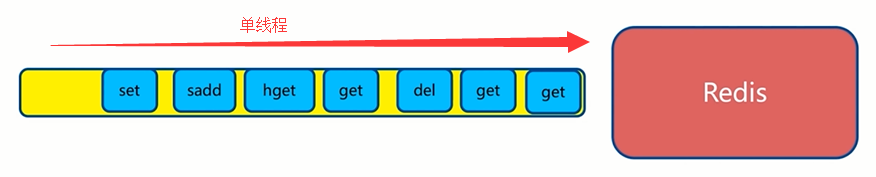
# Redis单线程模型

Redis是单线程的，在同一时刻只能处理一个请求！！



单线程为什么那么快？

1、纯内存（主要原因）

2、非阻塞IO

3、避免线程切换和竞态消耗

单线程需要注意的地方？

1、一次只能执行一条命令

2、拒绝长命令（耗时比较长），如keys ，flushdb等

# Redis数据类型

## Redis key

Redis的key是二进制安全的，这意味着你能够使用任意的二进制序列作为key，从一个简单字符串到一个JPEG图片的内容等，即使一个空的字符串也可以作为key。

作为一个redis的key最好遵循如下规则：

* key值不能过长，太长的key即消耗内存，也会增加key值查找的时间。如果key值实在过长，可以考虑以该key值的hash值代替。
* 太短的key值也是不可取的。尽量保证我们key的可读性。
* 尽量让我们的key值中包含一个有用的信息，如”object-type:id”类似这样的表示，“user:1000”。
* key值得最大是512MB

### 查看命令

* EXISTS key [key ...] 检查key是否存在
* KEYS pattern 查找所有符合给定模式pattern（正则表达式）的 key
* PTTL key 返回key剩余的过期时间（毫秒）
* TTL key返回key剩余的过期时间（秒）
* TYPE key 返回key所存储的value的数据结构类型，它可以返回string, list, set, zset 和 hash等不同的类型。
* SORT 对队列、集合、有序集合排序
* Dbsize 计算key的

### 删除命令

* DEL key [key ...] ：删除指定的一批keys，如果删除中的某些key不存在，则直接忽略，返回删除被删除key的数量。
* MIGRATE host port key destination-db timeout [COPY] [REPLACE]

将 key 原子性地从当前实例传送到目标实例的指定数据库上，一旦传送成功， key 保证会出现在目标实例上，而当前实例上的 key 会被删除。这个命令是一个原子操作，它在执行的时候会阻塞进行迁移的两个实例，直到以下任意结果发生：迁移成功，迁移失败，等到超时。

* RENAME key newkey 将key重命名为newkey，如果key与newkey相同，将返回一个错误。如果newkey已经存在，则值将被覆盖。
* RENAMENX key newkey 当且仅当 newkey 不存在时，将 key 改名为 newkey 。当 key 不存在时，返回一个错误。

### 设置命令

* + EXPIRE key seconds : 设置key的过期时间，超过时间后，将会自动删除该key
  + PEXPIRE key milliseconds : 设置key的过期时间（毫秒值）
  + PERSIST key 移除给定key的生存时间，将这个 key 从『易失的』(带生存时间 key )转换成『持久的』(一个不带生存时间、永不过期的 key )。
  + RANDOMKEY 从当前数据库返回一个随机的key

### 添加命令

* DUMP key ：序列化给定 key ，并返回被序列化的值
* Restore key ttl 序列化值 ：反序列化给定的序列化值，并将它和给定的 key 关联

## Redis Value

### Redis Strings

#### 查询指令

* GET key

返回key的value。如果key不存在，返回特殊值nil。如果key的value不是string，就返回错误，因为GET只处理string类型的values。

* GETRANGE key start end

返回key对应的字符串value的子串，这个子串是由start和end位移决定的（两者都在string内）。

* GETSET key value

自动将key对应到value，并且返回原来key对应的value。如果key存在但是对应的value不是字符串，就返回错误。

* MGET key [key ...]

返回所有指定的key的value。对于每个不对应string或者不存在的key，都返回特殊值nil。正因为此，这个操作从来不会失败。

* STRLEN key

返回key的string类型value的长度。如果key对应的非string类型，就返回错误。

#### 添加指令

* SET key value [EX seconds] [PX milliseconds] [NX|XX]

将键key设定为指定的“字符串”值。如果 key 已经保存了一个值，那么这个操作会直接覆盖原来的值，并且忽略原始类型。当set命令执行成功之后，之前设置的过期时间都将失效

 EX *seconds* – 设置键key的过期时间，单位时秒

 PX *milliseconds* – 设置键key的过期时间，单位时毫秒

 NX – 只有键key不存在的时候才会设置key的值 添加一个新的

 XX – 只有键key存在的时候才会设置key的值 修改老的

查看源文档有详细案例

* SETEX key seconds value

设置key对应字符串value，并且设置key在给定的seconds时间之后超时过期。SETEX是原子的，也可以通过把上面两个命令放到MULTI/EXEC块中执行的方式重现。相比连续执行上面两个命令，它更快，因为当Redis当做缓存使用时，这个操作更加常用。

* SETNX key value

将key设置值为value，如果key不存在，这种情况下等同SET命令。 当key存在时，什么也不做。SETNX是”SET if not exists”的简写。

* SETRANGE key offset value

这个命令的作用是覆盖key对应的string的一部分，从指定的offset处开始，覆盖value的长度

* MSET key value [key value ...]

对应给定的keys到他们相应的values上，MSET是原子的，所以所有给定的keys是一次性set的。

* MSETNX key value [key value ...]

对应给定的keys到他们相应的values上。只要有一个key已经存在，MSETNX一个操作都不会执行。 由于这种特性，MSETNX可以实现要么所有的操作都成功，要么一个都不执行，这样可以用来设置不同的key，来表示一个唯一的对象的不同字段。MSETNX是原子的，所以所有给定的keys是一次性set的

#### 修改指令

* APPEND key value

追加值到末尾，结果返回追加后字符串的长度

对数值型的字符串可以自增/自减（原子性）

* DECR key

对key对应的数字做减1操作。如果key不存在，那么在操作之前，这个key对应的值会被置为0。如果key有一个错误类型的value或者是一个不能表示成数字的字符串，就返回错误。这个操作最大支持在64位有符号的整型数字。

* DECRBY key decrement

将key对应的数字减decrement。如果key不存在，操作之前，key就会被置为0。如果key的value类型错误或者是个不能表示成数字的字符串，就返回错误。这个操作最多支持64位有符号的正型数字。

* INCR key

原文档中有详细的案例解释可以看看

* INCRBY key increment

### redis lists 列表(队列)

Redis lists基于Linked Lists实现。这意味着即使在一个list中有数百万个元素，在头部或尾部添加一个元素的操作，其时间复杂度也是常数级别的。用LPUSH 命令在十个元素的list头部添加新元素，和在千万元素list头部添加新元素的速度相同。

那么，坏消息是什么？在数组实现的list中利用索引访问元素的速度极快，而同样的操作在linked list实现的list上没有那么快。

Redis Lists用linked list实现的原因是：对于数据库系统来说，至关重要的特性是：能非常快的在很大的列表上添加元素。另一个重要因素是，正如你将要看到的：Redis lists能在常数时间取得常数长度。

如果快速访问集合元素很重要，建议使用可排序集合(sorted sets)。

list可被用来实现聊天系统。还可以作为不同进程间传递消息的队列。同时可以用来实现生产者和消费者模型

#### 查询指令

* LINDEX key index

返回列表里的元素的索引 index 存储在 key 里面。 下标是从0开始索引的。负数索引用于指定从列表尾部开始索引的元素

* LLEN key

返回存储在 key 里的list的长度。 如果 key 不存在，那么就被看作是空list，并且返回长度为 0。 当存储在 key 里的值不是一个list的话，会返回error。

* LRANGE key start stop

返回存储在 key 的列表里指定范围内的元素。 start 和 end 偏移量都是基于0的下标，即list的第一个元素下标是0（list的表头），第二个元素下标是1，以此类推。偏移量也可以是负数，表示偏移量是从list尾部开始计数。 例如， -1 表示列表的最后一个元素，-2 是倒数第二个，以此类推。

#### 添加指令

* LPUSH key value [value ...]

将所有指定的值插入列表的头部。如果 key 不存在，那么在进行 push 操作前会创建一个空列表。 如果 key 对应的值不是一个 list 的话，那么会返回一个错误。

* LPUSHX key value

只有当 key 已经存在并且存着一个 list 的时候，在这个 key 下面的 list 的头部插入 value。key 不存在的时候不会进行任何操作。

* RPUSH key value [value ...]

同LPUSH

* RPUSHX key value

同LPUSHX

* LINSERT key BEFORE|AFTER pivot value

把 value 插入存于 key 的列表中在基准值 pivot 的前面或后面。当 key 不存在时，这个list会被看作是空list，任何操作都不会发生。当 key 存在，但保存的不是一个list的时候，会返回error

* LSET key index value

设置 index 位置的list元素的值为 value。

#### 移除指令

* LPOP key

移除并且返回 key 对应的 list 的第一个元素。

* RPOP key

移除并且返回 key 对应的 list 的最后一个元素。

* RPOPLPUSH source destination

原子性地返回并移除存储在 source 的列表的最后一个元素（列表尾部元素）， 并把该元素放入存储在 destination 的列表的第一个元素位置（列表头部）。

* LREM key count value

从存于 key 的列表里移除前 count 次出现的值为 value 的元素。 这个 count 参数通过下面几种方式影响这个操作：

* + count > 0: 从头往尾移除值为 value 的元素。
  + count < 0: 从尾往头移除值为 value 的元素。
  + count = 0: 移除所有值为 value 的元素。
* LTRIM key start stop

修剪(trim)一个已存在的 list，这样 list 就会只包含指定范围的指定元素。

* BLPOP key [key ...] timeout

BLPOP 是阻塞式列表的弹出原语。它是命令 LPOP 的阻塞版本，这是因为当给定列表内没有任何元素可供弹出的时候， 连接将被 BLPOP 命令阻塞。 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头元素。

* BRPOP key [key ...] timeout

同BLPOP

* BRPOPLPUSH source destination timeout

BRPOPLPUSH 是 RPOPLPUSH 的阻塞版本。 当 source 包含元素的时候，这个命令表现得跟 RPOPLPUSH 一模一样。 当 source 是空的时候，Redis将会阻塞这个连接，直到另一个客户端 push 元素进入或者达到 timeout 时限。 timeout 为 0 能用于无限期阻塞客户端

当我们向一个聚合数据类型中添加元素时，如果目标键不存在，就在添加元素前创建空的聚合数据类型。当我们从聚合数据类型中移除元素时，如果值仍然是空的，键自动被销毁。对一个空的 key 调用一个只读的命令，比如 LLEN （返回 list 的长度），或者一个删除元素的命令， 将总是产生同样的结果。该结果和对一个空的聚合类型做同个操作的结果是一样的。

### Redis Hashes

Redis hash 看起来就像一个 “hash” 的样子，由键值对组成：Hash 便于表示 *objects*，实际上，你可以放入一个 hash 的域数量实际上没有限制（除了可用内存以外）。所以，你可以在你的应用中以不同的方式使用 hash。

#### 查询指令

* HEXISTS key field

返回hash里面field是否存在

* HGET key field

返回 key 指定的哈希集中该字段所关联的值，当key或field不存在时，返回nil

* HGETALL key

返回 key 指定的哈希集中所有的字段和值。返回值中，每个字段名的下一个是它的值，所以返回值的长度是哈希集大小的两倍

* HMGET key field [field ...]

返回 key 指定的哈希集中指定字段的值。对于哈希集中不存在的每个字段，返回 nil 值。因为不存在的keys被认为是一个空的哈希集，对一个不存在的 key 执行 HMGET 将返回一个只含有 nil 值的列表

* HKEYS key

获取hash的所有字段

* HVALS key

获取hash的所有值

* HLEN key

返回 key 指定的哈希集包含的字段的数量。

* HSTRLEN key field

返回hash指定field的value的字符串长度，如果hash或者field不存在，返回0.

#### 添加指令

* HSET key field value

设置 key 指定的哈希集中指定字段的值。如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。如果字段在哈希集中存在，它将被重写。

* HMSET key field value [field value ...]
* HSETNX key field value

只在 key 指定的哈希集中不存在指定的字段时，设置字段的值。如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。如果字段已存在，该操作无效果。

#### 删除指令

* HDEL key field [field ...]

从 key 指定的哈希集中移除指定的域。在哈希集中不存在的域将被忽略。如果 key 指定的哈希集不存在，它将被认为是一个空的哈希集，该命令将返回0。

* HINCRBY key field increment

增加 key 指定的哈希集中指定字段的数值。如果 key 不存在，会创建一个新的哈希集并与 key 关联。如果字段不存在，则字段的值在该操作执行前被设置为 0

* HINCRBYFLOAT key field increment

### Redis Sets

Redis Set 是 String 的无序排列。SADD 指令把新的元素添加到 set 中。对 set 也可做一些其他的操作，比如测试一个给定的元素是否存在，对不同 set 取交集，并集或差集等等。

#### 查询指令

* SCARD key

返回集合存储的key的基数 (集合元素的数量).

* SMEMBERS key

返回key集合所有的元素

* SRANDMEMBER key [count]

仅提供key参数,那么随机返回key集合中的一个元素.

Redis 2.6开始, 可以接受 count 参数,如果count是整数且小于元素的个数，返回含有 count 个不同的元素的数组,如果count是个整数且大于集合中元素的个数时,仅返回整个集合的所有元素,当count是负数,则会返回一个包含count的绝对值的个数元素的数组，如果count的绝对值大于元素的个数,则返回的结果集里会出现一个元素出现多次的情况.

#### 添加/删除指令

* SADD key member [member ...]

添加一个或多个指定的member元素到集合的 key中.指定的一个或者多个元素member 如果已经在集合key中存在则忽略.如果集合key 不存在，则新建集合key,并添加member元素到集合key中.

如果key 的类型不是集合则返回错误.

* SMOVE source destination member

将member从source集合移动到destination集合中. 如果source 集合不存在或者不包含指定的元素,这smove命令不执行任何操作并且返回0.否则对象将会从source集合中移除，并添加到destination集合中去，如果destination集合已经存在该元素，则smove命令仅将该元素充source集合中移除. 如果source 和destination不是集合类型,则返回错误.

* SPOP key [count]

从指定的集合对象中移除并且返回一个或者多个随意元素

* SREM key member [member ...]

在key集合中移除指定的元素. 如果指定的元素不是key集合中的元素则忽略 如果key集合不存在则被视为一个空的集合，该命令返回0.如果key的类型不是一个集合,则返回错误.

#### 交集、并集、差集操作指令

1. 求差集

* SDIFF key [key ...]

返回一个集合与给定集合的差集的元素.

* SDIFFSTORE destination key [key ...]

该命令类似于 [SDIFF](http://www.redis.cn/commands/sdiff.html), 不同之处在于该命令不返回结果集，而是将结果存放在destination集合中.

如果destination已经存在, 则将其覆盖重写.

1. 求交集

* SINTER key [key ...]

返回指定所有的集合的成员的交集.

* SINTERSTORE destination key [key ...]

这个命令与[SINTER](http://www.redis.cn/commands/sinter.html)命令类似, 但是它并不是直接返回结果集,而是将结果保存在 destination集合中.

如果destination 集合存在, 则会被重写.

1. 求并集

* SUNION key [key ...]

返回给定的多个集合的并集中的所有成员.

* SUNIONSTORE destination key [key ...]

该命令作用类似于[SUNION](http://www.redis.cn/commands/sunion.html)命令,不同的是它并不返回结果集,而是将结果存储在destination集合中.

如果destination 已经存在,则将其覆盖.

### Redis Sorted sets(ZSET)

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。有序集合的成员是唯一的,但分数(score)却可以重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。 集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

#### 查询指令

* ZCARD key

返回key的有序集元素个数。

* ZCOUNT key min max

返回有序集key中，score值在min和max之间(默认包括score值等于min或max)的成员。

* ZLEXCOUNT key min max

ZLEXCOUNT 命令用于计算有序集合中指定成员之间的成员数量。

* ZRANGE key start stop [WITHSCORES]

返回指定index区间的集合元素

* ZREVRANGE key start stop [WITHSCORES]

返回有序集key中，指定区间内的成员。其中成员的位置按score值递减(从大到小)来排列。具有相同score值的成员按字典序的反序排列。

* ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

返回key的有序集合中的分数在min和max之间的所有元素（包括分数等于max或者min的元素）。元素被认为是从低分到高分排序的。

具有相同分数的元素按字典序排列（这个根据redis对有序集合实现的情况而定，并不需要进一步计算）。

可选的LIMIT参数指定返回结果的数量及区间（类似SQL中SELECT LIMIT offset, count）。注意，如果offset太大，定位offset就可能遍历整个有序集合，这会增加O(N)的复杂度。

可选参数WITHSCORES会返回元素和其分数，而不只是元素。这个选项在redis2.0之后的版本都可用。

* ZREVRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

ZREVRANGEBYSCORE 返回有序集合中指定分数区间内的成员，分数由高到低排序。

* ZRANK key member

返回有序集key中成员member的排名。其中有序集成员按score值递增(从小到大)顺序排列。排名以0为底，也就是说，score值最小的成员排名为0。

* ZREVRANK  key member

获得成员按score值递减(从大到小)排列的排名。

* ZSCORE key member

返回有序集key中，成员member的score值。

#### 添加/删除指令

* ZADD key [NX|XX] [CH] [INCR] score member [score member ...]

将所有指定成员添加到键为key有序集合（sorted set）里面。 添加时可以指定多个分数/成员（score/member）对。 如果指定添加的成员已经是有序集合里面的成员，则会更新改成员的分数（scrore）并更新到正确的排序位置。

如果key不存在，将会创建一个新的有序集合（sorted set）并将分数/成员（score/member）对添加到有序集合，就像原来存在一个空的有序集合一样。如果key存在，但是类型不是有序集合，将会返回一个错误应答。

分数值是一个双精度的浮点型数字字符串。+inf和-inf都是有效值。

* XX: 仅仅更新存在的成员，不添加新成员。
* NX: 不更新存在的成员。只添加新成员。
* CH: 修改返回值为发生变化的成员总数，原始是返回新添加成员的总数 (CH 是 changed 的意思)。更改的元素是新添加的成员，已经存在的成员更新分数。 所以在命令中指定的成员有相同的分数将不被计算在内。注：在通常情况下，ZADD返回值只计算新添加成员的数量。
* INCR: 当ZADD指定这个选项时，成员的操作就等同ZINCRBY命令，对成员的分数进行递增操作。
* ZINCRBY key increment member

为有序集key的成员member的score值加上增量increment。

* ZREM key member [member ...]

删除有序集合中一个或多个元素

* ZREMRANGEBYLEX key min max

ZREMRANGEBYLEX 删除名称按字典由低到高排序成员之间所有成员。  
不要在成员分数不同的有序集合中使用此命令, 因为它是基于分数一致的有序集合设计的,如果使用,会导致删除的结果不正确。  
待删除的有序集合中,分数最好相同,否则删除结果会不正常。

* ZREMRANGEBYRANK key start stop

移除有序集key中，指定排名(rank)区间内的所有成员。下标参数start和stop都以0为底，0处是分数最小的那个元素。这些索引也可是负数，表示位移从最高分处开始数。例如，-1是分数最高的元素，-2是分数第二高的，依次类推。

* ZREMRANGEBYSCORE key min max

移除有序集key中，所有score值介于min和max之间(包括等于min或max)的成员。

#### 交集、并集指令

* ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight] [SUM|MIN|MAX]

计算给定的numkeys个有序集合的并集，并且把结果放到destination中。在给定要计算的key和其它参数之前，必须先给定key个数(numberkeys)。 默认情况下，结果集中某个成员的score值是所有给定集下该成员score值之和。

使用WEIGHTS选项，你可以为每个给定的有序集指定一个乘法因子，意思就是，每个给定有序集的所有成员的score值在传递给聚合函数之前都要先乘以该因子。如果WEIGHTS没有给定，默认就是1。

使用AGGREGATE选项，你可以指定并集的结果集的聚合方式。默认使用的参数SUM，可以将所有集合中某个成员的score值之和作为结果集中该成员的score值。如果使用参数MIN或者MAX，结果集就是所有集合中元素最小或最大的元素。

如果key destination存在，就被覆盖。

* ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight] [SUM|MIN|MAX]

计算给定的numkeys个有序集合的交集，并且把结果放到destination中。 在给定要计算的key和其它参数之前，必须先给定key个数(numberkeys)。

默认情况下，结果中一个元素的分数是有序集合中该元素分数之和，前提是该元素在这些有序集合中都存在。因为交集要求其成员必须是给定的每个有序集合中的成员，结果集中的每个元素的分数和输入的有序集合个数相等。

对于WEIGHTS和AGGREGATE参数的描述，参见命令[ZUNIONSTORE](http://www.redis.cn/commands/zunionstore.html)。

如果destination存在，就把它覆盖。

### Bitmaps

Bitmaps并不是实际的数据类型，而是定义在String类型上的一个面向字节操作的集合。因为字符串是二进制安全的块，他们的最大长度是512M，最适合设置成2^32个不同字节。

字节操作可以分为两类：恒定时间的单个字节操作如将一个字节设置为1或者0、或者获取一个字节的值，和批量字节操作如在给定的字节范围内计算设置字节的数量。

bitmaps 的最大优势之一在存储信息时极其节约空间。例如，在一个以增量用户ID来标识不同用户的系统中，记录四十亿用户的一个单独bit信息（例如，要知道用户是否想要接收最新的来信）仅仅使用512M内存。

* setbit key offset value

对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)。

* getbit key offset

对 key 所储存的字符串值，获取指定偏移量上的位(bit)。

* bitcount key start end

获取key所存储字符串的bit为1的个数

* BITPOS key bit

找到key存储的字符串的bit中第一次出现0或1的位置

* BITOP operation destkey key [key ...]

对一个或多个保存二进制位的字符串 key 进行位元操作，并将结果保存到 destkey 上。

BITOP 命令支持 AND 、 OR 、 NOT 、 XOR 这四种操作中的任意一种参数：

* BITOP AND destkey srckey1 srckey2 srckey3 ... srckeyN ，对一个或多个 key 求逻辑并，并将结果保存到 destkey 。
* BITOP OR destkey srckey1 srckey2 srckey3 ... srckeyN，对一个或多个 key 求逻辑或，并将结果保存到 destkey 。
* BITOP XOR destkey srckey1 srckey2 srckey3 ... srckeyN，对一个或多个 key 求逻辑异或，并将结果保存到 destkey 。
* BITOP NOT destkey srckey，对给定 key 求逻辑非，并将结果保存到 destkey 。

除了 NOT 操作之外，其他操作都可以接受一个或多个 key 作为输入。

执行结果将始终保持到destkey里面。

### HyperLogLogs

Redis 在 2.8.9 版本添加了 HyperLogLog 结构。

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的。

在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。

但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

* pfadd

Redis Pfadd 命令将所有元素参数添加到 HyperLogLog 数据结构中。

* pfcount

Redis Pfcount 命令返回给定 HyperLogLog 的基数估算值。

* perge

Redis Pgmerge 命令将多个 HyperLogLog 合并为一个 HyperLogLog ，合并后的 HyperLogLog 的基数估算值是通过对所有 给定 HyperLogLog 进行并集计算得出的。

### 总结

|  |  |  |
| --- | --- | --- |
| 数据类型 | 可以存储的值 | 操作 |
| STRING | 字符串、整数或浮点数 | 对整个字符串或字符串的一部分进行操作、对整数或者浮点数进行自增或自减操作 |
| LIST | 链表 | 从两端压入或者弹出数据、读取单个或者多个元素、进行修剪，只保留一个范围内的元素 |
| SET | 无序集合 | 添加、获取、移除单个元素、检查一个元素是否存在于集合中、计算集合的交集并集差集、从集合元素中随机获取元素 |
| HASH | 包含键值对的无序列表 | 添加、获取、移除单个键值对、获取所有键值对、检查某个键是否存在 |
| ZSET | 有序集合 | 添加、获取、删除元素，根据分值范围或者成员来获取元素，计算一个键的排名 |

Redis五种数据类型在一般互联网公司的应用场景：

* String：缓存、限流、计数器（incr、decr）、分布式锁、分布式Session
* Hash：存储用户信息、用户主页访问量、组合查询
* List：微博关注人时间轴列表、简单队列
* Set：赞、踩、标签、好友关系
* Zset：排行榜（有序）

# Redis过期时间

## 设置过期时间

* EXPIRE key seconds : 设置key的过期时间，超过时间后，将会自动删除该key
* PEXPIRE key milliseconds : 设置key的过期时间（毫秒值）

## 查看过期时间

* PTTL key 返回key剩余的过期时间（毫秒）。
* TTL key返回key剩余的过期时间（秒）。

## 刷新过期时间

* 对已经有过期时间的key执行EXPIRE操作，将会更新它的过期时间。

## 持久化key

* + PERSIST key 移除给定key的生存时间，将这个 key 从『易失的』(带生存时间 key )转换成『持久的』(一个不带生存时间、永不过期的 key )。

## 过期时间精度

* 在 Redis 2.4 及以前版本，过期期时间可能不是十分准确，有0-1秒的误差。
* 从 Redis 2.6 起，过期时间误差缩小到0-1毫秒。

## Redis如何淘汰过期的keys

Redis keys过期有两种方式：被动和主动方式。

当一些客户端尝试访问它时，key会被发现并主动的过期。

当然，这样是不够的，因为有些过期的keys，永远不会访问他们。 无论如何，这些keys应该过期，所以定时随机测试设置keys的过期时间。所有这些过期的keys将会从内存空间删除。

具体就是Redis每秒10次做的事情：

* + 测试随机的20个keys进行相关过期检测。
  + 删除所有已经过期的keys。
  + 如果有多于25%的keys过期，重复步奏1。

这是一个平凡的概率算法，基本上的假设是，我们的样本是这个密钥控件，并且我们不断重复过期检测，直到过期的keys的百分百低于25%,这意味着，在任何给定的时刻，最多会清除1/4的过期keys。

# LRU事件驱动(内存管理与数据淘汰机制)

## 如何用Redis做LRU-Cache

LRU(Least Recently Used)最近最少使用算法是众多置换算法中的一种。   
Redis中有一个maxmemory概念，主要是为了将使用的内存限定在一个固定的大小。Redis用到的LRU 算法，是一种近似的LRU算法。

### 设置maxmemory（可用最大内存！！ ）

上面已经说过maxmemory是为了限定Redis最大内存使用量。有多种方法设定它的大小。

* 其中一种方法是通过CONFIG SET指令设定，如下：

127.0.0.1:6379> CONFIG GET maxmemory

1) "maxmemory"

2) "0"

127.0.0.1:6379> CONFIG SET maxmemory 100MB

OK

127.0.0.1:6379> CONFIG GET maxmemory

1) "maxmemory"

2) "104857600"

* 另一种方法是修改配置文件redis.conf：

maxmemory 100mb

注意，在64bit系统下，maxmemory设置为0表示不限制Redis内存使用，在32bit系统下，maxmemory隐式不能超过3GB。   
当Redis内存使用达到指定的限制时，就需要选择一个置换的策略。

### 置换策略

当Redis内存使用达到maxmemory时，需要选择设置好的maxmemory-policy进行对老数据的置换。   
下面是可以选择的置换策略：

* noeviction: 不进行置换，表示即使内存达到上限也不进行置换，所有能引起内存增加的命令都会返回error
* allkeys-lru: 优先删除掉最近最不经常使用的key，用以保存新数据
* volatile-lru: 只从设置失效（expire set）的key中选择最近最不经常使用的key进行删除，用以保存新数据
* allkeys-random: 随机从all-keys中选择一些key进行删除，用以保存新数据
* volatile-random: 只从设置失效（expire set）的key中，选择一些key进行删除，用以保存新数据
* volatile-ttl: 只从设置失效（expire set）的key中，选出存活时间（TTL）最短的key进行删除，用以保存新数据

设置maxmemory-policy的方法和设置maxmemory方法类似，通过redis.conf或是通过CONFIG SET动态修改。

如果没有匹配到可以删除的key，那么volatile-lru、volatile-random和volatile-ttl策略和noeviction替换策略一样——不对任何key进行置换。

选择合适的置换策略是很重要的，这主要取决于你的应用的访问模式，当然你也可以动态的修改置换策略，并通过用Redis命令——INFO去输出cache的命中率情况，进而可以对置换策略进行调优。

一般来说，有这样一些常用的经验：

* 在所有的key都是最近最经常使用，那么就需要选择allkeys-lru进行置换最近最不经常使用的key，如果你不确定使用哪种策略，那么推荐使用allkeys-lru
* 如果所有的key的访问概率都是差不多的，那么可以选用allkeys-random策略去置换数据
* 如果对数据有足够的了解，能够为key指定hint（通过expire/ttl指定），那么可以选择volatile-ttl进行置换

volatile-lru和volatile-random经常在一个Redis实例既做cache又做持久化的情况下用到，然而，更好的选择使用两个Redis实例来解决这个问题。

设置失效时间expire会占用一些内存，而采用allkeys-lru就没有必要设置失效时间，进而更有效的利用内存。

### 置换策略是如何工作的

理解置换策略的执行方式是非常重要的，比如：

* 客户端执行一条新命令，导致数据库需要增加数据（比如set key value）
* Redis会检查内存使用，如果内存使用超过maxmemory，就会按照置换策略删除一些key
* 新的命令执行成功

我们持续的写数据会导致内存达到或超出上限maxmemory，但是置换策略会将内存使用降低到上限以下。

如果一次需要使用很多的内存（比如一次写入一个很大的set），那么，Redis的内存使用可能超出最大内存限制一段时间。

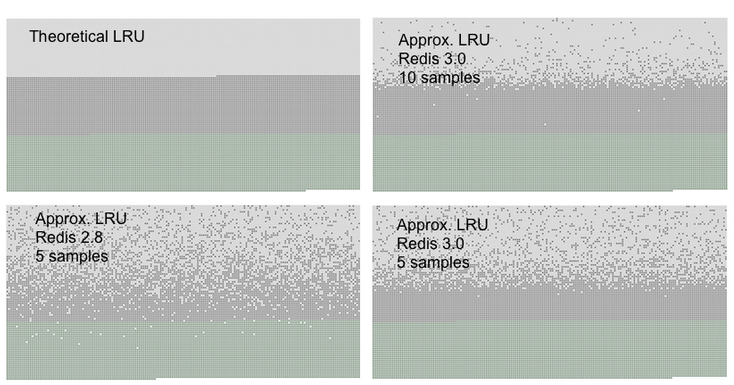
### 近似LRU算法

Redis中的LRU不是严格意义上的LRU算法实现，是一种近似的LRU实现，主要是为了节约内存占用以及提升性能。Redis有这样一个配置——maxmemory-samples，Redis的LRU是取出配置的数目的key，然后从中选择一个最近最不经常使用的key进行置换，默认的5，如下：

maxmemory-samples 5

可以通过调整样本数量来取得LRU置换算法的速度或是精确性方面的优势。

Redis不采用真正的LRU实现的原因是为了节约内存使用。虽然不是真正的LRU实现，但是它们在应用上几乎是等价的。下图是Redis的近似LRU实现和理论LRU实现的对比：



测试开始首先在Redis中导入一定数目的key，然后从第一个key依次访问到最后一个key，因此根据LRU算法第一个被访问的key应该最新被置换，之后再增加50%数目的key，导致50%的老的key被替换出去。   
在上图中你可以看到三种类型的点，组成三种不同的区域：

* 淡灰色的是被置换出去的key
* 灰色的是没有被置换出去的key
* 绿色的是新增加的key

理论LRU实现就像我们期待的那样，最旧的50%数目的key被置换出去，Redis的LRU将一定比例的旧key置换出去。

可以看到在样本数为5的情况下，Redis3.0要比Redis2.8做的好很多，Redis2.8中有很多应该被置换出去的数据没有置换出去。在样本数为10的情况下，Redis3.0很接近真正的LRU实现。

LRU是一个预测未来我们会访问哪些数据的模型，如果我们访问数据的形式接近我们预想——幂律，那么近似LRU算法实现将能处理的很好。

在模拟测试中我们可以发现，在幂律访问模式下，理论LRU和Redis近似LRU的差距很小或者就不存在差距。

如果你将maxmemory-samples设置为10，那么Redis将会增加额外的CPU开销以保证接近真正的LRU性能，可以通过检查命中率来查看有什么不同。

通过CONFIG SET maxmemory-samples <count>动态调整样本数大小，做一些测试验证你的猜想。

# Redis 持久化（很重要！！）

## Redis 持久化方式

### RDB快照

#### 概述

在默认情况下， Redis 将数据库快照保存在名字为 dump.rdb的二进制文件中。你可以对 Redis 进行设置， 让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。你也可以通过调用 SAVE或者 BGSAVE ， 手动让 Redis 进行数据集保存操作。

比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集:

save 60 1000

这种持久化方式被称为快照 snapshotting.

#### 何时执行快照

出现下面的情况redis会快照内存里的数据

* 1 用户发送bgsave命令(此时redis会fork一个子进程,子进程负责生成硬盘文件,父进程负责继续接受命令)
* 2 用户发送save命令(和bgsave命令不同,发送save命令后,到系统创建快照完成之前系统不会再接收新的命令,换句话说save命令会阻塞后面的命令,而bgsave不会)
* 3 用户在配置文件了配置了类似这样的命令
  + save 60 1000

这个的意思是说,自从上次快照成功算起,如果满足"60秒内有1000次写入"这个条件,系统就自动调用bgsave,如果配置文件里有多个save命令,只有满足一个就调用bgsave命令

* 4 用户发送shutdown,系统会先执行save命令阻塞客户端,然后关闭服务器
* 5 当有主从架构时,从服务器向主服务器发送sync命令来执行复制操作时,只有主服务器当时没有进行bgsave操作,那么主服务器就会执行bgsave操作。

#### 快照的配置信息

* save 60 1000
* stop-writes-on-bgsave-error no
* rdbcompression yes
* dbfilename dump.rdb
* dir ./

#### 工作方式

当 Redis 需要保存 dump.rdb 文件时， 服务器执行以下操作:

* Redis 调用forks. 同时拥有父进程和子进程。
* 子进程将数据集写入到一个临时 RDB 文件中。
* 当子进程完成对新 RDB 文件的写入时，Redis 用新 RDB 文件替换原来的 RDB 文件，并删除旧的 RDB 文件。

这种工作方式使得 Redis 可以从写时复制（copy-on-write）机制中获益。

### 只追加操作的文件（Append-only file，AOF）

#### 概述

快照功能并不是非常耐久（dura ble）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。 从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化。

你可以在配置文件中打开AOF方式:

appendonly yes

从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 SET）， 这个命令就会被追加到 AOF 文件的末尾。这样的话， 当 Redis 重新启时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

#### 日志重写

因为 AOF 的运作方式是不断地将命令追加到文件的末尾， 所以随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。举个例子， 如果你对一个计数器调用了 100 次 INCR ， 那么仅仅是为了保存这个计数器的当前值， AOF 文件就需要使用 100 条记录（entry）。然而在实际上， 只使用一条 SET 命令已经足以保存计数器的当前值了， 其余 99 条记录实际上都是多余的。

为了处理这种情况， Redis 支持一种有趣的特性： 可以在不打断服务客户端的情况下， 对 AOF 文件进行重建（rebuild）。执行 BGREWRITEAOF 命令， Redis 将生成一个新的 AOF 文件， 这个文件包含重建当前数据集所需的最少命令。Redis 2.2 需要自己手动执行 BGREWRITEAOF 命令； Redis 2.4 则可以自动触发 AOF 重写， 具体信息请查看 2.4 的示例配置文件。

#### AOF有多耐用

你可以配置 Redis 多久才将数据 fsync 到磁盘一次。有三种方式：

* 每次有新命令追加到 AOF 文件时就执行一次 fsync ：非常慢，也非常安全
* 每秒 fsync 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。
* 从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择。
* 推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

#### 如果AOF文件损坏了怎么办？

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：

1. 为现有的 AOF 文件创建一个备份。
2. 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

1. （可选）使用 diff -u 对比修复后的 AOF 文件和原始 AOF 文件的备份，查看两个文件之间的不同之处。
2. 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。

#### AOP重写原理

AOF 重写和 RDB 创建快照一样，都巧妙地利用了写时复制机制:

* redis 执行 fork() ，现在同时拥有父进程和子进程。
* 子进程开始将新 AOF 文件的内容写入到临时文件。
* 对于所有新执行的写入命令，父进程一边将它们累积到一个内存缓存中，一边将这些改动追加到现有 AOF 文件的末尾,这样样即使在重写的中途发生停机，现有的 AOF 文件也还是安全的。
* 当子进程完成重写工作时，它给父进程发送一个信号，父进程在接收到信号之后，将内存缓存中的所有数据追加到新 AOF 文件的末尾。
* 搞定！现在 Redis 原子地用新文件替换旧文件，之后所有命令都会直接追加到新 AOF 文件的末尾。

### 怎样从RDB方式切换为AOF方式

在 Redis 2.2 或以上版本，可以在不重启的情况下，从 RDB 切换到 AOF ：

* 为最新的 dump.rdb 文件创建一个备份。
* 将备份放到一个安全的地方。
* 执行以下两条命令:
* redis-cli config set appendonly yes
* redis-cli config set save “”
* 确保写命令会被正确地追加到 AOF 文件的末尾。
* 执行的第一条命令开启了 AOF 功能： Redis 会阻塞直到初始 AOF 文件创建完成为止， 之后 Redis 会继续处理命令请求， 并开始将写入命令追加到 AOF 文件末尾。

执行的第二条命令用于关闭 RDB 功能。 这一步是可选的， 如果你愿意的话， 也可以同时使用 RDB 和 AOF 这两种持久化功能。

重要:别忘了在 redis.conf 中打开 AOF 功能！ 否则的话， 服务器重启之后， 之前通过 CONFIG SET 设置的配置就会被遗忘， 程序会按原来的配置来启动服务器。

### AOF和RDB之间的相互作用

在版本号大于等于 2.4 的 Redis 中， BGSAVE 执行的过程中， 不可以执行 BGREWRITEAOF 。 反过来说， 在 BGREWRITEAOF 执行的过程中， 也不可以执行 BGSAVE。这可以防止两个 Redis 后台进程同时对磁盘进行大量的 I/O 操作。

如果 BGSAVE 正在执行， 并且用户显示地调用 BGREWRITEAOF 命令， 那么服务器将向用户回复一个 OK 状态， 并告知用户， BGREWRITEAOF 已经被预定执行： 一旦 BGSAVE 执行完毕， BGREWRITEAOF 就会正式开始。 当 Redis 启动时， 如果 RDB 持久化和 AOF 持久化都被打开了， 那么程序会优先使用 AOF 文件来恢复数据集， 因为 AOF 文件所保存的数据通常是最完整的。

### AOF VS RDB

* RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储.
* AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大.
* 如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.
* 你也可以同时开启两种持久化方式, 在这种情况下, 当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.
* 最重要的事情是了解RDB和AOF持久化方式的不同,让我们以RDB持久化方式开始:

#### **RDB的优缺点**

##### 优点

* RDB是一个非常紧凑的文件,它保存了某个时间点得数据集,非常适用于数据集的备份,比如你可以在每个小时报保存一下过去24小时内的数据,同时每天保存过去30天的数据,这样即使出了问题你也可以根据需求恢复到不同版本的数据集.
* RDB是一个紧凑的单一文件,很方便传送到另一个远端数据中心或者亚马逊的S3（可能加密），非常适用于灾难恢复.
* RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他IO操作，所以RDB持久化方式可以最大化redis的性能.
* 与AOF相比,在恢复大的数据集的时候，RDB方式会更快一些.

##### 缺点

* 如果你希望在redis意外停止工作（例如电源中断）的情况下丢失的数据最少的话，那么RDB不适合你.虽然你可以配置不同的save时间点(例如每隔5分钟并且对数据集有100个写的操作),是Redis要完整的保存整个数据集是一个比较繁重的工作,你通常会每隔5分钟或者更久做一次完整的保存,万一在Redis意外宕机,你可能会丢失几分钟的数据.
* RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级内不能响应客户端的请求.如果数据集巨大并且CPU性能不是很好的情况下,这种情况会持续1秒,AOF也需要fork,但是你可以调节重写日志文件的频率来提高数据集的耐久度.

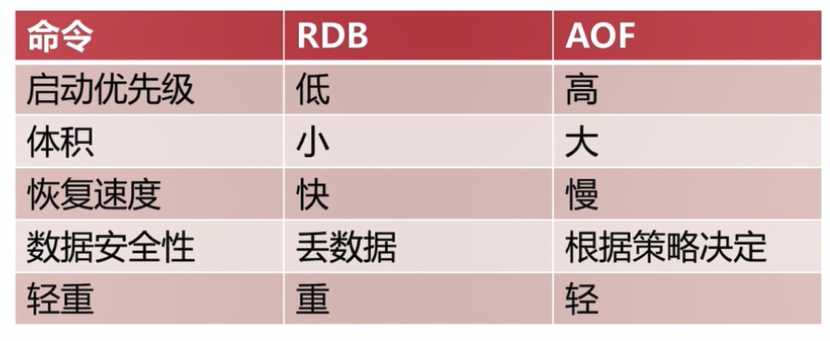
#### **AOF 优缺点**

##### 优点

* 使用AOF 会让你的Redis更加耐久: 你可以使用不同的fsync策略：无fsync,每秒fsync,每次写的时候fsync.使用默认的每秒fsync策略,Redis的性能依然很好(fsync是由后台线程进行处理的,主线程会尽力处理客户端请求),一旦出现故障，你最多丢失1秒的数据.
* AOF文件是一个只进行追加的日志文件,所以不需要写入seek,即使由于某些原因(磁盘空间已满，写的过程中宕机等等)未执行完整的写入命令,你也也可使用redis-check-aof工具修复这些问题.
* 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。 整个重写操作是绝对安全的，因为 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。
* 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。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

##### 缺点

* 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
* 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。



## 备份redis数据

在阅读这个小节前， 请牢记下面这句话: 确保你的数据由完整的备份. 磁盘故障， 节点失效， 诸如此类的问题都可能让你的数据消失不见， 不进行备份是非常危险的。

Redis 对于数据备份是非常友好的， 因为你可以在服务器运行的时候对 RDB 文件进行复制： RDB 文件一旦被创建， 就不会进行任何修改。 当服务器要创建一个新的 RDB 文件时， 它先将文件的内容保存在一个临时文件里面， 当临时文件写入完毕时， 程序才使用 rename(2) 原子地用临时文件替换原来的 RDB 文件。

这也就是说， 无论何时， 复制 RDB 文件都是绝对安全的。

* 创建一个定期任务（cron job）， 每小时将一个 RDB 文件备份到一个文件夹， 并且每天将一个 RDB 文件备份到另一个文件夹。
* 确保快照的备份都带有相应的日期和时间信息， 每次执行定期任务脚本时， 使用 find 命令来删除过期的快照： 比如说， 你可以保留最近 48 小时内的每小时快照， 还可以保留最近一两个月的每日快照。
* 至少每天一次， 将 RDB 备份到你的数据中心之外， 或者至少是备份到你运行 Redis 服务器的物理机器之外。
* #! /bin/bash
* PATH=/usr/local/bin:$PATH
* redis-cli SAVE
* date=$(date +"%Y%m%d")
* cp /var/lib/redis/6379/dump.rdb /data01/cache\_backup/$date.rdb
* echo "done!"

## 容灾备份

Redis 的容灾备份基本上就是对数据进行备份， 并将这些备份传送到多个不同的外部数据中心。容灾备份可以在 Redis 运行并产生快照的主数据中心发生严重的问题时， 仍然让数据处于安全状态。

因为很多 Redis 用户都是创业者， 他们没有大把大把的钱可以浪费， 所以下面介绍的都是一些实用又便宜的容灾备份方法：

* Amazon S3 ，以及其他类似 S3 的服务，是一个构建灾难备份系统的好地方。 最简单的方法就是将你的每小时或者每日 RDB 备份加密并传送到 S3 。 对数据的加密可以通过 gpg -c 命令来完成（对称加密模式）。 记得把你的密码放到几个不同的、安全的地方去（比如你可以把密码复制给你组织里最重要的人物）。 同时使用多个储存服务来保存数据文件，可以提升数据的安全性。
* 传送快照可以使用 SCP 来完成（SSH 的组件）。 以下是简单并且安全的传送方法： 买一个离你的数据中心非常远的 VPS ， 装上 SSH ， 创建一个无口令的 SSH 客户端 key ， 并将这个 key 添加到 VPS 的 authorized\_keys 文件中， 这样就可以向这个 VPS 传送快照备份文件了。 为了达到最好的数据安全性，至少要从两个不同的提供商那里各购买一个 VPS 来进行数据容灾备份。
* 需要注意的是， 这类容灾系统如果没有小心地进行处理的话， 是很容易失效的。最低限度下， 你应该在文件传送完毕之后， 检查所传送备份文件的体积和原始快照文件的体积是否相同。 如果你使用的是 VPS ， 那么还可以通过比对文件的 SHA1 校验和来确认文件是否传送完整。

另外， 你还需要一个独立的警报系统， 让它在负责传送备份文件的传送器（transfer）失灵时通知你。

# Redis事务

https://blog.csdn.net/liuhanfanshuang/article/details/52136438

## 事务概述

[MULTI](http://www.redis.cn/commands/multi.html) 、 [EXEC](http://www.redis.cn/commands/exec.html) 、 [DISCARD](http://www.redis.cn/commands/discard.html) 和 [WATCH](http://www.redis.cn/commands/watch.html) 是 Redis 事务相关的命令。事务可以一次执行多个命令， 并且带有以下两个重要的保证：

* 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
* 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

[EXEC](http://www.redis.cn/commands/exec.html) 命令负责触发并执行事务中的所有命令：

* 如果客户端在使用 [MULTI](http://www.redis.cn/commands/multi.html) 开启了一个事务之后，却因为断线而没有成功执行 [EXEC](http://www.redis.cn/commands/exec.html) ，那么事务中的所有命令都不会被执行。
* 另一方面，如果客户端成功在开启事务之后执行 [EXEC](http://www.redis.cn/commands/exec.html) ，那么事务中的所有命令都会被执行。

当使用 AOF 方式做持久化的时候， Redis 会使用单个 write() 命令将事务写入到磁盘中。

然而，如果 Redis 服务器因为某些原因被管理员杀死，或者遇上某种硬件故障，那么可能只有部分事务命令会被成功写入到磁盘中。

如果 Redis 在重新启动时发现 AOF 文件出了这样的问题，那么它会退出，并汇报一个错误。

使用redis-check-aof程序可以修复这一问题：它会移除 AOF 文件中不完整事务的信息，确保服务器可以顺利启动。

从 2.2 版本开始，Redis 还可以通过乐观锁（optimistic lock）实现 CAS （check-and-set）操作，具体信息请参考文档的后半部分。

## 用法

[MULTI](http://www.redis.cn/commands/multi.html) 命令用于开启一个事务，它总是返回 OK 。 [MULTI](http://www.redis.cn/commands/multi.html) 执行之后， 客户端可以继续向服务器发送任意多条命令， 这些命令不会立即被执行， 而是被放到一个队列中， 当 [EXEC](http://www.redis.cn/commands/exec.html)命令被调用时， 所有队列中的命令才会被执行。

另一方面， 通过调用 [DISCARD](http://www.redis.cn/commands/discard.html) ， 客户端可以清空事务队列， 并放弃执行事务。

以下是一个事务例子， 它原子地增加了 foo 和 bar 两个键的值：

> MULTI

OK

> INCR foo

QUEUED

> INCR bar

QUEUED

> EXEC

1) (integer) 1

2) (integer) 1

[EXEC](http://www.redis.cn/commands/exec.html) 命令的回复是一个数组， 数组中的每个元素都是执行事务中的命令所产生的回复。 其中， 回复元素的先后顺序和命令发送的先后顺序一致。

当客户端处于事务状态时， 所有传入的命令都会返回一个内容为 QUEUED 的状态回复（status reply）， 这些被入队的命令将在 EXEC 命令被调用时执行。

## 事务中的错误

使用事务时可能会遇上以下两种错误：

* 事务在执行 [EXEC](http://www.redis.cn/commands/exec.html) 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）。
* 命令可能在 [EXEC](http://www.redis.cn/commands/exec.html) 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。

对于发生在 [EXEC](http://www.redis.cn/commands/exec.html) 执行之前的错误，客户端以前的做法是检查命令入队所得的返回值：如果命令入队时返回 QUEUED ，那么入队成功；否则，就是入队失败。如果有命令在入队时失败，那么大部分客户端都会停止并取消这个事务。

不过，从 Redis 2.6.5 开始，服务器会对命令入队失败的情况进行记录，并在客户端调用 [EXEC](http://www.redis.cn/commands/exec.html) 命令时，拒绝执行并自动放弃这个事务。

在 Redis 2.6.5 以前， Redis 只执行事务中那些入队成功的命令，而忽略那些入队失败的命令。 而新的处理方式则使得在流水线（pipeline）中包含事务变得简单，因为发送事务和读取事务的回复都只需要和服务器进行一次通讯。

至于那些在 [EXEC](http://www.redis.cn/commands/exec.html) 命令执行之后所产生的错误， 并没有对它们进行特别处理： 即使事务中有某个/某些命令在执行时产生了错误， 事务中的其他命令仍然会继续执行。

从协议的角度来看这个问题，会更容易理解一些。 以下例子中， [LPOP](http://www.redis.cn/commands/lpop.html) 命令的执行将出错， 尽管调用它的语法是正确的：

MULTI

+OK

SET a 3

abc

+QUEUED

LPOP a

+QUEUED

EXEC

\*2

+OK

-ERR Operation against a key holding the wrong kind of value

[EXEC](http://www.redis.cn/commands/exec.html) 返回两条[bulk-string-reply](http://www.redis.cn/topics/protocol.html" \l "bulk-string-reply)： 第一条是 OK ，而第二条是 -ERR 。 至于怎样用合适的方法来表示事务中的错误， 则是由客户端自己决定的。

最重要的是记住这样一条， 即使事务中有某条/某些命令执行失败了， 事务队列中的其他命令仍然会继续执行 —— Redis 不会停止执行事务中的命令。

以下例子展示的是另一种情况， 当命令在入队时产生错误， 错误会立即被返回给客户端：

MULTI

+OK

INCR a b c

-ERR wrong number of arguments for 'incr' command

因为调用 [INCR](http://www.redis.cn/commands/incr.html) 命令的参数格式不正确， 所以这个 [INCR](http://www.redis.cn/commands/incr.html) 命令入队失败。

## 为什么 Redis 不支持回滚（roll back）

如果你有使用关系式数据库的经验， 那么 “Redis 在事务失败时不进行回滚，而是继续执行余下的命令”这种做法可能会让你觉得有点奇怪。

以下是这种做法的优点：

* Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中。
* 因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速。

有种观点认为 Redis 处理事务的做法会产生 bug ， 然而需要注意的是， 在通常情况下， 回滚并不能解决编程错误带来的问题。 举个例子， 如果你本来想通过 [INCR](http://www.redis.cn/commands/incr.html) 命令将键的值加上 1 ， 却不小心加上了 2 ， 又或者对错误类型的键执行了 [INCR](http://www.redis.cn/commands/incr.html) ， 回滚是没有办法处理这些情况的。

## 放弃事务

当执行 [DISCARD](http://www.redis.cn/commands/discard.html) 命令时， 事务会被放弃， 事务队列会被清空， 并且客户端会从事务状态中退出：

> SET foo 1

OK

> MULTI

OK

> INCR foo

QUEUED

> DISCARD

OK

> GET foo

"1"

## 使用 check-and-set 操作实现乐观锁

[WATCH](http://www.redis.cn/commands/watch.html) 命令可以为 Redis 事务提供 check-and-set （CAS）行为。

被 [WATCH](http://www.redis.cn/commands/watch.html) 的键会被监视，并会发觉这些键是否被改动过了。 如果有至少一个被监视的键在 [EXEC](http://www.redis.cn/commands/exec.html) 执行之前被修改了， 那么整个事务都会被取消， [EXEC](http://www.redis.cn/commands/exec.html) 返回[nil-reply](http://www.redis.cn/topics/protocol.html" \l "nil-reply)来表示事务已经失败。

举个例子， 假设我们需要原子性地为某个值进行增 1 操作（假设 [INCR](http://www.redis.cn/commands/incr.html) 不存在）。

首先我们可能会这样做：

val = GET mykey

val = val + 1

SET mykey $val

上面的这个实现在只有一个客户端的时候可以执行得很好。 但是， 当多个客户端同时对同一个键进行这样的操作时， 就会产生竞争条件。举个例子， 如果客户端 A 和 B 都读取了键原来的值， 比如 10 ， 那么两个客户端都会将键的值设为 11 ， 但正确的结果应该是 12 才对。

有了 [WATCH](http://www.redis.cn/commands/watch.html) ， 我们就可以轻松地解决这类问题了：

WATCH mykey //jian

val = GET mykey

val = val + 1

MULTI

SET mykey $val //mykey 有没有发生变化，有变化则放弃

EXEC

使用上面的代码， 如果在 [WATCH](http://www.redis.cn/commands/watch.html) 执行之后， [EXEC](http://www.redis.cn/commands/exec.html) 执行之前， 有其他客户端修改了 mykey 的值， 那么当前客户端的事务就会失败。 程序需要做的， 就是不断重试这个操作， 直到没有发生碰撞为止。

这种形式的锁被称作乐观锁， 它是一种非常强大的锁机制。 并且因为大多数情况下， 不同的客户端会访问不同的键， 碰撞的情况一般都很少， 所以通常并不需要进行重试。

## 了解 WATCH

[WATCH](http://www.redis.cn/commands/watch.html) 使得 [EXEC](http://www.redis.cn/commands/exec.html) 命令需要有条件地执行： 事务只能在所有被监视键都没有被修改的前提下执行， 如果这个前提不能满足的话，事务就不会被执行。

[WATCH](http://www.redis.cn/commands/watch.html) 命令可以被调用多次。 对键的监视从 [WATCH](http://www.redis.cn/commands/watch.html) 执行之后开始生效， 直到调用 [EXEC](http://www.redis.cn/commands/exec.html) 为止。

用户还可以在单个 [WATCH](http://www.redis.cn/commands/watch.html) 命令中监视任意多个键， 就像这样：

redis> WATCH key1 key2 key3

OK

当 [EXEC](http://www.redis.cn/commands/exec.html) 被调用时， 不管事务是否成功执行， 对所有键的监视都会被取消。

另外， 当客户端断开连接时， 该客户端对键的监视也会被取消。

使用无参数的 [UNWATCH](http://www.redis.cn/commands/unwatch.html) 命令可以手动取消对所有键的监视。 对于一些需要改动多个键的事务， 有时候程序需要同时对多个键进行加锁， 然后检查这些键的当前值是否符合程序的要求。 当值达不到要求时， 就可以使用 [UNWATCH](http://www.redis.cn/commands/unwatch.html) 命令来取消目前对键的监视， 中途放弃这个事务， 并等待事务的下次尝试。

## 使用 WATCH 实现 ZPOP

[WATCH](http://www.redis.cn/commands/watch.html) 可以用于创建 Redis 没有内置的原子操作。举个例子， 以下代码实现了原创的 [ZPOP](http://www.redis.cn/commands/zpop.html) 命令， 它可以原子地弹出有序集合中分值（score）最小的元素：

WATCH zset

element = ZRANGE zset 0 0

MULTI

ZREM zset element

EXEC

程序只要重复执行这段代码， 直到 [EXEC](http://www.redis.cn/commands/exec.html) 的返回值不是[nil-reply](http://www.redis.cn/topics/protocol.html" \l "nil-reply)回复即可。

# Redis发布订阅

https://www.cnblogs.com/linjiqin/p/6277537.html

## 概述

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

下图展示了频道 channel1 ， 以及订阅这个频道的三个客户端 —— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：



## 常用命令

### PSUBSCRIBE pattern [pattern ...]

订阅一个或多个符合给定模式(pattern)的频道。支持的模式(patterns)有:

* + h?llo subscribes to hello, hallo and hxllo
  + h\*llo subscribes to hllo and heeeello
  + h[ae]llo subscribes to hello and hallo, but not hillo

### SUBSCRIBE channel [channel ...]

命令用于订阅给定的一个或多个频道的信息。

### PUBLISH channel message

Redis Publish 命令用于将信息发送到指定的频道。

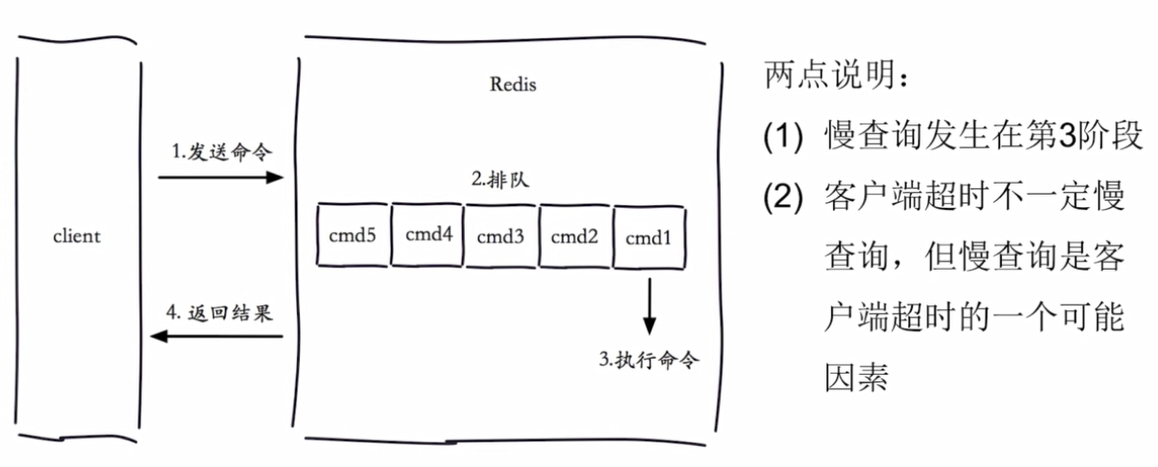
### PUNSUBSCRIBE [pattern [pattern ...]]

退订所有给定模式的频道。

### UNSUBSCRIBE [channel [channel ...]]

指退订给定的频道。

# Redis慢查询记录（运维）



## 慢查询的配置参数

1 慢查询的预设阀值 slowlog-log-slower-than

slowlog-log-slower-than参数就是预设阀值，单位是微秒,默认值是10000，如果一条命令的执行时间超过10000微妙，那么它将被记录在慢查询日志中。如果slowlog-log-slower-than的值是0，则会记录所有命令。如果slowlog-log-slower-than的值小于0，则任何命令都不会记录日志。

2 慢查询日志的长度 slowlog-max-len

slowlog-max-len只是说明了慢查询日志最多存储多少条。Redis使用一个列表来存储慢查询日志，showlog-max-len就是列表的最大长度。当慢查询日志已经到达列表的最大长度时，又有慢查询日志要进入列表，则最早插入列表的日志将会被移出列表，新日志被插入列表的末尾。

## 慢查询配置

1、默认值

config get slowlog-max-len = 128

config get slowlog-log-slower-than = 10000

2、配置方式

* 修改redis配置文件
* 执行命令动态配置

config set slowlog-max-len = 128

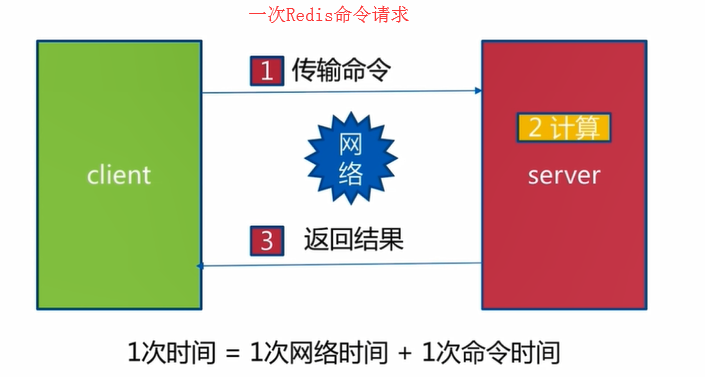
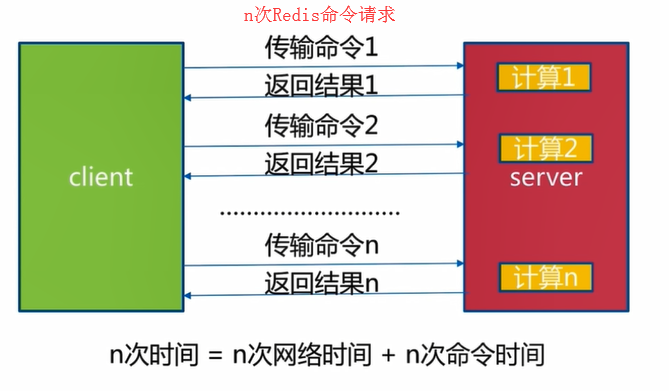
config set slowlog-log-slower-than = 10000

## 慢查询命令

* Slowlog get [n] ： 获取慢查询队列
* Slowlog len：获取慢查询队列长度
* Slowlog reset：清空慢查询

# Redis pipeline管道

当在使用redis时，有多条命令需要执行时，可以考虑使用pipeline管道来执行，这样可以减少网络开销，提高效率。如下图：

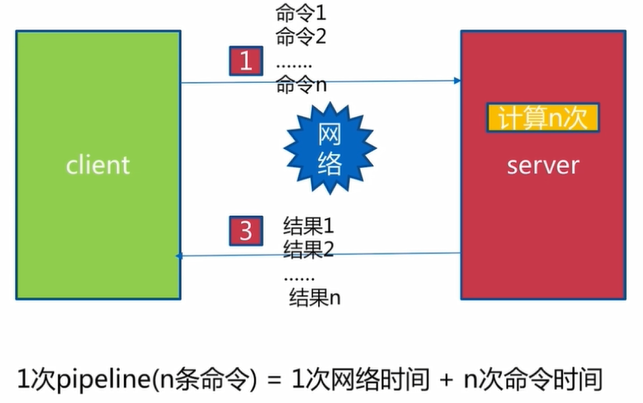
由此可见我们在执行多次Redis命令时，Redis命令的执行是非常快的，那么我们的时间消耗主要就花费在网络传输上了。如何解决呢，有两种方案：

方案一、使用Redis提供的批量执行的命令如mget、mset等；

方案二、使用pipeline管道。

## 什么是pipeline管道？

管道是将我们的许多命令打包，一次性发送到Redis服务器，然后执行，等全部命令执行完成后再将执行的结果，打包返回。



# 键空间通知(Redis Keyspace Notifications 监控！！)

## 概述

Keyspace notifications 允许客户端使用Redis的Pub/Sub去订阅那些影响Redis数据集的事件。这些事件包括：

* + All the commands affecting a given key.
  + All the keys receiving an LPUSH operation.
  + All the keys expiring in the database 0.

问题：

由于这些事件是通过Redis的Pub/Sub来传输的。这是不可靠的传输，如果我们我们的客户端挂掉后重连，那么redis在此期间产生的事件，redis客户端是无法接受的。如果我们的应用要求可靠性的消息传输，此处不适用。

后续会解决！！！！

## 事件类型

键空间通知发送的消息类型包括两种，例如del mykey 操作会发送两个消息，等同于如下操作：

* PUBLISH \_\_keyspace@0\_\_:mykey del
* PUBLISH \_\_keyevent@0\_\_:del mykey

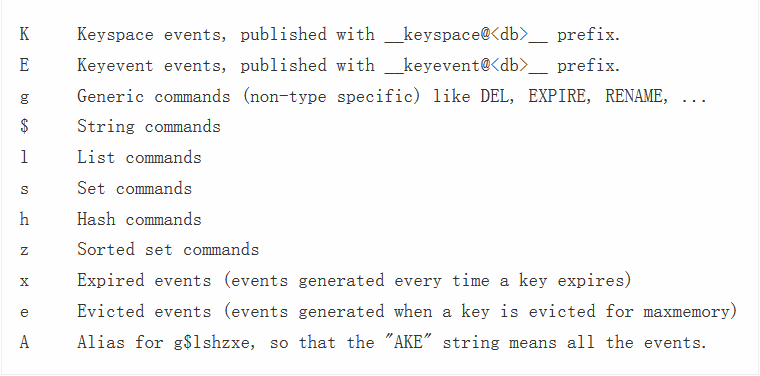
由上可知，我们的消息类型包括：

* **Key-space notification**：以 \_keyspace为前缀，可以监听当前key的所有事件
* **Key-event notification**：以 \_keyevent为前缀，可以监听发送指定事件的key

## 配置

默认键空间事件通知是不启动的，我们可以通过修改redis.conf配置文件的notify-keyspace-events或是通过执行**CONFIG SET** notify-keyspace-events**指令在redis运行时设置。**

通过字符串配置，字符串的每个字符的含义如下：



配置字符串至少包含K或E，否则将无任何事件通知。例如

* Kl: lists数据类型操作的Key-space事件通知；
* KEA:运行所有可能的事件通知。

## 不同命令产生的事件

Different commands generate different kind of events according to the following list.

* DEL generates a del event for every deleted key.
* RENAME generates two events, a rename\_from event for the source key, and a rename\_to event for the destination key.
* EXPIRE generates an expire event when an expire is set to the key, or a expired event every time setting an expire results into the key being deleted (see EXPIRE documentation for more info).
* SORT generates a sortstore event when STORE is used to set a new key. If the resulting list is empty, and the STORE option is used, and there was already an existing key with that name, the result is that the key is deleted, so a del event is generated in this condition.
* SET and all its variants (SETEX, SETNX,GETSET) generate set events. However SETEX will also generate an expire events.
* MSET generates a separated set event for every key.
* SETRANGE generates a setrange event.
* INCR, DECR, INCRBY, DECRBY commands all generate incrby events.
* INCRBYFLOAT generates an incrbyfloat events.
* APPEND generates an append event.
* LPUSH and LPUSHX generates a single lpush event, even in the variadic case.
* RPUSH and RPUSHX generates a single rpush event, even in the variadic case.
* RPOP generates an rpop event. Additionally a del event is generated if the key is removed because the last element from the list was popped.
* LPOP generates an lpop event. Additionally a del event is generated if the key is removed because the last element from the list was popped.
* LINSERT generates an linsert event.
* LSET generates an lset event.
* LTRIM generates an ltrim event, and additionally a del event if the resulting list is empty and the key is removed.
* RPOPLPUSH and BRPOPLPUSH generate an rpop event and an lpush event. In both cases the order is guaranteed (the lpush event will always be delivered after the rpop event). Additionally a del event will be generated if the resulting list is zero length and the key is removed.
* HSET, HSETNX and HMSET all generate a single hset event.
* HINCRBY generates an hincrby event.
* HINCRBYFLOAT generates an hincrbyfloat event.
* HDEL generates a single hdel event, and an additional del event if the resulting hash is empty and the key is removed.
* SADD generates a single sadd event, even in the variadic case.
* SREM generates a single srem event, and an additional del event if the resulting set is empty and the key is removed.
* SMOVE generates an srem event for the source key, and an sadd event for the destination key.
* SPOP generates an spop event, and an additional del event if the resulting set is empty and the key is removed.
* SINTERSTORE, SUNIONSTORE, SDIFFSTORE generate sinterstore, sunionostore, sdiffstore events respectively. In the special case the resulting set is empty, and the key where the result is stored already exists, a del event is generated since the key is removed.
* ZINCR generates a zincr event.
* ZADD generates a single zadd event even when multiple elements are added.
* ZREM generates a single zrem event even when multiple elements are deleted. When the resulting sorted set is empty and the key is generated, an additional del event is generated.
* ZREMBYSCORE generates a single zrembyscore event. When the resulting sorted set is empty and the key is generated, an additional del event is generated.
* ZREMBYRANK generates a single zrembyrank event. When the resulting sorted set is empty and the key is generated, an additional del event is generated.
* ZINTERSTORE and ZUNIONSTORE respectively generate zinterstore and zunionstore events. In the special case the resulting sorted set is empty, and the key where the result is stored already exists, a del event is generated since the key is removed.
* Every time a key with a time to live associated is removed from the data set because it expired, an expired event is generated.
* Every time a key is evicted from the data set in order to free memory as a result of the maxmemory policy, an evicted event is generated.

**IMPORTANT** ： all the commands generate events only if the target key is really modified. For instance an SREM deleting a non-existing element from a Set will not actually change the value of the key, so no event will be generated.

示例：

$ redis-cli config set notify-keyspace-events KEA 开启事件通知

$ redis-cli psubscribe '\_\_key\*\_\_:\*'

## Timing of expired events(失效时间的事件通知延时是不可避免的)

Keys with a time to live associated are expired by Redis in two ways:

* When the key is accessed by a command and is found to be expired.
* Via a background system that looks for expired keys in background, incrementally, in order to be able to also collect keys that are never accessed.

The expired events are generated when a key is accessed and is found to be expired by one of the above systems, as a result there are no guarantees that the Redis server will be able to generate the expired event at the time the key time to live reaches the value of zero.

If no command targets the key constantly, and there are many keys with a TTL associated, there can be a significant delay between the time the key time to live drops to zero, and the time the expired event is generated.

Basically expired events **are generated when the Redis server deletes the key** and not when the time to live theoretically reaches the value of zero.

# Redis锁（分布式锁）

http://www.cnblogs.com/liuyang0/p/6744076.html

## redis加锁分类

Redis能用的的加锁命令分别是INCR、SETNX、**SET(推介使用，前面两个不推介使用，会有死锁的可能)**

### 第一种锁命令INCR

这种加锁的思路是， key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作进行加一。   
然后其它用户在执行 INCR 操作进行加一时，如果返回的数大于 1 ，说明这个锁正在被使用当中。

1、 客户端A请求服务器获取key的值为1表示获取了锁 incr key del

2、 客户端B也去请求服务器获取key的值为2表示获取锁失败

3、 客户端A执行代码完成，删除锁

4、 客户端B在等待一段时间后在去请求的时候获取key的值为1表示获取锁成功

5、 客户端B执行代码完成，删除锁

$redis->incr($key);

$redis->expire($key, $ttl); //设置生成时间为1秒

### 第二种锁SETNX

这种加锁的思路是，如果 key 不存在，将 key 设置为 value   
如果 key 已存在，则 SETNX 不做任何动作

1、 客户端A请求服务器设置key的值，如果设置成功就表示加锁成功 1

2、 客户端B也去请求服务器设置key的值，如果返回失败，那么就代表加锁失败 0

3、 客户端A执行代码完成，删除锁

4、 客户端B在等待一段时间后在去请求设置key的值，设置成功 1

5、 客户端B执行代码完成，删除锁 del

$redis->setNX($key, $value);

$redis->expire($key, $ttl); 防止死锁！！

### 第三种锁SET

上面两种方法都有一个问题，会发现，都需要设置 key 过期。那么为什么要设置key过期呢？如果请求执行因为某些原因意外退出了，导致创建了锁但是没有删除锁，那么这个锁将一直存在，以至于以后缓存再也得不到更新。于是乎我们需要给锁加一个过期时间以防不测。   
但是借助 Expire 来设置就不是原子性操作了。所以还可以通过事务来确保原子性，但是还是有些问题，所以官方就引用了另外一个，使用 SET 命令本身已经从版本 2.6.12 开始包含了设置过期时间的功能。

1、 客户端A请求服务器设置key的值，如果设置成功就表示加锁成功

2、 客户端B也去请求服务器设置key的值，如果返回失败，那么就代表加锁失败

3、 客户端A执行代码完成，删除锁

4、 客户端B在等待一段时间后在去请求设置key的值，设置成功

5、 客户端B执行代码完成，删除锁

$redis->set($key, $value, array('nx', 'ex' => $ttl)); setnx //ex表示秒

### 其它问题

虽然上面一步已经满足了我们的需求，但是还是要考虑其它问题？   
1、 redis发现锁失败了要怎么办？中断请求还是循环请求？   
2、 循环请求的话，如果有一个获取了锁，其它的在去获取锁的时候，是不是容易发生抢锁的可能？   
3、 锁提前过期后，客户端A还没执行完，然后客户端B获取到了锁，这时候客户端A执行完了，会不会在删锁的时候把B的锁给删掉？

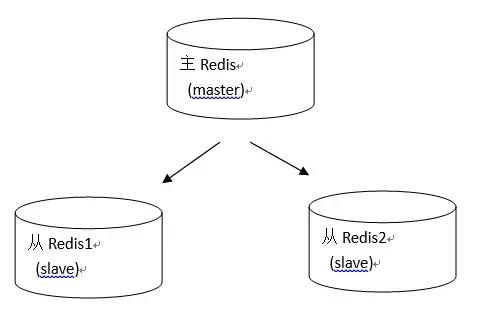
### 解决办法

针对问题1：使用循环请求，循环请求去获取锁   
针对问题2：针对第二个问题，在循环请求获取锁的时候，加入睡眠功能，等待几毫秒在执行循环   
针对问题3：在加锁的时候存入的key是随机的。这样的话，每次在删除key的时候判断下存入的key里的value和自己存的是否一样

# Redis架构

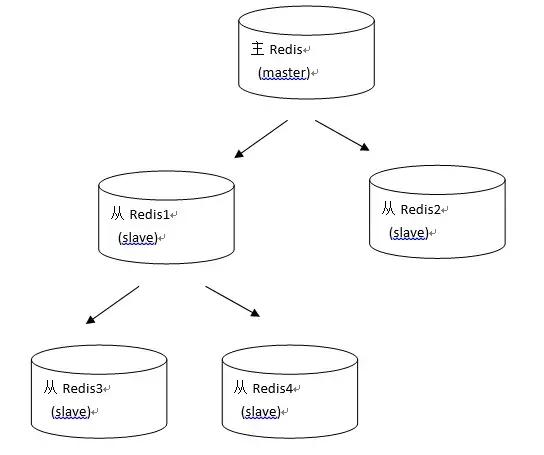
## 12.1、Redis主从复制

Redis持久化保证了即使 Redis服务重启也会丢失数据，因为 Redis服务重启后会将硬盘上持久化的数据恢复到内存中，但是当 Redis服务器的硬盘损坏了可能会导致数据丢失，如果通过 Redis的主从复制机制就可以避免这种单点故障。而且通过主从复制可以实现Redis的读写分离，提高Redis的读性能。如下图：



* 主 redis 中的数据有两个副本（replication）即从 redis1 和从 redis2，即使一台 redis 服务器宕机其它两台 redis 服务也可以继续提供服务；
* 主 redis 中的数据和从 redis 上的数据保持实时同步，当主 redis 写入数据时通过主从复制机制会复制到两个从 redis 服务上；
* 只有一个主 redis，可以有多个从 redis；
* 主从复制不会阻塞 master，在同步数据时，master 可以继续处理 client 请求。

当然一个 redis 可以即是主又是从，如下图：



### 12.2.1、如何实现主从复制

#### 配置方式

**主 redis 配置** ：无需特殊配置（默认Redis节点都是主节点）。

**从redis配置** ：修改从 redis 服务器上的 redis.conf 文件，添加

slaveof 主节点IP 主节点端口

Redis主从复制看这篇就够了

#### 命令方式

设置从节点：slaveof 主节点IP 主节点端口

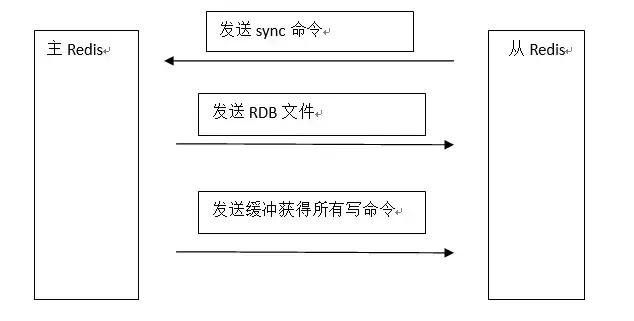
取消从节点：slaveof no one

查看节点主从详情：info replication

### 12.2.2、主从复制过程

#### 全量复制

在 redis2.8 版本之前主从复制过程如下图：



复制过程说明：

* slave 服务启动，slave 会建立和 master 的连接，发送 sync 命令；
* master 启动一个后台进程将数据库快照保存到 RDB 文件中；

注意：此时如果生成 RDB 文件过程中存在写数据操作会导致 RDB 文件和当前主 redis 数据不一致，所以此时 master 主进程会开始收集写命令并缓存起来。

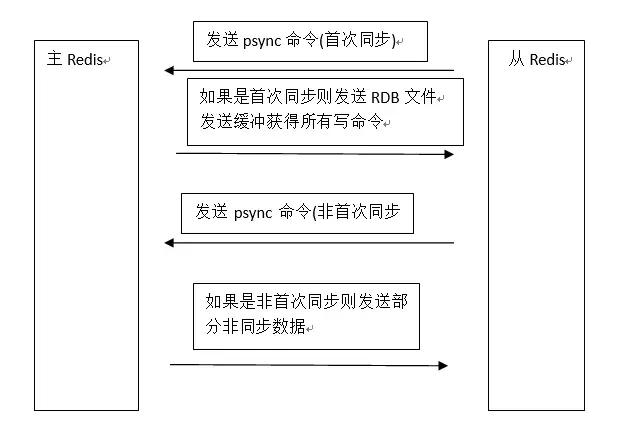
* master将RDB文件发送给slave；
* slave 将文件保存到磁盘上，先清空缓存，然后再加载到内存恢复；
* master 把缓存的命令转发给 slave

注意：后续 master 收到的写命令都会通过开始建立的连接发送给 slave。 当 master 和 slave 的连接断开时 slave 可以自动重新建立连接。如果 master 同时收到多个 slave 发来的同步连接命令，只会启动一个进程来写数据库镜像RDB文件，然后发送给所有slave。

上述过程称为全量复制。

在 redis2.8 之前slaver节点每次同步都会从master节点中复制全部的数据，如果slaver节点是新创建的，从master中复制全部的数据这是没有问题的，但是，如果当slaver节点停止运行，再启动时可能只有少部分数据和master节点不同步。此时还是执行的全量复制，这样对Redis的性能是由影响的。

#### 部分复制

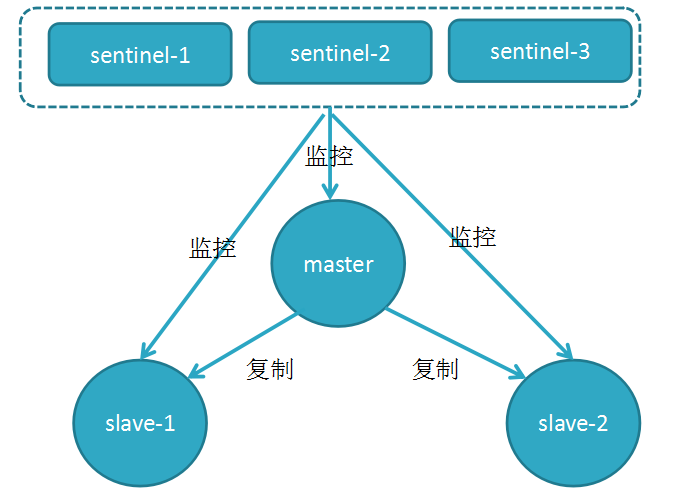


部分复制说明：

从机连接主机后，会主动发起 PSYNC 命令，从机会提供 master 的 runid(机器标识，随机生成的一个串) 和 offset（数据偏移量，如果offset主从不一致则说明数据不同步），主机验证 runid 和 offset 是否有效，runid 相当于主机身份验证码，用来验证从机上一次连接的主机，如果 runid 验证未通过则，则进行全同步，如果验证通过则说明曾经同步过，根据 offset 同步部分数据。

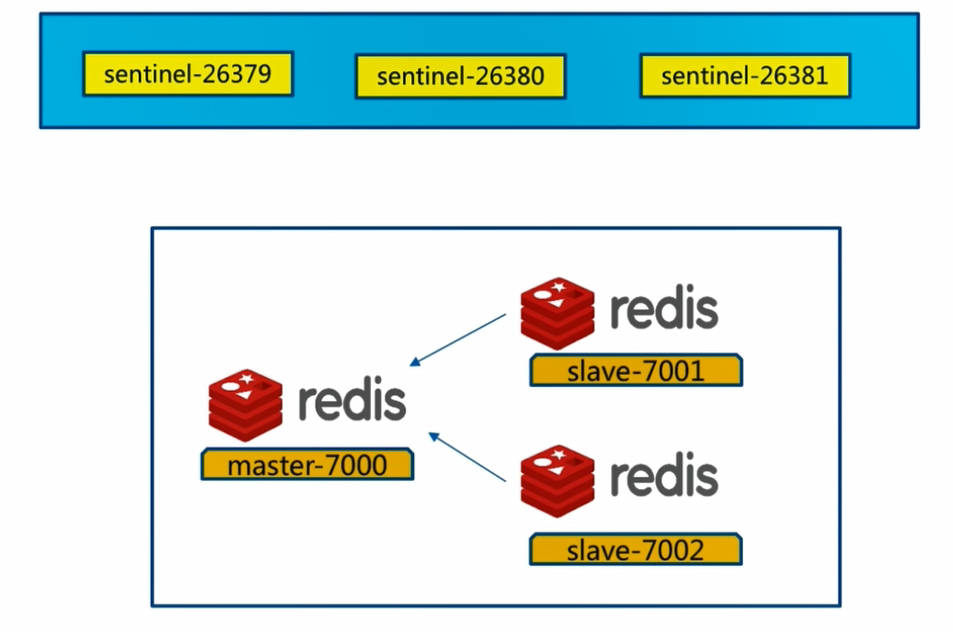
## 12.2、Redis哨兵（Sentinel）

上面所说的主从复制模式，当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这个时候我们可以考虑Redis的哨兵机制。它可以帮助我们在主从复制模式下，完成故障的自动转移。



一个哨兵Sentinel集群可以同时监控多个Master-Slave。

### 12.2.1、哨兵的安装与配置



第一步、搭建一个如上图的一个Redis的Master-Slaver主从复制结构。

第二步、配置sentinel哨兵监控Master主节点

进入redis的安装目录，找到sentinel.conf哨兵配置文件，进行对应的修改

port 26379

dir /tmp

logfile "26379.log"

daemonize yes

sentinel monitor mymaster 192.168.25.135 6379 2

sentinel down-after-milliseconds mymaster 30000

sentinel parallel-syncs mymaster 1

sentinel failover-timeout mymaster 180000

启动redis的哨兵

./redis-server sentinel.conf –sentinel

连接redis哨兵

./redis-cli -p 26379

指定info命令可以查看当前哨兵状态

### 12.2.2、通过java客户端连接redis哨兵

**public static void** main(String[] args) {  
 String masterName = **"mymaster"**;  
 Set<String> sentinels = **new** HashSet<>();  
 sentinels.add(**"192.168.25.135:26379"**);  
 sentinels.add(**"192.168.25.135:26380"**);  
 sentinels.add(**"192.168.25.135:26381"**);  
 JedisSentinelPool jedisSentinelPool = **new** JedisSentinelPool(masterName,sentinels);  
 **int** count=0;  
 **while** (**true**){  
 Jedis jedis = **null**;  
 **try** {  
 count++;  
 jedis = jedisSentinelPool.getResource();  
 **int** i = **new** Random().nextInt(100000);  
 String key = **"k-"**+i;  
 String value = **"v-"**+i;  
 jedis.set(key,value);  
 **if**(count%100==0){  
 System.***out***.println(key+**",value is "**+jedis.get(key));  
 }  
 TimeUnit.***MICROSECONDS***.sleep(100);  
 }**catch** (Exception e){  
 e.printStackTrace();  
 }**finally** {  
 **if**(jedis!=**null**){  
 jedis.close();  
 }  
 }  
 }  
  
}

### 12.2.3、故障转移演练

第一步、关闭6379 Redis的mater主节点

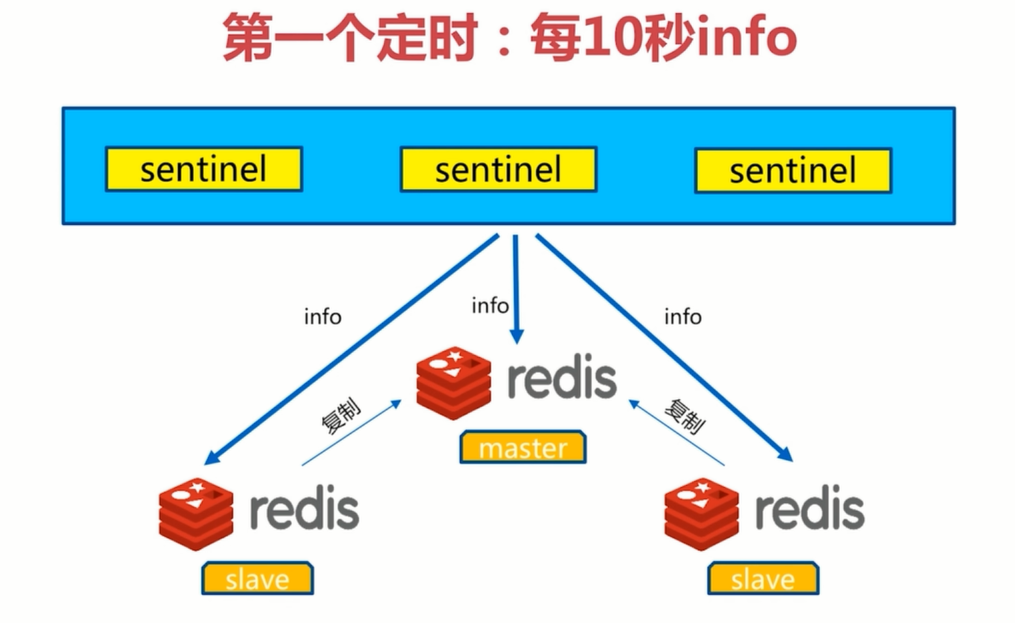
第二步、观察Redis java客户端代码的运行情况。

发现控制台会输出Redis的连接异常，过一会后，恢复正常

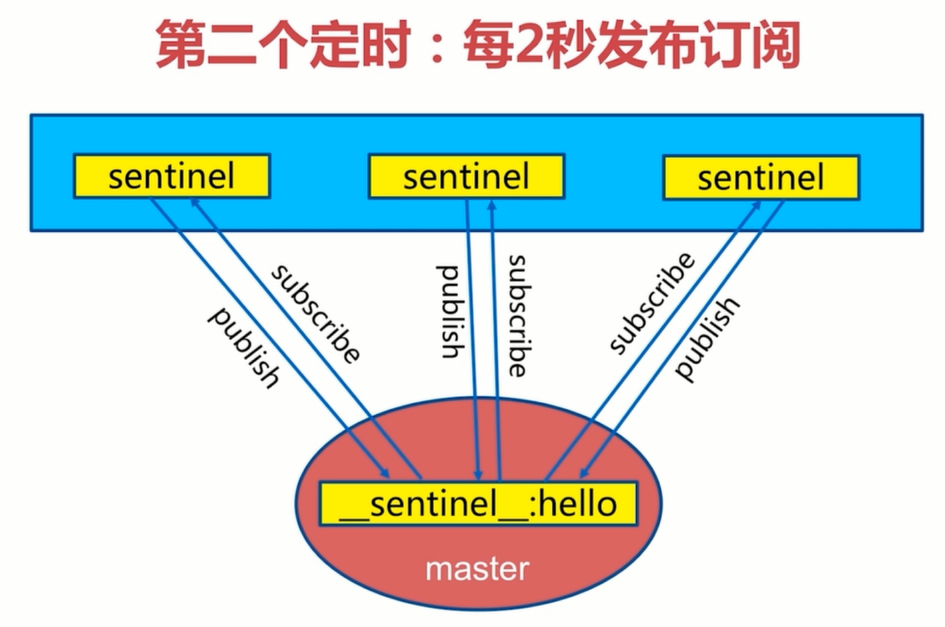
第三步、查看Redis相关日志

### 12.2.4、哨兵的定时监控任务

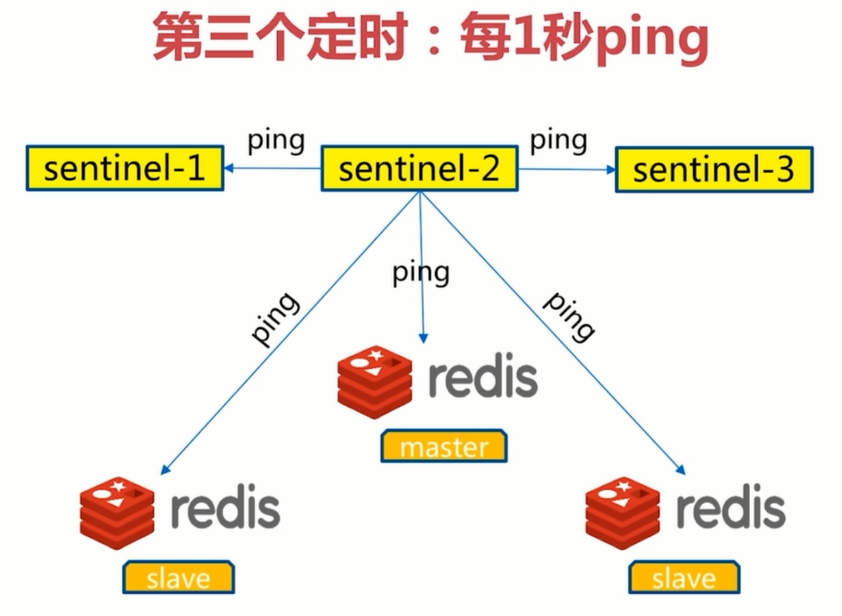
任务1：每个哨兵节点每10秒会向主节点和从节点发送info命令获取最拓扑结构图，哨兵配置时只要配置对主节点的监控即可，通过向主节点发送info，获取从节点的信息，并当有新的从节点加入时可以马上感知到



任务2：每个哨兵节点每隔2秒会向redis数据节点的指定频道上发送该哨兵节点对于主节点的判断以及当前哨兵节点的信息，同时每个哨兵节点也会订阅该频道，来了解其它哨兵节点的信息及对主节点的判断，其实就是通过消息publish和subscribe来完成的



任务3：每隔1秒每个哨兵会向主节点、从节点及其余哨兵节点发送一次ping命令做一次心跳检测，这个也是哨兵用来判断节点是否正常的重要依据



### 12.2.5、主观下线和客观下线

# 监听地址为ip:port的一个master

sentinel monitor mymaster 127.0.0.1 6379 2

# 指定了Sentinel认为Redis实例已经失效所需的毫秒数

sentinel down-after-milliseconds mymaster 30000

如果服务器在给定的毫秒数之内， 没有返回 Sentinel 发送的 PING 命令的回复， 或者返回一个错误， 那么 Sentinel 将这个服务器标记为**主观下线（subjectively down，简称 SDOWN ）**。

不过只有一个 Sentinel 将服务器标记为主观下线并不一定会引起服务器的自动故障迁移： 只有在足够数量的 Sentinel 都将一个服务器标记为主观下线之后， 服务器才会被标记为**客观下线（objectively down， 简称 ODOWN ）**， 这时自动故障迁移才会执行。将服务器标记为客观下线所需的 Sentinel 数量由对主服务器的配置决定。

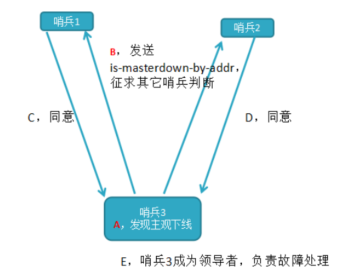
如果 Sentinel 在给定的时间范围内， 从其他 Sentinel 那里接收到了足够数量的主服务器下线报告， 那么 Sentinel 就会将主服务器的状态从主观下线改变为客观下线。 如果之后其他 Sentinel 不再报告主服务器已下线， 那么客观下线状态就会被移除。

客观下线条件只适用于主服务器： 对于任何其他类型的 Redis 实例， Sentinel 在将它们判断为下线前不需要进行协商， 所以从服务器或者其他 Sentinel 永远不会达到客观下线条件。

只要一个 Sentinel 发现某个主服务器进入了客观下线状态， 这个 Sentinel 就可能会被其他 Sentinel 推选出， 并对失效的主服务器执行自动故障迁移操作。

### 12.2.6、sentinel领导者选举（Raft leader election）

1. 每个在线的哨兵节点都可以成为领导者，当它确认（比如哨兵3）主节点下线时，会向其它哨兵发is-master-down-by-addr命令，征求判断并要求将自己设置为领导者，由领导者处理故障转移；
2. 当其它哨兵收到此命令时，可以同意或者拒绝它成为领导者；
3. 如果哨兵3发现自己在选举的票数大于等于num(sentinels)/2+1，且大于qroums，将成为领导者，如果没有超过，继续选举…………



### 12.2.7、master节点故障转移

1. 从slave节点中选择出一个合适的节点作为一个新的master节点；
2. 对上面选出的节点执行slaveof no one 命令让其成为master节点；
3. 向剩余的slave节点发送命令，让它们成为新的master节点的slave节点，复制规则由参数sentinel parallel-syncs mymaster n决定；
4. 通知Redis客户端程序主节点的变更；
5. 更新原来master节点的配置为slave，并保持对其“关注”，当它恢复后命令它去复制新的master节点。

**主节点master的选举：**

a) 过滤掉不健康的（下线或断线），没有回复过哨兵ping响应的从节点

b) 选择salve-priority从节点优先级最高（redis.conf）的

c) 选择复制偏移量最大，指复制最完整的从节点

### 12.2.8、哨兵节点相关命令

* ping：返回pong；
* sentinel masters ：展示所有被监控的主节点状态以及相关的统计信息；
* sentinel sentinels <master name> ：展示指定<master name>的Sentinel节点集合，不包括当前Sentinel节点；
* sentinel remove <master name> ：取消当前Sentinel节点对指定<master name>主节点的监控；
* sentinel slaves <master name> ：列出给定主服务器的所有从服务器，以及这些从服务器的当前状态；
* sentinel get-master-addr-by-name <master name> ： 返回给定名字的主服务器的 IP 地址和端口号。 如果这个主服务器正在执行故障转移操作， 或者针对这个主服务器的故障转移操作已经完成， 那么这个命令返回新的主服务器的 IP 地址和端口号；
* sentinel reset <pattern> ： 重置所有名字和给定模式 pattern 相匹配的主服务器。 pattern 参数是一个 Glob 风格的模式。 重置操作清楚主服务器目前的所有状态， 包括正在执行中的故障转移， 并移除目前已经发现和关联的， 主服务器的所有从服务器和 Sentinel ；
* sentinel failover <master name> ： 当主服务器失效时， 在不询问其他 Sentinel 意见的情况下， 强制开始一次自动故障迁移 （不过发起故障转移的 Sentinel 会向其他 Sentinel 发送一个新的配置，其他 Sentinel 会根据这个配置进行相应的更新）；
* sentinel ckquorum<master name>：检测当前可达的Sentinel节点总数是否达到<quorum>的个数。例如 quorum=3，而当前可达的Sentinel节点个数为2个，那么将无法进行故障转 移，Redis Sentinel的高可用特性也将失去；
* sentinel flushconfig：将Sentinel节点的配置强制刷到磁盘上，这个命令Sentinel节点自身用得 比较多，对于开发和运维人员只有当外部原因（例如磁盘损坏）造成配置文 件损坏或者丢失时，这个命令是很有用的；
* sentinel monitor<master name><ip><port><quorum>：这个命令和配置文件中的含义是完全一样的，只不过是通过命令的形式 来完成Sentinel节点对主节点的监控。

## 12.3、Redis集群

# 问题

绕过redis访问数据库！！ 10万

## 什么是Redis缓存穿透，如何解决？

**缓存穿透**

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

**解决方案**

有一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

## 什么是Redis雪崩，如何解决？

**缓存雪崩**

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

**解决方案**

简单方案就是将缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

## 什么是Redis击穿，如何解决？

**缓存击穿**

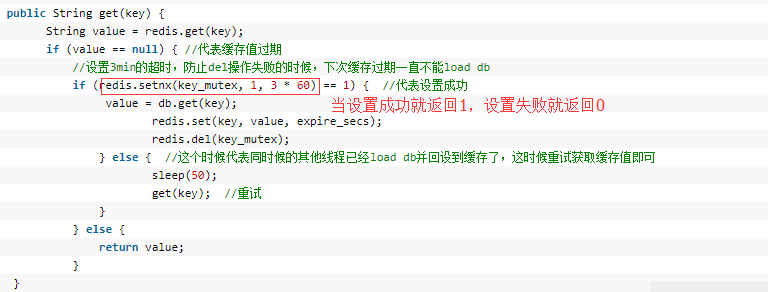
对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

**1.使用互斥锁(mutex key)**

使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。

SETNX，是「SET if Not exists」的缩写，也就是只有不存在的时候才设置，可以利用它来实现锁的效果。



**2. "永远不过期"：**

这里的“永远不过期”包含两层意思：

(1) 从redis上看，确实没有设置过期时间，这就保证了，不会出现热点key过期问题，也就是“物理”不过期。

(2) 从功能上看，如果不过期，那不就成静态的了吗？所以我们把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期

从实战看，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据，但是对于一般的互联网功能来说这个还是可以忍受。

## Redis使用超过最大内存会怎样？

Maxmemory 淘汰机制！！

## Redis可以实现消息发布订阅功能吗？

## 做过redis持久化吗，如何实现的？

## 搭建过redis集群吗，如何搭建的？

## Redis支持事务吗？

## Redis事务相关的命令有哪几个？

## Redis支持哪几种数据类型？

## Redis有哪几种数据淘汰策略？

## 一个字符串类型的值能存储最大容量是多少？

2\*32 512M

## Redis集群最大节点个数是多少？

<https://blog.csdn.net/tianyeshiye/article/details/79600014>

## Redis中的管道有什么用？

16384个 ？？？