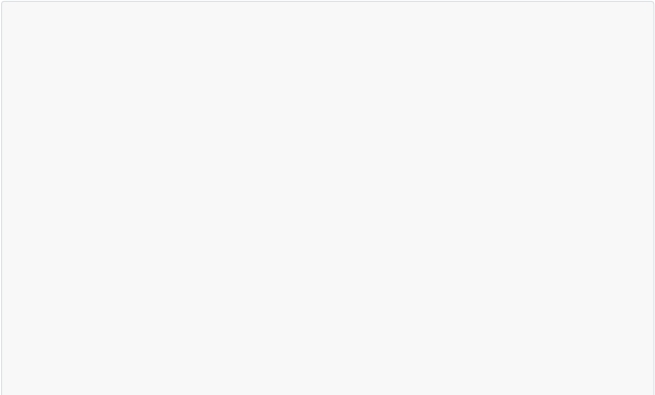
线程同步

|  |  |
| --- | --- |
| 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 |



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。

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)s  %(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  am  working  for  U')

cups  =  []

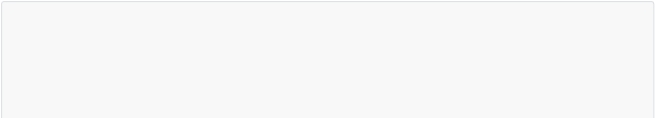
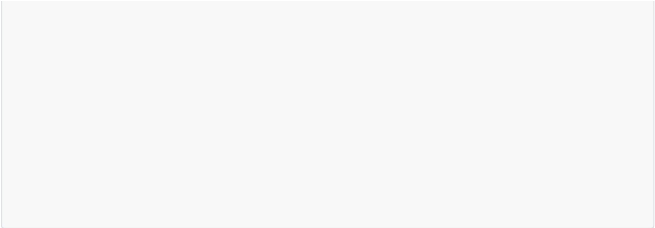
while  True:

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

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

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)s  %(message)s'

logging.basicConfig(format=FORMAT,  level=logging.INFO)

def  worker(event:Event,  interval:int):

while  not  event.wait(interval):  #  条件

logging.info('do  sth.')

e  =  Event()

Thread(target=worker,  args=(e,  3)).start()

e.wait(10)  #  等待

e.set()

print('====end====')

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。请忽略老板，关注工人生成杯子

from  threading  import  Thread,  Lock

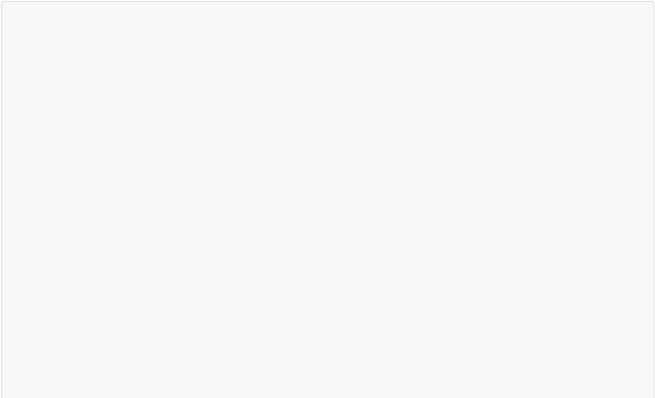
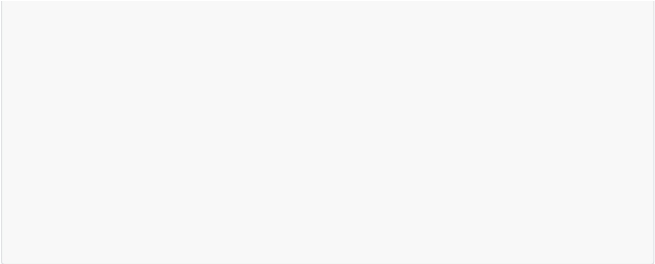
import  logging

import  time

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)s  %(message)s'

logging.basicConfig(format=FORMAT,  level=logging.INFO)

|  |  |
| --- | --- |
| 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。 |



cups  =  []

def  worker(count=10):

logging.info("I'm  working.")

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  my  job.  cups  =  {}'.format(len(cups)))

for  i  in  range(1,  11):

t  =  Thread(target=worker,  name="worker-{}".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？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

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

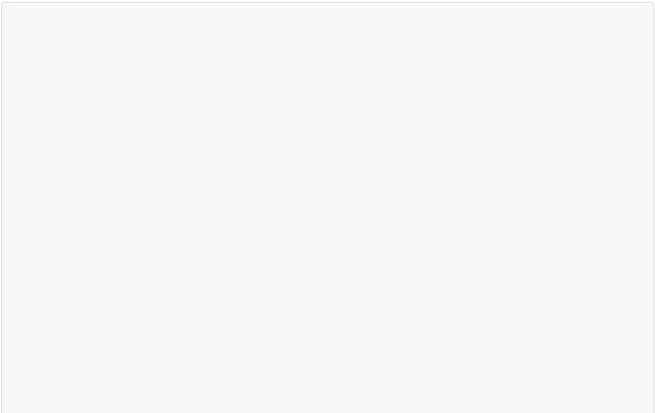
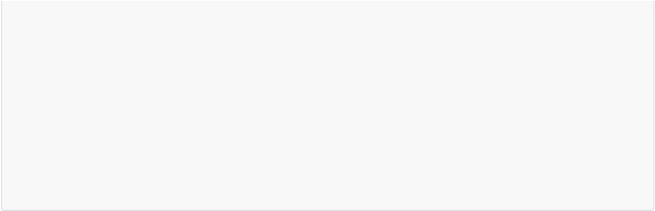
if  len(cups)  >=  count:

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

#  lock.release()  #  3  这里释放锁？

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3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。

上例中位置3，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

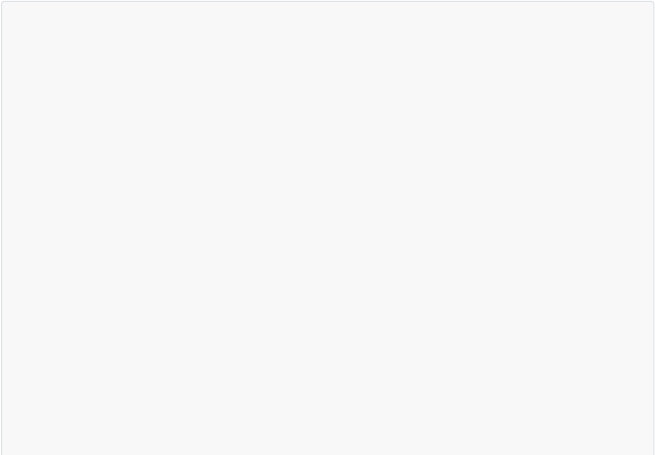
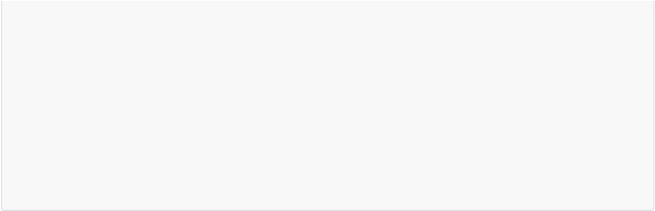
def  run(c:Counter,  count=100):

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:

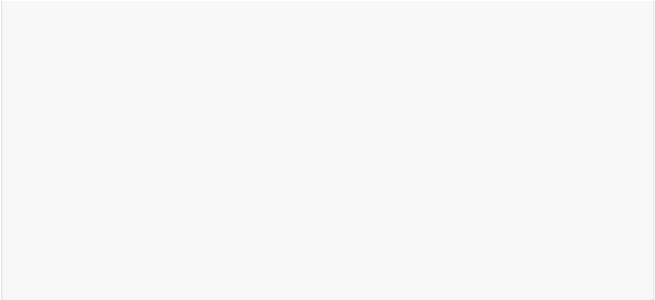
self.\_\_lock.release()

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。

这样的改造后，代码可以保证最后得到的value值一定是0。

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

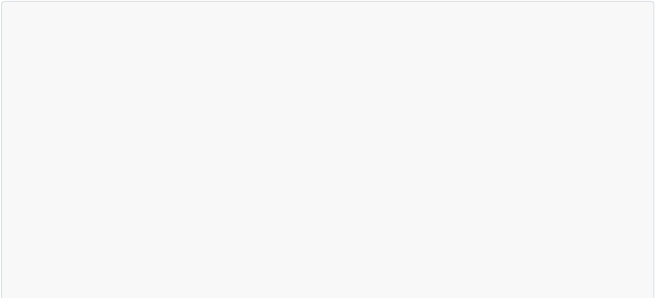
不使用锁，有了效率，但是结果是错的。

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

非阻塞锁使用

import  threading



import  logging

import  time

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)-10d  %(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):  #  不等待锁，但是还获取成功了

logging.info('{}.  {}  begin  to  work.'.format(threading.current\_thread().name,

task.name))

#  适当的时机释放锁，为了演示不释放

else:  #获取锁失败了，说明有线程已经获取了

logging.info('{}.  {}  is  working.'.format(threading.current\_thread().name,

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,)).start()

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  logging

import  threading

import  time

FORMAT  =  '%(asctime)s  %(threadName)s  %s(thread)  %(message)s'

logging.basicConfig(format="",  level=logging.INFO)

lock  =  threading.RLock()

print(lock.acquire())

print('-'  \*  30)

print(lock.acquire(blocking=False))

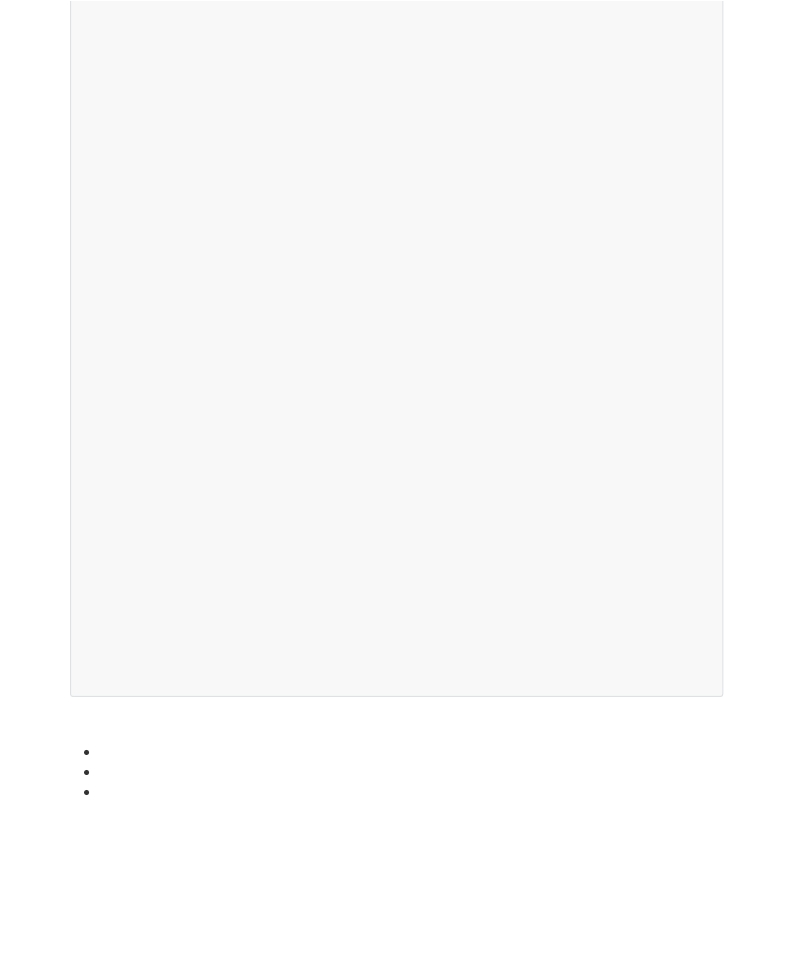
print(lock.acquire())

print(lock.acquire(timeout=3))

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.main\_thread().ident))

print('lock  in  main  thread  {}'.format(lock))

lock.release()

#lock.release()  #  release多了抛异常

print('-'  \*  30)

#  主线程获取锁

print(lock.acquire(blocking=False))  #  count  =  1

#  Thread(target=lambda  l:  l.release(),  args=(lock,)).start()  #  跨线程了，异常

lock.release()

print('-'  \*  30)

#  测试主线程

print(lock.acquire())  #  count  =  1

def  sub(l):

print('{}:  {}'.format(threading.current\_thread(),  l.acquire()))  #  阻塞

print('{}:  {}'.format(threading.current\_thread(),  l.acquire()))

print('lock  in  sub  thread  {}'.format(lock))

l.release()

print('release  in  sub  1')

l.release()

print('release  in  sub  2')

#l.relese()  #  不能多释放

print('+'  \*  30)

threading.Timer(2,  sub,  (lock,)).start()  #  为另一个线程传入同一个lock对象

print('in  main  thread,  {}'.format(lock.acquire()))  #  count  =  2

lock.release()

time.sleep(5)

print('release  lock  in  main  thread~~~~~~~~',  end='\n\n')

lock.release()  #  count  =  0

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多次获取锁

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

锁都应该使用完后释放。可重入锁也是锁，应该acquire多少次，就release多少次

Condition

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

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。

|  |  |
| --- | --- |
| 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 |

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  Event,  Thread,  Condition

import  logging

import  random

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)s  %(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  Dispachter:

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(1,  100)

logging.info(data)

self.data  =  data

self.event.wait(1)  #  模拟生产数据需要耗时1秒

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)  #  模拟消费速度

d  =  Dispachter()

p  =  Thread(target=d.produce,  name='producer',  args=(10,))

c  =  Thread(target=d.consume,  name='consumer')

c.start()

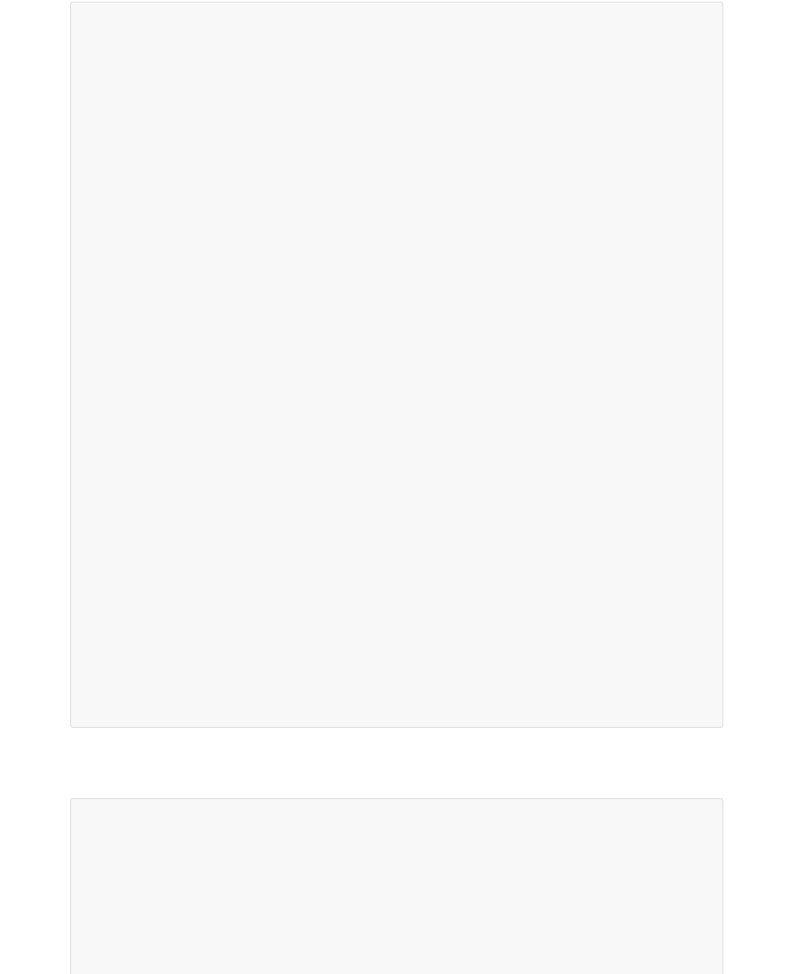
p.start()

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  Event,  Thread,  Condition



import  logging

import  random

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)s  %(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  Dispachter:

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(1,  100)

with  self.cond:

logging.info(data)

self.data  =  data

self.cond.notify\_all()

self.event.wait(1)  #  模拟生产数据需要耗时1秒

def  consume(self):

while  not  self.event.is\_set():

with  self.cond:

self.cond.wait()

data  =  self.data

logging.info('recieved  {}'.format(data))

#self.data  =  None

#self.event.wait(0.5)  #  模拟消费速度

d  =  Dispachter()

p  =  Thread(target=d.produce,  name='producer',  args=(10,))

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  Event,  Thread,  Condition

import  logging

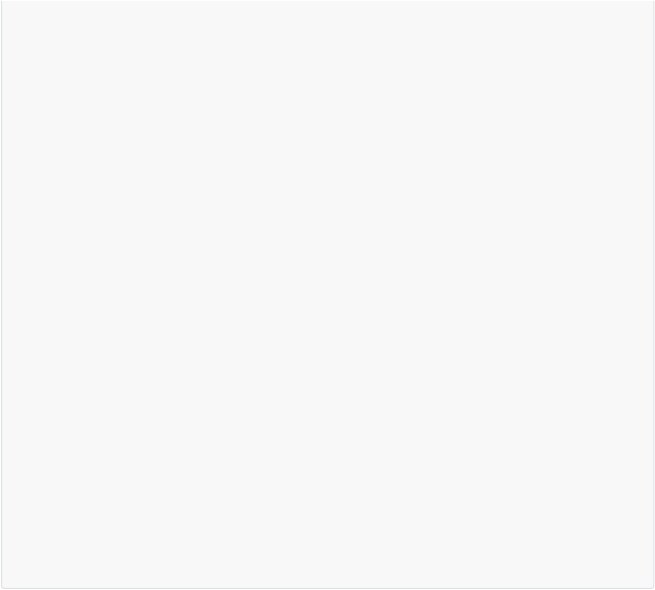
import  random

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)s  %(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  Dispachter:



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(1,  100)

with  self.cond:

logging.info(data)

self.data  =  data

self.cond.notify\_all()

self.event.wait(1)  #  模拟生产数据需要耗时1秒

def  consume(self):

while  not  self.event.is\_set():

with  self.cond:

self.cond.wait()

data  =  self.data

logging.info('recieved  {}'.format(data))

#self.data  =  None

#self.event.wait(0.5)  #  模拟消费速度

d  =  Dispachter()

p  =  Thread(target=d.produce,  name='producer',  args=(10,))

#  增加消费者

for  i  in  range(5):

c  =  Thread(target=d.consume,  name='consumer')

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。

semaphore 信号量

|  |  |
| --- | --- |
| 名称 | 含义 |
| Semaphore(value=1) | 构造方法。value小于0，抛ValueError异常 |
| acquire(blocking=True, timeout=None) | 获取信号量，计数器减1，获取成功返回True |
| release() | 释放信号量，计数器加1 |



和Lock很像，信号量对象内部维护一个倒计数器，每一次acquire都会减1，当acquire方法发现计数为0就阻塞请求

的线程，直到其它线程对信号量release后，计数大于0，恢复阻塞的线程。

计数器永远不会低于0，因为acquire的时候，发现是0，都会被阻塞。

from  threading  import  Thread,  Semaphore

import  logging

import  time

FORMAT  =  '%(asctime)s  %(threadName)-12s  %(thread)-8s  %(message)s'

logging.basicConfig(format=FORMAT,  level=logging.INFO)

def  worker(s:Semaphore):

logging.info("in  worker  thread")

logging.info(s.acquire())

logging.info('worker  thread  over')

#  信号量

s  =  Semaphore(3)

logging.info(s.acquire())

print(s.\_value)

logging.info(s.acquire())

print(s.\_value)

logging.info(s.acquire())

print(s.\_value)

Thread(target=worker,  args=(s,)).start()

time.sleep(2)

logging.info(s.acquire(False))

logging.info(s.acquire(timeout=3))

#  释放一个

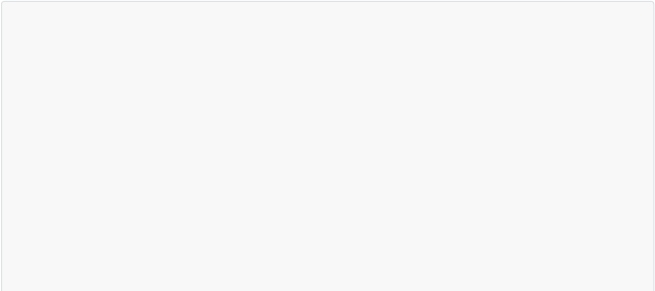
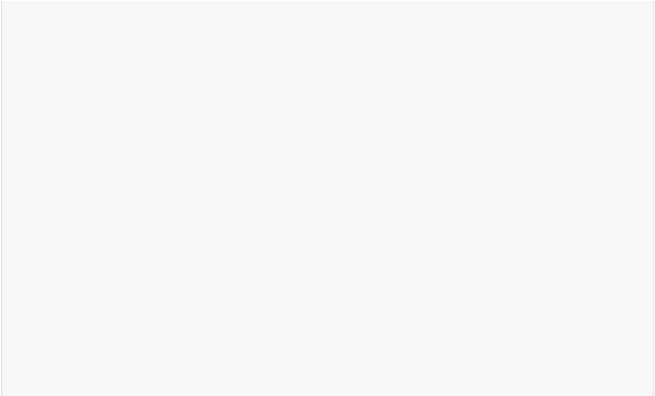
logging.info('release  one')

s.release()

release方法超界问题

假设如果还没有acquire信号量，就release，会怎么样？

import  logging



import  threading

sema  =  threading.Semaphore(3)

logging.warning(sema.\_\_dict\_\_)

for  i  in  range(3):

sema.acquire()

logging.warning('~~~~~~~~')

logging.warning(sema.\_\_dict\_\_)

for  i  in  range(4):

sema.release()

logging.warning(sema.\_\_dict\_\_)

for  i  in  range(3):

sema.acquire()

logging.warning('~~~~~~~~')

logging.warning(sema.\_\_dict\_\_)

sema.acquire()

logging.warning('~~~~~~~~')

logging.warning(sema.\_\_dict\_\_)

从上例输出结果可以看出，竟然内置计数器达到了4，这样实际上超出我们的最大值，需要解决这个问题。

BoundedSemaphore类

有界的信号量，不允许使用release超出初始值的范围，否则，抛出ValueError异常。

将上例的信号量改成有界的信号量试一试。

应用举例

连接池

因为资源有限，且开启一个连接成本高，所以，使用连接池。

一个简单的连接池

连接池应该有容量（总数），有一个工厂方法可以获取连接，能够把不用的连接返回，供其他调用者使用。

class  Conn:

def  \_\_init\_\_(self,  name):

self.name  =  name

class  Pool:

def  \_\_init\_\_(self,  count:int):

self.count  =  count

#  池中提前放着连接备用

self.pool  =  [self.\_connect('conn-{}'.format(i))  for  i  in  range(self.count)]

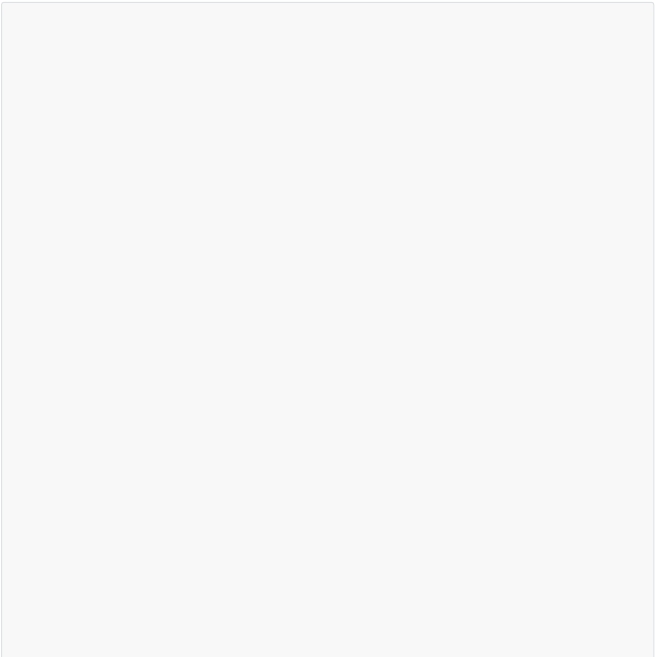
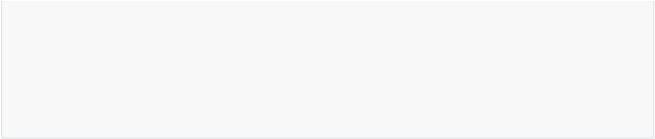
def  \_connect(self,  conn\_name):

#  创建连接的方法，返回一个连接对象

return  Conn(conn\_name)

def  get\_conn(self):

#  从池中拿走一个连接



if  len(self.pool)  >  0:

return  self.pool.pop()

def  return\_conn(self,  conn:Conn):

#  向池中返回一个连接对象

self.pool.append(conn)

真正的连接池的实现比上面的例子要复杂的多，这里只是简单的一个功能的实现。

本例中，get\_conn()方法在多线程的时候有线程安全问题。

假设池中正好有一个连接，有可能多个线程判断池的长度是大于0的，当一个线程拿走了连接对象，其他线程再来

pop就会抛异常的。如何解决？

1、加锁，在读写的地方加锁

2、使用信号量Semaphore

使用信号量对上例进行修改

import  random

import  threading

import  logging

import  time

FORMAT  =  '%(asctime)s  %(threadName)s  %(thread)-8d  %(message)s'

logging.basicConfig(level=logging.INFO,  format=FORMAT)

class  Conn:

def  \_\_init\_\_(self,  name):

self.name  =  name

class  Pool:

def  \_\_init\_\_(self,  count:int):

self.count  =  count

#  池中提前放着连接备用

self.pool  =  [self.\_connect('conn-{}'.format(i))  for  i  in  range(self.count)]

self.semaphore  =  threading.Semaphore(count)

def  \_connect(self,  conn\_name):

#  创建连接的方法，返回一个连接对象

return  Conn(conn\_name)

def  get\_conn(self):

#  从池中拿走一个连接

logging.info('get~~~~~~~~~~~~~')

self.semaphore.acquire()

logging.info('-------------------------')

return  self.pool.pop()

def  return\_conn(self,  conn:Conn):

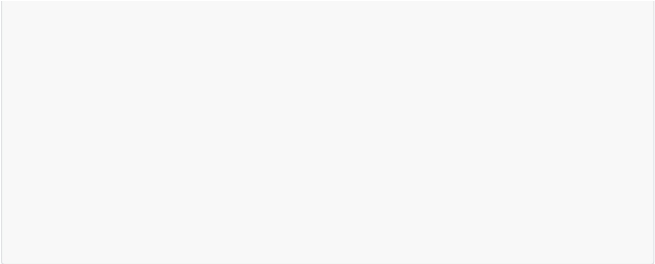
#  向池中返回一个连接对象

logging.info('return~~~~~~~~~~~~~')

self.pool.append(conn)

self.semaphore.release()

#  初始化连接池



pool  =  Pool(3)

def  worker(pool:Pool):

conn  =  pool.get\_conn()

logging.info(conn)

#  模拟使用了一段时间

time.sleep(random.randint(1,  5))

pool.return\_conn(conn)

for  i  in  range(6):

threading.Thread(target=worker,  name='worker-{}'.format(i),  args=(pool,)).start()

上例中，使用信号量解决资源有限的问题。

如果池中有资源，请求者获取资源时信号量减1，拿走资源。当请求超过资源数，请求者只能等待。当使用者用完

归还资源后信号量加1，等待线程就可以被唤醒拿走资源。

注意：这个连接池的例子不能用到生成环境，只是为了说明信号量使用的例子，连接池还有很多未完成功能。

问题

self.conns.append(conn) 这一句有哪些问题考虑？

1、边界问题分析

return\_conn方法可以单独执行，有可能多归还连接，也就是会多release，所以，要用有界信号量

BoundedSemaphore类。

这样用有界信号量修改源代码，保证如果多return\_conn就会抛异常。

self.pool.append(conn)

self.semaphore.release()

假设一种极端情况，计数器还差1就归还满了，有三个线程A、B、C都执行了第一句，都没有来得及release，这时

候轮到线程A release，正常的release，然后轮到线程C先release，一定出问题，超界了，直接抛异常。

因此信号量，可以保证，一定不能多归还。

如果归还了同一个连接多次怎么办，重复很容易判断。

这个程序还不能判断这些连接是不是原来自己创建的，这不是生成环境用的代码，只是简单演示。

2、正常使用分析

正常使用信号量，都会先获取信号量，然后用完归还。

创建很多线程，都去获取信号量，没有获得信号量的线程都阻塞。能归还的线程都是前面获取到信号量的线程，其

他没有获得线程都阻塞着。非阻塞的线程append后才release，这时候等待的线程被唤醒，才能pop，也就是没有

获取信号量就不能pop，这是安全的。

经过上面的分析，信号量比计算列表长度好，线程安全。

信号量和锁

信号量，可以多个线程访问共享资源，但这个共享资源数量有限。

锁，可以看做特殊的信号量，即信号量计数器初值为1。只允许同一个时间一个线程独占资源。