随笔-111 文章-0 评论-5

深入理解 Python 异步编程(上)

http://python.jobbole.com/88291/

前言

很多朋友对异步编程都处于"听说很强大"的认知状态。鲜有在生产项目中使用它。而使用它的同学，则大多数都停留在知道如何使用 Tornado、Twisted、Gevent 这类异步框架上，出现各种古怪的问题难以解决。而且使用了异步框架的部分同学，由于用法不对，感觉它并没牛逼到哪里去，所以很多同学做 Web 后端服务时还是采用 Flask、Django等传统的非异步框架。

从上两届 PyCon 技术大会看来，异步编程已经成了 Python 生态下一阶段的主旋律。如新兴的 Go、Rust、Elixir 等编程语言都将其支持异步和高并发作为主要"卖点"，技术变化趋势如此。Python 生态为不落人后，从2013年起由 Python 之父 Guido 亲自操刀主持了 Tulip(asyncio)项目的开发。

本系列教程分为上中下篇，让读者深入理解Python异步编程，解决在使用异步编程中的疑惑，深入学习Python3中新增的**asyncio**库和**async/await**语法，尽情享受 Python 带来的简洁优雅和高效率。

内容安排

上篇

- 了解 异步编程及其紧密相关的概念，如阻塞/非阻塞、同步/异步、并发/并行等
- 理解 异步编程是什么，以及异步编程的困难之处
- 理解 为什么需要异步编程
- 熟悉 如何从同步阻塞发展到异步非阻塞的
- 掌握**epoll + Callback + Event loop**是如何工作的
- 掌握 Python 是如何逐步从回调到生成器再到原生协程以支持异步编程的
- 掌握 **asyncio** 的工作原理

中篇

- 掌握 asyncio 标准库基本使用
- 掌握 asyncio 的事件循环
- 掌握 协程与任务如何使用与管理（如调度与取消调度）
- 掌握 同步原语的使用(Lock、Event、Condition、Queue)
- 掌握 asyncio 和多进程、多线程结合使用

下篇

- 理解 GIL 对异步编程的影响
- 理解 asyncio 踩坑经验
- 理解 回调、协程、绿程(Green-Thread)、线程对比总结
- 掌握 多进程、多线程、协程各自的适用场景
- 了解 Gevent/libev、uvloop/libuv 与asyncio的区别和联系
- 掌握 Python异步编程的一些指导细则

1 什么是异步编程

通过学习相关概念，我们逐步解释异步编程是什么。

1.1 阻塞

- 程序未得到所需计算资源时被挂起的状态。
- **程序在等待某个操作完成期间，自身无法继续干别的事情，则称该程序在该操作上是阻塞的。**
- 常见的阻塞形式有：网络I/O阻塞、磁盘I/O阻塞、用户输入阻塞等。

阻塞是无处不在的，包括CPU切换上下文时，所有的进程都无法真正干事情，它们也会被阻塞。（如果是多核CPU则正在执行上下文切换操作的核不可被利用。）

1.2 非阻塞

- **程序在等待某操作过程中，自身不被阻塞，可以继续运行干别的事情，则称该程序在该操作上是非阻塞的。**
- 非阻塞并**不是**在任何程序级别、任何情况下都可以存在的。
- 仅当程序封装的级别可以囊括独立的子程序单元时，它才可能存在非阻塞状态。

非阻塞的存在是因为阻塞存在，正因为某个操作阻塞导致的耗时与效率低下，我们才要把它变成非阻塞的。

1.3 同步

- 不同程序单元为了完成某个任务，在执行过程中需靠某种通信方式以**协调一致**，称这些程序单元是同步执行的。

- 例如购物系统中更新商品库存，需要用“行锁”作为通信信号，让不同的更新请求强制排队顺序执行，那更新库存的操作是同步的。
- 简言之，**同步意味着有序。**

1.4 异步

- 为完成某个任务，不同程序单元之间**过程中无需通信协调**，也能完成任务的方式。
- 不相关的程序单元之间可以是异步的。
- 例如，爬虫下载网页。调度程序调用下载程序后，即可调度其他任务，而无需与该下载任务保持通信以协调行为。不同网页的下载、保存等操作都是无关的，也无需相互通知协调。这些异步操作的完成时刻并不确定。
- 简言之，**异步意味着无序。**

上文提到的“通信方式”通常是指异步和并发编程提供的同步原语，如信号量、锁、同步队列等等。我们需知道，虽然这些通信方式是为了让多个程序在一定条件下同步执行，但正因为是异步的存在，才需要这些通信方式。如果所有程序都是按序执行，其本身就是同步的，又何需这些同步信号呢？

1.5 并发

- 并发描述的是程序的组织结构。指程序要被设计成多个可独立执行的子任务。
- **以利用有限的计算机资源使多个任务可以被实时或近实时执行为目的。**

1.6 并行

- 并行描述的是程序的执行状态。指多个任务同时被执行。
- **以利用富余计算资源（多核CPU）加速完成多个任务为目的。**

并发提供了一种程序组织结构方式，让问题的解决方案可以并行执行，但并行执行不是必须的。

1.7 概念总结

- **并行**是为了利用多核加速多任务完成的进度
- **并发**是为了让独立的子任务都有机会被尽快执行，但不一定能加速整体进度
- **非阻塞**是为了提高程序整体执行效率
- **异步**是高效地组织非阻塞任务的方式

要支持并发，必须拆分为多任务，不同任务相对而言才有阻塞/非阻塞、同步/异步。所以，并发、异步、非阻塞三个词总是如影随形。

1.8 异步编程

- **以进程、线程、协程、函数/方法作为执行任务程序的基本单位，结合回调、事件循环、信号量等机制，以提高程序整体执行效率和并发能力的编程方式。**

如果在某程序的运行时，能根据已经执行的指令准确判断它接下来要进行哪个具体操作，那它是同步程序，反之则为异步程序。（无序与有序的区别）

同步/异步、阻塞/非阻塞并非水火不容，要看讨论的程序所处的封装级别。例如购物程序在处理多个用户的浏览请求可以是异步的，而更新库存时必须是同步的。

1.9 异步之难(nán)

- 控制不住“计几”写的程序，因为其执行顺序不可预料，**当下正要发生什么事件不可预料。**在并行情况下更为复杂和艰难。

所以，几乎所有的异步框架都将异步编程模型**简化**：**一次只允许处理一个事件。**故而有关异步的讨论几乎都集中在了单线程内。

- 如果某事件处理程序需要长时间执行，所有其他部分都会被阻塞。

所以，**一旦采取异步编程，每个异步调用必须“足够小”**，不能耗时太久。如何拆分异步任务成了难题。

- 程序下一步行为往往依赖上一步执行结果，如何知晓上次异步调用已完成并获取结果？
- **回调（Callback）**成了必然选择。那又需要面临“回调地狱”的折磨。
- 同步代码改为异步代码，必然破坏代码结构。
- 解决问题的逻辑也要转变，不再是一条路走到黑，需要精心安排异步任务。

2 苦心异步为哪般

如上文所述，异步编程面临诸多难点，Python 之父亲自上阵打磨4年才使 asyncio 模块在Python 3.6中“转正”，如此苦心为什么？答案只有一个：它值得！下面我们看看为何而值得。

2.1 CPU的时间观

我们将一个 2.6GHz 的 CPU 拟人化，假设它执行一条命令的时间，他它感觉上过了一秒钟。CPU是计算机的处理核心，也是最宝贵的资源，如果有浪费CPU的运行时间，导致其利用率不足，那程序效率必然低下（因为实际上有资源可以使效率更高）。

如上图所示，在千兆网上传输2KB数据，CPU感觉过了14个小时，如果是在10M的公网上呢？那效率会低百倍！如果在这么长的一段时间内，CPU只是傻等结果而不能去干其他事情，是不是在浪费CPU的青春？

鲁迅说，浪费"CPU"的时间等于谋财害命。而凶手就是程序猿。

2.2 面临的问题

- **成本问题**

如果一个程序不能有效利用一台计算机资源，那必然需要更多的计算机通过运行更多的程序实例来弥补需求缺口。例如我前不久主导重写的项目，使用Python异步编程，改版后由原来的7台服务器削减至3台，成本骤降57%。一台AWS m4.xlarge 型通用服务器按需付费实例一年价格约 1.2 万人民币。

- **效率问题**

如果不在乎钱的消耗，那也会在意效率问题。当服务器数量堆叠到一定规模后，如果不改进软件架构和实现，加机器是徒劳，而且运维成本会骤然增加。比如别人家的电商平台支持6000单/秒支付，而自家在下单量才支撑2000单/秒，在双十一这种活动的时候，钱送上门也赚不到。

- **C10k/C10M挑战**

C10k（concurrently handling 10k connections）是一个在1999年被提出来的技术挑战，如何在一颗1GHz CPU，2G内存，1gbps网络环境下，让单台服务器同时为1万个客户端提供FTP服务。而到了2010年后，随着硬件技术的发展，这个问题被延伸为C10M，即如何利用8核心CPU，64G内存，在10gbps的网络上保持1000万并发连接，或是每秒钟处理100万的连接。（两种类型的计算机资源在各自的时代都约为1200美元）

成本和效率问题是从企业经营角度讲，C10k/C10M问题则是从技术角度出发挑战软硬件极限。C10k/C10M 问题得解，成本问题和效率问题迎刃而解。

2.3 解决方案

《约束理论与企业优化》中指出："**除了瓶颈之外，任何改进都是幻觉。**"

CPU告诉我们，它自己很快，而上下文切换慢、内存读数据慢、磁盘寻址与取数据慢、网络传输慢……总之，离开CPU 后的一切，除了一级高速缓存，都很慢。我们观察计算机的组成可以知道，主要由运算器、控制器、存储器、输入设备、输出设备五部分组成。运算器和控制器主要集成在CPU中，除此之外全是I/O，包括读写内存、读写磁盘、读写网卡全都是I/O。**I/O成了最大的瓶颈。**

异步程序可以提高效率，而最大的瓶颈在I/O，业界诞生的解决方案没出意料：**异步I/O吧，异步I/O吧，异步I/O吧吧！**

3 异步I/O进化之路

如今，地球上最发达、规模最庞大的计算机程序，莫过于因特网。而从CPU的时间观中可知，**网络I/O是最大的I/O瓶颈**，除了宕机没有比它更慢的。所以，诸多异步框架都对准的是网络I/O。

我们从一个爬虫例子说起，从因特网上下载10篇网页。

3.1 同步阻塞方式

最容易想到的解决方案就是依次下载，从建立socket连接到发送网络请求再到读取响应数据，顺序进行。

```python
import socket


def blocking_way():
    sock = socket.socket()
    # blocking
    sock.connect(('example.com', 80))
    request = 'GET / HTTP/1.0\r\nHost: example.com\r\n\r\n'
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        # blocking
        chunk = sock.recv(4096)
    return response


def sync_way():
    res = []
    for i in range(10):
        res.append(blocking_way())
    return len(res)
```

注：总体耗时约为4.5秒。（因网络波动每次测试结果有所变动，本文取多次平均值）

如上图所示，**blocking_way()** 的作用是建立 socket 连接，发送HTTP请求，然后从 socket读取HTTP响应并返回数据。示例中我们请求了 example.com 的首页。在**sync_way()** 执行了10次，即下载 example.com 首页10次。

在示例代码中有两个关键点。一是第10行的 **sock.connect((‘example.com’, 80))**，该调用的作用是向**example.com**主机的**80**端口发起网络连接请求。 二是第14行、第18行的**sock.recv(4096)**，该调用的作用是从socket上读取4K字节数据。

我们知道，创建网络连接，多久能创建完成不是客户端决定的，而是由网络状况和服务端处理能力共同决定。服务端什么时候返回了响应数据并被客户端接收到可供程序读取，也是不可预测的。**所以sock.connect()和sock.recv()这两个调用在默认情况下是阻塞的。**

> 注：**sock.send()**函数并不会阻塞太久，它只负责将请求数据拷贝到TCP/IP协议栈的系统缓冲区中就返回，并不等待服务端返回的应答确认。

假设网络环境很差，创建网络连接需要1秒钟，那么**sock.connect()**就得阻塞1秒钟，等待网络连接成功。这1秒钟对一颗2.6GHz的CPU来讲，仿佛过去了83年，然而它不能干任何事情。**sock.recv()**也是一样的必须得等到服务端的响应数据已经被客户端接收。我们下载10篇网页，这个阻塞过程就得重复10次。如果一个爬虫系统每天要下载1000万篇网页呢？！

上面说了很多，我们力图说明一件事：**同步阻塞的网络交互方式，效率低十分低下。**特别是在网络交互频繁的程序中。这种方式根本**不可能**挑战C10K/C10M。

3.2 改进方式：多进程

在一个程序内，依次执行10次太耗时，那开10个一样的程序同时执行不就行了。于是我们想到了多进程编程。**为什么会先想到多进程呢**？发展脉络如此。在更早的操作系统（Linux 2.4）及其以前，进程是 OS 调度任务的实体，是面向进程设计的OS。

```python
import socket
from concurrent import futures


def blocking_way():
    sock = socket.socket()
    # blocking
    sock.connect(('example.com', 80))
    request = 'GET / HTTP/1.0\r\nHost: example.com\r\n\r\n'
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        # blocking
        chunk = sock.recv(4096)
    return response


def process_way():
    workers = 10
    with futures.ProcessPoolExecutor(workers) as executor:
        futs = {executor.submit(blocking_way) for i in range(10)}
    return len([fut.result() for fut in futs])
```

注：总体耗时约为 0.6 秒。

改善效果立竿见影。但仍然有问题。总体耗时并没有缩减到原来的十分之一，而是九分之一左右，还有一些时间耗到哪里去了？**进程切换开销。**

进程切换开销不止像“CPU的时间观”所列的“上下文切换”那么低。CPU从一个进程切换到另一个进程，需要把旧进程运行时的寄存器状态、内存状态全部保存好，再将另一个进程之前保存的数据恢复。对CPU来讲，几个小时就干等着。**当进程数量大于CPU核心数量时，进程切换是必然需要的。**

除了切换开销，多进程还有另外的缺点。一般的服务器在能够稳定运行的前提下，可以同时处理的进程数在数十个到数百个规模。如果进程数量规模更大，系统运行将不稳定，而且可用内存资源往往也会不足。

多进程解决方案在面临每天需要成百上千万次下载任务的爬虫系统，或者需要同时搞定数万并发的电商系统来说，并不适合。

除了**切换开销大**，以及**可支持的任务规模小**之外，多进程还有其他缺点，如状态共享等问题，后文会有提及，此处不再细究。

3.3 继续改进：多线程

由于线程的数据结构比进程更轻量级，同一个进程可以容纳多个线程，从进程到线程的优化由此展开。后来的OS也把调度单位由进程转为线程，进程只作为线程的容器，用于管理进程所需的资源。而且OS级别的线程是可以被分配到不同的CPU核心同时运行的。

```python
import socket
from concurrent import futures


def blocking_way():
    sock = socket.socket()
    # blocking
    sock.connect(('example.com', 80))
    request = 'GET / HTTP/1.0\r\nHost: example.com\r\n\r\n'
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        # blocking
        chunk = sock.recv(4096)
    return response


def thread_way():
    workers = 10
    with futures.ThreadPoolExecutor(workers) as executor:
        futs = {executor.submit(blocking_way) for i in range(10)}
    return len([fut.result() for fut in futs])
```

注：总体运行时间约0.43秒。

结果符合预期，比多进程耗时要少些。从运行时间上看，多线程似乎已经解决了切换开销大的问题。而且可支持的任务数量规模，也变成了数百个到数千个。

但是，多线程仍有问题，特别是Python里的多线程。首先，Python中的多线程因为GIL的存在，它们并不能利用CPU多核优势，**一个Python进程中，只允许有一个线程处于运行状态。**那为什么结果还是如预期，耗时缩减到了十分之一？

因为在做阻塞的系统调用时，例如**sock.connect(),sock.recv()**时，当前线程会释放GIL，让别的线程有执行机会。但是单个线程内，在阻塞调用上还是阻塞的。

> 小提示：Python中 time.sleep 是阻塞的，都知道使用它要谨慎，但在多线程编程中，time.sleep 并不会阻塞其他线程。

除了GIL之外，所有的多线程还有通病。它们是被OS调度，调度策略是抢占式的，以保证同等优先级的线程都有均等的执行机会，那带来的问题是：并不知道下一时刻是哪个线程被运行，也不知道它正要执行的代码是什么。所以就可能存在**竞态条件。**

例如爬虫工作线程从任务队列拿待抓取URL的时候，如果多个爬虫线程同时来取，那这个任务到底该给谁？那就需要用到"锁"或"同步队列"来保证下载任务不会被重复执行。

而且线程支持的多任务规模，在数百到数千的数量规模。在大规模的高频网络交互系统中，仍然有些吃力。当然，**多线程最主要的问题还是竞态条件。**

3.4 非阻塞方式

终于，我们来到了非阻塞解决方案。先来看看最原始的非阻塞如何工作的。

```python
import socket


def nonblocking_way():
    sock = socket.socket()
    sock.setblocking(False)
    try:
        sock.connect(('example.com', 80))
    except BlockingIOError:
        # 非阻塞连接过程中也会抛出异常
        pass
    request = 'GET / HTTP/1.0\r\nHost: example.com\r\n\r\n'
    data = request.encode('ascii')
    # 不知道socket何时就绪，所以不断尝试发送
    while True:
        try:
            sock.send(data)
            # 直到send不抛异常，则发送完成
            break
        except OSError:
            pass

    response = b''
    while True:
        try:
            chunk = sock.recv(4096)
            while chunk:
                response += chunk
                chunk = sock.recv(4096)
            break
        except OSError:
            pass
    return response


def sync_way():
    res = []
    for i in range(10):
        res.append(nonblocking_way())
    return len(res)
```

注：总体耗时约4.3秒。

首先注意到两点，就感觉被骗了。一是耗时与同步阻塞相当，二是代码更复杂。要非阻塞何用？且慢。

上图第9行代码**sock.setblocking(False)**告诉OS，让socket上阻塞调用都改为非阻塞的方式。之前我们说到，非阻塞就是在做一件事的时候，不阻碍调用它的程序做别的事情。上述代码在执行完 **sock.connect()** 和 **sock.recv()** 后的确不再阻塞，可以继续往下执行请求准备的代码或者是执行下一次读取。

代码变得更复杂也是上述原因所致。第11行要放在**try**语句内，是因为**socket**在发送非阻塞连接请求过程中，系统底层也会抛出异常。

connect()被调用之后，立即可以往下执行第15和16行的代码。

需要while循环不断尝试 **send()**，是因为**connect()**已经非阻塞，在**send()**之时并不知道 **socket** 的连接是否就绪，只有不断尝试，尝试成功为止，即发送数据成功了。**recv()**调用也是同理。

虽然 connect() 和 recv() 不再阻塞主程序，空出来的时间段CPU没有空闲着，但并没有利用好这空闲去做其他有意义的事情，而是在循环尝试读写 socket （不停判断非阻塞调用的状态是否就绪）。还得处理来自底层的可忽略的异常。也不能同时处理多个 socket 。

然后10次下载任务仍然按序进行。所以总体执行时间和同步阻塞相当。如果非得这样子，那还不如同步阻塞算了。

3.5 非阻塞改进

3.5.1 epoll

判断非阻塞调用是否就绪如果 OS 能做，是不是应用程序就可以不用自己去等待和判断了，就可以利用这个空闲去做其他事情以提高效率。

所以**OS将I/O状态的变化都封装成了事件**，如可读事件、可写事件。并且**提供了专门的系统模块让应用程序可以接收事件通知。**这个模块就是**select**。让应用程序可以通过**select**注册文件描述符和回调函数。当文件描述符的状态发生变化时，**select** 就调用事先注册的回调函数。

select因其算法效率比较低，后来改进成了**poll**，再后来又有进一步改进，BSD内核改进成了**kqueue**模块，而Linux内核改进成了**epoll** 模块。这四个模块的作用都相同，暴露给程序员使用的API也几乎一致，区别在于**kqueue** 和 **epoll** 在处理大量文件描述符时效率更高。

鉴于 Linux 服务器的普遍性，以及为了追求更高效率，所以我们常常听闻被探讨的模块都是 **epoll** 。

3.5.2 回调(Callback)

把I/O事件的等待和监听任务交给了 OS，那 OS 在知道I/O状态发生改变后（例如socket连接已建立成功可发送数据），它又怎么知道接下来该干嘛呢？**只能回调。**

需要我们将发送数据与读取数据封装成独立的函数，让**epoll**代替应用程序监听**socket**状态时，得告诉**epoll**：“如果**socket**状态变为可以往里写数据（连接建立成功了），请调用HTTP请求发送函数。如果**socket** 变为可以读数据了（客户端已收到响应），请调用响应处理函数。”

于是我们利用**epoll**结合回调机制重构爬虫代码：

```python
import socket
from selectors import DefaultSelector, EVENT_WRITE, EVENT_READ

selector = DefaultSelector()
stopped = False
urls_todo = {'/', '/1', '/2', '/3', '/4', '/5', '/6', '/7', '/8', '/9'}


class Crawler:
    def __init__(self, url):
        self.url = url
        self.sock = None
        self.response = b''

    def fetch(self):
        self.sock = socket.socket()
        self.sock.setblocking(False)
        try:
            self.sock.connect(('example.com', 80))
        except BlockingIOError:
            pass
        selector.register(self.sock.fileno(), EVENT_WRITE, self.connected)

    def connected(self, key, mask):
        selector.unregister(key.fd)
        get = 'GET {0} HTTP/1.0\r\nHost: example.com\r\n\r\n'.format(self.url)
        self.sock.send(get.encode('ascii'))
        selector.register(key.fd, EVENT_READ, self.read_response)

    def read_response(self, key, mask):
        global stopped
        # 如果响应大于4KB，下一次循环会继续读
        chunk = self.sock.recv(4096)
        if chunk:
            self.response += chunk
        else:
            selector.unregister(key.fd)
            urls_todo.remove(self.url)
            if not urls_todo:
                stopped = True
```

此处和前面稍有不同的是，我们将下载不同的10个页面，相对URL路径存放于urls_todo集合中。现在看看改进在哪。

首先，不断尝试**send()** 和 **recv()** 的两个循环被消灭掉了。

其次，导入了**selectors**模块，并创建了一个**DefaultSelector** 实例。Python标准库提供的**selectors**模块是对底层**select/poll/epoll/kqueue**的封装。**DefaultSelector**类会根据 OS 环境自动选择最佳的模块，那在 Linux 2.5.44 及更新的版本上都是**epoll**了。

然后，在第25行和第31行分别注册了**socket**可写事件**(EVENT_WRITE)**和可读事件**(EVENT_READ)**发生后应该采取的回调函数。

虽然代码结构清晰了，阻塞操作也交给OS去等待和通知了，但是，我们要抓取10个不同页面，就得创建10个**Crawler**实例，就有20个事件

将要发生，那如何从**selector**里获取当前正发生的事件，并且得到对应的回调函数去执行呢？

3.5.3 事件循环（Event Loop）

为了解决上述问题，那我们只得采用老办法，写一个循环，去访问selector模块，等待它告诉我们当前是哪个事件发生了，应该对应哪个回调。**这个等待事件通知的循环，称之为事件循环。**

```
46 def loop():
47     while not stopped:
48         # 阻塞，直到一个事件发生
49         events = selector.select()
50         for event_key, event_mask in events:
51             callback = event_key.data
52             callback(event_key, event_mask)
```

上述代码中，我们用**stopped**全局变量控制事件循环何时停止。当**urls_todo**消耗完毕后，会标记**stopped**为**True**。

重要的是第49行代码，**selector.select() 是一个阻塞调用**，因为如果事件不发生，那应用程序就没事件可处理，所以就干脆阻塞在这里等待事件发生。那可以推断，如果只下载一篇网页，一定要**connect()**之后才能**send()**继而**recv()**，那它的效率和阻塞的方式是一样的。因为不在**connect()/recv()**上阻塞，也得在**select()**上阻塞。

所以，**selector机制(后文以此称呼代指epoll/kqueue)是设计用来解决大量并发连接的。**当系统中有大量非阻塞调用，能随时产生事件的时候，**selector**机制才能发挥最大的威力。

下面是如何启创建10个下载任务和启动事件循环的：

```
55 if __name__ == '__main__':
56     import time
57     start = time.time()
58     for url in urls_todo:
59         crawler = Crawler(url)
60         crawler.fetch()
61     loop()
62     print(time.time() - start)
```

注：总体耗时约0.45秒。

上述执行结果令人振奋。在单线程内用 **事件循环+回调** 搞定了10篇网页同时下载的问题。这，已经是**异步编程**了。虽然有一个for 循环顺序地创建Crawler 实例并调用 **fetch** 方法，但是**fetch** 内仅有**connect()**和注册可写事件，而且从执行时间明显可以推断，多个下载任务确实在同时进行！

上述代码异步执行的过程：

1. 创建**Crawler** 实例；
2. 调用**fetch**方法，会创建**socket**连接和在**selector**上注册可写事件；
3. fetch内并无阻塞操作，该方法立即返回；
4. 重复上述3个步骤，将10个不同的下载任务都加入事件循环；
5. 启动事件循环，进入第1轮循环，阻塞在事件监听上；
6. 当某个下载任务**EVENT_WRITE**被触发，回调其**connected**方法，第一轮事件循环结束；
7. 进入第2轮事件循环，当某个下载任务有事件触发，执行其回调函数；此时已经不能推测是哪个事件发生，因为有可能是上次**connected**里的**EVENT_READ**先被触发，也可能是其他某个任务的**EVENT_WRITE**被触发；（**此时，原来在一个下载任务上会阻塞的那段时间被利用起来执行另一个下载任务了**）
8. 循环往复，直至所有下载任务被处理完成
9. 退出事件循环，结束整个下载程序

3.5.4 总结

目前为止，我们已经从同步阻塞学习到了异步非阻塞。掌握了在单线程内同时并发执行多个网络I/O阻塞型任务的黑魔法。而且与多线程相比，连线程切换都没有了，执行回调函数是函数调用开销，在线程的栈内完成，因此性能也更好，单机支持的任务规模也变成了数万到数十万个。（不过我们知道：没有免费午餐，也没有银弹。）

部分编程语言中，对异步编程的支持就止步于此（不含语言官方之外的扩展）。需要程序猿直接使用epoll去注册事件和回调、维护一个事件循环，然后大多数时间都花在设计回调函数上。

通过本节的学习，我们应该认识到，不论什么编程语言，但凡要做异步编程，上述的"事件循环+回调"这种模式是逃不掉的，尽管它可能用的不是**epoll**，也可能不是**while**循环。如果你找到了一种不属于 "**等会儿告诉你**" 模型的异步方式，请立即给我打电话（注意，打电话是Call）。

为什么我们在某些异步编程中并没有看到 CallBack 模式呢？这就是我们接下来要探讨的问题。本节是学习异步编程的一个终点，也是另一个起点。毕竟咱们讲 Python 异步编程，还没提到其主角**协程**的用武之地。

4 Python 对异步I/O的优化之路

我们将在本节学习到 Python 生态对异步编程的支持是如何继承前文所述的“**事件循环+回调**”模式演变到asyncio的原生协程模式。

4.1 回调之痛，以终为始

在第3节中，我们已经学会了“**事件循环+回调**”的基本运行原理，可以基于这种方式在单线程内实现异步编程。也确实能够大大提高程序运行效率。但是，刚才所学的只是最基本的，然而在生产项目中，要应对的复杂度会大大增加。考虑如下问题：

- 如果回调函数执行不正常该如何？
- 如果回调里面还要嵌套回调怎么办？要嵌套很多层怎么办？
- 如果嵌套了多层，其中某个环节出错了会造成什么后果？
- 如果有个数据需要被每个回调都处理怎么办？
- ……

在实际编程中，上述系列问题不可避免。在这些问题的背后隐藏着回调编程模式的一些**缺点**：

- **回调层次过多时代码可读性差**

```
def callback_1():
  # processing ...
  def callback_2():
      # processing.....
      def callback_3():
          # processing ....
          def callback_4():
              #processing .....
              def callback_5():
                  # processing ......
              async_function(callback_5)
          async_function(callback_4)
      async_function(callback_3)
  async_function(callback_2)
async_function(callback_1)
```

- **破坏代码结构**

写同步代码时，关联的操作时自上而下运行：

Python

```
do_a()
do_b()
```

如果 b 处理依赖于 a 处理的结果，而 a 过程是异步调用，就不知 a 何时能返回值，需要将后续的处理过程以callback的方式传递给 a ，让 a 执行完以后可以执行 b。代码变化为：

Python

```
do_a(do_b())
```

如果整个流程中全部改为异步处理，而流程比较长的话，代码逻辑就会成为这样：

Python

```
1 do_a(do_b(do_c(do_d(do_e(do_f(......))))))
```

上面实际也是回调地狱式的风格，但这不是主要矛盾。主要在于，原本从上而下的代码结构，要改成从内到外的。先f，再e，再d，…，直到最外层 a 执行完成。在同步版本中，执行完a后执行b，这是线程的指令指针控制着的流程，而在回调版本中，流程就是程序猿需要注意和安排的。

- **共享状态管理困难**
  回顾第3节爬虫代码，同步阻塞版的**sock**对象从头使用到尾，而在回调的版本中，我们必须在**Crawler**实例化后的对象**self**里保存它自己的**sock**对象。如果不是采用OOP的编程风格，那需要把要共享的状态接力似的传递给每一个回调。多个异步调用之间，到底要共享哪些状态，事先就得考虑清楚，精心设计。
- **错误处理困难**
  一连串的回调构成一个完整的调用链。例如上述的 a 到 f。假如 d 抛了异常怎么办？整个调用链断掉，接力传递的状态也会丢失，这种现象称为**调用栈撕裂。** c 不知道该干嘛，继续异常，然后是 b 异常，接着 a 异常。好嘛，报错日志就告诉你，a 调用出错了，但实际是 d 出错。**所以，为了防止栈撕裂，异常必须以数据的形式返回，而不是直接抛出异常，然后每个回调中需要检查上次调用的返回值，以防错误吞没。**

如果说代码风格难看是小事，但栈撕裂和状态管理困难这两个缺点会让基于回调的异步编程很艰难。所以不同编程语言的生态都在致力于解决这个问题。才诞生了后来的**Promise、Co-routine**等解决方案。

Python 生态也以终为始，秉承着"程序猿不必难程序猿"的原则，让语言和框架开发者苦逼一点，也要让应用开发者舒坦。在**事件循环+回调**的基础上衍生出了基于协程的解决方案，代表作有 **Tornado、Twisted、asyncio** 等。接下来我们随着 Python 生态异步编程的发展过程，深入理解Python异步编程。

4.2 核心问题

通过前面的学习，我们清楚地认识到异步编程最大的困难：异步任务何时执行完毕？接下来要对异步调用的返回结果做什么操作?

上述问题我们已经通过事件循环和回调解决了。但是回调会让程序变得复杂。要异步，必回调，又是否有办法规避其缺点呢？那需要弄清楚其本质，为什么回调是必须的？还有使用回调时克服的那些缺点又是为了什么？

答案是程序为了知道自己已经干了什么？正在干什么？将来要干什么？**换言之，程序得知道当前所处的状态，而且要将这个状态在不同的回调之间延续下去。**

多个回调之间的状态管理困难，那让每个回调都能管理自己的状态怎么样？链式调用会有栈撕裂的困难，让回调之间不再链式调用怎样？不链式调用的话，那又如何让被调用者知道已经完成了？那就让这个回调通知那个回调如何？而且一个回调，不就是一个待处理任务吗?

任务之间得相互通知，每个任务得有自己的状态。那不就是很古老的编程技法：协作式多任务？然而要在单线程内做调度，**啊哈，协程！**每个协程具有自己的栈帧，当然能知道自己处于什么状态，协程之间可以协作那自然可以通知别的协程。

4.3 协程

- **协程(Co-routine)，即是协作式的例程。**

它是非抢占式的多任务子例程的概括，可以允许有多个入口点在例程中确定的位置来控制程序的暂停与恢复执行。

例程是什么？编程语言定义的可被调用的代码段，为了完成某个特定功能而封装在一起的一系列指令。一般的编程语言都用称为函数或方法的代码结构来体现。

4.4 基于生成器的协程

早期的 Pythoner 发现 Python 中有种特殊的对象——生成器（Generator），它的特点和协程很像。每一次迭代之间，会暂停执行，继续下一次迭代的时候还不会丢失先前的状态。

为了支持用生成器做简单的协程，Python 2.5 对生成器进行了增强(PEP 342)，该增强提案的标题是 "Coroutines via Enhanced Generators"。有了PEP 342的加持，生成器可以通过**yield**暂停执行和向外返回数据，也可以通过**send()**向生成器内发送数据，还可以通过**throw()**向生成器内抛出异常以便随时终止生成器的运行。

接下来，我们用基于生成器的协程来重构先前的爬虫代码。

4.4.1 未来对象(Future)

不用回调的方式了，怎么知道异步调用的结果呢？先设计一个对象，异步调用执行完的时候，就把结果放在它里面。这种对象称之为未来对象。

```python
import socket
from selectors import DefaultSelector, EVENT_WRITE, EVENT_READ

selector = DefaultSelector()
stopped = False
urls_todo = {'/', '/1', '/2', '/3', '/4', '/5', '/6', '/7', '/8', '/9'}


class Future:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for fn in self._callbacks:
            fn(self)
```

未来对象有一个**result**属性，用于存放未来的执行结果。还有个**set_result()**方法，是用于设置**result**的，并且会在给**result**绑定值以后运行事先给future添加的回调。回调是通过未来对象的**add_done_callback()**方法添加的。

不要疑惑此处的**callback**，说好了不回调的嘛？难道忘了我们曾经说的**要异步，必回调**。不过也别急，此处的回调，和先前学到的回调，还真有点不一样。

4.4.2 重构 Crawler

现在不论如何，我们有了未来对象可以代表未来的值。先用**Future**来重构爬虫代码。

```python
class Crawler:
    def __init__(self, url):
        self.url = url
        self.response = b''

    def fetch(self):
        sock = socket.socket()
        sock.setblocking(False)
        try:
            sock.connect(('example.com', 80))
        except BlockingIOError:
            pass
        f = Future()

        def on_connected():
            f.set_result(None)

        selector.register(sock.fileno(), EVENT_WRITE, on_connected)
        yield f
        selector.unregister(sock.fileno())
        get = 'GET {0} HTTP/1.0\r\nHost: example.com\r\n\r\n'.format(self.url)
        sock.send(get.encode('ascii'))

        global stopped
        while True:
            f = Future()

            def on_readable():
                f.set_result(sock.recv(4096))

            selector.register(sock.fileno(), EVENT_READ, on_readable)
            chunk = yield f
            selector.unregister(sock.fileno())
            if chunk:
                self.response += chunk
            else:
                urls_todo.remove(self.url)
                if not urls_todo:
                    stopped = True
                break
```

和先前的回调版本对比，已经有了较大差异。fetch 方法内有了yield表达式，使它成为了生成器。我们知道生成器需要先调用**next()**迭代一次或者是先**send(None)**启动，遇到yield之后便暂停。那这fetch生成器如何再次恢复执行呢？至少 **Future** 和 **Crawler**都没看到相关代码。

4.4.3 任务对象(Task)

为了解决上述问题，我们只需遵循一个编程规则：单一职责，每种角色各司其职，如果还有工作没有角色来做，那就创建一个角色去做。没人来恢复这个生成器的执行么？没人来管理生成器的状态么？创建一个，就叫**Task**好了，很合适的名字。

```
68 class Task:
69     def __init__(self, coro):
70         self.coro = coro
71         f = Future()
72         f.set_result(None)
73         self.step(f)
74 
75     def step(self, future):
76         try:
77             # send会进入到coro执行，即fetch，直到下次yield
78             # next_future 为yield返回的对象
79             next_future = self.coro.send(future.result)
80         except StopIteration:
81             return
82         next_future.add_done_callback(self.step)
```

上述代码中**Task**封装了**coro**对象，即初始化时传递给他的对象，被管理的任务是待执行的协程，故而这里的coro就是**fetch()**生成器。它还有个**step()**方法，在初始化的时候就会执行一遍。**step()**内会调用生成器的**send()**方法，初始化第一次发送的是None就驱动了coro即**fetch()**的第一次执行。

send()完成之后，得到下一次的**future**，然后给下一次的**future**添加**step()**回调。原来**add_done_callback()**不是给写爬虫业务逻辑用的。此前的**callback**可就干的是业务逻辑呀。

再看**fetch()**生成器，其内部写完了所有的业务逻辑，包括如何发送请求，如何读取响应。而且注册给selector的回调相当简单，就是给对应的**future**对象绑定结果值。两个yield表达式都是返回对应的future对象，然后返回**Task.step()**之内，这样**Task, Future, Coroutine**三者精妙地串联在了一起。

初始化Task对象以后，把**fetch()**给驱动到了第44行**yied f**就完事了，接下来怎么继续?

4.4.4 事件循环(Event Loop)驱动协程运行

该事件循环上场了。接下来，只需等待已经注册的EVENT_WRITE事件发生。事件循环就像心脏一般，只要它开始跳动，整个程序就会持续运行。

```
85 def loop():
86     while not stopped:
87         # 阻塞，直到一个事件发生
88         events = selector.select()
89         for event_key, event_mask in events:
90             callback = event_key.data
91             callback()
92 
93 
94 if __name__ == '__main__':
95     import time
96     start = time.time()
97     for url in urls_todo:
98         crawler = Crawler(url)
99         Task(crawler.fetch())
100     loop()
101     print(time.time() - start)
```

注：总体耗时约0.43秒。

现在loop有了些许变化，**callback()**不再传递**event_key**和**event_mask**参数。也就是说，这里的回调根本不关心是谁触发了这个事件，结合**fetch()**可以知道，它只需完成对**future**设置结果值即可**f.set_result()**。而且**future**是谁它也不关心，因为**协程**能够保存自己

的状态，知道自己的**future**是哪个。也不用关心到底要设置什么值，因为要设置什么值也是协程内安排的。

此时的loop()，真的成了一个心脏，它只管往外泵血，不论这份血液是要输送给大脑还是要给脚趾，只要它还在跳动，生命就能延续。

4.4.5 生成器协程风格和回调风格对比总结

在回调风格中：

- 存在链式回调（虽然示例中嵌套回调只有一层）
- 请求和响应也不得不分为两个回调以至于破坏了同步代码那种结构
- 程序员必须在回调之间维护必须的状态。

还有更多示例中没有展示，但确实存在的问题，参见4.1节。

而基于生成器协程的风格：

- 无链式调用
- **selector**的回调里只管给**future**设置值，不再关心业务逻辑
- loop 内回调**callback()**不再关注是谁触发了事件
- 已趋近于同步代码的结构
- 无需程序员在多个协程之间维护状态，例如哪个才是自己的**sock**

4.4.6 碉堡了，但是代码很丑！能不能重构?

如果说**fetch**的容错能力要更强，业务功能也需要更完善，怎么办？而且技术处理的部分（socket相关的）和业务处理的部分（请求与返回数据的处理）混在一起。

- 创建**socket**连接可以抽象复用吧?
- 循环读取整个**response**可以抽象复用吧?
- 循环内处理**socket.recv()**的可以抽象复用吧?

但是这些关键节点的地方都有**yield**，抽离出来的代码也需要是生成器。而且**fetch()**自己也得是生成器。生成器里玩生成器，代码好像要写得更丑才可以……

Python 语言的设计者们也认识到了这个问题，再次秉承着"程序猿不必为难程序猿"的原则，他们捣鼓出了一个**yield from**来解决生成器里玩生成器的问题。

4.5 用 yield from 改进生成器协程

4.5.1 yield from语法介绍

yield from 是Python 3.3 新引入的语法（PEP 380）。它主要解决的就是在生成器里玩生成器不方便的问题。它有两大主要功能。

第一个功能是：让嵌套生成器不必通过循环迭代yield，而是直接yield from。以下两种在生成器里玩子生成器的方式是等价的。

```
1 def gen_one():
2     subgen = range(10)    yield from subgendef gen_two():
3     subgen = range(10)    for item in subgen:        yield item
```

第二个功能就是在子生成器和原生成器的调用者之间打开双向通道，两者可以直接通信。

```
1  def gen():
2      yield from subgen()def subgen():
3      while True:
4          x = yield
5          yield x+1def main():
6      g = gen()
7      next(g)                # 驱动生成器g开始执行到第一个 yield
8      retval = g.send(1)     # 看似向生成器 gen() 发送数据
9      print(retval)          # 返回2
10     g.throw(StopIteration) # 看似向gen()抛入异常
```

通过上述代码清晰地理解了yield from的双向通道功能。关键字yield from在gen()内部为subgen()和main()开辟了通信通道。main()里可以直接将数据1发送给subgen(),subgen()也可以将计算后的数据2返回到main()里，main()里也可以直接向subgen()抛入异常以终止subgen()。

顺带一提，yield from 除了可以 yield from 还可以 yield from 。

4.5.2 重构代码

抽象socket连接的功能：

```
def connect(sock, address):
    f = Future()
    sock.setblocking(False)
    try:
        sock.connect(address)
    except BlockingIOError:
        pass

    def on_connected():
        f.set_result(None)

    selector.register(sock.fileno(), EVENT_WRITE, on_connected)
    yield from f
    selector.unregister(sock.fileno())
```

抽象单次**recv()**和读取完整的response功能:

```
def read(sock):
    f = Future()

    def on_readable():
        f.set_result(sock.recv(4096))

    selector.register(sock.fileno(), EVENT_READ, on_readable)
    chunk = yield from f
    selector.unregister(sock.fileno())
    return chunk


def read_all(sock):
    response = []
    chunk = yield from read(sock)
    while chunk:
        response.append(chunk)
        chunk = yield from read(sock)
    return b''.join(response)
```

三个关键点的抽象已经完成，现在重构**Crawler**类:

```
67 class Crawler:
68     def __init__(self, url):
69         self.url = url
70         self.response = b''
71 
72     def fetch(self):
73         global stopped
74         sock = socket.socket()
75         yield from connect(sock, ('example.com', 80))
76         get = 'GET {0} HTTP/1.0\r\nHost: example.com\r\n\r\n'.format(self.url)
77         sock.send(get.encode('ascii'))
78         self.response = yield from read_all(sock)
79         urls_todo.remove(self.url)
80         if not urls_todo:
81             stopped = True
```

上面代码整体来讲没什么问题，可复用的代码已经抽象出去，作为子生成器也可以使用 **yield from** 语法来获取值。但另外有个点需要**注意**：在第24和第35行返回**future**对象的时候，我们了**yield from f** 而不是原来的**yield f**。**yield**可以直接作用于普通Python对象，而**yield from**却不行，所以我们对**Future**还要进一步改造，把它变成一个**iterable**对象就可以了。

```
49 class Future:
50     def __init__(self):
51         self.result = None
52         self._callbacks = []
53 
54     def add_done_callback(self, fn):
55         self._callbacks.append(fn)
56 
57     def set_result(self, result):
58         self.result = result
59         for fn in self._callbacks:
60             fn(self)
61 
62     def __iter__(self):
63         yield self
64         return self.result
```

只是增加了**__iter__()**方法的实现。如果不把**Future**改成**iterable**也是可以的，还是用原来的**yield** f即可。那为什么需要改进呢?

首先，我们是在基于生成器做协程，而生成器还得是生成器，如果继续混用**yield**和**yield from** 做协程，代码可读性和可理解性都不好。其次，如果不改，协程内还得关心它等待的对象是否可被yield，如果协程里还想继续返回协程怎么办？如果想调用普通函数动态生成一个**Future**对象再返回怎么办?

所以，在Python 3.3 引入**yield from**新语法之后，就不再推荐用yield去做协程。全都使用**yield from**由于其双向通道的功能，可以让我们在协程间随心所欲地传递数据。

4.5.3 yield from改进协程总结

用**yield from**改进基于生成器的协程，代码抽象程度更高。使业务逻辑相关的代码更精简。由于其双向通道功能可以让协程之间随心所欲传递数据，使Python异步编程的协程解决方案大大向前迈进了一步。

于是Python语言开发者们充分利用**yield from**，使 **Guido** 主导的Python异步编程框架**Tulip**迅速脱胎换骨，并迫不及待得让它在Python 3.4 中换了个名字**asyncio**以"实习生"角色出现在标准库中。

4.5.4 asyncio 介绍

asyncio是Python 3.4 试验性引入的异步I/O框架（PEP 3156），提供了基于协程做异步I/O编写单线程并发代码的基础设施。其核心组

件有事件循环（Event Loop）、协程(Coroutine)、任务(Task)、未来对象(Future)以及其他一些扩充和辅助性质的模块。

在引入asyncio的时候，还提供了一个装饰器**@asyncio.coroutine**用于装饰使用了**yield from**的函数，以标记其为协程。但并不强制使用这个装饰器。

虽然发展到 Python 3.4 时有了**yield from**的加持让协程更容易了，但是由于协程在Python中发展的历史包袱所致，很多人仍然弄不明白**生成器**和**协程**的联系与区别，也弄不明白**yield** 和 **yield from** 的区别。这种混乱的状态也违背Python之禅的一些准则。

于是Python设计者们又快马加鞭地在 3.5 中新增了**async/await**语法（PEP 492），对协程有了明确而显式的支持，称之为**原生协程**。**async/await** 和 **yield from**这两种风格的协程底层复用共同的实现，而且相互兼容。

在Python 3.6 中**asyncio**库“转正”，不再是实验性质的，成为标准库的正式一员。

4.6 总结

行至此处，我们已经掌握了**asyncio**的核心原理，学习了它的原型，也学习了异步I/O在 CPython 官方支持的生态下是如何一步步发展至今的。

实际上，真正的**asyncio**比我们前几节中学到的要复杂得多，它还实现了零拷贝、公平调度、异常处理、任务状态管理等等使 Python 异步编程更完善的内容。理解原理和原型对我们后续学习有莫大的帮助。

5 asyncio和原生协程初体验

本节中，我们将初步体验**asyncio**库和新增语法**async/await**给我们带来的便利。由于Python2-3的过度期间，Python3.0-3.4的使用者并不是太多，也为了不让更多的人困惑，也因为**aysncio**在3.6才转正，所以更深入学习asyncio库的时候我们将使用**async/await**定义的原生协程风格，**yield from**风格的协程不再阐述（实际上它们可用很小的代价相互代替）。

```python
import asyncio
import aiohttp

host = 'http://example.com'
urls_todo = {'/', '/1', '/2', '/3', '/4', '/5', '/6', '/7', '/8', '/9'}

loop = asyncio.get_event_loop()


async def fetch(url):
    async with aiohttp.ClientSession(loop=loop) as session:
        async with session.get(url) as response:
            response = await response.read()
            return response


if __name__ == '__main__':
    import time
    start = time.time()
    tasks = [fetch(host + url) for url in urls_todo]
    loop.run_until_complete(asyncio.gather(*tasks))
    print(time.time() - start)
```

对比生成器版的协程，使用**asyncio**库后变化很大：

- 没有了**yield** 或 **yield from**，而是**async/await**
- 没有了自造的**loop()**，取而代之的是**asyncio.get_event_loop()**
- 无需自己在**socket**上做异步操作，不用显式地注册和注销事件，**aiohttp**库已经代劳
- 没有了显式的 **Future** 和 **Task**，**asyncio**已封装
- 更少量的代码，更优雅的设计

说明：我们这里发送和接收HTTP请求不再自己操作**socket**的原因是，在实际做业务项目的过程中，要处理妥善地HTTP协议会很复杂，我们需要的是功能完善的异步HTTP客户端，业界已经有了成熟的解决方案，DRY不是吗?

和同步阻塞版的代码对比：

- 异步化

- 代码量相当（引入**aiohttp**框架后更少）
- 代码逻辑同样简单，跟同步代码一样的结构、一样的逻辑
- 接近10倍的性能提升

结语

到此为止，我们已经深入地学习了异步编程是什么、为什么、在Python里是怎么样发展的。我们找到了一种让代码看起来跟同步代码一样简单，而效率却提升N倍（具体提升情况取决于项目规模、网络环境、实现细节）的异步编程方法。它也没有回调的那些缺点。

本系列教程接下来的一篇将是学习**asyncio**库如何的使用，快速掌握它的主要内容。后续我们还会深入探究**asyncio**的优点与缺点，也会探讨Python生态中其他异步I/O方案和asyncio的区别。

附

参考资料

- 《A Web Crawler With asyncio Coroutines》
- 《让 CPU 告诉你硬盘和网络到底有多慢》

相关代码

- http://github.com/denglj/aiotutorial