二、线程

2.1 什么是线程

在传统操作系统中，每个进程有一个地址空间，而且默认就有一个控制线程；多线程（即多个控制线程）的概念是，在一个进程中存在多个控制线程，多个控制线程共享该进程的地址空间。

进程只是用来把资源集中到一起（进程只是一个资源单位，或者说资源集合），而**线程**才是CPU上的执行单位。

2.2 为何要用多线程

多线程指的是在一个进程中开启多个线程，简单的讲：如果多个任务共用一块地址空间，那么必须在一个进程内开启多个线程。详细分为4点：

1. 多线程共享一个进程的地址空间；
2. 线程比进程更轻量级，线程比进程更容易创建可撤消。在许多操作系统中，创建一个线程比创建一个进程要快10-100倍，在有大量线程需要动态和快速修改时，这一特性很有用。
3. 对于计算/CPU密集型应用，多线程并不能提升性能，但对于I/O密集型应用，使用多线程会明显地提升速度（I/O密集型根本用不上多核优势）。
4. 在多核CPU系统中，为了最大限度的利用多核，可以开启多个线程（比开进程开销要小的多）。

2.3 多线程的应用举例

开启一个文本处理软件进程，该进程需要办不止一件事情，比如监听键盘输入，处理文字，定时自动将文字保存到硬盘，这三个任务操作的都是同一块数据，因而不能用多进程。只能在一个进程里并发地开启三个线程，如果是单线程，那就只能是：键盘输入时，不能处理文字和自动保存，自动保存时，又不能输入和处理文字。

2.4 线程与进程的区别

1. 线程共享创建它的进程的地址空间，进程有自己的地址空间。
2. 线程可以直接访问其进程的数据段，进程具有自己的父进程的数据段副本（也就是说子进程会将父进程的数据完全拷贝一份）。
3. 线程可以直接与其进程的其他线程进行通信，进程必须使用进程间通信来与兄弟进程进行通信。
4. 新线程创建起来更容易，而新进程创建时需要父进程的复本。
5. 线程可以对相同进程的线程进行相当的控制，而进程只能控制其子进程。
6. 对主线程的更改（取消，优先级更改等）可能会影响进程的其他线程的行为；而对父进程的更改不会影响子进程。

三、python并发编程之多进程

3.1 multiprocessing模块介绍

python中的多线程无法利用多核优势，如果想要充分地使用多核CPU的资源（os.cpu_count()查看），在python中大部分情况需要使用多进程。Python提供了非常好用的多进程包multiprocessing。
multiprocessing模块用来开启子进程，并在子进程中执行我们定制的任务（比如函数），该模块与多线程模块threading的编程接口类似。

multiprocessing模块的功能众多：支持子进程、通信和共享数据、执行不同形式的同步，提供了`Process`、`Queue`、`Pipe`、`Lock`等组件。

需要再次强调的一点是：**与线程不同，进程没有任何共享状态，进程修改的数据，改动仅限于该进程内。**

3.2 Process类的介绍

Process是创建进程的类，由该类实例化得到的对象，表示一个子进程中的任务（尚未启动）。

语法：
`Process(group=None, target=None, name=None, args=(), kwargs={})`

> **强调：**
>
> 1. 需要使用关键字的方式来指定参数；
> 2. args指定的是一个元组形式的参数，元组里的元素是传给target函数的位置参数。

参数介绍：

- `group` 参数未使用，留作未来扩展用，值始终为None
- `target` 参数指定子进程要执行的任务，是一个可调用对象，一般为函数。
- `args` 表示调用对象的位置参数元组，如args=(1,2,"egon",)
- `kwargs` 表示调用对象的字典，kwargs={"name":"egon","age":18}
- `name` 为子进程的名称

方法介绍：

- `p.daemon`：默认值为False，如果设为True，代表p为后台运行的守护进程，当p的父进程终止时，p也随之终止，并且设定为True后，p不能创建自己的新进程；必须在p.start()之前设置。
- `p.name`：进程的名称
- `p.pid`：进程的pid
- `p.exitcode`：进程在运行时为None，如果为-N，表示被信号N结束。（了解即可）
- `p.authkey`：进程的身份验证键，默认是由`os.urandom()`随机生成的32字符的字符串。这个键的用途是为涉及网络连接的底层进程间通信提供安全性，这类连接只有在具有相同的身份验证键时才能成功。（了解即可）

3.3 Process类的使用

> **注意：**在windows中，`Process()`必须放到`if __name__ == __main__:`下；由于Windows没有fork，多处理模块启动一个新的Python进程并导入调用模块。
> 如果在导入时调用Process（），那么这将启动无限继承的新进程（或直到机器耗尽资源）。
> 这是隐藏对Process（）内部调用的原，使用if **name** == “**main** ”，这个if语句中的语句将不会在导入时被调用。

创建并开启子进程的两种方式：

方式一

```
from multiprocessing import Process
import time
import random

def sing(name):
    print("%s is singing."%name)
```

```
    time.sleep(random.randint(1,3))
    print("%s is sing end."%name)

if __name__ == '__main__':
    p1 = Process(target=sing,args=("egon",)) #元组里必须加,逗号
    p2 = Process(target=sing,args=("alex",))
    p3 = Process(target=sing,args=("lisi",))
    p1.start()
    p2.start()
    p3.start()
    print("主进程")
```

输出：

```
主进程
egon is singing.
alex is singing.
lisi is singing.
lisi is sing end.
egon is sing end.
alex is sing end.
```

方式二

创建一个类，然后继承`Process`类，但是必须定义一个`run()`方法，因为子进程开启时调用的`start()`，实际上就是在调用`run()`方法。

```
from multiprocessing import Process
import time
import random

class sing(Process):
    def __init__(self,name):
        super(sing, self).__init__()
        self.name = name

    def run(self):
        print("%s is singing." % self.name)
        time.sleep(random.randrange(1,3))
        print("%s is sing end." % self.name)

if __name__ == '__main__':
    p1 = sing("egon")
    p2 = sing("alex")
    p3 = sing("lisi")

    p1.start()
    p2.start()
    p3.start()
    print("主进程")
```

输出：

```
主进程
egon is singing.
alex is singing.
lisi is singing.
alex is sing end.
lisi is sing end.
egon is sing end.
```

练习1：把上周所学的socket通信变成并发的形式
服务端代码：

```
from multiprocessing import Process
from socket import *

server = socket(AF_INET,SOCK_STREAM)
server.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
server.bind(("127.0.0.1",60000))
server.listen(5)

def communicate(conn,addr):
    '''通信循环'''
    while True:
        try:
            data = conn.recv(1024)
            if not data:break
            conn.send(data.upper())
        except Exception:
            break

if __name__ == '__main__':

    #连接循环
    print("Waiting for connections ...")
    while True:
        conn,addr = server.accept()
        print("Client address: ",addr)
        p = Process(target=communicate,args=(conn,addr,))
        p.start()
    conn.close()
server.close()
```

创建多个客户端：

```
from socket import *

client = socket(AF_INET,SOCK_STREAM)
client.connect(("127.0.0.1",60000))

#通信循环
while True:
    inp = input(">>: ").strip()
    if not inp:continue
    client.send(inp.encode("utf-8"))
    data =  client.recv(1024)
    print(data.decode("utf-8"))
```

> **注**：这么写服务端是有问题的，每来一个客户端连接，服务端就会开启一个进程，如果来一万个客户端，很可能会把你的机器资源耗尽，卡死。
> 解决方法：进程池（Pool）

Process对象的其他方法和属性

- `p.terminate()` 关闭进程，不会立即关闭，因为cpu会有延迟。
- `p.is_alive()` 判断进程是否存活，`True` 或 `False`
- `p.daemon=True` 默认为False，设置p为守护进程，禁止创建子进程，并且父进程死了，p跟着一起死。注意：一定要在p.start()前设置。
- `p.join(x)` 主进程等待p运行完毕，等待x秒就不再等了。

```
from multiprocessing import Process
import time
import random

def sing(name):
    print("%s is singing."%name)
    time.sleep(random.randint(1,3))
    print("%s is sing end."%name)

if __name__ == '__main__':
    p1 = Process(target=sing,args=("egon",)) #元组里必须加,逗号
    p2 = Process(target=sing,args=("alex",))
    p3 = Process(target=sing,args=("lisi",))

    p2.daemon = True  #设置p2为守护进程
    p1.start()
    p2.start()
    p1.terminate()    #关闭p1进程
    print("p1 is alive: ", p1.is_alive())  #判断p1是否还存在着
    p3.start()
    print(p3.pid) #查看pid

    p3.join()
    print("主进程")


'''
输出：
p1 is alive:  True
```

```
alex is singing.
lisi is singing.
egon is singing.
alex is sing end.
lisi is sing end.
egon is sing end.
主进程
'''
```

上述代码可以通过for循环简写：

```
from multiprocessing import Process
import time
import random

def sing(name):
    print("%s is singing."%name)
    time.sleep(random.randint(1,3))
    print("%s is sing end."%name)

if __name__ == '__main__':
    name_l = ['alex','egon','lisi','zhangsan','mazi']
    p_l = []
    #产生5个进程
    for name in name_l:
        p = Process(target=sing,args=(name,))
        p.start()
        p_l.append(p)

    for p in p_l:
        p.join()
    print("主进程")
```

3.4 进程间通信（IPC）

3.4.1 方式一：Queue（队列）

进程彼此之间是互相隔离的，要实现进程间通信（Interprocess Communication，即IPC），`multiprocessing`模块支持两种形式：`Queue`（队列）和管道，这两种 方式都是使用消息传递的。这里主要讲队列的实现方法。队列就是通过管道加锁实现的。

创建队列的类`Queue`

```
Queue([maxsize]):创建共享的进程队列，Queue是多进程安全的队列，可以使用Queue实现多进程之间的数据传递。

参数：
maxsize : 表示队列中允许放入最大的数据个数，省略则无大小限制。
```

方法介绍：

主要方法：

- `q.put`方法用以插入数据到队列中，put方法还有两个可选参数：`blocked`和`timeout`。如果blocked为True（默认值），并且timeout为正值，该方法会阻塞timeout指定的时间，直到该队列有剩余的空间。如果超时，会抛出Queue.Full异常。如果blocked为False，但该Queue已满，会立即抛出Queue.Full异常。

```
from multiprocessing import Process,Queue
q = Queue(3)

q.put(1)
q.put(2)
q.put(3)
q.put(4,timeout=3)  #等待3秒，如果队列中的数据已经超出个数限制，且3秒内队列中的数据没有被取走，则会报"queue.Full"异常
q.put(5,block=False)  #如果超出队列中的数据个数限制，则会立即报“queue.Full”异常
```

- `q.get`方法可以从队列读取并且删除一个元素。同样，get方法有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，那么在等待时间内没有取到任何元素，会抛出Queue.Empty异常。如果blocked为False，有两种情况存在，如果Queue有一个值可用，则立即返回该值，否则，如果队列为空，则立即抛出Queue.Empty异常.
- `q.get_nowait()`:同`q.get(False)`
- `q.put_nowait()`:同`q.put(False)`
- `q.empty()`:调用此方法时q为空则返回`True`，该结果不可靠，比如在返回True的过程中，如果队列中又加入了项目。
- `q.full()`：调用此方法时q已满则返回`True`，该结果不可靠，比如在返回`True`的过程中，如果队列中的项目被取走。
- `q.qsize()`:返回队列中目前项目的正确数量，结果也不可靠，理由同`q.empty()`和`q.full()`一样

其他方法（了解）：

- `q.cancel_join_thread()`:不会在进程退出时自动连接后台线程。可以防止`join_thread()`方法阻塞
- `q.close()`:关闭队列，防止队列中加入更多数据。调用此方法，后台线程将继续写入那些已经入队列但尚未写入的数据，但将在此方法完成时马上关闭。如果q被垃圾收集，将调用此方法。关闭队列不会在队列使用者中产生任何类型的数据结束信号或异常。例如，如果某个使用者正在被阻塞在`get()`操作上，关闭生产者中的队列不会导致`get()`方法返回错误。
- `q.join_thread()`：连接队列的后台线程。此方法用于在调用`q.close()`方法之后，等待所有队列项被消耗。默认情况下，此方法由不是q的原始创建者的所有进程调用。调用`q.cancel_join_thread`方法可以禁止这种行为。

应用：

```
from multiprocessing import Process,Queue
q = Queue(3)

q.put(1)
q.put(2)
q.put(3)
# q.put(4)
print(q.full())  #满了
```

```
print(q.get())
print(q.get())
print(q.get())
print(q.empty())  #空了


'''
输出:
True
1
2
3
True
'''
```

3.4.2 生产者消费者模型

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

为什么要使用生产者和消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

基于队列实现生产者消费者模型

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

def consumer(q,name):
    while True:
        time.sleep(random.randint(1,3))
        res = q.get()
        print("\033[41m消费者：%s,拿到了包子：%s"%(name,res))

def producer(seq,q):
    for i in seq:
        time.sleep(random.randint(1,3))
        print("\033[42m生产者生产了：%s"%i)
        q.put(i)

if __name__ == '__main__':
    q = Queue()
    consumer_l = [ "alex_%s"%i for i in range(5)]
    baozi_l = ["包子%s"%i for i in range(5)]
    for i in consumer_l:
```

```
        c = Process(target=consumer,args=(q,i,))
        c.start()

    producer(baozi_l,q)
```

主线程等待消费者结束（生产者发送结束信号给消费者）

```
from multiprocessing import Process,Queue
import time,random,os


def consumer(q):
    while True:
        time.sleep(random.randint(1,3))
        res=q.get()
        if res is None:break    #消费者获取到空后，就跳出
        print('\033[45m消费者拿到了：%s\033[0m' %res)

def producer(seq,q):
    for item in seq:
        time.sleep(random.randint(1,3))
        print('\033[46m生产者生产了：%s\033[0m' %item)

        q.put(item)

if __name__ == '__main__':
    q=Queue()

    c=Process(target=consumer,args=(q,))
    c.start()

    producer(('包子%s' %i for i in range(10)),q)
    q.put(None)    #向队列中上传一个空
    c.join()
    print('主线程')
```

3.4.3 创建队列的另外一个类（JoinableQueue）

`JoinableQueue([maxsize])` 就像是一个Queue对象，但队列允许项目的使用者通知生成者，项目已经被成功处理。通知进程是使用共享的信号和条件变量来实现的。

参数：
`maxsize`：是队列中允许的数据个数，省略则无个数限制

方法介绍：

`JoinableQueue` 的实例 `q` 除了与 `Queue` 对象相同的方法之外还具有：

- `q.task_done()`：使用者使用此方法发出信号，表示 `q.get()` 的返回项目已经被处理。如果调用此方法的次数大于从队列中删除项目的数量，将引发 `ValueError` 异常。
- `q.join()`:生产者调用此方法进行阻塞，直到队列中所有的项目均被处理。阻塞将持续到队列中的每个项目均调用 `q.task_done()` 法为止。

```
```

```python
from multiprocessing import Process,JoinableQueue
import time
import random

def consumer(q,name):
    while True:
        # time.sleep(random.randint(1,3))
        res=q.get()
        q.task_done()
        print('\033[41m消费者%s拿到了%s\033[0m' %(name,res))

def producer(seq,q,name):
    for item in seq:
        # time.sleep(random.randint(1,3))
        q.put(item)
        print('\033[42m生产者%s生产了%s\033[0m' %(name,item))
    q.join()
    print('============>>')

if __name__ == '__main__':
    q=JoinableQueue()
    c=Process(target=consumer,args=(q,'egon'),)
    c.daemon=True #设置守护进程，主进程结束c就结束,但是不用担心，producer内调用q.join保证了consumer已经处理完队列中的所有元素.
    c.start()

    seq=['包子%s' %i for i in range(10)]
    p=Process(target=producer,args=(seq,q,'厨师1'))
    p.start()

    # master--->producer----->q--->consumer(10次task_done)
    p.join() #主进程等待p结束，p等待c把数据都取完，c一旦取完数据,p.join就是不再阻塞,进
    # 而主进程结束,主进程结束会回收守护进程c,而且c此时也没有存在的必要了
    print('主进程')
```

3.4.5 方式二：管道（了解部分）

查看此博客第4.5小节

3.4.6 进程间通信方式三：共享数据

> **注：不推荐使用，了解即可**

> 展望未来，基于消息传递的并发编程是大势所趋

> 即便是使用线程，推荐做法也是将程序设计为大量独立的线程集合

> 通过消息队列交换数据。这样极大地减少了对使用锁定和其他同步手段的需求，

> 还可以扩展到分布式系统中

> **进程间通信应该尽量避免使用本节所讲的共享数据的方式**

> 进程间数据是独立的，可以借助于队列或管道实现通信，二者都是基于消息传递的

> 虽然进程间数据独立，但可以通过`multiprocessing`模块中的`Manager`类实现数据共享，事实上`Manager`的功能远不止于此。

```
from multiprocessing import Process,Manager
import os

def work(l,d,name):
    l.append(os.getpid())
    d[name] = os.getpid()

if __name__ == '__main__':
    m = Manager()
    l = m.list(['init',])   #创建一个共享列表
    d = m.dict({"name":"egon"})  #创建一个共享字典

    p_l = []
    for i in range(5):
        p = Process(target=work,args=(l,d,"进程_%s"%i))
        p.start()
        p_l.append(p)

    for p in p_l:
        p.join()   #必须要有join，否则会报错

    print(l)
    print(d)
```

输出结果：

```
['init', 8060, 10364, 9764, 9040, 10436]
{'进程_1': 8060, 'name': 'egon', '进程_3': 9040, '进程_4': 10436, '进程_2': 9764,
'进程_0': 10364}
```

> 从上述输出结果，可以看出，通过共享数据，不同的进程可以修改同一个数据。

3.4.7 进程之间通过文件共享数据，加锁实现

进程之间通过文件共享数据，需要自己加锁处理，否则会出现多个进程同时抢占文件资源的情况。加锁的目的就是为了保证多个进程修改同一块数据时，同一时间只能有一个修改，即串行的修改。这样速度肯定是慢了，但是牺牲了速度，却可以保证数据的安全。

> 进程之间数据隔离，但是共享一套文件系统，因而可以通过文件来实现进程之间的通信，但问题是必须自己加锁处理。

给文件加锁需要从`multiprocessing`模块中导入`Lock`类。

下面是模拟一个抢票的过程，把文件当作数据库。

```
#文件a.txt的内容为：{"count":1}
#注意一定要用双引号，不然json无法识别
from multiprocessing import Process,Lock
import json,time,random

def work(dbfile,lock,name):
    lock.acquire()  #加锁
    with open(dbfile,encoding='utf-8') as f:
        dic = json.loads(f.read())
```

```
    if dic['count'] > 0:
        dic['count'] -= 1
        time.sleep(random.randint(1,3))  #模拟网络延迟
        with open(dbfile,'w',encoding='utf-8') as f:
            f.write(json.dumps(dic))
        print("\033[32m%s 抢票成功\033[0m"%name)
    else:
        print("\033[31m%s 抢票失败\033[0m"%name)
    lock.release()   #释放锁

if __name__ == '__main__':
    lock = Lock()  #生成一个锁对象
    p_l = []
    for i in range(50):
        p = Process(target=work,args=("a.txt",lock,"用户_%s"%i))
        p.start()
        p_l.append(p)

    for p in p_l:
        p.join()
    print("主进程")
```

> **注意**：以上`lock.acquire()`和`lock.release()`可以修改为利用`with`语句进行上下文管理，修改如下：

修改后的代码：

```
#模拟抢票操作

from multiprocessing import Process,Lock
import json
import time,random

def work(dbfile,name,lock):
    # lock.acquire()
    with lock:
        with open(dbfile,encoding='utf-8') as f:
            dic = json.loads(f.read())

        if dic['count'] > 0:
            dic["count"] -= 1
            time.sleep(random.randint(1,3))   #模拟网络延迟
            with open(dbfile,'w',encoding='utf-8') as f:
                f.write(json.dumps(dic))
            print("\033[32m%s 抢票成功\033[0m"%name)
        else:
            print("\033[31m%s 抢票失败\033[0m" % name)
    # lock.release()

if __name__ == '__main__':
    p_l = []
    lock = Lock()    #生成锁对象
    for i in range(50):
        p = Process(target=work,args=("a.txt","用户_%s"%i,lock))
        p.start()
```

```
        p_l.append(p)

    for p in p_l:
        p.join()
    print("主进程")
```

3.5 进程池

3.5.1 进程池介绍

开启多进程的目的是为了并发，如果CPU有多核，通常有几核，就开几个进程，进程开启过多，效率反而会下降，因为开启进程是需要占用系统资源的，而且开启多于CPU核数的进程也无法做到并行。但是很明显，需要并发执行的任务要远大于核数，这时我们就可以通过维护一个**进程池**来控制进程数目，比如httpd的进程模式，规定最小进程数和最大进程数。

当被操作对象数目不大时，可以直接利用multiprocessing中的Process动态成生多个进程，十几个还好，但如果是上百个，上千个目标，手动的去限制进程数量却又太过繁琐，此时可以发挥**进程池**的功效。

而且对于远程过程调用的高级应用程序而言，应该使用进程池，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，那么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，就重用进程池中的进程。

在利用Python进行系统管理的时候，特别是同时操作多个文件目录，或者远程控制多台主机，并行操作可以节约大量的时间。

3.5.2 创建进程池的类 `Pool`

需要从`multiprocessing`模块中导入`Pool`

创建进程池：
`pool=Pool(processes=None, initializer=None, initargs=())`

参数介绍：

- `processes`：要创建的进程数，如果省略，将默认使用`cpu_count()`的值，即CPU核心数。
- `initializer`：是每个工作进程启动时要执行的可调用对象，默认为`None`；
- `initargs`：要传给`initializer`的参数组。

方法介绍：

主要方法：

- `p.apply(func,args=(), kwds={})`：在一个池工作进程中执行`func(*args,**kwargs)`,然后返回结果。同步执行，必须是一个进程执行完毕，接着再执行下一个进程。
- `p.apply_async(func,args=(),kwds={})`：在一个池工作进程中执行`func(*args,**kwargs)`,然后返回结果。异步执行，将进程丢入进程池，不会等待该进程执行完毕，再将其他进程丢入进程池。
- `p.close()`：关闭进程池，防止进一步操作。如果所有操作持续挂起，它们将在工作进程终止前完成。
- `p.join()`：等待所有工作进程退出。此方法只能在`close()`或`terminate()`之后调用。

其他方法（了解即可）：

- 方法`apply_async()`和`map_async()`的返回值是AsyncResul的实例obj。实例具有以下方法
- `obj.get()`:返回结果，如果有必要则等待结果到达。`timeout`是可选的。如果在指定时间内还没有到达，将引发一场。如果远程操作中引发了异常，它将在调用此方法时再次被引发。

- `obj.ready()`:如果调用完成，返回`True`
- `obj.successful()`:如果调用完成且没有引发异常，返回`True`，如果在结果就绪之前调用此方法，引发异常
- `obj.wait([timeout])`:等待结果变为可用。
- `obj.terminate()`：立即终止所有工作进程，同时不执行任何清理或结束任何挂起工作。如果p被垃圾回收，将自动调用此函数。

应用:
提交任务，并在主进程中拿到结果（之前的Process是执行任务，结果放到队列里，现在可以在主进程中直接拿到结果）

```
from multiprocessing import Pool
import time
def work(n):
    print('开工啦...')
    time.sleep(3)
    return n**2

if __name__ == '__main__':
    q=Pool()

    #异步apply_async用法：如果使用异步提交的任务，主进程需要使用jion，等待进程池内任务都处理完，然后可以用get收集结果，否则，主进程结束，进程池可能还没来得及执行，也就跟着一起结束了
    res=q.apply_async(work,args=(2,))
    q.close()
    q.join() #join在close之后调用
    print(res.get())

    #同步apply用法：主进程一直等apply提交的任务结束后才继续执行后续代码
    # res=q.apply(work,args=(2,))
    # print(res)
```

详解`apply_async`与`apply`:

一：使用进程池（非阻塞,apply_async）

```
#一：使用进程池（非阻塞,apply_async）
#coding: utf-8
from multiprocessing import Process,Pool
import time

def func(msg):
    print( "msg:", msg)
    time.sleep(1)
    return msg

if __name__ == "__main__":
    pool = Pool(processes = 3)
    res_l=[]
    for i in range(10):
        msg = "hello %d" %(i)
```

```
        res=pool.apply_async(func, (msg, ))    #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去
        res_l.append(res)
    print("==============================>") #没有后面的join，或get，则程序整体结束，进程池中的任务还没来得及全部执行完也都跟着主进程一起结束了

    pool.close() #关闭进程池，防止进一步操作。如果所有操作持续挂起，它们将在工作进程终止前完成
    pool.join()   #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束

    print(res_l) #看到的是<multiprocessing.pool.ApplyResult object at 0x10357c4e0>对象组成的列表,而非最终的结果,但这一步是在join后执行的,证明结果已经计算完毕,剩下的事情就是调用每个对象下的get方法去获取结果
    for i in res_l:
        print(i.get()) #使用get来获取apply_aync的结果,如果是apply,则没有get方法,因为apply是同步执行,立刻获取结果,也根本无需get
```

二：使用进程池（阻塞,apply）

```
#二：使用进程池（阻塞,apply）
#coding: utf-8
from multiprocessing import Process,Pool
import time

def func(msg):
    print( "msg:", msg)
    time.sleep(0.1)
    return msg

if __name__ == "__main__":
    pool = Pool(processes = 3)
    res_l=[]
    for i in range(10):
        msg = "hello %d" %(i)
        res=pool.apply(func, (msg, ))   #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去
        res_l.append(res) #同步执行，即执行完一个拿到结果，再去执行另外一个
    print("==============================>")
    pool.close()
    pool.join()   #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束

    print(res_l) #看到的就是最终的结果组成的列表
    for i in res_l: #apply是同步的，所以直接得到结果，没有get()方法
        print(i)
```

练习：使用进程池维护固定数目的进程

server端：

```
#Pool内的进程数默认是cpu核数，假设为4（查看方法os.cpu_count()）
#开启6个客户端，会发现2个客户端处于等待状态
```

```
#在每个进程内查看pid，会发现pid使用为4个，即多个客户端公用4个进程
from socket import *
from multiprocessing import Pool
import os

server=socket(AF_INET,SOCK_STREAM)
server.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
server.bind(('127.0.0.1',8080))
server.listen(5)

def talk(conn,client_addr):
    print('进程pid: %s' %os.getpid())
    while True:
        try:
            msg=conn.recv(1024)
            if not msg:break
            conn.send(msg.upper())
        except Exception:
            break

if __name__ == '__main__':
    p=Pool()
    while True:
        conn,client_addr=server.accept()
        p.apply_async(talk,args=(conn,client_addr))
        # p.apply(talk,args=(conn,client_addr)) #同步的话，则同一时间只有一个客户端能访问
```

客户端：

```
from socket import *

client=socket(AF_INET,SOCK_STREAM)
client.connect(('127.0.0.1',8080))


while True:
    msg=input('>>: ').strip()
    if not msg:continue

    client.send(msg.encode('utf-8'))
    msg=client.recv(1024)
    print(msg.decode('utf-8'))
```

3.5.3 回调函数

回调函数是`apply_async()`的扩展用法。

`apply_async(func, args=(), kwds={}, callback=None)`

参数`callback`指定一个函数名，`func`执行的结果，交给`callback`处理。

不需要用回调函数的场景：如果在主进程中等待进程池中所有任务都执行完毕后，再统一处理结果，则需使用回调函数。

如下例代码：

```
from multiprocessing import Pool
import time,random,os

def work(n):
    time.sleep(1)
    return n**2
if __name__ == '__main__':
    p=Pool()

    res_l=[]
    for i in range(5):
        res=p.apply_async(work,args=(i,))
        res_l.append(res)

    p.close()
    p.join() #等待进程池中所有进程执行完毕

    nums=[]
    for res in res_l:
        nums.append(res.get()) #拿到所有结果
    print(nums) #主进程拿到所有的处理结果,可以在主进程中进行统一进行处理


'''
输出结果:
[0, 1, 4, 9, 16]
'''
```

需要回调函数的场景：进程池中任何一个任务一旦处理完了，就立即告知主进程：“我好了，你可以处理我的结果了。”主进程则调用一个函数去处理该结果，该函数即**回调函数。**

我们可以把耗时（阻塞）的任务放到进程池中，然后指定回调函数（主进程负责执行），这样主进程在执行回调函数时，就省去了**I/O**的过程，直接拿到的是任务的结果。

```
from multiprocessing import Pool
import time,random,os

def get_page(url):
    print('(进程 %s) 正在下载页面 %s' %(os.getpid(),url))
    time.sleep(random.randint(1,3))
    return url #用url充当下载后的结果

def parse_page(page_content):
    print('<进程 %s> 正在解析页面: %s' %(os.getpid(),page_content))
    time.sleep(1)
    return '{%s 回调函数处理结果:%s}' %(os.getpid(),page_content)


if __name__ == '__main__':
    urls=[
        'http://maoyan.com/board/1',
        'http://maoyan.com/board/2',
        'http://maoyan.com/board/3',
```

```
        'http://maoyan.com/board/4',
        'http://maoyan.com/board/5',
        'http://maoyan.com/board/7',

    ]
    p=Pool()
    res_l=[]

    #异步的方式提交任务,然后把任务的结果交给callback处理
    #注意:会专门开启一个进程来处理callback指定的任务(单独的一个进程,而且只有一个)
    for url in urls:
        res=p.apply_async(get_page,args=(url,),callback=parse_page)
        res_l.append(res)

    #异步提交完任务后,主进程先关闭p(必须先关闭),然后再用p.join()等待所有任务结束(包括
callback)
    p.close()
    p.join()
    print('{主进程 %s}' %os.getpid())

    #收集结果,发现收集的是get_page的结果
    #所以需要注意了:
    #1. 当我们想要在将get_page的结果传给parse_page处理,那么就不需要i.get(),通过指定
callback,就可以将i.get()的结果传给callback执行的任务
    #2. 当我们想要在主进程中处理get_page的结果,那就需要使用i.get()获取后,再进一步处理
    for i in res_l: #本例中,下面这两步是多余的
        callback_res=i.get()
        print(callback_res)

'''
打印结果:
(进程 52346) 正在下载页面 http://maoyan.com/board/1
(进程 52347) 正在下载页面 http://maoyan.com/board/2
(进程 52348) 正在下载页面 http://maoyan.com/board/3
(进程 52349) 正在下载页面 http://maoyan.com/board/4
(进程 52348) 正在下载页面 http://maoyan.com/board/5
<进程 52345> 正在解析页面: http://maoyan.com/board/3
(进程 52346) 正在下载页面 http://maoyan.com/board/7
<进程 52345> 正在解析页面: http://maoyan.com/board/1
<进程 52345> 正在解析页面: http://maoyan.com/board/2
<进程 52345> 正在解析页面: http://maoyan.com/board/4
<进程 52345> 正在解析页面: http://maoyan.com/board/5
<进程 52345> 正在解析页面: http://maoyan.com/board/7
{主进程 52345}
http://maoyan.com/board/1
http://maoyan.com/board/2
http://maoyan.com/board/3
http://maoyan.com/board/4
http://maoyan.com/board/5
http://maoyan.com/board/7
'''
```

爬虫案例:

```
from multiprocessing import Pool
```

```
import time,random
import requests
import re

def get_page(url,pattern):
    response=requests.get(url)
    if response.status_code == 200:
        return (response.text,pattern)

def parse_page(info):
    page_content,pattern=info
    res=re.findall(pattern,page_content)
    for item in res:
        dic={
            'index':item[0],
            'title':item[1],
            'actor':item[2].strip()[3:],
            'time':item[3][5:],
            'score':item[4]+item[5]

        }
        print(dic)
if __name__ == '__main__':
    pattern1=re.compile(r'<dd>.*?board-index.*?>(\d+)<.*?title="(.*?)".*?star.*?
>(.*?)<.*?releasetime.*?>(.*?)<.*?integer.*?>(.*?)<.*?fraction.*?>(.*?)<',re.S)

    url_dic={
        'http://maoyan.com/board/7':pattern1,
    }

    p=Pool()
    res_l=[]
    for url,pattern in url_dic.items():
        res=p.apply_async(get_page,args=(url,pattern),callback=parse_page)
        res_l.append(res)

    for i in res_l:
        i.get()

    # res=requests.get('http://maoyan.com/board/7')
    # print(re.findall(pattern,res.text))
```

四、paramiko模块

4.1 介绍

paramiko是一个用于做远程控制的模块，使用该模块可以对远程服务器进程命令或文件操作，值得一说的是，fabric和ansible内部的远程管理就是使用paramiko来实现的。

4.2 下载安装

```
pycrypto，由于 paramiko 模块内部依赖pycrypto，所以先下载安装pycrypto ，python3可能不需要安装pycrypto.

yum -y install libffi-devel python-devel openssl-devel gcc

pip3 install pycrypto
pip3 install paramiko
```

4.3 使用

参数解释

执行命令

登录验证用户名+密码

```
import paramiko

ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())  #此代码作用是允许连接不在know_hosts文件中的主机。
ssh.connect('192.168.1.108', 22, 'alex', '123')
stdin, stdout, stderr = ssh.exec_command('df')
print(stdout.read())
ssh.close();
```

登录密钥验证：

```
import paramiko

private_key_path = '/home/auto/.ssh/id_rsa'
key = paramiko.RSAKey.from_private_key_file(private_key_path)

ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect('主机名 ', 端口, '用户名', key)

stdin, stdout, stderr = ssh.exec_command('df')
print(stdout.read())
ssh.close()
```

上传下载

登录验证用户名+密码

```
import os,sys
import paramiko

t = paramiko.Transport(('182.92.219.86',22))
t.connect(username='wupeiqi',password='123')
```

```
sftp = paramiko.SFTPClient.from_transport(t)
sftp.put('/tmp/test.py','/tmp/test.py')
t.close()


import os,sys
import paramiko

t = paramiko.Transport(('182.92.219.86',22))
t.connect(username='wupeiqi',password='123')
sftp = paramiko.SFTPClient.from_transport(t)
sftp.get('/tmp/test.py','/tmp/test2.py')
t.close()
```

登录验证密钥

```
import paramiko

pravie_key_path = '/home/auto/.ssh/id_rsa'
key = paramiko.RSAKey.from_private_key_file(pravie_key_path)

t = paramiko.Transport(('182.92.219.86',22))
t.connect(username='wupeiqi',pkey=key)

sftp = paramiko.SFTPClient.from_transport(t)
sftp.put('/tmp/test3.py','/tmp/test3.py')

t.close()

import paramiko

pravie_key_path = '/home/auto/.ssh/id_rsa'
key = paramiko.RSAKey.from_private_key_file(pravie_key_path)

t = paramiko.Transport(('182.92.219.86',22))
t.connect(username='wupeiqi',pkey=key)

sftp = paramiko.SFTPClient.from_transport(t)
sftp.get('/tmp/test3.py','/tmp/test4.py')

t.close()
```