# 概述

## 分类

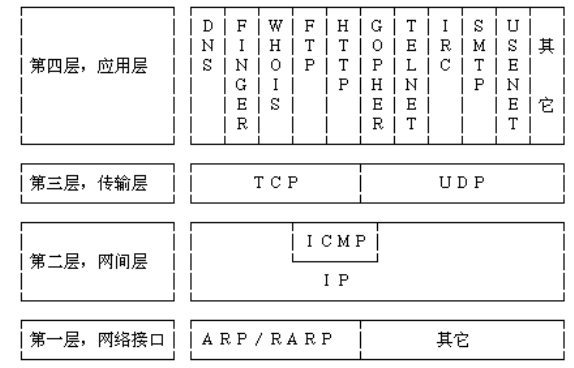
TCP/IP协议组大体上分为三部分：

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

## 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。

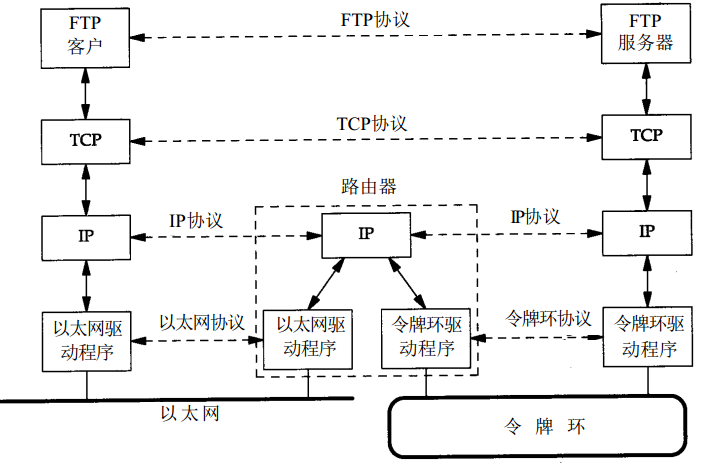


如上图所示，最底下的一层叫做"实体层"（Physical Layer），最上面的一层叫做"应用层"（Application Layer），中间的三层（自下而上）分别是"链接层"（Link Layer）、"网络层"（Network Layer）和"传输层"（Transport Layer）。越下面的层，越靠近硬件；越上面的层，越靠近用户。

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

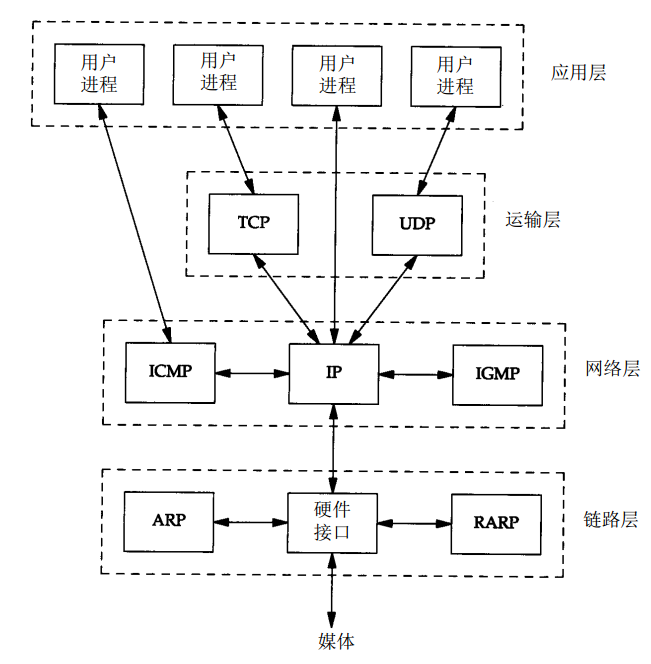




每一层都是为了完成一种功能。为了实现这些功能，就需要大家都遵守共同的规则。

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

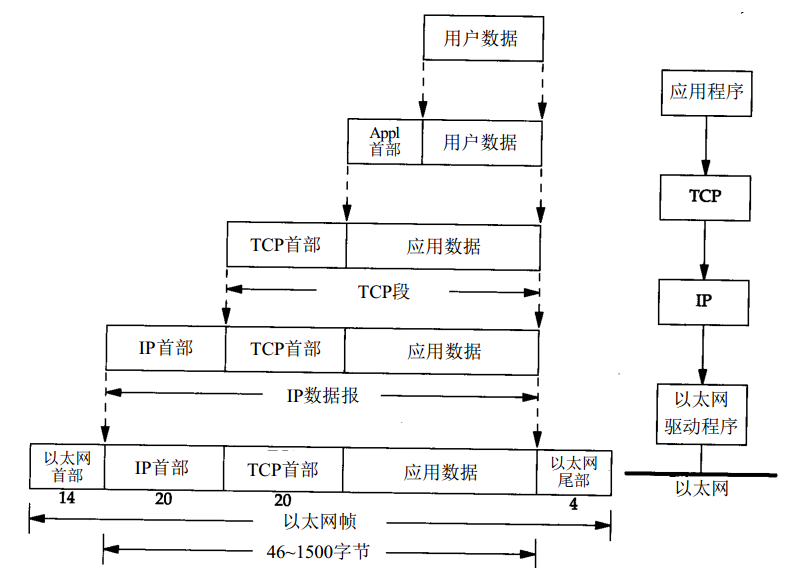
网络层协议：IP协议、ICMP协议、IGMP协议

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



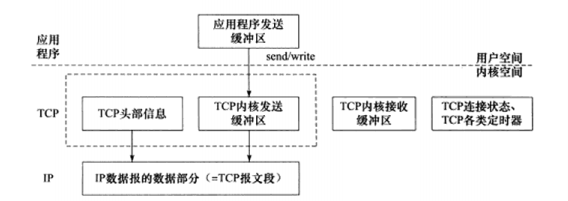
注：

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。

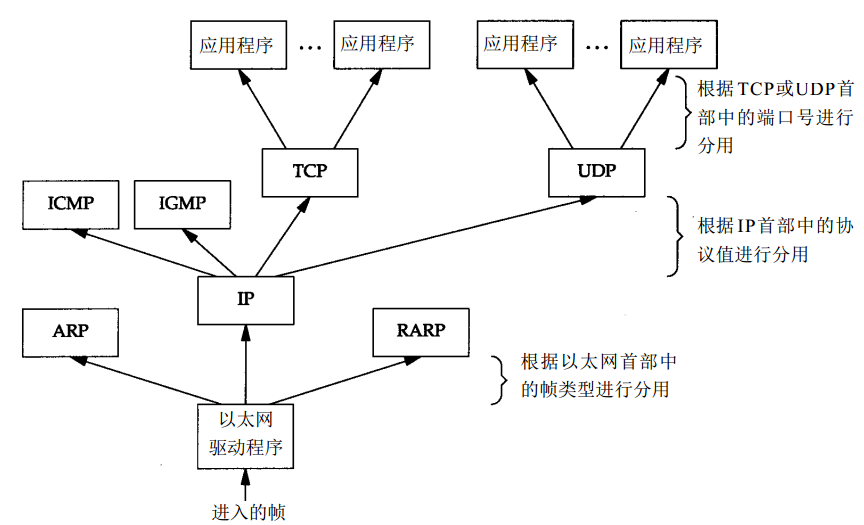
数据链路的传输媒介有多种，如果传输媒介不同，帧的类型也就不同。在以太网上传输的叫**以太网帧**，而令牌环网上的帧叫做**令牌环帧**。经过协议栈的层层封装，帧才是最终传输在物理媒介上的数据格式。

### 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。

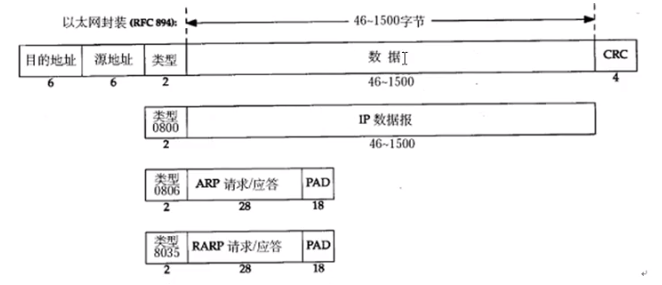
以太网规定，一组电信号构成一个数据包，叫做**"帧"（Frame）**。每一帧分成两个部分：**标头（Head）**和**数据（Data）**。

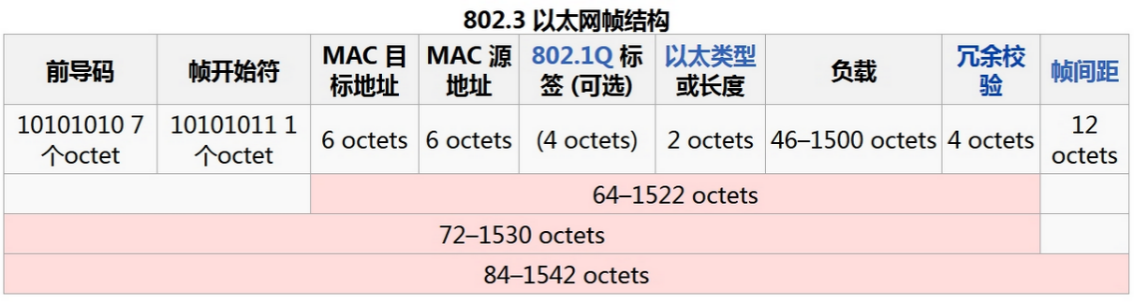


"标头"包含数据包的一些说明项，比如发送者、接受者、数据类型等等；"数据"则是数据包的具体内容。

"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。

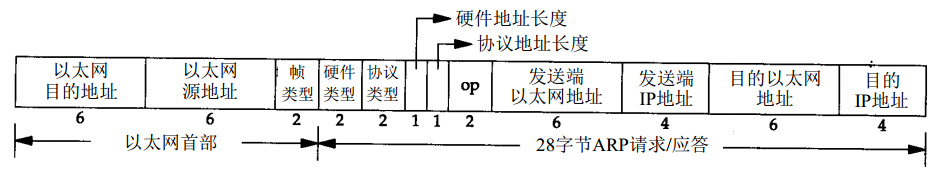
### 工作原理

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：



### 命令

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。

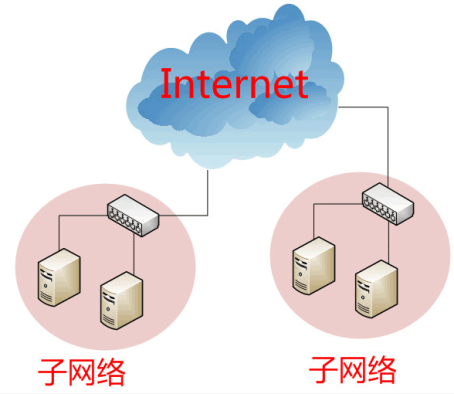
# 网络层

## 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。

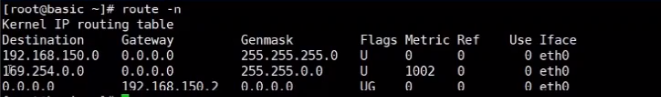


IP数据包的"标头"部分的长度为20到60字节，整个数据包的总长度最大为65,535字节。因此，理论上，一个IP数据包的"数据"部分，最长为65,515字节。前面说过，以太网数据包的"数据"部分，最长只有1500字节。因此，如果IP数据包超过了1500字节，它就需要分割成几个以太网数据包，分开发送了。

### 应用

#### route

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）

#### netstat

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

## 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）。

## 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**

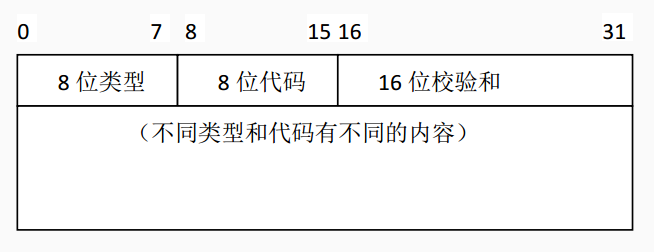
**私网-->公网：NAT映射**

**私网-->私网：NAT映射+打洞机制**

## ICMP协议

ICMP协议是Internet控制报文协议（Internet Control Message Protocol），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报文格式

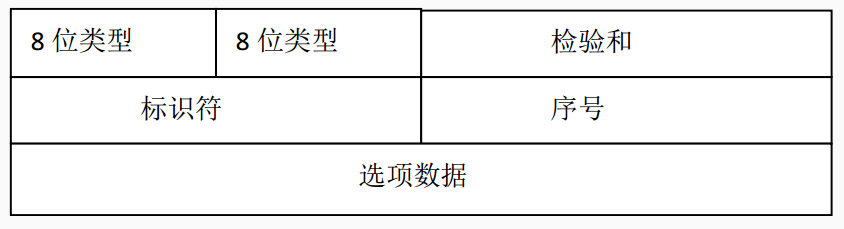


### ICMP查询报文

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

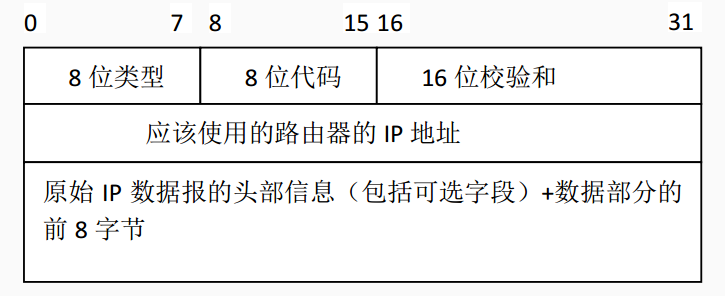
源站抑制

时间超过

参数问题

改变路由（重定向）

ICMP重定向报文格式：



### 应用

#### ping原理

#### traceroute原理

## 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。

# 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。

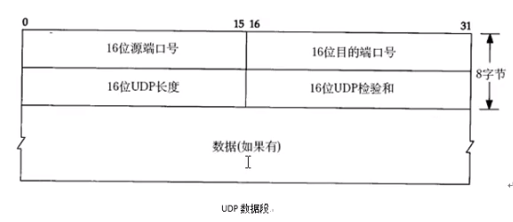
## UDP协议

### 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。

## 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。

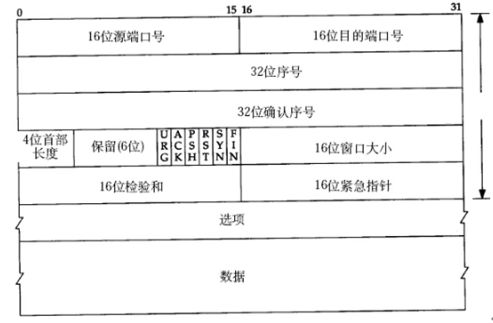
传输控制协议(TCP):

* 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：

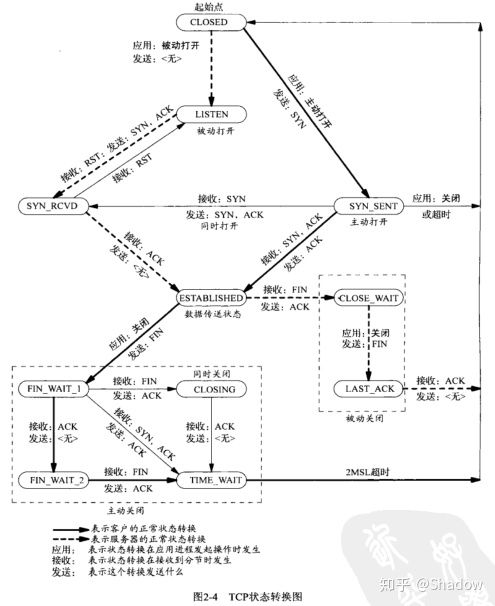
32位序号/32位确认序号：握手（以前发短信的回执消息即表示握手）

6个保留位：URG/ACK/PSH/SYN/FIN

16位窗口大小：65536

### TCP状态转移

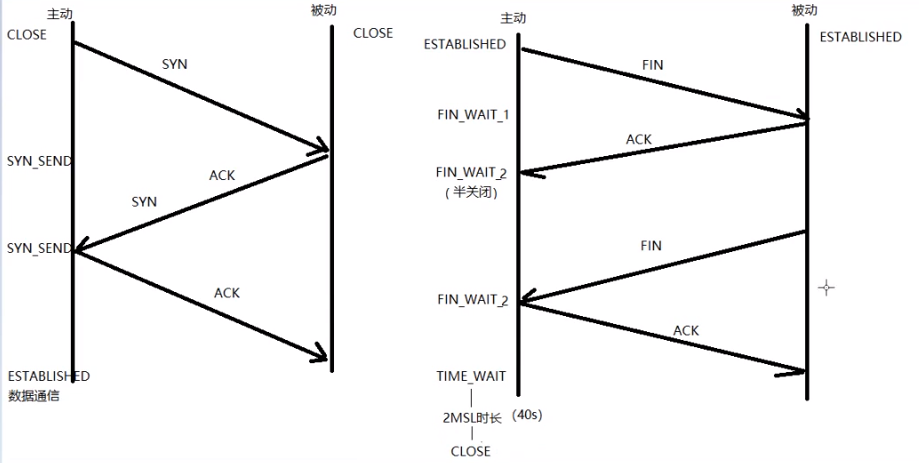
#### TCP状态转换图



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。

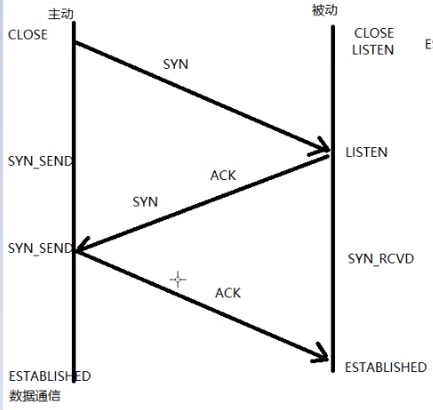
#### 主动关闭连接



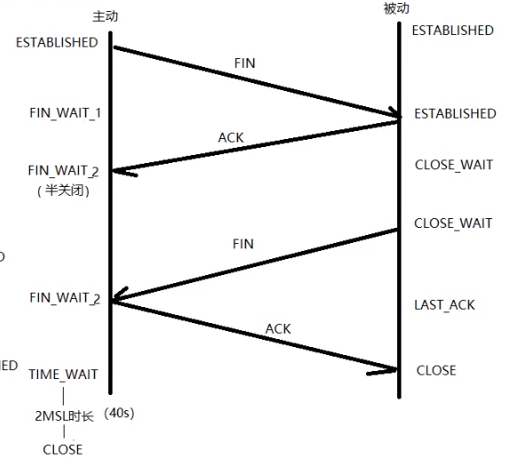
分析：

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



#### 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。

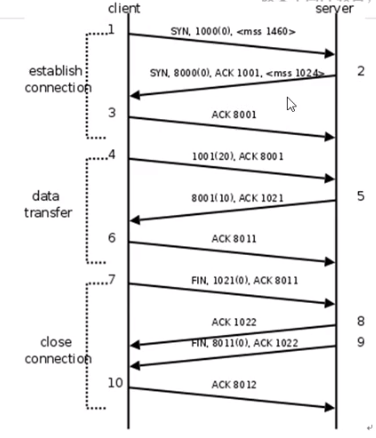
### TCP建立连接和关闭

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

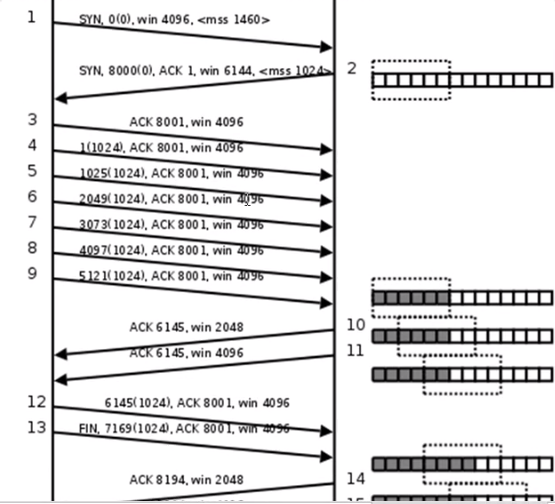
ACK8001

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：



##### 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

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）。

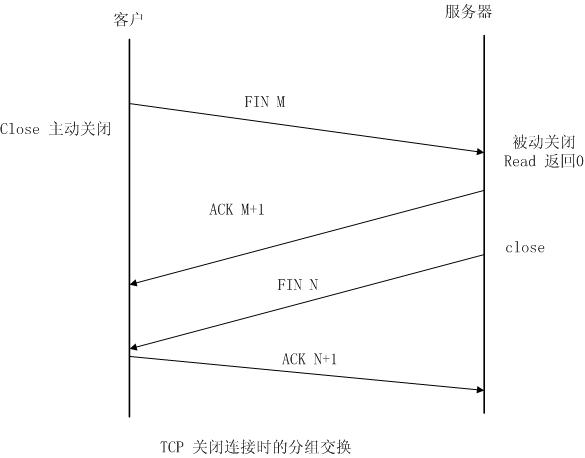
TIME\_WAIT状态存在的理由：

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。



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

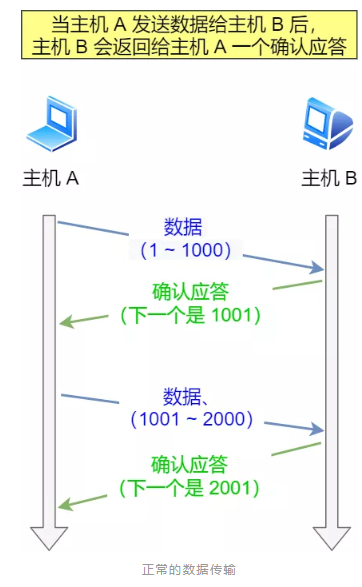
### TCP可靠传输机制



#### 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。

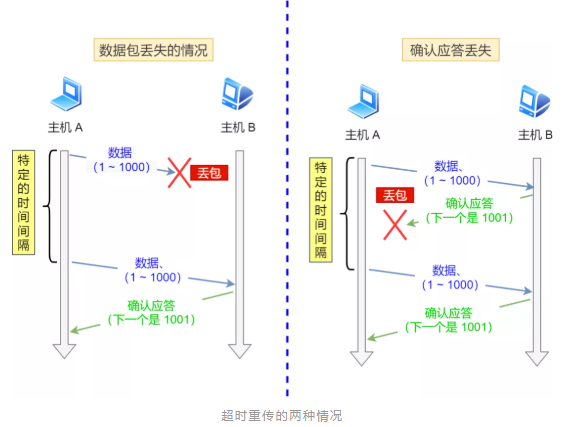
##### 超时重传

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。

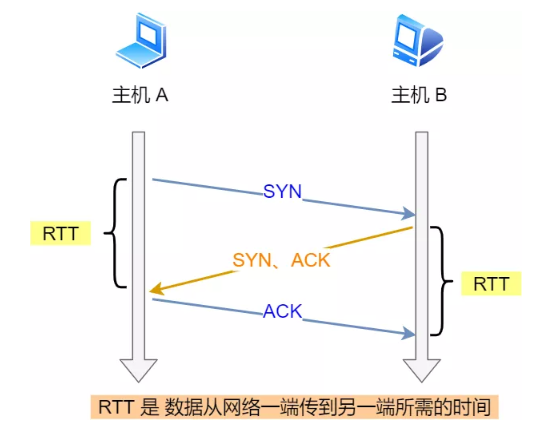
TCP会在以下两种情况发生超时重传：

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：

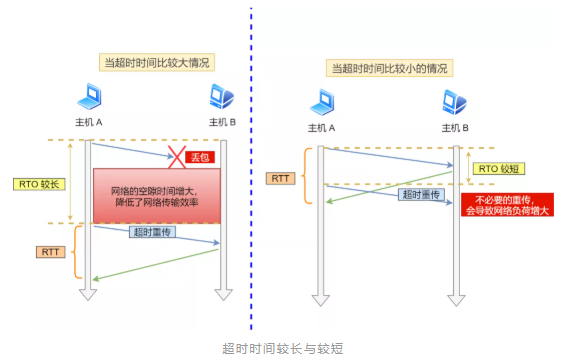


其中RTTd为偏差的加权平均值。

###### 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？



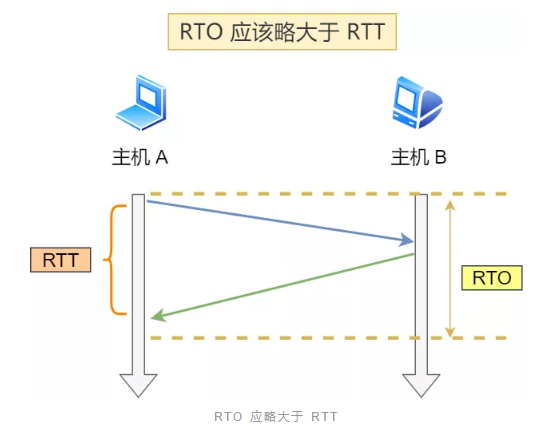
上图中有两种超时时间不同的情况：

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。

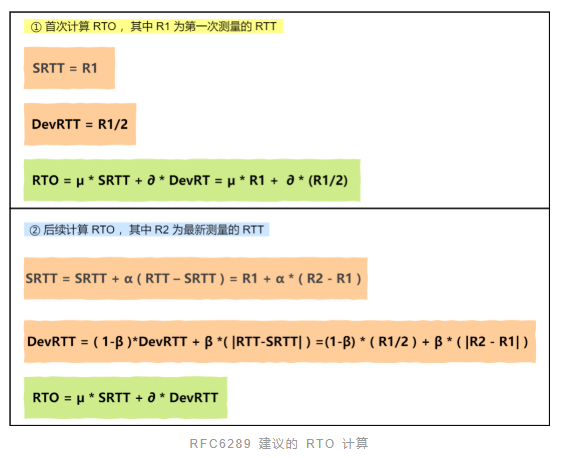
我们来看看 Linux 是如何计算 RTO 的呢？

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。

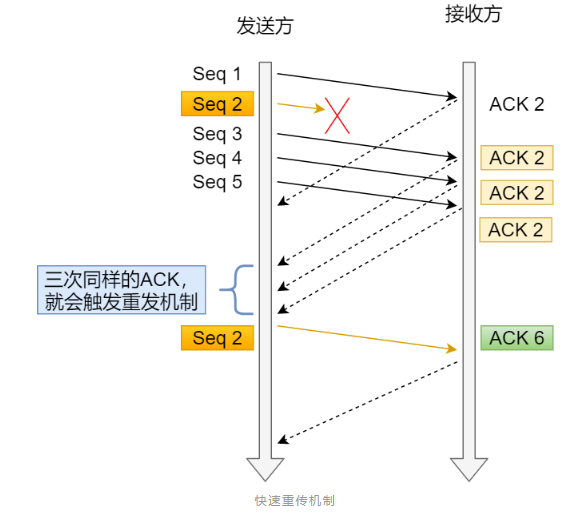
也就是每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。

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。

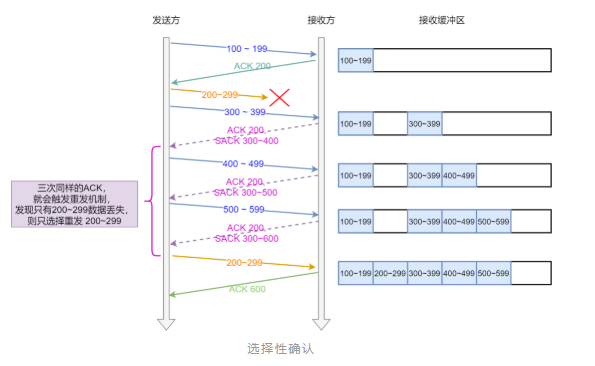
为了解决不知道该重传哪些TCP报文，于是就有SACK方法。

##### 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。

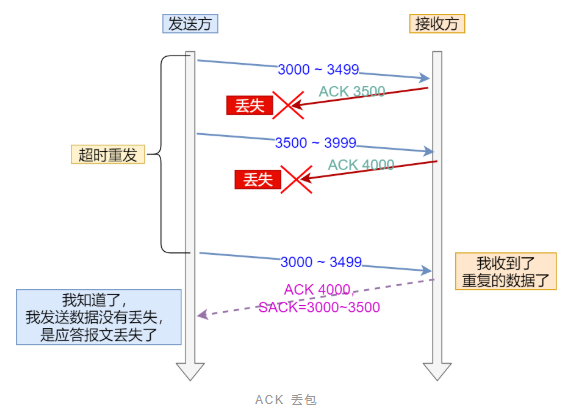


如果要支持SACK，必须双方都要支持。在Linux下，可以通过net.ipv4.tcp\_sack参数打开这个功能（Linux 2.4 后默认打开）。

##### 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**

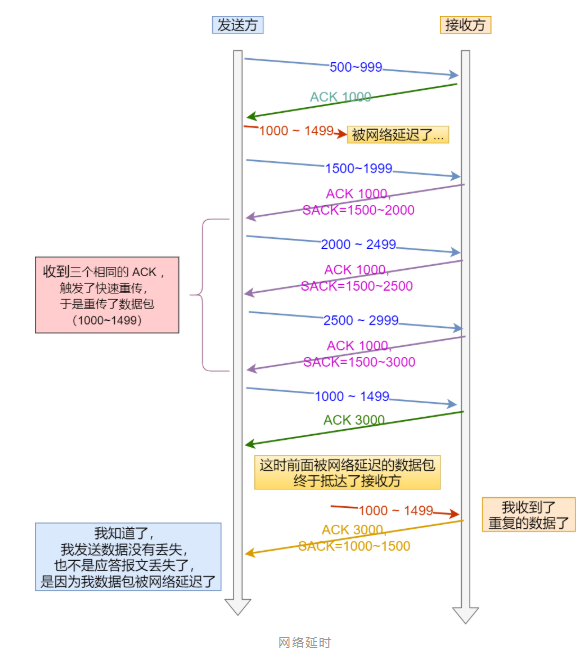


「接收方」发给「发送方」的两个 ACK 确认应答都丢失了，所以发送方超时后，重传第一个数据包（3000 ~ 3499）

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。

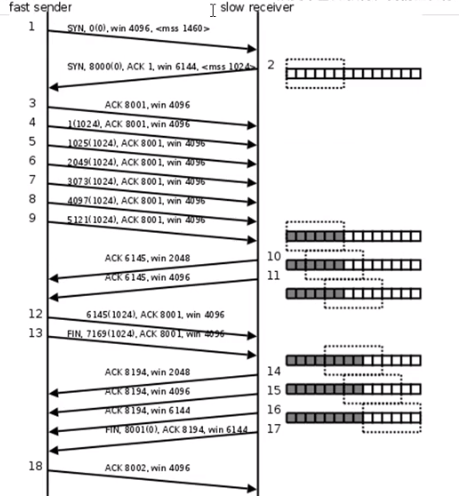
在 Linux 下可以通过 net.ipv4.tcp\_dsack 参数开启/关闭这个功能（Linux 2.4 后默认打开）。

#### TCP滑动窗口

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**。

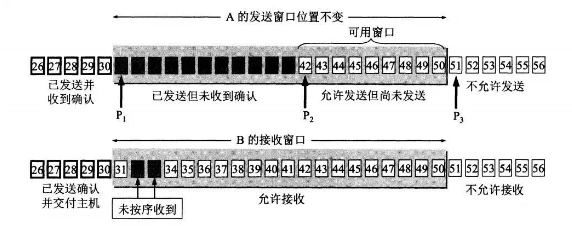
发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。如果发送窗口左部的字节已经发送并且收到了确认，那么就将发送窗口向右滑动一定距离，直到左部第一个字节不是已发送并且已确认的状态；接收窗口的滑动类似，接收窗口左部字节已经发送确认并交付主机，就向右滑动接收窗口。



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

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：

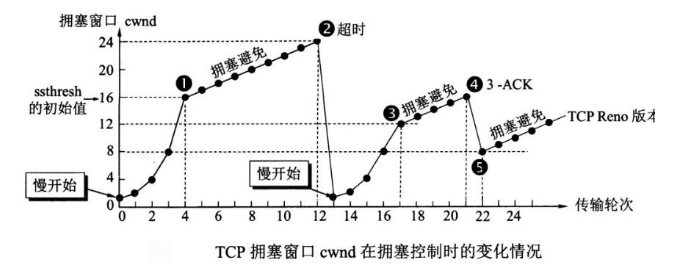
慢开始、拥塞避免、快重传、快恢复。

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。



**慢开始与拥塞避免**

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。

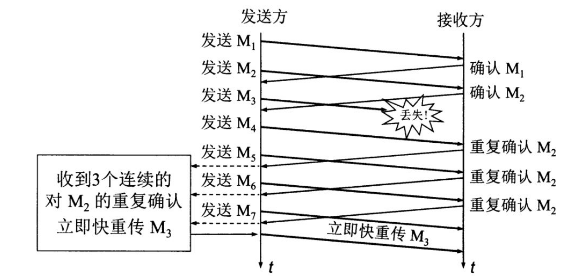
**快重传与快恢复**

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。



#### TCP粘包

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。

产生的原因：

TCP 协议是面向字节流的协议，它可能会组合或者拆分应用层协议的数据；

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；

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[^3]，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[^4]。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[^5]。

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。

# 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

## DNS协议

### 用途

DNS的作用：将机器的域名转换成IP地址。

### 工作原理

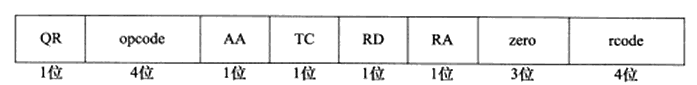
工作原理：对DNS的访问是通过一个地址解析器来完成的，解析器通过一个或多个名字服务器来完成这种相互转换。DNS是应用层的协议，在一个应用程序请求TCP打开一个连接或使用UDP发送一个数据之前，必须将一个主机名转换成一个IP地址。操作系统内核中的TCP/IP协议族对DNS一点都不知道。

### DNS查询和应答报文

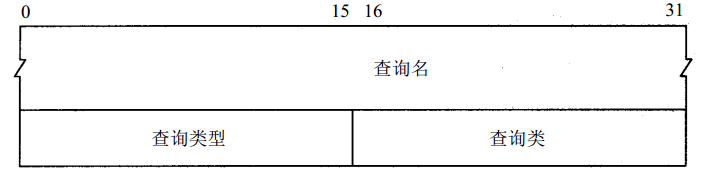
DNS查询/应答报文：



DNS报文头部的标志字段：



DNS查询问题的格式：



### Linux访问DNS服务

首先，查询DNS的服务器地址：

host –t –A [www.baidu.com](http://www.baidu.com)

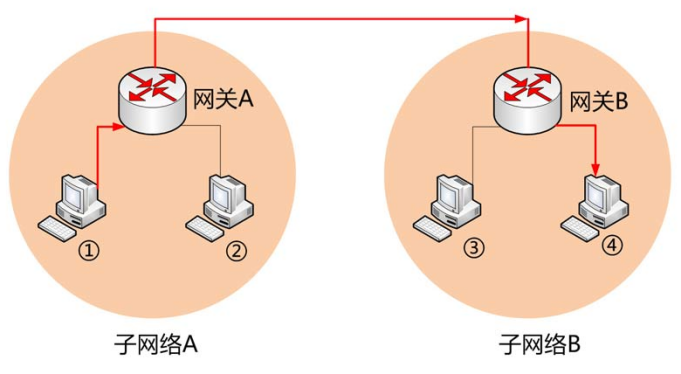
# 小结

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

## 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。