Python Redis操作

一.Redis连接:

1.1 直连模式

In [1]:
```
import redis
r = redis.Redis(host='172.16.70.251', port=6379)
r.set('foo', 'Bar')
print(r.get('foo'))
```

```
b'Bar'
```

1.2 连接池模式
通过connection pool来管理对一个redis server的所有连接，避免每次建立、释放连接的开销．默认每个Redis实例都会维护一个自己的连接池，可以直接建立一个连接池，然后将连接池作为参数传递给Redis类，这样就可以实现多个Redis实例共享一个连接池．

In [1]:
```
import redis
pool = redis.ConnectionPool(host='172.16.70.251', port=6379)
r = redis.Redis(connection_pool=pool)
r.set('foo', 'Bar')
print(r.get('foo'))
```

```
b'Bar'
```

二. Redis数据操作

2.1 String操作
Redis中的String在内存中按照一个name对应一个value来存储

In [3]:
```
help(r.set)
```

```
Help on method set in module redis.client:

set(name, value, ex=None, px=None, nx=False, xx=False) method of redis.client.Redis instance
    Set the value at key ``name`` to ``value``

    ``ex`` sets an expire flag on key ``name`` for ``ex`` seconds.

    ``px`` sets an expire flag on key ``name`` for ``px`` milliseconds.

    ``nx`` if set to True, set the value at key ``name`` to ``value`` only
        if it does not exist.

    ``xx`` if set to True, set the value at key ``name`` to ``value`` only
        if it already exists.
```

ex: 过期时间(秒) 等价于 setex() 方法
px: 过期时间(毫秒) 等价于 psetex()
nx: 如果设置为true，只有当name不存在时，当前的set操作才执行等价于(setnx)
xx: 如果设置为true，只有当name存在时，当前Set操作才执行.

In [4]:
```
help(r.get) #获取值
```

```
Help on method get in module redis.client:

get(name) method of redis.client.Redis instance
    Return the value at key ``name``, or None if the key doesn't exist
```

批量设置值

In [6]:
```
help(r.mset)
```

```
Help on method mset in module redis.client:

mset(mapping) method of redis.client.Redis instance
    Sets key/values based on a mapping. Mapping is a dictionary of
    key/value pairs. Both keys and values should be strings or types that
    can be cast to a string via str().
```

In [10]:
```
r.mset({"name1":"zhangsan", "name2":"lisi"})
```

Out[10]:
```
True
```

批量获取值

In [11]:
```
help(r.mget)
```

```
Help on method mget in module redis.client:

mget(keys, *args) method of redis.client.Redis instance
    Returns a list of values ordered identically to ``keys``
```

In [12]:
```
r.mget(["name1", "name2"])
```

Out[12]:
```
[b'zhangsan', b'lisi']
```

In [ ]:
```
设置新值打印原值
```

In [13]:
```
help(r.getset)
```

```
Help on method getset in module redis.client:

getset(name, value) method of redis.client.Redis instance
    Sets the value at key ``name`` to ``value``
    and returns the old value at key ``name`` atomically.
```

In [14]:
```
r.getset("name1", "wangwu")
```

Out[14]:
```
b'zhangsan'
```

2.2 Hash操作
Redis中的Hash在内存中类似于一个name对应一个字典来存储

In [15]:
```
help(r.hset)
```

```
Help on method hset in module redis.client:

hset(name, key, value) method of redis.client.Redis instance
    Set ``key`` to ``value`` within hash ``name``
    Returns 1 if HSET created a new field, otherwise 0
```

In [16]:
```
#设置hash
r.hset('dic_name', "tom", "cat")
```

Out[16]: 1

In [18]:
```
#获取hash
r.hget("dic_name", 'tom')
```

Out[18]: b'cat'

In [19]:
```
#获取对应hash的所有键值
r.hgetall("dic_name")
```

Out[19]: {b'tom': b'cat'}

In [20]:
```
#在指定name的hash中批量设置键值对
dic = {"x": 1, "y": 2, "z": 3}
r.hmset("dic_name", dic)
```

Out[20]: True

In [21]:
```
#在指定name对应的hash中获取多个key值
r.hmget("dic_name", ["tom", "x"])
```

Out[21]: [b'cat', b'1']

In [25]:
```
#hlen(name)获取hash中键值对的个数
#hkeys(name)获取hash中所有key的值
#hvals(name)获取hash中所有的value的值
print(r.hlen("dic_name"))
print(r.hkeys("dic_name"))
print(r.hvals("dic_name"))
```

```
4
[b'tom', b'x', b'y', b'z']
[b'cat', b'1', b'2', b'3']
```

In [27]:
```
#检查指定hash中是否存在指定的key
print(r.hexists("dic_name", "x"))
print(r.hexists("dic_name", "w"))
```

```
True
False
```

In [28]:
```
#删除指定hash中的键值对
r.hdel("dic_name", "tom")
```

Out[28]: 1

2.3 List操作
Redis中的List在内存中按照一个name对应一个List来存储

In [ ]:
```
#lpush与rpush从左边或右边向列表中添加元素
#lpop移除列表左侧第一个元素
#llen返回对应的list中的元素个数
```

In [44]:
```
list_count = r.llen("list_name")
for index in range(list_count):
    r.lpop("list_name")
r.lpush("list_name", 1,2,3) #[3,2,1]
r.rpush('list_name', 4,5,6) #[3,2,1,4,5,6]
list_count = r.llen("list_name")
for index in range(list_count):
    print(r.lindex("list_name", index))
```

```
b'3'
b'2'
b'1'
b'4'
b'5'
b'6'
```

In [46]:
```
#对list中的某一个索引位置重新赋值
help(r.lset)
```

```
Help on method lset in module redis.client:

lset(name, index, value) method of redis.client.Redis instance
    Set ``position`` of list ``name`` to ``value``
```

In [47]:
```
r.lset("list_name", 0, "apple")
```

Out[47]: True

In [49]:
```
#根据索引获取列表内的元素
help(r.lindex)
```

```
Help on method lindex in module redis.client:

lindex(name, index) method of redis.client.Redis instance
    Return the item from list ``name`` at position ``index``

    Negative indexes are supported and will return an item at the
    end of the list
```

In [50]:
```
r.lindex("list_name", 0)
```

Out[50]: b'apple'

In [51]:
```
#在列表的某一个值钱或后插入一个新值
help(r.linsert)
```

```
Help on method linsert in module redis.client:

linsert(name, where, refvalue, value) method of redis.client.Redis instance
    Insert ``value`` in list ``name`` either immediately before or after
    [``where``] ``refvalue``

    Returns the new length of the list on success or -1 if ``refvalue``
    is not in the list.
```

In [53]:
```
r.linsert("list_name", "before", 2, "haha")
list_count = r.llen("list_name")
for index in range(list_count):
    print(r.lindex("list_name", index))
```

```
b'apple'
b'haha'
b'haha'
b'2'
b'1'
b'4'
b'5'
b'6'
```

2.4 Set操作
集合就是不允许重复的列表

In [63]:
```
r.sadd("set_name", "aa", "bb", "cc", "dd", "ee")
```

Out[63]: 4

In [55]:
```
#获取集合中的所有成员
r.smembers("set_name")
```

Out[55]: {b'aa', b'bb'}

In [56]:
```
#获取集合中元素的个数
r.scard("set_name")
```

Out[56]: 2

In [58]:
```
#获取在集合A中不在集合B中的元素
r.sadd("A", 1,2,3)
r.sadd("B", 2,3,4)
print(r.sdiff("A", "B"))
```

```
{b'1'}
```

In [60]:
```
#从集合的左侧删除一个元素并将其返回
r.spop("set_name")
```

Out[60]: b'aa'

In [64]:
```
#从集合中随机获取指定数量个元素
r.srandmember("set_name", 2)
```

Out[64]: [b'aa', b'bb']

In [66]:
```
#检查一个元素是否是集合中的元素
help(r.sismember)
print(r.sismember("set_name", "ee"))
```

```
Help on method sismember in module redis.client:

sismember(name, value) method of redis.client.Redis instance
    Return a boolean indicating if ``value`` is a member of set ``name``

True
```

In [67]:
```
#获取多个集合的并集
r.sunion("set_name", "A", "B")
```

Out[67]: {b'1', b'2', b'3', b'4', b'aa', b'bb', b'cc', b'dd', b'ee'}

2.5 其他常用操作

In [ ]:
```
#根据name删除Redis中的任意数据类型
```

In [68]:
```
r.delete("set_name")
```

Out[68]: 1

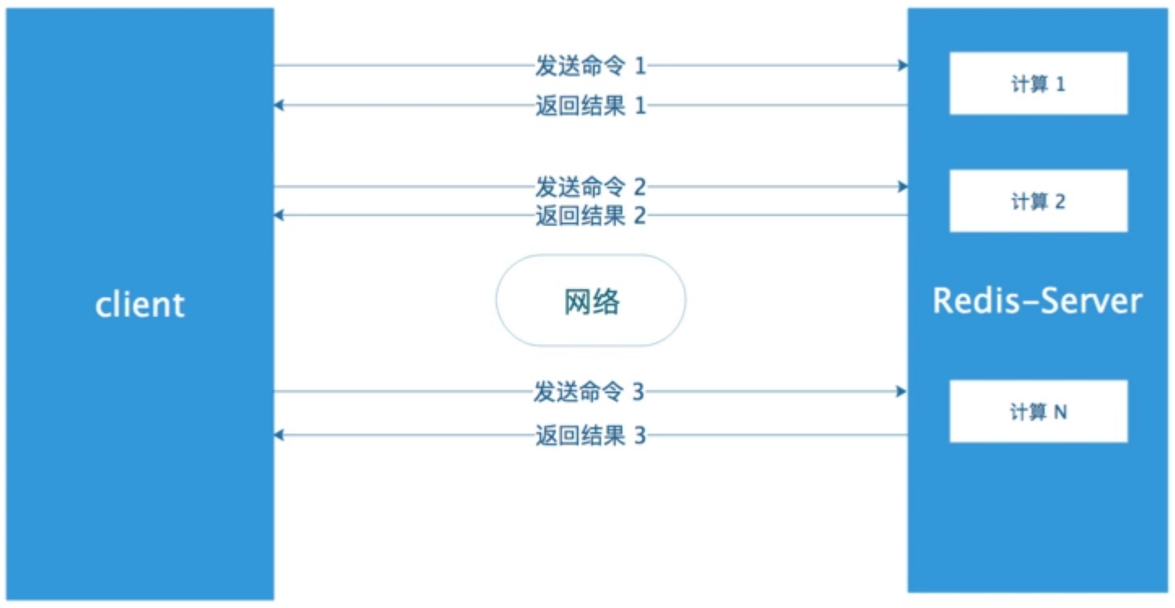
In [70]:
```
#检测redis的name是否存在
print(r.exists("A"))
print(r.exists("C"))
```

```
1
0
```

In [71]:
```
#获取对应值的类型
r.type("A")
```

Out[71]: b'set'

三. Pipeline流水线
Redis客户端执行一条命令分为以下四个步骤
1.发送命令
2.命令排队
3.命令执行
4.返回结果
第一步到第四步称为Round Trip Time(RTT,往返时间)



Redis提供了批量操作命令(例如mget,mset等), 有效的节约RTT. 但大部分命令是不支持批量操作的, 例如要执行n次hgetall命令,并没有mhgetall存在, 需要消耗n次RTT. Redis的客户端和服务端可能不是在不同的机器上, 例如客户端在北京, Redis服务端在上海, 两地直线距离为1300公里, 那么1次RTT时间=1300×2/(300000×2/3)=13毫秒(光在真空中传输速度为每秒30万公里,这里假设光纤的速度为光速的2/3),那么客户端在1秒内大约只能执行80次左右的命令,这个和Redis的高并发高吞吐背道而驰.

Pipeline(流水线)机制能改善上面这类问题, 它能将一组Redis命令进行组装, 通过一次RTT传输给Redis, 再将这组Redis命令按照顺序执行并装填结果返回给客户端. Pipeline并不是什么新的技术和机制, 很多技术上都使用过. 而且RTT在不同网络环境下会有不同, 例如同机房和同机器会比较快,跨机房跨地区会比较慢. Redis命令真正执行的时间通常在微秒级别,所以才会有Redis性能瓶颈是网络这样的说法.

2. 原生批量命令与Pipeline对比
可以使用Pipeline模拟出批量操作的效果,但是在使用时需要质疑它与原生批量命令的区别,具体包含几点:
1) 原生批量命令是原子性, Pipeline是非原子性的. 原子性表示组成一个事务的多个操作是一个不可分割的原子单元, 只有所有的操作执行成功, 整个事务才提交.
2) 原生批量命令是一个命令对应多个key, Pipeline支持多个命令.
3) 原生批量命令是Redis服务端支持实现的, 而Pipeline需要服务端与客户端的共同实现.

In [ ]:
示例代码:

In [2]:
```
import redis
pool = redis.ConnectionPool(host='172.16.70.251', port=6379)
r = redis.Redis(connection_pool=pool)
pipe = r.pipeline(transaction=True)
r.set('name', 'terry')
r.set('role', 'python')
pipe.execute()
print(r.get('name'))
print(r.get('role'))
```

```
b'terry'
b'python'
```

四.发布/订阅

In [ ]:
```
class RedisHelper:
    def __init__(self):
        self.__conn = redis.Redis(host='172.16.70.251')
        self.chan_sub = 'fm104.5'
        self.chan_pub = 'fm104.5'

    def public(self, msg):
        self.__conn.publish(self.chan_pub, msg)
        return True

    def subscribe(self):
        pub = self.__conn.pubsub()
        pub.subscribe(self.chan_sub)
        pub.parse_response()
        return pub
```

In [ ]:
订阅者

In [ ]:
```
from util import RedisHelper

obj = RedisHelper()
redis_sub = obj.subscribe()

while True:
    msg= redis_sub.parse_response()
    print msg
```

发布者:

In [ ]:
```
from demo4 import RedisHelper
obj = RedisHelper()
obj.public('hello')                                                    True
```

此时订阅者的Shell中将会看到发布者发送的内容.