# 网络

### 1.五层模型

从上到下分别为：应用层、传输层、网络层、数据链路层、物理层。在发送消息时，消息从上到下进行打包，每一层会在上一层基础上加包，而接受消息时，从下到上进行解包，最终得到原始信息。

应用层主要面向互联网中的应用场景，比如网页、邮件、文件中心等等，它的代表协议有 http、smtp、pop3、ftp、DNS 等等

传输层主要面向传输过程，比如 TCP 协议是为了保证可靠的传输，而 UDP 协议则是一种无连接的广播，它们提供了不同的传输方式

网络层主要解决如何定位目标以及如何寻找最优路径的问题，比如 IP 等等

数据链路层的作用是将数据在一个子网（广播域）内有效传输，MAC地址、交换机都是属于该层的

物理层是要解决二进制数据到信号之间的互转问题，集线器、双绞线、同轴电缆等都是属于盖层的设备

### 2.常见的请求方法

http常见的请求方法有那些？

* GET，表示向服务器获取资源
* POST，表示向服务器提交信息，通常用于产生新的数据，比如注册
* PUT，表示希望修改服务器的数据，通常用于修改
* DELETE，表示希望删除服务器的数据
* OPTIONS，发生在跨域的预检请求中，表示客户端向服务器申请跨域提交
* TRACE，回显服务器收到的请求，主要用于测试和诊断
* CONNECT，用于建立连接管道，通常在代理场景中使用，网页中很少用到

GET 和 POST 的区别？

从http协议的角度来说，他们都是请求行中的第一个单词，本质上是没有区别的，之所以在开发中会产生区别是因为浏览器的默认行为造成的，主要有以下区别：

1. 浏览器在发送GET请求时，不会附带请求体
2. GET请求的传递信息量有限，适合传递少量数据，POST请求传递信息量时没有限制的
3. GET请求只能传递ASCII数据，遇到非ASCII数据需要进行编码；POST没有限制
4. 大部分GET请求传递的数据都在path参数总，不适合传递比较敏感的数据
5. 刷新页面时，若当前页面时POST请求得到的，则浏览器会提示用户是否重新提交，GET则不会提示
6. GET请求地址可以保存为书签

### 3.cookie/sessionStorage/localStorage 的区别

cookie、sessionStorage、localStorage 都是保存本地数据的方式

其中，cookie 兼容性较好，所有浏览器均支持。浏览器针对 cookie 会有一些默认行为，比如当响应头中出现set-cookie字段时，浏览器会自动保存 cookie 的值；再比如，浏览器发送请求时，会附带匹配的 cookie 到请求头中。这些默认行为，使得 cookie 长期以来担任着维持登录状态的责任。与此同时，也正是因为浏览器的默认行为，给了恶意攻击者可乘之机，CSRF 攻击就是一个典型的利用 cookie 的攻击方式。虽然 cookie 不断的改进，但前端仍然需要另一种更加安全的保存数据的方式

HTML5 新增了 sessionStorage 和 localStorage，前者用于保存会话级别的数据，后者用于更持久的保存数据。浏览器针对它们没有任何默认行为，这样一来，就把保存数据、读取数据的工作交给了前端开发者，这就让恶意攻击者难以针对登录状态进行攻击。  
cookie 的大小是有限制的，一般浏览器会限制同一个域下的 cookie 总量为 4M，而 sessionStorage 和 localStorage 则没有限制  
cookie 会与 domain、path 关联，而 sessionStorage 和 localStorage 只与 domain 关联

### 4. 对称加密、非对称加密、摘要的概念

密钥

密钥是一种参数，它是在明文转换为密文或将密文转换为明文的算法中输入的参数。密钥分为对称密钥与非对称密钥，分别应用在对称加密和非对称加密上。

对称加密

对称加密又叫做私钥加密，即信息的发送方和接收方使用同一个密钥去加密和解密数据。对称加密的特点是算法公开、加密和解密速度快，适合于对大数据量进行加密，常见的对称加密算法有 DES、3DES、TDEA、Blowfish、RC5 和 IDEA。

非对称加密

非对称加密也叫做公钥加密。非对称加密与对称加密相比，其安全性更好。对称加密的通信双方使用相同的密钥，如果一方的密钥遭泄露，那么整个通信就会被破解。而非对称加密使用一对密钥，即公钥和私钥，且二者成对出现。私钥被自己保存，不能对外泄露。公钥指的是公共的密钥，任何人都可以获得该密钥。用公钥或私钥中的任何一个进行加密，用另一个进行解密。

摘要

摘要算法又称哈希/散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用 16 进制的字符串表示）。算法不可逆。

### 5. 请阐述JWT的令牌格式

token 分为三段，分别是 header、payload、signature（签名）

其中，header 标识签名算法和令牌类型；payload 标识主体信息，包含令牌过期时间、发布时间、发行者、主体内容等；signature 是使用特定的算法对前面两部分进行加密，得到的加密结果。

token 有防篡改的特点，如果攻击者改动了前面两个部分，就会导致和第三部分对应不上，使得 token 失效。而攻击者不知道加密秘钥，因此又无法修改第三部分的值。

所以，在秘钥不被泄露的前提下，一个验证通过的 token 是值得被信任的。

### 6. 同源策略

浏览器安全策略，协议、主机、端口号、同源策略对 ajax 的跨域限制的最为凶狠，默认情况下，它不允许 ajax 访问跨域资源

### 7.跨域的方式

#### 代理

适用场景：生产环境下不跨域、但开发环境发生跨域

最常见的配置vuecli中的vue.config.js

#### CORS

CORS是基于http1.1的一种跨域解决方案，它的全称是**C**ross-**O**rigin **R**esource **S**haring，跨域资源共享。

它的总体思路是：**如果浏览器要跨域访问服务器的资源，需要获得服务器的允许**

针对不同的请求，CORS 规定了三种不同的交互模式，分别是：

* **简单请求**
* **需要预检的请求**
* **附带身份凭证的请求**

这三种模式从上到下层层递进，请求可以做的事越来越多，要求也越来越严格。

下面分别说明三种请求模式的具体规范。

##### 简单请求的判定

1. **请求方法属于下面的一种：**
   * get
   * post
   * head
2. **请求头仅包含安全的字段，常见的安全字段如下：**
   * Accept
   * Accept-Language
   * Content-Language
   * Content-Type
   * DPR
   * Downlink
   * Save-Data
   * Viewport-Width
   * Width
3. **请求头如果包含Content-Type，仅限下面的值之一：**
   * text/plain
   * multipart/form-data
   * application/x-www-form-urlencoded

如果以上三个条件同时满足，浏览器判定为简单请求。

##### 简单请求的交互规范

1.请求头中会自动添加Origin字段，告诉服务器，是哪个源地址在跨域请求

2.当服务器收到请求后，如果允许该请求跨域访问，需要在响应头中添加Access-Control-Allow-Origin字段

该字段的值可以是：

\*：表示我很开放，什么人我都允许访问

具体的源：比如http://my.com，表示我就允许你访问

##### 需要预检的请求

简单的请求对服务器的威胁不大，所以允许使用上述的简单交互即可完成。

但是，如果浏览器不认为这是一种简单请求，就会按照下面的流程进行：

1. **浏览器发送预检请求，询问服务器是否允许**
2. **服务器允许**
3. **浏览器发送真实请求**
4. **服务器完成真实的响应**

预检请求有以下特征：

* 请求方法为OPTIONS
* 没有请求体
* 请求头中包含
  + Origin：请求的源，和简单请求的含义一致
  + Access-Control-Request-Method：后续的真实请求将使用的请求方法
  + Access-Control-Request-Headers：后续的真实请求会改动的请求头

服务器收到预检请求后，可以检查预检请求中包含的信息，如果允许这样的请求，需要响应下面的消息格式

对于预检请求，不需要响应任何的消息体，只需要在响应头中添加：

* Access-Control-Allow-Origin：和简单请求一样，表示允许的源
* Access-Control-Allow-Methods：表示允许的后续真实的请求方法
* Access-Control-Allow-Headers：表示允许改动的请求头
* Access-Control-Max-Age：告诉浏览器，多少秒内，对于同样的请求源、方法、头，都不需要再发送预检请求了

然后浏览器发送真实请求，服务器响应真实请求

##### 附带身份凭证的请求

默认情况下，ajax 的跨域请求并不会附带 cookie，这样一来，某些需要权限的操作就无法进行

不过可以通过简单的配置就可以实现附带 cookie

// xhr  
var xhr = new XMLHttpRequest();  
xhr.withCredentials = true;  
  
// fetch api  
fetch(url, {  
 credentials: 'include',  
});

这样一来，该跨域的 ajax 请求就是一个*附带身份凭证的请求*

当一个请求需要附带 cookie 时，无论它是简单请求，还是预检请求，都会在请求头中添加cookie字段

而服务器响应时，需要明确告知客户端：服务器允许这样的凭据

告知的方式也非常的简单，只需要在响应头中添加：Access-Control-Allow-Credentials: true即可

对于一个附带身份凭证的请求，若服务器没有明确告知，浏览器仍然视为跨域被拒绝。

另外要特别注意的是：**对于附带身份凭证的请求，服务器不得设置 Access-Control-Allow-Origin 的值为\***。这就是为什么不推荐使用\*的原因

补充：

在跨域访问时，JS 只能拿到一些最基本的响应头，如：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma，如果要访问其他头，则需要服务器设置本响应头。

Access-Control-Expose-Headers头让服务器把允许浏览器访问的头放入白名单，例如：

Access-Control-Expose-Headers: authorization, a, b

这样 JS 就能够访问指定的响应头了。

#### JSONP

通过Script标签，只能Get请求

当需要跨域请求时，不使用AJAX，转而生成一个script元素去请求服务器，由于浏览器并不阻止script元素的请求，这样请求可以到达服务器。服务器拿到请求后，响应一段JS代码，这段代码实际上是一个函数调用，调用的是客户端预先生成好的函数，并把浏览器需要的数据作为参数传递到函数中，从而间接的把数据传递给客户端

/\*\*  
 \*   
 \* @param {\*  
 \* url,  
 \* success,  
 \* } options  
 \* 随机生成一个名  
 \* 把这个方法挂载到window上面去  
 \* 生成script标签  
 \* 把回调函数名拼接到script标签的url上  
 \* 把script标签加入到body中  
 \* 后端会返回这个函数的执行  
 \*/  
function jsonP(options) {  
 //随机生成一个函数名  
 let callback = "cb" + Math.random().toString().substring(2, 8);  
 window[callback] = options.success;  
 const script = document.createElement('script');  
 let url = options.url;  
 if (url.indexOf('?') > 0) {  
 //已经有参数  
 url = url + `&callback=${callback}`;  
 } else {  
 url = url + `?callback=${callback}`;  
 }  
 script.src = url;  
 script.id = callback;  
 document.body.appendChild(script);  
}

### 文件上传

文件上传的本质仍然是一个数据提交，无非就是数据量大一些而已

### 输入url后

1. 浏览器自动补全协议、端口
2. 浏览器自动完成url编码
3. 浏览器根据url地址查找本地缓存，根据缓存规则看是否命中缓存，若命中缓存则直接使用缓存，不再发出请求
4. 通过DNS解析找到服务器的IP地址
5. 浏览器向服务器发出建立TCP连接的申请，完成三次握手后，连接通道建立
6. 若使用了HTTPS协议，则还会进行SSL握手，建立加密信道。使用SSL握手时，会确定是否使用HTTP2
7. 浏览器决定要附带哪些cookie到请求头中
8. 浏览器自动设置好请求头、协议版本、cookie，发出GET请求
9. 服务器处理请求，进入后端处理流程。完成处理后，服务器响应一个HTTP报文给浏览器。
10. 浏览器根据使用的协议版本，以及Connection字段的约定，决定是否要保留TCP连接。
11. 浏览器根据响应状态码决定如何处理这一次响应
12. 浏览器根据响应头中的Content-Type字段识别响应类型，如果是text/html，则对响应体的内容进行HTML解析，否则做其他处理
13. 浏览器根据响应头的其他内容完成缓存、cookie的设置
14. 浏览器开始从上到下解析HTML，若遇到外部资源链接，则进一步请求资源
15. 解析过程中生成DOM树、CSSOM树，然后一边生成，一边把二者合并为渲染树（rendering tree），随后对渲染树中的每个节点计算位置和大小（reflow），最后把每个节点利用GPU绘制到屏幕（repaint）
16. 在解析过程中还会触发一系列的事件，当DOM树完成后会触发DOMContentLoaded事件，当所有资源加载完毕后会触发load事件

### Cookie和Session的区别是什么？

1. cookie 的数据保存在浏览器端；session 的数据保存在服务器
2. cookie 的存储空间有限；session 的存储空间不限
3. cookie 只能保存字符串；session 可以保存任何类型的数据
4. cookie 中的数据容易被获取；session 中的数据难以获取

### 三次握手

TCP 协议通过三次握手建立可靠的点对点连接，具体过程是：

首先服务器进入监听状态，然后即可处理连接

第一次握手：建立连接时，客户端发送 syn 包到服务器，并进入 SYN\_SENT 状态，等待服务器确认。在发送的包中还会包含一个初始序列号 seq。此次握手的含义是客户端希望与服务器建立连接。

第二次握手：服务器收到 syn 包，然后回应给客户端一个 SYN+ACK 包，此时服务器进入 SYN\_RCVD 状态。此次握手的含义是服务端回应客户端，表示已收到并同意客户端的连接请求。

第三次握手：客户端收到服务器的 SYN 包后，向服务器再次发送 ACK 包，并进入 ESTAB\_LISHED 状态。

最后，服务端收到客户端的 ACK 包，于是也进入 ESTAB\_LISHED 状态，至此，连接建立完成

### 四次挥手

1. Client 向 Server 发送 FIN 包，表示 Client 主动要关闭连接，然后进入 FIN*WAIT*1 状态，等待 Server 返回 ACK 包。此后 Client 不能再向 Server 发送数据，但能读取数据。
2. Server 收到 FIN 包后向 Client 发送 ACK 包，然后进入 CLOSE\_WAIT 状态，此后 Server 不能再读取数据，但可以继续向 Client 发送数据。
3. Client 收到 Server 返回的 ACK 包后进入 FIN*WAIT*2 状态，等待 Server 发送 FIN 包。
4. Server 完成数据的发送后，将 FIN 包发送给 Client，然后进入 LAST\_ACK 状态，等待 Client 返回 ACK 包，此后 Server 既不能读取数据，也不能发送数据。
5. Client 收到 FIN 包后向 Server 发送 ACK 包，然后进入 TIME\_WAIT 状态，接着等待足够长的时间（2MSL）以确保 Server 接收到 ACK 包，最后回到 CLOSED 状态，释放网络资源。
6. Server 收到 Client 返回的 ACK 包后便回到 CLOSED 状态，释放网络资源。

### CSRF攻击

CSRF 是跨站请求伪造，是一种挟制用户在当前已登录的Web应用上执行非本意的操作的攻击方法

它首先引导用户访问一个危险网站，当用户访问网站后，网站会发送请求到被攻击的站点，这次请求会携带用户的cookie发送，因此就利用了用户的身份信息完成攻击

防御方案：

1. 不使用cookie （兼容性差）
2. 为表单添加校验的 token 校验
3. cookie中使用sameSite字段 （容易挡住自己人）
4. 服务器检查 referer 字段

### XSS攻击

XSS 是指跨站脚本攻击。攻击者利用站点的漏洞，在表单提交时，在表单内容中加入一些恶意脚本，当其他正常用户浏览页面，而页面中刚好出现攻击者的恶意脚本时，脚本被执行，从而使得页面遭到破坏，或者用户信息被窃取。

要防范 XSS 攻击，需要在服务器端过滤脚本代码，将一些危险的元素和属性去掉或对元素进行HTML实体编码。

### 请简述域名解析过程

1. 查找本机hosts文件中是否有解析记录，如果有，直接使用
2. 查找本地域名服务器中是否有解析记录，如果有，直接使用
3. 查询根域名服务器，得到顶级域名服务器ip
4. 查询顶级域名服务器中是否有解析记录，如果有，直接使用
5. 根据顶级域名服务器反馈的ip，查询权限域名服务器，如果有解析记录，直接使用
6. 如果以上都找不到，域名解析失败

本机和域名服务器一般都会有高速缓存，它存在的目的是为了减少查询次数和时间

### 介绍下HTTPS中间人攻击

针对 HTTPS 攻击主要有 SSL 劫持攻击和 SSL 剥离攻击两种。

SSL 劫持攻击是指攻击者劫持了客户端和服务器之间的连接，将服务器的合法证书替换为伪造的证书，从而获取 客户端和服务器之间传递的信息。这种方式一般容易被用户发现，浏览器会明确的提示证书错误，但某些用户 安全意识不强，可能会点击继续浏览，从而达到攻击目的。

SSL 剥离攻击是指攻击者劫持了客户端和服务器之间的连接，攻击者保持自己和服务器之间的 HTTPS 连接，但 发送给客户端普通的 HTTP 连接，由于 HTTP 连接是明文传输的，即可获取客户端传输的所有明文数据。

### 介绍下HTTPS握手过程

1. 客户端请求服务器，并告诉服务器自身支持的加密算法以及密钥长度等信息
2. 服务器响应公钥和服务器证书
3. 客户端验证证书是否合法，然后生成一个会话密钥，并用服务器的公钥加密密钥，把加密的结果通过请求发送给服务器
4. 服务器使用私钥解密被加密的会话密钥并保存起来，然后使用会话密钥加密消息响应给客户端，表示自己已经准备就绪
5. 客户端使用会话密钥解密消息，知道了服务器已经准备就绪。
6. 后续客户端和服务器使用会话密钥加密信息传递消息

### 介绍下 http1.0、http1.1、http2.0 协议的区别？

http1.0

它的特点是每次请求和响应完毕后都会销毁 TCP 连接，同时规定前一个响应完成后才能发送下一个请求。这样做有两个问题：

1. 无法复用连接

每次请求都要创建新的 TCP 连接，完成三次握手和四次挥手，网络利用率低

1. 队头阻塞

如果前一个请求被某种原因阻塞了，会导致后续请求无法发送。

http1.1

1. 长连接

http1.1 允许在请求时增加请求头connection:keep-alive，这样便允许后续的客户端请求在一段时间内复用之前的 TCP 连接

1. 管道化

基于长连接的基础，管道化可以不等第一个请求响应继续发送后面的请求，但响应的顺序还是按照请求的顺序返回。

1. 缓存处理

新增响应头 cache-control，用于实现客户端缓存。

1. 断点传输

在上传/下载资源时，如果资源过大，将其分割为多个部分，分别上传/下载，如果遇到网络故障，可以从已经上传/下载好的地方继续请求，不用从头开始，提高效率

http2.0

1. 二进制分帧

将传输的消息分为更小的二进制帧，每帧有自己的标识序号，即便被随意打乱也能在另一端正确组装

1. 多路复用

基于二进制分帧，在同一域名下所有访问都是从同一个 tcp 连接中走，并且不再有队头阻塞问题，也无须遵守响应顺序

1. 头部压缩

http2.0 通过字典的形式，将头部中的常见信息替换为更少的字符，极大的减少了头部的数据量，从而实现更小的传输量

1. 服务器推

http2.0 允许服务器直接推送消息给客户端，无须客户端明确的请求

### 为什么 HTTP1.1 不能实现多路复用

HTTP/1.1 的传输单元是整个响应文本，因此接收方必须按序接收完所有的内容后才能接收下一个传输单元，否则就会造成混乱。而HTTP2.0的传输单元更小，是一个二进制帧，而且每个帧有针对所属流的编号，这样即便是不同的流交替传输，也可以很容易区分出每个帧是属于哪个流的。

### 简单讲解一下 http2 的多路复用

在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。 帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。 多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。

### http1.1 是如何复用 tcp 连接的

客户端请求服务器时，通过请求行告诉服务器使用的协议是 http1.1，同时在请求头中附带connection:keep-alive（为保持兼容），告诉服务器这是一个长连接，后续请求可以重复使用这一次的 TCP 连接。

这样做的好处是减少了三次握手和四次挥手的次数，一定程度上提升了网络利用率。但由于 http1.1 不支持多路复用，响应顺序必须按照请求顺序抵达客户端，不能真正实现并行传输，因此在 http2.0 出现之前，实际项目中往往把静态资源，比如图片，分发到不同域名下的资源服务器，以便实现真正的并行传输。

### webSocket 协议是什么

websocket 协议 HTML5 带来的新协议，相对于 http，它是一个持久连接的协议，它利用 http 协议完成握手，然后通过 TCP 连接通道发送消息，使用 websocket 协议可以实现服务器主动推送消息。

首先，客户端若要发起 websocket 连接，首先必须向服务器发送 http 请求以完成握手，请求行中的 path 需要使用ws:开头的地址，请求头中要分别加入upgrade、connection、Sec-WebSocket-Key、Sec-WebSocket-Version标记

然后，服务器收到请求后，发现这是一个 websocket 协议的握手请求，于是响应行中包含Switching Protocols，同时响应头中包含upgrade、connection、Sec-WebSocket-Accept标记

当客户端收到响应后即可完成握手，随后使用建立的 TCP 连接直接发送和接收消息。

### webSocket 与传统的 http 有什么优势

当页面中需要观察实时数据的变化（比如聊天、k 线图）时，过去我们往往使用两种方式完成

第一种是短轮询，即客户端每隔一段时间就向服务器发送消息，询问有没有新的数据

第二种是长轮询，发起一次请求询问服务器，服务器可以将该请求挂起，等到有新消息时再进行响应。响应后，客户端立即又发起一次请求，重复整个流程。

无论是哪一种方式，都暴露了 http 协议的弱点，即响应必须在请求之后发生，服务器是被动的，无法主动推送消息。而让客户端不断的发起请求又白白的占用了资源。

websocket 的出现就是为了解决这个问题，它利用 http 协议完成握手之后，就可以与服务器建立持久的连接，服务器可以在任何需要的时候，主动推送消息给客户端，这样占用的资源最少，同时实时性也最高。