# 网络基础

* 网络协议模型

|  |
| --- |
|  |

* 网络协议模型对应协议

[网络层次模型及各层对应协议\_NoobMaster--CISSP的博客-CSDN博客\_计算机网络层次结构模型和各层协议](https://blog.csdn.net/qq_35686185/article/details/104436049)

文章内容很多，包括协议，建议好好过一遍

[中继器、集线器、交换机、网桥和路由器分别相应于哪一层？ - slgkaifa - 博客园 (cnblogs.com)](https://www.cnblogs.com/slgkaifa/p/6736144.html)

|  |
| --- |
| DNS属于七层中的会话层 |
| 总结例如以下：  物理层：中继器、集线器  数据链路层：交换机、网桥  网络层：路由器 |

* 输入域名到显示网页发生了什么

[面试官：说说地址栏输入 URL 敲下回车后发生了什么？ · Issue #150 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/150)

|  |
| --- |
| 简单的分析，从输入 URL到回车后发生的行为如下：   * URL解析 * DNS 查询 * TCP 连接 * HTTP 请求 * 响应请求（HTTP响应） * 页面渲染（DOM、CSSOM、Render Tree、重排、重绘） |

* 网址的www

[网址中的www是什么意思? - 知乎 (zhihu.com)](https://www.zhihu.com/question/310534288)

|  |
| --- |
|  |

* DNS

[字节面试被虐后，是时候搞懂 DNS 了 - 掘金 (juejin.cn)](https://juejin.cn/post/6990344840181940261)

文章很有用，建议看原文

|  |
| --- |
| 什么是 DNS  DNS 即域名系统，全称是 **D**omain **N**ame **S**ystem。当我们在浏览器输入一个 URL 地址时，浏览器要向这个 URL 的主机名对应的服务器发送请求，就得知道服务器的 IP，对于浏览器来说，DNS 的作用就是将**主机名转换成 IP 地址**。下面是摘自《计算机网络：自顶向下方法》的概念：  DNS 是：   1. 一个由分层的 DNS 服务器实现的**分布式数据库** 2. 一个使得主机能够查询分布式数据库的**应用层协议**   也就是，DNS 是一个应用层协议，我们发送一个请求，其中包含我们要查询的主机名，它就会给我们返回这个主机名对应的 IP；  其次，DNS 是一个分布式数据库，整个 DNS 系统由分散在世界各地的很多台 DNS 服务器组成，每台 DNS 服务器上都保存了一些数据，这些数据可以让我们最终查到主机名对应的 IP。  **所以 DNS 的查询过程，说白了，就是去向这些 DNS 服务器询问，你知道这个主机名的 IP 是多少吗，不知道？那你知道去哪台 DNS 服务器上可以查到吗？直到查到我想要的 IP 为止。** |
| **递归查询**  IMG_256  **迭代查询。**  IMG_256  **看原文** |
| DNS 缓存  为了让我们更快的拿到想要的 IP，DNS 广泛使用了缓存技术。DNS 缓存的原理非常简单，在一个 DNS 查询的过程中，当某一台 DNS 服务器接收到一个 DNS 应答（例如，包含某主机名到 IP 地址的映射）时，它就能够将映射缓存到本地，下次查询就可以直接用缓存里的内容。当然，缓存并不是永久的，每一条映射记录都有一个对应的生存时间，一旦过了生存时间，这条记录就应该从缓存移出。  事实上，**有了缓存，大多数 DNS 查询都绕过了根 DNS 服务器**，需要向根 DNS 服务器发起查询的请求很少。 |

* cookie, sessionStorage 和 localStorage

[HTTP cookies - HTTP | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Cookies)

[(3条消息) cookie的保存与获取\_xiahuaxuanlan1113的博客-CSDN博客\_存cookie](https://blog.csdn.net/chenwei1113/article/details/79648650)

[Web Storage API - Web API 接口参考 | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Storage_API)

[使用 Web Storage API - Web API 接口参考 | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API)

[cookie和localStorage以及sessionStorage的区别 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/207106785)

[JS 退出登录，清除localStorage、sessionStorage - 简书 (jianshu.com)](https://www.jianshu.com/p/8e86bf912b0e)

[本地存储—— Cookie 到 Web Storage、IndexDB - 掘金 (juejin.cn)](https://juejin.cn/post/6844903917189332999)

|  |
| --- |
| 基本概念  cookie：主要用来保存登录信息，比如登录某个网站市场可以看到“记住密码”这就是通过cookie中存入一段辨别用户身份的数据来实现的  sessionStorage：会话，是可以将一部分数据在当前的会话中保存下来，刷新页面数据依旧存在。但是页面关闭时，sessionStorage中的数据就会被清空。  localStorage：是HTML5标准找那个新加入的技术，当然早在IE6时代就有一个userData的东西用于本地存储，当时考虑到浏览器的兼容性，更通用的方案是flash，如今localStorage被大多数浏览器所支持。localStorage中的键值对总是以字符串的形式存储。localStorage类似[sessionStorage](https://link.zhihu.com/?target=https://developer.mozilla.org/zh-CN/docs/Web/API/Window/sessionStorage" \t "https://zhuanlan.zhihu.com/p/_blank)，但其区别在于：存储在localStorage 的数据可以长期保留；  补充：同源：协议相同/域名相同/端口相同 |
| 区别  1.存储大小  cookie：一般不超过4k  sessionStorage：5M甚至更多  localStorage：5M甚至更多  2. 数据有效期  cookie:一般由服务器生成，可以设置失效时间；若没有设置时间，关闭浏览器cookie失效，如果设置了时间，cookie就会存储在硬盘中，过期失效  sessionStorage：仅在当前浏览器窗口关闭之前有效，关闭页面或者浏览器会被清除  localStorage：永久有效，窗口或者浏览器关闭也会一直保存，除非手动永久删除  3. 作用域  cookie：在所有同源窗口中都是共享的  sessionStorage：在同一个浏览器窗口是共享的（不同浏览器，即使是统一页面也不共享）  localStorage：在所有同源窗口中共享  4. 通信  cookie：cookie在浏览器和服务器之间来回传递，如果使用cookie保存过多数据会造成性能问题  sessionStorage：仅在客户端（浏览器）中保存，不参与服务器的通信  localStorage：仅在客户端（浏览器）中保存，不参与服务器的通信  5. 应用场景  cookie：判断用户是否登录过网站，以便实现下次自动登录或记住密码；保存事件信息  sessionStorage：敏感账号一次性登录，单页面用的较多  localStorage：用于长期登录，适于长期保存在本地的数据 |
| 操作 1、localStorage.setItem(key,value)  保存数据  2、localStorage.getItem(key)           获取数据  3、localStorage.removeItem(key)    删除数据  4、localStorage.clear();                     删除全部数据  同理 sessionStorage  **cookie的保存**  document.cookie = cookieName+"="+cookieValue+";expires="+d.toGMTString();  **cookie的获取**  var cookieStr = unescape(document.cookie);  var arr = cookieStr.split("; ");  var cookieValue = "";  for(var i=0;i<arr.length;i++){  var temp = arr[i].split("=");  if(temp[0]==cookieName){  cookieValue = temp[1];  break;  }  }  return cookieValue;  **cookie的删除，设置过期时间让他过期**  document.cookie=encodeURIComponent(cookieName)+“=; expires=” + new Date(); |

* cookie 参数

[HTTP cookies - HTTP | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Cookies" \l "%E8%B7%9F%E8%B8%AA%E5%92%8C%E9%9A%90%E7%A7%81)

[Set-Cookie - HTTP | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Set-Cookie)

|  |
| --- |
| HTTP Cookie（也叫 Web Cookie 或浏览器 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，它用于告知服务端两个请求是否来自同一浏览器，如保持用户的登录状态。 |
| 创建cookie：  当服务器收到 HTTP 请求时，服务器可以在响应头里面添加一个 [Set-Cookie](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Set-Cookie) 选项。浏览器收到响应后通常会保存下 Cookie，之后对该服务器每一次请求中都通过  [Cookie](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Cookie) 请求头部将 Cookie 信息发送给服务器。 |
| Cookie 的生命周期可以通过两种方式定义：   * 会话期 Cookie 是最简单的 Cookie：浏览器关闭之后它会被自动删除，也就是说它仅在会话期内有效。会话期 Cookie 不需要指定过期时间（Expires）或者有效期（Max-Age）。需要注意的是，有些浏览器提供了会话恢复功能，这种情况下即使关闭了浏览器，会话期 Cookie 也会被保留下来，就好像浏览器从来没有关闭一样，这会导致 Cookie 的生命周期无限期延长。 * 持久性 Cookie 的生命周期取决于过期时间（Expires）或有效期（Max-Age）指定的一段时间。 |
| 标记为 Secure 的 Cookie 只应通过被 HTTPS 协议加密过的请求发送给服务端，因此可以预防 [man-in-the-middle](https://developer.mozilla.org/zh-CN/docs/Glossary/MitM) 攻击者的攻击。但即便设置了 Secure 标记，敏感信息也不应该通过 Cookie 传输，因为 Cookie 有其固有的不安全性，Secure 标记也无法提供确实的安全保障，例如，可以访问客户端硬盘的人可以读取它。  JavaScript [Document.cookie](https://developer.mozilla.org/zh-CN/docs/Web/API/Document/cookie) API 无法访问带有 HttpOnly 属性的 cookie；此类 Cookie 仅作用于服务器。例如，持久化服务器端会话的 Cookie 不需要对 JavaScript 可用，而应具有 HttpOnly 属性。此预防措施有助于缓解[跨站点脚本（XSS） (en-US)](https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks" \o "Currently only available in English (US))攻击。  Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly |
| Domain 和 Path 标识定义了 Cookie 的作用域：即允许 Cookie 应该发送给哪些 URL。  **Domain 属性**  Domain 指定了哪些主机可以接受 Cookie。如果不指定，默认为 [origin](https://developer.mozilla.org/zh-CN/docs/Glossary/Origin)，**不包含子域名**。如果指定了Domain，则一般包含子域名。因此，指定 Domain 比省略它的限制要少。但是，当子域需要共享有关用户的信息时，这可能会有所帮助。  例如，如果设置 Domain=mozilla.org，则 Cookie 也包含在子域名中（如developer.mozilla.org）。  **Path 属性**  Path 标识指定了主机下的哪些路径可以接受 Cookie（该 URL 路径必须存在于请求 URL 中）。以字符 %x2F ("/") 作为路径分隔符，子路径也会被匹配。  例如，设置 Path=/docs，则以下地址都会匹配：   * /docs * /docs/Web/ * /docs/Web/HTTP |
|  |

* cookie samesite

[Cookie 的 SameSite 属性 - 阮一峰的网络日志 (ruanyifeng.com)](https://www.ruanyifeng.com/blog/2019/09/cookie-samesite.html)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SameSite Cookie 允许服务器要求某个 cookie 在跨站请求时不会被发送，（其中  [Site (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/Site" \o "Currently only available in English (US)) 由可注册域定义），从而可以阻止跨站请求伪造攻击（[CSRF](https://developer.mozilla.org/zh-CN/docs/Glossary/CSRF)）。  Set-Cookie: key=value; SameSite=Strict  Copy to Clipboard  SameSite 可以有下面三种值：   * **None。**浏览器会在同站请求、跨站请求下继续发送 cookies，不区分大小写。 * Strict**。**浏览器将只在访问相同站点时发送 cookie。（在原有 Cookies 的限制条件上的加强，如上文 “Cookie 的作用域” 所述） * Lax**。**与 Strict类似，但用户从外部站点导航至 URL 时（例如通过链接）除外。 在新版本浏览器中，为默认选项，Same-site cookies 将会为一些跨站子请求保留，如图片加载或者 frames 的调用，但只有当用户从外部站点导航到 URL 时才会发送。如 link 链接。   2.1 Strict  Strict最为严格，完全禁止第三方 Cookie，跨站点时，任何情况下都不会发送 Cookie。换言之，只有当前网页的 URL 与请求目标一致，才会带上 Cookie。  Set-Cookie: CookieName=CookieValue; SameSite=Strict;  这个规则过于严格，可能造成非常不好的用户体验。比如，当前网页有一个 GitHub 链接，用户点击跳转就不会带有 GitHub 的 Cookie，跳转过去总是未登陆状态。**（即如果GitHub后台服务器set-cookie设置了samesite）**  2.2 Lax  Lax规则稍稍放宽，大多数情况也是不发送第三方 Cookie，但是导航到目标网址的 Get 请求除外。  Set-Cookie: CookieName=CookieValue; SameSite=Lax;  导航到目标网址的 GET 请求，只包括三种情况：链接，预加载请求，GET 表单。详见下表。   | 请求类型 | 示例 | 正常情况 | Lax | | --- | --- | --- | --- | | 链接 | <a href="..."></a> | 发送 Cookie | 发送 Cookie | | 预加载 | <link rel="prerender" href="..."/> | 发送 Cookie | 发送 Cookie | | GET 表单 | <form method="GET" action="..."> | 发送 Cookie | 发送 Cookie | | POST 表单 | <form method="POST" action="..."> | 发送 Cookie | 不发送 | | iframe | <iframe src="..."></iframe> | 发送 Cookie | 不发送 | | AJAX | $.get("...") | 发送 Cookie | 不发送 | | Image | <img src="..."> | 发送 Cookie | 不发送 |   设置了Strict或Lax以后，基本就杜绝了 CSRF 攻击。当然，前提是用户浏览器支持 SameSite 属性。 |

* 同源策略

[localstorage的跨域存储方案 - 简书 (jianshu.com)](https://www.jianshu.com/p/e86d92aeae69)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| html5标准中一个亮点就是提供了浏览器本地存储的功能。方式有两种：localStorage和 sessionStorage。 相对于cookie，他们具有存储空间大的特点，一般可以存储5M左右，而cookie一般只有4k。  另外，不同浏览器无法共享localStorage和sessionStorage中的信息。同一浏览器的相同域名和端口的不同页面间可以共享相同的 localStorage，但是不同页面间无法共享sessionStorage的信息。这里需要注意的是，页面仅指顶级窗口，如果一个页面包含多个iframe且他们属于同源页面，那么他们之间是可以共享sessionStorage的。在实际开发过程中，遇到的最多的问题就是localStorage的同源策略问题。为了了解这个问题，我们先得清楚什么是同源策略。同源策略（[same-origin policy](https://link.jianshu.com?t=https://en.wikipedia.org/wiki/Same-origin_policy" \t "_blank)）是浏览器执行的一种安全措施，目的是为了保证用户信息的安全，防止恶意的网站窃取数据。浏览器的同源策略具体如下：   | **URL** | **说明** | **是否允许通信** | | --- | --- | --- | | [http://www.a.com/a.js](https://link.jianshu.com?t=http://www.a.com/a.js" \t "_blank)  [http://www.a.com/b.js](https://link.jianshu.com?t=http://www.a.com/b.js" \t "_blank) | 同一域名下 | 允许 | | [http://www.a.com/lab/a.js](https://link.jianshu.com?t=http://www.a.com/lab/a.js" \t "_blank)  [http://www.a.com/script/b.js](https://link.jianshu.com?t=http://www.a.com/script/b.js" \t "_blank) | 同一域名下不同文件夹 | 允许 | | [http://www.a.com:8000/a.js](https://link.jianshu.com?t=http://www.a.com:8000/a.js" \t "_blank)  [http://www.a.com/b.js](https://link.jianshu.com?t=http://www.a.com/b.js" \t "_blank) | 同一域名，不同端口 | 不允许 | | [http://www.a.com/a.js](https://link.jianshu.com?t=http://www.a.com/a.js" \t "_blank)  [https://www.a.com/b.js](https://link.jianshu.com?t=https://www.a.com/b.js" \t "_blank) | 同一域名，不同协议 | 不允许 | | [http://www.a.com/a.js](https://link.jianshu.com?t=http://www.a.com/a.js" \t "_blank)  [http://70.32.92.74/b.js](https://link.jianshu.com?t=http://70.32.92.74/b.js" \t "_blank) | 域名和域名对应ip | 不允许 | | [http://www.a.com/a.js](https://link.jianshu.com?t=http://www.a.com/a.js" \t "_blank)  [http://script.a.com/b.js](https://link.jianshu.com?t=http://script.a.com/b.js" \t "_blank) | 主域相同，子域不同 | 不允许 | | [http://www.a.com/a.js](https://link.jianshu.com?t=http://www.a.com/a.js" \t "_blank)  [http://file.a.com/b.js](https://link.jianshu.com?t=http://file.a.com/b.js" \t "_blank) | 同一域名，不同二级域名（同上） | 不允许（cookie这种情况下也不允许访问） | | [http://www.cnblogs.com/a.js](https://link.jianshu.com?t=http://www.cnblogs.com/a.js" \t "_blank)  [http://www.a.com/b.js](https://link.jianshu.com?t=http://www.a.com/b.js" \t "_blank) | 不同域名 | 不允许 | |
| 只要不同源就不能共享localStorage的数据。但是在实际开发中又时常会遇到这样的需求，那我们该如何解决呢？  目前广泛采用的是postMessage和iframe相结合的方法。postMessage(data,origin)方法允许来自不同源的脚本采用异步方式进行通信，可以实现跨文本档、多窗口、跨域消息传递。接受两个参数：   * data：要传递的数据，[HTML5](https://link.jianshu.com?t=http://lib.csdn.net/base/html5" \t "_blank)规范中提到该参数可以是[JavaScript](https://link.jianshu.com?t=http://lib.csdn.net/base/javascript" \t "_blank)的任意基本类型或可复制的对象，然而并不是所有浏览器支持任意类型的参数，部分浏览器只能处理字符串参数，所以在传递参数时需要使用JSON.stringify()方法对对象参数序列化。 * origin：字符串参数，指明目标窗口的源，协议+主机+端口号[+URL]，URL会被忽略，所以可以不写，只是为了安全考虑，postMessage()方法只会将message传递给指定窗口，当然也可以将参数设置为"\*"，这样可以传递给任意窗口，如果要指定和当前窗口同源的话设置为"/"。   在safari浏览器下，用url传值的方法来实现跨域存储功能。 |

* 跨域

[计算机网络（一）——跨域 - 掘金 (juejin.cn)](https://juejin.cn/post/6949824600284856350)

[【前端工程师面试宝典】学习说明\_互联网校招面试真题面经汇总\_牛客网 (nowcoder.com)](https://www.nowcoder.com/tutorial/96/6b04165463a84068a931f70c14cdf0c4)

[面试官：跨域是什么？Vue项目中你是如何解决跨域的呢？ · Issue #30 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/30)

[9种常见的前端跨域解决方案（详解） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/81809258)

|  |
| --- |
| 1. 同源策略 2. 保护——cookie，防CSRF 3. 方案——jsonp手写；   CORS属性，简单请求（Origin，Access-Control-Allow-Origin，Access-Control-Allow-Credentials：可选），非简单请求（预检OPTIONS，Access-Control-Request-Method，Access-Control-Max-Age）PUT或DELETE，或者Content-Type字段的类型是application/json；  nginx代理 [nginx解决跨域的原理分析-Nginx-PHP中文网](https://www.php.cn/nginx/451679.html) |

|  |
| --- |
| 但是有三个标签是允许跨域加载资源：   * <img src='xxx'> * <link href='xxx'> * <script src='xxx'>   跨域并不是请求发不出去，请求能发出去，服务端能收到请求并正常返回结果，只是结果被浏览器拦截了。  跨域解决方案  解决方案有jsonp、cors、postMessage、websocket、Node中间件代理(两次跨域)、nginx反向代理、window.name + iframe、location.hash + iframe、document.domain + iframe，CORS支持所有类型的HTTP请求，是跨域HTTP请求的根本解决方案，JSONP只支持GET请求，JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。不管是Node中间件代理还是nginx反向代理，主要是通过同源策略对服务器不加限制。日常工作中，用得比较多的跨域方案是cors和nginx反向代理 |

* JSONP

|  |
| --- |
| <script>  var script = document.createElement('script');  script.type = 'text/javascript';  // 传参一个回调函数名给后端，方便后端返回时执行这个在前端定义的回调函数  script.src = 'http://www.domain2.com:8080/login?user=admin&callback=handleCallback';  document.head.appendChild(script);  // 回调执行函数  function handleCallback(res) {  alert(JSON.stringify(res));  }  </script> |
| 后端node.js代码：  var querystring = require('querystring');  var http = require('http');  var server = http.createServer();  server.on('request', function(req, res) {  var params = querystring.parse(req.url.split('?')[1]);  var fn = params.callback;  // jsonp返回设置  res.writeHead(200, { 'Content-Type': 'text/javascript' });  res.write(fn + '(' + JSON.stringify(params) + ')');  res.end();  });  server.listen('8080');  console.log('Server is running at port 8080...');  jsonp的缺点：只能发送get一种请求。 |

* CORS

[跨源资源共享（CORS） - HTTP | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/CORS)

|  |
| --- |
| CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。 它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。 CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。  浏览器将CORS跨域请求分为简单请求和非简单请求。  只要同时满足一下两个条件，就属于简单请求。 **(1)使用下列方法之一：**  **head**  **get**  **post**  **(2)请求的Header是**  **Accept**  **Accept-Language**  **Content-Language**  **Content-Type: 只限于三个值：application/x-www-form-urlencoded、multipart/form-data、text/plain**  不同时满足上面的两个条件，就属于非简单请求。浏览器对这两种的处理，是不一样的。  简单请求  对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段。  GET /cors HTTP/1.1  Origin: http://api.bob.com  Host: api.alice.com  Accept-Language: en-US  Connection: keep-alive  User-Agent: Mozilla/5.0...  上面的头信息中，Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。  CORS请求设置的响应头字段（即服务器端发来的），都以 Access-Control-开头:  **1）Access-Control-Allow-Origin：必选**  它的值要么是请求时Origin字段的值，要么是一个\*，表示接受任意域名的请求。  **2）Access-Control-Allow-Credentials：可选**  它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。  **3）Access-Control-Expose-Headers：可选**  CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。上面的例子指定，getResponseHeader(‘FooBar’)可以返回FooBar字段的值。  非简单请求  非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。  预检请求  预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。请求头信息里面，关键字段是Origin，表示请求来自哪个源。除了Origin字段，"预检"请求的头信息包括两个特殊字段。  OPTIONS /cors HTTP/1.1  Origin: http://api.bob.com  Access-Control-Request-Method: PUT  Access-Control-Request-Headers: X-Custom-Header  Host: api.alice.com  Accept-Language: en-US  Connection: keep-alive  User-Agent: Mozilla/5.0..  1**）Access-Control-Request-Method：必选**  用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。  **2）Access-Control-Request-Headers：可选**  该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。  预检请求的回应（即服务器端）  服务器收到"预检"请求以后，检查了Origin、Access-Control-Request-Method和Access-Control-Request-Headers字段以后，确认允许跨源请求，就可以做出回应。  HTTP回应中，除了关键的是Access-Control-Allow-Origin字段，其他CORS相关字段如下：  **1）Access-Control-Allow-Methods：必选**  它的值是逗号分隔的一个字符串，表明服务器支持的所有跨域请求的方法。注意，返回的是所有支持的方法，而不单是浏览器请求的那个方法。这是为了避免多次"预检"请求。  **2）Access-Control-Allow-Headers**  如果浏览器请求包括Access-Control-Request-Headers字段，则Access-Control-Allow-Headers字段是必需的。它也是一个逗号分隔的字符串，表明服务器支持的所有头信息字段，不限于浏览器在"预检"中请求的字段。  **3）Access-Control-Allow-Credentials：可选**  该字段与简单请求时的含义相同。  **4）Access-Control-Max-Age：可选**  用来指定本次预检请求的有效期，单位为秒。 |

# HTTP

* http

[面试官：什么是HTTP? HTTP 和 HTTPS 的区别? · Issue #134 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/134)

[面试官：说说 HTTP 常见的请求头有哪些? 作用？ · Issue #149 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/149)

[HTTP协议的Keep-Alive 模式 - 简书 (jianshu.com)](https://www.jianshu.com/p/49551bda6619)

[面试官：说说 HTTP1.0/1.1/2.0 的区别? · Issue #143 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/143)

|  |
| --- |
| HTTP1.0：   * 浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接   HTTP1.1：   * 引入了持久连接，即TCP连接默认不关闭，可以被多个请求复用（Connection: keep-alive） * 在同一个TCP连接里面，客户端可以同时发送多个请求 * 虽然允许复用TCP连接，但是同一个TCP连接里面，所有的数据通信是按次序进行的，服务器只有处理完一个请求，才会接着处理下一个请求。如果前面的处理特别慢，后面就会有许多请求排队等着。”队头堵塞” * 新增了一些请求方法 * 新增了一些请求头和响应头   HTTP2.0：   * 采用二进制格式而非文本格式   每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装，这也是多路复用同时发送数据的实现条件   * 完全多路复用，而非有序并阻塞的、只需一个连接即可实现并行   HTTP/2 复用TCP连接，在一个连接里，客户端和浏览器都可以****同时****发送多个请求或回应，而且不用按照顺序一一对应，这样就避免了”队头堵塞”   * 使用报头压缩，降低开销 * 服务器推送 |

* https 非对称加密

[面试官：为什么说HTTPS比HTTP安全? HTTPS是如何保证安全的？ · Issue #135 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/135)

[简述https的几种加密方式 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1464189" \l ":~:text=%E7%AE%80%E8%BF%B0%EE%80%80https%EE%80%81%E7%9A%84%E5%87%A0%E7%A7%8D%EE%80%80%E5%8A%A0%E5%AF%86%EE%80%81%E6%96%B9%E5%BC%8F%E3%80%82%E6%98%8E%E6%96%87<->%E5%AF%86%E9%92%A5<->%E5%AF%86%E6%96%87,%E5%B8%B8%E8%A7%81%E7%9A%84%E5%AF%B9%E7%A7%B0%EE%80%80%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95%EE%80%81%EF%BC%9ADES%E3%80%813DES%E3%80%81AES%E3%80%81Blowfish%E3%80%81IDEA%E3%80%81RC5%E3%80%81RC6%EF%BC%9B%E8%A7%A3%E5%86%B3%E7%9A%84%E6%96%B9%E6%B3%95%E6%98%AF%E5%B0%86%E5%AF%B9%E7%A7%B0%EE%80%80%E5%8A%A0%E5%AF%86%EE%80%81%E7%9A%84%E5%AF%86%E9%92%A5%E4%BD%BF%E7%94%A8%E9%9D%9E%E5%AF%B9%E7%A7%B0%EE%80%80%E5%8A%A0%E5%AF%86%EE%80%81%E7%9A%84%E5%85%AC%E9%92%A5%E8%BF%9B%E8%A1%8C%EE%80%80%E5%8A%A0%E5%AF%86%EE%80%81%EF%BC%8C%E7%84%B6%E5%90%8E%E5%8F%91%E9%80%81%E5%87%BA%E5%8E%BB%EF%BC%8C%E6%8E%A5%E6%94%B6%E6%96%B9%E4%BD%BF%E7%94%A8%E7%A7%81%E9%92%A5%E8%BF%9B%E8%A1%8C%E8%A7%A3%E5%AF%86%E5%BE%97%E5%88%B0%E5%AF%B9%E7%A7%B0%EE%80%80%E5%8A%A0%E5%AF%86%EE%80%81%E7%9A%84%E5%AF%86%E9%92%A5 )

|  |
| --- |
| **混合加密**  而HTTPS采用非对称加密解决秘钥交换的问题  具体做法是发送密文的一方使用对方的公钥进行加密处理“对称的密钥”，然后对方用自己的私钥解密拿到“对称的密钥”  [IMG_256](https://camo.githubusercontent.com/31677ec92cfdaad700f98b5030cdfb79bb6632bd471c294f1ad30d99d3151e6a/68747470733a2f2f7374617469632e7675652d6a732e636f6d2f66333735663239302d623266662d313165622d383566362d3666616337376330633962332e706e67)   * 常见的对称加密算法：DES、3DES、AES、Blowfish、IDEA、RC5、RC6； * 常见的非对称加密算法：RSA、Elgamal、背包算法、Rabin。 * RSA这种加密算法应用非常广泛，如SSH、HTTPS、TLS、电子证书、电子签名、电子身份证等。 |
| **摘要算法**  实现完整性的手段主要是摘要算法，也就是常说的散列函数、哈希函数  可以理解成一种特殊的压缩算法，它能够把任意长度的数据“压缩”成固定长度、而且独一无二的“摘要”字符串，就好像是给这段数据生成了一个数字“指纹”  [IMG_256](https://camo.githubusercontent.com/16ceb478a4cdc255009ba6912fff85ef91ddf5711b279527344a12bef47619e1/68747470733a2f2f7374617469632e7675652d6a732e636f6d2f30323337393065302d623330302d313165622d616239302d6439616538313462323430642e706e67) |
| **数字签名**  数字签名能确定消息确实是由发送方签名并发出来的，因为别人假冒不了发送方的签名  原理其实很简单，就是用私钥加密，公钥解密  签名和公钥一样完全公开，任何人都可以获取。但这个签名只有用私钥对应的公钥才能解开，拿到摘要后，再比对原文验证完整性，就可以像签署文件一样证明消息确实是你发的  [IMG_256](https://camo.githubusercontent.com/8995c606939e79af9d26a660a10968f455796af32f432c1287dc5cd9d2a391d0/68747470733a2f2f7374617469632e7675652d6a732e636f6d2f32316161363838302d623330302d313165622d383566362d3666616337376330633962332e706e67) |
| **CA验证机构**  数字证书认证机构处于客户端与服务器双方都可信赖的第三方机构的立场  CA 对公钥的签名认证要求包括序列号、用途、颁发者、有效时间等等，把这些打成一个包再签名，完整地证明公钥关联的各种信息，形成“数字证书”  流程如下图：  [IMG_256](https://camo.githubusercontent.com/0c515b43bb7352dfd4a341c467a2933838fe7bb5f760ba793146155da09c186b/68747470733a2f2f7374617469632e7675652d6a732e636f6d2f33393536343861302d623330302d313165622d383566362d3666616337376330633962332e706e67) |

* HTTP请求报文和响应报文结构

[一篇让你彻底了解http请求报文和响应报文的结构 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/346408612)

|  |
| --- |
|  |
| **请求报文**  HTTP的请求报文由四部分组成（请求行+请求头部+空行+请求体）：  IMG_256**真实数据**  **⑤是报文体**，它将一个页面表单中的组件值通过param1=value1¶m2=value2的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求URL也可以通过类似于“/chapter15/user.html? param1=value1¶m2=value2”的方式传递请求参数。IMG_257 |
| **响应报文**  IMG_256HTTP的响应报文也由四部分组成（ 响应行+响应头+空行+响应体）：  IMG_257**真实数据** |

* HTTP请求方法

[HTTP 请求方法 | 菜鸟教程 (runoob.com)](https://www.runoob.com/http/http-methods.html)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 根据 HTTP 标准，HTTP 请求可以使用多种请求方法。  HTTP1.0 定义了三种请求方法： GET, POST 和 HEAD 方法。  HTTP1.1 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方法。   |  |  |  | | --- | --- | --- | | **序号** | **方法** | **描述** | | 1 | GET | 请求指定的页面信息，并返回实体主体。 | | 2 | HEAD | 类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头 | | 3 | POST | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。 | | 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 | | 5 | DELETE | 请求服务器删除指定的页面。 | | 6 | CONNECT | HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。 | | 7 | OPTIONS | 允许客户端查看服务器的性能。 | | 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断。 | | 9 | PATCH | 是对 PUT 方法的补充，用来对已知资源进行局部更新 。 | |

* get 和 post

[面试官：说一下 GET 和 POST 的区别？ · Issue #145 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/145)

|  |
| --- |
| **GET** GET方法请求一个指定资源的表示形式，使用GET的请求应该只被用于获取数据 **POST** POST方法用于将实体提交到指定的资源，通常导致在服务器上的状态变化或****副作用****  本质上都是TCP链接，并无差别  但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中会体现出一些区别  从w3schools得到的标准答案的区别如下：   * GET在浏览器回退时是无害的，而POST会再次提交请求。 * GET产生的URL地址可以被Bookmark，而POST不可以。 * GET请求会被浏览器主动cache，而POST不会，除非手动设置。 * GET请求只能进行url编码，而POST支持多种编码方式。 * GET请求参数会被完整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中传送的参数是有长度限制的，而POST没有。 * 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。 * GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。 * GET参数通过URL传递，POST放在Request body中   **数据包**  对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）  对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok  并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次 |

* tcp 和 udp

[面试官：如何理解UDP 和 TCP? 区别? 应用场景? · Issue #136 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/136)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| UDP与TCP两者的都位于传输层  两者区别如下表所示：   |  | **TCP** | **UDP** | | --- | --- | --- | | 可靠性 | 可靠 | 不可靠 | | 连接性 | 面向连接 | 无连接 | | 报文 | 面向字节流 | 面向报文 | | 效率 | 传输效率低 | 传输效率高 | | 双共性 | 全双工 | 一对一、一对多、多对一、多对多 | | 流量控制 | 滑动窗口 | 无 | | 拥塞控制 | 慢开始、拥塞避免、快重传、快恢复 | 无 | | 传输效率 | 慢 | 快 |   TCP 是面向连接的协议，建立连接3次握手、断开连接四次挥手，UDP是面向无连接，数据传输前后不连接连接，发送端只负责将数据发送到网络，接收端从消息队列读取  TCP 提供可靠的服务，传输过程采用流量控制、编号与确认、计时器等手段确保数据无差错，不丢失。UDP 则尽可能传递数据，但不保证传递交付给对方  TCP 面向字节流，将应用层报文看成一串无结构的字节流，分解为多个TCP报文段传输后，在目的站重新装配。UDP协议面向报文，不拆分应用层报文，只保留报文边界，一次发送一个报文，接收方去除报文首部后，原封不动将报文交给上层应用  TCP 只能点对点全双工通信。UDP 支持一对一、一对多、多对一和多对多的交互通信 |

* 三次握手四次挥手

[面试官：说说TCP为什么需要三次握手和四次挥手？ · Issue #151 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/151)

[四次挥手 - 简书 (jianshu.com)](https://www.jianshu.com/p/cd801d1b3147)

|  |
| --- |
| 三次握手（Three-way Handshake）其实就是指建立一个TCP连接时，需要客户端和服务器总共发送3个包  主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备  过程如下：   * 第一次握手：客户端给服务端发一个 SYN 报文，并**指明客户端的初始化序列号 ISN(c)**，此时客户端处于 SYN\_SENT 状态 * 第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，为了确认客户端的 SYN，将客户端的 ISN+1作为ACK的值，此时服务器处于 SYN\_RCVD 的状态 * 第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，值为服务器的ISN+1。此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立起了连接   上述每一次握手的作用如下：   * 第一次握手：客户端发送网络包，服务端收到了 这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。 * 第二次握手：服务端发包，客户端收到了 这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常 * 第三次握手：客户端发包，服务端收到了。 这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常   通过三次握手，就能确定双方的接收和发送能力是正常的。之后就可以正常通信了。  **为什么不是两次握手?**  如果是两次握手，发送端可以确定自己发送的信息能对方能收到，也能确定对方发的包自己能收到，但接收端只能确定对方发的包自己能收到 无法确定自己发的包对方能收到  并且两次握手的话, 客户端有可能因为网络阻塞等原因会发送多个请求报文，延时到达的请求又会与服务器建立连接，浪费掉许多服务器的资源  [IMG_256](https://camo.githubusercontent.com/df92e0deefb619759aa435aca99720cf4113a5ba705eb09f1636f91c89d3eb1e/68747470733a2f2f7374617469632e7675652d6a732e636f6d2f66623438396663302d626562392d313165622d383566362d3666616337376330633962332e706e67)  **很直观的一件事就是 第二次握手后，如果客户端接收能力不正常，服务端也无从得知，如果之后第一次的操作是服务端推送，那么就不行了。** |
| tcp终止一个连接，需要经过四次挥手  过程如下：   * 第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于 FIN\_WAIT1 状态，停止发送数据，等待服务端的确认 * 第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 CLOSE\_WAIT状态 * 第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 LAST\_ACK 的状态 * 第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1 作为自己 ACK 报文的序列号值，此时客户端处于 TIME\_WAIT状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态     **四次挥手原因**  服务端在收到客户端断开连接Fin报文后，并不会立即关闭连接，而是先发送一个ACK包先告诉客户端收到关闭连接的请求，只有当服务器的所有报文发送完毕之后，才发送FIN报文断开连接，因此需要四次挥手 |

* http 状态码

[面试官：说说HTTP 常见的状态码有哪些，适用场景？ · Issue #144 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/144)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码第一位数字决定了不同的响应状态，有如下：   * 1 表示消息 * 2 表示成功 * 3 表示重定向 * 4 表示请求错误 * 5 表示服务器错误   **1xx**  代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束  常见的有：  100（客户端继续发送请求，这是临时响应）：这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应  101：服务器根据客户端的请求切换协议，主要用于websocket或http2升级 | **2xx**  代表请求已成功被服务器接收、理解、并接受  常见的有：  200（成功）：请求已成功，请求所希望的响应头或数据体将随此响应返回  201（已创建）：请求成功并且服务器创建了新的资源  202（已创建）：服务器已经接收请求，但尚未处理  203（非授权信息）：服务器已成功处理请求，但返回的信息可能来自另一来源  204（无内容）：服务器成功处理请求，但没有返回任何内容  205（重置内容）：服务器成功处理请求，但没有返回任何内容  206（部分内容）：服务器成功处理了部分请求 | **3xx**  表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向  常见的有：  300（多种选择）：针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择  301（永久移动）：请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置  302（临时移动）： 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求  303（查看其他位置）：请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码  305 （使用代理）： 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理  307 （临时重定向）： 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求 | **4xx**  代表了客户端看起来可能发生了错误，妨碍了服务器的处理  常见的有：  400（错误请求）： 服务器不理解请求的语法  401（未授权）： 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。  403（禁止）： 服务器拒绝请求  404（未找到）： 服务器找不到请求的网页  405（方法禁用）： 禁用请求中指定的方法  406（不接受）： 无法使用请求的内容特性响应请求的网页  407（需要代理授权）： 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理  408（请求超时）： 服务器等候请求时发生超时 | **5xx**  表示服务器无法完成明显有效的请求。这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生  常见的有：  500（服务器内部错误）：服务器遇到错误，无法完成请求  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（错误网关）： 服务器作为网关或代理，从上游服务器收到无效响应  503（服务不可用）： 服务器目前无法使用（由于超载或停机维护）  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 |

* 缓存逻辑 、etag

[彻底弄懂强缓存与协商缓存 - 简书 (jianshu.com)](https://www.jianshu.com/p/9c95db596df5)

[http面试必会的：强制缓存和协商缓存 - 掘金 (juejin.cn)](https://juejin.cn/post/6844903838768431118)

|  |
| --- |
| 强缓存：  当浏览器去请求某个文件的时候，服务端就在respone header里面对该文件做了缓存配置。缓存的时间、缓存类型都由服务端控制，具体表现为： respone header 的cache-control，常见的设置是max-age public private no-cache no-store等。  如下图, 设置了**cache-control:max-age=31536000,public,immutable**  · cache-control: max-age=xxxx，public 客户端和代理服务器都可以缓存该资源； 客户端在xxx秒的有效期内，如果有请求该资源的需求的话就直接读取缓存,statu code:200 ，如果用户做了刷新操作，就向服务器发起http请求  · cache-control: max-age=xxxx，private 只让客户端可以缓存该资源；代理服务器不缓存 客户端在xxx秒内直接读取缓存,statu code:200  · cache-control: max-age=xxxx，immutable 客户端在xxx秒的有效期内，如果有请求该资源的需求的话就直接读取缓存,statu code:200 ，即使用户做了刷新操作，也不向服务器发起http请求  · cache-control: no-cache 跳过设置强缓存，但是不妨碍设置协商缓存；一般如果你做了强缓存，只有在强缓存失效了才走协商缓存的，设置了no-cache就不会走强缓存了，每次请求都回询问服务端。  · cache-control: no-store 不缓存，这个会让客户端、服务器都不缓存，也就没有所谓的强缓存、协商缓存了。 |
| 强缓存就是给资源设置个过期时间，客户端每次请求资源时都会看是否过期；只有在过期才会去询问服务器。所以，强缓存就是为了给客户端自给自足用的。而当某天，客户端请求该资源时发现其过期了，这是就会去请求服务器了，而这时候去请求服务器的这过程就可以设置协商缓存。这时候，协商缓存就是需要客户端和服务器两端进行交互的。  请求资源时，把用户本地该资源的 etag 同时带到服务端，服务端和最新资源做对比。 如果资源没更改，返回304，浏览器读取本地缓存。 如果资源有更改，返回200，返回最新的资源。  // response header  etag: '5c20abbd-e2e8'  last-modified: Mon, 24 Dec 2018 09:49:49 GMT  // request header 变为  if-none-matched: '5c20abbd-e2e8'  if-modified-since: Mon, 24 Dec 2018 09:49:49 GMT |
| 后端服务器如nodejs: res.setHeader('max-age': '3600 public') res.setHeader(etag: '5c20abbd-e2e8') res.setHeader('last-modified': Mon, 24 Dec 2018 09:49:49 GMT)  index.html文件采用协商缓存，理由就是要用户每次请求index.html不拿浏览器缓存，直接请求服务器，这样就保证资源更新了，用户能马上访问到新资源，如果服务端返回304，这时候再拿浏览器的缓存的index.html，**切记不要设置强缓存！！！**  **就像我目前那个flutter的博客一样，每次新push上去都需要禁用缓存重新刷新一下blog页面，如果自己是服务器后台的话，可以设置index.html为协商缓存。** |

* CDN

[面试官：如何理解CDN？说说实现原理？ · Issue #142 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/142)

[什么是cdn？ - 简书 (jianshu.com)](https://www.jianshu.com/p/ce98fbff39ac)

|  |
| --- |
| 在没有应用CDN时，我们使用域名访问某一个站点时的路径为  用户提交域名→浏览器对域名进行解释→DNS 解析得到目的主机的IP地址→根据IP地址访问发出请求→得到请求数据并回复  应用CDN后，DNS 返回的不再是 IP 地址，而是一个CNAME(Canonical Name ) 别名记录，指向CDN的全局负载均衡  CNAME实际上在域名解析的过程中承担了中间人（或者说代理）的角色，这是CDN实现的关键 **负载均衡系统** 由于没有返回IP地址，于是本地DNS会向负载均衡系统再发送请求 ，则进入到CDN的全局负载均衡系统进行智能调度：  看用户的 IP 地址，查表得知地理位置，找相对最近的边缘节点  看用户所在的运营商网络，找相同网络的边缘节点  检查边缘节点的负载情况，找负载较轻的节点  其他，比如节点的“健康状况”、服务能力、带宽、响应时间等  结合上面的因素，得到最合适的边缘节点，然后把这个节点返回给用户，用户就能够就近访问CDN的缓存代理  整体流程如下图： **缓存代理** 缓存系统是 CDN 的另一个关键组成部分，缓存系统会有选择地缓存那些最常用的那些资源  其中有两个衡量CDN服务质量的指标：   * 命中率：用户访问的资源恰好在缓存系统里，可以直接返回给用户，命中次数与所有访问次数之比 * 回源率：缓存里没有，必须用代理的方式回源站取，回源次数与所有访问次数之比   缓存系统也可以划分出层次，分成一级缓存节点和二级缓存节点。一级缓存配置高一些，直连源站，二级缓存配置低一些，直连用户  回源的时候二级缓存只找一级缓存，一级缓存没有才回源站，可以有效地减少真正的回源  现在的商业 CDN命中率都在 90% 以上，相当于把源站的服务能力放大了 10 倍以上 |

# 网络安全

* XSS 和 防范

[HTTP cookies - HTTP | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Cookies" \l "%E8%B7%9F%E8%B8%AA%E5%92%8C%E9%9A%90%E7%A7%81)

[Cross-site scripting（跨站脚本攻击） - 术语表 | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Glossary/Cross-site_scripting)

[什么是跨站脚本？如何防范 XSS 攻击 | w3c笔记 (w3cschool.cn)](https://www.w3cschool.cn/article/1220867.html)

|  |
| --- |
| XSS  跨站脚本攻击（Cross-site scripting，XSS）是一种安全漏洞，攻击者可以利用这种漏洞在网站上注入恶意的客户端代码。若受害者运行这些恶意代码，攻击者就可以突破网站的访问限制并冒充受害者。根据开放式 Web 应用安全项目（OWASP），XSS 在 2017 年被认为 [7 种最常见的 Web 应用程序漏洞之一](https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A7-Cross-Site_Scripting_(XSS))。  如果 Web 应用程序没有部署足够的安全验证，那么，这些攻击很容易成功。浏览器无法探测到这些恶意脚本是不可信的，所以，这些脚本可以任意读取 cookie，session tokens，或者其它敏感的网站信息，或者让恶意脚本重写[HTML](https://developer.mozilla.org/zh-CN/docs/Glossary/HTML)内容。  在以下 2 种情况下，容易发生 XSS 攻击：   1. 数据从一个不可靠的链接进入到一个 Web 应用程序。 2. 没有过滤掉恶意代码的动态内容被发送给 Web 用户。   恶意内容一般包括 [JavaScript](https://developer.mozilla.org/zh-CN/docs/Glossary/JavaScript)，但是，有时候也会包括 HTML，FLASH 或是其他浏览器可执行的代码。XSS 攻击的形式千差万别，但他们通常都会：将 cookies 或其他隐私信息发送给攻击者，将受害者重定向到由攻击者控制的网页，或是经由恶意网站在受害者的机器上进行其他恶意操作。  XSS 攻击可以分为 3 类：存储型（持久型）、反射型（非持久型）、DOM 型。  **存储型 XSS**  注入型脚本永久存储在目标服务器上。当浏览器请求数据时，脚本从服务器上传回并执行。  **反射型 XSS**  当用户点击一个恶意链接，或者提交一个表单，或者进入一个恶意网站时，注入脚本进入被攻击者的网站。Web 服务器将注入脚本，比如一个错误信息，搜索结果等 返回到用户的浏览器上。由于浏览器认为这个响应来自"可信任"的服务器，所以会执行这段脚本。  **基于 DOM 的 XSS**  通过修改原始的客户端代码，受害者浏览器的 DOM 环境改变，导致有效载荷的执行。也就是说，页面本身并没有变化，但由于 DOM 环境被恶意修改，有客户端代码被包含进了页面，并且意外执行。 |
| 当数据通过不受信任的来源（如 Web 请求）进入 Web 应用程序并未经验证就发送给用户时，就会发生 XSS 攻击。  XSS 会导致脚本在用户的浏览器中执行，从而导致会话被劫持、网站篡改以及将用户重定向到恶意站点。  从本质上讲，攻击者将恶意代码输入到用户输入的部分中，服务器希望该部分是数据（但实际上是为了执行而设计的代码）。  也就是服务器接收一段数据的时候，结果接收过来是一段被注入的代码，服务器还执行了这段代码。 |
| **验证/过滤输入**  理想情况下，所有输入都应根据可接受的值列表进行验证。  **设置 HttpOnly 标志**  XSS 攻击通常使用 JavaScript 来窃取会话 cookie（而普通的 Web 应用程序很少需要使用 JavaScript 来访问会话 cookie）。因此，设置 HttpOnly 标志可以保护会话 cookie 免受攻击者的攻击，同时不会限制正常行为。大多数浏览器都支持设置这个标志。  **使用响应头**  与 HttpOnly 标志类似，任何不应包含 HTML 或 JavaScript 的 HTTP 响应都可以利用“Content-Type”和“X-Content-Type-Options”标头来确保浏览器仅以预期的方式解释响应。  **编码输出**  任何用户输入的数据都应该被编码以防止它被读取为活动的。这可能需要 CSS、JavaScript、URL 和/或 HTML 编码。  确保输出在任何暴露的点都被编码尤为重要，因为相同的数据可以存储并显示在多个位置。 |

* CSRF 和 防范

[CSRF - 术语表 | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Glossary/CSRF)

[什么是CSRF攻击？如何防御CRSF攻击？ - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/114750961" \l ":~:text=%E7%9B%AE%E5%89%8D%E9%98%B2%E5%BE%A1 CSRF %E6%94%BB%E5%87%BB%E4%B8%BB%E8%A6%81%E6%9C%89%E4%B8%89%E7%A7%8D%E7%AD%96%E7%95%A5%EF%BC%9A%E9%AA%8C%E8%AF%81 HTTP Referer %E5%AD%97%E6%AE%B5%EF%BC%9B%E5%9C%A8%E8%AF%B7%E6%B1%82%E5%9C%B0%E5%9D%80%E4%B8%AD%E6%B7%BB%E5%8A%A0 token %E5%B9%B6%E9%AA%8C%E8%AF%81%EF%BC%9B%E5%9C%A8,%E5%8D%8F%E8%AE%AE%EF%BC%8C%E5%9C%A8 HTTP %E5%A4%B4%E4%B8%AD%E6%9C%89%E4%B8%80%E4%B8%AA%E5%AD%97%E6%AE%B5%E5%8F%AB Referer%EF%BC%8C%E5%AE%83%E8%AE%B0%E5%BD%95%E4%BA%86%E8%AF%A5 HTTP %E8%AF%B7%E6%B1%82%E7%9A%84%E6%9D%A5%E6%BA%90%E5%9C%B0%E5%9D%80%E3%80%82 %E5%9C%A8%E9%80%9A%E5%B8%B8%E6%83%85%E5%86%B5%E4%B8%8B%EF%BC%8C%E8%AE%BF%E9%97%AE%E4%B8%80%E4%B8%AA%E5%AE%89%E5%85%A8%E5%8F%97%E9%99%90%E9%A1%B5%E9%9D%A2%E7%9A%84%E8%AF%B7%E6%B1%82%E6%9D%A5%E8%87%AA%E4%BA%8E%E5%90%8C%E4%B8%80%E4%B8%AA%E7%BD%91%E7%AB%99%EF%BC%8C%E6%AF%94%E5%A6%82%E9%9C%80%E8%A6%81%E8%AE%BF%E9%97%AE bank.example/withdraw?)

|  |
| --- |
| CSRF  跨站请求伪造（CSRF）是一种冒充受信任用户，向服务器发送非预期请求的攻击方式。例如，这些非预期请求可能是通过在跳转链接后的 [URL](https://developer.mozilla.org/zh-CN/docs/Glossary/URL) 中加入恶意参数来完成：  <img src="https://www.example.com/index.php?action=delete&id=123">  对于在 https://www.example.com 有权限的用户，这个 <img> 标签会在他们根本注意不到的情况下对 https://www.example.com 执行这个操作，即使这个标签根本不在 https://www.example.com 内亦可。  有很多预防 CSRF 的方法，比如实现 [RESTful API](https://developer.mozilla.org/zh-CN/docs/Glossary/REST)，增加 secure token 等等。 |
| 目前防御 CSRF 攻击主要有三种策略：验证 HTTP Referer 字段；在请求地址中添加 token 并验证；在 HTTP 头中自定义属性并验证。 |
| 1. 验证 HTTP Referer 字段  根据 HTTP 协议，在 HTTP 头中有一个字段叫 Referer，它记录了该 HTTP 请求的来源地址。在通常情况下，访问一个安全受限页面的请求来自于同一个网站，比如需要访问 [http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory](https://link.zhihu.com/?target=http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory" \t "https://zhuanlan.zhihu.com/p/_blank)，用户必须先登陆 bank.example，然后通过点击页面上的按钮来触发转账事件。这时，该转帐请求的 Referer 值就会是转账按钮所在的页面的 URL，通常是以 bank.example 域名开头的地址。而如果黑客要对银行网站实施 CSRF 攻击，他只能在他自己的网站构造请求，当用户通过黑客的网站发送请求到银行时，**该请求的 Referer 是指向黑客自己的网站**。因此，要防御 CSRF 攻击，银行网站只需要对于每一个转账请求验证其 Referer 值，如果是以 bank.example 开头的域名，则说明该请求是来自银行网站自己的请求，是合法的。如果 Referer 是其他网站的话，则有可能是黑客的 CSRF 攻击，拒绝该请求。 2. 在请求地址中添加 token 并验证  CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。  这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于 session 之中，然后在每次请求时把 token 从 session 中拿出，与请求中的 token 进行比对，但这种方法的难点在于如何把 token 以参数的形式加入请求。对于 GET 请求，token 将附在请求地址之后，这样 URL 就变成 [http://url](https://link.zhihu.com/?target=http://url" \t "https://zhuanlan.zhihu.com/p/_blank)?csrftoken=tokenvalue。 而对于 POST 请求来说，要在 form 的最后加上 <input type=”hidden” name=”csrftoken” value=”tokenvalue”/>，这样就把 token 以参数的形式加入请求了。但是，在一个网站中，可以接受请求的地方非常多，要对于每一个请求都加上 token 是很麻烦的，并且很容易漏掉，通常使用的方法就是在每次页面加载时，使用 javascript 遍历整个 dom 树，对于 dom 中所有的 a 和 form 标签后加入 token。这样可以解决大部分的请求，但是对于在页面加载之后动态生成的 html 代码，这种方法就没有作用，还需要程序员在编码时手动添加 token。  该方法还有一个缺点是难以保证 token 本身的安全。特别是在一些论坛之类支持用户自己发表内容的网站，黑客可以在上面发布自己个人网站的地址。由于系统也会在这个地址后面加上 token，黑客可以在自己的网站上得到这个 token，并马上就可以发动 CSRF 攻击。为了避免这一点，系统可以在添加 token 的时候增加一个判断，如果这个链接是链到自己本站的，就在后面添加 token，如果是通向外网则不加。不过，即使这个 csrftoken 不以参数的形式附加在请求之中，黑客的网站也同样可以通过 Referer 来得到这个 token 值以发动 CSRF 攻击。这也是一些用户喜欢手动关闭浏览器 Referer 功能的原因。 |