1. 进程

1.1 进程、程序

编写完毕的代码，在没有运行的时候，称之为程序

正在运行着的代码，就成为进程

进程，除了包含代码以外，还有需要运行的环境等，所以和程序是有区别的

- 程序是存放在存储介质上的一个文件，而进程是程序执行的过程。
- 进程的状态是变化的，其包括进程的创建、调度和消亡。
- 程序是静态的，进程是动态的。

操作系统是通过进程去完成一个一个的任务，进程是管理事务的基本单元，也是操作系统分配资源的基本单元。

进程拥有自己独立的处理环境（如：当前需要用到哪些环境变量，程序运行的目录在哪，当前是哪个用户在运行此程序等）和系统资源（如：处理器 CPU 占用率、存储器、I/O 设备、数据、程序）。

我们可以这么理解，公司相当于操作系统，部门相当于进程，公司通过部门来管理（系统通过进程管理），对于各个部门，每个部门有各自的资源，如桌子、电脑设备、打印机等。

1.2 进程的创建

1.2.1 多进程的实现-multiprocessing

multiprocessing 模块就是跨平台版本的多进程模块，提供了一个 Process 类来代表一个进程对象，这个对象可以理解为是一个独立的进程，可以执行另外的事情。　　看下面的例子：

```
import time
from multiprocessing import Process


def test1():
    while True:
        print("-----test1-----")
        time.sleep(1)


def test2():
    while True:
        print("-----test2-----")
        time.sleep(1)


if   name   == "  main  ":
    # 指定进程处理函数
    p1 = Process(target=test1)
    p2 = Process(target=test2)
    # 启动进程
```

```
    p1.start()
    p2.start()
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 Process 实例，用 start()方法启动。

1) Process 语法

Process 语法结构如下：

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

- group：指定进程组，大多数情况下用不到 target：如果传递了函数的引
- 用，可以任务这个子进程就执行这里的代码 name：给进程设定一个名字，
- 可以不设定
- args：给 target 指定的函数传递的参数，以元组的方式传递
- kwargs：给 target 指定的函数传递命名参数
- daemon：是否以守护进程运行，True 或 False

2) Process 创建的实例对象的常用方法

- start()：启动子进程实例（创建子进程） is_alive()：判断
- 进程子进程是否还在活着 join([timeout])：是否等待子进
- 程执行结束，或等待多少秒 terminate()：不管任务是否
- 完成，立即终止子进程

3) Process 创建的实例对象的常用属性

- name：当前进程的别名，默认为 Process-N，N 为从 1 开始递增的整数
- pid：当前进程的 pid（进程号）

1.2.2 进程的创建-Process子类

创建新的进程还能够使用类的方式，可以自定义一个类，继承Process类，每次实例化这个类的时候，就等同于实例化一个进程对象，请看下面的实例：

```
from multiprocessing import Process
import time
import os


# 继承Process类
class Process_Class(Process):
    # 因为Process类本身也有 init 方法，这个子类相当于重写了这个方法，
    # 但这样就会带来一个问题，我们并没有完全的初始化一个Process类，所以就不能使用从这个类继承的一些方法和属性，
    # 最好的方法就是将继承类本身传递给Process. init 方法，完成这些初始化操作
    def init (self, interval):
        Process. init (self)
        self.interval = interval

    # 重写了Process类的run()方法
```

```
    def run(self):
        print("子进程(%s) 开始执行，父进程为 (%s) " % (os.getpid(), os.getppid()))
        t_start = time.time()
        time.sleep(self.interval)
        t_stop = time.time()
        print("(%s)执行结束，耗时%0.2f秒" % (os.getpid(), t_stop - t_start))


if name == " main ":
    t_start = time.time()
    print("当前程序进程(%s)" %os.getpid())
    p1 = Process_Class(2)
    #    对一个不包含target属性的Process类执行start()方法，就会运行这个类中的run()方法，所以这里会执行
p1.run()
    p1.start()
    p1.join()
    t_stop = time.time()
    print("(%s)执行结束，耗时%0.2f" % (os.getpid(), t_stop - t_start))
```

1.3 进程号

每个进程都由一个进程号来标识，进程号总是唯一的。

```
import multiprocessing
import os


def func():
    pid = os.getpid()  #获取进程号
    ppid = os.getppid()  #获取父进程
    print("子进程号：%d，其父进程号：%d" % (pid, ppid))


def main():
    # 获取当前进程号
    pid = os.getpid()
    print("主线程的线程号：", pid)
    # 创建子进程
    p =multiprocessing.Process(target=func)
    # 启动进程
    p.start()


if   name   ==  "  main  ":
    main()
```

```
主线程的线程号： 7561
子进程号：7562，其父进程号：7561
```

1.4 给子进程指定的函数传递参数

```
import multiprocessing


def runProc(*args, **kwargs):
    print(args, kwargs)


if __name__ == '__main__':
    # 创建子进程，并传递指定参数
    p = multiprocessing.Process(target=runProc, args=('test', 18), kwargs={"name": "mike"})
    p.start()  # 启动子进程
    p.join()  # 等待子进程执行结束
    print("it is over!!!")
```

运行结果：

```
('test', 18) {'name': 'mike'}
it is over!!!
```

1.5 进程间不同享全局变量

```
import multiprocessing

g_nums = [11, 22, 33]


def work1():
    g_nums.append(44)
    print("-----work1----", g_nums)


def work2():
    print("-----work2----", g_nums)


if __name__ == '__main__':
    # 子进程 1
    p1 =multiprocessing.Process(target=work1)
    p1.start()
    p1.join()  # 等待 p1 进程结束
    # 子进程 2
    p2 =multiprocessing.Process(target=work2)
    p2.start()
```

运行结果：

```
-----work1---- [11, 22, 33, 44]
-----work2---- [11, 22, 33]
```

1.6 进程间通信：Queue

进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。

但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信(IPC：Inter Processes Communication)。

1.6.1 Queue的使用

可以使用 multiprocessing 模块的 Queue 实现多进程之间的数据传递，Queue 本身是一个消息列队程序，这是操作系统开辟的一个空间,可以让各个子进程把信息放到Queue中,也可以把自己需要的信息取走，这就相当于系统给python开辟了一个聊天室,让python创建的子进程可以在这个聊天室里畅所欲言，一个进程可以放多条消息到Queue中。

实例

```
import multiprocessing

q = multiprocessing.Queue(3)  # 初始化一个 Queue 对象，最多可接收三条 put 消息
q.put("消息 1")
q.put("消息 2")
print(q.full())  # False

q.put("消息 3")
print(q.full())  # True
# 因为消息列队已满下面的 try 都会抛出异常，第一个 try 会等待 2 秒后再抛出异常，第二个Try 会立刻抛出异常
try:
    q.put("消息 4", True, 2)
except:
    print("消息列队已满，现有消息数量:%s" % q.qsize())

try:
    q.put_nowait("消息 4")
except:
    print("消息列队已满，现有消息数量:%s" % q.qsize())
# 推荐的方式，先判断消息列队是否已满，再写入
if not q.full():
    q.put_nowait("消息 4")
# 读取消息时，先判断消息列队是否为空，再读取
if not q.empty():
    for i in range(q.qsize()):
        print(q.get_nowait())
```

运行结果：

```
False
True
消息列队已满，现有消息数量:3
消息列队已满，现有消息数量:3
消息 1
消息 2
消息 3
```

1.6.2 说明

初始化 Queue()对象时（例如：q = Queue()），若括号中没有指定最大可接收的消息数量，或数量为负值，那么就代表可接受的消息数量没有上限（直到内存的尽头）。

- Queue.qsize()：返回当前队列包含的消息数量
- Queue.empty()：如果队列为空，返回 True，反之 False
- Queue.full()：如果队列满了，返回 True,反之 False
- Queue.get([block[, timeout]])：获取队列中的一条消息，然后将其从列队中移除，block 默认值为 True
    - 如果 block 使用默认值，且没有设置 timeout（单位秒），消息列队如果为空，此时程序将被阻塞（停在读取状态），直到从消息列队读到消息为止，如果设置了timeout，则会等待 timeout 秒，若还没读取到任何消息，则抛出"Queue.Empty"异常
    - 如果block值为False，消息列队如果为空，则会立刻抛出"Queue.Empty"异常；
- Queue.get_nowait()：相当 Queue.get(False)
- Queue.put(item,[block[, timeout]])：将 item 消息写入队列，block 默认值为 True
    - 如果 block 使用默认值，且没有设置 timeout（单位秒），消息列队如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息列队腾出空间为止，
    - 如果设置了 timeout，则会等待 timeout 秒，若还没空间，则抛出"Queue.Full"异常,如果 block 值为 False，消息列队如果没有空间可写入，则会立刻抛出"Queue.Full"异常
- Queue.put_nowait(item)：相当 Queue.put(item, False)

1.6.3 Queue 实例

```
import multiprocessing
import time
import random


def write(q):
    for value in ['a', 'b', 'c']:
        print('put %s to queue...' %value)
        q.put(value)
        time.sleep(random.random())


def read(q):
    while True:
        if not q.empty():
```

```
            value = q.get(True)
            print('Get %s from queue' % value)
            time.sleep(random.random())
        else:
            break


if __name__ == '__main__':
    q = multiprocessing.Queue()
    pw = multiprocessing.Process(target=write, args=(q,))
    pr = multiprocessing.Process(target=read, args=(q,))

    pw.start()
    pw.join()

    pr.start()
    pr.join()

    print('')
    print('全部写入读取完成')
```

运行结果：

```
put a to queue...
put b to queue...
put c to queue...
Get a from queue
Get b from queue
Get c from queue

全部写入读取完成
```

1.7 进程池Pool

1.7.1 进程池的使用

当需要创建的子进程数量不多时，可以直接利用 multiprocessing 中的 Process 动态成生多个进程，但如果是上百甚至上千个目标，手动的去创建进程的工作量巨大，此时就可以用到multiprocessing 模块提供的 Pool 方法。

初始化 Pool 时，可以指定一个最大进程数，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指定的最大值，那么该请求就会等待，直到池中有进程结束，才会用之前的进程来执行新的任务。

```
import time
from multiprocessing import Pool

def run(f):
    time.sleep(1)
    return f*f
```

```
if name == " main  ":
    test = [1,2,3,4,5,6]
    print ('shunxu:') #顺序执行(也就是串行执行，单进程)
    s = time.time()
    for f in test:
        run(f)

    e1 = time.time()
    print("顺序执行时间：", int(e1 - s))

    print( 'concurrent:')#创建多个进程，并行执行
    pool = Pool(5) #创建拥有5个进程数量的进程池
    #test:要处理的数据列表，run：处理test列表中数据的函数
    rl =pool.map(run, test)
    pool.close()#关闭进程池，不再接受新的进程
    pool.join()#主进程阻塞等待子进程的退出
    e2 = time.time()
    print ("并行执行时间：",int(e2-e1))
    print (rl)
```

```
shunxu:
顺序执行时间： 6
concurrent:
并行执行时间： 2
[1, 4, 9, 16, 25, 36]
```

上例是一个创建多个进程并发处理与顺序执行处理同一数据，所用时间的差别。从结果可以看出，并发执行的时间明显比顺序执行要快很多，但是进程是要耗资源的，所以平时工作中，进程数也不能开太大。

程序中的r1表示全部进程执行结束后全局的返回结果集，run函数有返回值，所以一个进程对应一个返回结果，这个结果存在一个列表中，也就是一个结果堆中，实际上是用了队列的原理，等待所有进程都执行完毕，就返回这个列表（列表的顺序不定）。

对Pool对象调用join()方法会等待所有子进程执行完毕，调用join()之前必须先调用close()，让其不再接受新的Process了。

1.7.2 multiprocessing.Pool 常用函数解析：

- apply_async(func[, args[, kwds]])： 使用非阻塞方式调用 func（并行执行，堵塞方式必须等待上一个进程退出才能执行下一个进程），args 为传递给 func 的参数列表，kwds为传递给 func 的关键字参数列表
- close()： 关闭 Pool，使其不再接受新的任务
- terminate()： ： 不管任务是否完成，立即终止
- join()： 主进程阻塞，等待子进程的退出， 必须在 close 或 terminate 之后使用

1.7.3 进程池中的 Queue

如果要使用 Pool 创建进程，就需要使用 multiprocessing.Manager()中的 Queue()，而不是 multiprocessing.Queue()，否则会得到一条如下的错误信息：

```
RuntimeError: Queue objects should only be shared between processes thr
ough inheritance.
```

下面的实例演示了进程池中的进程如何通信：

```
import os, time, random, multiprocessing


def reader(q):
    print("reader 启动(%s),父进程为(%s)" % (os.getpid(), os.getppid()))
    for i in range(q.qsize()):
        print("reader 从 Queue 获取到消息：%s" % q.get(True))


def writer(q):
    print("writer 启动(%s),父进程为(%s)" % (os.getpid(), os.getppid()))
    for i in "itcast":
        q.put(i)


if __name__ == "__main__":
    print("(%s) start" %os.getpid())
    q = multiprocessing.Manager().Queue()  # 使用 Manager 中的 Queue
    po = multiprocessing.Pool()
    po.apply_async(writer, (q,))
    time.sleep(1) # 先让上面的任务向 Queue 存入数据，然后再让下面的任务开始从中取数据
    po.apply_async(reader, (q,))
    po.close()
    po.join()
    print("(%s) End" %os.getpid())
```

运行结果：

```
(3020) start
writer 启动(11508),父进程为(3020)
reader 启动(11236),父进程为(3020)
reader 从 Queue 获取到消息：i
reader 从 Queue 获取到消息：t
reader 从 Queue 获取到消息：c
reader 从 Queue 获取到消息：a
reader 从 Queue 获取到消息：s
reader 从 Queue 获取到消息：t
(3020) End
```

1.8 案例:多进程拷贝文件

```
import multiprocessing
import os
```

```python
import time

def copyFile(q, fileName, srcFolderName, destFolderName):
    """完成文件的 copy"""
    # print("正在 copy 文件%s" %fileName)

    srcFile = open(srcFolderName + "/" + fileName, "rb") #读方式打开
    destFile = open(destFolderName + "/" + fileName, "wb")#写方式打开

    #读取源文件内容，拷贝到目标文件
    content = srcFile.read()
    destFile.write(content)

    #关闭文件
    srcFile.close()
    destFile.close()

    #适当延时
    time.sleep(0.2)

    #文件名放入 Queue 中
    q.put(fileName)

def main():
    """完成整体的控制"""
    # 1. 获取需要 copy 的文件夹的名字
    srcFolderName = input("请输入需要 copy 的文件夹的名字: ")

    # 2. 根据需要 copy 的文件夹的名字，整理一个新的文件夹的名字
    destFolderName = srcFolderName +"[复件]"

    # 3. 创建一个新的文件夹
    try:
        os.mkdir(destFolderName)
    except Exception as ret:
        pass

    # 4. 获取文件夹中需要 copy 的文件名字
    fileNameList =os.listdir(srcFolderName)

    # 5.1 创建一个队列
    q =multiprocessing.Manager().Queue()

    # 5.2 创建一个进程池，完成 copy
    pool =multiprocessing.Pool(5)

    # 6. 向进程池中添加任务
    for fileName in fileNameList:
        pool.apply_async(copyFile, (q, fileName, srcFolderName, destFolderName))

        allNum = len(fileNameList)
        currentNum = 0
```

```
        while True:
            fileName = q.get() #取出完成的文件名
            currentNum += 1

            # print("已经完成了从%s-------(%s)------>%s" % (srcFolderName, fileName,
destFolderName))
            print("\r 进度%.2f%%" % (100*currentNum/allNum), end="")

            #拷贝完成
            if currentNum == allNum:
                break

        #  不再向进程池中添加任务，并且等待所有的任务结束
        pool.close()
        pool.join()
        print("\n\nok..")

if name == " main ":
    main()
```

2. 线程

2.1 进程和线程的区别

功能:

- 进程，能够完成多任务，比如在一台电脑上能够同时运行多个 QQ
- 线程，能够完成多任务，比如一个 QQ 中的多个聊天窗口

定义:

- 进程是系统进行资源分配和调度的一个独立单位
- 线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

区别:

- 一个程序至少有一个进程,一个进程至少有一个线程.
- 线程的划分尺度小于进程(资源比进程少)，使得多线程程序的并发性高。
- 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率
- 线线程不能够独立执行，必须依存在进程中

举个现实中生活例子来说明一下进程和线程的关系：

如果说进程是一个资源管家，负责从主人那里要资源的话，那么线程就是干活的苦力。一个管家必须完成一项工作，就需要最少一个苦力，也就是说，一个进程最少包含一个线程，也可以包含多个线程。苦力要干活，就需要依托于管家，所以说一个线程，必须属于某一个进程。

优缺点:

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。

2.2 threading模块

python 的 thread 模块是比较底层的模块，python 的 threading 模块是对 thread 做了一些包装的，可以更加方便的被使用。

单线程执行:

```
#coding=utf-8
import time


def test():
    print(大佬牛逼")
    time.sleep(1)


if name == " main ":
    begintime = time.time()
    for i in range(5):
        test()
    lasttime = time.time()
    print('耗时：%d' % (lasttime - begintime))
```

运行结果：

```
大佬牛逼
大佬牛逼
大佬牛逼
大佬牛逼
大佬牛逼
耗时：5
```

多线程执行:

方法一：将要执行的方法作为参数传给Thread的构造方法

```
# coding=utf-8
import threading
import time


def test():
    print("大佬牛逼")
    time.sleep(1)


if name == " main ":
    btime = time.time()
    for i in range(5):
        t =threading.Thread(target=test)
```

```
        t.start()  #启动线程，即让线程开始执行
    etime = time.time()
    print('耗时：%d' % (etime - btime))
```

```
大佬牛逼
大佬牛逼
大佬牛逼
大佬牛逼
大佬牛逼
耗时：0
```

方法二：为了让每个线程的封装性更完美，在使用 threading 模块时，往往会定义一个新的子类class，只要继承threading.Thread 就可以了，然后重写 run 方法。

```
import threading
import time


class MyThread(threading.Thread):
    def run(self):
        for i in range(5):
            time.sleep(1)
            print("大佬牛逼")


  if name == ' main ':
   btime = time.time()
   t = MyThread()
   t.start()
   etime = time.time()
   print('耗时：%d' % (etime - btime))
```

运行结果：

```
耗时：0
大佬牛逼
大佬牛逼
大佬牛逼
大佬牛逼
大佬牛逼
```

```
1. 可以明显看出使用了多线程并发的操作，花费时间要短很多
2. 创建好的线程，需要调用start()方法来启动
```

2.2.1 构造方法：

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

- group: 线程组，目前还没有实现，库引用中提示必须是None；
- target: 要执行的方法；
- name: 线程名；
- args/kwargs: 要传入方法的参数。

2.2.2 实例方法：

- isAlive(): 返回线程是否在运行。正在运行指启动后、终止前。
- get/setName(name): 获取/设置线程名。
- start(): 线程准备就绪，等待CPU调度
- is/setDaemon(bool): 获取/设置是后台线程（默认前台线程（False））。（在start之前设置）
    - 如果是后台线程，主线程执行过程中，后台线程也在进行，主线程执行完毕后，后台线程不论成功与否，主线程和后台线程均停止
    - 如果是前台线程，主线程执行过程中，前台线程也在进行，主线程执行完毕后，等待前台线程也执行完成后，程序停止
- start(): 启动线程。
- join([timeout]): 阻塞当前上下文环境的线程，直到调用此方法的线程终止或到达指定的timeout（可选参数）。