http协议
重新认识http协议
1.http协议是什么

英文名：HyperText Transfer Protocol，中文名：超文本传输协议

超文本传输协议，它可以拆成三个部分，分别是：超文本，传输和协议

- 超文本
  - 所谓文本（Text），就表示 HTTP 传输的不是 TCP/UDP 这些底层协议里被切分的杂乱无章的二进制包（datagram），而是完整的、有意义的数据，可以被浏览器、服务器这样的上层应用程序处理。在互联网早期，“文本”只是简单的字符文字，但发展到现在，文本的涵义已经被大大地扩展了，图片、音频、视频、甚至是压缩包，在 HTTP 眼里都可以算做是“文本。所谓超文本，就是超越了普通文本的文本，它是文字、图片、音频和视频等的混合体，最关键的是含有“超链接”，能够从一个超文本跳跃到另一个超文本，形成复杂的非线性、网状的结构关系。对于超文本，我们最熟悉的就应该是 HTML 了，它本身只是纯文字文件，但内部用很多标签定义了对图片、音频、视频等的链接，再经过浏览器的解释，呈现在我们面前的就是一个含有多种视听信息的页面
- 传输
  - 数据虽然是在 A 和 B 之间传输，但并没有限制只有 A 和 B 这两个角色，允许中间有**中转**或者**接力**
- 协议
  - 双方必须共同遵从的一组约定，它应该包括语法，语义，同步规则和错误处理
  - http是一种网络传输协议
  - HTTP **协议是一个双向协议**(A<=>B)
  - HTTP **不是一个孤立的协议**
    - 在互联网世界里，HTTP 通常跑在 TCP/IP 协议栈之上，依靠 IP 协议实现寻址和路由、TCP 协议实现可靠数据传输、DNS 协议实现域名查找、SSL/TLS 协议实现安全通信。此外，还有一些协议依赖于 HTTP，

例如 WebSocket、HTTPDNS 等。这些协议相互交织，构成了一个协议网，而 HTTP 则处于中心地位

2.http协议与网络，浏览器，web服务器，CND之间的关系

网络

互联网的正式名称是 Internet，里面存储着无穷无尽的信息资源，我们通常所说的**上网**实际上访问的只是互联网的一个子集万维网（World Wide Web），它基于 HTTP 协议，传输 HTML 等超文本资源，能力也就被限制在 HTTP 协议之内。

浏览器

浏览器的正式名字叫Web Browser，顾名思义，就是检索、查看互联网上网页资源的应用程序，名字里的 Web，实际上指的就是World Wide Web，也就是万维网。浏览器本质上是一个 HTTP 协议中的请求方，使用 HTTP 协议获取网络上的各种资源。

在 HTTP 协议里，浏览器的角色被称为User Agent即用户代理，意思是作为访问者的“代理”来发起 HTTP 请求。

在网络爬虫中经常会使用User Agent，来冒充正常的请求防止被识别。

Web 服务器

http的响应方

当我们谈到Web 服务器时有两个层面的含义：硬件和软件。

硬件含义就是物理形式或云形式的机器，在大多数情况下它可能不是一台服务器，而是利用反向代理、负载均衡等技术组成的庞大集群。但从外界看来，它仍然表现为一台机器，但这个形象是虚拟的。

软件含义的 Web 服务器可能我们更为关心，它就是提供 Web 服务的应用程序，通常会运行在硬件含义的服务器上。它利用强大的硬件能力响应海量的客户端 HTTP 请求，处理磁盘上的网页、图片等静态文件，或者把请求转发给后面的 Tomcat、Node.js 等业务应用，返回动态的信息

CDN

CDN，全称是Content Delivery Network，翻译过来就是内容分发网络。它应用了 HTTP 协议里的缓存和代理技术，代替源站响应客户端的请求。
CDN 本质就是给域名设置一条 cname 记录指向CDN厂商的域名。

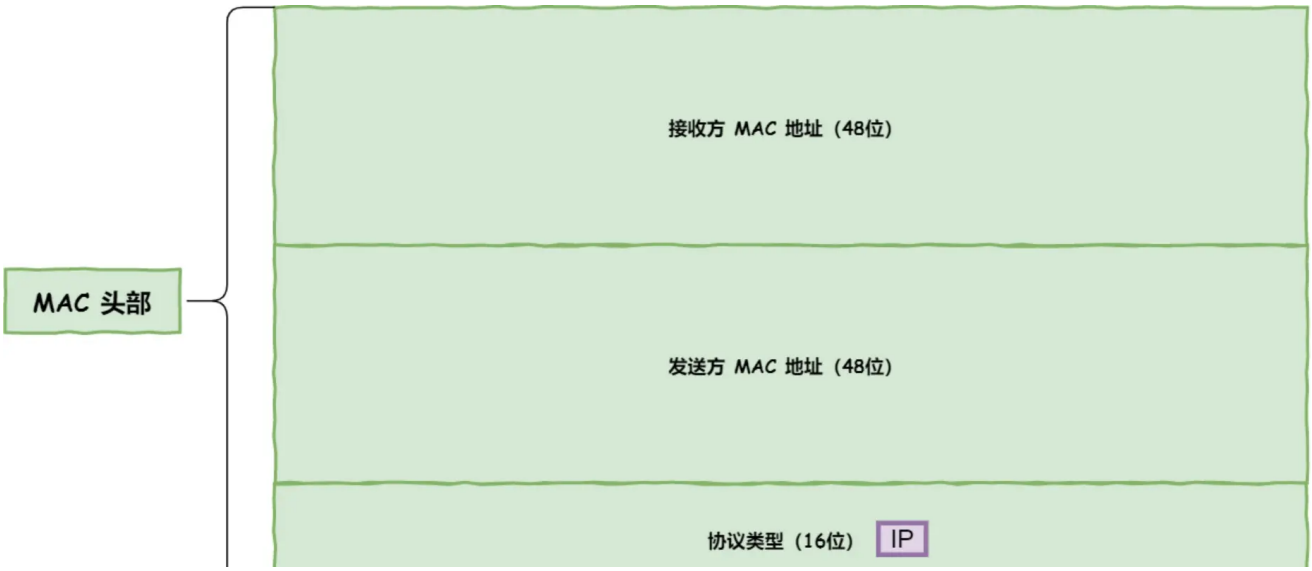
CDN 有什么好处呢?

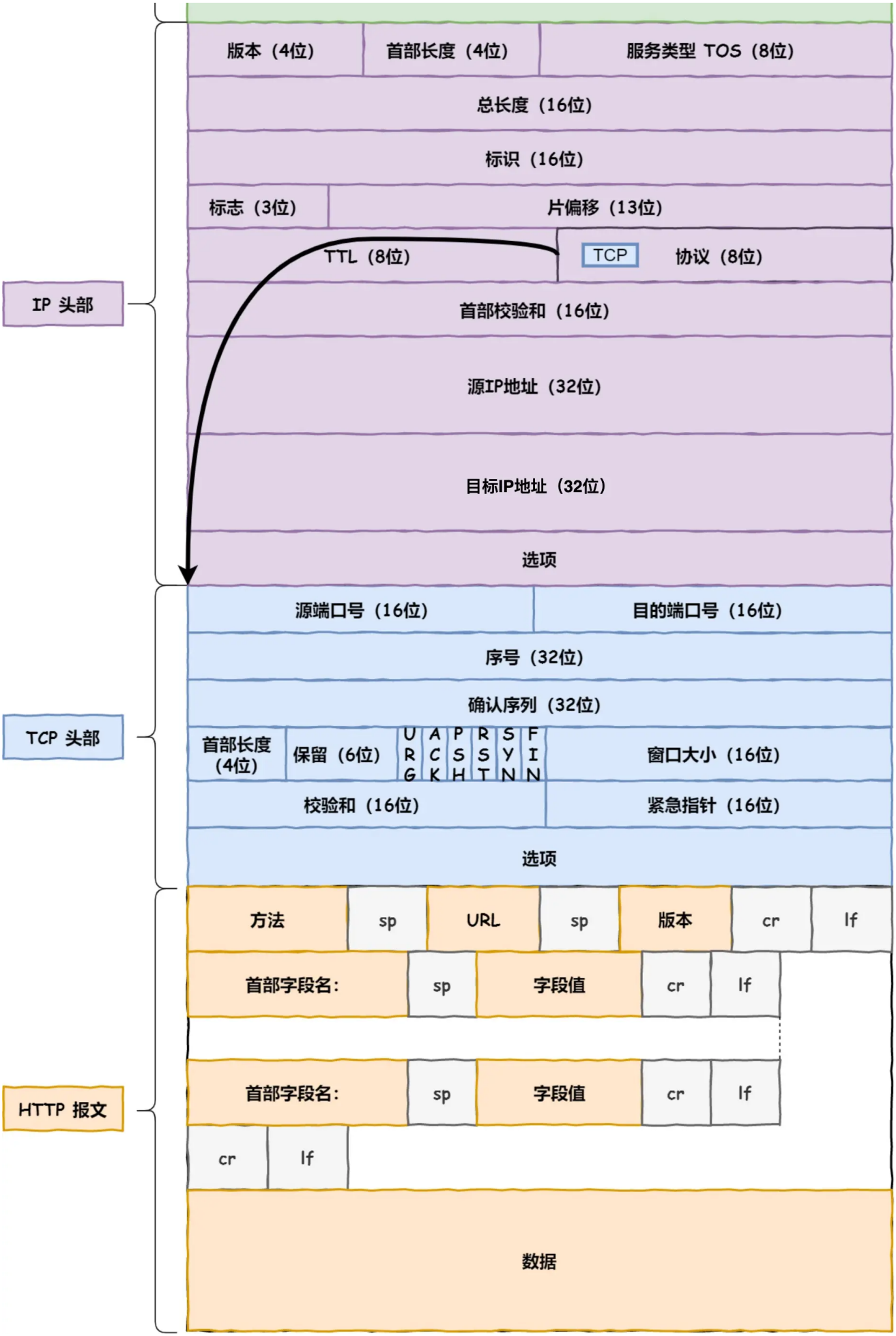
简单来说，它可以缓存源站的数据，让浏览器的请求不用“千里迢迢”地到达源站服务器，直接在“半路”就可以获取响应。如果 CDN 的调度算法很优秀，更可以找到离用户最近的节点，大幅度缩短响应时间。

CDN缓存后的网站的访问过程变为：

1. 用户向浏览器提供要访问的域名；
2. 浏览器调用域名解析库对域名进行解析，由于CDN对域名解析过程进行了调整，所以解析函数库一般得到的是该域名对应的CNAME（Canonical Name）记录，为了得到实际IP地址，浏览器需要再次对获得的CNAME域名进行解析以得到实际的IP地址；在此过程中，使用的全局负载均衡DNS解析，如根据地理位置信息解析对应的IP地址，使得用户能就近访问。
3. 此次解析得到CDN缓存服务器的IP地址，浏览器在得到实际的IP地址以后，向缓存服务器发出访问请求；
4. 缓存服务器根据浏览器提供的要访问的域名，通过Cache内部专用DNS解析得到此域名的实际IP地址，再由缓存服务器向此实际IP地址提交访问请求；
5. 缓存服务器从实际IP地址得得到内容以后，一方面在本地进行保存，以备以后使用，另一方面把获取的数据返回给客户端，完成数据服务过程；
6. 客户端得到由缓存服务器返回的数据以后显示出来并完成整个浏览的数据请求过程。

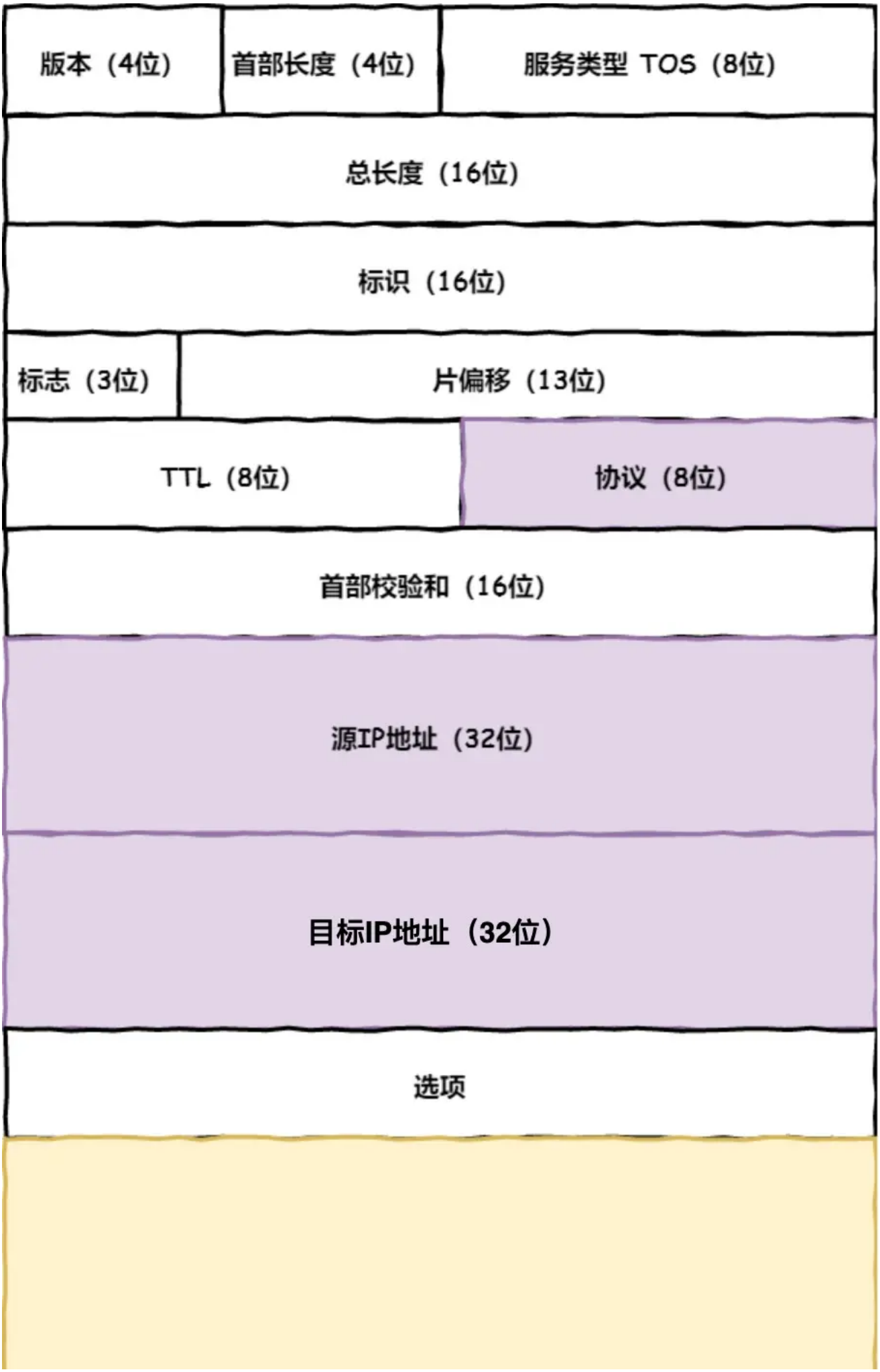
TCP/IP协议



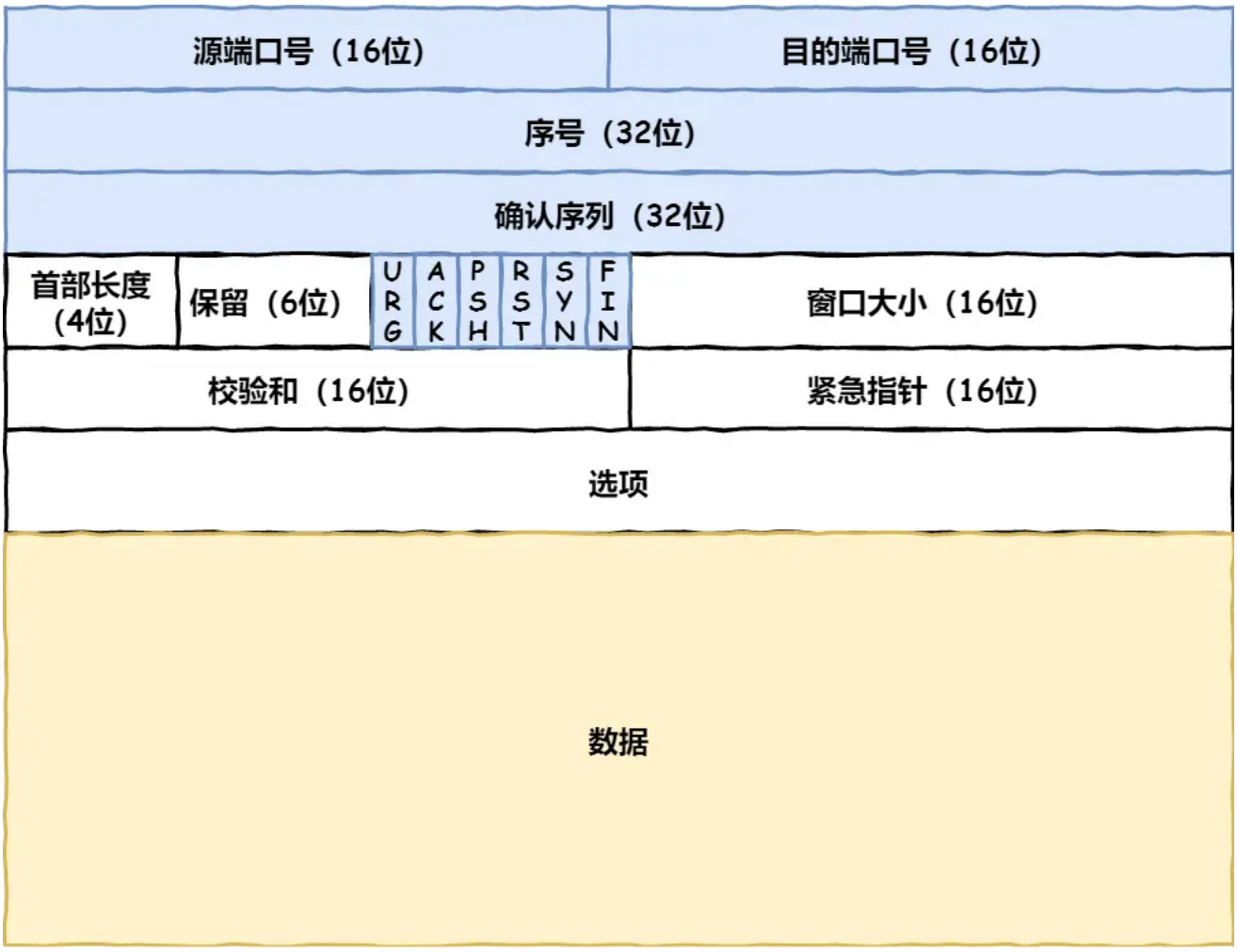


第一层叫“链接层”（link layer），负责在以太网、WiFi 这样的底层网络上发送原始数据包，工作在网卡这个层次，使用 MAC 地址来标记网络上的设备，所以

有时候也叫 MAC 层。

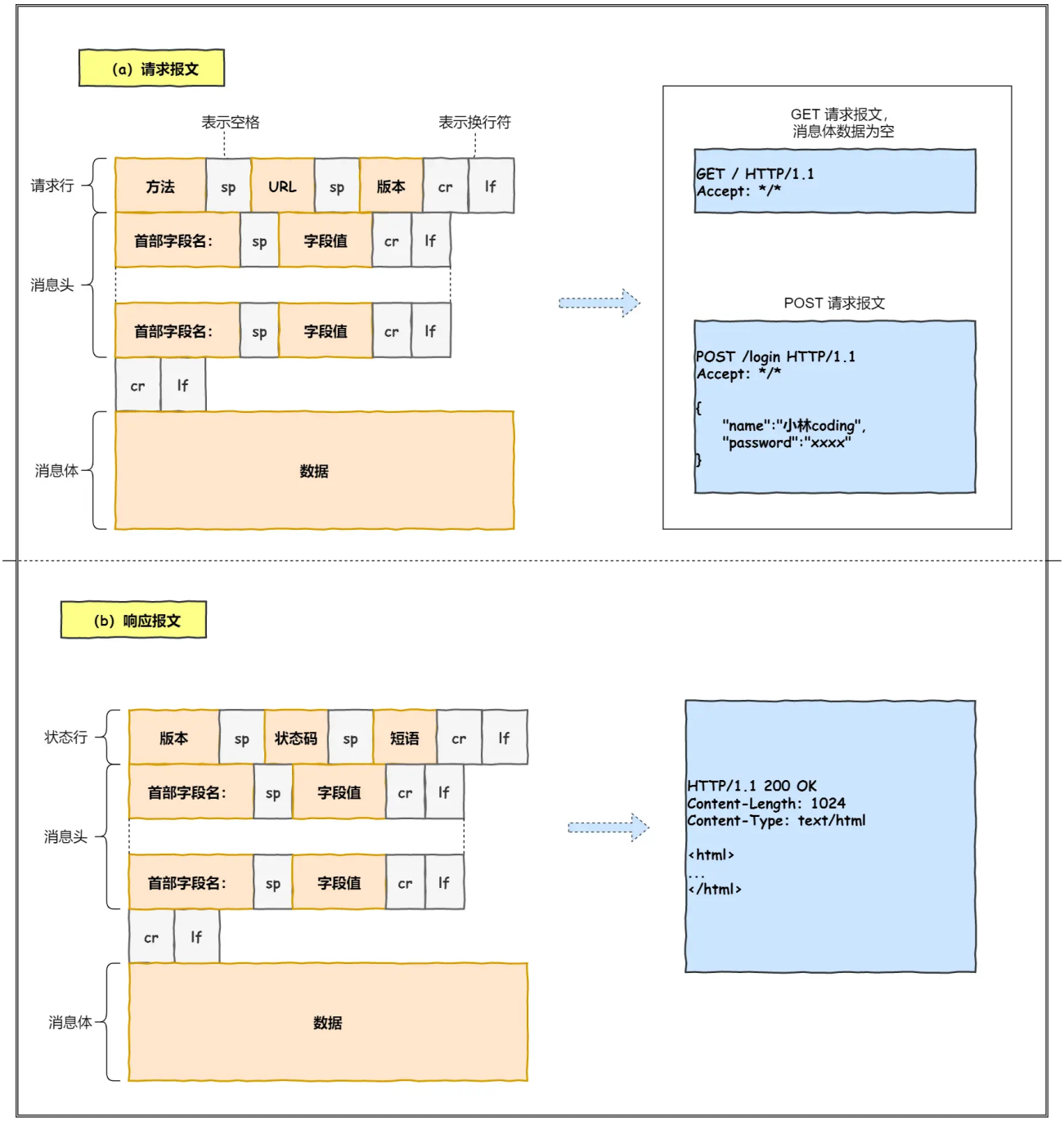

版本（4位）
首部长度（4位）
服务类型 TOS（8位）
总长度（16位）
标识（16位）
标志（3位）
片偏移（13位）
TTL（8位）
协议（8位）
首部校验和（16位）
源IP地址（32位）
目标IP地址（32位）
选项

第二层叫“网际层”或者“网络互连层”（internet layer），IP 协议就处在这一层。因为 IP 协议定义了“IP 地址”的概念，所以就可以在“链接层”的基础上，用 IP 地址取代 MAC 地址，把许许多多的局域网、广域网连接成一个虚拟的巨大网络，在这个网络里找设备时只要把 IP 地址再“翻译”成 MAC 地址就可以了。



第三层叫“传输层”（transport layer），这个层次协议的职责是保证数据在 IP 地址标记的两点之间“可靠”地传输，是 TCP 协议工作的层次，另外还有它的一个“小伙伴”UDP。TCP 是一个有状态的协议，需要先与对方建立连接然后才能发送数据，而且保证数据不丢失不重复。而 UDP 则比较简单，它无状态，不用事

先建立连接就可以任意发送数据，但不保证数据一定会发到对方。两个协议的另一个重要区别在于数据的形式。TCP 的数据是连续的“字节流”，有先后顺序，而 UDP 则是分散的小数据包，是顺序发，乱序收。



第四层叫“应用层”（application layer），由于下面的三层把基础打得非常好，所以在这一层就“百花齐放”了，有各种面向具体应用的协议。例如 Telnet、SSH、FTP、SMTP 等等，当然还有我们的 HTTP。

MAC 层的传输单位是帧（frame），IP 层的传输单位是包（packet），TCP 层的传输单位是段（segment），HTTP 的传输单位则是消息或报文（message）。但这些名词并没有什么本质的区分，可以统称为数据包。

http代理

可以参考：https://imququ.com/post/web-proxy.html

所谓的**代理服务**就是指服务本身不生产内容，而是处于中间位置转发上下游的请求和响应，具有双重身份：面向下游的用户时，表现为服务器，代表源服务器响应客户端的请求；而面向上游的源服务器时，又表现为客户端，代表客户端发送请求。
代理配置：

1. 手动配置，使用浏览器或者系统的代理
2. PAC（ProxyAutoConfiguration）代理自动配置
   1. https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Proxy_servers_and_tunneling/Proxy_Auto-Configuration_PAC_file#dnsdomainis
3. WPAD（WebProxy autodscovery Protocol）代理自动发现
   1. 一种客户端使用DHCP和/或DNS发现方法来定位一个配置文件URL的方法。在检测和下载配置文件后，它可以执行配置文件以测定特定URL应使用的代理

代理相关：

- TRACE
  - **HTTP** `TRACE` **方法** 实现沿通向目标资源的路径的消息环回（loop-back）测试，提供了一种实用的 debug 机制。
- Max-Forwards
  - `Max-Forwards` 请求标头被用于限制 `TRACE` 方法可经过的服务器（通常指代理服务器）数目。它的值是一个整数，指定可经过的服务器最大数目。服务器在转发 `TRACE` 请求之前，将递减 `Max-Forwards` 的值，直到到达目标服务器，或服务器接收到 `Max-Forwards` 的值为 0 的请求。而后直接返回一个 `200 OK` 的响应（可以包含一些标头）。
- `Via` 是一个通用首部，是由代理服务器添加的，适用于正向和反向代理，在请求和响应首部中均可出现。这个消息首部可以用来追踪消息转发情况，防止循环请求，以及识别在请求或响应传递链中消息发送者对于协议的支持能力。
  - https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Via
- `X-Forwarded-For` (XFF) 在客户端访问服务器的过程中如果需要经过 HTTP 代理或者负载均衡服务器，可以被用来获取最初发起请求的客户端的 IP 地址，这个消息首部成为事实上的标准。在消息流从客户端流向服务器的过程

中被拦截的情况下，服务器端的访问日志只能记录代理服务器或者负载均衡服务器的 IP 地址。如果想要获得最初发起请求的客户端的 IP 地址的话，那么 X-Forwarded-For 就派上了用场。
    - https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/X-Forwarded-For

代理服务协议：

- FTP代理服务器
    - 主要用于访问FTP服务器，一般有上传、下载功能。端口一般为20、21、2121等。
- HTTP代理服务器
    - 主要用于访问网页，一般有内容过滤和缓存功能。端口一般为80、8080、3128等。
- SSL/TLS代理（HTTPS代理）
    - 主要用于访问HTTPS网站。端口一般为443。
- RTSP代理
    - 主要用于Real Player访问Real流媒体服务器，一般有缓存功能。端口一般为554。
- Telnet代理
    - 主要用于Telnet远程控制（黑客入侵计算机时常用于隐藏身份）。端口一般为23。
- POP3/SMTP代理
    - 主要用于POP3/SMTP方式收发邮件，一般有缓存功能。端口一般为110、25。
- SOCKS代理
    - 只是单纯传递数据包，不关心具体协议和用法，所以速度快很多。端口一般为1080。

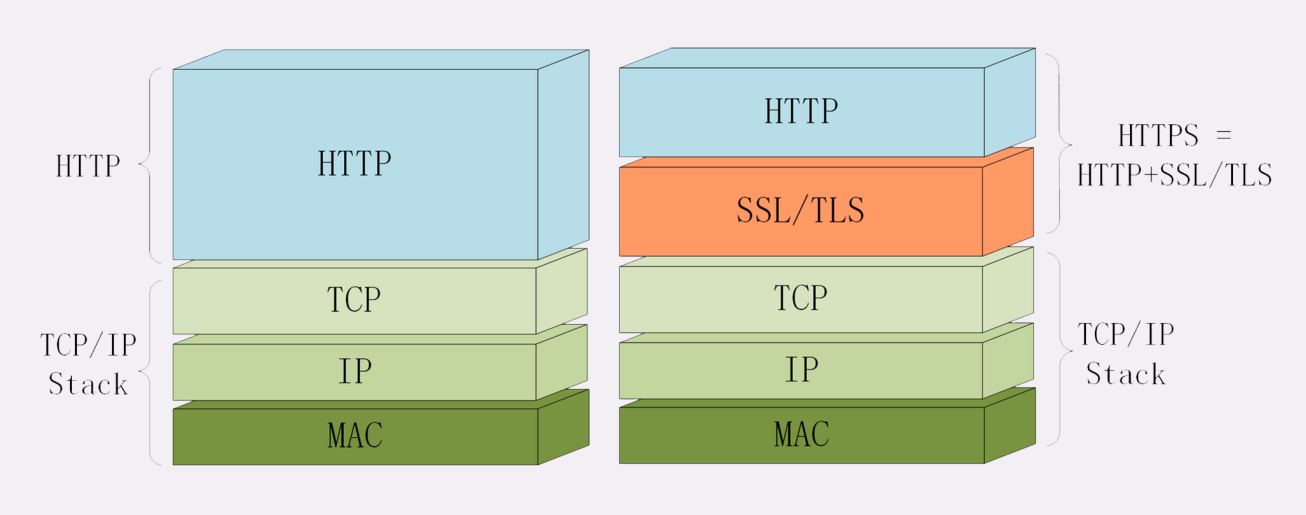
代理和网关的区别：

- 代理连接的是两个或多个使用相同协议的应用程序。
- 网关连接的是两个或多个使用不同协议的端点，扮演“协议转换器”的角色

HTTPS

HTTP 的一些缺点

- 无状态
- 明文和不安全（由于 HTTP 天生“明文”的特点，整个传输过程完全透明，任何人都能够在链路中截获、修改或者伪造请求 / 响应报文，数据不具有可信性）



TLS（Transport Layer Security，传输层安全性协议）以及其前身SSL（Secure Sockets Layer，安全套接层）是一种安全协议

HTTPS：即HTTP over SSL/TLS，用来解决HTTP明文传输导致的不安全问题。流程大致为：
使用对称加密算法加解密报文，保证机密性；使用摘要算法保证数据完整性；使用证书CA来进行身份认证;而不可否认则由非对称加密算法来实现。由于非对称加密算法耗时比对称加密算法长，所以用非对称加密算法来加解密给报文加密的对称算法的秘钥：即使用公钥对对称加密算法秘钥进行加密，私钥用来相应地解密。

对称和非对称加密

非对称加密基于大数运算，比如大素数或者椭圆曲线，是复杂的数学难题，所以消耗计算量，运算速度慢。（DH、DSA、RSA、ECC 等）

除了慢，可能还有一个缺点就是需要更多的位数，相同强度的对称密钥要比非对称密钥短。

对称密钥一般都128位、256位，而rsa一般要2048位，不过椭圆曲线的会短一点

混合加密

用非对称加密，加密对称加密的私钥。对称加密的私钥又是会话级的随机数=一次会话一个私钥。就算别人破解也只是破解了一个会话。

分组模式：DES和AES都属于分组密码，它们只能加密固定长度的明文。如果需要加密任意长度的明文，就需要对分组密码进行迭代，而分组密码的迭代方法就称为分组密码的“模式”。
主要模式：
ECB模式：Electronic Code Book mode（电子密码本模式）
CBC模式：Cipher Block Chaining mode（密码分组链接模式）（推荐使用）
CFB模式：Cipher FeedBack mode（密文反馈模式）
OFB模式：Output FeedBack mode（输出反馈模式）
CTR模式：CounTeR mode（计数器模式）（推荐使用）

总结

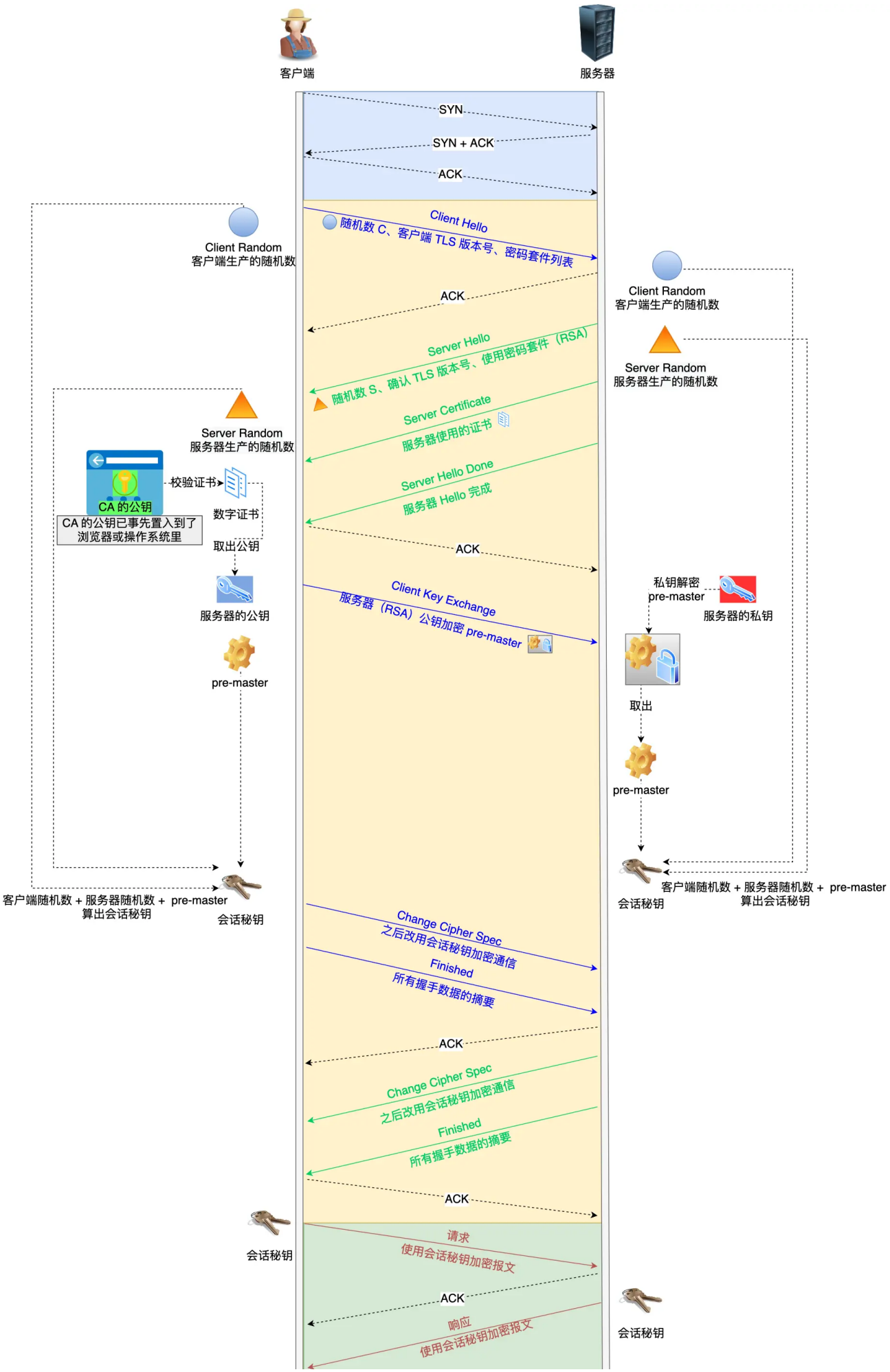
保密性：靠混合加密解决，非对称加密实现对称加密秘钥传递，对称加密实现内容加密。
完整性：靠摘要算法解决。
身份认证：靠数字证书解决，数字证书因为CA机构的信任变成一个完整信任链条，从而实现通过数字证书证明了对方真实身份，但注意身份真实也可能是挂羊头卖狗肉，是一个坏人，所以，有了CRL、OCSP，还有终止信任。
不可否认：靠数字签名解决，内容摘要算法得到摘要，私钥加密摘要，对方使用对应公钥解密，得到摘要，再和自己得到的服务器提供的原文摘要对比，一致说明这个内容就是原服务器提供的，由证书说明了服务器的身份。

关于证书验证：
服务器返回的是证书链（不包括根证书，根证书预置在浏览器中），然后浏览器就可以使用信任的根证书（根公钥）解析证书链的根证书得到一级证书的公钥+摘要验签，然后拿一级证书的公钥解密一级证书拿到二级证书的公钥和摘要验签，再然后拿二级证书的公钥解密二级证书得到服务器的公钥和摘要验签，验证过程就结束了。

单向认证和双向认证

“单向认证”握手过程，只认证了服务器的身份，而没有认证客户端的身份。这是因为通常单向认证通过后已经建立了安全通信，用账号、密码等简单的手段就能够确认用户的真实身份。但为了防止账号、密码被盗，有的时候（比如网上银行）还会使用 U 盾给用户颁发客户端证书，实现“双向认证”，这样会更加安全。

tsl


客户端
服务器
SYN
SYN + ACK
ACK
Client Hello
随机数 C、客户端 TLS 版本号、密码套件列表
ACK
Server Hello
随机数 S、确认 TLS 版本号、使用密码套件（RSA）
Server Certificate
服务器使用的证书
Server Hello Done
服务器 Hello 完成
ACK
Client Key Exchange
服务器（RSA）公钥加密 pre-master
Change Cipher Spec
之后改用会话秘钥加密通信
Finished
所有握手数据的摘要
ACK
Change Cipher Spec
之后改用会话秘钥加密通信
Finished
所有握手数据的摘要
ACK
请求
使用会话秘钥加密报文
ACK
响应
使用会话秘钥加密报文
Client Random
客户端生产的随机数
Server Random
服务器生产的随机数
CA 的公钥
CA 的公钥已事先置入到了
浏览器或操作系统里
校验证书
数字证书
取出公钥
服务器的公钥
pre-master
客户端随机数 + 服务器随机数 + pre-master
算出会话秘钥
会话秘钥
Client Random
客户端生产的随机数
Server Random
服务器生产的随机数
私钥解密
pre-master
服务器的私钥
取出
pre-master
会话秘钥
客户端随机数 + 服务器随机数 + pre-master
算出会话秘钥
会话秘钥
会话秘钥

总结下TLS的握手过程

第一阶段：C/S两端共享Client Random、Server Random 和 Server Params信息

客户端--->服务器：

- 客户端首先会发一个「Client Hello」消息
  - 消息包含客户端的版本号、支持的密码套件，还有一个随机数（Client Random）

服务端--->客户端：
当服务端收到客户端的「Client Hello」消息后，会确认 TLS 版本号是否支持，和从密码套件列表中选择一个密码套件，以及生成随机数（Server Random）。
比如：
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384、随机数随机数（Server Random）

这个随机数有啥用呢？
其实这两个随机数是后续作为生成「会话密钥」的条件，所谓的会话密钥就是数据传输时，所使用的对称加密密钥。

此外，服务端为了证明自己的身份，会发送「Server Certificate」给客户端，这个消息里含有数字证书。
最后，服务端发了「Server Hello Done」消息，目的是告诉客户端，我已经把该给你的东西都给你了。

第二阶段：证书验证

一个数字证书通常包含了：

- 公钥；
- 持有者信息；
- 证书认证机构（CA）的信息；
- CA 对这份文件的数字签名及使用的算法；
- 证书有效期；

- 还有一些其他额外信息；

客户端证书链逐级验证、证书公钥验证签名，服务端身份验证成功（证书合法）

客户端--->服务端
接着，客户端就会生成一个新的随机数 (pre-master)，用服务器的 RSA 公钥加密该随机数，通过「Client Key Exchange」消息传给服务端。

第三阶段：主密钥生成

TLS 1.2 中的密钥：
主密钥是由预备主密钥、ClientHello random 和 ServerHello random 通过 PRF 函数生成的。**主密钥长度是 48 字节**。可以看出，只要我们知道预备主密钥或者主密钥便可以解密抓包数据，所以 TLS 1.2 中抓包解密调试只需要一个主密钥即可，SSLKEYLOG 就是将主密钥导出来，在 Wireshark 里面导入就可以解密相应的抓包数据。

会话密钥(密钥块)是由主密钥、SecurityParameters.server_random 和 SecurityParameters.client_random 数通过 PRF 函数来生成，会话密钥里面包含对称加密密钥、消息认证和 CBC 模式的初始化向量，对于非 CBC 模式的加密算法来说，就没有用到这个初始化向量。

```python
master_secret = PRF(pre_master_secret, "master secret",
  ClientHello.random + ServerHello.random)
```

Session ID 缓存和 Session Ticket 里面保存的也是主密钥，而不是会话密钥，这样每次会话复用的时候再用双方的随机数和主密钥导出会话密钥，从而实现每次加密通信的会话密钥不一样，即使一个会话的主密钥泄露了或者被破解了也不会影响到另一个会话。

服务端收到后，用 RSA 私钥解密，得到客户端发来的随机数 (pre-master)。

最后，客户端、服务端分别使用Client Params、Server Params通过ECDHE算法计算出随机值pre-master，然后用Client Random、Server Random 和 Pre-Master 三个值作为原材料，用PRF伪随机数函数（利用密码套件的摘要算法再次强化结果值maser secert的随机性）计算出主密钥Master Secret。

注意：
主密钥并不是会话秘钥，还会再用PRF扩展出更多的密钥，比如客户端发送用

的会话密钥（client_write_key）、服务器发送用的会话密钥（server_write_key），避免只用一个密钥带来的安全隐患

至此，**客户端和服务端双方都共享了三个随机数，分别是** Client Random、Server Random、pre-master。

客户端--->服务端:
客户端发一个“Change Cipher Spec”，然后再发一个“Finished”消息，把之前所有发送的数据做个摘要，再加密一下，让服务器做个验证，验证加密通信「是否可用」和「之前握手信息是否有被中途篡改过」。

服务端--->客户端：
服务器也是同样的操作，发“Change Cipher Spec”和“Finished”消息，双方都验证加密解密 OK，握手正式结束.

为什么 RSA 密钥交换不具有“前向安全”（指的是长期使用的主密钥泄漏不会导致过去的会话密钥泄漏）？

答：RSA 握手中，Server Hello 后，客户端拿到服务器的证书，从中提取出服务器的公钥，然后用这个公钥去加密客户端生成的一个随机数（会话密钥）得到密文，然后将其返回给服务器。虽然每次 TLS 握手中的会话密钥都是不一样的，但服务器的私钥却始终不会变。一旦黑客拿到了服务器私钥，并且截获了之前的所有密文，就能拿到每次会话中的对称密钥，从而得到客户端和服务器的所有“历史会话记录”。 说到底，RSA 握手下，服务器私钥是不变的，从而导致不具有“前向安全”。而 ECDHE 的私钥却是动态的，黑客拿到了一个，也只能解密一个密文。

tsl1.3

TLS1.3 的三个主要改进目标：兼容、安全与性能。

兼容

在记录头的 Version 字段被兼容性“固定”的情况下，只要是 TLS1.3 协议，握手的“Hello”消息后面就必须有“supported_versions”扩展，它标记了 TLS 的版本号，使用它就能区分新旧协议。

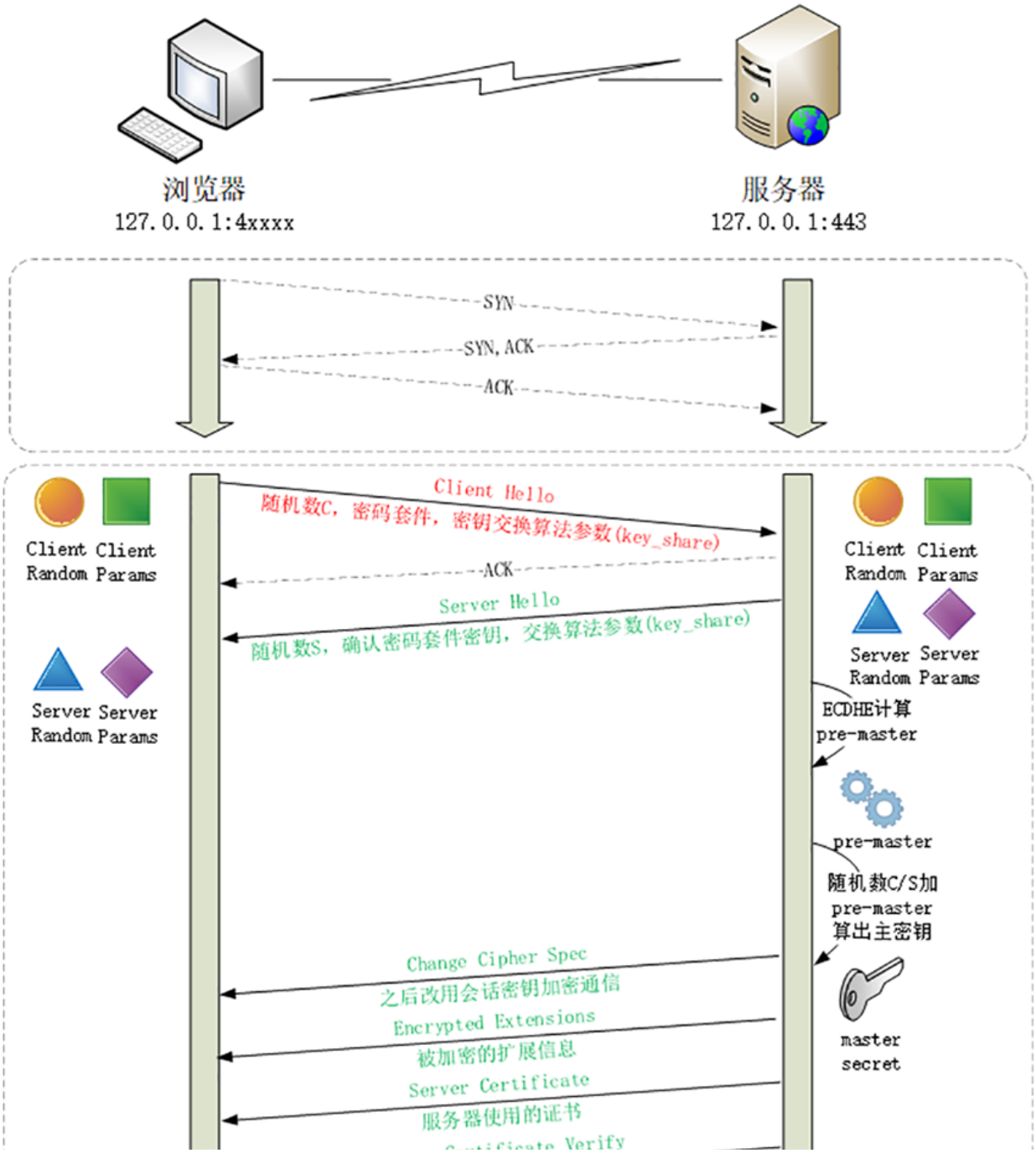
强化安全

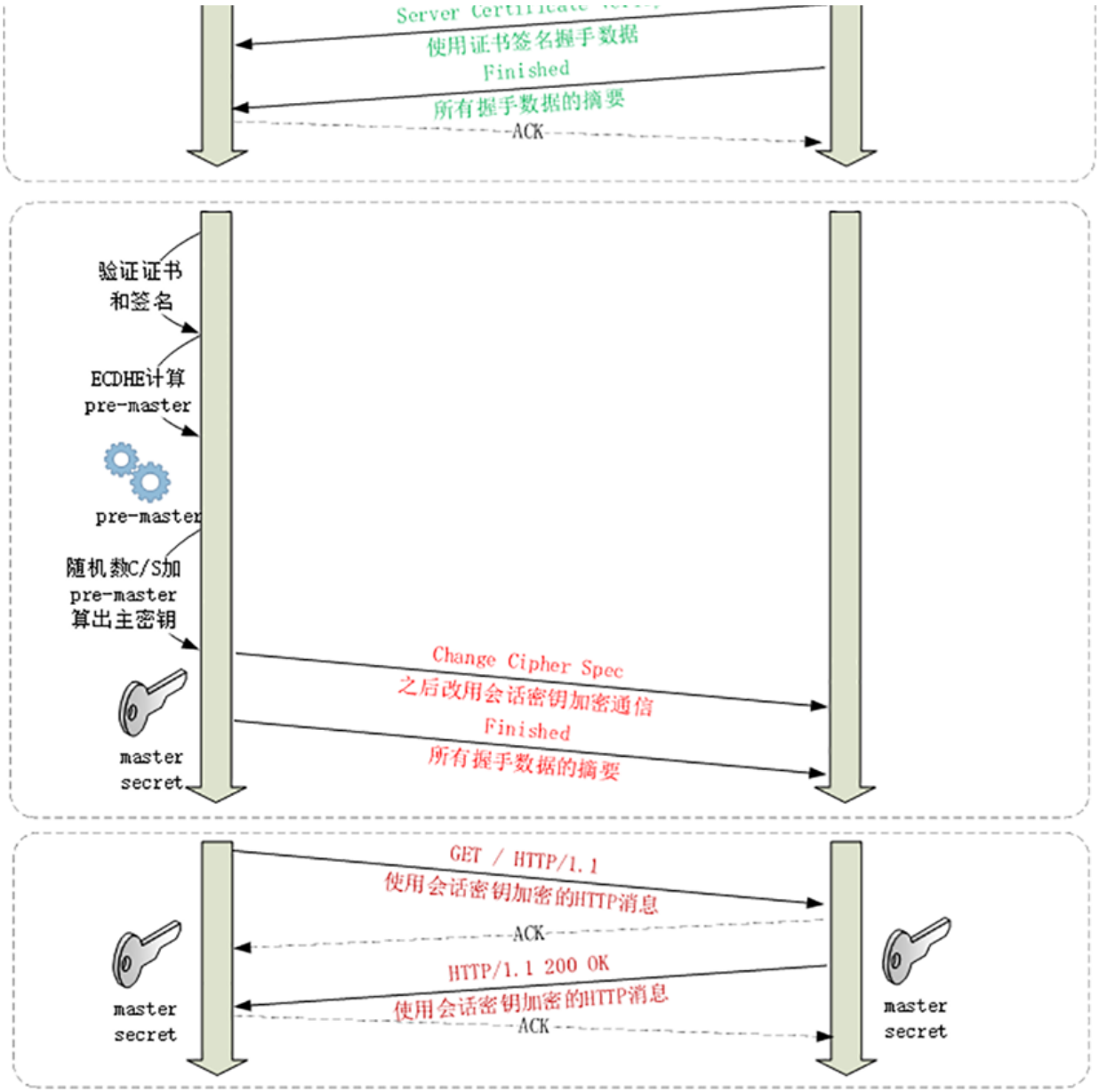
TLS1.3 就在协议里修补了这些不安全因素。比如：
伪随机数函数由 PRF 升级为 HKDF（HMAC-based Extract-and-Expand Key Derivation Function）；

明确禁止在记录协议里使用压缩；
废除了 RC4、DES 对称加密算法；
废除了 ECB、CBC 等传统分组模式；
废除了 MD5、SHA1、SHA-224 摘要算法；
废除了 RSA、DH 密钥交换算法和许多命名曲线。

提升性能

HTTPS 建立连接时除了要做 TCP 握手，还要做 TLS 握手，在 1.2 中会多花两个消息往返（2-RTT），可能导致几十毫秒甚至上百毫秒的延迟，在移动网络中延迟还会更严重。现在因为密码套件大幅度简化，也就没有必要再像以前那样走复杂的协商流程了。TLS1.3 压缩了以前的“Hello”协商过程，删除了“Key Exchange”消息，把握手时间减少到了“1-RTT”，效率提高了一倍。





tsl1.3中的密钥

在 TLS 1.3 中，不再使用 PRF 这种算法了，而是采用更标准的 HKDF 算法来进行密钥的推导。而且在 TLS 1.3 中对密钥进行了更细粒度的优化，每个阶段或者方向的加密都不是使用同一个密钥。TLS 1.3 在 ServerHello 消息之后的数据都是加密的，握手期间 Server 给 Client 发送的消息用 server_handshake_traffic_secret 通过 HKDF 算法导出的密钥加密的，Client 发送给 Server 的握手消息是用 client_handshake_traffic_secret 通过 HKDF 算法导出的密钥加密的。这两个密钥是通过 Handshake Secret 密钥来导出的，而 Handshake Secret 密钥又是由 PreMasterSecret 和 Early Secret 密钥导出，然后通过 Handshake Secret 密钥导出主密钥 Master Secret。

域名和DNS

url格式的基本组成

- 第一部分-协议（scheme）
  - 该URL的协议是什么
- 第二部分-域名（domain）
  - 是指存放资源的服务器的域名系统(DNS) 主机名或 IP 地址
  - 域名分为顶级域名，二级域名，三级域名
- 第三部分-端口（port）
  - 用来区分同一台服务器上不同服务的标识
- 第三部分-路劲（path）
  - 由零或多个“/”符号隔开的字符串，一般用来表示主机上的一个目录或文件地址。
- 第四部分-参数（parameters）
  - 这是用于指定特殊参数的可选项
- `#`锚点表示资源中的一种“书签”，给浏览器显示位于该“加书签”位置的内容的方向

可以参考：https://developer.mozilla.org/zh-CN/docs/Learn/Common_questions/What_is_a_URL

DNS

DNS使用UDP端口53。当前，对于每一级域名长度的限制是63个字符，域名总长度则不能超过253个字符。

DNS 的核心系统是一个**三层**的树状、分布式服务，基本对应域名的结构：

1. 根域名服务器（Root DNS Server）：管理顶级域名服务器，返回“com”“net”“cn”等顶级域名服务器的 IP 地址；
2. 顶级域名服务器（Top-level DNS Server）：管理各自域名下的权威域名服务器，比如 com 顶级域名服务器可以返回 apple.com 域名服务器的 IP 地址；
3. 权威域名服务器（Authoritative DNS Server）：管理自己域名下主机的 IP 地址，比如 apple.com 权威域名服务器可以返回 www.apple.com 的 IP 地址

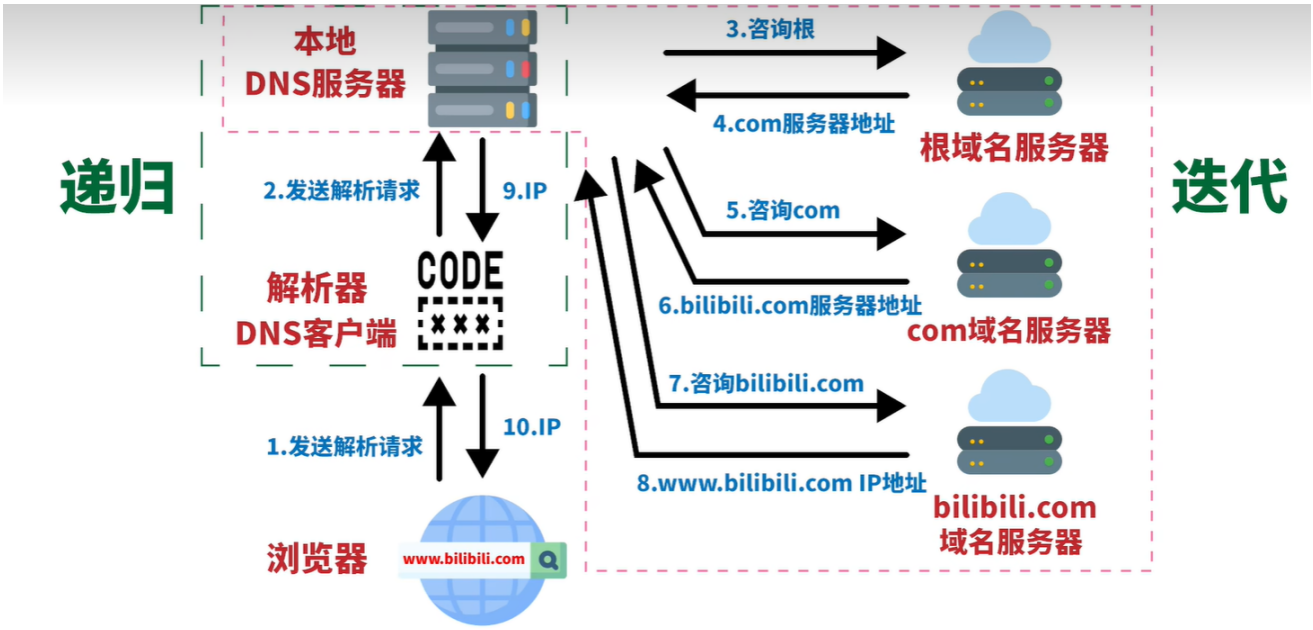
例如，你要访问“www.apple.com”，就要进行下面的三次查询：

1. 访问根域名服务器，它会告诉你“com”顶级域名服务器的地址；
2. 访问“com”顶级域名服务器，它再告诉你“apple.com”域名服务器的地址；

3. 最后访问“apple.com”域名服务器，就得到了“www.apple.com”的地址。

q：在浏览器地址栏里随便输入一个不存在的域名，比如就叫“www.不存在.com”，试着解释一下它的 DNS 解析过程

浏览器DNS缓存->操作系统缓存->Hosts文件->非权威域名服务器->根域名服务器->顶级域名服务器->权威域名服务器



操作：

```
> nslookup
> set type=ns // 设置类型为nameserver
> . //表示根域名服务器
> Non-authoritative answer:... // 缓存的结果
> set type=a // 设置类型为IPV4
> 选择一个ip地址(root)  nameserver = e.root-servers.net
> e.root-servers.net
> server xx.xx.xx.xx // 设置服务地址
> set type=a
> com. // 查询顶级域名
> 以此类推
```

CName（Canonical Name）

CNAME（Canonical Name）记录，指域名解析中的别名记录，用来把一个域名解析到另一个域名（CNAME域名），再由CNAME域名来解析到需要访问的服务

器IP地址。
例如，假设您有几个子域，例如www.mydomain.com，ftp.mydomain.com，mail.mydomain.com等，并且您希望这些子域指向您的主域名mydomain.com。您可以创建CNAME记录，而不是为每个子域创建A记录并将其绑定到您域的IP地址。
如果服务器的IP地址发生更改，则只需更新一个A记录，并且所有子域都会自动更新，因为所有CNAMES都指向带有A记录的主域

A记录

A (Address) 记录是用来指定主机名（或域名）对应的IP地址记录。用户可以将该域名下的网站服务器指向到自己的网页服务器(web server)上。同时也可以设置域名的子域名。

http报文

客户端

请求行

请求行由三部分构成：

1. 请求方法：是一个动词，如 GET/POST，表示对资源的操作；
2. 请求目标：通常是一个 URI，标记了请求方法要操作的资源；
3. 版本号：表示报文使用的 HTTP 协议版本。

请求头

空行

请求消息体

example：

```
GET / HTTP/1.1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,
image/avif,image/webp,image/apng,*/*;q=0.8,application
/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
```

```
Accept-Language: en
Cache-Control: max-age=0
Connection: keep-alive
Cookie: BIDUPSID=2A2820756FE97D56B95AD1670EB22953;
PSTM=1672130396;
BAIDUID=2A2820756FE97D561458D9B54EDC50FB:FG=1;
BD_HOME=1; BD_UPN=12314753;
BA_HECTOR=8l8g2hag0k2k240h200la0141hqlbqt1j;
BAIDUID_BFESS=2A2820756FE97D561458D9B54EDC50FB:FG=1;
ZFY=fCjGZZBna6gCKlQZqHLp4H4Txcr:BweizftS3S8kueo8:C;
H_PS_PSSID=36548_37647_37906_36920_37990_37794_37930_3
7952_37903_26350_37787_37881
DNT: 1
Host: www.baidu.com
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.0.0 Safari/537.36
sec-ch-ua: "Not?A_Brand";v="8", "Chromium";v="108",
"Google Chrome";v="108"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
```

服务端

状态行

意思是服务器响应的状态。比起请求行来说，状态行要简单一些，同样也是由三部分构成：

1. 版本号：表示报文使用的 HTTP 协议版本；
2. 状态码：一个三位数，用代码的形式表示处理的结果，比如 200 是成功，500 是服务器错误；

3. 原因短语：作为数字状态码补充，是更详细的解释文字，帮助人理解原因。

响应头
空行
响应消息体

example：

```
HTTP/1.1 200 OK
Bdpagetype: 1
Bdqid: 0x93efafd800036d11
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Tue, 27 Dec 2022 08:39:21 GMT
Server: BWS/1.1
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: BD_HOME=1; path=/
Set-Cookie:
H_PS_PSSID=36558_37647_37687_37907_37625_36920_36806_3
7948_37930_37993_37902_26350_37786_37881; path=/;
domain=.baidu.com
Strict-Transport-Security: max-age=172800
Traceid: 1672130361048462593010659932185257274641
X-Frame-Options: sameorigin
X-Ua-Compatible: IE=Edge,chrome=1
Transfer-Encoding: chunked
```

关于头部字段

请求头和响应头的结构是基本一样的。头部字段是 key-value 的形式，key 和 value 之间用“:”分隔，最后用 CRLF 换行表示字段结束。

使用头字段需要注意下面几点：

1. 字段名不区分大小写，例如“Host”也可以写成“host”，但首字母大写的可读性更好；

2. 字段名里不允许出现空格，可以使用连字符`-`，但不能使用下划线`_`。例如，test-name是合法的字段名，而test name，test_name是不正确的字段名；
3. 字段名后面必须紧接着`:`，不能有空格。而`:`后的字段值前可以有多个空格；
4. 字段的顺序是没有意义的，可以任意排列不影响语义；
5. 字段原则上不能重复，除非这个字段本身的语义允许，例如 Set-Cookie。

HTTP 协议规定了非常多的头部字段，实现各种各样的功能，但基本上可以分为四大类：

1. 通用字段：在请求头和响应头里都可以出现；
2. 请求字段：仅能出现在请求头里，进一步说明请求信息或者额外的附加条件；
3. 响应字段：仅能出现在响应头里，补充说明响应报文的信息；
4. 实体字段：它实际上属于通用字段，但专门描述 body 的额外信息。

Host：请求字段，只能出现在请求头。是必须出现的字段
User-Agent：是请求字段，只能出现在请求头里。
Date：是通用字段，通常出现在响应头，标识HTTP报文创建的时间，客户端可以使用这个时间再搭配其他字段决定缓存策略
Server字段是响应字段，只能出现在响应头里。告诉客户端当前正在提供Web服务的软件名称和版本号。
Content-Length：标识报文里body的长度。

http请求方法

目前 HTTP/1.1 规定了八种方法，单词都必须是大写的形式。

1. GET：获取资源，可以理解为读取或者下载数据；
2. HEAD：获取资源的元信息；HEAD 方法与 GET 方法类似，也是请求从服务器获取资源，服务器的处理机制也是一样的，但服务器不会返回请求的实体数据，只会传回响应头，也就是资源的“元信息”。
3. POST：向资源提交数据，相当于写入或上传数据；
4. PUT：PUT 的作用与 POST 类似，也可以向服务器提交数据，但与 POST 存在微妙的不同，通常 POST 表示的是新建create的含义，而 PUT 则是修改update的含义。

5. DELETE：删除资源；
6. CONNECT：建立特殊的连接隧道；
7. OPTIONS：列出可对资源实行的方法；
8. TRACE：追踪请求 - 响应的传输路径。

实际上最常用的事GET和POST请求，那为什么不使用其他的方法，答案在下面。

安全与幂等。

在 HTTP 协议里，所谓的“安全”是指请求方法不会“破坏”服务器上的资源，即不会对服务器上的资源造成实质的修改。按照这个定义，只有 GET 和 HEAD 方法是“安全”的，因为它们是“只读”操作，只要服务器不故意曲解请求方法的处理方式，无论 GET 和 HEAD 操作多少次，服务器上的数据都是“安全的”。而 POST/PUT/DELETE 操作会修改服务器上的资源，增加或删除数据，所以是“不安全”的。所谓的“幂等”实际上是一个数学用语，被借用到了 HTTP 协议里，意思是多次执行相同的操作，结果也都是相同的，即多次“幂”后结果“相等”。很显然，GET 和 HEAD 既是安全的也是幂等的，DELETE 可以多次删除同一个资源，效果都是“资源不存在”，所以也是幂等的。POST 和 PUT 的幂等性质就略费解一点。按照 RFC 里的语义，POST 是“新增或提交数据”，多次提交数据会创建多个资源，所以不是幂等的；而 PUT 是“替换或更新数据”，多次更新一个资源，资源还是会第一次更新的状态，所以是幂等的。

你可以对比一下 SQL 来加深理解：把 POST 理解成 INSERT，把 PUT 理解成 UPDATE，这样就很清楚了。多次 INSERT 会添加多条记录，而多次 UPDATE 只操作一条记录，而且效果相同。

关于幂等可以参考：https://developer.mozilla.org/zh-CN/docs/Glossary/Idempotent

数据类型和编码

关于MIME

MIME，“多用途互联网邮件扩展”（Multipurpose Internet Mail Extensions），它是一种多用途网际邮件扩充协议，在1992年最早应用于电子邮件系统，但后来也应用到浏览器。服务器会将它们发送的多媒体数据的类型告诉浏览器，而通知手段就是说明该多媒体数据的MIME类型，从而让浏览器知道接收到的信息，客户端需要在请求头里使用 Accept 等头字段与服务器进行“内容协商”，要求服务器返回最合适的数据；

HTTP 里经常遇到的几个类别：

1. text：即文本格式的可读数据，我们最熟悉的应该就是 text/html 了，表示超文本文档，此外还有纯文本 text/plain、样式表 text/css 等。
2. image：即图像文件，有 image/gif、image/jpeg、image/png 等。a
3. audio/video：音频和视频数据，例如 audio/mpeg、video/mp4 等。
4. application：数据格式不固定，可能是文本也可能是二进制，必须由上层应用程序来解释。常见的有 application/json，application/javascript、application/pdf 等，另外，如果实在是不知道数据是什么类型，像刚才说的“黑盒”，就会是 application/octet-stream，即不透明的二进制数据。

比如

客户端：

```
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

服务端：

```
content-encoding: gzip
content-type: text/html; charset=utf-8
vary: Accept-Encoding
```

POST 提交数据的几种方式：

application/x-www-form-urlencoded	**在发送前编码所有字符**
multipart/form-data	不对字符编码。在使用包含文件上传控件的表单时，必须使用该值。
text/plain	空格转换为 "+" 加号，但不对特殊字符编码。

但仅有 MIME type 还不够，因为 HTTP 在传输时为了节约带宽，有时候还会压缩数据，为了不要让浏览器继续“猜”，还需要有一个“Encoding type”，告诉数

据是用的什么编码格式，这样对方才能正确解压缩，还原出原始的数据。比起 MIME type 来说，Encoding type 就少了很多，

常用的只有下面三种：

1. gzip：GNU zip 压缩格式，也是互联网上最流行的压缩格式；
2. deflate：zlib（deflate）压缩格式，流行程度仅次于 gzip；
3. br：一种专门为 HTTP 优化的新压缩算法（Brotli）。

在 HTTP 协议里用 Accept、Accept-Encoding、Accept-Language 等请求头字段进行内容协商的时候，还可以用一种特殊的“q”参数表示权重来设定优先级，这里的“q”是“quality factor”的意思。

权重的最大值是 1，最小值是 0.01，默认值是 1，如果值是 0 就表示拒绝。

具体的形式是在数据类型或语言代码后面加一个“;”，然后是“q=value”。这里要提醒的是“;”的用法，在大多数编程语言里“;”的断句语气要强于“,”，而在 HTTP 的内容协商里却恰好反了过来，“;”的意义是小于“,”的。

例如下面的 Accept 字段：

```
Accept: text/html,application/xml;q=0.9,*/*;q=0.8
```

它表示浏览器最希望使用的是 HTML 文件，权重是 1，其次是 XML 文件，权重是 0.9，最后是任意数据类型，权重是 0.8。服务器收到请求头后，就会计算权重，再根据自己的实际情况优先输出 HTML 或者 XML。

内容协商的过程是不透明的，每个 Web 服务器使用的算法都不一样。但有的时候，服务器会在响应头里多加一个 Vary 字段，记录服务器在内容协商时参考的请求头字段，给出一点信息，

例如：Vary: Accept-Encoding,User-Agent,Accept这个 Vary 字段表示服务器依据了 Accept-Encoding、User-Agent 和 Accept 这三个头字段，然后决定了发回的响应报文。

http如何传输大文件

1. 数据压缩(压缩算法，gzip、deflate、br(brotli))
    1. Accept-Encoding(Request Header)
    2. Content-Encoding(Response Header)
2. 分块传输

    1. 分块传输可以流式收发数据，节约内存和带宽，使用响应头字段“Transfer-Encoding: chunked”来表示，分块的格式是 16 进制长度头 + 数据块；
3. 范围请求
    1. 范围请求可以只获取部分数据，即“分块请求”，实现视频拖拽或者断点续传，使用请求头字段“Range”和响应头字段“Content-Range”，响应状态码必须是 206；
4. 多段数据
    1. 也可以一次请求多个范围，这时候响应报文的数据类型是“multipart/byteranges”，body 里的多个部分会用 boundary 字符串分隔

注意：
`Content-Length`首部指示出报文中实体主体的字节大小。这个大小是包含了所有内容编码的， 比如，对文本文件进行了`gzip`压缩的话，`Content-Length`首部指的就是**压缩后的大小**而不是原始大小。

“Transfer-Encoding: chunked”和“Content-Length”这两个字段是互斥的，也就是说响应报文里这两个字段不能同时出现，一个响应报文的传输要么是长度已知，要么是长度未知（chunked），这一点你一定要记住

压缩算法：
`gzip`
表示采用 Lempel-Ziv coding （LZ77）压缩算法，以及 32 位 CRC 校验的编码方式。这个编码方式最初由 UNIX 平台上的 *gzip* 程序采用。出于兼容性的考虑，HTTP/1.1 标准提议支持这种编码方式的服务器应该识别作为别名的 `x-gzip` 指令。
`compress`
采用 Lempel-Ziv-Welch （LZW）压缩算法。这个名称来自 UNIX 系统的 *compress* 程序，该程序实现了前述算法。与其同名程序已经在大部分 UNIX 发行版中消失一样，这种内容编码方式已经被大部分浏览器弃用，部分因为专利问题（这项专利在 2003 年到期）。
`deflate`
采用 zlib 结构（在 RFC 1950 中规定），和 deflate 压缩算法（在 RFC 1951 中规定)。
`br`
表示采用 Brotli 算法的编码方式。

关于重定向

外部重定向和内部重定向的区别?

外部重定向，服务器会把重定向的地址给浏览器，然后浏览器再次的发起请求，地址栏的地址变化了。
内部重定向，服务器会直接把重定向的资源返给浏览器，不需要再次在浏览器发起请求，地址栏的地址不变。

重定向的应用场景
一个最常见的原因就是“资源不可用”，需要用另一个新的URI来代替。
不可用的原因：如域名变更、服务器变更、网站改版、系统维护。
另一个原因就是“避免重复”，让多个网址都跳转到一个URI，增加访问入口的同时还不会增加额外的工作量。如：有的网站会申请多个名称类似的域名，然后把它们重定向到主站上。

重定向的相关问题
第一个问题是“性能损耗”。重定向的机制决定了一个跳转会有两次请求-应答，比正常的访问多了一次。
第二个问题是“循环跳转”。如果重定向的策略设置欠考虑，可能会出现“A=>B=>C=>A”的无限循环。

301:表示永久性转移（Permanently Moved）

302:表示临时性转移（Temporarily Moved）

cookie应用

有了 Cookie，服务器就有了“记忆能力”，能够保存“状态”，那么应该如何使用 Cookie 呢?

Cookie 最基本的一个用途就是身份识别，保存用户的登录信息，实现会话事务。比如，你用账号和密码登录某电商，登录成功后网站服务器就会发给浏览器一个 Cookie，内容大概是“name=yourid”，这样就成功地把身份标签贴在了你身上。之后你在网站里随便访问哪件商品的页面，浏览器都会自动把身份 Cookie 发给服务器，所以服务器总会知道你的身份，一方面免去了重复登录的麻烦，另一方面也能够自动记录你的浏览记录和购物下单（在后台数据库或者也用 Cookie），实现了“状态保持”。

Cookie 的另一个常见用途是广告跟踪。你上网的时候肯定看过很多的广告图片，这些图片背后都是广告商网站（例如 Google），它会“偷偷地”给你贴上 Cookie 小纸条，这样你上其他的网站，别的广告就能用 Cookie 读出你的身

份，然后做行为分析，再推给你广告。这种 Cookie 不是由访问的主站存储的，所以又叫“第三方 Cookie”（third-party cookie）。如果广告商势力很大，广告到处都是，那么就比较“恐怖”了，无论你走到哪里它都会通过 Cookie 认出你来，实现广告“精准打击”。

Cookie 的缺点：
（1） 不安全。如果被中间人获取到 Cookie，完全将它作为用户凭证冒充用户。解决方案是使用 https 进行加密。
（2）有数量和大小限制。另外 Cookie 太大也不好，传输的数据会变大。
（3）客户端可能不会保存 Cookie。比如用 telnet 收发数据，用户禁用浏览器 Cookie 保存功能的情况。

cookie和session的区别

Cookie是存在客户端的，session是存在服务器的，都是用来保存用户的会话信息。

Cookie属于http协议的规定，而session不是协议规定，只是服务器的一种实现方式。

关于缓存

- `Cache-Control`，是一个相对时间；
- `Expires`，是一个绝对时间；

浏览器拿到一个网址的时候，先判断是否允许缓存，允许会先查看本地缓存:

1.有缓存并在缓存可用期那直接拿来用。

2.缓存不存在或者不可用 那需要请求。

浏览器拿到host，判断：

1.ip+port 那直接请求对应的服务器

2.域名 那开展一系列的dns递归查询：

先拿dns缓存，没有缓存->本地dns服务器->根dns服务器->顶级dns服务器->权威dns服务器->GSLB，查到ip返回最优ip组实现负载均衡，浏览器随机或者轮询取一个ip开始它的http请求之旅。

浏览器判断该网页是否允许缓存，然后添加Cache-Control的各种字段

参考：https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

- no-store 不允许缓存，用于某些变化非常频繁的数据
- no-cache 缓存必须进行验证，它的字面含义容易与 no-store 搞混，实际的意思并不是不允许缓存，而是可以缓存，但在使用之前必须要去服务器验证是否过期，是否有最新的版本；
- noly-if-cached 只接受代理的缓存等
- max-age 最大生存时间
- max-stale 短时间过期可用
- min-fresh 最短有效时间等
- If-Modified-Since/if-None-Match/Last-modified/ETag等字段用于判断服务端是否有更新。

其实不止服务器可以发“Cache-Control”头，浏览器也可以发“Cache-Control”，也就是说请求 - 应答的双方都可以用这个字段进行缓存控制，互相协商缓存的使用策略。当你点“刷新”按钮的时候，浏览器会在请求头里加一个“Cache-Control: max-age=0”。因为 max-age 是“生存时间”，max-age=0 的意思就是“我要一个最最新鲜的西瓜”，而本地缓存里的数据至少保存了几秒钟，所以浏览器就不会使用缓存，而是向服务器发请求。服务器看到 max-age=0，也就会用一个最新生成的报文回应浏览器

浏览器点击刷新和Ctrl+F5刷新时，会强制在请求头里加入Cache-Control字段，Cache-Control:max-age=0;no-cache的策略都是略过浏览器缓存直接向服务器请求最新的资源，但如果最新资源与缓存的是一致的，那么这里就存在重复只刷新数据浪费网络请求资源了。然而缓存也是有时间的，总不能直接用，所以这才引入了条件缓存，用于验证缓存是否失效。

那么，浏览器的缓存究竟什么时候才能生效呢？
试着点一下浏览器的“前进”“后退”按钮，再看开发者工具，你就会惊喜地发现“from disk cache”的字样，意思是没有发送网络请求，而是读取的磁盘上的缓存。

- 响应头部中的 `Last-Modified`：标示这个响应资源的最后修改时间；
- 请求头部中的 `If-Modified-Since`：当资源过期了，发现响应头中具有 Last-Modified 声明，则再次发起请求的时候带上 Last-Modified 的时间，服务器收到请求后发现有 If-Modified-Since 则与被请求资源的最后修改时间进行对比（Last-Modified），如果最后修改时间较新（大），说明资源又被改过，则返回最新资源，HTTP 200 OK；如果最后修改时间较旧（小），说明资源无新修改，响应 HTTP 304 走缓存。

- ETag 是什么呢?
  - ETag 是“实体标签”（Entity Tag）的缩写，是资源的一个唯一标识，主要是用来解决修改时间无法准确区分文件变化的问题。比如，一个文件在一秒内修改了多次，但因为修改时间是秒级，所以这一秒内的新版本无法区分。
- 请求头部中的 `If-None-Match`：当资源过期时，浏览器发现响应头里有 Etag，则再次向服务器发起请求时，会将请求头If-None-Match 值设置为 Etag 的值。服务器收到请求后进行比对，如果资源没有变化返回 304，如果资源变化了返回 200。

缓存代理

浏览器拿到一个网址的时候，先判断是否允许缓存，允许会先查看本地缓存:

```
1.有缓存并在缓存可用期那直接拿来用。
2.缓存不存在或者不可用那需要请求。
```

浏览器拿到host，判断：

```
1.ip+port 那直接请求对应的服务器
2.域名 那开展一系列的dns递归查询：先拿dns缓存，没有缓存->本地dns服务器->根dns服务器->顶级dns服务器->权威dns服务器->GSLB，查到ip返回最优ip组实现负载均衡，浏览器随机或者轮询取一个ip开始它的http请求之旅。
```

浏览器判断该网页是否允许缓存，然后添加Cache-Control的各种字段：

```
no-store是否允许缓存
no-cache缓存必须进行验证
noly-if-cached只接受代理的缓存等
max-age最大生存时间
max-stale 短时间过期可用
min-fresh 最短有效时间等。
If-Modified-Since/if-None-Match/Last-modified/ETag等字段用于判断服务端是否有更新。
```

然后将请求发给代理服务器。请求代理服务器，如果是第一次，要经历浏览器和代理服务器的3次tcp握手进行连接，连接成功，发送http请求。

代理服务器拿到请求，首先查看是否允许缓存，允许那就查看自己本地缓存有没有，通过查看max-age/max-stale/min-fresh等信息判断是否过期，没有过期直接拿来用，将数据返回给客户端。如果过期了，代理服务器将用客户端的请求，再次像真实服务器进行请求。如果也是第一次连接，需要经历代理服务器和真实服务器的3次tcp握手，连接成功，发送请求。
真实服务器收到请求之后，通过if-Modified-Since/Last-Modified/if-None-Match/ETag等字段判断是否有更新，没有更新，直接返回304。
如果有更新，则将数据打包http response 返回。返回头字段会添加Cache-Control字段，用来判断缓存的控制策略以及生存周期，no-store不允许缓存/no-cache使用缓存必须先验证/must-revalidate缓存不过期可用过期必须重新请求验证/proxy-revalidate缓存过期只要求代理进行请求验证 private不能在代理层保存只能在客户端保存/public缓存完全开放，s-maxage缓存在代理上可以缓存的时间，no-transform不允许代理对缓存做任何的改动。然后根据业务需求判断该地址是不是需要重定向，如果需要是短期的重定向还是永久的重定向，按需将状态码修改为301或者302。最后真实服务器将数据打包成http相应回给代理服务器。
代理服务器收到真实服务器的回应数据，首先会查看Cache-Control里的字段，是否允许它进行缓存，如果是private，代理服务器不进行缓存，直接返回给客户端。public则根据s-maxage/no-transform进行缓存，如果可以优化并且代理服务器需要优化，那可能会先优化数据，否则同时将数据回发给客户端。
客户端收到数据，如果是304，则直接拿缓存数据进行渲染，并修改相关缓存变量，比如时间，以及缓存使用策略。如果收到了301或者302，那么客户端会再次发起新的url请求，进行跳转到最终的页面。
最后，底层tcp 经过4次挥手，完成关闭连接。

TCP协议

TCP位于计算机网络七层模型中的传输层，在这一层，主要是网络的传输一些协议。由于网络自身无法保证可靠的分组传输（路由器负荷会导致丢弃数据包），所以TCP实现了自己的确认机制来保证数据的传输，它的特点：

- TCP 提供一种**面向连接的、可靠的**字节流服务
- 在一个 TCP 连接中，仅有两方进行彼此通信。广播和多播不能用于 TCP
- TCP 使用校验和，确认和重传机制来保证可靠传输
- TCP 给数据分节进行排序，并使用累积确认保证数据的顺序不变和非重复

- TCP 使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制
  TCP报文的几个重要的字段：

1. 源端口，16bits，范围0~65525。
2. 目的端口，16bits，范围同上。
3. sequence number： 数据序号，32bits，TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。
4. acknoledgement number： 确认号，32bits，期望收到对方的下一个报文段的数据的第一个字节的序号。
5. 数据偏移，4bits，单位为4字节，它指出报文数据距TCP报头的起始处有多远(TCP报文头长度)。
6. 保留字段 6bits，保留今后使用，目前置0处理。
7. URG：紧急比特，1bit，当 URG=1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)
   ACK：确认比特，1bit，只有当 ACK=1时确认号字段才有效。当 ACK=0 时，确认号无效
   PSH：推送比特，1bit，接收方 TCP 收到推送比特置1的报文段，就尽快地交付给接收应用进程，而不再等到整个缓存都填满了后再向上交付
   RST：复位比特，1bit，当RST=1时，表明TCP连接中出现严重差错(如由于主机崩溃或其他原因)，必须释放连接，然后再重新建立运输连接
   SYN：同步比特，1bit，同步比特 SYN 置为 1，就表示这是一个连接请求或连接接受报文
   FIN：终止比特，1bit，用来释放一个连接。当FIN=1 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接
8. 窗口大小，16bits，窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。
9. 检验和，16bits，检验和字段检验的范围包括首部和数据这两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。
10. 紧急指针字段，16bits，紧急指针指出在本报文段中的紧急数据的最后一个字节的序号。
11. 选项字段，长度可变。TCP首部可以有多达40字节的可选信息，用于把附加信息传递给终点，或用来对齐其它选项。 这部分最多包含40字节，因为TCP

头部最长是60字节（其中还包含前面讨论的20字节的固定部分）
选项的第一个字段kind说明选项的类型。有的TCP选项没有后面两个字段，仅包含1字节的kind字段。第二个字段length（如果有的话）指定该选项的总长度，该长度包括kind字段和length字段占据的2字节。第三个字段info（如果有的话）是选项的具体信息. kind=0是选项表结束选项
kind=1是空操作（nop）选项，没有特殊含义，一般用于将TCP选项的总长度填充为4字节的整数倍
kind=2是最大报文段长度选项,TCP连接初始化时，通信双方使用该选项来协商最大报文段长度（Max Segment Size，MSS）。TCP模块通常将MSS设置为（MTU-40）字节（减掉的这40字节包括20字节的TCP头部和20字节的IP头部）。这样携带TCP报文段的IP数据报的长度就不会超过MTU（假设TCP头部和IP头部都不包含选项字段，并且这也是一般情况），从而避免本机发生IP分片。对以太网而言，MSS值是1460（1500-40）字节。

说明

TCP如何保证可靠的连接

1. LISTEN 等待远程的TCP连接请求
2. SYN-SENT 发送了建立连接的请求，等待确认消息
3. SYN-RECIEVED 收到了对方建立连接的请求且发送了建立连接的请求（确认对方建立建立连接的请求）。等待对方确认自己发送的连接请求
4. ESTABLISHED 连接已经建立，可以正常进行数据传输
5. FIN-WAIT-1 等待对方确认刚刚发送的关闭连接的请求
6. FIN-WAIT-2 收到关闭连接请求的确认，等待对方发送关闭连接的请求
7. CLOSE-WAIT 确认了对方的关闭连接请求，等待本地用户关闭连接指令
8. LAST-ACK 被动关闭的一方，在CLOSE-WAIT状态下收到用户关闭连接的指令，发送关闭连接请求，等待确认
9. TIME-WAIT 主动关闭连接的一方收到对方发送的对方关闭连接请求的确认消息后,等待足够长的时间（2MSL）以确保对方接收到ACK包.最后回到CLOSED状态，释放网络资源
10. CLOSING 比较罕见的例外状态。正常情况下，发送FIN包后应该先收到（或同时收到）对方的ACK包，再收到对方的FIN包，而CLOSING状态表示发送FIN包后并没有收到对方的ACK包，却已收到了对方的FIN包。有两种情况可能导致这种状态：其一，如果双方几乎在同时关闭连接，那么就可能出现双

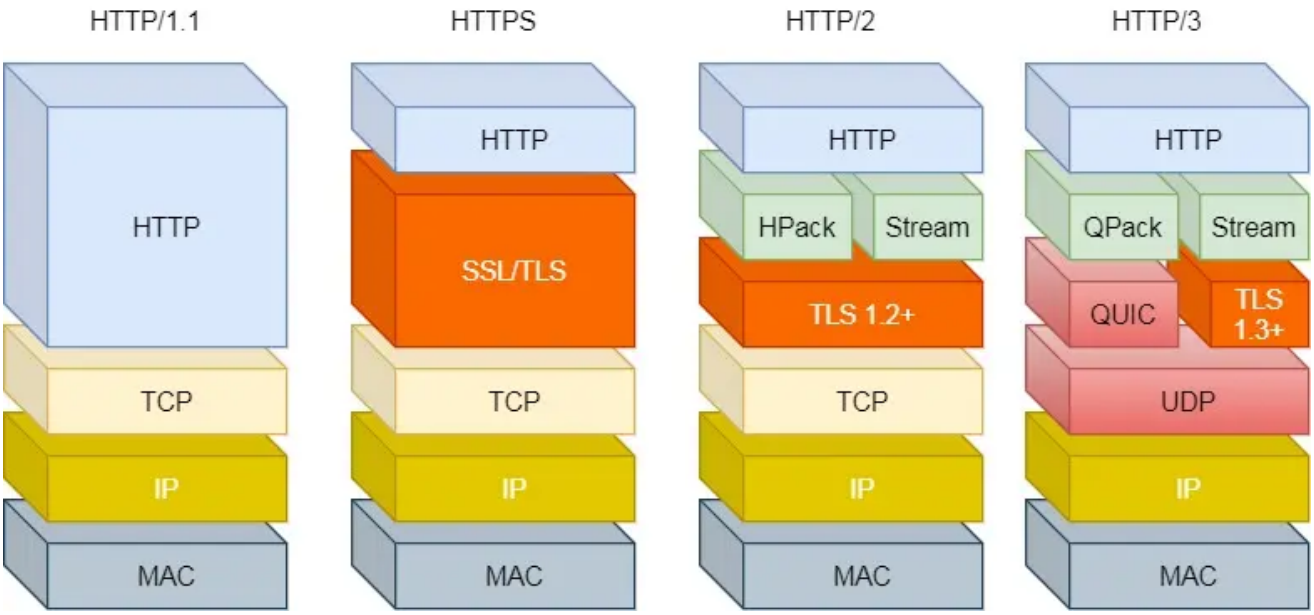
方同时发送FIN包的情况；其二，如果ACK包丢失而对方的FIN包很快发出，也会出现FIN先于ACK到达

11. CLOSED 关闭状态

TCP如何防止过载和拥塞

防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。https://xiaolincoding.com/network/3_tcp/tcp_feature.html#%E6%8B%A5%E5%A1%9E%E6%8E%A7%E5%88%B6

http的版本



短连接(HTTP/1.0)

HTTP 协议最初（0.9/1.0）是个非常简单的协议，通信过程也采用了简单的“请求 - 应答”方式。它底层的数据传输基于 TCP/IP，每次发送请求前需要先与服务器建立连接，收到响应报文后会立即关闭连接。因为客户端与服务器的整个连接过程很短暂，不会与服务器保持长时间的连接状态，所以就被称为“短连接”（short-lived connections）。早期的 HTTP 协议也被称为是“无连接”的协议。

短连接的缺点相当严重，因为在 TCP 协议里，建立连接和关闭连接都是非常“昂贵”的操作。TCP 建立连接要有“三次握手”，发送 3 个数据包，需要 1 个 RTT；关闭连接是“四次挥手”，4 个数据包需要 2 个 RTT。（定义往返时间(round trip time，简称RTT)）

长连接（HTTP/1.1）

针对短连接暴露出的缺点，HTTP 协议就提出了“长连接”的通信方式，也叫“持久连接”（persistent connections）、“连接保活”（keep alive）、“连接复用”（connection reuse）。

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。

但 HTTP/1.1 还是有性能瓶颈：

- 请求 / 响应头部（Header）未经压缩就发送，首部信息越多延迟越大。只能压缩 `Body` 的部分；
- 发送冗长的首部。每次互相发送相同的首部造成的浪费较多；
- 服务器是按请求的顺序响应的，如果服务器响应慢，会招致客户端一直请求不到数据，也就是队头阻塞；
- 没有请求优先级控制；
- 请求只能从客户端开始，服务器只能被动响应

HTTP/1.1 的队头阻塞

http是“半双工”，只能一来一回收发数据。

“队头阻塞”与短连接和长连接无关，而是由 HTTP 基本的“请求 - 应答”模型所导致的。因为 HTTP 规定报文必须是“一发一收”，这就形成了一个先进先出的“串行”队列。队列里的请求没有轻重缓急的优先级，只有入队的先后顺序，排在最前面的请求被最优先处理。如果队首的请求因为处理的太慢耽误了时间，那么队列里后面的所有请求也不得不跟着一起等待，结果就是其他的请求承担了不应有的时间成本。

队头阻塞问题会导致性能下降，可以用“并发连接”和“域名分片”技术缓解。

并发连接：

也就是同时对一个域名发起多个长连接，用数量来解决质量的问题。

域名分片：

申请多个域名，但这些域名最后都映射到同一个网站服务器，这样就可以突破浏览器的限制，让连接数增加

HTTP2.0(规范（RFC7540）在 2015 年发布)

1. 头部压缩

HTTP/2 会**压缩头**（Header）如果你同时发出多个请求，他们的头是一样的或是相似的，那么，协议会帮你**消除重复的部分**。

这就是所谓的 `HPACK` 算法：在客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就**提高速度**了。

2. 二进制格式

HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用了**二进制格式**，头信息和数据体都是二进制，并且统称为帧（frame）：**头信息帧（Headers Frame）和数据帧（Data Frame）**。

3. 并发传输

我们都知道 HTTP/1.1 的实现是基于请求-响应模型的。同一个连接中，HTTP 完成一个事务（请求与响应），才能处理下一个事务，也就是说在发出请求等待响应的过程中，是没办法做其他事情的，如果响应迟迟不来，那么后续的请求是无法发送的，也造成了**队头阻塞**的问题。

4. 服务器推送

HTTP/2 还在一定程度上改善了传统的「请求 - 应答」工作模式，服务端不再是被动地响应，可以**主动**向客户端发送消息。

客户端和服务器**双方都可以建立** Stream， Stream ID 也是有区别的，客户端建立的 Stream 必须是奇数号，而服务器建立的 Stream 必须是偶数号。

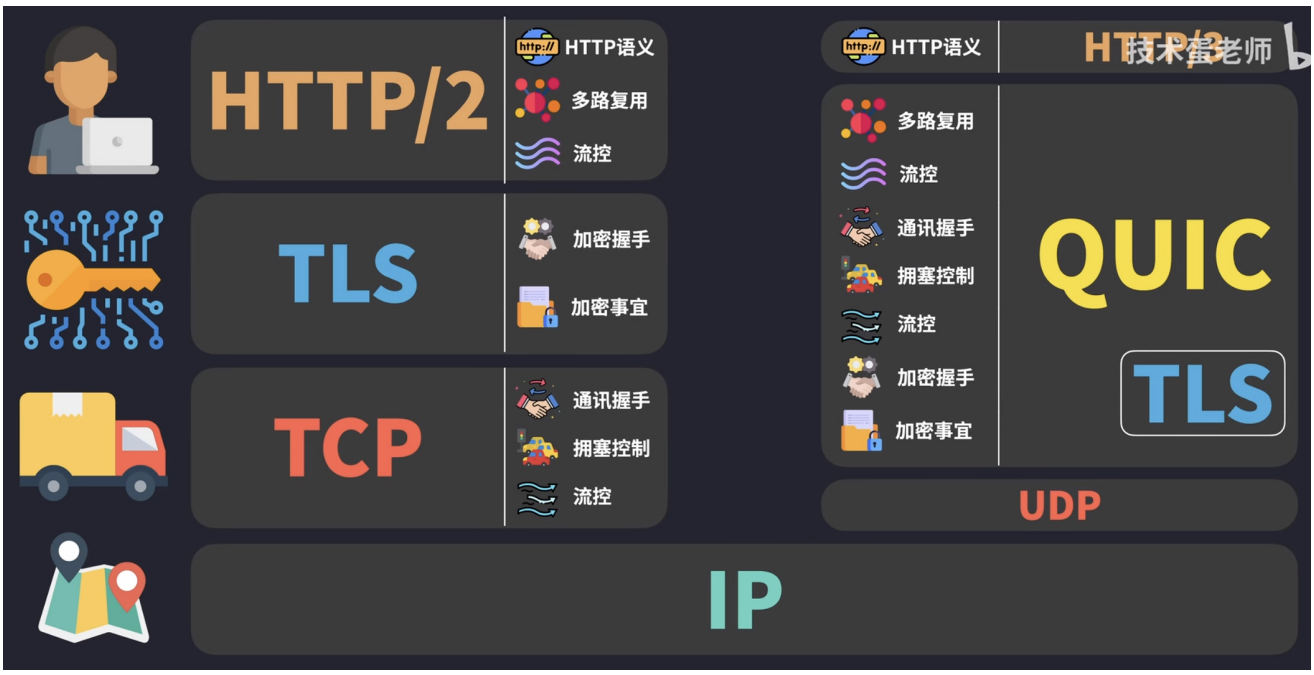
HTTP/2 通过 Stream 的并发能力，解决了 HTTP/1 队头阻塞的问题，看似很完美了，但是 HTTP/2 还是存在“队头阻塞”的问题，只不过问题不是在 HTTP 这一层面，而是在 TCP 这一层。

HTTP/2 是基于 TCP 协议来传输数据的，TCP 是字节流协议，TCP 层必须保证收到的字节数据是完整且连续的，这样内核才会将缓冲区里的数据返回给 HTTP 应用，那么当「前 1 个字节数据」没有到达时，后收到的字节数据只能存放在内核缓冲区里，只有等到这 1 个字节数据到达时，HTTP/2 应用层才能从内核中拿到数据，这就是 HTTP/2 队头阻塞问题。

缺点

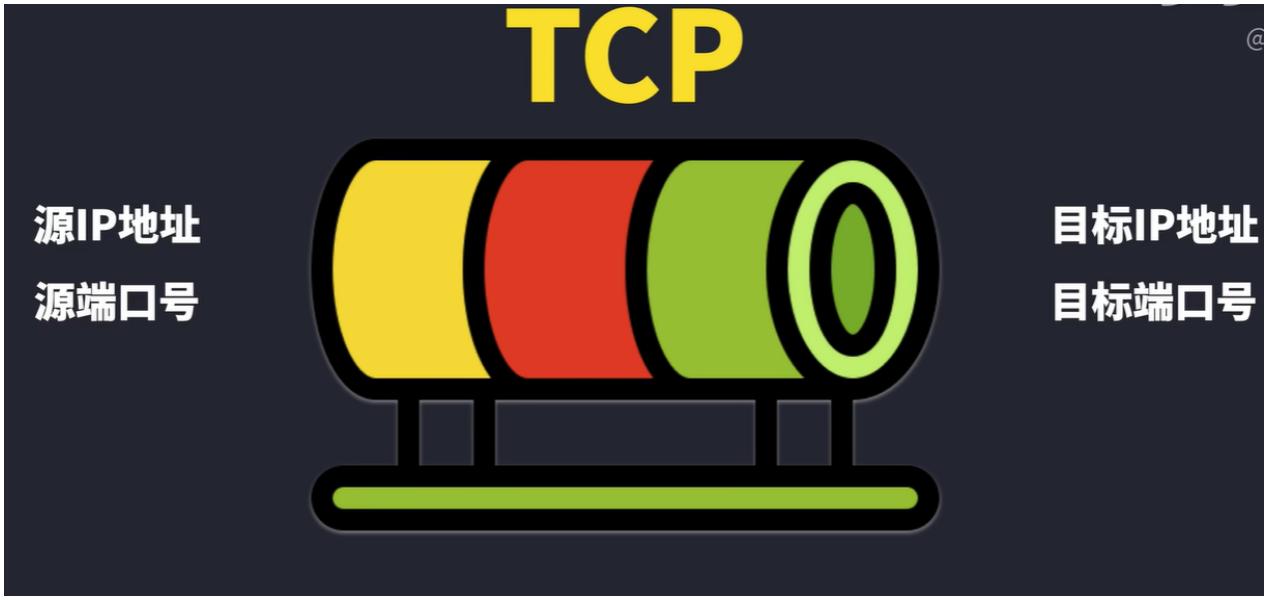
- **队头阻塞**，HTTP/2 多个请求跑在一个 TCP 连接中，如果序列号较低的 TCP 段在网络传输中丢失了，即使序列号较高的 TCP 段已经被接收了，应用层也无法从内核中读取到这部分数据，从 HTTP 视角看，就是多个请求被阻塞了；
- **TCP 和 TLS 握手时延**，TCL 三次握手和 TLS 四次握手，共有 3-RTT 的时延；
- **连接迁移需要重新连接**，移动设备从 4G 网络环境切换到 WIFI 时，由于 TCP 是基于四元组来确认一条 TCP 连接的，那么网络环境变化后，就会导致 IP 地址或端口变化，于是 TCP 只能断开连接，然后再重新建立连接，切换网络环境的成本高；

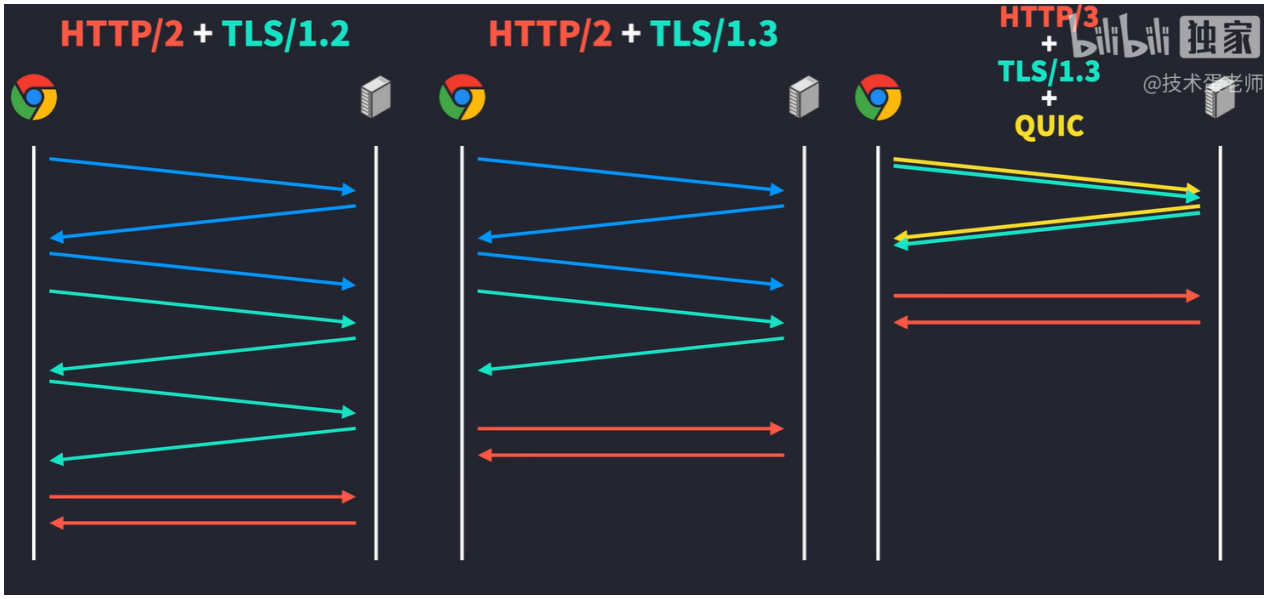
QUIC协议--HTTP/3.0



QUIC（Quick UDP Internet Connection）是谷歌制定的一种基于UDP的低时延的互联网传输层协议
QUIC 协议的特点：

- **无队头阻塞**，QUIC 连接上的多个 Stream 之间并没有依赖，都是独立的，也不会有底层协议限制，某个流发生丢包了，只会影响该流，其他流不受影响；
- **建立连接速度快**，因为 QUIC 内部包含 TLS1.3，因此仅需 1 个 RTT 就可以「同时」完成建立连接与 TLS 密钥协商，甚至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，达到 0-RTT 的效果。

- **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，客户端和服务器可以各自选择一组 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“无缝”地复用原连接，消除重连的成本；
  TCP的四元组：
  

  QUIC的连接：
  

android抓包

在HTTPS握手建立链接时，会使用非对称算法协商通讯使用的加密密钥，非对称加密算法需要两个密钥: 公钥和私钥，通常我们所说的HTTPS证书就可以简单理解成公钥，为了确保客户端接受到的证书的真实性，于是诞生了CA（Certification Authority）机构，CA机构发布的证书称为CA证书，基于CA机构的权威性，用户可以无条件的信任该机构发布的证书，为了鉴别CA证书的真实

性，防止CA证书被伪冒，同时避免套娃，因此在操作系统中内置了一个信任库，里面保存了可信任的CA证书集合，也称为系统根证书，用于校验服务端返回的证书的真实性。

证书验证的过程：
服务器返回的是证书链（不包括根证书，根证书预置在Android系统中），然后Android系统就可以使用信任的根证书（根公钥）解析证书链的根证书得到一级证书的公钥+摘要验签，然后拿一级证书的公钥解密一级证书拿到二级证书的公钥和摘要验签，再然后拿二级证书的公钥解密二级证书得到服务器的公钥和摘要验签，验证过程就结束了。

在Android系统中，可以点击`设置-系统安全-加密与凭据-信任的凭据`查看默认信任的CA证书。

在使用HTTPS协议进行网络通讯时，对HTTPS证书校验有多种处理方式，包括忽略证书链校验、系统证书链校验、证书绑定（SSL Pinning，代码校验和配置文件绑定）、双向校验等方式。

1. 忽略证书校验

在对HTTPS证书验证时，可以通过代码实现信任所有证书

2. 系统证书校验

OkHttp框架发起https请求时，默认就是使用的系统信任库证书链对服务端返回的证书进行验证

3. SSL PINNING

SSL PINNING是Google官方推荐的校验方式，原理是在客户端中预先设置好证书信息，握手时与服务端返回的证书进行比较，以确保服务端返回的证书的真实性，实现方式有两种，一种是在代码层实现，一种是通过network_security_config.xml配置文件完成。

需要使用到第三方插件，比如老牌的JustTrustMe模块，其绕过原理是通过Hook各网络框架的证书验证方法，替换其方法原有的逻辑，使校验失效

无论是系统证书链校验还是SSL PINNING，其本质都是客户端在校验服务端的证书，即单向校验。如果先保存一个证书在APP客户端中，HTTPS协议握手时客户端把APP中保存的证书发送给服务端，这种客户端校验服务端证书，同时服务端也校验客户端证书的方式称为双向校验。

4.代理检测

除了校验HTTPS证书防止中间人抓包以外，常见的方法还有通过检测代理防止抓包，其原理是检测到设备开启系统代理后，APP中通过代码实现禁用代理

可以使用iptables对请求进行强制转发，ProxyDroid全局代理工具就是通过iptables实现的，所以使用ProxyDroid开启代理，可以比较有效的绕过代理检测

arp

主机A的IP地址为192.168.1.1，MAC地址为0A-11-22-33-44-01；

主机B的IP地址为192.168.1.2，MAC地址为0A-11-22-33-44-02；

当主机A要与主机B通信时，地址解析协议可以将主机B的IP地址（192.168.1.2）解析成主机B的MAC地址，以下为工作流程：

第1步：根据主机A上的路由表内容，IP确定用于访问主机B的转发IP地址是192.168.1.2。然后A主机在自己的本地ARP缓存中检查主机B的匹配MAC地址。

第2步：如果主机A在ARP缓存中没有找到映射，它将询问192.168.1.2的硬件地址，从而将ARP请求帧广播到本地网络上的所有主机。源主机A的IP地址和MAC地址都包括在ARP请求中。本地网络上的每台主机都接收到ARP请求并且检查是否与自己的IP地址匹配。如果主机发现请求的IP地址与自己的IP地址不匹配，它将丢弃ARP请求。

第3步：主机B确定ARP请求中的IP地址与自己的IP地址匹配，则将主机A的IP地址和MAC地址映射添加到本地ARP缓存中。

第4步：主机B将包含其MAC地址的ARP回复消息直接发送回主机A。

第5步：当主机A收到从主机B发来的ARP回复消息时，会用主机B的IP和MAC地址映射更新ARP缓存。本机缓存是有生存期的，生存期结束后，将再次重复上面的过程。主机B的MAC地址一旦确定，主机A就能向主机B发送IP通信了。

Time to live (TTL)

IP包被路由器丢弃之前允许通过的最大网段数量。
一个数据包在经过一个路由器时，可传递的最长距离（跃点数）。每当数据包经过一个路由器时，其存活次数就会被减一。当其存活次数为0时，路由器便会取消数据包并发送一个ICMP TTL数据包给原数据包的发出者。其设计目的是防止数据包因不正确的路由表等原因造成的无限循环而无法送达及耗尽网络资源。

TTL与DNS TTL有区别。二者都是生存时间，前者指ICMP包的转发次数（跳数），后者指域名解析信息在DNS中的存在时间。
TL的作用是限制IP数据包在计算机网络中的存在的时间。TTL的最大值是255，TTL的一个推荐值是64。
自定义TTL 缺省值：（取值范围0~255）
Windows： 注册表展开“HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters”，找到“DefaultTTL”，将该值修改为十进制的“255”，重新启动服务器系统后即可

Linux： 编辑文件 /etc/sysctl.conf ，定义 net.ipv4.ip_default_ttl 值，保存后 sysctl -p 使其生效

ICMP

ping 是基于 ICMP 协议工作的。ICMP 全称 Internet Control Message Protocol，就是互联网控制报文协议。
主要用于在IP主机和路由器之间传递控制消息，用于报告主机是否可达、路由是否可用等。这些控制消息虽然并不传输用户数据，但是对于收集各种网络信息、诊断和排除各种网络故障以及用户数据的传递具有至关重要的作用。