3.3 Python 实现多线程

线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。
- 每个线程都有他自己的一组 CPU 寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的 CPU 寄存器的状态。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠）， 这就是线程的退让。

3.3.1 Python 的前后台线程

在 Python 中要启动一个线程，可以使用 threading 包中的 Thread 建立一个对象，这个 Thread 类的基本原型是：

t=Thread(target,args=None)

其中 target 是要执行的线程函数，args 是一个元组或者列表为 target 的函数提供参数，然后调用 t.start()就开始了线程。

例 3-3-1： 在主线程中启动一个子线程执行 reading 函数。

```
import threading
import time
import random

def reading():
    for i in range(10):
        print("reading",i)
        time.sleep(random.randint(1,2))

r=threading.Thread(target=reading)
r.setDaemon(False)
r.start()
print("The End")
```

程序结果：

```
reading 0
The End
reading 1
reading 2
reading 3
reading 4
```

从结果看到主线程启动子线程 r 后就结束了，但是子线程还没有结束，继续显示完 reasing 4 后才结束。其中的 r.setDaemon(False)就是设置线程 r 为后台线程，后台线程不因主线程的结束而结束。如何设置 r.setDaemon(True)，那么 r 就是前台线程。

例 3-3-2： 启动一个前台线程

```
import threading
import time
import random

def reading():
    for i in range(5):
        print("reading",i)
        time.sleep(random.randint(1,2))

r=threading.Thread(target=reading)
r.setDaemon(True)
r.start()
print("The End")
```

程序结果：

```
reading 0
The End
```

由此可见在主线程结束后子线程也结束，这就是前台线程。

例 3-3-3：前台与后台线程

```
import threading
import time
import random

def reading():
    for i in range(5):
        print("reading",i)
        time.sleep(random.randint(1,2))

def test():
    r=threading.Thread(target=reading)
    r.setDaemon(True)
    r.start()
    print("test end")

t=threading.Thread(target=test)
t.setDaemon(False)
t.start()
print("The End")
```

程序结果：

```
The End
```

reading 0
test end

由此可见主线程启动后台子线程 t 后就结束了，但是 t 还在执行，在 t 中启动前台 r 子线程，之后 t 结束，相应的 r 也结束。

3.3.2 线程的等待

在多线程的程序中往往一个线程（例如主线程）要等待其它线程执行完毕才继续执行，这可以用 join 函数，使用的方法是：

线程对象.join()

在一个线程代码中执行这条语句，当前的线程就会停止执行，一直等到指定的线程对象的线程执行完毕后才继续执行，即这条语句启动阻塞等待的作用。

例 3-3-4：主线程启动一个子线程并等待子线程结束后才继续执行。

```
import threading
import time
import random

def reading():
    for i in range(5):
        print("reading",i)
        time.sleep(random.randint(1,2))

t=threading.Thread(target=reading)
t.setDaemon(False)
t.start()
t.join()
print("The End")
```

程序结果：

```
reading 0
reading 1
reading 2
reading 3
reading 4
The End
```

由此可见主线程启动子线程 t 执行 reading 函数，t.join()就阻塞主线程，一直等到 t 线程执行完毕后才结束 t.join()，继续执行显示 The End。

例 3-3-5：在一个子线程启动另外一个子线程，并等待子线程结束后才继续执行。

```
import threading
import time
import random

def reading():
```

```
        for i in range(5):
            print("reading",i)
            time.sleep(random.randint(1,2))

    def test():
        r=threading.Thread(target=reading)
        r.setDaemon(True)
        r.start()
        r.join()
        print("test end")

    t=threading.Thread(target=test)
    t.setDaemon(False)
    t.start()
    t.join()
    print("The End")
```

程序结果：

```
reading 0
reading 1
reading 2
reading 3
reading 4
test end
The End
```

由此可见主线程启动 t 线程后 t.join()会等待 t 线程结束，在 test 中再次启动 r 子线程，而且 r.join()而阻塞 t 线程，等待 r 线程执行完毕后才结束 r.join()，然后显示 test end，之后 t 线程结束，再次结束 t.join()，主线程显示 The End 后结束。

3.3.3 多线程与资源

在多个线程的程序中一个普遍存在的问题是，如果多个线程要竞争同时访问与改写公共资源，那么应该怎么样协调各个线程的关系。一个普遍使用的方法是使用线程锁，Python 使用 threading.RLock 类来创建一个线程锁对象：

```
lock=threading.RLock()
```

这个对象 lock 有两个重要方法是 acquire()与 release()，当执行：

```
lock.acquire()
```

语句时强迫 lock 获取线程锁，如果已经有另外的线程先调用了 acquire()方法获取了线程锁而还没有调用 release()释放锁，那么这个 lock.acquire()就阻塞当前的线程，一直等待锁的控制权，直到别的线程释放锁后 lock.acquire()就获取锁并解除阻塞，线程继续执行，执行后线程要调用 lock.release()释放锁，不然别的线程会一直得不到锁的控制权。

使用 acquire /release 的工作机制我们可以把一段修改公共资源的代码用 acquire()与 release()夹起来，这样就保证一次最多只有一个线程在修改公共资源，别的线程如果也要修改就必须等待，直到本线程调用 release()释放锁后别的线程才能获取锁的控制权进行资源的修改。

例 3-3-6：一个子线程 A 把一个全局的列表 words 进行升序的排列，另外一个 D 线程把这个列表进行降序的排列。

```
import threading
import time

lock=threading._RLock()
words=["a","b","d","b","p","m","e","f","b"]

def increase():
    global words
    for count in range(5):
        lock.acquire()
        print("A acquired")
        for i in range(len(words)):
            for j in range(i+1,len(words)):
                if words[j]<words[i]:
                    t=words[i]
                    words[i]=words[j]
                    words[j]=t
        print("A ",words)
        time.sleep(1)
        lock.release()

def decrease():
    global words
    for count in range(5):
        lock.acquire()
        print("D acquired")
        for i in range(len(words)):
            for j in range(i+1,len(words)):
                if words[j]>words[i]:
                    t=words[i]
                    words[i]=words[j]
                    words[j]=t
        print("D ",words)
        time.sleep(1)
        lock.release()

A=threading.Thread(target=increase)
A.setDaemon(False)
A.start()

D=threading.Thread(target=decrease)
D.setDaemon(False)
```

```
D.start()
print("The End")
```

程序结果：

```
A acquired
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
The End
D acquired
D    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
D acquired
D    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
A acquired
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
A acquired
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
D acquired
D    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
D acquired
D    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
D acquired
D    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
A acquired
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
A acquired
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
```

由此可见无论是 increase 还是 decrease 的排序过程，都是在获得锁的控制权下进行的，因此排序过程中另外一个线程必然处于等待状态，不会干扰本次的排序，因此每次显示的结构不是升序的就是降序的。

如果我们不使用锁，那么在升序排序时降序排序也在工作，最后的结果既不是升序也不是降序，下面时不使用锁的一次结果：

```
The End
A    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
D    ['b', 'p', 'm', 'f', 'e', 'd', 'b', 'b', 'a']
D    ['a', 'b', 'e', 'p', 'm', 'f', 'd', 'b', 'b']
A    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
D    ['e', 'p', 'm', 'f', 'd', 'b', 'b', 'b', 'a']
D    ['a', 'b', 'p', 'm', 'f', 'e', 'd', 'b', 'b']
A    ['p', 'm', 'f', 'e', 'd', 'b', 'b', 'b', 'a']
D    ['f', 'p', 'm', 'e', 'd', 'b', 'b', 'b', 'a']
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
A    ['a', 'b', 'b', 'b', 'd', 'e', 'f', 'm', 'p']
```