# EasyRedis设计文档

使用python3实现的简易的Redis

文档编辑者：陈加鑫

编辑时间：2020.12.6

版本：1.0

目录

**[EasyRedis设计文档 1](#_Toc116)**

[1. 设计思路 2](#_Toc27231)

[2. 采用的语言、框架、通信协议说明 2](#_Toc28567)

[3. 主要代码截图 3](#_Toc13751)

[3.1. 客户端配置文件：config.py 3](#_Toc17902)

[3.2. 客户端文件：easyredis\_client.py 3](#_Toc436)

[3.3. 服务端配置文件config.py 5](#_Toc3218)

[3.4. 服务端文件easyredis\_server.py 5](#_Toc26388)

[4. 运行截图（重点） 13](#_Toc1325)

## 设计思路

EasyRedis：使用Python3实现的简易的Redis。

具体思路：EasyRedis使用python中的低级别的网络服务支持基本的Socket，去访问底层操作系统 Socket 接口。使用其创建服务端和客户端，服务端可以监听客户端发送的请求并创建连接。创建连接时，服务端会每个客户端创建一个进程，在进程中维持与客户端的通讯，服务端有一个连接列表，服务端每创建一个连接都会将所创建的进行放入其中。服务端与客户端之间通讯协议严格遵守Redis2.0以后使用的Redis服务其通讯新标准协议。

数据存储实现：

EasyRedis 使用一个名为er的全局字典存放所有数据

字符串：在er中以键值对的形式进行存储，键值都为字符串数据类型，并实现set、get、exist、incr指令

列表：在er中存放列表，列表名为字符串数据类型，列表的值使用Python已经封装好的列表数据类型，并实现lpush、rpop、lrange指令

集合：在er中存放集合，集合名为字符串数据类型，集合的值使用Python已经封装好的集合数据类型，并实现sadd、smembers指令

哈希：在er中存放哈希表，哈希表名为字符串数据类型，用Python的字典数据类型存放，并实现哈希命令：hset，hget

持久化：通过save命令，将er全局字典通过序列化为二进制数据，将数据存放进文件中。启动EasyRedis时先将数从文件中读取，并赋值给er，实现持久化数据。

## 采用的语言、框架、通信协议说明

开发语言：

Python 3.7.4

使用框架：

Python中低级别的网络服务支持基本的 Socket

通信协议：

严格遵守redis2.0中Redis服务器通讯的新标准协议。

持久化协议：

二进制序列化

二进制序列化框架：

Python pickle模块

## 主要代码截图

* 1. **客户端配置文件：config.py**

# 客户端连接服务器 IP地址 端口号

host = '127.0.0.1'

port = 6780

* 1. **客户端文件：easyredis\_client.py**

import config

import socket

import re

HOST = config.host

PORT = config.port

def respond\_handle(data: str) -> str:

    if data[0] == '+':

        l = re.findall('\+(.\*)\r\n', data)

        return l[0]

    if data[0] == ':':

        l = re.findall(':(.\*)\r\n', data)

        return '(integer)'+l[0]

    if data[0] == '$':

        l = re.findall('(.\*)\r\n', data)

        return l[1] if int(l[0][1:]) == len(l[1]) else 'error'

    if data[0] == '\*':

        pattern = '(.\*)\r\n'

        l = re.findall(pattern, data)  # 正则表达式解析协议内容

        c\_l = []  # 将数据再次解析到列表方便 后续操作

        for i in range(int(l[0][1:])):  # 根据 \*x x的个数获取传过来的参数

            if int(l[2 \* i + 1][1:]) == len(l[2 \* i + 2]):  # 根据 $x x的大小判断 参数值是否正确

                c\_l.append(l[2 \* i + 2])  # 将数据再次解析到列表方便

        return c\_l

    if data[0] == '-':

        l = re.findall('-(.\*)\r\n', data)

        if l[0] == '1':

            return '(error) ERR unknown command'

        elif l[0] == '2':

            return '(error) ERR wrong number of arguments'

        elif l[0] == '3':

            return '(error) WRONGTYPE Operation against a key holding the wrong kind of value'

        elif l[0] == '4':

            return '(error) ERR value is not an integer or out of range'

        else:

            return '(error)'

def is\_quit(str\_cmd: str) -> bool:

    if str\_cmd.strip() == 'quit':

        return True

    return False

def transform\_input(str\_cmd: str) -> str:

    message = ''  # 定义一个返回字符串

    temp = str\_cmd.strip()  # 去除两端多余空格

    list\_cmd = re.findall('\S+', temp)  # 根据空格转化为字符串列表

    message += '\*{}\r\n'.format(len(list\_cmd))  # 获取字符串列表长度 得到 \*len 第一个参数

    for i in list\_cmd:  # 获取每个参数长度 得到 $len

        message += '${}\r\n{}\r\n'.format(len(i), i)

    return message

def start():

    s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  # 定义socket类型，网络通信，TCP

    try:

        s.connect((HOST, PORT))  # 要连接的IP与端口

    except ConnectionRefusedError:

        print('由于目标计算机积极拒绝，无法连接。')  # 无法连接服务器

        return  # 停止程序

    while 1:

        cmd = input("EasyRedis->:")  # 与人交互，输入命令

        if is\_quit(cmd):  # 判断是否退出客户端

            break

        cmd = transform\_input(cmd)  # 将输入的命令协议化

        s.sendall(cmd.encode())  # 把命令转为二进制 并 发送给对服务器端

        data = s.recv(1024)  # 把接收的数据定义为变量

        print(respond\_handle(data.decode()))  # 把接受数据转为字符串 并 输出

start()

* 1. **服务端配置文件config.py**

# 服务器绑定的 IP地址 端口号

host = ''

port = 6780

# 持久化文件名

file\_name = 'ER\_data.byte'

* 1. **服务端文件easyredis\_server.py**

import config

import socket

import threading

import re

import os

import pickle

# 配置信息

HOST = config.host

PORT = config.port

FILE\_NAME = config.file\_name

# 全局字典存放各种类型的数据

global er

def init():

    """

    初始化EasyRedis

    初始化全局字典存数据

    或将持久化文件的数据读取到全局字典中

    """

    global er

    er = {}

    if os.path.exists(FILE\_NAME):  # 若持久化文件存在则读取文件并设置为全局自字典

        with open(FILE\_NAME, 'rb') as f:  # 打开文件

            er = pickle.load(f)  # 将二进制文件对象转换成 Python 对象

def string\_set(key, value):

    """

    字符数据类型的set方法 设置key的值为value

    :param key: key值

    :param value: value值

    :return: 设置成功返回 1 ，若参数为空返回 错误提示信息

    """

    if value:

        er[key] = value[0]

        return respond\_handle('integers', '1')

    else:

        return respond\_handle('errors', '2')

def string\_get(key, value=None):

    """

    字符数据类型的get方法 获取key的值

    :param key: key值

    :param value: value值

    :return: 值存在返回值，不存在返回nil

    """

    if key not in er:

        return respond\_handle('bulk\_strings', 'nil')

    if isinstance(er[key], str):

        return respond\_handle('bulk\_strings', er[key])

    else:

        return respond\_handle('errors', '3')

def exist(key, value=None):

    """

    字符数据类型的exist方法 判断key是否存在值

    :param key: key值

    :return: 存在返回1，不存在返回0

    """

    if key in er:

        return respond\_handle('integers', '1')

    else:

        return respond\_handle('integers', '0')

def string\_incr(key, value=None):

    """

    字符数据类型的incr方法

    若key的值为数字类型的值，则值自增 1

    若key的值不为数字类型，返回错误信息

    若key不存在，则设置key的值为 1

    :param key: key值

    :return: 成功自增返回 1 ，失败返回 错误信息

    """

    if key not in er:

        er[key] = '1'

        return respond\_handle('integers', '1')

    if not isinstance(er[key], str):

        return respond\_handle('errors', '3')

    if er[key].isdigit():

        er[key] = str(int(er[key]) + 1)

        return respond\_handle('integers', '1')

    return respond\_handle('errors', '4')

def list\_lpush(key, value):

    """

    将所有指定的值插入到存于 key 的列表的头部。

    如果 key 不存在，那么在进行 push 操作前会创建一个空列表。

    如果 key 对应的值不是一个 list 的话，那么会返回一个错误。

    可以使用一个命令把多个元素 push 进入列表，只需在命令末尾加上多个指定的参数。

    元素是从最左端的到最右端的、一个接一个被插入到 list 的头部。

    所以对于这个命令例子 LPUSH mylist a b c，

    返回的列表是 c 为第一个元素， b 为第二个元素， a 为第三个元素。

    :param key: key值

    :param value: 列表值，lpush的值

    :return:

    """

    if key not in er:

        er[key] = []

    if not isinstance(er[key], list):

        return respond\_handle('errors', 3)

    if value:

        for i in value:

            er[key].insert(0, i)

        return respond\_handle('integers', len(value))

    return respond\_handle('errors', 2)

def list\_rpop(key, value=None):

    """

    移除并返回存于 key 的 list 的最后一个元素。

    :param key: key值

    :return: 最后一个元素的值，或者当 key 不存在的时候返回 nil。

    """

    if key not in er:

        return respond\_handle('bulk\_strings', 'nli')

    if not isinstance(er[key], list):

        return respond\_handle('errors', 3)

    return respond\_handle('bulk\_strings', er[key].pop(-1))

def list\_lrange(key, value=None):

    """

    返回存储在 key 的列表里指定范围内的元素。

    start 和 end 偏移量都是基于0的下标，即list的第一个元素下标是0（list的表头），

    第二个元素下标是1，以此类推。

    :param key: key值

    :return: 指定范围里的列表元素。

    """

    if key not in er:

        return respond\_handle('bulk\_strings', 'nli')

    if not isinstance(er[key], list):

        return respond\_handle('errors', 3)

    start = 0

    end = len(er[key]) + 1

    if len(value) == 2 and str(value[0]).isdigit() and str(value[1]).isdigit():

        start = int(value[0])

        end = int(value[1]) + 1

    l = er[key][start:end]

    return respond\_handle('arrays', l)

def set\_sadd(key, value):

    """

    添加一个或多个指定的member元素到集合的 key中.指定的一个或者多个元素member

    如果已经在集合key中存在则忽略.如果集合key

    不存在，则新建集合key,并添加member元素到集合key中.

    如果key 的类型不是集合则返回错误.

    :param key: key值

    :param value: value列表

    :return: 添加成功返回添加数量

    """

    if key not in er:

        er[key] = set()

    if not isinstance(er[key], set):

        return respond\_handle('errors', 3)

    temp = er[key].copy()

    for i in value:

        er[key].add(i)

    return respond\_handle('integers', len(er[key] - temp))

def set\_smembers(key, value):

    if key not in er:

        return respond\_handle('bulk\_strings', 'nli')

    if not isinstance(er[key], set):

        return respond\_handle('errors', '3')

    data = list()

    for i in er[key]:

        data.append(i)

    return respond\_handle('arrays', data)

def hash\_hset(key, value):

    """

    设置 key 指定的哈希集中指定字段的值。

    如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。

    如果字段在哈希集中存在，它将被重写。

    :param key: key值

    :param value: value列表

    :return: 成功返回 1 ，失败返回错误

    """

    if key not in er:

        er[key] = dict()

    if not isinstance(er[key], dict):

        return respond\_handle('errors', 3)

    if len(value) != 2:

        return respond\_handle('errors', 2)  # 返回参数错误

    field = value.pop(0)

    value = value.pop(0)

    er[key][field] = value

    return respond\_handle('integers', 1)

def hash\_hget(key, value=None):

    """

    返回 key 指定的哈希集中该字段所关联的值

    :param key: key值

    :return: 该字段所关联的值。当字段不存在或者 key 不存在时返回nil。

    """

    if key not in er:

        return respond\_handle('bulk\_strings', 'nil')

    if len(value) != 1:

        return respond\_handle('errors', '2')

    if isinstance(er[key], dict) and value[0] in er[key]:

        return respond\_handle('bulk\_strings', er[key][value[0]])

    return respond\_handle('bulk\_strings', 'nil')

def er\_save(key=None, value=None):

    """

    持久化指令处理函数

    :return: 返回simple\_strings OK

    """

    with open(FILE\_NAME, 'wb') as f:  # 打开文件

        pickle.dump(er, f)  # 用 dump 函数将 Python 对象转成二进制对象文件

    return respond\_handle('simple\_strings', 'OK')

def er\_keys(key=None, value=None):

    """

    获取当前数据库的所有key

    :return: 返回当前数据库所有key

    """

    l = er.keys()

    return respond\_handle('arrays',l)

# 指令对应的处理函数

switch = {

    'set': string\_set,

    'get': string\_get,

    'exist': exist,

    'incr': string\_incr,

    'lpush': list\_lpush,

    'rpop': list\_rpop,

    'lrange': list\_lrange,

    'sadd': set\_sadd,

    'smembers': set\_smembers,

    'hset': hash\_hset,

    'hget': hash\_hget,

    'save': er\_save,

    'keys': er\_keys,

}

def data\_handle(command: str, key=None, value=None) -> str:

    """

    指令处理函数

    :param command: 指令

    :param key: key值

    :param value: value值

    :return: 返回操作结果信息

    """

    try:

        return switch[command](key, value)

    except KeyError:

        return respond\_handle('errors', 1)

def request\_handle(data: str) -> bytes:

    """

    消息处理函数

    :param data:

    :return:

    """

    pattern = '(.\*)\r\n'

    l = re.findall(pattern, data)  # 正则表达式解析协议内容

    c\_l = []  # 将数据再次解析到列表方便 后续操作

    for i in range(int(l[0][1:])):  # 根据 \*x x的个数获取传过来的参数

        if int(l[2 \* i + 1][1:]) == len(l[2 \* i + 2]):  # 根据 $x x的大小判断 参数值是否正确

            c\_l.append(l[2 \* i + 2])  # 将数据再次解析到列表方便

    if len(c\_l) > 1:  # 多个参数的指令

        message = data\_handle(c\_l[0], c\_l[1], c\_l[2:])

    elif c\_l[0] == 'save' or c\_l[0] == 'keys':  # 无参指令

        message = data\_handle(c\_l[0])

    elif c\_l[0] in switch:  # 需要参数的指令，但无参数传入，返回错误信息

        message = respond\_handle('errors', '2')

    elif c\_l[0] not in switch:  # 未知指令

        message = respond\_handle('errors', '1')

    else:  # 兜底

        message = respond\_handle('errors', '5')

    return message.encode()  # 解析成byte数据 为socket 传输用

def respond\_handle(type: str, value=None) -> str:

    """

    响应处理函数，将数据协议化后返回给客户端

    :param type: 五种：'simple\_strings'、'integers'、'bulk\_strings'、'arrays'、'errors'

    :param value: 需要协议化的数据

    :return: 返回协议化后数据

    """

    if type == 'simple\_strings':  # 简单

        return '+{}\r\n'.format(value)

    if type == 'integers':  # 整数

        return ':{}\r\n'.format(value)

    if type == 'bulk\_strings':  # 单行

        return '${}\r\n{}\r\n'.format(len(value), value)

    if type == 'arrays':  # 批数据

        message = str()

        message += '\*{}\r\n'.format(len(value))  # 获取字符串列表长度 得到 \*len 第一个参数

        for i in value:  # 获取每个参数长度 得到 $len

            message += '${}\r\n{}\r\n'.format(len(i), i)

        return message

    if type == 'errors':  # 错误

        return '-{}\r\n'.format(value)

    return '-(error)\r\n'

def thread\_conn(conn, address):

    """

    每个连接的线程的

    处理函数

    :param conn:  socket连接

    :param address:  连接IP

    :return:

    """

    print('Connected by', address)  # 输出客户端的IP地址

    while 1:

        try:

            data = conn.recv(1024)  # 把接收的数据实例化

        except (ConnectionResetError, ConnectionAbortedError):  # 客户端关闭连接

            break

        data = data.decode()

        if str(data).strip() == 'quit':  # 客户端退出连接

            break

        # 处理客户端发来请求 , 返回bytes响应数据

        message = request\_handle(str(data))

        conn.sendall(message)  # 把结果发送给客户端

    # 上面循环断开说明连接断开，下面打印连接断开，并关闭连接

    print('The connection of {} is closed.'.format(address))

    conn.close()  # 关闭连接

def start():

    """

    主线程

    """

    init()  # 初始化EasyRedis

    s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    s.bind((HOST, PORT))

    s.listen(20)  # 设置最大的监听数量

    list\_conn\_threading = []  # 连接线程存放列表

    print('The server starts waiting for the connection.')

    while True:

        conn, address = s.accept()  # 接受客户端连接

        # 将客户端信息包装成字典数据

        kwargs = {

            'conn': conn,

            'address': address

        }

        thread = threading.Thread(target=thread\_conn, kwargs=kwargs)  # 创建线程去处理每一个客户端发来的请求

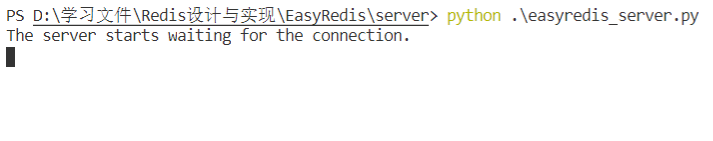
        thread.start()  # 启动线程

        list\_conn\_threading.append(thread)  # 将每条线程放如一个list中

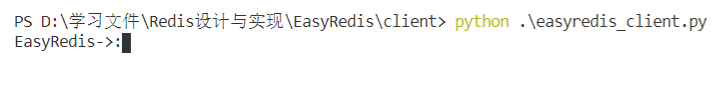
start()

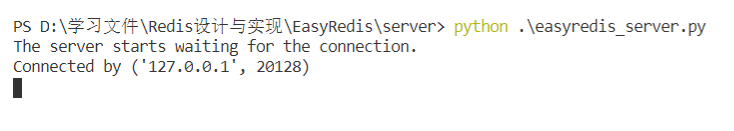
## 运行截图（重点）

* 1. 服务器启动：

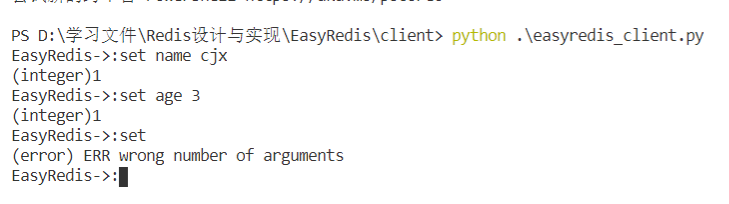


* 1. 客户端连接：

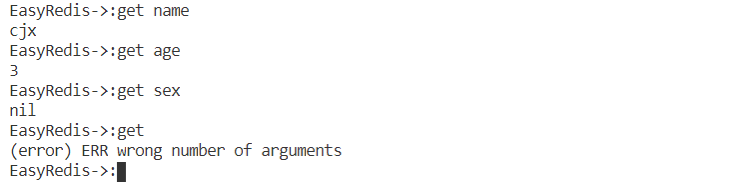




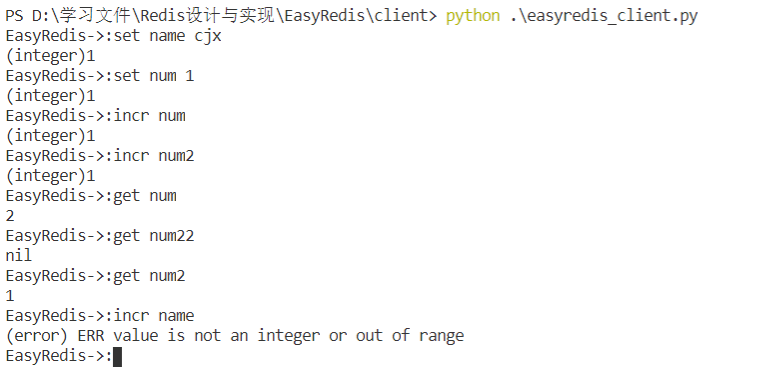
* 1. set指令



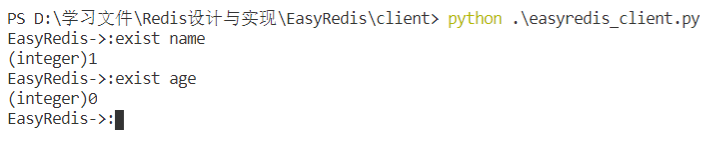
* 1. get指令



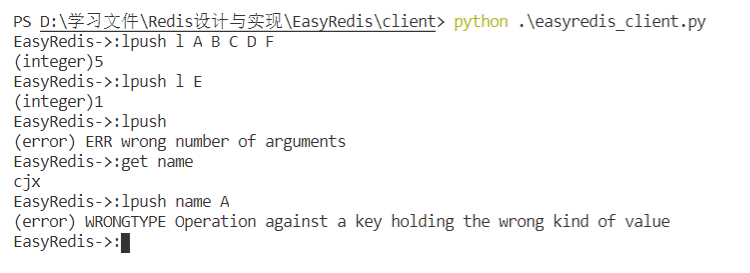
* 1. incr指令



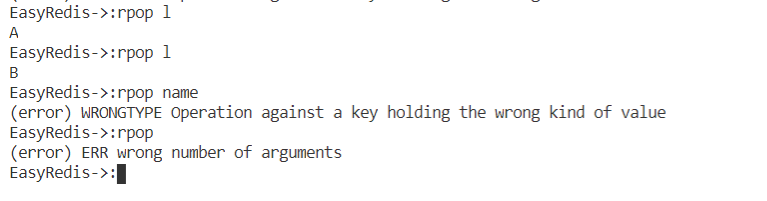
* 1. exist 指令



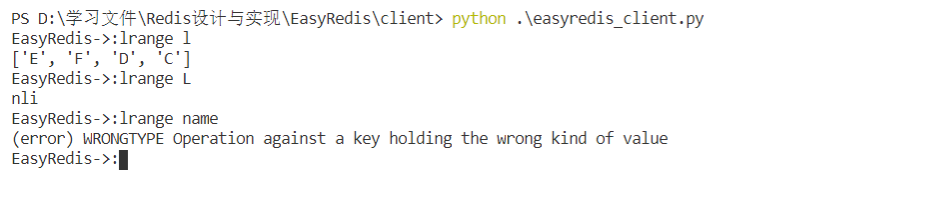
* 1. lpush指令



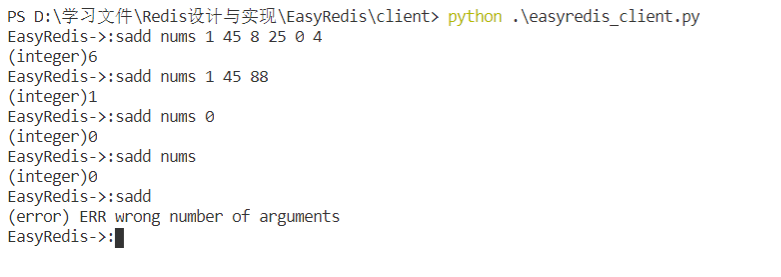
* 1. rpop指令



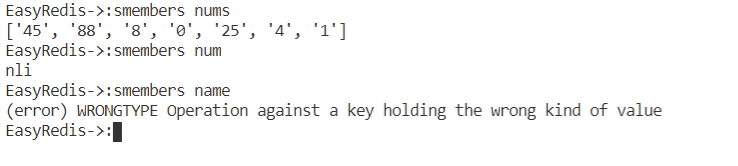
* 1. lrange指令



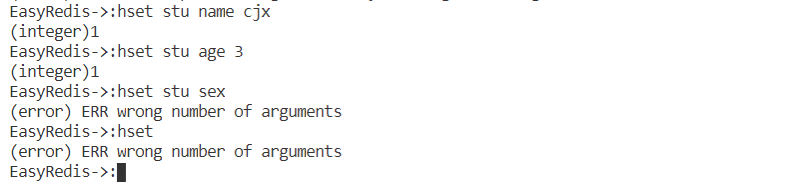
* 1. sadd指令



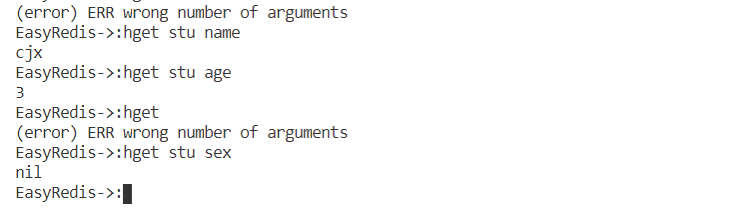
* 1. smembers指令



* 1. hset指令



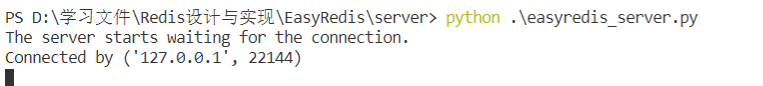
* 1. hget指令

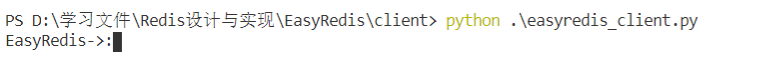


* 1. save指令



**重启服务端和客户端：**





**查询持久化数据：**

