字符串

set

```
set key values [ex|px] [nx|xx]
- ex ： 设置过期时间 单位秒
- px ：设置过期时间 单位毫秒
- nx ： 键 key 不存在，设置成功 键已存在设置失败
- xx ： 键存在设置成功 键不存在设置失败
```

setnx

```
setnx key value
- 若键存在则不做任何动作 键 key 不存在时设设置为 value
- setinx 是 set if not exists 的缩写
- 命令在设置成功时返回 1 ， 设置失败时返回 0 。
```

setex

```
setex key 60 10086
将 key 的值设置成 10086 ttl 为60
将键 key 的值设置为 value ， 并将键 key 的生存时间设置为 seconds 秒钟。
如果键 key 已经存在， 那么 SETEX 命令将覆盖已有的值。
命令在设置成功时返回 OK 。 当 seconds 参数不合法时， 命令将返回一个错误。
```

psetex

```
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
```

get

```
get key
如果键 key 不存在， 那么返回特殊值 nil ； 否则， 返回键 key 的值
```

getset

```
getset key value
将键 key 的值设为 value ， 并返回键 key 在被设置之前的旧值。

返回给定键 key 的旧值。
如果键 key 没有旧值， 也即是说， 键 key 在被设置之前并不存在， 那么命令返回 nil 。
当键 key 存在但不是字符串类型时， 命令返回一个错误。
```

strlen

```
strlen key value
返回键 key 储存的字符串值的长度。
```

append

```
append key value
如果键 key 已经存在并且它的值是一个字符串， append 命令将把 value 追加到键 key 现有值的末尾。
如果 key 不存在， append 就简单地将键 key 的值设为 value ， 就像执行 set key value 一样。

返回值
追加 value 之后， 键 key 的值的长度。
```

setrange

```
setrange key offset value
从偏移量 offset 开始， 用 value 参数覆写(overwrite)键 key 储存的字符串值。

redis> SET greeting "hello world"
OK
redis> SETRANGE greeting 6 "Redis"
(integer) 11
redis> GET greeting
"hello Redis"

SETRANGE 命令会确保字符串足够长以便将 value 设置到指定的偏移量上， 如果键 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
空白将用零字节(zerobytes, "\x00" )进行填充。
```

getrange

```
getrange key start end
返回键 key 储存的字符串值的指定部分， 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

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"
```

incr

```
incr key
为键 key 储存的数字值加上一
如果键 key 不存在， 那么它的值会先被初始化为 0 ， 然后再执行 INCR 命令。
如果键 key 储存的值不能被解释为数字， 那么 INCR 命令将返回一个错误。
```

incrby

```
incrby key increment
为键 key 储存的数字值加上增量 increment
如果键 key 不存在， 那么键 key 的值会先被初始化为 0 ， 然后再执行 INCRBY 命令。
如果键 key 储存的值不能被解释为数字， 那么 INCRBY 命令将返回一个错误
```

incrbyfloat

```
incrbyfloat key increment
为键 key 储存的值加上浮点数增量 increment 。
如果键 key 不存在， 那么 INCRBYFLOAT 会先将键 key 的值设为 0 ， 然后再执行加法操作。
如果命令执行成功， 那么键 key 的值会被更新为执行加法计算之后的新值， 并且新值会以字符串的形式返回给调用者。
```

decr

```
decr key
为键 key 储存的数字值减去一。
如果键 key 不存在， 那么键 key 的值会先被初始化为 0 ， 然后再执行 DECR 操作。
如果键 key 储存的值不能被解释为数字， 那么 DECR 命令将返回一个错误。
```

decrby

```
decrby key decrement
将键 key 储存的整数值减去减量 decrement 。
如果键 key 不存在， 那么键 key 的值会先被初始化为 0 ， 然后再执行 DECRBY 命令。
如果键 key 储存的值不能被解释为数字， 那么 DECRBY 命令将返回一个错误。
```

mset

```
mset key1 value1 key2 value2 key3 value3
同时为多个键设置值
如果某个给定键已经存在， 那么 MSET 将使用新值去覆盖旧值，
```

msetnx

```
msetnx key1 value1 key2 value2 key3 value3 ...

当且仅当所有给定键都不存在时， 为所有给定键设置值。
即使只有一个给定键已经存在， MSETNX 命令也会拒绝执行对所有键的设置操作。
MSETNX 是一个原子性(atomic)操作， 所有给定键要么就全部都被设置， 要么就全部都不设置， 不可能出现第三种状态。
```

mget

```
mget key1 key2 key3 ...
返回给定的一个或多个字符串键的值。
如果给定的字符串键里面， 有某个键不存在， 那么这个键的值将以特殊值 nil 表示。
```

exists

```
exists keys
检测 key 是否存在
```

哈希表

hset

```
hset hash field value

将哈希表 hash 中域 field 的值设置为 value 。
如果给定的哈希表并不存在， 那么一个新的哈希表将被创建并执行 HSET 操作。
如果域 field 已经存在于哈希表中， 那么它的旧值将被新值 value 覆盖。
redis> hset dog name Tom color white
"dog" : {"name" : "Tom" , "color" : "white" }
```

hsetnx

```
hsetnx hash field value
当且仅当域 field 尚未存在于哈希表的情况下， 将它的值设置为 value 。
如果给定域已经存在于哈希表当中， 那么命令将放弃执行设置操作。
如果哈希表 hash 不存在， 那么一个新的哈希表将被创建并执行 HSETNX 命令
```

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。
```

hget

```
hget hash field
返回哈希表中给定域的值。
redis> hget dog name
redis> hget dog color
```

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。
```

hgetall

```
hgetall key
返回哈希表 key 中，所有的域和值。
```

hexists

```
hexists hash field
检查给定域 field 是否存在于哈希表 hash 当中。
```

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。
```

hlen

```
hlen key
返回哈希表 key 中域的数量。
```

hstrlen

```
hsetlen key field
返回哈希表 key 中， 与给定域 field 相关联的值的字符串长度（string length）。
```

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。
```

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。
```

hkeys

```
hkeys key
返回哈希表 key 中的所有域。
redis > hkeys dog
1) "name"
2) "color"
3) "age"
```

hvals

```
hvals key
返回哈希表 key 中所有域的值。
redis > hvals dog
1) "Tom"
2) "white"
3) "6"
```

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： 比如说，对空列表 mylist 执行命令 LPUSH mylist a b c ，列表的值将是 c b a ，这等同于原子性地执行 LPUSH mylist a 、LPUSH mylist b 和 LPUSH mylist c 三个命令

lpush mylist a b c
```

lpushx

```
lpushx key value
条件 仅当 key 存在并且是一个列表

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 key value [value …] 命令相反，当 key 不存在时， LPUSHX 命令什么也不做。
```

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
```

rpushx

```
RPUSHX key value [value …]
条件 仅当 key 存在并且是一个列表

将一个或多个值 value 插入到列表 key 的表尾(最右边)。
```

lpop

```
LPOP key

移除并返回列表 key 的头元素。
```

rpop

```
RPOP key
移除并返回列表 key 的尾元素。
```

rpoplpush

```
rpoplpush list1 list2

将列表 list1 中的最后一个元素(尾元素)弹出，并返回给客户端。
将 list1 弹出的元素插入到列表 list2 ，作为 list2 列表的的头元素
list1=[a,b,c]
list2=[d,e,f]
redis > rpoplpush list1 list2
list1=[a,b]
list2=[c,d,e,f]
```

brpoplpush

```
BRPOPLPUSH source destination timeout
BRPOPLPUSH 是 RPOPLPUSH source destination 的阻塞版本，当给定列表 source 不为空时，BRPOPLPUSH 的表现和 RPOPLPUSH source destination 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 key value [value …] 或 RPUSH key value [value …] 命令为止。
```

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。
```

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
```

lindex

```
lindex key index
返回列表 key 中，下标为 index 的元素。
```

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
```

lset

```
lset key index value
将列表 key 下标为 index 的元素的值设置为 value 。
```

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

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
```

blpop

```
blpop key [key …] timeout

它是 LPOP key 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。
```

无序集合

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。
```

sismember

```
SISMEMBER key member
判断 member 元素是否在集合key中。
```

spop

```
SPOP key
移除并返回集合中的一个随机元素。
```

srandmember

srandmember key [count]
如果命令执行时，只提供了 key 参数，那么返回集合中的一个随机元素。

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。

smove

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

scard

scard key
返回集合 key 的基数(集合中元素的数量)。

smembers

smembers key
返回集合 key 中的所有成员。

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。 and

sinterstore

这个命令类似于 sinter key [key …] 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。

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。 or
不存在的 key 被视为空集。

sunionstore

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

sdiff

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。
前者有后者没有的元素
不存在的 key 被视为空集。

sdiffstore

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

有序集合

zadd

```
zadd key score member [[score member] [score member] …]

zadd set 数据 名字 数据 名字 .....
```

zrange

```
zrange rank 0 -1
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
```

zrevrange

```
zrevrange rank 0 -1
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(从大到小)来排序。

zrevrange rank 0 -1 withscores
打印人的同时打印数据
```

zlexcount

```
zlencount key min max

redis> ZADD myzset 0 a 0 b 0 c 0 d 0 e 0 f 0 g
(integer) 7
redis> ZLEXCOUNT myzset - +
(integer) 7
redis> ZLEXCOUNT myzset [b [f
(integer) 5
```

zincrby

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

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。
```

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 。
```

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 )的成员的数量。
```

zrangebyscore

```
zrangebyscore key min max
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。
```

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。
```

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。
```

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。
```

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。
```

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。
```

zrangebylex

```
zrangebylex key min max [LIMIT offset count]
当有序集合的所有成员都具有相同的分值时， 有序集合的元素会根据成员的字典序（lexicographical ordering）来进行排序， 而这个命令则可以返回给定的有序集合键 key 中， 值介于 min 和 max 之间的成员
```

zlexcount

```
zlexcount key min max

对于一个所有成员的分值都相同的有序集合键 key 来说， 这个命令会返回该集合中， 成员介于 min 和 max 范围内的元素数量。
```

zremrangebylex

```
zremrangebylex key min max
对于一个所有成员的分值都相同的有序集合键 key 来说， 这个命令会移除该集合中， 成员介于 min 和 max 范围内的所有元素。
```

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。

zinterstome

```
zinterstome 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 destination numkeys key [key …] [WEIGHTS weight [weight …]] [AGGREGATE SUM|MIN|MAX] 命令。

时间

`ttl`

```
ttl key
获得 key 的过期时间 单位是秒
```

`pttl`

```
pttl key
获得 key 的过期时间 单位是豪秒
```

doc

哈希表

```
dog
{
"name":"zjan",
"age":"18",
"color":"white"
}
```

列表

```
"name" : ["list","wangwu","maliou","yuansa"]
```

集合

```
"wzry" :  { "nig" , "aa" , "bb" , "af" , "dad" }
```