关于本文档¶

在线阅读地址¶

本译文可以在 redis.readthedocs.org 在线阅读。

翻译事项¶

对比 Redis 官方文档，本译文有以下改进：

- 修改了官方文档一些拗口的地方，统一了文风（官方文档是多人协作），对于一些不好的例子进行了结构重排甚至重写（比如 sort 命令）
- 所有命令都带有代码示例，补齐了一些官方文档没有触及到的地方
- 和 Redis 最新稳定版 2.4.2 保持一致，修改和删除了官方文档一些滞后的地方

总的来说，译文的质量比起官方文档有不少进步，希望这个译文能对各位喜欢 Redis 的朋友们有所帮助。

报告翻译错误或加入本项目¶

提交翻译错误、意见、建议，或加入本项目，请到项目 Github 页面。
或直接联系译者。

联系译者¶

联系译者请到 twitter、豆瓣或发送邮件到 Gmail：huangz1990 (友情提示：使用”#”替换”@”对于防范垃圾邮件一点效果也没有)。

版权宣告¶

1. **Redis 是什么**

1. Redis 是 REmote DIctionary 璔erver 的缩写,是一个 key-value 存储系统.

2. Redis 提供了一些丰富的数据结构,包括 Strings,Lists,

Hashes,Sets 和 Ordered Sets 以及 Hashes.包括对这些数据结构的操作支持.

3. Redis 可以替代 Memcached,并且解决了断电后数据完全丢失的问题.

4. Redis 官方网站: http://redis.io

Redis 作者 Blog: http://antirez.com

2. **Redis 安装**

Download, extract and compile Redis with:

$ wget http://redis.googlecode.com/files/redis-2.4.5.tar.gz

$ tar xzf redis-2.4.5.tar.gz

$ cd redis-2.4.5

$ make

The binaries that are now compiled are available in the src directory. Run Redis with:

$ src/redis-server

You can interact with Redis using the built-in client:

$ src/redis-cli

redis> set foo bar

OK

redis> get foo

"bar"

3. **Redis 优点**

1.性能极高,Redis 能支持 10 万每秒的读写频率.

2.丰富的数据类型及对应的操作.

3.Redis 的所有操作都是原子性的,同时 Redis 还支持对几个操作全并后的原子性执行,也即支持事务.

4.丰富的特性,Redis 还支持 publish/subscribe, key 过期等特性.

4. **Redis 性能**

以下摘自官方测试描述:

在 50 个并发的情况下请求 10W 次,写的速度是 11W 次/s,读的速度是 8.1w 次/s.

测试环境:

1.50 个并发,请求 10W 次.

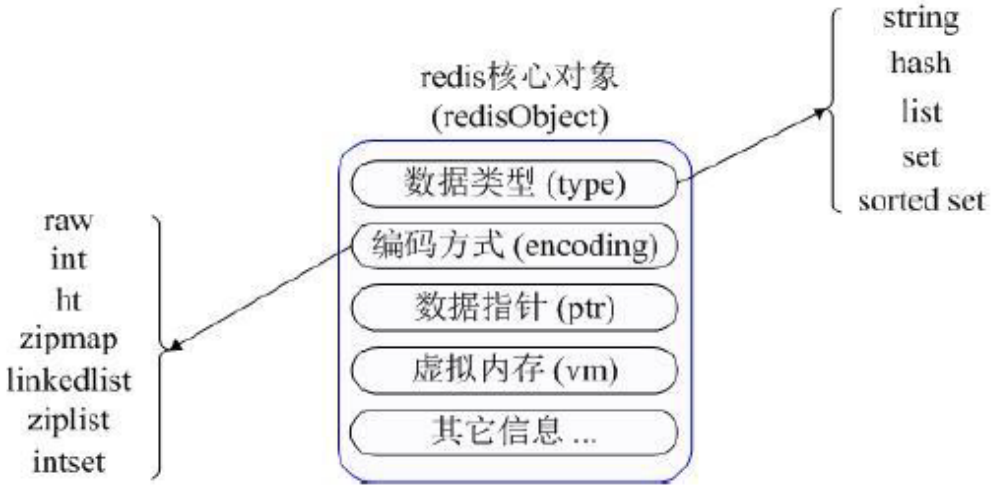
2.读和写大小为 256bytes 的字符串.

3.Linux2.6 Xeon X3320 2.5GHz 的服务器上.

4.通过本机的 loopback interface 接口上执行.

5.Redis 数据类型

redis 常用五种数据类型:string,hash,list,set,sorted set.



Redis 内部使用一个 redisObject 对象来表示所有的 key 和 value,redisObject 最主要的信息如上图所示:

type 代表一个 value 对象具体是何种数据类型,encoding 是不同数据类型在 redis 内部的存储方式,比如:

type=string 代表 value 存储的是一个普通字符串,那么对应的 encoding 可以是 raw 或者是 int,如果是 int

则代表实际 redis 内部是按数值型类存储和表示这个字符串的.

6.Redis 内存优化

Redis 为不同数据类型分别提供了一组参数来控制内存使用,我们在前面提到过 Redis Hash 的 value 内部是一个 HashMap,如果该 Map 的成员数比较少,则会采用一维数组的方式来紧凑存储该 Map,即省去了大量指针的内存开销,这个参数在 redis.conf 配置文件中下面 2 项:

hash-max-zipmap-entries 64

hash-max-zipmap-value 512

含义是当 value 这个 Map 内部不超过多少个成员时会采用线性紧凑格式存储,默认是 64,即 value 内部有 64 个以下的成员就是使用线性紧凑存储,超过该值自动转成真正的 HashMap. hash-max-zipmap-value 含义是当 value 这个 Map 内部的每个成员值长度不超过多少字节就会采用线性紧凑存储来节省空间.

7. 数据快照

数据快照的原理是将整个 Redis 内存中存的所有数据遍历一遍存到一个扩展名为 rdb 的数据文件中.通过 SAVE 命令可以调用这个过程.

数据快照配置

save 900 1

save 300 10

save 60 10000
以上在 redis.conf 中的配置指出在多长时间内,有多少次更新操作,就将数据同步到数据文件,这个可以多个条件配合.上面的含义是 900 秒后有一个 key 发生改变就执行 save,300 秒后有 10 个 key 发生改变执行 save,60 秒有 10000 个 key 发生改变执行 save
数据快照的缺点是持久化之后如果出现 crash 则会丢失一段数据 ,因此作者增加了另外一种追加式的操作日志记录,叫 append only file,其日志文件以 aof 结尾,我们一般称为 aof 文件.要开启 aof 日志的记录,需要在配置文件中进行如下设置:
appendonly yes
appendonly 配置如果不开启,可能会在断电时导致一段时间内的数据丢失.因为 redis 本身同步数据文件是按 save 条件来同步的,所以有的数据会在一段时间内只存在于内存中.
appendfsync no/always/everysec
1. no:表示等操作系统进行数据缓存同步到磁盘.
2. always:表示每次更新操作后手动调用 fsync() 将数据写到磁盘.
3. everysec:表示每秒同步一次.一般用 everysec.
AOF 文件只增不减会导致文件越来越大,重写过程如下
1. Redis 通过 fork 产生子进程.
2. 子进程将当前所有数据写入一个临时文件.
3. 父子进程是并行执行的,在子进程遍历并写临时文件的时候,父进程在照常接收请求,处理请求,写 AOF,不过这时他是把新来的 AOF 写在一个缓冲区中.
4. 子进程写完临时文件后就会退出.这时父进程会接收到子进程退出的消息,他会把自己现在收集在缓冲区中的所有 AOF 追加在临时文件中.
5. 最后把临时文件 rename 一下,改名为 appendonly.aof, 这时原来的 aof 文件被覆盖.整个过程完成.
当 Redis 服务器挂掉时,重启时将按以下优先级恢复数据到内存种:
1. 如果只配置了 AOF,重启时加载 AOF 文件恢复数据.
2.如果同时配置了 RBD 和 AOF,启动时只加载 AOF 文件恢复数据.
3.如果只配置了 RDB,启动时将加载 dump 文件恢复数据.
以上 2 个条件任意一个条件超过设置值都会转换成真正的 HashMap,也就不会再节省内存了,那么这个值是不是设置的越大越好呢,答案当然是否定的,HashMap 的优势就是查找和操作的时间复杂度都是 O(1)的,而放弃 Hash 采用一维存储则是 O(n)的时间复杂度,如果成员数量很少,则影响不大,否则会严重影响性能,所以要权衡好这个值的设置,总体上还是时间成本和空间成本上的权衡.

8. Redis 主从复制

Master/Slave 配置:

Master IP:175.41.209.118

Master Redis Server Port:6379

Slave 配置很简单,只需要在 slave 服务器的 redis.conf 加入:

slaveof 175.41.209.118 6379

启动 master 和 slave,然后写入数据到 master,读取 slave,可以看到数据被复制到 slave 了.

用途:读写分离,数据备份,灾难恢复等 Redis 主从复制过程:

配置好 slave 后,slave 与 master 建立连接,然后发送 sync 命令.

无论是第一次连接还是重新连接,master 都会启动一个后台进程,将数据库快照保存到文件中,同时 master 主进程会开始收集新的写命令并缓存. 后台进程完成写文件后,master 就发送文件给 slave,slave 将文件保存到硬盘上,再加载到内存中, 接着 master 就会把缓存的命令转发给 slave,后续 master 将收到的写命令发送给 slave. 如果 master 同时收到多个 slave 发来的同步连接命令,master 只会启动一个进程来写数据库镜像, 然后发送给所有的 slave.

Redis 主从复制特点:

1. master 可以拥有多个 slave.
2. 多个 slave 可以连接同一个 master 外,还可以连接到其他 slave.
3. 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.
4. 可以在 master 禁用数据持久化,注释掉 master 配置文件中的所有 save 配置,只需在 slave 上配置数据持久化.
5. 提高系统的伸缩性.

Redis 主从复制速度:

官方提供了一个数据, Slave 在 21 秒即完成了对 Amazon 网站 10G key set 的复制.

9. Redis 客户端

Redis 的客户端非常丰富,几乎所有流行的语言都有客户端.

客户端列表:http://redis.io/clients

Java 客户端推荐 Jedis: https://github.

com/xetorthio/jedis

Jedis 目前 Release 版本是 2.0.0,支持的特性如下,一句话概括,该有的都有了,不该有的也有了:

- Sorting
- Connection handling
- Commands operating on any kind of values
- Commands operating on string values
- Commands operating on hashes
- Commands operating on lists
- Commands operating on sets
- Commands operating on sorted sets

- Transactions
- Pipelining
- Publish/Subscribe
- Persistence control commands
- Remote server control commands
- Connection pooling
- Sharding (MD5, MurmureHash)
- Key-tags for sharding
- Sharding with pipelining

Jedis 使用

添加 Maven 依赖:

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.0.0</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

最简单的使用方式:

```
Jedis jedis = new Jedis("localhost");
jedis.set("foo", "bar");
String value = jedis.get("foo");
```

更多高级用法参考:https://github.com/xetorthio/jedis/wiki 目前,Redis server 没有提供 shard 功能,只能在 client 端实现.

Redis 有些客户端实现了 shard,比如 Java 客户端 Jedis.

Jedis 使用一致性哈希算法实现 shard,提供

JedisPool,JedisPoolConfig,JedisSharedInfo,

ShardedJedisPool 等相关类来使用 shared 功能.

Jedis shardedJedisPool 的创建例子:

```
<bean id="dataJedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxActive" value="200"/>
    <property name="maxIdle" value="200"/>
    <property name="maxWait" value="1000"/>
    <property name="testOnBorrow" value="true"/>
</bean>
<bean id="dataJedisShardInfo" class="redis.clients.jedis.JedisShardInfo">
    <constructor-arg index="0" value="${redis.host}"/>
```

```
        <constructor-arg index="1" value="${redis.port}"/>
        <constructor-arg index="2" value="10000"/>
    </bean>
    <bean id="dataShardedJedisPool"
class="redis.clients.jedis.ShardedJedisPool">
        <constructor-arg index="0" ref="dataJedisPoolConfig"/>
        <constructor-arg index="1">
            <list>
                <ref bean="dataJedisShardInfo"/>
            </list>
        </constructor-arg>
    </bean>
```

Jedis shardedJedisPool 的使用:

```
ShardedJedis shardedJedis    = shardedJedisPool.getResource();
//xxoo
shardedJedisPool.returnResource(shardedJedis);
```

当前稳定版本的 redis(2.4.5)只支持简单的 master-slave replication: 一个 master 写,多个 slave 读.只能通过客户端一致性哈希自己做 sharding.

Redis cluster 是下一阶段最重要的功能之一,会有集群的自动 sharding,多节点容错等.集群功能将在 3.0 版本推出.

Choice 的 Redis Server 是一主一从,使用亚马逊 AWS 虚拟机,机器配置如下:

7.5 GB memory

4 EC2 Compute Units (2 virtual cores with 2

EC2 Compute Units each)

64-bit platform

I/O Performance: High

现状:每天约更新 30 万左右的数据,现在库里有 400W 条纪录(400W 个 Key,使用的是 Hash 结构存储),每条数据都设有过期时间,占用了 2.5G 的内存.

为了提高读写性能 Master 关闭了 Persistence 功能,Slave 只负责同步备份 Master 的数据,不对外提供服务.

另一种可选方案是用 Master 提供写服务,Slave 提供读服务来实现读写分离.

Master 开启 save 功能的影响:在 dump 过程中,除了磁盘有大量的 IO 操作以外,Redis 是 fork 一个子进程来 dump 数据到硬盘,原有进程占用 30%+的 CPU,dump 数据的子进程单独另外占用一个 CPU.

Master 开启 save 对性能的直接影响:TPS 大概减少 30%.

Choice Master Redis 服务器开启 Save 功能后的明显影响是:

机器 Load 一直居高不下,大量请求超时. 关闭 Save 功能后服务器压力锐减,基本无请求超时情况发生.

Redis 开启 AOF 日志功能的影响:对性能有影响,但是由于每次追加的数据量小,所以对性能的影响相对小很多.

Choice Master Reids 开启 AOF 功能后,机器 load 微升,对性能无明显影响.

bgrewriteaof 对性能的影响:为了定时减小 AOF 文件的大小,Redis2.4 以后增加了自动的 bgrewriteaof 的功能,Redis 会选择一个自认为负载低的情况下执行 bgrewriteaof,这个重写 AOF 文件的过程是很影响性能的.

Choice Master 开启自动 bgrewriteaof 功能对系统的明显影响是:高并发时段有请求超时,机器 load 明显上升几倍.

目前较好的方案是:Mater 关闭 Save 功能,关闭 AOF 日志功能,以求达到性能最佳. Slave 开启 Save 并开启 AOF 日志功能,并开启 bgrewriteaof 功能,不对外提供服务,这样 Slave 的负载总体上会一直略高于 Master 负载,但 Master 性能达到最好.

总结：

从目前使用的情况来看,总体效果还是比较理想的, ChoiceHotels 的价格存储使用 Redis 的 Hash 结构也非常适合,HotelCode+Date 最为 key, 这样的 key 很容易设置过期,RatePlan+RoomType 作为 Filed, 价格和流量是 Value.

目前每天从 G璉A 到 Choice 大约 500w 个请求,整个过程 80%的请求在 500 毫秒内返回,基本无超时现象发生.

Redis Master Info

redis_version:2.4.4

connected_clients:171

connected_slaves:1

used_memory_human:2.37G

used_memory_peak_human:2.46G

aof_enabled:0

expired_keys:1595004

keyspace_hits:2611419705

keyspace_misses:55827727

role:master

aof_current_size:3874203906

aof_base_size:3850549480

db0:keys=4073286,expires=4073286

Commands 集合: http://redis.io/commands

Redis 五种数据类型介绍及操作命令：

键(Key)¶

DEL¶

DEL key [key ...]

移除给定的一个或多个 key 。

如果 key 不存在，则忽略该命令。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为要移除的 key 的数量。

移除单个字符串类型的 key ，时间复杂度为 O(1)。

移除单个列表、集合、有序集合或哈希表类型的 key ，时间复杂度为 O(M)， M 为以上数据结构内的元素数量。

返回值：

被移除 key 的数量。

```
#  删除单个 key

redis> SET name huangz
OK

redis> DEL name
(integer) 1


# 删除一个不存在的 key

redis> EXISTS phone
(integer) 0
```

```
redis> DEL phone # 失败，没有 key 被删除
(integer) 0


# 同时删除多个 key

redis> SET name "redis"
OK

redis> SET type "key-value store"
OK

redis> SET website "redis.com"
OK

redis> DEL name type website
(integer) 3
```

KEYS¶

KEYS pattern

查找所有符合给定模式 pattern 的 key 。

KEYS * 命中数据库中所有 key 。
KEYS h?llo 命中 hello ， hallo and hxllo 等。
KEYS h*llo 命中 hllo 和 heeeeello 等。
KEYS h[ae]llo 命中 hello 和 hallo ，但不命中 hillo 。

特殊符号用 "\" 隔开

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为数据库中 key 的数量。

返回值：

符合给定模式的 key 列表。

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 key ，你最好还是用 *集合(Set)* 来代替。

```
redis> MSET one 1 two 2 three 3 four 4  # 一次设置 4 个 key
OK

redis> KEYS *o*
1) "four"
2) "two"
3) "one"

redis> KEYS t??
1) "two"

redis> KEYS t[w]*
1) "two"

redis> KEYS *  # 匹配数据库内所有 key
1) "four"
2) "three"
3) "two"
4) "one"
```

RANDOMKEY¶

RANDOMKEY

从当前数据库中随机返回(不删除)一个 key 。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当数据库不为空时，返回一个 key 。

当数据库为空时，返回 nil 。

```
# 数据库不为空

redis> MSET fruit "apple" drink "beer" food "cookies"    # 设置多个 key
OK

redis> RANDOMKEY
"fruit"

redis> RANDOMKEY
"food"

redis> KEYS *      # 查看数据库内所有 key，证明 RANDOMKEY 并不删除 key
1) "food"
2) "drink"
3) "fruit"


# 数据库为空

redis> FLUSHDB   # 删除当前数据库所有 key
OK

redis> RANDOMKEY
(nil)
```

TTL¶

TTL key

以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 key 不存在或没有设置生存时间时，返回 -1 。

否则，返回 key 的剩余生存时间(以秒为单位)。

```
# 带 TTL 的 key

redis> SET name "redis"
OK

redis> EXPIRE name 30
(integer) 1

redis> TTL name
(integer) 27

redis> TTL name
(integer) 13

redis> TTL name    # 过期返回 -1
(integer) -1

redis> GET name    # 并且 key 被删除
(nil)

# 不带 TTL 的 key

redis> SET site wikipedia.org
OK

redis> TTL wikipedia.org
(integer) -1
```

不存在的 key

redis> EXISTS not_exists_key
(integer) 0

redis> TTL not_exists_key
(integer) -1

PTTL¶

PTTL key

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。

可用版本：

>= 2.6.0

复杂度：

O(1)

返回值：

如果 key 不存在，返回 -1 。

否则，返回以毫秒为单位表示的 key 的剩余生存时间。

redis> SET mykey "Hello"
OK

redis> EXPIRE mykey 1
(integer) 1

redis> PTTL mykey
(integer) 1000

对不存在的 key 返回 -1

redis> EXISTS some_key
(integer) 0

```
redis> PTTL some_key
(integer) -1
```

EXISTS¶

EXISTS key

检查给定 key 是否存在。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

若 key 存在，返回 1 ，否则返回 0 。

```
redis> SET db "redis"
OK

redis> EXISTS db
(integer) 1

redis> DEL db
(integer) 1

redis> EXISTS db
(integer) 0
```

MOVE¶

MOVE key db

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)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

移动成功返回 1 ，失败则返回 0 。

key 存在于当前数据库

redis> SELECT 0 # redis 默认使用数据库 0，为了清晰起见，这里再显式指定一次。
OK

redis> SET song "secret base - Zone"
OK

redis> MOVE song 1 # 将 song 移动到数据库 1
(integer) 1

redis> EXISTS song # song 已经被移走
(integer) 0

redis> SELECT 1 # 使用数据库 1
OK

redis:1> EXISTS song # 证实 song 被移到了数据库 1 (注意命令提示符变成了"redis:1"，表明正在使用数据库 1)
(integer) 1

当 key 不存在的时候

redis:1> EXISTS fake_key
(integer) 0

```
redis:1> MOVE fake_key 0  # 试图从数据库 1 移动一个不存在的 key 到数据库 0，失败
(integer) 0

redis:1> select 0  # 使用数据库 0
OK

redis> EXISTS fake_key  # 证实 fake_key 不存在
(integer) 0


# 当源数据库和目标数据库有相同的 key 时

redis> SELECT 0  # 使用数据库 0
OK
redis> SET favorite_fruit "banana"
OK

redis> SELECT 1  # 使用数据库 1
OK
redis:1> SET favorite_fruit "apple"
OK

redis:1> SELECT 0  # 使用数据库 0，并试图将 favorite_fruit 移动到数据库 1
OK

redis> MOVE favorite_fruit 1  # 因为两个数据库有相同的 key，MOVE 失败
(integer) 0

redis> GET favorite_fruit  # 数据库 0 的 favorite_fruit 没变
"banana"
```

```
redis> SELECT 1
OK

redis:1> GET favorite_fruit   # 数据库 1 的 favorite_fruit 也是
"apple"
```

RENAME¶

RENAME key newkey

将 key 改名为 newkey 。

当 key 和 newkey 相同或者 key 不存在时，返回一个错误。

当 newkey 已经存在时， RENAME 命令将覆盖旧值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

改名成功时提示 OK ，失败时候返回一个错误。

```
# key 存在且 newkey 不存在

redis> SET message "hello world"
OK

redis> RENAME message greeting
OK

redis> EXISTS message   # message 不复存在
(integer) 0

redis> EXISTS greeting   # greeting 取而代之
(integer) 1
```

当 key 不存在时，返回错误

redis> RENAME fake_key never_exists
(error) ERR no such key

newkey 已存在时， RENAME 会覆盖旧 newkey

redis> SET pc "lenovo"
OK

redis> SET personal_computer "dell"
OK

redis> RENAME pc personal_computer
OK

redis> GET pc
(nil)

redis:1> GET personal_computer # 原来的值 dell 被覆盖了
"lenovo"

RENAMENX¶

RENAMENX key newkey
当且仅当 newkey 不存在时，将 key 改为 newkey 。
出错的情况和 RENAME 一样(key 不存在时报错)。

可用版本：
>= 1.0.0
时间复杂度：
O(1)
返回值：
修改成功时，返回 1 。

如果 newkey 已经存在，返回 0 。

```
# newkey 不存在，改名成功

redis> SET player "MPlyaer"
OK

redis> EXISTS best_player
(integer) 0

redis> RENAMENX player best_player
(integer) 1


# newkey 存在时，失败

redis> SET animal "bear"
OK

redis> SET favorite_animal "butterfly"
OK

redis> RENAMENX animal favorite_animal
(integer) 0

redis> get animal
"bear"

redis> get favorite_animal
"butterfly"
```

TYPE¶

TYPE key

返回 key 所储存的值的类型。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

none (key 不存在)

string (字符串)

list (列表)

set (集合)

zset (有序集)

hash (哈希表)

```
redis> SET weather "sunny"  # 构建一个字符串
OK

redis> TYPE weather
string

redis> LPUSH book_list "programming in scala"  # 构建一个列表
(integer) 1

redis> TYPE book_list
list

redis> SADD pat "dog"  # 构建一个集合
(integer) 1

redis> TYPE pat
set
```

EXPIRE¶

EXPIRE key seconds

为给定 key 设置生存时间。当 key 过期时，它会被自动删除。

在 Redis 中，带有生存时间的 key 被称作『可挥发』(volatile)的。

生存时间可以通过使用 *DEL* 命令来删除整个 key 来移除，或者被 *SET* 和 *GETSET* 命令覆写(overwrite)，这意味着，如果一个命令只是修改(alter)一个带生存时间的 key 的值而不是用一个新的 key 值来代替(replace)它的话，那么生存时间不会被改变。

比如说，对一个 key 执行 *INCR* 命令，对一个列表进行 *LPUSH* 命令，或者对一个哈希表执行 *HSET* 命令，这类操作都不会修改 key 本身的生存时间。

使用 *PERSIST* 命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个持久化(persistent) key 。

另一方面，如果使用 *RENAME* 对一个 key 进行改名，那么改名后的 key 的生存时间和改名前一样。

RENAME 命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another_key ，这时旧的 another_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another_key ，因此，新的 another_key 的生存时间也和原本的 key 一样。

更新生存时间

可以对一个已经带有生存时间的 key 执行 *EXPIRE* 命令，新指定的生存时间会取代旧的生存时间。

过期时间的精确度

在 Redis 2.4 版本中，过期时间的延迟在 1 秒钟之内 —— 也即是，就算 key 已经过期，但它还是可能在过期之后一秒钟之内被访问到，而在新的 Redis 2.6 版本中，延迟被降低到 1 毫秒之内。

Redis 2.1.3 之前的不同之处

在 Redis 2.1.3 之前的版本中，修改一个带有生存时间的 key 会导致整个 key 被删除，这一行为是受当时复制(replication)层的限制而作出的，现在这一限制已经被修复。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功返回 1。

当 key 不存在或者不能为 key 设置生存时间时(比如在低于 2.1.3 中你尝试更新 key 的生存时间)，返回 0。

```
redis> SET cache_page "www.google.com"
```

OK

redis> EXPIRE cache_page 30 # 设置过期时间为 30 秒
(integer) 1

redis> TTL cache_page # 查看剩余生存时间
(integer) 23

redis> EXPIRE cache_page 30000 # 更新过期时间
(integer) 1

redis> TTL cache_page
(integer) 29996

模式：导航会话¶

假设你有一项 web 服务，打算根据用户最近访问的 N 个页面来进行物品推荐，并且假设用户停止阅览超过 60 秒，那么就清空阅览记录(为了减少物品推荐的计算量，并且保持推荐物品的新鲜度)。
这些最近访问的页面记录，我们称之为『导航会话』(Navigation session)，可以用 *INCR* 和 *RPUSH* 命令在 Redis 中实现它：每当用户阅览一个网页的时候，执行以下代码：

```
MULTI
    RPUSH pagewviews.user:<userid> http://.....
    EXPIRE pagewviews.user:<userid> 60
EXEC
```

如果用户停止阅览超过 60 秒，那么它的导航会话就会被清空，当用户重新开始阅览的时候，系统又会重新记录导航会话，继续进行物品推荐。

PEXPIRE¶

PEXPIRE key milliseconds

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。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功，返回 1

key 不存在或设置失败，返回 0

```
redis> SET mykey "Hello"
OK

redis> PEXPIRE mykey 1500
(integer) 1

redis> TTL mykey      # TTL 的返回值以秒为单位
(integer) 2

redis> PTTL mykey     # PTTL 可以给出准确的毫秒数
(integer) 1499
```

EXPIREAT¶

EXPIREAT key timestamp

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)。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

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，返回 0 。

```
redis> SET cache www.google.com
```

OK

redis> EXPIREAT cache 1355292000 # 这个 key 将在 2012.12.12 过期
(integer) 1

redis> TTL cache
(integer) 45081860

OBJECT¶

OBJECT subcommand [arguments [arguments]]
OBJECT 命令允许从内部察看给定 key 的 Redis 对象。

它通常用在除错(debugging)或者了解为了节省空间而对 key 使用特殊编码的情况。
当将 Redis 用作缓存程序时，你也可以通过 OBJECT 命令中的信息，决定 key 的驱逐策略(eviction policies)。

OBJECT 命令有多个子命令：

- OBJECT REFCOUNT <key> 返回给定 key 引用所储存的值的次数。此命令主要用于除错。
- OBJECT ENCODING <key> 返回给定 key 锁储存的值所使用的内部表示(representation)。
- OBJECT IDLETIME <key> 返回给定 key 自储存以来的空转时间(idle，没有被读取也没有被写入)，以秒为单位。

对象可以以多种方式编码：

- 字符串可以被编码为 raw (一般字符串)或 int (用字符串表示 64 位数字是为了节约空间)。
- 列表可以被编码为 ziplist 或 linkedlist 。 ziplist 是为节约大小较小的列表空间而作的特殊表示。
- 集合可以被编码为 intset 或者 hashtable 。 intset 是只储存数字的小集合的特殊表示。
- 哈希表可以编码为 zipmap 或者 hashtable 。 zipmap 是小哈希表的特殊表示。
- 有序集合可以被编码为 ziplist 或者 skiplist 格式。 ziplist 用于表示小的有序集合，而 skiplist 则用于表示任何大小的有序集合。

假如你做了什么让 Redis 没办法再使用节省空间的编码时(比如将一个只有 1 个元素的集合扩展为一个有 100 万个元素的集合)，特殊编码类型(specially encoded types)会自动

转换成通用类型(general type)。

可用版本：

>= 2.2.3

时间复杂度：

O(1)

返回值：

REFCOUNT 和 IDLETIME 返回数字。

ENCODING 返回相应的编码类型。

```
redis> SET game "COD"  # 设置一个字符串
OK

redis> OBJECT REFCOUNT game  # 只有一个引用
(integer) 1

redis> OBJECT IDLETIME game  # 等待一阵。。。然后查看空转时间
(integer) 90

redis> GET game  # 提取 game， 让它处于活跃(active)状态
"COD"

redis> OBJECT IDLETIME game  # 不再处于空转
(integer) 0

redis> OBJECT ENCODING game  # 字符串的编码方式
"raw"

redis> SET phone 15820123123  # 大的数字也被编码为字符串
OK

redis> OBJECT ENCODING phone
"raw"

redis> SET age 20  # 短数字被编码为 int
OK
```

redis> OBJECT ENCODING age
"int"

PEXPIREAT¶

key milliseconds timestamp

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。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

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)

redis> SET mykey "Hello"
OK

redis> PEXPIREAT mykey 1555555555005
(integer) 1

redis> TTL mykey # TTL 返回秒
(integer) 223157079

redis> PTTL mykey # PTTL 返回毫秒
(integer) 223157079318

PERSIST¶

PERSIST key

移除给定 key 的生存时间，将这个 key 从『可挥发』的(带生存时间 key)转换成『持久化』的(一个不带生存时间、永不过期的 key)。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

当生存时间移除成功时，返回 1 .

如果 key 不存在或 key 没有设置生存时间，返回 0 。

```
redis> SET mykey "Hello"
OK

redis> EXPIRE mykey 10   # 为 key 设置生存时间
(integer) 1

redis> TTL mykey
(integer) 10

redis> PERSIST mykey      # 移除 key 的生存时间
(integer) 1

redis> TTL mykey
(integer) -1
```

SORT¶

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

返回或保存给定列表、集合、有序集合 key 中经过排序的元素。

排序默认以数字作为对象，值被解释为双精度浮点数，然后进行比较。

一般 SORT 用法

最简单的 SORT 使用方法是 SORT key 。

假设 today_cost 是一个保存数字的列表， SORT 命令默认会返回该列表值的递增(从小到大)排序结果。

```
# 将数据一一加入到列表中
```

```
redis> LPUSH today_cost 30
(integer) 1

redis> LPUSH today_cost 1.5
(integer) 2

redis> LPUSH today_cost 10
(integer) 3

redis> LPUSH today_cost 8
(integer) 4

# 排序

redis> SORT today_cost
1) "1.5"
2) "8"
3) "10"
4) "30"
```

当数据集中保存的是字符串值时，你可以用 ALPHA 修饰符(modifier)进行排序。

```
# 将数据一一加入到列表中

redis> LPUSH website "www.reddit.com"
(integer) 1
redis> LPUSH website "www.slashdot.com"
(integer) 2
redis> LPUSH website "www.infoq.com"
(integer) 3

# 默认排序

redis> SORT website
```

1) "www.infoq.com"
2) "www.slashdot.com"
3) "www.reddit.com"

按字符排序

redis> SORT website ALPHA
1) "www.infoq.com"
2) "www.reddit.com"
3) "www.slashdot.com"

如果你正确设置了 !LC_COLLATE 环境变量的话，Redis 能识别 UTF-8 编码。

排序之后返回的元素数量可以通过 LIMIT 修饰符进行限制。
LIMIT 修饰符接受两个参数： offset 和 count 。
offset 指定要跳过的元素数量， count 指定跳过 offset 个指定的元素之后，要返回多少个对象。

以下例子返回排序结果的前 5 个对象(offset 为 0 表示没有元素被跳过)。

将数据一一加入到列表中

redis> LPUSH rank 30
(integer) 1
redis> LPUSH rank 56
(integer) 2
redis> LPUSH rank 42
(integer) 3
redis> LPUSH rank 22
(integer) 4
redis> LPUSH rank 0
(integer) 5
redis> LPUSH rank 11
(integer) 6
redis> LPUSH rank 32

(integer) 7
redis> LPUSH rank 67
(integer) 8
redis> LPUSH rank 50
(integer) 9
redis> LPUSH rank 44
(integer) 10
redis> LPUSH rank 55
(integer) 11

排序

redis> SORT rank LIMIT 0 5 # 返回排名前五的元素
1) "0"
2) "11"
3) "22"
4) "30"
5) "32"

修饰符可以组合使用。以下例子返回降序(从大到小)的前 5 个对象。

redis> SORT rank LIMIT 0 5 DESC
1) "78"
2) "67"
3) "56"
4) "55"
5) "50"

使用外部 key 进行排序

有时候你会希望使用外部的 key 作为权重来比较元素，代替默认的对比方法。
假设现在有用户(user)数据如下：

id	name	level
1	admin	9999

id	name	level
2	huangz	10
59230	jack	3
222	hacker	9999

id 数据保存在 key 名为 user_id 的列表中。
name 数据保存在 key 名为 user_name_{id} 的列表中
level 数据保存在 user_level_{id} 的 key 中。
先将要使用的数据加入到数据库中

admin

```
redis> LPUSH user_id 1
(integer) 1
redis> SET user_name_1 admin
OK
redis> SET user_level_1 9999
OK
```

huangz

```
redis> LPUSH user_id 2
(integer) 2
redis> SET user_name_2 huangz
OK
redis> SET user_level_2 10
OK
```

jack

```
redis> LPUSH user_id 59230
(integer) 3
redis> SET user_name_59230 jack
OK
```

```
redis> SET user_level_59230 3
OK

# hacker

redis> LPUSH user_id 222
(integer) 4
redis> SET user_name_222 hacker
OK
redis> SET user_level_222 9999
OK
```

如果希望按 level 从大到小排序 user_id ，可以使用以下命令：

```
redis> SORT user_id BY user_level_* DESC
1) "222"       # hacker
2) "1"         # admin
3) "2"         # huangz
4) "59230"    # jack
```

但是有时候只是返回相应的 id 没有什么用，你可能更希望排序后返回 id 对应的用户名，这样更友好一点，使用 GET 选项可以做到这一点：

```
redis> SORT user_id BY user_level_* DESC GET user_name_*
1) "hacker"
2) "admin"
3) "huangz"
4) "jack"
```

可以多次地、有序地使用 GET 操作来获取更多外部 key 。
比如你不但希望获取用户名，还希望连用户的密码也一并列出，可以使用以下命令：

```
# 先添加一些测试数据
```

```
redis> SET user_password_222 "hey,im in"
OK
redis> SET user_password_1 "a_long_long_password"
OK
redis> SET user_password_2 "nobodyknows"
OK
redis> SET user_password_59230 "jack201022"
OK

# 获取 name 和 password

redis> SORT user_id BY user_level_* DESC GET user_name_* GET
user_password_*
1) "hacker"           # 用户名
2) "hey,im in"       # 密码
3) "jack"
4) "jack201022"
5) "huangz"
6) "nobodyknows"
7) "admin"
8) "a_long_long_password"

# 注意 GET 操作是有序的，GET user_name_* GET user_password_* 和 GET
user_password_* GET user_name_*返回的结果位置不同

redis> SORT user_id BY user_level_* DESC GET user_password_* GET
user_name_*
1) "hey,im in"       # 密码
2) "hacker"           # 用户名
3) "jack201022"
4) "jack"
5) "nobodyknows"
6) "huangz"
```

7) "a_long_long_password"
8) "admin"

GET 还有一个特殊的规则——"GET #"，用于获取被排序对象(我们这里的例子是 user_id)的当前元素。
比如你希望 user_id 按 level 排序，还要列出 id 、 name 和 password ，可以使用以下命令：

redis> SORT user_id BY user_level_* DESC GET # GET user_name_* GET user_password_*
1) "222" # id
2) "hacker" # name
3) "hey,im in" # password
4) "1"
5) "admin"
6) "a_long_long_password"
7) "2"
8) "huangz"
9) "nobodyknows"
10) "59230"
11) "jack"
12) "jack201022"

只获取对象而不排序
BY 修饰符可以将一个不存在的 key 当作权重，让 SORT 跳过排序操作。
该方法用于你希望获取外部对象而又不希望引起排序开销时使用。

确保 fake_key 不存在

redis> EXISTS fake_key
(integer) 0

以 fake_key 作 BY 参数，不排序，只 GET name 和 GET password

```
redis> SORT user_id BY fake_key GET # GET user_name_* GET
user_password_*
1) "222"              # id
2) "hacker"         # user_name
3) "hey,im in"    # password
4) "59230"
5) "jack"
6) "jack201022"
7) "2"
8) "huangz"
9) "nobodyknows"
10) "1"
11) "admin"
12) "a_long_long_password"
```

保存排序结果

默认情况下， SORT 操作只是简单地返回排序结果，如果你希望保存排序结果，可以给 STORE 选项指定一个 key 作为参数，排序结果将以列表的形式被保存到这个 key 上。(若指定 key 已存在，则覆盖。)

```
redis> EXISTS user_info_sorted_by_level   # 确保指定 key 不存在
(integer) 0

redis> SORT user_id BY user_level_* GET # GET user_name_* GET
user_password_* STORE user_info_sorted_by_level       # 排序
(integer) 12   # 显示有 12 条结果被保存了

redis> LRANGE user_info_sorted_by_level 0 11   # 查看排序结果
1) "59230"
2) "jack"
3) "jack201022"
4) "2"
5) "huangz"
6) "nobodyknows"
```

7) "222"
8) "hacker"
9) "hey,im in"
10) "1"
11) "admin"
12) "a_long_long_password"

一个有趣的用法是将 SORT 结果保存，用 EXPIRE 为结果集设置生存时间，这样结果集就成了 SORT 操作的一个缓存。
这样就不必频繁地调用 SORT 操作了，只有当结果集过期时，才需要再调用一次 SORT 操作。
有时候为了正确实现这一用法，你可能需要加锁以避免多个客户端同时进行缓存重建(也就是多个客户端，同一时间进行 SORT 操作，并保存为结果集)，具体参见 *SETNX* 命令。

在 GET 和 BY 中使用哈希表

可以使用哈希表特有的语法，在 SORT 命令中进行 GET 和 BY 操作。

```
# 假设现在我们的用户表新增了一个 serial 项来为作为每个用户的序列号
# 序列号以哈希表的形式保存在 serial 哈希域内。

redis> HMSET serial 1 23131283 2 23810573 222 502342349 59230 2435829758
OK

# 我们希望以比较 serial 中的大小来作为排序 user_id 的方式

redis> SORT user_id BY *->serial
1) "222"
2) "59230"
3) "2"
4) "1"
```

符号 "->" 用于分割哈希表的关键字(key name)和索引域(hash field)，格式为 "key->field" 。

除此之外 ,哈希表的 BY 和 GET 操作和上面介绍的其他数据结构(列表、集合、有序集合)没有什么不同。

可用版本：

>= 1.0.0

时间复杂度：

O(N+M*log(M))， N 为要排序的列表或集合内的元素数量， M 为要返回的元素数量。

如果只是使用 SORT 命令的 GET 选项获取数据而没有进行排序，时间复杂度 O(N)。

返回值：

没有使用 STORE 参数，返回列表形式的排序结果。

使用 STORE 参数，返回排序结果的元素数量。

字符串(String)¶

常用命令:
set,get,decr,incr,mget 等.

应用场景:
String 是最常用的一种数据类型,普通的 key/value 存储.

实现方式:
String 在 redis 内部存储默认就是一个字符串,被 redisObject 所引用,当遇到 incr,
decr 等操作时会转成数值型进行计算,此时 redisObject 的 encoding 字段为 int.

SET¶

SET key value
将字符串值 value 关联到 key 。
如果 key 已经持有其他值， SET 就覆写旧值，无视类型。

可用版本：
>= 1.0.0
时间复杂度：
O(1)
返回值：
总是返回 OK ，因为 SET 不可能失败。

```
# 对字符串类型的 key 进行 SET

redis> SET apple www.apple.com
OK

redis> GET apple
"www.apple.com"


# 对非字符串类型的 key 进行 SET
```

```
redis> LPUSH greet_list "hello"                    # 建立一个列表
(integer) 1

redis> TYPE greet_list
list

redis> SET greet_list "yooooooooooooooooo"   # 覆盖列表类型
OK

redis> TYPE greet_list
string
```

SETNX¶

SETNX key value

将 key 的值设为 value ，当且仅当 key 不存在。

若给定的 key 已经存在，则 SETNX 不做任何动作。

SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 1 。

设置失败，返回 0 。

```
redis> EXISTS job                        # job 不存在
(integer) 0

redis> SETNX job "programmer"       # job 设置成功
(integer) 1

redis> SETNX job "code-farmer"     # 尝试覆盖 job ，失败
(integer) 0
```

redis> GET job # 没有被覆盖
"programmer"

模式：将 SETNX 用于加锁(locking)¶

警告

已经证实这个加锁算法带有竞争条件，在特定情况下会造成错误，请不要使用这个加锁算法，具体请参考 http://huangz.iteye.com/blog/1381538 。

SETNX 可以用作加锁原语(locking primitive)。比如说，要对关键字(key) foo 加锁，客户端可以尝试以下方式：
SETNX lock.foo <current Unix time + lock timeout + 1>
如果 SETNX 返回 1 ，说明客户端已经获得了锁， key 设置的 unix 时间则指定了锁失效的时间。之后客户端可以通过 DEL lock.foo 来释放锁。
如果 SETNX 返回 0 ，说明 key 已经被其他客户端上锁了。如果锁是非阻塞(non blocking lock)的，我们可以选择返回调用，或者进入一个重试循环，直到成功获得锁或重试超时(timeout)。
处理死锁(deadlock)
上面的锁算法有一个问题：如果因为客户端失败、崩溃或其他原因导致没有办法释放锁的话，怎么办？
这种状况可以通过检测发现——因为上锁的 key 保存的是 unix 时间戳，假如 key 值的时间戳小于当前的时间戳，表示锁已经不再有效。
但是，当有多个客户端同时检测一个锁是否过期并尝试释放它的时候，我们不能简单粗暴地删除死锁的 key ，再用 SETNX 上锁，因为这时竞争条件(race condition)已经形成了：

- C1 和 C2 读取 lock.foo 并检查时间戳， SETNX 都返回 0 ，因为它已经被 C3 锁上了，但 C3 在上锁之后就崩溃(crashed)了。
- C1 向 lock.foo 发送 *DEL* 命令。
- C1 向 lock.foo 发送 SETNX 并成功。
- C2 向 lock.foo 发送 *DEL* 命令。
- C2 向 lock.foo 发送 SETNX 并成功。
- 出错：因为竞争条件的关系，C1 和 C2 两个都获得了锁。

幸好，以下算法可以避免以上问题。来看看我们聪明的 C4 客户端怎么办：

- C4 向 lock.foo 发送 SETNX 命令。
- 因为崩溃掉的 C3 还锁着 lock.foo ，所以 Redis 向 C4 返回 0 。
- C4 向 lock.foo 发送 GET 命令，查看 lock.foo 的锁是否过期。如果不，则休眠(sleep)一段时间，并在之后重试。
- 另一方面，如果 lock.foo 内的 unix 时间戳比当前时间戳老，C4 执行以下命令：

GETSET lock.foo <current Unix timestamp + lock timeout + 1>

- 因为 GETSET 的作用，C4 可以检查看 GETSET 的返回值，确定 lock.foo 之前储存的旧值仍是那个过期时间戳，如果是的话，那么 C4 获得锁。
- 如果其他客户端 ,比如 C5 ,比 C4 更快地执行了 GETSET 操作并获得锁 ,那么 C4 的 GETSET 操作返回的就是一个未过期的时间戳(C5 设置的时间戳)。C4 只好从第一步开始重试。

注意，即便 C4 的 GETSET 操作对 key 进行了修改，这对未来也没什么影响。

警告

为了让这个加锁算法更健壮 ,获得锁的客户端应该常常检查过期时间以免锁因诸如 *DEL* 等命令的执行而被意外解开，因为客户端失败的情况非常复杂，不仅仅是崩溃这么简单，还可能是客户端因为某些操作被阻塞了相当长时间，紧接着 *DEL* 命令被尝试执行(但这时锁却在另外的客户端手上)。

SETEX¶

SETEX key seconds value

将值 value 关联到 key ，并将 key 的生存时间设为 seconds (以秒为单位)。
如果 key 已经存在， SETEX 命令将覆写旧值。
这个命令类似于以下两个命令：

```
SET key value
EXPIRE key seconds  # 设置生存时间
```

不同之处是， SETEX 是一个原子性(atomic)操作，关联值和设置生存时间两个动作会在同一时间内完成，该命令在 Redis 用作缓存时，非常实用。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功时返回 OK 。

当 seconds 参数不合法时，返回一个错误。

```
# 在 key 不存在时进行 SETEX

redis> SETEX cache_user_id 60 10086
OK

redis> GET cache_user_id   # 值
"10086"

redis> TTL cache_user_id   # 剩余生存时间
(integer) 49



# key 已经存在时，SETEX 覆盖旧值

redis> SET cd "timeless"
OK

redis> SETEX cd 3000 "goodbye my love"
OK

redis> GET cd
"goodbye my love"

redis> TTL cd
(integer) 2997
```

PSETEX¶

PSETEX key milliseconds value

这个命令和 *SETEX* 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 *SETEX* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功时返回 OK 。

```
redis> PSETEX mykey 1000 "Hello"
OK

redis> PTTL mykey
(integer) 999

redis> GET mykey
"Hello"
```

SETRANGE¶

SETRANGE key offset value

用 value 参数覆写(Overwrite)给定 key 所储存的字符串值，从偏移量 offset 开始。

不存在的 key 当作空白字符串处理。

SETRANGE 命令会确保字符串足够长以便将 value 设置在指定的偏移量上，如果给定 key 原来储存的字符串长度比偏移量小(比如字符串只有 5 个字符长，但你设置的 offset 是 10)，那么原字符和偏移量之间的空白将用零比特(zerobytes, "\x00")来填充。

注意你能使用的最大偏移量是 2^29-1(536870911)，因为 Redis 字符串的大小被限制在 512 兆(megabytes)以内。如果你需要使用比这更大的空间，你可以使用多个 key 。

警告

当生成一个很长的字符串时，Redis 需要分配内存空间，该操作有时候可能会造成服务器阻塞(block)。在 2010 年的 Macbook Pro 上，设置偏移量为 536870911(512MB 内存分配)，耗费约 300 毫秒， 设置偏移量为 134217728(128MB 内存分配)，耗费约 80 毫秒，设置偏移量 33554432(32MB 内存分配)，耗费约 30 毫秒，设置偏移量为 8388608(8MB 内存分配) ,耗费约 8 毫秒。注意若首次内存分配成功之后 ,再对同一个 key 调用 SETRANGE 操作，无须再重新内存。

可用版本：

>= 2.2.0

时间复杂度：

对小(small)的字符串，平摊复杂度 O(1)。(关于什么字符串是”小”的，请参考 *APPEND* 命令)

否则为 O(M)， M 为 value 参数的长度。

返回值：

被 SETRANGE 修改之后，字符串的长度。

```
# 对非空字符串进行 SETRANGE

redis> SET greeting "hello world"
OK

redis> SETRANGE greeting 6 "Redis"
(integer) 11

redis> GET greeting
"hello Redis"


# 对空字符串/不存在的 key 进行 SETRANGE

redis> EXISTS empty_string
(integer) 0

redis> SETRANGE empty_string 5 "Redis!"    # 对不存在的 key 使用 SETRANGE
```

(integer) 11

redis> GET empty_string # 空白处被"\x00"填充
"\x00\x00\x00\x00\x00Redis!"

模式¶

因为有了 SETRANGE 和 *GETRANGE* 命令 ,你可以将 Redis 字符串用作具有 O(1)随机访问时间的线性数组 ,这在很多真实用例中都是非常快速且高效的储存方式，具体请参考 *APPEND* 命令的『模式：时间序列』部分。

MSET¶

MSET key value [key value ...]
同时设置一个或多个 key-value 对。
如果某个给定 key 已经存在，那么 MSET 会用新值覆盖原来的旧值，如果这不是你所希望的效果，请考虑使用 MSETNX 命令：它只会在所有给定 key 都不存在的情况下进行设置操作。
MSET 是一个原子性(atomic)操作，所有给定 key 都会在同一时间内被设置，某些给定 key 被更新而另一些给定 key 没有改变的情况，不可能发生。

可用版本：
>= 1.0.1
时间复杂度：
O(N)， N 为要设置的 key 数量。
返回值：
总是返回 OK (因为 MSET 不可能失败)

redis> MSET date "2012.3.30" time "11:00 a.m." weather "sunny"
OK

redis> MGET date time weather
1) "2012.3.30"
2) "11:00 a.m."
3) "sunny"

MSET 覆盖旧值例子

redis> SET google "google.hk"
OK

redis> MSET google "google.com"
OK

redis> GET google
"google.com"

MSETNX¶

MSETNX key value [key value ...]
同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。
即使只有一个给定 key 已存在， MSETNX 也会拒绝执行所有给定 key 的设置操作。
MSETNX 是原子性的 ,因此它可以用作设置多个不同 key 表示不同字段(field)的唯一性逻辑对象(unique logic object)，所有字段要么全被设置，要么全不被设置。

可用版本：
>= 1.0.1
时间复杂度：
O(N)， N 为要设置的 key 的数量。
返回值：
当所有 key 都成功设置，返回 1 。
如果所有给定 key 都设置失败(最少有一个 key 已经存在)，那么返回 0 。
对不存在的 key 进行 MSETNX

redis> MSETNX rmdbs "MySQL" nosql "MongoDB" key-value-store "redis"
(integer) 1

redis> MGET rmdbs nosql key-value-store

1) "MySQL"
2) "MongoDB"
3) "redis"

MSET 的给定 key 当中有已存在的 key

redis> MSETNX rmdbs "Sqlite" language "python" # rmdbs 键已经存在，操作失败
(integer) 0

redis> EXISTS language # 因为 MSET 是原子性操作，language 没有被设置
(integer) 0

redis> GET rmdbs # rmdbs 也没有被修改
"MySQL"

APPEND¶

APPEND key value

如果 key 已经存在并且是一个字符串， APPEND 命令将 value 追加到 key 原来的值之后。

如果 key 不存在， APPEND 就简单地将给定 key 设为 value ，就像执行 SET key value 一样。

可用版本：

>= 2.0.0

时间复杂度：

平摊 O(1)

返回值：

追加 value 之后， key 中字符串的长度。

对不存在的 key 执行 APPEND

redis> EXISTS myphone # 确保 myphone 不存在
(integer) 0

```
redis> APPEND myphone "nokia"    # 对不存在的 key 进行 APPEND ，等同于 SET myphone "nokia"
(integer) 5                      # 字符长度


# 对已存在的字符串进行 APPEND

redis> APPEND myphone " - 1110"   # 长度从 5 个字符增加到 12 个字符
(integer) 12

redis> GET myphone
"nokia - 1110"
```

模式：时间序列(Time series)¶

APPEND 可以为一系列定长(fixed-size)数据(sample)提供一种紧凑的表示方式，通常称之为时间序列。
每当一个新数据到达的时候，执行以下命令：

```
APPEND timeseries "fixed-size sample"
```

然后可以通过以下的方式访问时间序列的各项属性：

- *STRLEN* 给出时间序列中数据的数量
- *GETRANGE* 可以用于随机访问。只要有相关的时间信息的话，我们就可以在 Redis 2.6 中使用 Lua 脚本和 *GETRANGE* 命令实现二分查找。
- *SETRANGE* 可以用于覆盖或修改已存在的的时间序列。

这个模式的唯一缺陷是我们只能增长时间序列，而不能对时间序列进行缩短，因为 Redis 目前还没有对字符串进行修剪(tirm)的命令，但是，不管怎么说，这个模式的储存方式还是可以节省下大量的空间。

注解

可以考虑使用 UNIX 时间戳作为时间序列的键名，这样一来，可以避免单个 key 因为保存过大的时间序列而占用大量内存，另一方面，也可以节省下大量命名空间。

下面是一个时间序列的例子：

redis> APPEND ts "0043"
(integer) 4

redis> APPEND ts "0035"
(integer) 8

redis> GETRANGE ts 0 3
"0043"

redis> GETRANGE ts 4 7
"0035"

GET¶

GET key

返回 key 所关联的字符串值。

如果 key 不存在则返回特殊值 nil 。

假如 key 储存的值不是字符串类型，返回一个错误，因为 GET 只能用于处理字符串值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 key 不存在时，返回 nil ，否则，返回 key 的值。

如果 key 不是字符串类型，那么返回一个错误。

对不存在的 key 或字符串类型 key 进行 GET

```
redis> GET db
(nil)

redis> SET db redis
OK

redis> GET db
"redis"

# 对不是字符串类型的 key 进行 GET

redis> DEL db
(integer) 1

redis> LPUSH db redis mongodb mysql
(integer) 3

redis> GET db
(error) ERR Operation against a key holding the wrong kind of value
```

MGET¶

MGET key [key ...]

返回所有(一个或多个)给定 key 的值。

如果给定的 key 里面，有某个 key 不存在，那么这个 key 返回特殊值 nil 。

因此，该命令永不失败。

可用版本：

>= 1.0.0

时间复杂度:

O(N) , N 为给定 key 的数量。

返回值：

一个包含所有给定 key 的值的列表。

```
redis> SET redis redis.com
OK
```

```
redis> SET mongodb mongodb.org
OK

redis> MGET redis mongodb
1) "redis.com"
2) "mongodb.org"

redis> MGET redis mongodb mysql        # 不存在的 mysql 返回 nil
1) "redis.com"
2) "mongodb.org"
3) (nil)
```

GETRANGE¶

GETRANGE key start end

返回 key 中字符串值的子字符串，字符串的截取范围由 start 和 end 两个偏移量决定(包括 start 和 end 在内)。

负数偏移量表示从字符串最后开始计数， -1 表示最后一个字符， -2 表示倒数第二个，以此类推。

GETRANGE 通过保证子字符串的值域(range)不超过实际字符串的值域来处理超出范围的值域请求。

注解

在 <= 2.0 的版本里，GETRANGE 被叫作 SUBSTR。

可用版本：

>= 2.4.0

时间复杂度：

O(N)， N 为要返回的字符串的长度。

复杂度最终由字符串的返回值长度决定，但因为从已有字符串中取出子字符串的操作非常廉价(cheap)，所以对于长度不大的字符串，该操作的复杂度也可看作 O(1)。

返回值：

截取得出的子字符串。

```
redis> SET greeting "hello, my friend"
```

OK

redis> GETRANGE greeting 0 4 # 返回索引 0-4 的字符，包括 4。
"hello"

redis> GETRANGE greeting -1 -5 # 不支持回绕操作
""

redis> GETRANGE greeting -3 -1 # 负数索引
"end"

redis> GETRANGE greeting 0 -1 # 从第一个到最后一个
"hello, my friend"

redis> GETRANGE greeting 0 1008611 # 值域范围不超过实际字符串，超过部分自动被符略
"hello, my friend"

GETSET¶

GETSET key value

将给定 key 的值设为 value ，并返回 key 的旧值(old value)。

当 key 存在但不是字符串类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

返回给定 key 的旧值。

当 key 没有旧值时，也即是， key 不存在时，返回 nil 。

redis> GETSET db mongodb # 没有旧值，返回 nil
(nil)

redis> GET db

"mongodb"

redis> GETSET db redis # 返回旧值 mongodb
"mongodb"

redis> GET db
"redis"

模式¶

GETSET 可以和 INCR 组合使用，实现一个有原子性(atomic)复位操作的计数器(counter)。
举例来说，每次当某个事件发生时，进程可能对一个名为 mycount 的 key 调用 INCR 操作，通常我们还要在一个原子时间内同时完成获得计数器的值和将计数器值复位为 0 两个操作。
可以用命令 GETSET mycounter 0 来实现这一目标。

redis> INCR mycount
(integer) 11

redis> GETSET mycount 0 # 一个原子内完成 GET mycount 和 SET mycount 0 操作
"11"

redis> GET mycount # 计数器被重置
"0"

STRLEN¶

STRLEN key
返回 key 所储存的字符串值的长度。
当 key 储存的不是字符串值时，返回一个错误。

可用版本：
>= 2.2.0

复杂度：

O(1)

返回值：

字符串值的长度。

当 key 不存在时，返回 0 。

获取字符串的长度

redis> SET mykey "Hello world"
OK

redis> STRLEN mykey
(integer) 11

不存在的 key 长度为 0

redis> STRLEN nonexisting
(integer) 0

DECR¶

DECR key

将 key 中储存的数字值减一。

如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 DECR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，请参见 INCR 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 DECR 命令之后 key 的值。

对存在的数字值 key 进行 DECR

```
redis> SET failure_times 10
OK

redis> DECR failure_times
(integer) 9


# 对不存在的 key 值进行 DECR

redis> EXISTS count
(integer) 0

redis> DECR count
(integer) -1


# 对存在但不是数值的 key 进行 DECR

redis> SET company YOUR_CODE_SUCKS.LLC
OK

redis> DECR company
(error) ERR value is not an integer or out of range
```

DECRBY¶

DECRBY key decrement

将 key 所储存的值减去减量 decrement 。

如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 DECRBY 操作。

如果值包含错误的类型 ,或字符串类型的值不能表示为数字 ,那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于更多递增(increment) / 递减(decrement)操作的更多信息，请参见 INCR 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

减去 decrement 之后， key 的值。

对已存在的 key 进行 DECRBY

```
redis> SET count 100
OK

redis> DECRBY count 20
(integer) 80
```

对不存在的 key 进行 DECRBY

```
redis> EXISTS pages
(integer) 0

redis> DECRBY pages 10
(integer) -10
```

INCR¶

INCR key

将 key 中储存的数字值增一。

如果 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。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

注解

这是一个针对字符串的操作，因为 Redis 没有专用的整数类型，所以 key 内储存的字符串被解释为十进制 64 位有符号整数来执行 INCR 操作。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 INCR 命令之后 key 的值。

```
redis> SET page_view 20
OK

redis> INCR page_view
(integer) 21

redis> GET page_view      # 数字值在 Redis 中以字符串的形式保存
"21"
```

模式：计数器¶

计数器是 Redis 的原子性自增操作可实现的最直观的模式了，它的想法相当简单：每当某个操作发生时，向 Redis 发送一个 INCR 命令。

比如在一个 web 应用程序中，如果想知道用户在一年中每天的点击量，那么只要将用户 ID 以及相关的日期信息作为键，并在每次用户点击页面时，执行一次自增操作即可。

比如用户名是 peter ，点击时间是 2012 年 3 月 22 日，那么执行命令 INCR peter::2012.3.22 。

可以用以下几种方式扩展这个简单的模式：

- 可以通过组合使用 INCR 和 *EXPIRE* ，来达到只在规定的生存时间内进行计数(counting)的目的。
- 客户端可以通过使用 *GETSET* 命令原子性地获取计数器的当前值并将计数器清零，更多信息请参考 *GETSET* 命令。

- 使用其他自增/自减操作，比如 *DECR* 和 *INCRBY* ，用户可以通过执行不同的操作增加或减少计数器的值，比如在游戏中的记分器就可能用到这些命令。

模式：限速器¶

限速器是特殊化的计算器，它用于限制一个操作可以被执行的速率(rate)。
限速器的典型用法是限制公开 API 的请求次数，以下是一个限速器实现示例，它将 API 的最大请求数限制在每个 IP 地址每秒钟十个之内：

```
FUNCTION LIMIT_API_CALL(ip)
ts = CURRENT_UNIX_TIME()
keyname = ip+":"+ts
current = GET(keyname)
IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
ELSE
    MULTI
        INCR(keyname,1)
        EXPIRE(keyname,10)
    EXEC
    PERFORM_API_CALL()
END
```

这个实现每秒钟为每个 IP 地址使用一个不同的计数器，并用 *EXPIRE* 命令设置生存时间(这样 Redis 就会负责自动删除过期的计数器)。
注意 ,我们使用事务打包执行 *INCR* 命令和 *EXPIRE* 命令 ,避免引入竞争条件，保证每次调用 API 时都可以正确地对计数器进行自增操作并设置生存时间。
以下是另一个限速器实现：

```
FUNCTION LIMIT_API_CALL(ip):
current = GET(ip)
IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
ELSE
    value = INCR(ip)
```

```
    IF value == 1 THEN
        EXPIRE(value,1)
    END
    PERFORM_API_CALL()
END
```

这个限速器只使用单个计数器，它的生存时间为一秒钟，如果在一秒钟内，这个计数器的值大于 10 的话，那么访问就会被禁止。
这个新的限速器在思路方面是没有问题的，但它在实现方面不够严谨，如果我们仔细观察一下的话，就会发现在 *INCR* 和 *EXPIRE* 之间存在着一个竞争条件，假如客户端在执行 *INCR* 之后，因为某些原因(比如客户端失败)而忘记设置 *EXPIRE* 的话，那么这个计数器就会一直存在下去，造成每个用户只能访问 10 次，噢，这简直是个灾难！
要消灭这个实现中的竞争条件，我们可以将它转化为一个 Lua 脚本，并放到 Redis 中运行(这个方法仅限于 Redis 2.6 及以上的版本)：

```
local current
current = redis.call("incr",KEYS[1])
if tonumber(current) == 1 then
    redis.call("expire",KEYS[1],1)
end
```

通过将计数器作为脚本放到 Redis 上运行，我们保证了 *INCR* 和 *EXPIRE* 两个操作的原子性，现在这个脚本实现不会引入竞争条件，它可以运作的很好。
关于在 Redis 中运行 Lua 脚本的更多信息，请参考 *EVAL* 命令。
还有另一种消灭竞争条件的方法，就是使用 Redis 的列表结构来代替 *INCR* 命令，这个方法无须脚本支持，因此它在 Redis 2.6 以下的版本也可以运行得很好：

```
FUNCTION LIMIT_API_CALL(ip)
current = LLEN(ip)
IF current > 10 THEN
    ERROR "too many requests per second"
ELSE
    IF EXISTS(ip) == FALSE
```

```
        MULTI
            RPUSH(ip,ip)
            EXPIRE(ip,1)
        EXEC
    ELSE
        RPUSHX(ip,ip)
    END
    PERFORM_API_CALL()
END
```

新的限速器使用了列表结构作为容器， *LLEN* 用于对访问次数进行检查，一个事务包裹着 *RPUSH* 和 *EXPIRE* 两个命令，用于在第一次执行计数时创建列表，并正确设置地设置过期时间，最后， *RPUSHX* 在后续的计数操作中进行增加操作。

INCRBY¶

INCRBY key increment

将 key 所储存的值加上增量 increment 。

如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCRBY 命令。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，参见 INCR 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

加上 increment 之后， key 的值。

```
# key 存在且是数字值

redis> SET rank 50
OK
```

```
redis> INCRBY rank 20
(integer) 70

redis> GET rank
"70"

# key 不存在时

redis> EXISTS counter
(integer) 0

redis> INCRBY counter 30
(integer) 30

redis> GET counter
"30"

# key 不是数字值时

redis> SET book "long long ago..."
OK

redis> INCRBY book 200
(error) ERR value is not an integer or out of range
```

INCRBYFLOAT¶

INCRBYFLOAT key increment

为 key 中所储存的值加上浮点数增量 increment 。

如果 key 不存在，那么 INCRBYFLOAT 会先将 key 的值设为 0 ，再执行加法操作。

如果命令执行成功，那么 key 的值会被更新为（执行加法之后的）新值，并且新值会以字符串的形式返回给调用者。

无论是 key 的值，还是增量 increment ，都可以使用像 2.0e7 、 3e5 、 90e-2 那样的指数符号(exponential notation)来表示，但是，**执行 INCRBYFLOAT 命令之后的值**总是以同样的形式储存，也即是，它们总是由一个数字，一个（可选的）小数点和一个任意位的小数部分组成（比如 3.14 、 69.768 ，诸如此类)，小数部分尾随的 0 会被移除，如果有需要的话，还会将浮点数改为整数（比如 3.0 会被保存成 3 ）。

除此之外，无论加法计算所得的浮点数的实际精度有多长， INCRBYFLOAT 的计算结果也最多只能表示小数点的后十七位。

当以下任意一个条件发生时，返回一个错误：

- key 的值不是字符串类型(因为 Redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- key 当前的值或者给定的增量 increment 不能解释(parse)为双精度浮点数(double precision floating point number）

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行命令之后 key 的值。

```
# 值和增量都不是指数符号

redis> SET mykey 10.50
OK

redis> INCRBYFLOAT mykey 0.1
"10.6"

# 值和增量都是指数符号

redis> SET mykey 314e-2
OK

redis> GET mykey                    # 用 SET 设置的值可以是指数符号
"314e-2"
```

redis> INCRBYFLOAT mykey 0 # 但执行 INCRBYFLOAT 之后格式会被改成非指数符号
"3.14"

可以对整数类型执行

redis> SET mykey 3
OK

redis> INCRBYFLOAT mykey 1.1
"4.1"

后跟的 0 会被移除

redis> SET mykey 3.0
OK

redis> GET mykey   # SET 设置的值小数部分可以是 0
"3.0"

redis> INCRBYFLOAT mykey 1.000000000000000000000   # 但 INCRBYFLOAT 会将无用的 0 忽略掉，有需要的话，将浮点变为整数
"4"

redis> GET mykey
"4"

SETBIT¶

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)。

位的设置或清除取决于 value 参数，可以是 0 也可以是 1 。

当 key 不存在时，自动生成一个新的字符串值。
字符串会进行伸展(grown)以确保它可以将 value 保存在指定的偏移量上。当字符串值进行伸展时，空白位置以 0 填充。
offset 参数必须大于或等于 0 ，小于 2^32 (bit 映射被限制在 512 MB 之内)。

警告

对使用大的 offset 的 SETBIT 操作来说，内存分配可能造成 Redis 服务器被阻塞。具体参考 SETRANGE 命令，warning(警告)部分。

可用版本：
>= 2.2.0
时间复杂度:
O(1)
返回值：
指定偏移量原来储存的位。

```
redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1

redis> GETBIT bit 100     # bit 默认被初始化为 0
(integer) 0
```

GETBIT¶

GETBIT key offset
对 key 所储存的字符串值，获取指定偏移量上的位(bit)。
当 offset 比字符串值的长度大，或者 key 不存在时，返回 0 。

可用版本：
>= 2.2.0
时间复杂度：
O(1)

返回值：

字符串值指定偏移量上的位(bit)。

```
# 对不存在的 key 或者不存在的 offset 进行 GETBIT， 返回 0

redis> EXISTS bit
(integer) 0

redis> GETBIT bit 10086
(integer) 0


# 对已存在的 offset 进行 GETBIT

redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1
```

哈希表(Hash)¶

常用命令：

hget,hset,hgetall 等.

应用场景：

比如,我们存储供应商酒店价格的时候可以采取此结构,用酒店编码作为 Key, RatePlan+RoomType 作为 Filed,价格信息作为 Value.

实现方式：

Hash 对应 Value 内部实际就是一个 HashMap,实际这里会有 2 种不同实现,这个 Hash 的成员比较少时 Redis 为了节省内存会采用类似一维数组的方式来紧凑存储,而不会采用真正的 HashMap 结构,对应的 value redisObject 的 encoding 为 zipmap,当成员数量增大时会自动转成真正的 HashMap,此时 encoding 为 ht.

HSET¶

HSET key field value

将哈希表 key 中的域 field 的值设为 value 。

如果 key 不存在，一个新的哈希表被创建并进行 HSET 操作。

如果域 field 已经存在于哈希表中，旧值将被覆盖。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果 field 是哈希表中的一个新建域，并且值设置成功，返回 1 。

如果哈希表中域 field 已经存在且旧值已被新值覆盖，返回 0 。

```
redis> HSET website google "www.g.cn"           # 设置一个新域
(integer) 1

redis> HSET website google "www.google.com" # 覆盖一个旧域
(integer) 0
```

HSETNX¶

HSETNX key field value

将哈希表 key 中的域 field 的值设置为 value ，当且仅当域 field 不存在。

若域 field 已经存在，该操作无效。

如果 key 不存在，一个新哈希表被创建并执行 HSETNX 命令。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 1 。

如果给定域已经存在且没有操作被执行，返回 0 。

```
redis> HSETNX nosql key-value-store redis
(integer) 1

redis> HSETNX nosql key-value-store redis   # 操作无效，域 key-value-store 已存在
(integer) 0
```

HMSET¶

HMSET key field value [field value ...]

同时将多个 field-value (域-值)对设置到哈希表 key 中。

此命令会覆盖哈希表中已存在的域。

如果 key 不存在，一个空哈希表被创建并执行 HMSET 操作。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， N 为 field-value 对的数量。

返回值：

如果命令执行成功，返回 OK 。

当 key 不是哈希表(hash)类型时，返回一个错误。

```
redis> HMSET website google www.google.com yahoo www.yahoo.com
OK
```

```
redis> HGET website google
"www.google.com"

redis> HGET website yahoo
"www.yahoo.com"
```

HGET¶

HGET key field

返回哈希表 key 中给定域 field 的值。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

给定域的值。

当给定域不存在或是给定 key 不存在时，返回 nil 。

```
redis> HSET site redis redis.com
(integer) 1

redis> HGET site redis
"redis.com"

redis> HGET site mysql
(nil)
```

HMGET¶

HMGET key field [field ...]

返回哈希表 key 中，一个或多个给定域的值。

如果给定的域不存在于哈希表，那么返回一个 nil 值。

因为不存在的 key 被当作一个空哈希表来处理，所以对一个不存在的 key 进行 HMGET 操作将返回一个只带有 nil 值的表。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， N 为给定域的数量。

返回值：

一个包含多个给定域的关联值的表 ,表值的排列顺序和给定域参数的请求顺序一样。

```
redis> HMSET pet dog "doudou" cat "nounou"       # 一次设置多个域
OK

redis> HMGET pet dog cat fake_pet                # 返回值的顺序和传入参数的顺序一样
1) "doudou"
2) "nounou"
3) (nil)                                         # 不存在的域返回 nil 值
```

HGETALL¶

HGETALL key

返回哈希表 key 中，所有的域和值。

在返回值里，紧跟每个域名(field name)之后是域的值(value)，所以返回值的长度是哈希表大小的两倍。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， N 为哈希表的大小。

返回值：

以列表形式返回哈希表的域和域的值。

若 key 不存在，返回空列表。

```
redis> HSET people jack "Jack Sparrow"
(integer) 1

redis> HSET people gump "Forrest Gump"
(integer) 1

redis> HGETALL people
1) "jack"                # 域
```

2) "Jack Sparrow" # 值
3) "gump"
4) "Forrest Gump"

HDEL¶

HDEL key field [field ...]

删除哈希表 key 中的一个或多个指定域，不存在的域将被忽略。

注解

在 Redis2.4 以下的版本里， HDEL 每次只能删除单个域，如果你需要在一个原子时间内删除多个域，请将命令包含在 *MULTI* / *EXEC* 块内。

可用版本：

>= 2.0.0

时间复杂度:

O(N)， N 为要删除的域的数量。

返回值:

被成功移除的域的数量，不包括被忽略的域。

测试数据

redis> HGETALL abbr
1) "a"
2) "apple"
3) "b"
4) "banana"
5) "c"
6) "cat"
7) "d"
8) "dog"

删除单个域

redis> HDEL abbr a
(integer) 1

删除不存在的域

```
redis> HDEL abbr not-exists-field
(integer) 0

# 删除多个域

redis> HDEL abbr b c
(integer) 2

redis> HGETALL abbr
1) "d"
2) "dog"
```

HLEN¶

HLEN key

返回哈希表 key 中域的数量。

时间复杂度：

O(1)

返回值：

哈希表中域的数量。

当 key 不存在时，返回 0 。

```
redis> HSET db redis redis.com
(integer) 1

redis> HSET db mysql mysql.com
(integer) 1

redis> HLEN db
(integer) 2

redis> HSET db mongodb mongodb.org
(integer) 1

redis> HLEN db
```

(integer) 3

HEXISTS¶

HEXISTS key field

查看哈希表 key 中，给定域 field 是否存在。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果哈希表含有给定域，返回 1 。

如果哈希表不含有给定域，或 key 不存在，返回 0 。

```
redis> HEXISTS phone myphone
(integer) 0

redis> HSET phone myphone nokia-1110
(integer) 1

redis> HEXISTS phone myphone
(integer) 1
```

HINCRBY¶

HINCRBY key field increment

为哈希表 key 中的域 field 的值加上增量 increment 。

增量也可以为负数，相当于对给定域进行减法操作。

如果 key 不存在，一个新的哈希表被创建并执行 HINCRBY 命令。

如果域 field 不存在，那么在执行命令前，域的值被初始化为 0 。

对一个储存字符串值的域 field 执行 HINCRBY 命令将造成一个错误。

本操作的值被限制在 64 位(bit)有符号数字表示之内。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

执行 HINCRBY 命令之后，哈希表 key 中域 field 的值。

```
# increment 为正数

redis> HEXISTS counter page_view      # 对空域进行设置
(integer) 0

redis> HINCRBY counter page_view 200
(integer) 200

redis> HGET counter page_view
"200"


# increment 为负数

redis> HGET counter page_view
"200"

redis> HINCRBY counter page_view -50
(integer) 150

redis> HGET counter page_view
"150"


# 尝试对字符串值的域执行 HINCRBY 命令

redis> HSET myhash string hello,world          # 设定一个字符串值
(integer) 1

redis> HGET myhash string
"hello,world"
```

```
redis> HINCRBY myhash string 1                    # 命令执行失败，错误。
(error) ERR hash value is not an integer

redis> HGET myhash string                         # 原值不变
"hello,world"
```

HINCRBYFLOAT¶

HINCRBYFLOAT key field increment

为哈希表 key 中的域 field 加上浮点数增量 increment 。

如果哈希表中没有域 field ，那么 HINCRBYFLOAT 会先将域 field 的值设为 0 ，然后再执行加法操作。

如果键 key 不存在，那么 HINCRBYFLOAT 会先创建一个哈希表，再创建域 field ，最后再执行加法操作。

当以下任意一个条件发生时，返回一个错误：

- 域 field 的值不是字符串类型(因为 redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- 域 field 当前的值或给定的增量 increment 不能解释(parse)为双精度浮点数(double precision floating point number)

HINCRBYFLOAT 命令的详细功能和 *INCRBYFLOAT* 命令类似，请查看 *INCRBYFLOAT* 命令获取更多相关信息。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行加法操作之后 field 域的值。

```
# 值和增量都是普通小数

redis> HSET mykey field 10.50
(integer) 1
redis> HINCRBYFLOAT mykey field 0.1
"10.6"

# 值和增量都是指数符号
```

```
redis> HSET mykey field 5.0e3
(integer) 0
redis> HINCRBYFLOAT mykey field 2.0e2
"5200"

# 对不存在的键执行 HINCRBYFLOAT

redis> EXISTS price
(integer) 0
redis> HINCRBYFLOAT price milk 3.5
"3.5"
redis> HGETALL price
1) "milk"
2) "3.5"

# 对不存在的域进行 HINCRBYFLOAT

redis> HGETALL price
1) "milk"
2) "3.5"
redis> HINCRBYFLOAT price coffee 4.5     # 新增 coffee 域
"4.5"
redis> HGETALL price
1) "milk"
2) "3.5"
3) "coffee"
4) "4.5"
```

HKEYS¶

HKEYS key

返回哈希表 key 中的所有域。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， N 为哈希表的大小。

返回值：

一个包含哈希表中所有域的表。

当 key 不存在时，返回一个空表。

哈希表非空

```
redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HKEYS website
1) "google"
2) "yahoo"
```

空哈希表/key 不存在

```
redis> EXISTS fake_key
(integer) 0

redis> HKEYS fake_key
(empty list or set)
```

HVALS¶

HVALS key

返回哈希表 key 中所有域的值。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， N 为哈希表的大小。

返回值：

一个包含哈希表中所有值的表。

当 key 不存在时，返回一个空表。

非空哈希表

```
redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HVALS website
1) "www.google.com"
2) "www.yahoo.com"


# 空哈希表/不存在的 key

redis> EXISTS not_exists
(integer) 0

redis> HVALS not_exists
(empty list or set)
```

表(List)¶

常用命令:
lpush,rpush,lpop,rpop,lrang 等.

应用场景:
Redis list 应用场景非常多,也是 Redis 最重要的数据结构之一,比如 twitter 的关注列表,粉丝列表等都可以用 Redis 的 list 结构来实现.

实现方式:
Redis list 的实现为一个双向链表,即可以支持反向查找和遍历,更方便操作,不过带来了部分额外的内存开销,Redis 内部的很多实现,包括发送缓冲队列等也都是用的这个数据结构.

头元素和尾元素

头元素指的是列表左端/前端第一个元素，尾元素指的是列表右端/后端第一个元素。

举个例子，列表 list 包含三个元素：x, y, z，其中 x 是头元素，而 z 则是尾元素。

空列表

指不包含任何元素的列表，Redis 将不存在的 key 也视为空列表。

LPUSH¶

LPUSH key value [value ...]

将一个或多个值 value 插入到列表 key 的表头

如果有多个 value 值，那么各个 value 值按从左到右的顺序依次插入到表头：比如对一个空列表 mylist 执行 LPUSH mylist a b c ，则结果列表为 c b a ，等同于执行执行命令 LPUSH mylist a 、 LPUSH mylist b 、 LPUSH mylist c 。

如果 key 不存在，一个空列表会被创建并执行 LPUSH 操作。

当 key 存在但不是列表类型时，返回一个错误。

注解

在 Redis 2.4 版本以前的 LPUSH 命令，都只接受单个 value 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 LPUSH 命令后，列表的长度。

```
# 加入单个元素

redis> LPUSH languages python
(integer) 1


# 加入重复元素

redis> LPUSH languages python
(integer) 2

redis> LRANGE languages 0 -1        # 列表允许重复元素
1) "python"
2) "python"


# 加入多个元素

redis> LPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "c"
2) "b"
3) "a"
```

LPUSHX¶

LPUSHX key value

将值 value 插入到列表 key 的表头，当且仅当 key 存在并且是一个列表。

和 *LPUSH* 命令相反，当 key 不存在时， LPUSHX 命令什么也不做。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

LPUSHX 命令执行之后，表的长度。

```
# 对空列表执行 LPUSHX

redis> LLEN greet                       # greet 是一个空列表
(integer) 0

redis> LPUSHX greet "hello"             # 尝试 LPUSHX，失败，因为列表为空
(integer) 0


# 对非空列表执行 LPUSHX

redis> LPUSH greet "hello"              # 先用 LPUSH 创建一个有一个元素的列表
(integer) 1

redis> LPUSHX greet "good morning"      # 这次 LPUSHX 执行成功
(integer) 2

redis> LRANGE greet 0 -1
1) "good morning"
2) "hello"
```

RPUSH¶

RPUSH key value [value ...]

将一个或多个值 value 插入到列表 key 的表尾(最右边)。

如果有多个 value 值，那么各个 value 值按从左到右的顺序依次插入到表尾：比如对一个空列表 mylist 执行 RPUSH mylist a b c ，得出的结果列表为 a b c ，等同于执行命令 RPUSH mylist a 、 RPUSH mylist b 、 RPUSH mylist c 。

如果 key 不存在，一个空列表会被创建并执行 RPUSH 操作。

当 key 存在但不是列表类型时，返回一个错误。

注解

在 Redis 2.4 版本以前的 RPUSH 命令，都只接受单个 value 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 RPUSH 操作后，表的长度。

```
# 添加单个元素

redis> RPUSH languages c
(integer) 1


# 添加重复元素

redis> RPUSH languages c
(integer) 2

redis> LRANGE languages 0 -1 # 列表允许重复元素
1) "c"
2) "c"
```

添加多个元素

redis> RPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "a"
2) "b"
3) "c"

RPUSHX¶

RPUSHX key value

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。

可用版本：
>= 2.2.0
时间复杂度：
O(1)
返回值：
RPUSHX 命令执行之后，表的长度。
key 不存在

redis> LLEN greet
(integer) 0

redis> RPUSHX greet "hello" # 对不存在的 key 进行 RPUSHX ,PUSH 失败。
(integer) 0

key 存在且是一个非空列表

redis> RPUSH greet "hi" # 先用 RPUSH 插入一个元素

(integer) 1

redis> RPUSHX greet "hello"　　# greet 现在是一个列表类型，RPUSHX 操作成功。
(integer) 2

redis> LRANGE greet 0 -1
1) "hi"
2) "hello"

LPOP¶

LPOP key
移除并返回列表 key 的头元素。

可用版本：
　　>= 1.0.0
时间复杂度：
　　O(1)
返回值：
　　列表的头元素。
　　当 key 不存在时，返回 nil 。
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> LPOP course　# 移除头元素
"algorithm001"

RPOP¶

RPOP key

移除并返回列表 key 的尾元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的尾元素。

当 key 不存在时，返回 nil 。

```
redis> RPUSH mylist "one"
(integer) 1

redis> RPUSH mylist "two"
(integer) 2

redis> RPUSH mylist "three"
(integer) 3

redis> RPOP mylist              # 返回被弹出的元素
"three"

redis> LRANGE mylist 0 -1       # 列表剩下的元素
1) "one"
2) "two"
```

BLPOP¶

BLPOP key [key ...] timeout

BLPOP 是列表的阻塞式(blocking)弹出原语。

它是 *LPOP* 命令的阻塞版本，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，直到等待超时或发现可弹出元素为止。

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。
假设现在有 job 、 command 和 request 三个列表，其中 job 不存在，command 和 request 都持有非空列表。考虑以下命令：
BLPOP job command request 0
BLPOP 保证返回的元素来自 command ，因为它是按”查找 job -> 查找 command -> 查找 request “这样的顺序，第一个找到的非空列表。

```
redis> DEL job command request                  # 确保 key 都被删除
(integer) 0

redis> LPUSH command "update system..."  # 为 command 列表增加一个值
(integer) 1

redis> LPUSH request "visit page"            # 为 request 列表增加一个值
(integer) 1

redis> BLPOP job command request 0          # job 列表为空，被跳过，紧接着 command 列表的第一个元素被弹出。
1) "command"                                        # 弹出元素所属的列表
2) "update system..."                          # 弹出元素所属的值
```

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，直到等待超时，或有另一个客户端对给定 key 的任意一个执行 *LPUSH* 或 *RPUSH* 命令为止。
超时参数 timeout 接受一个以秒为单位的数字作为值。超时参数设为 0 表示阻塞时间可以无限期延长(block indefinitely) 。

```
redis> EXISTS job                               # 确保两个 key 都不存在
```

(integer) 0
redis> EXISTS command
(integer) 0

redis> BLPOP job command 300 # 因为 key 一开始不存在，所以操作会被阻塞，直到另一客户端对 job 或者 command 列表进行 PUSH 操作。
1) "job" # 这里被 push 的是 job
2) "do my home work" # 被弹出的值
(26.26s) # 等待的秒数

redis> BLPOP job command 5 # 等待超时的情况
(nil)
(5.66s) # 等待的秒数

相同的 key 被多个客户端同时阻塞

相同的 key 可以被多个客户端同时阻塞。

不同的客户端被放进一个队列中，按『先阻塞先服务』(first-BLPOP，first-served)的顺序为 key 执行 BLPOP 命令。

在 MULTI/EXEC 事务中的 BLPOP

BLPOP 可以用于流水线(pipline,批量地发送多个命令并读入多个回复)，但把它用在 *MULTI* / *EXEC* 块当中没有意义。因为这要求整个服务器被阻塞以保证块执行时的原子性，该行为阻止了其他客户端执行 *LPUSH* 或 *RPUSH* 命令。

因此，一个被包裹在 *MULTI* / *EXEC* 块内的 BLPOP 命令，行为表现得就像 *LPOP* 一样，对空列表返回 nil ，对非空列表弹出列表元素，不进行任何阻塞操作。

对非空列表进行操作

redis> RPUSH job programming
(integer) 1

redis> MULTI
OK

```
redis> BLPOP job 30
QUEUED

redis> EXEC                 # 不阻塞，立即返回
1) 1) "job"
   2) "programming"


# 对空列表进行操作

redis> LLEN job          # 空列表
(integer) 0

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC              # 不阻塞，立即返回
1) (nil)
```

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果列表为空，返回一个 nil 。

否则，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 key ，第二个元素是被弹出元素的值。

模式：事件提醒¶

有时候，为了等待一个新元素到达数据中，需要使用轮询的方式对数据进行探查。另一种更好的方式是，使用系统提供的阻塞原语，在新元素到达时立即进行处理，而新元素还没到达时，就一直阻塞住，避免轮询占用资源。

对于 Redis ,我们似乎需要一个阻塞版的 *SPOP* 命令 ,但实际上 ,使用 BLPOP 或者 *BRPOP* 就能很好地解决这个问题。
使用元素的客户端(消费者)可以执行类似以下的代码：

```
LOOP forever
    WHILE SPOP(key) returns elements
        ... process elements ...
    END
    BRPOP helper_key
END
```

添加元素的客户端(消费者)则执行以下代码：

```
MULTI
    SADD key element
    LPUSH helper_key x
EXEC
```

BRPOP¶

BRPOP key [key ...] timeout
BRPOP 是列表的阻塞式(blocking)弹出原语。
它是 *RPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BRPOP 命令阻塞，直到等待超时或发现可弹出元素为止。
当给定多个 key 参数时，按参数 key 的先后顺序依次检查各个列表，弹出第一个非空列表的尾部元素。
关于阻塞操作的更多信息，请查看 *BLPOP* 命令， BRPOP 除了弹出元素的位置和 *BLPOP* 不同之外，其他表现一致。

可用版本：
>= 2.0.0
时间复杂度：
O(1)
返回值：
假如在指定时间内没有任何元素被弹出，则返回一个 nil 和等待时长。
反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 key ，第二

个元素是被弹出元素的值。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> BRPOP course 30
1) "course"                    # 弹出元素的 key
2) "c++101"                    # 弹出元素的值
```

LLEN¶

LLEN key

返回列表 key 的长度。

如果 key 不存在，则 key 被解释为一个空列表，返回 0 .

如果 key 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表 key 的长度。

```
# 空列表

redis> LLEN job
(integer) 0


# 非空列表
```

redis> LPUSH job "cook food"
(integer) 1

redis> LPUSH job "have lunch"
(integer) 2

redis> LLEN job
(integer) 2

LRANGE¶

LRANGE key start stop

返回列表 key 中指定区间内的元素，区间以偏移量 start 和 stop 指定。

下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。

注意 LRANGE 命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表，对该列表执行 LRANGE list 0 10 ，结果是一个包含 11 个元素的列表，这表明 stop 下标也在 LRANGE 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如 Ruby 的 Range.new 、 Array#slice 和 Python 的 range() 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 start 下标比列表的最大下标 end (LLEN list 减去 1)还要大，或者 start > stop ， LRANGE 返回一个空列表。

如果 stop 下标比 end 下标还要大，Redis 将 stop 的值设置为 end 。

可用版本：

>= 1.0.0

时间复杂度:

O(S+N)， S 为偏移量 start ， N 为指定区间内元素的数量。

返回值:

一个列表，包含指定区间内的元素。

redis> RPUSH fp-language lisp

(integer) 1

redis> LRANGE fp-language 0 0
1) "lisp"

redis> RPUSH fp-language scheme
(integer) 2

redis> LRANGE fp-language 0 1
1) "lisp"
2) "scheme"

LREM¶

LREM key count value

根据参数 count 的值，移除列表中与参数 value 相等的元素。

count 的值可以是以下几种：

- count > 0：从表头开始向表尾搜索，移除与 value 相等的元素，数量为 count 。
- count < 0：从表尾开始向表头搜索，移除与 value 相等的元素，数量为 count 的绝对值。
- count = 0：移除表中所有与 value 相等的值。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为列表的长度。

返回值：

被移除元素的数量。

因为不存在的 key 被视作空表(empty list)，所以当 key 不存在时， LREM 命令总是返回 0 。

先创建一个表，内容排列是
morning hello morning helllo morning

redis> LPUSH greet "morning"
(integer) 1

```
redis> LPUSH greet "hello"
(integer) 2
redis> LPUSH greet "morning"
(integer) 3
redis> LPUSH greet "hello"
(integer) 4
redis> LPUSH greet "morning"
(integer) 5

redis> LRANGE greet 0 4             # 查看所有元素
1) "morning"
2) "hello"
3) "morning"
4) "hello"
5) "morning"

redis> LREM greet 2 morning         # 移除从表头到表尾，最先发现的两个 morning
(integer) 2                         # 两个元素被移除

redis> LLEN greet                   # 还剩 3 个元素
(integer) 3

redis> LRANGE greet 0 2
1) "hello"
2) "hello"
3) "morning"

redis> LREM greet -1 morning        # 移除从表尾到表头，第一个 morning
(integer) 1

redis> LLEN greet                   # 剩下两个元素
(integer) 2
```

```
redis> LRANGE greet 0 1
1) "hello"
2) "hello"

redis> LREM greet 0 hello          # 移除表中所有 hello
(integer) 2                        # 两个 hello 被移除

redis> LLEN greet
(integer) 0
```

LSET¶

LSET key index value

将列表 key 下标为 index 的元素的值甚至为 value 。

当 index 参数超出范围，或对一个空列表(key 不存在)进行 LSET 时，返回一个错误。

关于列表下标的更多信息，请参考 *LINDEX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

对头元素或尾元素进行 LSET 操作，复杂度为 O(1)。

其他情况下，为 O(N)， N 为列表的长度。

返回值：

操作成功返回 ok ，否则返回错误信息。

```
# 对空列表(key 不存在)进行 LSET

redis> EXISTS list
(integer) 0

redis> LSET list 0 item
(error) ERR no such key
```

```
# 对非空列表进行 LSET

redis> LPUSH job "cook food"
(integer) 1

redis> LRANGE job 0 0
1) "cook food"

redis> LSET job 0 "play game"
OK

redis> LRANGE job  0 0
1) "play game"


# index 超出范围

redis> LLEN list                          # 列表长度为 1
(integer) 1

redis> LSET list 3 'out of range'
(error) ERR index out of range
```

LTRIM¶

LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

举个例子，执行命令 LTRIM list 0 2 ，表示只保留列表 list 的前三个元素，其余元素全部删除。

下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。

当 key 不是列表类型时，返回一个错误。
LTRIM 命令通常和 *LPUSH* 命令或 *RPUSH* 命令配合使用，举个例子：

```
LPUSH log newest_log
LTRIM log 0 99
```

这个例子模拟了一个日志程序，每次将最新日志 newest_log 放到 log 列表中，并且只保留最新的 100 项。注意当这样使用 LTRIM 命令时，时间复杂度是 O(1)，因为平均情况下，每次只有一个元素被移除。

注意 LTRIM 命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表 list ，对该列表执行 LTRIM list 0 10 ，结果是一个包含 11 个元素的列表，这表明 stop 下标也在 LTRIM 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如 Ruby 的 Range.new 、 Array#slice 和 Python 的 range() 函数。

超出范围的下标

超出范围的下标值不会引起错误。
如果 start 下标比列表的最大下标 end (LLEN list 减去 1)还要大，或者 start > stop ， LTRIM 返回一个空列表(因为 LTRIM 已经将整个列表清空)。
如果 stop 下标比 end 下标还要大，Redis 将 stop 的值设置为 end 。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 为被移除的元素的数量。

返回值:

命令执行成功时，返回 ok 。

```
# 一般情况下标

redis> LRANGE alpha 0 -1      # 建立一个 5 元素的列表
1) "h"
2) "e"
3) "l"
4) "l"
5) "o"
```

```
redis> LTRIM alpha 1 -1        # 删除索引为 0 的元素
OK

redis> LRANGE alpha 0 -1       # "h" 被删除
1) "e"
2) "l"
3) "l"
4) "o"


# stop 下标比元素的最大下标要大

redis> LTRIM alpha 1 10086
OK
redis> LRANGE alpha 0 -1
1) "l"
2) "l"
3) "o"


# start 和 stop 下标都比最大下标要大，且 start < sotp

redis> LTRIM alpha 10086 200000
OK
redis> LRANGE alpha 0 -1              # 整个列表被清空，等同于 DEL alpha
(empty list or set)


# start > stop

redis> LRANGE alpha 0 -1              # 在新建一个列表
1) "h"
2) "u"
```

3) "a"
4) "n"
5) "g"
6) "z"

redis> LTRIM alpha 10086 4
OK

redis> LRANGE alpha 0 -1 # 列表同样被清空
(empty list or set)

LINDEX¶

LINDEX key index

返回列表 key 中，下标为 index 的元素。

下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。

如果 key 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为到达下标 index 过程中经过的元素数量。

因此，对列表的头元素和尾元素执行 LINDEX 命令，复杂度为 O(1)。

返回值:

列表中下标为 index 的元素。

如果 index 参数的值不在列表的区间范围内(out of range)，返回 nil 。

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          # index 不在 mylist 的区间范围内
(nil)
```

LINSERT¶

LINSERT key BEFORE|AFTER pivot value

将值 value 插入到列表 key 当中，位于值 pivot 之前或之后。

当 pivot 不存在于列表 key 时，不执行任何操作。

当 key 不存在时， key 被视为空列表，不执行任何操作。

如果 key 不是列表类型，返回一个错误。

可用版本：

>= 2.2.0

时间复杂度:

O(N)， N 为寻找 pivot 过程中经过的元素数量。

返回值:

如果命令执行成功，返回插入操作完成之后，列表的长度。

如果没有找到 pivot ，返回 -1 。

如果 key 不存在或为空列表，返回 0 。

```
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"

对一个非空列表插入，查找一个不存在的 pivot

redis> LINSERT mylist BEFORE "go" "let's"
(integer) -1 # 失败

对一个空列表执行 LINSERT 命令

redis> EXISTS fake_list
(integer) 0

redis> LINSERT fake_list BEFORE "nono" "gogogog"
(integer) 0 # 失败

RPOPLPUSH¶

RPOPLPUSH source destination
命令 RPOPLPUSH 在一个原子时间内，执行以下两个动作：

- 将列表 source 中的最后一个元素(尾元素)弹出，并返回给客户端。
- 将 source 弹出的元素插入到列表 destination ，作为 destination 列表的的头元素。

举个例子，你有两个列表 source 和 destination ， source 列表有元素 a, b, c ， destination 列表有元素 x, y, z ，执行 RPOPLPUSH source destination 之后， source 列表包含元素 a, b ， destination 列表包含元素 c, x, y, z ，并且元素 c 会被返回给客户端。
如果 source 不存在，值 nil 被返回，并且不执行其他动作。
如果 source 和 destination 相同，则列表中的表尾元素被移动到表头，并返回该元素，可以把这种特殊情况视作列表的旋转(rotation)操作。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

被弹出的元素。

source 和 destination 不同

redis> LRANGE alpha 0 -1 # 查看所有元素
1) "a"
2) "b"
3) "c"
4) "d"

redis> RPOPLPUSH alpha reciver # 执行一次 RPOPLPUSH 看看
"d"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"
3) "c"

redis> LRANGE reciver 0 -1
1) "d"

redis> RPOPLPUSH alpha reciver # 再执行一次，证实 RPOP 和 LPUSH 的位置正确
"c"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"

redis> LRANGE reciver 0 -1

```
1) "c"
2) "d"


# source 和 destination 相同

redis> LRANGE number 0 -1
1) "1"
2) "2"
3) "3"
4) "4"

redis> RPOPLPUSH number number
"4"

redis> LRANGE number 0 -1                # 4 被旋转到了表头
1) "4"
2) "1"
3) "2"
4) "3"

redis> RPOPLPUSH number number
"3"

redis> LRANGE number 0 -1                # 这次是 3 被旋转到了表头
1) "3"
2) "4"
3) "1"
4) "2"
```

模式： 安全的队列¶

Redis 的列表经常被用作队列(queue)，用于在不同程序之间有序地交换消息(message)。一个客户端通过 *LPUSH* 命令将消息放入队列中，而另一个客户端通过 *RPOP* 或者 *BRPOP* 命令取出队列中等待时间最长的消息。
不幸的是，上面的队列方法是『不安全』的，因为在这个过程中，一个客户端可能在取出一个消息之后崩溃，而未处理完的消息也就因此丢失。
使用 RPOPLPUSH 命令(或者它的阻塞版本 *BRPOPLPUSH*)可以解决这个问题：因为它不仅返回一个消息，同时还将这个消息添加到另一个备份列表当中，如果一切正常的话，当一个客户端完成某个消息的处理之后，可以用 *LREM* 命令将这个消息从备份表删除。
最后，还可以添加一个客户端专门用于监视备份表，它自动地将超过一定处理时限的消息重新放入队列中去(负责处理该消息的客户端可能已经崩溃)，这样就不会丢失任何消息了。

模式：循环列表¶

通过使用相同的 key 作为 RPOPLPUSH 命令的两个参数，客户端可以用一个接一个地获取列表元素的方式，取得列表的所有元素，而不必像 *LRANGE* 命令那样一下子将所有列表元素都从服务器传送到客户端中(两种方式的总复杂度都是 O(N))。
以上的模式甚至在以下的两个情况下也能正常工作：

- 有多个客户端同时对同一个列表进行旋转(rotating)，它们获取不同的元素，直到所有元素都被读取完，之后又从头开始。
- 有客户端在向列表尾部(右边)添加新元素。

这个模式使得我们可以很容易实现这样一类系统：有 N 个客户端，需要连续不断地对一些元素进行处理，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的监控程序：它们需要在尽可能短的时间内，并行地检查一组网站，确保它们的可访问性。
注意，使用这个模式的客户端是易于扩展(scala)且安全(reliable)的，因为就算接收到元素的客户端失败，元素还是保存在列表里面，不会丢失，等到下个迭代来临的时候，别的客户端又可以继续处理这些元素了。

BRPOPLPUSH¶

BRPOPLPUSH source destination timeout

BRPOPLPUSH 是 *RPOPLPUSH* 的阻塞版本，当给定列表 source 不为空时，BRPOPLPUSH 的表现和 *RPOPLPUSH* 一样。

当列表 source 为空时， BRPOPLPUSH 命令将阻塞连接，直到等待超时，或有另一个客户端对 source 执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 timeout 接受一个以秒为单位的数字作为值。超时参数设为 0 表示阻塞时间可以无限期延长(block indefinitely) 。

更多相关信息，请参考 *RPOPLPUSH* 命令。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 nil 和等待时长。

反之，返回一个含有两个元素的列表，第一个元素是被弹出元素的值，第二个元素是等待时长。

```
# 非空列表

redis> BRPOPLPUSH msg reciver 500
"hello moto"                          # 弹出元素的值
(3.38s)                               # 等待时长

redis> LLEN reciver
(integer) 1

redis> LRANGE reciver 0 0
1) "hello moto"


# 空列表

redis> BRPOPLPUSH msg reciver 1
```

(nil)
(1.34s)

模式：安全队列¶

参考 *RPOPLPUSH* 命令的『安全队列』模式。

模式：循环列表¶

参考 *RPOPLPUSH* 命令的『循环列表』模式。

集合(Set)¶

常用命令：
sadd,spop,smembers,sunion 等.

应用场景：
Set 对外提供的功能与 list 类似,当你需要存储一个列表数据,又不希望出现重复数据时,set 是一个很好的选择,并且 set 提供了判断某个成员是否在一个 set 集合内的接口,这个也是 list 所不能提供的.

实现方式：
Set 的内部实现是一个 value 永远为 null 的 HashMap,实际就是通过计算 hash 的方
式来快速排除重复的,这也是 set 能提供判断一个成员是否在集合内的原因.
附录，常用集合运算：
A = {'a', 'b', 'c'}
B = {'a', 'e', 'i', 'o', 'u'}

inter(x, y): 交集，在集合 x 和集合 y 中都存在的元素。
inter(A, B) = {'a'}

union(x, y): 并集，在集合 x 中或集合 y 中的元素，如果一个元素在 x 和 y 中都出现，那只记录一次即可。
union(A,B) = {'a', 'b', 'c', 'e', 'i', 'o', 'u'}

diff(x, y): 差集，在集合 x 中而不在集合 y 中的元素。
diff(A,B) = {'b', 'c'}

card(x): 基数，一个集合中元素的数量。
card(A) = 3

空集： 基数为 0 的集合。

SADD¶

SADD key member [member ...]

将一个或多个 member 元素加入到集合 key 当中，已经存在于集合的 member 元素将被忽略。

假如 key 不存在，则创建一个只包含 member 元素作成员的集合。

当 key 不是集合类型时，返回一个错误。

注解

在 Redis2.4 版本以前， SADD 只接受单个 member 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 是被添加的元素的数量。

返回值:

被添加到集合中的新元素的数量，不包括被忽略的元素。

```
# 添加单个元素

redis> SADD bbs "discuz.net"
(integer) 1


# 添加重复元素

redis> SADD bbs "discuz.net"
(integer) 0


# 添加多个元素

redis> SADD bbs "tianya.cn" "groups.google.com"
(integer) 2

redis> SMEMBERS bbs
1) "discuz.net"
2) "groups.google.com"
```

3) "tianya.cn"

SREM¶

SREM key member [member ...]

移除集合 key 中的一个或多个 member 元素，不存在的 member 元素会被忽略。

当 key 不是集合类型，返回一个错误。

注解

在 Redis 2.4 版本以前， SREM 只接受单个 member 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 为给定 member 元素的数量。

返回值:

被成功移除的元素的数量，不包括被忽略的元素。

```
# 测试数据

redis> SMEMBERS languages
1) "c"
2) "lisp"
3) "python"
4) "ruby"


# 移除单个元素

redis> SREM languages ruby
(integer) 1


# 移除不存在元素

redis> SREM languages non-exists-language
(integer) 0
```

移除多个元素

redis> SREM languages lisp python c
(integer) 3

redis> SMEMBERS languages
(empty list or set)

SMEMBERS¶

SMEMBERS key

返回集合 key 中的所有成员。

不存在的 key 被视为空集合。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 为集合的基数。

返回值:

集合中的所有成员。

key 不存在或集合为空

redis> EXISTS not_exists_key
(integer) 0

redis> SMEMBERS not_exists_key
(empty list or set)

非空集合

redis> SADD language Ruby Python Clojure
(integer) 3

redis> SMEMBERS language
1) "Python"
2) "Ruby"
3) "Clojure"

SISMEMBER¶

SISMEMBER key member

判断 member 元素是否集合 key 的成员。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 member 元素是集合的成员，返回 1 。

如果 member 元素不是集合的成员，或 key 不存在，返回 0 。

redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SISMEMBER joe's_movies "bet man"
(integer) 0

redis> SISMEMBER joe's_movies "Fast Five"
(integer) 1

SCARD¶

SCARD key

返回集合 key 的基数(集合中元素的数量)。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值：

集合的基数。

当 key 不存在时，返回 0 。

```
redis> SADD tool pc printer phone
(integer) 3

redis> SCARD tool    # 非空集合
(integer) 3

redis> DEL tool
(integer) 1

redis> SCARD tool    # 空集合
(integer) 0
```

SMOVE¶

SMOVE source destination member

将 member 元素从 source 集合移动到 destination 集合。

SMOVE 是原子性操作。

如果 source 集合不存在或不包含指定的 member 元素，则 SMOVE 命令不执行任何操作，仅返回 0 。否则， member 元素从 source 集合中被移除，并添加到 destination 集合中去。

当 destination 集合已经包含 member 元素时， SMOVE 命令只是简单地将 source 集合中的 member 元素删除。

当 source 或 destination 不是集合类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 member 元素被成功移除，返回 1 。

如果 member 元素不是 source 集合的成员，并且没有任何操作对 destination 集合执行，那么返回 0 。

```
redis> SMEMBERS songs
1) "Billie Jean"
```

2) "Believe Me"

redis> SMEMBERS my_songs
(empty list or set)

redis> SMOVE songs my_songs "Believe Me"
(integer) 1

redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"

SPOP¶

SPOP key

移除并返回集合中的一个随机元素。

如果只想获取一个随机元素，但不想该元素从集合中被移除的话，可以使用 *SRANDMEMBER* 命令。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

被移除的随机元素。

当 key 不存在或 key 是空集时，返回 nil 。

redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"
3) "Redis"

redis> SPOP db
"Redis"

```
redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"

redis> SPOP db
"MySQL"

redis> SMEMBERS db
1) "MongoDB"
```

SRANDMEMBER¶

SRANDMEMBER key

返回集合中的一个随机元素。

该操作和 *SPOP* 相似，但 *SPOP* 将随机元素从集合中移除并返回，而 SRANDMEMBER 则仅仅返回随机元素，而不对集合进行任何改动。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

被选中的随机元素。 当 key 不存在或 key 是空集时，返回 nil 。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SRANDMEMBER joe's_movies
"Fast Five"

redis> SMEMBERS joe's_movies      # 集合中的元素不变
1) "hi, lady"
2) "Fast Five"
3) "2012"
```

SINTER¶

SINTER key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 key 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， N 为给定集合当中基数最小的集合， M 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

SINTERSTORE¶

SINTERSTORE destination key [key ...]

这个命令类似于 *SINTER* 命令，但它将结果保存到 destination 集合，而不是简单地返回结果集。

如果 destination 集合已经存在，则将其覆盖。

destination 可以是 key 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， N 为给定集合当中基数最小的集合， M 为给定集合的个数。

返回值:

结果集中的成员数量。

```
redis> SMEMBERS songs
1) "good bye joe"
2) "hello,peter"

redis> SMEMBERS my_songs
1) "good bye joe"
2) "falling"

redis> SINTERSTORE song_interset songs my_songs
(integer) 1

redis> SMEMBERS song_interset
1) "good bye joe"
```

SUNION¶

SUNION key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的并集。

不存在的 key 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 是所有给定集合的成员数量之和。

返回值:

并集成员的列表。

```
redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"
```

redis> SUNION songs my_songs
1) "Billie Jean"
2) "Believe Me"

SUNIONSTORE¶

SUNIONSTORE destination key [key ...]

这个命令类似于 *SUNION* 命令，但它将结果保存到 destination 集合，而不是简单地返回结果集。

如果 destination 已经存在，则将其覆盖。

destination 可以是 key 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

redis> SMEMBERS NoSQL
1) "MongoDB"
2) "Redis"

redis> SMEMBERS SQL
1) "sqlite"
2) "MySQL"

redis> SUNIONSTORE db NoSQL SQL
(integer) 4

redis> SMEMBERS db
1) "MySQL"
2) "sqlite"
3) "MongoDB"
4) "Redis"

SDIFF¶

SDIFF key [key ...]

返回一个集合的全部成员，该集合是所有给定集合之间的差集。

不存在的 key 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 是所有给定集合的成员数量之和。

返回值:

交集成员的列表。

```
redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SDIFF peter's_movies joe's_movies
1) "bet man"
2) "start war"
```

SDIFFSTORE¶

SDIFFSTORE destination key [key ...]

这个命令的作用和 *SDIFF* 类似，但它将结果保存到 destination 集合，而不是简单地返回结果集。

如果 destination 集合已经存在，则将其覆盖。

destination 可以是 key 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， N 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SDIFFSTORE joe_diff_peter joe's_movies peter's_movies
(integer) 2

redis> SMEMBERS joe_diff_peter
1) "hi, lady"
2) "Fast Five"
```

有序集(Sorted Set)¶

常用命令：

zadd,zrange,zrem,zcard 等.

使用场景：

Sorted set 的使用场景与 set 类似,区别是 set 不是自动有序的,而 sorted set 可以通过用户额外提供一个优先级(score)的参数来为成员排序,并且是插入有序的,即自动排序.当你需要一个有序的并且不重复的集合列表,那么可以选择 sorted set 数据结构.

实现方式：

Sorted set 的内部使用 HashMap 和跳跃表(SkipList)来保证数据的存储和有序,HashMap 里放的是成员到 score 的映射,而跳跃表里存放的是所有的成员,排序依据是 HashMap 里存的 score,使用跳跃表的结构可以获得比较高的查找效率.

ZADD¶

ZADD key score member [[score member] [score member] ...]

将一个或多个 member 元素及其 score 值加入到有序集 key 当中。

如果某个 member 已经是有序集的成员，那么更新这个 member 的 score 值 ,并通过重新插入这个 member 元素 ,来保证该 member 在正确的位置上。

score 值可以是整数值或双精度浮点数。

如果 key 不存在，则创建一个空的有序集并执行 ZADD 操作。

当 key 存在但不是有序集类型时，返回一个错误。

对有序集的更多介绍请参见 sorted set 。

注解

在 Redis 2.4 版本以前， ZADD 每次只能添加一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， N 是有序集的基数， M 为成功添加的新成员的数量。

返回值:

被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员。

```
# 添加单个元素
```

```
redis> ZADD page_rank 10 google.com
(integer) 1


# 添加多个元素

redis> ZADD page_rank 9 baidu.com 8 bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"


# 添加已存在元素，且 score 值不变

redis> ZADD page_rank 10 google.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # 没有改变
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"
```

添加已存在元素，但是改变 score 值

redis> ZADD page_rank 6 bing.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES # bing.com 元素的 score 值被改变
1) "bing.com"
2) "6"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

ZREM¶

ZREM key member [member ...]

移除有序集 key 中的一个或多个成员，不存在的成员将被忽略。

当 key 存在但不是有序集类型时，返回一个错误。

注解

在 Redis 2.4 版本以前， ZREM 每次只能删除一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， N 为有序集的基数， M 为被成功移除的成员的数量。

返回值:

被成功移除的成员的数量，不包括被忽略的成员。

测试数据

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"

6) "10"

移除单个元素

redis> ZREM page_rank google.com
(integer) 1

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"

移除多个元素

redis> ZREM page_rank baidu.com bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
(empty list or set)

移除不存在元素

redis> ZREM page_rank non-exists-element
(integer) 0

ZCARD¶

ZCARD key

返回有序集 key 的基数。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

当 key 存在且是有序集类型时，返回有序集的基数。

当 key 不存在时，返回 0 。

```
redis > ZADD salary 2000 tom      # 添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 1

redis > ZADD salary 5000 jack     # 再添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 2

redis > EXISTS non_exists_key     # 对不存在的 key 进行 ZCARD 操作
(integer) 0

redis > ZCARD non_exists_key
(integer) 0
```

ZCOUNT¶

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。

关于参数 min 和 max 的详细使用方法，请参考 *ZRANGEBYSCORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N)+M)， N 为有序集的基数， M 为值在 min 和 max 之间的元素的数量。

返回值:

score 值在 min 和 max 之间的成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES      # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZCOUNT salary 2000 5000            # 计算薪水在 2000-5000 之间的人数
(integer) 3

redis> ZCOUNT salary 3000 5000            # 计算薪水在 3000-5000 之间的人数
(integer) 2
```

ZSCORE¶

ZSCORE key member

返回有序集 key 中，成员 member 的 score 值。

如果 member 元素不是有序集 key 的成员，或 key 不存在，返回 nil 。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

member 成员的 score 值，以字符串形式表示。

```
redis> ZRANGE salary 0 -1 WITHSCORES      # 测试数据
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"
```

redis> ZSCORE salary peter　　# 注意返回值是字符串
"3500"

ZINCRBY¶

ZINCRBY key increment member

为有序集 key 的成员 member 的 score 值加上增量 increment 。

可以通过传递一个负数值 increment ，让 score 减去相应的值，比如 ZINCRBY key -5 member ，就是让 member 的 score 值减去 5 。

当 key 不存在，或 member 不是 key 的成员时， ZINCRBY key increment member 等同于 ZADD key increment member 。

当 key 不是有序集类型时，返回一个错误。

score 值可以是整数值或双精度浮点数。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N))

返回值:

member 成员的新 score 值，以字符串形式表示。

redis> ZSCORE salary tom
"2000"

redis> ZINCRBY salary 2000 tom　　# tom 加薪啦！
"4000"

ZRANGE¶

ZRANGE 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 值递增(从小到大)来排序。

具有相同 score 值的成员按字典序(lexicographical order)来排列。

如果你需要成员按 score 值递减(从大到小)来排列，请使用 ZREVRANGE 命令。

下标参数 start 和 stop 都以 0 为底，也就是说，以 0 表示有序集第一个成员，以 1 表示有序集第二个成员，以此类推。

你也可以使用负数下标，以 -1 表示最后一个成员， -2 表示倒数第二个成员，以此类推。
超出范围的下标并不会引起错误。
比如说，当 start 的值比有序集的最大下标还要大，或是 start > stop 时，ZRANGE 命令只是简单地返回一个空列表。
另一方面，假如 stop 参数的值比有序集的最大下标还要大，那么 Redis 将 stop 当作最大下标来处理。
可以通过使用 WITHSCORES 选项，来让成员和它的 score 值一并返回，返回列表以 value1,score1, ..., valueN,scoreN 的格式表示。
客户端库可能会返回一些更复杂的数据类型，比如数组、元组等。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， N 为有序集的基数，而 M 为结果集的基数。

返回值:

指定区间内，带有 score 值(可选)的有序集成员的列表。

```
redis > ZRANGE salary 0 -1 WITHSCORES                    # 显示整个有序集成员
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 1 2 WITHSCORES                     # 显示有序集下标区间 1 至 2 的成员
1) "tom"
2) "5000"
3) "boss"
4) "10086"
```

redis > ZRANGE salary 0 200000 WITHSCORES # 测试 end 下标超出最大下标时的情况
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 200000 3000000 WITHSCORES # 测试当给定区间不存在于有序集时的情况
(empty list or set)

ZREVRANGE¶

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(reverse lexicographical order)排列。

除了成员按 score 值递减的次序排列这一点外， ZREVRANGE 命令的其他方面和 ZRANGE 命令一样。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， N 为有序集的基数，而 M 为结果集的基数。

返回值:

指定区间内，带有 score 值(可选)的有序集成员的列表。

redis> ZRANGE salary 0 -1 WITHSCORES # 递增排列
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZREVRANGE salary 0 -1 WITHSCORES # 递减排列
1) "jack"
2) "5000"
3) "tom"
4) "4000"
5) "peter"
6) "3500"

ZRANGEBYSCORE¶

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

返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max)的成员。有序集成员按 score 值递增(从小到大)次序排列。

具有相同 score 值的成员按字典序(lexicographical order)来排列(该属性是有序集提供的，不需要额外的计算)。

可选的 LIMIT 参数指定返回结果的数量及区间(就像 SQL 中的 SELECT LIMIT offset, count)，注意当 offset 很大时，定位 offset 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间。

可选的 WITHSCORES 参数决定结果集是单单返回有序集的成员，还是将有序集成员及其 score 值一起返回。

该选项自 Redis 2.0 版本起可用。

区间及无限

min 和 max 可以是 -inf 和 +inf ，这样一来，你就可以在不知道有序集的最低和最高 score 值的情况下，使用 ZRANGEBYSCORE 这类命令。

默认情况下，区间的取值使用 闭区间 (小于等于或大于等于)，你也可以通过给参数前增加 (符号来使用可选的 开区间 (小于或大于)。

举个例子：

ZRANGEBYSCORE zset (1 5

返回所有符合条件 1 < score <= 5 的成员，而

ZRANGEBYSCORE zset (5 (10

则返回所有符合条件 5 < score < 10 的成员。

可用版本：

>= 1.0.5

时间复杂度:

O(log(N)+M)， N 为有序集的基数， M 为被结果集的基数。

返回值:

指定区间内，带有 score 值(可选)的有序集成员的列表。

```
redis> ZADD salary 2500 jack                    # 测试数据
(integer) 0
redis> ZADD salary 5000 tom
(integer) 0
redis> ZADD salary 12000 peter
(integer) 0

redis> ZRANGEBYSCORE salary -inf +inf           # 显示整个有序集
1) "jack"
2) "tom"
3) "peter"

redis> ZRANGEBYSCORE salary -inf +inf WITHSCORES    # 显示整个有序集及成员的 score 值
1) "jack"
2) "2500"
3) "tom"
4) "5000"
5) "peter"
6) "12000"

redis> ZRANGEBYSCORE salary -inf 5000 WITHSCORES    # 显示工资 <=5000 的所有成员
1) "jack"
2) "2500"
3) "tom"
4) "5000"
```

redis> ZRANGEBYSCORE salary (5000 400000 # 显示工资大于 5000 小于 400000 的成员
1) "peter"

ZREVRANGEBYSCORE¶

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

返回有序集 key 中， score 值介于 max 和 min 之间(默认包括等于 max 或 min)的所有的成员。有序集成员按 score 值递减(从大到小)的次序排列。具有相同 score 值的成员按字典序的逆序(reverse lexicographical order)排列。

除了成员按 score 值递减的次序排列这一点外， ZREVRANGEBYSCORE 命令的其他方面和 ZRANGEBYSCORE 命令一样。

可用版本：

>= 2.2.0

时间复杂度:

O(log(N)+M)， N 为有序集的基数， M 为结果集的基数。

返回值:

指定区间内，带有 score 值(可选)的有序集成员的列表。

```
redis > ZADD salary 10086 jack
(integer) 1
redis > ZADD salary 5000 tom
(integer) 1
redis > ZADD salary 7500 peter
(integer) 1
redis > ZADD salary 3500 joe
(integer) 1

redis > ZREVRANGEBYSCORE salary +inf -inf     # 逆序排列所有成员
1) "jack"
2) "peter"
3) "tom"
4) "joe"
```

redis > ZREVRANGEBYSCORE salary 10000 2000 # 逆序排列薪水介于 10000 和 2000 之间的成员

1) "peter"

2) "tom"

3) "joe"

ZRANK¶

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 。

使用 ZREVRANK 命令可以获得成员按 score 值递减(从大到小)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 member 是有序集 key 的成员，返回 member 的排名。

如果 member 不是有序集 key 的成员，返回 nil 。

redis> ZRANGE salary 0 -1 WITHSCORES # 显示所有成员及其 score 值

1) "peter"

2) "3500"

3) "tom"

4) "4000"

5) "jack"

6) "5000"

redis> ZRANK salary tom # 显示 tom 的薪水排名，第二

(integer) 1

ZREVRANK¶

ZREVRANK key member

返回有序集 key 中成员 member 的排名。其中有序集成员按 score 值递减(从大到小)排序。

排名以 0 为底，也就是说， score 值最大的成员排名为 0 。

使用 ZRANK 命令可以获得成员按 score 值递增(从小到大)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 member 是有序集 key 的成员，返回 member 的排名。

如果 member 不是有序集 key 的成员，返回 nil 。

```
redis 127.0.0.1:6379> ZRANGE salary 0 -1 WITHSCORES        # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZREVRANK salary peter        # peter 的工资排第二
(integer) 1

redis> ZREVRANK salary tom          # tom 的工资最高
(integer) 0
```

ZREMRANGEBYRANK¶

ZREMRANGEBYRANK key start stop

移除有序集 key 中，指定排名(rank)区间内的所有成员。

区间分别以下标参数 start 和 stop 指出，包含 start 和 stop 在内。

下标参数 start 和 stop 都以 0 为底，也就是说，以 0 表示有序集第一个成员，以 1 表示有序集第二个成员，以此类推。

你也可以使用负数下标，以 -1 表示最后一个成员， -2 表示倒数第二个成员，以此类推。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N)+M)， N 为有序集的基数，而 M 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZADD salary 2000 jack
(integer) 1
redis> ZADD salary 5000 tom
(integer) 1
redis> ZADD salary 3500 peter
(integer) 1

redis> ZREMRANGEBYRANK salary 0 1          # 移除下标 0 至 1 区间内的成员
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES       # 有序集只剩下一个成员
1) "tom"
2) "5000"
```

ZREMRANGEBYSCORE¶

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。

自版本 2.1.6 开始， score 值等于 min 或 max 的成员也可以不包括在内，详情请参见 ZRANGEBYSCORE 命令。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， N 为有序集的基数，而 M 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES               # 显示有序集内所有成员及其 score 值
```

1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZREMRANGEBYSCORE salary 1500 3500 # 移除所有薪水在 1500 到 3500 内的员工
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES # 剩下的有序集成员
1) "jack"
2) "5000"

ZINTERSTORE¶

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

计算给定的一个或多个有序集的交集，其中给定 key 的数量必须以 numkeys 参数指定，并将该交集(结果集)储存到 destination 。

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.

关于 WEIGHTS 和 AGGREGATE 选项的描述，参见 ZUNIONSTORE 命令。

可用版本：

2.0.0

时间复杂度:

O(N*K)+O(M*log(M))， N 为给定 key 中基数最小的有序集， K 为给定有序集的数量， M 为结果集的基数。

返回值:

保存到 destination 的结果集的基数。

redis > ZADD mid_test 70 "Li Lei"
(integer) 1
redis > ZADD mid_test 70 "Han Meimei"
(integer) 1

```
redis > ZADD mid_test 99.5 "Tom"
(integer) 1

redis > ZADD fin_test 88 "Li Lei"
(integer) 1
redis > ZADD fin_test 75 "Han Meimei"
(integer) 1
redis > ZADD fin_test 99.5 "Tom"
(integer) 1

redis > ZINTERSTORE sum_point 2 mid_test fin_test
(integer) 3

redis > ZRANGE sum_point 0 -1 WITHSCORES        # 显式有序集内所有成员及其 score 值
1) "Han Meimei"
2) "145"
3) "Li Lei"
4) "158"
5) "Tom"
6) "199"
```

ZUNIONSTORE¶

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

计算给定的一个或多个有序集的并集，其中给定 key 的数量必须以 numkeys 参数指定，并将该并集(结果集)储存到 destination 。

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* 。

WEIGHTS

使用 WEIGHTS 选项，你可以为 *每个* 给定有序集 *分别* 指定一个乘法因子(multiplication factor)，每个给定有序集的所有成员的 score 值在传递给聚合函数(aggregation function)之前都要先乘以该有序集的因子。

如果没有指定 WEIGHTS 选项，乘法因子默认设置为 1 。

AGGREGATE

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；使用参数 MIN ，可以将所有集合中某个成员的 *最小* score 值作为结果集中该成员的 score 值；而参数 MAX 则是将所有集合中某个成员的 *最大* score 值作为结果集中该成员的 score 值。

可用版本：

>= 2.0.0

时间复杂度:

O(N)+O(M log(M))， N 为给定有序集基数的总和， M 为结果集的基数。

返回值:

保存到 destination 的结果集的基数。

```
redis> ZRANGE programmer 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"

redis> ZRANGE manager 0 -1 WITHSCORES
1) "herry"
2) "2000"
3) "mary"
4) "3500"
5) "bob"
6) "4000"

redis> ZUNIONSTORE salary 2 programmer manager WEIGHTS 1 3    # 公司决定加薪。。。除了程序员。。。
(integer) 6
```

```
redis> ZRANGE salary 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"
7) "herry"
8) "6000"
9) "mary"
10) "10500"
11) "bob"
12) "12000"
```

发布/订阅(Pub/Sub)¶

Redis的发布/订阅(Publish/Subscribe)功能类似于传统的消息路由功能,发布者发布消息,订阅者接收消息,沟通发布者和订阅者之间的桥梁是订阅的 Channel 或者 Pattern.

订阅者和发布者之间的关系是松耦合的,发布者不指定哪个订阅者才能接收消息,订阅者不只接收特定发布者的消息.

PUBLISH¶

PUBLISH channel message

将信息 message 发送到指定的频道 channel 。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 N 是频道 channel 的订阅者数量，而 M 则是使用模式订阅(subscribed patterns)的客户端的数量。

返回值：

接收到信息 message 的订阅者数量。

```
# 对没有订阅者的频道发送信息

redis> publish bad_channel "can any body hear me?"
(integer) 0

# 向有一个订阅者的频道发送信息

redis> publish msg "good morning"
(integer) 1

# 向有多个订阅者的频道发送信息

redis> publish chat_room "hello~ everyone"
(integer) 3
```

SUBSCRIBE¶

SUBSCRIBE channel [channel ...]

订阅给定频道的信息。

可用版本：

>= 2.0.0

时间复杂度：

O(N)，其中 N 是订阅的频道的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 msg 和 chat_room 两个频道

# 1 - 6 行是执行 subscribe 之后的反馈信息
# 第 7 - 9 行才是接收到的第一条信息
# 第 10 - 12 行是第二条

redis> subscribe msg chat_room
Reading messages... (press Ctrl-C to quit)
1) "subscribe"          # 返回值的类型：显示订阅成功
2) "msg"                # 订阅的频道名字
3) (integer) 1          # 目前已订阅的频道数量

1) "subscribe"
2) "chat_room"
3) (integer) 2

1) "message"            # 返回值的类型：信息
2) "msg"                # 来源(从那个频道发送过来)
3) "hello moto"         # 信息内容

1) "message"
2) "chat_room"
3) "testing...haha"
```

PSUBSCRIBE¶

PSUBSCRIBE pattern [pattern ...]

订阅符合给定模式的频道。

每个模式以 * 作为匹配符，比如 it* 匹配所有以 it 开头的频道(it.news 、 it.blog 、 it.tweets 等等)， news.* 匹配所有以 news. 开头的频道(news.it 、 news.global.today 等等)，诸如此类。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， N 是订阅的模式的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 news.* 和 tweet.* 两个模式

# 第 1 - 6 行是执行 psubscribe 之后的反馈信息
# 第 7 - 10 才是接收到的第一条信息
# 第 11 - 14 是第二条
# 以此类推。。。

redis> psubscribe news.* tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"                    # 返回值的类型：显示订阅成功
2) "news.*"                        # 订阅的模式
3) (integer) 1                     # 目前已订阅的模式的数量

1) "psubscribe"
2) "tweet.*"
3) (integer) 2

1) "pmessage"                      # 返回值的类型：信息
2) "news.*"                        # 信息匹配的模式
3) "news.it"                       # 信息本身的目标频道
4) "Google buy Motorola"           # 信息的内容
```

1) "pmessage"
2) "tweet.*"
3) "tweet.huangz"
4) "hello"

1) "pmessage"
2) "tweet.*"
3) "tweet.joe"
4) "@huangz morning"

1) "pmessage"
2) "news.*"
3) "news.life"
4) "An apple a day, keep doctors away"

UNSUBSCRIBE¶

警告

此命令在新版 Redis 中似乎已经被废弃？

PUNSUBSCRIBE¶

警告

此命令在新版 Redis 中似乎已经被废弃？

事务(Transaction)¶

Redis 目前对事务支持还比较简单,也即支持一些简单的组合型的命令,只能保证一个 client 发起的事务中的命令可以连续的执行,而中间不会插入其他 client 的命令. 由于 Redis 是单线程来处理所有 client 的请求的所以做到这点是很容易的.

事务的执行过程中,如果 redis 意外的挂了,这时候事务可能只被执行了一半,可以用 redis-check-aof 工具进行修复.

WATCH¶

WATCH key [key ...]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

可用版本：

>= 2.2.0

时间复杂度：

O(1)。

返回值：

总是返回 OK 。

redis> WATCH lock lock_times
OK

UNWATCH¶

UNWATCH

取消 WATCH 命令对所有 key 的监视。

如果在执行 WATCH 命令之后， EXEC 命令或 DISCARD 命令先被执行了的话，那么就不需要再执行 UNWATCH 了。

因为 EXEC 命令会执行事务，因此 WATCH 命令的效果已经产生了；而 DISCARD 命令在取消事务的同时也会取消所有对 key 的监视，因此这两个命令执行之后，就没有必要执行 UNWATCH 了。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

总是 OK 。

```
redis> WATCH lock lock_times
OK

redis> UNWATCH
OK
```

MULTI¶

MULTI

标记一个事务块的开始。

事务块内的多条命令会按照先后顺序被放进一个队列当中，最后由 EXEC 命令在一个原子时间内执行。

可用版本：

>= 1.2.0

时间复杂度：

O(1)。

返回值：

总是返回 OK 。

```
redis> MULTI             # 标记事务开始
OK

redis> INCR user_id   # 多条命令按顺序入队
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED
```

redis> EXEC # 执行
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG

EXEC¶

EXEC

执行所有事务块内的命令。

假如某个(或某些) key 正处于 WATCH 命令的监视之下，且事务块中有和这个(或这些) key 相关的命令，那么 EXEC 命令只在这个(或这些) key 没有被其他命令所改动的情况下执行并生效，否则该事务被打断(abort)。

可用版本：

>= 1.2.0

时间复杂度：

事务块内所有命令的时间复杂度的总和。

返回值：

事务块内所有命令的返回值，按命令执行的先后顺序排列。

当操作被打断时，返回空值 nil 。

事务被成功执行

redis> MULTI
OK

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

```
redis> PING
QUEUED

redis> EXEC
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG


# 监视 key ，且事务成功执行

redis> WATCH lock lock_times
OK

redis> MULTI
OK

redis> SET lock "huangz"
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC
1) OK
2) (integer) 1


# 监视 key ，且事务被打断

redis> WATCH lock lock_times
OK
```

```
redis> MULTI
OK

redis> SET lock "joe"          # 就在这时，另一个客户端修改了 lock_times 的值
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC                    # 因为 lock_times 被修改， joe 的事务执行失败
(nil)
```

DISCARD¶

DISCARD

取消事务，放弃执行事务块内的所有命令。

如果正在使用 WATCH 命令监视某个(或某些) key，那么取消所有监视，等同于执行命令 UNWATCH 。

可用版本：

>= 2.0.0

时间复杂度：

O(1)。

返回值：

总是返回 OK 。

```
redis> MULTI
OK

redis> PING
QUEUED

redis> SET greeting "hello"
QUEUED
```

```
redis> DISCARD
OK
```

连接(Connection)¶

AUTH¶

AUTH password

通过设置配置文件中 requirepass 项的值(使用命令 CONFIG SET requirepass password)，可以使用密码来保护 Redis 服务器。

如果开启了密码保护的话 ,在每次连接 Redis 服务器之后 ,就要使用 AUTH 命令解锁，解锁之后才能使用其他 Redis 命令。

如果 AUTH 命令给定的密码 password 和配置文件中的密码相符的话，服务器会返回 OK 并开始接受命令输入。

反之，如果密码不匹配的话，服务器将返回一个错误，并要求客户端需重新输入密码。

警告

因为 Redis 高性能的特点，在很短时间内尝试猜测非常多个密码是有可能的，因此请确保使用的密码足够复杂和足够长，以免遭受密码猜测攻击。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

密码匹配时返回 OK ，否则返回一个错误。

```
# 设置密码

redis> CONFIG SET requirepass secret_password     # 将密码设置为 secret_password
OK

redis> QUIT                                       # 退出再连接，让新密码对客户端生效

[huangz@mypad]$ redis
```

```
redis> PING                                    # 未验证密码，操作被拒绝
(error) ERR operation not permitted

redis> AUTH wrong_password_testing             # 尝试输入错误的密码
(error) ERR invalid password

redis> AUTH secret_password                    # 输入正确的密码
OK

redis> PING                                    # 密码验证成功，可以正常操作命令了
PONG


# 清空密码

redis> CONFIG SET requirepass ""     # 通过将密码设为空字符来清空密码
OK

redis> QUIT

$ redis                                # 重新进入客户端

redis> PING                            # 执行命令不再需要密码，清空密码操作成功
PONG
```

PING¶

PING

客户端向服务器发送一个 PING ，然后服务器返回客户端一个 PONG 。

通常用于测试与服务器的连接是否仍然生效，或者用于测量延迟值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

PONG

```
redis> PING
PONG
```

SELECT¶

SELECT index

切换到指定的数据库，数据库索引号用数字值指定，以 0 作为起始索引值。

新的链接总是使用 0 号数据库。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

OK

```
redis> SET db_number 0            # 默认使用 0 号数据库
OK

redis> SELECT 1                   # 使用 1 号数据库
OK

redis[1]> GET db_number           # 已经切换到 1 号数据库，注意 Redis 现在的命令提示符多了个 [1]
(nil)

redis[1]> SET db_number 1
OK

redis[1]> GET db_number
"1"
```

```
redis[1]> SELECT 3                    # 再切换到 3 号数据库
OK

redis[3]>                             # 提示符从 [1] 改变成了 [3]
```

ECHO¶

ECHO message

打印一个特定的信息 message ，测试时使用。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

message 自身。

```
redis> ECHO "Hello Moto"
"Hello Moto"

redis> ECHO "Goodbye Moto"
"Goodbye Moto"
```

QUIT¶

QUIT

请求服务器关闭与当前客户端的连接。

一旦所有等待中的回复(如果有的话)顺利写入到客户端，连接就会被关闭。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 OK (但是不会被打印显示，因为当时 Redis-cli 已经退出)。

```
$ redis

redis> QUIT
```

$

服务器(Server)¶

TIME¶

TIME

返回当前服务器时间。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

一个包含两个字符串的列表： 第一个字符串是当前时间(以 UNIX 时间戳格式表示)，而第二个字符串是当前这一秒钟已经逝去的微秒数。

```
redis> TIME
1) "1332395997"
2) "952581"
redis> TIME
1) "1332395997"
2) "953148"
```

DBSIZE¶

DBSIZE

返回当前数据库的 key 的数量。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当前数据库的 key 的数量。

```
redis> DBSIZE
(integer) 5

redis> SET new_key "hello_moto"        # 增加一个 key 试试
OK
```

redis> DBSIZE
(integer) 6

BGREWRITEAOF¶

BGREWRITEAOF

异步(Asynchronously)重写 AOF 文件以反应当前数据库的状态。

即使 BGREWRITEAOF 命令执行失败，旧 AOF 文件中的数据也不会因此丢失或改变。

请移步 持久化文档 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为要追加到 AOF 文件中的数据数量。

返回值：

反馈信息。

redis> BGREWRITEAOF
Background append only file rewriting started

BGSAVE¶

在后台异步保存当前数据库的数据到磁盘。

BGSAVE 命令执行之后立即返回 OK ，然后 Redis fork 出一个新子进程，原来的 Redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出。

客户端可以通过 LASTSAVE 命令查看相关信息，判断 BGSAVE 命令是否执行成功。

请移步 持久化文档 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为要保存到数据库中的 key 的数量。

返回值：

反馈信息。

redis> BGSAVE

Background saving started

SAVE¶

SAVE

同步保存当前数据库的数据到磁盘。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， N 为要保存到数据库中的 key 的数量。

返回值：

总是返回 OK 。

redis> SAVE
OK

LASTSAVE¶

LASTSAVE

返回最近一次 Redis 成功执行保存操作的时间点(SAVE 、 BGSAVE 等)，以 UNIX 时间戳格式表示。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

一个 UNIX 时间戳。

redis> LASTSAVE
(integer) 1324043588

SLAVEOF¶

SLAVEOF host port

SLAVEOF 命令用于在 Redis 运行时动态地修改复制(replication)功能的行为。通过执行 SLAVEOF host port 命令 ,可以将当前服务器转变为指定服务器的从属服务器(slave server)。

如果当前服务器已经是某个主服务器(master server)的从属服务器，那么执行 SLAVEOF host port 将使当前服务器停止对旧主服务器的同步，丢弃旧数据集，转而开始对新主服务器进行同步。

另外，对一个从属服务器执行命令 SLAVEOF NO ONE 将使得这个从属服务器关闭复制功能，并从从属服务器转变回主服务器，原来同步所得的数据集*不会*被丢弃。

利用“ SLAVEOF NO ONE 不会丢弃同步所得数据集”这个特性，可以在主服务器失败的时候，将从属服务器用作新的主服务器，从而实现无间断运行。

可用版本：

>= 1.0.0

时间复杂度：

SLAVEOF host port ，O(N)， N 为要同步的数据数量。

SLAVEOF NO ONE ， O(1) 。

返回值：

总是返回 OK 。

```
redis> SLAVEOF 127.0.0.1 6379
OK

redis> SLAVEOF NO ONE
OK
```

FLUSHALL¶

FLUSHALL

清空整个 Redis 服务器的数据(删除*所有数据库的所有 key*)。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

尚未明确

返回值：

总是返回 OK 。

```
redis> DBSIZE                # 0 号数据库的 key 数量
(integer) 9
```

```
redis> SELECT 1                # 切换到 1 号数据库
OK

redis[1]> DBSIZE               # 1 号数据库的 key 数量
(integer) 6

redis[1]> flushall             # 清空所有数据库的所有 key
OK

redis[1]> DBSIZE               # 不但 1 号数据库被清空了
(integer) 0

redis[1]> SELECT 0             # 0 号数据库(以及其他所有数据库)也一样
OK

redis> DBSIZE
(integer) 0
```

FLUSHDB¶

FLUSHDB

清空*当前*数据库中的所有 key。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 OK 。

```
redis> DBSIZE      # 清空前的 key 数量
(integer) 4

redis> FLUSHDB
OK
```

redis> DBSIZE　　# 清空后的 key 数量
(integer) 0

SHUTDOWN¶

SHUTDOWN

SHUTDOWN 命令执行以下操作：

- 停止所有客户端
- 如果有至少一个保存点在等待，执行 *SAVE* 命令
- 如果 AOF 选项被打开，更新 AOF 文件
- 关闭 redis 服务器(server)

如果持久化被打开的话，SHUTDOWN 命令会保证服务器正常关闭而不丢失任何数据。

另一方面，假如只是单纯地执行 *SAVE* 命令，然后再执行 *QUIT* 命令，则没有这一保证 —— 因为在执行 *SAVE* 之后、执行 *QUIT* 之前的这段时间中间，其他客户端可能正在和服务器进行通讯，这时如果执行 *QUIT* 就会造成数据丢失。

SAVE 和 NOSAVE 修饰符

通过使用可选的修饰符，可以修改 SHUTDOWN 命令的表现。比如说：

- 执行 SHUTDOWN SAVE 会强制让数据库执行保存操作，即使没有设定(configure)保存点
- 执行 SHUTDOWN NOSAVE 会阻止数据库执行保存操作，即使已经设定有一个或多个保存点(你可以将这一用法看作是强制停止服务器的一个假想的 ABORT 命令)

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

执行失败时返回错误。

执行成功时不返回任何信息，服务器和客户端的连接断开，客户端自动退出。

redis> PING
PONG

redis> SHUTDOWN

$

$ redis
Could not connect to Redis at: Connection refused
not connected>

SLOWLOG¶

SLOWLOG subcommand [argument]

什么是 SLOWLOG

Slow log 是 Redis 用来记录查询执行时间的日志系统。

查询执行时间指的是不包括像客户端响应(talking)、发送回复等 IO 操作，而单单是执行一个查询命令所耗费的时间。

另外，slow log 保存在内存里面，读写速度非常快，因此你可以放心地使用它，不必担心因为开启 slow log 而损害 Redis 的速度。

设置 SLOWLOG

Slow log 的行为由两个配置参数(configuration parameter)指定，可以通过改写 redis.conf 文件或者用 CONFIG GET 和 CONFIG SET 命令对它们动态地进行修改。

第一个选项是 slowlog-log-slower-then ，它决定要对执行时间大于多少微秒(microsecond，1 秒 = 1,000,000 微秒)的查询进行记录。

比如执行以下命令将让 slow log 记录所有查询时间大于等于 100 微秒的查询：

CONFIG SET slowlog-log-slower-then 100 ，

而以下命令记录所有查询时间大于 1000 微秒的查询：

CONFIG SET slowlog-log-slower-then 1000 。

另一个选项是 slowlog-max-len ，它决定 slow log *最多*能保存多少条日志，slow log 本身是一个 LIFO 队列，当队列大小超过 slowlog-max-len 时，最旧的一条日志将被删除，而最新的一条日志加入到 slow log ，以此类推。

以下命令让 slow log 最多保存 1000 条日志：

CONFIG SET slowlog-max-len 1000 。

使用 CONFIG GET 命令可以查询两个选项的当前值：

redis> CONFIG GET slowlog-log-slower-than
1) "slowlog-log-slower-than"

2) "1000"

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"

查看 slow log

要查看 slow log ，可以使用 SLOWLOG GET 或者 SLOWLOG GET number 命令 ,前者打印所有 slow log ,最大长度取决于 slowlog-max-len 选项的值，而 SLOWLOG GET number 则只打印指定数量的日志。

最新的日志会最先被打印：

为测试需要，将 slowlog-log-slower-than 设成了 10 微秒

redis> SLOWLOG GET
1) 1) (integer) 12 # 唯一性(unique)的日志标识符
2) (integer) 1324097834 # 被记录命令的执行时间点，以 UNIX 时间戳格式表示
3) (integer) 16 # 查询执行时间，以微秒为单位
4) 1) "CONFIG" # 执行的命令，以数组的形式排列
2) "GET" # 这里完整的命令是 CONFIG GET slowlog-log-slower-than
3) "slowlog-log-slower-than"

2) 1) (integer) 11
2) (integer) 1324097825
3) (integer) 42
4) 1) "CONFIG"
2) "GET"
3) "*"

3) 1) (integer) 10
2) (integer) 1324097820
3) (integer) 11

4) 1) "CONFIG"
2) "GET"
3) "slowlog-log-slower-then"

...

日志的唯一 id 只有在 Redis 服务器重启的时候才会重置，这样可以避免对日志的重复处理(比如你可能会想在每次发现新的慢查询时发邮件通知你)。

查看当前日志的数量

使用命令 SLOWLOG LEN 可以查看当前日志的数量。

请注意这个值和 slower-max-len 的区别，它们一个是当前日志的数量，一个是允许记录的最大日志的数量。

redis> SLOWLOG LEN
(integer) 14

清空日志

使用命令 SLOWLOG RESET 可以清空 slow log 。

redis> SLOWLOG LEN
(integer) 14

redis> SLOWLOG RESET
OK

redis> SLOWLOG LEN
(integer) 0

可用版本：

>= 2.2.12

时间复杂度：

O(1)

返回值：

取决于不同命令，返回不同的值。

INFO¶

INFO

返回关于 Redis 服务器的各种信息和统计值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

具体请参见下面的测试代码。

```
redis> INFO
redis_version:2.4.4                    # Redis 的版本
redis_git_sha1:00000000
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
process_id:903                         # 当前 Redis 服务器进程 id
uptime_in_seconds:24612                # 运行时间(以秒计算)
uptime_in_days:0                       # 运行时间(以日计算)
lru_clock:283730
used_cpu_sys:3.38
used_cpu_user:2.15
used_cpu_sys_children:0.11
used_cpu_user_children:0.00
connected_clients:1                    # 连接的客户端数量
connected_slaves:0                     # 从属服务器的数量
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0
used_memory:557304                     # Redis 分配的内存总量
used_memory_human:544.24K
used_memory_rss:17879040               # Redis 分配的内存总量(包括内存碎
片)
used_memory_peak:565904
used_memory_peak_human:552.64K
mem_fragmentation_ratio:32.08          # 内存碎片比率
mem_allocator:jemalloc-2.2.5           # 目前使用的内存分配库
loading:0
```

aof_enabled:0
changes_since_last_save:2 # 上次保存数据库之后，执行命令的次数
bgsave_in_progress:0 # 后台进行中的 save 操作的数量
last_save_time:1324042687 # 最后一次成功保存的时间点，以 UNIX 时间戳格式显示
bgrewriteaof_in_progress:0 # 后台进行中的 aof 文件修改操作的数量
total_connections_received:16 # 运行以来连接过的客户端的总数量
total_commands_processed:87 # 运行以来执行过的命令的总数量
expired_keys:0 # 运行以来过期的 key 的数量
evicted_keys:0
keyspace_hits:14 # 命中 key 的次数
keyspace_misses:14 # 不命中 key 的次数
pubsub_channels:0 # 当前使用中的频道数量
pubsub_patterns:0 # 当前使用的模式的数量
latest_fork_usec:314
vm_enabled:0 # 是否开启了 vm
role:master
db0:keys=6,expires=0 # 各个数据库的 key 的数量，以及带有生存期的 key 的数量
db1:keys=6,expires=0
db2:keys=1,expires=0

CONFIG GET¶

CONFIG GET parameter

CONFIG GET 命令用于取得运行中的 Redis 服务器的配置参数(configuration parameters)，在 Redis 2.4 版本中， 有部分参数没有办法用 CONFIG GET 访问，但是在最新的 Redis 2.6 版本中，所有配置参数都已经可以用 CONFIG GET 访问了。

CONFIG GET 接受单个参数 parameter 作为搜索关键字，查找所有匹配的配置参数，其中参数和值以“键-值对”(key-value pairs)的方式排列。

比如执行 CONFIG GET s* 命令，服务器就会返回所有以 s 开头的配置参数及参数的值：

redis> CONFIG GET s*
1) "save" # 参数名：save

2) "900 1 300 10 60 10000" # save 参数的值
3) "slave-serve-stale-data" # 参数名： slave-serve-stale-data
4) "yes" # slave-serve-stale-data 参数的值
5) "set-max-intset-entries" # ...
6) "512"
7) "slowlog-log-slower-than"
8) "1000"
9) "slowlog-max-len"
10) "1000"

如果你只是寻找特定的某个参数的话，你当然也可以直接指定参数的名字：

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

使用命令 CONFIG GET * ，可以列出 CONFIG GET 命令支持的所有参数：

```
redis> CONFIG GET *
1) "dir"
2) "/var/lib/redis"
3) "dbfilename"
4) "dump.rdb"
5) "requirepass"
6) (nil)
7) "masterauth"
8) (nil)
9) "maxmemory"
10) "0"
11) "maxmemory-policy"
12) "volatile-lru"
13) "maxmemory-samples"
14) "3"
15) "timeout"
16) "0"
17) "appendonly"
18) "no"
```

...
49) "loglevel"
50) "verbose"

所有被 CONFIG SET 所支持的配置参数都可以在配置文件 redis.conf 中找到，不过 CONFIG GET 和 CONFIG SET 使用的格式和 redis.conf 文件所使用的格式有以下两点不同：

- 10kb 、 2gb 这些在配置文件中所使用的储存单位缩写，不可以用在 CONFIG 命令中， CONFIG SET 的值只能通过数字值显式地设定。

  像 CONFIG SET xxx 1k 这样的命令是错误的，正确的格式是 CONFIG SET xxx 1000 。
- save 选项在 redis.conf 中是用多行文字储存的，但在 CONFIG GET 命令中，它只打印一行文字。

  以下是 save 选项在 redis.conf 文件中的表示：

  ```
  save 900 1
  save 300 10
  save 60 10000
  ```

  但是 CONFIG GET 命令的输出只有一行：

  ```
  redis> CONFIG GET save
  1) "save"
  2) "900 1 300 10 60 10000"
  ```

  上面 save 参数的三个值表示：在 900 秒内最少有 1 个 key 被改动，或者 300 秒内最少有 10 个 key 被改动，又或者 60 秒内最少有 1000 个 key 被改动，以上三个条件随便满足一个，就触发一次保存操作。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

给定配置参数的值。

CONFIG SET¶

CONFIG SET parameter value

CONFIG SET 命令可以动态地调整 Redis 服务器的配置(configuration)而无须重启。

你可以使用它修改配置参数，或者改变 Redis 的持久化(Persistence)方式。

CONFIG SET 可以修改的配置参数可以使用命令 CONFIG GET * 来列出，所有被 CONFIG SET 修改的配置参数都会立即生效。

关于 CONFIG SET 命令的更多消息，请参见命令 CONFIG GET 的说明。

关于如何使用 CONFIG SET 命令修改 Redis 持久化方式，请参见 Redis Persistence 。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

当设置成功时返回 OK ，否则返回一个错误。

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1024"

redis> CONFIG SET slowlog-max-len 10086
OK

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "10086"
```

CONFIG RESETSTAT¶

CONFIG RESETSTAT

重置 INFO 命令中的某些统计数据，包括：

- Keyspace hits (键空间命中次数)
- Keyspace misses (键空间不命中次数)
- Number of commands processed (执行命令的次数)
- Number of connections received (连接服务器的次数)
- Number of expired keys (过期 key 的数量)

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

总是返回 OK 。

```
# 重置前的部分数据

redis> INFO
# ...
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:5
# ...

# 重置

redis> CONFIG RESETSTAT
OK

# 重置后的部分数据

redis> INFO
# ...
expired_keys:0
evicted_keys:0
keyspace_hits:0
```

keyspace_misses:0
pubsub_channels:0
...

DEBUG OBJECT¶

DEBUG OBJECT key

DEBUG OBJECT 是一个调试命令，它不应被客户端所使用。

查看 *OBJECT* 命令获取更多信息。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 key 存在时，返回有关信息。

当 key 不存在时，返回一个错误。

```
redis> DEBUG OBJECT my_pc
Value at:0xb6838d20 refcount:1 encoding:raw serializedlength:9
lru:283790 lru_seconds_idle:150

redis> DEBUG OBJECT your_mac
(error) ERR no such key
```

DEBUG SEGFAULT¶

DEBUG SEGFAULT

执行一个不合法的内存访问从而让 Redis 崩溃，仅在开发时用于 BUG 模拟。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

无

```
redis> DEBUG SEGFAULT
Could not connect to Redis at: Connection refused
```

not connected>

MONITOR¶

MONITOR

实时打印出 Redis 服务器接收到的命令，调试用。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

总是返回 OK 。

redis> MONITOR

OK

1324109476.800290 "MONITOR" # 第一个值是 UNIX 时间戳，之后是执行的命令和参数

1324109479.632445 "PING"

1324109486.408230 "SET" "greeting" "hello moto"

1324109490.800364 "KEYS" "*"

1324109509.023495 "lrange" "my_book_list" "0" "-1"

SYNC¶

SYNC

用于复制功能(replication)的内部命令。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

不明确

redis> SYNC

"REDIS0002\xfe\x00\x00\auser_id\xc0\x03\x00\anumbers\xc2\xf3\xe0\x01\x00\x00\tdb_number\xc0\x00\x00\x04name\x06huangz\x00\anew_key\nhello_moto\x00\bgreeting\nhello moto\x00\x05my_pc\bthinkpad\x00\x04lock\xc0\x01\x00\nlock_times\x

c0\x04\xfe\x01\t\x04info\x19\x02\x04name\b\x00zhangyue\x03age\x02\x0022\xff\t\aooredis,\x03\x04name\a\x00ooredis\aversion\x03\x001.0\x06author\x06\x00huangz\xff\x00\tdb_number\xc0\x01\x00\x05greet\x0bhello
world\x02\nmy_friends\x02\x05marry\x04jack\x00\x04name\x05value\xfe\x02\x0c\x01s\x12\x12\x00\x00\x00\r\x00\x00\x00\x02\x00\x00\x01a\x03\xc0f'\xff\xff"
(1.90s)

脚本(Script)¶

EVAL¶

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

从 Redis 2.6.0 版本开始，通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值。

script 参数是一段 Lua 5.1 脚本程序，它会被运行在 Redis 服务器上下文中，这段脚本不必(也不应该)定义为一个 Lua 函数。

numkeys 参数用于指定键名参数的个数。

键名参数 key [key ...] 从 EVAL 的第三个参数开始算起，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)。

上面这几段长长的说明可以用一个简单的例子来概括：

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

其中 "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 是被求值的 Lua 脚本，数字 2 指定了键名参数的数量， key1 和 key2 是键名参数，分别使用 KEYS[1] 和 KEYS[2] 访问，而最后的 first 和 second 则是附加参数，可以通过 ARGV[1] 和 ARGV[2] 访问它们。

在 Lua 脚本中，可以使用两个不同函数来执行 Redis 命令，它们分别是：

- redis.call()
- redis.pcall()

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误，在后面的『错误处理』部分会讲到这一点。

redis.call() 和 redis.pcall() 两个函数的参数可以是任何格式良好(well formed)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。
在 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。
以下列出的是详细的转换规则：
从 Redis 转换到 Lua ：

- 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

从 Lua 转换到 Redis：

- Lua number -> Redis integer reply / 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 / 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

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"
```

脚本的原子性

Redis 使用同一个 Lua 解释器去执行所有命令。与此同时， Redis 也保证脚本会以原子性(atomic)的方式执行：当一个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这一语义和 *MULTI* / *EXEC* 很相似。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

在前面我们说过， 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，并返回一个错误，这个错误会清晰地说明这是一个由脚本产生的错误：

```
redis 127.0.0.1:6379> lpush foo a
(integer) 1
redis 127.0.0.1:6379> eval "return redis.call('get', 'foo')" 0
(error) ERR Error running script (call to
f_282297a0228f48cd3fc6a55de6316f31422f5d17): 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，用于表示错误。

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。

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*
- *SCRIPT EXISTS*
- *SCRIPT LOAD*
- *SCRIPT KILL*

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 客户端中执行 *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，但假如用户需要的话，还是可以用一个简单的技巧写出带随机行为的脚本。

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])
    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)
```

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，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])
    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 会在复制连接(replication link)和 AOF 文件中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。

库

Redis 内置的 Lua 解释器加载了以下 Lua 库：

- base
- table
- string
- math
- debug
- cjson
- cmsgpack

其中 cjson 库可以让 Lua 以非常快的速度处理 JSON 数据，除此之外，其他别的都是 Lua 的标准库。

每个 Redis 实例都保证会加载上面列举的库，从而确保每个 Redis 脚本的运行环境都是相同的。

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 ,但对所有发送一般命令的客户端都只返回 BUSY 错误。在这一状态下唯一可以被接受的只有 SCRIPT KILL 和 SHUTDOWN NOSAVE 命令
- 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定义过。如果所有脚本都已经定义过了，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。

可用版本：

>= 2.6.0

时间复杂度：

EVAL 和 EVALSHA 可以在 O(1) 复杂度内找到要被执行的脚本，其余的复杂度取决于执行的脚本本身。

SCRIPT FLUSH¶

SCRIPT FLUSH

清除所有 Lua 脚本缓存。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

复杂度：

O(N) ， N 为缓存中脚本的数量。

返回值：

总是返回 OK

```
redis> SCRIPT FLUSH
OK
```

SCRIPT LOAD¶

SCRIPT LOAD script

将脚本 script 添加到脚本缓存中，但并不执行这个脚本。

EVAL 命令也会将脚本添加到脚本缓存中，但是它会立即对输入的脚本进行求值。

如果给定的脚本已经在缓存里面了，那么不做动作。

在脚本被加入到缓存之后，通过 EVALSHA 命令，可以使用脚本的 SHA1 校验和来调用这个脚本。

脚本可以在缓存中保留无限长的时间，直到执行 *SCRIPT FLUSH* 为止。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N)，N 为脚本的长度(以字节为单位)。

返回值：

给定 script 的 SHA1 校验和

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"
```

```
redis> EVALSHA 232fd51614574cf0867b83d384a5e898cfd24e5a 0
"hello moto"
```

SCRIPT EXISTS¶

SCRIPT EXISTS script [script ...]

给定一个或多个脚本的 SHA1 校验和，返回一个包含 0 和 1 的列表，表示校验和所指定的脚本是否已经被保存在缓存当中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N)，N 为给定的 SHA1 校验和的数量。

返回值：

一个列表，包含 0 和 1 。

列表中的元素和给定的 SHA1 校验和保持对应关系，比如列表的第三个元素的值就表示第三个 SHA1 校验和所指定的脚本在缓存中的状态。

```
redis> SCRIPT LOAD "return 'hello moto'"       # 载入一个脚本
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 1

redis> SCRIPT FLUSH        # 清空缓存
OK

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 0
```

SCRIPT KILL¶

SCRIPT KILL

杀死当前正在运行的 Lua 脚本，当且仅当这个脚本没有执行过任何写操作时，这个命令才生效。

这个命令主要用于终止运行时间过长的脚本，比如一个因为 BUG 而发生无限 loop 的脚本，诸如此类。

SCRIPT KILL 执行之后，当前正在运行的脚本会被杀死，执行这个脚本的客户端会从 *EVAL* 命令的阻塞当中退出，并收到一个错误作为返回值。
另一方面，假如当前正在运行的脚本已经执行过写操作，那么即使执行 SCRIPT KILL ，也无法将它杀死，因为这是违反 Lua 脚本的原子性执行原则的。在这种情况下，唯一可行的办法是使用 SHUTDOWN NOSAVE 命令，通过停止整个 Redis 进程来停止脚本的运行，并防止不完整(half-written)的信息被写入数据库中。
关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行成功返回 OK ，否则返回一个错误。

```
# 没有脚本在执行时

redis> SCRIPT KILL
(error) ERR No scripts in execution right now.

# 成功杀死脚本时

redis> SCRIPT KILL
OK
(1.30s)

# 尝试杀死一个已经执行过写操作的脚本，失败

redis> SCRIPT KILL
(error) ERR Sorry the script already executed write commands against the dataset. You can either wait the script termination or kill the server in an hard way using the SHUTDOWN NOSAVE command.
(1.69s)
```

以下是脚本被杀死之后，返回给执行脚本的客户端的错误：

```
redis> EVAL "while true do end" 0
(error) ERR Error running script (call to
f_694a5fe1ddb97a4c6a1bf299d9537c7d3d0f84e7): Script killed by user
with SCRIPT KILL...
(5.00s)
```

Redis::__construct

说明：

创建一个 Redis 客户端

范例：

```
$redis = new Redis();
```

connect, open

说明：

实例连接到一个 Redis.

参数：

Host：**string**，可以是一个 **host** 地址，也可以是一个 **unix socket**

port: **int**

timeout: **float** *秒数，（可选参数，默认值为 0 代表不限制）*

返回值：

BOOL 成功返回：**TRUE**;失败返回：**FALSE**

范例：

```
$redis->connect('127.0.0.1', 6379);
$redis->connect('127.0.0.1'); // port 6379 by default
$redis->connect('127.0.0.1', 6379, 2.5); // 2.5 sec timeout.
$redis->connect('/tmp/redis.sock'); // unix domain socket.
```

pconnect, popen

说明：

实例连接到一个 **Redis.**，或者连接到一个已经通过 pconnect / popen 创建的连接上。

连接直到遇到 **close** 或者 **php** 进程结束才会被关闭。

参数：

***host*: string**

***port*: int**

***timeout*: float**

***persistent_id*: string** 持久链接的身份验证

返回值：

BOOL 成功返回：**TRUE;**失败返回：**FALSE**

范例：

```
$redis->pconnect('127.0.0.1', 6379);
$redis->pconnect('127.0.0.1'); // port 6379 by default - same connection like before.
$redis->pconnect('127.0.0.1', 6379, 2.5); // 2.5 sec timeout and would be another connection than the two before.
$redis->pconnect('127.0.0.1', 6379, 2.5, 'x'); // x is sent as persistent_id and would be another connection the the three before.
$redis->pconnect('/tmp/redis.sock'); // unix domain socket - would be another connection than the four before.
```

close

说明：

断开一个 **Redis** 实例连接，除非他是通过 pconnect 链接的。

setOption

说明：

创建客户端选项。

参数：

Name

Value

返回值：

BOOL 成功返回：TRUE;失败返回：FALSE

范例：

```
$redis->setOption(Redis::OPT_SERIALIZER, Redis::SERIALIZER_NONE);     // don't serialize data
$redis->setOption(Redis::OPT_SERIALIZER, Redis::SERIALIZER_PHP);       // use built-in serialize/unserialize
$redis->setOption(Redis::OPT_SERIALIZER, Redis::SERIALIZER_IGBINARY);     // use igBinary serialize/unserialize

$redis->setOption(Redis::OPT_PREFIX, 'myAppName:'); // use custom prefix on all keys
```

getOption

获得客户端选项

参数：

Name

返回值：

Value

范例：

```
$redis->getOption(Redis::OPT_SERIALIZER);     // return Redis::SERIALIZER_NONE, Redis::SERIALIZER_PHP, or Redis::SERIALIZER_IGBINARY.
```

ping

说明：

检查当前的连接状态。

参数：

无

返回值：

STRING：PONG 失败则会返回一个 Redis 抛出的连接异常。

get

说明：

获得一个指定的 key 的值。

参数：

Key

返回值：

String or *Bool*: 如果值存在则返回值，否则返回 false。

范例：

```
$redis->get('key');
```

set

说明：

创建一个值

参数：

Key

Value

Timeout（可选）可以在一定的 timeout 时间内让 SETEX 优先调用。

返回值：

成功返回 true

范例：

```
$redis->set('key', 'value');
```

setex

说明：

创建一个有一定存活时间的值

参数：

Key TTL Value

返回值：

成功返回 true

范例：

```
$redis->setex('key', 3600, 'value'); // sets key → value, with 1h TTL.
```

setnx

如果 key 的值不存在，则创建 key 的值为 value

参数：

Key

Value

返回值：

成功返回 true 失败返回 false

范例：

```
$redis->setnx('key', 'value'); /* return TRUE */
$redis->setnx('key', 'value'); /* return FALSE */
```

del, delete

说明：

删除一个指定的 key 的值

参数：

可以是一个数组，也可以是一个多个字符串。

返回值：

成功删除的个数

范例：

```
$redis->set('key1', 'val1');
$redis->set('key2', 'val2');
$redis->set('key3', 'val3');
$redis->set('key4', 'val4');
```

```
$redis->delete('key1', 'key2'); /* return 2 */
$redis->delete(array('key3', 'key4')); /* return 2 */
```

multi, exec, discard.

说明：

进入或者退出事务模式

参数：

(可选)

Redis::MULTI 或 Redis::PIPELINE. 默认是 Redis::MULTI

Redis::MULTI：将多个操作当成一个事务执行

Redis::PIPELINE:让（多条）执行命令简单的，更加快速的发送给服务器，但是没有任何原子性的保证

discard:删除一个事务

返回值：

multi()，返回一个 redis 对象，并进入 multi-mode 模式，一旦进入 multi-mode 模式，以后调用的所有方法都会返回相同的对象，只到 exec() 方法被调用。

范例：

```
$ret = $redis->multi()
       ->set('key1', 'val1')
       ->get('key1')
       ->set('key2', 'val2')
       ->get('key2')
       ->exec();

/*
$ret == array(
       0 => TRUE,
       1 => 'val1',
       2 => TRUE,
       3 => 'val2');
*/
```

watch, unwatch

说明：

监测一个 key 的值是否被其它的程序更改。如果这个 key 在 watch 和 exec （方法）间被修改，这个 MULTI/EXEC 事务的执行将失败（return false）

unwatch 取消被这个程序监测的所有 key

参数：

*Keys:*一对 key 的列表

范例：

```
$redis->watch('x');
/* long code here during the execution of which other clients could well modify `x` */
$ret = $redis->multi()
       ->incr('x')
       ->exec();
/*
$ret = FALSE if x has been modified between the call to WATCH and the call to EXEC.
*/
```

subscribe

说明：

方法回调。注意，该方法可能在未来里发生改变

参数：

channels: array

callback: 回调函数名

范例：

```
function f($redis, $chan, $msg) {
       switch($chan) {
              case 'chan-1':
                     ...
                     break;

              case 'chan-2':
                     ...
                     break;

              case 'chan-2':
                     ...
                     break;
```

```
        }
}
```

```
$redis->subscribe(array('chan-1', 'chan-2', 'chan-3'), 'f'); // subscribe to 3 chans
```

Publish

说明：

发表内容到某一个通道。注意，该方法可能在未来里发生改变

参数：

Channel：

*Messsage：*string

范例：

```
$redis->publish('chan-1', 'hello, world!'); // send message.
```

exists

说明：

验证指定的值是否存在

参数：

Key

返回值：

成功返回 true 失败返回 false

范例：

```
$redis->set('key', 'value');
$redis->exists('key'); /*   TRUE */
$redis->exists('NonExistingKey'); /* FALSE */
```

incr, incrBy

说明：

key 中的值进行自增.如果第二个参数存在，它将被用来作为整数值递增

参数：

Key

Value

返回值：

返回新 value

范例：

```
$redis->incr('key1'); /* key1 didn't exists, set to 0 before the increment */
                                        /* and now has the value 1   */

$redis->incr('key1'); /* 2 */
$redis->incr('key1'); /* 3 */
$redis->incr('key1'); /* 4 */
$redis->incrBy('key1', 10); /* 14 */
```

decr, decrBy

说明：

删掉 key 中的值，用法同 incr

范例：

```
$redis->decr('key1'); /* key1 didn't exists, set to 0 before the increment */
                                        /* and now has the value -1   */

$redis->decr('key1'); /* -2 */
$redis->decr('key1'); /* -3 */
$redis->decrBy('key1', 10); /* -13 */
```

getMultiple

说明：

返回一组数据的值，如果这个数组中的 key 值不存在，则返回 false

参数：

Array

返回值：

Array

范例：

```
$redis->set('key1', 'value1');
$redis->set('key2', 'value2');
$redis->set('key3', 'value3');
$redis->getMultiple(array('key1', 'key2', 'key3')); /* array('value1', 'value2',
'value3');
$redis->getMultiple(array('key0', 'key1', 'key5')); /* array(`FALSE`, 'value2', `FALSE`);
```

lPush

说明：

在名称为 key 的 list 左边（头）添加一个值为 value 的 元素，如果这个 key 值不存在则创建一个。如果 key 值存在并且不是一个 list，则返回 false

参数：

Key

Value

返回值：

返回 key 值得长度。

范例：

```
$redis->delete('key1');
$redis->lPush('key1', 'C'); // returns 1
$redis->lPush('key1', 'B'); // returns 2
$redis->lPush('key1', 'A'); // returns 3
/* key1 now points to the following list: [ 'A', 'B', 'C' ] */
```

rPush

说明：

在名称为 key 的 list 右边（尾）添加一个值为 value 的 元素，如果这个 key 值不存在则创建一个。如果 key 值存在并且不是一个 list，则返回 false

参数：

Key

Value

返回值：

返回 key 值得长度。

范例：

```
$redis->delete('key1');
$redis->rPush('key1', 'A'); // returns 1
$redis->rPush('key1', 'B'); // returns 2
$redis->rPush('key1', 'C'); // returns 3
/* key1 now points to the following list: [ 'A', 'B', 'C' ] */
```

lPushx

说明：

在名称为 key 的 list 左边（头）添加一个值为 value 的 元素，如果这个 value 存在则不添加。

参数：

Key

Value

返回值：

返回 key 值得长度。

范例：

```
$redis->delete('key1');
$redis->lPushx('key1', 'A'); // returns 0
$redis->lPush('key1', 'A'); // returns 1
$redis->lPushx('key1', 'B'); // returns 2
$redis->lPushx('key1', 'C'); // returns 3
/* key1 now points to the following list: [ 'A', 'B', 'C' ] */
```

rPushx

说明：

在名称为 key 的 list 右边（尾）添加一个值为 value 的 元素，如果这个 value 存在则不添加。

参数：

Key

Value

返回值：

返回 key 值得长度。

范例：

```
$redis->delete('key1');
$redis->rPushx('key1', 'A'); // returns 0
$redis->rPush('key1', 'A'); // returns 1
$redis->rPushx('key1', 'B'); // returns 2
$redis->rPushx('key1', 'C'); // returns 3
/* key1 now points to the following list: [ 'A', 'B', 'C' ] */
```

lPop

说明：

输出名称为 key 的 list 左(头)起起的第一个元素，删除该元素

参数：

Key

返回值：

失败返回 **false**

范例：

```
$redis->rPush('key1', 'A');
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C'); /* key1 => [ 'A', 'B', 'C' ] */
$redis->lPop('key1'); /* key1 => [ 'B', 'C' ] */
```

rPop

说明：

输出名称为 key 的 list 右(尾)起起的第一个元素，删除该元素

参数：

Key

返回值：

失败返回 **false**

范例：

```
$redis->rPush('key1', 'A');
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C'); /* key1 => [ 'A', 'B', 'C' ] */
$redis->rPop('key1'); /* key1 => [ 'A', 'B' ] */
```

blPop, brPop

说明：

lpop 命令的 **block** 版本。即当 **timeout** 为 **0** 时，若遇到名称为 **key** 的 **list** 不存在或该 **list** 为空，则命令结束。如果 **timeout > 0**，则遇到上述情况时，等待 **timeout** 秒，如果问题没有解决，则对 **key+1** 开始的 **list** 执行 **pop** 操作

参数：

Key

Timeout

返回值：

Array array('listName', 'element')

范例：

```
/* Non blocking feature */
$redis->lPush('key1', 'A');
$redis->delete('key2');

$redis->blPop('key1', 'key2', 10); /* array('key1', 'A') */
/* OR */
$redis->blPop(array('key1', 'key2'), 10); /* array('key1', 'A') */

$redis->brPop('key1', 'key2', 10); /* array('key1', 'A') */
/* OR */
$redis->brPop(array('key1', 'key2'), 10); /* array('key1', 'A') */

/* Blocking feature */

/* process 1 */
$redis->delete('key1');
$redis->blPop('key1', 10);
```

```
/* blocking for 10 seconds */

/* process 2 */
$redis->lPush('key1', 'A');

/* process 1 */
/* array('key1', 'A') is returned*/
```

lSize

说明：

返回这个 **key** 值 **list** 的个数，如果这个 **list** 不存在或为空，则返回 **0**，如果这个值得类型并不是一个 **list** 则返回 **false**。

参数：

Key

返回值：

Long or bool

范例：

```
$redis->rPush('key1', 'A');
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C'); /* key1 => [ 'A', 'B', 'C' ] */
$redis->lSize('key1');/* 3 */
$redis->rPop('key1');
$redis->lSize('key1');/* 2 */
```

lIndex, lGet

说明：

返回名称为 key 的 list 中 index 位置的元素，0 代表第一个，1 代表第二个，-1 代表最后一个，-2 代表倒数第二个，当这个 key 值不存在于 list 中时，返回 false。

参数：

key index

返回值：

String or false

范例：

```
$redis->rPush('key1', 'A');
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C'); /* key1 => [ 'A', 'B', 'C' ] */
$redis->lGet('key1', 0); /* 'A' */
$redis->lGet('key1', -1); /* 'C' */
$redis->lGet('key1', 10); /* `FALSE` */
```

lSet

说明：

设置名称为 key 的 list 中 index 位置的元素赋值为 value

参数：

Key

Index

Value

返回值：

Bool 成功返回 true 失败返回 false

范例：

```
$redis->rPush('key1', 'A');
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C'); /* key1 => [ 'A', 'B', 'C' ] */
$redis->lGet('key1', 0); /* 'A' */
$redis->lSet('key1', 0, 'X');
$redis->lGet('key1', 0); /* 'X' */
```

lRange, lGetRange

说明：

返回名称为 key 的 list 中 start 至 end 之间的元素（end 为 -1 ，返回所有）

参数：

Key

Start

End

返回值：

Array

范例：

```
$redis->rPush('key1', 'A');
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C');
$redis->lRange('key1', 0, -1); /* array('A', 'B', 'C') */
```

lTrim, listTrim

说明：

截取名称为 key 的 list，保留 start 至 end 之间的元素

参数：

Key

Start

Stop

返回值：

Array

范例：

```
$redis->rPush('key1', 'A');
```

```
$redis->rPush('key1', 'B');
$redis->rPush('key1', 'C');
$redis->lRange('key1', 0, -1); /* array('A', 'B', 'C') */
$redis->lTrim('key1', 0, 1);
$redis->lRange('key1', 0, -1); /* array('A', 'B') */
```

lRem, lRemove

说明：

从列表中从头部开始移除 count 个匹配的值。如果 count 为零，所有匹配的元素都被删除。如果 count 是负数，内容从尾部开始删除。

参数：

key

value

count

返回值：

LONG or bool

范例：

```
$redis->lPush('key1', 'A');
$redis->lPush('key1', 'B');
$redis->lPush('key1', 'C');
$redis->lPush('key1', 'A');
$redis->lPush('key1', 'A');

$redis->lRange('key1', 0, -1); /* array('A', 'A', 'C', 'B', 'A') */
$redis->lRem('key1', 'A', 2); /* 2 */
$redis->lRange('key1', 0, -1); /* array('C', 'B', 'A') */
```

lInsert

说明：

在名称为 key 的 list 中，找到值为 *pivot* 的 value，并根据参数 Redis::BEFORE | Redis::AFTER，来确定，newvalue 是放在 pivot 的前面，或者后面。如果 key 不存在，不会插入，如果 pivot 不存在，return -1

参数：

key position Redis::BEFORE | Redis::AFTER *pivot value*

返回值：

返回这个 list 的长度

如果 pivot 不存在 返回-1

范例：

```
$redis->delete('key1');
$redis->lInsert('key1', Redis::AFTER, 'A', 'X'); /* 0 */

$redis->lPush('key1', 'A');
$redis->lPush('key1', 'B');
$redis->lPush('key1', 'C');

$redis->lInsert('key1', Redis::BEFORE, 'C', 'X'); /* 4 */
$redis->lRange('key1', 0, -1); /* array('A', 'B', 'X', 'C') */

$redis->lInsert('key1', Redis::AFTER, 'C', 'Y'); /* 5 */
$redis->lRange('key1', 0, -1); /* array('A', 'B', 'X', 'C', 'Y') */

$redis->lInsert('key1', Redis::AFTER, 'W', 'value'); /* -1 */
```

sAdd

说明：

向名称为 key 的 set 中添加元素 value,如果 value 存在，不写入，return false

参数：

key

value

返回值：

Bool 成功返回 true 失败或已存在 value 值则返回 false

范例：

```
$redis->sAdd('key1' , 'set1'); /* TRUE, 'key1' => {'set1'} */
$redis->sAdd('key1' , 'set2'); /* TRUE, 'key1' => {'set1', 'set2'}*/
$redis->sAdd('key1' , 'set2'); /* FALSE, 'key1' => {'set1', 'set2'}*/
```

sRem, sRemove

删除名称为 key 的 set 中的元素 value

参数:

key

member

返回值:

Bool

范例:

```
$redis->sAdd('key1' , 'set1');
$redis->sAdd('key1' , 'set2');
$redis->sAdd('key1' , 'set3'); /* 'key1' => {'set1', 'set2', 'set3'}*/
$redis->sRem('key1', 'set2'); /* 'key1' => {'set1', 'set3'} */
```

sMove

说明:

将 value 元素从名称为 srckey 的集合移到名称为 dstkey 的集合

参数:

srcKey

dstKey

member

返回值:

Bool 成功返回 true 失败返回 false

范例：

```
$redis->sAdd('key1' , 'set11');
$redis->sAdd('key1' , 'set12');
$redis->sAdd('key1' , 'set13'); /* 'key1' => {'set11', 'set12', 'set13'}*/
$redis->sAdd('key2' , 'set21');
$redis->sAdd('key2' , 'set22'); /* 'key2' => {'set21', 'set22'}*/
$redis->sMove('key1', 'key2', 'set13'); /* 'key1' =>  {'set11', 'set12'} */
                                        /* 'key2' =>  {'set21', 'set22', 'set13'} */
```

sIsMember, sContains

说明：

名称为 key 的集合中查找是否有 value 元素

参数：

key

value

返回值：

Bool 存在返回 true 不存在返回 false

范例：

```
$redis->sAdd('key1' , 'set1');
$redis->sAdd('key1' , 'set2');
$redis->sAdd('key1' , 'set3'); /* 'key1' => {'set1', 'set2', 'set3'}*/

$redis->sIsMember('key1', 'set1'); /* TRUE */
$redis->sIsMember('key1', 'setX'); /* FALSE */
```

sCard, sSize

说明：

返回名称为 key 的 set 的元素个数

参数：

Key

返回值：

Long 元素个数，不存在则返回 0

范例：

```
$redis->sAdd('key1' , 'set1');
$redis->sAdd('key1' , 'set2');
$redis->sAdd('key1' , 'set3'); /* 'key1' => {'set1', 'set2', 'set3'}*/
$redis->sCard('key1'); /* 3 */
$redis->sCard('keyX'); /* 0 */
```

sPop

说明：

随机返回并删除名称为 key 的 set 中一个元素

参数：

key

返回值：

返回被随机取得的值，如果失败返回 false

范例：

```
$redis->sAdd('key1' , 'set1');
$redis->sAdd('key1' , 'set2');
$redis->sAdd('key1' , 'set3'); /* 'key1' => {'set3', 'set1', 'set2'}*/
$redis->sPop('key1'); /* 'set1', 'key1' => {'set3', 'set2'} */
$redis->sPop('key1'); /* 'set3', 'key1' => {'set2'} */
```

sRandMember

说明：

随机返回名称为 key 的 set 中一个元素

参数：

Key

返回值：

返回的 value 的值，失败返回 false

范例：

```
$redis->sAdd('key1' , 'set1');
$redis->sAdd('key1' , 'set2');
$redis->sAdd('key1' , 'set3'); /* 'key1' => {'set3', 'set1', 'set2'}*/
$redis->sRandMember('key1'); /* 'set1', 'key1' => {'set3', 'set1', 'set2'} */
$redis->sRandMember('key1'); /* 'set3', 'key1' => {'set3', 'set1', 'set2'} */
```

sInter

说明：

求交集

参数：

Key1，key2….keyN

返回值：

Array 返回交集的数组，如果交集为空，则返回一个空数组

范例：

```
$redis->sAdd('key1', 'val1');
$redis->sAdd('key1', 'val2');
$redis->sAdd('key1', 'val3');
$redis->sAdd('key1', 'val4');

$redis->sAdd('key2', 'val3');
$redis->sAdd('key2', 'val4');

$redis->sAdd('key3', 'val3');
$redis->sAdd('key3', 'val4');
```

```
var_dump($redis->sInter('key1', 'key2', 'key3'));
```

Output:

```
array(2) {
   [0]=>
   string(4) "val4"
   [1]=>
   string(4) "val3"
}
```

sInterStore

说明：

执行 sInter 命令并把结果储存到新建的变量中。

参数：

Key

Key2:key1,key2...keyN

返回值：

范例：

```
$redis->sAdd('key1', 'val1');
$redis->sAdd('key1', 'val2');
$redis->sAdd('key1', 'val3');
$redis->sAdd('key1', 'val4');

$redis->sAdd('key2', 'val3');
$redis->sAdd('key2', 'val4');

$redis->sAdd('key3', 'val3');
$redis->sAdd('key3', 'val4');

var_dump($redis->sInterStore('output', 'key1', 'key2', 'key3'));
var_dump($redis->sMembers('output'));
```

Output:

```
int(2)

array(2) {
   [0]=>
```

```
  string(4) "val4"
  [1]=>
  string(4) "val3"
}
```

sUnion

说明：

合并多个 key 值

参数：

Keys：key1，key2…..keyN

返回值：

这些 key 生成的合集

范例：

```
$redis->delete('s0', 's1', 's2');

$redis->sAdd('s0', '1');
$redis->sAdd('s0', '2');
$redis->sAdd('s1', '3');
$redis->sAdd('s1', '1');
$redis->sAdd('s2', '3');
$redis->sAdd('s2', '4');

var_dump($redis->sUnion('s0', 's1', 's2'));
```

Return value: all elements that are either in s0 or in s1 or in s2.

```
array(4) {
  [0]=>
  string(1) "3"
  [1]=>
  string(1) "4"
  [2]=>
  string(1) "1"
  [3]=>
  string(1) "2"
}
```

sUnionStore

说明：

执行 sUnion 命令并把结果储存到新建的变量中。

参数：

Key：

Keys：

返回值：

范例：

```
$redis->delete('s0', 's1', 's2');

$redis->sAdd('s0', '1');
$redis->sAdd('s0', '2');
$redis->sAdd('s1', '3');
$redis->sAdd('s1', '1');
$redis->sAdd('s2', '3');
$redis->sAdd('s2', '4');

var_dump($redis->sUnionStore('dst', 's0', 's1', 's2'));
var_dump($redis->sMembers('dst'));
```

Return value: the number of elements that are either in s0 or in s1 or in s2.

```
int(4)
array(4) {
  [0]=>
  string(1) "3"
  [1]=>
  string(1) "4"
  [2]=>
  string(1) "1"
  [3]=>
  string(1) "2"
}
```

sDiff

说明：

求差集

参数：

Keys

返回值：

Array

范例：

```
$redis->delete('s0', 's1', 's2');

$redis->sAdd('s0', '1');
$redis->sAdd('s0', '2');
$redis->sAdd('s0', '3');
$redis->sAdd('s0', '4');

$redis->sAdd('s1', '1');
$redis->sAdd('s2', '3');

var_dump($redis->sDiff('s0', 's1', 's2'));
```

Return value: all elements of s0 that are neither in s1 nor in s2.

```
array(2) {
   [0]=>
   string(1) "4"
   [1]=>
   string(1) "2"
}
```

sDiffStore

说明：

求差集并把结果储存到新建的变量中。

参数：

Key

Keys

返回值：

范例：

```
$redis->delete('s0', 's1', 's2');

$redis->sAdd('s0', '1');
$redis->sAdd('s0', '2');
$redis->sAdd('s0', '3');
$redis->sAdd('s0', '4');

$redis->sAdd('s1', '1');
$redis->sAdd('s2', '3');

var_dump($redis->sDiffStore('dst', 's0', 's1', 's2'));
var_dump($redis->sMembers('dst'));
```

Return value: the number of elements of s0 that are neither in s1 nor in s2.

```
int(2)
array(2) {
   [0]=>
   string(1) "4"
   [1]=>
   string(1) "2"
}
```

sMembers, sGetMembers

说明：

返回名称为 key 的 set 的所有元素

参数：

Key

返回值：

array

范例：

```
$redis->delete('s');
$redis->sAdd('s', 'a');
$redis->sAdd('s', 'b');
$redis->sAdd('s', 'a');
$redis->sAdd('s', 'c');
var_dump($redis->sMembers('s'));
```

Output:

```
array(3) {
   [0]=>
   string(1) "c"
   [1]=>
   string(1) "a"
   [2]=>
   string(1) "b"
}
```

getSet

说明：

返回原来 key 中的值，并将 value 写入 key

参数：

Key

Value

返回值：

这个 key 的前一个值

范例：

```
$redis->set('x', '42');
```

```
$exValue = $redis->getSet('x', 'lol');   // return '42', replaces x by 'lol'
$newValue = $redis->get('x')'             // return 'lol'
```

randomKey

说明：

随机返回 key 空间的一个 key

参数：

无

返回值：

在 redis 中随机存在的一个 key

范例：

```
$key = $redis->randomKey();
$surprise = $redis->get($key);   // who knows what's in there.
```

select

说明：

选择一个数据库

参数：

Dbindex

返回值：

Bool

范例：

```
$redis->select(0);   // switch to DB 0
$redis->set('x', '42'); // write 42 to x
$redis->move('x', 1);     // move to DB 1
$redis->select(1);   // switch to DB 1
$redis->get('x');     // will return 42
```

move

说明：

转移一个 key 到另外一个数据库

参数：

Key

返回值：

Bool

范例：

```
$redis->select(0);   // switch to DB 0
$redis->set('x', '42'); // write 42 to x
$redis->move('x', 1);     // move to DB 1
$redis->select(1);   // switch to DB 1
$redis->get('x');     // will return 42
```

rename, renameKey

说明：

重命名 key

参数：

Srckey

dstkey

返回值：

Bool

范例：

```
$redis->set('x', '42');
$redis->rename('x', 'y');
$redis->get('y');     // → 42
$redis->get('x');     // → `FALSE`
```

renameNx

与 remane 类似，但是，如果重新命名的名字已经存在，不会替换成功

setTimeout, expire

说明：

设定一个 key 的活动时间（s）

参数：

Key

返回值：

Bool

范例：

```
$redis->set('x', '42');
```

```
$redis->setTimeout('x', 3); // x will disappear in 3 seconds.
sleep(5);                              // wait 5 seconds
$redis->get('x');             // will return `FALSE`, as 'x' has expired.
```

expireAt

说明：

key 存活到一个 unix 时间戳时间

参数：

Key

Unix timestamp

返回值：

Bool

范例：

```
$redis->set('x', '42');
$now = time(NULL); // current timestamp
$redis->expireAt('x', $now + 3);       // x will disappear in 3 seconds.
sleep(5);                              // wait 5 seconds
$redis->get('x');             // will return `FALSE`, as 'x' has expired.
```

keys, getKeys

说明：

返回满足给定 pattern 的所有 key

参数：

Pattern (可带*)

返回值：

Array

范例：

```
$allKeys = $redis->keys('*');     // all keys will match this.
$keyWithUserPrefix = $redis->keys('user*');
```

dbSize

说明：

查看现在数据库有多少 key

参数：

无

返回值：

DB size,

范例：

```
$count = $redis->dbSize();
echo "Redis has $count keys\n";
```

auth

说明：

密码验证

参数：

password

返回值：

BOOL

范例：

```
$redis->auth('foobared');
```

bgrewriteaof

说明：

使用 aof 来进行数据库持久化

参数：

无

返回值：

Bool

范例：

```
$redis->bgrewriteaof();
```

slaveof

说明：

选择从服务器

参数：

host (string) and port

返回值：

BOOL

范例：

```
$redis->slaveof('10.0.1.7', 6379);
/* ... */
$redis->slaveof();
```

object

说明：

获得 key 对象的详细内容

参数：

- "encoding"
- "refcount"
- "idletime"

返回值：

STRING for "encoding",

LONG for "refcount" and "idletime",

FALSE if the key doesn't exist.

范例：

```
$redis->object("encoding", "l"); // → ziplist
$redis->object("refcount", "l"); // → 1
$redis->object("idletime", "l"); // → 400 (in seconds, with a precision of 10 seconds).
```

save

说明：

将数据同步保存到磁盘

参数：

无

返回值：

Bool

范例：

```
$redis->save();
```

bgsave

说明：

将数据异步保存到磁盘

参数：

无

返回值：

Bool

范例：

```
$redis->bgSave();
```

lastSave

说明：

返回上次成功将数据保存到磁盘的 Unix 时间戳

参数：

无

返回值：

timestamp

范例：

```
$redis->lastSave();
```

type

说明：

返回 key 的类型值

参数：

Key

返回值：

根据指定的类型返回

string: Redis::REDIS_STRING

set: Redis::REDIS_SET

list: Redis::REDIS_LIST

zset: Redis::REDIS_ZSET

hash: Redis::REDIS_HASH

other: Redis::REDIS_NOT_FOUND

范例：

```
$redis->type('key');
```

append

说明：

在指定的一个 **key** 值后面追加一个值

参数：

Key

Value

返回值：

追加完之后这个 **key** 值得长度。

范例：

```
$redis->set('key', 'value1');
```

```
$redis->append('key', 'value2'); /* 12 */
$redis->get('key'); /* 'value1value2' */
```

getRange (方法不存在)

说明：

返回名称为 **key** 的 **string** 中 **start** 至 **end** 之间的字符

参数：

key

start

end

返回值：

截取之后的值

范例：

```
$redis->set('key', 'string value');
$redis->getRange('key', 0, 5); /* 'string' */
$redis->getRange('key', -5, -1); /* 'value' */
```

setRange

说明：

改变 **key** 的 **string** 中 **start** 至 **end** 之间的字符为 **value**

参数：

key

offset

value

返回值：

修改后字符的长度

范例：

```
$redis->set('key', 'Hello world');
$redis->setRange('key', 6, "redis"); /* returns 11 */
$redis->get('key'); /* "Hello redis" */
```

strlen

说明：

获得一个指定 **key** 的长度

参数：

key

返回值：

长度

范例：

```
$redis->set('key', 'value');
$redis->strlen('key'); /* 5 */
```

getBit

说明：

返回一个指定 **key** 的二进制信息

参数：

key

offset

返回值：

LONG

范例：

```
$redis->set('key', "\x7f"); // this is 0111 1111
$redis->getBit('key', 0); /* 0 */
$redis->getBit('key', 1); /* 1 */
```

setBit

说明：

给一个指定 **key** 的值得第 **offset** 位 赋值为 **value**。

参数：

key

offset

value: bool or int (1 or 0)

返回值：

LONG: 0 or 1

范例：

```
$redis->set('key', "*");       // ord("*") = 42 = 0x2f = "0010 1010"
$redis->setBit('key', 5, 1); /* returns 0 */
$redis->setBit('key', 7, 1); /* returns 0 */
$redis->get('key'); /* chr(0x2f) = "/" = b("0010 1111") */
```

flushDB

说明：

清空当前数据库

参数：

无

返回值：

Bool:永远都返回 true

范例：

```
$redis->flushDB();
```

flushAll

说明：

清空所有数据库

参数：

无

返回值：

Bool:永远都返回 true

范例：

```
$redis->flushAll();
```

sort

说明：

排序，分页等

参数：

'by' => 'some_pattern_*',

'limit' => array(0, 1),

'get' => 'some_other_pattern_*' or an array of patterns,

'sort' => 'asc' or 'desc',

'alpha' => TRUE,

'store' => 'external-key'

返回值：

Array

范例：

```
$redis->delete('s');
$redis->sadd('s', 5);
$redis->sadd('s', 4);
$redis->sadd('s', 2);
$redis->sadd('s', 1);
$redis->sadd('s', 3);

var_dump($redis->sort('s')); // 1,2,3,4,5
var_dump($redis->sort('s', array('sort' => 'desc'))); // 5,4,3,2,1
var_dump($redis->sort('s', array('sort' => 'desc', 'store' => 'out'))); // (int)5
```

info

说明：

返回 redis 的版本信息等详情

参数：

无

返回值：

范例：

```
$redis->info();
```

resetStat

说明：

重新统计输出 INFO 命令的结果

参数：

无

返回值：

BOOL

范例：

```
$redis->resetStat();
```

ttl

说明：

得到一个 **key** 的生存时间，如果这个 **key** 值不存在则返回 **false**

参数：

Key

返回值：

Long or bool

范例：

```
$redis->ttl('key');
```

persist

说明：

移除生存时间到期的 **key**

参数：

Key

返回值：

Bool 如果移除成功了返回 **true** ，如果值不存在或是还在生存时间内则返回 **false**

范例：

```
$redis->persist('key');
```

mset, msetnx

说明：

同时给多个 **key** 赋值 ， **MSETNX** 只有当给所有的值都创建成功的时候才会返回 **true**

参数：

array(key => value, ...)

返回值：

Bool

范例：

```
$redis->mset(array('key0' => 'value0', 'key1' => 'value1'));
var_dump($redis->get('key0'));
var_dump($redis->get('key1'));
```

Output:

```
string(6) "value0"
string(6) "value1"
```

rpoplpush （redis 版本 1.1 以上才可以）

说明：

返回并删除名称为 srckey 的 list 的尾元素，并将该元素添加到名称为 dstkey 的 list 的头部

参数：

Key: srckey

Key: dstkey

返回值：

Bool

范例：

```
$redis->delete('x', 'y');

$redis->lPush('x', 'abc');
$redis->lPush('x', 'def');
$redis->lPush('y', '123');
$redis->lPush('y', '456');

// move the last of x to the front of y.
var_dump($redis->rpoplpush('x', 'y'));
var_dump($redis->lRange('x', 0, -1));
var_dump($redis->lRange('y', 0, -1));
```

Output:

```
string(3) "abc"
array(1) {
   [0]=>
   string(3) "def"
}
array(3) {
   [0]=>
   string(3) "abc"
```

```
  [1]=>
  string(3) "456"
  [2]=>
  string(3) "123"
}
```

brpoplpush

说明：

Rpoplpush 命令的 **block** 版本。

参数：

Key: srckey

Key: dstkey

Long: timeout

返回值：

Bool

范例：

zAdd

说明：

向名称为 **key** 的 **zset** 中添加元素 **member**，**score** 用于排序。如果该元素已经存在，则根据 **score** 更新该元素的顺序。

参数：

key

score : double

value: string

返回值：

Long 元素被成功添加了返回 **1** 否则返回 **0**

范例：

```
$redis->zAdd('key', 1, 'val1');
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 5, 'val5');
$redis->zRange('key', 0, -1); // array(val0, val1, val5)
```

zRange

说明：

返回名称为 **key** 的 **zset**（元素已按 **score** 从小到大排序）中的 **index** 从 **start** 到 **end** 的所有元素

参数：

key

start: long

end: long

withscores: bool = false

返回值：

Array

范例：

```
$redis->zAdd('key1', 0, 'val0');
$redis->zAdd('key1', 2, 'val2');
$redis->zAdd('key1', 10, 'val10');
$redis->zRange('key1', 0, -1); /* array('val0', 'val2', 'val10') */

// with scores
$redis->zRange('key1', 0, -1, true); /* array('val0' => 0, 'val2' => 2, 'val10' => 10) */
```

zDelete, zRem

说明：

删除名称为 key 的 zset 中的元素 member

参数：

key

member

返回值：

LONG 成功返回 1 失败返回 0

范例：

```
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 2, 'val2');
$redis->zAdd('key', 10, 'val10');
$redis->zDelete('key', 'val2');
$redis->zRange('key', 0, -1); /* array('val0', 'val10') */
```

zRevRange

说明：

返回名称为 **key** 的 **zset**（元素已按 **score** 从大到小排序）中的 **index** 从 **start** 到 **end** 的所有元素.**withscores:** 是否输出 **socre** 的值，默认 **false**，不输出

参数：

key

start: long

end: long

withscores: bool = false

返回值：

Array

范例：

```
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 2, 'val2');
$redis->zAdd('key', 10, 'val10');
$redis->zRevRange('key', 0, -1); /* array('val10', 'val2', 'val0') */

// with scores
$redis->zRevRange('key', 0, -1, true); /* array('val10' => 10, 'val2' => 2, 'val0' => 0)
*/
```

zRangeByScore, zRevRangeByScore

说明：

返回名称为 **key** 值中 **score >= star** 且 **score <= end** 的所有元素

参数：

key

start: string

end: string

options: array

返回值：

Array

范例：

```
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 2, 'val2');
$redis->zAdd('key', 10, 'val10');
$redis->zRangeByScore('key', 0, 3); /* array('val0', 'val2') */
$redis->zRangeByScore('key', 0, 3, array('withscores' => TRUE); /* array('val0' => 0, 'val2' => 2) */
$redis->zRangeByScore('key', 0, 3, array('limit' => array(1, 1)); /* array('val2' => 2) */
$redis->zRangeByScore('key', 0, 3, array('limit' => array(1, 1)); /* array('val2') */
$redis->zRangeByScore('key', 0, 3, array('withscores' => TRUE, 'limit' => array(1, 1)); /* array('val2' => 2) */
```

zCount

说明：

返回名称为 **key** 值中 **score >= star** 且 **score <= end** 的所有元素的个数

参数：

key

start: string

end: string

返回值：

LONG 返回相应结果的长度

范例：

```
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 2, 'val2');
$redis->zAdd('key', 10, 'val10');
$redis->zCount('key', 0, 3); /* 2, corresponding to array('val0', 'val2') */
```

zRemRangeByScore, zDeleteRangeByScore

说明：

删除名称为 **key** 的值中 **score >= star** 且 **score <= end** 的所有元素，返回删除个数

参数：

key

start: double or "+inf" or "-inf" string

end: double or "+inf" or "-inf" string

返回值：

LONG 删除的个数

范例：

```
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 2, 'val2');
$redis->zAdd('key', 10, 'val10');
$redis->zRemRangeByScore('key', 0, 3); /* 2 */
```

zRemRangeByRank, zDeleteRangeByRank

说明：

移除有序集 **key** 中，指定排名**(rank)**区间内的所有成员。

区间分别以下标参数 **start** 和 **stop** 指出，包含 **start** 和 **stop** 在内。

参数：

key

start: LONG

end: LONG

返回值：

LONG

范例：

```
$redis->zAdd('key', 1, 'one');
$redis->zAdd('key', 2, 'two');
$redis->zAdd('key', 3, 'three');
$redis->zRemRangeByRank('key', 0, 1); /* 2 */
```

```
$redis->zRange('key', 0, -1, array('withscores' => TRUE)); /* array('three' => 3) */
```

zSize, zCard

说明：

返回名称为 **key** 的值的所有元素的个数

参数：

key

返回值：

Long

范例：

```
$redis->zAdd('key', 0, 'val0');
$redis->zAdd('key', 2, 'val2');
$redis->zAdd('key', 10, 'val10');
$redis->zSize('key'); /* 3 */
```

zScore

说明：

返回名称为 **key** 的值中元素 **member** 的 **score**

参数：

key

member

返回值：

Double

范例：

```
$redis->zAdd('key', 2.5, 'val2');
$redis->zScore('key', 'val2'); /* 2.5 */
```

zRank, zRevRank

说明：

返回名称为 **key** 的值（元素已按 **score** 从小到大排序）中 **member** 元素的 **rank**(即 **index**,从 **0** 开始),若没有 **member** 元素，返回**“null”**。**zRevRank** 是从大到小排序

参数：

key

member

返回值：

范例：

```
$redis->delete('z');
$redis->zAdd('key', 1, 'one');
$redis->zAdd('key', 2, 'two');
$redis->zRank('key', 'one'); /* 0 */
$redis->zRank('key', 'two'); /* 1 */
$redis->zRevRank('key', 'one'); /* 1 */
$redis->zRevRank('key', 'two'); /* 0 */
```

zIncrBy

说明：

如果在名称为 key 的值中已经存在元素 member，则该元素的 score 增加 increment；否则向集合中添加该元素，其 score 的值为 increment

参数：

key

value: (double) value that will be added to the member's score

member

返回值：

DOUBLE

范例：

```
$redis->delete('key');
$redis->zIncrBy('key', 2.5, 'member1'); /* key or member1 didn't exist, so member1's score is to 0 before the increment */
                                        /* and now has the value 2.5  */
$redis->zIncrBy('key', 1, 'member1'); /* 3.5 */
```

zUnion

说明：

对 N 个 ZSetKeys 求并集，并将最后的集合保存在 dstkeyN 中。对于集合中每一个元素的 score，在进行 AGGREGATE 运算前，都要乘以对于的 WEIGHT 参数。如果没有提供 WEIGHT，默认为 1。默认的 AGGREGATE 是 SUM，即结果集合中元素的 score

是所有集合对应元素进行 SUM 运算的值，而 MIN 和 MAX 是指，结果集合中元素的 score 是所有集合对应元素中最小值和最大值。

参数：

keyOutput

arrayZSetKeys

arrayWeights

aggregateFunction Either "SUM", "MIN", or "MAX": defines the behaviour to use on duplicate entries during the zUnion.

返回值：

LONG

范例：

```
$redis->delete('k1');
$redis->delete('k2');
$redis->delete('k3');
$redis->delete('ko1');
$redis->delete('ko2');
$redis->delete('ko3');

$redis->zAdd('k1', 0, 'val0');
$redis->zAdd('k1', 1, 'val1');

$redis->zAdd('k2', 2, 'val2');
$redis->zAdd('k2', 3, 'val3');

$redis->zUnion('ko1', array('k1', 'k2')); /* 4, 'ko1' => array('val0', 'val1', 'val2', 'val3') */

/* Weighted zUnion */
```

```
$redis->zUnion('ko2', array('k1', 'k2'), array(1, 1)); /* 4, 'ko1' => array('val0', 'val1', 'val2', 'val3') */
$redis->zUnion('ko3', array('k1', 'k2'), array(5, 1)); /* 4, 'ko1' => array('val0', 'val2', 'val3', 'val1') */
```

zInter

说明：

对 N 个 ZSetKeys 求交集，并将最后的集合保存在 dstkeyN 中。对于集合中每一个元素的 score，在进行 AGGREGATE 运算前，都要乘以对于的 WEIGHT 参数。如果没有提供 WEIGHT，默认为 1。默认的 AGGREGATE 是 SUM，即结果集合中元素的 score 是所有集合对应元素进行 SUM 运算的值，而 MIN 和 MAX 是指，结果集合中元素的 score 是所有集合对应元素中最小值和最大值。

参数：

keyOutput

arrayZSetKeys

arrayWeights

aggregateFunction Either "SUM", "MIN", or "MAX": defines the behaviour to use on duplicate entries during the zUnion.

返回值：

LONG

范例：

```
$redis->delete('k1');
$redis->delete('k2');
$redis->delete('k3');
$redis->delete('ko1');
$redis->delete('ko2');
$redis->delete('ko3');

$redis->zAdd('k1', 0, 'val0');
$redis->zAdd('k1', 1, 'val1');

$redis->zAdd('k2', 2, 'val2');
$redis->zAdd('k2', 3, 'val3');

$redis->zUnion('ko1', array('k1', 'k2')); /* 4, 'ko1' => array('val0', 'val1', 'val2',
'val3') */

/* Weighted zUnion */
$redis->zUnion('ko2', array('k1', 'k2'), array(1, 1)); /* 4, 'ko1' => array('val0', 'val1',
'val2', 'val3') */
$redis->zUnion('ko3', array('k1', 'k2'), array(5, 1)); /* 4, 'ko1' => array('val0', 'val2',
'val3', 'val1') */
```

hSet

说明：

向名称为 key 的 hash 中添加元素 hashKey—> value

参数：

key

hashKey

value

返回值：

Long:如果这个值不存在并且被添加成功返回 **1**，如果这个值存在并且被替代返回 **0**，错误返回 **false**

范例：

```
$redis->delete('h')
$redis->hSet('h', 'key1', 'hello'); /* 1, 'key1' => 'hello' in the hash at "h" */
$redis->hGet('h', 'key1'); /* returns "hello" */

$redis->hSet('h', 'key1', 'plop'); /* 0, value was replaced. */
$redis->hGet('h', 'key1'); /* returns "plop" */
```

hSetNx

说明：

向名称为 **key** 的 **hash** 中添加元素 **hashKey—> value** ，只当这个值不存在的时候生效。

参数：

key

hashKey

value

返回值：

Bool

范例：

```
$redis->delete('h')
$redis->hSetNx('h', 'key1', 'hello'); /* TRUE, 'key1' => 'hello' in the hash at "h" */
$redis->hSetNx('h', 'key1', 'world'); /* FALSE, 'key1' => 'hello' in the hash at "h". No change since the field wasn't replaced.
```

hGet

说明：

返回指定名称为 ***key*** 的 **hash** 中 ***hashKey*** 对应的值

参数：

key

hashKey

返回值：

成功取到这个值则返回这个值，否则返回 **false**

范例：

hLen

说明：

返回名称为 **key** 的 **hash** 中元素个数

参数：

key

返回值：

LONG 成功则返回 **hase** 中元素的个数，失败则返回 **false**

范例：

```
$redis->delete('h')
$redis->hSet('h', 'key1', 'hello');
$redis->hSet('h', 'key2', 'plop');
$redis->hLen('h'); /* returns 2 */
```

hDel

说明：

删除名称为 **key** 的 **hash** 中键为 ***hashKey*** 的域

参数：

key

hashKey

返回值：

bool

范例：

hKeys

说明：

返回名称为 **key** 的 **hash** 中所有键值

参数：

key

返回值：

Array 类似于 **php** 中的 **array_keys** 函数

范例：

```
$redis->delete('h');
$redis->hSet('h', 'a', 'x');
$redis->hSet('h', 'b', 'y');
$redis->hSet('h', 'c', 'z');
$redis->hSet('h', 'd', 't');
var_dump($redis->hKeys('h'));
```

Output:

```
array(4) {
   [0]=>
   string(1) "a"
   [1]=>
   string(1) "b"
   [2]=>
   string(1) "c"
   [3]=>
   string(1) "d"
}
```

hVals

说明：

返回名称为 **key** 的 **hash** 中所有值

参数：

Key

返回值：

Array

范例：

```
$redis->delete('h');
$redis->hSet('h', 'a', 'x');
$redis->hSet('h', 'b', 'y');
$redis->hSet('h', 'c', 'z');
$redis->hSet('h', 'd', 't');
var_dump($redis->hVals('h'));
```

Output:

```
array(4) {
   [0]=>
   string(1) "x"
   [1]=>
   string(1) "y"
   [2]=>
   string(1) "z"
   [3]=>
   string(1) "t"
}
```

hGetAll

说明：

返回一个完整的 hash

参数：

Key

返回值：

array

范例：

```
$redis->delete('h');
$redis->hSet('h', 'a', 'x');
$redis->hSet('h', 'b', 'y');
$redis->hSet('h', 'c', 'z');
$redis->hSet('h', 'd', 't');
var_dump($redis->hGetAll('h'));
```

Output:

```
array(4) {
   ["a"]=>
   string(1) "x"
   ["b"]=>
   string(1) "y"
   ["c"]=>
   string(1) "z"
   ["d"]=>
   string(1) "t"
}
```

hExists

说明：

名称为 **key** 的 **hash** 中是否存在键名字为 **memberKey** 的域

参数：

key

memberKey

返回值：

BOOL

范例：

```
$redis->hSet('h', 'a', 'x');
$redis->hExists('h', 'a'); /*   TRUE */
$redis->hExists('h', 'NonExistingKey'); /* FALSE */
```

hIncrBy

说明：

将名称为 **key** 的 **hash** 中 **member** 的值增加 **value**

参数：

key

member

value: (integer) value that will be added to the member's value

返回值：

LONG 返回这个新值

范例：

```
$redis->delete('h');
$redis->hIncrBy('h', 'x', 2); /* returns 2: h[x] = 2 now. */
$redis->hIncrBy('h', 'x', 1); /* h[x] ← 2 + 1. Returns 3 */
```

hMset

说明：

向名称为 **key** 的 **hash** 中批量添加元素

参数：

Key

members: key → value array

返回值：

BOOL

范例：

```
$redis->delete('user:1');
$redis->hMset('user:1', array('name' => 'Joe', 'salary' => 2000));
$redis->hIncrBy('user:1', 'salary', 100); // Joe earns 100 more now.
```

hMGet

说明：

返回名称为 **key** 的 **hash** 中 **member** 数组中的值所对应的在 **hash** 中的 **value**

参数：

key

member:Keys Array

返回值：

array

范例：

```
$redis->delete('h');
$redis->hSet('h', 'field1', 'value1');
$redis->hSet('h', 'field2', 'value2');
$redis->hmGet('h', array('field1', 'field2')); /* returns array('field1' => 'value1',
'field2' => 'value2') */
```