
ES6学习笔记

1 .定义变量

1.1 let

1.1.1 没有预解释

```
1.      {
2.          //1.let定义的变量没有变量的提前声明,不可以提前使用未被声明的变量（也叫没有
    预解释）
3.          console.log(a);
4.          //Uncaught ReferenceError: a is not defined
5.          let a=0;
6.
7.          function a(){};
8.          //Uncaught SyntaxError: Identifier 'a' has already been declare
    d
9.      }
```

1.1.2 不能重复定义

```
1.      //2.var 定义的变量可以多次声明，但let不行
```

```
    /*
        var b = 1;
        var b = 2;
        function b(){
            console.log(b);
        }
        console.log(b);//2
        b();//Uncaught TypeError: b is not a function
    */
    {
        //2
        //let定义的变量不能重复定义(不管是函数还是let声明的变量都不能重复)
        let b=1;
        //let b=2;//Identifier 'b' has already been declared
        function b(){};//Identifier 'b' has already been declared
        console.log(b);
    }
```

1.1.3 let 定义的变量会先判断有没有重复，然后才会执行

```
{
    //3
    //let定义的变量虽然不进行预解释，但是，代码执行的时候，一上来先进行的是将所有let声明的变量过滤一遍，一旦发现重复的（不合法），报错
    console.log(c);//这里不先报错的原因？
    let c=1;
    let c=2;//Uncaught SyntaxError: Identifier 'c' has already been declared
}
```

1.2 块级作用域

1.2.1 暂时性区

```
    let a = 10;
    if(1){
        //console.log(a);//Uncaught ReferenceError: a is not defined
        let a = 20;
        console.log(a);//20
    }
    console.log(a);//10
```

1.3 const

1.3.1 const 定义静态变量

```
    //1.没有变量提声
    //2.不可以重复声明
    //const 定义静态变量
    //一旦声明必须赋值
    const a=1;
    //a=2;
    let b;
    //const c;
```

1.3.2 全局属性

```
var a=1;
    //window.a=1
    console.log(window.a);
    console.log("a" in window);
    console.log(self == window);

    //let 声明的变量不会给全局当做属性
    let b=1;
    console.log(window.b);
```

2.数组赋值

2.1 数组的解构赋值

```
    //let a=1,b=2,c=3;
    //console.log(a,b,c);//1,2,3

    let [a,b,c] = [1,2,3];
    console.log(a,b,c);//1,2,3

    console.log('---------------');

    /*
    let ary1 = ["张三","李四","王五"];
    let [a,b,c]=ary1;//a' has already been declared
    console.log(a,b,c);
    */
    let ary1 = ["张三","李四","王五"];
    let [aa,bb,cc]=ary1;
    console.log(aa,bb,cc);//张三 李四 王五
```

2.2 数组的嵌套赋值

```
    let [a,b,[c],[[d,e]]]=[1,2,[4],[[5,6]]];
    console.log(a,b,c,d,e);//1 2 4 5 6
```

2.3 数组的省略赋值

```
    let ary=[1,2,3,4,5,6];
    //获取数组的指定索引位置的值
    //比如这里要索引1和索引4的值
    //let [a,b,c,d,e,f]
    //let [,x,,,y,]=ary;
    let [,x,,,y]=ary;
    console.log(x,y);//2 5
```

2.4 不定参数赋值

```
    let [a,b,...c]=[1,3,4,5,6,7];
    console.log(a);//1
    console.log(b);//3
    console.log(c);//(4) [4, 5, 6, 7]
```

2.5 默认值

```
    //1.let声明但未赋值的变量的值用undefined占位
    let [a,b]=[1];
    console.log(a);//1
    console.log(b);//undefined

    //2.let声明加赋值，只要赋的值不是undefined，都会赋值上
    //2.1
    let [c,d=2]=[1,"是我被赋值给d变量"];
    console.log(c);//1
    console.log(d);//是我被赋值给d变量

    //2.2
    let [e,f=2]=[1,undefined];
    console.log(e);//1
    console.log(f);//2

```

```
    //2.3
    let [g,h=2]=[1,null];
    console.log("g="+g);//g=1
    console.log("h="+h);//h=null

    //2.4
    function fn(){
        x=250;
        console.log('哈哈哈哈');
    }
    let [x=fn(),y=2]=[1];
    console.log("x="+x);//x=1
    console.log("y="+y);//y=2
```

3. 对象赋值

3.1对象的解构赋值

```
{
    //1.变量名==属性名
    let obj = {
        a:1,
        b:2,
        c:3
    }

    //let a = obj["a"];
    //let b = obj["b"];
    //let c = obj["c"];
    //console.log(a,b,c);//1,2,3

    let {a,b,c}=obj;
    console.log(a,b,c);//1,2,3
}

```

```
{
    //2.变量名与属性名不相等
    let {x,y,z} = {a:1,b:2,c:3};
    console.log(x,y,x);//undefined undefined undefined

    let {a,c,p} = {a:1,b:2,c:3};
    console.log(a,c,p);//1 3 undefined

    let obj =  {a:1,b:2,c:3};
```

```
    let {m,n} = obj;
    console.log(obj);//{a: 1, b: 2, c: 3}
}
```

```
    //2.对象解构赋值的原理
    //let {a:a,b:b,c:c} = {a:1,b:2,c:3};
    //console.log(a,b,c);//1,2,3
    //简写成如下
    //let {a,b,c} = {a:1,b:2,c:3};
    //console.log(a,b,c);//1,2,3

    //so
    //let {a:x,b:y,c:z} = {a:1,b:2,c:3};
    //console.log(x,y,z);//1,2,3

    //变量名! ==属性名
    let {a,b:x,c:y} = {a:1,b:2,c:3};
    console.log(a,x,y);//1,2,3
```

3.2对象数组的嵌套

```
{
    let obj = {
        a:[1,2,3],
        b:"250"
    }
    let {b,a:[x,y,z]}=obj;
    console.log(b,x,y,z);//"250" 1 2 3
}
```

```
{
        let obj = {
            o:{o1:1,o2:2,o3:[1,2,3]},
            p:[["p1","p2","p3"],[2,3,5]],
            q:[["qq"]]
        };

        let {
            o:{o2:xo2,o3:xo3},
            p:[py1,py2],
            q:[z],
            r="250"
        }=obj;

        console.log(xo2);//2
        console.log(xo3);//[1,2,3]
```

```
        console.log(py1);//(3) ["p1", "p2", "p3"]
        console.log(py2);//[2, 3, 5]
        console.log(z);//["qq"]
        console.log(r);//"250"
    }

```

3.3 对象解构赋值的默认值问题

```
{
        //1.对象解构赋值的默认值
        let obj={a:250};
        console.log(obj.b);//undefined

        let {x=1,y} = {y:250};
        console.log(x,y);//1 250
    }
```

```
{
        //2.解构赋值的默认值生效条件
        let {m:n} = {};
        console.log(n);//undefined

        let {p:q=2} = {};
        console.log(q);//2

        let {x:y=250}={x:null};
        console.log(y);//null

        //当且仅当  赋值对象的属性的值为undefined的时候，被赋值对象的属性的默认值才会生效
        let {a:b="bbnb"} = {a:undefined};
        console.log(b);//bbnb
```

```
{
        //3
        //
        // let {liu:{sex:1,age:25}} = {x:1,y:2};
        // console.log(liu);
        let {f:{b}} = {b:250};
        //Uncaught TypeError: Cannot destructure property `b` of 'undefined' or 'null'.
        、
        //f对应的后边的对象中没有这个f属性，所以f的值是undefined
        //给undefined 赋值等于一个 {b}，所以报错
        //console.log(b);
```

```
    }
```

3.4 对象解构赋值不是对象数据类型的问题

```
{
        //对象解构赋值的机制
        // 如果要赋值的  xxx  不是一个对象的时候，默认会执行Object(xxx),
        // 将xxx转为一个对象，然后再将转后进行赋值操作

        let {x,y} = {};
        console.log(x,y);//undefined,undefined

        //解构赋值后边的是 number类型的
        let {m,n,__proto__} = NaN;
        console.log(Object(NaN));//Number {NaN}
        console.log(m,n,__proto__);//undefined undefined Number {0, constructor: ƒ, toExponential: ƒ, toFixed: ƒ, toPrecision: ƒ, toString: ƒ, …}


        //let {p,q,__proto__,length} = "刘凯";
        //这里底层执行的是 Object("刘凯");
        //console.log( Object("刘凯"));//String {"刘凯"}

        //console.log(__proto__,length);
        //String {"", length: 0, constructor: ƒ, anchor: ƒ, big: ƒ, blink: ƒ, …} 2

    }
```

```
{
        //2 解构的是undefined和null  直接报错
        //let {x:b} = undefined;//Uncaught TypeError: Cannot destructure property `x` of 'undefined' or 'null'.
        let {y:a} =null;//Uncaught TypeError: Cannot destructure property `x` of 'undefined' or 'null'.
}
```

3.5 其他问题

```

let a ;
    //{a} = {a:"ahhh"};
    //javascript 引擎会将{a}理解成一个代码块
```

```
    //为了避免这样的错误，一般不要将{a}这样的代码卸载行首，
    //非得这么写参照eval，如下
    //Uncaught SyntaxError: Unexpected token =
    console.log(a);

    //eval("{a:'ahhh'}");
    //eval("({a:'ahhh'})");
    //
    ({a} = {a:"ahhh"});
    console.log(a);//ahhh
```

4. 字符串赋值

```

console.dir(new String('ss'));
String
    0: "s"
    1: "s"
    length: 2
    __proto__: String
        anchor: ƒ anchor()
        big: ƒ big()
        blink: ƒ blink()
        bold: ƒ bold()
        charAt: ƒ charAt()
        charCodeAt: ƒ charCodeAt()
        codePointAt: ƒ codePointAt()
        concat: ƒ concat()
        constructor: ƒ String()
        endsWith: ƒ endsWith()
        fixed: ƒ fixed()
        fontcolor: ƒ fontcolor()
        fontsize: ƒ fontsize()
        includes: ƒ includes()
        indexOf: ƒ indexOf()
        italics: ƒ italics()
        lastIndexOf: ƒ lastIndexOf()
        length: 0
        link: ƒ link()
        localeCompare: ƒ localeCompare()
        match: ƒ match()
        normalize: ƒ normalize()
        padEnd: ƒ padEnd()
        padStart: ƒ padStart()
        repeat: ƒ repeat()
        replace: ƒ replace()
        search: ƒ search()
        slice: ƒ slice()
```

```
        small: ƒ small()
        split: ƒ split()
        startsWith: ƒ startsWith()
        strike: ƒ strike()
        sub: ƒ sub()
        substr: ƒ substr()
        substring: ƒ substring()
        sup: ƒ sup()
        toLocaleLowerCase: ƒ toLocaleLowerCase()
        toLocaleUpperCase: ƒ toLocaleUpperCase()
        toLowerCase: ƒ toLowerCase()
        toString: ƒ toString()
        toUpperCase: ƒ toUpperCase()
        trim: ƒ trim()
        trimEnd: ƒ trimEnd()
        trimLeft: ƒ trimStart()
        trimRight: ƒ trimEnd()
        trimStart: ƒ trimStart()
        valueOf: ƒ valueOf()
        Symbol(Symbol.iterator): ƒ [Symbol.iterator]()
        __proto__: Object
             [[PrimitiveValue]]: ""
        [[PrimitiveValue]]: "ss"
```

4.1 字符串的解构赋值

```
let str = "abcdefg";
let [x,y,z,m,n,p,q] = str;
console.log(x,y,z,m,n,p,q);//a b c d e f g

let {length} = str;
console.log(length);//7
//原理：    是将字符串转为一个类数组对象，然后进行赋值操作
```

4.2 字符串中的一些新方法

4.2 .1 `includes(str,[index])`

```
let str="abacdaefgh";
//indexOf()
let return_indexOf =str.indexOf("a");
console.log(return_indexOf);//0

//includes("")
let return_includes = str.includes("a");
```

```
    console.log(return_includes);//true
    //指定从索引6开始查找
    let return_includes_index = str.includes("a",6);
    console.log(return_includes_index);//false
```

4.2.2 `startsWith(str,[index])` & `endsWith(str)`

```
    let str = "abacdaefgh";
    //startsWith()
    let return_startsWith = str.startsWith("a");
    let return_startsWith_index = str.startsWith("a",6);

    console.log(return_startsWith);//true
    console.log(return_startsWith_index);//false

    //endsWith()
    let return_endsWith = str.endsWith("a");
    console.log(return_endsWith);//false
```

4.2.3 `repeat(number)`

```
        let str = "abc";
        //repeat(number)
        let return_repeat = str.repeat(2);
        console.log(str,return_repeat);//abc abcabc

        //注意 这里的 repeat(number)方法中的number取值是向下取整的
        // 1) 大于1的数
        let Greater_than_1 = str.repeat(1.999);
        console.log(Greater_than_1);//abc

        // 2) 大于0小于1的数
        let Greater_than_0 = str.repeat(0);
        let Greater_than_0_is_less_than_1 = str.repeat(0.44);
        //返回一个空字符串
        console.log(Greater_than_0);// ""
        console.log(Greater_than_0_is_less_than_1);// ""

        // 3) 大于-1小于0的数(负小数Negative decima)
        let Negative_decima = str.repeat(-0.1);
        console.log(Negative_decima);//""

        // 4) 小于-1的数  Negative_number_less_then_fu1
        // 报错
        //let Negative_number_less_then_fu1 = str.repeat(-2);
        //console.log(Negative_number_less_then_fu1);
        //Uncaught RangeError: Invalid count value at String.repeat

```

```
        // 5) NaN
        // NaN转为0 ，返回一个空字符串
        let repeat_with_NaN = str.repeat(NaN);
        console.log(repeat_with_NaN);// ""

        // 6） Infinity
        //报错
        //let repeat_with_Infinity = str.repeat(Infinity);
        //Uncaught RangeError: Invalid count value at String.repeat
        //console.log(repeat_with_Infinity);// ""

        // 7) String with number
        // NaN转为0 ，返回一个空字符串
        let String_number = str.repeat("2");
        console.log(String_number);// "abcabc"

        let String_with_number = str.repeat("2ab");
        console.log(String_with_number);// ""
        ...

```

4.2.4 ES7草案-补白

```
    //API
    //str.padStart(len,content);
    //str:原字符串
    //len:补白后的字符串
    //content: 补白的内容
    let str = '1';
    let return_str = str.padStart(2,'0');
    console.log(str);//1
    console.log(return_str);//01
    console.log(str == return_str);//fasle

    console.log(str.padStart(20,'刘凯'));
    //刘凯刘凯刘凯刘凯刘凯刘凯刘凯刘凯刘凯刘凯1

    //后补白 padEnd(len,content)
```

4.3 模板字符串

```
    //1. `这里写html代码段`
    let str = `<h1>hello ES6</h1>`;
    document.body.innerHTML = str;

    //2.使用变量和js解析运算
```

```
    let person = {
        pname:"BigSpinach",
        age: 25,
        sex:1
    };

    let str2 = ` <ul>
        <li>${person.pname}</li>
        <li>${person.age}</li>
        <li>${person.sex==1?"男":"女"}</li>
    </ul>`
    document.body.innerHTML += str2;

    //3.使用函数
    function fn1(a){
        return a*=3;
    }
    console.log(`${fn1(3)}`);//9------3*3

    //
    let str3 = "return" +"`Hello ${a}`"  ;
    let fun = new Function('a',str3);
    console.log(fun("BigSpinach"));//Hello BigSpinach
```

4.4 标签模板

> 1怎么用?
> 函数名`形参·
> 2 在哪里用?
>
> - 在过滤字符串的时候，防止XSS攻击
> - 2.处理多语言转换

```
        let user = {
            uname:'liu',
            age:25,
            sex:1
        }

        function abc(s,a,b) {
            console.log(s,a,b);
        }

        //调用abc函数
        abc`i am ${user.uname},i am ${user.age}`;
        //["i am ", ",i am ", "", raw: Array(3)] "liu" 25
```

使用标签模板

```
        let user = {
            uname:'liu',
            age:25,
            sex:1
        }

        function abc(s,a,b) {
            //console.log(s,a,b);
            return s+a+b;
        }

        //abc`i am ${user.uname},i am ${user.age}`;
        let return_abc = abc`i am ${user.uname},i am ${user.age}`;
        console.log(return_abc);//i am ,,i am ,liu25
```

5 .数组扩展

5.1 Array类上的方法

```
    //1.问题引入
    //Array(),将传递进来的值变为数组返回
    let return_ary = Array(1,3,5,7,9);
    console.log(return_ary);//Array(5)

    //注意
    //当Array(number)中只传递一个参数的时候，代表返回数组的长度
    let return_length_ary = Array(4);
    console.log(return_length_ary);//(4) [empty × 4]

    //2.ES6中Array类上的方法 ----of()
    //为了解决想要生成一个单独的 [4],这样类似的数组，ES6的新方法 Array.of(numbe
r)
    //Array.of(number)
    let return_alone_ary = Array.of(3);
    console.log(return_alone_ary);//[3]

    //3.ES6中Array类上的方法 ----from(val)
    //  val是数组：克隆一份一模一样的数组返回,原数组不变
    //  val是类数字：转为数组

    //3.1  val是数组
    let ary = [1,2,3,4,5,6];
    let return_aryFrom = Array.from(ary);
```

```
    console.log(return_aryFrom,ary);//(6) [1, 2, 3, 4, 5, 6] (6) [1, 2, 3, 4, 5, 6]

    //3.1  val是类数组
    function toArray(){
        //之前的写法
        //return Array.prototype.slice.call(arguments);

        //ES6的写法
        return Array.from(arguments);
    }

    console.log(toArray(1,2,3));// [1, 2, 3]
```

5.2 Array原型上的方法

`copyWithin(target,[start,end])`

```
        //1.复制替换数组，生成新数组
        //copyWithin(target,[start,end])
        // target:被替换的目标索引位置
        // start：默认为0，end（默认为数组的length-1）指的是数据从start位置开始，到end结束

        //test1
        let ary = [1,2,3,4,5,6,7,8];
        let return_conpWitnin = ary.copyWithin(0);
        console.log(return_conpWitnin,ary);//(8) [1, 2, 3, 4, 5, 6, 7, 8] (8) [1, 2, 3, 4, 5, 6, 7, 8]

        //test2
        let ary2 = [1,2,3,4,5,6,7,8];
        let return_conpWitnin2 = ary.copyWithin(0,1,3);
        console.log(return_conpWitnin2,ary2);//(8) [2, 3, 3, 4, 5, 6, 7, 8]
        //过程
        //原数据    [1, 2, 3, 4, 5, 6 , 7, 8];
        //索引          0  1  2  3  4  5   6  7
        //
        //                [1,3)---------也就是所因为1和2的数据
        //                [2,3]
        //step1)       读取[1,3)位置的数据 ===> [2,3]
        //step2)       开始从索引0位置开始替换，替换的长度就是step1读取到的数据的长度，
        //
        //原数据    [1, 2, 3, 4, 5, 6 , 7, 8];
        //索引          0  1  2  3  4  5   6  7
        //          [2, 3]             后边的补上3, 4, 5, 6 , 7, 8];
        //          new [2, 3,3, 4, 5, 6 , 7, 8];
```

`find(callback);`

```
        //2.查找元素
        //find(callback);
        //find先 遍历ary，一项一项执行，一旦函数返回true，就不继续向下执行遍历了，反悔哦当前项
        //  如果一直到遍历结束函数都没有返回true，那么find返回undefined
        //  callback(item,index,input)
        //  item:当前项
        //  index：索引
        //  input：原数组
        //
        let ary = [1,2,3,5,6,7,89,10];
        let return_find = ary.find(function(item,index,input){
            console.log(item);//1
            console.log(index);//0
            console.log(input);//[1,2,3,5,6,7,89,10];
            return true;

        });
        console.log(return_find);//1

        //
        //callback中一直没有返回true
        let return_find2 = ary.find(function(item,index,input){
            //...
        });
        console.log(return_find2);//undefined


        //
        //正常使用
        let return_find3 = ary.find(function(item,index,input){
            return item==3;
        });
        console.log(return_find3);//3
```

`findIndex(callback)`

```
        //3.查找索引
        //findIndex()
        //返回当前项的索引，如果没找到，返回-1
        let ary = [1,2,3,5,6,7,89,10];
        let return_findIndex = ary.findIndex(function(item,index,input){
            return item==1;
        });
        console.log(return_findIndex);//0
```

`fill(value,[start,end]);`

```
        //4.填充数组
        //原数组会被修改
        //fill(value,[start,end]);
        let ary = [1,2,3,5,6,7,89,10];
        let return_fill = ary.fill("刘");
        console.log(return_fill);// ["刘", "刘", "刘", "刘", "刘", "刘", "刘", "刘"]
        console.log(ary==return_fill);//true

        let return_fill2 = ary.fill("kai",5,ary.lengyh-1);
        console.log(return_fill2);
```

`includes(item)`

```
    //5.判断数组中是否包含某一项
    //includes(item)
    let ary = [1,2,3,5,6,7,89,10];
    let return_includes = ary.includes(2);
    console.log(return_includes);//true
```

5.3 数组的空位

```
    //1.
        //数组中的空位
        //在ES5及其之前，数组对空位的处理都没有固定的说法，一般都是跳过
        //例如
        let ary1 = Array(3);
        console.log(ary1);//[empty × 3]

        let ary2 = [undefined,undefined];
        console.log(ary2);//[undefined, undefined]

        //
        //使用ES5之前的方法 forEach 遍历数组中的每一项
        let ary3 = [1,,,,,3,,2,,,5];
        console.log("ary3.length="+ary3.length);//ary3.length=11
        ary1.forEach( function(element, index) {
            console.log(element,index);
        });
        //由于都为Empty ，所以都跳过，没有输出


        ary3.forEach( function(element, index) {
            console.log(element,index);
        });
```

```
        //结果只会遍历有元素的部分，对于没有元素的部分直接跳过
        //1 0
        //3 5
        //2 7
        //5 10


        console.log('-------------------------');
        //ES6之后的方法会将空位处理 undefined
        //例如
        //find

        ary3.find(function(item){
            console.log(item);
        });
        //1
        //undefined*4
        //3
        //undefined
        //2
        //undefined
        //5

        //map()返回一个新的数组
        let return_mapArray = ary3.map(function(index, elem) {
            return elem;
        });
        console.log(return_mapArray);//(11) [0, empty × 4, 5, empty, 7, empty × 2, 10]
```

5.4 数组的遍历

```
    let ary = [1,3,4,5,6,7,8,9,"10"];
    //1.
    //for...in...
    console.log('------------for...in遍历----------');
    for(let key in ary){
        console.log(key);
    };
    //0,1,2,3,4,5,6,7,8
    //遍历的结果是数组中的每一个索引
    //要想得到数组中的每一项的值需要使用  ary[key];


    //2.
    //for...of...
    console.log('----------for..of遍历-------------');
    //2.1遍历数组中每一项的值
    for(let val of ary){
```

```
        console.log(val);
    };
    //1,3,4,5,6,7,8,9,"10"
    //遍历的结果是数组中的每项的值
    //如果
    //非得得到数组的每一项索引
    //使用数组的扩展方法 ary.keys()

    //2.2遍历数组中每一项的索引
    for(let key of ary.keys()){
        console.log(key);
    };

    //2.3遍历数组的每一项的值和对应的索引
    //ary.entries()
    for(let [index,item] of ary.entries()){
        console.log([index,item]);
    };
    //(2) [0, 1]
    //(2) [1, 3]
    //...
    //...
    //(2) [8, "10"]

```

6.函数扩展

6.1形参默认值的问题

6.1.1 默认值生效的时机

```
        //1.
        // 形参默认值生效的时机
        //当且仅当：函数执行不传递实参的时候，默认值才生效
        function fn1(pname="BigSpinac",age=25){
            console.log(pname,age);
        }
        fn1();//BigSpinac 25
        fn1("刘凯",250);//刘凯 250

```

6.1.2 两种书写默认值方式的异同

```
        //2.
```

```
        //形参默认值的两种书写方式
        function fn1 ({pname="BigSpinac",age=0}={}){
            console.log(pname,age);
        }

        function fn2 ({pname,age}={pname:"BigSpinac",age:0}){
            console.log(pname,age);
        }

        //不传实参的情况下，两种结果是一样的
        fn1();//BigSpinac 0
        fn2();//BigSpinac 0

        //传递实参
        fn1({pname:"张三",age:12});//张三 12
        fn2({pname:"李四",age:25});//李四 25
        //查看默认值
        fn1({pname:"张三六六六"});//张三六六六 0
        fn2({pname:"李四四十四"});//李四四十四 undefined
        //由此得出结论
        //函数的形参默认值的识别原理是：
        //fn1方法中的  {pname="BigSpinac",age=0} = {pname:"张三六六六"}
        //fn2方法中的  {pname,age}={pname:"李四",age:25}
        //然后
        //根据对象的解构赋值原则进行赋值操作
```

6.1.3 一般默认值的书写在后方

```
        //3.
        //一般函数的形参默认值都会放在最后边进行
        function fn(a,b,c=0){
            console.log(a,b,c);
        }
        fn(1,2,3);//1,2,3
        fn(10,20);//10 20 0

        //不放在后边
        function fn2(x,y=3,z){
            console.log(x,y,z);
        }

        fn2();//undefined 3 undefined
        fn2(99,88,33);//99 88 33
```

6.2 参数集合

6.2.1 实参的长度

```
//使用arguments的方式查看函数参数的个数
function fn(a,b,c){
    console.log(arguments.length);
}
fn();//0
fn(1,2);//2
```

6.2.2 形参的长度

> 函数的length属性是说
> 1.在函数没有给定参数默认值的情况下，length表示的是函数参数的长度(个数)
> 2.在函数的参数给定默认值的情况下，length表示的是当前默认值参数的位置（索引）

```
//函数的length属性---一般情况下 表示函数参数的长度（个数）
function fn2(a,b,c){}
console.log(fn2.length);//3
```

但是

```
//但是。注意。。
//当给形参加上默认值的时候，
function fn3(x,y,z=0){}
console.log(fn3.length);//2

function fn4(x,y=1,z){}
console.log(fn4.length);//1

function fn5(x=10,y,z=0){}
console.log(fn5.length);//0
//这是为什么呢？
//原因。。是
//函数的length属性是说
//1.在函数没有给定参数默认值的情况下，length表示的是函数参数的长度(个数)
//2.在函数的参数给定默认值的情况下，length表示的是当前默认值参数的位置（索引）
```

6.3 参数作用域的问题

形参赋值跟函数{}内的作用域是一个同的作用域

```
let n=0,m=1;
function fn(x=n,y=m) {
```

```
        let n=999,m=666;
        console.log(x,y);
        //console.log(n,m);

    }
    fn();//0 1
    //形参赋值跟函数{}内的作用域是一个同的作用域
```

举例验证

> 形参赋值 形参的作用域跟函数的作用域是同一个作用域，相当于let定义了两次x,所以报错

```
    let x=10;
    function fn2(x=1,y=x){
        let x = 10000000;
        console.log(x,y);
    }
    fn2();//03-参数的作用域问题.html:28 Uncaught SyntaxError: Identifier 'x' has already been declared

    //报错的原因:
    //形参赋值 形参的作用域跟函数的作用域是同一个作用域，相当于let定义了两次x,所以报错
    //在函数fn2的函数作用域内
    //第一步：形参赋值  let x=1 ,y=undefined
    //第二步:       y=x
    //              y=1

```

6.4 扩展(展开)运算符 ...

> 作用
> 1.可以将[]转为非数组
> 2.也可以将非数组转为数组

6.4.1 展开数组中的每一项

```
     //将数组中的每一项都输出
      let ary = [1,2,3,"xxx"];
      console.log(...ary);//1 2 3 "xxx"
```

6.4.2 展开字符串中 的每一个字符（包括每一个空格）

```
    //将非数组（字符串）转为一个数组
    let str ="BigSp     inach";
    let return_Ary = [...str];
    console.log(return_Ary);
    //(15) ["B", "i", "g", "S", "p", " ", " ", " ", " ", " ", "i", "n", "a", "c", "h"]
    console.log(...str);//B i g S p           i n a c h
```

6.4.3 `arguments`的展开运算符

```
//arguments的展开运算符
 function fn(a){
        console.log(...arguments);
        console.log([...arguments]);
        console.log({...arguments});
    }
    fn("a");
    //a
    //["a"]
    //{0: "a"}
```

6.4.4 展开运算符的应用

`1.------求最大值`

```
    //求一个数组中的最大值
    let ary = [1,2,3,4,5,6,25,7,88,8,89,12,25,2];
    //之前的思路
    //拼接一个 "Math.max("+1,23,4,56,7+"")"的字符串出来，然后使用eval方法将这个字符串的函数表达式执行
    //或者使用Math.max.apply(null,ary)
    let maxNum = Math.max.apply(null,ary);
    console.log(maxNum);

    //知识点巩固
    //apply方法
    //Function.apply(obj,[args])
    //apply这个方法的作用是，将执行函数的this改变，并且将args这个数组形里的每一项作为函数的参数传递给函数
    //解析一下
    //大概是这样子
    //Function.apply(obj,[args])
    //args=[1,2,3]
    //          | | |
    //          | | |
    //          V V V
    //  Function(1,2,3)
```

```
21.
22.     //现在使用展开运算符，就更简单了
23.     let maxNum_daindiandioan = Math.max(...ary);
24.     console.log(maxNum_daindiandioan);//89
```

2.数组的拼接

```
1.     //数组的拼接
2.     //之前使用concat方法
3.     let ary1 = [1,23,4];
4.     let ary2 = ["a","b","liukai"];
5.     let return_concat = ary1.concat(ary2);
6.     console.log(return_concat);//(6) [1, 23, 4, "a", "b", "liukai"]
7.
8.     //现在
9.     let return_ary_diandiandian = [...ary1,...ary2];
10.    console.log(return_ary_diandiandian);//(6) [1, 23, 4, "a", "b", "li
    ukai"]
```

6.5函数的name属性

1.声明式函数的`name`属性

```
1. //1.声明式函数的name属性
2.     let fn = function(){}
3.     console.log(fn.name);//fn
4.
5.     function fn2() {}
6.     console.log(fn2.name);//fn2
```

2.自执行函数的`name`属性

```
1. //2.自执行函数的name属性
2.     console.log(function(){}.name)// 输出一个空字符串
```

3.bind方法得到的函数的name属性

```
1. //3.bind方法得到的函数的name属性
2.     let obj={};
3.     let fn3 = fn.bind(obj);
4.     console.log(fn.name);//fn
5.     console.log(fn3.name);//bound fn
```

4.构造函数式的name属性

```
//4.构造函数式的name属性
    let fn4 = new Function('n',"return n*n");
    console.log(fn4.name);//anonymous-------adj. 匿名的，无名的；无个性特征的
```

6.6箭头函数

6.6.1 箭头函数的书写规则

```
    //ES5中函数的写法
    function fn (a) {
        a=a||1;
        let b = 1;
        return  a+b;
    }
    let return_fn_es5 = fn(2);
    console.log(return_fn_es5);//3

    //ES6中的箭头函数
    let fn2 = (a) => {
        a=a||1;
        let b = 1;
        return  a+b;
    }
    let return_fn_es6 = fn2(2);
    console.log(return_fn_es6);//3

    //(a),这个括号可以省略不写
    //如果箭头函数体内只有一个return的话
    //那个{}也可以省略不写
    //例如
    let fn3=x=>x*x;
    let return_fn_es6_simple = fn3(2) ;
    console.log(return_fn_es6_simple);//4
```

6.6.2 箭头函数中`this`指向问题

> 箭头函数中的this的问题
> 箭头函数是没有this指向的，要知道他的this是谁，记住一句话
> `箭头函数的上一级作用域是谁，那么他的this就是谁`

7.对象扩展

7.1 简洁表示法&属性表达式

7.1.1 简洁表示法

```
//1.简洁表示法
let a = 1;
let b = 2;
let es5_obj = {
    a:a,
    b:b,
    fn : function(){
        console.log(this.name);
    }
};
let es6_obj = {
    a,
    b,
    fn(){
        console.log(this.name);
    }
};
console.log(es5_obj,es6_obj);
//{a: 1, b: 2, fn: ƒ} {a: 1, b: 2, fn: ƒ}

```

7.1.2 属性表达式

> 属性可以使用`[变量名]`的方式来定义

```
//2.属性表达式
let a = 'aaa';
let b = 'bbb';
let obj = {
    [a]:"aaaaaaa",
    [b]:'bbbbbbb'
}
console.log(obj);
//{aaa: "aaaaaaa", bbb: "bbbbbbb"}

```

```
    let obj={
        name:"BigSpinach",
        age:25
    }
    //ES5中属性的获取和操作
```

```
    console.log(obj.name);//BigSpinach
    console.log(obj['age']);//25
    //增
    obj.sex = "男孩纸";
    console.log(obj);//{name: "BigSpinach", age: 25, sex: "男孩纸"}
    //删
    delete(obj.name);
    console.log(obj);//{age: 25, sex: "男孩纸"}
    //改
    obj['age'] = 250;
    console.log(obj);//{age: 250, sex: "男孩纸"}
```

```
//ES6中属性的获取和操作
    let name = "liukai";
    let age =25;
    let obj={
        [name] :"刘凯",
        [age+name]:"25liukai",
        fn() {
            console.log(this.liukai);
        }
    }
    //也就是说ES6中可以通过[变量]的方式定义属性了
    console.log(obj.name);//undefined?
    console.log(obj);//{liukai: "刘凯", 25liukai: "25liukai", fn: ƒ}
```

7.2 Object类上的新增API

```
▼ƒ Object()
    arguments: (...)
  ▶ assign: ƒ assign()
    caller: (...)
  ▶ create: ƒ create()
  ▶ defineProperties: ƒ defineProperties()
  ▶ defineProperty: ƒ defineProperty()
  ▶ entries: ƒ entries()
  ▶ freeze: ƒ freeze()
  ▶ getOwnPropertyDescriptor: ƒ getOwnPropertyDescriptor()
  ▶ getOwnPropertyDescriptors: ƒ getOwnPropertyDescriptors()
  ▶ getOwnPropertyNames: ƒ getOwnPropertyNames()
  ▶ getOwnPropertySymbols: ƒ getOwnPropertySymbols()
  ▶ getPrototypeOf: ƒ getPrototypeOf()
  ▶ is: ƒ is()
  ▶ isExtensible: ƒ isExtensible()
  ▶ isFrozen: ƒ isFrozen()
  ▶ isSealed: ƒ isSealed()
  ▶ keys: ƒ keys()
    length: 1
    name: "Object"
  ▶ preventExtensions: ƒ preventExtensions()
  ▶ prototype: {constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __look…
  ▶ seal: ƒ seal()
  ▶ setPrototypeOf: ƒ setPrototypeOf()
  ▶ values: ƒ values()
  ▶ __proto__: ƒ ()
  ▶ [[Scopes]]: Scopes[0]
```

7.2.1 Object.is(a,b)

Object.is()

```
//1. Object.is(a,b)
    //判断a===b,返回一个Boolean值
    let a = 10;
    let b = "10";
    let return_Object_is = Object.is(a,b);
    console.log(return_Object_is);//false

    //ES5有一个问题是 NaN===NaN永远返回的是false
    console.log( NaN===NaN);//false
    //现在用
    console.log( Object.is(NaN,NaN));//true
```

Object.is()的实际意义

```
//2.Object.is方法的实际意义，实现对数组中多个NaN的值的去重
        let arr= [NaN,1,2,3,NaN,NaN,NaN,NaN,NaN,45];

        //2.1 使用数组的includes方法判断数组中是否包含NaN
        console.log(arr.includes(NaN));//ture

        //************************
        noRepetition=[];
        arr.forEach((item)=>{
            let flag = true;
            for(let i=0;i<noRepetition.length;i++){
                if(Object.is(item,noRepetition[i])){
                    flag = false;
                }
            }
            if(flag){
                noRepetition.push(item);
            }
            return noRepetition;
        });
        console.log(noRepetition);

    /*
        function noRepetitionArray(arr){
            let noRepetition = [];
            for(let i=0;i<arr.length;i++){
                let cur = arr[i];
                var flag =true;
                for (let j = 0; j < noRepetition.length; j++) {
                    if(Object.is(cur,noRepetition[j])){
                        flag=false;
                    }
```

```
                }

                if(flag){
                        noRepetition.push(cur);
                }
            }
            return noRepetition;
        }
    */
        //console.log(noRepetition);
        //let return_arr = noRepetitionArray(arr);
        //console.log(return_arr);//[NaN, 1, 2, 3, 45]

```

7.2.2 Object.assign(obj1,obj2)

> 将obj1和obj2进行合并，返回obj1

```
        //Object.assign(obj1,obj2)------将obj1和obj2进行合并，返回obj1
        //assign
        //vt. 分配；指派；[计][数] 赋值
        //vi. 将财产过户（尤指过户给债权人）
        let obj1 = {a:1};
        let obj2 = {a:2,b:2};
        let return_assign = Object.assign(obj1,obj2);
        console.log(return_assign);//{a: 2, b: 2}
        console.log(obj1);//{a: 2, b: 2}
        console.log(obj2);//{a: 2, b: 2}
        console.log(obj1===return_assign);//true
        console.log(obj2===return_assign);//false
```

7.2.3 Object.getOwnPropertyDescriptor(obj,attr)

> 接收两个参数
> obj 要查看的对象
> attr 要查看的对象的属性
> 查看某一个对象的属性的描述

```
    //3.Object.getOwnPropertyDescriptor()
    //查看某一个对象的属性的描述
    let str= "abc";
    //console.log(str.big());//<big>abc</big>

    //console.dir(str.__proto__);

```

```
    //console.log(Object.getOwnPropertyDescriptor(str,length);//
    //{value: "a", writable: false, enumerable: true, configurable: false}
    //getOwnPropertyDescriptors
    console.log(Object.getOwnPropertyDescriptors(str));
    /*
            {
                0: {value: "a", writable: false, enumerable: true, configurable: false}
                1: {value: "b", writable: false, enumerable: true, configurable: false}
                2: {value: "c", writable: false, enumerable: true, configurable: false}
                length: {value: 3, writable: false, enumerable: false, configurable: false}
                __proto__: Object
            }
        */
    let obj={name:"liu",age:25};
    console.log(Object.getOwnPropertyDescriptor(obj));//undefined
    console.log(Object.getOwnPropertyDescriptors(obj));//{name: {…}, age: {…}}
```

7.2.4 Object.create(propObj,[propertiesObject])

> propObj 对象，这个对象错位新创建的对象的原型，
> propertiesObject 可选，指定若干个属性作为新创建的对象的原型上的属性

Object.create()的深入研究

原型引入----1.不指明类的原型，默认浏览器会自动为类创建一个构造函数指向类本身

```
    var obj = {
        getX : function(){}
    }

    function Fn(){

    }

    var f = new Fn();
    console.log(f.constructor);
    //指向当前类本身---------这个是浏览器自己给创建的
    /*
        f Fn(){

        }
     */
```

原型引入----2.指明类原型为一个堆内存，类的实例和类本身的constructor通过原型链查找机制找到Object

```
    console.log('--------不使用浏览器自动创建的constructor---------');
    var obj2 = {
        getX : function(){}
    }

    function Fn2(){

    }
    //指定一个对象（堆内存）作为类的原型
    Fn2.prototype = obj2;

    var f2 = new Fn2();
    console.log(f2.constructor);
    /*
        ƒ Object() { [native code] }
    */
    //原理：
    //1.f2先找obj这个对象的堆内存下找constructor属性，
    //  没找到
    //  然后
    //2.继续向obj的上一级查找，找到了Object的头上
    //  所以
    //  输出 ƒ Object() { [native code] }
```

原型引入----3.指定明确的对象作为类的原型，并为该对象指定明确的构造函数

```
console.log('---指定明确的对象作为类的原型，并为该对象指定明确的构造函数--------');
    //现在不想让指定prototype的类的construtor属性不要指向Object
    //这时候
    //就需要
    //在 要指定的对象的堆内存中明确告知，此obj的constructor是谁
    //一般这样写 对象.constructor = 指定类名（实例化对象的类名）
    //obj.constructor = Fn
    var obj = {
        getX : function(){}
    }

    function Fn(){

    }
    //指定一个对象（堆内存）作为类的原型
    Fn.prototype = obj;
    //指定obj对象的构造函数是Fn这个类
    obj.constructor = Fn;

    var f = new Fn();
```

```
    console.log(f.constructor);
    /*
    ƒ Fn(){

    }

     */

```

克隆对象-`for...in`

```
    console.log('---------克隆对象之for...in------');
    let obj={myname:"刘凯",myage:25};
    //function copyObj (o) {
        let copy_obj = {};
        for(let key in obj){
            copy_obj[key] = obj[key];
        }
        //return copy_obj;
    //}
    console.log(copy_obj);
    //{myname: "刘凯", myage: 25}

    obj.__proto__.getMyName = function () {};
    console.log(copy_obj);
    //{myname: "刘凯", myage: 25}
    obj['sex']=1;
    console.log(copy_obj);
    //{myname: "刘凯", myage: 25}
```

克隆对象-`Object.create()`

```
    console.log('---------克隆对象之Object.create()------');
    let obj={myname:"刘凯",myage:25};
    let copy_obj = Object.create(obj);

    console.log(copy_obj);
    /*
    {}
        __proto__:
             myage: 25
             myname: "刘凯"
             sex: 1
             __proto__: Object

     */

    //Object.create()方法克隆的到的对象与之前for...in...遍历的到的对象有何不同
```

```
    //1.Object.create()方法创建得到的对象，跟obj建立了动态的连接关系（通过__proto__建立上的连接关系）
    //2. 以后再给obj增加属性和方法的时候，通过Object.create()方法得到的对象依旧可以获取到新增加（删除）的方法，而for...in...不行

    obj.__proto__.getMyName = function () {};
    console.log(copy_obj);
    /*
    {}
        __proto__:
            myage: 25
            myname: "刘凯"
            sex: 1
            __proto__: Object

     */
    obj['sex']=1;
    console.log(copy_obj);
    /*
    {}
        __proto__:
            myage: 25
            myname: "刘凯"
            sex: 1
            __proto__: Object

     */
```

Object.create()方法的原理

```
console.log('--------自定义方法模拟Object.create()方法----------');
    //其实也就是Object.create()方法的实现原理
    //原理：
    //方法 返回一个指定对象作为新返回对象的原型
    //       原型喽， 给类指定对象作为它的原型
    //在不指定constructor的情况下，浏览器默认会自动创建一个constructor作为该类的构造函数，指向该类本身
    function myObjectCreate (obj) {
        function Fn () {};
        Fn.prototype = obj;
        return new Fn();
    }

    let o={myname:"刘凯",myage:25};
    console.log(myObjectCreate(o));
    /*
    Fn {}
        __proto__:
            myage: 25
            myname: "刘凯"
            __proto__: Object
```

```
21.      */
```

7.2.5 Object.keys();

> 对象属性的遍历

```
1.      let obj={name:"liu",age:25};
2.
3.      console.dir(obj.prototype);//undefined
4.      obj.__proto__.sayHi=function(){
5.          console.log("嗨....");
6.      }
7.
8.      //1. for...in...
9.      for(let key in obj){
10.         console.log(obj[key]);
11.         /*
12.             liu
13.             25
14.             ƒ (){
15.                 console.log("嗨....");
16.             }
17.         */
18.     }
19.
20.     //2.Object.keys();
21.     //返回一个数组，包括对象自己的所有可枚举(私有的)属性
22.     console.log(Object.keys(obj));// ["name", "age"]
23.
24.     //3.Object.getOwnPropertyNames()
25.     //返回一个数组，包括对象的自己的私有属性的所有属性（包括不可枚举的属性）
26.     console.log(Object.getOwnPropertyNames(obj));//  ["name", "age"]
27.
```

7.3 对象的扩展运算符

```
1.      //对象的扩展运算符
2.      let {a,b,...c}={a:"aaa",b:"bbb",c:"ccc",d:"ddd",e:"eee"};
3.      console.log(a,b,c);
4.      //aaa bbb {c: "ccc", d: "ddd", e: "eee"}
5.
```

8.Set和Map数据结构

8.1 set数据结构的数据的擦混关键方式

> let set1 = new Set(arr);
> 参数是一个数组
> 返回值是数组去重后的一个Set实例（类数组）

```
//set数据结构的创建方式
    //1.使用构造函数的方式创建
    let set1 = new Set([1,2,3,4,5,NaN,2,NaN,2,3,4,5]);
    console.log(set1);//Set(6) {1, 2, 3, 4, 5, …}
    /*
    Set(6)
        size: 6
         __proto__: Set
         [[Entries]]: Array(6)
         0: 1
         1: 2
         2: 3
         3: 4
         4: 5
         5: NaN
         length: 6
     */
    //参数是一个数组
    //返回值是数组去重后的一个Set实例（类数组）

    //将set实例转为数组
    console.log([...set1]);//(6) [1, 2, 3, 4, 5, NaN]
    //或者
    let return_array_from = Array.from(set1);
    console.log(return_array_from);//(6) [1, 2, 3, 4, 5, NaN]
```

8.2 set数据结构的一些方法

8.2.1 set.add(n)

> 向set实例中增加一个n，这个n可以是任意数据类型的数据

```
//2.set数据结构的一些方法
    let arr=[12,3,4,5];
```

```
    let set =new Set(arr) ;
    //2.1set.add(n)
    //n可以是任意数据类型的值
    set.add("哈哈哈");
    console.log(set);// Set(5) {12, 3, 4, 5, "哈哈哈"}
    set.add(100);
    console.log(set);// Set(5) {12, 3, 4, 5, "哈哈哈",100}
    set.add(arr);
    console.log(set);
    // Set(5) {12, 3, 4, 5, "哈哈哈",100,Array(4)}
    set.add({name:'liu' ,age:25});
    console.log(set);
    // Set(5) {12, 3, 4, 5, "哈哈哈",100,Array(4),Object }
    set.add(set);
    console.log(set);
    // Set(5) {12, 3, 4, 5, "哈哈哈",100,Array(4),Object ,Set(9)}

```

8.2.2 set.has(x);

> 判断set实例中是否含有x这一项，返回一个boolean值

```
let arr=[12,3,4,5];
let set =new Set(arr) ;
console.log(set.has(1));//false
console.log(set.has(12));//true
console.log(set.has(arr));//true
console.log(set.has(set));//true
```

8.2.3 set.delete(x)

> //删除set实例中的某一项x
> //返回值Boolean
> //原set会发生改变

```
let arr=[12,3,4,5];
let set =new Set(arr) ;
//2.3 set.delete(x)
//删除set实例中的某一项x
//返回值Boolean
//原set会发生改变
let return_set_delete = set.delete("哈哈哈");
console.log(return_set_delete);//ture
console.log(set);//Set(8) {12, 3, 4, 5, 100, …}
```

8.2.4 set.clera();

> 没有返回值——undefined
> 清空set中的所有数据

```
//2.4 set.clera();
//没有返回值------undefined
//清空set中的所有数据
let return_set_clear = set.clear();
console.log(return_set_clear);//undefined
console.log(set);//Set(0) {}
```

8.3 set数据的一些使用

```
let arr1 = [1,2,34,5,,6];
        let arr2 = [2,7,,,8,NaN,NaN];

        //并集
        console.log([...arr1,...arr2]);
        //(13) [1, 2, 34, 5, undefined, 6, 2, 7, undefined, undefined, 8, NaN, NaN]
        console.log(new Set([...arr1,...arr2]));
        //Set(9) {1, 2, 34, 5, undefined, …,6,7,8,NaN}


        //交集
        //filter()把传入的函数依次作用于每个元素，然后根据返回值是true还是false决定保留还是丢弃该元素。
        //返回值是true ，表示不保留
        let return_arr_filter = arr1.filter((item)=> {
            return arr2.includes(item);
        });
        console.log(arr1,arr2);
        //[1, 2, 34, 5, empty, 6]
        //[2, 7, empty × 2, 8, NaN, NaN]
        console.log(return_arr_filter);
        //[2]

        //差集=并集-交集
        //let rentun_new_set = new Set([...arr1,...arr2]);
        //console.log(rentun_new_set);//Set(9) {1, 2, 34, 5, undefined, …}


        let chaji = [...arr1,...arr2].filter((item)=>{
            return !(arr2.includes(item));
```

```
        });
        console.log(chaji);//[1, 34, 5, 6]
```

8.# array.filter()深入研究

```
    //Array.filter();
    let arr = [1,3,5,"aaa"];
    let return_arr_filter = arr.filter(function(){
        //console.log(arguments);
        //Arguments(3) [1, 0, Array(4), callee: ƒ, Symbol(Symbol.iterator): ƒ]
        //Arguments(3) [3, 1, Array(4), callee: ƒ, Symbol(Symbol.iterator): ƒ]
        //Arguments(3) [5, 2, Array(4), callee: ƒ, Symbol(Symbol.iterator): ƒ]
        //Arguments(3) ["aaa", 3, Array(4), callee: ƒ, Symbol(Symbol.iterator): ƒ]

        //arguments的解读
        //（item,index,input）
        //item 当前项
        //index 当前项的索引
        //input 当前过滤的数组

        //返回值
        //return false;
        return true;
    });
    //回调函数执行几次取决于 调用filter函数的那个数组中有多少项 arr.length
    //
    //return true/flase
    //false 说明过滤掉，从数组中删除
    console.log(return_arr_filter);//[]
    //true 不过滤
    console.log(return_arr_filter);//(4) [1, 3, 5, "aaa"]

    //返回一个新数组
    console.log(return_arr_filter===arr);//false
```

8.4 Map数据结构类型的数据

8.4.1 创建方式

> 1.使用构造函数方式创建一个实例 `let map1=new Map([[1,"a"],...]);`
> 2.参数是个数组, 并且 数组的每一项都是一个数组,这个数组有两项,第一项作为键key,第二项

作为值value
3.这里的key键可以是任意数据类型的

```
 let map1=new Map([[1,"a"],["a","A"],[{name:"liukai"},"250"],[/\d+/,"正则"]]);
 console.log(map1);
 /*
    Map(5)
        size: (...)
        __proto__: Map
        [[Entries]]: Array(5)
        0: {1 => "a"}
        1: {"a" => "A"}
        2: {Object => "250"}
        3: {/\d+/ => "正则"}
        4: {2 => "JS"}
        length: 5
  */

```

8.4.2 方法

```
//方法
     let map1=new Map([[1,"a"],["a","A"],[{name:"liukai"},"250"],[/\d+/,"正则"]]);
    //get(key)
    console.log(map1.get("a"));//A

    //set(key,value);
    map1.set(2,"JS");//
    console.log(map1);//Map(5)
    //delete,has,clear

    let ary=[1,2,3,4,5,6];//
    //将数组变成Map
    //1,[1]
    //2,[1,2]
    //3,[1,2,3]
    //....
    var map=new Map();
    ary.forEach((item,index)=>{
        map.set(index+1,ary.slice(0,index+1))
    });
    console.log(map);//Map(6)


```

8.4.3 map的遍历

> forEach(),keys(),values(),entries();

```
    //forEach(),keys(),values(),entries();
    map.forEach((val,key,map)=>{
        //val:值,
        //key:键
        //map:原Map实例
    })
    for(var key of map.keys()){
        //key:键
    }
    for(var val of map.values()){
        //val:值,
    }
    for (var [key,val] of map.entries()){
        //val:值,
        //key:键
    }
```

9 . Symbol

9.1 Symble的声明方式

```
//1.定义
let a = Symbol();
let b = Symbol();
console.log(a===b);//false
console.log(a==b);//false
console.log(a=b);//Symbol()
console.log(a);//Symbol()
```

9.2 Symbol.for("x")

```
    console.dir(Symbol().constructor);
    /*
        ƒ Symbol()
            arguments: (...)
            asyncIterator: Symbol(Symbol.asyncIterator)
            caller: (...)
```

```
            for: ƒ for()
            hasInstance: Symbol(Symbol.hasInstance)
            isConcatSpreadable: Symbol(Symbol.isConcatSpreadable)
            iterator: Symbol(Symbol.iterator)
            keyFor: ƒ keyFor()
            length: 0
            match: Symbol(Symbol.match)
            name: "Symbol"
            prototype: Symbol {constructor: ƒ, toString: ƒ, valueOf: ƒ,
Symbol(Symbol.toStringTag): "Symbol", …}
            replace: Symbol(Symbol.replace)
            search: Symbol(Symbol.search)
            species: Symbol(Symbol.species)
            split: Symbol(Symbol.split)
            toPrimitive: Symbol(Symbol.toPrimitive)
            toStringTag: Symbol(Symbol.toStringTag)
            unscopables: Symbol(Symbol.unscopables)
            __proto__: ƒ ()
            [[Scopes]]: Scopes[0]
     */
    //2.Symbol.for("x")
    //声明一个固定的 名为x的Symbol实例
    //以后再定义相同的 Symbol的实例的时候，指的是同一个Symbol实例
    let a = Symbol.for('aaa');
    let b = Symbol.for('aaa');
    console.log(a===b);//true
```

9.3 `Symbol`数据类型数据的 唯一性

```
    //Symbol.for('x')创建的x变量名跟字符串形式的 "x"属性名不相等
    let a = Symbol.for('aaa');
    let obj = {
        aaa:"我是名叫aaa的私有属性",
        [a]:"我是Symbol.for('aaa')声明的aaa变量名",
        bbb:'bbbbbbb'
    }
    console.log(obj);
    /*
        {
            aaa: "我是名叫aaa的私有属性",
            bbb: "bbbbbbb",
            Symbol(aaa): "我是Symbol.for('aaa')声明的aaa变量名"
        }
     */
```

9.4 `Symbol`的遍历

```
        let a = Symbol.for('aaa');
        let obj = {
            aaa:"我是名叫aaa的私有属性",
            [a]:"我是Symbol.for('aaa')声明的aaa变量名",
            bbb:'bbbbbbb'
        }

        //Symbol是不可枚举的
        //所以
        //一般的遍历方式是不能够遍历出来的
        for(let key in obj){
            console.log(key+':'+obj[key]);
        }
        //aaa:我是名叫aaa的私有属性
        //bbb:bbbbbbb

        for(let [key,index] of Object.entries(obj)){
            console.log(key+':'+obj[key]);
        }
        //aaa:我是名叫aaa的私有属性
        //bbb:bbbbbbb
        //...
        //遍历Symbol需要使用Object针对Symbol的API
        //Object.getOwnPropertySymbols(obj)
        //这个方法只能遍历出Symbol创建的变量
        Object.getOwnPropertySymbols(obj).forEach((item)=>{
            console.log(obj[item]);
        });
        //我是Symbol.for('aaa')声明的aaa变量名

        //如果 想要都遍历的到
        //使用ECMA2017的Reflect.ownKeys(obj)
        //返回结果是一个数组
        console.log(Reflect.ownKeys(obj));
        //["aaa", "bbb", Symbol(aaa)]

        console.dir(Reflect);
        /*
        ƒ Object()
            arguments: (...)
            assign: ƒ assign()
            caller: (...)
            create: ƒ create()
            defineProperties: ƒ defineProperties()
            defineProperty: ƒ defineProperty()
            entries: ƒ entries()
            freeze: ƒ freeze()
            getOwnPropertyDescriptor: ƒ getOwnPropertyDescriptor()
            getOwnPropertyDescriptors: ƒ getOwnPropertyDescriptors()
            getOwnPropertyNames: ƒ getOwnPropertyNames()
            getOwnPropertySymbols: ƒ getOwnPropertySymbols()
```

```
52.             getPrototypeOf: ƒ getPrototypeOf()
53.             is: ƒ is()
54.             isExtensible: ƒ isExtensible()
55.             isFrozen: ƒ isFrozen()
56.             isSealed: ƒ isSealed()
57.             keys: ƒ keys()
58.             length: 1
59.             name: "Object"
60.             preventExtensions: ƒ preventExtensions()
61.             prototype: {constructor: ƒ, __defineGetter__: ƒ, __defineSe
    tter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__: ƒ, …}
62.             seal: ƒ seal()
63.             setPrototypeOf: ƒ setPrototypeOf()
64.             values: ƒ values()
65.             __proto__: ƒ ()
66.             [[Scopes]]: Scopes[0]
67.          */
68.
```

10. RegExp

```
> console.dir(/i/);
  ▼/i/ i
     dotAll: false
     flags: ""
     global: false
     ignoreCase: false
     lastIndex: 0
     multiline: false
     source: "i"
     sticky: false
     unicode: false
    ▼__proto__:
     ▶compile: ƒ compile()
     ▶constructor: ƒ RegExp()
      dotAll: false
     ▶exec: ƒ exec()
      flags: ""
      global: false
      ignoreCase: false
      multiline: false
      source: "i"
      sticky: false
     ▶test: ƒ test()
     ▶toString: ƒ toString()
      unicode: (...)
     ▶Symbol(Symbol.match): ƒ [Symbol.match]()
     ▶Symbol(Symbol.replace): ƒ [Symbol.replace]()
```

10.1 构造函数

```
    //1.构造函数
    // 定义一个  '/abc/g' 这样子的正则
    let es5_reg1 = new RegExp('abc','g');
    //第一个参数是字符串，第二个是修饰符
    let es5_reg2 = new RegExp(/abc/g);
    //第一个参数是正则表达式，不接受第二个参数，否则会报错

    let es6_reg = new RegExp(/abc/ig, 'g');
    console.log(es6_reg); //    /abc/g
    //后边的i修饰符会覆盖前边出现的修饰符

```

10.2 ES6中新增的正则API

```
//2.正则属性和方法的扩展
  /*
  /abc/i
     dotAll: false
     flags: "i"
     global: false
     ignoreCase: true
     lastIndex: 0
     multiline: false
     source: "abc"
     sticky: false
     unicode: false
     __proto__:
       compile: ƒ compile()
       constructor: ƒ RegExp()
       dotAll: false
       exec: ƒ exec()
       flags: "i"
       global: false
       ignoreCase: true //忽略字母的大小写；忽略大小写
       multiline: false
       source: "abc"
       sticky: false
       test: ƒ test()
       toString: ƒ toString()
       unicode: false
    */

    //match(),replace(),search(),split()
  let s = "sss_ss_s_ssss";
  let reg1 = /s+/g;
  let reg2 = /s+/y;
  let reg3 = /s+/s;
  //查看修饰符 的属性 reg.flags
```

```
    console.log(reg1.flags,reg2.flags);//g y
    //查看是否启用了y修饰符 reg.sticky
    console.log(reg1.sticky,reg2.sticky);//false true
    //查看是否启用了s（空白修饰符）
    console.log(reg1.dotAll,reg2.dotAll,reg3.dotAll);// false false true
```

10.3 新增修饰符 `y`,`u`,`s`

10.3.1 `y`修饰符

```
//3.1 y修饰符
  let s = 'sss_ss_s_ssss';
  let reg_g = /s+/g;
  let reg_y = /s+/y;

  //第一次捕获
  console.log(reg_g.exec(s));
  //["sss", index: 0, input: "sss_ss_s_ssss", groups: undefined]
  console.log(reg_y.exec(s));
  //["sss", index: 0, input: "sss_ss_s_ssss", groups: undefined]

  //第二次捕获
  console.log(reg_g.exec(s));
  //["ss", index: 4, input: "sss_ss_s_ssss", groups: undefined]
  console.log(reg_y.exec(s));
  //null

  //第n次捕获，带y修饰符的都捕获不到
  //为什么呢？
  //因为
  //y修饰符的捕获规则是：
  //  从上一次捕获的位置开始，继续捕获，
  //  显然第一次捕获完后的下一次的位置是 _ss_s_sss
  // 所以他匹配的是 _ 字符，不是s
  // 所以捕获不到，返回null
```

10.3.2 `u`修饰符

```

 //3.2 u修饰符
  //使用u修饰符
  //匹配Unicode字符串的时候
  //  必须跟使用了u修饰符的正则中的内容完全一致
  //匹配字符串的时候
  //  它会先将字符串转换成Unicode编码的字符，然后再进行匹配

```

```
  //不使用u修饰符
  //匹配Unicode字符串的时候
  //  按照普通字符的方式去匹配
  //匹配字符串的时候
  //  直接匹配
  let str_unicode ='\uD83D\uDC2A' ;
  console.log(str_unicode);//🐪


  let reg_not_u = /^\uD83D/g;
  let reg_u = /^\uD83D/u;
  console.log(reg_not_u.test(str_unicode));//true
  console.log(reg_u.test(str_unicode));//false

  let str ='\uD83D';
  console.log(reg_u.test(str));//true

  //正则中使用Unicode
  console.log(/\u{61}/.test('a'));//false
  console.log(/\u{61}/u.test('a'));//true

  //u修饰符的作用：匹配大于2个字节的字符(0xffff);
  let more_then_2byte = "\u{21bba}";
  console.log(more_then_2byte);//𡮊
  let str_more_then_2byte = '𡮊';
  // . 通配符
  let reg_no_u = /^.$/;
  let reg_with_u = /^.$/u;
  console.log(reg_no_u.test(more_then_2byte));//false
  console.log(reg_with_u.test(more_then_2byte));//true
  console.log(reg_no_u.test(str_more_then_2byte));//false
  console.log(reg_with_u.test(str_more_then_2byte));//true

```

10.3.3 s修饰符

```

//3.3 s修饰符-------匹配到 换行符，tab 空格
  //正则表达式中，点（.）是一个特殊字符，代表任意的单个字符，但是行终止符（line terminator character）除外。

  //以下四个字符属于”行终止符“。
  //U+000A 换行符（\n）
  //U+000D 回车符（\r）
  //U+2028 行分隔符（line separator）
  //U+2029 段分隔符（paragraph separator）

  console.log(/foo.bar/.test('foo\nbar'));// false
//上面代码中，因为.不匹配\n，所以正则表达式返回false。
//但是，很多时候我们希望匹配的是任意单个字符，这时有一种变通的写法。
```

```
14.
15. console.log(/foo[^]bar/.test('foo\nbar'));// true
16.
17. //这种解决方案毕竟不太符合直觉，所以现在有一个提案，引入`/s`修饰符，使得.可以匹配任意单个字符。
18. console.log(/foo.bar/s.test('foo\nbar')); // true
19. //这被称为dotAll模式，即点（dot）代表一切字符。所以，正则表达式还引入了一个dotAll属性，返回一个布尔值，表示该正则表达式是否处在dotAll模式。
20.
21. //`/s`修饰符和多行修饰符`/m`不冲突，两者一起使用的情况下，`.`匹配所有字符，而^和$匹配每一行的行首和行尾。
```

11 数值扩展

11.1 各进制的表示法

```
1.      //1. 二进制的表示法  0B/0b
2.      console.log(0b11101010);//234
3.      console.log(0B11101010);//234
4.
5.      //2. 八进制表示法  0o/0O
6.      console.log(0o123745);//42981
7.      console.log(0O123745);//42981
8.
9.      //3. 十六进制表示法  0x/0X
10.     console.log(0x123745f12);//4889796370
11.     console.log(0X123745f12);//4889796370
```

11.2 Number类的扩展

```
▼Number
  ▼__proto__: Number
    ▼constructor: ƒ Number()
       EPSILON:
       2.220446049250313e-16
       MAX_SAFE_INTEGER: 9007199254740991
       MAX_VALUE:
       1.7976931348623157e+308
       MIN_SAFE_INTEGER: -9007199254740991
       MIN_VALUE:
       5e-324
       NEGATIVE_INFINITY: -Infinity
       NaN: NaN
       POSITIVE_INFINITY: Infinity
       arguments: (...)
       caller: (...)
      ▶isFinite: ƒ isFinite()
      ▶isInteger: ƒ isInteger()
      ▶isNaN: ƒ isNaN()
      ▶isSafeInteger: ƒ isSafeInteger()
       length: 1
       name: "Number"
      ▶parseFloat: ƒ parseFloat()
      ▶parseInt: ƒ parseInt()
      ▶prototype: Number {0, constructor: ƒ, toExponential: ƒ, toFixed: ƒ, toPrecision: ƒ, toString: ƒ, …}
      ▶__proto__: ƒ ()
      ▶[[Scopes]]: Scopes[0]
    ▶toExponential: ƒ toExponential()
    ▶toFixed: ƒ toFixed()
    ▶toLocaleString: ƒ toLocaleString()
    ▶toPrecision: ƒ toPrecision()
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
    ▶__proto__: Object
     [[PrimitiveValue]]: 0
   [[PrimitiveValue]]: 1
```

11.3 Math类

```
Math
    E: 2.718281828459045
    LN2: 0.6931471805599453
    LN10: 2.302585092994046
    LOG2E: 1.4426950408889634
    LOG10E: 0.4342944819032518
    PI: 3.141592653589793
    SQRT1_2: 0.7071067811865476
    SQRT2: 1.4142135623730951
    abs: ƒ abs()
    acos: ƒ acos()
    acosh: ƒ acosh()
    asin: ƒ asin()
    asinh: ƒ asinh()
    atan: ƒ atan()
    atan2: ƒ atan2()
    atanh: ƒ atanh()
    cbrt: ƒ cbrt()
    ceil: ƒ ceil()
    clz32: ƒ clz32()
    cos: ƒ cos()
    cosh: ƒ cosh()
    exp: ƒ exp()
```

```
    expm1: ƒ expm1()
    floor: ƒ floor()
    fround: ƒ fround()
    hypot: ƒ hypot()
    imul: ƒ imul()
    log: ƒ log()
    log1p: ƒ log1p()
    log2: ƒ log2()
    log10: ƒ log10()
    max: ƒ max()
    min: ƒ min()
    pow: ƒ pow()
    random: ƒ random()
    round: ƒ round()
    sign: ƒ sign()
    sin: ƒ sin()
    sinh: ƒ sinh()
    sqrt: ƒ sqrt()
    tan: ƒ tan()
    tanh: ƒ tanh()
    trunc: ƒ trunc()
    Symbol(Symbol.toStringTag): "Math"
    __proto__: Object
```