# 目录

# 第1章 Redis for Python 开发手册

## 1.1 redis 基本命令 String

### 1.1.1 Set

```
set(name, value, ex=None, px=None, nx=False, xx=False)
在 Redis 中设置值，默认，不存在则创建，存在则修改
```

参数说明：

ex，过期时间（秒）

px，过期时间（毫秒）

nx，如果设置为 True，则只有 name 不存在时，当前 set 操作才执行

xx，如果设置为 True，则只有 name 存在时，当前 set 操作才执行

- ex，过期时间（秒） 这里过期时间是 3 秒，3 秒后 p，键 food 的值就变成 None

```
import redis

pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)
r.set('food', 'mutton', ex=3)       # key 是"food" value 是"mutton"  将键值对存入 redis 缓存
print(r.get('food'))    # mutton  取出键 food 对应的值
```

- px，过期时间（豪秒） 这里过期时间是 3 豪秒，3 毫秒后，键 foo 的值就变成 None

```
import redis

pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)
r.set('food', 'beef', px=3)
print(r.get('food'))
```

- nx，如果设置为 True，则只有 name 不存在时，当前 set 操作才执行 （新建）

```
import redis

pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)
print(r.set('fruit', 'watermelon', nx=True))       # True--不存在
# 如果键 fruit 不存在，那么输出是 True；如果键 fruit 已经存在，输出是 None
```

```
❑ xx，如果设置为 True，则只有 name 存在时，当前 set 操作才执行 （修改）
print((r.set('fruit', 'watermelon', xx=True)))    # True--已经存在
# 如果键 fruit 已经存在，那么输出是 True；如果键 fruit 不存在，输出是 None
❑ setnx(name, value)
设置值，只有 name 不存在时，执行设置操作（添加）
print(r.setnx('fruit1', 'banana'))  # fruit1 不存在，输出为 True
```

### 1.1.2 setex

```
setex(name, value, time)
设置值
```

参数：

```
time，过期时间（数字秒 或 timedelta 对象）
import redis
import time


pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)
r.setex("fruit2", "orange", 5)
time.sleep(5)
print(r.get('fruit2'))  # 5 秒后，取值就从 orange 变成 None
```

### 1.1.3 psetex

```
psetex(name, time_ms, value)
设置值
```

参数：

```
time_ms，过期时间（数字毫秒 或 timedelta 对象）
r.psetex("fruit3", 5000, "apple")
time.sleep(5)
print(r.get('fruit3'))  # 5000 毫秒后，取值就从 apple 变成 None
8.mset(args, *kwargs)
```

批量设置值

如：

```
r.mget({'k1': 'v1', 'k2': 'v2'})
```

```
r.mset(k1="v1", k2="v2") # 这里 k1 和 k2 不能带引号 一次设置对个键值对
print(r.mget("k1", "k2"))    # 一次取出多个键对应的值
print(r.mget("k1"))
```

### 1.1.4 mget

mget(keys, *args)

批量获取

如：

```
print(r.mget('k1', 'k2'))
print(r.mget(['k1', 'k2']))
print(r.mget("fruit", "fruit1", "fruit2", "k1", "k2"))  # 将目前 redis 缓存中的键对应的值批量取出来
```

### 1.1.5 getset

getset(name, value)

设置新值并获取原来的值

```
print(r.getset("food", "barbecue"))  # 设置的新值是 barbecue 设置前的值是 beef1
```

### 1.1.6 getrange

```
getrange(key, start, end)
获取子序列（根据字节获取，非字符）
```

参数：

```
name，Redis 的 name
start，起始位置（字节）
end，结束位置（字节）
如： “君惜大大” ， 0-3 表示 “君”
r.set("cn_name", "君惜大大") # 汉字
print(r.getrange("cn_name", 0, 2))    # 取索引号是 0-2 前 3 位的字节 君 切片操作 （一个汉字 3 个字节 1 个字母一个字节 每个字节 8bit）
print(r.getrange("cn_name", 0, -1))  # 取所有的字节 君惜大大 切片操作
r.set("en_name","junxi") # 字母
print(r.getrange("en_name", 0, 2))  # 取索引号是 0-2 前 3 位的字节 jun 切片操作 （一个汉字 3 个字节 1 个字母一个字节 每个字节 8bit）
print(r.getrange("en_name", 0, -1)) # 取所有的字节 junxi 切片操作
```

### 1.1.7 setrange

setrange(name, offset, value)

修改字符串内容，从指定字符串索引开始向后替换（新值太长时，则向后添加）

参数：

offset，字符串的索引，字节（一个汉字三个字节）

value，要设置的值

r.setrange("en_name", 1, "ccc")

print(r.get("en_name"))　　# jccci 原始值是 junxi 从索引号是 1 开始替换成 ccc 变成 jccci2

### 1.1.8 setbit

setbit(name, offset, value)

对 name 对应值的二进制表示的位进行操作

参数：

name，redis 的 name

offset，位的索引（将值变换成二进制后再进行索引）

value，值只能是 1 或 0

注：如果在 Redis 中有一个对应： n1 = "foo"，

那么字符串 foo 的二进制表示为：01100110 01101111 01101111

所以，如果执行 setbit('n1', 7, 1)，则就会将第 7 位设置为 1，

那么最终二进制则变成 01100111 01101111 01101111，即："goo"

扩展，转换二进制表示：

source = "郭加磊"

source = "foo"

for i in source:

num = ord(i)

print bin(num).replace('b','')

特别的，如果 source 是汉字 ″郭加磊″怎么办？

答：对于 utf-8，每一个汉字占 3 个字节，那么 "郭加磊" 则有 9 个字节

对于汉字，for 循环时候会按照 字节 迭代，那么在迭代时，将每一个字节转换 十进制数，然后再将十进制数转换成二进制

11100110 10101101 10100110 11100110 10110010 10011011 11101001 10111101 10010000

### 1.1.9 getbit

getbit(name, offset)

获取 name 对应的值的二进制表示中的某位的值 （0 或 1）

```
print(r.getbit("foo1", 0)) # 0 foo1 对应的二进制 4 个字节 32 位 第 0 位是 0 还是 1
```

### 1.1.10 bitcount

```
bitcount(key, start=None, end=None)
获取 name 对应的值的二进制表示中 1 的个数
```

参数：

```
key，Redis 的 name
start 字节起始位置
end，字节结束位置
print(r.get("foo"))  # goo1 01100111
print(r.bitcount("foo",0,1))  # 11 表示前 2 个字节中，1 出现的个数
```

### 1.1.11 bittop

```
bitop(operation, dest, *keys)
获取多个值，并将值做位运算，将最后的结果保存至新的 name 对应的值
```

参数：

operation,AND（并） 、 OR（或） 、 NOT（非） 、 XOR（异或）

dest, 新的 Redis 的 name

*keys,要查找的 Redis 的 name

如：

```
bitop("AND", 'new_name', 'n1', 'n2', 'n3')
获取 Redis 中 n1,n2,n3 对应的值，然后讲所有的值做位运算（求并集），然后将结果保存
new_name 对应的值中
r.set("foo","1")  # 0110001
r.set("foo1","2")  # 0110010
print(r.mget("foo","foo1"))  # ['goo1', 'baaanew']
print(r.bitop("AND","new","foo","foo1"))  # "new" 0 0110000
print(r.mget("foo","foo1","new"))

source = "12"
for i in source:
num = ord(i)
print(num)  # 打印每个字母字符或者汉字字符对应的 ascii 码值 f-102-0b100111-01100111
print(bin(num))  # 打印每个 10 进制 ascii 码值转换成二进制的值 0b1100110（0b 表示二进
```

```
制）
print bin(num).replace('b','')   # 将二进制 0b1100110 替换成 01100110
```

### 1.1.12 strlen

```
strlen(name)
返回 name 对应值的字节长度（一个汉字 3 个字节）
print(r.strlen("foo"))   # 4 'goo1'的长度是 4
```

### 1.1.13 incr

incr(self, name, amount=1)

自增 name 对应的值，当 name 不存在时，则创建 name＝amount，否则，则自增。

参数：

```
name,Redis 的 name
amount,自增数（必须是整数）
注：同 incrby
r.set("foo", 123)
print(r.mget("foo", "foo1", "foo2", "k1", "k2"))
r.incr("foo", amount=1)
print(r.mget("foo", "foo1", "foo2", "k1", "k2"))
```

应用场景 - 页面点击数

```
假定我们对一系列页面需要记录点击次数。例如论坛的每个帖子都要记录点击次数，而点击次数比回帖的次数的多得多。如果使用关系数据库来存储点击，可能存在大量的行级锁争用。所以，点击数的增加使用 redis 的 INCR 命令最好不过了。
当 redis 服务器启动时，可以从关系数据库读入点击数的初始值（12306 这个页面被访问了 34634 次）
r.set("visit:12306:totals", 34634)
print(r.get("visit:12306:totals"))
```

每当有一个页面点击，则使用 INCR 增加点击数即可。

```
r.incr("visit:12306:totals")
r.incr("visit:12306:totals")
```

页面载入的时候则可直接获取这个值

```
print(r.get("visit:12306:totals"))
```

### 1.1.14 incrfloat

```
incrbyfloat(self, name, amount=1.0)
自增 name 对应的值，当 name 不存在时，则创建 name＝amount，否则，则自增。
```

参数：

```
name,Redis 的 name
amount,自增数（浮点型）
r.set("foo1", "123.0")
r.set("foo2", "221.0")
print(r.mget("foo1", "foo2"))
r.incrbyfloat("foo1", amount=2.0)
r.incrbyfloat("foo2", amount=3.0)
print(r.mget("foo1", "foo2"))
```

### 1.1.15 decr

```
decr(self, name, amount=1)
自减 name 对应的值，当 name 不存在时，则创建 name＝amount，否则，则自减。
```

参数：

```
name,Redis 的 name
amount,自减数（整数)
r.decr("foo4", amount=3) #  递减 3
r.decr("foo1", amount=1) #  递减 1
print(r.mget("foo1", "foo4"))
```

### 1.1.16 append

```
append(key, value)
在 redis name 对应的值后面追加内容
```

参数：

```
key, redis 的 name
value,  要追加的字符串
r.append("name", "haha")       # 在 name 对应的值 junxi 后面追加字符串 haha
print(r.mget("name"))
```

## 1.2 hash 操作

### 1.2.1 hset

1.单个增加–修改(单个取出)–没有就新增，有的话就修改

hset(name, key, value)

name 对应的 hash 中设置一个键值对（不存在，则创建；否则，修改）

参数：

name，redis 的 name

key，name 对应的 hash 中的 key

value，name 对应的 hash 中的 value

注：

hsetnx(name, key, value),当 name 对应的 hash 中不存在当前 key 时则创建（相当于添加）

```
import redis
import time


pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)


r.hset("hash1", "k1", "v1")
r.hset("hash1", "k2", "v2")
print(r.hkeys("hash1")) # 取 hash 中所有的 key
print(r.hget("hash1", "k1"))       # 单个取 hash 的 key 对应的值
print(r.hmget("hash1", "k1", "k2")) # 多个取 hash 的 key 对应的值
r.hsetnx("hash1", "k2", "v3") # 只能新建
print(r.hget("hash1", "k2"))
```

### 1.2.2 hmset

批量增加（取出）

hmset(name, mapping)

在 name 对应的 hash 中批量设置键值对

参数：

name，redis 的 name

mapping，字典，如：{‘k1’:’v1’, ‘k2’: ‘v2’}

如：

```
r.hmset("hash2", {"k2": "v2", "k3": "v3"})
```

hget(name,key)
在 name 对应的 hash 中获取根据 key 获取 value

### 1.2.3 hmget

hmget(name, keys, *args)
在 name 对应的 hash 中获取多个 key 的值
参数：

name，reids 对应的 name
keys，要获取 key 集合，如：[‘k1’, ‘k2’, ‘k3’]
*args，要获取的 key，如：k1,k2,k3

如：

```
print(r.hget("hash2", "k2"))  # 单个取出"hash2"的 key-k2 对应的 value
print(r.hmget("hash2", "k2", "k3"))  # 批量取出"hash2"的 key-k2 k3 对应的 value --方式 1
print(r.hmget("hash2", ["k2", "k3"]))  # 批量取出"hash2"的 key-k2 k3 对应的 value
```

### 1.2.4 hgetall

取出所有的键值对
hgetall(name)
获取 name 对应 hash 的所有键值
print(r.hgetall("hash1"))

### 1.2.5 hlen

得到所有键值对的格式 hash 长度
hlen(name)
获取 name 对应的 hash 中键值对的个数
print(r.hlen("hash1"))

### 1.2.6 hkeys

得到所有的 keys（类似字典的取所有 keys）
hkeys(name)
获取 name 对应的 hash 中所有的 key 的值
print(r.hkeys("hash1"))

### 1.2.7 hvals

```
得到所有的 value（类似字典的取所有 value）
hvals(name)
获取 name 对应的 hash 中所有的 value 的值
print(r.hvals("hash1"))
```

### 1.2.8 hexists

```
判断成员是否存在（类似字典的 in）
hexists(name, key)
检查 name 对应的 hash 是否存在当前传入的 key
print(r.hexists("hash1", "k4"))   # False  不存在
print(r.hexists("hash1", "k1"))   # True  存在
```

### 1.2.9 hdel

```
删除键值对
hdel(name,*keys)
将 name 对应的 hash 中指定 key 的键值对删除
print(r.hgetall("hash1"))
r.hset("hash1", "k2", "v222")    #  修改已有的 key k2
r.hset("hash1", "k11", "v1")    #  新增键值对  k11
r.hdel("hash1", "k1")      #  删除一个键值对
print(r.hgetall("hash1"))
```

### 1.2.10 hincrby

```
自增自减整数(将 key 对应的 value–整数  自增 1 或者 2，或者别的整数  负数就是自减)
hincrby(name, key, amount=1)
自增 name 对应的 hash 中的指定 key 的值，不存在则创建 key=amount
```

参数：

```
name，redis 中的 name
key，  hash 对应的 key
amount，自增数（整数）
r.hset("hash1", "k3", 123)
r.hincrby("hash1", "k3", amount=-1)
print(r.hgetall("hash1"))
```

```
r.hincrby("hash1", "k4", amount=1)  # 不存在的话，value 默认就是 1
print(r.hgetall("hash1"))
```

### 1.2.11 hincrbyfloat

```
自增自减浮点数(将 key 对应的 value–浮点数 自增 1.0 或者 2.0，或者别的浮点数 负数就是自减)
hincrbyfloat(name, key, amount=1.0)
自增 name 对应的 hash 中的指定 key 的值，不存在则创建 key=amount
```

参数：

```
name，redis 中的 name
key， hash 对应的 key
amount，自增数（浮点数）
自增 name 对应的 hash 中的指定 key 的值，不存在则创建 key=amount
r.hset("hash1", "k5", "1.0")
r.hincrbyfloat("hash1", "k5", amount=-1.0)      # 已经存在，递减-1.0
print(r.hgetall("hash1"))
r.hincrbyfloat("hash1", "k6", amount=-1.0)      # 不存在，value 初始值是-1.0 每次递减 1.0
print(r.hgetall("hash1"))
```

### 1.2.12 hscan

```
取值查看–分片读取
hscan(name, cursor=0, match=None, count=None)
增量式迭代获取，对于数据大的数据非常有用，hscan 可以实现分片的获取数据，并非一次性将数据全部获取完，从而放置内存被撑爆
```

参数：

```
name，redis 的 name
cursor，游标（基于游标分批取获取数据）
match，匹配指定 key，默认 None 表示所有的 key
count，每次分片最少获取个数，默认 None 表示采用 Redis 的默认分片个数
```

如：

```
第一次：cursor1, data1 = r.hscan('xx', cursor=0, match=None, count=None)
第二次：cursor2, data1 = r.hscan('xx', cursor=cursor1, match=None, count=None)
…
直到返回值 cursor 的值为 0 时，表示数据已经通过分片获取完毕
print(r.hscan("hash1"))
```

### 1.2.13 hscan_iter

```
hscan_iter(name, match=None, count=None)
利用 yield 封装 hscan 创建生成器，实现分批去 redis 中获取数据
```

参数：

```
match，匹配指定 key，默认 None 表示所有的 key
count，每次分片最少获取个数，默认 None 表示采用 Redis 的默认分片个数
如：
for item in r.hscan_iter('hash1'):
    print(item)
print(r.hscan_iter("hash1"))    # 生成器内存地址
```

## 1.3 list 列表操作

### 1.3.1 lpush

增加（类似于 list 的 append，只是这里是从左边新增加）- 没有就新建

lpush(name,values)

在 name 对应的 list 中添加元素，每个新的元素都添加到列表的最左边

如：

```
import redis
import time

pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)

r.lpush("list1", 11, 22, 33)
print(r.lrange('list1', 0, -1))
保存顺序为: 33,22,11
```

扩展：

```
r.rpush("list2", 11, 22, 33)  # 表示从右向左操作
print(r.llen("list2"))  # 列表长度
print(r.lrange("list2", 0, 3))  # 切片取出值，范围是索引号 0-3
```

### 1.3.2 rpush

```
增加（从右边增加）–没有就新建
r.rpush("list2", 44, 55, 66)    # 在列表的右边，依次添加 44,55,66
```

```
print(r.llen("list2"))   # 列表长度
print(r.lrange("list2", 0, -1)) # 切片取出值，范围是索引号 0 到-1(最后一个元素)
```

### 1.3.3 lpushx

```
往已经有的 name 的列表的左边添加元素，没有的话无法创建
lpushx(name,value)
在 name 对应的 list 中添加元素，只有 name 已经存在时，值添加到列表的最左边
```

更多：

```
r.lpushx("list10", 10)    # 这里 list10 不存在
print(r.llen("list10"))   # 0
print(r.lrange("list10", 0, -1))   # []
r.lpushx("list2", 77)    # 这里"list2"之前已经存在，往列表最左边添加一个元素，一次只能添加一个
print(r.llen("list2"))   # 列表长度
print(r.lrange("list2", 0, -1)) # 切片取出值，范围是索引号 0 到-1(最后一个元素
```

### 1.3.4 rpushx

```
往已经有的 name 的列表的右边添加元素，没有的话无法创建
r.rpushx("list2", 99)    # 这里"foo_list1"之前已经存在，往列表最右边添加一个元素，一次只能添加一个
print(r.llen("list2"))   # 列表长度
print(r.lrange("list2", 0, -1)) # 切片取出值，范围是索引号 0 到-1(最后一个元素)
```

### 1.3.5 linsert

```
新增（固定索引号位置插入元素）
linsert(name, where, refvalue, value))
在 name 对应的列表的某一个值前或后插入一个新值
```

参数：

```
name，redis 的 name
where，BEFORE 或 AFTER
refvalue，标杆值，即：在它前后插入数据
value，要插入的数据
r.linsert("list2", "before", "11", "00")    # 往列表中左边第一个出现的元素"11"前插入元素"00"
```

```
print(r.lrange("list2", 0, -1))     # 切片取出值，范围是索引号 0-最后一个元素
```

### 1.3.6 lset

```
修改（指定索引号进行修改）
r.lset(name, index, value)
对 name 对应的 list 中的某一个索引位置重新赋值
```

参数：

```
name，redis 的 name
index，list 的索引位置
value，要设置的值
r.lset("list2", 0, -11)       # 把索引号是 0 的元素修改成-11
print(r.lrange("list2", 0, -1))
```

### 1.3.7 lrem

```
删除（指定值进行删除）
r.lrem(name, value, num)
在 name 对应的 list 中删除指定的值
```

参数：

```
name，redis 的 name
value，要删除的值
num， num=0，删除列表中所有的指定值；
num=2,从前到后，删除 2 个； num=1,从前到后，删除左边第 1 个
num=-2,从后向前，删除 2 个
r.lrem("list2", "11", 1)       # 将列表中左边第一次出现的"11"删除
print(r.lrange("list2", 0, -1))
r.lrem("list2", "99", -1)       # 将列表中右边第一次出现的"99"删除
print(r.lrange("list2", 0, -1))
r.lrem("list2", "22", 0)       # 将列表中所有的"22"删除
print(r.lrange("list2", 0, -1))
```

### 1.3.8 lpop

```
删除并返回
lpop(name)
在 name 对应的列表的左侧获取第一个元素并在列表中移除，返回值则是第一个元素
```

更多：

```
rpop(name)  表示从右向左操作
r.lpop("list2")      # 删除列表最左边的元素，并且返回删除的元素
print(r.lrange("list2", 0, -1))
r.rpop("list2")      # 删除列表最右边的元素，并且返回删除的元素
print(r.lrange("list2", 0, -1))
```

### 1.3.9 ltrim

```
删除索引之外的值
ltrim(name, start, end)
在 name 对应的列表中移除没有在 start-end 索引之间的值
```

参数：

```
name，redis 的 name
start，索引的起始位置
end，索引结束位置
r.ltrim("list2", 0, 2)      # 删除索引号是 0-2 之外的元素，值保留索引号是 0-2 的元素
print(r.lrange("list2", 0, -1))
```

### 1.3.10 lindex

```
取值（根据索引号取值）
lindex(name, index)
在 name 对应的列表中根据索引获取列表元素
print(r.lindex("list2", 0))   # 取出索引号是 0 的值
```

### 1.3.11 rpoplpush

```
移动  元素从一个列表移动到另外一个列表
rpoplpush(src, dst)
从一个列表取出最右边的元素，同时将其添加至另一个列表的最左边
```

参数：

```
src，要取数据的列表的 name
dst，要添加数据的列表的 name
r.rpoplpush("list1", "list2")
print(r.lrange("list2", 0, -1))
```

### 1.3.12 brpoplpush

移动 元素从一个列表移动到另外一个列表 可以设置超时

brpoplpush(src, dst, timeout=0)

从一个列表的右侧移除一个元素并将其添加到另一个列表的左侧

参数：

src，取出并要移除元素的列表对应的 name

dst，要插入元素的列表对应的 name

timeout，当 src 对应的列表中没有数据时，阻塞等待其有数据的超时时间（秒），0 表示永远阻塞

```
r.brpoplpush("list1", "list2", timeout=2)
print(r.lrange("list2", 0, -1))
```

### 1.3.13 blpop

一次移除多个列表

blpop(keys, timeout)

将多个列表排列，按照从左到右去 pop 对应列表的元素

参数：

keys，redis 的 name 的集合

timeout，超时时间，当元素所有列表的元素获取完之后，阻塞等待列表内有数据的时间（秒），0 表示永远阻塞

更多：

r.brpop(keys, timeout) 同 blpop，将多个列表排列,按照从右像左去移除各个列表内的元素

```
r.lpush("list10", 3, 4, 5)
r.lpush("list11", 3, 4, 5)
while True:
    r.blpop(["list10", "list11"], timeout=2)
    print(r.lrange("list10", 0, -1), r.lrange("list11", 0, -1))
```

## 1.4 自定义增量迭代

由于 redis 类库中没有提供对列表元素的增量迭代，如果想要循环 name 对应的列表的所有元素，那么就需要：

获取 name 对应的所有列表

循环列表

但是，如果列表非常大，那么就有可能在第一步时就将程序的内容撑爆，所有有必要自定义一个增量迭代的功能：

```
def list_iter(name):
    """
    自定义 redis 列表增量迭代
    :param name: redis 中的 name，即：迭代 name 对应的列表
    :return: yield 返回 列表元素
    """
    list_count = r.llen(name)
    for index in range(list_count):
        yield r.lindex(name, index)

# 使用
for item in list_iter('list2'): # 遍历这个列表
    print(item)
```

## 1.5 集合 set 操作

### 1.5.1 sadd

新增

```
sadd(name,values)
name 对应的集合中添加元素
r.sadd("set1", 33, 44, 55, 66)  # 往集合中添加元素
print(r.scard("set1"))  # 集合的长度是 4
print(r.smembers("set1"))  # 获取集合中所有的成员
```

### 1.5.2 scard

```
获取元素个数 类似于 len
scard(name)
获取 name 对应的集合中元素个数
print(r.scard("set1"))  # 集合的长度是 4
```

### 1.5.3 smembers

```
获取集合中所有的成员
smembers(name)
获取 name 对应的集合的所有成员
print(r.smembers("set1"))  # 获取集合中所有的成员
```

```
获取集合中所有的成员–元组形式
sscan(name, cursor=0, match=None, count=None)
print(r.sscan("set1"))
1
获取集合中所有的成员–迭代器的方式
sscan_iter(name, match=None, count=None)
同字符串的操作，用于增量迭代分批获取元素，避免内存消耗太大
for i in r.sscan_iter("set1"):
    print(i)2
```

### 1.5.4 sdiff

差集

```
sdiff(keys, *args)
在第一个 name 对应的集合中且不在其他 name 对应的集合的元素集合
r.sadd("set2", 11, 22, 33)
print(r.smembers("set1"))    # 获取集合中所有的成员
print(r.smembers("set2"))
print(r.sdiff("set1", "set2"))    # 在集合 set1 但是不在集合 set2 中
print(r.sdiff("set2", "set1"))    # 在集合 set2 但是不在集合 set1 中
```

### 1.5.5 sdiffstore

```
差集–差集存在一个新的集合中
sdiffstore(dest, keys, *args)
获取第一个 name 对应的集合中且不在其他 name 对应的集合，再将其新加入到 dest 对应的集合中
r.sdiffstore("set3", "set1", "set2")      # 在集合 set1 但是不在集合 set2 中
print(r.smembers("set3"))    # 获取集合 3 中所有的成员
```

### 1.5.6 sinter

交集

```
sinter(keys, *args)
获取多一个 name 对应集合的交集
print(r.sinter("set1", "set2")) # 取 2 个集合的交集
```

### 1.5.7 sinterstore

```
交集–交集存在一个新的集合中
sinterstore(dest, keys, *args)
获取多一个 name 对应集合的并集，再将其加入到 dest 对应的集合中
print(r.sinterstore("set3", "set1", "set2")) #  取 2 个集合的交集
print(r.smembers("set3"))
```

### 1.5.8 sunion

并集

```
sunion(keys, *args)
获取多个 name 对应的集合的并集
print(r.sunion("set1", "set2")) #  取 2 个集合的并集 1
并集–并集存在一个新的集合
sunionstore(dest,keys, *args)
获取多一个 name 对应的集合的并集，并将结果保存到 dest 对应的集合中
print(r.sunionstore("set3", "set1", "set2")) #  取 2 个集合的并集
print(r.smembers("set3"))
```

### 1.5.9 sismember

判断是否是集合的成员 类似 in

```
sismember(name, value)
检查 value 是否是 name 对应的集合的成员，结果为 True 和 False
print(r.sismember("set1", 33))    # 33 是集合的成员
print(r.sismember("set1", 23))    # 23 不是集合的成员
```

### 1.5.10 smove

移动

```
smove(src, dst, value)
将某个成员从一个集合中移动到另外一个集合
r.smove("set1", "set2", 44)
print(r.smembers("set1"))
```

```
print(r.smembers("set2"))
```

### 1.5.11 spop

```
删除–随机删除并且返回被删除值
spop(name)
从集合移除一个成员，并将其返回,说明一下，集合是无序的，所有是随机删除的
print(r.spop("set2"))    # 这个删除的值是随机删除的，集合是无序的
print(r.smembers("set2"))
```

11. 删除 - 指定值删除

```
srem(name, values)
在 name 对应的集合中删除某些值
print(r.srem("set2", 11))    # 从集合中删除指定值 11
print(r.smembers("set2"))
```

## 1.6 有序集合 sort set 操作

redis 基本命令 有序 set

```
Set 操作，Set 集合就是不允许重复的列表，本身是无序的
有序集合，在集合的基础上，为每元素排序；元素的排序需要根据另外一个值来进行比较，
所以，对于有序集合，每一个元素有两个值，即：值和分数，分数专门用来做排序。
```

### 1.6.1 zadd

新增

```
zadd(name, args, *kwargs)
在 name 对应的有序集合中添加元素
```

如：

```
import redis
import time


pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)


r.zadd("zset1", n1=11, n2=22)
r.zadd("zset2", 'm1', 22, 'm2', 44)
print(r.zcard("zset1")) # 集合长度
```

```
print(r.zcard("zset2")) # 集合长度
print(r.zrange("zset1", 0, -1))     # 获取有序集合中所有元素
print(r.zrange("zset2", 0, -1, withscores=True))     # 获取有序集合中所有元素和分数 2
```

### 1.6.2 zcard

```
获取有序集合元素个数 类似于 len
zcard(name)
获取 name 对应的有序集合元素的数量
print(r.zcard("zset1")) # 集合长度 1
```

### 1.6.3 zrange

```
获取有序集合的所有元素
r.zrange( name, start, end, desc=False, withscores=False, score_cast_func=float)
按照索引范围获取 name 对应的有序集合的元素
```

参数：

```
name，redis 的 name
start，有序集合索引起始位置（非分数）
end，有序集合索引结束位置（非分数）
desc，排序规则，默认按照分数从小到大排序
withscores，是否获取元素的分数，默认只获取元素的值
score_cast_func，对分数进行数据转换的函数
3-1 从大到小排序(同 zrange，集合是从大到小排序的)
zrevrange(name, start, end, withscores=False, score_cast_func=float)
print(r.zrevrange("zset1", 0, -1))       # 只获取元素，不显示分数
print(r.zrevrange("zset1", 0, -1, withscores=True)) # 获取有序集合中所有元素和分数,分数倒
序
3-2 按照分数范围获取 name 对应的有序集合的元素
zrangebyscore(name, min, max, start=None, num=None, withscores=False,
score_cast_func=float)
for i in range(1, 30):
    element = 'n' + str(i)
    r.zadd("zset3", element, i)
print(r.zrangebyscore("zset3", 15, 25)) # # 在分数是 15-25 之间，取出符合条件的元素
print(r.zrangebyscore("zset3", 12, 22, withscores=True))       # 在分数是 12-22 之间，取出符合
```

```
条件的元素（带分数）
3-3 按照分数范围获取有序集合的元素并排序（默认从大到小排序）
zrevrangebyscore(name, max, min, start=None, num=None, withscores=False,
score_cast_func=float)
print(r.zrevrangebyscore("zset3", 22, 11, withscores=True)) # 在分数是 22-11 之间，取出符合条
件的元素 按照分数倒序 1
3-4 获取所有元素–默认按照分数顺序排序
zscan(name, cursor=0, match=None, count=None, score_cast_func=float)
print(r.zscan("zset3"))1
3-5 获取所有元素–迭代器
zscan_iter(name, match=None, count=None,score_cast_func=float)
for i in r.zscan_iter("zset3"): # 遍历迭代器
    print(i)
```

### 1.6.4 zcount

```
zcount(name, min, max)
获取 name 对应的有序集合中分数 在 [min,max] 之间的个数
print(r.zrange("zset3", 0, -1, withscores=True))
print(r.zcount("zset3", 11, 22))
```

### 1.6.5 zincrby

```
自增
zincrby(name, value, amount)
自增 name 对应的有序集合的 name 对应的分数
r.zincrby("zset3", "n2", amount=2)      # 每次将 n2 的分数自增 2
print(r.zrange("zset3", 0, -1, withscores=True))
```

### 1.6.6 zrank

```
获取值的索引号
zrank(name, value)
获取某个值在 name 对应的有序集合中的索引（从 0 开始）
更多：
zrevrank(name, value)，从大到小排序
print(r.zrank("zset3", "n1"))    # n1 的索引号是 0 这里按照分数顺序（从小到大）
print(r.zrank("zset3", "n6"))    # n6 的索引号是 1
```

```
print(r.zrevrank("zset3", "n1"))       # n1 的索引号是 29  这里安照分数倒序（从大到小）
```

### 1.6.7 zrem

```
删除–指定值删除
zrem(name, values)
删除 name 对应的有序集合中值是 values 的成员
r.zrem("zset3", "n3")     #  删除有序集合中的元素 n3  删除单个
print(r.zrange("zset3", 0, -1))
```

### 1.6.8 zremrangebyrank

```
删除–根据排行范围删除，按照索引号来删除
zremrangebyrank(name, min, max)
根据排行范围删除
r.zremrangebyrank("zset3", 0, 1)   #  删除有序集合中的索引号是 0, 1 的元素
print(r.zrange("zset3", 0, -1))
zremrangebyscore(name, min, max)
根据分数范围删除
r.zremrangebyscore("zset3", 11, 22)     #  删除有序集合中的分数是 11-22 的元素
print(r.zrange("zset3", 0, -1))
```

### 1.6.9 zscore

```
获取值对应的分数
zscore(name, value)
获取 name 对应有序集合中  value  对应的分数
print(r.zscore("zset3", "n27"))     #  获取元素 n27 对应的分数 271
```

## 1.7 其他常用操作

### 1.7.1 delete

```
delete (*names)
根据删除 redis 中的任意数据类型（string、hash、list、set、有序 set）
r.delete("gender")   #  删除 key 为 gender 的键值对 1
```

### 1.7.2 exists

检查名字是否存在

exists(name)

检测 redis 的 name 是否存在，存在就是 True，False 不存在

print(r.exists("zset1"))1

### 1.7.3 keys

模糊匹配

keys(pattern='')

根据模型获取 redis 的 name

更多：

KEYS 匹配数据库中所有 key 。

KEYS h?llo 匹配 hello ， hallo 和 hxllo 等。

KEYS h*llo 匹配 hllo 和 heeeello 等。

KEYS h[ae]llo 匹配 hello 和 hallo ，但不匹配 hillo

print(r.keys("foo*"))

1

### 1.7.4 expire

设置超时时间

expire(name ,time)

为某个 redis 的某个 name 设置超时时间

r.lpush("list5", 11, 22)

r.expire("list5", time=3)

print(r.lrange("list5", 0, -1))

time.sleep(3)

print(r.lrange("list5", 0, -1))

### 1.7.5 rname

重命名

rename(src, dst)

对 redis 的 name 重命名

r.lpush("list5", 11, 22)

r.rename("list5", "list5-1")

### 1.7.6 randomkey

```
随机获取 name
randomkey()
随机获取一个 redis 的 name（不删除）
print(r.randomkey())
1
```

### 1.7.7 type

```
获取类型
type(name)
获取 name 对应值的类型
print(r.type("set1"))
print(r.type("hash2"))
```

### 1.7.8 scan

```
查看所有元素
scan(cursor=0, match=None, count=None)
print(r.hscan("hash2"))
print(r.sscan("set3"))
print(r.zscan("zset2"))
print(r.getrange("foo1", 0, -1))
print(r.lrange("list2", 0, -1))
print(r.smembers("set3"))
print(r.zrange("zset3", 0, -1))
print(r.hgetall("hash1"))
```

### 1.7.9 scan_iter

```
查看所有元素–迭代器
scan_iter(match=None, count=None)
for i in r.hscan_iter("hash1"):
    print(i)

for i in r.sscan_iter("set3"):
    print(i)
```

```
for i in r.zscan_iter("zset3"):
    print(i)8
other 方法
print(r.get('name'))      # 查询 key 为 name 的值
r.delete("gender")   # 删除 key 为 gender 的键值对
print(r.keys()) # 查询所有的 Key
print(r.dbsize())    # 当前 redis 包含多少条数据
r.save()     # 执行"检查点"操作，将数据写回磁盘。保存时阻塞
# r.flushdb()            # 清空 r 中的所有数据
```

### 1.7.10 管道（pipeline）

redis 默认在执行每次请求都会创建（连接池申请连接）和断开（归还连接池）一次连接操作，

如果想要在一次请求中指定多个命令，则可以使用 pipline 实现一次请求指定多个命令，并且默认情况下一次 pipline 是原子性操作。

管道（pipeline）是 redis 在提供单个请求中缓冲多条服务器命令的基类的子类。它通过减少服务器-客户端之间反复的 TCP 数据库包，从而大大提高了执行批量命令的功能。

```
import redis
import time

pool = redis.ConnectionPool(host='127.0.0.1', port=6379, decode_responses=True)
r = redis.Redis(connection_pool=pool)

# pipe = r.pipeline(transaction=False)      # 默认的情况下，管道里执行的命令可以保证执行的原子性，执行 pipe = r.pipeline(transaction=False)可以禁用这一特性。
# pipe = r.pipeline(transaction=True)
pipe = r.pipeline() # 创建一个管道

pipe.set('name', 'jack')
pipe.set('role', 'sb')
pipe.sadd('faz', 'baz')
pipe.incr('num')      # 如果 num 不存在则 vaule 为 1，如果存在，则 value 自增 1
pipe.execute()

print(r.get("name"))
```

```
print(r.get("role"))
print(r.get("num"))
```

管道的命令可以写在一起，如：

```
pipe.set('hello', 'redis').sadd('faz', 'baz').incr('num').execute()
print(r.get("name"))
print(r.get("role"))
print(r.get("num"))
```