引言

- 在之前的章节中，爬虫都是串行下载网页的，只有前一次下载完成之后才会启动新下载。在爬取规模较小的示例网站时，串行下载尚可应对，但面对大型网站时就会显得捉襟见肘了。在爬取拥有100万网页的大型网站时，假设以每秒一个网页的速度持续下载，耗时也要超过11天。如果可以同时下载多个网页，那么下载时间将会得到显著改善。
- 本章将介绍使用多线程和多进程这两种下载网页的方式，并将它们与串行下载的性能进行比较。

内容提要

- 100万个网页
- 串行爬虫
- 多线程爬虫
- 多进程爬虫
- 性能比较

4.1 100万个网页

- 想要测试并发下载的性能，最好要有一个大型的目标网站。为此，我们将使用Alexa提供的最受欢迎的100万个网站列表，该列表的排名根据安装了Alexa工具栏的用户得出。尽管只有少数用户使用了这个浏览器插件，其数据并不权威，但它能够提供可以爬取的大列表，对于这个测试来说已经足够好了。
- 可以通过浏览Alexa网站 https://www.alexa.com/topsites 获取该数据。也可以直接下载这一列表的压缩文件，这样就不用再去抓取Alexa网站的数据了：

  http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

4.1.1 解析Alexa列表

- Alexa网站列表是以电子表格的形式提供的，表格中包含两列内容，分别是排名和域名。

	A	B
1	1	google.com
2	2	facebook.com
3	3	youtube.com
4	4	yahoo.com
5	5	baidu.com
6	6	wikipedia.org
7	7	amazon.com
8	8	twitter.com
9	9	taobao.com
10	10	qq.com

- 抽取数据包含如下4个步骤：

1．下载.zip文件。

2．从.zip文件中提取出CSV文件。

3．解析CSV文件。

4．遍历CSV文件中的每一行，从中抽取出域名数据。

```
from zipfile import ZipFile
from io import BytesIO, TextIOWrapper
import requests , csv, pprint

url='http://s3.amazonaws.com/alexa-static/top-1m.csv.zip'
resp = requests.get(url)
urls = []  # top 1 million URL's will be stored in this list
with ZipFile(BytesIO(resp.content)) as zf:
    csv_filename = zf.namelist()[0]
    with zf.open(csv_filename) as fp:
        for _, website in csv.reader(TextIOWrapper(fp)):
            urls.append('http://' + website)

pprint.pprint(urls)
```

- 压缩数据是在使用BytesIO类封装之后，才传给ZipFile的。这是因为ZipFile需要一个类似文件的接口，而不是原生字节对象。
- 该.zip文件中只包含一个文件，所以直接选择第一个文件名即可。
- 使用TextIOWrapper读取CSV文件，它将协助处理编码和读取问题。该文件之后会被遍历，并将第二列中的域名数据添加到URL列表中。

```
from zipfile import ZipFile
from io import BytesIO, TextIOWrapper
import requests , csv, pprint

class AlexaCallback:
    def __init__(self, max_urls=500):
        self.max_urls = max_urls
        self.seed_url = 'http://s3.amazonaws.com/alexa-static/top-1m.csv.zip'
        self.urls = []
    def __call__(self):
        resp = requests.get(self.seed_url)
        with ZipFile(BytesIO(resp.content)) as zf:
            csv_filename = zf.namelist()[0]
            with zf.open(csv_filename) as fp:
                for _, website in csv.reader(TextIOWrapper(fp)):
                    self.urls.append('http://' + website)
                    if len(self.urls) == self.max_urls:
                        break
        return self.urls
if __name__ == '__main__':
    alexa = AlexaCallback(max_urls=100)
    urls = alexa()  # alexa.__call__()

    pprint.pprint(urls)
```

要想在之前开发的爬虫中复用上述功能，还需将程序修改为一个简单的回调类。

```python
from zipfile import ZipFile
from io import TextIOWrapper
import  csv, pprint

class AlexaCallback:
    def __init__(self, max_urls=500):
        self.max_urls = max_urls
        self.filePath = 'd:/top-1m.csv.zip'
        self.urls = []

    def __call__(self):
        with ZipFile(self.filePath) as zf:
            csv_filename = zf.namelist()[0]
            with zf.open(csv_filename) as fp:
                for _, website in csv.reader(TextIOWrapper(fp)):
                    self.urls.append('http://' + website)
                    if len(self.urls) == self.max_urls:
                        break
        return self.urls

if __name__ == '__main__':
    alexa = AlexaCallback(max_urls=10)
    urls = alexa()  # alexa.__call__()
    pprint.pprint(urls)
```

也可以从事先下载后的压缩文件中读取数据。

```python
import csv
import pprint


class AlexaCallback:
    def __init__(self, max_urls=500):
        self.max_urls = max_urls
        self.filePath = 'top-1m.csv'
        self.urls = []

    def __call__(self):
        with open(self.filePath) as fp:
            for _, website in csv.reader(fp):
                self.urls.append('http://' + website)
                if len(self.urls) == self.max_urls:
                    break
        return self.urls


if __name__ == '__main__':
    alexa = AlexaCallback(max_urls=10)
    alexa()  # alexa.__call__()
    pprint.pprint(alexa.urls)
```

直接从csv文件中读取数据。

4.2 串行爬虫

- 现在可以对之前开发的链接爬虫进行少量修改，使用AlexaCallback串行下载Alexa的前500个URL。
  - 为了更新链接爬虫，现在需要传入起始URL列表。
  - 还需要更新对每个站点中robots.txt的处理方式。我们使用一个简单的字典来存储每个域名的解析器。
  - 此外，由于一些网站没有robots.txt文件，因此我们添加了一些额外的错误处理代码段，当在无法找到robots.txt文件时，仍然可以继续爬取。
  - 最后，添加了socket.setdefaulttimeout(60)，用于为robotparser以及第3章中Downloader类额外的timeout参数处理超时。

```
import sys
sys.path.append(r'..\ch3') # 注意，还需要在PyCharm中将ch3包设为源根
from downloader import Downloader
import re , time , requests , socket
from urllib.parse import urljoin, urlsplit
from urllib import robotparser
from alexaCallback2 import AlexaCallback
from commonFunctions import get_links, get_robots_parse, show_time_diff
def link_crawler(start_urls, link_regex, scrape_callback=None, delay=5,
                 user_agent='wswp', proxies=None, cache={},  num_retries=2): …

if __name__ == '__main__':
    timeout = 10
    socket.setdefaulttimeout(timeout)
    alexa = AlexaCallback(max_urls=500)
    start_urls = alexa()
    regex = '$^'  # 使用'$^'作为模式，避免收集每个页面的链接。
    start = time.time()
    link_crawler(start_urls, regex, scrape_callback=None, cache={})
    end = time.time()
    show_time_diff(end, start)
```

4.2 串行爬虫

- 修改get_robots_parse函数，在其中添加异常处理：

```
def get_robots_parse(robots_url):
    """ Return the robots parser object using the robots_url """
    try:
        rp = robotparser.RobotFileParser()  # 创建对象
        rp.set_url(robots_url)
        rp.read()  # 读取URL指定的文件
        #  re.read()方法以uft-8解码，若网站的robots.txt文件不是按uft-8编码的，则读取会出错。
        return rp
    except (UnicodeDecodeError, Exception) as e:
        return None
```

- 修改link_crawler函数，其中添加有关robots文件的处理

```
def link_crawler(start_urls, link_regex, scrape_callback=None, delay=5,
                 user_agent='wswp', proxies=None, cache={},  num_retries=2):
    crawl_queue = start_urls.copy()  # 注意如果直接crawl_queue = start_urls，
                                     则两个引用的是同一列表
    seen = set(crawl_queue)  # 根据列表创建一个新集合
    rp_dict = dict()  # 该字典用来存储每个域名的解析器
    for start_url in start_urls:
        rp = get_robots_parse(f'{start_url}/robots.txt')
        rp_dict[start_url] = rp

    D = Downloader(delay=delay, user_agent=user_agent, proxies=proxies, cache=cache)
    while crawl_queue:
        url = crawl_queue.pop()  # 弹出队列首元素（链接）
        start_url = 'http://' + urlsplit(url).netloc
        rp = rp_dict[start_url]  # 取出url所在网站的域名所对应的rp(即robots解析器)

        # 如果存在robots.txt文件且其中禁止访问该url，则跳过下载
        if (rp is not None ) and (not rp.can_fetch(user_agent, url)):
            print(f'Skipping: {url}')
            continue
        html = D(url, num_retries=num_retries)
        ......
```

['http://google.com', 'http://youtube.com', ...,'http://discordapp.com']
Downloading: http://discordapp.com
Download error: HTTPConnectionPool(host='discordapp.com', port=80): Max retries exceeded with url: / (Caused by ConnectTimeoutError
(<urllib3.connection.HTTPConnection object at 0x0000021B4C47A850>, 'Connection to discordapp.com timed out. (connect timeout=10)'))
code = 404
Downloading: http://zol.com.cn
encoding= GB2312
.....
Downloading: http://youtube.com
Download error: HTTPConnectionPool(host='youtube.com', port=80): Max retries exceeded with url: / (Caused by ConnectTimeoutError
(<urllib3.connection.HTTPConnection object at 0x0181DA60>, 'Connection to youtube.com timed out. (connect timeout=10)'))
Downloading: http://google.com
Download error: HTTPConnectionPool(host='google.com', port=80): Max retries exceeded with url: / (Caused by ConnectTimeoutError
(<urllib3.connection.HTTPConnection object at 0x01846178>, 'Connection to google.com timed out. (connect timeout=10)'))
Wall time: 0 hours 39 mins 15.783550 secs

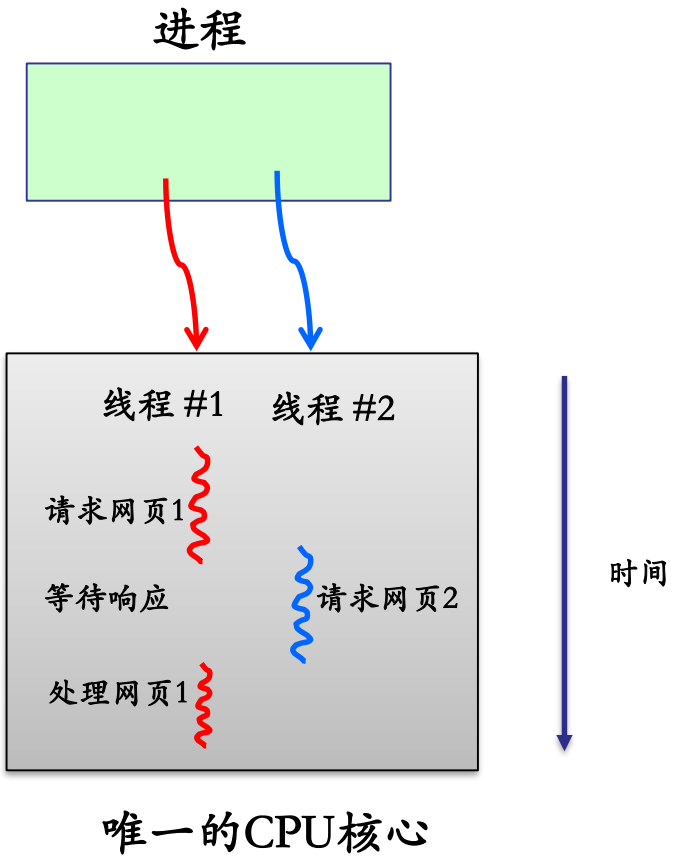
在串行下载时，只爬取每个站点的第一个页面所花费的时间，大约为每个URL平均5秒（未测试robots.txt文件）。

4.3 多线程爬虫

- 现在，将串行下载网页的爬虫扩展成并行下载。
- 需要注意的是，如果多线程爬虫请求内容速度过快，可能会造成服务器过载，或是IP地址被封禁。
- 为了避免这一问题，需要为爬虫设置一个delay标识，用于设定请求同一域名时的最小时间间隔。
- 作为本章示例的Alexa网站列表，由于包含了100万个不同的域名，因而不会出现该问题。但是，当需要爬取同一域名下的不同网页时，就需要注意两次下载之间至少需要1秒钟的延时。

4.3.1 线程和进程如何工作

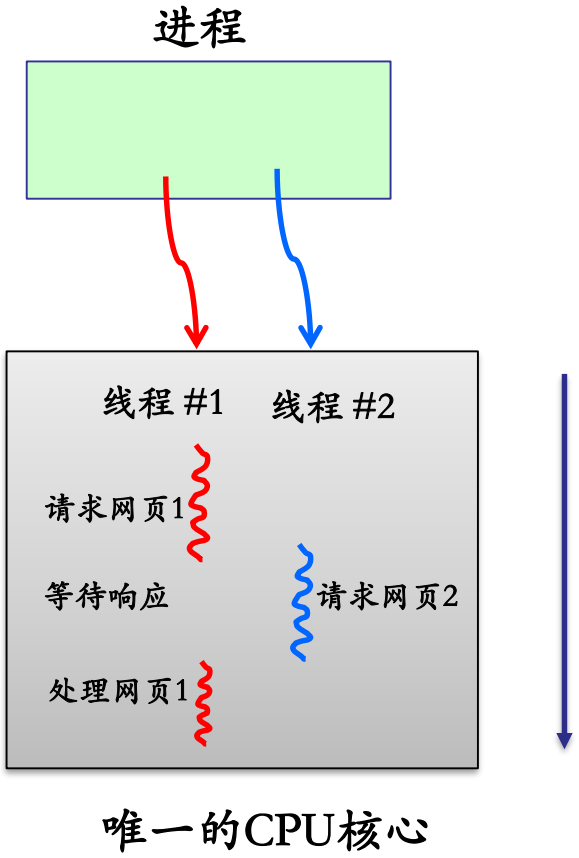
- 一个包含有多个线程的进程的执行过程



- 当运行Python脚本或其他计算机程序时，就会创建包含有代码、状态以及堆栈的进程。这些进程通过计算机的一个或多个CPU核心来执行。
- 不过，同一时刻每个核心只会执行一个进程，然后在不同进程间快速切换，这样就给人以多个程序同时运行的感觉。
- 同理，在一个进程中，程序的执行也是在不同线程间进行切换的，每个线程执行程序的不同部分。

4.3.1 线程和进程如何工作

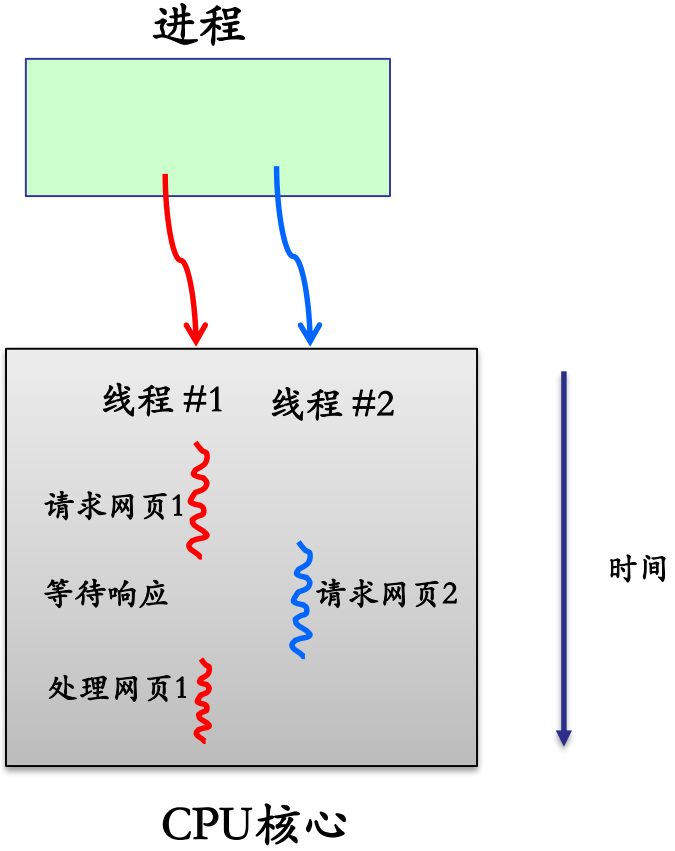
- 一个包含有多个线程的进程的执行过程



- 这就意味着当一个线程等待网页下载时，进程可以切换到其他线程执行，避免浪费CPU周期。
- 因此，为了充分利用计算机中的所有计算资源尽可能快地下载数据，需要将下载分发到多个进程和线程中。

4.3.1 线程和进程如何工作

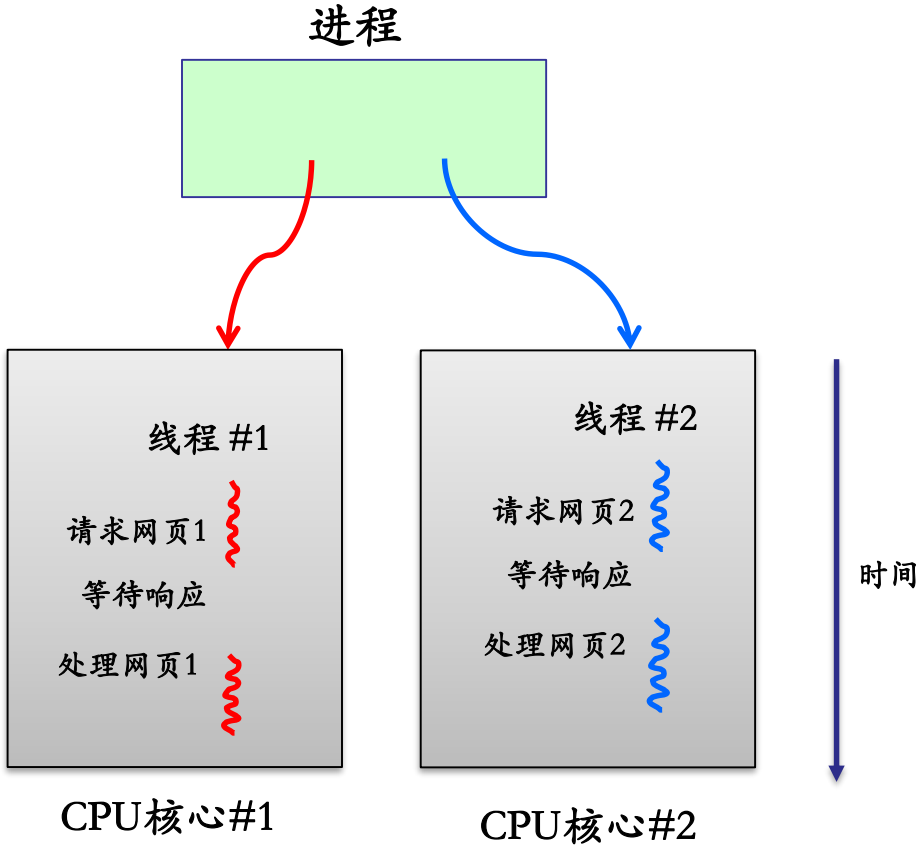
- 多线程与多进程的区别



（一个单核CPU）多线程特点：各个线程在唯一的核心中并发执行

4.3.1 线程和进程如何工作

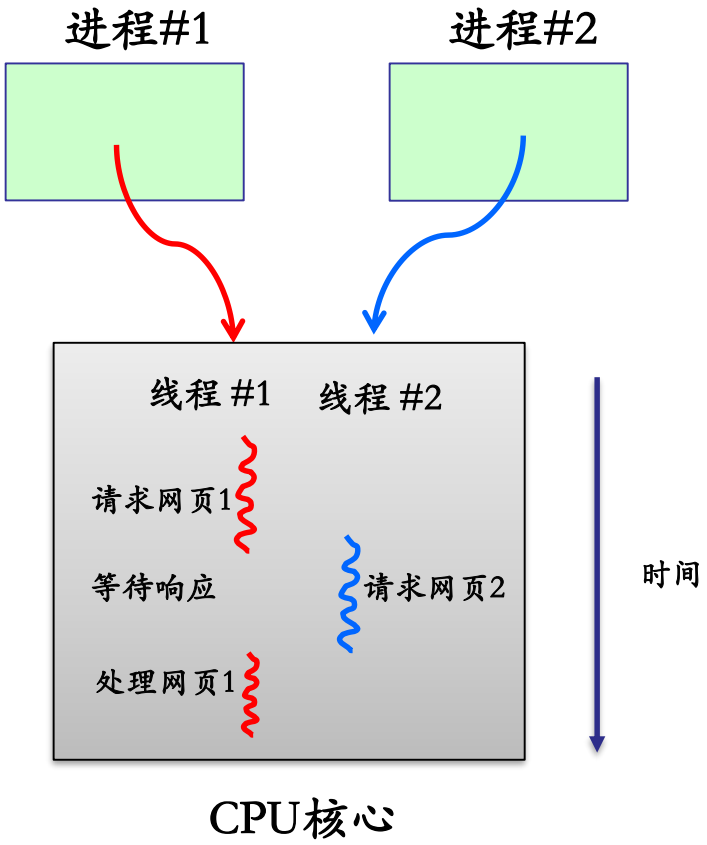
- 多线程与多进程的区别



（一个多核CPU）多线程特点：各线程在不同核心中并行执行。

4.3.1 线程和进程如何工作

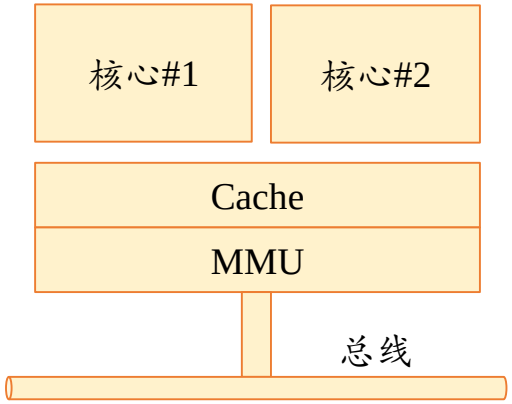
- 多线程与多进程的区别



（一个单核CPU）多进程（单线程）特点：各进程的线程在唯一的核心中并发执行。

4.3.1 线程和进程如何工作

- 多线程与多进程的区别



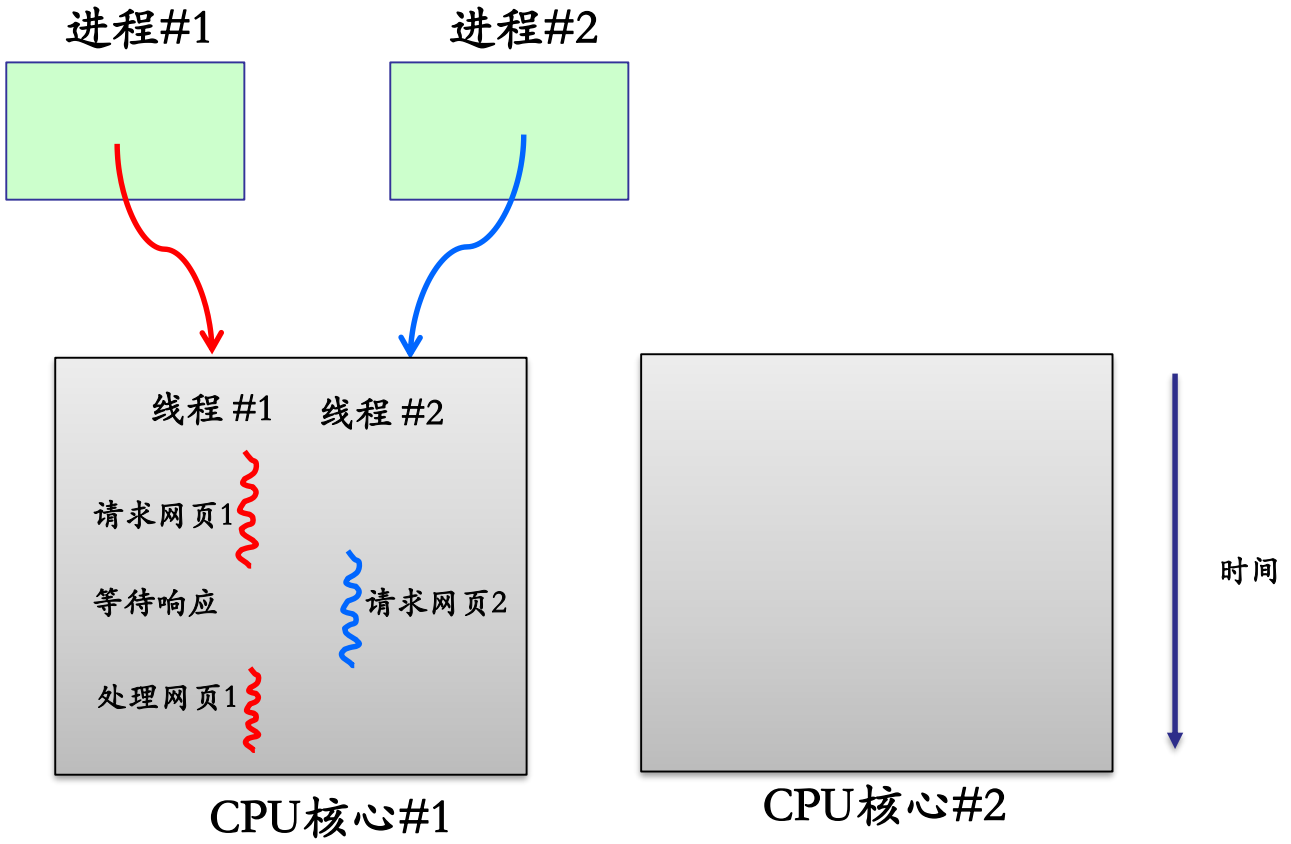
多核CPU架构

对于多核CPU的架构：

- 由于共用一套内存管理单元(Memory Management Unit，MMU)和缓存（Cache），所以地址空间是一个，同一时刻只能运行一个进程，此时进程不能并行只能并发。
- 同一个进程下的多线程可以并行执行，因为多线程共享同一套进程空间资源。

4.3.1 线程和进程如何工作

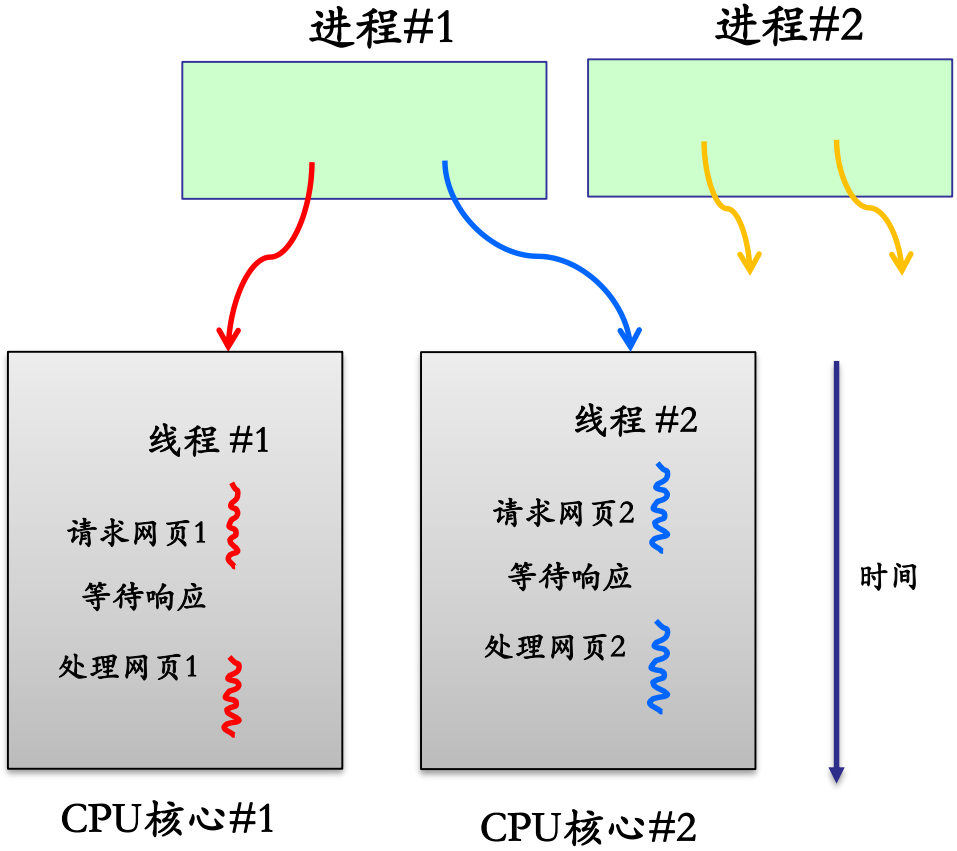
- 多线程与多进程的区别



（一个多核CPU）多进程（单线程）特点：各进程的线程仅在同一个CPU核心中并发执行。

4.3.1 线程和进程如何工作

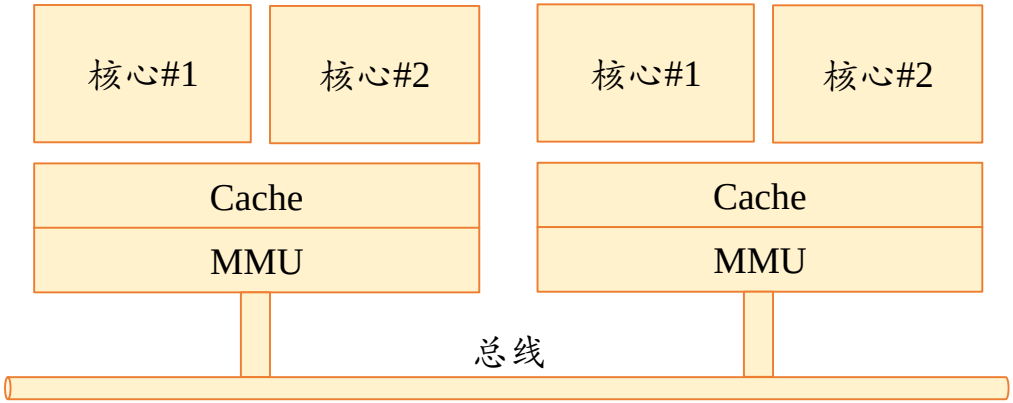
- 多线程与多进程的区别



（一个多核CPU）多进程（多线程）特点：各进程在CPU中并发执行，某个进程获得CPU执行权时，其中的多个线程在多个CPU核心中并行执行。

4.3.1 线程和进程如何工作

- 多线程与多进程的区别



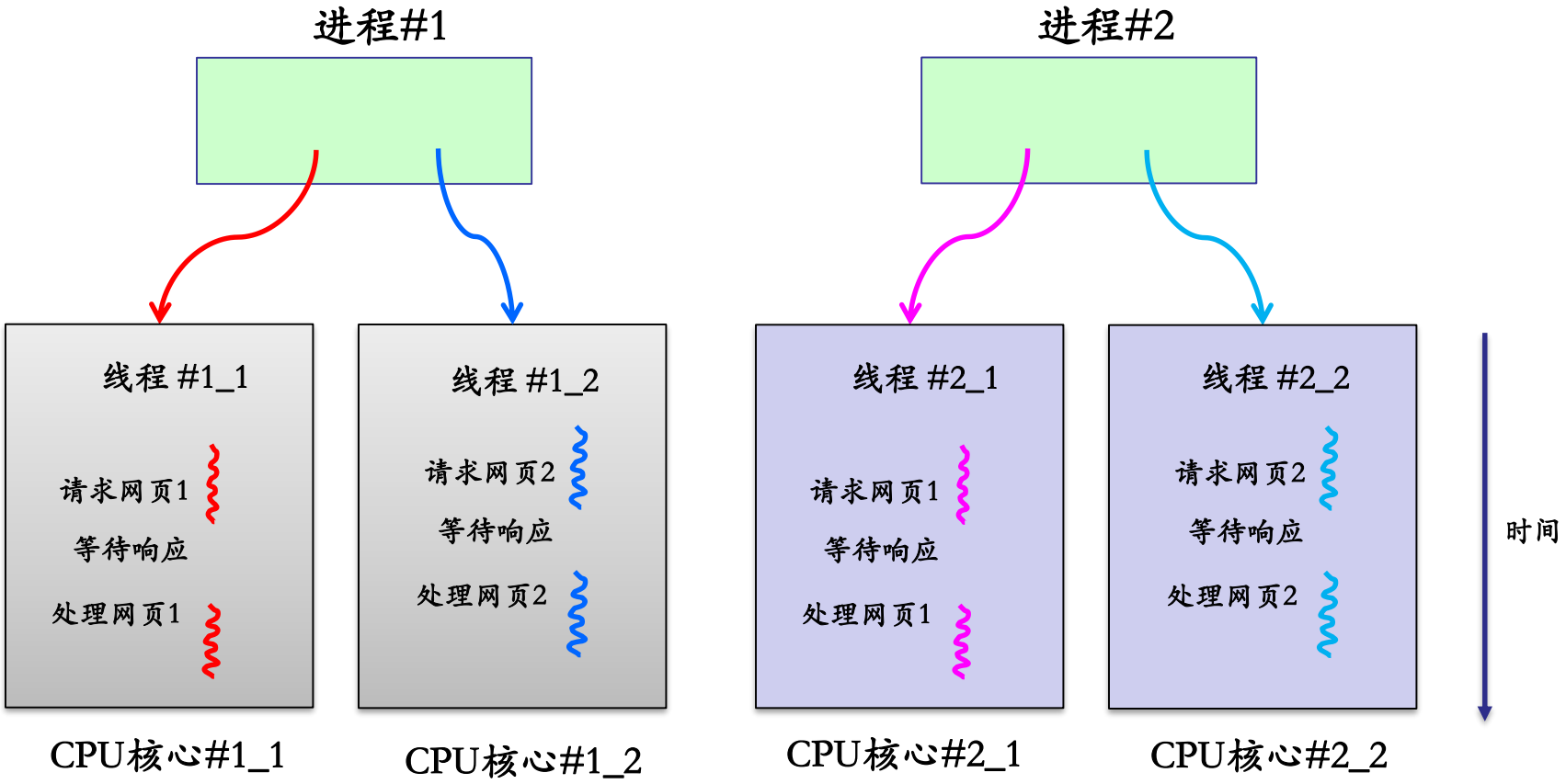
多CPU架构

对于多CPU的架构（一般用于计算工作站）：

- 每个CPU都拥有独立的MMU和Cache，因此它支持同一时刻运行多个进程，即进程并行执行；
- 同时，进程中的线程也可以并行执行。

4.3.1 线程和进程如何工作

- 多线程与多进程的区别



（多个多核CPU）多进程（多线程）特点：各进程在多个CPU中并行执行，某个进程中的多个线程在某个CPU的多个核心中并行执行。

4.3.2 threading模块

- Python实现多线程编程相对来说比较简单，需要借助于threading 模块，其中最核心的内容是Thread 类。
- 多线程编程：首先创建 Thread 对象，然后让它们运行，每个 Thread 对象代表一个线程，在每个线程中可以让程序处理不同的任务。值得注意的是，程序运行时默认就是在主线程上。
- 创建线程对象有2种手段

  1. 直接创建 Thread 对象，将一个 callable 对象从类的构造器传递进去，这个 callable 就是回调函数，用来处理任务。

  2. 编写一个继承 Thread类的子类，然后重写run() 法，在 其中编写任务处理代码，然后创建子类的实例。

（1）直接创建 Thread 对象

- Thread类的构造方法

```
Thread(group=None, target=None, name=None, args=(),
       kwargs={}, *, daemon=None)
```

  - Thread 的构造方法中，最重要的参数是target，需要将一个callable 对象赋值给它，线程才能正常运行。
  - 参数name—线程名字，默认为“Thread-N”的形式，N= 1,2,…
  - 参数args —传给callable 对象的位置参数
  - 参数kwargs —传给callable 对象的关键字参数
  - 参数daemon —决定线程是否为守护线程（后台线程），默认为非守护线程，也可以在线程创建完之后通过thread.setDaemon(True)将thread线程设置为守护线程。
  - 注意：这些参数必须以关键字参数形式传递。

- 一个线程对象创建好后，就可以调用其start() 方法来启动它。当线程获得CPU执行权后，target参数指定的回调函数将被执行。

（1）直接创建 Thread 对象

- 例：创建子线程，在其中计算并输出两个数的差

```
import threading  # 导入threading模块

def sub(x, y):  # 定义线程任务函数（回调函数）
    """计算并输出两个数的差"""
    print(f'{threading.currentThread().name}开始')
    print(f'{x} - {y} = {x - y}')
    print(f'{threading.currentThread().name}结束')
```

```
print(f'{threading.currentThread().name}开始')
thread1 = threading.Thread(target=sub, args=(5, 2))  # 创建一个子线程，给任务函数传递位置参数
thread1.start()  # 启动子线程
```

```
thread2 = threading.Thread(target=sub, kwargs={'x': 4, 'y': 10})
# 创建另一个子线程，给任务函数传递关键字参数
thread2.start()  # 启动子线程
print(f'{threading.currentThread().name}结束')
```

某次运行结果：

```
MainThread开始
Thread-1开始
5 - 2 = 3
Thread-1结束
Thread-2开始
MainThread结束
4 - 10 = -6
Thread-2结束
```

threading.current_thread() 方法当前正在执行代码所在的线程对象，线程名字同过name 属性或getName()方法获得。

（1）直接创建 Thread 对象

- 例：创建子线程，在其中计算并输出两个数的差

- 两次运行结果不尽相同，这是因为在多核单CPU架构中，本进程中的多线程是并行的，但是在每个线程在其执行的CPU核心上与其他进程的线程是并发的，因此各线程的执行步调在每次执行中都可能不同。
- 每个线程本身输出内容的顺序是确定的，但是多个线程混合在一起输出到控制台的顺序是不确定的。

另一次运行结果：

```
MainThread开始
Thread-1开始
Thread-2开始
4 - 10 = -6
MainThread结束
Thread-2结束
5 - 2 = 3
Thread-1结束
```

（1）直接创建 Thread 对象

- 例：在主线程上打印3次，输出线程名字和序号，在一个子线程上也这样打印5次

```
import threading
import time

def test(n):  # 任务函数
    for i in range(n):
        print(f'{threading.current_thread().name} 输出 {i}')
        time.sleep(1)
```

```
thread = threading.Thread(target= test, args=(5,))
thread.start()

test(3)
```

某次运行结果：

```
Thread-1 输出 0
MainThread 输出 0
MainThread 输出 1
Thread-1 输出 1
Thread-1 输出 2
MainThread 输出 2
Thread-1 输出 3
Thread-1 输出 4
```

（1）直接创建 Thread 对象

- 例：在主线程上打印3次，输出线程名字和序号，在一个子线程上也这样打印5次

```
import threading
import time

def test(n):  # 任务函数
    for i in range(n):
        print(f'{threading.current_thread().name} 输出 {i}')
        time.sleep(1)
```

```
thread = threading.Thread(target= test, args=(5,))
thread.start()

test(3)
```

另一次运行结果：

```
Thread-1 输出 0
MainThread 输出 0
Thread-1 输出 1
MainThread 输出 1
MainThread 输出 2
Thread-1 输出 2
Thread-1 输出 3
Thread-1 输出 4
```

（1）直接创建 Thread 对象

- 例：在主线程上打印 3次，在两个子线程上各打印3 次

```
import threading
import time

def test():
    for i in range(3):
        print(f'{threading.current_thread().name} 输出 {i}')
        time.sleep(1)

for i in range(2):
    thread = threading.Thread(target= test)
    thread.start()

test()
```

某次运行结果：

Thread-1 输出 0
Thread-2 输出 0
MainThread 输出 0
MainThread 输出 1
Thread-1 输出 1
Thread-2 输出 1
MainThread 输出 2
Thread-1 输出 2
Thread-2 输出 2

（1）直接创建 Thread 对象

- 例：在主线程上打印 3次，在两个子线程上各打印3 次

某次运行结果:

Thread-1 输出 0
Thread-2 输出 0
MainThread 输出 0
MainThread 输出 1
Thread-1 输出 1
Thread-2 输出 1
Thread-2 输出 2
MainThread 输出 2
Thread-1 输出 2

某次运行结果:

Thread-1 输出 0
Thread-2 输出 0
MainThread 输出 0
Thread-1 输出 1
MainThread 输出 1
Thread-2 输出 1
Thread-1 输出 2
MainThread 输出 2
Thread-2 输出 2

某次运行结果:

Thread-1 输出 0
Thread-2 输出 0
MainThread 输出 0
Thread-1 输出 1
Thread-2 输出 1
MainThread 输出 1
Thread-1 输出 2
Thread-2 输出 2
MainThread 输出 2

（2）自定义线程类

- 步骤：
  - 自定义一个 Thread 类的子类，然后重写其 run( ) 方法，在 其中编写任务处理代码。
  - 创建自定义线程类的对象。
  - 启动线程，此时run( )会被自动调用。

（2）自定义线程类

- 示例：自定义线程类，其中无限循环输出随机数字

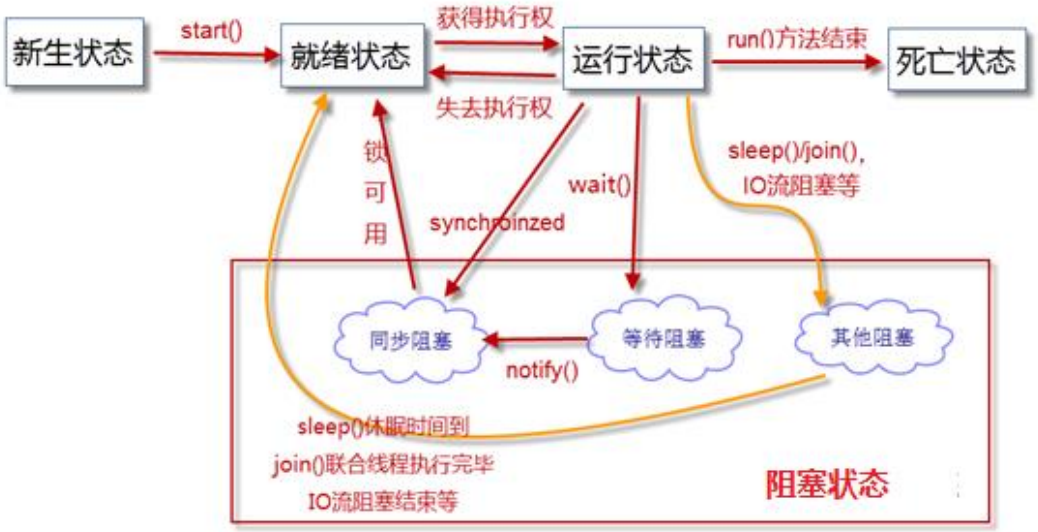
```
import threading, time, random

class MyThread(threading.Thread):  # 继承自Thread类
    def __init__(self, name=None):
        threading.Thread.__init__(self, name=name)
        # super().__init__(name=name)  #

    def run(self):   # 重写继承自父类的run()方法
        while True:
            m = int(random.random() * 10)
            print(f'{threading.current_thread().name} 输出 {m}')
            time.sleep(0.5)

print(f'{threading.current_thread().name} 开始')
for i in range(2):
    thread = MyThread()  # 创建自定义线程类对象
    thread.start()  # 启动线程，run()将会被自动调用。

print(f'{threading.current_thread().name} 结束')
```

某次运行结果：

```
MainThread 开始
Thread-1 输出 9
Thread-2 输出 3
MainThread 结束
Thread-1 输出 4
Thread-2 输出 5
Thread-1 输出 4
Thread-2 输出 3
Thread-2 输出 7
Thread-1 输出 0
.....
```

4.3.3 线程的生命周期

- **Thread 的生命周期**
  - 创建对象时，代表 Thread 内部被初始化。
  - 调用 start() 方法后，thread 进入就绪状态。
  - 获得CPU执行权限后，执行任务函数，线程进入运行状态。
  - thread 的任务函数执行完成（正常运行结束或异常终止）是，线程结束，处于消亡状态。
  - 因线程等待、休眠或同步导致阻塞状态。



线程生命周期状态图

4.3.4 线程等待

- 有时，一个线程需要等待一个或多个其他线程结束后，才能继续运行或结束。这可以通过两种方式来完成：
  - 循环等待存活线程
  - 使用join()方法阻塞线程

（1）循环等待存活线程

- 可以通过 Thread 对象的 is_alive()/isAlive()方法查询线程是否还在运行。
- 注意，只有Thread 对象被正常初始化、start() 方法被调用，且线程的代码还在正常运行时，is_alive( ) 方法才返回 True 。

4.3.4 线程等待

- 示例：让子线程比主线程早一点结束（等子线程先结束）

```
def test(n): # 线程任务函数
    for i in range(n):
        print(f'{threading.current_thread().name} 输出 {i}')
        time.sleep(1)
    print(f'{threading.current_thread().name} 结束')

threads = []  # 线程列表
for i in range(2):
    thread = threading.Thread(target=test, args=(3,))
    thread.start()
    threads.append(thread)

while True:
    some_alive = False  # 是否还有子线程在运行
    for thread in threads:
        if thread.is_alive():
            some_alive = True
            break
    if some_alive:  # 有子线程在运行，则sleep继续循环等待
        time.sleep(1)
    else:  # 否则，退出while无限循环
        break
print(f'{threading.current_thread().name} 结束')
```

某次运行结果：

```
Thread-1 输出 0
Thread-2 输出 0
Thread-2 输出 1
Thread-1 输出 1
Thread-2 输出 2
Thread-1 输出 2
Thread-2 结束
Thread-1 结束
MainThread 结束
```

另一次运行结果：

```
Thread-1 输出 0
Thread-2 输出 0
Thread-1 输出 1
Thread-2 输出 1
Thread-2 输出 2
Thread-1 输出 2
Thread-1 结束
Thread-2 结束
MainThread 结束
```

4.3.4 线程等待

（2）join( ) 方法提供了线程阻塞手段

- join( ) 方法的功能是在程序指定位置，让该方法的调用者优先使用 CPU资源。
- 在线程A中某处调用线程B的 join( ) 方法，则线程A此后将阻塞，直到线程B结束，代码才继续往下执行。
- 默认的情况是，join() 会一直等待对应线程的结束，但可以通过参数timeout赋值，只等待规定的时间（以秒为单位）。

```
thread.join(self, timeout=None)
```

4.3.4 线程等待

- 示例：阻塞主线程，等待所有子线程结束后才继续执行

```
threads = []  # 线程列表
for i in range(3):
    thread = threading.Thread(target=test, args=(5,))
    thread.start() # 启动子线程
    thread.join()  # 阻塞主线程，等待thread线程结束
    threads.append(thread)

print(f'{threading.current_thread().name} 结束')
```

- 线程1启动后，主线程阻塞，一直等到线程1结束后，才能创建线程2；
- 线程2启动后，主线程阻塞，一直等到线程2结束后，才能创建线程3；
- 线程3启动后，主线程阻塞，一直等到线程3结束后，主线程才能继续执行；

4.3.4 线程等待

- 示例：阻塞主线程，等待所有子线程结束后才继续执行

```
threads = []  # 线程列表
for i in range(3):
    thread = threading.Thread(target=test, args=(5,))
    thread.start() # 启动子线程
    thread.join()  # 阻塞主线程，等待thread线程结束
    threads.append(thread)

print(f'{threading.current_thread().name} 结束')
```

创建并启动一个子线程后，就阻塞主线程，因此同时存活的线程就只有两个，不满足要求。

运行结果：

```
Thread-1 输出 0
Thread-1 输出 1
Thread-1 输出 2
Thread-1 输出 3
Thread-1 输出 4
Thread-1 结束
Thread-2 输出 0
Thread-2 输出 1
Thread-2 输出 2
Thread-2 输出 3
Thread-2 输出 4
Thread-2 结束
Thread-3 输出 0
Thread-3 输出 1
Thread-3 输出 2
Thread-3 输出 3
Thread-3 输出 4
Thread-3 结束
MainThread 结束
```

4.3.4 线程等待

- 改进：先创建并启动所有子线程，然后阻塞主线程

```
threads = []  # 线程列表
for i in range(3):
    thread = threading.Thread(target=test, args=(5,))
# 创建子线程
    thread.start() # 启动子线程
    threads.append(thread)

# 阻塞主线程，等待thread线程结束
for thread in threads:
    thread.join()

print(f'{threading.current_thread().name} 结束')
```

某次运行结果：

```
Thread-1 输出 0
Thread-2 输出 0
Thread-3 输出 0
Thread-1 输出 1
Thread-3 输出 1
Thread-2 输出 1
Thread-3 输出 2
Thread-1 输出 2
Thread-2 输出 2
Thread-1 输出 3
Thread-3 输出 3
Thread-2 输出 3
Thread-3 输出 4
Thread-1 输出 4
Thread-2 输出 4
Thread-3 结束
Thread-1 结束
Thread-2 结束
MainThread 结束
```

4.3.5 创建守护线程

- 设置子线程为守护线程
  - 目的：MainThread 结束，子线程也立马结束。
  - 做法：在子线程调用 start() 方法之前设置 daemon属性的值为 True：

    thread.daemon = True

    或者： thread.setDaemon(True)
  - 也可以在子线程的构造器中传递 daemon 的值为 True:

    thread = threading.Thread(target=test, daemon=True)

4.3.5 创建守护线程

- 示例：子线程无限循环输出随机数字（0~9）

```
import threading
import time
import random

def test():
    while True:
        m = int(random.random()*10)
        print(f'{threading.current_thread().name} 输出 {m}')
        time.sleep(0.5)

print(f'{threading.current_thread().name} 开始')
for i in range(2):
    thread = threading.Thread(target= test)
    thread.start()

print(f'{threading.current_thread().name} 结束')
```

当主线程结束时，各子线程还在执行。

某次运行结果：

```
MainThread 开始
Thread-1 输出 8
Thread-2 输出 1
MainThread 结束
Thread-1 输出 9
Thread-2 输出 7
Thread-1 输出 5
Thread-2 输出 4
Thread-1 输出 9
Thread-2 输出 8
Thread-1 输出 9
Thread-2 输出 0
.....
```

4.3.5 创建守护线程

- 示例修改：子线程作为守护线程执行。

```
……
print(f'{threading.current_thread().name} 开始')
for i in range(2):
    thread = threading.Thread(target=test, daemon=True
    thread.start()
print(f'{threading.current_thread().name} 结束')
```

某次运行结果：
MainThread 开始
Thread-1 输出 9
Thread-2 输出 6
MainThread 结束

某次运行结果：
MainThread 开始
Thread-1 输出 0
Thread-2 输出 8
MainThread 结束

```
thread = threading.Thread(target=test, daemon=True)
```

```
thread = threading.Thread(target=test)
thread.setDaemon(True)
```

```
thread = threading.Thread(target=test)
thread.daemon = True
```

当主线程结束时，各子线程也很快就结束。

4.3.6 实现多线程爬虫

- 可以保留与第1章开发的链接爬虫类似的队列结构，只是改为在多个线程中启动爬虫循环，从而并行下载这些链接。
- 具体做法是将串行爬虫的link_crawler( )函数改为threaded_crawler( )函数，链接队列中各页面的爬取由多个线程负责实现。

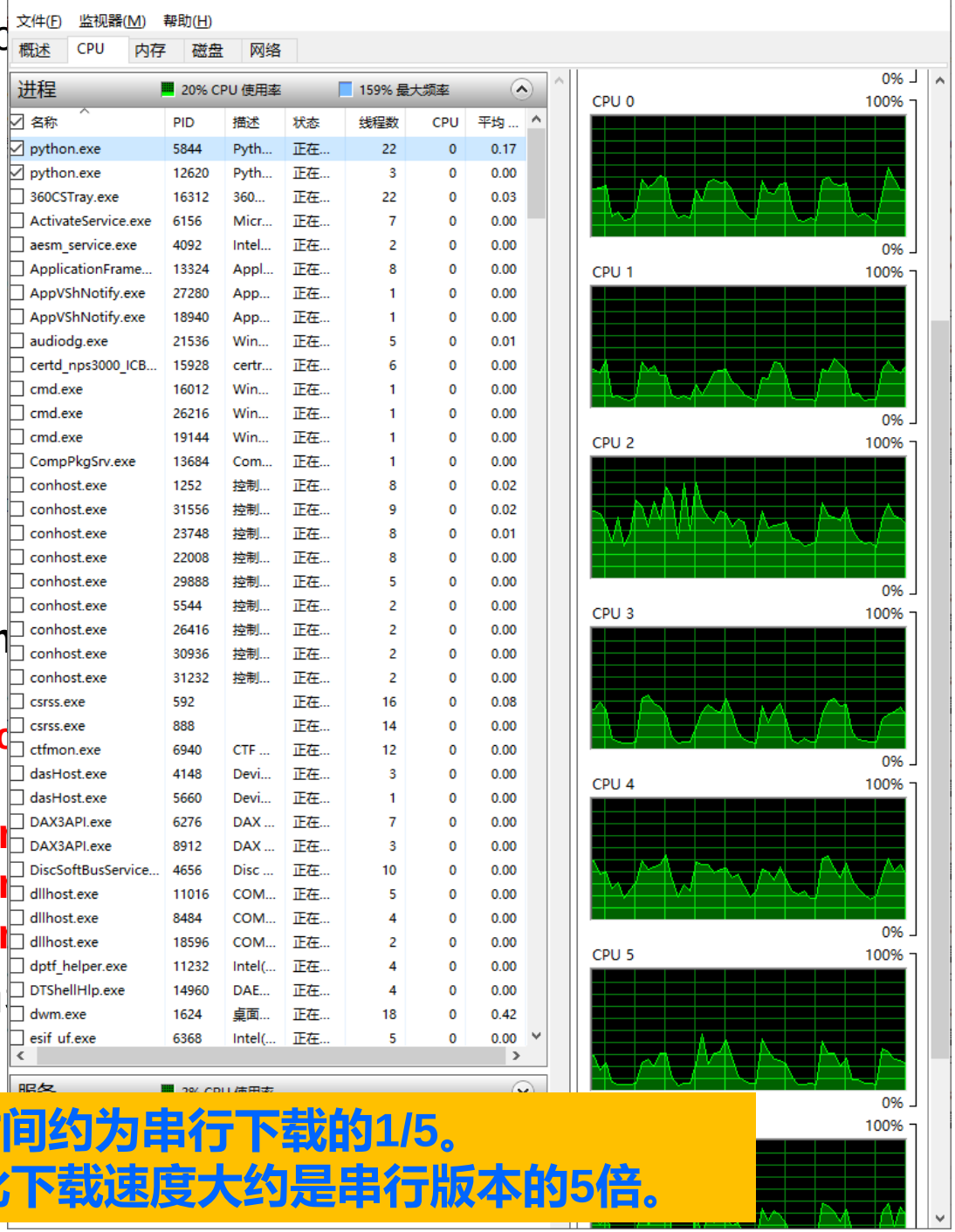
```
#---------------------------- threading_link_crawler.py----------------------------
import sys
sys.path.append(r'..\ch3')  # 注意，还需要在PyCharm中将ch3包设为源根
import socket
import requests
from downloader import Downloader
import re
import time
from urllib.parse import urljoin, urlsplit
from urllib import robotparser
from alexaCallback2 import AlexaCallback
import threading
from commonFunctions import get_links, get_robots_parse, show_time_diff

SLEEP_TIME = 1
……
def threaded_crawler(start_urls, link_regex, scrape_callback=None, delay=5,
        user_agent='wswp', proxies=None, cache={}, num_retries=2, max_threads=10):
    crawl_queue = start_urls.copy()  # 注意如果直接crawl_queue = start_urls，
                                                则两个引用的是同一列表
    seen = set(crawl_queue)   # 根据列表创建一个新集合


    rp_dict = dict()  # 该字典用来存储每个域名的解析器
    D = Downloader(delay=delay, user_agent=user_agent, proxies=proxies, cache=cache)
```

```
def threaded_crawler(…, max_threads=10):
    ......
    # ----------------各线程中执行的代码段---------------------
    def process_queue():
        while crawl_queue:
            url = crawl_queue.pop()  # 弹出队列首元素（链接）
            components = urlsplit(url)
            start_url = 'http://' + components.netloc
            if start_url not in rp_dict.keys():  # 该域名尚未建立对应的rp对象
                # robots_url = f'{start_url}/robots.txt'
                rp_dict[start_url] = None  # get_robots_parse(robots_url)
            rp = rp_dict[start_url] # 取出url所在网站的域名所对应的rp
            # 如果存在robots.txt文件且其中禁止访问该url，则跳过下载
            if (rp is not None) and (not rp.can_fetch(user_agent, url)):
                print(f'Skipping: {url}')
                continue
            html = D(url, num_retries=num_retries)  # 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)
            for link in get_links(html):
                if re.match(link_regex, link):
                    abs_link = urljoin(start_url, link)
                    if abs_link not in seen: # check if have already seen this link
                        seen.add(abs_link)
                        crawl_queue.append(abs_link)  # 绝对链接加入队列末尾
```

```
def threaded_crawler(…, max_threads=10):
    ......
    def process_queue():……

    # 根据当前链接队列和线程队列的情况，创建新线程或移除已停止的线程
    threads = []
    while len(threads) < max_threads and crawl_queue:
        # can start some more threads
        thread = threading.Thread(target=process_queue)
        print('开启一个新线程：', thread.getName())
        # set daemon so main thread can exit when receives ctrl-c
        thread.setDaemon(True)
        thread.start()
        threads.append(thread)
    # all threads have been processed
    # sleep temporarily so CPU can focus execution elsewhere
    for thread in threads:  # 等待所有线程完成
        thread.join()
        print(f'{thread.getName()}: {thread.is_alive()}')
```

```
#---------------------------- threading_link_crawler.py-----------------------------
.....
if __name__ == '__main__':
    timeout = 10
    socket.setdefaulttimeout(timeout)
    print(socket.getdefaulttimeout())

    alexa = AlexaCallback(max_urls=500)
    start_urls = alexa()
    print(start_urls)
    regex = '$^'  # 使用'$^'作为模式，避免收集每个页面的链接
    start = time.time()
    # link_crawler(start_urls, regex, scrape_callback=None, cache=RedisCache())
    threaded_crawler(start_urls, regex, max_threads=5)
    end = time.time()

    show_time_diff(end, start)
```

['http://google.com', 'http://youtube.c
开启一个新线程： Thread-1
开启一个新线程： Thread-2
……
开启一个新线程： Thread-5
Downloading: http://gome.com.cn
encoding= utf-8
Downloading: http://espn.com
Downloading: http://so.com
encoding= UTF-8
encoding= UTF-8
……
Downloading: http://youtube.com
Downloading: http://google.com
Download error: HTTPConnectionPoo
Downloading: http://360.cn
Download error: HTTPConnectio
Download error: HTTPConnectio
Download error: HTTPConnectio
Wall time: 0 hours 8 min



并行下载所用时间约为串行下载的1/5。
由于使用了5个线程，因此下载速度大约是串行版本的5倍。

4.3.7 使用线程池实现多线程

- 系统启动一个新线程的成本是比较高的，因为它涉及与操作系统的交互。在这种情形下，使用线程池可以很好地提升性能，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。

- 线程池在系统启动时即创建指定数量空闲的线程，程序只要将一个函数提交给线程池，线程池就会启动一个空闲的线程来执行它。当该函数执行结束后，该线程并不会死亡，而是再次返回到线程池中变成空闲状态，等待执行下一个函数。

4.3.7 使用线程池实现多线程

- 线程池的使用
  - 线程池的基类是concurrent包内futures模块中的Executor类，它提供了两个子类:
    - ThreadPoolExecutor类：用于创建线程池
    - ProcessPoolExecutor类：用于创建进程池
  - 如果使用线程池/进程池来管理并发编程，那么只要将相应的 task 函数提交给线程池/进程池，剩下的事情就由线程池/进程池来处理。

4.3.7 使用线程池实现多线程

- Exectuor类提供的常用方法
  - submit(fn, *args, **kwargs)：将fn函数提交给线程池。*args代表通过位置参数给fn函数传参数值，**kwargs代表通过关键字参数给fn函数传参数值。
  - map(func, *iterables, timeout=None, chunksize=1)：该函数类似于全局函数map(func, *iterables)，只是该函数将会启动多个线程，以异步方式立即对iterables执行map处理。
  - shutdown(wait=True)：关闭线程池。

程序将任务函数fn提交（submit）给线程池后，submit方法会返回一个Future对象。Future类主要用于获取线程任务函数的返回值。由于线程任务会在新线程中以异步方式执行，因此，线程执行的函数相当于一个“将来完成”的任务，所以Python使用Future来代表。

4.3.7 使用线程池实现多线程

- Exectuor类提供的常用方法
  - submit(fn, *args, **kwargs)：将fn函数提交给线程池。*args代表通过位置参数给fn函数传参数值，**kwargs代表通过关键字参数给fn函数传参数值。
  - map(func, *iterables, timeout=None, chunksize=1)：该函数类似于全局函数map(func, *iterables)，只是该函数将会启动多个线程，以异步方式立即对iterables执行map处理。
  - shutdown(wait=True)：关闭线程池。

线程池使用完毕后，应该调用该线程池的shutdown()方法，该方法将启动线程池的关闭序列。调用shutdown()方法后的线程池不再接收新任务，但会将以前所有的已提交任务执行完成。当线程池中的所有任务都执行完成后，该线程池中的所有线程都会死亡。

4.3.7 使用线程池实现多线程

- Future类提供了如下方法
  - cancel()：取消该Future代表的线程任务。如果该任务正在执行，不可取消，则该方法返回False；否则，程序会取消该任务，并返回True。
  - cancelled()：返回Future代表的线程任务是否被成功取消。
  - running()：如果该Future代表的线程任务正在执行，该方法返回True。
  - done()：如果该Funture代表的线程任务被成功取消或执行完成，则该方法返回True。

4.3.7 使用线程池实现多线程

- Future类提供了如下方法
  - result(timeout=None)：获取该Future代表的线程任务最后返回的结果。如果Future代表的线程任务还未完成，该方法将会阻塞当前线程，其中timeout参数指定最多阻塞多少秒。
  - exception(timeout=None)：获取该Future代表的线程任务所引发的异常。如果该任务成功完成，没有异常，则该方法返回None。
  - add_done_callback(fn)：为该Future代表的线程任务注册一个“回调函数”，当该任务成功完成时，程序会自动触发该fn函数。

4.3.7 使用线程池实现多线程

- 使用线程池来执行线程任务的一般步骤
  a) 调用ThreadPoolExecutor类的构造器创建线程池。
  b) 定义一个普通函数作为线程任务（task）。
  c) 调用ThreadPoolExecutor对象的submit()方法来提交线程任务。
  d) 当不想提交任何任务时，调用ThreadPoolExecutor对象的shutdown()方法来关闭线程池。

4.3.7 使用线程池实现多线程

- 链接爬虫的threaded_crawler()函数的线程池实现

```
def threaded_crawler(…, max_threads=10):
        ......
        def process_queue():……

        # 创建一个包含若干条线程的线程池
        pool = ThreadPoolExecutor(max_workers=max_threads)
        futures =[]

        # 线程任务完成后要执行的“回调函数”
        def finished(future):  # 需要传递一个参数，表示Future对象，可以从获取该线程运行结果
                print(threading.current_thread().name, "----线程完成!")
                futures.remove(future)
                print(f'len(futures) = {len(futures)}’)
```

4.3.7 使用线程池实现多线程

```
def threaded_crawler(…, max_threads=10):
    ......
    def finished(future): ……

    while len(futures) < max_threads and crawl_queue:
        # can start some more threads
        future = pool.submit(process_queue)
        future.add_done_callback(finished) # 指定线程完成后要执行的函数finished
        futures.append(future)
        print('向线程池加入一个新线程！')

    pool.shutdown(wait=True)  # 关闭线程池，该方法阻塞直到所有线程完成
    # wait – If True then shutdown will not return
    # until all running futures have finished executing
    # and the resources used by the executor have been reclaimed.
```

4.4 多进程爬虫

- 为了进一步改善性能，对多线程示例再度扩展，使其支持多进程。
- 目前，爬虫队列都是存储在本地内存当中的，其他进程都无法处理这一爬虫。
- 为了解决该问题，需要把爬虫队列转移到Redis中。单独存储队列，意味着即使是不同服务器上的爬虫也能够协同处理同一个爬虫任务。

4.4.1 multiprocessing包

- multiprocessing是一个使用类似于threading模块的API支持生成进程的包。该包提供本地和远程并发(local and remote concurrency)，从而允许程序员充分利用给定机器上的多个处理器。

4.4.1 multiprocessing包

- Process类表示进程，构造方法如下

  multiprocessing.Process(*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

  - group参数永远为None，该参数仅用于兼容threading.Thread
  - target指定一个可以调用对象，它会自动被进程类的run()方法调用
  - name是进程名，主进程默认名字为MainProcess，子进程默认名字为Process-1, Process-2，…
  - args是一个元组，是传给target所调用方法的位置参数
  - kwargs是一个字典，是传给target所调用方法的关键字参数
  - daemon默认为None，意味着从创建进程中继承，可设为True(守护进程)或False(非守护进程)

```
agrs=( 'yes', 1, 2 )
def func(a, x=0, y=0):
kwargs={'a': 'yes', 'x':1, 'y':2}
```

- 应该始终使用关键字参数调用构造方法

  例如：Multiprocessing.Process(target=func, agrs=('yes', 1, 2))

4.4.1 multiprocessing包

- Process.start( ) 方法
  - 启动进程，只能调用一次，它会在进程中调用run方法。
- Process.join([timeout])方法
  - 设主进程为m，子进程为s，m中调用s.join()：阻塞m，直到s进程结束，timeout是一个正数，它最多会阻塞timeout秒。只能在调用s.start( )后调用s.join( )。
  - 在主进程中对各子进程调用join()方法，这样主进程会阻塞，等待所有子进程结束后主进程才能结束。

4.4.1 multiprocessing包

- Process.run( )方法
  - 表示进程活动的方法。调用target指定的函数，如果有参数，会按顺序传入。自定义进程类要在子类中覆盖此方法。
  - 单独调用run不会开启子进程。
- Process.is_alive( )方法
  - 查看进程是否还活着。
  - 粗略地说，从start() 方法返回到子进程终止的那一刻，进程对象处于活动状态。
- Process.pid属性用于获取进程ID ， os.getpid( )也可以获取进程ID。
- Process.name属性用于获取进程。
- 静态方法multiprocessing.current_process( )返回当前进程。

4.4.1 multiprocessing包

- 示例：在主进程中通过multiprocessing.Process创建多个子进程，创建子进程时传入多个参数。

```
import multiprocessing
from multiprocessing import Process
import os
import time


def func(x, y):  # 子进程中要执行的任务函数
    cp: Process = multiprocessing.current_process()  # 当前进程
    z = x + y
    print(f'{cp.name}(pid={os.getpid()}，父进程pid={os.getppid()}):  {x} + {y} = {z}')
    time.sleep(60)
    print(f'{cp.name}(pid={os.getpid()}，父进程pid={os.getppid()})结束')
```

4.4.1 multiprocessing包


```
if __name__ == '__main__':
    cp: Process = multiprocessing.current_process() # 当前进程
    print(f'{cp.name}(pid={os.getpid()}，父进程pid={os.getppid()})开始')
    processes = [] # 子进程列表
    process = Process(target=func, args=(2, 5)) # 创建一个子进程
    process.start() # 启动子进程
    processes.append(process) # 创建一个子进程
    process = Process(target=func, kwargs={'x': 40, 'y': 16})
    process.start() # 启动子进程
    processes.append(process)

    for process in processes:
        print(f'子进程 {process.pid} 是否存活: {process.is_alive()}')
    print(f'{cp.name}(pid={os.getpid()}，父进程pid={os.getppid()})等待子进程结束...')
    for process in processes: # 等待所有子进程结束
        process.join()
    for process in processes:
        print(f'子进程 {process.pid} 是否存活: {process.is_alive()}')
    print(f'{cp.name}(pid={os.getpid()}，父进程pid={os.getppid()})结束')
```

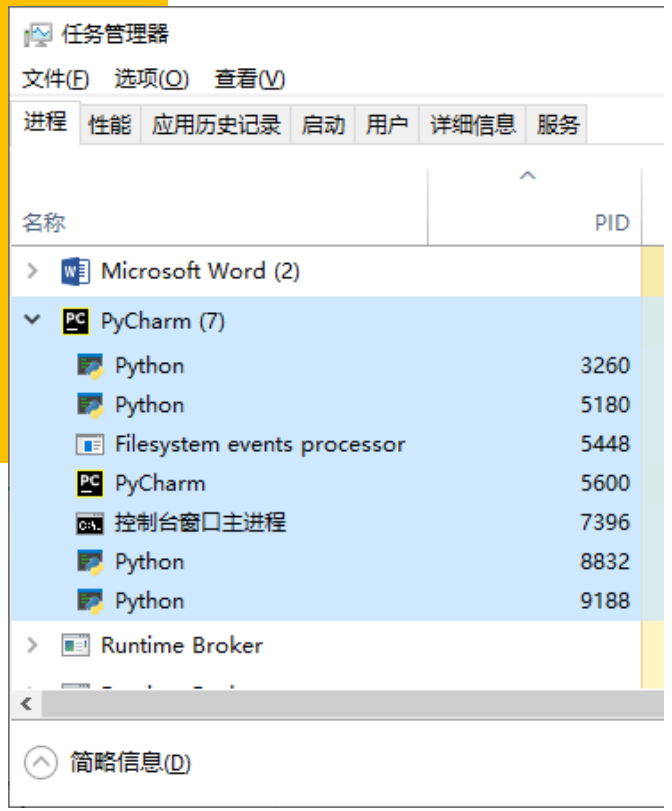
按位置参数传递给func

按关键字参数传递给func

4.4.1 multiprocessing包

某次运行结果：

```
MainProcess(pid=5180，父进程pid=5600)开始
子进程 8832是否存活: True
子进程 9188是否存活: True
MainProcess(pid=5180，父进程pid=5600)等待子进程结束...
Process-1(pid=8832，父进程pid=5180)：2 + 5 = 7
Process-2(pid=9188，父进程pid=5180)：40 + 16 = 56
Process-1(pid=8832，父进程pid=5180)结束
Process-2(pid=9188，父进程pid=5180)结束
子进程 8832是否存活: False
子进程 9188是否存活: False
MainProcess(pid=5180，父进程pid=5600)结束
```



4.4.1 multiprocessing包

- 示例：创建无限循环的守护进程，当主进程执行完毕，则子进程立即结束。

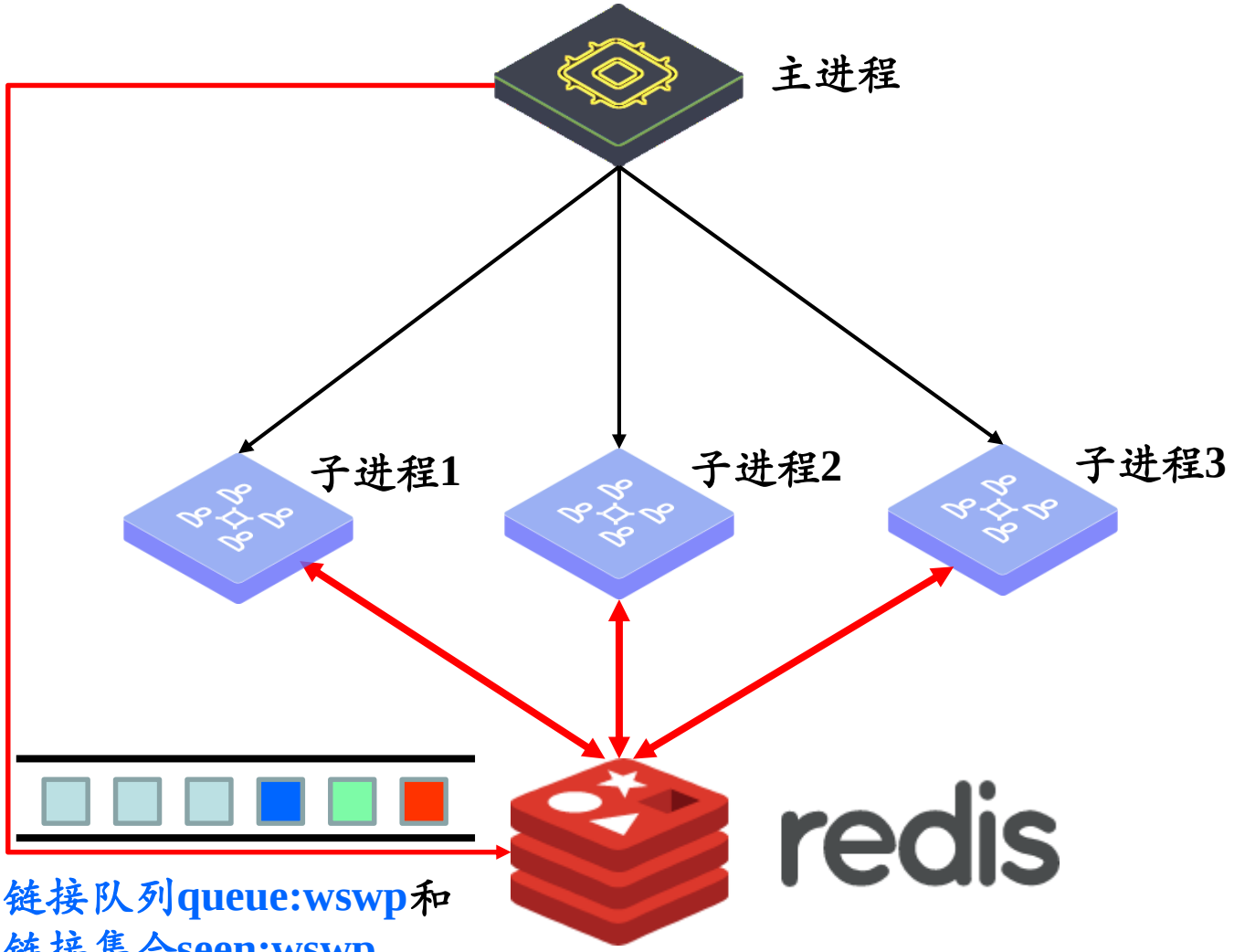
```
def func():
    cp: Process = multiprocessing.current_process()  # 当前进程
    while True:
        n = int(random.random()*10)
        print(f'{cp.name}(pid={os.getpid()}): {n}')

if __name__ == '__main__':
    cp: Process = multiprocessing.current_process()  # 当前进程
    print(f'{cp.name}(pid={os.getpid()})开始')
    for i in range(2):
        p = Process(target=func)
        p.daemon = True  # 设置为守护进程
        p.start()
    print(f'{cp.name}(pid={os.getpid()})结束')
```

```
MainProcess(pid=15224)开始
Process-1(pid=11596): 1
Process-2(pid=12188): 2
Process-1(pid=11596): 4
Process-2(pid=12188): 4
Process-2(pid=12188): 8
Process-1(pid=11596): 5
MainProcess(pid=15224)结束
```

4.4.2 实现多进程爬虫

- 主要思路
  - 爬虫链接队列存储在Redis服务器上，使各进程可以共享该队列。StrictRedis对象的lpush()和rpop()方法(对应Redis命令lpush和rpop)用于元素入列和出列（左入右出）。
  - 已爬取过的链接集合也存储在Redis服务器上，从而各进程可以共享该集合。StrictRedis对象的sadd()和sismember()方法(对应Redis命令sadd和sismember)用于添加集合成员和测试成员是否存在。
  - *主进程*负责初始化爬虫链接队列（其中应试图删除redis中之前存在的爬虫链接队列和已爬取过的链接集合），并创建各子进程。
  - *子进程*可以是单线程的，也可以是多线程的，分担爬取任务。

4.4.2 实现多进程爬虫



初始化爬虫链接队列queue:wswp和
已爬取过的链接集合seen:wswp

基于Redis实现的队列类的主要功能

```
# Based loosely on the Redis Cookbook FIFO Queue:
# http://www.rediscookbook.org/implement_a_fifo_queue.html
from redis import StrictRedis
class RedisQueue:
    # 初始化Redis连接
    def __init__(self, host='localhost', port=6379,
                 password='ruanzl', db=0, queue_name='wswp'):...
    # 关闭Redis连接
    def close(self):...
    # 重写方法，可以通过len()获得队列长度
    def __len__(self):...
    # 删除队列和集合
    def clear(self):...
    # Push an element to the tail of the queue
    def push(self, element):...
    # Pop an element from the head of the queue
    def pop(self):...
    # determine if an element has already been seen
    def already_seen(self, element):...
```

```
def __init__(self, host='localhost', port=6379, password='ruanzl',
                                                    db=0, queue_name='wswp'):
    """ 初始化Redis连接 """
    self.client = StrictRedis(host=host, port=port,
                                        password= password, db=db)
    self.name = "queue:%s" % queue_name  # Redis中的列表的名字
    self.seen_set = "seen:%s" % queue_name  # Redis中的集合的名字

def close(self):
    """关闭Redis连接 """
    if self.client is not None:
        self.client.close()

def __len__(self):
    """重写方法，可以通过len()获得队列长度"""
    return self.client.llen(self.name)

def clear(self):
    """删除队列和集合"""
    self.client.delete(*[self.name, self.seen_set])
    # self.client.delete(self.name, self.seen_set)
```

```
def push(self, elem):
    """Push an element to the tail of the queue"""
    if isinstance(elem, list):
        elem = [e.encode('UTF-8') for e in elem
                                    if not self.already_seen(e .encode('UTF-8') )]
        if len(elem):
            self.client.lpush(self.name, *elem)  # 元素加入队列
            self.client.sadd(self.seen_set, *elem)  # 元素加入集合
    else:
        elem = elem.encode('UTF-8')  # 按指定编码存储，否则按默认编码
        if not self.client.already_seen(elem):
            self.client.lpush(self.name, elem)
            self.client.sadd(self.seen_set, elem)

def pop(self):
    """Pop an element from the head of the queue"""
    # self.client.rpop(self.name) 返回的是字节数组,因此进行解码
    return self.client.rpop(self.name).decode(encoding='UTF-8')

def already_seen(self, elem):
    """ determine if an element has already been seen """
    return self.client.sismember(self.seen_set, elem)
```

子进程中运行的代码段：threaded_crawler_rq函数

```
def threaded_crawler_rq(link_regex, scrape_callback=None, delay=5,
        user_agent='wswp', proxies=None,cache={}, num_retries=2, max_threads=5,
        host='localhost', port=6379, db=0, password='ruanzl', queue_name='wswp'):
    # the queue of URL's that still need to be crawled
    crawl_RQ= RedisQueue(host=host, port=port, password=password, db=db,
                                    queue_name=queue_name)  # ① 初始化Redis连接
    rp_dict = dict()  # 该字典用来存储每个域名的解析器
    D = Downloader(delay=delay, user_agent=user_agent, proxies=proxies, cache=cache)
    print(f'我是子进程 ID = {os.getpid()}, 父进程 ID = {os.getpid()}')

    def process_queue(): #  ② 定义进程中各线程执行的代码段
        while len(crawl_RQ):  # 队列不为空
            print(f'{threading.current_thread().getName()} of 子进程(ID={os.getpid()})')
            url = crawl_RQ.pop()
            ......
            for link in get_links(html): # filter for links matching our regular expression
                if re.match(link_regex, link):
                    abs_link = urljoin(start_url, link)
                    crawl_RQ.push(abs_link)  # 绝对链接加入队列末尾
    threads = []
    while len(threads) < max_threads and len(crawl_RQ): # ③ 创建子线程，并发下载
    ......
    crawl_RQ.close()  # ④ 关闭Redis连接
    print(f'子进程(ID = {os.getpid()})结束')
```

主进程中启动子进程的代码段--mp_threaded_crawler函数

```
def mp_threaded_crawler(start_urls, **kwargs):
    """ 初始化Redis队列，启动多个进程 """
    # 若Redis中存在，则先清除它们
    RQ = RedisQueue(host=kwargs.get('host'), port=kwargs.get('port'),
                    password=kwargs.get('password'), db=kwargs.get('db'),
                    queue_name=kwargs.get('queue_name'))
    RQ.clear()
    RQ.push(start_urls)  # 在Redis中初始化队列
    RQ.close()  # 关闭redis连接

    # 进程个数由关键字参数num_procs传入，否则默认为CPU核心数
    num_procs = kwargs.pop('num_procs') # 从字典弹出一个键值对
    if not num_procs:
        num_procs = multiprocessing.cpu_count()
    processes = []  # 进程列表
    for i in range(num_procs):
        proc = multiprocessing.Process(  # 创建一个进程，执行的代码由target参数指定
            target=threaded_crawler_rq,  # target是一个可调用对象，它会被run方法调用
            kwargs=kwargs)  # kwargs是传给target所调用方法的关键字参数，字典类型
        proc.start()  # 启动子进程
        processes.append(proc)

    for proc in processes:  # wait for processes to complete
        proc.join()  # 阻塞主进程，等待所有子进程结束（join可防止产生僵尸进程）
```

测试代码段

```
if __name__ == '__main__':
    print(f'我是主进程 ID = {os.getpid()}')
    socket.setdefaulttimeout(10)
    alexa = AlexaCallback(max_urls=500)
    start_urls = alexa()
    print(start_urls)

    host = 'localhost'
    port = 6379
    password = 'ruanzl'
    db = 0
    queue_name = 'wswp'
    link_regex = '$^'  # 使用'$^'作为模式，避免收集每个页面的链接。

    kwargs = {'link_regex': link_regex, 'num_procs': 8, 'max_threads': 1,
                  'host': host, 'port': port,  'password': 'ruanzl', 'db': db,
                  'queue_name': queue_name}
     # 在多进程传参时首先要检测参数能不能序列化，例如 StrictRedis对象无法序列化。
     # 方法：在传参前检测一下是否会报错, 例如pickle.dumps(kwargs)
    start = time.time()
    mp_threaded_crawler(start_urls, **kwargs)
    end = time.time()
    show_time_diff(end, start)
```

- 某次运行结果

我是主进程 ID = 25744
['http://google.com', 'http://youtube.com',…, 'http://pixnet.net']
我是子进程 ID = 28824, 父进程 ID = 25744
我是子进程 ID = 25440, 父进程 ID = 25744
我是子进程 ID = 25156, 父进程 ID = 25744
我是子进程 ID = 16804, 父进程 ID = 25744
子进程(PID=28824)中开启一个新线程：Thread-1
Downloading: http://google.com
子进程(PID=25156)中开启一个新线程：Thread-1
子进程(PID=25440)中开启一个新线程：Thread-1
Downloading: http://youtube.com
Downloading: http://facebook.com
子进程(PID=16804)中开启一个新线程：Thread-1
Downloading: http://tmall.com
Download error: ('Connection aborted.', ConnectionAbortedError(10053, '你的主机中的软件中止了一个已建立的连接。', None, 10053, None))
code = 404
Downloading: http://baidu.com
encoding= ascii
Downloading: http://qq.com
encoding= utf-8
……

……
子进程(ID = 25440)结束
Download error: HTTPConnectionPool(host='discordapp.com', port=80): Max retries exceeded with url: / (Caused by ConnectTimeoutError (<urllib3.connection.HTTPConnection object at 0x0000027F08918D90>, 'Connection to discordapp.com timed out. (connect timeout=10)'))
code = 404
encoding= ascii
子进程(ID = 25156)结束
Download error: HTTPSConnectionPool(host='zara.com', port=443): Max retries exceeded with url: / (Caused by ConnectTimeoutError (<urllib3.connection.HTTPSConnection object at 0x0000023582BB6910>, 'Connection to zara.com timed out. (connect timeout=10)'))
code = 404
Download error: HTTPConnectionPool(host='cpic.com.cn', port=80): Max retries exceeded with url: / (Caused by ConnectTimeoutError (<urllib3.connection.HTTPConnection object at 0x00000206A87712E0>, 'Connection to cpic.com.cn timed out. (connect timeout=10)'))
code = 404
子进程(ID = 28824)结束
encoding= utf-8
子进程(ID = 16804)结束

并行下载所用时间约为串行下载的7.5/100，这里使用了4x5=20个线程，是单进程5线程的下载速度的3倍。

Wall time: 0 hours 2 mins 49.365217 secs

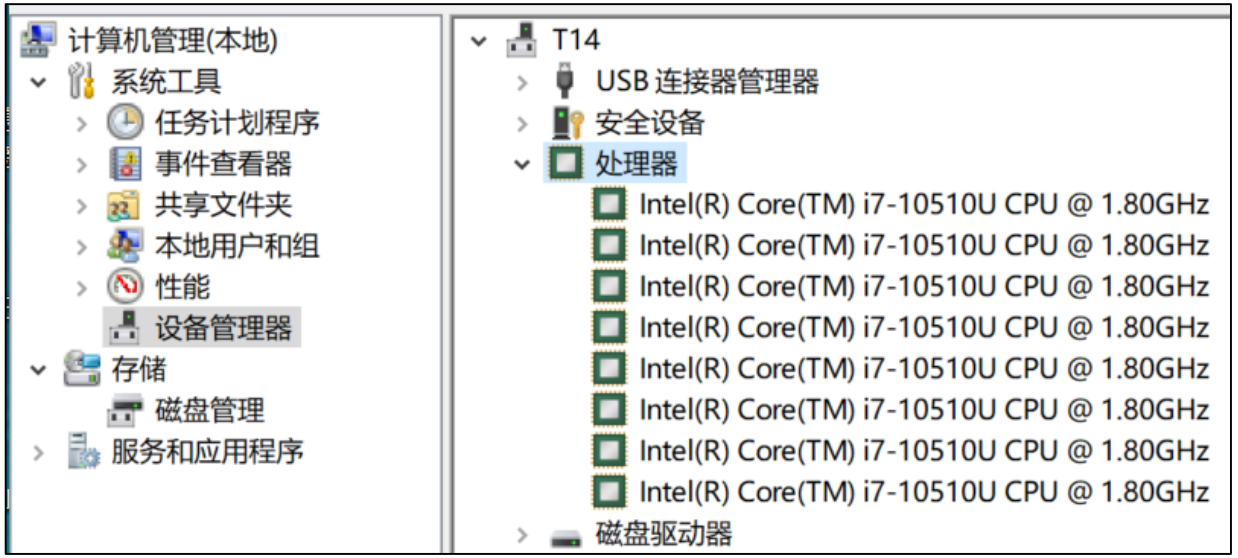
单进程5线程爬虫： **Wall time: 0 hours 8 mins 30.652355 secs**

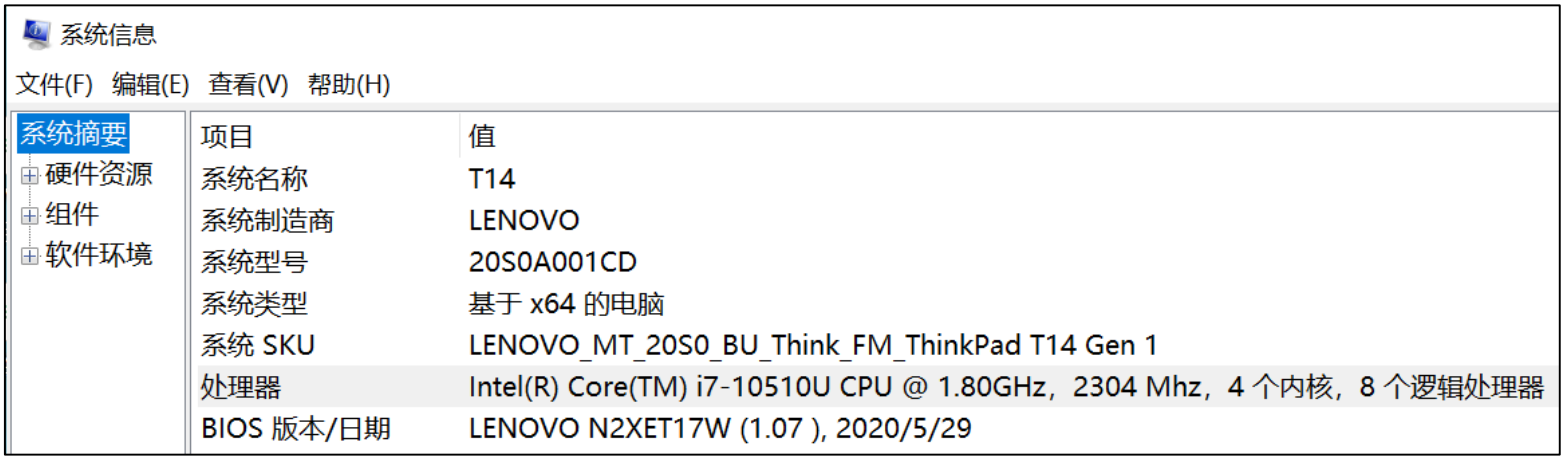
串行爬虫：**Wall time: 0 hours 39 mins 15.783550 secs**

4.5 性能对比

- 计算机CPU配置

我的笔记本电脑有8个CPU（4个物理核心、4个虚拟核心）





4.5 性能对比

- 1个主进程，4个子进程

名称	PID	21% CPU	38% 内存
Windows 资源管理器 (2)	11620	5.2%	698.0 MB
Windows 命令处理程序 (2)		0.1%	1.7 MB
命令提示符 - redis-server c:\Redis-x64-5.0.9\redis.windows...	12724	0%	0.4 MB
Redis for Windows, based on MS OpenTech port.	20220	0.1%	1.3 MB
PyCharm (11)		4.9%	1,397.0 MB
控制台窗口主进程	18944	0%	5.7 MB
控制台窗口主进程	26940	0%	5.7 MB
控制台窗口主进程	27344	0%	5.7 MB
Python	21436	0%	20.2 MB
Python	25744	0%	16.0 MB
Python	16804	0%	19.2 MB
Python	25156	0.1%	18.8 MB
Python	25440	3.7%	21.9 MB
Python	28824	0.9%	20.9 MB
PycharmProjects – threaded_crawler_with_RQ.py	27600	0.2%	1,262.5 MB
Filesystem events processor	27376	0%	0.4 MB

4.5 性能对比

- 进程数、线程数不同设置下的运行情况

{'num_procs': 4, 'max_threads': 2}: Wall time: 0 hours 5 mins 45.447522 secs

{'num_procs': 1, 'max_threads': 8}: Wall time: 0 hours 5 mins 45.128991 secs

{'num_procs': 8, 'max_threads': 1}: Wall time: 0 hours 5 mins 21.300344 secs

{'num_procs': 2, 'max_threads': 3}: Wall time: 0 hours 7 mins 10.207076 secs

{'num_procs': 4, 'max_threads': 5}: Wall time: 0 hours 2 mins 43.132503 secs

{'num_procs': 8, 'max_threads': 5}: Wall time: 0 hours 1 mins 47.151464 secs

串行爬虫：**Wall time: 0 hours 39 mins 15.783550 secs**

4.4.2 实现多进程爬虫

- 对爬取500个网页时的结果进行了对比（教材提供）

脚本	线程数	进程数	时间	相对串行的时间比	是否出现错误？
串行	1	1	1349.798s	1	否
多线程	5	1	361.504s	3.73	否
多线程	10	1	275.492s	4.9	否
多线程	20	1	298.168s	4.53	是
多进程	2	2	726.899s	1.86	否
多进程	2	4	559.93s	2.41	否
多进程	2	8	451.772s	2.99	是
多进程	5	2	383.438s	3.52	否
多进程	5	4	156.389s	8.63	是
多进程	5	8	296.610s	4.55	是

性能的增长与线程和进程的数量并不是成线性比例的，而是趋于对数，也就是说添加过多线程后反而会降低性能。比如，使用1个进程5个线程时，性能大约为串行时的4倍，使用10个线程时性能只达到了串行下载时的5倍，而使用20个线程时实际上还降低了性能。

4.4.2 实现多进程爬虫

- 对爬取500个网页时的结果进行了对比（教材提供）

脚本	线程数	进程数	时间	相对串行的时间比	是否出现错误？
串行	1	1	1349.798s	1	否
多线程	5	1	361.504s	3.73	否
多线程	10	1	275.492s	4.9	否
多线程	20	1	298.168s	4.53	是
多进程	2	2	726.899s	1.86	否
多进程	2	4	559.93s	2.41	否
多进程	2	8	451.772s	2.99	是
多进程	5	2	383.438s	3.52	否
多进程	5	4	156.389s	8.63	是
多进程	5	8	296.610s	4.55	是

根据系统的不同，性能的增加和损失可能会有所不同；不过，众所周知的是每个额外的线程都有助于加速执行，但这并不是一个线性加速的过程。这是可以预见到的现象，因为此时进程需要在更多线程之间进行切换，专门用于每一个线程的时间就会变少。

4.4.2 实现多进程爬虫

- 对爬取500个网页时的结果进行了对比（教材提供）

脚本	线程数	进程数	时间	相对串行的时间比	是否出现错误？
串行	1	1	1349.798s	1	否
多线程	5	1	361.504s	3.73	否
多线程	10	1	275.492s	4.9	否
多线程	20	1	298.168s	4.53	是
多进程	2	2	726.899s	1.86	否
多进程	2	4	559.93s	2.41	否
多进程	2	8	451.772s	2.99	是
多进程	5	2	383.438s	3.52	否
多进程	5	4	156.389s	8.63	是
多进程	5	8	296.610s	4.55	是

此外，下载的带宽是有限的，因此最终添加新线程将无法带来更快的下载速度。当你使用大量线程或进程时，可能会出现更多的错误，例如urlopen error [Errno 101] Network is unreachable。

4.5 本章小结

- 本章中，介绍了串行下载存在性能瓶颈的原因，给出了通过多线程和多进程高效下载大量网页的方法，并对比了什么时候优化或增加线程和进程可能是有用的，什么时候又是有害的。还实现了一个新的Redis队列，并且使用它实现跨机器或进程的处理。
- 下一章中，将介绍如何抓取使用JavaScript动态加载内容的网页

谢谢大家！