## 什么是浏览器的同源策略?

所谓同源是指：域名、协议、端口必须都一致。

DOM同源策略：禁止对不同源页面DOM进行操作。这里主要场景是iframe跨域的情况，不同域名的iframe是限制互相访问的。

XmlHttpRequest同源策略：禁止使用XHR对象向不同源的服务器地址发起HTTP请求。 只要协议、域名、端口有任何一个不同，都被当作是不同的域，之间的请求就是跨域操作。

跨域限制主要是为了安全考虑。

**跨域的解决方案?**

由于现在大部分都是采用前后分离的方式开发。经常会和跨域打交道。

Vue解决跨域:

proxyTable:

在webpack的配置文件config的index文件中添加proxyTable配置代理跨域.

proxyTable: {

'/apis': {

target: "https://cnodejs.org", // 需要进行代理跨域主机

secure: true, // 如果是https接口，需要配置这个参数

changeOrigin: true, //是否跨域

pathRewrite: {

'^/apis': '' //请求结束 将url里面 /apis重写成"" }

}

}

}

React解决跨域:

使用中间件http-proxy-middleware:

安装中间件之后,在src下建立setupProxy.js文件,写入配置,好处是可以配置多个代理:

const proxy = require('http-proxy-middleware');

module.exports = function (app) {

app.use(proxy('/apis', {

target: "https://api.juooo.com",

secure: false,

changeOrigin: true,

pathRewrite: {

"^/apis": "/"

}

}));

app.use(proxy('/apm', {

target: "https://m.juooo.com",

secure: false,

changeOrigin: true,

pathRewrite: {

"^/apm": "/"

}

}));

}

JSONP跨域: 动态创建 script 标签，然后利用 src 属性进行跨域.原理是src请求网络资源不受同源策略的影响:

大概的实现就是封装jsonp函数,传入url和回调函数,动态创建script标签请求跨域,跨域成功执行回调函数.

function jsonp({ url, params, callback }) {

return new Promise((resolve, reject) => {

let script = document.createElement('script')

window[callback] = function(data) {

resolve(data)

document.body.removeChild(script)

}

params = { ...params, callback }

let arrs = []

for (let key in params) {

arrs.push(`${key}=${params[key]}`)

}

script.src = `${url}?${arrs.join('&')}`

document.body.appendChild(script)

})

}

jsonp({

url: 'http://localhost:3000/say',

params: { wd: 'Iloveyou' },

callback: 'show'

}).then(data => {

console.log(data)

})

CORS跨域: 后台开启允许跨域.

此外还有location.hash ,postMessage , document.domain等实现跨域.

## TCP连接为什么需要三次握手，两次不可以吗？或者四次、五次可以吗？

TCP握手详解

建立连接协议（三次握手）：

第一次握手：客户端发送syn包（syn=x）的数据包到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

3次握手完成两个重要的功能，既要双方都做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

如果把三次握手改成仅需要两次握手，死锁是可能发生的。客户端认为连接还未建立成功，将忽略服务端发来的任何数据分组，等待连接确认应答。而服务端在发出的数据超时后，重复发送同样的数据。这样就形成了死锁。

**什么是Ajax? jQuery.ajax,和axios和fetch之间有什么区别?各有什么优缺点?**

Ajax的全称Asynchronous JavaScript + XML(异步JavaScript和XML).Ajax是一种技术方案，最核心的依赖是浏览器提供的XMLHttpRequest对象，是这个对象使得浏览器可以发出HTTP请求与接收HTTP响应。实现了在页面不刷新个情况下和服务器进行数据交互。

手写ajax:

var xhr = new XMLHttpRequest()

xhr.open('GET', 'http://hello.php', true)

xhr.onreadystatechange = function(){

if(xhr.readyState === 4) {

if((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){

//成功了

console.log(xhr.responseText)

} else {

console.log('服务器异常')

}

}

}

Jquery.ajax基于原生的XHR开发，XHR本身的架构不清晰.

针对MVC的编程,不符合现在前端MVVM的浪潮

JQuery整个项目太大，单纯使用ajax却要引入整个JQuery非常的不合理.

axios:

axios从 node.js 创建 http 请求;

支持 Promise API

客户端支持防止CSRF;

提供了一些并发请求的接口;

不支持jsonp，不过可以引入jsonp模块.

fetch:

fetch基于Promise,脱离了XHR，是ES规范里新的实现方式.

更加底层，提供的API丰富（request, response）

但是fetch只对网络请求报错，对400，500都当做成功的请求，需要封装去处理.

fetch默认不会带cookie，需要添加配置项

fetch没有办法原生监测请求的进度，而XHR可以.

fetch('https://api.myjson.com/bins/of6pw')

.then(function(response) {

return response.json();

})

.then(function(res) {

console.log(res);

});

**前端需要掌握的加密技术?**

使用 https;

使用md5加密;

将密码+时间戳加密后传输，然后服务端再解密.

**Get请求和post请求的区别?**

GET 和 POST 只是 HTTP 协议中两种请求方式（异曲同工），而 HTTP 协议是基于 TCP/IP 的应用层协议，无论 GET 还是 POST，用的都是同一个传输层协议，所以在传输上，没有区别。

1.url可见性：

get，参数url可见

post，url参数不可见

2.传输数据的大小：

get一般传输数据大小不超过2k-4k.

post请求传输数据的大小没有限制.

3.数据传输上：

get，通过拼接url进行传递参数

post，通过body体传输参数

4.后退页面的反应：

get请求页面后退时，不产生影响.

post请求页面后退时，会重新提交请求.

5.安全性：

原则上post肯定要比get安全，毕竟post传输参数时url不可见.

**什么是Web Workers**

H5规范提供了js多线程的实现, 取名为: Web Workers.

创建一个Worker对象，指向一个js文件，然后通过Worker对象往js文件发送消息，js文件内部的处理逻辑，处理完毕后，再发送消息回到当前页面，纯异步方式，不影响当前主页面渲染。

相关API:

\* Worker: 构造函数, 加载分线程执行的js文件

\* Worker.prototype.onmessage: 用于接收另一个线程的回调函数

\* Worker.prototype.postMessage: 向另一个线程发送消息

主线程:

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

// 绑定接收消息的监听

worker.onmessage = function (event) {

console.log('主线程接收分线程返回的数据: '+event.data)

alert(event.data)

}

// 向分线程发送消息

worker.postMessage(number)

console.log('主线程向分线程发送数据: '+number)

分线程myworker.js:

var number = event.data

console.log('分线程接收到主线程发送的数据: '+number)

//计算

var result = fibonacci(number)

postMessage(result)

console.log('分线程向主线程返回数据: '+result)

//分线程中的全局对象不再是window, 所以在分线程中不可能更新界面

不足

\* worker内代码不能操作DOM(更新UI)

\* 不能跨域加载JS

\* 不是每个浏览器都支持这个新特性

**对http协议有了解吗?**

1.HTTP也叫作超文本传输协议，全称：Hyper Transfer Protocol。可以传输文字、图像、视频和各种流文件。

2. HTTP是无状态的：同一个客户端，连续发起多次请求，对HTTP服务器来说，都是新的请求，HTTP没办法知道来自一个客户端。

3. HTTP消息结构:

客户端发送一个HTTP请求到服务器的请求消息包括以下格式：请求行（request line）、请求头部（header）、空行和请求数据四个部分组成

HTTP响应也由四个部分组成，分别是：状态行、消息报头、空行和响应正文.

4.基本请求方法:

GET：向特定的资源发出请求。

POST：向指定资源提交数据进行处理请求,数据被包含在请求体中。

PUT：向指定资源位置上传其最新内容.

HEAD：向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。

5.基本状态码:

以2开头 （请求成功）表示成功处理了请求的状态代码.

以3开头 （请求被重定向）表示要完成请求，需要进一步操作。

以4开头 （请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。

以5开头（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。

**http和https的区别?**

HTTP特点：

1.https协议需要申请证书，一般免费证书较少，因而需要一定费用.

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

5. 苹果现在强制所有iOS App使用HTTPS加密。安卓9.0以上也要使用https协议.

**一个url的是由哪些部分组成的?**

1.协议部分：URL的协议部分为http：或https:

2.服务器hostname(主机名）：是指存放资源的服务器的域名系统 (DNS) 主机名或 IP 地址。

3. 端口部分：跟在域名后面的是端口,默认是80

4.path部分：从域名后的第一个“/”开始到最后一个“/”为止，是path部分。

5. ?query(查询)：可选，用于给动态网页传递参数.

6. hash:从“#”开始到最后，都是hash部分。

**url中#(hash)的含义?**

1.hash 属性是一个可读可写的字符串，该字符串是 URL 的锚部分.

2.“#”代表网页中的一个位置。其右面的字符，就是该位置的标识符浏览器读取这个URL后，会自动将print位置滚动至可视区域。（单页应用）

3.HTTP请求不包括'#',所以单单改变#后的部分，浏览器只会滚动到相应位置，不会重新加载网页。

4.使用window.location.hash读取#值.

5.onhashchange事件,这是一个HTML 5新增的事件，当#值发生变化时，就会触发这个事件。

**用户输入一个url地址,按下回车之后发生了什么?**

互联网个网络xu设备间的通信都必须遵守TCP/IP协议标准。利用TCP/IP进行通信时，会通过分层顺序与对端进行通信，发送数据会依次经过应用层，传输层，网络层，数据链路层，物理层。而接受数据的顺序刚好相反。

1.用户在应用层输入url

2. 应用层DNS解析域名，获得对端IP地址.

3. 应用层客户端发送HTTP请求:

HTTP请求包括请求报头和请求主体。其中请求报头中包含了请求方法，请求资源，请求所使用协议，以及返回的资源是否需要缓存，客户端是否需要发送cookie等。

4. 传输层TCP传输报文:

传输层的TCP协议提供可靠的字节流服务，他为了方便传输，将大块的数据分割成以报文段为基本单位的数据包进行管理。为了保证可靠性传输，TCP采用三次握手来保证可靠性传输。

5. 网络层IP协议查询MAC地址.

IP协议把TCP分割的数据包传送给接收方。而要保证却是能够传送给对方主机还需要MAC地址，也就是物理地址。IP地址和MAC地址是一一对应关系。一个设备有且只有一个MAC地址。IP地址可以更换，MAC不会变。

6. 数据到达链路层

7. 服务器接受请求

服务端主机的网卡接收到数据后，驱动操作系统拿到数据，自下而上进行解包,从原路返回响应.

响应报文中包括协议名称/协议版本，状态码，状态码描述等信息。其中常见状态码：200 表示请求资源成功;301：永久重定向，表示资源已经永久性重定向到指定位置;304:缓存;500,服务器错误;404页面未找到.

答案2:

1. 域名解析,解析主机名得到对应的IP地址.

2. 在Internet中所有的传输都是通过TCP/IP进行的,发起TCP3次握手.

3. 建立TCP连接后发起HTTP请求.

4. 服务器端响应http请求，浏览器得到html代码

5. 浏览器解析html代码，并请求html代码中的资源.

6. 浏览器对页面进行渲染呈现给用户.

**请求报文包含了哪些部分?**

1. 请求行:

包括请求方法字段、URL字段和HTTP协议版本

例如：GET /index.html HTTP/1.1和GET/sample.jspHTTP/1.1

get方法将数据拼接在url后面，传递参数受限

请求方法有：GET、POST、HEAD、PUT、OPTIONS、TRACE、CONNECT

2. 请求头(key value形式), 包含浏览器，主机，接受的编码方式和压缩方式,例如:

Accept:image/gif.image/jpeg,\*/\* //请求格式

Accept-Language:zh-cn //语言

Connection:Keep-Alive //持续TCP连接

Host:localhost //主机

User-Agent:Mozila/4.0(compatible;MSIE5.01;Window NT5.0) //浏览器信息

Accept-Encoding:gzip,deflate // 压缩方式

3.空行, 是一个回车符和换行符,表示请求头结束, 通知服务器以下不再有请求头.

4. 请求体(数据) : post方法中，会把数据以key value形式发送请求,包含在请求体(body)里.

**说说你所知道的web安全及防护措施.**

SQL注入、XSS、

CSRF攻击:

跨站点请求伪造（Cross-Site Request Forgeries），冒充用户发起请求.

预防:

验证码；强制用户必须与应用进行交互，才能完成最终请求.

尽量使用 post ，限制 get 使用；get 太容易被拿来做 csrf 攻击；

token 验证 CSRF 防御机制是公认最合适的方案。

XSS攻击:

攻击者想尽一切办法将可以执行的代码注入到网页中. 比如攻击的代码被服务端写入进数据库中，这种攻击危害性很大，因为如果网站访问量很大的话，就会导致大量正常访问页面的用户都受到攻击。

预防:

转义字符:

首先，对于用户的输入应该是永远不信任的。最普遍的做法就是转义输入输出的内容，对于引号、尖括号、斜杠进行转义.

使用函数过滤script 标签.

CSP

CSP 本质上就是建立白名单，开发者明确告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截是由浏览器自己实现的。

设置 HTTP Header 中的 Content-Security-Policy

设置 meta 标签的方式 <meta http-equiv="Content-Security-Policy">开启白名单.

**了解token加密吗? token的原理：**

第一步：后端随机产生一个 token，把这个token 保存到 session 状态中；同时后端把这个token 交给前端页面；

第二步：前端页面提交请求时，把 token 加入到请求数据或者头信息中，一起传给后端；

后端验证前端传来的 token 与 session 是否一致，一致则合法，否则是非法请求。

**Cookie,session, LocalStorage和sessionStorage的区别?**

Cookie:

1.由服务端生成，保存在客户端.

2. cookie中保存的是字符串,存储大小有限（最大 4kb ).

3.可接受过期时间, 设置了过期时间,直到过期时间结束后才消失.

4. 一般用来判断用户是否登陆过网站，以便下次登录时能够实现自动登录（或者记住密码）保存登录时间、浏览次数等信息.

Session:

1. session依赖cookie才能正常工作,当需要记住用户时，在服务端会设置一个响应头Set-Cookie，作为sessionID 保存在cookie中返回给客户端，客户端接收到这个响应后，此后发送的每一个请求浏览器都会自动带上Cookie请求头，服务端将客户端发送的请求中的cookie值的sessionID的与内存中的session进行对比，就可以识别这个客户端了.

2. session保存在服务端,若访问较多会影响服务器性能

3. session中保存的是对象,存储大小无限制.

webStorage:

webStorage 是 html5 提供的本地存储方案，包括两种方式：sessionStorage 和 localStorage。

1. 保存在客户端，不与服务器通信，因此比 cookie 更安全、速度更快

2. 存储空间有限，但比 cookie 大（5MB)

3. 仅支持 String 类型的存储内容,存取需要用JSON转化.

localStorage和sessionStorage的区别在于localStorage是持久化的本地存储，浏览器关闭重新打开数据依然存在. sessionStorage浏览器窗口关闭后数据就被销毁。

**怎么在获取当前页面的url地址?**

1. window.location.href(设置或获取整个 URL 为字符串).

2. window.location.protocol(设置或获取 URL 的协议部分).

3. window.location.host(设置或获取 URL 的主机部分).

4. window.location.port(设置或获取与 URL 关联的端口号码).

5. window.location.pathname(设置或获取与 URL 的路径部分（就是文件地址）)

6. window.location.search(设置或获取 href 属性中跟在问号后面的部分)

7. window.location.hash(设置或获取 href 属性中在井号“#”后面的分段).

在Vue中: this.$route.path获取路由路径, this.$route.params

获取路由参数.

在React中:

1.this.props. match：包含了具体的 url 信息，在 params 字段中可以获取到各个路由参数的值。

2. this.props.history:包含了组件可以使用的各种路由系统的方法，常用的有 push 和 replace，两者都是跳转页面，但是 replace 不会引起页面的刷新，仅仅是改变 url。

3.this.props.location:相当于URL 的对象形式表示，通过 search 字段可以获取到 url 中的 query 信息。

**什么是ajax轮询和长轮询(long poll)?**

HTTP是一个无状态协议, 同一个客户端，连续发起多次请求，对HTTP服务器来说，都是新的请求，HTTP没办法知道来自一个客户端.

HTTP连接是被动的，服务端不能主动联系客户端，只能有客户端发起。

ajax轮询和长轮询其实都是在不断地建立HTTP连接，然后等待服务端处理.

短轮询:

ajax轮询的原理非常简单，利用setInterval让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息。

轮询大量耗费服务器内存和宽带资源，因为不停的请求服务器，很多时候 并没有新的数据更新，因此绝大部分请求都是无效请求

数据不一定是实时更新，要看设定的请求间隔，基本会有延迟。

长轮询:

long poll(长轮询) 其实原理跟 ajax轮询 差不多，都是采用轮询的方式，不过采取的是阻塞模型（相当于一直打电话，没收到就不挂电话），也就是说，客户端发起连接后，如果没消息，就一直不返回Response给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。

解决了轮询的两个大问题，数据实时更新；

唯一的缺点是服务器在挂起的时候比较耗内存.

前端在发起请求的时候设置timeout超时时间,长轮询实现由服务端代码实现.

**什么是webSocket?**

知道了短轮询和长轮询之后我们知道这两种方式都不是最好的方式，需要很多资源。

短轮询需要服务端更快的处理速度,长轮询需要服务端更好的处理高并发能力.

在传统的方式上，要不断的建立，关闭HTTP协议，由于HTTP是非状态性的，每次都要重新传输 identity info （鉴别信息），来告诉服务端你是谁。

WebSocket是HTML5出的东西（协议）, 它是HTTP协议上的一种补充, 是一个持久化的协议.当服务器完成协议升级后（HTTP->Websocket），服务端就可以主动推送信息给客户端, 只需要经过一次HTTP请求，就可以做到源源不断的信息传送了。

对于不支持websocket的浏览器,我们可以通过轮询和长轮询来模拟实现.

**高并发请求优化方案(秒杀活动)?**

1.限流，点击提交按钮后按钮置灰，显示为正在排队中，等处理结束或若干秒后，才允许用户点击.

2. 对于所有的数据和接口设置响应头，利用express模拟，如果两次请求间隔小于5秒，直接返回304，不需要服务器进行处理.

app.all('\*', function(req, res, next){

res.set('Cache-Control','public,max-age=5')

if ((new Date().getTime() - req.headers['if-modified-since'] )< 5000) {

// 检查时间戳

res.statusCode = 304

res.end()

}

else {

var time =(new Date()).getTime().toString()

res.set('Last-Modified', time)

}

next()

})

3. 做限定，5秒内刷新页面只获取一次列表数据，避免高频刷新带给服务器的压力.

async init() {

try {

const store = JSON.parse(util.getStore('hopoActiveInfo'))

// 避免高频刷新增加服务器压力

if (store && (new Date() - new Date(store.getTime)) < 5000) {

this.basicInfo = store

} else {

this.basicInfo = await getActiveInfo()

this.basicInfo.getTime = new Date()

}

util.setStore(this.basicInfo, 'hopoActiveInfo')

this.btn.noPeopleAndStart.detail[0].text = `${

this.basicInfo.directBuyPrice

} 元直接购买`

this.computedStatus()

} catch (error) {

console.log(error)

}

4. 静态资源上传cdn.

**谈一谈浏览器的缓存机制.**

缓存可以说是性能优化中简单高效的一种优化方式了。一个优秀的缓存策略可以缩短网页请求资源的距离，减少延迟，并且由于缓存文件可以重复利用，还可以减少带宽，降低网络负荷。

1.缓存分类:

强缓存:

用户发送的请求，直接从客户端缓存中获取，不发送请求到服务器，不与服务器发生交互行为。

通过设置两种 HTTP Header 实现：Expires 和 Cache-Control。

Expires是Web服务器响应消息头字段，在响应http请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。

Cache-Control在HTTP/1.1中是最重要的规则，主要用于控制网页缓存。比如当Cache-Control:max-age=300时，则代表在这个请求正确返回时间（浏览器也会记录下来）的5分钟内再次加载资源，就会命中强缓存。

协商缓存:

用户发送的请求，发送到服务器后，由服务器判定是否从缓存中获取资源。

强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：1. 协商缓存生效,返回304和Not Modified;2. 协商缓存失效,返回200和请求结果

协商缓存可以通过设置两种 HTTP Header 实现：Last-Modified 和 ETag 。

协商缓存生效，返回304和Not Modified

2.缓存位置:

(1)Service Worker 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用 Service Worker的话，传输协议必须为 HTTPS。

实现Service Worker功能:首先需要先注册 Service Worker，然后监听到 install 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。

(2) Memory Cache, 也就是内存中的缓存，主要包含的是当前中页面中已经抓取到的资源. 计算机中的内存一定比硬盘容量小得多，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。

内存缓存中有一块重要的缓存资源是preloader相关指令（例如<link rel="prefetch">）下载的资源。总所周知preloader的相关指令已经是页面优化的常见手段之一，它可以一边解析js/css文件，一边网络请求下一个资源。

(3) Disk Cache,就是存储在硬盘中的缓存，读取速度相比内存读取慢点. 在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 HTTP Herder 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。

(4) Push Cache

3.缓存机制:

强制缓存优先于协商缓存进行，若强制缓存(Expires和Cache-Control)生效则直接使用缓存，若不生效则进行协商缓存.

协商缓存由服务器决定是否使用缓存，若协商缓存失效，那么代表该请求的缓存失效，返回200，重新返回资源和缓存标识，再存入浏览器缓存中；生效则返回304，继续使用缓存。