PYTHON 网络编程 QUICK START

目录

更改履历

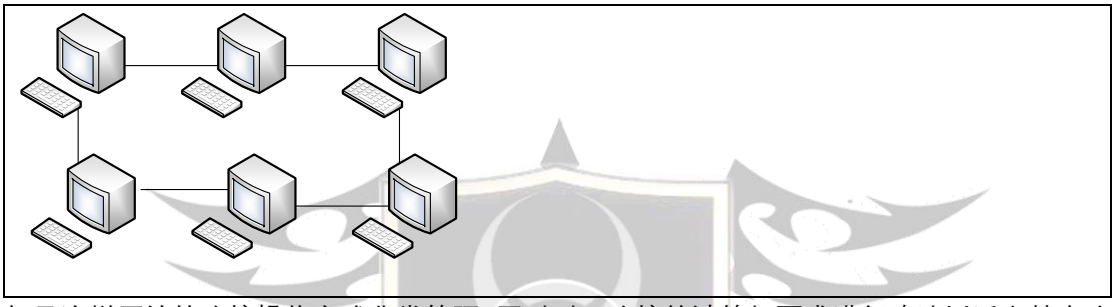
编辑	版本	时间	内容	备注
牟文斌	V1.0.0	2018-08-08	创建文档目录	
牟文斌	V2.0.0	2018-08-13	完成基础概念部分内容	
牟文斌	V2.1.0	2018-08-14	完成 TCP/UDP 编程部分内容	
牟文斌	V3.0.0	2018-08-15	完善 UDP 补充 socketserver 部分内容	

1.网络概述

1.1.概述

1.1.1. 什么是网络

早期的计算机都是互相独立的一台一台以数据运算为主的机器，随着时代的变迁，用于进行运算的数据的共享需求变得更加迫切，于是有人通过物理线路将多台计算机连接起来组成一个互联计算机平台，实现了多台计算机之间特定的数据交互模式，这就是最早期的网络和它的意义！

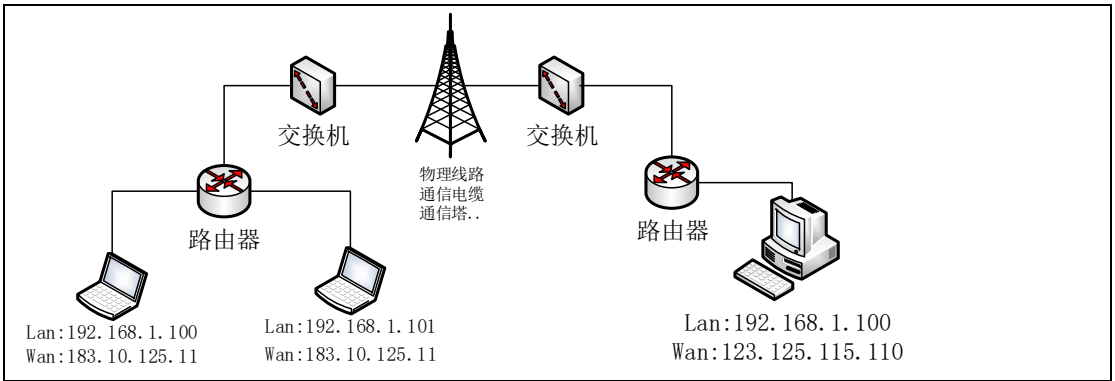


但是这样原始的连接操作方式非常简陋，同时对于连接的计算机要求进行改造以适应某个连接场合！极大程度的限制了数据共享的范围和计算机的普遍适用性能！

在为了迫切的数据通信共享需求的刺激下，计算机网络技术有了快速的发展同时也取得了非常有意义的成果，为了能在众多计算机中找到特定的计算机，Vint cerf 在实验室模拟阶段使用了 32 位标记的网络地址协议[internet protocal]，用于标识网络上唯一的一台计算机，也就是后来的 ip 地址；

为了能在一台计算机中找到特定的某个程序，将计算机中的每个和外界连接的程序设定了数据通信的通道，也就是后来经常听到的端口 port；

同一时间国际标准化组织 ISO 为了大家能方便的在网络上进行数据的传输通信，定义了网络数据传输模型，就是经典的 OSI/RM 参考模型，模型规范了适用于不同网络层级的传输协议，方便大家可以通过不同的协议进行数据的有效传播和共享！



REMARK：根据网络部署规模，区分为局域网络，城域网络、广域网络

1.1.2. 客户端 & 服务端

什么是客户端？什么是服务端？
对于不同的人不同的角度理解起来会有很多种含义，首先先明确一下关于客户端和服务端的本身的意义：
在生活场景中，提供各种人们需要的服务的人群属于服务员，对应了软件中的服务端，如餐厅服务员提供就餐服务、售票员提供售票服务等等；另外一部分消费者人群是享受这些服务的，对应了软件中的客户端；理想情况下服务端要求二十四小时不间断提供服务，因为你根本不确定客户端会在什么时刻需要访问这些服务！

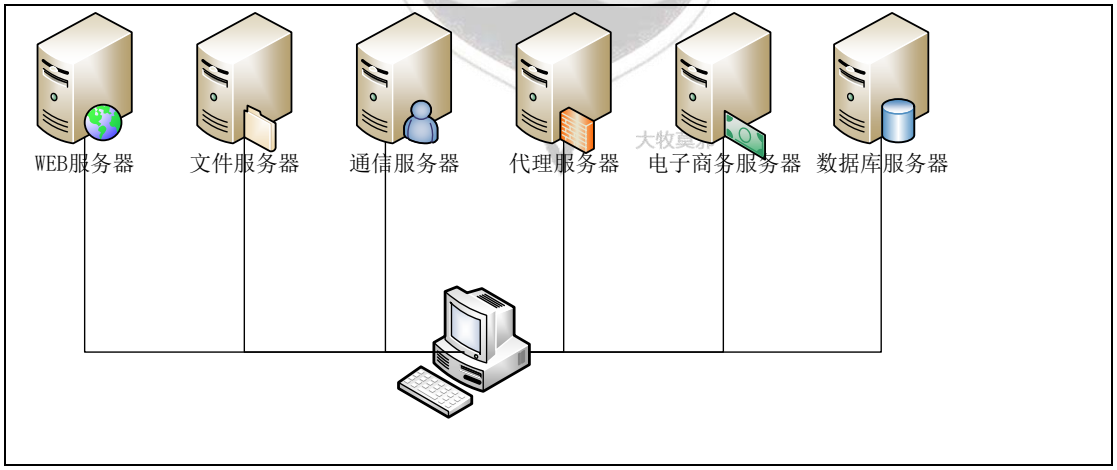
在计算机中，客户端和服务端的架构模式，区分为硬件模式和软件模式

- 硬件客户端/服务端

这种模式下，一台机器可以作为提供服务的机器，如同我们生活中的打印机就是一个硬件服务器，在计算机通过网络传输给打印机具体的数据时，打印机就会工作提供打印服务

更多的是运行在网络上的各种硬件服务器，如 CDN 服务器[网络文件分发服务器]，将我们需要的各种资源文件存储在网络上的一台主机中，在使用的时候只需要通过每个文件对应的一个 url 地址进行访问即可；如 FTP 服务器[文件传输服务器]，将我们的计算机文件可以存放在这样的服务上，并且在联网的情况下可以像操作本地计算机文件一样操作他们，给我们的生活带来了非常友好的帮助！

工作在网络上的硬件服务器，通常情况下也会根据在服务器上安装的特定软件和应用场景来命名不同性质的服务器名称，如：web 服务器、数据库服务器、代理服务器等等

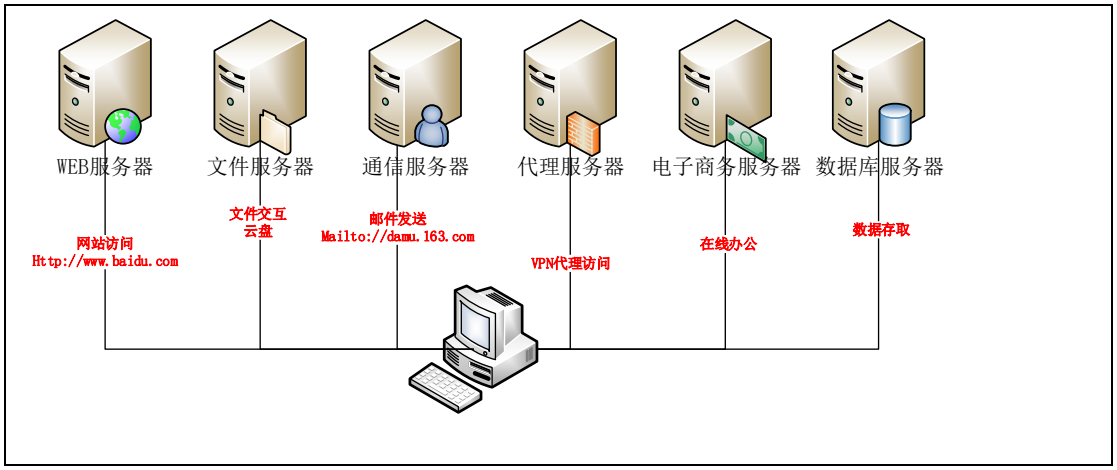


- 软件客户端/服务端

软件服务器和硬件服务器不同的是：软件服务器是安装在硬件上的一种特殊的软件，可以通过程序的执行完成数据检索、数据处理、数据存取等等各种特性，再通过网络进行数据的共享通信，完成提供网络数据服务的功能！

常见的如 web 软件服务器，主要是将一个服务器软件安装到工作在网络上的硬件主机上，然后就可以在 web 服务器中添加网站程序，启动一个网站来给用户提供各种服务操作；数

据库软件服务器，将数据库软件安装在硬件主机上，提供数据的存取服务等等



1.2.IP & PORT

1.2.1. ip 地址

ip: internet protocal 网络互联协议，中文缩写：网协；英文缩写：ip

网络互联协议就是为了多台计算机能够在网络中进行网络互联通信而设计的协议，是能够使得网络上工作的任意一台计算机可以和其他计算机计算机之间实现网络通信的一套通用规则，任何计算机制造厂商生产的计算机必须遵守这套规则的情况下，就可以接入现行的网络实现联网通信的功能!

- IP 地址根据使用的用户性质的不同，主要区分为 5 类 IP 地址

A类地址	0	网络地址7位	主机地址24位
B类地址	10	网络地址14位	主机地址16位
C类地址	110	网络地址21位	主机地址18位
D类地址	1110	多目的广播地址28位	
E类地址	11110	保留用于实验和将来使用	

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”， 地址范围从 1.0.0.0 到 126.0.0.0。可用的 A 类网络有 126 个，每个网络能容纳 1 亿多个主机。

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”，地址范围从 128.0.0.0 到 191.255.255.255。可用的 B 类网络有 16382 个，每个网络能容纳 6 万多个主机 。

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”。范围从 192.0.0.0 到 223.255.255.255。C 类网络可达 209 万余个，每个网络能容纳 254 个主机。

D 类地址用于多点广播（Multicast）。 D 类 IP 地址第一个字节以“lll0”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，它标识共享同一协议的一组计算机。224.0.0.0 到 239.255.255.255 用于多点广播 。

E 类 IP 地址 以“llll0”开始，为将来使用保留。240.0.0.0 到 255.255.255.254 255.255.255.255 用于广播地址

全零（“0 .0 .0 .0”）地址对应于当前主机。全“1”的 IP 地址（“255 .255 .255 .255”）是当前子网的广播地址。

- 私有地址：适用于局域网络的地址

在 IP 地址 3 种主要类型里，各保留了 3 个区域作为私有地址，其地址范围如下： A 类地址：10.0.0.0～10.255.255.255 B 类地址：172.16.0.0～172.31.255.255 C 类地址：192.168.0.0～192.168.255.255

- 本机地址：适用于当前主机进行网络回路测试

IP 地址为 127.0.0.1~127.255.255.255

> *So，出路就是 ipv6，你们都知道，我们几乎用尽了 ipv4 的地址空间，对此我感到有点尴尬，因为我就是决定 32 位 ip 地址足够因特网实验使用的那个人，我唯一能够辩驳的就是当时是在 1977 年作出的那个选择，并且当时我认为它仅仅是一个实验，然而，问题是这个实验并没有结束，所以我们才陷入了这个困境!*
> *——Vint Cerf 2011 年 1 月 linux.conf.au 会议*

- 子网掩码：专门针对不同类型的 IP 地址进行的网络地址和主机地址的规则定义

一个有效的 IP 地址，为了方便在网络上被寻址，通常情况包含三个部分：
类型标志
网络地址
主机地址

类型标志有着非常明确的规范，网络地址和主机地址的区分主要通过子网掩码进行划分
子网掩码和 IP 地址一样都是 32 位，左边为 1 用于表示网络地址，右边为 0 表示主机地址

```
某台主机 ip 地址：192.168.1.100
子网掩码：255.255.255.0
```

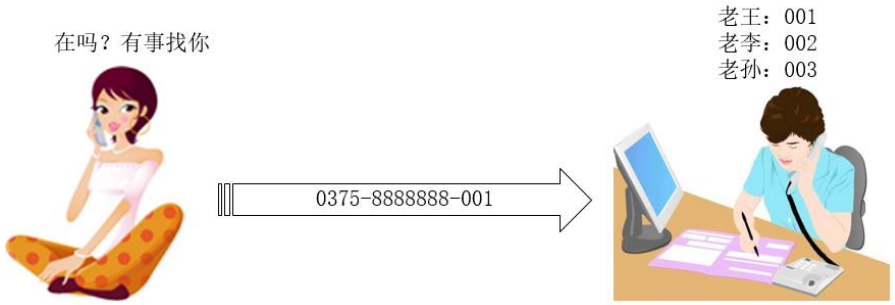
此时子网掩码对 IP 地址进行了划分，左边 24 位为网络地址，右边 8 位为主机地址
通常情况下我们所说的在不在一个网段，说的就是不是在同一个网络地址下!

1.2.2. port 端口

在通过 IP 地址确定了网络上的某个具体主机之后，具体的数据通信主要是通过工作在计算机中的软件执行的，不同的软件通信很容易造成问题

小丽发送的消息，具体会由老王的电脑上那个程序接收呢？

如果计算机中出现了不确定的因素，只能是设计人员/开发人员出现了设计上的问题!
想想我们生活中怎么执行这样的操作的



生活中给公司的某个人打座机电话，一般会在电话号码后面添加分机号用于直接找到对应的人

结合生活场景，计算机中出现了端口的概念，端口 port 主要是用于区别不同的软件的通信渠道，用于正确的将数据通过指定的端口渠道传输给对应的软件!

- 不同的端口号分类

计算机中的端口号的范围是 0~65535 之间

端口号根据其使用场景，一般区分为公用端口、动态端口、保留端口
公用端口：0~1023
动态端口：1024~65535
保留端口：一般是 unix 系统中超级用户进程分配保留端口号

- 常见端口号

端口号	***描述***
21	*FTP 文件传输端口*
22	*ssh 服务端口*
23	*telnet 服务端口*
25	*smtp 邮件服务端口*
80	*http 超文本传输协议端口*
110	*pop3 邮局协议端口*
115	*sftp 安全文件传输协议端口*
443	*https 安全超文本传输协议端口*
more:更多端口请参考网络资料 *remark:查看当前主机正在使用端口号: netstat -ano* *remark:查看当前主机中某个端口号是否被使用: netstat -ano | findstr* ***port***	

1.3.协议

1.3.1. 什么是协议

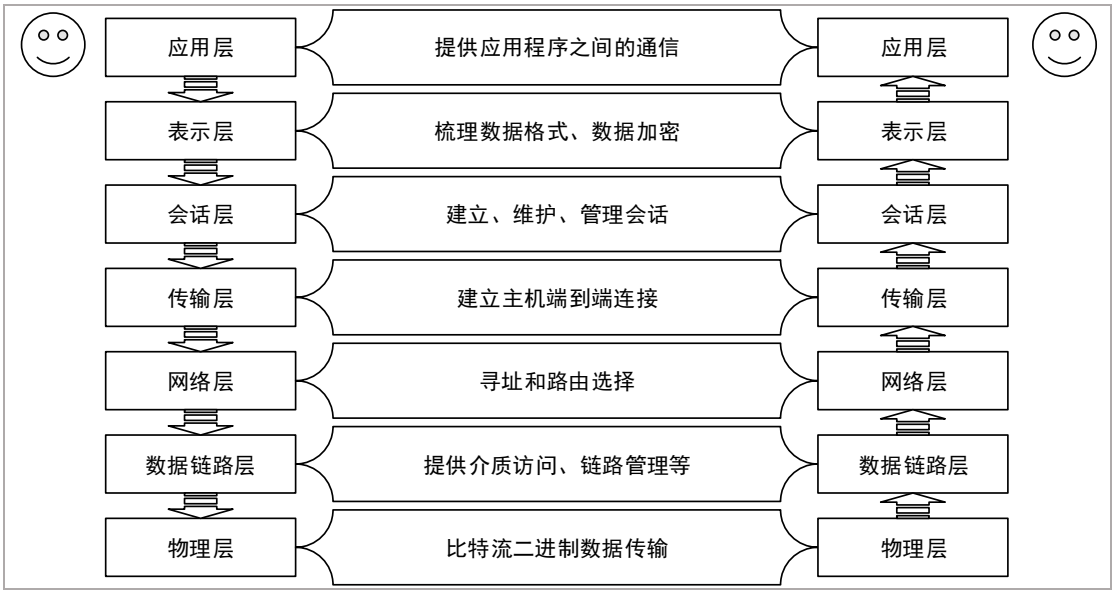
协议：英文名称 protocal，是多方协商计议之后得出的约定、规则、规范

通常情况下为了让工作在网络中的多台计算机之间能友好的完成不同软件的数据的通信，出现了互联网数据传输协议的概念，通过协议的约束，不同地域环境的计算机可以通过网络完成流畅的正确的数据交互。

计算机网络数据传输协议目前最主流的就是欧洲计算机制造协会联盟，也称为国际标准化组织 ISO 指定的 OSI/RM 七层网络传输模型!

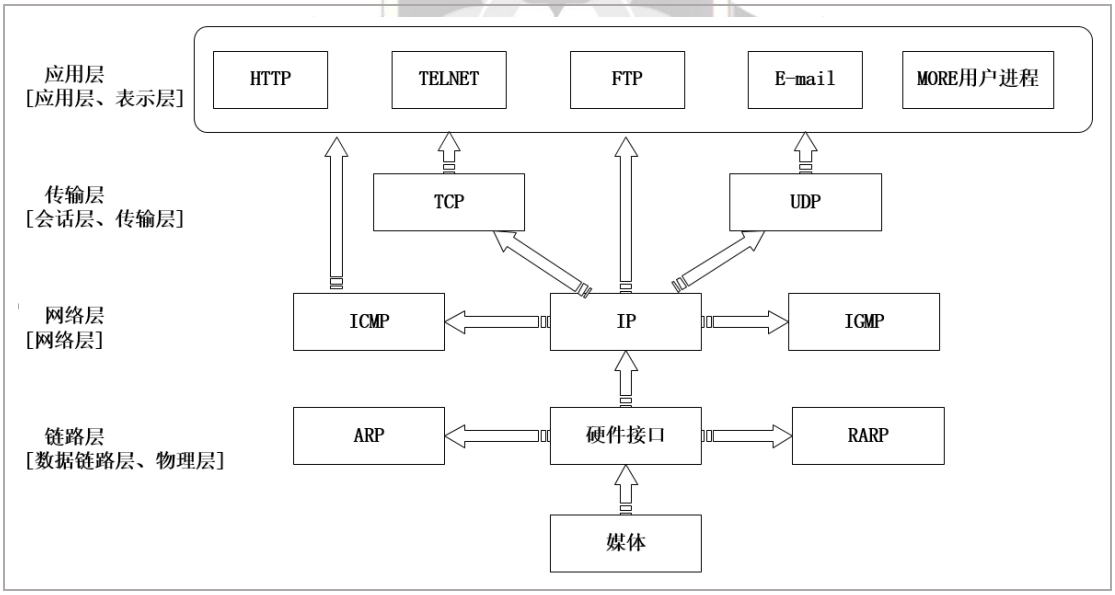
1.3.2. OSI/RM 模型

国际标准化组织根据用户从操作应用程序到数据经过物理线路的传输通信，将网络数据传输划分成了七层模型(也有五层模型、四层模型的概念，都是从七层模型的基础上再次抽象出来的)，通过不同层级模型的协议规范，将数据的传输进行了标准化处理，任何厂商生产的计算机都必须遵守这样的规范，才能在互联网中满足和其他计算机实现网络互联数据共享通信的功能



1.3.3. 协议族

协议通常指代单独的一个协议，协议族通常指代互相关联的一组协议，协议栈指代某一组互相关联的协议和他们所属的 OSI 模型的层级结构。



1.3.4. 常见网络传输协议

协议名称	*协议描述*
HTTP	*超文本传输协议*
HTTPS	*提供安全通道的超文本传输协议*
FTP	*文件传输协议*
TELNET	*虚拟终端协议*

SSH	*安全外壳协议*
POP3	*邮局协议(版本 3)*
SMTP	*简单邮件传输协议*
IP	*数据包交换协议*
TCP	*端对端传输协议*
UDP	*数据广播协议*
DNS	*域名解析协议，可以通过 nslookup 查看域名解析信息*
DHCP	*动态主机配置协议*

2. PYTHON 传输层网络编程

python2.x/python3.x 对于网络编程的支持都是非常友好的，本身支持两部分非常有用的网络编程方式

- 传统网络编程
- 非阻塞异步网络编程

底层通过套接字 socket 对象的连接，完成多种协议的网络程序服务端/客户端的开发和数据通信。

2.1.socket

2.1.1. socket 的由来

socket 是一种计算机网络数据结构，中文译名：套接字

20 世纪 70 年代，socket 是作为加利福尼亚大学的伯克利版本 Unix 系统的一部分出现的某些情况下也会被称为伯克利套接字或者 BSD 套接字

socket 最初是为同一主机上的应用程序创建，是的主机上运行的一个程序与另一个程序之间可以完成数据通信操作，程序也称为进程，就出现了不同进程间进行数据交互的操作，因为交互的模式不同，所以也一般将程序中的套接字区分为两种

- 面向文件的套接字：通过文件进行数据交互
- 面向网络的套接字：通过网络进行数据交互

套接字最初出现之后，为了区分不同平台不同体系的套接字模型对象，定义了地址家族的概念:Adress Family，也经常简写为 AF。
Unix 系统中的套接字就是第一种家族套接字，称为 AF_UNIX，或者 AF_LOCAL，这是 POSIX1 标准中规范的。
同样也有一些体系平台下，对于套接字的划分通过协议家族(Protocal Family)进行规范，可以简写为 PF，考虑到兼容性的要求，PF 和 AF 在一些平台上都集成了互相兼容的实

现，我们的 python 本身支持的是更加标准化的 AF 地址家族

另一种地址家族就是面向网络的套接字模型对象了，经典的如 AF_INET 支持的是传统第四个版本的因特网协议寻址[IPv4]，同时也提供了 AF_INET6 用于支持第六个版本的因特网协议寻址[IPv6]

2.1.2. PYTHON 中的套接字

套接字模型对象，为了能明确的表示网络中一台数据交互的主机，需要通过 IP 地址寻址确定主机位置，通过 PORT 端口号确定主机交互接口

在网络套接字交互过程中，出现了两种类型的套接字模型
- 面向连接的套接字模型
- 面向无连接的套接字模型

面向连接的套接字模型，在进行网络数据传输过程中，首先要创建一个连接模型，通过指定的连接模型进行数据的交互，类似我们生活中拨打电话一样，首先保证通话连接的基础上才能完成通话内容的交互，比较经典的如 TCP 端对端传输协议就是面向连接的套接字对象

面向无连接的套接字模型，在进行网络数据传输过程中，不需要有效的网络连接模型，在数据传输过程中只负责发送/接受，不保证数据的完整性和实效性；类似我们生活中的广播电台、电视信号等等，操作效率要比面向连接的套接字模型更加高效；比较经典的 UDP 广播协议使用的就是面向无连接的套接字对象

python 中提供的网络连接套接字，主要包含在 socket 模块中
socket 模块提供了 socket()函数可以完成如上描述的各种网络套接字的构建、通信等操作

- 基本语法结构

```
socket.socket(socket_family, socket_type, protocal=0)
    socket_family：socket 地址家族，AF_UNIX/AF_LOCAL 或者 AF_INET
    socket_type：socket 连接类型
        面向连接的(SOCK_STREAM)，面向无连接的(SOCK_DGRAME)
    protocal：传输协议，一般不用设置，使用默认值进行自动匹配就好
```

- 创建 TCP 协议的套接字 socket 对象

```
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- 创建 UDP 协议的套接字 socket 对象

```
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

2.1.3. socket 套接字常用操作

属性	描述
s.bind()	绑定(主机名称、端口)到一个套接字上
s.listen()	设置并启动 TCP 监听
s.accept()	等待客户端连接
s.connect()	连接指定服务器
s.connect_ex()	连接指定服务器，如果出现错误返回错误信息
s.recv()	接受 TCP 消息
s.recv_into()	接受 TCP 消息到缓冲区
s.send()	发送 TCP 消息
s.sendall()	完整发送 TCP 消息
s.recvfrom()	接受 UDP 消息
s.recvfrom_into()	接受 UDP 消息到缓冲区
s.sendto()	发送 UDP 消息
s.shutdown()	关闭连接对象
s.close()	关闭套接字对象

2.2.TCP 编程

TCP：Transmission Control Protocal 传输控制协议
是一种面向连接的，可靠的、基于字节流的传输层通信协议

数据传输的可靠性的处理方式，经典的 TCP 模型中通过如下两种方式完成连接的可靠性

- 三次握手建立连接
  - A- > B：发送一个寻址请求码 seq=100；B->A：返回一个应答 ack=101
  - A->B: 发送一个确认请求码 seq=101，确认连接；B->A: 返回一个应答 ack=300
  - A->B：发送一个连接请求码 ack=300；B->A：返回应答 ack=80
  - A 和 B 之间开始进行数据交互
- 四次挥手断开连接
  - A->B：发送一个数据验证请求码 seq=100，B->A：返回一个应答 ack=101
  - A->B：发送一个传输结束标记：seq=101；B->A：返回一个应答 ack=200
  - A->B：发送一个确认结束标记：seq=200；B->A：返回一个应答 ack=300
  - A->B：发送连接断开标记：seq=300；B->A：返回断开连接应答：ack=400

正是有了三次握手和四次挥手对于连接可靠性的保障，才让 TCP 协议端对端的数据交互变得可行，但是同样由于该协议的过于可靠，被有心人利用经常实施 DDOS 拒绝服务攻击!

2.2.1. 服务端开发

服务端，就是提供服务的一端，服务端的编程操作步骤如下:

- *定义需要监听的主机 IP 和端口号*
- *绑定 IP 地址和端口号到套接字对象*
- *开始监听*
- *等待连接*
- *连接成功-开始数据通信*
- *断开连接*

操作伪代码描述如下

```
# 定义常量数据
HOST = ‘’
PORT=8888
ADDRESS=(HOST, PORT)
# 创建 TCP 服务对象
tcp_server = socket.socket(socket.AF_INET, socket.SOCKET_STREAM)
# 绑定主机端口
tcp_server.bind(ADDRESS)
# 监听
tcp_server.listen()
# 等待连接
while True:
    sk = tcp_server.accept()
    # 收发消息
    while True:
        sk.send()/recv()
   # 关闭套接字，释放资源
   sk.close()
tcp_server.close()
```

实际项目代码开发:

```
'''
AUTHOR: DAMU 牟文斌
VERSION: V1.0.000
DESC: TODO
'''
import socket

# 定义服务器信息
```

```
print("初始化服务器信息...")
HOST = "192.168.10.140"
PORT = 8000
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 创建一个 TCP 服务端程序
print("创建 TCP 服务端对象")
tcp_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 绑定主机信息
print("绑定服务器主机")
tcp_server.bind(ADDRESS)

# 启动监听
print("启动监听")
tcp_server.listen(5)

# 轮询等待链接
while True:
    # 接收来自客户端的连接
    print("等待连接")
    client, addr = tcp_server.accept()
    print("客户端", addr, "已经接入服务器")

    # 接收消息
    while True:
        # 发送给客户端的数据
        msg = input("enter msg:")
        client.send(msg.encode("utf-8"))

        # 接收客户端发送的数据
        info = client.recv(BUFFER)
        # 打印展示数据
        print(info.decode("utf-8"))
        if info.decode("utf-8").lower() == "bye":
            print("客户端退出..bye")
            # 关闭和客户端保持连接的套接字
            client.close()
            break
```

2.2.2. 客户端开发

满足 TCP 协议的客户端，主要操作方式是主动发起请求连接服务器
完成和服务器的连接之后，进行和服务器之间的通信

开发操作步骤如下:

- *创建 TCP 连接套接字对象*
- *向指定的 IP 和 PORT 发起请求，请求连接*
- *连接成功，进行数据收发操作*
- *关闭连接，释放资源*

我们定义伪代码如下

```
client = socket()     # 创建 socket 对象

client.connect()      # 主动向指定的 IP 和 PORT 发起连接请求

while True:
    # 连接成功，进行消息的收发操作
    client.send()/recv()
# 关闭连接，释放资源
client.close()
```

项目代码开发

```
'''
AUTHOR: DAMU 牟文斌
VERSION: V1.0.000
DESC: TODO
'''
import socket

# 定义要连接的服务器信息
print("定位服务器信息...")
HOST = "192.168.10.140"
PORT = 8000
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 创建 TCP 客户端套接字对象
tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 连接指定服务器
tcp_client.connect(ADDRESS)
```

```
# 和服务器进行数据交互
while True:
    # 接收服务器的信息
    info = tcp_client.recv(BUFFER)
    print("server:", info.decode("utf-8"))

    # 给服务器发送消息
    msg = input("enter message:")
    tcp_client.send(msg.encode("utf-8"))

    if msg.lower() == 'bye':
        tcp_client.close()
        print("客户端退出...")
        break
```

2.2.3. 简易对讲机：QQ 雏形

需求分析：对讲机程序，主要实现使用软件的双方之间一对一通信功能

- 服务端程序开发

```
'''
AUTHOR: DAMU 牟文斌
VERSION: V1.0.000
DESC: TODO
'''
import socket

# 定义服务器信息
print("初始化服务器信息...")
HOST = "192.168.10.140"
PORT = 8000
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 创建一个 TCP 服务端程序
print("创建 TCP 服务端对象")
tcp_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 绑定主机信息
print("绑定服务器主机")
tcp_server.bind(ADDRESS)
```

```
# 启动监听
print("启动监听")
tcp_server.listen(5)

# 轮询等待链接
while True:
    # 接收来自客户端的连接
    print("等待连接")
    client, addr = tcp_server.accept()
    print("客户端", addr, "已经接入服务器")

    # 接收消息
    while True:
        # 发送给客户端的数据
        msg = input("enter msg:")
        client.send(msg.encode("utf-8"))

        # 接收客户端发送的数据
        info = client.recv(BUFFER)
        # 打印展示数据
        print("[", addr, "]client:", info.decode("utf-8"))
        if info.decode("utf-8").lower() == "bye":
            print("客户端退出..bye")
            # 关闭和客户端保持连接的套接字
            client.close()
            break
```

- 客户端程序开发

```
'''
AUTHOR: DAMU 牟文斌
VERSION: V1.0.000
DESC: TODO
'''
import socket

# 定义要连接的服务器信息
print("定位服务器信息...")
HOST = "192.168.10.140"
PORT = 8000
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 创建TCP 客户端套接字对象
tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# 连接指定服务器
tcp_client.connect(ADDRESS)

# 和服务器进行数据交互
while True:
    # 接收服务器的信息
    info = tcp_client.recv(BUFFER)
    print("server:", info.decode("utf-8"))

    # 给服务器发送消息
    msg = input("enter message:")
    tcp_client.send(msg.encode("utf-8"))

    if msg.lower() == 'bye':
        tcp_client.close()
        print("客户端退出...")
        break
```

2.3.UDP 编程

UDP: User Datagram Protocal 用户数据报协议
是 OSI/RM 模型中隶属于传输层的面向无连接的网络数据传输协议
UDP 协议本身没有连接可靠性的保证，没有数据顺序 ACK 记录，没有数据重发等机制，因为没有那么多的数据传输控制特性，所以 UDP 进行数据传输过程中延迟较小，数据传输效率较高，比较适合对可靠性要求不是很高的程序!

由于 UDP 面向无连接的特性，在编程处理的过程中，服务端和客户端的操作模式类似
唯一的区别就是服务端需要主动等待客户端发送消息，要将自己的 AF 暴露给客户端操作

2.3.1. 服务端开发

UDP 服务端开发步骤如下:

- *引入依赖的模块*
- *定义服务器描述信息*
- *创建 UDP 套接字对象并绑定主机*
- *消息循环: 和连接进入的客户端之间发送/收取消息*
- *关闭连接，释放资源*

UDP 服务端开发伪代码如下:

```
import socket                # 引入必须模块
```

```
HOST=''                          # 定义服务器信息
PORT=25556
ADDRESS = (HOST, PORT)
BUFFER=1024

udp_socket = socket.socket(…)  # 创建 UDP 套接字对象
udp_socket.bind(ADDRESS) # 绑定主机

while True:
    msg, addr= udp_socket.recvfrom(..)      # 接受消息，同时获取客户端
地址

    udp_socket.sendto(msg, addr)    # 向指定的客户端发送 msg 消息
```

服务端代码开发

```
# 引入需要的模块
import socket

# 定义服务器描述信息
HOST = ''
PORT = 8080
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 创建 UDP 套接字对象，并绑定当前主机
udp_server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_server.bind(ADDRESS)

# 消息循环
while True:
    # 接受消息
    print("waiting for message from client....")
    msg, addr = udp_server.recvfrom(BUFFER)
    # 打印展示消息
    print("client:", msg)
    # 发送消息
    udp_server.sendto("hello, my name is server".encode("utf-8"),
addr)
udp_server.close()
```

2.3.2. 客户端开发

客户端的开发更加简单，只需要明确目标主机信息
就可以完成和目标主机之间的消息发送/收取

```
udp_socket = socket.socket(..) # 创建 UDP 套接字对象

while True:       # 消息循环
    udp_socket.sendto(data, addr) / recvfrom()   发送接收消息

udp_socket.close()  # 关闭套接字
```

客户端编程实现

```
# 引入需要的模块
import socket

# 定义目标主机信息
HOST = '192.168.1.107'
PORT = 8080
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 创建 UDP 套接字对象
udp_client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# 消息循环
while True:
    msg = input("please enter the message for send:")

    udp_client.sendto(msg.encode("utf-8"), ADDRESS)

    if msg == "BYE":
        break

    info = udp_client.recvfrom(BUFFER)
    print("server: ", info)

udp_client.close()
```

2.3.3. 简易聊天室

案例开发

2.4.socketserver

2.4.1. socketserver 概述

socketserver 是 python 中提供的一个可以用于面向对象服务端开发的一个高级模块，封装了底层 socket 的大量操作实现，通过提供面向对象的操作方式简化程序开发

- socketserver 模块

属性/函数/类型	描述
BaseServer	基础服务类型，包含核心功能
TCPServer/UDPServer	TCP/UDP 服务类型
UnixStreamServer/UnixDatagramServer	面向文件的 TCP/UDP 服务类型
ForkingMixin/ThreadingMixin	核心派生线程类型
ForkingTCPServer/ForkingUDPServer	线程派生类型和 TCP/UDP 混合类型
ThreadingTCPServer/ThreadingUDPServer	同上
BaseRequestHandler	基础请求处理类型
StreamReqeustHandler/DatagramRequestHandler	面向连接/无连接的请求处理类型

2.4.2. TCP 编程

基于 socketserver 的 TCP 服务端的编程开发

```
# 导入依赖的模块
import socketserver

# 定义服务主机信息
HOST = '192.168.1.107'
PORT = 8080
ADDRESS = (HOST, PORT)
BUFFER = 1024


# 自定义处理类
class MyRequestHandler(socketserver.StreamRequestHandler):
    # 定义处理方法
    def handle(self):
        # 通过 client_address 属性查看连接进来的服务器
        print("连接上的服务器: %s" % str(self.client_address))
```

```
        # 接收客户端发送的数据
        msg = self.request.recv(BUFFER)
        print("客户端发过来消息: %s" % msg.decode("UTF-8"))
        # 给客户端返回接收信息
        self.request.sendall("已经成功接收您发送的消息".encode("UTF-
8"))


# 程序从主线程直接运行
if __name__ == "__main__":
    # 创建服务端对象，指定处理类，并监听 8888 端口
    server = socketserver.TCPServer(ADDRESS, MyRequestHandler)
    print("server is starting, waiting for connect.....")
    # 启动服务端程序
    server.serve_forever()
```

客户端的操作也相对比较简单，唯一需要注意的是：使用 TCPServer 构建的 TCP 服务端在和客户端进行数据通信的过程中，每一次的数据通信都会使用一个新的套接字对象，所以客户端程序需要改造~每次和服务器进行数据交互时都需要创建一个新套接字

```
# 导入依赖的模块
import socket

# 定义主机信息
HOST = '192.168.1.107'
PORT = 8080
ADDRESS = (HOST, PORT)


# 消息循环
while True:

    # 创建一个 TCP 套接字对象
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # 连接目标主机
    client.connect(ADDRESS)

    # 消息收发
    msg = input("enter message:")
    client.send(msg.encode("utf-8"))
    if msg == "BYE":
        break

    info = client.recv(1024)
    print(info)
```

```
    # 关闭套接字
    client.close()
```

2.4.3. UDP 编程

在 socketserver 中 UDP 编程操作方式和 TCP 编程操作方式及其类似

服务端的开发

```
# 引入依赖的模块
from socketserver import (UDPServer as UDP, DatagramRequestHandler as DRH)

# 服务器描述信息
HOST = '192.168.1.107'
PORT = 8888
ADDRESS = (HOST, PORT)

# 自定义消息处理类
class MyRequestHandler(DRH):

    # 消息处理函数
    def handle(self):

        print('客户端接入成功，开始收发消息：', self.client_address)

        # 接受消息
        msg = self.rfile.readline()
        print(msg.decode('utf-8'))

        # 发送消息
        self.wfile.write("my name is server，服务器".encode("utf-8"))


if __name__ == "__main__":
    # 创建 UDP 服务器
    udp_server = UDP(ADDRESS, MyRequestHandler)
    print("server is starting...")
    # 启动 UDP 服务器
    udp_server.serve_forever()
```

客户端开发

```
import socket

HOST = '192.168.1.107'
PORT = 8888
ADDRESS = (HOST, PORT)


while True:
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


    # 发送消息
    info = input("enter message:")
    client.sendto(info.encode("utf-8"), ADDRESS)

    # 接受消息
    msg, addr = client.recvfrom(1024)
    print(addr, " say : " , msg)

    client.close()
```

3. PYTHON 应用层网络编程

前面描述的基于 TCP/UDP 协议的网络程序开发，主要是针对传输层协议的底层代码实现
在实际操作过程中，更多的情况是直接操作应用层的数据协议的网络程序开发，如文件传输协议 FTP，邮件协议 SMTP 等等

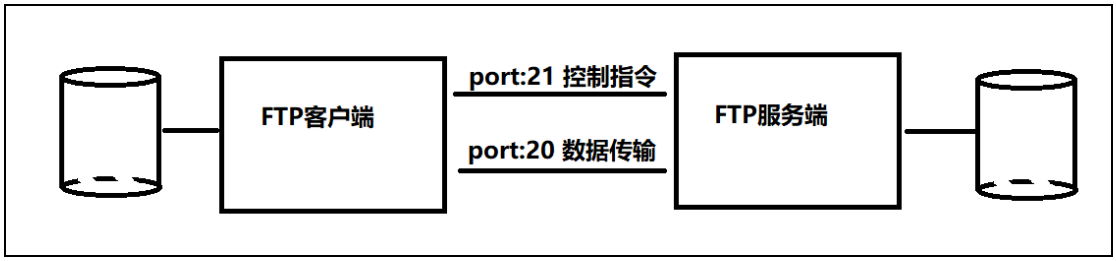
这一部分针对应用层的协议下网络程序的开发做一个深入的处理

3.1. FTP 文件服务

FTP：File Transfer Protocal 文件传输协议
工作在应用层的协议，由 Jon Postel 和 Joyce Reynolds 开发并记录在 RPC959 号文档中
FTP 协议主要用于匿名下载文件或者在两台计算机之间进行文件的传输工作!

FTP 协议底层采用的是 **TCP 协议**完成的网络数据传输，为了方便两台计算机上的文件正确的进行交互，FTP 协议封装了两个套接字完成文件操作，第一个工作在 21 端口的套接字专

门用于传输命令控制指令，第二个工作在 20 端口的套接字专门传输具体文件数据



3.1.1. FTP 客户端程序开发

FTP 网络程序的开发，我们不需要关注服务端的程序，服务端的软件程序开发和文件管理操作可以有大量的现成的工具去操作完成，而 FTP 如果作为应用软件的一部分功能，客户端程序的操作才是应用软件中最常规的操作

PYTHON 中提供了对 FTP 操作友好的支持，通过内建标准模块 ftplib 提供

针对 FTP 客户端的逻辑流程，进行如下步骤分析:

- 客户端——连接到服务器
- 客户端——账号+密码登录服务器
- 发出服务请求——控制指令、数据传输指令——处理响应数据
- 客户端退出

伪代码操作如下

```
from ftplib import FTP

ftp = FTP(“ftp.server.com”)

ftp.login(‘account’, ‘password’)

# 数据交互

ftp.quit()
```

3.1.2. FTP 类型常见属性方法

属性/方法	描述
login(user=’anonymous’, passwd=’’, acct=’’)	登录 FTP 服务器
pwd()	查看当前路径
cwd(path)	切换路径到指定的 path 路径
dir(path [,…[,cb]])	显示 path 路径中文件的内容

nlst([path [,….]])	类似 dir()，返回文件名称列表
retrlines(cmd [, cb])	给定 ftp 命令，下载文本文件 回调函数 cb 用于处理每一行文本
retrbinary(cmd, cb [, bs=8192 [, ra]])	给定 ftp 命令，下载二进制文件 回调函数 cb 处理每次读取的 8k 数据
storlines(cmd, f)	给定 ftp 命令，上传文本文件 f
storbinary(cmd, f [, bs=8192])	给定 ftp 命令，上传二进制文件 f
rename(old, name)	重命名 old 文件为 new
delete(path)	删除 path 指定的某个文件
mkd(directory)	创建一个目录 directory
rmd(directory)	删除指定的目录 directory
quit()	关闭连接，退出 FTP

3.1.3. 客户端 FTP 程序开发

搭建好我们的服务器环境，指定连接的 FTP 服务器信息、文件路径信息

```
# 引入需要的模块
import ftplib, socket

# 定义 FTP 主机信息
HOST = '192.168.1.108'
DIRN = '/home/damu/ftpwork/'
FILE = 'my_demo.tar.gz'


class FtpClient:
    '''自定义 FTP 客户端类型'''
    def __init__(self, host, username, password):
        '''
        初始化方法
        :param host:目标主机
        :param username:登录账号
        :param password:登录密码
        '''
        self.host = host

        self.username = username
        self.password = password

    def ftp_dowload(self, directory, file):
        '''下载文件主要方法'''
```

```
        self.directory = directory
        self.file = file

        self.__ftp_connect()
        self.__ftp_login()
        self.__ftp_cwd()
        self.__ftp_download()
        self.__ftp_quit()

    def __ftp_connect(self):
        '''连接服务器的方法'''
        try:
            self.ftp = ftplib.FTP(self.host)
        except (socket.error, socket.gaierror) as e:
            print("目标主机不可访问: ", self.host)
        else:
            print("主机连接成功", self.host)

    def __ftp_login(self):
        '''登录 FTP 服务器的方法'''
        try:
            self.ftp.login()
        except ftplib.error_perm:
            print("目标主机不能匿名登录，使用账号密码登录")
            try:
                self.ftp.login(user=self.username,
passwd=self.password)
            except:
                print("账号或者密码有误")
            else:
                print("登录成功")

    def __ftp_cwd(self):
        '''修改工作目录的方法'''
        try:
            self.ftp.cwd(self.directory)
        except ftplib.error_perm as e:
            print("路径修改失败，没有权限", self.directory)
        else:
            print("路径修改完成")

    def __ftp_download(self):
        '''下载文件核心方法'''
        try:
```

```
            self.ftp.retrbinary('RETR        {}'.format(self.file),
open(self.file, 'wb').write)
        except ftplib.error_perm as e:
            print("文件下载失败...")
        else:
            print("文件下载完成")

    def __ftp_quit(self):
        '''退出 FTP 客户端'''
        self.ftp.quit()
        print('程序退出...')


if __name__ == "__main__":
    # 创建 FTP 客户端对象
    ftp = FtpClient(HOST, 'damu', '123456')
    # 下载指定路径下的文件
    ftp.ftp_dowload(DIRN, FILE)
```

3.2.SMTP/POP3 邮件收发

3.3.网络新闻

4.简化版聊天系统

通过 PYTHON 网络编程结合 PYTHON 中的多线程操作，可以完成多用户在线无缝聊天的效果，通过在线登录，加入聊天工具实现用户端对端聊天

4.1.服务端程序开发

```
import threading, socket

# 存储用户的字典 key 用户昵称: value 用户 ip 地址
users = dict()
```

```
# 定义服务器地址
HOST = ''
PORT = 25666
ADDRESS = (HOST, PORT)
BUFFER = 1024


# 引擎主函数
def server_engine():
    # 创建服务端套接字
    server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    # 绑定服务器主机
    server_socket.bind(ADDRESS)
    # 监听主机连接
    server_socket.listen(10)
    # 开始等待客户端连接
    while True:
        # 等待连接
        s_socket, addr = server_socket.accept()
        # 等待客户端输入昵称
        s_socket.send("请输入昵称: ".encode("utf-8"))
        nickname = s_socket.recv(BUFFER).decode("utf-8")
        # 记录客户端并开始服务
        users[nickname] = (s_socket, addr)
        print(users)
        # 开始提供服务
        threading.Thread(target=server_transfer,
args=(s_socket,)).start()


# 转发消息线程
def server_transfer(s_socket):
    while True:
        # 接受客户端消息: 昵称:消息,进行转换
        msg = s_socket.recv(BUFFER)
        print("服务器接受到消息", msg.decode("utf-8"))
        # 拆分消息
        nickname, info = msg.decode("utf-8").split(":")
        # 发送消息 socket
        send_socket = users.get(nickname)[0]
        send_socket.send(msg)
```

```
# 启动服务
if __name__ == "__main__":
    server_engine()
```

4.2.客户端程序开发

```
import threading, socket

# 定义服务器地址
HOST = '192.168.1.101'
PORT = 25666
ADDRESS = (HOST, PORT)
BUFFER = 1024

# 客户端聊天主函数
def chat_client():
    # 创建套接字，连接服务器
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(ADDRESS)

    # 创建消息发送线程
    threading.Thread(target=msg_send, args=(client, )).start()
    # 创建消息接受线程
    threading.Thread(target=msg_recv, args=(client, )).start()


def msg_send(s_socket):
    while True:
        msg = input("发送消息(昵称:消息): ")
        s_socket.send(msg.encode("utf-8"))

def msg_recv(s_socket):
    while True:
        msg = s_socket.recv(BUFFER)
        print("》》》》 消息: ", msg.decode("utf-8"))


if __name__ == "__main__":
    chat_client()
```