javascript笔记总结

一.Dom

- **Dom（Document Object Model）文档对象模型**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
   <style>
            .wrapper{
                  font-size: 20px;
                  color: #01a1ff;
                  border: 1px solid #000000;
             }
    </style>
</head>
<body>
        <h1>hello world</h1>
        <h2 id="a01">标题2</h2>
        <h3>五花肉</h3>
        <button onclick="addNode()">增加节点</button>
        <button class="replace-btn">替换节点</button>
        <button>删除节点</button>
</body>
</html>
```

1.1 添加Dom元素

- **方法一**

```
function addNode(){
    let a = document.createElement("a"); //创建元素节点
    let arr = document.createAttribute("class"); //创建属性节点
    arr.value ="wrapper"; //为属性设值
    let text = document.createTextNode("跳转"); //创建文本节点
    a.appendChild(text);//将文本节点追加在元素节点上
    a.setAttributeNode(arr)//在元素节点上添加属性节点
    doucment.body.appendChild(a);//将新增元素节点添加页面中
}
```

```
如果获取body?
方式1：document.body（写法新颖）
方式2：let body =document.querySelector('body')；限定于文档中只有一个body标签
```

- **方法二(简便)**

```
 function addNode () {
          let a = document.createElement("a");
              a.setAttribute("class","wrapper"); //为元素添加属性节点及属性值
              a.innerHTML = "跳转";
              document.body.appendChild(a);
      }
```

1.2 替换Dom元素

```
 document.querySelector('.replace-btn').addEventListener('click', function () {
      let h1 = document.getElementsByTagName('h1')[0];
      let new_h1 = document.createElement('h1');//<h1></h1>
      let new_h1_text = document.createTextNode('你好 世界');
      new_h1.appendChild(new_h1_text) ;//<h1>你好世界</h1>
      document.body.replaceChild(new_h1, h1);//(新节点,老节点)
  })
```

1.3 删除Dom元素

```
  document.querySelector('.replace-btn
').addEventListener('click', function () {
      let btn_arr = document.getElementsByTagName('button')
[2];
      btn_arr.addEventListener('click', function () {
          let h2 = document.getElementById('a01');
          document.body.removeChild(h2);
      })
  })
```

1.4 插入Dom元素

```
在现有元素之前插入一个新元素  父元素.insertBefore(新元素,目标元
素);
```

```
  let p = document.createElement("p");
  p.innerHTML = "我是暖心";
  document.body.appendChild(p);
  let New1 = document.getElementsByTagName("h1")[0];
  document.body.insertBefore(p,New1);
```

1.5 检测是否有子节点

```
    // hasChildNodes() 如果这个方法返回true，说明该节点有一个或者
多个子节点
     <button id="hasNode">校验是否有子节点</button>
```

```
    const hasNode = document.getElementById("hasNode");
    hasNode.addEventListener('click', () => {
        console.log("app hasChildNodes",app.hasChildNodes());
//true
        const app2 = document.getElementById("app2");
        console.log("app2
hasChildNodes",app2.hasChildNodes());//false
    });

```

1.6 Document 类型

- **文档写入**
  - **document.write**

```
  if(Math.random()*10 >5){
                document.write("<h1>新增的内容</h1>");
        }else {
                document.write("<h1>new Text</h1>");
        }
            document.write("<strong>" + (new Date()).toString() +"</strong>");

//增添onload事件,等页面全部加载完,在执行里面的内容,此时输出内容将会重写整个页面
 window.onload = () =>{
                if(Math.random()*10 >5){
                    document.write("<h1>新增的内容</h1>");
                }else {
                    document.write("<h1>new Text</h1>");
                }
            }
```

  - innerHtml && innerText

```
<div class="con">
      我是文本
      <p>我是段落</p>
      <a href = "https://baidu.com"></a>
  </div>
  <button id="btn1">修改app的内容</button>
  <button id="btn2">修改p内容的内容</button>
  <button id="btn3">修改p内容的innerText</button>
```

```
   const btn1 = document.querySelector("#btn1");
    btn1.addEventListener("click",()=>{
        const con = document.getElementsByClassName("con")[0];
        con.innerHTML = `<h1>标题1</h1>
                    <h2>标题·2</h2>`;
    })
    const btn2 = document.querySelector("#btn2");
    btn2.addEventListener("click",()=>{
```

```
        const p = document.querySelector("p");
        p.innerHTML = `<h1>标题1</h1>
                    <h2>标题·2</h2>`;  //识别html标签，再输出
    })
    const btn3 = document.querySelector("#btn3");
    btn3.addEventListener("click",()⇒{
        const p = document.querySelector("p");
        p.innerText = `<h1>标题1</h1>
                    <h2>标题·2</h2>`;  //会原样输出，不识别标签
    })
```

- **定位练习**

```
document.anchors 包含文档中所有带 name 属性的<a>元素。
document.forms 包含文档中所有<form>元素
document.images 包含文档中所有<img>元素
document.links 包含文档中所有带 href 属性
```

- **取得属性 && 设置属性**

```
    <div id="app" class="a" style="background-color:#50baff">
        你好,暖心
    </div>
    <button onclick="changeColor()">改变颜色</button>
    <button onclick="changeColor2()">改变颜色2</button>
    <button onclick="create_Attribute()">创建属性</button>
<style>
       .a{
            width:100px;
            height: 100px;
        }
        .b{
            width:100px;
            height: 50px;
            background-color:pink;
        }
</style>
```

```
const app = document.getElementById("app");
    console.log("getAttribute id",app.getAttribute("id"));
    console.log("getAttribute class",app.getAttribute("class"));
    console.log("getAttribute style",app.getAttribute("style"));

    function changeColor () {
        app.setAttribute("style","background-color:purple");
    }

    function changeColor2 () {
        app.setAttribute("class","b");
    }

    function create_Attribute () {
        const attr = document.createAttribute("align");
        attr.value ="center";
        app.setAttributeNode(attr);
    }
```

- **dataset**

```
<div id="app" data-person="暖心" data-score="100"></div>
```

```
    const app = document.querySelector("#app");
    //dataset属性是DOMStringMap的一个实例
    console.log(app.dataset);//DOMStringMap {person: '暖心', score: '100'}
    console.log(app.dataset.person); // 暖心
    console.log(app.dataset.score);  //100
```

1.7 dom小练习

- **要求**

```
点击增加按钮，会有序的增加，列表1,2,3,4依次增加。
根据文本框输入的行数，定向删除所在的行。
```

```
    <button>增加</button>
    <br>
    <input placeholder="要删除第几行"><button onclick="del()">删除</button>
    <br>
    <ul>
        <li>小类1</li>
        <li>小类2</li>
        <li>小类3</li>
    </ul>
```

```
<script>
 const btn1 = document.getElementsByTagName("button")[0];
 btn1.addEventListener('click',()=>{
 const ul = document.getElementsByTagName('ul')[0];
 const li = document.createElement("li");
 //查找ul下面的标签li
 const nextNode = Array.from(ul.childNodes).filter(item => item.nodeType === Node.ELEMENT_NODE );
 const li_text = document.createTextNode("小类" + (nextNode.length +1));
 // const li_text = document.createTextNode("小类"+(ul.children.length+1));
        li.appendChild(li_text);
        ul.appendChild(li);
    })

 const del = function () {
     const inpVal = document.getElementsByTagName("input")[0].value;
     const ul = document.getElementsByTagName("ul")[0];
     // index  rowNum-1
     // ul.children[index]
     const delLi = ul.children[inpVal-1];
     ul.removeChild(delLi);
 }
</script>
```

1.8 打字机效果案例

```
<!DOCTYPE html>
<html lang = "en">
<head>
    <meta charset = "UTF-8">
    <title>打字机效果</title>
    <style>
        #con{
            font-weight:bold;
        }
        .keyword{
              color:#7f0055 ;
            font-weight:bolder ;
        }
        .key{
            color: lightskyblue;
            font-weight: bold;
        }
        .keys{
            color: mediumpurple;
        }
    </style>
</head>
<body>
   <div id="con">
        <span class="key">/* 注释 */</span><br>
        <span class="keyword">const</span> str ="hello world.<span class="keyword">substring</span>(8);<br>
       <span class="keyword">console</span>.<span class="key">log</span>("str",str);<br>
       <span class="keyword">function</span> ( <span class="keys">a</span> , <span class="keys">b</span> ) {<br>
       <span class="keyword">return</span> a+b ;<br>};<br>
       <span class="keyword">function</span>(1,5);
   </div>
</body>
</html>
```

```
<script>
    const con = document.getElementById("con");
    const clone = con.innerHTML;
    con.innerHTML = "";
    let step = 0 ;
```

```
    const timer = setInterval(()=>{
         const currentChar = clone.substr(step,1);
            if(currentChar === "<"){
                step = clone.indexOf(">",step)+1;
            }else{
                step++;
            }
            const newContext = clone.substring(0,step);
            con.innerHTML = newContext + (step%3 === 0 ? "_":"");
            if(step>clone.length){
                clearInterval(timer);
            }
    },100)
</script>
```

二.BOM

- **Bom (浏览器对象模型)**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>BOM</title>
</head>
<body>
    <button onclick="openNewWin()">弹出新窗口</button>
    <button id="closewin">关闭窗口</button>
    <button id="movewin">移动窗口</button>
    <button onclick="getinfo()">获取浏览器硬件</button>
    <button onclick="getpos()">获取定位信息</button>
    <button onclick="getposres()">获取定位结果</button>
</body>
</html>
```

2.1 window对象

- open.moveTo(100, 100); // 移动到新位置的绝对坐标
- open.moveBy(100, 100); // 相对当前位置的移动
- open.resizeTo(500, 500); // 接收新的宽度和高度值
- open.resizeBy(300,300); //接收宽度和高度各要缩放多少
- open.close(); // 关闭新窗口

```
let newwin=null;
    let pos;
    function openNewWin(){

newwin=window.open("about_blank","_blank","height=200px,width=300px,top=200px,left=300px");

document.getElementById("closewin").addEventListener('click',function (){
                newwin.close();
            })

document.getElementById("movewin").addEventListener("click",function (){
                newwin.moveTo(100,100);
            })
    }
```

2.2 navigator对象

- navigator.appName // 浏览器全名
- navigator.platform // 返回浏览器运行的系统平台
- navigator.deviceMemory // 返回单位为 GB 的设备内存容量
- navigator.onLine //是否联网，但不能判断是互联网还是局域网
- navigator.getBattery().then((b) => console.log(b)); // 查看电量

```
 //点击获取浏览器硬件 页面显示硬件信息
function getinfo(){
        //let div = document.createElement("div");
        // let text1 =document.createTextNode('浏览器全名:'+window.navigator.appName+',');
        // let text2 =document.createTextNode('系统平台:'+window.navigator.platform+',');
```

```
        // let text3 =document.createTextNode('内存容量:'+navigator.deviceMemory+",");
        // let text4 =document.createTextNode('是否联网:'+window.navigator.onLine+",");
        let template =`
        ${window.navigator.appName} ;
        ${window.navigator.platform} ;
        ${navigator.deviceMemory} ;
        ${window.navigator.onLine} ;
        `
        /*div.appendChild(text1);
        div.appendChild(text2);
        div.appendChild(text3);
        div.appendChild(text4);*/
        // document.body.appendChild(div);

document.body.appendChild(document.createTextNode(template));
        navigator.getBattery().then(function(res){
            console.log(res);
            let res_template =`
            ${res.charging} ;
            ${res.chargingTime} ;
            ${res.dischargingTime} ;
            ${res.level} ;
            `
            let t =document.createTextNode(res_template);
            document.body.appendChild(t);
        })
```

2.3 location 对象

```
    let pos = navigator.geolocation.getCurrentPosition((position)⇒{return pos});
   console.log(pos.timestamp); //时间戳
   console.log(pos.coords);   //坐标
   console.log(pos.coords.latitude,pos.coords.longitude); //纬度  经度
```

```
//点击获取获取位置
function  getpos(){
```

```
  navigator.geolocation.getCurrentPosition(function(res){
    console.log(res);
    pos=res;
  })
}
//点击获取位置信息
function getposres(){
  let a =document.createElement("a");
  let node1 =document.createTextNode("时间
戳"+pos.timestamp+",");
  let node2=document.createTextNode("坐标"+pos.coords+",");
  let node3=document.createTextNode("维
度"+pos.coords.latitude+",");
  let node4=document.createTextNode("经
度"+pos.coords.longitude+",");
  a.appendChild(node1);
  a.appendChild(node2);
  a.appendChild(node3);
  a.appendChild(node4);
  document.body.appendChild(a);
}
```

2.4 history对象

- history.go(-1); // 后退一页
- history.go(1); // 前进一页
- history.go(2); // 前进两页
- history.back(); // 后退一页
- history.forward(); // 前进一页

三.语言基础

3.1 语法

- **区分大小写**

```
let a = 'hello';
let A =  '你好';
console.log(a,A);
console.log(typeof A); //返回A的数据类型
```

- **声明规则**

```
let _xxv ='2000'; // 下划线开头的变量一般代表私有变量
```

- **注释**

```
    //    *单行注释*
    /*
          多行注释
    */
```

快捷键： 单行注释： `ctrl` + `/` ； 多行注释 ： `ctrl` + `shift` + `/` ；

- **严格模式**

```
"use strict";  // 严格模式
ss =2000;
console.log(ss);
const obj_1 = {
    name: '暖心',
    name: '暖心',
}
console.log(obj_1);  //报错 严格模式不允许相同的name属性和不声明变量
```

- **语句**

```
 let aa =10;
 let bb = 20;
 let sum = aa + bb;
 console.log("sum",sum); //30
```

- **逻辑/条件 语句**
  - **if 语句**

```
表达式的返回值 true/false
            js 会强制转型
                数值 to 布尔类: 当数值≠0   ⇒ true
                                当数值 == 0  ⇒ false
                对象 to 布尔类: 当对象有值时  ⟹ true
                                当对象null/undefined ⟹ false
```

```
      let flag = -1;
         if(flag){
             console.log("进入条件语句");
         }

        let ooo = new Object(); // 空对象
        let oo; //undefined
        let oooo = null; // 空值
         if(ooo){
             console.log("对象类型测试,进入条件语句");
         }

         if(oooo){
             console.log("对象类型测试,进入条件语句");
         }
```

- **if ...else**

```

         function sub(a,b){
             if(a-b≥0){
                 console.log("结果是正数或零");
             }
             else{
                 console.log("负数");
             }
         }

         //
```

- **if ... else if ... else**

```
function sub_new(a,b){
            if(a-b>0){
                console.log("+");
            }else if(a-b ==0){
                console.log("0");
            }else{
                console.log("-");
            }
        }
```

- 赋值类型

```
let num =10;
num = 9;
num ="ten";
console.log(num);

// 简便/省事的赋值
var person_name ="zhangsan",
      age=22,
      gender ="man";
  console.log(person_name,age,gender);
```

- **作用域**

```
        var OUT="out";
         if(1===1){
             console.log(OUT);
             var IN_2 ="if_in";
         }
         console.log(IN_2);

         // js里 只有function里面才有作用域概念
         function area(){
             let IN_3 ="area_in";
             console.log(OUT); //out
         }
                 area();
         console.log(IN_3);   //报错
```

```
    //代码块
```

```
    {
        console.log("代码块:",global);
        var block ="形同虚设的代码块"
    }

    console.log("外部",block);

    function area(){
        console.log("函数中",global);
        var area_1 = "局部";
        console.log("函数中:",area_1); // 函数中:局部
    }
    area();
    // console.log("函数外部:" ,area_1); //报错
```

3.2 变量

- **var**
  - var 声明变量为全局变量;

```
var a = 1000; 等价于 ⇒  window.a =1000;  ⇒ a = 1000;
var b = "暖心";
var c = true ;

//var 声明提升
   function promotion(){
       console.log(a);
       var a = 2 ; //undefined
   }
   promotion();

   //等价于 →
   /*  function promotion(){
       var a;
       console.log(a);
       a= 2;
   }
     promotion(); */
```

- **let**
  - **ES5**: var （声明全局作用域）

    ES6：let const（声明块作用域）

```
    {
        let aa = 100;
        console.log("let: 代码块",aa);
    }
    // console.log("let: 代码块",aa); // 报错 没有定义

    if(2>1){
        let cc="if的内部变量";
        console.log(cc);
    }
    // console.log(cc);  // 报错 not defined

    /*
        var  :作用域提升
        ler  :暂时性死区(没有作用域提升)
    */

    function nopromotion(){
        let a =1000;
        console.log(a);
        // var a =1000; // undefined
        // let a =1000; // error not get "a" before initail
    }
    nopromotion();

    for(let j=0; j<2; j++){
        console.log("j:",j);  // j:0  j:1
    }
    console.log("j bottom",j); //  报错  j is not defined 未定义
```

- **const**
  - const声明变量时必须初始化，且不能修改变量的值；
  - const 声明的限制只适用于它指向变量的引用；
  - 如果const变量引用是一个对象，修改这个对象内部属性是并不违反const限制；

```
    let a; // undefined
    a=1;
    console.log(1000);

    //const 必须初始化(地址)
    const b=100;
    console.log(b);
    // 一旦初始化,就不能更改(地址)
    const c = {
```

```
        name:"张三",
        origin:"china"
    }
    console.log(c);   //{name:"张三", origin:"china"}

    c.name = "李四";
    console.log(c);  //{name:"李四", origin:"china"}
```

- **易错点**
  - **for循环中let的声明**

```
for(var i = 0; i<5; i++){

}
console.log(i); //5

var改成let;
for(let i =0 ;i<5; i++){

}
console.log(i); //ReferenceError: i 没有定义

  **为什么推荐使用let**
for(var i = 0; i <3; i++){
    setTimeout(function(){    //  定时器是异步执行的
        console.log(i);
    },500);
}       // 3、3、3

var 改成let之后
 for (let i = 0; i < 3; i++) {
         setTimeout(function () {
             console.log(i);
         },500)
     }  //  0、1、2

```

  - **结论**：let 比var更容易控制；
  - **let的缺点**： 1. 造成内存的浪费 (过度使用)
    2. 会造成内存泄漏

3.3 定时器

```
1. setTimeout(function(){},sleeping);
⇒setTimeout(函数名,sleeping);
隔多少sleeping毫秒之后,再运行第一个参数(运行匿名函数);
a.  一般用"中途"结束
b.  一般用classTimeout模拟异步

2. setInterval(function(){},sleeping);
 每隔多少sleeping毫秒之后,运行第一个参数();
a.可以"中途"结束它
b.为什么推荐它的?
性能不高,只能通过调整sleeping优化性能,当sleeping没法统一标准
解决方法:当你需要"轮询"操作时,推荐是
RAF(requestAnimationFrame).
```

- **setTimeout**

```
let timer01 = null;
    timer01 = setTimeout(function (){
        console.log("hello,world");
    },100);

    function stop01(){
        clearTimeout(timer01);    // 暂停计时器
    };

   //模拟异步
    console.log("start hello world");
    console.log("start hello world");
    console.log("start hello world");
    console.log("start hello world");
    setTimeout(function () {
        console.log("我来执行异步了");
    },2000);

```

- **setInterval**

```
    let step = 0; //全局变量计步器
    let timer02 = setInterval(function () {
        if(step>5){
            clearInterval(timer02);
            return
        console.log("又到1s了",step);
        step += 1;
        }
     },1000)

     function stop02 () {
         clearInterval(timer02);
     }
```

3.4 数据类型

- 简单类型 : Undefind 、Null 、Number、 Boolean、String 、Symbol（符号）。
- 引用类型 ： Object

```
     const a = true;
     const b = 3.14;
     const c = "hello world";
     const d = {
         name : "hello world"
     }
     const e = null ;
     let f ;
    // typeof 运算符 检测变量的数据类型
     console.log(typeof a); // boolean
     console.log(typeof b); // number
     console.log(typeof c); // string
     console.log(typeof d); // object
     console.log(typeof e); // null(对象的占位符) ⇒ object;
     console.log(typeof f); // undefined
```

- 实际开发遇到的问题* ：

  好像没有办法判断一个对象到底是null还是undefined;

- 解决方案 ：

  String(null) // null ; String(undefined) // undefined

```
let abc ;
console.log(String(abc));    // "undefined"
console.log(String(abc) == "undefined");  // true

var abc = null ;
console.log(String(abc));   // "null"
console.log(String(abc) == "null");   // true
```

- **NaN（*not a number*）**
  - NaN 非常特殊, 它是唯一一个自己不等于自己的值
  - 凡是有NaN参与的运算,返回值都是NaN

```
let aa = 0/0;  //NaN
let bb = "hello world" - 3;  //NaN
console.log(aa,bb);  // NaN NaN

console.log(aa == bb); //false
console.log(aa !== bb);  // true
console.log(NaN !== NaN); //true

console.log("aa +3:" ,aa+3);  //NaN
console.log("bb*10:" ,bb*10); //NaN
```

- **数值转型**
  - parseInt 将字符串转型成整数 => parseInt(string,radix);
  - parseFloat 将字符串转型成浮点数 => parseFloat (string);

```
let str ="3";
console.log(str,typeof str);
let num = parseInt(str); //不传第二个参数,它自动解析
console.log(num,typeof num);

//第二个参数为进制数(radix)
let str2 = "10";
console.log(parseInt(str2,2));  //2
console.log(parseInt(str2,8));  //8
console.log(parseInt(str2,10)); //10
console.log(parseInt(str2,16)); //16

console.log("不传第二个参数:",parseInt("010"));  //10
console.log("第二个参数按8进制:",parseInt("010",8)); //8
```

- **值的范围**
  - Number.Mix_VALUE 和 Number.MAX_VALU

```
    console.log(0.0000001); // 1e-7
    let res= console.log(Number.MAX_VALUE+Number.MAX_VALUE);
//Infinity
    console.log(isFinite(res)); //false   //isFinite 确定一个值是不是有限大
```

- **模板字面量 （两个反引号）**

```
    let hw ="hello\n world";
    console.log(hw);
    let hw_1 =`hello
     world`;
    console.log(hw_1);
    //通过${}插入变量
    let aaa =3,bbb=5;
    let sum_express =`$(aaa)+$(bbb)=${aaa+bbb}`;
    console.log(sum_express);

```

- **String**
  - **substr**: (字符串开始的索引位置，返回字符串的数量)
    - 当第一个参数为负值，处理成： 字符串长度加上负值；
    - 当第二参数为负值， 处理成 ： 直接将参数为 0

```
let str = "hello world";
           012345678910
console.log("str",str.substr(3))   //"lo world"
console.log("str",str.substr(3,7)) // "lo worl"
console.log("str",str.sbstr(3,-3)) //"" ⇒ substr(0,3)
```

  - **substring** `[` 字符串开始的索引位置，提取结束的位置 `)`
    - 当参数为负值，处理成： ： 不管是第一个还是第二个 ， 都直接将参数为 0

```
let str = "hello world";
           012345678910
console.log("str",str.substring(3))   //"lo world"
console.log("str",str.substring(3,7)) // "lo w"
console.log("str",str.substring(3,-3)) // "hel" ⇒
substring(0,3)
```

- **slice** [字符串开始的索引位置，提取结束的位置)
  - 当参数为负值, 处理成 不管是第一个还是第二个， 字符串长度加上负值，两个进行从小到大排序所得的范围;

```
let str = "hello world";   // length = 10
           012345678910
console.log("str",str.slice(3))   //"lo world"
console.log("str",str.slice(3,7)) // "lo worl"
console.log("str",str.slice(3,-3)) //"lo w" ⇒ slice(3,7)
console.lo("str",str.slice(-3,-9))  // "ello w"
⇒slice(1,7)
```

- **0bject**

```
//三种创建对象方式
const person00 = new Object();
    person00.name = "wangwu";
    person00.say = function () {
        return "i was wangwu"
    }
    console.log(person00.name);
    console.log(person00.say());

const person01 = {
        name : "zhangsan",
        origin : "china",
        say:function () {
            return "hello"
        },
        run:function () {
            return "i am running"
        }
    }
```

```
20
21      console.log(person01.name);
22      console.log(person01.say());
23      console.log(person01.run());
24
25  function Person () {
26          this.name = "zhaosi",
27          this.orgin = "usa" ,
28          this.say = function () {
29              return "i am zhaosi"
30          }
31      }
32
33      const person02 = new Person();
34
35      console.log(person02.name);
36      console.log(person02.orgin);
37      console.log(person02.say());
38
39   小总结 :
40      平时声明简单对象的时候,你可以使用{},Object就可以了
41      如果声明一些特殊的对象,或者需要定制化的对象时,
42      那么你需要定义自己的对象.
43      如何定义自己的对象
44      function 起一个合法的名称首字母大写(){} ;
```

- 关于对象一些属性的信息和配置

```
1      const obj ={
2          naame:"nuanxin",
3          age:22,
4          score:100
5      }
6
7      //获得对象属性描述信息
8      const val =
   Object.getOwnPropertyDescriptor(obj,"name");
9      console.log(val);
10     //控制台打印信息
11     configurable: true  // 属性是否可以删除
12     enumerable: true   //属性是否可以枚举（遍历）
13     value: "nuanxin"   //属性值
14     writable: true    //属性是否可以修改
15
16     Object.defineProperty(obj,"gender",{
```

```
        configurable:true,  //可以删除
        enumerable:false, // 不可枚举
        value:"男",
        writable:true   //可以修改属性值
    }) //给对象增加性别属性
    console.log(obj)
//{name:"nuanxin",age:"22",score:"100",gender:"男"}

  // delete obj.gender;  删除性别属性
  //
console.log(obj);//{name:"nuanxin",age:"22",score:"100"}
  for(let i in obj) // 性别属性已经设置不可枚举
  console.log(i); // name age score
  obj.gender = "女"; //性别由男修改成女
  console.log(obj); //{name: 'nuanxin', age: 22, score:
100, gender: '女'}

// 需求，动态能获取obj对象的age属性值和修改age属性值（根据
number值的变化）
  let number = 18;
  const 0bj = {
      name:"暖心",
      sex:男,
  }

   Object.defineProperty(obj,"age",{
        // 当有人读取obj的age属性时，get函数（getter）就会被
调用，且返回值就是age的值
        get(){
            return number;
        }
        // 当有人修改obj的age属性时，set函数（setter）就会被
调用，且会收到具体修改的值
        set(value){
           number = value
        }
    })
```

- ***constructor* (构造器)**

  构造器：定义首字母大写的函数;

```
// 如果这是构造器(首字母大写的函数),那么我们就new它
function Cat(){
    this.kind="家猫"
}
console.log(new Cat())
// 函数
// 如果是函数,我们直接加小括号调用就行
function cat(){
    return "i am cat"
}
console.log(cat())
```

- ***prototype*（原型）**
  - 怎么获取原型?

    > 1.对象通过 _ _proto _ _ 获取原型;
    >
    > 2.构造函数通过prototype属性获得
    >
    > 3.通过ES6中类的prototype属性

```
const c = new Cat();
```

  - new Cat() 创建了一个原型Prototype

```
prototype {
            health:true,
            constructor:function Cat(){
                            this.kind="家猫"
                        }
        }
```

  - 通过构造器区实例化 instance
  - 最后把 instance 赋值给 c

```
console.log("c: ",c)
console.log("原型prtotype:",Cat.prototype)
console.log(Cat.prototype.constructor == Cat)  //true
console.log(Cat.prototype.constructor === Cat) //true

const c01 = new Cat()
c01.name = "小白"
```

```
console.log("c01",c01)  // {kind:"家猫", name:"小白"}

const c02 = new Cat()
c02.name = "灰灰"
console.log("c02",c02)  // {kind:"家猫", name:"灰灰"}

const c03 = new Cat()
c03.name = "胖胖"
console.log("c03",c03)  // {kind:"家猫", name:"胖胖"}

Cat.prototype.health = true

console.log("c03 是否健康:",c03.health)  // c03 是否健康:
true
```

总结 ： **原型的作用,动态的增加或者修改属性或方法**

修改后的原型属性不仅对即将new的对象有影响,

同时对已经实例化的对象也有影响;

- **Object.defineProperty**（给实例增加属性）
  - **Object.definesProperty(** **实例** **,属性名字** **,** **配置项** **)**

```
  function Car () {}
   const car = new Car();
   car.wheel = "马牌";
   car.color = "red";
   Car.prototype.width= "1m";
   Object.defineProperty(car,"weight",{
       value:"2t"
})
```

- **hasOwnProperty**
  - 判断对象是否有某个特定的属性。必须用字符串指定该属性。

```
    console.log(car.hasOwnProperty("wheel")); //true
    console.log(car.hasOwnProperty("color")); //true
    console.log(car.hasOwnProperty("width")); //false
    console.log(car.hasOwnProperty("weight")); //true
```

- **isPrototypeOf**
  - 原型.isPrototypeOf(实例) 判断该实例是不是某个原型;

```
    console.log("实例car的原型是Car吗:",Car.prototype.isPrototypeOf(car)); //true
    console.log("实例c03的原型是Car吗:",Car.prototype.isPrototypeOf(c03)); //false
    console.log("实例c03的原型是Cat吗:",Cat.prototype.isPrototypeOf(c03)); //true

    //propertyIsEnumerable 判断给定的属性是否可以用 for..in 语句进行枚举,必须用字符串指定该属性
    console.log(car.propertyIsEnumerable("wheel")); //true
    console.log(car.propertyIsEnumerable("color")); //true
    console.log(car.propertyIsEnumerable("width")); //false

console.log(car.propertyIsEnumerable("weight"));//false
```

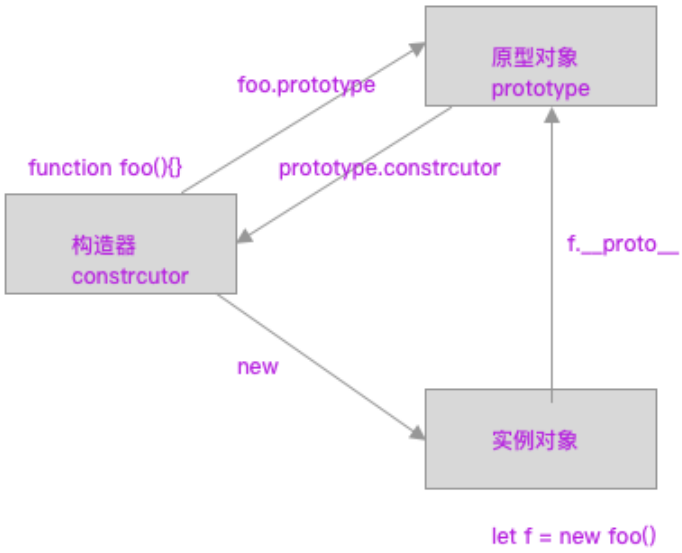
- **ToString()**
  - 返回对象的原始字符串表示。

```
    console.log("hello".toString()); //"hello"
    console.log("3,14".toString()); //"3.14"
    console.log(true.toString()); //"true"
    console.log(car.toString()); //[[object object]]
```

- **ValueOf()**
  - 返回最适合该对象的原始值。对于许多对象，该方法返回的值都与 ToString() 的返回值相同。

```
    console.log("hello".valueOf()); //"hello"
    console.log("3.14".valueOf());  //"3.14"
    console.log(false.valueOf());   //"false"
    console.log(car.valueOf());    // {wheel: '马牌', color: 'red', weight: '2t'}
```

原型总结图



3.5 操作符

- **(syntax sugar) 语法糖**
- **一元操作符**（自加++ 自减--）

```
// 1.一元操作符
let a = 1 ;
a = a+1 ;
console.log(a);  // 2

//++
a++;
console.log(a); // 3
++a;
console.log(a) // 4
console.log(a++);  // 4
console.log(++a)  // 6

//--
let b= 5;
b = b-1 ;
console.log(b)  // 4
```

```
    b--;
    console.log(b); // 3
    --b;
    console.log(b); // 2
    console.log(b--); // 2
    console.log(--b)  // 0
```

- **位运算**

```
概念 : 位数 一补数(反码) 二补数(补码)
      正数 : 原码 = 补码 = 反码
      负数 : -18 从+18二进制开始
      +18:原码:0000 0000 0000 0000 0000 0000 0001 0010
      反码:1111 1111 1111 1111 1111 1111 1110 1101
      补码:1111 1111 1111 1111 1111 1111 1110 1110
```

- **布尔操作符 (&& 、 || 、！)**

```
    const boolean_express = true;
    const num_express = 1<0;
    console.log(!boolean_express); //false
    console.log(!num_express);  //true

    console.log("逻辑与:",boolean_express && !num_express);//true
    console.log("逻辑与:",!boolean_express && !num_express);//false
    console.log("逻辑与:",!boolean_express && num_express);//false

    console.log("逻辑或:",boolean_express || num_express) //true
    console.log("逻辑或:",!boolean_express || num_express) // false
    console.log("逻辑或:",boolean_express || !num_express) //true
    console.log("逻辑或:",!boolean_express || !num_express) // true
```

- **全等（===）**

```
console.log("123" == 123)  //true 先比较值 或者比较自动转型的值
    console.log("123" === 123) //false 先比较类型 再比较值

    const obj = {
        name:"nuanxin",
    }
    const obj2 = {
        name: "nuanxin"
    }

    console.log("引用类型标记 ==",obj == obj2); //false
    console.log("引用类型比较 ===",obj === obj2); //false

    const obj3 = {
        a:1
    }

    const obj4 = obj3;
    console.log("引用类型比较 ==", obj3 == obj4); //true
    console.log("引用类型比较 ===", obj3 === obj4);//true

```

- **三元表达式（lamda）**

```
    const res = 2-1;
    if(res>0){
        console.log("结果为正数");
    }else{
        console.log("记过为负数或为0")
    }

    //表达式 ? 真 :假
    const res2 = res >0 ? console.log("结果为正数") :
console.log("结果为负数或为0")
    console.log(res2);
```

3.5 Date对象

- **全球计时规则**
  - 格林威治 GMT (不标准)
  - UTC (很准)

- **获取当前时间**

```
let NOW = new Date();
```

- **指定日期**

```
  //第一种指定日期格式
  let someDate01 = new Date(Date.parse("Mar 1,2022"));
  let someDate02 = new Date(Date.parse("3/1/2022"));
  console.log(someDate01);
  console.log(someDate02);
  //第二种指定日期格式
  let somedate03 = new Date("3/8/2022");
  let somedate04 = new Date("2022-3-7-17:00:00");
  console.log(somedate03);
  console.log(somedate04);
```

- **Date.UTC()**

```
   //以下例子证明UTC和GMT相差8个小时
    const d1 = new Date(Date.UTC(2022,2,8,1,0,0));
    console.log(d1);  //Tue Mar 08 2022 09:00:00 GMT+0800
(中国标准时间)
    const d2 = new Date(Date.UTC(2022,3));
    console.log(d2);  //Fri Apr 01 2022 08:00:00 GMT+0800
(中国标准时间)
    //我们日常使用可以省略Date
    //这种写法系统会将UTC → GMT 无8小时差别
    const d4 = new Date(2022,2);
    console.log(d4); //Tue Mar 01 2022 00:00:00 GMT+0800 (中
国标准时间)
    const d5 = new Date(2022,2,8,9,0,0);
    console.log(d5); //Tue Mar 08 2022 09:00:00 GMT+0800 (中
国标准时间)
```

- **归纳总结(五种获取时间)**

```
    let sd1 = new Date("Mar 1,2022");
    let sd2 = new Date("3/8/2022");
    let sd3 = new Date("2022-03-01T09:28:00");
    let sd4 = new Date(2022,4);
    let sd5 = new Date(2022,4,18,9,30,0);

    console.log(sd1,sd2,sd3,sd4,sd5);
```

- **Date.now()**
  - **返回一串数值.此时到1970毫秒数值差**
  - **他的使用场景, 测试 方法的 运行时间**

```
    const start = Date.now();
    loop();
    const end = Date.now();
    console.log(`loop执行了${end-start}毫秒`);
    console.log(`loop执行了${(end-start)/1000}秒`);*/
```

- **Date的常用方法**

```
    const date = new Date();

    console.log("getFUllYear获取年份",date.getFullYear()); //获取年
    console.log("getMonth获取年份",date.getMonth());  //从0开始  获取月
    console.log("getDate获取月中的哪一天",date.getDate()); //获取天
    console.log("getDay获取周中哪一天",date.getDay());  //从1开始 0结束  获取星期几

    const sepDate = new Date(2022,2,12);
    console.log("sepDate是哪一天:",sepDate); //2022-3-12 00:00:00
    console.log("3月12号是一周哪一天",sepDate.getDay()); // 6(星期六)
```

- 作业

1. html,input,yyyy-mm-dd
2. 根据yyyy-mm-dd转成指定date,获取星期几 getDay
3. 再input 下方输出一句话,这个星期几是所在月份中的第几个星期几?

- 答案解析

```
<!DOCTYPE html>
<html lang = "en">
<head>
    <meta charset = "UTF-8">
    <title>js修改</title>
</head>
<body>
  <input type = "text" id="input_el" placeholder="yyyy-MM-dd">
  <button onclick="sub()">转换</button>
  <p id="ex"></p>
</body>
</html>
```

```
 function sub () {
        const input_val = document.querySelector("#input_el").value; //获取表单的值
        const arr_val = input_val.split("-"); //将字符串以“-"形式进行分割成字符串数组
        const  month_int = parseInt(arr_val[1],10);
        const date_int = parseInt(arr_val[2],10);
        const date = new Date(arr_val[0],month_int-1,date_int); //指定日期
        const res_month = date.getMonth()+1;  //月份是从0开始的 所以要加1
        const res_day  = date.getDay() === 0 ? "日":date.getDay(); //显示星期日而不是0
        let count = 0 ;
        for(i=date_int;i>0;i=i-7){
            count++;
           }
document.getElementById("ex").innerHTML =`这是${res_month}月份中第${count}个星期${res_day}`;}
```

3.6 RegExp

- **正则表达式**
  - 根据正则的语法可以达到很方便的模糊查询
    - **match方法**

```
 //在js如何声明
 let express = /正则的语法 / 宽容度 ;
 let express = /pattern /flags ;
 let express = new RegExp(pattern ,flags);

/* flags
          g - global ;
          i - 大小写不敏感
          m - 支持换行
          y - 粘附模式 lastIndex
          u - unicode
*/

    const pattern1 = /l/g;
    const res1 = "hello".match(pattern1);
    console.log('res1', res1);  // ['l', 'l']

    const pattern2 = /l/;
    const res2 = "hello".match(pattern2);
    console.log("res2",res2); // ["l"]

    const pattern3 = new RegExp("l","g");
    const res3 = "HELLO".match(pattern3);
    console.log('res3', res3);  //null

    const pattern4 = new RegExp("l","i");
    const res4 = "HELLO".match(pattern4);
    console.log('res4', res4); //["l"]

    const pattern5 = new RegExp("l","gi");
    const res5 = "HELLO".match(pattern5);
    console.log('res5', res5); // ['L', 'L']

    const pattern6 = /[HEL]/g;
    const res6 = "HELLO".match(pattern6);
    console.log('res6', res6);  // ["H","E","L","L"]

    const pattern7 = /[^HEL]/g;
```

```
    const res7 = "HELLO".match(pattern7);
    console.log('res7', res7);//["O"]

    const pattern8 = /[0-9]/g;
    const res8 = "hello178".match(pattern8);
    console.log('res8',res8); //["1","7","8"]

    const pattern9 = /./g;
    const res9 = `!
    `.match(pattern9);
    console.log('res9', res9);  //["!","\t"]

    //match    ⇒ 不返回原来字符串的索引
    //(要搜索的目标字符串).match(正则表达式)
    const patternMatch1= new RegExp("^[A-z]\\d+$","g");
    const patternMatch2= new RegExp("^[A-z][0-9]+$","g");
    const patternMatch3 = /^[A-z]\d+$/g;

console.log("patternMatch1","a123".match(patternMatch1));

console.log("patternMatch2","a123".match(patternMatch2));

console.log("patternMatch3","a123".match(patternMatch3));
```

- **匹配手机号 / 座机号**

```
*^:匹配任何开头为n的字符串*

    *$:匹配任何结尾为n的字符串*
    *target:17812345678*
    *第一种表示形式*
    *^1:第一位为1*
    *\d{9}$:最后9位是0-9*
    *[3456789]:第二位是3-9*

    *第二种表示形式*
    *^1:第一位为1*
    *\d{9}$:最后9位是0-9*
    *(3|4|5|6|7|8|9) :第二位 3-9*

    *座机号码 target:0552-6903596*
    *^((d{3,4})|d{3,4}-|s):*
    *(\d{3,4})  3-4 个数字*
    *\d{3,4}=  3-4个数字和-*
```

```
*\s      空格*

*\d{7,14}: 结尾7-14个数字*
*(3|4|5|6|7|8|9);第二位 3-9*
```

```
const pattern12 = /^1[3456789]\d{9}$/
const pattern13 = /^1(3|4|5|6|7|8|9)\d{9}$/
const pattern14 = /^((\d{3,4})|\d{3,4}-|\s)?\d{7,14}$/

const target1 = "17882671234";
const target2 = "0552-6903596";
const res12 = target2.match(pattern12);
const res13 = target2.match(pattern13);
const res14 = target2.match(pattern14);
console.log("res12",res12);
console.log("res12",res13);
console.log("res12",res14);
```

- **exce** ()
  - 只接受一个参数，找到匹配项，则就返回包含第一个匹配信息的数组；要想查找多个匹配项，需要多次调用；
  - 经常与while循环搭配使用

```
const pattern = /o/g;
    const exec_res1 = pattern.exec("hello world");
    console.log("exec_res1",exec_res1); //  ['o', index: 4, input: 'hello world']

    const exec_res2 = pattern.exec("hello world");
    console.log("exec_res2",exec_res2); //[ ['o', index: 7, input: 'hello world']]

    const exec_res3 = pattern.exec("hello world");
    console.log("exec_res3",exec_res3); // ['o', index: 4, input: 'hello world']

//exec 配合 while循环
    const str = "i am NunXin,hello world. l love study";
    const pattern3 = /d/g;
    let exec_while = pattern3.exec(str);
    while (exec_while !== null){
        console.log("exec_while",exec_while);
        exec_while = pattern3.exec(str);
```

```
18      }
19
```

- **text** ()
  - 接受一个字符串参数，如果输入文本与模式匹配，则返回true，否则返回false，适用于只想测试模式是否匹配；
  - 经常用在if语句中；

```
1  // test
2      // 调用者是正则表达式  返回值是布尔  true/false
3      const patternTest1 = /^我/g;
4      let str1 = "我是暖心";
5      const res_test1 = patternTest1.test(str1);
6      console.log("res_test1",res_test1);
7
8      const patternTest2 = /\.$/g;
9      const str2 = `我是暖心.`;
10     const res_test2 = patternTest2.test(str2);
11     console.log("res_test2",res_test2);
12
13     const patternTest3 = /包*/g; //任意次
14     const patternTest4 = /包+/g; // 必须出现一次
15     const patternTest5 = /包?/g; // 0次或者1次
16     const res_test_3 = patternTest3.test("我要买包");
17     const res_test_4 = patternTest4.test("我要买衣服");
18     const res_test_5 = patternTest5.test("我要包子");
19     console.log("res_test_3",res_test_3); //true
20     console.log("res_test_4",res_test_4); //false
21     console.log("res_test_5",res_test_5); //true
```

3.7 Math方法

- **随机数**

```
1      //随机数
2      console.log(Math.random()); //[0,1)
3      //常规的使用方法是乘以一个倍数
4      console.log(Math.random()*5); //[0,5)
5      console.log(Math.random()*10); //[1,10)
6      //获取随机整数
7      console.log(Math.floor(Math.random()*10)+1);// [1,10)
```

- **舍入方法**

```
    //Math.ceil 向上取整
    console.log(Math.ceil(1.1)) // 2
    //Math.round 四舍五入
    console.log(Math.round(1.5)) // 2
    console.log(Math.round(1.4)) // 1
    //Math.floor 向下取整
    console.log(Math.floor(1.9)) //1
```

- **min ( ) 和 max ( )**

```
let max = Math.max(3,34,12,65);
console.log(max) //65

let min = Math.min(3,34,12,65);
console.log(min) //3
```

3.8 循环语句

- **while循环**

```
let step = 10;
    while (step>0){
        console.log(step);
        step--;
    }
```

- **for循环**

```
for (let i = 0; i <3 ; i++) {
        console.log("for i",i);
    }

//for循环改写while循环
    let i =0,target = 3;
    while(i<target){
        console.log(i);
        i++;
    }
```

- **for-in**
  - for-in 对象

```
const obj = {
        name:"nuanxin",
        score:100,
        work:true,
        a:1,
        b:2
    }

    for (let key in obj) {
        console.log("obj:",key);
    }

   //对象属性描述信息
   const val =
Object.getOwnPropertyDescriptor(obj,"name");
   console.log(val);
   //控制台打印信息
   configurable: true  // 属性是否可以删除
   enumerable: true   //属性是否可以枚举
   value: "nuanxin"   //属性值
   writable: true    //属性是否可以修改


    function Obj1 () {
        this.name = "naunxin";
        this.score = 90;
        this.work = true;
    }

```

```
    const obj1 = new Obj1();

    Object.defineProperty(obj1,"a",{
        value:1000,
        enumerable:true //for-in 可迭代
    })

    Object.defineProperty(obj1,"b",{
        value:2000,
        enumerable:false // for-in 不可迭代
    })

    for (let key in obj1) {
        console.log("obj1:",key);
    }
```

- for-in 数组索引

```
const arr = ["a","b","c","d","e"]

    for (let i in arr) {
        console.log("arr的索引:",i);
    }
```

- **for-of**
  - **for-of只能迭代有iterable的属性的对象,**

    但一般声明对象时是没有iterable的属性.

    for-of 一般情况下,只能迭代数组

```
const arrForof=[
        {a:1},
        {b:2},
        {c:3},
        {d:4},
        {e:5}
    ]

    for (let val of arrForof) {
        console.log("arrForof的元素:",val);   // {a:1},{b:2},{c:3},{d:4},{e:5}
    }
```

```
var student={
    name: "wujunchuan",
    age: 22,
    country: "china",
    city: "xiamen",
    school: "XMUT"
}
for(let key of student){
    console.log(key + ":"+ student[key] );
}   // student is not iterable

// 解决方案 遍历对象的属性
for(var key of Object.keys(student)){
    //使用Object.keys()方法获取对象key的数组
    console.log(key+":"+student[key]);
}
```

- **总结 迭代方式**
  - **while**
  - **for{;;} for-loop**
  - **for-in**
  - **for-of**
- **我们同时循环一个大数组,然后看哪个最快**

```
    let testArr = [];
    for (let i = 0; i < 10000000; i++) {
        testArr.push(i);
    }
     //while循环
    let start_while = Date.now();
    let while_step = 0;
    while (while_step<testArr.length){
        while_step++;
    }
    let duration_while = Date.now()-start_while;
    console.log(`while: ${duration_while} ms`);

    //for循环
    let start_for = Date.now();
    for (let i = 0; i <testArr.length ; i++) {
    }
    let duration_for = Date.now() - start_for;
    console.log( `for :${duration_for}ms`);

```

```
    //for-in
    let start_in = Date.now();
    for (let i in testArr) {
    }
    let duration_in = Date.now() - start_in;
    console.log(`for-in :${duration_in}ms`);

    //for-of
    let start_of = Date.now();
    for (let i of testArr) {
    }
    let duration_of = Date.now() - start_of;
    console.log(`for-of :${duration_of}ms`);

    结果: while: 28 ms
         for :11ms
         for-in :2047ms
         for-of :90ms
```

- break

```
    let num = 0;
    let arr = [];
    for (let i = 1; i < 10 ; i++) {
        if(i % 5 === 0){
                break;
        }
          arr.push(i);
          num++;
    }
    console.log(arr);
    console.log(num);
```

- **continue**

```
      let num1 = 0;
      let arr1 = [];
      for (let i = 1; i <10 ; i++) {
          if( i % 5 === 0){
                  continue;
          }
            arr1.push(i);
            num1++;
      }
      console.log(arr1);
      console.log(num1);
```

- **switch**

```
 function checkNumOrRule2 (val) {
        switch (true){
            case val === "+" || val === "-" || val === "/" || val==="*":
                console.log("运算符号");
                break;
            case val>=0 && val<=9:
                console.log("数值");
                break;
            default:
                console.log("非法输入");
                break;
        }
    }

  function dict (val) {
        let res = "";
        switch (val) {
            case 0 :
                console.log("00000");
                res = "创建成功";
                break;
            case 1 :
                console.log("11111");
                res = "准备发货";
                break;
            case 2 :
                res = "运输中";
                break;
```

```
29                 case 3 :
30                     res = "物流分炼";
31                     break;
32                 case 4 :
33                     res = "配送中";
34                     break;
35                 case 5 :
36                     res = "订单完毕";
37                     break;
38                 default :
39                     res = "该值属于非法值";
40                     break;
41             }
42             return res;
43         }
44
```

四.数组

- **创建数组**

```
1 const arr1 = [];  //[]
2 const arr2 = new Array();  //[]
3 const arr3 = new Array(3);  //[, , ,] //创建一个包含3个元素的数
  组
4 const arr5 = new Array("a","b","c"); //["a","b","c"]
```

- Array.from() 用于将 类数组 结构转换为 数组实例

```
1 // 字符串会被拆分为单字符数组
2 console.log(Array.from("Matt")); // ["M", "a", "t", "t"]
3 //可以传回调函数
4 const arr9 = Array.from([1,2,3],function (val) {
5         return val**3;
6     })
7  console.log(arr9);  // [1,8,27]
```

- **Array.of() 可以把一组参数转换为数组**

```
console.log(Array.og(a,b,c,d));  //[a.b.c.d]
```

- **检测数组** `instanceof`

```
const arr = [1,2,3];
console.log(arr instanceof Array)  // true
```

- **转换方法** （`toString` `valueOf`）

  `toString`：数组中每个值的等效字符串拼接而成的一个逗号分隔字符串；

  `valueOf`：返回数组本身；

```
const colors = ["red","green","blue"];
console.log("toString",colors.toString()); // red,green,blue
console.log("valueOf",colors.valueOf());//
["red","green",blue]
```

- **类似栈的方法** `Stack`

  `push`：添加数组；

  `pop`：删除数组的最后一项；

```
const arr11 = [];
    arr11.push("a");
    arr11.push("b");
    arr11.push("c");
    console.log("arr11:",arr11); //["a","b","c"]
    const popRes = arr11.pop();
    console.log("pop",popRes); // ["c"]
    console.log("arr11:",arr11); // ["a","b"]
```

- **队列方法**

  `shift`:删除数组第一项；

  `unshift`:给数组开头添加数据；

```
const arr12 = [];
    arr12.push("a");
    arr12.push("b");
```

```
    console.log("arr12:",arr12); // ["a","b"]
    const shiftRes = arr12.shift();
    console.log("shift:",shiftRes); // ["a"]
    console.log("arr12:",arr12);    // ["b"]

  //unshift方法(插队)
    const arr13 = [];
    arr13.push("1");
    arr13.push("2");
    arr13.push("3");
    console.log(arr13); // ["1","2","3"]
    arr13.pop(); //"3"
    const shift_res = arr13.shift();//删除第一个 "1"
    console.log(shift_res);// "1"
    console.log("arr13:",arr13); //["2"]
  arr13.unshift("3"); //插队一个"3"
    console.log("arr13:",arr13); //["3","2"]
```

- **数组逆置** （**reverse**）

```
    const arr14 = [1,2,3,4,5];
    const arr15 = ["a","b","c"];
    console.log(arr14.reverse());  //[5,4,3,2,1]
    console.log(arr15.reverse());  //["c","b","a"]
```

- 数组排序（**sort**）

```
js中内置的sort它的规则,将数组先转成string,
然后再从左往右,一位一位的比较;
```

```
    const arr19 = [1,"a","hello",true,5,11]

console.log(arr19.sort());//[1,11,5,"a","hello",true]

   //自定义排序
    const arr20 = [1,5,3,2,6,8];
    arr20.sort(function (val1,val2) {
          if(val1>val2){
                return 1;
            } else if(val1<val2){
                    return -1;
          } else{
```

```
            return 0 ;
        }
    })
    console.log(arr20);//升序

    const arr21 = [1,534,54,223,24,33,45,3];
    arr21.sort((a,b)=> a > b ? -1 : a<b ? 1 : 0)
    console.log(arr21); //降序
```

- **concat ()**

  concat 和 push 的区别

  push

  - 将整个参数作为一个元素放到原数组中
  - 修改原始数组

  concat

  - 将参数先平铺flat,然后再一个一个追加
  - 不会修改原始数组,但会将新的数组作为concat方法的返回值返回

```
    const arr = [1,2,3];  //原数组
    const append_arr = [4,5,6];
    arr.push(append_arr);  //push有返回值 但返回是参数
    console.log("arr原数组:",arr); //[1,2,3,[4,5,6]]

    const arr2 = [1,2,3];
    const arr2_concat_res = arr2.concat(append_arr);
[1,2,3,4,5,6]
    console.log("concat结果",arr2_concat_res);
    console.log("arr2原数组:",arr2);  //原数组不变 [1,2,3]

   //优化字符串拼接性能
    const str1 = "hello";
    const str2 = "concat";
    const arr3 = Array.from(str1); //['h', 'e', 'l',
'l', 'o']
    const arr4 = Array.from(str2); //[c', 'o', 'n', 'c',
'a', 't']
    const arr5 = arr3.concat(" ",arr4);//['h', 'e', 'l',
'l', 'o', ' ', 'c', 'o', 'n', 'c', 'a', 't']
    const str3 = arr5.join("");// 以“”形式连接字符串
    console.log(str3); //hello concat
```

- **slice** **[开始位置,数组长度)**

```
    const arr6 = ["a","b","c","d","e"];
    console.log("slice 结果",arr6.slice(3));
    console.log("原数组",arr6); //['d', 'e']

    console.log("slice 结果",arr6.slice(1,3));
    console.log("原数组",arr6); // ['b', 'c']

    console.log("slice 结果",arr6.slice(-3));
    console.log("原数组",arr6); //['c', 'd', 'e'] 负数加上数组长度  ⇒ -3 + 5 = 2

    console.log("slice 结果 ",arr6.slice(-3,-2)); // -3+5=2 / -2 +5 =3 ⇒[2,3)
    console.log("原数组",arr6); // ["c"]
```

- **splice** **(删除 插入 替换)**

  `splice(p1,p2,p3 ... )`

  `splice会修改原数组`

  - **p1(必传) :定位数组的索引,开始操作的位置**
  - **p2(必传) :需要删除的个数**
  - **p3(可选) :可以传一个,多个,甚至不传**

  插入操作

```
const arr7 = ["a","b","c","d","e"];
const addSpliceRes1 = arr7.splice(1,0,"w","x","y","z");
console.log("addSpliceRes1",addSpliceRes1);// [] 会返回空数组
console.log("splice 增加",arr7);// ['a', 'w', 'x', 'y', 'z', 'b', 'c', 'd', 'e']

const arr8 = Array.from("helloworld");
const addSpliceRes2 = arr8.splice(5,0," ");
console.log(addSpliceRes2); // ["h"]
console.log("arr8 增加",arr8);
console.log(arr8.join(""));
```

  删除操作

```
    const arr9 = ["a","b","c","d","e"]
    const delSpliceRes1 = arr9.splice(1,3);
    console.log("delSpliceRes1",delSpliceRes1);
    console.log("splice 删除",arr9) //["a","e"]
```

替换操作

```
    const arr10 =["a","b","c","d","e"];
    const replaceSpliceRes1 = arr10.splice(1,3,"1",'2',"3");
    console.log("replaceSpliceRes1",replaceSpliceRes1); //["b","c","d"]
    console.log("arr10",arr10);  //["a","1",'2',"3","e"]

    const arr11 =["a","b","c","d","e"];
    const replaceSpliceRes2 = arr11.splice(1,3,"≠");
    console.log("replaceSpliceRes2",replaceSpliceRes2); //["b","c","d"]
    console.log("arr11",arr11); //["a","≠","e"];
```

```
      /*
        js增强语法( ... )
        数据的平铺/展开
    */

    //小练习
    const arr12 = Array.from("i am man");
    arr12.splice(5,0,"w","o")
    console.log(arr12.join("")); //i am woman
    console.log("=======")
    const arr13 = Array.from("i am man");
   // arr13.splice(5,3,"w","o","m","a","n");
    arr13.splice(5,3,...Array.from("woman"));//采用平铺的方式(flat)
    console.log(arr13.join(""));  //i am woman
```

- **indexOf**
  - **第一种方法(indexOf只传第一个参数)**

```
    const str_time = "2022-3-11";
    const indexOfLine = str_time.indexOf("-");//4;
    const year = str_time.substring(0,indexOfLine);//2022
    const rest2 = str_time.substring(indexOfLine+1); //截取剩余的字符串 3-11
    const indexOfLine2 = rest2.indexOf("-");//重新获取"-"索引
    const month = rest2.substring(0,indexOfLine2); //3
    const day = rest2.substring(indexOfLine2+1); //11
    console.log(year,"年",month,"月",day,"日"); //2022年3月11日
```

- **第二种方法(indexOf传两个参数)**

```
    const str_time2 = "2022-3-11";
    //                  012345678
    const i = str_time2.indexOf("-"); // 4
    const i2 = str_time2.indexOf("-",i+1); //截取第二个"-" → 6
    const Year = str_time2.substring(0,i) //下标[0,4) → 2022
    const Month = str_time2.substring(i+1,i2);// 截取下标[5,6) → 3
    const Day = str_time2.substring(i2+1); //下标 [7,8] 剩下全部截取完 ⇒11
    console.log(Year,"年",Month,"月",Day,"日"); //2022年3月11日
```

- **五种迭代方法** **(不改变调用它们的数组)**
  - `every()`: 对数组每一项都运行传入的函数，如果对每一项函数都返回 true.,则这个方法返回 true。

```
    /*[a,b,c].every(function(p1,p2,p3)){}
      p1:每次迭代的元素值
      p2:每次迭代的元素索引
      p3:原始数组
    */
     const arr = [1,2,3,4,5,6];
     const res1 = arr.every(function (item,index,obj){
          return item > 0;
      })
```

```
    console.log("arr中是否大于0",res1); //true

    const res2 = arr.every(function (item,index){
        console.log( `index${i} value ${item}`)
        return item <5;
    })
    console.log("arr中是否都小于5",res2);    //false
```

- `some()`:对数组每一项都运行传入的函数，如果有一项函数返回 true,则这 个方法返回 true。

```
/*[a,b,c].some(function(p1,p2,p3)){}
    p1:每次迭代的元素值
    p2:每次迭代的元素索引
    p3:原始数组
*/
 const arr = [1,2,3,4,5,6];
 const res = arr.some(function (val) {
     return val % 5 ===0;
 })
 console.log("数组中是否存在能被5整除的",res);  //true
```

- `filter()`:对数组每一项都运行传入的函数，函数返回 true 的项会组成数组之后返回。**(根据条件进行过滤)**

```
/*[a,b,c].filter(function(p1,p2,p3)){}
    p1:每次迭代的元素值
    p2:每次迭代的元素索引
    p3:原始数组
*/
 const arr = [1,2,3,4,5,6];
 const res1 = arr.filter(function (item,i){
       console.log(`${item} %2 = ${item%2}`);
       return item % 2 === 0;
   })
 console.log("filter过滤数组",res1); // [2, 4, 6]
 console.log("原数组",arr); // [1,2,3,4,5,6]

 const res2 = arr.filter((item,i)=>{
       return item % 2 !== 0;
   })
```

```
    console.log("filter 过滤的数组",res2) ; //[1.3.5]
    console.log("原数组",arr); // [1,2,3,4,5,6]

    const res3 = arr.filter((item,i)⇒{
          return item % 4 === 0 ;
    })
    console.log("filter 过滤数组 ",res3); //[4]
    console.log("原数组",arr);  // [1,2,3,4,5,6]
```

- `map()`：对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组。

```
  /*[a,b,c].map(function(p1,p2,p3)){}
      p1:每次迭代的元素值
      p2:每次迭代的元素索引
      p3:原始数组
  */
    const arr = [1,2,3,4,5,6];
    const res = arr.map((val)⇒{
         return val ** 2 ;
     })
     console.log("map 加工后的",res); //[1, 4, 9, 16, 25, 36]
     console.log("原数组",arr); //  [1,2,3,4,5,6];
```

- `forEach()`：对数组每一项都运行传入的函数，没有返回值。

```
 /*[a,b,c].forEach(function(p1,p2,p3)){}
      p1:每次迭代的元素值
      p2:每次迭代的元素索引
      p3:原始数组
 */
// forEach就是迭代或循环数组的语法糖
// for while for-in for-of foreach
  const arr = [1,2,3,4,5,6];
  arr.forEach(function (item,i){
        console.log(`索引${i},值${item}`);
    }) //进行数组遍历
```

- **归并方法（`reduce`）**
  - reduce (匿名函数，`初始值`) ：迭代数组所有项，在此基础上构建一个最终返回值；

```
 /*[a,b,c].reduce(function(p1,p2,p3,p4)){}
       p1:上一个归并值
       p1:当前项
       p2:当前项的索引
       p3:原始数组
 */
    const arr = [1,2,3,4,5,6];
    const res1 = arr.reduce(function (acc,item,i,obj){
        return acc + item ;
    },0)
    console.log("数组累加值",res1); // 0+1+2+3+4+5+6 =21

/*
    当reduce不传第二个参数也就是初始值时，
    我们把数组中索引为0的那个元素作为初始值
    然后reduce的累加从索引为1的元素开始
    明显的不同就是，不传第二个参数时，少遍历了一次
*/
    const res2 = arr.reduce(function (acc,item,i,obj){
        return acc + item ;
    })
    console.log("res2数组累加值",res2); // 1+2+3+4+5+6
=21
```

- **作业练习**

作业要求：在0 ~ 1000 正整数中 (0,1000],计算出反时能被5整除的元素的累加值

要求:用两种方案解决

- 答案解析

```
//方法一
    const arr1 = [] ;
    for(let i=1;i⩽1000;i++){
        arr1.push(i);
    }
    const res3 = arr1.filter(function (item,i){
        return item % 5 === 0;
    })

    const res4 = res3.reduce(function (acc,item){
         return  acc + item;
    })
    console.log("能被5整除的累加值:",res4);     //100500
```

```
14
15      //方法二
16      let sum = 0;
17      const res5 = res3.forEach(function (val) {
18           return sum += val ;
19      });
20      console.log(sum);   //100500
21      //方法三
22      const res6 = arr1.reduce(function (acc, val) {
23           if(val %5 == 0){
24               return acc + val ;
25           }else{
26               return acc;
27           }
28      },0)
29      console.log(res6);  //100500
```

五.函数

5.1 声明函数

- 函数声明

```
function sum (a,b) {
      return a+b ;
}
```

- 函数表达式

```
const sum = function(a,b){
      return a+b ;
}
```

- 使用Function构造函数 (不推荐使用 代码被解释二次，显然影响性能)

```
const sum2 = new Function("a","b","return a+b");
console.log(sum2(3,5)); //8
```

5.2 函数内置对象

- **arguments**是一个 类数组对象 (**不是Array的实例**),可以用length查看长度还可以根据[ ] 访问数组的某个元素

```
function foo (a,b) {
        console.log("arguments",arguments);  [123,"abc"]
        console.log("arguments.length",arguments.length); //2
        console.log("arguments",arguments[0]); // 123
        console.log("arguments",arguments[1]); // abc
    }
    foo(123,"abc");

//严格模式下 arguments还是之前传入的参数 改值无效
"use strict";
    function abr(p1,p2){
        p2 =100 ;
        console.log("p1",p1);   //123
        console.log('p2', p2);  //456
        console.log('arguments[0]', arguments[0]); //123
        console.log('arguments[1]', arguments[1]); //456
    }
abr(123,456);

/*arguments对象有一个属性callee, 是一个指向arguments对象所在函数的指针;
  但是arguments.callee 不能再严格模式下执行
 */
 function fn (str) {
        console.log(str);
        if (str === "hello"){
             arguments.callee("world"); 等价于⇒ fn("world");
            //这样写 可以解除函数逻辑和函数名的耦合性（解除耦合性）
        }
    }
    fn("hello");
```

5.3 箭头函数

- **箭头函数: 使用胖箭头 => 定义的函数就是箭头函数;**
- **箭头函数的声明方式只有函数表达式 ;**
- **箭头函数不绑定上下文; lambda(一句话编程)**
- **箭头函数不能使用 arguments 、super、new.target , 也不能使用 构造函数 ;**
- **箭头函数也没有 prototype 属性;**

```
    let sum1 = (a,b) => {
        return a+b ;
    }
    console.log(sum(1, 2)); //3
   //如果只有一个参数, 可以省略小括号
   //函数体只有一行代码或者一个表达式 可以省略大括号
    const sum2 = (c,d) => c+d;
    console.log(sum2(1,4)); //5

    const sum3 = () => console.log(2000);
    const sum4 = (p1,p2) => console.log(p1+p2); // 9
    sum4(3,6);
```

5.4 this

this在普通函数指向

- **function 绑定上下文(this) 规则是 看最后是"谁"调用的 ,那么this就指向"谁"**
- 构造函数中new关键字做了什么? =>new 会创建对象，将构造函数中this指向创建出来的对象;
- 计时器 里面的函数是 全局函数 , this会指向window;
- **在严格模式下，this 会返回 undefined**

```
    //场景一
    var name = "外面的名字";
    const obj = {
        name:"里面的名字",
        fn:function () {
            console.log('场景一', this.name);
        }
    }
    obj.fn();  //  里面的名字

    //场景二
    var name2= "外面的名字";
    const obj2 = {
```

```
        fn:function () {
            console.log('场景二', this.name2);
        }
    }
    obj2.fn();  //undefined

    //场景三
    var name3 = "外面的名字";
    const obj3 = {
        name3:"里面的名字",
        fn:function () {
            console.log('场景三', this.name3);
        }
    }
    const fn_tmp = obj3.fn;
    fn_tmp();   //外面的名字

    //场景四
    var name4 = "外面的名字";
    function fn4() {
        var name4 = "里面的名字";
        innerFn();
        function innerFn(){
            console.log("场景四",this.name4)
        }
    }
    fn4();   //外面的名字

    //场景五
    window.identity = "i am window";
    const object = {
        identity:"i am object",
        getId:function () {
            return function () {
                return this.identity;
            }
        }
    }                       //IIFE(立即执行函数)
    console.log('场景五', object.getId()());  // i am window

    window.identity2 = "i am window";
    const object2 = {
        identity2:"i am object",
        getId:function () {
```

```
            let that = this;  //相当于把object做了一次快照,赋给that
            return function () {
                return that.identity2;
            }
        }
    }
    console.log('场景五', object2.getId()());  //i am object
```

this在箭头函数指向

- 箭头函数不绑定上下优点:在书写代码时,就能确定上下文this指向是谁,永远指向的是箭头函数声明时,所在的作用域
- 箭头函数外指向谁就指向谁 或者 在哪里定义就指向谁

```
//场景一
var name_1 = "外面的名字";   // window.name = "外面的名字";
    const obj_1 = {
        name_1:"里面的名字",
        fn:() =>{
            console.log('场景一', this.name_1);
        }
    }
    obj_1.fn();    //外面的名字

//场景二
    var name_2 = "外面的名字";
    const obj_2 = {
        fn:() =>{
            console.log('场景二', this.name_2);
        }
    }
    obj_2.fn();  // 外面的名字

//场景三
    var name_3 = "外面的名字";
    const obj_3 = {
        name_3:"里面的名字",
        fn:() =>{
            console.log('场景三', this.name_3);
        }
    }
    const fn_Three = obj_3.fn;
    fn_Three();     //外面的名字
```

```
30
31  //场景四
32      let name_4 = "外面的名字";
33      function fn_4(){
34          var name_4 = "里面的名字";
35          const inner_Fn = () => {
36              console.log("场景四",this.name_4);  //undefined
37          }
38        inner_Fn();
39      }
40  fn_4();
41
42
43  //总结几个例子
44      window.name ="init data";
45      const  obj = {
46          name:"zhang",
47          fn:() => console.log("场景一",this)
48      }
49
50      function foo(){
51          this.name = "li"
52          return {
53              name : "wang",
54              fn:() => console.log("场景二",this)
55          }
56      }
57
58      const bar = () => {
59          this.name = 1000
60          return {
61              name:2000,
62              fn: () => console.log("场景三",this)
63          }
64      }
65
66      obj.fn();  //window
67      foo().fn(); //window
68      bar().fn(); //window
```

- **call　apply　bind ： 都可以明确指定函数中的this是谁**
  - `call`和`apply`能够调用函数，但是apply传入的是数组或者对象，
  - `bind`不能调用函数，所以它是已返回值形式，返回一个函数，在进行调用；

```
1       function fn (){
```

```
            console.log("我是" + this.name);
    }

    const cat = {
        name:"喵喵",
    }
    fn.call(cat); // 我是喵喵

const dog = {
        name: "小黑",
        sayName() {
            console.log("我是" + this.name);
        },
        eat(food1, food2) {
            console.log("我喜欢吃" + food1 + food2);
        }
    }

    const cat = {
        name: "喵喵",
    }
    // dog.sayName();  // 我是小黑
     dog.sayName.call(cat); // 我是喵喵
     dog.eat.call(cat, "鱼", "肉"); //  我喜欢吃鱼肉
     dog.eat.apply(cat, ["鱼","肉"]); // 我喜欢吃鱼肉
    const fn =  dog.eat.bind(cat, "鱼","肉");  // 我喜欢吃鱼肉
     fn();
```

5.5 闭包

- **两个条件**

> ```
> 函数嵌套
> 内层函数使用/引用外层函数的变量
> ```

> **闭包的特点:**
>
> 1. 内部的函数一旦引用或使用外部函数的变量,
> 2. 那么这些变量就像"快照"一样,存在内存当中,不会被销毁，随时都可以引用
>
> **缺点**：容易造成 **内存泄漏**

```
function outer(a){
```

```
      function inner(){
          console.log(a);
      }
       return inner
  }

   const  inner_print = outer(2000);
   inner_print();  //2000

 function acc () {
      let res = 0;
      function sum () {
           res++
            console.log(res);  //1
      }
        eturn sum
   }
   const a = acc();
   a();

var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function dosomething() {
        console.log(something);
    }

    function doanother() {
        console.log(another.join("!"));
    }

    return {
        dosomething: dosomething,
        doanother: doanother
    }
})()

foo.dosomething(); //cool
foo.doanother(); //1!2!3
```

```
  // 闭包中循环
 for (var i = 1; i ⩽ 5; i++){
    setTimeout(function () {
        console.log(i);
```

```
    }
    ,i*1000)
}    // 5 5 5 5 5

 for (var i = 1; i ⩽ 5; i++) {
    (function () {
        setTimeout(function () {
            console.log(i);
        }, i * 1000)
    })();
}   //5 5 5 5 5


for (var i = 1; i ⩽ 5; i++){
    (function (j) {
        setTimeout(function () {
            console.log(j);
        }, j * 1000)
    })(i);
}   // 1 2 3 4 5
⟹ 相当于 let 闭包实现块作用域
 for (let i = 1; i ⩽ 5; i++){
    setTimeout(function () {
        console.log(i);
    }
    ,i*1000)
}  // 1 2 3 4 5
```

六. Canvas

```
 <canvas id="canvas"></canvas>
```

```
const canvas = document.getElementById("canvas");
    if (!canvas.getContext) {
        //不支持的时候
        const div = document.createElement("div");
        div.innerHTML = "你的浏览器不支持Canvas";
        document.body.appendChild(div);
    }
```

```
    const w = 500,h = 500;
    canvas.width = w;  // 画框宽度
    canvas.height = h; // 画框高度
    canvas.style.width = w + "px";  // 画板宽度
    canvas.style.height = h + "px"; // 画板高度
    const ctx = canvas.getContext("2d");

    ctx.beginPath();   // 开始绘制新路径
    ctx.moveTo(100, 50);  //起点
    ctx.lineTo(300, 50);  //终点
    ctx.lineWidth = "20"; //线条的宽度
    ctx.stroke();         //开始描画路径(描边)

    ctx.beginPath();
    /*
    butt  :    默认,向线条的每个末端添加平直的边缘
    round :  向线条的每个末端添加圆形线帽
    square : 向线条的每个末端添加方形线帽
    */
    ctx.moveTo(100, 100);
    ctx.lineTo(300, 100);
    ctx.lineCap = "round";
    ctx.lineWidth = "20";
    ctx.stroke();

    ctx.beginPath();
    ctx.moveTo(100, 150);
    ctx.lineTo(300, 150);
    ctx.lineCap = "square";
    ctx.lineWidth = "20";
    ctx.stroke();

    //绘制三角形
    ctx.beginPath();
    ctx.lineWidth = 1; // 再重新调整线的宽度
    ctx.moveTo(200, 200);
    ctx.lineTo(280, 280);
    ctx.lineTo(120, 280);
    ctx.closePath(); //直接回到起点
    ctx.strokeStyle = "purple" //描边颜色
    ctx.fillStyle = "skyblue"; // 填充颜色
    ctx.fill(); // 填充
    ctx.stroke(); // 描边
```