Redis

Key

> Rediskey是二进制安全的，意味着你可以使用任意的二进制序列作为一个key

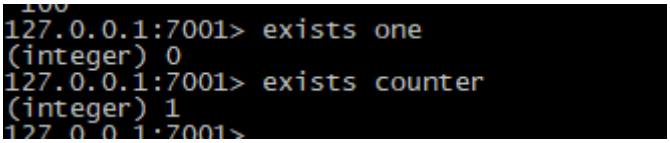
key的规则

- 非常长的key是不建议的，如果key超过1024字节不是一个好的主意，而且在数据集中查找对内存和带宽的消耗是非常昂贵的
- 非常短的key也不是一个好的建议，需要设置有意义的key
- 通常我们设置key的名字为："object-type:id" 例如：user:1000
- key的最大长度为512MB

key操作

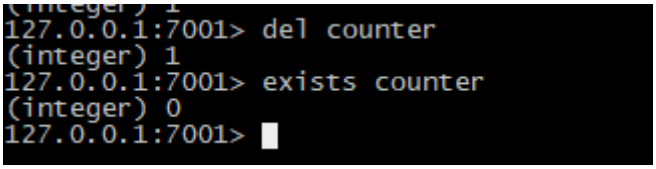
exists

> 判断key是否存在，存在返回1 不存在返回0



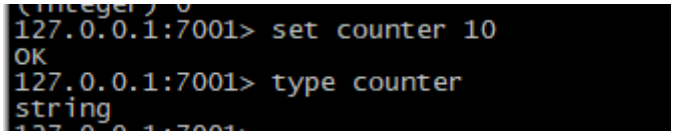
del

> 删除指定key



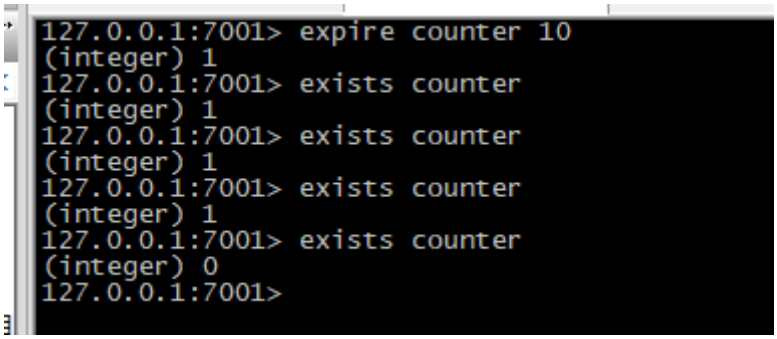
type

> 获取key的类型



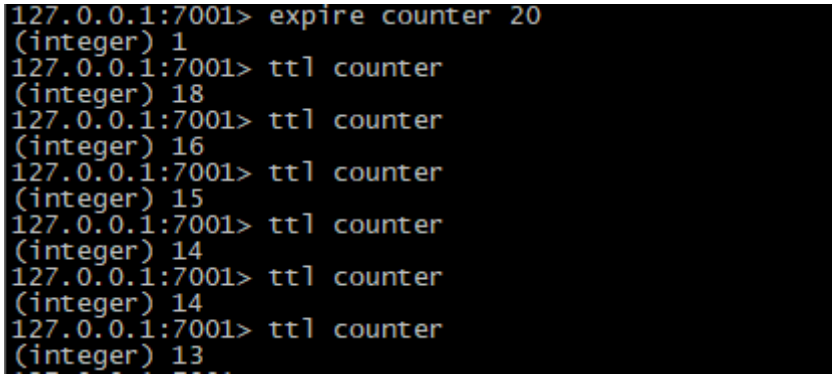
expire

> 指定key的过期时间



ttl

> 查看key还有多久过期

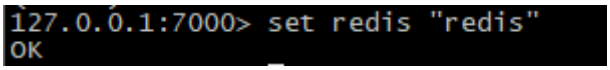


Redis Strings

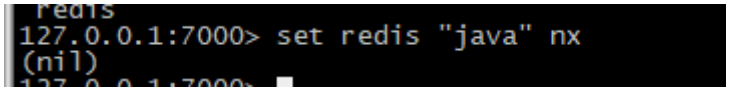
> redis string 类型是可以与rediskey关联的最简单的数据类型。
>
> 它是Memcached中唯一的数据类型，使用起来简单。
>
> 可以用来缓存html页面。

set

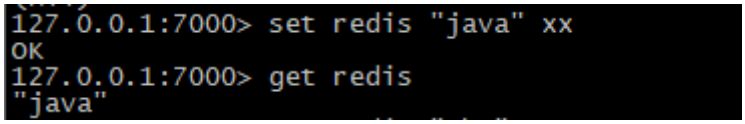
```
set key value
```
注意：set 会覆盖已存在的key的值



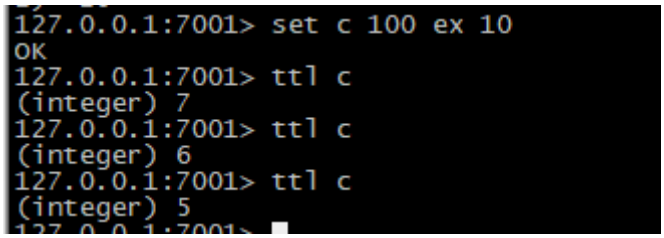

```
set key value nx
```
存在不插入




```
set key value xx
```
存在覆盖，等同set key value



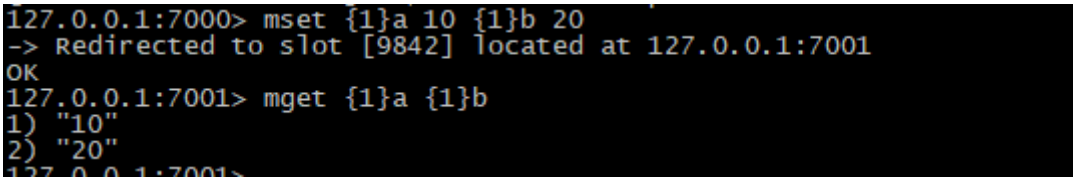
```
set key value ex | px time
设置key 的过期时间 等同与
set key vlaue
expire key time
ex:按秒
px:按毫秒
```



mset

> 批量添加key和值

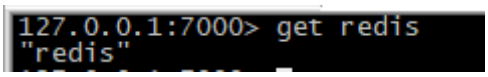
```
mset key1 val1 key2 val2
```



get

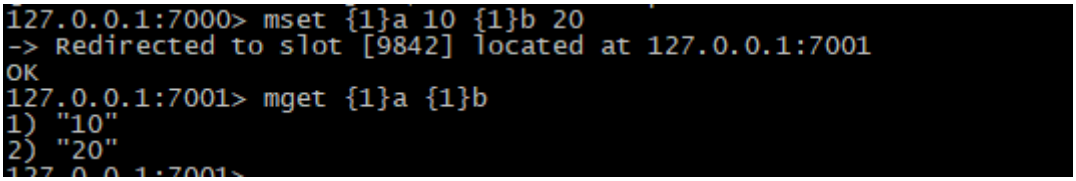
> 语法：get keyName

```
get redis
```



mget

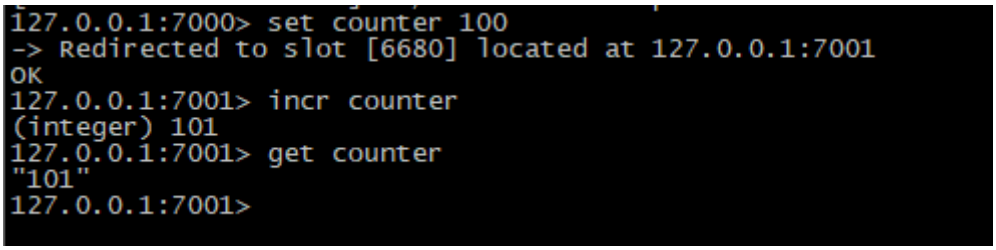
> 批量获取多个key的值



incr

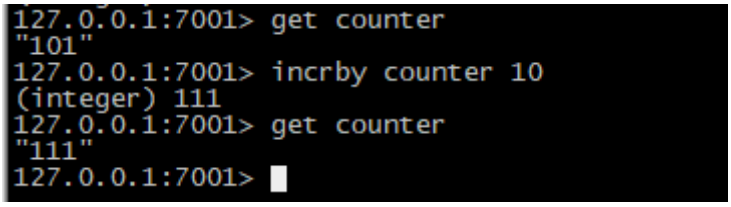
> 描述：该命令是如果key的值是integer类型，该值就自增1
>
> 与其相似的命令有 incrby,decr,decrby

```
set key value
incr key
```



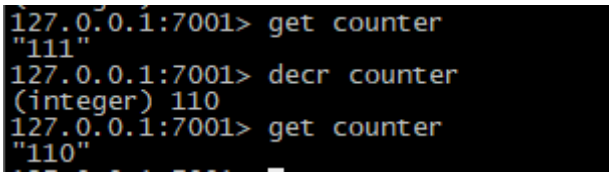
incrby

```
incryby key increment
指定增加的数量
```



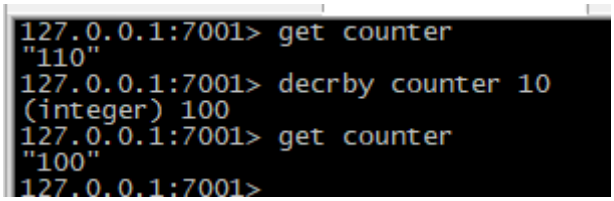
decr

```
decr key
每次递减1
```



decrby

```
decrby key decrement
按照指定值递减
```



Redis List

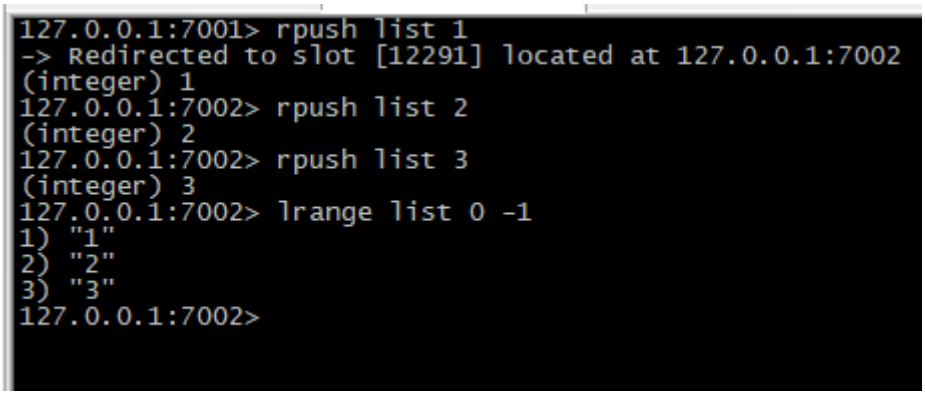
> redis list是通过链表实现的
>
> 链表意味着即使list中有很多数据，但是在头或者尾插入新的元素很快。
>
> Redis列表是用链表实现的，因为对于数据库系统来说，能够以非常快的方式将元素添加到非常长的列表中是至关重要的。另一个强大的优势，是Redis列表可以在固定的时间内以固定的长度获取。

> 当快速访问大量元素集合的中间非常重要时，可以使用不同的数据结构，称为排序集。排序集将在本教程后面介绍

rpush

> 向链表的尾部插入新的元素

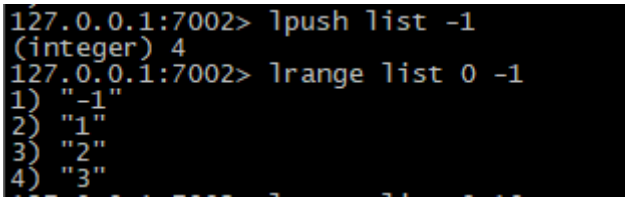
```
rpush key value ...
```



lpush

> 向链表的头插入新元素

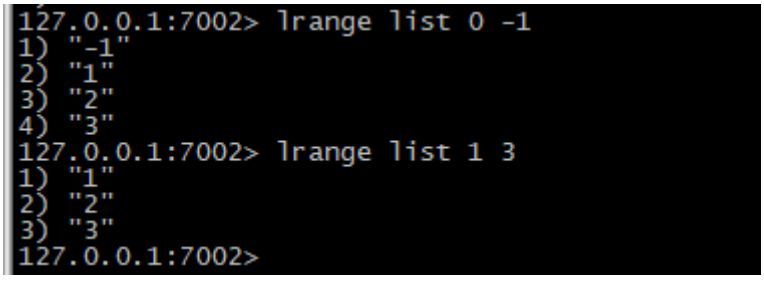
```
lpush key value ...
```



lrange

> 从链表中获取元素的范围。
>
> **注意**：lrange有两个索引，第一个到最后一个元素的范围，两个索引都能为负数，表示倒数第几个开始

```
lrange key start end
```

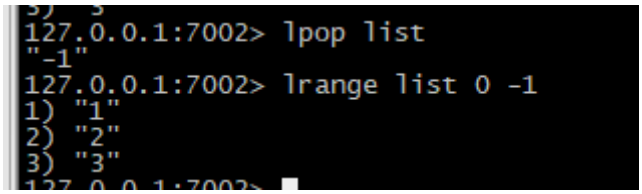


lpop

> Redis列表上定义的一个重要操作是弹出元素的能力。弹出元素是同时从列表中检索元素和从列表中删除元素的操作。可以从左侧和右侧弹出元素

> **操作**：lpop左侧弹出元素，rpop 右侧弹出元素

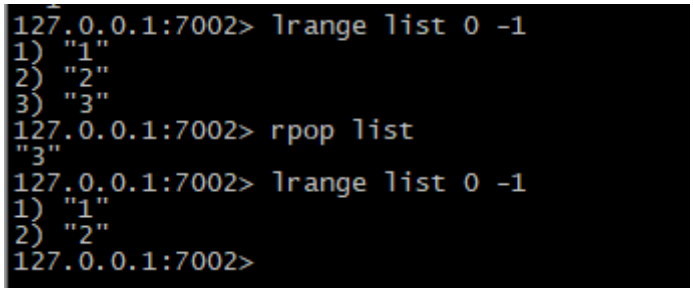
```
lpop key
```



rpop

> 右侧弹出元素

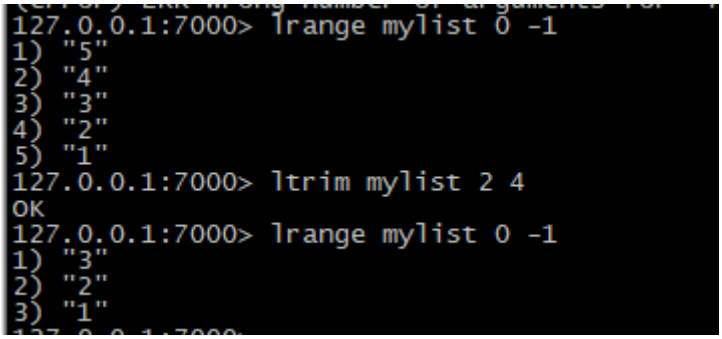
```
rpop key
```



ltrim

> 和lrange相似，但是ltrim没有显示指定的元素范围，而是将此范围设置为新的list，所有超出给定范围的元素都将被删除

```
ltrim key start end
```

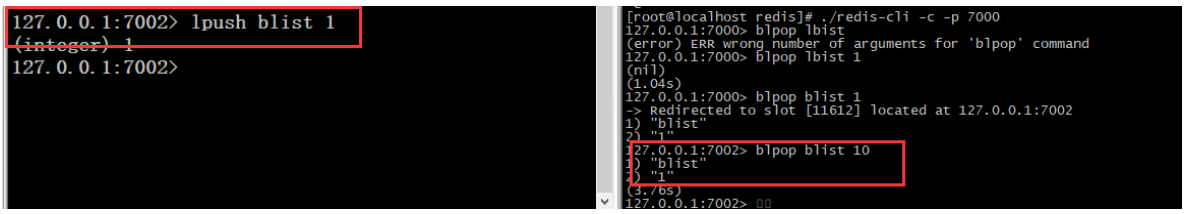


阻塞队列

> **List有一些特殊的特性，它适用于队列的实现，和进程间的阻塞通信：阻塞操作**
>
> 生产者/消费者模式：一个进程LPUSH，一个进程RPOP
>
> 然而，有些时候这个list是空的，因此RPOP只会返回NULL，在生产者消费者模式中消费者将被迫等待一段时间，然后使用RPOP重试。这称为轮询，在这种情况下不是一个好主意，因为它有几个缺点：
>
> - 强制redis和客户端处理无用的命令，因为list是空的，所有的请求都只会返回NULL

> - 将延迟添加到项的处理中，因为在worker接收到NULL之后，它会等待一段时间。为了减少延迟，我们可以减少RPOP调用之间的等待时间，其效果是放大问题1，即更多无用的Redis调用
>
> 因此，redis实现了BRPOP和BLPOP命令，如果list是空的，就会阻塞一段时间

```
blpop key timeout
timeout单位是秒
```



> ***注意***：
>
> 1.BROPO和BLPOP是按照客户端被服务的顺序来的第一个客户端被阻塞等待list，当list被push数据的时候，首先被服务
>
> 2.返回值与RPOP不同，BRPOP有两个元素的数组，包含key的名字
>
> 3.如果超时时间到了，会将null返回

总结

> key的自动创建和删除，下面是总结的三种规则
>
> - 当我们增加一个元素到指定的数据类型时，如果key不存在，那么就会在添加元素前先创建一个空的数据类型
> - 当我们从聚合数据类型中删除元素时，如果值仍然为空，则键将自动销毁。流数据类型是此规则的唯一例外
> - 使用空键调用只读命令，如LLEN（返回列表的长度）或删除元素的write命令，总是会产生相同的结果，就好像该键包含该命令期望找到的类型的空聚合类型一样

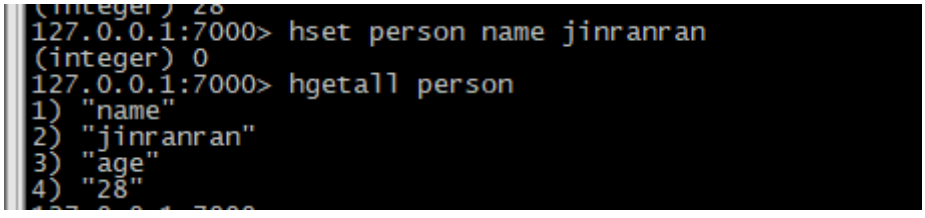
Redis Hash

> redis hash和普通的hash一样，具有key - value
>
> Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）。

hset

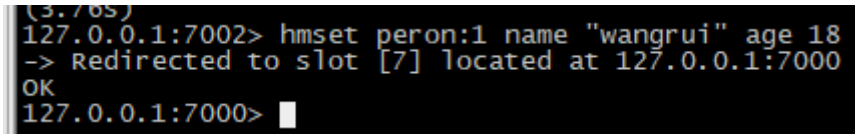
> 设置单个字段的值,一般用于修改操作

```
hset key field value
```



hmset

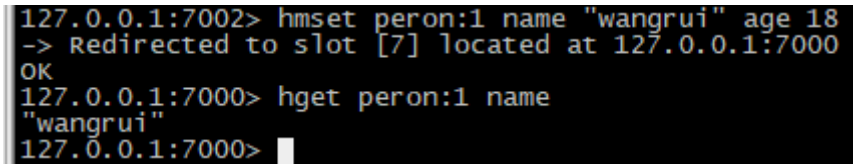
```
hmset key  field1 value1 field2 value2
```



hget

> 返回单个的字段

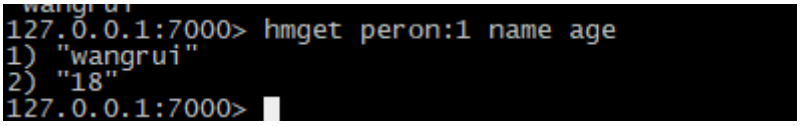
```
hget key field
```



hmget

> 获取多个字段值

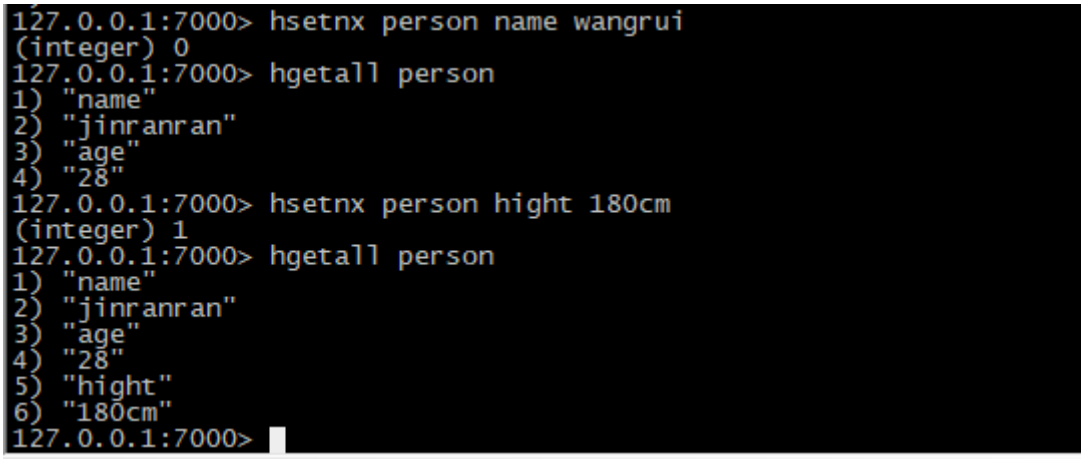
```
hmget key field1 field2
```



hsetnx

> 当field不存在时，设置hash字段的值

```
hsetnx key field value
```



hgetall

> 获取key中的所有字段值

```
hgetall key
```

```
127.0.0.1:7000> hmset person name wangrui age 18
OK
127.0.0.1:7000> hgetall person
1) "name"
2) "wangrui"
3) "age"
4) "18"
127.0.0.1:7000>
```

hincrby

> 对单个字段进行增加操作,只能对数字类型的字段进行操作

```
hincrby key field increment
```

错误示例：

```
127.0.0.1:7000> hincrby person name 10
(error) ERR hash value is not an integer
127.0.0.1:7000>
```

正确示例：

```
OK
127.0.0.1:7000> hgetall person
1) "name"
2) "wangrui"
3) "age"
4) "18"
127.0.0.1:7000> hincrby person name 10
(error) ERR hash value is not an integer
127.0.0.1:7000> hincrby person age 10
(integer) 28
127.0.0.1:7000>
```

hdel

> 删除一个或多个字段,用于删除对象属性

```
hdel key field1 [field2]
```

```
127.0.0.1:7000> hdel peron:1 name size
(integer) 2
127.0.0.1:7000> hgetall peron:1
1) "age"
2) "18"
127.0.0.1:7000>
```

hexists

> 判断对象中是否存在某个属性
>
> 存在返回1，不存在返回0

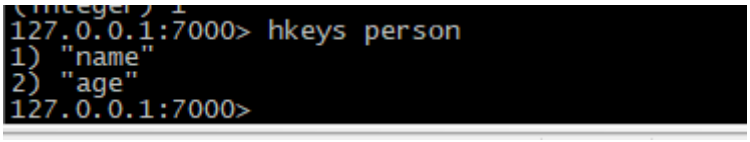
```
hexists key field
判断key中是否存在field
```

```
127.0.0.1:7000> hexists person size
(integer) 0
127.0.0.1:7000> hexists person name
(integer) 1
127.0.0.1:7000>
```

hkeys

> 获取所有hash表中的字段

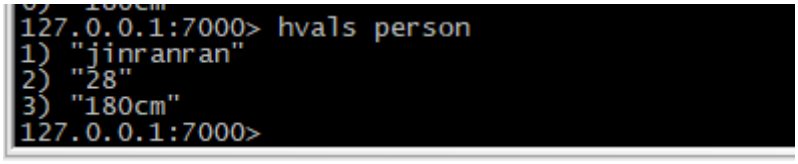
```
hkeys key
```



hvals

> 获取hash表所有的值

```
hvals key
```



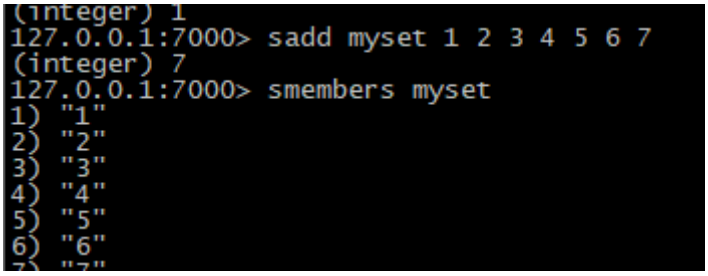
Redis Set

> Redis Sets是无序的,不可重复的字符串的集合

sadd

> 向集合中添加元素

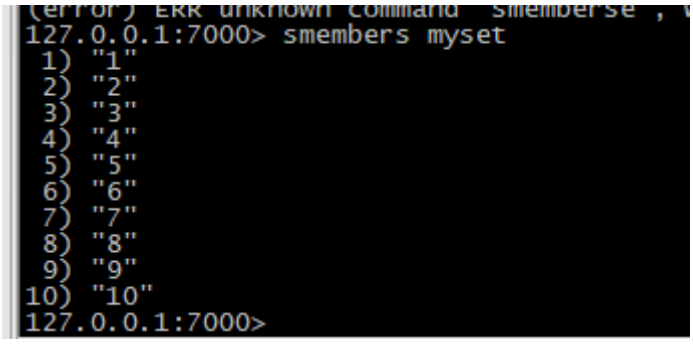
```
sadd key value1 value2 ...
```



smembers

> 获取集合中所有元素

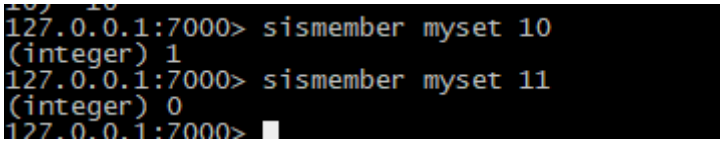
```
smembers key
```



sismember

> 判断元素是否存在，存在返回1，不存在返回0

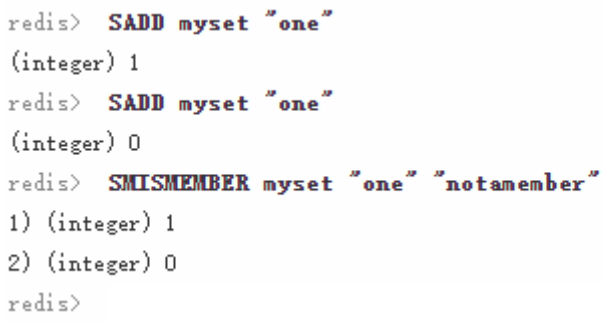
```
sismember key value
```



smismember

> **注：**该命令在**6.2.0.**版本有效
>
> 批量判断元素是否存在集合中，存在返回1，不存在返回0
>
> 返回结果的顺序和请求的参数顺序一致

```
smismember key member [member]
```



sinter

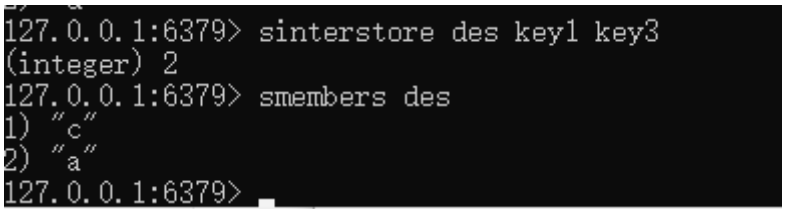
> 求出各个集合中的并集

```
sinter set1 set2 set3 ...
```

sinterstore

> 和sinter命令一样，但是这个命令会将返回值存到目标集合中

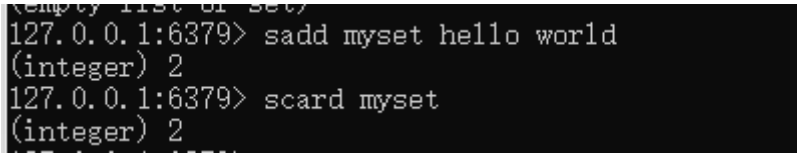
```
sinterstore destination key [key]
```



scard

> 返回集合中的元素的个数

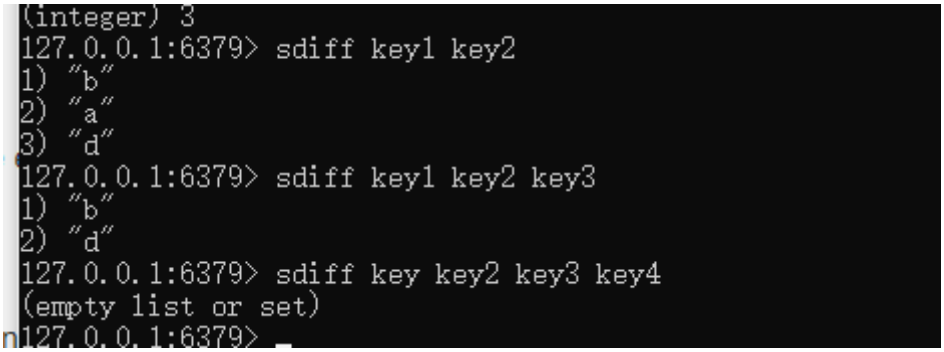
```
scard key
```



sdiff

> 返回由第一个集合和所有连续集合之间的差异产生的集合成员
>
> ***注意***：如果key不存在的时候，就返回一个空集合

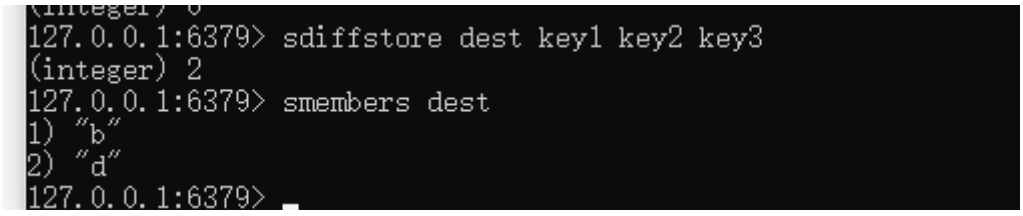
```
sdiff key1 key2 ...
```



sdiffstore

> 和sdiff命令一样，但是这个命令会将结果集存储到目标集合中

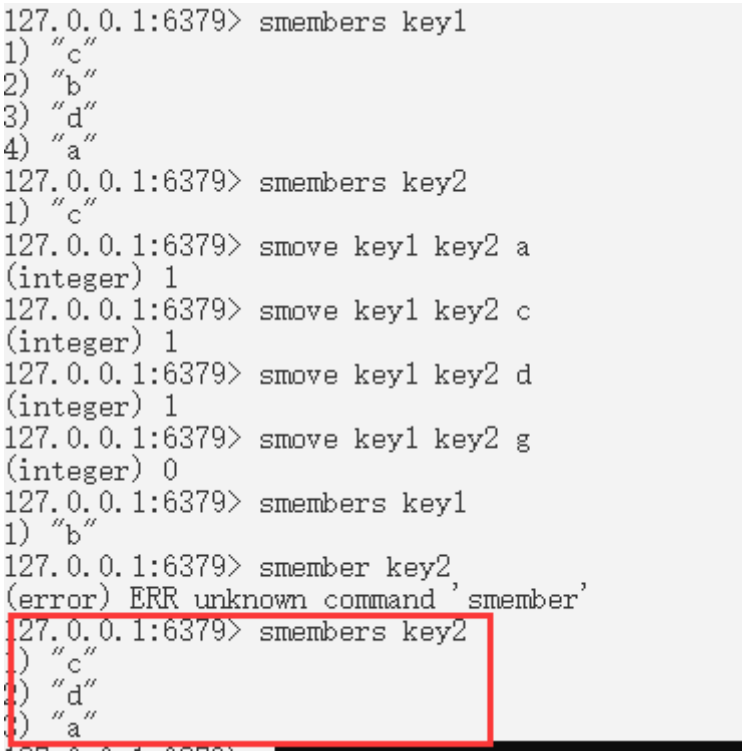
```
sdiffstore destination_key key1 key2 ...
```



smove

> - 移动元素 从**源**集合到**目标**集合，并将**源**集合中的元素删除。
> - 如果元素不存在源集合中，或者源集合不存在，不做任何操作并返回0。
> - 如果移动的元素在目标集合中存在，仅仅会移除源集合中的元素。
> - 如果源和目标不是集合结构，就返回错误

```
smove source destination member
```



spop

> - 移除并返回一个或多个随机数量的元素
> - 这个操作和srandmember相似，但是它不会移除元素
> - 默认的这个命令弹出单个元素

```
spop key [count]  count可选
```

srandmember

```
srandmember key [count]
```

> - 当count不传时，从key的集合中随机返回一个元素
> - 当count为正数时，从集合中随机返回count数的元素，不会重复返回元素
> - 当count为负数时，允许重复返回元素

```
127.0.0.1:6379> sadd myset one tow three
(integer) 3
127.0.0.1:6379> srandmember myset
"world"
127.0.0.1:6379> srandmember myset 2
1) "hello"
2) "three"
127.0.0.1:6379> srandmember myset -4
1) "three"
2) "one"
3) "one"
4) "one"
127.0.0.1:6379>
```

srem

```
srem key member [member]
```

> - 删除集合中的指定的元素
> - 指定的元素如果集合中不存在就会被忽略
> - 如果key不存在，返回0
> - 如果key不是集合结构，返回错误

```
127.0.0.1:6379> smembers key2
1) "f"
2) "g"
3) "k"
4) "j"
127.0.0.1:6379> srem key2 f
(integer) 1
127.0.0.1:6379> smembers key2
1) "g"
2) "k"
3) "j"
127.0.0.1:6379> srem key2 h
(integer) 0
127.0.0.1:6379> smembers key2
1) "g"
2) "k"
3) "j"
127.0.0.1:6379> srem name a
(error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379>
```

Redis Sorted Set

zadd

```
zadd key [nx|xx] [gt|lt] [ch] [incr] score member [score member] ...
```

> - 添加元素并指定此元素的分数
> - 如果指定的元素存在分数，则这个分数会被更新，并将元素插入到正确位置，确保排序正确
> - 如果key不存在，则一个新的排序集合被创建
> - 如果key存在，但不是排序集合结构，则返回错误
> - 分数值应该是双精度浮点数的字符串表示形式+inf和-inf值也是有效值。

> ***参数说明***
>
> **XX** : 仅仅更新存在的元素，不新增新的元素

> **NX** : 仅仅新增新元素，不更新已存在的元素
>
> **LT** ： 新增新元素，但是如果元素存在，新的分数小于当前元素的分数则更新
>
> **GT**： 新增新元素，但是如果元素存在，新的分数大于当前元素的分数则更新
>
> **CH**： 修改返回值为发生变化的成员总数，原始是返回新添加成员的总数 (CH 是 *changed* 的意思)。更改的元素是**新添加的成员**，已经存在的成员**更新分数**。 所以在命令中指定的成员有相同的分数将不被计算在内。注：在通常情况下，`ZADD`返回值只计算新添加成员的数量
>
> **INCR** : 对元素的分数进行递增操作

```
127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "two"
2) "2"
3) "three"
4) "3"
5) "one"
6) "4"
127.0.0.1:6379> zadd myzset xx 1 one
(integer) 0
127.0.0.1:6379> zrange myzset 0 -1
1) "one"
2) "two"
3) "three"
127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
127.0.0.1:6379>
```

```
127.0.0.1:6379> zadd myzset xx ch 4 one
(integer) 1
127.0.0.1:6379>
```

zcard

```
zcard key
```

> 返回有序集合的成员个数

```
127.0.0.1:6379> zcard myzset
(integer) 3
127.0.0.1:6379>
```

zcount

```
zcount key min max
```

- 统计key的元素的分数在min到max之间的元素个数

```
127.0.0.1:6379> zcount myzset 2 2
(integer) 1
127.0.0.1:6379>
```

zdiff

```
zdiff numkeys key [key ...] [withscores]
```

> 这个命令和ZDIFFSTORE 相似，但是zdiffsotre会存储结果集
>
> ***注意*** ： 该命令在**6.2.0**支持

```
redis> ZADD zset1 1 "one"
(integer) 1
redis> ZADD zset1 2 "two"
(integer) 1
redis> ZADD zset1 3 "three"
(integer) 1
redis> ZADD zset2 1 "one"
(integer) 1
redis> ZADD zset2 2 "two"
(integer) 1
redis> ZDIFF 2 zset1 zset2
1) "three"
redis> ZDIFF 2 zset1 zset2 WITHSCORES
1) "three"
2) "3"
redis>
```

zdiffstore

```
zdiffstore destination numKeys key [key ...]
```

> - 计算第一个和所有连续输入排序集之间的差值，并将结果存储在目标中。输入键的总数由numkeys指定
> - 如果destination存在将会被覆盖

zincrby

```
zincrby key increment member
```

> - member按照increment的值增加
> - 如果member不存在，会新增一个member并且score等于increment
> - 如果key不存在，会为指定的member创建一个新的set
> - 如果key存在但是key不是sorted set结构，则会报错
> - increment可以是负数，双精度浮点数

```
127.0.0.1:6379> zincrby zset1 10 one
"11"
127.0.0.1:6379> zrange zset1 0 -1 withscores
1) "two"
2) "2"
3) "three"
4) "3"
5) "one"
6) "11"
127.0.0.1:6379>
```

zinter

```
zinter numKeys key [key ...] [WEIGHTS weight [weight ...]] AGGREGATE
SUM|MIN|MAX] [WITHSCORES]
```

> ***注意*** ： 适用于**6.2.0.**以上版本
>
> WEIGHTS：给指定的key的成员的分数乘以该权重的值
>
> AGGREGATE：聚合操作， 默认是sum

```
redis> ZADD zset1 1 "one"
(integer) 1
redis> ZADD zset1 2 "two"
(integer) 1
redis> ZADD zset2 1 "one"
(integer) 1
redis> ZADD zset2 2 "two"
(integer) 1
redis> ZADD zset2 3 "three"
(integer) 1
redis> ZINTER 2 zset1 zset2
1) "one"
2) "two"
redis> ZINTER 2 zset1 zset2 WITHSCORES
1) "one"
2) "2"
3) "two"
4) "4"
redis>
```

zinterstore

```
zinterstore destination numkeys key [key ...] [WEIGHT weight [weight ...]]
[AGGREGATE SUM|MIN|MAX]
```

> - 计算指定key的交集并将结果存储到destination中

```
127.0.0.1:6379> zadd zset1 1 one 2 two
(integer) 2
127.0.0.1:6379> zadd zset2 1 one 2 two 3 three
(integer) 3
127.0.0.1:6379> zinterstore out 2 zset1 zset2  WEIGHTS 2 3 AGGREGATE MIN
(integer) 2
127.0.0.1:6379> zrange out 0 -1 withscores
1) "one"
2) "2"
3) "two"
4) "4"
127.0.0.1:6379>
```

zlexcount

```
zlexcount key min max
```

> 返回key中成员分数在 min 到max之间的个数

```
127.0.0.1:6379> zadd myzset 0 a 0 b 0 c 0 d 0 e 0 f 0 g
(integer) 7
127.0.0.1:6379> zlexcount myzset - +
(integer) 7
127.0.0.1:6379>
```

zmscore

```
zmscore key member [member ...]
```

> - 返回指定成员的分数
> - 如果成员不存在就返回nil
> - 该命令在**6.2.0**版本以上可用

```
redis> ZADD myzset 1 "one"
(integer) 1
redis> ZADD myzset 2 "two"
(integer) 1
redis> ZMSCORE myzset "one" "two" "nofield"
1) "1"
2) "2"
3) (nil)
redis>
```

zpopmax

```
zpopmax key [count]
```

> - 删除和返回分数最高的成员和成员的分数
> - count没有指定默认是1
> - count大于1时，成员按照分数倒序排序

```
27.0.0.1:6379> zadd myzset 1 one 2 two 3 three
integer) 3
27.0.0.1:6379> zpopmax myzset
) "three"
) "3"
27.0.0.1:6379> zpopmax myzset 2
) "two"
) "2"
) "one"
) "1"
```

zpopmin

```
zpopmin key [count]
```

> - 删除和返回分数最低的成员和成员的分数
> - count没有指定默认是1
> - count大于1时，成员按照分数正序排序

zrandmember

```
zrandmember  key  [count [WITHSCORES]]
```

- 当只有key参数时，随机返回一个成员
- 如果count>0 返回的是不同的元素，长度等于count
- 如果count>0 且大于集合长度，则该命令将只返回整个排序集，而不返回其他元素
- 如果count<0 允许多次返回相同的成员

```
redis>
ZADD dadi 1 uno 2 due 3 tre 4 quattro 5 cinque 6 sei
(integer) 6
redis>  ZRANDMEMBER dadi
"sei"
redis>  ZRANDMEMBER dadi
"tre"
redis>  ZRANDMEMBER dadi -5 WITHSCORES
1) "quattro"
 2) "4"
 3) "tre"
 4) "3"
 5) "cinque"
 6) "5"
 7) "tre"
 8) "3"
 9) "quattro"
10) "4"
redis>
```

zrange

```
zrange key min max [BYSCORE|BYLEX] [REV] [LIMIT offset count][WITHSCORES]
```

- 该命令在 **6.2.0** 及以上版本可以替代ZREVRANGE, ZRANGEBYSCORE, ZREVRANGEBYSCORE, ZRANGEBYLEX and ZREVRANGEBYLEX.命令
- 返回指定范围的成员集合

```
127.0.0.1:6379> ZADD dadi 1 uno 2 due 3 tre 4 quattro 5 cinque 6 sei
(integer) 6
127.0.0.1:6379> zrange dadi 1 5
1) "due"
2) "tre"
3) "quattro"
4) "cinque"
5) "sei"
127.0.0.1:6379> zrange dadi 1 5 withscores
 1) "due"
 2) "2"
 3) "tre"
 4) "3"
 5) "quattro"
 6) "4"
 7) "cinque"
 8) "5"
 9) "sei"
10) "6"
```

zrangebylex

```
zrangebylex key min max [LIMIT offset count]
```

- 当所有的成员的分数相同，则根据字典排序
- 返回分数在min和max之间的成员
- 指定间隔
  - 有效的开始和结束必须以 `(` or `[` 表示不包含或者包含
  - 如果指定 `+`or `-`表示正无穷和负无穷

```
127.0.0.1:6379> ZADD myzset 0 a 0 b 0 c 0 d 0 e 0 f 0 g
(integer) 7
127.0.0.1:6379> zrangebylex myzset - [c
1) "a"
2) "b"
3) "c"
127.0.0.1:6379> zrangebylex myzset - (c
1) "a"
2) "b"
127.0.0.1:6379> zrangebylex myzset [a (g
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
127.0.0.1:6379>
```

zrangebyscore

```
zrangebyscore key min max [WITHSOCRES] [LIMIT offset count]
```

- 返回成员分数在min和max之间的所有成员
- 返回的顺序是分数从低到高
- 区间设置
  - min和max可以用 -inf和+inf表示从负无穷到正无穷
  - 可以用 `(` 表示不包含

```
127.0.0.1:6379> zrangebyscore myzset -inf +inf withscores
 1) "a"
 2) "0"
 3) "b"
 4) "0"
 5) "c"
 6) "0"
 7) "d"
 8) "0"
 9) "e"
10) "0"
11) "f"
12) "0"
13) "g"
14) "0"
127.0.0.1:6379> zrangebyscore myzset (0 (2
(empty list or set)
127.0.0.1:6379>
```

zrangestore

```
zrangestore dst src min max [BYSCORE|BYLEX] [REV] [LIMIT offset count]
```

- 和zrange一样，但是取出来的成员会存储到 `dst` 中
- 该命令在`6.2.0`以上版本使用

```
redis> ZADD srczset 1 "one" 2 "two" 3 "three" 4 "four"
(integer) 4
redis> ZRANGESTORE dstzset srczset 2 -1
(integer) 2
redis> ZRANGE dstzset 0 -1
1) "three"
2) "four"
redis>
```

zrank

```
zrank key member
```

- 返回成员在key中的索引，分数是从小到大排序
- rank是从0开始
- 使用ZREVRANK获得从大到小排序的索引

```
127.0.0.1:6379> zadd myzset 1 one 2 two 3 three
(integer) 3
127.0.0.1:6379> zrank myzset one
(integer) 0
127.0.0.1:6379> zrank myzset three
(integer) 2
127.0.0.1:6379> zrank myzset four
(nil)
127.0.0.1:6379>
```

zrem

```
zrem key member [member ...]
```

- 删除key中指定的成员，如果成员不存在就忽略
- 如果key不是一个排序集合则返回错误

```
127.0.0.1:6379> zrange myzset 0 -1
1) "one"
2) "two"
3) "three"
127.0.0.1:6379> zrem myzset one
(integer) 1
127.0.0.1:6379> zrange myzset 0 - 1
(error) ERR value is not an integer or out of range
127.0.0.1:6379> zrange myzset 0 -1
1) "two"
2) "three"
127.0.0.1:6379>
```

zremrangebylex

```
zremrangebylex key min max
```

- 删除存储在由min和max指定的字典范围之间的键处的排序集中的所有元素

```
127.0.0.1:6379> zrange myzset 0 -1
 1) "ALPHA"
 2) "aaaa"
 3) "alpha"
 4) "b"
 5) "c"
 6) "d"
 7) "e"
 8) "foo"
 9) "zap"
10) "zip"
127.0.0.1:6379> zremrangebylex myzset [alpha [omega
(integer) 6
127.0.0.1:6379> zrange myzset 0 -1
1) "ALPHA"
2) "aaaa"
3) "zap"
4) "zip"
127.0.0.1:6379>
```

zremrangebyrank

```
zremrangebyrank key start stop
```

- 根据索引删除，从0开始

```
127.0.0.1:6379> zrange myzset 0 -1
1) "ALPHA"
2) "aaaa"
3) "alpha"
4) "foo"
5) "zap"
6) "zip"
127.0.0.1:6379> zremrangebyrank myzset 1 3
(integer) 3
127.0.0.1:6379> zrange myzset 0 -1
1) "ALPHA"
2) "zap"
3) "zip"
127.0.0.1:6379>
```

zremrangebyscore

```
zremrangebyscore key min max
```

- 删除分数在min到max之间的成员

```
127.0.0.1:6379> zadd myzset 1 one 2 two 3 three
(integer) 3
127.0.0.1:6379> zremrangebyscore myzset -inf (2
(integer) 1
127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "two"
2) "2"
3) "three"
4) "3"
127.0.0.1:6379>
```

zrevrange

```
zrevrange key start stop [WITHSCORES]
```

- 返回指定范围的成员倒序返回
- `6.2.0`这个命令废弃，用zrange代替，里面的REV参数

```
127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "on"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
127.0.0.1:6379> zrevrange myzset 0 -1 withscores
1) "three"
2) "3"
3) "two"
4) "2"
5) "on"
6) "1"
127.0.0.1:6379>
```

zrevrangebylex

```
zrevrangebylex  key max min [LIMIT offset count]
```

- 根据字典倒序排序，并返回指定大小范围的成员
- LIMIT相当于分页，可用于获取TOP10等需求
- 这个命令通常是用于分数一样的成员

```
127.0.0.1:6379> zrange myzset 0 -1 withscores
 1) "a"
 2) "0"
 3) "b"
 4) "0"
 5) "c"
 6) "0"
 7) "d"
 8) "0"
 9) "e"
10) "0"
11) "f"
12) "0"
13) "g"
14) "0"
127.0.0.1:6379> zrevrangebylex myzset [g [d limit 0 4
1) "g"
2) "f"
3) "e"
4) "d"
127.0.0.1:6379>
```

zrevrangebyscore

```
zrevrangebyscore key max min [WITHSCORES] [LIMIT offset count]
```

- 根据分数倒序排序，并返回指定大小范围的成员
- LIMIT相当于分页，可用于获取TOP10等需求

```
(integer) 3
127.0.0.1:6379> zrange myzset 0 -1
1) "one"
2) "two"
3) "three"
127.0.0.1:6379> zrevrangebyscore myzset 3 1 withscores limit 0 2
1) "three"
2) "3"
3) "two"
4) "2"
127.0.0.1:6379>
```

zrevrank

```
zrevrank key member
```

- 获取member在key中从后往前排第几，默认从0开始

```
127.0.0.1:6379> zrange myzset 0 -1 withscores
1) "one"
2) "1"
3) "two"
4) "2"
5) "three"
6) "3"
127.0.0.1:6379> zrevrank myzset one
(integer) 2
127.0.0.1:6379>
```

zscan

```
zscan key cursor [MATCH pattern] [COUNT count]
```

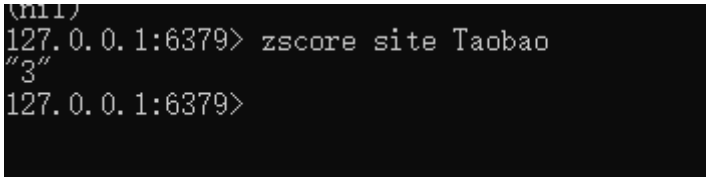
- 用于迭代有序集合中的元素（包括元素成员和元素分值）
- curosr:游标
- pattern:匹配模式
- count:指定从数据集里返回多少个元素，默认值为10

```
127.0.0.1:6379> ZADD site 1 "Google" 2 "Runoob" 3 "Taobao" 4 "Weibo"
(integer) 4
127.0.0.1:6379> zscan site 0 match "b*"
1) "0"
2) (empty list or set)
127.0.0.1:6379> zscan site 0 match "*b*"
1) "0"
2) 1) "Runoob"
   2) "2"
   3) "Taobao"
   4) "3"
   5) "Weibo"
   6) "4"
```

zscore

```
zscore key member
```

- 返回成员的分数



zunion

```
zunion numKeys key [key ...] [WEIGHTS weight [weight]] [AGGREGATE SUM|MIN|MAX]
[WITHSCORES]
```

> - 和zunionstore命令相似，但是这个命令只结果不存储结果
> - 该命令在`6.2.0`版本以上可以使用

```
redis> ZADD zset1 1 "one"
(integer) 1
redis> ZADD zset1 2 "two"
(integer) 1
redis> ZADD zset2 1 "one"
(integer) 1
redis> ZADD zset2 2 "two"
(integer) 1
redis> ZADD zset2 3 "three"
(integer) 1
redis> ZUNION 2 zset1 zset2
1) "one"
2) "three"
3) "two"
redis> ZUNION 2 zset1 zset2 WITHSCORES
1) "one"
2) "2"
3) "three"
4) "3"
5) "two"
6) "4"
```

zunionstore

```
zunionstore dst numkeys key [key ...] [WEIGHTS weight [weight]] [AGGREGATE
SUM|MIN|MAX]
```

> - 求并集并存储到`dst`目标集合中
> - WEIGHT：使用“权重”选项，可以为每个输入排序集指定乘法因子。这意味着在传递给聚合函数之前，每个输入排序集中每个元素的得分都要乘以这个因子。如果未给定权重，则乘法因子默认为1
> - AGGREGATE：使用AGGREGATE选项，可以指定如何聚合联合的结果。此选项默认为SUM，其中元素的分数在其存在的输入之间求和。当此选项设置为“最小”或“最大”时，结果集将包含元素在存在的输入之间的最小或最大得分
> - 如果`dst`存在则被覆盖

```
27.0.0.1:6379> zunionstore out 2 zset1 zset2 weights 2 3
integer) 2
27.0.0.1:6379> zrange out 0 -1
) "one"
) "two"
27.0.0.1:6379> zrange out 0 -1 withscores
) "one"
) "2"
) "two"
) "4"
27.0.0.1:6379>
```

Redis HyperLogLog

> - Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的
> - 在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比
> - 但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素

pfadd

```
pfadd key element [element ...]
```

> - 添加指定元素到 HyperLogLog 中

```
127.0.0.1:6379> PFADD h11 a b c d e f g
(integer) 1
127.0.0.1:6379> pfcount h11
(integer) 7
127.0.0.1:6379>
```

pfcount

```
pfcount key
```

> - 返回HyperLogLog中的元素个数

```
127.0.0.1:6379> PFADD h11 a b c d e f g
(integer) 1
127.0.0.1:6379> pfcount h11
(integer) 7
127.0.0.1:6379>
```

pfmerge

```
pfmerge dest key [key ...]
```

> - 将多个HyperLogLog值合并到一个唯一值中，该值将近似于所观察到的源HyperLogLog结构集合的联合基数。
> - 计算的合并HyperLogLog设置为目标变量，如果不存在，则创建该变量（默认为空超日志）。
> - 如果目标变量存在，则将其视为源集之一，其基数将包含在计算的HyperLogLog的基数中

```
127.0.0.1:6379> pfadd h111 foo bar zap a
(integer) 1
127.0.0.1:6379> pfadd h112 a b c foo
(integer) 1
127.0.0.1:6379> pfmerge h113 h111 h112
OK
127.0.0.1:6379> pfcount h113
(integer) 6
127.0.0.1:6379>
```

- HyperLogLog并不是一种新的数据结构（实际类型为字符串类型），而是一种基础算法，通过HyperLogLog可以利用极小的内存空间完成独立总数的统计，数据集可以是IP、Email、ID等
- 基数：一个集合中不同元素的个数，集合中的元素可重复
- 应用场景：
  - 比如电商统计有多少人浏览了某个商品
  - 下面示例中表示总共有4个人点击了`goods:url`链接

```
127.0.0.1:6379> pfadd goods:url user:001
(integer) 1
127.0.0.1:6379> pfadd goods:url user:001
(integer) 0
127.0.0.1:6379> pfadd goods:url user:002
(integer) 1
127.0.0.1:6379> pfadd goods:url user:003
(integer) 1
127.0.0.1:6379> pfadd goods:url user:005
(integer) 1
127.0.0.1:6379> pfadd goods:url user:003
(integer) 0
127.0.0.1:6379> pfcount goods:url
(integer) 4
127.0.0.1:6379>
```

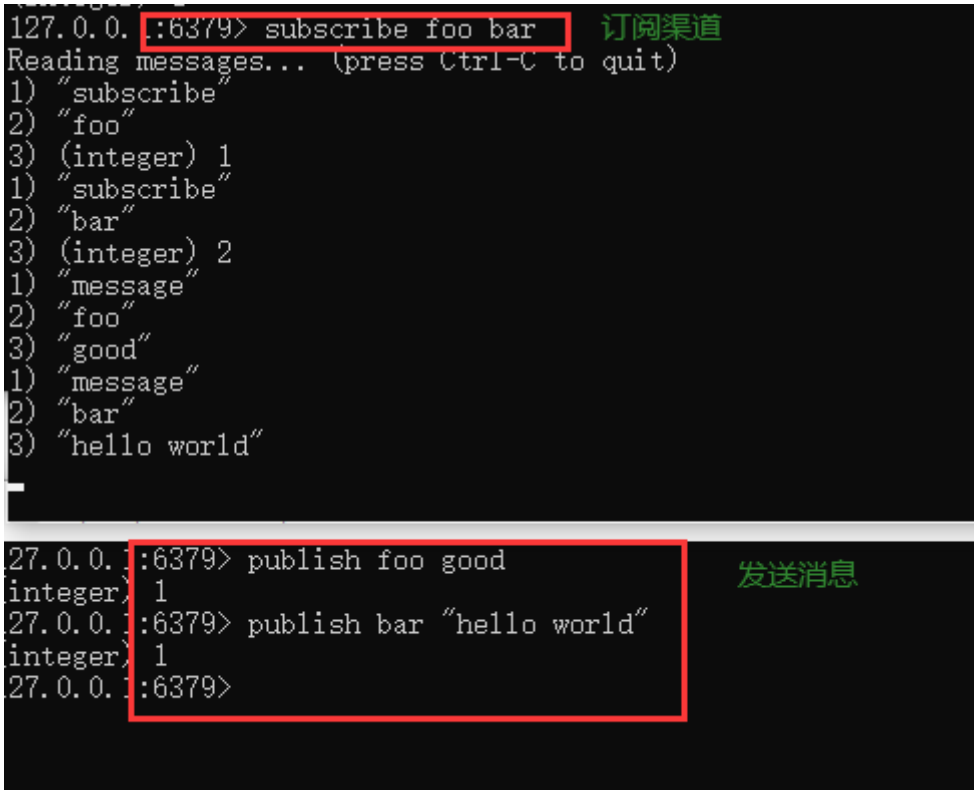
Reids Pub/Sub

- Redis消息发布与订阅实现了`消息传递范式`
- 发布者不需要直接发送消息给接收者，而是把消息封装到channel中，而不知道有哪些订阅者
- 订阅者可能对一个或者多个channel感兴趣，并且只接收感兴趣的消息，而不知道有什么发布者。
- 发布者和订阅者的这种解耦可以允许更大可伸缩性和更动态的网络拓扑

subscribe

```
subscribe channel [channel ...]
```

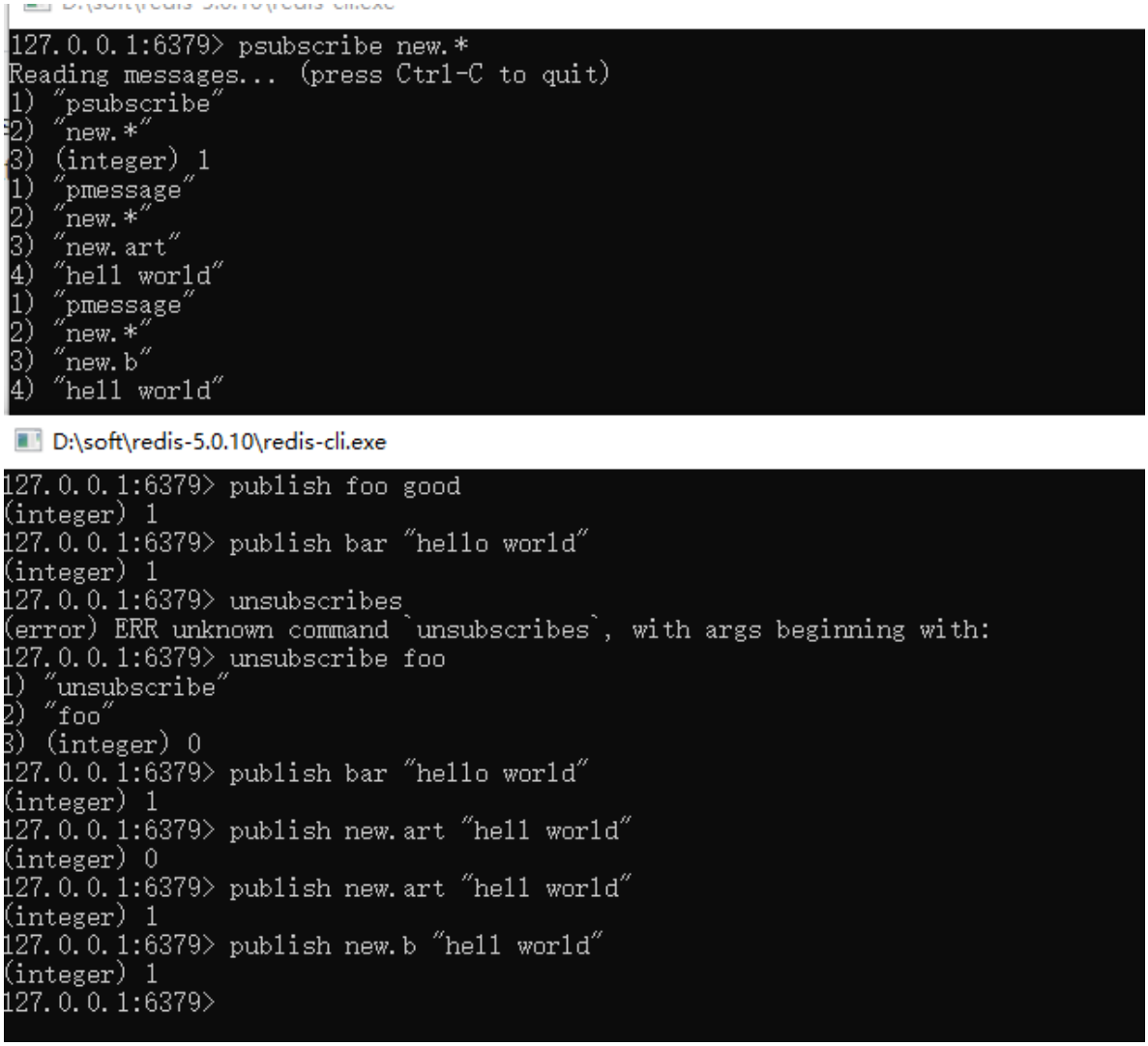
- 订阅了 多个channel



psubscribe

```
psubscribe channel_pattern
```

> - 订阅某种规则的channel



事件通知

> - 通知允许客户端订阅发布/子通道，以便接收以某种方式影响Redis数据集的事件。
> - 例如这些事件可以被通知
>   - 影响给定key的所有命令
>   - 所有接收LPUSH操作的键
>   - database0中过期的key
> - 事件通过Pub/Sub发送，因为客户端实现Pub/Sub就能使用这些特征

博客

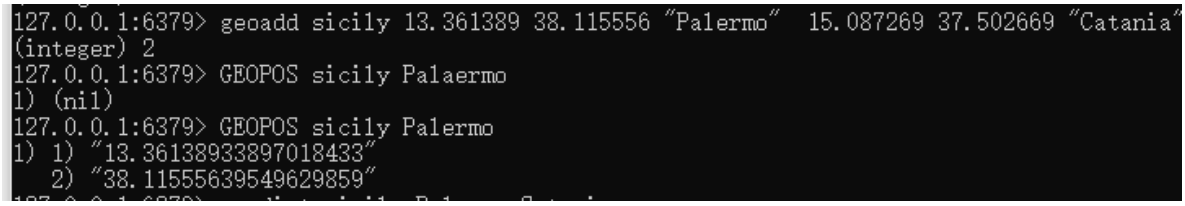
Redis GEO

> Redis GEO 主要用于存储地理位置信息，并对存储的信息进行操作，该功能在 Redis 3.2 版本新增。

geoadd

```
geoadd key longitude latitude member [longitude latitude membe ...]
```

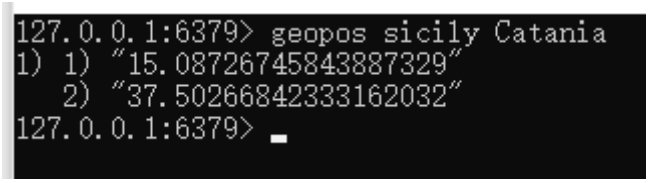
> 用于存储指定的地理空间位置，可以将一个或多个经度(longitude)、纬度(latitude)、位置名称(member)添加到指定的 key 中



geopos

```
geopos key member [member ...]
```

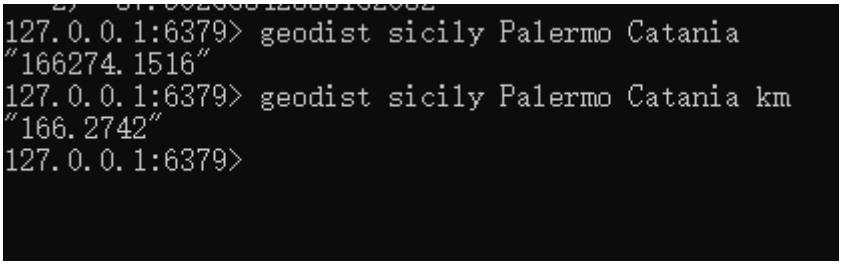
> geopos 用于从给定的 key 里返回所有指定名称(member)的位置（经度和纬度），不存在的返回 nil。



geodist

```
geodist key member1 member2 [m|km|ft|mi]
```

> - geodist 用于返回两个给定位置之间的距离
> - m：米 km：千米 ft：英尺 mi：英里，**默认 m**



georadius

```
georadius key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST]
[WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]
```

> - georadius 以给定的经纬度为中心， 返回键包含的位置元素当中， 与中心的距离不超过给定最大距离的所有位置元素
> - WITHCOORD: 将位置元素的经度和维度也一并返回。
> - WITHDIST: 在返回位置元素的同时， 将位置元素与中心之间的距离也一并返回
> - WITHHASH: 以 52 位有符号整数的形式， 返回位置元素经过原始 geohash 编码的有序集合分值。 这个选项主要用于底层应用或者调试， 实际中的作用并不大。
> - COUNT 限定返回的记录数
> - ASC: 查找结果根据距离从近到远排序。
> - DESC: 查找结果根据从远到近排序。

```
127.0.0.1:6379> georadius sicily 15 37 200 km withdist
1) 1) "Palermo"
   2) "190.4424"
2) 1) "Catania"
   2) "56.4413"
127.0.0.1:6379> georadius sicily 15 37 200 km withcoord
1) 1) "Palermo"
   2) 1) "13.361389338970184 33"
      2) "38.115556395496299"
2) 1) "Catania"
   2) 1) "15.087267458438873 29"
      2) "37.502668423331620 32"
127.0.0.1:6379> georadius sicily 15 37 200 km withcoord withdist
1) 1) "Palermo"
   2) "190.4424"
   3) 1) "13.361389338970184 33"
      2) "38.115556395496299"
2) 1) "Catania"
   2) "56.4413"
   3) 1) "15.087267458438873 29"
      2) "37.502668423331620 32"
```

georadiusbymember

```
GEORADIUSBYMEMBER key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]
```

> georadiusbymember 和 GEORADIUS 命令一样， 都可以找出位于指定范围内的元素， 但是 georadiusbymember 的中心点是由给定的位置元素决定的， 而不是使用经度和纬度来决定中心点

```
127.0.0.1:6379> georadiusbymember sicily Palermo 200 km withdist
1) 1) "Palermo"
   2) "0.0000"
2) 1) "Catania"
   2) "166.2742"
127.0.0.1:6379>
```

geohash

```
geohash key member [member ...]
```

> 用于获取一个或多个位置元素的geohash值
>
> Redis GEO使用geohash来保存地理位置的坐标

```
127.0.0.1:6379> geohash sicily Palermo
1) "sqc8b49rny0"
127.0.0.1:6379>
```

总结

> - 常适用于附近的人
> - 打车软件等应用

Redis Cluster

概念

- Redis集群提供了一种运行redis安装的方法，在该安装中，数据会自动分片到多个Redis节点上
- Redis集群在分区期间也提供了一定程度的可用性，即在某个节点出现故障时。
- Redis集群的好处
    - 自动分片的能力
    - 高可用的实现，即出现故障时业务还能继续正常操作

TCP 端口

- 每个Redis集群节点要求打开两个端口
- 给redis客户端使用的端口 默认 6379
- 数据端口加10000得到的端口，示例中为16379。
- 16379端口被用于集群总线，即节点与节点间通过二进制协议的通信
- 集群总线被节点用于故障检测、配置更新、故障转移授权等。

集群数据分片规则

Hash Slot

- Redis没有使用一致性hash算法，而是一种不同形式的切分，其中每个键在概念上都是我们称为的hash槽的一部分
- 在Redis集群中总共有16384个hash槽，并且使用CRC16算法来计算每个key放在哪个hash槽中
- 举例：假设集群有三个节点 NodeA NodeB NodeC
    - NodeA 的hash槽的范围在0 -5500
    - NodeB的hash槽的范围在5501-11000
    - NodeC的hash槽的范围在11001-16384
- 在集群中移除和增加节点时非常简单的
    - 比如我们添加一个新节点D，我们需要移动A B C三个节点的一些hash槽到D节点
    - 相似的如果我们要移除A节点，我们只要移动A的hash槽到B C即可
    - 将hash槽从一个节点移动到另一个节点是不需要停止任何操作的
    - 添加和移除节点或者改变节点持有hash的百分比也是不需要停机的
- Redis集群支持多个键操作，只要参与单个命令的所有key属于同一个hash槽。
    - 用户可以通过使用名为Hash Tags的概念强制多个key成为同一个hash槽的一部分
    - 通过{hashTag} key形式将多个key强制成为同一个hash槽的一部分

Redis 主从

- 为了保证集群的可用性，Redis集群使用了主从模型，主节点的每个hash槽都有一个到N个副本

Redis集群一致性

- Redis集群不能保证强一致性，这就意味着在某些情况下，redis集群可能会丢失向客户端确认的写操作。
- redis集群丢失写操作因为它是一个异步复制，这就意味着写操作会发生下面情况
    - 向master B写入数据
    - master B向客户端答复OK
    - master B将写操作备份到B1 B2 B3

> - 可以看出，在B向客户端答复之前没有等到从B1 B2 B3的确认，因为这对Redis来说是一个禁止性的延迟惩罚，所以如果你的客户端写了一些东西，B会确认写操作，但是在能够将写操作发送到它的从属服务器之前崩溃了，那么其中一个从属服务器（没有收到写操作）可以升级到主服务器，永远丢失写操作。
> - Redis Cluster在绝对需要时支持同步写入，通过WAIT命令实现。这就大大降低了失败的可能性。但是，请注意，即使使用了同步复制，Redis Cluster也不能实现强一致性：在更复杂的故障场景下，始终有可能将无法接收写入的从设备选为主设备。
> - 另一个值得注意的场景是Redis集群将丢失写操作，这种情况发生在一个网络分区期间，其中一个客户机与少数实例（至少包括一个主实例）隔离

Redis集群配置

参数解释

> - **cluster-enabled <yes/no>**： yes开启redis集群支持，no就是单节点启动
> - **cluster-config-file** ： 请注意，尽管有此选项的名称，但这不是用户可编辑的配置文件，而是Redis集群节点在每次发生更改时自动保存集群配置（基本上是状态）的文件，以便能够在启动时重新读取。该文件列出了集群中的其他节点、它们的状态、持久变量等等。通常，由于接收到一些消息，此文件会被重写并刷新到磁盘上
> - **cluster-node-timeout** ： Redis群集节点不可用的最长时间，而不被视为失败。如果主节点在超过指定的时间内不可访问，则其从属节点将进行故障转移。此参数控制Redis集群中的其他重要内容。值得注意的是，在指定的时间内无法到达大多数主节点的每个节点都将停止接受查询。
> - **cluster-slave-validity-factor** ： 如果设置为零，从属者将始终认为自己是有效的，因此总是尝试故障转移主机，而不管主和从机之间的链路保持断开的时间量无关。如果该值为正，则计算最大断开连接时间，即节点超时值乘以此选项提供的系数，如果节点是从属节点，则如果主链路断开连接的时间超过指定的时间，则不会尝试启动故障转移。例如，如果节点超时设置为5秒，有效性因子设置为10，则从节点与主节点断开连接超过50秒的从节点将不会尝试故障转移其主节点。请注意，如果没有能够进行故障转移的从属服务器，任何不同于零的值都可能导致Redis集群在主服务器故障后不可用。在这种情况下，只有当原始主机重新加入集群时，集群才会恢复可用
> - **cluster-migration-barrier** ： 主服务器将保持与之连接的最小从属数量，以便另一个从属机迁移到不再由任何从属服务器覆盖的主服务器
> - **cluster-require-full-coverage <yes/no>**： 如果设置为yes（默认情况下是这样），那么如果任何节点都没有覆盖一定百分比的密钥空间，集群将停止接受写操作。如果该选项设置为no，则即使只能处理关于密钥子集的请求，集群仍将提供查询
> - **cluster-allow-reads-when-down <`yes/no`>**： 如果设置为no（默认情况下），Redis集群中的节点将在集群标记为failed（失败）时停止服务所有流量，无论是节点无法达到主节点的仲裁，还是未达到完全覆盖率。这可以防止从不知道集群中的更改的节点读取可能不一致的数据。可以将此选项设置为“是”，以允许在失败状态下从节点进行读取，这对于希望优先考虑读取可用性但仍希望防止不一致写入的应用程序很有用。它还可以用于只有一个或两个碎片的Redis集群，因为它允许节点在主节点出现故障时继续提供写操作，但无法进行自动故障切换。

创建和使用redis集群

- *创建文件夹*

```
mkdir 7000 7001 7002 7003 7004 7005
```

- *编辑配置文件*

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

> 分别在不同文件夹编辑配置文件，**注意**：`port`需要配置不同，这里port等于文件夹的名字，方便测试

- *启动redis server*

```
#进入不同目录启动详情的reids实例
cd 7000
../redis-server ./redis.conf
```

- 创建集群

```
redis-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 \
--cluster-replicas 1
```

> cluster-replicas： 表示每个master的slave的数量，这里1表示每个master有1个slave