<译> A Web Crawler With asyncio Coroutines

重点不在爬虫，值得学习！估计也只有我能看懂自己的译文。。。

Python (/categories/#python) - Coroutine (/tags/#coroutine) 翻译 (/tags/#翻译)

By Damnever on October 12, 2015

糙译还请指教！我最大的敌人是我自己，看我如何击败我……

原文链接：http://aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html (http://aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html)

作者：A. Jesse Jiryu Davis and Guido van Rossum

A. Jesse Jiryu Davis 是纽约 MongoDB 的一名工程师。他编写了 Motor，一个异步的 MongoDB Python 驱动程序，而且，他也是 MongoDB 的 C语言驱动程序的主力开发者，同时也是 PyMongo 团队的成员。他为 asyncio 和 Tornado 贡献过源码，他写的东西在：http://emptysqua.re/ (http://emptysqua.re/)。
Guido van Rossum 是 Python 的创始人，Python 是互联网上的主要编程语言之一，Python 社区中 BDFL（Benevolent Dictator For Life：仁慈的独裁者）指的就是他，这个头衔来源于 Monty Python 短剧。Guido 的 Web 主页是 http://www.python.org/~guido/ (http://www.python.org/~guido/)。

介绍

传统的计算机科学强高效的算法，尽可能快的完成计算。但是大多数网络程序花费的时间不在计算上，而是保持打开许多慢速链接，或者有极少的事件。这些程序面临着一个非常不同的挑战：高效的等待数量巨大的网络事件。一个处理这一问题的现代方法是异步I/O，或“async”。

这章介绍了一个简单的爬虫。这个爬虫是一个典型的异步应用程序，因为它等待许多响应，但是做很少的计算。它一次能获取的网页越多，就越早完成。如果它致力于每个线程一个请求，那么当并发的请求越来越多，在耗尽 socket 之前它将耗尽内存或者其它线程相关的资源。使用异步 I/O 能避免线程的必要性。

我们的示例分三个阶段。第一阶段，我们展现一个异步的事件循环，并且勾画了一个使用事件循环和回调的爬虫：它非常高效，但是衍生出的更复杂问题会导致难以管理的面条式的代码。第二阶段，因此我们展现了 Python 协程的高效和可扩展。我们用 Python 的生成器函数实现了简单的协程。第三阶段，我们使用 Python “asyncio” 标准库中功能全面的协程，并用一个异步队列来协调它们。

任务

一个网页爬虫找出一个网站中的所有网页并下载，可能存档或者索引它们。从根 URL 开始，它抓取一个网页，从中解析出还没抓取过的网页链接，并且把新链接存入队列中。当抓取到一个网页里面到没有未抓取过的链接，并且队列为空时，它就停止。

我们可以通过并发的下载网页来加速这个过程。当爬虫找到新链接时，它在不同的 socket 中同时启动抓取网页的操作。解析到达的响应并将新链接添加到队列中。可能会出现一些收益递减点，由于并发太多而导致性能降低，因此我们限制并发请求的数量，并且把剩下的链接保留在队列里直到一些请求完成。

传统的方式

我们怎样才能使爬虫变成并发的？传统上我们将会创建一个线程池。每个线程负责在一个 socket 中一次下载一个网页。例如，从 xkcd.com 下载一个网页：

```
def fetch(url):
    sock = socket.socket()
    sock.connect(('xkcd.com', 80))
    request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        chunk = sock.recv(4096)

    # Page is now downloaded.
    links = parse_links(response)
    q.add(links)
```

socket 操作默认是阻塞的：当线程调用一个像 connect 或 recv 的方法，他将会暂停直到操作完成。因此一次下载很多网页的话，我们需要很多线程。一个复杂应用程序通过线程池中保持空闲的线程来平摊线程创建的开销，然后检查它们来为后续任务重用；在连接池中 socket 做着相同的事。

然而，线程开销很大，并且操作系统限制了一个进程、用户、或者机器所能拥有的线程数量。在 Jesse 的系统中，一个 Python 线程占用 50K 的内存，然后开启成千上万的线程会以失败告终。如果我们扩大同时操作在并发 socket 上的线程到数以万计，我们会在耗尽 socket 之前耗尽线程.线程的开销或者系统对线程的限制是瓶颈。

在 Dan Kegel 富有影响力的文章 “The C10K problem” 中，描述了多线程并发 I/O 的局限性，他说：

> *是时候让 Web 服务器能同时处理一万个客户端了，你不这样认为吗？毕竟，Web 已经是个大地方了。*

Kegel 在1999年创造了“C10k”这个词。1万的连接现在听起来文雅，然而这个问题没有发生质变，只发生了量变。当时，为每个连接开启一个线程的方式来解决 C10K 问题是不现实的。现在这个数量级的上限更高了。确实，我们的玩具爬虫用线程也能工作得很好。然而对于超大规模的应用，有成千上万的连接，限制依然存在：在限制下大多数系统仍能够创建 socket，但是线程还是会耗尽。我们怎样才能攻克这个难题？

异步

异步 I/O 框架能在一个单独的线程里处理并发的操作。让我们瞧瞧这是如何做到的。

异步框架使用非阻塞的 sockets。在我们的异步爬虫中，在连接到服务端之前，我们设置 socket 为非阻塞的。

```
sock = socket.socket()
sock.setblocking(False)
try:
    sock.connect(('xkcd.com', 80))
except BlockingIOError:
    pass
```

烦人的是，非阻塞的 socket 从 connect 抛出异常，即使在正常工作时候。这个异常来自底层的 C 函数，它设置 errno 为 EINPROGRESS 来告诉你异常发生了。

现在我们的爬虫需要一个方法去知道连接是什么时候建立的，这样我们才能发送 HTTP 请求。很简单，我们能用循环不断尝试：

```
request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
encoded = request.encode('ascii')

while True:
    try:
        sock.send(encoded)
        break  # Done.
    except OSError as e:
        pass

print('sent')
```

上面这个方法不仅费电，而且不能在多个套接字上等待事件。过去，BSD Unix 的解决方案是 select ，一个等待事件发生在一个或者一组非阻塞 socket 上的 C 函数。现在由于互联网程序面对的连接巨大，逐步使用 poll 取代，之后是 BSD 上的 kqueue ，Linux 上的 epoll 。他们的接口与 select 相似，但是有大量连接时表现的很好。

Python 3.4 的 DefaultSelector 根据你所用的系统来使用最好的类 select 函数。为了给网络 I/O 注册通知，我们创建了一个非阻塞的 socket 并且用 DefaultSelector 来注册它。

```
from selectors import DefaultSelector, EVENT_WRITE

selector = DefaultSelector()

sock = socket.socket()
sock.setblocking(False)
try:
    sock.connect(('xkcd.com', 80))
except BlockingIOError:
    pass

def connected():
    selector.unregister(sock.fileno())
    print('connected!')

selector.register(sock.fileno(), EVENT_WRITE, connected)
```

我们忽略了欺骗性的错误并调用了 selector.register ，传入了 socket 的文件描述符和一个常数表达了我们要等待的事件。为了在连接建立后收到通知，我们传入 EVENT_WRITE ：我们想知道什么时候 socket 是“可写的”。我们也传入了一个 Python 函数 connected ，在事件发生的时候执行。这个函数被称作回调。

我们在 selector 收到 I/O 通知的时候处理它们，在一个循环里：

```
def loop():
    while True:
        events = selector.select()
        for event_key, event_mask in events:
            callback = event_key.data
            callback()
```

connected 回调函数存储在 event_key.data 里面，一旦连接建立我们就取得这个回调函数并执行一次。

不像我们前面快速运转的循环，调用 select 的时候会暂停，直到下一个 I/O 事件发生。然后再循环里执行这些一直在等待事件的回调函数。操作还未完成的继续等待直到后面事件循环的信号到来。

我们已经展示了什么？我们展示了怎么样去开始一个操作并且在操作准备好的时候去执行它。一个异步的框架建立在我们已经展示过的两个特性上–非阻塞的 socket 和事件循环–在一个单独的线程里执行并发操作。

这里我们已经完成了“并发”，但是不是传统上被称作的“并行”。也就是说，我们打造了一个能复用 I/O 的小系统。当其它的操作还在运行的时候我们能够开始新的操作。它实际上并没有利用多核去执行并行计算。不过，这个系统为解决 I/O 密集型的问题而设计，而不是 CPU 密集型。

因此我们的事件循环在并发 I/O 上很高效，因为它不需要为每个连接都投入线程资源。但在我们继续之前，纠正一个普遍的误解是很重要的，那就是异步比多线程要快。往往不是–确实，在 Python 中，我们的事件循环在服务数量很少的活跃连接的时候比多线程要慢一些。在一个没有全局性解释器锁的运行时，线程会在这样的工作负载上表现得更好。异步 I/O 的使用场景是什么，是有很多慢速或者困倦的连接与罕见的事件的应用程序。

用回调编程

到目前为止我们建立了短小的异步框架，我们怎样才能建立一个 Web 爬虫呢？即便是一个简单的 URL 提取写起来也很痛苦。

我们用全局的 URL 集合开始，一个是我们还没提取的，一个是我们已经见过的：

```
urls_todo = set(['/'])
seen_urls = set(['/'])
```

seen_urls 是 urls_todo 和已经完成的 URL 的总和。这两个集合用根 URL “/”来初始化。

提取一个网页需要一系列的回调函数。 connected 回调在 socket 已经连接的时候触发，然后发送一个 GET 请求到服务端。然后它必须等待响应，因此我们注册另一个回调。如果回调触发的时候，它不能完整的读取完整个响应，就再注册一次，如此反复。

让我们把这些回调收集在一个 Fetcher 对象里。他需要一个 URL，一个 socket 对象，和一个空间去累积响应的字节数据：

```
class Fetcher:
    def __init__(self, url):
        self.response = b''  # Empty array of bytes.
        self.url = url
        self.sock = None
```

我们通过调用 Fetcher.fetch 来开始：

```
    # Method on Fetcher class.
    def fetch(self):
        self.sock = socket.socket()
        self.sock.setblocking(False)
        try:
            self.sock.connect(('xkcd.com', 80))
        except BlockingIOError:
            pass

        # Register next callback.
        selector.register(self.sock.fileno(),
                          EVENT_WRITE,
                          self.connected)
```

fetch 方法开始连接一个 socket。但是请注意，这个连接建立之前方法就返回了。它必须将控制权还给事件循环去等待连接。为了理解为什么，想象一下我们的整个应用结构是这样的：

```
# Begin fetching http://xkcd.com/353/
fetcher = Fetcher('/353/')
fetcher.fetch()

while True:
    events = selector.select()
    for event_key, event_mask in events:
        callback = event_key.data
        callback(event_key, event_mask)
```

所有的事件通知都在调用 select 的时候被处理。因此 fetch 必须把控制权交给事件循环，这样程序才能知道什么时候 socket 已经连接了。只有之后循环执行 connected 回调的时候， fetch 最后的动作被注册。

```
    # Method on Fetcher class.
    def connected(self, key, mask):
        print('connected!')
        selector.unregister(key.fd)
        request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(self.url)
        self.sock.send(request.encode('ascii'))

        # Register the next callback.
        selector.register(key.fd,
                          EVENT_READ,
                          self.read_response)
```

这个方法发送一个 GET 请求。真正意义上的应用程序会检测 send 的返回值，以防不能一次发完消息。但是我们的请求很小并且应用程序很简单。它无顾虑的调用 send ，然后等待响应。当然，还必须注册另一个回调并且放弃控制权以交给事件循环。下一个也是最后一个回调，处理服务器的回应：

```
    # Method on Fetcher class.
    def read_response(self, key, mask):
        global stopped

        chunk = self.sock.recv(4096)  # 4k chunk size.
        if chunk:
            self.response += chunk
        else:
            selector.unregister(key.fd)  # Done reading.
            links = self.parse_links()

            # Python set-logic:
            for link in links.difference(seen_urls):
                urls_todo.add(link)
                Fetcher(link).fetch()  # <- New Fetcher.

            seen_urls.update(links)
            urls_todo.remove(self.url)
            if not urls_todo:
                stopped = True
```

这个回调在 selector 发现 socket 是可读的时候执行一次，这可能意味着两件事：socket 有数据可读或者它被关闭了。

回调要求从 socket 读取 4KB 的数据。如果少于 4KB 数据准备好了， chunk 包含任何可用的数据。如果多于 4KB， chunk 长度为 4KB 并且 socket 仍然保持可读状态，因此事件循环在下一次时钟滴答的时候再次执行这个回调。当响应完成时，服务器关闭 socket 并且 chunk 为空。

parse_links 方法这里不展示，它返回一个 URL 集合。我们为每个新 URL 开始一个新的 fetcher，不考虑并发上限。注意用回调异步编程的一个特性：我们在改变共享数据的时候不需要互斥，例如我们添加链接到 seen_urls 。没有先发制人的多任务，所以我们代码中的任何地方都不会被中断。

我们添加了一个全局 stopped 并用它来控制循环：

```
stopped = False

def loop():
    while not stopped:
        events = selector.select()
        for event_key, event_mask in events:
            callback = event_key.data
            callback()
```

一旦所有的网页被下载完成，fetcher 就停止全局的事件循环然后程序退出。

这个例子使异步问题变得简单：面条式代码。

我们需要一些方式来表达一系列的计算和 I/O 操作，并且调度许多一系列这样的操作并发的执行。但是不用线程，一系列这样的操作不能被放在一个单独的函数里面：无论什么时候一个函数开始一个 I/O 操作，它都需要明确的保存那些后面会用到的状态，然后返回。你得尽责的思考并编写相关的状态保存代码。

让我们解释我们在说什么。考虑下我们用传统的阻塞的 socket 在一个线程了抓取一个网页是多简单：

```
# Blocking version.
def fetch(url):
    sock = socket.socket()
    sock.connect(('xkcd.com', 80))
    request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
        response += chunk
        chunk = sock.recv(4096)

    # Page is now downloaded.
    links = parse_links(response)
    q.add(links)
```

在一个 socket 操作和下一个之间这个函数保存了什么状态？它有一个 socket，一个 URL，还能累 response 。一个在线程里运行的函数用编程语言最基础的特性把临时状态保存在局部变量里面，即栈里面。这个函数也有“continuation”–也就是说，代码能在 I/O 操作完成后继续执行。运行时通过保存线程的指令指针来记住继续的地方。你不需要在 I/O 操作之后考虑恢复这些局部变量并继续。这是编程语言本身实现的。

但是基于回调的异步框架，语言特性是没有帮助的。当等待 I/O 的时候，函数必须明确的保存它的状态，因为函数返回之后会在 I/O 完成之前丢失栈贞。代替局部变量的是，我们基于回调例子存储 sock 和 response 作为 self 的属性， self 是 Fetcher 的实例。代替指令指针的是，通过注册回调 connected 和 read_response 来存储继续执行的代码（it stores its continuation？？）。当应用程序的复杂性增加，我们手动存储回调之间状态的复杂性也会增加。如此繁重的记账会让编码者越来越头痛。

更糟的是，在调度回调链的下一个回调的时候，如果回调抛出异常的话就不知道确切的错误。例如我们在 parse_links 方法里写的代码很糟，然后在解析 HTML 的时候抛出了异常：

```
Traceback (most recent call last):
  File "loop-with-callbacks.py", line 111, in <module>
    loop()
  File "loop-with-callbacks.py", line 106, in loop
    callback(event_key, event_mask)
  File "loop-with-callbacks.py", line 51, in read_response
    links = self.parse_links()
  File "loop-with-callbacks.py", line 67, in parse_links
    raise Exception('parse error')
Exception: parse error
```

堆栈跟踪只显示了事件循环正在调用一个回调。它并不知道是什么导致了这个错误。链条的两端都断了：我们忘记了我们要去哪里也不知道我们从哪里来的。这样的上下文丢失被叫做“stack ripping”，并且在大多数情况下都使研究者迷惑。堆栈撕裂也阻止了我们给回调链添加一个异常处理程序，用 try/except 块包裹函数及其子孙后代的方式是行不通的。

因此，除了关于多线程和异步的效率的长期讨论，还有其它的关于谁更易出错的讨论：线程对数据的竞争是敏感的如果你在同步它们的时候犯了错误，但是回调由于堆栈撕裂是极难调试的。

协程

我们用一个承若引诱你。写出兼具回调的高效性和传统线程方式的可读性的异步代码是可能的。这个组合通过一个叫做“协程”的模式实现。使用 Python 3.4 的 asyncio 标准库和一个叫做“aiohttp”的包，在一个协程里面抓取一个 URL 很直接：

```
    @asyncio.coroutine
    def fetch(self, url):
        response = yield from self.session.get(url)
        body = yield from response.read()
```

它也是可扩展的。相比于每个线程占用 50K 的内存和操作系统对线程数量的限制，在 Jesse 的系统上一个 Python 协程最多占用 30K 内存。Python 能轻易的启动成千上万的协程。

协程的概念，在计算机科学的老年时代，很简单：一个子程序能被暂停和回复。鉴于线程通过操作系统调度的先入为主的任务，协程的多任务是相互合作的：它们自己决定什么时候停止，哪个协程接下来运行。

协程有很多实现方式；甚至在 Python 里面都有几种。Python 3.4 的 asyncio 标准库中的协程是建立在生成器上的，一个 Future 类和 `yield from` 语句。从 Python 3.5 开始，协程是语言原生的特性；当然，考虑到协程第一次在 Python 3.4 中实现，使用以经存在的语言设施，是阻挡 Python 3.5 原生协程的基础。

为了解释 Python 3.4 中基于生成器的协程，我们将从生成器开始阐述然后说明它们是怎样在 ayncio 中用作协程的，相信你将会乐意去阅读这些东西就像我们很乐意写下这些东西一样。一旦我们解释完基于生成器的协程，我们将把它运用在我们的异步 Web 爬虫中。

Python 生成器是如何工作的

在我们了解 Python 生成器之前，你必须理解 Python 的正常函数是如何工作的。一般来说，一个 Pyhotn 函数被称作一个子程序，子程序拥有控制权直到它返回或者抛出异常。然后控制权返还给调用者：

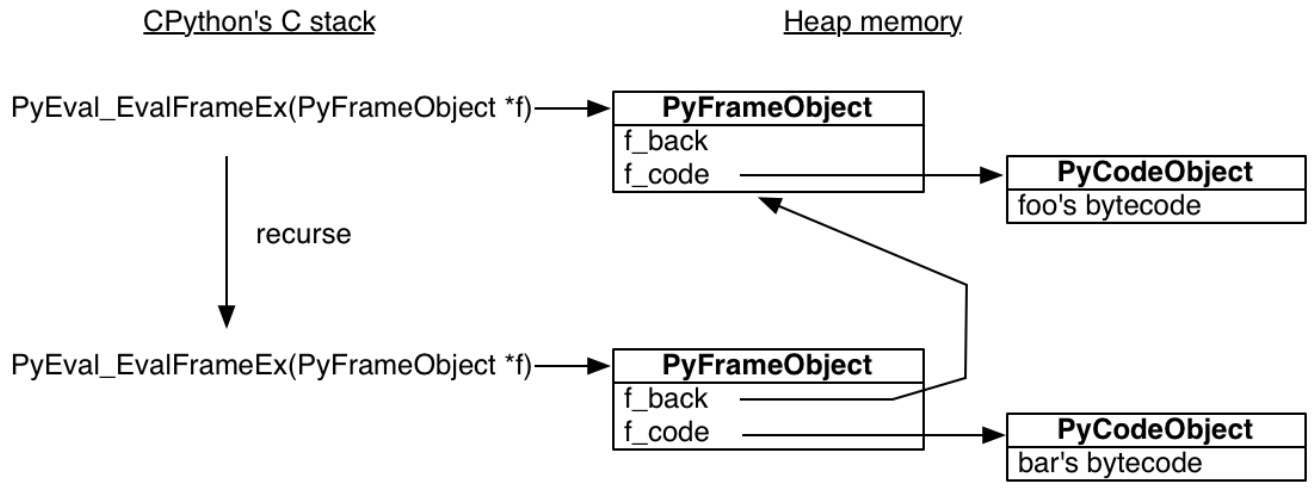
```
>>> def foo():
...     bar()
...
>>> def bar():
...     pass
```

标准的 Python 解释器使用 C 写的。C 函数执行一个被调用的 Python 函数，mellifluously， `PyEval_EvalFrameEx` 。它接受一个 Python 栈贞对象并计算贞上下文中 Python 的字节码。这是 `foo` 的字节码：

```
>>> import dis
>>> dis.dis(foo)
  2           0 LOAD_GLOBAL              0 (bar)
              3 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
              6 POP_TOP
              7 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

foo 函数把 bar 加载到自己的栈里面并调用它，然后把它的返回值从栈里弹出来，再把 None 加载到栈里面，然后返回 None 。

当 PyEval_EvalFrameEx 遇到 CALL_FUNCTION 字节码时，它创建一个新的 Python 栈贞并递归：也就是说，通过新贞递归的调用 PyEval_EvalFrameEx ，以此来调用 bar 。



关键是要知道 Python 的栈贞是在堆上分配的内存！Python 解释器是正常的 C 程序，因此它的栈贞是正常的栈贞。但是 Python 的栈贞是在堆上面操作的。其它的惊喜，这意味着 Python 的栈贞能存活在函数调用之外。在交互式解释器里看看，在 bar 里保存当前栈贞：

```
>>> import inspect
>>> frame = None
>>> def foo():
...     bar()
...
>>> def bar():
...     global frame
...     frame = inspect.currentframe()
...
>>> foo()
>>> # The frame was executing the code for 'bar'.
>>> frame.f_code.co_name
'bar'
>>> # Its back pointer refers to the frame for 'foo'.
>>> caller_frame = frame.f_back
>>> caller_frame.f_code.co_name
'foo'
```

现阶段为 Python 生成器，用相同的方式–代码对象和栈贞–推此及彼。

这是一个生成器函数：

```
>>> def gen_fn():
...     result = yield 1
...     print('result of yield: {}'.format(result))
...     result2 = yield 2
...     print('result of 2nd yield: {}'.format(result2))
...     return 'done'
...
```

当 Python 把 gen_fn 编译成字节码时，它看到 yield 语句就知道 gen_fn 是一个生成器函数，而不是平常的函数。它设置了一个标识来记住这个事实：

```
>>> # The generator flag is bit position 5.
>>> generator_bit = 1 << 5
>>> bool(gen_fn.__code__.co_flags & generator_bit)
True
```

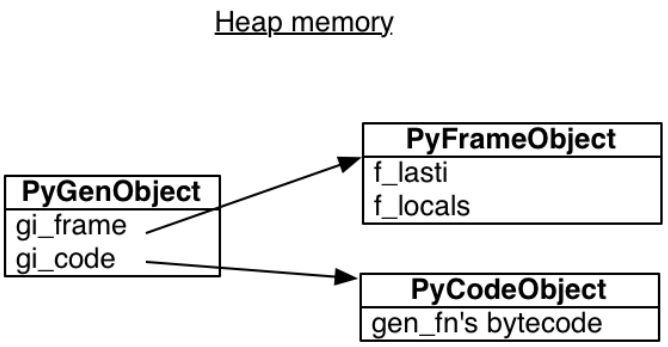
当你调用一个生成器函数，Python 看到了生成器标识，但是实际上它并不执行这个函数。代替的是创建一个生成器：

```
>>> gen = gen_fn()
>>> type(gen)
<class 'generator'>
```

一个 Python 生成器封装了一个栈贞附加一些代码的引用， gen_fn 的主体：

```
>>> gen.gi_code.co_name
'gen_fn'
```

所有调用 gen_fn 的生成器都指向这个相同的代码。但是它们每一个都有自己的栈贞。栈贞不在实际的栈上，它被分配在堆内存上等待被使用：



贞有一个“最后指令”的指针，这个指令是最后执行过的。开始的时候，最后指令的指针是 -1，意味着生成器还没有开始：

```
>>> gen.gi_frame.f_lasti
-1
```

当我们调用 send ，生成器到达第一个 yield ，并且暂停。 send 的返回值是 1，因为那是 gen 传给 yield 表达式的：

```
>>> gen.send(None)
1
```

现在生成器指令指针是开始时候的字节码 3，部分已经被编译过的 Python 代码是 56 字节：

```
>>> gen.gi_frame.f_lasti
3
>>> len(gen.gi_code.co_code)
56
```

生成器能在任何时候被任何函数恢复，因为它的栈贞实际上不是在栈上：在堆上。在调用层次结构中的位置不是固定的，并且不必遵循像正常函数那样先进后出的执行顺序。它被解放了，像云一样漂浮。

我们可以给生成器发送一个值“hello”然后他就成了 yield 表达式的结果，然后生成器继续直到它产生 2：

```
>>> gen.send('hello')
result of yield: hello
2
```

它的栈贞现在包含了局部变量 result：

```
>>> gen.gi_frame.f_locals
{'result': 'hello'}
```

其它从 gen_fn 创建的生成器会拥有他们自己的栈贞和它们自己的局部变量。

当我们再次调用 send 的时候，生成器从第二个 yield 继续，然后以抛出一个特殊异常 StopIteration 的方式结束：

```
>>> gen.send('goodbye')
result of 2nd yield: goodbye
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration: done
```

这个异常有一个值，是生成器的返回值：字符串“done”。

用生成器来打造协程

那么一个生成器可以暂停，也能被一个值恢复，还能返回一个值。这听起来像是一个建立异步编程模型的良好的开端，没有面条式的回调！我们想建立一个“协程”：一个协同程序在程序里和其它的协程合作调度。我们的协程将会是 Python 的 “asyncio” 标准库里的协程的简单版本。就像在 asyncio 里一样，我们将使用生成器、futures、和 `yield from` 语句。

首先，我们需要一个方法来表示协程正在等待的一些将来时的结果。一个简装版本：

```
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)
```

一个 future 最初是“pending”。通过调用“set_result”来“resolved”。

让我们用 futures 和协程来改写 fetcher。回顾我们用回调写的 `fetch`：

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

    def connected(self, key, mask):
        print('connected!')
        # And so on....
```

`fetch` 方法开始连接一个 socket，然后注册一个回调 `connected`，为了在 socket 准备好的时候执行。现在我们能组合这两步在一个协程里面：

```
    def fetch(self):
        sock = socket.socket()
        sock.setblocking(False)
        try:
            sock.connect(('xkcd.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())
        print('connected!')
```

现在 fetch 是一个生成器函数，而不是平常的那种，因为它拥有 yield 语句。我们创建了一个等待中的 future，然后 yield 它来暂停 fetch 知道 socket 准备好。内部函数 on_connected 解决 future。

但是当 future 被解决了，怎样恢复生成器？我们需要一个协程驱动。让我们把它叫做“task”：

```
class Task:
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)
    def step(self, future):
        try:
            next_future = self.coro.send(future.result)
        except StopIteration:
            return

        next_future.add_done_callback(self.step)

# Begin fetching http://xkcd.com/353/
fetcher = Fetcher('/353/')
Task(fetcher.fetch())

loop()
```

task 通过发送一个 None 来开始生成器。然后 fetch 运行直到它产生一个 future，task 捕获它并把它作为 next_future 。当 socket 连接之后，事件循环执行回调 on_connected ，解决 future 并调用 step ，重新开始 fetch 。

用 yield from 制造协程

一旦 socket 已经连接了，我们就发送 HTTP GET 请求并读取服务器的响应。这些步骤不再分散在很多回调里面了；我们把它们放在同一个生成器函数里面：

```
    def fetch(self):
        # ... connection logic from above, then:
        sock.send(request.encode('ascii'))

        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:
                # Done reading.
                break
```

这段代码从 socket 里面读取完整的消息，看起来很有用。我们怎样才能把它从 fetch 里面分解到一个子程序中？现在 Python 3 的 yield from 该隆重登场了。它可以把一个生成器传给另一个。

为了看如何做的，让我们返回到我们简单的生成器例子：

```
>>> def gen_fn():
...     result = yield 1
...     print('result of yield: {}'.format(result))
...     result2 = yield 2
...     print('result of 2nd yield: {}'.format(result2))
...     return 'done'
...
```

从另一个生成器调用这个生成器，用 yield from 委托给另一个：

```
>>> # Generator function:
>>> def caller_fn():
...     gen = gen_fn()
...     rv = yield from gen
...     print('return value of yield-from: {}'
...           .format(rv))
...
>>> # Make a generator from the
>>> # generator function.
>>> caller = caller_fn()
```

生成器 caller 的行为就像 gen 一样，生成器已经被委托了：

```
>>> caller.send(None)
1
>>> caller.gi_frame.f_lasti
15
>>> caller.send('hello')
result of yield: hello
2
>>> caller.gi_frame.f_lasti  # Hasn't advanced.
15
>>> caller.send('goodbye')
result of 2nd yield: goodbye
return value of yield-from: done
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

当 caller 从 gen 产生的时候， caller 还没有向前推进。注意指令指针保持在 15， yield from 语句的位置也没变，即使生成器 gen 从第一个 yield 语句进展到了下一个。从 caller 的外部来看，我们不知道值是从 caller 产生还是从委托的生成器产生。并且从 gen 里面来看，我们不知道值是从 caller 里面发送过来还是从 caller 外面发送过来。 yield from 语句是没有摩擦的通道，通过它流入并从 gen 流出直到 gen 完成。

一个协程能用 yield from 委托工作给子程序并接收工作的结果。主要，上面的 caller 打印了“return value of yield-from: done”。当 gen 完成后， gen 的返回值成为了 caller 里面 yield from 语句的值：

```
    rv = yield from gen
```

前面，我们批判基于回调的异步编程时，我们最不满的的是“堆栈撕裂”：当回调抛出异常的时候，堆栈追踪是没用的。只说明了事件循环正在执行一个回调，没有为什么。协程又怎么样呢？

```
>>> def gen_fn():
...     raise Exception('my error')
>>> caller = caller_fn()
>>> caller.send(None)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 3, in caller_fn
  File "<input>", line 2, in gen_fn
Exception: my error
```

这非常有用！发生错误时，堆栈追踪展示了 caller_fn 被委托给了 gen_fn 。更令人欣慰的是，我们能给调用子程序的地方包裹一个异常处理函数，跟正常的子程序一样：

```
>>> def gen_fn():
...     yield 1
...     raise Exception('uh oh')
...
>>> def caller_fn():
...     try:
...         yield from gen_fn()
...     except Exception as exc:
...         print('caught {}'.format(exc))
...
>>> caller = caller_fn()
>>> caller.send(None)
1
>>> caller.send('hello')
caught uh oh
```

我们用子协程的分解逻辑就像用正常子程序一样。让我们从 fetcher 分解出一些有用的子协程。我们编写了一个 read 协程去接收一块数据。

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

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

    selector.register(sock.fileno(), EVENT_READ, on_readable)
    chunk = yield f  # Read one chunk.
    selector.unregister(sock.fileno())
    return chunk
```

我们基于 read 用 read_all 协程来接收一个完整的消息：

```
def read_all(sock):
    response = []
    # Read whole response.
    chunk = yield from read(sock)
    while chunk:
        response.append(chunk)
        chunk = yield from read(sock)

    return b''.join(response)
```

如果你用正确的方式看， yield from 语句消失了并且就像传统上的函数处理阻塞 I/O 一样。但事实上， read 和 read_all 是协程。暂停 read_all 从 read 产出直到 I/O 完成。 read_all 暂停时，aysncio 的事件循环继续做其它的工作也等待其它的 I/O 事件；下一次事件循环时钟滴答到来的时候 read_all 被 read 的结果恢复，只要事件准备好了。

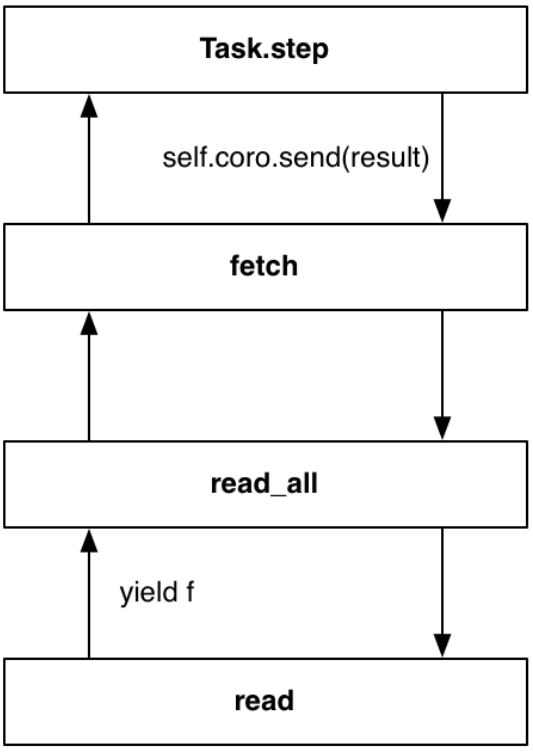
在栈的根部， fetch 调用 read_all ：

```
class Fetcher:
    def fetch(self):
         # ... connection logic from above, then:
        sock.send(request.encode('ascii'))
        self.response = yield from read_all(sock)
```

奇迹般的，我们的 Task 类不需要更改。就像上面的一样驱动 fetch 协程：

```
Task(fetcher.fetch())
loop()
```

当 read 产生一个 future 时，task 通过 yield from 语句的通道收到，就像 future 是由 fetch 直接产生的一样。当循环解决 future 时，task 将它的结果发送给 fetch ，值被 read 收到，就像 task 直接驱动 read 一样：



为了完善我们的协程实现，我们擦除一个污点：当等待一个 future 的时候我们用 yield ，当委托给一个子协程是使用 yield from 。如果无论什么时候协程暂停时我们都用 yield from 会更优雅。这样协程就不需要关心它等待的是什么类型的事件。

我们利用 Python 中迭代器和生成器深度一致的优势。提升一个协程为调用者，就像提升一个迭代器一样。因此我们通过实现一个特殊的方法使我们的 Future 类可迭代：

```
    # Method on Future class.
    def __iter__(self):
        # Tell Task to resume me here.
        yield self
        return self.result
```

future 的 __iter__ 方法是一个返回它自己的协程。现在我们把下面的代码替换：

```
# f is a Future.
yield f
```

…用下面这段代码：

```
# f is a Future.
yield from f
```

…结果是一致的！驱动程序 Task 从 `self.coro.send(result)` 接收 future，当 future 被解决的时候，它把新的结果送回协程。

任何地方都用 `yield from` 的优势是什么？为什么比用 `yield` 来等待 futures 和用 `yield from` 来委托给子协程要好？更好是因为现在，一个方法能自由的更改它的实现而不影响调用者：它可能是一个正常的方法，它返回一个解决 future 的值，或者它可能是一个包含 `yield from` 的协程并返回一个值。在任何情况下，调用者只需要 `yield from` 一个为了等待结果的方法。

亲爱的读者，我们尽情享受的关于 asyncio 中的协程阐述已经到达尾声了。我们窥探了协程的机制，并勾画了 future 和 task 的实现。我们概述了 asyncio 如何汲取两个世界的优势：并发 I/O 比线程更高效，比回调更清晰。当然，真正的 asyncio 比我们的示例更复杂。真实的框架地址零拷贝 I/O，合理调度，异常处理，和诸多其它特性。

写给 asyncio 用户，用协程编码比你这里看到的更简单。在上面的代码中我们从基本的原理开始实现协程，因此你看到了回调、tasks、futures。你甚至见到了非阻塞的 sockets 和 `select` 调用。但是当你真正用 asyncio 建立一个应用程序的时候，这些都不会出现在你的代码中。就像我们承若过的，你能如此顺畅的抓取一个 URL：

```
    @asyncio.coroutine
    def fetch(self, url):
        response = yield from self.session.get(url)
        body = yield from response.read()
```

已经满意了，我们回到我们最开始的任务：用 asyncio 写一个异步的 Web 爬虫。

协调协程

我们以描述我们想要我们的爬虫怎样工作作为开始。现在是时候用 asyncio 协程写一个爬虫了。

我们的爬虫要抓取第一个网页，解析里面的链接，把它们加入队列里。之后开始遍历网站，并发的抓取网页。但要限制客户端和服务端的负载，我们需要一个运行中 workers 的最大数量，而不是更多。无论什么时候一个 worker 完成对一个链接抓取任务，立即从队列中取出下一个链接。我们传入一个期限，当没有更多的任务做的时候，一些 workers 必须暂停。但是当一个 worker 找到一个富含新链接的网页后，队列迅速膨胀，那么暂停的 workers 应该马上醒来并开始工作。最后我们的程序必须退出一旦工作完成。

想象一下如果我们的 workers 是线程。我们应该怎样表达爬虫的算法？我们可以用来自 Python 标准库的同步队列。每次一个任务被放入队列中，队列增加任务的计数。worker 线程在任务完成后调用 `task_done`。主线程阻塞在 `Queue.join` 知道每个被放入队列中的任务匹配了一个 `task_done` 调用，然后程序退出。

协程和来自 asyncio 的队列使用相同的模式！首先我们导入 asyncio 的队列：

```
try:
    from asyncio import JoinableQueue as Queue
except ImportError:
    # In Python 3.5, asyncio.JoinableQueue is
    # merged into Queue.
    from asyncio import Queue
```

我们把爬虫的共享状态放在一个 crawler 类里面，把主逻辑写在 `crawl` 方法里。我们在协程里开始 `crawl` 并执行 asyncio 的事件循环知道 `crawl` 完成：

```
loop = asyncio.get_event_loop()

crawler = crawling.Crawler('http://xkcd.com',
                           max_redirect=10)

loop.run_until_complete(crawler.crawl())
```

crawler 以根 URL 和 `max_redirect` 开始， `max_redirect` 是抓取链接时跟随重定向的次数。把 (URL，`max_redirect`) 放入队列。（为什么这么做？敬请期待）

```
class Crawler:
    def __init__(self, root_url, max_redirect):
        self.max_tasks = 10
        self.max_redirect = max_redirect
        self.q = Queue()
        self.seen_urls = set()

        # aiohttp's ClientSession does connection pooling and
        # HTTP keep-alives for us.
        self.session = aiohttp.ClientSession(loop=loop)

        # Put (URL, max_redirect) in the queue.
        self.q.put((root_url, self.max_redirect))
```

现在队列里面未完成的任务数为 1，回到我们的主要脚本，我们启动事件循环和 crawl 方法：

```
loop.run_until_complete(crawler.crawl())
```

crawl 协程开始 workers 的比赛。就像一个主线程：阻塞在 join 直到所有的任务完成，当 workers 在后台运行的时候。

```
    @asyncio.coroutine
    def crawl(self):
        """Run the crawler until all work is done."""
        workers = [asyncio.Task(self.work())
                   for _ in range(self.max_tasks)]

        # When all work is done, exit.
        yield from self.q.join()
        for w in workers:
            w.cancel()
```

如果 workers 是线程我们可能不希望一次启动所有的任务。避免创建昂贵的线程直到迫不得已，线程池通常根据需求扩充。但是协程是廉价的，因此我们直接启动最大数量的协程是允许的。

注意我们关闭爬虫的方式是有趣的。当 join 解决 future 后，worker 是活着的但是挂起了：它们等待更多的 URL 但是没有。因此，主协程在退出前取消它们。否则，当 Python 解释器关闭时调用所有对象的析构函数，留下 tasks 哭泣：

```
ERROR:asyncio:Task was destroyed but it is pending!
```

cancel 是怎样工作的？生成器有一个我们还没有展示给你的特性。你能从生成器外面抛出一个异常进去：

```
>>> gen = gen_fn()
>>> gen.send(None)  # Start the generator as usual.
1
>>> gen.throw(Exception('error'))
Traceback (most recent call last):
  File "<input>", line 3, in <module>
  File "<input>", line 2, in gen_fn
Exception: error
```

生成器被 throw 恢复，但是现在抛出了一个异常。如果生成器内部调用栈没有代码去捕获异常，异常冒泡到顶部。这样取消一个 task 协程：

```
    # Method of Task class.
    def cancel(self):
        self.coro.throw(CancelledError)
```

无论生成器在哪里暂停，在一些 yield from 语句，他能重新开始协程并抛出一个异常。我们在 task 的 step 方法里面处理 CancelledError：

```
    # Method of Task class.
    def step(self, future):
        try:
            next_future = self.coro.send(future.result)
        except CancelledError:
            self.cancelled = True
            return
        except StopIteration:
            return

        next_future.add_done_callback(self.step)
```

现在 task 就知道它被取消了，因此当它被销毁了就不会再反对死亡之光了。。。

一旦 crawl 取消了 workers，就退出。在事件循环看来协程完成了（我们后面讲解），它也退出：

```
loop.run_until_complete(crawler.crawl())
```

crawl 方法由我们的主协程必须做的一切组成。里面的 worker 从队列里取得 URL，抓取 URL，然后解析里面的新链接。每个 worker 独立的执行一个 work 协程：

```
    @asyncio.coroutine
    def work(self):
        while True:
            url, max_redirect = yield from self.q.get()

            # Download page and add new links to self.q.
            yield from self.fetch(url, max_redirect)
            self.q.task_done()
```

Python 看到代码包含 yield from 语句，就把它编译成生成器函数。因此在 crawl 里，当主协程调用 work 十次，实际上不会执行这个方法：仅仅创建十个引用这段代码的生成器对象。它们每个都被包裹在一个 Task 里面。Task 接收生成器产生的每个 future，当 future 解决后，通过调用 send 方法发送 future 的结果来驱动生成器。因为生成器有它们自己的栈贞，所以它们独立的运行，局部变量和指令指针也是分开的。

worker 通过队列来协调。用下面的方式等待新 URL：

```
    url, max_redirect = yield from self.q.get()
```

队列的 get 方法也是一个协程：它暂停直到某个协程往队列里加入任务，然后重新开始并返回。

顺便说一句，worker 在 crawl 的结尾会被暂停，当主协程取消它时。以协程看来，当 yield from 抛出一个 CancelledError 时就是循环结束的最后一轮循环。

当 worker 抓取了一个网页，它解析网页并把新链接放入队列，然后调用 task_done 减少计数器。最终，worker 会发现抓取的页面里的链接全部被访问过了，并且队列里面也没有任务剩余。因此 worker 调用 task_done 减少计数器到零。然后， crawl 等待队列的 join 方法，被取消暂停然后结束。

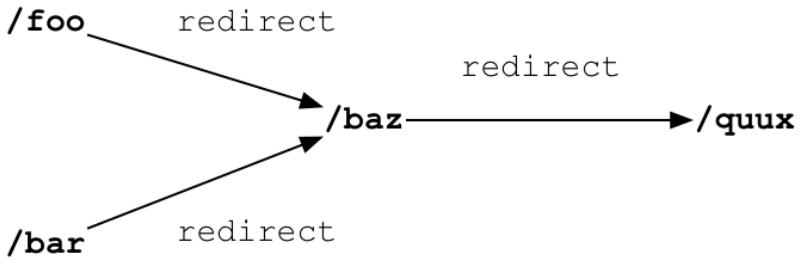
我们承若过要解释为什么队列的每一项都像下面所示的一组数据：

```
# URL to fetch, and the number of redirects left.
('http://xkcd.com/353', 10)
```

新的 URL 有十次重定向的机会。抓取这样的链接时会导致它被重定向到一个有尾随反斜杠的新地方。我们就减少重定向剩余次数，然后把下一个位置放置放入队列：

```
# URL with a trailing slash. Nine redirects left.
('http://xkcd.com/353/', 9)
```

我们使用的 aiohttp 默认会跟随重定向并给我们最终的响应。我们告诉它不要这样做，当然，爬虫里面是处理重定向的，因此这样可以把指向相同目的地的重定向路径合并：如果我们已经访问个那个 URL，它就会存在于 seen_urls ，并且我们已经从不同的入口开始了这条路径：



爬虫抓取“foo”发现它重定向到“baz”，因此把“baz”添加到待处理队列和 seen_urls 中。如果抓取的下一个是“bar”，它也重定向到“baz”，fetcher 不再把“baz”放入队列中。

```
    @asyncio.coroutine
    def fetch(self, url, max_redirect):
        # Handle redirects ourselves.
        response = yield from self.session.get(
            url, allow_redirects=False)

        try:
            if is_redirect(response):
                if max_redirect > 0:
                    next_url = response.headers['location']
                    if next_url in self.seen_urls:
                        # We have been down this path before.
                        return

                    # Remember we have seen this URL.
                    self.seen_urls.add(next_url)

                    # Follow the redirect. One less redirect remains.
                    self.q.put_nowait((next_url, max_redirect - 1))
                else:
                    links = yield from self.parse_links(response)
                    # Python set-logic:
                    for link in links.difference(self.seen_urls):
                        self.q.put_nowait((link, self.max_redirect))
                    self.seen_urls.update(links)
        finally:
            # Return connection to pool.
            yield from response.release()
```

如果响应是一个网页而不是一个重定向， fetch 解析里面的链接然后把新链接放入队列中。

如果这是多线程的代码，用竞态条件会很糟糕。举个例子，在最后几行里 worker 检查链接是否在 seen_urls 里面，如果不在里面， worker 将其加入队列并放入 seen_urls 。如果在两个操作之间被中断，然后另一个 worker 可能从另一个页面解析到同一个链接，它也发现链接不在 seen_urls 中，然后这个链接又被放入队列了。现在队列中出现同一个链接两次，这样会导致（最好）重复工作和错误的统计数字。

无论如何，协程只会在 yield from 语句处中断。这是协程代码比多线程代码更不容易出现竞争的关键区别：多线程代码必须明确的以获取锁的方式进入一个临界区，否则就是可中断的。Python 协程，无论如何在默认情况下都是不可中断的，只有在 yield 的时候交出控制权。

我们不再需要一个类似我们基于回调的程序里所拥有的 fetcher 类。这个类是为了弥补回调的不足：当等待 I/O 时它们需要一个地方存储状态，因为它们的局部变量不能保持到调用之外。但是 fetch 协程能像正常函数一样保存状态在局部变量里面，因此不再需要 fetcher 类

了。

当 fetch 完成了对服务器响应的处理，就返回到调用者 work 里面。 work 方法调用队列的 task_done 然后从队列里取得下一个要抓取的链接。

当 fetch 把新链接放入到队列里，就增加未完成任务的数量并保持正在等待 q.join 的主协程暂停。如果，如果没有未见过的链接并且队列中只剩最后一个 URL，当 work 调用 task_done 后减少未完成的任务数量为零。事件取消暂停然后主协程结束。

协调协程的队列代码和主协程长成这样：

```
class Queue:
    def __init__(self):
        self._join_future = Future()
        self._unfinished_tasks = 0
        # ... other initialization ...

    def put_nowait(self, item):
        self._unfinished_tasks += 1
        # ... store the item ...

    def task_done(self):
        self._unfinished_tasks -= 1
        if self._unfinished_tasks == 0:
            self._join_future.set_result(None)

    @asyncio.coroutine
    def join(self):
        if self._unfinished_tasks > 0:
            yield from self._join_future
```

主协程 crawl 从 join 产生。因此当最后一个 wroker 减少未完成的任务数为 0 时，它给 crawl 发重新开始的信号，然后结束。

旅程基本上结束了。我们开始调用 crawl ：

```
loop.run_until_complete(self.crawler.crawl())
```

程序是如何结束的？因为 crawl 是一个生成器函数，调用它返回一个生成器。为了驱动这个生成器，asyncio 用 Task 包裹它：

```
class EventLoop:
    def run_until_complete(self, coro):
        """Run until the coroutine is done."""
        task = Task(coro)
        task.add_done_callback(stop_callback)
        try:
            self.run_forever()
        except StopError:
            pass

class StopError(BaseException):
    """Raised to stop the event loop."""

def stop_callback(future):
    raise StopError
```

当任务完成，抛出 StopError，事件循环把它作为正常结束的信号。

但这是什么？task 有叫作 add_done_callback 和 result 的方法吗？你可能会猜 task 就是 future。你的直觉是正确的。我们必须承认我们隐藏了 task 的一个细节：task 就是 future：

```
class Task(Future):
    """A coroutine wrapped in a Future."""
```

正常情况下 future 在某人调用 set_result 的时候被解决。但是 task 当协程停止时解决自己。记得我们早前探究过生成器，当生成器返回的时候，它抛出一个特殊的异常 StopIteration：

```
    # Method of class Task.
    def step(self, future):
        try:
            next_future = self.coro.send(future.result)
        except CancelledError:
            self.cancelled = True
            return
        except StopIteration as exc:
            # Task resolves itself with coro's return
            # value.
            self.set_result(exc.value)
            return

        next_future.add_done_callback(self.step)
```

当事件循环调用 task.add_done_callback(stop_callback) 时，它就准备被 task 停止。这里又是 run_until_complete ：

```
    # Method of event loop.
    def run_until_complete(self, coro):
        task = Task(coro)
        task.add_done_callback(stop_callback)
        try:
            self.run_forever()
        except StopError:
            pass
```

当 task 捕获到 StopIteration 的时候就解决掉自己，从回调中抛出 StopError 。循环停止并且调用栈从 run_until_complete 解开。我们的程序结束。

结论

越来越多的现代程序是 I/O 密集型的而不是 CPU 密集型的。对于这些程序，Python 的线程在这两种世界下都是最糟的：全局解释器锁是它们不能进行真正的并行计算，而抢占式切换使它们有竞争倾向。

异步通常是正确的模式。但是基于回调的异步代码变多时，会变得混乱不堪。协程是一个整洁的替代。它们可以自然分解成子程序，有正确的异常处理和堆栈追踪。

如果我们忽略 yield from 语句，一个协程就像一个处理传统阻塞 I/O 的线程。我们甚至可以用多线程编程的模式来协调协程。不需要再造轮子。因此，相比于回调，协程是有多线程编码经验的程序员的福音。

但是当你睁开眼睛并集中注意力在 yield from 语句上，我们发现它们在交出控制权时设置标记并允许其它协程允许。不像线程，协程展示了我们的代码在哪里会被中断哪里不会。Glyph Lefkowitz 在他精彩的论文“Unyielding”里写道“Threads make local reasoning difficult, and local reasoning is perhaps the most important thing in software development.”。显然的，不管怎么样，“understand the behavior (and thereby, the correctness) of a routine by examining the routine itself rather than examining the entire system.”成为可能。

这一章被写在 Python 和异步历史的文艺复兴时期。基于生成器的协程，你刚学过它是如何设计的，已经在 2014年3月被发布在 Python 3.4 的“asyncio”模块中。在2015年9月，Python 3.5 被发布了，协程成为了语言本身的特性。原生的协程通过新语法“async

def”来定义，“yield from”也被代替，用新的关键字“await”来委托协程去等待 Future。

尽管改进了许多，但是核心思想留存了下来。Python 原生的协程只是语法与生成器不同而工作方式相似；确实，它们会在 Python 解释器里面共享一个实现。Task，Future 和事件循环会继续扮演在 asyncio 中的角色。

现在你已经知道了 asyncio 的协程是怎么工作的，你可能会在很大程度上忘记这些细节。机制被掩盖在一个短小精悍的接口后面。但是你掌握的基础使你能够在现代异步编程环境中编写出正确高效的代码。

---

1. Guido introduced the standard asyncio library, called “Tulip” then, at PyCon 2013. ↩

2. Even calls to send can block, if the recipient is slow to acknowledge outstanding messages and the system’s buffer of outgoing data is full. ↩

3. http://www.kegel.com/c10k.html ↩

4. Python’s global interpreter lock prohibits running Python code in parallel in one process anyway. Parallelizing CPU-bound algorithms in Python requires multiple processes, or writing the parallel portions of the code in C. But that is a topic for another day. ↩

5. Jesse listed indications and contraindications for using async in “What Is Async, How Does It Work, And When Should I Use It?”:. Mike Bayer compared the throughput of asyncio and multithreading for different workloads in “Asynchronous Python and Databases”：↩

6. For a complex solution to this problem, see http://www.tornadoweb.org/en/stable/stack_context.html ↩

7. The @asyncio.coroutine decorator is not magical. In fact, if it decorates a generator function and the PYTHONASYNCIODEBUG environment variable is not set, the decorator does practically nothing. It just sets an attribute, _is_coroutine, for the convenience of other parts of the framework. It is possible to use asyncio with bare generators not decorated with @asyncio.coroutine at all. ↩

8. Python 3.5’s built-in coroutines are described in PEP 492, “Coroutines with async and await syntax.” ↩

9. This future has many deficiencies. For example, once this future is resolved, a coroutine that yields it should resume immediately instead of pausing, but with our code it does not. See asyncio’s Future class for a complete implementation. ↩

10. In fact, this is exactly how “yield from” works in CPython. A function increments its instruction pointer before executing each statement. But after the outer generator executes “yield from”, it subtracts 1 from its instruction pointer to keep itself pinned at the “yield from” statement. Then it yields to its caller. The cycle repeats until the inner generator throws StopIteration, at which point the outer generator finally allows itself to advance to the next instruction. ↩

11. https://docs.python.org/3/library/queue.html ↩

12. https://docs.python.org/3/library/asyncio-sync.html ↩

13. The actual asyncio.Queue implementation uses an asyncio.Event in place of the Future shown here. The difference is an Event can be reset, whereas a Future cannot transition from resolved back to pending. ↩

14. https://glyph.twistedmatrix.com/2014/02/unyielding.html ↩

← PREVIOUS POST (/2015/09/19/HOW-TO-DESIGN-A-PERMISSION-SYSTEM/)

说点什么吧...

发布

 (/feed.xml)  (http://www.douban.com/people/LastD001/)

 (https://github.com/Damnever)