https://segmentfault.com/a/1190000020026378

# 1. for in和for of的区别

各种循环的底层机制以及性能对比 forEach while

for in 性能不会好 因为会原型链查找 公有的属性如果不需要迭代，需要筛选 Object.hasOwnProperty(key) 遍历所有私有和公有的可枚举属性， 优先迭代数字属性， 无法迭代symbol类型的私有属性， 封装eachObject 如果要获取所有的key，且不包括公有 Object.keys() / Object.getOwnPropertyNames() 获取symbol Object.getOwnPropertySymbols(obj)

区别： for in for(let i in arr) { console.log(i,'ccccccccccccc') } 循环数组 => 得到索引 循环对象 => key

for of 遍历数组 for(let i of arr) { console.log(i,'ccccccccccccc') } 循环数组 => 数组的每一项 循环对象 =>报错 obj is not iterable

for...of...是es6引入的新特性

for of可以迭代数组，string，map，set，arguments对象，dom集合

for of语句在**可迭代对象**上创建一个迭代循环，调用自定义迭代钩子，并为每个不同属性的值执行语句 在for of的循环中，可以由break，continue return终止

可以迭代显式实现可迭代协议的对象。 var iterable = { Symbol.iterator { return { i:0, next() { if(this.i < 3) { return {value:this.i++,done:false} } return {value:undefined,done:true} } } } } for(var value of iterable) { console.log(value) // 0 1 2 }

array或map是内置的可迭代对象，object不是，要成为可迭代对象，一个对象必须实现@@iterator方法，使用Symbol.iterator访问该属性，当一个对象需要被迭代的时候，如for of循环，会不带参数调用@@iterator方法，使用此方法返回迭代器获得要迭代的值

# 2. 在给对象使用扩展运算符时，它会抛出异常

TypeError:obj is not iterable 我们能否以某种方式为下面的语句使用展开运算符而不导致类型错误？ let obj = {x:1,y:2,z:3} console.log([...obj]) // TypeError:obj is not iterable obj[Symbol.iterator] = function() { return { index:0, next: function() { let keys = Object.keys(obj) if(this.index < keys.length) { let key = keys[this.index] this.index ++ return {value:obj[key],done:false} } return {value:undefined, done:true} } } }

# 默认可以使用for of的

String let someString = "hi"; typeof someString[Symbol.iterator]; // "function"

[...someString] ['h', 'i']

我们可以通过提供自己的 @@iterator 方法，重新定义迭代行为： // 必须构造 String 对象以避免字符串字面量 auto-boxing var someString = new String("hi"); someString[Symbol.iterator] = function() { return { // 只返回一

次元素，字符串 "bye"，的迭代器对象 next: function() { if (this._first) { this._first = false; return { value: "bye", done: false }; } else { return { done: true }; } }, _first: true }; };

目前所有的内置可迭代对象如下：String、Array、TypedArray、Map 和 Set，它们的原型对象都实现了 @@iterator 方法。

var myIterable = {}; myIterable[Symbol.iterator] = function* () { yield 1; yield 2; yield 3; }; [...myIterable]; // [1, 2, 3]

一些语句和表达式需要可迭代对象，比如 for...of 循环、展开语法、yield*，和解构赋值。

for(let value of ["a", "b", "c"]){ console.log(value); } // "a" // "b" // "c"

[..."abc"]; // ["a", "b", "c"]

function* gen() { yield* ["a", "b", "c"]; } gen().next(); // { value: "a", done: false }

[a, b, c] = new Set(["a", "b", "c"]); a // "a"

# 3. 扩展运算符的10种用法

1. 复制数组（浅拷贝） 不适用于多层数组 var arr1 = [1,2,3] var arr2 = [...arr1]
2. 合并数组 var arr1 = [1,2,3] var arr2 = [4,5,6] var arr3= [...arr2,...arr1,...arr4,...arr5,...arr6] console.log(arr3) // [4,5,6,1,2,3]
3. 向数组中添加元素 let arr1 = [1,2,3] arr1 = [...arr1,9]
4. 向对象添加属性 const user = {a:1,b:2} const output = {...user,age:3}
5. 使用Math函数获取最小值最大值 var arr = [1,-1.0,5,3] const min = Math.min(...arr) const max = Math.max(...arr)

不使用扩展运算符 const min = Math.min(arr) console.log(min, '99999999999999') // NaN

6. rest参数 有一个函数，三个参数 const myFunc(x1, x2, x3) => {

   console.log(x1);

   console.log(x2);

   console.log(x3);

}

可以这样调用：myFunc(1, 2, 3);

如果传递一个数组 myFunc(...arr);

7. 向函数传递无限参数 假设有一个函数，它接受无限个参数 const myFunc = (...args) => { console.log(args) } 调用时候传递：myFunc(1, 'a', new Date()); 返回： [1,'a']
8. 将nodeList转换为数组 const el = [...document.querySelectorAll('p')];

console.log(el);

// (3) [p, p, p]

9. 解构对象 const user = {

firstname: 'Chris',

lastname: 'Bongers',

age: 31

};

const {firstName, ...rest} = user console.log(firstName) // 'Chris' console.log(rest) // {lastname: 'Bongers', age: 31}

10. 展开字符串 const str = 'hello' const arr = [...str] console.log(arr) // ['h','e','l','l','o']

# 4. js数据类型

基本类型：Number、Boolean、String、null(表示一个空对象指针)，undefined,symbol.BigInt 引用类型：Object Array RegExp Function Date

BigInt 表示大于2^53-1的整数，Number可以表示的最大整数2^53-1 可以用在一个整数字面量后面加 n 的方式定义一个 BigInt ，如：10n，或者调用函数BigInt()。

## 基本数据类型和引用数据类型的存储区别

基本数据类型存储在栈中 引用数据类型存储在堆中,栈中存放堆内存地址

// 基本数据类型 let a = 10; let b = a; // 赋值操作 b = 20; console.log(a); // 10值

// 引用数据类型

var obj1 = {} var obj2 = obj1; obj2.name = "Xxx"; console.log(obj1.name); // xxx

## 不同的类型数据导致赋值变量时的不同：

简单类型赋值，是生成相同的值，两个对象对应不同的地址 复杂类型赋值，是将保存对象的内存地址赋值给另一个变量。也就是两个变量指向堆内存中同一个对象

# 5. js数据类型检测

1. typeof 检测基本类型 引用数据类型都是Object，typeof function =>'function' typeof null => 'object' typeof undefined => undefined typeof NaN => 'number'
2. instanceof object instanceof constructor

instanceof运算符用于检测构造函数的prototype属性是否出现在某个实例对象的原型链上

'a' instanceof String => false var a = new String('a') a instanceof String => true

String，Number，Boolen类型需要new创建的实例才能正确的检测，字面量创建的不能检测

null instanceof Null 会报错 => Null is not defined undefined instanceof Undefined 报错 => Undefined is not defined

function instanceof Function => true 可以检测 date instanceof Date => true 可以检测 {} instanceof Object => true 可以检测 regexp instanceof RegExp => true 可以检测 [] instanceof Array => true 可以检测

3. constructor 可以检测除了null undefined 但是原型可以被改写，改写之后就不准确了 将构造函数的原型重新赋值，这样原型上的constructor指向就不会指向构造函数了

(1).constructor === Number => true可以检测 ('1').constructor === String => true 可以检测 (true).constructor === Boolean => true 可以检测 (null).constructor=== Null => 报错 (undefined).constructor === Udefined => 报错 (function).constructor === Function => true 可以检测 ([]).constructor === Array => true 可以检测 ({}).constructor === Object => true 可以检测 (/\d/).constructor === RegExp => true 可以检测 (date).constructor === Date => true 可以检测

改写constructor function F() {} F.prototype = new Array() var f = new F() f.constructor === F // false f.constructor === Array // true

4. Object.prototype.toString.call() Object.prototype.toString()方法返回一个表示该对象的字符串 每个对象都有一个toString()方法。当该对象表示为一个文本值时，或者一个对象以预期的字符串方式引用时自动调用。

Object.prototype.toString.call(1) => '[object Number]' Object.prototype.toSting.call('1') => '[object String]' Object.prototype.toSting.call(true) => '[object Boolean]' Object.prototype.toSting.call(null) => '[object Null]' Object.prototype.toSting.call(undefined) => '[object Undefined]' Object.prototype.toSting.call([]) => '[object Array]' Object.prototype.toSting.call({}) => '[object Object]' Object.prototype.toSting.call(function() {}) => '[object Function]' Object.prototype.toSting.call(/\d+/) => '[object RegExp]' Object.prototype.toSting.call(new Date()) => '[object Date]' Object.prototype.toSting.call(Symbol(1)) => '[object Symbol]'

Object.prototype上有哪些方法呢 1. toString 使用call方法可以检测数据类型 Object.prototype.toSting.call(obj) 2. constructor 指向构造函数 Object.prototype.constructor === Object 3. hasOwnProperty 判断是不是有这个属性 自身的私有属性 忽略从原型上继承的属性 4. isPrototypeOf 5. valueOf

# 4. 判断是不是纯对象

function isObject(obj) { return Object.prototype.toString.call(obj) === '[object Object]' }

# 5. 函数节流和函数防抖

防抖和节流都是为了解决短时间内大量触发某函数而导致的性能问题。二者对应的业务需求不一样，所以实现原理也不一样。

1. 防抖 是什么: 在事件被触发n秒之后再执行回调函数，如果在这n秒内又被触发，则重新记时

应用场景：(1) 用户在输入框连续输入一段字符，只会在输入完后去执行最后一次的查询ajax请求，这样可以有效减少请求次数，节约请求资源 (2) window的resize，scroll事件，不断地调整浏览器的窗口大小或者滚动时会触发对应事件，防抖让其只触发一次

实现： function debounce(fun,delay) { return function(...args) { window.clearTimeout(fun.id) fun.id = setTimeout(() => { fun.call(this,...args) }, delay) } }

2. 节流 是什么: 规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位内某事件被触发多次，只有一次生效。 应用场景：(1) 鼠标连续不断地操作如点击 (2) 在页面的无限加载场景下，需要用户在滚动页面时，每隔一段时间发一次ajax请求，而不是用户停下滚动页面操作时候才请求

实现： https://blog.csdn.net/Beijiyang999/article/details/79836463 const throttle = function throttle(func,wait) { wait = wait || 300 let timer = null,previous = 0; return function (...parms) { let now = +new Date(), // 时间戳 remaining = wait - (now - previous), result; if(remaining <=0 ){ // //当距上一次执行的间隔大于规定，可以直接执行 result = func.call(this,...parms) previous = +new Date() timer = clearTimer(timer) }else if(!timer) { // 否则继续等待，结尾执行一次 timer = setTimeout(() => { func.call(this, ...parms); previous = +new Date() timer = clearTimer(timer) }, remaining) } return result } }

- 区别 在事件持续被触发的时候，如果还没有到达等待的时间则： 防抖：回调将一直被推迟，可能不会执行 节流：函数将每个n秒执行一次

# 3. 图片懒加载 进入页面的时候只请求可视区域的图片资源

- 1. offsetTop(当前元素相对于其offsetParent元素的顶部的高度) - scrollTop(浏览器滚动的距离) < clientHeight(可视区域的高度) offsetTop - scrollTop < clientHeight function getTop(e) { var T = e.offsetTop; while(e = e.offsetParent) { T += e.offsetTop; } return T; } function lazyLoad(imgs) { var H = document.documentElement.clientHeight;//获取可视区域高度 var S = document.documentElement.scrollTop || document.body.scrollTop; for (var i = 0; i < imgs.length; i++) { if (H + S > getTop(imgs[i])) { imgs[i].src = imgs[i].getAttribute('data-src'); } } }

2. 利用getBoundingClientRect() { bottom: 8272.8125 height: 10172.8125 left: 0 right: 1903 top: -1900 width: 1903 x: 0 y: -190 } 返回元素的大小以及其相对于视口的位置 let box = document.querySelector('.box'), boxImg = box.querySelector('img'), HTML = document.documentElement; function lazyImg() { let {bottom,top} = box.getBoundingClientRect() if(bottom < document.documentElement.clientHeight && top > 0) { boxImg.src = boxImg.getAttribute('data-img'); boxImg.onload = () => { // 加载成功 boxImg.style.opacity = 1; }; } } window.addEventListener('scroll', throttle(lazyImage));

3. 利用IntersectionObserver IntersectionObserver提供了一种异步观察目标元素与其祖先元素或顶级文档视窗(viewport)交叉状态的方法。 当一个IntersectionObserver对象被创建时，其被配置为监听根中一段给定比例的可见区域

可以自动检测某个元素是否出现在页面可视区，交叉观察器

Intersection Observer API 提供了一种异步检测目标元素与祖先元素或 viewport 相交情况变化的方法。

- 图片懒加载——当图片滚动到可见时才进行加载
- 内容无限滚动——也就是用户滚动到接近内容底部时直接加载更多，而-无需用户操作翻页，给用户一种网页可以无限滚动的错觉
- 检测广告的曝光情况——为了计算广告收益，需要知道广告元素的曝光情况
- 在用户看见某个区域时执行任务或播放动画

let box = document.querySelector('.box'), boxImg = box.querySelector('img') let ob = new IntersectionObserver(changes => { let {isIntersecting,target} = changes[0] // 判断target元素在root中的可见性是否发生变化 if(isIntersecting) { box.src = boxImg.getAttribute('data-img') boxImg.onload = () => { boxImg.style.opacity = 1 ob.unobserver(target) } },{ threshold:[1] }

}) ob.observer(box)