1. 主线程,子线程

在并发编程里，我们通常将一个过程切分成几块，然后让每个线程各自负责一块工作。当一个程序启动时，即在一个单独的线程中运行，我们叫它主线程。新创建的线程，我们叫它子线程。

通常主线程会等待所有子线程先结束

1.1 线程数量

```
#coding=utf-8
import threading
from time import sleep, ctime


def eat():
    for i in range(3):
        print("正在吃饭...%d" % i)
        sleep(1)


def drink():
    for i in range(3):
        print("正在喝水...%d" % i)
        sleep(1)


if   name   == '  main  ':
    print('---开始---:%s' % ctime())

    print("t1创建前: ", len(threading.enumerate()))
    t1 = threading.Thread(target=eat)

    print("t1创建后, t2创建前: ", len(threading.enumerate()))

    t2 = threading.Thread(target=drink)
    print("t2创建后: ",  len(threading.enumerate()))
    # 当调用 start 方法的时候，才会去真正的创建一个线程

    t1.start()
    print("t1启动后, t2启动前: ", len(threading.enumerate()))

    t2.start()
    print("所有线程都启动后: ", len(threading.enumerate()))

    print('---结束---:%s' % ctime())
```

运行结果:

```
t1创建前： 1
t1创建后，t2创建前： 1
t2创建后： 1
t1启动后，t2启动前： 2
正在吃饭...0
所有线程都启动后： 3
正在喝水...0
正在吃饭...1
正在喝水...1
正在吃饭...2
正在喝水...2
```

1.2 主线程会等待所有子线程先结束

```
import threading
from time import sleep,ctime


def eat():
    for i in range(3):
        print("正在吃饭...%d" % i)
        sleep(1)


def drink():
    for i in range(3): print("正
        在喝水...%d" % i) sleep(1)

if name == ' main ': print('---开
    始---:%s'%ctime())

    t1 = threading.Thread(target=eat)
    t2 =threading.Thread(target=drink)

    t1.start()
    t2.start()

    #sleep(5) # 屏蔽此行代码，试试看，程序是否会立马结束?
    print('---结束---:%s'%ctime())
```

运行结果：

```
---开始---:Sat Apr 28 16:27:50 2018
正在吃饭...0
---结束---:Sat Apr 28 16:27:50 2018
正在喝水...0
正在吃饭...1
正在喝水...1
正在吃饭...2
正在喝水...2
```

2. 线程执行顺序

```
#coding=utf-8
import threading
import time

class MyThread(threading.Thread):
    def run(self):
        for i in range(3):
            time.sleep(1)
            msg = "I'm "+self.name+' @'+str(i)
            print(msg)
def test():
    for i in range(5):
        t = MyThread()
        t.start()
if name == ' main ':
    test()
```

执行结果：(运行的结果可能不一样，但是大体是一致的)

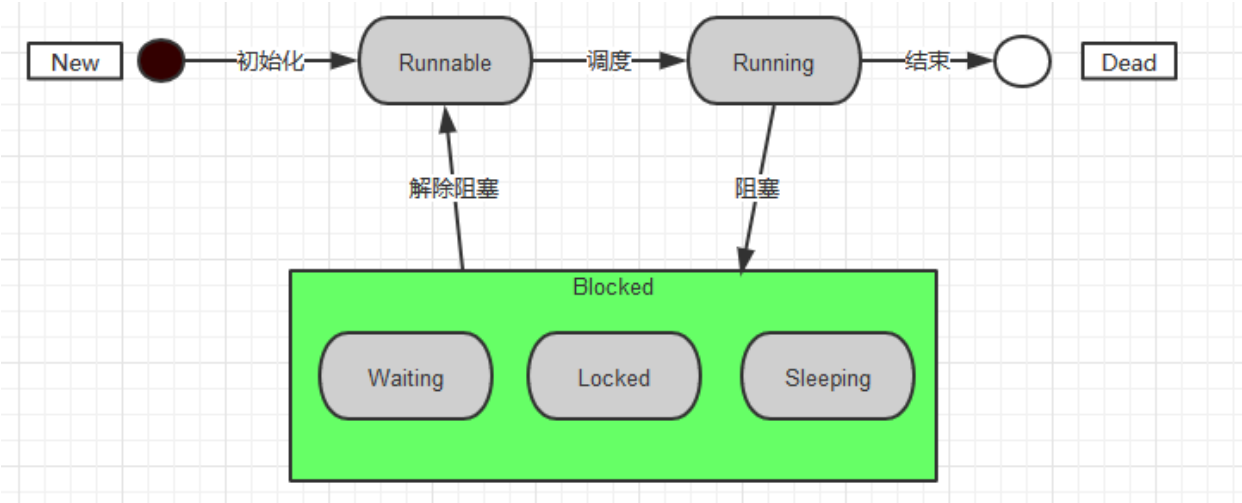
```
I'm Thread-1 @ 0
I'm Thread-2 @ 0
I'm Thread-3 @ 0
I'm Thread-4 @ 0
I'm Thread-5 @ 0
I'm Thread-1 @ 1
I'm Thread-2 @ 1
I'm Thread-3 @ 1
I'm Thread-4 @ 1
I'm Thread-5 @ 1
I'm Thread-1 @ 2
I'm Thread-2 @ 2
I'm Thread-3 @ 2
I'm Thread-4 @ 2
I'm Thread-5 @ 2
```

说明:

从代码和执行结果我们可以看出，多线程程序的执行顺序是不确定的。当执行到sleep语句时，线程将被阻塞（Blocked），到sleep结束后，线程进入就绪（Runnable）状态，等待调度。而线程调度将自行选择一个线程执行。上面的代码中只能保证每个线程都运行完整个run函数，但是线程的启动顺序、run函数中每次循环的执行顺序都不能确定。

2.1 总结:

1. 每个线程一定会有一个名字，尽管上面的例子中没有指定线程对象的name，但是python会自动为线程指定一个名字。
2. 当线程的run()方法结束时该线程完成。
3. 无法控制线程调度程序，但可以通过别的方式来影响线程调度的方式。
4. 线程的生命周期:



各个状态的说明如下：

1）． new新建。新创建的线程经过初始化后，进入Runnable状态。

2）. Runnable就绪。等待线程调度。调度后进入运行状态。

3）. Running运行。

4）. Blocked阻塞，暂停运行，解除阻塞后进入Runnable状态重新等待调度。

5）． Dead消亡。线程方法执行完毕返回或者异常终止。

可能有3种情况从Running进入Blocked：

- 同步：线程中获取同步锁，但是资源已经被其他线程锁定时，进入Locked状态，直到该资源可获取（获取的顺序由Lock队列控制）
- 睡眠：线程运行sleep()或join()方法后，线程进入Sleeping状态。区别在于sleep等待固定的间，而join是等待子线程执行完。当然join也可以指定一个“超时时间”。从语义上来说，如果两个线程a，b,在a中调用b.join()，相当于合并(join)成一个线程。最常见的情况是在主线程中join所有的子线程。等
- 待：线程中执行wait()方法后，线程进入Waiting状态，等待其他线程的通知(notify）。

3. 多线程共享全局变量验证

3.1 普通全局变量

```
import threading
import time

num = 5

def test1():
    global num
    num += 1
    print("-----num=%d in test1 -----" % num)

def test2():
    print("-----num=%d in test2 -----" % num)

t1 = threading.Thread(target=test1)
t1.start()

time.sleep(1) #保证 t1 先执行完
t2 = threading.Thread(target=test2)
t2.start()
```

运行结果：

```
-----num=6 in test1 -----
-----num=6 in test2 -----
```

3.2 列表当做实参传递到线程中

```
import time
import threading

tlist = [1, 2]

def test1(temp):
    temp.append(33)
    print("-----in test1 tlist=",tlist)

def test2():
    print("-----in test2 tlist=",tlist)

t1 = threading.Thread(target=test1, args=(tlist,))#以元组方式传参
t1.start()

time.sleep(1) #保证 test1()先执行完

t2 = threading.Thread(target=test2)
t2.start()
```

运行结果：

```
-----in test1 tlist= [1, 2, 33]
-----in test2 tlist= [1, 2, 33]
```

3.3 总结

- 在一个进程内的所有线程共享全局变量，很方便在多个线程间共享数据
- 缺点就是，线程是对全局变量随意遂改可能造成多线程之间对全局变量的混乱（即线程非安全）

4. 多线程共享全局变量带来的问题

如果多个线程同时对同一个全局变量操作，会出现资源竞争问题，从而数据结果会不正确。

```
import threading
import time

testnum = 0

def test1(num):
    global testnum
    for i in range(num):
        testnum += 1
    print("----testnum=%d in test1 ---" % testnum)

def test2(num):
    global testnum
    for i in range(num):
        testnum += 1
    print("----testnum=%d in test2 ---" % testnum)

t1 = threading.Thread(target=test1, args=(6666666,))
t2 = threading.Thread(target=test2, args=(6666666,))
t1.start()
t2.start()

time.sleep(3) #延时的目的：让线程都执行完毕

print("-----in main thread testnum=%d----" % testnum)
```

运行结果：

```
----testnum=8898256 in test1 ---
----testnum=9237198 in test2 ---
-----in main thread testnum=9237198----
```

由于是多线程同时操作，有可能出现下面情况：

- 在 g_num=0 时，t1 取得 g_num=0。此时系统把 t1 调度为”sleeping”状态，把 t2 转换为”running”状态，t2也获得 g_num=0
- 然后 t2 对得到的值进行加 1 并赋给 g_num，使得 g_num=1

- 然后系统又把 t2 调度为”sleeping”，把 t1 转为”running”。线程 t1 又把它之前得到的 0 加 1 后赋值给 g_num
- 这样导致虽然 t1 和 t2 都对 g_num 加 1，但结果仍然是 g_num=1

5. 同步与互斥

什么是同步，什么是互斥?

现代操作系统基本都是多任务操作系统，即同时有大量可调度实体在运行。在多任务操作系统中，同时运行的多个任务可能：

- 都需要访问/使用同一种资源
- 多个任务之间有依赖关系，某个任务的运行依赖于另一个任务

这两种情形是多任务编程中遇到的最基本的问题，也是多任务编程中的核心问题，同步和互斥就是用于解决这两个问题的。

- 互斥 ：是指散布在不同任务之间的若干程序片断，当某个任务运行其中一个程序片段时，其它任务就不能运行它们之中的任一程序片段，只能等到该任务运行完这个程序片段后才可以运行。最基本的场景就是：一个公共资源同一时刻只能被一个进程或线程使用，多个进程或线程不能同时使用公共资源。
- 同步：是指散步在不同任务之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。最基本的场景就是：两个或两个以上的进程或线程在运行过程中协同调用，按预定的先后次序运行。比如 A 任务的运行依赖于 B 任务产生的数据。

同步是一种更为复杂的互斥，而互斥是一种特殊的同步。

6. 互斥锁

6.1. 互斥锁简介

当多个线程几乎同时修改某一个共享数据的时候，需要进行同步控制

线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁。

互斥锁为资源引入一个状态：锁定/非锁定。

- 某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改
- 直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源
- 互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性

6.2. 互斥锁基本操作

threading 模块中定义了 Lock 类，可以方便的处理锁定：

```
# 创建锁
mutex = threading.Lock()

# 锁定
mutex.acquire()

# 释放
mutex.release()
```

- 如果这个锁之前是没有上锁的，那么 acquire 不会堵塞
- 如果在调用 acquire 对这个锁上锁之前它已经被其他线程上了锁，那么此时 acquire 会堵塞，直到这个锁被解锁为止

6.3. 使用互斥锁来解决多线程共享全局变量的问题

(1)示例：

```
import threading
import time

testnum = 0

def test1(num):
    global testnum
    for i in range(num):
        mutex.acquire() # 上锁
        testnum += 1
        mutex.release() # 解锁
    print("----in test1 testnum=%d---" % testnum)

def test2(num):
    global testnum
    for i in range(num):
        mutex.acquire() # 上锁
        testnum += 1
        mutex.release() # 解锁
    print("----in test2 testnum=%d---" % testnum)

if __name__ == '__main__':
    mutex = threading.Lock() # 创建一个互斥锁，默认没有上锁
    t1 = threading.Thread(target=test1, args=(6666666,))
    t2 = threading.Thread(target=test2, args=(6666666,))
    t1.start()
    t2.start()

    time.sleep(3)

    print("-----in main thread testnum=%d----" % testnum)
```

运行结果：

```
-----in main thread testnum=4120798----
----in test1 testnum=12982934---
----in test2 testnum=13333332---
```

(2)过程:

当一个线程调用锁的　acquire()方法获得锁时，锁就进入“locked”状态。

每次只有一个线程可以获得锁。如果此时另一个线程试图获得这个锁，该线程就会变为“blocked”状态，称为“阻塞”，直到拥有锁的线程调用锁的 release()方法释放锁之后，锁进入“unlocked”状态。

线程调度程序从处于同步阻塞状态的线程中选择一个来获得锁，并使得该线程进入运行（running）状态。

(3)总结

锁的好处:

- 确保了某段关键代码只能由一个线程从头到尾完整地执行

锁的坏处:

- 阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了
- 由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁

7. 死锁

如果有多个公共资源，在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，这会引起什么问题?

所谓死锁就是指两个或两个以上的进程在执行过程中，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。由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

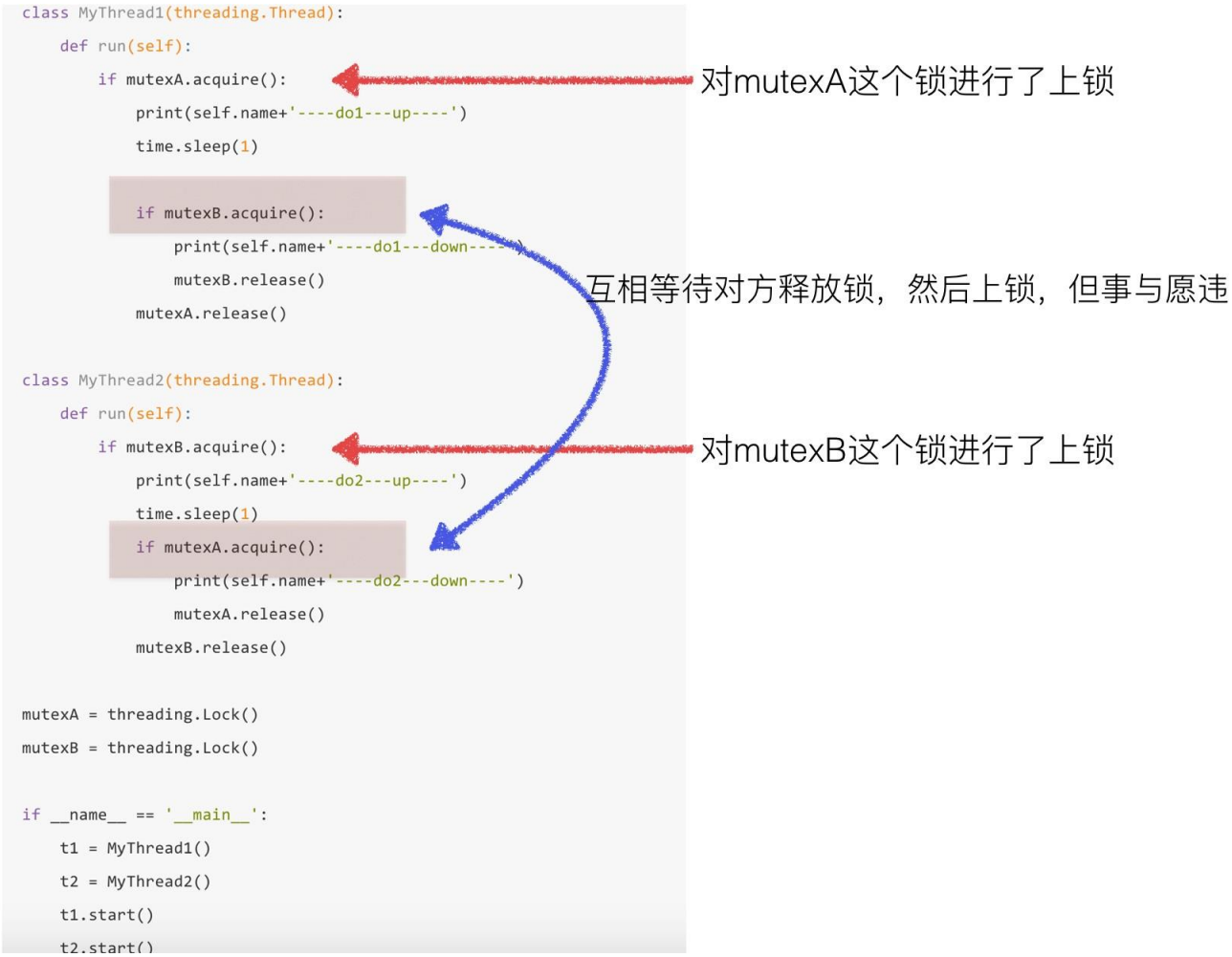
```
import threading
import time

class MyThread1(threading.Thread):
    def run(self):
        # 对 mutexA 上锁
        mutexA.acquire()
        # mutexA 上锁后，延时 1 秒，等待另外那个线程 把 mutexB 上锁
        print('----'+ self.name+'----do1---up----')
        time.sleep(1)
        # 此时会堵塞，因为这个 mutexB 已经被另外的线程抢先上锁了
        mutexB.acquire()
        print('----'+ self.name+'----do1---down----')
        mutexB.release()
        # 对 mutexA 解锁
        mutexA.release()
```

```
class MyThread2(threading.Thread):
    def run(self):
        # 对 mutexB 上锁
        mutexB.acquire()
        # mutexB 上锁后，延时 1 秒，等待另外那个线程 把 mutexA 上锁
        print('----'+ self.name+'----do2---up----')
        time.sleep(1)
        # 此时会堵塞，因为这个 mutexA 已经被另外的线程抢先上锁了
        mutexA.acquire()
        print('----'+ self.name +'----do2---down----')
        mutexA.release()
        # 对 mutexB 解锁
        mutexB.release()

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

t1 = MyThread1()
t2 = MyThread2()

t1.start()
t2.start()
```

运行结果：

此时已经进入到了死锁状态，可以使用 ctrl-c 退出。可通过添加超时时间避免死锁。

说明：



避免死锁:

- 程序设计时要尽量避免（银行家算法）
- 添加超时时间等

8. 案例：多线程实现生产者消费者模式

需求:

- 有2个线程
- 线程 1 用来生产数据
- 线程 2 用来消费数据

```
from queue import Queue
import random
import threading
import time


class Producer(threading.Thread):
    # 生产者类
    def init (self, name, queue):
        threading.Thread. init (self, name=name)
        self.data = queue
```

```
    def run(self):
        for i in range(5):
            print("%s 正在生产 %d !" % (self.getName(), i))
            self.data.put(i)
            time.sleep(random.randint(1, 2))
        print("%s 生产结束!" % self.getName())


class Consumer(threading.Thread):
    # 消费者类
    def init (self, name, queue):
        threading.Thread. init (self, name=name)
        self.data = queue

    def run(self):
        for i in range(5):
            val = self.data.get()
            print("%s 正在消费 %d !" % (self.getName(), val))
            time.sleep(random.randint(1, 2))
        print("%s 消费结束!" % self.getName())


def main():
    queue = Queue()
    producer = Producer('Producer', queue)
    consumer = Consumer('Consumer', queue)

    producer.start()
    consumer.start()

    producer.join()
    consumer.join()
    print('线程结束!')


if   name   == '  main  ':
    main()
```

9. 协程

9.1. 协程是什么

协程，又称微线程，纤程。英文名 Coroutine。

协程是 python 个中另外一种实现多任务的方式，只不过比线程更小占用更小执行单元（理 解为需要的资源）。为啥说它是一个执行单元，因为它自带 CPU 上下文。这样只要在合适 的时机， 我们可以把一个协程切换到另一个协程。 只要这个过程中保存或恢复 CPU 上下 文那么程序还是可以运行的。

通俗的理解：在一个线程中的某个函数，可以在任何地方保存当前函数的一些临时变量等信 息，然后切换到另外一个函数中执行，注意不是通过调用函数的方式做到的，并且切换的次 数以及什么时候再切换到原来的函数都由开发者自己确定

9.2. 协程和线程差异

在实现多任务时，线程切换从系统层面远不止保存和恢复 CPU 上下文这么简单。

操作系统为了程序运行的高效性每个线程都有自己缓存 Cache 等等数据，操作系统还会帮你 做这些数据的恢复操作。

所以线程的切换非常耗性能,协程的切换是由程序自身控制，因此，没有线程切换的开销，和多线程比,线程数量越多，协程的性能优势就越明显。

协程不需要多线程的锁机制。在协程中控制共享资源不加锁，只需要判断状态就好了。

简单总结:

- 进程是资源分配的单位
- 线程是操作系统调度的单位
- 进程切换需要的资源很最大，效率很低
- 线程切换需要的资源一般，效率一般（当然了在不考虑 GIL 的情况下）
- 协程切换任务资源很小，效率高
- 多进程、多线程根据cpu核数不一样,可能是并行的，但是协程是在一个线程中,所以是并发

9.3. 协程的简单创建-yield

协程是指一个过程，这个过程与调用方协作，产出有调用方提供的值。因此，生成器可以作为协程使用

```
import time

def test1():
    while True:
        print("----test1---")
        yield
        time.sleep(0.5)

def test2():
    while True:
        print("----test2---")
        yield
        time.sleep(0.5)

def main():
    t1 = test1()
    t2 = test2()
    while True:
        next(t1)
        next(t2)

if name == " main ":
    main()
```

9.4. 协程的创建-greenlet和gevent

9.4.1 greenlet

为了更好使用协程来完成多任务，python 中的 greenlet 模块对协程进行了相应封装，从而使 得切换任务变的更加简单。

1) 安装:

```
sudo pip install greenlet
```

2) greenlet 的使用

```
import greenlet
import time

def test1():
    while True:
        print("----test1---")
        gr2.switch()
        time.sleep(0.5)

def test2():
    while True:
        print("----test2---")
        gr1.switch()
        time.sleep(0.5)

g1 = greenlet.greenlet(test1)
g2 = greenlet.greenlet(test2)

#切换到 gr1 中运行
g1.switch()
```

8.4.1. gevent

1) gevent 是什么?

greenlet 已经实现了协程，但是这个还的人工切换，太麻烦了，python 还有一个比 greenlet 更强大的并且能够自动切换任务的模块 gevent。

其原理是当一个 greenlet 遇到 IO(指的是 input output 输入输出，比如网络、文件操作等)操 作时，比如访问网络，就自动切换到其他的 greenlet，等到 IO 操作完成，再在适当的时候切 换回来继续执行。

由于 IO 操作非常耗时，经常使程序处于等待状态，有了 gevent 为我们自动切换协程，就保 证总有 greenlet 在运行，而不是等待 IO

安装:

```
sudo pip install gevent
```

9.4.2 gevent 的使用

1) 顺序执行

```
import gevent

def f(n):
    for i in range(n):
        print(gevent.getcurrent(), i)

g1 = gevent.spawn(f, 5)
g2 = gevent.spawn(f, 5)
g3 = gevent.spawn(f, 5)
g1.join()
g2.join()
g3.join()
```

运行结果：

```
<Greenlet at 0x7f92438f5210: f(5)> 0
<Greenlet at 0x7f92438f5210: f(5)> 1
<Greenlet at 0x7f92438f5210: f(5)> 2
<Greenlet at 0x7f92438f5210: f(5)> 3
<Greenlet at 0x7f92438f5210: f(5)> 4
<Greenlet at 0x7f92438f5f20: f(5)> 0
<Greenlet at 0x7f92438f5f20: f(5)> 1
<Greenlet at 0x7f92438f5f20: f(5)> 2
<Greenlet at 0x7f92438f5f20: f(5)> 3
<Greenlet at 0x7f92438f5f20: f(5)> 4
<Greenlet at 0x7f92433cc048: f(5)> 0
<Greenlet at 0x7f92433cc048: f(5)> 1
<Greenlet at 0x7f92433cc048: f(5)> 2
<Greenlet at 0x7f92433cc048: f(5)> 3
<Greenlet at 0x7f92433cc048: f(5)> 4
```

可以看到，3 个协程是依次运行而不是交替运行。

2) 切换执行

```
import gevent

def f(n):
    for i in range(n):
        print(gevent.getcurrent(), i)
        #用来模拟一个耗时操作，注意不是 time 模块中的 sleep
        gevent.sleep(1)

g1 = gevent.spawn(f, 5)
g2 = gevent.spawn(f, 5)
g3 = gevent.spawn(f, 5)
```

```
g1.join()
g2.join()
g3.join()
```

运行结果：

```
<Greenlet at 0x7f651bb2c210: f(5)> 0
<Greenlet at 0x7f651bb2cf20: f(5)> 0
<Greenlet at 0x7f651b603048: f(5)> 0
<Greenlet at 0x7f651bb2c210: f(5)> 1
<Greenlet at 0x7f651bb2cf20: f(5)> 1
<Greenlet at 0x7f651b603048: f(5)> 1
<Greenlet at 0x7f651bb2c210: f(5)> 2
<Greenlet at 0x7f651bb2cf20: f(5)> 2
<Greenlet at 0x7f651b603048: f(5)> 2
<Greenlet at 0x7f651bb2c210: f(5)> 3
<Greenlet at 0x7f651bb2cf20: f(5)> 3
<Greenlet at 0x7f651b603048: f(5)> 3
<Greenlet at 0x7f651bb2c210: f(5)> 4
<Greenlet at 0x7f651bb2cf20: f(5)> 4
<Greenlet at 0x7f651b603048: f(5)> 4
```

3) 给程序打补丁

```
from gevent import monkey
import gevent
import random
import time

# 有耗时操作时需要
monkey.patch_all()# 将程序中用到的耗时操作的代码，换为 gevent 中自己实现的模块

def coroutineWork(coroutineName):
    for i in range(10):
        print(coroutineName, i)
        time.sleep(random.random())

gevent.joinall([
        gevent.spawn(coroutineWork,"work1"),
        gevent.spawn(coroutineWork,"work2")
])
```

运行结果：

```
work1 0
work2 0
work2 1
work2 2
work1 1
work2 3
```

```
work2 4
work1 2
work2 5
work1 3
work2 6
work2 7
work2 8
work2 9
work1 4
work1 5
work1 6
work1 7
work1 8
work1 9
```

9.5. 案例：并发下载

(1) 并发下载原理

```
from gevent import monkey
import gevent

# 有耗时操作时需要
monkey.patch_all()
import requests


def my_downLoad(url):
    print('GET: %s' % url)
    data = requests.get(url).text
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(my_downLoad, 'http://www.baidu.com/'),
        gevent.spawn(my_downLoad, 'http://www.hqjy.com/'),
        gevent.spawn(my_downLoad,  'http://www.blogseo.cn/'),
])
```

运行结果：

```
GET: http://www.baidu.com/
GET: http://www.hqjy.com/
GET: http://www.blogseo.cn/
114856 bytes received from http://www.baidu.com/
53178 bytes received from http://www.hqjy.com/
47410 bytes received from http://www.blogseo.cn/
```

从上能够看到是先发送的获取 baidu 的相关信息，然后依次是 hqjy、blogseo，但是收到数据的先后顺序不一定与发送顺序相同，这也就体现出了异步，即不确定什么时候会收到数据，顺序不一定。

(2) 实现其他类型文件下载

```
from gevent import monkey
import gevent


# 有 IO 才做时需要这一句
monkey.patch_all()
import requests


def myDownLoad(file_name, url):
    print('GET: %s' % url)
    data = requests.get(url).content
    with open(file_name, "wb") as f:
        f.write(data)
    print('%d bytes received from %s.' % (len(data), url))


gevent.joinall([ gevent.spawn(myDown
    Load, "1.jpg",
                "https://timgsa.baidu.com/timg?
image&quality=80&size=b9999_10000&sec=1525519093578&di=4621c05242da07d877967308b043d1ca&imgtype=
0&src=http%3A%2F%2Fbmp.skxox.com%2F201412%2F08%2F204192087.191525.jpg"),
    gevent.spawn(myDownLoad, "2.mp3", "http://sc1.111ttt.cn:8282/2017/1/11/11/304112002072.mp3"),
])
```