**网络协议和框架学习**

# 1网络协议

网络协议(Network Protocol)为计算机网络中进行数据交换而建立的规则、标准或约定的集合。

例如，网络中一个微机用户和一个大型主机的操作员进行通信，由于这两个数据终端所用字符集不同，因此操作员所输入的命令彼此不认识。为了能进行通信，规定每个终端都要将各自字符集中的字符先变换为标准字符集的字符后，才进入网络传送，到达目的终端之后，再变换为该终端字符集的字符。当然，对于不相容终端，除了需变换字符集字符外还需转换其他特性，如显示格式、行长、行数、屏幕滚动方式等也需作相应的变换。

## 1.1要素

网络协议是由三个要素组成：

(1) 语义。语义是解释控制信息每个部分的意义。它规定了需要发出何种控制信息，以及完成的动作与做出什么样的响应。

(2) 语法。语法是用户数据与控制信息的结构与格式，以及数据出现的顺序。

(3) 时序。时序是对事件发生顺序的详细说明。（也可称为“同步”）。[2]

人们形象地把这三个要素描述为：语义表示要做什么，语法表示要怎么做，时序表示做的顺序。

## 1.2工作方式

网络上的计算机之间又是如何交换信息的呢？就像我们说话用某种语言一样，在网络上的各台计算机之间也有一种语言，这就是网络协议，不同的计算机之间必须使用相同的网络协议才能进行通信。

网络协议是网络上所有设备（网络服务器、计算机及交换机、路由器、防火墙等）之间通信规则的集合，它规定了通信时信息必须采用的格式和这些格式的意义。大多数网络都采用分层的体系结构，每一层都建立在它的下层之上，向它的上一层提供一定的服务，而把如何实现这一服务的细节对上一层加以屏蔽。一台设备上的第 n层与另一台设备上的第n层进行通信的规则就是第n层协议。在网络的各层中存在着许多协议，接收方和发送方同层的协议必须一致，否则一方将无法识别另一方发出的信息。网络协议使网络上各种设备能够相互交换信息。常见的协议有：TCP/IP协议、IPX/SPX协议、NetBEUI协议等。

当然了，网络协议也有很多种，具体选择哪一种协议则要看情况而定。Internet上的计算机使用的是TCP/IP协议。

ARPANET成功的主要原因是因为它使用了TCP/IP标准网络协议，TCP/IP（Transmission Control Protocol/Internet Protocol）----传输控制协议/互联网协议是Internet采用的一种标准网络协议。它是由ARPA于1977年到1979年推出的一种网络体系结构和协议规范。随着Internet网的发展，TCP/IP也得到进一步的研究开发和推广应用，成为Internet网上的"通用语言"。

## 1.3层次结构

由于网络节点之间联系的复杂性，在制定协议时，通常把复杂成分分解成一些简单成分，然后再将它们复合起来。最常用的复合技术就是层次方式，网络协议的层次结构如下：

(1)结构中的每一层都规定有明确的服务及接口标准。

(2)把用户的应用程序作为最高层

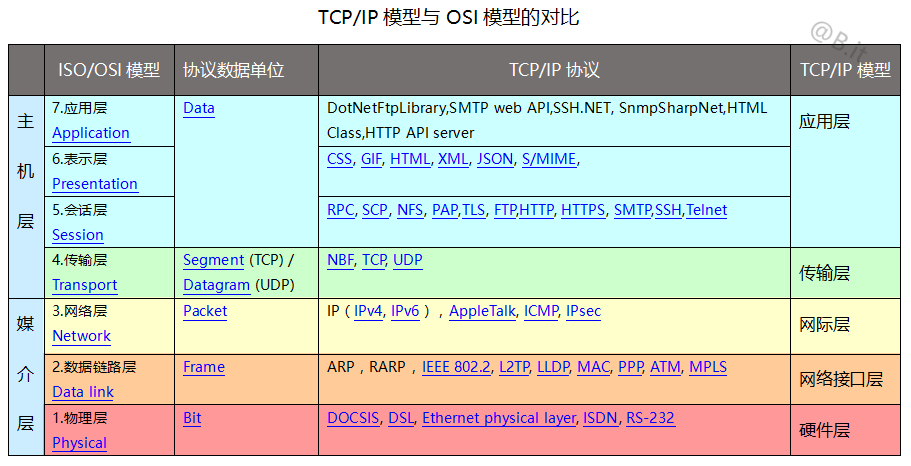
(3)除了最高层外，中间的每一层都向上一层提供服务，同时又是下一层的用户。

(4)把物理通信线路作为最低层，它使用从最高层传送来的参数，是提供服务的基础。

## 1.4层次划分

为了使不同计算机厂家生产的计算机能够相互通信，以便在更大的范围内建立计算机网络，国际标准化组织(International Standard Organization,ISO)在1978年提出了“开放系统互联参考模型”，即著名的OSI/RM模型(Open System Interconnection/Reference Model)。它将计算机网络体系结构的通信协议划分为七层，自下而上依次为：物理层(Physics Layer)、数据链路层(Data Link Layer)、网络层(Network Layer)、传输层(Transport Layer)、会话层(Session Layer)、表示层(Presentation Layer)、应用层(Application Layer)。

TCP/IP(Transmission Control Protocol/Internet Protocol)作为Internet的核心协议。它是个协议族，包含多种协议。



发送请求的过程是从最顶层（应用层）出发，每一层负责封装属于自己的信息到请求中，最后将一整个请求发送给对方。

接收请求的过程是从最底层（网络接口层）开始，每一层的协议负责解析属于自己的东西，比如网络层(IP)处理ip信息，传输层(TCP)处理点对点的端口，应用层(HTTP)处理Request或Response的Line\Header\Body。

## 1.5常用协议

TCP/IP协议毫无疑问是这三大协议中最重要的一个，作为互联网的基础协议，没有它就根本不可能上网，任何和互联网有关的操作都离不开TCP/IP协议。不过TCP/IP协议也是这三大协议中配置起来最麻烦的一个，单机上网还好，而通过局域网访问互联网的话，就要详细设置IP地址，网关，子网掩码，DNS服务器等参数。

TCP/IP尽管是目前最流行的网络协议，但TCP/IP协议在局域网中的通信效率并不高，使用它在浏览“网上邻居”中的计算机时，经常会出现不能正常浏览的现象。此时安装NetBEUI协议就会解决这个问题。

NetBEUI即NetBios Enhanced User Interface ，或NetBios增强用户接口。它是NetBIOS协议的增强版本，曾被许多操作系统采用，例如Windows for Workgroup、Win 9x系列、Windows NT等。NETBEUI协议在许多情形下很有用，是WINDOWS98之前的操作系统的缺省协议。NetBEUI协议是一种短小精悍、通信效率高的广播型协议，安装后不需要进行设置，特别适合于在“网络邻居”传送数据。所以建议除了TCP/IP协议之外，小型局域网的计算机也可以安上NetBEUI协议。另外还有一点要注意，如果一台只装了TCP/IP协议的WINDOWS98机器要想加入到WINNT域，也必须安装NetBEUI协议。

IPX/SPX协议本来就是Novell开发的专用于NetWare网络中的协议，但是也非常常用--大部分可以联机的游戏都支持IPX/SPX协议，比如星际争霸，反恐精英等等。虽然这些游戏通过TCP/IP协议也能联机，但显然还是通过IPX/SPX协议更省事，因为根本不需要任何设置。除此之外，IPX/SPX协议在非局域网络中的用途似乎并不是很大.如果确定不在局域网中联机玩游戏，那么这个协议可有可无。

# 2TCP

TCP(Transmission Control Protocol，传输控制协议)是一种面向连接(连接导向)的、可靠的基于字节流的传输层通信协议。TCP将用户数据打包成报文段，它发送后启动一个定时器，另一端收到的数据进行确认、对失序的数据重新排序、丢弃重复数据。

TCP的特点有：

 a)TCP是面向连接的传输层协议

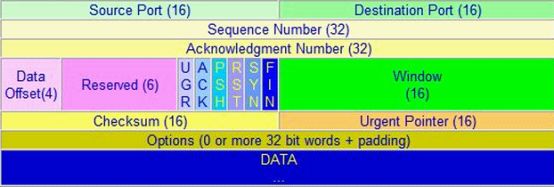
 b)每一条TCP连接只能有两个端点，每一条TCP连接只能是点对点的

 c)TCP提供可靠交付的服务

 d)TCP提供全双工通信。数据在两个方向上独立的进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

 e)面向字节流。面向字节流的含义：虽然应用程序和TCP交互是一次一个数据块，但TCP把应用程序交下来的数据仅仅是一连串的无结构的字节流。

## 2.1 TCP头格式

IMG_256

(1)Source Port(源端口号）：数据发起者的端口号，16bit。

(2)Destination Port(目的端口号）：数据接收者的端口号，16bit。

(3)Sequence Number(顺序号码，Seq）：用于在数据通信中解决网络包乱序(reordering)问题，以保证应用层接收到的数据不会因为网络上的传输问题而乱序(TCP会用这个顺序号码来拼接数据），32bit。

(4)Acknowledgment Number(确认号码，ack）：是数据接收方期望收到发送方在下一个报文段的顺序号码（Seq），因此确认号码应当是上次已成功收到顺序号码(Seq)加1，32bit。

(5)Offset(TCP报文头长度)：用于存储报文头中有多少个32bit(上图的一行)，存储长度为4bit，最大可表示（2^3+2^2+2^1+1）\*32bit=60bytes的报文头。最小取值5，5\*32bit=20bytes。

(6)Reserved(保留)：6bit, 均为0

(7)TCP Flags(TCP标识位)每个长度均为1bit

   CWR：压缩，TCP Flags值0x80。

    ECE：拥塞，0x40。

    URG：紧急，0x20。当URG=1时，表示报文段中有紧急数据，应尽快传送。

    ACK：确认，0x10。当ACK = 1时，代表这是一个确认的TCP包，取值0则不是确认包。

    PSH：推送，0x08。当发送端PSH=1时，接收端尽快的交付给应用进程。

    RST：复位，0x04。当RST=1时，表明TCP连接中出现严重差错，必须释放连接，再重新建立连接。

   SYN：同步，0x02。在建立连接时用来同步序号。SYN=1， ACK=0表示一个连接请求报文段。SYN=1，ACK=1表示同意建立连接。

    FIN：终止，0x01。当FIN=1时，表明此报文段的发送端的数据已经发送完毕，并要求释放传输连接。

(8)窗口：用来控制对方发送的数据量，通知发放已确定的发送窗口上限。

(9)检验和：该字段检验的范围包括头部和数据这两部分。由发端计算和存储，并由收端进行验证。

(10)紧急指针：紧急指针在URG=1时才有效，它指出本报文段中的紧急数据的字节数。

(11)TCP选项：长度可变，最长可达40字节

备注：ISN（Inital Sequence Number）：初始化Sequence Number，发生在建立连接时。

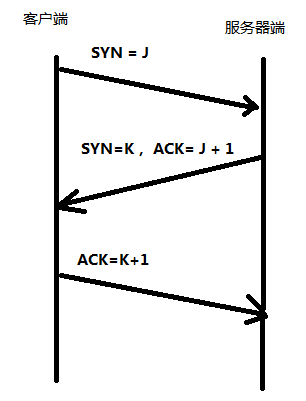
## 2.2 TCP三次握手过程

a)主机A通过向主机B 发送一个含有同步序列号的标志位（SYN）的数据段给主机B ,向主机B 请求建立连接,通过这个数据段, 主机A告诉主机B 两件事:我想要和你通信;你可以用哪个序列号作为起始数据段来回应我.

b)主机B 收到主机A的请求后,用一个带有确认应答(ACK)和同步序列号(SYN)标志位的数据段响应主机A,也告诉主机A两件事: 我已经收到你的请求了,你可以传输数据了;你要用哪佧序列号作为起始数据段来回应我

c)主机A收到这个数据段后,再发送一个确认应答（ACK）,确认已收到主机B 的数据段:"我已收到回复,我现在要开始传输实际数据了。这样3次握手就完成了,主机A和主机B 就可以传输数据了.

3次握手的特点 没有应用层的数据 SYN这个标志位只有在TCP建立长连接时才会被置1 握手完成后SYN标志位被置0



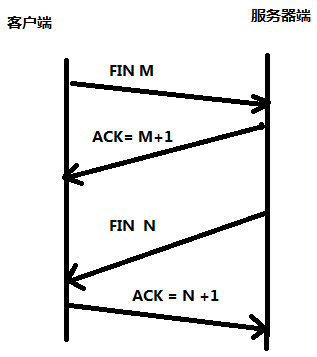
## 2.3 TCP四次挥手

a)当主机A完成数据传输后,将控制位FIN置1,提出停止TCP连接的请求

b)主机B收到FIN后对其作出响应,确认这一方向上的TCP连接将关闭,将ACK置1

c)由B 端再提出反方向的关闭请求,将FIN置1

d)主机A对主机B的请求进行确认,将ACK置1,双方向的关闭结束.



名词解释：

ACK TCP报头的控制位之一,对数据进行确认.确认由目的端发出,用它来告诉发送端这个序列号之前的数据段 都收到了.比如,确认号为X,则表示前X-1个数据段都收到了,只有当ACK>1时,确认号才有效,当ACK=0时,确认号无效,这时会要求重传数据,保证数据的完整性.

SYN 同步序列号,TCP建立连接时将这个位 重置为1

FIN 发送端完成发送任务位,当TCP完成数据传输需要断开时,提出断开连接的一方将这位置1

由TCP的三次握手和四次断开可以看出,TCP使用面向连接的通信方式,大大提高了数据通信的可靠性,使发送数据端 和接收端在数据正式传输前就有了交互,为数据正式传输打下了可靠的基础。

## 2.4案例

服务器端:

|  |
| --- |
| **import** socket **import** threading **import** time   **def** tcplink(sock, addr):  *'''  处理tcp连接* **:param** *sock: 当前连接* **:param** *addr: 当前连接的客服端ip和端口* **:return***:  '''* print(**'Accept new connection from %s:%s...'** % (sock, addr))  sock.send(**b'Welcome!'**)  **while True**:  *# 接收最大1024个字节的数据* data = sock.recv(1024)  time.sleep(1)  *# 当接收到 exit 之后，退出获取数据的循环* **if not** data **or** data.decode(**'utf-8'**) == **'exit'**:  print(**'退出连接，准备中断'**)  **break  else**:  print(**'接收到数据：'**, data.decode())  *# 发送 一个数据到客服端* sock.send((**'Hello, %s!'** % data.decode(**'utf-8'**)).encode(**'utf-8'**))   *# 关闭当前连接* sock.close()  print(**'Connection from %s:%s closed.'** % addr)  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  *# 创建一个 socoket* s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  *# 绑定 ip和端口  # s.bind(('127.0.0.1', 9999))* s.bind((**'192.168.2.132'**, 9999))   *# 最多允许5个 连接* s.listen(5)  print(**"waiting for connection...."**)  **while True**:  *# 获取连接的 sock 和 地址* sock, addr = s.accept()  *# 多 线程* t = threading.Thread(target=tcplink, args=(sock, addr))  *# 线程启动* t.start() |

客户端:

|  |
| --- |
| **import** socket s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) *# 建立连接:* s.connect((**'192.168.2.159'**, 9999)) *# 接收欢迎消息:* print(s.recv(1024).decode(**'utf-8'**))  *# for data in [b'Michael', b'Tracy', b'Sarah']: # # 发送数据: # s.send(data) # print(s.recv(1024).decode('utf-8')) # s.send(b'exit')* **while** 1:  d = input(**'请输入发送内容：'**)  s.send(d.strip().encode())  print(s.recv(1024).decode(**'utf-8'**))   **if** d == **'exit'**:  print(**'关闭连接！！！'**)  **break** s.close() |

# 3UDP

UDP(User Datagram Protocol,用户数据报协议)是一种简单的面向数据报、无连接、传输层协议，并且保留了信息边界。UDP不提供错误校正，不保证有序，无法去重复，没有流量控制和拥塞控制，不能保证数据一定到达目的地，但是可以通过校验和提供错误侦测。UDP提供的的是不可靠传输，因此要有应用层来提供这些功能。

UDP的特点：

a)是无连接的。相比于TCP协议，UDP协议在传送数据前不需要建立连接，当然也就没有释放连接。

b)是尽最大努力交付的。也就是说UDP协议无法保证数据能够准确的交付到目的主机。也不需要对接收到的UDP报文进行确认。

c)是面向报文的。也就是说UDP协议将应用层传输下来的数据封装在一个UDP包中，不进行拆分或合并。因此，运输层在收到对方的UDP包后，会去掉首部后，将数据原封不动的交给应用进程。

d)没有拥塞控制。因此UDP协议的发送速率不受网络的拥塞度影响。

e)UDP支持一对一、一对多、多对一和多对多的交互通信。

f)UDP的头部占用较小，只占用8个字节。

## 3.1 UDP头格式



## 3.2案例

服务器端:

|  |
| --- |
| **import** socket s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) *# 绑定端口:* s.bind((**'127.0.0.1'**, 9999)) print(**'Bind UDP on 9999...'**) **while True**:  *# 接收数据:* data, addr = s.recvfrom(1024)  print(**'Received from %s:%s.'** % addr)  s.sendto(**b'Hello, %s!'** % data, addr) |

客户端:

|  |
| --- |
| **import** socket s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) **for** data **in** [**b'Michael'**, **b'Tracy'**, **b'Sarah'**]:  *# 发送数据:* s.sendto(data, (**'127.0.0.1'**, 9999))  *# 接收数据:* print(s.recv(1024).decode(**'utf-8'**)) s.close() |

# 4FTP

文件传输协议FTP(File Transfer Protocol)是因特网中使用最广泛的文件传输协议。FTP使用交互式的访问，允许客户指定文件的类型和格式(如指明是否使用ASCII码)，并允许文件具有存取权限(如访问文件的用户必须经过授权，并输入有效的口令)。

文件传输协议有基于TCP的FTP和基于UDP的简单文件传输协议TFTP，它们都是文件共享协议中的一大类，即复制整个文件，其特点是：若要存取一个文件，就必须先获得一个本地的文件副本。如果要修改文件，只能对文件的副本进行修改，然后再将修改后的文件传回到原节点。

## 4.1 FTP基本工作原理

FTP屏蔽了各计算机系统的细节，因而适合在异构网络中任意计算机之间传送文件。FTP只提供文件传送的一些基本服务，它使用TCP可靠地运输服务，FTP主要功能是减小或消除在不同系统下处理文件的不兼容性。

FTP使用客户端-服务器模型，一个FTP服务器进程可以为多个客户进程提供服务。FTP服务器有两大部分组成：一个主进程，负责接受新的请求；还有若干从属进程，负责处理单个请求。主进程工作步骤

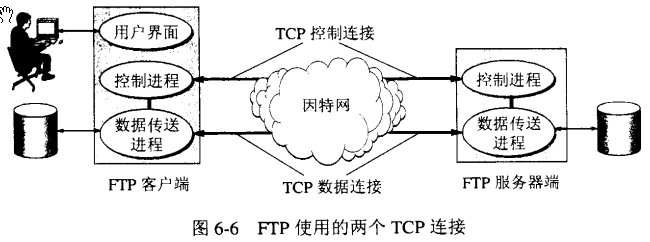
a)打开熟知端口(21)，使客户进程能够连接上

b)等待客户进程发送连接请求

c)启动从属进程处理客户进程发送的连接请求，从属进程处理完请求后结束，从属进程在运行期间可能根据需要可创建其他一些子进程

d)回到等待状态，继续接受其他客户进程发起的请求，主进程与从属进程的处理是并发进行的

**FTP工作时情况**



FTP控制连接在整个会话期间都保持打开，只用来发送连接/传送请求。当客户进程向服务器发送连接请求时，寻找连接服务器进程的熟知端口21，同时还要告诉服务器进程自己的另一个端口号码，用于建立数据传送连接。接着，服务器进程用自己传送数据的熟知端口20与客户进程所提供的端口号码建立数据传送连接，FTP使用了2个不同的端口号，所以数据连接和控制连接不会混乱。

1.1 FTP数据表示

FTP协议规定了控制协议传送与存储的多种选择，在以下4个方面必须做出一个选择。

 文件类型：ASCII码文件(默认的)/ 图像文件类型(二进制的)/ 本地文件类型(用于在具有不同字节大小主机间传送二进制数据)

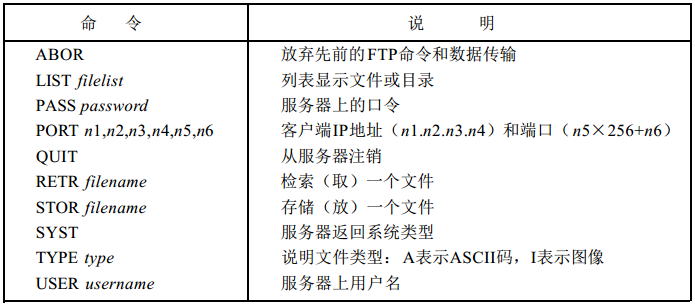
 格式控制：该选项针对ASCII类型文件适用，非打印(默认选择，文件中不包含垂直格式信息)/ 远程登录格式控制

 结构：文件结构(默认选择，文件被认为是一个连续的字节流，不存在内部的文件结构)/ 记录结构(用于文本文件)

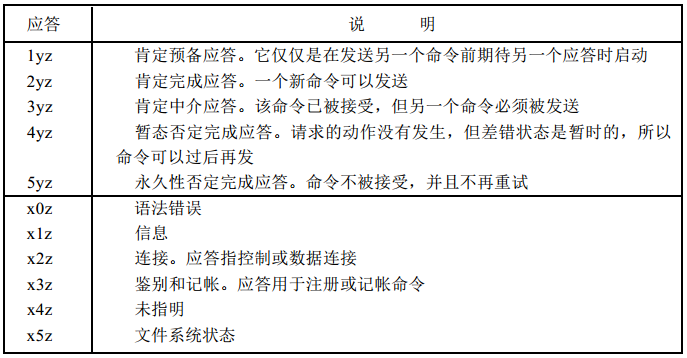
 传输方式：流方式(模式选择，文件以字节流方式传输，对于文件结构，发送方在文件尾提示关闭数据连接，对于记录结构，有专用的两字节序列码记录结束和文件结束)/ 块方式(文件以一系列块来传送，每块前面有一个或多个首部字节)/ 压缩方式

1.2 FTP命令和应答

命令和应答在客户和服务器的控制连接上以 NVT ASCII码形式传送。这就要求在每行结尾都要返回C R、 L F对（也就是每个命令或每个应答）。这些命令都是3或4个字节的大写ASCII字符，其中一些带选项参数。从客户向服务器发送的FTP命令超过30种。下图是比较常用的几种命令：



应答都是A S C I I码形式的3位数字，并跟有报文选项。其原因是软件系统需要根据数字代码来决定如何应答，而选项串是面向人工处理的。由于客户通常都要输出数字应答和报文串，一个可交互的用户可以通过阅读报文串（而不必记忆所有数字回答代码的含义）来确定应答的含义。

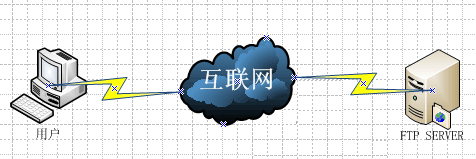


## 4.2 FTP工作模式

FTP有两种工作模式，分别是主动模式(PORT)和被动模式(PASV)两种模式，这两种模式是按照FTP服务器的“角度”来说的，更通俗一点说就是：在传输数据时，如果是服务器主动连接客户端，那就是主动模式；如果是客户端主动连接服务器，那就是被动模式。

**主动模式(PORT)**

如下图所示，用户主机直接暴露在互联网中，用户连接FTP SERVER使用主动模式遵循以下一个过程：



a)用户主机一个随机端口连接FTP SERVER的TCP21端口进行协商；

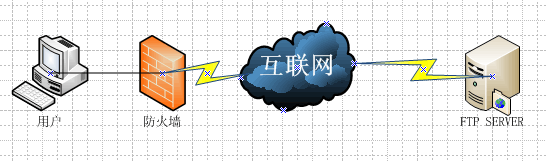
b)用户主机告诉FTP SERVER，我的XXXX端口已经打开，你可以放心大胆的连过来；

c)然后FTP SERVER就用TCP20端口连接用户主机的XXXX端口，数据传输开始。

**被动模式(PASV)**

当用户主机前端多了一道防火墙(客户机使用地址转换接入互联网)，事情就变的复杂了，首先，主动模式是FTP SERVER连接客户机，防火墙肯定不干；即便乐意干客户机与服务器协商的XXXX端口也是随机端口，你让迷茫的防火墙情何以堪，这时就需要客户端软件使用被动模式主动连接防火墙，这么一来正合防火墙的胃口。

客户端使用被动模式遵循以下过程：



a)首先用户使用随机端口连接FTP SERVER的TCP 21端口进行协商；

b)FTP SERVER告诉客户机：我的XXXX端口开放了，你连过来吧；

c)客户机使用一个新的随机端口连接FTP SERVER的XXXX端口传输数据。

可以看出，当不存在防火墙的情况下，主动模式和被动模式均适用，但就服务器安全方面考虑，使用主动模式为佳，因为多一个被动连接服务器就要打多开一个端口，开放的端口越多，就意味着服务器面临的危险越大。

## 4.3 案例

Python中默认安装的ftplib模块定义了**FTP类**，其中函数有限，可用来实现简单的ftp客户端，用于上传或下载文件，函数列举如下

|  |
| --- |
| **from** ftplib **import** FTP *#加载ftp模块* ftp=FTP() *#设置变量* ftp.set\_debuglevel(2) *#打开调试级别2，显示详细信息* ftp.connect(**"IP"**,**"port"**) *#连接的ftp sever和端口* ftp.login(**"user"**,**"password"**) *#连接的用户名，密码print ftp.getwelcome() #打印出欢迎信息* ftp.cmd(**"xxx/xxx"**) *#进入远程目录* bufsize=1024 *#设置的缓冲区大小* filename=**"filename.txt"** *#需要下载的文件* file\_handle=open(filename,**"wb"**).write *#以写模式在本地打开文件* ftp.retrbinaly(**"RETR filename.txt"**,file\_handle,bufsize) *#接收服务器上文件并写入本地文件* ftp.set\_debuglevel(0) *#关闭调试模式* ftp.quit() *#退出ftp  #ftp相关命令操作* ftp.cwd(pathname) *#设置FTP当前操作的路径* ftp.dir() *#显示目录下所有目录信息* ftp.nlst() *#获取目录下的文件* ftp.mkd(pathname) *#新建远程目录* ftp.pwd() *#返回当前所在位置* ftp.rmd(dirname) *#删除远程目录* ftp.delete(filename) *#删除远程文件* ftp.rename(fromname, toname)*#将fromname修改名称为toname。* ftp.storbinaly(**"STOR filename.txt"**,file\_handle,bufsize) *#上传目标文件* ftp.retrbinary(**"RETR filename.txt"**,file\_handle,bufsize) *#下载FTP文件* |

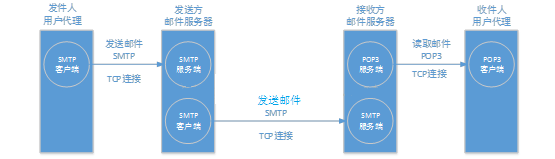
案例

|  |
| --- |
| **from** ctypes **import** \* **import** os **import** sys **import** ftplib   **class** MyFtp:  ftp = ftplib.FTP()  bIsDir = **False** path = **""   def** \_\_init\_\_(self, host, port=21):  self.ftp.set\_debuglevel(2) *# 打开调试级别2，显示详细信息  # self.ftp.set\_pasv(0) #0主动模式 1 #被动模式* self.ftp.connect(host, port) *# 连接IP和PORT* **def** login(self, user, passwd):  *# 登录ftp* self.ftp.login(user, passwd)  *# 打印欢迎信息* print(self.ftp.welcome)   **def** downLoadFile(self, localFile, remoteFile):  *# 打开一个本地文件句柄* **with** open(localFile, **'wb'**) **as** file\_handler:  *# 下载服务器的指定文件，保存为 localFile* self.ftp.retrbinary(**"RETR %s"** % (remoteFile), file\_handler.write)   **return True   def** upLoadFile(self, localFile, remoteFile):  *# 判断文件是否存在* **if** os.path.isfile(localFile) == **False**:  **return False  with** open(localFile, **"rb"**) **as** file\_handler:  *# 上传文件到服务器，如果文件太大，需要分段上传* self.ftp.storbinary(**'STOR %s'** % remoteFile, file\_handler, 4096)   **return True   def** upLoadFileTree(self, localDir, remoteDir):  *# 判断是否存在* **if** os.path.isdir(localDir) == **False**:  **return False** *# 获取目录下的所有文件和文件夹名字* localNames = os.listdir(localDir)  print(remoteDir)  *# 切换到服务器的指定目录中* self.ftp.cwd(remoteDir)  *# 循环遍历* **for** local **in** localNames:  *# 拼接 路径  # 不要 localDir + '/' + local* src = os.path.join(localDir, local)  *# 判断是否文件夹* **if** os.path.isdir(src):  self.upLoadFileTree(src, local)  **else**:  self.upLoadFile(src, local)  self.ftp.cwd(**".."**)  **return   def** downLoadFileTree(self, localDir, remoteDir):  *# 判断文件夹是否存在* **if** os.path.isdir(localDir) == **False**:  *# 没有则创建目录* os.makedirs(localDir)  *# 远程服务器端 切换到 remoteDir 目录* self.ftp.cwd(remoteDir)  *# 获取服务器当前目录的所有文件和文件夹名字* remoteNames = self.ftp.nlst()  *# 循环* **for** file **in** remoteNames:  *# 本地对应的文件目录路径* local = os.path.join(localDir, file)  *# 判断是否为文件夹* **if** self.isDir(file):  self.downLoadFileTree(local, file)  **else**:  *# 直接下载* self.downLoadFile(local, file)  self.ftp.cwd(**".."**)  **return   def** show(self, list):  result = list.lower().split(**" "**)  **if "<dir>" in** result:  self.bIsDir = **True   def** isDir(self, path):  self.bIsDir = **False** self.path = path  *# this ues callback function ,that will change bIsDir value* self.ftp.retrlines(**'LIST'**, self.show)  **return** self.bIsDir   **def** close(self):  self.ftp.quit()   ftp = MyFtp(**'localhost'**) ftp.login(ftp\_user.user, ftp\_user.pwd) *# ftp.downLoadFile('test.txt', 'others\\runtime.log')#ok # ftp.upLoadFile('ftp\_up.txt', 'others\\ftp\_up.txt')#ok* ftp.downLoadFileTree(**'bcd'**, **'others\\abc'**)*#ok # ftp.upLoadFileTree('aaa',"others\\" )* ftp.close() print(**"ok!"**) |

# 5SMTP

## 5.1简介

SMTP（Simple Mail TransferProtocol）即简单邮件传输协议，它是一种TCP协议支持的提供可靠且有效电子邮件传输的应用层协议。smtp服务器是遵循smtp协议的发送邮件服务器，当接收时作为smtp服务端，当发送时做smtp客户端。SMTP是一个推协议，它不允许根据需要从远程服务器上“拉”来消息。如果客户使用邮件客户端收取邮件，需要使用POP3或IMAP协议，向邮件服务器拉取邮件数据，此时该服务器作为POP3或IMAP服务器。如下图所示：



说明：

a)上图读取邮件的箭头是指邮件的流向，该操作是由收件人主动发起的。

b)用户代理就是用户与电子邮件系统的接口，大多数情况下就是运行在用户PC机上的一个程序，如Foxmail。

c)发件人用户代理为何不直接将邮件发送给收件人的用户代理，而是通过邮件服务器来传输?

因为用户电脑、手机性能有限，无法运行收发邮件的程序，并且无法不间断地运行并连接到互联网上. 因此只能将信件暂时存放在邮件服务器中，用户需要时就可以去下载信件。

d)为什么用户收发邮件时感觉不到SMTP过程？

如果用户使用用户代理软件，则这个复杂的过程被用户代理屏蔽了，用户只需要进行简单的发送操作就可完成邮件的发送. 如果用户使用Web方式，则所有过程都由邮件服务器完成。

e)smtp一般不使用中间邮件服务器进行发送，如果接收方邮件服务器未准备好，发送方会保留邮件，稍后进行尝试。

## 5.2通讯过程

a)建立TCP连接

b)客户端发送HELO命令以标识发件人自己的身份，然后客户端发送MAIL命令；

       服务器端正希望以OK作为响应，表明准备接收

c)客户端发送RCPT命令，以标识该电子邮件的计划接收人，可以有多个RCPT行；

       服务器端则表示是否愿意为收件人接收邮件

d)协商结束，发送邮件，用命令DATA发送

e)以.表示结束输入内容一起发送出去

f)结束此次发送，用QUIT命令退出

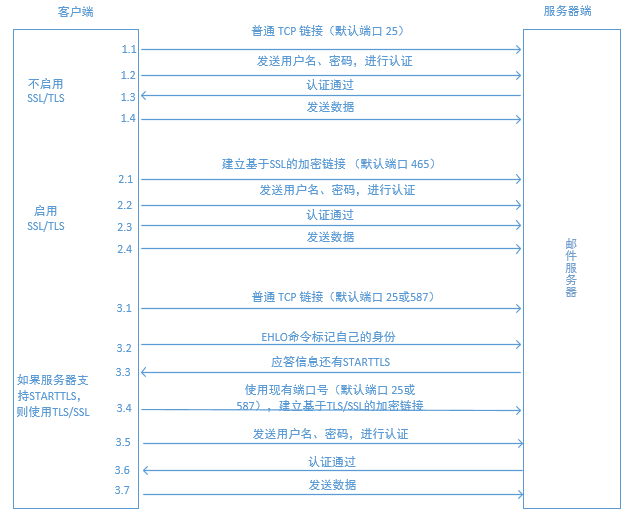
示例如下：

|  |
| --- |
| C: telnet SMTP.163.com 25 //以telenet方式连接163邮件服务器  S: 220 163.com Anti-spam GT for Coremail System //220为响应数字，其后的为欢迎信息  C: HELO SMTP.163.com //除了HELO所具有的功能外，EHLO主要用来查询服务器支持的扩充功能  S: 250-mail  S: 250-AUTH LOGIN PLAIN  S: 250-AUTH=LOGIN PLAIN  S: 250 8BITMIME //最后一个响应数字应答码之后跟的是一个空格，而不是'-'  C: AUTH LOGIN //请求认证  S: 334 dxNlcm5hbWU6 //服务器的响应——经过base64编码了的“Username”=  C: Y29zdGFAYW1heGl0Lm5ldA== //发送经过BASE64编码了的用户名  S: 334 UGFzc3dvcmQ6 //经过BASE64编码了的"Password:"=  C: MTk4MjIxNA== //客户端发送的经过BASE64编码了的密码  S: 235 auth successfully //认证成功  C: MAIL FROM: bripengandre@163.com //发送者邮箱  S: 250 … . //“…”代表省略了一些可读信息  C: RCPT TO: bripengandre@smail.hust.edu.cn　//接收者邮箱  S: 250 … . // “…”代表省略了一些可读信息  C: DATA //请求发送数据  S: 354 Enter mail, end with "." on a line by itself  C: Enjoy Protocol Studing  C: .  S: 250 Message sent  C: QUIT //退出连接  S: 221 Bye |

说明：

1)25端口是SMTP的标准服务端口，数据不经过加密处理，以明文方式发送。

当然也支持加密方式SSL/TLS，另有些发送邮箱服务器支持STARTTLS协议。下图是他们之间的区别（并没有详细描述中间的交互过程）



需要注意的是：

a)不加密默认使用25端口；如明确使用SSL/TLS加密，则使用465端口

b)在启用STARTTLS协议的时候，使用25端口较好，虽然587主要是针对STARTTLS协议的，但支持该协议的并不一定开放该端口。

c)STARTTLS协议不是说只能用TLS加密，也可以使用SSL加密。

2)标识自己的身份

可通过HELO命令和EHLO命令标识自己的身份，两者的区别是：

HELO命令可以不进行身份认证（除非服务器要求）；

EHLO命令则需要进行身份认证，并会返回更多的信息给客户端，如是否支持starttls协议。

如下面所示：

220 smtp.example.com ESMTP Service(Lotus Domino Release 8.5.3FP6) ready a

t Thu, 23 Jul 2015 15:58:58 +0800

EHLO zjf

250-smtp.example.com Hello zjf([10.10.34.22]), pleased to meet you

250-TLS

250-HELP

250-AUTH LOGIN

250-STARTTLS < ------------------------- 说明支持starttls协议

250-SIZE 18432000

250 PIPELINING

可见要想知道一个邮件服务器是否支持ssl/tls、starttls，可通过telnet登入相应的端口号，和使用EHLO命令来查看。

## 5.3常用命令

SMTP命令不区分大小写，但参数区分大小写。常用命令如下：

HELO <domain> <CRLF>——向服务器标识用户身份发送者能欺骗、说谎，但一般情况下服务器都能检测到

RCPT TO: <forward-path> <CRLF>——<forward-path>用来标志邮件接收者的地址，常用在MAIL FROM后，可以有多个RCPT TO

DATA <CRLF>——将之后的数据作为数据发送，以<CRLF>.<CRLF>标志数据的结尾

REST <CRLF>——重置会话，当前传输被取消

NOOP <CRLF>——要求服务器返回OK应答，一般用作测试

QUIT <CRLF>——结束会话

VRFY <string> <CRLF>——验证指定的邮箱是否存在，由于安全方面的原因，服务器大多禁止此命令

EXPN <string> <CRLF>——验证给定的邮箱列表是否存在，由于安全方面的原因，服务器大多禁止此命令

HELP <CRLF>——查询服务器支持什么命令

## 5.4常用响应

501——参数格式错误

502——命令不可实现

503——错误的命令序列

504——命令参数不可实现

211——系统状态或系统帮助响应

214——帮助信息

220<domain>——服务器就绪

221<domain>——服务关闭

421<domain>——服务器未就绪，关闭传输信道

250——要求的邮件操作完成

251——用户非本地，将转发向<forward-path>

450——要求的邮件操作未完成，邮箱不可用

550——要求的邮件操作未完成，邮箱不可用

451——放弃要求的操作，处理过程中出错

551——用户非本地，请尝试<forward-path>

452——系统存储不足，要求的操作未执行

552——过量的存储分配，要求的操作未执行

553——邮箱名不可用，要求的操作未执行

354——开始邮件输入，以“.”结束

554——操作失败

## 5.5 案例

文本邮件

|  |
| --- |
| **from** email.header **import** Header **from** email.mime.text **import** MIMEText **from** email.utils **import** parseaddr, formataddr  **import** smtplib  **def** \_format\_addr(s):  name, addr = parseaddr(s)  **return** formataddr((Header(name, **'utf-8'**).encode(), addr))  from\_addr = **'auratest2018@163.com'** password = **'auratest2016'** to\_addr = **'auratest2018@163.com'** smtp\_server = **'smtp.163.com'** *# 构建一个文本的mail对象* msg = MIMEText(**'hello, send by Python...'**, **'plain'**, **'utf-8'**) msg[**'From'**] = \_format\_addr(**'Python爱好者 <%s>'** % from\_addr) msg[**'To'**] = \_format\_addr(**'管理员 <%s>'** % to\_addr) msg[**'Subject'**] = Header(**'来自SMTP的问候……'**, **'utf-8'**).encode()  *# 构建一个 smtp 对象* server = smtplib.SMTP(smtp\_server, 25) *# 设置一个调试级别* server.set\_debuglevel(1) *# 登录* server.login(from\_addr, password) *# 发送邮件* server.sendmail(from\_addr, [to\_addr], msg.as\_string()) server.quit() |

Html邮件

|  |
| --- |
| **from** email.header **import** Header **from** email.mime.text **import** MIMEText **from** email.utils **import** parseaddr, formataddr  **import** smtplib  **def** \_format\_addr(s):  name, addr = parseaddr(s)  **return** formataddr((Header(name, **'utf-8'**).encode(), addr))  from\_addr = **'auratest2018@163.com'** password = **'auratest2016'** to\_addr = **'auratest2018@163.com'** smtp\_server = **'smtp.163.com'** *# 读取html文件内容* **with** open(**'login.html'**, **'rb'**) **as** f:  mail\_body = f.read()  f.close()  msg = MIMEText(mail\_body, **'html'**, **'utf-8'**) msg[**'From'**] = \_format\_addr(**'Python爱好者 <%s>'** % from\_addr) msg[**'To'**] = \_format\_addr(**'管理员 <%s>'** % to\_addr) msg[**'Subject'**] = Header(**'来自SMTP的问候……'**, **'utf-8'**).encode()  server = smtplib.SMTP(smtp\_server, 25) server.set\_debuglevel(1) server.login(from\_addr, password) server.sendmail(from\_addr, [to\_addr], msg.as\_string()) server.quit() |

附件邮件

|  |
| --- |
| **from** email.header **import** Header **from** email.mime.text **import** MIMEText **from** email.utils **import** parseaddr, formataddr **from** email.mime.image **import** MIMEImage **from** email.mime.multipart **import** MIMEMultipart **import** smtplib  **def** \_format\_addr(s):  name, addr = parseaddr(s)  **return** formataddr((Header(name, **'utf-8'**).encode(), addr))  from\_addr = **'auratest2018@163.com'** password = **'auratest2016'** to\_addr = **'auratest2018@163.com'** smtp\_server = **'smtp.163.com'** msg = MIMEMultipart()  msg\_text = MIMEText(**'hello, send by Python...'**, **'plain'**, **'utf-8'**) msg.attach(msg\_text)  file = **'1.png'** fp = open(**'1.png'**, **'rb'**) msgImage = MIMEImage(fp.read()) fp.close() msgImage.add\_header(**'Content-ID'**, **'attachment'**, filename = file) msg.attach(msgImage)  msg[**'From'**] = \_format\_addr(**'Python爱好者 <%s>'** % from\_addr) msg[**'To'**] = \_format\_addr(**'管理员 <%s>'** % to\_addr) msg[**'Subject'**] = Header(**'来自SMTP的问候……'**, **'utf-8'**).encode()  server = smtplib.SMTP(smtp\_server, 25) server.set\_debuglevel(1) server.login(from\_addr, password) server.sendmail(from\_addr, [to\_addr], msg.as\_string()) server.quit() |

# 6 POP3

## 6.1简介

POP3是Post Office Protocol 3的简称，即邮局协议的第3个版本,它规定怎样将个人计算机连接到Internet的邮件服务器和下载电子邮件的电子协议。它是因特网电子邮件的第一个离线协议标准,POP3允许用户从服务器上把邮件存储到本地主机（即自己的计算机）上,同时删除保存在邮件服务器上的邮件，而POP3服务器则是遵循POP3协议的接收邮件服务器，用来接收电子邮件的。

## 6.2命令和响应

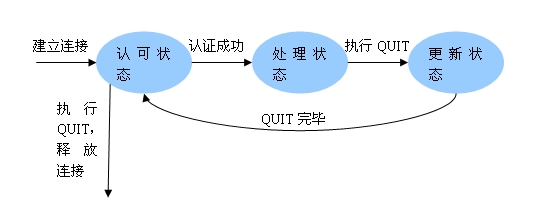
### 6.2.1格式

POP3的命令不多，它的一般形式是：COMMAND [Parameter] <CRLF>。其中COMMAND是ASCII形式的命令名，Parameter是相应的命令参数，<CRLF>是回车换行符(0DH, 0AH)。

服务器响应是由一个单独的命令行组成，或多个命令行组成，响应第一行“+OK”或“-ERR”开头，然后再加上一些ASCII文本。“+OK”和“-ERR”分别指出相应的操作状态是成功的还是失败的。

### 6.2.2三个状态

POP3协议中有三种状态，认证状态，处理状态，和更新状态。 命令的执行可以改变协议的状态，而对于具体的某命令，它只能在具体的某状态下使用。客户机与服务器刚与服务器建立连接时，它的状态为认证状态；一旦客户机提供了自己身份并被成功地确认，即由认可状态转入处理状态； 在完成相应的操作后客户机发出QUIT命令（具体说明见后续内容），则进入更新状态，更新之后又重返认可状态；当然在认可状态下执行QUIT命令，可释放连接。



## 6.3通讯过程

用户从邮件服务器上接收邮件的典型通信过程如下。

1)用户运行用户代理(如Foxmail, Outlook Express)。

2)客户端与服务器端的110端口建立TCP连接。

3)客户端向服务器端发出各种命令，来请求各种服务(如查询邮箱信息，下载某封邮件等)。

4)服务端解析用户的命令，做出相应动作并返回给客户端一个响应。

5)3)和4)交替进行，直到接收完所有邮件转到步骤6)，或两者的连接被意外中断而直接退出。

1. 用户代理解析从服务器端获得的邮件，以适当地形式(如可读)的形式呈现给用户。

示例如下：

|  |
| --- |
| C：telnet pop3.126.com 110 /\*　以telnet方式连接126邮件服务器　\*/  S：+OK Welcome to coremail Mail Pop3 Server (126coms[3adb99eb4207ae5256632eecb8f8b4855])　/\* +OK,代表命令成功，其后的信息则随服务器的不同而不同\*/  C：USER bripengandre /\* 采用明文认证 \*/  S：+OK core mail  C：PASS Pop3world /\* 发送邮箱密码 \*/  S：+OK 654 message(s) [30930370 byte(s)] /\* 认证成功，转入处理状态 \*/  C：LIST 1 /\* 显示第一封邮件的信息 \*/  S：+OK 1 5184 ./\* 第一封邮件的大小为5184 字节 \*/  C：UIDL 1 /\* 返回第一封邮件的唯一标识符 \*/  S：+OK 1 1tbisBsHaEX9byI9EQAAsd /\* 数字1 后的长字符串就是第一封邮件的唯一标志符 \*/  C：RETR 1 /\* 下载第一封邮件 \*/  S：+OK 5184 octets /\* 第一封邮件的大小为5184字节 \*/  S：Receive… /\* 第一封邮件的具体内容 \*/  S：…  C：QUIT /\* 转入更新状态，接着再转入认证状态 \*/  S：+OK  C: QUIT /\* 退出连接 \*/  S：+OK core mail /\* 成功地退出了连接 \*/ |

## 6.4常用命令

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 参数 | 状态 | 描述 |
| USER | username | 认可 | 此命令与下面的pass命令若成功，将导致状态转换 |
| PASS | password | 认可 |  |
| APOP | Name , Digest | 认可 | Digest是MD5消息摘要 |
| STAT | None | 处理 | 请求服务器发回关于邮箱的统计资料，如邮件总数和总字节数 |
| UIDL | [ Msg # ] | 处理 | 返回邮件的唯一标识符，POP3会话的每个标识符都将是唯一的 |
| LIST | [ Msg # ] | 处理 | 返回邮件数量和每个邮件的大小 |
| RETR | [ Msg # ] | 处理 | 返回由参数标识的邮件的全部文本 |
| DELE | [ Msg # ] | 处理 | 服务器将由参数标识的邮件标记为删除，由quit命令执行 |
| RSET | None | 处理 | 服务器将重置所有标记为删除的邮件，用于撤消DELE命令 |
| TOP | [ Msg # ] | 处理 | 服务器将返回由参数标识的邮件前n行内容，n必须是正整数 |
| NOOP | None | 处理 | 服务器返回一个肯定的响应 |
| QUIT | None | 更新 |  |

## 6.5案例

|  |
| --- |
| **import** poplib **import** sys **from** importlib **import** reload **from** email.parser **import** Parser **from** email.parser **import** BytesParser **from** email.header **import** decode\_header **from** email.utils **import** parseaddr **import** email.iterators  *#解析消息头中的字符串* **def** decode\_str(s):  value, charset = decode\_header(s)[0]  **if** charset:  value = value.decode(charset)  **return** value  *#将邮件附件或内容保存至文件 #即邮件中的附件数据写入附件文件* **def** savefile(filename, data, path):  **try**:  filepath = path + filename  print(**'Save as: '** + filepath)  f = open(filepath, **'wb'**)  **except**:  print(filepath + **' open failed'**)  *#f.close()* **else**:  f.write(data)  f.close()  *#获取邮件的字符编码，首先在message中寻找编码，如果没有，就在header的Content-Type中寻找* **def** guess\_charset(msg):  charset = msg.get\_charset()  **if** charset **is None**:  content\_type = msg.get(**'Content-Type'**, **''**).lower()  pos = content\_type.find(**'charset='**)  **if** pos >= 0:  charset = content\_type[pos+8:].strip()  **return** charset  *#解析邮件的函数，首先打印收件人、发件人、标题 #然后调用message的walk循环处理邮件中的每一个子对象（包括文本、html、附件一次或多次） #邮件头属性中的filename存在则该子对象是附件，对附件名称进行编码并将附件下载到指定目录 #由于网络上传输的邮件都是编码以后的格式，需要在get\_payload的时候指定decode=True来转换成可输出的编码 #如果邮件是text或者html格式，打印格式并输出转码以后的子对象内容* **def** print\_info(msg):  **for** header **in** [**'From'**, **'To'**, **'Subject'**]:  value = msg.get(header, **''**)  **if** value:  **if** header == **'Subject'**:  value = decode\_str(value)  **else**:  hdr, addr = parseaddr(value)  name = decode\_str(addr)  value = name + **' < '** + addr + **' > '** print(header + **':'** + value)  **for** part **in** msg.walk():  filename = part.get\_filename()  content\_type = part.get\_content\_type()  charset = guess\_charset(part)  **if** filename:  filename = decode\_str(filename)  data = part.get\_payload(decode = **True**)  **if** filename != **None or** filename != **''**:  print(**'Accessory: '** + filename)  savefile(filename, data, mypath)  **else**:  email\_content\_type = **''** content = **''  if** content\_type == **'text/plain'**:  email\_content\_type = **'text'  elif** content\_type == **'text/html'**:  email\_content\_type = **'html'  if** charset:  content = part.get\_payload(decode=**True**).decode(charset)  print(email\_content\_type + **' '** + content)  email = **'myit\_test02@163.com'** password = **'y666666'** pop3\_server = **'pop.163.com'** mypath = **'D://email/'** server = poplib.POP3(pop3\_server, 110) *#print(server.getwelcome())* server.user(email) server.pass\_(password) print(**'Message: %s. Size: %s'** % server.stat())  resp, mails, objects = server.list() *#print(mails)* index = len(mails) *#取出某一个邮件的全部信息* resp, lines, octets = server.retr(index) *#邮件取出的信息是bytes，转换成Parser支持的str* lists = [] **for** e **in** lines:  lists.append(e.decode()) msg\_content = **'\r\n'**.join(lists) msg = Parser().parsestr(msg\_content) print\_info(msg) *#server.dele(index) #提交操作信息并退出* server.quit() |

# 7HTTP

## 7.1简介

超文本传输协议(HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。所有的WWW文件都必须遵守这个标准。设计HTTP最初的目的是为了提供一种发布和接收HTML页面的方法。

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过几年的使用与发展，得到不断地完善和扩展。在2015年已推出HTTP/2版本，并被主要的web浏览器和web服务容器支持。但目前使用最广泛的还是HTTP/1.1版本。

HTTP协议的主要特点可概括如下：

1)支持客户/服务器模式。

2)简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST、OPTIONS。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

3)灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

4)无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

5)无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

## 7.2URL

URL(Uniform Resource Locator,统一资源定位符)是对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

HTTP 是一个基于请求与响应模式的、无状态的、应用层的协议，常基于TCP的连接方式，HTTP1.1版本中给出一种持续连接的机制，绝大多数的Web开发，都是构建在HTTP协议之上的Web应用。

HTTP URL (URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：

http://host[":"port][abs\_path]?参数A=value1&参数2=value2

https://host[":"port][abs\_path] ?参数A=value1&参数2=value2

http表示要通过HTTP协议来定位网络资源；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用缺省端口80(https是443)；abs\_path指定请求资源的URI；如果URL中没有给出abs\_path，那么当它作为请求URI时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

eg:

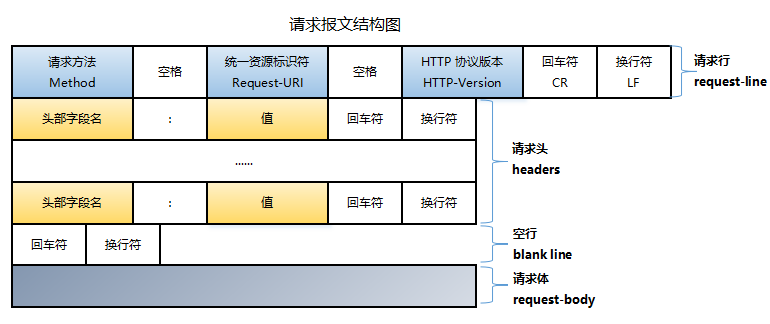
a)输入：www.guet.edu.cn

浏览器自动转换成：http://www.guet.edu.cn/

b)http:192.168.0.116:8080/index.jsp

## 7.3http请求

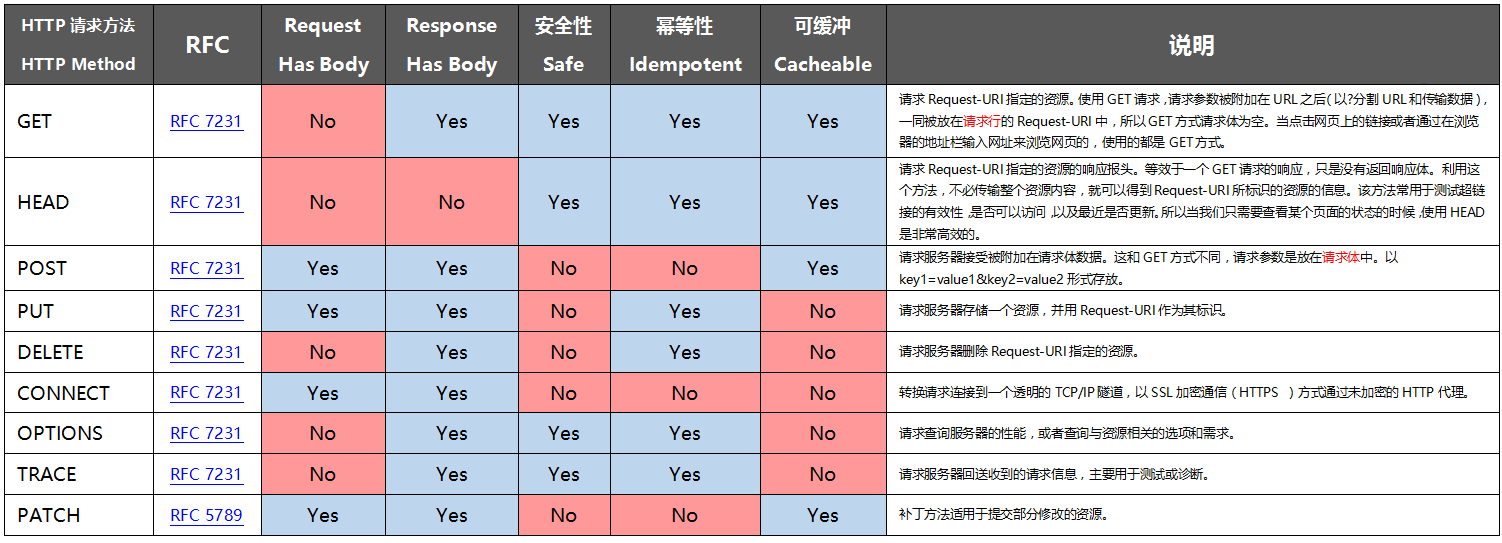
http请求由三部分组成，分别是：请求行、消息报头、请求正文



**请求行**以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：Method Request-URI HTTP-Version CRLF

其中 Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

**请求方法**（所有方法全为大写）有多种，各个方法的解释如下：



关于HTTP请求GET和POST的区别

(1)提交形式：

GET提交的数据会放在URL之后，以?分割URL和传输数据，参数之间以&相连，如EditPosts.aspx?name=test1&id=123456.

POST方法是把提交的数据放在HTTP包的Body中.

(2)传输数据的大小：

HTTP协议本身没有对传输的数据大小进行限制，HTTP协议规范也没有对URL长度进行限制。 而在实际开发中存在的限制主要有：

   GET:特定浏览器和服务器对URL长度有限制，例如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。

   因此对于GET提交时，传输数据就会受到URL长度的限制。

   POST:由于不是通过URL传值，理论上数据不受限。但实际各个WEB服务器会规定对post提交数据大小进行限制，Apache、IIS6都有各自的配置。

(3)安全性：

    POST的安全性要比GET的安全性高，具有真正的Security的含义。而且通过GET提交数据，用户名和密码将明文出现在URL上，因为登录页面有可能被浏览器缓存，其他用户浏览历史纪录就可以拿到账号和密码了。

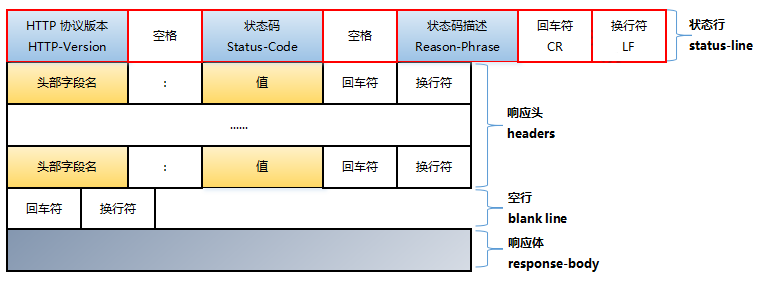
**请求头**



## 7.4http响应

在接收和解释请求消息后，服务器返回一个HTTP响应消息。

HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文



状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

1xx：指示信息--表示请求已接收，继续处理

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--要完成请求必须进行更进一步的操作

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

常见状态代码、状态描述、说明：

200 OK //客户端请求成功

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

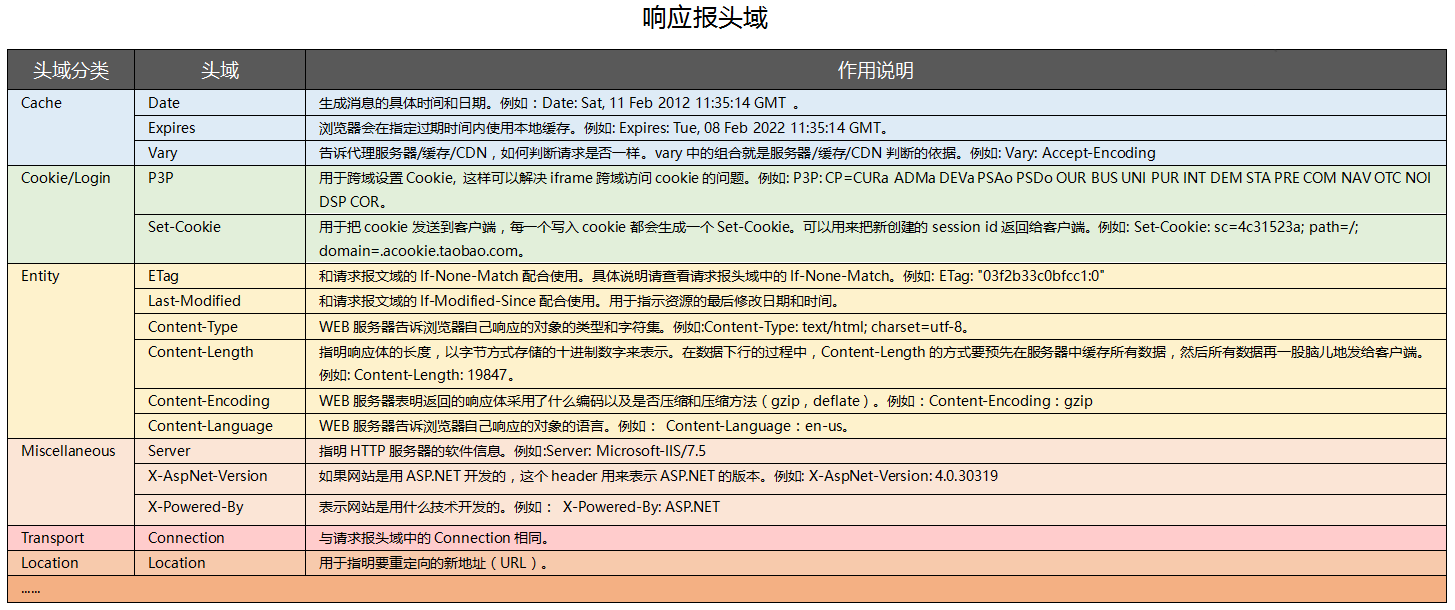
404 Not Found //请求资源不存在，eg：输入了错误的URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

eg：HTTP/1.1 200 OK (CRLF)

**响应报头域**

****

## 7.5案例

利用python自带的库模拟http请求，为以后利用python做API测试做准备。只讲述模拟http的过程，具体到自己用的时候，要以自己的应用为准做出适当的调整。

http是一个包，里面含有多个模块：http.client，http.server，http.cookies，http.cookiejar。

客户端:

Get请求

|  |
| --- |
| **import** http.client conn = http.client.HTTPSConnection(**"www.python.org"**) conn.request(**"GET"**, **"/"**) r1 = conn.getresponse() print(r1.status, r1.reason) data1 = r1.read() *# This will return entire content. # The following example demonstrates reading data in chunks.* conn.request(**"GET"**, **"/"**) r1 = conn.getresponse() **while not** r1.isclosed():  print(r1.read(200)) *# 200 bytes  # Example of an invalid request* conn.request(**"GET"**, **"/parrot.spam"**) r2 = conn.getresponse() print(r2.status, r2.reason)  data2 = r2.read() conn.close() |

Post请求

|  |
| --- |
| **import** http.client, urllib.parse params = urllib.parse.urlencode({**'@number'**: 12524, **'@type'**: **'issue'**, **'@action'**: **'show'**}) headers = {**"Content-type"**: **"application/x-www-form-urlencoded"**,  **"Accept"**: **"text/plain"**} conn = http.client.HTTPConnection(**"bugs.python.org"**) conn.request(**"POST"**, **""**, params, headers) response = conn.getresponse() print(response.status, response.reason)  data = response.read() print(data) conn.close() |

# 8框架入门

Web程序

## Web程序的本质

对于所有的Web应用，本质上其实就是一个socket服务端，用户的浏览器其实就是一个socket客户端。

|  |
| --- |
| **import** socket  **def** handle\_request(client):  buf = client.recv(1024)  client.send(**b"HTTP/1.1 200 OK\r\n\r\n"**)  client.send(**b"Hello, web"**)  **def** main():  sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  sock.bind((**'localhost'**, 8000))  sock.listen(5)   **while True**:  connection, address = sock.accept()  handle\_request(connection)  connection.close()  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  main() |

上述通过socket来实现了其本质，而对于真实开发中的python web程序来说，一般会分为两部分：服务器程序和应用程序。服务器程序负责对socket服务器进行封装，并在请求到来时，对请求的各种数据进行整理。应用程序则负责具体的逻辑处理。为了方便应用程序的开发，就出现了众多的Web框架，例如：Django、Flask、web.py 等。不同的框架有不同的开发方式，但是无论如何，开发出的应用程序都要和服务器程序配合，才能为用户提供服务。这样，服务器程序就需要为不同的框架提供不同的支持。这样混乱的局面无论对于服务器还是框架，都是不好的。对服务器来说，需要支持各种不同框架，对框架来说，只有支持它的服务器才能被开发出的应用使用。这时候，标准化就变得尤为重要。我们可以设立一个标准，只要服务器程序支持这个标准，框架也支持这个标准，那么他们就可以配合使用。一旦标准确定，双方各自实现。这样，服务器可以支持更多支持标准的框架，框架也可以使用更多支持标准的服务器。

## 常见的web服务器标准

### CGI

通用网关接口（Common Gateway Interface/CGI）描述了客户端和服务器程序之间传输数据的一种标准，可以让一个客户端，从网页浏览器向执行在网络服务器上的程序请求数据。CGI 独立于任何语言的，CGI 程序可以用任何脚本语言或者是完全独立编程语言实现，只要这个语言可以在这个系统上运行。Unix shell script, Python, Ruby, PHP, perl, Tcl, C/C++, 和 Visual Basic 都可以用来编写 CGI 程序。最初，CGI 是在 1993 年由美国国家超级电脑应用中心（NCSA）为 NCSA HTTPd Web 服务器开发的。这个 Web 服务器使用了 UNIX shell 环境变量 来保存从 Web 服务器传递出去的参数，然后生成一个运行 CGI 的独立的进程。

### FastCGI

快速通用网关接口（Fast Common Gateway Interface／FastCGI）是通用网关接口（CGI）的改进，描述了客户端和服务器程序之间传输数据的一种标准。FastCGI致力于减少Web服务器与CGI程式之间互动的开销，从而使服务器可以同时处理更多的Web请求。与为每个请求创建一个新的进程不同，FastCGI使用持续的进程来处理一连串的请求。这些进程由FastCGI进程管理器管理，而不是web服务器。

### WSGI

Web服务器网关接口（Python Web Server Gateway Interface，缩写为WSGI）是为Python语言定义的Web服务器和Web应用程序或框架之间的一种简单而通用的接口。自从WSGI被开发出来以后，许多其它语言中也出现了类似接口。WSGI是作为Web服务器与Web应用程序或应用框架之间的一种低级别的接口，以提升可移植Web应用开发的共同点。WSGI是基于现存的CGI标准而设计的。

WSGI区分为两个部份：一为“服务器”或“网关”，另一为“应用程序”或“应用框架”。在处理一个WSGI请求时，服务器会为应用程序提供环境资讯及一个回呼函数（Callback Function）。当应用程序完成处理请求后，透过前述的回呼函数，将结果回传给服务器。所谓的 WSGI 中间件同时实现了API的两方，因此可以在WSGI服务和WSGI应用之间起调解作用：从WSGI服务器的角度来说，中间件扮演应用程序，而从应用程序的角度来说，中间件扮演服务器。“中间件”组件可以执行以下功能：

重写环境变量后，根据目标URL，将请求消息路由到不同的应用对象。

允许在一个进程中同时运行多个应用程序或应用框架。

负载均衡和远程处理，通过在网络上转发请求和响应消息。

进行内容后处理，例如应用XSLT样式表。

以前，如何选择合适的Web应用程序框架成为困扰Python初学者的一个问题，这是因为，一般而言，Web应用框架的选择将限制可用的Web服务器的选择，反之亦然。那时的Python应用程序通常是为CGI，FastCGI，mod\_python中的一个而设计，甚至是为特定Web服务器的自定义的API接口而设计的。WSGI没有官方的实现, 因为WSGI更像一个协议。只要遵照这些协议,WSGI应用(Application)都可以在任何服务器(Server)上运行, 反之亦然。WSGI就是Python的CGI包装，相对于Fastcgi是PHP的CGI包装。

WSGI将 web 组件分为三类： web服务器，web中间件,web应用程序， wsgi基本处理模式为 ： WSGI Server -> (WSGI Middleware)\* -> WSGI Application

WSGI是一种规范，它定义了使用python编写的web app与web server之间接口格式，实现web app与web server间的解耦。

python标准库提供的独立WSGI服务器称为wsgiref。

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server  **def** RunServer(environ, start\_response):  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [bytes(**'<h1>Hello, web!</h1>'**, encoding=**'utf-8'**), ]  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  httpd = make\_server(**''**, 8000, RunServer)  print(**"Serving HTTP on port 8000..."**)  httpd.serve\_forever() |

## 自定义框架

a)通过python标准库提供的wsgiref模块开发一个自己的Web框架。

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server  **def** index():  **return** [bytes(**'<h1>index!</h1>'**, encoding=**'utf-8'**), ]  **def** login():  **return** [bytes(**'<h1>login!</h1>'**, encoding=**'utf-8'**), ]  **def** routers():  urlpatterns = (  (**'/index/'**, index),  (**'/login/'**, login),  )  **return** urlpatterns  **def** RunServer(environ, start\_response):  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  url = environ[**'PATH\_INFO'**]  urlpatterns = routers()  func = **None  for** item **in** urlpatterns:  **if** item[0] == url:  func = item[1]  **break  if** func:  **return** func()  **else**:  **return** [bytes('<h1>404 error!</h1>', encoding='utf-8') ]  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  httpd = make\_server(**''**, 8000, RunServer)  print(**"Serving HTTP on port 8000...")** httpd.serve\_forever() |

b)模板引擎

在上一步骤中，对于所有的login、index均返回给用户浏览器一个简单的字符串，在现实的Web请求中一般会返回一个复杂的符合HTML规则的字符串，所以我们一般将要返回给用户的HTML写在指定文件中，然后再返回。如：

Index.html

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head lang="en"**>  <**meta charset="UTF-8"**>  <**title**></**title**> </**head**> <**body**>  <**h1**>Index</**h1**>  </**body**> </**html**> |

login.html

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head lang="en"**>  <**meta charset="UTF-8"**>  <**title**></**title**> </**head**> <**body**>  <**form**>  <**input type="text"** />  <**input type="text"** />  <**input type="submit"** />  </**form**> </**body**> </**html**> |

修改py 文件

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** index():  *# return 'index'* f = open(**'index.html'**)  data = f.read()  **return** [bytes(data, encoding=**'utf-8'**),]   **def** login():  *# return 'login'* f = open(**'login.html'**)  data = f.read()  **return** [bytes(data, encoding=**'utf-8'**),]   **def** routers():  urlpatterns = (  (**'/index/'**, index),  (**'/login/'**, login),  )   **return** urlpatterns   **def** run\_server(environ, start\_response):  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  url = environ[**'PATH\_INFO'**]  urlpatterns = routers()  func = **None  for** item **in** urlpatterns:  **if** item[0] == url:  func = item[1]  **break  if** func:  **return** func()  **else**:  **return** ['404 not found'.encode()]  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  httpd = make\_server(**''**, 8000, run\_server)  print( **"Serving HTTP on port 8000..."**)   httpd.serve\_forever() |