主要内容

1. Gevent协程

2. Select\Poll\Epoll异步IO与事件驱动

3. selectors 模块 多并发演示

协程

协程，又称微线程，纤程。英文名Coroutine。一句话说明什么是线程：**协程是一种用户态的轻量级线程**。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：

协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

协程的好处：

- 无需线程上下文切换的开销
- 无需原子操作锁定及同步的开销
- "原子操作(atomic operation)是不需要synchronized"，所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切换到另一个线程）。原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序是不可以被打乱，或者切割掉只执行部分。视作整体是原子性的核心。
- 方便切换控制流，简化编程模型
- 高并发+高扩展性+低成本：一个CPU支持上万的协程都不是问题。所以很适合用于高并发处理。

缺点：

- 无法利用多核资源：协程的本质是个单线程,它不能同时将 单个CPU 的多个核用上,协程需要和进程配合才能运行在多CPU上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是cpu密集型应用。
- 进行阻塞（Blocking）操作（如IO时）会阻塞掉整个程序

协程一个标准定义，即符合什么条件就能称之为协程：

1. 必须在只有一个单线程里实现并发

2. 修改共享数据不需加锁

3. 用户程序里自己保存多个控制流的上下文栈

4. 一个协程遇到IO操作自动切换到其它协程

Greenlet

greenlet是一个用C实现的协程模块，相比与python自带的yield，它可以使你在任意函数之间随意切换，而不需把这个函数先声明为generator

```
from greenlet import greenlet def test1():  print(12)  gr2.switch()  print(34)  gr2.switch()  def test2():  print(56)  gr1.switch()  print(78) gr1 = greenlet(test1) gr2 = greenlet(test2) gr1.switch()
```

感觉确实用着比generator还简单了呢，但好像还没有解决一个问题，就是遇到IO操作，自动切换，对不对?

Gevent

Gevent 是一个第三方库，可以轻松通过gevent实现并发同步或异步编程，在gevent中用到的主要模式是**Greenlet**, 它是以C扩展模块形式接入Python的轻量级协程。 Greenlet全部运行在主程序操作系统进程的内部，但它们被协作式地调度。

```
import gevent def func1():  print('我在吃西瓜...')  gevent.sleep(2)  print('我回来继续吃西瓜...')  def func2():  print('我去吃芒果...')  gevent.sleep(1)  print('我吃完西瓜，回去继续吃芒果...')  gevent.joinall([  gevent.spawn(func1),  gevent.spawn(func2),  ])
```

输出：

我在吃西瓜...
我去吃芒果...
我回来继续吃西瓜...
我吃完西瓜，回去继续吃芒果...

同步与异步的性能区别

```
# _*_coding:utf-8_*_ # Author:Jaye He import gevent, time from urllib.request import urlopen from gevent import monkey monkey.patch_all()  # 把当前程序的所有的IO操作给我单独的做标记触发gevent的遇到IO自动切换线程 def func(url):     print('[*] Get %s' % url)     res = urlopen(url)     data = res.read()     print('%d bytes received from %s' % (len(data), url)) url = [     'https://www.python.org/',     'https://www.yahoo.com/',     'https://github.com/' ] time_start = time.time() for i in url:     func(i) print('同步cost', time.time() - time_start) async_time_start = time.time() gevent.joinall(     [         gevent.spawn(func, 'https://www.python.org/'),         gevent.spawn(func, 'https://www.yahoo.com/'),         gevent.spawn(func, 'https://github.com/')     ] ) print('异步cost', time.time() - async_time_start)
```

输出：

```
[*] Get https://www.python.org/
48708 bytes received from https://www.python.org/
[*] Get https://www.yahoo.com/
478886 bytes received from https://www.yahoo.com/
[*] Get https://github.com/
55867 bytes received from https://github.com/
同步cost 3.859987258911133
[*] Get https://www.python.org/
[*] Get https://www.yahoo.com/
[*] Get https://github.com/
48708 bytes received from https://www.python.org/
55867 bytes received from https://github.com/
475663 bytes received from https://www.yahoo.com/
异步cost 1.8283183574676514
```

效果 很明显

通过gevent实现单线程下的多socket并发

server side

```
import socket import gevent from gevent import socket, monkey monkey.patch_all() def server(port):     s = socket.socket()     s.bind(('0.0.0.0', port))     s.listen(500)     while True:         cli, addr = s.accept()         gevent.spawn(handle_request, cli) def handle_request(conn):     try:         while True:             data = conn.recv(1024)             print("recv:", data)             conn.send(data)             if not data:                 conn.shutdown(socket.SHUT_WR)     except Exception as ex:         print(ex)     finally:         conn.close() if __name__ == '__main__':     server(8001)
```

并发100个socket连接

```
import socket import threading def sock_conn():     client = socket.socket()
    client.connect(("localhost", 8001))     count = 0     while True:
        client.send(("hello %s" % count).encode("utf-8"))         data = client.recv(1024)
        print("[%s]recv from server:" % threading.get_ident(), data.decode())    # 结果
        count += 1     client.close() for i in range(100):     t = threading.Thread(target=sock_conn)
    t.start()
```

论事件驱动与异步IO

通常，我们写服务器处理模型的程序时，有以下几种模型：

（1）每收到一个请求，创建一个新的进程，来处理该请求；

（2）每收到一个请求，创建一个新的线程，来处理该请求；

（3）每收到一个请求，放入一个事件列表，让主进程通过非阻塞I/O方式来处理请求

上面的几种方式，各有千秋，

第（1）中方法，由于创建新的进程的开销比较大，所以，会导致服务器性能比较差,但实现比较简单。

第（2）种方式，由于要涉及到线程的同步，有可能会面临死锁等问题。

第（3）种方式，在写应用程序代码时，逻辑比前面两种都复杂。

综合考虑各方面因素，一般普遍认为第（3）种方式是大多数网络服务器采用的方式

看图说话讲事件驱动模型

在UI编程中，常常要对鼠标点击进行相应，首先如何获得鼠标点击呢?

方式一：创建一个线程，该线程一直循环检测是否有鼠标点击，那么这个方式有以下几个缺点：

1. CPU资源浪费，可能鼠标点击的频率非常小，但是扫描线程还是会一直循环检测，这会造成很多的CPU资源浪费；如果扫描鼠标点击的接口是阻塞的呢?

2. 如果是堵塞的，又会出现下面这样的问题，如果我们不但要扫描鼠标点击，还要扫描键盘是否按下，由于扫描鼠标时被堵塞了，那么可能永远不会去扫描键盘；

3. 如果一个循环需要扫描的设备非常多，这又会引来响应时间的问题；

所以，该方式是非常不好的。

方式二：就是事件驱动模型

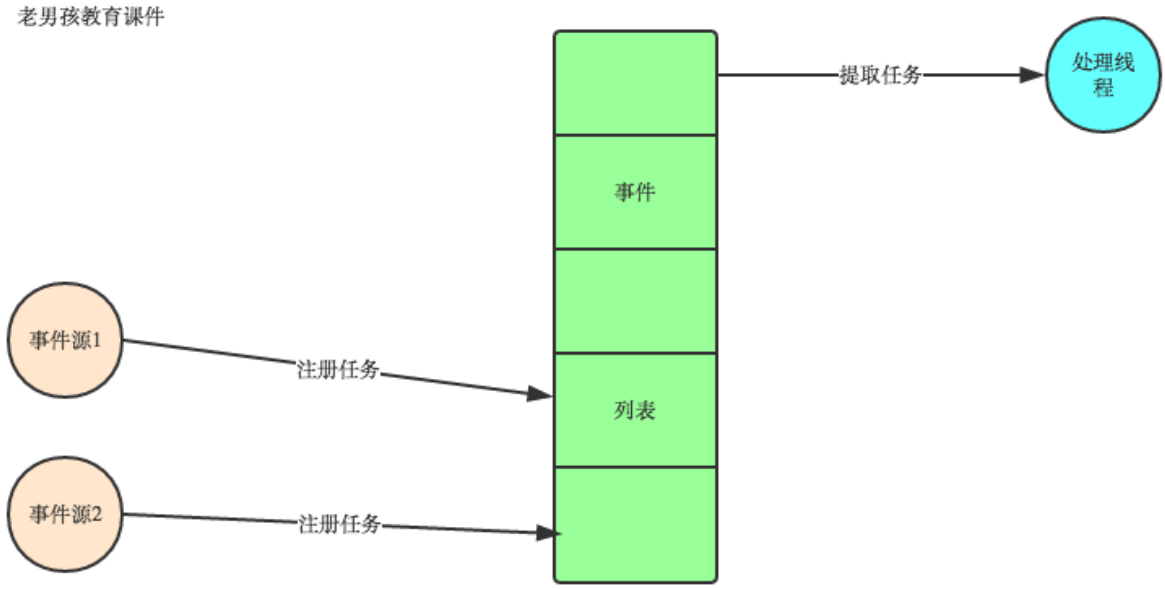
目前大部分的UI编程都是事件驱动模型，如很多UI平台都会提供onClick()事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：

1. 有一个事件（消息）队列；

2. 鼠标按下时，往这个队列中增加一个点击事件（消息）；

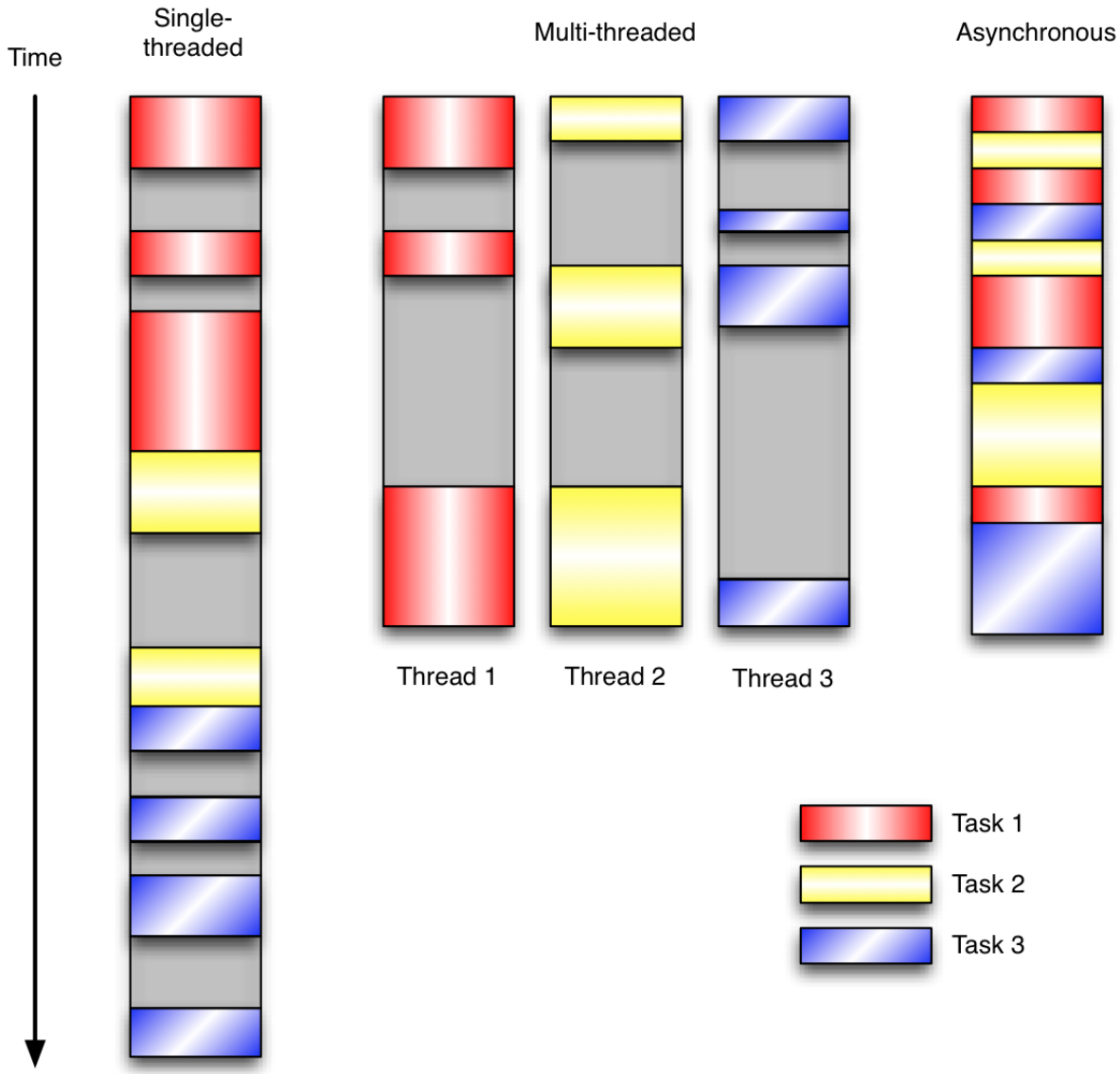
3. 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如onClick()、onKeyDown()等；

4. 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；



事件驱动编程是一种编程范式，这里程序的执行流由外部事件来决定。它的特点是包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理。另外两种常见的编程范式是（单线程）同步以及多线程编程。

让我们用例子来比较和对比一下单线程、多线程以及事件驱动编程模型。下图展示了随着时间的推移，这三种模式下程序所做的工作。这个程序有3个任务需要完成，每个任务都在等待I/O操作时阻塞自身。阻塞在I/O操作上所花费的时间已经用灰色框标示出来了。



在单线程同步模型中，任务按照顺序执行。如果某个任务因为I/O而阻塞，其他所有的任务都必须等待，直到它完成之后它们才能依次执行。这种明确的执行顺序和串行化处理的行为是很容易推断得出的。如果任务之间并没有互相依赖的关系，但仍然需要互相等待的话这就使得程序不必要的降低了运行速度。

在多线程版本中，这3个任务分别在独立的线程中执行。这些线程由操作系统来管理，在多处理器系统上可以并行处理，或者在单处理器系统上交错执行。这使得当某个线程阻塞在某个资源的同时其他线程得以继续执行。与完成类似功能的同步程序相比，这种方式更有效率，但程序员必须写代码来保护共享资源，防止其被多个线程同时访问。多线程程序更加难以推断，因为这类程序不得不通过线程同步机制如锁、可重入函数、线程局部存储或者其他机制来处理线程安全问题，如果实现不当就会导致出现微妙且令人痛不欲生的bug。

在事件驱动版本的程序中，3个任务交错执行，但仍然在一个单独的线程控制中。当处理I/O或者其他昂贵的操作时，注册一个回调到事件循环中，然后当I/O操作完成时继续执行。回调描述了该如何处理某个事件。事件循环轮询所有的事件，当事件到来时将它们分配给等待处理事件的回调函数。这种方式让程序尽可能的得以执行而不需要用到额外的线程。事件驱动型程序比多线程程序更容易推断出行为，因为程序员不需要关心线程安全问题。

当我们面对如下的环境时，事件驱动模型通常是一个好的选择：

1. 程序中有许多任务，而且...
2. 任务之间高度独立（因此它们不需要互相通信，或者等待彼此）而且...
3. 在等待事件到来时，某些任务会阻塞。

当应用程序需要在任务间共享可变的数据时，这也是一个不错的选择，因为这里不需要采用同步处理。

网络应用程序通常都有上述这些特点，这使得它们能够很好的契合事件驱动编程模型。

Select\Poll\Epoll异步IO

参考alex老师 讲解的 Select\Poll\Epoll 发展 和 Select详解

http://www.cnblogs.com/alex3714/p/4372426.html

select 多并发socket 例子

select socket server

select socket client

selectors模块多并发演示

使用selectors 模块(协程)实现500并发上传下载, 在本机win10上测试超过500就出现以下情况,主要是windows上selectors 使用的是select, 显示使用的fd太多,有限制

ValueError: too many file descriptors in select()

selectors 服务端

```
# _*_coding:utf-8_*_ # Author:Jaye He import selectors import socket import os
class SelectorsServer(object):      def __init__(self, address):          self.address = address
        self.server = socket.socket()          self.server.setblocking(False)
        self.sel = selectors.DefaultSelector()          self.fd = {}    # 储存每个conn的对应文件句柄和文件大小      def server_start(self):          """开启Server,然后交给self.accept监听连接请求(接受数据注册EVENT_READ事件)"""          self.server.bind(self.address)          self.server.listen(1000)
        self.sel.register(self.server, selectors.EVENT_READ, self.accept)
    def accept(self, server):          """建立连接,然后交给self.read接受从客户端来的数据(接受数据注册EVENT_READ事件)"""          conn, addr = server.accept()          conn.setblocking(False)
        # print('accepted form', conn, addr)
        self.sel.register(conn, selectors.EVENT_READ, self.read)      def read(self, conn):
        """接受从客户端来的数据,然后判断需要的操作"""          data = conn.recv(1024)          if data:
            if data == b'get':                  # 接收到'get', 就打开文件'歌词.txt',把文件句柄传入self.fd
                # 然后注册EVENT_WRITE事件交给self.get_write发数据给客户端
                f = open('歌词.txt', 'rb')                  file_size = os.path.getsize('歌词.txt')
                conn.send(str(file_size).encode())
                self.fd.update({conn: {'fd': f, 'file_size': file_size}})                  self.sel.unregister(conn)
                self.sel.register(conn, selectors.EVENT_WRITE, self.get_write)
            elif data == b'put':                  # 接收到'put', 就注册EVENT_READ事件交给self.put_read处理                  self.sel.unregister(conn)
                self.sel.register(conn, selectors.EVENT_READ, self.put_read)          else:              # 接受到空数据就从事件列表中注销conn的事件,同时关闭conn
            print('\033[1;33mclosing %s\033[0m' % conn)              self.sel.unregister(conn)
            conn.close()      def put_read(self, conn):          """接受并处理客户端发来的数据"""
        data = conn.recv(1024)          if data:      # 有数据,打印数据大小和对应conn
            print(len(data), conn)          else:          # 没有就注销conn,并关闭conn
            print('\033[1;33mclosing %s\033[0m' % conn)              self.sel.unregister(conn)
            conn.close()      def get_write(self, conn):          """给客户端发送数据"""
```

```
        f = self.fd[conn]['fd']          file_size = self.fd[conn]['file_size']          conn.send(f.readline())
        progress = f.tell()          if progress == file_size:              # 文件发送完毕,然后继续注册
EVENT_READ事件,交给self.read处理              self.sel.unregister(conn)
            self.sel.register(conn, selectors.EVENT_READ, self.read)      def monitor(self):
        """监听EVENT事件列表,有活动的事件就交给相应方法处理"""          while True:
            events_list = self.sel.select()              for key, mask in events_list:
                callback = key.data                  callback(key.fileobj) def main():
    fs = SelectorsServer(('0.0.0.0', 9000))      fs.server_start()      fs.monitor()
if __name__ == '__main__':      main()
```

selectors 500并发客户端

```
# _*_coding:utf-8_*_ # Author:Jaye He import socket import threading def sock_conn():
    pid = threading.current_thread()      client = socket.socket()
    client.connect(('localhost', 9000))      # 从服务器下载数据      client.send(b'get')
    file_size = int(client.recv(1024).decode())      get_size = 0      while True:
        get_data = client.recv(1024)          get_size += len(get_data)
        print('get', '%sbytes' % len(get_data), pid)          if get_size == file_size:              break
    print('\033[1;33mGet Completed %s\033[0m' % pid)      # 向服务器上传数据
    client.send(b'put')      f = open('歌词.txt', 'rb')      while True:          data = f.readline()
        if len(data) != 0:              client.send(data)              print('put', '%sbytes' % len(data), pid)
        else:              print('\033[1;33mPut Completed %s\033[0m' % pid)              break
    f.close()      client.close() threading_list = [] for i in range(500):
    t = threading.Thread(target=sock_conn)      threading_list.append(t)      t.start()
for t in threading_list:      t.join()
```

selectors 详细请参考官方文档