目录

基础篇之JavaScript 部分

1. JavaScript 有哪些数据类型，它们的区别？

JavaScript 共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是ES6 中新增的数据类型：

●Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。

●BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数， 使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了Number能够表示的安全整数范围。

这些数据可以分为原始数据类型和引用数据类型：

●栈：原始数据类型（Undefined、Null、Boolean、Number、String）

●堆：引用数据类型（对象、数组和函数） 两

种类型的区别在于存储位置的不同：

●原始数据类型直接存储在栈（stack）中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储；

●引用数据类型存储在堆（heap）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。堆和栈的概念存在于数据结构和操作系统内存中，在数据结构中：

●在数据结构中，栈中数据的存取方式为先进后出。

●堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。

在操作系统中，内存被分为栈区和堆区：

●栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

●堆区内存一般由开发着分配释放，若开发者不释放，程序结束时可能由垃圾回收机制回收。

2. 数据类型检测的方式有哪些

（1） typeof

```
console.log(typeof 2);              // number
console.log(typeof true);           // boolean
console.log(typeof 'str');          // string
console.log(typeof []);             // object
console.log(typeof function(){});   // function
console.log(typeof {});             // object
console.log(typeof undefined);      // undefined
console.log(typeof null);           // object
```

其中数组、对象、null 都会被判断为object，其他判断都正确。

（2） instanceof

instanceof 可以正确判断对象的类型，其内部运行机制是判断在其原型链中能否找到该类型的原型。

```
console.log(2 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);                // false

console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

可以看到，instanceof 只能正确判断引用数据类型，而不能判断基本数据类型。instanceof 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。

（3） constructor

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
```

constructor 有两个作用，一是判断数据的类型，二是对象实例通过 constrcutor 对象访问它的构造函数。需要注意，如果创建一个对象来改变它的原型，constructor 就不能用来判断数据类型了：

```
function Fn(){};

Fn.prototype = new Array();

var f = new Fn();

console.log(f.constructor===Fn);    // false
console.log(f.constructor===Array); // true
```

（4）Object.prototype.toString.call()

Object.prototype.toString.call() 使用 Object 对象的原型方法 toString 来判断数据类型：

```
var a = Object.prototype.toString;

console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
```

同样是检测对象obj 调用toString 方法，obj.toString()的结果和 Object.prototype.toString.call(obj) 的结果不一样， 这是为什么？

这是因为toString 是Object 的原型方法，而 Array、function 等类型作为Object 的实例，都重写了 toString 方法。不同的对象类型调用 toString 方法时，根据原型链的知识，调用的是对应的重写之后

的 toString方法（function 类型返回内容为函数体的字符串，Array 类型返回元素组成的字符串…），而不会去调用 Object 上原型 toString 方法（返回对象的具体类型），所以采用 obj.toString() 不能得到其对象类型，只能将 obj 转换为字符串类型；因此，在想要得到对象的具体类型时，应该调用Object 原型上的toString 方法。

3. null 和undefined 区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

undefined 代表的含义是未定义，null 代表的含义是空对象。一般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。

undefined 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断。我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回“object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

4. intanceof 操作符的实现原理及实现

instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

```
function myInstanceof(left, right) {
  // 获取对象的原型
  let proto = Object.getPrototypeOf(left)
  // 获取构造函数的 prototype 对象
  let prototype = right.prototype;

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;
    // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf方法用来获取指定对象的原型
    proto = Object.getPrototypeOf(proto);
  }
}
```

5. 如何获取安全的 undefined 值？

因为 undefined 是一个标识符,所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。表达式 void____没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。因此可以用 void 0 来获得 undefined。

6. Object.is() 与比较操作符 “===”、“==” 的区别？

使用双等号（==）进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。

使用三等号（===）进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。

使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

7. 什么是 JavaScript 中的包装类型？

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象，如：

```
const a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"
```

在访问'abc'.length 时， JavaScript 将'abc' 在后台转换成String('abc')，然后再访问其length 属性。

JavaScript 也可以使用Object 函数显式地将基本类型转换为包装类型：

```
var a = 'abc'
Object(a) // String {"abc"}
```

也可以使用valueOf 方法将包装类型倒转成基本类型：

```
var a = 'abc'
var b = Object(a)
var c = b.valueOf() // 'abc'
```

看看如下代码会打印出什么：

```
var a = new Boolean( false );
if (!a) {
  console.log( "Oops" ); // never runs
}
```

答案是什么都不会打印，因为虽然包裹的基本类型是 false，但是 false 被包裹成包装类型后就成了对象，所以其非值为 false，所以循环体中的内容不会运行。

8. 为什么会有 BigInt 的提案?

JavaScript 中Number.MAX_SAFE_INTEGER 表示最大安全数字，计算结果是 9007199254740991，即在这个数范围内不会出现精度丢失（小数除外）。但是一旦超过这个范围，js 就会出现计算不准确的情况，
这在大数计算的时候不得不依靠一些第三方库进行解决,因此官方提出了 BigInt 来解决此问题。

9. 如何判断一个对象是空对象

使用JSON 自带的.stringify 方法来判断：

```
if(Json.stringify(Obj) == '{}' ){
    console.log('空对象');
}
```

使 用 ES6 新增的方法Object.keys()来判断：

```
if(Object.keys(Obj).length < 0){
    console.log('空对象');
}
```

10. const 对象的属性可以修改吗

const 保证的并不是变量的值不能改动，而是变量指向的那个内存地址不能改动。对于基本类型的数据（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。

但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const 只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

11. 如果new 一个箭头函数的会怎么样

箭头函数是ES6中的提出来的，它没有prototype，也没有自己的this指向，更不可以使用arguments 参数，所以不能New 一个箭头函数。

new 操作符的实现步骤如下：

1.创建一个对象

2.将构造函数的作用域赋给新对象（也就是将对象的__proto__属性指向构造函数的 prototype属性）

3. 指向构造函数中的代码，构造函数中的 this 指向该对象（也就是为这个对象添加属性和方法）

4. 返回新的对象

所以，上面的第二、三步，箭头函数都是没有办法执行的。

12. 箭头函数的this 指向哪里？

箭头函数不同于传统JavaScript 中的函数，箭头函数并没有属于自己的 this，它所谓的this 是捕获其所在上下文的 this 值，作为自己的 this 值，并且由于没有属于自己的 this，所以是不会被 new 调用的，这个所谓的this 也不会被改变。

可以用Babel 理解一下箭头函数:

```
// ES6
const obj = {
  getArrow() {
    return () => {
      console.log(this === obj);
    };
  }
}
```

转化后：

```
// ES5, 由 Babel 转译
var obj = {
  getArrow: function getArrow() {
    var _this = this;
    return function () {
      console.log(_this === obj);
    };
  }
};
```

13. 扩展运算符的作用及使用场景

（1）对象扩展运算符

对象的扩展运算符(...)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中。

```
let bar = { a: 1, b: 2 };
let baz = { ...bar }; // { a: 1, b: 2 }
```

上述方法实际上等价于：

```
let bar = { a: 1, b: 2 };
let baz = Object.assign({}, bar); // { a: 1, b: 2 }
```

Object.assign 方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。Object.assign 方法的第一个参数是目标对象，后面的参数都是源对象。(如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性)。

同样，如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```
let bar = {a: 1, b: 2};
let baz = {...bar, ...{a:2, b: 4}};  // {a: 2, b: 4}
```

利用上述特性就可以很方便的修改对象的部分属性。在 redux 中的 reducer 函数规定必须是一个纯函数，reducer 中的state 对象要求不能直接修改，可以通过扩展运算符把修改路径的对象都复制一遍， 然后产生一个新的对象返回。

需要注意：扩展运算符对对象实例的拷贝属于浅拷贝。

（2） 数组扩展运算符

数组的扩展运算符可以将一个数组转为用逗号分隔的参数序列，且每次只能展开一层数组。

```
console.log(...[1, 2, 3])
// 1 2 3
console.log(...[1, [2, 3, 4], 5])
// 1 [2, 3, 4] 5
```

下面是数组的扩展运算符的应用：

将数组转换为参数序列

```
function add(x, y) {
  return x + y;
}
const numbers = [1, 2];
add(...numbers) // 3
```

复制数组

```
const arr1 = [1, 2];
const arr2 = [...arr1];
```

要记住：扩展运算符(…)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中，这里参数对象是个数组，数组里面的所有对象都是基础数据类型，将所有基础数据类型重新拷贝到新的数组中。

合并数组

如果想在数组内合并数组，可以这样：

```
const arr1 = ['two', 'three'];
const arr2 = ['one', ...arr1, 'four', 'five'];
// ["one", "two", "three", "four", "five"]
```

扩展运算符与解构赋值结合起来，用于生成数组

```
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]
```

需要注意：如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...rest, last] = [1, 2, 3, 4, 5];         // 报错
const [first, ...rest, last] = [1, 2, 3, 4, 5];  // 报错
```

将字符串转为真正的数组

```
[...'hello']    // [ "h", "e", "l", "l", "o" ]
```

任何 Iterator 接口的对象，都可以用扩展运算符转为真正的数组

比较常见的应用是可以将某些数据结构转为数组：

```
// arguments对象
function foo() {
  const args = [...arguments];
}
```

用于替换es5 中的Array.prototype.slice.call(arguments)写法。

使用Math 函数获取数组中特定的值

```
const numbers = [9, 4, 7, 1];
Math.min(...numbers); // 1
Math.max(...numbers); // 9
```

14. Proxy 可以实现什么功能？

在 Vue3.0 中通过 Proxy 来替换原本的 Object.defineProperty 来实现数据响应式。

Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

```
let p = new Proxy(target, handler)
```

代表需要添加代理的对象，handler 用来自定义对象中的操作，比如可以用来自定义 set 或者 get 函数。

下面来通过 Proxy 来实现一个数据响应式：

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    },
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
    }
  }
  return new Proxy(obj, handler)
}
let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`'${property}' = ${target[property]}`)
  }
)
p.a = 2 // 监听到属性a改变
p.a // 'a' = 2
```

在上述代码中，通过自定义 set 和 get 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了在对对象任何属性进行读写时发出通知。

当然这是简单版的响应式实现，如果需要实现一个 Vue 中的响应式， 需要在 get 中收集依赖，在 set 派发更新，之所以 Vue3.0 要使用Proxy 替换原本的 API 原因在于 Proxy 无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 Proxy 可以完美监听到任何方式的数据改变，唯一缺陷就是浏览器的兼容性不好。

15. 常用的正则表达式有哪些？

```
// (1) 匹配 16 进制颜色值
var regex = /#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g;

// (2) 匹配日期，如 yyyy-mm-dd 格式
var regex = /^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/;

// (3) 匹配 qq 号
var regex = /^[1-9][0-9]{4,10}$/g;

// (4) 手机号码正则
var regex = /^1[34578]\d{9}$/g;

// (5) 用户名正则
var regex = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4,16}$/;
```

16. 对 JSON 的理解

JSON 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。

在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列化为

JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，

JSON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。

JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问。

17. JavaScript 脚本延迟加载的方式有哪些？

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

defer 属性：给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性

的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。

async 属性：给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。

动态创建 DOM 方式：动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入js 脚本。

使用 setTimeout 延迟方法：设置一个定时器来延迟加载 js 脚本文件

让 JS 最后加载：将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

18. 什么是 DOM 和 BOM?

DOM 指的是文档对象模型，它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。

BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待， 这个对象主要定义了与浏览器进行交互的法和接口。BOM 的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方 法存在。window 对象含有 location 对象、navigator 对象、screen

对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

19. escape、encodeURI、encodeURIComponent 的区别

encodeURI 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。

encodeURIComponent 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。

escape 和 encodeURI 的作用相同，不过它们对于 unicode 编码为 0xff 之外字符的时候会有区别，escape 是直接在字符的 unicode 编码前加上 %u,而 encodeURI 首先会将字符转换为 UTF-8 的格式，再在每个字节前加上 %。

20. 对 AJAX 的理解，实现一个 AJAX 请求

AJAX 是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

创建AJAX 请求的步骤：

创建一个 XMLHttpRequest 对象。

在这个对象上使用 open 方法创建一个 HTTP 请求,open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。

在发起请求前，可以为这个对象添加一些信息和监听函数。比如说可以通过 setRequestHeader 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发onreadystatechange 事件，可以

通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 response 中的数据来对页面进行更新了。

当对象的属性和监听函数设置完成后，最后调用 sent 方法来向服务器发起请求，可以传入参数作为发送的数据体。

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", url, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};
// 设置请求头信息
xhr.responseType = "json";
xhr.setRequestHeader("Accept", "application/json");
// 发送 Http 请求
xhr.send(null);
```

使用Promise 封装AJAX：

```
// promise 封装实现：
function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();
    // 新建一个 http 请求
    xhr.open("GET", url, true);
    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      // 当请求成功或失败时，改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    // 设置错误监听函数
    xhr.onerror = function() {
      reject(new Error(this.statusText));
    };
    // 设置响应的数据类型
    xhr.responseType = "json";
    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");
    // 发送 http 请求
    xhr.send(null);
  });
  return promise;
}
```

21. 什么是尾调用，使用尾调用有什么好处？

尾调用指的是函数的最后一步调用另一个函数。代码执行是基于执行栈的，所以当在一个函数里调用另一个函数时，会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话， 因为已经是函数的最后一步，所以这时可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。但是 ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

22. ES6 模块与 CommonJS 模块有什么异同？

ES6 Module 和CommonJS 模块的区别：

CommonJS 是对模块的浅拷贝，ES6 Module 是对模块的引用，即 ES6 Module 只存只读，不能改变其值，也就是指针指向不能变，类似const；

import 的接口是read-only（只读状态），不能修改其变量值。 即不能修改其变量的指针指向，但可以改变变量内部指针指向，可以对commonJS 对重新赋值（改变指针指向），但是对ES6 Module 赋值会编译报错。

ES6 Module 和CommonJS 模块的共同点：

CommonJS 和ES6 Module 都可以对引入的对象进行赋值，即对对象内部属性的值进行改变。

23. for...in 和for...of 的区别

for…of 是 ES6 新增的遍历方式，允许遍历一个含有 iterator 接口的数据结构（数组、对象等）并且返回各项的值，和ES3 中的for… in 的区别如下

for…of 遍历获取的是对象的键值，for…in 获取的是对象的键名；

for… in 会遍历对象的整个原型链，性能非常差不推荐使用，而 for … of 只遍历当前对象不会遍历原型链；

对于数组的遍历，for…in 会返回数组中所有可枚举的属性(包括原型链上可枚举的属性)，for…of 只返回数组的下标对应的属性值；

总结：for...in 循环主要是为了遍历对象而生，不适用于遍历数组；for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象。

24. ajax、axios、fetch 的区别

（1）AJAX

Ajax 即“AsynchronousJavascriptAndXML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。它是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点如下：

本身是针对MVC 编程，不符合前端MVVM 的浪潮

基于原生XHR 开发，XHR 本身的架构不清晰

不符合关注分离（Separation of Concerns）的原则

配置和调用方式非常混乱，而且基于事件的异步模型不友好。

（2）Fetch

fetch 号称是 AJAX 的替代品，是在 ES6 出现的，使用了 ES6 中的 promise 对象。Fetch 是基于 promise 设计的。Fetch 的代码结构比

起 ajax 简单多。fetch 不是ajax 的进一步封装，而是原生 js，没有使用 XMLHttpRequest 对象。

fetch 的优点：

语法简洁，更加语义化

基于标准 Promise 实现，支持 async/await

更加底层，提供的API 丰富（request，response）脱

离了XHR，是ES 规范里新的实现方式

fetch 的缺点：

fetch 只对网络请求报错，对 400，500 都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。

fetch 默认不会带 cookie ， 需要添加配置项： fetch(url, {credentials: 'include'})

fetch 不支持 abort ， 不支持超时控制， 使用 setTimeout 及 Promise.reject 的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费

fetch 没有办法原生监测请求的进度，而XHR 可以

（3） Axios

Axios 是一种基于Promise 封装的HTTP 客户端，其特点如下：

浏览器端发起XMLHttpRequests 请求

node 端发起http 请求

支 持 Promise API

监听请求和返回

对请求和返回进行转化

取消请求

自动转换json 数据

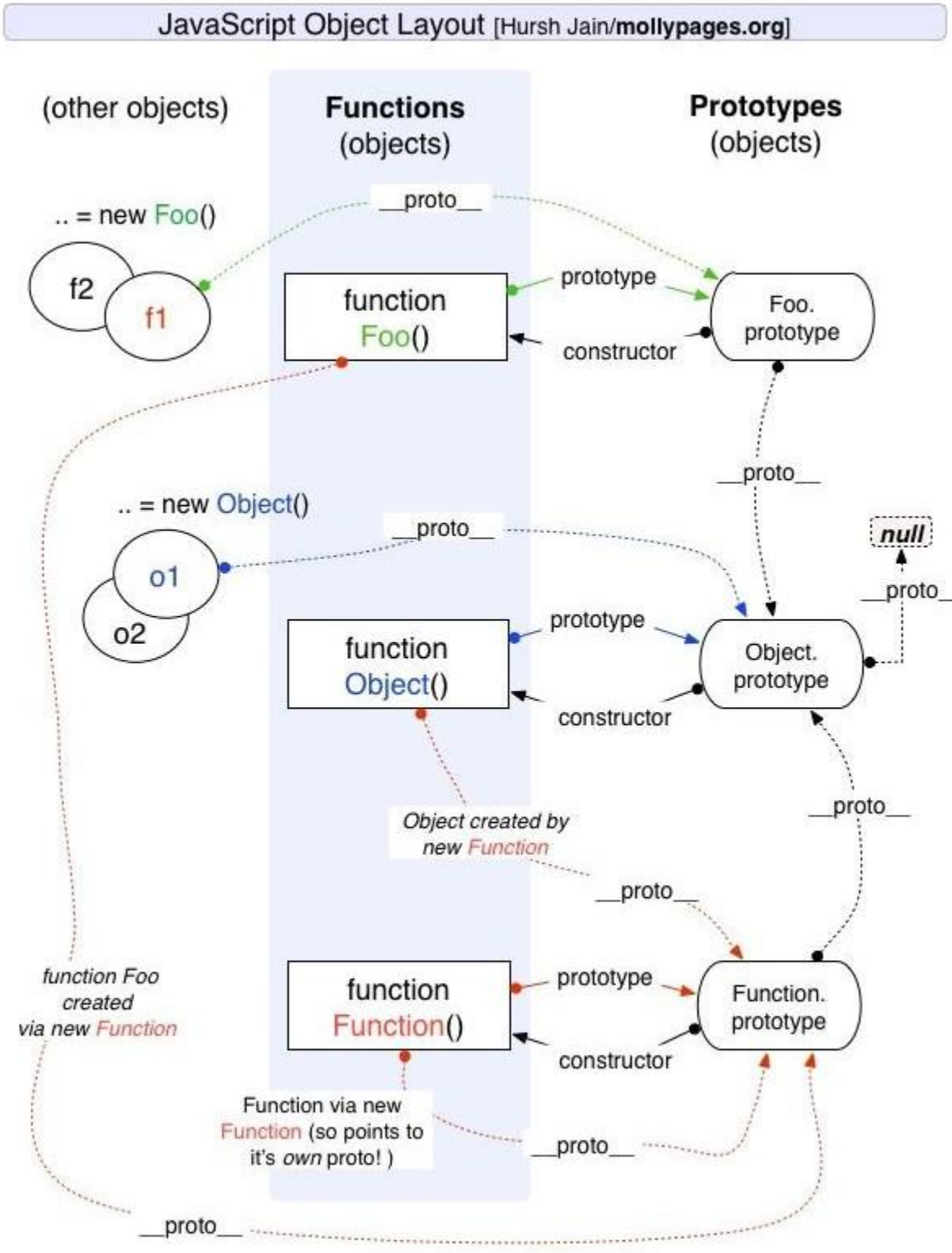
客户端支持抵御XSRF 攻击

25. 对原型、原型链的理解

在 JavaScript 中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说不应该能够获取到这个值的，但是现在浏览器中都实现了 proto_ 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

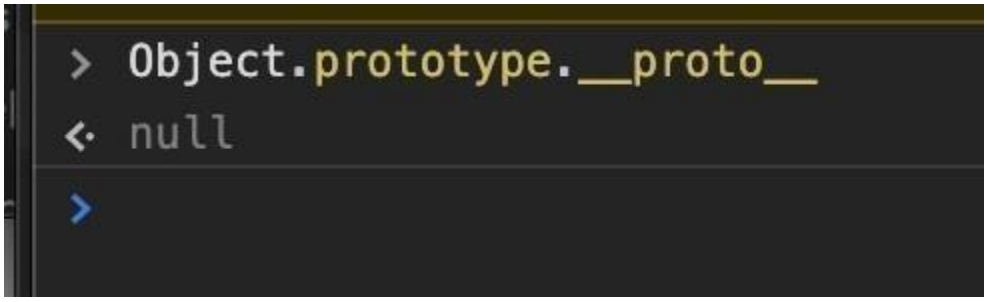
当访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。

特点：JavaScript 对象是通过引用来传递的，创建的每个新对象实体中并没有一份属于自己的原型副本。当修改原型时，与之相关的对象也会继承这一改变。



26. 原型链的终点是什么？如何打印出原型链的终点？

由于 Object是构造函数原型链终点Object.prototype.____proto__，而 Object.prototype._proto__=== null // true，所以，原型链的终点是 null。原型链上的所有原型都是对象，所有的对象最终都 是由 Object构造的，而 Object.prototype的下一级是 Object.prototype.__proto__。



27. 对作用域、作用域链的理解

1）全局作用域和函数作用域

（1） 全局作用域

最外层函数和最外层函数外面定义的变量拥有全局作用域

所有未定义直接赋值的变量自动声明为全局作用域

所有window 对象的属性拥有全局作用域

全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突。

（2） 函数作用域

函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到

作用域是分层的，内层作用域可以访问外层作用域，反之不行2）

块级作用域

使用 ES6 中新增的let 和const 指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由{ }包裹的代码片段）

let 和const 声明的变量不会有变量提升，也不可以重复声明

在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部。

作用域链：

在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到 window 对象就被终止， 这一层层的关系就是作用域链。

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当查找一个变量时，如果当前执行环境中没有找到，可以沿着作用域链向后查找。

28. 对 this 对象的理解

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模式来判断。

第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。

第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时， this 指向这个对象。

第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数： 一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数， 第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对 象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

这四种方式，使用构造器调用模式的优先级最高，然后是 applycall 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

29. call() 和 apply() 的区别？

它们的作用一模一样，区别仅在于传入参数的形式的不同。

apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向， 第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类 数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。

call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

30. 异步编程的实现方式？

JavaScript 中的异步机制可以分为以下几种：

回调函数 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

generator 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

async 函数 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个await语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

31. 对 Promise 的理解

Promise 是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息，他的出现大大改善了异步编程的困境，避免了地狱回调， 它比传统的解决方案回调函数和事件更合理和更强大。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API， 各种异步操作都可以用同样的方法进行处理。

（1） Promise 的实例有三个状态:

Pending（进行中）

Resolved（已完成）

Rejected（已拒绝）

当把一件事情交给promise 时，它的状态就是Pending，任务完成了状态就变成了Resolved、没有完成失败了就变成了Rejected。

（2） Promise 的实例有两个过程：

pending -> fulfilled : Resolved（已完成）

pending -> rejected：Rejected（已拒绝）

注意：一旦从进行状态变成为其他状态就永远不能更改状态了。

Promise 的特点：

对象的状态不受外界影响。promise 对象代表一个异步操作，有三种状态，pending（进行中）、fulfilled（已成功）、rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，这也是 promise 这个名字的由来——“承诺”；

一旦状态改变就不会再变，任何时候都可以得到这个结果。promise 对象的状态改变，只有两种可能：从 pending 变为 fulfilled，从 pending 变为rejected。这时就称为 resolved（已定型）。如果改变已经发生了，你再对 promise 对象添加回调函数，也会立即得到这个结果。这与事件（event）完全不同，事件的特点是：如果你错过了它，再去监听是得不到结果的。

Promise 的缺点：

无法取消Promise，一旦新建它就会立即执行，无法中途取消。

如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。

当处于pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

总结：

Promise 对象是异步编程的一种解决方案，最早由社区提出。Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由pending 转变 resolved 或者rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。

状态的改变是通过 resolve() 和 reject() 函数来实现的，可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

注意：在构造 Promise 的时候，构造函数内部的代码是立即执行的

32. Promise 解决了什么问题

在工作中经常会碰到这样一个需求，比如我使用ajax 发一个A 请求后，成功后拿到数据，需要把数据传给 B 请求；那么需要如下编写代码：

上面的代码有如下缺点：

```
let fs = require('fs')
fs.readFile('./a.txt','utf8',function(err,data){
  fs.readFile(data,'utf8',function(err,data){
    fs.readFile(data,'utf8',function(err,data){
      console.log(data)
    })
  })
})
```

后一个请求需要依赖于前一个请求成功后，将数据往下传递，会导致多个 ajax 请求嵌套的情况，代码不够直观。

如果前后两个请求不需要传递参数的情况下，那么后一个请求也需要前一个请求成功后再执行下一步操作，这种情况下，那么也需要如上编写代码，导致代码不够直观。

Promise 出现之后，代码变成这样：

```
let fs = require('fs')
function read(url){
  return new Promise((resolve,reject)=>{
    fs.readFile(url,'utf8',function(error,data){
      error && reject(error)
      resolve(data)
    })
  })
}
read('./a.txt').then(data=>{
  return read(data)
}).then(data=>{
  return read(data)
}).then(data=>{
  console.log(data)
})
```

这样代码看起了就简洁了很多，解决了地狱回调的问题。

33. 对 async/await 的理解

async/await 其实是 Generator 的语法糖，它能实现的效果都能用 then 链来实现，它是为优化then 链而开发出来的。从字面上来看，async 是“异步”的简写，await 则为等待，所以很好理解async 用于申明一个 function 是异步的，而 await 用于等待一个异步方法执行完成。当然语法上强制规定 await 只能出现在asnyc 函数中，先来看看async 函数返回了什么：

```
async function testAsy(){
    return 'hello world';
}
let result = testAsy();
console.log(result)
```

所以，async 函数返回的是一个 Promise 对象。async 函数（包含函数语句、函数表达式、Lambda 表达式）会返回一个 Promise 对象， 如果在函数中 return 一个直接量，async 会把这个直接量通过 Promise.resolve() 封装成 Promise 对象。

async 函数返回的是一个 Promise 对象,所以在最外层不能用await 获取其返回值的情况下，当然应该用原来的方式：then() 链来处理这个 Promise 对象，就像这样：

```
async function testAsy(){
    return 'hello world'
}
let result = testAsy()
console.log(result)
result.then(v=>{
    console.log(v)   // hello world
})
```

那如果 async 函数没有返回值，又该如何？很容易想到，它会返回 Promise.resolve(undefined)。

联想一下 Promise 的特点——无等待，所以在没有 await 的情况下执行 async 函数，它会立即执行，返回一个 Promise 对象，并且，绝不会阻塞后面的语句。这和普通返回 Promise 对象的函数并无二致。

注意：Promise.resolve(x) 可以看作是 new Promise(resolve => resolve(x)) 的简写，可以用于快速封装字面量对象或其他对象，将其封装成 Promise 实例。

34. async/await 的优势

单一的 Promise 链并不能发现 async/await 的优势，但是，如果需要处理由多个 Promise 组成的 then 链的时候，优势就能体现出来了（很有意思，Promise 通过 then 链来解决多层回调的问题，现在又用 async/await 来进一步优化它）。

假设一个业务，分多个步骤完成，每个步骤都是异步的，而且依赖于上一个步骤的结果。仍然用 setTimeout 来模拟异步操作：

```
/**
 * 传入参数 n，表示这个函数执行的时间（毫秒）
 * 执行的结果是 n + 200，这个值将用于下一步骤
 */
function takeLongTime(n) {
    return new Promise(resolve => {
        setTimeout(() => resolve(n + 200), n);
    });
}
function step1(n) {
    console.log(`step1 with ${n}`);
    return takeLongTime(n);
}
function step2(n) {
    console.log(`step2 with ${n}`);
    return takeLongTime(n);
}
function step3(n) {
    console.log(`step3 with ${n}`);
    return takeLongTime(n);
}
```

现在用 Promise 方式来实现这三个步骤的处理：

```
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
            console.timeEnd("doIt");
        });
}
doIt();
// c:\var\test>node --harmony_async_await .
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
// doIt: 1507.251ms
```

输出结果 result 是 step3() 的参数 700 + 200 = 900。doIt() 顺序执行了三个步骤，一共用了 300 + 500 + 700 = 1500 毫秒，和 console.time()/console.timeEnd() 计算的结果一致。

如果用 async/await 来实现呢，会是这样：

```
async function doIt() {
    console.time("doIt");
    const time1 = 300;
    const time2 = await step1(time1);
    const time3 = await step2(time2);
    const result = await step3(time3);
    console.log(`result is ${result}`);
    console.timeEnd("doIt");
}
doIt();
```

结果和之前的 Promise 实现是一样的，但是这个代码看起来是不是清晰得多，几乎跟同步代码一样

35. async/await 对比Promise 的优势

代码读起来更加同步，Promise 虽然摆脱了回调地狱，但是 then 的链式调用也会带来额外的阅读负担

Promise 传递中间值非常麻烦，而async/await几乎是同步的写法，非常优雅

错误处理友好，async/await 可以用成熟的 try/catch，Promise 的错误捕获非常冗余

调试友好，Promise 的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个.then 代码块中使用调试器的步进(step-over)功能，调试器并不会进入后续的.then 代码块，因为调试器只能跟踪同步代码的每一步。

36. 对象创建的方式有哪些？

一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js 和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是可以使用函数来进行模拟，从而产生出可复用的对象创建方式，常见的有以下几种：

（1） 第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

（2） 第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

（3） 第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

（4） 第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

（5） 第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

（6） 第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型， 在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

37. 对象继承的方式有哪些？

（1） 第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享， 容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

（2） 第二种方式是使用借用构造函数的方式，这种方式是通过在子 类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不 能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函 数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

（3） 第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

（4） 第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象， 然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 Object.create() 方法就是原型式继承的实现。缺点与原型链方式相同。

（5） 第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本， 然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是自定义类型时。缺点是没有办法实现函数的复用。

（6） 第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

38. 哪些情况会导致内存泄漏

以下四种情况会造成内存的泄漏：

意外的全局变量：由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。

被遗忘的计时器或回调函数：设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。

脱离 DOM 的引用：获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。

闭包：不合理的使用闭包，从而导致某些变量一直被留在内存当中。

基础篇之HTML 部分

1. 对HTML 语义化的理解

语义化是指根据内容的结构化（内容语义化），选择合适的标签（代码语义化）。通俗来讲就是用正确的标签做正确的事情。

语义化的优点如下：

对机器友好，带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，有利于 SEO。除此之外，语义类还支持读屏软件，根据文章可以自动生成目录；

对开发者友好，使用语义类标签增强了可读性，结构更加清晰，开发者能清晰的看出网页的结构，便于团队的开发与维护。

常见的语义化标签：

```
<header></header> 头部

<nav></nav> 导航栏

<section></section> 区块（有语义化的div）

<main></main> 主要区域

<article></article> 主要内容

<aside></aside> 侧边栏

<footer></footer> 底部
```

2. DOCTYPE(文档类型) 的作用

DOCTYPE 是HTML5 中一种标准通用标记语言的文档类型声明，它的目的是告诉浏览器（解析器）应该以什么样（html 或xhtml）的文档类型定义来解析文档，不同的渲染模式会影响浏览器对 CSS 代码甚至 JavaScript 脚本的解析。它必须声明在HTML文档的第一行。

浏览器渲染页面的两种模式（可通过 document.compatMode 获取，比如，语雀官网的文档类型是CSS1Compat）：

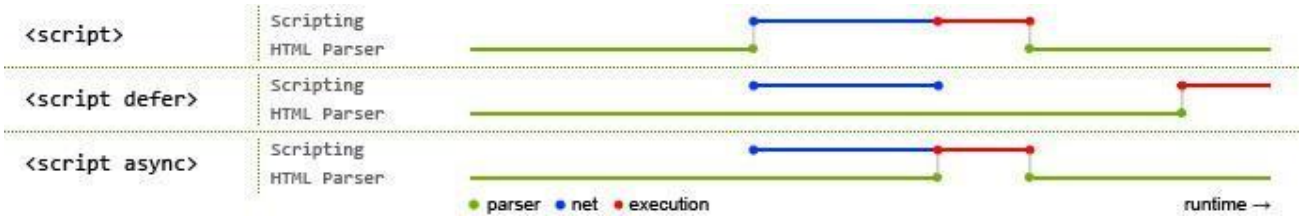
CSS1Compat：标准模式（Strickmode），默认模式，浏览器使用 W3C 的标准解析渲染页面。在标准模式中，浏览器以其支持的最高标准呈现页面。

BackCompat：怪异模式(混杂模式)(Quickmode)，浏览器使用自己的怪异模式解析渲染页面。在怪异模式中，页面以一种比较宽松的向后兼容的方式显示。

3. script 标签中 defer 和async 的区别

如果没有defer 或async 属性，浏览器会立即加载并执行相应的脚本。它不会等待后续加载的文档元素，读取到就会开始加载和执行，这样就阻塞了后续文档的加载。

下图可以直观的看出三者之间的区别:



其中蓝色代表js 脚本网络加载时间，红色代表 js 脚本执行时间，绿色代表html 解析。

defer 和 async 属性都是去异步加载外部的JS 脚本文件，它们都不会阻塞页面的解析，其区别如下：

执行顺序：多个带 async 属性的标签，不能保证加载的顺序；多个带defer 属性的标签，按照加载顺序执行；

脚本是否并行执行：async 属性，表示后续文档的加载和执行与 js 脚本的加载和执行是并行进行的，即异步执行；defer 属性，加载后续文档的过程和 js 脚本的加载(此时仅加载不执行)是并行进行的

(异步)，js 脚本需要等到文档所有元素解析完成之后才执行，DOMContentLoaded 事件触发执行之前。

4. 行内元素有哪些？块级元素有哪些？ 空(void)元素有那些？

行内元素有：a b span img input select strong;

块级元素有：div ul ol li dl dt dd h1 h2 h3 h4 h5 h6 p;

空元素，即没有内容的 HTML 元素。空元素是在开始标签中关闭的， 也就是空元素没有闭合标签：

常见的有：
、<hr>、<img>、<input>、<link>、<meta>;

鲜见的有：<area>、<base>、<col>、<colgroup>、<command>、<embed>、<keygen>、<param>、<source>、<track>、<wbr>。

5. 浏览器是如何对 HTML5 的离线储存资源进行管理和加载？

在线的情况下，浏览器发现 html 头部有 manifest 属性，它会请求 manifest 文件，如果是第一次访问页面 ，那么浏览器就会根据manifest 文件的内容下载相应的资源并且进行离线存储。如果已经访问过页面并且资源已经进行离线存储了，那么浏览器就会使用离线的资源加载页面，然后浏览器会对比新的 manifest 文件与旧的manifest 文件，如果文件没有发生改变，就不做任何操作，如果文件改变了，就会重新下载文件中的资源并进行离线存储。

离线的情况下，浏览器会直接使用离线存储的资源。

6. Canvas 和SVG 的区别

（1）SVG:

SVG 可缩放矢量图形（Scalable Vector Graphics）是基于可扩展标记语言 XML 描述的 2D 图形的语言，SVG 基于 XML 就意味着 SVG DOM 中的每个元素都是可用的，可以为某个元素附加 Javascript 事件处理器。在 SVG 中，每个被绘制的图形均被视为对象。如果 SVG 对象的属性发生变化，那么浏览器能够自动重现图形。

其特点如下：不

依赖分辨率 支

持事件处理器

最适合带有大型渲染区域的应用程序（比如谷歌地图）

复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）

不适合游戏应用

（2）Canvas：

Canvas 是画布，通过 Javascript 来绘制 2D 图形，是逐像素进行渲染的。其位置发生改变，就会重新进行绘制。

其特点如下：

依赖分辨率

不支持事件处理器

弱的文本渲染能力

能够以 .png 或 .jpg 格式保存结果图像

最适合图像密集型的游戏，其中的许多对象会被频繁重绘

注：矢量图，也称为面向对象的图像或绘图图像，在数学上定义为一系列由线连接的点。矢量文件中的图形元素称为对象。每个对象都是

一个自成一体的实体，它具有颜色、形状、轮廓、大小和屏幕位置等属性。

7. 说一下 HTML5 drag API

dragstart:事件主体是被拖放元素,在开始拖放被拖放元素时触发。darg:事件主体是被拖放元素，在正在拖放被拖放元素时触发。dragenter：事件主体是目标元素，在被拖放元素进入某元素时触发。dragover：事件主体是目标元素，在被拖放在某元素内移动时触发。

dragleave：事件主体是目标元素，在被拖放元素移出目标元素是触发。

drop:事件主体是目标元素,在目标元素完全接受被拖放元素时触发。dragend:事件主体是被拖放元素，在整个拖放操作结束时触发。

基础篇之CSS 部分

1. display 的block、inline 和inline-block 的区别

（1）block：会独占一行，多个元素会另起一行，可以设置width、height、margin 和padding 属性;

（2）inline：元素不会独占一行，设置width、height 属性无效。但可以设置水平方向的margin 和padding 属性，不能设置垂直方向的 padding 和margin;

（3）inline-block：将对象设置为 inline 对象，但对象的内容作为 block 对象呈现，之后的内联对象会被排列在同一行内。

对于行内元素和块级元素，其特点如下：

（1）行内元素

设置宽高无效；

可以设置水平方向的margin 和padding 属性，不能设置垂直方向的padding 和margin；

不会自动换行；

（2）块级元素

可以设置宽高；

设置margin 和padding 都有效；

可以自动换行；

多个块状，默认排列从上到下。

2. link 和@import 的区别

两者都是外部引用CSS 的方式，它们的区别如下：

link 是XHTML 标签，除了加载CSS 外，还可以定义RSS 等其他事务；@import 属于CSS 范畴，只能加载CSS。

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。

link 是XHTML 标签，无兼容问题；@import 是在CSS2.1 提出的，低版本的浏览器不支持。

link 支持使用Javascript 控制DOM 去改变样式；而@import 不支持。

3. CSS3 中有哪些新特性

新增各种CSS 选择器 （: not(.input)：所有 class 不是“input”的节点）

圆角 （border-radius:8px）

多列布局 （multi-column layout）

阴影和反射 （Shadoweflect）

文字特效 （text-shadow）

文字渲染 （Text-decoration）

线性渐变 （gradient）

旋转 （transform）

增加了旋转,缩放,定位,倾斜,动画,多背景

4. 对 CSSSprites 的理解

CSSSprites（精灵图），将一个页面涉及到的所有图片都包含到一张大图中去，然后利用CSS的 background-image,background-repeat，background-position 属性的组合进行背景定位。

优点：

利用 CSS Sprites 能很好地减少网页的http 请求，从而大大提高了页面的性能，这是CSS Sprites 最大的优点；

CSS Sprites 能减少图片的字节，把 3 张图片合并成 1 张图片的字节总是小于这 3 张图片的字节总和。

缺点：

在图片合并时，要把多张图片有序的、合理的合并成一张图片，还要留好足够的空间，防止板块内出现不必要的背景。在宽屏及高分辨率下的自适应页面，如果背景不够宽，很容易出现背景断裂；

CSSSprites 在开发的时候相对来说有点麻烦，需要借助 photoshop 或其他工具来对每个背景单元测量其准确的位置。

维护方面：CSS Sprites 在维护的时候比较麻烦，页面背景有少许改动时，就要改这张合并的图片，无需改的地方尽量不要动，这样避免改动更多的CSS，如果在原来的地方放不下，又只能（最好）往下加图片，这样图片的字节就增加了，还要改动CSS。

5. CSS 优化和提高性能的方法有哪些？

加载性能：

（1）css 压缩：将写好的css 进行打包压缩，可以减小文件体积。

（2）css 单一样式：当需要下边距和左边距的时候，很多时候会选择使用 margin:top 0 bottom 0 ；但 margin-bottom:bottom;margin-left:left;执行效率会更高。

（3）减少使用@import，建议使用 link，因为后者在页面加载时一起加载，前者是等待页面加载完成之后再进行加载。

选择器性能：

（1）关键选择器（key selector）。选择器的最后面的部分为关键选择器（即用来匹配目标元素的部分）。CSS 选择符是从右到左进行匹配的。当使用后代选择器的时候，浏览器会遍历所有子元素来确定是否是指定的元素等等；

（2）如果规则拥有 ID 选择器作为其关键选择器，则不要为规则增加标签。过滤掉无关的规则（这样样式系统就不会浪费时间去匹配它们了）。

（3）避免使用通配规则，如*{}计算次数惊人，只对需要用到的元素进行选择。

（4）尽量少的去对标签进行选择，而是用class。

（5） 尽量少的去使用后代选择器，降低选择器的权重值。后代选择器的开销是最高的，尽量将选择器的深度降到最低，最高不要超过三层，更多的使用类来关联每一个标签元素。

（6） 了解哪些属性是可以通过继承而来的，然后避免对这些属性重复指定规则。

渲染性能：

（1） 慎重使用高性能属性：浮动、定位。

（2） 尽量减少页面重排、重绘。

（3） 去除空规则：｛｝。空规则的产生原因一般来说是为了预留样式。去除这些空规则无疑能减少css 文档体积。

（4） 属性值为 0 时，不加单位。

（5） 属性值为浮动小数 0.**，可以省略小数点之前的 0。

（6） 标准化各种浏览器前缀：带浏览器前缀的在前。标准属性在后。

（7） 不使用@import 前缀，它会影响css 的加载速度。

（8） 选择器优化嵌套，尽量避免层级过深。

（9） css 雪碧图，同一页面相近部分的小图标，方便使用，减少页面的请求次数，但是同时图片本身会变大，使用时，优劣考虑清楚， 再使用。

（10） 正确使用 display 的属性，由于 display 的作用，某些样式组合会无效，徒增样式体积的同时也影响解析性能。

（11） 不滥用 web 字体。对于中文网站来说WebFonts 可能很陌生， 国外却很流行。web fonts 通常体积庞大，而且一些浏览器在下载 web fonts 时会阻塞页面渲染损伤性能。

可维护性、健壮性：

（1） 将具有相同属性的样式抽离出来，整合并通过class 在页面中进行使用，提高css 的可维护性。

（2） 样式与内容分离：将css 代码定义到外部css 中。

6. 对 CSS 工程化的理解

CSS 工程化是为了解决以下问题：

1. 宏观设计：CSS 代码如何组织、如何拆分、模块结构怎样设计？

2. 编码优化：怎样写出更好的 CSS？

3. 构建：如何处理我的 CSS，才能让它的打包结果最优？

4. 可维护性：代码写完了，如何最小化它后续的变更成本？如何确保任何一个同事都能轻松接手？

以下三个方向都是时下比较流行的、普适性非常好的 CSS 工程化实践：

预处理器：Less、 Sass 等； 重

要的工程化插件： PostCss；

Webpack loader 等 。

基于这三个方向，可以衍生出一些具有典型意义的子问题，这里我们逐个来看：

（1） 预处理器：为什么要用预处理器？它的出现是为了解决什么问题？

预处理器，其实就是 CSS 世界的“轮子”。预处理器支持我们写一种类似 CSS、但实际并不是 CSS 的语言，然后把它编译成 CSS 代码：

那为什么写 CSS 代码写得好好的，偏偏要转去写“类 CSS”呢？这就和本来用 JS 也可以实现所有功能，但最后却写 React 的 jsx 或者 Vue 的模板语法一样——为了爽！要想知道有了预处理器有多爽， 首先要知道的是传统 CSS 有多不爽。随着前端业务复杂度的提高， 前端工程中对 CSS 提出了以下的诉求：

1. 宏观设计上：我们希望能优化 CSS 文件的目录结构，对现有的 CSS 文件实现复用；

2. 编码优化上：我们希望能写出结构清晰、简明易懂的 CSS，需要它具有一目了然的嵌套层级关系，而不是无差别的一铺到底写法；我们希望它具有变量特征、计算能力、循环能力等等更强的可编程性，这样我们可以少写一些无用的代码；

3. 可维护性上：更强的可编程性意味着更优质的代码结构，实现复用意味着更简单的目录结构和更强的拓展能力，这两点如果能做到，自然会带来更强的可维护性。

这三点是传统 CSS 所做不到的，也正是预处理器所解决掉的问题。预处理器普遍会具备这样的特性：

嵌套代码的能力，通过嵌套来反映不同 css 属性之间的层级关系 ；

支持定义 css 变量；

提供计算函数；

允许对代码片段进行 extend 和 mixin；

支持循环语句的使用；

支持将 CSS 文件模块化，实现复用。

（2） PostCss：PostCss 是如何工作的？我们在什么场景下会使用 PostCss？

PostCss 仍然是一个对 CSS 进行解析和处理的工具，它会对 CSS 做这样的事情：

它和预处理器的不同就在于，预处理器处理的是 类CSS，而 PostCss 处理的就是 CSS 本身。Babel 可以将高版本的 JS 代码转换为低版本的 JS 代码。PostCss 做的是类似的事情：它可以编译尚未被浏览器广泛支持的先进的 CSS 语法，还可以自动为一些需要额外兼容的语法增加前缀。更强的是，由于 PostCss 有着强大的插件机制，支持各种各样的扩展，极大地强化了 CSS 的能力。

PostCss 在业务中的使用场景非常多：

提高 CSS 代码的可读性：PostCss 其实可以做类似预处理器能做的工作；

当我们的CSS 代码需要适配低版本浏览器时，PostCss 的Autoprefixer 插件可以帮助我们自动增加浏览器前缀；

允许我们编写面向未来的 CSS：PostCss 能够帮助我们编译 CSS next 代码；

（3）Webpack 能处理 CSS 吗？如何实现？

Webpack 能处理 CSS 吗：

Webpack 在裸奔的状态下，是不能处理 CSS 的，Webpack 本身是一个面向 JavaScript 且只能处理 JavaScript 代码的模块化打包工具；

Webpack 在 loader 的辅助下，是可以处理 CSS 的。

如何用 Webpack 实现对 CSS 的处理：

Webpack 中操作 CSS 需要使用的两个关键的 loader：css-loader 和 style-loader

注意，答出“用什么”有时候可能还不够，面试官会怀疑你是不是在背答案，所以你还需要了解每个 loader 都做了什么事情：

css-loader：导入 CSS 模块，对 CSS 代码进行编译处理；

style-loader：创建style 标签，把 CSS 内容写入标签。

在实际使用中,css-loader 的执行顺序一定要安排在 style-loader 的前面。因为只有完成了编译过程，才可以对 css 代码进行插入； 若提前插入了未编译的代码，那么 webpack 是无法理解这坨东西的， 它会无情报错。

7. 常见的 CSS 布局单位

常用的布局单位包括像素（px），百分比（%），em，rem，vw/vh。

（1） 像素（px）是页面布局的基础，一个像素表示终端（电脑、手机、平板等）屏幕所能显示的最小的区域，像素分为两种类型：CSS 像素和物理像素：

CSS 像素：为web 开发者提供，在CSS 中使用的一个抽象单位；

物理像素：只与设备的硬件密度有关，任何设备的物理像素都是固定的。

（2） 百分比（%），当浏览器的宽度或者高度发生变化时，通过百分 比单位可以使得浏览器中的组件的宽和高随着浏览器的变化而变化， 从而实现响应式的效果。一般认为子元素的百分比相对于直接父元素。

（3） em 和rem 相对于 px 更具灵活性，它们都是相对长度单位，它们之间的区别：em 相对于父元素，rem 相对于根元素。

em： 文本相对长度单位。相对于当前对象内文本的字体尺寸。如果当前行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸(默认 16px)。(相对父元素的字体大小倍数)。

rem： rem 是CSS3 新增的一个相对单位，相对于根元素（html 元素） 的font-size 的倍数。作用：利用rem 可以实现简单的响应式布局， 可以利用 html 元素中字体的大小与屏幕间的比值来设置 font-size 的值，以此实现当屏幕分辨率变化时让元素也随之变化。

（4） vw/vh 是与视图窗口有关的单位，vw 表示相对于视图窗口的宽度vh表示相对于视图窗口高度，除了vw和 vh外，还有vmin和 vmax两个相关的单位。

vw：相对于视窗的宽度，视窗宽度是 100vw；

vh：相对于视窗的高度，视窗高度是 100vh；

vmin：vw 和vh 中的较小值；

vmax：vw 和vh 中的较大值；

vw/vh 和百分比很类似，两者的区别：

百分比（%）：大部分相对于祖先元素，也有相对于自身的情况比如（border-radius、translate 等）

vw/vm：相对于视窗的尺寸

8. 水平垂直居中的实现

利用绝对定位，先将元素的左上角通过 top:50%和 left:50%定位到页面的中心，然后再通过 translate 来调整元素的中心点到页面的中心。该方法需要考虑浏览器兼容问题。

```
.parent {
    position: relative;
}

.child {
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%,-50%);
}
```

利用绝对定位，设置四个方向的值都为 0，并将 margin 设置为auto， 由于宽高固定，因此对应方向实现平分，可以实现水平和垂直方向上的居中。该方法适用于盒子有宽高的情况：

```
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    margin: auto;
}
```

利用绝对定位，先将元素的左上角通过 top:50%和 left:50%定位到页面的中心，然后再通过margin 负值来调整元素的中心点到页面的中心。该方法适用于盒子宽高已知的情况

```
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 50%;
    left: 50%;
    margin-top: -50px;     /* 自身 height 的一半 */
    margin-left: -50px;    /* 自身 width 的一半 */
}
```

使用 flex 布局，通过 align-items:center 和 justify-content:center设置容器的垂直和水平方向上为居中对齐，然后它的子元素也可以实现垂直和水平的居中。该方法要考虑兼容的问题，该方法在移动端用的较多：

```
.parent {
    display: flex;
    justify-content:center;
    align-items:center;
}
```

9. 对BFC 的理解，如何创建 BFC

先来看两个相关的概念：

Box: Box 是 CSS 布局的对象和基本单位，一个页面是由很多个 Box 组成的，这个Box 就是我们所说的盒模型。

Formatting context：块级上下文格式化，它是页面中的一块渲染区域，并且有一套渲染规则，它决定了其子元素将如何定位，以及和其他元素的关系和相互作用。

块格式化上下文（Block Formatting Context，BFC）是 Web 页面的可视化CSS 渲染的一部分，是布局过程中生成块级盒子的区域，也是浮动元素与其他元素的交互限定区域。

通俗来讲：BFC 是一个独立的布局环境，可以理解为一个容器，在这个容器中按照一定规则进行物品摆放，并且不会影响其它环境中的物品。如果一个元素符合触发 BFC 的条件，则BFC 中的元素布局不受外部影响。

创建BFC 的条件：

根元素：body；

元素设置浮动：float 除 none 以外的值；

元素设置绝对定位：position (absolute、fixed)；

display 值为:inline-block、table-cell、table-caption、flex 等；

overflow 值为:hidden、auto、scroll；BFC

的特点：

垂直方向上，自上而下排列，和文档流的排列方式一致。

在 BFC 中上下相邻的两个容器的margin 会重叠

计算BFC 的高度时，需要计算浮动元素的高度

BFC 区域不会与浮动的容器发生重叠

BFC 是独立的容器，容器内部元素不会影响外部元素

每个元素的左margin 值和容器的左border 相接触

BFC 的作用：

解决 margin 的重叠问题：由于BFC 是一个独立的区域，内部的元素和外部的元素互不影响，将两个元素变为两个 BFC，就解决了 margin 重叠的问题。

解决高度塌陷的问题：在对子元素设置浮动后，父元素会发生高度塌陷，也就是父元素的高度变为 0。解决这个问题，只需要把父元素变成一个 BFC。常用的办法是给父元素设置overflow:hidden。

创建自适应两栏布局：可以用来创建自适应两栏布局：左边的宽度固定，右边的宽度自适应。

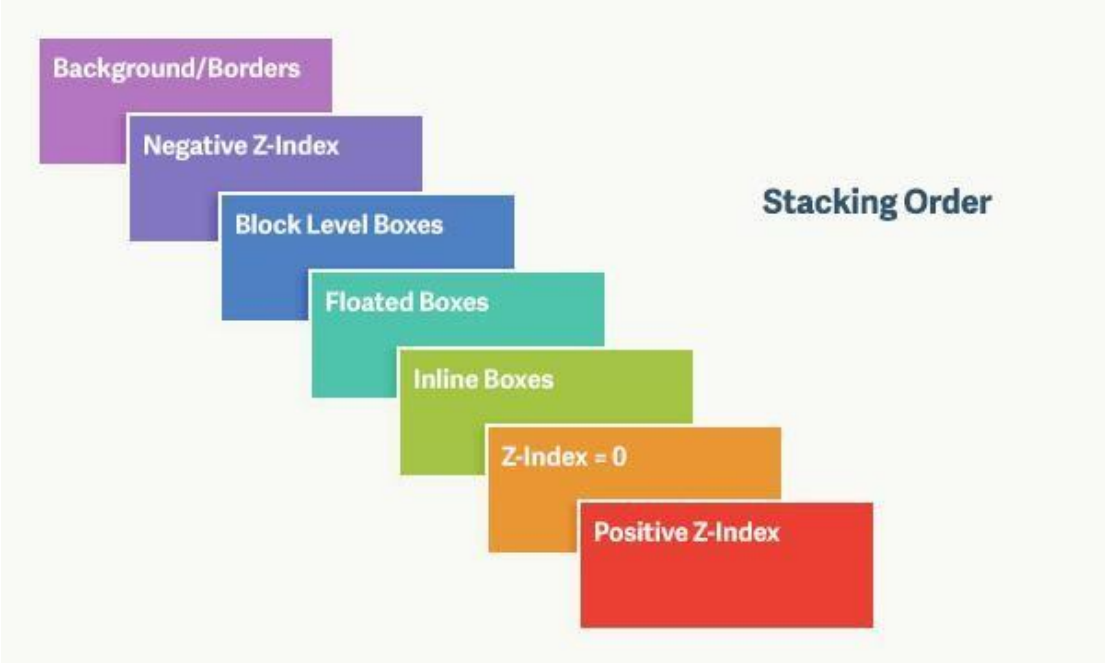
```
.left{
    width: 100px;
    height: 200px;
    background: red;
    float: left;
}
.right{
    height: 300px;
    background: blue;
    overflow: hidden;
}

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

左侧设置 float:left，右侧设置 overflow: hidden。这样右边就触发了 BFC，BFC 的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠，实现了自适应两栏布局。

10. 元素的层叠顺序

层叠顺序，英文称作 stacking order，表示元素发生层叠时有着特定的垂直显示顺序。下面是盒模型的层叠规则：



对于上图，由上到下分别是：

（1） 背景和边框：建立当前层叠上下文元素的背景和边框。

（2） 负的 z-index：当前层叠上下文中，z-index 属性值为负的元素。

（3） 块级盒：文档流内非行内级非定位后代元素。

（4） 浮动盒：非定位浮动元素。

（5） 行内盒：文档流内行内级非定位后代元素。

（6）z-index:0：层叠级数为 0 的定位元素。

（7）正 z-index：z-index 属性值为正的定位元素。

注意：当定位元素 z-index:auto，生成盒在当前层叠上下文中的层级为 0，不会建立新的层叠上下文，除非是根元素。

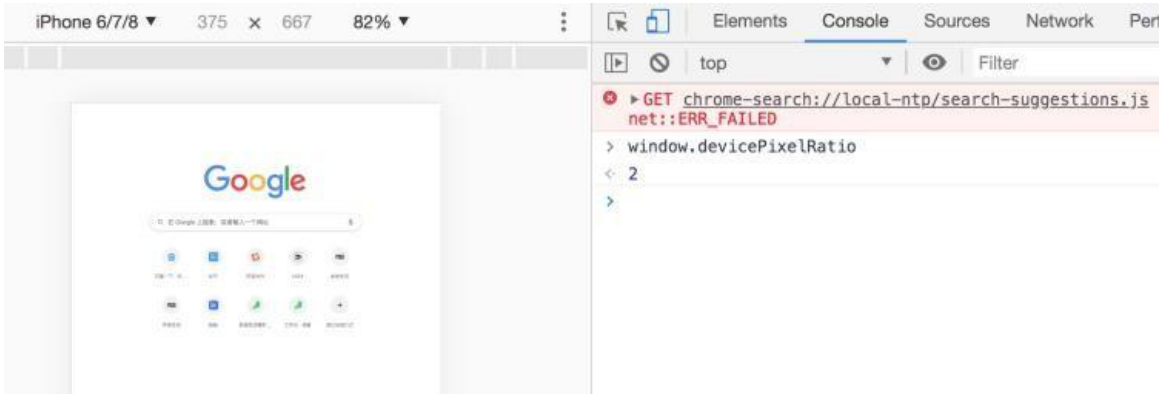
11. 如何解决 1px 问题？

1px 问题指的是：在一些 Retina 屏幕 的机型上，移动端页面的 1px 会变得很粗，呈现出不止 1px 的效果。原因很简单——CSS 中的 1px

并不能和移动设备上的 1px 划等号。它们之间的比例关系有一个专门的属性来描述：

```
window.devicePixelRatio = 设备的物理像素 / CSS像素。
```

打开 Chrome 浏览器，启动移动端调试模式，在控制台去输出这个 devicePixelRatio 的值。这里选中 iPhone6/7/8 这系列的机型，输出的结果就是 2：



这就意味着设置的 1pxCSS 像素，在这个设备上实际会用 2 个物理像素单元来进行渲染，所以实际看到的一定会比 1px 粗一些。

解决 1px 问题的三种思路：

思路一：直接写 0.5px

如果之前 1px 的样式这样写：

```
border:1px solid #333
```

可以先在 JS 中拿到 window.devicePixelRatio 的值，然后把这个值通过 JSX 或者模板语法给到 CSS 的 data 里，达到这样的效果（这里用 JSX 语法做示范）：

```
<div id="container" data-device={{window.devicePixelRatio}}></div>
```

然后就可以在 CSS 中用属性选择器来命中 devicePixelRatio 为某一值的情况，比如说这里尝试命中 devicePixelRatio 为 2 的情况：

```
#container[data-device="2"] {
  border:0.5px solid #333
}
```

直接把 1px 改成 1/devicePixelRatio 后的值，这是目前为止最简单的一种方法。这种方法的缺陷在于兼容性不行，IOS 系统需要 8 及以上的版本，安卓系统则直接不兼容。

思路二：伪元素先放大后缩小

这个方法的可行性会更高，兼容性也更好。唯一的缺点是代码会变多。

思路是先放大、后缩小：在目标元素的后面追加一个 ::after 伪元素，让这个元素布局为 absolute 之后、整个伸展开铺在目标元素上，然后把它的宽和高都设置为目标元素的两倍，border 值设为 1px。接着借助 CSS 动画特效中的放缩能力，把整个伪元素缩小为原来的50%。此时，伪元素的宽高刚好可以和原有的目标元素对齐，而 border 也缩小为了 1px 的二分之一，间接地实现了 0.5px 的效果。

代码如下：

```css
#container[data-device="2"] {
    position: relative;
}
#container[data-device="2"]::after{
      position:absolute;
      top: 0;
      left: 0;
      width: 200%;
      height: 200%;
      content:"";
      transform: scale(0.5);
      transform-origin: left top;
      box-sizing: border-box;
      border: 1px solid #333;
    }
}
```

本文收集整理于语雀

博主 [CUGGZ]的原创

文章原 文 链 接 ：

https://www.yuque.

com/cuggz/intervie

w