# JavaScript

## Js执行机制

Js是单线程语言，同一时间内只能干一件事情；

**为什么JS不能是多线程？**

是否多线程取决于这个语言的用途。

一个简单的例子，如果同一时间，一个添加DOM，一个删除了DOM，这时候语言就不知道是该添还是删了。

**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，才能执行后一个任务。

**异步：**不进入主线程，而进入“任务队列”（task queue）的任务，只有“任务队列”通知主线程，该任务才会进入主线程执行

**运行机制**

1. 所有任务都在主线程上执行，形成一个执行栈
2. 主线程之外，还存在一个“任务队列”。只要异步任务有一个运行结果，就在“任务队列”之中放置一个事件。
3. 一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。于是那些对应的异步任务结束等待状态，进入执行栈，开始执行。
4. 主线程不断重复上面三步。

## 事件循环

* **js代码可以分为同步代码和异步代码；**

1. Js引擎在执行代码时，首先会将同步代码加入到主线程中执行，形成一个执行栈；

（2） 异步代码会放到event table中注册回调函数；

（3） 当异步任务处理完成时，JS引擎将异步任务的回调函数从event table移到“任务队列”中；

（4） 当主线程执行栈中的同步任务执行完毕，js引擎会从“任务队列”中读取最先进入队列中的任务，放入执行栈执行；

* **JS任务又可以分为宏任务和微任务**

宏任务有：主代码快、定时器（setTimeout、setInterval）、nodejs的setImmediate请求动画帧（requestAnimationFrame）、http请求、IO请求等；

微任务有：promise.then()、promise.catch()、promise.finally、MutationObserver、nodejs的process.nextTick

主代码块可以看成是第一个宏任务；

（1） 宏任务进入宏任务队列，每一个宏任务对应一个微任务队列；

（2） 一个宏任务执行过程中，遇到微任务会将其放入对应的微任务队列中；

（3） 当这个宏任务执行完成，会将对应的微任务队列中所有的任务执行完成；

（4） 然后再去宏任务队列获取下一个宏任务。

SetTimeOut()的执行原理：setTimeOut第一个参数是回调函数，第二个参数是指定多少毫秒后执行这个回调函数；

首先：将这个回调函数放入event table中，

浏览器中有一个定时器用于计算等待时间，当到达了指定的时间，将回调函数移动到宏任务队列中。

过程描述

1. JavaScript脚本进入主线程，开始执行；
2. 执行过程中如果遇到宏任务和微任务，分别将其挂起，只有当任务就绪时将事件放入相应的任务队列；
3. 脚本执行完成，执行栈清空；
4. 去微任务队列依次读取事件，并将响应的回调函数放入执行栈运行，如果运行过程中遇到了宏任务和微任务，处理方式同2，直到微任务队列为空；
5. 浏览器执行渲染动作，GUI渲染线程接管，直到渲染结束；
6. JS线程接管，去宏任务队列依次读取事件，并将相应的回调函数放入执行栈，开始下一个宏任务的执行，过程为2 -> 3 -> 4 -> 5 -> 6，如此循环。
7. 直到执行栈，宏任务队列、微任务 队列都为空，脚本执行结束。

## **JS变量提升**

函数和变量的声明都将被解释器提升到函数的最顶部

变量可以先使用再声明

变量初始化不会提升

## **作用域和作用域链**

作用域是可访问变量的集合。

在JavaScript中，对象和函数同样也是变量；

在JavaScript中，作用域为可访问变量、对象、函数的集合；

**局部作用域**

变量在函数内声明，变量为局部作用域；

局部变量：只能在函数内部访问；

局部变量在函数开始执行时创建，函数执行完后局部变量会自动销毁。

**全局作用域**

变量在函数外定义，即为全局变量

全局变量有全局作用域，网页中所有的脚本和函数均可使用；

如果变量在函数中没有声明（没有使用var关键字），该变量为全局变量。

**块级作用域**

ES6之前，没有块级作用域的概念

ES6可以使用let关键字来实现块级作用域

Let声明的变量只在let命令所在的代码块{}内有效，在{}之外不能访问

**变量生命周期**

Js变量生命周期在它声明时初始化；

局部变量在函数执行完毕后销毁；

全局变量在页面关闭后销毁。

**作用域链**

一般情况下，变量是在定义这个变量的函数作用域中取值，

但是如果在当前作用域中没有查到值，就会向上级作用域去查，直到查到全局作用域，这么一个查找过程形成的链条就叫做作用域链。

## this的执行

this是JavaScript语言的一个关键字。

它是函数运行时，在函数体内部自动生成的一个对象，只能在函数体内部使用。

this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底执行谁，实际上this的最终指向的是哪个调用它的对象。

this是当前执行上下文（global、function或eval）的一个属性，在非严格模式下，总是指向一个对象，在严格模式下可以是任意值。

* 全局上下文

无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this都指向全局对象。

* 函数上下文

在函数内部，this的值取决于函数被调用的方式。

var length = 10

function fn(){

    console.log(this.length)

}

function fn2(){

    var length = 5

    fn()

}

输出： 10

* 类上下文

在类的构造函数中，this是一个常规对象。

* 箭头函数中的this

定义时绑定this，继承的是父执行上下文里面的this

* 作为对象方法

当函数作为对象里的方法被调用，this被设置为调用该函数的对象。

* 构造函数中的this

当一个函数用作构造函数（使用new关键字），它的this被绑定到正在构建的新对象。

* 作为一个DOM事件处理函数

当函数被用作事件处理函数时，它的this指向触发事件的一个元素。

普通函数的this：指向它的调用者，如果没有调用者则默认指向window

箭头函数的this：指向箭头函数定义时所处的对象，而不是箭头函数使用时所在的对象，默认使用父级的this。

       var length = 10

        function fn(){

            console.log(this.length)

        }

        var obj = {

            length: 5,

            f : function(fn){

                fn()

                arguments[0]()

            }

        }

        //输出是10和2

        ////arguments是一个数组，也算是一个对象，被对象调用的方法的this指向该对象

## Call()、apply()、bind()

## 继承

ES5实现继承的方式

* **构造函数实现继承**

原理：在子类的构造函数中指向父类的构造函数，并为其绑定子类的this，让父类的构造函数把成员属性和方法都挂到子类的this上去

缺点：继承不到父类原型上的属性和方法

* **原型链实现继承**

原理：将子类的原型对象指向父类的实例

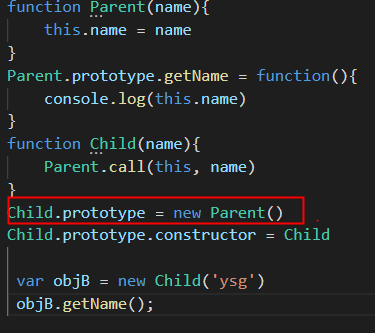
缺点：无法向父类构造函数传参

* **组合式继承**

原理：组合原型链继承和构造函数继承

用构造函数定义属性，用原型定义方法。

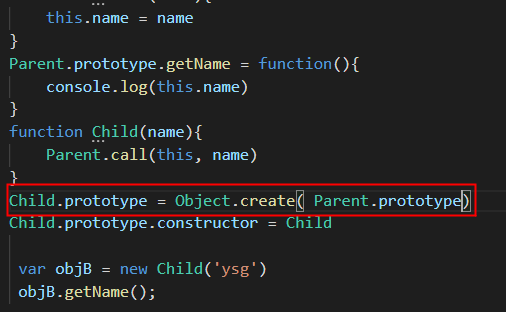
缺点：每次创建子类实例都执行了两次构造函数（Parent.call()和new Parent()），虽然这并不影响对父类的继承，但子类创建实例时，原型中会存在两份相同的属性和方法，这并不优雅。



* **寄生式组合继承**

为了解决构造函数被执行两次的问题，我们将执行父类实例改为指向父类原型，减去一次构造函数的执行。

但这种方式存在一个问题，由于子类原型和父类原型指向同一个对象，我们对子类原型的操作会影响到父类原型，为解决这个问题，我们给父类原型做一个浅拷贝。



寄生组合式继承，是目前ES5最成熟的继承方式，babel对ES6继承的转化也是使用寄生组合式继承。

* **class extends继承**

## 原型和原型链

**普通对象和函数对象**

在javascript的世界中，全部都是对象，而对象之间也是存在区别的，我们一般区分普通对象和函数对象。

**原型**

原型也是一个对象，原型的作用，是实现对象的继承。

所有的js对象，都存在一个函数指针的属性，该属性指向实例对象的构造函数的原型。

Js的函数对象也是对象，所以它有一个原型指针，；除此之外，js的函数对象也有自己的原型。

普通对象的\_\_proto\_\_指向Object的prototype

函数对象的\_\_proto\_\_指向Function的prototype

函数对象的prototype的\_\_proto\_\_指向Object的prototype

Object的\_\_proto\_\_指向Function的prototype

Object的prototype的\_\_proto\_\_指向null

Function的\_\_proto\_\_指向Function的prototype

Function的prototype的\_\_proto\_\_指向Object的prototype

**原型链**

## Let、const和var区别

在HTML中，全局作用域是针对window对象；

使用var关键字声明的全局作用域变量属于window对象；

使用let关键字声明的全局作用域变量不属于window对象，使用window. 不能访问

**Let和const的异同点**

**相同点：**

二者都是块级作用域

都不能和它所有作用域内的其他变量或者函数拥有相同的名称

**区别：**

1、const声明的常量必须初始化，而let声明的变量不用。

2、const 只能进行一次赋值，即声明后不能再修改。

3、const 声明的是复合类型数据，可以修改其属性。

即const指针指向的地址不可以变化，指向地址的内容可以变化。

## 变量类型：原始类型、复合类型

6中基本数据类型：number、string、boolean、undefined、null、symbol

复合类型：Object、function、array

## 类型判断typeof、instanceof

**Typeof**

Typeof xxx的结果有7种：

Number、string、boolean、undefined、object、function、symbol

注意：

1. Typeof null的结果是object，虽然null是原始类型，而非引用类型

这是因为在js的最初版本中，使用的是32位系统，为了性能考虑使用低位存储变量的类型信息。000开头代表的就是对象，另外null表示为全零。因此将null错误的判断为Objec。

1. typeof 复合类型，除了function类型的结果是function，其他的都是object，注意typeof xxx并没有array这一项，typeof [] 为object。因此使用typeof 并不能准确的判断变量的类型。
2. es6新增：typeof symbol 类型的值，结果是symbol。

* **Instanceof**

用于实例和构造函数的对应

Function fun(){}

Var f = new fun()

F instanceof fun

上面介绍过typeof [] 的结果是object，无法判断出数组类型，但是可以用instanceof来判断。

[] instanceof Array //true

## 值类型、引用类型

根据变量类型传递方法，变量又可以分为值类型和引用类型。

值类型：存储的是在栈中的数据。在参数传递方式上，值类型是按值传递。

引用类型：真实的数据存放在堆内存里，存储的是该对象在栈中引用。在参数传递方法上，引用类型是按共享传递。

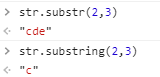
Typeof 识别出来的类型中，出去object和function是引用类型，其他的都是值类型。

## 字符串方法substr和sunstring的区别

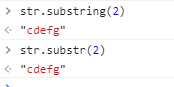
substr第一个参数是起始索引号，第二个参数是提取的长度；

substring 第一个参数是起始索引号，第二个参数是结束索引号。



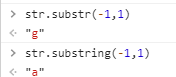


两者第一个参数一样，第二个参数为空时，结果一样



substr第一个参数为负数时，会反着提取值

substring第一个参数为负数时，相当于0



## 深拷贝/浅拷贝

* **概念**

1. 深拷贝和浅拷贝针对的是引用类型。基本类型的值存储在栈中，当复制时，栈内存会开辟一个栈内存。所以二者修改时，彼此不会影响。
2. 浅拷贝复制的是指向对象的指针，并没有开辟新的堆内存，原对象和新对象还是共享一块内存，修改新对象自然会影响原对象。

浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址

1. 深拷贝会开辟新的堆内存，原对象和新对象不共享同一块内存，修改新对象不会影响原对象。

* **浅拷贝的实现方式**

1、Object.assign()

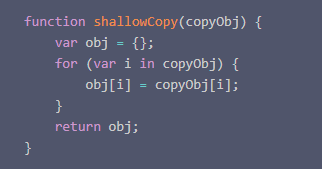
Object.assign()方法可以把任意多个的原对象自身可枚举属性拷贝给目标对象，然后返回目标对象。

2、Array.prototype.concat()

Contat()方法用于合并两个或多个数组。该方法不会更改现有数组，而是返回一个新数组

3、Array.prototype.slice()

4、循环复制



* **深拷贝的实现方式**

1. JSON.parse(JSON.stringify())

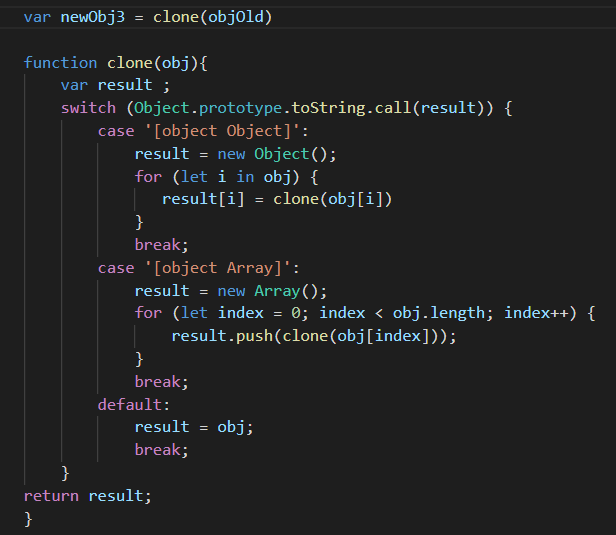
使用JSON.stringify将对象转成JSON字符串，在用JSON.parse()把字符串解析生对象。

此方法不能处理函数

1. 递归方法

遍历对象、数组，直到里面都是基本数据类型，然后再去复制，可实现深度拷贝。

该方法如果属性值是函数、undefined、symbol时，会忽略掉



1. jquery的$.extend()
2. js库loadsh中的\_.cloneDeep()

## setTimout和setInterval实现原理

setTimeout的主要思想是通过一个定时器，让函数在计时结束后再执行；

setInterval则是每隔一定的时间，就启动一次函数的执行；

setTimeout原理是：在指定的时间之后，把回调函数加入到任务队列末尾。

如果这个时间点任务队列中还存在其他的任务，那么该函数会在这些任务之后再执行，所以回调函数的实际执行时间大于等于指定的时间。

setInterval本质上就是每隔一定的时间向任务队列添加回调函数。但setInterval有一个原则：在向队列中添加回调函数时，如果队列中存在之前由其添加的回调函数，就放弃本次添加（不会影响之后的计时）。

## 函数的防抖、节流

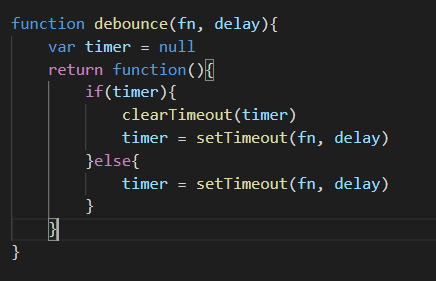
防抖和节流都是为了提升运行效率，减少函数的执行次数；

当某个事件频繁触发时，事件处理函数会频繁执行，如果处理函数有一些费时、耗性能的操作，就会导致页面出现卡顿甚至浏览器崩溃，这时就需要防抖和节流。

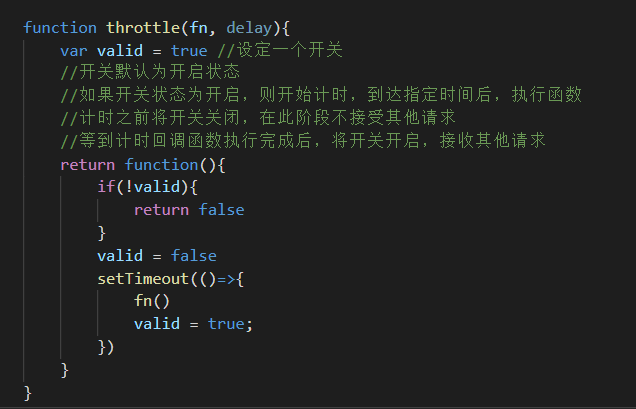
防抖：多次触发只执行最后一次；

节流：多次触发只执行第一次；

防抖：把多次函数执行，合并成一次执行。给定一个间隔时间，当两次函数执行的间隔时间大于给定的间隔时间，就执行一次函数。



节流：减少函数执行的频率。规定一个单位时间，在单位时间内触发一个事件回调，触发时超过间隔则执行，否则不执行。



防抖和节流的应用场景：

1. 搜索框input事件
2. 页面滚动时

相关库：undercore.js

## requestAnimationFrame

window.requestAnimationFrame()该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行

注意：若你想在浏览器下次重绘之前继续更新下一帧动画，那么回调函数自身必须再次调用window.requestAnimationFrame

* **通过requestAnimationFrame获取帧率**

<body>

    <div id="text"></div>

    <script>

        var lastLoop =  (new Date()).getMilliseconds();

        var count = 1,arr = [],fps = 0, index = 0

        function step(){

          var currentLoop = (new Date()).getMilliseconds(); //获取毫秒

          if(lastLoop > currentLoop){  //从新开始新的一轮0-10000

              fps = count;

              arr.push(fps)

              count = 1;

          }else{//0-1000毫秒

              count += 1

          }

          document.getElementById('text').innerHTML = lastLoop + currentLoop

          console.log(count,fps)

          lastLoop = currentLoop

          index = index + 1

          if(index < 1000){

            window.requestAnimationFrame(step);

          }else{

              console.log('每秒的帧数')

              console.log(arr) //(16) [54, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60]

          }

        }

        step()

    </script>

</body>

* FPS 每秒传输帧数（Frames Per Second），刷新率 单位HZ；
* 浏览器的重绘频率一般会和显示器的刷新率保持同步。大多数采用W3C规范，浏览器的渲染页面的标准频率也为60FPS即每秒重绘60次，
* 当页面隐藏或最小化时，会被暂停，页面显示，会进行执行。节省了CPU开销。

## new操作符

1. 创建一个新的对象
2. 将对象的\_\_proto\_\_执行构造函数的prototype
3. 将构造函数的this指向这个新对象，执行构造函数的代码，为这个对象添加属性、方法等
4. 返回新对象

## 什么是polyfill？

Polyfill是用于实现浏览器并不支持的原生API的代码

比如说Object.assign()是很多现代浏览器都支持的原生Web API，但是古老的浏览器并不支持，那么假设有人写了一段代码来实现这个功能是这些浏览器也支持了这个功能，那么这就可以成为一个polyfill。

## 事件代理/委托

是JavaScript中常用的绑定事件的常用技巧。

即是把原来需要绑定在子元素的响应事件（click、keydown…）委托给父元素，

让父元素担当事件监听的职务

事件代理的原理是DOM元素的事件冒泡

## 事件冒泡

有两个fatherDiv和sonDiv，fatherDiv是sonDiv的父节点；

fatherDiv绑定fatherClickEvent，sonDiv绑定sonClickEvent;

当点击sonDiv后会先执行sonClickEvent再执行fatherClickEvent，

这是因为sonDiv将click事件传递给了fatherDiv；类似这种事件向上传递的现象，就叫做**事件冒泡；**

**如何取消事件冒泡？**

 function stopBubble(e){

        e = e?e:window.event

        if(e && e.stopPropagation){

            e.stopPropagation()

        }

        e.cancelBubble = true

    }

## Ajax、Axios、Fetch的核心区别

Ajax = Asynchronous JavaScriptand XML 异步网络请求。

AJAX带来的最大影响就是页面可以无刷新的请求数据。

1. **JavaScript原生实现AJAX**

所有现代浏览器均支持XMLHttpRequest对象（IE5和IE6使用ActiveObject）

* **构造函数**

XMLHttpRequest()

该构造函数用于初始化一个XMLHttpRequest实例对象。在调用下列任何其他方法之前，必须先调用该构造函数。

* **属性**

此接口继承了XMLHttpRequestEventTarget和EventTarget的属性。

|  |  |
| --- | --- |
| onreadystatechange |  |
| readyState | 代表请求状态码   1. unsent 代理被创建，但尚未调用open方法； 2. opened open方法已经调用； 3. headers\_recelived send()方法已经调用，并且头部和状态已经可获得； 4. loading 下载中，reponseText属性已经包含部分数据； 5. done 下载操作已完成。 |
| response | 返回一个ArrayBuffer、Blob、Document，或DOMString，具体是哪种类型取决于responseType的值。其中包含整个响应体。 |
| responseText | 返回一个DOMString，该DOMString包含对请求的响应，如果请求未成功或尚未放送，则返回null。 |
| responseType | 用于定义响应类型的枚举值  1、“” 空字符串，采用默认类型DOMString，与设置为text相同；  2、arraybuffer response是一个包含二进制数据的JavaScript ArrayBuffer；  3、blob response是一个包含二进制数据的blob对象；  4、document response是一个HTMLDocument或XML XMLDocument，这取决于接收到的数据是MIME类型。  5、json  6、text  7、ms-stream 下载流的一部分 |
| responseURL | 返回经过序列化的响应URL，如果该URL为空，则返回空字符串 |
| responseXML | 返回一个Document，其中包含该请求的响应，如果请求未成功，尚未发生或不能解析为XML或HTML，则返回null。 |
| status | 整形数字，代表请求的响应状态 |
| statusText | 包含完整的响应状态文本，如200 K |
| timeout | 表示该请求的最大请求时间（毫秒），若超出该请求，请求会自动终止 |
| ontimeout |  |
| upload | 代表上传进度 |
| withCredentials / krer den chuo s/ | 一个布尔值指定跨域请求是否应该带有授权信息，如cookie或授权header头 |

* **方法**

|  |  |
| --- | --- |
| abort() [əˈbɔːt] | 如果请求已被发出，则立即中止请求 |
| getAllResponseHeaders |  |
| getResponseHeader() | 返回包含指定响应头的字符串 |
| open() | 初始化一个请求 |
| overrideMimeType | 覆写有服务器返回的MIME类型 |
| end() | 发送请求，如果请求是异步的，那么该方法将在请求发送后立即返回 |
| setRequestHeader() | 设置HTTP请求头的值，必须在open之后，send之前调用该方法。 |

* **事件**

可以设置on\*属性，也提供标准的监听器addEventListener()API来监听事件。

|  |  |
| --- | --- |
| abort | 当request被停止时触发，如当程序调用XMLHttpRequest.abort()时，也可以使用onabort属性。 |
| error | 当request遭遇错误时触发。也可以使用onerror属性。 |
| load | XMLHttpRequest请求成功完成时触发。也可以使用onload属性。 |
| loadend | 当请求结束时触发，无论请求成功（load）还是失败（abort或eoor）。也可以使用onloadend属性。 |
| loadstart | 接收到响应数据时触发。  也可以使用onloadstart属性。 |
| progress [ˈprəʊɡres] | 当请求接收到更多数据时，周期性地触发。  也可以使用onprogress属性。 |
| progress事件会在请求接收到数据的时候被周期性触发。 |
| Timeout | 在预设时间内没有接收到应用时触发。  也可以使用ontimeout属性。 |

1. **jquery实现AJAX**

Jquery Ajax 是对原生XHR的封装，除此之外还增添了对JSONP的支持；

Ajax本身是针对MVC的编程，不符合现在前端MVVC的趋势；

很多情况下我们只需要使用ajax，但是却需要引入整个jquery，这是非常不合理的，于是便有了fetch的解决方案。

Jquery ajax 函数库

<https://www.w3school.com.cn/jquery/jquery_ref_ajax.asp>

1. **Axios是一个基于promise用于浏览器和nodejs的HTTP客户端；**

<http://www.axios-js.com/zh-cn/blogs/>

本质上也是对原生XML的封装，只不过它是promise的实现版本，符合最新的ES标准，有以下特点：

从浏览器中创建XMLHttpRequest

从node.js创建http请求

支持promise API

拦截请求和响应

转换请求数据和响应数据

取消请求

自动转换JSON数据

客户端支持防御XSRF

如何封装axios

* Axios的拦截器：interceptors [ˌɪntəˈsɛptəz]

添加请求拦截器：axios.interceptors.request.use()

1. **Fetch是ES6新增的通信方式，不是ajax，但是它本身实现数据通信，也是基于Promise管理的。**

Fetch 是一种HTTP数据请求的方式，是XML httpRequest的一种替代方案；

不是Ajax的进一步封装，而是原生JS，没有使用XML httpRequest对象。

串行：请求是异步的，需要等到上一个请求成功，才能执行下一个请求

并行：同时发送多个请求，HTTP请求可以同时进行，但是JS的操作都是一步步的，因为JS是单线程，等待所有请求都成功，我们再去做什么事情。

**Promise.all([**

**])**

**重写Promise.all()**

**同时并发几百个请求，怎么处理**

**Asyncpool**

## CommonJs模块和Es6模块的区别

* **CommonJS**

1. 对于基本数据类型，属于复制。及会被模块缓存。同时，在另一个模块可以对该模块输出的变量重新复制。
2. 对于复杂数据类型，属于浅拷贝。由于两个模块引用的对象指向同一个内存空间，因此对该模块的值做修改时会影响另一个模块。
3. 当使用require命令加载某个模块时，就会运行整个模块的代码。
4. 当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJs模块无论加载多少次，就只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。

* **ES6模块**

1. es6模块中的值属于【动态只读引用】
2. 对于只读来说，即不允许修改引用变量的值，import的变量是只读的，不论是基本数据类型还是复杂数据类型。当模块遇到import命令时，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。
3. 对于动态来说，原始值发生变化，import加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。

## 模块的循环加载

1. CommonJS模块的加载原理

* CommonJS 的一个模块，就是一个脚本文件。require命令第一次加载该脚本，就会执行整个脚本，然后在内存生成一个对象。
* 内存中缓存的对象的属性有：

1. id：模块名
2. exports： 模块输出的各个接口
3. loaded： 布尔值，表示模块的脚本是否执行完毕
4. …

* 以后要用到这个模块的时候，会在exports属性上面取值。
* 即使再次执行require命令，也不会再次执行该模块，而是到缓存之中取值。

1. CommonJS模块的循环加载

CommonJS的做法是：一旦出现某个模块对“循环加载”，就只输出已经执行的部分，还未执行的部分不会输出。

1. ES6 模块的循环加载

ES6模块的运行机制与CommonJS不一样，它遇到模块加载命令import时，不会去执行模块，而是只生成一个引用，等到真正需要用到时，再到模块里面去取值。

ES6根本不需要关系是否发生“循环加载”。

## Web-Worker

**一．主线程**

* 主线程采用new命令，调用Worker()构造函数，新建一个Worker线程；

Worker()构造函数的参数是一个脚本文件，该文件就是Worker线程所要执行的任务；

var worker = new Worker('/hash.js')

* 主线程通过调用worker.postMessage()方法，向Worker发消息；

worker.postMessage()方法的参数，就是主线程传给Worker的数据。

* **主线程通过worker.onmessage知道监听函数，接收子线程发回来的消息。**
* **主线程通过worker.onerror监听worker是否发生错误；**
* **主线程通过worker.terminate()关闭worker。**

**二、子线程**

* **Worker线程内部需要有一个监听函数，监听message事件**

|  |
| --- |
| **self.addEventListener(‘message’, function(e){**  **self.postMessage(‘you said:’ + e.data)**  **},false)** |

**Self代表子线程自身，即子线程的全局对象。**

**除了使用self.addEventListener()指定监听函数，也可以使用self.onmessage指定。**

* **Worker内部通过importScripts()方法加载其他脚本。**
* **Worker内部通过self.close()关闭。**

## **<script>标签属性defer和async**

当浏览器碰到script脚本的时候：

* 没有defer或者async，浏览器会立即加载并执行指定的脚本，“立即”指的是在渲染该script标签之下的文档元素之前，也就是说不等待后续载入的文档元素，读到就加载并执行。
* 有async，加载和渲染后续文档元素的过程和a.js的加载与执行并行进行（异步）。

执行a.js时，html解析暂停。

* 有defer，加载后续文档元素的过程和a.js的加载并行进行，但是a.js的执行要在所有元素解析完成之后，DOMContentLoaded事件触发之前完成。

HTML、JavaScript和css中，除了css，没有文件可以边下载、边解析。

## 懒加载（延迟加载）图片的原理

在图片没有进入可视化区域时，先不给src赋值，这样浏览器就不会发送请求了，

等到图片进入可视化区域再给src赋值，

图片的真实地址需要存储在data-src中，

图片没有进入可视区域，也就是说图片的offsetTop需要小于页面的可视高度，

当页面存在滚动条的时候，则需要满足img.scrollTop小于页面的可视化区域高度 + 页面滚动的高度。

## 异步解决方案

1. 回调函数
2. 发布订阅
3. Promise
4. Generator
5. Aysnc/await

## 柯里化和反柯里化

* 概念：柯里化就是把接收多个参数的函数变成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下参数而且返回结果的新函数的技术。
* 好处

1. 参数复用
2. 提前返回
3. 延迟计算/执行

## 高阶函数实现函数缓存（备忘模式）

function cacheFunction(fn){

    let caches = {}

    //扩展运算符(...): 将一个数组转化为用逗号分割的参数序列

    return function(...args){

        let \_args = JSON.stringify(args)

        if(!caches[\_args]){

            caches[\_args] = fn.apply(fn,args)

        }

        return caches[\_args];

    }

}

function add(a,b){

    return a+b;

}

let result = cacheFunction(add)(2,3)

## 设计模式

<https://segmentfault.com/a/1190000012422055>

## target和currentTarget的区别，以及currentTarget为null的问题

1. target：点击的元素
2. currentTarget：绑定事件的元素

事件绑定在外面的button上时，currentTarget为ul

当点击事件点在外面的button上时，target也为button等于currentTarget

当点击事件点在里面的span上时，target为span，currentTarget还是为button，两者不同。

* **CurrentTarget在浏览器打印出来的event中为null的问题**

解决方案：直接使用就可以，在浏览器控制台看不到，但实际是存在的。

## Onload和DomContentLoaded

当onload事件触发时，页面上所有的dom、样式表、脚本、图片、flash都已经加载完成了。

当DomContentLoaded事件出发时，仅当dom加载完成，不包括样式表、图片、flash

数组sort