1、OSI 的七层模型分别是？各自的功能是什么？

简要概括

物理层：底层数据传输，如网线；网卡标准。

数据链路层：定义数据的基本格式，如何传输，如何标识；如网卡 MAC 地址。

网络层：定义 IP 编址，定义路由功能；如不同设备的数据转发。

传输层：端到端传输数据的基本功能；如 TCP、UDP。

会话层：控制应用程序之间会话能力；如不同软件数据分发给不同软件。

表示层：数据格式标识，基本压缩加密功能。

应用层：各种应用软件，包括 Web 应用。

说明

- 在四层，既传输层数据被称作 **tcp 报文段或 udp 用户数据报**（Segments）；
- 三层网络层数据被称做**包**（Packages）；
- 二层数据链路层时数据被称为**帧**（Frames）；
- 一层物理层时数据被称为**比特流**（Bits）。

总结

- 网络七层模型是一个标准，而非实现。
- 网络四层模型是一个实现的应用模型。
- 网络四层模型由七层模型简化合并而来。

2、说一下一次完整的 HTTP 请求过程包括哪些内容？

第一种回答

- 建立起客户机和服务器连接。
- 建立连接后，客户机发送一个请求给服务器。
- 服务器收到请求给予响应信息。
- 客户端浏览器将返回的内容解析并呈现，断开连接。

第二种回答

域名解析 --> 发起 TCP 的 3 次握手 --> 建立 TCP 连接后发起 http 请求 --> 服务器响应 http 请求，浏览器得到 html 代码 --> 浏览器解析 html 代码，并请求 html 代码中的资源（如 js、css、图片等） --> 浏览器对页面进行渲染呈现给用户。

3、你知道 DNS 是什么？

官方解释：DNS（Domain Name System，域名系统），因特网上作为**域名和 IP 地址相互映射**的一个**分布式数据库**，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的 IP 数串。

通过主机名，最终得到该主机名对应的 IP 地址的过程叫做域名解析（或主机名解析）。

通俗的讲，我们更习惯于记住一个网站的名字，比如 www.baidu.com,而不是记住它的 ip 地址，比如：167.23.10.2。

4、DNS 的工作原理？

将主机域名转换为 ip 地址，属于应用层协议，使用 UDP 传输。（DNS 应用层协议，以前有个考官问过）

过程：

总结： 浏览器缓存，系统缓存，路由器缓存，ISP 服务器缓存，根域名服务器缓存，顶级域名服务器缓存，主域名服务器缓存。

一、主机向本地域名服务器的查询一般都是采用递归查询。 二、本地域名服务器向根域名服务器的查询的迭代查询。

1. 当用户输入域名时，浏览器先检查自己的缓存中是否包含这个域名映射的 ip 地址，有解析结束。 2）若没命中，则检查操作系统缓存（如 Windows 的 hosts）中有没有解析过的结果，有解析结束。 3）若无命中，则请求本地域名服务器解析（LDNS）。4)若 LDNS 没有命中就直接跳到根域名服务器请求解析。根域名服务器返回给 LDNS 一个 主域名服务器地址。 5）此时 LDNS 再发送请求给上一步返回的 gTLD（ 通用顶级域）， 接受请求的 gTLD 查找并返回这个域名对应的 Name Server 的地址 6）Name Server 根据映射关系表找到目标 ip，返回给 LDNS
2. LDNS 缓存这个域名和对应的 ip， 把解析的结果返回给用户，用户根据 TTL 值缓存到本地系统缓存中，域名解析过程至此结束

5、为什么域名解析用 UDP 协议？

因为 UDP 快啊！UDP 的 DNS 协议只要一个请求、一个应答就好了。

而使用基于 TCP 的 DNS 协议要三次握手、发送数据以及应答、四次挥手，但是 UDP 协议传输内容不能超过 512 字节。

不过客户端向 DNS 服务器查询域名，一般返回的内容都不超过 512 字节，用 UDP 传输即可。

6、为什么区域传送用 TCP 协议？

因为 TCP 协议可靠性好啊！

你要从主 DNS 上复制内容啊，你用不可靠的 UDP？ 因为 TCP 协议传输的内容大啊，你用最大只能传 512 字节的 UDP 协议？万一同步的数据大于 512 字节，你怎么办？所以用 TCP 协议比较好！

7、HTTP 长连接和短连接的区别

在 HTTP/1.0 中默认使用短连接。也就是说，客户端和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。

而从 HTTP/1.1 起，默认使用长连接，用以保持连接特性。

8、什么是 TCP 粘包/拆包？发生的原因？

一个完整的业务可能会被 TCP 拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这个就是 TCP 的拆包和粘包问题。

原因

1、应用程序写入数据的字节大小大于套接字发送缓冲区的大小.

2、进行 MSS 大小的 TCP 分段。(MSS=TCP 报文段长度-TCP 首部长度)

3、以太网的 payload 大于 MTU 进行 IP 分片。（ MTU 指：一种通信协议的某一层上面所能通过的最大数据包大小。）

解决方案

1、消息定长。

2、在包尾部增加回车或者空格符等特殊字符进行分割

3、将消息分为消息头和消息尾

4、使用其它复杂的协议，如 RTMP 协议等。

9、为什么服务器会缓存这一项功能?如何实现的？

原因

- 缓解服务器压力；
- 降低客户端获取资源的延迟：缓存通常位于内存中，读取缓存的速度更快。并且缓存服务器在地理位置上也有可能比源服务器来得近，例如浏览器缓存。

实现方法

- 让代理服务器进行缓存；
- 让客户端浏览器进行缓存。

10、HTTP 请求方法你知道多少？

客户端发送的 **请求报文** 第一行为请求行，包含了方法字段。

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 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方法。

序号	方法	描述
1	GET	请求指定的页面信息，并返回实体主体。
2	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
3	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。
4	PUT	从客户端向服务器传送的数据取代指定的文档的内容。
5	DELETE	请求服务器删除指定的页面。
6	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。
7	OPTIONS	允许客户端查看服务器的性能。
8	TRACE	回显服务器收到的请求，主要用于测试或诊断。
9	PATCH	是对 PUT 方法的补充，用来对已知资源进行局部更新 。

11、GET 和 POST 的区别，你知道哪些？

get 是获取数据，post 是修改数据

get 把请求的数据放在 url 上， 以?分割 URL 和传输数据，参数之间以&相连，所以 get 不太安全。而 post 把数据放在 HTTP 的包体内（request body 相对安全）

get 提交的数据最大是 2k（ 限制实际上取决于浏览器）， post 理论上没有限制。

GET 产生一个 TCP 数据包，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); POST 产生两个 TCP 数据包，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)。

GET 请求会被浏览器主动缓存，而 POST 不会，除非手动设置。

本质区别：GET 是幂等的，而 POST 不是幂等的

> 这里的幂等性：幂等性是指一次和多次请求某一个资源应该具有同样的副作用。简单来说意味着对同一 URL 的多个请求应该返回同样的结果。

正因为它们有这样的区别，所以不应该且**不能用 get 请求做数据的增删改这些有副作用的操作**。因为 get 请求是幂等的，**在网络不好的隧道中会尝试重试**。如果用 get 请求增数据，会有**重复操作**的风险，而这种重复操作可能会导致副作用（浏览器和操作系统并不知道你会用 get 请求去做增操作）。

12、一个 TCP 连接可以对应几个 HTTP 请求？

如果维持连接，一个 TCP 连接是可以发送多个 HTTP 请求的。

13、一个 TCP 连接中 HTTP 请求发送可以一起发送么（比如一起发三个请求，再三个响应一起接收）？

HTTP/1.1 存在一个问题，单个 TCP 连接在同一时刻只能处理一个请求，意思是说：两个请求的生命周期不能重叠，任意两个 HTTP 请求从开始到结束的时间在同一个 TCP 连接里不能重叠。

在 HTTP/1.1 存在 Pipelining 技术可以完成这个多个请求同时发送，但是由于浏览器默认关闭，所以可以认为这是不可行的。在 HTTP2 中由于 Multiplexing 特点的存在，多个 HTTP 请求可以在同一个 TCP 连接中并行进行。

那么在 HTTP/1.1 时代，浏览器是如何提高页面加载效率的呢？主要有下面两点：

- 维持和服务器已经建立的 TCP 连接，在同一连接上顺序处理多个请求。
- 和服务器建立多个 TCP 连接。

14、浏览器对同一 Host 建立 TCP 连接到的数量有没有限制？

假设我们还处在 HTTP/1.1 时代，那个时候没有多路传输，当浏览器拿到一个有几十张图片的网页该怎么办呢？

肯定不能只开一个 TCP 连接顺序下载，那样用户肯定等的很难受，但是如果每个图片都开一个 TCP 连接发 HTTP 请求，那电脑或者服务器都可能受不了，要是有 1000 张图片的话总不能开 1000 个 TCP 连接吧，你的电脑同意 NAT 也不一定会同意。

有。Chrome 最多允许对同一个 Host 建立六个 TCP 连接。不同的浏览器有一些区别。

如果图片都是 HTTPS 连接并且在同一个域名下，那么浏览器在 SSL 握手之后会和服务器商量能不能用 HTTP2，如果能的话就使用 Multiplexing 功能在这个连接上进行多路传输。

不过也未必会所有挂在这个域名的资源都会使用一个 TCP 连接去获取，但是可以确定的是 Multiplexing 很可能会被用到。

如果发现用不了 HTTP2 呢？或者用不了 HTTPS（现实中的 HTTP2 都是在 HTTPS 上实现的，所以也就是只能使用 HTTP/1.1）。

那浏览器就会在一个 HOST 上建立多个 TCP 连接，连接数量的最大限制取决于浏览器设置，这些连接会在空闲的时候被浏览器用来发送新的请求，如果所有的连接都正在发送请求呢？那其他的请求就只能等等了。

15、在浏览器中输入 url 地址后显示主页的过程?

- 根据域名，进行 DNS 域名解析；
- 拿到解析的 IP 地址，建立 TCP 连接；
- 向 IP 地址，发送 HTTP 请求；
- 服务器处理请求；
- 返回响应结果；
- 关闭 TCP 连接；
- 浏览器解析 HTML；
- 浏览器布局渲染；

16、在浏览器地址栏输入一个 URL 后回车，背后会进行哪些技术步骤？

第一种回答

1、查浏览器缓存，看看有没有已经缓存好的，如果没有

2 、检查本机 host 文件，

3、调用 API，Linux 下 Socket 函数 gethostbyname

4、向 DNS 服务器发送 DNS 请求，查询本地 DNS 服务器，这其中用的是 UDP 的协议

5、如果在一个子网内采用 ARP 地址解析协议进行 ARP 查询如果不在一个子网那就需要对默认网关进行 DNS 查询，如果还找不到会一直向上找根 DNS 服务器，直到最终拿到 IP 地址（全球 400 多个根 DNS 服务器，由 13 个不同的组织管理）

6、这个时候我们就有了服务器的 IP 地址 以及默认的端口号了，http 默认是 80 https 是 443 端口号，会，首先尝试 http 然后调用 Socket 建立 TCP 连接，

7、经过三次握手成功建立连接后，开始传送数据，如果正是 http 协议的话，就返回就完事了，

8、如果不是 http 协议，服务器会返回一个 5 开头的的重定向消息，告诉我们用的是 https，那就是说 IP 没变，但是端口号从 80 变成 443 了，好了，再四次挥手，完事，

9、再来一遍，这次除了上述的端口号从 80 变成 443 之外，还会采用 SSL 的加密技术来保证传输数据的安全性，保证数据传输过程中不被修改或者替换之类的，

10、这次依然是三次握手，沟通好双方使用的认证算法，加密和检验算法，在此过程中也会检验对方的 CA 安全证书。

11、确认无误后，开始通信，然后服务器就会返回你所要访问的网址的一些数据，在此过程中会将界面进行渲染，牵涉到 ajax 技术之类的，直到最后我们看到色彩斑斓的网页

17、谈谈 DNS 解析过程，具体一点

- 请求一旦发起，若是 chrome 浏览器，先在浏览器找之前**有没有缓存过的域名所对应的 ip 地址**，有的话，直接跳过 dns 解析了，若是没有，就会**找硬盘的 hosts 文件**，看看有没有，有的话，直接找到 hosts 文件里面的 ip
- 如果本地的 hosts 文件没有能得到对应的 ip 地址，浏览器会发出一个 **dns 请求到本地 dns 服务器**，**本地 dns 服务器一般都是你的网络接入服务器商提供**，比如中国电信，中国移动等。

- 查询你输入的网址的 DNS 请求到达本地 DNS 服务器之后，**本地 DNS 服务器会首先查询它的缓存记录**，如果缓存中有此条记录，就可以直接返回结果，此过程是**递归的方式进行查询**。如果没有，本地 DNS 服务器还要向 **DNS 根服务器**进行查询。
- 本地 DNS 服务器继续向域服务器发出请求，在这个例子中，请求的对象是.com 域服务器。.com 域服务器收到请求之后，也不会直接返回域名和 IP 地址的对应关系，而是告诉本地 DNS 服务器，你的域名的解析服务器的地址。
- 最后，本地 DNS 服务器向**域名的解析服务器**发出请求，这时就能收到一个域名和 IP 地址对应关系，本地 DNS 服务器不仅要把 IP 地址返回给用户电脑，还要把这个对应关系保存在缓存中，以备下次别的用户查询时，可以直接返回结果，加快网络访问。

18、DNS 负载均衡是什么策略？

当一个网站有足够多的用户的时候，假如每次请求的资源都位于同一台机器上面，那么这台机器随时可能会崩掉。

处理办法就是用 DNS 负载均衡技术，它的原理是在 **DNS 服务器中为同一个主机名配置多个 IP 地址,在应答 DNS 查询时,DNS 服务器对每个查询将以 DNS 文件中主机记录的 IP 地址按顺序返回不同的解析结果,将客户端的访问引导到不同的机器上去,使得不同的客户端访问不同的服务器**，从而达到负载均衡的目的。例如可以根据每台机器的负载量，该机器离用户地理位置的距离等等。

19、HTTPS 和 HTTP 的区别

1. HTTP 协议传输的数据都是未加密的，也就是明文的，因此使用 HTTP 协议传输隐私信息非常不安全，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，要比 http 协议安全。
2. https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。 3、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。

20、什么是 SSL/TLS ？

SSL 代表安全套接字层。它是一种用于加密和验证应用程序（如浏览器）和 Web 服务器之间发送的数据的协议。 身份验证 ， 加密 Https 的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。

SSL/TLS 协议作用：认证用户和服务，加密数据，维护数据的完整性的应用层协议加密和解密需要两个不同的密钥，故被称为非对称加密；加密和解密都使用同一个密钥的

对称加密：优点在于加密、解密效率通常比较高 ，HTTPS 是基于非对称加密的， 公钥是公开的，

21、HTTPS 是如何保证数据传输的安全，整体的流程是什么？（SSL 是怎么工作保证安全的）

（1）客户端向服务器端发起 SSL 连接请求； （2） 服务器把公钥发送给客户端，并且服务器端保存着唯一的私钥 （3）客户端用公钥对双方通信的对称秘钥进行加密，并发送给服务器端 （4）服务器利用自己唯一的私钥对客户端发来的对称秘钥进行解密， （5）进行数据传输，服务器和客户端双方用公有的相同的对称秘钥对数据进行加密解密，可以保证在数据收发过程中的安全，即是第三方获得数据包，也无法对其进行加密，解密和篡改。

因为数字签名、摘要是证书防伪非常关键的武器。 “摘要”就是对传输的内容，通过 hash 算法计算出一段固定长度的串。然后，通过发送方的私钥对这段摘要进行加密，加密后得到的结果就是“数字签名”

SSL/TLS 协议的基本思路是采用公钥加密法，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。

补充：SSL/TLS 的四次握手，目前网上的主流答案都在重复阮一峰老师的博客，属于 TLS 1.0 版本的答案，使用 RSA 密钥交换算法。但是现在 TLS 1.2 已经成为主流，使用 ECDHE 算法，如果面试可以说出这个版本的答案，应该会更好。

22、如何保证公钥不被篡改？

将公钥放在数字证书中。只要证书是可信的，公钥就是可信的。

公钥加密计算量太大，如何减少耗用的时间？

每一次对话（session），客户端和服务器端都生成一个"对话密钥"（session key），用它来加密信息。由于"对话密钥"是对称加密，所以运算速度非常快，而服务器公钥只用于加密"对话密钥"本身，这样就减少了加密运算的消耗时间。（1） 客户端向服务器端索要并验证公钥。（2） 双方协商生成"对话密钥"。（3）双方采用"对话密钥"进行加密通信。上面过程的前两步，又称为"握手阶段"（handshake）。

25、Cookie 有什么用途？用途

- 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）
- 个性化设置（如用户自定义设置、主题等）
- 浏览器行为跟踪（如跟踪分析用户行为等）

27、Session 的工作原理是什么？

session 的工作原理是客户端登录完成之后，服务器会创建对应的 session，session 创建完之后，会把 session 的 id 发送给客户端，客户端再存储到浏览器中。这样客户端每次访问服务器时，都会带着 sessionid，服务器拿到 sessionid 之后，在内存找到与之对应的 session 这样就可以正常工作了。

23、HTTP 请求和响应报文有哪些主要字段？

请求报文

简单来说：

- 请求行：Request Line
- 请求头：Request Headers
- 请求体：Request Body

响应报文

简单来说：

- 状态行：Status Line
- 响应头：Response Headers
- 响应体：Response Body

24、Cookie 是什么？

HTTP 协议是**无状态**的，主要是为了让 HTTP 协议尽可能简单，使得它能够处理大量事务，HTTP/1.1 引入 Cookie 来保存状态信息。

Cookie 是**服务器发送到用户浏览器并保存在本地的一小块数据**，它会在浏览器之后向同一服务器再次发起请求时被携带上，用于告知服务端两个请求是否来自同一浏览器。由于之后每次请求都会需要携带 Cookie 数据，因此会带来额外的性能开销（尤其是在移动环境下）。

Cookie 曾一度用于客户端数据的存储，因为当时并没有其它合适的存储办法而作为唯一的存储手段，但现在随着现代浏览器开始支持各种各样的存储方式，Cookie 渐渐被淘汰。

新的浏览器 API 已经允许开发者直接将数据存储到本地，如使用 Web storage API（本地存储和会话存储）或 IndexedDB。

cookie 的出现是因为 HTTP 是无状态的一种协议，换句话说，服务器记不住你，可能你每刷新一次网页，就要重新输入一次账号密码进行登录。这显然是让人无法接受的，cookie 的作用就好比服务器给你贴个标签，然后你每次向服务器再发请求时，服务器就能够 cookie 认出你。

抽象地概括一下：一个 cookie 可以认为是一个「变量」，形如 name=value，存储在浏览器；一个 session 可以理解为一种数据结构，多数情况是「映射」（键值对），存储在服务器上。

26、Session 知识大总结

除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

Session 可以存储在服务器上的文件、数据库或者内存中。也可以将 Session 存储在 Redis 这种内存型数据库中，效率会更高。

使用 Session 维护用户登录状态的过程如下：

1. 用户进行登录时，用户提交包含用户名和密码的表单，放入 HTTP 请求报文中；
2. 服务器验证该用户名和密码，如果正确则把用户信息存储到 Redis 中，它在 Redis 中的 Key 称为 Session ID；
3. 服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；
4. 客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之前的业务操作。

   注意：Session ID 的安全性问题，不能让它被恶意攻击者轻易获取，那么就不能产生一个容易被猜到的 Session ID 值。此外，还需要经常重新生成 Session ID。在对安全性要求极高的场景下，例如转账等操作，除了使用 Session 管理用户状态之外，还需要对用户进行重新验证，比如重新输入密码，或者使用短信验证码等方式。

28、Cookie 与 Session 的对比

HTTP 作为无状态协议，必然需要在某种方式保持连接状态。这里简要介绍一下 Cookie 和 Session。

Cookie

Cookie 是客户端保持状态的方法。

Cookie 简单的理解就是存储由服务器发至客户端并由客户端保存的一段字符串。为了保持会话，服务器可以在响应客户端请求时将 Cookie 字符串放在 Set-Cookie 下，客户机收到 Cookie 之后保存这段字符串，之后再请求时候带上 Cookie 就可以被识别。

除了上面提到的这些，Cookie 在客户端的保存形式可以有两种，一种是会话 Cookie 一种是持久 Cookie，会话 Cookie 就是将服务器返回的 Cookie 字符串保持在内存中，关闭浏览器之后自动销毁，持久 Cookie 则是存储在客户端磁盘上，其有效时间在服务器响应头中被指定，在有效期内，客户端再次请求服务器时都可以直接从本地取出。需要说明的是，存储在磁盘中的 Cookie 是可以被多个浏览器代理所共享的。

Session

Session 是服务器保持状态的方法。

首先需要明确的是，Session 保存在服务器上，可以保存在数据库、文件或内存中，每个用户有独立的 Session 用户在客户端上记录用户的操作。我们可以理解为每个用户有一个独一无二的 Session ID 作为 Session 文件的 Hash 键，通过这个值可以锁定具体的 Session 结构的数据，这个 Session 结构中存储了用户操作行为。

当服务器需要识别客户端时就需要结合 Cookie 了。每次 HTTP 请求的时候，客户端都会发送相应的 Cookie 信息到服务端。

实际上大多数的应用都是用 Cookie 来实现 Session 跟踪的，第一次创建 Session 的时候，服务端会在 HTTP 协议中告诉客户端，需要在 Cookie 里面记录一个 Session ID，以后每次请求把这个会话 ID 发送到服务器，我就知道你是谁了。

如果客户端的浏览器禁用了 Cookie，会使用一种叫做 URL 重写的技术来进行会话跟踪，即每次 HTTP 交互，URL 后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户，这样就可以帮用户完成诸如用户名等信息自动填入的操作了。

29、SQL 注入攻击了解吗？

攻击者在 HTTP 请求中注入恶意的 SQL 代码，服务器使用参数构建数据库 SQL 命令时，恶意 SQL 被一起构造，并在数据库中执行。

用户登录，输入用户名 lianggzone，密码 ' or '1'='1 ，如果此时使用参数构造的方式，就会出现 select * from user where name = 'lianggzone' and password = '' or '1'='1' 不管用户名和密码是什么内容，使查询出来的用户列表不为空。如何防范 SQL 注入攻击使用预编译的 PrepareStatement 是必须的，但是一般我们会从两个方面同时入手。

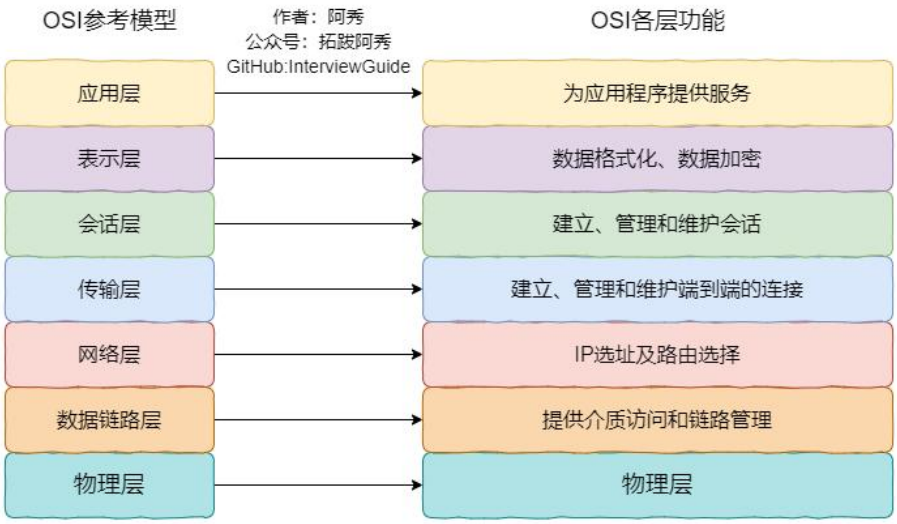
Web 端

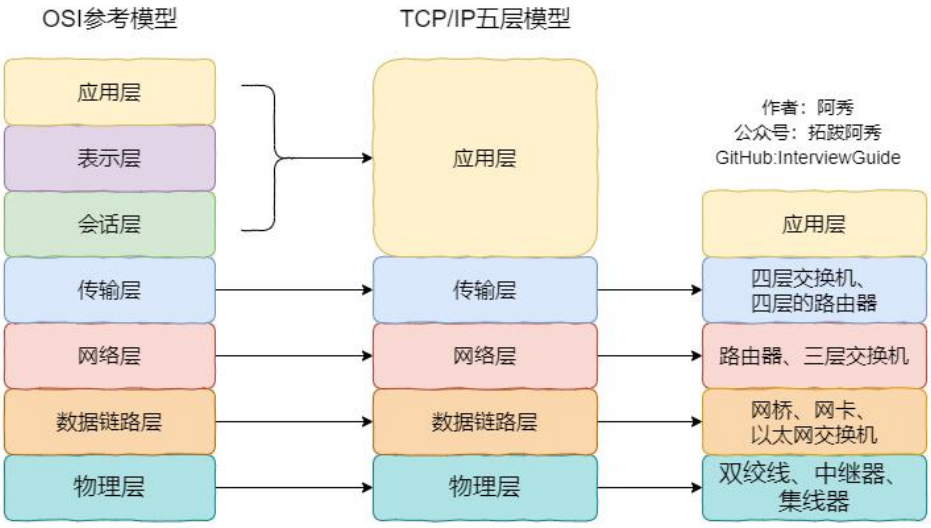
- 1）有效性检验。
- 2）限制字符串输入的长度。

服务端

- 1）不用拼接 SQL 字符串。
- 2）使用预编译的 PrepareStatement。
- 3）有效性检验。(为什么服务端还要做有效性检验？第一准则，外部都是不可信的，防止攻击者绕过 Web 端请求)
- 4）过滤 SQL 需要的参数中的特殊字符。比如单引号、双引号。

30、网络的七层模型与各自的功能（图片版）





32、端口有效范围是多少到多少？

0-1023 为知名端口号，比如其中 HTTP 是 80，FTP 是 20（数据端口）、21（控制端口）

UDP 和 TCP 报头使用两个字节存放端口号，所以端口号的有效范围是从 0 到 65535。动态端口的范围是从 1024 到 65535

31、什么是 RARP？工作原理

概括： 反向地址转换协议，网络层协议，RARP 与 ARP 工作方式相反。 RARP 使只知道自己硬件地址的主机能够知道其 IP 地址。

RARP 发出要反向解释的物理地址并希望返回其 IP 地址，应答包括能够提供所需信息的 RARP 服务器发出的 IP 地址。

原理：

- (1)网络上的每台设备都会有一个独一无二的硬件地址，通常是由设备厂商分配的 MAC 地址。主机从网卡上读取 MAC 地址，然后在网络上发送一个 RARP 请求的广播数据包，请求 RARP 服务器回复该主机的 IP 地址
- (2)RARP 服务器收到了 RARP 请求数据包，为其分配 IP 地址，并将 RARP 回应发送给主机。
- (3)PC1 收到 RARP 回应后，就使用得到的 IP 地址进行通讯。

33、为何需要把 TCP/IP 协议栈分成 5 层（或 7 层）？开放式回答。

答：ARPANET 的研制经验表明，对于复杂的计算机网络协议，其结构应该是层次式的。

分层的好处：

①各层之间是独立的

②灵活性好

③结构上可以分隔开

④易于实现和维护

⑤能促进标准化工作。

35、HTTP 中缓存的私有和共有字段？知道吗？

private 指令规定了将资源作为私有缓存，只能被单独用户使用，一般存储在用户浏览器中。

```
Cache-Control: private
```

public 指令规定了将资源作为公共缓存，可以被多个用户使用，一般存储在代理服务器中。

```
Cache-Control: public
```

34、DNS 查询方式有哪些？

递归解析

当局部 DNS 服务器自己不能回答客户机的 DNS 查询时，它就需要向其他 DNS 服务器进行查询。此时有两种方式。

局部 DNS 服务器自己负责向其他 DNS 服务器进行查询，一般是先向该域名的根域服务器查询，再由根域名服务器一级级向下查询。最后得到的查询结果返回给局部 DNS 服务器，再由局部 DNS 服务器返回给客户端。

迭代解析

当局部 DNS 服务器自己不能回答客户机的 DNS 查询时，也可以通过迭代查询的方式进行解析。

局部 DNS 服务器不是自己向其他 DNS 服务器进行查询，**而是把能解析该域名的其他 DNS 服务器的 IP 地址返回给客户端 DNS 程序**，客户端 DNS 程序再继续向这些 DNS 服务器进行查询，直到得到查询结果为止。

也就是说，迭代解析只是帮你找到相关的服务器而已，而不会亲自帮你去查。

举个例子，比如说："baidu.com 的服务器 ip 地址在 192.168.4.5 这里，你自己去查吧，本人比较忙，只能帮你到这里了。"差不多就是这个意思了

36、GET 方法参数写法是固定的吗？

在约定中，我们的参数是写在 ？后面，用 & 分割。

我们知道，解析报文的过程是通过获取 TCP 数据，用正则等工具从数据中获取 Header 和 Body，从而提取参数。

比如 header 请求头中添加 token，来验证用户是否登录等权限问题。

也就是说，我们可以自己约定参数的写法，只要服务端能够解释出来就行，万变不离其宗。

37、GET 方法的长度限制是怎么回事？

网络上都会提到浏览器地址栏输入的参数是有限的。

首先说明一点，HTTP 协议没有 Body 和 URL 的长度限制，对 URL 限制的大多是浏览器和服务器的原因。

浏览器原因就不说了，服务器是因为处理长 URL 要消耗比较多的资源，为了性能和安全（防止恶意构造长 URL 来攻击）考虑，会给 URL 长度加限制。

38、POST 方法比 GET 方法安全？

有人说 POST 比 GET 安全，因为数据在地址栏上不可见。

然而，从传输的角度来说，他们都是不安全的，因为 HTTP 在网络上是明文传输的，只要在网络节点上捉包，就能完整地获取数据报文。

要想安全传输，就只有加密，也就是 HTTPS。

39、POST 方法会产生两个 TCP 数据包？你了解吗？

有些文章中提到，POST 会将 header 和 body 分开发送，先发送 header，服务端返回 100 状态码再发送 body。

HTTP 协议中没有明确说明 POST 会产生两个 TCP 数据包，而且实际测试(Chrome)发现，header 和 body 不会分开发送。

所以，header 和 body 分开发送是部分浏览器或框架的请求方法，不属于 post 必然行为。

40、Session 是什么？

除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

Session 可以存储在服务器上的文件、数据库或者内存中。也可以将 Session 存储在 Redis 这种内存型数据库中，效率会更高。

42、Session 和 cookie 应该如何去选择（适用场景）？

- Cookie 只能存储 ASCII 码字符串，而 Session 则可以存储任何类型的数据，因此在考虑数据复杂性时首选 Session；
- Cookie 存储在浏览器中，容易被恶意查看。如果非要将一些隐私数据存在 Cookie 中，可以将 Cookie 值进行加密，然后在服务器进行解密；
- 对于大型网站，如果用户所有的信息都存储在 Session 中，那么开销是非常大的，因此不建议将所有的用户信息都存储到 Session 中。

41、使用 Session 的过程是怎样的？

过程如下：

- 用户进行登录时，用户提交包含用户名和密码的表单，放入 HTTP 请求报文中；
- 服务器验证该用户名和密码，如果正确则把用户信息存储到 Redis 中，它在 Redis 中的 Key 称为 Session ID；
- 服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；
- 客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之前的业务操作。

注意：Session ID 的安全性问题，不能让它被恶意攻击者轻易获取，那么就不能产生一个容易被猜到的 Session ID 值。此外，还需要经常重新生成 Session ID。在对安全性要求极高的场景下，例如转账等操作，除了使用 Session 管理用户状态之外，还需要对用户进行重新验证，比如重新输入密码，或者使用短信验证码等方式。

43、Cookies 和 Session 区别是什么？

Cookie 和 Session 都是客户端与服务器之间保持状态的解决方案。

1、存储的位置不同

cookie：存放在客户端

session：存放在服务端。Session 存储的数据比较安全。

2、存储的数据类型不同

两者都是 key-value 的结构，但针对 value 的类型是有差异的。

cookie：value 只能是字符串类型，session：value 是 Object 类型

3、存储的数据大小限制不同

cookie：大小受浏览器的限制，很多是是 4K 的大小

session：理论上受当前内存的限制，

4、生命周期的控制

cookie 的生命周期当浏览器关闭的时候，就消亡了 (1)cookie 的生命周期是累计的，从创建时，就开始计时，20 分钟后，cookie 生命周期结束， (2)session 的生命周期是间隔的，从创建时，开始计时如在 20 分钟，没有访问 session，那么 session 生命周期被销毁

44、DDos 攻击了解吗？

客户端向服务端发送请求链接数据包，服务端向客户端发送确认数据包，客户端不向服务端发送确认数据包，服务器一直等待来自客户端的确认。

没有彻底根治的办法，除非不使用 TCP。

补充：DDos 预防的三种方法

- 1）限制同时打开 SYN 半链接的数目
- 2）缩短 SYN 半链接的 Time out 时间
- 3）关闭不必要的服务

45、MTU 和 MSS 分别是什么？

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。

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。

46、HTTP 中有个缓存机制，但如何保证缓存是最新的呢？（缓存过期机制）

max-age 指令出现在请求报文，并且缓存资源的缓存时间小于该指令指定的时间，那么就能接受该缓存。

max-age 指令出现在响应报文，表示缓存资源在缓存服务器中保存的时间。

```
Cache-Control: max-age=31536000
```

Expires 首部字段也可以用于告知缓存服务器该资源什么时候会过期。

```
Expires: Wed, 04 Jul 2012 08:26:05 GMT
```

- 在 HTTP/1.1 中，会优先处理 max-age 指令；
- 在 HTTP/1.0 中，max-age 指令会被忽略掉。

47、TCP 头部中有哪些信息？

序号（32bit）：传输方向上字节流的字节编号。初始时序号会被设置一个随机的初始值（ISN），之后每次发送数据时，序号值 = ISN + 数据在整个字节流中的偏移。假设 A -> B 且 ISN = 1024，第一段数据 512 字节已经到 B，则第二段数据发送时序号为 1024 + 512。用于解决网络包乱序问题。

确认号（32bit）：接收方对发送方 TCP 报文段的响应，其值是收到的序号值 + 1。

首部长（4bit）：标识首部有多少个 4 字节 * 首部长，最大为 15，即 60 字节。

标志位（6bit）：

URG：标志紧急指针是否有效。

ACK：标志确认号是否有效（确认报文段）。用于解决丢包问题。

PSH：提示接收端立即从缓冲读走数据。

RST：表示要求对方重新建立连接（复位报文段）。

SYN：表示请求建立一个连接（连接报文段）。

FIN：表示关闭连接（断开报文段）。

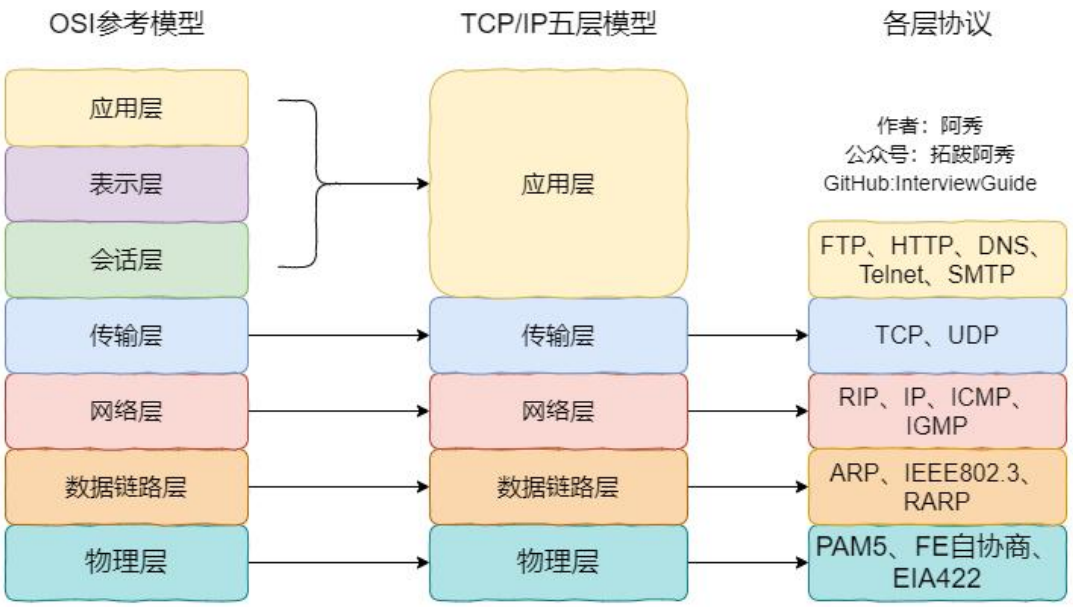
窗口（16bit）：接收窗口。用于告知对方（发送方）本方的缓冲还能接收多少字节数据。用于解决流控。

校验和（16bit）：接收端用 CRC 检验整个报文段有无损坏。

48、常见 TCP 的连接状态有哪些？

- CLOSED：初始状态。
- LISTEN：服务器处于监听状态。
- SYN_SEND：客户端 socket 执行 CONNECT 连接，发送 SYN 包，进入此状态。
- SYN_RECV：服务端收到 SYN 包并发送服务端 SYN 包，进入此状态。
- ESTABLISH：表示连接建立。客户端发送了最后一个 ACK 包后进入此状态，服务端接收到 ACK 包后进入此状态。
- FIN_WAIT_1：终止连接的一方（通常是客户机）发送了 FIN 报文后进入。等待对方 FIN。
- CLOSE_WAIT：（假设服务器）接收到客户机 FIN 包之后等待关闭的阶段。在接收到对方的 FIN 包之后，自然是需要立即回复 ACK 包的，表示已经知道断开请求。但是本方是否立即断开连接（发送 FIN 包）取决于是否还有数据需要发送给客户端，若有，则在发送 FIN 包之前均为此状态。
- FIN_WAIT_2：此时是半连接状态，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务器的 ACK 包，但并没有立即接收到服务端的 FIN 包，进入 FIN_WAIT_2 状态。
- LAST_ACK：服务端发动最后的 FIN 包，等待最后的客户端 ACK 响应，进入此状态。
- TIME_WAIT：客户端收到服务端的 FIN 包，并立即发出 ACK 包做最后的确认，在此之后的 2MSL 时间称为 TIME_WAIT 状态。

49、网络的七层/五层模型主要的协议有哪些？



50、TCP 是什么？

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

51、TCP 头部报文字段介绍几个？各自的功能？

source port 和 destination port

> 两者分别为「源端口号」和「目的端口号」。源端口号就是指本地端口，目的端口就是远程端口。

可以这么理解，我们有很多软件，每个软件都对应一个端口，假如，你想和我数据交互，咱们得互相知道你我的端口号。

再来一个很官方的：

> 扩展：应用程序的端口号和应用程序所在主机的 IP 地址统称为 socket（套接字），IP:端口号，在互联网上 socket 唯一标识每一个应用程序，源端口+源 IP+目的端口+目的 IP 称为"套接字对"，一对套接字就是一个连接，一个客户端与服务器之间的连接。

Sequence Number

> 称为「序列号」。用于 TCP 通信过程中某一传输方向上字节流的每个字节的编号，为了确保数据通信的有序性，避免网络中乱序的问题。接收端根据这个编号进行确认，保证分割的数据段在原始数据包的位置。初始序列号由自己定，而后绪的序列号由对端的 ACK 决定：SN_x = ACK_y (x 的序列号 = y 发给 x 的 ACK)。

说白了，类似于身份证一样，而且还得发送此时此刻的所在的位置，就相当于身份证上的地址一样。

Acknowledge Number

> 称为「确认序列号」。确认序列号是接收确认端所期望收到的下一序列号。确认序号应当是上次已成功收到数据字节序号加 1，只有当标志位中的 ACK 标志为 1 时该确认序列号的字段才有效。主要用来解决不丢包的问题。

TCP Flag

TCP 首部中有 6 个标志比特，它们中的多个可同时被设置为 1，主要是用于操控 TCP 的状态机的，依次为 URG，ACK，PSH，RST，SYN，FIN。

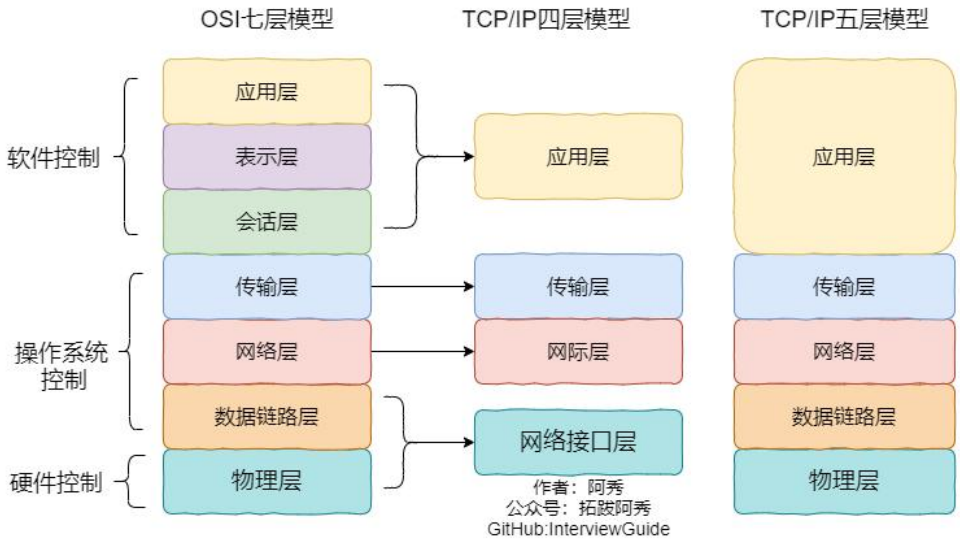
当然只介绍三个：

1. **ACK**：这个标识可以理解为发送端发送数据到接收端，发送的时候 ACK 为 0，标识接收端还未应答，一旦接收端接收数据之后，就将 ACK 置为 1，发送端接收到之后，就知道了接收端已经接收了数据。
2. **SYN**：表示「同步序列号」，是 TCP 握手的发送的第一个数据包。用来建立 TCP 的连接。SYN 标志位和 ACK 标志位搭配使用，当连接请求的时候，SYN=1，ACK=0 连接被响应的时候，SYN=1，ACK=1；这个标志的数据包经常被用来进行端口扫描。扫描者发送一个只有 SYN 的数据包，如果对方主机响应了一个数据包回来 ，就表明这台主机存在这个端口。
3. **FIN**：表示发送端已经达到数据末尾，也就是说双方的数据传送完成，没有数据可以传送了，发送 FIN 标志位的 TCP 数据包后，连接将被断开。这个标志的数据包也经常被用于进行端口扫描。发送端只剩最后的一段数据了，同时要告诉接收端后边没有数据可以接受了，所以用 FIN 标识一下，接收端看到这个 FIN 之后，哦！这是接受的最后的数据，接受完就关闭了；**TCP 四次分手必然问**。

Window size

> 称为滑动窗口大小。所说的滑动窗口，用来进行流量控制。

52、OSI 的七层模型的主要功能？



物理层：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。

数据链路层：接收来自物理层的位流形式的数据，并封装成帧，传送到上一层

网络层：将网络地址翻译成对应的物理地址，并通过路由选择算法为分组通过通信子网选择最适当的路径。

传输层：在源端与目的端之间提供可靠的透明数据传输

会话层：负责在网络中的两节点之间建立、维持和终止通信

表示层：处理用户信息的表示问题，数据的编码，压缩和解压缩，数据的加密和解密

应用层：为用户的应用进程提供网络通信服务

53、应用层常见协议知道多少？了解几个？

协议	名称	默认端口	底层协议
HTTP	超文本传输协议	80	TCP
HTTPS	超文本传输安全协议	443	TCP
Telnet	远程登录服务的标准协议	23	TCP
FTP	文件传输协议	20 传输和 21 连接	TCP
TFTP	简单文件传输协议	69	UDP
SMTP	简单邮件传输协议（发送用）	25	TCP

POP	邮局协议（接收用）	110	TCP
DNS	域名解析服务	53	服务器间进行域传输的时候用 TCP 客户端查询 DNS 服务器时用 UDP

54、浏览器在与服务器建立了一个 TCP 连接后是否会在一个 HTTP 请求完成后断开？什么情况下会断开？

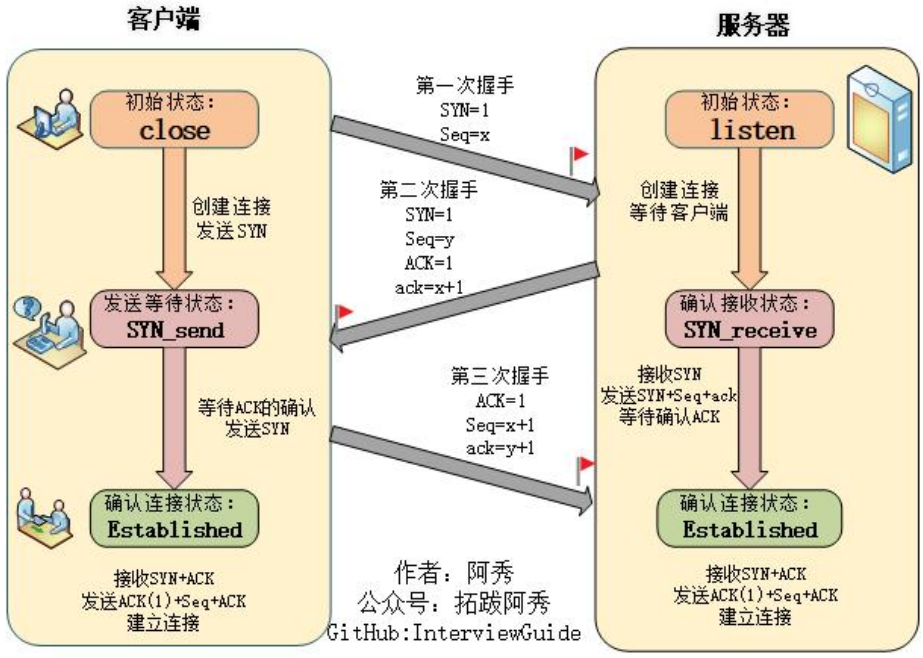
在 HTTP/1.0 中，一个服务器在发送完一个 HTTP 响应后，会断开 TCP 链接。但是这样每次请求都会重新建立和断开 TCP 连接，代价过大。所以虽然标准中没有设定，**某些服务器对 Connection: keep-alive 的 Header 进行了支持**。

意思是说，完成这个 HTTP 请求之后，不要断开 HTTP 请求使用的 TCP 连接。这样的好处是连接可以被重新使用，之后发送 HTTP 请求的时候不需要重新建立 TCP 连接，以及如果维持连接，那么 SSL 的开销也可以避免。

持久连接：既然维持 TCP 连接好处这么多，HTTP/1.1 就把 Connection 头写进标准，并且默认开启持久连接，除非请求中写明 Connection: close，那么浏览器和服务器之间是会维持一段时间的 TCP 连接，不会一个请求结束就断掉。

默认情况下建立 TCP 连接不会断开，只有在请求报头中声明 Connection: close 才会在请求完成后关闭连接。

55、三次握手相关内容



三次握手（Three-way Handshake）其实就是指建立一个 TCP 连接时，需要客户端和服务器总共发送 3 个包。进行三次握手的主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备。

实质上其实就是连接服务器指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号，交换 TCP 窗口大小信息。

第一种回答

刚开始客户端处于 Closed 的状态，服务端处于 Listen 状态，进行三次握手：

- 第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN(c)。此时客户端处于 SYN_SEND 状态。

  首部的同步位 SYN=1，初始序号 seq=x，SYN=1 的报文段不能携带数据，但要消耗掉一个序号。

- 第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)。同时会把客户端的 ISN + 1 作为 ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 SYN_RCVD 的状态。

  在确认报文段中 SYN=1，ACK=1，确认号 ack=x+1，初始序号 seq=y。

- 第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立起了连接。

  确认报文段 ACK=1，确认号 ack=y+1，序号 seq=x+1（初始为 seq=x，第二个报文段所以要+1），ACK 报文段可以携带数据，不携带数据则不消耗序号。

发送第一个 SYN 的一端将执行主动打开（active open），接收这个 SYN 并发回下一个 SYN 的另一端执行被动打开（passive open）。

在 socket 编程中，客户端执行 connect()时，将触发三次握手。

第二种回答

- **初始状态**：客户端处于 closed(关闭)状态，服务器处于 listen(监听) 状态。
- **第一次握手**：客户端发送请求报文将 SYN = 1 同步序列号和初始化序列号 seq = x 发送给服务端，发送完之后客户端处于 SYN_Send 状态。（验证了客户端的发送能力和服务端的接收能力）
- **第二次握手**：服务端受到 SYN 请求报文之后，如果同意连接，会以自己的同步序列号 SYN(服务端) = 1、初始化序列号 seq = y 和确认序列号（期望下次收到的数据包）ack = x+ 1 以及确认号 ACK = 1 报文作为应答，服务器为 SYN_Receive 状态。（问题来了，两次握手之后，站在客户端角度上思考：我发送和接收都 ok，服务端的发送和接收也都 ok。但是站在服务端的角度思考：哎呀，我服务端接收 ok，但是我不清楚我的发送 ok 不 ok 呀，而且我还不知道你接受能力如何呢？所以老哥，你需要给我三次握手来传个话告诉我一声。你要是不告诉我，万一我认为你跑了，然后我可能出于安全性的考虑继续给你发一次，看看你回不回我。）

- **第三次握手：** 客户端接收到服务端的 SYN + ACK 之后，知道可以下次可以发送了下一序列的数据包了，然后发送同步序列号 ack = y + 1 和数据包的序列号 seq = x + 1 以及确认号 ACK = 1 确认包作为应答，客户端转为 established 状态。（分别站在双方的角度上思考，各自 ok）

试想如果是用两次握手，则会出现下面这种情况：

> 如客户端发出连接请求，但因连接请求报文丢失而未收到确认，于是客户端再重传一次连接请求。后来收到了确认，建立了连接。
>
> 数据传输完毕后，就释放了连接，客户端共发出了两个连接请求报文段，其中第一个丢失，第二个到达了服务端，但是第一个丢失的报文段只是在**某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达服务端**。
>
> 此时服务端误认为客户端又发出一次新的连接请求，于是就向客户端发出确认报文段，同意建立连接，不采用三次握手，只要服务端发出确认，就建立新的连接了，此时客户端忽略服务端发来的确认，也不发送数据，则服务端一致等待客户端发送数据，浪费资源。

57、什么是半连接队列？

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其连接，服务器会把此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**。

当然还有一个**全连接队列**，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满了就有可能会出现丢包现象。

这里在补充一点关于 **SYN-ACK 重传次数**的问题： 服务器发送完 SYN-ACK 包，如果未收到客户确认包，服务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传。

如果重传次数超过系统规定的最大重传次数，系统将该连接信息从半连接队列中删除。 注意，每次重传等待的时间不一定相同，一般会是指数增长，例如间隔时间为 1s，2s，4s，8s……

58、 ISN(Initial Sequence Number)是固定的吗？

当一端为建立连接而发送它的 SYN 时，它为连接选择一个初始序号。ISN 随时间而变化，因此每个连接都将具有不同的 ISN，ISN 是一个有可以看作是一个 32 比特的计数器，但并不是简单的计数器，大概每 4ms 加 1 。

> ISN = M + F(localhost, localport, remotehost, remoteport)(M 为计数器)，
> ISN 应该由这个公式确定，F 为哈希算法，不是一个简单计数器。

这样选择序号的目的在于防止在网络中被延迟的分组在以后又被传送，而导致某个连接的一方对它做错误的解释。

三次握手的其中一个重要功能是客户端和服务端交换 ISN(Initial Sequence Number)，以便让对方知道接下来接收数据的时候如何按序列号组装数据。如果 ISN 是固定的，攻击者很容易猜出后续的确认号，因此 ISN 是动态生成的。

59、 三次握手过程中可以携带数据吗？

其实第三次握手的时候，是可以携带数据的。但是，**第一次、第二次握手不可以携带数据**

为什么这样呢?大家可以想一个问题，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。

因为攻击者根本就不理服务器的接收、发送能力是否正常，然后疯狂着重复发 SYN 报文的话，这会让服务器花费很多时间、内存空间来接收这些报文。

也就是说，**第一次握手不可以放数据，其中一个简单的原因就是会让服务器更加容易受到攻击了。而对于第三次的话，此时客户端已经处于 ESTABLISHED 状态。**

对于客户端来说，他已经建立起连接了，并且也已经知道服务器的接收、发送能力是正常的了，所以能携带数据也没啥毛病。

60、SYN 攻击是什么？

服务器端的资源分配是在二次握手时分配的，而客户端的资源是在完成三次握手时分配的，所以服务器容易受到 SYN 洪泛攻击。

SYN 攻击就是 Client 在短时间内伪造大量不存在的 IP 地址，并向 Server 不断地发送 SYN 包，Server 则回复确认包，并等待 Client 确认。

由于源地址不存在，因此 Server 需要不断重发直至超时，这些伪造的 SYN 包将长时间占用未连接队列，导致正常的 SYN 请求因为队列满而被丢弃，从而引起网络拥塞甚至系统瘫痪。

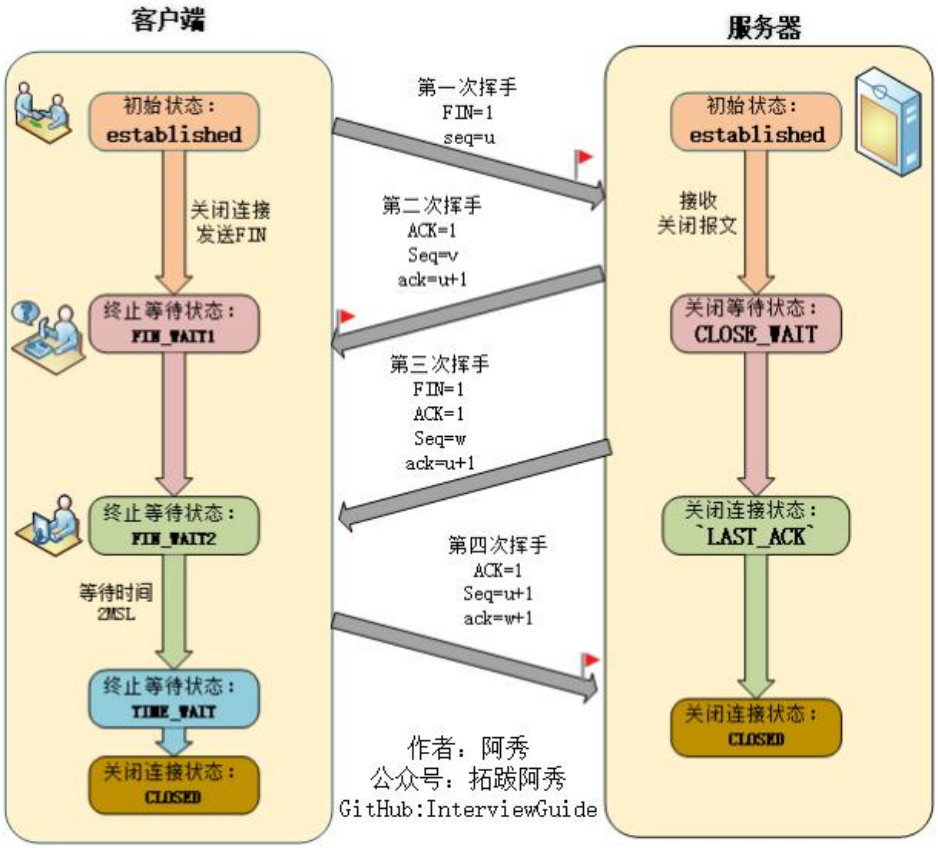
SYN 攻击是一种典型的 DoS/DDoS 攻击。检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源 IP 地址是随机的，基本上可以断定这是一次 SYN 攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

```
netstat -n -p TCP | grep SYN_RECV
```
复制代码

常见的防御 SYN 攻击的方法有如下几种：

- 缩短超时（SYN Timeout）时间
- 增加最大半连接数
- 过滤网关防护
- SYN cookies 技术

61、 四次挥手相关内容



建立一个连接需要三次握手，而终止一个连接要经过四次挥手（也有将四次挥手叫做四次握手的）。这由 TCP 的**半关闭**（half-close）造成的。所谓的半关闭，其实就是 TCP 提供了连接的一端在结束它的发送后还能接收来自另一端数据的能力。

TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，客户端或服务器均可主动发起挥手动作。

第一种回答

刚开始双方都处于 ESTABLISHED 状态，假如是客户端先发起关闭请求。四次挥手的过程如下：

- 第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于 FIN_WAIT1 状态。 即发出**连接释放报文段**（FIN=1，序号 seq=u），并停止再发送数据，主动关闭 TCP 连接，进入 FIN_WAIT1（终止等待 1）状态，等待服务端的确认。
- 第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 CLOSE_WAIT 状态。即服务端收到连接释放报文段后即发出**确认报文段**（ACK=1，

确认号 ack=u+1，序号 seq=v），服务端进入 CLOSE_WAIT（关闭等待）状态，此时的 TCP 处于半关闭状态，客户端到服务端的连接释放。客户端收到服务端的确认后，进入 FIN_WAIT2（终止等待 2）状态，等待服务端发出的连接释放报文段。

- 第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 LAST_ACK 的状态。 即服务端没有要向客户端发出的数据，服务端发出**连接释放报文段**（FIN=1，ACK=1，序号 seq=w，确认号 ack=u+1），服务端进入 LAST_ACK（最后确认）状态，等待客户端的确认。
- 第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1 作为自己 ACK 报文的确认号值，此时客户端处于 TIME_WAIT 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。 即客户端收到服务端的连接释放报文段后，对此发出**确认报文段**（ACK=1，seq=u+1，ack=w+1），客户端进入 TIME_WAIT（时间等待）状态。此时 TCP 未释放掉，需要经过时间等待计时器设置的时间 2MSL 后，客户端才进入 CLOSED 状态。

收到一个 FIN 只意味着在这一方向上没有数据流动。**客户端执行主动关闭并进入 TIME_WAIT 是正常的，服务端通常执行被动关闭，不会进入 TIME_WAIT 状态。**

在 socket 编程中，任何一方执行 close()操作即可产生挥手操作。

第二种回答

- **初始化状态**：客户端和服务端都在连接状态，接下来开始进行四次分手断开连接操作。
- **第一次分手**：第一次分手无论是客户端还是服务端都可以发起，因为 TCP 是全双工的。

  假如客户端发送的数据已经发送完毕，发送 FIN = 1 **告诉服务端，客户端所有数据已经全发完了，服务端你可以关闭接收了**，但是如果你们服务端有数据要发给客户端，客户端照样可以接收的。此时客户端处于 FIN = 1 等待服务端确认释放连接状态。

- **第二次分手**：服务端接收到客户端的释放请求连接之后，**知道客户端没有数据要发给自己了，然后服务端发送 ACK = 1 告诉客户端收到你发给我的信息**，此时服务端处于 CLOSE_WAIT 等待关闭状态。（服务端先回应给客户端一声，我知道了，但服务端的发送数据能力即将等待关闭，于是接下来第三次就来了。）
- **第三次分手**：此时服务端向客户端把所有的数据发送完了，然后发送一个 FIN = 1，**用于告诉客户端，服务端的所有数据发送完毕，客户端你也可以关闭接收数据连接了**。此时服务端状态处于 LAST_ACK 状态，来等待确认客户端是否收到了自己的请求。（服务端等客户端回复是否收到呢，不收到的话，服务端不知道客户端是不是挂掉了还是咋回事呢，所以服务端不敢关闭自己的接收能力，于是第四次就来了。）
- **第四次分手**：此时如果客户端收到了服务端发送完的信息之后，就发送 ACK = 1，告诉服务端，客户端已经收到了你的信息。**有一个 2 MSL 的延迟等待**。

62、挥手为什么需要四次？

第一种回答

因为当服务端收到客户端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中 **ACK 报文是用来应答的，SYN 报文是用来同步的**。但是关闭连接时，当服务端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉客户端，"你发的 FIN 报文我收到了"。只有等到我服务端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四次挥手。

第二种回答

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了 TCP 连接。举个例子：A 和 B 打电话，通话即将结束后，A 说“我没啥要说的了”，B 回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”，A 回答“知道了”，这样通话才算结束。

63、2MSL 等待状态？

TIME_WAIT 状态也称为 2MSL 等待状态。每个具体 TCP 实现必须选择一个报文段最大生存时间 MSL（Maximum Segment Lifetime），它是任何报文段被丢弃前在网络内的最长时间。这个时间是有限的，因为 TCP 报文段以 IP 数据报在网络内传输，而 IP 数据报则有限制其生存时间的 TTL 字段。

对一个具体实现所给定的 MSL 值，处理的原则是：当 TCP 执行一个主动关闭，并发回最后一个 ACK，该连接必须在 TIME_WAIT 状态停留的时间为 2 倍的 MSL。这样可让 TCP 再次发送最后的 ACK 以防这个 ACK 丢失（另一端超时并重发最后的 FIN）。

这种 2MSL 等待的另一个结果是这个 TCP 连接在 2MSL 等待期间，定义这个连接的插口（客户的 IP 地址和端口号，服务器的 IP 地址和端口号）不能再被使用。这个连接只能在 2MSL 结束后才能再被使用。

67、OSI 七层模型中表示层和会话层功能是什么？

表示层：图像、视频编码解，数据加密。

会话层：建立会话，如 session 认证、断点续传。

64、四次挥手释放连接时，等待 2MSL 的意义?

> **MSL** 是 Maximum Segment Lifetime 的英文缩写，可译为“最长报文段寿命”，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。

为了保证客户端发送的最后一个 ACK 报文段能够到达服务器。因为这个 ACK 有可能丢失，从而导致处在 LAST-ACK 状态的服务器收不到对 FIN-ACK 的确认报文。服务器会超时重传这个 FIN-ACK，接着客户端再重传一次确认，重新启动时间等待计时器。最后客户端和服务器都能正常的关闭。假设客户端不等待 2MSL，而是在发送完 ACK 之后直接释放关闭，一但这个 ACK 丢失的话，服务器就无法正常的进入关闭连接状态。

两个理由

1、保证客户端发送的最后一个 ACK 报文段能够到达服务端。

这个 ACK 报文段有可能丢失，使得处于 LAST-ACK 状态的 B 收不到对已发送的 FIN+ACK 报文段的确认，服务端超时重传 FIN+ACK 报文段，而客户端能在 2MSL 时间内收到这个重传的 FIN+ACK 报文段。

接着客户端重传一次确认，重新启动 2MSL 计时器。最后客户端和服务端都进入到 CLOSED 状态，若客户端在 TIME-WAIT 状态不等待一段时间，而是发送完 ACK 报文段后立即释放连接，则无法收到服务端重传的 FIN+ACK 报文段，所以不会再发送一次确认报文段，则服务端无法正常进入到 CLOSED 状态。

2、防止“已失效的连接请求报文段”出现在本连接中。

客户端在发送完最后一个 ACK 报文段后，再经过 2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失，使下一个新的连接中不会出现这种旧的连接请求报文段。

65、为什么 TIME_WAIT 状态需要经过 2MSL 才能返回到 CLOSE 状态？

第一种回答

理论上，四个报文都发送完毕，就可以直接进入 CLOSE 状态了，但是可能网络是不可靠的，有可能最后一个 ACK 丢失。所以 **TIME_WAIT 状态就是用来重发可能丢失的 ACK 报文**。

第二种回答

对应这样一种情况，最后客户端发送的 ACK = 1 给服务端的**过程中丢失**了，服务端没收到，服务端怎么认为的？

我已经发送完数据了，怎么客户端没回应我？是不是中途丢失了？然后服务端再次发起断开连接的请求，一个来回就是 2MSL。

客户端给服务端发送的 ACK = 1 丢失，**服务端等待 1MSL 没收到，然后重新发送消息需要 1MSL**。如果再次接收到服务端的消息，则**重启 2MSL 计时器，发送确认请求**。

客户端只需等待 2MSL，如果没有再次收到服务端的消息，就说明服务端已经接收到自己确认消息；此时双方都关闭的连接，TCP 四次分手完毕

66、TCP 粘包问题是什么？你会如何去解决它？

TCP 粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

- 由 TCP **连接复用**造成的粘包问题。
- 因为 TCP 默认会使用 **Nagle 算法**，此算法会导致粘包问题。
    - 只有上一个分组得到确认，才会发送下一个分组；
    - 收集多个小分组，在一个确认到来时一起发送。
- **数据包过大**造成的粘包问题。
- 流量控制，**拥塞控制**也可能导致粘包。
- **接收方不及时接收缓冲区的包，造成多个包接收**

解决：

1. **Nagle 算法**问题导致的，需要结合应用场景适当关闭该算法
2. 尾部标记序列。通过特殊标识符表示数据包的边界，例如\n\r，\t，或者一些隐藏字符。
3. 头部标记分步接收。在 TCP 报文的头部加上表示数据长度。
4. 应用层发送数据时**定长**发送。

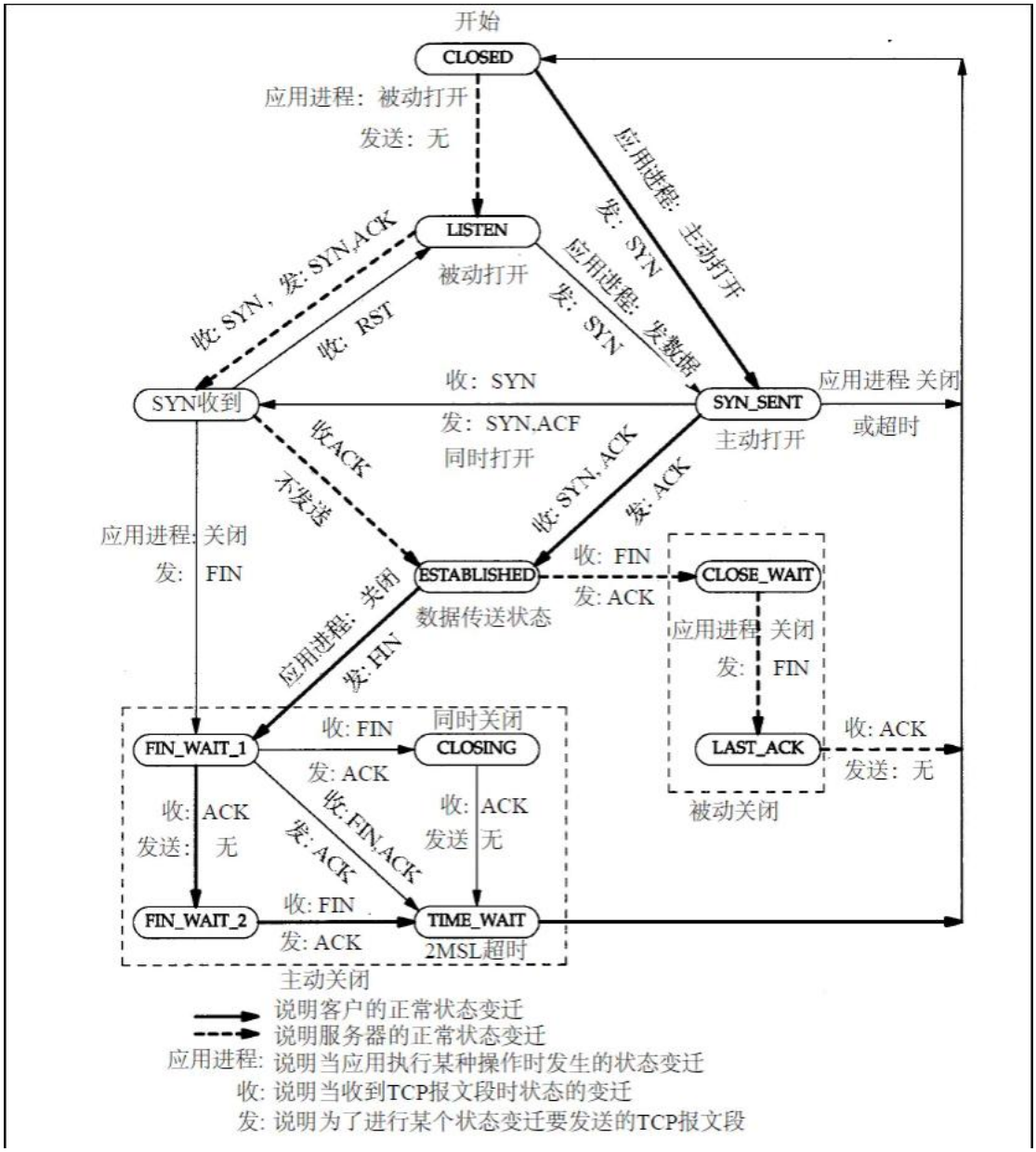
69、对称密钥加密的优点缺点？

对称密钥加密（Symmetric-Key Encryption），加密和解密使用同一密钥。

- 优点：运算速度快
- 缺点：无法安全地将密钥传输给通信方

68、三次握手四次挥手的变迁图

《TCP/IP 详解 卷 1:协议》有一张 TCP 状态变迁图，很具有代表性，有助于大家理解三次握手和四次挥手的状态变化。如下图所示，粗的实线箭头表示正常的客户端状态变迁，粗的虚线箭头表示正常的服务器状态变迁。



71、HTTPS 是什么？

HTTPS 并不是新协议，而是让 **HTTP 先和 SSL（Secure Sockets Layer）通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信。**

通过使用 SSL，HTTPS 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）。

70、非对称密钥加密你了解吗？优缺点？

非对称密钥加密，又称公开密钥加密（Public-Key Encryption），加密和解密使用不同的密钥。

公开密钥所有人都可以获得，**通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密，接收方收到通信内容后使用私有密钥解密**。

非对称密钥除了用来加密，还可以用来进行签名。

因为私有密钥无法被其他人获取，因此通信发送方使用其私有密钥进行签名，通信接收方使用发送方的公开密钥对签名进行解密，就能判断这个签名是否正确。

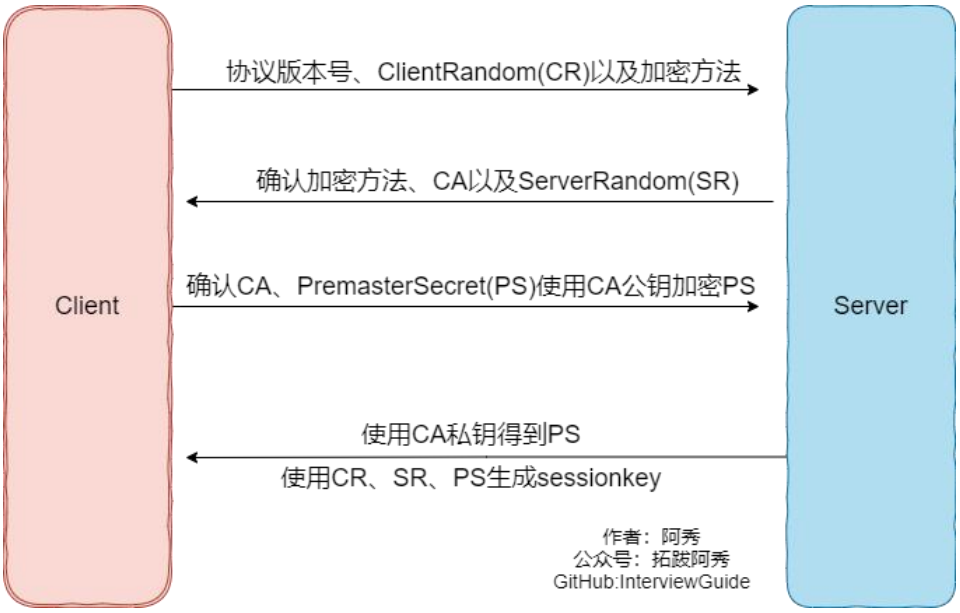
- 优点：可以更安全地将公开密钥传输给通信发送方；
- 缺点：运算速度慢。

72、HTTP 的缺点有哪些？

- 使用明文进行通信，内容可能会被窃听；
- 不验证通信方的身份，通信方的身份有可能遭遇伪装；
- 无法证明报文的完整性，报文有可能遭篡改。

73、HTTPS 采用的加密方式有哪些？是对称还是非对称？

HTTPS 采用混合的加密机制，使用**非对称密钥加密用于传输对称密钥来保证传输过程的安全性**，之后使用**对称密钥加密进行通信来保证通信过程的效率**。



确保传输安全过程（其实就是 rsa 原理）：

1. Client 给出协议版本号、一个客户端生成的随机数（Client random），以及客户端支持的加密方法。
2. Server 确认双方使用的加密方法，并给出数字证书、以及一个服务器生成的随机数（Server random）。
3. Client 确认数字证书有效，然后生成一个新的随机数（Premaster secret），并使用数字证书中的公钥，加密这个随机数，发给 Server。
4. Server 使用自己的私钥，获取 Client 发来的随机数（Premaster secret）。
5. Client 和 Server 根据约定的加密方法，使用前面的三个随机数，生成"对话密钥"（session key），用来加密接下来的整个对话过程。

74、为什么有的时候刷新页面不需要重新建立 SSL 连接？

TCP 连接有的时候会被浏览器和服务端维持一段时间，TCP 不需要重新建立，SSL 自然也会用之前的。

75、SSL 中的认证中的证书是什么？了解过吗？

通过使用 **证书** 来对通信方进行认证。

数字证书认证机构（CA，Certificate Authority）是客户端与服务器双方都可信赖的第三方机构。

服务器的运营人员向 CA 提出公开密钥的申请，CA 在判明提出申请者的身份之后，会对已申请的公开密钥做数字签名，然后分配这个已签名的公开密钥，并将该公开密钥放入公开密钥证书后绑定在一起。

进行 HTTPS 通信时，服务器会把证书发送给客户端。客户端取得其中的公开密钥之后，先使用数字签名进行验证，如果验证通过，就可以开始通信了。

76、HTTP 如何禁用缓存？如何确认缓存？

HTTP/1.1 通过 Cache-Control 首部字段来控制缓存。

禁止进行缓存

no-store 指令规定不能对请求或响应的任何一部分进行缓存。

```
Cache-Control: no-store
```

强制确认缓存

no-cache 指令规定缓存服务器需要先向源服务器验证缓存资源的有效性，只有当缓存资源有效时才能使用该缓存对客户端的请求进行响应。

```
Cache-Control: no-cache
```

77、GET 与 POST 传递数据的最大长度能够达到多少呢？

get 是通过 URL 提交数据，因此 GET 可提交的数据量就跟 URL 所能达到的最大长度有直接关系。

很多文章都说 GET 方式提交的数据最多只能是 1024 字节，而实际上，URL 不存在参数上限的问题，HTTP 协议规范也没有对 URL 长度进行限制。

这个限制是特定的浏览器及服务器对它的限制，比如 IE 对 URL 长度的限制是 2083 字节(2K+35 字节)。

对于其他浏览器，如 FireFox，Netscape 等，则没有长度限制，这个时候其限制取决于服务器的操作系统；即如果 url 太长，服务器可能会因为安全方面的设置从而拒绝请求或者发生不完整的数据请求。

post 理论上讲是没有大小限制的，HTTP 协议规范也没有进行大小限制，但实际上 post 所能传递的数据量大小取决于服务器的设置和内存大小。

因为我们一般 post 的数据量很少超过 MB 的，所以我们很少能感觉的到 post 的数据量限制，但实际中如果你上传文件的过程中可能会发现这样一个问题，即上传个头比较大的文件到服务器时候，可能上传不上去。

以 php 语言来说，查原因的时候你也许会看到有说 PHP 上传文件涉及到的参数 PHP 默认的上传有限定，一般这个值是 2MB，更改这个值需要更改 php.conf 的 post_max_size 这个值。这就很明白的说明了这个问题了。

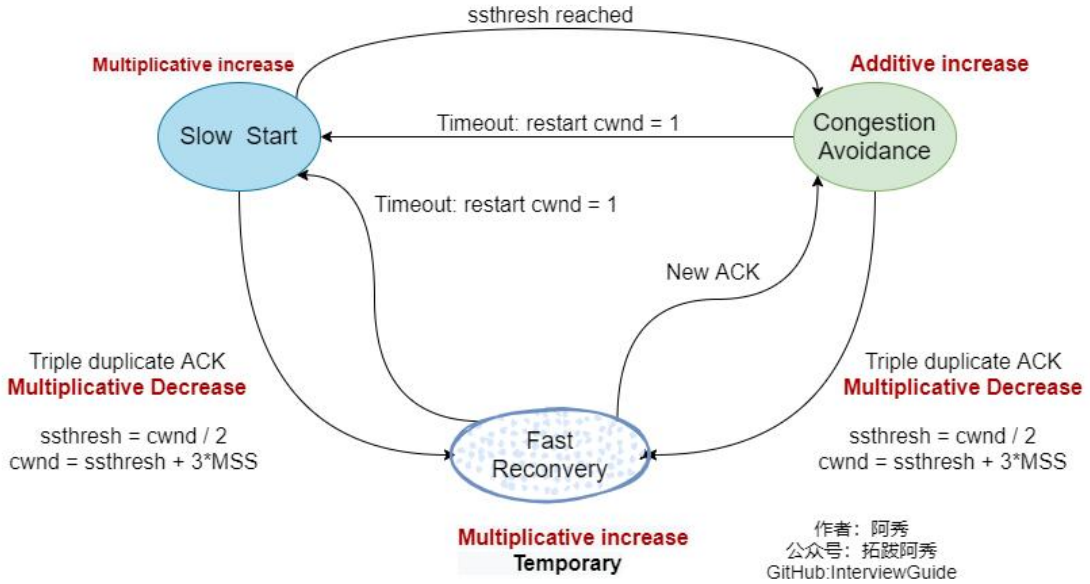
78、网络层常见协议？可以说一下吗？

协议	名称	作用
IP	网际协议	IP 协议不但定义了数据传输时的基本单元和格式，还定义了数据报的递交方法和路由选择
ICMP	Internet 控制报文协议	ICMP 就是一个“错误侦测与回报机制”，其目的就是让我们能够检测网路的连线状况，也能确保连线的准确性，是 ping 和 traceroute 的工作协议
RIP	路由信息协议	使用“跳数”(即 metric)来衡量到达目标地址的路由距离
IGMP	Internet 组管理协议	用于实现组播、广播等通信

79、TCP 四大拥塞控制算法总结？（极其重要）

四大算法

拥塞控制主要是四个算法：1）慢启动，2）拥塞避免，3）拥塞发生，4）快速恢复。这四个算法不是一天都搞出来的，这个四算法的发展经历了很多时间，到今天都还在优化中。



慢热启动算法 – Slow Start

所谓慢启动，也就是 TCP 连接刚建立，一点一点地提速，试探一下网络的承受能力，以免直接扰乱了网络通道的秩序。

慢启动算法：

1. 连接建好的开始先初始化拥塞窗口 cwnd 大小为 1，表明可以传一个 MSS 大小的数据。
2. 每当收到一个 ACK，cwnd 大小加一，呈线性上升。
3. 每当过了一个往返延迟时间 RTT(Round-Trip Time)，cwnd 大小直接翻倍，乘以 2，呈指数让升。
4. 还有一个 ssthresh（slow start threshold），是一个上限，当 cwnd >= ssthresh 时，就会进入"拥塞避免算法"（后面会说这个算法）

拥塞避免算法 – Congestion Avoidance

如同前边说的，当拥塞窗口大小 cwnd 大于等于慢启动阈值 ssthresh 后，就进入拥塞避免算法。算法如下：

1. 收到一个 ACK，则 cwnd = cwnd + 1 / cwnd
2. 每当过了一个往返延迟时间 RTT，cwnd 大小加一。

过了慢启动阈值后，拥塞避免算法可以避免窗口增长过快导致窗口拥塞，而是缓慢的增加调整到网络的最佳值。

拥塞发生状态时的算法

一般来说，TCP 拥塞控制默认认为网络丢包是由于网络拥塞导致的，所以一般的 TCP 拥塞控制算法以丢包为网络进入拥塞状态的信号。

对于丢包有两种判定方式，一种是超时重传 RTO[Retransmission Timeout]超时，另一个是收到三个重复确认 ACK。

超时重传是 TCP 协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时器，在一定时间内如果没有得到发送数据报的 ACK 报文，那么就重新发送数据，直到发送成功为止。

但是如果发送端接收到 3 个以上的重复 ACK，TCP 就意识到数据发生丢失，需要重传。这个机制不需要等到重传定时器超时，所以叫做快速重传，而快速重传后没有使用慢启动算法，而是拥塞避免算法，所以这又叫做快速恢复算法。

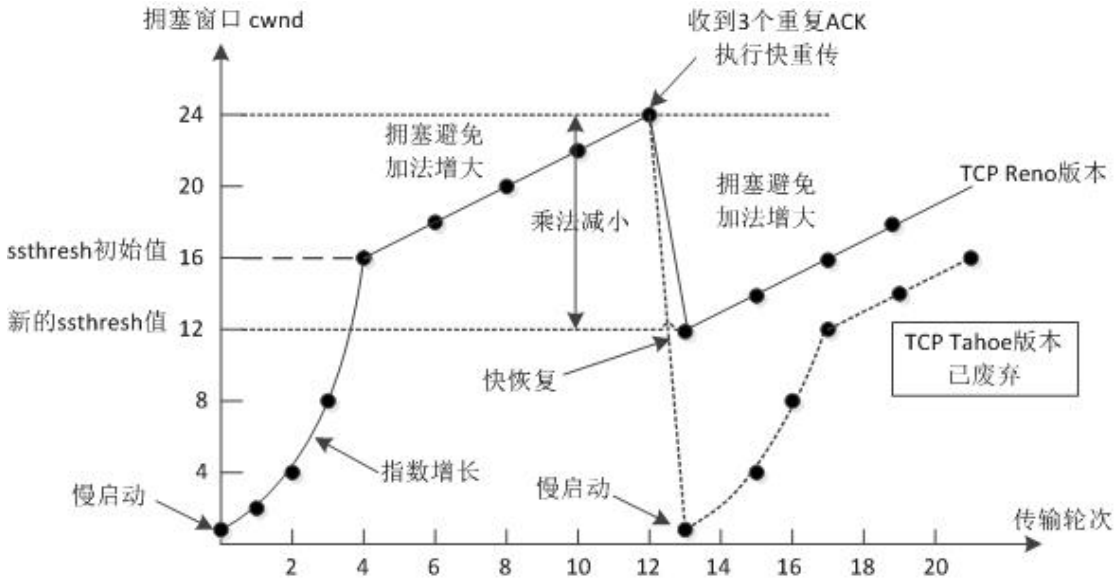
超时重传 RTO[Retransmission Timeout]超时，TCP 会重传数据包。TCP 认为这种情况比较糟糕，反应也比较强烈：

- 由于发生丢包，将慢启动阈值 ssthresh 设置为当前 cwnd 的一半，即 ssthresh = cwnd / 2.
- cwnd 重置为 1
- 进入慢启动过程

最为早期的 TCP Tahoe 算法就只使用上述处理办法，但是由于一丢包就一切重来，导致 cwnd 又重置为 1，十分不利于网络数据的稳定传递。

所以，TCP Reno 算法进行了优化。当收到三个重复确认 ACK 时，TCP 开启快速重传 Fast Retransmit 算法，而不用等到 RTO 超时再进行重传：

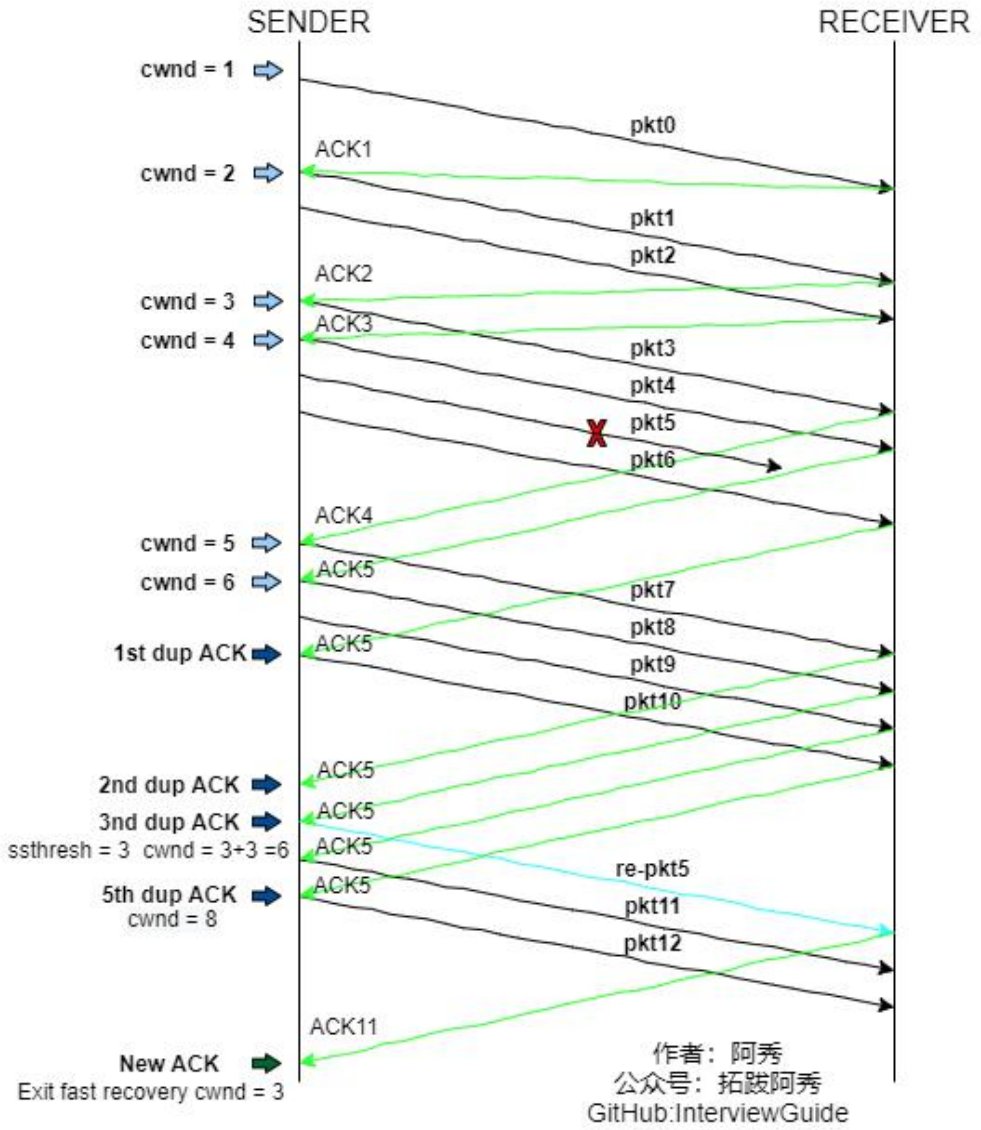
- cwnd 大小缩小为当前的一半
- ssthresh 设置为缩小后的 cwnd 大小
- 然后进入快速恢复算法 Fast Recovery。



快速恢复算法 – Fast Recovery

TCP Tahoe 是早期的算法，所以没有快速恢复算法，而 Reno 算法有。在进入快速恢复之前，cwnd 和 ssthresh 已经被更改为原有 cwnd 的一半。快速恢复算法的逻辑如下：

> cwnd = cwnd + 3 MSS，加 3 MSS 的原因是因为收到 3 个重复的 ACK。
>
> 重传 DACKs 指定的数据包。
>
> 如果再收到 DACKs，那么 cwnd 大小增加一。
>
> 如果收到新的 ACK，表明重传的包成功了，那么退出快速恢复算法。将 cwnd 设置为 ssthresh，然后进入拥塞避免算法



如图所示，第五个包发生了丢失，所以导致接收方接收到三次重复 ACK，也就是 ACK5。所以将 ssthresh 设置当当时 cwnd 的一半，也就是 6/2 = 3，cwnd 设置为 3 + 3 = 6。

然后重传第五个包。当收到新的 ACK 时，也就是 ACK11，则退出快速恢复阶段，将 cwnd 重新设置为当前的 ssthresh，也就是 3，然后进入拥塞避免算法阶段。

80、为何快速重传是选择 3 次 ACK？

主要的考虑还是要区分包的丢失是由于链路故障还是乱序等其他因素引发。

两次 duplicated ACK 时很可能是乱序造成的！三次 duplicated ACK 时很可能是丢包造成的！四次 duplicated ACK 更更更可能是丢包造成的，但是这样的响应策略太慢。

丢包肯定会造成三次 duplicated ACK!综上是选择收到三个重复确认时窗口减半效果最好，这是实践经验。

在没有 fast retransmit / recovery 算法之前，重传依靠发送方的 retransmit timeout，就是在 timeout 内如果没有接收到对方的 ACK，默认包丢了，发送方就重传，包的丢失原因

1）包 checksum 出错

2）网络拥塞

3）网络断，包括路由重收敛，但是发送方无法判断是哪一种情况，于是采用最笨的办法，就是将自己的发送速率减半，即 CWND 减为 1/2，这样的方法对 2 是有效的，可以缓解网络拥塞，3 则无所谓，反正网络断了，无论发快发慢都会被丢；但对于 1 来说，丢包是因为偶尔的出错引起，一丢包就对半减速不合理。

于是有了 fast retransmit 算法，基于在反向还可以接收到 ACK，可以认为网络并没有断，否则也接收不到 ACK，如果在 timeout 时间内没有接收到> 2 的 duplicated ACK，则概率大事件为乱序，乱序无需重传，接收方会进行排序工作；

而如果接收到三个或三个以上的 duplicated ACK，则大概率是丢包，可以逻辑推理，发送方可以接收 ACK，则网络是通的，可能是 1、2 造成的，先不降速，重传一次，如果接收到正确的 ACK，则一切 OK，流速依然（包出错被丢）。

而如果依然接收到 duplicated ACK，则认为是网络拥塞造成的，此时降速则比较合理。

80、为何快速重传是选择 3 次 ACK？

主要的考虑还是要区分包的丢失是由于链路故障还是乱序等其他因素引发。

两次 duplicated ACK 时很可能是乱序造成的！三次 duplicated ACK 时很可能是丢包造成的！四次 duplicated ACK 更更更可能是丢包造成的，但是这样的响应策略太慢。丢包肯定会造成三次 duplicated ACK!综上是选择收到三个重复确认时窗口减半效果最好，这是实践经验。

在没有 fast retransmit / recovery 算法之前，重传依靠发送方的 retransmit timeout，就是在 timeout 内如果没有接收到对方的 ACK，默认包丢了，发送方就重传，包的丢失原因。

1）包 checksum 出错

2）网络拥塞

3）网络断，包括路由重收敛，但是发送方无法判断是哪一种情况，于是采用最笨的办法，就是将自己的发送速率减半，即 CWND 减为 1/2，这样的方法对 2 是有效的，可以缓解网络拥塞，3 则无所谓，反正网络断了，无论发快发慢都会被丢；但对于 1 来说，丢包是因为偶尔的出错引起，一丢包就对半减速不合理。

于是有了 fast retransmit 算法，基于在反向还可以接收到 ACK，可以认为网络并没有断，否则也接收不到 ACK，如果在 timeout 时间内没有接收到> 2 的 duplicated ACK，则概率大事件为乱序，乱序无需重传，接收方会进行排序工作；

而如果接收到三个或三个以上的 duplicated ACK，则大概率是丢包，可以逻辑推理，发送方可以接收 ACK，则网络是通的，可能是 1、2 造成的，先不降速，重传一次，如果接收到正确的 ACK，则一切 OK，流速依然（包出错被丢）。

而如果依然接收到 duplicated ACK，则认为是网络拥塞造成的，此时降速则比较合理。

81、对于 FIN_WAIT_2，CLOSE_WAIT 状态和 TIME_WAIT 状态？你知道多少?

FIN_WAIT_2：

半关闭状态。

发送断开请求一方还有接收数据能力，但已经没有发送数据能力。

CLOSE_WAIT 状态：

被动关闭连接一方接收到 FIN 包会立即回应 ACK 包表示已接收到断开请求。

被动关闭连接一方如果还有剩余数据要发送就会进入 CLOSE_WAIT 状态。

TIME_WAIT 状态：

- 又叫 2MSL 等待状态。
- 如果客户端直接进入 CLOSED 状态，如果服务端没有接收到最后一次 ACK 包会在超时之后重新再发 FIN 包，此时因为客户端已经 CLOSED，所以服务端就不会收到 ACK 而是收到 RST。所以 TIME_WAIT 状态目的是防止最后一次握手数据没有到达对方而触发重传 FIN 准备的。
- 在 2MSL 时间内，同一个 socket 不能再被使用，否则有可能会和旧连接数据混淆（如果新连接和旧连接的 socket 相同的话）。

82、你了解流量控制原理吗？

目的是接收方通过 TCP 头窗口字段告知发送方本方可接收的最大数据量，用以解决发送速率过快导致接收方不能接收的问题。所以流量控制是点对点控制。

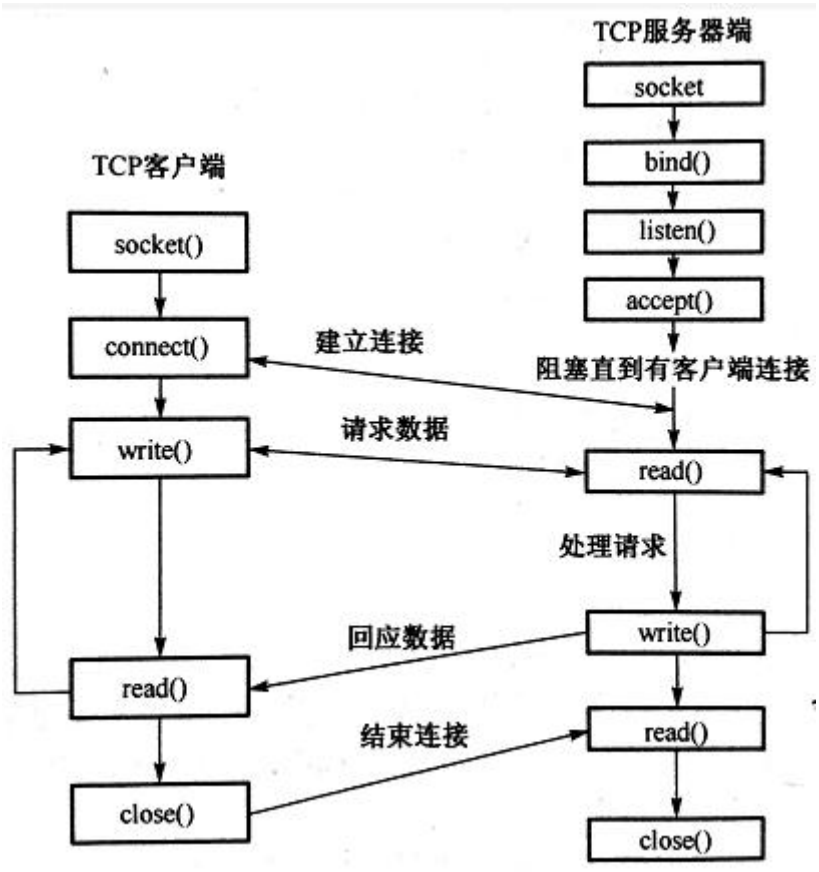
TCP 是双工协议，双方可以同时通信，所以发送方接收方各自维护一个发送窗和接收窗。

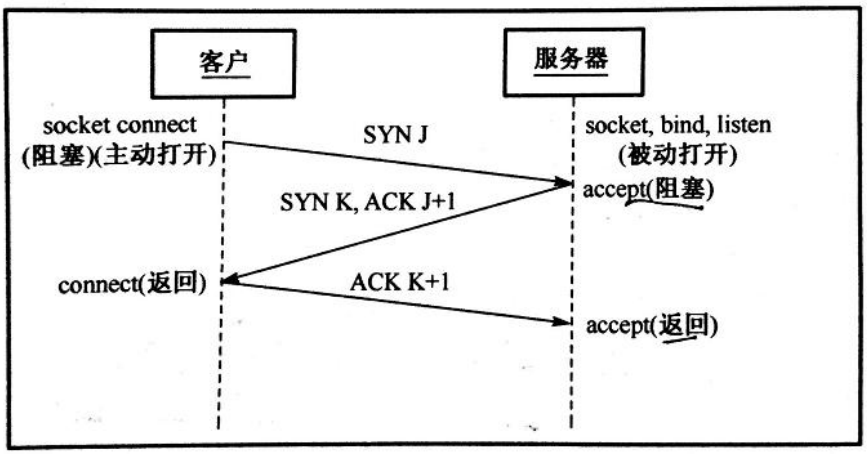
> 发送窗：用来限制发送方可以发送的数据大小，其中发送窗口的大小由接收端返回的 TCP 报文段中窗口字段来控制，接收方通过此字段告知发送方自己的缓冲（受系统、硬件等限制）大小。
>
> 接收窗：用来标记可以接收的数据大小。

TCP 是流数据，发送出去的数据流可以被分为以下四部分：已发送且被确认部分 | 已发送未被确认部分 | 未发送但可发送部分 | 不可发送部分，其中发送窗 = 已发送未确认部分 + 未发但可发送部分。接收到的数据流可分为：已接收 | 未接收但准备接收 | 未接收不准备接收。接收窗 = 未接收但准备接收部分。

发送窗内数据只有当接收到接收端某段发送数据的 ACK 响应时才移动发送窗，左边缘紧贴刚被确认的数据。接收窗也只有接收到数据且最左侧连续时才移动接收窗口。

83、建立 TCP 服务器的各个系统调用过程是怎样的？





服务器：

创建 socket -> int socket(int domain, int type, int protocol);

domain：协议域，决定了 socket 的地址类型，IPv4 为 AF_INET。

type：指定 socket 类型，SOCK_STREAM 为 TCP 连接。

protocol：指定协议。IPPROTO_TCP 表示 TCP 协议，为 0 时自动选择 type 默认协议。

绑定 socket 和端口号 -> int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

sockfd：socket 返回的套接字描述符，类似于文件描述符 fd。

addr：有个 sockaddr 类型数据的指针，指向的是被绑定结构变量。

```
// IPv4 的 sockaddr 地址结构
struct sockaddr_in {
    sa_family_t sin_family;    // 协议类型，AF_INET
    in_port_t sin_port;    // 端口号
    struct in_addr sin_addr;    // IP 地址
};
struct in_addr {
    uint32_t s_addr;
}
```

- addrlen：地址长度。

监听端口号 -> int listen(int sockfd, int backlog);

sockfd：要监听的 sock 描述字。

backlog：socket 可以排队的最大连接数。

接收用户请求 -> int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

sockfd：服务器 socket 描述字。

addr：指向地址结构指针。

addrlen：协议地址长度。

注：一旦 accept 某个客户机请求成功将返回一个全新的描述符用于标识具体客户的 TCP 连接。

从 socket 中读取字符 -> ssize_t read(int fd, void *buf, size_t count);

fd：连接描述字。

buf：缓冲区 buf。

count：缓冲区长度。

注：大于 0 表示读取的字节数，返回 0 表示文件读取结束，小于 0 表示发生错误。

关闭 socket -> int close(int fd);

fd：accept 返回的连接描述字，每个连接有一个，生命周期为连接周期。

注：sockfd 是监听描述字，一个服务器只有一个，用于监听是否有连接；fd 是连接描述字，用于每个连接的操作。

客户机：

创建 socket -> int socket(int domain, int type, int protocol);

连接指定计算机 -> int connect(int sockfd, struct sockaddr* addr, socklen_t addrlen);

sockfd 客户端的 sock 描述字。

addr：服务器的地址。

addrlen：socket 地址长度。

向 socket 写入信息 -> ssize_t write(int fd, const void *buf, size_t count);

fd、buf、count：同 read 中意义。

大于 0 表示写了部分或全部数据，小于 0 表示出错。

关闭 oscket -> int close(int fd);

- fd：同服务器端 fd。

84、TCP 协议如何保证可靠传输？

第一种回答

- **确认和重传**：接收方收到报文就会确认，发送方发送一段时间后没有收到确认就会重传。
- **数据校验**：TCP 报文头有校验和，用于校验报文是否损坏。
- **数据合理分片和排序**：tcp 会按最大传输单元(MTU)合理分片，接收方会缓存未按序到达的数据，重新排序后交给应用层。而 UDP：IP 数据报大于 1500 字节，大于 MTU。这个时候发送方的 IP 层就需要分片，把数据报分成若干片，是的每一片都小于 MTU。而接收方 IP 层则需要进行数据报的重组。由于 UDP 的特性，某一片数据丢失时，接收方便无法重组数据报，导致丢弃整个 UDP 数据报。
- **流量控制**：当接收方来不及处理发送方的数据，能通过滑动窗口，提示发送方降低发送的速率，防止包丢失。
- **拥塞控制**：当网络拥塞时，通过拥塞窗口，减少数据的发送，防止包丢失。

第二种回答

建立连接（标志位）：通信前确认通信实体存在。

序号机制（序号、确认号）：确保了数据是按序、完整到达。

数据校验（校验和）：CRC 校验全部数据。

超时重传（定时器）：保证因链路故障未能到达数据能够被多次重发。

窗口机制（窗口）：提供流量控制，避免过量发送。

拥塞控制：同上。

第三种回答

首部校验 这个校验机制能够确保数据传输不会出错吗？ 答案是不能。

原因

TCP 协议中规定，TCP 的首部字段中有一个字段是校验和，发送方将伪首部、TCP 首部、TCP 数据使用累加和校验的方式计算出一个数字，然后存放在首部的校验和字段里，接收者收到 TCP 包后重复这个过程，然后将计算出的校验和和接收到的首部中的校验和比较，如果不一致则说明数据在传输过程中出错。

这就是 TCP 的数据校验机制。 但是这个机制能够保证检查出一切错误吗？**显然不能**。

因为这种校验方式是累加和，也就是将一系列的数字（TCP 协议规定的是数据中的每 16 个比特位数据作为一个数字）求和后取末位。

但是小学生都知道 A+B=B+A，假如在传输的过程中有前后两个 16 比特位的数据前后颠倒了（至于为什么这么巧合？我不知道，也许路由器有 bug？也许是宇宙中的高能粒子击中了电缆？反正这个事情的概率不为零，就有可能会发生），那么校验和的计算结果和颠倒之前是一样的，那么接收端肯定无法检查出这是错误的数据。

解决方案

传输之前先使用 MD5 加密数据获得摘要，跟数据一起发送到服务端，服务端接收之后对数据也进行 MD5 加密，如果加密结果和摘要一致，则认为没有问题

85、UDP 是什么？

提供**无连接**的，尽最大努力的数据传输服务（**不保证数据传输的可靠性**）。

86、TCP 和 UDP 的区别

1、TCP 面向连接（如打电话要先拨号建立连接）;UDP 是无连接的，即发送数据之前不需要建立连接

2、TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP 尽最大努力交付，即不保证可靠交付

3、TCP 面向字节流，实际上是 TCP 把数据看成一连串无结构的字节流;UDP 是面向报文的

UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 IP 电话，实时视频会议等）

4、每一条 TCP 连接只能是点到点的;UDP 支持一对一，一对多，多对一和多对多的交互通信

5、TCP 首部开销 20 字节;UDP 的首部开销小，只有 8 个字节

6、TCP 的逻辑通信信道是全双工的可靠信道，UDP 则是不可靠信道

7、UDP 是面向报文的，发送方的 UDP 对应用层交下来的报文，不合并，不拆分，只是在其上面加上首部后就交给了下面的网络层，论应用层交给 UDP 多长的报文，它统统发送，一次发送一个。

而对接收方，接到后直接去除首部，交给上面的应用层就完成任务了。因此，它需要应用层控制报文的大小

TCP 是面向字节流的，它把上面应用层交下来的数据看成无结构的字节流会发送，可以想象成流水形式的，发送方 TCP 会将数据放入“蓄水池”（缓存区），等到可以发送的时候就发送，不能发送就等着 TCP 会根据当前网络的拥塞状态来确定每个报文段的大小。

补充题：封包和拆包你听说过吗？它是基于 TCP 还是 UDP 的？

封包和拆包都是基于 TCP 的概念。因为 TCP 是无边界的流传输，所以需要对 TCP 进行封包和拆包，确保发送和接收的数据不粘连。

- 封包：封包就是在发送数据报的时候为每个 TCP 数据包加上一个包头，将数据报分为包头和包体两个部分。包头是一个固定长度的结构体，里面包含该数据包的总长度。
- 拆包：接收方在接收到报文后提取包头中的长度信息进行截取。

87、UDP 的特点有哪些（附赠 TCP 的特点）？

- UDP 是**无连接的**；
- UDP 使用**尽最大努力交付**，即不保证可靠交付，因此主机不需要维持复杂的链接状态（这里面有许多参数）；
- UDP 是**面向报文**的；
- UDP **没有拥塞控制**，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 IP 电话，实时视频会议等）；
- UDP **支持一对一、一对多、多对一和多对多**的交互通信；
- UDP 的**首部开销小**，只有 8 个字节，比 TCP 的 20 个字节的首部要短。

那么，再说一次 TCP 的特点：

- **TCP 是面向连接的**。（就好像打电话一样，通话前需要先拨号建立连接，通话结束后要挂机释放连接）；
- 每一条 TCP 连接只能有两个端点，每一条 TCP 连接只能是点对点的（**一对一**）；
- TCP **提供可靠交付的服务**。通过 TCP 连接传送的数据，无差错、不丢失、不重复、并且按序到达；
- TCP **提供全双工通信**。TCP 允许通信双方的应用进程在任何时候都能发送数据。TCP 连接的两端都设有发送缓存和接收缓存，用来临时存放双方通信的数据；
- **面向字节流**。TCP 中的“流”（stream）指的是流入进程或从进程流出的字节序列。“面向字节流”的含义是：虽然应用程序和 TCP 的交互是一次一个数据块（大小不等），但 TCP 把应用程序交下来的数据仅仅看成是一连串的无结构的字节流。

88、TCP 对应的应用层协议

FTP：定义了文件传输协议，使用 21 端口. Telnet：它是一种用于远程登陆的端口,23 端口 SMTP：定义了简单邮件传送协议，服务器开放的是 25 号端口。 POP3：它是和 SMTP 对应，POP3 用于接收邮件。

89、UDP 对应的应用层协议

DNS：用于域名解析服务，用的是 53 号端口 SNMP：简单网络管理协议，使用 161 号端口 TFTP(Trival File Transfer Protocal)：简单文件传输协议，69

90、数据链路层常见协议？可以说一下吗？

协议	名称	作用
ARP	地址解析协议	根据 IP 地址获取物理地址
RARP	反向地址转换协议	根据物理地址获取 IP 地址
PPP	点对点协议	主要是用来通过拨号或专线方式建立点对点连接发送数据，使其成为各种主机、网桥和路由器之间简单连接的一种共通的解决方案

91、Ping 命令基于什么协议？原理是什么？

ping 是基于网络层的 ICMP 协议实现的。通过向对方发送一个 **ICMP 回送请求报文**，如果对方主机可达的话会收到该报文，并响应一个 **ICMP 回送回答报文**。

扩展：ICMP 报文的介绍。ICMP 报文分为两个种类：

1. ICMP 差错报告报文，常见的有
    1. 终点不可达
    2. 时间超过
    3. 参数问题
    4. 改变路由
2. ICMP 询问报文

    1. 回送请求和回答：向特定主机发出**回送请求报文**，收到回送请求报文的主机响应**回送回答报文**。
    2. 时间戳请求和回答：询问对方当前的时间，返回的是一个 32 位的时间戳。

92、在进行 UDP 编程的时候，一次发送多少 bytes 好?

当然,这个没有唯一答案，相对于不同的系统,不同的要求,其得到的答案是不一样的。

我这里仅对像 ICQ 一类的发送聊天消息的情况作分析，对于其他情况，你或许也能得到一点帮助:首先,我们知道,TCP/IP 通常被认为是一个四层协议系统,包括链路层,网络层,运输层,应用层.UDP 属于运输层,

下面我们由下至上一步一步来看:以太网(Ethernet)数据帧的长度必须在 46-1500 字节之间,这是由以太网的物理特性决定的。

这个 1500 字节被称为链路层的 MTU(最大传输单元).但这并不是指链路层的长度被限制在 1500 字节,其实这这个 MTU 指的是链路层的数据区.并不包括链路层的首部和尾部的 18 个字节.

所以,事实上,这个 1500 字节就是网络层 IP 数据报的长度限制。

因为 IP 数据报的首部为 20 字节,所以 IP 数据报的数据区长度最大为 1480 字节.而这个 1480 字节就是用来放 TCP 传来的 TCP 报文段或 UDP 传来的 UDP 数据报的。

又因为 UDP 数据报的首部 8 字节,所以 UDP 数据报的数据区最大长度为 1472 字节.这个 1472 字节就是我们可以使用的字节数。

当我们发送的 UDP 数据大于 1472 的时候会怎样呢？

这也就是说 IP 数据报大于 1500 字节,大于 MTU.这个时候发送方 IP 层就需要分片(fragmentation).

把数据报分成若干片,使每一片都小于 MTU.而接收方 IP 层则需要进行数据报的重组.

这样就会多做许多事情,而更严重的是,由于 UDP 的特性,当某一片数据传送中丢失时,接收方便，无法重组数据报.将导致丢弃整个 UDP 数据报。

因此,在普通的局域网环境下，我建议将 UDP 的数据控制在 1472 字节以下为好.

进行 Internet 编程时则不同,因为 Internet 上的路由器可能会将 MTU 设为不同的值.

如果我们假定 MTU 为 1500 来发送数据的,而途经的某个网络的 MTU 值小于 1500 字节,那么系统将会使用一系列的机制来调整 MTU 值,使数据报能够顺利到达目的地,这样就会做许多不必要的操作。

鉴于 Internet 上的标准 MTU 值为 576 字节，所以我建议在进行 Internet 的 UDP 编程时，最好将 UDP 的数据长度控件在 548 字节(576-8-20)以内。

93、TCP 利用滑动窗口实现流量控制的机制？

> 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。TCP 利用滑动窗口实现流量控制。

TCP 中采用滑动窗口来进行传输控制，滑动窗口的大小意味着**接收方还有多大的缓冲区可以用于接收数据**。

发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0 时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据。

> 例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个 1 字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

94、可以解释一下 RTO，RTT 和超时重传分别是什么吗？

> 超时重传：发送端发送报文后若长时间未收到确认的报文则需要重发该报文。可能有以下几种情况：
>
> > 发送的数据没能到达接收端，所以对方没有响应。
> >
> > 接收端接收到数据，但是 ACK 报文在返回过程中丢失。
> >
> > 接收端拒绝或丢弃数据。
>
> RTO：从上一次发送数据，因为长期没有收到 ACK 响应，到下一次重发之间的时间。就是重传间隔。
>
> > 通常每次重传 RTO 是前一次重传间隔的两倍，计量单位通常是 RTT。例：1RTT，2RTT，4RTT，8RTT……
> >
> > 重传次数到达上限之后停止重传。
>
> RTT：数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中一个往返用时。大小不稳定。

95、XSS 攻击是什么？（低频）

跨站点脚本攻击，指攻击者通过篡改网页，嵌入恶意脚本程序，在用户浏览网页时，控制用户浏览器进行恶意操作的一种攻击方式。如何防范 XSS 攻击 1）前端，服务端，同时需要字符串输入的长度限制。 2）前端，服务端，同时需要对 HTML 转义处理。将其中的”<”,”>” 等特殊字符进行转义编码。 防 XSS 的核心是必须对输入的数据做过滤处理。

96、CSRF 攻击？你知道吗？

跨站点请求伪造，指攻击者通过跨站请求，以合法的用户的身份进行非法操作。可以这么理解 CSRF 攻击：攻击者盗用你的身份，以你的名义向第三方网站发送恶意请求。CRSF 能做的事情包括利用你的身份发邮件，发短信，进行交易转账，甚至盗取账号信息。

97、如何防范 CSRF 攻击

1、**安全框架**，例如 Spring Security。

2、**token 机制**。

在 HTTP 请求中进行 token 验证，如果请求中没有 token 或者 token 内容不正确，则认为 CSRF 攻击而拒绝该请求。

3、**验证码**。

通常情况下，验证码能够很好的遏制 CSRF 攻击，但是很多情况下，出于用户体验考虑，验证码只能作为一种辅助手段，而不是最主要的解决方案。

4、**referer 识别**。

在 HTTP Header 中有一个字段 Referer，它记录了 HTTP 请求的来源地址。如果 Referer 是其他网站，就有可能是 CSRF 攻击，则拒绝该请求。

但是，服务器并非都能取到 Referer。很多用户出于隐私保护的考虑，限制了 Referer 的发送。在某些情况下，浏览器也不会发送 Referer，例如 HTTPS 跳转到 HTTP。 1）验证请求来源地址； 2）关键操作添加验证码； 3）在请求地址添加 token 并验证。

98、文件上传漏洞是如何发生的？你有经历过吗？

文件上传漏洞，指的是用户上传一个可执行的脚本文件，并通过此脚本文件获得了执行服务端命令的能力。

许多第三方框架、服务，都曾经被爆出文件上传漏洞，比如很早之前的 Struts2，以及富文本编辑器等等，可被攻击者上传恶意代码，有可能服务端就被人黑了。

99、如何防范文件上传漏洞

文件上传的目录设置为不可执行。

1）判断文件类型。在判断文件类型的时候，可以结合使用 MIME Type，后缀检查等方式。因为对于上传文件，不能简单地通过后缀名称来判断文件的类型，因为攻击者可以将可执行文件的后缀名称改为图片或其他后缀类型，诱导用户执行。

2）对上传的文件类型进行白名单校验，只允许上传可靠类型。

3）上传的文件需要进行重新命名，使攻击者无法猜想上传文件的访问路径，将极大地增加攻击成本，同时向 shell.php.rar.ara 这种文件，因为重命名而无法成功实施攻击。

4）限制上传文件的大小。

5）单独设置文件服务器的域名。

100、拥塞控制原理听说过吗？

拥塞控制目的是防止数据过多注入到网络中导致网络资源（路由器、交换机等）过载。

因为拥塞控制涉及网络链路全局，所以属于全局控制。控制拥塞使用拥塞窗口。

TCP 拥塞控制算法：

- 慢开始 & 拥塞避免：先试探网络拥塞程度再逐渐增大拥塞窗口。假设窗口长度为 d，收到一个确认就加 1，正好收到了 d 个确认，所以一共加 d，正好是翻倍，直到达到阀值 ssthresh，这部分是慢开始过程。达到阀值后每次以一个 MSS 为单位增长拥塞窗口大小，当发生拥塞（超时未收到确认），将阀值减为原先一半，继续执行增加，这个过程为拥塞避免。
- 快速重传 & 快速恢复：略。
- 最终拥塞窗口会收敛于稳定值。

101、如何区分流量控制和拥塞控制？

流量控制属于通信双方协商；拥塞控制涉及通信链路全局。

流量控制需要通信双方各维护一个发送窗、一个接收窗，对任意一方，接收窗大小由自身决定，发送窗大小由接收方响应的 TCP 报文段中窗口值确定；拥塞控制的拥塞窗口大小变化由试探性发送一定数据量数据探查网络状况后而自适应调整。

实际最终发送窗口 = min{流控发送窗口，拥塞窗口}。

102、常见的 HTTP 状态码有哪些？

状态码	类别	含义
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出

1xx 信息

100 Continue：表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应。

2xx 成功

- **200 OK**
- **204 No Content**：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
- **206 Partial Content**：表示客户端进行了范围请求，响应报文包含由 Content-Range 指定范围的实体内容。

3xx 重定向

- **301 Moved Permanently**：永久性重定向
- **302 Found**：临时性重定向
- **303 See Other**：和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源。
- **304 Not Modified**：如果请求报文首部包含一些条件，例如：If-Match，If-Modified-Since，If-None-Match，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码。
- **307 Temporary Redirect**：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法。

4xx 客户端错误

- **400 Bad Request**：请求报文中存在语法错误。

- **401 Unauthorized** ：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。
- **403 Forbidden** ：请求被拒绝。
- **404 Not Found**

5xx 服务器错误

- **500 Internal Server Error** ：服务器正在执行请求时发生错误。
- **503 Service Unavailable** ：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。

103、服务器出现大量 close_wait 的连接的原因是什么？有什么解决方法？

close_wait 状态是在 TCP 四次挥手的时候收到 FIN 但是没有发送自己的 FIN 时出现的，服务器出现大量 close_wait 状态的原因有两种：

- 服务器内部业务处理占用了过多时间，都没能处理完业务；或者还有数据需要发送；或者服务器的业务逻辑有问题，没有执行 close()方法
- 服务器的父进程派生出子进程，子进程继承了 socket，收到 FIN 的时候子进程处理但父进程没有处理该信号，导致 socket 的引用不为 0 无法回收

处理方法：

- 停止应用程序
- 修改程序里的 bug

104、一台机器能够使用的端口号上限是多少，是否可以修改？如果想要用的端口超过这个限制怎么办？

65536.因为 TCP 的报文头部中源端口号和目的端口号的长度是 16 位，也就是可以表示 2^16=65536 个不同端口号，因此 TCP 可供识别的端口号最多只有 65536 个。但是由于 0 到 1023 是知名服务端口，所以实际上还要少 1024 个端口号。

而对于服务器来说，可以开的端口号与 65536 无关，其实是受限于 Linux 可以打开的文件数量，并且可以通过 MaxUserPort 来进行配置。