进程和线程

1. 大部分操作系统的任务调度采用时间片轮转的方式
    一个任务执行一小段时间(us)，会强制暂停，执行下一个任务，每个任务轮回执行
    时间片：一小段时间
    运行状态：任务执行的状态
    就绪状态：暂停的任务
2. CPU的执行效率高，时间片短，个个任务之间的切换就快，人感觉不到的---感觉并行
    宏观并行，微观串行

进程

1. 进程是一个独立功能的程序，在一个数据集合的一次动态执行过程，是操作系统进行资源分配和调度的以独立单位
    进程是正在执行的程序
    操作系统将时间片分配给了进程
    进程和进程之间相互独立，资源不共享
2. 进程是一个抽象的概念，没有统一的定义，由：程序，数据集合，进程块三部分构成
    程序：用于描述进程要完成的功能，是进程执行的指令
    数据集合：程序在执行的时候所需要的数据和工作区
    进程控制块：包含进程的描述信息，是进程存在的唯一标识
3. 进程的特性：
    1. 动态性：
        进程实质时候程序在多道程序中的执行过程，进程是动态产生，动态消亡
    2. 并发性：
        任何进程都可以同其他进程一起并发执行
    3. 独立性：
        进程是一个独立运行的基本单位，同时也是系统分配资源和调度的独立单
    4. 结构性

线程

1. 一个进程可以有很多线程,每个线程执行的任务不同
    同一个进程中有很线程，将共享进程中的全部资源
2. 一个线程由：线程ID，指令集，栈，寄存器组成
    线程ID：线程的唯一标识
    指令集：线程首先本质上就是一个指令集，通过这个指令集告诉CPU自己要做的事情
    栈：当创建一个线程时，就应该有一个专属的内存空间来存储线程的相关信息，这块内存空间就是栈
    寄存器：也是工作内存多线程共享一块主内存，工作内存

总结：

1. 线程是进程执行的最小单位，进程是操作系统分配资源的最小单位
2. 线程是一个进程中代码的不同执行路线
3. 线程和线程之间互相独立，但是资源共享（共享进程的资源）
4. 线程的调度和切换，线程的上下文切换要比进程的切换速度快的多

单线程和多线程

单线程：一个进程只有一个线程
多线程：一个进程中有多个线程

任何一个程序至少有一个线程====主线程

```
#单线程
def fun(music,count):
    for i in range(count):
        print('listen to the music %s遍'%(i+1))

def fun2(movie,count):
    for i in range(count):
        print('watch movie %s遍'%(i+1))


if __name__=='__main__':
    fun('凉凉',3)
    fun2('中国机长',3)
```

多线程模块：threading

Python提供了多种模块用来支持多线程编程
创建线程的方式
1. 继承Thread类并实现run方法
        threading.Thread类构造方法
        __init__(self,group=None,traget=None,name=None,args())
        group:预留参数，用于扩展功能
        target：可调对象（目标函数）
        name:线程的名字

```
import threading

# 继承thread类，实现run方法
class MyThead(threading.Thread):
    def __init__(self,name):
        super().__init__()
        self.name=name

    def run(self):
        #线程要做的事
        for i in range(10):
            print(self.name)
t1=MyThead('hello')
t2=MyThead('world')

t1.start()
t2.start()
```

2. 通过Thrad类的构造方法直接创建线程

```
def fun(music,count):
```

```
    for i in range(count):
        print('listen to the music %s遍'%(i+1))

def fun2(movie,count):
    for i in range(count):
        print('watch movie %s遍'%(i+1))


if __name__=='__main__':
    t1=threading.Thread(target=fun,args=('野狼disco！',3))
    t2=threading.Thread(target=fun2,args=('音乐僵尸',3))
    t1.start()
    t2.start()
    print(11111)
```

- Python的多线程创建形式

```
1. 创建线程（两种形式）
2. 开启线程
```

- threading.Thread类的方法

```
1. getName()
        获取当前线程的名字--Thread-N
2. setName（newname）
        设置当前的线程名字
        也可以在父类的构造方法中直接设置name=new—name
3. ident
        返回当前线程id
4. join()
        调用Thread.join（）会阻塞当前线程（其他的不会阻塞），直到被调用join的线程执行完毕，再继续执行当前线程
5. setDaemon()
        设置守护线程（精灵线程）
        主线程A，启动了子线程B，调用B.setDaemon(True)，则主线程结束，会把所有的守护杀死，必须在运行线程之前设置
6. is_alive
        返回当前的线程状态

1. 子线程
        threading.Thread创建的
2. 主线程
        系统创建的线程
```

线程状态

2019-11-05_100132

GIL全局锁

GIL：全局解释锁
定义：解释器通过这种机制获取全局解释锁来限制同一时间点只允许一个线程执行，任何一个时间点只有一个线程处于执行状态
1. GIL锁不是Python的特性，Python完全可以不依赖GIL
2. 为了更有效的利用多核性质，出现了多进程的编程方式
3. GIL锁极大影响了多线程的多核性质，多线程的效率受到了影响，让多线程几乎等同于单线程
        低效的
4. 当多核cpu处理多线程时，GIL锁的释放和重新获取时间间隔，过于短暂，导致其他线程再激活其他核之后，长时间占用核的性能，但是不能得到GIL而不能执行--极大的降低了CPU多核的性质
5. 单核不受影响
6. Python 多线程多用于IO密集型程序
        任务类型：
            IO密集型任务：网络IO磁盘IO，cpu消耗很少
            计算密集型任务：消耗CPU多，是要进行大量计算
优点：
    1. 提高单线程的执行速度
    2. 更易于集成C扩展模块
缺点：
    无法利用多核，无法很好计算密集型任务
解决方案：
    1. 利用多进程
        进程池会在另一个进程中启动一个单独的Python解释器环境来工作，当线程等待结果的时候会释放GIL
    2. 使用C扩展编程技术（cpython，ctypes在调用c自动释放GIL）
    3. 多解释器（pypy，jpython）

线程同步

数据安全问题--线程同步
GIL--锁--数据的安全
原子操作：不可分割的逻辑
线程同步需要加锁
        threading.Lock()
加了锁的程序---效率低---安全
不加锁---效率高---数据不安全

```
import threading,time
lock=threading.Lock()
l=['A','B','','','']
index=2

def fun(char):
    global index
    lock.acquire()
    l[index]=char
    time.sleep(1)
    index+=1
```

```
    lock.release()


if __name__=='__main__':
    t1=threading.Thread(target=fun,args=('C',))
    t2=threading.Thread(target=fun,args=('D',))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(l)
```

死锁

Lock：只能acquire一次，下一次acquire必须在release，不然会造成死锁

```
import threading,time
l=['A','B','','','']
index=2
lock=threading.Lock()

def fun(char):
    global index
    lock.acquire()
    print(1111111111111111)
    lock.acquire()
    l[index]=char
    time.sleep(0.02)
    index+=1
    lock.release()
    lock.release()

if __name__=='__main__':
    t1=threading.Thread(target=fun,args=('C',))
    t2=threading.Thread(target=fun,args=('D',))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(l)
```

解决死锁：只需要把lock=threading.RLock()即可，在同一线程内对RLock进行多次acquire，，程序不会阻塞，一定要注意acquire要和release次数相等

```
#不在同一线程内：
#以下代码形成死锁
import threading,time
```

```
lock1=threading.Lock()
lock2=threading.Lock()

def fun1():
    print('fun1  starting...')
    lock1.acquire()
    print('fun1申请了lock1')
    time.sleep(2)
    print('fun1等待lock2')
    lock2.acquire()
    print('fun1申请了lock2')
    lock2.release()
    print('fun1释放了lock2')
    lock1.release()
    print('fun1释放了lock1')
    print('fun1 end')

def fun2():
    print('fun2  starting')
    lock2.acquire()
    print('fun2申请了lock2')
    time.sleep(4)
    print('fun2等待lock1')
    lock1.acquire()
    print('fun2申请了lock1')
    lock1.release()
    print('fun2释放了lock1')
    lock2.release()
    print('fun2释放了lock2')
    print('fun2 end')


if __name__=='__main__':
    t1=threading.Thread(target=fun1,args=())
    t2=threading.Thread(target=fun2,args=())
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

```
#使用if解决死锁问题:
def fun4():
    print('fun4 staring')
    lock4.acquire(timeout=4)
    print('fun4申请了lock4')
    time.sleep(2)
    print('fun4等待lock3')
    rst=lock3.acquire(timeout=2)
    if rst:
        print('fun3得到了lock3')
```

```
        lock3.release()
        print('fun4释放lock3')
    else:
        print('fun4没有申请到lock3')
    lock4.release()
    print('fun4释放了lock4')
    print('fun4  end')

if __name__=='__main__':
    t3=threading.Thread(target=fun3,args=())
    t4=threading.Thread(target=fun4,args=())
    t3.start()
    t4.start()
    t3.join()
    t4.join()
```

python的多进程multiprocessing（模块（包））

1. Python的多线程中无法利用多核优势，如果想要充分利用多核CPU的资源，在Python中应该使用多进程
2. 多进程不会受到GIL影响，可以达到并行状态
3. 所包含的组件（类）
    1. Process---进程类
    2. Queue --队列，实现多进程之间的数据传递
    3. Pipe --管道，实现管道模式下消息的发送和接收
    4. Lock --进程锁，可以避免访问资源时的冲突（进程同步）
    5. Pool --进程池，可以提供制定数量的进程

- Process--进程类

1. p.start()
    启动进程
2. p.run()
    进程启动时运行的方法

3. p.terminate()
    强制终止P进程，如果p还保存了一个锁，进而导致死锁
4. p.is_alive
    判断进程是否运行
5. join（）
6. deamon
7. name
8. pid
    进程的id

进程的创建：

```
import multiprocessing
import time
def music(name,loop):
    for i in range(loop):
```

```
        print('listen to the %s %s遍'%(name,i+1))
        time.sleep(0.01)

def movie(name,loop):
    for i in range(loop):
        print('watch  the %s %s遍'%(name,i+1))
        time.sleep(0.01)

if __name__=='__main__':
    p1=multiprocessing.Process(target=music,args=('那女孩对我说',3))
    p2=multiprocessing.Process(target=movie,args=('僵尸先生',3))
    p1.start()
    p2.start()
```

- Lock

```
import multiprocessing
import time
lock=multiprocessing.Lock()
def music(name,loop):
    lock.acquire()
    for i in range(loop):
        print('listen to the %s %s遍'%(name,i+1))
        time.sleep(0.01)
    lock.release()

def movie(name,loop):
    lock.acquire()
    for i in range(loop):
        print('watch  the %s %s遍'%(name,i+1))
        time.sleep(0.01)
    lock.release()
if __name__=='__main__':
    p1=multiprocessing.Process(target=music,args=('那女孩对我说',3))
    p2=multiprocessing.Process(target=movie,args=('僵尸先生',3))
    p1.start()
    p2.start()
```

- Pool

进程池
可以提供制定数量的进程，当有新请求提交到pool中，如果池满了则需要等待池中的进程执行完毕才能进入
`Pool([maxsize])`--设置进程池的容量

- pool中的方法

```
1. apply()
        可以阻塞进程池中的进程---一个一个执行
        Python3取消
2. apply_async()
        非阻塞的执行池中的进程，宏观可以达到并行，当主进程执行完毕，其他子进程不在执行
3. close()
        关闭进程池，不在接受任何新的请求（任务）
4. terminate()
        结束工作进程，不在处理未完成的任务
5. join()
        阻塞当前进程，等待子进程退出，在close和terminate之后使用
```

```python
import multiprocessing,time

def work(num):
    print('hello world:%s'%num)
    time.sleep(1)

if __name__=='__main__':
    p=multiprocessing.Pool(5)
    for i in range(5):
        p.apply_async(func=work,args=(i+1,))
    p.close()
    p.join()
    print('end')
```

- Queue

```
队列：先进先出
进程和进程之间是相互独立的，进程间的通信（IPC）
Queue，Pipe
Queue（maxsize=0）
maxsize=0：不限制大小
创建共享进程队列，可以使用Queue实现多进程之间的数据传递
```

- Queue的方法

```
1. put()
        把数据插入到队列中
        blocked：默认True
        timeout：超出的时间
2. get()
        从队列中读取并删除一个元素
        blocked：默认True
        timeout：超出的时间
3. get_nowait
4. put_nowait
5. empty()
```

```
        队列为空返回True,反之返回False
6. full ( )
        队列满了返回True , 反之返回False
7. qsize()
        返回队列中的数量
```

```python
import multiprocessing,time

q=multiprocessing.Queue(5)

q.put(123)
q.put(456)
q.put(789)
q.put(100)
q.put(101)
print(q.get())
q.put(102)
print(q.get())
```

```python
import multiprocessing,time


def put(q):
    for i in ['A','B','C']:
        print('发送%s到队列中'%i)
        q.put(i)
        time.sleep(1)

def get(q):
    while 1:
        value=q.get()
        print('从队列中获取到%s'%value)

if __name__=='__main__':
    q=multiprocessing.Queue()
    p_put=multiprocessing.Process(target=put,args=(q,))
    p_get=multiprocessing.Process(target=get,args=(q,))
    p_put.start()
    p_get.start()
    p_put.join()
    p_get.terminate()
```

- Pipe

管道
1. Pipe (duplex=True)
        返回一个元祖 (conn1 , conn2) 代表管道的两端
        duplex=True , conn1和conn2都可以接受发送 , 如果改为False   conn1只负责接收消息 , conn2只负责发送
2. send() recv()
    分别表示发送和接受消息

```
import multiprocessing,time


def put(q):
    for i in ['A','B','C']:
        print('发送%s到队列中'%i)
        q[1].send(i)
        time.sleep(1)

def get(q):
    while 1:
        value=q[0].recv()
        print('从队列中获取到%s'%value)

if __name__=='__main__':
    pipe=multiprocessing.Pipe()
    p_put=multiprocessing.Process(target=put,args=(pipe,))
    p_get=multiprocessing.Process(target=get,args=(pipe,))
    p_put.start()
    p_get.start()
    p_put.join()
    p_get.terminate()
```

生产者和消费者

1. multiprocessing.Queue
        进程队列，表示进程和进程间通信
2. queue.queue
        线程和线程之间的通信

```
#线程模型

def producer(name): #生产者
    count=1
    while 1:
        q.put('第%s个汽车'%count)
        print('%s生产了%s个汽车'%(name,count))
        count+=1
        time.sleep(1)

def consumer(name):
```

```
    while 1:
        print('%s取到了%s'%(name,q.get()))

if __name__=='__main__':
    prod=threading.Thread(target=producer,args=('老王',))
    cons1=threading.Thread(target=consumer,args=('赵晓龙',))
    cons2=threading.Thread(target=consumer,args=('杨洪举',))
    prod.start()
    cons1.start()
    cons2.start()
```