第十八章 多线程编程

- 引言/动机
- 线程和进程
- Python、线程和全局解释器锁
- thread模块
- threading模块
- 相关模块
- 练习

引言/动机

无论是任务本身要求顺序执行还是整个程序是由多个子任务组成，程序都是按顺序执行的。

运算密集型的任务一般都比较容易分隔成多个子任务，可以顺序执行或以多线程的方式执行。单线程处理多个外部输入源的的任务就不是那么容易了。

> 使用多线程是可以实现多个任务的快速处理,有一个线程负责读取外部输入,一个线程处理这些外部输入,还有一个线程输出这些处理结果。每一个线程都有自己明确的任务。

这种编程任务如果不用多线程的方式处理，则一定要使用一个或多个计时器来实现。

> 使用计时器的原因是要为每一个程序的执行设置时间片。

用非阻塞或者带有计时器的阻塞IO的方法解决一个顺序执行的程序从每个IO终端信道读取用户信息时阻塞的问题。

由于顺序执行的程序只有一个线程在运行。它要保证它要做的多任务，不会有某个任务占用太多的时间，而且要合理地分配用户的响应时间。

使用多线程编程和一个共享的数据结构,计算密集型(可以分为多个子任务)程序任务可以用几个功能单一的线程来组织:

- UserRequestThread: 负责读取客户的输入,可能是一个I/O信道。程序可能创建多个线程,每个客户一个,请求会被放入队列中。
- RequestProcessor: 一个负责从队列中获取并处理请求的线程,它为下面那种线程提供输出。
- ReplyThread: 负责把给用户的输出取出来，如果是网络应用程序就把结果发送出去，否则就保存到本地文件系统或数据库中。

每一个线程负责自己的任务即可。

18.2 线程和进程

进程

计算机程序只不过是磁盘中可执行的，二进制（或其它类型）的数据。**静态**

它们只有在被读取到内存中，被操作系统调用的时候才开始它们的生命期。

进程（有时被称为重量级进程）是程序的一次执行。–**动态**

线程

跟进程有些相似，不同的是，所有的线程运行在同一个进程中,共享相同的运行环境。它们可以想像成是在主进程或“主线程”中并行运行的“迷你进程”。

线程一般都是并发执行的，正是由于这种并行和数据共享的机制使得多个任务的合
作变为可能。

如果多个线程共同访问同一片数据，则由于数据访问的顺序不一样，有可能导致数据结果的不一致的问题。这叫做竞态条件(race condition)。

大多数线程库都带有一系列的同步原语，来控制线程的执行和数据的访问。

> 解决竞态条件

18.3 Python、线程和全局解释器锁

Python代码的执行由Python虚拟机(也叫解释器主循环)来控制。Python 在设计之初就考虑到要在主循环中，同时只有一个线程在执行，就像单CPU的系统中运行多个进程那样,内存中可以存放多个程序,但任意时刻,只有一个程序在CPU中运行。

对 Python 虚拟机的访问由全局解释器锁(GIL)来控制，正是这个锁能保证同一时刻只有一个线程在运行。

在多线程环境中，Python 虚拟机按以下方式执行:

1. 设置GIL
2. 切换到一个线程去运行
3. 运行
    - 指定数量的字节码指令或者
    - 线程主动让出控制
4. 把线程设置为睡眠状态
5. 解锁GIL
6. 再次重复以上所有步骤

18.3.2 退出线程

`thread.extit()`
`sys.exit()`
`kill`
三种方法退出

不推荐使用thread模块,一个原因是，当主线程退出的时候，所有其它线程没有被清除就退出了。

另一个模块 threading 就能确保所有“重要的”子线程都退出后，进程才会结束。

> 主线程应该是一个好的管理者，它要了解每个线程都要做些什么事，线程都需要什么数据和什么参数，以及在线程结束的时候，它们都提供了什么结果。这样，主线程就可以把各个线程的结果组合成一个有意义的最后结果。

18.3.3 在 Python 中使用线程

`pthread`

想要从解释器里判断线程是否可用，只要简单的在交互式解释器里尝试导入thread模块就行了，只要没出现错误就表示线程可用。

如果出现错误,需要重新编译你的Python解释器才能使用线程，在配置的时候加上`--with-thread`参数。

18.3.4 没有线程支持的情况

使用`time.sleep()`函数来演示线程是怎样工作的。

```
from time import sleep,ctime

def loop0():
    print '\tstart loop 0 at:',ctime()
    sleep(4)
    print '\tloop 0 done at:',ctime()

def loop1():
    print '\tstart loop 1 at:',ctime()
    sleep(2)
    print '\tloop 1 done at:',ctime()

# 在单线程中顺序执行两个循环。一定要一个循环结束后，另一个才能开始。总时间是各个循环运行时间之和。
def main():
    print 'starting at:',ctime()
    loop0() # 运算结果1
    loop1() # 运算结果2
    print 'all done at:',ctime()

if __name__ == "__main__":
    main()
```

18.3.5 Python 的 threading 模块

Python提供的多线程编程模块包括thread(不推荐使用)threading,Queue。
thread 模块提供了基本的线程和锁的支持，而 threading提供了更高级别，功能更强的线程管理的功能。

> 只建议那些有经验的专家在想访问线程的底层结构的时候，才使用 thread 模块。

18.4 thread模块

thread模块除了提供线程外,也提供了基本的同步数据结构锁对象(lock object,也叫原语锁,简单锁,互斥锁,互斥量,二值信号量)。同步原语与线程管理密切相关。

`start_new_thread(functuion,args,kwargs=None)`–函数是 thread 模块的一个关键函数，其参数为：函数，函数的参数以及可选的关键字参数。前两个参数必须有。产生一个新的线程来运行这个函数。

`allocate_lock()`–分配一个LockType类型的锁对象
`exit()`–退出线程

LockType类型锁对象方法

方法	描述
`acquire(wait=None)`	尝试获取锁对象
`locked()`	如果获取了锁对象返回`True`,否则返回`False`
`release()`	释放锁

这一次使用thread提供的多线程机制，两个循环并发地执行(显然，时间短的那个先结束)。总运行时间为最慢的那个线程的运行时间，而不是所有的线程运行时间之和。

```python
import thread
from time import sleep,ctime

def loop0():
    print '\tstart loop 0 at:',ctime()
    sleep(4)
    print '\tloop 0 done at:',ctime()

def loop1():
    print '\tstart loop 1 at:',ctime()
    sleep(2)
    print '\tloop 1 done at:',ctime()

# 在单线程中顺序执行两个循环。一定要一个循环结束后，另一个才能开始。总时间是各个循环运行时间之和。
def main():
    print 'starting at:',ctime()
    loop0() # 运算结果1
    loop1() # 运算结果2
    print 'all done at:',ctime()

def mainthread():
    print 'thread starting at:',ctime()
    thread.start_new_thread(loop0,())
    thread.start_new_thread(loop1,())
    '''
    我们没有写让主线程停下来等所有子线程结束之后再继续运行的代码。这就是我们之前说线程
    需要同步的原因。在这里，我们使用了 sleep()函数做为我们的同步机制。

    写sleep(6)是因为,如果我们没有让主线程停下来，那主线程就会运行下一条语句，显示
    “all done”，然后就关闭运行着 loop0()和 loop1()的两个线程，退出了。
```

```
    如果我们的循环的执行时间不能事先确定的话，那怎么办呢？
    这可能造成主线程过早或过晚退出。这就是锁的用武之地了。
'''
    sleep(6)
    print 'thread all done at:',ctime()

# 在线程中加入锁机制
loops = [4,2] # 列表中的值为循环执行的时间

# nloop为循环的号码,second为sleep()时间,lock为锁对象
def loop(nloop, second, lock):
    print 'start loop',nloop,'at:',ctime()
    sleep(second)
    print 'loop',nloop,'done at:',ctime()
    lock.release() # 释放锁，通知主线程这个线程结束了

def mutexthread():
    print 'mutex thread starting at:',ctime()
    locks = []
    nloops = range(len(loops)) #n次循环

    # 分配n个锁
    for i in nloops:
        lock = thread.allocate_lock() # 分配一个锁
        lock.acquire() # 获取锁对象,表示“把锁锁上”。
        locks.append(lock)

    for i in nloops:
        thread.start_new_thread(loop,(i, loops[i], locks[i])) # 启动n个线程

    # 获取了锁对象，则locked返回True，达到暂停主线程的作用。
    # 直到两个锁都被解锁为止才继续运行。
```

```
    for i in nloops:
        while locks[i].locked:
            pass
    print 'mutex thread all done at:',ctime()


if __name__ == "__main__":
    #main()
    #mainthread()
    mutexthread()
```

锁解决线程之间同步操作的困境。
像上面使用`sleep()`函数来做线程同步操作是不靠谱的。如果程序的循环时间不确定的话,这就又肯造成主线程过早或过晚退出。锁就是为了解决这样的困境设计的。

thread小结

thread模块中使用多线程是主要使用`thread.start_new_thread(function,args,kwargs=None)`来启动一个新线程,解决线程之间的同步操作使用锁机制。分配一个锁使用`thread.allocate_lock()`方法，分配锁对象之后，为了实现线程之间的同步操作，需要调用锁对象的`acquire()`方法获取锁,`locked()`方法用于判断获取锁是否成功,如果获取了锁对象返回`True`,否则返回`False`。`release()`方法用于释放锁，以通知主线程该子进程结束了。

18.5 threadding模块

threading为一个更高级的module，提供了更好用的同步机制。

threading模块对象	描述

`Thread`	一个线程执行的对象
`Lock`	锁原语对象-与thread中的锁对象相同
`RLock`	可重入锁对象
`Coondition`	条件对象能让一个线程停下来， 等待其它线程满足了某个 “条件”。
`Event`	事件对象–多个线程可以等待某个事件的发生，在事件发生后，所有的线程都会被激活。
`Semaphore`	信号量同步机制
`BounderSemaphore`	与`Semaphore`类似,只是它不允许超过初始值
`Timer`	与`Thread`相似，只是，它要等待一段时间后才开始运行。

> 另一个避免使用 thread 模块的原因是，它不支持守护线程。而threading模块支持
>
> threading 模块支持守护线程， 它们是这样工作的： 守护线程一般是一个等待客户请求的服务器，如果没有客户提出请求，它就在那等着。
>
> 如果你的主线程要退出的时候，不用等待那些子线程完成，那就设定这些线程的 daemon 属性。即，在线程开始(调用Thread.start())之前， 调用setDaemon()函数设定线程的daemon标志(Thread.setDaemon(True))就表示这个线程“不重要”。如果你想要等待子线程完成再退出，那就什么都不用做，或者显式地调用Thread.setDaemon(False)以保证其daemon 标志为 False。 可以使用Thread.isDaemon()函数来判断其daemon标志的值。

18.5.1 Thread

用Thread类可以用多种方法创建线程。

- 创建一个Thread实例，传给他一个函数
- 创建一个Thread实例，传给他一个可调用的类对象
- 从Thread派生出一个子类，创建一个这个类的实例

Thread对象的函数

函数	描述
`start()`	开始线程的执行
`run()`	定义线程功能的函数(子类会重写)
`join(timeout=None)`	join the current thread–加入到主线程的含义. Wait until the thread terminates.
`setName(name)`	设置现成的名字
`getName()`	返回线程的名字
`isAlive()`	返回线程是否在运行中
`isDaemon()`	返回线程的daemon标志
`setDaemon(daemonic)`	把线程的 daemon 标志设为daemonic(一定要在调用 start()函数前调用)

> `join()`会等到线程结束，或者在给了 timeout 参数的时候，等到超时为止。
>
> `join()`的另一个比较重要的方面是它可以完全不用调用。一旦线程启动后，就会一直运行，直到线程的函数结束，退出为止。
>
> 如果你的主线程除了等线程结束外，还有其它的事情要做(如处理或等待其它的客户请求),那就不用调用`join()`,**只有在你要等待线程结束的时候才要调用 join()。**

创建一个Thread实例，传给它一个函数

```
loos = [4,2]

def loopthreading1(nloop, second):
    print '\tstart loop', nloop, 'at:',ctime()
    sleep(second)
    print '\tloop',nloop, 'done at:',ctime()

def mainthreading():
    print 'threading starting at:',ctime()
    threads = []
    nloops = range(len(loops)) #n次循环

    # 创建n个线程
    for i in nloops:
        lock = threading.Thread(target=loopthreading1,args=(i, loops[i]))  #b 并不立即开始线程
        threads.append(lock)

    for i in nloops:
        threads[i].start() # 启动线程

    # join()的作用是等待所有线程结束
    for i in nloops:
        threads[i].join() # thread to finish, join()会等到线程结束，或者在给了 timeout 参数的时候，等到超时为
止。

    print 'threading all done at:',ctime()
```

创建一个 Thread 的实例，传给它一个可调用的类对象

```
loops = [4,2]

class ThreadFunc(object):
    def __init__(self, func, args, name=''):
        self.name = name
        self.func = func
        self.args = args

    def __call__(self):
        apply(self.func,self.args)

def loopthreading2(nloop, second):
    print '\tstart loop', nloop, 'at:',ctime()
    sleep(second)
    print '\tloop',nloop, 'done at:',ctime()

def mainthreading2():
    print 'threading starting at:',ctime()
    threads = []
    nloops = range(len(loops))

    for i in nloops:
        t = threading.Thread(target=ThreadFunc(loopthreading2,(i,loops[i]),loopthreading2.__name__))
        threads.append(t)

    for i in nloops:
        threads[i].start()

    # wait for all
    for i in nloops:
        threads[i].join()
```

```
    print 'threading all done at:',ctime()
```

从Thread派生出一个子类,创建一个这个子类的实例

```
# 从 Thread 派生出一个子类，创建一个这个子类的实例
class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        super(MyThread,self).__init__(target=func, name=name, args=args) #调用父类Thread的初始化方法必须
要以关键字参数形式调用
        self.func = func
        self.args = args
        self.name = name

    def run(self):
        apply(self.func, self.args)

def loopthreading3(nloop, second):
    print '\tstart loop', nloop, 'at:',ctime()
    sleep(second)
    print '\tloop',nloop, 'done at:',ctime()


def mainthreading3():
    print 'threading starting at:',ctime()
    threads = []
    nloops = range(len(loops))

    for i in nloops:
        t = MyThread(loopthreading3,(i,loops[i]),loopthreading3.__name__)
        threads.append(t)

    for i in nloops:
        threads[i].start()

    # wait for all
```

```
    for i in nloops:
        threads[i].join()
    print 'threading all done at:',ctime()
```

18.5.2 斐波那契，阶乘和累加和

```
import threading
from time import sleep,ctime

loops = [4, 2]

class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def run(self):
        #print '\t\tstarting',self.name,'at:',ctime()
        self.res = apply(self.func, self.args)
        #print self.name,'\t\tfinished at:',ctime()

    def getResult(self):
        return self.res

def loop(nloop, second):
    print '\tstart loop',nloop,'at:',ctime()
    sleep(second)
    print '\tloop',nloop,'done at:',ctime()

def main():
    print 'starting at:',ctime()
    threads = []
    nloops = range(len(loops))

    for i in nloops:
```

```
        t = MyThread(loop,(i, loops[i]), loop.__name__)
        threads.append(t)

    for i in nloops:
        threads[i].start()

    for i in nloops:
        threads[i].join()

    print 'all done at:',ctime()

def fib(x):
    sleep(0.005)
    if x < 2:return 1
    return (fib(x-2) + fib(x-1))

def fac(x):
    sleep(0.1)
    if x < 2:return 1
    return (x*fac(x-1))

def sum(x):
    sleep(0.1)
    if x < 2:return 1
    return (x + sum(x-1))

funcs = [fib,fac,sum]
n = 12
```

```
def main2():
    nfuncs = range(len(funcs))

    print '*'*10,'SINGLE THREAD','*'*10
    for i in nfuncs:
        print '\tstarting [',funcs[i].__name__,'] at:',ctime()
        print '\rresult:',funcs[i](n)
        print '\t',funcs[i].__name__,'finished at:',ctime()

    print '*'*10,'MULTI THREADS','*'*10
    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i],(n,),funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        print '\tmultithread starting at:',ctime()
        threads[i].start()

    for i in nfuncs:
        threads[i].join()
        print '\tresult:',threads[i].getResult()
        print '\tmultithread finished at:',ctime()

    print 'all done'



if __name__ == "__main__":
    #main()
    main2()
```

18.5.3 threading模块中的其他函数

`activeCount()` –当前活动线程数量
`currentThread()` –返回当前线程对象
`enumerate()` –返回当前线程的列表
`settrace(func)` –为所有线程设置一个跟踪函数
`setprofile(func)` –为所有线程设置一个profile函数

18.5.4 生产者-消费者问题和 Queue 模块

常用函数
`queue(size)`, `qsize()`, `empty()`, `full()`, `put(item,block=0)`, `get(block=0)`

```
from random import randint
from time import sleep
from Queue import Queue

class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def run(self):
        #print '\t\tstarting',self.name,'at:',ctime()
        self.res = apply(self.func, self.args)
        #print self.name,'\t\tfinished at:',ctime()

    def getResult(self):
        return self.res


# writeQ和readQ一次往队列中放入/读取一个对象
def writeQ(queue):
    print '[producing] object for Q...',
    queue.put('xxx',1)
    print 'size now',queue.qsize()

def readQ(queue):
    val = queue.get(1)
    print '[consumed] object from Q...size now',queue.qsize()

# writer,reader函数总共循环loops次放入对象,一次放入一个对象
```

```
def writer(queue, loops):
    for i in range(loops):
        writeQ(queue)
        sleep(randint(1,3))

def reader(queue, loops):
    for i in range(loops):
        readQ(queue)
        sleep(randint(2,5))

funcs = [writer, reader]
nfuncs = range(len(funcs))

def productorconsumer():
    nloops = randint(2, 9) # 循环次数随机,即生产者放入对象的次数
    print 'loops=',nloops
    q = Queue(32) # Queue的大小

    threads = []
    # 产生两个线程
    for i in nfuncs:
        t = MyThread(funcs[i],(q,nloops),funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
        threads[i].join()

    print 'all done'
```

```
if __name__ == "__main__":
    #main()
    #main2()
    productorconsumer()
```

18.6 相关模块

`thread`, `threading`, `Queue`, `mutex`, `SocketServer`