线程同步

概念

线程同步，线程间协同，通过某种技术，让一个线程访问某些数据时，其他线程不能访问这些数据，直到该线程完成对数据的操作。

Event ***

Event事件，是线程间通信机制中最简单的实现，使用一个内部的标记flag，通过flag的True或False的变化来进行操作。

名称	含义
set()	标记设置为True
clear()	标记设置为False
is_set()	标记是否为True
wait(timeout=None)	设置等待标记为True的时长，None为无限等待。等到返回True，未等到超时了返回False

练习

老板雇佣了一个工人，让他生产杯子，老板一直等着这个工人，直到生产了10个杯子

```
# 下面的代码是否能够完成功能？
from threading import Event, Thread
import logging
import time

FORMAT = '%(asctime)s %(threadName)s %(thread)s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []
flag = False

def boss():
    logging.info("I'm boss, waiting for U")
    while True:
        time.sleep(1)
        if flag:
            break
    logging.info('Good Job.')

def worker(count=10):
    logging.info('I am working for U')

    while True:
        logging.info('make 1 cup')
        time.sleep(0.5)
        cups.append(1)

```

```
        if len(cups) >= count:
            flag = True
            break
    logging.info('I finished my job. cups={}'.format(cups))

b = Thread(target=boss, name='boss')
w = Thread(target=worker, name='worker')
b.start()
w.start()
```

上面代码基本能够完成，但上面代码问题有：

- bug，应该将worker中的flag定义为global就可解决
- 老板一直要不停的查询worker的状态变化

```
from threading import Event, Thread
import logging
import time

FORMAT = '%(asctime)s %(threadName)s %(thread)s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)


def boss(event:Event):
    logging.info("I'm boss, waiting for U")
    event.wait() # 阻塞等待
    logging.info('Good Job.')

def worker(event:Event, count=10):
    logging.info('I am working for U')
    cups = []
    while True:
        logging.info('make 1 cup')
        time.sleep(0.5)
        cups.append(1)

        if len(cups) >= count:
            event.set()
            break
    logging.info('I finished my job. cups={}'.format(cups))


event = Event()
b = Thread(target=boss, name='boss', args=(event,))
w = Thread(target=worker, name='worker', args=(event,))
b.start()
w.start()
```

总结

需要使用同一个Event对象的标记flag。
谁wait就是等到flag变为True，或等到超时返回False。
不限制等待者的个数，通知所有等待者。

wait的使用

```python
# 修改上例worker中的 while 条件
def worker(event:Event, count=10):
    logging.info('I am working for U')
    cups = []
    while not event.wait(0.5): # 使用wait阻塞等待
        logging.info('make 1 cup')
        cups.append(1)

        if len(cups) >= count:
            event.set()
            #break # 为什么可以注释break呢?
    logging.info('I finished my job. cups={}'.format(cups))
```

Lock ***

- Lock类是mutex互斥锁
- 一旦一个线程获得锁，其它**试图获取锁的线程**将被阻塞，只到拥有锁的线程释放锁
- 凡是存在共享资源争抢的地方都可以使用锁，从而保证只有一个使用者可以完全使用这个资源。

名称	含义
acquire(blocking=True, timeout=-1)	默认阻塞，阻塞可以设置超时时间。非阻塞时，timeout禁止设置。 成功获取锁，返回True，否则返回False
release()	释放锁。可以从任何线程调用释放。 已上锁的锁，会被重置为unlocked 未上锁的锁上调用，抛RuntimeError异常。

锁的基本使用

```python
import threading
import time

lock = threading.Lock() # 互斥mutex

lock.acquire()
print('-' * 30)


def worker(l):
    print('worker start', threading.current_thread())
    l.acquire()
    print('worker done', threading.current_thread())

for i in range(10):
    threading.Thread(target=worker, name="w{}".format(i),
                     args=(lock,), daemon=True).start()

print('-' * 30)

```

```
while True:
    cmd = input(">>>")
    if cmd == 'r': # 按r后枚举所有线程看看
        lock.release()
        print('released one locker')
    elif cmd == 'quit':
        lock.release()
        break
    else:
        print(threading.enumerate())
        print(lock.locked())
```

上例可以看出不管在哪一个线程中，只要对一个已经上锁的锁发起阻塞地请求，该线程就会阻塞。

练习

订单要求生产1000个杯子，组织10个工人生产。请忽略老板，关注工人生成杯子

```
import threading
from threading import Thread, Lock
import time
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []

def worker(count=1000):
    logging.info("I'm working")
    while True:
        if len(cups) >= count:
            break
        time.sleep(0.0001) # 为了看出线程切换效果，模拟杯子制作时间
        cups.append(1)
    logging.info('I finished my job. cups = {}'.format(len(cups)))

for i in range(1, 11):
    t = Thread(target=worker, name="w{}".format(i), args=(1000,))
    t.start()
```

从上例的运行结果看出，多线程调度，导致了判断失效，多生产了杯子。

如何修改解决这个问题？加锁

上例使用锁实现如下：

```
import threading
from threading import Thread, Lock
import time
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)
```

```
8
9  cups = []
10 lock = Lock() # 锁
11
12 def worker(count=1000):
13     logging.info("I'm working")
14     while True:
15         lock.acquire() # 获取锁
16
17         if len(cups) >= count:
18             #lock.release() # 1
19             break
20         #lock.release()     # 2
21         time.sleep(0.0001) # 为了看出线程切换效果，模拟杯子制作时间
22         cups.append(1)
23         lock.release()     # 3
24     logging.info('I finished my job. cups = {}'.format(len(cups)))
25
26 for i in range(1, 11):
27     t = Thread(target=worker, name="w{}".format(i), args=(1000,))
28     t.start()
```

锁分析

位置2分析

- 假设某一个瞬间，有一个工作线程A获取了锁，len(cups)正好有999个，然后就释放了锁，可以继续执行下面的语句，生产一个杯子，这地方不阻塞，但是正好杯子也没有生产完。锁释放后，其他线程就可以获得锁，线程B获得了锁，发现len(cups)也是999个，然后释放锁，然后也可以去生产一个杯子。锁释放后，其他的线程也可能获得锁。就说A和B线程都认为是999个，都会生产一个杯子，那么实际上最后一定会超出1000个。
- 假设某个瞬间一个线程获得锁，然后发现杯子到了1000个，没有释放锁就直接break了，由于其他线程还在阻塞等待锁释放，这就成了**死锁**了。

位置3分析

- 获得锁的线程发现是999，有资格生产杯子，生产一个，释放锁，看似很完美
- 问题在于，获取锁的线程发现杯子有1000个，直接break，没释放锁离开了，死锁了

位置1分析

- 如果线程获得锁，发现是1000，break前释放锁，没问题
- 问题在于，A线程获得锁后，发现小于1000，继续执行，其他线程获得锁全部阻塞。A线程再次执行循环后，自己也阻塞了。死锁了。

问题：究竟怎样加锁才正确呢？

要在位置1和位置3同时加release。

上下文支持

锁是典型必须释放的，Python提供了上下文支持。查看Lock类的上下文方法，__enter__方法返回bool表示是否获得锁，__exit__方法中释放锁。

由此上例可以修改为

```python
import threading
from threading import Thread, Lock
import time
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

cups = []
lock = Lock() # 锁

def worker(count=1000):
    logging.info("I'm working")
    while True:
        with lock: # 获取锁，离开with释放锁
            if len(cups) >= count:
                logging.info('leaving')
                break
            time.sleep(0.0001) # 为了看出线程切换效果，模拟杯子制作时间
            cups.append(1)
            logging.info(lock.locked())

    logging.info('I finished my job. cups = {}'.format(len(cups)))

for i in range(1, 11):
    t = Thread(target=worker, name="w{}".format(i), args=(1000,))
    t.start()
```

感觉一下正确得到结果了吗？感觉到了执行速度了吗？慢了还是快了，为什么？

锁的应用场景

锁适用于访问和修改同一个共享资源的时候，即读写同一个资源的时候。

如果全部都是读取同一个共享资源需要锁吗?
不需要。因为这时可以认为共享资源是不可变的，每一次读取它都是一样的值，所以不用加锁

使用锁的注意事项：

- 少用锁，必要时用锁。使用了锁，多线程访问被锁的资源时，就成了串行，要么排队执行，要么争抢执行
    - 举例，高速公路上车并行跑，可是到了省界只开放了一个收费口，过了这个口，车辆依然可以在多车道上一起跑。过收费口的时候，如果排队一辆辆过，加不加锁一样效率相当，但是一旦出现争抢，就必须加锁一辆辆过。注意，不管加不加锁，只要是一辆辆过，效率就下降了。
- 加锁时间越短越好，不需要就立即释放锁
- 一定要避免死锁

不使用锁，有了效率，但是结果是错的。
使用了锁，效率低下，但是结果是对的。
所以，我们是为了效率要错误结果呢？还是为了对的结果，让计算机去计算吧

Queue的线程安全

标准库queue模块，提供FIFO的Queue、LIFO的队列、优先队列。
Queue类是线程安全的，适用于同一进程内多线程间安全的交换数据。内部使用了Lock和Condition。

特别注意下面的代码在多线程中使用

```
import queue

q = queue.Queue(8)

if q.qsize() == 7:
    q.put() # 上下两句可能被打断

if q.qsize() == 1:
    q.get() # 未必会成功

```

如果不加锁，是不可能获得准确的大小的，因为你刚读取到了一个大小，还没有取走数据，就有可能被其他线程改了。
Queue类的size虽然加了锁，但是，依然不能保证立即get、put就能成功，因为读取大小和get、put方法是分开的。