ES6知识点

1.模板字符串

可以换行定义,可以解析变量,空格回车会输出出来

以下为模板字符串:

```
var a = "张三";
        var b = `张三`;
        var age = 18
        var c = "我叫"+a+"今年多大"+age+"岁"
        var c = `我叫${a}今年多大${age}岁`
        console.log(a,b)
```

2.let 和 const

let 声明变量注意不能被定义过,变量不能被重复定义

使用let关键字声明的变量没有提升,使用var关键字声明的变量,有变量提升

块级作用域, 比如if后的{} switch的{} 都能生成块级作用域,作用域外部不能在访问这个变量了

const 一但被定义不能重新赋值,数组可以通过下标操作、或者用一些数组函数操作比如push(),

对象也可以修改但是不要使用赋值的形式去修改,用obj.name='张三'这样行

const关键字声明打的变量要有初始值

const 在{}里面也能生成块级作用域

声明的变量必须要赋值

3.解构赋值

1.可以同时给多个变量赋值

2.不使用第三个变量的前提下也能交换两个变量的值

3.函数可以使用解构赋值的方式返回值同时可以给多个变量赋值

4.函数传参 function aa({name="王五",age}){} aa({age:"18",name:"张三"}) 可以不用考虑顺序问题了

4.箭头函数

箭头函数的5种形式

```
//1. 没有返回值的,没参数的

        function a (){

             console.log(111)

        }

        var a = () => console.log(111);

        a()


//2. 没有返回值的,有参数的
         function a (name){
              console.log(name)
          }
         var a = (name) => console.log(name+1);
         a(18)


//3. 有参数的,有返回值的,
         function a (name){
              return name+2
          }

    //箭头函数 里面的代码体如果是单行的情况下 可以省略 return 关键字,同时也可以省略代码体的{}
     var a = (name) => name+2;
        var b = a(18)
        console.log(b)
```

```
// 4.有参数的,有返回值的,多行代码体的
          function a(name){
              name +1;
              name *2;
              return name
          }

          var a = (name) =>{
                  name +=1;
                  name *=2;
                  return name
              }

              console.log(a(9))


//5. 如果返回值是一个对象  单行的情况下需要加()
          var a =  () => ({name:"张三"})
          var a =  () => {name:"张三"}  //undefined
          console.log(a())
```

this指向问题

一、在全局环境中

在全局执行环境中（在任何函数体外部），this都是指向全局对象。在浏览器中，window对象即是全局对象：

```
console.log(this); //Window
var a = 1;
console.log(window.a);  //1
this.b = 3;
console.log(b); // 3
console.log(window.b) //3
```

二.单独的this，指向的是window这个对象

```
alert(this); // this -> window
```

三.函数调用的时候，前面加上new关键字

所谓构造函数，就是通过这个函数生成一个新对象，这时，this就指向这个对象

```
function demo() {
 //alert(this); // this -> object
 this.testStr = 'this is a test';
}
let a = new demo();
alert(a.testStr); // 'this is a test'
```

四.用call与apply的方式调用函数

call()与apply()都是在特定的作用域中调用函数，等于设置函数体内this对象的值，以扩充函数赖以运行的作用域。

一般来说，this总是指向调用某个方法的对象，但是使用call()和apply()方法时，就会改变this的指向。

五.定时器中的this，指向的是window

六.元素绑定事件，事件触发后，执行的函数中的this，指向的是当前元素

七.函数调用时如果绑定了bind，那么函数中的this指向了bind中绑定的元素

八.对象中的方法，该方法被哪个对象调用了，那么方法中的this就指向该对象

```
let name = 'finget'
let obj = {
 name: 'FinGet',
 getName: function() {
 alert(this.name);
 }
}
obj.getName(); // FinGet
let fn = obj.getName;
fn(); //finget this -> window
```

call 与 apply 的不同点：两者传入的列表形式不一样

- call可以传入多个参数；
- apply只能传入两个参数，所以其第二个参数往往是作为数组形式传入

它们各自的定义：

apply：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.apply(A, arguments);即A对象应用B对象的方法。

call：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.call(A, args1,args2);即A对象调用B对象的方法。

它们的共同之处：

都“可以用来代替另一个对象调用一个方法，将一个函数的对象上下文从初始的上下文改变为由thisObj指定的新对象”。

五.闭包函数

特点:

总结下有三个特点：

- 函数嵌套函数
- 内部的函数可以引用外部函数的参数或者变量
- 参数和变量不会被垃圾回收机制回收，因为内部函数还在引用

好处:

①保护变量安全 ，实现封装，防止变量声明冲突和全局的污染
②在内存中维持一个变量，可以做缓存（但使用多了同时也是一项缺点，消耗内存）
③匿名自执行函数可以减少内存消耗

缺点

①被引用的私有变量不能被销毁，增大了内存消耗，造成内存泄漏，解决方法是可以在使用完变量后手动为它赋值为null；
②其次由于闭包涉及跨域访问，所以会导致性能损失，我们可以通过把跨作用域变量存储在局部变量中，然后直接访问局部变量，来减轻对执行速度的影响

六.作用域和生命周期

作用域 : 变量在哪个范围区间内 可以使用

生命周期 : 变量从创建后, 什么时候被销毁掉

局部变量

函数内部定义的变量, 只能在函数内部使用, 所以称作**局部变量**. 一般情况,函数执行结束,局部变量会被**销毁.**

```
<script>
        function test(){
        let a = 100
    }

    test()
    alert(a) // Uncaught ReferenceError: a is not defined
             // 在函数外不能访问 函数内部定义的变量

</script>
```

全局变量

所有函数外部定义的变量, 即可以在函数内部使用,也可以在函数外部使用, 所以称为**全局变量**

```
<script>
        let a = 100
    test()
    function test(){
        alert(a)  // 100  函数内部 也可以访问 全局变量
    }

</script>


<script>
        let a = 100
    test()
    function test(){
        let a = 77
        alert(a)  // 77 先在当前作用域找,没有,再尝试到更大的作用域中查找
    }

</script>
```

块级作用域

函数内部是一个范围, 函数外部又是另一个范围,

在代码中单纯的一对大括号 **{ }** 也会被看成特殊的范围, 叫做 块级作用域.

块级作用域中用 **let** 和 **const** 声明定义的变量和常量, 会在块级作用域外无效.

```
{
        let str = '凡6真帅'
}
alert(str)  // 报错. str 变量未定义
```

生命周期

局部变量可以理解为, 在函数中定义,在函数中使用,函数结束时,自动销毁

全局变量从定义位置开始, 一直到整个代码结束, 才会销毁

因为局部变量占用内存空间的时间短, 所以**尽量使用局部变量**

七.js执行顺序

1.js 编辑阶段

先把 函数 变量 加载到内存中

开内存,生成作用域(作用域不会因为你的函数调用位置而发生改变)

this执行

2.阶段 执行上下文 一个页面中可以有无数个执行上下文环境,比如一个函数你定义了没有去调用,那

这个函数内部的执行上下文环境就不会被执行

赋值

"use strict" //加上这个就是严格模式

八.新增数组方法

1.indexof()

```
var a = arr.indexOf('王五');
```

检测数组或字符串中是否包含某个值,如果包含返回第一次查找到那个值所对应的下标,如果查不到返回-1

2.①forEach()遍历数组

```
//遍历数组
        arr.forEach(function(value,index,arr){
            // 参数1 数组里面对应的每个值
            // 参数2 数组里面的索引
            // 参数3 原数组
            // console.log(value)
            // console.log(index)
            console.log(arr)
            console.log(arr[index])
        })
```

②for循环遍历

```
for(let i=0; i<arr.length; i++){
           console.log(arr[i]+"索引是"+i)
       }
```

③while遍历

```
let i =0;
        while(i<arr.length){
            console.log(arr[i]+"索引是"+i)
            i++
        }
```

④do while遍历 (不建议使用)

```
        let i=0
        do{
            console.log(arr[i]+"索引是"+i)
            i++
        }while(i<arr.length)
```

⑤for in 遍历(只能遍历对象)

```
for(let i in arr){
            console.log(i)
            console.log(arr[i])
         }
```

⑥for of遍历(对象,数组都能遍历)

```
for(let i of arr){
            console.log(i)
        }
```

3.Array.from(伪数组)将伪数组转为数组

4.find(回调函数) 根据查找指定值,符合条件的,会返回出来这个值,返回第一值之后就会停止查找

5.filter(回调函数)会把所有符合条件的值放到一个数组中返回出来

6.map(回调函数) 映射 把数组里面的每一个元素按照你传入回调函数里面的代码都执行一遍

7.some(回调函数)有一个满足条件的就会返回true,都不满足会返回false,如果第一个就满足,有短路效果,就会停止循环不会在继续搜索

8.every(回调函数) 必须每一个元素都符合条件才会返回true,同样有短路效果

9.assign(object,object) //合并对象

九.深拷贝与浅拷贝

```
//下面这种为深拷贝
        var a = 123;
        var b = a;
        console.log(a) //123
        console.log(b) //123
        a=456
        console.log(b) //123
```

数组和对象直接等号赋值的通常是浅拷贝 一个发生变化,另外一个也会跟着变,他只是指向另外一个地址

```
var obj1 = {'name':"张三"}
var obj2 = obj1
console.log(obj1) //张三
console.log(obj2) //张三
obj1.name="李四"
console.log(obj2) //李四

var arr1 = [11,22,33]
var arr2 = arr1
arr1[0]='aaa'
console.log(arr2)
        // var arr2 = ['aaa',22,33]
```

深拷贝 把里面的内容一个一个的复制出来,开一个新内存放进去,内存地址不一样,互不影响

```
var arr1 = [11,22,33]
var arr2=[]
for(let i in arr1){
    arr2[i] = arr1[i]
}
arr1[0]='abc';
console.log(arr2[0]) //11 不受第一个数组影响
```

十.zepto

类似于jQuery的一种框架,跟jQuery使用差不多,他的内存小,所以性能要比jQuery好.

十一.ES6新语法...

... 深拷贝

```
// ... 深拷贝
          var obj = {name:"张三"}
          var obj2 = {...obj}

           obj2.name='李四'
           console.log(obj)
           console.log(obj2)


//...不会拷贝原型里面的内容
         function aa(name){
             this.name = name
             aa.prototype.age=18
             aa.prototype.sex='男'
         }

         function bb(name){
             this.name=name
         }

         var a = new  aa('张三')

         var b = {...a}

         b.name='666'
         console.log(a)
         console.log(a.age)

         console.log(b)

         // bb.prototype = {...aa.prototype}
         b.__proto__={...aa.prototype}
         b.age=99
         console.log(b.age)
         console.log(a.age)
```

十二.JS事件绑定（addEventListener）和普通事件（onclick）有什么区别

普通事件（onclick）

普通事件就是直接触发事件，同一时间只能指向唯一对象，所以会被覆盖掉

```
var btn = document.getElementById("btn");
btn.onclick = function(){
        alert("你好111");
}
btn.onclick = function(){
        alert("你好222");
}
```

输出的结果只会有<**你好222**>，一个click处理器在同一时间只能指向唯一的对象。所以就算一个对象绑定了多次，其结果只会出现最后的一次绑定的对象。

事件绑定（addEventListener）

事件绑定就是对于一个可以绑定的事件对象，进行多次绑定事件都能运行。

```
var btn = document.getElementById("btn");
btn.addEventListener("click",function(){
        alert("你好111");
},false);
btn.addEventListener("click",function(){
        alert("你好222");
},false);
```

运行结果会依次弹出**你好111，你好222**的弹出框。

- 友情提示（了解即可）
  - ie9 以前：使用
    attachEvent/detachEvent
    进行绑定
    - 进行事件类型传参需要带上 on 前缀
    - 这种方式只支持事件冒泡，不支持事件捕获
  - ie9 开始：使用**addEventListener**进行绑定

二者区别

addEventListener对任何DOM都是有效的，而onclick仅限于HTML

addEventListener可以控制listener的触发阶段，（捕获/冒泡）。对于多个相同的事件处理器，不会重复触发，不需要手动使用removeEventListener清除。

总的来说：事件绑定是指把事件注册到具体的元素之上，普通事件指的是可以用来注册的事件

十三.Promise

Promise 是抽象的异步处理对象

语法 它本身就是一个类,也可以说是构造函数

构造函数需要一个 函数作为参数. 这个函数叫做"执行器函数".

1)执行器函数在创建Promise对象时立即执行.

2)Promise对象有状态和值

a.pending undefined

b.resolved/fulfilled '哈哈' 表示成工

c.rejected '呜呜' 表示失败

细节:状态不可逆

let p1 = new Promise((resolve(成功),reject(失败))=>{})

Promise(里面的内容叫做执行器函数)

如果想控制它什么时候执行,可以在外层用一个函数包住它.(这样使代码更加灵活更加可控)

```
let p1 = new Promise((resolve,reject)=>{
                        // console.log('执行器函数会立即执行~')
                        setTimeout(()=>{
                                console.log('a')
                resolve('哈哈')
                                reject('呜呜')
                        },3000)
                })
```

- **then()**方法接收两个函数作为参数，

第一个参数是 Promise 执行成功时的回调，

第二个参数是 Promise 执行失败时的回调，两个函数只会有一个被调用

```
function p1 () {
                        return new Promise((resolve,reject)=>{
                                setTimeout(()=>{
                                        console.log('a')
                                        resolve()
                                },3000)
                        })
                }

function p2() {
                        return new Promise((resolve,reject)=>{
                                setTimeout(()=>{
                                        console.log('a')
                                        resolve()
                                },3000)
                        })
                }

function p3(){
                        return new Promise((resolve,reject)=>{
                                setTimeout(()=>{
                                        console.log('a')
                                        resolve()
                                },3000)
                        })
                }
p1().then(p2).then(p3)
```

细节:

```
p1().then(p2()) //这样p2会提前调用.  注意:这里的p2不能写括号
p1().then(p2)   //把p2这个函数当做参数传给then

//这里尽量不要在then()里面使用回调函数的方法
p1().then(()=>{p2()}).then(()=>{p3()})
相当于:p1().then(()=>{p2()};return undefined).then(()=>{p3()})
//这里会出现错误 p2() p3() 会同时执行
```

- **catch()**方法，它可以和 then 的第二个参数一样，用来指定 reject 的回调，另一个作用是，当执行 resolve 的回调（也就是上面 then 中的第一个参数）时，如果抛出异常了（代码出错了），那么也不会报错卡死 js，而是会进到这个 catch 方法中。
- **all()**方法，Promise 的 all 方法提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调, 会把所有异步操作的结果放进一个数组中传给 then。
- **race()**方法，race 按字面解释，就是赛跑的意思。race 的用法与 all 一样，只不过 all 是等所有异步操作都执行完毕后才执行 then 回调。而 race 的话只要有一个异步操作执行完毕，就立刻执行 then 回调。

- 注意：其它没有执行完毕的异步操作仍然会继续执行，而不是停止。

await 和 async(语法糖)

用了它就可以不使用then()方法,

注意:它们两个必须同时使用,如下:

```
async function foo(){
                        await p1()
                        await p2()
                        await p3()
}
foo()
```

十四.什么是作用域链?

1、简单说就是作用域集合 当前作用域 -> 父级作用域 -> ... -> 全局作用域 形成的作用域链条

全局作用域的变量和方法都可以进行调用

局部的变量和方法只能局部进行调用(除闭包外)

局部可以访问全局的变量和方法

十五.什么是原型链?

当访问一个对象的某个属性时，会先在这个对象本身属性上查找，如果没有找到，则会去它的 `__proto__` 隐式原型上查找，即它的构造函数的 `prototype` ，如果还没有找到就会再在构造函数的 `prototype` 的 `__proto__` 中查找，这样一层一层向上查找就会形成一个链式结构，我们称为 `原型链` 。

十六.Proxy代理

Proxy 也就是代理，可以帮助我们完成很多事情，例如对数据的处理，对构造函数的处理，对数据的验证，说白了，就是在我们访问对象前添加了一层拦截，可以过滤很多操作，而这些过滤，由你来定义。

语法

```
 let p = new Proxy(target, handler);
           //set 可以做过滤
           // 参数1 原对象 var obj = {'name':"zhangsan",age:18,img:'1.png'};
           // 参数2 调用的属性名 age
           // 参数3 你传入的属性值 19
//get()
//属性读取操作的捕捉器。
```

参数

1. `target` ：需要使用 `Proxy` 包装的目标对象（可以是任何类型的对象，包括原生数组，函数，甚至另一个代理）。
2. `handler`：一个对象，其属性是当执行一个操作时定义代理的行为的函数(可以理解为某种触发器)。具体的 `handler` 相关函数请查阅官网

十七.reduce()

```
var arr = [1,2,3,4];
        //reduce    用来计算数组里面值的和
        //参数1 回调函数   (a,b)   a是数组里面第一个值     b是后面的那些值
        //参数2   初始值
var a = arr.reduce(function(a,b){

            return a+b
        },10)
        console.log(a) //20
```

十八.ES6 Symbol

ES6 引入了一种新的原始数据类型 Symbol ，表示独一无二的值，最大的用法是用来定义对象的唯一属性名。

ES6 数据类型除了 Number 、 String 、 Boolean 、 Objec t、 null 和 undefined ， 还新增了 Symbol 。

用法

由于每一个 Symbol 的值都是不相等的，所以 Symbol 作为对象的属性名，可以保证属性不重名。

十九.Ajax的几个步骤

第一步：创建 XMLHttpRequest对象

var ajax = new XMLHttpRequest()

第二步：规定请求的类型、URL 以及是否异步处理请求

ajax.open('GET',url,true)

第三步：发送信息至服务器时内容编码类型

ajax.setRequestHeader("Content-type","application/x-www-form-urlencoded")

第四步：发送请求

ajax.send(null);

第五步：接受服务器响应数据

ajax.onreadystatechange = function () {if (obj.readyState == 4 && (obj.status == 200 || obj.status == 304)){}

二十.set()和map()的区别

ES6 新增数据类型 set集合,里面的值不能有重复的

set不能通过索引直接获取里面的值，a.next().value 遍历set使用 for of 或者 forEach

set 有key也有value 但是它的键值对相同 a a

map 映射

new Map(存放二维数组) 有去重效果，相同key的值会被后面的替换掉

map 是由键值对组成的 键和值可以不相同 而 set 键值对相同

map和set不同点

1 map 是由键值对组成的 键和值可以不相同 而 set 键值对相同

2 set使用 add() map 使用 set() 做添加

3 new的时候 set([]) map([],[],[])

二十二.post和get的区别

GET和POST的区别，何时使用POST?

1.**GET：一般用于信息获取，使用URL传递参数，对所发送信息的数量也有限制，一般在2000个字符**

2.**GET： 传送的数据量较小，不能大于2KB**。post 传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。 用IIS过滤器的只接受get参数，**所以一般大型搜索引擎都是用get方式**

3.**GET:是从服务器上获取数据，post 是向服务器传送数据。 get 请求返回 request – URI 所指出的任意信息。**

4.GET:是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTP post机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址，用户看不到这个过程。

5.POST：一般用于修改服务器上的资源，对所发送的信息没有限制。

6.GET方式需要使用Request.QueryString来取得变量的值，而POST方式通过Request.Form来获取变量的值，也就是说Get是通过地址栏来传值，而Post是通过提交表单来传值。

然而，在以下情况中，请使用 POST 请求：

无法使用缓存文件（更新服务器上的文件或数据库）

向服务器发送大量数据（POST 没有数据量限制）

发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠