# 常见的跨域方式

## 1、jsonp

**jsonp的原理**：所有非同源的请求，都会被作为跨域请求，浏览器会将非同源的响应数据丢弃。而通过一些标签发出的请求则不会被进行同源检查，比如script标签，img标签等等。

JSONP就是通过script标签加载数据的方式去获取数据当做JS代码来执行; 提前在页面上声明一个函数, 函数名通过接口传参的方式传给后台, 后台解析到函数名后在原始数据上包裹这个函数名, 发送给前端。

**前端需要传递的数据**：请求参数和接收数据的回调函数的函数名。

**后台**：将数据用这个函数名包裹后返回。

**请求：**

<script src="http://www.xxx.com/jsonp.php?a=1&b=2&cb=w124358539"></script>

**后台：**

|  |
| --- |
| const http = require('http');  const url = require('url');  http.createServer((req, res) => {  const query = url.parse(req.url, true).query;  const cb = query.cb;  const data = JSON.stringify({ status: 'success', data: { a: query.a, b: query.b,c:’其他数据’ } });  res.writeHead(200, { 'Content-Type': 'application/javascript' });  res.end(`${cb}(${data})`);  }).listen(3000, () => console.log('Server running at http://localhost:3000/')); |

## 2、cors

**原理**：CORS 的原理是通过在 HTTP 请求头中添加一些额外的字段，来告诉服务器是否允许跨域请求。浏览器在发送跨域请求时，会先发送一个“预检请求” OPTIONS 请求，来询问服务器是否允许跨域请求，并带上一些自定义的 HTTP 头部字段。如果服务器返回允许跨域请求的响应头部字段（Access-Control-Allow-Origin），浏览器就会发送真正的跨域请求。

**为什么**：因为它提供了一种标准化的机制来授权跨域请求，同时保证了跨域请求的安全性和可控性。由于 Web 应用程序的复杂性和互联网的开放性，CORS 已经成为了一种广泛使用的跨域方案，可以让 Web 应用程序更加灵活地进行跨域交互。

**后台需要做什么？**

Access-Control-Allow-Origin 指定允许跨域访问的源地址，可以是一个具体的域名或通配符 \*；

**前端发送cookie给后台,需要怎么做？**

后台需要在响应头中添加 Access-Control-Allow-Origin 和 Access-Control-Allow-Credentials 字段。其中，Access-Control-Allow-Origin 指定允许跨域访问的源地址；Access-Control-Allow-Credentials 指定是否允许发送带有 Cookie 等认证信息的跨域请求，需要设置为 true。

如果后台允许跨域请求并允许发送 Cookie，前端可以在发送跨域请求时，设置 XMLHttpRequest.withCredentials 为 true，或者使用 fetch API 时，将 credentials 选项设置为 'include'。

# 二、HTTPS:

**https的大致加密流程是怎么样的？**

1. 客户端向服务器发起请求，请求中包含使用的协议版本号、生成的一个随机数、以及客户端支持的加密方法。

2. 服务器端接收到请求后，确认双方使用的加密方法、并给出服务器的证书、以及一个服务器生成的随机数。

3. 客户端确认服务器证书有效后，生成一个新的随机数，并使用数字证书中的公钥，加密这个随机数，然后发给服务器。并且还会提供一个前面所有内容的 hash 的值，用来供服务器检验。

4. 服务器使用自己的私钥，来解密客户端发送过来的随机数。并提供前面所有内容的 hash 值来供客户端检验。

5. 客户端和服务器端根据约定的加密方法使用前面的三个随机数，生成对话秘钥，以后的对话过程都使用这个秘钥来加密信息。

**有什么常用的加密算法**

1. 对称加密算法：使用同一密钥进行加密和解密，例如AES、DES等。
2. 非对称加密算法：使用一对公钥和私钥进行加密和解密，例如RSA、DSA等。
3. 摘要算法：将消息压缩成一个固定长度的摘要值，例如MD5、SHA-1等。
4. 密钥交换算法：协议中用于交换密钥的算法，例如Diffie-Hellman算法等。

**对称加密和非对称加密发生在什么环节**

对称加密算法通常用于加密数据的传输，在对称加密中，发送方使用密钥加密数据，接收方使用相同的密钥解密数据。

非对称加密算法通常用于身份验证和密钥交换。在非对称加密中，发送方使用接收方的公钥加密数据，接收方使用自己的私钥解密数据，因此接收方的私钥必须保密不被泄露。

**非对称加密加密的是什么内容**

非对称加密算法加密的内容可以是任意数据。非对称加密算法通常用于身份验证和密钥交换。身份验证是指确认通信双方的身份，通常是通过使用数字证书来实现的。密钥交换是指双方通过加密方式协商出对称加密算法所需的密钥，以便在之后的通信中使用对称加密算法进行加密传输。在这些过程中，非对称加密算法用于加密数字证书和密钥交换所需的数据。

**为什么要有两种加密方式**

对称加密算法的优点是算法公开、计算量小、加密速度快、加密效率高。

对称加密算法的缺点是在数据传送前，发送方和接收方必须商定好秘钥，同时双方都要保存好秘钥。

非对称加密使用了一对密钥，公钥与私钥，安全性高，但加密与解密速度慢。

解决的办法是将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。

# 三、HTTP2:

**http2对比http1有什么优化和提升的地方？这些优化点背后的原理？**

多路复用

HTTP/2使用二进制协议传输数据，通过多路复用技术，可以在同一个连接上同时传输多个请求和响应，避免了HTTP/1中的“队头阻塞”问题。当客户端请求的多个资源时，HTTP/2可以将这些请求同时发出，服务器可以乱序的将响应数据返回给客户端，客户端再根据请求的ID和标识符进行响应的重组和处理。

首部压缩

HTTP/2对请求和响应的头部信息使用了HPACK算法进行压缩，减少了首部传输的大小，减轻了网络带宽的负担。

服务器推送

HTTP/2中，服务器可以在收到客户端请求后，主动向客户端发送一些该请求需要的资源，避免了客户端多次请求服务器的情况，从而提高了页面加载速度。

二进制传输

HTTP/2采用了二进制格式传输，相比于HTTP/1中的文本格式，二进制格式更加紧凑，减少了数据传输的大小，提高了传输的效率。

流量控制

HTTP/2中支持流量控制，通过设置窗口大小和流控制标识符，可以在连接上动态地控制数据流的速率，避免了因为服务器传输速率过快而导致客户端无法处理数据的情况。

# 四、QUIC/HTTP3

**HTTP3对比HTTP2有什么优化和提升的地方？**

HTTP/3是一种新的网络协议，它的主要改进是通过使用QUIC协议来减少网络延迟和提高安全性。HTTP/3相对于HTTP/2的主要优点包括：

减少网络延迟：HTTP/3使用QUIC协议作为传输层协议，与TCP相比，QUIC通过在连接建立时同时进行加密和多路复用，从而降低了网络延迟，减少了连接建立的时间。

减少丢包重传：在HTTP/3中，使用QUIC协议的数据传输采用了更加智能的拥塞控制算法，并且它具有快速的重传机制，这些优化可以减少丢包和重传数据的次数，从而提高了网络性能和稳定性。

更加安全：HTTP/3通过在传输层进行加密，提供了更加安全的通信。QUIC协议的加密方式使用了TLS1.3的加密机制，这意味着HTTP/3具有与TLS1.3相同的安全性和可靠性。

更加灵活：HTTP/3采用了一个全新的协议栈，这使得它能够更加灵活地适应未来的网络需求。HTTP/3还支持动态更新传输层密钥，这样就可以避免长时间的连接对安全性的威胁。

总之，HTTP/3的改进主要是通过使用QUIC协议来减少网络延迟和提高安全性，这使得它比HTTP/2更加高效和可靠。

**HTTP 3是TCP还是UDP的？与HTTP2底层协议有什么区别**

HTTP/3使用QUIC作为传输层协议，而QUIC则是基于UDP协议的。相比之下，HTTP/2使用的是TCP协议作为传输层协议。

**UDP与TCP的区别**

|  |  |  |
| --- | --- | --- |
|  | UDP | TCP |
| 是否连接 | 无连接 | 面向连接 |
| 是否可靠 | 不可靠传输，不使用流量控制和拥塞控制 | 可靠传输（数据顺序和正确性），使用流量控制和拥塞控制 |
| 连接对象个数 | 支持一对一，一对多，多对一和多对多交互通信 | 只能是一对一通信 |
| 传输方式 | 面向报文 | 面向字节流 |
| 首部开销 | 首部开销小，仅8字节 | 首部最小20字节，最大60字节 |
| 适用场景 | 适用于实时应用，例如视频会议、直播 | 适用于要求可靠传输的应用，例如文件传输 |

# 五、WebSocket

**WebSocket有什么特点**

WebSocket是一种基于TCP协议的双向通信协议，它与HTTP协议一样使用标准的TCP端口(80,443)，因此可以通过防火墙和代理服务器。与传统的HTTP请求响应模式不同，WebSocket通过一次握手即可建立客户端和服务器之间的持久连接，可以实现实时的双向数据传输。

双向通信：WebSocket支持双向通信，客户端和服务器可以在任意时刻相互发送消息。

较少的网络流量：在传统的HTTP协议中，客户端需要不停地发送请求，而服务器也需要不停地响应请求，这样会造成大量的网络流量。WebSocket只需要进行一次握手，之后可以通过双向通信实现实时数据的传输，因此可以大大减少网络流量。

快速响应：由于WebSocket连接的持久性，客户端和服务器之间可以实时地传输数据，响应速度非常快。

低延迟：由于WebSocket连接的双向通信特性，可以避免HTTP协议中常见的“轮询”和“长轮询”等技术，从而大大降低延迟。

节省服务器资源：传统的HTTP协议需要每次请求都重新建立连接，而WebSocket只需要一次握手即可建立持久连接，可以大大节省服务器资源。

# 六、Vue

**Vue的路由导航守卫的实现原理是什么，有哪几种模式，不同模式对URL的影响是什么？**

Vue 的路由导航守卫是一种用于在用户导航到不同路由时执行的钩子函数。它可以用来控制路由的访问权限、检查表单是否填写完整等等。Vue 路由导航守卫的实现原理是基于 Vue Router 的路由钩子函数，包括全局前置守卫、全局后置守卫、路由独享守卫和组件内的守卫。

全局前置守卫beforeEach

全局前置守卫是在用户导航到路由之前执行的钩子函数。在这个钩子函数中，你可以检查用户是否有足够的权限访问该路由，或者检查某些必要条件是否满足。如果这个钩子函数返回 false，导航就会被中断，用户将留在当前路由。

全局后置守卫afterEach

全局后置守卫是在用户导航到路由之后执行的钩子函数。在这个钩子函数中，你可以做一些与路由相关的清理工作，比如关闭对话框、清空表单等等。由于全局后置守卫没有办法中断导航，所以它通常用于收集路由跳转的统计数据等任务。

路由独享守卫beforeEnter

路由独享守卫是在某个路由上定义的钩子函数。在这个钩子函数中，你可以做与该路由相关的特定任务，比如检查路由参数是否合法等等。如果这个钩子函数返回 false，导航也会被中断。

组件内的守卫

组件内的守卫是在某个组件内定义的钩子函数。在这个钩子函数中，你可以做与该组件相关的特定任务，比如检查用户是否已登录等等。这些钩子函数可以分为三种类型：beforeRouteEnter、beforeRouteUpdate 和 beforeRouteLeave。其中，beforeRouteEnter 和 beforeRouteUpdate 都是在组件渲染之前执行的，而 beforeRouteLeave 则是在组件卸载之前执行的。

Vue 路由中，有两种路由模式：history 和 hash

1. history通过历史记录站的API实现，hash通过监听location hash值变化来实现
2. history没有#,比较美观，但是需要后端配合。hash有#号，不需要后端配合，且兼容性好。
3. hashRouter原理：通过window.onhashchange获取url中hash值。historyRouter原理：通过history.pushState,使用它来跳转不会触发页面刷新，使用window.onpopstate监听浏览器的前进和后退。

**Vue有什么组件通信的方式。**

**1.props和$emit**

子组件可以用props接收父组件传递过来的数据

子组件通过$emit触发父组件的事件，可以向父组件传递数据

**2.全局事件总线**

通过Vue.prototype.$bus=this定义事件总线。

a组件：this.$bus.$emit()触发事件，传递数据

b组件：this.$bus.$on()响应事件，获取数据

**3.发布与订阅**

安装引入

import pubsub from 'pubsub-js'

Vue.use(pubsub)

a组件：pubsub.publish()发布消息，传递数据

b组件：pubsub.subscribe ()订阅消息，获取数据

**4.ref / $refs**

**父子通信**

原理：ref这个属性用在子组件上，它的引用就指向了子组件的实例。可以通过实例来访问组件的数据和方法。父组件触发子组件的事件，就实现父给子传递数据。

父组件通过 this.$refs.子组件名.函数名 调用

**5.provid/inject依赖注入**

**组件与后代组件通信**

父组件：provide() { return {num: this.num}}；

子组件：inject: ['num'] 获取数据

# 七WebWorker/ServiceWorker/WebAsssmembly

**WebWorker的作用是什么？**

Web Worker是一种在后台运行的JavaScript脚本，用于在主线程之外执行耗时的计算任务。它可以在不阻塞UI线程的情况下执行代码，从而提高Web应用程序的性能和响应速度。

**可以用于什么场景？**

大规模计算：在Web Worker中执行大量的计算任务，如数学计算、图像处理等。这些任务可能需要较长的时间才能完成，如果在主线程中执行，会导致UI线程阻塞，使应用程序的响应变慢或不稳定。

后台数据处理：Web Worker可以处理大量的数据，如文件上传、下载、解析、转换等任务。这些任务通常需要在后台执行，以避免阻塞UI线程。

实时通信：Web Worker可以与Web应用程序的主线程进行实时通信，从而实现复杂的交互效果。例如，可以通过Web Worker在后台计算数据，然后将结果发送到主线程以更新UI。

**如何通信（监听消息和发送消息）？**

主线程可以使用Worker对象的postMessage()方法向Web Worker发送消息。Web Worker可以使用self.onmessage事件监听来自主线程的消息，并使用postMessage()方法向主线程发送响应。主线程可以使用worker.onmessage事件监听来自Web Worker的响应。

**主线程代码**

// 创建Web Worker

var worker = new Worker('worker.js');

// 监听Web Worker的响应

worker.onmessage = function(event) {

console.log('Received message from worker: ' + event.data);

};

// 向Web Worker发送消息

worker.postMessage('Hello from main thread!');

**Web Worker代码（worker.js）：**

// 监听主线程的消息

self.onmessage = function(event) {

console.log('Received message from main thread: ' + event.data);

// 向主线程发送响应

self.postMessage('Hello from worker!');

};

**ServiceWorker的作用是什么，PWA是什么？**

ServiceWorker是一种在Web应用程序背后运行的JavaScript脚本，可以用于缓存和离线处理等功能。它能够让Web应用程序在离线状态下也能够运行，同时提高Web应用程序的性能和响应速度。

PWA是一种渐进式Web应用程序（Progressive Web App）的缩写，它是一种结合Web技术和应用程序功能的Web应用程序，可以实现类似原生应用程序的用户体验。PWA通常使用ServiceWorker、Web App Manifest和响应式设计等技术来实现。

**PWA具有以下特点：**

离线访问：PWA可以在离线状态下访问Web应用程序，并且可以缓存用户最近使用过的数据，从而提高用户体验。

应用程序外观：PWA可以像原生应用程序一样显示在设备的主屏幕上，并且可以在应用程序内部运行，从而提供更加直观和便捷的用户体验。

推送通知：PWA可以通过Web推送通知API向用户发送推送通知，实现即时通知和消息提醒功能。

响应式设计：PWA使用响应式设计技术，可以适应不同设备和不同分辨率的屏幕，从而提供一致的用户体验。

**WebAssembly的作用是什么，能应用什么场景里？**

WebAssembly是一种新型的低级字节码语言，它可以在Web浏览器中运行原生代码，从而实现更快的性能和更广泛的应用场景。WebAssembly通常用于需要高性能计算和密集计算的Web应用程序，如游戏、音视频处理和机器学习等。

# 八、前端安全

**xss是什么，有什么攻击场景，具体如何防范**

XSS（跨站脚本攻击）是一种常见的Web安全漏洞，它利用了网站的漏洞，将恶意脚本注入到网页中，从而导致攻击者能够窃取用户的敏感信息，如会话令牌、密码等。

**攻击场景**

XSS 可以分为存储型、反射型和 DOM 型：

• 存储型指的是恶意脚本会存储在目标服务器上，当浏览器请求数据时，脚本从服务器传回并执行。

• 反射型指的是攻击者诱导用户访问一个带有恶意代码的 URL 后，服务器端接收数据后处理，然后把带有恶意代码的数据发送到浏览器端，浏览器端解析这段带有 XSS 代码的数据后当做脚本执行，最终完成 XSS 攻击。

• DOM 型指的通过修改页面的 DOM 节点形成的 XSS。

**防范措施：**

过滤用户输入：网站应该过滤用户输入，防止用户输入恶意代码，如去掉特殊字符、限制输入长度等。

转义输出：将输出的特殊字符转义，使其不被浏览器解释为HTML代码。这可以通过使用一些开源的安全库来实现，如OWASP ESAPI、HTML Purifier等。

使用HTTP Only Cookie：使用HTTP Only Cookie可以防止通过JavaScript来获取敏感信息，这样即使攻击者成功注入恶意脚本，也无法窃取Cookie中的信息。

CSP：CSP（内容安全策略）是一种在浏览器端限制页面资源加载和执行的机制，可以有效减少XSS攻击。CSP可以限制页面可以加载的资源类型和来源，只允许特定来源的代码执行。

**CSP是什么，有什么作用**

CSP（Content Security Policy）是一种安全策略，它允许网站管理员控制浏览器中的内容加载和执行。CSP通过指定特定的策略来限制网页可以加载的资源类型和来源，从而提高Web应用程序的安全性，减少恶意脚本攻击。

**CSP的作用包括以下几个方面：**

防范XSS攻击：通过限制网页中可以执行的脚本来源和类型，CSP可以有效地防止XSS攻击。当攻击者尝试注入恶意脚本时，这些脚本将被CSP拦截并阻止执行。

防范数据盗取：CSP还可以防范攻击者利用恶意脚本窃取用户敏感信息的行为。通过限制网页中可以加载的资源类型和来源，CSP可以确保用户数据只被发送到受信任的服务器。

减少代码注入：CSP还可以减少代码注入攻击，如SQL注入和命令注入。通过限制网页中可以加载的资源类型和来源，CSP可以防止攻击者注入恶意代码来执行非法操作。

**CSRF是什么，有什么攻击场景，具体如何防范**

CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作。

**攻击场景**

• GET 类型的 CSRF 攻击，比如在网站中的一个 img 标签里构建一个请求，当用户打开这个网站的时候就会自动发起提交。

• POST 类型的 CSRF 攻击，比如构建一个表单，然后隐藏它，当用户进入页面时，自动提交这个表单。

• 链接类型的 CSRF 攻击，比如在 a 标签的 href 属性里构建一个请求，然后诱导用户去点击。

**具体如何防范**

1进行同源检测，服务器根据 http 请求头中 origin 或者 referer 信息来判断请求是否为允许访问的站点，从而对请求进行过滤。

2使用CSRF Token：CSRF Token是一种与用户会话相关的令牌，它可以防止攻击者伪造请求。网站应该向每个页面添加CSRF Token，并在每个请求中验证该Token是否正确。

3在设置 cookie 属性的时候设置 Samesite ，限制 cookie 不能作为被第三方使用，从而可以避免被攻击者利用。

4.对 Cookie 进行双重验证，服务器在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串，然后当用户再次向服务器发送请求的时候，从 cookie 中取出这个字符串，添加到 URL 参数中，然后服务器通过对 cookie 中的数据和参数中的数据进行比较，来进行验证。