线程同步

Barrier

有人翻译成栅栏，我建议使用 `屏障` ，可以想象成路障、道闸。

3.2引入Python的新功能。

名称	含义
Barrier(parties, action=None, timeout=None)	构建Barrier对象，指定参与方数目。timeout是wait方法未指定超时的默认值
n_waiting	当前在屏障中等待的线程数
parties	各方数，就是需要多少个等待
wait(timeout=None)	等待通过屏障。返回0到线程数-1的整数，每个线程返回不同。如果wait方法设置了超时，并超时发送，屏障将处于broken状态

Barrier实例

```
import threading
import logging

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


def worker(barrier:threading.Barrier):
    logging.info('waiting for {} threads.'.format(barrier.n_waiting))
    try:
        barrier_id = barrier.wait()
        logging.info('after barrier {}'.format(barrier_id))
    except threading.BrokenBarrierError:
        logging.info('Broken Barrier')
```

```
barrier = threading.Barrier(3)

for x in range(3): # 改成4、5、6试一试
    threading.Thread(target=worker,name='worker-{}'.format(x), args=(barrier,)).sta
rt()

logging.info('started')

# 运行结果
2017-12-13 23:21:27,912     [worker-0,    10276] waiting for 0 threads.
2017-12-13 23:21:27,912     [worker-1,     2712] waiting for 1 threads.
2017-12-13 23:21:27,912     [worker-2,    12096] waiting for 2 threads.
2017-12-13 23:21:27,912     [worker-2,    12096] after barrier 2
2017-12-13 23:21:27,912     [worker-0,    10276] after barrier 0
2017-12-13 23:21:27,912     [MainThread,     3296] started
2017-12-13 23:21:27,912     [worker-1,     2712] after barrier 1
```

从运行结果看出：
所有线程冲到了Barrier前等待，直到到达parties的数目，屏障打开，所有线程停止等待，继续执行。
再有线程wait，屏障就绪等到到达参数方数目。
举例，赛马比赛所有马匹就位，开闸。下一批马匹陆续来到闸门前等待比赛。

名称	含义
broken	如果屏障处于打破的状态，返回True
abort()	将屏障置于broken状态，等待中的线程或者调用等待方法的线程中都会抛出BrokenBarrierError异常，直到reset方法来恢复屏障
reset()	恢复屏障，重新开始拦截

```
import threading
import logging

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


def worker(barrier:threading.Barrier):
```

```
    logging.info('waiting for {} threads.'.format(barrier.n_waiting))
    try:
        barrier_id = barrier.wait()
        logging.info('after barrier {}'.format(barrier_id))
    except threading.BrokenBarrierError:
        logging.info('Broken Barrier. run.')


barrier = threading.Barrier(3)

for x in range(0, 9):
    if x == 2:
        barrier.abort()
    elif x == 6:
        barrier.reset()
    threading.Event().wait(1)
    threading.Thread(target=worker,name='worker-{}'.format(x), args=(barrier,)).sta
rt()

# 运行结果
2017-12-13 23:36:27,507     [worker-0,     2324] waiting for 0 threads.
2017-12-13 23:36:28,519     [worker-1,    13180] waiting for 1 threads.
2017-12-13 23:36:28,520     [worker-0,     2324] Broken Barrier. run.
2017-12-13 23:36:28,521     [worker-1,    13180] Broken Barrier. run.
2017-12-13 23:36:29,534     [worker-2,    10320] waiting for 0 threads.
2017-12-13 23:36:29,535     [worker-2,    10320] Broken Barrier. run.
2017-12-13 23:36:30,547     [worker-3,    10136] waiting for 0 threads.
2017-12-13 23:36:30,548     [worker-3,    10136] Broken Barrier. run.
2017-12-13 23:36:31,562     [worker-4,     7032] waiting for 0 threads.
2017-12-13 23:36:31,563     [worker-4,     7032] Broken Barrier. run.
2017-12-13 23:36:32,575     [worker-5,     9256] waiting for 0 threads.
2017-12-13 23:36:32,575     [worker-5,     9256] Broken Barrier. run.
2017-12-13 23:36:33,590     [worker-6,    10300] waiting for 0 threads.
2017-12-13 23:36:34,603     [worker-7,     7988] waiting for 1 threads.
2017-12-13 23:36:35,618     [worker-8,     7492] waiting for 2 threads.
2017-12-13 23:36:35,619     [worker-8,     7492] after barrier 2
2017-12-13 23:36:35,619     [worker-6,    10300] after barrier 0
2017-12-13 23:36:35,620     [worker-7,     7988] after barrier 1
```

上例中，屏障中等待了2个，屏障就被break了，waiting的线程抛了BrokenBarrierError异常，新wait

的线程也抛异常，直到屏障恢复，才继续按照parties数目要求继续拦截线程。

wait方法超时实例

如果wait方法超时发生，屏障将处于broken状态，直到reset

```
import threading
import logging

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


def worker(barrier:threading.Barrier, i:int):
    logging.info('waiting for {} threads.'.format(barrier.n_waiting))
    try:
        logging.info(barrier.broken) # 是否broken
        if i<3:
            barrier_id = barrier.wait(1) # 超时后，屏障broken
        else:
            if i == 6:
                barrier.reset() # 恢复屏障
            barrier_id = barrier.wait()
        logging.info('after barrier {}'.format(barrier_id))
    except threading.BrokenBarrierError:
        logging.info('Broken Barrier. run.')


barrier = threading.Barrier(3)

for x in range(0, 9):
    threading.Event().wait(2)
    threading.Thread(target=worker,name='worker-{}'.format(x), args=(barrier, x)).s
tart()
```

Barrier应用

并发初始化

所有线程都必须初始化完成后，才能继续工作，例如运行前加载数据、检查，如果这些工作没完成，就开始运行，将不能正常工作。
10个线程做10种工作准备，每个线程负责一种工作，只有这10个线程都完成后，才能继续工作，先完成的要等待后完成的线程。
例如，启动一个程序，需要先加载磁盘文件、缓存预热、初始化连接池等工作，这些工作可以齐头并进，不过只有都满足了，程序才能继续向后执行。假设数据库连接失败，则初始化工作失败，就要abort，barrier置为broken，所有线程收到异常退出。

工作量
有10个计算任务，完成6个，就算工作完成。

---

semaphore 信号量

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。

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

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。

```
import threading
import logging
import time

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

def worker(s:threading.Semaphore):
    logging.info('in sub thread')
    logging.info(s.acquire()) # 阻塞
    logging.info('sub thread over')

# 信号量
```

```
s = threading.Semaphore(3)
logging.info(s.acquire())
print(s._value)
logging.info(s.acquire())
print(s._value)
logging.info(s.acquire())
print(s._value)

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

time.sleep(2)

logging.info(s.acquire(False))
logging.info(s.acquire(timeout=3)) # 阻塞3秒

# 释放
logging.info('released')
s.release()
```

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
        # 池中是连接对象的列表
        self.pool = [self._connect("conn-{}".format(x)) for x in range(self.count)]

    def _connect(self, conn_name):
        # 创建连接的方法，返回一个名称
```

```
        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):
        # 向池中添加一个连接
        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 threading
import logging
import random

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


class Conn:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return self.name

class Pool:
    def __init__(self, count:int):
        self.count = count
        # 池中是连接对象的列表
        self.pool = [self._connect("conn-{}".format(x)) for x in range(count)]
        self.semaphore = threading.Semaphore(count) # threading.Semaphore()
```

```
    def _connect(self, conn_name):
        # 返回一个名称
        return Conn(conn_name)

    def get_conn(self):
        # 从池中拿走一个连接
        print('------------')
        self.semaphore.acquire()
        print('===============')
        conn = self.pool.pop()
        return conn

    def return_conn(self, conn:Conn):
        # 向池中添加一个连接
        self.pool.append(conn)
        self.semaphore.release()


# 连接池初始化
pool = Pool(3)

def worker(pool:Pool):
    conn = pool.get_conn()
    logging.info(conn)
    # 模拟使用了一段时间
    threading.Event().wait(random.randint(1,4))
    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：这个例子不能用到生成环境，只是为了说明信号量使用的例子，还有很多未完成功能。

问题

self.conns.append(conn) 这一句要不要加锁？

1、从程序逻辑上分析

1.1 假设如果还没有使用信号量，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。

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。
保证了多归还连接抛出异常。

如果归还了同一个连接多次怎么办，去重很容易判断出来。

1.2 如果使用了信号量，但是还没有用完

```
self.pool.append(conn)
self.semaphore.release()
```

假设一种极端情况，计数器还差1就满了，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。

1.3 很多线程用完了信号量

没有获得信号量的线程都阻塞，没有线程和归还的线程争抢，当append后才release，这时候才能等待的线程被唤醒，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

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

数据结构和GIL

Queue
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，适用于多线程间安全的交换数据。内部使用了Lock和Condition。

为什么讲魔术方法时，说实现容器的大小，不准确？
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，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。

```
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
```

GIL全局解释器锁

CPython 在解释器进程级别有一把锁，叫做GIL 全局解释器锁。
GIL 保证CPython进程中，只有一个线程执行字节码。甚至是在多核CPU的情况下，也是如此。

CPython中
IO密集型，由于线程阻塞，就会调度其他线程；
CPU密集型，当前线程可能会连续的获得GIL，导致其它线程几乎无法使用CPU。
在CPython中由于有GIL存在，IO密集型，使用多线程；CPU密集型，使用多进程，绕开GIL。

新版CPython正在努力优化GIL的问题，但不是移除。
如果非要使用多线程的效率问题，请绕行，选择其它语言erlang、Go等。

> Python中绝大多数内置数据结构的读写都是原子操作。
> 由于GIL的存在，Python的内置数据类型在多线程编程的时候就变成了安全的了，但是实际上它们本身不是线程安全类型的。

保留GIL的原因：
Guido坚持的简单哲学，对于初学者门槛低，不需要高深的系统知识也能安全、简单的使用Python。
而且移除GIL，会降低CPython单线程的执行效率。

测试下面2个程序

```
import logging
import datetime

logging.basicConfig(level=logging.INFO, format="%(thread)s %(message)s")
start = datetime.datetime.now()

# 计算
def calc():
    sum = 0
    for _ in range(1000000000):
        sum += 1

calc()
```

```
calc()
calc()
calc()
calc()

delta = (datetime.datetime.now() - start).total_seconds()
logging.info(delta)
```

```
import threading
import logging
import datetime

logging.basicConfig(level=logging.INFO, format="%(thread)s %(message)s")
start = datetime.datetime.now()

# 计算
def calc():
    sum = 0
    for _ in range(1000000000):
        sum += 1

t1 = threading.Thread(target=calc)
t2 = threading.Thread(target=calc)
t3 = threading.Thread(target=calc)
t4 = threading.Thread(target=calc)
t5 = threading.Thread(target=calc)
t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
t1.join()
t2.join()
t3.join()
t4.join()
t5.join()


delta = (datetime.datetime.now() - start).total_seconds()
```

```
logging.info(delta)
```

从两段程序测试的结果来看，CPython中多线程根本没有任何优势，和一个线程执行时间相当。因为GIL的存在，尤其是像上面的计算密集型程序。