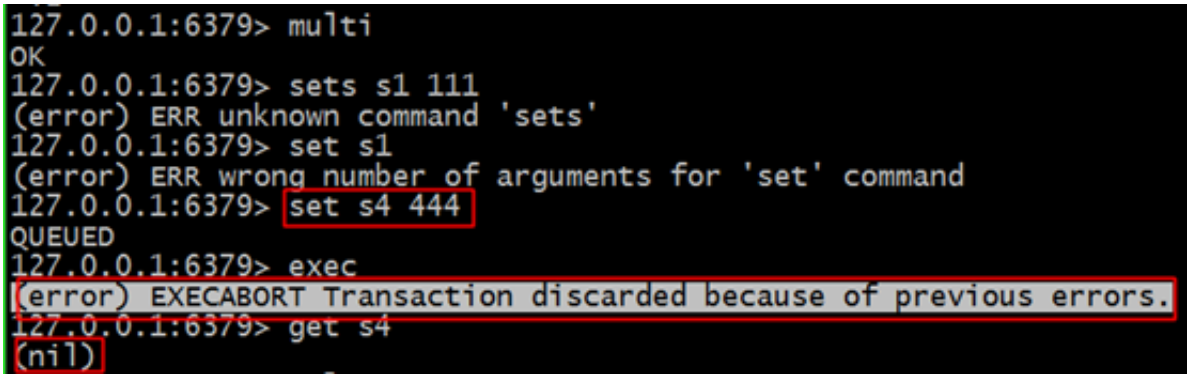
```
127.0.0.1:6379> discard
OK
127.0.0.1:6379> exec
(error) ERR EXEC without MULTI

127.0.0.1:6379> watch s1
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s1 555
QUEUED
127.0.0.1:6379> exec      # 此时在没有exec之前，通过另一个命令窗口对监控的s1字段进行修改
(nil)
127.0.0.1:6379> get s1
111
```



事务失败处理

- `Redis`语法错误

  整个事务的命令在队列里都清除



- `Redis`运行错误

  在队列里正确的命令可以执行 （弱事务性）

  弱事务性：

  1、在队列里正确的命令可以执行 （非原子操作）

  2、不支持回滚

- `Redis`不支持事务回滚（为什么呢）

  1、大多数事务失败是因为**语法错误或者类型错误**，这两种错误，在开发阶段都是可以预见的

  2、`Redis`为了**性能方面**就忽略了事务回滚。 （回滚记录历史版本）

Redis乐观锁

乐观锁基于CAS（Compare And Swap）思想（比较并替换），是不具有互斥性，不会产生锁等待而消耗资源，但是需要反复的重试，但也是因为重试的机制，能比较快的响应。因此我们可以利用redis来实现乐观锁。具体思路如下：

1、利用redis的watch功能，监控这个redisKey的状态值
2、获取redisKey的值
3、创建redis事务
4、给这个key的值+1
5、然后去执行这个事务，如果key的值被修改过则回滚，key不加1

```
public void watch() {
    try {
        String watchKeys = "watchKeys";
```

```
        //初始值 value=1
        jedis.set(watchKeys, 1);
        //监听key为watchKeys的值
        jedis.watch(watchkeys);

        //开启事务
        Transaction tx = jedis.multi();

        //watchKeys自增加一
        tx.incr(watchKeys);

        //执行事务，如果其他线程对watchKeys中的value进行修改，则该事务将不会执行
        //通过redis事务以及watch命令实现乐观锁
        List<Object> exec = tx.exec();
        if (exec == null) {
            System.out.println("事务未执行");
        } else {
            System.out.println("事务成功执行，watchKeys的value成功修改");
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        jedis.close();
    }
}
```

Redis乐观锁实现秒杀

```
public class SecKill {
  public static void main(String[] arg) {
        //库存key
    String redisKey = "stock";

    ExecutorService executorService = Executors.newFixedThreadPool(20);
    try {
      Jedis jedis = new Jedis("127.0.0.1", 6378);
      // 可以被秒杀的库存的初始值，库存总共20个
      jedis.set(redisKey, "0");
      jedis.close();
    } catch (Exception e) {
      e.printStackTrace();
    }

    for (int i = 0; i < 1000; i++) {

      executorService.execute(() -> {
```

```
        Jedis jedis1 = new Jedis("127.0.0.1", 6378);
        try {
          jedis1.watch(redisKey);
          String redisValue = jedis1.get(redisKey);
          int valInteger = Integer.valueOf(redisValue);
          String userInfo = UUID.randomUUID().toString();

          // 没有秒完
          if (valInteger < 20) {
            Transaction tx = jedis1.multi();
            tx.incr(redisKey);
            List list = tx.exec();
            // 秒成功   失败返回空list而不是空
            if (list != null && list.size() > 0) {

              System.out.println("用户: " + userInfo + ", 秒杀成功! 当前成功人数: "
+ (valInteger + 1));

            }
            // 版本变化，被别人抢了。
            else {
              System.out.println("用户: " + userInfo + ", 秒杀失败");
            }
          }
          // 秒完了
          else {
            System.out.println("已经有20人秒杀成功，秒杀结束");

          }
        } catch (Exception e) {
          e.printStackTrace();
        } finally {
          jedis1.close();
        }

      });
    }
    executorService.shutdown();
  }
}
```

Redis持久化

Redis是一个**内存**数据库，为了保证数据的持久性，它提供了两种持久化方案：

RDB方式（默认）

RDB方式是通过**快照**（`snapshotting`）完成的，当**符合一定条件**时Redis会自动将内存中的数据进行快照并持久化到硬盘。

触发快照的时机

1. 符合自定义配置的快照规则　[redis.conf]
2. 执行save或者bgsave命令
3. 执行flushall命令
4. 执行主从复制操作 (第一次)

设置快照规则

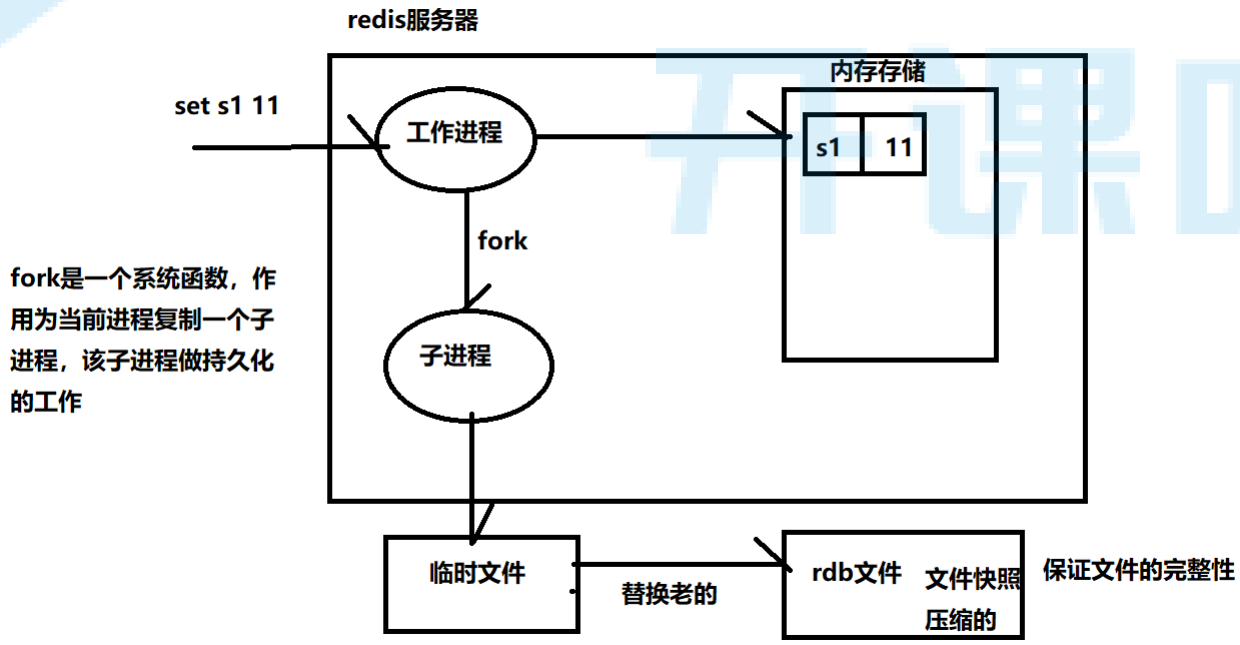
save　　多少秒内 数据变了多少

save "" ：不使用RDB存储

save 900 1 ： 表示15分钟（900秒钟）内至少1个键被更改则进行快照。
save 300 10： 表示5分钟（300秒）内至少10个键被更改则进行快照。
save 60 10000： 表示1分钟内至少10000个键被更改则进行快照。

过滤条件是或的关系，而且是漏斗型的过滤顺序。

原理图



注意事项

1. `Redis`在进行快照的过程中不会修改`RDB`文件，只有快照操作结束后才会将旧的文件替换成新的，也就是说任何时候`RDB`文件都是完整的。
2. 这就使得我们可以通过定时备份`RDB`文件来实现`Redis`数据库的备份， `RDB`文件是`经过压缩的二进制文件`，占用的空间会小于内存中的数据，更加利于传输。

RDB优缺点

- **缺点：**使用`RDB`方式实现持久化，一旦`Redis`**异常退出，就会丢失最后一次快照以后更改的所有数据**。这个时候我们就需要根据具体的应用场景，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受范围。如果数据相对来说比较重要，希望将损失降到最小，则可以使用`AOF`方式进行持久化
- **优点：** `RDB`可以最大化`Redis`的性能：父进程在保存`RDB`文件时唯一要做的就是`fork`出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无需执行任何磁盘`I/O`操作。同时这个也是一个缺点，如果数据集比较大的时候， `fork`可以能比较耗时，造成服务器在一段时间内停止处理客户端的请求；

AOF方式

AOF介绍

默认情况下`Redis`没有开启`AOF`（`append only file`）方式的持久化。

开启`AOF`持久化后，每执行一条会**更改`Redis`中的数据的命令**，`Redis`就会将该命令写入硬盘中的`AOF`文件，这一过程显然**会降低`Redis`的性能**，但大部分情况下这个影响是能够接受的，另外使**用较快的硬盘可以提高`AOF`的性能**。

`redis.conf`：

```
# 可以通过修改redis.conf配置文件中的appendonly参数开启
appendonly yes

# 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。
dir    ./

# 默认的文件名是appendonly.aof，可以通过appendfilename参数修改
appendfilename    appendonly.aof
```

用SET命令来举例说明RESP协议的格式。

```
redis> SET mykey "Hello"
"OK"
```

实际发送的请求数据：

```
*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$5\r\nHello\r\n

*3
$3
SET
$5
mykey
$5
Hello
```

同步磁盘数据

`Redis`每次更改数据的时候， `aof`机制都会将命令记录到`aof`文件，但是实际上由于操作系统的**缓存机制**，数据并没有实时写入到硬盘，而是进入**硬盘缓存**。再通过**硬盘缓存机制**去刷新到保存到文件。

filesync命令

参数说明：

```
# 每次执行写入都会进行同步， 这个是最安全但是是效率比较低的方式
appendfsync always

# 每一秒执行(默认)
appendfsync everysec

# 不主动进行同步操作，由操作系统去执行，这个是最快但是最不安全的方式
appendfsync no
```

AOF重写原理（优化AOF文件）

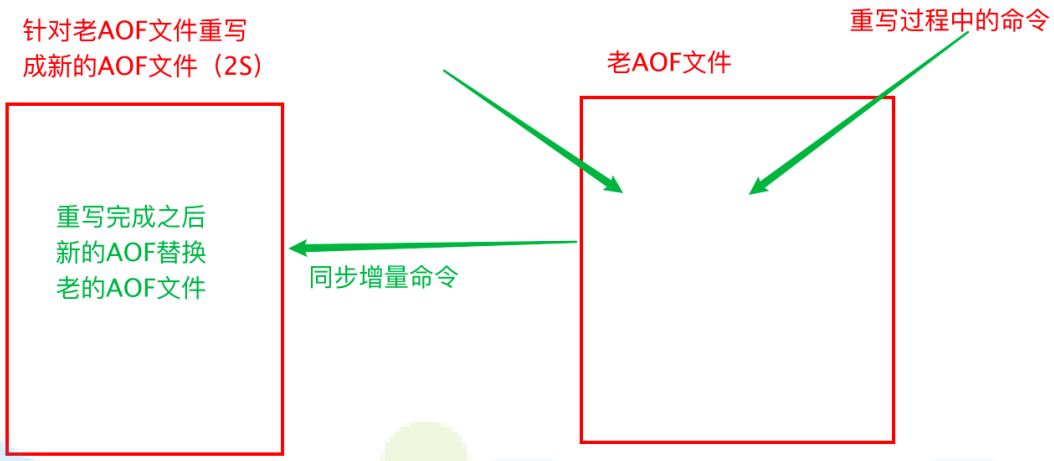
```
set s1 111

set s1 222

set s1 333

set s1 444
```

```
set s1 444
```

> `Redis` 可以在 `AOF` 文件体积变得过大时，自动地在后台对 `AOF` 进行重写。重写后的新 `AOF` 文件包含了恢复当前数据集所需的最小命令集合。

`AOF` 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 `Redis` 协议（RESP）的格式保存， 因此 `AOF` 文件的内容非常容易被人读懂， 对文件进行分析（`parse`）也很轻松。

**重写过程分析（整个重写操作是绝对安全的）：



`Redis` 在创建新 `AOF` 文件的过程中，会继续将命令追加到现有的 `AOF` 文件里面，即使重写过程中发生停机，现有的 `AOF` 文件也不会丢失。 而一旦新 `AOF` 文件创建完毕， `Redis` 就会从旧 `AOF` 文件切换到新 `AOF` 文件，并开始对新 `AOF` 文件进行追加操作。

参数说明：

```
# 表示当前aof文件大小超过上一次aof文件大小的百分之多少的时候会进行重写。如果之前没有重写过，以启动时aof文件大小为准
auto-aof-rewrite-percentage 100

# 限制允许重写最小aof文件大小，也就是文件大小小于64mb的时候，不需要进行优化
auto-aof-rewrite-min-size 64mb
```

AOF文件损坏以后如何修复

问题描述：

> 服务器可能在程序正在对 `AOF` 文件进行写入时停机， 如果停机造成了 `AOF` 文件出错（`corrupt`）， 那么 `Redis` 在重启时会拒绝载入这个 `AOF` 文件， 从而确保数据的一致性不会被破坏。

当发生这种情况时， 可以用以下方法来修复出错的 `AOF` 文件：

- 为现有的 `AOF` 文件创建一个备份。

- 使用 `Redis` 附带的 `redis-check-aof`程序，对原来的 `AOF` 文件进行修复。

```
redis-check-aof --fix readonly.aof
```

- 重启 `Redis` 服务器，等待服务器载入修复后的 `AOF` 文件，并进行数据恢复。

如何选择RDB和AOF

一般来说,如果对数据的安全性要求非常高的话，应该同时使用两种持久化功能。

如果可以承受数分钟以内的数据丢失，那么可以只使用 `RDB` 持久化。

有很多用户都只使用 `AOF` 持久化， 但并不推荐这种方式： 因为定时生成 `RDB` 快照（`snapshot`）非常便于进行数据库备份， 并且 `RDB` 恢复数据集的速度也要比 `AOF` 恢复的速度要快 。

```
# 禁止RDB方式
save ""
```

两种持久化策略可以同时使用，也可以使用其中一种。如果同时使用的话， 那么`Redis`重启时，会优先使用`AOF`文件来还原数据。

如何选择RDB和AOF（4.0之前的还需要考虑）

内存数据库　rdb（redis database）+aof　数据不能丢

缓存服务器　rdb

不建议 只使用 aof (性能差)

恢复时： 先aof再rdb

1.3 混合持久化方式

- Redis 4.0 之后新增的方式，混合持久化是结合了 RDB 和 AOF 的优点，在写入的时候，先把当前的数据以 RDB 的形式写入文件的开头，再将后续的操作命令以 AOF 的格式存入文件，这样既能保证 Redis 重启时的速度，又能减低数据丢失的风险。

RDB 和 AOF 持久化各有利弊，RDB 可能会导致一定时间内的数据丢失，而 AOF 由于文件较大则会影响 Redis 的启动速度，为了能同时拥有 RDB 和 AOF 的优点，Redis 4.0 之后新增了混合持久化的方式，因此我们在必须要进行持久化操作时，应该选择混合持久化的方式。

查询是否开启混合持久化可以使用 `config get aof-use-rdb-preamble` 命令，执行结果

```
127.0.0.1:6379> config get aof-use-rdb-preamble
1) "aof-use-rdb-preamble"
2) "yes"
```

其中 yes 表示已经开启混合持久化，no 表示关闭，Redis 5.0 默认值为 yes。如果是其他版本的 Redis 首先需要检查一下，是否已经开启了混合持久化，如果关闭的情况下，可以通过以下两种方式开启：

- 通过命令行开启
- 通过修改 Redis 配置文件开启

① 通过命令行开启

使用命令 `config set aof-use-rdb-preamble yes`

命令行设置配置的缺点是重启 Redis 服务之后，设置的配置就会失效。

② 通过修改 Redis 配置文件开启

在 Redis 的根路径下找到 redis.conf 文件，把配置文件中的 `aof-use-rdb-preamble no` 改为 `aof-use-rdb-preamble yes`

配置完成之后，需要重启 Redis 服务器，配置才能生效，但修改配置文件的方式，在每次重启 Redis 服务之后，配置信息不会丢失。

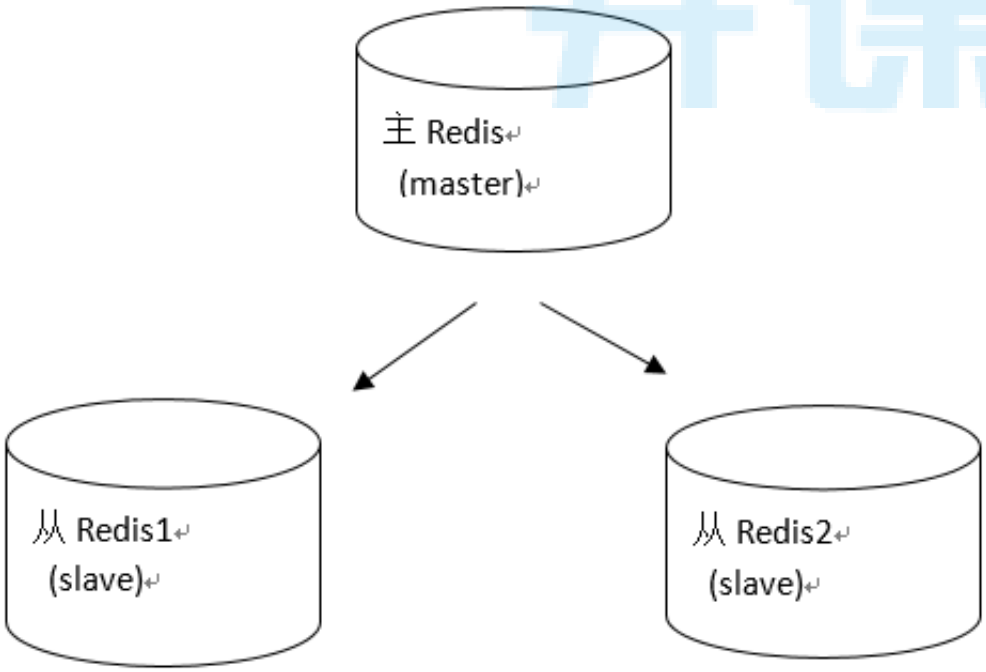
需要注意的是，在非必须进行持久化的业务中，可以关闭持久化，这样可以有效的提升 Redis 的运行速度，不会出现间歇性卡顿的困扰。

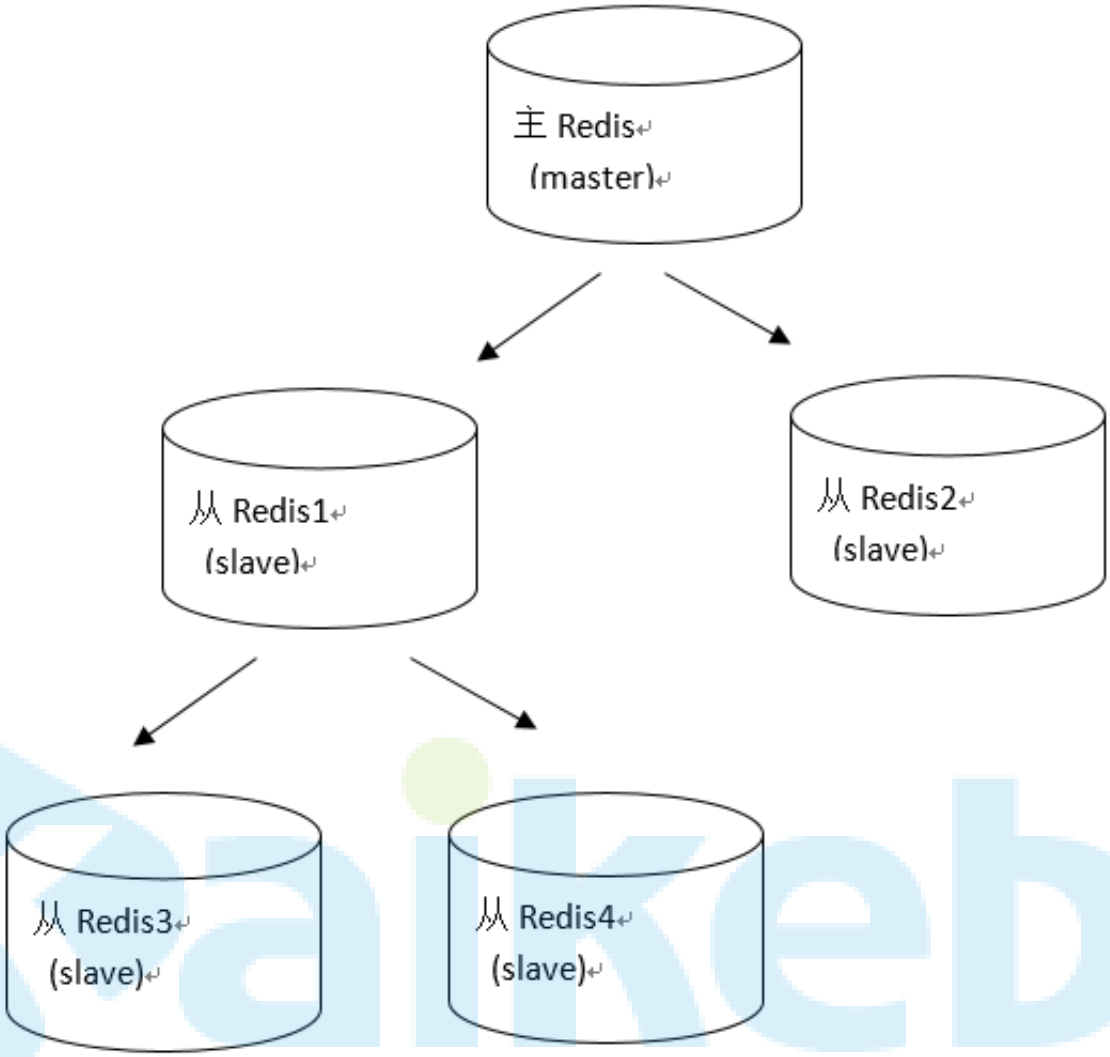
混合持久化 是RDB+指令

如果AOF文件过大时候，会重写：把当前数据已RDB格式保存，后续指令用aof

Redis主从复制

什么是主从复制





主对外从对内，主可写从不可写

主挂了，从不可为主

主从配置

主Redis配置

无需特殊配置

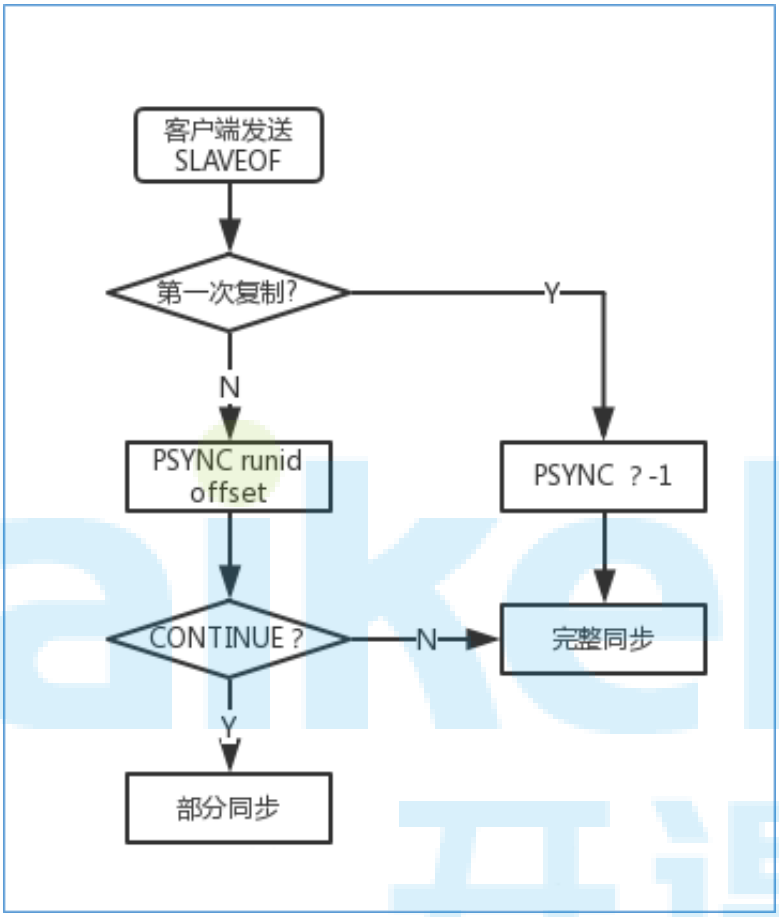
从Redis配置

修改从服务器上的`redis.conf`文件：

```
# slaveof <masterip> <masterport>
# 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。
slaveof 127.0.0.1 6379
replicaof <masterip> <masterport>
```

实现原理

- `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）。

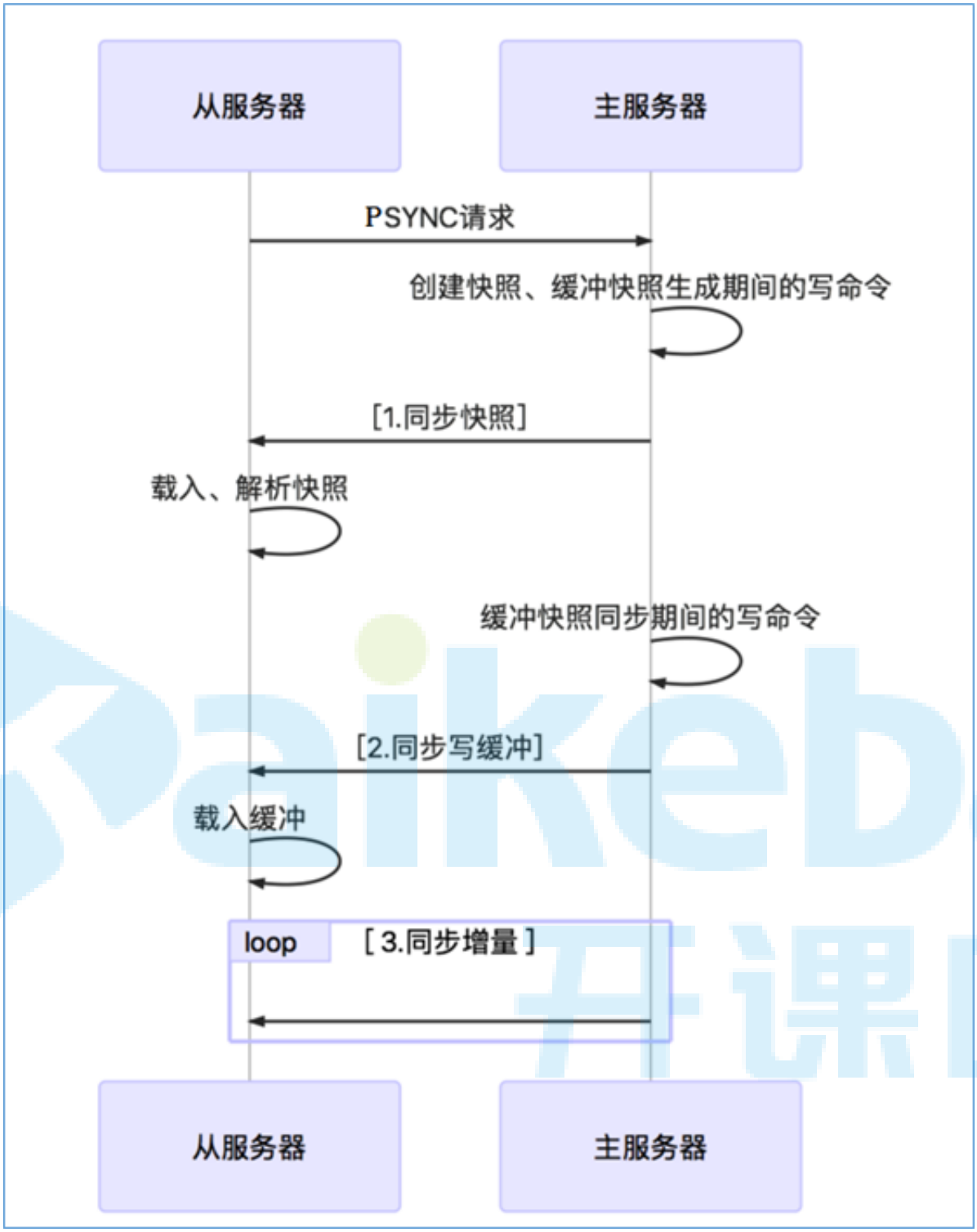


- 除此之外的情况都是**增量同步**。

全量同步

`Redis`的全量同步过程主要分三个阶段：

- **同步快照阶段：**`Master`创建并发送**快照**RDB给`Slave`，`Slave`载入并解析快照。`Master`同时将此阶段所产生的新的写命令存储到缓冲区。
- **同步写缓冲阶段：**`Master`向`Slave`同步存储在缓冲区的写操作命令。
- **同步增量阶段：**`Master`向`Slave`同步写操作命令。



增量同步

- Redis`增量同步主要指**`Slave`完成初始化后开始正常工作时，`Master`发生的写操作同步到`Slave`的过程**。
- 通常情况下，`Master`每执行一个写命令就会向`Slave`发送相同的**写命令**，然后`Slave`接收并执行。

Redis哨兵机制

Redis主从复制的缺点：没有办法对master进行动态选举（master宕机后，需要重新选举master），需要使用Sentinel机制完成动态选举。

简介

Redis的哨兵模式到了2.8版本之后

Sentinel（哨兵）进程是用于监控Redis集群中Master主服务器工作的状态

在Master主服务器发生故障的时候，可以实现Master和Slave服务器的切换，保证系统的高可用（HA）

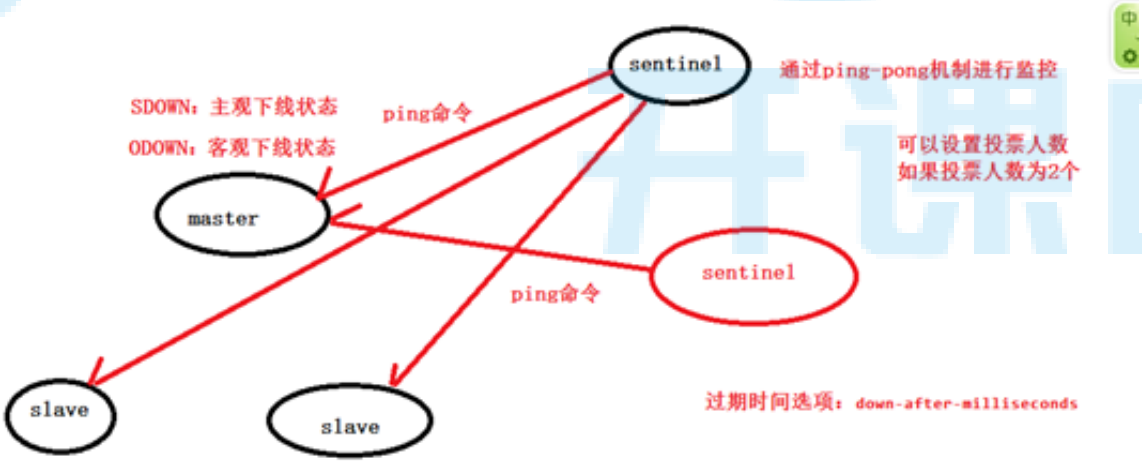
哨兵进程的作用

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

故障判定原理分析



1. 每个Sentinel（哨兵）进程以**每秒钟一次**的频率向整个集群中的Master主服务器，Slave从服务器以及其他Sentinel（哨兵）进程发送一个PING命令。
2. 如果一个实例（instance）距离最后一次有效回复PING命令的时间超过down-after-milliseconds选项所指定的值， 则这个实例会被Sentinel（哨兵）进程标记为**主观下线**（**SDOWN**）。
3. 如果一个Master主服务器被标记为主观下线（SDOWN），则正在监视这个Master主服务器的**所有 Sentinel（哨兵）**进程要以每秒一次的频率**确认Master主服务器**的确**进入了主观下线状态**。
4. 当**有足够数量的 Sentinel（哨兵）**进程（大于等于配置文件指定的值）在指定的时间范围内确认Master主服务器进入了主观下线状态（SDOWN）， 则Master主服务器会被标记为**客观下线**

（**ODOWN**）。

5. 在一般情况下， 每个 Sentinel （哨兵）进程会以每 10 秒一次的频率向集群中的所有Master主服务器、 Slave从服务器发送 INFO 命令。
6. 当 Master主服务器被 Sentinel （哨兵）进程标记为**客观下线（ODOWN）**时， Sentinel （哨兵）进程向下线的 Master主服务器的所有 Slave从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。
7. 若没有足够数量的 Sentinel （哨兵）进程同意 Master主服务器下线， Master主服务器的客观下线状态就会被移除。若 Master主服务器重新向 Sentinel （哨兵）进程发送 PING 命令返回有效回复， Master主服务器的主观下线状态就会被移除。

自动故障迁移

- 它会将失效 Master 的其中一个 Slave 升级为新的 Master , 并让失效 Master 的其他 Slave 改为复制新的 Master ；
- 当客户端试图连接失效的 Master 时，集群也会向客户端返回新 Master 的地址，使得集群可以使用现在的 Master 替换失效 Master 。
- Master 和 Slave 服务器切换后， Master 的 redis.conf 、 Slave 的 redis.conf 和 sentinel.conf 的配置文件的内容都会发生相应的改变，即， Master 主服务器的 redis.conf 配置文件中会多一行 slaveof 的配置， sentinel.conf 的监控目标会随之调换。

案例演示

- 修改从机的 **sentinel.conf** ：

```
# 哨兵sentinel监控的redis主节点的 ip port
# master-name  可以自己命名的主节点名字 只能由字母A-z、数字0-9 、这三个字符".-_"组成。
# quorum 当这些quorum个数sentinel哨兵认为master主节点失联 那么这时 客观上认为主节点失联了
# sentinel monitor <master-name> <master ip> <master port> <quorum>
sentinel monitor mymaster 192.168.10.133 6379 1
```

- **其他配置项说明**

sentinel.conf

```
# 哨兵sentinel实例运行的端口 默认26379
port 26379

# 哨兵sentinel的工作目录
dir /tmp

# 哨兵sentinel监控的redis主节点的 ip port
# master-name  可以自己命名的主节点名字 只能由字母A-z、数字0-9 、这三个字符".-_"组成。
# quorum 当这些quorum个数sentinel哨兵认为master主节点失联 那么这时 客观上认为主节点失联了
# sentinel monitor <master-name> <ip> <redis-port> <quorum>
sentinel monitor mymaster 127.0.0.1 6379 1
```

```
# 当在Redis实例中开启了requirepass foobared 授权密码 这样所有连接Redis实例的客户端都要提供密码
# 设置哨兵sentinel 连接主从的密码 注意必须为主从设置一样的验证密码
# sentinel auth-pass <master-name> <password>
sentinel auth-pass mymaster MySUPER--secret-0123passw0rd


# 指定多少毫秒之后 主节点没有应答哨兵sentinel 此时 哨兵主观上认为主节点下线 默认30秒
# sentinel down-after-milliseconds <master-name> <milliseconds>
sentinel down-after-milliseconds mymaster 30000

# 这个配置项指定了在发生failover主备切换时最多可以有多少个slave同时对新的master进行 同步，这个数字越小，完成failover所需的时间就越长，
但是如果这个数字越大，就意味着越 多的slave因为replication而不可用。
可以通过将这个值设为 1 来保证每次只有一个slave 处于不能处理命令请求的状态。
# sentinel parallel-syncs <master-name> <numslaves>
sentinel parallel-syncs mymaster 1


# 故障转移的超时时间 failover-timeout 可以用在以下这些方面：
#1. 同一个sentinel对同一个master两次failover之间的间隔时间。
#2. 当一个slave从一个错误的master那里同步数据开始计算时间。直到slave被纠正为向正确的master那里同步数据时。
#3.当想要取消一个正在进行的failover所需要的时间。
#4.当进行failover时，配置所有slaves指向新的master所需的最大时间。不过，即使过了这个超时，slaves依然会被正确配置为指向master，但是就不按parallel-syncs所配置的规则来了
# 默认三分钟
# sentinel failover-timeout <master-name> <milliseconds>
sentinel failover-timeout mymaster 180000

# SCRIPTS EXECUTION

#配置当某一事件发生时所需要执行的脚本，可以通过脚本来通知管理员，例如当系统运行不正常时发邮件通知相关人员。
#对于脚本的运行结果有以下规则：
#若脚本执行后返回1，那么该脚本稍后将会被再次执行，重复次数目前默认为10
#若脚本执行后返回2，或者比2更高的一个返回值，脚本将不会重复执行。
#如果脚本在执行过程中由于收到系统中断信号被终止了，则同返回值为1时的行为相同。
#一个脚本的最大执行时间为60s，如果超过这个时间，脚本将会被一个SIGKILL信号终止，之后重新执行。

#通知型脚本:当sentinel有任何警告级别的事件发生时（比如说redis实例的主观失效和客观失效等等），将会去调用这个脚本，这时这个脚本应该通过邮件，SMS等方式去通知系统管理员关于系统不正常运行的信息。调用该脚本时，将传给脚本两个参数，一个是事件的类型，一个是事件的描述。
#如果sentinel.conf配置文件中配置了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则sentinel无法正常启动成功。
#通知脚本
# sentinel notification-script <master-name> <script-path>
```

```
sentinel notification-script mymaster /var/redis/notify.sh

# 客户端重新配置主节点参数脚本
# 当一个master由于failover而发生改变时，这个脚本将会被调用，通知相关的客户端关于master地址已经发生改变的信息。
# 以下参数将会在调用脚本时传给脚本：
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
# 目前<state>总是“failover”,
# <role>是“leader”或者“observer”中的一个。
# 参数 from-ip, from-port, to-ip, to-port是用来和旧的master和新的master(即旧的slave)通信的
# 这个脚本应该是通用的，能被多次调用，不是针对性的。
# sentinel client-reconfig-script <master-name> <script-path>
 sentinel client-reconfig-script mymaster /var/redis/reconfig.sh
```

- 通过`redis-sentinel`启动哨兵服务

```
./redis-sentinel sentinel.conf
```

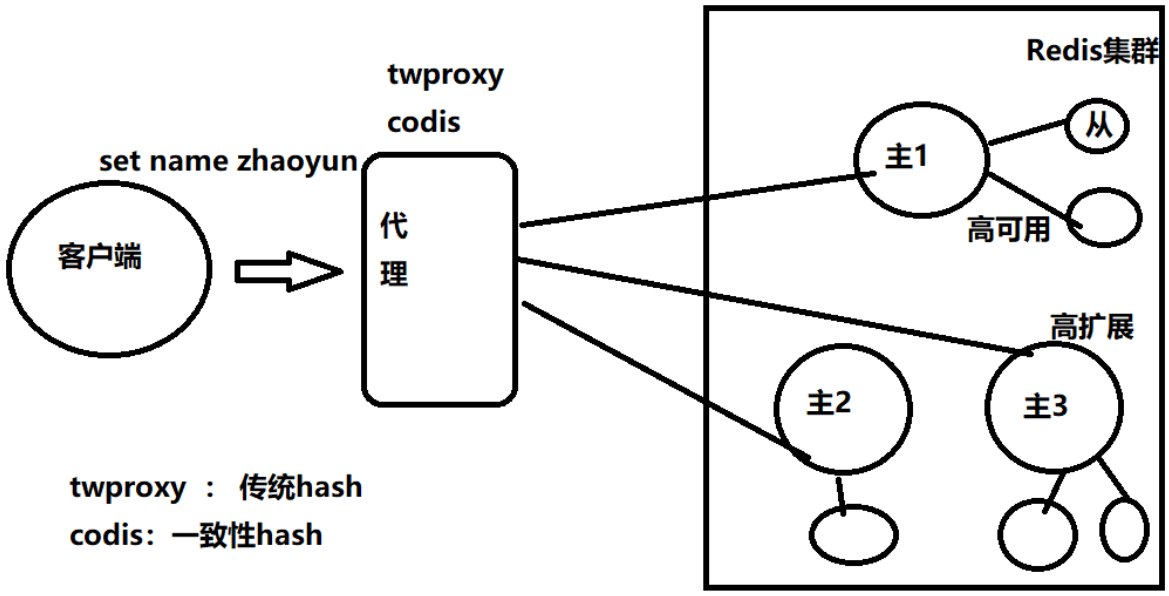
Redis集群

Redis的集群策略

twproxy

codis（豌豆荚）

代理方案



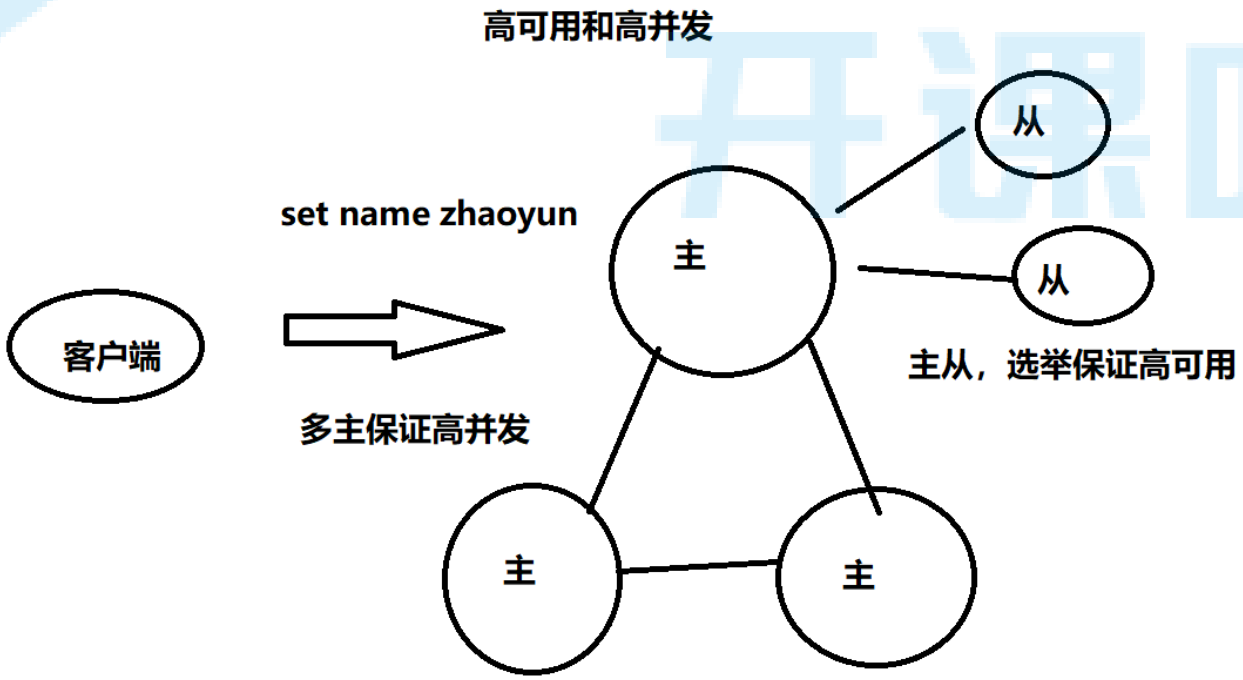
缺点：健壮性相对差，性能相对差

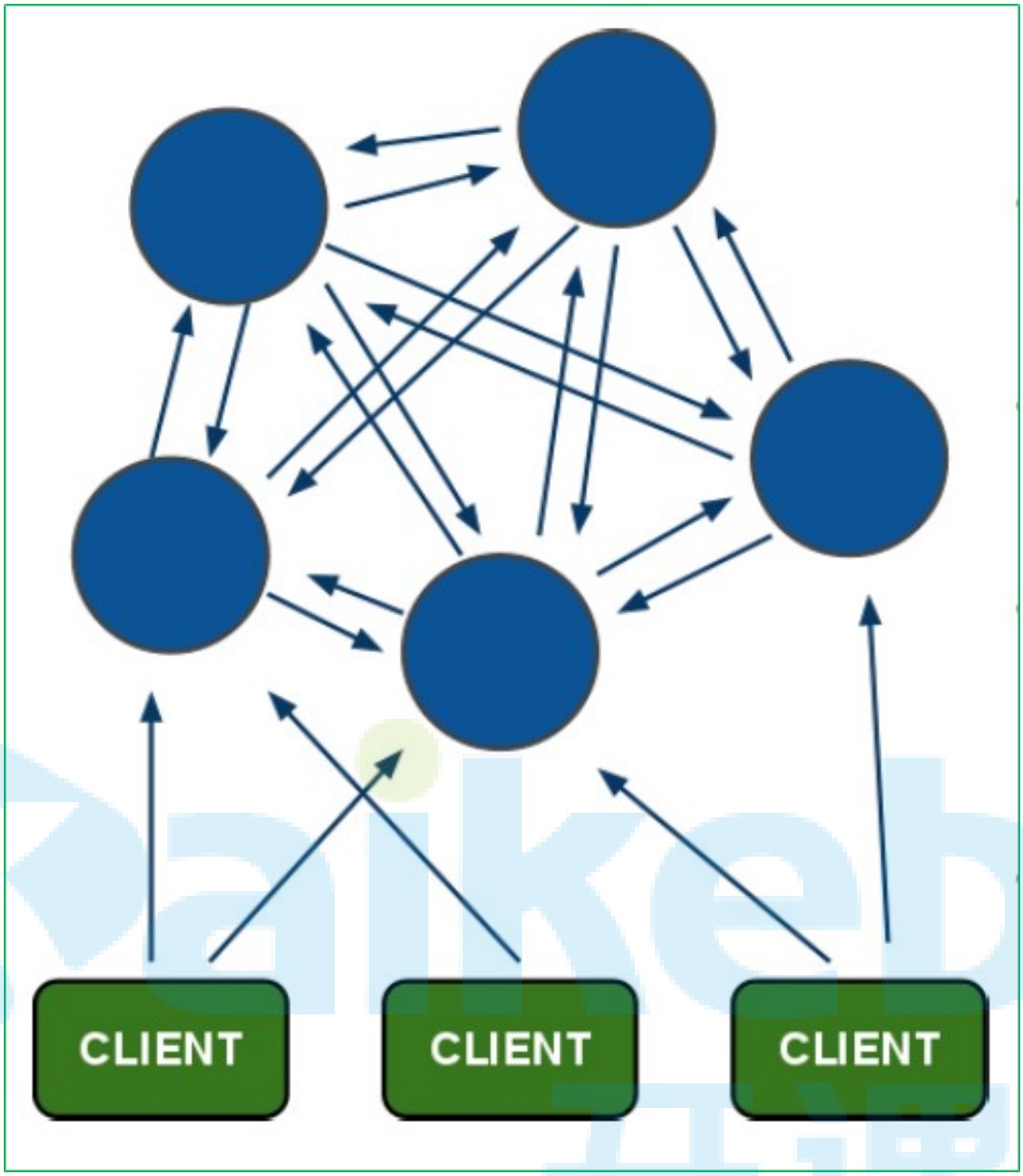
Redis-cluster架构图

Redis3以后，官方的集群方案 Redis-Cluster

Redis3 使用lua脚本实现

Redis5 直接实现





架构细节:

(1)所有的redis主节点彼此互联(**PING-PONG机制**),内部使用二进制协议优化传输速度和带宽.

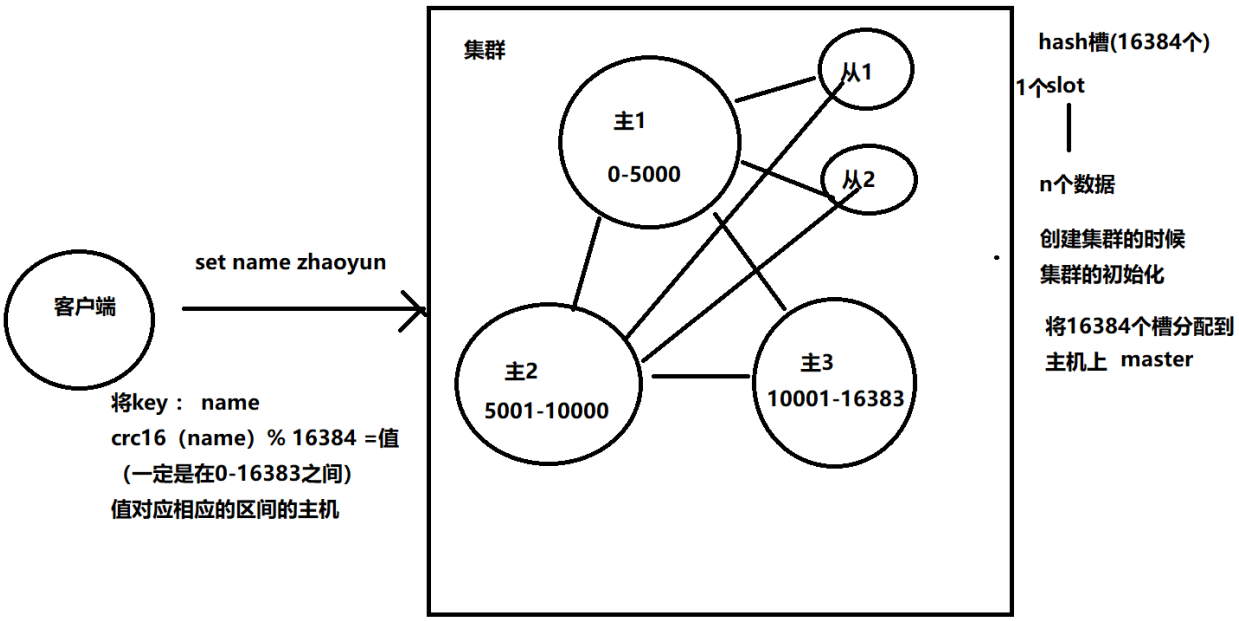
(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

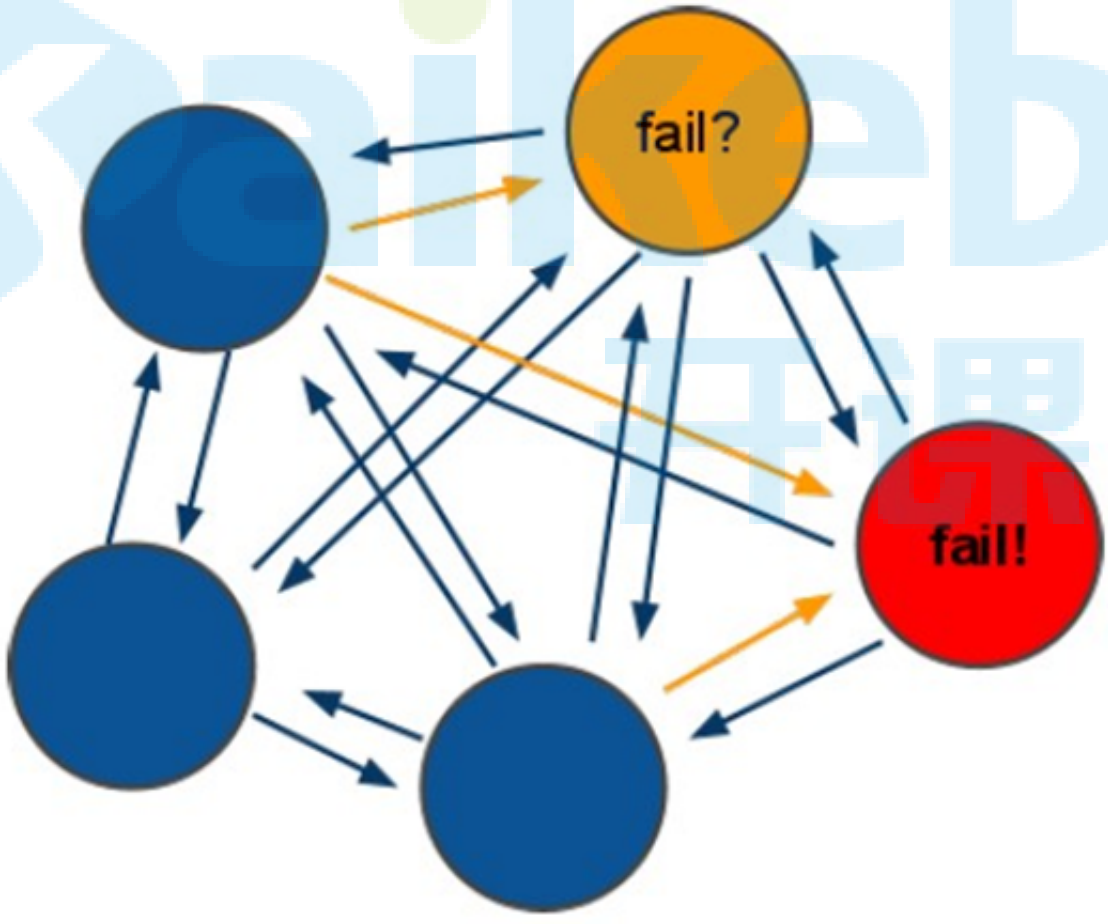
(4)redis-cluster把所有的物理节点映射到[0-16383]**slot**上,cluster 负责维护**node<->slot<->value**

```
Redis 集群中内置了 16384个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点
```

示例如下:



Redis-cluster投票:容错



1、主节点投票，如果超过半数的主都认为某主down了，则该主就down了 （主选择单数）

2、主节点投票，选出挂了的主的从升级为主

注：

集群挂了的情况：

1、半数的主挂了，不能投票生效，则集群挂了

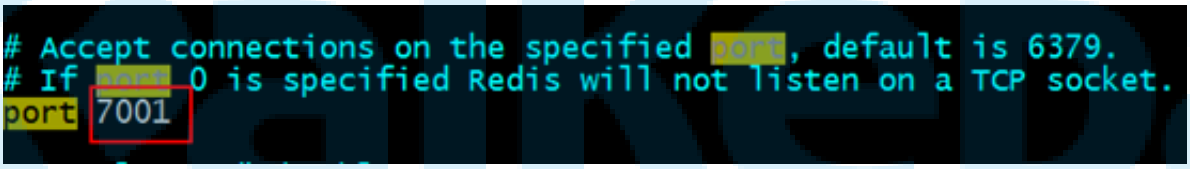
2、挂了的主机的从也挂了，造成slot槽分配不连续（16384不能完全分配），集群就挂了

安装RedisCluster

> Redis集群最少需要三台主服务器，三台从服务器。
>
> 端口号分别为：7001~7006

- 第一步：创建7001实例，并编辑redis.conf文件，修改port为7001。

  注意：创建实例，即拷贝单机版安装时，生成的bin目录，为7001目录。



- 第二步：修改redis.conf配置文件，打开cluster-enable yes

- 第三步：复制7001，创建7002~7006实例，**注意端口修改**。
- 第四步：创建start.sh，启动所有的实例

```
cd 7001
./redis-server redis.conf
cd ..
```

```
cd 7002
./redis-server redis.conf
cd ..

cd 7003
./redis-server redis.conf
cd ..

cd 7004
./redis-server redis.conf
cd ..

cd 7005
./redis-server redis.conf
cd ..

cd 7006
./redis-server redis.conf
cd ..
```

chomd u+x start.sh

- 第五步：创建Redis集群

```
[root@localhost 7001]# ./redis-cli --cluster create 127.0.0.1:7001
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006
--cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:7005 to 127.0.0.1:7001
Adding replica 127.0.0.1:7006 to 127.0.0.1:7002
Adding replica 127.0.0.1:7004 to 127.0.0.1:7003
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   replicates d277cd2984639747a17ca79428602480b28ef070
```

```
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
....
>>> Performing Cluster Check (using node 127.0.0.1:7001)
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
[root@localhost-0723 redis]#
```

命令客户端连接集群

命令：

```
./redis-cli  -h 127.0.0.1 -p 7001 -c
```

注意：**-c** 表示是以redis集群方式进行连接

```
[root@localhost redis-cluster]# cd 7001
[root@localhost 7001]# ./redis-cli  -h 127.0.0.1 -p 7001 -c
127.0.0.1:7001> set name1 aaa
-> Redirected to slot [12933] located at 127.0.0.1:7003
OK
127.0.0.1:7003>
```

查看集群的命令

- 查看集群状态

```
127.0.0.1:7003> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:3
cluster_stats_messages_sent:926
cluster_stats_messages_received:926
```

- 查看集群中的节点：

```
127.0.0.1:7003> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 myself,master - 0
1570457306000 3 connected 10923-16383
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 master - 0
1570457307597 1 connected 0-5460
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570457308605 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570457309614 2 connected 5461-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570457307000 4 connected
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570457309000 6 connected
127.0.0.1:7003>
```

维护节点

集群创建成功后可以继续向集群中添加节点

添加主节点

- 先创建7007节点
- 添加7007结点作为新节点,并启动

  执行命令：

```
[root@localhost 7007]# ./redis-cli --cluster add-node 127.0.0.1:7007
127.0.0.1:7001
>>> Adding node 127.0.0.1:7007 to cluster 127.0.0.1:7001
>>> Performing Cluster Check (using node 127.0.0.1:7001)
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
```

```
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 127.0.0.1:7007 to make it join the cluster.
[OK] New node added correctly.
```

- 查看集群结点发现7007已添加到集群中

```
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570457568602 3 connected 10923-16383
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570457567000 0 connected
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570457569609 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570457566000 2 connected 5461-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570457567000 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570457567000 1 connected 0-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570457567593 6 connected
```

hash槽重新分配（数据迁移）

添加完主节点需要对主节点进行hash槽分配，这样该主节才可以存储数据。

- **查看集群中槽占用情况**

```
cluster nodes
```

redis集群有16384个槽，集群中的每个结点分配自已槽，通过查看集群结点可以看到槽占用情况。

```
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570457568602 3 connected 10923-16383
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570457567000 0 connected
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570457569609 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570457566000 2 connected 5461-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570457567000 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570457567000 1 connected 0-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570457567593 6 connected
```

给刚添加的7007结点分配槽

- **第一步：连接上集群（连接集群中任意一个可用结点都行）**

```
[root@localhost 7007]# ./redis-cli --cluster reshard 127.0.0.1:7007
>>> Performing Cluster Check (using node 127.0.0.1:7007)
M: 50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007
   slots: (0 slots) master
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

- **第二步：输入要分配的槽数量**

```
How many slots do you want to move (from 1 to 16384)? 3000
```

输入：**3000**，表示要给目标节点分配3000个槽

- **第三步：输入接收槽的结点id**

```
What is the receiving node ID?
```

输入：50b073163bc4058e89d285dc5dfc42a0d1a222f2

```
PS：这里准备给7007分配槽，通过cluster nodes查看7007结点id为：

50b073163bc4058e89d285dc5dfc42a0d1a222f2
```

- **第四步：输入源结点id**

```
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
```

输入：**all**

- **第五步：输入yes开始移动槽到目标结点id**

```
Do you want to proceed with the proposed reshard plan (yes/no)?
```

输入：**yes**

```
Moving slot 11899 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11900 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11901 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11902 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11903 from 127.0.0.1:7003 to 127.0.0.1:7007:
```

```
Moving slot 11904 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11905 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11906 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11907 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11908 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11909 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11910 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11911 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11912 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11913 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11914 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11915 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11916 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11917 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11918 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11919 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11920 from 127.0.0.1:7003 to 127.0.0.1:7007:
Moving slot 11921 from 127.0.0.1:7003 to 127.0.0.1:7007:
```

查看结果

```
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570458285557 3 connected 11922-16383
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570458284000 7 connected 0-998 5461-6461 10923-11921
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570458283000 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570458284546 2 connected 6462-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570458283538 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570458283000 1 connected 999-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570458284000 6 connected
```

添加从节点

- **添加7008从结点，将7008作为7007的从结点**

命令：

```
./redis-cli --cluster add-node 新节点的ip和端口    旧节点ip和端口 --cluster-slave --cluster-master-id 主节点id
```

例如：

```
./redis-cli --cluster add-node 127.0.0.1:7008 127.0.0.1:7007 --cluster-slave --cluster-master-id 50b073163bc4058e89d285dc5dfc42a0d1a222f2
```

50b073163bc4058e89d285dc5dfc42a0d1a222f2是7007结点的id，可通过cluster nodes查看。

```
[root@localhost 7008]# ./redis-cli --cluster add-node 127.0.0.1:7008 127.0.0.1:7007 --cluster-slave --cluster-master-id 50b073163bc4058e89d285dc5dfc42a0d1a222f2
>>> Adding node 127.0.0.1:7008 to cluster 127.0.0.1:7007
>>> Performing Cluster Check (using node 127.0.0.1:7007)
M: 50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007
   slots:[0-998],[5461-6461],[10923-11921] (2999 slots) master
S: 51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004
   slots: (0 slots) slave
   replicates af559fc6c82c83dc39d07e2dfe59046d16b6a429
S: 78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006
   slots: (0 slots) slave
   replicates d277cd2984639747a17ca79428602480b28ef070
S: e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005
   slots: (0 slots) slave
   replicates 068b678923ad0858002e906040b0fef6fff8dda4
M: af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001
   slots:[999-5460] (4462 slots) master
   1 additional replica(s)
M: 068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002
   slots:[6462-10922] (4461 slots) master
   1 additional replica(s)
M: d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003
   slots:[11922-16383] (4462 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 127.0.0.1:7008 to make it join the cluster.
Waiting for the cluster to join
.....
>>> Configure node as replica of 127.0.0.1:7007.
[OK] New node added correctly.
```

注意：如果原来该结点在集群中的配置信息已经生成到cluster-config-file指定的配置文件中（如果cluster-config-file没有指定则默认为**nodes.conf**），这时可能会报错：

```
[ERR] Node XXXXXX is not empty. Either the node already knows other nodes
(check with CLUSTER NODES) or contains some key in database 0
```

解决方法是删除生成的配置文件nodes.conf，删除后再执行./redis-cli --cluster add-node 指令

- **查看集群中的结点，刚添加的7008为7007的从节点：**

```
[root@localhost 7008]# ./redis-cli  -h 127.0.0.1 -p 7001 -c
127.0.0.1:7001> cluster nodes
d277cd2984639747a17ca79428602480b28ef070 127.0.0.1:7003@17003 master - 0
1570458650720 3 connected 11922-16383
c3272565847bf9be8ae0194f7fb833db40b98ac4 127.0.0.1:7008@17008 slave
50b073163bc4058e89d285dc5dfc42a0d1a222f2 0 1570458648710 7 connected
50b073163bc4058e89d285dc5dfc42a0d1a222f2 127.0.0.1:7007@17007 master - 0
1570458649000 7 connected 0-998 5461-6461 10923-11921
e7b1f1962de2a1ffef2bf1ac5d94574b2e4d67d8 127.0.0.1:7005@17005 slave
068b678923ad0858002e906040b0fef6fff8dda4 0 1570458650000 5 connected
068b678923ad0858002e906040b0fef6fff8dda4 127.0.0.1:7002@17002 master - 0
1570458649715 2 connected 6462-10922
51c3ebdd0911dd6564040c7e20b9ae69cabb0425 127.0.0.1:7004@17004 slave
af559fc6c82c83dc39d07e2dfe59046d16b6a429 0 1570458648000 4 connected
af559fc6c82c83dc39d07e2dfe59046d16b6a429 127.0.0.1:7001@17001 myself,master - 0
1570458650000 1 connected 999-5460
78dfe773eaa817fb69a405a3863f5b8fcf3e172f 127.0.0.1:7006@17006 slave
d277cd2984639747a17ca79428602480b28ef070 0 1570458651725 6 connected
127.0.0.1:7001>
```

删除结点

命令：

```
./redis-cli --cluster del-node 127.0.0.1:7008
41592e62b83a8455f07f7797f1d5c071cffedb50
```

删除已经占有hash槽的结点会失败，报错如下：

```
[ERR] Node 127.0.0.1:7008 is not empty! Reshard data away and try again.
```

需要将该结点占用的hash槽分配出去（参考hash槽重新分配章节）。

Jedis连接集群

需要开启防火墙，或者直接关闭防火墙。

```
service iptables stop
```

代码实现

创建JedisCluster类连接redis集群。

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

使用spring

Ø 配置applicationContext.xml

```
<!-- 连接池配置 -->
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
  <!-- 最大连接数 -->
  <property name="maxTotal" value="30" />
  <!-- 最大空闲连接数 -->
  <property name="maxIdle" value="10" />
  <!-- 每次释放连接的最大数目 -->
  <property name="numTestsPerEvictionRun" value="1024" />
  <!-- 释放连接的扫描间隔（毫秒） -->
  <property name="timeBetweenEvictionRunsMillis" value="30000" />
  <!-- 连接最小空闲时间 -->
  <property name="minEvictableIdleTimeMillis" value="1800000" />
  <!-- 连接空闲多久后释放, 当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->
  <property name="softMinEvictableIdleTimeMillis" value="10000" />
  <!-- 获取连接时的最大等待毫秒数,小于零:阻塞不确定的时间,默认-1 -->
  <property name="maxWaitMillis" value="1500" />
  <!-- 在获取连接的时候检查有效性, 默认false -->
  <property name="testOnBorrow" value="true" />
  <!-- 在空闲时检查有效性, 默认false -->
  <property name="testWhileIdle" value="true" />
  <!-- 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true -->
  <property name="blockWhenExhausted" value="false" />
</bean>
<!-- redis集群 -->
<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">
  <constructor-arg index="0">
    <set>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7001"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7002"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7003"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7004"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
```

```
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7005"></constructor-arg>
      </bean>
      <bean class="redis.clients.jedis.HostAndPort">
        <constructor-arg index="0" value="192.168.101.3"></constructor-arg>
        <constructor-arg index="1" value="7006"></constructor-arg>
      </bean>
    </set>
  </constructor-arg>
  <constructor-arg index="1" ref="jedisPoolConfig"></constructor-arg>
</bean>
```

Ø 测试代码

```
private ApplicationContext applicationContext;
  @Before
  public void init() {
    applicationContext = new ClassPathXmlApplicationContext(
        "classpath:applicationContext.xml");
  }

  // redis集群
  @Test
  public void testJedisCluster() {
    JedisCluster jedisCluster = (JedisCluster) applicationContext
        .getBean("jedisCluster");

    jedisCluster.set("name", "zhangsan");
    String value = jedisCluster.get("name");
    System.out.println(value);
  }
```