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。

不同操作系统实现技术有所不同，有临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件Event等

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

需求：
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

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

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

def boss(event:Event):
    logging.info("I'm boss, waiting for U.")
    # 等待
```

```
    event.wait()
    logging.info("Good Job.")


def worker(event:Event, count=10):
    logging.info("I'm working for U.")
    cups = []
    while True:
        logging.info('make 1')
        time.sleep(0.5)
        cups.append(1)
        if len(cups) >= count:
            # 通知
            event.set()
            break
    logging.info('I finished my job. cups={}'.format(cups))

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

总结
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。不限制等待的个数。

wait的使用

```
from threading import Event, Thread
import logging
logging.basicConfig(level=logging.INFO)

def do(event:Event, interval:int):
    while not event.wait(interval): # 条件中使用，返回True或者False
        logging.info('do sth.')

e = Event()
Thread(target=do, args=(e, 3)).start()

e.wait(10) # 也可以使用time.sleep(10)
```

```
e.set()
print('main exit')
```

Event的wait优于time.sleep，它会更快的切换到其它线程，提高并发效率。

Event练习

实现Timer，延时执行的线程
延时计算add(x,y)

思路
Timer的构造函数中参数得有哪些？
如何实现start启动一个线程执行函数
如何cancel取消待执行任务

思路实现

```
from threading import Event, Thread
import logging
logging.basicConfig(level=logging.INFO)

def add(x:int, y:int):
    logging.info(x + y)

class Timer:
    def __init__(self, interval, fn, *args, **kwargs):
        pass

    def start(self):
        pass

    def cancel(self):
        pass
```

完整实现

```
from threading import Event, Thread
import datetime
import logging
logging.basicConfig(level=logging.INFO)
```

```
def add(x:int, y:int):
    logging.info(x + y)

class Timer:
    def __init__(self, interval, fn, *args, **kwargs):
        self.interval = interval
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.event = Event()

    def start(self):
        Thread(target=self.__run).start()

    def cancel(self):
        self.event.set()

    def __run(self):
        start = datetime.datetime.now()
        logging.info('waiting')

        self.event.wait(self.interval)
        if not self.event.is_set():
            self.fn(*self.args, **self.kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        logging.info('finished {}'.format(delta))
        self.event.set()


t = Timer(10, add, 4, 50)
t.start()
e = Event()
e.wait(4)
#t.cancel()
print('==========')
```

或者

```
class Timer:
    def __init__(self, interval, fn, *args, **kwargs):
```

```
        self.interval = interval
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.event = Event()

    def start(self):
        Thread(target=self.__run).start()

    def cancel(self):
        self.event.set()

    def __run(self):
        start = datetime.datetime.now()
        logging.info('waiting')

        if not self.event.wait(self.interval):
            self.fn(*self.args, **self.kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        logging.info('finished {}'.format(delta))

        self.event.set()
```


Lock ***

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。

需求：
订单要求生产1000个杯子，组织10个工人生产

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

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

```
cups = []

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

cups = []

def worker(count=10):
    logging.info("I'm working for U.")
    while len(cups) < count:
        time.sleep(0.0001) # 为了看出线程切换效果
        cups.append(1)
    logging.info('I finished. cups = {}'.format(len(cups)))


for _ in range(10):
    Thread(target=worker, args=(1000,)).start()

# 运行结果
2017-12-13 17:30:43,800 Thread-1 11256 I'm working for U.
2017-12-13 17:30:43,801 Thread-2 9660 I'm working for U.
2017-12-13 17:30:43,801 Thread-3 11824 I'm working for U.
2017-12-13 17:30:43,801 Thread-4 12236 I'm working for U.
2017-12-13 17:30:43,801 Thread-5 11528 I'm working for U.
2017-12-13 17:30:43,801 Thread-6 12308 I'm working for U.
2017-12-13 17:30:43,801 Thread-7 12376 I'm working for U.
2017-12-13 17:30:43,802 Thread-8 12140 I'm working for U.
2017-12-13 17:30:43,802 Thread-9 11988 I'm working for U.
2017-12-13 17:30:43,802 Thread-10 13272 I'm working for U.
2017-12-13 17:30:43,903 Thread-1 11256 I finished. cups = 1007
2017-12-13 17:30:43,903 Thread-4 12236 I finished. cups = 1008
2017-12-13 17:30:43,903 Thread-8 12140 I finished. cups = 1008
2017-12-13 17:30:43,904 Thread-2 9660 I finished. cups = 1008
2017-12-13 17:30:43,904 Thread-9 11988 I finished. cups = 1008
2017-12-13 17:30:43,904 Thread-7 12376 I finished. cups = 1008
2017-12-13 17:30:43,904 Thread-10 13272 I finished. cups = 1008
2017-12-13 17:30:43,904 Thread-5 11528 I finished. cups = 1009
2017-12-13 17:30:43,905 Thread-3 11824 I finished. cups = 1009
2017-12-13 17:30:43,905 Thread-6 12308 I finished. cups = 1009
```

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？加锁

Lock
锁，一旦线程获得锁，其它试图获取锁的线程将被阻塞

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。

上例的锁的实现

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

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

cups = []
lock = Lock()

def worker(count=10):
    logging.info("I'm working for U.")
    flag = False
    while True:
        lock.acquire() # 获取锁

        if len(cups) >= count:
            flag = True
        # lock.release() # 1 这里释放锁?
        time.sleep(0.0001) # 为了看出线程切换效果
        if not flag:
            cups.append(1)
        # lock.release() # 2 这里释放锁?
        if flag:
            break
```

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

for _ in range(10):
    Thread(target=worker, args=(1000,)).start()
```

思考
上面代码中，共有2处可以释放锁。请问，放在何处合适？

假设位置1的lock.release()合适，分析如下：
有一个时刻，在某一个线程中len(cups)正好是999，flag=True，释放锁，正好线程被打断。另一个线程判断发现也是999，flag=True，可能线程被打断。可能另外一个线程也判断是999，flag也设置为True。这三个线程只要继续执行到cups.append(1)，一定会导致cups的长度超过1000的。
假设位置2的lock.release()合适，分析如下：
在某一个时刻len(cups)，正好是999，flag=True，其它线程试图访问这段代码的线程都阻塞获取不到锁，直到当前线程安全的增加了一个数据，然后释放锁。其它线程有一个抢到锁，但发现已经1000了，只好break打印退出。再其它线程都一样，发现已经1000了，都退出了。
所以位置2 释放锁 是正确的。
但是我们发现锁保证了数据完整性，但是性能下降很多。

上例中if flag: break是为了保证release方法被执行，否则，就出现了死锁，得到锁的永远没有释放锁。

计数器类，可以加、可以减。

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

class Counter:
    def __init__(self):
        self._val = 0

    @property
    def value(self):
        return self._val

    def inc(self):
        self._val += 1

    def dec(self):
        self._val -= 1
```

```
def run(c:Counter, count=100):
    for _ in range(count):
        for i in range(-50,50):
            if i < 0:
                c.dec()
            else:
                c.inc()

c = Counter()
c1 = 10 # 线程数
c2 = 10
for i in range(c1):
    Thread(target=run, args=(c, c2)).start()

print(c.value)
```

c1取10、100、1000看看
c2取10、100、1000看看

self._val += 1或self._val -= 1在线程中执行的时候，有可能被打断。
要加锁。怎么加？

加锁、解锁

一般来说，加锁就需要解锁，但是加锁后解锁前，还要有一些代码执行，就有可能抛异常，一旦出现异常，锁是无法释放，但是当前线程可能因为这个异常被终止了，这就产生了死锁。

加锁、解锁常用语句：
1、使用try...finally语句保证锁的释放
2、with上下文管理，锁对象支持上下文管理

改造Couter类，如下

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

class Counter:
    def __init__(self):
        self._val = 0
```

```
        self.__lock = Lock()

    @property
    def value(self):
        with self.__lock:
            return self._val

    def inc(self):
        try:
            self.__lock.acquire()
            self._val += 1
        finally:
            self.__lock.release()

    def dec(self):
        with self.__lock:
            self._val -= 1

def run(c:Counter, count=100):
    for _ in range(count):
        for i in range(-50,50):
            if i < 0:
                c.dec()
            else:
                c.inc()

c = Counter()
c1 = 10 # 线程数
c2 = 1000
for i in range(c1):
    Thread(target=run, args=(c, c2)).start()

print(c.value) # 这一句合适吗?
```

最后一句修改如下

```
while True:
    time.sleep(1)
    if threading.active_count() == 1:
        print(threading.enumerate())
```

```
            print(c.value)
            break
        else:
            print(threading.enumerate())
```

`print(c.value)` 这一句在主线程中，很早就执行了。退出条件是，只剩下主线程的时候。

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。
- 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

非阻塞锁使用

```
import threading
import logging
import time

FORMAT = '%(asctime)-15s\t [%(threadName)s, %(thread)8d] %(message)s'
logging.basicConfig(level=logging.INFO, format=FORMAT)

def worker(tasks):
    for task in tasks:
        time.sleep(0.001)
        if task.lock.acquire(False): # 获取锁则返回True
```

```
            logging.info('{} {} begin to start'.format(threading.current_thread(),
task.name))
            #  适当的时机释放锁，为了演示不释放
        else:
            logging.info('{} {} is working'.format(threading.current_thread(), task
.name))

class Task:
    def __init__(self, name):
        self.name = name
        self.lock = threading.Lock()

# 构造10个任务
tasks = [Task('task-{}'.format(x)) for x in range(10)]

# 启动5个线程
for i in range(5):
    threading.Thread(target=worker, name='worker-{}'.format(i), args=(tasks,)).star
t()
```

可重入锁RLock

可重入锁，是**线程相关**的锁。
线程A获得可重复锁，并可以多次成功获取，不会阻塞。最后要在线程A中做和acquire次数相同的release。

```
import threading
import time

lock = threading.RLock()
print(lock.acquire())
print('------------')
print(lock.acquire(blocking=False))
print(lock.acquire())
print(lock.acquire(timeout=3.55))
print(lock.acquire(blocking=False))
#print(lock.acquire(blocking=False, timeout=10)) # 异常
lock.release()
```

```
lock.release()
lock.release()
lock.release()
print('main thread {}'.format(threading.current_thread().ident))
print("lock in main thread {}".format(lock)) # 注意观察lock对象的信息
lock.release()
#lock.release() #多了一次
print('===============')
print()

print(lock.acquire(blocking=False)) # 1次
#threading.Timer(3, lambda x:x.release(), args=(lock,)).start() # 跨线程了，异常
lock.release()
print('~~~~~~~~~~~~~~~~~~')
print()

# 测试多线程
print(lock.acquire())
def sub(l):
    print('{}: {}'.format(threading.current_thread(), l.acquire())) # 阻塞
    print('{}: {}'.format(threading.current_thread(), l.acquire(False)))
    print('lock in sub thread {}'.format(lock))
    l.release()
    print('sub 1')
    l.release()
    print('sub 2')
    # l.release() # 多了一次

threading.Timer(2, sub, args=(lock,)).start() # 传入同一个lock对象
print('++++++++++++++++++++++')
print()

print(lock.acquire())
lock.release()
time.sleep(5)
print('释放主线程锁')
lock.release()
```

可重入锁，与线程相关，可在一个线程中获取锁，并可继续在同一线程中不阻塞获取锁。当锁未释放完，其它线程获取锁就会阻塞，直到当前持有锁的线程释放完锁。

Condition

构造方法Condition(lock=None)，可以传入一个Lock或RLock对象，默认是RLock。

名称	含义
acquire(*args)	获取锁
wait(self, timeout=None)	等待或超时
notify(n=1)	唤醒至多指定数目个数的等待的线程，没有等待的线程就没有任何操作
notify_all()	唤醒所有等待的线程

Condition用于生产者、消费者模型，为了解决生产者消费者速度匹配问题。

先看一个例子，消费者消费速度大于生产者生产速度

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

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

## 此例只是为了演示，不考虑线程安全问题

class Dispatcher:
    def __init__(self):
        self.data = None
        self.event = Event() # event只是为了使用方便，与逻辑无关

    def produce(self, total):
        for _ in range(total):
            data = random.randint(0,100)
            logging.info(data)
            self.data = data
            self.event.wait(1)
        self.event.set()
```

```
    def consume(self):
        while not self.event.is_set():
            data = self.data
            logging.info("recieved {}".format(data))
            self.data = None
            self.event.wait(0.5)

d = Dispatcher()
p = Thread(target=d.produce, args=(10,), name='producer')
c = Thread(target=d.consume, name='consumer')
c.start()
p.start()
```

这个例子采用了消费者主动消费，消费者浪费了大量时间，主动来查看有没有数据。
能否换成一种通知机制，有数据通知消费者来消费呢？
使用Condition对象。

```
from threading import Thread, Event, Condition
import logging
import random

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

## 此例只是为了演示，不考虑线程安全问题

class Dispatcher:
    def __init__(self):
        self.data = None
        self.event = Event() # event只是为了使用方便，与逻辑无关
        self.cond = Condition()

    def produce(self, total):
        for _ in range(total):
            data = random.randint(0,100)
            with self.cond:
                logging.info(data)
                self.data = data
                self.cond.notify_all()
            self.event.wait(1) # 模拟产生数据速度
```

```
        self.event.set()

    def consume(self):
        while not self.event.is_set():
            with self.cond:
                self.cond.wait() # 阻塞等通知
                logging.info("received {}".format(self.data))
                self.data = None
            self.event.wait(0.5) # 模拟消费的速度

d = Dispatcher()
p = Thread(target=d.produce, args=(10,), name='producer')
c = Thread(target=d.consume, name='consumer')
c.start()
p.start()
```

上例中，消费者等待数据等待，如果生产者准备好了会通知消费者消费，省得消费者反复来查看数据是否就绪。
如果是1个生产者，多个消费者怎么改？

```
from threading import Thread, Event, Condition
import logging
import random

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

## 此例只是为了演示，不考虑线程安全问题

class Dispatcher:
    def __init__(self):
        self.data = None
        self.event = Event() # event只是为了使用方便，与逻辑无关
        self.cond = Condition()

    def produce(self, total):
        for _ in range(total):
            data = random.randint(0,100)
            with self.cond:
                logging.info(data)
```

```
                self.data = data
                self.cond.notify_all()
            self.event.wait(1) # 模拟产生数据速度
        self.event.set()

    def consume(self):
        while not self.event.is_set():
            with self.cond:
                self.cond.wait() # 阻塞等通知
                logging.info("received {}".format(self.data))
            self.event.wait(0.5) # 模拟消费的速度

d = Dispatcher()
p = Thread(target=d.produce, args=(10,), name='producer')
# 增加消费者
for i in range(5):
    c = Thread(target=d.consume, name='consumer-{}'.format(i))
    c.start()
p.start()
```

self.cond.notify_all() # 发通知
修改为
self.cond.notify(n=2)
试一试看看结果？
这个例子，可以看到实现了消息的 `一对多` ，这其实就是 **广播** 模式。

注：上例中，程序本身不是线程安全的，程序逻辑有很多瑕疵，但是可以很好的帮助理解Condition的使用，和生产者消费者模型。

Condition总结

Condition用于生产者消费者模型中，解决生产者消费者速度匹配的问题。
采用了通知机制，非常有效率。

使用方式
使用Condition，必须先acquire，用完了要release，因为内部使用了锁，默认使用RLock锁，最好的方式是使用with上下文。
消费者wait，等待通知。
生产者生产好消息，对消费者发通知，可以使用notify或者notify_all方法。