在上一课时我们了解了多线程的基本概念，同时我们也提到，Python 中的多线程是不能很好发挥多核优势的，如果想要发挥多核优势，最好还是使用多进程。

那么本课时我们就来了解下多进程的基本概念和用 Python 实现多进程的方法。

多进程的含义

进程（Process）是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的一个独立单位。

顾名思义，多进程就是启用多个进程同时运行。由于进程是线程的集合，而且进程是由一个或多个线程构成的，所以多进程的运行意味着有大于或等于进程数量的线程在运行。

Python 多进程的优势

通过上一课时我们知道，由于进程中 GIL 的存在，Python 中的多线程并不能很好地发挥多核优势，一个进程中的多个线程，在同一时刻只能有一个线程运行。

而对于多进程来说，每个进程都有属于自己的 GIL，所以，在多核处理器下，多进程的运行是不会受 GIL 的影响的。因此，多进程能更好地发挥多核的优势。

当然，对于爬虫这种 IO 密集型任务来说，多线程和多进程影响差别并不大。对于计算密集型任务来说，Python 的多进程相比多线程，其多核运行效率会有成倍的提升。

总的来说，Python 的多进程整体来看是比多线程更有优势的。所以，在条件允许的情况下，能用多进程就尽量用多进程。

不过值得注意的是，由于进程是系统进行资源分配和调度的一个独立单位，所以各个进程之间的数据是无法共享的，如多个进程无法共享一个全局变量，进程之间的数据共享需要有单独的机制来实现，这在后面也会讲到。

多进程的实现

在 Python 中也有内置的库来实现多进程，它就是 multiprocessing。

multiprocessing 提供了一系列的组件，如 Process（进程）、Queue（队列）、Semaphore（信号量）、Pipe（管道）、Lock（锁）、Pool（进程池）等，接下来让我们来了解下它们的使用方法。

直接使用 Process 类

在 multiprocessing 中，每一个进程都用一个 Process 类来表示。它的 API 调用如下：

```
Process([group [, target [, name [, args [, kwargs]]]]])
```

- target 表示调用对象，你可以传入方法的名字。
- args 表示被调用对象的位置参数元组，比如 target 是函数 func，他有两个参数 m，n，那么 args 就传入 [m, n] 即可。
- kwargs 表示调用对象的字典。
- name 是别名，相当于给这个进程取一个名字。
- group 分组。

我们先用一个实例来感受一下：

```
import multiprocessing

def process(index):
    print(f'Process: {index}')

if __name__ == '__main__':
    for i in range(5):
        p = multiprocessing.Process(target=process, args=(i,))
        p.start()
```

这是一个实现多进程最基础的方式：通过创建 Process 来新建一个子进程，其中 target 参数传入方法名，args 是方法的参数，是以元组的形式传入，其和被调用的方法 process 的参数是一一对应的。

> 注意：这里 args 必须要是一个元组，如果只有一个参数，那也要在元组第一个元素后面加一个逗号，如果没有逗号则和单个元素本身没有区别，无法构成元组，导致参数传递出现问题。

创建完进程之后，我们通过调用 start 方法即可启动进程了。运行结果如下：

```
Process: 0
Process: 1
Process: 2
Process: 3
Process: 4
```

可以看到，我们运行了 5 个子进程，每个进程都调用了 process 方法。process 方法的 index 参数通过 Process 的 args 传入，分别是 0~4 这 5 个序号，最后打印出来，5 个子进程运行结束。

由于进程是 Python 中最小的资源分配单元，因此这些进程和线程不同，各个进程之间的数据是不会共享的，每启动一个进程，都会独立分配资源。

另外，在当前 CPU 核数足够的情况下，这些不同的进程会分配给不同的 CPU 核来运行，实现真正的并行执行。

multiprocessing 还提供了几个比较有用的方法，如我们可以通过 cpu_count 的方法来获取当前机器 CPU 的核心数量，通过 active_children 方法获取当前还在运行的所有进程。

下面通过一个实例来看一下：

```
import multiprocessing
import time

def process(index):
    time.sleep(index)
    print(f'Process: {index}')

if __name__ == '__main__':
    for i in range(5):
        p = multiprocessing.Process(target=process, args=[i])
        p.start()
    print(f'CPU number: {multiprocessing.cpu_count()}')
    for p in multiprocessing.active_children():
        print(f'Child process name: {p.name} id: {p.pid}')
    print('Process Ended')
```

运行结果如下：

```
Process: 0
CPU number: 8
Child process name: Process-5 id: 73595
Child process name: Process-2 id: 73592
Child process name: Process-3 id: 73593
Child process name: Process-4 id: 73594
Process Ended
Process: 1
Process: 2
Process: 3
Process: 4
```

在上面的例子中我们通过 cpu_count 成功获取了 CPU 核心的数量：8 个，当然不同的机器结果可能不同。

另外我们还通过 active_children 获取到了当前正在活跃运行的进程列表。然后我们遍历了每个进程，并将它们的名称和进程号打印出来了，这里进程号直接使用 pid 属性即可获取，进程名称直接通过 name 属性即可获取。

以上我们就完成了多进程的创建和一些基本信息的获取。

继承 Process 类

在上面的例子中，我们创建进程是直接使用 Process 这个类来创建的，这是一种创建进程的方式。不过，创建进程的方式不止这一种，同样，我们也可以像线程 Thread 一样来通过继承的方式创建一个进程类，进程的基本操作我们在子类的 run 方法中实现即可。

通过一个实例来看一下：

```
from multiprocessing import Process
import time

class MyProcess(Process):
    def __init__(self, loop):
        Process.__init__(self)
        self.loop = loop
```

```
    def run(self):
        for count in range(self.loop):
            time.sleep(1)
            print(f'Pid: {self.pid} LoopCount: {count}')

if __name__ == '__main__':
    for i in range(2, 5):
        p = MyProcess(i)
        p.start()
```

我们首先声明了一个构造方法，这个方法接收一个 loop 参数，代表循环次数，并将其设置为全局变量。在 run 方法中，又使用这个 loop 变量循环了 loop 次并打印了当前的进程号和循环次数。

在调用时，我们用 range 方法得到了 2、3、4 三个数字，并把它们分别初始化了 MyProcess 进程，然后调用 start 方法将进程启动起来。

注意：这里进程的执行逻辑需要在 run 方法中实现，启动进程需要调用 start 方法，调用之后 run 方法便会执行。

运行结果如下：

```
Pid: 73667 LoopCount: 0
Pid: 73668 LoopCount: 0
Pid: 73669 LoopCount: 0
Pid: 73667 LoopCount: 1
Pid: 73668 LoopCount: 1
Pid: 73669 LoopCount: 1
Pid: 73668 LoopCount: 2
Pid: 73669 LoopCount: 2
Pid: 73669 LoopCount: 3
```

可以看到，三个进程分别打印出了 2、3、4 条结果，即进程 73667 打印了 2 次 结果，进程 73668 打印了 3 次结果，进程 73669 打印了 4 次结果。

注意，这里的进程 pid 代表进程号，不同机器、不同时刻运行结果可能不同。

通过上面的方式，我们也非常方便地实现了一个进程的定义。为了复用方便，我们可以把一些方法写在每个进程类里封装好，在使用时直接初始化一个进程类运行即可。

守护进程

在多进程中，同样存在守护进程的概念，如果一个进程被设置为守护进程，当父进程结束后，子进程会自动被终止，我们可以通过设置 daemon 属性来控制是否为守护进程。

还是原来的例子，增加了 deamon 属性的设置：

```
from multiprocessing import Process
import time

class MyProcess(Process):
    def __init__(self, loop):
        Process.__init__(self)
        self.loop = loop

    def run(self):
        for count in range(self.loop):
            time.sleep(1)
            print(f'Pid: {self.pid} LoopCount: {count}')

if __name__ == '__main__':
    for i in range(2, 5):
        p = MyProcess(i)
        p.daemon = True
        p.start()

print('Main Process ended')
```

运行结果如下：

```
Main Process ended
```

结果很简单，因为主进程没有做任何事情，直接输出一句话结束，所以在这时也直接终止了子进程的运行。

这样可以有效防止无控制地生成子进程。这样的写法可以让我们在主进程运行结束后无需额外担心子进程是否关闭，避免了独立子进程的运行。

进程等待

上面的运行效果其实不太符合我们预期：主进程运行结束时，子进程（守护进程）也都退出了，子进程什么都没来得及执行。

能不能让所有子进程都执行完了然后再结束呢？当然是可以的，只需要加入 join 方法即可，我们可以将代码改写如下：

```
processes = []
for i in range(2, 5):
    p = MyProcess(i)
    processes.append(p)
    p.daemon = True
    p.start()
for p in processes:
    p.join()
```

运行结果如下：

```
Pid: 40866 LoopCount: 0
Pid: 40867 LoopCount: 0
Pid: 40868 LoopCount: 0
Pid: 40866 LoopCount: 1
Pid: 40867 LoopCount: 1
Pid: 40868 LoopCount: 1
Pid: 40867 LoopCount: 2
Pid: 40868 LoopCount: 2
Pid: 40868 LoopCount: 3
Main Process ended
```

在调用 start 和 join 方法后，父进程就可以等待所有子进程都执行完毕后，再打印出结束的结果。

默认情况下，join 是无限期的。也就是说，如果有子进程没有运行完毕，主进程会一直等待。这种情况下，如果子进程出现问题陷入了死循环，主进程也会无限等待下去。怎么解决这个问题呢？可以给 join 方法传递一个超时参数，代表最长等待秒数。如果子进程没有在这个指定秒数之内完成，会被强制返回，主进程不再会等待。也就是说这个参数设置了主进程等待该子进程的最长时间。

例如这里我们传入 1，代表最长等待 1 秒，代码改写如下：

```
processes = []
for i in range(3, 5):
    p = MyProcess(i)
    processes.append(p)
    p.daemon = True
    p.start()
for p in processes:
    p.join(1)
```

运行结果如下：

```
Pid: 40970 LoopCount: 0
Pid: 40971 LoopCount: 0
Pid: 40970 LoopCount: 1
Pid: 40971 LoopCount: 1
Main Process ended
```

可以看到，有的子进程本来要运行 3 秒，结果运行 1 秒就被强制返回了，由于是守护进程，该子进程被终止了。

到这里，我们就了解了守护进程、进程等待和超时设置的用法。

终止进程

当然，终止进程不止有守护进程这一种做法，我们也可以通过 terminate 方法来终止某个子进程，另外我们还可以通过 is_alive 方法判断进程是否还在运行。

下面我们来看一个实例：

```
import multiprocessing
import time

def process():
    print('Starting')
    time.sleep(5)
    print('Finished')

if __name__ == '__main__':
    p = multiprocessing.Process(target=process)
    print('Before:', p, p.is_alive())

    p.start()
    print('During:', p, p.is_alive())

    p.terminate()
    print('Terminate:', p, p.is_alive())

    p.join()
    print('Joined:', p, p.is_alive())
```

在上面的例子中，我们用 Process 创建了一个进程，接着调用 start 方法启动这个进程，然后调用 terminate 方法将进程终止，最后调用 join 方法。

另外，在进程运行不同的阶段，我们还通过 is_alive 方法判断当前进程是否还在运行。

运行结果如下：

```
Before: <Process(Process-1, initial)> False
During: <Process(Process-1, started)> True
Terminate: <Process(Process-1, started)> True
Joined: <Process(Process-1, stopped[SIGTERM])> False
```

这里有一个值得注意的地方，在调用 terminate 方法之后，我们用 is_alive 方法获取进程的状态发现依然还是运行状态。在调用 join 方法之后，is_alive 方法获取进程的运行状态才变为终止状态。

所以，在调用 terminate 方法之后，记得要调用一下 join 方法，这里调用 join 方法可以为进程提供时间来更新对象状态，用来反映出最终的进程终止效果。

进程互斥锁

在上面的一些实例中，我们可能会遇到如下的运行结果：

```
Pid: 73993 LoopCount: 0
Pid: 73993 LoopCount: 1
Pid: 73994 LoopCount: 0Pid: 73994 LoopCount: 1

Pid: 73994 LoopCount: 2
Pid: 73995 LoopCount: 0
Pid: 73995 LoopCount: 1
Pid: 73995 LoopCount: 2
Pid: 73995 LoopCount: 3
Main Process ended
```

我们发现，有的输出结果没有换行。这是什么原因造成的呢？

这种情况是由多个进程并行执行导致的，两个进程同时进行了输出，结果第一个进程的换行没有来得及输出，第二个进程就输出了结果，导致最终输出没有换行。

那如何来避免这种问题？如果我们能保证，多个进程运行期间的任一时间，只能一个进程输出，其他进程等待，等刚才那个进程输出完毕之后，另一个进程再进行输出，这样就不会出现输出没有换行的现象了。

这种解决方案实际上就是实现了进程互斥，避免了多个进程同时抢占临界区（输出）资源。我们可以通过 multiprocessing 中的 Lock 来实现。Lock，即锁，在一个进程输出时，加锁，其他进程等待。等此进程执行结束后，释放锁，其他进程可以进行输出。

我们首先实现一个不加锁的实例，代码如下：

```
from multiprocessing import Process, Lock
import time

class MyProcess(Process):
```

```
    def __init__(self, loop, lock):
        Process.__init__(self)
        self.loop = loop
        self.lock = lock

    def run(self):
        for count in range(self.loop):
            time.sleep(0.1)
            # self.lock.acquire()
            print(f'Pid: {self.pid} LoopCount: {count}')
            # self.lock.release()

if __name__ == '__main__':
    lock = Lock()
    for i in range(10, 15):
        p = MyProcess(i, lock)
        p.start()
```

运行结果如下：

```
Pid: 74030 LoopCount: 0
Pid: 74031 LoopCount: 0
Pid: 74032 LoopCount: 0
Pid: 74033 LoopCount: 0
Pid: 74034 LoopCount: 0
Pid: 74030 LoopCount: 1
Pid: 74031 LoopCount: 1
Pid: 74032 LoopCount: 1Pid: 74033 LoopCount: 1

Pid: 74034 LoopCount: 1
Pid: 74030 LoopCount: 2
...
```

可以看到运行结果中有些输出已经出现了不换行的问题。

我们对其加锁，取消掉刚才代码中的两行注释，重新运行，运行结果如下：

```
Pid: 74061 LoopCount: 0
Pid: 74062 LoopCount: 0
Pid: 74063 LoopCount: 0
Pid: 74064 LoopCount: 0
Pid: 74065 LoopCount: 0
Pid: 74061 LoopCount: 1
Pid: 74062 LoopCount: 1
Pid: 74063 LoopCount: 1
Pid: 74064 LoopCount: 1
Pid: 74065 LoopCount: 1
Pid: 74061 LoopCount: 2
Pid: 74062 LoopCount: 2
Pid: 74064 LoopCount: 2
...
```

这时输出效果就正常了。

所以，在访问一些临界区资源时，使用 Lock 可以有效避免进程同时占用资源而导致的一些问题。

信号量

进程互斥锁可以使同一时刻只有一个进程能访问共享资源，如上面的例子所展示的那样，在同一时刻只能有一个进程输出结果。但有时候我们需要允许多个进程来访问共享资源，同时还需要限制能访问共享资源的进程的数量。

这种需求该如何实现呢？可以用信号量，信号量是进程同步过程中一个比较重要的角色。它可以控制临界资源的数量，实现多个进程同时访问共享资源，限制进程的并发量。

如果你学过操作系统，那么一定对这方面非常了解，如果你还不了解信号量是什么，可以先熟悉一下这个概念。

我们可以用 multiprocessing 库中的 Semaphore 来实现信号量。

那么接下来我们就用一个实例来演示一下进程之间利用 Semaphore 做到多个进程共享资源，同时又限制同时可访问的进程数量，代码如下：

```
from multiprocessing import Process, Semaphore, Lock, Queue
```

```
import time

buffer = Queue(10)
empty = Semaphore(2)
full = Semaphore(0)
lock = Lock()

class Consumer(Process):
    def run(self):
        global buffer, empty, full, lock
        while True:
            full.acquire()
            lock.acquire()
            buffer.get()
            print('Consumer pop an element')
            time.sleep(1)
            lock.release()
            empty.release()

class Producer(Process):
    def run(self):
        global buffer, empty, full, lock
        while True:
            empty.acquire()
            lock.acquire()
            buffer.put(1)
            print('Producer append an element')
            time.sleep(1)
            lock.release()
            full.release()

if __name__ == '__main__':
    p = Producer()
    c = Consumer()
    p.daemon = c.daemon = True
    p.start()
    c.start()
    p.join()
    c.join()
    print('Main Process Ended')
```

如上代码实现了经典的生产者和消费者问题。它定义了两个进程类，一个是消费者，一个是生产者。

另外，这里使用 multiprocessing 中的 Queue 定义了一个共享队列，然后定义了两个信号量 Semaphore，一个代表缓冲区空余数，一个表示缓冲区占用数。

生产者 Producer 使用 acquire 方法来占用一个缓冲区位置，缓冲区空闲区大小减 1，接下来进行加锁，对缓冲区进行操作，然后释放锁，最后让代表占用的缓冲区位置数量加 1，消费者则相反。

运行结果如下：

```
Producer append an element
Producer append an element
Consumer pop an element
Consumer pop an element
Producer append an element
Producer append an element
Consumer pop an element
Consumer pop an element
Producer append an element
Producer append an element
Consumer pop an element
Consumer pop an element
Producer append an element
Producer append an element
```

我们发现两个进程在交替运行，生产者先放入缓冲区物品，然后消费者取出，不停地进行循环。 你可以通过上面的例子来体会信号量 Semaphore 的用法，通过 Semaphore 我们很好地控制了进程对资源的并发访问数量。

队列

在上面的例子中我们使用 Queue 作为进程通信的共享队列使用。

而如果我们把上面程序中的 Queue 换成普通的 list，是完全起不到效果的，因为进程和进程之间的资源是不共享的。即使在一个进程中改变了这个 list，在另一个进程也不能获取到这个 list 的状态，所以声明全局变量对多进程是没有用处的。

那进程如何共享数据呢？可以用 Queue，即队列。当然这里的队列指的是 multiprocessing 里面的 Queue。

依然用上面的例子，我们一个进程向队列中放入随机数据，然后另一个进程取出数据。

```
from multiprocessing import Process, Semaphore, Lock, Queue
import time
from random import random

buffer = Queue(10)
empty = Semaphore(2)
full = Semaphore(0)
lock = Lock()

class Consumer(Process):
    def run(self):
        global buffer, empty, full, lock
        while True:
            full.acquire()
            lock.acquire()
            print(f'Consumer get {buffer.get()}')
            time.sleep(1)
            lock.release()
            empty.release()

class Producer(Process):
    def run(self):
        global buffer, empty, full, lock
        while True:
            empty.acquire()
            lock.acquire()
            num = random()
            print(f'Producer put {num}')
            buffer.put(num)
            time.sleep(1)
            lock.release()
            full.release()

if __name__ == '__main__':
    p = Producer()
    c = Consumer()
    p.daemon = c.daemon = True
    p.start()
    c.start()
    p.join()
    c.join()
    print('Main Process Ended')
```

运行结果如下：

```
Producer put  0.719213647437
Producer put  0.44287326683
Consumer get 0.719213647437
Consumer get 0.44287326683
Producer put  0.722859424381
Producer put  0.525321338921
Consumer get 0.722859424381
Consumer get 0.525321338921
```

在上面的例子中我们声明了两个进程，一个进程为生产者 Producer，另一个为消费者 Consumer，生产者不断向 Queue 里面添加随机数，消费者不断从队列里面取随机数。

生产者在放数据的时候调用了 Queue 的 put 方法，消费者在取的时候使用了 get 方法，这样我们就通过 Queue 实现两个进程的数据共享了。

管道

刚才我们使用 Queue 实现了进程间的数据共享，那么进程之间直接通信，如收发信息，用什么比较好呢？可以用 Pipe，管道。

管道，我们可以把它理解为两个进程之间通信的通道。管道可以是单向的，即 half-duplex：一个进程负责发消息，另一个进程负责

收消息；也可以是双向的 duplex，即互相收发消息。

默认声明 Pipe 对象是双向管道，如果要创建单向管道，可以在初始化的时候传入 deplex 参数为 False。

我们用一个实例来感受一下：

```
from multiprocessing import Process, Pipe

class Consumer(Process):
    def __init__(self, pipe):
        Process.__init__(self)
        self.pipe = pipe

    def run(self):
        self.pipe.send('Consumer Words')
        print(f'Consumer Received: {self.pipe.recv()}')

class Producer(Process):
    def __init__(self, pipe):
        Process.__init__(self)
        self.pipe = pipe

    def run(self):
        print(f'Producer Received: {self.pipe.recv()}')
        self.pipe.send('Producer Words')

if __name__ == '__main__':
    pipe = Pipe()
    p = Producer(pipe[0])
    c = Consumer(pipe[1])
    p.daemon = c.daemon = True
    p.start()
    c.start()
    p.join()
    c.join()
    print('Main Process Ended')
```

在这个例子里我们声明了一个默认为双向的管道，然后将管道的两端分别传给两个进程。两个进程互相收发。观察一下结果：

```
Producer Received: Consumer Words
Consumer Received: Producer Words
Main Process Ended
```

管道 Pipe 就像进程之间搭建的桥梁，利用它我们就可以很方便地实现进程间通信了。

进程池

在前面，我们讲了可以使用 Process 来创建进程，同时也讲了如何用 Semaphore 来控制进程的并发执行数量。

假如现在我们遇到这么一个问题，我有 10000 个任务，每个任务需要启动一个进程来执行，并且一个进程运行完毕之后要紧接着启动下一个进程，同时我还需要控制进程的并发数量，不能并发太高，不然 CPU 处理不过来（如果同时运行的进程能维持在一个最高恒定值当然利用率是最高的）。

那么我们该如何来实现这个需求呢？

用 Process 和 Semaphore 可以实现，但是实现起来比较我们可以用 Process 和 Semaphore 解决问题，但是实现起来比较烦琐。而这种需求在平时又是非常常见的。此时，我们就可以派上进程池了，即 multiprocessing 中的 Pool。

Pool 可以提供指定数量的进程，供用户调用，当有新的请求提交到 pool 中时，如果池还没有满，就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行它。

我们用一个实例来实现一下，代码如下：

```
from multiprocessing import Pool
import time


def function(index):
    print(f'Start process: {index}')
    time.sleep(3)
    print(f'End process {index}', )
```

```
if __name__ == '__main__':
    pool = Pool(processes=3)
    for i in range(4):
        pool.apply_async(function, args=(i,))

    print('Main Process started')
    pool.close()
    pool.join()
    print('Main Process ended')
```

在这个例子中我们声明了一个大小为 3 的进程池，通过 processes 参数来指定，如果不指定，那么会自动根据处理器内核来分配进程数。接着我们使用 apply_async 方法将进程添加进去，args 可以用来传递参数。

运行结果如下：

```
Main Process started
Start process: 0
Start process: 1
Start process: 2
End process 0
End process 1
End process 2
Start process: 3
End process 3
Main Process ended
```

进程池大小为 3，所以最初可以看到有 3 个进程同时执行，第4个进程在等待，在有进程运行完毕之后，第4个进程马上跟着运行，出现了如上的运行效果。

最后，我们要记得调用 close 方法来关闭进程池，使其不再接受新的任务，然后调用 join 方法让主进程等待子进程的退出，等子进程运行完毕之后，主进程接着运行并结束。

不过上面的写法多少有些烦琐，这里再介绍进程池一个更好用的 map 方法，可以将上述写法简化很多。

map 方法是怎么用的呢？第一个参数就是要启动的进程对应的执行方法，第 2 个参数是一个可迭代对象，其中的每个元素会被传递给这个执行方法。

举个例子：现在我们有一个 list，里面包含了很多 URL，另外我们也定义了一个方法用来抓取每个 URL 内容并解析，那么我们可以直接在 map 的第一个参数传入方法名，第 2 个参数传入 URL 数组。

我们用一个实例来感受一下：

```
from multiprocessing import Pool
import urllib.request
import urllib.error


def scrape(url):
    try:
        urllib.request.urlopen(url)
        print(f'URL {url} Scraped')
    except (urllib.error.HTTPError, urllib.error.URLError):
        print(f'URL {url} not Scraped')


if __name__ == '__main__':
    pool = Pool(processes=3)
    urls = [
        'https://www.baidu.com',
        'http://www.meituan.com/',
        'http://blog.csdn.net/',
        'http://xxxyxxx.net'
    ]
    pool.map(scrape, urls)
    pool.close()
```

这个例子中我们先定义了一个 scrape 方法，它接收一个参数 url，这里就是请求了一下这个链接，然后输出爬取成功的信息，如果发生错误，则会输出爬取失败的信息。

首先我们要初始化一个 Pool，指定进程数为 3。然后我们声明一个 urls 列表，接着我们调用了 map 方法，第 1 个参数就是进程对应的执行方法，第 2 个参数就是 urls 列表，map 方法会依次将 urls 的每个元素作为 scrape 的参数传递并启动一个新的进程，加到进程

池中执行。

运行结果如下：

```
URL https://www.baidu.com Scraped
URL http://xxxyxxx.net not Scraped
URL http://blog.csdn.net/ Scraped
URL http://www.meituan.com/ Scraped
```

这样，我们就可以实现 3 个进程并行运行。不同的进程相互独立地输出了对应的爬取结果。

可以看到，我们利用 Pool 的 map 方法非常方便地实现了多进程的执行。后面我们也会在实战案例中结合进程池来实现数据的爬取。

以上便是 Python 中多进程的基本用法，本节内容比较多，后面的实战案例也会用到这些内容，需要好好掌握。