multiprocess管理并发操作

要让Python支持多进程，我们应先了解操作系统的相关知识，Linux操作系统提供了一个fork()系统调用，它相比普通的函数来说相对特殊，因为普通函数调用一次返回一次，而这个fork()调用一次返回两次，因为操作系统把当前的进程(父进程)复制一份生成了子进程， 然后分别在父进程和子进程中返回，子进程永远返回0，而父进程返回子进程的ID，这样做的理由是父进程可以fork(复制)出很多子进程，所以父进程要记下每个子进程的ID,而子进程需要调用 getppid()就可以拿到父进程.

In [3]:
```
#示例代码1:
import os

def f1():
    print('My process is is %s'%os.getpid())

    pid = os.fork()

    if pid == 0:
        print('pid is %s My father is %s'%(os.getpid(), os.getppid()))
    else:
        print('I has a son process pid is %s' %pid)


if __name__ == '__main__':
    f1()
```

```
My process is is 18792
I has a son process pid is 18980
pid is 18980 My father is 18792
```

1.1 单进程串行执行两个函数,所耗费的时间为两个函数分别耗费的时间之和.

In [6]:
```
import time
def f1():
    time.sleep(1)
    print('this is f1')

def f2():
    time.sleep(1)
    print('this is f2')

if __name__ == '__main__':
    beg = time.time()
    f1()
    f2()
    end = time.time()
    print('Cost time %s' %(end - beg))
```

```
this is f1
this is f2
Cost time 2.0070688724517822
```

1.2 一个简单的多进程例子，通过多进程的方式并发执行两个函数，实现提升代码执行效率的作用.

In [3]:
```
import time
import multiprocessing

def f1():
    time.sleep(1)
    print('this is f1')

def f2():
    time.sleep(1)
    print('this is f2')


beg = time.time()
m1 = multiprocessing.Process(target=f1)
m2 = multiprocessing.Process(target=f2)
print(multiprocessing.cpu_count())


for j in [m1, m2]:
    j.start()
    print(j.pid, j.name, j.is_alive())

print(multiprocessing.active_children())

for j in [m1, m2]:
    j.join()

end = time.time()
print("cost time %s " %(end-beg))
```

```
8
37678 Process-5 True
37679 Process-6 True
[<Process(Process-6, started)>, <Process(Process-5, started)>]
this is f1
this is f2
cost time 1.022265911102295
```

进程对象pid获取当前进程的进程ID，name获取当前进程的名字，is_alive()判断进程是否存在.

通过多进程并发执行两个耗时为1s的函数，最终执行完代码所花费的时间为1s，可见通过该多进程方式起到了提升代码执行效率的作用.

join()方法可以等待子进程结束后再继续往下运行父进程中的代码，通常用于进程间的同步. 默认情况下守护进程会无限阻塞，可以传入一个超时参数，即使进程在这个超时的时间内没有完成join()也可以返回，例join(10)

1.2.2 派生进程
要在一个单独的进程中开始工作，尽管最简单的方法是在multiprocess.Process中传入一个目标函数，也可以采用子类定制的形式.
示例代码:

In [3]:
```
import multiprocessing
import time

class Worker(multiprocessing.Process):
    def run(self):
        time.sleep(1)
        print("This is %s"%self.name)

if __name__ == '__main__':
    beg = time.time()
    jobs = []
    for i in range(5):
        p = Worker()
        jobs.append(p)
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    end  = time.time()
    print('Cost Time %s' %(end-beg))
```

```
This is Worker-6
This is Worker-7
This is Worker-8
This is Worker-9
This is Worker-10
Cost Time 1.027980089187622
```

1.3 守护进程--daemon属性的作用

在脚本运行过程中有一个主进程，若在主进程中创建了子进程，当主进程结束时根据子进程daemon属性值的不同可能会发生下面的两种情况之一：

1. 如果某个子进程的daemon属性为False，主进程结束时会检测该子进程是否结束，如果该子进程还在运行，则主进程会等待它完成后再退出.
2. 如果某个子进程的daemon属性为True，主进程运行结束时不对这个子进程进行检查而直接退出，同时所有daemon值为True的子进程将随主进程一起结束，而不论是否运行完成.

属性daemon的值默认为False，如果需要修改，必须在调用start()方法启动进程之前进行设置。

In [7]:
```
import  multiprocessing
import time
import sys

def daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    sys.stdout.flush()
    time.sleep(100)
    print('Exiting:', p.name, p.pid)
    sys.stdout.flush()

def non_daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    sys.stdout.flush()
    print('Exiting', p.name, p.pid)
    sys.stdout.flush()

d = multiprocessing.Process(name = 'daemon', target = daemon)
d.daemon = False

n = multiprocessing.Process(name = 'non_daemon', target = non_daemon)
n.daemon = False

d.start()
time.sleep(1)
n.start()
```

```
Starting: daemon 41444
Starting: non_daemon 41446
Exiting non_daemon 41446
Exiting: daemon 41444
```

输出中没有守护进程的exiting，因为在守护进程从其100秒的睡眠时间唤醒之前，所有的非守护进程都已经退出，守护进程会在主程序退出之前自动终止，以免留下孤进程继续运行.

1.4 终止进程
当一个进程看起来已经挂起或已经进入死锁状态，则需要能够强制性的将其结束，对一个对象调用terminate() 会结束该子进程

In [25]:
```
#示例代码:

import time
import multiprocessing


def f1():
    print('process begin..')
    time.sleep(1)
    print('process end...')

m = multiprocessing.Process(target=f1)

print("begin: ", m, m.is_alive())

m.start()
print("during", m, m.is_alive())

m.terminate()
print("terminate", m, m.is_alive())

m.join()
print("after: ", m, m.is_alive())


#after:  <Process(Process-27, stopped[SIGTERM])> False 表示通过terminate做了强制退出
```

```
begin:  <Process(Process-27, initial)> False
during <Process(Process-27, started)> True
terminate <Process(Process-27, started)> True
after:  <Process(Process-27, stopped[SIGTERM])> False
```

进程退出状态：

进程退出时的状态码可以通过exitcode属性来访问，multiprocess退出码有以下几种：

```
0    未生成任何错误
>0   进程有一个错误，并以该状态码退出
<0   进程由一个-1* exitcode信号结束
```

示例代码：

In [2]:
```
import multiprocessing
import sys
import time

def exit_error():
    sys.exit(1)

def exit_ok():
    return

def return_value():
    return 1

def raise_error():
    raise RuntimeError("sdfsdfsdf")

def terminated():
    time.sleep(3)


func = [exit_error, exit_ok, return_value, raise_error, terminated]
jobs = []
for j in func:
    print('Starting process %s' %j.__name__)
    j = multiprocessing.Process(target=j, name=j.__name__)
    jobs.append(j)
    j.start()
jobs[-1].terminate()

for j in jobs:
    j.join()
    print('process_name: %s, exit_code: %s' %(j.name, j.exitcode))
```

```
Starting process exit_error
Starting process exit_ok
Starting process return_value
Starting process raise_error
Starting process terminated

Process raise_error:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/multiprocessing/process.py", line 249, in _bootstrap
    self.run()
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/multiprocessing/process.py", line 93, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-2-f0c5a5a30a67>", line 15, in raise_error
    raise RuntimeError("sdfsdfsdf")
RuntimeError: sdfsdfsdf

process_name: exit_error, exit_code: 1
process_name: exit_ok, exit_code: 0
process_name: return_value, exit_code: 0
process_name: raise_error, exit_code: 1
process_name: terminated, exit_code: -15
```

1.5 进程锁

进程锁进程与进程之间是独立的，为何需要锁？对于进程，屏幕的输出只有一个，此时就涉及到资源的竞争．在Linux的Python2.x中可能出现问题，这仅仅是一种情况,多个进程之间虽然是独立的，但仅限于内存和运算，如果涉及到其它一些资源，就可能存在竞争问题，在实际使用过程中要注意思考和防范错误．

In [27]:
```
import time
import multiprocessing
import sys

def f1(stream, lock=None):
    with lock:
     time.sleep(0.1)
     stream.write("hello!!!\n")


def f2(stream, lock=None):
    with lock:
     time.sleep(1)
     stream.write("world\n")

lock = multiprocessing.Lock()
task = []
for _ in range(5):
    m1 = multiprocessing.Process(target=f1, args=(sys.stdout, lock))
    m2 = multiprocessing.Process(target=f2, args=(sys.stdout, lock))
    task.extend([m1,m2])

for j in task:
    j.start()

for j in task:
    j.join()

#比较一下加锁与不加锁时，在输出结果时的差别
```

```
hello!!!
world
hello!!!
world
hello!!!
world
hello!!!
world
hello!!!
world
```

2.1 进程池

在实际工作中，我们会有控制并发任务数的需求，我们可以使用Pool与Semaphore来管理并发任务的数量．

In [1]:
```
#示例代码1: 使用Pool来控制并发任务数量, 并关注每个进程的执行结果.

#coding:utf8
import multiprocessing
import time
import os

def func(msg):
    pid = os.getpid()
    print('msg: %s' %pid),  msg
    time.sleep(3)
    print('end')
    return 'return msg %s' %msg

pool = multiprocessing.Pool(processes=3)
beg = time.time()
jobs = []
for i in range(4):
    msg = 'hello %s' %i
    jobs.append(pool.apply_async(func, (msg, )))
print('mark!' * 3)
pool.close()
pool.join()
end = time.time()
print(end - beg)
print('sub-process done!')

for j in jobs:
    print(j.get())
```

```
msg: 42895
msg: 42896
msg: 42894
mark!mark!mark!
end
end
end
msg: 42895
end
6.03801965713501
sub-process done!
return msg hello 0
return msg hello 1
return msg hello 2
return msg hello 3
```

In [30]:
```
#示例代码2: 通过Semaphore来控制并发任务数量

import multiprocessing
import time
import os

def f1(s, msg):
    s.acquire()
    pid = os.getpid()
    print("hello: %s" %msg, pid)
    time.sleep(1)
    print("end!")
    s.release()

beg =time.time()
s = multiprocessing.Semaphore(2)
process = []

for j in range(4):
    msg = "this is %s" %j
    m = multiprocessing.Process(target=f1, args=(s, msg))
    process.append(m)

for j in process:
    j.start()
for j in process:
    j.join()
end = time.time()
print(end-beg)
```

```
hello: this is 0 22242
hello: this is 1 22243
end!
end!
hello: this is 2 22244
hello: this is 3 22245
end!
end!
2.0342071056365967
```

使用: multiprocessing.Semaphore 与 multiprocessing.Pool 的区别

multiprocessing.Semaphore
信号量的限制进程数是控制整体数量，比如我总共十个进程，信号量是2个，那一次性最多运行2个进程，但是进程的创建销毁过程还是10次。

multiprocessing.Pool(processes=2)
进程池从表面来看和信号量的功能相同，都是最多运行2两个，但是进程的创建销毁过程只有2次。

2.2 进程之间的数据共享

进程之间产生的数据默认情况下是相互独立的

In [31]:
```
#2.2.1 示例代码：进程之间的数据是相互独立的

import multiprocessing

l = []
def f1(args):
    print('processing start..')
    l.append(args)
    print('processing end..', len(l))

p = []
for i in range(3):
    m = multiprocessing.Process(target=f1, args=(i, ))
    p.append(m)

for j in p:
    j.start()

for j in p:
    j.join()

print(l, len(l))
```

```
processing start..
processing end.. 1
processing start..
processing end.. 1
processing start..
processing end.. 1
[] 0
```

In [ ]:
```
#2.2.2 示例代码：通过Manager管理共享状态

import multiprocessing

mgr = multiprocessing.Manager()
l = mgr.list()

def f1(args):
    print('processing start..')
    l.append(args)
    print('processing end..', len(l))

p = []
for i in range(3):
    m = multiprocessing.Process(target=f1, args=(i, ))
    p.append(m)

for j in p:
    j.start()

for j in p:
    j.join()

print(list(l), len(l))
```

2.2.3 通过PIPE实现进程之间的数据传输，Pipe方法返回(conn1, conn2)代表一个管道的两端，Pipe方法有duplex参数，如果duplex参数为True，那么这个管道是一个全双工的模式，也就是 conn1 Conn2 均可以收发，duplex为false的话conn1负责接受 conn2负责发送;send和recv方法分别是发送和接受消息的方法， 例如在全双工模式下，可以调用conn1.send发送消息，conn1.recv接受消息. 如果没有消息可以接受，那么recv方法会一直堵塞.

#示例代码：如果管道已经被关闭，那么recv方法会报EOFError.

In [10]:
```
import multiprocessing
import time

pipe = multiprocessing.Pipe()

pipe[0].send(1)
print(pipe[1].recv())
pipe[0].close()
print(pipe[1].recv())
```

```
1

---------------------------------------------------------------------------
EOFError                                  Traceback (most recent call last)
<ipython-input-10-171a59d390d8> in <module>
      7 print(pipe[1].recv())
      8 pipe[0].close()
----> 9 print(pipe[1].recv())

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/multiprocessing/connection.py in recv(self)
    248         self._check_closed()
    249         self._check_readable()
--> 250         buf = self._recv_bytes()
    251         return _ForkingPickler.loads(buf.getbuffer())
    252

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/multiprocessing/connection.py in _recv_bytes(self, maxsize)
    405
    406     def _recv_bytes(self, maxsize=None):
--> 407         buf = self._recv(4)
    408         size, = struct.unpack("!i", buf.getvalue())
    409         if maxsize is not None and size > maxsize:

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/multiprocessing/connection.py in _recv(self, size, read)
    381             if n == 0:
    382                 if remaining == size:
--> 383                     raise EOFError
    384                 else:
    385                     raise OSError("got end of file during message")

EOFError:
```

In [8]:
```
#示例代码: pipe的使用

import multiprocessing
import time

pipe = multiprocessing.Pipe()
print(pipe)

def proc1(pipe):
    for i in range(3):
        print('send: %s' %(i))
        pipe.send(i)
    pipe.close()

def proc2(pipe):
    while True:
        try:
            print('proc2 rev: %s' %pipe.recv())
            if not pipe.poll():
                break
        except Exception as e:
            print(str(e))

p1 = multiprocessing.Process(target=proc1, args=(pipe[0], ))
p2 = multiprocessing.Process(target=proc2, args=(pipe[1], ))
p1.start()
p2.start()
p1.join()
p2.join()
```

```
(<multiprocessing.connection.Connection object at 0x110d8c550>, <multiprocessing.connection.Connection object at 0x110d8c668>)
send: 0
send: 1
send: 2
proc2 rev: 0
proc2 rev: 1
proc2 rev: 2
```

2.2.4 使用进程安全的Queue实现进程之间的数据传递，Queue是多进程的安全队列，可以使用Queue实现多进程之间的数据传递，put方法用来插入数据到队列中. put方法还有两个可选的参数blocked和timeout，如果blocked为True并且timeout为正值，该方法会阻塞timeout指定的时间，直到该队列有剩余的空间，如果超时则会报 queue.full异常；如果blocked为False但该序列Queue已满，会立即抛出Queue.Full异常. get方法可以从队列读取并删除一个元素，同样get方法有两个可选的参数blocked和timeout，如果blocked为True并且timeout为正值，那么在时间内没有取到元素，会报queue.empty异常； 如果blocked为False有两种情况存在
1）如果queue有一个值可用，则立即返回该值.
2）如果队列为空则立即抛出Queue.Empty异常.

In [1]:
```
#示例代码:

import time
import multiprocessing

def writer_proc(q):
    for i in range(3):
        print("Send %s" %i)
        q.put(i, block=False)

def reader_proc(q):
    while 1:
        time.sleep(0.1)
        print('Receiver msg: %s' %q.get(block = False))
        if q.empty():
            break

q = multiprocessing.Queue()
p1 = multiprocessing.Process(target=writer_proc, args=(q, ))
p2 = multiprocessing.Process(target=reader_proc, args=(q,))
p1.start()
p2.start()
p1.join()
p2.join()
```

```
Send 0
Send 1
Send 2
Receiver msg: 0
Receiver msg: 1
Receiver msg: 2
```

2.3 在多进程环境下应用logging

In [ ]:
```
#示例代码1:

import multiprocessing
import logging
import time

from multiprocessing import Queue

logQ = Queue()

def f1():
    try:
        time.sleep(2)
        x = 1 / 2
    except:
        s = {"type": "error", "message": "found error!"}
        logQ.put(s)
        return 'error'
    else:
        s = {"type": "success", "message": "ok"}
        logQ.put(s)
        return 'heelo'


def f2():
    try:
        time.sleep(2)
        x = 1 / 0
    except Exception as e:
        s = {"type": "error", "message": str(e)}
        logQ.put(s)
        return 'error'
    else:
        s = {"type": "success", "message": "ok"}
        logQ.put(s)
        return 'heelo'


def f3():
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s',
                        datefmt='%Y-%m-%d %H:%M:%S',
                        filename='myapp.log',
                        filemode='w')


    console = logging.StreamHandler()
    console.setLevel(logging.DEBUG)
    formatter = logging.Formatter(
        '%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s')
    console.setFormatter(formatter)
    logging.getLogger().addHandler(console)

    while 1:
        if not logQ.empty():
            s = logQ.get()
            if s["type"] == "success":
                logging.info(s["message"])
            elif s["type"] == "error":
                logging.error(s["message"])



beg = time.time()

task = []
for j in range(100):
    m1 = multiprocessing.Process(target=f1)
    m2 = multiprocessing.Process(target=f2)
    task.append(m1)
    task.append(m2)

m3 = multiprocessing.Process(target=f3, name="multiprocess-log")
m3.daemon = True
task.insert(0, m3)

print task

for j in task:
    j.start()

print task

for j in task:
    j.join(1)

print task
end = time.time()
print end -beg
```

In [ ]:
```
#示例代码2:

from logging import getLogger, INFO, Formatter
from cloghandler import ConcurrentRotatingFileHandler
import os
import time
import multiprocessing



def f1():
    log = getLogger()
    logfile = os.path.abspath("mylogfile.log")
    rotateHandler = ConcurrentRotatingFileHandler(logfile, "a", 512 * 1024, 5)
    formatter = Formatter(
        '%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s')
    rotateHandler.setFormatter(formatter)
    log.addHandler(rotateHandler)
    log.setLevel(INFO)
    try:
        time.sleep(2)
        x = 1 / 2
    except Exception as e:
        log.error(str(e))
        return 'error'
    else:
        log.info("hello")
        return 'heelo'


def f2():
    log = getLogger()
    logfile = os.path.abspath("mylogfile.log")
    rotateHandler = ConcurrentRotatingFileHandler(logfile, "a", 512 * 1024, 5)
    formatter = Formatter(
        '%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s')
    rotateHandler.setFormatter(formatter)
    log.addHandler(rotateHandler)
    log.setLevel(INFO)
    try:
        time.sleep(2)
        x = 1 / 0
    except Exception as e:
        log.error(str(e))
        return 'error'
    else:
        log.info("hello")
        return 'heelo'

beg = time.time()

task = []
for j in range(100):
    m1 = multiprocessing.Process(target=f1)
    m2 = multiprocessing.Process(target=f2)
    task.append(m1)
    task.append(m2)

for j in task:
    j.start()

print task

for j in task:
    j.join(1)

print task
end = time.time()
print end -beg
```

3.1 Condition条件阻塞
    传统进程技术实现互斥只能是一个进程单独工作，不能实现一个进程工作一段时间停止后再通知另一个进程来工作，Condition就是解决这个问题，进程1执行(cond)，进程1执行一半等待(cond.wait())，进程2开始执行(cond)，进程2执行完毕后(cond.notify())，进程1再接着执行.

示例代码：

In [1]:
```
import multiprocessing
import time
import os

def go1():
    with cond:  # 使用条件变量(资源 Lock)
        for i in range(8):
            time.sleep(1)
            print('process-id: %s'%os.getpid(), i, "go11")
            if i == 5:
                cond.wait()  # 等待cond.notify()，再继续执行。（释放条件变量(资源 Lock)）


def go2():
    with  cond:  # 使用条件变量(资源 Lock)
        for i in range(7):
            time.sleep(1)
            print('process-id: %s' %os.getpid(), i)
        cond.notify()  # 通知，触发 cond.wait()。（释放条件变量(资源 Lock)）



cond = multiprocessing.Condition()  # 线程条件变量
multiprocessing.Process(target=go1).start()  # 和下面的线程的次序不能调。这个线程先拿到cond条件变量(资源 Lock)
multiprocessing.Process(target=go2).start()
```

```
process-id: 27793 0 go11
process-id: 27793 1 go11
process-id: 27793 2 go11
process-id: 27793 3 go11
process-id: 27793 4 go11
process-id: 27793 5 go11
process-id: 27794 0
process-id: 27794 1
process-id: 27794 2
process-id: 27794 3
process-id: 27794 4
process-id: 27794 5
process-id: 27794 6
process-id: 27793 6 go11
process-id: 27793 7 go11
```

3.2 向进程传递消息

使用Queue来回传递消息可以让进程之间可以相互通信，在本例中使用JoinableQueue，主进程使用任务队列的join()方法等待所有任务都完成后才开始处理结果.
示例代码：

In [2]:
```
import time
import multiprocessing
class Task:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self):
        time.sleep(0.1)
        return '%s * %s = %s' %(self.a, self.b, self.a * self.b)

    def __str__(self):
        return '%s * %s' %(self.a, self.b)

tasks = multiprocessing.JoinableQueue()
result = multiprocessing.Queue()

def f1():
    while 1:
        next_task = tasks.get()
        print(next_task, 'next_task')
        if next_task is None:
            print('%s: Exiting' %next_task)
            tasks.task_done()
            break
        print(next_task)
        answer = next_task()
        tasks.task_done()
        result.put(answer)

jobs = []
for _ in range(4):
    p = multiprocessing.Process(target=f1) ##创建4个进程分别执行函数f1
    jobs.append(p)

for w in jobs:
    w.start()

for i in range(10):
    tasks.put(Task(i, i))  #创建10个任务传递到Task joinablequeue

for _ in range(4):
    tasks.put(None)   #创建4个None传递到task joinablequeue

tasks.join()  ##开始执行任务

for w in jobs:
    w.join()

for _ in range(10):
    f = result.get()
    print(f)
```

```
0 * 0 next_task
2 * 2 next_task
1 * 1 next_task
3 * 3 next_task
1 * 1
0 * 0
2 * 2
3 * 3
4 * 4 next_task
5 * 5 next_task
6 * 6 next_task
6 * 6
5 * 5
4 * 4
7 * 7 next_task
7 * 7
9 * 9 next_task
8 * 8 next_task
None next_task
8 * 8
9 * 9
None: Exiting
None next_task
None: Exiting
None next_task
None next_task
None: Exiting
None: Exiting
0 * 0 = 0
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81
```

3.3 进程间信号传输

进程对象包含一个可由进程设置的信号标志，它允许进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flag被设置为false，如果有进程等待一个Event对象，而这个Event对象的标志为假，那么这个进程将会被一直阻塞直至该标志为true．一个进程如果将一个Event对象的信号标志设置为真，它将唤醒所有等待这个Event对象的进程，如果一个进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．

Event几种方法：

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。

示例代码：

In [4]:
```
import multiprocessing
event = multiprocessing.Event()
def xiao_fan(event):
    print('生产...')
    print('售卖...')
    print('等待就餐')
    event.set()
    print(1)
    event.clear()
    print(2)
    event.wait()
    print(3)
    print('谢谢光临')


def gu_ke(event):
    print('准备买早餐：%s' %event.is_set())
    event.wait() ##阻塞
    print('买到早餐：%s' %event.is_set())
    print('享受美食：%s' %event.is_set())
    print('付款，真好吃...%s' %event.is_set())
    event.set()
    print(4)
    event.clear()
    print(5)


if __name__ == '__main__':
    # 创建进程
    xf = multiprocessing.Process(target=xiao_fan, args=(event,))
    gk = multiprocessing.Process(target=gu_ke, args=(event, ))
    # 启动进程
    gk.start()
    xf.start()
```

```
准备买早餐：False
生产...
售卖...
等待就餐
1
买到早餐：True
享受美食：True
2
付款，真好吃...False
4
3
5
谢谢光临
```

In [ ]: