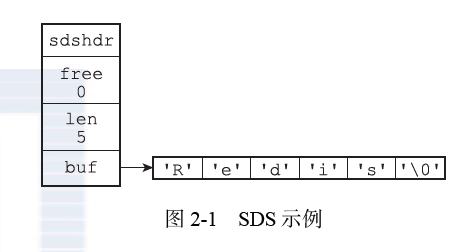
## 

## 一、Redis数据结构

### 简单动态字符串（simple dynamic string，SDS）

1. **struct** sdshdr {
2. // 记录buf 数组中已使用字节的数量
3. // 等于SDS 所保存字符串的长度
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. };



* free 属性的值为0，表示这个SDS 没有分配任何未使用空间。
* len 属性的值为5，表示这个SDS 保存了一个五字节长的字符串。
* buf属性是一个char类型的数组，数组的前五个字节分别保存了'R'、'e'、'd'、'i'、's' 五个字符，而最后一个字节则保存了空字符'\0'。

比起C 字符串，SDS 具有以下优点：

1）常数复杂度获取字符串长度。

2）杜绝缓冲区溢出。

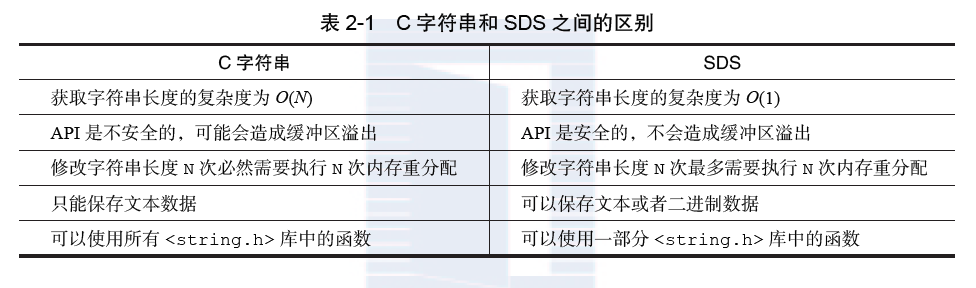
3）减少修改字符串长度时所需的内存重分配次数。

SDS可以空间预分配、惰性释放（不马上释放）

4）二进制安全。

二进制数据可能包含’\0’传统的C字符串存储不安全

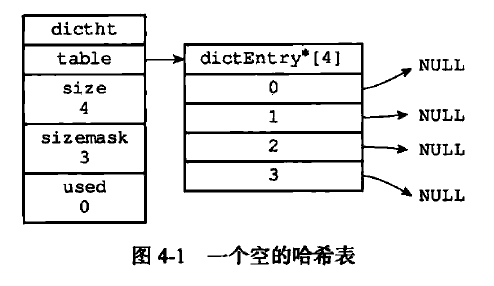
5）兼容部分C 字符串函数。



### 2、字典

Hash表

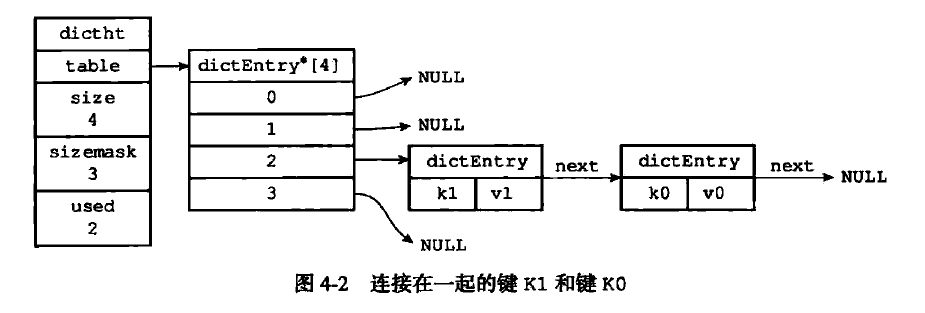
1. **typedef** **struct** dictht {
2. //哈希表数组
3. dictEntry \*\*table;
4. //哈希表大小
5. unsigned **long** size;
6. //哈希表大小掩码，用于计算索引值
7. //总是等于size - 1
8. unsigned **long** sizemask;
9. //该哈希表已有节点的数量
10. unsigned **long** used;
11. } dictht;



* table属性是一个数组，数组中的每个元素都是一个指向dict.h/dictEntry 结构的指针，每个dictEntry结构保存着一个键值对。
* size属性记录了哈希表的大小，也即是table数组的大小
* used属性则记录了哈希表目前已有节点（键值对）的数量。
* sizemask属性的值总是等于size-1，这个属性和哈希值一起决定一个键应该被放到table数组的哪个索引上面。

Hash节点

1. **typedef** **struct** dictEntry {
2. // 键
3. **void** \*key;
4. // 值
5. **union** {
6. **void** \*val;
7. uint64\_t u64;
8. int64\_t s64;
9. } v;
10. // 指向下个哈希表节点，形成链表
11. **struct** dictEntry \*next;
12. } dictEntry;



* key属性保存着键值对中的键
* v属性则保存着键值对中的值，其中键值对的值可以是一个指针，或者是一个uint64\_t整数，又或者是一个int64\_t整数。
* next属性是指向另一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对连接在一次，以此来解决键冲突（collision）的问题。

字典

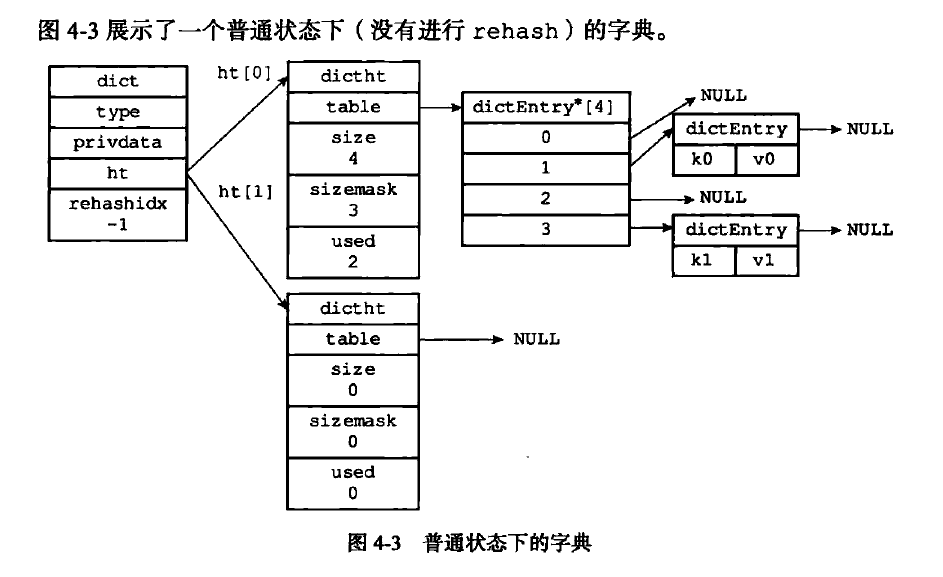
1. **typedef** **struct** dict {
2. // 类型特定函数
3. dictType \*type;
5. // 私有数据
6. **void** \*privdata;
8. // 哈希表
9. dictht ht[2];
11. // rehash 索引
12. // 当rehash 不在进行时，值为-1
13. **int** rehashidx; /\* rehashing not in progress if rehashidx == -1\*/
14. } dict;

type属性和privdata属性是针对不同类型的键值对，为创建多态字典而设置的：

* type属性是一个指向dictType结构的指针，每个dictType结构保存了一簇用于操作特定类型键值对的函数，Redis会为用途不同的字典设置不同的类型特定函数。
* privdata属性则保存了需要传给那些类型特定函数的可选参数。

1. **typedef** **struct** dictType {
2. // 计算哈希值的函数
3. unsigned **int**(\*hashFunction)(**const** **void** \*key);
4. // 复制键的函数
5. **void** \*(\*keyDup)(**void** \*privdata, **const** **void** \*key);
6. // 复制值的函数
7. **void** \*(\*valDup)(**void** \*privdata, **const** **void** \*obj);
8. // 对比键的函数
9. **int**(\*keyCompare)
10. (**void** \*privdata, **const** **void** \*key1, **const** **void** \*key2);
11. // 销毁键的函数
12. **void**(\*keyDestructor)(**void** \*privdata, **void** \*key);
13. // 销毁值的函数
14. **void**(\*valDestructor)(**void** \*privdata, **void** \*obj);
15. } dictType;

* ht属性是一个包含两个项的数组，数组中的每个项都是一个dictht哈希表，一般情况下，字典只使用ht[0]哈希表，ht[1]哈希表只会在对ht[0]哈希表进行rehash时使用。
* 另一个和rehash有关的属性就是rehashidx，它记录了rehash目前的进度，如果目前没有在进行rehash，那么它的值为-1。



Hash算法

1. //使用字典设置的哈希函数，计算键key 的哈希值
2. hash = dict->type->hashFunction(key);
3. //使用哈希表的sizemask 属性和哈希值，计算出索引值
4. //根据情况不同，ht[x] 可以是ht[0] 或者ht[1]
5. index = hash & dict->ht[x].sizemask;

举一个例子：

如果我们要将一个键值对k0和v0添加到字典里面，那么程序会先使用语句：

hash = dict->type->hashFunction(k0);

计算键k0的哈希值。假设计算得出的哈希值为8，那么程序会继续使用语句：

index = hash&dict->ht[0].sizemask = 8 & 3 = 0;（超过3位全是0，所以之和hash值最后几位相关）

计算出键k0的索引值0，这表示包含键值对k0和v0的节点应该被放置到哈希表数组的索引0位置上

#### Rehash：

随着操作的不断执行，哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表的负载因子（load factor）维持在一个合理的范围之内，当哈希表保存的键值对数量太多或者太少时，程序需要对哈希表的大小进行相应的扩展或者收缩。

扩展和收缩哈希表的工作可以通过执行rehash（重新散列）操作来完成，Redis对字典的哈希表执行rehash的步骤如下：

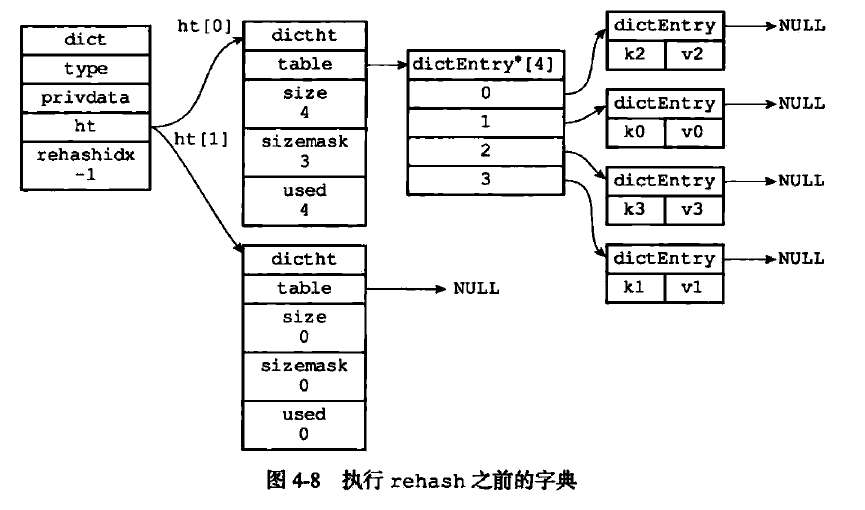
1）为字典的ht[1]哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及ht[0]当前包含的键值对数量（也即是ht[0].used属性的值）：

* 如果执行的是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used\*2的2^n（2的n次方幂）；
* 如果执行的是收缩操作，那么ht[1]的大小为第一个大于等于ht[0].used的2^n。

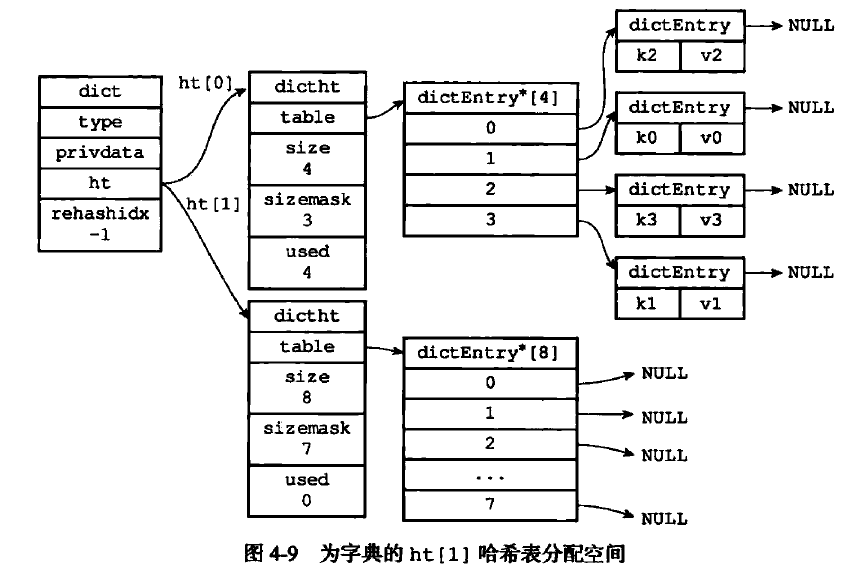
2）将保存在ht[0]中的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]哈希表的指定位置上。

3）当ht[0]包含的所有键值对都迁移到了ht[1]之后（ht[0]变为空表），释放ht[0]，将ht[1]设置为ht[0]，并在ht[1]新创建一个空白哈希表，为下一次rehash做准备。

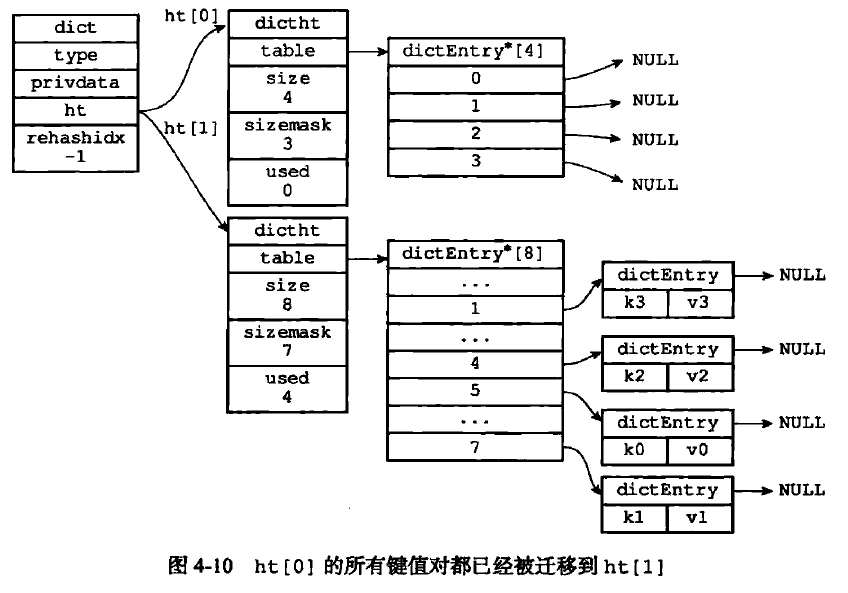
举一个栗子：



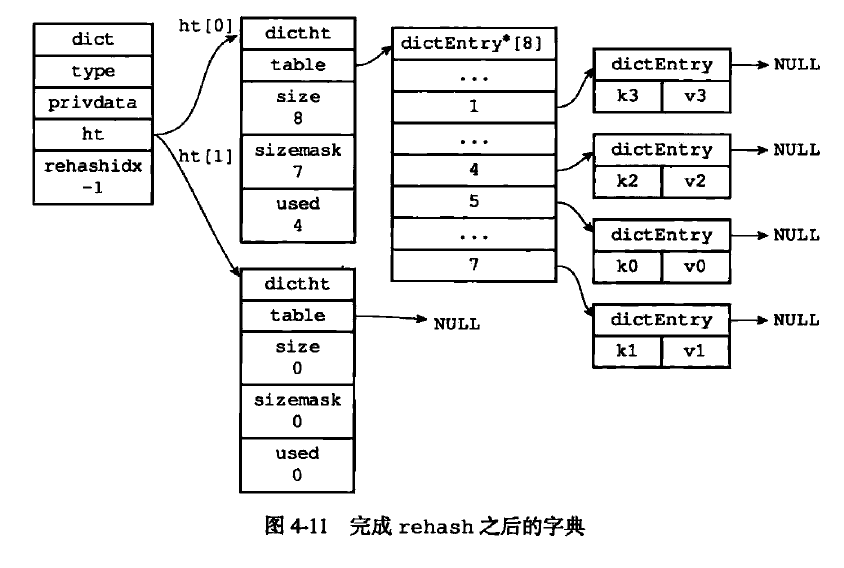
1）ht[0].used当前的值为4，4\*2=8，而8（2 3）恰好是第一个大于等于4的2的n次方，所以程序会将ht[1]哈希表的大小设置为8。图4-9展示了ht[1]在分配空间之后，字典的样子。



2）将ht[0]包含的四个键值对都rehash到ht[1]，如图4-10所示。



3）释放ht[0]，并将ht[1]设置为ht[0]，然后为ht[1]分配一个空白哈希表，如图4-11所示。至此，对哈希表的扩展操作执行完毕，程序成功将哈希表的大小从原来的4改为了现在的8。



扩展与收缩

负载因子：

1. //负载因子= 哈希表已保存节点数量/ 哈希表大小
2. load\_factor = ht[0].used / ht[0].size

1）服务器目前没有在执行BGSAVE命令或者BGREWRITEAOF命令，并且哈希表的负载

因子大于等于1。

2）服务器目前正在执行BGSAVE命令或者BGREWRITEAOF命令，并且哈希表的负载因

子大于等于5。

另一方面，当哈希表的负载因子小于0.1时，程序自动开始对哈希表执行收缩操作。

#### 渐进式rehash：

因此，为了避免rehash对服务器性能造成影响，服务器不是一次性将ht[0]里面的所有键值对全部rehash到ht[1]，而是分多次、渐进式地将ht[0]里面的键值对慢慢地rehash到ht[1]。以下是哈希表渐进式rehash的详细步骤：

1）为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表。

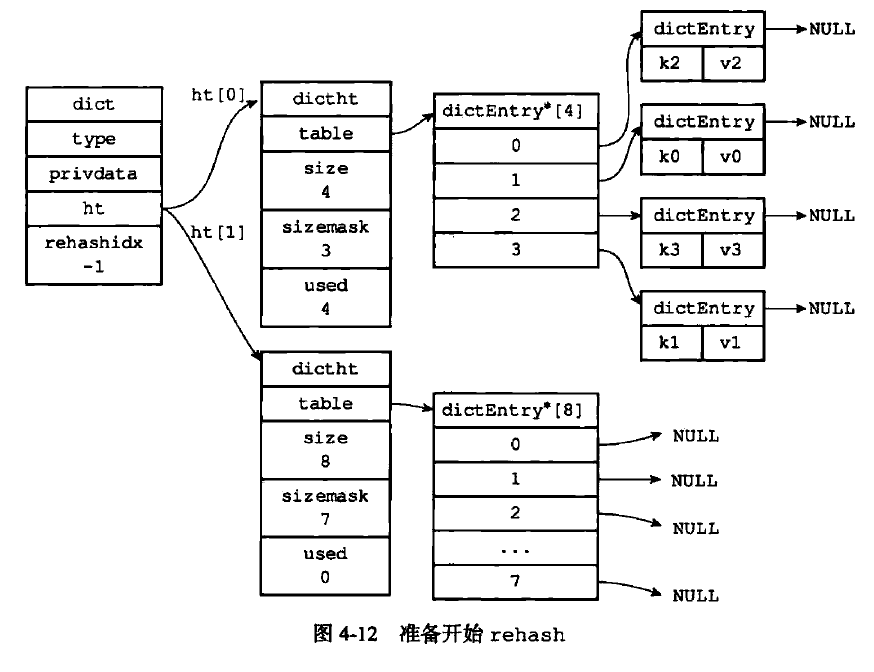
2）在字典中维持一个索引计数器变量rehashidx，并将它的值设置为0，表示rehash工作正式开始。

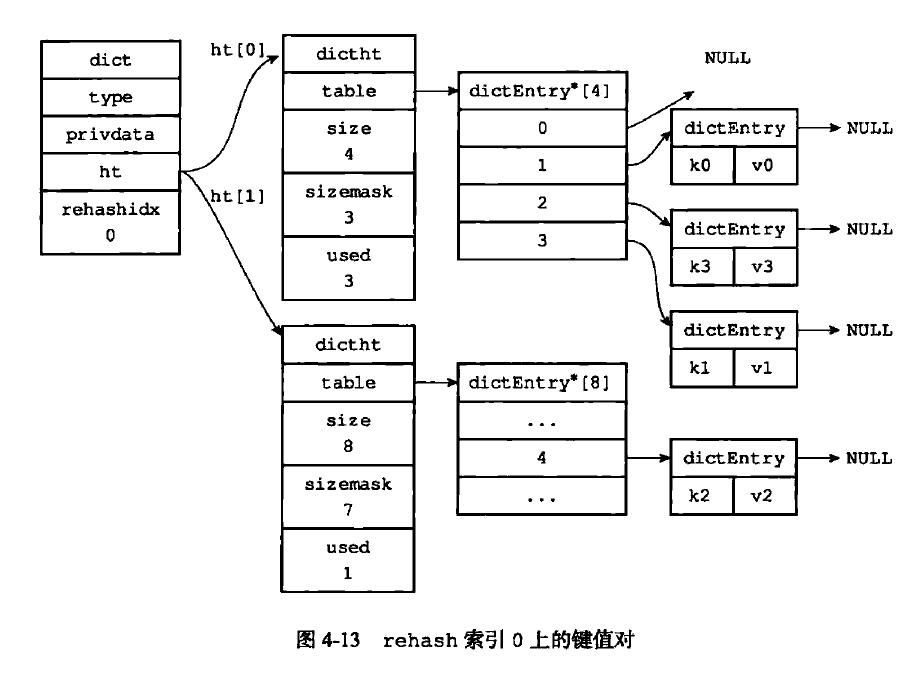
3）在rehash进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将ht[0]哈希表在rehashidx索引上的所有键值对rehash到ht[1]，当rehash工作完成之后，程序将rehashidx属性的值增一。

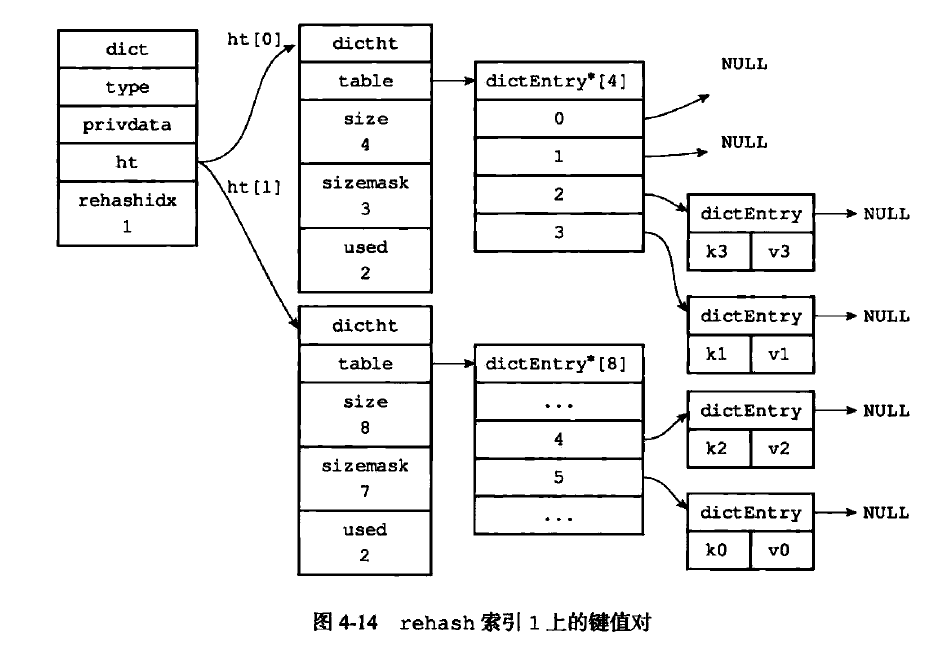
4）随着字典操作的不断执行，最终在某个时间点上，ht[0]的所有键值对都会被rehash至ht[1]，这时程序将rehashidx属性的值设为-1，表示rehash操作已完成。

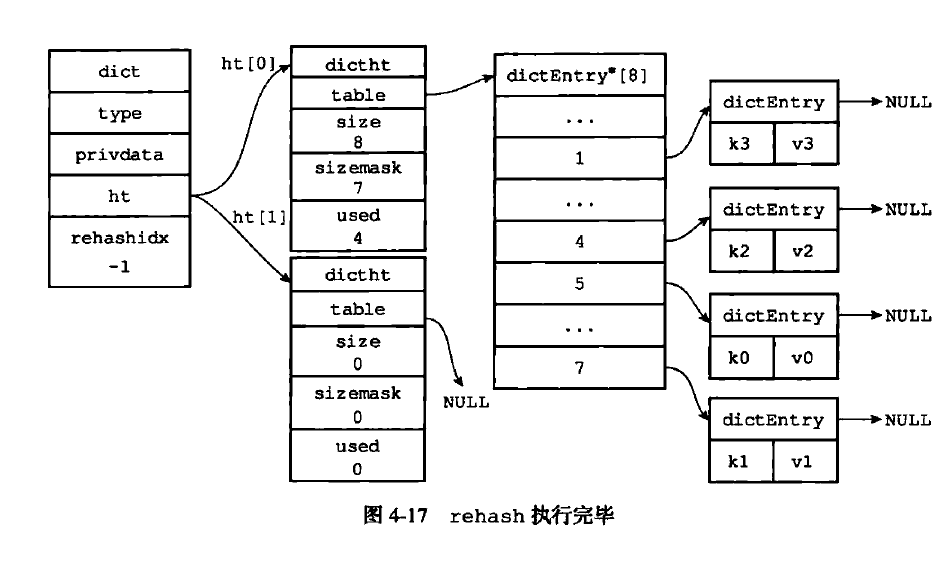
渐进式rehash的好处在于它采取分而治之的方式，将rehash键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式rehash而带来的庞大计算量。

举一个例子：









因为在进行渐进式rehash的过程中，字典会同时使用ht[0]和ht[1]两个哈希表，所以在渐进式rehash进行期间，字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行。例如，要在字典里面查找一个键的话，程序会先在ht[0]里面进行查找，如果没找到的话，就会继续到ht[1]里面进行查找，诸如此类。

另外，在渐进式rehash执行期间，新添加到字典的键值对一律会被保存到ht[1]里面，而ht[0]则不再进行任何添加操作，这一措施保证了ht[0]包含的键值对数量会只减不增，并随着rehash操作的执行而最终变成空表。

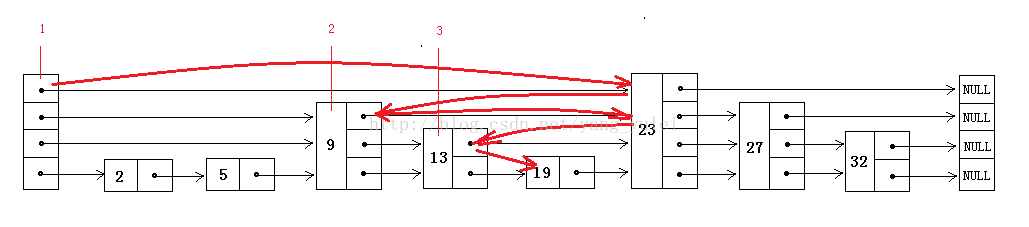
### 3、跳跃表

**为什么用跳表：**

* 期望上O(NlogN)
* 不用像AVL一样平衡调整

**跳表的查找：**

我们以查找19为例，图解查找过程。



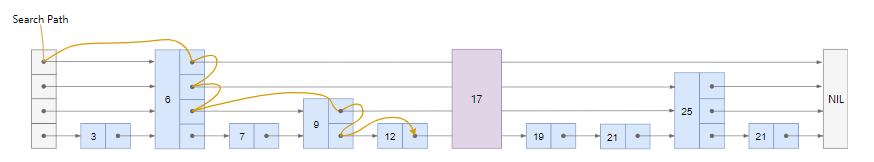
先从最上层的跳跃区间大的层开始查找。从头结点开始，首先和23进行比较，小于23，（此时查找指针在图中“1”位置处），查找指针到下一层继续查找。

然后和9进行判断，大于9，查找指针再往前走一步和23比较，小于23，（此时查找指针在图中“2”位置处） 此时这个值肯定在9结点和23结点之间。查找指针到下一层继续查找。

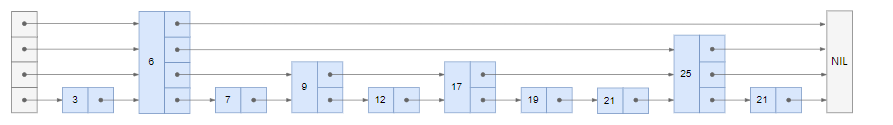
然后和13进行判断，大于13，查找指针再往前走一步和23比较，小于23，（此时查找指针在图中“3”位置处）此时这个值肯定在13结点和23结点之间。查找指针到下一层继续查找。此时，我们和19进行判断，找到了。

**跳表的构造：**

* 期望插入一个17节点



* 任硬币，第二次出现正面，所以层数为2
* 17在6~null之间，但是层数为4，不管，往下走
* 17在6到25之间，但是层数为3，不管，往下走
* 17不在6~9之间，往右走，来到9
* 17在9~25之间，且层数为2，建一个“中转点”，往下走
* 17不在9~12之间，往右走
* 17在12~19之间，且层数为1，<=2，建一个“中转点”
* 将所有“中转点”包括在节点17之中，完成



**Redis中的跳表：**

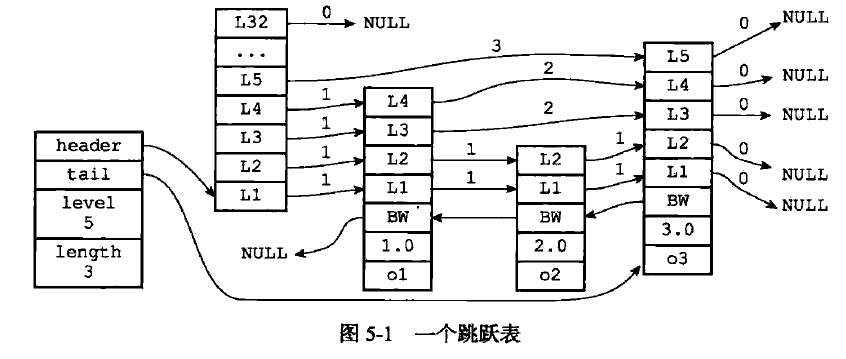


图5-1展示了一个跳跃表示例，位于图片最左边的是zskiplist结构，该结构包含以下属性：

* ·header：指向跳跃表的表头节点。
* ·tail：指向跳跃表的表尾节点。
* ·level：记录目前跳跃表内，层数最大的那个节点的层数（表头节点的层数不计算在内）。
* ·length：记录跳跃表的长度，跳跃表目前包含节点的数量（表头节点不计算在内）。

位于zskiplist结构右方的是四个zskiplistNode结构，该结构包含以下属性：

* ·层（level）：节点中用L1、L2、L3等字样标记节点的各个层，L1代表第一层，L2代表第二层，以此类推。每个层都带有两个属性：前进指针和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。在上面的图片中，连线上带有数字的箭头就代表前进指针，而那个数字就是跨度。当程序从表头向表尾进行遍历时，访问会沿着层的前进指针进行。
* ·后退（backward）指针：节点中用BW字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。
* ·分值（score）：各个节点中的1.0、2.0和3.0是节点所保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。
* ·成员对象（obj）：各个节点中的o1、o2和o3是节点所保存的成员对象（可以直接理解成value）。注意表头节点和其他节点的构造是一样的：表头节点也有后退指针、分值和成员对象，不过表头节点的这些属性都不会被用到，所以图中省略了这些部分，只显示了表头节点的各个层。

层

跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其他节点的指针，程序可以通过这些层来加快访问其他节点的速度，一般来说，层的数量越多，访问其他节点的速度就越快。

每次创建一个新跳跃表节点的时候，程序都根据幂次定律（power law，越大的数出现的概率越小）随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是层的“高度”。（扔硬币，经过N次出现正面，则高度为N）

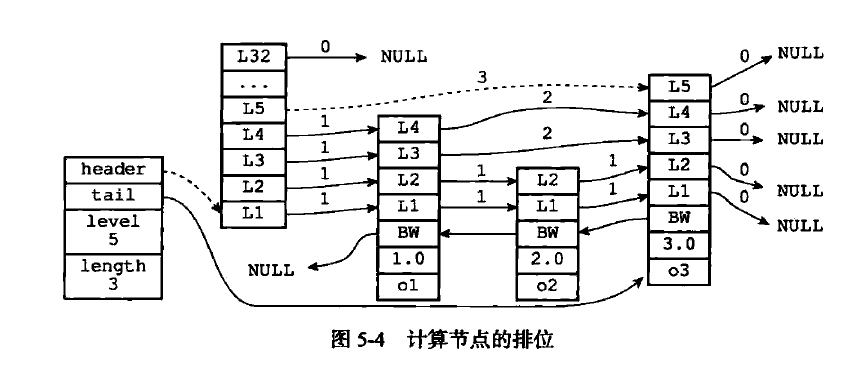
跨度

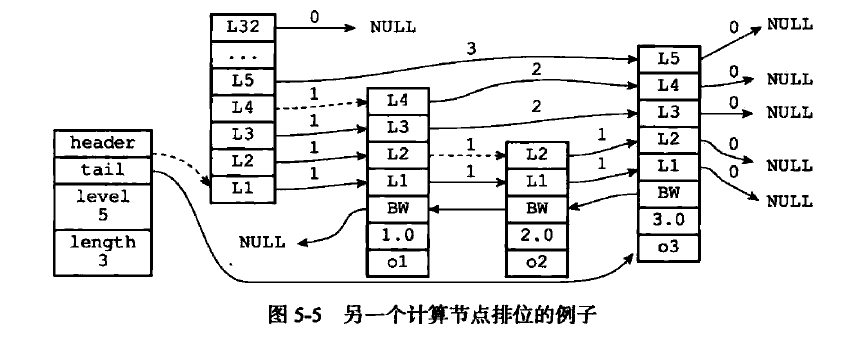
层的跨度（level[i].span属性）用于记录两个节点之间的距离：

* ·两个节点之间的跨度越大，它们相距得就越远。
* ·指向NULL的所有前进指针的跨度都为0，因为它们没有连向任何节点。

初看上去，很容易以为跨度和遍历操作有关，但实际上并不是这样，遍历操作只使用前进指针就可以完成了，跨度实际上是用来计算排位（rank）的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。

举个例子，图5-4用虚线标记了在跳跃表中查找分值为3.0、成员对象为o3的节点时，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为3，所以目标节点在跳跃表中的排位为3。

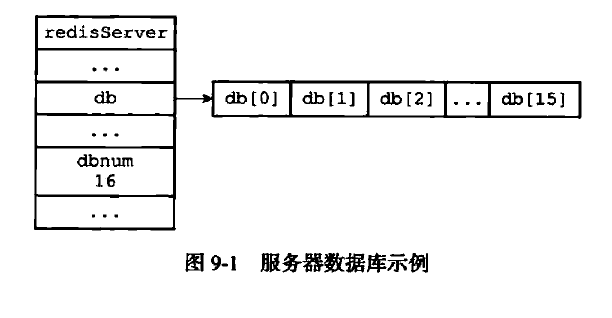




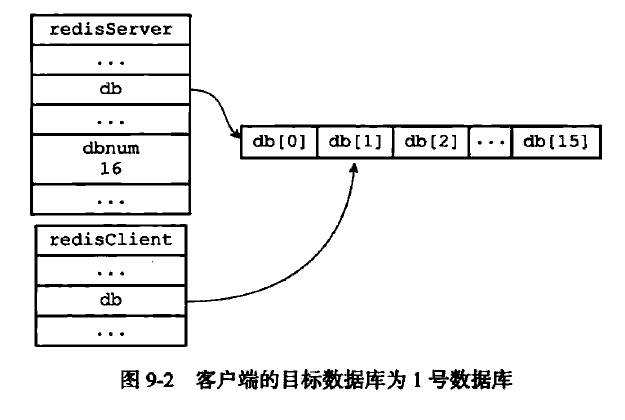
再举个例子，图5-5用虚线标记了在跳跃表中查找分值为2.0、成员对象为o2的节点时，沿途经历的层：在查找节点的过程中，程序经过了两个跨度为1的节点，因此可以计算出，目标节点在跳跃表中的排位为2。

## 二、单机数据库

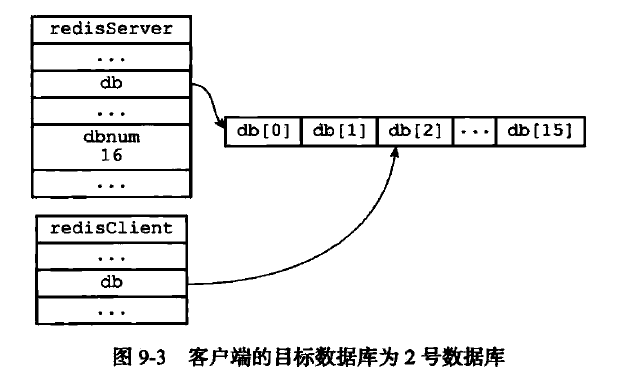
dbnum属性的值由服务器配置的database选项决定，默认情况下，该选项的值为16，所以Redis服务器默认会创建16个数据库



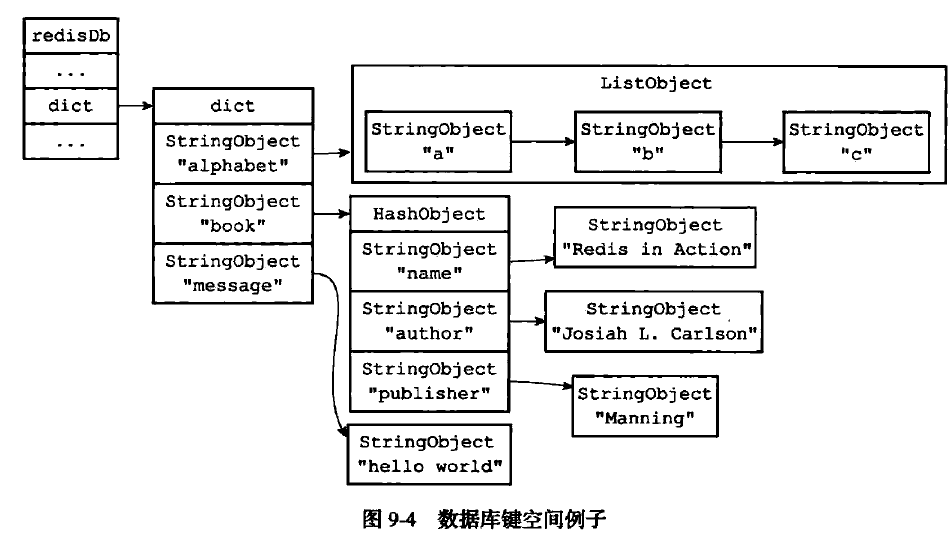
每个Redis客户端都有自己的目标数据库，每当客户端执行数据库写命令或者数据库读命令的时候，目标数据库就会成为这些命令的操作对象。默认情况下，Redis客户端的目标数据库为0号数据库，但客户端可以通过执行SELECT命令来切换目标数据库。



通过select 2切换数据库：



Redis服务器将所有数据库都保存在服务器状态redis.h/redisServer结构的db数组中，db数组的每个项都是一个redis.h/redisDb结构，每个redisDb结构代表一个数据库：



### 设定过期时间：

通过EXPIRE命令或者PEXPIRE命令，客户端可以以秒或者毫秒精度为数据库中的某个键设置生存时间（Time To Live，TTL）（因为redis是一个缓存系统，设定什么时候过期是必要的），在经过指定的秒数或者毫秒数之后，服务器就会自动删除生存时间为0的键：

redis> SET key value

OK

redis> EXPIRE key 5

(integer) 1

redis> GET key // 5 秒之内

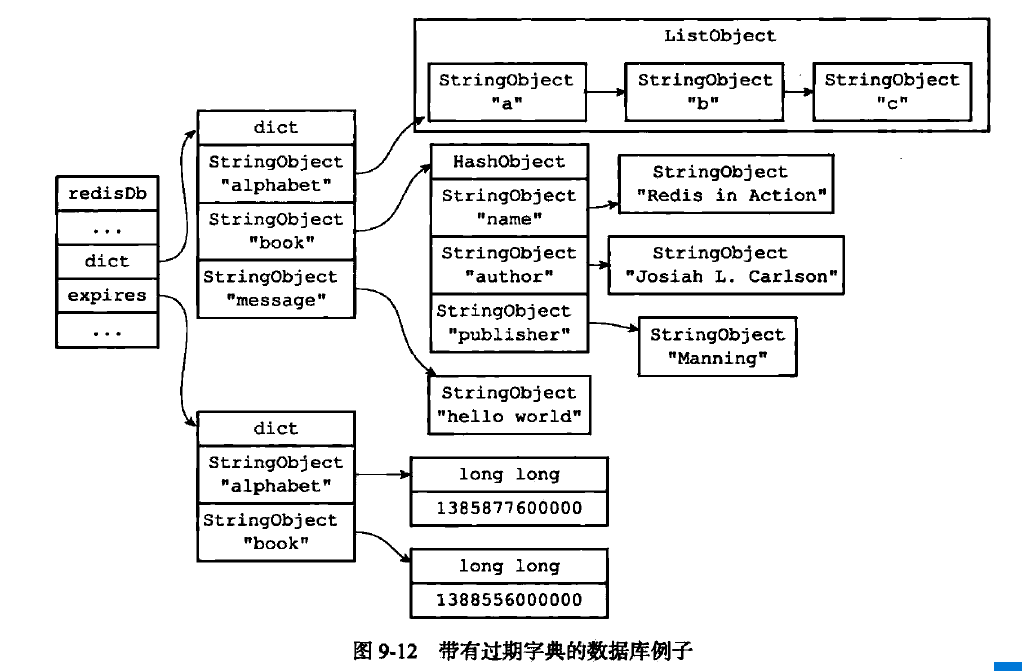
"value"

redis> GET key // 5 秒之后

(nil)

redisDb结构的expires字典保存了数据库中所有键的过期时间，我们称这个字典为过期字典：

* ·过期字典的键是一个指针，这个指针指向键空间中的某个键对象（也即是某个数据库键）。
* ·过期字典的值是一个long long类型的整数，这个整数保存了键所指向的数据库键的过期时间——一个毫秒精度的UNIX时间戳。



* ·第一个键值对的键为alphabet键对象，值为1385877600000，这表示数据库键alphabet的过期时间为1385877600000（2013年12月1日零时）。
* ·第二个键值对的键为book键对象，值为1388556000000，这表示数据库键book的过期时间为1388556000000（2014年1月1日零时）。

### 过期键删除策略

经过上一节的介绍，我们知道了数据库键的过期时间都保存在过期字典中，又知道了如何根据过期时间去判断一个键是否过期，现在剩下的问题是：如果一个键过期了，那么它什么时候会被删除呢？

这个问题有三种可能的答案，它们分别代表了三种不同的删除策略：

* ·定时删除：在设置键的过期时间的同时，创建一个定时器（timer），让定时器在键的过期时间来临时，立即执行对键的删除操作。
* ·惰性删除：放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键；如果没有过期，就返回该键。
* ·定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

在这三种策略中，第一种和第三种为主动删除策略，而第二种则为被动删除策略。

##### 定时删除

定时删除策略对内存是最友好的：通过使用定时器，定时删除策略可以保证过期键会尽可能快地被删除，并释放过期键所占用的内存。

另一方面，定时删除策略的缺点是，它对CPU时间是最不友好的：在过期键比较多的情况下，删除过期键这一行为可能会占用相当一部分CPU时间，在内存不紧张但是CPU时间非常紧张的情况下，将CPU时间用在删除和当前任务无关的过期键上，无疑会对服务器的响应时间和吞吐量造成影响。

##### 惰性删除

惰性删除策略对CPU时间来说是最友好的：程序只会在取出键时才对键进行过期检查，这可以保证删除过期键的操作只会在非做不可的情况下进行，并且删除的目标仅限于当前处理的键，这个策略不会在删除其他无关的过期键上花费任何CPU时间。

惰性删除策略的缺点是，它对内存是最不友好的：如果一个键已经过期，而这个键又仍然保留在数据库中，那么只要这个过期键不被删除，它所占用的内存就不会释放。

在使用惰性删除策略时，如果数据库中有非常多的过期键，而这些过期键又恰好没有被访问到的话，那么它们也许永远也不会被删除（除非用户手动执行FLUSHDB），我们甚至可以将这种情况看作是一种内存泄漏——无用的垃圾数据占用了大量的内存，而服务器却不会自己去释放它们，这对于运行状态非常依赖于内存的Redis服务器来说，肯定不是一个好消息。

##### 定期删除

从上面对定时删除和惰性删除的讨论来看，这两种删除方式在单一使用时都有明显的缺陷：

* ·定时删除占用太多CPU时间，影响服务器的响应时间和吞吐量。
* ·惰性删除浪费太多内存，有内存泄漏的危险。

定期删除策略是前两种策略的一种整合和折中：

* ·定期删除策略每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率来减少删除操作对CPU时间的影响。
* ·除此之外，通过定期删除过期键，定期删除策略有效地减少了因为过期键而带来的内存浪费。

定期删除策略的难点是确定删除操作执行的时长和频率：

* ·如果删除操作执行得太频繁，或者执行的时间太长，定期删除策略就会退化成定时删除策略，以至于将CPU时间过多地消耗在删除过期键上面。
* ·如果删除操作执行得太少，或者执行的时间太短，定期删除策略又会和惰性删除策略一样，出现浪费内存的情况。

因此，如果采用定期删除策略的话，服务器必须根据情况，合理地设置删除操作的执行时长和执行频率。

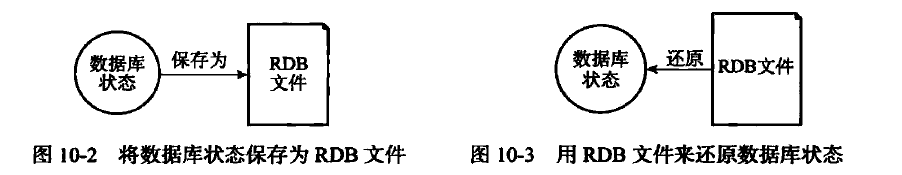
Redis服务器实际使用的是惰性删除和定期删除两种策略：通过配合使用这两种删除策略，服务器可以很好地在合理使用CPU时间和避免浪费内存空间之间取得平衡。

## 三、RDB持久化与AOF持久化

RDB（RedisDB）：本质上就是一个内存数据的快照

AOF（Append Only Log）：本质上记录日志，通过回放实现数据恢复

### RDB



* SAVE命令执行时的服务器状态

当SAVE命令执行时，Redis服务器会被阻塞，所以当SAVE命令正在执行时，客户端发送的所有命令请求都会被拒绝。

只有在服务器执行完SAVE命令、重新开始接受命令请求之后，客户端发送的命令才会被处理。

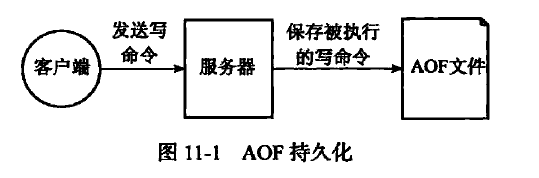
* BGSAVE命令执行时的服务器状态

因为BGSAVE命令的保存工作是由子进程执行的，所以在子进程创建RDB文件的过程中，Redis服务器仍然可以继续处理客户端的命令请求，但是，在BGSAVE命令执行期间，服务器处理SAVE、BGSAVE、BGREWRITEAOF三个命令的方式会和平时有所不同。

A、首先，在BGSAVE命令执行期间，客户端发送的SAVE命令会被服务器拒绝，服务器禁止SAVE命令和BGSAVE命令同时执行是为了避免父进程（服务器进程）和子进程同时执行两个rdbSave调用，防止产生竞争条件。

B、其次，在BGSAVE命令执行期间，客户端发送的BGSAVE命令会被服务器拒绝，因为同时执行两个BGSAVE命令也会产生竞争条件。

### AOF：



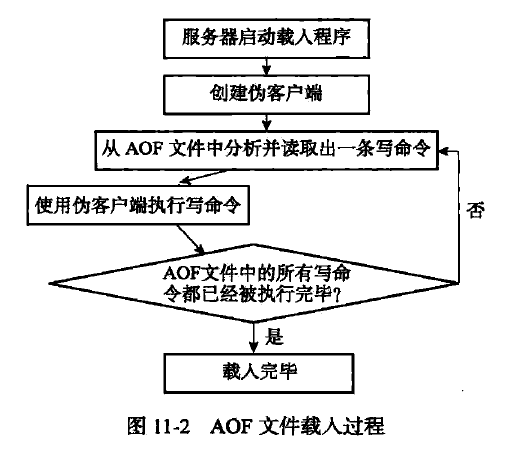
Redis读取AOF文件并还原数据库状态的详细步骤如下：

1）创建一个不带网络连接的伪客户端（fake client）：因为Redis的命令只能在客户端上下文中执行，而载入AOF文件时所使用的命令直接来源于AOF文件而不是网络连接，所以服务器使用了一个没有网络连接的伪客户端来执行AOF文件保存的写命令，伪客户端执行命令的效果和带网络连接的客户端执行命令的效果完全一样。

2）从AOF文件中分析并读取出一条写命令。

3）使用伪客户端执行被读出的写命令。

4）一直执行步骤2和步骤3，直到AOF文件中的所有写命令都被处理完毕为止。当完成以上步骤之后，AOF文件所保存的数据库状态就会被完整地还原出来



#### AOF重写

A、为什么要重写？

因为AOF持久化是通过保存被执行的写命令来记录数据库状态的，所以随着服务器运行时间的流逝，AOF文件中的内容会越来越多，文件的体积也会越来越大，如果不加以控制的话，体积过大的AOF文件很可能对Redis服务器、甚至整个宿主计算机造成影响，并且AOF文件的体积越大，使用AOF文件来进行数据还原所需的时间就越多。

为了解决AOF文件体积膨胀的问题，Redis提供了AOF文件重写（rewrite）功能。通过该功能，Redis服务器可以创建一个新的AOF文件来替代现有的AOF文件，新旧两个AOF文件所保存的数据库状态相同，但新AOF文件不会包含任何浪费空间的冗余命令，所以新AOF文件的体积通常会比旧AOF文件的体积要小得多。

B、怎样重写

AOF文件重写并不需要对现有的AOF文件进行任何读取、分析或者写入操作，这个功能是通过读取服务器当前的数据库状态来实现的。

redis> RPUSH list "A" "B" // ["A", "B"]

(integer) 2

redis> RPUSH list "C" // ["A", "B", "C"]

(integer) 3

redis> RPUSH list "D" "E" // ["A", "B", "C", "D", "E"]

(integer) 5

redis> LPOP list // ["B", "C", "D", "E"]

"A"

redis> LPOP list // ["C", "D", "E"]

"B"

redis> RPUSH list "F" "G" // ["C", "D", "E", "F", "G"]

(integer) 5

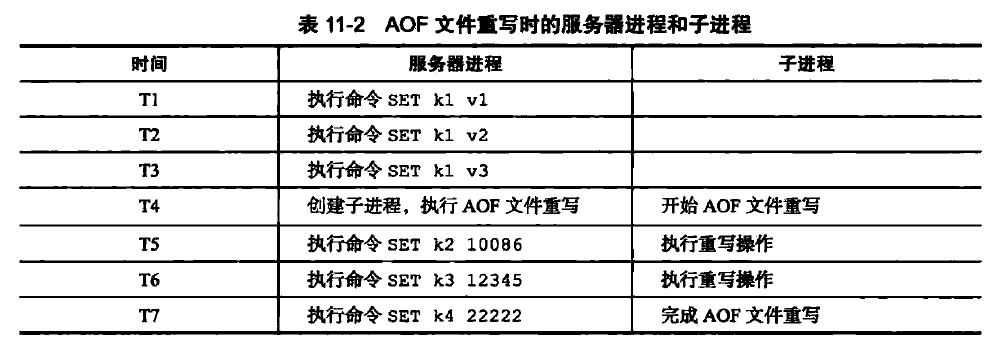
如果服务器想要用尽量少的命令来记录list键的状态，那么最简单高效的办法不是去读取和分析现有AOF文件的内容，而是直接从数据库中读取键list的值，然后用一条

RPUSH list"C""D""E""F""G"命令来代替保存在AOF文件中的六条命令，这样就可以将保存list键所需的命令从六条减少为一条了。

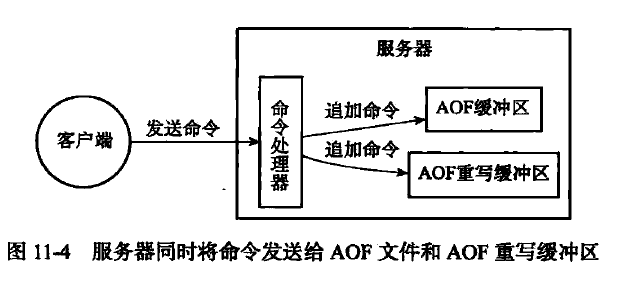
Redis将AOF重写程序放到子进程里执行，这样做可以同时达到两个目的：

* ·子进程进行AOF重写期间，服务器进程（父进程）可以继续处理命令请求。
* ·子进程带有服务器进程的数据副本，使用子进程而不是线程，可以在避免使用锁的情况下，保证数据的安全性。（因为是子进程，地址空间是父进程的拷贝）

不过，使用子进程也有一个问题需要解决，因为子进程在进行AOF重写期间，服务器进程还需要继续处理命令请求，而新的命令可能会对现有的数据库状态进行修改，从而使得服务器当前的数据库状态和重写后的AOF文件所保存的数据库状态不一致。



为了解决这种数据不一致问题，Redis服务器设置了一个AOF重写缓冲区，这个缓冲区在服务器创建子进程之后开始使用，当Redis服务器执行完一个写命令之后，它会同时将这个写命令发送给AOF缓冲区和AOF重写缓冲区



这也就是说，在子进程执行AOF重写期间，服务器进程需要执行以下三个工作：

1）执行客户端发来的命令。

2）将执行后的写命令追加到AOF缓冲区。

3）将执行后的写命令追加到AOF重写缓冲区。

这样一来可以保证：

* ·AOF缓冲区的内容会定期被写入和同步到AOF文件，对现有AOF文件的处理工作会如常进行。
* ·从创建子进程开始，服务器执行的所有写命令都会被记录到AOF重写缓冲区里面。

当子进程完成AOF重写工作之后，它会向父进程发送一个信号，父进程在接到该信号之

后，会调用一个信号处理函数，并执行以下工作：

1）将AOF重写缓冲区中的所有内容写入到新AOF文件中，这时新AOF文件所保存的数

据库状态将和服务器当前的数据库状态一致。

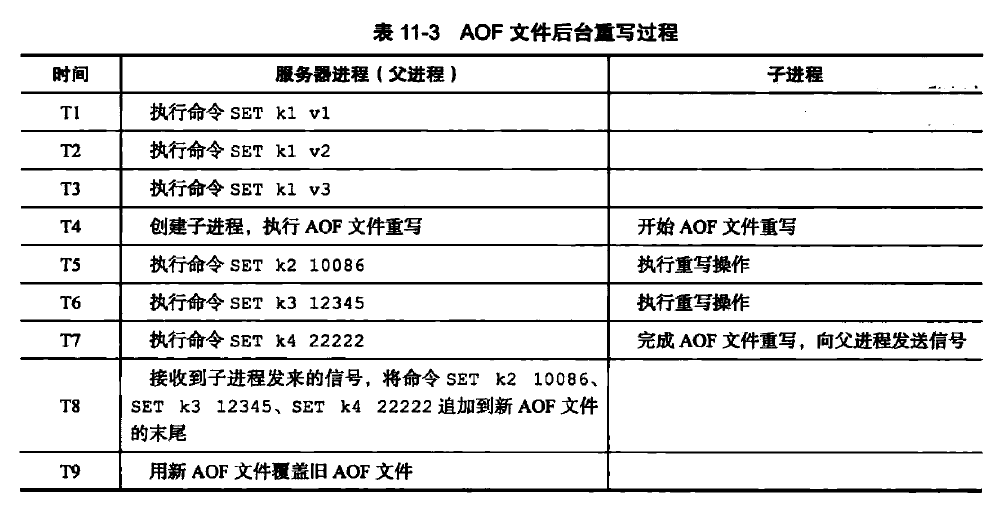
2）对新的AOF文件进行改名，原子地（atomic）覆盖现有的AOF文件，完成新旧两个

AOF文件的替换。

这个信号处理函数执行完毕之后，父进程就可以继续像往常一样接受命令请求了。

在整个AOF后台重写过程中，只有信号处理函数执行时会对服务器进程（父进程）造成阻塞，在其他时候，AOF后台重写都不会阻塞父进程，这将AOF重写对服务器性能造成的影

响降到了最低。



## 四、多机数据库复制

### 旧版本复制：

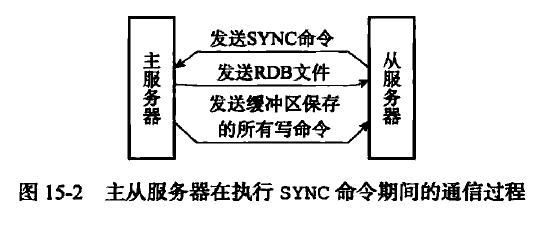
当客户端向从服务器发送SLAVEOF命令，要求从服务器复制主服务器时，从服务器首先需要执行同步操作，也即是，将从服务器的数据库状态更新至主服务器当前所处的数据库状态。从服务器对主服务器的同步操作需要通过向主服务器发送SYNC命令来完成，以下是SYNC命令的执行步骤：

1）从服务器向主服务器发送SYNC命令。

2）收到SYNC命令的主服务器执行BGSAVE命令，在后台生成一个RDB文件，并使用一个缓冲区记录从现在开始执行的所有写命令。

3）当主服务器的BGSAVE命令执行完毕时，主服务器会将BGSAVE命令生成的RDB文件发送给从服务器，从服务器接收并载入这个RDB文件，将自己的数据库状态更新至主服务器执行BGSAVE命令时的数据库状态。

4）主服务器将记录在缓冲区里面的所有写命令发送给从服务器，从服务器执行这些写命令，将自己的数据库状态更新至主服务器数据库当前所处的状态。



#### 旧版复制功能的缺陷

在Redis中，从服务器对主服务器的复制可以分为以下两种情况：

* ·初次复制：从服务器以前没有复制过任何主服务器，或者从服务器当前要复制的主服务器和上一次复制的主服务器不同。
* ·断线后重复制：处于命令传播阶段的主从服务器因为网络原因而中断了复制，但从服务器通过自动重连接重新连上了主服务器，并继续复制主服务器。

对于初次复制来说，旧版复制功能能够很好地完成任务，但对于断线后重复制来说，旧版复制功能虽然也能让主从服务器重新回到一致状态，但效率却非常低。



虽然再次发送SYNC命令可以让主从服务器重新回到一致状态，但如果我们仔细研究这个断线重复制过程，就会发现传送RDB文件这一步实际上并不是非做不可的：

* ·主从服务器在时间T0至时间T10086中一直处于一致状态，这两个服务器保存的数据大部分都是相同的。
* ·从服务器想要将自己更新至主服务器当前所处的状态，真正需要的是主从服务器连接中断期间，主服务器新添加的k10087、k10088、k10089三个键的数据。
* ·可惜的是，旧版复制功能并没有利用以上列举的两点条件，而是继续让主服务器生成并向从服务器发送包含键k1至键k10089的RDB文件，但实际上RDB文件包含的键k1至k10086的数据对于从服务器来说都是不必要的。

### 新版本复制：

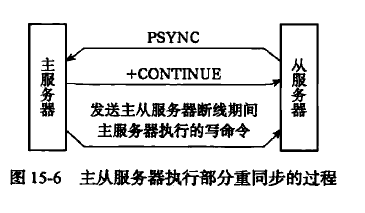
为了解决旧版复制功能在处理断线重复制情况时的低效问题，Redis从2.8版本开始，使用PSYNC命令代替SYNC命令来执行复制时的同步操作。

PSYNC命令具有完整重同步（full resynchronization）和部分重同步（partial resynchronization）两种模式：

* ·其中完整重同步用于处理初次复制情况：完整重同步的执行步骤和SYNC命令的执行步骤基本一样，它们都是通过让主服务器创建并发送RDB文件，以及向从服务器发送保存在缓冲区里面的写命令来进行同步。
* ·而部分重同步则用于处理断线后重复制情况：当从服务器在断线后重新连接主服务器时，如果条件允许，主服务器可以将主从服务器连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以将数据库更新至主服务器当前所处的状态。



虽然SYNC命令和PSYNC命令都可以让断线的主从服务器重新回到一致状态，但执行部分重同步所需的资源比起执行SYNC命令所需的资源要少得多，完成同步的速度也快得多。执行SYNC命令需要生成、传送和载入整个RDB文件，而部分重同步只需要将从服务器缺少的写命令发送给从服务器执行就可以了。



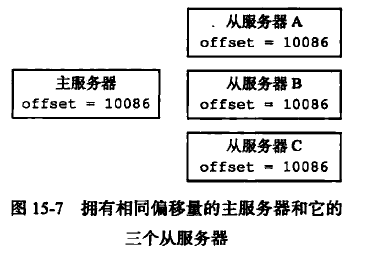
部分重同步功能由以下三个部分构成：

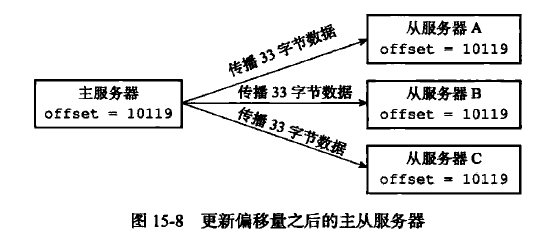
* ·主服务器的复制偏移量（replication offset）和从服务器的复制偏移量。
* ·主服务器的复制积压缓冲区（replication backlog）。
* ·服务器的运行ID（run ID）。

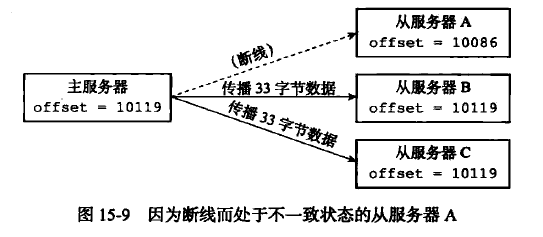
#### 复制偏移量：

执行复制的双方——主服务器和从服务器会分别维护一个复制偏移量：

* ·主服务器每次向从服务器传播N个字节的数据时，就将自己的复制偏移量的值加上N。
* ·从服务器每次收到主服务器传播来的N个字节的数据时，就将自己的复制偏移量的值加上N。







#### 复制积压缓冲区

复制积压缓冲区是由主服务器维护的一个固定长度（fixed-size）先进先出（FIFO）队列，默认大小为1MB。

固定长度先进先出队列的入队和出队规则跟普通的先进先出队列一样：新元素从一边进入队列，而旧元素从另一边弹出队列。

和普通先进先出队列随着元素的增加和减少而动态调整长度不同，固定长度先进先出队列的长度是固定的，当入队元素的数量大于队列长度时，最先入队的元素会被弹出，而新元素会被放入队列。

举个例子，如果我们要将'h'、'e'、'l'、'l'、'o'五个字符放进一个长度为3的固定长度

先进先出队列里面，那么'h'、'e'、'l'三个字符将首先被放入队列：

['h'，'e'，'l']

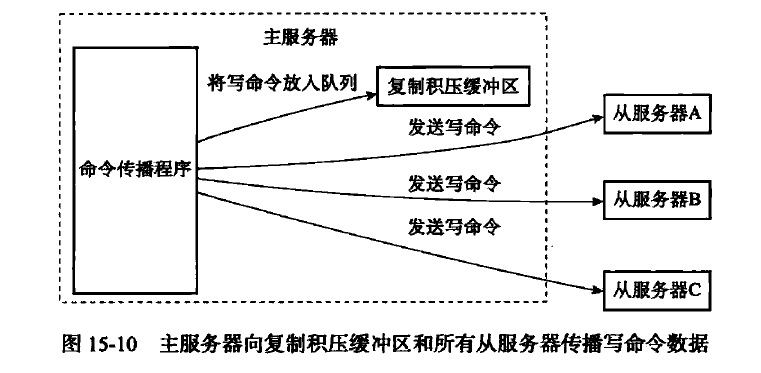
但是当后一个'l'字符要进入队列时，队首的'h'字符将被弹出，队列变成：

['e'，'l'，'l']

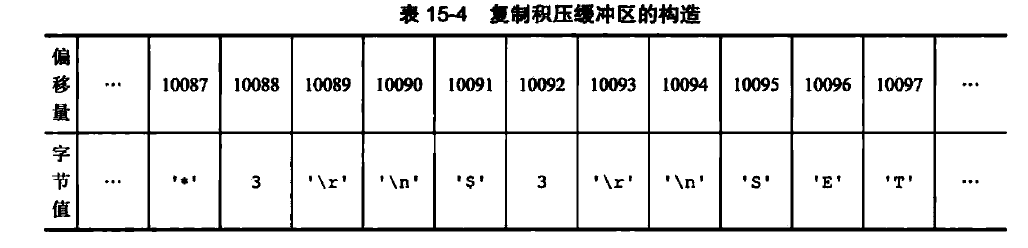
接着，'o'的入队会引起'e'的出队，队列变成：

['l'，'l'，'o']

当主服务器进行命令传播时，它不仅会将写命令发送给所有从服务器，还会将写命令入队到复制积压缓冲区里面



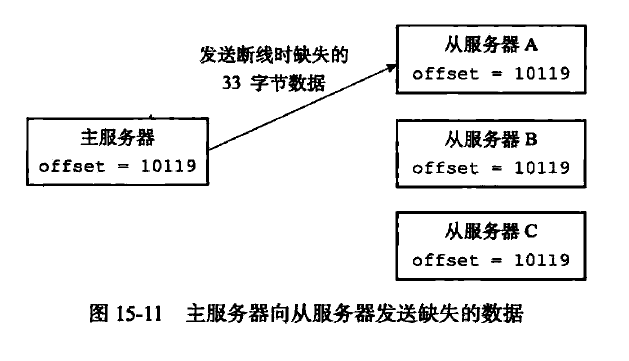
主服务器的复制积压缓冲区里面会保存着一部分最近传播的写命令，并且复制积压缓冲区会为队列中的每个字节记录相应的复制偏移量



当从服务器重新连上主服务器时，从服务器会通过PSYNC命令将自己的复制偏移量offset发送给主服务器，主服务器会根据这个复制偏移量来决定对从服务器执行何种同步操作：

* ·如果offset偏移量之后的数据（也即是偏移量offset+1开始的数据）仍然存在于复制积压缓冲区里面，那么主服务器将对从服务器执行部分重同步操作。
* ·相反，如果offset偏移量之后的数据已经不存在于复制积压缓冲区，那么主服务器将对从服务器执行完整重同步操作。

回到之前图15-9展示的断线后重连接例子：



* ·当从服务器A断线之后，它立即重新连接主服务器，并向主服务器发送PSYNC命令，报告自己的复制偏移量为10086。
* ·主服务器收到从服务器发来的PSYNC命令以及偏移量10086之后，主服务器将检查偏移量10086之后的数据是否存在于复制积压缓冲区里面，结果发现这些数据仍然存在，于是主服务器向从服务器发送+CONTINUE回复，表示数据同步将以部分重同步模式来进行。
* ·接着主服务器会将复制积压缓冲区10086偏移量之后的所有数据（偏移量为10087至10119）都发送给从服务器。
* ·从服务器只要接收这33字节的缺失数据，就可以回到与主服务器一致的状态，如图15-11所示。

#### 服务器运行ID

除了复制偏移量和复制积压缓冲区之外，实现部分重同步还需要用到服务器运行ID（run ID）：

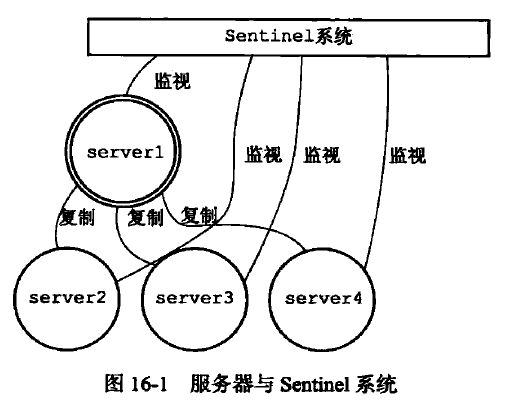
* ·每个Redis服务器，不论主服务器还是从服务，都会有自己的运行ID。
* ·运行ID在服务器启动时自动生成，由40个随机的十六进制字符组成，例如53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3。

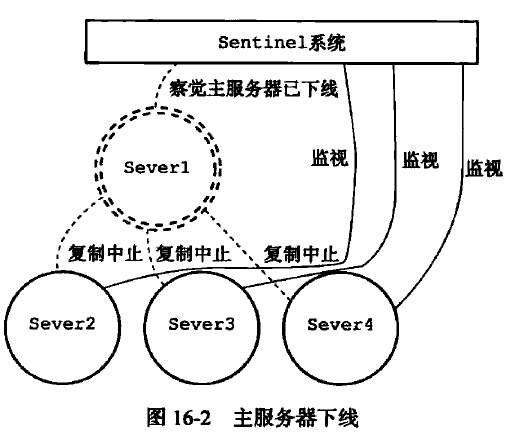
当从服务器对主服务器进行初次复制时，主服务器会将自己的运行ID传送给从服务器，而从服务器则会将这个运行ID保存起来。

当从服务器断线并重新连上一个主服务器时，从服务器将向当前连接的主服务器发送之前保存的运行ID.

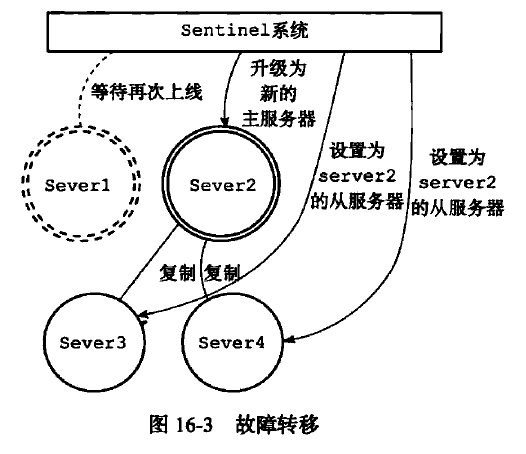
## 五、Sentinel哨兵

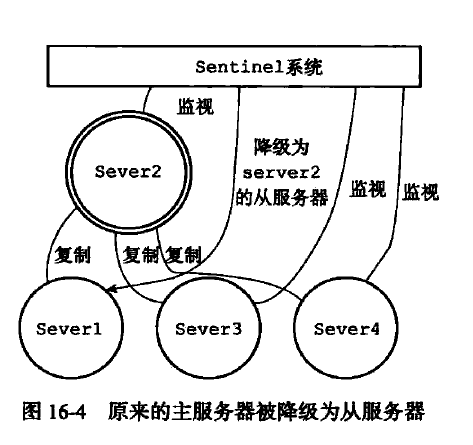
Sentinel（哨岗、哨兵）是Redis的高可用性（high availability ）解决方案：由一个或多个Sentinel实例（instance）组成的Sentinel系统（sy stem）可以监视任意多个主服务器，及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器，然后由新的主服务器代替已下线的主服务器继续处理命令请求。





如上图所示：·server2、server3、server4三个从服务器正在复制主服务器server1，而Sentinel系统则在监视所有四个服务器。假设这时，主服务器server1进入下线状态，那么从服务器server2、server3、server4对主服务器的复制操作将被中止，并且Sentinel系统会察觉到server1已下线





当server1的下线时长超过用户设定的下线时长上限时，Sentinel系统就会对server1执行故障转移操作：

* ·首先，Sentinel系统会挑选server1属下的其中一个从服务器，并将这个被选中的从服务器升级为新的主服务器。
* ·之后，Sentinel系统会向server1属下的所有从服务器发送新的复制指令，让它们成为新的主服务器的从服务器，当所有从服务器都开始复制新的主服务器时，故障转移操作执行完毕。
* ·另外，Sentinel还会继续监视已下线的server1，并在它重新上线时，将它设置为新的主服务器的从服务器。

### 选举领头Sentinel

当一个主服务器被判断为客观下线时，监视这个下线主服务器的各个Sentinel会进行协商，选举出一个领头Sentinel，并由领头Sentinel对下线主服务器执行故障转移操作。以下是Redis选举领头Sentinel的规则和方法：

* ·所有在线的Sentinel都有被选为领头Sentinel的资格，换句话说，监视同一个主服务器的多个在线Sentinel中的任意一个都有可能成为领头Sentinel。
* ·每次进行领头Sentinel选举之后，不论选举是否成功，所有Sentinel的配置纪元（configuration epoch）的值都会自增一次。配置纪元实际上就是一个计数器，并没有什么特别的。
* ·在一个配置纪元里面，所有Sentinel都有一次将某个Sentinel设置为局部领头Sentinel的机会，并且局部领头一旦设置，在这个配置纪元里面就不能再更改。
* ·每个发现主服务器进入客观下线的Sentinel都会要求其他Sentinel将自己设置为局部领头Sentinel。
* ·当一个Sentinel（源Sentinel）向另一个Sentinel（目标Sentinel）发送SENTINEL ismaster-down-by -addr命令，并且命令中的runid参数不是\*符号而是源Sentinel的运行ID时，这表示源Sentinel要求目标Sentinel将前者设置为后者的局部领头Sentinel。
* ·Sentinel设置局部领头Sentinel的规则是先到先得：最先向目标Sentinel发送设置要求的源Sentinel将成为目标Sentinel的局部领头Sentinel，而之后接收到的所有设置要求都会被标Sentinel拒绝。
* ·目标Sentinel在接收到SENTINEL is-master-down-by -addr命令之后，将向源Sentinel回一条命令回复，回复中的leader\_runid参数和leader\_epoch参数分别记录了目标Sentinel的局部领头Sentinel的运行ID和配置纪元。
* ·源Sentinel在接收到目标Sentinel返回的命令回复之后，会检查回复中leader\_epoch参数的值和自己的配置纪元是否相同，如果相同的话，那么源Sentinel继续取出回复中的leader\_runid参数，如果leader\_runid参数的值和源Sentinel的运行ID一致，那么表示目标Sentinel将源Sentinel设置成了局部领头Sentinel。
* ·如果有某个Sentinel被半数以上的Sentinel设置成了局部领头Sentinel，那么这个Sentinel成为领头Sentinel。举个例子，在一个由10个Sentinel组成的Sentinel系统里面，只要有大于等于10/2+1=6个Sentinel将某个Sentinel设置为局部领头Sentinel，那么被设置的那个Sentinel就会成为领头Sentinel。
* ·因为领头Sentinel的产生需要半数以上Sentinel的支持，并且每个Sentinel在每个配置纪元里面只能设置一次局部领头Sentinel，所以在一个配置纪元里面，只会出现一个领头Sentinel。
* ·如果在给定时限内，没有一个Sentinel被选举为领头Sentinel，那么各个Sentinel将在一段时间之后再次进行选举，直到选出领头Sentinel为止。

### 判断下线与故障转移：

* ·Sentinel以每秒一次的频率向实例（包括主服务器、从服务器、其他Sentinel）发送PING命令，并根据实例对PING命令的回复来判断实例是否在线，当一个实例在指定的时长中连续向Sentinel发送无效回复时，Sentinel会将这个实例判断为主观下线。
* ·当Sentinel将一个主服务器判断为主观下线时，它会向同样监视这个主服务器的其他Sentinel进行询问，看它们是否同意这个主服务器已经进入主观下线状态。
* ·当Sentinel收集到足够多的主观下线投票之后，它会将主服务器判断为客观下线，并发起一次针对主服务器的故障转移操作（先选出主sentinel，再由主sentinal进行转移）。

## 六、集群

一个Redis集群通常由多个节点（node）组成，在刚开始的时候，每个节点都是相互独立的，它们都处于一个只包含自己的集群当中，要组建一个真正可工作的集群，我们必须将各个独立的节点连接起来，构成一个包含多个节点的集群。

向一个节点node发送CLUSTER MEET命令，可以让node节点与ip和port所指定的节点进行握手（handshake），当握手成功时，node节点就会将ip和port所指定的节点添加到node节点当前所在的集群中。

### 槽指派：

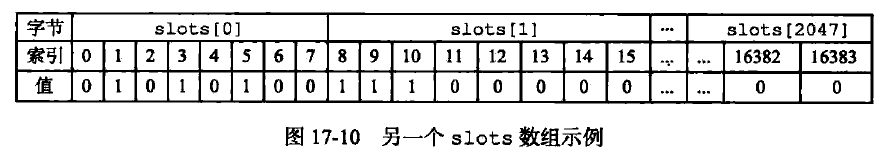
Redis集群通过分片的方式来保存数据库中的键值对：集群的整个数据库被分为16384个槽（slot），数据库中的每个键都属于这16384个槽的其中一个，集群中的每个节点可以处理0个或最多16384个槽。

slots属性是一个二进制位数组（bit array ），这个数组的长度为16384/8=2048个字节，共包含16384个二进制位。

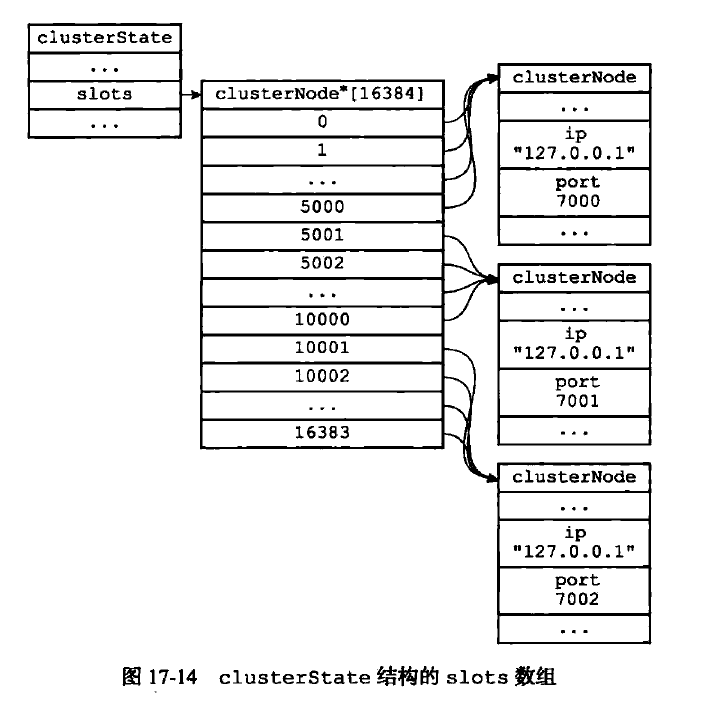
Redis以0为起始索引，16383为终止索引，对slots数组中的16384个二进制位进行编号，并根据索引i上的二进制位的值来判断节点是否负责处理槽i：

* ·如果slots数组在索引i上的二进制位的值为1，那么表示节点负责处理槽i。
* ·如果slots数组在索引i上的二进制位的值为0，那么表示节点不负责处理槽i。

下图数组索引1、3、5、8、9、10上的二进制位的值都为1，而其余所有二进制位的值都为0，这表示节点负责处理槽1、3、5、8、9、10。



集群中的每个节点都会将自己的slots数组通过消息发送给集群中的其他节点，并且每个接收到slots数组的节点都会将数组保存到相应节点的clusterNode结构里面，因此，集群中的每个节点都会知道数据库中的16384个槽分别被指派给了集群中的哪些节点。



当节点计算出键所属的槽i之后，节点就会检查自己在clusterState.slots数组中的项i，判断键所在的槽是否由自己负责：

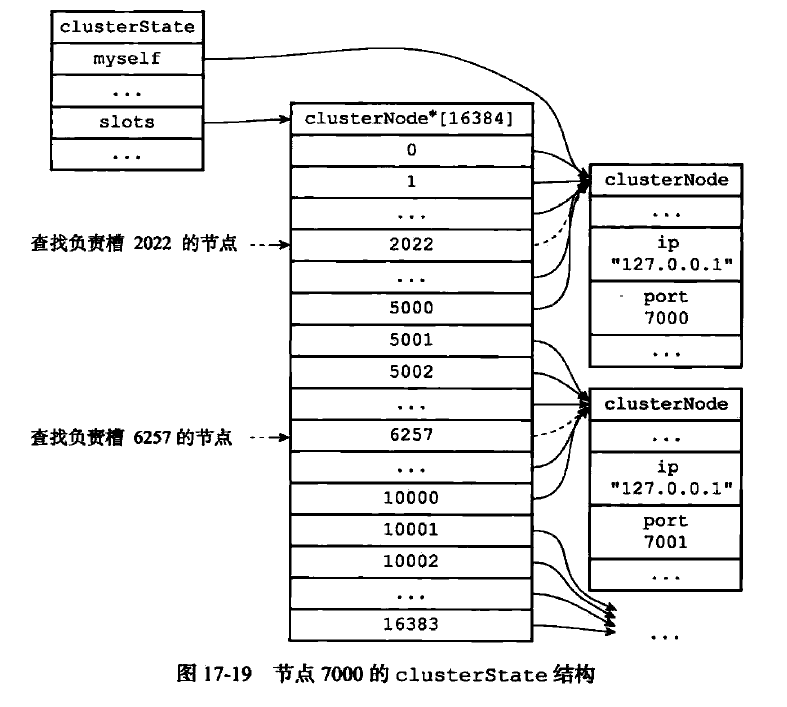
1）如果clusterState.slots[i]等于clusterState.my self，那么说明槽i由当前节点负责，节点可以执行客户端发送的命令。

2）如果clusterState.slots[i]不等于clusterState.my self，那么说明槽i并非由当前节点负责，节点会根据clusterState.slots[i]指向的clusterNode结构所记录的节点IP和端口号，向客户端返回MOVED错误，指引客户端转向至正在处理槽i的节点。

举个例子，假设图17-19为节点7000的clusterState结构：

·当客户端向节点7000发送命令SET date"2013-12-31"的时候，节点首先计算出键date属于槽2022，然后检查得出clusterState.slots[2022]等于clusterState.my self，这说明槽2022正是由节点7000负责，于是节点7000直接执行这个SET命令，并将结果返回给发送命令的客户端。

·当客户端向节点7000发送命令SET msg"happy new y ear！"的时候，节点首先计算出键msg属于槽6257，然后检查clusterState.slots[6257]是否等于clusterState.my self，结果发现两者并不相等：这说明槽6257并非由节点7000负责处理，于是节点7000访问clusterState.slots[6257]所指向的clusterNode结构，并根据结构中记录的IP地址127.0.0.1和端口号7001，向客户端返回错误MOVED 6257 127.0.0.1:7001，指引节点转向至正在负责处理槽6257的节点7001。



除了将键值对保存在数据库里面之外，节点还会用clusterState结构中的slots\_to\_key s跳跃表来保存槽和键之间的关系：

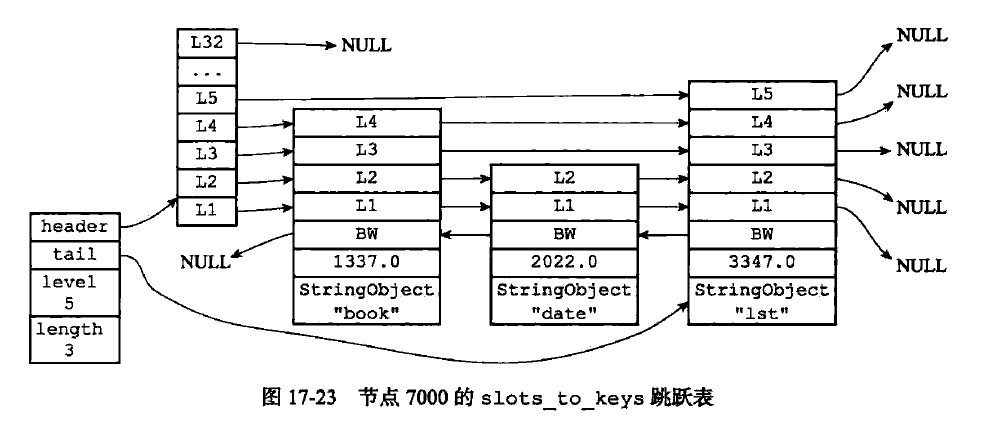
slots\_to\_key s跳跃表每个节点的分值（score）都是一个槽号，而每个节点的成员（member）都是一个数据库键：

举个例子，对于图17-22所示的数据库，节点7000将创建类似图17-23所示的slots\_to\_key

·键"book"所在跳跃表节点的分值为1337.0，这表示键"book"所在的槽为1337。

·键"date"所在跳跃表节点的分值为2022.0，这表示键"date"所在的槽为2022。

·键"lst"所在跳跃表节点的分值为3347.0，这表示键"lst"所在的槽为3347。



## 七、事务

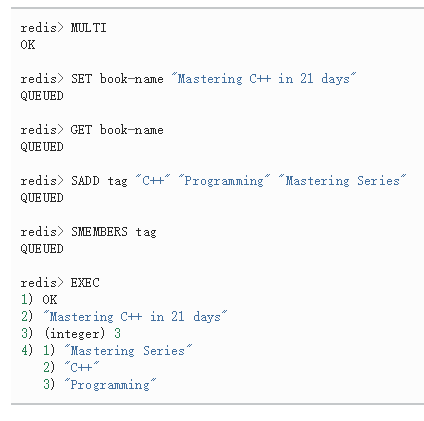
首先，键事务支持有限，当多个key分布在不同节点时无法使用事务，同一节点是支持事务。

其次，在单机上的事务的支持也只部分支持。

Redis 通过 MULTI 、 DISCARD 、 EXEC 和 WATCH 四个命令来实现事务功能

事务提供了一种“将多个命令打包， 然后一次性、按顺序地执行”的机制， 并且事务在执行的期间不会主动中断 —— 服务器在执行完事务中的所有命令之后， 才会继续处理其他客户端的其他命令。

以下是一个事务的例子， 它先以 MULTI 开始一个事务， 然后将多个命令入队到事务中， 最后由 EXEC 命令触发事务， 一并执行事务中的所有命令：



一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

下文将分别介绍事务的这三个阶段。

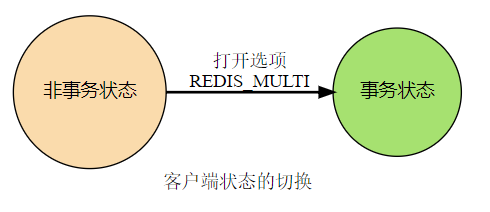
### 开始事务。

MULTI 命令的执行标记着事务的开始：

redis> MULTI

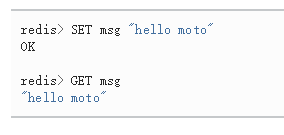
OK

这个命令唯一做的就是， 将客户端的 REDIS\_MULTI 选项打开， 让客户端从非事务状态切换到事务状态。

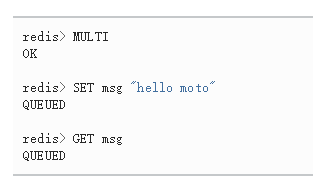


### 命令入队。

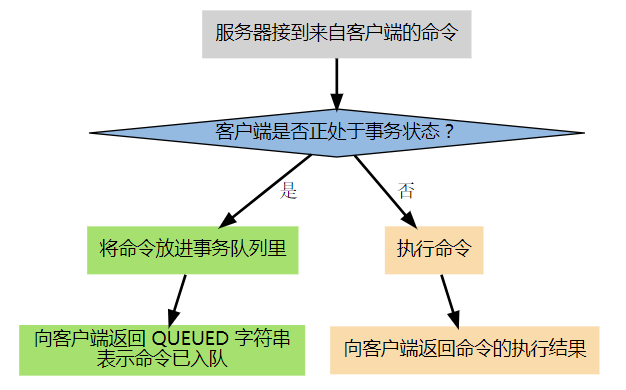
当客户端处于非事务状态下时， 所有发送给服务器端的命令都会立即被服务器执行：



但是， 当客户端进入事务状态之后， 服务器在收到来自客户端的命令时， 不会立即执行命令， 而是将这些命令全部放进一个事务队列里， 然后返回 QUEUED ， 表示命令已入队：



以下流程图展示了这一行为：



### 执行事务。

前面说到， 当客户端进入事务状态之后， 客户端发送的命令就会被放进事务队列里。

但其实并不是所有的命令都会被放进事务队列， 其中的例外就是 EXEC 、 DISCARD 、 MULTI 和 WATCH 这四个命令 —— 当这四个命令从客户端发送到服务器时， 它们会像客户端处于非事务状态一样， 直接被服务器执行。如果客户端正处于事务状态， 那么当 EXEC 命令执行时， 服务器根据客户端所保存的事务队列， 以先进先出（FIFO）的方式执行事务队列中的命令： 最先入队的命令最先执行， 而最后入队的命令最后执行。

比如说，对于以下事务队列：

| **数组索引** | **cmd** | **argv** | **argc** |
| --- | --- | --- | --- |
| 0 | SET | ["book-name", "Mastering C++ in 21 days"] | 2 |
| 1 | GET | ["book-name"] | 1 |
| 2 | SADD | ["tag", "C++", "Programming", "Mastering Series"] | 4 |
| 3 | SMEMBERS | ["tag"] | 1 |

程序会首先执行 [SET](http://redis.readthedocs.org/en/latest/string/set.html#set) 命令， 然后执行 [GET](http://redis.readthedocs.org/en/latest/string/get.html#get) 命令， 再然后执行 [SADD](http://redis.readthedocs.org/en/latest/set/sadd.html#sadd) 命令， 最后执行 [SMEMBERS](http://redis.readthedocs.org/en/latest/set/smembers.html#smembers) 命令。

执行事务中的命令所得的结果会以 FIFO 的顺序保存到一个回复队列中。

比如说，对于上面给出的事务队列，程序将为队列中的命令创建如下回复队列：

| **数组索引** | **回复类型** | **回复内容** |
| --- | --- | --- |
| 0 | status code reply | OK |
| 1 | bulk reply | "Mastering C++ in 21 days" |
| 2 | integer reply | 3 |
| 3 | multi-bulk reply | ["Mastering Series", "C++", "Programming"] |

当事务队列里的所有命令被执行完之后， [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令会将回复队列作为自己的执行结果返回给客户端， 客户端从事务状态返回到非事务状态， 至此， 事务执行完毕。

**在事务状态和非事务状态下的区别：**

无论在事务状态下， 还是在非事务状态下， Redis 命令都由同一个函数执行， 所以它们共享很多服务器的一般设置， 比如 AOF 的配置、RDB 的配置，以及内存限制，等等。

不过事务中的命令和普通命令在执行上还是有一点区别的，其中最重要的两点是：

1. 非事务状态下的命令以单个命令为单位执行，前一个命令和后一个命令的客户端不一定是同一个；

而事务状态则是以一个事务为单位，执行事务队列中的所有命令：除非当前事务执行完毕，否则服务器不会中断事务，也不会执行其他客户端的其他命令。

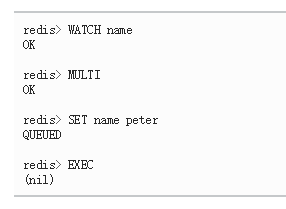
1. 在非事务状态下，执行命令所得的结果会立即被返回给客户端；

而事务则是将所有命令的结果集合到回复队列，再作为 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令的结果返回给客户端。

### 带watch的事务

[WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 命令用于在事务开始之前监视任意数量的键： 当调用 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令执行事务时， 如果任意一个被监视的键已经被其他客户端修改了， 那么整个事务不再执行， 直接返回失败。

以下示例展示了一个执行失败的事务例子：



以下执行序列展示了上面的例子是如何失败的：

| **时间** | **客户端 A** | **客户端 B** |
| --- | --- | --- |
| T1 | WATCH name |  |
| T2 | MULTI |  |
| T3 | SET name peter |  |
| T4 |  | SET name john |
| T5 | EXEC |  |

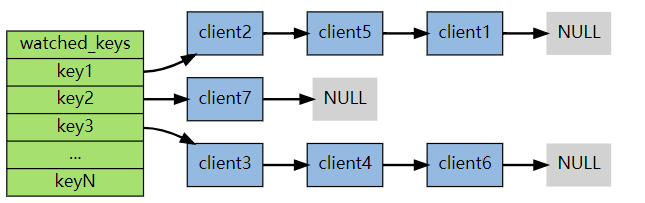
在时间 T4 ，客户端 B 修改了 name 键的值， 当客户端 A 在 T5 执行 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 时，Redis 会发现 name 这个被监视的键已经被修改， 因此客户端 A 的事务不会被执行，而是直接返回失败。

下文就来介绍 [WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 的实现机制，并且看看事务系统是如何检查某个被监视的键是否被修改，从而保证事务的安全性的。

### Watch命令的实现

在每个代表数据库的 redis.h/redisDb 结构类型中， 都保存了一个 watched\_keys 字典， 字典的键是这个数据库被监视的键， 而字典的值则是一个链表， 链表中保存了所有监视这个键的客户端。

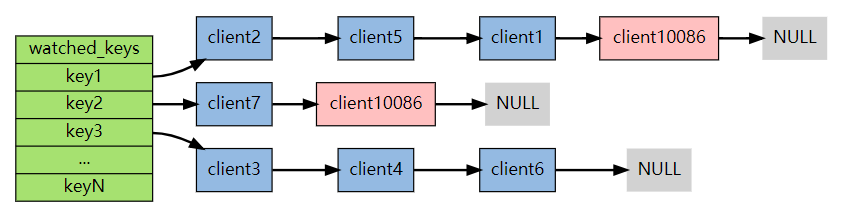
比如说，以下字典就展示了一个 watched\_keys 字典的例子：



其中， 键 key1 正在被 client2 、 client5 和 client1 三个客户端监视， 其他一些键也分别被其他别的客户端监视着。

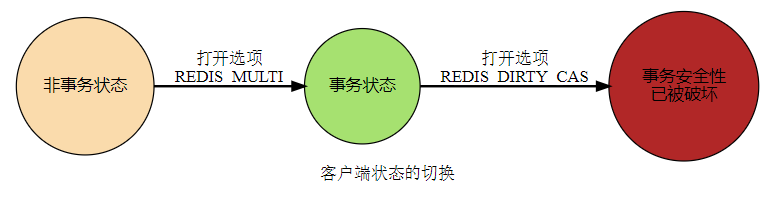
[WATCH](http://redis.readthedocs.org/en/latest/transaction/watch.html#watch) 命令的作用， 就是将当前客户端和要监视的键在 watched\_keys 中进行关联。

举个例子， 如果当前客户端为 client10086 ， 那么当客户端执行 WATCH key1 key2 时， 前面展示的 watched\_keys 将被修改成这个样子：



通过 watched\_keys 字典， 如果程序想检查某个键是否被监视， 那么它只要检查字典中是否存在这个键即可； 如果程序要获取监视某个键的所有客户端， 那么只要取出键的值（一个链表）， 然后对链表进行遍历即可。

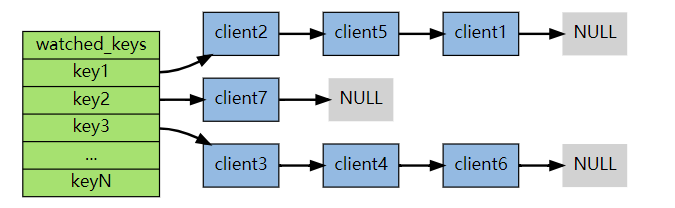
在任何对数据库键空间（key space）进行修改的命令成功执行之后 （比如 [FLUSHDB](http://redis.readthedocs.org/en/latest/server/flushdb.html#flushdb) 、 [SET](http://redis.readthedocs.org/en/latest/string/set.html#set) 、 [DEL](http://redis.readthedocs.org/en/latest/key/del.html#del) 、 [LPUSH](http://redis.readthedocs.org/en/latest/list/lpush.html#lpush) 、 [SADD](http://redis.readthedocs.org/en/latest/set/sadd.html#sadd) 、 [ZREM](http://redis.readthedocs.org/en/latest/sorted_set/zrem.html#zrem) ，诸如此类）， multi.c/touchWatchedKey 函数都会被调用 —— 它检查数据库的 watched\_keys 字典， 看是否有客户端在监视已经被命令修改的键， 如果有的话， 程序将所有监视这个/这些被修改键的客户端的 REDIS\_DIRTY\_CAS 选项打开：



当客户端发送 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令、触发事务执行时， 服务器会对客户端的状态进行检查：

* 如果客户端的 REDIS\_DIRTY\_CAS 选项已经被打开，那么说明被客户端监视的键至少有一个已经被修改了，事务的安全性已经被破坏。服务器会放弃执行这个事务，直接向客户端返回空回复，表示事务执行失败。
* 如果 REDIS\_DIRTY\_CAS 选项没有被打开，那么说明所有监视键都安全，服务器正式执行事务。

举个例子，假设数据库的 watched\_keys 字典如下图所示：



如果某个客户端对 key1 进行了修改（比如执行 DEL key1 ）， 那么所有监视 key1 的客户端， 包括 client2 、 client5 和 client1 的 REDIS\_DIRTY\_CAS 选项都会被打开， 当客户端 client2 、 client5 和 client1 执行 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 的时候， 它们的事务都会以失败告终。

最后，当一个客户端结束它的事务时，无论事务是成功执行，还是失败， watched\_keys 字典中和这个客户端相关的资料都会被清除。

### Redis事务的ACID：

### **原子性（Atomicity）**

要么不做，要么全做。Redis事务是具有原子性的。但是redis不支持回滚操作，事务队列中某个命令执行期间出现错误，整个事务也会继续执行下去，直到事务队列中的所有命令都执行完成。

### **一致性（Consistency）**

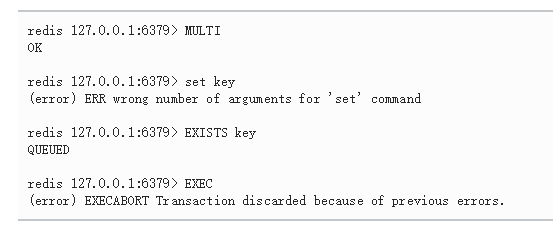
执行前处于一个一致的状态，执行后也处于一个一致的状态

Redis 的一致性问题可以分为三部分来讨论：入队错误、执行错误、Redis 进程被终结。

#### 入队错误

在命令入队的过程中，如果客户端向服务器发送了错误的命令，比如命令的参数数量不对，等等， 那么服务器将向客户端返回一个出错信息， 并且将客户端的事务状态设为 REDIS\_DIRTY\_EXEC 。

当客户端执行 [EXEC](http://redis.readthedocs.org/en/latest/transaction/exec.html#exec) 命令时， Redis 会拒绝执行状态为 REDIS\_DIRTY\_EXEC 的事务， 并返回失败信息。



因此，带有不正确入队命令的事务不会被执行，也不会影响数据库的一致性。

#### 执行错误

如果命令在事务执行的过程中发生错误，比如说，对一个不同类型的 key 执行了错误的操作， 那么 Redis 只会将错误包含在事务的结果中， 这不会引起事务中断或整个失败，不会影响已执行事务命令的结果，也不会影响后面要执行的事务命令， 所以它对事务的一致性也没有影响。

#### Redis 进程被终结

如果 Redis 服务器进程在执行事务的过程中被其他进程终结，或者被管理员强制杀死，那么根据 Redis 所使用的持久化模式，可能有以下情况出现：

* 内存模式：如果 Redis 没有采取任何持久化机制，那么重启之后的数据库总是空白的，所以数据总是一致的。
* RDB 模式：在执行事务时，Redis 不会中断事务去执行保存 RDB 的工作，只有在事务执行之后，保存 RDB 的工作才有可能开始。所以当 RDB 模式下的 Redis 服务器进程在事务中途被杀死时，事务内执行的命令，不管成功了多少，都不会被保存到 RDB 文件里。恢复数据库需要使用现有的 RDB 文件，而这个 RDB 文件的数据保存的是最近一次的数据库快照（snapshot），所以它的数据可能不是最新的，但只要 RDB 文件本身没有因为其他问题而出错，那么还原后的数据库就是一致的。
* AOF 模式：因为保存 AOF 文件的工作在后台线程进行，所以即使是在事务执行的中途，保存 AOF 文件的工作也可以继续进行，因此，根据事务语句是否被写入并保存到 AOF 文件，有以下两种情况发生：

1）如果事务语句未写入到 AOF 文件，或 AOF 未被 SYNC 调用保存到磁盘，那么当进程被杀死之后，Redis 可以根据最近一次成功保存到磁盘的 AOF 文件来还原数据库，只要 AOF 文件本身没有因为其他问题而出错，那么还原后的数据库总是一致的，但其中的数据不一定是最新的。

2）如果事务的部分语句被写入到 AOF 文件，并且 AOF 文件被成功保存，那么不完整的事务执行信息就会遗留在 AOF 文件里，当重启 Redis 时，程序会检测到 AOF 文件并不完整，Redis 会退出，并报告错误。需要使用 redis-check-aof 工具将部分成功的事务命令移除之后，才能再次启动服务器。还原之后的数据总是一致的，而且数据也是最新的（直到事务执行之前为止）。

### **隔离性（Isolation）**

Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

### **持久性（Durability）**

因为事务不过是用队列包裹起了一组 Redis 命令，并没有提供任何额外的持久性功能，所以事务的持久性由 Redis 所使用的持久化模式决定：

* 在单纯的内存模式下，事务肯定是不持久的。
* 在 RDB 模式下，服务器可能在事务执行之后、RDB 文件更新之前的这段时间失败，所以 RDB 模式下的 Redis 事务也是不持久的。
* 在 AOF 的“总是 SYNC ”模式下，事务的每条命令在执行成功之后，都会立即调用 fsync 或 fdatasync 将事务数据写入到 AOF 文件。但是，这种保存是由后台线程进行的，主线程不会阻塞直到保存成功，所以从命令执行成功到数据保存到硬盘之间，还是有一段非常小的间隔，所以这种模式下的事务也是不持久的。

其他 AOF 模式也和“总是 SYNC ”模式类似，所以它们都是不持久的。

## 八、常见面试问题

<https://juejin.im/post/5ad6e4066fb9a028d82c4b66>

### 用redis实现一个分布式锁

当多个进程不在同一个系统中，就需要用分布式锁控制多个进程对资源的访问。

使用redis来实现分布式锁主要用到以下命令：

* SETNX KEY VALUE 如果key不存在，就设置key对应字符串value
* expire KEY seconds 设置key的过期时间
* del KEY 删除key

SETNX key value：

只在键 key 不存在的情况下， 将键 key 的值设置为 value 。若键 key 已经存在， 则 SETNX 命令不做任何动作。SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

返回值：

命令在设置成功时返回 1 ， 设置失败时返回 0 。

1. $redis->multi();
2. $redis->setNX($key, $value);
3. $redis->expire($key, $ttl);
4. $res = $redis->exec();
5. **if**($res[0]) {
6. //获取到锁
7. ... **do** something ...
8. $redis->del($key);
9. }

### MySQL里有2000w数据，Redis中只存20w的数据，如何保证Redis中的数据都是热点数据（redis有哪些数据淘汰策略？）

   相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略（回收策略）。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（快到要设定的expire time）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

* volatile-\* 系列指令在无键值满足条件时（例如未设置过期时间），表现noeviction
* volatile-lru和volatile-ttl都是随机选择几个（默认是5）键值，在其中找lru和ttl，不是精确的，是基于概率的。
* 注意与用户主动设定expire的区别，淘汰策略是内存中装不下的时候进行淘汰和置换。设定expire是无论内存状态，到期剔除。

### Redis的并发竞争问题如何解决?

Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问（前面提到的事务）。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Redis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有2种解决方法：

1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。

2.服务器角度，利用setnx实现锁。

注：对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

===================================

* 如果是单机情况（可能有副本拷贝，但所有数据都存在一台节点上），直接使用redis提供的事务机制即可
* 如果是集群，结合setnx实现一个简单的基于redis自己的锁，进行同步

### Redis回收使用的是什么算法?

就是前面提到的淘汰策略

### Redis 大量数据插入

官方文档给的解释：[www.redis.cn/topics/mass…](https://link.juejin.im?target=http%3A%2F%2Fwww.redis.cn%2Ftopics%2Fmass-insert.html)

### Redis 分区的优势、不足以及分区类型

官方文档提供的讲解：[www.redis.net.cn/tutorial/35…](https://link.juejin.im?target=http%3A%2F%2Fwww.redis.net.cn%2Ftutorial%2F3524.html)

分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集。（这里分区就是前面的槽机制，多个redis构成一个集群）

**分区的优势**

通过利用多台计算机内存的和值，允许我们构造更大的数据库。

通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

**分区的不足**

redis的一些特性在分区方面表现的不是很好：

* 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
* 涉及多个key的redis事务不能使用。
* 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
* 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

### Redis持久化数据和缓存怎么做扩容？

**《redis的持久化和缓存机制》** ：[github.com/Snailclimb/…](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2FSnailclimb%2FJava-Guide%2Fblob%2Fmaster%2F%25E6%2595%25B0%25E6%258D%25AE%25E5%25AD%2598%25E5%2582%25A8%2F%25E6%2598%25A5%25E5%25A4%258F%25E7%25A7%258B%25E5%2586%25AC%25E5%258F%2588%25E4%25B8%2580%25E6%2598%25A5%25E4%25B9%258BRedis%25E6%258C%2581%25E4%25B9%2585%25E5%258C%2596.md)

扩容的话可以通过redis集群实现，之前做项目的时候用过自己搭的redis集群 然后写了一篇关于redis集群的文章：**《一文轻松搞懂redis集群原理及搭建与使用》**：[juejin.im/post/5ad54d…](https://juejin.im/post/5ad54d76f265da23970759d3)

* 持久化：RDB、AOF
* 扩容：槽机制

### Redis常见性能问题和解决方案:

1. Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
2. 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次
3. 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内
4. 尽量避免在压力很大的主库上增加从库

### Redis与消息队列

作者：翁伟 链接：https://www.zhihu.com/question/20795043/answer/345073457

不要使用redis去做消息队列，这不是redis的设计目标。但实在太多人使用redis去做去消息队列，redis的作者看不下去，另外基于redis的核心代码，另外实现了一个消息队列disque： antirez/disque:[github.com/antirez/dis…](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fantirez%2Fdisque)部署、协议等方面都跟redis非常类似，并且支持集群，延迟消息等等。

redis做缓存才是正道，首页缓存等等。

redis实现消息队列可以参看此博客：<https://segmentfault.com/a/1190000012244418>

#### 为什么要用Redis实现轻量级MQ？

在业务的实现过程中，就算没有大量的流量，解耦和异步化几乎也是处处可用，此时MQ就显得尤为重要。但与此同时MQ也是一个蛮重的组件，例如我们如果用RabbitMQ就必须为它搭建一个服务器，同时如果要考虑可用性，就要为服务端建立一个集群，而且在生产如果有问题也需要查找功能。在中小型业务的开发过程中，可能业务的其他整个实现都没这个重。过重的组件服务会成倍增加工作量。

所幸的是，Redis提供的list数据结构非常适合做消息队列。

但是如何实现即时消费？如何实现ack机制？这些是实现的关键所在。

#### 如何实现即时消费？

网上所流传的方法是使用Redis中list的操作BLPOP或BRPOP，即列表的阻塞式(blocking)弹出。

让我们来看看阻塞式弹出的使用方式：

BRPOP key [key ...] timeout

此命令的说明是：

1、当给定列表内没有任何元素可供弹出的时候，连接将被 BRPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

2、当给定多个key参数时，按参数 key 的先后顺序依次检查各个列表，弹出第一个非空列表的尾部元素。

另外，BRPOP 除了弹出元素的位置和 BLPOP 不同之外，其他表现一致。

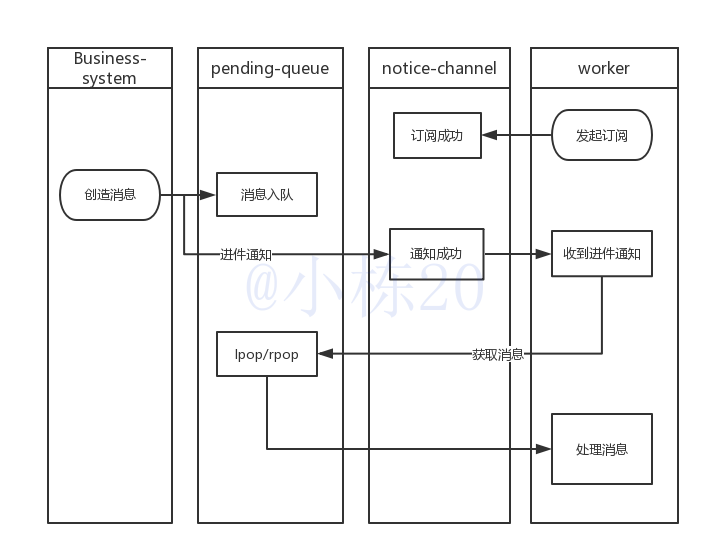
以此来看，列表的阻塞式弹出有两个特点：

1、如果list中没有任务的时候，该连接将会被阻塞

2、连接的阻塞有一个超时时间，当超时时间设置为0时，即可无限等待，直到弹出消息

由此看来，此方式是可行的，但此为传统的观察者模式，业务简单则可使用，如A的任务只由B去执行。但如果A和Z的任务，B和C都能执行，那使用这种方式就相形见肘。这个时候就应该使用订阅/发布模式，使业务系统更加清晰。

好在Redis也支持Pub/Sub（发布/订阅）。在消息A入队list的同时发布（PUBLISH）消息B到频道channel，此时已经订阅channel的worker就接收到了消息B，知道了list中有消息A进入，即可循环lpop或rpop来消费list中的消息。流程如下：



其中的worker可以是单独的线程，也可以是独立的服务，其充当了Consumer和业务处理者角色。

#### Redis实现ACK

主要解决两个问题：

* work处理失败后，要回滚消息到原始pending队列
* 假如worker挂掉，也要回滚消息到原始pending队列

具体做法：

1、维护两个队列：pending队列和doing表（hash表）。

2、workers定义为ThreadPool。

3、由pending队列出队后，workers分配一个线程（单个worker）去处理消息——给目标消息append一个当前时间戳和当前线程名称，将其写入doing表，然后该worker去消费消息，完成后自行在doing表擦除信息。

4、启用一个定时任务，每隔一段时间去扫描doing队列，检查每隔元素的时间戳，如果超时，则由worker的ThreadPoolExecutor去检查线程是否存在，如果存在则取消当前任务执行，并把事务rollback。最后把该任务从doing队列中pop出，再重新push进pending队列。

5、在worker的某线程中，如果处理业务失败，则主动回滚，并把任务从doing队列中移除，重新push进pending队列。