引言

1. 几个概念

同步vs异步

同步：调用方发出一个请求时，在没有得到被调用发返回结果之前，这个请求不会得到响应，一旦调用方得到返回结果，请求得到了响应。简单概括为调用者主动等待被调用方返回结果。

异步：和同步相反，异步是调用方发出请求之后，不会立刻得到反馈结果，而是被调用方通过状态、通知来通知调用者，或者通过回调函数来处理这个调用。 I 例如：阻塞，非阻塞 I

串行vs并行

串行：指的是执行多个任务时，各个任务按照顺序执行，完成一个任务后才能进行下一个任务；

并行：多个任务可以同时执行

例如：发试卷，班长发和第一排下发

并行vs并发

时间上：并行指的是多个事件在同一时刻发生；并发指的是多个事件在同一个时间间隔发生

作用点：并行是作用于多个实体上的多个事件，并发是作用于一个实体上的多个事件；并行是多个处理器处理多个任务，并发是一个处理器处理多个任务

2. cpu密集型vsI/O密集型（CPU-bound VS I/O-bound）

cpu密集型

CPU bound的程序一般而言CPU占用率相当高，而硬盘/内存相对较低。CPU 要读/写 I/O (硬盘/内存)，I/O在很短的时间就可以完成。这可能是因为任务本身不太需要访问I/O设备，也可能是因为程序是多线程实现因此屏蔽掉了等待I/O的时间。

计算密集型：大量消耗CPU的数学与逻辑运算，也就是我们这里说的平行计算

I/O密集型

I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。大部分的状况是 CPU 在等 I/O (硬盘/内存) 的读/写，此时 CPU Loading 不高。这可能是因为任务本身需要大量I/O操作，却没有充分利用处理器能力

IO密集型：读取文件，读取网络套接字频繁

思考;

Python 的线程。在 Python 中，哪一种多线程的程序表现得更好，I/O 密集型的还是计算 密集型的?

3. 并发三个层次

1.低阶：

这个并发的级别主要是针对于操作系统底层来实现并发。 这一层次的并发主要是直接使用原子操作的并发，是针对库编写者而不是针对应用程序开发者的，因为这很容易出错，并且调试起来非常困难。尽管Python并发的实现通常是使用低级操作构建的，但是Python不支持这种并发。

2.中阶：

这个并发的级别几乎所有的语言都支持。 这一层次的并发并不使用显式的原子操作，但是会使用显式的锁。Python提供了对这个层次编程的支持，例如线程，进程，信号量，线程锁定和多处理。锁定等类。通常使用这种级别的并发支持。

3.高阶：

一些现代语言开始支持高级并发 这一层次的并发没有显式的原子操作和显式的锁（锁定和原子操作很可能发生在底层，但我们不必关心它们），python语言中，对中阶实现的过程进行了封装，出现的一些高阶并发框架，如协程，queue, 多进程多线程等处理。concurrent.futures, asyinico, gevent等以支持高级并发

进程

CPU在同一时刻只能处理一个任务,只是因为cpu执行速度很快, cpu在各个任务之间来回的进行切换。

进程的概念：正在进行的一个过程或者说一个任务，而负责执行任务的则是CPU，进程本身是 一个抽象的概念,即进程就是一个过程、一个任务。

一、进程

进程 是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。

进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体

如何做到 **并发** 的： CPU在同一时刻只能处理一个任务,只是因为cpu执行速度很快, CPU在各个任务之间来回的进行切换，并在切换的过程当中保存当前进程的执行状态。

1. 创建单进程

1.1 fork()函数

Unix/Linux/Mac操作系统都可以使用fork()函数来创建子进程，分别在父进程和子进程内返回

```Python
import os  #  导入os模块

print ('当前进程的ID是：%s' % os.getpid()) # os.getpid()返回的是进程的id不是线程
ID = os.fork()  # 创建子进程，并返回进程的id，父进程返回的是父进程的id，子进程返回的
if ID == 0:
    print ('这是子进程，ID是：%s。。父进程ID是：%s' % (os.getpid(), os.getppid(
else:
    print ('这是父进程，ID是：%s' % os.getpid())
```

上面是操作系统层面创建进程的一个示例，实际中基本不用。

1.2 multiprocessing.Process类

```Python
#coding: utf-8
import os

from multiprocessing import Process
import os

# 进程要执行的代码
def run_proc(name):
    print('子线程 %s ，ID是：%s' % (name, os.getpid()))


print('当前线程(父线程)的ID是：  %s' % os.getpid())
p = Process(target=run_proc, args=('test',))  # 创建Process的实例，并传入子线
p.start()  # 子线程开始执行
p.join()  # join方法用于线程间的同步，等线程执行完毕后再往下执行
print('子线程执行完毕，回到主线程%s' % os.getpid())
```

二、python 多进程

计算1+……+n的和

```python
#coding: utf-8
import time
def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))
if __name__ == '__main__':
    start = time.time()
    myflow(5000000)
    myflow(6000000)
    end = time.time()
    print(end-start)
```

如上面的例子，我们进行两个计算任务，我们想加速, 多进程（进程池）实现。 python中实现多进程的方式有multiprocessing，concurrent.futures.ThreadPoolExecutor

1. multiprocessing

1.1 创建多进程

> - start 通过调用start方法启动进程，跟线程差不多
> - run：如果在创建Process对象的时候不指定target，那么就会默认执行Process的run方法
> - join 阻塞当前进程，直到调用join方法的那个进程执行完，再继续执行当前进程
> - mutilprocess.setDaemon(True) 守护进程就是不阻挡主程序退出，自己干自己的

```python
#coding: utf-8

from multiprocessing import Process
import time
def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))

if __name__ == "__main__":
        start = time.time()
        print("main process run...")
        p1 = Process(target=my_flow, args=(5000000, ))       #target:指定进
        p2 = Process(target=my_flow, args=(6000000, ))

        p1.start()
        p2.start()
        p1.join()
        p2.join()
        #p1.join()
        #p2.join()
        print("main process runned all lines...")
        end = time.time()
        print('time cost {}'.format(end-start))
```

1.2 使用进程池

如果我们要启动大量的子进程且操作函数相同, 我们可以使用进程池的方式创建进程：

> - apply_async(func[, args[, kwds[, callback]]]) 它是非阻塞， apply(func[, args[, kwds]])是阻塞的
> - close() 关闭pool，使其不在接受新的任务。
> - terminate() 结束工作进程，不在处理未完成的任务。
> - join() 主进程阻塞，等待子进程的退出， join方法要在close或terminate之后使用

```python
#coding: utf-8
import multiprocessing
import time

def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))

if __name__ == "__main__":
    start = time.time()
    pool = multiprocessing.Pool(processes = 3)
    for i in [500000, 600000]:
        print( "第 %d 个 task" %(i))
        pool.apply(my_flow, (i, ))   #维持执行的进程总数为processes，当一个进程
        # result = pool.apply(my_flow, (i, ))   #维持执行的进程总数为processe
        # pool.apply_async(my_flow, (i, ))   #维持执行的进程总数为processes，

    pool.close()
    pool.join()   #调用join之前，先调用close函数，否则会出错。执行完close后不会有
    end = time.time()
    print('time cost {}'.format(end-start))
```

其他：

- python进程间传递消息，一般的情况是Queue来传递
- Event提供一种简单的方法，可以在进程间传递状态信息。事件可以切换设置和未设置状态

2.concurrent.futures.ProcessPoolExecutor

```python
import os
import time
from concurrent.futures import ProcessPoolExecutor as Pool
from concurrent.futures import as_completed

def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))
    return (n, total)

if __name__ == "__main__":
    # 创建多个进程去统计目录中所有文件的行数和字符数
    start_time = time.time()
    result_list = []
    with Pool(max_workers=3) as executor:
        future_tasks = [executor.submit(my_flow, item) for item in [500000

    result_list = as_completed(future_tasks)
    for r in result_list:
        if r.done():
            print(r.result())


    end_time = time.time()
    print("used time is ", end_time - start_time)
```

线程

一、线程

1。 线程概念

线程 是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源

2. 全局解释器锁

所有访问python对象的进程都将被一个全局锁序列化，这是因为大部分解释程序代码和第三方C代码都不是线程安全的，需要保护

Python 代码的执行是由Python虚拟机(又名解释器主循环)进行控制的。Python在设计时是这样考虑的：

在主循环中同时只能有一个控制线程在执行,就像单核CPU系统中的多进程一样。

内存中可以有许多程序,但是在任意给定时刻只能有一个程序在运行。尽管Python解释器中可以运行多个线程，但是在任意给定时刻只有一个线程会被解释器执行。

对Python虚拟机的访问是由全局解释器锁(GIL)控制的。这个锁就是用来保证同时只能有一个线程运行的。

```
在多线程环境中,Python虚拟机将按照下面所述的方式执行：
1.设置 GIL。
2.切换进一个线程去运行。
3.执行下面操作之一：
    a.指定数量的字节码指令。
    b.线程主动让出控制权(可以调用 time.sleep(0)来完成)。
4.把线程设置回睡眠状态(切换出线程)。
5.解锁 GIL。
6.重复上述步骤。
```

对于任意面向I/O的Python例程(调用了内置的操作系统C代码的那种), GIL会在I/O调用前被释放，以允许其他线程在I/O执行的时候运行。

而对于那些没有太多I/O操作的代码而言,更倾向于在该线程整个时间片内始终占有处理器(和GIL)。

换句话说：I/O密集型的Python程序要比计算密集型的代码能够更好地利用多线程环境

二、多线程

Python提供了多个模块来支持多线程编程，包括thread、threading和Queue模块等。可以使用thread和threading模块来创建与管理线程。

thread模块提供了基本的线程和锁定支持;而threading模块提供了更高级别、功能更全面的线程管理。使用Queue模块,用户可以创建一个队列数据结构,用于在多线程之间进行共享。

我们将分别来查看这几个模块, 并给出几个例子和中等规模的应用。

1. thread模块

函数式：调用thread模块中的start*new*thread()函数来产生新线程。语法如下：

thread.start*new*thread (function, args[, kwargs]) 参数说明:

function - 线程函数。 args - 传递给线程函数的参数,他必须是个tuple类型。 kwargs - 可选参数。

```python
import thread
import time

# 为线程定义一个函数
def print_time(threadName, delay):
    """
    threadName 线程, delay 延迟时间
    """
    count = 0
    for i in range(4):
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))


# 创建两个线程
try:
    thread.start_new_thread(print_time, ("线程-1", 2,))
    thread.start_new_thread(print_time, ("线程-2", 4,))
except:
    print "Error: unable to start thread"
```

它对于进程何时退出没有控制, thread 模块拥有的同步原语很少

推荐 使用更高级别的 threading 模块,而不使用 thread 模块。

> - threading模块更加先进，有更好的线程支持，而且使用thread模块里的属性有可能会与threading出现冲突。
> - 低级别的thread模块拥有的同步原语很少，而threading模块则有很多。
> - 它对于进程何时退出没有控制。当主线程结束时，所有其他线程也都强制结束,不会发出警告或者进行适当的清理。如前所述，至少threading模块能确保重要的子线程在进程退出前结束。
> - 只建议那些想访问线程的更底层级别的专家使用 thread 模块。为了强调这一点, 在 Python3 中该模块被重命名为_thread。你创建的任何多线程应用都应该使用 threading 模 块或其他更高级别的模块。

2. threading模块

下面是threading 模块里所有的对象：

threading模块对象	描述
Thread	表示一个线程的执行的对象
Lock	锁原语对象（跟thread模块里的锁对象相同）
RLock	可重入锁对象。使单线程可以再次获得已经获得了的锁（递归锁定）
Condition	条件变量对象能让一个线程停下来，等待其他线程满足了某个“条件”。如状态的改变或值的改变
Event	通用的条件变量。多个线程可以等待某个时间的发生，在事件发生后，所有的线程都被激活
Semaphore	为等待锁的线程提供一个类似"等候室"的结构
BoundedSemaphore	与Semaphore类似，只是它不允许超过初始值
Timer	与thread类似，只是它要等待一段时间后才开始运行

2.1 守护线程

另一个避免使用 thread 模块的原因是，它不支持守护线程。当主线程退出时，所有的子线程不论它们是否还在工作，都会被强行退出。有时，我们并不期望这种行为，这时，就引入了守护线程的概念

threading 模块支持守护线程，它们是这样工作的:

守护线程一般是一个等待客户请求的服务器，如果没有客户出请求，它就在那等着。如果你设定一个线程为守护线程，就表示你在说这个线程是不重要的，在进程退出的时候，不用等待这个线程退出。

可以通过daemon属性来设置守护线程

2.1 Thread 类

threading的Thread类是你主要的运行对象。它有很多thread模块里没有的函数。

函数	描述
start()	开始线程的执行
run()	定义线程的功能的函数（一般会被子类重写）
join(timeout=None)	程序挂起，直到线程结束；如果给了timeout，则最多阻塞timeout秒
getName()	返回线程的名字
setName(name)	设置线程的名字
isAlive()	布尔标志，表示这个线程是否还在运行中
isDaemon()	返回线程的daemon标志
setDaemon(daemonic)	把线程的daemon标志设为daemonic（一定要在调用start()函数前调用）

Lock & RLock

原语锁定是一个同步原语，状态是锁定或未锁定。两个方法acquire()和release() 用于加锁和释放锁。

Event

事件用于在线程间通信。一个线程发出一个信号，其他一个或多个线程等待。Event 通过通过内部标记来协调多线程运 。 方法 wait() 阻塞线程执 ， 直到标记为 True。 set() 将标记设为 True，clear() 更

改标记为 False。isSet() 用于判断标记状态。

Condition

条件变量和 Lock 参数一样，也是一个，也是一个同步原语，当需要线程关注特定的状态变化或事件的发生时使用这个锁定。

```python
from time import ctime
import threading

def coding(language):
    for i in range(5):
        print('I\'m coding ',language, ' program at ', ctime() )

def music():
    for i in range(5):
        print('I\'m listening music at ', ctime())

if __name__ == '__main__':

    print('thread %s is running...' % threading.current_thread().name)

    thread_list = []
    t1 = threading.Thread(target=coding, args=('Python',))
    t2 = threading.Thread(target=music)
    thread_list.append(t1)
    thread_list.append(t2)

    for t in thread_list:
        t.setDaemon(True)  # 设置为守护线程
        t.start()
        t.join()  # 在这个子线程完成运行之前，主线程将一直被阻塞

    print('thread %s ended.' % threading.current_thread().name)
```

3. concurrent.futures

Python标准库为我们提供了threading和multiprocessing模块编写相应的多线程/多进程代码，但是当项目达到一定的规模，频繁创建/销毁进程或者线程是非常消耗资源的，这个时候我们就要编写自己的线程池/进程池，以空间换时间。

但从Python3.2开始，标准库为我们提供了concurrent.futures模块，它提供了ThreadPoolExecutor和ProcessPoolExecutor两个类(python2 需要安装)，实现了对threading和multiprocessing的进一步抽象，对编写线程池/进程池提供了直接的支持。

Python

```python
3.2 concurrent.futures.ThreadPoolExecutor
# 指定目录下所有文件信息（行数和字符数）
import os
import time
from concurrent.futures import ThreadPoolExecutor as Pool
from concurrent.futures import as_completed


def get_file_list(path):
    # 获取目录下以.py和.txt结尾的文件list
    file_list = []
    for root, dirs, files in list(os.walk(path)):
        for i in files:
            if i.endswith('.txt') or i.endswith('.py'):
                file_list.append(root + "/" + i)
    return file_list


def read_file_info(file_path):
    # 统计每个文件中行数和字符数，并返回
    fp = open(file_path, encoding="utf-8")
    content = fp.readlines()
    fp.close()
    lines = len(content)
    char_num = 0
    for i in content:
        char_num += len(i.strip('\n'))
    return lines, char_num, file_path


def out(list1, write_file_path):
    # 将统计结果写入结果文件中
    file_lines = 0
    char_num = 0
    fp = open(write_file_path, 'a', encoding="utf-8")

    for f in list1:
        ret = f.done()
        if ret:
            f_ret = f.result()
            fp.write(f_ret[2] + " 行数: " + str(f_ret[0]) + " 字符数: " + st
            file_lines += f_ret[0]
            char_num += f_ret[1]


    fp.close()
    print("信息汇总:  {}行,  {}字符".format(file_lines, char_num))
```

```
48  if __name__ == "__main__":
49      # 创建多个进程去统计目录中所有文件的行数和字符数
50      start_time = time.time()
51      file_path = "/Users/liuzhanbing/Documents/分支文件"
52      file_list = get_file_list(file_path)
53      result_list = []
54      with Pool(max_workers=3) as executor:
55          future_tasks = [executor.submit(read_file_info, file) for file in
56
57          result_list = as_completed(future_tasks)
58
59      write_file_path = r"/Users/liuzhanbing/Documents/分支文件/res.txt"
60      out(result_list, write_file_path)
61      end_time = time.time()
62      print("used time is ", end_time - start_time)
63
```

协程模块

协程

协程（Coroutine），又可以称作为微线程。协程是一种用户态轻量级的线程。

线程是系统级别，是由系统来统一调度的；而协程是程序级别的，由开发者根据自己的需要来调度。同一线程下的一段代码<1>执行着执行着就可以中断， 然后跳去执行另一段代码，当再次回来执行代码块<1>的时候，接着从之前中断的地方开始执行。这个就是协程。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复之前保存的寄存器上下文和栈。 所以协程能保留上一次调用时的状态，当每次过程重新载入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

协程基于generator，Python3中内置了异步IO。遇到IO密集型的业务时，总是很费时间啦，多线程加上协程，你磁盘在那该读读该写写，我还能去干点别的。在WEB应用中效果尤为明显

下面我们来具体学习协程。

1.生成器函数-> 协程

在 Python2.2 中，第一次引入了生成器，生成器实现了一种惰性、多次取值的方法，此时还是通过 next 构造生成迭代链或 next 进行多次取值。

直到在Python2.5 中，yield 关键字被加入到语法中，这时生成器有了记忆功能，下一次从生成器中取值可以恢复到生成器上次 yield 执行的位置。

之前的生成器都是关于如何构造迭代器，在 Python2.5 中生成器还加入了 send 方法，与 yield 搭配使

用。

我们发现，此时，生成器不仅仅可以 yield 暂停到一个状态，还可以往它停止的位置通过 send 方法传入一个值改变其状态。

1.1 生成器函数定义协程：

```python
#! -*- coding: utf-8 -*-

import inspect

# 协程使用生成器函数定义：定义体中有yield关键字。
def simple_coroutine():
    print('-> coroutine started')
    # yield 在表达式中使用；如果协程只需要从客户那里接收数据，yield关键字右边不需要加
    x = yield
    print('-> coroutine received:', x)


my_coro = simple_coroutine()
my_coro # 和创建生成器的方式一样，调用函数得到生成器对象。
# 协程处于 GEN_CREATED (等待开始状态)
print(inspect.getgeneratorstate(my_coro))

my_coro.send(None)
# 首先要调用next()函数，因为生成器还没有启动，没有在yield语句处暂停，所以开始无法发送数
# 发送 None 可以达到相同的效果 my_coro.send(None)
next(my_coro)
# 此时协程处于 GEN_SUSPENDED (在yield表达式处暂停)
print(inspect.getgeneratorstate(my_coro))

# 调用这个方法后，协程定义体中的yield表达式会计算出42；现在协程会恢复，一直运行到下一个
my_coro.send(42)
print(inspect.getgeneratorstate(my_coro))


GEN_CREATED
-> coroutine started
GEN_SUSPENDED
-> coroutine received: 42

# 这里，控制权流动到协程定义体的尾部，导致生成器像往常一样抛出StopIteration异常
Traceback (most recent call last):
  File "/Users/gs/coroutine.py", line 18, in <module>
    my_coro.send(42)
StopIteration
```

send方法的参数会成为暂停yield表达式的值，所以，仅当协程处于暂停状态是才能调用send方法。

如果协程还未激活（GEN*CREATED 状态）要调用next(my*coro) 激活协程，也可以调用 my_coro.send(None)

协程状态： 协程有四个状态，可以使用inspect.getgeneratorstate()函数确定：

> - GEN_CREATED # 等待开始执行
> - GEN_RUNNING # 解释器正在执行（只有在多线程应用中才能看到这个状态）
> - GEN_SUSPENDED # 在yield表达式处暂停
> - GEN_CLOSED # 执行结束

思考：有什么好的方式来预激活协程？（装饰器）

2. 协程异常处理和终止

```python
#! -*- coding: utf-8 -*-

from functools import wraps

def coroutinue(func):
    '''
    装饰器：  向前执行到第一个`yield`表达式，预激`func`
    :param func: func name
    :return: primer
    '''

    @wraps(func)
    def primer(*args, **kwargs):
        # 把装饰器生成器函数替换成这里的primer函数；调用primer函数时，返回预激后的生
        gen = func(*args, **kwargs)
        # 调用被被装饰函数，获取生成器对象
        next(gen)  # 预激生成器
        return gen  # 返回生成器
    return primer


@coroutinue
def averager():
    # 使用协程求平均值
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count

coro_avg = averager()
print(coro_avg.send(40))
print(coro_avg.send(50))
print(coro_avg.send('123')) # 由于发送的不是数字，导致内部有异常抛出。
```

从Python2.5 开始，我们可以在生成器上调用两个方法，显式的把异常发给协程。 这两个方法是throw和close。

generator.throw(exc*type[, excvalue*[, traceback]]) 这个方法使生成器在暂停的yield表达式处抛出指定的异常。如果生成器处理了抛出的异常，代码会向前执行到下一个yield表达式，而产出的值会成为调用throw方法得到的返回值。 如果没有处理，则向上冒泡，直接抛出， 传到调用方的上下文中。

generator.close() 使得生成器在暂停的yield表达式处抛出GeneratorExit异常。如果生成器没有处理这

个异常或者抛出StopIteration异常（一般指运行到结尾），调用方不会报错。 如果收到StopIteration异常， 生成器一定不能产出值，否则解释器会抛出RuntimeError异常。

3. 协程返回值

Python

```python

#! -*- coding: utf-8 -*-

from collections import namedtuple

Result = namedtuple('Result', 'count average')

def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            break  # 为了返回值，协程必须正常终止；这里是退出条件
        total += term
        count += 1
        average = total/count
    # 返回一个namedtuple，包含count和average两个字段。在python3.3前，如果生成器返
    return Result(count, average)

coro_avg = averager()
next(coro_avg)
coro_avg.send(10)
coro_avg.send(20)

try:
    coro_avg.send(None)
except StopIteration as exc:
    result = exc.value
```

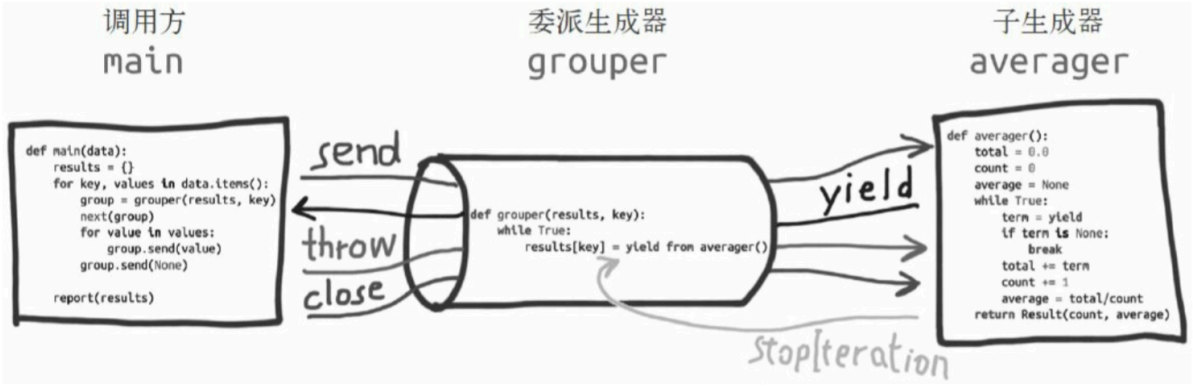
发送None会导致协程结结束。

```python
def generator_word():
    for c in 'ABCDE':
        yield c
    for i in range(1, 3):
        yield i
list(generator_word())
['A', 'B', 'C', 'D', 'E', 1, 2]


def generator_word():
    yield from 'ABCDE':
    yield from range(1, 3):
list(generator_word())
['A', 'B', 'C', 'D', 'E', 1, 2]
```

- yield from 结果会在内部自动捕获StopIteration 异常。这种处理方式与 for 循环处理StopIteration异常的方式一样。
- 对于yield from 结构来说，解释器不仅会捕获StopIteration异常，还会把value属性的值变成yield from 表达式的值。
- 在函数外部不能使用yield from（yield也不行）。
- yield from 是 Python3.3 后新加的语言结构。和其他语言的await关键字类似，
- yield from x 表达式对x对象做的第一件事是，调用 iter(x)，获取迭代器。所以要求x是可迭代对象
- yield from 的主要功能是打开双向通道，把最外层的调用方与最内层的子生成器连接起来，使两者可以直接发送和产出值，还可以直接传入异常，而不用在中间的协程添加异常处理的代码。

4. 使用yield from



上图：委派生成器在yeild from 表达式处暂停，调用方可以直接把数据发给子生成器，子生成器再把产出的值发给调用方。子生成器返回后，解释器会抛出 StopIteration异常，并把返回值附加在异常对象上，此时委派生成器恢复。

yield from 包含几个概念：

委派生成器： 包含yield from 表达式的生成器函数

子生成器： 从yield from 部分获取的生成器。

调用方： 调用委派生成器的客户端（调用方）代码 委派生成器在 yield from 表达式处暂停时，调用方可以直接把数据发给字生成器，子生成器再把产出的值发送给调用方。子生成器返回之后，解释器会抛出StopIteration异常，并把返回值附加到异常对象上，只是委派生成器恢复

Python

```python

#! -*- coding: utf-8 -*-

from collections import namedtuple


Result = namedtuple('Result', 'count average')


# 子生成器
# 这个例子和上边示例中的 averager 协程一样，只不过这里是作为字生成器使用
def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        # main 函数发送数据到这里
        term = yield
        if term is None: # 终止条件
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average) # 返回的Result 会成为grouper函数中yield fro


# 委派生成器
def grouper(results, key):
     # 这个循环每次都会新建一个averager 实例，每个实例都是作为协程使用的生成器对象
    while True:
        # grouper 发送的每个值都会经由yield from 处理，通过管道传给averager 实例。
        results[key] = yield from averager()


# 调用方
def main(data):
    results = {}
```

```
    for key, values in data.items():
        # group 是调用grouper函数得到的生成器对象，传给grouper 函数的第一个参数是r
        group = grouper(results, key)
        next(group)
        for value in values:
            # 把各个value传给grouper 传入的值最终到达averager函数中；
            # grouper并不知道传入的是什么，同时grouper实例在yield from处暂停
            group.send(value)
        # 把None传入groupper，传入的值最终到达averager函数中，导致当前实例终止。然
        # 如果没有group.send(None)，那么averager子生成器永远不会终止，委派生成器也
        group.send(None)
    report(results)


# 输出报告
def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print('{:2} {:5} averaging {:.2f}{}'.format(result.count, group, r


data = {
    'girls;kg':[40, 41, 42, 43, 44, 54],
    'girls;m': [1.5, 1.6, 1.8, 1.5, 1.45, 1.6],
    'boys;kg':[50, 51, 62, 53, 54, 54],
    'boys;m': [1.6, 1.8, 1.8, 1.7, 1.55, 1.6],
}

if __name__ == '__main__':
main(data)
```

PEP380 分6点说明了yield from 的行为。

- 子生成器产出的值都直接传给委派生成器的调用方（客户端代码）
- 使用send() 方法发给委派生成器的值都直接传给子生成器。如果发送的值是None，那么会调用子生成器的 **next**()方法。如果发送的值不是None，那么会调用子生成器的send()方法。如果调用的方法抛出StopIteration异常，那么委派生成器恢复运行。任何其他异常都会向上冒泡，传给委派生成器。
- 生成器退出时，生成器（或子生成器）中的return expr 表达式会触发 StopIteration(expr) 异常抛出。
- yield from表达式的值是子生成器终止时传给StopIteration异常的第一个参数。
- 传入委派生成器的异常，除了 GeneratorExit 之外都传给子生成器的throw()方法。如果调用throw()方法时抛出 StopIteration 异常，委派生成器恢复运行。StopIteration之外的异常会向上冒泡。传给委派生成器。
- 如果把 GeneratorExit 异常传入委派生成器，或者在委派生成器上调用close() 方法，那么在子生成器上调用close() 方法，如果他有的话。如果调用close() 方法导致异常抛出，那么异常会向上冒

泡，传给委派生成器；否则，委派生成器抛出 GeneratorExit 异常。

用于自动预激的装饰器与yield from 语句不兼容。如async。 @type

5 协程的优缺点

优点

> - 协程轻量级的，不需要系统调度，故可以提高性能；
> - 没有原子操作，同步&&锁
> - 高并发：一个CPU支持上万的协程都不是问题

缺点

> - 不能利用多核资源：协程的本质是个单线程,它不能同时将 单个CPU 的多个核用上,协程需要和进程配合才能运行在多CPU上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是cpu密集型应用
> - 阻塞操作会阻塞调整个程序。

6. 实现协程的方法：

6.1 greenlet

可以实现协程，不过每一次都要人为的去指向下一个该执行的协程

6.2 gevent 第三方包

比greenlet更强大的并且能够自动切换任务的模块gevent, gevent每次遇到io操作，需要耗时等待时，会自动跳到下一个协程继续执行

使用猴子补丁，gevent能够修改标准库里面大部分的阻塞式系统调用，包括socket、ssl、threading和select等模块，而变为协作式运行。也就是通过猴子补丁的monkey.patch_xxx()来将python标准库中模块或函数改成gevent中的响应的具有协程的协作式对象。这样在不改变原有代码的情况下，将应用的阻塞式方法，变成协程式的

```python
# python2
from gevent import monkey; monkey.patch_all()
import gevent
import urllib2

def f(url):
    print('GET: %s' % url)
    resp = urllib2.urlopen(url)
    data = resp.read()
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(f, 'https://www.python.org/'),
        gevent.spawn(f, 'https://www.yahoo.com/'),
        gevent.spawn(f, 'https://github.com/'),
])
```

6.3 asyncio

asyncio包详解见下一章

asyncio的编程模型就是一个消息循环。我们从asyncio模块中直接获取一个EventLoop的引用，然后把需要执行的协程扔到EventLoop中执行，就实现了异步IO

@asyncio.coroutine

```python
import asyncio

@asyncio.coroutine
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

6.4 asyncio.async(…)* await 函数

```python
async def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = await asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

@asyncio.coroutine -> async yield -> from yield

总结：

协程由编程语言提供，由程序员控制进行切换，所以没有线程安全问题，可以用来处理状态机，并发请求等。但是无法利用多核优势。不过，这个可以通过协程+进程的方式来解决

并发实现模块和相关包

一、concurrent.futures模块

concurrent.futures模块，可以利用multiprocessing实现真正的平行计算。

核心原理是：concurrent.futures会以子进程的形式，平行的运行多个python解释器，从而令python程序可以利用多核CPU来提升执行速度。由于子进程与主解释器相分离，所以他们的全局解释器锁也是相互独立的。每个子进程都能够完整的使用一个CPU内核。

这个小节主要讨论python3.2 引入的concurrent.futures模块。之前的版本使用这个模块需要手动安装 futures 这个模块。

这里会介绍future的概念。 future是一种对象，表示异步执行的操作。它是concurrent.futures模块和下一节asyncio包的基础。

Python标准库为我们提供了threading和multiprocessing模块编写相应的多线程/多进程代码。而concurrent.futures模块，它提供了ThreadPoolExecutor和ProcessPoolExecutor两个类， 实现了对threading和multiprocessing的更高级的抽象，对编写线程池/进程池提供了直接的支持。concurrent.futures基础模块是executor和submit。

先来看一段代码

```python
from concurrent import futures

def download_pic(url):
    image = get_image(url)  # 获取图片
    save(image)  # 存储图片
    return url


def down_pictures(pic_list):
    """
    pic_list: 表示图片地址列表
    """
    results = []
    with futures.ThreadPoolExecutor(max_workers=3) as executor:
        tasks = []
        for url in pic_list[:5]:
            future = executor.submit(download_pic, url)
            tasks.append(future)
            print(url, future)

        for future in futures.as_completed(tasks):
            res = future.result()
            print(future, res)
            results.append(res)

    return results
```

1.1 executor

Executor是一个抽象类，它不能被直接使用。它为具体的异步执行定义了一些基本的方法。ThreadPoolExecutor和ProcessPoolExecutor继承了Executor。 值得一提的是Executor实现了**enter**和**exit**使得其对象可以使用with操作符。使得当任务执行完成之后，自动执行shutdown函数，而无需编写相关释放代码。

- ThreadPoolExecutor对象

ThreadPoolExecutor类是Executor子类，使用线程池执行异步调用.

class concurrent.futures.ThreadPoolExecutor(max_workers)

使用max_workers数目的线程池执行异步调用

- ProcessPoolExecutor对象

ThreadPoolExecutor类是Executor子类，使用进程池执行异步调用.

class concurrent.futures.ProcessPoolExecutor(max_workers=None)

使用max*workers数目的进程池执行异步调用，如果max*workers为None则使用机器的处理器数目（如4核机器max_worker配置为None时，则使用4个进程进行异步并发）

- submit()

Executor中定义了 submit(fn, *args, **kwargs)方法， fn执行的函数fn(*args, **kwargs)。这个方法的作用是提交一个可执行回调的任务， 然后返回一个Futurn对象，也就是给定的回调。

Python

```
# -*- coding:utf-8 -*-

from concurrent import futures
import time


def show_time(num):
    return time.ctime(), num


with futures.ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(show_time, 1)
    print(future.result())
```

如果要提交多个任务， 可以通过循环，多次submit()

- map()

Executor.map(func, *iterables, timeout=None)相当于map(func, *iterables)，但是func是异步执行。timeout的值可以是int或float，如果操作超时，会返回raisesTimeoutError；如果不指定timeout参数，则不设置超时间。

> - func：需要异步执行的函数
> - *iterables：可迭代对象，如列表等。每一次func执行，都会从iterables中取参数。
> - timeout：设置每次异步操作的超时时间

```python
# -*- coding:utf-8 -*-

from concurrent import futures

import requests


def request_data(url, timeout=10):
    response = requests.get(url, timeout=timeout)
    return url, response.status_code


with futures.ThreadPoolExecutor(max_workers=3) as executor:
    url_list = ['http://www.baidu.com', 'http://www.jd.com', 'http://sina.
    results = executor.map(request_data, url_list)
    for future in results:
        print(future)
```

1.2 Future

Future可以理解为一个在未来完成的操作，这是异步编程的基础。通常情况下，我们执行io操作，访问url时在等待结果返回之前会产生阻塞， cpu不能做其他事情，而Future的引入帮助我们在等待的这段时间可以完成其他的操作。

Executor.submit函数返回future对象，future提供了跟踪任务执行状态的方法。比如判断任务是否执行中future.running()，判断任务是否执行完成future.done()等等。

- as_completed

as_completed方法传入futures迭代器和timeout两个参数。

默认timeout=None，阻塞等待任务执行完成，并返回执行完成的future对象迭代器，迭代器是通过yield实现的。

timeout>0，等待timeout时间，如果timeout时间到仍有任务未能完成，不再执行并抛出异常TimeoutError

as_completed不是按照元素的顺序返回的。

```python
from concurrent import futures


def download_pic(url):
    image = get_image(url)  # 获取图片
    save(image)  # 存储图片
    return url


def down_pictures(pic_list):
    """
    pic_list: 表示图片地址列表
    """
    results = []
    with futures.ThreadPoolExecutor(max_workers=3) as executor:
        tasks = []
        for url in pic_list[:5]:
            future = executor.submit(download_pic, url)
            tasks.append(future)
            print(url, future)

        for future in futures.as_completed(tasks):
            res = future.result()
            print(future, res)
            results.append(res)
    return results
```

- wait

wait方法接会返回一个tuple(元组)，tuple中包含两个set(集合)，一个是completed(已完成的)另外一个是uncompleted(未完成的)。使用wait方法的一个优势就是获得更大的自由度，它接收三个参数FIRST*COMPLETED, FIRST*EXCEPTION和ALL*COMPLETE，默认设置为ALL*COMPLETED。

> - FIRST_COMPLETED - Return when any future finishes or is cancelled.
> - FIRST*EXCEPTION - Return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL*COMPLETED.
> - ALL_COMPLETED - Return when all futures finish or are cancelled.

```python

import time
from concurrent.futures import ProcessPoolExecutor, wait, ALL_COMPLETED, F


def gcd(pair):
    """
    计算最大公约数
    :param pair:
    :return:
    """
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i

if __name__ == '__main__':
    numbers = [
        (1963309, 2265973), (1879675, 2493670), (2030677, 3814172),
        (1551645, 2229620), (1988912, 4736670), (2198964, 7876293)
    ]

    start = time.time()
    with ProcessPoolExecutor(max_workers=2) as pool:
        futures = [ pool.submit(gcd, pair) for pair in numbers]
        for future in futures:
            print('执行中:%s, 已完成:%s' % (future.running(), future.done())
        print('#### 分界线 ####')
        done, unfinished = wait(futures, timeout=2, return_when=ALL_COMPLE
        for d in done:
            print('执行中:%s, 已完成:%s' % (d.running(), d.done()))
            print(d.result())
    end = time.time()
    print('Took %.3f seconds.' % (end - start))
```

总结

executor.map VS futures.as_completed&&executor.submit

(1) executor.map()函数易于使用，他的一个特点很独特：函数的返回结果的顺序和调用开始的顺序一致。具体使用可以根据实际情况来选定。 如果第一个调用生成结果耗时10s, 其他的调用耗时2s, 代码会阻塞10s, 获取map方法返回的生成器产出的第一个结果，后续的结果不会阻塞，因为后序的调用痘印完成。

如果我们处理的时候必须等到获取到所有结果才处理或者要求严格的顺序，操作没有问题。 而我们通

常的情况是，不管提交的顺序，只要获取到最后的结果就行，故结合submit()以及features.as_completed函数结合起来使用。

(2) Executor.submit和futures.as_completed这个组合比executor.map更灵活，因为submit能处理不同的可调用对象和参数，而executor.map只能 处理参数不同的同一个可调用对象。

此外futures.as_completed函数的future集合可以来自多个executor实例。例如一些ThreadPoolExecutor实例，一些 ProcessPoolExecutor实例。

二、sched

并发定时任务

```python
"""
def enter(self, delay, priority, action, argument=(), kwargs=_sentinel):
    # A variant that specifies the time as a relative time.

    # This is actually the more commonly used interface.
    time = self.timefunc() + delay
    return self.enterabs(time, priority, action, argument, kwargs)
"""

import sched, time
s = sched.scheduler(time.time, time.sleep)
def print_time(a='default'):
    print("From print_time", time.ctime(), a)

def print_some_times():
    print(time.ctime())
    s.enter(10, 1, print_time)
    s.enter(5, 2, print_time, argument=('positional',))
    s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
    s.run()
    print(time.ctime())

print_some_times()
```

三、Queue/queue

Queue用于建立和操作队列，常和threading类一起用来建立一个简单的线程队列。

队列有很多种，根据进出顺序来分类，可以分成

- Queue.Queue(maxsize)　　FIFO（先进先出队列）
- Queue.LifoQueue(maxsize)　　FIFO（先进先出队列）
- Queue.PriorityQueue(maxsize)　　为优先度越低的越先出来

注意： 如果设置的maxsize小于1，则表示队列的长度无限长

FIFO是常用的队列，其一些常用的方法有：

- Queue.qsize()　　返回队列大小
- Queue.empty()　　判断队列是否为空
- Queue.full()　　判断队列是否满了
- Queue.get([block[,timeout]])　　从队列头删除并返回一个item，block默认为True，表示当队列为空却去get的时候会阻塞线程，等待直到有有item出现为止来get出这个item。如果是False的话表明当队列为空你却去get的时候，会引发异常。在block为True的情况下可以再设置timeout参数。表示当队列为空，get阻塞timeout指定的秒数之后还没有get到的话就引发Full异常。
- Queue.put(…[,block[,timeout]])　　向队尾插入一个item，同样若block=True的话队列满时就阻塞等待有空位出来再put，block=False时引发异常。同get的timeout，put的timeout是在block为True的时候进行超时设置的参数。
- Queue.task*done()*　　*从场景上来说，处理完一个get出来的item之后，调用task*done将向队列发出一个信号，表示本任务已经完成
- Queue.join() 监视所有item并阻塞主线程，直到所有item都调用了task_done之后主线程才继续向下执行

python2.x中，模块名为Queue python3.x中，模块名为queue

模块生产者消费者的场景是：生产者生产货物，然后把货物放到一个队列之类的数据结构中，生产货物所要花费的时间无法预先确定。消费者消耗生产者生产的货物的时间也是不确定的

```python

import queue,threading,time

q=queue.Queue()


def product(arg):
    while True:
        time.sleep(1)
        q.put(str(arg) + '包子')
        print('生产线程 %s 生产包子, 现有%s 个' % (str(arg), q.qsize()))


def consumer(arg):
    while True:
        q.get()
        print('消费线程 %s 消费包子, 现有%s 个' % (str(arg), q.qsize()))
        time.sleep(2)
for i in range(3):
    t=threading.Thread(target=product,args=(i,))
    t.start()
for j in range(2):
    t=threading.Thread(target=consumer,args=(j,))
    t.start()
```

四、asyncio包

asyncio是Python3.4版本引入的标准库，至少是Python3.3(需要手动安装）才可以使用。因为自 3.3 版本以来，

yield from 的引入使 Python 具备运行 asyncio 的基本条件。并发编程同时执行多条独立的逻辑流，即使它们是都工作在同个线程中的。每个协程都有独立的栈空间。

3.5版本中， 新语法的引入使Python具备了原生协程，而不再是一种新的生成器类型。

asyncio包使用事件循环驱动的协程实现并发。

4.1 Future

不用回调方法编写异步代码后，为了获取异步调用的结果，引入一个 Future 未来对象。Future 封装了与 loop 的交互行为，add*done*callback 方法向 epoll 注册回调函数，当 result 属性得到返回值后，会运行之前注册的回调函数，向上传递给 coroutine。

但是，每一个角色各有自己的职责，用 Future 向生成器 send result 以恢复工作状态并不合适，Future 对象本身的生存周期比较短，每一次注册回调、产生事件、触发回调过程后工作已经完成。所以这里

又需要在生成器协程与 Future 对象中引入一个新的对象 Task，对生成器协程进行状态管理。

run*until*complete() 执行事件循环，直到Future完成。返回Future结果或者对应的异常。

Python

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

Future with run_forever() 执行事件循环，直到调用stop()方法。

Python

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

def got_result(future):
    print(future.result())
    loop.stop()

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
future.add_done_callback(got_result)
try:
    loop.run_forever()
finally:
    loop.close()
```

PS:Python 里另一个 Future 对象是 concurrent.futures.Future，与 asyncio.Future 互不兼容，但容易产生混淆。

concurrent.futures 是线程级的 Future 对象，当使用 concurrent.futures.Executor 进行多线程编程时用于在不同的 thread 之间传递结果。

4.2 Task

Task，顾名思义，是维护生成器协程状态处理执行逻辑的的任务，Task 内的_step 方法负责生成器协程与 EventLoop 交互过程的状态迁移:向协程 send 一个值，恢复其工作状态，协程运行到断点后，得到新的未来对象，再处理 future 与 loop 的回调注册过程。

Task function

> Note In the functions below, the optional loop argument allows explicitly setting the event loop object used by the underlying task or coroutine. If it's not provided, the default event loop is used

wait()

wait(futures, *, loop=None, timeout=None, return*when=ALL*COMPLETED) 可以获取一个future的列表, 同时返回一个将它们全包括在内的单独的协程(done, pending)。类似concurrent.futures.wait()

所以我们可以这样写：

```
loop.run_until_complete(asyncio.wait([print_page('http://example.com/foo')
                                      print_page('http://example.com/bar')
```

as_completed()

asyncio.as_completed(fs, *, loop=None, timeout=None)， 通过它可以获取一个协程的列表，同时返回一个按完成顺序生成协程的迭代器，因此当你用它迭代时，会尽快得到每个可用的结果。

```
return an iterator whose values are coroutines.

result = loop.run_until_complete(asyncio.as_completed(*tasks)

for f in as_completed(fs):
    result = yield from f  # The 'yield from' may raise
    # Use result
```

gather()

gather 的返回值为协程运行的结果。

```
Return a future aggregating results from the given coroutines
    or futures.

result = loop.run_until_complete(asyncio.gather(*tasks)
for resultin results:
   print('Task ret: ', result)
```

ensure_future()

asyncio.ensure*future(coro*or_future, *, loop=None)

```
Schedule the execution of a coroutine object: wrap it in a future. Return

If the argument is a Future, it is returned directly.

New in version 3.4.4. Changed in version 3.5.1: The function accepts any a
```

4.3 Loop

事件循环的工作方式与用户设想存在一些偏差，理所当然的认知应是每个线程都可以有一个独立的 loop。但是在运行中，在主线程中才能通过 asyncio.get*event*loop() 创建一个新的 loop，而在其他线程时，使用 get*event*loop() 却会抛错，正确的做法应该是 asyncio.set*event*loop() 进行当前线程与 loop 的显式绑定。由于 loop 的运作行为并不受 Python 代码的控制，所以无法稳定的将协程拓展到多线程中运行。

协程在工作时，并不了解是哪个 loop 在对其调度，即使调用 asyncio.get*event*loop() 也不一定能获取到真正运行的那个 loop。因此在各种库代码中，实例化对象时都必须显式的传递当前的 loop 以进行绑定。

4.4 async与await

Python3.5 中引入了这两个关键字用以取代 asyncio.coroutine 与 yield from，从语义上定义了原生协程关键字，避免了使用者对生成器协程与生成器的混淆。这个阶段(3.0-3.4)使用 Python 的人不多，因此历史包袱不重，可以进行一些较大的革新。

await 的行为类似 yield from，但是它们异步等待的对象并不一致，yield from 等待的是一个生成器对象，而await接收的是定义了_*await*_方法的 awaitable 对象。

在 Python 中，协程也是 awaitable 对象。

4.5 asynico 协程

- async def函数必定是协程，即使里面不含有await语句。

- 如果在async函数里面使用yield或yield from语句，会引发SyntaxError异常。
- 在CPython内部，引入两个新的代码对象标识（code object flags）：

> (1) CO_COROUTINE表示这是原生协程。（由新语法定义）
>
> (2) CO*ITERABLE*COROUTINE表示这是用生成器实现的协程，但是和原生协程兼容。（用装饰器types.coroutine()装饰过的生成器协程）

- 调用一个普通生成器，返回一个生成器对象（generator object）；相应的，调用一个协程返回一个协程对象（coroutine object）。
- 协程不再抛出StopIteration异常，因为抛出的StopIteration异常会被包装（wrap）成一个RuntimeError异常。（在Python 3.5，对于普通生成器要想这样需要进行future import，见PEP 479）。
- 如果一个协程从未await等待就被垃圾收集器销毁了，会引发一个RuntimeWarning异常（见“调试特性”）

asyncio的编程模型就是一个消息循环。我们从asyncio模块中直接获取一个EventLoop的引用，然后把需要执行的协程扔到EventLoop中执行，就实现了异步IO 我们需要一个事件循环。

我们可以通过asyncio.get*event*loop()得到一个标准的事件循环，之后使用它的run*until*complete()方法来运行协同程序。 所以，为了使之前的协同程序运行，我们只需要做下面的步骤：

```
loop = asyncio.get_event_loop()
loop.run_until_complete(print_page('http://example.com'))
```

> 除非想阻塞主线程，从而冻结事件循环或者整个应用，否则不要在asyncio协程中使用time.sleep(delay)。如果协程需要在一段时间内什么也不做， 应该使用yield from asyncio sleep(delay)或者await asyncio.sleep(delay)。

装饰器+yield from

@types.coroutine()

types模块添加了一个新函数coroutine(fn)，使用它，“生成器实现的协程”和“原生协程”之间可以进行互操作。

Python

```
@types.coroutine
def process_data(db):
    data = yield from read_data(db)
    ...
```

coroutine(fn)函数给生成器的代码对象(code object)设置CO*ITERABLE*COROUTINE标识，使它返回一个协程对象。

@types.coroutine装饰器仅给生成器函数设置一个CO*ITERABLE*COROUTINE标识，除此之外什么也不做。但是如果生成器函数没有这个标识，await语句不会接受它的对象作为参数

@asyncio.coroutine把一个generator标记为coroutine类型

async + await

新的await表达式用于获得协程执行的结果：

```python
async def read_data(db):
    data = await db.fetch('SELECT ...')
    ...
```

以CPython内部，await使用了yield from的实现，但加入了一个额外步骤——验证它的参数类型。await只接受awaitable对象。

如果在async def函数之外使用await语句，会引发SyntaxError异常。这和在def函数之外使用yield语句一样。

如果await右边不是一个awaitable对象，会引发TypeError异常。

```
import asyncio
import datetime

async def display_date(loop):
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
# Blocking call which returns when the display_date() coroutine is done
loop.run_until_complete(display_date(loop))
loop.close()
```

协程嵌套

```python
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

异步上下文管理器和“async with”

异步上下文管理器（asynchronous context manager），可以在它的enter和exit方法里挂起、调用异步代码。

为此，我们设计了一套方案，添加了两个新的魔术方法：__aenter__和__aexit__，它们必须返回一个awaitable。

```python
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

数据库异步事务

```python
async def commit(session, data):
    ...

    async with session.transaction():
        ...
        await session.update(data)
        ...
```

async for

调试

新手在使用协程时可能忘记使用yield from语句，比如：

```python
@asyncio.coroutine
def useful():
    asyncio.sleep(1) # 前面忘写yield from，所以程序在这里不会挂起1秒
```

在asyncio里，对于此类错误，有一个特定的调试方法。装饰器 @coroutine用一个特定的对象包装（wrap）所有函数，这个对象有一个析构函数（destructor）用于记录警告信息。无论何时，一旦被装饰过的生成器被垃圾回收，会生成一个详细的记录信息（具体哪个函数、回收时的stack trace等等）。包装对象提供一个**repr**方法用来输出关于生成器的详细信息。

唯一的问题是如何启用这些调试工具，由于这些调试工具在生产模式里什么也不做，比如 @coroutine必须是在系统变量PYTHONASYNCIODEBUG出现时才具有调试功能。这时可以给asyncio程序进行如下设置：EventLoop.set_debug(true)，这时使用另一套调试工具，对 @coroutine的行为没有影响。

五、其他

5.1 subprocess 子进程

在Python中，我们通过标准库中的subprocess包来fork一个子进程，并运行一个外部的程序。

subprocess包中定义有数个创建子进程的函数，这些函数分别以不同的方式创建子进程，所以我们可以根据需要来从中选取一个使用。另外subprocess还提供了一些管理标准流(standard stream)和管道(pipe)的工具，从而在进程间使用文本通信。

并发并行

这一部分主要以实验对比形式展现。

一、IO任务任务实验：

方面方式：base, 多线程，多进程，进程+queue, 协程 百度图片下载

```python
def getManyPages(keyword,pages):
    # 获取下载图片的url路径
    params=[]
    for i in range(30, 30*pages+30, 30):
        params.append({
                       'tn': 'resultjson_com', 'ipn': 'rj',
                       'ct': 201326592, 'is': '', 'fp': 'result',
                       'queryWord': keyword,
                       'cl': 2, 'lm': -1, 'ie': 'utf-8', 'oe': 'utf-8',
                       'adpicid': '', 'st': -1, 'z': '', 'ic': 0,
                       'word': keyword, 's': '', 'se': '',
                       'tab': '', 'width': '', 'height': '', 'face': 0,
                       'istype': 2, 'qc': '',
                       'nc': 1, 'fr': '',
                       'pn': i,
                       'rn': 30, 'gsm': '1e', '1488942260214': ''
                   })
    url = 'https://image.baidu.com/search/acjson'
    urls = []
    for i in params:
        try:
            urls.extend(requests.get(url,params=i).json().get('data'))
        except:
            break

    return urls
```

1.1 base: 单线程

```python

def getImg(dataList, localPath):
    if not os.path.exists(localPath):  # 新建文件夹
        os.mkdir(localPath)

    x = 0
    for i in dataList:
        x += 1
        if i.get('thumbURL') != None:
            print('正在下载 {0}: {1}'.format(x, i.get('thumbURL')))
            ir = requests.get(i.get('thumbURL'))
            open(localPath + '%d.jpg' % (x), 'wb').write(ir.content)
        else:
            print('图片 {} 链接不存在'.format(x))


if __name__ == '__main__':
    dataList = getManyPages(u'火影', 30)  # 参数1:关键字, 参数2:要下载的页数
    getImg(dataList,'data/')  # 参数2:指定保存的路径
```

930次图片下载请求，耗时 133.3475s

1.2 多线程

```python
def getImg(url, localPath, timestamp):

    if url is not None:
        print('正在下载 : {}'.format(url))
        ir = requests.get(url)
        open(localPath + '%d.jpg' % (timestamp*1000000), 'wb').write(ir.co
        return Result(1)
    else:
        return Result(0)

def save_pic(dataList, localPath):
    if not os.path.exists(localPath):  # 新建文件夹
        os.mkdir(localPath)

    futures_set = set()
    with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:
        for item in dataList:
            url = item.get('thumbURL', None)
            if url is not None:
                timestamp = time.time()
                futures_set.add(executor.submit(getImg, url, localPath, ti

        summary = wait_for(futures_set)
        if summary.canceled:
            executor.shutdown()
        summarize(summary, 4)
        return summary

if __name__ == '__main__':
    dataList = getManyPages('火影', 30)  # 参数1:关键字，参数2:要下载的页数
    summary = save_pic(dataList,'data/')  # 参数2:指定保存的路径
```

930次图片下载请求， 耗时83.56204199790955

1.3 多进程

```python
def getImg(url, localPath, timestamp):

    if url is not None:
        print('正在下载 : {}'.format(url))
        ir = requests.get(url)
        open(localPath + '%d.jpg' % (timestamp*1000000), 'wb').write(ir.co
        return Result(1)
    else:
        return Result(0)

def save_pic(dataList, localPath):
    if not os.path.exists(localPath):  # 新建文件夹
        os.mkdir(localPath)

    futures_set = set()
    with concurrent.futures.ProcessPoolExecutor(max_workers=4) as executor
        for item in dataList:
            url = item.get('thumbURL', None)
            if url is not None:
                timestamp = time.time()
                futures_set.add(executor.submit(getImg, url, localPath, ti

        summary = wait_for(futures_set)
        if summary.canceled:
            executor.shutdown()
        summarize(summary, 4)
        return summary

if __name__ == '__main__':
    dataList = getManyPages('火影', 30)  # 参数1:关键字，参数2:要下载的页数
    summary = save_pic(dataList,'data/')  # 参数2:指定保存的路径
```

930次图片下载请求， 耗时82.85834980010986

1.4 process+queue

```python
def add_jobs(dataList, jobs, localPath):
    for index, item in enumerate(dataList, start=1):
        jobs.put((item.get('thumbURL', None), localPath))

    return index


def save_pic(datalist, localPath, concurrency):
    if not os.path.exists(localPath):  # 新建文件夹
        os.mkdir(localPath)
```

```
    canceled = False
    jobs = multiprocessing.JoinableQueue()  # task queue
    # jobs = multiprocessing.Queue()  # task queue
    results = multiprocessing.Queue()  # 结果队列
    create_process(jobs, results, concurrency)
    todo = add_jobs(datalist, jobs, localPath)  # 将待办任务加入任务队列

    try:
        jobs.join()
    except KeyboardInterrupt:
        canceled = True
    success = 0
    while not results.empty():
        result = results.get_nowait()
        success += result.success

    return Summary(todo, success, canceled)


def create_process(jobs, results, concurrency):
    for _ in range(concurrency):
        process = multiprocessing.Process(target=worker, args=(jobs, resul
        process.daemon = True
        process.start()


def worker(jobs, results):
    while True:
        try:
            url, localpath = jobs.get()
            try:
                result = getImg(url, localPath=localpath)
                results.put(result)
            except Exception as e:
                print(e)
        finally:
            jobs.task_done()


if __name__ == '__main__':
    dataList = getManyPages('火影', 30)  # 参数1:关键字, 参数2:要下载的页数
    summary = save_pic(dataList,'data/')  # 参数2:指定保存的路径
```

930次图片下载请求， 耗时 84.98532915115356

1.5 协程

```python
async def getImg(i, localPath):

    if not os.path.exists(localPath):  # 新建文件夹
        os.mkdir(localPath)

    down_pic(i, localPath)

def down_pic(i, localPath):
    timestamp = time.time() * 1000000
    if i.get('thumbURL') != None:
        print('正在下载 {0}'.format(i.get('thumbURL')))
        ir = requests.get(i.get('thumbURL'))
        open(localPath + '%d.jpg' % (timestamp), 'wb').write(ir.content)
    else:
        print('图片 {} 链接不存在'.format(i.get('thumbURL')))


if __name__ == '__main__':
    dataList = getManyPages(u'火影', 30)  # 参数1:关键字, 参数2:要下载的页数

    loop = asyncio.get_event_loop()
    tasks = [getImg(i, 'data/') for i in dataList]
    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()
```

930次图片下载请求， 耗时65.67994403839111

上述方式比较

项目	描述	耗时	加速
base	单线程, 930下载	133.35s	1
thread	4线程 930下载	83.56s	1.60
multiprocess	4进程,930下载	82.86s	1.61
process + queue	4进程+队列,930下载	84.98s	1.57
coroutine	协程, 930下载	65.78s	2.03

总结：对于IO密集型任务，协程表现好

二、CPU任务：

方面方式：base, 多线程， 多进程， 进程+queue, 协程

数值计算：x (a, b为正整数， a < x < b),

计算式子1 / x + math.sqrt(x) + math.sin(x) + math.cos(x)最小值

$$f(x) = \sum_{n=1}^{x} \left(\cos n + \sin n + \frac{1}{n} + \sqrt{n} \right)$$

公式实现

```python
def getSum(num):
    print('计算{}'.format(num))
    total = 0.0
    for x in range(1, num):
        total += 1 / x + math.sqrt(x) + math.sin(x) + math.cos(x)
    return Result(num, total, 1)
```

1. base

```python
def getSum(num):
    for i in range(1, num):
        total = 0.0
        for x in range(1, i):
            total += 1/x + math.sqrt(x) + math.sin(x) + math.cos(x)
        print(total)

    return total

print(getSum(20000))
```

20000次计算，平均耗时131.50s

2. 多线程

```python

def save_pic(num):

    futures_set = set()
    with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:
        for i in range(1, num):
            if i is not None:
                futures_set.add(executor.submit(getSum, i))

        summary = wait_for(futures_set)
        if summary.canceled:
            executor.shutdown()
        # summarize(summary, 4)
        return summary

if __name__ == '__main__':
    summary = save_pic(20000)  #
```

20000次计算，平均耗时124.02s

3. multiprocess

```python
def save_pic(num, concurrency):

    futures_set = set()
    with concurrent.futures.ProcessPoolExecutor(max_workers=concurrency) a
        # for url in get_job(num):
        for i in range(1, num):
            if i is not None:
                futures_set.add(executor.submit(getSum, i))

        summary = wait_for(futures_set)
        if summary.canceled:
            executor.shutdown()
        # summarize(summary, 4)
        return summary
```

20000次计算，平均耗时71.51s

4. queue + process

```python
def add_jobs(num, jobs):
    for index in range(1, num):
        jobs.put(index)

    return index

def save_pic(num, concurrency):
    # num 计算次数, concurrency 进程个数
    canceled = False
    jobs = multiprocessing.JoinableQueue()  # task queue
    # jobs = multiprocessing.Queue()  # task queue
    results = multiprocessing.Queue()  # 结果队列
    create_process(jobs, results, concurrency)
    todo = add_jobs(num, jobs)  # 将待办任务加入任务队列

    try:
        jobs.join()
    except KeyboardInterrupt:
        canceled = True
    success = 0
    while not results.empty():
        result = results.get_nowait()
        success += result.success

    return Summary(todo, success, canceled)

def create_process(jobs, results, concurrency):
    for _ in range(concurrency):
        process = multiprocessing.Process(target=worker, args=(jobs, resul
        process.daemon = True
        process.start()


def worker(jobs, results):
    while True:
        try:
            sum_result = jobs.get()
            try:
                result = getSum(sum_result)
                results.put(result)
            except Exception as e:
                print(e)
        finally:
            jobs.task_done()
```

20000次计算，平均耗时77.31s

5. 协程

```python
async def getSum(num):
    print('计算{}'.format(num))
    total = 0.0
    for x in range(1, num):
        total += 1 / x + math.sqrt(x) + math.sin(x) + math.cos(x)
    return total


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    tasks = [getSum(i) for i in range(1, 20000)]
    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()
```

20000次计算，平均耗时124.85s

上述方式比较:

项目	描述	耗时	加速
base	单线程, 20000次计算	131.50s	1
thread	4线程 20000次计算	124.02s	1.06
multiprocess	4进程,20000次计算	71.51s	1.84
process + queue	4进程+队列,20000次计算	77.31s	1.70
coroutine	协程, 20000次计算	124.85s	1.05

总结：面对CPU任务，多进程表现好