12.12 使用生成器代替线程¶

问题¶

你想使用生成器（协程）替代系统线程来实现并发。这个有时又被称为用户级线程或绿色线程。

解决方案¶

要使用生成器实现自己的并发，你首先要对生成器函数和 `yield` 语句有深刻理解。 `yield` 语句会让一个生成器挂起它的执行，这样就可以编写一个调度器， 将生成器当做某种“任务”并使用任务协作切换来替换它们的执行。 要演示这种思想，考虑下面两个使用简单的 `yield` 语句的生成器函数：

```
# Two simple generator functions
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1
    print('Blastoff!')

def countup(n):
    x = 0
    while x < n:
        print('Counting up', x)
        yield
        x += 1
```

这些函数在内部使用yield语句，下面是一个实现了简单任务调度器的代码：

```
from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        '''
        Admit a newly started task to the scheduler
        '''
        self._task_queue.append(task)

    def run(self):
        '''
        Run until there are no more tasks
        '''
        while self._task_queue:
            task = self._task_queue.popleft()
            try:
                # Run until the next yield statement
                next(task)
                self._task_queue.append(task)
            except StopIteration:
                # Generator is no longer executing
                pass

# Example use
sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

`TaskScheduler` 类在一个循环中运行生成器集合——每个都运行到碰到yield语句为止。 运行这个例子，输出如下：

```
T-minus 10
T-minus 5
```

```
Counting up 0
T-minus 9
T-minus 4
Counting up 1
T-minus 8
T-minus 3
Counting up 2
T-minus 7
T-minus 2
...
```

到此为止，我们实际上已经实现了一个“操作系统”的最小核心部分。 生成器函数就是任务，而yield语句是任务挂起的信号。 调度器循环检查任务列表直到没有任务要执行为止。

实际上，你可能想要使用生成器来实现简单的并发。 那么，在实现actor或网络服务器的时候你可以使用生成器来替代线程的使用。

下面的代码演示了使用生成器来实现一个不依赖线程的actor：

```
from collections import deque

class ActorScheduler:
    def __init__(self):
        self._actors = {}          # Mapping of names to actors
        self._msg_queue = deque()   # Message queue

    def new_actor(self, name, actor):
        '''
        Admit a newly started actor to the scheduler and give it a name
        '''
        self._msg_queue.append((actor,None))
        self._actors[name] = actor

    def send(self, name, msg):
        '''
        Send a message to a named actor
        '''
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor,msg))

    def run(self):
        '''
        Run as long as there are pending messages.
        '''
        while self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try:
                 actor.send(msg)
            except StopIteration:
                 pass

# Example use
if __name__ == '__main__':
    def printer():
        while True:
            msg = yield
            print('Got:', msg)

    def counter(sched):
        while True:
            # Receive the current count
            n = yield
            if n == 0:
                break
            # Send to the printer task
            sched.send('printer', n)
            # Send the next count to the counter task (recursive)
            sched.send('counter', n-1)
```

```
    sched = ActorScheduler()
    # Create the initial actors
    sched.new_actor('printer', printer())
    sched.new_actor('counter', counter(sched))

    # Send an initial message to the counter to initiate
    sched.send('counter', 10000)
    sched.run()
```

完全弄懂这段代码需要更深入的学习，但是关键点在于收集消息的队列。 本质上，调度器在有需要发送的消息时会一直运行着。 计数生成器会给自己发送消息并在一个递归循环中结束。

下面是一个更加高级的例子，演示了使用生成器来实现一个并发网络应用程序：

```
from collections import deque
from select import select

# This class represents a generic yield event in the scheduler
class YieldEvent:
    def handle_yield(self, sched, task):
        pass

    def handle_resume(self, sched, task):
        pass

# Task Scheduler
class Scheduler:
    def __init__(self):
        self._numtasks = 0       # Total num of tasks
        self._ready = deque()    # Tasks ready to run
        self._read_waiting = {}  # Tasks waiting to read
        self._write_waiting = {} # Tasks waiting to write

    # Poll for I/O events and restart waiting tasks
    def _iopoll(self):
        rset,wset,eset = select(self._read_waiting,
                                self._write_waiting,[])
        for r in rset:
            evt, task = self._read_waiting.pop(r)
            evt.handle_resume(self, task)
        for w in wset:
            evt, task = self._write_waiting.pop(w)
            evt.handle_resume(self, task)

    def new(self,task):
        '''
        Add a newly started task to the scheduler
        '''
        self._ready.append((task, None))
        self._numtasks += 1

    def add_ready(self, task, msg=None):
        '''
        Append an already started task to the ready queue.
        msg is what to send into the task when it resumes.
        '''
        self._ready.append((task, msg))

    # Add a task to the reading set
    def _read_wait(self, fileno, evt, task):
        self._read_waiting[fileno] = (evt, task)

    # Add a task to the write set
    def _write_wait(self, fileno, evt, task):
        self._write_waiting[fileno] = (evt, task)

    def run(self):
        '''
        Run the task scheduler until there are no tasks
        '''
```

```
        while self._numtasks:
             if not self._ready:
                  self._iopoll()
             task, msg = self._ready.popleft()
             try:
                 # Run the coroutine to the next yield
                 r = task.send(msg)
                 if isinstance(r, YieldEvent):
                     r.handle_yield(self, task)
                 else:
                     raise RuntimeError('unrecognized yield event')
             except StopIteration:
                 self._numtasks -= 1

# Example implementation of coroutine-based socket I/O
class ReadSocket(YieldEvent):
    def __init__(self, sock, nbytes):
        self.sock = sock
        self.nbytes = nbytes
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        data = self.sock.recv(self.nbytes)
        sched.add_ready(task, data)

class WriteSocket(YieldEvent):
    def __init__(self, sock, data):
        self.sock = sock
        self.data = data

    def handle_yield(self, sched, task):
        sched._write_wait(self.sock.fileno(), self, task)

    def handle_resume(self, sched, task):
        nsent = self.sock.send(self.data)
        sched.add_ready(task, nsent)

class AcceptSocket(YieldEvent):
    def __init__(self, sock):
        self.sock = sock

    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)

    def handle_resume(self, sched, task):
        r = self.sock.accept()
        sched.add_ready(task, r)

# Wrapper around a socket object for use with yield
class Socket(object):
    def __init__(self, sock):
        self._sock = sock

    def recv(self, maxbytes):
        return ReadSocket(self._sock, maxbytes)

    def send(self, data):
        return WriteSocket(self._sock, data)

    def accept(self):
        return AcceptSocket(self._sock)

    def __getattr__(self, name):
        return getattr(self._sock, name)

if __name__ == '__main__':
    from socket import socket, AF_INET, SOCK_STREAM
    import time

    # Example of a function involving generators.  This should
    # be called using line = yield from readline(sock)
```

```
def readline(sock):
    chars = []
    while True:
        c = yield sock.recv(1)
        if not c:
            break
        chars.append(c)
        if c == b'\n':
            break
    return b''.join(chars)

# Echo server using generators
class EchoServer:
    def __init__(self,addr,sched):
        self.sched = sched
        sched.new(self.server_loop(addr))

    def server_loop(self,addr):
        s = Socket(socket(AF_INET,SOCK_STREAM))

        s.bind(addr)
        s.listen(5)
        while True:
            c,a = yield s.accept()
            print('Got connection from ', a)
            self.sched.new(self.client_handler(Socket(c)))

    def client_handler(self,client):
        while True:
            line = yield from readline(client)
            if not line:
                break
            line = b'GOT:' + line
            while line:
                nsent = yield client.send(line)
                line = line[nsent:]
        client.close()
        print('Client closed')

sched = Scheduler()
EchoServer(('',16000),sched)
sched.run()
```

这段代码有点复杂。不过，它实现了一个小型的操作系统。 有一个就绪的任务队列，并且还有因I/O休眠的任务等待区域。 还有很多调度器负责在就绪队列和I/O等待区域之间移动任务。

讨论¶

在构建基于生成器的并发框架时，通常会使用更常见的yield形式：

```
def some_generator():
    ...
    result = yield data
    ...
```

使用这种形式的yield语句的函数通常被称为“协程”。 通过调度器，yield语句在一个循环中被处理，如下：

```
f = some_generator()

# Initial result. Is None to start since nothing has been computed
result = None
while True:
    try:
        data = f.send(result)
        result = ... do some calculation ...
    except StopIteration:
        break
```

这里的逻辑稍微有点复杂。不过，被传给 `send()` 的值定义了在yield语句醒来时的返回值。 因此，如果一个yield准备在

对之前yield数据的回应中返回结果时，会在下一次 `send()` 操作返回。 如果一个生成器函数刚开始运行，发送一个None值会让它排在第一个yield语句前面。

除了发送值外，还可以在一个生成器上面执行一个 `close()` 方法。 它会导致在执行yield语句时抛出一个 `GeneratorExit` 异常，从而终止执行。 如果进一步设计，一个生成器可以捕获这个异常并执行清理操作。 同样还可以使用生成器的 `throw()` 方法在yield语句执行时生成一个任意的执行指令。 一个任务调度器可利用它来在运行的生成器中处理错误。

最后一个例子中使用的 `yield from` 语句被用来实现协程，可以被其它生成器作为子程序或过程来调用。 本质上就是将控制权透明的传输给新的函数。 不像普通的生成器，一个使用 `yield from` 被调用的函数可以返回一个作为 `yield from` 语句结果的值。 关于 `yield from` 的更多信息可以在 PEP 380 中找到。

最后，如果使用生成器编程，要提醒你的是它还是有很多缺点的。 特别是，你得不到任何线程可以提供的好处。例如，如果你执行CPU依赖或I/O阻塞程序， 它会将整个任务挂起直到操作完成。为了解决这个问题， 你只能选择将操作委派给另外一个可以独立运行的线程或进程。 另外一个限制是大部分Python库并不能很好的兼容基于生成器的线程。 如果你选择这个方案，你会发现你需要自己改写很多标准库函数。 作为本节提到的协程和相关技术的一个基础背景，可以查看 PEP 342 和 “协程和并发的一门有趣课程”

PEP 3156 同样有一个关于使用协程的异步I/O模型。 特别的，你不可能自己去实现一个底层的协程调度器。 不过，关于协程的思想是很多流行库的基础， 包括 gevent, greenlet, Stackless Python 以及其他类似工程。