phpredis 是 redis 的 php 的一个扩展，效率是相当高有链表排序功能，对创建内存级的模块业务关系

以下是 redis 官方提供的命令使用技巧(摘取天上星)收集整理:

摘取天上星:一个热爱互联网艺术的人(happy.yin@qq.com)
phpredis 下载地址如下:

https://github.com/owlient/phpredis

Redis::__construct 构造函数

$redis = new Redis();

connect, open 链接 redis 服务
参数
host: string，服务地址
port: int,端口号
timeout: float,链接时长 (可选, 默认为 0 ，不限链接时间)
注: 在 redis.conf 中也有时间，默认为 300

pconnect, popen 不会主动关闭的链接
参考上面

setOption 设置 redis 模式

getOption 查看 redis 设置的模式

ping 查看连接状态

KEY 相关操作

DEL

移除给定的一个或多个 key。

如果 key 不存在，则忽略该命令。

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。

```
//DEL
# 情况 1: 删除单个 key
$redis->set('myname','ikodota');
echo $redis->get('myname').'<br>'; # 返回:ikodota
```

```
$redis->del('myname');# 返回 TRUE(1)
var_dump($redis->get('myname')); # 返回 bool(false)

# 情况 2： 删除一个不存在的 key
if(!$redis->exists('fake_key')) # 不存在
var_dump($redis->del('fake_key')); # 返回 int(0)

# 情况 3： 同时删除多个 key
$array_mset=array('first_key'=>'first_val',
        'second_key'=>'second_val',
        'third_key'=>'third_val');
$redis->mset($array_mset); #用 MSET 一次储存多个值
$array_mget=array('first_key','second_key','third_key');
var_dump($redis->mget($array_mget)); #一次返回多个值 //array(3) { [0]=> string(9)
"first_val" [1]=> string(10) "second_val" [2]=> string(9) "third_val" }

$redis->del($array_mget); #同时删除多个 key
var_dump($redis->mget($array_mget)); #返回 array(3) { [0]=> bool(false) [1]=>
bool(false) [2]=> bool(false) }
```

KEYS

KEYS pattern
查找符合给定模式的 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

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)*。

```
//KEYS
#$redis->FLUSHALL();
$array_mset_keys=array('one'=>'1',
        'two'=>'2',
        'three '=>'3',
        'four'=>'4');
$redis->mset($array_mset_keys); #用 MSET 一次储存多个值
var_dump($redis->keys('*o*')); //array(3) { [0]=> string(4) "four" [1]=>
```

```
string(3) "two" [2]=> string(3) "one" }
var_dump($redis->keys('t??')); //array(1) { [0]=> string(3) "two" }
var_dump($redis->keys('t[w]*')); //array(1) { [0]=> string(3) "two" }
print_r($redis->keys('*')); //Array ( [0] => four [1] => three [2] => two [3] =>
one )
```

RANDOMKEY

从当前数据库中随机返回(不删除)一个 key。

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。

```
//RANDOMKEY
$redis->FLUSHALL();
# 情况 1：数据库不为空
$array_mset_randomkey=array('fruit'=>'apple',
              'drink'=>'beer',
              'food'=>'cookis');
$redis->mset($array_mset_randomkey);
echo $redis->randomkey();
print_r($redis->keys('*')); # 查看数据库内所有 key，证明 RANDOMKEY 并不删除 key//Array
( [0] => food [1] => drink [2] => fruit )

# 情况 2：数据库为空
$redis->flushdb();  # 删除当前数据库所有 key
var_dump($redis-> randomkey()); //bool(false)
```

TTL

TTL key

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

时间复杂度：

O(1)

返回值：

key 的剩余生存时间(以秒为单位)。

当 key 不存在或没有设置生存时间时，返回 -1 。

```
//TTL
```

```php
# 情况1：带TTL的key
$redis->flushdb();
//$redis->set('name','ikodota'); # 设置一个key
$redis->expire('name',30);  # 设置生存时间为30秒 //return (integer) 1
echo $redis->get('name'); //return ikodota
echo $redis->ttl('name'); //(integer) 25

//echo $redis->ttl('name');  # 30秒过去，name过期 //(integer) -1
var_dump($redis->get('name')); # 过期的key将被删除 //return bool(false);

# 情况2：不带TTL的key
$redis->set('site','wikipedia.org');//OK
var_dump($redis->ttl('site'));//int(-1)

# 情况3：不存在的key
$redis->EXISTS('not_exists_key');//int(0)
var_dump($redis->TTL('not_exists_key'));//int(-1)
```

EXISTS

EXISTS key

检查给定 key 是否存在。

时间复杂度：

O(1)

返回值：

若 key 存在，返回 1，否则返回 0。

```php
//EXISTS
echo '<br>EXISTS<br>';
$redis->set('db',"redis"); //bool(true)
var_dump($redis->exists('db'));  # key存在 //bool(true)
$redis->del('db');   # 删除key //int(1)
var_dump($redis->exists('db'))  # key不存在 //bool(false)
```

MOVE

MOVE key db

将当前数据库(默认为 0)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。

时间复杂度：

O(1)

返回值：

移动成功返回 1，失败则返回 0。

```
//MOVE
echo '<br><br>MOVE<br>';
# 情况1： 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
$redis->SET('song',"secret base - Zone"); //OK
var_dump ($redis->MOVE('song',1));  # 将song移动到数据库1 //bool(true)

# 情况2：当key不存在的时候
$redis->SELECT(1);
var_dump ($redis->EXISTS('fake_key'));//bool(false);
var_dump($redis->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) //bool(false)

$redis->SELECT(0); # 使用数据库0
var_dump($redis->EXISTS('fake_key'));  # 证实fake_key不存在 //bool(false)

# 情况3：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
$redis->SET('favorite_fruit',"banana");

$redis->SELECT(1);  # 使用数据库1
$redis->SET('favorite_fruit',"apple");

$redis->SELECT(0);  # 使用数据库0，并试图将favorite_fruit移动到数据库1
var_dump($redis->MOVE('favorite_fruit',1));  # 因为两个数据库有相同的key，MOVE失败 //return bool(false)
echo $redis->GET('favorite_fruit');  # 数据库0的favorite_fruit没变 //return banana

$redis->SELECT(1);
echo $redis->GET('favorite_fruit');   # 数据库1的favorite_fruit也是 //return 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。

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。

```php
//RENAME
echo '<br><br>RENAME<br>';
# 情况1：key存在且newkey不存在
$redis->SET('message',"hello world");
var_dump($redis->RENAME('message','greeting'));  //bool(true)
var_dump($redis->EXISTS('message'));  # message不复存在 //bool(false)
var_dump($redis->EXISTS('greeting'));   # greeting取而代之 //bool(true)

# 情况2：当key不存在时，返回错误 ,php返回false;
var_dump($redis->RENAME('fake_key','never_exists'));  //bool(false)

# 情况3：newkey已存在时，RENAME会覆盖旧newkey
$redis->SET('pc',"lenovo");
$redis->SET('personal_computer',"dell");
var_dump($redis->RENAME('pc','personal_computer')); //bool(true)
var_dump($redis->GET('pc')); //(nil)   bool(false)
var_dump($redis->GET('personal_computer'));  # dell“没有”了 //string(6) "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)。

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 。

```php
//RENAMENX
echo '<br><br>RENAMENX<br>';

# 情况1：newkey不存在，成功
$redis->SET('player',"MPlyaer");
$redis->EXISTS('best_player'); //int(0)
var_dump($redis->RENAMENX('player','best_player')); // bool(true)

# 情况2：newkey存在时，失败
$redis->SET('animal',"bear");
$redis->SET('favorite_animal', "butterfly");

var_dump($redis->RENAMENX('animal', 'favorite_animal'));// bool(false)

var_dump($redis->get('animal')); //string(4) "bear"
var_dump($redis->get('favorite_animal')); //string(9) "butterfly"
```

TYPE

TYPE key

返回 key 所储存的值的类型。

时间复杂度：

O(1)

返回值：

none(key 不存在) int(0)
string(字符串) int(1)
list(列表) int(3)
set(集合) int(2)
zset(有序集) int(4)
hash(哈希表) int(5)

```
//TYPE
$redis->flushALL();
echo '<br><br>TYPE<br>';

var_dump($redis->TYPE('fake_key')); //none /int(0)

$redis->SET('weather',"sunny");  # 构建一个字符串
var_dump($redis->TYPE('weather'));//string / int(1)

$redis->SADD('pat',"dog");  # 构建一个集合
var_dump($redis->TYPE('pat')); //set /int(2)

$redis->LPUSH('book_list',"programming in scala");  # 构建一个列表
var_dump($redis->TYPE('book_list'));//list / int(3)

$redis->ZADD('pats',1,'cat');  # 构建一个 zset (sorted set) // int(1)
$redis->ZADD('pats',2,'dog');
$redis->ZADD('pats',3,'pig');
var_dump($redis->zRange('pats',0,-1)); // array(3) { [0]=> string(3) "cat" [1]=>
string(3) "dog" [2]=> string(3) "pig" }
var_dump($redis->TYPE('pats')); //zset / int(4)

$redis->HSET('website','google','www.g.cn');   # 一个新域
var_dump($redis->HGET('website','google')); //string(8) "www.g.cn"
var_dump($redis->TYPE('website')); //hash /int(5)
```

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“易失的”(volatile)。

在低于 2.1.3 版本的 Redis 中，已存在的生存时间不可覆盖。
从 2.1.3 版本开始，key 的生存时间可以被更新，也可以被 PERSIST 命令移除。(详情参见 http://redis.io/topics/expire)。

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。

```
//EXPIRE
$redis->select(7);
//$redis->flushdb();

echo '<br><br>EXPIRE<br>';
$redis->SET('cache_page',"www.cnblogs.com/ikodota");
$redis->EXPIRE('cache_page', 30);  # 设置 30 秒后过期
sleep(6);
echo $redis->TTL('cache_page').'<br>';   # 查看给定 key 的剩余生存时间 //(integer) 24

$redis->EXPIRE('cache_page', 3000);  # 更新生存时间，3000 秒
sleep(4);
echo $redis->TTL('cache_page').'<br>';   //(integer) 2996
```

EXPIREAT

EXPIREAT key timestamp

EXPIREAT 的作用和 EXPIRE 一样，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)。

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。

```
//EXPIREAT
echo '<br><br>EXPIREAT<br>';
$redis->SET('cache','www.google.com');
echo $redis->EXPIREAT('cache','1355292000'); # 这个 key 将在 2012.12.12 过期

echo ($redis->TTL('cache')); //return 124345085
```

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)。

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。

```
//OBJECT
$redis->select(8);
echo '<br><br>OBJECT<br>';
$redis->SET('game',"WOW");  # 设置一个字符串
$redis->OBJECT('REFCOUNT','game');  # 只有一个引用

//sleep(5);
echo $redis->OBJECT('IDLETIME','game');  # 等待一阵。。。然后查看空转时间 //(integer)
```

```
10
//echo $redis->GET('game');  # 提取game， 让它处于活跃(active)状态  //return WOW
//echo $redis->OBJECT('IDLETIME','game');  # 不再处于空转 //(integer) 0
var_dump($redis->OBJECT('ENCODING','game'));  # 字符串的编码方式 //string(3) "raw"
$redis->SET('phone',15820123123);  # 大的数字也被编码为字符串
var_dump($redis->OBJECT('ENCODING','phone')); //string(3) "raw"
$redis->SET('age',20);  # 短数字被编码为int
var_dump($redis->OBJECT('ENCODING','age')); //string(3) "int"
```

PERSIST

PERSIST key

移除给定 key 的生存时间。

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。

```
//PERSIST
echo '<br><br>PERSIST<br>';
$redis->SET('time_to_say_goodbye',"886...");
$redis->EXPIRE('time_to_say_goodbye', 300);
sleep(3);
echo $redis->TTL('time_to_say_goodbye'); # (int) 297
echo '<br>';

$redis->PERSIST('time_to_say_goodbye');  # 移除生存时间
echo $redis->TTL('time_to_say_goodbye');  # 移除成功  //int(-1)
```

SORT

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

排序，分页等

参数

array(
‘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);
$redis->LPUSH('today_cost', 1.5);
$redis->LPUSH('today_cost', 10);
$redis->LPUSH('today_cost', 8);
# 排序
var_dump($redis->SORT('today_cost')); //array(4) { [0]=> string(3) "1.5" [1]=>
string(1) "8" [2]=> string(2) "10" [3]=> string(2) "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");
$redis->LPUSH('website', "www.slashdot.com");
$redis->LPUSH('website', "www.infoq.com");
# 默认排序
var_dump($redis->SORT('website'));//array(3) { [0]=> string(13) "www.infoq.com"
[1]=> string(16) "www.slashdot.com" [2]=> string(14) "www.reddit.com" }

# 按字符排序 ALPHA=true
var_dump($redis->SORT('website', array('ALPHA'=>TRUE))); //array(3) { [0]=>
string(13) "www.infoq.com" [1]=> string(14) "www.reddit.com" [2]=> string(16)
"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
$redis_sort_option=array('LIMIT'=>array(0,5));
var_dump($redis->SORT('rank',$redis_sort_option));   # 返回排名前五的元素 //
array(5) { [0]=> string(1) "0" [1]=> string(2) "11" [2]=> string(2) "22" [3]=>
string(2) "30" [4]=> string(2) "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。

```
$redis_sort_option=array(
          'LIMIT'=>array(0,5),
          'SORT'=>'DESC'
          );
var_dump($redis->SORT('rank',$redis_sort_option)); //array(5) { [0]=> string(2)
"67" [1]=> string(2) "56" [2]=> string(2) "55" [3]=> string(2) "50" [4]=>
string(2) "44" }
```

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
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');
$redis->SET('user_level_1',9999);

# huangz
$redis->LPUSH('user_id', 2);//(integer) 2
$redis->SET('user_name_2', 'huangz');
$redis->SET('user_level_2', 10);

# jack
$redis->LPUSH('user_id', 59230);//(integer) 3
$redis->SET('user_name_59230','jack');
$redis->SET('user_level_59230', 3);

# hacker
$redis->LPUSH('user_id', 222);  //(integer) 4
$redis->SET('user_name_222', 'hacker');
$redis->SET('user_level_222', 9999);
```

如果希望按 `level` 从大到小排序 `user_id`，可以使用以下命令：

```
$redis_sort_option=array('BY'=>'user_level_*',
          'SORT'=>'DESC'
          );
var_dump($redis->SORT('user_id',$redis_sort_option)); //array(4) { [0]=>
string(3) "222" [1]=> string(1) "1" [2]=> string(1) "2" [3]=> string(5)
"59230" }

#---------------------------
#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：

```
$redis_sort_option=array('BY'=>'user_level_*',
          'SORT'=>'DESC',
          'GET'=>'user_name_*'
          );
var_dump($redis->SORT('user_id', $redis_sort_option)); //array(4) { [0]=>
string(6) "hacker" [1]=> string(5) "admin" [2]=> string(6) "huangz" [3]=>
string(4) "jack" }

#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");
$redis->SET('user_password_1', "a_long_long_password");
$redis->SET('user_password_2', "nobodyknows");
$redis->SET('user_password_59230', "jack201022");

# 获取 name 和 password
$redis_sort_option=array('BY'=>'user_level_*',
          'SORT'=>'DESC',
          'GET'=>array('user_name_*','user_password_*')
          );
var_dump($redis->SORT('user_id',$redis_sort_option));//array(8) { [0]=>
string(6) "hacker" [1]=> string(9) "hey,im in" [2]=> string(5) "admin" [3]=>
string(20) "a_long_long_password" [4]=> string(6) "huangz" [5]=> string(11)
"nobodyknows" [6]=> string(4) "jack" [7]=> string(10) "jack201022" }

#-------------------------------------
#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

```
# 获取 name 和 password 注意 GET 操作是有序的
$redis_sort_option=array('BY'=>'user_level_*',
          'SORT'=>'DESC',
          'GET'=>array('user_password_*','user_name_*')
          );
var_dump($redis->SORT('user_id',$redis_sort_option));// array(8) { [0]=>
```

```
string(9) "hey,im in" [1]=> string(6) "hacker" [2]=> string(20)
"a_long_long_password" [3]=> string(5) "admin" [4]=> string(11) "nobodyknows"
[5]=> string(6) "huangz" [6]=> string(10) "jack201022" [7]=> string(4) "jack" }
```

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：

```
$redis_sort_option=array('BY'=>'user_level_*',
          'SORT'=>'DESC',
          'GET'=>array('#','user_password_*','user_name_*')
          );
var_dump($redis->SORT('user_id',$redis_sort_option));//array(12) { [0]=>
string(3) "222" [1]=> string(9) "hey,im in" [2]=> string(6) "hacker" [3]=>
string(1) "1" [4]=> string(20) "a_long_long_password" [5]=> string(5) "admin"
[6]=> string(1) "2" [7]=> string(11) "nobodyknows" [8]=> string(6) "huangz"
[9]=> string(5) "59230" [10]=> string(10) "jack201022" [11]=> string(4) "jack" }

#------------------------------------------------------------
#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_option=array('BY'=>'fake_key',
          'SORT'=>'DESC',
```

```
          'GET'=>array('#','user_name_*','user_password_*')
          );
var_dump($redis->SORT('user_id',$redis_sort_option));//array(12) { [0]=>
string(3) "222" [1]=> string(6) "hacker" [2]=> string(9) "hey,im in" [3]=>
string(5) "59230" [4]=> string(4) "jack" [5]=> string(10) "jack201022" [6]=>
string(1) "2" [7]=> string(6) "huangz" [8]=> string(11) "nobodyknows" [9]=>
string(1) "1" [10]=> string(5) "admin" [11]=> string(20)
"a_long_long_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_option=array('BY'=>'user_level_*',
          'GET'=>array('#','user_name_*','user_password_*'),
          'STORE'=>'user_info_sorted_by_level'
          );

var_dump($redis->SORT('user_id',$redis_sort_option)); //int(12)
var_dump($redis->LRANGE('user_info_sorted_by_level', 0 ,11));  # 查看排序结果
//array(12) { [0]=> string(5) "59230" [1]=> string(4) "jack" [2]=> string(10)
"jack201022" [3]=> string(1) "2" [4]=> string(6) "huangz" [5]=> string(11)
"nobodyknows" [6]=> string(3) "222" [7]=> string(6) "hacker" [8]=> string(9)
"hey,im in" [9]=> string(1) "1" [10]=> string(5) "admin" [11]=> string(20)
"a_long_long_password" }

#-------------------------------------------------------------------
#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_hash_testdata_array=array(1=>'23131283',
             2=>'23810573',
             222=>'502342349',
             59230=>'2435829758'
             );

$redis->HMSET('serial',$redis_hash_testdata_array);

# 我们希望以比较 serial 中的大小来作为排序 user_id 的方式
$redis_sort_option=array('BY'=>'*->serial');
var_dump($redis->SORT('user_id', $redis_sort_option)); //array(4) { [0]=>
string(3) "222" [1]=> string(5) "59230" [2]=> string(1) "2" [3]=> string(1)
"1" }

#---------------------------------------
#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。

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)

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。

时间复杂度：O(1)**返回值：**总是返回 OK(TRUE)，因为 SET 不可能失败。

```
# 情况 1：对字符串类型的 key 进行 SET
$redis->SET('apple', 'www.apple.com');#OK  //bool(true)
$redis->GET('apple');//"www.apple.com"

# 情况 2：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 //int(1)
$redis->TYPE('greet_list');#list //int(3)

$redis->SET('greet_list', "yooooooooooooooo");    # 覆盖列表类型 #OK //bool(true)
$redis->TYPE('greet_list');#string //int(1)
```

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。

时间复杂度：
O(1)
返回值：
设置成功，返回 1。
设置失败，返回 0。

```
//SETNX
```

```
echo '<br><br>SETNX<br>';
$redis->EXISTS('job');  # job不存在 //bool(false);
$redis->SETNX('job', "programmer");  # job设置成功 //bool(true)
$redis->SETNX('job', "code-farmer");  # job设置失败 //bool(false)
echo $redis->GET('job');  # 没有被覆盖 //"programmer"
```

设计模式(Design pattern): 将 SETNX 用于加锁(locking)

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。
(这里是不是有点问题? C4 的确是可以重试，但 C5 怎么办? 它的锁的过期被 C4 修改了。——译注)

> **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：

```
$redis->SET('key', 'value');
$redis->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。

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。

```
# 情况 1：key 不存在
$redis->SETEX('cache_user_id', 60,10086);//bool(true)
echo $redis->GET('cache_user_id');  # 值 //"10086"

sleep(4);
echo $redis->TTL('cache_user_id');  # 剩余生存时间 //int(56)

# 情况 2：key 已经存在，key 被覆写
$redis->SET('cd', "timeless"); //bool(true);
$redis->SETEX('cd', 3000,"goodbye my love"); //bool(true);
echo $redis->GET('cd');//"goodbye my love"
```

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 的字符串被限制在 512 兆(megabytes)内。如果你需要使用比这更大的空间，你得使用多个 key。

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。

> **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。

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。

```
# 情况 1：对非空字符串进行 SETRANGE
$redis->SET('greeting', "hello world");
$redis->SETRANGE('greeting', 6, "Redis"); //int(11)
$redis->GET('greeting');//"hello Redis"

# 情况 2：对空字符串/不存在的 key 进行 SETRANGE
$redis->EXISTS('empty_string');//bool(false)
$redis->SETRANGE('empty_string', 5 ,"Redis!");  # 对不存在的 key 使用 SETRANGE
//int(11)
var_dump($redis->GET('empty_string'));  # 空白处被"\x00"填充
#"\x00\x00\x00\x00\x00Redis!"   //return string(11) "Redis!"
```

MSET

MSET key value [key value ...]

同时设置一个或多个 key-value 对。

当发现同名的 key 存在时，MSET 会用新值覆盖旧值，如果你不希望覆盖同名 key，请使用 MSETNX 命令。

MSET 是一个原子性(atomic)操作，所有给定 key 都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。

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)

```
#MSET
echo '<br><br>MSET<br>';
$redis->select(0);
$redis->flushdb();
$array_mset=array('date'=>'2012.3.5',
       'time'=>'9.09a.m.',
       'weather'=>'sunny'
       );
$redis->MSET($array_mset); //bool(true)

var_dump($redis->KEYS('*'));   # 确保指定的三个 key-value 对被插入 //array(3) { [0]=>
string(4) "time" [1]=> string(7) "weather" [2]=> string(4) "date" }

# MSET 覆盖旧值的例子 但是经过测试覆盖不了
var_dump($redis->SET('google', "google.cn"));   //bool(true)
var_dump($redis->MSET('google',"google.hk")); //bool(false)
echo $redis->GET('google'); //google.cn  与 redis 手册的示例结果不符
```

MSETNX

MSETNX key value [key value ...]

同时设置一个或多个 key-value 对，当且仅当 key 不存在。

即使*只有一个* key 已存在，MSETNX 也会拒绝*所有*传入 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。

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 key 都设置失败(最少有一个 key 已经存在)，那么返回 0。

```
# 情况 1：对不存在的 key 进行 MSETNX
$array_mset=array('rmdbs'=>'MySQL',
```

```
        'nosql'=>'MongoDB',
        'key-value-store'=>'redis'
        );
$redis->MSETNX($array_mset);//bool(true)


# 情况 2：对已存在的 key 进行 MSETNX
$array_mset=array('rmdbs'=>'Sqlite',
        'language'=>'python'
        );
var_dump($redis->MSETNX($array_mset));  # rmdbs 键已经存在，操作失败 //bool(false)
var_dump($redis->EXISTS('language'));  # 因为操作是原子性的，language 没有被设置
bool(false)

echo $redis->GET('rmdbs');  # rmdbs 没有被修改 //"MySQL"

$array_mset_keys=array( 'rmdbs', 'nosql', 'key-value-store');
print_r($redis->MGET($array_mset_keys)); //Array ( [0] => MySQL [1] => MongoDB
[2] => redis )
```

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。

时间复杂度：

平摊复杂度 O(1)

返回值：

追加 value 之后，key 中字符串的长度。

```
# 情况 1：对不存在的 key 执行 APPEND

$redis->EXISTS('myphone');  # 确保 myphone 不存在 //bool(false)
$redis->APPEND('myphone',"nokia");  # 对不存在的 key 进行 APPEND，等同于 SET myphone
"nokia" //int(5) # 字符长度

# 情况 2：对字符串进行 APPEND
$redis->APPEND('myphone', " - 1110");# 长度从 5 个字符增加到 12 个字符 //int(12)

echo $redis->GET('myphone');  # 查看整个字符串 //"nokia - 1110"
```

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。

时间复杂度:

O(1)

返回值:

key 的值。

如果 key 不存在，返回 nil。

```
//GET
var_dump($redis->GET('fake_key')); #(nil) //return bool(false)
$redis->SET('animate', "anohana"); //return bool(true)
var_dump($redis->GET('animate')); //return string(7) "anohana"
```

MGET

MGET key [key ...]

返回所有(一个或多个)给定 key 的值。

如果某个指定 key 不存在，那么返回特殊值 nil。因此，该命令永不失败。

时间复杂度:

O(1)

返回值:

一个包含所有给定 key 的值的列表。

```
//MGET
echo '<br><br>MGET<br>';
$redis_mget_data_array=array('name'=>'ikodota','blog'=>'cnblogs.com/ikodota');
$redis->MSET($redis_mget_data_array);#用 MSET 一次储存多个值

$redis_mget_key_array=array('name','blog');
var_dump($redis->MGET($redis_mget_key_array)); //array(2) { [0]=> string(7)
"ikodota" [1]=> string(19) "cnblogs.com/ikodota" }

$redis->EXISTS('fake_key'); //bool(false)

$redis_mget_key_array=array('name','fake_key');
var_dump($redis->MGET($redis_mget_key_array));  # 当 MGET 中有不存在 key 的情况
//array(2) { [0]=> string(7) "ikodota" [1]=> bool(false) }
```

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。

时间复杂度：

O(N)，N 为要返回的字符串的长度。

复杂度最终由返回值长度决定，但因为从已有字符串中建立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。

注解:在<=2.0 的版本里，GETRANGE 被叫作 SUBSTR。

```
//GETRANGE
echo '<br><br>GETRANGE<br>';
$redis->SET('greeting', "hello, my friend");
echo $redis->GETRANGE('greeting', 0, 4).'<br>';  # 返回索引 0-4 的字符，包括 4。
//"hello"
echo $redis->GETRANGE('greeting', -1 ,-5).'<br>';  # 不支持回绕操作  //""
echo $redis->GETRANGE('greeting', -3 ,-1).'<br>';  # 负数索引 //"end"
echo $redis->GETRANGE('greeting', 0, -1).'<br>';  # 从第一个到最后一个 //"hello, my
friend"
echo $redis->GETRANGE('greeting', 0, 1008611).'<br>';  # 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。

当 key 存在但不是字符串类型时，返回一个错误。

时间复杂度：

O(1)

返回值:

返回给定 key 的旧值(old value)。

当 key 没有旧值时，返回 nil。

```
//GETSET
echo '<br><br>GETSET<br>';
var_dump($redis->EXISTS('mail'));//return bool(false);
var_dump($redis->GETSET('mail','xxx@google.com'));  # 因为 mail 之前不存在，没有旧值，返回 nil ,#(nil)   //bool(false)

var_dump($redis->GETSET('mail','xxx@yahoo.com'));  # mail 被更新，旧值被返回
//string(14) "xxx@google.com"
```

设计模式

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->SELECT(2);
echo $redis->INCR('mycount').'<br>'; #(integer) 11

if($redis->GET('mycount')>19){
    echo $redis->GETSET('mycount', 0).'<br>';  # 一个原子内完成 GET mycount 和 SET mycount 0 操作 #"11"
}
echo $redis->GET('mycount'); #"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。

复杂度:

O(1)

返回值:

字符串值的长度。

当 key 不存在时，返回 0。

```
$redis->SET('mykey', "Hello world");
echo $redis->STRLEN('mykey'); //int(11)
echo $redis->STRLEN('nonexisting'); # 不存在的 key 长度视为 0  //int(0)
```

INCR

`INCR key`

将 `key` 中储存的数字值增一。

如果 `key` 不存在，以 `0` 为 `key` 的初始值，然后执行 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。

时间复杂度：

O(1)

返回值：

执行 INCR 命令之后 `key` 的值。

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。

```
$redis->SET('page_view', 20);
var_dump($redis->INCR('page_view')); //int(21)
var_dump($redis->GET('page_view'));    # 数字值在 Redis 中以字符串的形式保存
//string(2) "21
```

INCRBY

`INCRBY key increment`

将 `key` 所储存的值加上增量 `increment`。

如果 `key` 不存在，以 `0` 为 `key` 的初始值，然后执行 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。

关于更多递增(increment)/递减(decrement)操作信息，参见 INCR 命令。

时间复杂度：

O(1)

返回值：

加上 `increment` 之后，`key` 的值。

```
//INCRBY
echo '<br><br>INCRBY<br>';
# 情况 1：key 存在且是数字值
$redis->SET('rank', 50);  # 设置 rank 为 50
```

```
$redis->INCRBY('rank', 20);  # 给rank加上20
var_dump($redis->GET('rank')); #"70"   //string(2) "70"

# 情况2：key不存在
$redis->EXISTS('counter'); //bool(false)
$redis->INCRBY('counter'); #int 30  //bool(false)
var_dump($redis->GET('counter')); #30 //经测试 与手册上结果不一样，不能直接从bool型转为int型。 return bool(false)

# 情况3：key不是数字值
$redis->SET('book', "long long ago...");
var_dump($redis->INCRBY('book', 200)); #(error) ERR value is not an integer or out of range   // bool(false)
```

DECR

DECR key

将 key 中储存的数字值减一。

如果 key 不存在，以 0 为 key 的初始值，然后执行 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。

关于更多递增(increment)/递减(decrement)操作信息，参见 INCR 命令。

时间复杂度：

O(1)

返回值：

执行 DECR 命令之后 key 的值。

```
//DECR
$redis->SELECT(3);
$redis->flushdb();
echo '<br><br>DECR<br>';
# 情况1：对存在的数字值key进行DECR
$redis->SET('failure_times', 10);
$redis->DECR('failure_times'); //int(9)
echo $redis->GET('failure_times').'<br>';  //string(1) "9"

# 情况2：对不存在的key值进行DECR
$redis->EXISTS('count'); #(integer) 0 //bool(false)
$redis->DECR('count');  //int(-1)
echo $redis->GET('count').'<br>'; //string(2) "-1"

# 情况3：对存在但不是数值的key进行DECR
```

```php
$redis->SET('company', 'YOUR_CODE_SUCKS.LLC');
var_dump($redis->DECR('company')); #(error) ERR value is not an integer or out
of range   //bool(false)
echo $redis->GET('company').'<br>'; //YOUR_CODE_SUCKS.LLC
```

DECRBY

DECRBY key decrement

将 key 所储存的值减去减量 decrement。

如果 key 不存在，以 0 为 key 的初始值，然后执行 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)操作信息，参见 INCR 命令。

时间复杂度：

O(1)

返回值：

减去 decrement 之后，key 的值。

```php
# 情况 1：对存在的数值 key 进行 DECRBY
$redis->SET('count', 100);
var_dump($redis->DECRBY('count', 20)); //int(80)
var_dump($redis->GET('count'));  //string(2) "80"

# 情况 2：对不存在的 key 进行 DECRBY
$redis->EXISTS('pages');#(integer) 0  //bool(false)
var_dump($redis->DECRBY('pages', 10));  //int(-10)
var_dump($redis->GET('pages')); //string(3) "-10"
```

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。

字符串会增长(grown)以确保它可以将 `value` 保存在指定的偏移量上。当字符串值增长时，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 512MB 内)。

时间复杂度:

O(1)

返回值:

指定偏移量原来储存的位（"0"或"1"）.

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。

```
//SETBIT
echo '<br><br>SETBIT<br>';
$bit_val=67;
echo decbin($bit_val).'<br>'; //1000011
var_dump($redis->SETBIT('bit',1,1));//int(0)  空位上都是0
var_dump($redis->SETBIT('bit',2,0));//int(0)
var_dump($redis->SETBIT('bit',3,0));//int(0)
var_dump($redis->SETBIT('bit',4,0));//int(0)
var_dump($redis->SETBIT('bit',5,0));//int(0)
var_dump($redis->SETBIT('bit',6,1));//int(0)
var_dump($redis->SETBIT('bit',7,1));//int(0)

var_dump($redis->GET('bit')); //string(1) "C" ,二进制为：1000011 ,ASCII:67

var_dump($redis->GETBIT('bit', 6 )); //int(1)  取出第6位（从左到右）为“1”

var_dump($redis->SETBIT('bit',5,1));//int(0)  把第5位的0改为1
var_dump($redis->SETBIT('bit',6,0));//int(1)  把第6位的1改为0

var_dump($redis->GET('bit')); //string(1) "E ,二进制为：1000101,ASCII:69l
```

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`。

时间复杂度:

O(1)

返回值:

字符串值指定偏移量上的位(bit)。

#参见 SETBIT 的示例

哈希表(Hash)

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。

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。

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。

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。

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。

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。

HGET

```
HGET key field
```

返回哈希表 key 中给定域 field 的值。

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。

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。

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。

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。

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。

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。

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。

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。

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。

HEXISTS

```
HEXISTS key field
```

查看哈希表 key 中，给定域 field 是否存在。

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`。

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。

本操作的值限制在 64 位(bit)有符号数字表示之内。

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。

HKEYS

`HKEYS key`

返回哈希表 `key` 中的所有域。

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。

HVALS

`HVALS key`

返回哈希表 `key` 中的所有值。

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。

表(List)

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。

时间复杂度：

O(1)

返回值：

执行 LPUSH 命令后，列表的长度。

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

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。

时间复杂度：
O(1)
返回值：
LPUSHX 命令执行之后，表的长度。

RPUSH

`RPUSH key value [value ...]`

将一个或多个值 `value` 插入到列表 `key` 的*表尾*。

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`，则结果列表为 `a b c`，等同于执行命令 `RPUSHmylist 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。

时间复杂度：
O(1)
返回值：
执行 RPUSH 操作后，表的长度。

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

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。

时间复杂度：
O(1)
返回值：
RPUSHX 命令执行之后，表的长度。

LPOP

`LPOP key`

移除并返回列表 `key` 的头元素。

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。

RPOP

```
RPOP key
```

移除并返回列表 key 的尾元素。

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。

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。

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) 。

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。

时间复杂度：O(1)返回值：

如果列表为空，返回一个 nil。
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。

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。

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。

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。

时间复杂度：

O(1)

返回值：

列表 key 的长度。

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。

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。

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。

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`。

LSET

`LSET key index value`

将列表 `key` 下标为 `index` 的元素的值甚至为 `value`。

更多信息请参考 LINDEX 操作。

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。

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

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：

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。

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。

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。

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。

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。

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`。

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` 被返回。

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。

时间复杂度：

O(1)

返回值：

被弹出的元素。

设计模式： 一个安全的队列

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)。一个程序(称之为生产者，producer)通过 LPUSH 命令将消息放入队列中，而另一个程序(称之为消费者，consumer)通

过 RPOP 命令取出队列中等待时间最长的消息。

不幸的是，在这个过程中，一个消费者可能在获得一个消息之后崩溃，而未执行完成的消息也因此丢失。

使用 RPOPLPUSH 命令可以解决这个问题，因为它在返回一个消息之余，还将该消息添加到另一个列表当中，另外的这个列表可以用作消息的备份表：假如一切正常，当消费者完成该消息的处理之后，可以用 LREM 命令将该消息从备份表删除。

另一方面，助手(helper)程序可以通过监视备份表，将超过一定处理时限的消息重新放入队列中去(负责处理该消息的消费者可能已经崩溃)，这样就不会丢失任何消息了。

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。

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。

集合(Set)

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。

set

描述:
设置关键值参数
参数：Key Value
返回值：BOOL
范例:
$redis->set('key', 'value')

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。

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。

注解:在 Redis2.4 版本以前，SADD 只接受单个 `member` 值。

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。

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。

注解:在 Redis2.4 版本以前，SREM 只接受单个 `member` 值。

SMEMBERS

SMEMBERS key

返回集合 key 中的所有成员。

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。

SISMEMBER

SISMEMBER key member

判断 member 元素是否是集合 key 的成员。

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。

SCARD

SCARD key

返回集合 key 的**基数**(集合中元素的数量)。

时间复杂度:

O(1)

返回值:

集合的基数。

当 key 不存在时，返回 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。

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。

SPOP

SPOP key

移除并返回集合中的一个随机元素。

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。

也可以参考: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。

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。

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`。

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)。

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。

SINTERSTORE

```
SINTERSTORE destination key [key ...]
```

此命令等同于 SINTER，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。

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。

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。

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。

SUNIONSTORE

SUNIONSTORE destination key [key ...]

此命令等同于 SUNION，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。

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。

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的*差集* 。

不存在的 key 被视为空集。

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。

SDIFFSTORE

```
SDIFFSTORE destination key [key ...]
```

此命令等同于 SDIFF，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。

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。

有序集(Sorted Set)

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。

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。

注解:在 Redis2.4 版本以前，ZADD 每次只能添加一个元素。

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。

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。

注解：在 Redis2.4 版本以前，ZREM 每次只能删除一个元素。

ZCARD

ZCARD key

返回有序集 key 的基数。

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。

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。

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。

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。

时间复杂度:

O(1)

返回值:

member 成员的 score 值，以字符串形式表示。

ZINCRBY

ZINCRBY key increment member

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

你也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。

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。

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。

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。

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。

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。

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：

返回所有符合条件 1 < score <= 5 的成员；

返回所有符合条件 5 < score < 10 的成员。

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。

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，ZREVRANGEBYSCORE 命令的其他方面和 ZRANGEBYSCORE 命令一样。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

发布/订阅(Pub/Sub)

PUBLISH

```
PUBLISH channel message
```

将信息 message 发送到指定的频道 channel 。

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。

SUBSCRIBE

```
SUBSCRIBE channel [channel ...]
```

订阅给定频道的信息。

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)。

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 huangz* 匹配所有以 huangz 开头的频道(huangzmsg 、 huangz-blog 、 huangz.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。

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)。

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)

WATCH

```
ATCH 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。

时间复杂度：

O(1)。

返回值：

总是返回 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。

时间复杂度：

O(1)

返回值：

总是 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。

时间复杂度：

O(1)。

返回值：

总是返回 OK 。

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)。

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` 。

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 。

时间复杂度：

O(1)。

返回值：

总是返回 `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。

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。

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。

时间复杂度：
O(1)
返回值：
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。

时间复杂度：
O(1)
返回值：
OK

ECHO

ECHO message

打印一个特定的信息 message，测试时使用。

时间复杂度：
O(1)
返回值：
message 自身。

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。

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)。

服务器(Server)

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。

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。

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。

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。

SAVE

SAVE

同步保存当前数据库的数据到磁盘。

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 。

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。

时间复杂度：

O(1)

返回值：

一个 UNIX 时间戳。

DBSIZE

DBSIZE

返回当前数据库的 key 的数量。

时间复杂度：

O(1)

返回值：

当前数据库的 key 的数量。

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。

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。

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。

时间复杂度：

尚未明确

返回值：

总是返回 OK。

FLUSHDB

FLUSHDB

清空*当前*数据库中的所有 key。

此命令从不失败。

时间复杂度：

O(1)

返回值：

总是返回 OK。

SHUTDOWN

SHUTDOWN

SHUTDOWN 命令执行以下操作:

- 停止所有客户端
- 如果有最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
- 服务器关闭

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。

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。

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。

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：

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：

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。

清空日志

使用命令 `SLOWLOG RESET` 可以清空 slow log 。

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。

时间复杂度：
O(1)
返回值：
具体请参见下面的测试代码。

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)，不过并非所有配置参数都被 `CONFIG GET` 命令所支持。

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：

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：

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

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。

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 。

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。

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)

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

DEBUG OBJECT

```
DEBUG OBJECT key
```

返回给定 `key` 的调试信息。

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。

DEBUG SEGFAULT

DEBUG SEGFAULT

令 Redis 服务器崩溃，调试用。

时间复杂度：

不明确

返回值：

无

MONITOR

MONITOR

实时打印出 Redis 服务器接收到的命令，调试用。

时间复杂度：

不明确

返回值：

总是返回 OK 。

SYNC

YNC

用于复制功能(replication)的内部命令。

时间复杂度：

不明确

返回值：

不明确