- 线程的信号量

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


def func(index, sem):
    sem.acquire()
    print(index)
    time.sleep(1)
    sem.release()


if __name__ == '__main__':
    sem = Semaphore(5)
    for i in range(10):
        Thread(target=func, args=(i, sem)).start()
```

- 线程的事件

```python
from threading import Event, Thread
import time
import random


def check(e):
    print("开始检测数据库的连接")
    time.sleep(random.randint(1, 5))  # 检测数据库的连接（所需时间）
    e.set()  # flag = True


def connect(e):
    for i in range(3):  # 0 1 2
        e.wait(1.5)   # 阻塞1.5      # 超过1.5秒就算超时
        if e.is_set():
            print("数据库连接成功")
            break
        else:
```

```
            print("尝试连接数据库第{}次失败".format(i + 1))


e = Event()  # 默认 flag = Fales
Thread(target=connect, args=(e, )).start()
Thread(target=check, args=(e, )).start()
```

- 线程池

```
from concurrent.futures import ThreadPoolExecutor
import time

def func(i):
    print("thread", i)
    time.sleep(1)
    print("thread {} end".format(i))

tp = ThreadPoolExecutor(5)
# 提交任务

for i in range(20):
    tp.submit(func, i)  # 注意这里传的参数，不是元组
tp.shutdown()
```

如何获取返回值?

```
from concurrent.futures import ThreadPoolExecutor
import time

def func(i):
    print("thread", i)
    time.sleep(1)
    print("thread {} end".format(i))
    # 获取返回值
    return i

tp = ThreadPoolExecutor(5)
# 提交任务

for i in range(20):
```

```
    ret = tp.submit(func, i)  # 注意这里传的参数，不是元组
    print(ret.result())
tp.shutdown()
```

上述的程序虽然能拿到返回值，但是结果是一个同步的效果，我们希望是异步的

```
from concurrent.futures import ThreadPoolExecutor
import time

def func(i):
    # print("thread", i)
    time.sleep(1)
    # print("thread {} end".format(i))
    # 获取返回值
    return i

tp = ThreadPoolExecutor(5)
# 提交任务
ret_li = []
for i in range(20):
    ret = tp.submit(func, i)  # 注意这里传的参数，不是元组
    ret_li.append(ret)
for ret in ret_li:
    print(ret.result())
tp.shutdown()
```

协程

- 进程：计算机中最小的资源分配单位
- 线程：计算机中能被cpu调度的最小单位

  现在有一个问题：

  能开启的线程和进程是有限的，但是我们要处理的任务是无限的。

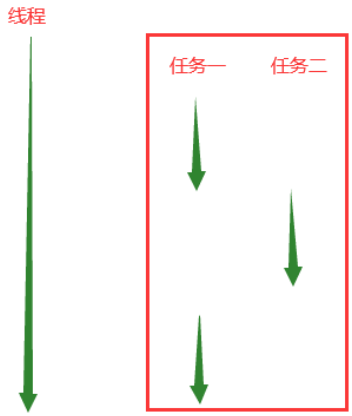
  比如我现在有5万个任务要执行，难道我去开5万个线程吗？

  然后假设我开了300个线程，但是这300个线程都阻塞了，我仍然没有充分利用cpu

  （理想的情况是开启一个线程，没有IO只有计算，这叫充分利用cpu）

  （但大多数的情况不是这样的，多多少少会有一些阻塞情况）
- 协程：又叫纤程
  - 一条线程在多个任务之间来回切换

- 切换这个动作是浪费时间的
- 对于cpu、操作系统来说协程是不存在的
- 他们只能看到线程



把一个线程的工作分成了好几个，可以在多个任务来回切换的现象称为协程

举个例子：

任务来回切换

```
# 写一个最简单的生产者消费者模型
def consumer():
    g = producer()
    for num in g:
        print(num)

def producer():
    for i in range(1000):
        yield i


consumer()

# 那如何更好的利用协程？
# 协程能把一个线程的执行明确的区分开
# 两个任务，  协程帮助我记住哪个任务执行到哪个位置上了，并且实现安全的切换
# 一个任务不得不陷入阻塞了，在这个任务阻塞的过程中切换到另一个任务继续执行
# 你的程序只要还有任务需要执行   你的当前线程永远不会阻塞
# 利用协程在多个任务陷入阻塞的时候进行切换，来保证一个线程在处理多个任务的时候总是在忙碌
# 能够更加充分的利用cpu，抢占更多的时间片
```

```
# 无论是进程还是线程都是由操作系统来切换的，开启过多的线程、进程会给操作系统的调度增加负担
# 如果我们是使用的协程，协程在程序之间的切换操作系统感知不到
# 无论开启多少个协程对操作系统来说总是一个线程在执行
# 操作系统的调度不会有任务的压力
# 协程的本质就是一条线程，所以完全不会产生数据安全的问题
```

- 协程模块
    - greenlet （gevent的底层，主要控制协程的切换）
    - gevent（需要安装，直接用，功能比greenlet更全面）

```
from greenlet import greenlet
import time


def eat():
    print("eating  1")
    g2.switch()
    print("eating  2")
    g2.switch()


def play():
    print("playing 1")
    g1.switch()
    print("playing 2")


g1 = greenlet(eat)
g2 = greenlet(play)
g1.switch()
```

优化一下

```
# from greenlet import greenlet
import time
import gevent


def eat():
    print("eating  1")
```

```
    gevent.sleep(1)
    print("eating  2")


def play():
    print("playing 1")
    gevent.sleep(1)
    print("playing 2")


g1 = gevent.spawn(eat)   # 自动检测阻塞事件，遇见阻塞就会进行切换，有些阻塞不认识
g2 = gevent.spawn(play)
g1.join()
g2.join()
```

在优化一下

```
# from greenlet import greenlet

from gevent import monkey
monkey.patch_all()
import time
import gevent


def eat():
    print("eating  1")
    time.sleep(1)
    print("eating  2")


def play():
    print("playing 1")
    time.sleep(1)
    print("playing 2")

```

```
g1 = gevent.spawn(eat)   # 自动检测阻塞事件，遇见阻塞就会进行切换，有些阻塞不认识
g2 = gevent.spawn(play)
g1.join()
g2.join()
```