# 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。

## Cluster

### CLUSTER ADDSLOTS slot [slot ...]

起始版本：3.0.0

时间复杂度：O(N) N是参数的哈希插槽总数

这个命令是用于修改某个节点上的集群配置。具体的说它把一组hash slots分配给接收命令的节点。 如果命令执行成功，节点将指定的hash slots映射到自身，节点将获得指定的hash slots，同时开始向集群广播新的配置。

需要注意：

该命令只有当所有指定的slots在接收命令的节点上还没有分配得的情况下生效。节点将 拒绝接纳已经分配到其他节点的slots（包括它自己的）。

同一个slot被指定多次的情况下命令会失败。

执行这个命令有一个副作用，如果slot作为其中一个参数设置为importing，一旦节点向自己分配该slot（以前未绑定）这个状态将会被清除。

例子

例如以下命令分配 1 2 3 slot到接收命令的节点：

> CLUSTER ADDSLOTS 1 2 3

OK

但是试图再次执行命令结果将会错误，因为slots已经被分配了。

> CLUSTER ADDSLOTS 1 2 3

ERR Slot 1 is already busy

在Redis集群中的应用

这个命令仅在cluster 模式下生效，而且作用于redis集群以下操作：

创建新集群时，ADDSLOTS用于主节点初始化分配可用的hash slots。

为了修复有未分配slots的坏集群。

有关slots的传播和警告

注意一旦一个节点为自己分配了一个slot集合，它就会开始将这个信息在心跳包的头里传播出去。然而其他节点只有在他们有slot没有被其他节点绑定或者传播的新的hash slot的配置年代大于列表中的节点时才会接受这个信息。

这意味着这个命令应该仅通过redis集群应用管理客户端例如redsi-trib谨慎使用，而且这个命令如果使用了错误的上下文会导致集群处于错误的状态或者导致数据丢失。

返回值

simple-string-reply: 如果命令执行成功返回OK，否则返回错误信息。

### CLUSTER COUNT-FAILURE-REPORTS node-id

起始版本：3.0.0

时间复杂度：O(N) N是故障报告的数量

这个命令返回指定节点的故障报告个数，故障报告是Redis Cluster用来使节点的PFAIL状态（这意味着节点不可达）晋升到FAIL状态而的方式，这意味着集群中大多数的主节点在一个事件窗口内同意节点不可达。

A few more details:

更多细节：

一个节点会用PFAIL标记一个不可达时间超过配置中的超时时间的节点，这个超时时间是 Redis Cluster 配置中的基本选项。

处于PFAIL状态的节点会将状态信息提供在心跳包的流言（gossip）部分。 failure reports, remembering that a given node said another given node is in PFAIL condition.

每当一个节点处理来自其他节点的流言（gossip）包时，该节点会建立故障报告（如果需要会刷新TTL），并且会记住发送消息包的节点所认为处于PFAIL状态下的其他节点。

每个故障报告的生存时间是节点超时时间的两倍。

如果在一段给定的事件内，一个节点被另一个节点标记为PFAIL状态，并且在相同的时间内收到了其他大多数主节点关于该节点的故障报告（如果该节点是主节点包括它自己），那么该节点的故障状态会从PFAIL晋升为FAIL，并且会广播一个消息，强制所有可达的节点将该节点标记为FAIL。

该命令返回当前节点没有过期的故障报告个数（在两倍的节点超时时间收到的）。该计数值不包含当前节点，该节点是我们要求这个计数值是以我们作为参数所传递的ID的节点，这个计数值只包含该节点从其他节点接收到的故障报告。

当Redis Cluster的故障检测器不能正常工作时，这个命令主要用来调试。

返回值

Integer reply：这个节点有效的故障报告个数。

### CLUSTER COUNTKEYSINSLOT slot

起始版本：3.0.0

时间复杂度：O(1)

返回连接节点负责的指定hash slot的key的数量。该命令只查询连接节点的数据集，所以如果连接节点指派到该hash slot会返回0。

> CLUSTER COUNTKEYSINSLOT 7000

(integer) 50341

返回值

Integer reply: 返回连接节点负责的指定hash slot的key的数量, 如果hash slot不合法则返回错误

### CLUSTER DELSLOTS slot [slot ...]

起始版本：3.0.0

时间复杂度：O(N) where N is the total number of hash slot arguments

在Redis Cluster中，每个节点都会知道哪些主节点正在负责哪些特定的哈希槽

DELSLOTS命令使一个特定的Redis Cluster节点去忘记一个主节点正在负责的哈希槽，这些哈希槽通过参数指定。

在已经接收到DELSLOTS命令的节点环境中，并且因此已经去除了指定哈希槽的关联，我们认为这些哈希槽是未绑定的 。请注意，当一个节点还没有被配置去负责他们（可以通过ADDSLOTS完成槽的分配）并且如果该节点没有收到关于谁拥有这些哈希槽的消息时（节点通过心跳包或者更新包获取消息），这些未绑定的哈希槽是自然而然本来就存在的。

如果一个节点认为一些哈希槽是未绑定的，但是从其他节点接收到一个心跳包，得知这些哈希槽已经被其他节点负责，那么会立即确立其关联关系。而且，如果接收到一个心跳包或更新包的配置纪元比当前节点的大，那么会重新建立关联。

但是，请注意：

命令只在参数指定的哈希槽已经和某些节点关联时有效。

如果同一个哈希槽被指定多次，该命令会失败。

命令执行的副作用是，因为不在负责哈希槽，节点可能会进入下线状态。

例如

以下命令会移除槽5000和槽5001与接收该命令节点的关联

> CLUSTER DELSLOTS 5000 5001

OK

Redis Cluster中的用法

命令只在集群模式下工作，并且对调试非常有用，并且当创建新的集群时，为了可以手动的协调集群配置。当前没有被redis-trib使用，并且主要为了API的完整性存在。

返回值

simple-string-reply：如果命令成功执行返回OK，否则返回一个错误。

### CLUSTER FAILOVER [FORCE|TAKEOVER]

### CLUSTER FORGET node-id

### CLUSTER GETKEYSINSLOT slot count

起始版本：3.0.0

时间复杂度：O(log(N)) where N is the number of requested keys

本命令返回存储在连接节点的指定hash slot里面的key的列表。key的最大数量通过count参数指定，所以这个API可以用作keys的批处理。

这个命令的主要是用于rehash期间slot从一个节点移动到另外一个节点。集群rehash的具体做法在Redis集群规范文档，或者你可以查询CLUSTER SETSLOT命令文档的附录。

> CLUSTER GETKEYSINSLOT 7000 3

"47344|273766|70329104160040|key\_39015"

"47344|273766|70329104160040|key\_89793"

"47344|273766|70329104160040|key\_92937"

返回值

array-reply: 返回count个key的列表。

### CLUSTER INFO

起始版本：3.0.0

时间复杂度：O(1)

CLUSTER INFO 命令使用 INFO 风格的形式展现了关于Redis集群的重要参数。下面是该命令的典型输出，后面是对每个输出项的说明。

cluster\_state:ok cluster\_slots\_assigned:16384 cluster\_slots\_ok:16384 cluster\_slots\_pfail:0 cluster\_slots\_fail:0 cluster\_known\_nodes:6 cluster\_size:3 cluster\_current\_epoch:6 cluster\_my\_epoch:2 cluster\_stats\_messages\_sent:1483972 cluster\_stats\_messages\_received:1483968

cluster\_state: ok状态表示集群可以正常接受查询请求。fail 状态表示，至少有一个哈希槽没有被绑定（说明有哈希槽没有被绑定到任意一个节点），或者在错误的状态（节点可以提供服务但是带有FAIL 标记），或者该节点无法联系到多数master节点。.

cluster\_slots\_assigned: 已分配到集群节点的哈希槽数量（不是没有被绑定的数量）。16384个哈希槽全部被分配到集群节点是集群正常运行的必要条件.

cluster\_slots\_ok: 哈希槽状态不是FAIL 和 PFAIL 的数量.

cluster\_slots\_pfail: 哈希槽状态是 PFAIL的数量。只要哈希槽状态没有被升级到FAIL状态，这些哈希槽仍然可以被正常处理。PFAIL状态表示我们当前不能和节点进行交互，但这种状态只是临时的错误状态。

cluster\_slots\_fail: 哈希槽状态是FAIL的数量。如果值不是0，那么集群节点将无法提供查询服务，除非cluster-require-full-coverage被设置为no .

cluster\_known\_nodes: 集群中节点数量，包括处于握手状态还没有成为集群正式成员的节点.

cluster\_size: 至少包含一个哈希槽且能够提供服务的master节点数量.

cluster\_current\_epoch: 集群本地Current Epoch变量的值。这个值在节点故障转移过程时有用，它总是递增和唯一的。

cluster\_my\_epoch: 当前正在使用的节点的Config Epoch值. 这个是关联在本节点的版本值.

cluster\_stats\_messages\_sent: 通过node-to-node二进制总线发送的消息数量.

cluster\_stats\_messages\_received: 通过node-to-node二进制总线接收的消息数量.

更多关于Current Epoch 和 Config Epoch变量的说明，请参考Redis集群规范文档.

@return

@bulk-string-reply: 行的格式如 <field>:<value> ,行后面跟着一个 CRLF。

### CLUSTER KEYSLOT key

起始版本：3.0.0

时间复杂度：O(N) where N is the number of bytes in the key

返回一个整数，用于标识指定键所散列到的哈希槽。该命令主要用来调试和测试，因为它通过一个API来暴露Redis底层哈希算法的实现。该命令的使用示例：

客户端库可能会使用Redis来测试他们自己的哈希算法，生成随机的键并且使用他们自己实现的算法和Redis的CLUSTER KEYSLOT命令来散列这些键，然后检查结果是否相同。

人们会使用这个命令去检查哈希槽是哪个，然后关联Redis Cluster的节点，并且负责一个给定的键。

例如

> CLUSTER KEYSLOT somekey

11058

> CLUSTER KEYSLOT foo{hash\_tag}

(integer) 2515

> CLUSTER KEYSLOT bar{hash\_tag}

(integer) 2515

注意该命令实现了完整的哈希算法，包括支持 hash tags，这是Redis Cluster键一个特殊的哈希算法，如果键名中存在左右大括号的模式，只会散列在 { 和 } 之间的字符串，为了去强制将多个键由一个节点来处理。

返回值

Integer reply：哈希槽的值。

### CLUSTER MEET ip port

起始版本：3.0.0

时间复杂度：O(1)

CLUSTER MEET命令被用来连接不同的开启集群支持的 Redis 节点，以进入工作集群。

基本的思想是每个节点默认都是相互不信任的，并且被认为是未知的节点，以便万一因为系统管理错误或地址被修改，而不太可能将多个不同的集群节点混成一个集群。

因此，为了使给定的节点能将另一个节点接收到组成 Redis Cluster 的节点列表中，这里只有两种方法：

系统管理员发送一个CLUSTER MEET命令强制一个节点去会面另一个节点。

一个已知的节点发送一个保存在 gossip 部分的节点列表，包含着未知的节点。如果接收的节点已经将发送节点信任为已知节点，它会处理 gossip 部分并且发送一个握手消息给未知的节点。

请注意，Redis Cluster 需要形成一个完整的网络（每个节点都连接着其他每个节点），但是为了创建一个集群，不需要发送形成网络所需的所有CLUSTER MEET命令。发送CLUSTER MEET消息以便每个节点能够到达其他每个节点只需通过一条已知的节点链就足够了。由于在心跳包中会交换 gossip 信息，将会创建节点间缺失的链接。

所以，如果我们通过CLUSTER MEET链接节点 A 和节点 B ，并且节点 B 和 C 有链接，那么节点 A 和节点 C 会发现他们握手和创建链接的方法。

另一个例子：如果我们想象一个由四个分别叫 A，B，C，和D 的节点组成，我们可能只发送以下一组命令给节点 A ：

CLUSTER MEET B-ip B-port

CLUSTER MEET C-ip C-port

CLUSTER MEET D-ip D-port

作为A知道和被其他所有节点知道的副作用，它将会在发送的心跳包中包含gossip部分，这将允许其他每个节点彼此都创建一个链接，即使集群很大，也能在数秒钟之内形成一个完整的网络。

而且CLUSTER MEET不必相互执行，如果发送命令给 A 以加入B ，那么就不必也发送给 B 以加入 A 。

实现细节：MEET 和 PING 包

当一个给定的节点接收到一个CLUSTER MEET消息时，命令中指定的节点仍然不知道我们发送了命令，所以为了使节点强制将接收命令的节点将它作为信任的节点接受它，它会发送MEET包而不是PING包。两个消息包有相同的格式，但是MEET强制使接收消息包的节点确认发送消息包的节点为可信任的。

返回值

simple-string-reply：如果命令执行成功返回OK。如果指定地址或端口无效则返回一个错误。

### **CLUSTER REPLICATE node-id**

### **CLUSTER RESET [HARD|SOFT]**

### **CLUSTER SAVECONFIG**

### **CLUSTER SET-CONFIG-EPOCH config-epoch**

### **CLUSTER SETSLOT slot IMPORTING|MIGRATING|STABLE|NODE [node-id]**

### **CLUSTER SLAVES node-id**

### CLUSTER SLOTS

起始版本：3.0.0

时间复杂度：O(N) where N is the total number of Cluster nodes

CLUSTER SLOTS命令返回哈希槽和Redis实例映射关系。这个命令对客户端实现集群功能非常有用，使用这个命令可以获得哈希槽与节点（由IP和端口组成）的映射关系，这样，当客户端收到（用户的）调用命令时，可以根据（这个命令）返回的信息将命令发送到正确的Redis实例.

（嵌套对象）结果数组

每一个（节点）信息:

哈希槽起始编号

哈希槽结束编号

哈希槽对应master节点，节点使用IP/Port表示

master节点的第一个副本

第二个副本

…直到所有的副本都打印出来

每个结果包含该哈希槽范围的所有存活的副本，没有存活的副本不会返回.

（每个节点信息的）第三个（行）对象一定是IP/Port形式的master节点。之后的所有IP/Port都是该哈希槽范围的Redis副本。

如果一个集群实例中的哈希槽不是连续的（例如1-400,900,1800-6000），那么哈希槽对应的master和replica副本在这些不同的哈希槽范围会出现多次。

返回值

array-reply: 描述每个哈希槽范围的包含嵌套对象的列表，嵌套对象包含 IP/Port

输出示例

127.0.0.1:7001> cluster slots

1) 1) (integer) 0

2) (integer) 4095

3) 1) "127.0.0.1"

2) (integer) 7000

4) 1) "127.0.0.1"

2) (integer) 7004

2) 1) (integer) 12288

2) (integer) 16383

3) 1) "127.0.0.1"

2) (integer) 7003

4) 1) "127.0.0.1"

2) (integer) 7007

3) 1) (integer) 4096

2) (integer) 8191

3) 1) "127.0.0.1"

2) (integer) 7001

4) 1) "127.0.0.1"

2) (integer) 7005

4) 1) (integer) 8192

2) (integer) 12287

3) 1) "127.0.0.1"

2) (integer) 7002

4) 1) "127.0.0.1"

2) (integer) 7006

### READONLY

起始版本：3.0.0

时间复杂度：O(1)

开启与 Redis Cluster 从节点连接的读请求

通常，从节点将重定向客户端到认证过的主节点，以获取在指定命令中所涉及的哈希槽，然而客户端能通过READONLY命令将从节点设置为只读模式。

READONLY告诉 Redis Cluster 从节点客户端愿意读取可能过时的数据并且对写请求不感兴趣。

当连接处于只读模式，只有操作涉及到该从节点的主节点不服务的键时，集群将会发送一个重定向给客户端。这可能是因为：

客户端发送一个有关这个从节点的主节点不服务哈希槽的命令。

集群被重新配置（例如重新分片）并且从节点不在服务给定哈希槽的命令。

返回值

simple-string-reply

### READWRITE

起始版本：3.0.0

时间复杂度：O(1)

禁止与Redis Cluster从节点连接的读请求。

默认情况下禁止Redis Cluster从节点的读请求，但是可以使用READONLY去在每个连接的基础上改变这个行为，READWRITE命令将连接的只读模式重置为读写模式。

返回值

simple-string-reply

﻿

## Connection

### 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

### ECHO message

起始版本：1.0.0

返回消息

返回值

Bulk reply

例子

redis> ECHO HelloWorld!

HelloWorld!

redis>

### PING

起始版本：1.0.0

如果后面没有参数时返回PONG，否则会返回后面带的参数。

这个命令经常用来测试一个连接是否还是可用的，或者用来测试一个连接的延时。

如果客户端处于频道订阅模式下，它将是一个multi-bulk返回，第一次时返回”pong”，之后返回空（empty bulk），除非命令后面更随了参数。

返回值

simple-string-reply

例子

redis> PING

PONG

redis> PING "hello world"

"hello world"

redis>

### QUIT

起始版本：1.0.0

请求服务器关闭连接。连接将会尽可能快的将未完成的客户端请求完成处理。

返回值

simple-string-reply: 始终返回 OK.

### SELECT index

起始版本：1.0.0

选择一个数据库，下标值从0开始，一个新连接默认连接的数据库是DB0。

返回值

simple-string-reply

## Geo

### GEOADD key longitude latitude member [longitude latitude member ...]

时间复杂度：每一个元素添加是O(log(N)) ，N是sorted set的元素数量。

将指定的地理空间位置（纬度、经度、名称）添加到指定的key中。这些数据将会存储到sorted set这样的目的是为了方便使用GEORADIUS或者GEORADIUSBYMEMBER命令对数据进行半径查询等操作。

该命令以采用标准格式的参数x,y,所以经度必须在纬度之前。这些坐标的限制是可以被编入索引的，区域面积可以很接近极点但是不能索引。具体的限制，由EPSG:900913 / EPSG:3785 / OSGEO:41001 规定如下：

有效的经度从-180度到180度。

有效的纬度从-85.05112878度到85.05112878度。

当坐标位置超出上述指定范围时，该命令将会返回一个错误。

它是如何工作的？

sorted set使用一种称为Geohash的技术进行填充。经度和纬度的位是交错的，以形成一个独特的52位整数. 我们知道，一个sorted set 的double score可以代表一个52位的整数，而不会失去精度。

这种格式允许半径查询检查的1 + 8个领域需要覆盖整个半径，并丢弃元素以外的半径。通过计算该区域的范围，通过计算所涵盖的范围，从不太重要的部分的排序集的得分，并计算得分范围为每个区域的sorted set中的查询。

使用什么样的地球模型（Earth model）？

这只是假设地球是一个球体，因为使用的距离公式是Haversine公式。这个公式仅适用于地球，而不是一个完美的球体。当在社交网站和其他大多数需要查询半径的应用中使用时，这些偏差都不算问题。但是，在最坏的情况下的偏差可能是0.5%，所以一些地理位置很关键的应用还是需要谨慎考虑。

返回值

integer-reply, 具体的:

添加到sorted set元素的数目，但不包括已更新score的元素。

例子

redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"

(integer) 2

redis> GEODIST Sicily Palermo Catania

"166274.15156960039"

redis> GEORADIUS Sicily 15 37 100 km

1) "Catania"

redis> GEORADIUS Sicily 15 37 200 km

1) "Palermo"

2) "Catania"

redis>

### GEODIST key member1 member2 [unit]

时间复杂度：O(log(N))

返回两个给定位置之间的距离。

如果两个位置之间的其中一个不存在， 那么命令返回空值。

指定单位的参数 unit 必须是以下单位的其中一个：

m 表示单位为米。

km 表示单位为千米。

mi 表示单位为英里。

ft 表示单位为英尺。

如果用户没有显式地指定单位参数， 那么 GEODIST 默认使用米作为单位。

GEODIST 命令在计算距离时会假设地球为完美的球形， 在极限情况下， 这一假设最大会造成 0.5% 的误差。

返回值

bulk-string-reply, 具体的:

计算出的距离会以双精度浮点数的形式被返回。 如果给定的位置元素不存在， 那么命令返回空值。

例子

redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"

(integer) 2

redis> GEODIST Sicily Palermo Catania

"166274.15156960039"

redis> GEODIST Sicily Palermo Catania km

"166.27415156960038"

redis> GEODIST Sicily Palermo Catania mi

"103.31822459492736"

redis> GEODIST Sicily Foo Bar

(nil)

redis>

### GEOHASH key member [member ...]

时间复杂度：O(log(N))

返回一个或多个位置元素的 Geohash 表示。

通常使用表示位置的元素使用不同的技术，使用Geohash位置52点整数编码。由于编码和解码过程中所使用的初始最小和最大坐标不同，编码的编码也不同于标准。此命令返回一个标准的Geohash，在维基百科和geohash.org网站都有相关描述

Geohash字符串属性

该命令将返回11个字符的Geohash字符串，所以没有精度Geohash，损失相比，使用内部52位表示。返回的geohashes具有以下特性：

他们可以缩短从右边的字符。它将失去精度，但仍将指向同一地区。

它可以在 geohash.org 网站使用，网址 http://geohash.org/<geohash-string>。查询例子：http://geohash.org/sqdtr74hyu0.

与类似的前缀字符串是附近，但相反的是不正确的，这是可能的，用不同的前缀字符串附近。

返回值

integer-reply, 具体的:

一个数组， 数组的每个项都是一个 geohash 。 命令返回的 geohash 的位置与用户给定的位置元素的位置一一对应。

例子

redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"

(integer) 2

redis> GEOHASH Sicily Palermo Catania

1) "sqc8b49rny0"

2) "sqdtr74hyu0"

redis>

### GEOPOS key member [member ...]

时间复杂度：O(log(N))

从key里返回所有给定位置元素的位置（经度和纬度）。

给定一个sorted set表示的空间索引，密集使用 geoadd 命令，它以获得指定成员的坐标往往是有益的。当空间索引填充通过 geoadd 的坐标转换成一个52位Geohash，所以返回的坐标可能不完全以添加元素的，但小的错误可能会出台。

因为 GEOPOS 命令接受可变数量的位置元素作为输入， 所以即使用户只给定了一个位置元素， 命令也会返回数组回复。

返回值

array-reply, 具体的:

GEOPOS 命令返回一个数组， 数组中的每个项都由两个元素组成： 第一个元素为给定位置元素的经度， 而第二个元素则为给定位置元素的纬度。

当给定的位置元素不存在时， 对应的数组项为空值。

例子

redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"

(integer) 2

redis> GEOPOS Sicily Palermo Catania NonExisting

1) 1) "13.361389338970184"

2) "38.115556395496299"

2) 1) "15.087267458438873"

2) "37.50266842333162"

3) (nil)

redis>

### GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]

时间复杂度：O(N+log(M)) where N is the number of elements inside the bounding box of the circular area delimited by center and radius and M is the number of items inside the index.

以给定的经纬度为中心， 返回键包含的位置元素当中， 与中心的距离不超过给定最大距离的所有位置元素。

范围可以使用以下其中一个单位：

m 表示单位为米。

km 表示单位为千米。

mi 表示单位为英里。

ft 表示单位为英尺。

在给定以下可选项时， 命令会返回额外的信息：

WITHDIST: 在返回位置元素的同时， 将位置元素与中心之间的距离也一并返回。 距离的单位和用户给定的范围单位保持一致。

WITHCOORD: 将位置元素的经度和维度也一并返回。

WITHHASH: 以 52 位有符号整数的形式， 返回位置元素经过原始 geohash 编码的有序集合分值。 这个选项主要用于底层应用或者调试， 实际中的作用并不大。

命令默认返回未排序的位置元素。 通过以下两个参数， 用户可以指定被返回位置元素的排序方式：

ASC: 根据中心的位置， 按照从近到远的方式返回位置元素。

DESC: 根据中心的位置， 按照从远到近的方式返回位置元素。

在默认情况下， GEORADIUS 命令会返回所有匹配的位置元素。 虽然用户可以使用 COUNT <count> 选项去获取前 N 个匹配元素， 但是因为命令在内部可能会需要对所有被匹配的元素进行处理， 所以在对一个非常大的区域进行搜索时， 即使只使用 COUNT 选项去获取少量元素， 命令的执行速度也可能会非常慢。 但是从另一方面来说， 使用 COUNT 选项去减少需要返回的元素数量， 对于减少带宽来说仍然是非常有用的。

返回值

bulk-string-reply, 具体的:

在没有给定任何 WITH 选项的情况下， 命令只会返回一个像 [“New York”,”Milan”,”Paris”] 这样的线性（linear）列表。

在指定了 WITHCOORD 、 WITHDIST 、 WITHHASH 等选项的情况下， 命令返回一个二层嵌套数组， 内层的每个子数组就表示一个元素。

在返回嵌套数组时， 子数组的第一个元素总是位置元素的名字。 至于额外的信息， 则会作为子数组的后续元素， 按照以下顺序被返回：

以浮点数格式返回的中心与位置元素之间的距离， 单位与用户指定范围时的单位一致。

geohash 整数。

由两个元素组成的坐标，分别为经度和纬度。

举个例子， GEORADIUS Sicily 15 37 200 km WITHCOORD WITHDIST 这样的命令返回的每个子数组都是类似以下格式的：

["Palermo","190.4424",["13.361389338970184","38.115556395496299"]]

例子

redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"

(integer) 2

redis> GEORADIUS Sicily 15 37 200 km WITHDIST

1) 1) "Palermo"

2) "190.4424"

2) 1) "Catania"

2) "56.4413"

redis> GEORADIUS Sicily 15 37 200 km WITHCOORD

1) 1) "Palermo"

2) 1) "13.361389338970184"

2) "38.115556395496299"

2) 1) "Catania"

2) 1) "15.087267458438873"

2) "37.50266842333162"

redis> GEORADIUS Sicily 15 37 200 km WITHDIST WITHCOORD

1) 1) "Palermo"

2) "190.4424"

3) 1) "13.361389338970184"

2) "38.115556395496299"

2) 1) "Catania"

2) "56.4413"

3) 1) "15.087267458438873"

2) "37.50266842333162"

redis>

### GEORADIUSBYMEMBER key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]

时间复杂度：O(N+log(M)) where N is the number of elements inside the bounding box of the circular area delimited by center and radius and M is the number of items inside the index.

这个命令和 GEORADIUS 命令一样， 都可以找出位于指定范围内的元素， 但是 GEORADIUSBYMEMBER 的中心点是由给定的位置元素决定的， 而不是像 GEORADIUS 那样， 使用输入的经度和纬度来决定中心点

指定成员的位置被用作查询的中心。

关于 GEORADIUSBYMEMBER 命令的更多信息， 请参考 GEORADIUS 命令的文档。

例子

redis> GEOADD Sicily 13.583333 37.316667 "Agrigento"

(integer) 1

redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"

(integer) 2

redis> GEORADIUSBYMEMBER Sicily Agrigento 100 km

1) "Agrigento"

2) "Palermo"

redis>

## Hashes

### HDEL key field [field ...]

起始版本：2.0.0

时间复杂度：O(N) N是被删除的字段数量。

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。

返回值

integer-reply： 返回从哈希集中成功移除的域的数量，不包括指出但不存在的那些域

历史

在 2.4及以上版本中 ：可接受多个域作为参数。小于 2.4版本 的 Redis 每次调用只能移除一个域 要在早期版本中以原子方式从哈希集中移除多个域，可用 MULTI/EXEC块。

例子

redis> HSET myhash field1 "foo"

(integer) 1

redis> HDEL myhash field1

(integer) 1

redis> HDEL myhash field2

(integer) 0

redis>

### HEXISTS key field

起始版本：2.0.0

时间复杂度：O(1)

返回hash里面field是否存在

返回值

integer-reply, 含义如下：

1 hash里面包含该field。

0 hash里面不包含该field或者key不存在。

例子

redis> HSET myhash field1 "foo"

(integer) 1

redis> HEXISTS myhash field1

(integer) 1

redis> HEXISTS myhash field2

(integer) 0

redis>

### HGET key field

起始版本：2.0.0

时间复杂度：O(1)

返回 key 指定的哈希集中该字段所关联的值

返回值

bulk-string-reply：该字段所关联的值。当字段不存在或者 key 不存在时返回nil。

例子

redis> HSET myhash field1 "foo"

(integer) 1

redis> HGET myhash field1

"foo"

redis> HGET myhash field2

(nil)

redis>

### HGETALL key

起始版本：2.0.0

时间复杂度：O(N) where N is the size of the hash.

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

返回值

array-reply：哈希集中字段和值的列表。当 key 指定的哈希集不存在时返回空列表。

例子

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HSET myhash field2 "World"

(integer) 1

redis> HGETALL myhash

1) "field1"

2) "Hello"

3) "field2"

4) "World"

redis>

### HINCRBY key field increment

起始版本：2.0.0

时间复杂度：O(1)

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

HINCRBY 支持的值的范围限定在 64位 有符号整数

返回值

integer-reply：增值操作执行后的该字段的值。

例子

redis> HSET myhash field 5

(integer) 1

redis> HINCRBY myhash field 1

(integer) 6

redis> HINCRBY myhash field -1

(integer) 5

redis> HINCRBY myhash field -10

(integer) -5

redis>

### HINCRBYFLOAT key field increment

起始版本：2.6.0

时间复杂度：O(1)

为指定key的hash的field字段值执行float类型的increment加。如果field不存在，则在执行该操作前设置为0.如果出现下列情况之一，则返回错误：

field的值包含的类型错误(不是字符串)。

当前field或者increment不能解析为一个float类型。

此命令的确切行为与INCRBYFLOAT命令相同，请参阅INCRBYFLOAT命令获取更多信息。

返回值

bulk-string-reply： field执行increment加后的值

例子

redis> HSET mykey field 10.50

(integer) 1

redis> HINCRBYFLOAT mykey field 0.1

"10.6"

redis> HSET mykey field 5.0e3

(integer) 0

redis> HINCRBYFLOAT mykey field 2.0e2

"5200"

redis>

实现细节

该命令始终是在复制和模仿HSET，因此，在底层的浮点数运算不会出现数据不一致性问题。

### HKEYS key

起始版本：2.0.0

时间复杂度：O(N) where N is the size of the hash.

返回 key 指定的哈希集中所有字段的名字。

返回值

array-reply：哈希集中的字段列表，当 key 指定的哈希集不存在时返回空列表。

例子

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HSET myhash field2 "World"

(integer) 1

redis> HKEYS myhash

1) "field1"

2) "field2"

redis>

### HLEN key

起始版本：2.0.0

时间复杂度：O(1)

返回 key 指定的哈希集包含的字段的数量。

返回值

integer-reply： 哈希集中字段的数量，当 key 指定的哈希集不存在时返回 0

例子

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HSET myhash field2 "World"

(integer) 1

redis> HLEN myhash

(integer) 2

redis>

### HMGET key field [field ...]

起始版本：2.0.0

时间复杂度：O(N) where N is the number of fields being requested.

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

返回值

array-reply：含有给定字段及其值的列表，并保持与请求相同的顺序。

例子

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HSET myhash field2 "World"

(integer) 1

redis> HMGET myhash field1 field2 nofield

1) "Hello"

2) "World"

3) (nil)

redis>

### HMSET key field value [field value ...]

起始版本：2.0.0

时间复杂度：O(N) where N is the number of fields being set.

设置 key 指定的哈希集中指定字段的值。该命令将重写所有在哈希集中存在的字段。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

返回值

simple-string-reply

例子

redis> HMSET myhash field1 "Hello" field2 "World"

OK

redis> HGET myhash field1

"Hello"

redis> HGET myhash field2

"World"

redis>

### HSCAN key cursor [MATCH pattern] [COUNT count]

起始版本：2.8.0

时间复杂度：O(1) for every call. O(N) for a complete iteration, including enough command calls for the cursor to return back to 0. N is the number of elements inside the collection..

请参考 SCAN命令， HSCAN与之类似 。

### HSET key field value

起始版本：2.0.0

时间复杂度：O(1)

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。

返回值

integer-reply：含义如下

1如果field是一个新的字段

0如果field原来在map里面已经存在

例子

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HGET myhash field1

"Hello"

redis>

### HSETNX key field value

起始版本：2.0.0

时间复杂度：O(1)

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。

返回值

integer-reply：含义如下

1：如果字段是个新的字段，并成功赋值

0：如果哈希集中已存在该字段，没有操作被执行

例子

redis> HSETNX myhash field "Hello"

(integer) 1

redis> HSETNX myhash field "World"

(integer) 0

redis> HGET myhash field

"Hello"

redis>

### HSTRLEN key field

起始版本：3.2.0

时间复杂度：O(1)

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.

返回值

integer-reply: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.

例子

redis> HMSET myhash f1 HelloWorld f2 99 f3 -256

OK

redis> HSTRLEN myhash f1

(integer) 10

redis> HSTRLEN myhash f2

(integer) 2

redis> HSTRLEN myhash f3

(integer) 4

redis>

### HVALS key

起始版本：2.0.0

时间复杂度：O(N) where N is the size of the hash.

返回 key 指定的哈希集中所有字段的值。

返回值

array-reply：哈希集中的值的列表，当 key 指定的哈希集不存在时返回空列表。

例子

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HSET myhash field2 "World"

(integer) 1

redis> HVALS myhash

1) "Hello"

2) "World"

redis>

## HyperLogLog

### PFADD key element [element ...]

起始版本：2.8.9

时间复杂度：O(1) to add every element.

将除了第一个参数以外的参数存储到以第一个参数为变量名的HyperLogLog结构中.

这个命令的一个副作用是它可能会更改这个HyperLogLog的内部来反映在每添加一个唯一的对象时估计的基数(集合的基数).

如果一个HyperLogLog的估计的近似基数在执行命令过程中发了变化， PFADD 返回1，否则返回0，如果指定的key不存在，这个命令会自动创建一个空的HyperLogLog结构（指定长度和编码的字符串）.

如果在调用该命令时仅提供变量名而不指定元素也是可以的，如果这个变量名存在，则不会有任何操作，如果不存在，则会创建一个数据结构（返回1）.

了解更多HyperLogLog数据结构，请查阅PFCOUNT命令页面.

返回值

integer-reply

如果 HyperLogLog 的内部被修改了,那么返回 1,否则返回 0 .

例子

redis> PFADD hll a b c d e f g

(integer) 1

redis> PFCOUNT hll

(integer) 7

redis>

### PFCOUNT key [key ...]

起始版本：2.8.9

时间复杂度：O(1) with every small average constant times when called with a single key. O(N) with N being the number of keys, and much bigger constant times, when called with multiple keys.

当参数为一个key时,返回存储在HyperLogLog结构体的该变量的近似基数，如果该变量不存在,则返回0.

当参数为多个key时，返回这些HyperLogLog并集的近似基数，这个值是将所给定的所有key的HyperLoglog结构合并到一个临时的HyperLogLog结构中计算而得到的.

HyperLogLog可以使用固定且很少的内存（每个HyperLogLog结构需要12K字节再加上key本身的几个字节）来存储集合的唯一元素.

返回的可见集合基数并不是精确值， 而是一个带有 0.81% 标准错误（standard error）的近似值.

例如为了记录一天会执行多少次各不相同的搜索查询， 一个程序可以在每次执行搜索查询时调用一次PFADD， 并通过调用PFCOUNT命令来获取这个记录的近似结果.

注意: 这个命令的一个副作用是可能会导致HyperLogLog内部被更改，出于缓存的目的,它会用8字节的来记录最近一次计算得到基数,所以PFCOUNT命令在技术上是个写命令.

返回值

integer-reply:

PFADD添加的唯一元素的近似数量.

例子

redis> PFADD hll foo bar zap

(integer) 1

redis> PFADD hll zap zap zap

(integer) 0

redis> PFADD hll foo bar

(integer) 0

redis> PFCOUNT hll

(integer) 3

redis> PFADD some-other-hll 1 2 3

(integer) 1

redis> PFCOUNT hll some-other-hll

(integer) 6

redis>

性能

当调用PFCOUNT命令时指定一个key为参数,性能表现很好，甚至和处理一个HyperLogLog所需要的时间一样短.这可能和PFCOUNT命令能够直接使用缓存的的估计基数有关，大多数的PFADD也不会更新任何寄存器，所以这个值也很少被更改.理论上能达到每秒几百次操作.

当调用PFCOUNT命令时指定多个key,由于要在多个HperLogLog结构中执行一比较慢合并操作,而且这个通过并集计算得到的基数是不能够被缓存, PFCOUNT命令还要消耗毫秒量级的时间来进行多个key的并集操作，消耗的时间会比较长一些，所以不要滥用这种多个key的方式.

使用者需要明白这个命令来处理1个key和多个key执行的语义是不同的，并且执行的性能也不相同.

更多的信息请参考这篇文章. 源代码 hyperloglog.c文件也很简单易理解， 包含了稀松与密集两种实现的编码.

### PFMERGE destkey sourcekey [sourcekey ...]

起始版本：2.8.9

时间复杂度：O(N) to merge N HyperLogLogs, but with high constant times.

将多个 HyperLogLog 合并（merge）为一个 HyperLogLog ， 合并后的 HyperLogLog 的基数接近于所有输入 HyperLogLog 的可见集合（observed set）的并集.

合并得出的 HyperLogLog 会被储存在目标变量（第一个参数）里面， 如果该键并不存在， 那么命令在执行之前， 会先为该键创建一个空的.

返回值

simple-string-reply: 这个命令只会返回 OK.

例子

redis> PFADD hll1 foo bar zap a

(integer) 1

redis> PFADD hll2 a b c foo

(integer) 1

redis> PFMERGE hll3 hll1 hll2

OK

redis> PFCOUNT hll3

(integer) 6

redis>

## Keys

### DEL key [key ...]

起始版本：1.0.0

时间复杂度：O(N) 将要被删除的key的数量，当删除的key是字符串以外的复杂数据类型时比如List,Set,Hash删除这个key的时间复杂度是O(1)。

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。

返回值

integer-reply： 被删除的keys的数量

例子

redis> SET key1 "Hello"

OK

redis> SET key2 "World"

OK

redis> DEL key1 key2 key3

(integer) 2

redis>

### DUMP key

起始版本：2.6.0

时间复杂度：O(1) to access the key and additional O(N\*M) to serialized it, where N is the number of Redis objects composing the value and M their average size. For small string values the time complexity is thus O(1)+O(1\*M) where M is small, so simply O(1).

序列化给定 key ，并返回被序列化的值，使用 RESTORE 命令可以将这个值反序列化为 Redis 键。

序列化生成的值有以下几个特点：

它带有 64 位的校验和，用于检测错误，RESTORE 在进行反序列化之前会先检查校验和。

值的编码格式和 RDB 文件保持一致。

RDB 版本会被编码在序列化值当中，如果因为 Redis 的版本不同造成 RDB 格式不兼容，那么 Redis 会拒绝对这个值进行反序列化操作。

序列化的值不包括任何生存时间信息。

返回值

如果 key 不存在，那么返回 nil。</br> 否则，返回序列化之后的值。

例子

redis> SET mykey 10

OK

redis> DUMP mykey

"\u0000\xC0\n\u0006\u0000\xF8r?\xC5\xFB\xFB\_("

redis>

### EXISTS key [key ...]

起始版本：1.0.0

时间复杂度：O(1)

返回key是否存在。

返回值

integer-reply，如下的整数结果

1 如果key存在

0 如果key不存在

例子

redis> SET key1 "Hello"

OK

redis> EXISTS key1

(integer) 1

redis> EXISTS key2

(integer) 0

redis>

### EXPIRE key seconds

起始版本：1.0.0

时间复杂度：O(1)

设置key的过期时间，超过时间后，将会自动删除该key。在Redis的术语中一个key的相关超时是不确定的。

超时后只有对key执行DEL命令或者SET命令或者GETSET时才会清除。 这意味着，从概念上讲所有改变key的值的操作都会使他清除。 例如，INCR递增key的值，执行LPUSH操作，或者用HSET改变hash的field所有这些操作都会触发删除动作。

使用PERSIST命令可以清除超时，使其变成一个永久的key。

如果key被RENAME命令修改，相关的超时时间会转移到新key上面。

如果key被RENAME命令修改，比如原来就存在Key\_A,然后调用RENAME Key\_B Key\_A命令，这时不管原来Key\_A是永久的还是设置为超时的，都会由Key\_B的有效期状态覆盖。

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。有很多应用有这种业务场景，例如记录会话的session。

返回值

integer-reply, 具体的:

1 如果成功设置过期时间。

0 如果key不存在或者不能设置过期时间。

例子

redis> SET mykey "Hello"

OK

redis> EXPIRE mykey 10

(integer) 1

redis> TTL mykey

(integer) 10

redis> SET mykey "Hello World"

OK

redis> TTL mykey

(integer) -1

redis>

案例: Navigation session

想象一下，你有一个网络服务器，你对用户最近访问的N个网页感兴趣，每一个相邻的页面设置超时时间为60秒。在概念上你为这些网页添加Navigation session，如果你的用户，可能包含有趣的信息，他或她正在寻找什么样的产品，你可以推荐相关产品。

你可以使用下面的策略模型，使用这种模式：每次用户浏览网页调用下面的命令：

MULTI

RPUSH pagewviews.user:<userid> http://.....

EXPIRE pagewviews.user:<userid> 60

EXEC

如果用户60秒没有操作，这个key将会被删除，不到60秒的话，后续网页将会被继续记录。

这个案例很容易用INCR代替RPUSH

附录: Redis 过期时间

Keys的过期时间

通常Redis keys创建时没有设置相关过期时间。他们会一直存在，除非使用显示的命令移除，例如，使用DEL命令。

EXPIRE一类命令能关联到一个有额外内存开销的key。当key执行过期操作时，Redis会确保按照规定时间删除他们。

key的过期时间和永久有效性可以通过EXPIRE和PERSIST命令（或者其他相关命令）来进行更新或者删除过期时间。

过期精度

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。

过期和持久

Keys的过期时间使用Unix时间戳存储(从Redis 2.6开始以毫秒为单位)。这意味着即使Redis实例不可用，时间也是一直在流逝的。

要想过期的工作处理好，计算机必须采用稳定的时间。 如果你将RDB文件在两台时钟不同步的电脑间同步，有趣的事会发生（所有的 keys装载时就会过期）。

即使正在运行的实例也会检查计算机的时钟，例如如果你设置了一个key的有效期是1000秒，然后设置你的计算机时间为未来2000秒，这时key会立即失效，而不是等1000秒之后。

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将会从密钥空间删除。

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。

在复制AOF文件时如何处理过期

为了获得正确的行为而不牺牲一致性，当一个key过期，DEL将会随着AOF文字一起合成到所有附加的slaves。在master实例中，这种方法是集中的，并且不存在一致性错误的机会。

然而，当slaves连接到master时，不会独立过期keys（会等到master执行DEL命令），他们任然会在数据集里面存在，所以当slave当选为master时淘汰keys会独立执行，然后成为master。

### EXPIREAT key timestamp

起始版本：1.2.0

时间复杂度：O(1)

EXPIREAT 的作用和 EXPIRE类似，都用于为 key 设置生存时间。不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳 Unix timestamp 。

返回值

integer-reply，如下的整数结果

1 如果设置了过期时间 0 如果没有设置过期时间，或者不能设置过期时间

例子

redis> SET mykey "Hello"

OK

redis> EXISTS mykey

(integer) 1

redis> EXPIREAT mykey 1293840000

(integer) 1

redis> EXISTS mykey

(integer) 0

redis>

### KEYS pattern

起始版本：1.0.0

时间复杂度：O(N) with N being the number of keys in the database, under the assumption that the key names in the database and the given pattern have limited length.

查找所有符合给定模式pattern（正则表达式）的 key 。

时间复杂度为O(N)，N为数据库里面key的数量。

例如，Redis在一个有1百万个key的数据库里面执行一次查询需要的时间是40毫秒 。

警告: KEYS 的速度非常快，但在一个大的数据库中使用它仍然可能造成性能问题，如果你需要从一个数据集中查找特定的 KEYS， 你最好还是用 Redis 的集合结构 SETS 来代替。

支持的正则表达模式：

h?llo 匹配 hello, hallo 和 hxllo

h\*llo 匹配 hllo 和 heeeello

h[ae]llo 匹配 hello 和 hallo, 但是不匹配 hillo

h[^e]llo 匹配 hallo, hbllo, … 但是不匹配 hello

h[a-b]llo 匹配 hallo 和 hbllo

如果你想取消字符的特殊匹配（正则表达式，可以在它的前面加\。

返回值

array-reply: 所有符合条件的key

例子

redis> MSET one 1 two 2 three 3 four 4

OK

redis> KEYS \*o\*

1) "four"

2) "one"

3) "two"

redis> KEYS t??

1) "two"

redis> KEYS \*

1) "four"

2) "three"

3) "one"

4) "two"

redis>

### MIGRATE host port key destination-db timeout [COPY] [REPLACE]

起始版本：2.6.0

时间复杂度：This command actually executes a DUMP+DEL in the source instance, and a RESTORE in the target instance. See the pages of these commands for time complexity. Also an O(N) data transfer between the two instances is performed.

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。

命令的内部实现是这样的：它在当前实例对给定 key 执行 DUMP 命令 ，将它序列化，然后传送到目标实例，目标实例再使用 RESTORE 对数据进行反序列化，并将反序列化所得的数据添加到数据库中；当前实例就像目标实例的客户端那样，只要看到 RESTORE 命令返回 OK ，它就会调用 DEL 删除自己数据库上的 key 。

timeout 参数以毫秒为格式，指定当前实例和目标实例进行沟通的最大间隔时间。这说明操作并不一定要在 timeout 毫秒内完成，只是说数据传送的时间不能超过这个 timeout 数。

MIGRATE 命令需要在给定的时间规定内完成 IO 操作。如果在传送数据时发生 IO 错误，或者达到了超时时间，那么命令会停止执行，并返回一个特殊的错误： IOERR 。

当 IOERR 出现时，有以下两种可能：

key 可能存在于两个实例。

key 可能只存在于当前实例。

唯一不可能发生的情况就是丢失 key ，因此，如果一个客户端执行 MIGRATE, 命令，并且不幸遇上 IOERR 错误，那么这个客户端唯一要做的就是检查自己数据库上的 key 是否已经被正确地删除。

如果有其他错误发生，那么 MIGRATE 保证 key 只会出现在当前实例中。（当然，目标实例的给定数据库上可能有和 key 同名的键，不过这和 MIGRATE 命令没有关系）。

返回值

simple-string-reply: 迁移成功时返回 OK ，否则返回相应的错误。

### MOVE key db

起始版本：1.0.0

时间复杂度：O(1)

将当前数据库的 key 移动到给定的数据库 db 当中。

如果当前数据库(源数据库)和给定数据库(目标数据库)有相同名字的给定 key ，或者 key 不存在于当前数据库，那么 MOVE 没有任何效果。

因此，也可以利用这一特性，将 MOVE 当作锁(locking)原语(primitive)。

返回值

integer-reply:

移动成功返回 1

失败则返回 0

### OBJECT subcommand [arguments [arguments ...]]

起始版本：2.2.3

时间复杂度：O(1) for all the currently implemented subcommands.

OBJECT 命令可以在内部调试(debugging)给出keys的内部对象，它用于检查或者了解你的keys是否用到了特殊编码 的数据类型来存储空间z。 当redis作为缓存使用的时候，你的应用也可能用到这些由OBJECT命令提供的信息来决定应用层的key的驱逐策略(eviction policies)

OBJECT 支持多个子命令:

OBJECT REFCOUNT 该命令主要用于调试(debugging)，它能够返回指定key所对应value被引用的次数.

OBJECT ENCODING 该命令返回指定key对应value所使用的内部表示(representation)(译者注：也可以理解为数据的压缩方式).

OBJECT IDLETIME 该命令返回指定key对应的value自被存储之后空闲的时间，以秒为单位(没有读写操作的请求) ，这个值返回以10秒为单位的秒级别时间，这一点可能在以后的实现中改善

对象可以用多种方式编码:

字符串可以被编码为 raw (常规字符串) 或者int (用字符串表示64位无符号整数这种编码方式是为了节省空间).

列表类型可以被编码为ziplist 或者 linkedlist. ziplist 是为了节省较小的列表空间而设计一种特殊编码方式.

集合被编码为 intset 或者 hashtable. intset 是为了存储数字的较小集合而设计的一种特殊编码方式.

哈希表可以被编码为 zipmap 或者hashtable. zipmap 是专为了较小的哈希表而设计的一种特殊编码方式

有序集合被编码为ziplist 或者 skiplist 格式. ziplist可以表示较小的有序集合, skiplist 表示任意大小多的有序集合.

一旦你做了一个操作让redis无法再使用那些节省空间的编码方式，它将自动将那些特殊的编码类型转换为普通的编码类型.

返回值

不同的子命令会对应不同的返回值.

refcount 和 idletime 返回整数.

encoding 返回编码类型.

如果你尝试检查的参数缺失，将会返回为空

例子

redis> lpush mylist "Hello World"

(integer) 4

redis> object refcount mylist

(integer) 1

redis> object encoding mylist

"ziplist"

redis> object idletime mylist

(integer) 10

接下来的例子你可以看到redis一旦不能够实用节省空间的编码类型时编码方式的改变.

redis> set foo 1000

OK

redis> object encoding foo

"int"

redis> append foo bar

(integer) 7

redis> get foo

"1000bar"

redis> object encoding foo

"raw"

### PEXPIRE key milliseconds

起始版本：2.6.0

时间复杂度：O(1)

这个命令和EXPIRE命令的作用类似，但是它以毫秒为单位设置 key 的生存时间，而不像EXPIRE命令那样，以秒为单位。

返回值

integer-reply, 只有以下两种值:

设置成功，返回 1

key 不存在或设置失败，返回 0

例子

redis> SET mykey "Hello"

OK

redis> PEXPIRE mykey 1500

(integer) 1

redis> TTL mykey

(integer) 1

redis> PTTL mykey

(integer) 1499

redis>

### PEXPIREAT key milliseconds-timestamp

起始版本：2.6.0

时间复杂度：O(1)

PEXPIREAT 这个命令和EXPIREAT命令类似，但它以毫秒为单位设置 key 的过期 unix 时间戳，而不是像EXPIREAT那样，以秒为单位。

返回值

integer-reply, 只有以下两种值:

如果生存时间设置成功，返回 1 。

当 key 不存在或没办法设置生存时间时，返回 0 。 (查看: EXPIRE命令获取更多信息).

例子

redis> SET mykey "Hello"

OK

redis> PEXPIREAT mykey 1555555555005

(integer) 1

redis> TTL mykey

(integer) 192569170

redis> PTTL mykey

(integer) 192569169649

redis>

### PTTL key

起始版本：2.6.0

时间复杂度：O(1)

这个命令类似于TTL命令，但它以毫秒为单位返回 key 的剩余生存时间，而不是像TTL命令那样，以秒为单位。

在Redis 2.6和之前版本，如果key不存在或者key存在且无过期时间将返回-1。

从 Redis 2.8开始，错误返回值发送了如下变化：

如果key不存在返回-2

如果key存在且无过期时间返回-1

返回值

integer-reply: TTL以毫秒为单位,或者返回一个错误值 (参考上面的描述).

例子

redis> SET mykey "Hello"

OK

redis> EXPIRE mykey 1

(integer) 1

redis> PTTL mykey

(integer) 999

redis>

### RENAME key newkey

起始版本：1.0.0

时间复杂度：O(1)

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。

返回值

simple-string-reply

例子

redis> SET mykey "Hello"

OK

redis> RENAME mykey myotherkey

OK

redis> GET myotherkey

"Hello"

redis>

### RANDOMKEY

起始版本：1.0.0

时间复杂度：O(1)

从当前数据库返回一个随机的key。

返回值

bulk-string-reply:如果数据库没有任何key，返回nil，否则返回一个随机的key。

### RENAMENX key newkey

起始版本：1.0.0

时间复杂度：O(1)

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。

返回值

integer-reply：

修改成功时，返回 1 。

如果 newkey 已经存在，返回 0 。

例子

redis> SET mykey "Hello"

OK

redis> SET myotherkey "World"

OK

redis> RENAMENX mykey myotherkey

(integer) 0

redis> GET myotherkey

"World"

redis>

### RESTORE key ttl serialized-value [REPLACE]

起始版本：2.6.0

时间复杂度：O(1) to create the new key and additional O(N\*M) to reconstruct the serialized value, where N is the number of Redis objects composing the value and M their average size. For small string values the time complexity is thus O(1)+O(1\*M) where M is small, so simply O(1). However for sorted set values the complexity is O(N\*M\*log(N)) because inserting values into sorted sets is O(log(N)).

反序列化给定的序列化值，并将它和给定的 key 关联。

参数 ttl 以毫秒为单位为 key 设置生存时间；如果 ttl 为 0 ，那么不设置生存时间。

RESTORE 在执行反序列化之前会先对序列化值的 RDB 版本和数据校验和进行检查，如果 RDB 版本不相同或者数据不完整的话，那么 RESTORE 会拒绝进行反序列化，并返回一个错误。

返回值

如果反序列化成功那么返回 OK ，否则返回一个错误。

例子

redis> DEL mykey

0

redis> RESTORE mykey 0 "\n\x17\x17\x00\x00\x00\x12\x00\x00\x00\x03\x00\

x00\xc0\x01\x00\x04\xc0\x02\x00\x04\xc0\x03\x00\

xff\x04\x00u#<\xc0;.\xe9\xdd"

OK

redis> TYPE mykey

list

redis> LRANGE mykey 0 -1

1) "1"

2) "2"

3) "3"

### SCAN cursor [MATCH pattern] [COUNT count]

起始版本：2.8.0

时间复杂度：O(1) for every call. O(N) for a complete iteration, including enough command calls for the cursor to return back to 0. N is the number of elements inside the collection.

SCAN 命令及其相关的 SSCAN, HSCAN 和 ZSCAN 命令都用于增量迭代一个集合元素。

SCAN 命令用于迭代当前数据库中的key集合。

SSCAN 命令用于迭代SET集合中的元素。

HSCAN 命令用于迭代Hash类型中的键值对。

ZSCAN 命令用于迭代SortSet集合中的元素和元素对应的分值

以上列出的四个命令都支持增量式迭代，它们每次执行都只会返回少量元素，所以这些命令可以用于生产环境，而不会出现像 KEYS 或者 SMEMBERS 命令带来的可能会阻塞服务器的问题。

不过，SMEMBERS 命令可以返回集合键当前包含的所有元素， 但是对于SCAN这类增量式迭代命令来说，有可能在增量迭代过程中，集合元素被修改，对返回值无法提供完全准确的保证。

因为 SCAN, SSCAN, HSCAN 和 ZSCAN 四个命令的工作方式都非常相似， 所以这个文档会一并介绍这四个命令，需要注意的是SSCAN, HSCAN ,ZSCAN命令的第一个参数总是一个key； SCAN 命令则不需要在第一个参数提供任何key，因为它迭代的是当前数据库中的所有key。

SCAN命令的基本用法

SCAN命令是一个基于游标的迭代器。这意味着命令每次被调用都需要使用上一次这个调用返回的游标作为该次调用的游标参数，以此来延续之前的迭代过程

当SCAN命令的游标参数被设置为 0 时， 服务器将开始一次新的迭代， 而当服务器向用户返回值为 0 的游标时， 表示迭代已结束。

以下是一个 SCAN 命令的迭代过程示例 :

redis 127.0.0.1:6379> scan 0

1) "17"

2) 1) "key:12"

2) "key:8"

3) "key:4"

4) "key:14"

5) "key:16"

6) "key:17"

7) "key:15"

8) "key:10"

9) "key:3"

10) "key:7"

11) "key:1"

redis 127.0.0.1:6379> scan 17

1) "0"

2) 1) "key:5"

2) "key:18"

3) "key:0"

4) "key:2"

5) "key:19"

6) "key:13"

7) "key:6"

8) "key:9"

9) "key:11"

在上面这个例子中， 第一次迭代使用 0 作为游标， 表示开始一次新的迭代。第二次迭代使用的是第一次迭代时返回的游标 17 ，作为新的迭代参数 。

显而易见，SCAN命令的返回值 是一个包含两个元素的数组， 第一个数组元素是用于进行下一次迭代的新游标， 而第二个数组元素则是一个数组， 这个数组中包含了所有被迭代的元素。

在第二次调用 SCAN 命令时， 命令返回了游标 0 ， 这表示迭代已经结束， 整个数据集已经被完整遍历过了。

full iteration ：以 0 作为游标开始一次新的迭代， 一直调用 SCAN 命令， 直到命令返回游标 0 ， 我们称这个过程为一次完整遍历。

Scan命令的保证

SCAN命令以及其他增量式迭代命令， 在进行完整遍历的情况下可以为用户带来以下保证 ：

从完整遍历开始直到完整遍历结束期间， 一直存在于数据集内的所有元素都会被完整遍历返回； 这意味着， 如果有一个元素， 它从遍历开始直到遍历结束期间都存在于被遍历的数据集当中， 那么 SCAN 命令总会在某次迭代中将这个元素返回给用户。

同样，如果一个元素在开始遍历之前被移出集合，并且在遍历开始直到遍历结束期间都没有再加入，那么在遍历返回的元素集中就不会出现该元素。

然而因为增量式命令仅仅使用游标来记录迭代状态， 所以这些命令带有以下缺点：

同一个元素可能会被返回多次。 处理重复元素的工作交由应用程序负责， 比如说， 可以考虑将迭代返回的元素仅仅用于可以安全地重复执行多次的操作上。

如果一个元素是在迭代过程中被添加到数据集的， 又或者是在迭代过程中从数据集中被删除的， 那么这个元素可能会被返回， 也可能不会。

SCAN命令每次执行返回的元素数量

SCAN增量式迭代命令并不保证每次执行都返回某个给定数量的元素,甚至可能会返回零个元素， 但只要命令返回的游标不是 0 ， 应用程序就不应该将迭代视作结束。

不过命令返回的元素数量总是符合一定规则的， 对于一个大数据集来说， 增量式迭代命令每次最多可能会返回数十个元素；而对于一个足够小的数据集来说， 如果这个数据集的底层表示为编码数据结构（小的sets, hashes and sorted sets）， 那么增量迭代命令将在一次调用中返回数据集中的所有元素。

如果需要的话，用户可以通过增量式迭代命令提供的COUNT选项来指定每次迭代返回元素的最大值。

COUNT选项

对于增量式迭代命令不保证每次迭代所返回的元素数量，我们可以使用COUNT选项， 对命令的行为进行一定程度上的调整。COUNT 选项的作用就是让用户告知迭代命令， 在每次迭代中应该从数据集里返回多少元素。使用COUNT 选项对于对增量式迭代命令相当于一种提示， 大多数情况下这种提示都比较有效的控制了返回值的数量。

COUNT 参数的默认值为 10 。

数据集比较大时，如果没有使用MATCH 选项, 那么命令返回的元素数量通常和 COUNT 选项指定的一样， 或者比 COUNT 选项指定的数量稍多一些。

在迭代一个编码为整数集合（intset，一个只由整数值构成的小集合）、 或者编码为压缩列表（ziplist，由不同值构成的一个小哈希或者一个小有序集合）时， 增量式迭代命令通常会无视 COUNT 选项指定的值， 在第一次迭代就将数据集包含的所有元素都返回给用户。

注意: \*\*并非每次迭代都要使用相同的 COUNT 值 \*\*，用户可以在每次迭代中按自己的需要随意改变 COUNT 值， 只要记得将上次迭代返回的游标用到下次迭代里面就可以了。

MATCH 选项

类似于KEYS 命令，增量式迭代命令通过给定 MATCH 参数的方式实现了通过提供一个 glob 风格的模式参数， 让命令只返回和给定模式相匹配的元素。

以下是一个使用 MATCH 选项进行迭代的示例:

redis 127.0.0.1:6379> sadd myset 1 2 3 foo foobar feelsgood

(integer) 6

redis 127.0.0.1:6379> sscan myset 0 match f\*

1) "0"

2) 1) "foo"

2) "feelsgood"

3) "foobar"

redis 127.0.0.1:6379>

MATCH功能对元素的模式匹配工作是在命令从数据集中取出元素后和向客户端返回元素前的这段时间内进行的， 所以如果被迭代的数据集中只有少量元素和模式相匹配， 那么迭代命令或许会在多次执行中都不返回任何元素。

以下是这种情况的一个例子:

redis 127.0.0.1:6379> scan 0 MATCH \*11\*

1) "288"

2) 1) "key:911"

redis 127.0.0.1:6379> scan 288 MATCH \*11\*

1) "224"

2) (empty list or set)

redis 127.0.0.1:6379> scan 224 MATCH \*11\*

1) "80"

2) (empty list or set)

redis 127.0.0.1:6379> scan 80 MATCH \*11\*

1) "176"

2) (empty list or set)

redis 127.0.0.1:6379> scan 176 MATCH \*11\* COUNT 1000

1) "0"

2) 1) "key:611"

2) "key:711"

3) "key:118"

4) "key:117"

5) "key:311"

6) "key:112"

7) "key:111"

8) "key:110"

9) "key:113"

10) "key:211"

11) "key:411"

12) "key:115"

13) "key:116"

14) "key:114"

15) "key:119"

16) "key:811"

17) "key:511"

18) "key:11"

redis 127.0.0.1:6379>

可以看出，以上的大部分迭代都不返回任何元素。在最后一次迭代， 我们通过将 COUNT 选项的参数设置为 1000 ， 强制命令为本次迭代扫描更多元素， 从而使得命令返回的元素也变多了。

并发执行多个迭代

在同一时间， 可以有任意多个客户端对同一数据集进行迭代， 客户端每次执行迭代都需要传入一个游标， 并在迭代执行之后获得一个新的游标， 而这个游标就包含了迭代的所有状态， 因此， 服务器无须为迭代记录任何状态。

中止迭代

因为迭代的所有状态都保存在游标里面， 而服务器无须为迭代保存任何状态， 所以客户端可以在中途停止一个迭代， 而无须对服务器进行任何通知。即使有任意数量的迭代在中途停止， 也不会产生任何问题。

使用错误的游标

使用SCAN 命令传入间断的（broken）、负数、超出范围或者其他非正常的游标来执行增量式迭代并不会造成服务器崩溃， 但可能会让命令产生未定义的行为。未定义行为指的是， 增量式命令对返回值所做的保证可能会不再为真。 只有两种游标是合法的:

在开始一个新的迭代时， 游标必须为 0 。

增量式迭代命令在执行之后返回的， 用于延续迭代过程的游标。

迭代能终止的前提

增量式迭代命令所使用的算法只保证在数据集的大小有界的情况下， 迭代才会停止， 换句话说， 如果被迭代数据集的大小不断地增长的话， 增量式迭代命令可能永远也无法完成一次完整迭代。

从直觉上可以看出， 当一个数据集不断地变大时， 想要访问这个数据集中的所有元素就需要做越来越多的工作， 能否结束一个迭代取决于用户执行迭代的速度是否比数据集增长的速度更快。

返回值

SCAN, SSCAN, HSCAN 和 ZSCAN 命令都返回一个包含两个元素的 multi-bulk 回复： 回复的第一个元素是字符串表示的无符号 64 位整数（游标）， 回复的第二个元素是另一个 multi-bulk 回复， 包含了本次被迭代的元素。

SCAN 命令返回的每个元素都是一个key。

SSCAN 命令返回的每个元素都是一个集合成员。

HSCAN 命令返回的每个元素都是一个键值对，一个键值对由一个键和一个值组成。

ZSCAN命令返回的每个元素都是一个有序集合元素，一个有序集合元素由一个成员（member）和一个分值（score）组成。

另外的例子

迭代hash中的键值对：

redis 127.0.0.1:6379> hmset hash name Jack age 33

OK

redis 127.0.0.1:6379> hscan hash 0

1) "0"

2) 1) "name"

2) "Jack"

3) "age"

4) "33"

### SORT key [BY pattern] [LIMIT offset count] [GET pattern] [ASC|DESC] [ALPHA] destination

起始版本：1.0.0

时间复杂度：O(N+M\*log(M)) where N is the number of elements in the list or set to sort, and M the number of returned elements. When the elements are not sorted, complexity is currently O(N) as there is a copy step that will be avoided in next releases.

返回或存储key的list、 set 或sorted set 中的元素。默认是按照数值类型排序的，并且按照两个元素的双精度浮点数类型值进行比较。下面是SORT的最简形式：

SORT mylist

假设mylist是一个数字列表，这条命令将返回一个元素从小到大排序的相同大小列表。如果想从大到小排序，可以使用 !DESC 修饰符。

SORT mylist DESC

当 mylist 包含的是字符串值并且需要按照字典顺序排序，可以使用 ALPHA 修饰符：

SORT mylist ALPHA

假设正确地设置了环境变量 !LC\_COLLATE ，Redis可以感知UTF-8编码。

返回元素的数量可以通过 LIMIT 修饰符限制。此修饰符有一个 offset 参数，指定了跳过的元素数量；还带有一个 count 参数，指定了从 offset 开始返回的元素数量。下面的例子将会返回排序后的列表 mylist 从第0个元素（offset 是从0开始的）开始的10个元素：

SORT mylist LIMIT 0 10

几乎所有的修饰符可以一起使用。下述例子将返回按字典顺序降序排序后的前5个元素：

SORT mylist LIMIT 0 5 ALPHA DESC

通过外部key排序

有时我们需要使用外部的key作为权重来排序，而不是使用列表、集合或有序集合中实际的元素值。假设列表 mylist包含元素1、 2 和 3，分别代表了存储在object\_1、object\_2 和 object\_3中的对象的唯一ID。当这些对象关联到存储在weight\_1、 weight\_2 和 weight\_3 中的权重后， SORT 命令就能使用这些权重按照下述语句来对 mylist 排序：

SORT mylist BY weight\_\*

BY 选项带有一个模式（此例中的 weight\_\* ），用于生成用于排序的 Key 。这些key的名称指向首次与列表(本例中的1、 2 和 3)中元素的实际值出现 \*

跳过排序的元素

BY 选项可以是一个并不存在的key，这会导致 SORT 命令跳过排序操作。这在我们获取未经排序的外部key(参考下文的 GET 选项)时非常有用。

SORT mylist BY nosort

获取外部key

前面的例子仅仅是返回排序后的ID。在某些情况下，获取实际的对象而不是他们的ID更加重要(object\_1、object\_2 和 object\_3)。获取存储在一个列表、集合或者有序集合中的key可以使用以下命令：

SORT mylist BY weight\_\* GET object\_\*

GET 选项可多次使用，以便获取每一个原始列表、集合或有序集合中元素的key。

还可以通过使用特殊 # 模式获取 GET 元素本身：

SORT mylist BY weight\_\* GET object\_\* GET #

保存排序操作的结果

默认的，SORT　命令返回排序后的元素给客户端。使用　STORE　选项，可以将结果存储于一个特定的列表中，以代替返回到客户端。

SORT mylist BY weight\_\* STORE resultkey

SORT ... STORE的一种有趣应用模式，是联合　EXPIRE　超时命令返回key，以便在应用中可以缓存SORT操作的返回结果。 其他客户端将会使用已缓存的列表，代替每个请求的 SORT 调用。当key即将过期时，一个更新版本的缓存将会通过 SORT ... STORE 再次创建。

注意，为了正确实现这种模式，很重要的一点是防止多个客户端同时重建缓存。 此时需要使用一些锁（具体的使用 SETNX）。

在 BY 和 GET中使用hash

可以在hash的属性上按下述语法使用 BY 和 GET 选项：

SORT mylist BY weight\_\*->fieldname GET object\_\*->fieldname

字符串 -> 用于区分key名称和哈希属性的名称。key被替换为上面所记录的，结果key中存储的hash用于获取特定hash的属性。

返回值

array-reply: 返回排序后的元素列表

### TTL key

起始版本：1.0.0

时间复杂度：O(1)

返回key剩余的过期时间。 这种反射能力允许Redis客户端检查指定key在数据集里面剩余的有效期。

在Redis 2.6和之前版本，如果key不存在或者已过期时返回-1。

从Redis2.8开始，错误返回值的结果有如下改变：

如果key不存在或者已过期，返回 -2

如果key存在并且没有设置过期时间（永久有效），返回 -1 。

另见PTTL命令返回相同的信息，只不过他的时间单位是毫秒（仅适用于Redis 2.6及更高版本）。

返回值

Integer reply： key有效的秒数（TTL in seconds）,或者一个负值的错误 (参考上文)。

例子

redis> SET mykey "Hello"

OK

redis> EXPIRE mykey 10 # 设置mykey 10秒后过期

(integer) 1

redis> TTL mykey # 查看mykey剩余的过期时间

(integer) 10

redis>

### TYPE key

起始版本：1.0.0

时间复杂度：O(1)

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。

返回值

simple-string-reply: 返回当前key的数据类型，如果key不存在时返回none。

例子

redis> SET key1 "value"

OK

redis> LPUSH key2 "value"

(integer) 1

redis> SADD key3 "value"

(integer) 1

redis> TYPE key1

string

redis> TYPE key2

list

redis> TYPE key3

set

redis>

### WAIT numslaves timeout

起始版本：3.0.0

时间复杂度：O(1)

此命令阻塞当前客户端，直到所有以前的写命令都成功的传输和指定的slaves确认。如果超时，指定以毫秒为单位，即使指定的slaves还没有到达，命令任然返回。

命令始终返回之前写命令发送的slaves的数量，无论是在指定slaves的情况还是达到超时。

注意点:

当’WAIT’返回时，所有之前的写命令保证接收由WAIT返回的slaves的数量。

如果命令呗当做事务的一部分发送，该命令不阻塞，而是只尽快返回先前写命令的slaves的数量。

如果timeout是0那意味着永远阻塞。

由于WAIT返回的是在失败和成功的情况下的slaves的数量。客户端应该检查返回的slaves的数量是等于或更大的复制水平。

一致性（Consistency and WAIT）

Note that WAIT does not make Redis a strongly consistent store: while synchronous replication is part of a replicated state machine, it is not the only thing needed. However in the context of Sentinel or Redis Cluster failover, WAIT improves the real world data safety.

Specifically if a given write is transferred to one or more slaves, it is more likely (but not guaranteed) that if the master fails, we’ll be able to promote, during a failover, a slave that received the write: both Sentinel and Redis Cluster will do a best-effort attempt to promote the best slave among the set of available slaves.

However this is just a best-effort attempt so it is possible to still lose a write synchronously replicated to multiple slaves.

Implementation details

Since the introduction of partial resynchronization with slaves (PSYNC feature) Redis slaves asynchronously ping their master with the offset they already processed in the replication stream. This is used in multiple ways:

Detect timed out slaves.

Perform a partial resynchronization after a disconnection.

Implement WAIT.

In the specific case of the implementation of WAIT, Redis remembers, for each client, the replication offset of the produced replication stream when a given write command was executed in the context of a given client. When WAIT is called Redis checks if the specified number of slaves already acknowledged this offset or a greater one.

@return

@integer-reply: The command returns the number of slaves reached by all the writes performed in the context of the current connection.

@examples

> SET foo bar OK > WAIT 1 0 (integer) 1 > WAIT 2 1000 (integer) 1

In the following example the first call to WAIT does not use a timeout and asks for the write to reach 1 slave. It returns with success. In the second attempt instead we put a timeout, and ask for the replication of the write to two slaves. Since there is a single slave available, after one second WAIT unblocks and returns 1, the number of slaves reached.

## Lists

### 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。

### LINDEX key index

起始版本：1.0.0

时间复杂度：O(N) where N is the number of elements to traverse to get to the element at index. This makes asking for the first or the last element of the list O(1).

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，所以 0 是表示第一个元素， 1 表示第二个元素，并以此类推。 负数索引用于指定从列表尾部开始索引的元素。在这种方法下，-1 表示最后一个元素，-2 表示倒数第二个元素，并以此往前推。

当 key 位置的值不是一个列表的时候，会返回一个error。

返回值

bulk-reply：请求的对应元素，或者当 index 超过范围的时候返回 nil。

例子

redis> LPUSH mylist "World"

(integer) 1

redis> LPUSH mylist "Hello"

(integer) 2

redis> LINDEX mylist 0

"Hello"

redis> LINDEX mylist -1

"World"

redis> LINDEX mylist 3

(nil)

redis>

### LINSERT key BEFORE|AFTER pivot value

起始版本：2.2.0

时间复杂度：O(N) where N is the number of elements to traverse before seeing the value pivot. This means that inserting somewhere on the left end on the list (head) can be considered O(1) and inserting somewhere on the right end (tail) is O(N).

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。

返回值

integer-reply: 经过插入操作后的list长度，或者当 pivot 值找不到的时候返回 -1。

例子

redis> RPUSH mylist "Hello"

(integer) 1

redis> RPUSH mylist "World"

(integer) 2

redis> LINSERT mylist BEFORE "World" "There"

(integer) 3

redis> LRANGE mylist 0 -1

1) "Hello"

2) "There"

3) "World"

redis>

### LLEN key

起始版本：1.0.0

时间复杂度：O(1)

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。

返回值

integer-reply: key对应的list的长度。

例子

redis> LPUSH mylist "World"

(integer) 1

redis> LPUSH mylist "Hello"

(integer) 2

redis> LLEN mylist

(integer) 2

redis>

### LPOP key

起始版本：1.0.0

时间复杂度：O(1)

移除并且返回 key 对应的 list 的第一个元素。

返回值

bulk-string-reply: 返回第一个元素的值，或者当 key 不存在时返回 nil。

例子

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> LPOP mylist

"one"

redis> LRANGE mylist 0 -1

1) "two"

2) "three"

redis>

### LPUSH key value [value ...]

起始版本：1.0.0

时间复杂度：O(1)

将所有指定的值插入到存于 key 的列表的头部。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。

可以使用一个命令把多个元素 push 进入列表，只需在命令末尾加上多个指定的参数。元素是从最左端的到最右端的、一个接一个被插入到 list 的头部。 所以对于这个命令例子 LPUSH mylist a b c，返回的列表是 c 为第一个元素， b 为第二个元素， a 为第三个元素。

返回值

integer-reply: 在 push 操作后的 list 长度。

历史

= 2.4: 接受多个 value 参数。版本老于 2.4 的 Redis 只能每条命令 push 一个值。

例子

redis> LPUSH mylist "world"

(integer) 1

redis> LPUSH mylist "hello"

(integer) 2

redis> LRANGE mylist 0 -1

1) "hello"

2) "world"

redis>

### LPUSHX key value

起始版本：2.2.0

时间复杂度：O(1)

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。 与 LPUSH 相反，当 key 不存在的时候不会进行任何操作。

返回值

integer-reply: 在 push 操作后的 list 长度。

例子

redis> LPUSH mylist "World"

(integer) 1

redis> LPUSHX mylist "Hello"

(integer) 2

redis> LPUSHX myotherlist "Hello"

(integer) 0

redis> LRANGE mylist 0 -1

1) "Hello"

2) "World"

redis> LRANGE myotherlist 0 -1

(empty list or set)

redis>

### LRANGE key start stop

起始版本：1.0.0

时间复杂度：O(S+N) where S is the distance of start offset from HEAD for small lists, from nearest end (HEAD or TAIL) for large lists; and N is the number of elements in the specified range.

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。

在不同编程语言里，关于求范围函数的一致性

需要注意的是，如果你有一个list，里面的元素是从0到100，那么 LRANGE list 0 10 这个命令会返回11个元素，即最右边的那个元素也会被包含在内。 在你所使用的编程语言里，这一点可能是也可能不是跟那些求范围有关的函数都是一致的。（像Ruby的 Range.new，Array#slice 或者Python的 range() 函数。）

超过范围的下标

当下标超过list范围的时候不会产生error。 如果start比list的尾部下标大的时候，会返回一个空列表。 如果stop比list的实际尾部大的时候，Redis会当它是最后一个元素的下标。

返回值

array-reply: 指定范围里的列表元素。

例子

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> LRANGE mylist 0 0

1) "one"

redis> LRANGE mylist -3 2

1) "one"

2) "two"

3) "three"

redis> LRANGE mylist -100 100

1) "one"

2) "two"

3) "three"

redis> LRANGE mylist 5 10

(empty list or set)

redis>

### LREM key count value

起始版本：1.0.0

时间复杂度：O(N) where N is the length of the list.

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。

比如， LREM list -2 “hello” 会从存于 list 的列表里移除最后两个出现的 “hello”。

需要注意的是，如果list里没有存在key就会被当作空list处理，所以当 key 不存在的时候，这个命令会返回 0。

返回值

integer-reply: 被移除的元素个数。

例子

redis> RPUSH mylist "hello"

(integer) 1

redis> RPUSH mylist "hello"

(integer) 2

redis> RPUSH mylist "foo"

(integer) 3

redis> RPUSH mylist "hello"

(integer) 4

redis> LREM mylist -2 "hello"

(integer) 2

redis> LRANGE mylist 0 -1

1) "hello"

2) "foo"

redis>

### LSET key index value

起始版本：1.0.0

时间复杂度：O(N) where N is the length of the list. Setting either the first or the last element of the list is O(1).

设置 index 位置的list元素的值为 value。 更多关于 index 参数的信息，详见 LINDEX。

当index超出范围时会返回一个error。

返回值

simple-string-reply

例子

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> LSET mylist 0 "four"

OK

redis> LSET mylist -2 "five"

OK

redis> LRANGE mylist 0 -1

1) "four"

2) "five"

3) "three"

redis>

### LTRIM key start stop

起始版本：1.0.0

时间复杂度：O(N) where N is the number of elements to be removed by the operation.

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。start 和 stop 都是由0开始计数的， 这里的 0 是列表里的第一个元素（表头），1 是第二个元素，以此类推。

例如： LTRIM foobar 0 2 将会对存储在 foobar 的列表进行修剪，只保留列表里的前3个元素。

start 和 end 也可以用负数来表示与表尾的偏移量，比如 -1 表示列表里的最后一个元素， -2 表示倒数第二个，等等。

超过范围的下标并不会产生错误：如果 start 超过列表尾部，或者 start > end，结果会是列表变成空表（即该 key 会被移除）。 如果 end 超过列表尾部，Redis 会将其当作列表的最后一个元素。

LTRIM 的一个常见用法是和 LPUSH / RPUSH 一起使用。 例如：

LPUSH mylist someelement

LTRIM mylist 0 99

这一对命令会将一个新的元素 push 进列表里，并保证该列表不会增长到超过100个元素。这个是很有用的，比如当用 Redis 来存储日志。 需要特别注意的是，当用这种方式来使用 LTRIM 的时候，操作的复杂度是 O(1) ， 因为平均情况下，每次只有一个元素会被移除。

返回值

simple-string-reply

例子

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> LTRIM mylist 1 -1

OK

redis> LRANGE mylist 0 -1

1) "two"

2) "three"

redis>

### RPOP key

起始版本：1.0.0

时间复杂度：O(1)

移除并返回存于 key 的 list 的最后一个元素。

返回值

bulk-string-reply: 最后一个元素的值，或者当 key 不存在的时候返回 nil。

例子

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> RPOP mylist

"three"

redis> LRANGE mylist 0 -1

1) "one"

2) "two"

redis>

### RPOPLPUSH source destination

起始版本：1.2.0

时间复杂度：O(1)

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）。

例如：假设 source 存储着列表 a,b,c， destination存储着列表 x,y,z。 执行 RPOPLPUSH 得到的结果是 source 保存着列表 a,b ，而 destination 保存着列表 c,x,y,z。

如果 source 不存在，那么会返回 nil 值，并且不会执行任何操作。 如果 source 和 destination 是同样的，那么这个操作等同于移除列表最后一个元素并且把该元素放在列表头部， 所以这个命令也可以当作是一个旋转列表的命令。

返回值

bulk-string-reply: 被移除和放入的元素

例子

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> RPOPLPUSH mylist myotherlist

"three"

redis> LRANGE mylist 0 -1

1) "one"

2) "two"

redis> LRANGE myotherlist 0 -1

1) "three"

redis>

模式：安全的队列

Redis通常都被用做一个处理各种后台工作或消息任务的消息服务器。 一个简单的队列模式就是：生产者把消息放入一个列表中，等待消息的消费者用 RPOP 命令（用轮询方式）， 或者用 BRPOP 命令（如果客户端使用阻塞操作会更好）来得到这个消息。

然而，因为消息有可能会丢失，所以这种队列并是不安全的。例如，当接收到消息后，出现了网络问题或者消费者端崩溃了， 那么这个消息就丢失了。

RPOPLPUSH (或者其阻塞版本的 BRPOPLPUSH） 提供了一种方法来避免这个问题：消费者端取到消息的同时把该消息放入一个正在处理中的列表。 当消息被处理了之后，该命令会使用 LREM 命令来移除正在处理中列表中的对应消息。

另外，可以添加一个客户端来监控这个正在处理中列表，如果有某些消息已经在这个列表中存在很长时间了（即超过一定的处理时限）， 那么这个客户端会把这些超时消息重新加入到队列中。

模式：循环列表

RPOPLPUSH 命令的 source 和 destination 是相同的话， 那么客户端在访问一个拥有n个元素的列表时，可以在 O(N) 时间里一个接一个获取列表元素， 而不用像 LRANGE 那样需要把整个列表从服务器端传送到客户端。

上面这种模式即使在以下两种情况下照样能很好地工作： \* 有多个客户端同时对同一个列表进行旋转（rotating）：它们会取得不同的元素，直到列表里所有元素都被访问过，又从头开始这个操作。 \* 有其他客户端在往列表末端加入新的元素。

这个模式让我们可以很容易地实现这样一个系统：有 N 个客户端，需要连续不断地对一批元素进行处理，而且处理的过程必须尽可能地快。 一个典型的例子就是服务器上的监控程序：它们需要在尽可能短的时间内，并行地检查一批网站，确保它们的可访问性。

值得注意的是，使用这个模式的客户端是易于扩展（scalable）且安全的（reliable），因为即使客户端把接收到的消息丢失了， 这个消息依然存在于队列中，等下次迭代到它的时候，由其他客户端进行处理。

### RPUSH key value [value ...]

起始版本：1.0.0

时间复杂度：O(1)

向存于 key 的列表的尾部插入所有指定的值。如果 key 不存在，那么会创建一个空的列表然后再进行 push 操作。 当 key 保存的不是一个列表，那么会返回一个错误。

可以使用一个命令把多个元素打入队列，只需要在命令后面指定多个参数。元素是从左到右一个接一个从列表尾部插入。 比如命令 RPUSH mylist a b c 会返回一个列表，其第一个元素是 a ，第二个元素是 b ，第三个元素是 c。

返回值

integer-reply: 在 push 操作后的列表长度。

历史

= 2.4: 接受多个 value 参数。 在老于 2.4 的 Redis 版本中，一条命令只能 push 单一个值。

例子

redis> RPUSH mylist "hello"

(integer) 1

redis> RPUSH mylist "world"

(integer) 2

redis> LRANGE mylist 0 -1

1) "hello"

2) "world"

redis>

### RPUSHX key value

起始版本：2.2.0

时间复杂度：O(1)

将值 value 插入到列表 key 的表尾, 当且仅当 key 存在并且是一个列表。 和 RPUSH 命令相反, 当 key 不存在时，RPUSHX 命令什么也不做。

返回值

integer-reply: RPUSHX 命令执行之后，表的长度。

例子

redis> RPUSH mylist "Hello"

(integer) 1

redis> RPUSHX mylist "World"

(integer) 2

redis> RPUSHX myotherlist "World"

(integer) 0

redis> LRANGE mylist 0 -1

1) "Hello"

2) "World"

redis> LRANGE myotherlist 0 -1

(empty list or set)

redis>

## Pub/Sub

### PSUBSCRIBE pattern [pattern ...]

起始版本：2.0.0

时间复杂度：O(N) where N is the number of patterns the client is already subscribed to.

订阅给定的模式(patterns)。

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

如果想输入普通的字符，可以在前面添加\

### PUBLISH channel message

起始版本：2.0.0

时间复杂度：O(N+M) where N is the number of clients subscribed to the receiving channel and M is the total number of subscribed patterns (by any client).

将信息 message 发送到指定的频道 channel

返回值

integer-reply: 收到消息的客户端数量。

### PUBSUB subcommand [argument [argument ...]]

起始版本：2.8.0

时间复杂度：O(N) for the CHANNELS subcommand, where N is the number of active channels, and assuming constant time pattern matching (relatively short channels and patterns). O(N) for the NUMSUB subcommand, where N is the number of requested channels. O(1) for the NUMPAT subcommand.

The PUBSUB command is an introspection command that allows to inspect the state of the Pub/Sub subsystem. It is composed of subcommands that are documented separately. The general form is:

PUBSUB <subcommand> ... args ...

PUBSUB CHANNELS [pattern]

Lists the currently active channels. An active channel is a Pub/Sub channel with one or more subscribers (not including clients subscribed to patterns).

If no pattern is specified, all the channels are listed, otherwise if pattern is specified only channels matching the specified glob-style pattern are listed.

@return

@array-reply: a list of active channels, optionally matching the specified pattern.

PUBSUB NUMSUB [channel-1 ... channel-N]

Returns the number of subscribers (not counting clients subscribed to patterns) for the specified channels.

@return

@array-reply: a list of channels and number of subscribers for every channel. The format is channel, count, channel, count, …, so the list is flat. The order in which the channels are listed is the same as the order of the channels specified in the command call.

Note that it is valid to call this command without channels. In this case it will just return an empty list.

PUBSUB NUMPAT

Returns the number of subscriptions to patterns (that are performed using the PSUBSCRIBE command). Note that this is not just the count of clients subscribed to patterns but the total number of patterns all the clients are subscribed to.

@return

@integer-reply: the number of patterns all the clients are subscribed to.

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

起始版本：2.0.0

时间复杂度：O(N+M) where N is the number of patterns the client is already subscribed and M is the number of total patterns subscribed in the system (by any client).

Unsubscribes the client from the given patterns, or from all of them if none is given.

When no patterns are specified, the client is unsubscribed from all the previously subscribed patterns. In this case, a message for every unsubscribed pattern will be sent to the client.

### SUBSCRIBE channel [channel ...]

起始版本：2.0.0

时间复杂度：O(N) where N is the number of channels to subscribe to.

订阅给指定频道的信息。

一旦客户端进入订阅状态，客户端就只可接受订阅相关的命令SUBSCRIBE、PSUBSCRIBE、UNSUBSCRIBE和PUNSUBSCRIBE除了这些命令，其他命令一律失效。

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

起始版本：2.0.0

时间复杂度：O(N) where N is the number of clients already subscribed to a channel.

Unsubscribes the client from the given channels, or from all of them if none is given.

When no channels are specified, the client is unsubscribed from all the previously subscribed channels. In this case, a message for every unsubscribed channel will be sent to the client.

## Scripting

### EVAL script numkeys key [key ...] arg [arg ...]

起始版本：2.6.0

时间复杂度：取决于脚本本身的执行的时间复杂度。

EVAL简介

EVAL 和 EVALSHA 命令是从 Redis 2.6.0 版本开始的，使用内置的 Lua 解释器，可以对 Lua 脚本进行求值。

EVAL的第一个参数是一段 Lua 5.1 脚本程序。 这段Lua脚本不需要（也不应该）定义函数。它运行在 Redis 服务器中。

EVAL的第二个参数是参数的个数，后面的参数（从第三个参数），表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 KEYS 数组，用 1 为基址的形式访问( KEYS[1] ， KEYS[2] ，以此类推)。

在命令的最后，那些不是键名参数的附加参数 arg [arg …] ，可以在 Lua 中通过全局变量 ARGV 数组访问，访问的形式和 KEYS 变量类似( ARGV[1] 、 ARGV[2] ，诸如此类)。

举例说明：

> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second

1) "key1"

2) "key2"

3) "first"

4) "second"

注：返回结果是Redis multi bulk replies的Lua数组，这是一个Redis的返回类型，您的客户端库可能会将他们转换成数组类型。

这是从一个Lua脚本中使用两个不同的Lua函数来调用Redis的命令的例子：

redis.call()

redis.pcall()

redis.call() 与 redis.pcall()很类似, 他们唯一的区别是当redis命令执行结果返回错误时， redis.call()将返回给调用者一个错误，而redis.pcall()会将捕获的错误以Lua表的形式返回

redis.call() 和 redis.pcall() 两个函数的参数可以是任意的 Redis 命令：

> eval "return redis.call('set','foo','bar')" 0

OK

需要注意的是，上面这段脚本的确实现了将键 foo 的值设为 bar 的目的，但是，它违反了 EVAL 命令的语义，因为脚本里使用的所有键都应该由 KEYS 数组来传递，就像这样：

> eval "return redis.call('set',KEYS[1],'bar')" 1 foo

OK

要求使用正确的形式来传递键(key)是有原因的，因为不仅仅是 EVAL 这个命令，所有的 Redis 命令，在执行之前都会被分析，籍此来确定命令会对哪些键进行操作。

因此，对于 EVAL 命令来说，必须使用正确的形式来传递键，才能确保分析工作正确地执行。 除此之外，使用正确的形式来传递键还有很多其他好处，它的一个特别重要的用途就是确保 Redis 集群可以将你的请求发送到正确的集群节点。 (对 Redis 集群的工作还在进行当中，但是脚本功能被设计成可以与集群功能保持兼容。)不过，这条规矩并不是强制性的， 从而使得用户有机会滥用(abuse) Redis 单实例配置(single instance configuration)，代价是这样写出的脚本不能被 Redis 集群所兼容。

Lua 脚本能返回一个值，这个值能按照一组转换规则从Lua转换成redis的返回类型。

Lua 数据类型和 Redis 数据类型之间转换

当 Lua 通过 call() 或 pcall() 函数执行 Redis 命令的时候，命令的返回值会被转换成 Lua 数据结构。 同样地，当 Lua 脚本在 Redis 内置的解释器里运行时，Lua 脚本的返回值也会被转换成 Redis 协议(protocol)，然后由 EVAL 将值返回给客户端。

数据类型之间的转换遵循这样一个设计原则：如果将一个 Redis 值转换成 Lua 值，之后再将转换所得的 Lua 值转换回 Redis 值，那么这个转换所得的 Redis 值应该和最初时的 Redis 值一样。

换句话说， Lua 类型和 Redis 类型之间存在着一一对应的转换关系。

Redis 到 Lua 的转换表。

Redis integer reply -> Lua number / Redis 整数转换成 Lua 数字

Redis bulk reply -> Lua string / Redis bulk 回复转换成 Lua 字符串

Redis multi bulk reply -> Lua table (may have other Redis data types nested) / Redis 多条 bulk 回复转换成 Lua 表，表内可能有其他别的 Redis 数据类型

Redis status reply -> Lua table with a single ok field containing the status / Redis 状态回复转换成 Lua 表，表内的 ok 域包含了状态信息

Redis error reply -> Lua table with a single err field containing the error / Redis 错误回复转换成 Lua 表，表内的 err 域包含了错误信息

Redis Nil bulk reply and Nil multi bulk reply -> Lua false boolean type / Redis 的 Nil 回复和 Nil 多条回复转换成 Lua 的布尔值 false

Lua 到 Redis 的转换表。

Lua number -> Redis integer reply (the number is converted into an integer) / Lua 数字转换成 Redis 整数

Lua string -> Redis bulk reply / Lua 字符串转换成 Redis bulk 回复

Lua table (array) -> Redis multi bulk reply (truncated to the first nil inside the Lua array if any) / Lua 表(数组)转换成 Redis 多条 bulk 回复

Lua table with a single ok field -> Redis status reply / 一个带单个 ok 域的 Lua 表，转换成 Redis 状态回复

Lua table with a single err field -> Redis error reply / 一个带单个 err 域的 Lua 表，转换成 Redis 错误回复

Lua boolean false -> Redis Nil bulk reply. / Lua 的布尔值 false 转换成 Redis 的 Nil bulk 回复

从 Lua 转换到 Redis 有一条额外的规则，这条规则没有和它对应的从 Redis 转换到 Lua 的规则：

Lua boolean true -> Redis integer reply with value of 1. / Lua 布尔值 true 转换成 Redis 整数回复中的 1

还有下面两点需要重点注意：

lua中整数和浮点数之间没有什么区别。因此，我们始终Lua的数字转换成整数的回复，这样将舍去小数部分。如果你想从Lua返回一个浮点数，你应该将它作为一个字符串（见比如ZSCORE命令）。

There is no simple way to have nils inside Lua arrays, this is a result of Lua table semantics, so when Redis converts a Lua array into Redis protocol the conversion is stopped if a nil is encountered.

以下是几个类型转换的例子：

> eval "return 10" 0

(integer) 10

> eval "return {1,2,{3,'Hello World!'}}" 0

1) (integer) 1

2) (integer) 2

3) 1) (integer) 3

2) "Hello World!"

> eval "return redis.call('get','foo')" 0

"bar"

最后一个例子展示如果是Lua直接命令调用它是如何可以从redis.call()或redis.pcall()接收到准确的返回值。

下面的例子我们可以看到浮点数和nil将怎么样处理：

> eval "return {1,2,3.3333,'foo',nil,'bar'}" 0

1) (integer) 1

2) (integer) 2

3) (integer) 3

4) "foo"

正如你看到的 3.333 被转换成了3，并且 nil后面的字符串bar没有被返回回来。

返回redis类型的辅助函数

有两个辅助函数从Lua返回Redis的类型。

redis.error\_reply(error\_string) returns an error reply. This function simply returns the single field table with the err field set to the specified string for you.

redis.status\_reply(status\_string) returns a status reply. This function simply returns the single field table with the ok field set to the specified string for you.

There is no difference between using the helper functions or directly returning the table with the specified format, so the following two forms are equivalent:

return {err="My Error"}

return redis.error\_reply("My Error")

脚本的原子性

Redis 使用单个 Lua 解释器去运行所有脚本，并且， Redis 也保证脚本会以原子性(atomic)的方式执行： 当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。 这和使用 MULTI / EXEC 包围的事务很类似。 在其他别的客户端看来，脚本的效果(effect)要么是不可见的(not visible)，要么就是已完成的(already completed)。 另一方面，这也意味着，执行一个运行缓慢的脚本并不是一个好主意。写一个跑得很快很顺溜的脚本并不难， 因为脚本的运行开销(overhead)非常少，但是当你不得不使用一些跑得比较慢的脚本时，请小心， 因为当这些蜗牛脚本在慢吞吞地运行的时候，其他客户端会因为服务器正忙而无法执行命令。

错误处理

前面的命令介绍部分说过， redis.call() 和 redis.pcall() 的唯一区别在于它们对错误处理的不同。

当 redis.call() 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚本错误，错误的输出信息会说明错误造成的原因：

> del foo

(integer) 1

> lpush foo a

(integer) 1

> eval "return redis.call('get','foo')" 0

(error) ERR Error running script (call to f\_6b1bf486c81ceb7edf3c093f4c48582e38c0e791): ERR Operation against a key holding the wrong kind of value

和 redis.call() 不同， redis.pcall() 出错时并不引发(raise)错误，而是返回一个带 err 域的 Lua 表(table)，用于表示错误：

redis 127.0.0.1:6379> EVAL "return redis.pcall('get', 'foo')" 0

(error) ERR Operation against a key holding the wrong kind of value

带宽和 EVALSHA

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗， Redis 实现了 EVALSHA 命令，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 SHA1 校验和(sum)。

EVALSHA 命令的表现如下：

如果服务器还记得给定的 SHA1 校验和所指定的脚本，那么执行这个脚本 如果服务器不记得给定的 SHA1 校验和所指定的脚本，那么它返回一个特殊的错误，提醒用户使用 EVAL 代替 EVALSHA 以下是示例：

> set foo bar

OK

> eval "return redis.call('get','foo')" 0

"bar"

> evalsha 6b1bf486c81ceb7edf3c093f4c48582e38c0e791 0

"bar"

> evalsha ffffffffffffffffffffffffffffffffffffffff 0

(error) `NOSCRIPT` No matching script. Please use [EVAL](/commands/eval).

客户端库的底层实现可以一直乐观地使用 EVALSHA 来代替 EVAL ，并期望着要使用的脚本已经保存在服务器上了，只有当 NOSCRIPT 错误发生时，才使用 EVAL 命令重新发送脚本，这样就可以最大限度地节省带宽。

这也说明了执行 EVAL 命令时，使用正确的格式来传递键名参数和附加参数的重要性：因为如果将参数硬写在脚本中，那么每次当参数改变的时候，都要重新发送脚本，即使脚本的主体并没有改变，相反，通过使用正确的格式来传递键名参数和附加参数，就可以在脚本主体不变的情况下，直接使用 EVALSHA 命令对脚本进行复用，免去了无谓的带宽消耗。

脚本缓存

Redis 保证所有被运行过的脚本都会被永久保存在脚本缓存当中，这意味着，当 EVAL 命令在一个 Redis 实例上成功执行某个脚本之后，随后针对这个脚本的所有 EVALSHA 命令都会成功执行。

刷新脚本缓存的唯一办法是显式地调用 SCRIPT FLUSH 命令，这个命令会清空运行过的所有脚本的缓存。通常只有在云计算环境中，Redis 实例被改作其他客户或者别的应用程序的实例时，才会执行这个命令。

缓存可以长时间储存而不产生内存问题的原因是，它们的体积非常小，而且数量也非常少，即使脚本在概念上类似于实现一个新命令，即使在一个大规模的程序里有成百上千的脚本，即使这些脚本会经常修改，即便如此，储存这些脚本的内存仍然是微不足道的。

事实上，用户会发现 Redis 不移除缓存中的脚本实际上是一个好主意。比如说，对于一个和 Redis 保持持久化链接(persistent connection)的程序来说，它可以确信，执行过一次的脚本会一直保留在内存当中，因此它可以在流水线中使用 EVALSHA 命令而不必担心因为找不到所需的脚本而产生错误(稍候我们会看到在流水线中执行脚本的相关问题)。

SCRIPT 命令

Redis 提供了以下几个 SCRIPT 命令，用于对脚本子系统(scripting subsystem)进行控制：

SCRIPT FLUSH ：清除所有脚本缓存 SCRIPT EXISTS ：根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存 SCRIPT LOAD ：将一个脚本装入脚本缓存，但并不立即运行它 SCRIPT KILL ：杀死当前正在运行的脚本

纯函数脚本

在编写脚本方面，一个重要的要求就是，脚本应该被写成纯函数(pure function)。

也就是说，脚本应该具有以下属性：

对于同样的数据集输入，给定相同的参数，脚本执行的 Redis 写命令总是相同的。脚本执行的操作不能依赖于任何隐藏(非显式)数据，不能依赖于脚本在执行过程中、或脚本在不同执行时期之间可能变更的状态，并且它也不能依赖于任何来自 I/O 设备的外部输入。

使用系统时间(system time)，调用像 RANDOMKEY 那样的随机命令，或者使用 Lua 的随机数生成器，类似以上的这些操作，都会造成脚本的求值无法每次都得出同样的结果。

为了确保脚本符合上面所说的属性， Redis 做了以下工作：

Lua 没有访问系统时间或者其他内部状态的命令

Redis 会返回一个错误，阻止这样的脚本运行： 这些脚本在执行随机命令之后(比如 RANDOMKEY 、 SRANDMEMBER 或 TIME 等)，还会执行可以修改数据集的 Redis 命令。如果脚本只是执行只读操作，那么就没有这一限制。注意，随机命令并不一定就指那些带 RAND 字眼的命令，任何带有非确定性的命令都会被认为是随机命令，比如 TIME 命令就是这方面的一个很好的例子。

每当从 Lua 脚本中调用那些返回无序元素的命令时，执行命令所得的数据在返回给 Lua 之前会先执行一个静默(slient)的字典序排序(lexicographical sorting)。举个例子，因为 Redis 的 Set 保存的是无序的元素，所以在 Redis 命令行客户端中直接执行 SMEMBERS ，返回的元素是无序的，但是，假如在脚本中执行 redis.call(“smembers”, KEYS[1]) ，那么返回的总是排过序的元素。

对 Lua 的伪随机数生成函数 math.random 和 math.randomseed 进行修改，使得每次在运行新脚本的时候，总是拥有同样的 seed 值。这意味着，每次运行脚本时，只要不使用 math.randomseed ，那么 math.random 产生的随机数序列总是相同的。

尽管有那么多的限制，但用户还是可以用一个简单的技巧写出带随机行为的脚本(如果他们需要的话)。

假设现在我们要编写一个 Redis 脚本，这个脚本从列表中弹出 N 个随机数。一个 Ruby 写的例子如下：

require 'rubygems'

require 'redis'

r = Redis.new

RandomPushScript = <<EOF

local i = tonumber(ARGV[1])

local res

while (i > 0) do

res = redis.call('lpush',KEYS[1],math.random())

i = i-1

end

return res

EOF

r.del(:mylist)

puts r.eval(RandomPushScript,[:mylist],[10,rand(2\*\*32)])

这个程序每次运行都会生成带有以下元素的列表：

> lrange mylist 0 -1

1) "0.74509509873814"

2) "0.87390407681181"

3) "0.36876626981831"

4) "0.6921941534114"

5) "0.7857992587545"

6) "0.57730350670279"

7) "0.87046522734243"

8) "0.09637165539729"

9) "0.74990198051087"

10) "0.17082803611217"

上面的 Ruby 程序每次都只生成同样的列表，用途并不是太大。那么，该怎样修改这个脚本，使得它仍然是一个纯函数(符合 Redis 的要求)，但是每次调用都可以产生不同的随机元素呢？

一个简单的办法是，为脚本添加一个额外的参数，让这个参数作为 Lua 的随机数生成器的 seed 值，这样的话，只要给脚本传入不同的 seed ，脚本就会生成不同的列表元素。

以下是修改后的脚本：

RandomPushScript = <<EOF

local i = tonumber(ARGV[1])

local res

math.randomseed(tonumber(ARGV[2]))

while (i > 0) do

res = redis.call('lpush',KEYS[1],math.random())

i = i-1

end

return res

EOF

r.del(:mylist)

puts r.eval(RandomPushScript,1,:mylist,10,rand(2\*\*32))

尽管对于同样的 seed ，上面的脚本产生的列表元素是一样的(因为它是一个纯函数)，但是只要每次在执行脚本的时候传入不同的 seed ，我们就可以得到带有不同随机元素的列表。

Seed 会在复制(replication link)和写 AOF 文件时作为一个参数来传播，保证在载入 AOF 文件或附属节点(slave)处理脚本时， seed 仍然可以及时得到更新。

注意，Redis 实现保证 math.random 和 math.randomseed 的输出和运行 Redis 的系统架构无关，无论是 32 位还是 64 位系统，无论是小端(little endian)还是大端(big endian)系统，这两个函数的输出总是相同的。

全局变量保护

为了防止不必要的数据泄漏进 Lua 环境， Redis 脚本不允许创建全局变量。如果一个脚本需要在多次执行之间维持某种状态，它应该使用 Redis key 来进行状态保存。

企图在脚本中访问一个全局变量(不论这个变量是否存在)将引起脚本停止， EVAL 命令会返回一个错误：

redis 127.0.0.1:6379> eval 'a=10' 0

(error) ERR Error running script (call to f\_933044db579a2f8fd45d8065f04a8d0249383e57): user\_script:1: Script attempted to create global variable 'a'

Lua 的 debug 工具，或者其他设施，比如打印（alter）用于实现全局保护的 meta table ，都可以用于实现全局变量保护。

实现全局变量保护并不难，不过有时候还是会不小心而为之。一旦用户在脚本中混入了 Lua 全局状态，那么 AOF 持久化和复制（replication）都会无法保证，所以，请不要使用全局变量。

避免引入全局变量的一个诀窍是：将脚本中用到的所有变量都使用 local 关键字定义为局部变量。

使用选择内部脚本

在正常的客户端连接里面可以调用SELECT选择内部的Lua脚本，但是Redis 2.8.11和Redis 2.8.12在行为上有一个微妙的变化。在2.8.12之前，会将脚本传送到调用脚本的当前数据库。从2.8.12开始Lua脚本只影响脚本本身的执行，但不修改当前客户端调用脚本时选定的数据库。

从补丁级发布的语义变化是必要的，因为旧的行为与Redis复制层固有的不相容是错误的原因。

可用库

Redis Lua解释器可用加载以下Lua库：

base lib.

table lib.

string lib.

math lib.

debug lib.

struct lib.

cjson lib.

cmsgpack lib.

bitop lib.

redis.sha1hex function.

每一个Redis实例都拥有以上的所有类库，以确保您使用脚本的环境都是一样的。

struct, CJSON 和 cmsgpack 都是外部库, 所有其他库都是标准。 Lua库。

struct 库

struct 是一个Lua装箱/拆箱的库

Valid formats:

> - big endian

< - little endian

![num] - alignment

x - pading

b/B - signed/unsigned byte

h/H - signed/unsigned short

l/L - signed/unsigned long

T - size\_t

i/In - signed/unsigned integer with size `n' (default is size of int)

cn - sequence of `n' chars (from/to a string); when packing, n==0 means

the whole string; when unpacking, n==0 means use the previous

read number as the string length

s - zero-terminated string

f - float

d - double

' ' - ignored

例子：

127.0.0.1:6379> eval 'return struct.pack("HH", 1, 2)' 0

"\x01\x00\x02\x00"

127.0.0.1:6379> eval 'return {struct.unpack("HH", ARGV[1])}' 0 "\x01\x00\x02\x00"

1) (integer) 1

2) (integer) 2

3) (integer) 5

127.0.0.1:6379> eval 'return struct.size("HH")' 0

(integer) 4

CJSON 库

CJSON 库为Lua提供极快的JSON处理

例子：

redis 127.0.0.1:6379> eval 'return cjson.encode({["foo"]= "bar"})' 0

"{\"foo\":\"bar\"}"

redis 127.0.0.1:6379> eval 'return cjson.decode(ARGV[1])["foo"]' 0 "{\"foo\":\"bar\"}"

"bar"

cmsgpack 库

cmsgpack 库为Lua提供了简单、快速的MessagePack操纵

例子：

127.0.0.1:6379> eval 'return cmsgpack.pack({"foo", "bar", "baz"})' 0

"\x93\xa3foo\xa3bar\xa3baz"

127.0.0.1:6379> eval 'return cmsgpack.unpack(ARGV[1])' 0 "\x93\xa3foo\xa3bar\xa3baz"

1) "foo"

2) "bar"

3) "baz"

bitop 库

bitop库为Lua的位运算模块增加了按位操作数。 它是Redis 2.8.18开始加入的。

例子：

127.0.0.1:6379> eval 'return bit.tobit(1)' 0

(integer) 1

127.0.0.1:6379> eval 'return bit.bor(1,2,4,8,16,32,64,128)' 0

(integer) 255

127.0.0.1:6379> eval 'return bit.tohex(422342)' 0

"000671c6"

它支持几个其他功能： bit.tobit, bit.tohex, bit.bnot, bit.band, bit.bor, bit.bxor, bit.lshift, bit.rshift, bit.arshift, bit.rol, bit.ror, bit.bswap. 所有可用的功能请参考Lua BitOp documentation

redis.sha1hex

对字符串执行SHA1算法

例子：

127.0.0.1:6379> eval 'return redis.sha1hex(ARGV[1])' 0 "foo"

"0beec7b5ea3f0fdbc95d0dd47f3c5bc275da8a33"

使用脚本散发 Redis 日志

在 Lua 脚本中，可以通过调用 redis.log 函数来写 Redis 日志(log)：

redis.log(loglevel,message)

其中， message 参数是一个字符串，而 loglevel 参数可以是以下任意一个值：

redis.LOG\_DEBUG

redis.LOG\_VERBOSE

redis.LOG\_NOTICE

redis.LOG\_WARNING

上面的这些等级(level)和标准 Redis 日志的等级相对应。

对于脚本散发(emit)的日志，只有那些和当前 Redis 实例所设置的日志等级相同或更高级的日志才会被散发。

以下是一个日志示例：

redis.log(redis.LOG\_WARNING, "Something is wrong with this script.")

执行上面的函数会产生这样的信息：

[32343] 22 Mar 15:21:39 # Something is wrong with this script.

沙箱(sandbox)和最大执行时间

脚本应该仅仅用于传递参数和对 Redis 数据进行处理，它不应该尝试去访问外部系统(比如文件系统)，或者执行任何系统调用。

除此之外，脚本还有一个最大执行时间限制，它的默认值是 5 秒钟，一般正常运作的脚本通常可以在几分之几毫秒之内完成，花不了那么多时间，这个限制主要是为了防止因编程错误而造成的无限循环而设置的。

最大执行时间的长短由 lua-time-limit 选项来控制(以毫秒为单位)，可以通过编辑 redis.conf 文件或者使用 CONFIG GET 和 CONFIG SET 命令来修改它。

当一个脚本达到最大执行时间的时候，它并不会自动被 Redis 结束，因为 Redis 必须保证脚本执行的原子性，而中途停止脚本的运行意味着可能会留下未处理完的数据在数据集(data set)里面。

因此，当脚本运行的时间超过最大执行时间后，以下动作会被执行：

Redis 记录一个脚本正在超时运行

Redis 开始重新接受其他客户端的命令请求，但是只有 SCRIPT KILL 和 SHUTDOWN NOSAVE 两个命令会被处理，对于其他命令请求， Redis 服务器只是简单地返回 BUSY 错误。

可以使用 SCRIPT KILL 命令将一个仅执行只读命令的脚本杀死，因为只读命令并不修改数据，因此杀死这个脚本并不破坏数据的完整性

如果脚本已经执行过写命令，那么唯一允许执行的操作就是 SHUTDOWN NOSAVE ，它通过停止服务器来阻止当前数据集写入磁盘

流水线(pipeline)上下文(context)中的 EVALSHA

在流水线请求的上下文中使用 EVALSHA 命令时，要特别小心，因为在流水线中，必须保证命令的执行顺序。

一旦在流水线中因为 EVALSHA 命令而发生 NOSCRIPT 错误，那么这个流水线就再也没有办法重新执行了，否则的话，命令的执行顺序就会被打乱。

为了防止出现以上所说的问题，客户端库实现应该实施以下的其中一项措施：

总是在流水线中使用 EVAL 命令

检查流水线中要用到的所有命令，找到其中的 EVAL 命令，并使用 SCRIPT EXISTS 命令检查要用到的脚本是不是全都已经保存在缓存里面了。如果所需的全部脚本都可以在缓存里找到，那么就可以放心地将所有 EVAL 命令改成 EVALSHA 命令，否则的话，就要在流水线的顶端(top)将缺少的脚本用 SCRIPT LOAD 命令加上去。

## Server

## Sets

## Sorted Sets

## Strings

## Transactions