练识课堂 -- Go语言资深工程师培训课程

常见网络协议

1 协议的概念

1.1 什么是协议

从应用的角度出发，协议可理解为“**规则**”，是数据传输和数据解释的规则。

假设，A、B双方欲传输文件。规定：

- 第一次，传输文件名，接收方接收到文件名，应答OK给传输方；
- 第二次，发送文件的尺寸，接收方接收到该数据再次应答一个OK；
- 第三次，传输文件内容。同样，接收方接收数据完成后应答OK表示文件内容接收成功。

由此，无论A、B之间传递何种文件，都是通过三次数据传输来完成。A、B之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B之间达成的这个相互遵守的规则即为协议。

这种仅在A、B之间被遵守的协议称之为**原始协议**。当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个**标准协议**。最早的 `FTP` 协议就是由此衍生而来。

`TCP` 协议注重数据的传输。`HTTP` 协议着重于数据的解释。

1.2 典型协议

传输层 常见协议有TCP/UDP协议。

应用层 常见的协议有HTTP协议，FTP协议。

网络层 常见协议有IP协议、ICMP协议、IGMP协议。

网络接口层 常见协议有ARP协议、RARP协议。

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

UDP用户数据报协议（User Datagram Protocol）是OSI参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

HTTP超文本传输协议（Hyper Text Transfer Protocol）是互联网上应用最为广泛的一种网络协议。

FTP文件传输协议（File Transfer Protocol）

IP协议是因特网互联协议（Internet Protocol）

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。

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。

ARP协议是正向地址解析协议（Address Resolution Protocol），通过已知的IP，寻找对应主机的MAC地址。

RARP是反向地址转换协议，通过MAC地址确定IP地址。

1.3 分层模型

- **物理层**：

  主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输，到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。

- **数据链路层**：

  定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的115200、8、N、1

- **网络层**：

  在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

- **传输层**：

  定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。

- **会话层**：

  通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。

- **表示层**：

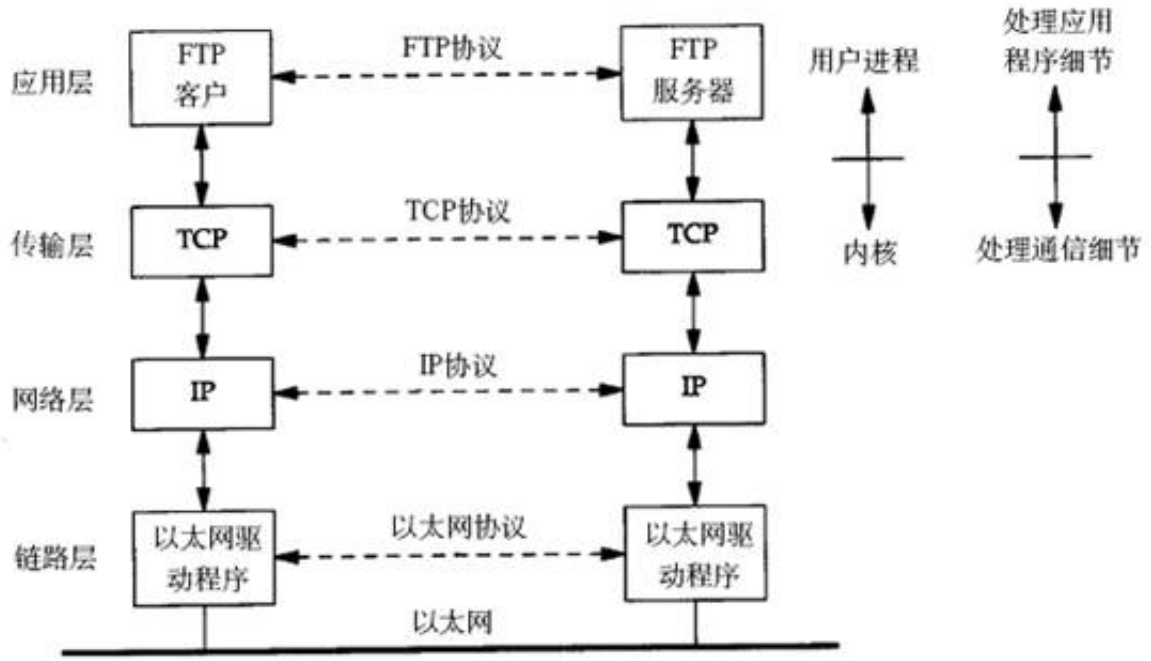
  可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。

- **应用层**：

  是最靠近用户的OSI层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

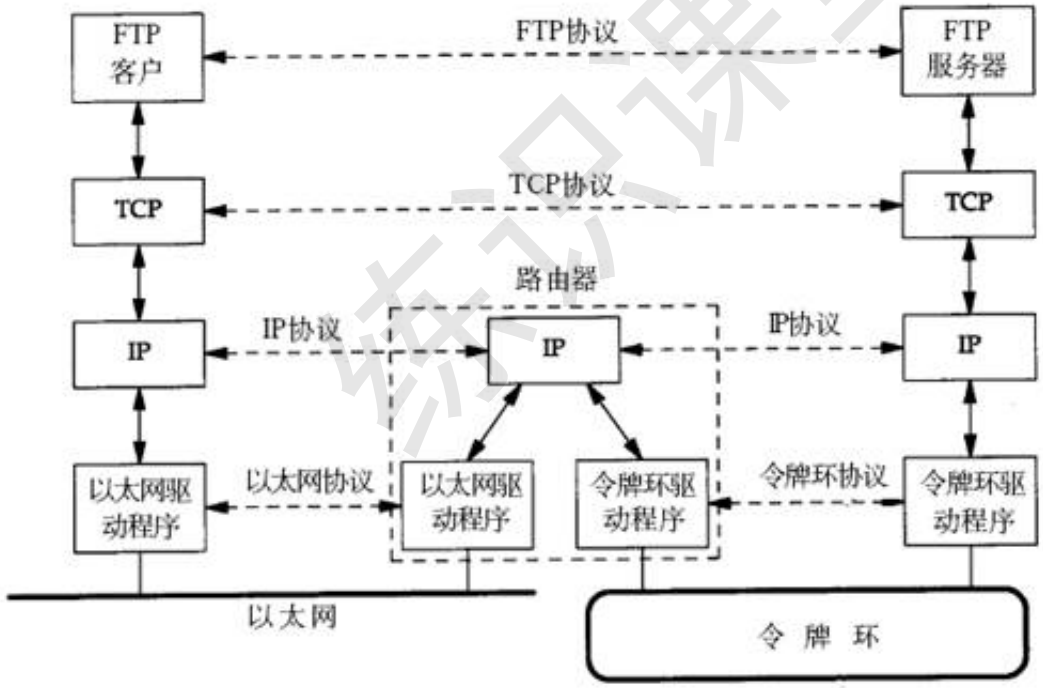
2 通信过程

两台计算机通过TCP/IP协议通讯的过程如下所示：



TCP/IP通信过程

上图对应两台计算机在同一网段中的情况，如果两台计算机在不同的网段中，那么数据从一台计算机到另一台计算机传输过程中要经过一个或多个路由器，如下图所示：



跨路由通信

链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（即从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。

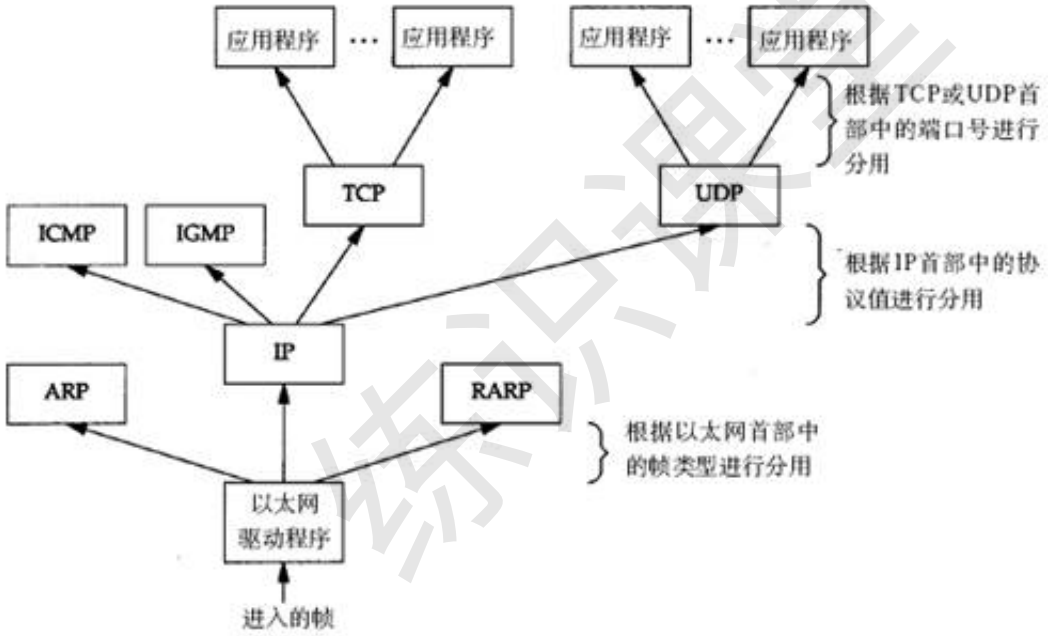
网络层的IP协议是构成Internet的基础。Internet上的主机通过IP地址来标识，Internet上有大量路由器负责根据IP地址选择合适的路径转发数据包，数据包从Internet上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

网络层负责点到点（ptop，point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（etoe，end-to-end）的传输（这里的“端”指源主机和目的主机）。传输层可选择TCP或UDP协议。

TCP是一种面向连接的、可靠的协议，有点像打电话，双方拿起电话互通身份之后就建立了连接，然后说话就行了，这边说的话那边保证听得到，并且是按说话的顺序听到的，说完话挂机断开连接。也就是说TCP传输的双方需要首先建立连接，之后由TCP协议保证数据收发的可靠性，丢失的数据包自动重发，上层应用程序收到的总是可靠的数据流，通讯之后关闭连接。

UDP是无连接的传输协议，不保证可靠性，有点像寄信，信写好放到邮筒里，既不能保证信件在邮递过程中不会丢失，也不能保证信件寄送顺序。使用UDP协议的应用程序需要自己完成丢包重发、消息排序等工作。

目的主机收到数据包后，如何经过各层协议栈最后到达应用程序呢？其过程如下图所示：



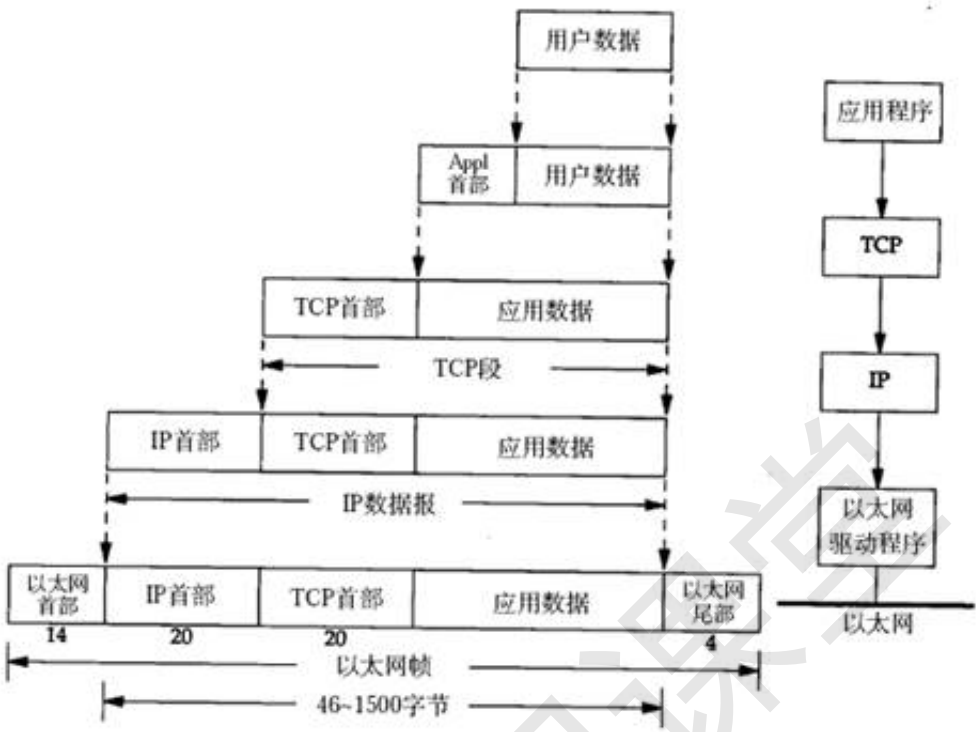
以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷（payload，指除去协议首部之外实际传输的数据）是IP、ARP还是RARP协议的数据报，然后交给相应的协议处理。假如是IP数据报，IP协议再根据IP首部中的“上层协议”字段确定该数据报的有效载荷是TCP、UDP、ICMP还是IGMP，然后交给相应的协议处理。假如是TCP段或UDP段，TCP或UDP协议再根据TCP首部或UDP首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP地址是标识网络中不同主机的地址，而端口号就是同一台主机上标识不同进程的地址，IP地址和端口号合起来标识网络中唯一的进程。

虽然IP、ARP和RARP数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP和RARP属于链路层，IP属于网络层。虽然ICMP、IGMP、TCP、UDP的数据都需要IP协议来封装成数据报，但是从功能上划分，ICMP、IGMP与IP同属于网络层，TCP和UDP属于传输层。

3 协议格式 （了解）

3.1 数据包封装

传输层及其以下的机制由内核提供，应用层由用户进程提供（后面将介绍如何使用socket API编写应用程序），应用程序对通讯数据的含义进行解释，而传输层及其以下处理通讯的细节，将数据从一台计算机通过一定的路径发送到另一台计算机。**应用层数据通过协议栈发到网络上时，每层协议都要加上一个数据首部（header），称为封装**（Encapsulation），如下图所示：

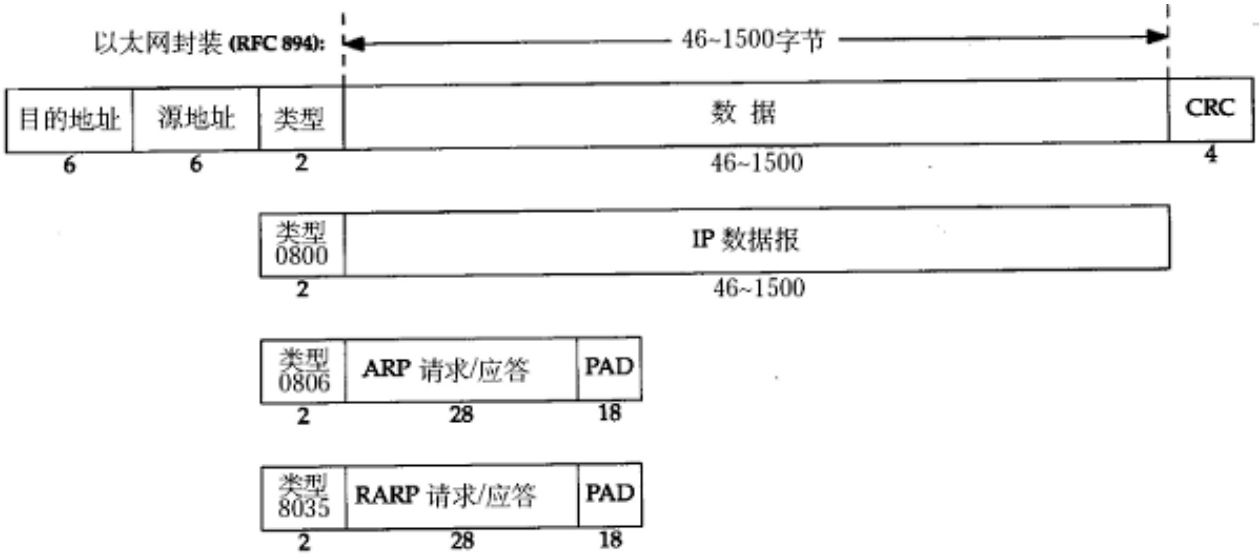


TCP/IP数据包封装

不同的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。

3.2 以太网帧格式

以太网的帧格式如下所示：



以太网帧格式

其中的源地址和目的地址是指网卡的硬件地址（也叫MAC地址），长度是48位，是在网卡出厂时固化的。可在shell中使用ifconfig命令查看，“HWaddr 00:15:F2:14:9E:3F”部分就是硬件地址。协议字段有三种值，分别对应IP、ARP、RARP。帧尾是CRC校验码。

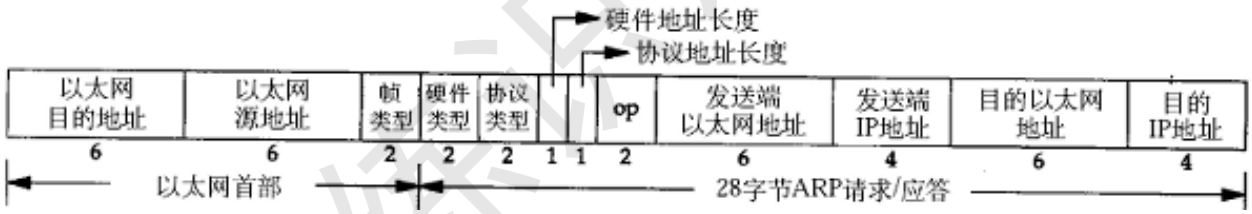
以太网帧中的数据长度规定最小46字节，最大1500字节，ARP和RARP数据包的长度不够46字节，要在后面补填充位。最大值1500称为以太网的最大传输单元（MTU），不同的网络类型有不同的MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的MTU，则需要对数据包进行分片（fragmentation）。ifconfig命令输出中也有“MTU:1500”。注意，MTU这个概念指数据帧中有效载荷的最大长度，不包括帧头长度。

3.3 ARP数据报格式

在网络通讯时，源主机的应用程序知道目的主机的IP地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP协议就起到这个作用。源主机发出ARP请求，询问“IP地址是192.168.0.1的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填FF:FF:FF:FF:FF:FF表示广播），目的主机接收到广播的ARP请求，发现其中的IP地址与本机相符，则发送一个ARP应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个ARP缓存表，可以用 `arp -a` 命令查看。缓存表中的表项有过期时间（一般为20分钟），如果20分钟内没有再次使用某个表项，则该表项失效，下次还要发ARP请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效?

ARP数据报的格式如下所示：



ARP数据报格式

源MAC地址、目的MAC地址在以太网首部和ARP请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1为以太网，协议类型指要转换的地址类型，0x0800为IP地址，后面两个地址长度对于以太网地址和IP地址分别为6和4（字节），op字段为1表示ARP请求，op字段为2表示ARP应答。

看一个具体的例子。

请求帧如下（为了清晰在每行的前面加了字节计数，每行16个字节）：

以太网首部（14字节）

```
0000: ff ff ff ff ff ff 00 05 5d 61 58 a8 08 06
```

ARP帧（28字节）

```
0000: 00 01

0010: 08 00 06 04 00 01 00 05 5d 61 58 a8 c0 a8 00 37

0020: 00 00 00 00 00 00 c0 a8 00 02
```

填充位（18字节）

```
0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00
```

以太网首部：目的主机采用广播地址，源主机的MAC地址是00:05:5d:61:58:a8，上层协议类型0x0806表示ARP。

ARP帧：硬件类型0x0001表示以太网，协议类型0x0800表示IP协议，硬件地址（MAC地址）长度为6，协议地址（IP地址）长度为4，op为0x0001表示请求目的主机的MAC地址，源主机MAC地址为00:05:5d:61:58:a8，源主机IP地址为c0 a8 00 37（192.168.0.55），目的主机MAC地址全0待填写，目的主机IP地址为c0 a8 00 02（192.168.0.2）。

由于以太网规定最小数据长度为46字节，ARP帧长度只有28字节，因此有18字节填充位，填充位的内容没有定义，与具体实现相关。

应答帧如下：

以太网首部

```
0000: 00 05 5d 61 58 a8 00 05 5d a1 b8 40 08 06
```

ARP帧

```
0000: 00 01

0010: 08 00 06 04 00 02 00 05 5d a1 b8 40 c0 a8 00 02

0020: 00 05 5d 61 58 a8 c0 a8 00 37
```

填充位

```
0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00
```

以太网首部：目的主机的MAC地址是00:05:5d:61:58:a8，源主机的MAC地址是00:05:5d:a1:b8:40，上层协议类型0x0806表示ARP。

ARP帧：硬件类型0x0001表示以太网，协议类型0x0800表示IP协议，硬件地址（MAC地址）长度为6，协议地址（IP地址）长度为4，op为0x0002表示应答，源主机MAC地址为00:05:5d:a1:b8:40，源主机IP地址为c0 a8 00 02（192.168.0.2），目的主机MAC地址为00:05:5d:61:58:a8，目的主机IP地址为c0 a8 00 37（192.168.0.55）。

3.4 IP段格式

IP数据报格式

IP数据报的首部长度和数据长度都是可变长的，但总是4字节的整数倍。

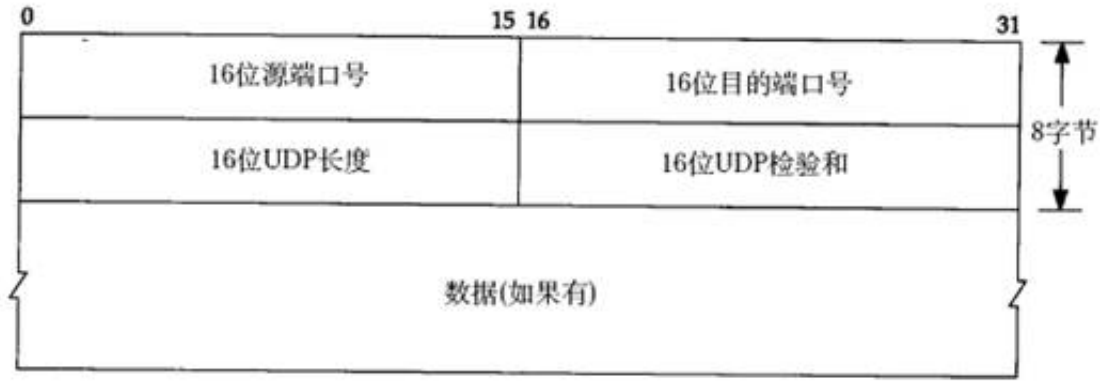
对于IPv4，4位版本字段是4。4位首部长度的数值是以4字节为单位的，最小值为5，也就是说首部长度最小是4x5=20字节，也就是不带任何选项的IP首部，4位能表示的最大值是15，也就是说首部长度最大是60字节。

8位TOS字段有3个位用来指定IP数据报的优先级（目前已经废弃不用），还有4个位表示可选的服务类型（最小延迟、最大吐量、最大可靠性、最小成本），还有一个位总是0。总长度是整个数据报（包括IP首部和IP层payload）的字节数。每传一个IP数据报，16位的标识加1，可用于分片和重新组装数据报。3位标志和13位片偏移用于分片。

TTL（Time to live)是这样用的：源主机为数据包设定一个生存时间，比如64，每过一个路由器就把该值减1，如果减到0就表示路由已经太长了仍然找不到目的主机的网络，就丢弃该包，因此这个生存时间的单位不是秒，而是跳（hop）。

协议字段指示上层协议是TCP、UDP、ICMP还是IGMP。然后是校验和，只校验IP首部，数据的校验由更高层协议负责。IPv4的IP地址长度为32位。

3.5 UDP数据报格式



UDP数据段

下面分析一帧基于UDP的TFTP协议帧。

以太网首部

```
0000: 00 05 5d 67 d0 b1 00 05 5d 61 58 a8 08 00
```

IP首部

```
0000: 45 00

0010: 00 53 93 25 00 00 80 11 25 ec c0 a8 00 37 c0 a8

0020: 00 01
```

UDP首部

```
0020:  05 d4 00 45 00 3f ac 40
```

TFTP协议

```
0020: 00 01 'c'':''\''q'

0030: 'w''e''r''q''.''q''w''e'00 'n''e''t''a''s''c''i'

0040: 'i'00 'b''l''k''s''i''z''e'00 '5''1''2'00 't''i'

0050: 'm''e''o''u''t'00 '1''0'00 't''s''i''z''e'00 '0'
```

以太网首部：源MAC地址是00:05:5d:61:58:a8，目的MAC地址是00:05:5d:67:d0:b1，上层协议类型0x0800表示IP。

IP首部：每一个字节0x45包含4位版本号和4位首部长度，版本号为4，即IPv4，首部长度为5，说明IP首部不带有选项字段。服务类型为0，没有使用服务。16位总长度字段（包括IP首部和IP层payload的长度）为0x0053，即83字节，加上以太网首部14字节可知整个帧长度是97字节。IP报标识是0x9325，标志字段和片偏移字段设置为0x0000，就是DF=0允许分片，MF=0此数据报没有更多分片，没有分片偏移。TTL是0x80，也就是128。上层协议0x11表示UDP协议。IP首部校验和为0x25ec，源主机IP是c0 a8

00 37（192.168.0.55），目的主机IP是c0 a8 00 01（192.168.0.1）。

UDP首部：源端口号0x05d4（1492）是客户端的端口号，目的端口号0x0045（69）是TFTP服务的well-known端口号。UDP报长度为0x003f，即63字节，包括UDP首部和UDP层pay-load的长度。UDP首部和UDP层payload的校验和为0xac40。

TFTP是基于文本的协议，各字段之间用字节0分隔，开头的00 01表示请求读取一个文件，接下来的各字段是：

```
c:\qwerq.qwe

netascii

blksize 512

timeout 10

tsize 0
```

一般的网络通信都是像TFTP协议这样，通信的双方分别是客户端和服务器，客户端主动发起请求（上面的例子就是客户端发起的请求帧），而服务器被动地等待、接收和应答请求。客户端的IP地址和端口号唯一标识了该主机上的TFTP客户端进程，服务器的IP地址和端口号唯一标识了该主机上的TFTP服务进程，由于客户端是主动发起请求的一方，它必须知道服务器的IP地址和TFTP服务进程的端口号，所以，一些常见的网络协议有默认的服务器端口，例如HTTP服务默认TCP协议的80端口，FTP服务默认TCP协议的21端口，TFTP服务默认UDP协议的69端口（如上例所示）。在使用客户端程序时，必须指定服务器的主机名或IP地址，如果不明确指定端口号则采用默认端口，请读者查阅ftp、tftp等程序的man page了解如何指定端口号。/etc/services中列出了所有well-known的服务端口和对应的传输层协议，这是由IANA（Internet Assigned Numbers Authority）规定的，其中有些服务既可以用TCP也可以用UDP，为了清晰，IANA规定这样的服务采用相同的TCP或UDP默认端口号，而另外一些TCP和UDP的相同端口号却对应不同的服务。

很多服务有well-known的端口号，然而客户端程序的端口号却不必是well-known的，往往是每次运行客户端程序时由系统自动分配一个空闲的端口号，用完就释放掉，称为ephemeral的端口号，想想这是为什么？

前面提过，UDP协议不面向连接，也不保证传输的可靠性，例如：

发送端的UDP协议层只管把应用层传来的数据封装成段交给IP协议层就算完成任务了，如果因为网络故障该段无法发到对方，UDP协议层也不会给应用层返回任何错误信息。

接收端的UDP协议层只管把收到的数据根据端口号交给相应的应用程序就算完成任务了，如果发送端发来多个数据包并且在网络上经过不同的路由，到达接收端时顺序已经错乱了，UDP协议层也不保证按发送时的顺序交给应用层。

通常接收端的UDP协议层将收到的数据放在一个固定大小的缓冲区中等待应用程序来提取和处理，如果应用程序提取和处理的速度很慢，而发送端发送的速度很快，就会丢失数据包，UDP协议层并不报告这种错误。

因此，使用UDP协议的应用程序必须考虑到这些可能的问题并实现适当的解决方案，例如等待应答、超时重发、为数据包编号、流量控制等。一般使用UDP协议的应用程序实现都比较简单，只是发送一些对可靠性要求不高的消息，而不发送大量的数据。例如，基于UDP的TFTP协议一般只用于传送小文件（所以才叫trivial的ftp），而基于TCP的FTP协议适用于 各种文件的传输。TCP协议又是如何用面向连接的服务来代替应用程序解决传输的可靠性问题呢。

3.6 TCP数据报格式

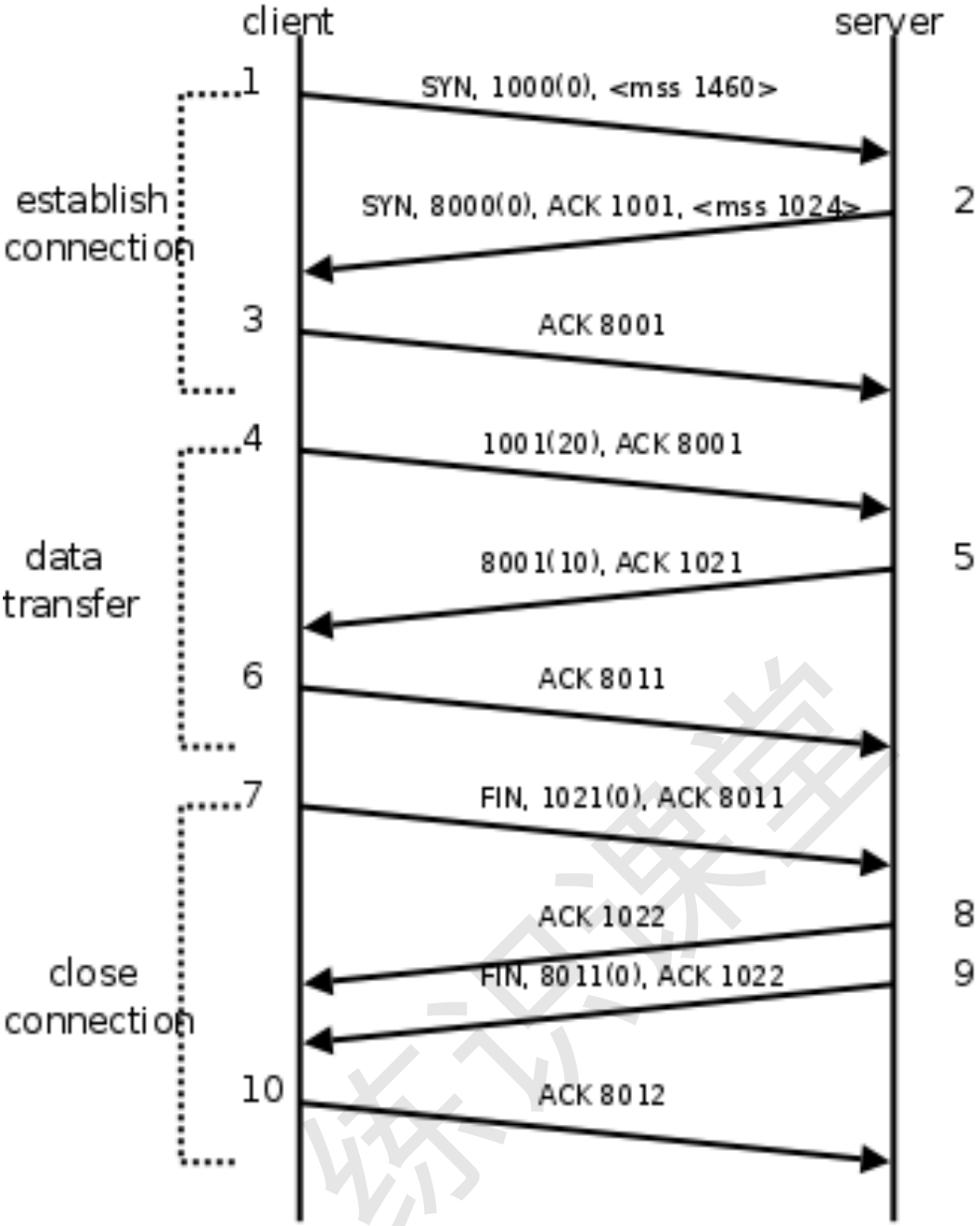
TCP数据段

与UDP协议一样也有源端口号和目的端口号，通讯的双方由IP地址和端口号标识。32位序号、32位确认序号、窗口大小稍后详细解释。4位首部长度和IP协议头类似，表示TCP协议头的长度，以4字节为单位，因此TCP协议头最长可以是4x15=60字节，如果没有选项字段，TCP协议头最短20字节。URG、ACK、PSH、RST、SYN、FIN是六个控制位，本节稍后将解释SYN、ACK、FIN、RST四个位，其它位的解释从略。16位检验和将TCP协议头和数据都计算在内。紧急指针和各种选项的解释从略。

4 TCP协议

4.1 TCP通信时序

下图是一次TCP通讯的时序图。TCP连接建立断开。包含大家熟知的三次握手和四次握手。



TCP通讯时序

在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序，注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。双方发送的段按时间顺序编号为1-10，各段中的主要信息在箭头上标出，例如段2的箭头上标着SYN, 8000(0), ACK1001, ，表示该段中的SYN位置1，32位序号是8000，该段不携带有效载荷（数据字节数为0），ACK位置1，32位确认序号是1001，带有一个**mss**（Maximum Segment Size，**最大报文长度**）选项值为1024。

建立连接（三次握手）的过程：

- 客户端发送一个带SYN标志的TCP报文到服务器。这是三次握手过程中的段1。

客户端发出段1，SYN位表示连接请求。序号是1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况，另外，规定SYN位和FIN位也要占一个序号，这次虽然没发数据，但是由于发了SYN位，因此下次再发送应该用序号1001。mss表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大帧长

度，就必须在IP层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

- 服务器端回应客户端，是三次握手中的第2个报文段，同时带ACK标志和SYN标志。它表示对刚才客户端SYN的回应；同时又发送SYN给客户端，询问客户端是否准备好进行数据通讯。

服务器发出段2，也带有SYN位，同时置ACK位表示确认，确认序号是1001，表示“我接收到序号1000及其以前所有的段，请你下次发送序号为1001的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为1024。

- 客户必须再次回应服务器端一个ACK报文，这是报文段3。

客户端发出段3，对服务器的连接请求进行应答，确认序号是8001。在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出，因此一共有三个段用于建立连接，称为“三方握手（three-way-handshake）”。在建立连接的同时，双方协商了一些信息，例如双方发送序号的初始值、最大段尺寸等。

在TCP通讯中，如果一方收到另一方发来的段，读出其中的目的端口号，发现本机并没有任何进程使用这个端口，就会应答一个包含RST位的段给另一方。例如，服务器并没有任何进程使用8080端口，我们却用telnet客户端去连接它，服务器收到客户端发来的SYN段就会应答一个RST段，客户端的telnet程序收到RST段后报告错误Connection refused：

```
$ telnet 192.168.0.200 8080

Trying 192.168.0.200...

telnet: Unable to connect to remote host: Connection refused
```

数据传输的过程：

- 客户端发出段4，包含从序号1001开始的20个字节数据。
- 服务器发出段5，确认序号为1021，对序号为1001-1020的数据表示确认收到，同时请求发送序号1021开始的数据，服务器在应答的同时也向客户端发送从序号8001开始的10个字节数据，这称为piggyback。
- 客户端发出段6，对服务器发来的序号为8001-8010的数据表示确认收到，请求发送序号8011开始的数据。

在数据传输过程中，ACK和确认序号是非常重要的，应用程序交给TCP协议发送的数据会暂存在TCP层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的ACK段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的ACK段，经过等待超时后TCP协议自动将发送缓冲区中的数据包重发。

关闭连接（四次握手）的过程：

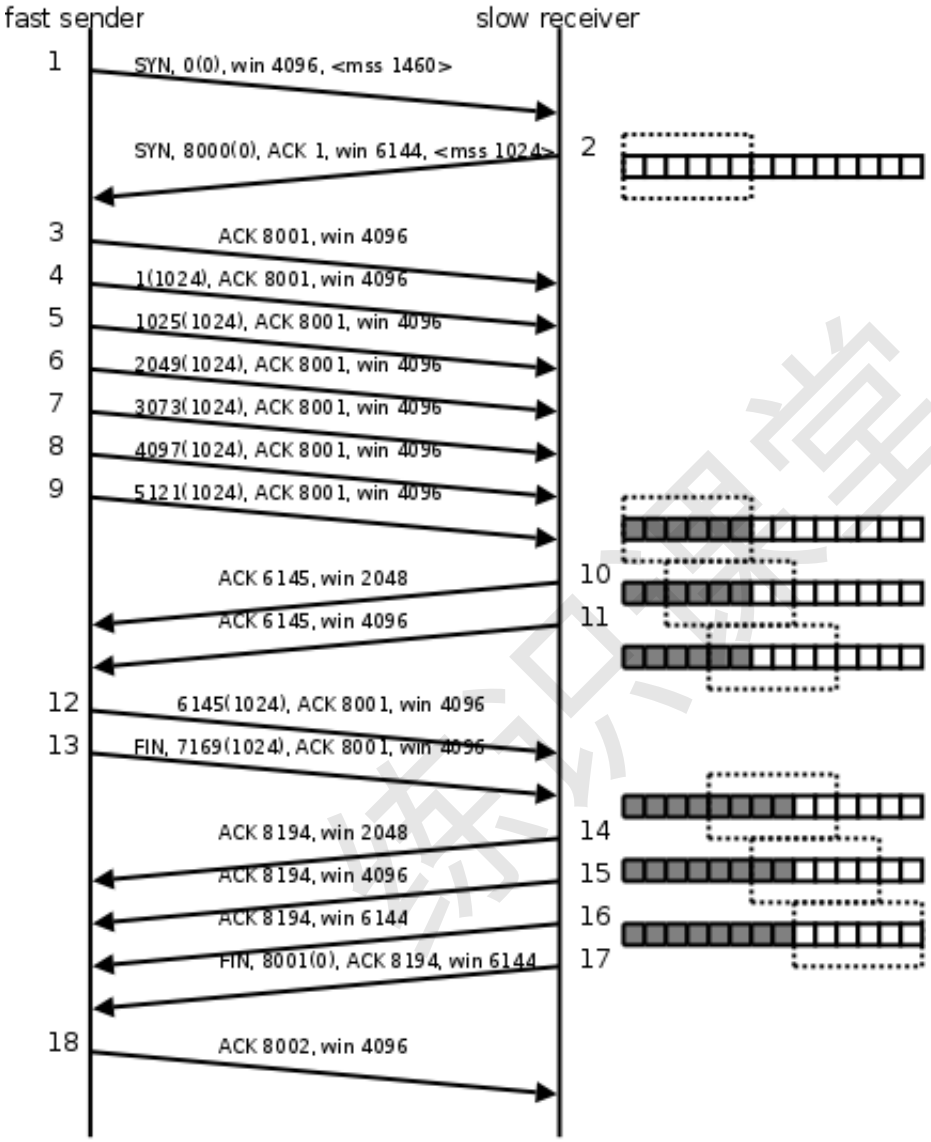
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

- 客户端发出段7，FIN位表示关闭连接的请求。
- 服务器发出段8，应答客户端的关闭连接请求。
- 服务器发出段9，其中也包含FIN位，向客户端发送关闭连接请求。
- 客户端发出段10，应答服务器的关闭连接请求。

建立连接的过程是三方握手，而关闭连接通常需要4个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以发送数据给客户端，直到服务器也关闭连接为止。

4.2 滑动窗口 (TCP流控制)

介绍UDP时描述了这样的问题：如果发送端发送的速度较快，接收端接收到数据后处理的速度较慢，而接收缓冲区的大小是固定的，就会丢失数据。TCP协议通过“滑动窗口（Sliding Window）”机制解决这一问题。看下图的通讯过程：



滑动窗口

- 发送端发起连接，声明最大段尺寸是1460，初始序号是0，窗口大小是4K，表示“我的接收缓冲区还有4K字节空闲，你发的数据不要超过4K”。接收端应答连接请求，声明最大段尺寸是1024，初始序号是8000，窗口大小是6K。发送端应答，三方握手结束。
- 发送端发出段4-9，每个段带1K的数据，发送端根据窗口大小知道接收端的缓冲区满了，因此停止发送数据。
- 接收端的应用程序提走2K数据，接收缓冲区又有了2K空闲，接收端发出段10，在应答已收到6K数据的同时声明窗口大小为2K。
- 接收端的应用程序又提走2K数据，接收缓冲区有4K空闲，接收端发出段11，重新声明窗口大小为4K。

- 发送端发出段12-13，每个段带2K数据，段13同时还包含FIN位。
- 接收端应答接收到的2K数据（6145-8192），再加上FIN位占一个序号8193，因此应答序号是8194，连接处于半关闭状态，接收端同时声明窗口大小为2K。
- 接收端的应用程序提走2K数据，接收端重新声明窗口大小为4K。
- 接收端的应用程序提走剩下的2K数据，接收缓冲区全空，接收端重新声明窗口大小为6K。
- 接收端的应用程序在提走全部数据后，决定关闭连接，发出段17包含FIN位，发送端应答，连接完全关闭。

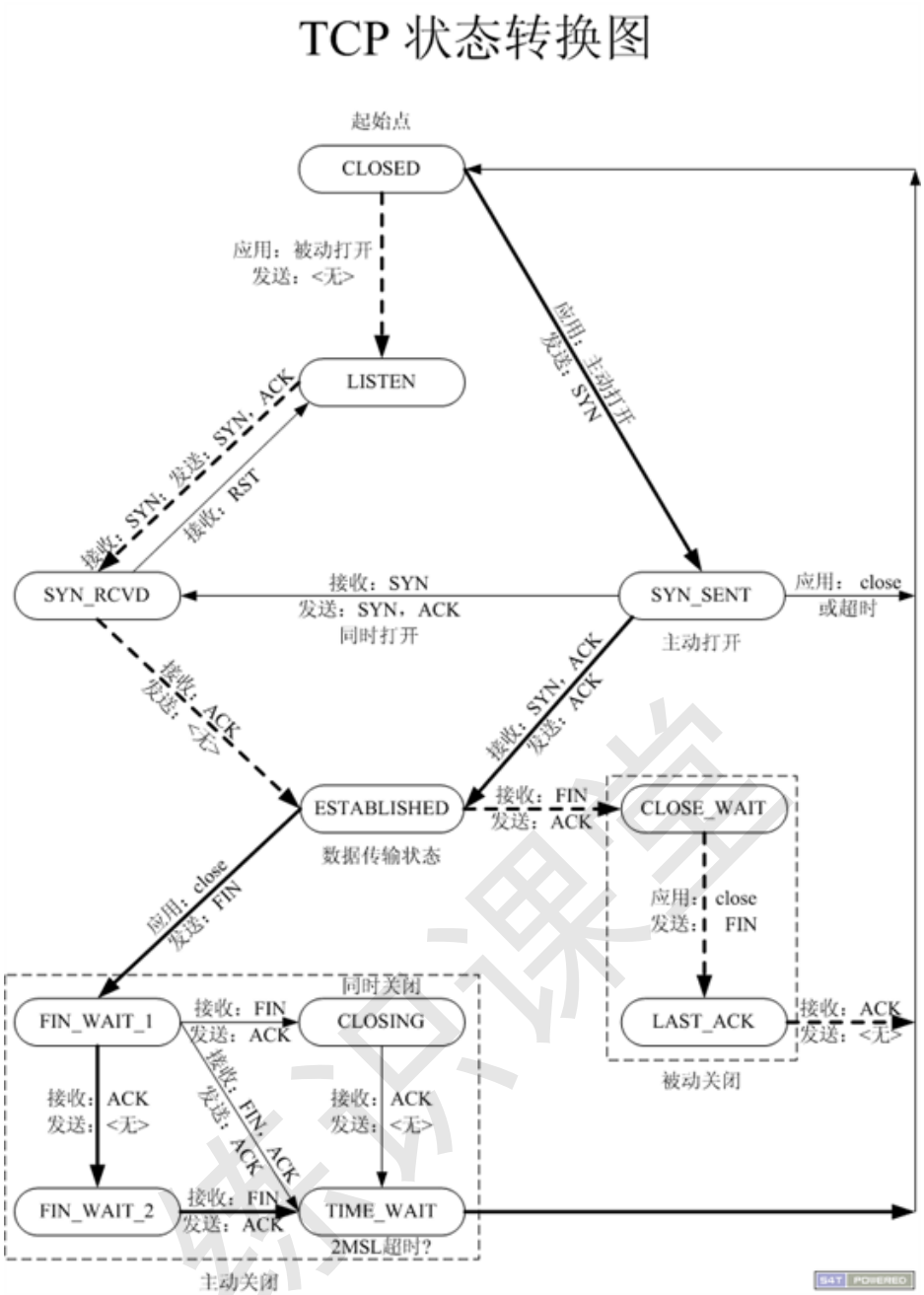
上图在接收端用小方块表示1K数据，实心的小方块表示已接收到的数据，虚线框表示接收缓冲区，因此套在虚线框中的空心小方块表示窗口大小，从图中可以看出，随着应用程序提走数据，虚线框是向右滑动的，因此称为滑动窗口。

从这个例子还可以看出，发送端是一K一K地发送数据，而接收端的应用程序可以两K两K地提走数据，当然也有可能一次提走3K或6K数据，或者一次只提走几个字节的数据。也就是说，应用程序所看到的数据是一个整体，或说是一个流（stream），在底层通讯中这些数据可能被拆成很多数据包来发送，但是一个数据包有多少字节对应用程序是不可见的，因此TCP协议是面向流的协议。而UDP是面向消息的协议，每个UDP段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据，这一点和TCP是很不同的。

推荐阅读：TCP滑动窗口机制深度剖析

4.3 TCP状态转换

这个图N多人都知道，它排除和定位网络或系统故障时大有帮助，但是怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻的认识，不能只停留在一知半解之中。下面对这张图的11种状态详细解析一下，以便加强记忆！不过在这之前，先回顾一下TCP建立连接的三次握手过程，以及 关闭连接的四次握手过程。



TCP状态转换图

CLOSED：表示初始状态。

LISTEN：该状态表示服务器端的某个SOCKET处于监听状态，可以接受连接。

SYN_SENT：这个状态与SYN_RCVD遥相呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，随即进入到了SYN_SENT状态，并等待服务端的发送三次握手中的第2个报文。SYN_SENT状态表示客户端已发送SYN报文。

SYN_RCVD: 该状态表示接收到SYN报文，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂。此种状态时，当收到客户端的ACK报文后，会进入到ESTABLISHED状态。

ESTABLISHED：表示连接已经建立。

FIN_WAIT_1: FIN_WAIT_1和FIN_WAIT_2状态的真正含义都是表示等待对方的FIN报文。区别是：

FIN_WAIT_1状态是当socket在ESTABLISHED状态时，想主动关闭连接，向对方发送了FIN报文，此时该socket进入到FIN_WAIT_1状态。

FIN_WAIT_2状态是当对方回应ACK后，该socket进入到FIN_WAIT_2状态，正常情况下，对方应马上回应ACK报文，所以FIN_WAIT_1状态一般较难见到，而FIN_WAIT_2状态可用netstat看到。

FIN_WAIT_2：主动关闭链接的一方，发出FIN收到ACK以后进入该状态。称之为半连接或半关闭状态。该状态下的socket只能接收数据，不能发。

TIME_WAIT: 表示收到了对方的FIN报文，并发送出了ACK报文，等2MSL后即可回到CLOSED可用状态。如果FIN_WAIT_1状态下，收到对方同时带 FIN标志和ACK标志的报文时，可以直接进入到TIME_WAIT状态，而无须经过FIN_WAIT_2状态。

CLOSING: 这种状态较特殊，属于一种较罕见的状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的 ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢? 如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

CLOSE_WAIT: 此种状态表示在等待关闭。当对方关闭一个SOCKET后发送FIN报文给自己，系统会回应一个ACK报文给对方，此时则进入到CLOSE_WAIT状态。接下来呢，察看是否还有数据发送给对方，如果没有可以 close这个SOCKET，发送FIN报文给对方，即关闭连接。所以在CLOSE_WAIT状态下，需要关闭连接。

LAST_ACK: 该状态是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，即可以进入到CLOSED可用状态。

4.4 半关闭

当TCP链接中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方处在半链接状态，此时A可以接收B发送的数据，但是A已不能再向B发送数据。

4.5 2MSL

2MSL (Maximum Segment Lifetime) TIME_WAIT状态的存在有两个理由：

（1）让4次握手关闭流程更加可靠；4次握手的最后一个ACK是是由主动关闭方发送出去的，若这个ACK丢失，被动关闭方会再次发一个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 uplicate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，在网络中消失；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，并先于它达到了目的地，因此它的命运也就注定被TCP协议栈抛弃。

另外一个概念叫做incarnation connection，指跟上次的socket pair一摸一样的新连接，叫做incarnation of previous connection。lost uplicate加上incarnation connection，则会对我们的传输造成致命的错误。

TCP是流式的，所有包到达的顺序是不一致的，依靠序列号由TCP协议栈做顺序的拼接；假设一个incarnation connection这时收到的seq=1000, 来了一个lost duplicate为seq=1000，len=1000, 则TCP认为这个lost duplicate合法，并存放入了receive buffer，导致传输出现错误。通过一个2MSL TIME_WAIT状态，确保所有的lost duplicate都会消失掉，避免对新连接造成错误。

该状态为什么设计在**主动关闭这一方**：

（1）发最后ACK的是主动关闭一方。

（2）只要有一方保持TIME_WAIT状态，就能起到避免incarnation connection在2MSL内的重新建立，不需要两方都有。

如何正确对待2MSL TIME_WAIT?

RFC要求`socketpair`在处于`TIME_WAIT`时，不能再起一个incarnation connection。但绝大部分TCP实现，强加了更为严格的限制。在2MSL等待期间，socket中使用的本地端口在默认情况下不能再被使用。

若A `10.234.5.5:1234`和B `10.55.55.60:6666`建立了连接，A主动关闭，那么在A端只要port为1234，无论对方的port和ip是什么，都不允许再起服务。这甚至比RFC限制更为严格，RFC仅仅是要求`socketpair`不一致，而实现当中只要这个port处于TIME_WAIT，就不允许起连接。这个限制对主动打开方来说是无所谓的，因为一般用的是临时端口；但对于被动打开方，一般是server，就悲剧了，因为server一般是熟知端口。比如http，一般端口是80，不可能允许这个服务在2MSL内不能起来。

解决方案是给服务器的socket设置SO_REUSEADDR选项，这样的话就算熟知端口处于TIME_WAIT状态，在这个端口上依旧可以将服务启动。当然，虽然有了SO_REUSEADDR选项，但`socktpair`这个限制依旧存在。比如上面的例子，A通过SO_REUSEADDR选项依旧在1234端口上起了监听，但这时我们若是从B通过6666端口去连它，TCP协议会告诉我们连接失败，原因为Address already in use.

RFC 793中规定MSL为2分钟，实际应用中常用的是30秒，1分钟和2分钟等。

RFC (Request For Comments)，是一系列以编号排定的文件。收集了有关因特网相关资讯，以及UNIX和因特网社群的软件文件。

推荐阅读：TCP漫谈之keepalive和time_wait

常见Web协议

对于TCP，HTTP，Socket这些名词，是面试常被问到的名词，我们需要有一定的了解

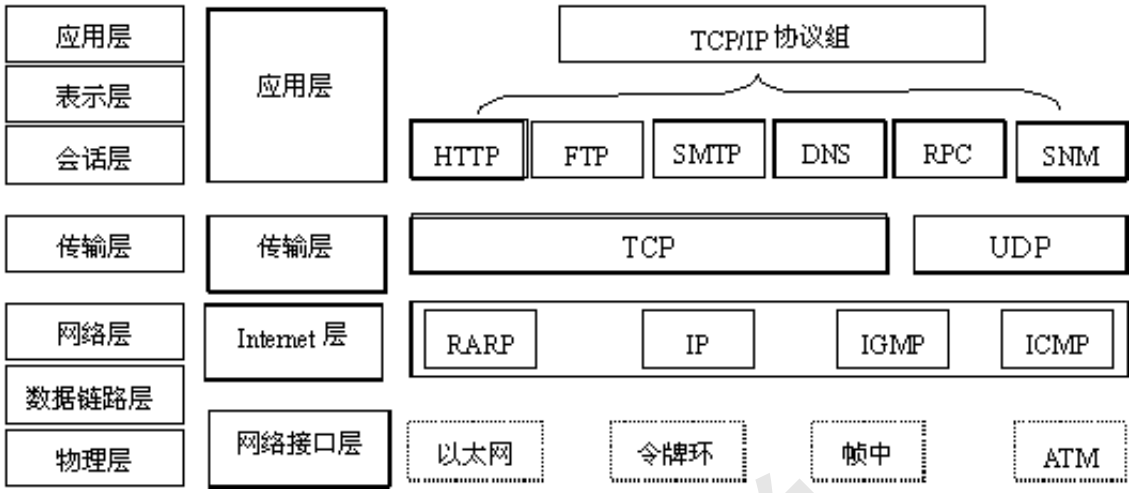
故事还要从七层网络协议开始...

七层网络协议

- **物理层** 建立、维护、断开物理连接。（由底层网络定义协议）
- **数据链路层** 建立逻辑连接、进行硬件地址寻址、差错校验等功能。（由底层网络定义协议）
- **网络层** 进行逻辑地址寻址，实现不同网络之间的路径选择。 常见协议有：ICMP IGMP IP（IPV4 IPV6） ARP RARP
- **传输层** 定义传输数据的协议端口号，以及流控和差错校验。 常见协议有：TCP UDP，数据包一旦离开网卡即进入网络传输层
- **会话层** 建立、管理、终止会话。（在五层模型里面已经合并到了应用层） 对应主机进程，指本地

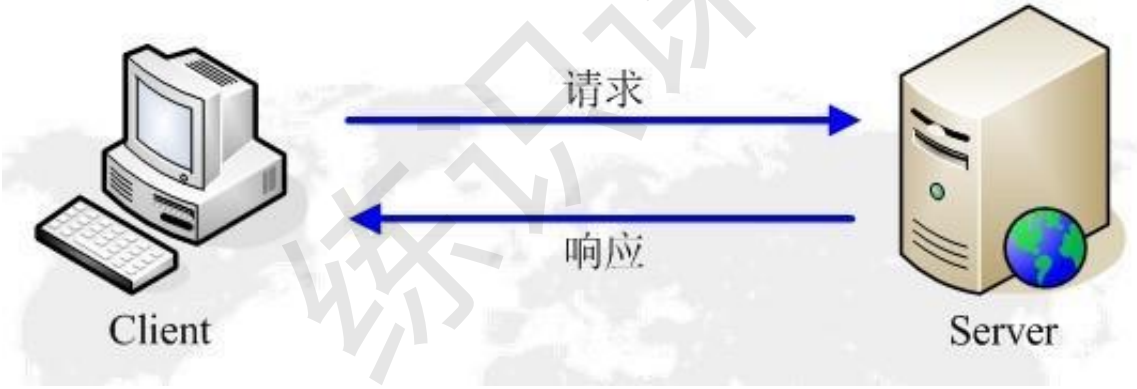
主机与远程主机正在进行的会话。

- **表示层** 数据的表示、安全、压缩。（在五层模型里面已经合并到了应用层） 常见格式有，JPEG、ASCll、DECOIC、加密格式等
- **应用层** 网络服务与最终用户的一个接口。 常见协议有：HTTP FTP TFTP SMTP SNMP DNS TELNET HTTPS POP3 DHCP



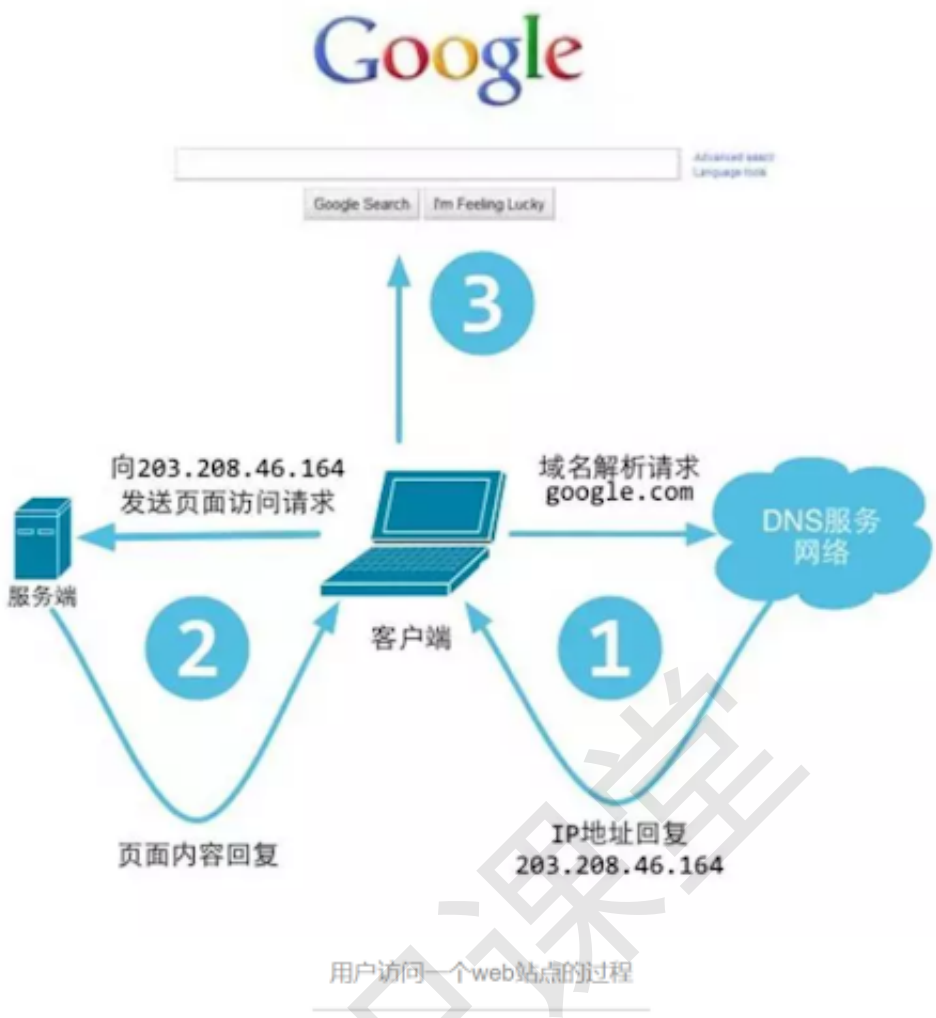
Web工作方式

HTTP协议工作于客户端-服务端架构为上。浏览器作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。



我们平时浏览网页的时候，会打开浏览器，输入网址后按下回车键，然后就会显示出你想要浏览的内容。在这个看似简单的用户行为背后，到底隐藏了些什么呢?

对于普通的上网过程，系统其实是这样做的：浏览器本身是一个客户端，当你输入URL的时候，首先浏览器会去请求DNS服务器，通过DNS获取相应的域名对应的IP，然后通过IP地址找到IP对应的服务器后，要求建立TCP连接，等浏览器发送完HTTP Request (请求)包后，服务器接收到请求包之后才开始处理请求包，服务器调用自身服务，返回HTTP Response (响应)包；客户端收到来自服务器的响应后开始渲染这个Response包里的主体(body)，等收到全部的内容随后断开与该服务器之间的TCP连接。



用户访问一个web站点的过程

一个Web服务器也被称为HTTP服务器，它通过HTTP协议与客户端通信。这个客户端通常指的是Web浏览器(其实手机端客户端内部也是浏览器实现的)。 Web服务器的工作原理可以简单地归纳为:

- 客户端通过TCP/IP协议建立到服务器的TCP连接
- 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档
- 服务器向客户端发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言 的解释引擎负责处理“ 动态内容”，并将处理得到的数据返回给客户端
- 客户端与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

一个简单的HTTP事务就是这样实现的，看起来很复杂，原理其实是挺简单的。需要注意的是客户端与服务器之间的通信是非持久连接的，也就是当服务器发送了应答后就与客户端断开连接，等待下一次请求。

> 第一次请求url，服务器返回的是html页面，然后浏览器开始渲染HTML。当解析到HTML DOM里面的图片连接，css脚本和js脚本的连接，浏览器会自动发起一个请求静态资源的HTTP请求，获取相应静态资源，然后浏览器会渲染出来，最终将所有资源整合、渲染、完整展现在屏幕上。(网页优化有一向措施是减少HTTP请求次数，把尽量多的css和js资源合并在一起)。

URL和DNS解析

URL

我们浏览网页都是通过URL访问的，那么URL到底是怎么样的呢? URL (Uni form Resource Locator)是“统一资源定位符” 的英文缩写，用于描述一个网络 上的资源，基本格式如下

`scheme://host[:port#]/path/.../[?query-string][#anchor]`

- scheme

  指定低层使用的协议(例如: http, https, ftp)

- host

  HTTP服务器的I P地址或者域名

- port#

  HTTP服务器的默认端口是80,这种情况下端口号可以省略。如果使用了别的端口,必须指明，例

- path

  访问资源的路径

- query-string

  发送给http服务器的数据

- anchor

  锚

举个例子：

让我们来解析一下下面这一段：

`http://mail.163.com/index.html`

1、http://:这个是协议，也就是HTTP超文本传输协议，也就是网页在网上传输的协议。

2、mail：这个是服务器名，代表着是一个邮箱服务器，所以是mail.

3、163.com:这个是域名，是用来定位网站的独一无二的名字。

4、mail.163.com：这个是网站名，由服务器名+域名组成。

5、/：这个是根目录，也就是说，通过网站名找到服务器，然后在服务器存放网页的根目录

6、index.html：这个是根目录下的默认网页（当然，163的默认网页是不是这个我不知道，只是大部分的默认网页，都是index.html）

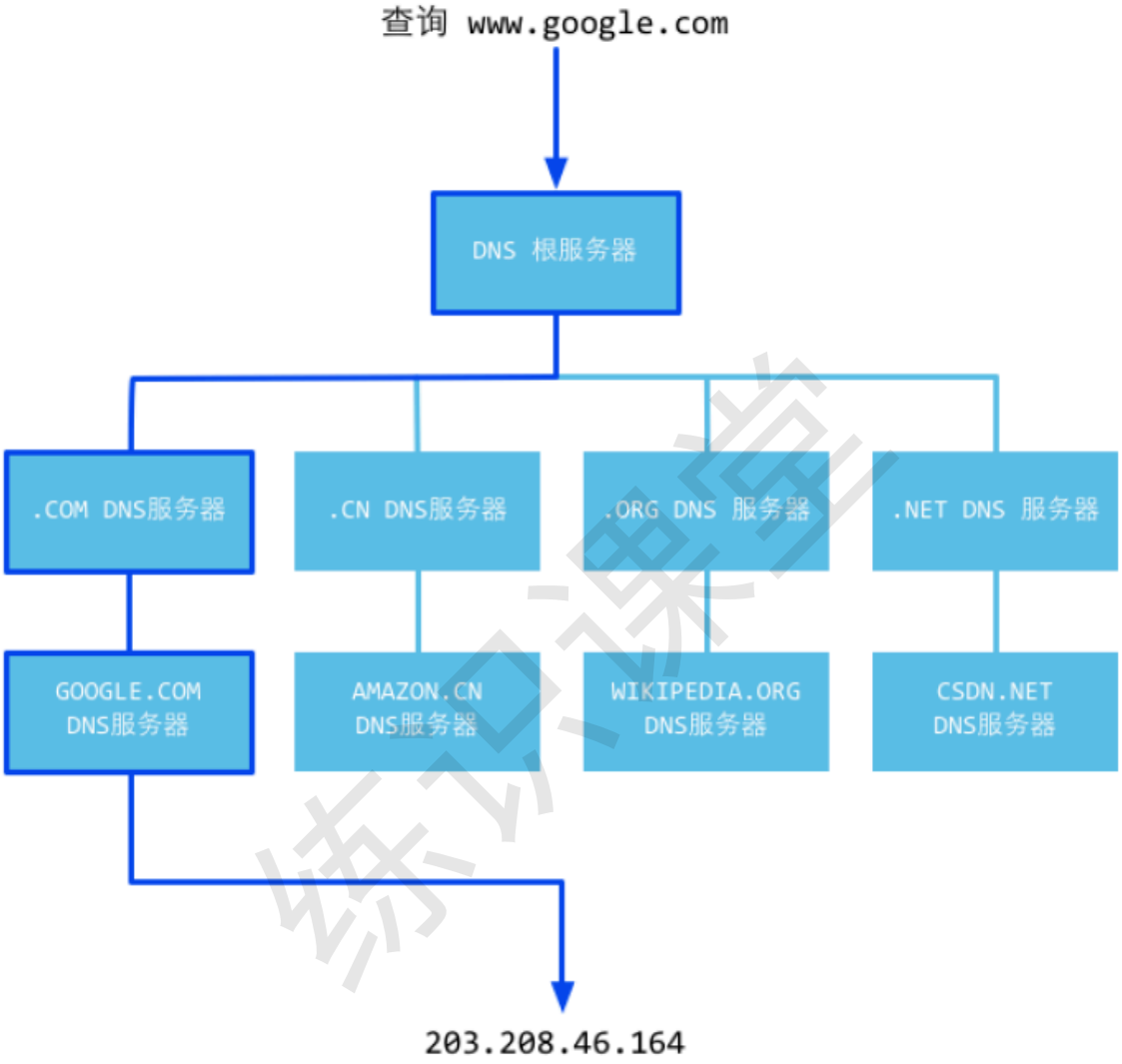
7、http://mail.163.com/index.html:这个叫做URL，统一资源定位符，全球性地址，用于定位网上的资源。

> URI：uniform resource identifier， 统一资源标识符，用来唯一的标识一个资源。
>
> URL：uniform resource locator， 统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。
>
> URN：uniform resource name， 统一资源命名，是通过名字来标识资源，比如mailto:lianshi@lianshiclass.com。

> 也就是说，URI是以一种抽象的，高层次概念定义统一资源标识，而URL和URN则是具体的资源标识的方式。URL和URN都是一种URI。

DNS

DNS (Domain Name System)是 “城名系统”的英文缩写，是一种组织成域层次结构的计算机和网络服务命名系统，它用于TCP/IP网络，它从事将主机名或域名转换为实际IP地址的工作。DNS就是这样的一位“翻译官”，它的基本工作原理可用下图来表示。



DNS解析过程

1. 浏览器中输入域名，操作系统会先检查自己本地的hosts文件是否有这个网络映射关系，如果有，就先调用这个IP地址映射，完成域名解析。
2. 如果hosts没有域名，查找本地DNS解析器缓存，如果有直接返回，完成域名解析。
3. 如果还没找到，会查找TCP/IP参数中设置的首选DNS服务器，我们叫它本地DNS服务器，此服务收到查询时，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。
4. 如果要查询的域名，不由本地DNS服务器区域解析，但该服务已经缓存了地址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。
5. 如果上述过程失败，则根据本地DNS服务器的设置进行查询，如果未用转发模式，则把请求发给根服务器，根服务器返回一个负责该顶级服务器的IP，本地DNS服务器收到IP信息后，再连接该IP上的服务器进行解析，如果仍然无法解析，则发送下一级DNS服务器，重复操作，直到找到。
6. 如果是转发模式则把请求转发至上一级DNS服务器，假如仍然不能解析，再转发给上上级。不管是

否转发，最后都把结果返回给本地DNS服务器上述一个是迭代查询，一个是递归查询。递归查询的过程是查询者发生了更替，而迭代查询过程，查询者不变。

通过上面的步骤，我们最后获取的是IP地址，也就是浏览器最后发起请求的时候是基于IP来和服务器做信息交互的。

举个例子来说，你想知道某个一起上法律课的女孩的电话，并且你偷偷拍了她的照片，回到寝室告诉一个很仗义的哥们儿，这个哥们儿二话没说，拍着胸脯告诉你，甭急，我替你查(此处完成了-次递归查询，即，问询者的角色更替)。然后他拿着照片问了学院大四学长，学长告诉他，这姑娘是xx系的；然后这哥们儿马不停蹄又问了xx系的办公室主任助理同学，助理同学说是xx系yy班的，然后很仗义的哥们儿去xx系yy班的班长那里取到了该女孩儿电话。(此处完成若干次迭代查询，即，问询者角色不变，但反复更替问询对象)最后，他把号码交到了你手里。完成整个查询过程。

HTTP协议

1 什么是HTTP协议

HTTP协议是Web工作的核心，所以要了解清楚Web的工作方式就需要详细的了解清楚HTTP是怎么样工作的。

HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写是一个基于TCP/IP通信协议来传递数据，服务器传输超文本到本地浏览器的传送协议，建立在TCP协议之上，一般采用80端口。HTTP协议工作于客户端-服务端架构上。浏览器可作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。它是一个无状态的请求/响应协议。

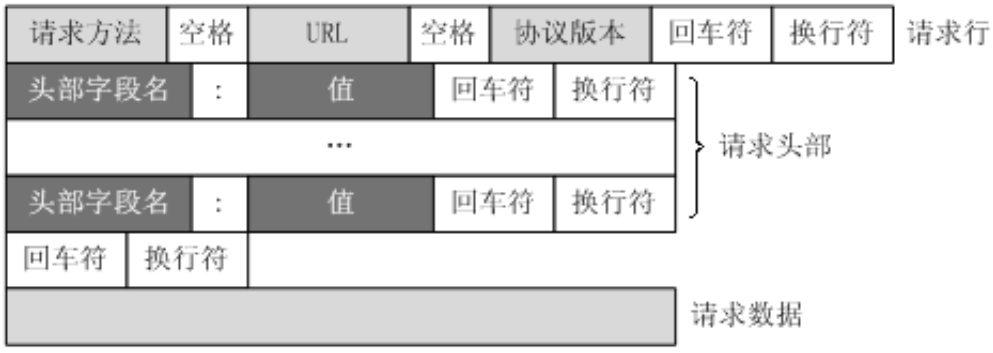
客户端请求消息和服务器响应消息都会包含请求头和请求体。HTTP请求头提供了关于请求或响应，发送实体的信息，如：Content-Type、Content-Length、Date等。当浏览器接收并显示网页前，此网页所在的服务器会返回一个包含HTTP状态码的信息头（server header）用以响应浏览器的请求。

> HTTP是一种让web服务器与客户端通过Internet发送与接受数据的协议，它是一个请求、响应协议，客户端建立连接并发送请求。服务器不能主动去与客户端联系，也不能发送一个回调连接，客户端可提前中断连接。
>
> HTTP请求是无状态的，同一个客户端的每个请求之间没有关联，对HTTP服务器来说，它并不知道这两个请求是否来自同一个客户端。为了解决这个问题引入了cookie机制来维护链接的可持续状态。

2 HTTP请求包

我们先来看看Request包的结构：

由 请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。



第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的HTTP版本

GET说明请求类型为GET,`/`为要访问的资源，该行的最后一部分说明使用的是`HTTP1.1`版本。

第二部分：请求头部，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息

从第二行起为请求头部，HOST将指出请求的目的地.User-Agent,服务器端和客户端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.该信息由你的浏览器来定义,并且在每个请求中自动发送等等

第三部分：空行，请求头部后面的空行是必须的

即使第四部分的请求数据为空，也必须有空行。

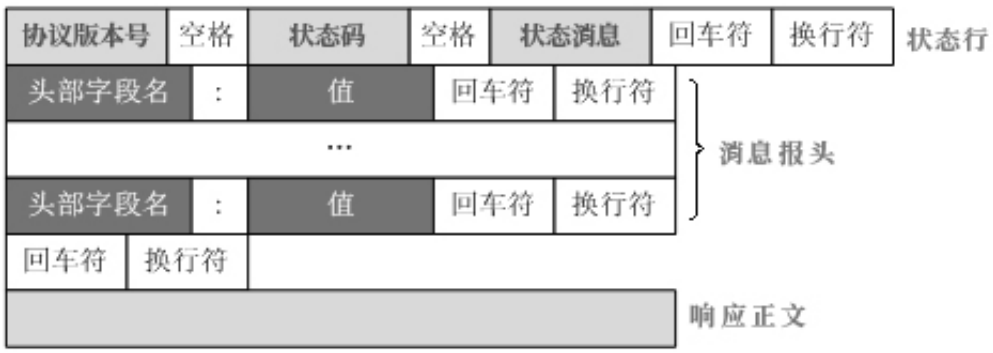
第四部分：请求数据也叫主体，可以添加任意的其他数据

请求包的例子：

```
GET http://edu.kongyixueyuan.com/ HTTP/1.1      // 请求行：请求方法请求URI HTTP协议/ 协议版本
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/gif, image/pjpeg, application/x-ms-xbap, */*          // 客户端能接收的数据格式
Accept-Language: zh-CN
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Win64; x64; Trident/4.0; .NET CLR 2.0.50727; SLCC2; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
UA-CPU: AMD64
Accept-Encoding: gzip, deflate                  // 是否支持流压缩
Host: edu.kongyixueyuan.com                     // 服务端的主机名
Connection: Keep-Alive
//空行，用于分割请求头和消息体
//消息体,请求资源参数,例如POST传递的参数
```


3 HTTP响应包

我们再来看看HTTP的response包，也由四个部分组成，分别是：状态行、消息报头、空行和响应正文。



第一部分：状态行，由HTTP协议版本号， 状态码， 状态消息 三部分组成。

第一行为状态行，（HTTP/1.1）表明HTTP版本为1.1版本，状态码为200，状态消息为（ok）

第二部分：消息报头，用来说明客户端要使用的一些附加信息

第二行和第三行为消息报头， Date:生成响应的日期和时间；Content-Type:指定了MIME类型的HTML(text/html),编码类型是UTF-8

第三部分：空行，消息报头后面的空行是必须的

第四部分：响应正文，服务器返回给客户端的文本信息。

空行后面的html部分为响应正文。

响应包的例子：

```
HTTP/1.1 200 OK                                          //状态行
Server: nginx                                            //服务器使用的WEB软件名及版本
Content-Type: text/html; charset=UTF-8                   //服务器发送信息的类型
Connection: keep-alive                                   //保持连接状态
Set-Cookie: PHPSESSID=mjup58ggbefu7ni9jea7908kub; path=/; HttpOnly
Cache-Control: no-cache
Date: Wed, 14 Nov 2018 08:27:32 GMT                      //发送时间
Content-Length: 99324                                    //主体内容长度
                                                         //空行用来分割消息头和主体
<!DOCTYPE html>...                                       //消息体
```

状态码用来告诉HTTP客户端, HTTP服务器是否产生了预期的Response。HTTP/1. 1协议中定义了5类状态码，状态码由三位数字组成，第一个数字定义了响应的类别。(HTTP状态码的英文为HTTP Status Code)

- 1XX 提示信息一表示请求已被成功接收，继续处理
- 2XX 成功一表示请求已被成功接收，理解，接受
- 3XX 重定向-要完成请求必须进行更进一步的处理
- 4XX 客户端错误-请求有语法错误或请求无法实现
- 5XX 服务器端错误-服务器未能实现合法的请求

> 常见状态码：
>
> ```
> 200 OK                          //客户端请求成功
> 400 Bad Request                 //客户端请求有语法错误，不能被服务器所理解
> 401 Unauthorized                //请求未经授权，这个状态代码必须和WWW-
> Authenticate报头域一起使用
> 403 Forbidden                   //服务器收到请求，但是拒绝提供服务
> 404 Not Found                   //请求资源不存在，eg：输入了错误的URL
> 500 Internal Server Error       //服务器发生不可预期的错误
> 503 Server Unavailable          //服务器当前不能处理客户端的请求，一段时间后可能恢复
> 正常
> ```

HTTP协议是无状态的和Connection: keep alive的区别 无状态是指协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。从另一方面讲，打开一个服务器上的网页和你之前打开这个服务器上的网页之间没有任何联系。 HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议(面对无连接)。

从HTTP/1.1起，默认都开启了Keep-Alive保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的TCP连接。 Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同服务器软件(如Apache) 中设置这个时间。

4 请求方法

根据HTTP标准，HTTP请求可以使用多种请求方法。 HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方法。 HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

```
GET         请求指定的页面信息，并返回实体主体。
HEAD        类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头
POST        向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。
PUT         从客户端向服务器传送的数据取代指定的文档的内容。
DELETE      请求服务器删除指定的页面。
CONNECT     HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。
OPTIONS     允许客户端查看服务器的性能。
TRACE       回显服务器收到的请求，主要用于测试或诊断。
```

常用方法: Get\Post\Head

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是**GET，POST，PUT，DELETE，**对应着对这个资源的查 ，改 ，增 ，删 4个操作。

另外还有个Head方法. 类似GET方法，只请求页面的首部，不响应页面Body部分，用于获取资源的基本信息，即检查链接的可访问性及资源是否修改。

GET和POST的区别：

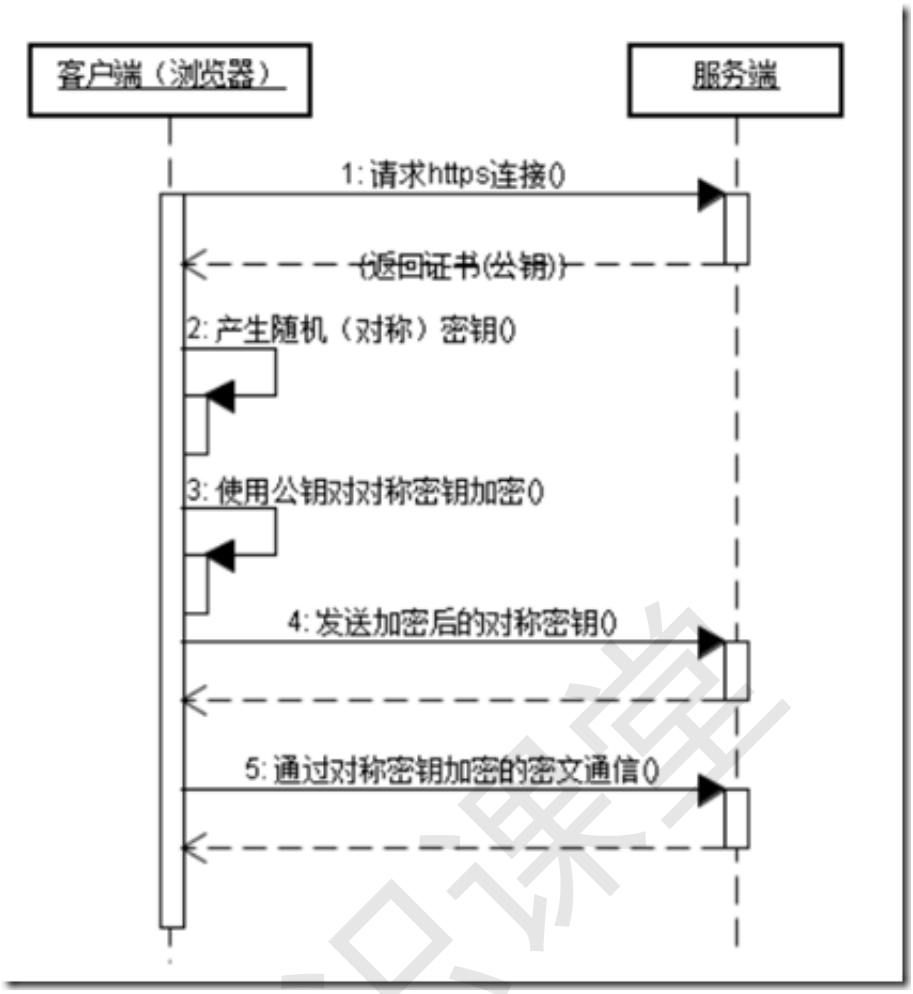
- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST没有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中。

HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）

HTTPS通信原理

HTTPS（Secure Hypertext Transfer Protocol）安全超文本传输协议 它是一个安全通信通道

HTTPS是HTTP over SSL/TLS，HTTP是应用层协议，TCP是传输层协议，在应用层和传输层之间，增加了一个安全套接层SSL。



服务器 用RSA生成公钥和私钥把公钥放在证书里发送给客户端，私钥自己保存客户端首先向一个权威的服务器检查证书的合法性，如果证书合法，客户端产生一段随机数，这个随机数就作为通信的密钥，我们称之为对称密钥，用公钥加密这段随机数，然后发送到服务器服务器用密钥解密获取对称密钥，然后，双方就已对称密钥进行加密解密通信了。

Https的作用

- 内容加密 建立一个信息安全通道，来保证数据传输的安全；
- 身份认证 确认网站的真实性
- 数据完整性 防止内容被第三方冒充或者篡改

Https和Http的区别

- https协议需要到CA申请证书。
- http是超文本传输协议，信息是明文传输；https 则是具有安全性的ssl加密传输协议。
- http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

HTTP/2

1. 二进制传输

2. Header压缩
3. 多路复用
4. Server Push

推荐阅读：详解HTTP2四大核心特性

IP TCP UDP HTTP

> 通过对七层网络协议的了解，IP协议对应网络层，TCP协议对应于传输层，而http协议对应于应用层，从本质上来说，三者是不同层面的东西，如果打个比方的话，IP就像高速公路，TCP就如同卡车，http就如同货物，货物要装载在卡车并通过高速公路才能从一个地点送到另一个地点。

那TCP与UDP的区别又是什么呢?

- **TCP** 传输控制协议，Transmission Control Protocol TCP是一种面向连接的、可靠的、基于字节流的传输层通信协议。
- **UDP** 用户数据报协议,User Datagram Protocol UDP是OSI（Open System Interconnection，开放式系统互联） 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

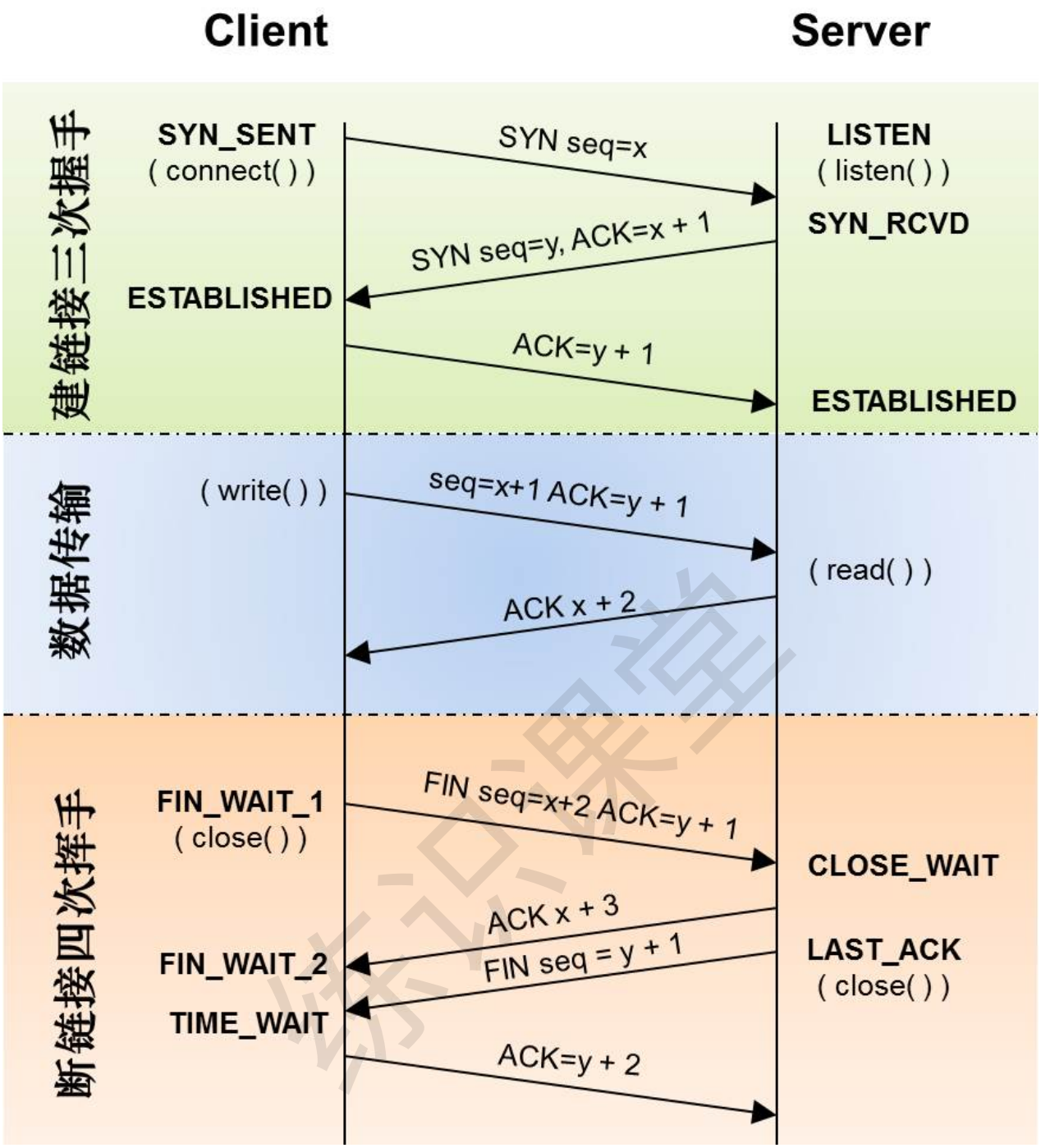
TCP是面向连接的传输控制协议，提供可靠的数据服务（类似于打电话） UDP是提供无连接的数据报服务，传输不可靠，可能丢包（类似于发短信） TCP首部开销20字节，UDP首部开销8字节 TCP只能是点到点的连接，UDP支持一对一，一对多，多对一，多对多的交互通信 TCP逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

注:什么是单工、半双工、全工通信？ 信息只能单向传送为单工； 信息能双向传送但不能同时双向传送称为半双工； 信息能够同时双向传送则称为全双工。

TCP的三次握手

> TCP建立一个连接需要3次握手IP数据包，断开连接需要4次握手。TCP因为建立连接、释放连接、IP分组校验排序等需要额外工作，速度较UDP慢许多。TCP适合传输数据，UDP适合流媒体

第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN_SEND状态，等待服务器确认； 第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态； 第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

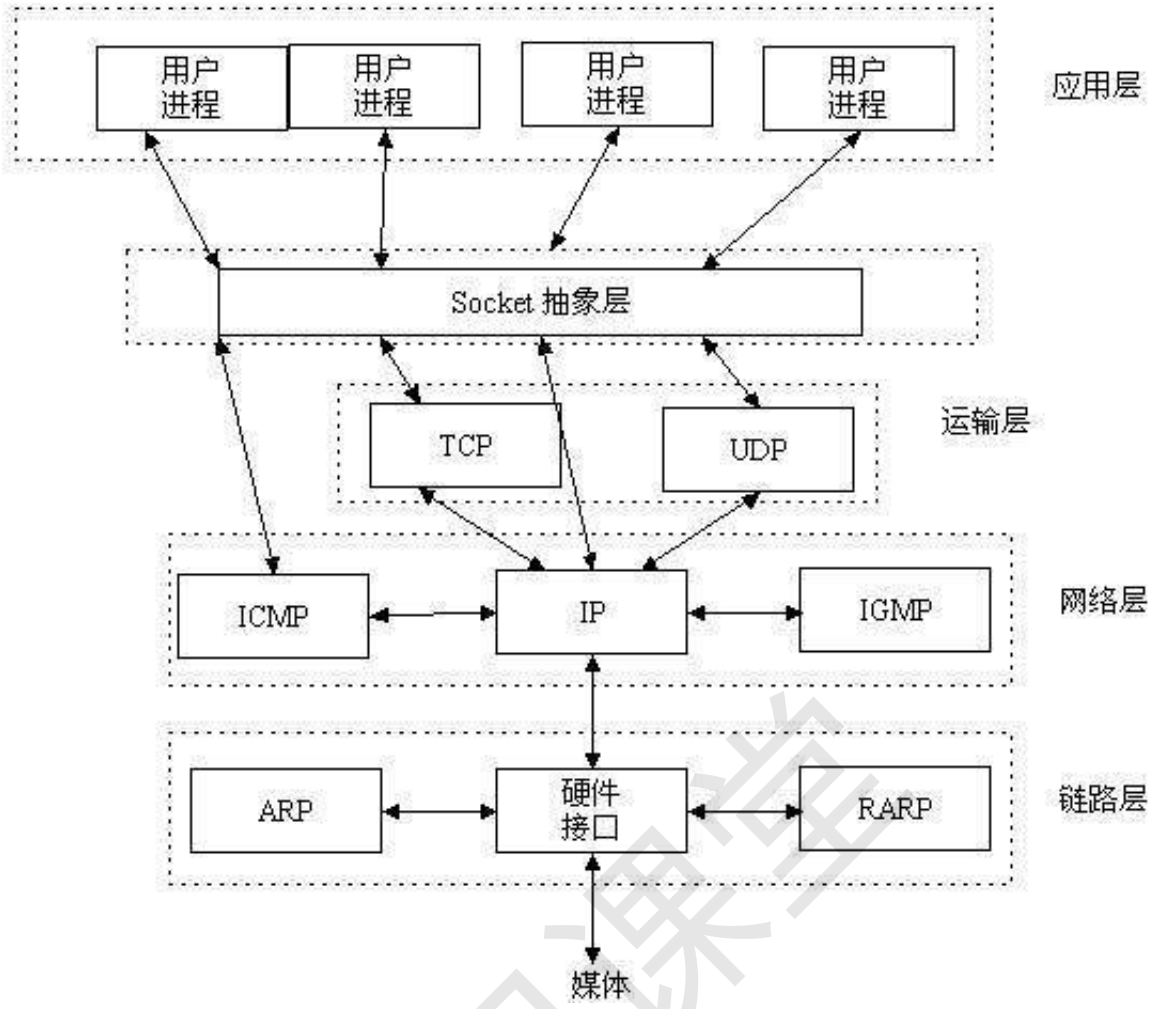


tcp-ip-handshark.png

Socket

我们知道两个进程如果需要进行通讯最基本的一个前提能能够唯一的标示一个进程，在本地进程通讯中我们可以使用PID来唯一标示一个进程，但PID只在本地唯一，网络中的两个进程PID冲突几率很大，这时候我们需要另辟它径了，我们知道IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这样我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。

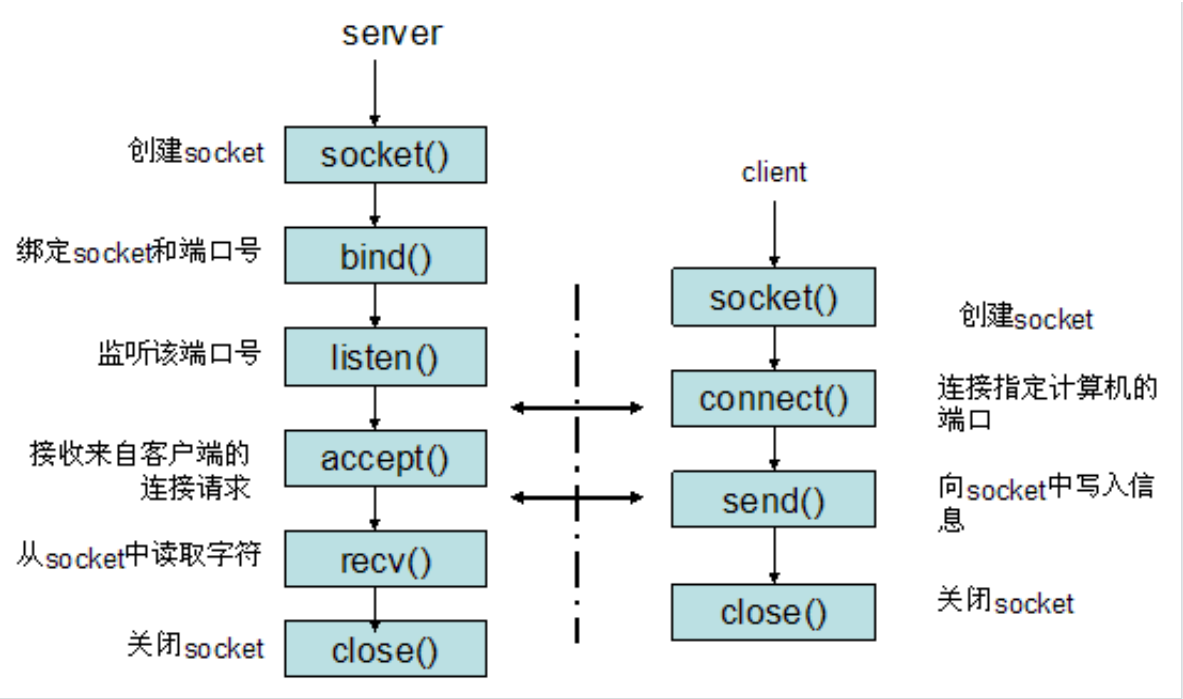
能够唯一标示网络中的进程后，它们就可以利用socket进行通信了，什么是socket呢？我们经常把socket翻译为套接字，socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。



socket.jpg

socket起源于UNIX，在Unix一切皆文件哲学的思想下，socket是一种"打开—读/写—关闭"模式的实现，服务器和客户端各自维护一个"文件"，在建立连接打开后，可以向自己文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。

Socket通信流程 socket是"打开—读/写—关闭"模式的实现，以使用TCP协议通讯的socket为例，其交互流程大概是下图这样的：



Go代码示例

服务端

```
// tcp/server/main.go

// TCP server端

// 处理函数
func process(conn net.Conn) {
  defer conn.Close() // 关闭连接
  for {
    reader := bufio.NewReader(conn)
    var buf [128]byte
    n, err := reader.Read(buf[:]) // 读取数据
    if err != nil {
      fmt.Println("read from client failed, err:", err)
      break
    }
    recvStr := string(buf[:n])
    fmt.Println("收到client端发来的数据：", recvStr)
    conn.Write([]byte(recvStr)) // 发送数据
  }
}

func main() {
  listen, err := net.Listen("tcp", "127.0.0.1:20000")
  if err != nil {
    fmt.Println("listen failed, err:", err)
    return
  }
  for {
```

```
    conn, err := listen.Accept() // 建立连接
    if err != nil {
      fmt.Println("accept failed, err:", err)
      continue
    }
    go process(conn) // 启动一个goroutine处理连接
  }
}
```

client端

```
// tcp/client/main.go

// 客户端
func main() {
  conn, err := net.Dial("tcp", "127.0.0.1:20000")
  if err != nil {
    fmt.Println("err :", err)
    return
  }
  defer conn.Close() // 关闭连接
  inputReader := bufio.NewReader(os.Stdin)
  for {
    input, _ := inputReader.ReadString('\n') // 读取用户输入
    inputInfo := strings.Trim(input, "\r\n")
    if strings.ToUpper(inputInfo) == "Q" { // 如果输入q就退出
      return
    }
    _, err = conn.Write([]byte(inputInfo)) // 发送数据
    if err != nil {
      return
    }
    buf := [512]byte{}
    n, err := conn.Read(buf[:])
    if err != nil {
      fmt.Println("recv failed, err:", err)
      return
    }
    fmt.Println(string(buf[:n]))
  }
}
```

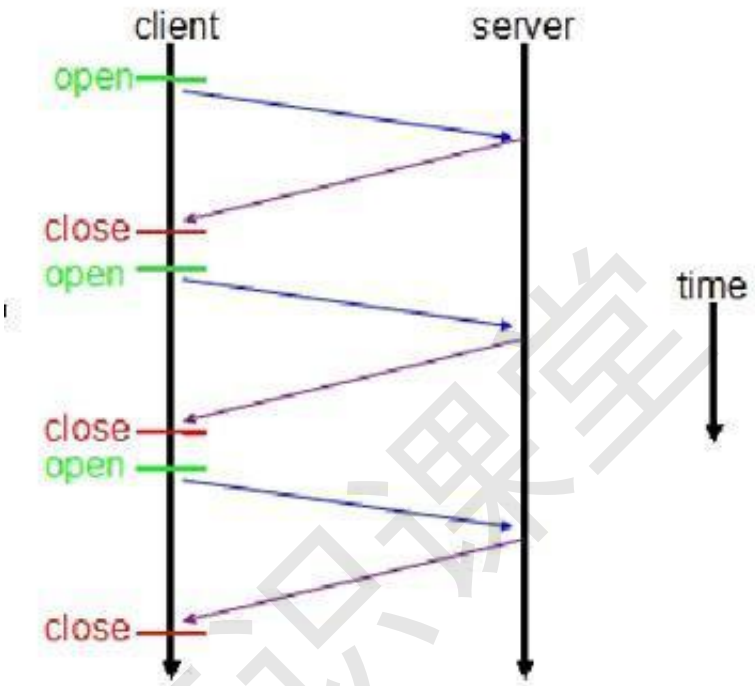

WebSocket

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信，能更好的节省服务器资源和带宽并达到实时通讯它建立在TCP之上，同 HTTP一样通过TCP来传输数据。WebSocket同HTTP一样也是应用层的协议，并且一开始的握手也需要借助HTTP请求完成。

它和 HTTP 最大不同是：

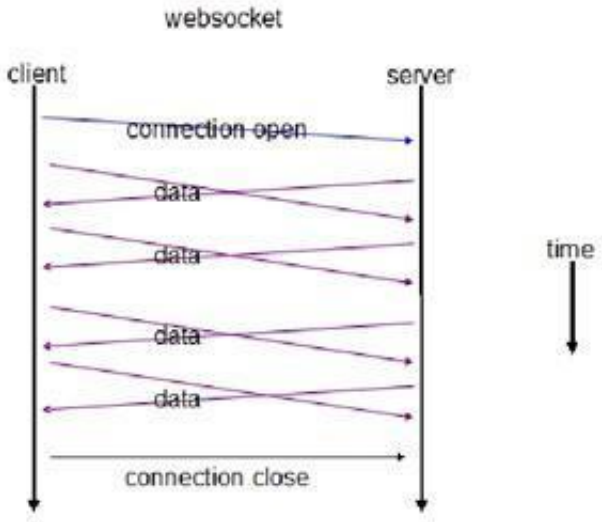
- WebSocket 是一种双向通信协议，在建立连接后，WebSocket 服务器和 Browser/Client Agent 都能主动的向对方发送或接收数据，就像 Socket 一样；
- WebSocket 需要类似 TCP 的客户端和服务器端通过握手连接，连接成功后才能相互通信。

HTTP请求客户端服务器交互图



http-client-server.jpg

WebSocket客户端服务器交互图



websocket-client-server.jpg

上图对比可以看出，相对于传统 HTTP 每次请求-应答都需要客户端与服务端建立连接的模式，WebSocket 是类似 Socket 的 TCP 长连接的通讯模式，一旦 WebSocket 连接建立后，后续数据都以帧序列的形式传输。在客户端断开 WebSocket 连接或 Server 端断掉连接前，不需要客户端和服务端重新发起连接请求。在海量并发及客户端与服务器交互负载流量大的情况下，极大的节省了网络带宽资源的消耗，有明显的性能优势，且客户端发送和接受消息是在同一个持久连接上发起，实时性优势明显。

WebSocket连接过程（握手）

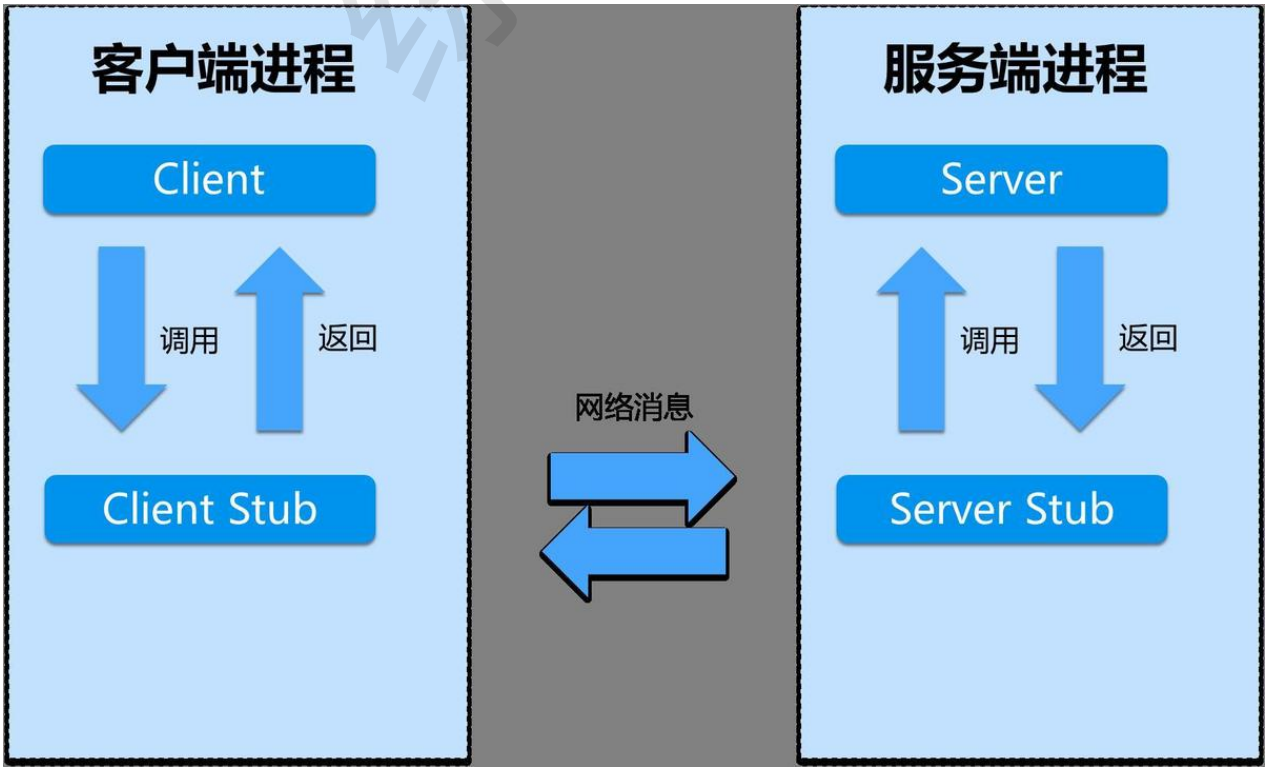
从WebSocket客户端服务器交互图可以看出，在WebSocket中，只需要服务器和浏览器通过HTTP协议进行一个握手的动作，然后单独建立一条TCP的通信通道进行数据的传送。

1. 浏览器，服务器建立TCP连接，三次握手。这是通信的基础，传输控制层，若失败后续都不执行。
2. TCP连接成功后，浏览器通过HTTP协议向服务器传送WebSocket支持的版本号等信息。（开始前的HTTP握手）
3. 服务器收到客户端的握手请求后，同样采用HTTP协议回馈数据。
4. 当收到了连接成功的消息后，通过TCP通道进行传输通信。

RPC

> Remote Procedure Call 远程过程调用 它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

先说说RPC服务的基本架构吧。一个完整的RPC架构里面包含了四个核心的组件，分别是Client ,Server,Client Stub以及Server Stub，这个Stub大家可以理解为存根。



rpc-architecture.jpg

- 客户端（Client），服务的调用方。
- 服务端（Server），真正的服务提供者。
- 客户端存根，存放服务端的地址消息，再将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方。
- 服务端存根，接收客户端发送过来的消息，将消息解包，并调用本地的方法。

RPC采用客户机/服务器模式，通信是建立在Socket之上的,出于一种类比的愿望,在一台机器上运行的主程序,可以调用另一台机器上准备好的子程序,就像LPC(本地过程调用)。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用过程接收答复信息，获得进程结果，然后调用执行继续进行。

RPC vs HTTP

- 论复杂度，RPC框架肯定是高于简单的HTTP接口的。但毋庸置疑，HTTP接口由于受限于HTTP协议，需要带HTTP请求头，还有三次握手，导致传输起来效率或者说安全性不如RPC。
- HTTP是一种协议,RPC可以通过HTTP来实现,也可以通过Socket自己实现一套协议来实现.
- RPC更是一个软件结构概念，是构建分布式应用的理论基础。就好比为啥你家可以用到发电厂发出来的电？是因为电是可以传输的。至于用铜线还是用铁丝还是其他 种类的导线，也就是用http还是用其他协议的问题了。

REST & RESTful

> REST全称是Representational State Transfer，中文意思是表述性状态转移。REST指的是一组架构约束条件和原则。如果一个架构符合REST的约束条件和原则，我们就称它为RESTful架构。

然而REST本身并没有创造新的技术、组件或服务，而隐藏在RESTful背后的理念就是使用Web的现有特征和能力， 更好地使用现有Web标准中的一些准则和约束。我们现在所说的Rest是基于HTTP协议之上来讲的，但Rest架构风格并不是绑定在HTTP上，只不过目前HTTP是唯一与REST相关的实例。

REST架构的主要原则

- 在REST中的一切都被认为是一种资源。
- 每个资源由URI标识。
- 使用统一的接口。处理资源使用POST，GET，PUT，DELETE操作类似创建，读取，更新和删除（CRUD）操作。
- 无状态：每个请求是一个独立的请求。从客户端到服务器的**每个请求**都必须包含所有必要的信息，以便于理解。
- 同一个资源具有多种表现形式，例如XML，JSON

RESTful API 简单例子

```
[POST]      http://localhost/users    // 新增
[GET]       http://localhost/users/1  // 查询
[PATCH]     http://localhost/users/1  // 更新
[PUT]       http://localhost/users/1  // 覆盖，全部更新
[DELETE]    http://localhost/users/1  // 删除
```

微信搜一搜

练识课堂