Openstack Eventlet分析(1)

本来打算总结一下eventlet在OpenStack中的应用，正巧在网上找到几篇别人已经总结好的资料，而且总结的很好，这里直接转载过来。同时也向作者表示感谢。

Eventlet库在OpenStack服务中上镜率很高，尤其是在服务的多线程和WSGI Server并发处理请求的情况下，深入了解eventlet库是很必要的。Eventlet库是由second life开源的高性能网络库，从Eventlet的源码可以知道，其主要依赖于两个关键的库：

1. greenlet

2. select.epoll（或者epoll等类似的库）

greenlet库过程了其并发的基础,eventlet库简单的对其封装之后，就构成了GreenTread。

select库中的epoll则是其默认的网络通信模型。正由于这两个库的相对独立性，可以从两个方面来学习eventlet库，首先是greenlet。

greenlet

1. greenlet官方文档

2. greenlet官方文档翻译

3. greentlet原理详细介绍

还补充一篇文档，写的很好。

openstack nova基础知识之eventlet

通过这三篇循序渐渐的文章，大概可以了解到greenlet是一个称为协程(coroutine)的东西，有下面几个特点。

1. 每个协程都有自己的私有stack及局部变量；

2. 同一时间内只有一个协程在运行，故无须对某些共享变量加锁；

3. 协程之间的执行顺序，完成由程序来控制；

总之，协程就是运行在一个线程内的伪并发方式，最终只有一个协程在运行，然后程序来控制执行的顺序。可以看下面的例子来理解上面的意思。

```
import greenlet

def test1(n):
    print "test1:",n
    gr2.switch(32)
    print "test1: over"

def test2(n):
    print "test2:",n
    print "test2: over"

current = greenlet.getcurrent()
gr1 = greenlet(test1,current)
gr2 = greenlet(test2,current)
gr1.switch(2)
```

这段程序的执行结果如下：

```
test1: 2
test2: 32
test1: over
```

整个程序的过程很直白，首先创建两个协程，创建的过程传入了要执行的函数和父greenlet（在前面给出的三个链接中有详细介绍），然后调用其中的一个协程的switch函数，并且传递参数进去，就开始执行test1，然后到了gr2.switch(32)语句，切换到test2函数来，最后又切换回去。最终test1运行结束，回到父greenlet中，执行结束。这个过程就是始终只有一个协程在运行，函数的执行流由程序自己来控制。这个过程在上面的链接中描述的更加具体。

GreenThread

那么在eventlet中对greenlet进行了简单的封装，就成了GreenThread，并且上面的程序还会引来一个问题，如果我们想要写一个

协程，那到底该如何来控制函数的执行过程了，如果协程多了，控制岂不是很复杂了。带着这个问题来看eventlet的实现。

在介绍下面的内容之前，先贴出eventlet官方的文档，这个上面详细的介绍了该如何来使用eventlet库。我们从其中选出一个接口来分析。spawn函数，调用该函数，将会使用一个GreenThread来执行用户传入的函数。函数具体接口如下：

```
def spawn(func, *args, **kwargs):
```

参数很清晰，想要执行的函数以及函数的参数。该函数实际上只做了三件事，最后返回创建的greenthread，因此该函数相比于spawn_n可以，得到函数调用的结果。

```
1. hub = hubs.get_hub()
2. g = GreenThread(hub.greenlet)
3. hub.schedule_call_global(0,g.switch,func,args,kwargs)
4. return g
```

第一，我们要先知道hubs的作用，在eventlet的官方文档有介绍，在greenlet的官方文档开始就是我们可以自己构造greenlet的调度器，那么hub的第一个作用就是greenthread的调度器。另外一个作用于网络相关，所以hub有多个实现，对应于epoll，select，poll，pyevent等，我们先看前面的第一个作用。

hub在eventlet中是一个单太实例，也也就是全局就这有这一个实例，其包含一个greenlet实例，该greenlet实例是self.greenlet = greenlet(self.run)，这个实例就是官方文档说的MAINLOOP，主循环，更加具体就是其中的run方法，是一个主循环。并且该hub还有两个重要的列表变量，self.timers 和 self.next_timers，前者是一个列表，但是在这个列表上实现了一个最小堆，用来存储将被调度运行的greenthread，后者，用来存储新加入的greenthread。

第二，创建一个GreenThread的实例，greenthread继承于greenlet，简单封装了下，该类的构造函数只需要一个参数，父

greenlet，然后再自己的构造函数中，调用父类greenlet的构造函数，传递两个参数，GreenTread的main函数和一个greenlet的实例。第二代码就知道，hubs中作为MAINLOOP的greenlet是所有先创建的greenthread的父greenlet。由前面介绍greenlet的例子中，我们可以知道，当调用该greenthread的switch方法时，将会开始执行该才传递给父类的self.main函数。

第三，然后单态的hub调用schedule_call_global函数，该函数的作用可以看其注释，用来调度函数去执行。

```
"""
Schedule a callable to be called after 'seconds' seconds have
elapsed. The timer will NOT be canceled if the current greenlet has
exited before the timer fires.
seconds: The number of seconds to wait.
cb: The callable to call after the given time.
*args: Arguments to pass to the callable when called.
**kw: Keyword arguments to pass to the callable when called.
"""
t = timer.Timer(seconds, cb, *args, **kw)
self.add_timer(t)
return t
```

注释中提到的timer是指，传递进来的参数会构造成Timer的实例最后添加到self.next_timer列表中。注意在spawn中传递进来的g.switch函数，如果调用了这个g.switch函数，则触发了它所在的greenthread的运行。

这三步结束之后，对spawn的调用就返回了，然而现在只是创建了一个GreenThread，还没有调度它去执行，最后还需要再返回的结果上调用g.wait()方法，这样就开始GreenThread的神奇之旅了。

我们看GreenThread的wait方法的具体代码：

```
def __init__(self, parent):
    greenlet.greenlet.__init__(self, self.main, parent)
```

```
    self._exit_event = event.Event()
    self._resolving_links = False

def wait(self):
    """ Returns the result of the main function of this GreenThread.  If the
    result is a normal return value, :meth:`wait` returns it.  If it raised
    an exception, :meth:`wait` will raise the same exception (though the
    stack trace will unavoidably contain some frames from within the
    greenthread module)."""
    return self._exit_event.wait()
```

wait方法调用了Event实例的wait方法，就是在这个wait函数中，调用了我们前面提到的单态实例hub的switch方法，然后该switch真正的去调用hub的self.greenlet.switch()，我们已经所过该greenlet是所有调用spwan创建的greenlet的父greenlet，该self.greenlet在初始时传递了一个self.run方法，就是所谓的MAINLOOP。最终，程序的运行会由于switch的调用，开始run方法中的while循环了，这是多线程开发者最熟悉的while循环了。

在该while循环中，就对self.next_timers中的timers做处理：

```
def prepare_timers(self):
    heappush = heapq.heappush
    t = self.timers
    for item in self.next_timers:
        if item[1].called:
            self.timers_canceled -= 1
        else:
            heappush(t, item)
    del self.next_timers[:]
```

首先处理next_timers中没有被调用的timers，push到最小堆中去，也就是时间最小者排前面，越先被执行。然后将所有已经调用了的timer删除掉，这是不是会有一个疑问：如果删除了的timers没有运行

结束，那么下次岂不是没有机会再被调度来运行了。再看了greenthread.py中的sleep函数之后，就会明白。

加入到heap中的timers这会按照顺序开始依次遍历，如果到了他们的执行时间点了，timer对象就会直接被调用。看下面的代码：

```
1.  t = self.timers
2.  heappop = heapq.heappop
3.  while t:
4.      exp = next[0]
5.      timer = next[1]
6.      if when < exp:
7.          break
8.      heappop(t)
9.      try:
10.         if timer.called:
11.             self.timers_canceled -= 1
12.         else:
13.     except self.SYSTEM_EXCEPTIONS:
14.         raise
15.     except:
16.         self.squelch_timer_exception(timer, sys.exc_info())
17.         clear_sys_exc_info()
```

我们知道，如果我们自定义的函数要运行时间很长，怎么办，其他的greenthread则没有机会去运行了，在openstack nova官方文档中介绍thread中也提到这个问题，此时我们需要在自己定义的函数中调用greenthread.sleep(0)函数，来进行切换，使其他的greenthread也能被调度运行。看看greenthread.sleep函数的代码。

```
1. def sleep(seconds=0):
2.  """Yield control to another eligible coroutine until at least *seconds*
have
3.  elapsed.
4.  *seconds* may be specified as an integer, or a float if fractional seconds
```

```
    are desired. Calling :func:`~greenthread.sleep` with *seconds* of 0 is the
    canonical way of expressing a cooperative yield. For example, if one is
    looping over a large list performing an expensive calculation without
    calling any socket methods, it's a good idea to call ``sleep(0)``
    occasionally; otherwise nothing else will run.
    """
    hub = hubs.get_hub()
    current = getcurrent() # 当前正在执行的greenthread，调用这个sleep函数
    assert hub.greenlet is not current, 'do not call blocking functions from
the mainloop'
    try:
        hub.switch()
    finally:
        timer.cancel()
```

从该sleep函数可以知道，我们又重新调用了一遍hub.schedule_call_global函数，然后直接调用hub.switch，这样在运行的子greenlet中，开始触发父greenlet（也就是MAINLOOP的greenlet）的执行，上次该greenlet正运行到 fire_timers 的timer()函数处，此时父greenlet则接着运行，开始新的调度。

至此，调度的过程就大致描述结束了。

greenthread中其他的函数都基本同样，如果我们的函数只是简单的进行CPU运行，而不涉及到IO处理，上面的知识就可以理解eventlet了，然而，eventlet是一个高性能的网络库，还有很大一部分是很网络相关的。在留给下次。

Openstack Eventlet分析(2)

上一篇博客OpenStack-Eventlet分析(1)以eventlet库中的spawn函数为代表，详细的介绍了spawn函数的运行过程。最终的重点是在hubs用来进行调度的一面，而hubs关于网络的一面还没有涉及。然而，上一

篇只专注于每一行代码的执行流程了，而没有eventlet用来调度greenthread的框架分析，这样容易导致阅读源码会出现的一个常见问题，只见树木不见树林。所以在这一篇，再详细分析eventlet用来调度greenthread的框架问题，下次再将其网络部分补齐，构成一个整体。

首先来看一个例子。

```
1. from eventlet import hubs
2. from eventlet import greenthread
3. 
4. def tellme(secret):
5.     print "a secret:",secret
6. 
7. 
8. hub = hubs.get_hub()
9. hub.schedule_call_global(0,tellme,"you are so beautiful")
10. hub.switch()
11. #greenthread.sleep(0)
```

在这个例子中，没有使用提供的spawn函数，而是直接使用hub来调度来运行我们定义的tellme函数，结果很显然，打印完a secrete: you are so beautiful 之后，并没有结束。我们在上一篇文章中提到，hub是单态的，存在一个greenlet，作为MAINLOOP，使用hub的switch函数来开始这个MAINLOOP的运行，也就是说，MAINLOOP的循环运行，需要触发。在MAINLOOP中完成调度，执行tellme然后就返回到MAINLOOP中继续运行了。

当我们使用greenthread.sleep(0)来代替上面的hub.switch(),程序就能正常结束了。sleep函数将自己所在的greenlet的switch函数加入到hub的调度列表中，然后调用switch来触发MAINLOOP的调度。我们知道如果一个greenthread运行结束了，那么就会回到父greenlet来，正是因为如此，sleep函数中向hub添加的current.switch函数运行之后，就结束了sleep函数的整个内容，返回到父greenlet来，父greenlet正式我们自己写的这片代码。

从上面的例子和spawn的例子对比，都是使用hub来调度一个函数的运行，差别在于，上面的例子，是调度一个普通函数运行，spawn在于调度一个greenlet的switch函数运行。这就引入了一个基本问题，hub调度的是什么？

Timer

对于任何传入到hub的函数，首先就会封装成Timer，代表了该函数将会在多久之后被执行。实际上，我们知道了，hub调度的是一个个Timer，不管这个Timer中存储的是什么函数，普通的函数还是greenlet的switch函数，都是一样的被处理。对于普通函数，我们可以让等待一定时间运行，我们关注的函数hub如何来调度greenthread。这才是重点。

event

再来看一个例子。

```
1. from eventlet import event
2. from eventlet.support import greenlets as greenlet
3. from eventlet import hubs
4. import eventlet
5. 
6. evt = event.Event()
7. 
8. def waiter():
9.     print "about to wait"
10.     result = evt.wait()
11.     print 'waited for,',result
12. 
13. 
14. hub = hubs.get_hub()
15. g = eventlet.spawn(waiter)
16. eventlet.sleep(0)
17. evt.send('a')
18. eventlet.sleep(0)
```

在eventlet的官方上有这段代码，引入这段代码是因为event在调度greenthread中有重要的作用。上面的代码运行结果如下：

```
1. about to wait
2. waited for,a
```

首先解释下，调用spawn会创建一个greenthread放入到hub中，然后使用sleep(0)从当前的greenlet切换到刚才创建的greenthread，就开始执行waiter函数，打印第一行。然后函数就在此wait了，我们前面介绍了wait会触发hub的switch方法，回到MAINLOOP的循环中，由于在每一次循环都将next_timer清空了，所有要执行的timer都添加到self.timer这个小堆中去了。在MAINLOOP中，由于这个包含timer的wait已经被执行过一次，所以下次循环时不会再执行了，sleep函数就让程序切换到了我们写的代码上来，接着运行evt.send(’a’),这一行同样触发了hub的调度，接着运行到waiter阻塞的地方，我们发现，这儿send有一个很关键的作用，用来在不同的greenthread中传递结果。所以后面紧接着打印了waited for,a。最后一句sleep则从MAINLOOP的空循环中切回到我们的代码尾，然后结束。

通过event，就明白了event可以用来再不同的greenthread中进行值的传递。官方文档介绍了，event和队列类似，只是event中只有一个元素，send函数能够用来唤醒正在等待的waiters，是不是和线程中的诸多概念相似了。

总结

我们回过头来看整个hub作为调度模块的结构，hub调度对象是Timer实例，只是有的timer实例封装了greenthread的switch函数，从而切换到greenthread的执行。不同的greenthread中进行通信，这需要用来event，每个greenthread封装一个event实例，event完成对本身greenthread的结果传递。而我们普通使用的spawn系列函数则是整个调度系统提供对外的api，使用该api，则可以将我们的任务作为一个greenthread添加到hub中，让它调度。至此，可以大致看到eventlet的

调度框架。并且后面将提到的greenpool则是一个greenthread的池，使用也差不多了。