js部分

- ## 简述同步和异步的区别

同步：在主线程上排队的任务，只有前一个任务执行完成，才会去执行下一个任务

```
for(let i = 0 ; i < 10 ; i++) {
    console.log(i)
}
console.log('执行for循环再执行console')//先输出1~9 再输出 执行for循环再执行console
```

异步：执行顺序不确定，由触发条件确定，异步任务不进入主线程上排队，而是进入任务队列，当任务队列通知主线程某个任务可以执行了，这个任务才会进入主线程，四种属于异步，如下

1. 回调函数（将B函数作为参数的形式传给A函数，当A函数执行完成再执行B函数）

```
function f1(callback){
    console.lolg('f1')
    setTimeout(function(){
        callback()
    },300)
}
function f2(){
    console.log('f2')
}
f1(f2) //f1  f2
```

2. 事件监听（这种方式下，异步事件的执行取决于某个事件是否发生，而不取决于代码的顺序），如点击事件

3. 发布/订阅（观察者模式），定义对象间一对多的依赖关系，当某一个对象改变，所有依赖于他的对象都会得到通知

4. promise（es6）

```
var Pro = function(){
    return new Promise(resolve , reject){
        console.log('start')
        resolve('成功') // pro.then
        reject('失败') // pro.catch
    }
}
Pro.then((data)=>{
    console.log('success11:'+data)
})then((data)=>{
```

```
    console.log('success22:'+data)
})
//start --> success11:成功 --> success22:成功

Pro.catch((data)=>{
    console.log('success11:'+data)
})catch((data)=>{
    console.log('success22:'+data)
})
//start --> success11:失败 --> success22:失败
```

5.async/await（es7）

1. async函数会返回一个Promise对象，如果函数中直接return一个直接量，会把这个直接量通过Promise.resolve()直接封装成一个Promise,可以同Promise最原始的then接收数据，如果没有返回值，会接收到一个undefined

```
async function AsyncFun () {
    return 'hello async'
}
AsyncFun().then(data=>{
    console.log(data) //hello async
})


async function AsyncFun () {
}
AsyncFun().then(data=>{
    console.log(data) //undefined
})
```

2.await:等待的是一个表达式，没有特殊说明

- [1] 如果await等到的不是一个Promise对象，那await等待的就是这个表达式的运算结果
- [2]如果await等到的是一个Promise对象，await会阻塞后面的代码，等待Promise的resolve，得到这个resolve的值作为await的表达式

```
function f1 () {
    return '111'
}
async function f2 (){
    return new Promise((resolve , reject) => {
        resolve('222')
    })
}
async function f3 (){
    return new Promise((resolve , reject) => {
    setTimeout(() = > resolve('222'),2000)
        resolve('333')
    })
}
```

```
async function test () {
    const a = await f1()
    const b = await f2()
    const c = await f3()
    console.log(a , b) // 111   222

    console.log(c)
    console.log(a,b)
    //333 111 222
}
```

- 优势：处理then链，Promise多个函数的请求参数传递太复杂，如下

1. Promise请求

```
function taskLongTime (n) {
    return new Promise((resolve , reject) =>{
        setTimeout(() =>resolve(n+200) , n)
    })
}
function step1(n){
    console.log(`the step1 is ${n}`)
    return taskLongTime(n)
}
function step2(n){
    console.log(`the step2 is ${n}`)
    return taskLongTime(n)
}
function step3(n){
    console.log(`the step3 is ${n}`)
    return taskLongTime(n)
}
function doStep(){
    console.time('timeIs')
    const time1 = 200
    step1(time1)
    .then(time2=>step2(time2))
    .then(time3=>step3(time3))
    .then(result=>{
        console.log(`result is ${result}`)
        console.timeEnd('timeIs')
    })
}
doStep()
//the step1 is 200
//the step2 is 400
//the step3 is 600
//result is 800
//timeIs: 1205.845947265625ms
```

2. async/await

```
async function doStep(){
    console.time('timeIs')
    const time1 = 200
    const time2 = await step1(time1)
    const time3 = await step2(time2)
    const result = await step3(time3)
    console.log(`result is ${result}`)
    console.timeEnd('timeIs')
}

//the step1 is 200
// the step2 is 400
// the step3 is 600
// result is 800
// timeIs: 1211.1494140625ms
```

- **怎么添加、删除、复制和查找节点**

查找节点：css选择器 querySeletor用法

```
<div class='box' id='box' name='box'></div>
<div class='box1' id='box1' name='box1'></div>
//根据id
var domId = document.getElementById('box')
//根据class名字
var domClass = document.getElementsByClassName('box')
//根据name属性,返回的是这个元素的数组，即nodeList
var domName = document.getElemetsByName('box')
//根据标签的tagName查找元素，返回一个HTMLCollection集合
var domTag = document.getElementsByTagName('div')
//指定一个或者多个css选择器，来获取节点对象，css选择如下面链接
var domselect = document.querySelector('div')
//通过匹配css选择器，来获取节点对象集合(NodeList)
var domSelectAll = document.querySelectorAll('.box , .box1')
```

添加节点

```
var div = documnent.getElementById('div')
var p = document.createElement('p')
p.innerHTML = 'p0'
var p1 = document.createElement('p')
p1.innerHTML = 'p1'
var p2 = document.createElement('p')
p2.innerHTML = 'p2'
var h2 = document.createElement('h2')
h2.innerHTML = 'h2'
//在父节点末尾添加 obj.appendChild(node)
div.appendChild(p1)
div.appendChild(p2)
//在父元素中间添加 obj.insertBefore(node,existingnode)
div.appendChild(p)
```

```
div.insertBefore(p1 , p)
//新节点替换某个子节点 obj.replaceChild(newnode , oldnode)
div.replaceChild(h2 , p)
```

删除节点 removeChild()

- **实现一个clone函数对JavaScript的五种主要数据类型（number，string，Boolean，Array，Object）的的复制**

```
let testObj = {
    a:'string',
    b:123,
    c:true,
    d:[1,2,3],
    e:{
        f:true,
        g:[4,5,6]
    },
    h:null,
}
```

反序列法（JSON.parse(JSON.stringify(ob)))

```
function deepClone(obj){
    let deep = JSON.parse(JSON.stringify(obj))
    return deep
}
deepClone(testObj)
```

```
function cloneType(obj){
    if(typeof(obj) === 'string' || typeof(obj) === 'number' || typeof(obj) === 'boolean'
|| obj === null){
        return obj
    }
    if(Object.prototype.toString.call(obj) === '[object Array]'){
        return obj.map(v => cloneType(v))
    }
    if(Object.prototype.toString.call(obj) === '[object Object]'){
        return Object.keys(obj).reduce((prev , key)=>{
            prev[key] = cloneType(obj[key])
            return prev
        },{})
    }
}
```

- **数组去重**

```
let arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null,
NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}]
```

1. for嵌套for循环

```
function unique(arr){
    for(let i = 0 ; i < arr.length ; i++){
        for(let j = i + 1 ; j < arr.length ; j++){
            if(arr[i] == arr[j]){
                arr.splice(j,1)
                j--
            }
        }
    }
    return arr
}
//[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}]     NaN和{}没有去重，两个null直接消失了
```

Set去重（es6）

```
let newarr = Array.from(new Set(arr))
<!--或者以下-->
let newarr = [...new Set(arr)]
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

indexOf去重

```
function unique(arr){
    if(!Array.IsArray(arr)){
        throw new Error('不是数组')
    }
    let newArr = []
    for(let i = 0 ; i < arr.length ; i ++){
        if(newArr.indexOf(arr[i] == -1){
            newArr.push(arr[i])
        }
    }
    return newArr
}
console.log(unique(arr))
// [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {…}, {…}]
//NaN、{}没有去重
```

相邻排序去重（sort（））

```
/**
sort()方法进行排序
比较会打乱顺序，temp每次和arr比较最后一个，不排序会导致比较缺失
*/
function unique(arr){
    arr.sort()
    for(let i = 0 ; i < arr.length ; i ++){
        if(arr[i) !==temp[temp.length - 1]){
            temp.push(arr[i])
        }
    }
    return temp
}
```

includes

```
function unique(arr){
    let newArr = []
    for(let i = 0 ; i < arr.length ; i ++){
        if(!newArr.includes(arr[i])){
            newArr.push(arr[i])
        }
    }
    return newArr
}
```

- 闭包函数

定义：一个函数嵌套另一个函数，这个函数对于其容器是私有的，一个闭包可以有自己的独立环境和变量的表达式，能够读取其他函数内部变量的函数

特性：

1. 函数嵌套函数
2. 参数和变量不会被垃圾回收机制回收
3. 内部函数可以引用外部函数的变量和参数

优点：

1. 避免造成全局污染
2. 希望一个变量长期储存在内存中
3. 私有成员的存在

缺点：

1. 常驻内存，增加内存使用量
2. 使用不当会造成内存泄露

```
<!--案例1-->
function outter(){
    var speed = 1
    function inner (){
        speed ++
        console.log(speed)
    }
    return inner
}
var speedFn = outter()
speedFn() //2
speedFn() //3
speedFn() //4
//改造过后
var Outter = (function(){
    function inner(speed){
        speed ++
        console.log(speed)
    }
    return inner
})()
Outter(1) //2
Outter(2) //3
Outter(3) //4


<!--案例2-->

<html>
<!--在这里插入内容-->
    <div>
        <p>111</p>
        <p>222</p>
        <p>333</p>
    </div>
    <script>
        var dom = document.getElementsByTagName('p')
        for(var i = 0 ; i < dom.length ; i++ ){
            dom[i].onclick = function(){
                console.log(i) // 3
            }
        }
        //改造过后
        <!--用let-->
        for(let i = 0 ; i < dom.length ; i++ ){
            dom[i].onclick = function(){
                console.log(i) //点击哪个p标签得到哪个的下标
            }
        }
        <!--闭包-->
        for(let i = 0 ; i < dom.length ; i++ ){
            dom[i].onclick = function(i){
                return function(){
                    console.log(i) //点击哪个p标签得到哪个的下标
                }
            }(i)
```

```
        }
    </script>
</html>
```

- ## 用递归函数写1-100的累加

```
function sum (num){
    if(num == 1) return 1
    else num + sum(num - 1)
}
console.log(sum(100)) // 5050
```

- ## 如何判断数据类型

1. typeof()：返回六种数据类型（object , string , number , boolean , function , undefined ）

```
//   object ,     object    ,  string     ,  number   , undefined
typeof(null),typeof([1,2,3]),typeof('123'),typeof(123),typeof(undefined)
//boolean   ,      function      , object
typeof(true),typeof(function(){}),typeof({})
```

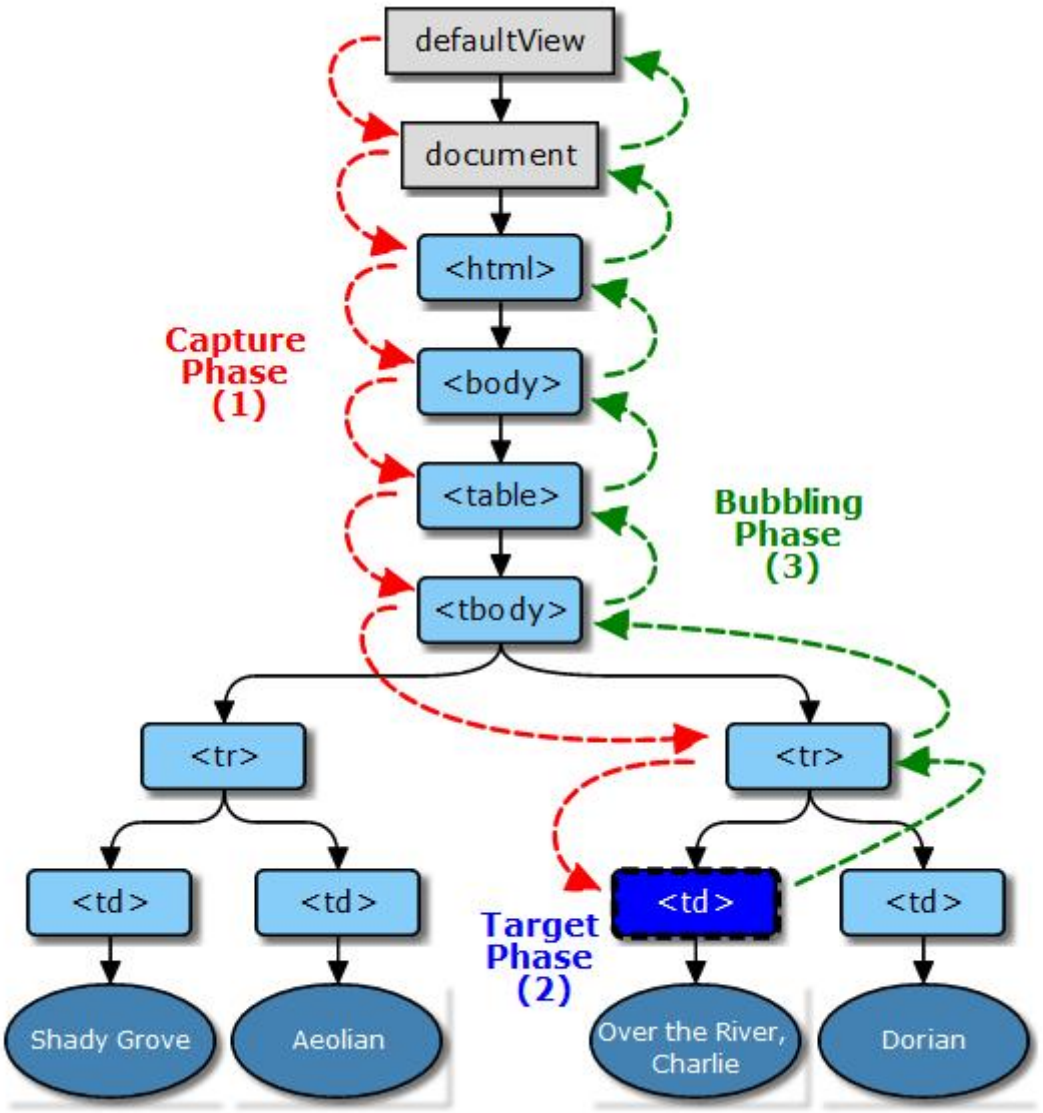
2.Object.prototype.toString.call()

```
Object.prototype.toString.call(null), //[object Null]
Object.prototype.toString.call([1,2,3]), //[object Array]
Object.prototype.toString.call('123'), //[object String]
Object.prototype.toString.call(123), //[object Number]
Object.prototype.toString.call(undefined) //[object Undefined]
Object.prototype.toString.call(true), //[object Boolean]
Object.prototype.toString.call(function(){}), //[object Function]
Object.prototype.toString.call({}) // [object Object]
```

- ## js事件委托（事件代理）是什么？其原理是什么？

事件委托：字面上的意思就是自己的事情，委托给别人去做，利用事件冒泡的原理，把具体dom上面的事件委托到更大更广泛的dom去处理，就像快递员一样，一家一家派送非常繁琐，可以交给一个更大的代收点，比如驿站，那事情就变得很简单

原理：如下图



1.捕获阶段（红色箭头）

2.冒泡阶段（绿色箭头）

在事件捕获阶段，事件从defaultView（整个页面）一直传播到具体的目标（target），从广泛到具体。 在冒泡阶段，事件从事件源（target）一直传播到defaultView，从具体到广泛，就像冒泡一样，范围越来越大，影响越来越大

```
<html>
<!--在这里插入内容-->
<div>
    <ul>
        <li>111</li>
        <li>222</li>
        <li>333</li>
    </ul>
</div>

<script>
```

```
<!--普通写法-->
    var Oul = document.getElementsByTagName('ul')
    var Oli = document.getElementsByTagName('li')
    for(let i = 0 ; i < Oli.length ; i ++){
        Oli[i].onclick = function(){
            console.log(i)
        }
    }
    <!--事件代理-->
    for(let i = 0 ; i < Oli.length ; i ++){
        Oli[i]..onclick = function(ev){
        var ev = ev || window.event
        var target = ev.target || ev.srcElement;
        if(target.nodeName.toLowerCase() == 'li'){
             console.log(i)
        }
        }
    }
</script>
</html>
```

- **this指向**

定义：this指向在定义的时候是确定不了的，只有当函数执行的时候才能决定this指向谁

1. 单独的this

```
console.log(this) //window
```

2. 全局函数的this

```
function demo(){
    console.log(this) // window
}
```

3. 严格模式下的 this

```
function demo(){
    'use strict'
    console.log(this)
}
demo() // undefined
```

4. 函数调用时，在前面加上一个new，即构造函数,this就指向这个对象

```
function demo(){
   this.aaa = 'fffffff'
}
```

```
var a = new demo()
console.log(a) //  {aaa:'fffffff'}
```

5. 用call和apply的方式调用函数

```
function demo (){
    console.log(this)
}
demo.call('abc') //String{"abc"}
demo.apply(123) //Number{123}
demo.call([1,2,3]) //[1,2,3]
```

6. 定时器中的this指向window
7. 元素绑定事件，事件触发后，执行函数中的this指的是当前元素
8. bind绑定的事件，this指向函数中绑定的事件

```
window.onload = function() {
 let $btn = document.getElementById('btn');
 $btn.addEventListener('click',function() {
 alert(this); // window
 }.bind(window))
}
```

9.对象中的方法，哪个对象调用了该方法，this就指向哪个

```
let obj = {
    name:'luoli',
    getName:function(){
        console.log(this)
    }
}
obj.getName() // luoli
let fn = obj.getName
console.log(fn)   //f (){ console.log(this)}
fn() // window
```

- **如何改变函数内部的this指向**

1. apply() 和 call()

apply() : 有两个参数，第一个是要改变的this指向的那个对象，第二个是一个数组，即apply(obj,[arg1,arg2,arg3,..]),传递的参数用数组包裹

call() : 有无数个参数，第一个参数是要改变的this指向的那个对象，即call(obj,arg1,arg2,arg3,.....)

```
<!--this指向的对象没有标明-->
function demo (x,y,z){
```

```
    console.log(x,y,z) //2,3,undefined
    console.log(this.x,this.y,this.z) // undefined  undefined undefined
}
demo.call(1,2,3)
demo.apply(1,[2,3])

<!--this指向的对象标明-->
function demo (x,y,z){
    console.log(x,y,z) //2,3,undefined
    console.log(this.x,this.y,this.z) // 1  2 3
}
let obj = {
    x:1,
    y:2,
    z:3
}
demo.call(obj,2,3)
demo.apply(obj,[2,3])
```

2. **bind()**

定义：bind()改变this作用域会返回一个新的函数（不会改变原函数，直接拿新返的函数来用），和call()的传参一样

```
function demo (x,y,z){
     console.log(x,y,z) //1,2,undefined
     console.log(this.x,this.y,this.z) // 1  2 3
}
let obj = {
    x:1,
    y:2,
    z:3
}
demo.bind(obj,1,2) //这样没有打印任何东西，输出这个表达式，代码的是demo函数
var fn = demo.bind(obj,1,2)
fn()
```

- **跨域的方式及原理**

同源策略：是浏览器最基本的安全功能，如果少了同源策略，浏览器容易受xss,csrf的攻击，同源指的是 ‘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。一个域名地址的组成如下图：

同源策略限制的内容有

1. cookie、localStrong等储存性内容
2. dom节点
3. ajax发送请求后，被浏览器拦截

有三个标签允许跨域加载资源

1. <link href=''></link>
2. <script src=''></script>
3. img标签

常见跨域场景

URL	说明	是否允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://70.32.92.74/b.js	域名和域名对应ip	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许（cookie这种情况下也不允许访问）
http://www.cnblogs.com/a.js http://www.a.com/b.js	不同域名	不允许

注意：如果是协议和端口跨域，前台是不能解决的、

- 跨域解决方法

1. jsonp **原理**：利用script标签没有跨域漏洞的限制，可以请求到其他来源动态产生的json数据，不过jsonp需要对方的服务器做支持才能够请求

   jsonp和ajax对比：jsonp和ajax都是向服务器发送请求，从服务端请求数据，但ajax属于同源策略，jsonp是非同源

   jsonp的优缺点：简单兼容性好，能解决主流浏览器的跨域访问问题，缺点是只能支持get请求，容易xss攻击

```
function jsonp({url , params , callback}){
    return new Promise ( (resolve , reject) => {
        let script = document.createElement('script')
        window[callback] = function (data) {
        <!--请求成功，回调函数返回数据-->
            resolve(data)
            document.body.removeChild(script)
        }
        params = {...params , callback}
        let arr = []
        for(let key in params){
            arr.push(`${key} = ${params[key]}`)
        }
        script.src = `${url}?${arr.join('&')}`
        document.body.appendChild(script)
    })
}

jsonp({
    url:'http:localhost:3000/say/',
    params:{
        wd:'i am luoli'
    },
    callback:'showfun'
}).then(data=>{
    console.log(data)
})
```

2.cors请求

开启cors：设置Access-Control-Allow-Origin 就可以开启cors

简单请求：

条件1 ：

- get请求
- head请求
- post请求

条件2 ： content-type的值只能是下面三者之一

- text/plain

- multipart/form-data
- application/x-www-form-urlencoded

复杂请求：不满足以上条件都属于复杂请求

```
let xhr = new XMLHttpRequest()
document.cookie = 'name=luoli' //cookie不能跨域
xhr.withCredentials = true // 前端设置是否带cookie
xhr.open('put','http://localhost:4000/getData/',true)
xhr.setRequestHeader('name','requestHeader')
xhr.onreadystatechange = function () {
    if(xhr.readyState == 4){
        if((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            console.log(xhr.response)
        }
    }
}
xhr.send()
```

3. websocket

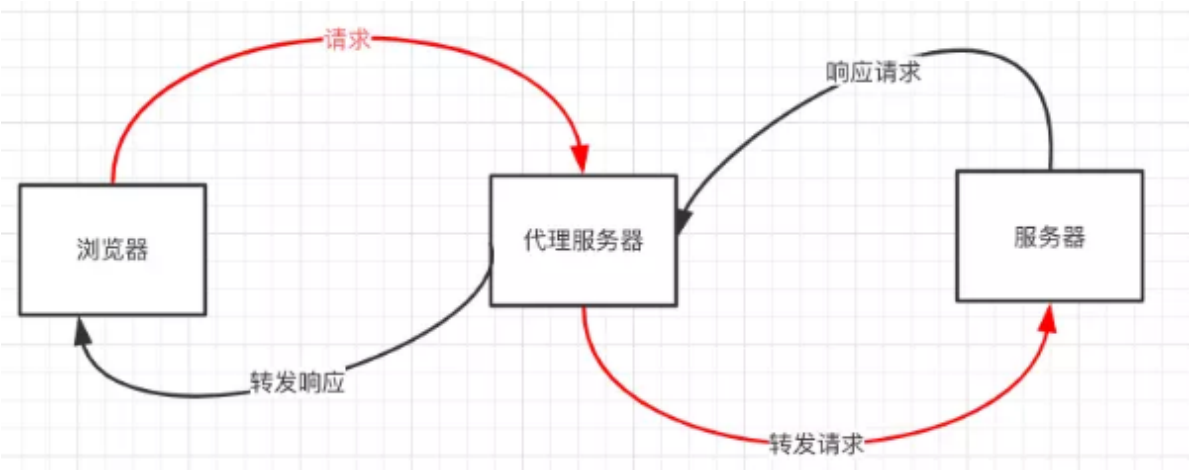
定义：websocket是HTML5的一种持久化协议，实现了浏览器和服务器的全双工通信，websocket和http都是应用层协议，都是基于tcp的，但是websocket是一种双向通信协议，建立成功过后，websocket的server和client都能主动发送和接收数据，但建立websocket连接时，需要http协议的帮助，其余的与http就无关了

4. node中间件代理（两次代理）

同源策略是浏览器标准，如果是服务器向服务器发送请求就不遵循同源策略，即node作为一个中间件，来传递客户端发送的数据到服务器，服务器响应数据给中间件，中间件再返回给客户端

以下是中间件（代理服务器）的作用：

- 接收客户端请求
- 将请求发给服务器
- 服务器接收请求，响应请求给代理服务器
- 代理服务器把响应转发给客户端



5.nginx反向代理：类似于node中间件，需要搭建一个nginx代理服务器，用于转发请求

- **垃圾回收机制的方式及内存管理**

垃圾回收机制：即找出那些不需要再用的变量，释放其占用的内存，所以，垃圾收集器会按照固定的时间间隔（或者代码执行中预定的收集时间），周期性的执行这一操作，浏览器中有两种垃圾回收机制方式，如下：

```
<!--fn1定义了一个局部变量，fn1引用过后就释放了这个变量
    fn2引用过程中，返回了一个全局变量，所以这个变量不会被垃圾回收机制释放
-->
function  fn1 (){
    var str = '123'
}
function fn2 () {
    var str = '123'
    return str
}
fn1()
fn2()
```

1. 标记清除

定义：标记清除有两个概念（进入环境和离开环境），进入环境是指变量进入执行的环境，离开环境是指变量完成执行，离开所执行的环境，垃圾收集器在运行的时候会给所有的变量加上标记，当变量在环境中或者被环境变量引用的变量，这些标记会被去掉，其余的就是被当做被回收的变量

2. 引用计数

定义：跟踪一个值的引用次数，当定义一个变量，并将引用类型赋值给这个变量，该值的引用次数加1，当该变量指向其他的值时，这个值减1，当这个值的引用次数等于0是，垃圾收集器就会回收这个值，但这个方法会引起内存泄露(不能回收也不能使用)，因为他不能回收循环引用的问题，如下

```
function demo (){
    var a = {}
    var b = {}
    a.prop = b
    b.prop = a
}
```

内存管理

为了给浏览器更好的性能，优化内存的最好方式就是保存代码中必要的数据，不要的就全部设置为null，这种方法叫解除引用（dereferencing）

```
function demo(){
    var obj = {}
    obj.name = '111'
```

```
}
var demoFun = demo()
demmoFun = null
```

- **写一个函数，去除字符串前后的空格**

```
function trim(){
    var str =' abc '
    var len = str.split('')
    return str.split('').slice(1,len -1).join('')
    <!--或者-->
    return str.trim()
}

function trim(){
    var str =' abc '
    return str.resplace(/(^\s*) | (\s*$)/g,'')
}
```

- **判断一个变量是否是数组**

1. instanceof()

```
var arr = [1,2,3,4]
let isArray = arr instanceof Array
```

2. Array.isArray()

```
var arr = [1,2,3,4]
let isArray = Array.isArray(arr)
```

3. Object.prototype.toString.call()

```
var arr = [1,2,3,4]
let isArray = Object.prototype.toString.call(arr)
```

- let , var , const 的区别

let和var的区别：

var：

1. var和function定义的变量会存在变量提升，而let不会
2. 作用域不同，var是函数作用域，let是块级作用域

3. 暂时性死区：在块级作用域中，规范我们把变量定义在作用域的最开始。暂时性死区就是当我们在块级作用域中let了一个变量时，在其他块级作用域定义同名的变量时互相不受影响，定义的这个变量就属于这一个块级里面
4. let声明的变量不允许重复声明变量

var和const的区别：

const和let一样，也是块级作用域，也存在暂时性死区，而且只声明不赋值会报错，声明一个变量，不允许重新赋值，但如果声明一个对象，对象里面的属性可以重新赋值的

- 箭头函数和普通函数的不同

1. 外形不同
2. 箭头函数全是匿名函数

```
<!--普通函数-->
function demo(){
    log('具名函数')
}
let demo = function(){
    log('匿名函数')
}
<!--箭头函数-->
let demo = ()=>{
    log('匿名函数')
}
```

3. 箭头函数不能用于构造函数

```
let demo = ()=>{

}
log(new demo()) //demo is not a constructor
```

4. 箭头函数不具备arguments对象，用rest参数

```
function demo(a){
    log(arguments)
}
demo([1,2,3]) //[Array(3), callee: ƒ, Symbol(Symbol.iterator): ƒ]

let demo = (a) =>{
    log(arguments) //Uncaught ReferenceError: arguments is not defined

}
let demo = (...a) =>{
    log(a) //[1, 2, 3, 4, 5]
```

```
}
```

5. 箭头函数没有原型链

- **new操作符到底做了什么?**

new一共经历了四个过程

```
function fn(){

}
var fnobj = new fn()
```

1. 创建一个空的对象

```
var obj = new Object()
```

2. 将所创建的对象的_proto_指向构造函数的prototype

```
obj._proto_ = fn.prototype
```

3. 将构造函数的this指向obj，并指向函数

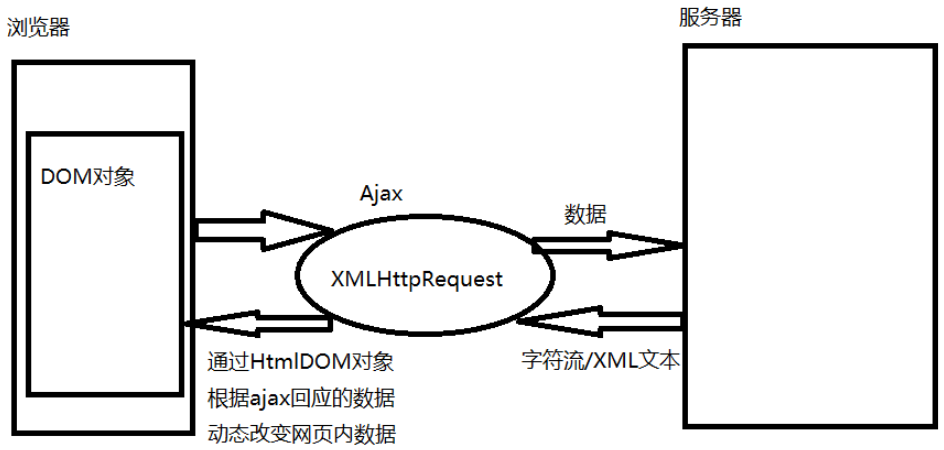
```
var result = fn.call(obj)
```

4. 判断函数的返回值类型，返回的是对象就用该对象，没有的话就创建一个对象

```
if(typeof(result) == 'object'){
    fnobj = result
}else{
    fnobj = obj
}
```

- **ajax原理**

ajax的原理：相当于在浏览器和服务器之间加了一个中间层（ajax引擎），当需要新的数据时，由ajax引擎像服务器提交请求，实现浏览器和服务器之间的数据交换



ajax核心：XMLHttpRequest对象

```
var xhr = new XMLHttpRequest()
xhr.open('get',url,async)
xhr.send(data)
xhr.onreadystatechange = function(){
//onreadystatechange表示请求状态改变的触发器
    if(xhr.readystate == 4){
    //readystate表示请求状态完成
        if(xhr.status == 200){
            log(xhr.responseText)
        }
    }
}
```

- **模块化开发**

1. 模块就是一个特定的功能文件，通过加载这些模板得到特定的功能
2. 模块开发就是js的功能分离，通过需求引入不同的文件
3. 模块开发可以降低代码的耦合性，避免代码在页面的多次使用，

遵循的规则：

1. ADM规范：异步模板加载规范，在此规范下模块会异步加载且不会影响后面的语句的执行，可以用defined来定义，用require来加载
2. CMD规范：一个模块一个文件，按需加载
3. CommonJs规范：服务器模块的规范，node采用的这个规范，每个模块都有一个单独的作用域，模块内部的变量无法被外部读取，除非定义为global的对象和属性 js模块化开发写法
4. 原始模式

```
function fn(){

}
```

```
function fn1 (){

}
//缺点：污染全局变量，无法保证变量不与其他的冲突，且模块成员之间看不出直接关系
```

2. 对象模式

```
let module = {
    var1 : 1 ,
    var2 : 2 ,
    fn1:function(){

    },
    fn2:function(){

    }
}
//优点：直接通过module.fn1()调用对象的属性和方法，避免了全局污染，模块直接也有了关系，
//缺点：外部可以随意修改内部成员，module.var1 = 100
```

3. 立即执行函数模式

```
let module = (function(){
    var var1 = 1
    var var2 = 2
    function fn1(){

    }
    function fn2(){

    }
    return {
        fn1:fn1,
        fn2:fn2
    }
})()
```

4. 放大模式:如果一个模块很大，必须引入几部分，或者一个模块需要继承另一个模块，上面的代码为module模块添加了一个新方法m3(),然后返回新的module模块;

```
let module = (function(mod){
    mod.md1 = function(){

    }
    return mod
})(module1)
```

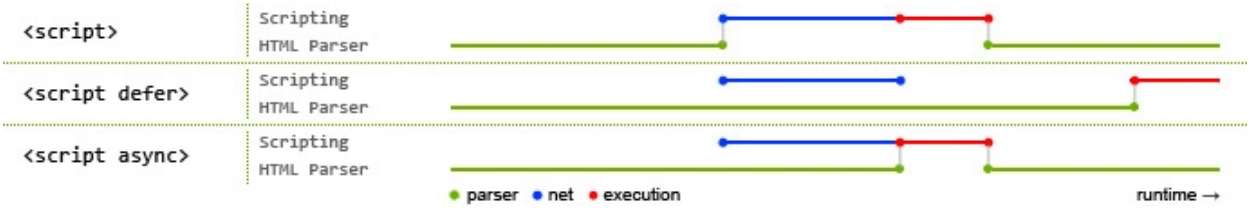
5. 宽放大模式：在浏览器环境中，模块的各个部分通常都是从网上获取的，有时无法知道哪个部分会先加载。如果采用放大模式的写法，第一个执行的部分有可能加载一个不存在空对象，这时就要采用"宽放大模式"。就是立即执行函数，因为立即执行函数的参数可以是空{}

```
var module = (function(mod){
   // ...
   return mod;
 })(window.module || {});
```

- **异步加载js的方法**

1.defer:<script>标签的defer="defer"属性，仅ie能用，需等到dom文档全部解析完成才会被执行，执行途中不会影响页面构造，这个属性是告诉浏览器立即下载，但延迟执行，HTML5规定按照他们的先后顺序执行，因此第一个延迟脚本会先于第二个脚本执行，但在现实生活中，延迟脚本并不一定会顺序执行，所以最好只有一个延迟脚本

2.async：<script>标签的async="async"属性，异步加载，加载完即执行，适用于外部脚本文件，但标记这个属性的执行脚本并不能保证其的执行顺序，因此保证两个脚本互不依赖



> 蓝色线代表网络读取，红色线代表执行时间，这俩都是针对脚本的；绿色线代表 HTML 解析。也就是说async是乱序的，而defer是顺序执行，这也就决定了async比较适用于百度分析或者谷歌分析这类不依赖其他脚本的库。从图中可以看到一个普通的<script>标签的加载和解析都是同步的，会阻塞DOM的渲染，这也就是我们经常会把<script>写在<body>底部的原因之一，为了防止加载资源而导致的长时间的白屏，另一个原因是js可能会进行DOM操作，所以要在DOM全部渲染完后再执行。

3.动态创建script标签

```
function asyncLoader(url){
    var script = document.createElement('script')
    script.type = 'text/javascript'
    if(script.readystate){ //兼容ie
        script.onreadystatechange = function(){
            if(script.readyState == 'loaded' || script.readyState == 'complete'){
                script.onreadystatechange = null
                console.log(111);

            }
        }
    }else{
        script.onload = function(e){
            //兼容火狐  谷歌等浏览器
            console.log(222);

        }
    }
    script.src = url
    document.body.appendChild(script)
```

```
}
asyncLoader('../JS/jquery.js')
```

- **常见的web安全及防护原理**

1. xss 定义：跨站脚本攻击，指通过存在安全漏洞的web网站，注册用户浏览器内运行非法的HTML标签和JavaScript进行的一种攻击

原理：攻击者往web网站嵌入恶意可执行的网页脚本代码，当用户浏览该页面时，会执行这一段脚本代码，从而获取用户信息或者侵犯用户安全隐私的目的

防御：

- csp 本质：建立白名单，开发者明确的告诉浏览器哪些外部资源是可以加载的

开启csp:

> 设置http的请求头中的Content-Security-Policy

> 或者给HTML页面的meta标签加Content-Security-Policy

```
//只允许加载本站网站
<meta http-equiv='Content-Security-Policy' content='default-src 'self''></meta>
//只允许加载HTTPS资源的图片
<meta http-equiv='Content-Security-Policy' content='img-src https:'></meta>
//允许加载任何来源框架
<meta http-equiv='Content-Security-Policy' content='child-src 'none''></meta>
```

- 转义字符

```
function escape(str) {
  str = str.replace(/&/g, '&amp;')
  str = str.replace(/</g, '&lt;')
  str = str.replace(/>/g, '&gt;')
  str = str.replace(/"/g, '&quto;')
  str = str.replace(/'/g, '&#39;')
  str = str.replace(/`/g, '&#96;')
  str = str.replace(/\//g, '&#x2F;')
  return str
}
```

```
//node里面使用
const xss = require('xss')
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html)

//html里面使用
 <script src="https://rawgit.com/leizongmin/js-xss/master/dist/xss.js"></script>
```

```
    var html = filterXSS('<h1 id="title">XSS Demo</h1><script>alert("xss");</scr' +
'ipt>');
    console.log(html)

//以上示例使用了 js-xss 来实现，可以看到在输出中保留了 h1 标签且过滤了 script 标签。
```

2. csrf 定义：跨站请求伪造，利用用户已经登录的状态，在用户不知情的情况下，以用户的名义 完成非法的操作

完成CSRF攻击的三个条件：

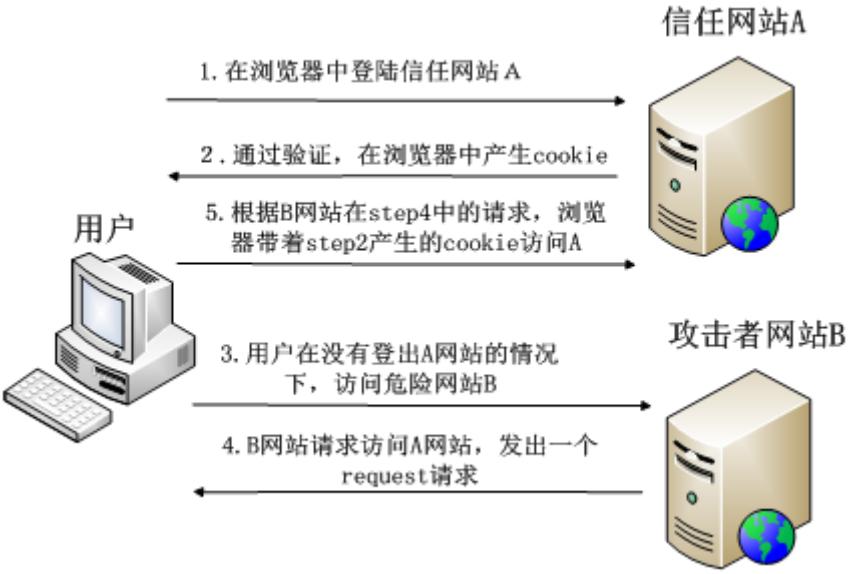
> （1）用户登录A网址，并记录了cookie

> （2）在名义登出A网址的时候，访问了恶意攻击者提供的引诱危险站点 B (B 站点要求访问站点 A)。

> （3）A网站没有做CSRF防御

我们来看一个例子： 当我们登入转账页面后，突然眼前一亮惊现"XXX隐私照片，不看后悔一辈子"的链接，耐不住内心躁动，立马点击了该危险的网站（页面代码如下图所示），但当这页面一加载，便会执行submitForm这个方法来提交转账请求，从而将10块转给黑客。

原理：如下图



防御：

- Get 请求不对数据进行修改
- 不让第三方网站访问到用户 Cookie
- 阻止第三方网站请求接口
- 请求时附带验证信息，比如验证码或者 Token

3. sql注入 定义：SQL注入是一种常见的Web安全漏洞，攻击者利用这个漏洞，可以访问或修改数据，或者利用潜在的数据库漏洞进行攻击。

原理：


```
<!--//前端代码-->
<form action="/login" method="POST">
    <p>Username: <input type="text" name="username" /></p>
    <p>Password: <input type="password" name="password" /></p>
    <p><input type="submit" value="登陆" /></p>
</form>

<!--//后端代码-->
let querySQL = `
```

```
    SELECT *
    FROM user
    WHERE username='${username}'
    AND psw='${password}'
`;

<!--这是我们经常见到的登录页面，但如果有一个恶意攻击者输入的用户名是 admin' --，密码随意输入，就可以直接登入系统了。why! ----这就是SQL注入-->

<!--我们之前预想的SQL 语句是:-->
SELECT * FROM user WHERE username='admin' AND psw='password'
<!--//但是恶意攻击者用奇怪用户名将你的 SQL 语句变成了如下形式: -->
SELECT * FROM user WHERE username='admin' --' AND psw='xxxx'
<!--在 SQL 中,' --是闭合和注释的意思，-- 是注释后面的内容的意思，所以查询语句就变成了: -->
SELECT * FROM user WHERE username='admin'
```

- **设计模式**

- **为什么要同源策略**

同源策略本质是一种约定，web的行为就是构建在这种约定上面的，就像人类的行为必须建立来法律的基础上一样，同源策略的目的就是限制不同源的dom或者脚本之间的相互访问，以免造成干扰和混乱

- **offsetWidth/offsetHeight,clientWidth/clientHeight与scrollWidth/scrollHeight的区别**

1. offsetWidth / offsetHeight = content + padding*2 + border
2. clientWidth/clientHeight = content + padding * 2
3. scrollWidth/scrollHeight = content + padding * 2 + 溢出内容的尺寸,无溢出，则和client相等

- **JavaScript中有哪些方法定义对象**

1. 字面量表示法

```
let obj = {
    name:'luoli'
}
```

2. 构造函数

```
function demo(){

}
let obj = new demo()
```

3. Object.create()

```
let obj = Object.create({a:1,b:2})
log(obj)  //{}--> obj._proto_{a:1,b:2}
```

- # 谈谈对promise的理解

一、promise是什么?

最早由社区提出和实现的一种解决异步编程的方案,es6写进语言标准，统一了写法，原生提供了promise对象，es6规定，promise是一个构造函数，用来生成promise实例

二、 promise因为什么产生的?

为了解决异步函数回调金字塔问题而产生

三、promise的两个特点

1. promise的状态不受外界改变，promise的状态只有异步操作的时候才能决定是哪种状态，其他任何操作都不能改变这个状态

- pending 初始状态
- fulfilled 成功状态
- reject 失败状态 2.promise 的状态一旦不改，就不会再变，任何时候都可以得到这个结果，状态不可逆，只能从pending到fulfilled或者pending到reject

四、promise的三个缺点

1. promise一旦新建就会立即执行，无法中途取消
2. 如果不设置回调函数，promise将会抛出错误，，不会反应到外部
3. 当处于pending状态时，无法得知进展到哪一步，是刚刚开始还是即将完成

五、promise方法

1. 链式写法

```
//<!--第一个promise成功或者失败没有返回东西，第二个promise输出undefined-->
function demo(){
    var promise = new Promise((resolve , reject) => {
        resolve('成功')  //reject('失败')
    })
    return promise
}

demo().then(success => {
    log(success) //成功
},error => {
    log(error)
```

```
}).then( data => {
    log(data) // undefined
})

//<!--d第一个promise成功返回 success + 100 ，第二个promise输出200 ，第一个promise失败返回
 'error' ， 第二个输出error-->
function demo(){
    var promise = new Promise((resolve , reject) => {
        resolve(100)  //reject('失败')
    })
    return promise
}

demo().then(success => {
    return success + 100
},error => {
    return 'error'
}).then( data => {
    log(data) // 200
})
```

2. promise.catch()

原理：只传失败的回调

```
function demo(){
    var promise = new Promise((resolve , reject) => {
        resolve('成功')  //reject('失败')
    })
    return promise
}

demo().then(success => {
    log(success) //成功
},error => {
    log(error)
})
//<!--等同于-->
demo().then(success => {
    log(success) //成功
}).catch(error=>{

})
```

3. promise.all()

Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。

```
    var flag = true
    let p1 = new Promise((resolve, reject) => {
        if (flag) resolve('成功11')
        else reject('失败11')
```

```
    })
    let p2 = new Promise((resolve, reject) => {
        if (flag) resolve('成功22')
        else reject('失败22')
    })
    let p3 = new Promise((resolve, reject) => {

        if (flag) resolve('成功33')
        else reject('失败33')
    })
    let p4 = new Promise((resolve, reject) => {
        if (flag) resolve('成功44')
        else reject('失败44')
    })
     Promise.all([p1, p2, p3, p4]).then((result) => {
            console.log(result, '111aaa');  <!--//["成功11", "成功22", "成功33", "成功44"]
"111aaa"-->
        }).catch((error) => {
            console.log(error, '222aaa');  <!--//失败11 222aaa-->

        })
```

- **web端会话跟踪的方法有哪些?**

会话跟踪：主要解决http无状态的问题，当用户发出请求时，服务器会给回应，客户端与服务器端的联系是离散的、非连续的，当用户在一个网站的多个页面之间转换时，根本无法确定是否是同一个用户，会话跟踪可以解决这个问题，当用户在多个页面之间切换时，服务器会保存该用户的信息。

1. cookie
2. url重写
3. session（SSL会话{Secure Socket Layer））
4. 隐藏的表单域

```
HTML 表单中可以含有如下的条目：
<input type="hidden" name="session" value="a1234">
 <!--这个条目的意思是：在提交表单时，要将指定的名称和值自动包括在 GET 或 POST数据中。这个隐藏域
可以用来存储有关会话的信息，但它的主要缺点是：仅当每个页面都是由表单提交而动态生成时，才能使用这种
方法。单击常规的超文本链接并不产生表单提交，因此隐藏的表单域不能支持通常的会话跟踪，只能用于一系列
特定的操作中，比如在线商店的结账过程。-->
```

- **js的内置对象**

1. 数据封装类对象：String，Boolean，Number，Array，和Object;
2. 其他对象：Function，Arguments，Math，Date，RegExp，Error

- **JavaScript的基本规范**

1. 不要在同一行声明多个变量
2. 不要用 === 或者 !== 来比较true / false 或者数值
3. for和if要使用大括号
4. 不要使用全局函数
5. 单行注释 // 多行注释 /**/
6. 函数不应该有时候有返回值有时候没有返回值
7. for..in中的变量，应该var来定义其的作用域，避免作用域污染
8. 使用字面量代替new Array()这种形式

• eval()的作用

把字符串参数解析成js代码运行，并返回执行结果

```
var jsonstr ='{a:1,b:2}'
log(eval('2+4')) //6
log(eval('varage=10')) //var age = 10 输出10
console.log(eval('('+jsonstr+')')); //{a: 1, b: 2}
```

• null和undefined的区别

区别：null表示’无‘的对象，转换数值为0，undefined表示’无‘的原始值，转为数组为NAN，即Number(null) = 0,Number(undefined) = NAN，typeof两个都是false，所以两个==

1. null表示尚未存在的对象，常用来表示函数企图返回一个不存在的对象

```
console.log(Object.getPrototypeOf(Object.prototype)); //null
```

2. undefined表示'缺少值'，典型用法

- 声明一个变量，但是没有赋予值
- 调用函数时，应提供的参数没有提供，
- 对象没有赋值属性，该属性的值为undefined
- 函数没有返回值时，默认返回为undefined

```
var a ;
log(a) //undefined

function demo(x){
    log(x) //undefined
}
demo()

let obj = new Object()
log(obj.p) //undefined
```

```
function fun () {

}
log(fun()) // undefined
```

- **严格模式的规则**

1. 不能只声明变量不赋值
2. 函数的参数不能有同名属性，不然报错
3. 禁止this指向全局对象
4. 不能使用‘eval’ 和 ‘arguments’作为变量名
5. 不能使用转义字符

- 优点

> 消除JavaScript语法不严谨，不合理之处，减少怪异活动；提高编译器效率，增加代码的运行速度

- **JavaScript的函数节流和函数防抖的区别和原理**

概念：两个都是优化js代码的一种手段，函数节流是指在一定的时间内js代码只执行一次，就像人眨眼一样，在一定的时间只眨一次，函数防抖指的是频繁触发的情况下，只有足够的空闲时间js代码才会执行一次，就比如坐公交的时候，一定时间内，有人频繁刷卡，司机就不会开车，等到没有人刷卡了，司机才会开车

适用场景：

- 函数节流

1. 页面元素滚动，触发某个事件
2. 高频率点击表单提交按钮，表单重复提交

```
/*
函数节流
**/
// 声明一个变量当标记，记录当前代码是否在运行
var flag = true
var dom = document.getElementById('box').onscroll = function(){
    if(!flag){
        return
    }
    flag = flase
    setTimeout(function(){
        log('函数节流')
        flag = true
    },300)
}
```

- 函数防抖

1. 表单验证，用户输入完，开始验证
2. 轮播图

```
/*
函数防抖
**/
// 巧用setTimeout做缓存池，可以轻易的清除执行的代码
var timer = true
var dom = document.getElementById('box').onscroll = function(){
   clearTimeout(timer)
   timer =  setTimeout(function(){
        log('函数抖动')
    },300)
}
```

- ## 原始类型有哪几种？null是对象吗？

原始类型存储的都是值，是没有函数可以调用的，比如undefined.toString()，输出报错

1. null
2. string
3. undefined
4. boolean
5. symbol
6. number

- ## typeof()和instanceof()的用法区别

1. typeof()： 是一个一元运算符，放在一个运算数之前，运算数可以是任意数，返回的是一个字符串，表示数据类型，返回的数据类型有（number，string，undefined，object，boolean，function）
2. instanceof： 实例，表示A instanceof B（A是B的实例？ ）

- ## prototype 和 proto 区别是什么?

> 区别：prototype是每个函数都会有的属性，只有函数才有的属性，他是一个指针，指向一个对象，而_proto_是除了null以外的对象都支持的属性，_proto_用来相连对象与该对象的原型的介质。

- ## 谈谈对JS执行上下文栈和作用域链的理解

- 执行上下文

定义：当前JavaScript代码解析和执行时所在的环境的抽象概念，JavaScript中运行的任何代码都是在执行上下文中，

- 执行上下文的类型

- 全局执行上下文

最基础、默认的上下文，不在任何函数的代码都位于全局执行上下文，做了两件事（1.创建了一个全局对象，浏览器中是Windows；2.将this指向这个全局对象，一个程序只能有一个全局执行上下文）

- 函数执行上下文

每次函数调用时，都会产生一个新的函数上下文

- eval函数执行上下文

- 执行上下文的生命周期（创建-执行-回收阶段）

1. 创建阶段

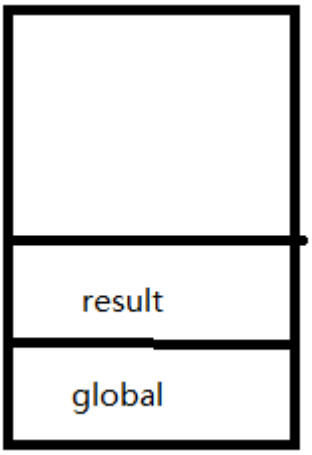
> 当函数被执行时，但未调用其内部代码之前，有做以下三件事
>
> 1. 创建变量对象：初始化函数的参数arguments，提升变量和函数
> 2. 创建作用域链：作用域相当于一个独立的盘，让变量之间互相不干扰，当前作用域没有定义的变量，这成为了自由变量，自由变量会一直向上寻找，要到创建这个函数的作用域去寻找，如果没有找到就变成undefined，这一过程就形成了一个作用域链
> 3. 确定this的指向

2. 执行阶段：变量、代码的执行
3. 回收阶段：变量和函数执行完成，执行上下文出栈等待虚拟机回收执行上下文

- 执行上下文栈

执行栈：也叫调用栈，具有LIFO（后进先出）的特点，用于存储代码执行过程中创建的所有执行上下文

规则：首次运行JavaScript代码时，会创建一个全局上下文push到执行栈的底部，当函数调用时，会创建一个新的函数上下文push到执行栈的栈顶，当栈顶的函数运行完成，其对应的函数上下文会从栈顶pop出，当浏览器关闭时，全局上下文pop出



执行上下文总结：

1. 执行上下文是单线程的
2. 全局上下文只有一个，浏览器关闭时全局上下文出栈
3. 函数执行上下文没有个数限制
4. 每个函数调用时，会有新的执行上下文为其创建，即便是调用自身也是一样的

- 作用域

定义：作用域就是变量和函数的可访问范围，控制这个变量或者函数可访问行和生命周期。

- 作用域链

我们知道函数在执行时是有个执行栈，在函数执行的时候会创建执行环境，也就是执行上下文，在上下文中有个大对象，保存执行环境定义的变量和函数，在使用变量的时候，就会访问这个大对象，这个对象会随着函数的调用而创建，函数执行结束出栈而销毁，那么这些大对象组成一个链，就是作用域链。那么函数内部未定义的变量，就会顺着作用域链向上查找，一直找到同名的属性。

- ## prototype 和 proto 区别是什么?

1.js红万物皆对象，方法（Function）是对象。function。prototype是对象，所以有对象具有的特点，都有_proto_属性，可称为隐式原型，一个对象的隐式原型指向构造该对象的构造函数的原型，保证实例能够访问构造函数原型中定义的属性和方法

2. 方法（Function）除了有_proto_属性外，还有自带的prototype（原型属性），这是函数独有的属性，这个属性是一个指针，指向一个对象，这个对象包含所有实例共享的属性和方法，这个对象包含一个constructor属性，指向原构造函数

- ## 取数组的最大值（ES5、ES6）

```
//es5
Math.max.apply(null,[1,4,67,666])
//es5
Math.max(...[1,4,67,666])
```

```
[1,4,67,666].reduce((total,currentvalue)=>{
    return total = total > currentvalue ? total : currentvalue
})
```

- **如何判断img加载完成**

```
<html>
<!--在这里插入内容-->
<body>
    <img
src="https://ss3.bdstatic.com/70cFv8Sh_Q1YnxGkpoWK1HF6hhy/it/u=2141623099,2896788564&fm=26&gp=0.jpg" alt="">
     <img
src="https://ss1.bdstatic.com/70cFuXSh_Q1YnxGkpoWK1HF6hhy/it/u=2996011778,1200189855&fm=26&gp=0.jpg" alt="">
    <img
src="https://ss1.bdstatic.com/70cFuXSh_Q1YnxGkpoWK1HF6hhy/it/u=345047253,1474150922&fm=26&gp=0.jpg" alt="">
    <div id="text">loading</div>
    <script>
        var img1 = document.querySelectorAll('img')
        var text = document.getElementById('text')
        //方法一
        for(let i = 0 ; i < img1.length ; i ++){
            img1[i].onload = function(){
                text.innerHTML = 'endding'
            }
        }
        // 方法二
        function loaderImg(img , callback){
            var timer = setInterval(function(){
                if(img.complete){
                    callback()
                    clearInterval(timer)
                }
            },50)
        }
        for(let i = 0 ; i < img1.length ; i ++){
            loaderImg(img1[i],function(){
                 text.innerHTML = 'endding'
            })
        }
    </script>
</body>
</html>
```

- **如何阻止冒泡和默认事件?**

1. 阻止冒泡

```
<!--弹框关闭会跳转到百度-->
 <form id="form1" runat="server">
       <div id="divOne" onclick="alert('我是最外层');">
              <div id="divTwo" onclick="alert('我是中间层! ')">
                     <a id="hr_three" href="http://www.baidu.com"
mce_href="http://www.baidu.com"
            onclick="alert('我是最里层! ')">点击我</a>
               </div>
        </div>
    </form>

    <script>
        $('#hr_three').click(function (event) {
            <!--return false  不建议用，会阻止冒泡和默认事件-->
            <!--兼容模式-->
            if (event && event.stopPropagation) {
                event.stopPropagation()
            } else {
                window.event.cancelBubble = true
            }
        })
```

2. 取消默认事件

```
<!--弹框关闭不会跳转到百度，会有冒泡-->
<form id="form1" runat="server">
       <div id="divOne" onclick="alert('我是最外层');">
              <div id="divTwo" onclick="alert('我是中间层! ')">
                     <a id="hr_three" href="http://www.baidu.com"
mce_href="http://www.baidu.com"
            onclick="alert('我是最里层! ')">点击我</a>
               </div>
        </div>
    </form>

    <script>
        $('#hr_three').click(function (event) {
            <!--return false  不建议用，会阻止冒泡和默认事件-->
            <!--兼容模式-->
           if (event && event.preventDefault) {
                event.preventDefault()
            } else {
                window.event.returnValue = false
            }
        })
```

- **实现拖拽的方法**

1. 使用原生js实现拖拽

```
<div id="drag"></div>

var dragId = document.getElementById('drag')
<!--鼠标按下事件，获取鼠标距离元素左边框的距离，是元素上的事件-->
window.onload = function(){
    dragId.onmounsedown = function(event){
        <!--获取鼠标距离元素左边框和上边框的距离-->
        var event = event || window.event
        var diffx = event.clientX - dragId.offsetLeft
        var diffy = event.clientY - dragId.offsetTop
        <!--鼠标移动事件，是全局区域-->
        document.onmounsemove = function(moveEvent){
            <!--获取元素距离左边和上边的距离，
            左边距离 = 鼠标到左边的距离 - 鼠标到元素左边框的距离
            -->
            var movex = moveEvent.clientX - diffx
            var movey = moveEvent.clientY - diffy
            <!--判断元素是否超出边框-->
            if(movex < 0 ){
                movex = 0
            }else if(movex > window.innerWidth - dragId.offsetWidth){
                movex = window.innerWidth - dragId.offsetWidth
            }

            if(movey < 0 ){
                movey = 0
            }else if(movey > window.innerHeight - dragId.offsetHeight){
                movex = window.innerHeight - dragId.offsetHeight
            }
            dragId.style.left = movex + 'px'
            dragId.style.top = movey + 'px'
            <!--鼠标抬起，情况移动和抬起事件-->
            document.onmounseup = function(){
                this.onmounsemove = null
                this.onmounseu = null
            }
        }

    }
}
```

2. 使用HTML5元素拖拽drag和拖放drop

```
<div id="drag" draggable = 'true'>
    <p data-id='00'>111</p>
    <p data-id='00'>222</p>
    <p data-id='00'>333</p>
    <p data-id='00'>444</p>
    <p data-id='00'>555</p>
</div>
<div id="part2" class="part2"></div>
```

```
var startx , starty
var dragId = document.getElementById('drag')
<!--元素开始拖拽-->
dragId.ondragstart = function(event){
<!--获取元素开始拖拽时鼠标距离左边的距离-->
    startx = event.clientX
    starty = event.clienty
    var pList = document.querySelectorAll('p')
    let text =  Array.from(pList).map(item=>{
        return item.innerHTML
        })
        e.dataTransfer.setData('text',text )
    });
    log('元素开始拖拽')
}
<!--元素拖拽中-->
dragId.drag = function(){
    log('元素拖拽中')
}
dragId.ondragend = function(event){
<!--元素拖拽结束时，鼠标距离左边的距离（event.clientX）-->
    var movex =parseFloat(dragId.style.left) + event.clientX - startx
    var movey = parseFloat(dragId.style.top) + event.clientY - starty
    if(movex < 0){
        movex = 0
    }else if(movex > window.innerWidth - part1.offsetWidth){
        movex = window.innerWidth - part1.offsetWidth
    }

    if(movey < 0){
        movey = 0
    }else if(movey > window.innerHeight - part1.offsetHeight){
        movey = window.innerHeight - part1.offsetHeight
    }

dragId.setAttribute('style',"left:"+parseFloat(movex)+'px;'+"top:"+parseFloat(movey)+'px'
)
    log('元素拖拽结束')
}
<!--dataTransfer用于交换数据-->
    var part2 = document.getElementById('part2');

    part2.addEventListener('dragenter', function(){
        // console.log('part2 dragenter');
    });

    part2.addEventListener('dragover', function(event){
        // console.log('part2 dragover');
        event.preventDefault()

    });

    part2.addEventListener('dragleave', function(){
        // console.log('part2 dragleave');
    });
```

```
    part2.addEventListener('drop', function(e){

        var data = e.dataTransfer.getData('text')
        console.log(data.split(','),'data');

        // debugger;
        // console.log('part2  drop');
    });
```

- ## $(document).ready()方法和window.onload有什么区别?

$(document).ready()是dom树绘制完成就执行，不必等到图片或者其他外部文件都加载完毕就执行，而window.onload必须等到页面所有元素都加载完毕，而且只能执行一次

```
$(document).ready(function(){
    console.log(11);
})
$(document).ready(function(){
    console.log(22);
})
window.onload = function(){
    console.log(33);
}
window.onload = function(){
    console.log(44);
}
//输出 11   22    44
```

- ## jquery中.get()提交和$.post()提交有区别吗?

1. 请求方式不同： get是通过get来异步请求，而post()是通过post的方法来异步请求
2. 传递的参数方式不同：get的参数是放在url后面，而post的参数是通过http实体内容发送给服务器，这种传递方式对用户不可见
3. 数据传输大小不同：get请求最大是2KB，而post要大的多
4. 安全问题：get请求的数据会被浏览器存储去了，所以post安全性比get高

- ## 对前端路由的理解？前后端路由的区别?

- 路由：根据不同的url展示不同的页面和内容
- 前端路由：最重要的一点是不刷新，前端路由就是把不同路由对应的页面和应用交给前端来做，每次跳转到不同的url都是用前端的喵点路由，随着单页面（SPA）的普及，前端后端分离，现在普遍使用前端路由（大部分页面结构不变，只改变部分内容时使用）

> - 优点:
>
> 1. 用户体验感好，不需要每次从服务器返回给用户，快速
> 2. 可以在浏览器输入想要访问的url
> 3. 实现前后端分离，现在普遍框架都支持前后端分离，方便开发

> - 缺点：
>
> 1. 使用浏览器前进或者后退的时候会重新发起请求，没有利用好缓存

- 后端路由：在浏览器的地址栏切换url时，会向服务器发起请求，服务器响应请求，再把拼接好的HTML文件返回给前端显示，所以意味着会再次刷新浏览器

> - 优点：分担前端压力，HTML和数据的拼接都是由后端完成
> - 缺点：当项目十分庞大时，加大服务器压力，而且网速过慢时，页面可能会白屏，对用户体验感并不好

• 手写一个类的继承

1. 构造函数实现继承

```
function Parent(){
    this.name = 'parent'
}
Parent.prototype.say = function(){
}
function Child(){
    this.namec = 'child'
    Parent.call(this)//改变函数上下文，利用call改变Child函数的this指向，增加name属性值
}
log(new Child()) //{name:"parent",namec:"child"}
log(new Child().say()) //undefined
```

> 缺点：这种继承并不会改变child函数原型链，所以parent和child的原型链是互相没有关系的

2. 原型链继承

```
function Parent(){
    this.name = 'parent'
    this.array = [1,2,3]
}
Parent.prototype.say = function(){
}
function Child(){
    this.namec = 'child'
}
Child.prototype = new Parent()
let c1 = new Child()
let c2 = new Child()
c1.array.push(4)
log(c1.prototype === c2.prototype) // true
log(c1.__proto__ === c2.__proto__) // true
log(c1.array , c2.array) // ,[1,2,3,4]  [1,2,3,4]
```

> 缺点：当改变c1的实例对象，c2的实例对象也跟着变，因为c1.__proto === c2.**proto**

3. 构造函数和原型链方式组合继承

```
function Parent(){
    this.name = 'parent'
    this.array = [1,2,3]
}
Parent.prototype.say = function(){
}
function Child(){
    Parent.call(this)
    this.namec = 'child'
}
Child.prototype = new Parent()
let c1 = new Child()
let c2 = new Child()
c1.array.push(4)
log(c1.array , c2.array) // [1,2,3,4]   [1,2,3]
```

> 缺点：父类的构造方法使用了2次，分别是Parent.call(this)和Child.prototype = new Parent()
> 4.优化

```
<!--优化1-->
<!--缺点：child的实例的构造函数都来自Parent-->
function Parent(){
    this.name = 'parent'
    this.array = [1,2,3]
}
Parent.prototype.say = function(){
}
function Child(){
    Parent.call(this)
    this.namec = 'child'
}
Child.prototype = Parent.prototype
let c1 = new Child()
let c2 = new Child()
c1.array.push(4)
log(c1.array , c2.array) // [1,2,3,4]   [1,2,3]
log(c1.constructor) // Parent

<!--优化2-->
function Parent(){
    this.name = 'parent'
    this.array = [1,2,3]
}
Parent.prototype.say = function(){
}
function Child(){
    Parent.call(this)
    this.namec = 'child'
}
<!--Objec.create是在原型（__proto__）上面创建对象，-->
Child.prototype = Object.create(Parent.prototype)
```

```
Child.prototype.constructor = Child
log(c1.constructor) // Child
```

- **XMLHttpRequest：XMLHttpRequest.readyState;状态码的意思**

返回值	说明
0	未初始化。表示对象已经建立，但是尚未初始化，尚未调用 open() 方法
1	初始化。表示对象已经建立，尚未调用 send() 方法
2	发送数据。表示 send() 方法已经调用，但是当前的状态及 HTTP 头未知
3	数据传送中。已经接收部分数据，因为响应及 HTTP 头不安全，这时通过 responseBody 和 responseText 获取部分数据会出现错误
4	完成。数据接收完毕，此时可以通过 responseBody 和 responseText 获取完整的响应数据

```
window.onload = function () {  //页面初始化
    var b = document.getElementsByTagName("input")[0];
    b.onclick = function () {
        var url = "server.php";  //设置请求的地址
        var xhr = createXHR();  //实例化XMLHttpRequest对象
        xhr.open("POST", url, true);  //建立间接，要求异步响应
        xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');  //设
置为表单方式提交
        xhr.onreadystatechange = function () {  //绑定响应状态事件监听函数
            if (xhr.readyState == 4) {  //监听readyState状态
                if (xhr.status == 200 || xhr.status == 0) {  //监听HTTP状态码
                    console.log(xhr.responseText);  //接收数据
                }
            }
        }
        xhr.send("callback=functionName");  //发送请求
    }
}
```