4. 协程gevent

Greenlet是什么
在Gevent中用到的主要模式是greenlet，它是以C扩展模块的形式接入Python的轻量级协程．Greenlet全部运行在主程序进程的内部，它们被协作式的调度．与多进程不同，多进程使用的是操作系统调度的进程，实现的是真正的并行．在任意时刻只有一个协程在运行，所以协程执行的是并发操作．

创建Greenlets
gevent对greentlet初始化提供了一些封装，最常见的方法是使用gevent.spawn如下

示例代码：

In [ ]:
```python
import gevent

def foo():
    print("func foo start...") #1
    gevent.sleep(1)  #2
    print("func foo end..") #5

def bar():
    print("func bar start...") #3
    gevent.sleep(1) #4
    print('func bar end...') #6

if __name__ == '__main__':
    gevent.joinall([gevent.spawn(foo), gevent.spawn(bar)])
```

当我们在受限于网络或者IO的函数中使用gevent，这些函数会被协作式调度，gevent的真正能力会得到发挥．gevent处理了所有的细节，来保证网络库在合适的时候，隐式的交出greenlet上下文的执行权．

继续看下面的例子，通过协程实现并发，提升代码的执行效率．

示例代码：

In [ ]:
```python
import gevent
import random
import time

def task(i):
    sleep = random.randint(1, 2)
    gevent.sleep(sleep)
    print("Task %s sleep %ss" %(i, sleep))


def sync():
    for i in range(1, 5):
        task(i)
beg = time.time()
sync()
end = time.time()
print("sync cost time %s" %(end-beg))

def async():
    result = []
    for i in range(1, 5):
        result.append(gevent.spawn(task, i))
    gevent.joinall(result)

beg = time.time()
async()
end = time.time()
print("async cost time %s" %(end-beg))
```

```
Task 1 sleep 2s
Task 2 sleep 2s
Task 3 sleep 2s
Task 4 sleep 2s
sync cost time 8.01527881622

Task 1 sleep 1s
Task 2 sleep 2s
Task 3 sleep 2s
Task 4 sleep 2s
async cost time 2.00166201591
```

同步任务执行所消耗的时间为每个任务消耗的时间之和，异步任务消耗的时间等于耗时最大的那个任务所消耗的时间．除了使用基本的greenlet类之外，也可以创建一个greenlet类的子类，重载其_run()方法．

示例代码：

In [ ]:
```python
import gevent
import time

class MyGreenlet(gevent.Greenlet):
    def __init__(self, message, n):
        gevent.Greenlet.__init__(self)
        self.message = message
        self.n = n

    def _run(self):
        print(self.message)
        gevent.sleep(self.n)

if __name__ == '__main__':
    x = MyGreenlet("hello", 2)
    y = MyGreenlet("haah", 1)
    z = MyGreenlet("haahzz", 1)

    beg = time.time()
    l = [x, y, z]
    for j in l:
        j.start()
    for j in l:
        j.join()
    end = time.time()
    print("cost time %s" %(end-beg))
```

Greenlet状态

```
started  --> bool 指示此greenlet是否已经启动
ready()  --> bool 指示此greenlet是否已经停止
successful() --> bool 此greenlet是否已经停止而且没抛出异常
value      Greenlet代码返回的值
exception  此Greenlet内抛出的未捕获异常
```

示例代码：

In [ ]:
```
import gevent

def win():
    return 'you win!'

def fail():
    raise Exception('you fail at failing')

winner = gevent.spawn(win)
loser = gevent.spawn(fail)
print(winner.started)  #true
print(loser.started)  #true

try:
    gevent.joinall([winner, loser])
except:
    print('This will never be reached')


print(winner.value)  #you win
print(loser.value)   #None

print(winner.successful())  #true
print(loser.successful())   #false

print(loser.exception)  #you faild at failing
```

猴子补丁(Monkey patching)

通过Monkey patching将标准库当中的阻塞模块编程非阻塞如(socket ssl threading select)等.

示例代码:

In [ ]:
```
import socket
print(socket.socket)
from gevent import monkey
monkey.patch_socket()
print('after monkey patch')
print(socket.socket)
```

事件(event):
在Greenlet之间异步通信的一种形式

示例代码:

In [ ]:
```
import gevent
from gevent.event import Event

evt = Event()

def setter():
    print("Cost 2 seconds")
    gevent.sleep(2)
    print("sleep over")
    evt.set()

def waiter():
    print('i will wait')
    evt.wait()
    print("wait start..")


def main():
    gevent.joinall([
        gevent.spawn(setter),
        gevent.spawn(waiter),
        gevent.spawn(waiter),
        gevent.spawn(waiter),
        gevent.spawn(waiter),
        gevent.spawn(waiter),
    ])

main()
```

```
Cost 2 seconds
i will wait
i will wait
i will wait
i will wait
i will wait
sleep over
wait start..
wait start..
wait start..
wait start..
wait start..
```

可以通过事件对象从一个协程向另外一个协程传值, 示例代码:

In [ ]:
```
import gevent
from gevent.event import AsyncResult

a = AsyncResult()

def setter():
    print('setter fun  cost 3s')
    gevent.sleep(3)
    a.set("hello")


def waiter():
    print("waiter start...")
    print(a.get())


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

```
setter fun  cost 3s
waiter start...
waiter start...
hello
hello
```

队列
队列是一个排序的数据集合, 常见有put/get操作, 协程队列可以在greenlet之间安全的进行操作, 如果一个greentlet从队列中取出一项, 此项就不会被同时执行的其他Greenlet再取到了.

示例代码:

In [ ]:
```
import gevent
from gevent.queue   import Queue

q = Queue()
def worker(n):
    while not q.empty():
        t = q.get()
        print("worker %s got task %s"%(n, t))
        gevent.sleep(0.1)

        def boss():
    for j in range(1, 20):
        q.put_nowait(j)

gevent.spawn(boss).join()

bob = gevent.spawn(worker, "bob")
lily = gevent.spawn(worker, "lily")
lucy = gevent.spawn(worker, "lucy")

gevent.joinall([bob, lily, lucy])
```

如果需要，队列也可以阻塞在put或get操作上，put和get都有非阻塞的版本，put_nowait和get_nowait不会阻塞，在操作不能完成时抛出gevent.queue.Empty或gevent.queue.full异常.

下定义一个长度为3的queue， 让boss与worker同时运行，指定序列长度意味着直到queue有空余空间否则put被阻塞，反之如果队列中没有元素，get操作会被阻塞. 同时带有一个timeout参数，允许在超时时间内没有拿到元素的话则报 gevent.queue.Empty异常.

示例代码:

In [ ]:
```
import gevent
from gevent.queue import Queue

q = Queue(maxsize=3)

def boss():
    for j in range(1, 10):
        q.put(j)
    print("first task share finished!")

    for j in range(10, 20):
        q.put(j)
    print("second task share finished!")

def worker(name):
    try:
        while not q.empty():
            val = q.get(timeout=1)
            print("worker %s got task %s" %(name, val))
            gevent.sleep(0.1)
    except:
        print("Queue is timeout!")

boss = gevent.spawn(boss)
lily = gevent.spawn(worker, 'lily')
lucy = gevent.spawn(worker, 'lucy')
gevent.joinall([boss, lily, lucy])
```

组和池

group是一个运行中的greenlet的集合，集合中的Greenlet像一个组一样会被共同管理和调度.

示例代码:

In [ ]:
```
import gevent
from gevent.pool import Group

def talk(msg):
    for i in xrange(3):
        print(msg)

g1 = gevent.spawn(talk, 'bar')
g2 = gevent.spawn(talk, 'foo')
g3 = gevent.spawn(talk, 'hello')

group1 = Group()
group1.add(g1)
group1.add(g2)
group1.join()

group2 = Group()
group2.add(g3)
group2.join()
```

以上例子spawn了好几个talk，然后加入不同的组，group.join()等待所有的spawn完成，每完成一个就会从group里面删掉. 由于函数没返回值，以上例子看起来比较简单，我们看一个稍微复杂点的例子.

In [ ]:
```
import gevent
from gevent.pool import Group

group = Group()

def hello_from(n):
    print('Size of group %s' % len(group))
    print('Hello from Greenlet %s' % id(gevent.getcurrent()))
    return n + 10

x = group.map(hello_from, range(3))
print(type(x))
print x
```

使用group.map()这个函数来取得各spawn的返回值. map()是由第二个参数控制迭代次数，并且将其中的每一个元素传递给前面的函数. 以这个函数为例，这里会返回一个list构成这个list的对象就是迭代的参数传入函数后的返回值，结果为[10, 11, 12]. 如果将上面的map换成imap，则返回值变成一个迭代器. imap中有参数maxsize，指定同时执行的任务数，代码如下:

In [ ]:
```
import gevent
from gevent.pool import Group

group = Group()

def hello_from(n):
    gevent.sleep(2)
    print('Size of group %s' % len(group))
    print('Hello from Greenlet %s' % id(gevent.getcurrent()))
    return n + 10

x = group.imap(hello_from, range(20), maxsize=3)

for j in x:
    print j
```

代码运行时，并行任务数量为3个，也就是每2秒执行3个

group().imap_unordered()，先返回的greenlet先回来．比较一下两串代码的输出，示例代码：

In [ ]:
```
import gevent
from gevent.pool import Group

group = Group()

def hello_from(n):
    gevent.sleep(3-n)
    return 'task', n

x = group.imap(hello_from, range(3), )
for j in x:
    print j
```

```
('task', 0)
('task', 1)
('task', 2)
```

In [ ]:
```
import gevent
from gevent.pool import Group
group = Group()

def hello_from(n):
    gevent.sleep(3-n)
    return 'task', n

x = group.imap_unordered(hello_from, range(3), )
for j in x:
    print j
```

```
('task', 2)
('task', 1)
('task', 0)
```

池(Pool)是一个为处理数量变化并且需要限制并发的Greenlet数量而设计的结构，在需要并行的做很多受限于网络和IO的任务时常常需要用到它．

示例代码：

In [ ]:
```
import gevent
from gevent.pool import Pool

pool = Pool(2)
def hello_from(n):
    print('Size of len pool %s' %len(pool))

pool.map(hello_from, range(3))
```

锁和信号量

信号量可以用来限制并发访问互斥的资源．信号量有两个方法，acquire和release．

In [ ]:
```
import gevent
from gevent.pool import Pool
try:
    from gevent.coros import RLock, Semaphore
except:
    from gevent.lock import RLock, Semaphore

sem = Semaphore(2)
def worker1(n):
    sem.acquire()
    print('Worker %i acquired semaphore' % n)
    gevent.sleep(1)
    sem.release()
    print('Worker %i released semaphore' % n)

pool = Pool()
pool.map(worker1, xrange(0,3))
```

```
Worker 0 acquired semaphore
Worker 1 acquired semaphore
Worker 0 released semaphore
Worker 1 released semaphore
Worker 2 acquired semaphore
Worker 2 released semaphore
```

线程局部变量

gevent允许定义局部与greenlet上下文的数据，实现方式为以greenlet的getcurrent()为键，在一个私有命名空间寻址的全局查找．

示例代码：

In [ ]:
```
import gevent
from gevent.local import local

x = local()
def f1():
    x.x = 1
    print(x.x)
    print(x.__dict__)

def f2():
    x.y=2
    print(x.y)
    try:
        print x.__dict__
    except AttributeError:
        print("x is not local to f2")

g1 = gevent.spawn(f1)
g2 = gevent.spawn(f2)
gevent.joinall([g1, g2])
```

这里先初始化了一个线程本地对象local．然后它会把保存给她的属性当做线程本地变量存储起来．当其他的Greenlet去访问它的时候是无法访问到的，它只在自己的命名空间中有效．

子进程

可以通过gevent实现非阻塞的系统调用，可以通过gevent.subprocess来实现．示例代码如下：

In [ ]:
```
from gevent.pool import Pool
from gevent.subprocess import  PIPE, Popen

p = Pool()
def worker(n):
    print('start... task %s' %n)
    sub = Popen(["sleep 2; uname -r"], stdout=PIPE, shell=True)
    output, err = sub.communicate()
    print('end..')
    return output.strip()

x = p.map(worker, range(10))
for j in x:
    print j
```

进程+协程实现生产者与消费者

示例代码:

In [ ]:
```
from multiprocessing import Process, cpu_count, Queue, JoinableQueue
import gevent
import datetime


class Consumer(object):
    def __init__(self, q, no_tasks, name):
        self._no_tasks = no_tasks
        self._queue = q
        self.name = name
        self._rungevent(self._no_tasks)

    def _rungevent(self,  no_tasks):
        jobs = [gevent.spawn(self._printq) for _ in xrange(no_tasks)]
        gevent.joinall(jobs)

    def _printq(self):
        while 1:
            value = self._queue.get()
            if value is None:
                self._queue.task_done()
                break
            else:
                print("{0} time: {1}, value: {2}".format(self.name, datetime.datetime.now(), value))
        return


class Producer(object):
    def __init__(self, q, no_tasks, name, consumers_tasks):
        print(name)
        self._q = q
        self._no_tasks = no_tasks
        self.name = name
        self.consumer_tasks = consumers_tasks
        self._rungevent()

    def _rungevent(self):
        jobs = [gevent.spawn(self.produce) for _ in xrange(self._no_tasks)]
        gevent.joinall(jobs)
        for x in xrange(self.consumer_tasks):
            self._q.put_nowait(None)
        self._q.close()

    def produce(self):
        print("I am producer %s" %self.name)
        for no in xrange(10000):
            print no
            self._q.put(no, block=False)
        return


def main():
    total_cores = cpu_count()
    total_processes = total_cores * 2
    q = JoinableQueue()
    print("Gevent on top multiprocessing with 17 gevent coroutines 10 producers gevent and 7 consumers gevent")
    producer_gevents = 10
    consumer_gevents = 7
    jobs = []
    start = datetime.datetime.now()

    print total_cores, total_processes, producer_gevents, consumer_gevents, jobs, start


    for x in xrange(total_cores):
        if not x % 2:
            p = Process(target=Producer, args=(q, producer_gevents, "producer %d" %x, consumer_gevents))
            p.start()
            jobs.append(p)
        else:
            p = Process(target=Consumer, args=(q, consumer_gevents, "consumer %d" %x))
            p.start()
            jobs.append(p)

    for job in jobs:
        job.join()

    print("{0} process with {1} producer gevents and {2} consumer gevents took{3}\
           seconds to produce {4} numbers and consume".format(total_processes,
                                                              producer_gevents * total_cores,
                                                              consumer_gevents * total_cores,
                                                              datetime.datetime.now() - start,
                                                              producer_gevents * total_cores * 10000))


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

Actors
actor模型是一个并发模型, 简单的说它的主要思想就是许多个独立的Actor, 每个Actor有一个可以从其它Actor接收消息的收件箱; Actor内部的主循环遍历它收到的消息, 并根据它期望的行为来采取行动.

示例代码:

In [ ]:
```
import gevent
from gevent.queue import Queue

class Actor(gevent.Greenlet):

    def __init__(self):
        self.inbox = Queue()
        Greenlet.__init__(self)

    def receive(self, message):
        """
        Define in your subclass.
        """
        raise NotImplemented()

    def _run(self):
        self.running = True

        while self.running:
            message = self.inbox.get()
            self.receive(message)

import gevent
from gevent.queue import Queue
from gevent import Greenlet

class Pinger(Actor):
    def receive(self, message):
        print(message)
        pong.inbox.put('ping')
        gevent.sleep(0)

class Ponger(Actor):
    def receive(self, message):
        print(message)
        ping.inbox.put('pong')
        gevent.sleep(0)

ping = Pinger()
pong = Ponger()

ping.start()
pong.start()

ping.inbox.put('start')
gevent.joinall([ping, pong])
```