# 概述

## 分类

TCP/IP协议组大体上分为三部分：

1、Internet协议(IP)

2、传输控制协议(TCP)和用户数据报文协议(UDP)

3、处于TCP和UDP之上的一组协议专门开发的应用程序。他们包括：TELNET,文件传输协议(FTP),域名服务(DNS)和简单的邮件传送程序(SMTP)等许多协议

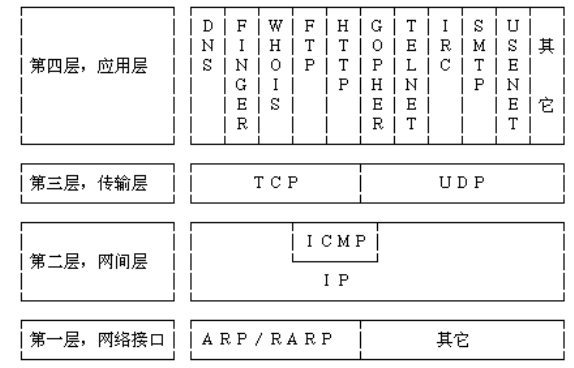


低三层为通信子网，负责数据传输；

高三层为资源子网，相当于计算机系统，完成数据处理；

传输层承上启下。

## TCP/IP四层模型



注：应用层属于应用程序，传输层、网际层、网络接口层属于内核。

## TCP/IP五层模型

互联网的实现，分成好几层。每一层都有自己的功能，就像建筑物一样，每一层都靠下一层支持。

用户接触到的，只是最上面的一层，根本没有感觉到下面的层。要理解互联网，必须从最下层开始，自下而上理解每一层的功能。

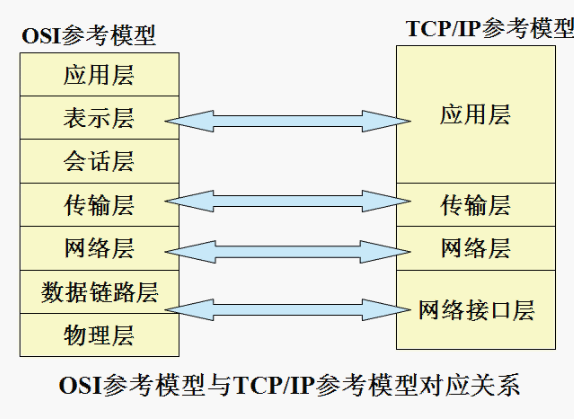
如何分层有不同的模型，有的模型分七层，有的分四层。我觉得，把互联网分成五层，比较容易解释。

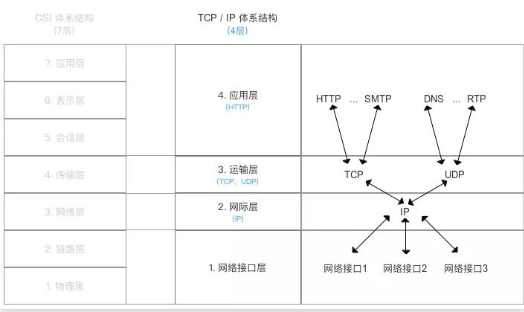


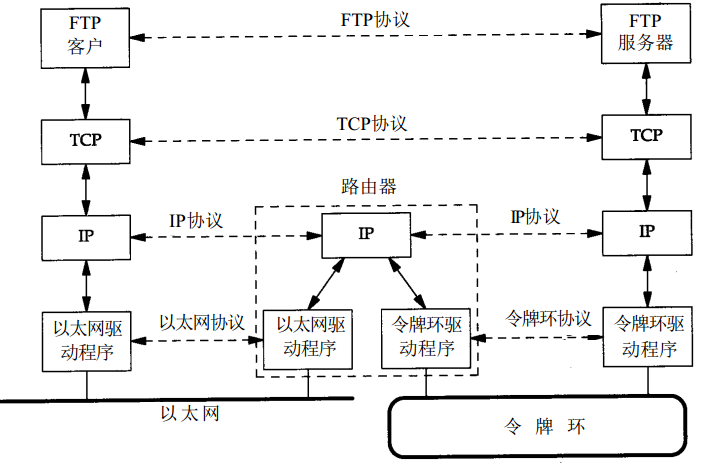
如上图所示，最底下的一层叫做"实体层"（Physical Layer），最上面的一层叫做"应用层"（Application Layer），中间的三层（自下而上）分别是"链接层"（Link Layer）、"网络层"（Network Layer）和"传输层"（Transport Layer）。越下面的层，越靠近硬件；越上面的层，越靠近用户。

注：应用层属于应用程序，传输层、网络层、链接层、实体层属于内核。

## 层与协议



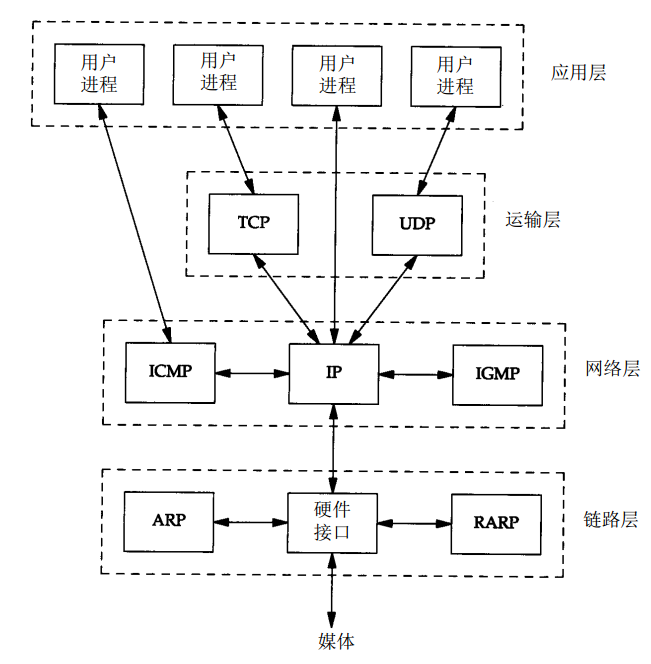




每一层都是为了完成一种功能。为了实现这些功能，就需要大家都遵守共同的规则。

大家都遵守的规则，就叫做"协议"（protocol）。

互联网的每一层，都定义了很多协议。这些协议的总称，就叫做"互联网协议"（Internet Protocol Suite），它们是互联网的核心。



应用层协议：HTTP协议、FTP协议

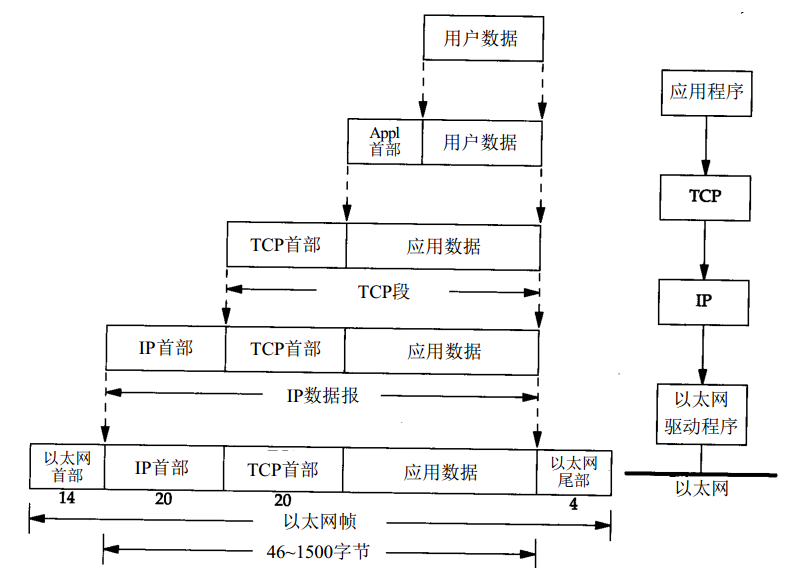
网络层协议：IP协议、ICMP协议、IGMP协议

传输层协议：TCP协议、UDP协议

网络接口层协议：ARP协议、RARP协议。

## 数据封装与分用

### 数据的封装形式



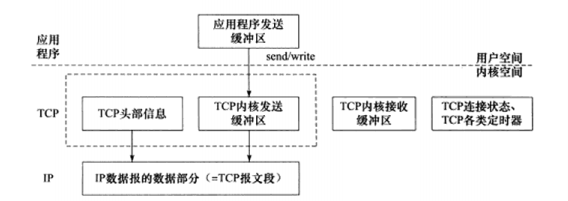
注：

不同协议层对数据包有不同的称谓，在传输层叫做**段（segment）**，在网络层叫做**数据报（datagram）**，在链路层叫做**帧（frame）**。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

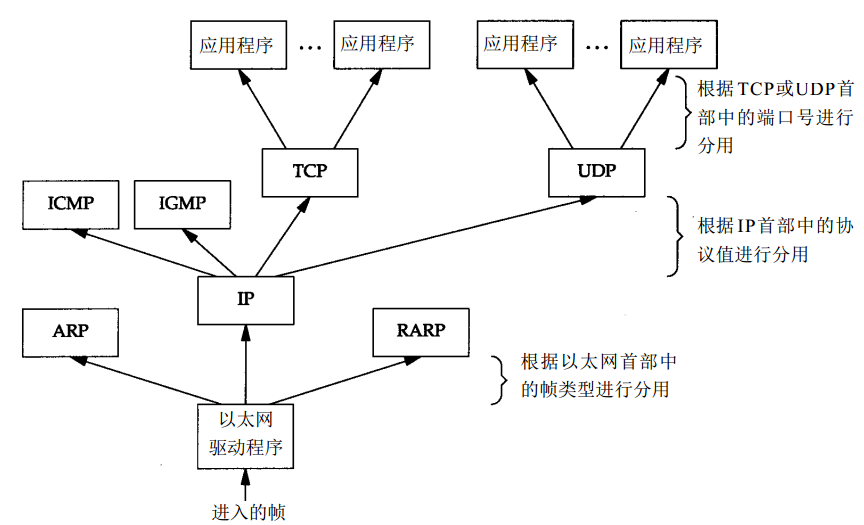
数据链路的传输媒介有多种，如果传输媒介不同，帧的类型也就不同。在以太网上传输的叫**以太网帧**，而令牌环网上的帧叫做**令牌环帧**。经过协议栈的层层封装，帧才是最终传输在物理媒介上的数据格式。

### TCP报文段传输过程

TCP报文段的封装过程：



### 数据分用的过程



# 实体层

从最底下的一层开始。电脑要组网，第一件事要干什么？当然是先把电脑连起来，可以用光缆、电缆、双绞线、无线电波等方式。

这就叫做"实体层"，它就是把电脑连接起来的物理手段。它主要规定了网络的一些电气特性，作用是负责传送0和1的电信号。

# 链接层

## 定义

单纯的0和1没有任何意义，必须规定解读方式：多少个电信号算一组？每个信号位有何意义？

这就是"链接层"的功能，它在"实体层"的上方，确定了0和1的分组方式。

## 以太网协议

早期的时候，每家公司都有自己的电信号分组方式。逐渐地，一种叫做"以太网"（Ethernet）的协议，占据了主导地位。

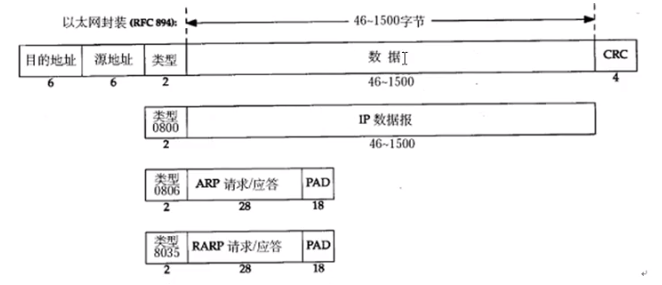
以太网规定，一组电信号构成一个数据包，叫做**"帧"（Frame）**。每一帧分成两个部分：**标头（Head）**和**数据（Data）**。

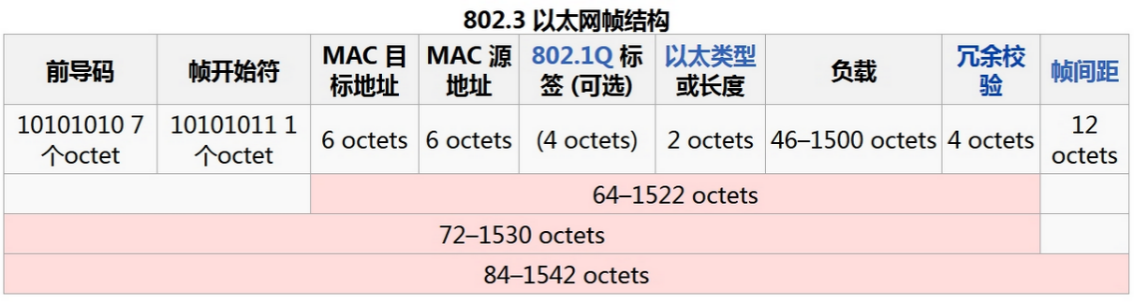


"标头"包含数据包的一些说明项，比如发送者、接受者、数据类型等等；"数据"则是数据包的具体内容。

"标头"的长度，固定为18字节。"数据"的长度，最短为46字节，最长为1500字节。因此，整个"帧"最短为64字节，最长为1518字节。如果数据很长，就必须分割成多个帧进行发送。

### 以太网帧格式





其中的源地址和目的地址指的是网卡的硬件地址（即MAC地址），长度48位，是在网卡出厂时固化的。可以使用shell中ifconfig命令查看。

### 最大传输单元(MTU)

在数据链路层传输的数据包称为帧，帧的最大传输单元（Max Transmit Unit）称为MTU。

### MAC地址

前面提到，以太网数据包的"标头"，包含了发送者和接受者的信息。那么，发送者和接受者是如何标识呢？

以太网规定，连入网络的所有设备，都必须具有"网卡"接口。数据包必须是从一块网卡，传送到另一块网卡。网卡的地址，就是数据包的发送地址和接收地址，这叫做MAC地址。

每块网卡出厂的时候，都有一个全世界独一无二的MAC地址，长度是48个二进制位，通常用12个十六进制数表示。

前6个十六进制数是厂商编号，后6个是该厂商的网卡流水号。有了MAC地址，就可以定位网卡和数据包的路径了。

注：以太网MAC地址的获取需要借助ARP协议！

## 广播

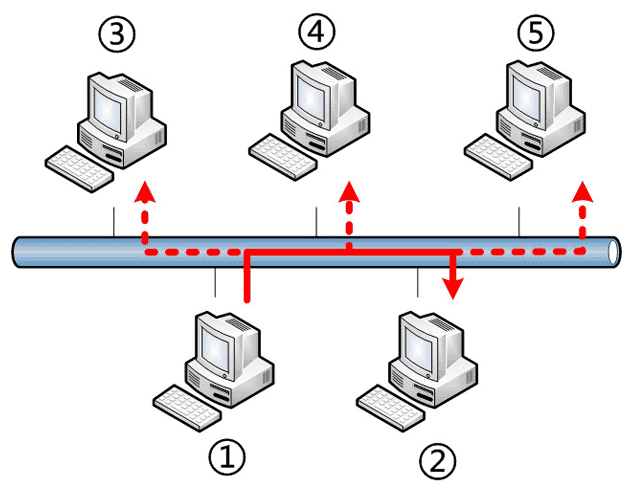
定义地址只是第一步，后面还有更多的步骤。

首先，一块网卡怎么会知道另一块网卡的MAC地址？

回答：有一种ARP协议，可以解决这个问题。以太网数据包必须知道接收方的MAC地址，然后才能发送。

其次，就算有了MAC地址，系统怎样才能把数据包准确送到接收方？

回答：以太网采用了一种很"原始"的方式，它不是把数据包准确送到接收方，而是向本网络内所有计算机发送，让每台计算机自己判断，是否为接收方。



上图中，1号计算机向2号计算机发送一个数据包，同一个子网络的3号、4号、5号计算机都会收到这个包。它们读取这个包的"标头"，找到接收方的MAC地址，然后与自身的MAC地址相比较，如果两者相同，就接受这个包，做进一步处理，否则就丢弃这个包。这种发送方式就叫做"广播"（broadcasting）。

有了数据包的定义、网卡的MAC地址、广播的发送方式，"链接层"就可以在多台计算机之间传送数据了。

## ARP协议

因为IP数据包是放在以太网数据包里发送的，所以我们必须同时知道两个地址，一个是对方的MAC地址，另一个是对方的IP地址。通常情况下，对方的IP地址是已知的（后文会解释），但是我们不知道它的MAC地址。

所以，我们需要一种机制，能够从IP地址得到MAC地址。

这里又可以分成两种情况：

第一种情况，如果两台主机不在同一个子网络，那么事实上没有办法得到对方的MAC地址，只能把数据包传送到两个子网络连接处的"网关"（gateway），让网关去处理。

第二种情况，如果两台主机在同一个子网络，那么我们可以用ARP协议，得到对方的MAC地址。ARP协议也是发出一个数据包（包含在以太网数据包中），其中包含它所要查询主机的IP地址，在对方的MAC地址这一栏，填的是FF:FF:FF:FF:FF:FF，表示这是一个"广播"地址。它所在子网络的每一台主机，都会收到这个数据包，从中取出IP地址，与自身的IP地址进行比较。如果两者相同，都做出回复，向对方报告自己的MAC地址，否则就丢弃这个包。

总之，有了ARP协议之后，我们就可以得到同一个子网络内的主机MAC地址，可以把数据包发送到任意一台主机之上了。

### 作用

ARP协议作用：能实现任意网络地址到任意物理地址的转换。

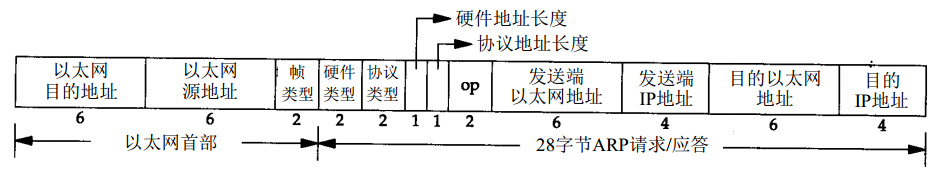
### 工作原理

ARP工作原理：广播一个ARP请求，被请求的机器会回应一个ARP应答，其中包含自己的物理地址。

注：网络上所有的机器都会接收到这个广播，但是只有被请求的机器会回应ARP应答。

### 报文格式

以太网ARP请求/应答报文：



### arp命令

arp命令用于操作主机arp缓冲区，可以显示arp缓冲区的所有条目、删除指定条目或增加静态IP地址与MAC地址的对应关系。

**命令格式：**

arp（选项）（参数）

**选项：**

-a<主机>：显示arp缓冲区的所有条目

-H<地址类型>：指定arp指令使用的地址类型

-d<主机>：从arp缓冲区中删除指定主机的arp条目

-D：使用指定接口的硬件地址

-e：以Linux的显示风格显示arp缓冲区中的条目

-i<接口>：指定要操作arp缓冲区的网络接口

-s<主机><硬件地址>：设置指定主机的IP地址与MAC地址的静态映射

-n：以数字方式显示arp缓冲区中的条目

-v：显示详细的arp缓冲区条目，包括缓冲区条目的统计信息

-f<文件>：设置主机的IP地址与MAC地址的静态映射

**参数：**

主机：查询arp缓冲区中指定主机的arp条目。

查看ARP缓存：arp –a

删除IP对应的ARP缓存：arp –d ip

添加IP对应的ARP缓存：arp –s ip mac

## RARP协议

RARP是反向地址转换协议，通过MAC地址确定IP地址。

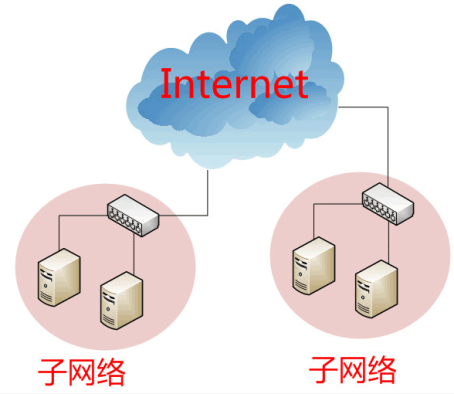
# 网络层

## 网络层的由来

以太网协议，依靠MAC地址发送数据。理论上，单单依靠MAC地址，上海的网卡就可以找到洛杉矶的网卡了，技术上是可以实现的。

但是，这样做有一个重大的缺点。以太网采用广播方式发送数据包，所有成员人手一"包"，不仅效率低，而且局限在发送者所在的子网络。也就是说，如果两台计算机不在同一个子网络，广播是传不过去的。这种设计是合理的，否则互联网上每一台计算机都会收到所有包，那会引起灾难。

互联网是无数子网络共同组成的一个巨型网络，很像想象上海和洛杉矶的电脑会在同一个子网络，这几乎是不可能的。



因此，必须找到一种方法，能够区分哪些MAC地址属于同一个子网络，哪些不是。如果是同一个子网络，就采用广播方式发送，否则就采用"路由"方式发送。（"路由"的意思，就是指如何向不同的子网络分发数据包）遗憾的是，MAC地址本身无法做到这一点。它只与厂商有关，与所处网络无关。

这就导致了"网络层"的诞生。它的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做"网络地址"，简称"网址"。

于是，"网络层"出现以后，每台计算机有了两种地址，一种是MAC地址，另一种是网络地址。两种地址之间没有任何联系，MAC地址是绑定在网卡上的，网络地址则是管理员分配的，它们只是随机组合在一起。

网络地址帮助我们确定计算机所在的子网络，MAC地址则将数据包送到该子网络中的目标网卡。因此，从逻辑上可以推断，必定是先处理网络地址，然后再处理MAC地址。

## TTL

**TTL：time to live**，设置数据包在路由节点中的跳转上限，每经过一个路由节点，该值-1，当减为0时，该路由有义务将该数据包丢弃。

注：如果TTL=20表示可以跳过20个路由器，超过即丢弃，这样可以防止网络拥塞。

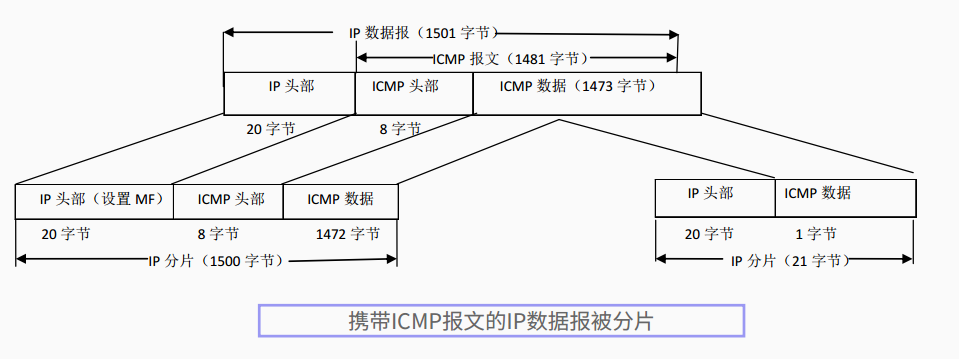
## 网络层作用

### IP分片

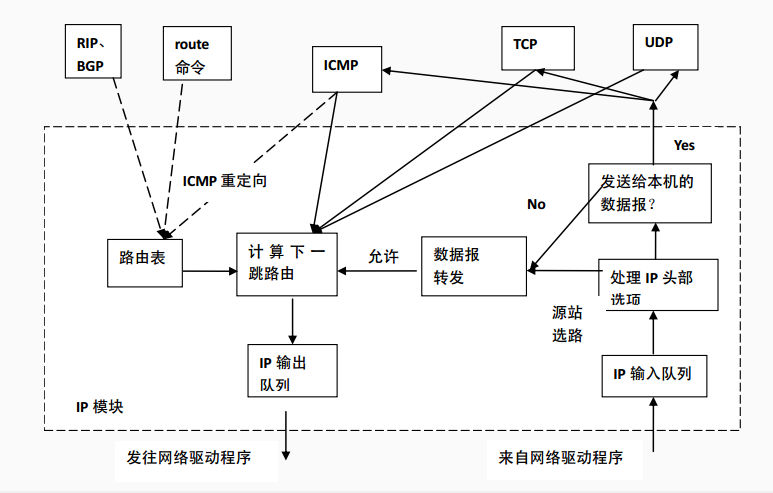
**IP数据报的长度超过帧的MTU时，将会被分片传输**。分片可能发生在发送端，传输过程中或者接收端，而且在发送过程中可能出现多次分片。

IP头部中的三个字段给IP分片和重组提供了足够的信息：数据报标识、标识和片偏移。

**以太网的MTU为1500字节，因此它携带的IP数据报最大为1480字节**。



### IP路由选择机制



IP路由选择的过程：

1. 查找完全匹配的主机IP地址
2. 查找相同网络ID的网络IP地址
3. 选择默认路由项

### IP转发

路由器转发模块的执行过程如下：

1. 检查数据报头部的TTL值
2. 查看数据包的严格源路由选择
3. 给源端发送一个ICMP重定向报文
4. 将TTL值减1
5. 处理IP头部选项
6. 如果有必要，执行IP分片操作

### 路由表更新

## IP协议

规定网络地址的协议，叫做IP协议。它所定义的地址，就被称为IP地址。

目前，广泛采用的是IP协议第四版，简称IPv4。这个版本规定，网络地址由32个二进制位组成。

习惯上，我们用分成四段的十进制数表示IP地址，从0.0.0.0一直到255.255.255.255。

互联网上的每一台计算机，都会分配到一个IP地址。这个地址分成两个部分，前一部分代表网络，后一部分代表主机。比如，IP地址172.16.254.1，这是一个32位的地址，假定它的网络部分是前24位（172.16.254），那么主机部分就是后8位（最后的那个1）。处于同一个子网络的电脑，它们IP地址的网络部分必定是相同的，也就是说172.16.254.2应该与172.16.254.1处在同一个子网络。

但是，问题在于单单从IP地址，我们无法判断网络部分。还是以172.16.254.1为例，它的网络部分，到底是前24位，还是前16位，甚至前28位，从IP地址上是看不出来的。

那么，怎样才能从IP地址，判断两台计算机是否属于同一个子网络呢？这就要用到另一个参数"子网掩码"（subnet mask）。

所谓"子网掩码"，就是表示子网络特征的一个参数。它在形式上等同于IP地址，也是一个32位二进制数字，它的网络部分全部为1，主机部分全部为0。比如，IP地址172.16.254.1，如果已知网络部分是前24位，主机部分是后8位，那么子网络掩码就是11111111.11111111.11111111.00000000，写成十进制就是255.255.255.0。

知道"子网掩码"，我们就能判断，任意两个IP地址是否处在同一个子网络。方法是将两个IP地址与子网掩码分别进行AND运算（两个数位都为1，运算结果为1，否则为0），然后比较结果是否相同，如果是的话，就表明它们在同一个子网络中，否则就不是。

比如，已知IP地址172.16.254.1和172.16.254.233的子网掩码都是255.255.255.0，请问它们是否在同一个子网络？两者与子网掩码分别进行AND运算，结果都是172.16.254.0，因此它们在同一个子网络。

### 作用

总结一下，IP协议的作用主要有两个，一个是为每一台计算机分配IP地址，另一个是确定哪些地址在同一个子网络。

IP主要有一下四个主要功能：

* 数据传送
* 寻址
* 路由选择
* 数据报文的分段

### IP服务特点

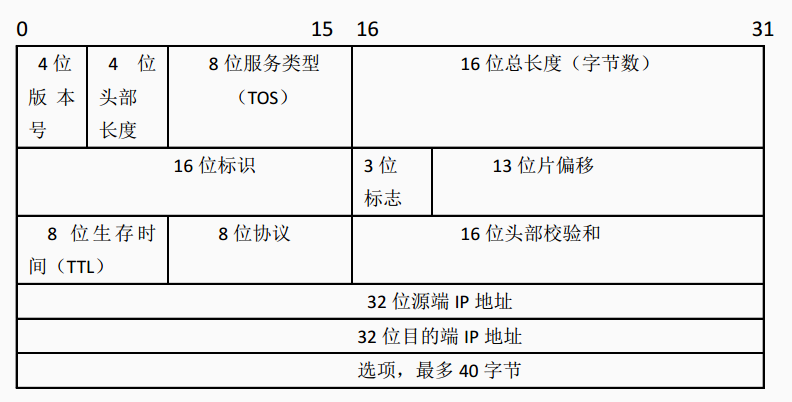
IP协议为上层协议提供**无状态、无连接、不可靠**的服务。

无状态：IP通讯的双方不传输状态，所有发送的数据没有上下文关系，简单高效，但是这决定了IP协议无法处理乱序、重复的报文，UDP、HTTP协议都是无状态的。

无连接：IP通信双方不长久维持连接。

不可靠：不保证数据包发送到接收方，只是尽力而为。不提供重传、数据确认。

### IPv4头部信息



版本：IPV4、IPV6

TTL（Time To Live生存时间）：数据包在路由之间的最大时长（每经过一个路由器-1，如果最后为0还没有发送成功则当前的路由器丢弃该数据包）

IPv4头部结构可变长的选项部分，最多包含40个字节，可选的IP选项包括：

记录路由

时间戳

松散源路由选择

严格源路由选择

### IP数据包

根据IP协议发送的数据，就叫做IP数据包。不难想象，其中必定包括IP地址信息。

但是前面说过，以太网数据包只包含MAC地址，并没有IP地址的栏位。那么是否需要修改数据定义，再添加一个栏位呢？

回答：不需要，我们可以把IP数据包直接放进以太网数据包的"数据"部分，因此完全不用修改以太网的规格。这就是互联网分层结构的好处：上层的变动完全不涉及下层的结构。

具体来说，IP数据包也分为"标头"和"数据"两个部分。

"标头"部分主要包括版本、长度、IP地址等信息，"数据"部分则是IP数据包的具体内容。它放进以太网数据包后，以太网数据包就变成了下面这样。



IP数据包的"标头"部分的长度为20到60字节，整个数据包的总长度最大为65,535字节。因此，理论上，一个IP数据包的"数据"部分，最长为65,515字节。前面说过，以太网数据包的"数据"部分，最长只有1500字节。因此，如果IP数据包超过了1500字节，它就需要分割成几个以太网数据包，分开发送了。

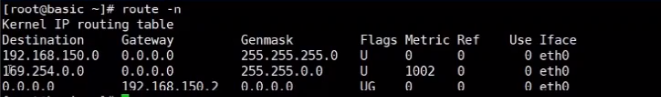
### 应用

#### route

route命令用来显示并设置Linux内核中的网络路由表，route命令设置的路由主要是静态路由。

要注意的是，直接在命令行下执行route命令来添加路由，不会永久保存，当网卡重启或者机器重启之后，该路由就失效了；可以在\*/etc/rc.local中添加route命令来保证该路由设置永久有效。

网络层查看路由表：route -n



注：第一行表示不需要网关，eth0这个网卡直接就可以访问192.168.150.0网段下的主机（这个就是我们网络配置文件中配置的）

第三行表示网关地址（也是配置文件中的）

**命令格式：**

route（选项）（参数）

**选项：**

-A：设置地址类型；

-C：打印将Linux核心的路由缓存；

-v：详细信息模式；

-n：不执行DNS反向查找，直接显示数字形式的IP地址；

-e： netstat格式显示路由表；

-net：到一个网络的路由表；

-host：到一个主机的路由表。

**参数：**

Add：增加指定的路由记录

Del：删除指定的路由记录

Target：目的网络或目的主机

gw：设置默认网关

mss：设置TCP的最大区块长度(MSS)，单位为MB

window：指定通过路由表的TCP连接的TCP窗口大小

dev：路由记录所表示的网络接口

#### ifconfig

ifconfig命令被用于配置和显示Linux内核中网络接口的网络参数。用ifconfig命令配置的网卡信息，在网卡重启后机器重启后，配置就不存在。要想将上述的配置信息永远的存的电脑里，那就要修改网卡的配置文件了。

**命令格式：**

ifconfig（参数）

**参数：**

add<地址>：设置网络设备IPv6的ip地址

del<地址>：删除网络设备IPv6的IP地址

down：关闭指定的网络设备

io\_addr< I/O地址>：设置网络设备的I/O地址

irq< IRQ地址>：设置网络设备的IRQ；

media<网络媒介类型>：设置网络设备的媒介类型；

mem\_start<内存地址>：设置网络设备在主内存所占用的起始地址；

metric<数目>：指定在计算数据包的转送次数时，所要加上的数目；

mtu<字节>：设置网络设备的MTU；

netmask<子网掩码>：设置网络设备的子网掩码；

tunnel<地址>：建立IPv4与IPv6之间的隧道通信地址；

up：启动指定的网络设备；

-broadcast<地址>：将要送往指定地址的数据包当成广播数据包来处理；

-pointopoint<地址>：与指定地址的网络设备建立直接连线，此模式具有保密功能；

-promisc：关闭或启动指定网络设备的promiscuous模式；

IP地址：指定网络设备的IP地址；

网络设备：指定网络设备的名称。

#### netstat

netstat命令用来打印Linux中网络系统的状态信息，可让你得知整个Linux系统的网络情况。

查看路由表或者端口信息：netstate

传输控制层查看路由表：netstate -ntp

查看监听端口：netstate -lnp

**选项：**

-a或–all：显示所有连线中的Socket；

-A<网络类型>或–<网络类型>：列出该网络类型连线中的相关地址；

-c或–continuous：持续列出网络状态；

-C或–cache：显示路由器配置的快取信息；

-e或–extend：显示网络其他相关信息；

-F或–fib：显示FIB；

-g或–groups：显示多重广播功能群组组员名单；

-h或–help：在线帮助；

-i或–interfaces：显示网络界面信息表单；

-l或–listening：显示监控中的服务器的Socket；

-M或–masquerade：显示伪装的网络连线；

-n或–numeric：直接使用ip地址，而不通过域名服务器；

-N或–netlink或–symbolic：显示网络硬件外围设备的符号连接名称；

-o或–timers：显示计时器；

-p或–programs：显示正在使用Socket的程序识别码和程序名称；

-r或–route：显示Routing Table；

-s或–statistice：显示网络工作信息统计表；

-t或–tcp：显示TCP传输协议的连线状况；

-u或–udp：显示UDP传输协议的连线状况；

-v或–verbose：显示指令执行过程；

-V或–version：显示版本信息；

-w或–raw：显示RAW传输协议的连线状况；

-x或–unix：此参数的效果和指定”-A unix”参数相同；

–ip或–inet：此参数的效果和指定”-A inet”参数相同

#### nc

**查看端口是否被占用：**

$ nc -l 6379

nc: Address already in use

当看到Address already in use的时候，就知道这个端口被占用了，否则它就会在这个端口监听。

**端口扫描：**

例如需要扫描某个机器上21-30哪些端口是开放的：

$ nc -n 127.0.0.1 -z 1230-1234 -v

nc: connect to 127.0.0.1 port 1230 (tcp) failed: Connection refused

nc: connect to 127.0.0.1 port 1231 (tcp) failed: Connection refused

nc: connect to 127.0.0.1 port 1232 (tcp) failed: Connection refused

nc: connect to 127.0.0.1 port 1233 (tcp) failed: Connection refused

Connection to 127.0.0.1 1234 port [tcp/\*] succeeded!

其中-n说明直接使用ip地址，而不使用域名，-z指定端口范围，-v输出详细信息。从结果就可以看到，1234端口是可连接的。

**TCP/UDP连接测试：**

当你在学习TCP相关的内容时，是否想着如何自己实践观察里面的状态或者数据包？虽然推荐自己去写一个TCP连接的服务端-客户端程序，但是也可以利用nc命令来完成，例如，在一个终端中输入如下内容：

$ nc -l 1234

hello 编程珠玑

表明在1234端口监听，然后可以在另外一个终端进行连接;

$ nc 127.0.0.1 1234

hello 编程珠玑

连接本地地址1234端口，这个过程中就可以抓包，分析TCP的三次握手了。

除此之外，你还可以在建立好的连接之间发送消息（简单的聊天功能），让你分析更多！

默认情况，它创建的是TCP连接，而使用-u(UDP)参数可以测试UDP连通性。

$ nc -v -u 182.3.226.35 80

Connection to 182.3.226.35 80 port [udp/http] succeeded!

除此之外，还有很多参数用于调试，例如：

-b 允许广播消息

-D 开启socket调试模式

-d 禁止从标准输入读取内容

-k 客户端退出后，保持连接

-v 显示详细信息

**HTTP连接测试：**

例如查看HTTP头信息：

$ nc www.baidu.com 80

HEAD / HTTP/1.1

HTTP/1.1 302 Found

Connection: Keep-Alive

Content-Length: 17931

Content-Type: text/html

Date: Sun, 23 Jun 2019 13:52:12 GMT

Etag: "54d9748e-460b"

Server: bfe/1.0.8.18

连接后，输入HEAD / HTTP/1.1或HEAD / HTTP/1.0，然后输入两个回车，可查看web服务器相关信息。

**数据传输：**

还是利用前面的连接，借助重定向符可以进行文件传输，例如服务端监听，并把内容输出到out.txt：

$ nc -l 1234 > out.txt

而客户端连接：

$ nc 127.0.0.1 1234 < in.txt

这样客户端in.txt的内容，就会传输到out.txt

当然了，对于文件传输，scp命令能更好地完成这件事。如果你没有其他办法了，可以试试nc。

**网络测试：**

前面说到可以通过nc传输数据，同样如果我们想测试两个主机间的网络速度（当然你可以利用iperf工具来完成这个工作），nc也是可以帮忙的：

#服务端监听：

$ nc -vl 1234 >/dev/null

其中重定向到/dev/null，表示将数据丢弃。

然后在另一台主机上执行:

$ dd if=/dev/zero bs=1M count=10 |nc -vn 127.0.0.1 1234

Connection to 127.0.0.1 1234 port [tcp/\*] succeeded!

10+0 records in

10+0 records out

10485760 bytes (10 MB, 10 MiB) copied, 0.0333675 s, 314 MB/s

dd拷贝数据，这里从/dev/zero拷贝数据0，且一次1M，拷贝10次，最后通过nc命令发送到服务端。结束后，就可以看到统计信息了。

**总结：**

nc命令短小精悍，但在很多方面能够帮助我们，例如：

端口扫描

连接测试

TCP/UDP服务端客户端监听与连接

网络测试

代理

## NAT映射

一般从我们主机出来的网络包不是直接发送到路由器的，而是发送到交换机，交换机进行IP地址的转发，转发给路由器（一般路由器都具备交换机的功能）。

路由器中存有路由表，NAT映射表。NAT映射表中记录的是连接到该路由器的终端的局域IP地址和公网IP地址的对应关系（客户端不可能记住公网IP，记住的是URL，通过URL转换为局域IP）。

NAT网络地址转换(Network Address Translation)属接入广域网(WAN)技术，是一种**将私有（保留）地址转化为合法IP地址的转换技术**，它被广泛应用于各种类型Internet接入方式和各种类型的网络中。原因很简单，NAT不仅完美地解决了lP地址不足的问题，而且还能够有效地避免来自网络外部的攻击，隐藏并保护网络内部的计算机。

## DHCP

DHCP是Dynamic Host Configuration Protocol(动态主机分配协议)缩写，它分为两个部份：一个是服务器端，而另一个是客户端。所有的IP网络设定数据都由DHCP服务器集中管理，并负责处理客户端的DHCP要求；而客户端则会使用从服务器分配下来的IP环境数据。

DHCP动态主机设置协议（Dynamic Host Configuration Protocol）是一个局域网的网络协议，使用UDP协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配IP地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。

## 打洞机制

路由器中的NAT映射表有一个自我保护机制，即对于第一次发送过来的陌生的IP会屏蔽或丢弃，主要是防止恶意攻击。

在进行网络通信的时候，有时候对于视频语音通信，可以在客户端和服务端的路由器之间再次开通一个便捷的通道，称为“打洞”，这样可以节省时间。中间的公网路由器帮助打洞，**实现私网IP到公网IP的通信，而不再经过中间的公网路由器**。

总结：

**公网-->公网：直接访问**

**公网-->私网：NAT映射**

**私网-->公网：NAT映射**

**私网-->私网：NAT映射+打洞机制**

## ICMP协议

### 背景

我们都知道，**IP协议是一个不可靠协议**，如果IP包在传输过程中出现错误，比如checksum对不上，拥塞，超时等等，那么IP包是会直接被丢弃的，之后也不会有进一步的努力来修正。

这是IP协议的一个设计准则决定的，也就是best effort，尽力而为，这样的好处是让IP协议尽量保持简单的形态，只负责有效率的数据传输，而更多的质量控制交给高层的协议去处理（比如TCP）。

但高层能提供质量控制的协议毕竟在少数，所以就需要在下层有协议来辅助IP完成必要的网络质量管理。ICMP协议自然就被提出来了。

通过ICMP协议，当IP包发生错误的时候，上层发送IP包的主机或路由器并不知道下层发生了错误，这个时候，下层的主机或路由器就可以通过发送ICMP包，将错误信息汇报给上层，从而让上层的主机或路由器进行调整。

不过需要注意的是，**ICMP仅仅只能提供某些特定类型的错误信息汇报，并不能帮助 IP 协议成为可靠的协议**。它能做的事还是有限，但用于基本的网络质量管理是足够了。

ICMP产生的原因：由于互联网之间通讯会涉及很多网关和主机，为了能够报告数据错误，所以产生了ICMP协议。也就是说**ICMP协议就是为了更高效的转发IP数据报和提高交付成功的机会**。

### 概述

ICMP协议是Internet控制报文协议（Internet Control Message Protocol互联网控制报文协议），它是TCP/IP协议族的一个子协议，用于**在IP主机、路由器之间传递控制消息**。

ICMP通常被认为是IP协议的一部分，它封装在IP层中，使用IP协议进行传输。因此，严格来说，ICMP既不是一个网络层协议，也不是一个传输层协议，而是介于两者之间的一个协议。

1、属于IP层协议

2、ICMP报文不是高层协议，而是作为IP层数据报的数据，加上数据报首部，组成IP数据报发出去

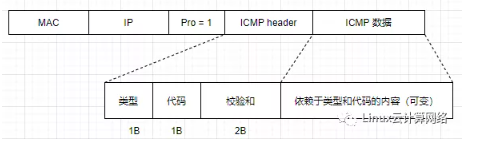
**作用：**

更有效地转发IP数据包，提高交付成功的机会

### ICMP报文

#### 格式

IP报头中的Protocol字段为1即表示该报文携带的是ICMP报文。



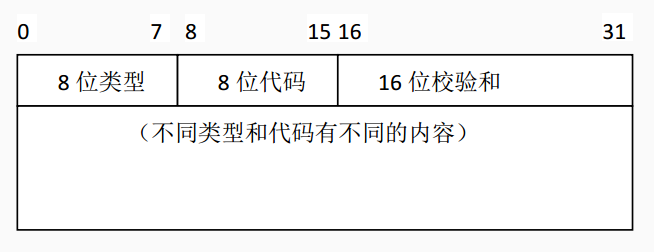
ICMP报头为4个字节：

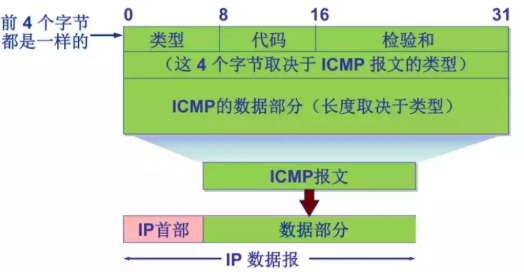
类型type：占1个字节（8位），表示较大范围类型分类的ICMP报文

代码code：占1个字节，表示较小范围类型分类的ICMP报文（type的细分）

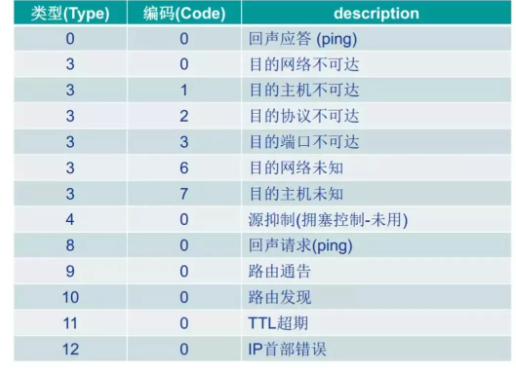
校验和checksum：占2个字节，ICMP checksum的计算方法类似于IP checksum，但是不同的是IP只校验头部，ICMP校验头部+数据部分

后面紧接的ICMP数据部分，根据前面的类型和代码字段的不同，具有不同的内容。

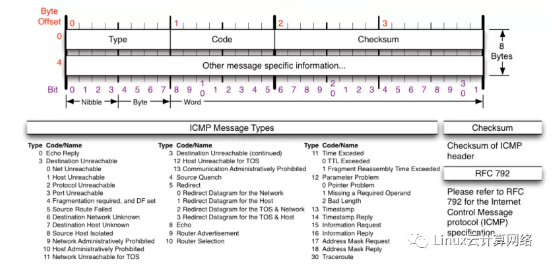




**类型和代码释义列表：**



#### 类型



类型字段指代了一大类，代码字段又细分了几大小类。

第一张表：类型表



注：R表示查询报文，E表示差错报文

进一步，对于每种类型，又可以根据代码字段细分多种子类型，请看第二张表：

第二张表：类型细分表



**源端抑制**

属于差错信息。如果某个源主机向目的主机快速地发送数据包，但目的主机来不及处理，就会向源主机发出该类型的ICMP包，提醒源主机放慢发送速度。

**重定向**

属于差错信息。如果某个源主机向网络中发送一个IP包，路径中某个路由器收到这个IP包，对照其路由表，发现自己不应该接收该包（包需要原路返回，或者不是最佳路由路径），就会向源主机发送该类型的ICMP包，提醒源主机修改自己的路由表，下次路由到另外一个更好的路由器。

**需要分片但设置了不分片位**

属于差错信息。如果某个源主机在发送一个IP包之前，对该IP包中的首部字段DF位设为1，也就是“分片禁止位=1”，表示该包在传输的过程中不允许分片，但是中间某个路由器允许传输的最大路径MTU小于该包大小，需要分片才能传输，但是由于设置不分片位，路由器会将该包丢弃，并向源主机发送一个携带MTU信息的ICMP包，提醒源主机下次发包的大小不应超过该MTU的值。

这种类型的ICMP包通常用来发现传输路径上的MTU值。

**TTL超时**

属于差错信息。超时定义了数据包在网络中存活的最长时间，IPv4中的TTL字段和IPv6中的Hop Limit字段都表示了这层意思，它们是一个整数值，会随着经过的路由器而递减，当减为0时，就认为该IP包超时，然后当前减为0的路由器会向源主机发送ICMP包，通知它发生了超时错误。

### 分类

它的主要功能是传输网络诊断信息，信息主要包括两类：

一类是查询类报文：主要用于信息的查询和采集，比如采集传输路径上的每个路由器都是谁，本次传输的报文是否达到目的地等等。

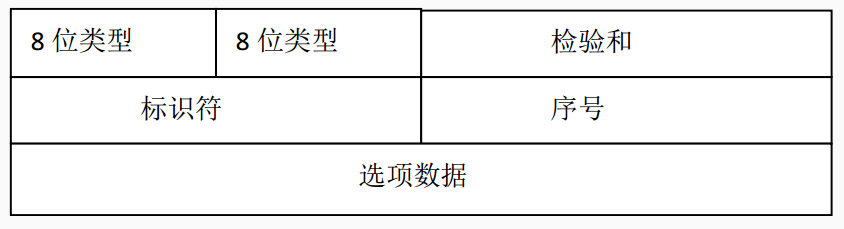
另一类是差错诊断类报文：主要用于诊断网络故障，比如传输报文被丢弃的原因是什么等等。

#### ICMP查询报文

ICMP查询报告报文共有4种：

1. 回送请求和回答
2. 时间戳请求和回答
3. 掩码地址请求和回答
4. 路由器询问和通过

回送请求和应答报文：



#### ICMP差错报文

ICMP差错报文共有5种：

终点不可达

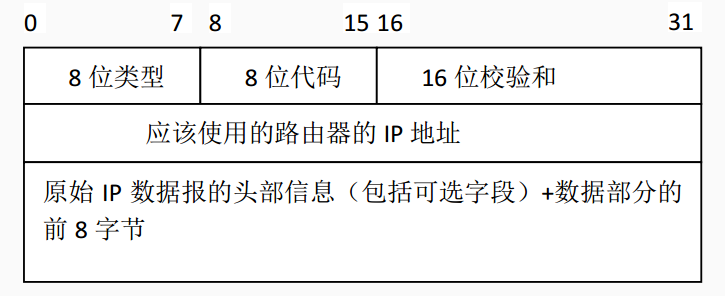
源站抑制

时间超过

参数问题

改变路由（重定向）

ICMP重定向报文格式：



### 问题

#### 泛洪攻击

ICMP协议是IP协议的助手，能够为IP协议提供相关的故障诊断和控制信息，但ICMP仍然不能为IP提供可靠性，最常见的丢包（路由器缓冲区溢出）并不会触发任何的ICMP信息，只能由其他协议如TCP来处理这种情况。

此外，正因为ICMP能够查询网络设备相关的配置信息，并且使用简单，黑客们都比较青睐使用ICMP报文来构建攻击报文。所以很多的网络设备都会用防火墙来阻止ICMP报文，这让很多诊断工具，比如上面介绍的几种，都很难发挥用武之地。

常见的ICMP攻击是ICMP泛洪攻击，这是一种DDoS攻击。简单说就是攻击者向一个子网的的广播地址发送多个ICMP echo包，包的源地址伪装成他想要攻击的目的主机的IP，然后该子网的所有主机的ICMP reply包都会送到被攻击主机，该主机瞬时收到大量的ICMP回复包，消耗大量资源，来不及处理，便会进入瘫痪或无法提供正常服务。

解决ICMP泛洪攻击最简单的方法就是禁ping。只要禁ping，不管黑客有多少肉机，他都无可奈何。

#### 禁ping

Linux默认是允许Ping响应的，系统是否允许Ping由2个因素决定的：

1、内核参数

2、防火墙

需要2个因素同时允许才能允许Ping，2个因素有任意一个禁Ping就无法Ping。

##### 内核参数设置

禁止ping设置：

1、临时禁止ping命令如下所示：

# 如果想要临时允许的话只需要把下面的1换成0即可

echo 1 >/proc/sys/net/ipv4/icmp\_echo\_ignore\_all

2、永久禁止ping命令如下所示(如果想要永久允许的话只需要把下面的1换成0即可)

在/etc/sysctl.conf 文件中增加一行

net.ipv4.icmp\_echo\_ignore\_all=1

修改完成后执行 sysctl -p 使新的配置生效。

##### 防火墙设置

(注意：此处的方法的前提是内核配置是默认值，也就是没有禁止ping)

这里以iptables防火墙为例：

1、允许ping设置

iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT

iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT

或者可以临时停止防火墙操作：

service iptables stop

2、禁止ping设置

iptables -A INPUT -p icmp --icmp-type 8 -s 0/0 -j DROP

### 应用

ICMP 的这些包的类型，用户可以充分用来诊断网络的故障情况。

因此诞生了一些利用ICMP协议的网络诊断工具，其中比较知名的就是ping和traceroute。这两工具分别利用两种类型的ICMP报文：

**ping使用查询类型报文**

**traceroute使用差错类型报文**

ping（分组网间探测）、traceroute（跟踪1个分组从源点到终点的路径，原理等同于从源主机向目的主机发送一连串的IP数据报）。

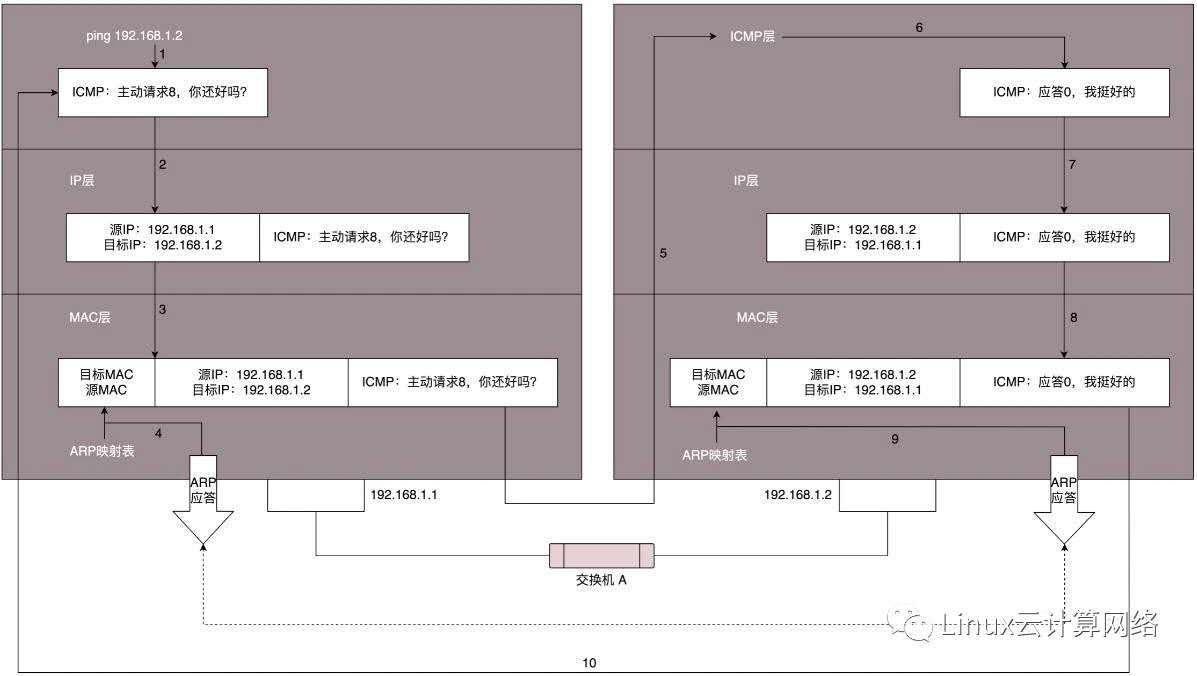
#### ping

一般我们用ping查看网络情况，主要是检查两个指标：

1. 第一个是看看是不是超时
2. 第二个看看是不是延迟太高

如果超时那么肯定是网络有问题（禁ping情况除外）；如果延迟太高，网络情况肯定也是很糟糕的。

##### 原理



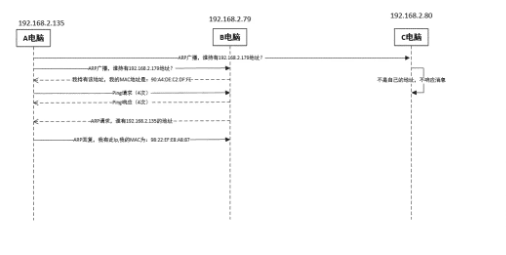
ping命令不是依托于TCP或者UDP这种传输层协议的，而是**依托于ICMP协议实现的**。

ping命令除了依托于ICMP，在局域网下还要借助于ARP协议，ARP协议能根据IP地址反查出计算机的MAC地址（即如果目标MAC地址未知的话，需要先发出ARP请求拿到，然后再进行封装）。另外ARP是有缓存的，为了保证 ARP的准确性，计算机会更新ARP缓存。

简单的说，ping就是一个测试程序，如果ping运行正确，大体上就可以排除网络访问层、网卡、Modem的输入输出线路、电缆和路由器等存在的故障，从而缩小问题的范围。

Ping过程解析，其流程如下：

1. A电脑（192.168.2.135）发起ping请求，ping192.168.2.179
2. A电脑广播发起ARP请求，查询192.168.2.179的MAC地址。
3. B电脑应答ARP请求，向A电脑发起单向应答，告诉A电脑自己的MAC地址为 90:A4:DE:C2:DF:FE
4. 知道了MAC地址后，开始进行真正的ping请求，由于B电脑可以根据A电脑发送的请求知道源MAC地址（缓存），所以就可以根据源MAC地址进行响应了。



Ping 4次请求和响应结束后，还有一次B电脑对 A电脑的ARP请求，这是为什么呢？这里应该是有2个原因：

1. 由于ARP有缓存机制，为了防止ARP过期，结束后重新更新下ARP缓存，保证下次请求能去往正确的路径，如果ARP过期就会导致出现一次错误，从而影响测试准确性。
2. 由于ping命令的响应时间是根据请求包和响应包的时间戳计算出来的，所以一次ARP过程也是会消耗时间。这里提前缓存最新的ARP结果就是节省了下次ping的ARP时间。

##### 选项

-d：使用Socket的SO\_DEBUG功能；

-c<完成次数>：设置完成要求回应的次数；

-f：极限检测；

-i<间隔秒数>：指定收发信息的间隔时间；

-L<网络界面>：使用指定的网络界面送出数据包；

-l<前置载入>：设置在送出要求信息之前，先行发出的数据包；

-n：只输出数值；

-p<范本样式>：设置填满数据包的范本样式；

-q：不显示指令执行过程，开头和结尾的相关信息除外；

-r：忽略普通的Routing Table，直接将数据包送到远端主机上；

-R：记录路由过程；

-s<数据包大小>：设置数据包的大小；

-v：详细显示指令的执行过程。

-t<存活数值>：设置存活数值TTL的大小

其中–c count次数，也就是ping的次数；-i interval间隔 ，每次ping之间的时间空格。

#### traceroute

traceroute是类Linux系统自带的工具，Windows上类似的工具是tracert，两者有些许不同，tracert默认使用ICMP报文探测，而traceroute默认使用UDP，但是也可以使用TCP/ICMP三种报文探测。

##### 原理

raceroute命令用于追踪数据包在网络上传输时的全部路径，它默认发送的数据包大小是40字节。

通过traceroute我们可以知道信息从你的计算机到互联网另一端的主机是走的什么路径。当然每次数据包由某一同样的出发点（source）到达某一同样的目的地(destination)走的路径可能会不一样，但基本上来说大部分时候所走的路由是相同的。

traceroute通过发送小的数据包到目的设备直到其返回，来测量其需要多长时间。一条路径上的每个设备traceroute要测3次。输出结果中包括每次测试的时间(ms)和设备的名称（如有的话）及其ip地址。

**traceroute利用ICMP差错报文**，主要用来确定这几件事：

1. 确定通信双方路径上经过的路由器设备

确定通信双方路径上经过的路由器设备。就是利用超时类型的ICMP报文来实现。traceroute向目的地发送IP包，刚开始的时候，将TTL设置为1，当经过第一个路由器时，TTL -1 = 0引发超时错误，第一个路由器回复ICMP超时报文，源主机就可以知道路径第一个路由器的信息，随后TTL被设置为2、3、4, ...，直到到达目的地，这样，沿途每个路由器都会向源主机回复ICMP超时报文，traceroute 就可以拿到所有的路由器信息了。

不过这里要注意 ，并不是所有路由器都会返回ICMP报文，因为出于安全性考虑，大多数防火墙以及启用了防火墙功能的路由器都默认配置为不返回任何ICMP报文，管理员也会主动配置，所以这时使用traceroute就不一定能拿到所有路由器信息了。

1. 确定UDP包是否成功达到目的地

确定UDP包是否成功达到目的地 。使用上面的方法能拿到路由器信息，但并不能确定发的包是否到达目的地。traceroute通过发送UDP包来解决了这个问题，因为UDP包的可用端口号范围<3000，所以就可以在发送UDP包的时候填入一个>3000的端口号，这样，如果当包确实到达了目的地，由于端口不匹配，就会返回一个端口不可达的ICMP报文，源主机就可以确定包确实到了目的地了。

1. 发现路径MTU

发现路径MTU，traceroute就是利用这种类型报文来逐一地确认传输路径上各个路由器之间的MTU值。

##### 选项

**命令格式：**

traceroute（选项）（参数）

**选项：**

-d：使用Socket层级的排错功能

-f<存活时间>：设置第一个检测数据包的存活数值TTL的大小

-F：设置勿离断位

-g<网关>：设置来源路由网关，最多可设置8个

-i<网络界面>：使用指定的网络界面送出数据包

-l：使用ICMP回应取代UDP资料信息

-m<存活数值>：设置检测数据包的最大存活数值TTL的大小

-n：直接使用IP地址而非主机名

-p<通信端口>：设置UDP传输协议的通信端口

-r：忽略普通的Routing Table，直接将数据包送到远端主机上

-s<来源地址>：设置本地主机送出数据包的TOS数值

-v：详细显示指令的执行过程

-w<超时秒数>：设置等待远端主机回报的时间

-x：开启或关闭数据包的正确性检验

**参数：**

主机：指定目的主机IP地址或主机名

#### MTR

MTR全称my traceroute，相对以上两个其实是更好的网络排障工具，只是用的人不多，导致它不太出名。之所以说它好，是因为它结合了ping、nslookup、traceroute三款工具的特性。

#### tcptraceroute

这块工具从名称就可以看出，是基于TCP的traceroute，也就是它使用TCP包（具体是TCP的SYN包）来进行网络探测，而不是ICMP包。

从上面我们已经知道，traceroute -T就是使用TCP包进行探测，所以tcptraceroute其实等效于traceroute -T。

使用TCP包进行探测的原因，主要是因为现代广泛使用的防火墙，出于安全的考虑，都会拦截UDP包和ICMP包，而通常不会拦截TCP SYN包。所以使用TCP包探测能够通过大多数的网络设备，使探测结果更加精确。

## IGMP协议

IGMP协议是Internet组管理协议（Internet Group Management Protocol），是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

## 路由器/交换机



# 传输层

## 传输层的由来

有了MAC地址和IP地址，我们已经可以在互联网上任意两台主机上建立通信。

接下来的问题是，同一台主机上有许多程序都需要用到网络，比如，你一边浏览网页，一边与朋友在线聊天。当一个数据包从互联网上发来的时候，你怎么知道，它是表示网页的内容，还是表示在线聊天的内容？

也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做"端口"（port），它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。

"端口"是0到65535之间的一个整数，正好16个二进制位。0到1023的端口被系统占用，用户只能选用大于1023的端口。不管是浏览网页还是在线聊天，应用程序会随机选用一个端口，然后与服务器的相应端口联系。

"传输层"的功能，就是建立"端口到端口"的通信。相比之下，"网络层"的功能是建立"主机到主机"的通信。只要确定主机和端口，我们就能实现程序之间的交流。因此，Unix系统就把主机+端口，叫做"套接字"（socket）。有了它，就可以进行网络应用程序开发了。

## UDP协议

### 数据包

现在，我们必须在数据包中加入端口信息，这就需要新的协议。最简单的实现叫做UDP协议，它的格式几乎就是在数据前面，加上端口号。

UDP数据包，也是由"标头"和"数据"两部分组成。





"标头"部分主要定义了发出端口和接收端口（端口用于标识某一进程，**大小16字节=65535**），"数据"部分就是具体的内容。然后，把整个UDP数据包放入IP数据包的"数据"部分，而前面说过，IP数据包又是放在以太网数据包之中的，所以整个以太网数据包现在变成了下面这样：



UDP数据包非常简单，"标头"部分一共只有8个字节，总长度不超过65,535字节，正好放进一个IP数据包。

注：**源端口和目的端口号最大是16字节**，即端口号范围0~65535。

### 特点

用户数据报文协议(UDP):

* 它允许在源和目的地站点之间传送数据，而不必再传送数据之前建立对话。
* 不使用TCP使用的端对端差错检验。
* 传输层功能全部发挥，而开销却比较低。
* 主要用于那些不要求TCP协议的非连接型的应用程序。例如：名字服务，网络管理，视频点播和网络会议。

### 应用

要求通信速度高

如：

域名转换：DNS协议

文件传输：FTP协议

网络管理：SNMP协议

远程文件服务器：NFS协议

## TCP协议

UDP协议的优点是比较简单，容易实现，但是缺点是可靠性较差，一旦数据包发出，无法知道对方是否收到。

为了解决这个问题，提高网络可靠性，TCP协议就诞生了。这个协议非常复杂，但可以近似认为，它就是有确认机制的UDP协议，每发出一个数据包都要求确认。如果有一个数据包遗失，就收不到确认，发出方就知道有必要重发这个数据包了。

因此，TCP协议能够确保数据不会遗失。它的缺点是过程复杂、实现困难、消耗较多的资源。

TCP数据包和UDP数据包一样，都是内嵌在IP数据包的"数据"部分。TCP数据包没有长度限制，理论上可以无限长，但是为了保证网络的效率，通常TCP数据包的长度不会超过IP数据包的长度，以确保单个TCP数据包不必再分割。

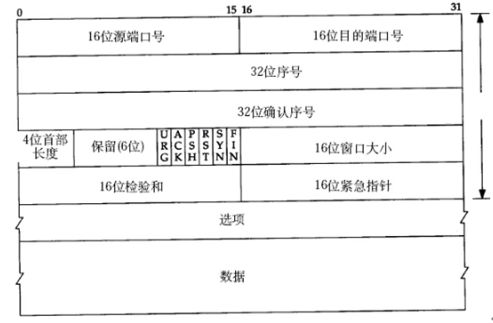
传输控制协议(TCP):

* 监听输入对话建立请求
* 请求另一网络站点对话
* 可靠的发送和接收数据
* 适度的关闭对话

### TCP服务特点

### TCP头部结构

TCP数据报格式：



注：

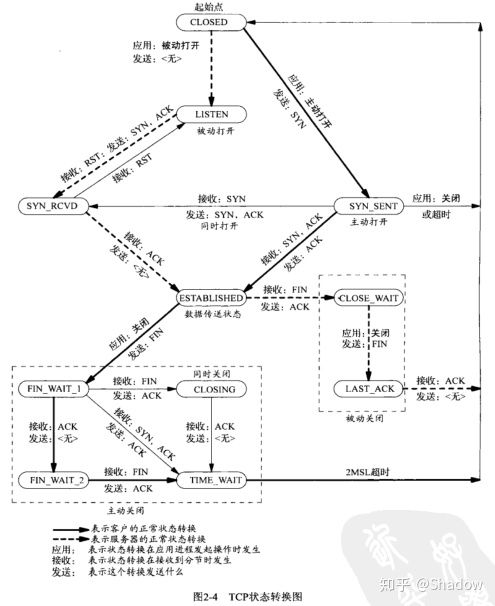
32位序号/32位确认序号：握手（以前发短信的回执消息即表示握手）

6个保留位：URG/ACK/PSH/SYN/FIN

16位窗口大小：65536

### TCP状态转移

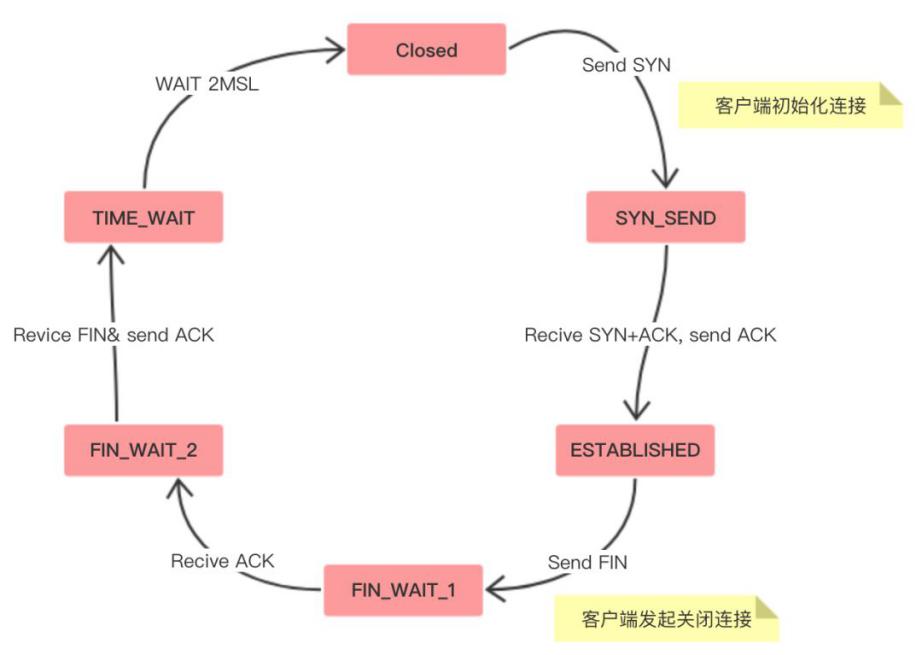
#### TCP状态转换图



注：状态图结合三次/四次握手，并且在服务器通信出现问题时，可以根据状态去排查问题。

实线对应主动关闭连接，虚线对应被动接收连接。

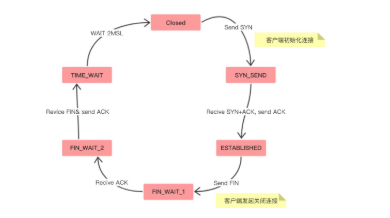
客户端状态变更图：

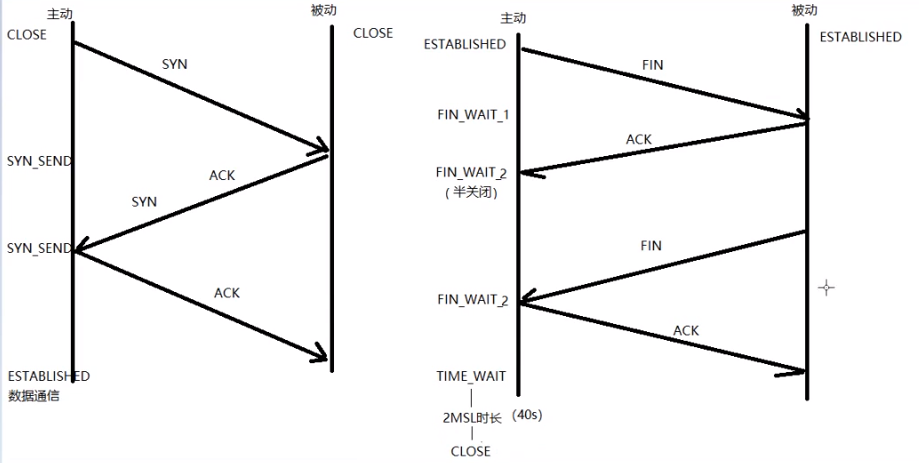


服务端状态变更图：

#### 主动关闭连接

客户端状态转移图：



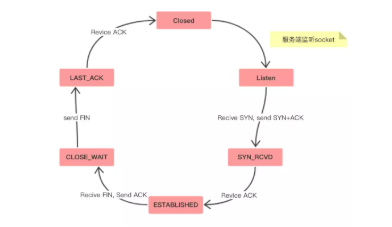


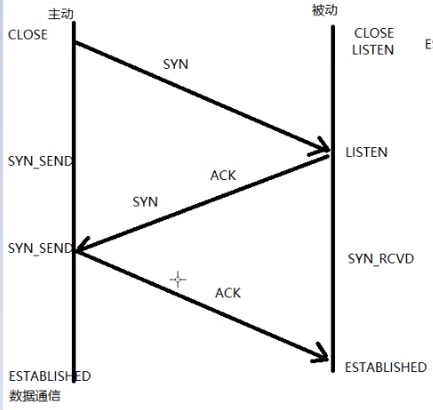
分析：

1. FIN\_WAIT\_1：主动发起关闭的一方在发送FIN消息后，状态变为FIN\_WAIT\_1
2. FIN\_WAIT\_2：接收到服务端的ACK信号后，客户端状态变化FIN\_WAIT\_2（即半关闭）
3. TIME\_WAIT：服务端发送FIN客户端接收后，状态仍为FIN\_WAIT\_2，只有发送给服务端ACK信号后，客户端状态才会变为TIME\_WAIT，此时并不代表成功关闭
4. CLOSED：客户端状态变为TIME\_WAIT后需要经历2MSL时长（大约40s），状态变为CLOSED

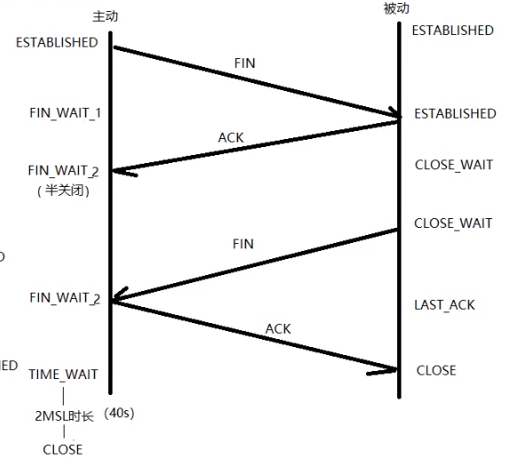
#### 被动接收连接

服务端状态转移图：





#### 被动关闭连接



#### 其他状态

#### 半关闭

当TCP连接中A发送FIN请求关闭，B端回应ACK后（A端进入FIN\_WAIT\_2状态），B没有立即发送FIN给A时，A放处于半连接状态，此时A可以接收B发送的数据，但是A不能再向B发送数据。

注：通常情况下我们都是直接close，而不设置半关闭状态。

从程序的角度，可以使用API控制实现半连接状态：

int shutdown(int sock, int howto); //Linux

int shutdown(SOCKET s, int howto); //Windows

参数说明：

sock为需要断开的套接字，howto为断开方式。

**howto在Linux下有以下取值：**

SHUT\_RD（0）：断开输入流/关闭socket上的读功能。套接字无法接收数据（即使输入缓冲区收到数据也被抹去），无法调用输入相关函数。

SHUT\_WR（1）：断开输出流/关闭socket写功能。套接字无法发送数据，但如果输出缓冲区中还有未传输的数据，则将传递到目标主机。

SHUT\_RDWR（2）：同时断开I/O流。相当于分两次调用shutdown()，其中一次以SHUT\_RD为参数，另一次以SHUT\_WR为参数。

**howto在Windows下有以下取值：**

SD\_RECEIVE：关闭接收操作，也就是断开输入流。

SD\_SEND：关闭发送操作，也就是断开输出流。

SD\_BOTH：同时关闭接收和发送操作。

使用close终止一个连接，但是它只是减少描述符的引用计数，并不是直接关闭连接，只有当描述符的引用计数为0时才关闭连接。

Shutdown不考虑描述符的引用计数，直接关闭描述符。也可选择终止一个方向的连接，只终止读或只终止写。

**注意：**

1. 如果有多个进程共享一个套接字，close每被调用一次，计数减1，直到计数为0时，也就是所有进程都调用了close，套接字将被释放；
2. 在多进程中如果有一个进程调用了shutdown(sfd,SHUT\_RDWR)后，其他的进程将无法进行通信。但是，如果一个进程close(sfd)将不会影响到其他进程。

**在调用dup2的时候，shutdown把所有的描述符都关闭，close只关闭一个**。

#### 2MSL

2MSL（Maximum Segment Lifetime）TIME\_WAIT状态存在的两个理由：

1. 让4次握手关闭流程更加可靠：4次握手的最后一个ACK是由主动关闭放发送出去的，若这个ACK丢失，被动关闭方会再次发送一个FIN过来，若主动关闭方能够保持一个2MSL的TIME\_WAIT状态，则有更大的机会让丢失的ACK被再次发送出去。
2. 防止lost duplicate对后续新建正常链接的传输造成破坏。Lost duplicate在实际的网络中非常常见，经常是由于路由器产生故障，路径无法收敛，导致一个packet在路由器A，B，C之间做类似死循环的跳转。IP头部有个TTL，限制了一个包在网络中的最大跳数，因此这个包有两种命运，要么最后TTL变为0，在网络中小时；要么TTL在变为0之前路由器路径收敛，它凭借剩余的TTL跳数终于到达目的地。但是非常可惜的是TCP通过超时重传机制在早些时候发送了一个跟它一模一样的包，并先于它达到目的地，因此它的命运也就注定被TCP协议栈抛弃。

2MSL存在的意义：最后一步客户端发送ACK给服务端的时候，发送完不能立刻更改客户端状态为CLOSE，因为还无法确定是否被服务端成功接收，如果服务端没有接收到ACK消息，则会继续向客户端发送FIN信号，设置这个等待的时长，如果在这个时长内没有发送FIN则认为发送成功，否则发送失败。即，2MSL的作用是保证最后一个ACK能成功被对端接收。

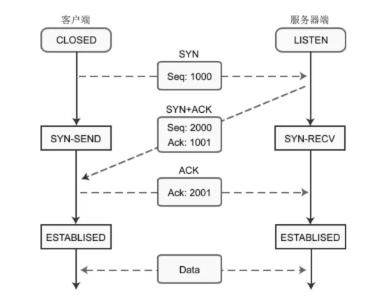
### TCP建立连接和关闭

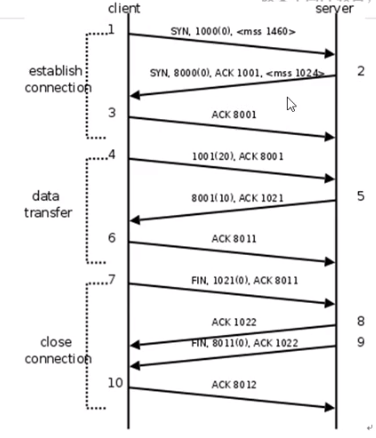
之所以会建立连接机制（三次握手、四次握手），是由于硬件网络设备的原因，导致IP网络层不稳定导致的，针对网络的抖动，传输层有两种应对方式：

1. 完全不弥补：UDP（无连接，不可靠的报文传输）
2. 完全弥补：TCP（面向连接的可靠的数据包传输）

#### TCP通信时序

##### 三次握手建立连接





**三次握手过程：**

1. SYN,1000(0) <mss1460>

客户端发送消息携带SYN标志位（开始握手的标志位）

1000表示数据包的编号（一般从0开始递增）

0表示包携带数据大小（0代表没有数据仅仅建立连接使用）

mss表示传递数据的上限

1. SYN,8000(0),ACK 1001,<mss 1024>

SYN,8000(0)服务端发往客户端的建立连接消息，消息号8000，数据包大小0

ACK服务端应答客户端请求，表示允许通信

1001表示客户端编号1001号报文之前的数据都已经收到

SYN表示服务器端跟客户端建立连接，编号与客户端的不一样，各自维护自己的编号

1. ACK 8001

客户端接收到服务器端8001之前的数据

至此，三次握手完成。

注：三次握手发生在内核，我们无需关心。在用户态的表现分别为：客户端connect，服务端accept，这两个函数成功返回表示三次握手成功。

**数据通信过程：**

1、1001(20),ACK 8001

客户端调用write发送数据，1001是客户端发送数据包编号

20客户端发送数据大小

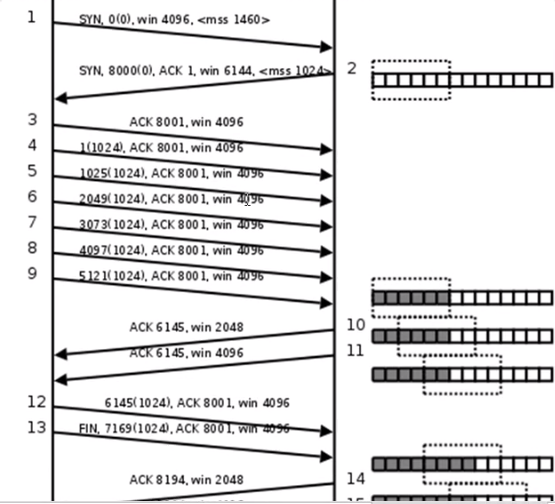
ACK8001

2、8001(10),ACK 1021

1. ACK 8011

8001+10=8011

当然，TCP实际发送数据并不是按照这样一个个发送和接收的，是批量执行的：



**拓展：为什么不是两次或者四次握手？**

1、为什么两次握手不可以呢？

为了防止已经失效的连接请求报文段突然又传送到了B（Server），因而产生错误。比如下面这种情况：A（Client）发出的第一个连接请求报文段并没有丢失，而是在网路结点长时间滞留了，以致于延误到连接释放以后的某个时间段才到达B。本来这是一个早已失效的报文段。但是B收到此失效的链接请求报文段后，就误认为A又发出一次新的连接请求。于是就向A发出确认报文段，同意建立连接。

对于上面这种情况，如果不进行第三次握手，B发出确认后就认为新的运输连接已经建立了，并一直等待A发来数据。B的许多资源就这样白白浪费了。

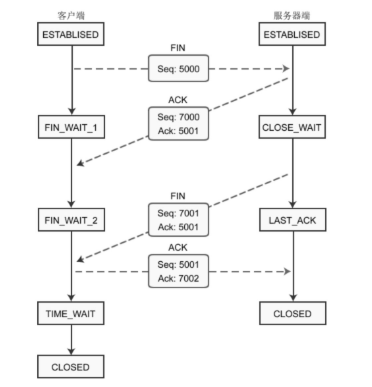
如果采用了三次握手，由于A实际上并没有发出建立连接请求，所以不会理睬B的确认，也不会向B发送数据。B由于收不到确认，就知道A并没有要求建立连接。

**2、为什么不需要四次握手？**

有人可能会说A发出第三次握手的信息后在没有接收到B的请求就已经进入了连接状态，那如果A的这个确认包丢失或者滞留了怎么办？

我们需要明白一点，完全可靠的通信协议是不存在的。在经过三次握手之后，客户端和服务端已经可以确认之前的通信状况，都收到了确认信息。所以即便再增加握手次数也不能保证后面的通信完全可靠，所以是没有必要的。

##### 四次握手关闭连接



**四次握手关闭连接：**

1. FIN,1021(0),ACK 8011

FIN客户端主动关闭连接请求

1021(0)客户端数据包的编号和大小（客户端主动发起的关闭，所以使用的是客户端的包号）

1. ACK 1022

服务端返回的信息

至此，**半关闭（客户端与服务端关闭）完成**，即客户端无法发数据，但是服务端还可以收数据，下面需要完成服务器端与客户端的关闭。

注：按照原来建立连接三次握手的思路，这里应该直接发送FIN,8011(0),ACK 1022即可，但是实际上并不是这样。

1. FIN,8011(0),ACK 1022

服务器端发送FIN给客户端，与客户端执行关闭操作

1. ACK 8012

注：**导致TCP四次握手关闭的原因在于半关闭的存在**。

如果服务器一直没有收到ACK，则会不断重试发送FIN信号。

**客户端和服务端的一个socket怎么做到客户端不能发数据，但是服务端还可以接收数据的？**

**半关闭不是关闭这个socket连接，关闭的是客户端/服务端的写缓冲区**，即客户端不会在去写数据（与服务端通信），但是这个连接还存在，还可以读取服务端的消息。

#### TIME\_WAIT状态

##### 原理

大量的TIME\_WAIT这个问题在日常中经常看到，流量一高就出现大量的该情况。该状态出现在主动发起关闭的一方。该状态一般等待的时间设为2MSL后自动关闭，MSL是Maximum Segment Lifetime，报文最大生存时间，如果报文超过这个时间，就会被丢弃。处于该状态下的socket也是不能被回收使用的。线上我就遇到这种情况，每次大流量的时候，每台机器处于该状态的socket就多达10w+，远远比处于Established 状态的socket多的多，导致很多时候服务响应能力下降。这个一方面可以通过调整内核参数处理，另一方面避免使用太多的短链接，可以采用连接池来提升性能。另外在代码层面可能是由于某些地方没有关闭连接导致的，也需要检查业务代码。

主动关闭的Socket端会进入TIME\_WAIT状态，并且持续2MSL时间长度，MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。MSL在RFC 1122上建议是2分钟，而源自berkeley的TCP实现传统上使用30秒，因而，TIME\_WAIT状态一般维持在1-4分钟。

注：2MSL一定出现在主动关闭连接请求端，保证最后一个ACK能成功被对端接收（等待期间，对端没收到ACK，对端会再次发送FIN请求）。

##### 作用

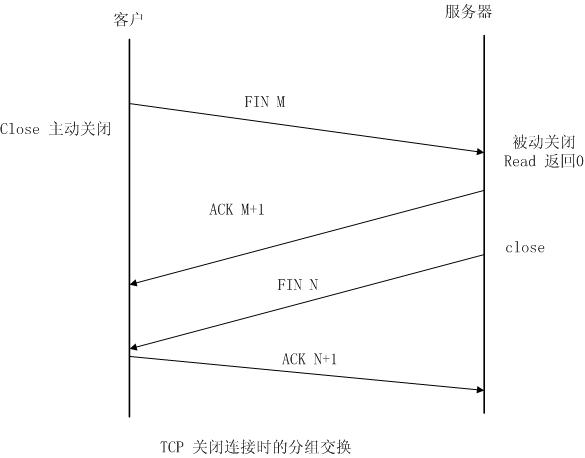
TIME\_WAIT状态存在的理由：

1）可靠地实现TCP全双工连接的终止

在进行关闭连接四路握手协议时，最后的ACK是由主动关闭端发出的，如果这个最终的ACK丢失，服务器将重发最终的FIN，因此客户端必须维护状态信息允许它重发最终的ACK。如果不维持这个状态信息，那么客户端将响应RST分节，服务器将此分节解释成一个错误（在java中会抛出connection reset的SocketException)。因而，要实现TCP全双工连接的正常终止，必须处理终止序列四个分节中任何一个分节的丢失情况，主动关闭 的客户端必须维持状态信息进入TIME\_WAIT状态。

2）允许老的重复分节在网络中消逝

TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个 原来的迷途分节就称为lost duplicate。在关闭一个TCP连接后，马上又重新建立起一个相同的IP地址和端口之间的TCP连接，后一个连接被称为前一个连接的化身（incarnation)，那么有可能出现这种情况，前一个连接的迷途重复分组在前一个连接终止后出现，从而被误解成从属于新的化身。为了避免这个情况，TCP不允许处于TIME\_WAIT状态的连接启动一个新的化身，因为TIME\_WAIT状态持续2MSL，就可以保证当成功建立一个TCP连接的时候，来自连接先前化身的重复分组已经在网络中消逝。



##### 故障

短时间后，所有的TIME\_WAIT全都消失，被回收，端口包括服务，均正常。即，在高并发的场景下，TIME\_WAIT连接存在，属于正常现象。

**线上场景中，持续的高并发场景：**

1、一部分TIME\_WAIT连接被回收，但新的TIME\_WAIT连接产生；

2、一些极端情况下，会出现大量的TIME\_WAIT连接。

Think：上述大量的TIME\_WAIT状态TCP连接，有什么业务上的影响吗？

Nginx作为反向代理时，大量的短链接，可能导致Nginx上的TCP连接处于time\_wait状态：

1、每一个time\_wait状态，都会占用一个本地端口，上限为65535(16 bit，2 Byte)；

2、当大量的连接处于time\_wait时，新建立TCP连接会出错，address already in use : connect 异常

统计 TCP 连接的状态：

// 统计：各种连接的数量

$ netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

ESTABLISHED 1154

TIME\_WAIT 1645

Tips：TCP 本地端口数量，上限为 65535（6.5w），这是因为 TCP 头部使用 16 bit，存储「端口号」，因此约束上限为 65535。

**问题分析：**

大量的 TIME\_WAIT 状态 TCP 连接存在，其本质原因是什么？

1、大量的短连接存在

2、特别是HTTP请求中，如果connection头部取值被设置为close时，基本都由服务端发起主动关闭连接

3、TCP 四次挥手关闭连接机制中，为了保证ACK重发和丢弃延迟数据，设置time\_wait为2倍的MSL（报文最大存活时间）

TIME\_WAIT 状态：

1、TCP连接中，主动关闭连接的一方出现的状态；（收到FIN命令，进入TIME\_WAIT状态，并返回ACK命令）

2、保持 2个MSL时间，即4分钟（MSL为2分钟）

**解决办法：**

解决上述time\_wait状态大量存在，导致新连接创建失败的问题，一般解决办法：

1、客户端，HTTP请求的头部，connection设置为 keep-alive，保持存活一段时间：现在的浏览器，一般都这么进行了

2、服务器端

允许time\_wait状态的socket被重用

缩减time\_wait时间，设置为1 MSL（即，2 mins）

##### 注意事项

在实际应用中，**对于高并发服务器，应该尽可能在服务器端避免出现TIME\_WAIT状态**：

1. 如果服务器端主动关闭连接（先于client调用close），服务端就会进入TIME\_WAIT（这样在一段时间内就会占用系统资源，使得服务器并发能力下降）；
2. 在协议设计上，应该让客户端主动断开连接，这样就把TIME\_WAIT状态分散到大量的客户端；
3. 如果客户端不活跃了，一些客户端不断开连接，这样就会占用服务器端的连接资源，因此服务器端也要有机制来踢掉不活跃的客户端连接（TIME\_WAIT存在的必要性）。

### RST复位报文段

#### RST报文段的作用

#### 发送RST报文段的情况

### TCP可靠传输机制

参考：

[https://mp.weixin.qq.com/s?\_\_biz=MzI5MzYzMDAwNw==&mid=2247489443&idx=1&sn=e535cd3845fe449667517089eeeadd79&chksm=ec6e6efbdb19e7ed56ca158323c9c2c08cebe95ad40eef1c917637e0586ac4bfa01800b9a46c&mpshare=1&srcid=&sharer\_sharetime=1592839542573&sharer\_shareid=38e4e5763736e36eb4485053caecfedf&from=timeline&scene=2&subscene=1&clicktime=1592840091&enterid=1592840091&ascene=2&devicetype=android-27&version=27000f3f&nettype=WIFI&abtest\_cookie=AAACAA%3D%3D&lang=zh\_CN&exportkey=AXwRBm5xMwYKKaBfnrnyUSU%3D&pass\_ticket=Aw1%2BPhFHNNJ0d8nWueBAITgV2OW0kcKac%2BJMDzRRlo5X5uPjaoq5gcgoJ5sRSz1B&wx\_header=1](https://mp.weixin.qq.com/s?__biz=MzI5MzYzMDAwNw==&mid=2247489443&idx=1&sn=e535cd3845fe449667517089eeeadd79&chksm=ec6e6efbdb19e7ed56ca158323c9c2c08cebe95ad40eef1c917637e0586ac4bfa01800b9a46c&mpshare=1&srcid=&sharer_sharetime=1592839542573&sharer_shareid=38e4e5763736e36eb4485053caecfedf&from=timeline&scene=2&subscene=1&clicktime=1592840091&enterid=1592840091&ascene=2&devicetype=android-27&version=27000f3f&nettype=WIFI&abtest_cookie=AAACAA==&lang=zh_CN&exportkey=AXwRBm5xMwYKKaBfnrnyUSU=&pass_ticket=Aw1+PhFHNNJ0d8nWueBAITgV2OW0kcKac+JMDzRRlo5X5uPjaoq5gcgoJ5sRSz1B&wx_header=1)

TCP的几个特点是：

顺序问题，依靠序号

丢包问题，依靠序号

流量控制，依靠滑动窗口

拥塞控制，依靠拥塞窗口+滑动窗口

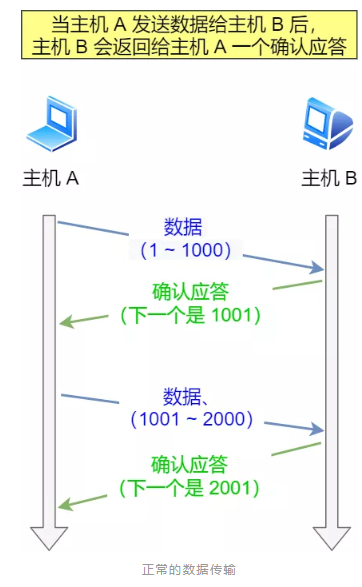
连接维护，三次握手/四次挥手



#### TCP重传机制

TCP实现可靠传输的方式之一，是通过**序列号**与确认应答。

在TCP中，当发送端的数据到达接收主机时，接收端主机会返回一个确认应答消息，表示已收到消息。



但在错综复杂的网络，并不一定能如上图那么顺利能正常的数据传输，万一数据在传输过程中丢失了呢？所以 TCP 针对数据包丢失的情况，会用重传机制解决。

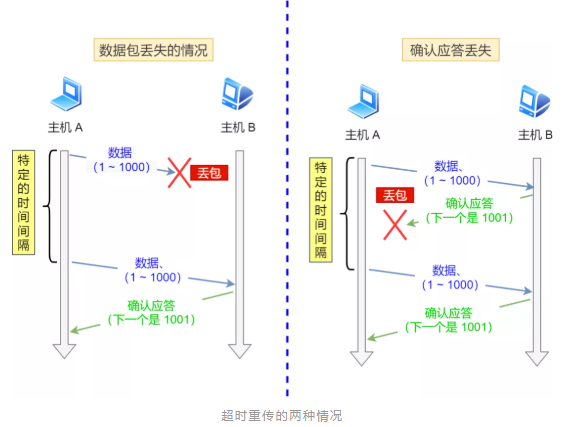
##### 超时重传

重传机制的其中一个方式，就是在发送数据时，设定一个定时器，当超过指定的时间后，没有收到对方的ACK确认应答报文，就会重发该数据，也就是我们常说的超时重传。

TCP使用超时重传来实现可靠传输：如果一个已经发送的报文段在超时时间内没有收到确认，那么就重传这个报文段。

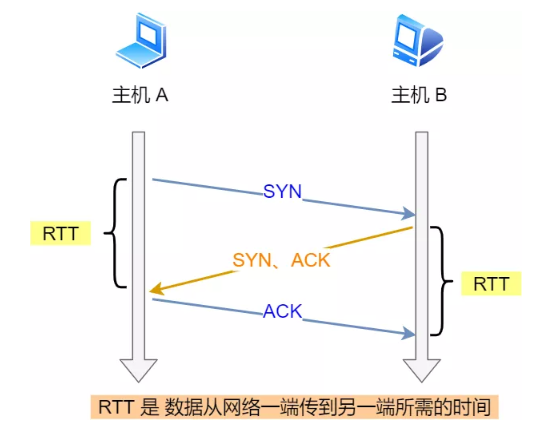
TCP会在以下两种情况发生超时重传：

1. 数据包丢失
2. 确认应答丢失



###### RTT

一个报文段从发送再到接收到确认所经过的时间称为往返时间RTT（Round-Trip Time 往返时延）。



加权平均往返时间 RTTs 计算如下：



其中，0 ≤a＜1，RTTs随着a的增加更容易受到RTT的影响。

超时时间RTO应该略大于 RTTs，TCP使用的超时时间计算如下：

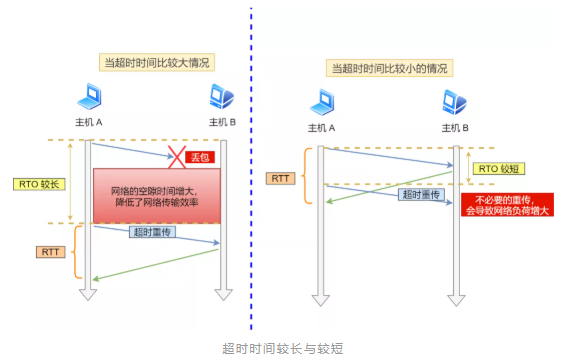


其中RTTd为偏差的加权平均值。

###### RTO

超时重传时间是以RTO（Retransmission Timeout 超时重传时间）表示。

假设在重传的情况下，超时时间RTO「较长或较短」时，会发生什么事情呢？



上图中有两种超时时间不同的情况：

当超时时间RTO较大时，重发就慢，丢了老半天才重发，没有效率，性能差；

当超时时间RTO较小时，会导致可能并没有丢就重发，于是重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

精确的测量超时时间 RTO 的值是非常重要的，这可让我们的重传机制更高效。

根据上述的两种情况，我们可以得知，超时重传时间RTO的值应该略大于报文往返 RTT的值。



至此，可能大家觉得超时重传时间 RTO 的值计算，也不是很复杂嘛。

好像就是在发送端发包时记下t0 ，然后接收端再把这个ack回来时再记一个t1，于是 RTT = t1–t0。没那么简单，这只是一个采样，不能代表普遍情况。

实际上「报文往返 RTT 的值」是经常变化的，因为我们的网络也是时常变化的。也就因为「报文往返 RTT 的值」 是经常波动变化的，所以「超时重传时间 RTO 的值」应该是一个动态变化的值。

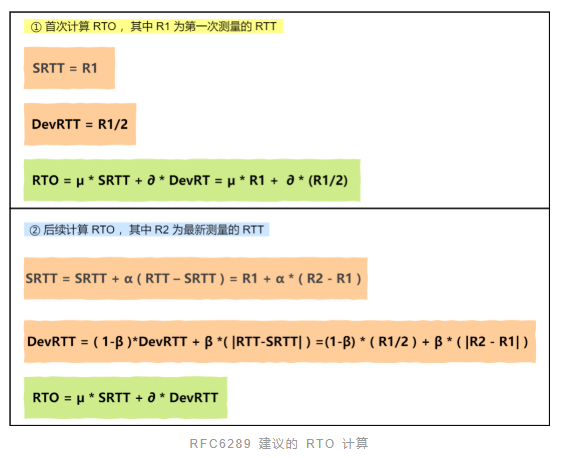
我们来看看 Linux 是如何计算 RTO 的呢？

估计往返时间，通常需要采样以下两个：

需要 TCP 通过采样 RTT 的时间，然后进行加权平均，算出一个平滑 RTT 的值，而且这个值还是要不断变化的，因为网络状况不断地变化。

除了采样 RTT，还要采样 RTT 的波动范围，这样就避免如果 RTT 有一个大的波动的话，很难被发现的情况。

RFC6289 建议使用以下的公式计算 RTO：



其中 SRTT 是计算平滑的RTT ，DevRTR 是计算平滑的RTT 与 最新 RTT 的差距。

在 Linux 下，α= 0.125，β= 0.25，μ= 1，∂= 4。别问怎么来的，问就是大量实验中调出来的。

如果超时重发的数据，再次超时的时候，又需要重传的时候，TCP 的策略是超时间隔加倍。

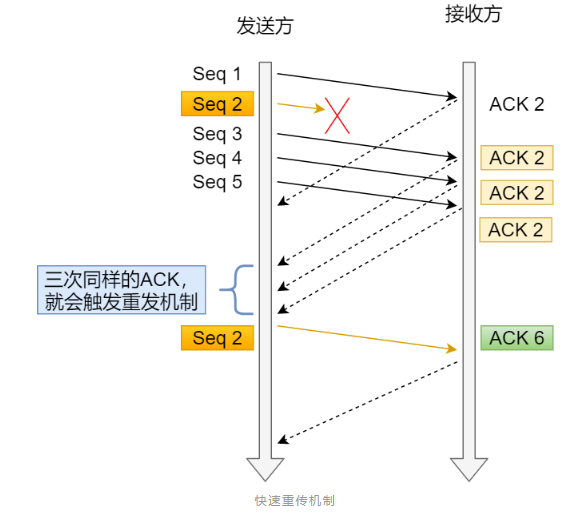
也就是每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。

超时触发重传存在的问题是，超时周期可能相对较长。那是不是可以有更快的方式呢？

于是就可以用「快速重传」机制来解决超时重发的时间等待。

##### 快速重传

TCP还有另外一种快速重传（Fast Retransmit）机制，它不以时间为驱动，而是以数据驱动重传。



在上图，发送方发出了1，2，3，4，5份数据：

1. 第一份Seq1先送到了，于是就Ack回2；
2. 结果Seq2因为某些原因没收到，Seq3到达了，于是还是Ack回2；
3. 后面的Seq4和Seq5都到了，但还是Ack回2，因为Seq2还是没有收到；
4. 发送端收到了三个Ack = 2的确认，知道了Seq2还没有收到，就会在定时器过期之前，重传丢失的Seq2。
5. 最后，接收到收到了Seq2，此时因为Seq3，Seq4，Seq5都收到了，于是Ack回6。

所以，快速重传的工作方式是当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。

快速重传机制只解决了一个问题，就是超时时间的问题，但是它依然面临着另外一个问题。就是重传的时候，是重传之前的一个，还是重传所有的问题。

比如对于上面的例子，是重传Seq2呢？还是重传Seq2、Seq3、Seq4、Seq5呢？因为发送端并不清楚这连续的三个Ack 2是谁传回来的。

根据TCP不同的实现，以上两种情况都是有可能的。可见，这是一把双刃剑。

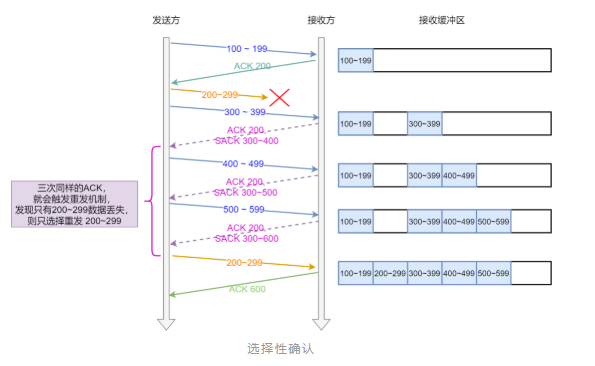
为了解决不知道该重传哪些TCP报文，于是就有SACK方法。

##### SACK

还有一种实现重传机制的方式叫：SACK（ Selective Acknowledgment选择性确认）。

这种方式需要在TCP头部「选项」字段里加一个SACK的东西，它可以将缓存的地图发送给发送方，这样发送方就可以知道哪些数据收到了，哪些数据没收到，知道了这些信息，就可以只重传丢失的数据。

如下图，发送方收到了三次同样的ACK确认报文，于是就会触发快速重发机制，通过SACK信息发现只有200~299这段数据丢失，则重发时，就只选择了这个TCP段进行重复。



如果要支持SACK，必须双方都要支持。在Linux下，可以通过net.ipv4.tcp\_sack参数打开这个功能（Linux 2.4 后默认打开）。

##### D-SACK

Duplicate SACK又称D-SACK，其主要使用了SACK来告诉「发送方」有哪些数据被重复接收了。

**ACK丢包**

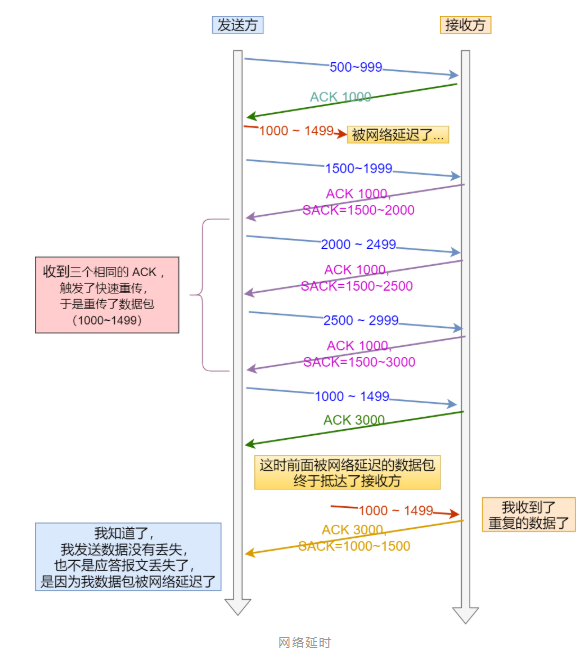


「接收方」发给「发送方」的两个 ACK 确认应答都丢失了，所以发送方超时后，重传第一个数据包（3000 ~ 3499）

于是「接收方」发现数据是重复收到的，于是回了一个 SACK = 3000~3500，告诉「发送方」 3000~3500 的数据早已被接收了，因为 ACK 都到了 4000 了，已经意味着 4000 之前的所有数据都已收到，所以这个 SACK 就代表着 D-SACK。

这样「发送方」就知道了，数据没有丢，是「接收方」的 ACK 确认报文丢了。

**网络延时**



数据包（1000~1499） 被网络延迟了，导致「发送方」没有收到 Ack 1500 的确认报文。

而后面报文到达的三个相同的 ACK 确认报文，就触发了快速重传机制，但是在重传后，被延迟的数据包（1000~1499）又到了「接收方」；

所以「接收方」回了一个 SACK=1000~1500，因为 ACK 已经到了 3000，所以这个 SACK 是 D-SACK，表示收到了重复的包。

这样发送方就知道快速重传触发的原因不是发出去的包丢了，也不是因为回应的 ACK 包丢了，而是因为网络延迟了。

可见，D-SACK 有这么几个好处：

1. 可以让「发送方」知道，是发出去的包丢了，还是接收方回应的 ACK 包丢了；
2. 可以知道是不是「发送方」的数据包被网络延迟了；
3. 可以知道网络中是不是把「发送方」的数据包给复制了。

在 Linux 下可以通过 net.ipv4.tcp\_dsack 参数开启/关闭这个功能（Linux 2.4 后默认打开）。

#### TCP滑动窗口

##### 概述

**窗口的实现实际上是操作系统开辟的一个缓存空间，发送方主机在等到确认应答返回之前，必须在缓冲区中保留已发送的数据**。如果按期收到确认应答，此时数据就可以从缓存区清除。

窗口是**缓存**的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，接收方通过TCP报文段中的窗口字段告诉发送方自己的窗口大小，发送方根据这个值和其它信息设置自己的窗口大小。

**滑动窗口主要是为了防止数据丢失**。

##### 慢启动

服务器发送数据包，当然越快越好，最好一次性全发出去。但是，发得太快，就有可能丢包。带宽小、路由器过热、缓存溢出等许多因素都会导致丢包。线路不好的话，发得越快，丢得越多。

最理想的状态是，在线路允许的情况下，达到最高速率。但是我们怎么知道，对方线路的理想速率是多少呢？答案就是慢慢试。

TCP协议为了做到效率与可靠性的统一，设计了一个慢启动（slow start）机制。开始的时候，发送得较慢，然后根据丢包的情况，调整速率：如果不丢包，就加快发送速度；如果丢包，就降低发送速度。

Linux内核里面设定了（常量TCP\_INIT\_CWND），刚开始通信的时候，发送方一次性发送10个数据包，即"发送窗口"的大小为10。然后停下来，等待接收方的确认，再继续发送。

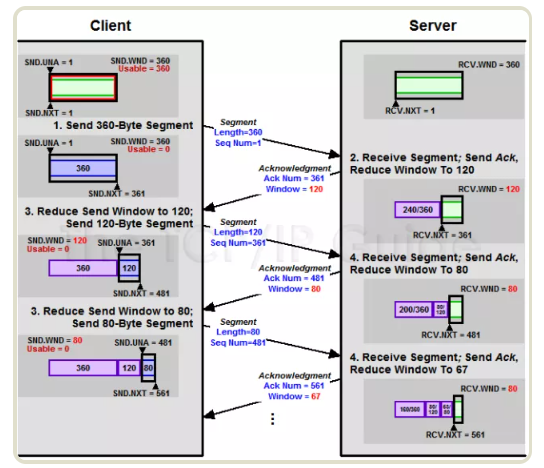
默认情况下，接收方每收到两个TCP 数据包，就要发送一个确认消息。"确认"的英语是 acknowledgement，所以这个确认消息就简称ACK。

ACK 携带两个信息：

期待要收到下一个数据包的编号

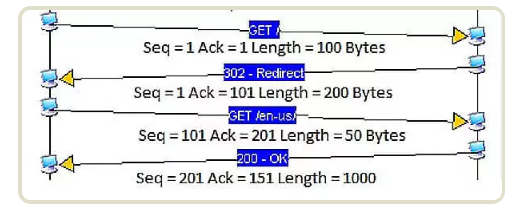
接收方的接收窗口的剩余容量

发送方有了这两个信息，再加上自己已经发出的数据包的最新编号，就会推测出接收方大概的接收速度，从而降低或增加发送速率。这被称为"发送窗口"，这个窗口的大小是可变的。



图片说明：每个ACK都带有下一个数据包的编号，以及接收窗口的剩余容量。双方都会发送ACK。

注意，由于TCP通信是双向的，所以双方都需要发送ACK。两方的窗口大小，很可能是不一样的。而且ACK只是很简单的几个字段，通常与数据合并在一个数据包里面发送。



图片说明：上图一共4次通信。第一次通信，A主机发给B主机的数据包编号是1，长度是100字节，因此第二次通信B主机的ACK编号是1 + 100 = 101，第三次通信A主机的数据包编号也是101。同理，第二次通信B主机发给 A 主机的数据包编号是1，长度是200字节，因此第三次通信A主机的ACK是201，第四次通信B主机的数据包编号也是201。

即使对于带宽很大、线路很好的连接，TCP也总是从10个数据包开始慢慢试，过了一段时间以后，才达到最高的传输速率。这就是TCP的慢启动。

##### 原理

**窗口大小由哪一方决定？**

TCP 头里有一个字段叫 Window，也就是窗口大小。

这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据。于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。

所以，通常窗口的大小是由接收方的决定的。

发送方发送的数据大小不能超过接收方的窗口大小，否则接收方就无法正常接收到数据

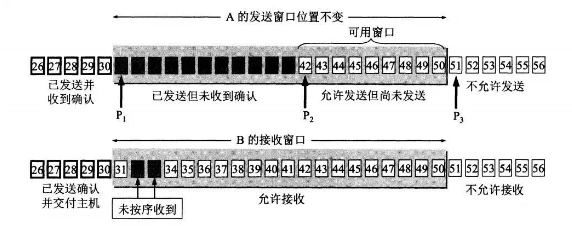
发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。如果发送窗口左部的字节已经发送并且收到了确认，那么就将发送窗口向右滑动一定距离，直到左部第一个字节不是已发送并且已确认的状态；接收窗口的滑动类似，接收窗口左部字节已经发送确认并交付主机，就向右滑动接收窗口。



**分析：**发送消息中的win 4096表示滑动窗口大小，指的是发送端/客户端窗口大小，服务端返回消息中也有一个win 6144表示服务端的滑动窗口大小。

客户端不断发送数据，当发送大小到达5121时，继续发送大小将达到6145，此时超出服务器的缓存大小，此时客户端不会继续发送，而是仅发送5121字节大小给服务端，然后等待服务端反馈。此时，服务器缓存接收完客户端5121字节数据后，进行数据的处理，然后腾出部分窗口，此时窗口剩余2048字节大小，此时会将窗口设置为2048发送给客户端，客户端判断是否满足自身数据要求，不满足则加倍。经过不断的处理之后，服务端的窗口（缓存）中的数据会全部处理完，即恢复原来的6144窗口大小。

接收窗口只会对窗口内最后一个按序到达的字节进行确认，例如接收窗口已经收到的字节为 {31, 34, 35}，其中 {31} 按序到达，而 {34, 35} 就不是，因此只对字节 31 进行确认。发送方得到一个字节的确认之后，就知道这个字节之前的所有字节都已经被接收。



注：滑动窗口其实就是调整服务器接收缓存的大小。

#### TCP流量控制

发送方不能无脑的发数据给接收方，要考虑接收方处理能力。

如果一直无脑的发数据给对方，但对方处理不过来，那么就会导致触发重发机制，从而导致网络流量的无端的浪费。

为了解决这种现象发生，TCP 提供一种机制可以让发送方根据接收方的实际接收能力控制发送的数据量，这就是所谓的流量控制。

流量控制是为了控制发送方发送速率，保证接收方来得及接收。

接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为0，则发送方不能发送数据。

实际上，为了避免此问题的产生，发送端主机会时不时的发送一个叫做**窗口探测**的数据段，此数据段仅包含一个字节来获取最新的窗口大小信息。

#### TCP拥塞控制过程

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

TCP 主要通过四个算法来进行拥塞控制：

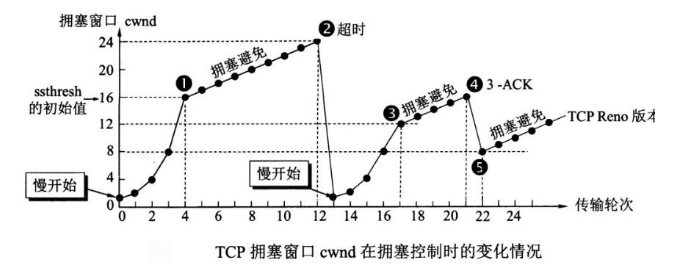
慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

为了便于讨论，做如下假设：

接收方有足够大的接收缓存，因此不会发生流量控制；

虽然TCP的窗口基于字节，但是这里设窗口的大小单位为报文段。



##### 慢开始

发送的最初执行慢开始，令cwnd = 1，发送方只能发送1个报文段；当收到确认后，将cwnd加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将cwnd加倍，这样会让cwnd增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限ssthresh，当cwnd >= ssthresh时，进入拥塞避免，每个轮次只将cwnd加1。

如果出现了超时，则令ssthresh = cwnd / 2，然后重新执行慢开始。

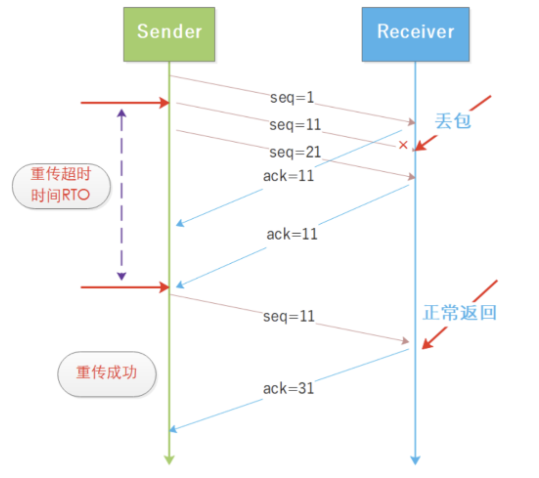
##### 拥塞避免

当慢启动阶段cwnd的值到达ssthresh时就不再疯狂增长，进入更加理性的线性阶段直至发送丢包，本次的阈值ssthresh是上一次发生丢包时cwnd的1/2，因此这是一个承上启下的过程。

本次发送丢包时仍然会调整ssthresh的值，具体拥塞避免增长过程：发送方每收到一个ACK数据包时将cwnd=cwnd+1/cwnd，每经过一个RTT将cwnd自增1。

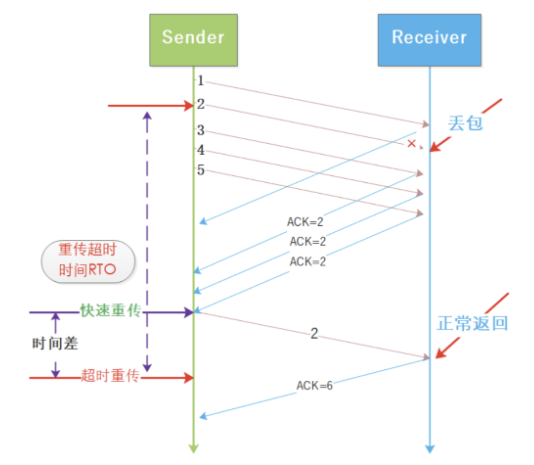
##### 快重传

TCP作为一个可靠的协议面临的很大的问题就是丢包，丢包就要重传因此发送方需要根据接收方回复的ACK来确认是否丢包了，并且发送方在发送数据之后启动定时器，如图所示：



RTO是随着复杂网络环境而动态变化的，在拥塞控制中发生超时重传将会极大拉低cwnd，如果网络状况并没有那么多糟糕，偶尔出现网络抖动造成丢包或者阻塞也非常常见，因此触发的慢启动将降低通信性能，故出现了快速重传机制。

所谓快速重传时相比超时重传而言的，重发等待时间会降低并且后续尽量避免慢启动，来保证性能损失在最小的程度，如图所示：



快速重传和超时重传的区别在于cwnd在发生拥塞时的取值，超时重传会将cwnd修改为最初的值，也就是慢启动的值，快速重传将cwnd减半，二者都将ssthresh设置为cwnd的一半。

从二者的区别可以看到，快速重传更加主动，有利于保证链路的传输性能，但是有研究表明3个ACK的机制同样存在问题。

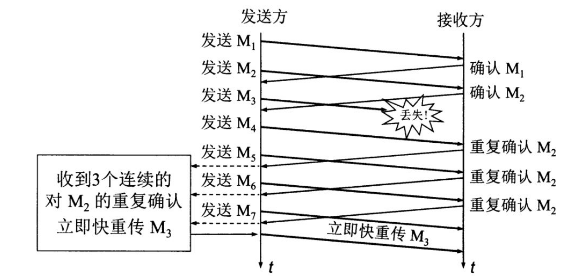
快速重传是基于对网络状况没有那么糟糕的假设，因此在实际网络确实还算好的时候，快速重传还是很有用的，在很差的网络环境很多算法都很难保证效率的。

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到M1和M2，此时收到M4，应当发送对M2的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个M2，则M3丢失，立即重传M3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令ssthresh = cwnd / 2，cwnd = ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是cwnd的设定值，而不是cwnd的增长速率。慢开始cwnd设定为1，而快恢复cwnd设定为ssthresh。



##### 快恢复

在快速重传之后就会进入快速恢复阶段，此时的cwnd为上次发生拥塞时的cwnd的1/2，之后cwnd再线性增加重复之前的过程。

#### 提高网络利用率

##### Nagle算法

Nagle算法确实能够在数据包较小时提高网络带宽的利用率并减少TCP和IP协议头带来的额外开销，但是使用该算法也可能会导致应用层协议多次写入的数据被合并或者拆分发送，当接收方从TCP协议栈中读取数据时会发现不相关的数据出现在了同一个数据段中，应用层协议可能没有办法对它们进行拆分和重组。

除了Nagle算法之外，TCP协议栈中还有另一个用于延迟发送数据的选项TCP\_CORK，如果我们开启该选项，那么当发送的数据小于MSS时，TCP协议就会延迟200ms发送该数据或者等待缓冲区中的数据超过MSS。

无论是TCP\_NODELAY还是TCP\_CORK，它们都会通过延迟发送数据来提高带宽的利用率，它们会对应用层协议写入的数据进行拆分和重组，而这些机制和配置能够出现的最重要原因是— TCP协议是基于字节流的协议，其本身没有数据包的概念，不会按照数据包发送数据。

发送端即使还有应该发送的数据，但如果这部分数据很少的话，则进行延迟发送的一种处理机制。具体来说，就是仅在下列任意一种条件下才能发送数据。如果两个条件都不满足，那么暂时等待一段时间以后再进行数据发送。

已发送的数据都已经收到确认应答。

可以发送最大段长度的数据时。

##### 延迟确认应答

接收方收到数据之后可以并不立即返回确认应答，而是延迟一段时间的机制。

在没有收到 2\*最大段长度的数据为止不做确认应答。

其他情况下，最大延迟0.5秒发送确认应答。

TCP文件传输中，大多数是每两个数据段返回一次确认应答。

##### 捎带应答

在一个 TCP 包中既发送数据又发送确认应答的一种机制（MQ可靠性中采用二次确认就是基于捎带应答），由此，网络利用率会提高，计算机的负荷也会减轻，但是这种应答必须等到应用处理完数据并将作为回执的数据返回为止。

#### TCP粘包/拆包

**为什么常说TCP有粘包和拆包的问题而不说UDP？**

UDP是基于报文发送的，UDP首部采用了16bit来指示UDP数据报文的长度，因此在应用层能很好的将不同的数据报文区分开，从而避免粘包和拆包的问题。

而TCP是基于字节流的，虽然应用层和TCP传输层之间的数据交互是大小不等的数据块，但是TCP并没有把这些数据块区分边界，仅仅是一连串没有结构的字节流；另外从TCP的帧结构也可以看出，在TCP的首部没有表示数据长度的字段，基于上面两点，在使用TCP传输数据时，才有粘包或者拆包现象发生的可能。

当应用层协议使用TCP协议传输数据时，TCP协议可能会将应用层发送的数据分成多个包依次发送，而数据的接收方收到的数据段可能有多个应用层数据包组成，所以当应用层从TCP缓冲区中读取数据时发现粘连的数据包时，需要对收到的数据进行拆分。

粘包并不是TCP协议造成的，它的出现是因为应用层协议设计者对TCP协议的错误理解，忽略了TCP协议的定义并且缺乏设计应用层协议的经验。

##### 简介

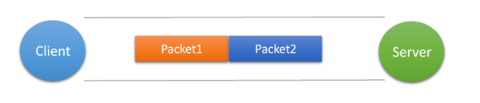
**什么是粘包、拆包？**

假设Client向Server连续发送了两个数据包，用packet1和packet2来表示，那么服务端收到的数据可以分为三种情况，现列举如下：

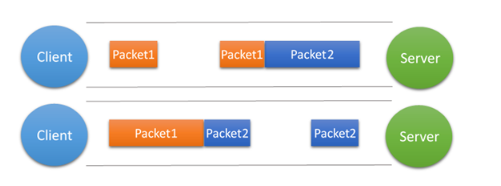
第一种情况，接收端正常收到两个数据包，即没有发生拆包和粘包的现象。



第二种情况，接收端只收到一个数据包，但是这一个数据包中包含了发送端发送的两个数据包的信息，这种现象即为**粘包**。这种情况由于接收端不知道这两个数据包的界限，所以对于接收端来说很难处理。



第三种情况，这种情况有两种表现形式，如下图。接收端收到了两个数据包，但是这两个数据包要么是不完整的，要么就是多出来一块，这种情况即发生了**拆包和粘包**。这两种情况如果不加特殊处理，对于接收端同样是不好处理的。



**发送方产生粘包**

采用TCP协议传输数据的客户端与服务器经常是保持一个长连接的状态（**一次连接发一次数据不存在粘包**），双方在连接不断开的情况下，可以一直传输数据。但当发送的数据包过于的小时，那么TCP协议默认的会启用**Nagle算法**，将这些较小的数据包进行合并发送（缓冲区数据发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说**数据发送出来它已经是粘包的状态了**。

**接收方产生粘包**

接收方采用TCP协议接收数据时的过程是这样的：数据到接收方，从网络模型的下方传递至传输层，传输层的TCP协议处理是将其放置接收缓冲区，然后由应用层来主动获取（C语言用recv、read等函数）；这时会出现一个问题，就是我们**在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包**。（放数据的速度>应用层拿数据速度）

##### 原因

TCP 协议粘包问题是因为应用层协议开发者的错误设计导致的，他们忽略了 TCP 协议数据传输的核心机制 — 基于字节流，其本身不包含消息、数据包等概念，所有数据的传输都是流式的，需要应用层协议自己设计消息的边界，即消息帧（Message Framing）。

**为什么会发生 TCP 粘包、拆包？**

要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。

待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。

要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。

接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

**产生的原因：**

1. TCP协议是面向字节流的协议（其中不存在消息和数据包的概念），它可能会组合或者拆分应用层协议的数据；
2. 应用层协议的没有定义消息的边界导致数据的接收方无法拼接数据，即应用层协议没有使用基于长度或者基于终结符的消息边界，导致多个消息的粘连；

##### 解决办法

TCP协议是面向连接的、可靠的、基于字节流的传输层通信协议，应用层交给TCP协议的数据并不会以消息为单位向目的主机传输，这些数据在某些情况下会被组合成一个数据段发送给目标的主机。

Nagle算法是一种通过减少数据包的方式提高TCP传输性能的算法。因为网络带宽有限，它不会将小的数据块直接发送到目的主机，而是会在本地缓冲区中等待更多待发送的数据，这种批量发送数据的策略虽然会影响实时性和网络延迟，但是能够降低网络拥堵的可能性并减少额外开销。

Nagle算法确实能够在数据包较小时提高网络带宽的利用率并减少TCP和IP协议头带来的额外开销，但是使用该算法也可能会导致应用层协议多次写入的数据被合并或者拆分发送，当接收方从TCP协议栈中读取数据时会发现不相关的数据出现在了同一个数据段中，应用层协议可能没有办法对它们进行拆分和重组。

除了Nagle算法之外，TCP协议栈中还有另一个用于延迟发送数据的选项TCP\_CORK，如果我们开启该选项，那么当发送的数据小于MSS时，TCP协议就会延迟200ms发送该数据或者等待缓冲区中的数据超过MSS。

无论是TCP\_NODELAY还是TCP\_CORK，它们都会通过延迟发送数据来提高带宽的利用率，它们会对应用层协议写入的数据进行拆分和重组，而这些机制和配置能够出现的最重要原因是：TCP协议是基于字节流的协议，其本身没有数据包的概念，不会按照数据包发送数据。

由于TCP本身是面向字节流的，无法理解上层的业务数据，所以在底层是无法保证数据包不被拆分和重组的，这个问题只能通过上层的应用协议栈设计来解决，根据业界的主流协议的解决方案，归纳如下：

1、**消息定长**：发送端将每个数据包封装为固定长度（在包头首都添加数据包的长度，不够的可以通过补0填充），这样接收端每次接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

2、**设置消息边界**：服务端从网络流中按消息边界分离出消息内容。在包尾增加回车换行符进行分割，例如FTP协议。

3、**将消息分为消息头和消息体**：消息头中包含表示消息总长度（或者消息体长度）的字段。

4、更复杂的应用层协议比如Netty中实现的一些协议都对粘包、拆包做了很好的处理。

**分包机制一般有两个通用的解决方法：**

1、特殊字符控制；

2、在包头首都添加数据包的长度。

如果使用 netty 的话，就有专门的编码器和解码器解决拆包和粘包问题了。

注意：UDP没有粘包问题，但是有丢包和乱序。不完整的包是不会有的，收到的都是完全正确的包。传送的数据单位协议是UDP报文或用户数据报，发送的时候既不合并，也不拆分。

### 应用

每一个应用层（TCP/IP参考模型的最高层）协议一般都会使用到两个传输层协议之一：

**运行在TCP协议上的协议：**

HTTP（HypertextTransferProtocol，超文本传输协议），主要用于普通浏览。

HTTPS（HTTP over SSL，安全超文本传输协议），HTTP协议的安全版本。

FTP（FileTransferProtocol，文件传输协议），用于文件传输。

POP3（PostOfficeProtocol,version3，邮局协议），收邮件用。

SMTP（SimpleMailTransferProtocol，简单邮件传输协议），用来发送电子邮件。

TELNET（Teletypeover theNetwork，网络电传），通过一个终端（terminal）登陆到网络。

SSH（SecureShell，用于替代安全性差的TELNET），用于加密安全登陆用。

**运行在 UDP协议上的协议：**

BOOTP（BootProtocol，启动协议），应用于无盘设备。

NTP（NetworkTimeProtocol，网络时间协议），用于网络同步。

DHCP（DynamicHostConfigurationProtocol，动态主机配置协议），动态配置IP地址。

**运行在 TCP和 UDP协议上：**

DNS（DomainNameService，域名服务），用于完成地址查找，邮件转发等工作。

## tcpdump

tcpdump命令是一款抓取数据包的工具，它可以打印所有经过网络接口的数据包的头信息，也可以使用-w选项将数据包保存到文件中，方便以后分析。

**命令格式：**

tcpdump（选项）

**选项：**

-a：尝试将网络和广播地址转换成名称；

-c<数据包数目>：收到指定的数据包数目后，就停止进行倾倒操作；

-d：把编译过的数据包编码转换成可阅读的格式，并倾倒到标准输出；

-dd：把编译过的数据包编码转换成C语言的格式，并倾倒到标准输出；

-ddd：把编译过的数据包编码转换成十进制数字的格式，并倾倒到标准输出；

-e：在每列倾倒资料上显示连接层级的文件头；

-f：用数字显示网际网络地址；

-F<表达文件>：指定内含表达方式的文件；

-i<网络界面>：使用指定的网络截面送出数据包；

-l：使用标准输出列的缓冲区；

-n：不把主机的网络地址转换成名字；

-N：不列出域名；

-O：不将数据包编码最佳化；

-p：不让网络界面进入混杂模式；

-q ：快速输出，仅列出少数的传输协议信息；

-r<数据包文件>：从指定的文件读取数据包数据；

-s<数据包大小>：设置每个数据包的大小；

-S：用绝对而非相对数值列出TCP关联数；

-t：在每列倾倒资料上不显示时间戳记；

-tt：在每列倾倒资料上显示未经格式化的时间戳记；

-T<数据包类型>：强制将表达方式所指定的数据包转译成设置的数据包类型；

-v：详细显示指令执行过程；

-vv：更详细显示指令执行过程；

-x：用十六进制字码列出数据包资料；

-w<数据包文件>：把数据包数据写入指定的文件。

# Socket

即套接字，是应用层与TCP/IP协议族通信的中间软件抽象层，表现为一个封装了TCP/IP协议族的编程接口（API）。



Socket不是一种协议，而是一个编程调用接口（API），**属于传输层**（主要解决数据如何在网络中传输）

即：通过Socket，我们才能在Andorid平台上通过 TCP/IP协议进行开发

对用户来说，只需调用Socket去组织数据，以符合指定的协议，即可通信。

成对出现，一对套接字：

Socket ={(IP地址1:PORT端口号)，(IP地址2:PORT端口号)}

一个Socket实例唯一代表一个主机上的一个应用程序的通信链路。

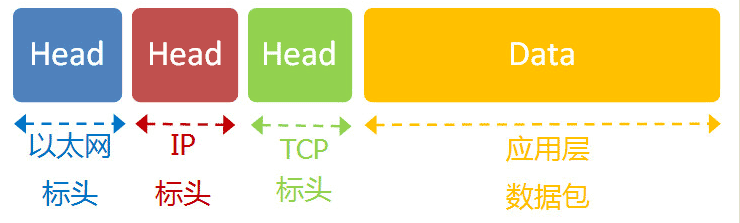
# 应用层

应用程序收到"传输层"的数据，接下来就要进行解读。由于互联网是开放架构，数据来源五花八门，必须事先规定好格式，否则根本无法解读。

"应用层"的作用，就是规定应用程序的数据格式。

举例来说，TCP协议可以为各种各样的程序传递数据，比如Email、WWW、FTP等等。那么，必须有不同协议规定电子邮件、网页、FTP数据的格式，这些应用程序协议就构成了"应用层"。

这是最高的一层，直接面对用户。它的数据就放在TCP数据包的"数据"部分。因此，现在的以太网的数据包就变成下面这样。



## HTTP协议

## FTP协议

## NFS协议

## Telnet协议

telnet命令用于登录远程主机，对远程主机进行管理。telnet因为采用明文传送报文，安全性不好，很多Linux服务器都不开放telnet服务，而改用更安全的ssh方式了。

**命令格式：**

telnet (选项) (参数)

-8：允许使用8位字符资料，包括输入与输出；

-a：尝试自动登入远端系统；

-b<主机别名>：使用别名指定远端主机名称；

-c：不读取用户专属目录里的.telnetrc文件；

-d：启动排错模式；

-e<脱离字符>：设置脱离字符；

-E：滤除脱离字符；

-f：此参数的效果和指定”-F”参数相同；

-F：使用Kerberos V5认证时，加上此参数可把本地主机的认证数据上传到远端主机；

-k<域名>：使用Kerberos认证时，加上此参数让远端主机采用指定的领域名，而非该主机的域名；

-K：不自动登入远端主机；

-l<用户名称>：指定要登入远端主机的用户名称；

-L：允许输出8位字符资料；

-n<记录文件>：指定文件记录相关信息；

-r：使用类似rlogin指令的用户界面；

-S<服务类型>：设置telnet连线所需的ip TOS信息；

-x：假设主机有支持数据加密的功能，就使用它；

-X<认证形态>：关闭指定的认证形态。

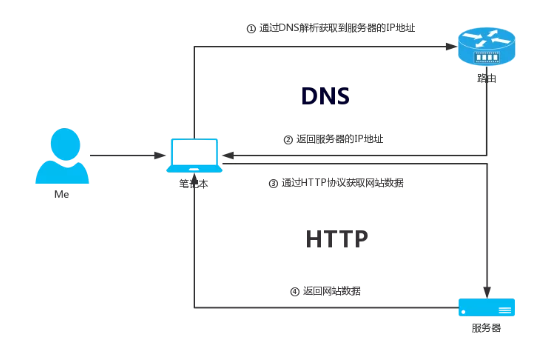
## DNS协议

DNS协议简单说就是为了将用户可读的域名转换为IP地址。域名就是IP地址的代号，为什么互联网会有域名这一说法，主要是为了方便记忆，也为了掩盖对象本身的真实信息，使信息对用户透明。

我们把用户访问网站的流程大概分为了两大部分：

DNS(用于解析域名的IP地址)

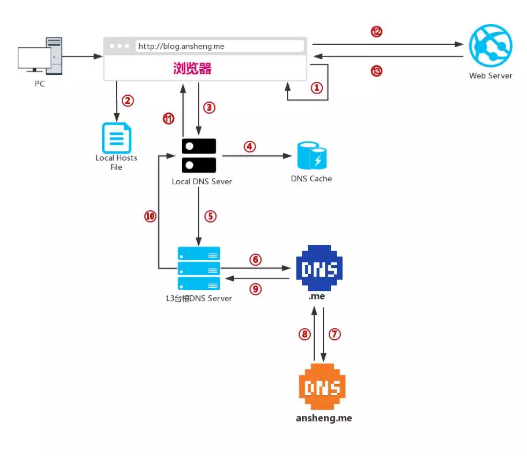
HTTP(得到IP地址之后从服务器获取数据)



DNS的作用：将机器的域名转换成IP地址。

### 原理

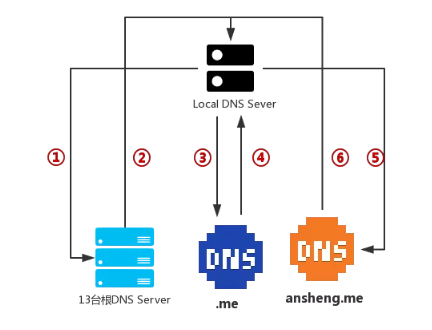
工作原理：对DNS的访问是通过一个地址解析器来完成的，解析器通过一个或多个名字服务器来完成这种相互转换。**DNS是应用层的协议**，在一个应用程序请求TCP打开一个连接或使用UDP发送一个数据之前，必须将一个主机名转换成一个IP地址。操作系统内核中的TCP/IP协议族对DNS一点都不知道。



当用户在浏览器输入http://blog.ansheng.me，是如何得到对应服务器的IP地址呢？具体步骤如下：

1. 浏览器会在**本地的缓存**中查找是否存在当前访问域名的缓存，如果有就返回数据，否则就继续第二步，通常情况下浏览器都会缓存网站的数据，为了就是加速访问；
2. 查看当前**操作系统的hosts文件**是否写有IP和域名对应的关系，如果有则得到IP地址，否则继续第三步；
3. 通过**本地的Local DNS**获取域名对应的IP地址，那么Local DNS其实就是我们的交换机或者路由器配置的DNS，通常情况下都有两个，一个主一个辅；
4. Local DNS会先看**缓存DNS**中是否有这条记录，如果没有就继续往下走，否则就返回IP地址；
5. Local DNS会继续请求**根服务器(.)**，根服务器全球只有十三台，根服务器也没有blog.ansheng.me域名的解析，但是根有.me域名的解析，然后根就会继续发起请求，当然只有在转发模式下才会这么做。
6. 根服务器请求.me服务器有没有blog.ansheng.me域名的解析，.me服务器服务器没有对应的解析，但是我有ansheng.me域名的解析，然后由.me服务器继续发起请求；
7. .meDNS服务器问ansheng.me DNS服务器有没有blog.ansheng.me域名的解析，ansheng.meDNS服务器说我有 blog.ansheng.me 域名的解析；
8. ansheng.me DNS服务器把 blog.ansheng.me 域名解析的IP地址返回给，.meDNS服务器；
9. .meDNS服务器把记录返回给根服务器；
10. 根服务器把记录返回给Local DNS服务器
11. Local DNS服务器会把记录缓存在DNS
12. Cache服务器上，并且把记录再次返回给浏览器；
13. 浏览器得到了对应的IP地址后就向blog.ansheng.me对应的服务器发起HTTP请求；
14. blog.ansheng.me服务器经过一些列的处理，最后把数据返回给浏览器，最后在页面上展示；

**下面是未启用转发模式工作流程：**



1. Local DNS服务器向根服务器发起请求问有没有blog.ansheng.me域名的解析记录；
2. 根服务器说我没有blog.ansheng.me域名的解析记录，但是我有.me的解析记录，并且把.me的解析记录返回给Local DNS服务器；
3. Local DNS服务器得到.me的服务器IP之后发起请求问：你有没有blog.ansheng.me\*域名的解析记录；
4. .me服务器说我没有，但是我有ansheng.me域名的解析记录，并且把IP地址返回给Local DNS服务器；
5. Local DNS服务器得到ansheng.me的服务器IP之后又发起请求问：你有没有blog.ansheng.me域名的解析记录；
6. ansheng.me的服务器说我有blog.ansheng.me域名的解析记录，并且把对应解析的记录返回给Local DNS服务器

不管用的是转发模式，还是根提示，最后都是把结果返回给Local DNS服务器，由此DNS服务器再返回给客户机。

**从浏览器到本地DNS服务器是属于递归查询，而DNS服务器之间就是的交互查询就是迭代查询。**

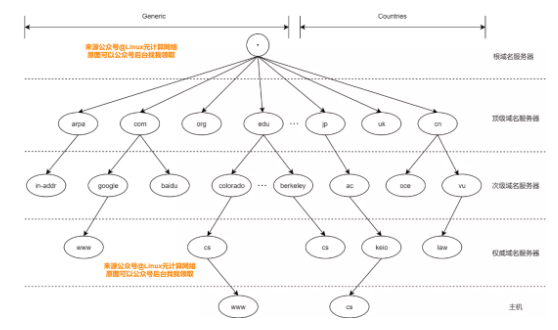
### 分类

为了根据域名得到IP地址，需要建立一个域名和IP地址之间的映射关系，简单点就是用一个文件来完成，这种叫**静态映射** ，复杂点就是用专门的服务器来完成，这种叫**动态映射**，提供动态映射服务的服务器就叫DNS服务器 。

注：windows用户比较熟悉的静态映射方法就是通过hosts文件，位置在 C:\Windows\System32\drivers\etc\hosts，只要在hosts文件中写上你想访问的域名和地址，系统内置的DNS解析器会优先解析，不会访问DNS服务器。Linux下是/etc/hosts。

### DNS服务器

由于存在单点故障，DNS服务器不可能设计成集中式的，由于需要解析的域名信息众多且复杂，DNS服务器必须足够多，为了方便管理，DNS服务器最后呈现的是具有层次结构的大型分布式集群系统。如下图所示：



总体形成一个分级的树状体系，每个节点为一个DNS服务器，每个节点都有自己的IP地址。每一层的角色不一样，从上到下分别表示：

根域名服务器（Root Domain）：用·（句点）表示，根域名服务器全球的数量是固定的，为13个（当然这里的“个”是“组”的意思），至于为什么是13个？

顶级域名服务器（Top-Level Domain, TLD）：指代某个国家/地区或组织使用的类型名称，如com、cn、edu等。

次级域名服务器（Second-Level Domain, SLD）：个人或组织在Internet上注册的名称，如qq.com、gitHub.com等

三次域名服务器或权威域名服务器（如果有）：这层严格来说是次级域名的子域，是二层域名派生的域名，通俗说就是网站名，如cs.berkeley.edu

主机名（如果有）：主机名称标签，通常在 DNS 域名最左侧，标识网络上的特定计算机，如www.berkeley.edu

附：13 个根服务器的配置文件：https://www.internic.net/domain/named.root

DNS服务器有一个区域的概念。由于DNS服务器要为全球的用户提供查询和数据库的服务，对计算、存储、网络带宽都要求很高，所以就必须组织成域名服务器集群，使它们协同工作，共同提供域名解析服务。每个集群就负责管理相应的域名名字空间，也就是说每个集群负责不同的DNS区域（DNS Zone）。比如每个根域名服务器节点指向的下一级就是它管理的DNS区域，同样再往下，cn顶级域名服务器节点下面就管理中国区的DNS区域。

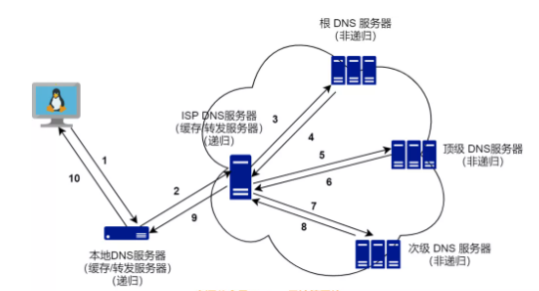
通过这种区域管理的方法，就很容易进行DNS查询。每一级的DNS区域域名服务器会保存下一级的DNS区域域名服务器的信息，如果从根域名服务器出发，一级级的往下查找，就可以得到目标域名对应的IP信息。

可能你会有问题，怎么找到根域名服务器呢？

上面已经提到过，全球的根域名服务器总的只有13个，这13个的信息都是公开的，也就是说每一个用户的出口DNS服务器上都有这些根域名服务器的已知信息。有了这些信息，查找就不是问题了。

除了以上所列的 DNS 服务器，严格来说还有三种 DNS 服务器角色不容忽视。一种是本地DNS服务器，一种是ISP DNS服务器，还有一种是公共DNS服务器。

一个大概的图示如下所示：



**为什么根域名服务器全球只有13组？**

主要是基于当时的网络环境，从UDP分片上考虑，保证报文能畅通无阻。

绝大多数的网络接口类型支持IP报文≤576 字节无需分片自由通行，考虑到以上诸因素，IETF决定将DNS报文体限制在512字节。每一个根域名服务器占用32字节，其中包括根域名的名称、IP地址、TTL(Time To Live)等参数。

13根域名服务器一共占用416字节，剩余的96字节用于包装DNS报文头以及其它协议参数。所以从空间上来说，没有多余的空间容纳第14个根域名服务器的32字节。

容易被大众误解的是，这13个根域名服务器并不等于13台物理服务器，而是代表着13个全球IP地址，由13个机构来管理，对应有不同的名字，分别为“A” 到 “M”，其中美国最大电信运营商Verizon管理两个根域名全球IP地址。

截至到（2019.09）为止，全球一共有 1011台服务器实例（Instances），遍布 5大洲 4大洋。

#### 本地DNS服务器

本地DNS服务器 ，特指内网的DNS服务器，提供内网主机之间的域名查询服务，一般作为缓存/转发之用，当在该服务器上找不到相应域名时，它会将请求转发到上级DNS服务器作进一步查询。它会依赖于两个文件，以Linux为例，就是/etc/hosts和/etc/resolv.conf，前者记录了内网主机hostname和IP之间的映射关系，后者记录了外网的DNS服务器地址。

当本地DNS服务器在/etc/hosts和自身的缓存中都找不到相应的域名时，就会从/etc/resolv.conf中记录的地址进行进一步查询。

这个角色也可以用软件的形式来实现，我们常见的Linux操作系统中内置的DNS域名解析模块（DNS Resolver）或者类似Dnsmasq这种软件都可以完成。

#### ISP DNS服务器

ISP DNS服务器 ，如果你的电脑是连接到运营商（ISP）的网络，那么外网DNS服务器就是指代的ISP DNS服务器。ISP DNS服务器相比本地DNS服务器，多了递归和迭代向各分级DNS服务器查询的功能。

#### 公共DNS服务器

公共DNS服务器 ，如果不想通过ISP的DNS服务器查询，也可以向公共的DNS服务器去查询。现在很多公共的DNS服务器，相信大家应该很熟悉了，随便列几个：

# Google 提供公共 DNS：

8.8.8.8 # 主

8.8.8.4 # 备

# 国内三大运营商提供的公共 DNS：

114.114.114.114

114.114.115.115

# 阿里提供的

223.5.5.5

223.6.6.6

# 百度提供的：

180.76.76.76

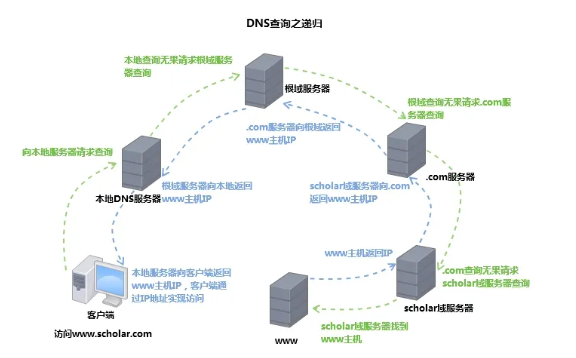
当然，在实际中，我们也经常看到，这三种服务器角色都可以看做是一个整体，没有那么严格地区分。

### DNS查询

DNS 查询有两种方法，一种是递归查询，一种是迭代查询。

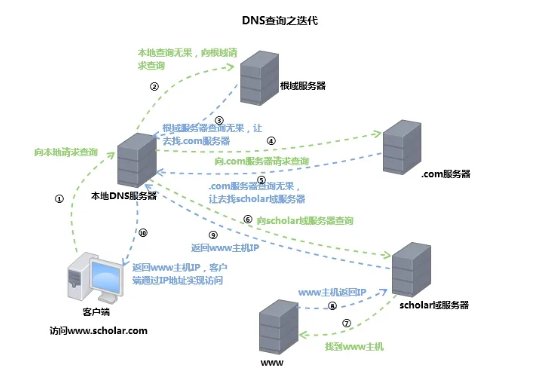
#### 递归查询

递归查询就是一层接一层地去查询，直到查询到最终的结果之后才返回。如下图所示：



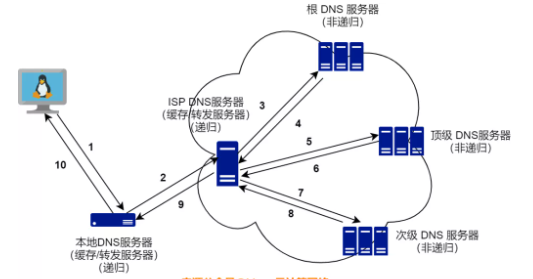
#### 迭代查询

迭代查询就是每一层查询完之后都会返回，返回的结果是下一层DNS服务器的地址，根据返回的结果，接着去查询下一层，直到查询到最终的结果为止，如下图所示：



实际中，根域名服务器和其他的TLD服务器，它们不执行递归查询，因为这些服务器是相当宝贵的资源，需要为全球的用户提供 DNS 解析服务，执行DNS递归查询会导致全球互联网性能不佳。

所以，通常这两种查询方式是一起用的，本地DNS服务器和ISP DNS服务器可以执行递归查询，而其他层级DNS服务器执行迭代查询，对照上面的图：



我们总结整个查询过程如下：

1. 本地终端发出域名解析请求，到本地 DNS 服务器（如果是 web 请求，会先查看浏览器缓存）
2. 本地 DNS 服务器首先查看自身缓存，如果不存在则向 ISP DNS 服务器发出查询请求
3. ISP DNS 服务器同样先查看缓存，不存在再向众所周知的全球 13 台根服务器发出请求
4. 根 DNS 服务器收到请求后会判断该域名（比如.com）由谁授权管理，返回管理这个域名的顶级 DNS 服务器的 IP，给到 ISP DNS 服务器
5. ISP DNS 服务器根据返回的 IP 继续请求顶级 DNS 服务器
6. 该顶级 DNS 服务器收到请求后，如果自己无法解析，也会判断对应域名（比如qq.com）由下一级的哪个 DNS 服务器授权管理，并将该次级 DNS 服务器的 IP 发给 ISP DNS 服务器。
7. ISP DNS 服务器继续根据返回的 IP 请求次级 DNS 服务器
8. 次级 DNS 服务器如果解析出对应的域名（比如 www.qq.com），就将该域名对应的 IP 返回给 ISP DNS 服务器，如果没有，就继续重复上述动作。
9. ISP DNS 服务器缓存一份域名与 IP 的映射关系，并将结果返回给本地 DNS 服务器
10. 本地 DNS 服务器同样会缓存一份，然后将结果给到请求的终端，完成本次查询。

### DNS缓存

我们都知道大名鼎鼎的 80/20 原则，用在网站访问上就是 80% 的时间我们都在看那些 20% 的网站 。

所以，如果把常用的信息缓存起来，那么后面的请求将会大大缩短访问时间。对于 DNS 请求来说，如果中间的 DNS 服务器都设有缓存，缓存那些经常会用到的域名信息，那将大大提高 DNS 的解析效率。

在 DNS 查询解析的过程会用到的 DNS 缓存有：

浏览器缓存：当用户通过浏览器访问某域名时，浏览器首先会在自己的缓存中查找是否有该域名对应的IP地址

系统缓存：一般 DNS Resolver 模块通常设有 DNS cache，hosts 文件也可以看做是缓存

1. 本地 DNS 缓存（如果有）：这里特指硬件服务器，和系统缓存对应
2. 路由器缓存：一般现在的出口路由器也会设有 DNS 缓存，当以上都查询不到，会在路由器缓存中查询
3. ISP DNS 缓存：ISP 网络提供的 DNS 服务器缓存。
4. 根域名/顶级域名/次级域名等缓存

当完成一次完整的查询之后，中间的 DNS 服务器缓存会将返回的结果一次存放入自己的缓存中，以备下次使用。

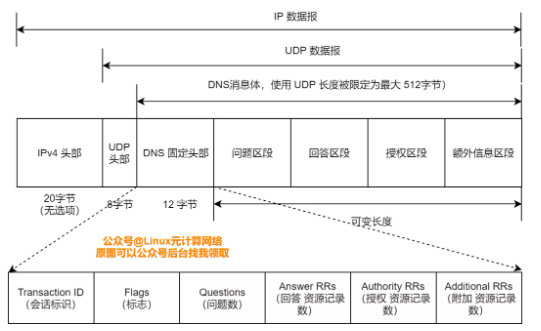
DNS 缓存当然能够加快 DNS 查询的效率，但有时也会影响网络问题排查。这个时候需要将 DNS 缓存清除，清除的方法如下：

1. 对于 Windows 系统，在命令行输入 ipconfig /flushdns 即可
2. 对于 Linux 系统，看有没有使用 dnsmasq 之类的缓存服务器，如果使用的话，只需重启缓存服务即可，如重启 dnsmasq 服务：sudo systemctl restart dnsmasq.service，如果没有使用，则根据各大 Linux 发行版各自的缓存服务来清除，比如 ubuntu 系统，默认使用 systemd 解析的服务来缓存 DNS 条目，可以通过 sudo systemd-resolve --flush-caches 来清除缓存
3. 对于 Web 浏览器的 DNS 缓存，可以在地址栏输入 chrome://net-internals/#dnsChrome，然后再点击 "clear host cache"，也可以通过 Ctrl+Shift+Del 打开清除浏览器数据窗口进行清除

### DNS协议格式

DNS 是用户层协议，传输层主要使用UDP协议，只有大包才可能使用TCP协议。

首先看看DNS使用UDP协议整个报文的格式如下：



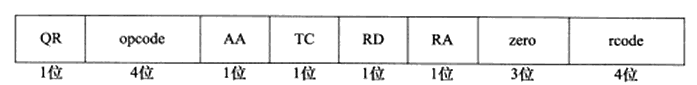
DNS报文由12字节固定长度的首部和4个长度可变的字段组成。使用UDP时，整个DNS报文长度限定为512字节，如果使用TCP或者扩展域，DNS消息体的长度可以进行扩展，并在传输时借助TCP来分段。

关于DNS报文每一个字段的释义，可以参考《TCP/IP 详解 · 卷一》。

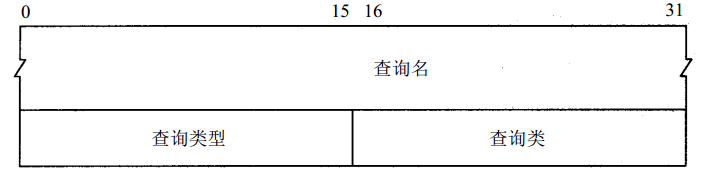
**DNS查询/应答报文：**



**DNS报文头部的标志字段：**



**DNS查询问题的格式：**



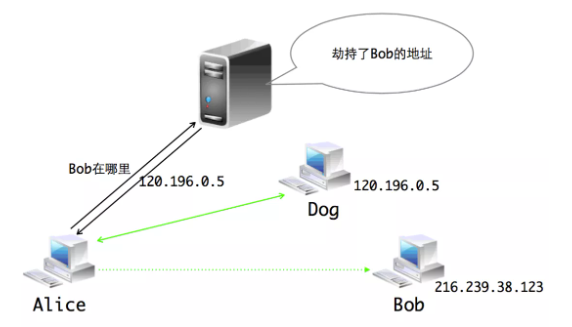
### DNS安全问题

DNS 安全问题，总的可以总结出以下几个：

1. 域名抢注：有些黄牛会批量抢注大量域名，然后转身高价卖给那些对域名感兴趣的人，谋取暴力
2. DNS Cache污染：中间DNS服务器将上级服务器返回结果进行修改，给客户端一个改变了（污染）的信息
3. DNS劫持：和DNS Cache污染比较像，不过这种可以被外部第三者劫持进行恶意修改
4. DNS欺骗：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，并且忽略真正的DNS应答
5. DNS放大攻击：这是一种DDoS攻击，利用DNS回复包比请求包大的特点，放大流量，伪造请求包的源IP地址为受害者IP，将应答包的流量引入受害的服务器
6. 系统上运行的DNS服务存在漏洞，导致被黑客获取权限，从而篡改DNS信息
7. DNS设置不当，导致泄漏一些敏感信息。提供给黑客进一步攻击提供有力信息

#### DNS劫持

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。如下图所示：



针对DNS劫持，我们可以简单地更换域名服务器，比较靠谱的一个是Google 提供的 8.8.8.8。下面用8.8.8.8 来解析一下www.google.com 就能看到正确的地址了。

$ nslookup www.google.com 8.8.8.8

Server:8.8.8.8

Address:8.8.8.8#53

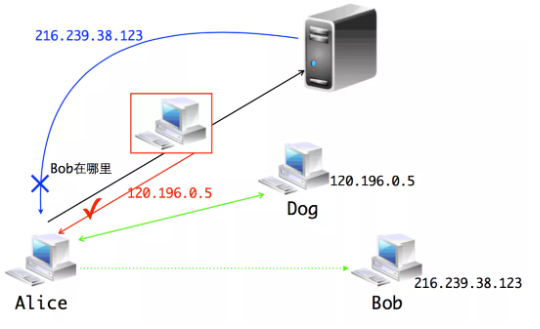
Non-authoritative answer:

Name:www.google.com

Address: 216.58.221.68

#### 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。

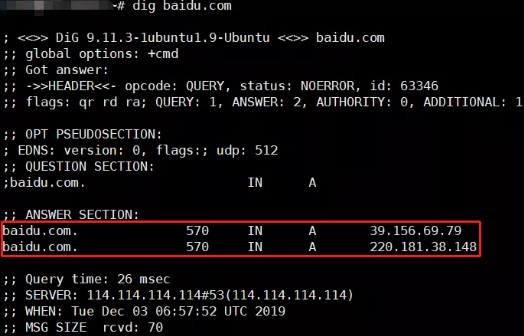


### DNS应用

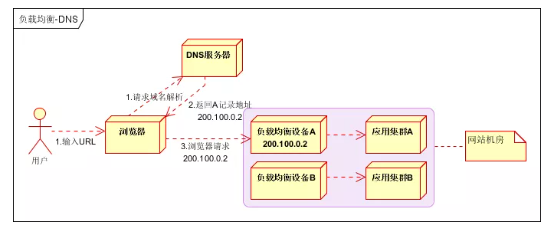
#### DNS负载均衡器

利用 DNS 服务器实现访问的负载均衡，这是一种较早的负载均衡技术。

具体做法就是在DNS服务器中，配置多个A记录，这些A记录对应的服务器构成集群，然后就可以利用DNS轮询解析出不同的IP地址，以此实现最简单的轮询式负载均衡。比如dig baidu.com，我们可以看到baidu.com对应多个A记录：



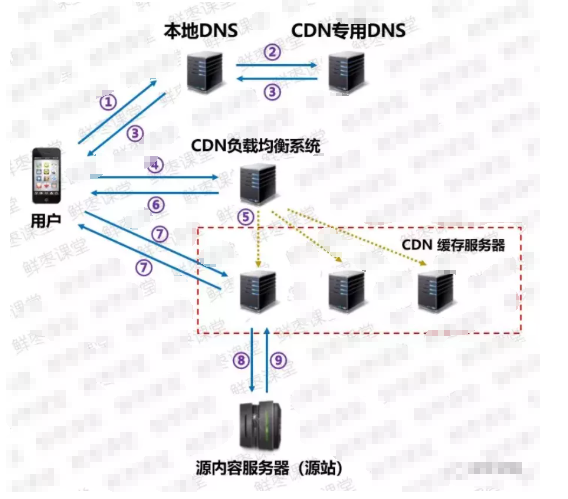
通常的做法是将DNS服务器作为第一级的负载均衡设备，A记录对应内部负载均衡器或服务器的IP地址，通过请求DNS服务器，将流量引导内部的负载均衡器或服务器上，作进一步处理。



但这种方案缺点也很明显，就是DNS有缓存，如果后端的某个机器出故障，域名解析还是返回那个机器的IP，那所有的访问都会出问题，所以现在的LB都不采用这种DNS解析这种做法了。

#### CDN专用DNS

DNS服务器作为CDN专用访问服务器，也是作为第一级负载均衡器来使用。CDN专用DNS服务器会缓存CDN的全局负载均衡设备的IP地址，而CDN全局负载均衡设备知晓全局范围内的所有CDN缓存服务器的地址，从而实现就近访问。如下图所示：



具体步骤：

1、当用户点击APP上的内容，APP会根据URL地址去本地DNS（域名解析系统）寻求IP地址解析。

2、本地DNS系统会将域名的解析权交给CDN专用DNS服务器。

3、CDN专用DNS服务器，将CDN的全局负载均衡设备IP地址返回用户。

4、用户向CDN的负载均衡设备发起内容URL访问请求。

5、CDN负载均衡设备根据用户IP地址，以及用户请求的内容URL，选择一台用户所属区域的缓存服务器。

6、负载均衡设备告诉用户这台缓存服务器的IP地址，让用户向所选择的缓存服务器发起请求。

7、用户向缓存服务器发起请求，缓存服务器响应用户请求，将用户所需内容传送到用户终端。

8、如果这台缓存服务器上并没有用户想要的内容，那么这台缓存服务器就要网站的源服务器请求内容。

9、源服务器返回内容给缓存服务器，缓存服务器发给用户，并根据用户自定义的缓存策略，判断要不要把内容缓存到缓存服务器上。

#### 服务发现

DNS能够将一个服务的名字转换为IP这种特性，天然就适合作为一种服务发现手段。在目前的集群场景中，很多项目都会使用DNS来为集群提供服务发现的功能，比如Kubernetes集群中，就常使用KubeDNS和CoreDNS这两个DNS项目来解决服务发现的问题。

从Kubernetes v1.12开始，CoreDNS便取代了Kube-DNS，成为Kubernetes推荐的服务发现项目，CoreDNS是Golang编写的一个插件式DNS服务器。

关于CoreDNS：

https://coredns.io/manual/toc/

### DNS工具

常用的DNS调试工具：

dig

host

whois

nslookup

这三个命令都属于bind-utils包，CentOS系用户可以使用yum install bind-utils安装。

#### dig

dig(Domain Information Groper，域名信息搜索)，是一款Unix/BSD系统自带的DNS诊断工具。

使用dig查询从根域名到指定域名中间可能经过的所有域名服务器，使用+trace选项。

#### host

host命令是常用的分析域名查询工具，可以检测域名系统工作是否正常。

**命令格式：**

host（选项）（参数）

**选项：**

-a：显示详细的DNS信息

-c<类型>：指定查询类型，默认值为“IN”

-C：查询指定主机的完整SOA记录

-r：在查询域名时，不使用递归的查询方式

-t<类型>：指定查询第域名信息类型

-v：显示指令执行的详细信息

-a：显示详细的DNS信息；

-w：如果域名服务器没有给出应答信息，则一直等待，直到域名服务器给出应答

-W<时间>：指定域名查询的最长时间，如果在指定时间内域名服务器没有给出应答信息，则退出指令

-4：使用IPv4

-6：使用IPv6

**参数：**

主机：要查询的主机信息

host可以看做是dig的简化版，主要将域名解析成IP地址或将IP地址解析成域名。

1、将域名解析为 IP 地址：

# host baidu.com

baidu.com has address 39.156.69.79

baidu.com has address 220.181.38.148

baidu.com mail is handled by 20 jpmx.baidu.com.

baidu.com mail is handled by 10 mx.maillb.baidu.com.

baidu.com mail is handled by 20 mx50.baidu.com.

baidu.com mail is handled by 20 mx1.baidu.com.

baidu.com mail is handled by 15 mx.n.shifen.com.

可以看到除了返回 IP 地址，还返回域名相关的各种记录。

2、将 IP 地址解析为域名

DNS 逆向查询，等同于 dig -x。

# host 13.229.188.59

59.188.229.13.in-addr.arpa domain name pointer ec2-13-229-188-59.ap-southeast-1.compute.amazonaws.com.

whois

查询域名的注册情况。

# whois coolshell.cn

Domain Name: coolshell.cn

ROID: 20090825s10001s91994755-cn

Domain Status: ok

Registrant ID: hc401628324-cn

Registrant: 陈皓

Registrant Contact Email: haoel@hotmail.com

Sponsoring Registrar: 阿里云计算有限公司（万网）

Name Server: f1g1ns1.dnspod.net

Name Server: f1g1ns2.dnspod.net

Registration Time: 2009-08-25 00:40:26

Expiration Time: 2023-08-25 00:40:26

DNSSEC: unsigned

#### whois

查询域名的注册情况。

# whois coolshell.cn

Domain Name: coolshell.cn

ROID: 20090825s10001s91994755-cn

Domain Status: ok

Registrant ID: hc401628324-cn

Registrant: 陈皓

Registrant Contact Email: haoel@hotmail.com

Sponsoring Registrar: 阿里云计算有限公司（万网）

Name Server: f1g1ns1.dnspod.net

Name Server: f1g1ns2.dnspod.net

Registration Time: 2009-08-25 00:40:26

Expiration Time: 2023-08-25 00:40:26

DNSSEC: unsigned

#### nslookup

nslookup也是一款DNS诊断工具，几乎所有平台带这款工具，使用非常方便。

nslookup的优势在可以交互式地查询域名记录，比如：

# nslookup

> github.com

Server:114.114.114.114

Address:114.114.114.114#53

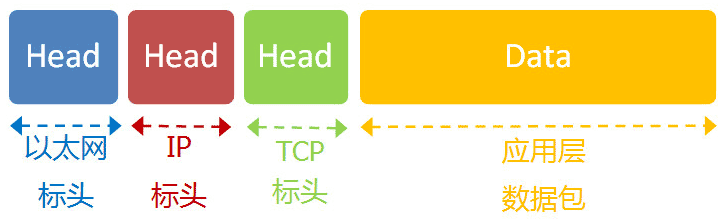
Non-authoritative answer:

Name:github.com

Address: 13.229.188.59

>

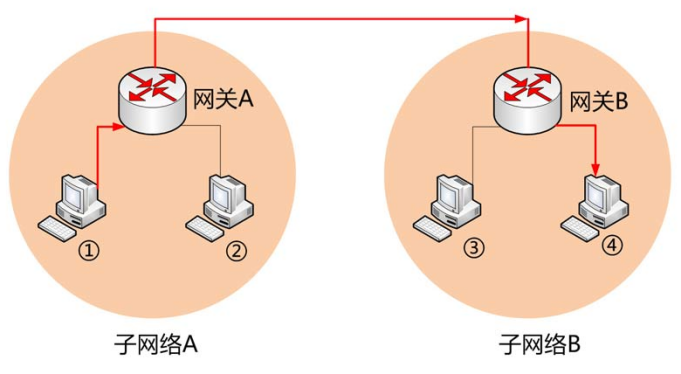
# 小结

我们已经知道，网络通信就是交换数据包。电脑A向电脑B发送一个数据包，后者收到了，回复一个数据包，从而实现两台电脑之间的通信。数据包的结构，基本上是下面这样：

发送这个包，需要知道两个地址：

　　\* 对方的MAC地址

　　\* 对方的IP地址

有了这两个地址，数据包才能准确送到接收者手中。但是，前面说过，MAC地址有局限性，如果两台电脑不在同一个子网络，就无法知道对方的MAC地址，必须通过网关（gateway）转发。

上图中，1号电脑要向4号电脑发送一个数据包。它先判断4号电脑是否在同一个子网络，结果发现不是（后文介绍判断方法），于是就把这个数据包发到网关A。网关A通过路由协议，发现4号电脑位于子网络B，又把数据包发给网关B，网关B再转发到4号电脑。

1号电脑把数据包发到网关A，必须知道网关A的MAC地址。所以，数据包的目标地址，实际上分成两种情况：

|  |  |
| --- | --- |
| 场景 | 数据包地址 |
| 同一个子网络 | 对方的MAC地址，对方的IP地址 |
| 非同一个子网络 | 网关的MAC地址，对方的IP地址 |

发送数据包之前，电脑必须判断对方是否在同一个子网络，然后选择相应的MAC地址。接下来，我们就来看，实际使用中，这个过程是怎么完成的。

# 用户的上网设置

## 静态IP地址

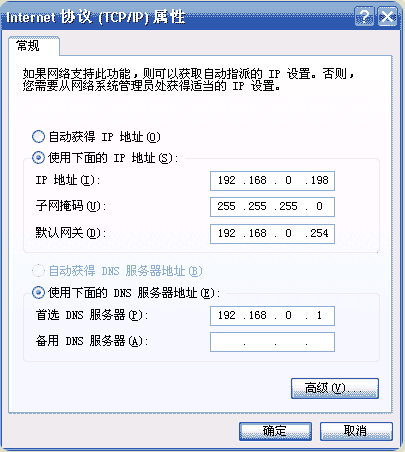
通常你必须做一些设置。有时，管理员（或者ISP）会告诉你下面四个参数，你把它们填入操作系统，计算机就能连上网了：

　　\* 本机的IP地址

　　\* 子网掩码

　　\* 网关的IP地址

　　\* DNS的IP地址

下图是Windows系统的设置窗口。

这四个参数缺一不可，后文会解释为什么需要知道它们才能上网。由于它们是给定的，计算机每次开机，都会分到同样的IP地址，所以这种情况被称作"静态IP地址上网"。

但是，这样的设置很专业，普通用户望而生畏，而且如果一台电脑的IP地址保持不变，其他电脑就不能使用这个地址，不够灵活。出于这两个原因，大多数用户使用"动态IP地址上网"。

## 动态IP地址

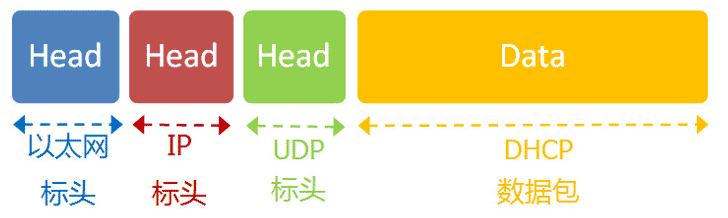
所谓"动态IP地址"，指计算机开机后，会自动分配到一个IP地址，不用人为设定。它使用的协议叫做DHCP协议。

这个协议规定，每一个子网络中，有一台计算机负责管理本网络的所有IP地址，它叫做"DHCP服务器"。新的计算机加入网络，必须向"DHCP服务器"发送一个"DHCP请求"数据包，申请IP地址和相关的网络参数。

前面说过，如果两台计算机在同一个子网络，必须知道对方的MAC地址和IP地址，才能发送数据包。但是，新加入的计算机不知道这两个地址，怎么发送数据包呢？

DHCP协议做了一些巧妙的规定。

## DHCP协议

首先，它是一种应用层协议，建立在UDP协议之上，所以整个数据包是这样的：

（1）最前面的"以太网标头"，设置发出方（本机）的MAC地址和接收方（DHCP服务器）的MAC地址。前者就是本机网卡的MAC地址，后者这时不知道，就填入一个广播地址：FF-FF-FF-FF-FF-FF。

（2）后面的"IP标头"，设置发出方的IP地址和接收方的IP地址。这时，对于这两者，本机都不知道。于是，发出方的IP地址就设为0.0.0.0，接收方的IP地址设为255.255.255.255。

（3）最后的"UDP标头"，设置发出方的端口和接收方的端口。这一部分是DHCP协议规定好的，发出方是68端口，接收方是67端口。

这个数据包构造完成后，就可以发出了。以太网是广播发送，同一个子网络的每台计算机都收到了这个包。因为接收方的MAC地址是FF-FF-FF-FF-FF-FF，看不出是发给谁的，所以每台收到这个包的计算机，还必须分析这个包的IP地址，才能确定是不是发给自己的。当看到发出方IP地址是0.0.0.0，接收方是255.255.255.255，于是DHCP服务器知道"这个包是发给我的"，而其他计算机就可以丢弃这个包。

接下来，DHCP服务器读出这个包的数据内容，分配好IP地址，发送回去一个"DHCP响应"数据包。这个响应包的结构也是类似的，以太网标头的MAC地址是双方的网卡地址，IP标头的IP地址是DHCP服务器的IP地址（发出方）和255.255.255.255（接收方），UDP标头的端口是67（发出方）和68（接收方），分配给请求端的IP地址和本网络的具体参数则包含在Data部分。

新加入的计算机收到这个响应包，于是就知道了自己的IP地址、子网掩码、网关地址、DNS服务器等等参数。

## 上网设置：小结

这个部分，需要记住的就是一点：不管是"静态IP地址"还是"动态IP地址"，电脑上网的首要步骤，是确定四个参数。这四个值很重要，值得重复一遍：

　　\* 本机的IP地址

　　\* 子网掩码

　　\* 网关的IP地址

　　\* DNS的IP地址

有了这几个数值，电脑就可以上网"冲浪"了。接下来，我们来看一个实例，当用户访问网页的时候，互联网协议是怎么运作的。

# CS/BS模型

C/S模型优缺点：

1. 优点
   1. 缓存大量数据
   2. 协议选择灵活
   3. 速度快
2. 缺点
   1. 安全性
   2. 开发工作量大（需要维护客户端和服务端两套程序）

B/S模型优缺点：

1. 优点
   1. 安全性
   2. 跨平台
   3. 开发工作量较小
2. 缺点
   1. 不能缓存大量数据
   2. 严格遵守HTTP协议

# 一个实例：访问网页

打开一个网页，整个过程会使用哪些协议？



## 本机参数

我们假定，经过上一节的步骤，用户设置好了自己的网络参数：

　　\* 本机的IP地址：192.168.1.100

　　\* 子网掩码：255.255.255.0

　　\* 网关的IP地址：192.168.1.1

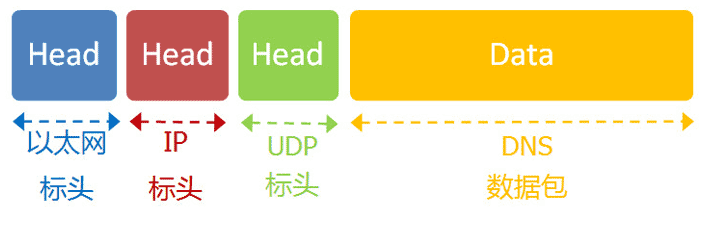
　　\* DNS的IP地址：8.8.8.8

然后他打开浏览器，想要访问Google，在地址栏输入了网址：www.google.com。

这意味着，浏览器要向Google发送一个网页请求的数据包。

## DNS协议

我们知道，发送数据包，必须要知道对方的IP地址。但是，现在，我们只知道网址www.google.com，不知道它的IP地址。

DNS协议可以帮助我们，将这个网址转换成IP地址。已知DNS服务器为8.8.8.8，于是我们向这个地址发送一个DNS数据包（53端口）。

然后，DNS服务器做出响应，告诉我们Google的IP地址是172.194.72.105。于是，我们知道了对方的IP地址。

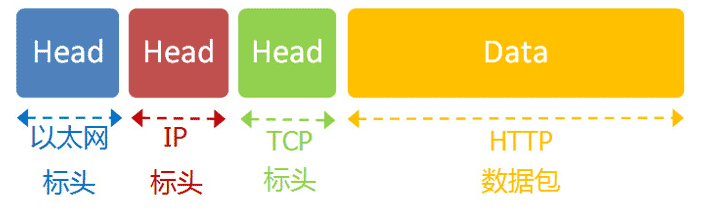
## 子网掩码

接下来，我们要判断，这个IP地址是不是在同一个子网络，这就要用到子网掩码。

已知子网掩码是255.255.255.0，本机用它对自己的IP地址192.168.1.100，做一个二进制的AND运算（两个数位都为1，结果为1，否则为0），计算结果为192.168.1.0；然后对Google的IP地址172.194.72.105也做一个AND运算，计算结果为172.194.72.0。这两个结果不相等，所以结论是，Google与本机不在同一个子网络。

因此，我们要向Google发送数据包，必须通过网关192.168.1.1转发，也就是说，接收方的MAC地址将是网关的MAC地址。

## 应用层协议

浏览网页用的是HTTP协议，它的整个数据包构造是这样的：

HTTP部分的内容，类似于下面这样：

　　GET / HTTP/1.1

　　Host: www.google.com

　　Connection: keep-alive

　　User-Agent: Mozilla/5.0 (Windows NT 6.1) ......

　　Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

　　Accept-Encoding: gzip,deflate,sdch

　　Accept-Language: zh-CN,zh;q=0.8

　　Accept-Charset: GBK,utf-8;q=0.7,\*;q=0.3

　　Cookie: ... ...

我们假定这个部分的长度为4960字节，它会被嵌在TCP数据包之中。

## TCP协议

TCP数据包需要设置端口，接收方（Google）的HTTP端口默认是80，发送方（本机）的端口是一个随机生成的1024-65535之间的整数，假定为51775。

TCP数据包的标头长度为20字节，加上嵌入HTTP的数据包，总长度变为4980字节。

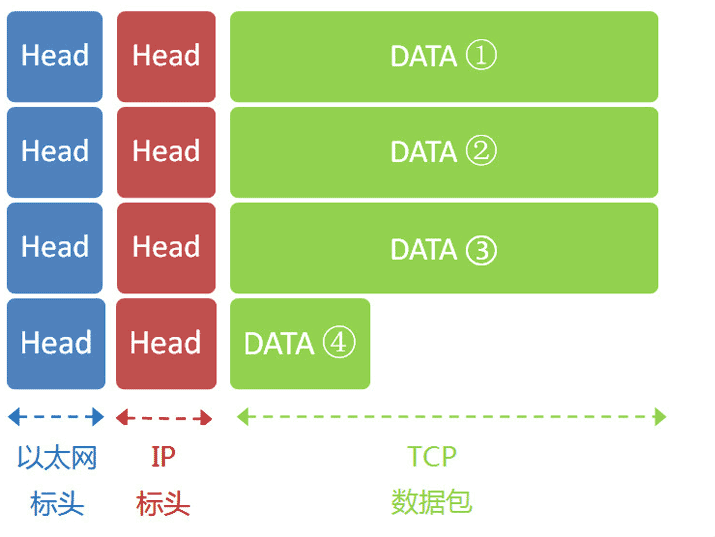
## IP协议

然后，TCP数据包再嵌入IP数据包。IP数据包需要设置双方的IP地址，这是已知的，发送方是192.168.1.100（本机），接收方是172.194.72.105（Google）。

IP数据包的标头长度为20字节，加上嵌入的TCP数据包，总长度变为5000字节。

## 以太网协议

最后，IP数据包嵌入以太网数据包。以太网数据包需要设置双方的MAC地址，发送方为本机的网卡MAC地址，接收方为网关192.168.1.1的MAC地址（通过ARP协议得到）。

以太网数据包的数据部分，最大长度为1500字节，而现在的IP数据包长度为5000字节。因此，IP数据包必须分割成四个包。因为每个包都有自己的IP标头（20字节），所以四个包的IP数据包的长度分别为1500、1500、1500、560。

## 服务器端响应

经过多个网关的转发，Google的服务器172.194.72.105，收到了这四个以太网数据包。

根据IP标头的序号，Google将四个包拼起来，取出完整的TCP数据包，然后读出里面的"HTTP请求"，接着做出"HTTP响应"，再用TCP协议发回来。

本机收到HTTP响应以后，就可以将网页显示出来，完成一次网络通信。

这个例子就到此为止，虽然经过了简化，但它大致上反映了互联网协议的整个通信过程。