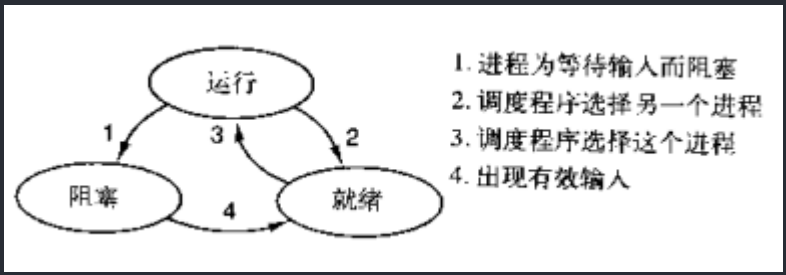
# 1. 并发编程基础

1. 并发编程本质：切换+保存状态

2. cpu正在运行一个任务，会在两种情况下切走去执行其他的任务（切换由操作系统强制控制）：

```
1. 任务发生了阻塞(发生IO操作)
2. 任务计算的时间过长(固定时间)
```




3. 第二种情况并不能提升效率，只是为了让cpu能够雨露均沾，实现看起来所有任务都被“同时”执行的效果，如果多个任务都是纯计算的，这种切换反而会降低效率

    1. yield

```
yiled可以保存状态，yield的状态保存与操作系统的保存线程状态很像，但是yield是代码级别控制的，更轻量级
send可以把一个函数的结果传给另外一个函数，以此实现单线程内程序之间的切换
```

    2. 单纯地切换反而会降低运行效率

```
#串行执行
import time
def consumer(res):
    '''任务1:接收数据,处理数据'''
    pass

def producer():
    '''任务2:生产数据'''
    res=[]
    for i in range(10000000):
        res.append(i)
    return res

start=time.time()
#串行执行
res=producer()
consumer(res) #写成consumer(producer())会降低执行效率
stop=time.time()
print(stop-start) #1.5536692142486572

#基于yield并发执行
```

```python
import time
def consumer():
    '''任务1:接收数据,处理数据'''
    while True:
        x=yield

def producer():
    '''任务2:生产数据'''
    g=consumer()
    next(g)
    for i in range(10000000):
        g.send(i)

start=time.time()
#基于yield保存状态,实现两个任务直接来回切换,即并发的效果
#PS:如果每个任务中都加上打印,那么明显地看到两个任务的打印是你一次我一次,即并发执行的.
producer()

stop=time.time()
print(stop-start) #2.0272178649902344
```

4. 第一种情况的切换。在任务一遇到io情况下，切到任务二去执行，这样就可以利用任务一阻塞的时间完成任务二的计算，效率的提升就在于此。

    1. yield并不能实现遇到io切换

```python
import time
def consumer():
    '''任务1:接收数据,处理数据'''
    while True:
        x=yield

def producer():
    '''任务2:生产数据'''
    g=consumer()
    next(g)
    for i in range(10000000):
        g.send(i)
        time.sleep(2)

start=time.time()
producer() #并发执行,但是任务producer遇到io就会阻塞住,并不会切到该线程内的其他任务去执行

stop=time.time()
print(stop-start)
```

5. 总结

```
对于单线程下，我们不可避免程序中出现io操作，但如果我们能在自己的程序中
（即用户程序级别，而非操作系统级别）控制单线程下的多个任务能在一个任务遇到io阻塞时
就切换到另外一个任务去计算，这样就保证了该线程能够最大限度地处于就绪态，即随时都可以被cpu执行的状态，
```

```
相当于我们在用户程序级别将自己的io操作最大限度地隐藏起来，从而可以迷惑操作系统，
让其看到：该线程好像是一直在计算，io比较少，从而更多的将cpu的执行权限分配给我们的线程。
```

1. 协程的本质就是在单线程下，由用户自己控制一个任务遇到io阻塞了就切换另外一个任务去执行，以此来提升效率。为 了实现它，我们需要找寻一种可以同时满足以下条件的解决方案：

```
1. 可以控制多个任务之间的切换，切换之前将任务的状态保存下来，以便重新运行时，
   可以基于暂停的位置继续执行。
2. 作为1的补充：可以检测io操作，在遇到io操作的情况下才发生切换
```

# 2. 协程介绍

1. 定义：是单线程下的并发，又称微线程，纤程；
   协程是一种用户态的轻量级线程，即协程是由用户程序自己控制调度的

2. 注意：

```
1. python的线程属于内核级别的，即由操作系统控制调度（如单线程遇到io或执行时间过长就会被迫交出cpu执行权限，
   切换其他线程运行）
2. 单线程内开启协程，一旦遇到io，就会从应用程序级别（而非操作系统）控制切换，
   以此来提升效率（！！！非io操作的切换与效率无关）
```

3. 对比操作系统控制线程的切换，用户在单线程内控制协程的切换

   优点如下：

```
1. 协程的切换开销更小，属于程序级别的切换，操作系统完全感知不到，因而更加轻量级
2. 单线程内就可以实现并发的效果，最大限度地利用cpu
```

   缺点如下：

```
1. 协程的本质是单线程下，无法利用多核，可以是一个程序开启多个进程，
   每个进程内开启多个线程，每个线程内开启协程
2. 协程指的是单个线程，因而一旦协程出现阻塞，将会阻塞整个线程
```

4. 总结协程特点

```
1. 必须在只有一个单线程里实现并发
2. 修改共享数据不需加锁
3. 用户程序里自己保存多个控制流的上下文栈
4. 附加：一个协程遇到IO操作自动切换到其它协程
   （如何实现检测IO，yield、greenlet都无法实现，就用到了gevent模块（select机制））
```

# 3. Greenlet

1. greenlet简介
   当你创建一个greenlet时，它得到一个开始时为空的栈；当你第一次切换到它时，它会执行指定的函数，

这个函数可能会调用其他函数、切换跳出greenlet等等。当最终栈底的函数执行结束出栈时，这个greenlet的栈又变成空的，这个greenlet也就死掉了。greenlet也会因为一个未捕捉的异常死掉。

```
from greenlet import greenlet

def eat(name):
    print('%s eat 1' %name)
    g2.switch('egon')
    print('%s eat 2' %name)
    g2.switch()
def play(name):
    print('%s play 1' %name)
    g1.switch()
    print('%s play 2' %name)

g1=greenlet(eat)
g2=greenlet(play)

g1.switch('egon')#可以在第一次switch时传入参数，以后都不需要
```

2. 单纯的切换（在没有io的情况下或者没有重复开辟内存空间的操作），反而会降低程序的执行速度

```
#顺序执行
import time
def f1():
    res=1
    for i in range(100000000):
        res+=i

def f2():
    res=1
    for i in range(100000000):
        res*=i

start=time.time()
f1()
f2()
stop=time.time()
print('run time is %s' %(stop-start)) #10.985628366470337

#切换
from greenlet import greenlet
import time
def f1():
    res=1
    for i in range(100000000):
        res+=i
        g2.switch()

def f2():
    res=1
    for i in range(100000000):
```

```
        res*=i
        g1.switch()

start=time.time()
g1=greenlet(f1)
g2=greenlet(f2)
g1.switch()
stop=time.time()
print('run time is %s' %(stop-start)) # 52.763017892837524
```

# 4. Gevent介绍

1. 实现原理：gevent是第三方库，通过greenlet实现协程
2. 基本思想：

```
当一个greenlet遇到IO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
```

3. 参数介绍

```
#用法
g1=gevent.spawn(func,1,,2,3,x=4,y=5)创建一个协程对象g1，
spawn括号内第一个参数是函数名，如eat，后面可以有多个参数，可以是位置实参或关键字实参，都是传给函数eat的
g2=gevent.spawn(func2)
g1.join() #等待g1结束
g2.join() #等待g2结束
#或者上述两步合作一步：gevent.joinall([g1,g2])
g1.value#拿到func1的返回值

注意：

time.sleep(2)或其他的阻塞,gevent是不能直接识别的需要用下面一行代码，

打补丁,就可以识别了

from gevent import monkey;monkey.patch_all()必须放到被打补丁者的前面，

如time，socket模块之前

或者我们干脆记忆成：要用gevent，需要将from gevent import monkey;

monkey.patch_all()放到文件的开头
```

4. 示例:遇到IO阻塞时会自动切换任务

```
import gevent
import time
```

```python

def eat(name):
    print('%s eat 1'%name)
    gevent.sleep(2)
    print('%s eat 2'%name)

def play(name):
    print('%s play 1'%name)
    gevent.sleep(2)
    print('%s play 2'%name)

start=time.time()
g1=gevent.spawn(eat,'egon')
g2=gevent.spawn(play,name='lex')

gevent.joinall([g1,g2])

print('主',time.time()-start)

# egon eat 1
# lex play 1
# egon eat 2
# lex play 2
# 主 2.0011146068573
```

5. 实际代码里，我们不会用gevent.sleep()去切换协程，而是在执行到IO操作时，gevent自动切换

```python
from gevent import monkey;monkey.patch_all()

import gevent,time

def eat(name):
    print('%s eat 1'%name)
    time.sleep(1)
    print('%s eat 2'%name)

def play(name):
    print('%s play 1'%name)
    time.sleep(1)
    print('%s play 2'%name)

g1=gevent.spawn(eat,'lex')
g2=gevent.spawn(play,'egon')

gevent.joinall([g1,g2])
print('主')
```