

Python Talk 10



这个指引文档在 知识共享 署名-相同方式共享 3.0协议之条款下提供

This guidance is available under the Creative Commons Attribution-ShareAlike License

多线程编程

- ▶ 多线程编程可以让程序更加灵活
 - ▶ e.g. 一个网络服务器需要处理很多个客户端的请求，但每个请求之间没有很大的关联
- ▶ 多线程在某些情况下可以提升程序效率
 - ▶ e.g. 一个程序需要处理很多互相独立的图像，此时可以利用多个CPU的优势平行处理
- ▶ 多线程程序相对容易出错、难以调试
- ▶ 为了防止竞争条件，需要用锁、条件变量等进行控制
- ▶ Python的threading库支持多线程编程



计算一百万个1相加

► 单线程做法

```
s = 0
for i in range(1000000) :
    s += 1
print(s)
```

► 多线程做法

- 开10个线程，共享变量s；每个线程进行100000次加法



多线程程序

```
import time, threading

s = 0

class MyThread(threading.Thread) :
    def run(self) :
        global s
        for i in range(100000) :
            s += 1

if __name__ == '__main__' :
    for i in range(10) :
        MyThread().start()
    time.sleep(3)
    print(s)
```

- ▶ import: 引入threading库
- ▶ threading.Thread: 线程的基类
 - ▶ run()为线程的执行内容
 - ▶ 调用start()以开始线程
 - ▶ 如果调用run()就不是多线程调用了
- ▶ global表示引用全局变量
- ▶ 通过time.sleep(3)等待所有线程结束
- ▶ 问题: 得到的结果是1000000吗?



竞争条件

- ▶ 看似简单的 `s += 1` 其实包含多个指令

```
load  register,      memory[0x1234]
add   register,      1
store memory[0x1234], register
```

- ▶ 翻译成Python就是

```
reg = mem[0x1234]    # load
reg += 1              # execute
mem[0x1234] = reg     # store
```

- ▶ CPU可能在任何两个指令中间暂停执行，并切换到另一个线程



竞争条件

- ▶ 思考如下左右两个线程的执行先后顺序，时间从上到下

# Thread 1	# Thread 2	
reg1 = memory[]		# mem[] = 0
	reg2 = memory[]	# reg1 = 1
reg1 += 1		# reg2 = 1
	reg2 += 1	# reg1 = 2
mem[] = reg1		# reg2 = 2
	mem[] = reg2	# mem[] = 2
		# mem[] = 2

- ▶ 竞争条件（**race condition**）的定义：两个线程同时访问一个内存地址，其中至少一个访问是写入
- ▶ 竞争条件有很多解决方法
 - ▶ 其中最常用的是通过锁控制访问



锁

- ▶ 锁（Lock）是一种特殊变量，每个线程需要使用某些资源时获取锁，在用完后释放锁
- ▶ 通过各种硬件和软件机制，任何时刻只能有最多一个线程获得锁
 - ▶ 如果线程尝试获取被占用的锁，一直等待到锁被释放

```
lock = threading.Lock() # 创建锁  
lock.acquire()           # 获取锁  
lock.release()           # 释放锁
```



锁示例

```
import time, threading

s = 0
s_lock = threading.Lock()

class MyThread(threading.Thread) :
    def run(self) :
        global s, s_lock
        for i in range(100000) :
            s_lock.acquire()
            s += 1
            s_lock.release()

if __name__ == '__main__' :
    for i in range(10) :
        MyThread().start()
    time.sleep(5)
    print(s)
```

- ▶ 锁s_lock用于管理对s的访问
- ▶ 访问s之前用acquire获取锁
- ▶ 访问s之后用release获取锁
- ▶ 程序的运算时间有所增长，因此sleep从3秒变为5秒
 - ▶ 因为当另一个线程在访问s时，当前线程需要等待
 - ▶ 如何避免使用sleep?
 - ▶ 真实的程序中无法预测线程执行时间



条件变量

- ▶ 实现线程之间的信号传递
- ▶ 问题：线程A希望读取变量X，但是需要在线程B写入变量之后进行。X的访问通过X_lock进行限制，通过X_ready标明是否已经写入。
- ▶ 实现1
 - ▶ 线程A通过一个循环，不断获取X_lock并通过X_ready判断是否可以读取X
 - ▶ 问题：如果X一直没有被写入，A会进行不停的无用计算
- ▶ 实现2
 - ▶ A的每次循环后用time.sleep等待一个很短的时间
 - ▶ 问题：虽然等待很短，但还是有延迟
- ▶ 实现3
 - ▶ 线程B在写入后“通知”线程A



条件变量

- ▶ 条件变量允许线程传递“某个事件发生”的信息
- ▶ 条件变量需要和锁共同使用，当一个线程使用条件变量时必须已获得对应的锁

```
cv = threading.Condition(lock)
lock.acquire()           # 必须先获取lock才能使用cv
cv.wait()                 # 等待一个事件的发生
cv.notify()              # 通知一个等待的线程
cv.notify_all()          # 通知所有等待的线程
lock.release()
```

- ▶ `cv.wait()` 返回时不保证是被`notify`的
 - ▶ 因此应该用一个`while`循环重新判断是否等待
- ▶ `cv.wait()` 时会释放`lock`，否则会阻塞其它线程



条件变量示例

- ▶ 线程A读取X，线程B写入X

```
X = 0
X_lock = threading.Lock()
X_ready = False
X_cv = threading.Condition(X_lock)
class ThreadA(threading.Thread) :
    def run(self) :
        X_lock.acquire()
        while not X_ready :
            X_cv.wait()
        print(X)
        X_lock.release()
class ThreadB(threading.Thread) :
    def run(self) :
        X_lock.acquire()
        X = 1234
        X_cv.notify()
        X_lock.release()
```



with

- ▶ Python中with有多种用途，在锁中可以用于获取一个锁

```
lock = threading.Lock()
with lock :
    # 锁被获取
    # Do something
# 锁被释放
```

- ▶ 等价于

```
lock = threading.Lock()
lock.acquire()
# 锁被获取
# Do something
lock.release()
# 锁被释放
```



条件变量示例2

- ▶ 让主线程等待所有线程退出

```
r = 0
r_lock = threading.Lock()
r_cv = threading.Condition(r_lock)
```

```
class MyThread(threading.Thread) :
    def run(self) :
        global r, r_lock, r_cv
        ...
        with r_lock :
            r -= 1
            r_cv.notify()
```

```
if __name__ == '__main__' :
    for i in range(10) :
        with r_lock :
            r += 1
        MyThread().start()
    with r_lock :
        while r :
```



练习

1. 实现一个线程安全的队列

- ▶ 队列的长度限制为 n
- ▶ 对于`enqueue(elem)`操作，如果队列不到 n 个元素，将`elem`插入队列；如果队列已经有 n 个元素，阻塞至队列长度变小
- ▶ 对于`dequeue()`操作，如果队列中有元素，返回队列中最早被插入的元素；否则阻塞至队列长度变大

2. 实现一个主要是读的读写锁

- ▶ 读写锁允许很多个读锁定线程，但是不可以多个写阻塞线程
- ▶ 读和写锁定不能同时存在
- ▶ 读锁定和解锁应该尽可能被优化
- ▶ 写不能被长时间阻塞（当没有读时写不能被阻塞）
- ▶ 方法: `read_acquire()`, `read_release()`, `write_acquire()`, `write_release()`



参考资料

- ▶ Operating Systems - Principles & Practice, Second Edition, by Thomas Anderson and Mike Dahlin
- ▶ <https://docs.python.org/3/library/threading.html>



感谢参加此次活动

