1、有关前端安全问题：sql注入、xss攻击等

2、计算机网络相关问题：http、tcp等

3、数据结构相关

4、HTML、CSS、JavaScript相关

·JavaScript 异步、函数使用、函数嵌套

·CSS 盒模型、布局、移位变化

·表单

5.Vue、React相关

目录

[通用 3](#_Toc149286675)

[浏览器引擎 3](#_Toc149286676)

[cookie、sessionStorage、localStorage区别 3](#_Toc149286677)

[回流和重绘 3](#_Toc149286678)

[URL详解 4](#_Toc149286679)

[逻辑结构和存储结构 5](#_Toc149286680)

[常见端口号 5](#_Toc149286681)

[RPC和REST 6](#_Toc149286682)

[SOAP 6](#_Toc149286683)

[OSI模型 7](#_Toc149286684)

[XMLHttpRequest 8](#_Toc149286685)

[GET和POST区别 8](#_Toc149286686)

[名词解释 8](#_Toc149286687)

[JavaScript 10](#_Toc149286688)

[常见的不支持冒泡事件 10](#_Toc149286689)

[script 标签 async 和 defer 属性 10](#_Toc149286690)

[事件传播的三个阶段 11](#_Toc149286691)

[Promise.all 和 Promise.race 11](#_Toc149286692)

[箭头函数与显式绑定 12](#_Toc149286693)

[this绑定规则优化级 12](#_Toc149286694)

[Function 和 Object 关系 13](#_Toc149286695)

[使用Promise实现限制并发请求函数 13](#_Toc149286696)

[ES6 模块与 CommonJS 模块的差异 14](#_Toc149286697)

[Function.name 15](#_Toc149286698)

[箭头函数特点 15](#_Toc149286699)

[Promise 例题 15](#_Toc149286700)

[类型转换例题 17](#_Toc149286701)

[作用域例题 18](#_Toc149286702)

[其他例题 19](#_Toc149286703)

[CSS 20](#_Toc149286704)

[white-space 20](#_Toc149286705)

[first-child 20](#_Toc149286706)

[padding 21](#_Toc149286707)

[常用库 22](#_Toc149286708)

[Axios 22](#_Toc149286709)

# 通用

## 浏览器引擎

Trident --- IE/Edge、360、猎豹、搜狗

Gecko --- Firefox

Webkit --- Safari、chrome(Blink)、猎豹、搜狗

## cookie、sessionStorage、localStorage区别

（1）存储大小：cookie数据大小不能超过4KB，sessionStorage和localStorage虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大。

（2）有效时间：localStorage存储持久数据，浏览器关闭后数据不丢失除非主动删除数据； sessionStorage数据在当前浏览器窗口关闭后自动删除；cookie设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭。

（3）数据与服务器之间的交互方式：cookie的数据会自动的传递到服务器，服务器端也可以写cookie到客户端； sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。

## 回流和重绘

在HTML中，每个元素都可以理解成一个盒子，在浏览器解析过程中，会涉及到回流与重绘：

（1）回流：布局引擎会根据各种样式计算每个盒子在页面上的大小与位置；

（2）重绘：当计算好盒模型的位置、大小及其他属性后，浏览器根据每个盒子特性进行绘制。

回流触发时机

·添加或删除可见的DOM元素

·元素的位置发生变化

·元素的尺寸发生变化（包括外边距、内边框、边框大小、高度和宽度等）

·内容发生变化，比如文本变化或图片被另一个不同尺寸的图片所替代

·页面一开始渲染的时候（这避免不了）

·浏览器的窗口尺寸变化（因为回流是根据视口的大小来计算元素的位置和大小的）

·font-size、font-family、font-weight、line-height改变也会回流

重绘触发时机

·触发回流一定会触发重绘

·颜色的修改

·文本方向的修改

·阴影的修改

减少回流和重绘的方法

·如果想设定元素的样式，通过改变元素的 class 类名 (尽可能在 DOM 树的最里层)

·避免设置多项内联样式

·避免使用 table 布局，table 中每个元素的大小以及内容的改动，都会导致整个 table 的重新计算

·对于那些复杂的动画，对其设置 position: fixed/absolute，尽可能地使元素脱离文档流，从而减少对其他元素的影响

·使用css3硬件加速，可以让transform、opacity、filters这些动画不会引起回流重绘

·避免使用 CSS 的 JavaScript 表达式

·离线操作——我们可以通过设置元素属性display: none，将其从页面上去掉，然后再进行后续操作，这些后续操作不会触发回流与重绘

## URL详解

使用 URL 组件

给指定的 URL 创建一个 URL 对象将解析 URL 并通过其属性对其组成部分的快速访问。

let addr = new URL("https://developer.mozilla.org/zh-CN/docs/Web/API/URL\_API");

let host = addr.host;

let path = addr.pathname;

修改 URL

URL对象的大部分属性都是可设置的。

如：

let myUsername = "someguy";

let addr = new URL("https://mysite.com/login");

addr.username = myUsername;

属性

hash

包含'#'的USVString，后跟 URL 的片段标识符。

host

一个USVString，其中包含域（即主机名），后跟（如果指定了端口）“：”和 URL 的端口。

hostname

包含 URL 域名的 USVString。

href

包含完整 URL 的 USVString。

origin（只读）

返回一个包含协议名、域名和端口号的 USVString。

password

包含在域名前面指定的密码的 USVString 。

pathname

以 '/' 起头紧跟着 URL 文件路径的 DOMString。

port

包含 URL 端口号的 USVString。

protocol

包含 URL 协议名的 USVString，末尾带 ':'。

search

一个USVString ，指示 URL 的参数字符串；如果提供了任何参数，则此字符串包括所有参数，并以开头的“？”开头 字符。

searchParams（只读）

URLSearchParams对象，可用于访问search中找到的各个查询参数。

username

包含在域名前面指定的用户名的 USVString。

如：

<https://AAA:BBB@cn.bing.com:8080/monkey/story?A=1&B=2#KILL%20THE%20MAN>

·origin: 'https://cn.bing.com:8080',

·protocol: 'https:',

·username: 'AAA',

·password: 'BBB',

·host: 'cn.bing.com:8080',

·hostname: 'cn.bing.com',

·port: '8080',

·pathname: '/monkey/story',

·search: '?A=1&B=2',

·searchParams: URLSearchParams { 'A' => '1', 'B' => '2' },

·hash: '#KILL%20THE%20MAN'

## 逻辑结构和存储结构

逻辑结构

数据的逻辑结构是指数据元素之间的逻辑关系，即从逻辑关系上描述数据。它与数据的存储无关，是独立于计算机的。

数据的逻辑结构分为线性结构和非线性结构。

线性表是典型的线性结构；集合、树和图是典型的非线性结构。

存储结构

存储结构是指数据结构在计算机中的表示（又称映像），包括数据元素的表示和关系的表示，也称物理结构。数据的存储结构是逻辑结构用计算机语言的实现，它依赖于计算机语言。

数据的存储结构主要有：顺序存储、链式存储、索引存储和散列存储。

## 常见端口号

21/tcp端口：FTP 文件传输服务

22/tcp端口：SSH协议、SCP（文件传输）、端口号重定向

23/tcp端口：TELNET 终端仿真服务、远程登陆

25/tcp端口：SMTP 简单邮件传输服务

53端口：DNS 域名解析服务

69/udp：TFTP

80/tcp：HTTP 超文本传送协议 (WWW)

443/tcp、443/udp：HTTPS

## RPC和REST

RPC

在分布式计算，RPC（Remote Procedure Call，远程过程调用）是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一个地址空间的子程序，而程序员就像调用本地程序一样，无需额外地为这个交互作用编程。RPC是一种服务器­-客户端模式，经典实现是一个通过发送请求-接受回应进行信息交互的系统。

使用场景

强调性能传输，期望使用时像本地方法一样使用，适用于同一个系统内的不同服务之间，比如现在的微服务架构系统。

REST

REST(Representational State Transfer，表现层状态转换）是一种万维网软件架构风格，目的是便于不同软件/程序在网络中互相传递信息。表现层状态转换是根基于超文本传输协议之上而确定的一组约束和属性，是一种设计提供万维网络服务的软件构建风格。

使用场景

对外暴露接口服务，但需要严格控制请求类型和信息格式，适合于网站对外服务。

对比

（1）RPC使用的是client/server模式，REST更倾向于browser/server；

（2）RPC 适用于内网服务调用，对外提供服务请走 REST；

（3）IO 密集的服务调用用 RPC，低频服务用 REST；

（4）服务调用过于密集与复杂，RPC 就比较适用；

（5）RPC高效低延迟，性能高；

（6）REST耦合性低，兼容性好，提高开发效率。

## SOAP

SOAP（Simple Object Access Protocol，简单对象访问协议）是Web Service中很常见的一个协议，确定了一种通过XML实现跨语言、跨机器传输调用的协议。同时，SOAP也是一种 RPC 框架。

相关概念

SOAP 封装(envelop)：定义了一个框架，描述消息中的内容是什么，是谁发送的，谁应当接受并处理。

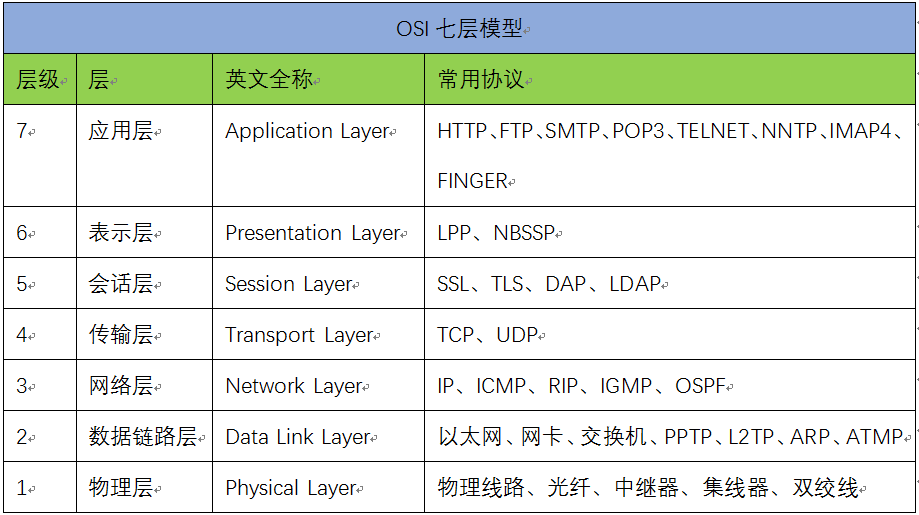
SOAP 编码规则(encoding rules)：定义了一种序列化的机制，表示应用程序需要使用的数据类型的实例。

SOAP RPC表示(RPC representation)：定义了一个协定，用于表示远程过程调用和应答。

SOAP 绑定(binding)：定义了SOAP使用哪种协议交换信息。使用HTTP/TCP/UDP协议都可以。

## OSI模型

OSI模型，即开放式通信系统互联参考模型（Open System Interconnection Reference Model），是国际标准化组织（ISO）提出的一个试图使各种计算机在世界范围内互连为网络的标准框架，简称OSI。



（1）应用层

为应用程序或用户请求提供各种请求服务。

数据传输单位：通常是消息（Message）或应用数据单元（Application Data Unit）。

（2）表示层

数据编码、格式转换、数据加密。

数据传输单位：表示数据单元（Presentation Data Unit）。

（3）会话层

创建、管理和维护会话。

数据传输单位：会话数据单元（Session Data Unit）。

（4）传输层

数据通信。

数据传输单位：报文段（Segment TCP）、数据报（Datagram UDP）或报文（Message）。

（5）网络层

IP选址及路由选择。

数据传输单位：数据包 / 分组（Packet）。

（6）数据链路层

提供介质访问和链路管理。

数据传输单位：帧（Frame）。

（7）物理层

管理通信设备和网络媒体之间的互联互通。

数据传输单位：比特（Bit）。

## XMLHttpRequest

XMLHttpRequest（XHR）对象用于与服务器交互。通过 XMLHttpRequest 可以在不刷新页面的情况下请求特定 URL，获取数据。这允许网页在不影响用户操作的情况下，更新页面的局部内容。XMLHttpRequest 在 AJAX 编程中被大量使用。

XMLHttpRequest.readyState

XMLHttpRequest.readyState 属性返回一个 XMLHttpRequest 代理当前所处的状态。一个 XHR 代理总是处于下列状态中的一个：



## GET和POST区别

GET和POST，两者是HTTP协议中发送请求的方法。

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。

两者本质上都是TCP链接，并无差别。但由于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。

（1）GET产生的URL地址可以被bookmark，而POST不可以。

（2）GET请求会被浏览器主动cache，而POST不会，除非手动设置。

（3）GET请求只能进行url编码，而POST支持多种编码方式。

（4）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。

（5）GET请求在URL中传送的参数是有长度限制的，而POST没有。

（6）对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

（7）GET比POST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。

（8）GET参数通过URL传递，POST放在Request body中。

## 名词解释

ARIA

ARIA（Accessible Rich Internet Applications，无障碍富互联网应用）是一组角色和属性，用于定义使残障人士更易于访问 web 内容和 web 应用程序（尤其是使用 JavaScript 开发的应用程序）的方法。

它补充了 HTML，以便在没有其他机制时可以将应用程序中常用的交互和小部件传递给辅助技术。

MIME

MIME (Multipurpose Internet Mail Extensions，多用途互联网邮件扩展)是当前广泛应用的一种电子邮件技术规范。

MIME 类型用来表示文档、文件或一组数据的性质和格式。在把输出结果传送到浏览器上的时候，浏览器必须启动适当的应用程序来处理这个输出文档，这可以通过多种类型MIME来完成。在HTTP中，MIME类型被定义在Content-Type header中。

MIME 类型通常仅包含两个部分：类型（type）和子类型（subtype），中间由斜杠 / 分割，中间没有空白字符。此外还有一个可选的参数，能够提供额外的信息。

如：

text/plain、application/json、font/woff、image/png

# JavaScript

## 常见的不支持冒泡事件

①focus

②blur

③mouseenter

④mouseleave

⑤load

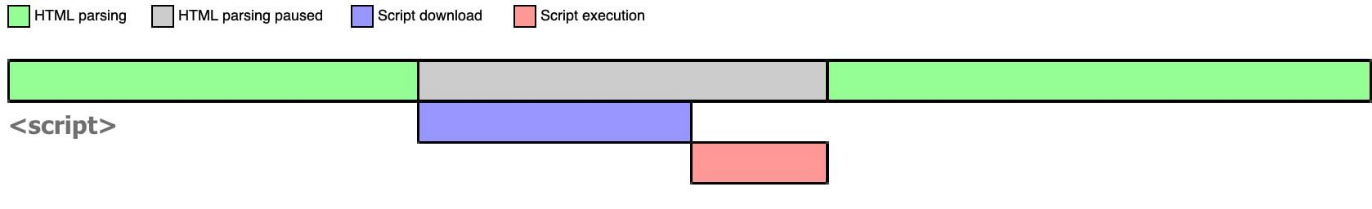
⑥unload

⑦resize

## script 标签 async 和 defer 属性

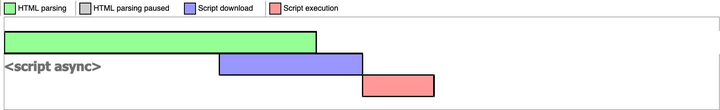
（1）<script src='xxx'></script>

浏览器在解析 HTML 的时候，如果遇到一个没有任何属性的 script 标签，就会暂停解析，先发送网络请求获取该 JS 脚本的代码内容，然后让 JS 引擎执行该代码，当代码执行完毕后恢复解析。



（2）<script src='xxx' async></script>

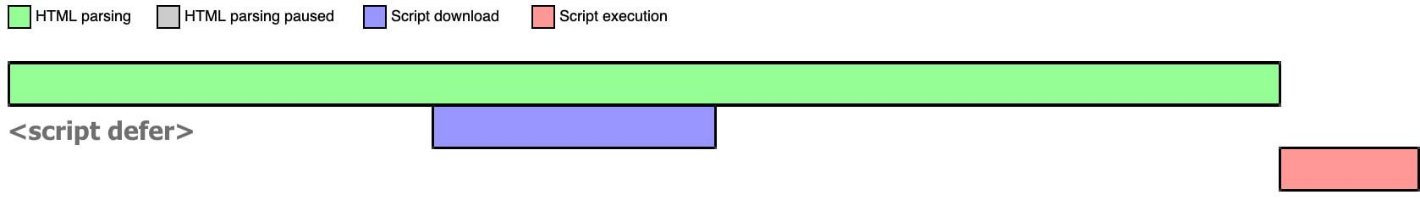
当浏览器遇到带有 async 属性的 script 时，请求该脚本的网络请求是异步的，不会阻塞浏览器解析 HTML。一旦网络请求回来之后，如果此时 HTML 还没有解析完，浏览器会暂停解析，先让 JS 引擎执行代码，执行完毕后再进行解析。



（3）<script src='xxx' defer></script>

当浏览器遇到带有 defer 属性的 script 时，获取该脚本的网络请求也是异步的，不会阻塞浏览器解析 HTML。一旦网络请求回来之后，如果此时 HTML 还没有解析完，浏览器不会暂停解析并执行 JS 代码，而是等待 HTML 解析完毕再执行 JS 代码。

当浏览器遇到带有 defer 属性的 script 时，获取该脚本的网络请求也是异步的，不会阻塞浏览器解析 HTML，一旦网络请求回来之后，如果此时 HTML 还没有解析完，浏览器不会暂停解析并执行 JS 代码，而是等待 HTML 解析完毕再执行 JS 代码



## 事件传播的三个阶段

事件传播是JS DOM事件中各阶段的概括, 由三个阶段组成:

（1）捕获阶段 capture phase

事件从 window 对象逐级向下传播, 直到目标对象的父节点的过程

（2）目标阶段 target phase

事件到达目标节点自身

（3）冒泡阶段 bubble phase

事件从目标的父节点逐级向上传播, 一直到达 window 对象的过程

## Promise.all 和 Promise.race

Promise.all

Promise.all() 静态方法接受一个 Promise 可迭代对象作为输入，并返回一个 Promise。当所有输入的 Promise 都被兑现时，返回的 Promise 也将被兑现（即使传入的是一个空的可迭代对象），并返回一个包含所有兑现值的数组。如果输入的任何 Promise 被拒绝，则返回的 Promise 将被拒绝，并带有第一个被拒绝的原因。

Promise.race

Promise.race() 静态方法接受一个 promise 可迭代对象作为输入，并返回一个 Promise。这个返回的 promise 会随着第一个 promise 的敲定而敲定。

两者传入空数组

Promise.all([]).then((res) => {

console.log('all');

});

Promise.race([]).then((res) => {

console.log('race');

});

结果：all 会被输出，而 race 不会被输出

原因：Promise.all 会立即返回 resolved 状态，因而会立马输出，而 Promise.race 则一直处于 pending 状态，不会走到 then，所以永远不会输出。

## 箭头函数与显式绑定

题目：

function fun () {

return () => {

return () => {

return () => {

console.log(this.name)

}

}

}

}

var f = fun.call({name: 'foo'})

var t1 = f.call({name: 'bar'})()()

var t2 = f().call({name: 'baz'})()

var t3 = f()().call({name: 'qux'})

答案：

foo

foo

foo

解析：

（1）箭头函数没有 this，它的 this 是通过作用域链查到外层作用域的 this，且指向函数定义时的 this 而非执行时。

（2）箭头函数，不能通过 call\apply\bind 来修改 this 指向，但可以通过修改外层作用域的 this 来达成间接修改。

（3）JavaScript 是静态作用域，即函数的作用域在函数定义的时候就决定了，而箭头函数的 this 是通过作用域链查到的，因此箭头函数定义后，它的作用域链就定死了。

f = fun.call({name: 'foo'}): 将 fun 函数的 this 指向 {name: 'foo'}，并返回一个箭头函数，因此箭头函数的 this 也指向 {name: 'foo'}

t1 = f.call({name: 'bar'})()(): 对第一层箭头函数执行 call 操作，无效，当前 this 仍指向 {name: 'foo'}，第二层、第三层都是箭头函数，第三层的 this 也指向 {name: 'foo'}，打印 foo

后续 t2 t3 分别对第二层、第三层箭头函数使用 call，无效，最终都打印 foo。

## this绑定规则优化级

默认绑定 < 隐式绑定 < 显式绑定 < new绑定

先查看函数的调用位置，然后再通过绑定规则来判定this指向：

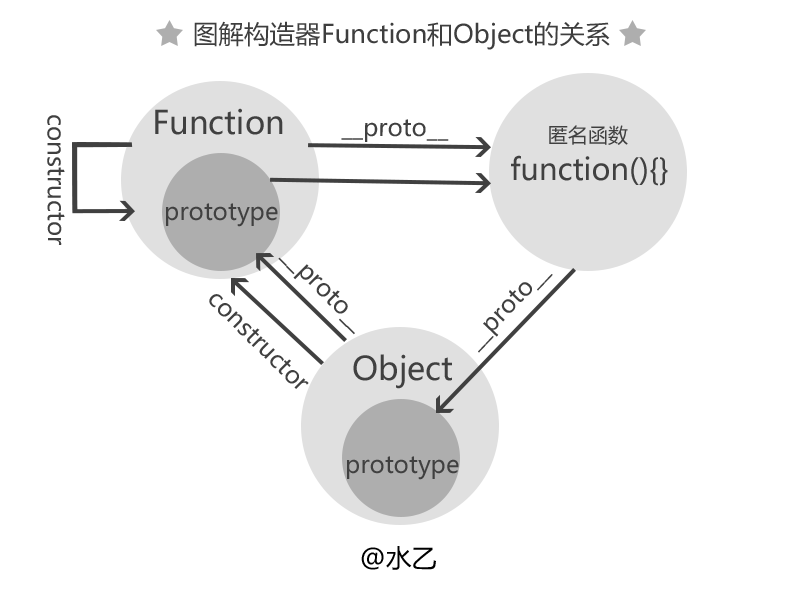
（1）函数如在new中调用，则函数中this绑定的就是新创建的对象。

（2）函数如通过 call/apply 或者 硬绑定（bind）调用，则this绑定是就是指定的对象。

（3）函数如果是在某个上下文对象中调用，即隐式绑定，则函数中this绑定的就是那个上下文对象。

（4）如果以上情况都不是的话，就是默认绑定，这里分两种情况——如处于严格模式，则this被绑定到undefined上；如处于非严格模式，则this绑定到全局对象上。

## Function 和 Object 关系



console.log(Function instanceof Object, Object instanceof Function); // true true

Function.\_\_proto\_\_ === Function.prototype

Object.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype

## 使用Promise实现限制并发请求函数

（1）使用Promise

/\*\*

\* arrs 请求数据源数组

\* limit 是每次并行发起多少个请求

\* handleFn 就是异步处理函数

\*/

function limitQueueFn(arrs, limit, handleFn) {

// 完成任务数

let index = 0;

// 第一次的时候 一次性执行 limit 个任务

for (let i = 0; i < limit; i++) run();

// 执行一个任务

function run() {

// 构造待执行任务 当该任务完成后 如果还有待完成的任务 继续执行任务

new Promise((resolve, reject) => {

const value = arrs[index];

index++; // 这个是同步操作

// resolve 返回 promise

resolve(handleFn(value))

}).then(() => {

if (index < arrs.length) {

run()

}

})

}

};

（2）使用Promise.all

async function asyncPool(poolLimit, array, iteratorFn) {

const ret = [];

const executing = [];

for (const item of array) {

const p = Promise.resolve().then(() => iteratorFn(item));

ret.push(p);

if (poolLimit <= array.length) {

const e = p.then(() => executing.splice(executing.indexOf(e), 1));

executing.push(e);

if (executing.length >= poolLimit) {

await Promise.race(executing);

}

}

}

return Promise.all(ret);

}

## ES6 模块与 CommonJS 模块的差异

差异

（1）CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

（2）CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

（3）CommonJS 模块的require()是同步加载模块，ES6 模块的import命令是异步加载，有一个独立的模块依赖的解析阶段。

详细

（1）第一个差异

CommonJS 模块输出的是值的拷贝，一旦输出一个值，模块内部的变化就影响不到这个值。除非写成一个函数，才能得到内部变动后的值。对于原始类型的值，CommonJS 模块会缓存运行结果。

ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。有点像 Unix 系统的“符号连接”，原始值变了，import加载的值也会跟着变。ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

（2）第二个差异

因为 CommonJS 加载的是一个对象（即module.exports属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

## Function.name

Function 实例的 name 数据属性表示函数在创建时指定的名称，或者如果函数是匿名函数，则名称可以是 anonymous 或 ''（空字符串）。

如：

const a = function () {

console.log("a");

};

console.log(a.name);

输出 a

函数的 name 属性可用于在调试工具或错误消息中标识该函数。它对语言本身没有任何意义。name 属性是只读的，不能用赋值运算符修改。

## 箭头函数特点

·函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象；

·无法当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误；

·无法使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替；

·不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

## Promise 例题

（1）

const p1 = new Promise((resolve, reject) => {

console.log(1);

setTimeout(() => {

resolve(6);

console.log(2);

}, 3000)

})

const p2 = new Promise((resolve, reject) => {

console.log(3);

setTimeout(() => {

resolve(p1);

console.log(4);

}, 1000)

})

p1.then(result => {console.log(5)});

p2.then(result => {console.log(result)});

输出：1 3 4 2 5 6

如果resolve参数是一个promise实例，那么直到该promise实例改变状态，其本身才相应改变。

（2）

async function async1() {

console.log('1');

await async2();

console.log('2');

}

async function async2() {

console.log('3');

}

console.log('4');

setTimeout(function () {

console.log('5');

}, 0)

async1();

new Promise(function (resolve) {

console.log('6');

resolve();

}).then(function () {

console.log('7');

});

输出：4 1 3 6 2 7 5

执行 async1() 输出 async1 start，然后执行 async2()，输出 async2，把 async2() 后面的代码 console.log('async1 end') 放到微任务队列中。

（3）

console.log('start');

setTimeout(() => {

console.log('children2');

Promise.resolve().then(() => {

console.log('children3');

});

}, 0);

new Promise(function (resolve, reject) {

console.log('children4');

setTimeout(function () {

console.log('children5');

resolve('children6')

}, 0)

}).then((res) => {

console.log('children7');

setTimeout(() => {

console.log(res);

}, 0)

})

输出：start children4 children2 children3 children5 children7 children6

（4）

(async () => {

console.log(1);

setTimeout(() => {

console.log(2);

}, 0);

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

console.log(3);

}).then(() => {

console.log(4);

});

console.log(5);

})();

输出：1 3 2

没有 4、5，因为Promise状态为pending

## 类型转换例题

（1）[] = = ![] // true

原因：

Boolean([]) -> true

Boolean(![]) -> false

Number(false) -> 0

Number([]) -> 0

0 == 0 -> true

（2）[1] + [2] – 3 // 9

## 作用域例题

（1）

const bar = 1;

function foo() {

console.log(bar); // A

const bar = 2;

console.log(bar); // B

}

foo();

报错：ReferenceError: Cannot access 'bar' before initialization

（2）

// 或者 const bar = 1 在这个位置

function foo() {

console.log(bar); // A

var bar = 2;

console.log(bar); // B

}

const bar = 1;

foo();

输出：undefined 2

（3）

var length = 10;

function fn() {

console.log(this.length);

}

var obj = {

length: 5,

method: function (fn) {

fn(); // 10

arguments[0](); // 2

},

};

obj.method(fn, 1);

浏览器环境输出：10 2

（4）

var a = 1;

let obj = {

a: 2,

fn1: () => {

console.log(this.a);

},

fn2() {

console.log(this.a);

},

};

obj.fn1();

obj.fn2();

let tfn1 = obj.fn1;

let tfn2 = obj.fn2;

tfn1();

tfn2();

浏览器环境输出：1 2 1 1

（5）

var a = 10;

(function () {

console.log(a)

a = 5

console.log(window.a)

var a = 20;

console.log(a)

})()

浏览器环境输出：undefined 10 20

（6）

var a = 0;

var b = 0;

var c = 0;

function fn(a) {

console.log(a++, c);

function fn2(b) {

console.log(a, b, c);

}

var c = 4;

fn = fn2;

}

fn(1);

fn(2);

console.log(a, b, c);

// 1 undefined

// 2 2 4

// 0 0 0

## 其他例题

（1）

const arr = ["a", , "b", ,];

console.log(arr.length);

输出：4

（2）

console.log(String(true) + 10);

console.log(true + 10);

输出：true10 11

（3）

console.log(typeof null);

console.log(null instanceof Object);

输出：object false

# CSS

## white-space

CSS white-space 属性用于设置如何处理元素内的空白字符。

这个属性指定了两件事：

（1）空白字符是否合并，以及如何合并。

（2）是否换行，以及如何换行。

white-space 属性可以被指定为从下面的值列表中选择的单个关键字，或者是表示 white-space-collapse 和 text-wrap 属性的简写的两个值。

属性值如下：

normal

连续的空白符会被合并。源码中的换行符会被当作空白符来处理。并根据填充行框盒子的需要来换行。

nowrap

和 normal 一样合并空白符，但阻止源码中的文本换行。

pre

连续的空白符会被保留。仅在遇到换行符或 <br> 元素时才会换行。

pre-wrap

连续的空白符会被保留。在遇到换行符或 <br> 元素时，或者根据填充行框盒子的需要换行。

pre-line

连续的空白符会被合并。在遇到换行符或 <br> 元素时，或者根据填充行框盒子的需要换行。

break-spaces

与 pre-wrap 的行为相同，除了：

·任何保留的空白序列总是占用空间，包括行末的。

·每个保留的空白字符后（包括空白字符之间）都可以被截断。

·这样保留的空间占用空间而不会挂起，从而影响盒子的固有尺寸（最小内容——min-content——大小和最大内容——max-content——大小）。

## first-child

:first-child CSS 伪类表示在一组兄弟元素中的第一个元素。

如：

HTML：

<div>

<p>此文本已选中！</p>

<p>此文本未选中。</p>

</div>

<div>

<h2>此文本未被选中：它不是一个 `p`。</h2>

<p>此文本未被选中。</p>

</div>

CSS：

p:first-child {

color: lime;

background-color: black;

padding: 5px;

}

## padding

padding CSS 简写属性控制元素所有四条边的内边距区域。

padding 属性接受 1~4 个值，每个值可以是 <length> 或 <percentage>：

（1）当只指定一个值时，该值会统一应用到全部四个边的内边距上。

（2）指定两个值时，第一个值会应用于上边和下边的内边距，第二个值应用于左边和右边。

（3）指定三个值时，第一个值应用于上边，第二个值应用于右边和左边，第三个则应用于下边的内边距。

（4）指定四个值时，依次（顺时针方向）作为上边，右边，下边，和左边的内边距。

注意：padding取值不能为负。

# 常用库

## Axios

Axios 是一个基于 promise 网络请求库，作用于node.js 和浏览器中。 它是 isomorphic 的(即同一套代码可以运行在浏览器和node.js中)。在服务端它使用原生 node.js http 模块, 而在客户端 (浏览端) 则使用 XMLHttpRequests。

特性

·从浏览器创建 XMLHttpRequests

·从 node.js 创建 http 请求

·支持 Promise API

·拦截请求和响应

·转换请求和响应数据

·取消请求

·自动转换JSON数据

·客户端支持防御XSRF

拦截请求和响应

在请求或响应被 then 或 catch 处理前拦截它们。

如：

// 添加请求拦截器

axios.interceptors.request.use(function (config) {

// 在发送请求之前做些什么

return config;

}, function (error) {

// 对请求错误做些什么

return Promise.reject(error);

});

// 添加响应拦截器

axios.interceptors.response.use(function (response) {

// 2xx 范围内的状态码都会触发该函数。

// 对响应数据做点什么

return response;

}, function (error) {

// 超出 2xx 范围的状态码都会触发该函数。

// 对响应错误做点什么

return Promise.reject(error);

});

移除拦截器

如：

const myInterceptor = axios.interceptors.request.use(function () {/\*...\*/});

axios.interceptors.request.eject(myInterceptor);

取消请求

CancelToken(deprecated)

如：

const CancelToken = axios.CancelToken;

const source = CancelToken.source();

axios.get('/user/12345', {

cancelToken: source.token

}).catch(function (thrown) {

if (axios.isCancel(thrown)) {

console.log('Request canceled', thrown.message);

} else {

// 处理错误

}

});

axios.post('/user/12345', {

name: 'new name'

}, {

cancelToken: source.token

})

// 取消请求（message 参数是可选的）

source.cancel('Operation canceled by the user.');

AbortController

如：

const controller = new AbortController();

axios.get('/foo/bar', {

signal: controller.signal

}).then(function(response) {

//...

});

// 取消请求

controller.abort()