**网络概述、udp**

**<1>网络通信概述**

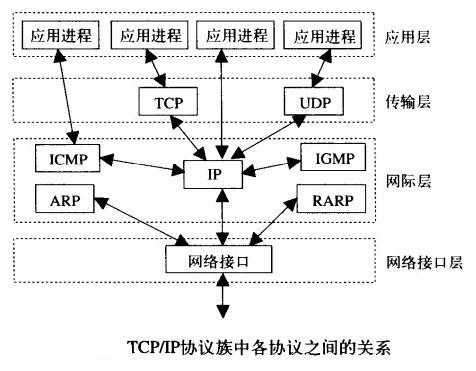
**1、**使用网络能够把多方链接在一起，然后可以进行数据传递 2、所谓的网络编程就是，让在不同的电脑上的软件能够进行数据传递，即进程之间的通信

**<2>tcp/ip简介**

**TCP/IP协议(族)**

为了把全世界的所有不同类型的计算机都连接起来，就必 须规定一套全球通用的协议，为了实现互联网这个目标，互联 网协议簇（Internet Protocol Suite）就是通用协议标准。

因为互联网协议包含了上百种协议标准，但是最重要的两 个协议是TCP和IP协议，所以，大家把互联网的协议简称 TCP/IP协议



**<3>端口**

**1. 什么是端口**

端口就好一个房子的门，是出入这间房子的必经之路。如 果一个进程需要收发网络数据，那么就需要有这样的端口在 linux系统中，端口可以有65536（2的16次方）个之多！既 然有这么多，操作系统为了统一管理，所以进行了编号，这就 是端口号。

**2. 端口号**

端口是通过端口号来标记的，端口号只有整数，范围是从 0到65535

**3. 端口是怎样分配的**

端口号不是随意使用的，而是按照一定的规定进行分配。 端口的分类标准有好几种，我们这里不做详细讲解，只介绍一 下知名端口和动态端口。

**3.1 知名端口（Well Known Ports）**

知名端口是众所周知的端口号，范围从0到1023

80端口分配给HTTP服务

21端口分配给FTP服务

一般情况下，如果一个程序需要使用知名端口的需要有root权限

**3.2 动态端口（Dynamic Ports）**

动态端口的范围是从1024到65535之所以称为动态端 口，是因为它一般不固定分配某种服务，而是动态分配。 动态分配是指当一个系统进程或应用程序进程需要网络 通信时，它向主机申请一个端口，主机从可用的端口号中 分配一个供它使用。当这个进程关闭时，同时也就释放了 所占用的端口号。

**3.3 怎样查看端口 ？**

用“netstat －an”查看端口状态

**4. 小总结**

端口有什么用呢 ？ 我们知道，一台拥有IP地址的主机 可以提供许多服务，比如HTTP（万维网服务）、FTP（文件传 输）、SMTP（电子邮件）等，这些服务完全可以通过1个IP地 址来实现。那么，主机是怎样区分不同的网络服务呢？显然不 能只靠IP地址，因为IP地址与网络服务的关系是一对多的 关系。实际上是通过“IP地址+端口号”来区分不同的服务的。 需要注意的是，端口并不是一一对应的。比如你的电脑作为客 户机访问一台WWW服务器时，WWW服务器使用“80”端口与你的 电脑通信，但你的电脑则可能使用“3457”这样的端口。

**<4>IP地址**

**1. ip地址的作用**

ip地址：用来在网络中标记一台电脑的一串数字，比如 192.168.1.1；在本地局域网上是惟一的。

**2. ip地址的分类**

每一个IP地址包括两部分：网络地址和主机地址



**1>A类IP地址**

一个A类IP地址由1字节的网络地址和3字节主机地 址组成，网络地址的最高位必须是“0”，地址范围1.0.0.1- 126.255.255.254二进制表示为：00000001 00000000 00000000 00000001 - 01111110 11111111 11111111 11111110可用的A类网络有126个，每个网络能容纳 1677214个主机

**2>B类IP地址**

一个B类IP地址由2个字节的网络地址和2个字节的 主机地址组成，网络地址的最高位必须是“10”，地址范围 128.1.0.1-191.255.255.254二进制表示为：10000000 00000001 00000000 00000001 - 10111111 11111111 11111111 11111110可用的B类网络有16384个，每个网 络能容纳65534主机

**3>C类IP地址**

一个C类IP地址由3字节的网络地址和1字节的主机 地址组成，网络地址的最高位必须是“110”范围 192.0.1.1-223.255.255.254二进制表示为: 11000000 00000000 00000001 00000001 - 11011111 11111111 11111110 11111110C类网络可达2097152个，每个网络 能容纳254个主机

**4>D类地址用于多点广播**

D类IP地址第一个字节以“1110”开始，它是一个专门 保留的地址。它并不指向特定的网络，目前这一类地址被 用在多点广播（Multicast）中多点广播地址用来一次寻址 一组计算机地址范围224.0.0.1-239.255.255.254

**5>E类IP地址**

以“1111”开始，为将来使用保留,仅作实 验和开发用

**6>私有ip**

**在这么多网络IP中，国际规定有一部分IP地址是用 于我们的局域网使用，也就是属于私网IP，不在公网中使 用的，它们的范围是：**

10.0.0.0～10.255.255.255

172.16.0.0～172.31.255.255

192.168.0.0～192.168.255.255

**7>注意**

**IP地址127．0．0．1~127．255．255．255用于回路 测试，如：127.0.0.1可以代表本机IP地址，用 http://127.0.0.1就可以测试本机中配置的Web服务器。**

**<5>子网掩码**

它必须结合IP地址一起使用,子网掩码只有一个作用，就是将某个IP地址划分成网络地址和主机地址两部分,子网掩码的设定必须遵循一定的规则。

与IP地址相同，子网掩码的长度也是32位，

左边是网络位，用二进制数字“1”表示；

右边是主机位，用二进制数字“0”表示。

子网掩码是“255.255.255.0”的网络：

最后面一个数字可以在0~255范围内任意变化，因此可以提供 256个IP地址。

但是实际可用的IP地址数量是256-2，即254个，因为主机号 不能全是“0”或全是“1”。

主机号全为0，表示网络号

主机号全为1，表示网络广播

**假设IP地址为“192.168.1.1”子网掩码为“255.255.255.0”**

其中，“1”有24个，代表与此相对应的IP地址左边24位是网络号；

“0”有8个，代表与此相对应的IP地址右边8位是主机号。

这样，子网掩码就确定了一个IP地址的32位二进制数字中哪些是网络号、哪些是主机号。

这对于采用TCP/IP协议的网络来说非常重要，只有通过子网掩码，才能表明一台主机所在的子网与其他子网的关系，使网络正常工作。

**<6>socket简介**

**1.本地的进程间通信（IPC）有很多种方式**

队列

同步（互斥锁、条件变量等）

以上通信方式都是在一台机器上不同进程之间的通信方式

**2. 网络中进程之间如何通信**

其实TCP/IP协议族已经帮我们解决了这个问题，网络层 的“ip地址”可以唯一标识网络中的主机，而传输层的“协议+ 端口”可以唯一标识主机中的应用程序（进程）。

这样利用ip地址，协议，端口就可以标识网络的进程了， 网络中的进程通信就可以利用这个标志与其它进程进行交互

**3. 什么是socket**

socket(简称 套接字) 是进程间通信的一种方式，它与其 他进程间通信的一个主要不同是：

它能实现不同主机间的进程间通信，我们网络上各种各样 的服务大多都是基于 Socket 来完成通信的

例如我们每天浏览网页、QQ 聊天、收发 email 等等

**4. 创建socket**

在 Python 中 使用socket 模块的函数 socket 就可以 完成：

socket.socket(AddressFamily, Type)

**说明：**

函数 socket.socket 创建一个 socket，返回该 socket 的描述符，该函数带有两个参数：

Address Family：可以选择 AF\_INET（用于 Internet 进 程间通信） 或者 AF\_UNIX（用于同一台机器进程间通信）, 实际工作中常用AF\_INET

Type：套接字类型，可以是 SOCK\_STREAM（流式套接字， 主要用于 TCP 协议）或者 SOCK\_DGRAM（数据报套接字，主 要用于 UDP 协议）

**创建一个tcp socket（tcp套接字）**

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

print'Socket Created'

**创建一个udp socket（udp套接字）**

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

print'Socket Created'

**<7>UDP介绍**

UDP是一种面向无连接的协议，每个数据报都是一个独立的信息，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

**UDP特点：**

UDP是面向无连接的通讯协议，UDP数据包括目的端口号 和源端口号信息，由于通讯不需要连接，所以可以实现广播发 送。UDP传输数据时有大小限制，每个被传输的数据报必须 限定在64KB之内。UDP是一个不可靠的协议，发送方所发送 的数据报并不一定以相同的次序到达接收方。

【适用情况】

UDP是面向消息的协议，通信时不需要建立连接，数据的 传输自然是不可靠的，UDP一般用于多点通信和实时的数据业 务，比如：

1、语音广播

2、视频

3、QQ

4、TFTP(简单文件传送）

5、SNMP（简单网络管理协议）

6、RIP（路由信息协议，如报告股票市场，航空信息）

7、DNS(域名解释）

注重速度流畅

UDP操作简单，而且仅需要较少的监护，因此通常用于局域网高可靠性的分散系统中client/server应用程序。

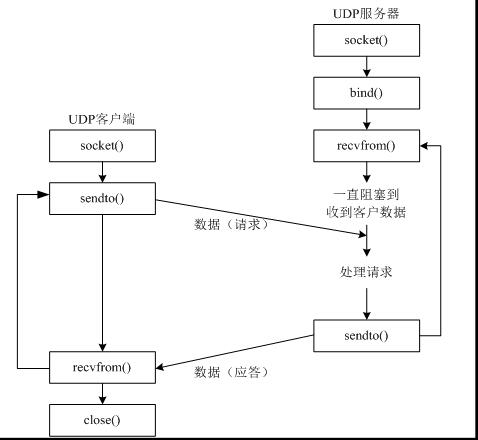
**<8>udp网络程序-发送数据**

创建一个udp客户端程序的流程是简单，具体步骤如下：

1、创建客户端套接字

2、发送/接收数据

3、关闭套接字



from socket import \*

#1. 创建套接字

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

#2. 准备接收方的地址

sendAddr = ('192.168.1.103', 8080)

#3. 从键盘获取数据

sendData = input("请输入要发送的数据:")

#4. 发送数据到指定的电脑上

udpSocket.sendto(sendData, sendAddr)

#5. 关闭套接字

udpSocket.close()

**<9>udp网络程序-发送、接收数据**

from socket import \*

#1. 创建套接字

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

#2. 准备接收方的地址

sendAddr = ('192.168.1.103', 8080)

#3. 从键盘获取数据

sendData = input("请输入要发送的数据:")

#4. 发送数据到指定的电脑上

udpSocket.sendto(sendData, sendAddr)

#5. 等待接收对方发送的数据

recvData = udpSocket.recvfrom(1024) # 1024表示本次接收 的最大字节数

#6. 显示对方发送的数据

print(recvData)

#7. 关闭套接字

udpSocket.close()

**<10>udp网络程序-端口问题**

当重新运行程序时，如果没有确定到底用哪个端口号，系统默认会随机分配端口。这个网络程序在运行的过程中，这个就唯一标识这个程序，所以如果其他电脑上的网络程序如果想要向此程序发送数据，那么就需要向这个数字（即端口）标识的程序发送即可。

**<11>udp绑定信息**

一般情况下，在一天电脑上运行的网络程序有很多，而各自用的端口号很多情况下不知道，为了不与其他的网络程序占用同一个端口号，往往在编程中，udp的端口号一般不绑定，但是如果需要做成一个服务器端的程序的话，是需要绑定的。

from socket import \*

#1. 创建套接字

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

#2. 绑定本地的相关信息，如果一个网络程序不绑定，则系统会随 机分配

bindAddr = ('', 7788) # ip地址和端口号，ip一般不用写，表 示本机的任何一个ip

udpSocket.bind(bindAddr)

#3. 等待接收对方发送的数据

recvData = udpSocket.recvfrom(1024) # 1024表示本次接收 的最大字节数

#4. 显示接收到的数据

print(recvData)

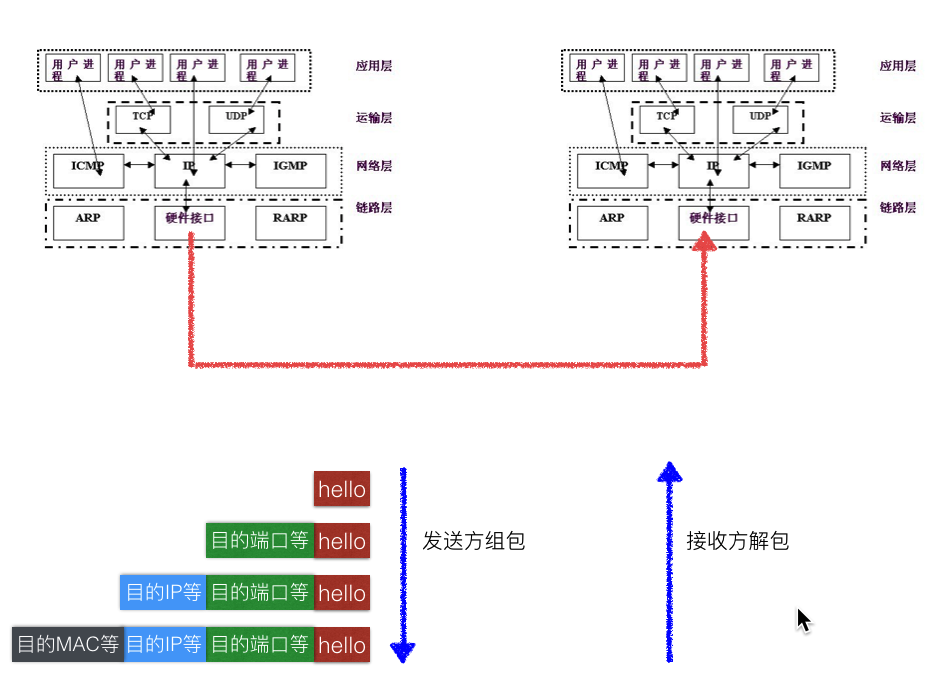
#5. 关闭套接字

udpSocket.close()

一个udp网络程序，可以不绑定，此时操作系统会随机进 行分配一个端口，如果重新运行次程序端口可能会发生变化

一个udp网络程序，也可以绑定信息（ip地址，端口号）， 如果绑定成功，那么操作系统用这个端口号来进行区别收到 的网络数据是否是此进程的

**<12>udp网络通信过程**



**<13>udp应用：echo服务器**

from socket import \*

#1. 创建套接字

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

#2. 绑定本地的相关信息

bindAddr = ('', 7788) # ip地址和端口号，ip一般不用写，表 示本机的任何一个ip

udpSocket.bind(bindAddr)

num = 1

whileTrue:

#3. 等待接收对方发送的数据

recvData = udpSocket.recvfrom(1024) # 1024表示本次 接收的最大字节数

#4. 将接收到的数据再发送给对方

udpSocket.sendto(recvData[0], recvData[1])

#5. 统计信息

print('已经将接收到的第%d个数据返回给对方,内容 为:%s'%(num,recvData[0]))

num+=1

#5. 关闭套接字

udpSocket.close()

**<14>udp应用：聊天室**

from socket import \*

from time import ctime

#1. 创建套接字

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

#2. 绑定本地的相关信息

bindAddr = ('', 7788) # ip地址和端口号，ip一般不用写，表 示本机的任何一个ip

udpSocket.bind(bindAddr)

whileTrue:

#3. 等待接收对方发送的数据

recvData = udpSocket.recvfrom(1024) # 1024表示本次 接收的最大字节数

#4. 打印信息

print('【%s】%s:%s'%(ctime(),recvData[1][0],recvData[0]))

#5. 关闭套接字

udpSocket.close()

**<15>udp总结**

**1. udp是TCP/IP协议族中的一种协议能够完成不同机器上的程序间的数据通信**

**2. udp服务器、客户端**

udp的服务器和客户端的区分：往往是通过请求服务和提 供服务来进行区分

请求服务的一方称为：客户端

提供服务的一方称为：服务器

**3. udp绑定问题**

一般情况下，服务器端，需要绑定端口，目的是为了让 其他的客户端能够正确发送到此进程

客户端，一般不需要绑定，而是让操作系统随机分配，这 样就不会因为需要绑定的端口被占用而导致程序无法运行的 情况

**TFTP项目、TCP编程**

**<1> wireshark抓包工具使用**

1. 安装wireshark

2. wireshark的使用

**<2> TFTP下载演示**

**<3> 应用：TFTP客户端**

1. TFTP协议介绍

TFTP（Trivial File Transfer Protocol,简单文件传输协议）

是TCP/IP协议族中的一个用来在客户端与服务器之间进行简单文件传输的协议

特点：

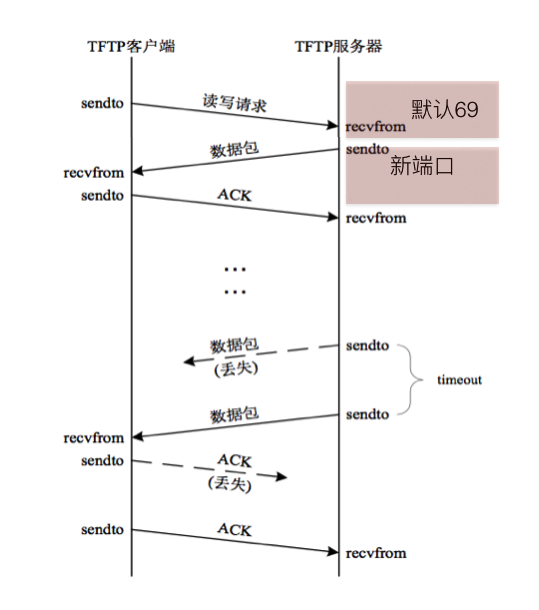
1. 简单
2. 占用资源小
3. 适合传递小文件
4. 适合在局域网进行传递
5. 端口号为69
6. 基于UDP实现

2. TFTP下载过程

TFTP服务器默认监听69号端口

当客户端发送“下载”请求（即读请求）时，需要向服务器的69端口发送

服务器若批准此请求,则使用一个新的、临时的 端口进行数据传输



当服务器找到需要现在的文件后，会立刻打开文件，把文件中的数据通过TFTP协议发送给客户端

如果文件的总大小较大（比如3M），那么服务器分多次发送，每次会从文件中读取512个字节的数据发送过来

因为发送的次数有可能会很多，所以为了让客户端对接收到的数据进行排序，所以在服务器发送那512个字节数据的时候，会多发2个字节的数据，用来存放序号，并且放在512个字节数据的前面，序号是从1开始的

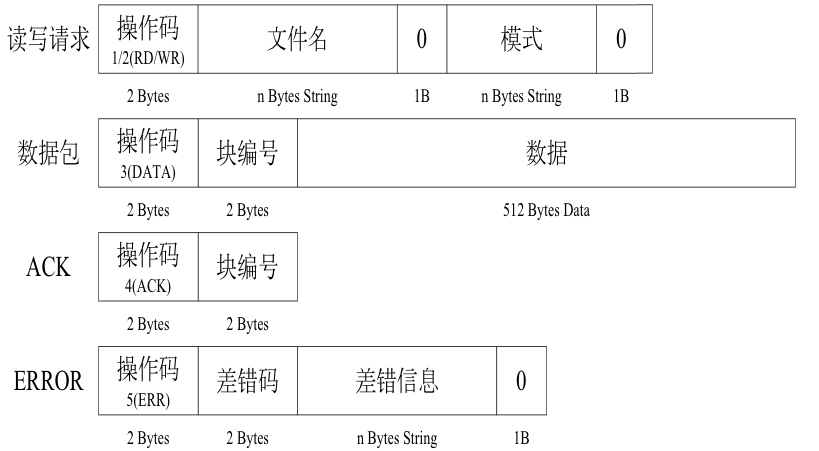
因为需要从服务器上下载文件时，文件可能不存在，那么此时服务器就会发送一个错误的信息过来，为了区分服务发送的是文件内容还是错误的提示信息，所以又用了2个字节 来表示这个数据包的功能（称为操作码），并且在序号的前面

| **操作码** | **功能** |
| --- | --- |
| 1 | 读请求，即下载 |
| 2 | 写请求，即上传 |
| 3 | 表示数据包，即DATA |
| 4 | 确认码，即ACK |
| 5 | 错误 |

因为udp的数据包不安全，即发送方发送是否成功不能确定，所以TFTP协议中规定，为了让服务器知道客户端已经接收到了刚刚发送的那个数据包，所以当客户端接收到一个数据包的时候需要向服务器进行发送确认信息，即发送收到了，这样的包成为ACK(应答包)

为了标记数据已经发送完毕，所以规定，当客户端接收到的数据小于516（2字节操作码+2个字节的序号+512字节数据）时，就意味着服务器发送完毕了

TFTP数据包的格式如下:



参考代码如下：

from socket import \*

import struct

import sys

if len(sys.argv) != 2:

print('-'\*30)

print("tips:")

print("python xxxx.py 192.168.1.1")

print('-'\*30)

exit()

else:

ip = sys.argv[1]

# 创建udp套接字

udpSocket = socket(AF\_INET, SOCK\_DGRAM)

#构造下载请求数据

cmd\_buf = struct.pack("!H8sb5sb",1,"test.jpg",0,"octet",0)

#发送下载文件请求数据到指定服务器

sendAddr = (ip, 69)

udpSocket.sendto(cmd\_buf, sendAddr)

p\_num = 0

recvFile = ''

while True:

recvData,recvAddr = udpSocket.recvfrom(1024)

recvDataLen = len(recvData)

# print recvAddr # for test

# print len(recvData) # for test

cmdTuple = struct.unpack("!HH", recvData[:4])

# print cmdTuple # for test

cmd = cmdTuple[0]

currentPackNum = cmdTuple[1]

if cmd == 3: #是否为数据包

# 如果是第一次接收到数据，那么就创建文件

if currentPackNum == 1:

recvFile = open("test.jpg", "a")

# 包编号是否和上次相等

if p\_num+1 == currentPackNum:

recvFile.write(recvData[4:]);

p\_num +=1

print '(%d)次接收到的数据'%(p\_num)

ackBuf = struct.pack("!HH",4,p\_num)

udpSocket.sendto(ackBuf, recvAddr)

# 如果收到的数据小于516则认为出错

if recvDataLen<516:

recvFile.close()

print '已经成功下载！！！'

break

elif cmd == 5: #是否为错误应答

print "error num:%d"%currentPackNum

break

udpSocket.close()

**<4> udp广播**

网络编程中的广播

import socket, sys

dest = ('<broadcast>', 7788)

# 创建udp套接字

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

# 对这个需要发送广播数据的套接字进行修改设置，否则不能发送广播数据

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_BROADCAST,1)

# 以广播的形式发送数据到本网络的所有电脑中

s.sendto("Hi", dest)

print "等待对方回复（按ctrl+c退出）"

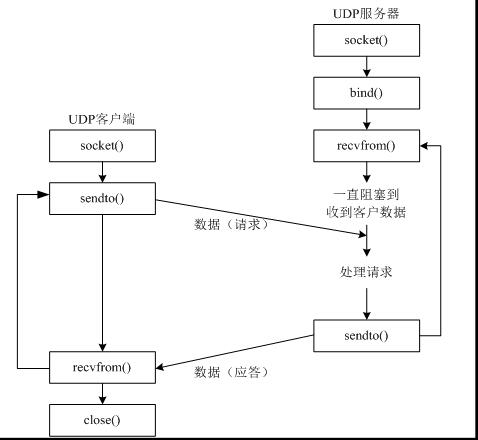
while True:

(buf, address) = s.recvfrom(2048)

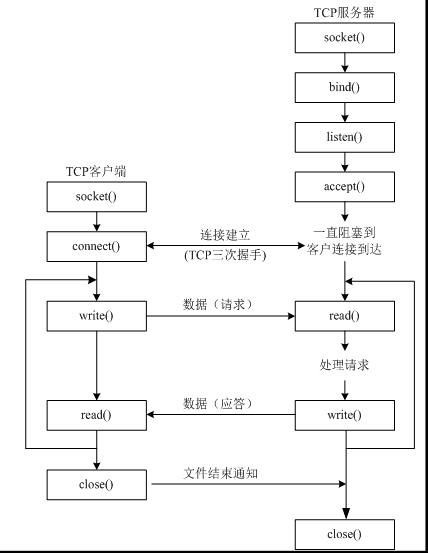
print "Received from %s: %s" % (address, buf)

**<5> tcp相关介绍**

udp通信模型中，在通信开始之前，不需要建立相关的链接，只需要发送数据即可，类似于生活中，"写信"



tcp通信模型中，在通信开始之前，一定要先建立相关的链接，才能发送数据，类似于生活中，"打电话"



**<6> tcp服务器（类似生活中的电话机）**

1. 买个手机
2. 插上手机卡
3. 设计手机为正常接听状态（即能够响铃）
4. 静静的等着别人拨打

**tcp服务器**

要完成一个tcp服务器的功能，需要的流程如下：

1. socket创建一个套接字
2. bind绑定ip和port
3. listen使套接字变为可以被动链接
4. accept等待客户端的链接
5. recv/send接收发送数据

from socket import \*

# 创建socket

tcpSerSocket = socket(AF\_INET, SOCK\_STREAM)

# 绑定本地信息

address = ('', 7788)

tcpSerSocket.bind(address)

# 使用socket创建的套接字默认的属性是主动的，使用listen将其变为被动的，这样就可以接收别人的链接了

tcpSerSocket.listen(5)

# 如果有新的客户端来链接服务器，那么就产生一个新的套接字专门为这个客户端服务器

# newSocket用来为这个客户端服务

# tcpSerSocket就可以省下来专门等待其他新客户端的链接

newSocket, clientAddr = tcpSerSocket.accept()

# 接收对方发送过来的数据，最大接收1024个字节

recvData = newSocket.recv(1024)

print '接收到的数据为:',recvData

# 发送一些数据到客户端

newSocket.send("thank you !")

# 关闭为这个客户端服务的套接字，只要关闭了，就意味着为不能再为这个客户端服务了，如果还需要服务，只能再次重新连接

newSocket.close()

# 关闭监听套接字，只要这个套接字关闭了，就意味着整个程序不能再接收任何新的客户端的连接

tcpSerSocket.close()

**<7> tcp客户端**

**tcp客户端构建流程**

from socket import \*

# 创建socket

tcpClientSocket = socket(AF\_INET, SOCK\_STREAM)

# 链接服务器

serAddr = ('192.168.1.102', 7788)

tcpClientSocket.connect(serAddr)

# 提示用户输入数据

sendData = raw\_input("请输入要发送的数据：")

tcpClientSocket.send(sendData)

# 接收对方发送过来的数据，最大接收1024个字节

recvData = tcpClientSocket.recv(1024)

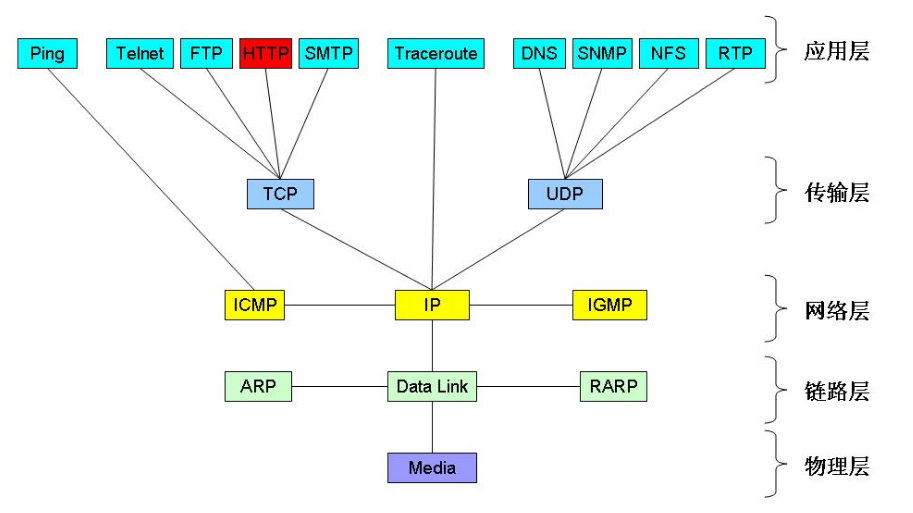
print '接收到的数据为:',recvData

# 关闭套接字

tcpClientSocket.close()

**网络通信过程详解**

**<**[**1> Packet Tracer网络通信过程**](file:///C:\Users\CPIC\Desktop\Python\Python%E6%A0%B8%E5%BF%83%E7%BC%96%E7%A8%8B\%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B\html%E7%89%88\%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B-%E5%85%A8%E9%83%A8%E8%AF%BE%E4%BB%B6\03day\section.1.1.html)



**<2> 通过集线器组网(hub)**

1、hub（集线器）能够完成多个电脑的链接

2、每个数据包的发送都是以广播的形式进行的，容易堵塞网络

**<3>** **通过交换机组网**

**交换机的作用：**

转发过滤：当一个数据帧的目的地址在MAC地址表中有映射时，它被转发到连接目的节点的端口而不是所有端口（如该数据帧为广播帧则转发至所有端口）

学习功能：以太网交换机了解每一端口相连设备的MAC地址，并将地址同相应的端口映射起来存放在交换机缓存中的MAC地址表中

**总结:**

1、交换机能够完成多个电脑的链接

2、每个数据包的发送都是以广播的形式进行的，容易堵塞网络

3、如果PC不知目标IP所对应的的MAC，那么可以看出，pc会先发送arp广播，得到对方的MAC然后，在进行数据的传送

4、当switch第一次收到arp广播数据，会把arp广播数据包转发给所有端口（除来源端口）；如果以后还有pc询问此IP的MAC，那么只是向目标的端口进行转发数据

**<4>** **通过路由器组网**

**总结**

1、不在同一网段的pc，需要设置默认网关才能把数据传送过去通常情况下，都会把路由器默认网关

2、当路由器收到一个其它网段的数据包时，会根据“路由表”来决定，把此数据包发送到哪个端口；路由表的设定有静态和动态方法

3、每经过一次路由器，那么TTL值就会减一

**<5> 交换机、路由器、服务器组网**

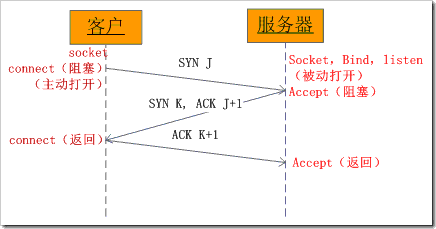
**总结**

1、DNS服务器用来解析出IP（类似电话簿）

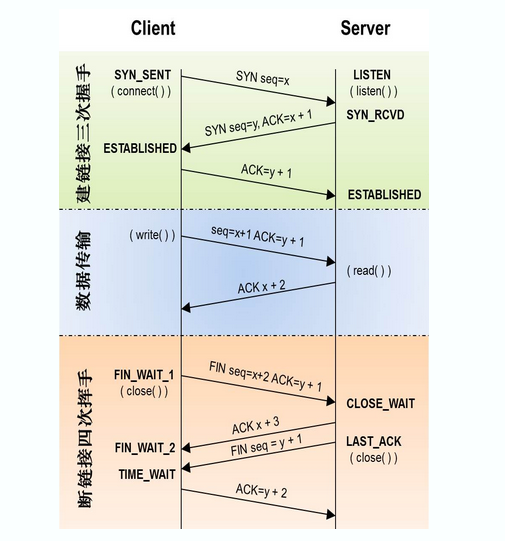
2、DFGATEWAY（默认网关）用来对顶，当发送的数据包的目的ip不是当前网络时，此数据包转发的目的ip

3、在路由器中路由表指定数据包的”下一跳”的地址

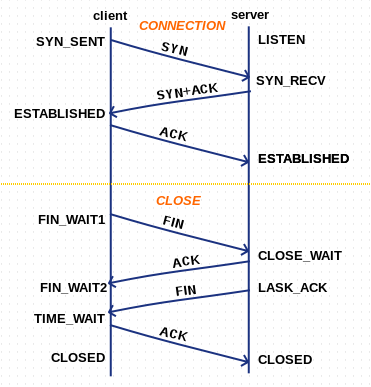
**<6>** **tcp三次握手**



**<7>** **tcp四次挥手**



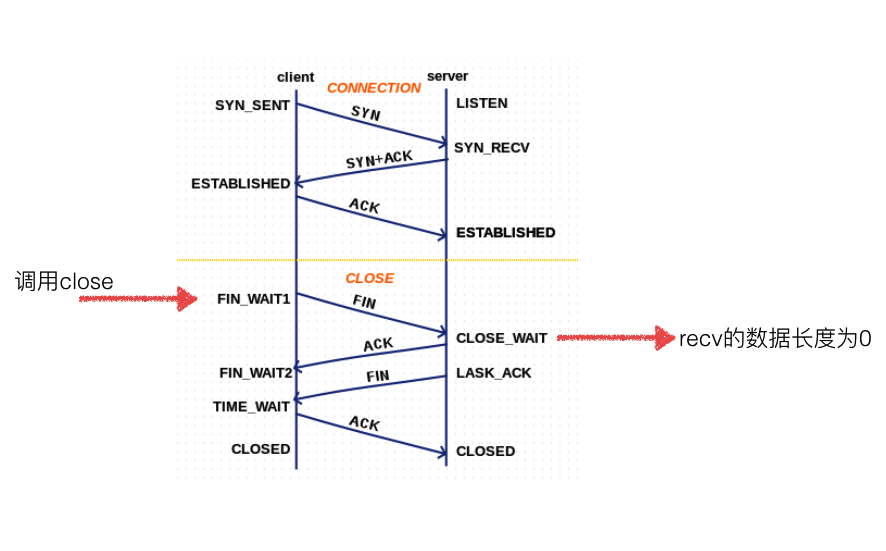
<8> **tcp十种状态**



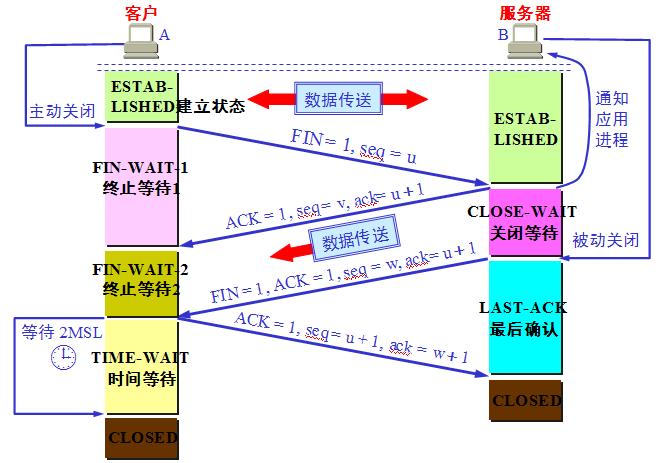
**注意:**

1、当一端收到一个FIN，内核让read返回0来通知应用层另一端已经终止了向本端的数据传送

2、发送FIN通常是应用层对socket进行关闭的结果



**<9>** **tcp的2MSL问题**



2MSL即两倍的MSL，TCP的TIME\_WAIT状态也称为2MSL等待状态，

当TCP的一端发起主动关闭，在发出最后一个ACK包后，

即第3次握 手完成后发送了第四次握手的ACK包后就进入了TIME\_WAIT状态，

必须在此状态上停留两倍的MSL时间，

等待2MSL时间主要目的是怕最后一个 ACK包对方没收到，

那么对方在超时后将重发第三次握手的FIN包，

主动关闭端接到重发的FIN包后可以再发一个ACK应答包。

在TIME\_WAIT状态 时两端的端口不能使用，要等到2MSL时间结束才可继续使用。

当连接处于2MSL等待阶段时任何迟到的报文段都将被丢弃。

不过在实际应用中可以通过设置 SO\_REUSEADDR选项达到不必等待2MSL时间结束再使用此端口

**<10>** **tcp长连接和短连接**

**1. TCP短连接**

模拟一种TCP短连接的情况:

1. client 向 server 发起连接请求
2. server 接到请求，双方建立连接
3. client 向 server 发送消息
4. server 回应 client
5. 一次读写完成，此时双方任何一个都可以发起 close 操作

在第 步骤5中，一般都是 client 先发起 close 操作。当然也不排除有特殊的情况。

从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作！

**2. TCP长连接**

模拟一种长连接的情况:

1. client 向 server 发起连接
2. server 接到请求，双方建立连接
3. client 向 server 发送消息
4. server 回应 client
5. 一次读写完成，连接不关闭
6. 后续读写操作...
7. 长时间操作之后client发起关闭请求

<11> **listen的队列长度**

**服务器端运行**

#coding=utf-8

from socket import \*

from time import sleep

# 创建socket

tcpSerSocket = socket(AF\_INET, SOCK\_STREAM)

# 绑定本地信息

address = ('', 7788)

tcpSerSocket.bind(address)

connNum = int(raw\_input("请输入要最大的链接数:"))

# 使用socket创建的套接字默认的属性是主动的，使用listen将其变为被动的，这样就可以接收别人的链接了

tcpSerSocket.listen(connNum)

while True:

# 如果有新的客户端来链接服务器，那么就产生一个新的套接字专门为这个客户端服务器

newSocket, clientAddr = tcpSerSocket.accept()

print clientAddr

sleep(1)

**客户端运行**

#coding=utf-8

from socket import \*

connNum = raw\_input("请输入要链接服务器的次数:")

for i in range(int(connNum)):

s = socket(AF\_INET, SOCK\_STREAM)

s.connect(("192.168.1.102", 7788))

print(i)

1、listen中的black表示已经建立链接和半链接的总数

2、如果当前已建立链接数和半链接数以达到设定值，那么新客户端就不会connect成功，而是等待服务器

**<12>** **常见网络攻击案例**

**1. tcp半链接攻击**

tcp半链接攻击也称为：SYN Flood (SYN洪水)是种典型的DoS (Denial of Service，拒绝服务) 攻击效果就是服务器TCP连接资源耗尽，停止响应正常的TCP连接请求

**2. dns攻击**

**dns服务器被劫持**

我们知道一个域名服务器对其区域内的用户解析请求负责，但是并没有一个机制去监督它有没有真地负责。也就是说域名服务器的权力并没有被关在笼子里，所以它既可以认真地“为人民服务”，也可以“指鹿为马”。于是有些流氓的域名服务器故意更改一些域名的解析结果，将用户引向一个错误的目标地址。这就叫作 DNS 劫持，主要用来阻止用户访问某些特定的网站，或者是将用户引导到广告页面。

**dns欺骗**

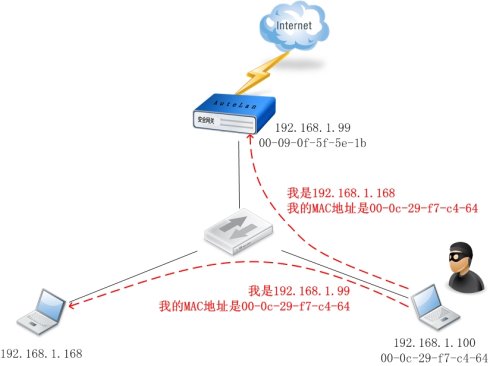
DNS 欺骗简单来说就是用一个假的 DNS 应答来欺骗用户计算机，让其相信这个假的地址，并且抛弃真正的 DNS 应答。

在一台主机发出 DNS 请求后，它就开始等待应答，如果此时有一个看起来正确（拥有和DNS请求一样的序列号）的应答包，它就会信以为真，并且丢弃稍晚一点到达的应答。

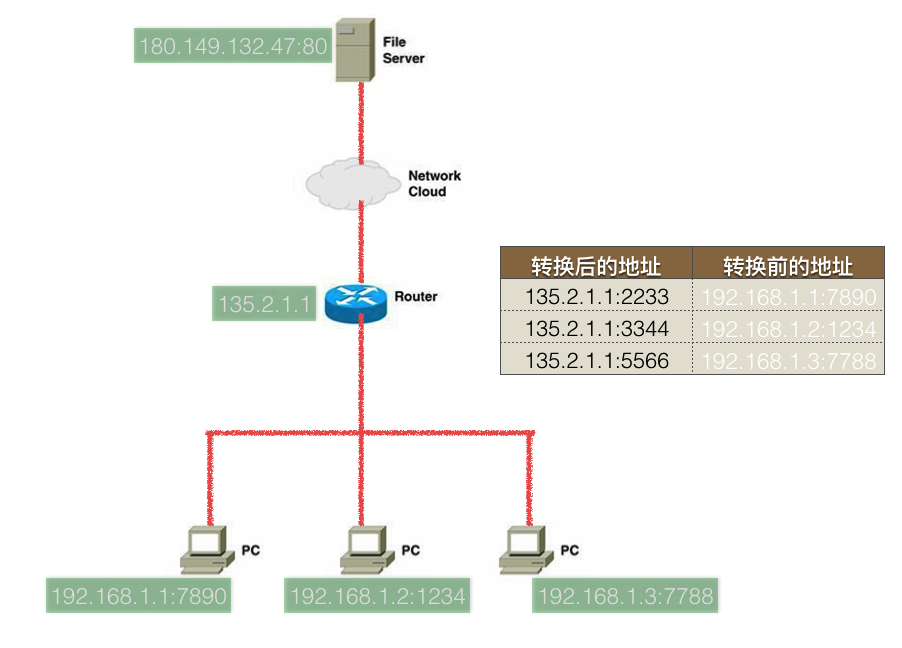
**查看域名解析的ip地址方法**

nslookup 域名 (例如 nslookup baidu.com)

**3. arp攻击**



<13> **家庭上网解析**



**并发服务器、HTTP协议**

**<1>单进程服务器**

from socket import \*

serSocket = socket(AF\_INET, SOCK\_STREAM)

# 重复使用绑定的信息

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

localAddr = ('', 7788)

serSocket.bind(localAddr)

serSocket.listen(5)

while True:

print('-----主进程，，等待新客户端的到来------')

newSocket,destAddr = serSocket.accept()

print('-----主进程，，接下来负责数据处理[%s]-----'%str(destAddr))

try:

while True:

recvData = newSocket.recv(1024)

if len(recvData)>0:

print('recv[%s]:%s'%(str(destAddr), recvData))

else:

print('[%s]客户端已经关闭'%str(destAddr))

break

finally:

newSocket.close()

serSocket.close()

**同一时刻只能为一个客户进行服务，不能同时为多个客户服务**

**当recv接收数据时，返回值为空，即没有返回数据，那么意味着客户端已经调用了close关闭了；因此服务器通过判断recv接收数据是否为空来判断客户端是否已经下线**

**<2>多进程服务器**

from socket import \*

from multiprocessing import \*

from time import sleep

# 处理客户端的请求并为其服务

def dealWithClient(newSocket,destAddr):

while True:

recvData = newSocket.recv(1024)

if len(recvData)>0:

print('recv[%s]:%s'%(str(destAddr), recvData))

else:

print('[%s]客户端已经关闭'%str(destAddr))

break

newSocket.close()

def main():

serSocket = socket(AF\_INET, SOCK\_STREAM)

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

localAddr = ('', 7788)

serSocket.bind(localAddr)

serSocket.listen(5)

try:

while True:

print('-----主进程，，等待新客户端的到来------')

newSocket,destAddr = serSocket.accept()

print('-----主进程，，接下来创建一个新的进程负责数据处理[%s]-----'%str(destAddr))

client = Process(target=dealWithClient, args=(newSocket,destAddr))

client.start()

#因为已经向子进程中copy了一份（引用），并且父进程中这个套接字也没有用处了

#所以关闭

newSocket.close()

finally:

#当为所有的客户端服务完之后再进行关闭，表示不再接收新的客户端的链接

serSocket.close()

if \_\_name\_\_ == '\_\_main\_\_':

main()

**<3>多线程服务器**

#coding=utf-8

from socket import \*

from threading import Thread

from time import sleep

# 处理客户端的请求并执行事情

def dealWithClient(newSocket,destAddr):

while True:

recvData = newSocket.recv(1024)

if len(recvData)>0:

print('recv[%s]:%s'%(str(destAddr), recvData))

else:

print('[%s]客户端已经关闭'%str(destAddr))

break

newSocket.close()

def main():

serSocket = socket(AF\_INET, SOCK\_STREAM)

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

localAddr = ('', 7788)

serSocket.bind(localAddr)

serSocket.listen(5)

try:

while True:

print('-----主进程，，等待新客户端的到来------')

newSocket,destAddr = serSocket.accept()

print('-----主进程，，接下来创建一个新的进程负责数据处理[%s]-----'%str(destAddr))

client = Thread(target=dealWithClient, args=(newSocket,destAddr))

client.start()

#因为线程中共享这个套接字，如果关闭了会导致这个套接字不可用，

#但是此时在线程中这个套接字可能还在收数据，因此不能关闭

#newSocket.close()

finally:

serSocket.close()

if \_\_name\_\_ == '\_\_main\_\_':

main()

**<4>** **单进程服务器-非堵塞模式**

**服务器**

#coding=utf-8

from socket import \*

import time

# 用来存储所有的新链接的socket

g\_socketList = []

def main():

serSocket = socket(AF\_INET, SOCK\_STREAM)

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

localAddr = ('', 7788)

serSocket.bind(localAddr)

#可以适当修改listen中的值来看看不同的现象

serSocket.listen(1000)

#将套接字设置为非堵塞

#设置为非堵塞后，如果accept时，恰巧没有客户端connect，那么accept会

#产生一个异常，所以需要try来进行处理

serSocket.setblocking(False)

while True:

#用来测试

#time.sleep(0.5)

try:

newClientInfo = serSocket.accept()

except Exception as result:

pass

else:

print("一个新的客户端到来:%s"%str(newClientInfo))

newClientInfo[0].setblocking(False)

g\_socketList.append(newClientInfo)

# 用来存储需要删除的客户端信息

needDelClientInfoList = []

for clientSocket,clientAddr in g\_socketList:

try:

recvData = clientSocket.recv(1024)

if len(recvData)>0:

print('recv[%s]:%s'%(str(clientAddr), recvData))

else:

print('[%s]客户端已经关闭'%str(clientAddr))

clientSocket.close()

g\_needDelClientInfoList.append((clientSocket,clientAddr))

except Exception as result:

pass

for needDelClientInfo in needDelClientInfoList:

g\_socketList.remove(needDelClientInfo)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**客户端**

#coding=utf-8

from socket import \*

import random

import time

serverIp = raw\_input("请输入服务器的ip:")

connNum = raw\_input("请输入要链接服务器的次数(例如1000):")

g\_socketList = []

for i in range(int(connNum)):

s = socket(AF\_INET, SOCK\_STREAM)

s.connect((serverIp, 7788))

g\_socketList.append(s)

print(i)

while True:

for s in g\_socketList:

s.send(str(random.randint(0,100)))

# 用来测试用

#time.sleep(1)

**<5> select版-TCP服务器**

**1. select 原理**

在多路复用的模型中，比较常用的有select模型和epoll模型。这两个都是系统接口，由操作系统提供。当然，Python的select模块进行了更高级的封装。

网络通信被Unix系统抽象为文件的读写，通常是一个设备，由设备驱动程序提供，驱动可以知道自身的数据是否可用。支持阻塞操作的设备驱动通常会实现一组自身的等待队列，如读/写等待队列用于支持上层(用户层)所需的block或non-block操作。设备的文件的资源如果可用（可读或者可写）则会通知进程，反之则会让进程睡眠，等到数据到来可用的时候，再唤醒进程。

这些设备的文件描述符被放在一个数组中，然后select调用的时候遍历这个数组，如果对于的文件描述符可读则会返回改文件描述符。当遍历结束之后，如果仍然没有一个可用设备文件描述符，select让用户进程则会睡眠，直到等待资源可用的时候在唤醒，遍历之前那个监视的数组。每次遍历都是依次进行判断的。

**2. select 回显服务器**

import select

import socket

import sys

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.bind(('', 7788))

server.listen(5)

inputs = [server, sys.stdin]

running = True

while True:

# 调用 select 函数，阻塞等待

readable, writeable, exceptional = select.select(inputs, [], [])

# 数据抵达，循环

for sock in readable:

# 监听到有新的连接

if sock == server:

conn, addr = server.accept()

# select 监听的socket

inputs.append(conn)

# 监听到键盘有输入

elif sock == sys.stdin:

cmd = sys.stdin.readline()

running = False

break

# 有数据到达

else:

# 读取客户端连接发送的数据

data = sock.recv(1024)

if data:

sock.send(data)

else:

# 移除select监听的socket

inputs.remove(sock)

sock.close()

# 如果检测到用户输入敲击键盘，那么就退出

if not running:

break

server.close()

**3. 总结**

**优点**

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。

**缺点**

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上一般为1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048.对socket进行扫描时是依次扫描的，即采用轮询的方法，效率较低。当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度，不管哪个Socket是活跃的，都遍历一遍。这会浪费很多CPU时间。

**<6>** **epoll版-TCP服务器**

**1. epoll的优点：**

1、没有最大并发连接的限制，能打开的FD(指的是文件描述符，通俗的理解就是套接字对应的数字编号)的上限远大于1024

2、效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；即epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，epoll的效率就会远远高于select和poll。

**2. epoll使用参考代码**

import socket

import select

# 创建套接字

s = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

# 设置可以重复使用绑定的信息

s.setsockopt(socket.SOL\_SOCKET,socket.SO\_REUSEADDR,1)

# 绑定本机信息

s.bind(("",7788))

# 变为被动

s.listen(10)

# 创建一个epoll对象

epoll=select.epoll()

# 测试，用来打印套接字对应的文件描述符

# print s.fileno()

# print select.EPOLLIN|select.EPOLLET

# 注册事件到epoll中

# epoll.register(fd[, eventmask])

# 注意，如果fd已经注册过，则会发生异常

# 将创建的套接字添加到epoll的事件监听中

epoll.register(s.fileno(),select.EPOLLIN|select.EPOLLET)

connections = {}

addresses = {}

# 循环等待客户端的到来或者对方发送数据

while True:

# epoll 进行 fd 扫描的地方 -- 未指定超时时间则为阻塞等待

epoll\_list=epoll.poll()

# 对事件进行判断

for fd,events in epoll\_list:

# print fd

# print events

# 如果是socket创建的套接字被激活

if fd == s.fileno():

conn,addr=s.accept()

print('有新的客户端到来%s'%str(addr))

# 将 conn 和 addr 信息分别保存起来

connections[conn.fileno()] = conn

addresses[conn.fileno()] = addr

# 向 epoll 中注册 连接 socket 的 可读 事件

epoll.register(conn.fileno(), select.EPOLLIN | select.EPOLLET)

elif events == select.EPOLLIN:

# 从激活 fd 上接收

recvData = connections[fd].recv(1024)

if len(recvData)>0:

print('recv:%s'%recvData)

else:

# 从 epoll 中移除该 连接 fd

epoll.unregister(fd)

# server 侧主动关闭该 连接 fd

connections[fd].close()

print("%s---offline---"%str(addresses[fd]))

**说明**

EPOLLIN （可读）

EPOLLOUT （可写）

EPOLLET （ET模式）

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：

LT模式：当epoll检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll时，会再次响应应用程序并通知此事件。

ET模式：当epoll检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll时，不会再次响应应用程序并通知此事件。

**<7> 协程**

协程，又称微线程，纤程。英文名Coroutine。

**协程是啥**

首先我们得知道协程是啥？协程其实可以认为是比线程更小的执行单元。 为啥说他是一个执行单元，因为他自带CPU上下文。这样只要在合适的时机， 我们可以把一个协程 切换到另一个协程。 只要这个过程中保存或恢复 CPU上下文那么程序还是可以运行的。

通俗的理解：在一个线程中的某个函数，可以在任何地方保存当前函数的一些临时变量等信息，然后切换到另外一个函数中执行，注意不是通过调用函数的方式做到的，并且切换的次数以及什么时候再切换到原来的函数都由开发者自己确定

**协程和线程差异**

那么这个过程看起来比线程差不多。其实不然, 线程切换从系统层面远不止保存和恢复 CPU上下文这么简单。 操作系统为了程序运行的高效性每个线程都有自己缓存Cache等等数据，操作系统还会帮你做这些数据的恢复操作。 所以线程的切换非常耗性能。但是协程的切换只是单纯的操作CPU的上下文，所以一秒钟切换个上百万次系统都抗的住。

**协程的问题**

但是协程有一个问题，就是系统并不感知，所以操作系统不会帮你做切换。 那么谁来帮你做切换？让需要执行的协程更多的获得CPU时间才是问题的关键。

**例子**

目前的协程框架一般都是设计成 1:N 模式。所谓 1:N 就是一个线程作为一个容器里面放置多个协程。 那么谁来适时的切换这些协程？答案是有协程自己主动让出CPU，也就是每个协程池里面有一个调度器， 这个调度器是被动调度的。意思就是他不会主动调度。而且当一个协程发现自己执行不下去了(比如异步等待网络的数据回来，但是当前还没有数据到)， 这个时候就可以由这个协程通知调度器，这个时候执行到调度器的代码，调度器根据事先设计好的调度算法找到当前最需要CPU的协程。 切换这个协程的CPU上下文把CPU的运行权交个这个协程，直到这个协程出现执行不下去需要等等的情况，或者它调用主动让出CPU的API之类，触发下一次调度。

**那么这个实现有没有问题？**

其实是有问题的，假设这个线程中有一个协程是CPU密集型的他没有IO操作， 也就是自己不会主动触发调度器调度的过程，那么就会出现其他协程得不到执行的情况， 所以这种情况下需要程序员自己避免。这是一个问题，假设业务开发的人员并不懂这个原理的话就可能会出现问题。

**协程的好处**

在IO密集型的程序中由于IO操作远远慢于CPU的操作，所以往往需要CPU去等IO操作。 同步IO下系统需要切换线程，让操作系统可以在IO过程中执行其他的东西。 这样虽然代码是符合人类的思维习惯但是由于大量的线程切换带来了大量的性能的浪费，尤其是IO密集型的程序。

所以人们发明了异步IO。就是当数据到达的时候触发我的回调。来减少线程切换带来性能损失。 但是这样的坏处也是很大的，主要的坏处就是操作被 “分片” 了，代码写的不是 “一气呵成” 这种。 而是每次来段数据就要判断 数据够不够处理哇，够处理就处理吧，不够处理就在等等吧。这样代码的可读性很低，其实也不符合人类的习惯。

但是协程可以很好解决这个问题。比如 把一个IO操作 写成一个协程。当触发IO操作的时候就自动让出CPU给其他协程。要知道协程的切换很轻的。 协程通过这种对异步IO的封装 既保留了性能也保证了代码的容易编写和可读性。在高IO密集型的程序下很好。但是高CPU密集型的程序下没啥好处。

**协程一个简单实现**

import time

def A():

while True:

print("----A---")

yield

time.sleep(0.5)

def B(c):

while True:

print("----B---")

c.next()

time.sleep(0.5)

if \_\_name\_\_=='\_\_main\_\_':

a = A()

B(a)

**<8> 协程-greenlet版**

为了更好使用协程来完成多任务，python中的greenlet模块对其封装，从而使得切换任务变的更加简单

**安装方式**

使用如下命令安装greenlet模块:

sudo pip install greenlet

**实例：**

#coding=utf-8

from greenlet import greenlet

import time

def test1():

while True:

print "---A--"

gr2.switch()

time.sleep(0.5)

def test2():

while True:

print "---B--"

gr1.switch()

time.sleep(0.5)

gr1 = greenlet(test1)

gr2 = greenlet(test2)

#切换到gr1中运行

gr1.switch()

**<9> 协程- gevent版**

greenlet已经实现了协程，但是这个还的人工切换，是不是觉得太麻烦了，不要捉急，python还有一个比greenlet更强大的并且能够自动切换任务的模块gevent

其原理是当一个greenlet遇到IO(指的是input output 输入输出，比如网络、文件操作等)操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。

由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO

**1. gevent的使用**

#coding=utf-8

#请使用python 2 来执行此程序

import gevent

def f(n):

for i in range(n):

print gevent.getcurrent(), i

g1 = gevent.spawn(f, 5)

g2 = gevent.spawn(f, 5)

g3 = gevent.spawn(f, 5)

g1.join()

g2.join()

g3.join()

**2. gevent切换执行**

import gevent

def f(n):

for i in range(n):

print gevent.getcurrent(), i

#用来模拟一个耗时操作，注意不是time模块中的sleep

gevent.sleep(1)

g1 = gevent.spawn(f, 5)

g2 = gevent.spawn(f, 5)

g3 = gevent.spawn(f, 5)

g1.join()

g2.join()

g3.join()

**3. gevent并发下载器**

当然，实际代码里，我们不会用gevent.sleep()去切换协程，而是在执行到IO操作时，gevent自动切换

#coding=utf-8

from gevent import monkey;

import gevent

import urllib2

#有IO才做时需要这一句

monkey.patch\_all()

def myDownLoad(url):

print('GET: %s' % url)

resp = urllib2.urlopen(url)

data = resp.read()

print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([

gevent.spawn(myDownLoad, 'http://www.baidu.com/'),

gevent.spawn(myDownLoad, 'http://www.itcast.cn/'),

gevent.spawn(myDownLoad, 'http://www.itheima.com/'),

])

从上能够看到是先发送的获取baidu的相关信息，然后依次是itcast、itheima，但是收到数据的先后顺序不一定与发送顺序相同，这也就体现出了异步，即不确定什么时候会收到数据，顺序不一定

<10> **gevent版-TCP服务器**

import sys

import time

import gevent

from gevent import socket,monkey

monkey.patch\_all()

def handle\_request(conn):

while True:

data = conn.recv(1024)

if not data:

conn.close()

break

print("recv:", data)

conn.send(data)

def server(port):

s = socket.socket()

s.bind(('', port))

s.listen(5)

while True:

cli, addr = s.accept()

gevent.spawn(handle\_request, cli)

if \_\_name\_\_ == '\_\_main\_\_':

server(7788)