HTTP知识体系

http 是我们几乎天天都要打交道的东西，相关知识点有点多，所以也有不少面试必问的点，这里做了一些整理，帮且大家树立完整的 http 知识体系，对面试官说 so easy

HTTP 的特点和缺点

特点：`无连接`、`无状态`、`灵活`、`简单快速`

- **无连接**：每一次请求都要连接一次，请求结束就会断掉，不会保持连接
- **无状态**：每一次请求都是独立的，请求结束不会记录连接的任何信息(**提起裤子就不认人的意思**)，减少了网络开销，这`是优点也是缺点`
- **灵活**：通过http协议中头部的`Content-Type`标记，可以传输任意数据类型的数据对象(文本、图片、视频等等)，非常灵活
- **简单快速**：发送请求访问某个资源时，只需传送请求方法和URL就可以了，使用简单，正由于http协议简单，使得http服务器的程序规模小，因而通信速度很快

缺点：`无状态`、`不安全`、`明文传输`、`队头阻塞`

- **无状态**：请求不会记录任何连接信息，没有记忆，就无法区分多个请求发起者身份是不是同一个客户端的，意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大
- **不安全**：`明文传输`可能被窃听不安全，缺少`身份认证`也可能遭遇伪装，还有缺少`报文完整性验证`可能遭到篡改
- **明文传输**：报文(header部分)使用的是明文，直接将信息暴露给了外界，`WIFI陷阱`就是复用明文传输的特点，诱导你连上热点，然后疯狂抓取你的流量，从而拿到你的敏感信息
- **队头阻塞**：开启`长连接`(下面有讲)时，只建立一个TCP连接，同一时刻只能处理一个请求，那么当请求耗时过长时，其他请求就只能阻塞状态(如何解决下面有讲)

HTTP 报文组成部分

http报文：由`请求报文`和`响应报文`组成

请求报文：由`请求行`、`请求头`、`空行`、`请求体`四部分组成

响应报文：由`状态行`、`响应头`、`空行`、`响应体`四部分组成

- **请求行**：包含http方法，请求地址，http协议以及版本
- **请求头/响应头**：就是一些key:value来告诉服务端我要哪些内容，要注意什么类型等，请求头/响应头每一个字段详解
- **空行**：用来区分首部与实体，因为请求头都是key:value的格式，当解析遇到空行时，服务端就知道下一个不再是请求头部分，就该当作请求体来解析了
- **请求体**：请求的参数
- **状态行**：包含http协议及版本、数字状态码、状态码英文名称
- **响应体**：服务端返回的数据

HTTP 请求方法(9种)

HTTP1.0：`GET`、`POST`、`HEAD`

HTTP1.1：`PUT`、`PATCH`、`DELETE`、`OPTIONS`、`TRACE`、`CONNECT`

方法	描述
GET	获取资源
POST	传输资源，通常会造成服务器资源的修改
HEAD	获得报文首部
PUT	更新资源
PATCH	对PUT的补充，对已知资源部分更新 菜鸟
DELETE	删除资源
OPTIONS	列出请求资源支持的请求方法，用来跨域请求
TRACE	追踪请求/响应路径，用于测试或诊断
CONNECT	将连接改为管道方式用于代理服务器(`隧道代理`下面有讲)

GET 和 POST 的区别

- `GET`在浏览器回退时是无害的，而`POST`会再次发起请求
- `GET`请求会被浏览器主动缓存，而`POST`不会，除非手动设置
- `GET`请求参数会被安逗保留在浏览器历史记录里，而`POST`中的参数不会被保留
- `GET`请求在`URL`中传递的参数有长度限制(浏览器限制大小不同)，而`POST`没有限制
- `GET`参数通过`URL`传递，`POST`放在`Request body`中
- `GET`产生的URL地址可以被收藏，而`POST`不可以
- `GET`没有`POST`安全，因为`GET`请求参数直接暴露在`URL`上，所以不能用来传递敏感信息
- `GET`请求只能进行`URL`编码，而`POST`支持多种编码方式
- 对参数的数据类型，`GET`只接受`ASCII`字符，而`POST`没有限制
- `GET`产生一个TCP数据包，`POST`产生两个数据包(Firefox只发一次)。GET浏览器把 http header和data一起发出去，响应成功200，POST先发送header，响应100 continue，再发送data，响应成功200

常见 HTTP 状态码

1xx: 指示信息——表示请求已接收，继续处理

2xx: 成功——表示请求已被成功接收

3xx: 重定向——表示要完成请求必须进行进一步操作

4xx: 客户端错误——表示请求有语法错误或请求无法实现

5xx: 服务端错误——表示服务器未能实现合法的请求

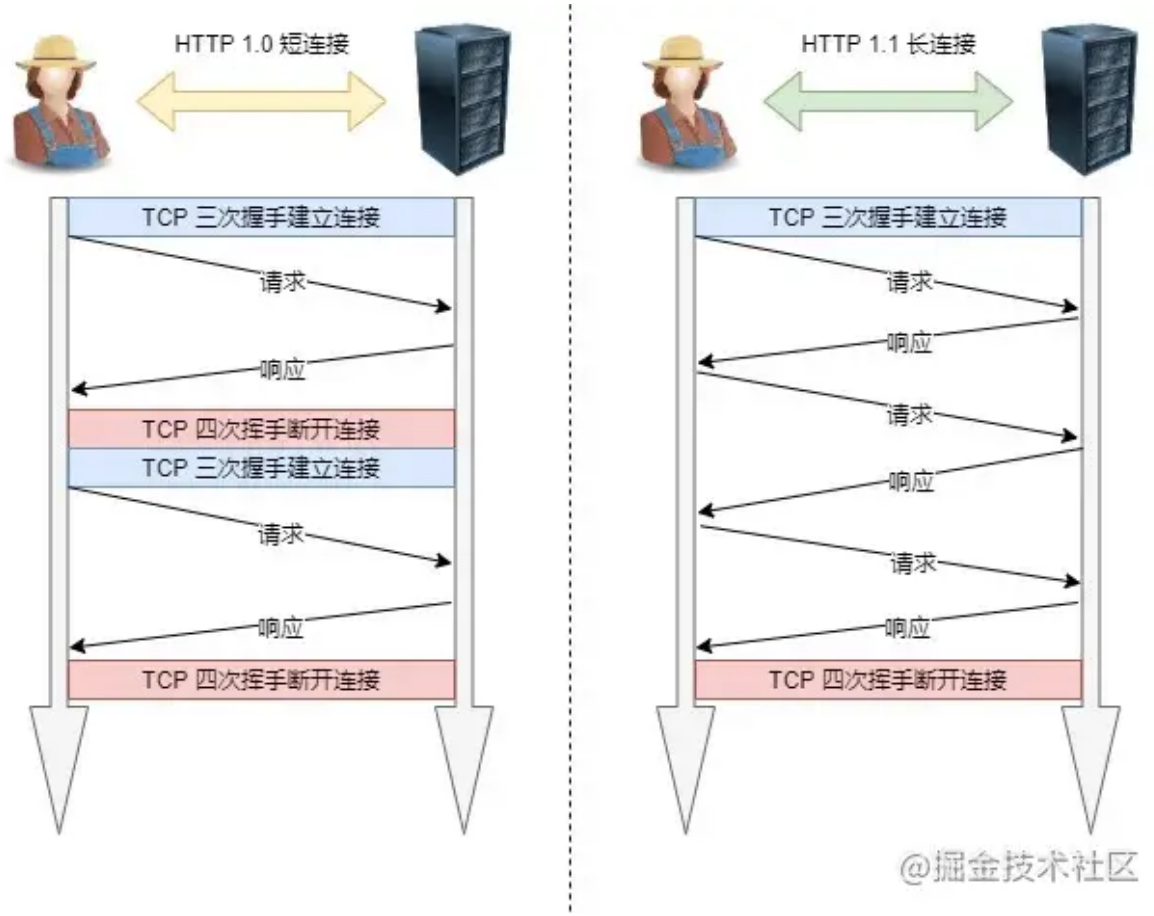
常见状态码：

状态码	描述
200	请求成功
206	已完成指定范围的请求(带Range头的GET请求),场景如video,audio播放文件较大,文件分片时
301	永久重定向
302	临时重定向
304	请求资源未修改，可以使用缓存的资源，不用在服务器取
400	请求有语法错误
401	没有权限访问
403	服务器拒绝执行请求，场景如不允许直接访问，只能通过服务器访问时
404	请求资源不存在
500	服务器内部错误，无法完成请求
503	请求未完成，因服务器过载、宕机或维护等

什么是持久连接/长连接

`http1.0`协议采用的是"请求-应答"模式，当使用普通模式，每个请求/应答客户与服务器都要新建一个连接，完成之后立即断开连接(`http`协议为`无连接`的协议)

`http1.1`版本支持长连接，即请求头添加`Connection: Keep-Alive`，使用Keep-Alive模式(又称持久连接，连接复用)建立一个`TCP`连接后使客户端到服务端的连接持续有效，可以发送/接受多个`http`请求/响应，当出现对服务器的后续请求时，Keep-Alive功能避免了建立或者重新建立连接



如图：短连接极大的降低了传输效率

长连接优缺点

优点

- `减少CPU及内存的使用`，因为不需要经常建立和关闭连接
- `支持管道化`的请求及响应模式
- `减少网络堵塞`，因为减少了TCP请求
- `减少了后续请求的响应时间`，因为不需要等待建立TCP、握手、挥手、关闭TCP的过程
- 发生错误时，也`可在不关闭连接的情况下进行错误提示`

缺点

一个长连接建立后，如果一直保持连接，对服务器来说是多么的浪费资源呀，而且长连接时间的长短，直接影响到服务器的并发数

还有就是可能造成`队头堵塞`(下面有讲)，造成信息延迟

如何避免长连接资源浪费?

- **客户端请求头声明**：`Connection: close`，本次通信后就关闭连接
- **服务端配置**：如Nginx，设置`keepalive_timeout`设置长连接超时时间，`keepalive_requests`设置长连接请求次数上限
- 系统内核参数设置

  :

  - `net.ipv4.tcp_keepalive_time = 60`，连接闲置60秒后，服务端尝试向客户端发送侦测包，判断TCP连接状态，如果没有收到ack反馈就在

- `net.ipv4.tcp_keepalive_intvl = 10`，就在10秒后再次尝试发送侦测包，直到收到ack反馈，一共会
- `net.ipv4.tcp_keepalive_probes = 5`，一共会尝试5次，要是都没有收到就关闭这个TCP连接了

什么是管线化(管道化)

`http1.1`在使用`长连接`的情况下，建立一个连接通道后，连接上消息的传递类似于

> 请求1 -> 响应1 -> 请求2 -> 响应2 -> 请求3 -> 响应3

`管理化`连接的消息就变成了类似这样

> 请求1 -> 请求2 -> 请求3 -> 响应1 -> 响应2 -> 响应3

`管线化`是在同一个TCP连接里**发一个请求后不必等其回来就可以继续发请求出去**，这可以减少整体的响应时间，但是服务器还是**会按照请求的顺序响应**请求，所以如果有许多请求，而前面的请求响应很慢，就产生一个著名的问题`队头堵塞`(下面有讲解决方法)

管线化的特点：

- 管线化机制通过持久连接完成，在`http1.1`版本才支持
- 只有`GET`请求和`HEAD`请求才可以进行管线化，而`POST`有所限制
- 初次创建连接时不应启动管线化机制，因为服务器不一定支持http1.1版本的协议
- 管线化不会影响响应到来的顺序，如上面的例子所示，响应返回的顺序就是请求的顺序
- 要求`客户端`和`服务端`都支持管线化，但并不要求服务端也对响应进行管线化处理，只是要求对于管线化的请求不失败即可
- 由于上面提到的服务端问题，开户管线化很可能并不会带来大幅度的性能提升，而且很多服务端和代理程序对管线化的支持并不好，因为浏览器(Chrome/Firefox)默认并未开启管线化支持

如何解决 HTTP 的队头阻塞问题

`http1.0`协议采用的是`请求-应答`模式，报文必须是`一发一收`，就形成了一个`先进先出`的串行队列，没有轻重缓急的优先级，只有入队的先后顺序，排在最前面的请求最先处理，就导致如果队首的请求耗时过长，后面的请求就只能处于阻塞状态，这就是著名的`队头阻塞`问题。解决如下：

并发连接

因为一个域名允许分配多个长连接，就相当于增加了任务队列，不至于一个队列里的任务阻塞了其他全部任务。以前在RFC2616中规定过客户端最多只能并发2个连接，但是现实是很多浏览器不按套路出牌，就是遵守这个标准T_T，所以在RFC7230把这个规定取消掉了，现在的浏览器标准中一个域名`并发连接`可以有`6~8`个，记住是6~8个，不是6个(**Chrome6个/Firefox8个**)

如果这个还不能满足你

继续，不要停...

域名分片

一个域名最多可以并发6~8个，那咱就多来几个域名

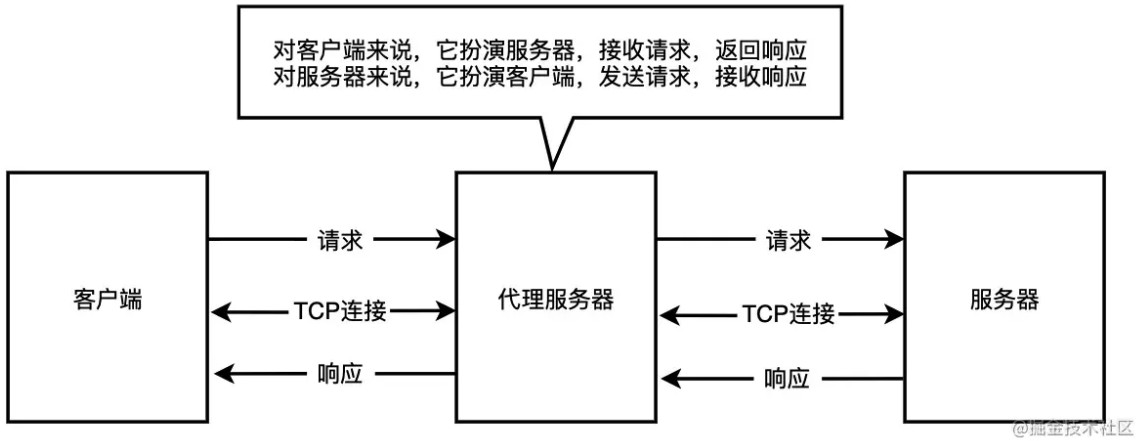
比如a.baidu.com，b.baidu.com，c.baidu.com，多准备几个`二级域名`，当我们访问baidu.com时，可以让不同的资源从不同的二域名中获取，而它们都指向同一台服务器，这样能够并发更多的长连接了

而在`HTTP2.0`下，可以一瞬间加载出来很多资源，因为支持多路复用，可以在一个TCP连接中发送多个请求

说一下 HTTP 代理

常见的代理有两种：`普通代理(中间人代理)`，`隧道代理`

普通代理(中间人代理)



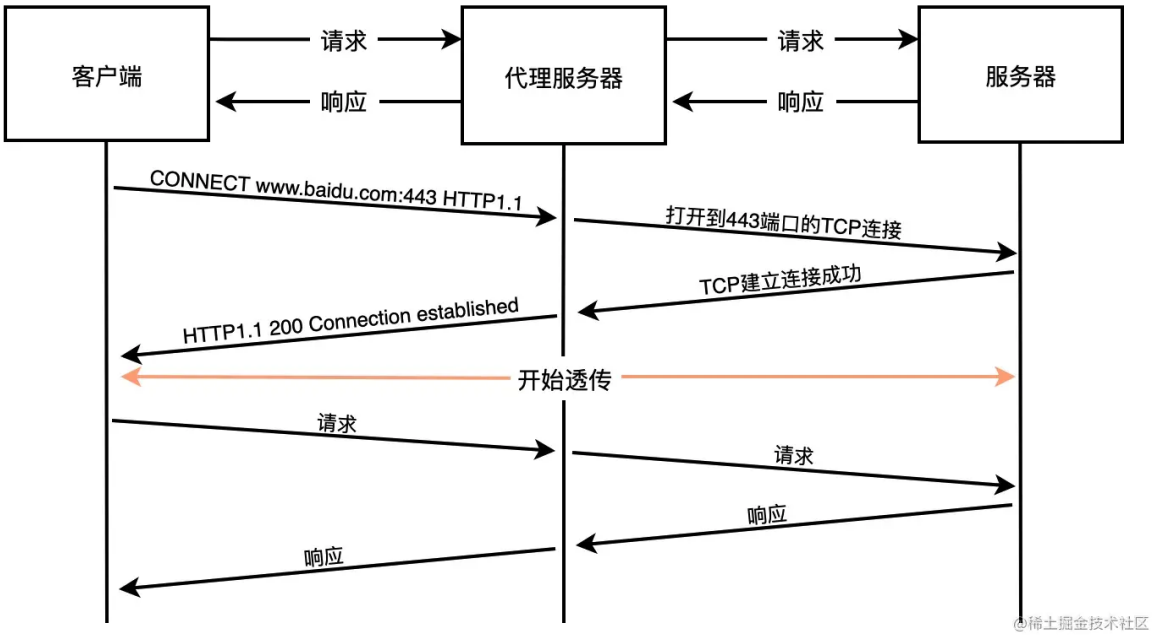
如图：代理服务器相当于一个中间人，一直帮两边传递东西，好可怜~~

不过它可以在中间可以帮我们过滤、缓存、负载均衡(多台服务器共用一台代理情况下)等一些处理

注意，实际场景中客户端和服务器之间可能有多个代理服务器

隧道代理

客户端通过 `CONNECT` 方法请求隧道代理创建一个可以到任意目标服务器和端口号的TCP连接，创建成功之后隧道代理只做请求和响应数据的转发，中间它不会做任何处理



为什么需要隧道代理呢?

我们都知道 `https` 服务是需要网站有证书的，而代理服务器显然没有，所以浏览器和代理之间无法创建 `TLS`，所以就有了隧道代理，它把浏览器的数据原样透传，这样就实现了通过中间代理和服务端进行TLS握手，然后进行加密传输

可能有人会问，那还要代理干嘛，直接请求服务器不是更好吗

代理服务器，到底有什么好处呢?

- **突破访问限制**：如访问一些单位或集团内部资源，或用国外代理服务器(翻墙)，就可以上国外网站看片等
- **安全性更高**：上网者可以通过这种方式隐藏自己的IP，免受攻击。还可以对数据过滤，对非法IP限流等
- **负载均衡**：客户端请求先到代理服务器，而代理服务器后面有多少源服务器，IP是多少，客户端是不知道的。因此，代理服务器收到请求后，通过特定的算法(随机算法、轮询、一致性hash、LUR(最近最少使用) 算法这里不细说了)把请求分发给不同的源服务器，让各个源服务器负载尽量均衡
- **缓存代理**：将内容缓存到代理服务器(这个下面一节详细说)

代理最常见的请求头

Via

是一个能用首部，由代理服务器添加，适用于正向和反向代理，在请求和响应首部均可出现，这个消息首部可以用来追踪消息转发情况，防止循环请求，还可以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，详情请看MDN

```
Via: 1.1 vegur
Via: HTTP/1.1 GWA
Via: 1.0 fred, 1.1 p.example.net
复制代码
```

X-Forwarded-For

记录客户端请求的来源IP，每经过一级代理(匿名代理除外)，代理服务器都会把这次请求的来源IP追加进去

```
X-Forwarded-For: client,proxy1,proxy2
复制代码
```

注意：与服务器直连的代理服务器的IP不会被追加进去，该代理可能过TCP连接的`Remote Address`字段获取到与服务器直连的代理服务器IP

X-Real-IP

一般记录真实发出请求的客户端的IP，还有`X-Forwarded-Host`和`X-Forwarded-Proto`分别记录真实发出请求的客户端的`域名`和`协议名`

代理中客户端IP伪造问题以及如何预防?

`X-Forwarded-For`是可以伪造的，比如一些通过X-Forwarded-For获取到客户端IP来限制刷票的系统就可以通过伪造该请求头达到刷票的目的，如果客户端请求显示指定了

```
X-Forwarded-For：192.168.1.108
复制代码
```

那么服务端收到的这个请求头，第一个IP就是伪造的

预防

1. 在对外Nginx服务器上配置

```
location / {
  proxy_set_header X-Forwarded-For $remote_addr
}
```
复制代码

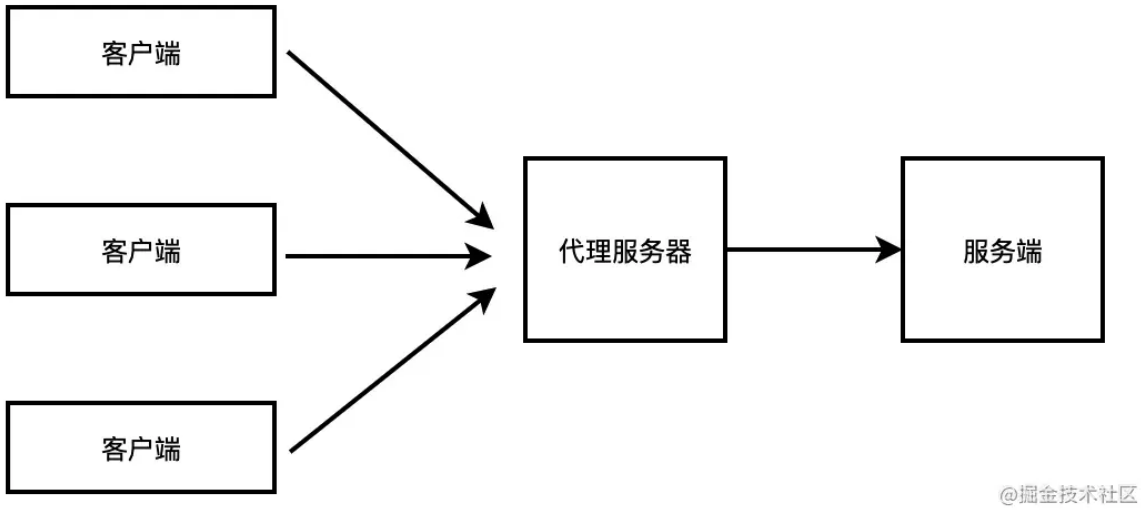
这样第一个IP就是从`TCP`连接客户端的IP，不会读取伪造的

1. 从右到左遍历`X-Forwarded-For`的IP，排除已知代理服务器IP和内网IP，获取到第一个符合条件的IP就可以了

正向代理和反向代理

正向代理

工作在客户端的代理为正向代理。使用正向代理的时候，需要在客户端配置需要使用的代理服务器，正向代理对服务端透明。比如抓包工具Fiddler、Charles以及访问一些外网网站的代理工具都是正向代理

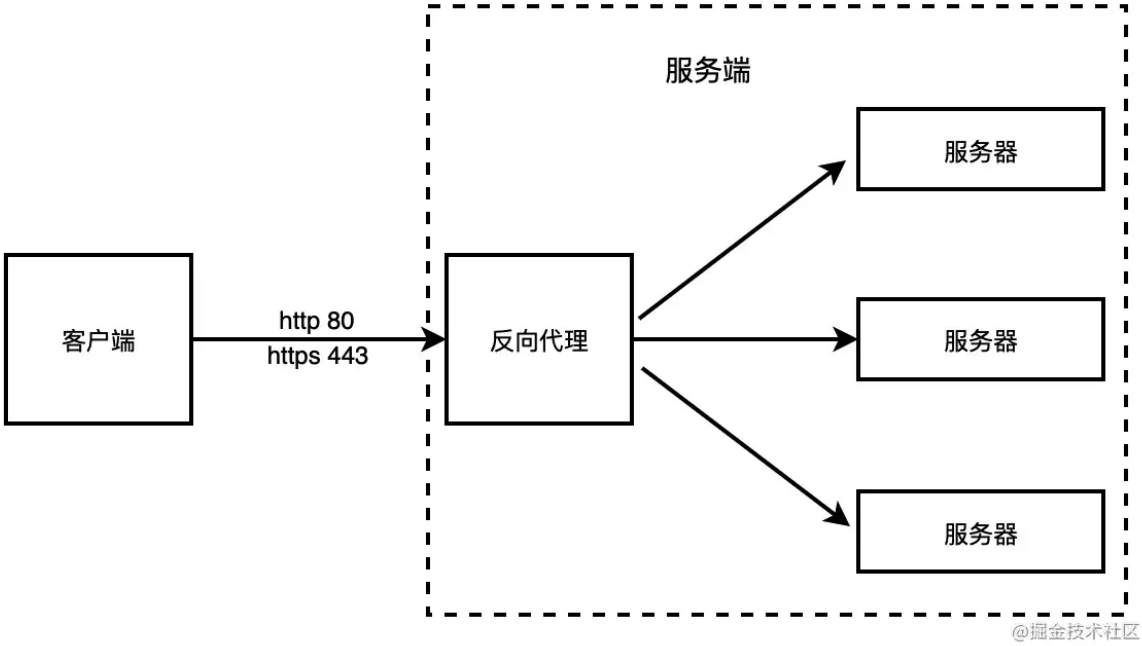


正向代理通常用于

- 缓存
- 屏蔽某些不健康的网站
- 通过代理访问原本无法访问的网站
- 上网认证，对用户访问进行授权

反向代理

工作在服务端的代理称为反向代理。使用反向代理的时候，不需要在客户端进行设置，反向代理对客户端透明。如Nginx就是反向代理



反向代理通常用于：`负载均衡`、`服务端缓存`、`流量隔离`、`日志`、`金丝雀发布`

代理中的长连接

在各个代理和服务器、客户端节点之间是一段一段的TCP连接，客户端通过代理访问目标服务器也叫逐段传输，用于逐段传输的请求头叫`逐段传输头`。

逐段传输头会在每一段传输的中间代理中处理掉，不会传给下一个代理

标准的逐段传输头有：`Keep-Alive`、`Transfer-Encoding`、`TE`、`Connection`、`Trailer`、`Upgrade`、`Proxy-Authorization`、`Proxy-Authenticate`。

Connection头决定当前事务完成后是否关闭连接，如果该值为keep-alive，则连接是`持久连接`不会关闭，使得对同一服务器的请求可以继续在该连接上完成

说一下 HTTP 缓存及缓存代理

关于http缓存在上一篇文章里有了详细介绍 (建议收藏)为什么第二次打开页面快？五步吃透前端缓存，让页面飞起

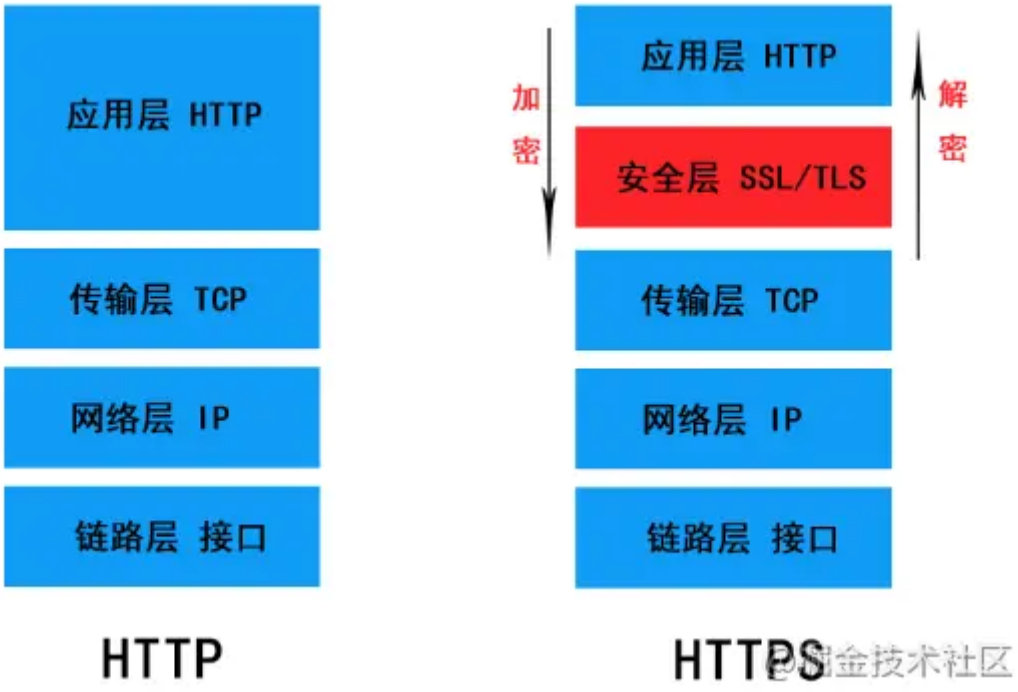
`缓存代理`就是让代理服务器接管一部分的服务端的http缓存，客户端缓存过期之后就近到代理服务器的缓存中获取，代理缓存过期了才请求源服务器，这样流量大的时候能明显降低源服务器的压力

注意`响应头`字段

- **Cache-Control**：值有`public`时，表示可以被所有终端缓存，包括`代理服务器`、`CDN`。值有`private`时，只能被`终端浏览器`缓存，CDN、代理等中继服务器都不可以缓存。

HTTPS

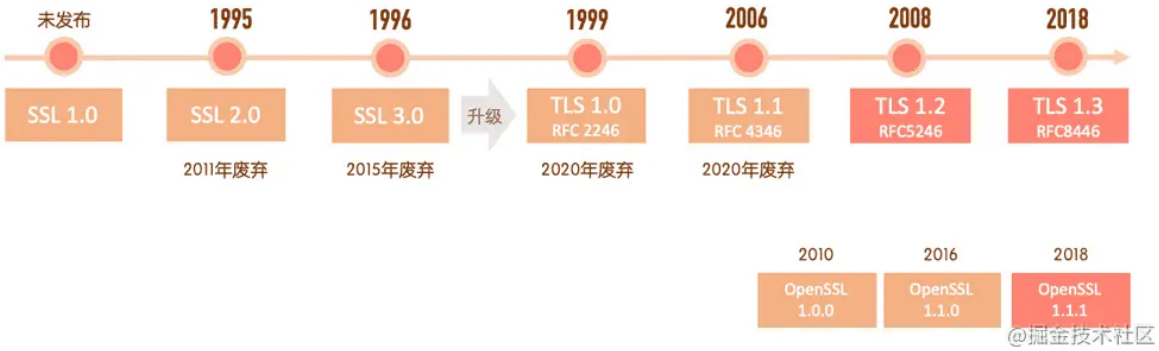
`HTTPS`是超文本传输安全协议，即`HTTP + SSL/TLS`。说白了，就是一个加强版的HTTP



HTTP本文开始讲了，所以我们要理解HTTPS的精华，就要先弄清楚这个`SSL/TLS`了

SSL/TLS

一张图让你理解SSL和TLS的关系

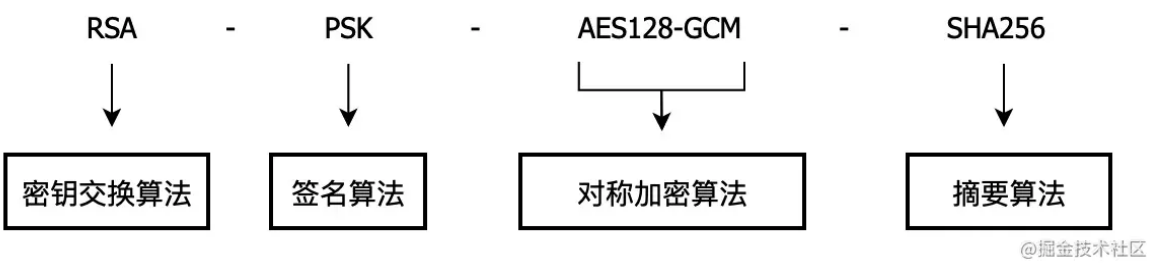


如图，TLS是SSL的升级版，而且TLS1.2版本以下都已废弃，目前主要用的是`TLS 1.2`和`TLS 1.3`。而**OpenSSL**则是**开源版本**的

那么它到底是个啥呢?

浏览器和服务器通信之前会先协商，选出它们都支持的`加密套件`，用来实现安全的通信。[常见加密套件]

随便拿出一个加密套件举例，如：**RSA-PSK-AES128-GCM-SHA256**，就是长这样，代表什么意思呢，我们看图



- `RSA`：表示握手时用RSA算法交换密钥

- `PSK`：表示使用PSK算法签名
- `AES128-GCM`：表示使用AES256对称加密算法通信，密钥长度128，分组模式GCM。TLS 1.3中只剩下称加密算法有**AES**和**CHACHA20**，分组模式只剩下**GCM**和**POLY1305**
- `SHA256`：表示使用SHA256算法验证信息完整性并生成随机数。TLS 1.3中哈希摘要算法只剩下**SHA256**和**SHA384**了

为什么需要用到这么多算法呢?

为了保证安全，TLS需要保证信息的：`机密性`、`可用性`、`完整性`、`认证性`、`不可否认性`，每一种算法都有其特定的用处

HTTPS 中 TLS 的加密算法

为什么说https是安全的?
https一定是安全的吗？（考察https中间人劫持，我另一篇有关于详细介绍网络安全）
有什么解决办法?
https的证书校验过程是怎么样的?
证书校验用到了哪些算法?

对称加密算法

就是加密和解密使用同一个密钥。如`AES、DES`。加解密过程：

1. 浏览器给服务器发送一个随机数`client-random`和一个支持的加密方法列表
2. 服务器给浏览器返回另一个随机数`server-random`和双方都支持的加密方法
3. 然后两者用加密方法将两个随机数混合生成密钥，这就是通信双上加解密的密钥

问题是双方如何安全的传递两个随机数和加密方法，直接传给客户端，那过程中就很可能被窃取，别人就能成功解密拿到数据，往下看

不对称加密算法

就是一对密钥，有`公钥`(public key)和`私钥`(private key)，其中一个密钥加密后的数据，只能让另一个密钥进行解密。如`RSA`、`ECDHE`。加解密过程：

1. 浏览器给服务器发送一个随机数`client-random`和一个支持的加密方法列表
2. 服务器把另一个随机数`server-random`、`加密方法`、`公钥`传给浏览器
3. 然后浏览器用公钥将两个随机数加密，生成密钥，这个密钥只能用`私钥`解密

使用公钥反推出私钥是非常困难，但不是做不到，随着计算机运算能力提高，非对称密钥`至少要2048位才能保证安全性`，这就导致性能上要比对称加密要差很多

所以!

TLS实际用的是`两种算法的混合加密`。**通过 非对称加密算法 交换 对称加密算法 的密钥，交换完成后，再使用对称加密进行加解密传输数据。**这样就保证了会话的机密性。过程如下

1. 浏览器给服务器发送一个随机数`client-random`和一个支持的加密方法列表
2. 服务器把另一个随机数`server-random`、`加密方法`、`公钥`传给浏览器
3. 浏览器又生成另一个随机数`pre-random`，并用公钥加密后传给服务器
4. 服务器再用私钥解密，得到`pre-random`
5. 浏览器和服务器都将三个随机数用加密方法混合生成最终密钥

这样即便被截持，中间人没有私钥就拿不到`pre-random`，就无法生成最终密钥。

可又有问题来了，如果一开始就被DNS截持，我们拿到的公钥是中间人的，而不是服务器的，数据还是会被窃取，所以`数字证书`来了，往下看，先简单说一下摘要算法

摘要算法

主要用于保证信息的完整性。常见的**MD5算法**、**散列函数**、**哈希函数**都属于这类算法，其特点就是`单向性`、`无法反推原文`

假如信息被截取，并重新生成了摘要，这时候就判断不出来是否被篡改了，所以需要给摘要也通过会话密钥进行加密，这样就看不到明文信息，保证了安全性，同时也保证了完整性

如何保证数据不被篡改？签名原理和证书?

数字证书(数字签名)

它可以帮我们`验证服务器身份`。因为如果没有验证的话，就可能被`中间人劫持`，假如请求被中间人截获，中间人把他自己的公钥给了客户端，客户端收到公钥就把信息发给中间人了，中间人解密拿到数据后，再请求实际服务器，拿到服务器公钥，再把信息发给服务器

这样不知不觉间信息就被人窃取了，所以在结合对称和非对称加密的基础上，又添加了数字证书认证的步骤，让服务器证明自己的身份

数字证书需要向有权威的`认证机构(CA)`获取授权给服务器。首先，`服务器`和`CA机构``分别有一对密钥`(公钥和私钥)，然后是如何生成数字证书的呢?

- CA机构通过摘要算法生成服务器公钥的`摘要`(哈希摘要)
- CA机构通过**CA私钥**及特定的签名算法**加密**摘要，生成`签名`
- 把`签名`、`服务器公钥`等信息打包`放入数字证书`，并返回给服务器

服务器配置好证书，以后客户端连接服务器，都先把证书发给客户端验证并获取服务器的公钥。

证书验证流程:

- 使用**CA公钥**和声明的签名算法对CA中的签名进行**解密**，得到服务器公钥的`摘要内容`
- 再用摘要算法对证书里的服务器公钥生成摘要，再把这个摘要和上一步得到的摘要`对比`，如果一致说明证书合法，里面的公钥也是正确的，否则就是非法的

证书认证又分为`单向认证`和`双向认证`

单向认证：服务器发送证书，客户端验证证书
双向认证：服务器和客户端分别提供证书给对方，并互相验证对方的证书

不过大多数https服务器都是单向认证，如果服务器需要验证客户端的身份，一般通过用户名、密码、手机验证码等之类的凭证来验证。只有更高级别的要求的系统，比如大额网银转账等，就会提供双向认证的场景，来确保对客户身份提供认证性

HTTPS 连接过程和优化

我们知道了https就只是比http多了一步TLS连接

TLS连接是怎么回事呢，根据TLS版本和密钥交换法不同，过程也不一样，有三种方式

RSA握手

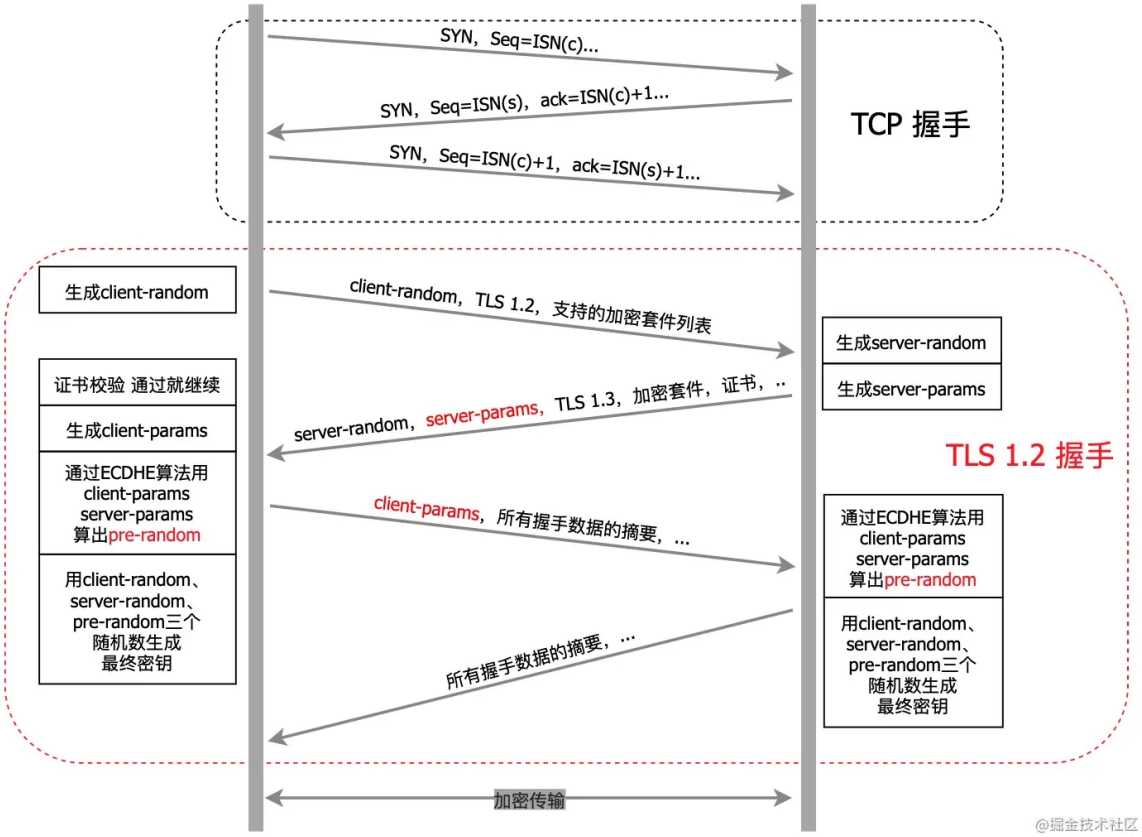
早期的TLS密钥交换法都是使用RSA算法，它的握手流程是这样子的

1. 浏览器给服务器发送一个随机数`client-random`和一个支持的加密方法列表
2. 服务器把另一个随机数`server-random`、`加密方法`、`公钥`传给浏览器
3. 浏览器又生成另一个随机数`pre-random`，并用公钥加密后传给服务器
4. 服务器再用私钥解密，得到`pre-random`，此时浏览器和服务器都得到三个随机数了，各自将三个随机数用加密方法混合生成最终密钥

然后开始通信

TLS 1.2 版

`TLS 1.2` 版的用的是 `ECDHE密钥交换法`，看图



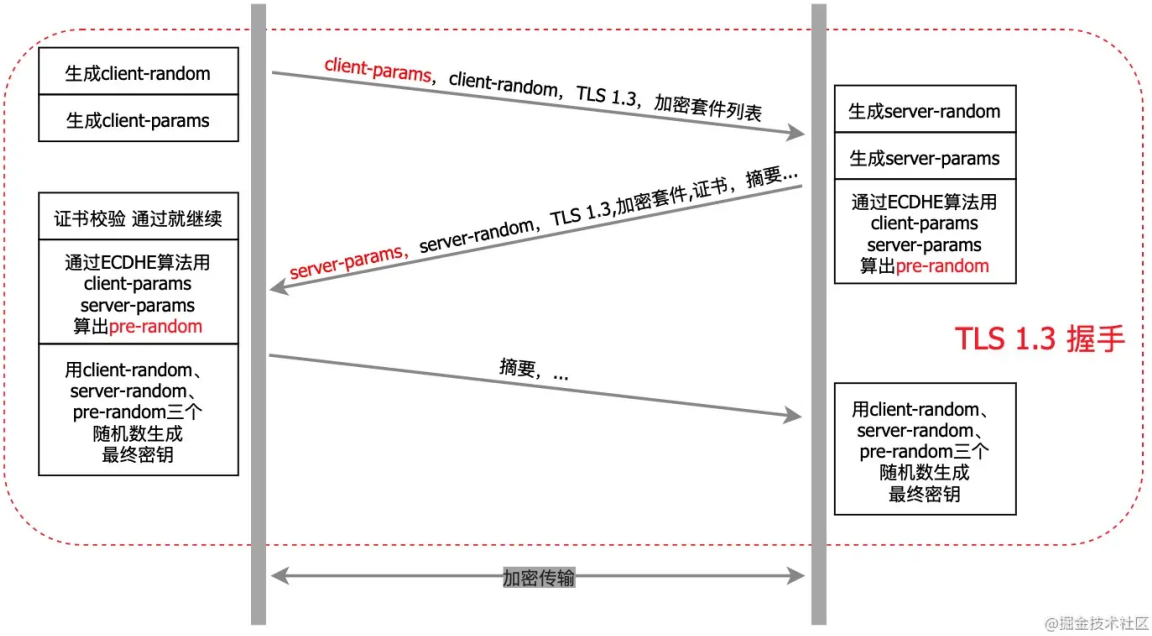
1. 浏览器给服务器发送一个随机数 `client-random`、TLS版本和一个支持的加密方法列表
2. 服务器生成一个椭圆曲线参数 `server-params`、随机数 `server-random`、`加密方法`、`证书` 等传给浏览器
3. 浏览器又生成椭圆曲线参数 `client-params`，握手数据摘要等信息传给服务器
4. 服务器再返回摘要给浏览器确认应答

这个版本不再生成椭圆曲线参数 `cliend-params` 和 `server-params`，而是在服务器和浏览器两边都得到 `server-params` 和 `client-params` 之后，就用ECDHE算法直接算出 `pre-random`，这就两边都有了三个随机数，然后各自再将三个随机加密混合生成最终密钥

TLS 1.3版

在TLS1.3版本中废弃了RSA算法，因为RSA算法可能泄露私钥导致历史报文全部被破解，而ECDHE算法每次握手都会生成临时的密钥，所以就算私钥被破解，也只能破解一条报文，而不会对之前的历史信息产生影响，，所以在TLS 1.3中彻底取代了RSA。目前主流都是用 `ECDHE算法` 来做密钥交换的

TLS1.3版本中握手过程是这样子的



1. 浏览器生成`client-params`、和`client-random`、TLS版本和加密方法列表发送给服务器
2. 服务器返回`server-params`、`server-random`、`加密方法`、`证书`、`摘要`等传给浏览器
3. 浏览器确认应答，返回握手数据摘要等信息传给服务器

简单说就是简化了握手过程，只有三步，把原来的两个RTT打包成一个发送了，所以减少了传输次数。这种握手方式也叫`1-RTT`握手

这种握手方还有优化空间吗?

有的，用会话复用

会话复用

会话复用有两种方式：`Session ID`和`Session Ticket`

`Session ID`：就是客户端和服务器首次连接手各自保存会话ID，并存储会话密钥，下次再连接时，客户端发送ID过来，服务器这边再查找ID，如果找到了就直接复用会话，密钥也不用重新生成

可是这样的话，在客户端数量庞大的时候，对服务器的存储压力可就大了

所以出来了第二种方式`Session Ticket`：就是双方连接成功后服务器加密会话信息，用Session Ticket消息发给客户端存储起来，下次再连接时就把这个Session Ticket解密，验证有没有过期，如果没有过期就复用会话。原理就是把存储压力分给客户端。

这样就万无一失了吗?

No，这样也存在安全问题。因为每次要用一个固定的密钥来解密Session Ticket，一旦密钥被窃取，那所有历史记录也就被破解了，所以只能尽量避免这种问题`定期更换密钥`。毕竟节省了不少生成会话密钥和这些算法的耗时，性能还是提升了嘛

那刚说了`1-RTT`，那能不能优化到`0-RTT`呢

还真可以，做法就是发送Session Ticket的时候带上应用数据，不用等服务端确认。这种方式被称为`PSK`(Pre-Shared Key)

这样万无一失了吗?

尴了个尬，还是不行。这PSK要是被窃取，人家不断向服务器重发，就直接增加了服务器被攻击的风险

虽然不是绝对安全，但是现行架构下最安全的解决文案了，大大增加了中间人的攻击成本

HTTPS优缺点

优点

- 内容加密，中间无法查看原始内容
- 身份认证，保证用户访问正确。如访问百度，即使DNS被劫持到第三方站点，也会提醒用户没有访问百度服务，可能被劫持
- 数据完整性，防止内容被第三方冒充或篡改
- 虽然不是绝对安全，但是现行架构下最安全的解决文案了，大大增加了中间人的攻击成本

缺点

- 要钱，功能越强大的证书费用越贵
- 证书需要绑定IP，不能在同一个IP上绑定多个域名
- https双方加解密，耗费更多服务器资源
- https握手更耗时，降低一定用户访问速度(优化好就不是缺点了)

HTTP 和 HTTPS 的区别

- HTTP是`明文传输`，不安全的，HTTPS是`加密传输`，安全的多
- HTTP标准端口是`80`，HTTPS标准端口是`443`
- HTTP不用认证证书`免费`，HTTPS需要认证证书`要钱`
- `连接方式不同`，HTTP三次握手，HTTPS中TLS1.2版本7次，TLS1.3版本6次
- HTTP在OSI网络模型中是在`应用层`，而HTTPS的TLS是在`传输层`
- HTTP是`无状态`的，HTTPS是`有状态`的

HTTPS 的性能优化

访问速度优化

1. `会话复用`，上面说了，复用session可以减少 CPU 消耗，因为不需要进行非对称密钥交换的计算。可以提升访问速度，不需要进行完全握手阶段二，节省了一个 RTT 和计算耗时。
2. 使用 `SPDY` 或者 `HTTP2`。SPDY 最大的特性就是多路复用，能将多个 HTTP 请求在同一个连接上一起发出去，不像目前的 HTTP 协议一样，只能串行地逐个发送请求。Pipeline 虽然支持多个请求一起发送，但是接收时依然得按照顺序接收，本质上无法解决并发的问题。HTTP2支持多路复用，有同样的效果。
3. 设置`HSTS`，服务端返回一个 HSTS 的 http header，浏览器获取到 HSTS 头部之后，在一段时间内，不管用户输入www.baidu.com还是http://www.baidu.com ，都会默认将请求内部跳转成https://www.baidu.com。Chrome, firefox, ie 都支持了 HSTS。
4. `Nginx`设置`Ocsp stapling`。Ocsp 全称在线证书状态检查协议 (rfc6960)，用来向 CA 站点查询证书状态，比如是否撤销。通常情况下，浏览器使用 OCSP 协议发起查询请求，CA 返回证书状态内容，然后浏览器接受证书是否可信的状态。这个过程非常消耗时间，因为 CA 站点有可能在国外，网络不稳定，RTT 也比较大。如果不需要查询则可节约时间。
5. `False start`。简单概括 False start 的原理就是在 clientkeyexchange 发出时将应用层数据一起发出来，能够节省一个 RTT。

计算性能优化

1. 优先使用 ECC椭圆加密算术
2. 使用最新版的 OpenSSL
3. TLS 远程代理计算
4. 硬件加速方案

HTTP 版本

1991年HTTP 0.9版，只有一个GET，而且只支持纯文本内容，早已过时就不讲了

HTTP 1.0(1996年)

- 任意数据类型都可以发送
- 有GET、POST、HEAD三种方法
- 无法复用TCP连接(长连接)
- 有丰富的请求响应头信息。以header中的`Last-Modified`/`If-Modified-Since`和`Expires`作为缓存标识

HTTP 1.1(1997年)

- 引入更多的请求方法类型`PUT`、`PATCH`、`DELETE`、`OPTIONS`、`TRACE`、`CONNECT`
- 引入长连接，就是TCP连接默认不关闭，可以被多个请求复用，通过请求头connection:keep-alive设置
- 引入管道连接机制，可以在同一TCP连接里，`同时发送`多个请求
- 强化了缓存管理和控制`Cache-Control`、`ETag`/`If-None-Match`
- 支持分块响应，断点续传，利于大文件传输，能过请求头中的`Range`实现
- 使用了`虚拟网络`，在一台物理服务器上可以存在多个虚拟主机，并且共享一个IP地址

缺点：主要是连接缓慢，服务器只能按顺序响应，如果某个请求花了很长时间，就会出现请求队头阻塞

虽然出了很多优化技巧：为了增加并发请求，做域名拆分、资源合并、精灵图、资源预取...等等

最终为了推进从协议上进行优化，Google跳出来，推出`SPDY`协议

SPDY(2009年)

SPDY（读作“SPeeDY”）是Google开发的基于TCP的`会话层协议`

主要通过帧、多路复用、请求优先级、HTTP报头压缩、服务器推送以最小化网络延迟，提升网络速度，优化用户的网络使用体验

原理是在SSL层上增加一个SPDY会话层，以在一个TCP连接中实现并发流。通常的HTTP GET和POST格式仍然是一样的，然而SPDY为编码和传输数据设计了一个新的帧格式。因为流是双向的，所以可以在客户端和服务端启动

虽然诞生后很快被所有主流浏览器所采用，并且服务器和代理也提供了支持，但是SPDY核心人员后来都参加到HTTP 2.0开发中去了，自HTTP2.0开发完成就不再支持SPDY协议了，并在Chrome 51中删掉了SPDY的支持

HTTP 2.0(2015年)

说出http2中至少三个新特性?

- 使用新的`二进制协议`，不再是纯文本，避免文本歧义，缩小了请求体积
- `多路复用`，同域名下所有通信都是在单链接(双向数据流)完成，提高连接的复用率，在拥塞控制方面有更好的能力提升
- 使用`HPACK算法将头部压缩`，用`哈夫曼编码`建立索表，传送索引大大节约了带宽
- 允许`服务端主动推送`数据给客户端
- 增加了安全性，使用HTTP 2.0，要求必须至少TLS 1.2
- 使用虚拟的流传输消息，解决了应用层的队头阻塞问题

缺点

- TCP以及TCP+TLS建立连接的延时，HTTP2使用TCP协议来传输的，而如果使用HTTPS的话，还需要TLS协议进行安全传输，而使用TLS也需要一个握手过程，在传输数据之前，导致我们花掉3~4个

RTT
- TCP的队头阻塞并没有彻底解决。在HTTP2中，多个请求跑在一个TCP管道中，但当HTTP2出现丢包时，整个TCP都要开始等待重传，那么就会阻塞该TCP连接中的所有请求

SPDY 和 HTTP2 的区别

- 头部压缩算法，SPDY是通用的`deflate算法`，HTTP2是专门为压缩头部设计的`HPACK算法`
- SPDY必须在`TLS上`运行，HTTP2可在`TCP`上直接使用，因为增加了HTTP1.1的Upgrade机制
- SPDY更加完善的协议商讨和确认流程
- SPDY更加完善的Server Push流程
- SPDY增加控制帧的种类，并对帧的格式考虑的更细致

HTTP1 和 HTTP2

- HTTP2是一个`二进制协议`，HTTP1是`超文本协议`，传输的内容都不是一样的
- HTTP2报头压缩，可以使用HPACK进行`头部压缩`，HTTP1则不论什么请求都会发送
- HTTP2`服务端推送`(Server push)，允许服务器预先将网页所需要的资源push到浏览器的内存当中
- HTTP2遵循`多路复用`，代替同一域名下的内容，只建立一次连接，HTTP1.x不是，对域名有6~8个连接限制
- HTTP2引入`二进制数据帧`和`流`的概念，其中帧对数据进行顺序标识，这样浏览器收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况，同样是因为有了序列，服务器就可以并行的传输数据，这就是流所做的事情。HTTP2对同一域名下所有请求都是基于流的，也就是说同一域名下不管访问多少文件，只建立一次连接

HTTP 3.0/QUIC

由于HTTP 2.0依赖于TCP，TCP有什么问题那HTTP2就会有什么问题。最主要的还是队头阻塞，在应用层的问题解决了，可是在TCP协议层的队头阻塞还没有解决。

TCP在丢包的时候会进行重传，前面有一个包没收到，就只能把后面的包放到缓冲区，应用层是无法取数据的，也就是说HTTP2的多路复用并行性对于TCP的丢失恢复机制不管用，因此丢失或重新排序的数据都会导致交互挂掉

为了解决这个问题，Google又发明了`QUIC协议`

并在2018年11月将QUIC正式改名为`HTTP 3.0`

特点:

- 在传输层直接干掉TCP，用`UDP`替代
- 实现了一套新的`拥塞控制算法`，彻底解决TCP中队头阻塞的问题
- 实现了类似TCP的`流量控制`、传输可靠性的功能。虽然UDP不提供可靠性的传输，但QUIC在UDP的基础之上增加了一层来保证数据可靠性传输。它提供了数据包重传、拥塞控制以及其他一些TCP中存在的特性
- 实现了`快速握手`功能。由于QUIC是基于UDP的，所以QUIC可以实现使用0-RTT或者1-RTT来建立连接，这意味着QUIC可以用最快的速度来发送和接收数据。
- 集成了TLS加密功能。目前QUIC使用的是TLS1.3