# Reids命令

Redis命令十分丰富，包括的命令组有Cluster、Connection、Geo、Hashes、HyperLogLog、Keys、Lists、Pub/Sub、Scripting、Server、Sets、Sorted Sets、Strings、Transactions一共14个redis命令组两百多个redis命令，下面我们一个一个来介绍如何使用Redis命令。

## APPEND key value

起始版本：2.0.0

时间复杂度：O(1)。均摊时间复杂度是O(1)， 因为redis用的动态字符串的库在每次分配空间的时候会增加一倍的可用空闲空间，所以在添加的value较小而且已经存在的 value是任意大小的情况下，均摊时间复杂度是O(1) 。

如果 key 已经存在，并且值为字符串，那么这个命令会把 value 追加到原来值（value）的结尾。 如果 key 不存在，那么它将首先创建一个空字符串的key，再执行追加操作，这种情况 APPEND 将类似于 SET 操作。

返回值

Integer reply：返回append后字符串值（value）的长度。

例子

redis> EXISTS mykey

(integer) 0

redis> APPEND mykey "Hello"

(integer) 5

redis> APPEND mykey " World"

(integer) 11

redis> GET mykey

"Hello World"

redis>

模式：节拍序列(Time series)

APPEND 命令可以用来连接一系列固定长度的样例,与使用列表相比这样更加紧凑. 通常会用来记录节拍序列. 每收到一个新的节拍样例就可以这样记录:

APPEND timeseries "fixed-size sample"

在节拍序列里, 可以很容易地访问序列中的每个元素:

STRLEN 可以用来计算样例个数.

GETRANGE 允许随机访问序列中的各个元素. 如果序列中有明确的节拍信息, 在Redis 2.6中就可以使用GETRANGE配合Lua脚本来实现一个二分查找算法.

SETRANGE 可以用来覆写已有的节拍序列.

该模式的局限在于只能做追加操作. Redis目前缺少剪裁字符串的命令, 所以无法方便地把序列剪裁成指定的尺寸. 但是, 节拍序列在空间占用上效率极好.

小贴士: 在键值中组合Unix时间戳, 可以在构建一系列相关键值时缩短键值长度,更优雅地分配Redis实例.

使用定长字符串进行温度采样的例子(在实际使用时,采用二进制格式会更好).

redis> APPEND ts "0043"

(integer) 4

redis> APPEND ts "0035"

(integer) 8

redis> GETRANGE ts 0 3

"0043"

redis> GETRANGE ts 4 7

"0035"

redis>

## AUTH password

起始版本：1.0.0

为redis服务请求设置一个密码。redis可以设置在客户端执行commands请求前需要通过密码验证。通过修改配置文件的requirepass就可以设置密码。 如果密码与配置文件里面设置的密码一致，服务端就会发会一个OK的状态码，接受客户端发送其他的请求命令，否则服务端会返回一个错误码，客户端需要尝试使用新的密码来进行连接。

注意: 因为redis的高性能能在短时间接受非常多的尝试性密码，所以请务必设置一个足够复杂的密码以防止可能的攻击。

返回值

simple-string-reply

## BGREWRITEAOF

起始版本：1.0.0

Redis BGREWRITEAOF 命令用于异步执行一个 AOF（AppendOnly File）文件重写操作。重写会创建一个当前AOF文件的体积优化版本。

即使 BGREWRITEAOF 执行失败，也不会有任何数据丢失，因为旧的AOF文件在BGREWRITEAOF 成功之前不会被修改。

AOF 重写由 Redis 自行触发， BGREWRITEAOF仅仅用于手动触发重写操作。

具体内容:

如果一个子Redis是通过磁盘快照创建的，AOF重写将会在RDB终止后才开始保存。这种情况下BGREWRITEAOF任然会返回OK状态码。从Redis 2.6起你可以通过INFO命令查看AOF重写执行情况。

如果只在执行的AOF重写返回一个错误，AOF重写将会在稍后一点的时间重新调用。

从 Redis 2.4 开始，AOF重写由 Redis 自行触发，BGREWRITEAOF仅仅用于手动触发重写操作。

请参考Redis 持久化了解更多详细信息。

返回值

simple-string-reply: 总是返回 OK。

## BGSAVE

起始版本：1.0.0

后台保存DB。会立即返回 OK 状态码。 Redis forks, 父进程继续提供服务以供客户端调用，子进程将DB数据保存到磁盘然后退出。如果操作成功，可以通过客户端命令LASTSAVE来检查操作结果。

请参考Redis 持久化了解更多详细信息。

返回值

simple-string-reply

## BITCOUNT key [start end]

起始版本：2.6.0

时间复杂度：O(N)

统计字符串被设置为1的bit数.

一般情况下，给定的整个字符串都会被进行计数，通过指定额外的 start 或 end 参数，可以让计数只在特定的位上进行。

start 和 end 参数的设置和 GETRANGE 命令类似，都可以使用负数值：比如 -1 表示最后一个位，而 -2 表示倒数第二个位，以此类推。

不存在的 key 被当成是空字符串来处理，因此对一个不存在的 key 进行 BITCOUNT 操作，结果为 0 。

返回值

Integer reply

被设置为 1 的位的数量。

例子

redis> SET mykey "foobar"

OK

redis> BITCOUNT mykey

(integer) 26

redis> BITCOUNT mykey 0 0

(integer) 4

redis> BITCOUNT mykey 1 1

(integer) 6

redis>

模式：使用 bitmap 实现用户上线次数统计

Bitmap 对于一些特定类型的计算非常有效。

假设现在我们希望记录自己网站上的用户的上线频率，比如说，计算用户 A 上线了多少天，用户 B 上线了多少天，诸如此类，以此作为数据，从而决定让哪些用户参加 beta 测试等活动 —— 这个模式可以使用 SETBIT 和 BITCOUNT 来实现。

比如说，每当用户在某一天上线的时候，我们就使用 SETBIT ，以用户名作为 key ，将那天所代表的网站的上线日作为 offset 参数，并将这个 offset 上的为设置为 1 。

举个例子，如果今天是网站上线的第 100 天，而用户 peter 在今天阅览过网站，那么执行命令 SETBIT peter 100 1 ；如果明天 peter 也继续阅览网站，那么执行命令 SETBIT peter 101 1 ，以此类推。

当要计算 peter 总共以来的上线次数时，就使用 BITCOUNT 命令：执行 BITCOUNT peter ，得出的结果就是 peter 上线的总天数。

更详细的实现可以参考博文 Fast, easy, realtime metrics using Redis bitmaps (需要翻墙)

性能

前面的上线次数统计例子，即使运行 10 年，占用的空间也只是每个用户 10\*365 比特位(bit)，也即是每个用户 456 字节。对于这种大小的数据来说， BITCOUNT 的处理速度就像 GET 和 INCR 这种 O(1) 复杂度的操作一样快。

如果你的 bitmap 数据非常大，那么可以考虑使用以下两种方法：

将一个大的 bitmap 分散到不同的 key 中，作为小的 bitmap 来处理。使用 Lua 脚本可以很方便地完成这一工作。

使用 BITCOUNT 的 start 和 end 参数，每次只对所需的部分位进行计算，将位的累积工作(accumulating)放到客户端进行，并且对结果进行缓存 (caching)。

## BITFIELD key [GET type offset] [SET type offset value] [INCRBY type offset increment] [OVERFLOW WRAP|SAT|FAIL]

起始版本：3.2.0

时间复杂度：O(1) for each subcommand specified

本命令会把Redis字符串当作位数组，并能对变长位宽和任意未字节对齐的指定整型位域进行寻址。在实践中，可以使用该命令对一个有符号的5位整型数的1234位设置指定值，也可以对一个31位无符号整型数的4567位进行取值。类似地，在对指定的整数进行自增和自减操作，本命令可以提供有保证的、可配置的上溢和下溢处理操作。

BITFIELD命令能操作多字节位域，它会执行一系列操作，并返回一个响应数组，在参数列表中每个响应数组匹配相应的操作。

例如，下面的命令是对一个8位有符号整数偏移100位自增1，并获取4位无符号整数的值：

> BITFIELD mykey INCRBY i5 100 1 GET u4 0

1) (integer) 1

2) (integer) 0

提示：

用GET指令对超出当前字符串长度的位（含key不存在的情况）进行寻址，执行操作的结果会对缺失部分的位（bits）赋值为0。

用SET或INCRBY指令对超出当前字符串长度的位（含key不存在的情况）进行寻址，将会扩展字符串并对扩展部分进行补0，扩展方式包括：按需扩展、按最小长度扩展和按最大寻址能力扩展。

支持子命令和整型

下面是已支持的命令列表：

GET <type> <offset> – 返回指定的位域

SET <type> <offset> <value> – 设置指定位域的值并返回它的原值

INCRBY <type> <offset> <increment> – 自增或自减（如果increment为负数）指定位域的值并返回它的新值

还有一个命令通过设置溢出行为来改变调用INCRBY指令的后序操作：

OVERFLOW [WRAP|SAT|FAIL]

当需要一个整型时，有符号整型需在位数前加i，无符号在位数前加u。例如，u8是一个8位的无符号整型，i16是一个16位的有符号整型。

有符号整型最大支持64位，而无符号整型最大支持63位。对无符号整型的限制，是由于当前Redis协议不能在响应消息中返回64位无符号整数。

位和位偏移

bitfield命令有两种方式来指定位偏移。如果未定带数字的前缀，将会以字符串的第0位作为起始位。

不过，如果偏移量带有#前缀，那么指定的偏移量需要乘以整型宽度，例如：

BITFIELD mystring SET i8 #0 100 i8 #1 200

将会在第1个i8整数的偏移0位和第2个整数的偏移8位进行设值。如果想得到一个给定长度的普通整型数组，则不一定要在客户端进行计算。

溢出控制

使用OVERFLOW命令，用户可以通过指定下列其中一种行为来调整自增或自减操作溢出（或下溢）后的行为：

WRAP: 回环算法，适用于有符号和无符号整型两种类型。对于无符号整型，回环计数将对整型最大值进行取模操作（C语言的标准行为）。对于有符号整型，上溢从最负的负数开始取数，下溢则从最大的正数开始取数，例如，如果i8整型的值设为127，自加1后的值变为-128。

SAT: 饱和算法，下溢之后设为最小的整型值，上溢之后设为最大的整数值。例如，i8整型的值从120开始加10后，结果是127，继续增加，结果还是保持为127。下溢也是同理，但量结果值将会保持在最负的负数值。

FAIL: 失败算法，这种模式下，在检测到上溢或下溢时，不做任何操作。相应的返回值会设为NULL，并返回给调用者。

注意每种溢出（OVERFLOW）控制方法，仅影响紧跟在INCRBY命令后的子命令，直到重新指定溢出（OVERFLOW）控制方法。

如果没有指定溢出控制方法，默认情况下，将使用WRAP算法。

> BITFIELD mykey incrby u2 100 1 OVERFLOW SAT incrby u2 102 1

1) (integer) 1

2) (integer) 1

> BITFIELD mykey incrby u2 100 1 OVERFLOW SAT incrby u2 102 1

1) (integer) 2

2) (integer) 2

> BITFIELD mykey incrby u2 100 1 OVERFLOW SAT incrby u2 102 1

1) (integer) 3

2) (integer) 3

> BITFIELD mykey incrby u2 100 1 OVERFLOW SAT incrby u2 102 1

1) (integer) 0

2) (integer) 3

返回值

本命令返回一个针对子命令给定位置的处理结果组成的数组。OVERFLOW子命令在响应消息中，不会统计结果的条数。

下面是OVERFLOW FAIL返回NULL的样例：

> BITFIELD mykey OVERFLOW FAIL incrby u2 102 1

1) (nil)

动机（Motivations）

本命令的动机是为了能够在单个大位图（large bitmap）中高效地存储多个小整数（或对键分成多个key，避免出现超大键），同时开放Redis提供的新使用案例，尤其是在实时分析领域。这种使用案例可以通过指定的溢出控制方法来支持。

性能考虑（Performance considerations）

通常，BITFIELD是一个非常快的命令，但是注意，对短字符串的远地址（fat bits）寻址，将会比在存在的位执行命令更加耗时。

字节序（Orders of bits）

BITFIELD命令使用的位图表现形式，可看作是从0位开始的，例如：把一个5位的无符号整数23，对一个所有位事先置0的位图，从第7位开始赋值，其结果如下所示：

+--------+--------+

|00000001|01110000|

+--------+--------+

当偏移量和整型大小是字节边界对齐时，此时与大端模式（big endian）相同，但是，当字节边界未对齐时，那么理解字节序将变得非常重要。

## BITOP operation destkey key [key ...]

起始版本：2.6.0

时间复杂度：O(N)

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。

处理不同长度的字符串

当 BITOP 处理不同长度的字符串时，较短的那个字符串所缺少的部分会被看作 0 。

空的 key 也被看作是包含 0 的字符串序列。

返回值

Integer reply

保存到 destkey 的字符串的长度，和输入 key 中最长的字符串长度相等。

例子

redis> SET key1 "foobar"

OK

redis> SET key2 "abcdef"

OK

redis> BITOP AND dest key1 key2

(integer) 6

redis> GET dest

"`bc`ab"

redis>

模式：使用 bitop 实现用户上线次数统计

BITOP是对BITCOUNT命令一个很好的补充。

不同的bitmaps进行组合操作可以获得目标bitmap以进行人口统计操作。

Fast easy realtime metrics using Redis bitmaps这篇文章介绍了一个有趣的用例。

性能

BITOP可能是一个缓慢的命令，它的时间复杂度是O(N)。 在处理长字符串时应注意一下效率问题。

对于实时的指标和统计，涉及大输入一个很好的方法是 使用bit-wise操作以避免阻塞主实例。

## BITPOS key bit [start] [end]

起始版本：2.8.7

时间复杂度：O(N)

返回字符串里面第一个被设置为1或者0的bit位。

返回一个位置，把字符串当做一个从左到右的字节数组，第一个符合条件的在位置0，其次在位置8，等等。

GETBIT 和 SETBIT 相似的也是操作字节位的命令。

默认情况下整个字符串都会被检索一次，只有在指定start和end参数(指定start和end位是可行的)，该范围被解释为一个字节的范围，而不是一系列的位。所以start=0 并且 end=2是指前三个字节范围内查找。

注意，返回的位的位置始终是从0开始的，即使使用了start来指定了一个开始字节也是这样。

和GETRANGE命令一样，start和end也可以包含负值，负值将从字符串的末尾开始计算，-1是字符串的最后一个字节，-2是倒数第二个，等等。

不存在的key将会被当做空字符串来处理。

返回值

Integer reply

命令返回字符串里面第一个被设置为1或者0的bit位。

如果我们在空字符串或者0字节的字符串里面查找bit为1的内容，那么结果将返回-1。

如果我们在字符串里面查找bit为0而且字符串只包含1的值时，将返回字符串最右边的第一个空位。如果有一个字符串是三个字节的值为0xff的字符串，那么命令BITPOS key 0将会返回24，因为0-23位都是1。

基本上，我们可以把字符串看成右边有无数个0。

然而，如果你用指定start和end范围进行查找指定值时，如果该范围内没有对应值，结果将返回-1。

例子

redis> SET mykey "\xff\xf0\x00"

OK

redis> BITPOS mykey 0 # 查找字符串里面bit值为0的位置

(integer) 12

redis> SET mykey "\x00\xff\xf0"

OK

redis> BITPOS mykey 1 0 # 查找字符串里面bit值为1从第0个字节开始的位置

(integer) 8

redis> BITPOS mykey 1 2 # 查找字符串里面bit值为1从第2个字节(12)开始的位置

(integer) 16

redis> set mykey "\x00\x00\x00"

OK

redis> BITPOS mykey 1 # 查找字符串里面bit值为1的位置

(integer) -1

redis>

## BLPOP key [key ...] timeout

起始版本：2.0.0

时间复杂度：O(1)

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。

非阻塞行为

当 BLPOP 被调用时，如果给定 key 内至少有一个非空列表，那么弹出遇到的第一个非空列表的头元素，并和被弹出元素所属的列表的名字 key 一起，组成结果返回给调用者。

当存在多个给定 key 时， BLPOP 按给定 key 参数排列的先后顺序，依次检查各个列表。 我们假设 key list1 不存在，而 list2 和 list3 都是非空列表。考虑以下的命令：

BLPOP list1 list2 list3 0

BLPOP 保证返回一个存在于 list2 里的元素（因为它是从 list1 –> list2 –> list3 这个顺序查起的第一个非空列表）。

阻塞行为

如果所有给定 key 都不存在或包含空列表，那么 BLPOP 命令将阻塞连接， 直到有另一个客户端对给定的这些 key 的任意一个执行 LPUSH 或 RPUSH 命令为止。

一旦有新的数据出现在其中一个列表里，那么这个命令会解除阻塞状态，并且返回 key 和弹出的元素值。

当 BLPOP 命令引起客户端阻塞并且设置了一个非零的超时参数 timeout 的时候， 若经过了指定的 timeout 仍没有出现一个针对某一特定 key 的 push 操作，则客户端会解除阻塞状态并且返回一个 nil 的多组合值(multi-bulk value)。

timeout 参数表示的是一个指定阻塞的最大秒数的整型值。当 timeout 为 0 是表示阻塞时间无限制。

什么 key 会先被处理？是什么客户端？什么元素？优先顺序细节。

当客户端为多个 key 尝试阻塞的时候，若至少存在一个 key 拥有元素，那么返回的键值对(key/element pair)就是从左到右数第一个拥有一个或多个元素的key。 在这种情况下客户端不会被阻塞。比如对于这个例子 BLPOP key1 key2 key3 key4 0，假设 key2 和 key4 都非空， 那么就会返回 key2 里的一个元素。

当多个客户端为同一个 key 阻塞的时候，第一个被处理的客户端是等待最长时间的那个（即第一个因为该key而阻塞的客户端）。 一旦一个客户端解除阻塞那么它就不会保持任何优先级，当它因为下一个 BLPOP 命令而再次被阻塞的时候，会在处理完那些 被同个 key 阻塞的客户端后才处理它（即从第一个被阻塞的处理到最后一个被阻塞的）。

当一个客户端同时被多个 key 阻塞时，若多个 key 的元素同时可用（可能是因为事务或者某个Lua脚本向多个list添加元素）， 那么客户端会解除阻塞，并使用第一个接收到 push 操作的 key（假设它拥有足够的元素为我们的客户端服务，因为有可能存在其他客户端同样是被这个key阻塞着）。 从根本上来说，在执行完每个命令之后，Redis 会把一个所有 key 都获得数据并且至少使一个客户端阻塞了的 list 运行一次。 这个 list 按照新数据的接收时间进行整理，即是从第一个接收数据的 key 到最后一个。在处理每个 key 的时候，只要这个 key 里有元素， Redis就会对所有等待这个key的客户端按照“先进先出”(FIFO)的顺序进行服务。若这个 key 是空的，或者没有客户端在等待这个 key， 那么将会去处理下一个从之前的命令或事务或脚本中获得新数据的 key，如此等等。

当多个元素被 push 进入一个 list 时 BLPOP 的行为

有时候一个 list 会在同一概念的命令的情况下接收到多个元素：

像 LPUSH mylist a b c 这样的可变 push 操作。

在对一个向同一个 list 进行多次 push 操作的 MULTI 块执行完 EXEC 语句后。

使用 Redis 2.6 或者更新的版本执行一个 Lua 脚本。

当多个元素被 push 进入一个被客户端阻塞着的 list 的时候，Redis 2.4 和 Redis 2.6 或者更新的版本所采取行为是不一样的。

对于 Redis 2.6 来说，所采取的行为是先执行多个 push 命令，然后在执行了这个命令之后再去服务被阻塞的客户端。看看下面命令顺序。

Client A: BLPOP foo 0

Client B: LPUSH foo a b c

如果上面的情况是发生在 Redis 2.6 或更高版本的服务器上，客户端 A 会接收到 c 元素，因为在 LPUSH 命令执行后，list 包含了 c,b,a 这三个元素，所以从左边取一个元素就会返回 c。

相反，Redis 2.4 是以不同的方式工作的：客户端会在 push 操作的上下文中被服务，所以当 LPUSH foo a b c 开始往 list 中 push 第一个元素，它就被传送给客户端A，也就是客户端A会接收到 a（第一个被 push 的元素）。

Redis 2.4的这种行为会在复制或者持续把数据存入AOF文件的时候引发很多问题，所以为了防止这些问题，很多更一般性的、并且在语义上更简单的行为被引入到 Redis 2.6 中。

需要注意的是，一个Lua脚本或者一个 MULTI / EXEC 块可能会 push 一堆元素进入一个 list 后，再 删除这个 list。 在这种情况下，被阻塞的客户端完全不会被服务，并且只要在执行某个单一命令、事务或者脚本后 list 中没有出现元素，它就会被继续阻塞下去。

在一个 MULTI / EXEC 事务中的 BLPOP

BLPOP 可以用于流水线（pipeline，发送多个命令并且批量读取回复），特别是当它是流水线里的最后一个命令的时候，这种设定更加有意义。

在一个 MULTI / EXEC 块里面使用 BLPOP 并没有很大意义，因为它要求整个服务器被阻塞以保证块执行时的原子性，这就阻止了其他客户端执行一个 push 操作。 因此，一个在 MULTI / EXEC 里面的 BLPOP 命令会在 list 为空的时候返回一个 nil 值，这跟超时(timeout)的时候发生的一样。

如果你喜欢科幻小说，那么想象一下时间是以无限的速度在 MULTI / EXEC 块中流逝……

返回值

多批量回复(multi-bulk-reply): 具体来说:

当没有元素的时候会弹出一个 nil 的多批量值，并且 timeout 过期。

当有元素弹出时会返回一个双元素的多批量值，其中第一个元素是弹出元素的 key，第二个元素是 value。

例子

redis> DEL list1 list2

(integer) 0

redis> RPUSH list1 a b c

(integer) 3

redis> BLPOP list1 list2 0

1) "list1"

2) "a"

可靠的队列

当 BLPOP 返回一个元素给客户端的时候，它也从 list 中把该元素移除。这意味着该元素就只存在于客户端的上下文中：如果客户端在处理这个返回元素的过程崩溃了，那么这个元素就永远丢失了。

在一些我们希望是更可靠的消息传递系统中的应用上，这可能会导致一些问题。在这种时候，请查看 BRPOPLPUSH 命令，这是 BLPOP 的一个变形，它会在把返回元素传给客户端之前先把该元素加入到一个目标 list 中。

模式：事件提醒

用来阻塞 list 的操作有可能是不同的阻塞原语。 比如在某些应用里，你也许会为了等待新元素进入 Redis Set 而阻塞队列，直到有个新元素加入到 Set 中，这样就可以在不轮询的情况下获得元素。 这就要求要有一个 SPOP 的阻塞版本，而这事实上并不可用。但是我们可以通过阻塞 list 操作轻易完成这个任务。

消费者会做的：

LOOP forever

WHILE SPOP(key) returns elements

... process elements ...

END

BRPOP helper\_key

END

而在生产者这角度我们可以这样简单地使用：

MULTI

SADD key element

LPUSH helper\_key x

EXEC

## BRPOP key [key ...] timeout

起始版本：2.0.0

时间复杂度：O(1)

BRPOP 是一个阻塞的列表弹出原语。 它是 RPOP 的阻塞版本，因为这个命令会在给定list无法弹出任何元素的时候阻塞连接。 该命令会按照给出的 key 顺序查看 list，并在找到的第一个非空 list 的尾部弹出一个元素。

请在 BLPOP 文档 中查看该命令的准确语义，因为 BRPOP 和 BLPOP 基本是完全一样的，除了它们一个是从尾部弹出元素，而另一个是从头部弹出元素。

返回值

多批量回复(multi-bulk-reply): 具体来说:

当没有元素可以被弹出时返回一个 nil 的多批量值，并且 timeout 过期。

当有元素弹出时会返回一个双元素的多批量值，其中第一个元素是弹出元素的 key，第二个元素是 value。

例子：

redis> DEL list1 list2

(integer) 0

redis> RPUSH list1 a b c

(integer) 3

redis> BRPOP list1 list2 0

1) "list1"

2) "c"

## BRPOPLPUSH source destination timeout

起始版本：2.2.0

时间复杂度：O(1)

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。

查看 RPOPLPUSH 以了解更多信息。

返回值

批量回复(bulk-reply): 元素从 source 中弹出来，并压入 destination 中。 如果达到 timeout 时限，会返回一个空的多批量回复(nil-reply)。

模式：可靠的队列

请参考RPOPLPUSH 命令文档。

模式：循环列表

请参考RPOPLPUSH 命令文档。

## CLIENT KILL [ip:port] [ID client-id] [normal|slave|pubsub] [ADDR ip:port] [SKIPME yes/no]

起始版本：2.4.0

时间复杂度：O(N) N是客户端连接数量

CLIENT KILL关闭一个指定的连接。在Redis2.8.11时可以根据客户端地址关闭指定连接，关闭方式如下：

CLIENT KILL addr:port

ip:port应该是CLIENT LIST 命令里面列出的客户端连接之一。

然而，从Redis 2.8.12开始，这个命令改为如下格式：

CLIENT KILL <filter> <value> ... ... <filter> <value>

新的格式可以根据不同属性杀死客户端而不是只按地址杀死。他们有以下一些格式：

CLIENT KILL ADDR ip:port. 和旧版的三个参数时的行为完全一样。

CLIENT KILL ID client-id. 可以通过唯一ID字段杀死一个客户端，唯一ID可以通过Redis2.8.12的CLIENT LIST命令查询。

CLIENT KILL TYPE type, 这里的 type 可以是 normal, slave, pubsub。 这将关闭所有特殊类的客户端. 请注意被认为是属于正常类的客户端将会被MONITOR命令监视到。

CLIENT KILL SKIPME yes/no. By default this option is set to yes, that is, the client calling the command will not get killed, however setting this option to no will have the effect of also killing the client calling the command.

It is possible to provide multiple filters at the same time. The command will handle multiple filters via logical AND. For example:

CLIENT KILL addr 127.0.0.1:6379 type slave

is valid and will kill only a slaves with the specified address. This format containing multiple filters is rarely useful currently.

When the new form is used the command no longer returns OK or an error, but instead the number of killed clients, that may be zero.

CLIENT KILL and Redis Sentinel

Recent versions of Redis Sentinel (Redis 2.8.12 or greater) use CLIENT KILL in order to kill clients when an instance is reconfigured, in order to force clients to perform the handshake with one Sentinel again and update its configuration.

Notes

Due to the single-threaded nature of Redis, it is not possible to kill a client connection while it is executing a command. From the client point of view, the connection can never be closed in the middle of the execution of a command. However, the client will notice the connection has been closed only when the next command is sent (and results in network error).

@return

When called with the three arguments format:

@simple-string-reply: OK if the connection exists and has been closed

When called with the filter / value format:

@integer-reply: the number of clients killed.

## CLIENT LIST

起始版本：2.4.0

时间复杂度：O(N) N是客户端连接数量

Redis CLIENT LIST命令用于返回所有连接到服务器的客户端信息和统计数据。

返回值

bulk-string-reply: 一个独特的字符串，格式如下：

每个已连接客户端对应一行（以 LF 分割）

每行字符串由一系列 属性=值（property=value） 形式的域组成，每个域之间以空格分开。

下面是各字段的含义：:

id: 唯一的64位的客户端ID(Redis 2.8.12加入)。

addr: 客户端的地址和端口

fd: 套接字所使用的文件描述符

age: 以秒计算的已连接时长

idle: 以秒计算的空闲时长

flags: 客户端 flag

db: 该客户端正在使用的数据库 ID

sub: 已订阅频道的数量

psub: 已订阅模式的数量

multi: 在事务中被执行的命令数量

qbuf: 查询缓冲区的长度（字节为单位， 0 表示没有分配查询缓冲区）

qbuf-free: 查询缓冲区剩余空间的长度（字节为单位， 0 表示没有剩余空间）

obl: 输出缓冲区的长度（字节为单位， 0 表示没有分配输出缓冲区）

oll: 输出列表包含的对象数量（当输出缓冲区没有剩余空间时，命令回复会以字符串对象的形式被入队到这个队列里）

omem: 输出缓冲区和输出列表占用的内存总量

events: 文件描述符事件

cmd: 最近一次执行的命令

客户端 flag 可以由以下部分组成：

O: 客户端是 MONITOR 模式下的附属节点（slave）

S: 客户端是一般模式下（normal）的附属节点

M: 客户端是主节点（master）

x: 客户端正在执行事务

b: 客户端正在等待阻塞事件

i: 客户端正在等待 VM I/O 操作（已废弃）

d: 一个受监视（watched）的键已被修改， EXEC 命令将失败

c: 在将回复完整地写出之后，关闭链接

u: 客户端未被阻塞（unblocked）

U: 通过Unix套接字连接的客户端

r: 客户端是只读模式的集群节点

A: 尽可能快地关闭连接

N: 未设置任何 flag

文件描述符事件可以是：

r: 客户端套接字（在事件 loop 中）是可读的（readable）

w: 客户端套接字（在事件 loop 中）是可写的（writeable）

注意

新字段会随着测试有规律的添加。某些字段将来可能会被删除。一个版本安全的Redis客户端使用这个命令时应该根据字段解析相应内容。（比如：处理未知的字段，应跳过该字段）。