gevent教程

gevent，它是一个并发网络库。它的协程是基于greenlet的，并基于libev实现快速事件循环（Linux上是epoll，FreeBSD上是kqueue，Mac OS X上是select）。有了gevent，协程的使用将无比简单，你根本无须像greenlet一样显式的切换，每当一个协程阻塞时，程序将自动调度，gevent处理了所有的底层细节。让我们看个例子来感受下吧。

Python

```
import gevent

def test1():
print 12
gevent.sleep(0)
print 34

def test2():
print 56
gevent.sleep(0)
print 78

gevent.joinall([
gevent.spawn(test1),
gevent.spawn(test2),
])
```

解释下，”gevent.spawn()”方法会创建一个新的greenlet协程对象，并运行它。”gevent.joinall()”方法会等待所有传入的greenlet协程运行结束后再退出，这个方法可以接受一个”timeout”参数来设置超时时间，单位是秒。运行上面的程序，执行顺序如下：

1. 先进入协程test1，打印12
2. 遇到”gevent.sleep(0)”时，test1被阻塞，自动切换到协程test2，打印56
3. 之后test2被阻塞，这时test1阻塞已结束，自动切换回test1，打印34
4. 当test1运行完毕返回后，此时test2阻塞已结束，再自动切换回test2，打印78
5. 所有协程执行完毕，程序退出

所以，程序运行下来的输出就是：

```
12
56
34
78
```

注意，这里与上一篇greenlet中第一个例子运行的结果不一样，greenlet一个协程运行完后，必须显式切换，不然会返回其父协程。而在gevent中，一个协程运行完后，它会自动调度那些未完成的协程。

我们换一个更有意义的例子：

Python

```
import gevent
import socket

urls = ['www.baidu.com', 'www.gevent.org', 'www.python.org']
jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
gevent.joinall(jobs, timeout=5)

print [job.value for job in jobs]
```

我们通过协程分别获取三个网站的IP地址，由于打开远程地址会引起IO阻塞，所以gevent会自动调度不同的协程。另外，我们可以通过协程对象的”value”属性，来获取协程函数的返回值。

猴子补丁 Monkey patching

细心的朋友们在运行上面例子时会发现，其实程序运行的时间同不用协程是一样的，是三个网站打开时间的总和。可是理论上协程是非阻塞的，那运行时间应该等于最长的那个网站打开时间呀？其实这是因为Python标准库里的socket是阻塞式的，DNS解析无法并发，包括像urllib库也一样，所以这种情况下用协程完全没意义。那怎么办？

一种方法是使用gevent下的socket模块，我们可以通过”from gevent import socket”来导入。不过更常用的

方法是使用猴子布丁（Monkey patching）：

Python

```
from gevent import monkey; monkey.patch_socket()
import gevent
import socket

urls = ['www.baidu.com', 'www.gevent.org', 'www.python.org']
jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
gevent.joinall(jobs, timeout=5)

print [job.value for job in jobs]
```

上述代码的第一行就是对socket标准库打上猴子补丁，此后socket标准库中的类和方法都会被替换成非阻塞式的，所有其他的代码都不用修改，这样协程的效率就真正体现出来了。Python中其它标准库也存在阻塞的情况，gevent提供了”monkey.patch_all()”方法将所有标准库都替换。

Python

```
from gevent import monkey; monkey.patch_all()
```

使用猴子补丁褒贬不一，但是官网上还是建议使用”patch_all()”，而且在程序的第一行就执行。

获取协程状态

协程状态有已启动和已停止，分别可以用协程对象的”started”属性和”ready()”方法来判断。对于已停止的协程，可以用”successful()”方法来判断其是否成功运行且没抛异常。如果协程执行完有返回值，可以通过”value”属性来获取。另外，greenlet协程运行过程中发生的异常是不会被抛出到协程外的，因此需要用协程对象的”exception”属性来获取协程中的异常。下面的例子很好的演示了各种方法和属性的使用。

Python

```
#coding:utf8
import gevent

def win():
return 'You win!'

def fail():
raise Exception('You failed!')

winner = gevent.spawn(win)
loser = gevent.spawn(fail)

print winner.started # True
print loser.started # True

# 在Greenlet中发生的异常，不会被抛到Greenlet外面。
# 控制台会打出Stacktrace，但程序不会停止
try:
gevent.joinall([winner, loser])
except Exception as e:
# 这段永远不会被执行
print 'This will never be reached'

print winner.ready() # True
print loser.ready() # True

print winner.value # 'You win!'
print loser.value # None

print winner.successful() # True
print loser.successful() # False

# 这里可以通过raise loser.exception 或 loser.get()
# 来将协程中的异常抛出
print loser.exception
```

协程运行超时

之前我们讲过在”gevent.joinall()”方法中可以传入timeout参数来设置超时，我们也可以在全局范围内设置超时时间：

Python

```
import gevent
from gevent import Timeout

timeout = Timeout(2) # 2 seconds
timeout.start()

def wait():
gevent.sleep(10)

try:
gevent.spawn(wait).join()
except Timeout:
print('Could not complete')
```

上例中，我们将超时设为2秒，此后所有协程的运行，如果超过两秒就会抛出”Timeout”异常。我们也可以将超时设置在with语句内，这样该设置只在with语句块中有效：

Python

```
with Timeout(1):
gevent.sleep(10)
```

此外，我们可以指定超时所抛出的异常，来替换默认的”Timeout”异常。比如下例中超时就会抛出我们自定义的”TooLong”异常。

Python

```
class TooLong(Exception):
pass

with Timeout(1, TooLong):
gevent.sleep(10)
```

协程间通讯

greenlet协程间的异步通讯可以使用事件（Event）对象。该对象的”wait()”方法可以阻塞当前协程，而”set()”方法可以唤醒之前阻塞的协程。在下面的例子中，5个waiter协程都会等待事件evt，当setter协程在3秒后设置evt事件，所有的waiter协程即被唤醒。

Python

```
#coding:utf8
import gevent
from gevent.event import Event

evt = Event()

def setter():
print 'Wait for me'
gevent.sleep(3) # 3秒后唤醒所有在evt上等待的协程
print "Ok, I'm done"
evt.set() # 唤醒

def waiter():
print "I'll wait for you"
evt.wait() # 等待
print 'Finish waiting'

gevent.joinall([
gevent.spawn(setter),
gevent.spawn(waiter),
gevent.spawn(waiter),
gevent.spawn(waiter),
gevent.spawn(waiter),
gevent.spawn(waiter)
])
```

除了Event事件外，gevent还提供了AsyncResult事件，它可以在唤醒时传递消息。让我们将上例中的setter和waiter作如下改动:

Python

```
from gevent.event import AsyncResult
aevt = AsyncResult()

def setter():
print 'Wait for me'
gevent.sleep(3) # 3秒后唤醒所有在evt上等待的协程
print "Ok, I'm done"
aevt.set('Hello!') # 唤醒，并传递消息

def waiter():
print("I'll wait for you")
message = aevt.get() # 等待，并在唤醒时获取消息
print 'Got wake up message: %s' % message
```

队列 Queue

队列Queue的概念相信大家都知道，我们可以用它的put和get方法来存取队列中的元素。gevent的队列对象可以让greenlet协程之间安全的访问。运行下面的程序，你会看到3个消费者会分别消费队列中的产品，且消费过的产品不会被另一个消费者再取到：

Python

```
import gevent
from gevent.queue import Queue

products = Queue()

def consumer(name):
while not products.empty():
print '%s got product %s' % (name, products.get())
gevent.sleep(0)

print '%s Quit'

def producer():
for i in xrange(1, 10):
products.put(i)

gevent.joinall([
gevent.spawn(producer),
gevent.spawn(consumer, 'steve'),
gevent.spawn(consumer, 'john'),
gevent.spawn(consumer, 'nancy'),
])
```

put和get方法都是阻塞式的，它们都有非阻塞的版本：put_nowait和get_nowait。如果调用get方法时队列为空，则抛出”gevent.queue.Empty”异常。

信号量

信号量可以用来限制协程并发的个数。它有两个方法，acquire和release。顾名思义，acquire就是获取信号量，而release就是释放。当所有信号量都已被获取，那剩余的协程就只能等待任一协程释放信号量后才能得以运行：

Python

```
import gevent
from gevent.coros import BoundedSemaphore

sem = BoundedSemaphore(2)

def worker(n):
sem.acquire()
print('Worker %i acquired semaphore' % n)
gevent.sleep(0)
sem.release()
print('Worker %i released semaphore' % n)

gevent.joinall([gevent.spawn(worker, i) for i in xrange(0, 6)])
```

上面的例子中，我们初始化了”BoundedSemaphore”信号量，并将其个数定为2。所以同一个时间，只能有两个worker协程被调度。程序运行后的结果如下：

```
Worker 0 acquired semaphore
Worker 1 acquired semaphore
Worker 0 released semaphore
Worker 1 released semaphore
Worker 2 acquired semaphore
Worker 3 acquired semaphore
Worker 2 released semaphore
Worker 3 released semaphore
Worker 4 acquired semaphore
Worker 4 released semaphore
Worker 5 acquired semaphore
Worker 5 released semaphore
```

如果信号量个数为1，那就等同于同步锁。

协程本地变量

同线程类似，协程也有本地变量，也就是只在当前协程内可被访问的变量：

Python

```
import gevent
from gevent.local import local

data = local()

def f1():
data.x = 1
print data.x

def f2():
try:
print data.x
except AttributeError:
print 'x is not visible'

gevent.joinall([
gevent.spawn(f1),
gevent.spawn(f2)
])
```

通过将变量存放在local对象中，即可将其的作用域限制在当前协程内，当其他协程要访问该变量时，就会抛出异常。不同协程间可以有重名的本地变量，而且互相不影响。因为协程本地变量的实现，就是将其存放在以的”greenlet.getcurrent()”的返回为键值的私有的命名空间内。