协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

**def** **A**():
**print** '1'
**print** '2'
**print** '3'
**def** **B**():
**print** 'x'
**print** 'y'
**print** 'z'

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

1
2
x
y
3
z

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，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。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

**import** time
**def** **consumer**():
r = ''
**while** True:
n = **yield** r
**if** **not** n:
**return**
print('[CONSUMER] Consuming %s...' % n)
time.sleep(1)
r = '200 OK'
**def** **produce**(c):
c.next()
n = 0
**while** n < 5:
n = n + 1
print('[PRODUCER] Producing %s...' % n)
r = c.send(n)
print('[PRODUCER] Consumer return: %s' % r)
c.close()
**if** \_\_name\_\_=='\_\_main\_\_':
c = consumer()
produce(c)

执行结果：

[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK

注意到consumer函数是一个generator（生成器），把一个consumer传入produce后：

首先调用c.next()启动生成器；

然后，一旦生产了东西，通过c.send(n)切换到consumer执行；

consumer通过yield拿到消息，处理，又通过yield把结果传回；

produce拿到consumer处理的结果，继续生产下一条消息；

produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

作者：陈果果果果果栋

链接：<http://www.zhihu.com/question/32218874/answer/55469714>

来源：知乎

著作权归作者所有，转载请联系作者获得授权。

协程是啥

首先我们得知道协程是啥？协程其实可以认为是比线程更小的执行单元。为啥说他是一个执行单元，因为他自带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。

协程和线程差异

那么这个过程看起来比线程差不多哇。其实不然 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。但是协程的切换只是单纯的操作CPU的上下文，所以一秒钟切换个上百万次系统都抗的住。

协程的问题

但是协程有一个问题，就是系统并不感知，所以操作系统不会帮你做切换。那么谁来帮你做切换？让需要执行的协程更多的获得CPU时间才是问题的关键。

笔者知道协程的实现相关的

目前的协程框架一般都是设计成 1:N 模式。所谓 1:N 就是一个线程作为一个容器里面放置多个协程。那么谁来适时的切换这些协程？答案是有协程自己主动让出CPU，也就是每个协程池里面有一个调度器，这个调度器是被动调度的。意思就是他不会主动调度。而且当一个协程发现自己执行不下去了(比如异步等待网络的数据回来，但是当前还没有数据到)，这个时候就可以由这个协程通知调度器，这个时候执行到调度器的代码，调度器根据事先设计好的调度算法找到当前最需要CPU的协程。切换这个协程的CPU上下文把CPU的运行权交个这个协程，直到这个协程出现执行不下去需要等等的情况，或者它调用主动让出CPU的API之类，触发下一次调度。对的没错就是类似于 领导人模式

那么这个实现有没有问题？

其实是有问题的，假设这个线程中有一个协程是CPU密集型的他没有IO操作，也就是自己不会主动触发调度器调度的过程，那么就会出现其他协程得不到执行的情况，所以这种情况下需要程序员自己避免。这是一个问题，假设业务开发的人员并不懂这个原理的话就可能会出现问题。

最后讲讲协程的好处

在IO密集型的程序中由于IO操作远远小于CPU的操作，所以往往需要CPU去等IO操作。同步IO下系统需要切换线程，让操作系统可以再IO过程中执行其他的东西。这样虽然代码是符合人类的思维习惯但是由于大量的线程切换带来了大量的性能的浪费，尤其是IO密集型的程序。

所以人们发明了异步IO。就是当数据到达的时候触发我的回调。来减少线程切换带来性能损失。但是这样的坏处也是很大的，主要的坏处就是操作被 “分片” 了，代码写的不是 “一气呵成” 这种。 而是每次来段数据就要判断 数据够不够处理哇，够处理就处理吧，不够处理就在等等吧。这样代码的可读性很低，其实也不符合人类的习惯。

但是协程可以很好解决这个问题。比如 把一个IO操作 写成一个协程。当触发IO操作的时候就自动让出CPU给其他协程。要知道协程的切换很轻的。协程通过这种对异步IO的封装 既保留了性能也保证了代码的 容易编写和可读性。在高IO密集型的程序下很好。但是高CPU密集型的程序下没啥好处。