# 引言

- 在上一章中，学习了如何从已爬取到网页中抓取数据，以及将抓取到的结果保存到CSV文件中。
- 如果还想抓取另外一个字段，比如国旗图片的URL，那又该怎么做呢?
- 要想抓取这些新增的字段，需要重新下载整个网站。对于这个小型的示例网站而言，这可能不算特别大的问题。但是，对于那些拥有数百万个网页的网站来说，重新爬取可能需要消耗的几个星期的时间。
- 爬虫避免此类问题的方式之一，就是从开始时就缓存被爬取的网页，这样就可以让每个网页只下载一次。


# 内容提要



- 何时使用缓存
- 为链接爬虫添加缓存支持
- 磁盘缓存
- 键值对数据库Redis
- 实现Redis缓存

# 3.1 何时使用缓存

- 缓存，还是不缓存？ 对于很多程序员、数据科学家以及进行网络抓取的人来说，是一个需要回答的问题。
- 如果你需要执行一个大型爬取工作，那么它可能由于错误或异常被中断，缓存可以帮助你无需重新爬取那些可能已经抓取过的页面。缓存还可以，让你在离线时访问这些页面（出于数据分析或开发的目的）。
- 不过，如果你的最高优先级是获得网站最新和当前的信息，那此时缓存就没有意义。此外，如果你没有计划实现大型或可重复的爬虫。那么可能只需要每次去抓取页面即可。

# 3.2 为链接爬虫添加缓存支持

- 要想支持缓存，需要修改第1章中编写的download函数，使其在URL下载之前进行缓存检查。
- 另外，还需要把限速功能移至函数内部，只有在真正发生下载时才会触发限速，而在加载缓存时不会触发。
- 为了避免每次下载都要传入多个参数，借此机会将download函数重构为一个类(Download类)，这样只需在构造方法中设置参数，就能在后续下载时多次复用。

```python
# ----------------download.py-------------------
import sys
sys.path.append('../ch1')  # 注意，还需要在PyCharm中将ch1包设为源根

from throttle import Throttle
import random
import requests
import time


class Downloader:
    def __init__(self, delay=5, user_agent='wswp', proxies=None, cache={}):
        self.throttle = Throttle(delay)
        self.user_agent = user_agent
        self.proxies = proxies
        self.cache = cache
    def __call__(self, url, num_retries=2):
        ……
    def download(self, url: str, headers, proxies, num_retries=2):
        ……
```

```python
class Downloader:
    def __init__(self, delay=5, user_agent='wswp', proxies=None, cache={}):
        ......
    def __call__(self, url, num_retries=2):
        try:
            result = self.cache[url]  # 尝试从缓存获取页面
            print('Loaded from cache:', url)
        except KeyError:
            result = None
        if result and 500 <= result['code'] <= 600:
            # server error so ignore result from cache and re-download
            result = None
        if result is None:
            # result was not loaded from cache, so still need to download
            self.throttle.wait(url)
            proxies = random.choice(self.proxies) if self.proxies else None
            headers = {'User-Agent': self.user_agent}
            result = self.download(url, headers, proxies, num_retries)  # 下载页面
            if self.cache:
                self.cache[url] = result  # save result to cache
        return result['html']
```

```python
class Downloader:
    ......
    def download(self, url: str, headers, proxies , num_retries=2):
        print('Downloading:', url)
        response = None
        try:
            response = requests.get(url=url, headers=headers, proxies=proxies, timeout=10)
            response.encoding = response.apparent_encoding
            html = response.text
            if response.status_code != 200:  # >= 400
                print('Error code:', response.status_code )
                html = None
                if num_retries > 0 and response.status_code >= 400:
                    delay = 5  # 延迟发送请求的秒数
                    print(f'Pause for {delay} seconds.')
                    time.sleep(delay)  # 暂停执行若干秒
                    print('Retry to download.')
                    return self.download(url, headers, proxies, num_retries - 1)  # 重试下载
            else:
                print('encoding=', response.encoding)
            code = response.status_code
        except Exception as e:
            print('Download error:', e)
            if hasattr(e, 'code'):
                print('Error code:', e.code)
            html = None
            code = (404 if response is None else response.status_code)
        return {'html': html, 'code': code}
```

注意设置一下超时，有的页面请求会长时间无响应。

# 3.2 为链接爬虫添加缓存支持

- Dowlnoad类的__call__( )方法是一个特殊的方法，在对象被作为函数调用时，实际执行的就是该方法。
- 在该方法中，实现了下载前检查缓存功能。
  - 首先检查URL之前是否已经放入缓存中。默认情况下，缓存是一个Python字典。
  - 如果URL已经被缓存，则检查之前下载中是否遇到了服务器端错误。
  - 最后，如果没有发生过服务器端错误，则表明该缓存结果可用。
- 如果上述检查中的任何一项失败，都需要正常下载该URL，然后将得到的结果添加到缓存中。


# 3.2 为链接爬虫添加缓存支持



- 这里的dowlnoad( )方法和之前的download函数基本一样，只是现在返回了HTTP状态码，以便在缓存中存储错误码。
- 当然，如果只需要一个简单的下载功能，而不需要限速或缓存，则可以直接调用该方法，这样就不会通过__call__( )方法调用了。


# 3.2 为链接爬虫添加缓存支持



- 而对于cache类，可以通过执行 result = cache[url] 从cache中加载数据，并通过cache[url] = result 向 cache中保存结果，这是一个来自Python内置字典数据类型的便捷接口。为了支持该接口，cache类需要定义两个特殊的方法：__getitem__( )和__setitem__( )。
- 此外，为了支持缓存功能。链接爬虫的代码也需要进行一些微调，包括添加cache参数、移除限速以及将download( )函数替换为新的类等，如下面的代码所示。

```
# ----------------avdadvanced_link_crawler.py-------------------

from downloader import Downloader
import re
from urllib.parse import urljoin
from lxml.html import fromstring

FIELDS = (‘area’, ‘population‘,……., 'languages', 'neighbours')
def scrape_callback(url, html):
    ……
def get_links(html):
    ……
def link_crawler(start_url, link_regex, scrape_callback=None, delay=5,
                 user_agent='wswp', proxies=None, cache={},
                 num_retries=2):
    """ Crawl from given start URL following links matched by link_regex """
    crawl_queue = [start_url]
    # keep track which URL's have been before
    seen = set(crawl_queue)
```

```
……
D = Downloader(delay, user_agent, proxies, cache)
i = 0
while crawl_queue:
    url = crawl_queue.pop()  # 弹出队列首元素（链接）
     html = D(url, num_retries)  # 从缓存读取，或下载页面并缓存
     if html is None:
        continue
    # -------------抓取网页数据-------------
    if scrape_callback is not None:
          scrape_callback(url, html)

    # filter for links matching our regular expression
    for link in get_links(html):
        if re.match(link_regex, link):
            abs_link = urljoin(start_url, link)
            # check if have already seen this link
            if abs_link not in seen:
                seen.add(abs_link)
                crawl_queue.append(abs_link)  # 绝对链接加入队列末尾
```

# 3.3 磁盘缓存

- 要想缓存下载结果。先来尝试最容易想到的方案，将下载到的网页存储到文件系统中。
- 为了实现该功能，需要将URL安全的映射为跨平台文件名。表3.1所示为几大主流文件系统的限制。

表3.1

| 操作系统 | 文件系统 | 非法文件名字符 | 文件名最大长度 |
| --- | --- | --- | --- |
| Linux | Ext3/Ext4 | / 和 \0 | 255字节 |
| OS X | HFS Plus | : 和 \0 | 255个UFT-16编码单元 |
| Windows | NTFS | \、/、？、:、*、”、>、< 和 \| | 255个字符 |

# 3.3.1 路径映射

- 为了保证在不同文件系统中的文件路径都是安全的，就需要限制其只能包含数字，字母和基本符号，并将其他字符替换为下划线，其实现代码如下所示。

```
>>> import re
>>> url = 'http://example.python-scraping.com/default/view/Australia-1'
>>> re.sub(pattern='[^/0-9a-zA-Z~.;_-{}!@#%&+]',repl='_',string=url)  # 不要出现^()[]$等
'http_//example.python_scraping.com/default/view/Australia_1'
```

- 此外，文件名及其父目录的长度需要限制在255个字符以内，以满足表3.1中给出的长度限制。

```
>>> filename=re.sub(pattern='[^/0-9a-zA-Z~.;_-{}!@#%&+]',repl='_',string=url)
>>> filename='/'.join(segment[:255] for segment in filename.split('/'))
>>> print(filename)
http_//example.python_scraping.com/default/view/Australia_1
```

pattern应改为 [^-/0-9a-zA-Z~.;_{}!@#%&+]

# 3.3.1 路径映射

- 由于这里的URL部分没有超过255个字符，因此文件路径不需要改变。还有一种边界情况需要考虑，那就是URL路径可能会以斜杠（/）结尾，这时斜杆后面的空字符串就会成为一个非法的文件名。但是，如果移除这个斜杠，使用其父字符串作为文件名，又会造成无法保存其他URL的问题。考虑下面两个URL：
  - **http://example.python-scraping.com/index/**
  - **http://example.python-scraping.com/index/1**

# 3.3.1 路径映射

- 如果希望这两个URL都能保存下来，就需要以index作为目录名，以文件名1作为子页面。
- 对于像第一个URL路径这样以斜杠结尾的情况，这里使用的解决方案是添加index.html为其文件名。
- 同样地，当URL路径为空时也需要进行相同的操作。
- 为了解析URL，需要使用urlsplit( )函数，将URL分割成几个部分。该函数提供了解析和处理URL的便捷接口

```
>>> from urllib.parse import urlsplit
>>> url='http://example.python-scraping.com/index/'
components=urlsplit(url)
>>> print(components)
SplitResult(scheme='http', netloc='example.python-scraping.com',
path='/index/', query='', fragment='')
>>> print(components.netloc)
example.python-scraping.com
>>> print(components.path)
/index/
```

# 3.3.1 路径映射

- 示例：使用urlsplit模块对上述边界情况添加index.html

```
from urllib.parse import urlsplit

# url = 'http://example.python-scraping.com'          # path为None
# url = 'http://example.python-scraping.com/'         # path为/
url = 'http://example.python-scraping.com/index/'  # path为/index/
# url = 'http://example.python-scraping.com/places/default/index/1'
# url = 'http://example.python-scraping.com/places/default/view/Afghanistan-1'
components = urlsplit(url)
path = components.path
if not path: # 相对路径path不为空
    path = '/index.html'
elif path.endswith('/'):
    path += 'index.html'
filename = components.netloc + path + components.query
print(filename)
```

example.python-scraping.com/index/index.html

# 3.3.1 路径映射

- 根据所抓取网站的不同，可能需要修改边界情况处理功能。例如，由于Web服务器有其期望的URL传输方式，一些站点会在每个URL后面添加/。对于这些站点，你可能只需要去除每个URL尾部的斜杆即可。
- 再次重申，你需要评估并更新网络爬虫的代码，以最佳适应想要抓取的网站。


# 3.3.2 实现磁盘缓存



- 上一小节中，介绍了创建基于磁盘的缓存时需要考虑的文件系统限制，包括允许使用哪些字符、文件名长度限制，以及确保文件和目录的创建位置不同。把URL到文件名的逻辑映射与代码结合起来，就形成了磁盘缓存的主要部分。
- 下面是DiskCache类的初始实现代码。

```python
# ----------------diskcache.py----------------------
import os,  re ,  json
from urllib.parse import urlsplit

class DiskCache:
    def __init__(self, cache_dir='cache', max_len=255):
        self.cache_dir = cache_dir
        self.max_len = max_len

    def url_to_path(self, url):
        """Return file system path string for given URL"""
        components = urlsplit(url)
        # append index.html to empty paths
        path = components.path
        if not path:
            path = '/index.html'
        elif path.endswith('/'):
            path += 'index.html'
        filename = components.netloc + path + components.query
        # replace invalid characters
        filename = re.sub(pattern='[^-/0-9a-zA-Z~.;_{}!@#%&+]', repl='_', string=filename)
       # restrict maximum number of characters
        filename = '/'.join(segment[: self.max_len] for segment in filename.split('/'))

        return os.path.join(self.cache_dir, filename)
```

# 3.3.2 实现磁盘缓存

- 在上面的代码中，构造方法传入了一个用于设定缓存位置的参数cache_dir，然后在url_to_path( )方法中应用前面讨论的文件名限制映射url到磁盘文件的路径。
- 现在，还缺少根据文件名存取数据的两个方法。
  - __getitem__( )方法
  - __setitem__( )方法

```
……
class DiskCache:
    ……
    def __getitem__(self, url):
        """Load data from disk for given URL"""
        path = self.url_to_path(url)
        if os.path.exists(path):
            with open(path, 'tr') as fp:
                return json.load(fp)
        else:  # URL has not been cached
            raise KeyError(url + 'does not exist')

    def __setitem__(self, url, result):
        """Save data to disk for given URL"""
        path = self.url_to_path(url)
        folder = os.path.dirname(path)
        if not os.path.exists(folder):
            os.makedirs(folder)
        with open(path, 'tw') as fp:
            json.dump(result, fp)
```

与Download类的__call__( )方法中的result = self.cache[url]对应

与Download类的__call__( )方法中的self.cache[url] = result 对应

# 3.3.2 实现磁盘缓存

- 在__setitem__()方法
  - 使用url_to_path( )方法将url映射为安全文件名，在必要情况下，还需要创建父目录。
  - 这里使用的json模块会对Python对象（数据）进行系列化处理，然后保存的磁盘中。
- 在__getitem__()方法
  - 首先使用url_to_path()方法将url映射为安全文件名。
  - 如果文件存在，则使用json加载其内容，并恢复其原始数据类型。
  - 如果文件不存在（即缓存中还没有改URL的数据），则抛出KeyError异常。

# 3.3.3 磁盘缓存测试

- 向爬虫传递cache关键字参数，来检验DiskCache类。

```
# ----------------advanced_link_crawler_tester.py---------------------
from advanced_link_crawler  import link_crawler, scrape_callback
from diskcache import DiskCache
import time

url = 'http://example.python-scraping.com'
# url = 'http://180.201.165.235:8000/places/'
regex = '/places/default/(index|view)/'
start = time.time()
link_crawler(url, regex, scrape_callback=scrape_callback, cache=DiskCache())
end = time.time()

seconds = end - start
hours = int(seconds / 3600)
mins = int(seconds % 3600 // 60)
secs = seconds % 60

print("Wall time: %d hours % d mins %f secs" % (hours, mins, secs))
```

# 3.3.3 磁盘缓存测试

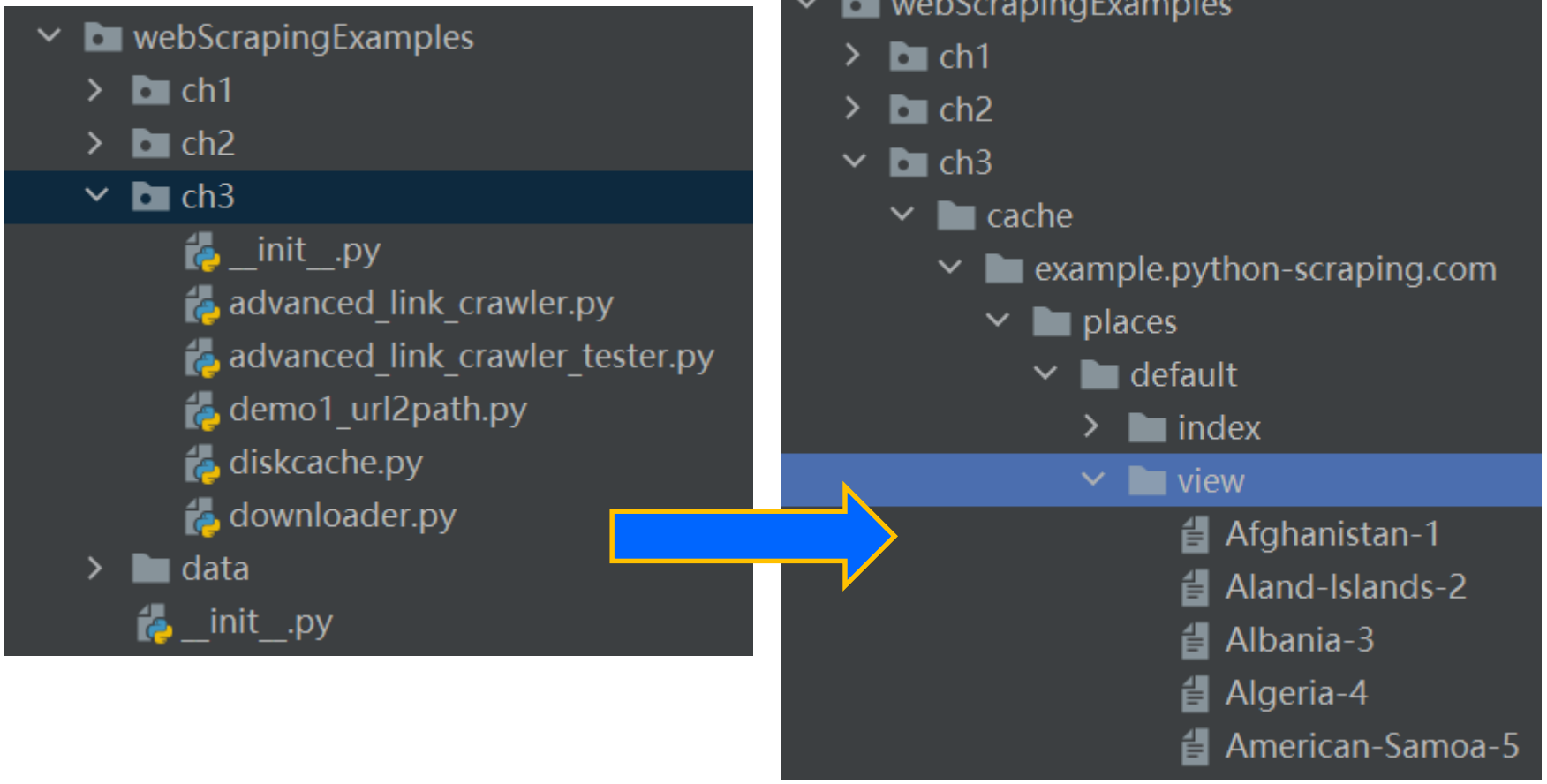
- 首次运行的结果

```
Downloading: http://example.python-scraping.com
Downloading: http://example.python-scraping.com/places/default/index/1
Downloading: http://example.python-scraping.com/places/default/index/2
……
Downloading: http://example.python-
scraping.com/places/default/view/Afghanistan-1
http://example.python-scraping.com/places/default/view/Afghanistan-1 ['647,500
square kilometres', '29,121,286', 'AF', 'Afghanistan', 'Kabul', 'AS', '.af', 'AFN',
'Afghani', '93', '', '', 'fa-AF,ps,uz-AF,tk', 'TM IR TJ PK UZ ']
Wall time: 0 hours  22 mins 35.938094 secs
```

- 首次执行脚本，由于缓存为空，因此网页会被正常下载。
- 可以预见，当第二次执行脚本时，网页加载自缓存，爬虫应该更快完成执行。

# 3.3.3 磁盘缓存测试

- 文件目录的变化



# 3.3.3 磁盘缓存测试

- 缓存后再次运行的结果

```
Loaded from cache: http://example.python-scraping.com
Loaded from cache: http://example.python-scraping.com/places/default/index/1
Loaded from cache: http://example.python-scraping.com/places/default/index/2
……
Loaded from cache: http://example.python-scraping.com/places/default/view/Afghanistan-1
http://example.python-scraping.com/places/default/view/Afghanistan-1 ['647,500 square kilometres', '29,121,286', 'AF', 'Afghanistan', 'Kabul', 'AS', '.af', 'AFN', 'Afghani', '93', '', '', 'fa-AF,ps,uz-AF,tk', 'TM IR TJ PK UZ ']
wall time: 0 hours  0 mins 0.438830 secs
```

- 可见，和上面的预期一样，爬取操作很快就完成了。
- 注意，由于硬件和网络连接速度的差异，在不同计算机中的准确执行时间也会有所区别。不过毋庸置疑的是，磁盘缓存比通过HTTP下载速度更快。

# 3.3.4 节省磁盘空间

- 为了最小化缓存所需的磁盘空间，可以对下载得到的HTML文件进行压缩处理。
- 处理的实现方法很简单，只需在保存到磁盘之前使用zlib压缩序列化字符串即可。
- zlib库基于GNU zip
  - compress( )方法用于压缩数据，压缩前需要将数据转换为字节序列。
  - decompress( )方法用于解压缩数据。
- 使用zlib库后，DiskCachel类修改如下：

```
……
import zlib
class DiskCache:
    def __init__(self, cache_dir='/cache_zlib', max_len=255,
                 compress=True, encoding='utf-8'):
        ……
        self.compress = compress
        self.encoding = encoding

    def url_to_path(self, url):
        ……
    def __getitem__(self, url):
        ……
            mode = ('br' if self.compress else 'tr')
            with open(path, mode) as fp:
                if self.compress:
                    unzip_data = zlib.decompress(fp.read())
                    data = unzip_data.decode(self.encoding)
                    return json.loads(data)  # 将字符串形式的字典转换字典类型
                return json.load(fp)
        ……
```

```
……
import zlib
class DiskCache:
    ……
    def __setitem__(self, url, result):
        ……
        mode = ('bw' if self.compress else 'tw')
        with open(path, mode) as fp:
            if self.compress:
                bytes_data = bytes(json.dumps(result), self.encoding)
                zip_data = zlib.compress(bytes_data)
                fp.write(zip_data)
            else:
                json.dump(result, fp)
```

缓存大小对比

**cache文件夹:　　2.72 MB**

**cache_zlib文件夹:　756 KB**


# 3.3.5 清理过期数据



- 当前版本的磁盘缓存使用键值对的形式在磁盘上保存缓存，未来无论何时请求都会返回结果。对于缓存网页而言，该功能可能不太理想，因为网页内容随时都有可能发生变化，存储在缓存中的数据存在过期风险。
- 为此，将为缓存数据添加过期时间，以便爬虫知道何时需要下载网页的最新版本。具体做法很简单，即在缓存网页时，向字典中存储一个过期时间戳。例如：

  **{"html": "…", "code": 200, "expires": 2022-11-26T16:54:56}**

- 下面的代码为该功能的而实现。

```
……
from datetime import datetime, timedelta

class DiskCache:
    def __init__(self, cache_dir='cache_zlib', max_len=255,
                 compress=True, encoding='utf-8',
                 expires=timedelta(days=30)):
        ……
        self.expires = expires
    def url_to_path(self, url):
        ……
    def __getitem__(self, url):
        ……
            with open(path, mode) as fp:
                if self.compress:
                    unzip_data = zlib.decompress(fp.read())
                    data = unzip_data.decode(self.encoding)
                    data = json.loads(data)  # 将字符串形式的字典转换字典类型
                else:
                    data = json.load(fp)
```

```
def __getitem__(self, url):
    ......
        with open(path, mode) as fp:
            if self.compress:
                unzip_data = zlib.decompress(fp.read())
                data = unzip_data.decode(self.encoding)
                data = json.loads(data)  # 将字符串形式的字典转换字典类型
            else:
                data = json.load(fp)
            exp_date = data['expires']
            if exp_date and datetime.strptime(exp_date,
                                              '%Y-%m-%dT%H:%M:%S') <= datetime.utcnow():
                print('Cache expired!', exp_date)
                raise KeyError(url + 'has expired.')
            return data
    ......
def __setitem__(self, url, result):
    """Save data to disk for given URL"""
    result['expires'] = (datetime.utcnow() +
                         self.expires).isoformat(timespec='seconds')
    path = self.url_to_path(url)
    ......
```


# 3.3.5 清理过期数据

- 在构造方法中，使用timedelta对象将默认过期时间设置为30天。
- 在__setitem__( )方法中，把过期时间戳作为键保存到结果字典中。
- 在__getitem__( )方法中，对比当前UTC时间和缓存过期时间戳，检查是否过期。
- 为了测试过期时间功能，可以将其缩短为5秒，如下所示。

```
# ----------------expires_tester.py-----------------
from diskcache_zlib_expires import DiskCache
from datetime import timedelta
import time

url = 'http://example.python-scraping.com'
cache = DiskCache(expires=timedelta(seconds=5))
result = {'html': '<HTML></HTML>'}
cache[url] = result

result = cache[url]  # 缓存结果最初时可用的
print(result)

time.sleep(5)  #

result = cache[url]  #
print(result)
```

```
{'html': '<HTML></HTML>', 'expires': '2020-10-17T16:54:56’}

Cache expired! 2020-10-17T16:54:56

Traceback (most recent call last):
  File "D:/PycharmProjects/webScrapingExamples/ch3/expires_tester.py
", line 18, in <module>
    result = cache[url]
  File "D:\PycharmProjects\webScrapingExamples\ch3\diskcache_zlib
_expires.py", line 55, in __getitem__
    raise KeyError(url + ' has expired.’)

KeyError: 'http://example.python-scraping.com has expired.’
```

- 缓存结果最初时可用的
- 经过**5**秒睡眠之后，再次读取同一**URL**的缓存，则会抛出**KeyError**异常，也就是说缓存过期失效了

# 3.3.6 磁盘缓存缺点

- 基于磁盘的缓存系统比较容易实现，无须安装其它模块，并且在文件管理中就能查看结果。但是，该方法存在一个缺点，即受制于本地文件系统的限制。
- 之前，为了将URL映射为安全文件名，应用了多种限制，然而这样又可能会引发另一个问题，那就是一些URL会被映射为相同的文件名。
- 如下几个URL进行字符替换后就会得到相同的文件名
  - **http://example.com/?a*b**
  - **http://example.com/?a=b**
  - **http://example.com/?a|b**

# 3.3.6 磁盘缓存缺点

- 这就意味着，如果其中一个URL生成的缓存。其他几个个URL也会被认为已经生成缓存，因为它们映射到了同一个文件名。
- 另外，如果一些长URL只在第255个字符之后存在区别，截断后的版本也会被映射为相同的文件名。可是，URL的最大长度并没有明确限制，尽管在实践中，URL很少会超过2000个字符。
- 避免这些限制的一种解决方案是使用URL的哈希值作为文件名。

# 3.3.6 磁盘缓存缺点

- 尽管该方法可以带来一定改善，但是最终还是会面临许多文件系统具有的一个关键问题，那就是每个卷和每个目录下的文件数目是有限制的。
- 在FAT32文件系统中，每个目录的最大文件数是65535，文件系统可存储的文件总数也是有限制的。而一个大型的网站往往可能拥有超过1亿个网页。DiskCache方法想要通用的话，存在太多限制。
- 解决办法是可以把多个缓存网页合并到一个文件中，并使用B+树或类似的数据结构进行索引。无需自己实现，而是使用已有的键值对存储。

# 3.4 键值对存储缓存

- 如果需要缓存大量网页数据，可以使用高效的键值对存储，它要比传统关系型数据库甚至大多数NoSQL数据库更加易于扩展。
- 键值对存储类似于Python字典。存储中的每个对象都有一个键和一个值。在设计DiskCache时，键值对模型可以很好的解决该问题。
- 这里将使用键值对存储Redis(REmote Dictionary Server, 远程字典服务器)作为缓存。Redis可以很容易通过集群进行扩展，并且已经在一些大公司（例如Twitter）中作为海量缓存存储使用，例如Twitter的一个B树拥有大约65TB的分配堆内存。
- 如果要为每个文档提供更多的信息，则可以使用基于文档的数据库，例如MongoDB。

# 3.4.1 Redis简介

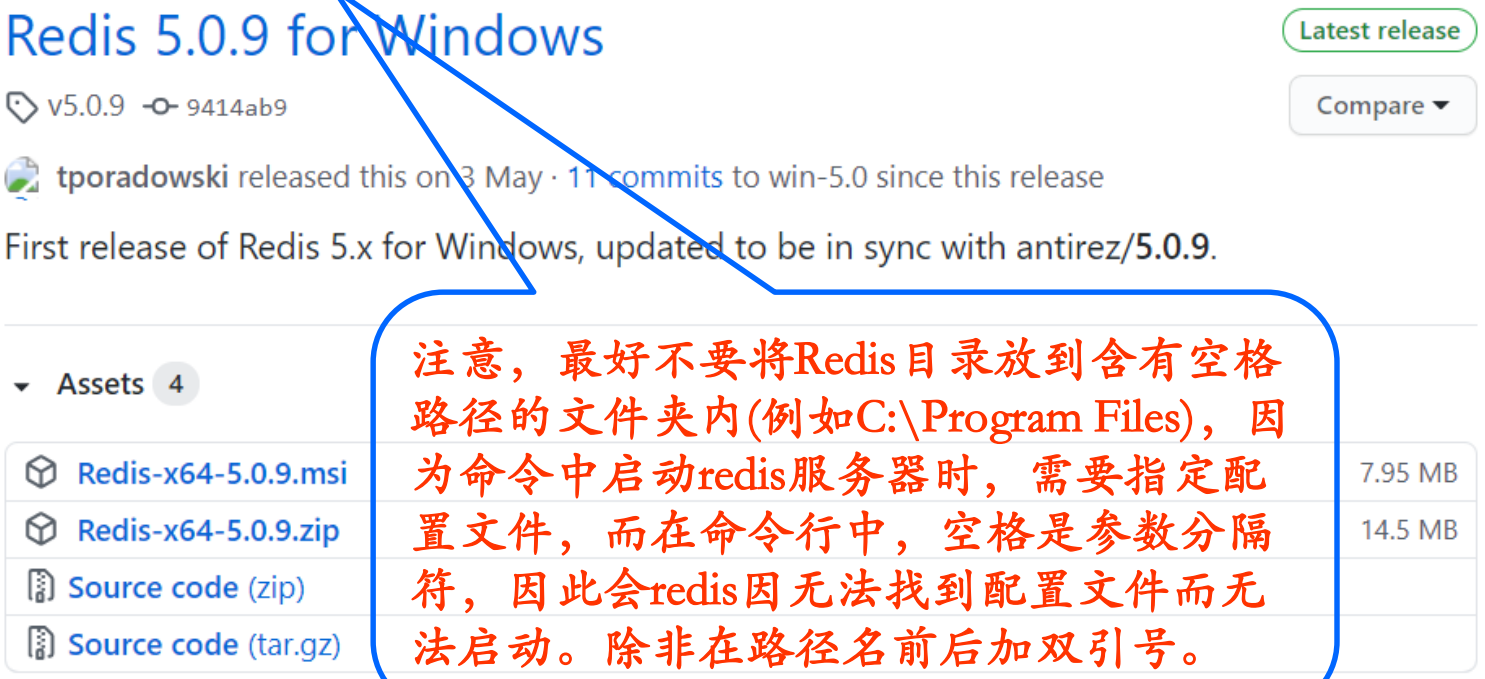
- Redis 是完全开源的，遵守 BSD 协议，是一个高性能的 key-value 数据库。
- Redis 的特点与优势：
  - Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
  - Redis不仅支持简单的key-value类型的数据，同时还提供list，set，zset（有序集合），hash等数据结构的存储。
  - Redis支持数据备份，即master-slave模式的数据备份。

# 3.4.1 Redis简介

- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
- 原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等特性。

# 3.4.2 Redis安装、配置、启动与登录

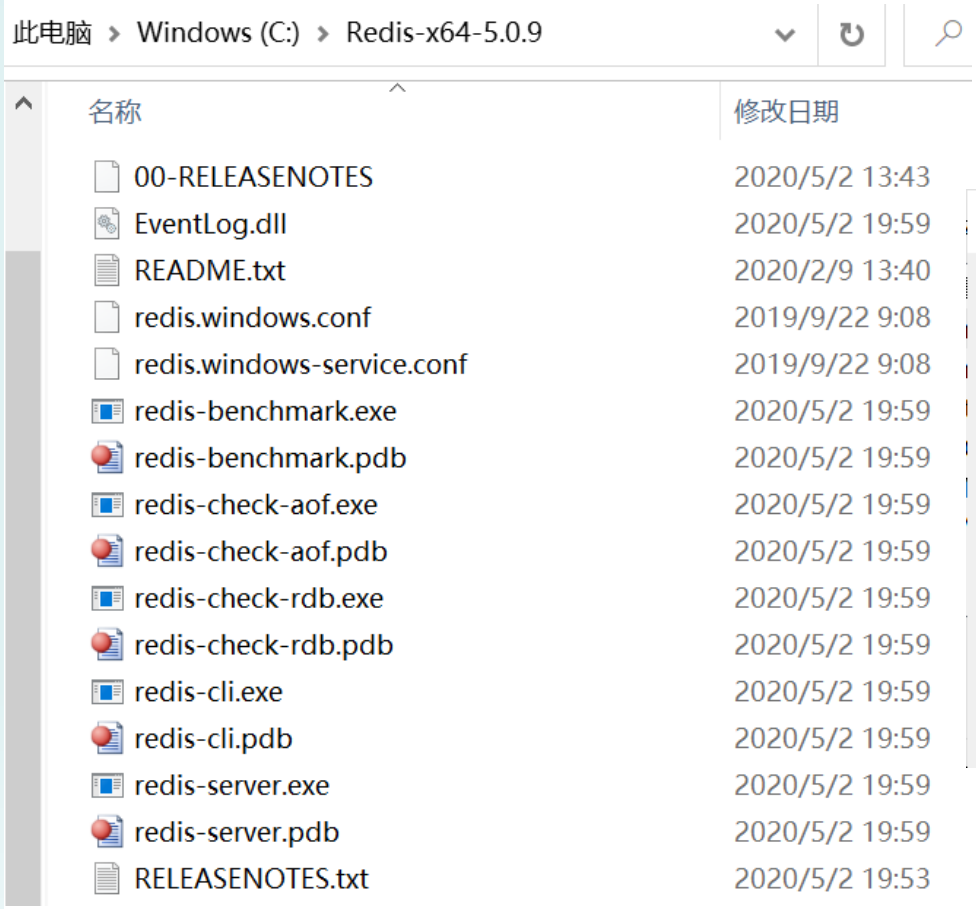
- **Window 下安装**
  - **下载：https://github.com/tporadowski/redis/releases**
  - **Redis 支持 32 位和 64 位。这个需要根据你系统平台的实际情况选择，这里下载Redis-x64-xxx.zip压缩包到 c:\Redis-x64-xxx。**

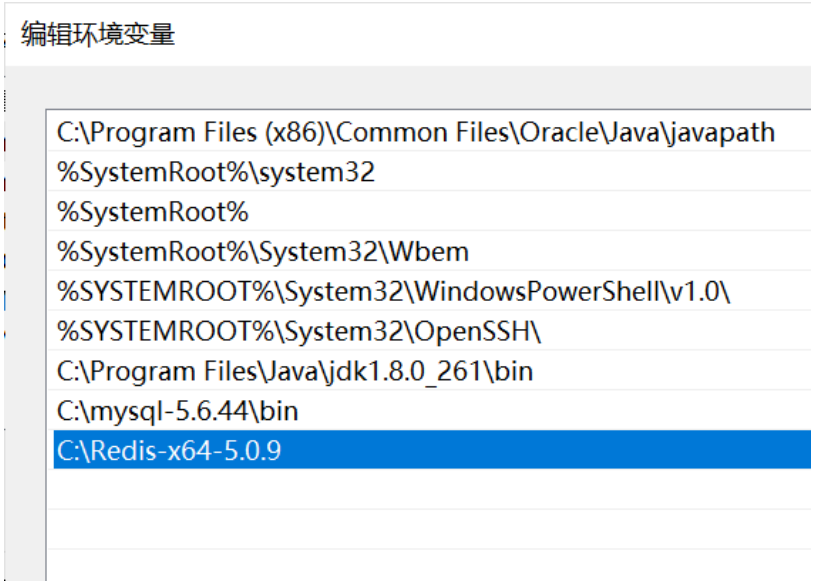


# 3.4.2 Redis安装、配置、启动与登录



- Window 下安装
  - 注册系统环境变量：把到 c:\Redis-x64-xxx的路径加到系统的环境变量path中



# 3.4.2 Redis安装、配置、启动与登录

- 配置Redis

  打开文本文件 redis.windows.conf

  或 redis.windows-service.conf，配置如下参数：

- 绑定允许访问的网络接口（默认bind 127.0.0.1）
  - bind IP地址(指定本机一个IP)
  - bind 127.0.0.1 (用于环回测试)
  - bind 0.0.0.0 (所有IP，包括环回测试)
- 指定访问密码（默认没有密码）

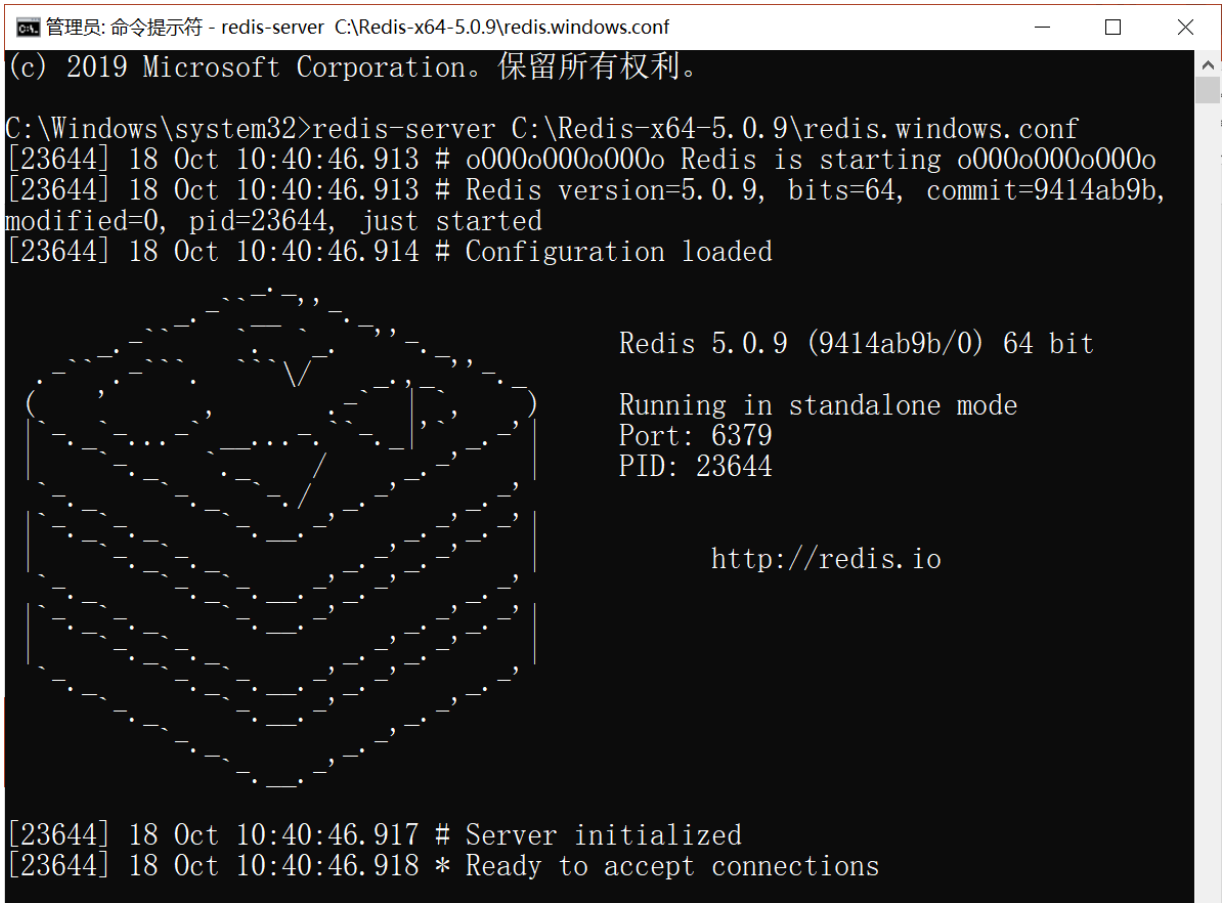
  requirepass ruanzl （注意移除requirepass前的#号）
- 指定数据存储文件名（默认为dump.rdb）

  dbfilename dump.rdb
- 指定数据存储目录（默认为./，即启动redis-server.exe时的工作目录）

  dir C:/Redis-x64-5.0.9

# 3.4.2 Redis安装、配置、启动与登录

- 启动Redis服务--指定配置文件为redis.windows.conf
  - 打开命令行CMD窗口（以管理员身份），运行：
    redis-server C:\Redis-x64-xxx\redis.windows.conf



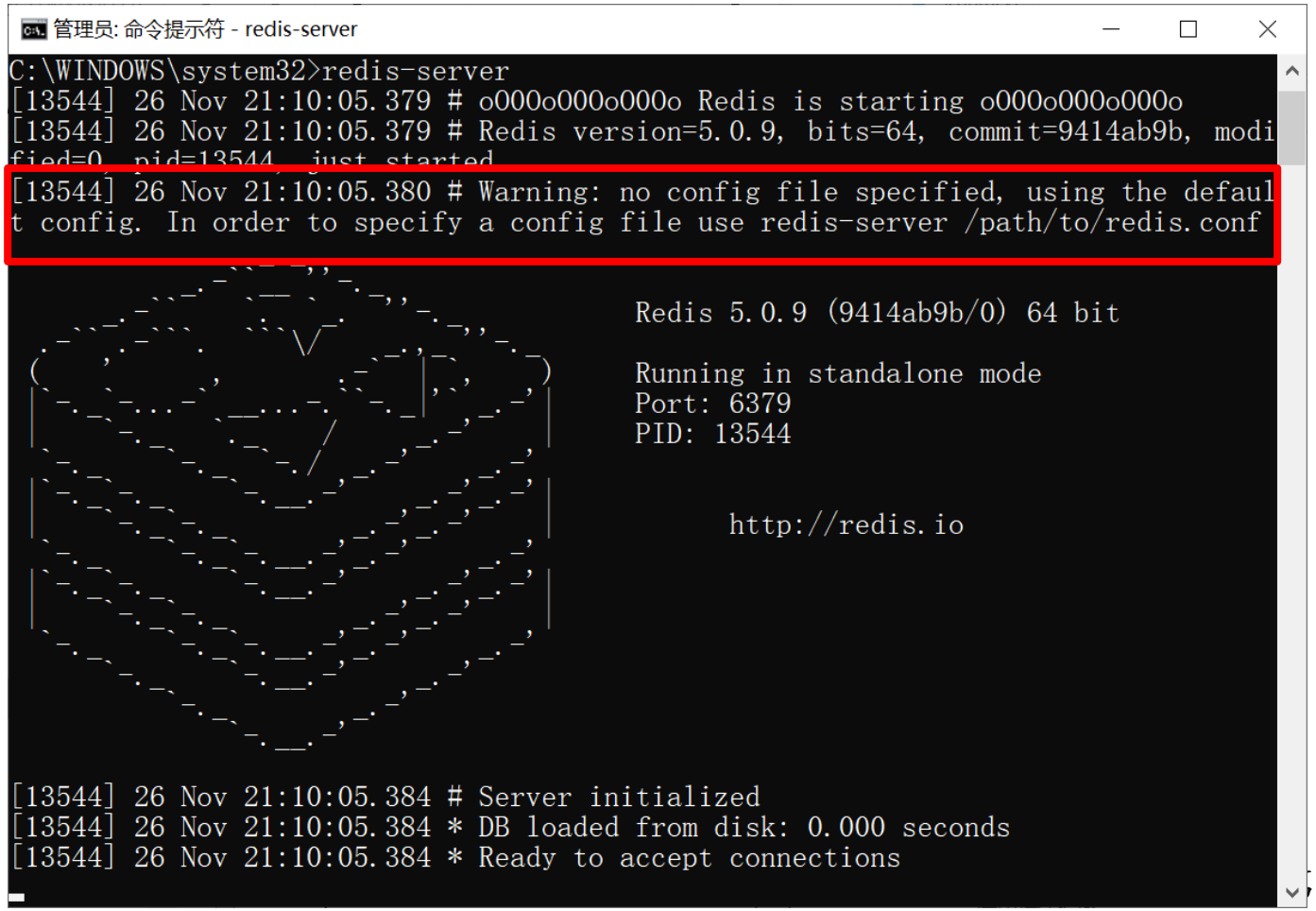
# 3.4.2 Redis安装、配置、启动与登录

- 启动Redis服务--指定配置文件为redis. windows-service.conf
  - 打开命令行CMD窗口（以管理员身份），运行：
    redis-server C:\Redis-x64-xxx\redis. windows-service.conf


注意，此时命令行中不输出服务器启动及运行时的相关提示信息，请不要关闭该窗口

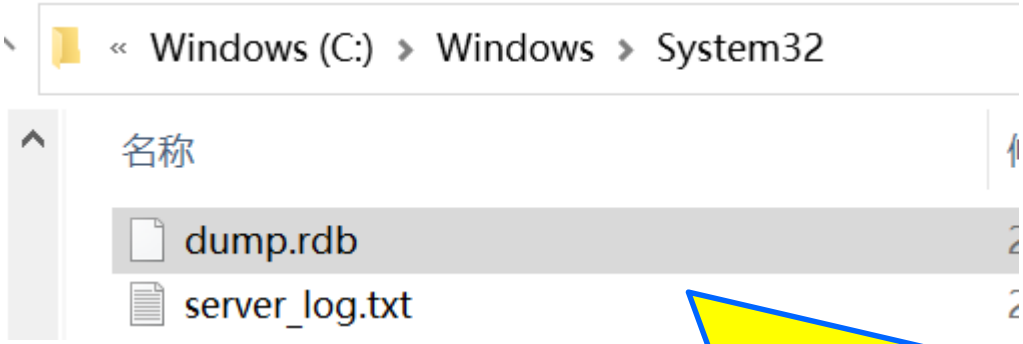
# 3.4.2 Redis安装、配置、启动与登录

- 启动Redis服务--不指定配置文件，则采用默认的配置
  - C:\WINDOWS\system32>redis-server



# 3.4.2 Redis安装、配置、启动与登录

- 启动Redis服务--不指定配置文件，则采用默认的配置
  - C:\WINDOWS\system32>redis-server




注意
- 此时数据库文件dump.rdb位于命令行当前工作目录，这里为C:\WINDOWS\system32
- 无法指定密码等相关配置

# 3.4.2 Redis安装、配置、启动与登录



- 测试Redis服务
  - 打开另一个CMD窗口（不要关闭，不然就无法访问服务端了），运行如下命令启动Redis客户端：

    redis-cli -h 127.0.0.1 -p 6379
  - 执行PING命令: 127.0.0.1:6379> ping

    输出：PONG

    说明已经成功安装了redis


如果服务器设置了密码，则有两种方式访问：

- 先登录，后确认授权：
  auth ruanzl
- 登录时指定密码：
  redis-cli -h IP -p 6379 -a ruanzl

# 3.4.2 Redis安装、配置、启动与登录

- 测试Redis服务
  - 打开另一个CMD窗口（不要关闭，不然就无法访问服务端了），运行如下命令启动Redis客户端：

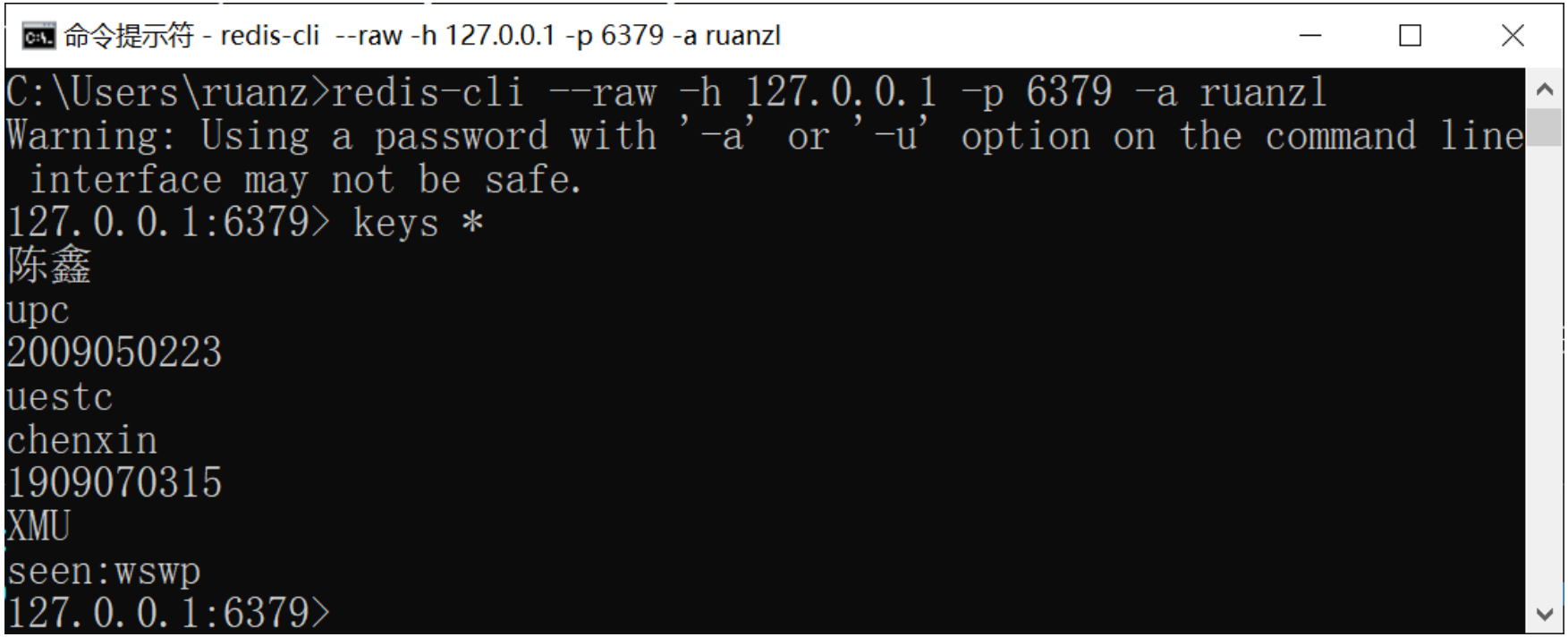
    **redis-cli -h 127.0.0.1 -p 6379**
  - 设置键值对: 127.0.0.1:6379> **set** myKey abc
  - 取出键值对: 127.0.0.1:6379> **get** myKey

```
127.0.0.1:6379> set myKey abc
OK
127.0.0.1:6379> get myKey
"abc"
127.0.0.1:6379> exit

C:\Users\ruanz>
```

# 3.4.2 Redis安装、配置、启动与登录

- 客户端中文乱码问题
  - 登录时，添加—raw选项即可

**redis-cli --raw -h 127.0.0.1 -p 6379**



# 3.4.2 Redis安装、配置、启动与登录

- 在客户端获取和修改服务器配置（在线配置）
  - 获取配置参数值：**config get** 参数名
  - 修改配置参数值：**config set** 参数名 参数值

# 3.4.2 Redis安装、配置、启动与登录



- Ubuntu系统下安装
  - 安装

    # sudo apt update

    # sudo apt install redis-server
  - 启动 Redis：# redis-server
  - 查看 redis 是否启动：# redis-cli

    以上命令将打开以下终端：redis 127.0.0.1:6379>

    其中127.0.0.1 是本机 IP ，6379 是 redis 服务端口。
  - 现在输入 PING 命令：redis 127.0.0.1:6379> ping

    PONG

    以上说明已经成功安装了redis。

# 3.4.3 Redis基本命令

- 启动Redis服务

C:\Users\ruanz> **redis-server** C:\Redis-x64-5.0.9\redis.windows.conf

- 启动Redis命令行客户端

C:\Users\ruanz>**redis-cli** -h 127.0.0.1 -p 6379

- Redis命令SET—添加键值对

127.0.0.1:6379> help set

SET key value [expiration EX seconds|PX milliseconds] [NX|XX]
summary: Set the string value of a key
since: 1.0.0
group: string

127.0.0.1:6379> **SET** UPC Great
OK
127.0.0.1:6379> **SET** SDU A+
OK

设置有效期为30秒

127.0.0.1:6379> **SET** UPC Great EX 30

过期后get UPC则显示“nil”，表示不存在。

# 3.4.3 Redis基本命令

- Redis中所有键都是string类型
- 简单类型值的类型和符合类型值的元素类型也都是string类型
- 字符串应用双引号作为开启和结束边界，但如果不含空白（一个或多个空格），双引号可以省略。
- 例如：
  - ✓ set " Albert Einstein" "伟大的物理学家"
  - ✓ set 三民主义 "民族 民权 民生"

- **Redis命令GET—获取指定键的值**

```
127.0.0.1:6379> help get

  GET key
  summary: Get the value of a key
  since: 1.0.0
  group: string
127.0.0.1:6379> GET UPC
"Great"
```

# 3.4.3 Redis基本命令



- **Redis命令KEYS—按给定模式（pattern）查询键**

```
127.0.0.1:6379> help keys

  KEYS pattern
  summary: Find all keys matching the given pattern
  since: 1.0.0
  group: generic
127.0.0.1:6379> KEYS UP*
1) "UPC"
127.0.0.1:6379> KEYS *U*
1) "SDU"
2) "UPC"
127.0.0.1:6379> SET XMU A+
OK
127.0.0.1:6379> KEYS *
1) "SDU"
2) "XMU"
3) "UPC"
```

# 3.4.3 Redis基本命令

- Redis列表操作
  - Redis 列表是按插入顺序排序的字符串列表。可以在列表的头部（左边）或尾部（右边）添加元素。
  - 列表可以包含超过 40 亿 个元素 ( $2^{32} - 1$ )。

| 常用命令 | 描述 |
| --- | --- |
| BLPOP / BRPOP | 移出并获取列表的第一个/最后一个元素。 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| LINDEX | 通过索引获取列表中的元素 |
| LINSERT | 在列表的元素前或者后插入元素 |
| LLEN | 获取列表长度 |
| LPOP / RPOP | 移出并获取列表的第一个/最后一个元素 |
| LPUSH / RPUSH | 将一个或多个值插入到列表头部/尾部。如果 key 不存在，那么在进行 push 操作前会创建一个空列表。 |
| LRANGE | 获取列表指定范围内的元素 |
| LREM | 移除列表元素 |
| LSET | 通过索引设置列表元素的值 |

# 3.4.3 Redis基本命令



- **Redis命令LPUSH / RPUSH—一个或多个值加入列首/尾**

```
127.0.0.1:6379> help LPUSH
  LPUSH key value [value ...]
  summary: Prepend one or multiple values to a list
  since: 1.0.0
  group: list

127.0.0.1:6379> LPUSH mylist 2 5 8
(integer) 3

127.0.0.1:6379> LRANGE mylist 0 -1
"8"
"5"
"2"

127.0.0.1:6379> LLEN mylist
(integer)3
```

```
127.0.0.1:6379> RPUSH mylist 4 7
(integer)5

127.0.0.1:6379> LRANGE mylist 0 -1
"8"
"5"
"2"
"4"
"7"
```

# 3.4.3 Redis基本命令



- **Redis命令LRANGE—从列表获取指定索引范围的元素**

```
127.0.0.1:6379> help LRANGE
  LRANGE key start stop
  summary: Get a range of elements from a list
  since: 1.0.0
  group: list
127.0.0.1:6379> LRANGE mylist 0 20
8
5
2
4
7
```

获取从第一个元素到倒数第一个元素，也就是获取全部元素

```
127.0.0.1:6379> LRANGE mylist 0 -1
8
5
2
4
7
```

# 3.4.3 Redis基本命令



- **Redis命令LINDEX—从列表获取一个指定索引的元素**

```
127.0.0.1:6379> LINDEX mylist 1
5
```

- **Redis命令LSET—修改列表中指定索引的元素值**

```
127.0.0.1:6379> LSET mylist 1 55
OK
127.0.0.1:6379> LRANGE mylist 0 -1
8
55
2
4
7
```

- **Redis命令LLEN—获取列表长度**

```
127.0.0.1:6379> LLEN mylist
5
```

# 3.4.3 Redis基本命令



- **Redis命令LPOP/RPOP—弹出并获取列表首/尾元素**

```
127.0.0.1:6379> help LPOP
  LPOP key
  summary: Remove and get the first element in a list
  since: 1.0.0
  group: list
127.0.0.1:6379> LPOP mylist
8
127.0.0.1:6379> LRANGE myList 0 -1
55
2
4
7
127.0.0.1:6379> RPOP myList
7
127.0.0.1:6379> LRANGE myList 0 -1
55
2
4
```

# 3.4.3 Redis基本命令

- Redis集合（Set）操作
  - Redis 的 Set 是 string 类型的无序集合。
  - 集合成员是唯一的，即集合中没有重复的数据。
  - 集合中最大的成员数为 $2^{32} - 1$ (4294967295), 每个集合可存储 40 多亿个成员。

| 常用命令 | 描述 |
| --- | --- |
| SADD | 向集合添加一个或多个成员 |
| SCARD | 获取集合的成员数 |
| SDIFF | 返回给定所有集合的差集 |
| SDIFFSTORE | 返回给定所有集合的差集并存储在 destination 中 |
| SINTER | 返回给定所有集合的交集 |
| SINTERSTORE | 返回给定所有集合的交集并存储在 destination 中 |
| SISMEMBER | 判断 member 元素是否是集合 key 的成员 |
| SMEMBERS | 返回集合中的所有成员 |

# 3.4.3 Redis基本命令

- **Redis集合（Set）操作**

| 常用命令 | 描述 |
| --- | --- |
| SMOVE | 将 member 元素从 source 集合移动到 destination 集合 |
| SPOP | 移除并返回集合中的一个随机元素 |
| SRANDMEMBER | 返回集合中一个或多个随机元素 |
| SREM | 移除集合中一个或多个成员 |
| SUNION | 返回所有给定集合的并集 |
| SUNIONSTORE | 所有给定集合的并集存储在 destination 集合中 |

# 3.4.3 Redis基本命令

- **Redis命令SADD—添加一个或多个成员到集合**

```
127.0.0.1:6379> help SADD

  SADD key member [member ...]
  summary: Add one or more members to a set
  since: 1.0.0
  group: set
  127.0.0.1:6379> SADD countries Russia
(integer) 1
127.0.0.1:6379> SADD countries China USA
(integer) 2
```

# 3.4.3 Redis基本命令

- **Redis命令SMEMBERS—获取集合所有成员**

```
127.0.0.1:6379> help SMEMBERS

  SMEMBERS key
  summary: Get all the members in a set
  since: 1.0.0
  group: set
127.0.0.1:6379> SMEMBERS countries
1) "China"
2) "USA"
3) "Russia"
```

# 3.4.3 Redis基本命令

- **Redis命令SRANDMEMBER—返回集合中一个或多个随机元素**

```
127.0.0.1:6379> help SRANDMEMBER

  SRANDMEMBER key [count]
  summary: Get one or multiple random members from a set
  since: 1.0.0
  group: set

127.0.0.1:6379> SRANDMEMBER countries 2
1) "Russia "
2) "USA"
127.0.0.1:6379> SRANDMEMBER countries 2
1) "China"
2) "Russia"
```

# 3.4.3 Redis基本命令

- **Redis命令SISMEMBER—判断给定值是否是集合成员**

```
127.0.0.1:6379> help SISMEMBER

  SISMEMBER key member
  summary: Determine if a given value is a member of a set
  since: 1.0.0
  group: set

127.0.0.1:6379> SISMEMBER countries Russia
(integer) 1
127.0.0.1:6379> SISMEMBER countries Tom
(integer) 0
```

# 3.4.3 Redis基本命令



- **Redis命令SREM—从集合移除一个或多个成员**

```
127.0.0.1:6379> help SREM

  SREM key member [member ...]
  summary: Remove one or more members from a set
  since: 1.0.0
  group: set

127.0.0.1:6379> SMEMBERS countries
1) "China"
2) "Russia"
3) "USA"
127.0.0.1:6379> SREM countries USA
(integer) 1
127.0.0.1:6379> SMEMBERS countries
1) "China"
2) "Russia"
```

# 3.4.3 Redis基本命令



- **Redis命令del—删除一个或多个键**

```
127.0.0.1:6379> help del

  DEL key [key ...]
  summary: Delete a key
  since: 1.0.0
  group: generic
127.0.0.1:6379> keys *
1) "SDU"
2) "XMU"
3) "UPC"
4) "countries"
127.0.0.1:6379> DEL UPC
(integer) 1
127.0.0.1:6379> keys *
1) "SDU"
2) "XMU"
3) "countries"
```

```
127.0.0.1:6379> DEL UPC SDU
(integer) 1
127.0.0.1:6379> keys *
1) "XMU"
2) "countries"
```

# 3.4.3 Redis基本命令



- **Redis命令dbsize—获取数据库中键值对的数量**

```
127.0.0.1:6379> help dbsize

  DBSIZE -
  summary: Return the number of keys in the selected database
  since: 1.0.0
  group: server

127.0.0.1:6379> dbsize
(integer) 3
```

# 3.4.3 Redis基本命令

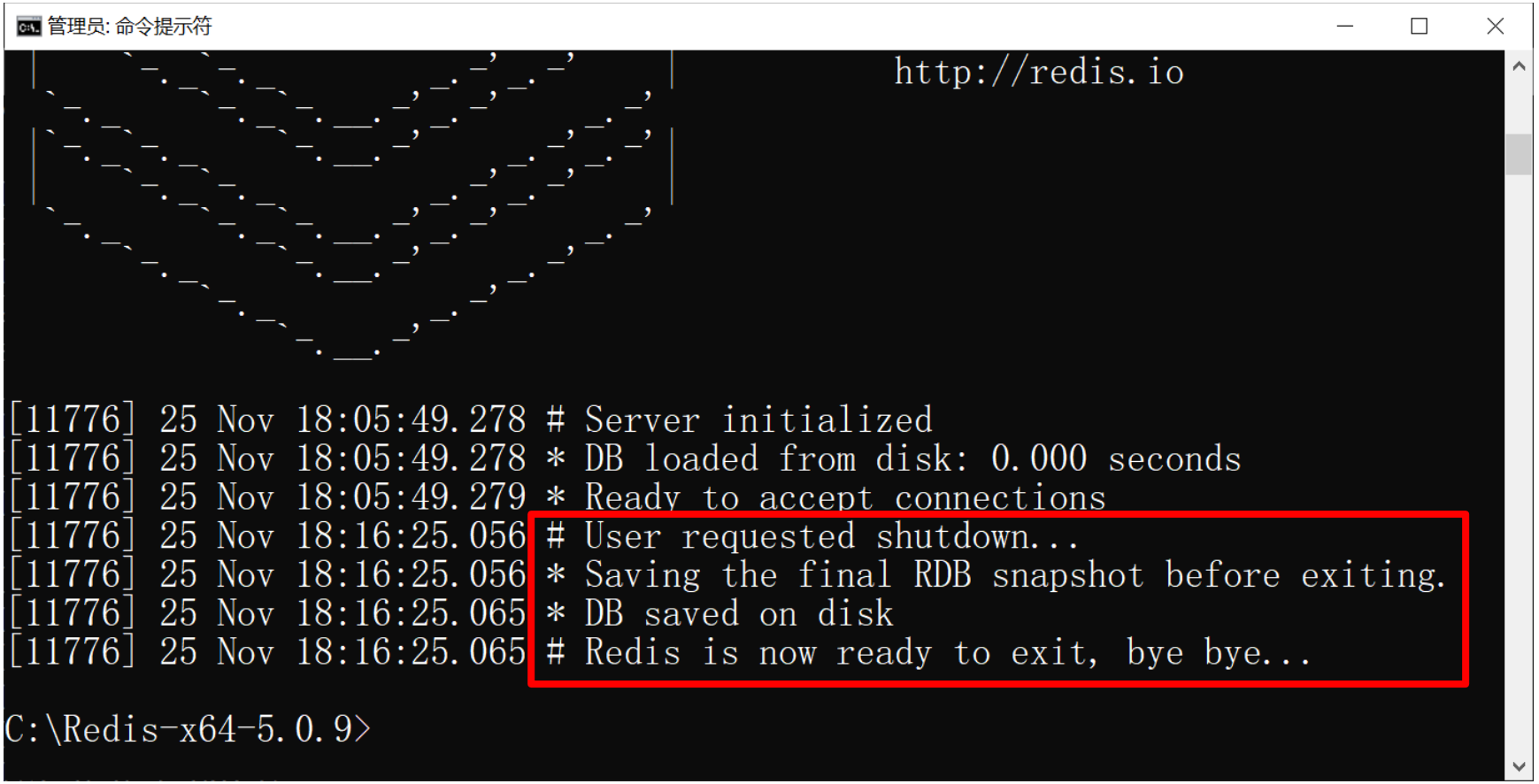
- **Redis命令flushdb—删除当前数据库中所有键值对，慎用**

```
127.0.0.1:6379> help flushdb

  FLUSHDB [ASYNC]
  summary: Remove all keys from the current database
  since: 1.0.0
  group: server

127.0.0.1:6379> FLUSHDB
OK
127.0.0.1:6379> keys *

127.0.0.1:6379>
```

如果不小心清空了redis，或者删除了某些键，可以执行**SHUTDOWN nosave**命令（后面即将介绍），不保存数据集到磁盘，并关闭服务器；或者直接关闭redis服务CMD窗口。

# 3.4.3 Redis基本命令

注意，如果按 ctrl+c 关闭redis服务，则会自动保存数据。



# 3.4.3 Redis基本命令



- **Redis命令shutdown—停止redis服务**

```
127.0.0.1:6379> help shutdown

  SHUTDOWN [NOSAVE|SAVE]
  summary: Synchronously save the dataset to disk and then shut down
the server
  since: 1.0.0
  group: server
```

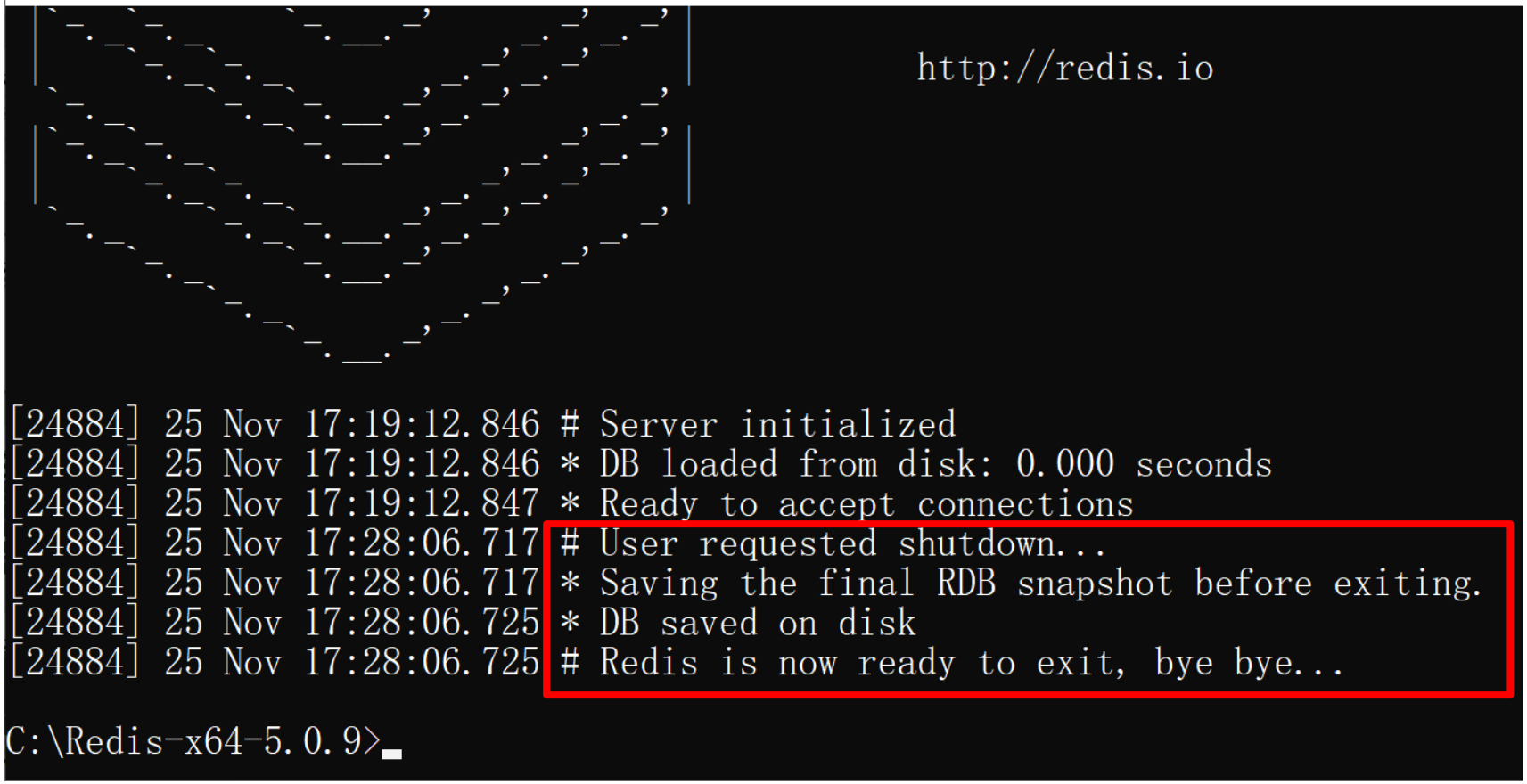
- SHUTDOWN：等价于 SHUTDOWN save，保存数据集到磁盘，然后停止redis服务
- SHUTDOWN nosave：不保存数据集到磁盘就停止redis服务

# 3.4.3 Redis基本命令

- **Redis命令shutdown—停止redis服务**

127.0.0.1:6379> SHUTDOWN
not connected>

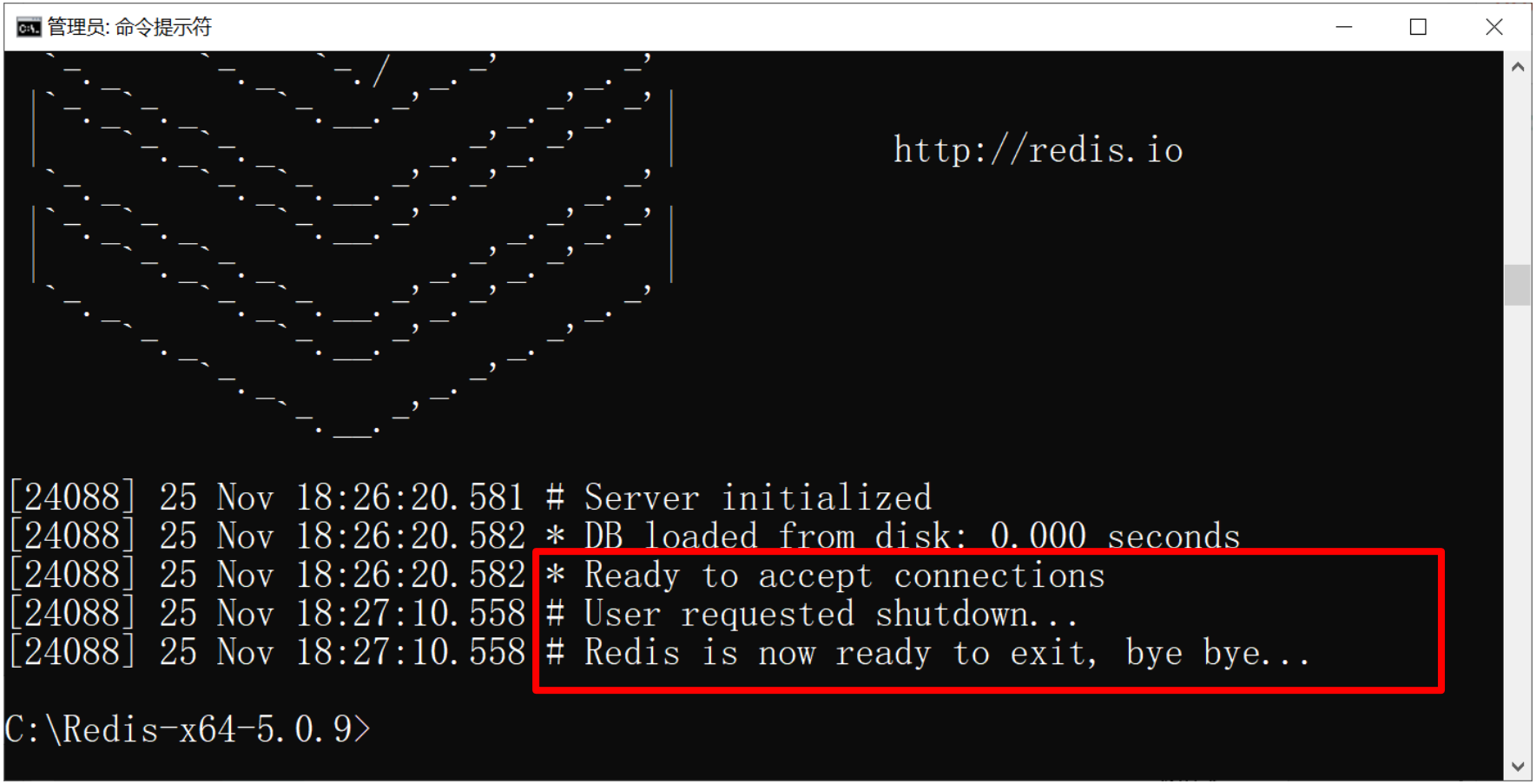
管理员: 命令提示符



# 3.4.3 Redis基本命令

- **Redis命令shutdown—停止redis服务**

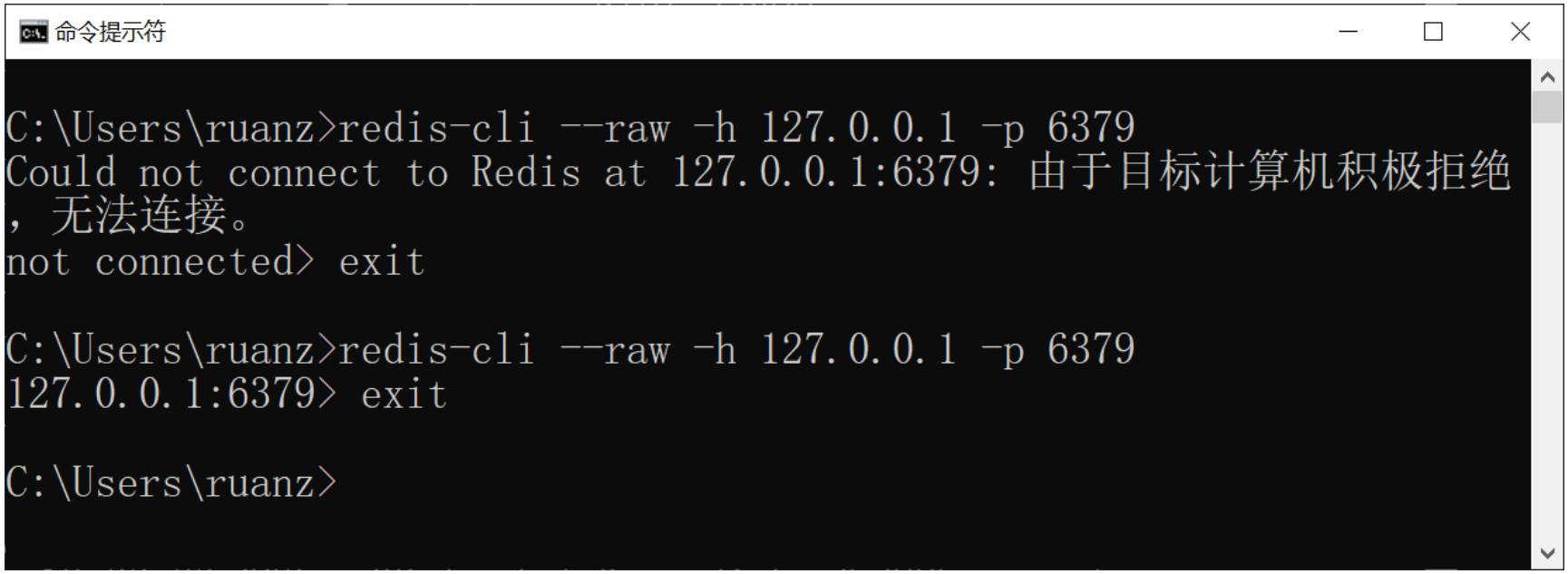
127.0.0.1:6379> SHUTDOWN nosave
not connected>



# 3.4.3 Redis基本命令

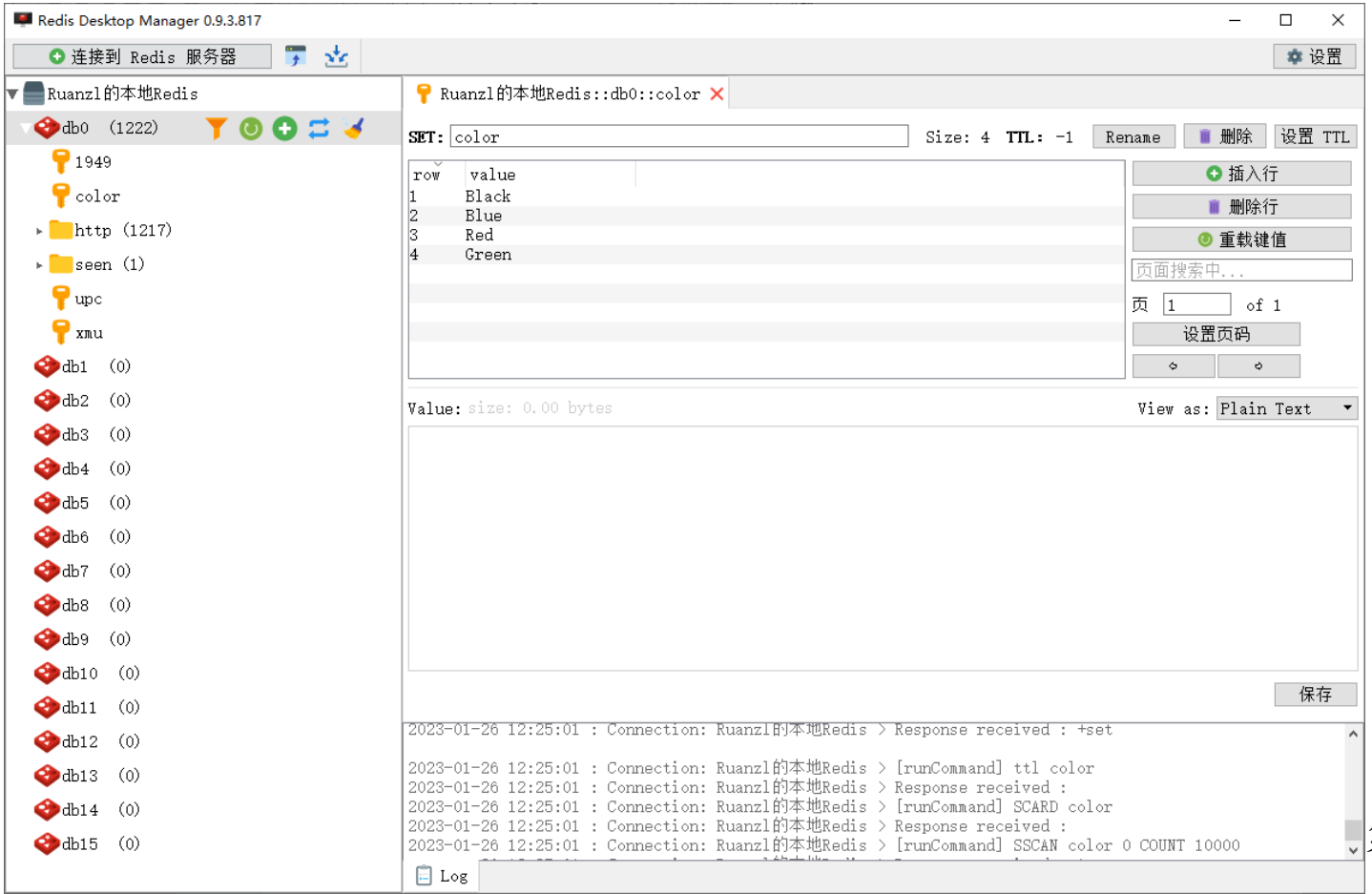
- **Redis命令exit—退出redis客户端**

127.0.0.1:6379> Exit
C:\Users\ruanz>



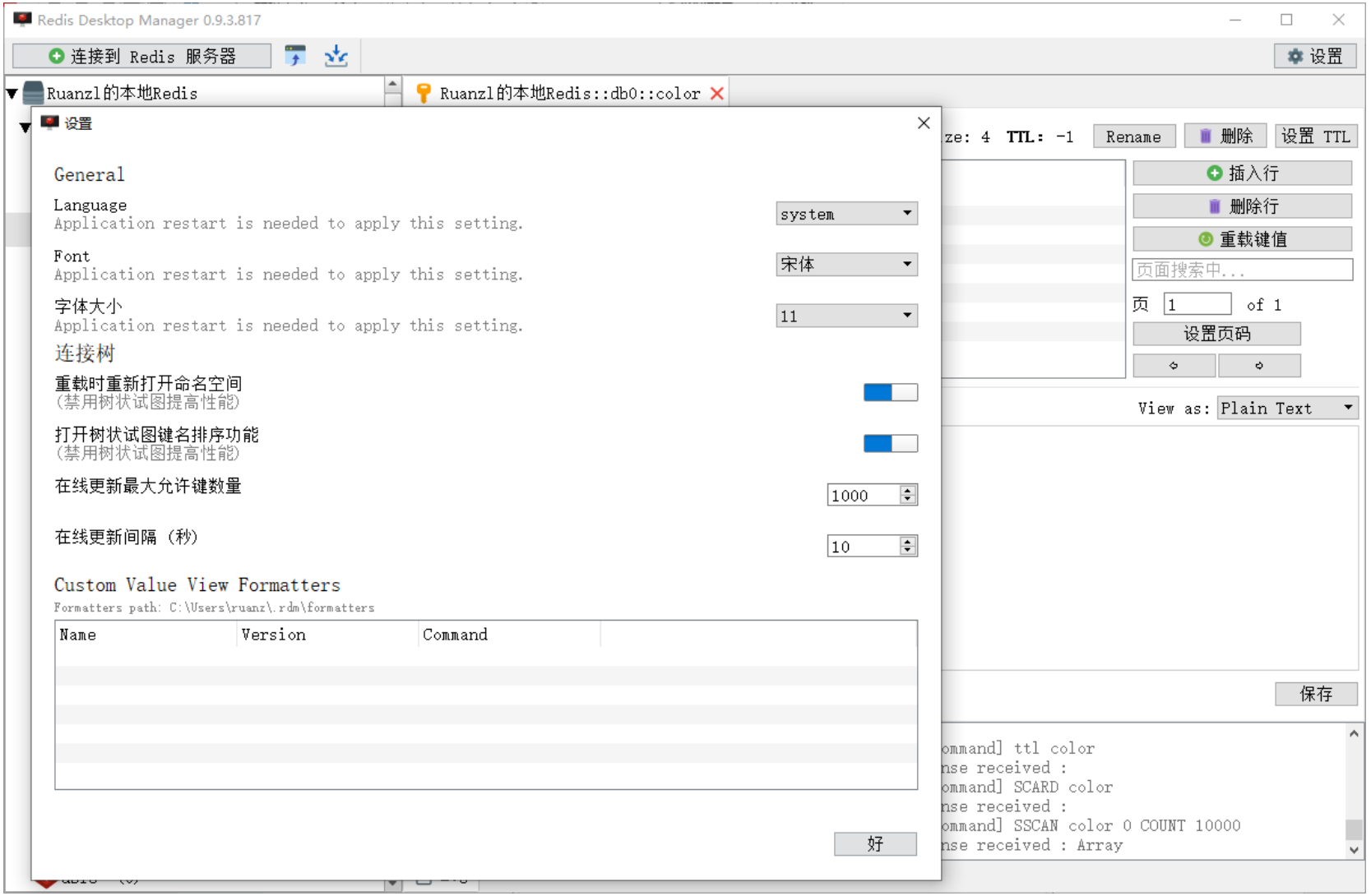
# 3.4.4 Redis可视化工具

- Redis Desktop Manager（RDM）是一款简单快速、跨平台的Redis桌面可视化管理工具。0.9.3.817是最后一个免费的版本。
- 连接到Redis服务器后，支持命令控制台的常用操作，包括键值对的添加、查询、重命名、删除等操作。



# 3.4.4 Redis可视化工具

- RDM基本设置：设置界面语言、字体等

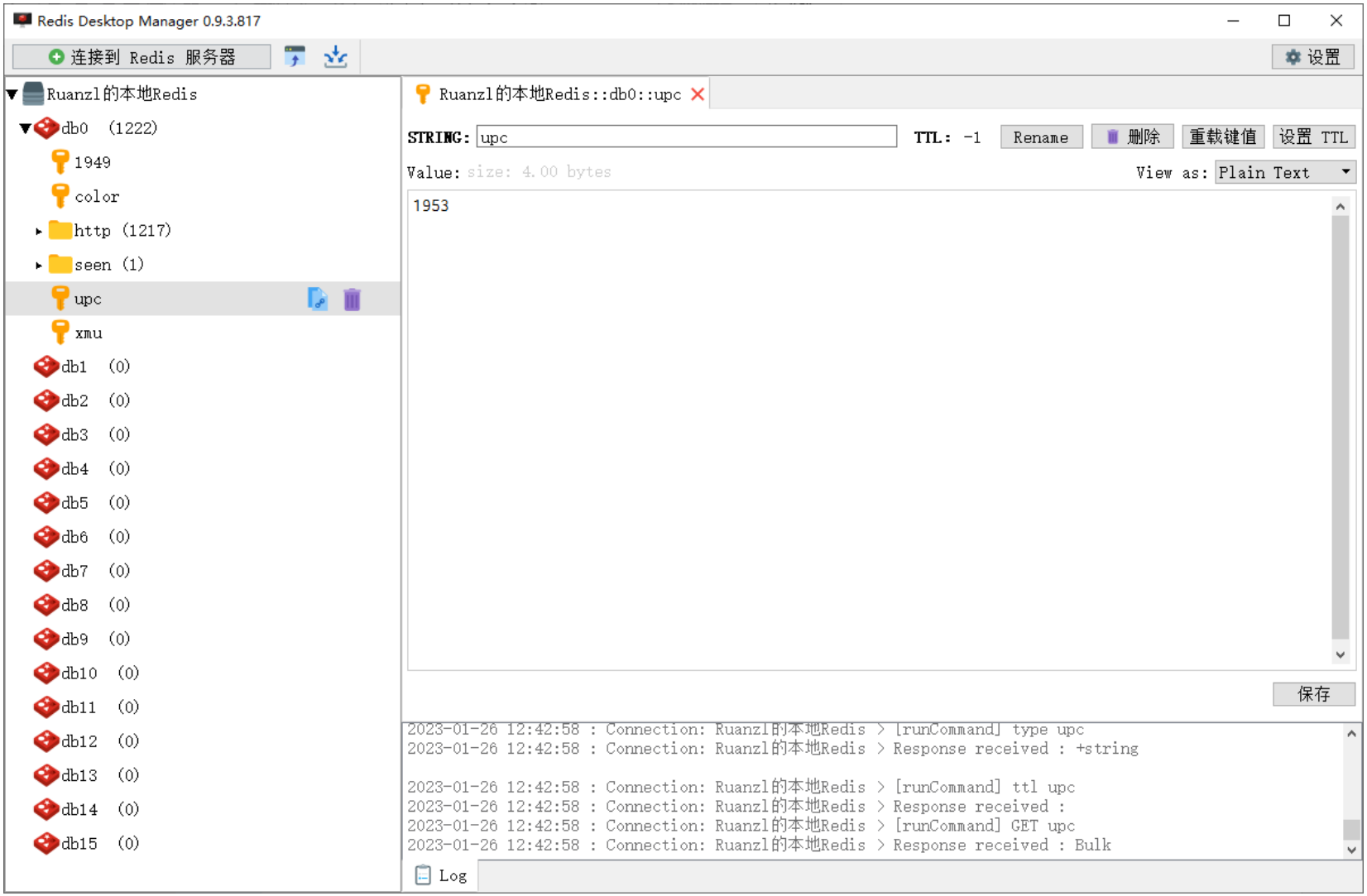


# 3.4.4 Redis可视化工具

- 连接到Redis服务器：设置连接名称、服务器IP地址、端口号及密码

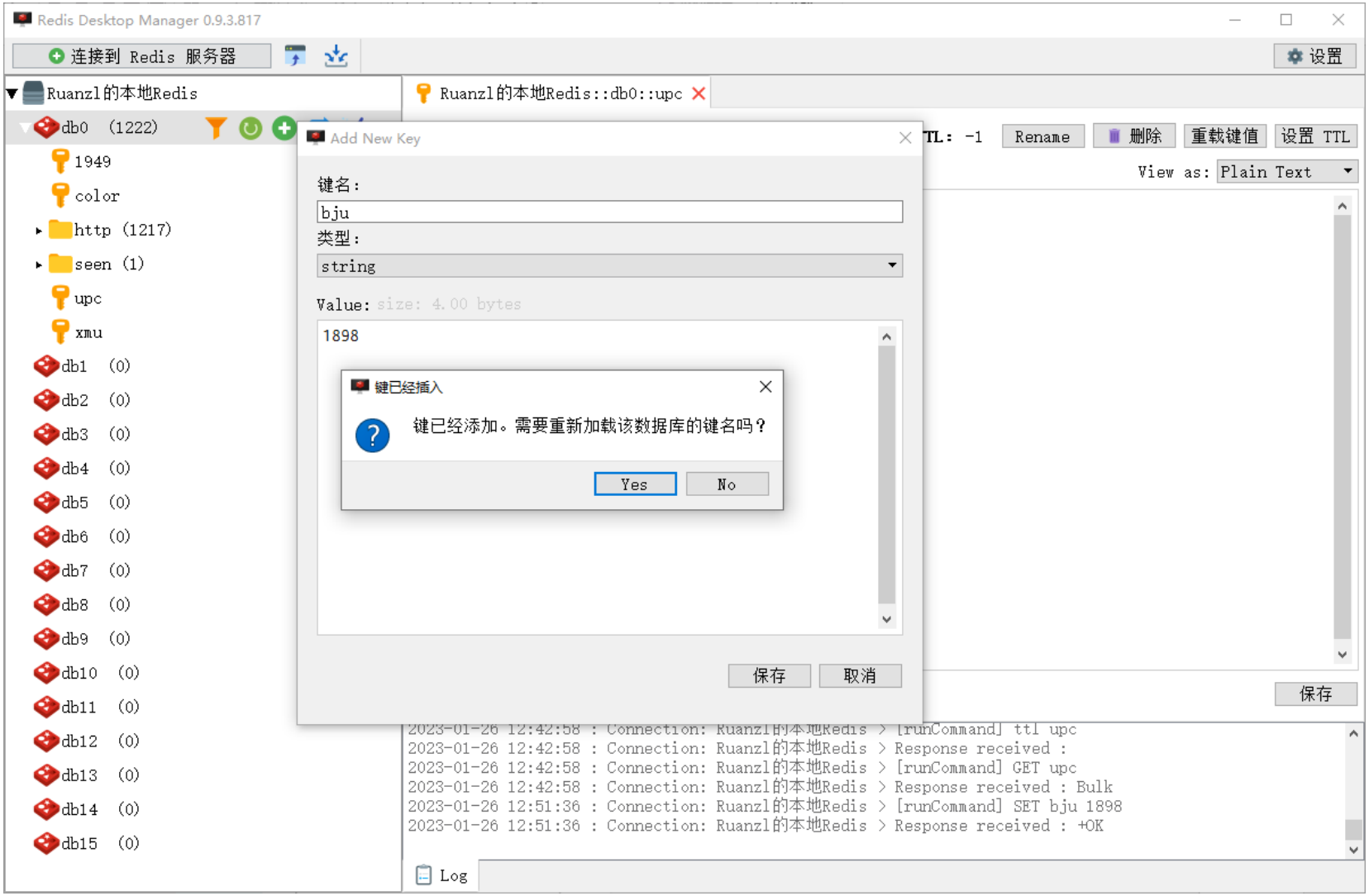
# 3.4.4 Redis可视化工具

- 查看键值对、重命名键、修改值、删除键值对



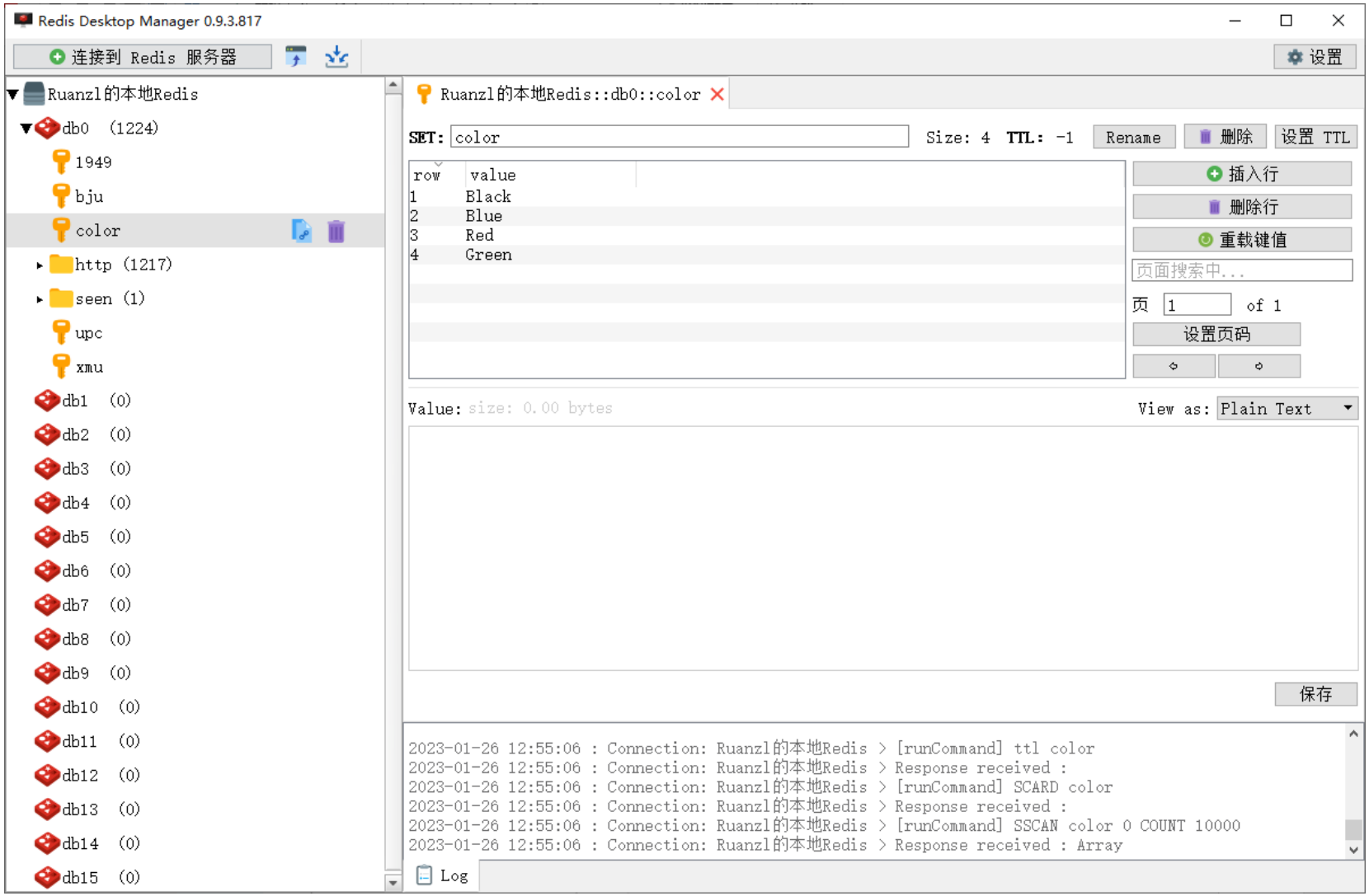
# 3.4.4 Redis可视化工具

- 添加键值对，值类型包括stirng、list、set等



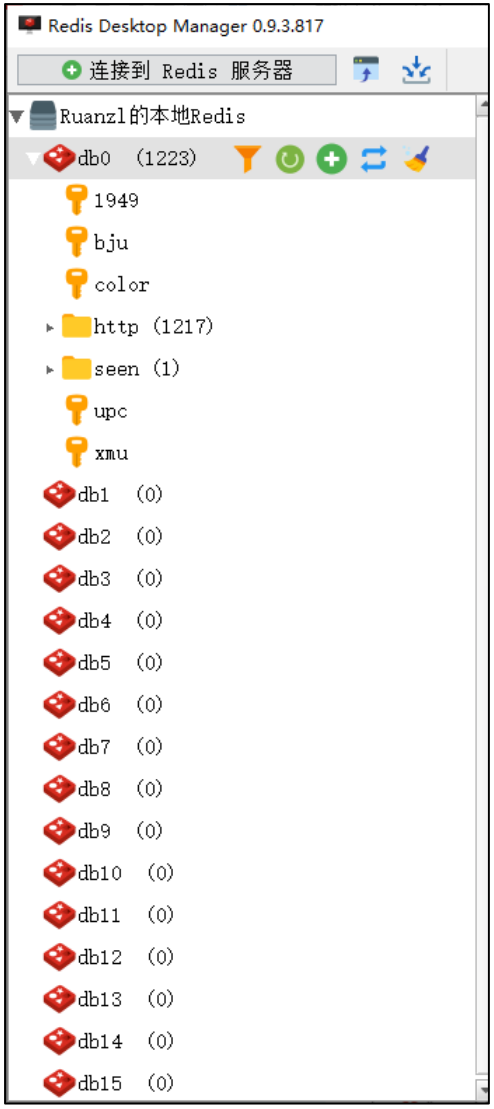
# 3.4.4 Redis可视化工具

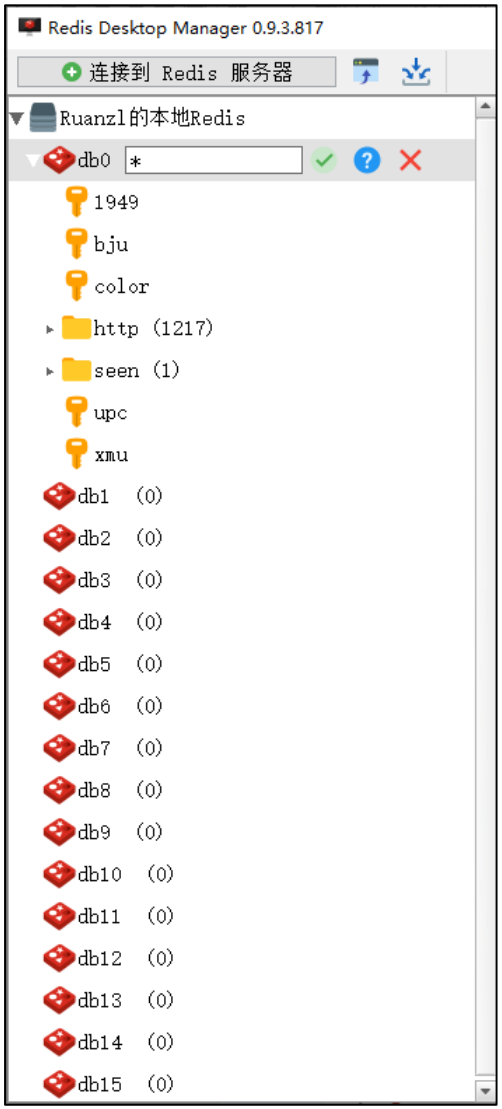
- 访问set：增删改元素

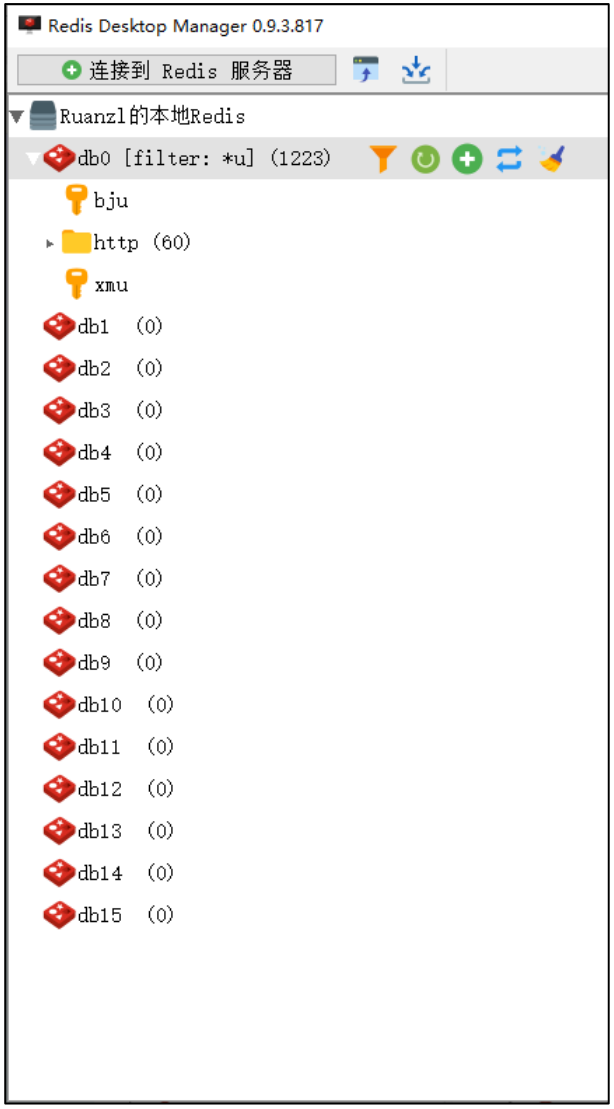


# 3.4.4 Redis可视化工具

- 过滤keys：默认不过滤

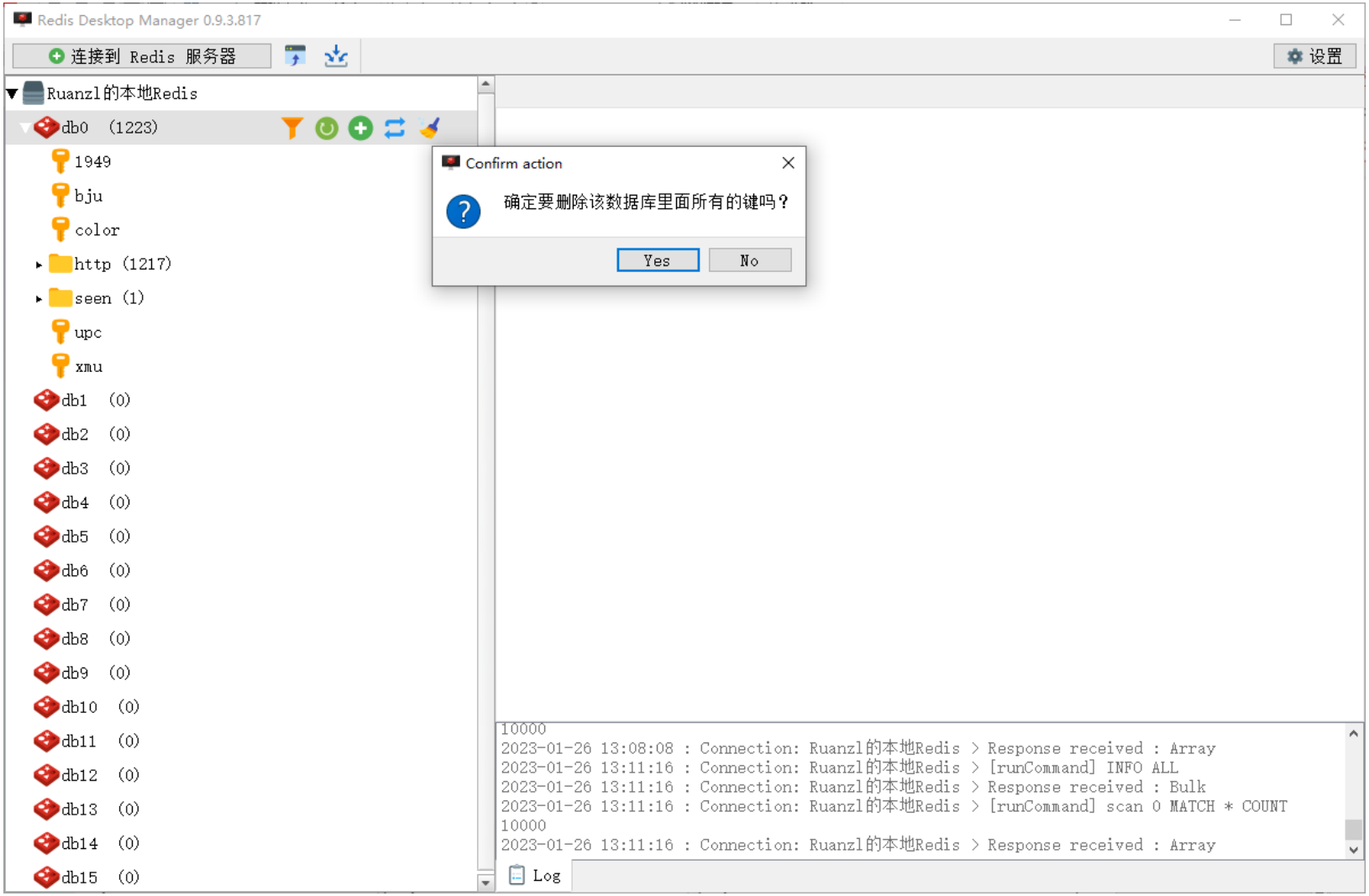


# 3.4.4 Redis可视化工具

- **Flush DB：删除需谨慎，三思而后行！**



# 3.4.5 Python中使用Redis

- 安装Redis的Python客户端（redis模块）
  - 安装：pip install redis
  - 测试Python客户端连接Redis和存储键值对

```
>>> import redis
>>> r=redis.StrictRedis(host='localhost',port=6379,db=0)
>>> r.set('test','UPC')
True
>>> r.get('test')
b'UPC'
```

```
>>> r_cli.ping()
True
>>> r_cli.dbsize()
1213
```

```
>>> r.close()
```

不再使用时关闭Redis链接

```
redis.StrictRedis(host='172.25.40.110', port=6379,
                  password='ruanzl', db=0)
```

# 3.4.5 Python中使用Redis

- 示例：将示例网站数据存入Redis，然后加载它。

Python 控制台

```
>>> import redis
>>> import json
>>> r = redis.StrictRedis(host='localhost', port=6379, db=0)
>>> url = 'http://example.python-scraping.com'
>>> html = '......'
>>> results = {'html': html, 'code': 200}
>>> r.set(url, json.dumps(results))
True
>>> r.get(url)
b'{"html": "......", "code": 200}'
```

- Redis中插入的数据类型可以是str、float或bytes，对于字典等其他数据类型可以使用json模块转换为字符串后再插入。
- 从Redis存储中返回的类型为bytes

# 3.4.5 Python中使用Redis



- 如果需要更新URL的内容，会发生什么？

```
>>> html = '<HTML></HTML>'
>>> results = {'html': html, 'code': 200}
>>> r.set(url, json.dumps(results))
True
>>> r.get(url)
b'{"html": "<HTML></HTML>", "code": 200}'
```

- **Redis的set命令只是简单地覆盖了之前的值，这对于类似网络爬虫这样的简单存储来说非常合适.**
- **对于的需求而言，只需要每个URL有一个内容集合即可，因此它能很好地映射为键值对。**

# 3.4.5 Python中使用Redis



- 看一下Redis存储里有什么，并且清除不需要的数据

```
>>> r.keys()
[b'test', b'http://example.python-scraping.com', b'myKey']
>>> r.keys('test')
[b'test']
>>> r.keys('test2')
[]
>>> r.delete('test')
1
>>> r.keys()
[b'http://example.python-scraping.com', b'myKey']
```

- keys()方法返回了所有可用键的列表
- delete()方法可以让传递一个或多个键，并从存储存储中删除它们。

# 3.4.5 Python中使用Redis



- 还可以删除所有的键

```
>>> r.flushdb()
True
>>> r.keys()
[]
```

- **Redis还有很多命令和工具**
- **请阅读Redis官方文档https://redis.io/documentation，或Python客户端文档，或其他网络资料（例如菜鸟教程https://www.runoob.com/redis/redis-tutorial.html）**


# 3.4.6 Redis缓存实现

- 使用与之前DiskCache类相同的类接口，构建Redis缓存

```
import json
from redis import StrictRedis
from datetime import timedelta

class RedisCache:
    def __init__(self, client=None, expires=timedelta(days=30), encoding='utf-8'):
        if client is None:  # 连接到redis
            self.client = StrictRedis(host='localhost', port=6379,
                                      password='ruanzl', db=0)
        else:
            self.client = client
        self.expires = expires
        self.encoding = encoding

    def close(self):
        if self.client is not None:
            self.client.close()
```

# 3.4.6 Redis缓存实现

- 使用与之前DiskCache类相同的类接口，构建Redis缓存

```
...
class RedisCache:
    def __init__(self, client=None, expires=timedelta(days=30), encoding='utf-8'): ...
    def close(self): …
    def __getitem__(self, url):
        """Load value from Redis for given URL"""
        record = self.client.get(url)  # type:bytes
        if record:
            return json.loads(record.decode(self.encoding))
        else:  # URL has not been cached
            raise KeyError(url + 'does not exist')

    def __setitem__(self, url, result):
        """Save value in redis for given URL"""
        data = json.dumps(result)
        data = bytes(data, self.encoding)
         self.client.setex(url, self.expires, data)
```

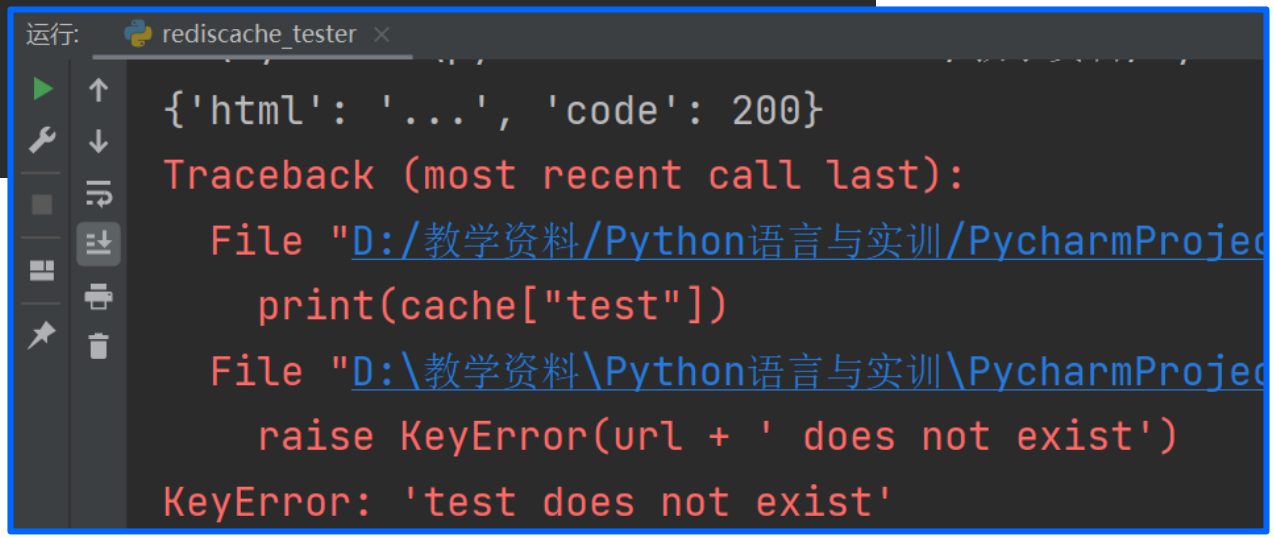

# 3.4.6 Redis缓存实现

- setex( )方法支持在设置键值时附带过期时间戳。Setex既可以接受date.timedelta，也可以接受以秒为单位数值。
- 这是一个非常方便的Redis功能，可以在指定时间后自动删除记录。这就意味着不再需要像DiskCache类那样手工检查记录是否在的过期规则内。
- 下面使用20秒的时间差在Python控制台进行尝试，观察缓存过期。

# 3.4.6 Redis缓存实现



```
from rediscache import RedisCache
from datetime import timedelta
import time
cache = RedisCache(expires=timedelta(seconds=20))
cache["test"] = {'html': '...', 'code': 200}
print(cache["test"])
time.sleep(20)
print(cache["test"])
cache.close()
```




- 结果显示缓存可以按照预期工作，可以在json、字典和Redis键值对存储间进行序列化与反序列化操作，并且能够对结果进行过期处理。

# 3.4.7 压缩

- 类似于磁盘缓存压缩，先对数据进行序列化，然后使用zlib进行压缩。

```
#---------------rediscache_zlib.py---------------------
import json
from redis import StrictRedis
from datetime import timedelta
import zlib

class RedisCache:
    def __init__(self, client=None, expires=timedelta(days=30), encoding='utf-8',
                 compress=True):
        if client is None:  # 连接到redis
            self.client = StrictRedis(host='localhost', port=6379,
                                      password='ruanzl', db=0)
        else:
            self.client = client
        self.expires = expires
        self.encoding = encoding
        self.compress = compress
```

# 3.4.7 压缩

```
……
class RedisCache:
    ……
   def close(self): …
    def __getitem__(self, url):
        record = self.client.get(url)  # type:bytes
        if record:
            if self.compress:  # 如果是压缩存储的，则先解压
                record = zlib.decompress(record)
            return json.loads(record.decode(self.encoding))
        else:  # URL has not been cached
            raise KeyError(url + ' does not exist')

    def __setitem__(self, url, result):
        data = json.dumps(result)
        data = bytes(data, self.encoding)
        if self.compress:  # 要求压缩，则存储前先进行压缩
            data = zlib.compress(data)
        self.client.setex(url, self.expires, data)
```

# 3.4.8 测试缓存

- 与使用DiskCache类的链接爬虫类似，对使用RedisCache类的链接爬虫进行测试，并对两种爬虫进行性能对比。

```
from advanced_link_crawler   import link_crawler, scrape_callback
from rediscache_zlib import RedisCache
import time
url = 'http://example.python-scraping.com'
regex = '/places/default/(index|view)/'
redis_cli = StrictRedis(host='localhost', port=6379, password='ruanzl', db=0)
redisCash = RedisCache(client=redis_cli)
start = time.time()
link_crawler(url, regex, scrape_callback=scrape_callback, cache= redisCash)
end = time.time()
seconds = end - start
hours = int(seconds / 3600)
mins = int(seconds % 3600 // 60)
secs = seconds % 60
print("Wall time: %d hours % d mins %f secs" % (hours, mins, secs))
redisCash.close()
```

# 3.4.8 测试缓存

- 首次运行结果

```
Downloading: http://example.python-scraping.com
Downloading: http://example.python-scraping.com/places/default/index/1
Downloading: http://example.python-scraping.com/places/default/index/2
……
Downloading: http://example.python-scraping.com/places/default/view/Afghanistan-1
http://example.python-scraping.com/places/default/view/Afghanistan-1 ['647,500 square kilometres', '29,121,286', 'AF', 'Afghanistan', 'Kabul', 'AS', '.af', 'AFN', 'Afghani', '93', '', '', 'fa-AF,ps,uz-AF,tk', 'TM IR TJ PK UZ ']
Wall time: 0 hours  22 mins 39.138469secs
```

# 3.4.8 测试缓存

- 第二次运行结果

```
Loaded from cache: http://example.python-scraping.com
Loaded from cache: http://example.python-scraping.com/places/default/index/1
Loaded from cache: http://example.python-scraping.com/places/default/index/2
……
Loaded from cache: http://example.python-
scraping.com/places/default/view/Afghanistan-1
http://example.python-scraping.com/places/default/view/Afghanistan-1 ['647,500
square kilometres', '29,121,286', 'AF', 'Afghanistan', 'Kabul', 'AS', '.af', 'AFN',
'Afghani', '93', '', '', 'fa-AF,ps,uz-AF,tk', 'TM IR TJ PK UZ ']
wall time: 0 hours  0 mins 2.410605 secs
```

- 在第一次迭代中花费的时间与DiskCache基本相同。不过，Redis的速度在缓存加载时才能真正体现出来，与未压缩的磁盘缓存系统相比，有着超过3倍的速度增长?
- 缓存代码可读性的增加，以及Redis集群在高可用性大数据解决方案上的可扩展能力，则是锦上添花。

# 3.4.8 测试缓存

- 注意：如果Redis中先前使用未压缩的RedisCache缓存了url键值对，则在运行带压缩缓存的爬虫程序时，应先清除缓存（执行FLUSHDB命令）；否则出现解压异常：

```
"D:\PyCharm Community Edition 2022.2.3\pythonProject\venv\Scripts\python.exe" "D:\PyCharm Community Editio
Traceback (most recent call last):
  File "D:\PyCharm Community Edition 2022.2.3\pythonProject\venv\代码\test8.py", line 48, in <module>
    link_crawler(url, regex, scrape_callback=scrape_callback,cache=redisCash)
  File "D:\PyCharm Community Edition 2022.2.3\pythonProject\venv\代码\test7.py", line 123, in link_crawler
    html=D(url=url,num_retries=num_retries)   #下载页面
  File "D:\PyCharm Community Edition 2022.2.3\pythonProject\venv\代码\test7.py", line 38, in __call__
    result=self.cache[url]
  File "D:\PyCharm Community Edition 2022.2.3\pythonProject\venv\代码\test8.py", line 29, in __getitem__
    record=zlib.decompress(record)
zlib.error: Error -3 while decompressing data: incorrect header check
```

# 谢谢大家！