

公告

wiki和教程：[www.pythontav.com](http://www.pythontav.com)

免费教学视频：[B站](#)：凸头统治地球

高级专题教程：[网易云课堂](#)：武沛齐

聊技术，加武Sir微信



扫一扫上面的二维码图案，加我微信



Python技术交流群：737658057

软件测试开发交流群：721023555

昵称：武沛齐  
园龄：8年  
粉丝：9942  
关注：44  
+加关注

我的标签

Python(17)  
ASP.NET MVC(15)  
python之路(7)  
Tornado源码分析(5)  
每天一道Python面试题(5)  
crm项目(4)  
面试都在问什么？(2)  
Python开源组件 - Tyrion(1)  
Python面试315题(1)  
Python企业面试题讲解(1)

积分与排名

积分 - 425135  
排名 - 778

随笔分类

JavaScript(1)  
MVC(15)  
Python(17)  
面试都在问什么系列？【图】(2)  
其他(37)

随笔 - 140 文章 - 164 评论 - 887

Python之路【第六篇】： socket

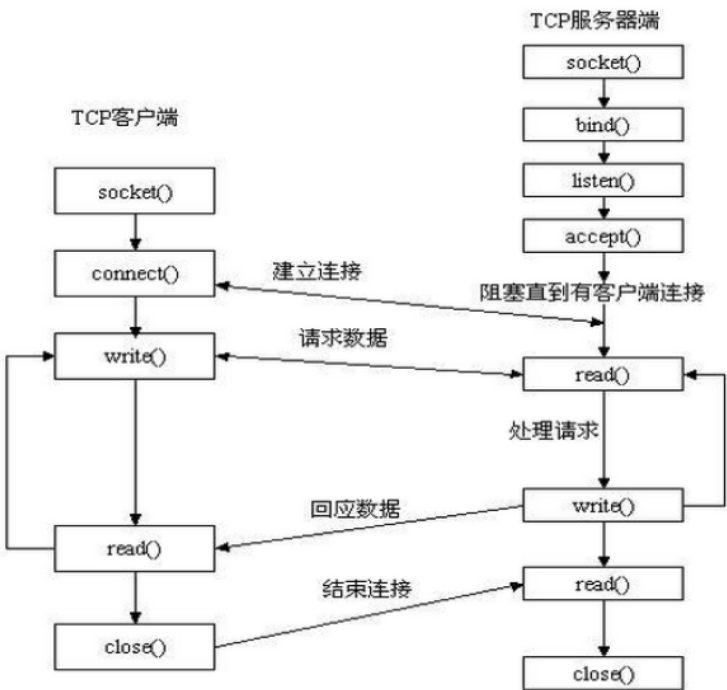
Socket

socket通常也称作"套接字"，用于描述IP地址和端口，是一个通信链的句柄，应用程序通常通过"套接字"向网络发出请求或者应答网络请求。

socket起源于Unix，而Unix/Linux基本哲学之一就是"一切皆文件"，对于文件用【打开】【读写】【关闭】模式来操作。socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭）

socket和file的区别：

- file模块是针对某个指定文件进行【打开】【读写】【关闭】
- socket模块是针对 服务器端 和 客户端Socket 进行【打开】【读写】【关闭】



```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1',9999)

sk = socket.socket()
sk.bind(ip_port)
sk.listen(5)

while True:
    print 'server waiting...'
    conn,addr = sk.accept()

    client_data = conn.recv(1024)
    print client_data
```

企业面试题及答案(1)  
请求响应(6)  
设计模式(9)  
微软C#(34)

随笔档案

- 2020年6月(1)
- 2020年5月(1)
- 2019年11月(1)
- 2019年10月(1)
- 2019年9月(4)
- 2018年12月(1)
- 2018年8月(1)
- 2018年5月(2)
- 2018年4月(1)
- 2017年8月(1)
- 2017年5月(1)
- 2017年3月(1)
- 2016年10月(1)
- 2016年7月(1)
- 2015年10月(1)
- 2015年8月(1)
- 2015年7月(1)
- 2015年6月(2)
- 2015年4月(2)
- 2014年3月(3)
- 2014年1月(3)
- 2013年12月(2)
- 2013年11月(2)
- 2013年10月(7)
- 2013年8月(17)
- 2013年7月(1)
- 2013年6月(14)
- 2013年5月(23)
- 2013年4月(3)
- 2013年3月(13)
- 2013年2月(1)
- 2012年11月(26)

相册

git(14)

最新评论

- 1. Re:1. 路过面了个试就拿到2个offer。是运气吗?  
听了大王的讲课，觉得大王真是厉害，年轻有为!  

--Xiyue666
- 2. Re: Celery  
s3.py中 使用async为变量是关键字，会报错把?  

--killer-147
- 3. Re:不吹不擂，你想要的Python面试都在这里了【315+道题】

```
conn.sendall('不要回答,不要回答,不要回答')

conn.close()
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import socket

ip_port = ('127.0.0.1',9999)

sk = socket.socket()
sk.connect(ip_port)

sk.sendall('请求占领地球')

server_reply = sk.recv(1024)
print server_reply

sk.close()
```

sk.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1)

WEB服务应用：

```
1  #!/usr/bin/env python
2  #coding:utf-8
3  import socket
4
5  def handle_request(client):
6      buf = client.recv(1024)
7      client.send("HTTP/1.1 200 OK\r\n\r\n")
8      client.send("Hello, World")
9
10 def main():
11     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12     sock.bind(('localhost',8080))
13     sock.listen(5)
14
15     while True:
16         connection, address = sock.accept()
17         handle_request(connection)
18         connection.close()
19
20 if __name__ == '__main__':
21     main()
```

更多功能

sk = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM,0)

参数一：地址簇

- socket.AF\_INET IPv4（默认）
- socket.AF\_INET6 IPv6
- socket.AF\_UNIX 只能够用于单一的Unix系统进程间通信

参数二：类型

- socket.SOCK\_STREAM 流式socket，for TCP（默认）
- socket.SOCK\_DGRAM 数据报式socket，for UDP

先赞了再说 日后再说 哈哈

--Xiyue666

#### 4. Re:Python开发【第十九篇】：Python操作MySQL

武大大 我学到这里想放弃了 怎么办？学不进去了！

--Xiyue666

#### 5. Re:为什么很多IT公司不喜欢进过培训机构的人呢？

@toEverybody 您搁这混淆概念呢？？明明是在讲两个国家间以前的技术人才的区别，这咋能让培训班的背锅？那科班的那些孩子呢？那些计算机专业的研究生们呢？思想深度不是培训不培训就能划分清楚的。...

--温故而新

socket.SOCK\_RAW 原始套接字，普通的套接字无法处理ICMP、IGMP等网络报文，而SOCK\_RAW可以；其次，SOCK\_RAW也可以处理特殊的IPv4报文；此外，利用原始套接字，可以通过IP\_HDRINCL套接字选项由用户构造IP头。

socket.SOCK\_RDM 是一种可靠的UDP形式，即保证交付数据报但不保证顺序。SOCK\_RDM用来提供对原始协议的低级访问，在需要执行某些特殊操作时使用，如发送ICMP报文。SOCK\_RDM通常仅限于高级用户或管理员运行的程序使用。

socket.SOCK\_SEQPACKET 可靠的连续数据包服务

参数三：协议

0 （默认）与特定的地址家族相关的协议,如果是 0，则系统就会根据地址格式和套接类别,自动选择一个合适的协议

☐



```
import socket
ip_port = ('127.0.0.1',9999)
sk = socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0)
sk.bind(ip_port)

while True:
    data = sk.recv(1024)
    print data

import socket
ip_port = ('127.0.0.1',9999)

sk = socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0)
while True:
    inp = raw_input('数据: ').strip()
    if inp == 'exit':
        break
    sk.sendto(inp,ip_port)

sk.close()
```



#### sk.bind(address)

s.bind(address) 将套接字绑定到地址。address地址的格式取决于地址族。在AF\_INET下，以元组 (host,port) 的形式表示地址。

#### sk.listen(backlog)

开始监听传入连接。backlog指定在拒绝连接之前，可以挂起的最大连接数量。

backlog等于5，表示内核已经接到了连接请求，但服务器还没有调用accept进行处理。连接个数最大为5

这个值不能无限大，因为要在内核中维护连接队列

#### sk.setblocking(bool)

是否阻塞（默认True），如果设置False，那么accept和recv时一旦无数据，则报错。

#### sk.accept()

接受连接并返回 (conn,address)，其中conn是新的套接字对象，可以用来接收和发送数据。address是连接客户端的地址。

接收TCP 客户的连接（阻塞式）等待连接的到来

**sk.connect(address)**

连接到address处的套接字。一般，address的格式为元组 (hostname,port) ,如果连接出错，返回socket.error错误。

**sk.connect\_ex(address)**

同上，只不过会有返回值，连接成功时返回 0，连接失败时候返回编码，例如：10061

**sk.close()**

关闭套接字

**sk.recv(bufsize[,flag])**

接受套接字的数据。数据以字符串形式返回，bufsize指定最多可以接收的数量。flag提供有关消息的其他信息，通常可以忽略。

**sk.recvfrom(bufsize[,flag])**

与recv()类似，但返回值是 (data,address) 。其中data是包含接收数据的字符串，address是发送数据的套接字地址。

**sk.send(string[,flag])**

将string中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于string的字节大小。即：可能未将指定内容全部发送。

**sk.sendall(string[,flag])**

将string中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。

内部通过递归调用send，将所有内容发送出去。

**sk.sendto(string[,flag],address)**

将数据发送到套接字，address是形式为 (ipaddr, port) 的元组，指定远程地址。返回值是发送的字节数。该函数主要用于UDP协议。

**sk.settimeout(timeout)**

设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如 client 连接最多等待5s）

**sk.getpeername()**

返回连接套接字的远程地址。返回值通常是元组 (ipaddr,port) 。

**sk.getsockname()**

返回套接字自己的地址。通常是一个元组(ipaddr,port)

**sk.fileno()**

套接字的文件描述符

```

# 服务端
import socket

ip_port = ('127.0.0.1',9999)
sk = socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0)
sk.bind(ip_port)

while True:
    data,(host,port) = sk.recvfrom(1024)
    print(data,host,port)
    sk.sendto(bytes('ok', encoding='utf-8'), (host,port))
```

```
#客户端
import socket
ip_port = ('127.0.0.1',9999)

sk = socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0)
while True:
    inp = input('数据: ').strip()
    if inp == 'exit':
        break
    sk.sendto(bytes(inp, encoding='utf-8'),ip_port)
    data = sk.recvfrom(1024)
    print(data)

sk.close()
```



## 实例：智能机器人

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1',8888)
sk = socket.socket()
sk.bind(ip_port)
sk.listen(5)

while True:
    conn,address = sk.accept()
    conn.sendall('欢迎致电 10086, 请输入1xxx,0转人工服务.')
    Flag = True
    while Flag:
        data = conn.recv(1024)
        if data == 'exit':
            Flag = False
        elif data == '0':
            conn.sendall('通过可能会被录音.balabala一大推')
        else:
            conn.sendall('请重新输入.')
    conn.close()
```



```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1',8005)
sk = socket.socket()
sk.connect(ip_port)
sk.settimeout(5)

while True:
    data = sk.recv(1024)
    print 'receive:',data
    inp = raw_input('please input:')
```



```

sk.sendall(inp)

if inp == 'exit':
    break

sk.close()

```



## IO多路复用

I/O多路复用指：通过一种机制，可以**监视多个描述符**，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

## Linux

Linux中的 select, poll, epoll 都是IO多路复用的机制。

```

1  select
2
3  select最早于1983年出现在4.2BSD中，它通过一个select()系统调用来监视多个文件描述符
4  select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点，事实上从现在看
5  select的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在Linux上一般
6  另外，select()所维护的存储大量文件描述符的数据结构，随着文件描述符数量的增大，其复
7
8  poll
9
10 poll在1986年诞生于System V Release 3，它和select在本质上没有多大差别，但是poll
11 poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核
12 另外，select()和poll()将就绪的文件描述符告诉进程后，如果进程没有对其进行IO操作，
13
14  epoll
15
16 直到Linux2.6才出现了由内核直接支持的实现方法，那就是epoll，它几乎具备了之前所说的
17  epoll可以同时支持水平触发和边缘触发（Edge Triggered，只告诉进程哪些文件描述符刚刚
18  epoll同样只告知那些就绪的文件描述符，而且当我们调用epoll_wait()获得就绪文件描述符
19  另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在i

```

## Python

Python中有一个select模块，其中提供了：select、poll、epoll三个方法，分别调用系统的 select, poll, epoll 从而实现IO多路复用。

```

1  Windows Python:
2      提供: select
3  Mac Python:
4      提供: select
5  Linux Python:
6      提供: select、poll、epoll

```

*注意：网络操作、文件操作、终端操作等均属于IO操作，对于windows只支持Socket操作，其他系统支持其他IO操作，但是无法检测普通文件操作自动上次读取是否已经变化。*

对于select方法：

```

1  句柄列表11，句柄列表22，句柄列表33 = select.select(句柄序列1，句柄序列2，句柄序
2
3  参数： 可接受四个参数（前三个必须）
4  返回值：三个列表
5
6  select方法用来监视文件句柄，如果句柄发生变化，则获取该句柄。
7  1、当 参数1 序列中的句柄发生可读时（accept和read），则获取发生变化的句柄并添加到
8  2、当 参数2 序列中含有句柄时，则将该序列中所有的句柄添加到 返回值2 序列中
9  3、当 参数3 序列中的句柄发生错误时，则将该发生错误的句柄添加到 返回值3 序列中
10 4、当 超时时间 未设置，则select会一直阻塞，直到监听的句柄发生变化
11 当 超时时间 = 1时，那么如果监听的句柄均无任何变化，则select会阻塞 1 秒，之后

```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import select
import threading
import sys

while True:
    readable, writeable, error = select.select([sys.stdin], [], [], 1)
    if sys.stdin in readable:
        print 'select get stdin', sys.stdin.readline()
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket
import select

sk1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sk1.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sk1.bind(('127.0.0.1', 8002))
sk1.listen(5)
sk1.setblocking(0)

inputs = [sk1,]

while True:
    readable_list, writeable_list, error_list = select.select(inputs, [], in
    for r in readable_list:
        # 当客户端第一次连接服务端时
        if sk1 == r:
            print 'accept'
            request, address = r.accept()
            request.setblocking(0)
            inputs.append(request)
        # 当客户端连接上服务端之后，再次发送数据时
        else:
            received = r.recv(1024)
            # 当正常接收客户端发送的数据时
            if received:
                print 'received data:', received
            # 当客户端关闭程序时
            else:
                inputs.remove(r)

sk1.close()
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1', 8002)
```

```
sk = socket.socket()
sk.connect(ip_port)

while True:
    inp = raw_input('please input:')
    sk.sendall(inp)
sk.close()
```



此处的Socket服务端相比与原生的Socket，他支持当某一个请求不再发送数据时，服务器端不会等待而是可以去处理其他请求的数据。但是，如果每个请求的耗时比较长时，select版本的服务器端也无法完成同时操作。

```
#!/usr/bin/env python
#coding:utf8

'''
    服务器的实现 采用select的方式
'''

import select
import socket
import sys
import Queue

#创建套接字并设置该套接字为非阻塞模式

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

#绑定套接字
server_address = ('localhost', 10000)
print >>sys.stderr, 'starting up on %s port %s'% server_address
server.bind(server_address)

#将该socket变成服务模式
#backlog等于5，表示内核已经接到了连接请求，但服务器还没有调用accept进行处理的连接个数最大
#这个值不能无限大，因为要在内核中维护连接队列

server.listen(5)

#初始化读取数据的监听列表，最开始时希望从server这个套接字上读取数据
inputs = [server]

#初始化写入数据的监听列表，最开始并没有客户端连接进来，所以列表为空

outputs = []

#要发往客户端的数据
message_queues = {}

while inputs:
    print >>sys.stderr, 'waiting for the next event'
    #调用select监听所有监听列表中的套接字，并将准备好的套接字加入到对应的列表中
    readable, writable, exceptional = select.select(inputs, outputs, inputs) #列表
    #监控文件句柄有某一处发生了变化 可写 可读 异常属于Linux中的网络编程
    #属于同步I/O操作，属于I/O复用模型的一种
    #rlist--等待到准备好读
    #wlist--等待到准备好写
    #xlist--等待到一种异常
    #处理可读的套接字

    '''
        如果server这个套接字可读，则说明有新链接到来
    '''
```



此时在server套接字上调用accept,生成一个与客户端通讯的套接字  
 并将与客户端通讯的套接字加入inputs列表,下一次可以通过select检查连接是否可读  
 然后在发往客户端的缓冲中加入一项,键名为:与客户端通讯的套接字,键值为空队列  
 select系统调用是用来让我们的程序监视多个文件句柄(file descriptor)的状态变化的  
 直到被监视的文件句柄有某一个或多个发生了状态改变

```
'''
```

若可读的套接字不是server套接字,有两种情况:一种是有数据到来,另一种是链接断开  
 如果有数据到来,先接收数据,然后将收到的数据填入往客户端的缓冲区中的对应位置,最后  
 将客户端通讯的套接字加入到写数据的监听列表:  
 如果套接字可读,但没有接收到数据,则说明客户端已经断开。这时需要关闭与客户端连接的  
 进行资源清理

```
'''
```

```
for s in readable:
    if s is server:
        connection,client_address = s.accept()
        print >>sys.stderr,'connection from',client_address
        connection.setblocking(0) #设置非阻塞
        inputs.append(connection)
        message_queues[connection] = Queue.Queue()
    else:
        data = s.recv(1024)
        if data:
            print >>sys.stderr,'received "%s" from %s'% \
                (data,s.getpeername())
            message_queues[s].put(data)
            if s not in outputs:
                outputs.append(s)
        else:
            print >>sys.stderr,'closing',client_address
            if s in outputs:
                outputs.remove(s)
            inputs.remove(s)
            s.close()
            del message_queues[s]
```

#处理可写的套接字

```
'''
```

在发送缓冲区中取出响应的数据,发往客户端。  
 如果没有数据需要写,则将套接字从发送队列中移除,select中不再监视

```
'''
```

```
for s in writable:
    try:
        next_msg = message_queues[s].get_nowait()

    except Queue.Empty:
        print >>sys.stderr,' ',s,getpeername(),'queue empty'
        outputs.remove(s)
    else:
        print >>sys.stderr,'sending "%s" to %s'% \
            (next_msg,s.getpeername())
        s.send(next_msg)
```

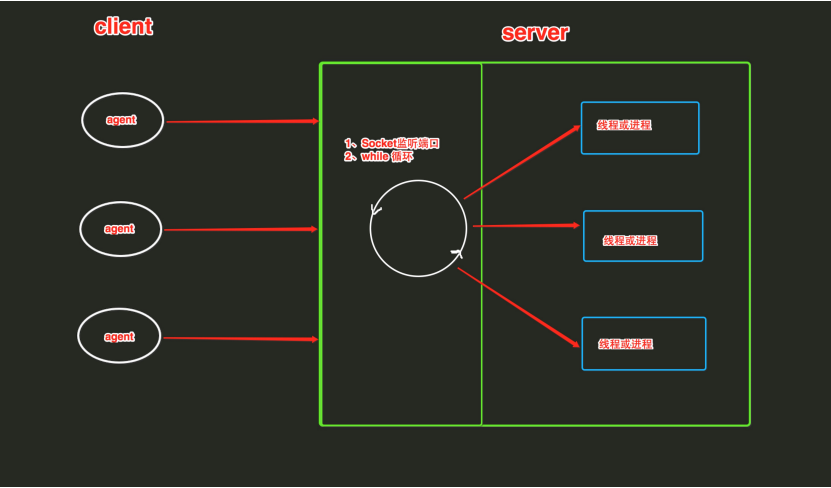
#处理异常情况

```
for s in exceptional:
    for s in exceptional:
        print >>sys.stderr,'exception condition on',s.getpeername()
        inputs.remove(s)
        if s in outputs:
            outputs.remove(s)
```

```
s.close()
del message_queues[s]
```

SocketServer模块

SocketServer内部使用 IO多路复用 以及 “多线程” 和 “多进程”， 从而实现并发处理多个客户端请求的Socket服务端。即： 每个客户端请求连接到服务器时， Socket服务端都会在服务器是创建一个“线程”或者“进程” 专门负责处理当前客户端的所有请求。



ThreadingTCPServer

ThreadingTCPServer实现的Soket服务器内部会为每个client创建一个“线程”， 该线程用来和客户端进行交互。

1、ThreadingTCPServer基础

使用ThreadingTCPServer:

- 创建一个继承自 SocketServer.BaseRequestHandler 的类
- 类中必须定义一个名称为 handle 的方法
- 启动ThreadingTCPServer

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import SocketServer

class MyServer(SocketServer.BaseRequestHandler):

    def handle(self):
        # print self.request,self.client_address,self.server
        conn = self.request
        conn.sendall('欢迎致电 10086, 请输入1xxx,0转人工服务.')
        Flag = True
        while Flag:
            data = conn.recv(1024)
            if data == 'exit':
                Flag = False
            elif data == '0':
                conn.sendall('通过可能会被录音.balabala一大推')
            else:
                conn.sendall('请重新输入.')

if __name__ == '__main__':
    server = SocketServer.ThreadingTCPServer(('127.0.0.1',8009),MyServer)
    server.serve_forever()
```



```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1',8009)
sk = socket.socket()
sk.connect(ip_port)
sk.settimeout(5)

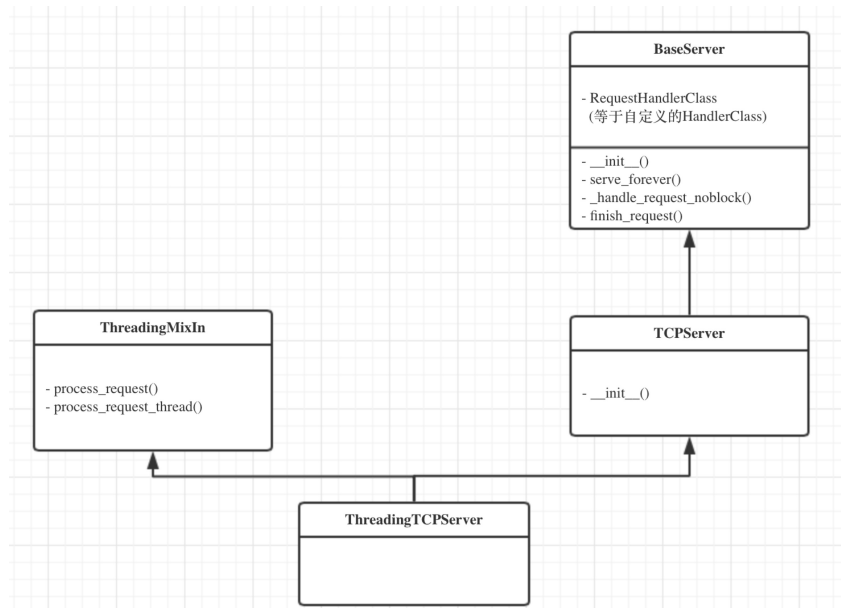
while True:
    data = sk.recv(1024)
    print 'receive:',data
    inp = raw_input('please input:')
    sk.sendall(inp)
    if inp == 'exit':
        break

sk.close()
```



## 2、ThreadingTCPServer源码剖析

ThreadingTCPServer的类图关系如下：



内部调用流程为：

- 启动服务端程序
- 执行 TCPServer.\_\_init\_\_ 方法，创建服务端Socket对象并绑定 IP 和 端口
- 执行 BaseServer.\_\_init\_\_ 方法，将自定义的继承自 SocketServer.BaseRequestHandler 的类 MyRequestHandle赋值给 **self.RequestHandlerClass**
- 执行 BaseServer.server\_forever 方法，While 循环一直监听是否有客户端请求到达 ...
- 当客户端连接到达服务器
- 执行 ThreadingMixIn.process\_request 方法，创建一个“线程”用来处理请求
- 执行 ThreadingMixIn.process\_request\_thread 方法

- 执行 BaseServer.finish\_request 方法，执行 `self.RequestHandlerClass()` 即：执行 自定义 MyRequestHandler 的构造方法（自动调用基类 BaseRequestHandler的构造方法，在该构造方法中又会调用 MyRequestHandler 的handle方法）

ThreadingTCPServer相关源码：

```

class BaseServer:

    """Base class for server classes.

    Methods for the caller:

    - __init__(server_address, RequestHandlerClass)
    - serve_forever(poll_interval=0.5)
    - shutdown()
    - handle_request() # if you do not use serve_forever()
    - fileno() -> int # for select()

    Methods that may be overridden:

    - server_bind()
    - server_activate()
    - get_request() -> request, client_address
    - handle_timeout()
    - verify_request(request, client_address)
    - server_close()
    - process_request(request, client_address)
    - shutdown_request(request)
    - close_request(request)
    - handle_error()

    Methods for derived classes:

    - finish_request(request, client_address)

    Class variables that may be overridden by derived classes or
    instances:

    - timeout
    - address_family
    - socket_type
    - allow_reuse_address

    Instance variables:

    - RequestHandlerClass
    - socket

    """

    timeout = None

    def __init__(self, server_address, RequestHandlerClass):
        """Constructor. May be extended, do not override."""
        self.server_address = server_address
        self.RequestHandlerClass = RequestHandlerClass
        self._is_shut_down = threading.Event()
        self._shutdown_request = False

    def server_activate(self):
        """Called by constructor to activate the server.

        May be overridden.

```

```

"""
pass

def serve_forever(self, poll_interval=0.5):
    """Handle one request at a time until shutdown.

    Polls for shutdown every poll_interval seconds. Ignores
    self.timeout. If you need to do periodic tasks, do them in
    another thread.
    """
    self.__is_shut_down.clear()
    try:
        while not self.__shutdown_request:
            # XXX: Consider using another file descriptor or
            # connecting to the socket to wake this up instead of
            # polling. Polling reduces our responsiveness to a
            # shutdown request and wastes cpu at all other times.
            r, w, e = _eintr_retry(select.select, [self], [], [],
                                    poll_interval)

            if self in r:
                self._handle_request_noblock()
        finally:
            self.__shutdown_request = False
            self.__is_shut_down.set()

def shutdown(self):
    """Stops the serve_forever loop.

    Blocks until the loop has finished. This must be called while
    serve_forever() is running in another thread, or it will
    deadlock.
    """
    self.__shutdown_request = True
    self.__is_shut_down.wait()

# The distinction between handling, getting, processing and
# finishing a request is fairly arbitrary. Remember:
#
# - handle_request() is the top-level call. It calls
#   select, get_request(), verify_request() and process_request()
# - get_request() is different for stream or datagram sockets
# - process_request() is the place that may fork a new process
#   or create a new thread to finish the request
# - finish_request() instantiates the request handler class;
#   this constructor will handle the request all by itself

def handle_request(self):
    """Handle one request, possibly blocking.

    Respects self.timeout.
    """
    # Support people who used socket.settimeout() to escape
    # handle_request before self.timeout was available.
    timeout = self.socket.gettimeout()
    if timeout is None:
        timeout = self.timeout
    elif self.timeout is not None:
        timeout = min(timeout, self.timeout)
    fd_sets = _eintr_retry(select.select, [self], [], [], timeout)
    if not fd_sets[0]:
        self.handle_timeout()
    return
    self._handle_request_noblock()

def _handle_request_noblock(self):

```

```

        """Handle one request, without blocking.

        I assume that select.select has returned that the socket is
        readable before this function was called, so there should be
        no risk of blocking in get_request().
        """
        try:
            request, client_address = self.get_request()
        except socket.error:
            return
        if self.verify_request(request, client_address):
            try:
                self.process_request(request, client_address)
            except:
                self.handle_error(request, client_address)
                self.shutdown_request(request)

    def handle_timeout(self):
        """Called if no new request arrives within self.timeout.

        Overridden by ForkingMixIn.
        """
        pass

    def verify_request(self, request, client_address):
        """Verify the request. May be overridden.

        Return True if we should proceed with this request.

        """
        return True

    def process_request(self, request, client_address):
        """Call finish_request.

        Overridden by ForkingMixIn and ThreadingMixIn.

        """
        self.finish_request(request, client_address)
        self.shutdown_request(request)

    def server_close(self):
        """Called to clean-up the server.

        May be overridden.

        """
        pass

    def finish_request(self, request, client_address):
        """Finish one request by instantiating RequestHandlerClass."""
        self.RequestHandlerClass(request, client_address, self)

    def shutdown_request(self, request):
        """Called to shutdown and close an individual request."""
        self.close_request(request)

    def close_request(self, request):
        """Called to clean up an individual request."""
        pass

    def handle_error(self, request, client_address):
        """Handle an error gracefully. May be overridden.

        The default is to print a traceback and continue.

```

```

"""

print '-'*40
print 'Exception happened during processing of request from',
print client_address
import traceback
traceback.print_exc() # XXX But this goes to stderr!
print '-'*40

```



```

class TCPServer(BaseServer):

    """Base class for various socket-based server classes.

    Defaults to synchronous IP stream (i.e., TCP).

    Methods for the caller:

    - __init__(server_address, RequestHandlerClass, bind_and_activate=True)
    - serve_forever(poll_interval=0.5)
    - shutdown()
    - handle_request() # if you don't use serve_forever()
    - fileno() -> int # for select()

    Methods that may be overridden:

    - server_bind()
    - server_activate()
    - get_request() -> request, client_address
    - handle_timeout()
    - verify_request(request, client_address)
    - process_request(request, client_address)
    - shutdown_request(request)
    - close_request(request)
    - handle_error()

    Methods for derived classes:

    - finish_request(request, client_address)

    Class variables that may be overridden by derived classes or
    instances:

    - timeout
    - address_family
    - socket_type
    - request_queue_size (only for stream sockets)
    - allow_reuse_address

    Instance variables:

    - server_address
    - RequestHandlerClass
    - socket

    """

    address_family = socket.AF_INET

    socket_type = socket.SOCK_STREAM

    request_queue_size = 5

```

```

allow_reuse_address = False

def __init__(self, server_address, RequestHandlerClass, bind_and_activate=True):
    """Constructor. May be extended, do not override."""
    BaseServer.__init__(self, server_address, RequestHandlerClass)
    self.socket = socket.socket(self.address_family,
                                self.socket_type)

    if bind_and_activate:
        try:
            self.server_bind()
            self.server_activate()
        except:
            self.server_close()
            raise

def server_bind(self):
    """Called by constructor to bind the socket.

    May be overridden.

    """
    if self.allow_reuse_address:
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    self.socket.bind(self.server_address)
    self.server_address = self.socket.getsockname()

def server_activate(self):
    """Called by constructor to activate the server.

    May be overridden.

    """
    self.socket.listen(self.request_queue_size)

def server_close(self):
    """Called to clean-up the server.

    May be overridden.

    """
    self.socket.close()

def fileno(self):
    """Return socket file number.

    Interface required by select().

    """
    return self.socket.fileno()

def get_request(self):
    """Get the request and client address from the socket.

    May be overridden.

    """
    return self.socket.accept()

def shutdown_request(self, request):
    """Called to shutdown and close an individual request."""
    try:
        #explicitly shutdown. socket.close() merely releases
        #the socket and waits for GC to perform the actual close.
        request.shutdown(socket.SHUT_WR)
    except socket.error:
        pass #some platforms may raise ENOTCONN here

```



```
self.close_request(request)
```

```
def close_request(self, request):
    """Called to clean up an individual request."""
    request.close()
```



```
class ThreadingMixIn:
    """Mix-in class to handle each request in a new thread."""

    # Decides how threads will act upon termination of the
    # main process
    daemon_threads = False

    def process_request_thread(self, request, client_address):
        """Same as in BaseServer but as a thread.

        In addition, exception handling is done here.

        """
        try:
            self.finish_request(request, client_address)
            self.shutdown_request(request)
        except:
            self.handle_error(request, client_address)
            self.shutdown_request(request)

    def process_request(self, request, client_address):
        """Start a new thread to process the request."""
        t = threading.Thread(target = self.process_request_thread,
                             args = (request, client_address))
        t.daemon = self.daemon_threads
        t.start()
```



```
class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass
```

### RequestHandler相关源码



```
class BaseRequestHandler:

    """Base class for request handler classes.

    This class is instantiated for each request to be handled. The
    constructor sets the instance variables request, client_address
    and server, and then calls the handle() method. To implement a
    specific service, all you need to do is to derive a class which
    defines a handle() method.

    The handle() method can find the request as self.request, the
    client address as self.client_address, and the server (in case it
    needs access to per-server information) as self.server. Since a
    separate instance is created for each request, the handle() method
    can define arbitrary other instance variables.

    """

    def __init__(self, request, client_address, server):
        self.request = request
```

```

        self.client_address = client_address
        self.server = server
        self.setup()
        try:
            self.handle()
        finally:
            self.finish()

    def setup(self):
        pass

    def handle(self):
        pass

    def finish(self):
        pass

```



实例:



```

#!/usr/bin/env python
# -*- coding:utf-8 -*-
import SocketServer

class MyServer(SocketServer.BaseRequestHandler):

    def handle(self):
        # print self.request,self.client_address,self.server
        conn = self.request
        conn.sendall('欢迎致电 10086, 请输入1xxx,0转人工服务.')
        Flag = True
        while Flag:
            data = conn.recv(1024)
            if data == 'exit':
                Flag = False
            elif data == '0':
                conn.sendall('通过可能会被录音.balabala一大推')
            else:
                conn.sendall('请重新输入.')

if __name__ == '__main__':
    server = SocketServer.ThreadingTCPServer(('127.0.0.1',8009),MyServer)
    server.serve_forever()

```



```

#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1',8009)
sk = socket.socket()
sk.connect(ip_port)
sk.settimeout(5)

while True:
    data = sk.recv(1024)
    print 'receive:',data

```

```
inp = raw_input('please input:')
sk.sendall(inp)
if inp == 'exit':
    break

sk.close()
```



源码精简：

```
import socket
import threading
import select

def process(request, client_address):
    print request, client_address
    conn = request
    conn.sendall('欢迎致电 10086, 请输入1xxx, 0转人工服务.')
    flag = True
    while flag:
        data = conn.recv(1024)
        if data == 'exit':
            flag = False
        elif data == '0':
            conn.sendall('通过可能会被录音.balabala一大推')
        else:
            conn.sendall('请重新输入.')

sk = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sk.bind(('127.0.0.1', 8002))
sk.listen(5)

while True:
    r, w, e = select.select([sk], [], [], 1)
    print 'looping'
    if sk in r:
        print 'get request'
        request, client_address = sk.accept()
        t = threading.Thread(target=process, args=(request, client_address))
        t.daemon = False
        t.start()

sk.close()
```



如精简代码可以看出，SocketServer的ThreadingTCPServer之所以可以同时处理请求得益于 **select** 和 **Threading** 两个东西，其实本质上就是在服务器端为每一个客户端创建一个线程，当前线程用来处理对应客户端的请求，所以，可以支持同时n个客户端链接（长连接）。

## ForkingTCPServer

ForkingTCPServer和ThreadingTCPServer的使用和执行流程基本一致，只不过在内部分别为请求者建立“线程”和“进程”。

基本使用：

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import SocketServer
```

```

class MyServer(SocketServer.BaseRequestHandler):

    def handle(self):
        # print self.request,self.client_address,self.server
        conn = self.request
        conn.sendall('欢迎致电 10086, 请输入1xxx,0转人工服务.')
        Flag = True
        while Flag:
            data = conn.recv(1024)
            if data == 'exit':
                Flag = False
            elif data == '0':
                conn.sendall('通过可能会被录音.balabala一大推')

            else:
                conn.sendall('请重新输入.')

if __name__ == '__main__':
    server = SocketServer.ForkingTCPServer(('127.0.0.1',8009),MyServer)
    server.serve_forever()

```



```

#!/usr/bin/env python
# -*- coding:utf-8 -*-

import socket

ip_port = ('127.0.0.1',8009)
sk = socket.socket()
sk.connect(ip_port)
sk.settimeout(5)

while True:
    data = sk.recv(1024)
    print 'receive:',data
    inp = raw_input('please input:')
    sk.sendall(inp)
    if inp == 'exit':
        break

sk.close()

```



以上ForkingTCPServer只是将 ThreadingTCPServer 实例中的代码:

```

1 server = SocketServer.ThreadingTCPServer(('127.0.0.1',8009),MyRequestHandle
2 变更为:
3 server = SocketServer.ForkingTCPServer(('127.0.0.1',8009),MyRequestHandler)

```



SocketServer的ThreadingTCPServer之所以可以同时处理请求得益于 **select** 和 **os.fork** 两个东西, 其实本质上就是在服务器端为每一个客户端创建一个进程, 当前新建的进程用来处理对应客户端的请求, 所以, 可以支持同时n个客户端链接(长连接)。

源码剖析参考 ThreadingTCPServer

Twisted

Twisted是一个事件驱动的网络框架，其中包含了诸多功能，例如：网络协议、线程、数据库管理、网络操作、电子邮件等。

Package	application	Configuration objects for Twisted Applications.
Package	conch	Twisted Conch: The Twisted Shell. Terminal emulation, SSHv2 and telnet.
Module	copyright	Copyright information for Twisted.
Package	cred	Twisted Cred: Support for verifying credentials, and providing services to user based on those credentials.
Package	enterprise	Twisted Enterprise: Database support for Twisted services.
Package	internet	Twisted Internet: Asynchronous I/O and Events.
Package	logger	Twisted Logger: Classes and functions to do granular logging.
Package	mail	Twisted Mail: Servers and clients for POP3, ESMTP, and IMAP.
Package	manhole	Twisted Manhole: interactive interpreter and direct manipulation support for Twisted.
Package	names	Twisted Names: DNS server and client implementations.
Package	news	Twisted News: A NNTP-based news service.
Package	pair	Twisted Pair: The framework of your ethernet.
Package	persisted	Twisted Persisted: Utilities for managing persistence.
Module	plugin	Plugin system for Twisted.
Package	plugins	Plugins for services implemented in Twisted.
Package	positioning	Twisted Positioning: Framework for applications that make use of positioning.
Package	protocols	Twisted Protocols: A collection of internet protocol implementations.
Package	python	Twisted Python: Utilities and Enhancements for Python.
Package	runner	Twisted Runner: Run and monitor processes.
Package	scripts	Subpackage containing the modules that implement the command line tools.
Package	spread	Twisted Spread: Spreadable (Distributed) Computing.
Package	tap	Twisted TAP: Twisted Application Persistence builders for other Twisted servers.
Package	test	Twisted's unit tests.
Package	trial	Twisted Trial: Asynchronous unit testing framework.
Package	web	Twisted Web: HTTP clients and servers, plus tools for implementing them.
Package	words	Twisted Words: Client and server implementations for IRC, XMPP, and other chat services.

事件驱动

简而言之，事件驱动分为二个部分：第一，注册事件；第二，触发事件。

自定义事件驱动框架，命名为：“弑君者”：



```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

# event_drive.py

event_list = []

def run():
    for event in event_list:
        obj = event()
        obj.execute()


class BaseHandler(object):

    """
    用户必须继承该类，从而规范所有类的方法（类似于接口的功能）
    """

    def execute(self):
        raise Exception('you must overwrite execute')
```




程序员使用“弑君者框架”：



```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

from source import event_drive

class MyHandler(event_drive.BaseHandler):
```

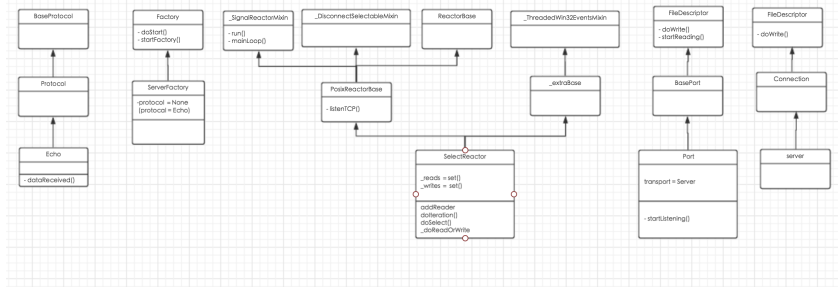


```
def execute(self):  
    print 'event-drive execute MyHandler'  
  
event_drive.event_list.append(MyHandler)  
event_drive.run()
```

如上述代码，事件驱动只不过是框架规定了执行顺序，程序员在使用框架时，可以向原执行顺序中注册“事件”，从而在框架执行时可以出发已注册的“事件”。

## 基于事件驱动Socket

```
1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3
4  from twisted.internet import protocol
5  from twisted.internet import reactor
6
7  class Echo(protocol.Protocol):
8      def dataReceived(self, data):
9          self.transport.write(data)
10
11  def main():
12      factory = protocol.ServerFactory()
13      factory.protocol = Echo
14
15      reactor.listenTCP(8000, factory)
16      reactor.run()
17
18  if __name__ == '__main__':
19      main()
```



程序执行流程：

- 运行服务端程序
- 创建Protocol的派生类Echo
- 创建ServerFactory对象，并将Echo类封装到其protocol字段中
- 执行reactor的 listenTCP 方法，内部使用 tcp.Port 创建socket server对象，并将该对象添加到了 reactor的set类型的字段 \_read 中
- 执行reactor的 run 方法，内部执行 while 循环，并通过 select 来监视 \_read 中文件描述符是否有变化，循环中...
- 客户端请求到达
- 执行reactor的 \_doReadOrWrite 方法，其内部通过反射调用 tcp.Port 类的 doRead 方法，内部 accept 客户端连接并创建Server对象实例（用于封装客户端socket信息）和 创建 Echo 对象实例（用于处理请求），然后调用 Echo 对象实例的 makeConnection 方法，创建连接。
- 执行 tcp.Server 类的 doRead 方法，读取数据，
- 执行 tcp.Server 类的 \_dataReceived 方法，如果读取数据内容为空（关闭链接），否则，出发 Echo 的 dataReceived 方法
- 执行 Echo 的 dataReceived 方法

从源码可以看出，上述实例本质上使用了事件驱动的方法 和 IO多路复用的机制来进行Socket的处理。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

from twisted.internet import reactor, protocol
from twisted.web.client import getPage
from twisted.internet import reactor
import time

class Echo(protocol.Protocol):

    def dataReceived(self, data):
        deferred1 = getPage('http://cnblogs.com')
        deferred1.addCallback(self.printContents)

        deferred2 = getPage('http://baidu.com')
        deferred2.addCallback(self.printContents)

        for i in range(2):
            time.sleep(1)
            print 'execute ',i

    def execute(self,data):
        self.transport.write(data)

    def printContents(self,content):
        print len(content),content[0:100],time.time()

def main():

    factory = protocol.ServerFactory()
    factory.protocol = Echo

    reactor.listenTCP(8000,factory)
    reactor.run()

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

更多请见：

<https://twistedmatrix.com/trac>

<http://twistedmatrix.com/documents/current/api/>



作者：武沛齐

出处：<http://www.cnblogs.com/wupeiqi/>

本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接。

好文要顶

关注我

收藏该文



武沛齐

关注 - 44

粉丝 - 9942

11

0

[+加关注](#)

posted @ 2015-12-12 10:02 武沛齐 阅读(36712) 评论(6) 编辑 收藏

评论列表

- #1楼 2017-08-17 19:21 benjamin杨

好文

回复 引用

支持(0) 反对(0)
- #2楼 2017-08-30 23:04 六神酱

select伪同步真是巧妙

回复 引用

支持(0) 反对(0)
- #3楼 2017-09-13 14:09 seve\_Y

ConnectionRefusedError: [WinError 10061] 由于目标计算机积极拒绝，无法连接。  
按照你的方法学习，出现了这个错误，如何解决，谢谢

回复 引用

支持(0) 反对(0)
- #4楼 2017-09-27 22:45 那是谁的领地

@ seve\_Y  
应该是win的问题Linux试试

回复 引用

支持(0) 反对(0)
- #5楼 2017-10-25 16:48 六神酱

"不要回答,不要回答,不要回答" -----三体

回复 引用

支持(0) 反对(0)
- #6楼 2017-12-25 17:05 mihon

我就要回答





回复 引用

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

发表评论

编辑 预览

B    

支持 Markdown

[提交评论](#) [退出](#) [订阅评论](#)

[Ctrl+Enter快捷键提交]

- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】有道智云周年庆，API服务大放送，注册即送100元体验金！
- 【推荐】30+视频&10+案例纵横文件与IO领域 | Java开发者高级应用站





#### 相关博文：

- python之路 socket、socket server
  - Python之路（第六篇）
  - python之路第六篇
  - python之路 socket、socket server
  - Python之路【第六篇】：socket
- » 更多推荐...

#### 最新 IT 新闻：

- 这家搞出「手指不离开屏幕」比赛的公司，背后的鬼点子还真不少
  - 理想汽车今晚登陆纳斯达克：认购火爆 提前一天上市
  - 千万级大V独家合作，西瓜视频用什么吸引创作人？
  - 雷军B站第一支视频出炉：网友弹幕狂刷“Are You OK”
  - 危险迫在眉睫！马斯克预警AI可能5年内超越人类
- » 更多新闻...