ISO（国际标准化组织）制定的网络七层模型

物理层、数据链路层、网络层、传输层、会话层、表示层、应用层

TCP/IP网络五层模型

物理层、数据链路层、网络层、传输层、应用层

+++

应用层

TCP（HTTP）长链接和短连接的区别

> HTTP 的长连接和短连接本质上是 TCP 长连接和短连接。HTTP 属于应用层协议，在传输层使用 TCP 协议，在网络层使用 IP 协议。IP 协议主要解决网络路由和寻址问题，TCP 协议主要解决如何在 IP 层之上可靠的传递数据包，使在网络上的另一端收到发端发出的所有包，并且顺序与发出顺序一致。TCP 有可靠，面向连接的特点。

- tcp短连接
  client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次读写就完成了，这时候双方任何一个都可以发起close操作，不过一般都是client先发起close操作。为什么呢，一般的server不会回复完client后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在client/server间传递一次读写操作
  客户端和服务器之间的TCP链接只能为一个HTTP请求服务，当服务器处理完客户的一次HTTP请求之后就会主动将TCP连接关闭。此后如果客户与同一个服务器进行多次HTTP请求的话还需要重新建立TCP连接。也就是说客户的多次HTTP请求不能共用一个TCP连接。
- tcp长链接

client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。多次HTTP请求共用同一个TCP连接。
在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。
但是我不能一直连接啊，如果客户端消失了难倒我服务器端还傻傻的等待吗？这里面有一个机制叫做保活机制。在一段时间内连接处于非活动状态（一般是两个小时），那么服务器端将向客户端发送一个报文，如果服务端没有收到回应，则在一定时间间隔内还会继续发送。当发送次数达到阈值（保活次数）后，确认对方主机不可达，断开连接。发送探测消息，客户端有四种状态：

i. 对方主机正常，但对方不想赢，超过次数后关闭
ii. 对方正常，但因为网络原因没收到确认报文
iii. 对方主机崩溃了，不会响应报文
iv. 对方主机崩溃然后重启，服务器会受到响应报文，然后关掉链接

TCP Keepalive 和 HTTP Keep-Alive 是一个东西吗？

这是两个完全不一样的东西，毕竟长得都不一样

> HTTP 的 Keep-Alive，是由**应用层（用户态）** 实现的，称为 HTTP 长连接；
>
> TCP 的 Keepalive，是由 **TCP 层（内核态）** 实现的，称为 TCP 保活机制；

- **HTTP 的 Keep-Alive**
  http协议采用的是请求—应答模式，即客户端发起请求，服务端会返回响应。
  但是有一个网页有很多组成部分，除了文本还有图片，视频等静态资源。如果每一个资源都创建一个连接然后关闭，代价太大了。而且每次请求都是建立 TCP -> 请求资源 -> 响应资源 -> 释放连接 这样的**短连接**方式太累了，一次只能请求一个资源。所以想能不能在第一个 HTTP 请求完后，先不断开 TCP 连接，让后续的 HTTP 请求继续使用此连接？
  所以HTTP 的 Keep-Alive 就是实现了这个功能，可以使用同一个 TCP 连接来发送和接收多个 HTTP 请求/应答，避免了连接建立和释放的开销，这个方法称为 **HTTP**

长连接。
HTTP1.0中是默认关闭的。通过headers设置"Connection: Keep-Alive"开启。HTTP的Keep-Alive是HTTP1.1中默认开启的功能。通过headers设置"Connection: close "关闭。

- **TCP 的 Keepalive**
  TCP 的 Keepalive 这东西其实就是 **TCP 的保活机制**
  如果两端的 TCP 连接一直没有数据交互，达到了触发 TCP 保活机制的条件，那么内核里的 TCP 协议栈就会发送探测报文。
    - 如果对端程序是正常工作的。当 TCP 保活的探测报文发送给对端, 对端会正常响应，这样 **TCP 保活时间会被重置**，等待下一个 TCP 保活时间的到来。
    - 如果对端主机崩溃，或对端由于其他原因导致报文不可达。当 TCP 保活的探测报文发送给对端后，石沉大海，没有响应，连续几次，达到保活探测次数后，**TCP 会报告该 TCP 连接已经死亡**。

  所以，TCP 保活机制可以在双方没有数据交互的情况，通过探测报文，来确定对方的 TCP 连接是否存活，这个工作是在内核完成的。

HTTP协议详解

参考

请求报文

HTTP协议是以ASCⅡ码传输，建立在TCP/IP协议之上的应用层规范。
HTTP请求报文由请求行（request line）、请求头部（header）、空行和请求数据四个部分组成，下图是请求报文的一般格式。

请求换行请求行是由请求方法、url字段以及HTTP协议版本字段三个部分组成，它们用空格分开。HTTP协议的请求方法有GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT。

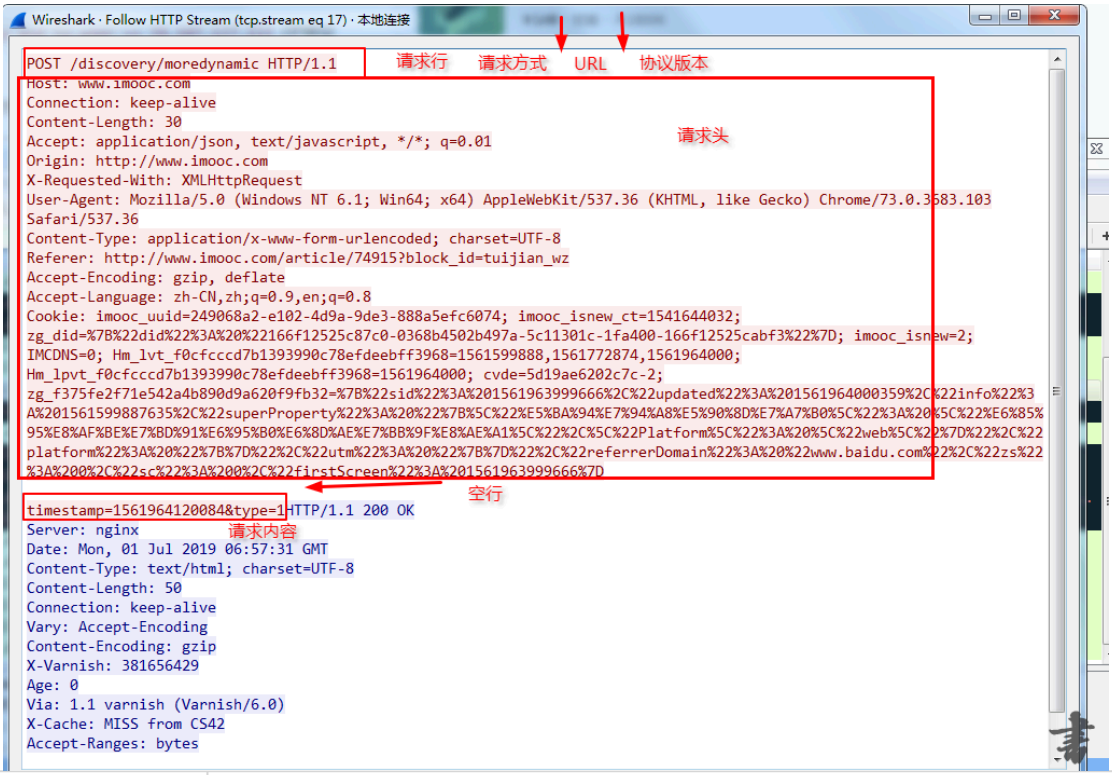
而常见的有如下几种：

> **GET**
>
> 当客户端要从服务器读取数据时，点击网页上的链接，或者通过在浏览器的地址栏输入网址来浏览网页的，使用的都是GET方式。GET的方法要求服务器将URL定位的资源放在响应报文的数据部分，回送给客户端。回送给客户端。使用GET的方法，请求参数和对应的值附加在URL后面，利用问号（？），代表URL的结尾和请求参数的开始，传递的参数长度受到限制。例如，/index.html?id=1&password=123,这样通过GET的方式传递的数据直接显示在地址上，所以我们可以将请求以链接的方式发送给接收方。缺点：但是这种方式显然不能传递私密的数据。另外不同浏览器对地址的字符长度限制有不同的数据，一般最多不超过1024个字符，所以大量的数据传输，不适合使用GET方式。
>
> **POST**
>
> POST将请求参数封装在HTTP的请求数据中，可以大量的传递数据，理论上对数据得大小的没有限制，但实际各个WEB服务器会规定对post提交数据大小进行限制。而且可以不显示在URL中。一个post的http请求如下：

POST方法向服务器提交数据，比如表单的数据的提交。GET的方法一般用于获取/查询资源信息。

- 请求头

请求头部由键值对组成，关键字和值之间用：分开，所以一般用unordered_map来存储请求头。请求头封装了有关客户端请求的信息，典型的请求头有：



请求头	含义
User-Agent	产生请求的浏览器类型，User-Agent请求报头域允许客户端将它的操作系统
Accept	客户端可识别的响应内容类型列表。eg：Accept：image/gif，表明客户端希
Accept-Language	客户端可接受的自然语言
Accept-chartset	客户端可接受应答的字符集。eg：Accept-Charset:iso-8859-1,gb2312.如果
Accept-Encoding	客户端可接受的编码压缩格式
HOST	请求的主机名称，允许多个域名同处一个IP之地，即虚拟主机
Connection	连接方式（close或keep-alive）
Cookie	存储于客户端扩展字段，向同一域名的服务端发送该域的cookie
Authorization	Authorization请求报头域主要用于证明客户端有权查看某个资源。当浏览器

- 空行

最后一个请求头之后是一个空行，发送回车符和换行符(\r\n)，通知服务器以下不会再有请求头。

- 请求体
  请求数据不再GET方法中使用，而是在POST方法中使用。POST方法适用于客户端提交表单。与请求数据相关的最常使用的请求头是包体类型 Content-Type 和包体长度 Content-Length

响应报文

HTTP响应报文是由状态行、响应头部、空行和响应包体四个部分组成，如下图所示：

- 状态行
  状态行由HTTP协议版本（HTTP-Version），状态码（Statue-Code） 和 状态码描述文本（Reason-Phrase） 三个部分组成，它们之间用空格隔开；

> 状态码有三位数字组成，第一位定义了响应的类别，可能有五种取值：
>
> - **1xx**：表示服务端已经接收到客户端请求，客户端可以继续发送请求；
> - **2xx**：表示服务端已经成功接收请求并处理；
> - **3xx**：表示服务器要求客户端重定向；
> - **4xx**：客户端请求有问题；
> - **5xx**：服务端未能正常处理客户端的请求出现错误；
>
> 常见状态码描述文本有如下：
>
> - **200 OK**：请求成功；
> - **400 Bad Request**：客户端请求语法有问题，不能被服务端理解；
> - **401 Unauthorized**：请求未经授权，必须与Authorization请求报头域一起使用（eg：BASE64用户身份验证）；

- **403 Forbidden**：服务器收到请求但是拒绝提供服务，通常会在响应正文中给出不提供服务的原因；
- **404 Not Found**：请求的资源不存在，eg，输错了URL；
- **500 Internal Server Error**：服务器发生错误，无法完成客户端请求；
- **503 Service Unavailable**：表示服务器当前不能处理客户端请求，一段时间之后可能恢复正常；

- 响应头

  响应头可能包括以下信息：

响应头	描述
Server	Server 响应报头域包含了服务器用来处理请求的软件信息及其版本。它和 U 请求报头域是相对应的，前者发送服务器端软件的信息，后者发送客户端软
Vary	指示不可缓存的请求头列表
Connection	连接方式
www-Authenticate	WWW-Authenticate响应报头域必须被包含在401 (未授权的)响应消息中，这 请求报头域是相关的，当客户端收到 401 响应消息，就要决定是否请求服务器对其进行验证。如果要求服务器对其进 报头域的请求

- 响应体

  服务器返回给客户端的文本信息，如下图：

http的无状态性

HTTP协议是无状态的（stateless）。也就是说，同一个客户端第二次访问同一个服务器上面的页面时，服务器无法得知这个客户端曾经访问过，服务器无法辨别不同的客户端。HTTP的无状态性简化了服务器的设计，是服务器更容易支持大量并发的HTTP请求。HTTP协议是采用请求-响应的模型。客户端向服务端发送一个请求报文，服务端以一个状态作为回应。当使用普通模式，即非keep-alive模式时，每个请求-应答都要重新建立一个连接，连接完成后立即断开；

HTTP1.1 使用持久连接keep-alive，所谓持久连接，就是服务器在发送响应后仍然在一段时间内保持这条连接，允许在同一个连接中存在多次数据请求和响应，即在持久连接情况下，服务器在发送完响应后并不关闭TCP 连接，而客户端可以通过这个连接继续请求其他对象。

HTTP 长连接不可能一直保持，例如 Keep-Alive: timeout=5, max=100，表示这个TCP通道可以保持5秒，max=100，表示这个长连接最多接收100次请求就断开。HTTP 是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive 没能改变这个结果。另外，Keep-Alive也不能保证客户端和服务器之间的连接一定是活跃的，在 HTTP1.1 版本中也如此。唯一能保证的就是当连接被关闭时你能得到一个通知，所以不应该让程序依赖于 Keep-Alive 的保持连接特性，否则会有意想不到的后果。

浏览器控制台中的http报文

https和http区别

> https=http+ssl但是现在ssl都被tsl所取代

- 首先说明为什么http协议不安全？**首先要知道一个安全的机制应该满足三个特性：机密性，完整性，不可否认性。**http协议通信使用明文进行通信，且没有任何其他保护措施，我们依次来看一下不安全特性。首先是机密性，由于使用明文进行通信，导致信息会被窃听泄露；然后是完整性，一段裸奔的明文在网络上传递，会被轻而易举的篡改；最后是不可否认性，由于http的请求和响应不会对通信方的身份进行确认，导致很容会被欺骗，而且用户无法察觉
  但是https使得上述三个特性都满足。
  HTTPS并非是应用层的一种新协议。只是HTTP通信接口部分用SSL（Secure Socket Layer）和TLS（Transport Layer Security）协议代替而已。
  通常，HTTP直接和TCP通信。当使用SSL时，则演变成先和SSL通信，再由SSL和TCP通信了。简言之，**所谓HTTPS，其实就是身披SSL协议这层外壳的**

HTTP（SSL协议位于TCP和HTTP协议之间）。SSL主要包括几大块：

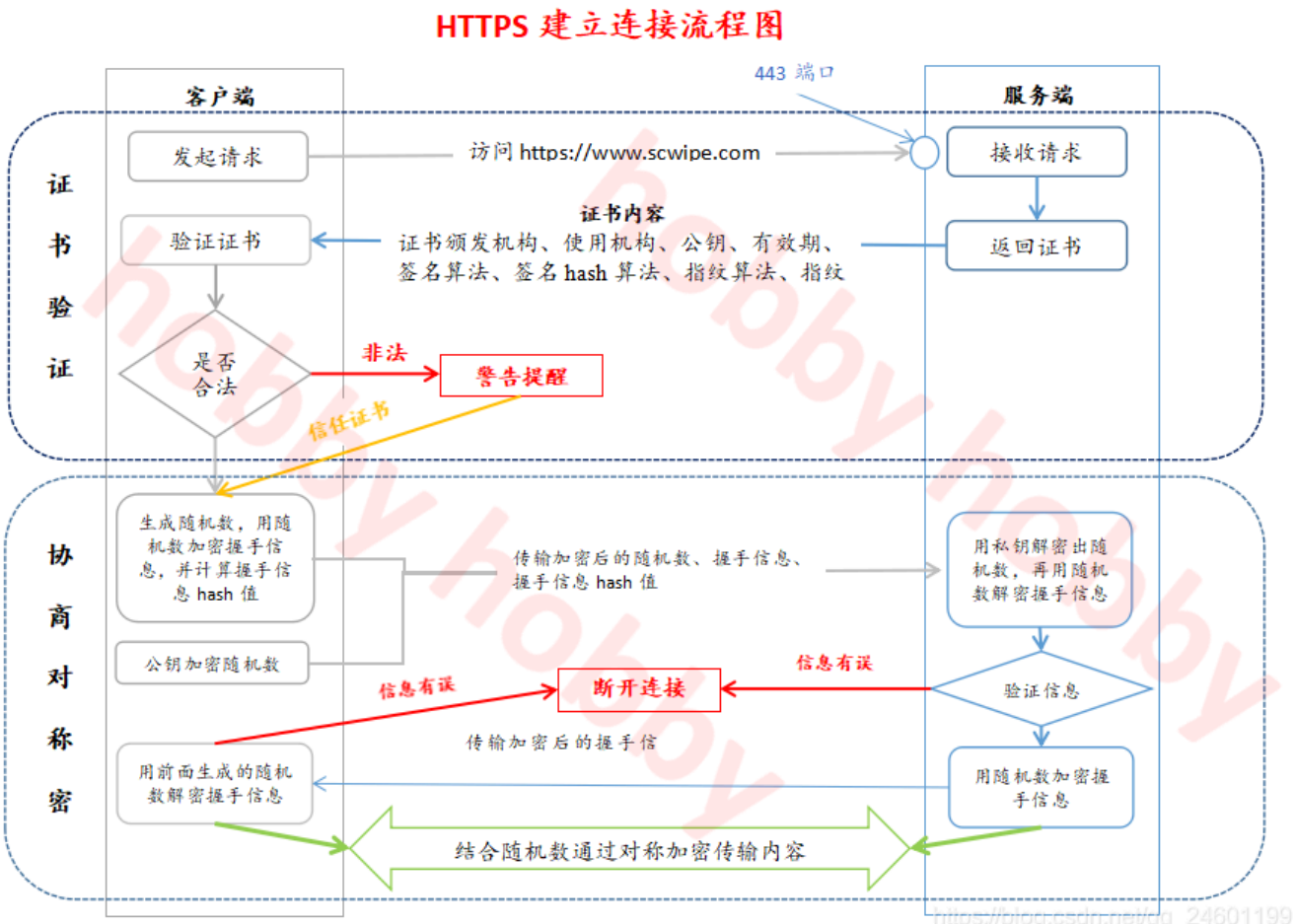
i. 解决机密性。对请求和应答消息进行加密。公钥加密效率比较低，对称加密效率较高。因此对于密钥协商这块，使用公钥两方协商密钥，然后用密钥执行对称加密对消息加密

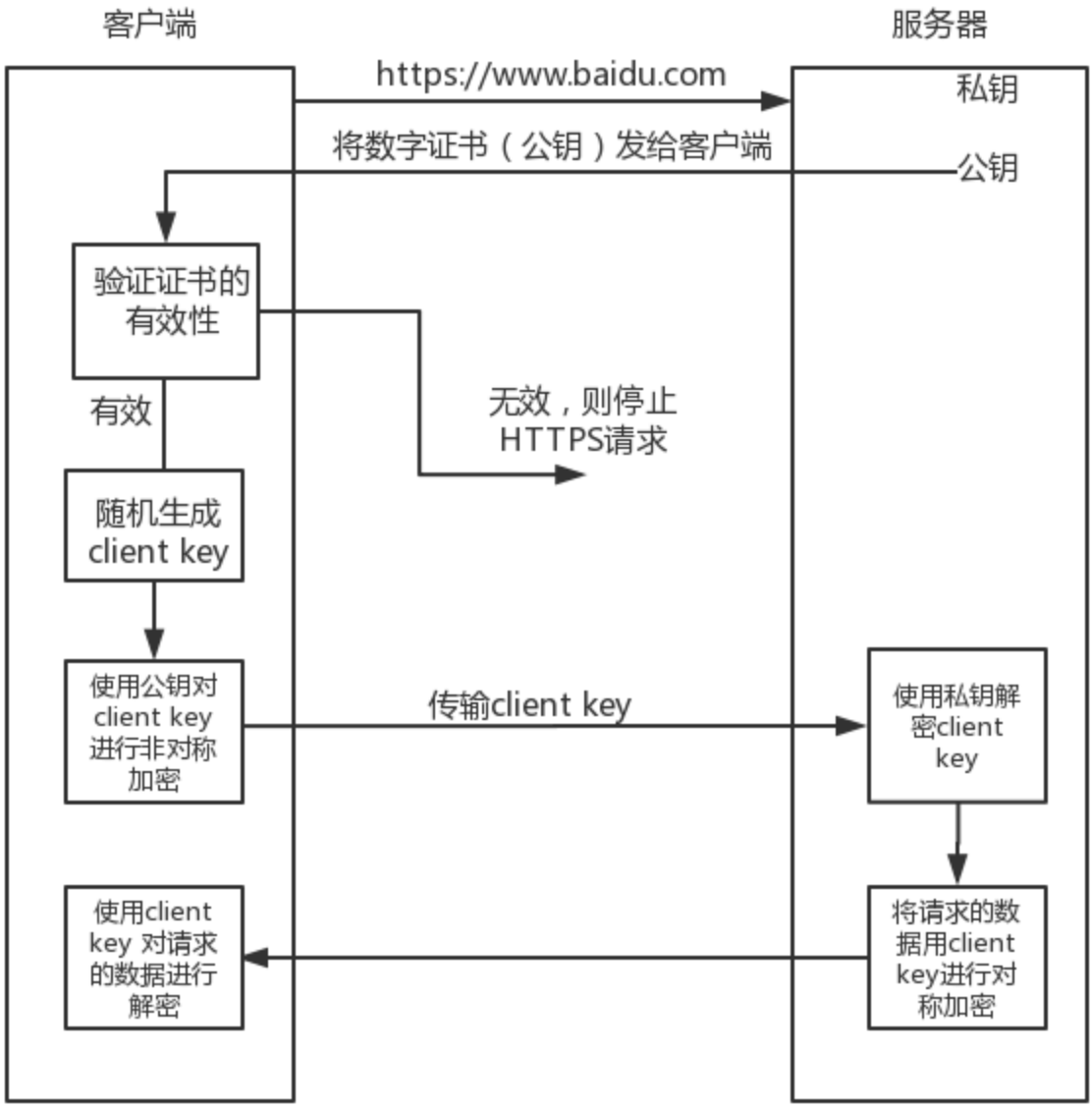
ii. 完整性。依靠单向散列函数实现，比如MD5或者SHA1

iii. 不可否认性。依靠数字签名来实现。

https加密算法

https的ssl连接过程如下图：





SSL/TLS密钥交换算法

两台机器如何用HTTP找到对方？

计算机网络输入URL到看到网页

访问一个网页的过程

这三个问题本质上都是一个问题

参考链接，写的不错

- **1、浏览器首先会解析URL**
  比如有一个网址：`https://www.webserver.com/dir/file`，其解析过程如下图所示：

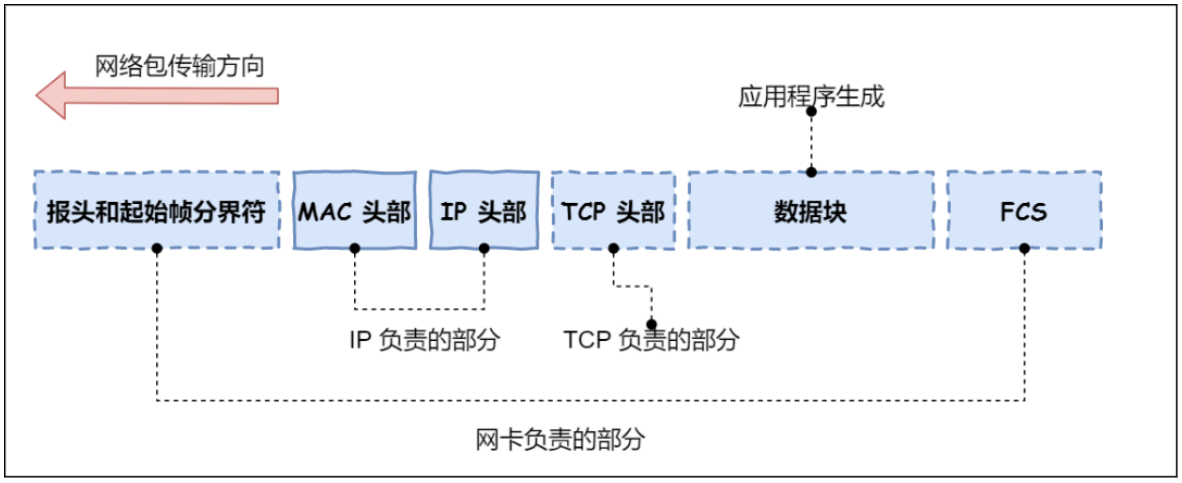
- **2、浏览器确定了服务器和文件名后，会产生HTTP请求消息**
  可以根据具体请求来封装成get请求或者是post请求
- 3、DNS协议
  在发送给服务器地址之前，要先查询服务器域名对应的IP地址
  常考知识点：DNS解析过程
- **4、走本机网络的协议栈**
  通过 DNS 获取到 IP 后，就可以把 HTTP 的传输工作交给操作系统中的协议栈。
  HTTP本质是基于TCP/IP协议栈的，所以在发送HTTP报文之前需要先和对方主机建立TCP连接，连接建立后经过TCP协议和IP协议的封装后，形成一个网络报文，即**IP头部+TCP头部+HTTP报文**。
  但是当到达对方网络后还需要经过数据链路层传送给对方，因此还需要在ip头部前加入MAC头部，所以最终从我们客户端主机出去的报文就是**MAC头部+IP头部+TCP**

头部+HTTP报文这么一个东西

- **5、经过本机网卡**

  这些数据本身是一串二进制信息，需要用网卡转成电信号，才能在网线上传输。

  网卡收到来自内核协议栈的数据后，会在开头加上报头和起始帧分界符，在末尾加上帧校验序列，如下图



- **6、到达交换机**

  交换机主要工作在二层中，内部维护了一个MAC地址表，通过这个表查找对应MAC地址所在的端口上，从这个端口转发出去。

  当地址表中找不到指定的 MAC 地址时，交换机无法判断应该把包转发到哪个端口，只能将包转发到除了源端口之外的所有端口上，无论该设备连接在哪个端口上都能收到这个包。只有相应的接收者才接收包，而其他设备则会忽略这个包。

- **7、路由器**

  路由器是工作在三层的网络设备，这一步转发的工作原理和交换机类似，也是通过查表判断包转发的目标。

  路由器有MAC地址的，所以他可以成为接收方和发送方。

  收到包后会解开MAC地址查看其目的IP地址，根据路由表中的转发规则选择转发，如果没有匹配规则就使用默认路由，路由表中子网掩码为 `0.0.0.0` 的记录表示默认路由。

  知道对方的 IP 地址之后，接下来需要通过 `ARP` 协议根据 IP 地址查询 MAC 地址，并将查询的结果作为接收方 MAC 地址。路由器也有 ARP 缓存，因此首先会在 ARP 缓存中查询，如果找不到则发送 ARP 查询请求。

traceroute原理

traceroute的作用：当源主机A向目标主机B发送消息，发现消息无法送达。此时，可能是某个中

间节点发生了问题，比如路由器02因负载过高产生了丢包。此时，可以通过traceroute进行初步的检测，定位网络包在是在哪个节点丢失的，之后才可以进行针对性的处理。命令如下：

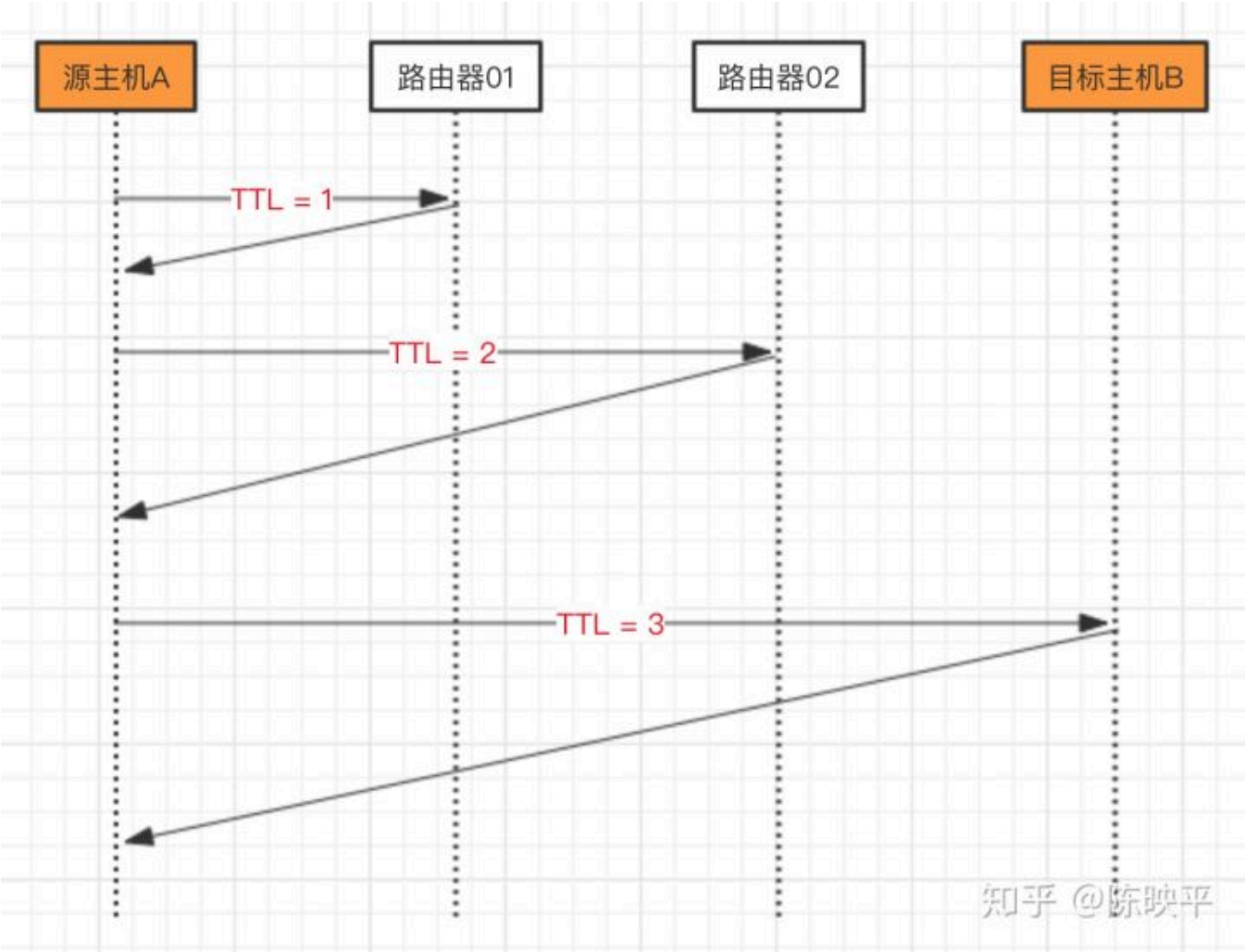
```
traceroute www.iqiyi.com
traceroute: Warning: www.iqiyi.com has multiple addresses; using 121.9.221.96
traceroute to static.dns.iqiyi.com (121.9.221.96), 64 hops max, 52 byte packets
 1  xiaoqiang (192.168.31.1)  1.733 ms  1.156 ms  1.083 ms
 2  192.168.1.1 (192.168.1.1)  2.456 ms  1.681 ms  1.429 ms
 # ... 忽略部分输出结果
 9  121.9.221.96 (121.9.221.96)  6.607 ms  9.049 ms  6.706 ms
```

> 注意，每次检测都同时发送3个数据包，因此打印出来三个时间。此外，如果某一个数据报超时没有返回，则时间显示为 *，此时需要特别注意，因为有可能出问题了

原理如下：

主机之间通信，网络层IP数据报的首部中，有个TTL字段(Time To Live)。TTL的作用是，设置IP数据报被丢弃前，最多能够经过的节点数。此外，每经过一个中间节点，再向下一个节点转发数据前，都会将TTL减1。如果TTL不为0，则将数据报转发到下一个节点；否则，丢弃数据报，并返回错误。

从源主机向目标主机发送IP数据报，并按顺序将TTL设置为从1开始递增的数字（假设为N），导致第N个节点（中间节点 or 目标主机）丢弃数据报并返回出错信息。源主机根据接收到的错误信息，确定到达目标主机路径上的所有节点的IP，以及对应的耗时。原理图如下：



- 问题1：用的UDP还是TCP协议
  答：UDP
- 问题二：当TTL为0时，接收节点会丢弃接数据报，并向源主机报错。这里的报错信息是什么？
  答：报错信息是ICMP（Internet Control Message Protocol）报文，它用于在主机、路由之间传递控制信息。
- 问题三：假设到达目标主机一共有N跳，且TTL刚好设置为N，那么，目标主机成功收到数据报，此时并没有错误回报，traceroute如何确定已经到达目标主机？
  答：traceroute发送UDP报文时，将目标端口设置为较大的值（ 33434 - 33464），避免目标主机B上该端口有在实际使用。当报文到达目标主机B，目标主机B发现目标端口不存在，则向源主机A发送ICMP报文（Type=3，Code=3），表示目标端口不可达。所以知道已经到达了目标主机

dns的解析过程

DNS是应用层协议，事实上他是为其他应用层协议工作的，包括不限于HTTP和SMTP以及FTP，用于将用户提供的主机名解析为ip地址。

DNS服务器一般分三种，根DNS服务器，顶级DNS服务器，二级DNS服务器。

如果某个用户正在用浏览器`mail.baidu.com`的网址，当你敲下回车键的一瞬间：

1. 检查**浏览器缓存**中是否存在该域名与IP地址的映射关系，如果有则解析结束，没有则继续
2. 到**系统本地**查找映射关系，一般在`hosts`文件中，如果有则解析结束，否则继续
3. 到**本地域名服务器**去查询，有则结束，否则继续
4. **本地域名服务器**查询**根域名服务器**，该过程并不会返回映射关系，只会告诉你去下级服务器(顶级域名服务器)查询
5. **本地域名服务器**查询**顶级域名服务器**(即`com`服务器)，同样不会返回映射关系，只会引导你去二级域名服务器查询
6. **本地域名服务器**查询**二级域名服务器**(即`baidu.com`服务器)，引导去三级域名服务器查询
7. **本地域名服务器**查询**三级域名服务器**(即`mail.baidu.com`服务器)，此时已经是最后一级了，如果有则返回映射关系，则**本地域名服务器**加入自身的映射表中，方便下次查询或其他用户查找，同时返回给该用户的计算机，没有找到则网页报错
8. 如果还有下级服务器，则依此方法进行查询，直至返回映射关系或报错

像该过程中的第1、2、3点，仅限于在`本地域名服务器`中查找，如果有则直接返回映射关系，否则就去其他`DNS`服务器中查询，这种查询方式我们叫做**递归查询**。

第3、4、5、6、7、8过程，他们只会给出下级`DNS`服务器的地址，并不会直接返回映射关系，这种查询方式叫做**迭代查询**

所有DNS请求和回答报文使用的UDP数据报经过端口53发送

为什么用UDP？因为一次UDP名字服务器交换可以短到两个包：一个查询包、一个响应包。一次TCP交换则至少包含9个包：三次握手初始化TCP会话、一个查询包、一个响应包以及四次分手的包交换。考虑到效率原因，TCP连接的开销大得，故采用UDP作为DNS的运输层协议，这也将导致只有13个根域名服务器的结果。

但不一定全部都是UDP，比如DNS的规范规定了2种类型的DNS服务器，一个叫主DNS服务

器，一个叫辅助DNS服务器。在一个区中主DNS服务器从自己本机的数据文件中读取该区的DNS数据信息，而辅助DNS服务器则从区的主DNS服务器中读取该区的DNS数据信息。当一个辅助DNS服务器启动时，它需要与主DNS服务器通信，并加载数据信息，这就叫做区传送(zone transfer)。区域传送时使用TCP，主要有一下两点考虑：②辅助域名服务器会定时(一般是3小时)向主域名服务器进行查询以便了解数据是否有变动。如有变动，则会执行一次区域传送，进行数据同步。②区域传送将使用TCP而不是UDP，因为数据同步传送的数据量比一个请求和应答的数据量要多得多。

cookie和session

参考

UDP实现TCP功能

> 首先要明白为什么tcp没有基于udp实现？
>
> ①历史原因，tcp其实早于udp出现，改用udp重新实现的话，会导致一大波人要修改自己的代码。不管多烂，只要做的人多了你都没办法大改
>
> ②首先UDP就八个字节，加了个校验码啥都没有了。所以没什么能被tcp利用的，那我不如在ip层自己封装

参考

如果用UDP实现TCP功能，需要实现以下事情：

1. 增加ack机制，确保能发送到对端
2. 增加seq机制，实现顺序化传输
3. 需要用队列实现缓冲区，主要是为了重传
4. 校验机制

问：多说一嘴，为啥现在很多都自己实现TCP协议？

答：因为慢启动和拥塞避免的缺点。

问：在哪一层进行封装？

答：在应用层。下一层的功能要用上一层来进行封装。

TCP网络编程的本质（三个半事件）

TCP网络编程最本质的是处理三个半事件：

1. 连接建立：包括服务器端被动接受连接（accept）和客户端主动发起连接（connect）。TCP连接一旦建立，客户端和服务端就是平等的，可以各自收发数据。
2. 连接断开：包括主动断开（close、shutdown）和被动断开（read()返回0）。
3. 消息到达：文件描述符可读。这是最为重要的一个事件，对它的处理方式决定了网络编程的风格（阻塞还是非阻塞，如何处理分包，应用层的缓冲如何设计等等）。
4. 消息发送完毕：这算半个。对于低流量的服务，可不必关心这个事件；另外，**这里的“发送完毕”是指数据写入操作系统缓冲区（内核缓冲区）**，将由TCP协议栈负责数据的发送与重传，不代表对方已经接收到数据。

正向代理和反向代理

参考链接

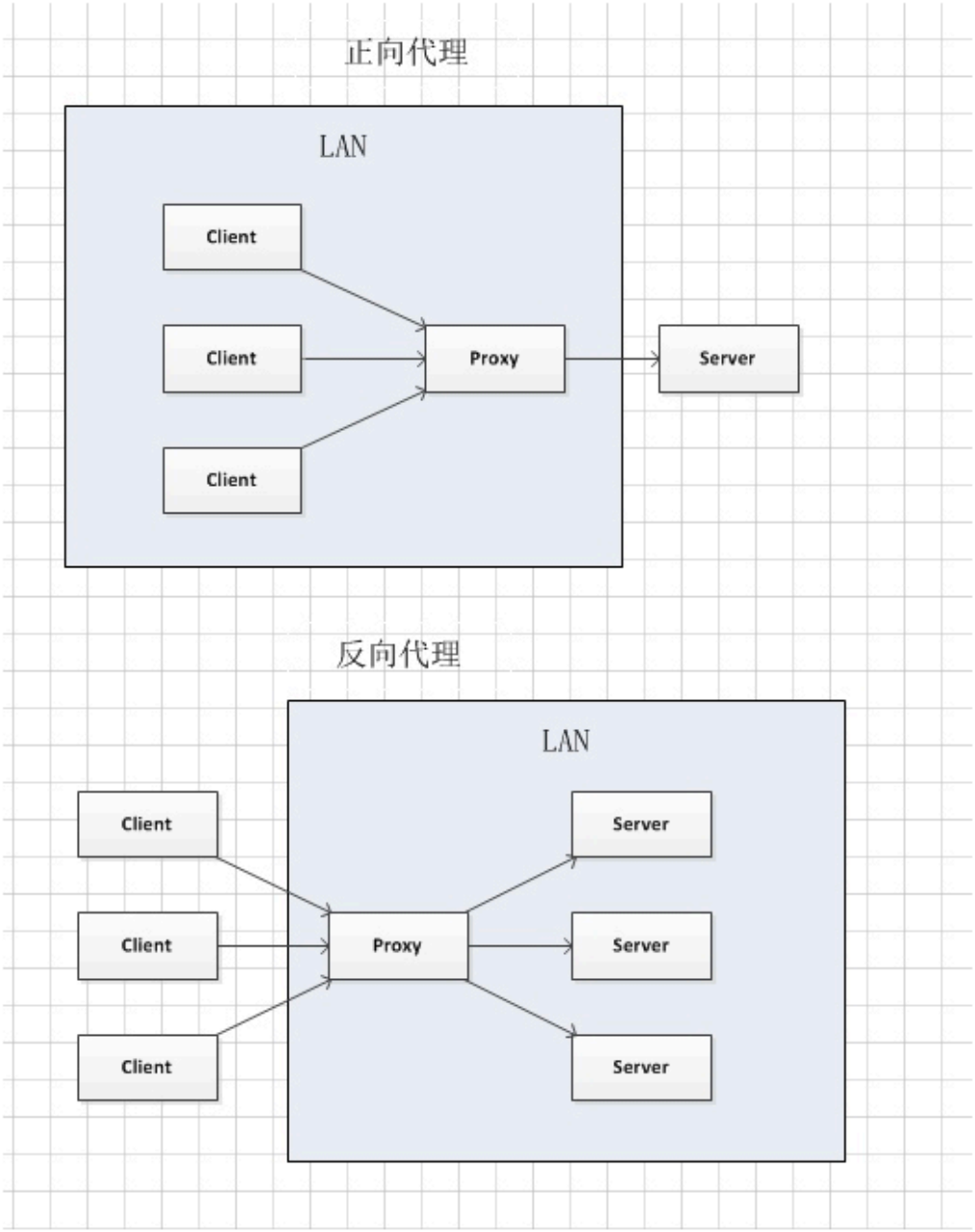
一张图就能说明问题

202

阿笠硕士

202 人赞同

上面的答案都说的很好，我画了张图能更直观地解释为何反向代理叫"反向"代理



正向代理中，proxy和client同属一个LAN，对server透明；
反向代理中，proxy和server同属一个LAN，对client透明。
实际上proxy在两种代理中做的事都是代为收发请求和响应，不过从结构上来看正好左右互换了下，所以把后出现的那种代理方式叫成了反向代理。

HTTP1.0和HTTP1.1的一些区别

HTTP1.0最早在网页中使用是在1996年，那个时候只是使用一些较为简单的网页上和网络请求上，而HTTP1.1则在1999年才开始广泛应用于现在的各大浏览器网络请求中，同时HTTP1.1也

是当前使用最为广泛的HTTP协议。主要区别主要体现在：

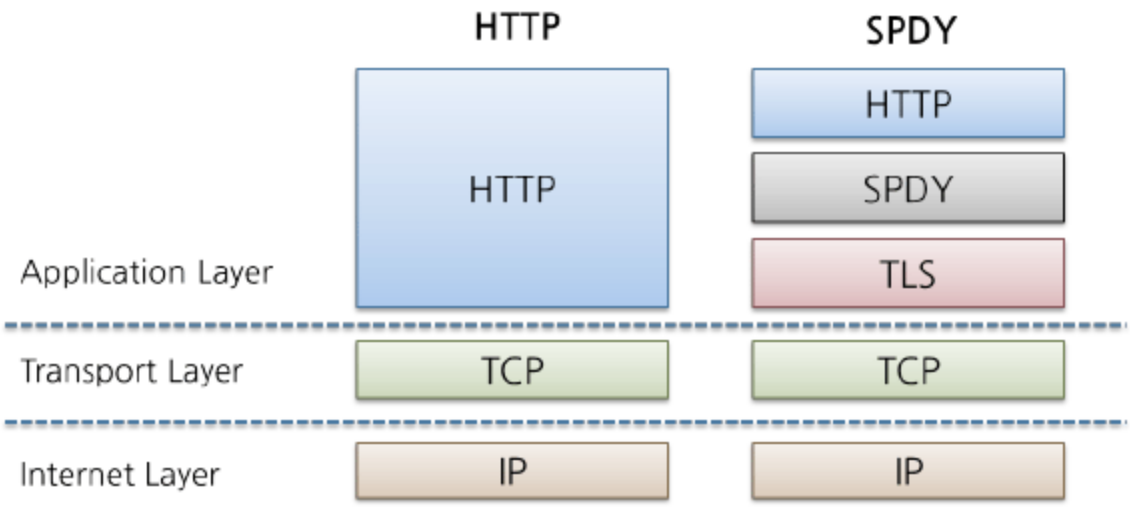
1. **缓存处理**，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
2. **带宽优化及网络连接的使用**，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
3. **错误通知的管理**，在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
4. **Host头处理**，在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。
5. **长连接**，HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

HTTP1.0和HTTP2.0有什么区别

背景

1989年作为万维网交流标准，属于应用层协议，在客户端和服务器端进行信息交换。这个过程中，客户端发送一个基于文本的请求到服务器通过调用GET 或者POST方法。 相应的，服务器返回资源给客户端。

http2.0起初是通过SPDY协议而来的，SPDY是一个由 Google 主导的研究项目发明的HTTP替代协议。Google为了减少加载延迟而开发的，通过使用例如压缩，多路复用，优化等。 SPDY位于HTTP之下、TCP/SSL之上，这样可以轻松兼容老版本的HTTP协议，同时可以使用已有的SSL功能。SPDY优化了HTTP1.X的请求延迟，解决了HTTP1.X的安全性



新特性1：二进制分帧

TCP连接相当于两根管道（一个用于服务器到客户端，一个用于客户端到服务器），管道里面数据传输是通过字节码传输，传输是有序的，每个字节都是一个一个来传输。例如客户端要向服务器发送Hello、World两个单词，只能是先发送Hello再发送World，没办法同时发送这两个单词。不然服务器收到的可能就是HWeolrllod（注意是穿插着发过去了，但是顺序还是不会乱）。但是能否同时发送Hello和World两个单词能，当然也是可以的，可以将数据拆成包，给每个包打上标签。发的时候是这样的①H ②W ①e ②o ①l ②r ①l ②l ①o ②d。这样到了服务器，服务器根据标签把两个单词区分开来。

二进制分帧层 在 应用层(HTTP/2)和传输层(TCP or UDP)之间。HTTP/2并没有去修改TCP协议而是尽可能的利用TCP的特性。在二进制分帧层中， HTTP/2 会将所有传输的信息分割为帧（frame）,并对它们采用二进制格式的编码。

> 二进制为何能提高数据传输效率：
>
> 1. 根据协议约定，省去参数名所占用的字节，缩减了数据。
> 2. 将数值类型的数据打包至相应范围内的二进制，节省了空间，4bytes能表示 32 位的文本数值，但文本数据值要 32bytes。
> 3. 在一定程度上可以起到加密数据的作用，如果第三方不知道数据协议，就没有办法截取相应的字节为获取数据

新特性2：多路复用技术

HTTP 性能优化的关键并不在于高带宽，而是低延迟。TCP 有个慢启动的特性，起初会限制连接的最大速度，如果数据成功传输，会随着时间的推移提高传输的速度。由于这种原因，让原本就具有突发性和短时性的 HTTP 连接变的十分低效。HTTP/2 通过让所有数据流共用同一个连接，可以更有效地使用 TCP 连接，让高带宽也能真正的服务于 HTTP 的性能提升。

新特性3：头压缩

在 HTTP/1 中，HTTP 请求和响应都是由「状态行、请求 / 响应头部、消息主体」三部分组成。一般而言，消息主体都会经过 gzip 压缩，或者本身传输的就是压缩过后的二进制文件（例如图片、音频），但状态行和头部却没有经过任何压缩，直接以纯文本传输。着 Web 功能越来越复杂，每个页面产生的请求数也越来越多，导致消耗在头部的流量越来越多，尤其是每次都要传输 UserAgent、Cookie 这类不会频繁变动的内容，完全是一种浪费。

> 压缩的原理:

> - 维护一份相同的静态表（Static Table），包含常见的头部名称，以及特别常见的头部名称与值的组合；
> - 维护一份相同的动态表（Dynamic Table），可以动态的添加内容；
> - 支持基于静态哈夫曼码表的哈夫曼编码（Huffman Coding）；

+++

传输层

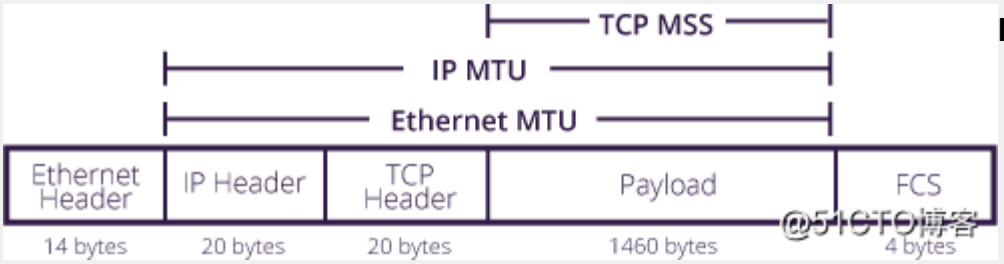
TCP 与 UDP 的区别

> 参考书籍笔记中linux高性能服务中第一个笔记

1. **连接**：
   TCP是面向连接的，传输数据前要先建立连接。
   UDP是无连接的
2. **可靠性**：
   通过TCP传输的数据无差错，不丢失，不重复，而且按序到达。
   UDP是尽最大努力交付，不保证数据到达后的可靠性。
3. **服务对象**：
   TCP链接是点到点服务，一条连接只能有两个端点。
   UDP可以一对一、一对多、多对多、多对一交互
4. **传输方式**：
   TCP是面向字节流的，但保证顺序和可靠。
   UDP面向报文，不会出现该问题、
5. **拥塞，流量控制**：
   UDP没有拥塞控制，导致网络出现拥塞时不会使得源主机发送数据速率降低
   TCP 有拥塞控制和流量控制机制，保证数据传输的安全性。
6. **首部开销**：
   TCP首部开销20字节，如果使用了「选项」字段则会变长的。
   UDP首部8字节并且是固定不变的，开销较小。
7. **分片不同**
   TCP 的数据大小如果大于 MSS 大小，则会在传输层进行分片，目标主机收到后，也同样在传输层组装 TCP 数据包，如果中途丢失了一个分片，只需要传输丢失的这

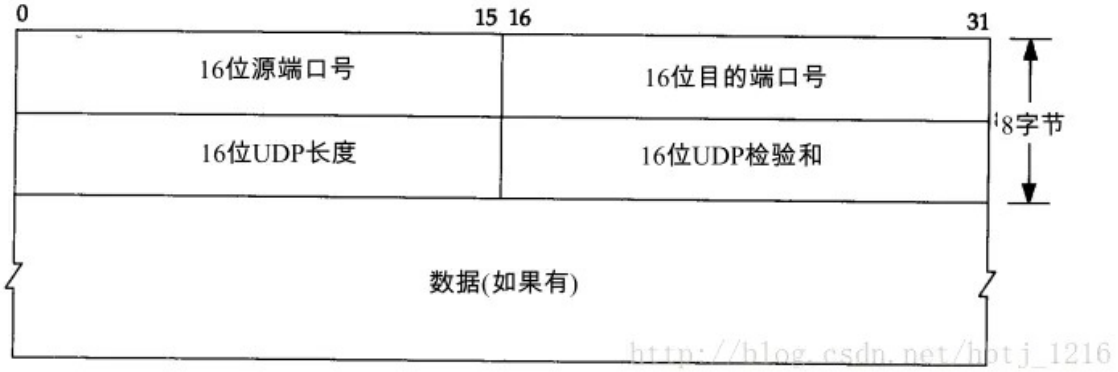
个分片。

UDP 的数据大小如果大于 MTU 大小，则会在 IP 层进行分片，目标主机收到后，在 IP 层组装完数据，接着再传给传输层，但是如果中途丢了一个分片，则就需要重传所有的数据包，这样传输效率非常差，所以通常 UDP 的报文应该小于 MTU。

> **MSS和MTU**
>
> 
>
>
> **MTU**：maximum transmission unit，最大传输单元，由硬件规定，如以太网的MTU为1500字节。主要在数据链路层
>
> **MSS**：maximum segment size，最大分节大小，为TCP数据包每次传输的最大数据分段大小，一般由发送端向对端TCP通知对端在每个分节中能发送的最大TCP数据。MSS值为MTU值减去IPv4 Header（20 Byte）和TCP header（20 Byte）得到。

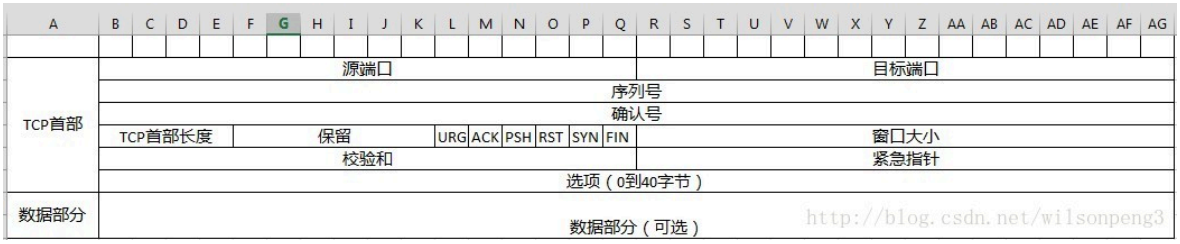
UDP 、TCP 首部格式

- UDP首部格式



  i. 源端口号。需要对方回信时候可以用，不需要回复可以设置为0
  ii. 目的端口。必须有，交付报文要用到。
  iii. 长度。表示UDP首部和UDP数据的长度和，最小为8个字节。因为首部就8个字节
  iv. 校验和。发送端计算，接收端验证，为了发现在数据收发间有无改动

- TCP首部格式



i. 源端口（16位）和目的端口（16位）
ii. 32位序列号。表示当前tcp数据第一个字节在整个字节流中的相对位置。一次TCP通信中某一个方向上的字节流的编号，第一个报文的序号值被系统初始化为某个随机值ISN，后序的TCP报文的序号值被设置成ISN+报文所携带数据的第一个字节序号。比如某个报文要传输的数据时字节流中的1025-2048字节，序号就是ISN+1025。
iii. 32位确认号。用来响应收到的TCP报文段。ack就是收到的32位序列号+1.
iv. 4位头部长度。表示TCP头部由多少个4字节组成。因为最多能表示15，所以15*4=60。
v. 标志位(6位)
    - URG--表示本报文段的紧急指针是否有效。
    - ACK--确认标志，表示确认号是否有效。携带ack标志的报文段是确认报文段。
    - PSH--表示接收端应用程序是否应该立即从TCP缓存中读出该数据。如果应用程序不读走，就会一直在停留在TCP的接收缓冲区中。
    - SYN--同步标志，用于数据同步，有syn的报文段成为同步报文段。
    - FIN--表示数据是否发送完毕。FIN=1表示数据发送完毕，可以释放连接。有fin标志的成为结束报文段。
    - RST--该位置为1时表示tcp连接中出现异常必须强制断开
vi. 16位窗口大小。指的是通告窗口(Receiver Window, RWND)。这个用来告诉对方自己的接收缓冲区还能容纳多少数据，这样可以控制对方的发送速度。
vii. 16位校验和。检验数据的正确性，保证可靠性。不仅校验头部，也校验数据部分
viii. 16位紧急指针（2字节）。标记紧急字段在数据中的位置
ix. 以上一共20字节，还有40字节可选用字段。
x. 数据部分

TCP的特点

TCP 是**面向连接的、可靠的、基于字节流**的传输层通信协议。

面向连接：一定是「一对一」才能连接，不能像 UDP 协议可以一个主机同时向多个主机发送消息，也就是一对多是无法做到的；

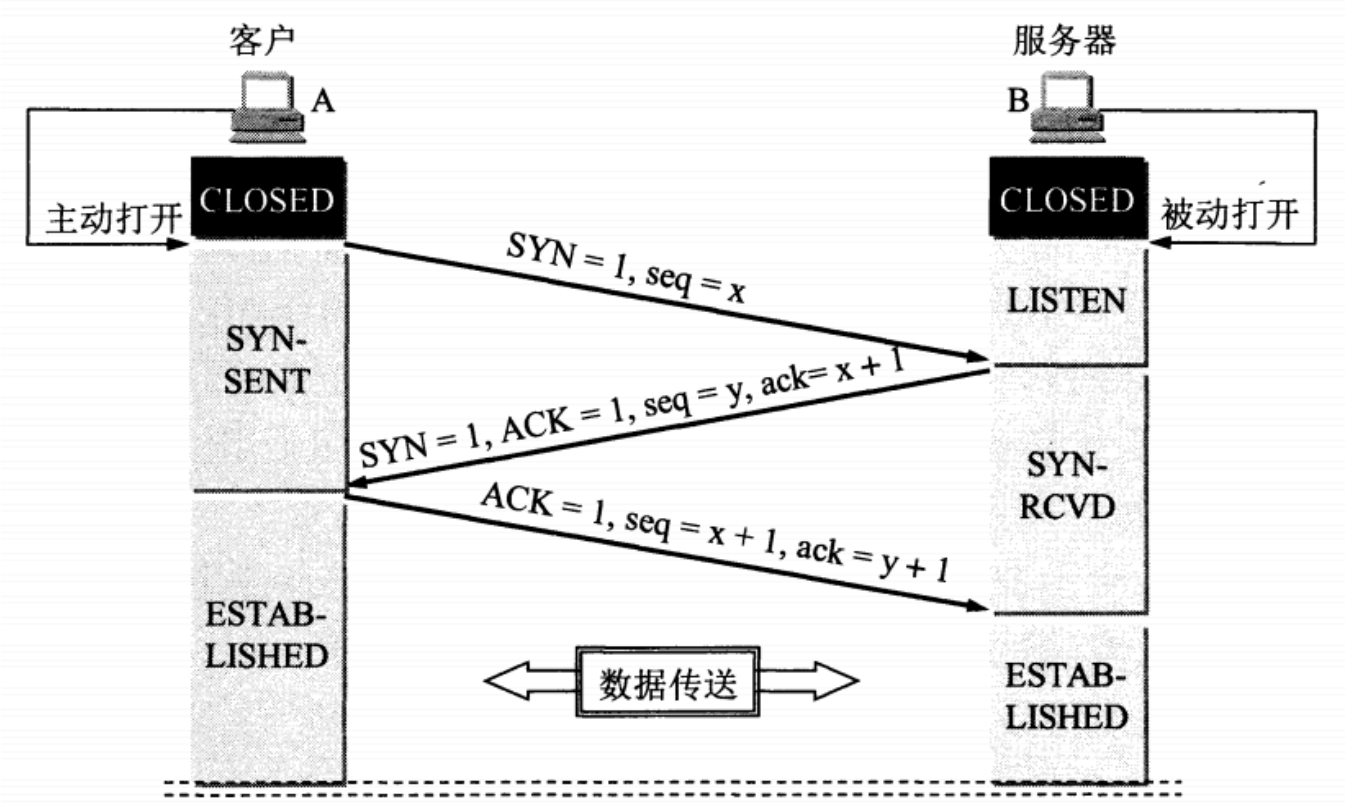
可靠的：无论的网络链路中出现了怎样的链路变化，TCP 都可以保证一个报文一定能够到达接收端；

字节流：(无边界+有序的概念)消息是「没有边界」的，所以无论我们消息有多大都可以进行传输。并且消息是「有序的」，当「前一个」消息没有收到的时候，即使它先收到了后面的字节，那么也不能扔给应用层去处理，同时对「重复」的报文会自动丢弃。

TCP四元组可以唯一的确定一个连接。所以有一个问题比如：有一个 IP 的服务器监听了一个端口，它的 TCP 的最大连接数是多少？

首先服务器端地址确定了，源端口确定了，那么只有客户端的IP数和目标端口不确定，那不就是 $2^{32} \times 2^{16} = 2^{48}$。当然实际上并发的TCP连接数肯定达不到这么多：①**文件描述符限制**，Socket 都是文件，所以首先要通过 `ulimit` 配置文件描述符的数目；②**内存限制**，每个 TCP 连接都要占用一定内存，操作系统的内存是有限的。

☆TCP三次握手



注意该图客户端课服务器端分别所处于的阶段，客户端由两个阶段，服务器方有三个。

- 开始建立连接。服务器主动监听某个端口，处于`LISTEN`状态。
- 第一次握手
  客户端会随机初始化一个序号（ client_isn ），将此序号置于 TCP 首部的「序列号」字段中，同时把 SYN 标志位置为 1，表示这是一个 SYN 报文。接着把第一个 SYN 报文发送给服务端，表示向服务端发起连接，该报文不包含应用层数据，之后客户端处于 SYN-SENT 状态。
- 第二次握手
  服务端收到客户端的 SYN 报文后，首先服务端也随机初始化自己的序列号（ server_isn ），将此序号填入TCP 首部的「序列号」字段中，其次把 TCP 首部的「确认应答号」字段填入 client_isn + 1 , 接着把 SYN和 ACK 标志位置为 1 。最后把该报文发给客户端，该报文也不包含应用层数据，之后服务端处于SYNRCVD 状态。

  > 其实第二次握手细分的话包含两次，可以想一下。首先是服务器端收到了客户端的同步报文，然后发送一个ACK确认报文，ack=clien_isn+1，ACK标志位置为1。然后在发送一个同步报文段，FIN标志位置为1，随机初始化一个属于自

> 己的序列号server_isn。这两部可以合并为1步发送而已

- 第三次握手
  客户端收到服务端报文后，还要向服务端回应最后一个应答报文，首先该应答报文 TCP 首部 ACK 标志位置为 1，其次「确认应答号」字段填入 server_isn + 1，最后把报文发送给服务端，**这次报文可以携带客户到服务器的数据**，之后客户端处于 ESTABLISHED 状态。等到服务器收到客户端的应答报文后，也进入 ESTABLISHED 状态。

所以从上面可以看到，三次握手的前两次是不能携带数据的，只有第三次可以携带数据。因为经过两次握手，客户端收到ack确认报文段之后就已经建立了连接，表明服务端的接受和发送能力是正常的，当然可以传输局。等到服务器收到来自客户端你的确认报文后，也表明客户端的接受和发送能力是正常的，就可以进行正常的全双工通信了。

为什么是三次握手

TCP最重要的就是序列号这个东西

之所以是三次握手的原因有三个，我想从大局上说一下为什么是三次，不是细致的解释。从客户端发起FIN同步报文段开始，其实是四段。TCP是一个全双工的通信方式，因此可以理解为两条单向通道。那么客户端发送FIN报文段给服务器端，服务器端发送ACK报文段这一个过程是一个单向通道的建立过程，表明服务器是完好的，可以收发数据。然后服务器到客户端这条通道同理也需要相同的过程，服务器发送FIN同步报文段给客户端，客户端返回一个ACK确认帧给服务器，这样一个全双工的通道就建立好了。而服务器发送的ACK确认帧和服务器发送的FIN同步报文段完全可以放在一起，不用分两次发送，所以就造成了三次握手。

然后在分析一下三次握手的原因或者说好处：

1. 阻止重复的历史连接
2. 同步双方的初始序列号
3. 避免资源浪费

- **避免历史连接**
  这个是RFC 793文档中说明的三次握手的主要原因。就是为了防止旧的重复连接初始化造成混乱。
  当网络拥堵的情况下，三次握手时一个客户端连续多次发送SYN建立连接的报文，造成一个旧的SNY报文比新的SYN早到服务器端。服务器端会回复一个SYN+ACK

报文给客户端，客户端收到后会根据自身的上下文判断这是否是一个历史连接，如果是历史连接则发送一个RST报文给服务器端表示终止。
比如SYN1的ISN=90，SYN2的ISN=100，那么如果服务器端先收到SYN1的话，发送过来的ACK=90+1，但是客户端期望收到ACK=100+1，所以客户端知道这个给我回复的ACK报文段是历史连接的，我必须发送一个RST报文告诉服务器端终止由这个SYN90报文引起的连接。等SYN100到了我就发送ACK确认报文给服务器端，从而建立连接。

- **同步双方序列号**
  要记住“序列号”这个东西是TCP保持可靠顺序传输的关键。通过序列号这个东西接收方可以做三件事：
  i. 去除重复连接
  ii. 根据数据报的序列号按序接收
  iii. 可以标识发出去的数据包有哪些被接受了
  这就是我上面说的双方同步序列号其实是四次握手，但是二三次握手可以优化成一步，也就是三次握手了。而两次握手只能保证一方的序列号成功被对方接受，而不能保证双方序列号都被互相接受。
- **避免资源浪费**
  在多次行不行，当然行，但是四次已经优化成三次了，再多的话就浪费没必要了。

初始序列号（ISN）

`ISN` 全称是 `Initial Sequence Number`，是 TCP 发送方的字节数据编号的原点，告诉对方我要开始发送数据的初始化序列号。

ISN不是固定的， 如果是固定的，攻击者很容易猜出后续的确认序号，为了安全起见，避免被第三方猜到从而发送伪造的 `RST` 报文，因此 ISN 是动态生成的

> - 为什么ISN是动态生成的？
>   i. 通过SYN来判断这个连接是失效的还是有效的，不然你无法判断会出现错乱。所以每次连接前重新初始化一个序列号就是为了能够使得通信双反根据序号将不属于本次连接的报文段丢弃掉
>   ii. 安全性，防止黑客伪造相同序号的TCP报文被对方接受
> - 为什么客户端和服务端的初始序列号 ISN 是不相同的？
>   因为相同的话就可动态生成违背了啊，前面说了ISN只能动态生成，你还让客户端和服务端相同，那不得提前协商？还没建立连接了在提前协商？

- 初始序列号 ISN 是如何随机产生的？
  $ISN = M + F(localhost, localport, remotehost, remoteport)$
  M 是一个计时器，这个计时器每隔 4 毫秒加 1。
  是一个 Hash 算法，根据源 IP、目的 IP、源端口、目的端口生成一个随机数值。 MD5 ,SHA1都可以

既然 IP 层会分片，为什么 TCP 层还需要 MSS 呢？

关于什么是MSS，什么是MTU在[TCP 与 UDP 的区别](#TCP 与 UDP 的区别)中有详细说明

当 IP 层有一个超过 MTU大小的数据（TCP 头部 + TCP 数据）要发送，那么 IP 层就要进行分片，保证每一个分片都小于 MTU。把一份 IP 数据报进行分片以后，由目标主机的 IP 层来进行重新组装后再交给上一层 TCP 传输层。**那么当如果一个 IP 分片丢失，整个 IP 报文的所有分片都得重传**。这是因为IP层本身没有重传机制，他只是高效的发送数据而已，没有协议来保证完整性。当接收方TCP报文不完整后不会响应ACK的，那么TCP发送方就会触发超时重传，重发整个TCP报文段。这样做就非常没有效率可言。

所以为了提高效率，在建立连接的时候会剔除IPHeader和TCPHeader，剩下的就是协商的MSS值，当TCP发现超过MSS后就会分片，这样形成的IP包长度也就不会大于MTU了，也就不用IP分片了。

半连接队列(SYN队列)和全连接队列(accept队列)

当服务器第一次收到客户端的 `SYN` 之后，就会处于 `SYN_RCVD` 状态，此时双方还没有完全建立连接。服务器会把这种状态下请求连接放在一个队列里，我们把这种队列称之为半连接队列也叫做SYN队列

在第三次握手后，服务端收到ACK确认报文后，从SYN队列中移除放到accept队列中，通过调用accept() socketAPI 从accept队列中取出连接

队列满了就有可能会出现丢包现象。

- 如何增大半连接队列？
  首先 `max_qlen_log` 表示半连接队列的值
  如果 `max_sys_backlog > min(somaxconn, backing)`，
  则 `max_qlen_log = min(somaxconn, backing)*2`

如果`max_sys_backlog <= min(somaxconn, backing)`，
则`max_qlen_log = max_sys_backlog*2`

- 如何增大全连接队列
  TCP 全连接队列的最大值取决于 somaxconn 和backlog之间的最小值，也就是 min(somaxconn, backlog)。
  `somaxconn`是 Linux 内核的参数，默认值是 128，可以通过`/proc/sys/net/core/somaxconn`来设置其值；
  `backlog`是listen(int sockfd, int backlog) 函数中的 backlog大小，Nginx 默认值是 511，可以通过修改配置文件设置其长度；

什么是 SYN 攻击？如何避免 SYN 攻击？

定义：假设攻击者短时间伪造不同 IP 地址的 SYN 报文，服务端每接收到一个 SYN 报文，就进入SYN_RCVD 状态，但是服务器发送的ACK+SYN无法得到应答， 久而久之服务器端的SYN接收，即半连接队列就会被占满，正常用户发送的STN报文无法被存储，服务器无法提供其他服务。

如何避免：

1. 通过修改 Linux 内核参数，控制队列大小和当队列满时应做什么处理。扩大队列大小，队列满时对新的SYN报文直接回复RST，然后丢弃掉。这样治标不治本吧
2. 当SYN队列满了之后，服务器如果再次收到SYN同步报文，不将该连接放入同步队列。服务器会计算出一个cookie值，以SYN+ACK中的序列号返回给客户端。当服务端再次收到客户端的ACK时会检查其合法性，如果合法再放到ACCEPT队列中

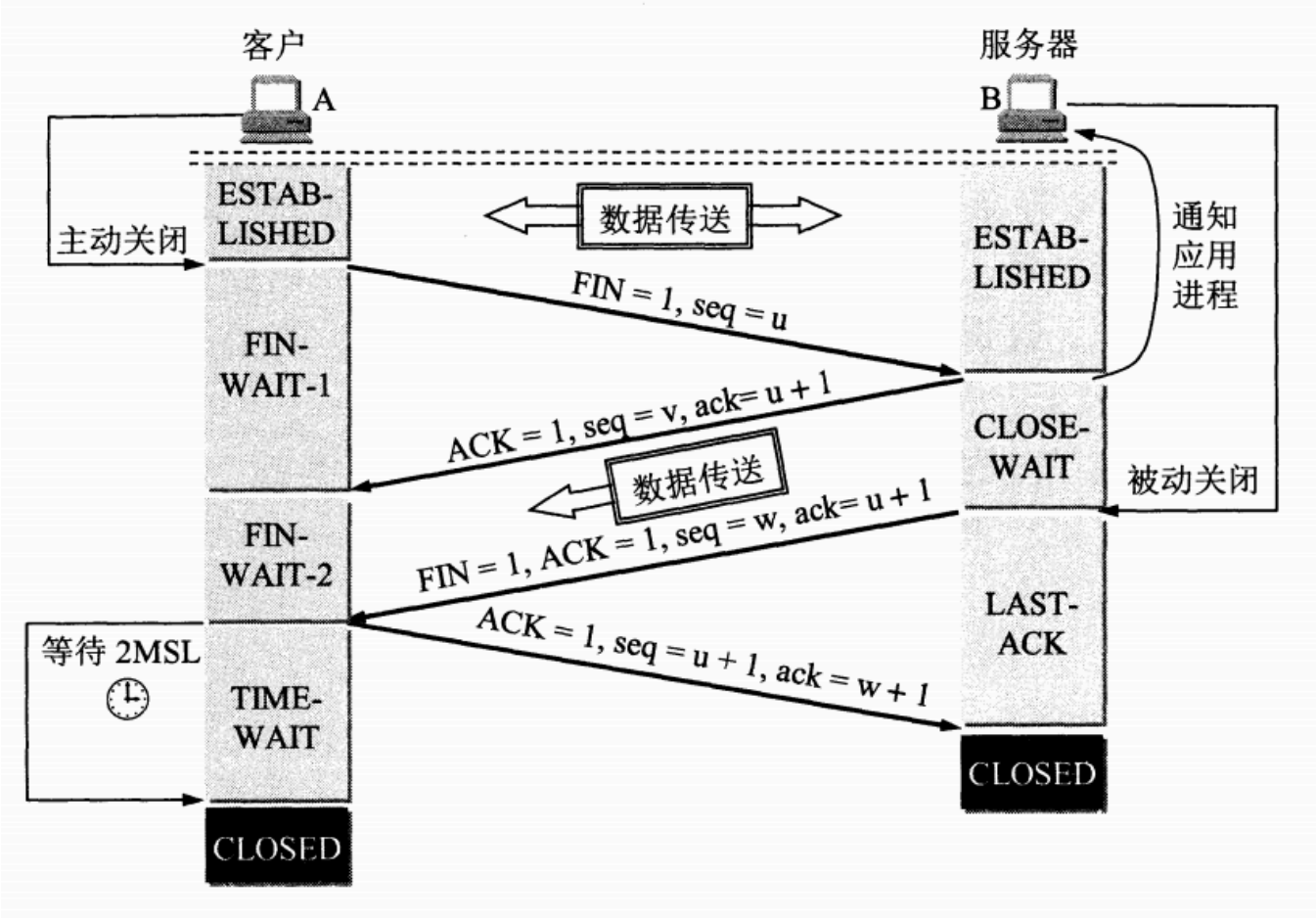
三次握手中可以携带数据吗

第一次、第二次握手不可以携带数据，而第三次握手是可以携带数据的。

假如第一次握手可以携带数据的话，如果有人要恶意攻击服务器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据，疯狂着重复发 SYN 报文，这会让服务器花费大量的内存空间来缓存这些报文，这样服务器就更容易被攻击了。

对于第三次握手，此时客户端已经处于连接状态，他已经知道服务器的接收、发送能力是正常的了，所以可以携带数据是情理之中。

☆TCP四次挥手



TCP是全双工的通信，因此收发双方都各有一个写缓存和读缓存。

- 第一次握手
  客户端打算关闭连接，此时会发送一个 TCP首部 FIN 标志位被置为 1 的报文，也即 FIN 报文，之后客户端进入FIN_WAIT_1 状态。
- 第二次握手
  服务端收到该报文后，就向客户端发送 `ACK` 应答报文，接着服务端进入 `CLOSED_WAIT` 状态。客户端收到服务端的 `ACK` 应答报文后，之后进入 `FIN_WAIT_2` 状态。
- 第三次握手
  等待服务端处理完数据后，也向客户端发送 `FIN` 报文，之后服务端进入 `LAST_ACK` 状态。客户端收到服务端的 `FIN` 报文后，回一个 `ACK` 应答报文，之后进入 `TIME_WAIT` 状态
- 第四次握手
  服务器收到了 `ACK` 应答报文后，就进入了 `CLOSED` 状态，至此服务端已经完成连接

的关闭。客户端在经过 `2MSL` 一段时间后，自动进入 `CLOSED` 状态，至此客户端也完成连接的关闭。

为什么需要四次挥手？

我一直把TCP之间的通信看两条管道组成的全双工通信。我们来看这种抽象的比喻，如果要关闭这两条管道的通信要几个步骤？很明显是四个，首先A端告诉B端我要关了，B端收到后说好的。那么A到B端的就关掉了。然后B端告诉A端我这边也要关掉了，A端说好的，那么B到A的也就关掉了，你看，这也需要四次。

那么回到TCP中来，首先第一次客户端向服务器端发送FIN通知服务器端我这边要断掉了，不给你发数据了，这一次是必然的。还有最后一次，客户端给服务器端恢复一个ACK表示你关吧我知道了。这两次是必须的。那么第二次和第三次我们看一下。由于服务器端要处理一些数据然后发送给客户端，所以第二次和第三次是无法合并到一起的。所以当服务器端收到客户端的FIN报文段后，必须马上回一个ACK确认报文表示可以关，但此时可能服务器端有一些数据需要处理，所以说等处理完数据我再发一个FIN关闭报文给客户端告诉客户端我这边也要关闭了。

TIME_WAIT状态

客户端收到了来自服务器的带有FIN、ACK的结束报文段后并没有直接关闭，而是进入了TIME_WAIT状态。

> 在这个状态中客户端只有等待2MSL(Maximum Segment Life，报文段最大生存时间)才能完全关闭。它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃，RFC文档的建议值是2分钟。为啥有这个，因为TCP 报文基于是 IP 协议的，而 IP 头中有一个 `TTL` 字段，是 IP 数据报可以经过的最大路由数，每经过一个处理他的路由器此值就减 1，当此值为 0 则数据报将被丢弃，同时发送 ICMP 报文通知源主机。MSL 与 TTL 的区别：MSL 的单位是时间，而 TTL 是经过路由跳数。所以 **MSL 应该要大于等于 TTL 消耗为 0 的时间**，以确保报文已被自然消亡。

- **为什么 TIME_WAIT 等待的时间是 2MSL？以及为什么被动关闭以防不需要？**
  答：通信要解决的首要问题是同步，即双方处于信息对称的情况下。我们来看最后两次挥手，当服务器端发送FIN给客户端的时候，服务端是不能当方面释放掉连接的，因为他不知道客户端收没收到自己的FIN，所以服务器必须要等待客户端发送的ACK过来，才能安全的释放TCP连接所占用的内存资源，端口号。所以服务器作为被动关闭的一方，是无需任何TIME_WAIT状态的。服务器收到ACK就表明我已经知

道客户端知道我要关闭连接了，放心关闭

那么此时看客户端，他发送ACK给服务器端后，客户端并不知道服务端有没有收到这个报文，客户端会这么想：

i. 服务器没收到ACK，我就等着超时重传

ii. 服务器收到自己的ACK了，也不会发消息

可以发现，不管上面哪一种情况，客户端都必须要等带，而且要取最大值以应对最坏的情况发生，**这个最坏情况就是①客户端发送ACK到服务器端的报文最大存活时间+②服务器端从新发送FIN到客户端的报文最大存活时间即2MSL。**

- **为什么需要TIME_WAIT？如果没有TIME_WAIT或者TIME_WAIT很短怎么办？**

  首先要明白，只有主动发起关闭的一方才有TIME_WAIT状态。

  - 防止具有相同「四元组」的「旧」数据包被收到

    假如在客户端向服务端发送FIN即第一次挥手之前，服务器端发送了一个报文给客户端，但因为网络原因这个报文延迟到达了，那么如果TIME_WAIT没有或者太短，由于这个报文具有相同的四元组的旧报文，可能会和新TCP连接的新报文起冲突。
  - 保证连接正确关闭

    即为什么需要2MSL的原因，上面有不再说了。
- **TIME_WAIT过多有什么危害？**
  - 占用内存资源
  - 占用端口的资源

    由于客户端使用系统自动分配的临时端口号来建立连接，所以一般不用考虑这个问题。但是服务器可能要考虑，因为服务器的端口号一般是固定的。
- **如何优化TIME_WAIT？**
  - Linux内核中有一个默认值18000，当系统中的TIME_WAIT一旦超过这个值，系统就会将后面TIME_WAIT连接状态重置。
- **服务端出现大量TIME_WAIT的原因**

  先来说一说长连接和短连接，在HTTP1.1协议中，有个 Connection 头，Connection有两个值，close和keep-alive，这个头就相当于客户端告诉服务端，服务端你执行完成请求之后，是关闭连接还是保持连接。如果服务器使用的短连接，那么每次客户端请求后，服务器都会主动发送FIN关闭连接。最后进入time_wait状态。可想而知，对于访问量大的Web Server，会存在大量的TIME_WAIT状态。让服务器能够快速回收和重用那些TIME_WAIT的资源，可以修改内核参数。
- **解决办法**

  i. 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

ii. 服务端：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）

建立连接但是客户端出现故障怎么办？

TCP的保活机制：当在某个时间段内没有任何连接相关的活动，该机制就会每隔一个时间间隔发送一个探测报文，该报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前连接已经死亡，内核会通知应用程序然后中断连接

Linux的默认值如下：

1. net.ipv4.tcp_keepalive_time=7200 //保活时间
2. net.ipv4.tcp_keepalive_intvl=75 //每次检测间隔时间
3. net.ipv4.tcp_keepalive_probes=9 //探测次数，9此后中断连接

有三种情况需要注意一下：

1. 对端程序是正常工作的。当 TCP 保活的探测报文发送给对端, 对端会正常响应，这样 **TCP 保活时间会被重置**，等待下一个 TCP 保活时间的到来。
2. 对端程序崩溃并重启。当 TCP 保活的探测报文发送给对端后，对端是可以响应的，但由于没有该连接的有效信息，**会产生一个 RST 报文**，这样很快就会发现 TCP 连接已经被重置
3. 对端程序崩溃没有重启，或者报文不可达，达到保活探测的最大次数之后会报告该连接已经死亡。

TCP 黏包问题

原因：

1. 本质原因：由于TCP是面向字节流传输的，因此数据没有确切的边界，会出现前后两个数据包黏在一起的情况。
2. 发送方。TCP默认使用nagle算法，为了减少网络中报文段的数量。主要有两步，首先上一个分组得到确认后才能发送下一个分组。其次，收集多个小分组然后一起发送。

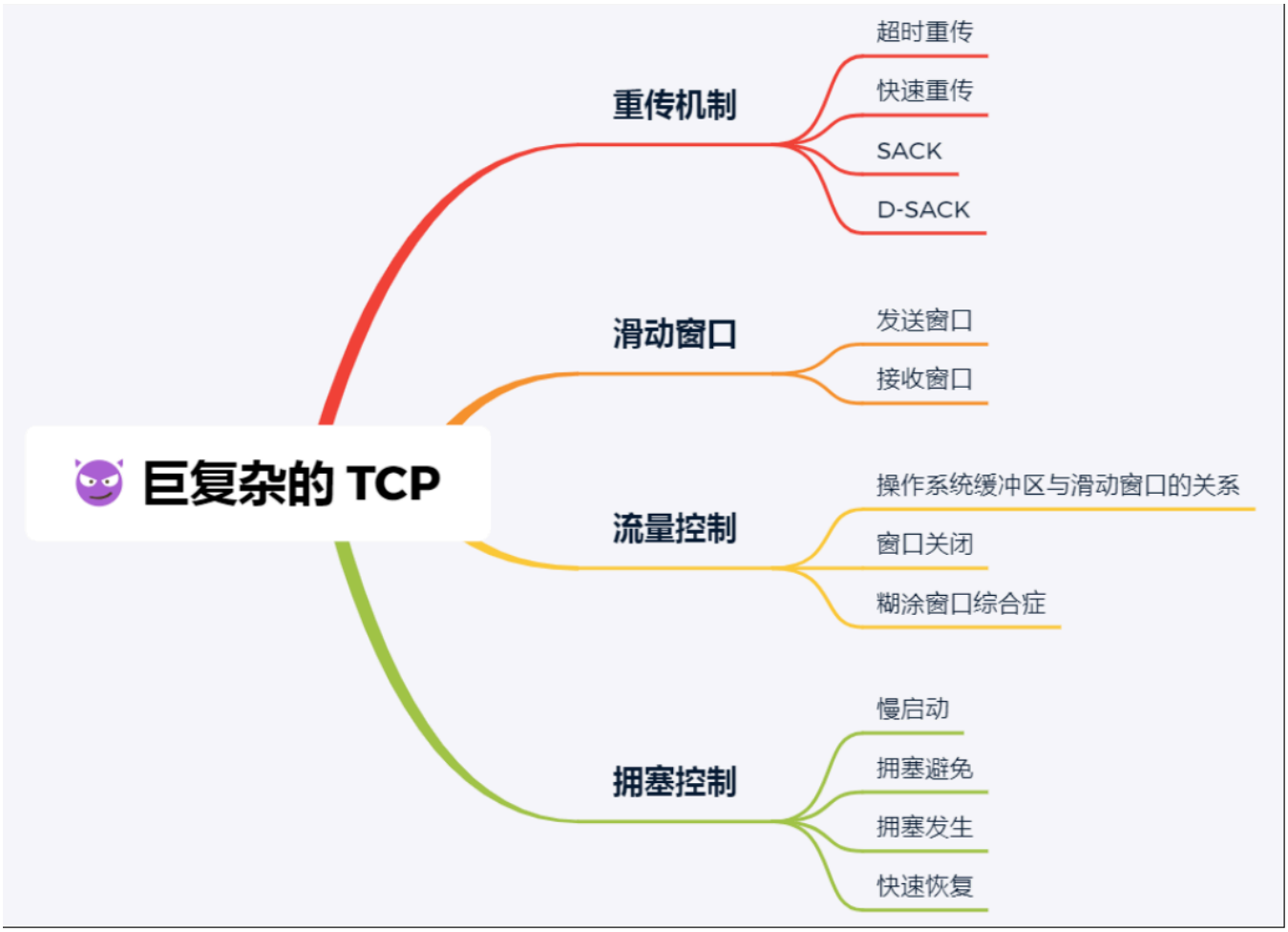
3. 接收方。当接收方收到数据后不会马上交到应用层去处理，因为发送方和接收方各有两个缓冲空间。接收方收到数据后会将数据放到接收方的读缓存中。这样如果读取的速度小于接收的速度，应用程序可能会读到多个首尾相连的包。

解决方案：

1. 发送方。关闭nagle算法就行+
2. 接收方。没办法交给应用层
3. 应用层。粘包的原因是不确定消息的边界。所以只要能识别消息边界就行。比如加入特殊标志，头标志消息尾标志这样，或者加入每段消息的长度

TCP 协议保证可靠传输的手段

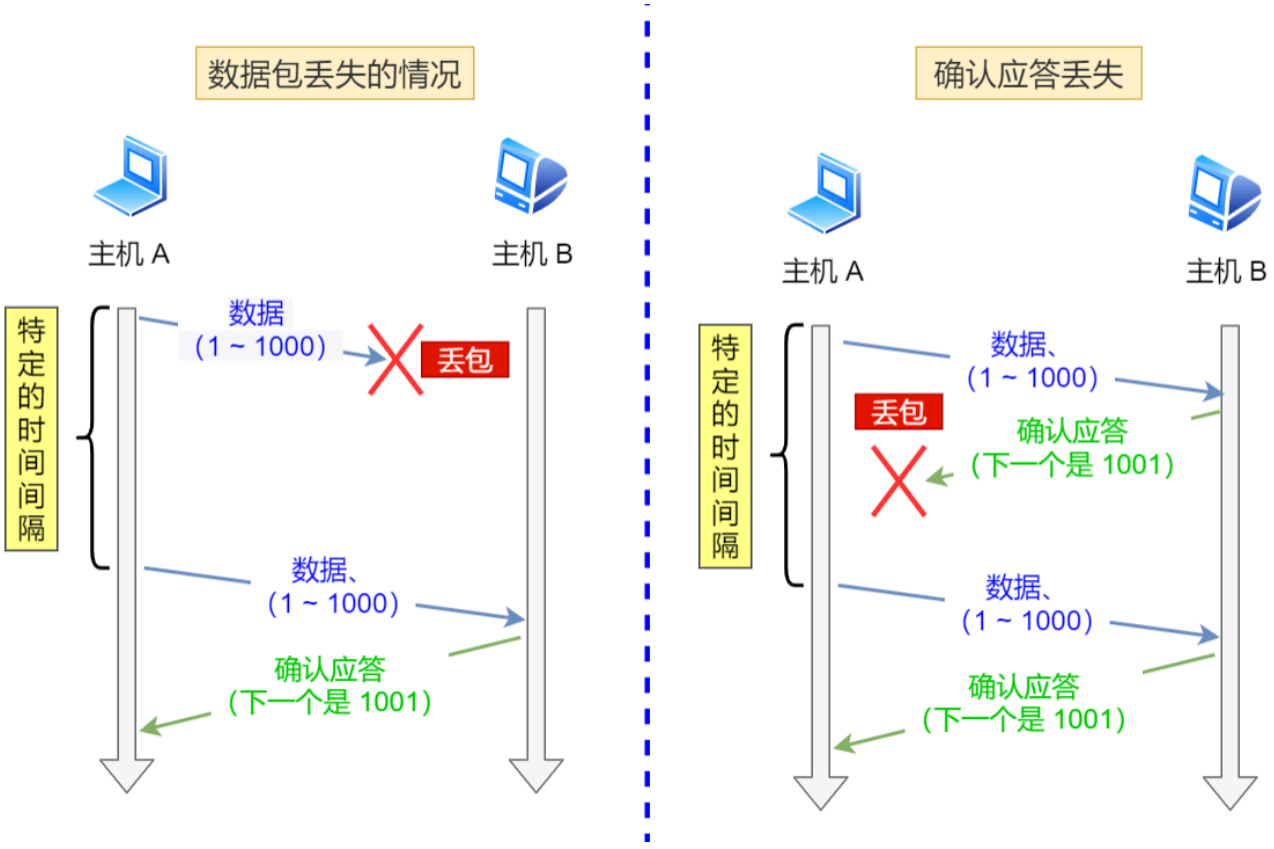
- 三次握手，建立可信的传输信道
- 校验和，接收端可以检测出来数据是否异常。
- 流量控制
- 拥塞控制
- 停止等待ARQ协议
  它的基本原理就是每发完一个分组就停止发送，等待对方确认（回复ACK）。如果过了一段时间（超时时间后），还是没有收到 ACK 确认，说明没有发送成功，需要重新发送，直到收到确认后再发下一个分组。在停止等待协议中，若接收方收到重复分组，就丢弃该分组，但同时还要发送确认。
- 重传机制



重传机制

超时重传

超时重传指的是发送数据包在一定的时间周期内没有收到相应的ACK，等待一定的时间，超时之后就认为这个数据包丢失，就会重新发送。这个等待时间被称为RTO.



检测丢失segment的方法从概念上讲还是比较简单的，每一次开始发送一个TCP segment的时候，就启动重传定时器，定时器的时间一开始是一个预设的值（Linux 规定为1s），随着通讯的变化以及时间的推移，这个定时器的溢出值是不断的在变化的，有相关算法计算RTO，如果在ACK收到之前，定时器到期，协议栈就会认为这个片段被丢失，重新传送数据。

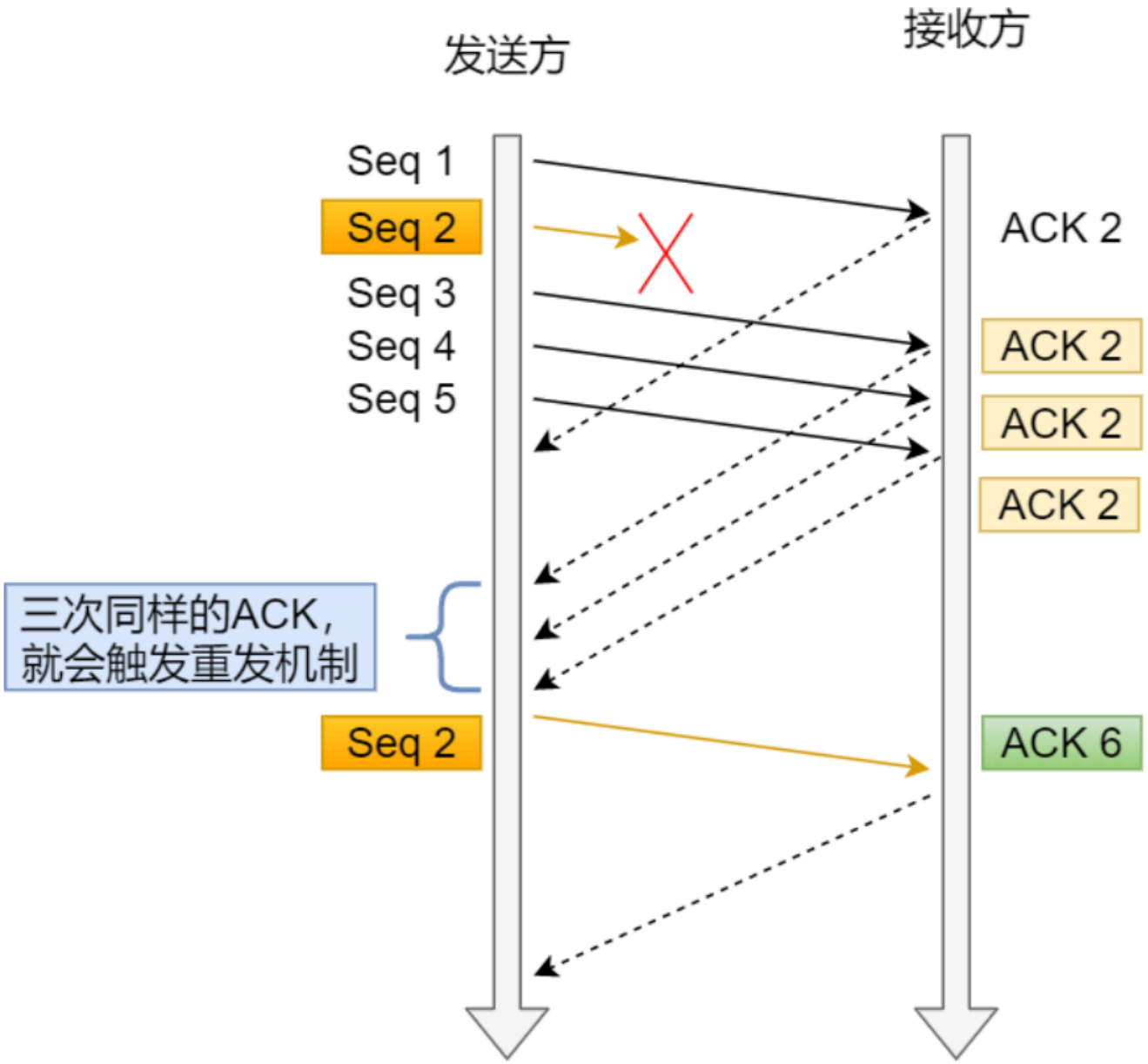
TCP在实现重传机制的时候，需要保证能够在同一时刻有效的处理多个没有被确认的ACK，也保证在合适的时候对每一个片段进行重传，有这样几点原则：

1. 这些被发送的片段放在一个窗口中，等待被确认，没有确认不会从窗口中移走，定时器在重传时间到期内，每个片段的位置不变，这个地方其实在滑动窗口的时候也有提到过
2. 只有等到ACK收到的时候，变成发送并ACK的片段，才会被从窗口中移走。
3. 如果定时器到期没有收到对应ACK，就重传这个TCP segment

超时时间应该率大于报文往返RTT的值，RTT即网络从一段传送到另一端所需的时间。由于网络是在时常变换，因此RTT的值也经常波动。如果超时重发的数据又再次超时的时候，又需要重传，那么这个超时时间间隔加倍。当两次超时时，就说明网络环境不行，不适合频繁反复发送数据。超时重传的问题是超时时期可能比较长，有没有更快的方式呢？有的，就是下面的快速重传，用来解决超时重发的时间等待问题。

快速重传

看一张图你就明白了什么叫做快速重传。



当seq1发送后被接收方收到，接收方会返回一个ACK2和一个同步报文段。但是seq2没有到达接收方，则后面不论接收方收到了那个数据都是回ACk2，当发送方连续收到了三个ACK2就表明seq2没有被接收方收到，就会重发。但是问题在于通过重复确认机制，我们每次只能传送一个包，如果有多个包丢失，在传送过程中可能会造成timeout，造成带宽利用率下降

SACK

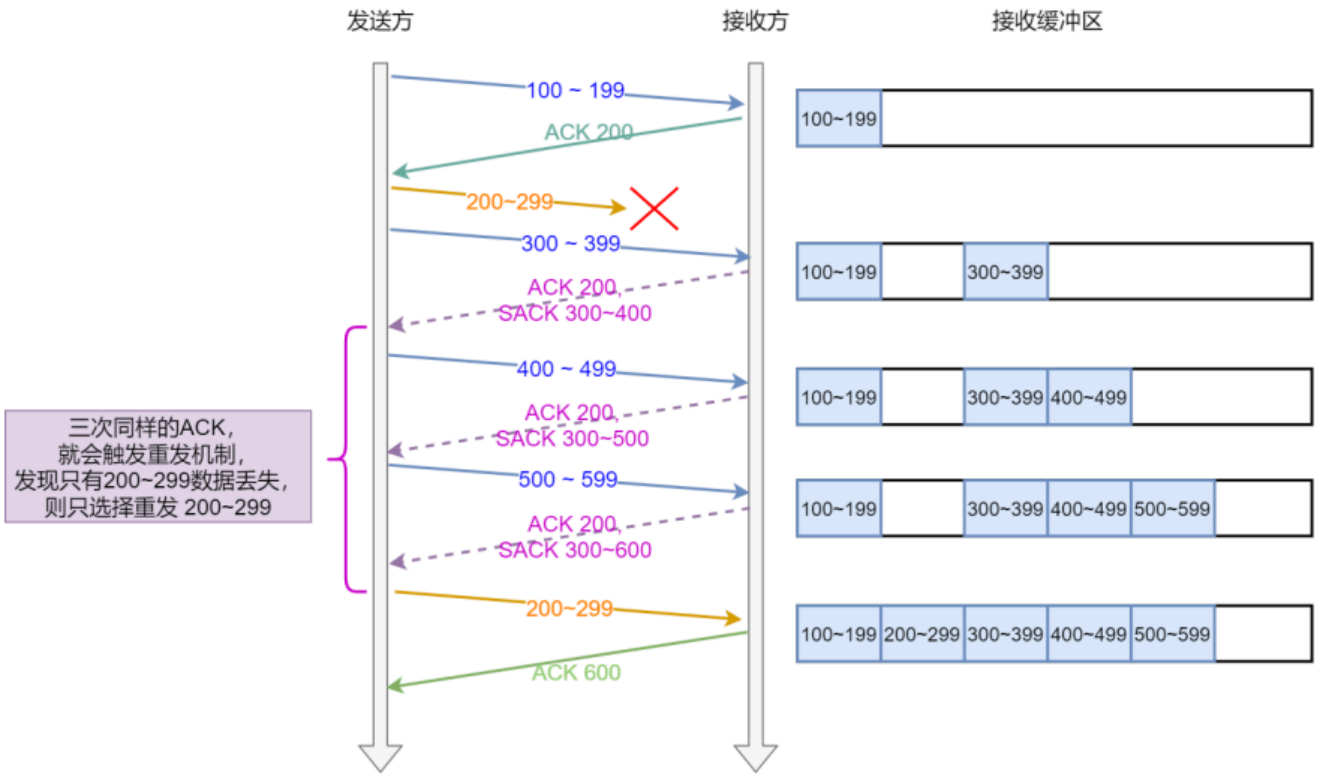
SACK是一个TCP的选项，来允许TCP单独确认非连续的片段，用于告知真正丢失的包，只重传

丢失的片段。

超时重传和快速重传的困境：快速重传和超时重传都会面临到一个重传什么包的问题，因为发送端也不清楚丢失包后面传送的数据是否有成功的送到。主要原因还是对于TCP的确认系统，不是特别的好处理这种不连续确认的状况了，只有低于ACK number的片段都被收到才有进行ACK，out-of-order的片段只能是等待，同时，这个时间窗口是无法向右移动的。

> 举个例子：服务器发送4个片段给客户端，seg1(seq=1,len=80),seg2(seq=81,len=120),seg3(seq=201,len=160),seg4(seq=361,len=140)。服务器收到seg1和seg2的ACK = 201，所以此时seg1 seg2变成发送并已经确认范畴的数据包，被移除滑动窗口，此时服务器又可以多发80+120 byte数据。假设seg3由于某些原因丢失，这个时候服务器仍然可以像客户端发送数据，但是服务器会等待seg3的ACK，否则窗口无法滑动，卡主了。seg3丢失了，即使后面的seg4收到了，客户端也无法告知服务器已经收到了seg4，试想一下，如果窗口也够大，服务器可以继续持续发送更多的片段，那么这些片段被客户端接收，只能存放到队列中，无法进行确认。
>
> 在这种情况下如果丢失了一个包那还好，只传一个丢失的包。但是如果丢失了很多包，就需要一个一个等待，浪费时间。

还有一个实现重传机制的方式就是sack（选择性确认）。如果开启sack，每一个sack段记录的是已经收到的连续的包，sack段与sack段之间断片的，也就是还没收到的（可能已经丢失，也可能是reorder）。通过sack段便可以知道多个可能已经丢失的包，这样便可以一次性的重传，而不是一个一个重传，避免因等待时间长造成的timeout问题。



要注意的是开启sack选项，也是有弊端的，因为丢包意味着网络很可能已经拥塞，这时如果一次重传多个包，很可能会造成网络更加拥塞。

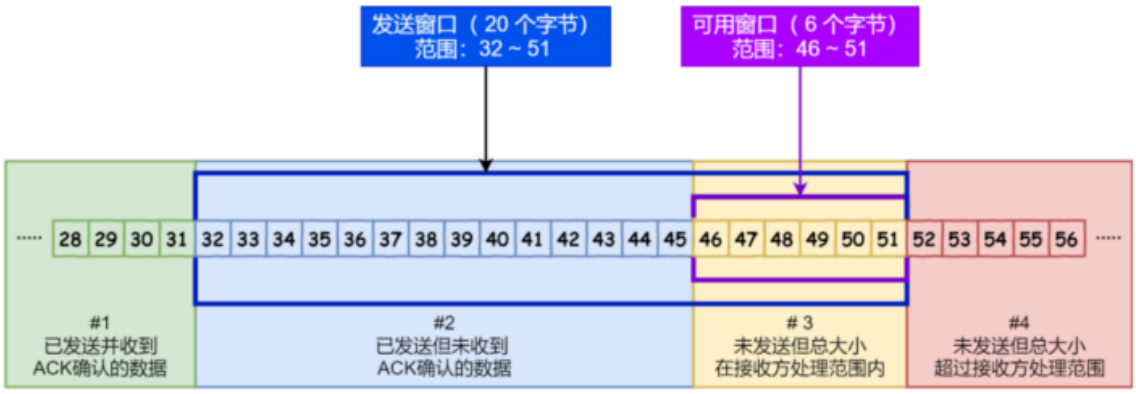
滑动窗口

最开始TCP是每发送一个数据就要进行一次确认应答，当上一个数据包收到了应答，再发送下一个。这种模式就像是面对面聊天，效率比较低下。因此引入窗口的概念：

既然引入了窗口那么必然有窗口大小这个概念，窗口大小就是无需等待确认应答，可以继续发送数据的最大值。窗口的实现实际上是操作系统开辟了一个缓存空间，发送方主机在确认应答返回之前必须在缓冲区中保留发送的数据，如果收到确认应答就可以抹去缓存的数据。

总结：滑动窗口机制是TCP的一种流量控制方法，该机制允许发送方在停止并等待确认前连续发送多个分组，而不必每发送一个分组就停下来等待确认，从而增加数据传输的速率提高应用的吞吐量。

- 发送窗口



- #1 是已发送并收到 ACK确认的数据：1~31 字节
- #2 是已发送但未收到 ACK确认的数据：32~45 字节
- #3 是未发送但总大小在接收方处理范围内（接收方还有空间）：46~51字节
- #4 是未发送但总大小超过接收方处理范围（接收方没有空间）：52字节以后

其中2和3部分是需要重点关注的。有两个指针和一个位偏移量来控制着窗口的大小。

i. 第一个指针指向已发送但未收到确认的第一个字节的序列号
ii. 第二个指针指向未发送但是可以发送的第一个字节的序列号
iii. 第三个是偏移量，是第一个指针+发送窗口大小

- 接收窗口（接收方的）

image

因为滑动窗口并不是一成不变的。比如，当接收方的应用进程读取数据的速度非常快的话，这样的话接收窗口可以很快的就空缺出来。那么新的接收窗口大小，是通过 TCP 报文中的 Windows 字段来告诉发送方。那么这个传输过程是存在时延的，所以接收窗口和发送窗口是约等于的关系。

TCP流量控制

TCP不能无脑发，如果接收方处理能差，发送方还无脑发送时候，就会触发重发机制，使得网络拥塞，因此TCP提供了一种机制让发送方根据接收方的实际接受能力控制发送的数据量。

这种机制如何工作？

接收方每次收到数据包，可以在发送确定报文的时候，同时告诉发送方自己的缓存区还剩余多少是空闲的，我们也把缓存区的剩余大小称之为接收窗口大小，用变量win来表示接收窗口的大小。发送方收到之后，便会调整自己的发送速率，也就是调整自己发送窗口的大小，当发送方收到接收窗口的大小为0时，发送方就会停止发送数据，防止出现大量丢包情况的发生。

当发送方停止发送数据后，该怎样才能知道自己可以继续发送数据？有两个方案：

1. 当接收方处理好数据，接受窗口 win > 0 时，接收方发个通知报文去通知发送方，告诉他可以继续发送数据了。当发送方收到窗口大于0的报文时，就继续发送数据。问题：假如接收方发送的通知报文，由于某种网络原因，这个报文丢失了，这时候就会引发一个问题：接收方发了通知报文后，继续等待发送方发送数据，而发送方则在等待接收方的通知报文，此时双方会陷入一种僵局（死锁）。
2. 当发送方收到接受窗口 win = 0 时，这时发送方停止发送报文，并且同时开启一个定时器，每隔一段时间就发个测试报文（窗口探测报文）去询问接收方，打听是否可以继续发送数据了，如果可以，接收方就告诉他此时接受窗口的大小；如果接受窗口大小还是为0，则发送方再次刷新启动定时器。

详细查看此链接

+++

TCP拥塞控制

> 流量控制是防止发送的报文无脑发送填满接收方的缓存，这其中发送方和接收方不知道网络环境是如何变化的，因此当网络出现拥塞的时候，可能会导致包的传输时延增加，丢包等问题，由于重传机制，会进入恶性循环。
>
> 因此当网络发生拥塞的时候，TCP就要牺牲自我。所以可以冲如何知道拥塞，怎么避免拥塞两个方向入手。
>
> 拥塞控制最直接的目的就是避免发送方的数据填满整个网络

为了能够达到发送方随时调节发送数据量的问题，定义了一个叫做拥塞窗口的概念。拥塞窗口(CWND)是发送方维护的一个状态变量，根据网络阻塞情况实时的变化。当加入拥塞窗口的概念后，发送窗口=min(接收窗口，拥塞窗口)。拥塞窗口的变化规则也非常的简单：网络中没有出现拥塞，CNWD就会增大；网络中出现拥塞，CNWD就会检查少。

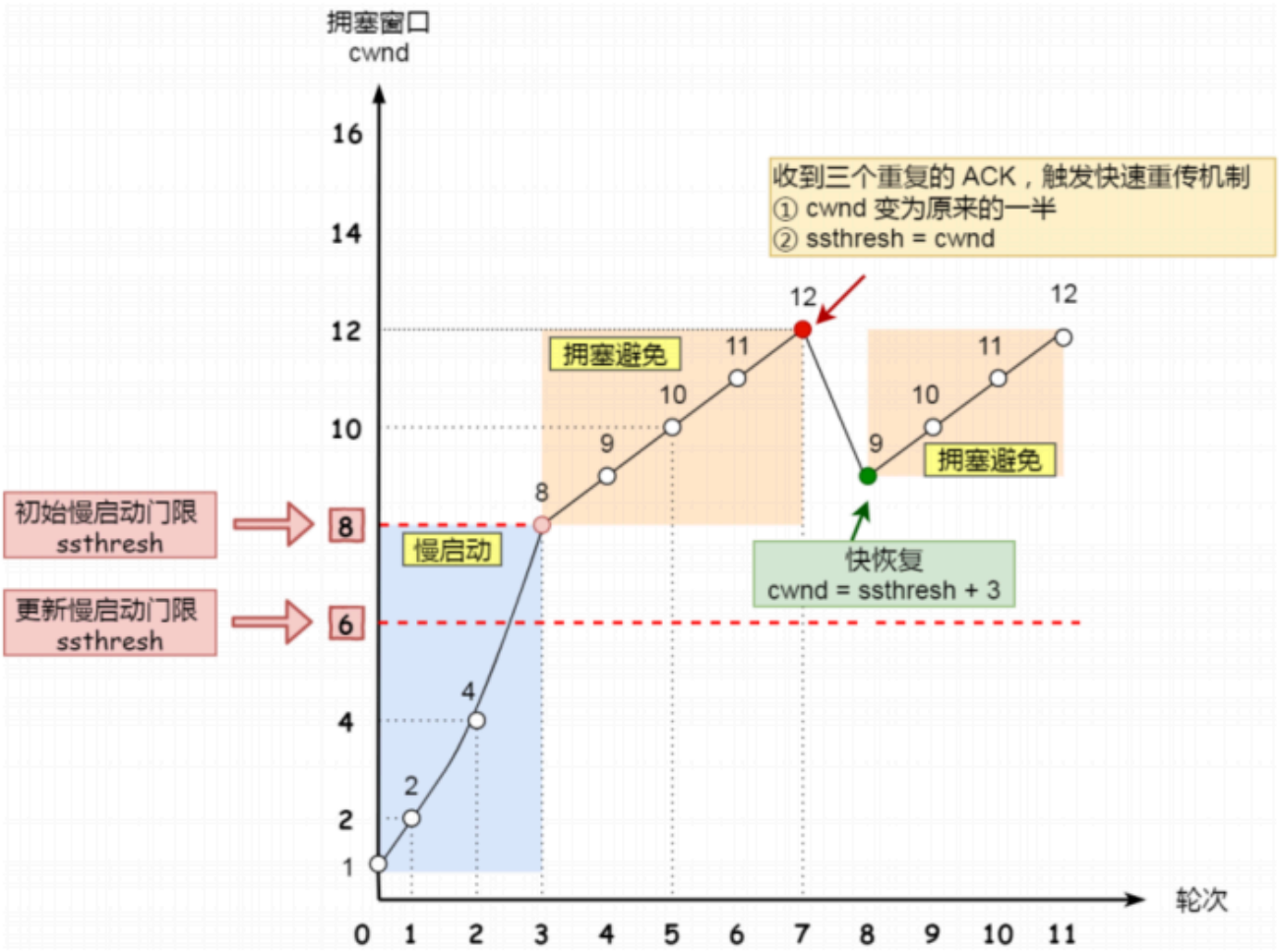
那么问题就来了：如何定义网络拥塞？

答：只要发送方没有在规定时间内收到ACK确认报文，也就是发生了超时重传，就认为网络出现了拥塞。

拥塞控制主要由四部分组成：

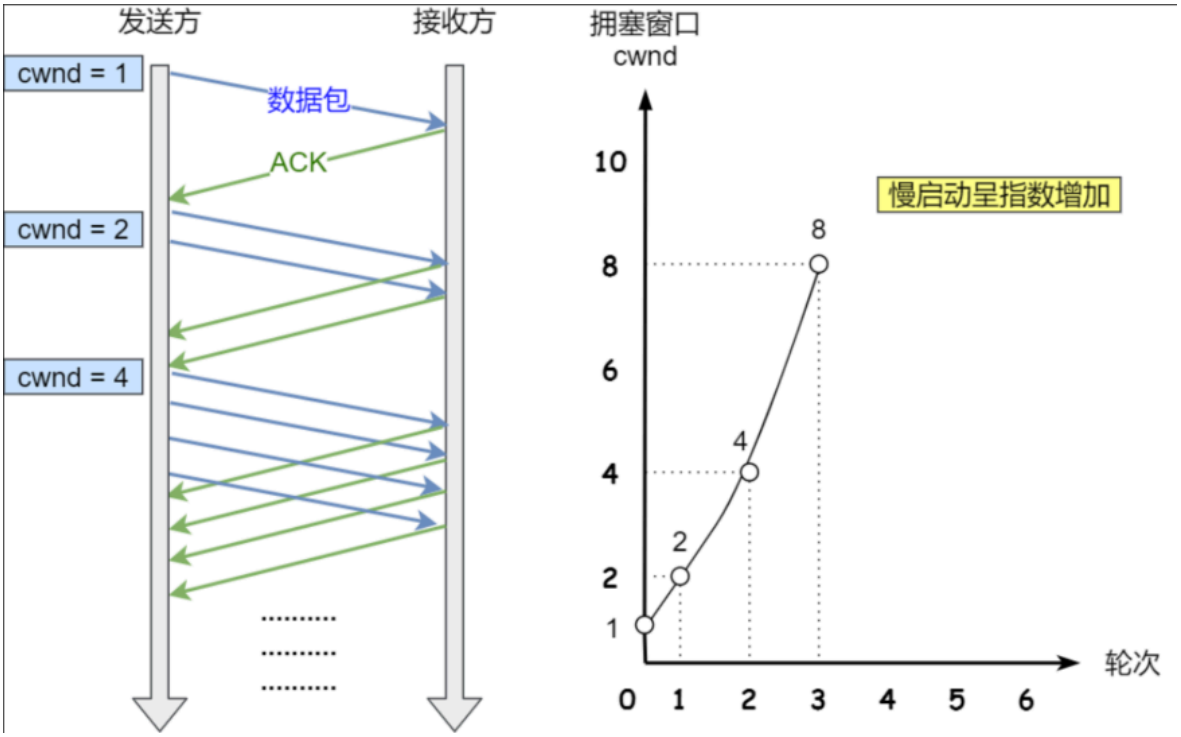
1. 慢启动
2. 拥塞避免
3. 拥塞发生
4. 快速恢复

如下图所示：



- **慢启动**

  TCP在刚建立连接后，就会一点一点的提高发送数据包的速率，规则是：**当发送方每收到一个ACK确认报文，就会将拥塞窗口CWND的大小+1。**如图所示：



CNWD也不是无限增长的，总有一个门限阈值，记作ssthresh。当cnwd小于这个阈值的时候就用慢启动算法，当大于等于这个阈值的时候就是用拥塞避免算法。

- **拥塞避免**

  当慢启动的cnwd窗口大于阈值，就会启动拥塞避免算法。

  拥塞避免的规则是：**每收到一个ACK时，cnwd增加1/cnwd**，就变成了线性增长，在图上就是线性的。

  从原先的指数增长变成了线性增长，增长速度变得缓慢了。

  最开始只有慢启动和拥塞避免两个算法，因为有时个别报文会在网络中丢失导致超时重传并误认为网络中发生拥塞，这个时候发送发错误的启动慢开始算法，并重新把拥塞窗口的值设置为1，降低了传输效率。如下图所示：

  image这样看，是不是有点傻逼呢？因此又引入了快重传和快恢复这种情况。

- **快重传**

  由于TCP有两种重传机制，即超时重传和快速重传，因此这两种重传机制引起的拥塞发生算法不同。

  当发生超时重传时，ssthresh即阈值设为cnwd/2，然后cnwd=1，接着重新开始慢启动。

  但是这种方式太激进，一下子就会回到解放前，就是我之前上面说的，拥塞控制只有这两种方案。

  但是TCP还是有更好的方式，即快速重传算法。快重传就是要求接受方赶紧进行重传，不要等待计时器到时后再重传。当接收方发现中丢包时，就会发送同一个ACK三次，发送端就需要快速重传，不必等待超时重传。这种情况下TCP机制认为网络

阻塞不是很严重，因为大部分都收到了，只有一小部分没有收到。

- **快速恢复**

  快速恢复有很多算法

  快速重传机制和快速恢复机制一般都是同时使用的，机制如下：

  i. 门限阈值设置为cnwd/2。
  ii. 拥塞窗口cnwd=阈值ssthresh+3（3是因为收到三个重复确认表示有三个报文段离开网络，这三个报文在接收方缓存中而不再网络中）
  iii. 重传丢失的数据包
  iv. 如果收到了重复的ACK，则cnwd+1
  v. 如果收到新的ACK后，窗口值设为ssthresh，退出

+++

tcp内核参数—优化

参考链接

tcpdump

命令：

支持命令行格式，常在Linux服务器中抓取和分析网络报

```
tcpdump -i eth0 'port 1111' -X -c 3
```

`-i`：指定监听的网络接口；

`-c`：在收到指定的包的数目后，tcpdump就会停止；

`-X`：告诉tcpdump命令，需要把协议头和包内容都原原本本的显示出来（tcpdump会以16进制和ASCII的形式显示），这在进行协议分析时是绝对的利器。

```
[root@linux ~]# tcpdump -i lo -nn
 1 tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
 2 listening on lo, link-type EN10MB (Ethernet), capture size 96 bytes
 3 11:02:54.253777 IP 127.0.0.1.32936 > 127.0.0.1.22: S 933696132:933696132(0) win 32767
 4 11:02:54.253831 IP 127.0.0.1.22 > 127.0.0.1.32936: S 920046702:920046702(0) ack 933696133 win 3
 5 11:02:54.253871 IP 127.0.0.1.32936 > 127.0.0.1.22: . ack 1 win 8192
 6 11:02:54.272124 IP 127.0.0.1.22 > 127.0.0.1.32936: P 1:23(22) ack 1 win 8192
 7 11:02:54.272375 IP 127.0.0.1.32936 > 127.0.0.1.22: . ack 23 win 8192
```

能抓那些包？

TCP，UDP

抓取 HTTP GET和 HTTP POST 请求流量

抓取 ICMP 数据包：`tcpdump -n icmp`

抓取 DHCP 报文: `tcpdump -v -n port 67 or 68`

wireshark：除了可以抓包，可以提供可视化的网络报分析同行

tcp延迟确认与nagle算法

tcp报文传输的数据很小，甚至小于头部20字节的时候，由于报文有效占比很低，有两种策略来减少小报文的传输：Nagle算法和延迟确认。

- Nagle算法
  Nagle算法主要是避免发送小的数据包，要求TCP连接上最多只能有一个未被确认的小分组，在该分组的确认到达之前不能发送其他的小分组。
  当发送缓冲中有多个这样的小分组的话，就收集这些小分组，组成一个大的tcp报文发送出去
  这个算法是默认打开的，如果一些小数据包交互的场景比如ssh这种，则需要关掉
- 延迟确认
  由于ACK确认报文没有携带数据，却有20字节的tcp头部和20字节的IP头部，因此传输的时候效率是很低的。
  延迟确认如下：

i. 当有响应数据需要发送的时候，ack会随着响应数据一起发送
ii. 当没有响应数据时，ACK将会延迟一段时间，以等待是否有数据然后一起发送
iii. 如果在第二步延迟确认等待期间第二个数据报文有到达了，就会立刻发送ACK

TCP 四种定时器

- 重传计时器：Retransmission Timer
  重传定时器：为了控制丢失的报文段或丢弃的报文段，也就是对报文段确认的等待时间。当TCP发送报文段时，就创建这个特定报文段的重传计时器，可能发生两种情况：若在计时器超时之前收到对报文段的确认，则撤销计时器；若在收到对特定报文段的确认之前计时器超时，则重传该报文，并把计时器复位；
- 坚持计时器：Persistent Timer
  当发送端收到零窗口的确认时，就启动坚持计时器，当坚持计时器截止期到时，发送端TCP就发送一个特殊的报文段，叫探测报文段，这个报文段只有一个字节的数据。探测报文段有序号，但序号永远不需要确认，甚至在计算对其他部分数据的确认时这个序号也被忽略。探测报文段提醒接收端TCP，确认已丢失，必须重传。
  坚持计时器的截止期设置为重传时间的值，但若没有收到从接收端来的响应，则发送另一个探测报文段，并将坚持计时器的值加倍和并复位，发送端继续发送探测报文段，将坚持计时器的值加倍和复位，知道这个值增大到阈值为止（通常为60秒）。之后，发送端每隔60s就发送一个报文段，直到窗口重新打开为止；
- 保活计时器：Keeplive Timer
  每当服务器收到客户的信息，就将keeplive timer复位，超时通常设置2小时，若服务器超过2小时还没有收到来自客户的信息，就发送探测报文段，若发送了10个探测报文段（没75秒发送一个）还没收到响应，则终止连接。
- 时间等待计时器：Time_Wait Timer。
  在连接终止期使用，当TCP关闭连接时，并不认为这个连接就真正关闭了，在时间等待期间，连接还处于一种中间过度状态。这样就可以时重复的fin报文段在到达终点后被丢弃，这个计时器的值通常设置为一格报文段寿命期望值的两倍。

服务器bind后没有调用listen监听socket连接会发生什么？

首先服务器会报错，肯定是连接不上

那么在TCP这里是表现为客户端对服务器发送SYN同步报文之后，服务端回复了RST报文。

分析：由于没用调用listen，因此找不到监听该端口的socket，因此函数找不到socket后会跳转到函数`no_tcp_socket`这个错误处理函数中，然后错误处理函数有好几种if语句，会检验豹纹的校验和，报文校验和没问题就会调用函数`tcp_send_reset`函数，发送RST报文终止连接。

如何正确关闭TCP连接？

参考

参考

服务端主动关闭连接，客户端会发生什么？

如果是服务端主动发起关闭，此时四次挥手的顺序会颠倒。那么此时客户端再向服务端发送数据时，根据TCP协议的规定，认为它是一个异常终止连接，客户端将会收到一个**RST复位**响应(而不是ACK响应)，如果客户端再次向服务端发送数据，系统将会发送一个**SIGPIPE信号**给客户端进程，告诉客户端进程该连接已关闭，不要再写了。系统给SIGPIPE信号的默认处理是直接终止收到该信号的进程，所以此时客户端进程会被极不情愿地终止。

+++

+++

网络层

IP协议

IP协议（Internet Protocol，互联网协议），是TCP/IP协议栈中最核心的协议之一，通过IP地

址，保证了联网设备的唯一性，实现了网络通信的面向无连接和不可靠的传输功能。

版本：IP协议的版本，目前的IP协议版本号为4，下一代IP协议版本号为6。

首部长度：IP报头的长度。固定部分的长度（20字节）和可变部分的长度之和。共占4位。最大为1111，即10进制的15，代表IP报头的最大长度可以为15个32bits（4字节），也就是最长可为15*4=60字节，除去固定部分的长度20字节，可变部分的长度最大为40字节。

服务类型：Type Of Service。

总长度：IP报文的总长度。报头的长度和数据部分的长度之和。

标识：唯一的标识主机发送的每一分数据报。通常每发送一个报文，它的值加一。当IP报文长度超过传输网络的MTU（最大传输单元）时必须分片，这个标识字段的值被复制到所有数据分片的标识字段中，使得这些分片在达到最终目的地时可以依照标识字段的内容重新组成原先的数据。

标志：共3位。R、DF、MF三位。目前只有后两位有效，DF位：为1表示不分片，为0表示分片。MF：为1表示“更多的片”，为0表示这是最后一片。

片位移：本分片在原先数据报文中相对首位的偏移位。（需要再乘以8）

生存时间：IP报文所允许通过的路由器的最大数量。每经过一个路由器，TTL减1，当为0时，路由器将该数据报丢弃。TTL 字段是由发送端初始设置一个 8 bit字段.推荐的初始值由分配数字 RFC 指定，当前值为 64。发送 ICMP 回显应答时经常把 TTL 设为最大值 255。

协议：指出IP报文携带的数据使用的是那种协议，以便目的主机的IP层能知道要将数据报上交到哪个进程（不同的协议有专门不同的进程处理）。和端口号类似，此处采用协议号，TCP的协议号为6，UDP的协议号为17。ICMP的协议号为1，IGMP的协议号为2.

首部校验和：计算IP头部的校验和，检查IP报头的完整性。

源IP地址：标识IP数据报的源端设备。

目的IP地址：标识IP数据报的目的地址

网关是什么呢？

什么是ARP欺骗

为什么IP地址和MAC地址缺一不可？

首先要充分了解网络分层的作用。网络层使用Ip协议将不同数据链路类型（以太网，3g，LAN）的数据报文统一成相同的形式在网络层中可以互相传播，达到任意具有IP地址的主机。但是数据传输总要经过底层，经过线路传输。这时候就是数据链路层起到了作用，不同链路类型所使用的协议方案都不尽相同，但是在链路层不能使用IP地址了，这时候要找到相对应的主机我就需要MAC地址的作用。

第二若是只使用MAC进行通信，那么一个路由表就要维护差不多2的48次方数据，大约256TB的内存，这样存储和通信效率严重降低。

ICMP协议

确认网络是否正常以及遇到异常进行异常诊断的协议。其主要功能包括：

1）确认IP包是否成功送达目标地址。

2）返回在发送过程中IP包被废弃的真正原因。

3）改善网络设置

ARP协议

主要解决MAC和IP地址之间的问题。确定了IP地址之后可以向对应IP地址的主机发送数据报，但是在底层数据要通过数据链路进行传输，必须知道每个IP地址对应的MAC地址才行。

RARP协议

将ARP反过来，从MAC地址定位到IP地址的协议。主要用于无法获取IP地址的设备，比如嵌入式设备。

DHCP协议

逐一为每台设备设置IP地址是一件非常繁琐的事情，因此为了实现IP自动分配而生的协议，即插即用。

NAT协议

为了解决IP地址日益不足的问题，出现了一种技术。不要求为每一台设备分配一个固定的IP，而是在必要的时候为一定数量的设备分配唯一的IP（路由器）。若是设备没有联网，只要保证IP地址在相应网络内部是唯一的即可。所以出现了全局IP和私有IP的问题，下面是私有地址：

```
10.0.0.0-10.255.255.255         A类
172.16.0.0–172.31.255.255                B类
192.168.0.0-192.168.255.255         C类
```

全局IP要在整个互联网范围内唯一，但是私有地址不需要。NAT就是将私有地址在链接互联网时转换成全局IP是使用的技术。但是实际上nat本质上除了私有地址公有地址转换这一种用处之外最重要的就是将处于内网中的机器进行统一的管理和维护。

数据链路层

集线器、网桥、交换机、路由器

- 历史—更好理解
  以太网最早出现的时候是一种非常松散的设计思路，当时的网络设计者希望只用一根线，就能让所有连在这根线上的计算机都能够互联。所以最早的以太网是采用同轴电缆，而所有的计算机在这根电缆上通过变压器耦合（可以理解为搭线）在同一根电缆上收发数据。这样就出现了一个问题：一次在这根线上只能有一对收发，否则就会造成干扰。
  所以这样的局域网采用了一种叫做载波侦听/冲突重发的机制：就是如果一个计算机打算向连在这条线上的另一台计算机发送数据，它首先要侦听一下这条线上有没有数据已经在传送，如果没有，就可以发，如果有就再等一段时间后再去侦听一下，直到它可以拿到这根电缆的收发权限。
  这样就造成一个问题：同一时刻只能有一对计算机通信，其他人都要等着。
  假设一个场景，一楼101、102、103和二楼201、202、203连在一根电缆上。假设一楼101和103收发数据很频繁，这样就造成二楼基本没法通信了。怎么办？办法之一就是一楼一根电缆连101、102、103、二楼一根电缆连201、202、203，大家各发各的互不干扰。但是如果，比如201要给102发数据怎么办，这时候就需要一个设备把两根电缆连起来，变成一根电缆。平时一楼和二楼连根电缆是隔离的，只有两个楼层有数据交换才把线连起来，干这个事情的设备就叫网桥。后来交换机出现了，这个问题就不存在了，交换机（理论上）允许任意两个计算机通信互不干扰
- 集线器
  hub，以太网设计之初就是想要一个网络接口和多台电脑相连。因此hub就是最简单的连接多台电脑的方式，本质上就是一个信号放大器，这样的话一台电脑发出去的信号就可以被连接到hub上的其他电脑接收到，也就可以通信。一个口发送到的信号原封不动的发送大其他口，只是简单的转发，工作在物理层。
- 网桥
  工作在数据链路层，因此跟其相关的就是mac地址。与hub无差别的转发相比，网桥会过滤mac，只有目的地址mac被匹配了才会发送到出口。（这样可以减少碰撞，冲突）一个网桥就是指的1个输入到1个输出。通俗 说网桥就是把两边设备桥起来，但不是所有流量都放行，而是放行相应mac匹配的。
- 交换机
  随着设备越来越多，如果用网桥来隔离冲突域链接不同网段比较繁琐，因此就出现

了交换机。交换机的基本功能与网桥一样，具有帧转发、帧过滤和生成树算法功能。
但是交换机是多个网桥功能的集合。即网桥一般分有两个端口，而交换机具有高密度的端口。由于交换机能够支持多个端口，因此可以把网络系统划分成为更多的物理网段，这样使得整个网络系统具有更高的带宽。而网桥仅仅支持两个端口，所以，网桥划分的物理网段是相当有限的。
网桥和交换机，mac不匹配就flood（泛洪），因此泛洪是一种数据链路层的说法。

- 路由器
  路由器工作在网络层，处理IP头部，查路由表，作三层转发。连接在路由器不同端口的设备属于不同的IP子网。
  基于IP地址做转发。

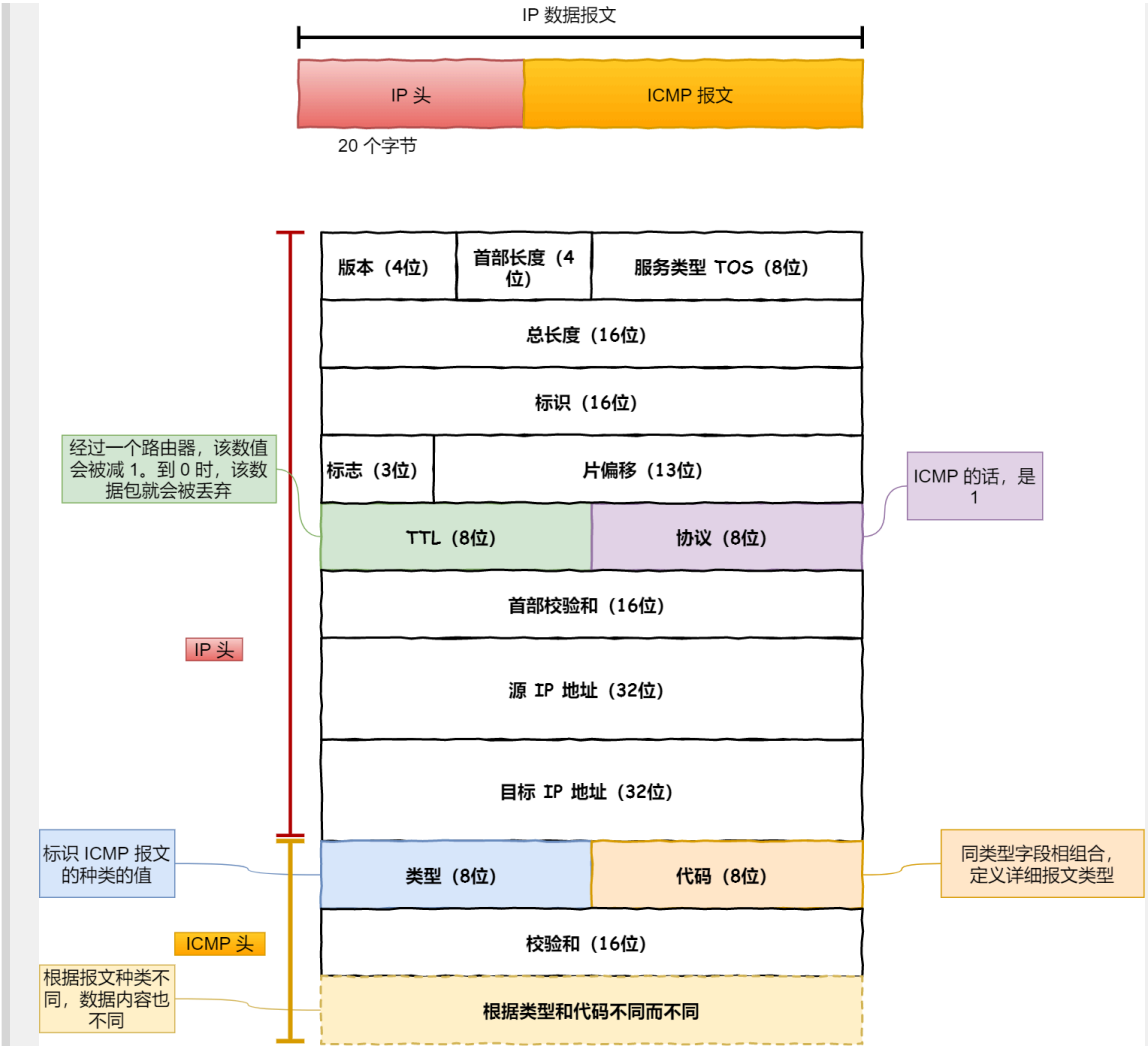
ping的原理，ping的时候到底发生了什么

- **是什么、有什么用**
  ping在潜水中主要是指收到的声呐脉冲，看看多少米深。在计算机网络中的ping是一个工具，用这个工具来检测网络的连通情况和查看网络的连接速度。
  具体就是用ping命令向想要测试的网络发送测试数据报，以此来查看对方网络是否有响应以及响应速度，达到测试的目的
  ping命令本身使用的的是网络层的ICMP协议，因此ping的本质就是ICMP协议，ICMP又需要通过IP协议进行发送，发送的数据包有一定长度，如果对方网络地址存在并且网络畅通无阻，就会收到同样大小的数据包，当然也有可能超时。因此ping本质就是IMCP协议，通过IP协议发送给对方主机。

  > ping的流程：主机A ping 主机B ----> 构建ICMP数据包 ----> 构建IP数据包 ----> IP分组 ----> 解析硬件地址封装成帧 ----> 物理层 ----> 网络层传输 ----> 到达主机B ----> 提取IP数据包交给IP层协议 ----> 提取ICMP数据包信息，构建ICMP应答包 ----> 发送给主机A

- **原理（ICMP的解析）**
  ICMP： Internet Control Message Protocol 互联网控制报文协议
  ICMP主要功能：确认发送的IP包是否到达对方主机，报告发送过程中IP包被丢弃的原因

  > ICMP报文格式如图所示：



其中注意ICMP的**类型**字段，可以分为两大类：**查询报文类型**和**差错报文类型**，如图所示：

ICMP 类型		
	内容	种类
0	回送应答（Echo Reply）	查询报文类型
3	目标不可达（Destination Unreachable）	差错报文类型
4	原点抑制（Source Quench）	差错报文类型
5	重定向或改变路由（Redirect）	差错报文类型
8	回送请求（Echo Request）	查询报文类型
11	超时（Time Exceeded）	差错报文类型

- 查询报文类型

  可以看到查询报文类型主要由两个，0和8，是回送消息。回送消息主要用于相互通信的主机和路由器之间，判断所发送的数据包是否已经成功抵达对端，ping就是用这个消息实现的。

  因此发送端会发送一个**ICMP回送请求(类型 8)给对端主机，对端主机如果可以通信会发送ICMP回送响应(类型 0)**
- 差错报文类型
  - 目标不可达消息 —— 类型 为 `3`

    当路由器无法将IP数据包发送给对端主机的时候，路由器会返回一个目标不可达的ICMP消息给发起ping命令的主机，这个时候类型字段是类型 `8`，代码字段是具体不可到达的愿意，有以下几个：

    a. 网络不可达代码为 `0` —— 当IP地址区分主机号和网络号的时候，由于路由器中的路由表匹配不到对方的网络号

    b. 主机不可达代码为 `1` —— 对端主机没有连接上网络

    c. 协议不可达代码为 `2` —— 当主机以TCP协议访问对端主机的时候，找到对端主

机，但是防火墙进制TCP协议访问

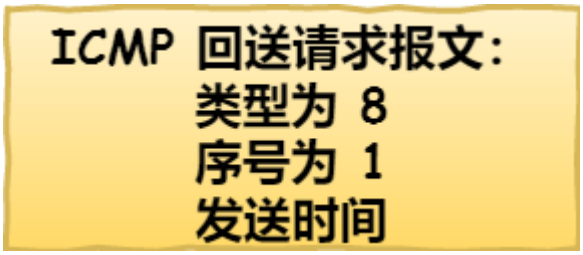
d. 端口不可达代码为 3 —— 找到对端主机，防火墙也没限制，但是没有进程监听相应端口

e. 需要进行分片但设置了不分片位代码为 4 —— 发送端主机发送IP数据包的时候，将IP首部的分片禁止标志位设置为1，由于不进行分片，当途中路由器遇到超过MTU大小的数据包时就直接扔掉，然后报告发送端主机

- 原点抑制消息 —— 类型 4
  在低速线路中会遇到网络拥堵问题，这个消息就是为了缓和这个拥堵。具体做法是当路由器向低速线路发送数据的时候，其发送队列的缓存变为0而无法发送出去，这个时候路由器向IP源地址即发送端发送一个ICMP原点抑制消息。收到该消息的主机端知道了某个线路发生了拥堵，从而增大IP包的传输间隔，减少网络拥堵。
  一般不被使用，因为可能会引起网络不公现象。
- 重定向消息 —— 类型 5
  如果路由器发送数据报不是从最优路径发送数据的话，会返回一个ICMP重定向消息给发送主机。这个消息中包含了最合适的路由信息和源数据。
- 超时消息 —— 类型 11
  IP包中有一个叫做TTL(time to live)的东西，他的值每经过一次路由器就会减1，当为0的时候该包就会被丢弃。丢弃的时候路由器就会发送一个ICMP超时消息给发送端主机，通知其该包已经被丢弃了。
  设置TTL的目的就是为了防止路由器发送循环状况，避免IP包无休止的在网络上转发。

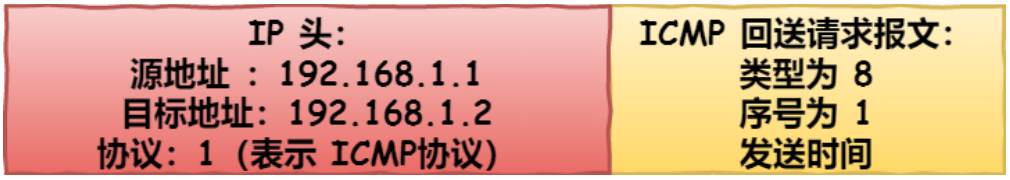
- **ping具体发送过程**
  - 在主机A执行ping命令向主机B通信的时候
    主机A会构建一个**ICMP回送请求消息数据报**，这个数据报最重要的字段是**类型和序号**。回送消息请求的类型是8，序号的作用是为了区分连续

ping的时候发出的多个数据包。因此每发送一个请求数据包，序号就会自动加1，同时为了计算往返时间RTT，还在报文的数据部分插入发送时间，因此报文如下图：



- 协议栈所做的事情

  ICMP报文和目的地址一同交给IP层进行封装，构建一个IP数据包：



  为了在数据链路层进行传输，加入MAC头：

- 在物理线路中传输
- 主机B收到数据包

  首先检查MAC地址然后和本机MAC地址进行比对，如果相同则接受，不同则丢弃

  接收后提取IP数据包交给本机IP层协议栈处理，抽出IP后然后抽出ICMP检查

  接着主机B会构建一个**ICMP回送响应消息**数据包，类型是0，序号为接受请求数据包中的序号，然后和上面一样的流程发送给主机A

  在规定时间内如果收到，则减去该数据包最初的时间得到延迟
- 通过以上可以看出，ping命令使用的是ICMP协议中类型字段**ECHO REQUEST（类型为 8 ） 和 ECHO REPLY （类型为 0）**

- **traceroute**

  traceroute的命令如下：`traceroute 192.168.1.100`

  主要作用有如下：

  i. **故意设置特殊的 TTL，来追踪去往目的地时沿途经过的路由器。**

  原理：它的原理就是利用 IP 包的生存期限 从 1 开始按照顺序递增的同时发送 UDP 包，强制接收 ICMP 超时消息的一种方法。比如，将 TTL 设置 为 1，则遇到第一个路由器，就牺牲了，接着返回 ICMP 差错报文网络包，类型是时间超时。接下来将 TTL 设置为 2，第一个路由器过

了，遇到第二个路由器也牺牲了，也同时返回了 ICMP 差错报文数据包，如此往复，直到到达目的主机。这样的过程，traceroute 就可以拿到了所有的路由器 IP。

> **发送方如何知道发出的 UDP 包是否到达了目的主机呢？**
>
> 答：traceroute 在发送 UDP 包时，会填入一个不可能的端口号值作为 UDP 目标端口号（大于 3000 ）。当目的主机，收到 UDP 包后，会返回 ICMP 差错报文消息，但这个差错报文消息的类型是「端口不可达」。所以，当差错报文类型是端口不可达时，说明发送方发出的 UDP 包到达了目的主机。

2. **故意设置不分片，从而确定路径的 MTU**

原理：首先在发送端主机发送 IP 数据报时，将 IP 包首部的分片禁止标志位设置为 1。根据这个标志位，途中的路由器不会对大数据包进行分片，而是将包丢弃。随后，通过一个 ICMP 的不可达消息将数据链路上 MTU 的值一起给发送主机，不可达消息的类型为「需要进行分片但设置了不分片位」。发送主机端每次收到 ICMP 差错报文时就减少包的大小，以此来定位一个合适的 MTU 值，以便能到达目标主机。

产生信道争用问题，解决办法？

共享介质性网络（多个设备共享一个通信介质），基本采用半双工通信

非共享介质网络（计算机之间不互相链接，而是都与交换机直接相连

CSMA/CD（载波侦听多路访问/冲突检测）

工作原理：发送数据前 先侦听信道是否空闲 ,若空闲，则立即发送数据。若信道忙碌，则等待一段时间至信道中的信息传输结束后再发送数据；若在上一段信息发送结束后，同时有两个或两个以上的节点都提出发送请求，则判定为冲突。若侦听到冲突,则立即停止发送数据，等待一段随机时间,再重新尝试。总结：先听后发，边发边听，冲突停发，随机延迟后重发。