


计算机网络

Contents

- Contents
- 计算机网络模型
 - OSI 的 7 层网络模型？
 - TCP/IP 的 4 层网络模型？
 - 5 层因特网协议栈？
 - 从键入网址到网页显示期间发生了什么
 - Linux 如何收网络包？
 - Linux 如何发网络包？
 - 发送网络数据涉及几次内存拷贝？
-  HTTP 和 HTTPs
 - HTTP 状态码的含义？
 - 100 类
 - 200 类
 - 300 类
 - 400 类
 - 500 类
 - HTTP 常见字段
 - HTTP 缓存有哪些实现方式？
 - HTTP 和 HTTPS 的区别？
 - HTTPS 的加密与认证过程？
 - ClientHello
 - ServerHello
 - 客户端回应
 - 服务端回应
 - HTTPS 一定安全可靠吗？
 - HTTP1.0、HTTP1.1、HTTP2 和 HTTP3 的区别？
 - HTTP1.0
 - HTTP1.1
 - HTTP2.0
 - HTTP3(QUIC)

- QUIC 协议的概念和特点?
 - 概念
 - 特点
- QUIC 如何保证可靠传输?
- HTTP 的 GET 和 POST 方法区别?
- GET 请求可以带 body 吗?
- 既然有 HTTP 协议，为什么还要有 RPC?
 - 定义
 - 服务发现
 - 底层连接方式
 - 传输内容
 - 历史原因
- 什么是 XSS 攻击？有什么解决办法？
 - 概念
 - 分类
 - 危害
 - 防范
- 什么是 CSRF 攻击？有什么解决办法？
 - 概念
 - 防范
- 中间人攻击以及如何防范？
 - 概念
 - 防范
- HTTP 1.1 如何优化？
 - 避免发送 HTTP 请求
 - 减少 HTTP 请求次数
 - 减少 HTTP 响应数据大小
- HTTPs 如何优化？
 - 硬件优化
 - 软件优化
 - 协议优化
 - 证书优化
 - 会话复用
-  TCP 和 UDP
 - TCP 的头部结构
 - TCP 如何保证可靠传输
 - TCP 的三次握手
 - TCP 为什么要三次握手？

- 如果 TCP 的三次握手丢失会发生什么？
- TCP 为什么不是两次握手？
- TCP 的四次挥手？
- TCP 为什么要四次挥手？
- 在 FIN_WAIT_2 状态下，是如何处理收到的乱序到 FIN 报文，然后 TCP 连接又是什么时候才进入到 TIME_WAIT 状态？
- 如果 TCP 的四次挥手丢失会发生什么？
- TCP 为什么不是三次挥手？
- TCP 的延迟应答和累计应答？
- TCP 的 MSL
- 已经建立了连接，客户端突然出现故障了会怎样？
- 什么时候用长连接，短连接？
- TCP 的半连接队列和全连接队列？
- 什么是 SYN 攻击？如何避免？
 - 概念
 - 避免方法
- TIME_WAIT 作用，过多如何解决？
 - 作用
 - 危害
 - 避免方法
- tcp_tw_reuse 为什么默认是关闭的？
- PAWS 的保护机制
 - 潜在问题：
- TIME_WAIT 状态为什么需要经过 2MSL
- CLOSE_WAIT 状态过多如何解决？
- TCP 和 UDP 的区别？
- 粘包和拆包问题的解决办法？
 - 概念
 - 解决办法
- TCP 的 keepalive 和 HTTP 的 keepalive 的区别？
- IP 层会分片，为什么 TCP 层还需要 MSS 呢？
- IP
 - DNS 查询服务器的基本流程？
 - DNS 采用 TCP 还是 UDP，为什么？
 - DNS 劫持是什么？解决办法？
 - 概念
 - 解决方法
 - 浏览器输入一个 URL 到显示器显示的过程？

- PING 是怎么工作的？
- Cookie 和 Session 的关系和区别是什么？
 - Cookie 概念
 - Cookie 作用
 - Session 概念
 - 差别
- IPv4 和 IPv6 的区别？
- 什么是跨域，什么情况下会发生跨域请求？
 - 概念
 - 解决方法

计算机网络模型

OSI 的 7 层网络模型？

- **应用层**：确定**进程之间通信的性质**以及满足用户需要以及提供网络和用户应用，为应用程序提供服务
- **表示层**：主要解决用户信息的语法表示问题
- **会话层**：会话层就是**负责建立、管理和终止表示层实体之间的通信会话**
- **传输层**：实现网络不同主机上的用户进程之间的数据通信
- **网络层**：本层通过 IP 寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点
- **数据链路层**：将上层数据封装成帧，用 MAC 地址访问媒介，并由错误检测和修正
- **物理层**：设备之间比特流的传输

TCP/IP 的 4 层网络模型？

- **应用层**：确定**进程之间通信的性质**以及满足用户需要以及提供网络和用户应用，为应用程序提供服务
- **传输层**：实现网络不同主机上的用户进程之间的数据通信
- **网络层**：本层通过 IP 寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点
- **数据链路层**：将上层数据封装成帧，用 MAC 地址访问媒介，并由错误检测和修正

5 层因特网协议栈？

- **应用层**：确定**进程之间通信的性质**以及满足用户需要以及提供网络和用户应用，为应用程序提供服务
- **传输层**：实现网络不同主机上的用户进程之间的数据通信
- **网络层**：本层通过 IP 寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点
- **数据链路层**：将上层数据封装成帧，用 MAC 地址访问媒介，并由错误检测和修正
- **物理层**：设备之间比特流的传输

从键入网址到网页显示期间发生了什么

- **解析 URL 生成 http 请求**：获取 URL 代表的域名，把它填充到请求头中的 host 字段中并生成 http 请求
- **通过 DNS 协议解析出 IP 地址**：浏览器首先查询本地的缓存，如果没有的话查询 hosts 文件。如果仍然没有，再向本地的 DNS 服务器查询。如果本地 DNS 服务器没有结果，它将向递归 DNS 服务器发起查询，递归服务器会通过迭代查询向根域名服务器、顶级域名服务器和权威域名服务器请求最终的解析结果，最后缓存在 DNS 本地服务器
- **DNS 获取到 IP 地址以后把 HTTP 的传输工作交给协议栈来进行处理**：通过调用 socket 函数，由用户态进入内核态，接下来就是数据传输的过程
- **TCP**：三次握手建立连接；数据分片，数据长度不能超过 MSS；生成 TCP 报文
- **IP**：把数据包转发到对应的子网；通过路由器发送给下一跳；超过一定的跳数则丢弃
- **转发到对应的 MAC 地址**：通过 IP 地址找到对应的 MAC 地址，具体是查看 ARP 缓存，然后如果缓存中没有，那就通过在子网广播来去询问 MAC 地址，找到后再保存到 ARP 缓存中。然后通过交换机转发给对应的 MAC 地址
- **网卡通过数字信号转化成电信号**：接下来就是服务端进行处理

Linux 如何收网络包？

- 网卡受到网络包后通过 DMA 技术将网络包写入到 **Ring Buffer** 环形缓冲区
- 现代 Linux 通常通过 **NAPI 机制** 对接受数据的服务程序进行中断唤醒，通过 poll 的方法轮询数据
- 即网卡向 CPU 发起硬件中断，当 CPU 收到硬件中断请求后根据中断表调用已经注册的中断处理函数
- 首先进行**关中断**，硬件中断函数会首先通知网卡在下次有数据时直接写入内存并恢复中断屏蔽，进入**软中断**
- 内核中的 **ksoftirqd** 线程专门负责软中断的处理，当 ksoftirqd 内核线程收到软中断后就会轮询处理数据

- **ksoftirqd 线程**会从 Ring Buffer 中获取一个数据帧，用 `sk_buff` 表示，从而可以作为一个网络包交给网络协议栈进行逐层处理
- 首先进入到**网络接口层**，在这一层会检查报文的合法性，如果不合法则丢弃，合法则会找出该网络包的上层协议的类型，然后交给网络层
- **网络层**取出 IP 包，判断网络包下一步的走向，比如是交给上层处理还是转发出去。当确认这个网络包要发送给本机后会从 IP 头部查看上一层协议的类型是 TCP 还是 UDP，接着去掉 IP 头，交给传输层
- **传输层**取出 TCP 头或 UDP 头，根据四元组「源 IP、源端口、目的 IP、目的端口」作为标识，找出对应的 Socket，并把数据放到 Socket 的接收缓冲区
- **应用层**程序调用 Socket 接口，将内核的 Socket 接收缓冲区的数据「拷贝」到应用层的缓冲区，然后唤醒用户进程

Linux 如何发网络包？

- 首先应用程序调用 **Socket 发送数据包**的接口，由于这个是系统调用，所以会从用户态陷入到内核态中的 Socket 层，内核会申请一个内核态的 **sk_buff 内存**，将用户待发送的数据拷贝到 `sk_buff` 内存，并将其加入到发送缓冲区
- 接下来网络协议栈从 Socket 发送缓冲区中**取出 sk_buff**,并按照 TCP/IP 协议栈从上到下逐层处理
- 如果使用的是 TCP 传输协议发送数据，那么先**拷贝一个新的 sk_buff 副本**
- **接着对 sk_buff 填充 TCP 头**。`sk_buff` 可以表示各个层的数据包，在应用层数据包叫 data，在 TCP 层我们称为 segment,在 IP 层我们叫 packet，在数据链路层称为 frame
- **网络层**：选取路由(确认下一跳的 IP)、填充 IP 头、netfilter 过滤、对超过 MTU 大小的数据包进行分片。处理完这些工作后会交给网络接口层处理
- **网络接口层**：通过 ARP 协议获得下一跳的 MAC 地址，然后对 `sk_buff` 填充帧头和帧尾，接着将 `sk_buff` 放到网卡的发送队列中
- 当收到这个 TCP 报文的 ACK 应答时**传输层就会释放原始的 sk_buff**

发送网络数据涉及几次内存拷贝？

三次：

1. **调用发送数据的系统调用时**内核会申请一个内核态的 `sk_buff` 内存，将用户待发送的数据拷贝到 `sk_buff` 内存，并将其加入到发送缓冲区
2. **在使用 TCP 传输协议的情况下**，从传输层进入网络层时每一个 `sk_buff` 都会被克隆一个新的副本出来。副本 `sk_buff` 会被送往网络层，等它发送完的时候就会释放掉，然后原始的 `sk_buff` 还保留在传输层，目的是为了**实现 TCP 的可靠传输**，等收到这个数据包的 ACK 时，才会释放原始的 `sk_buff`

3. 当 IP 层发现 `sk_buff` 大于 MTU 时才需要进行。会再申请额外的 `sk_buff` 并将原来的 `sk_buff` 拷贝为多个小的 `sk_buff`

HTTP 和 HTTPS

HTTP 状态码的含义?

100 类

100 类状态码属于提示信息，是协议处理中的一种中间状态，实际用到的比较少

- **100(继续)**: 请求者应当继续提出请求。 服务器返回此代码表示已收到请求的第一部分，正在等待其余部分
- **101(切换协议)**: 请求者已要求服务器切换协议，服务器已确认并准备切换

200 类

200 类状态码表示服务器成功处理了客户端的请求。

- **200(成功)**: 表示服务器响应成功，也就是服务器找到了客户端请求的内容，并且将内容返回给客户端
- **204(已创建)**: 请求成功并且服务器创建了新的资源
- **206(部分内容)**: 服务器成功处理了部分 GET 请求

300 类

300 类状态码表示客户端请求的资源发生了变动，需要客户端用新的 URL 重新发送请求获取资源，也就是重定向。

- **301(永久移动)**: 代表**永久性的重定向**，值得注意的是，这种重定向跳转，从严格意义来讲不是服务器跳转，而是**客户端跳转**的。这个“跳”的动作是服务器是通过回传状态码 301 来下达给客户端的，让客户端完成跳转
- **302(临时移动)**: 代表临时跳转。例如：URL 地址 A 可以向 URL 地址 B 上跳转，但这并不是永久性的，在经过一段时间后，URL 地址 A 还可能向 URL 地址 C 上跳转
- **304(未修改)**: 服务器通过返回状态码304可以告诉客户端请求资源成功，但是这个资源不是由服务器提供返回给客户端的，而是客户端本地浏览器缓存中就有的这个资源，因为可以从缓存中获取这个资源，从而节省传输的开销

400 类

400 类状态码表示客户端发送的报文有误，服务器无法处理。

- **400(错误请求)**：服务器不理解请求的语法
- **403(禁止)**：代表请求的服务器资源权限不够，也就是没有权限去访问服务器的资源，或者请求的 IP 地址被封掉了
- **404(未找到)**：代表服务器上没有该资源，或者说服务器找不到客户端请求的资源，是最常见的请求错误码

500 类

500 类状态码表示客户端请求报文正确，但是服务器处理时内部发生了错误，属于服务器端的错误码。

- **500(服务器内部错误)**：代表程序错误，也就是说请求的网页程序本身报错了。在服务器端的网页程序出错。由于现在的浏览器都会对状态码 500 做一定的处理，所以在一般情况下会返回一个定制的错误页面。
- **501(尚未实施)**：服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码。
- **502(网关错误)**：通常是服务器作为网关或代理时返回的错误码，表示服务器自身工作正常，访问后端服务器发生了错误。
- **503(服务不可用)**：表示服务器当前很忙，暂时无法响应客户端。
- **504(网关超时)**：服务器作为网关或代理，但是没有及时从上游服务器收到请求。
- **505(HTTP 版本不受支持)**：服务器不支持请求中所用的 HTTP 协议版本

HTTP 常见字段

- **Host**：指定服务器域名
- **Content-Length**：为了解决 TCP 粘包问题所设计的 HTTP 数据长度
- **Connection**：是否使用长连接
- **Content-Type**：指定数据格式，例如文本或图像、编码方式
- **Content-Encoding**：数据的压缩方法，如 gzip 等等

HTTP 缓存有哪些实现方式？

- **强制缓存**：强制缓存指的是只要浏览器判断缓存没有过期，则直接使用浏览器的本地缓存，决定是否使用缓存的主动性在于浏览器这边。相关字段有 **Cache-Control** 与 **Expires**，前者优先级更高

- **协商缓存**：请求的响应码 304，告诉浏览器可以使用本地缓存的资源，通过服务端告知客户端是否可以使用缓存的方式被称为协商缓存。请求头部用 **If-Modified-Since**；响应头部使用 **Last-Modified** 来维护。当发现资源过期时请求方会带上 **Last-Modified** 字段，如果修改过则返回 200 OK 以及相关资源，否则继续 304 走缓存
- 协商缓存还可以通过 **If-None-Match** 与 **ETag** 两种字段实现。前者属于请求头部，后者属于响应头部。这种方法基于唯一标识，相对来说可以更准确判断文件是否被修改，**ETag 的优先级也比 Last-Modified 更高**
- **协商缓存需要配合强制缓存中的 Cache-Control 使用**。当缓存没有对应资源时请求服务器

HTTP 和 HTTPS 的区别？

- **工作端口不同**：HTTP 80；HTTPS 443
- HTTP 明文传输，数据都是未加密的，安全性较差，HTTPS(SSL+HTTP)数据传输过程是加密的，安全性较好
- 使用 HTTPS 协议需要到 CA 申请证书
- **HTTP 页面响应速度比 HTTPS 快**，主要是因为 HTTP 使用 TCP 三次握手建立连接，而 HTTPS 除了 TCP 的三个包，还要加上 SSL 握手的消耗
- HTTPS 其实就是建构在 SSL/TLS 之上的 HTTP 协议，所以，要比较 HTTPS 比 HTTP 要更耗费服务器资源

HTTPS 的加密与认证过程？

ClientHello

首先，由客户端向服务端发起加密通信请求。客户端主要向服务端发送：

- **客户端支持的 SSL/TLS 协议版本**
- 客户端产生的**随机数**(Client Random)
- 客户端支持的**密码套件列表**

ServerHello

服务器收到客户端请求后，向客户端发出响应。服务端回应的内容有：

- **确认 SSL/ TLS 协议版本**(如果浏览器不支持，则关闭加密通信)
- 服务端生产的**随机数**(Server Random)
- 确认的**密码套件列表**
- 服务端的**数字证书**

客户端回应

客户端收到服务端的回应之后，首先通过浏览器或者操作系统中的 CA 公钥，确认服务端的数字证书的真实性。如果证书没有问题，客户端会从数字证书中取出服务端的公钥，然后使用它加密报文，向服务端发送如下信息：

- **一个随机数**，该随机数会被服务端公钥加密
- **加密通信算法改变通知**，表示随后的信息都将用会话密钥加密通信
- **客户端握手结束通知**，表示客户端的握手阶段已经结束
- 之前所有内容的发生的数据做个**摘要**，用来供服务端校验

服务端和客户端有了三个随机数，接着用双方协商的加密算法，各自生成本次通信的会话密钥

服务端回应

服务端收到客户端的第三个随机数(pre-master key)之后，通过协商的加密算法，计算出本次通信的会话密钥。服务端向客户端发送最后的信息：

- **加密通信算法改变通知**，表示随后的信息都将用「会话密钥」加密通信
- **服务端握手结束通知**，表示服务端的握手阶段已经结束
- 之前所有内容的发生的数据做个**摘要**，用来供客户端校验

HTTPS 一定安全可靠吗？

不安全是因为用户点击接受了中间人服务器的证书。中间人服务器与客户端在 TLS 握手过程中，实际上发送了自己伪造的证书给浏览器，而这个伪造的证书是能被浏览器(客户端)识别出是非法的，于是就会提醒用户该证书存在问题。如果用户点击「继续浏览此网站」，相当于用户接受了中间人伪造的证书，那么后续整个 HTTPS 通信都能被中间人监听了。

HTTPS 协议本身到目前为止还是没有任何漏洞的，即使你成功进行中间人攻击，本质上是利用了客户端的漏洞(用户点击继续访问或者被恶意导入伪造的根证书)，并不是 HTTPS 不够安全。

HTTP1.0、HTTP1.1、HTTP2 和 HTTP3 的区别？

HTTP1.0

- **无连接**：每次请求都要建立连接，需要使用 **keep-alive** 参数建立长连接，建立连接十分消耗资源
- **队头阻塞**：HTTP1.0 规定下一个请求必须在前一个请求响应到达之前才能发送，假设前一个请求响应一直不到达，那么下一个请求就不发送，后面的请求就阻塞了

- **缓存**：在 HTTP 1.0 中主要使用 **header** 里的协商缓存 **last-modified** 和 **if-modified-since**，强制缓存 **Expires** 来做为缓存判断的标准

HTTP1.1

- **长连接**：好处在于减少了 TCP 连接的重复建立和断开所造成的额外开销，减轻了服务器端的负载
- **支持管道(pipeline)网络传输**：只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间
- **缓存处理**：HTTP1.1 引入了更多的缓存控制策略，许多可供选择的缓存头来控制缓存策略
- **断点续传**：HTTP1.1 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

HTTP2.0

- **header 压缩**：HTTP1 的 header 带有大量信息，而且每次都要重复发送，HTTP2 使用 encoder 来减少需要传输的 header 大小，通讯双方各自 cache 一份 header fields 表，既避免了重复 header 的传输，又减小了需要传输的大小
- **多路复用**：使用多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比 HTTP1.1 大了好几个数量级。一个 TCP 连接中有多个 Stream(Stream 之间有依赖，前面的 Stream 阻塞后后面的 Stream)；一个 Stream 中包含多个 Message，对应 HTTP1 中的请求或响应；Message 包含多个 Frame，以二进制压缩格式存放 HTTP1 中的内容，对应一个请求的头部或者数据
- **二进制帧格式**：HTTP2 把请求在应用层切分成二进制帧并标上序号，服务器接收到二进制帧后组装成请求进行处理，从而达到并发发送请求的效果，对于服务器端，响应可以通过序号确定是哪个请求，从而不会出现混乱的问题
- **服务器推送**：HTTP2 引入了 server push，它允许服务端推送资源给浏览器，在浏览器明确地请求之前，免得客户端再次创建连接发送请求到服务器端获取。这样客户端可以直接从本地加载这些资源，不用再通过网络

HTTP3(QUIC)

- HTTP3 直接放弃使用 TCP，将传输层协议改成 UDP，**使用 UDP 实现可靠传输**
- **0-RTT**：缓存当前会话的上下文，下次恢复会话的时候，只需要将之前的缓存传递给服务器，验证通过，就可以进行传输了 (**这是 QUIC 协议相比 HTTP2.0 的最大优势**)
- **多路复用**：QUIC 基于 UDP，一个连接上的多个 stream 之间没有依赖，即使丢包，只需要重发丢失的包即可，不需要重传整个连接
- **更好的移动端表现**：QUIC 在移动端的表现比 TCP 好，因为 TCP 是基于 IP 识别连接，而 QUIC 是通过 ID 识别链接。无论网络环境如何变化，只要 ID 不变，就能迅速重新连上

- **加密认证的报文**：TCP 协议头没有经过任何加密和认证，在传输过程中很容易被中间网络设备篡改、注入和窃听。QUIC 的 packet 除了个别报文，所有报文头部都是经过认证的，报文 Body 都是经过加密的。只要对 QUIC 做任何更改，接收端都能及时发现，有效地降低了安全风险
- **向前纠错机制**：向前纠错(Foward Error Connec, FEC)，每个数据包除了它本身的内容之外还包括了其他数据包的数据，因此少量的丢包可以通过其他包的冗余数据直接组装而无需重传。向前纠错牺牲了每个数据包可以发送数据的上限，但是带来的提升大于丢包导致的数据重传，因为数据重传将会消耗更多的时间(包括确认数据包丢失，请求重传，等待新数据包等步骤的时间消耗)

QUIC 协议的概念和特点？

概念

HTTP3 基于 UDP 协议在应用层实现了 QUIC 协议，具有类似 TCP 的**连接管理、拥塞窗口、流量控制**的网络特性，让 UDP 协议变得可靠

特点

- **无队头阻塞**：QUIC 协议有类似 HTTP2 Stream 与多路复用的概念，可以在**同一条连接上并发传输多个 Stream**。UDP 不关心数据包的顺序，也不关心是否丢包。UDP 将每个数据包都有一个序号唯一标识。当某个流中的一个数据包丢失了，该流的其他数据包到达了，数据也无法被 HTTP3 读取，QUIC 重传丢失的报文之后数据才会交给 HTTP3。而其他流的数据报文只要被完整接收，HTTP3 就可以读取到数据。QUIC 连接上的多个 Stream 之间并没有依赖，都是独立的，某个流发生丢包了，只会影响该流，其他流不受影响
- **快速连接建立**：HTTP3 在传输数据前虽然需要 QUIC 协议握手，这个**握手过程只需要 1 RTT**，握手的目的是为确认双方的「连接 ID」，**连接迁移就是基于连接 ID 实现的**。HTTP3 的 QUIC 协议不与 TLS 分层，QUIC 内部包含了 TLS，它在自己的帧会携带 TLS 记录，只需 1 个 RTT 就可以完成建立连接与密钥协商。在第二次连接的时候，应用数据包可以和 QUIC 握手信息(连接信息 + TLS 信息)一起发送
- **连接迁移**：QUIC 协议没有用四元组的方式来绑定连接，而是通过 **连接 ID** 来标记通信的两个端点，客户端和服务端可以各自选择一组 ID 来标记自己，因此即使移动设备的网络变化后，导致 IP 地址变化了，只要仍保有上下文信息(比如连接 ID、TLS 密钥等)，就可以复用原连接，消除重连的成本，达到了**连接迁移**的功能

QUIC 如何保证可靠传输？

- **Packet Header**：分为 Long Packet Header 用于首次建立连接和 Short Packet Header 用于日常传输数据。QUIC 也是需要三次握手来建立连接的，目的是为了协商连接 ID。协商出连接 ID 后，

后续传输时，双方只需要固定住连接 ID，从而实现连接迁移功能。Short Packet Header 中的 Packet Number 每个报文有独一无二的编号，并且严格递增。单调递增的设计，可以让数据包不再像 TCP 那样必须有序确认，当数据包 Packet N 丢失后，只要有新的已接收数据包确认，当前窗口就会继续向右滑动，从而解决了队头阻塞的问题

- **QUIC Frame Header**：一个 Packet 报文中可以存放多个 QUIC Frame。用于传输的 Stream Frame 有 Stream ID、Offset 和 length 字段，Stream ID 用于多个并发传输的 HTTP 消息，通过不同的 Stream ID 加以区别、Offset 字段类似于 TCP 协议中的 Seq 序号，保证数据的顺序性和可靠性；Length 标识了 Frame 数据的长度。如果发生丢包了进行重传，通过比较两个数据包的 Stream ID 与 Stream Offset，如果都是一致，就说明这两个数据包的内容一致

所以，QUIC 通过**单向递增的 Packet Number**，配合 **Stream ID 与 Offset 字段信息**，可以支持乱序确认而不影响数据包的正确组装，摆脱了 TCP 必须按顺序确认的限制。

HTTP 的 GET 和 POST 方法区别？

- GET 一般用来从服务器上获取资源，POST 一般用来更新服务器上的资源
- **GET 是幂等的**，即读取同一个资源总是得到相同的数据，而 **POST 不是幂等的**，因为每次请求对资源的改变并不是相同的
- GET 不会改变服务器上的资源，而 POST 会对服务器资源进行改变
- GET 请求的数据会附在 URL 之后，即将请求数据放置在 HTTP 报文的请求头中，以 **？分割 URL 和传输数据**，参数之间以 **& 相连**；而 POST 请求会把提交的数据则放置在是 HTTP 请求报文的请求体中
- **POST 的安全性要比 GET 的安全性高**，因为 GET 请求提交的数据将明文出现在 URL 上，而且 POST 请求参数则被包装到请求体中，相对更安全(实际上这个论断不准确：无论是 GET 还是 POST 底层都是 TCP/IP，GET 和 POST 能做的事情是一样的。你要给 GET 加上 request body，给 POST 带上 url 参数，技术上是完全行的通的。所以对于两者的安全性论断**没有统一答案**，两者**本质相同**)
- 从请求的大小看，GET 请求的长度受限于浏览器或服务器对 URL 长度的限制，允许发送的数据量比较小，而 POST 请求则是没有大小限制的

GET 请求可以带 body 吗？

RFC 规范并没有规定 GET 请求不能带 body。任何请求都可以带 body。GET 请求是获取资源，所以根据这个语义不需要用到 body。URL 中的查询参数也不是 GET 所独有的，POST 请求的 URL 中也可以有参数的。

既然有 HTTP 协议，为什么还要有 RPC？

定义

TCP 是传输层的协议，基于 TCP 造出来的 HTTP 和各类 RPC 协议都只是**定义了不同消息格式的应用层协议**而已。HTTP 协议是**超文本传输协议**；RPC 是**远程过程调用**，它不是一个具体的协议，而是一种调用方式。虽然大部分 RPC 协议底层使用 TCP，但实际上它们不一定非要用 TCP，也可以改用 UDP 或者 HTTP。HTTP 主要用于 b/s 架构，而 RPC 更多用于 c/s 架构

服务发现

HTTP 中知道服务的域名，就可以通过 DNS 服务去解析得到 IP 地址，默认 80 端口。RPC 一般会有专门的中间服务去保存服务名和 IP 信息，比如 consul 或者 etcd，或者是 redis。想要访问某个服务，就去这些中间服务去获得 IP 和端口信息。由于 dns 也是服务发现的一种，所以也有基于 DNS 去做服务发现的组件，比如 CoreDNS

底层连接方式

HTTP 协议默认在建立底层 TCP 连接之后会一直保持这个连接(keep alive)，之后的请求和响应都会复用这条连接。RPC 协议也是通过建立 TCP 长连接进行数据交互，但 **RPC 协议一般还会再建个连接池**，在请求量大的时候，建立多条连接放在池内，要发数据的时候就从池里取一条连接出来，用完放回去便于下次再复用

传输内容

HTTP 设计初是用于做网页文本展示的，传的内容以字符串为主，有 header 和 body。在 body 这块，它使用 json 来序列化结构体数据，**内容会有冗余**。RPC 因为定制化程度更高，可以采用**体积更小**的 protobuf 或其他序列化协议去保存结构体数据，同时也不需要像 HTTP 那样考虑各种浏览器行为，比如 302 重定向。因此性能也会更好一些，这也是在公司内部微服务中抛弃 HTTP，选择使用 RPC 的最主要原因

历史原因

RPC 其实比 HTTP 出现的要早，且比目前主流的 HTTP1.1 性能要更好，所以大部分公司内部都还在使用 RPC。HTTP2 在 HTTP1.1 的基础上做了优化，性能可能比很多 RPC 协议都要好，但由于是这几年才出来的，所以也不太可能取代掉 RPC

什么是 XSS 攻击？有什么解决办法？

概念

XSS(Cross(X) Site Scripting, 跨站脚本攻击) 是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些脚本代码嵌入到 web 页面中去，使别的用户访问都会执行相应的嵌入代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式

分类

- **反射性 XSS 攻击(非持久性 XSS 攻击)**：之所以称为反射型 XSS，是因为这种攻击方式的注入代码是从目标服务器通过错误信息、搜索结果等等方式“反射”回来的：发出请求时，XSS 代码出现在 URL 中，作为输入提交到服务器端，服务器端解析后响应，XSS 代码随响应内容一起传回给浏览器，最后浏览器解析执行 XSS 代码。这个过程像一次反射，故叫反射型 XSS。而称为非持久型 XSS，则是因为这种攻击方式具有一次性，由于代码注入的是一个动态产生的页面而不是永久的页面，因此这种攻击方式只在点击链接的时候才产生作用
- **持久性 XSS 攻击(留言板场景)**：指 XSS 攻击代码存储在网站数据库，每当用户使用浏览器打开指定页面时，脚本就执行。与非持久性 XSS 攻击相比，持久性 XSS 攻击危害性更大

危害

- 盗取各类用户帐号，如机器登录帐号、用户网银帐号、各类管理员帐号
- 控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力
- 盗窃企业重要的具有商业价值的资料
- 非法转账
- 强制发送电子邮件
- 网站挂马
- 控制受害者机器向其它网站发起攻击

防范

漏洞产生的根本原因是**太相信用户提交的数据，对用户所提交的数据过滤不足**。解决方案也应该从这个方面入手：

- **将重要的 cookie 标记为 http only**：Javascript 中的 document.cookie 语句就不能获取到 cookie 了(如果在 cookie 中设置了 HttpOnly 属性，那么通过 js 脚本将无法读取到 cookie 信息，这样能有效防止 XSS 攻击)
- **表单数据规定值的类型**：例如：年龄应为只能为 int、name 只能为字母数字组合

- 对数据进行 HTML 编码处理
- 过滤或移除特殊的 HTML 标签：例如：

`<script>` , `<iframe>` , `< for <` , `> for>`

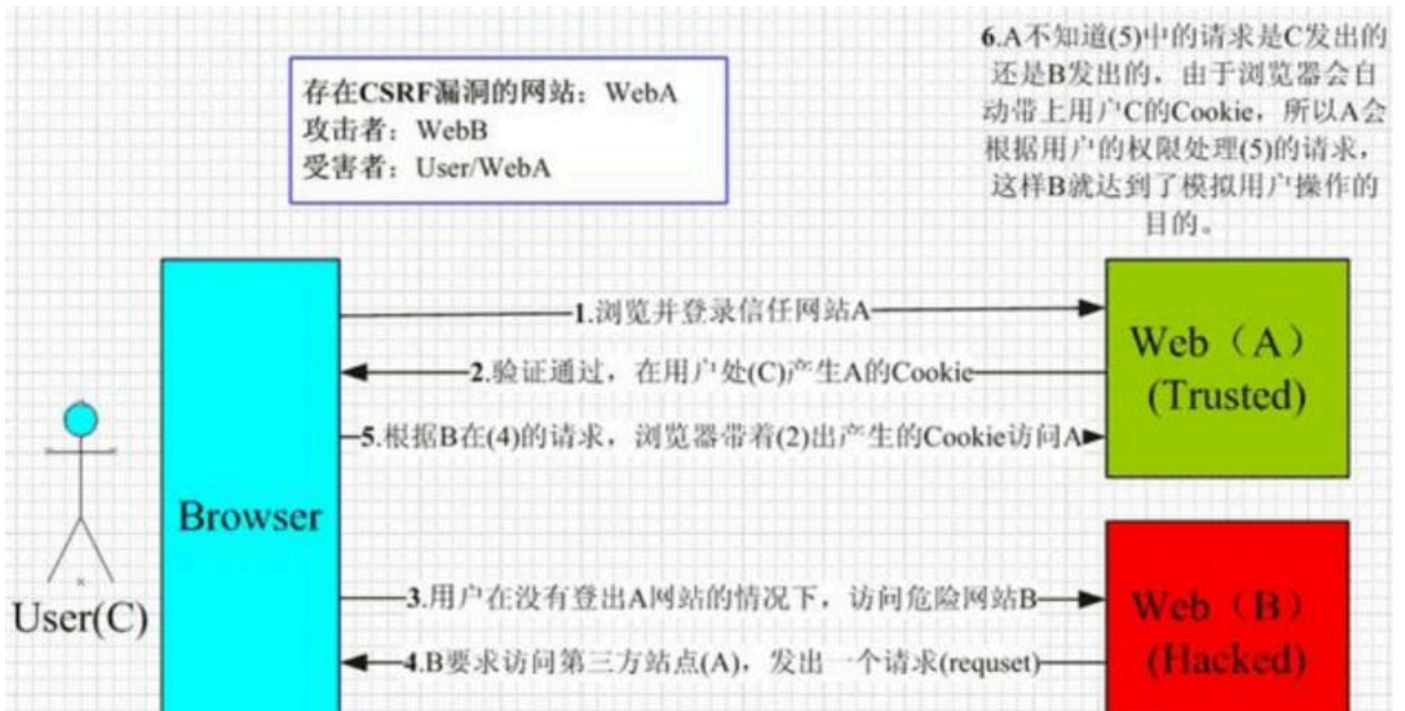
- 过滤 JavaScript 事件的标签：例如 "onclick=", "onfocus="

什么是 CSRF 攻击？有什么解决办法？

概念

CSRF 就是**跨站请求伪造**。是一种对网站的恶意利用。尽管听起来像跨站脚本(XSS)，但它与 XSS 非常不同，XSS 利用站点内的信任用户，而 CSRF 则通过伪装成受信任用户请求受信任的网站。

- 登录受信任网站 A，并在本地生成 Cookie(如果用户没有登录网站 A，那么网站 B 在诱导的时候，请求网站 A 的 api 接口时，会提示你登录)
- 在不登出 A 的情况下，访问危险网站 B(其实是利用了网站 A 的漏洞)



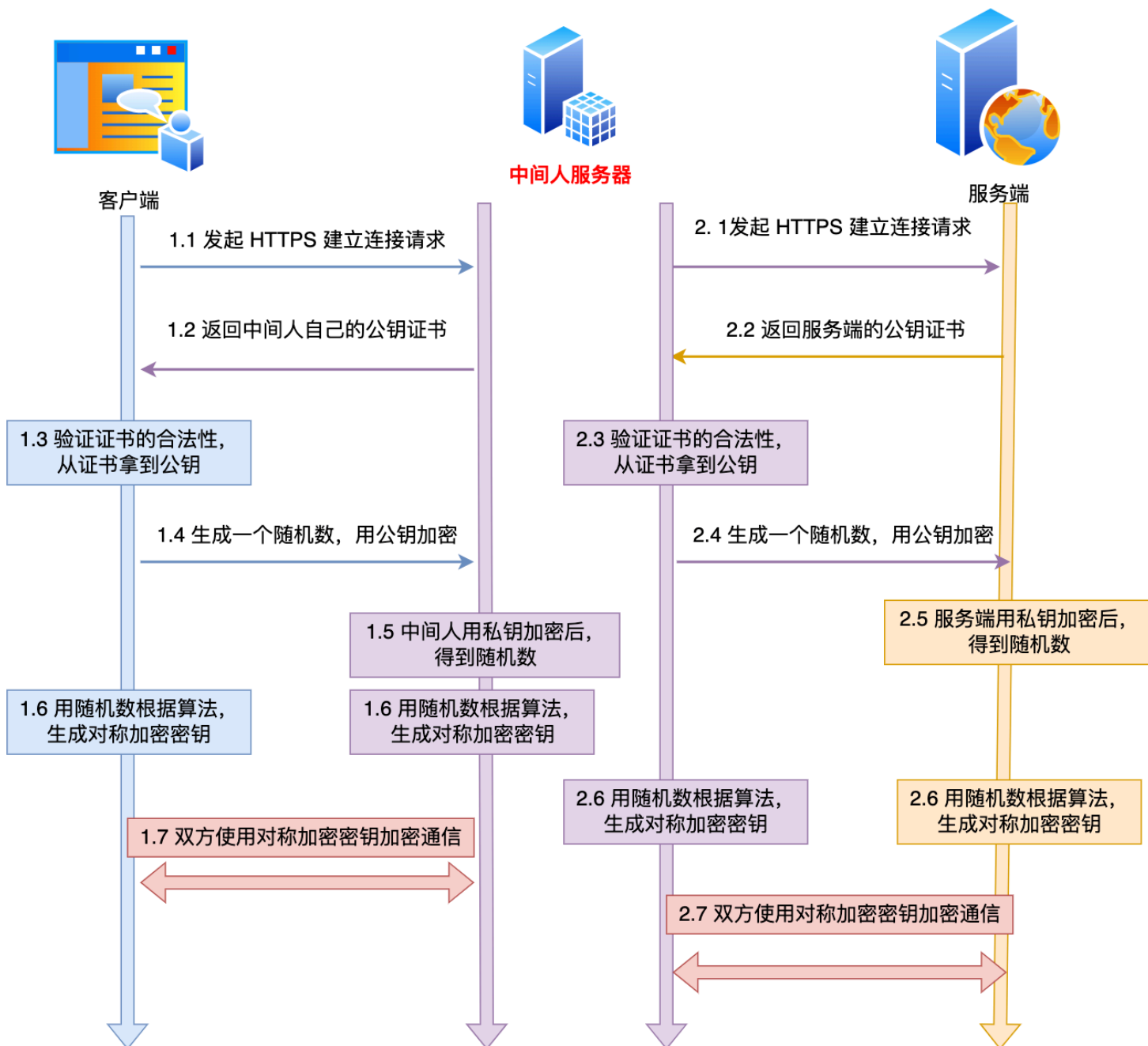
防范

- **Token 验证**：服务器发送给客户端一个 token，客户端提交的表单中带着这个 token，如果这个 token 不合法，那么服务器拒绝这个请求
- **隐藏令牌**：把 token 隐藏在 http 的 head 头中
- **Referer 验证**：只接受本站的请求，服务器才做响应；如果不是，就拦截

中间人攻击以及如何防范？

概念

指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制。



- **嗅探：**嗅探或数据包嗅探 是一种用于捕获流进和流出系统/网络的数据包的技术。网络中的数据包嗅探就好像电话中的监听
- **数据包注入：**攻击者会将恶意数据包注入常规数据中。这样用户便不会注意到文件/恶意软件，因为它们都是合法通讯流的一部分。在中间人攻击和拒绝式攻击中，这些文件是很常见的

- **会话劫持**：当客户端和服务端在进行一个会话时，会话中包含了很多重要信息，一些黑客会潜伏在这个会话中，最终控制这个会话，这既是会话劫持
- **SSL 剥离**：SSL/TLS 证书通过加密保护着的通讯安全。在 SSL 剥离攻击中，攻击者使 SSL/TLS 连接剥落，随之协议便从安全的 HTTPS 变成了不安全的 HTTP

防范

使用 HTTPS 协议，禁用不安全的 SSL 协议，启用虚拟专用网(VPN)。

通过 **HTTPS 双向认证**来避免这种问题：一般 HTTPS 是单向认证，客户端只会验证了服务端的身份，但是服务端并不会验证客户端的身份。如果用了双向认证方式，不仅客户端会验证服务端的身份，而且服务端也会验证客户端的身份。服务端一旦验证到请求自己的客户端为不可信任的，服务端就拒绝继续通信，客户端如果发现服务端为不可信任的，那么也中止通信。

HTTP 1.1 如何优化？

避免发送 HTTP 请求

通过**缓存技术**将请求-响应的数据都缓存在本地，下次请求直接读取本地的数据。

客户端会将第一次请求以及响应的数据保存在本地硬盘，其中将请求的 URL 作为 key，将相应作为 value。当后续发起相同的请求时可以先在本地硬盘上通过 key 查找对应的 value。

服务器在发送 HTTP 响应时会估算一个过期时间，并把信息放到响应头部中。当客户端发现响应过期时会重新向服务器发送请求，如果服务器上资源没有变更，那么服务器会返回**不含有响应体的 304 Not Modified 响应**。

减少 HTTP 请求次数

- **减少重定向请求次数**：重定向的工作可以交由代理服务器完成
- **合并请求**：一般浏览器会同时发起 5-6 个请求，服务端通过将很多小图片利用 CSS Image Sprites 技术将它们合称为一个大图片，或者通过 Base64 编码将图片嵌入到 HTML 文件实现合并请求
- **延迟发送请求**：只有在用户向下滑动页面的时候再发送请求，对资源**按需获取**

减少 HTTP 响应数据大小

- **无损压缩**：去除一些为了代码格式美观而使用的空格、换行；利用霍夫曼编码对资源进行压缩
- **有损压缩**：例如对图片可以使用 **WebP** 格式压缩、对视频使用 **H264、H265** 等格式压缩

HTTPS 如何优化?

硬件优化

HTTPS 是**计算密集型**，所以应从优化 CPU 入手。可以选择支持 AES-NI 特性的 CPU，因为这种 CPU 在指令级别优化了 AES 算法，加速加密数据的传输。

软件优化

- **软件升级**：升级 Linux 内核版本
- **协议升级**：升级协议软件版本

协议优化

- **密钥交换算法优化**：尽量选用 ECDHE 密钥交换算法替换 RSA 算法
- **TLS 升级**：TLS1.3 大幅简化了握手的步骤，**完成握手仅需 1 RTT**，安全性也更高。实现方法是 TLS1.3 将 Hello 和公钥交换两个消息合并成了一个消息

证书优化

- **证书传输优化**：尽量选择椭圆曲线 ECDSA 证书
- **证书验证优化**：客户端进行逐级验证证书可靠性是一个复杂的过程，可以着手在这方面进行优化

会话复用

会话复用使用的以下三种技术都有不能避免**重放攻击**：

- **Session ID**：客户端和服务端首次 TLS 握手连接后，双方在内存缓存会话密钥，并用唯一的 Session ID 进行标识。当客户端再次连接时，Hello 报文中会携带 Session ID，只要 ID 没有过期就可以通过 1 RTT 实现握手
- **Session Ticket**：服务器不再缓存每个客户端的会话密钥，转而将缓存工作交给客户端。客户端与服务器首次连接时服务器加密会话密钥作为 Ticket 发送给客户端，客户端缓存该 Ticket。客户端再次连接服务器时发送该 Ticket，服务器解码 Ticket 并检查有效期即可恢复会话
- **Pre-shared Key**：TLS1.3 通过 Pre-shared Key 实现 0 RTT 完成握手

● TCP 和 UDP

TCP 的头部结构



- **源端口**：16 位，标识报文的返回地址
- **目的端口**：16 位，指明接收方计算机上的应用程序接口
- **序列号**：32 位，在建立连接时由计算机生成的随机数作为其初始值，通过 SYN 包传给接收端主机，每发送一次数据，就累加一次该数据字节数的大小。用来解决网络包乱序问题
- **确认号**：32 位，指下一次期望收到的数据的序列号，发送端收到这个确认应答以后可以认为在这个序号以前的数据都已经被正常接收。用来解决丢包的问题
- **数据偏移/首部长度**：4 位，TCP 首部可能含有可选项内容，所以 TCP 报头的长度是不确定的，报头不包含任何任选字段则长度为 20 字节，4 位首部长度字段所能表示最大长度为 60 字节。首部长度也叫数据偏移，因为首部长度实际上指示了数据区在报文段中的起始偏移值
- **保留**：6 位，为将来定义新的用途保留，现在一般置 0
- **校验和**：16 位，由发送端填充，接收端对 TCP 报文段执行 CRC 算法以检验 TCP 报文段在传输过程中是否损坏，这个校验不仅包括 TCP 头部，也包括数据部分。这是 TCP 实现可靠传输的一个重要保障
- **窗口**：16 位，是 TCP 流量控制的一个手段。通过窗口告诉对方本端的 TCP 接收缓冲区还能容纳多少字节的数据，这样对方可以控制发送数据的速度，从而达到流量控制。窗口大小为 16 bit 字段，因而窗口大小最大为 65535
- **紧急指针**：16 位，只有当 URG 标志置 1 时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。使用紧急指针是发送端向另一端发送紧急数据的一种方式

- **选项和填充**：TCP 头部的最后一个选项字段是可变长的可选信息。这部分最多包含 40 字节，因为 TCP 头部最长是 60 字节。最常见的可选字段是最长报文大小 MSS，每个连接方通常都在通信的第一个报文段中指明这个选项，它表示本端所能接受的最大报文段的长度
- **数据部分**：TCP 报文段中的数据部分是可选的。在连接建立或者终止时，双方交换的报文段仅有 TCP 首部；如果一方没有数据要发送，也会使用没有任何数据的首部来确认收到的数据；在处理超时的许多情况中，也会发送不带任何数据的报文段

还包括控制位：

- **URG**：紧急指针标志，为 1 时表示紧急指针有效，该报文应该优先传送，为 0 则忽略紧急指针
- **ACK**：确认序号标志，为 1 时表示确认号有效，为 0 表示报文中不含确认信息。携带 ACK 标识的 TCP 报文段被称为确认报文段
- **RST**：重置连接标志，用于重置由于主机崩溃或其他原因而出现错误的连接，或者用于拒绝非法的报文段和拒绝连接请求。称携带 RST 标志的 TCP 报文段为复位报文段
- **SYN**：表示请求建立一个连接。称携带 SYN 标志的 TCP 报文段为同步报文段
- **FIN**：finish 标志，用于释放连接，为 1 时表示发送方已经没有数据发送了，即关闭本方数据流。称携带 FIN 标志的 TCP 报文段为结束报文段
- **PSH**：push 标志，为 1 表示是带有 push 标志的数据，指示接收方在接收到该报文段以后，应优先将这个报文段交给应用程序，而不是在缓冲区排队

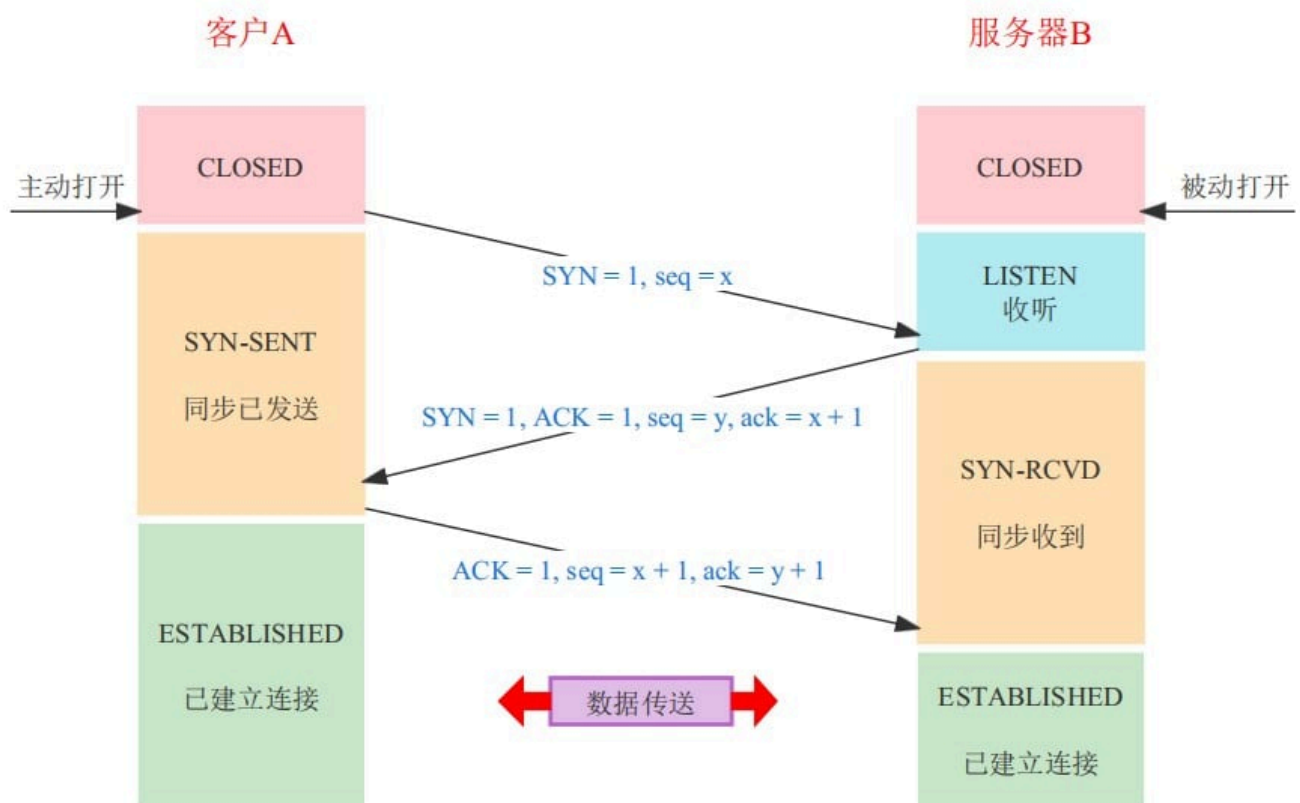
TCP 如何保证可靠传输

- **校验和**：目的是为了验证 TCP 首部和数据在发送过程中没有任何改动，一旦发现校验和有差错，直接丢弃 TCP 段并重新发送
- **序列号/确认号**：序列号用于解决包乱序问题，确认号用于解决包丢失问题
- **连接管理**：三次握手、四次挥手
- **流量控制**：基于滑动窗口机制实现。TCP 的报文信息中有一个 16 位字段来标识滑动窗口，窗口大小就是接收方剩余缓冲区大小，在回复 ACK 时，接收方将自己剩余缓冲区大小填入。发送方根据窗口大小来调整自己的发送速度，如果缓冲区大小为 0，那么发送方会停止发送数据。并且发送方定期会发送探测报文，来获取缓冲区大小
- **拥塞控制**：拥塞控制与流量控制相比，更加关注网络线路中的拥塞程度，即网络的包转发还要受到当前网络端到端路由的拥塞情况
 - **慢启动算法**：一开始不发送大量数据，而是应该先发一小部分探测数据，然后由小到大逐渐增大发送窗口。通常在刚刚开始发送报文段时，先把拥塞窗口 cwnd 设置为 1，每次接收到报文之后将窗口大小翻倍。如果指数增长到避免拥塞算法的门限 ssthresh，则改用避免拥塞算法
 - **拥塞避免算法**：每当收到一个 ACK 时，cwnd 增加 1，变为线性增长。一旦发现丢包和超时重传，就进入拥塞处理状态
 - **拥塞发生**：

- **超时重传**：ssthresh 设为 $cwnd/2$ ；cwnd 重置为 1；回到慢启动过程
- **快速重传**：当发送方在超时之前收到来自接收方的 3 个冗余 ACK， $cwnd = cwnd/2$ ，也就是设置为原来的一半；ssthresh = cwnd；进入快速恢复算法
- **快速恢复**：拥塞窗口 $cwnd = ssthresh + 3$ ；因为 3 个冗余 ACK 已经收到，所以窗口应该增加 3。重传丢失的数据包

如果再收到重复的 ACK，那么 cwnd 增加 1；如果收到新数据的 ACK 后，把 cwnd 设置为第一步中的 ssthresh 的值，原因是该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态

TCP 的三次握手



- **第一次握手**： $SYN = 1; seq = x$ ；进入 SYN_SENT 状态
- **第二次握手**： $SYN = 1; ACK = 1; seq = y; ack = x + 1$ ；进入 SYN_RCVD 状态
- **第三次握手**： $ACK = 1; seq = x + 1; ack = y + 1$ ；进入 ESTABLISHED 状态

TCP 为什么要三次握手？

只有三次握手才能证明服务端和客户端的收发能力都是正常的。

- **第一次握手**：服务端可以知道客户端发消息的能力是正常的，自己接收消息的能力是正常的
- **第二次握手**：客户端可以知道自己发送接收消息的能力和服务端发送接收消息的能力是正常的
- **第三次握手**：服务端可以知道自己发送消息的能力是正常的，客户端接收消息的能力是正常的

由此经过三次握手之后双方就可以都知道自己的发送和接收消息的能力是正常的。

如果 TCP 的三次握手丢失会发生什么？

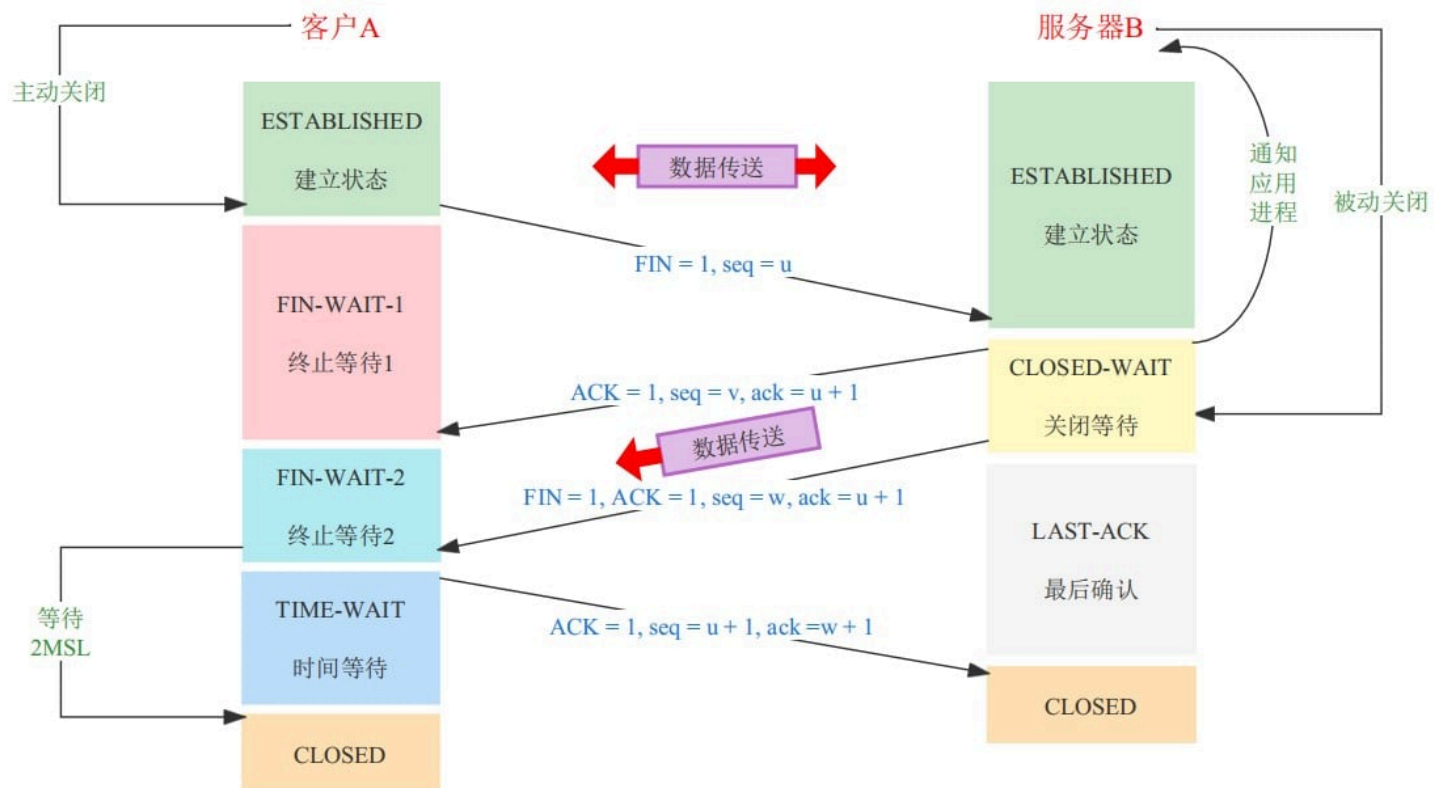
- **第一次丢失**：客户端发送的 SYN 报文会收不到服务端的响应，从而会**触发超时重传**，重传的 SYN 报文序列号和之前相同，重传最大重传次数由内核参数控制，一般是 5。如果超过最大次数客户端**仍没有收到回复就会断开连接**
- **第二次丢失**：服务端在收到客户端的报文之后会回复 SYN + ACK 报文。如果第二次握手丢失了客户端会认为自己丢包了，**触发超时重传**，重新发送 SYN 报文，服务端因为收不到确认的 ACK **自身也会重传**
- **第三次丢失**：客户端收到服务端的 SYN-ACK 报文后会给服务端回一个 ACK 报文，此时客户端状态进入到 ESTABLISH 状态。如果发生了丢包，服务端收不到 ACK 会触发超时重传机制，重传 SYN-ACK 报文，直到收到确认 ACK 或者达到最大重传次数

TCP 为什么不是两次握手？

最好再加上 **如果 TCP 的三次握手丢失会发生什么？**

- **避免历史连接(首要原因)**：如果使用的是两次握手建立连接，可能客户端发送的第一个请求连接并且没有丢失，只是因为网络中滞留的时间太长了，由于 TCP 的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务端经过两次握手完成连接，传输数据，然后关闭连接。之前滞留的那一次请求连接，因为网络通畅了，到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务端再次建立连接，这将导致不必要的错误和资源的浪费
- **同步双方初始序列号**：为了实现可靠数据传输，TCP 协议的通信双方，都必须维护一个序列号，以标识发送出去的数据包中，哪些是已经被对方收到的。三次握手的过程即是通信双方相互告知序列号起始值，并确认对方已经收到了序列号起始值的必经步骤。如果只是两次握手，至多只有连接发起方的起始序列号能被确认，另一方选择的序列号则得不到确认

TCP 的四次挥手？



- **第一次挥手：**客户端发送释放报文，并停止发送数据，将首部的 FIN 标识位置为 1，序列号 seq = u 发送给服务器，值等于前面已经传送过来的数据的最后一个字节的序号加 1，此时客户端进入 **FIN_WAIT_1** 状态。即便 FIN 报文不携带数据，也要消耗一个序列号
- **第二次挥手：**服务器在收到释放报文后，发送确认报文，ACK 标识位置为 1，ack 值为客户端发送的序列号 u + 1，并带上自己的序列号 v，然后服务器进入 **CLOSE_WAIT** 关闭等待状态。这时服务器 TCP 通知高级应用进程：客户端向服务器的连接释放了，进入半关闭状态。但是服务器如果向客户端发送数据，客户端仍然可以接收，这个状态要持续一段时间，也就是 **CLOSE_WAIT** 关闭等待持续的时间。客户端收到服务器的确认请求后，进入 **FIN_WAIT_2** 状态，等待服务器发送释放报文
- **第三次挥手：**服务器数据处理完毕后向客户端发送释放连接报文，FIN 标识位置为 1，ack 的值为客户端的序列号 u + 1。由于在半关闭状态，服务器很可能又发送一些数据，假定此时序列号为 w，服务器进入 **LAST_ACK** 状态，等待客户端确认
- **第四次挥手：**客户端在收到服务器的释放连接报文后，会发送确认报文，ACK 标识位置为 1，ack 值为服务器发送的序列号 w + 1，自己的序列号是 u + 1，然后客户端就进入 **TIME_WAIT** 状态。此时 TCP 连接还没有释放，必须经过**两个 MSL 时间**(一个 MSL 指的是报文段最长寿命)，当客户端撤销 TCB，才进入 **CLOSED** 状态。服务器只要收到客户端发送的确认请求，立即进入 **CLOSED** 状态。同时会撤销 TCB，TCP 连接至此结束

TCP 为什么要四次挥手？

关闭连接时，客户端向服务端发送 **FIN** 时，仅仅表示客户端不再发送数据了但是还能接收数据。

服务端收到客户端的 **FIN** 报文时，先回一个 **ACK** 应答报文，而服务端可能还有数据需要处理和发送，等服务端不再发送数据时，才发送 **FIN** 报文给客户端来表示同意现在关闭连接。

在 **FIN_WAIT_2** 状态下，是如何处理收到的乱序到 **FIN** 报文，然后 **TCP** 连接又是什么时候才进入到 **TIME_WAIT** 状态？

在 **FIN_WAIT_2** 状态时，如果收到乱序的 **FIN** 报文会加入到乱序队列，并不会进入到 **TIME_WAIT** 状态。等再次收到前面被网络延迟的数据包时，会判断乱序队列有没有数据，检测乱序队列中是否有可用的数据，如果能在乱序队列中找到与当前报文的序列号保持的顺序的报文，就会看该报文是否有 **FIN** 标志，如果发现有 **FIN** 标志，才会进入 **TIME_WAIT** 状态

如果 **TCP** 的四次挥手丢失会发生什么？

- **第一次丢失**：客户端发送的报文 **FIN** 报文收不到服务端的 **ACK** 响应，会触发超时重传，重传 **FIN** 报文，重发次数由内核参数控制
- **第二次丢失**：服务端回复的 **ACK** 报文发生丢失，客户端会触发超时重传，重传 **FIN** 报文，直到收到服务端的 **ACK** 或者达到最大的重传次数。超过最大重传次数还没收到 **ACK** 会等待一段时间，再断开连接
- **第三次丢失**：服务端收到客户端的 **FIN** 报文后内核会自动回复 **ACK**，同时连接处于 **CLOSE_WAIT** 状态。服务端处于 **CLOSE_WAIT** 状态时，调用了 **close** 函数，内核会发出 **FIN** 报文，同时连接进入 **LAST_ACK** 状态，等待客户端返回 **ACK** 来确认连接关闭。收不到 **ACK** 的话会重发 **FIN** 报文直到最大次数为止
- **第四次丢失**：最后一次的 **ACK** 发生了丢失，服务端没有收到 **ACK** 报文前是处于 **LAST_ACK** 状态。超时之后服务端会重传 **FIN** 报文，客户端此时是在 **TIME_WAIT** 状态，开启时长为 **2MSL** 的定时器，如果途中再次收到第三次挥手(**FIN** 报文)后，会重置定时器，当等待 **2MSL** 时长后，客户端会断开连接

TCP 为什么不是三次挥手？

当被动关闭方在 **TCP** 挥手过程中，没有数据要发送并且开启了延迟应答，第二和第三次挥手就会合并传输，这样就出现了三次挥手。

TCP 的延迟应答和累计应答？

- **延迟应答**：TCP 在接收到对端的报文后并不会立即发送 ACK，而是等待一段时间发送 ACK，以便将 ACK 和要发送的数据一块发送。延迟时间不能无限延长，否则对方端会认为丢包超时而造成超时重传。Linux 采用动态调节算法来确定等待的时间
- **累计应答**：为了保证顺序性，每一个包都有一个 ID(序号)，在建立连接的时候，双方会商定起始的 ID 是多少，然后按照 ID 一个个发送。为了保证不丢包，对应发送的包都要进行应答，但不是一个个应答，而是会应答**某个之前的 ID**，该模式称为**累计应答**

TCP 的 MSL

MSL 是任何报文在网络中被丢弃前的最长存活时间，这个时间是有限的，因为 TCP 是以 IP 数据报的形式在网络中传输，IP 有限制其生存的时间 TTL，RFC793 指出 MSL 为 **2 分钟**，现实中常用 **30 秒** 或 **1 分钟**

已经建立了连接，客户端突然出现故障了会怎样？

TCP 存在保活计时器，如果客户端故障，服务器不会一直等待。通常计时器设置为两小时，在每次收到客户端发来的报文都会重置计时器，超时之后服务端就会向客户端发送探测报文，每隔 75s 发送一次，如果连续 10 个探测报文都没有收到回复，服务器会认为客户端发生故障，中断此次连接

什么时候用长连接，短连接？

长连接多用于**操作频繁，点对点的通讯，而且连接数不能太多**的情况。每个 TCP 连接都需要三步握手，这需要时间。如果每个操作都是先连接，再操作的话那么处理速度会降低很多。所以每个操作完后都不断开，下次处理时直接发送数据包就可以，不用建立 TCP 连接。例如：**数据库的连接用长连接**。

WEB 网站的 HTTP 服务一般都用短连接，因为长连接对于服务端来说会耗费一定的资源，而 WEB 网站成千上万客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用戶，如果每个用戶都占用一个连接的话，所以并发量大，用短连接可以快速释放资源。

TCP 的半连接队列和全连接队列？

- **半连接队列**：也称 **SYN 队列**，服务端收到客户端发起的 SYN 请求后，内核会把该连接存储到半连接队列，并向客户端发 **SYN + ACK**
- **全连接队列**：也称 accept 队列，服务端收到第三次握手的 ACK 后，**内核会把连接从半连接队列移除，然后创建新的完全的连接，并将其添加到全连接队列**，等待进程调用 accept 函数时把连接取

出来

什么是 SYN 攻击？如何避免？

概念

SYN 攻击是指利用合理的服务请求来占用过多的服务资源，从而使合法用户无法得到服务的响应。如果向某个服务器端口发送大量的 SYN 报文，接收到客户端发来的 SYN 报文之后，服务端就需要为每个请求分配一个进程控制块 TCB，并返回一个 SYN-ACK 报文，并立即转为 SYN_RECV 半开连接状态，收不到对端 ACK 回复的服务端还会重传 SYN-ACK 报文，系统会为此耗尽资源

避免方法

- **Cache**：系统在收到一个 SYN 报文时，在一个专用 HASH 表中保存这种半连接信息，直到收到正确的回应 ACK 报文再分配 TCB。这个开销远小于 TCB 的开销
- **Cookie**：利用算法，通过对方的 IP、端口、己方 IP、端口的固定信息，以及对方无法知道而己方比较固定的一些信息，如 MSS(最大报文段大小)、时间等，在收到对方的 ACK 报文后，重新计算一遍，看其是否与对方回应报文中的**(Sequence Number-1)**相同，从而决定是否分配 TCB 资源
- **Proxy 防火墙**：设立中间层防火墙，防火墙在确认了连接的有效性后，才向内部的服务器发起 SYN 请求，所有的无效连接均无法到达内部的服务器。而防火墙采用的验证连接有效性的方法则可以是 **Cookie** 或 **Cache** 等其他技术
- **减少 SYN+ACK 重传次数**：减少 SYN-ACK 的重传次数，以加快处于 **SYN_RECV** 状态的 TCP 连接断开
- **无效连接监视释放**：不停监视系统的半开连接和不活动连接，当达到一定阈值时拆除这些连接，从而释放系统资源。这种方法对于所有的连接一视同仁，正常连接请求也会被这种方式误释放掉
- **增大半连接队列**：修改 TCP 的内核参数，增大半连接队列以及全连接队列大小
- **调大 netdev_max_backlog**：当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包，可以调大队列大小

TIME_WAIT 作用，过多如何解决？

作用

- **实现全双工的可靠释放连接**：假设发起主动关闭的一方最后发送的 ACK 在网络中丢失，由于 TCP 的重传机制，被动关闭的一方会重新发送 FIN 报文，在 FIN 在被主动关闭方接收之前，主动关闭方都需要维护这条连接状态，包括对应的 IP 地址和端口号。如果发送方不维护 TIME_WAIT 状态，

那么当 FIN 到达主动关闭方的时候，主动关闭方会发送 RST 包来响应，被动关闭方就会认为有错误发生

- **为使旧的数据包在网络因过期而消失**：如果不存在 **TIME_WAIT** 状态，当前的一个 TCP 四元组因为某些原因关闭之后，假设有一个新的相同的四元组建立了 TCP 连接，因为 TCP 连接是由四元组唯一标识的，所以没法区分新旧连接。旧的已经关闭的 TCP 连接发送的数据到达接受方之后，会被当作正常数据而向上传输，从而导致数据错乱。有了 **TIME_WAIT** 状态之后，可以使旧 TCP 产生的数据包全部在网络中消亡

危害

- **占用系统资源**，比如文件描述符、内存资源、CPU 资源、线程资源等
- **占用端口资源**，端口资源也是有限的，一般可以开启的端口为 32768~61000，也可以通过 **net.ipv4.ip_local_port_range** 参数指定范围

避免方法

- 修改短连接为长连接
- 扩大可使用端口号的范围
- 客户端机器打开 **tcp_tw_reuse** 和 **tcp_timestamps** 选项：**tcp_tw_reuse** 调用 connect() 函数时，内核会随机找一个 time_wait 状态超过 1 秒的连接给新的连接复。复用连接之后需要更新 **timestamps** 参数，当旧的 TCP 数据包到达时，根据时间戳判断是旧连接的数据可以舍弃
- 客户端机器打开 **tcp_tw_recycle** 和 **tcp_timestamps** 选项：当开启之后内核会快速回收 **TIME_WAIT** 状态的连接，时间是一个 RTO，远小于两个 MSL。在启用该配置，当连接进入 **TIME_WAIT** 状态后，内核里会记录包括该连接对应五元组的一些统计数据，包括从该对方 IP 所接收到的最近的一次数据包时间。当有新的数据包到达，只要时间晚于内核记录的这个时间，数据包都会被统统的丢掉
- 缩小 **net.ipv4.tcp_max_tw_buckets**：当系统中处于 **TIME_WAIT** 的连接一旦超过这个值时，系统就会将后面的 **TIME_WAIT** 连接状态重置
- 程序中使用 **SO_LINGER**：调用 close 后，会立刻发送一个 RST 标志给对端，该 TCP 连接将跳过四次挥手，也就跳过了 **TIME_WAIT** 状态，直接关闭

tcp_tw_reuse 为什么默认是关闭的？

- 历史 RST 报文可能会终止后面相同四元组的连接，因为 PAWS 检查到即使 RST 是过期的，也不会丢弃
- 如果第四次挥手的 ACK 报文丢失了，有可能被动关闭连接的一方不能被正常的关闭

PAWS 的保护机制

- **序列号回绕**：当序列号在高数据传输速率下回绕时，旧的数据包可能被误认为新的。PAWS 通过时间戳机制，确保只有时间上更新的数据包被接受。
- **连接重启后的旧数据包**：在连接关闭后，若同一对端口重新建立连接，之前的旧数据包可能会被误认为新的。PAWS 通过时间戳机制，确保只有在当前连接中接收到的最新数据包被接受

潜在问题：

- **NAT(网络地址转换)设备的时间戳问题**：某些 NAT 设备可能会修改 TCP 数据包中的时间戳，导致 PAWS 机制错误地丢弃合法的数据包
- **时间戳同步问题**：如果连接双方的系统时间不同步，可能导致 PAWS 机制误判数据包的有效性

TIME_WAIT 状态为什么需要经过 2MSL

因为客户端最后一个发送的 **ACK** 有可能丢失。假如服务器没有收到客户端发送的最后一个 ACK，就会重新发送 FIN 报文，为了确保服务器收到了 FIN 报文，客户端在 **TIME_WAIT** 状态需要经过 2MSL，在这个期间客户端收到重发的 FIN 报文就会重新发送 ACK 并且重设计时器。MSL 指一个片段在网络中最大的存活时间，2MSL 就是一个发送和一个回复所需的最大时间。第一个 MSL 是保证最后一次挥手客户端响应服务端的 ACK 到达了服务端。第二个 MSL 是保证服务端没有重发新的报文给客户端，没有超时重传。

如果客户端直接关闭，然后向服务器建立新连接，如果新连接和老连接的端口是一样的。假设老连接还有一些数据，因为网络或者其他原因，一直滞留没有发送成功，新连接建立后，就直接发送到新连接里面去了，造成数据的紊乱，因此，需要等到2MSL，让滞留在网络中的报文失效，再去建立新的连接。

2MSL 的时间是从客户端接收到 FIN 后发送 ACK 开始计时的。如果在 TIME-WAIT 时间内，因为客户端的 ACK 没有传输到服务端，客户端又接收到了服务端重发的 FIN 报文，那么 2MSL 时间将重新计时

CLOSE_WAIT 状态过多如何解决？

如果一直保持在 **CLOSE_WAIT** 状态，原因是在对方关闭连接之后服务器程序自己没有进一步发出 ACK 信号。

CLOSE_WAIT 的解决办法是：查代码。因为问题出在服务器程序

TCP 和 UDP 的区别？

- **TCP 是面向连接的。**在通信之前需要三次握手建立连接，通信之后断开连接时需要四次挥手；UDP 不需要进行连接建立
- **TCP 是可靠传输服务。**通过 TCP 传输数据可以保证数据无差错、不丢失、不重复；UDP 尽最大努力交付，不保证可靠交付
- **每个 TCP 对应的是点对点的连接；UDP 支持一对一、一对多、多对一、多对多等多种方式的通讯**
- **UDP 对系统资源要求较少，通讯效率高，实时性好，**应用于高速传输并且对实时性有要求的通信；**TCP 适合需要可靠连接，**比如付费、加密数据等等方向都需要依靠 TCP
- **TCP 首部长度较长，**会有一定的开销，首部在没有使用「选项」字段时是 20 个字节，如果使用了「选项」字段则会变长的；**UDP 首部只有 8 个字节，并且是固定不变的**
- **TCP 是流式传输，**没有边界，但保证顺序和可靠。**UDP 是一个包一个包的发送，**是有边界的，可能会丢包和乱序
- **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
- 应用场景：TCP 用于 FTP 文件传输、HTTP / HTTPS；UDP 用于包总量较少的通信，如 DNS、SNMP 等、视频、音频等多媒体通信、广播通信

粘包和拆包问题的解决办法？

概念

- TCP 的特点之一就是面向字节流的，也就是说传输时候数据像“水流一样”，是没有边界的，因此**拆包这个功能本身就不在 TCP 来完成**
- 所谓的粘包拆包就是 TCP 流的特性导致的，而且根本不能说是问题，拆包本身就应该在应用层来完成

解决办法

- 遇到这个面试题，作者个人认为是面试官基础不扎实才会问出来，就直接怼它

TCP 的 keepalive 和 HTTP 的 keepalive 的区别？

- 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 Keep-Alive** 是指使用同一个 TCP 连接来发送和接收多个 HTTP 请求/应答，好处是避免了连接建立和释放的开销，只要任意一端没有明确提出断开连接，就保持 TCP 连接状态；**TCP Keepalive** 是指建立 TCP 连接的两端一直没有数据交互达到触发 TCP 保活机制的条件，内核里的 TCP 协议栈就会发送探测报文，如果对端程序正常工作，收到探测报文之后就会回复响应，同时保活时间重置，如果对端主机崩溃没有响应或者网络原因报文不可达，连续几次探测报文之后 TCP 会报告该 TCP 连接已经死亡
- web 服务软件一般都会提供 `keepalive_timeout` 参数来指定 HTTP 长连接的超时时间。例如设置了 HTTP 长连接的超时时间是 60 秒，web 服务软件就会启动一个定时器，如果客户端在完成一个 HTTP 请求后，在 60 秒内都没有再发起新的请求，定时器的时间一到，就会触发回调函数来释放该连接

IP 层会分片，为什么 TCP 层还需要 MSS 呢？

- **MTU**：一个网络包的最大长度，以太网中一般为 1500 字节
- **MSS**：除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度

如果交给 IP 来进行分片，一个 IP 分片丢失，整个 IP 报文的所有分片都得重传。因为 IP 层本身没有超时重传机制，它由传输层的 TCP 来负责超时和重传。当某一个 IP 分片丢失后，接收方的 IP 层就无法组装成一个完整的 TCP 报文(头部 + 数据)，也就无法将数据报文送到 TCP 层，所以接收方不会响应 ACK 给发送方，因为发送方迟迟收不到 ACK 确认报文，所以会触发超时重传，就会重发整个 TCP 报文(头部 + 数据)。



DNS 查询服务器的基本流程？

- 客户机向其本地域名服务器发出 DNS 请求报文
- 本地域名服务器收到请求后，查询本地缓存，假设没有该记录，则以 DNS 客户的身份向**根域名服务器**发出解析请求
- 根域名服务器收到请求后，判断该域名所属域，将对应的**顶级域名服务器**的 IP 地址返回给本地域名服务器
- 本地域名服务器向顶级域名服务器发出解析请求报文
- 顶级域名服务器收到请求后，将所对应的**授权域名服务器**的 IP 地址返回给**本地域名服务器**
- 本地域名服务器向授权域名服务器发起解析请求报文
- 授权域名服务器收到请求后，将**查询结果返回给本地域名服务器**
- 本地域名服务器将查询结果**保存到本地缓存**，同时返回给客户机

DNS 采用 TCP 还是 UDP，为什么？

DNS 在**进行区域传输的时候使用 TCP 协议**，其它时候则使用 UDP 协议。TCP 与 UDP 传送字节的长度限制不同，一般情况下一个 DNS 的 UDP 包的最大长度是 512 字节。

区域传输使用 TCP 协议的原因大概是：

- 区域传输的数据量相比单次 DNS 查询的数据量要大得多
- 区域传输对数据的可靠性和准确性相比普通的 DNS 查询要高得多，因此使用 TCP 协议

域名解析时一般返回的内容都不超过 512 字节，首选的通讯协议是 UDP。使用 UDP 传输，不用经过 TCP 三次握手，这样 DNS 服务器负载更低，响应更快。

DNS 劫持是什么？解决办法？

概念

- **本地 DNS 劫持**：攻击者在用户的计算机上安装木马恶意软件，并更改本地 DNS 设置以将用户重定向到恶意站点
- **路由器 DNS 劫持**：攻击者接管路由器并覆盖 DNS 设置，从而影响连接到该路由器的所有用户
- **中间人 DNS 攻击**：攻击者拦截用户和 DNS 服务器之间的通信，并提供指向恶意站点的不同目标 IP 地址

解决方法

- **加强域名账户的安全防护能力**，使用有别于其他平台的用户名和强密码，定期对密码进行更换
- **定期查看域名账户信息、域名 whois 信息、域名解析状态**，每天 site 网站检查是否存在非个人设定网页，发现异常及时联系域名服务商
- **锁定域名解析状态**，不允许通过 DNS 服务商网站修改记录，使用此方法后，需要做域名解析都要通过服务商来完成，这样就可以从根本上杜绝通过攻击服务商修改解析记录的方法

浏览器输入一个 URL 到显示器显示的过程？

- **键盘输入**：输入键盘字符后键盘就会产生扫描数据，并将其缓冲存在寄存器中，然后键盘通过总线给 CPU 发送中断请求。CPU 收到中断请求后，操作系统会保存被中断进程的 CPU 上下文，然后调用键盘的中断处理程序。键盘中断处理函数从键盘的寄存器的缓冲区读取扫描码，再根据扫描码找到用户在键盘输入的字符的 ASCII 码。然后把 ASCII 码放到读缓冲区队列，显示器会定时从读缓

缓冲区队列读取数据放到写缓冲区队列，最后把写缓冲区队列的数据一个一个写入到显示器的寄存器中的数据缓冲区，最后将这些数据显示在屏幕里

- **URL 解析**：浏览器会首先从缓存中找是否存在域名，如果存在就直接取出对应的 IP 地址，如果没有就开启一个 DNS 域名解析器。DNS 域名解析器会首先访问顶级域名服务器，将对应的 IP 发给客户端；然后访问根域名解析器，将对应的 IP 发给客户端；最后访问本地域名服务器，得到最终的 IP 地址
- **TCP 连接**：在 URL 解析过程中得到真实的 IP 地址之后，会调用 Socket 函数建立 TCP 连接
- **HTTP 请求**：浏览器向服务器发起一个 HTTP 请求，HTTP 协议是建立在 TCP 协议之上的应用层协议，其本质是建立起的 TCP 连接中，按照 HTTP 协议标准发送一个索要网页的请求。请求包含请求行、请求头、请求体三个部分组成，有 GET、POST 等主要方法
- **浏览器接受响应**：服务器在收到浏览器发送的 HTTP 请求之后，会将收到的 HTTP 报文封装成 HTTP 的 Request 对象，并通过不同的 Web 服务器进行处理，处理完的结果以 HTTP 的 Response 对象返回，主要包括状态码，响应头，响应报文三个部分
- **页面渲染**：浏览器根据响应开始显示页面，首先解析 HTML 文件构建 DOM 树，然后解析 CSS 文件构建渲染树，等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上
- **断开连接**：客户端和服务端通过四次挥手终止 TCP 连接

PING 是怎么工作的？

ping 命令执行的时候，源主机首先会构建一个 **ICMP 回送请求消息数据包**，由 ICMP 协议将这个数据包连同服务端 IP 一起交给 IP 层，IP 层将以服务端 IP 作为目的地址，本机 IP 地址作为源地址，协议字段设置为 1，再加上一些其他控制信息，构建一个 IP 数据包；然后加入 MAC 头；如果在本地 ARP 映射表中查找出服务端 IP 所对应的 MAC 地址，则可以直接使用，如果没有，则需要发送 ARP 协议查询 MAC 地址。**获得 MAC 地址后，由数据链路层构建一个数据帧，目的地址是 IP 层传过来的 MAC 地址，源地址则是本机的 MAC 地址**；还要附加上一些控制信息，依据以太网的介质访问规则将它们传出去。

目的主机收到这个数据帧后，先检查它的目的 MAC 地址，并和本机的 MAC 地址对比，如符合，则接收，否则就丢弃。接收后检查该数据帧，将 IP 数据包从帧中提取出来，交给本机的 IP 层。IP 层检查后，将有用的信息提取后交给 ICMP 协议。**主机 B 会构建一个 ICMP 回送响应消息数据包**，回送响应数据包的类型字段为 0，序号为接收到的请求数据包中的序号，然后再发送出去给主机 A。

在规定的时间内，源主机如果没有接到 ICMP 的应答包，则说明目标主机不可达；如果接收到了 ICMP 回送响应消息，则说明目标主机可达。

Cookie 和 Session 的关系和区别是什么？

Cookie 概念

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，它会在浏览器下次向同一服务器再发起请求时被携带并发送到服务器上。通常，它用于告知服务器两个请求是否来自同一浏览器。

如保持用户的登录状态，**Cookie 使基于无状态的 HTTP 协议记录稳定的状态信息成为了可能。**

Cookie 作用

- **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)

Session 概念

Session 代表着服务器和客户端一次会话的过程。Session 对象存储特定用户会话所需的属性及配置信息。这样，当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。**当客户端关闭会话，或者 Session 超时失效时会话结束。**

差别

- **作用范围不同**：Cookie 保存在客户端(浏览器)，Session 保存在服务端
- **存取方式的不同**：Cookie 只能保存 ASCII，Session 可以存任意数据类型
- **有效期不同**：Cookie 可设置为长时间保持，比如经常使用的默认登录功能，Session 一般失效时间较短，客户端关闭或者 Session 超时都会失效
- **隐私策略不同**：Cookie 存储在客户端，比较容易遭到不法获取；Session 存储在服务端，安全性相对 Cookie 要好一些
- **存储大小不同**：单个 Cookie 保存的数据不能超过 4K，Session 可存储数据远高于 Cookie

IPv4 和 IPv6 的区别？

- **IPv6 的首部长度是 40 个字节**，相对 IPv4 的首部长度 24 字节要长，但 IPv6 首部结构比 IPv4 简单
- **IPv6 把 IP 地址由 32 位增加到 128 位**，从而能够支持更大的地址空间。IPv6 简化了路由，加快了路由速度
- **IPv6 的可选项不放入报头，而是放在一个个独立的扩展头部**。如果不指定路由器不会打开处理扩展头部，IPv6 放宽了对可选项长度的严格要求 (IPv4 的可选项总长最多为 40 字节)，并可根据需要

随时引入新选项

- **IPv6 协议支持地址自动配置**，这是一种即插即用的机制。IPv6 节点通过地址自动配置得到 IPv6 地址和网关地址。IPv6 支持无状态地址自动配置和状态地址自动配置两种地址自动配置方式。它会给配置 128 位的地址带来很大的方便，特别是无状态地址自动配置
- 在 IPv6 中加入了关于身份验证、数据一致性和保密性的内容

什么是跨域，什么情况下会发生跨域请求？

概念

指的是**浏览器不能执行其他网站的脚本**。它是由浏览器的同源策略造成的。a 页面想获取 b 页面资源，如果 a、b 页面的协议、域名、端口、子域名不同，所进行的访问行动都是跨域的，而浏览器为了安全问题一般都限制了跨域访问，也就是不允许跨域请求资源

解决方法

- **Nginx**：使用 Nginx 作为代理服务器和用户交互，用户就只需要在 80 端口上进行交互就可以了，这样就避免了跨域问题
- **JSONP**：网页通过添加一个 script 元素，向服务器请求 JSON 数据，服务器收到请求后，将数据放在一个指定名字的回调函数的参数位置传回来。缺点是只支持 get 请求，不支持 post 请求
- **CORS**：跨域资源分享