1. threading模块介绍

multiprocess模块的完全模仿了threading模块的接口，二者在使用层面，有很大的相似性
具体见multiprocess

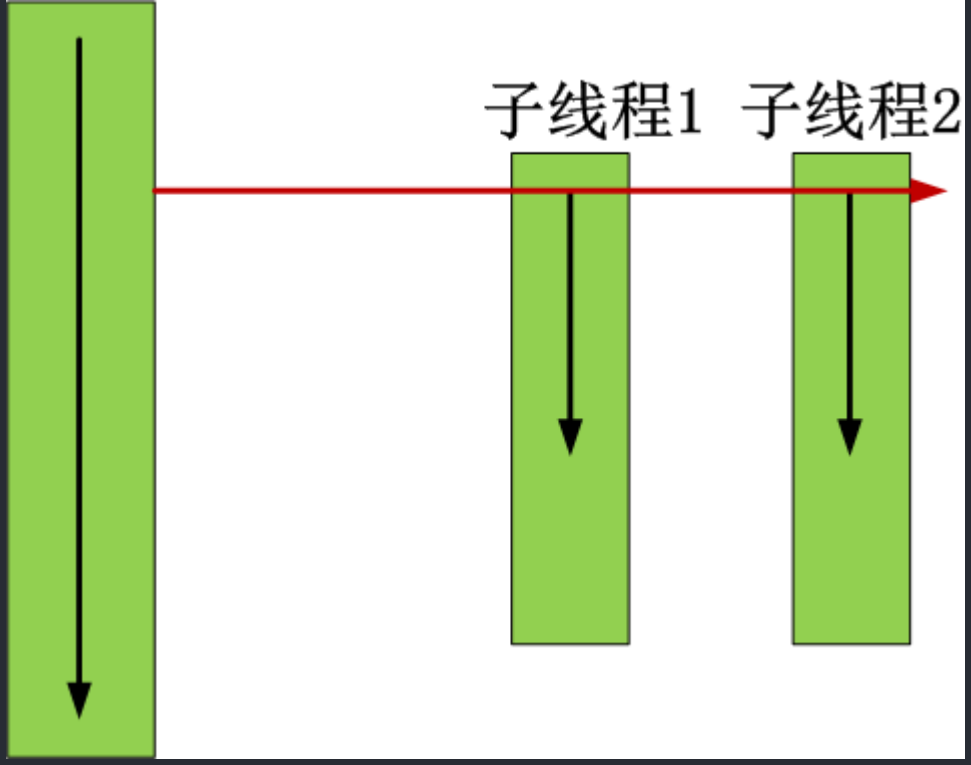
2. 开启线程的两种方式

1. Thread类调用开始线程

```python
from threading import Thread
import time

def foo(name):
    time.sleep(2)
    print('%s say hello'%name)

if __name__ == '__main__':
    s=Thread(target=foo,args=('egon',))
    s.start()
    print('主线程')
# 主线程
# egon say hello
```

2. Thread类继承开始线程

```python
from threading import Thread
import time

class Foo(Thread):

    def __init__(self, name):
        super().__init__()
        self.name=name

    def run(self):
        time.sleep(2)
        print('%s say hello'% self.name)

if __name__ == '__main__':
    f=Foo('egon')
    f.start()
    print('主线程')

# 主线程
# egon say hello
```



3. 多线程实现socket服务

1. 服务端

```python
from threading import Thread
import socket

sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
ip_port=('127.0.0.1',9000)
BUFSIZE=1024

sock.bind(ip_port)
sock.listen(5)

def action(conn):
    while True:
        try:
            date=conn.recv(BUFSIZE)
            print(date.decode('utf-8'))
            conn.send(date.upper())
        except Exception:
            break


if __name__ == '__main__':

    while True:
        conn,addr=sock.accept()
        p=Thread(target=action,args=(conn,))
        p.start()
```

2. 客户端

```python
import socket

sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
ip_port=('127.0.0.1',9000)
BUFSIZE=1024

sock.connect(ip_port)

while True:
    msg=input('>>>: ').strip()
    if not msg:continue
    sock.send(msg.encode('utf-8'))
    msg_recv=sock.recv(BUFSIZE)
    print(msg_recv.decode('utf-8'))
```

4. 线程相关的其他方法

1. Thread实例对象的方法

```
  isAlive(): 返回线程是否活动的。
  getName(): 返回线程名。
  setName(): 设置线程名。

threading模块提供的一些方法:
  threading.currentThread(): 返回当前的线程变量。
  threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
  threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。
```

2. 实例

```python
from threading import Thread
import threading
import time

def work():
    time.sleep(2)
    print(threading.current_thread().getName())


if __name__ == '__main__':
    t=Thread(target=work)
    t.start()
    print(threading.current_thread().getName())
    print(threading.current_thread())
    print(threading.enumerate())
    print(threading.active_count())
    print('主线程')

# MainThread
```

```
# <_MainThread(MainThread, started 57052)>
# [<_MainThread(MainThread, started 57052)>, <Thread(Thread-1, started 57484)>]
# 2
# 主线程
# Thread-1
```

5. 守护线程

1. 无论是进程还是线程，都遵循：守护xxx会等待主xxx运行完毕后被销毁

2. 需要强调的是：运行完毕并非终止运行

```
1.对主进程来说，运行完毕指的是主进程代码运行完毕
2.对主线程来说，运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕，主线程才算运行完毕
```

3. 详细解释

```
1. 主进程在其代码结束后就已经算运行完毕了（守护进程在此时就被回收），
   然后主进程会一直等非守护的子进程都运行完毕后回收子进程的资源(否则会产生僵尸进程)，才会结束，
2. 主线程在其他非守护线程运行完毕后才算运行完毕（守护线程在此时就被回收）。
   因为主线程的结束意味着进程的结束，进程整体的资源都将被回收，
   而进程必须保证非守护线程都运行完毕后才能结束。
```

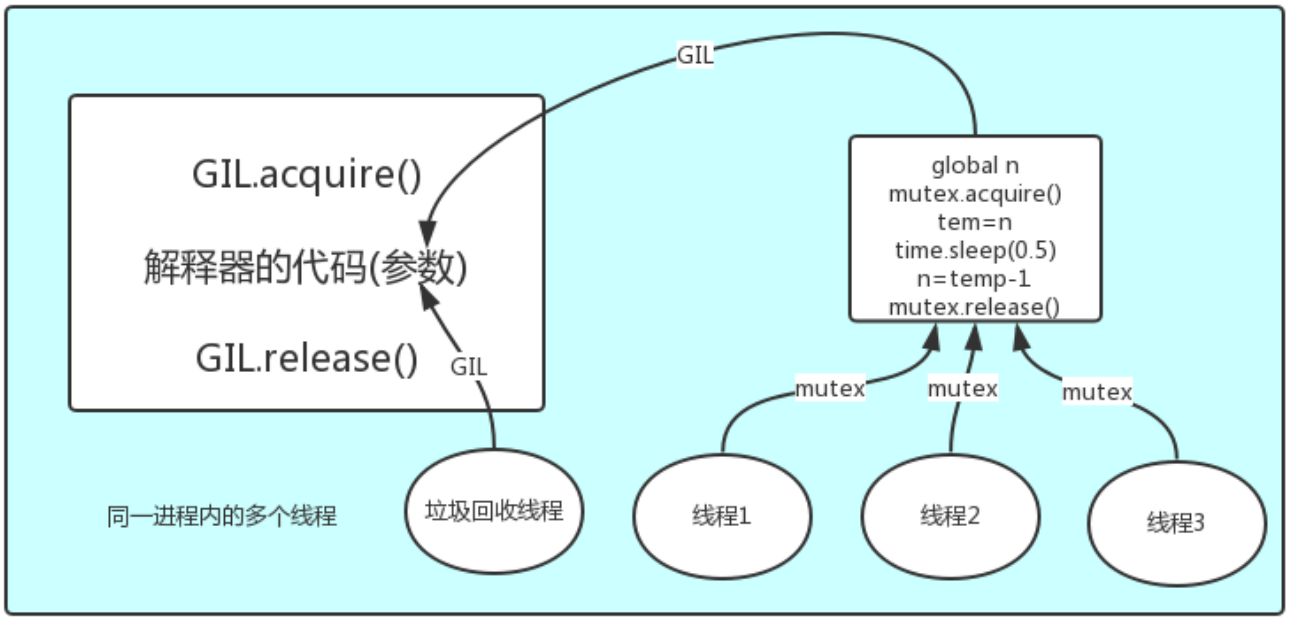
6. GIL(Global Interpreter Lock/全局解释器锁)

1. 介绍:

```
1. 在Cpython解释器中，同一个进程下开启的多线程，同一时刻只能有一个线程执行，无法利用多核优势
2. GIL并不是Python的特性，Python完全可以不依赖于GIL(JPython就没有GIL)
```

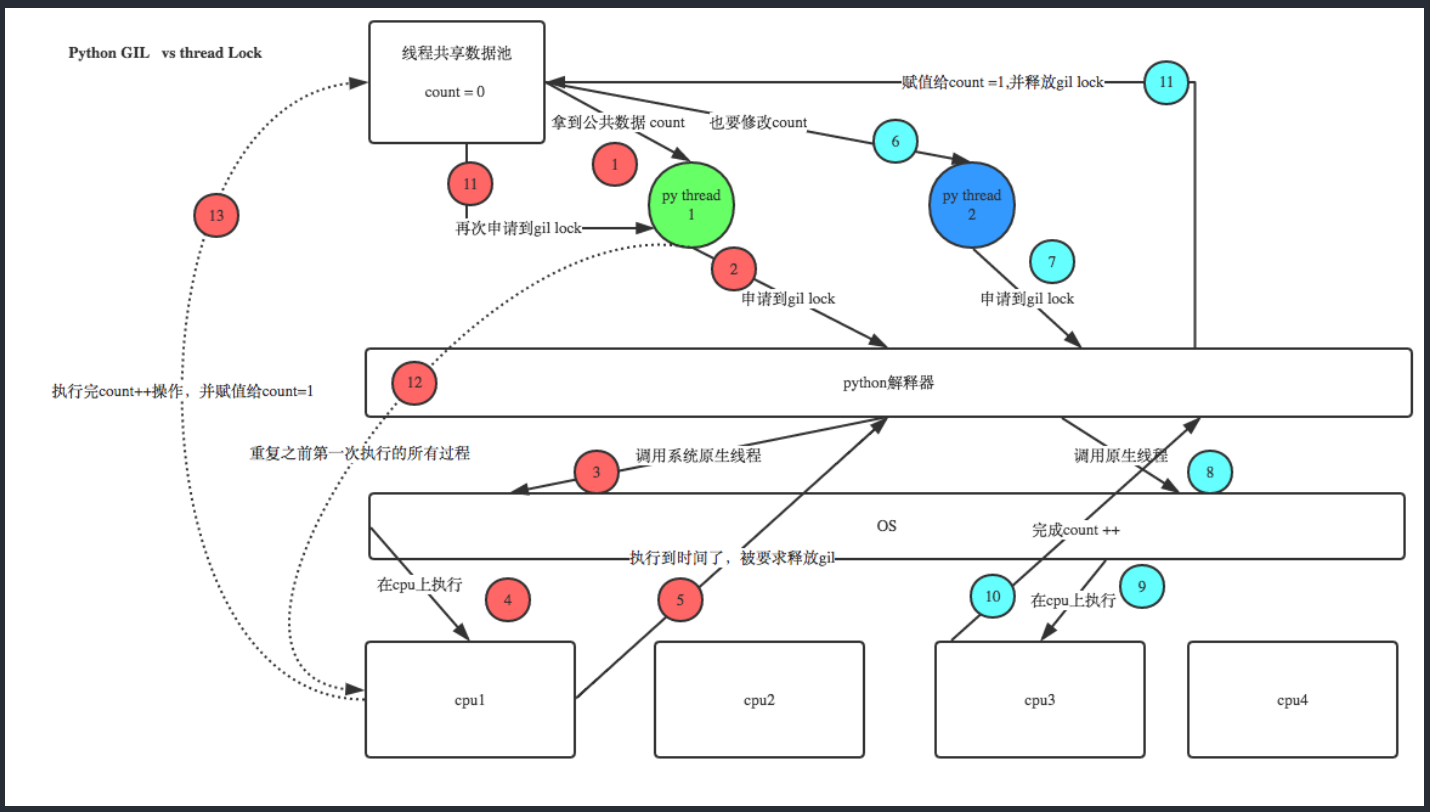
2. GIL介绍
   GIL本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，
   都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安

全



3. GIL与Lock

GIL保护的是解释器级的数据，保护用户自己的数据则需要自己加锁处理



4. GIL与多线程

现在的计算机基本上都是多核，python对于计算密集型的任务开多线程的效率并不能带来多大性能上的提升，甚至不如串行(没有大量切换)，但是，对于IO密集型的任务效率还是有显著提升的。

应用：

多线程用于IO密集型，如socket，爬虫，web

多进程用于计算密集型，如金融分析

7. 同步锁

1. 注意

```
1. 线程抢的是GIL锁，GIL锁相当于执行权限，拿到执行权限后才能拿到互斥锁Lock，
   其他线程也可以抢到GIL，但如果发现Lock仍然没有被释放则阻塞，即便是拿到执行权限GIL也要立刻交出来

2. join是等待所有，即整体串行，而锁只是锁住修改共享数据的部分，即部分串行，
   要想保证数据安全的根本原理在于让并发变成串行，join与互斥锁都可以实现，毫无疑问，互斥锁的部分串行效率要更高
```

2. GIL VS Lock

```
1. 锁的目的是为了保护共享的数据,保护不同的数据就应该加不同的锁
2. GIL 与Lock是两把锁，保护的数据不一样，前者是解释器级别的（当然保护的就是解释器级别的数据，比如垃圾回收的数据），
   后者是保护用户自己开发的应用程序的数据，很明显GIL不负责这件事，只能用户自定义加锁处理，即Lock
```

3. 同步锁：

```
锁通常被用来实现对共享资源的同步访问。为每一个共享资源创建一个Lock对象，
当你需要访问该资源时，调用acquire方法来获取锁对象（如果其它线程已经获得了该锁，则当前线程需等待其被释放），
待资源访问完后，再调用release方法释放锁
```

4. 示例代码

```python
from threading import Thread,Lock
import os,time
def work():
    global n
    lock.acquire()
    temp=n
    time.sleep(0.1)
    n=temp-1
    lock.release()
if __name__ == '__main__':
    lock=Lock()
    n=100
    l=[]
    for i in range(100):
        p=Thread(target=work)
        l.append(p)
        p.start()
    for p in l:
        p.join()

    print(n) #结果肯定为0，由原来的并发执行变成串行，牺牲了执行效率保证了数据安全
```

5. GIL锁与互斥锁综合分析

```
1. 100个线程去抢GIL锁，即抢执行权限
2. 肯定有一个线程先抢到GIL（暂且称为线程1），然后开始执行，一旦执行就会拿到lock.acquire()
3. 极有可能线程1还未运行完毕，就有另外一个线程2抢到GIL，然后开始运行，
   但线程2发现互斥锁lock还未被线程1释放，于是阻塞，被迫交出执行权限，即释放GIL
4. 直到线程1重新抢到GIL，开始从上次暂停的位置继续执行，
   直到正常释放互斥锁lock，然后其他的线程再重复2 3 4的过程
```

6. 互斥锁实例

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

def foo():

    global n

    lock.acquire()
    temp=n
    time.sleep(0.5)
    n=temp-1
    lock.release()

if __name__ == '__main__':

    n=100
    l=[]
    lock=Lock()

    for i in range(100):
        p=Thread(target=foo)
        l.append(p)
        p.start()

    for p in l:
        p.join()

    print(n)

#0
```

8. 死锁现象与递归锁

1. 死锁：
   是指两个或两个以上的进程或线程在执行过程中，因争夺资源而造成的一种互相等待的现象，
   若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程

2. 示例代码

```python
import threading
import time

mutexA = threading.Lock()
mutexB = threading.Lock()

class MyThread(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        self.fun1()
        self.fun2()

    def fun1(self):

        mutexA.acquire()  # 如果锁被占用,则阻塞在这里,等待锁的释放

        print ("I am %s , get res: %s---%s" %(self.name, "ResA",time.time()))

        mutexB.acquire()
        print ("I am %s , get res: %s---%s" %(self.name, "ResB",time.time()))
        mutexB.release()
        mutexA.release()


    def fun2(self):

        mutexB.acquire()
        print ("I am %s , get res: %s---%s" %(self.name, "ResB",time.time()))
        time.sleep(0.2)

        mutexA.acquire()
        print ("I am %s , get res: %s---%s" %(self.name, "ResA",time.time()))
        mutexA.release()

        mutexB.release()

if __name__ == "__main__":

    print("start----------------------------%s"%time.time())

    for i in range(0, 10):
        my_thread = MyThread()
        my_thread.start()
```

3. 递归锁

定义：在Python中为了支持在同一线程中多次请求同一资源，python提供了可重入锁RLock

```
这个RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，
从而使得资源可以被多次require。直到一个线程所有的acquire都被release，
```

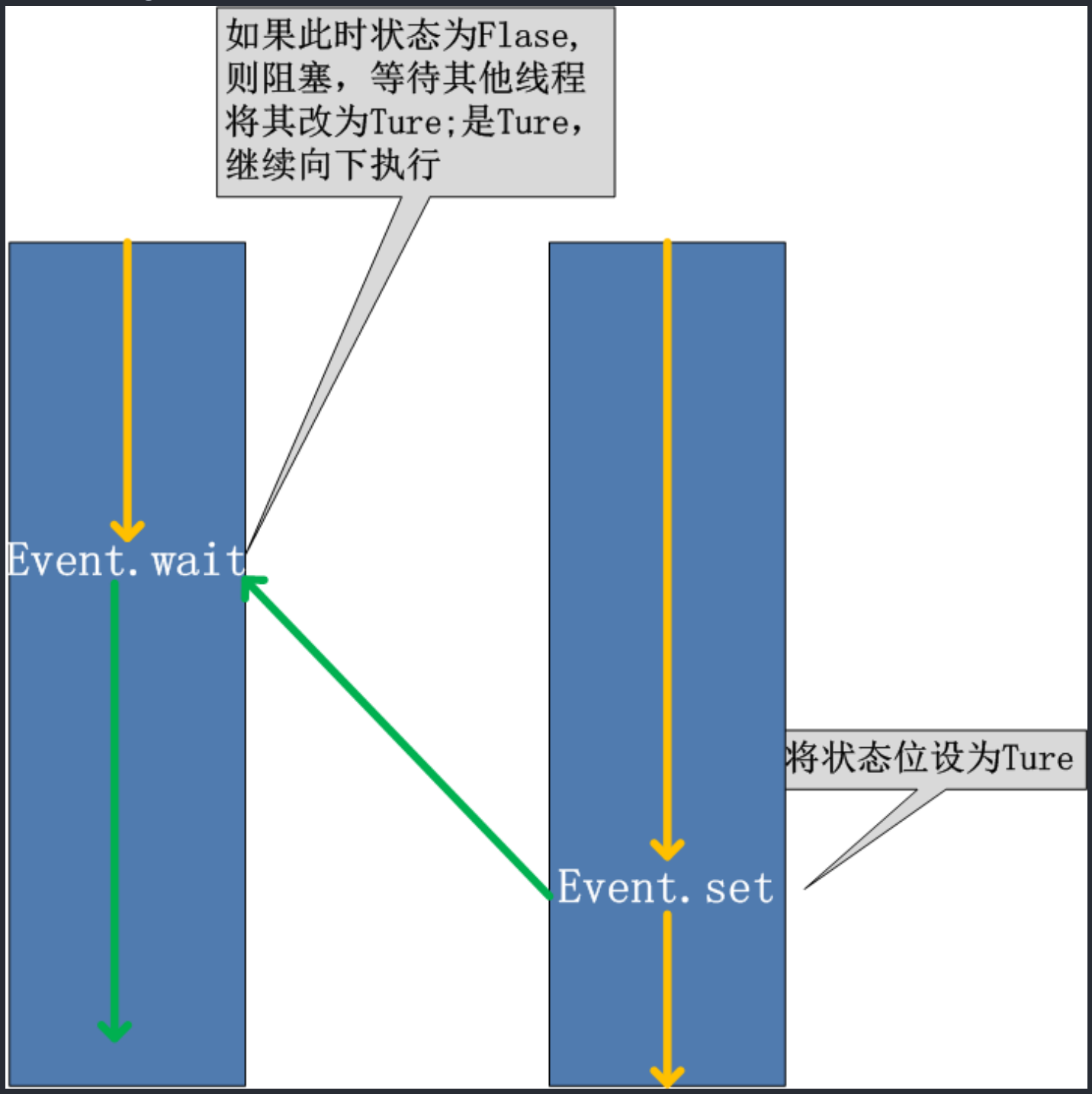
```
其他的线程才能获得资源。上面的例子如果使用RLock代替Lock，则不会发生死锁
```

注意：

```
mutexA=mutexB=threading.RLock() #一个线程拿到锁，counter加1，
该线程内又碰到加锁的情况，则counter继续加1，这期间所有其他线程都只能等待，
等待该线程释放所有锁，即counter递减到0为止
```

9. Event对象

1. event:
   调用threading库中的Event对象




```
对象包含一个可由线程设置的信号标志,它允许线程等待某些事件的发生。
在初始情况下,Event对象中的信号标志被设置为假。
如果有线程等待一个Event对象， 而这个Event对象的标志为假,那么这个线程将会被一直阻塞直至该标志为真.
```

一个线程如果将一个Event对象的信号标志设置为真,它将唤醒所有等待这个Event对象的线程。
如果一个线程等待一个已经被设置为真的Event对象,那么它将忽略这个事件, 继续执行

2. 使用示例

```
event.isSet()：返回event的状态值；
event.wait()：如果 event.isSet()==False将阻塞线程；
event.set()： 设置event的状态值为True，所有阻塞池的线程激活进入就绪状态， 等待操作系统调度；
event.clear()：恢复event的状态值为False。
```

3. 实际应用

```
#有多个工作线程尝试链接MySQL，我们想要在链接前确保MySQL服务正常才让那些工作线程去连接MySQL服务器，
# 如果连接不成功，都会去尝试重新连接。那么我们就可以采用threading.Event机制来协调各个工作线程的连接操作

from threading import Thread,Event
import time,threading

def conn_mysql():
    count=1
    while not event.is_set():
        print('<%s>第%s次尝试连接'%(threading.current_thread().getName(),count))
        event.wait(0.5)
        count += 1


    print('<%s>链接成功'%(threading.current_thread()))


def check_mysql():
    time.sleep(1)
    event.set()

if __name__ == '__main__':

    event=Event()

    conn1=Thread(target=conn_mysql)
```

```
    conn2=Thread(target=conn_mysql)
    check=Thread(target=check_mysql)


    conn1.start()
    conn2.start()
    check.start()


# <Thread-1>第1次尝试连接
# <Thread-2>第1次尝试连接
# <Thread-1>第2次尝试连接
# <Thread-2>第2次尝试连接
# <<Thread(Thread-1, started 99548)>>链接成功
# <<Thread(Thread-2, started 102636)>>链接成功
```

10. 信号量Semaphore

1. 解析

```
Semaphore管理一个内置的计数器，
每当调用acquire()时内置计数器-1；
调用release() 时内置计数器+1；
计数器不能小于0；当计数器为0时，acquire()将阻塞线程直到其他线程调用release()。
```

2. 示例代码

```python
import threading
import time

semaphore = threading.Semaphore(5)

def func():
    if semaphore.acquire():
        print (threading.currentThread().getName() + ' get semaphore')
        time.sleep(2)
        semaphore.release()

for i in range(20):
```

```
    t1 = threading.Thread(target=func)
    t1.start()
```

11. 线程\进程队列queue

1. get与put方法

```
创建一个“队列”对象

import Queue
q = Queue.Queue(maxsize = 10)
Queue.Queue类即是一个队列的同步实现。队列长度可为无限或者有限。可通过Queue的构造函数的可选参数
maxsize来设定队列长度。如果maxsize小于1就表示队列长度无限。

将一个值放入队列中
q.put(10)
调用队列对象的put()方法在队尾插入一个项目。put()有两个参数，第一个item为必需的，为插入项目的值;
第二个block为可选参数，默认为

1。如果队列当前为空且block为1，put()方法就使调用线程暂停,直到空出一个数据单元。如果block为0，

put方法将引发Full异常。



将一个值从队列中取出

q.get()

调用队列对象的get()方法从队头删除并返回一个项目。可选参数为block，默认为True。如果队列为空且
block为True，get()就使调用线程暂停，直至有项目可用。如果队列为空且block为False，队列将引发Em
pty异常。
```

2. join与task_done方法

```
join() 阻塞进程，直到所有任务完成，需要配合另一个方法task_done。

    def join(self):
     with self.all_tasks_done:
      while self.unfinished_tasks:
       self.all_tasks_done.wait()

task_done() 表示某个任务完成。每一条get语句后需要一条task_done。

import queue
q = queue.Queue(5)
q.put(10)
q.put(20)
```

```
print(q.get())
q.task_done()
print(q.get())
q.task_done()

q.join()

print("ending!")
```

3. 其他方法

```
此包中的常用方法(q = Queue.Queue()):

q.qsize() 返回队列的大小
q.empty() 如果队列为空，返回True,反之False
q.full() 如果队列满了，返回True,反之False
q.full 与 maxsize 大小对应
q.get([block[, timeout]]) 获取队列，timeout等待时间
q.get_nowait() 相当q.get(False)非阻塞
q.put(item) 写入队列，timeout等待时间
q.put_nowait(item) 相当q.put(item, False)
q.task_done() 在完成一项工作之后，q.task_done() 函数向任务已经完成的队列发送一个信号
q.join() 实际上意味着等到队列为空，再执行别的操作
```

4. 三种模式

```
Python Queue模块有三种队列及构造函数:
1、Python Queue模块的FIFO队列先进先出。   class queue.Queue(maxsize)
2、LIFO类似于堆，即先进后出。             class queue.LifoQueue(maxsize)
3、还有一种是优先级队列级别越低越先出来。  class queue.PriorityQueue(maxsize)

import queue

#先进后出
q=queue.LifoQueue()

q.put(34)
q.put(56)
q.put(12)

#优先级
q=queue.PriorityQueue()
q.put([5,100])
q.put([7,200])
q.put([3,"hello"])
q.put([4,{"name":"alex"}])

while 1:
  data=q.get()
  print(data)
```

12. 生产者消费者模型

1. 生产者消费者模型简介

```
在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。
该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。
```

2. 为什么要使用生产者和消费者模式

```
在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，
如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。
同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。
为了解决这个问题于是引入了生产者和消费者模式。
```

3. 什么是生产者消费者模式

```
生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，
而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，
消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，
平衡了生产者和消费者的处理能力。
```

4. 示例代码

```python
import time,random
import queue,threading

q = queue.Queue()

def Producer(name):
  count = 0
  while count <10:
    print("making........")
    time.sleep(random.randrange(3))
    q.put(count)
    print('Producer %s has produced %s baozi..' %(name, count))
    count +=1
    #q.task_done()
    #q.join()
    print("ok......")
def Consumer(name):
  count = 0
  while count <10:
    time.sleep(random.randrange(4))
    if not q.empty():
        data = q.get()
        #q.task_done()
        #q.join()
        print(data)
        print('\033[32;1mConsumer %s has eat %s baozi...\033[0m' %(name, data))
```

```
    else:
        print("-----no baozi anymore----")
    count +=1

p1 = threading.Thread(target=Producer, args=('A',))
c1 = threading.Thread(target=Consumer, args=('B',))
# c2 = threading.Thread(target=Consumer, args=('C',))
# c3 = threading.Thread(target=Consumer, args=('D',))
p1.start()
c1.start()
# c2.start()
# c3.start()
```