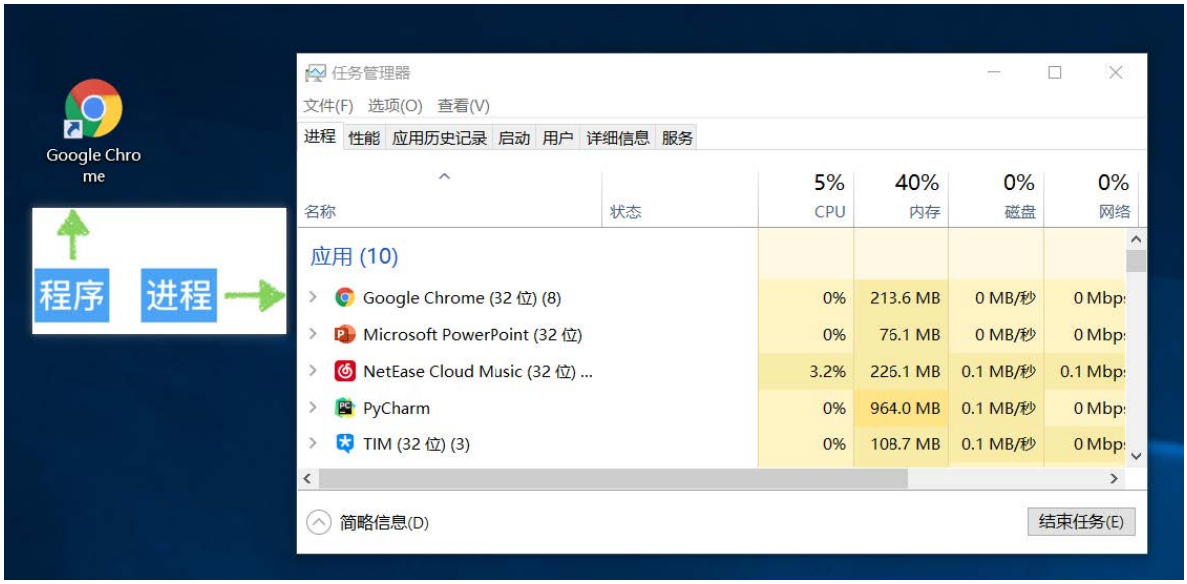
> 上节课我们讲了多线程，今天我们学习多进程

1、多进程的概念

在Python中，想要实现多任务可以使用**多进程**来完成。

进程的概念

进程（Process）是资源分配的最小单位，**它是操作系统进行资源分配和调度运行的基本单位**，通俗理解：一个正在运行的程序就是一个进程 。例如：正在运行的qq，微信等 他们都是一个进程 。



一个程序运行后至少有一个进程。如果对于一个任务想让很多人同时去做，可以用多进程的方式实现。多进程对应的python模块是**multiprocessing**。

2、multiprocessing模块的使用

`multiprocessing` 包是Python中的多进程管理包。与threading.Thread类似，它可以使用multiprocessing.Proces 对象来创建一个进程。

该进程可以运行在Python程序内部编写的函数。该Process对象与Thread对象的用法相同，也start(),run()的方法。

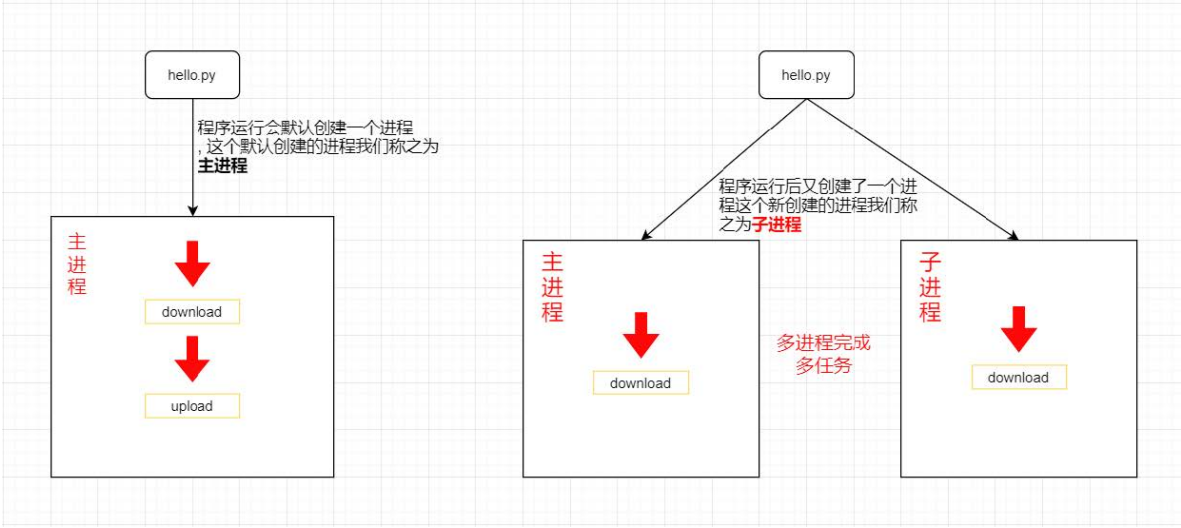
此外multiprocessing包中也有Lock/Event/Semaphore/Condition类 (这些对象可以像多线程那样，通过参数传递给各个进程)，用以同步进程，其用法与threading包中的同名类一致。

所以，multiprocessing的很大一部份与threading使用同一套API，只不过换到了多进程的情境。接下来我们通过一个案例学习：

```
import time
import multiprocessing


def download():
    print("开始下载文件...")
    time.sleep(1)
    print("完成下载文件...")
def upload():
```

```
    print("开始上传文件...")
    time.sleep(1)
    print("完成上传文件...")


# download()
# upload()
# 多进程与多线程的使用方式是差不多的
download_process = multiprocessing.Process(target=download)
upload_process = multiprocessing.Process(target=upload)
if __name__ == '__main__':
    # 多进程必须要在 if __name__ == "__main__" 里面
    download_process.start()
    upload_process.start()
    # 默认情况下，主进程代码运行完毕之后会等待子进程结束
    print('--主进程运行完了---')
```

上述代码是一个非常简单的程序，一旦运行这个程序，按照代码的执行顺序，download 函数执行完毕后才能执行upload 函数 . 如果可以让 download 和 upload 同时运行，显然执行这个程序的效率会大大提升 。



要点:

- 进程（Process）是资源分配的最小单位
- 多进程是Python程序中实现多任务的一种方式，使用多进程可以大大提高程序的执行效率

2、1 进程的创建

- 导入进程包
  - **import multiprocessing**
- 通过进程类创建进程对象
  - **进程对象 = multiprocessing.Process()**
- 启动进程执行任务
  进程对象.start()

通过进程类创建进程对象

进程对象 = multiprocessing.Process(target=任务名)

参数名	说明
target	执行的目标任务名,这里指的是函数名(方法名)
name	进程名，一般不用设置
group	进程组，目前只能使用None

进程创建与启动的代码：

```
# 创建子进程
coding_process = multiprocessing.Process(target=coding)
# 创建子进程
music_process = multiprocessing.Process(target=music)
# 启动进程
coding_process.start()
music_process.start()
```

2、2 进程的参数传递

带有参数的任务

参数名	说明
args	以元组的方式给执行任务传参
kwargs	以字典方式给执行任务传参

参数的使用

```
import multiprocessing


# 多进程多进程传参
def run_process(*args, **kwargs):
    print(args)
    print(kwargs)


if __name__ == '__main__':
    # target： 进程执行的函数名
    # args： 表示以元组的方式给函数传参
    process1 = multiprocessing.Process(target=run_process, args=(3,))
    process1.start()
```

kwargs参数的使用

```
# kwargs: 表示以字典的方式给函数传参
dance_process = multiprocessing.Process(target=run_process, kwargs={"num": 3})
dance_process.start()
```

进程执行带有参数的任务传参有两种方式：

- **元组方式传参** ： 元组方式传参一定要和参数的**顺序保持一致。**
- **字典方式传参**：字典方式传参字典中的**key**一定要和**参数名保持一致。**

2、3 多进程的运行顺序

主进程会等待所有的子进程执行结束再结束

```
import multiprocessing
import time


def work():
    for i in range(10):
        print('工作中...')
        time.sleep(0.2)
if __name__ == '__main__':
    # 创建子进程
    work_process = multiprocessing.Process(target=work)
    work_process.start()
    # 让主进程等待1秒钟
    time.sleep(1)
    print("主进程执行完成了啦")
    # 总结： 主进程会等待所有的子进程执行完成以后程序再退出
```

设置守护主进程

为了保证子进程能够正常的运行，主进程会等所有的子进程执行完成以后再销毁，设置守护主进程的目的是**主进程退出子进程销毁**，不让主进程再等待子进程去执行。

设置守护主进程方式： **子进程对象.daemon = True**

销毁子进程方式： **子进程对象.terminate()**

```
if __name__ == '__main__':
    # 创建子进程
    work_process = multiprocessing.Process(target=work)
    # 设置守护主进程，主进程退出后子进程直接销毁，不再执行子进程中的代码
    work_process.daemon = True
    work_process.start()
    # 让主进程等待1秒钟
    time.sleep(1)
    print("主进程执行完成了啦")
    # 总结： 主进程会等待所有的子进程执行完成以后程序再退出
```

销毁子进程

```
if __name__ == '__main__':
    # 创建子进程
    work_process = multiprocessing.Process(target=work)
    # # 设置守护主进程，主进程退出后子进程直接销毁，不再执行子进程中的代码
    # work_process.daemon = True
    work_process.start()
    # 让主进程等待1秒钟
    time.sleep(1)
    # 让子进程直接销毁，表示终止执行， 主进程退出之前，把所有的子进程直接销毁就可以了
    work_process.terminate()
    print("主进程执行完成了啦")
    # 总结： 主进程会等待所有的子进程执行完成以后程序再退出
```

提示:以上两种方式都能保证主进程退出子进程销毁

2、4 多进程嵌套多线程

多进程与多线程是可以嵌套使用的，需要注意的是不管是多线程还是多进程都只能操作单独的函数对象。可以参考如下案例：

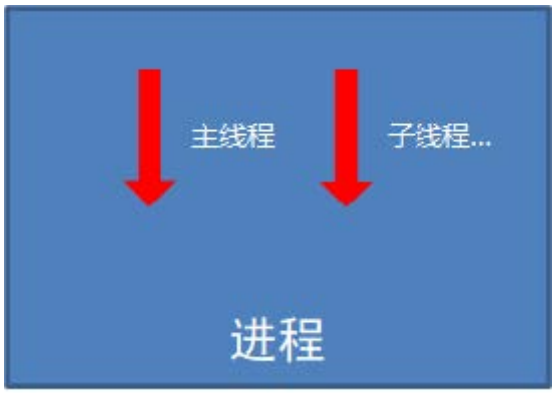
```
import requests
import threading
import time
import concurrent.futures


def send_request(url):
    """发送请求的方法"""
    json_data = requests.get(url=url)
    return json_data


def parse_data(data):
    """解析数据的方法，传入数据进行解析"""
    data_list = data['data']['object_list']
    img_url_list = []
    for data in data_list:
        img_url = data['album']['covers'][0]
        img_url_list.append(img_url)
    return img_url_list


def save_data(filename, data):
    """
    保存数据的方法
    :param filename: 文件名
    :param data: 数据
    :return: None
    """
    with open('img\\' + filename, mode='wb') as f:
        f.write(data)
        print('下载完成', filename)


def main(url):
    """主函数，根据一个url地址爬取表情包图片"""
    json_data = send_request(url).json()
    imgUrl_list = parse_data(json_data)
    for imgUrl in imgUrl_list:
        file_name = imgUrl.split('/')[-1]
        img_data = send_request(imgUrl).content
        save_data(file_name, img_data)


def many_thread(url):
    with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
        executor.submit(main, url)

if __name__ == '__main__':
    start_time = time.time()
    with concurrent.futures.ProcessPoolExecutor(max_workers=10) as executor:
        for page in range(0, 217, 24):
```

```
        url = 'https://www.duitang.com/napi/blog/list/by_search/?
kw=%E8%9C%A1%E7%AC%94%E5%B0%8F%E6%96%B0&type=feed&include_fields=top_comments%2Cis_root%2Csource_link%2
Citem%2Cbuyable%2Croot_id%2Cstatus%2Clike_count%2Clike_id%2Csender%2Calbum%2Creply_count%2Cfavorite_blo
g_id&_type=&start={}&_=1604864250098'.format(page)
        executor.submit(many_thread, url)
    print('花费时间: ', time.time() - start_time)
```

3、进程与线程的对比

关系对比

- 线程是依附在进程里面的，没有进程就没有线程。
- 一个进程默认提供一条线程，进程可以创建多个线程。



区别对比

- 进程之间不共享全局变量
- 线程之间共享全局变量，但是要注意资源竞争的问题，解决办法: 互斥锁或者线程同步
- 创建进程的资源开销要比创建线程的资源开销要大
- 进程是操作系统资源分配的基本单位，线程是CPU调度的基本单位
- 线程不能够独立执行，必须依存在进程中
- 多进程开发比单进程多线程开发稳定性要强

优缺点对比

进程优缺点:

- 优点：可以用多核
- 缺点：资源开销大

线程优缺点:

- 优点：资源开销小
- 缺点：不能使用多核

要点总结

1. 进程和线程都是完成多任务的一种方式
2. 多进程要比多线程消耗的资源多，但是多进程开发比单进程多线程开发稳定性要强，某个进程挂掉
   不会影响其它进程。
3. 多进程可以使用cpu的多核运行，多线程可以共享全局变量。

4. 线程不能单独执行必须依附在进程里面

@拓展

GIL

在非 python 环境中，单核情况下，同时只能有一个任务执行。多核时可以支持多个线程同时执行。但是在python中，无论有多少核，同时只能执行一个线程。究其原因，这就是由于GIL的存在导致的。

GIL的全称是 Global Interpreter Lock (全局解释器锁)，来源是python设计之初的考虑，为了数据安全所做的决定。某个线程想要执行，必须先拿到GIL，我们可以把 GIL 看作是“通行证”，并且在一个python进程中，GIL只有一个。拿不到通行证的线程，就不允许进入CPU执行。GIL只在cpython中才有，因为cpython调用的是c语言的原生线程，所以他不能直接操作cpu，只能利用GIL保证同一时间只能有一个线程拿到数据。而在pypy和jpython中是没有GIL的。

Python多线程的工作过程：
python在使用多线程的时候，调用的是c语言的原生线程。

1. 拿到公共数据
2. 申请 gil
3. python 解释器调用 os 原生线程
4. os 操作 cpu 执行运算
5. 当该线程执行时间到后，无论运算是否已经执行完，gil 都被要求释放
6. 进而由其他进程重复上面的过程
7. 等其他进程执行完后，又会切换到之前的线程（从他记录的上下文继续执行）
   整个过程是每个线程执行自己的运算，当执行时间到就进行切换（context switch）。

- python针对不同类型的代码执行效率也是不同的：

  1、CPU密集型代码(各种循环处理、计算等等)，在这种情况下，由于计算工作多，ticks计数很快就会达到阈值，然后触发GIL的释放与再竞争（多个线程来回切换当然是需要消耗资源的），所以python下的多线程对CPU密集型代码并不友好。
  2、IO密集型代码(文件处理、网络爬虫等涉及文件读写的操作)，多线程能够有效提升效率(单线程下有IO操作会进行IO等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序执行效率)。所以python的多线程对IO密集型代码比较友好。

- 使用建议?

  python下想要充分利用多核CPU，就用多进程。因为每个进程有各自独立的GIL，互不干扰，这样就可以真正意义上的并行执行，在python中，多进程的执行效率优于多线程(仅仅针对多核CPU而言)。

- GIL在python中的版本差异：

  1、在python2.x里，GIL的释放逻辑是当前线程遇见 `IO操作` 或者 `ticks计数达到100` 时进行释放。（ticks可以看作是python自身的一个计数器，专门做用于GIL，每次释放后归零，这个计数可以通过sys.setcheckinterval 来调整）。而每次释放GIL锁，线程进行锁竞争、切换线程，会消耗资源。并且由于GIL锁存在，python里一个进程永远只能同时执行一个线程(拿到GIL的线程才能执行)，这就是为什么在多核CPU上，python的多线程效率并不高。
  2、在python3.x中，GIL不使用ticks计数，改为使用计时器（执行时间达到阈值后，当前线程释放GIL），这样对CPU密集型程序更加友好，但依然没有解决GIL导致的同一时间只能执行一个线程的问题，所以效率依然不尽如人意。

案例：单线程、多线程、多进程效果对比

IO密集型对比

```
# -*- coding: utf-8 -*-
import concurrent.futures
import time
import random

urls = [
      f'https://maoyan.com/board/4?offset={page}' for page in range(1000)
]


def download(url):
      # print(url)
      # 延时从操作
      time.sleep(0.0000001)


if __name__ == '__main__':
      """单线程"""
      start_time = time.time()
      for url in urls:
            download(url)
      print("单线程执行：" + str(time.time() - start_time), "秒")

      """多线程"""
      start_time_1 = time.time()
      with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
            for url in urls:
                  executor.submit(download, url)
      print("线程池计算的时间：" + str(time.time() - start_time_1), "秒")

      """多进程"""
      start_time_1 = time.time()
      with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
            for url in urls:
                  executor.submit(download, url)
      print("线程池计算的时间：" + str(time.time() - start_time_1), "秒")
```

运行这个代码，我们可以看到运行时间的输出：

```
单线程执行：1.064488172531128 秒
线程池计算的时间：0.4077413082122803 秒
线程池计算的时间：0.46396422386169434 秒
```

CPU密集型

```
# -*- coding: utf-8 -*-
import concurrent.futures
import time

number_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


def evaluate_item(x):
      """计算总和，这里只是为了消耗时间"""
```

```
        a = 0
        for i in range(0, 100):
                # 重复计算 消耗时间 cpu计算能力
                a = a + i
                time.sleep(0.00000001)
        return x


if __name__ == '__main__':
        """单线程"""
        start_time = time.time()
        for item in number_list:
                evaluate_item(item)
        print("单线程执行：" + str(time.time() - start_time), "秒")

        """多线程"""
        start_time_1 = time.time()
        with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
                for item in number_list:
                        executor.submit(evaluate_item, item)
        print("线程池计算的时间：" + str(time.time() - start_time_1), "秒")

        """多进程"""
        start_time_2 = time.time()
        with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
                for item in number_list:
                        executor.submit(evaluate_item, item)
        print("进程池计算的时间：" + str(time.time() - start_time_2), "秒")
```