11.12 理解事件驱动的IO¶

问题¶

你应该已经听过基于事件驱动或异步I/O的包，但是你还不能完全理解它的底层到底是怎样工作的， 或者是如果使用它的话会对你的程序产生什么影响。

解决方案¶

事件驱动I/O本质上来讲就是将基本I/O操作（比如读和写）转化为你程序需要处理的事件。 例如，当数据在某个socket上被接受后，它会转换成一个 `receive` 事件，然后被你定义的回调方法或函数来处理。 作为一个可能的起始点，一个事件驱动的框架可能会以一个实现了一系列基本事件处理器方法的基类开始：

```
class EventHandler:
    def fileno(self):
        'Return the associated file descriptor'
        raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Return True if receiving is allowed'
        return False

    def handle_receive(self):
        'Perform the receive operation'
        pass

    def wants_to_send(self):
        'Return True if sending is requested'
        return False

    def handle_send(self):
        'Send outgoing data'
        pass
```

这个类的实例作为插件被放入类似下面这样的事件循环中：

```
import select

def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv, wants_send, [])
        for h in can_recv:
            h.handle_receive()
        for h in can_send:
            h.handle_send()
```

事件循环的关键部分是 `select()` 调用，它会不断轮询文件描述符从而激活它。 在调用 `select()` 之前，事件循环会询问所有的处理器来决定哪一个想接受或发生。 然后它将结果列表提供给 `select()` 。然后 `select()` 返回准备接受或发送的对象组成的列表。 然后相应的 `handle_receive()` 或 `handle_send()` 方法被触发。

编写应用程序的时候，`EventHandler` 的实例会被创建。例如，下面是两个简单的基于UDP网络服务的处理器例子：

```
import socket
import time

class UDPServer(EventHandler):
    def __init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)

    def fileno(self):
        return self.sock.fileno()
```

```
    def wants_to_receive(self):
        return True

class UDPTimeServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)

if __name__ == '__main__':
    handlers = [ UDPTimeServer(('',14000)), UDPEchoServer(('',15000))  ]
    event_loop(handlers)
```

测试这段代码，试着从另外一个Python解释器连接它：

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'',('localhost',14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello',('localhost',15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>
```

实现一个TCP服务器会更加复杂一点，因为每一个客户端都要初始化一个新的处理器对象。 下面是一个TCP应答客户端例子：

```
class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

    def handle_receive(self):
        client, addr = self.sock.accept()
        # Add the client to the event loop's handler list
        self.handler_list.append(self.client_handler(client, self.handler_list))

class TCPClient(EventHandler):
    def __init__(self, sock, handler_list):
        self.sock = sock
        self.handler_list = handler_list
        self.outgoing = bytearray()

    def fileno(self):
        return self.sock.fileno()

    def close(self):
        self.sock.close()
        # Remove myself from the event loop's handler list
        self.handler_list.remove(self)

    def wants_to_send(self):
        return True if self.outgoing else False
```

```
    def handle_send(self):
        nsent = self.sock.send(self.outgoing)
        self.outgoing = self.outgoing[nsent:]

class TCPEchoClient(TCPClient):
    def wants_to_receive(self):
        return True

    def handle_receive(self):
        data = self.sock.recv(8192)
        if not data:
            self.close()
        else:
            self.outgoing.extend(data)

if __name__ == '__main__':
   handlers = []
   handlers.append(TCPServer(('',16000), TCPEchoClient, handlers))
   event_loop(handlers)
```

TCP例子的关键点是从处理器中列表增加和删除客户端的操作。 对每一个连接，一个新的处理器被创建并加到列表中。当连接被关闭后，每个客户端负责将其从列表中删除。 如果你运行程序并试着用Telnet或类似工具连接，它会将你发送的消息回显给你。并且它能很轻松的处理多客户端连接。

讨论¶

实际上所有的事件驱动框架原理跟上面的例子相差无几。实际的实现细节和软件架构可能不一样， 但是在最核心的部分，都会有一个轮询的循环来检查活动socket，并执行响应操作。

事件驱动I/O的一个可能好处是它能处理非常大的并发连接，而不需要使用多线程或多进程。 也就是说，`select()` 调用（或其他等效的）能监听大量的socket并响应它们中任何一个产生事件的。 在循环中一次处理一个事件，并不需要其他的并发机制。

事件驱动I/O的缺点是没有真正的同步机制。 如果任何事件处理器方法阻塞或执行一个耗时计算，它会阻塞所有的处理进程。 调用那些并不是事件驱动风格的库函数也会有问题，同样要是某些库函数调用会阻塞，那么也会导致整个事件循环停止。

对于阻塞或耗时计算的问题可以通过将事件发送个其他单独的线程或进程来处理。 不过，在事件循环中引入多线程和多进程是比较棘手的， 下面的例子演示了如何使用 `concurrent.futures` 模块来实现：

```
from concurrent.futures import ThreadPoolExecutor
import os

class ThreadPoolHandler(EventHandler):
    def __init__(self, nworkers):
        if os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                  socket.SOCK_STREAM)
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def fileno(self):
        return self.done_sock.fileno()

    # Callback that executes when the thread is done
    def _complete(self, callback, r):
```

```
        self.pending.append((callback, r.result()))
        self.signal_done_sock.send(b'x')

    # Run a function in a thread pool
    def run(self, func, args=(), kwargs={},*,callback):
        r = self.pool.submit(func, *args, **kwargs)
        r.add_done_callback(lambda r: self._complete(callback, r))

    def wants_to_receive(self):
        return True

    # Run callback functions of completed work
    def handle_receive(self):
        # Invoke all pending callback functions
        for callback, result in self.pending:
            callback(result)
            self.done_sock.recv(1)
        self.pending = []
```

在代码中，`run()` 方法被用来将工作提交给回调函数池，处理完成后被激发。 实际工作被提交给 `ThreadPoolExecutor` 实例。 不过一个难点是协调计算结果和事件循环，为了解决它，我们创建了一对socket并将其作为某种信号量机制来使用。 当线程池完成工作后，它会执行类中的 `_complete()` 方法。 这个方法再某个socket上写入字节之前会讲挂起的回调函数和结果放入队列中。 `fileno()` 方法返回另外的那个socket。 因此，这个字节被写入时，它会通知事件循环，然后 `handle_receive()` 方法被激活并为所有之前提交的工作执行回调函数。 坦白讲，说了这么多连我自己都晕了。 下面是一个简单的服务器，演示了如何使用线程池来实现耗时的计算：

```
# A really bad Fibonacci implementation
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

class UDPFibServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(128)
        n = int(msg)
        pool.run(fib, (n,), callback=lambda r: self.respond(r, addr))

    def respond(self, result, addr):
        self.sock.sendto(str(result).encode('ascii'), addr)

if __name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [ pool, UDPFibServer(('',16000))]
    event_loop(handlers)
```

运行这个服务器，然后试着用其它Python程序来测试它：

```
from socket import *
sock = socket(AF_INET, SOCK_DGRAM)
for x in range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
    print(resp[0])
```

你应该能在不同窗口中重复的执行这个程序，并且不会影响到其他程序，尽管当数字便越来越大时候它会变得越来越慢。

已经阅读完了这一小节，那么你应该使用这里的代码吗？也许不会。你应该选择一个可以完成同样任务的高级框架。不过，如果你理解了基本原理，你就能理解这些框架所使用的核心技术。 作为对回调函数编程的替代，事件驱动编码有时候会使用到协程，参考12.12小节的一个例子。