

let变量

①

变量不能重复声明

var可以重复声明

②

块级作用域

var没有块级作用域

```
<script>
    //获取div元素对象
    let items = document.getElementsByClassName('item');

    //遍历并绑定事件
    for(var i = 0;i<items.length;i++){
        items[i].onclick = function(){
            //修改当前元素的背景颜色
            this.style.background = 'pink';
        }
    }
</script>
```

```
//获取div元素对象
let items = document.getElementsByClassName('item');

//遍历并绑定事件
for(let i = 0;i<items.length;i++){
    items[i].onclick = function(){
        //修改当前元素的背景颜色
        // this.style.background = 'pink';
        items[i].style.background = 'pink';
    }
}
```

③

不存在变量提升

var有变量提升

④

不影作用域链

const

一定要赋初始值

一般常量大写

常量的值不能修改

存在块级作用域

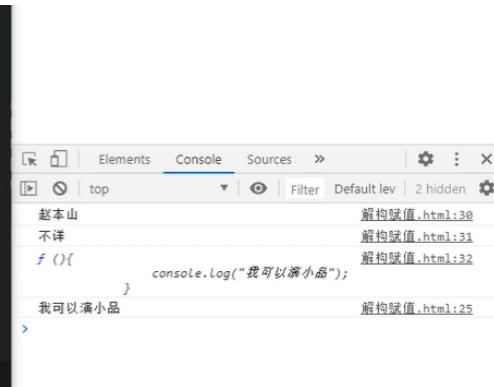
对于数组元素的修改不算对常量的修改，因为数组指向的地址始终没有改变

变量的解构赋值

数组的解构

```
//1. 数组的结构
const F4 = ['小沈阳', '刘能', '赵四', '宋小宝'];
let [xiao, liu, zhao, song] = F4;
console.log(xiao);
console.log(liu);
console.log(zhao);
console.log(song);
```

方法的解构



```
//2. 对象的解构
const zhao = {
  name: '赵本山',
  age: '不详',
  xiaopin: function(){
    console.log("我可以演小品");
  }
};
let {name, age, xiaopin} = zhao;
console.log(name);
console.log(age);
console.log(xiaopin);
xiaopin();
```

The screenshot shows the browser's developer tools console tab. It displays the following output:

Value	Source
赵本山	解构赋值.html:30
不详	解构赋值.html:31
f () { console.log("我可以演小品"); }	解构赋值.html:32
我可以演小品	解构赋值.html:25

模板字符串字符串

在es6中有一个新的引号：` ` (即键盘上的破浪号)

` `引号使用时字符串可以换行

[ES6模板字符串之标签模板](#)

首先，模板字符串和标签模板是两个东西。

标签模板不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。

但是，如果模板字符串中有变量，就不再是简单的调用了，而是要将模板字符串先处理成多个参数，再调用函数。([ES6标准入门-阮一峰 4.12标签模板]

由此引出此文，先上代码：

```
var a = 5;
var b = 10;

tag`Hello ${ a + b } world ${ a * b }`;
//等同于
tag(['Hello ', ' world ', ''], 15, 50);
```

这里我产生了疑问，为什么数组第三个参数是空字符串，书中也未曾讲到，很是疑惑。

然后再看下一段代码：

```
var total = 30;
var msg = passthru`The total is ${total} (${total * 1.05} with tax)`;
...
//由此可以得出 上面的方法等同于
var msg = passthru(['The total is ', '(', ' with tax'], 30, 31.5)
```

在这段代码中数组参数并没有产生空字符串，原因我也不知道，百度了一番，然后懂了。

模板字符串由变量和非变量组成，什么是变量，\${}就是变量。模板标签函数调用的第一个参数是数组，是由模板字符串中那些非变量部分组成，包括空格。

那么不难理解，假设模板字符串中的变量为A，非变量为B，那么模板字符串的组成可能就是：

```
tpl1 = ABABA; -> ['', B, B, '']
tpl2 = BAB;     -> [B, B]
tpl3 = ABAB    -> ['', B, B]
...
```

可以解读到：模板字符串中变量必须是由非变量包含着的，如果变量在开始位置或者结束位置且没有非变量包含，那么该位置就是空字符串。

对象的简化写法

```
<script>
    //ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。
    //这样的书写更加简洁
    let name = '尚硅谷';
    let change = function(){
        console.log('我们可以改变你!!');
    }

    const (property) name: never
        name,
        change,
        improve(){
            console.log("我们可以提高你的技能");
        }
}
```

函数可以省略function

箭头函数

①this是静态的，始指向行函数声明时所在的作用域下的this值

并不会像之前构造函数，谁调用this就指向哪个实例对象

②箭头函数并不能作为构造函数实例化对象

③不能使用arguments变量

例子：

```
<script>
    //需求-1 点击 div 2s 后颜色变成『粉色』
    //获取元素
    let ad = document.getElementById('ad');
    //绑定事件
    ad.addEventListener("click", function(){
        //保存 this 的值
        let _this = this;
        //定时器
        setTimeout(function(){
            //修改背景颜色 this
            // console.log(this);
            _this.style.backgroundColor = 'pink';
        }, 2000);
    });
}
```

```
//需求-1 点击 div 2s 后颜色变成『粉色』  
//获取元素  
let ad = document.getElementById('ad');  
//绑定事件  
ad.addEventListener("click", function(){  
    //保存 this 的值  
    // let _this = this;  
    //定时器  
    setTimeout(() => {  
        //修改背景颜色 this  
        // console.log(this);  
        // _this.style.background = 'pink';  
        this.style.background = 'pink';  
    }, 2000);  
});
```

例子：

```
//需求-2 从数组中返回偶数的元素  
const arr = [1,6,9,10,100,25];  
// const result = arr.filter(function(item){  
//     if(item % 2 === 0){  
//         return true;  
//     }else{  
//         return false;  
//     }  
// });  
  
const result = arr.filter(item => item % 2 === 0);  
  
console.log(result);
```

箭头函数适合于this无关的回调，定时器、数组方法的回调

箭头函数不适合与this有关的回调，事件的回调函数、对象的方法

参数默认值设置

es6允许给函数参数赋初始值

①形参初始值

具有默认值的参数，一般位置要靠后（放在前面不会报错，但是使用不了，想要省略某一个参数的赋值，参数传进去后，被省略的总是最后一个形参，前面的都被赋值了，所以放前面说意义不大）

```
//ES6 允许给函数参数赋值初始值
//1. 形参初始值 具有默认值的参数，一般位置要靠后
function add(a,b,c=10) {
    return a + b + c;
}
let result = add(1,2);
console.log(result);
```

②与解构赋值结合

```
//2. 与解构赋值结合
function connect({host="127.0.0.1", username, password, port}){
    console.log(host)
    console.log(username)
    console.log(password)
    console.log(port)
}
connect({
    host: 'localhost',
    username: 'root',
    password: 'root',
    port: 3306
})
```

rest参数

es6中引入rest参数，用来获取函数的实参，用来代替arguments

扩展运算符

扩展运算符（spread）是三个点（...），可以将一个数组转为用逗号分隔的参数序列。

说的通俗易懂点，有点像化骨绵掌，**把一个大元素给打散成一个个单独的小元素。**

基本用法：拆解字符串与数组

```
var array = [1,2,3,4];
console.log(...array); //1 2 3 4
var str = "String";
console.log(...str); //S t r i n g
```

扩展运算符应用

2.1 某些场景可以替代apply

在使用Math.max()求数组的最大值时，ES5可以通过apply做到（用一种不友好且繁琐的方式）

```
// ES5 apply 写法
var array = [1,2,3,4,3];
var max1 = Math.max.apply(null, array);
console.log(max1); //4
```

幸运的是JavaScript的世界在不断改变，扩展运算符可用于数组的析构，优雅的解决了这个问题。

```
// ES6 扩展运算符 写法
var array = [1,2,3,4,3];
var max2 = Math.max(...array);
console.log(max2); //4
```

先把array打散成1 2 3 4 3，再在里面找最大的那一个，就显而易见了。

2.2 代替数组的push、concat等方法

实现把arr2塞到arr1中

```
// ES5 apply 写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
Array.prototype.push.apply(arr1, arr2);
//arr1 [0, 1, 2, 3, 4, 5]
```

扩展运算符又要施展化骨大法了

```
// ES6 扩展运算符 写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
//arr1 [0, 1, 2, 3, 4, 5]
```

通俗的解释下，扩展运算符先把arr2打散成3 4 5，之后再往arr1里push，就轻松多了。

同理可推，**concat**合并数组的时候：

```

var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];

// ES5的合并数组
arr1.concat(arr2, arr3) // [ 'a', 'b', 'c', 'd', 'e' ]

// ES6的合并数组
[...arr1, ...arr2, ...arr3] // [ 'a', 'b', 'c', 'd', 'e' ]

```

ES5的合并数组写法，像是 arr1 把 arr2, arr3 给吸收了。

而ES6的合并数组写法，则是先把 arr1, arr2, arr3 拆解，之后塞到新的数组中。

2.3 拷贝数组或对象

```

//拷贝数组
var array0 = [1,2,3];
var array1 = [...array0];
console.log(array1); // [1, 2, 3]

//拷贝对象
var obj = {
    age:1,
    name:"lis",
    arr:{
        a1:[1,2]
    }
}
var obj2 = {...obj};
console.log(obj2); // {age: 1, name: "lis", arr: {}}

```

无论是像克隆数组还是对象，先用化骨绵掌之扩展运算符，将其打散，之后再拼装的到一起就可以了，多么简单易用。

2.4 将伪数组转化为数组

```

//伪数组转换为数组
var nodeList = document.querySelectorAll('div');
console.log([...nodeList]); // [div, div, div ... ]

```

上面代码中，`querySelectorAll` 方法返回的是一个 `nodeList` 对象。它不是数组，而是一个类似数组的对象。

这时，**扩展运算符可以将其转为真正的数组**，原因就在于 `NodeList` 对象实现了 `Iterator`。

注意：使用扩展运算符将伪数组转换为数组有局限性，这个类数组必须得有默认的迭代器且伪可遍历的。

rest 运算符

剩余运算符 (the rest operator)，它的样子看起来和展开操作符一样，但是它是用于**解构数组和对象**。在某种程度上，剩余元素和展开元素相反，展开元素会“展开”数组变成多个元素，**剩余元素会收集多个元素和“压缩”成一个单一的元素。**

说的通俗点，有点像吸星大法，收集多个元素，压缩成单一的元素。

rest参数用于获取函数的多余参数，这样就不需要使用**arguments对象**了。rest参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

如果函数有实参，rest参数要接收多余的实参，**它必须放在最后**

例如实现计算传入所有参数的和

使用arguments参数：

```
function sumArgu () {
    var result = 0;
    for (var i = 0; i < arguments.length; i++) {
        result += arguments[i];
    }
    return result
}
console.log(sumArgu(1,2,3));//6
```

使用rest参数：

```
function sumRest (...m) {
    var total = 0;
    for(var i of m){
        total += i;
    }
    return total;
}
console.log(sumRest(1,2,3));//6
```

上面代码利用 rest 参数，可以向该函数传入任意数目的参数。传递给 sumRest 函数的一组参数值，被整合成了数组 m。

就像是吸星大法，把分散的元素收集到一起。

所以在某些场景中，**无需将arguments转为真正的数组，可以直接使用rest参数代替。**

rest 运算符应用

4.1 rest 参数代替arguments变量

```
// arguments变量的写法，将arguments转化为数组，然后用sort进行排序
function sortNumbers() {
    return Array.prototype.slice.call(arguments).sort();
}

// rest参数的写法，将arguments转化为数组，并进行排序
const sortNumbers = (...numbers) => numbers.sort();
```

上面的两种写法，比较后可以发现，rest 参数的写法更自然也更简洁。

不过，rest参数和arguments对象有一定的区别：

rest参数	arguments对象
rest参数只包含那些没有对应形参的实参	arguments 对象包含了传给函数的所有实参
rest参数是真实的 Array 实例，也就是说你能够在它上面直接使用所有的数组方法	arguments 对象不是一个真正的数组
rest参数无附加的属性	arguments 对象还有一些附加的属性(比如callee属性) 知乎 @前端大彬哥

4.2 与解构赋值组合使用

```
var array = [1,2,3,4,5,6];
var [a,b,...c] = array;
console.log(a); //1
console.log(b); //2
console.log(c); // [3, 4, 5, 6]
```

备注：rest参数可理解为剩余的参数，所以必须在最后一位定义，如果定义在中间会报错。

```
var array = [1,2,3,4,5,6];
var [a,b,...c,d,e] = array;
// Uncaught SyntaxError: Rest element must be last element
```

总结

5.1 扩展运算符和rest运算符是逆运算

扩展运算符：数组=>分割序列

rest运算符：分割序列=>数组

5.2 扩展运算符应用场景

由于其繁琐的语法，apply 方法使用起来并不是很方便。当需要拿一个数组的元素作为函数调用的参数时，

扩展运算符是一个不错的选择。

扩展运算符还改善了数组字面量的操作，你可以更方便的初始化、连接、复制数组了。

使用析构赋值你可以提取数组的一部分。通过与迭代器协议的组合，你可以以一种更灵活的方式使用该表达式。

5.3 rest运算符应用场景

rest运算符主要是处理不定数量参数，rest参数使得收集参数变得非常简单。它是类数组对象 arguments一个合理的替代品。

rest参数还可以与解构赋值组合使用。

在实际项目中灵活应用扩展运算符、rest运算符，能写出更精简、易读性高的代码。

Symbol

ES6引入了一种新的原始数据类型 Symbol表示独一无二的值。

它是Javascript语言的第七种数据类型，是一种类似于字符串的数据类型。

Symbol特点

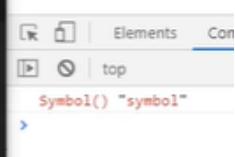
1) Symbol的值是唯一的，用来解决命名冲突的问题

2) Symbol 值不能与其他数据进行运算

3) Symbol定义的对象属性不能使用for...in循环遍历，但是可以使用Reflect. own Keys来获取对象的所有键名

创建方法

①

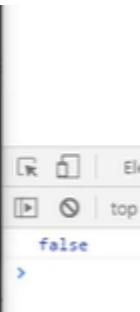


```
<body>
  <script>
    //创建Symbol
    let s = Symbol();
    console.log(s, typeof s);| I
  </script>
</body>
</html>
```

Symbol创建：通过Symbol()函数调用，来返回一个Symbol值

Symbol创建的是唯一的一个值，但是通过console.log打印出来仍是Symbol()，在这里是观测不到的

还可以往Symbol()函数中传入一个参数，这个参数只是用来描述Symbol值而已，并不是随机因子的作用，参数相同的Symbol函数得到的Symbol值也是不一样的



```
<script>
  //创建Symbol
  let s = Symbol();
  // console.log(s, typeof s);
  let s2 = Symbol('尚硅谷');
  let s3 = Symbol('尚硅谷');| I
  //|
  console.log(s2 === s3);
```

②

可以通过Symbol.for([参数])，这种方式来创建SYmbol值，同样，里面的参数可选

但不同的是：1.这种方法不是通过函数创建的了，而是一个函数对象

2.这种方法传入的参数，相同参数得到的Symbol值是一样的

```
//创建Symbol
let s = Symbol();
// console.log(s, typeof s);
let s2 = Symbol('尚硅谷');
let s3 = Symbol('尚硅谷');
//Symbol.for 创建
let s4 = Symbol.for('尚硅谷');
let s5 = Symbol.for('尚硅谷');
|
console.log(s4 === s5);
```

应用

给对象添加属性和方法

当想向一个对象中添加方法时，不确定这个对象中有没有已经存在相应的方法名，担心方法名重复，使得新加的方法覆盖了原来的方法

这时就可以使用Symbol值，使用方法如下，将方法名通过Symbol值来进行命名

```
//向对象中添加方法 up down
let game = { ... }

//声明一个对象
let methods = {
  up: Symbol(),
  down: Symbol()
};

game[methods.up] = function(){
  console.log("我可以改变形状");
}

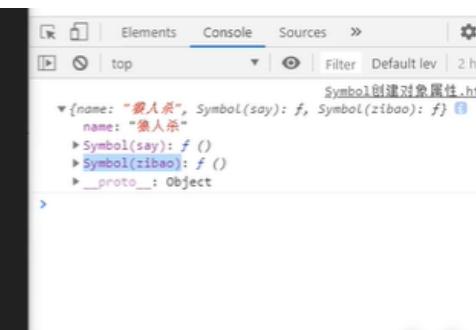
game[methods.down] = function(){
  console.log("我可以快速下降!!!");
}

console.log(game); |
```



```
let youxi = {
  name: "狼人杀",
  [Symbol('say')]: function(){
    console.log("我可以发言")
  },
  [Symbol('zibao')]: function(){
    console.log('我可以自爆');
  }
}

console.log(youxi)
```



Symbol的内置属性

除了定义自己使用的 `Symbol` 值以外，ES6 还提供了 11 个内置的 `Symbol` 值，指向语言内部使用的方法。

<code>Symbol.hasInstance</code>	当其他对象使用 <code>instanceof</code> 运算符，判断是否为该对象的实例时，会调用这个方法
<code>Symbol.isConcatSpreadable</code>	对象的 <code>Symbol.isConcatSpreadable</code> 属性等于的是一个布尔值，表示该对象用于 <code>Array.prototype.concat()</code> 时，是否可以展开。
<code>Symbol.unscopables</code>	该对象指定了使用 <code>with</code> 关键字时，哪些属性会被 <code>with</code> 环境排除。
<code>Symbol.match</code>	当执行 <code>str.match(myObject)</code> 时，如果该属性存在，会调用它，返回该方法的返回值。
<code>Symbol.replace</code>	当该对象被 <code>str.replace(myObject)</code> 方法调用时，会返回该方法的返回值。
<code>Symbol.search</code>	当该对象被 <code>str.search (myObject)</code> 方法调用时，会返回该方法的返回值。
<code>Symbol.split</code>	当该对象被 <code>str.split (myObject)</code> 方法调用时，会返回该方法的返回值。

这些内置属性是控制对象在特定场景下的一个表现，

举个例子，`Symbol.hasInstance` 方法决定了使用 `instanceof` 方法判断某个对象是否是某个构造函数或者类的实例

再例如

```
// console.log("找被用来检测类型了");
// return false;
// }
// }

// let o = {};

// console.log(o instanceof Person);

isConcatSpreadable决定了数组在使用concat方法时是否可以展开

const arr = [1,2,3];
const arr2 = [4,5,6];
arr2[Symbol.isConcatSpreadable] = false;
console.log(arr.concat(arr2));
</script>
```

迭代器 (Iterator)

es6中创造了一种新的遍历命令 `for...of` 循环

`for...of` 和 `for...in` 作用有区别

for...of 循环的是数组值/对象的属性值

for...in 循环的是数组索引/对象的属性名

```
// 声明一个数组
const xiyou = ['唐僧', '孙悟空', '猪八戒', '沙僧'];

// 使用 for...of 遍历数组
for(let v of xiyou){
    console.log(v);
}
```



只要对象中提供了Iterator接口就可以使用for...of循环

Iterator接口就是对象中**Symbol.iterator属性**, 它的值是一个**函数**

下面这些数据都具备Iterator接口, 上面的例子中的**数组**就是其中之一

2) 原生具备 iterator 接口的数据(可用 for of 遍历)

- a) Array
- b) Arguments
- c) Set
- d) Map
- e) String
- f) TypedArray
- g) NodeList

工作原理

3) 工作原理

- a) 创建一个指针对象, 指向当前数据结构的起始位置
- b) 第一次调用对象的 next 方法, 指针自动指向数据结构的第一个成员
- c) 接下来不断调用 next 方法, 指针一直往后移动, 直到指向最后一个成员
- d) 每调用 next 方法返回一个包含 value 和 done 属性的对象

注: 需要自定义遍历数据的时候, 要想到迭代器。

下面是查看对象中的next方法, **next是Symbol.iterator属性中的一个方法**

```
// 声明一个数组
const xiyou = ['唐僧', '孙悟空', '猪八戒', '沙僧'];

// 使用 for...of 遍历数组
// for(let v of xiyou){
//     console.log(v);
// }

let iterator = xiyou[Symbol.iterator]();

console.log(iterator);
```



```
//声明一个数组
const xiyou = ['唐僧', '孙悟空', '猪八戒', '沙僧'];

//使用 for...of 遍历数组
// for(let v of xiyou){
//     console.log(v);
// }

let iterator = xiyou[Symbol.iterator]();

//调用对象的next方法
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

```
[value: "唐僧", done: false]
▶ {value: "孙悟空", done: false}
▶ {value: "猪八戒", done: false}
▶ {value: "沙僧", done: false}
▶ {value: undefined, done: true}
```

迭代器的编程实现

```
name: "终极一班",
stus: [
    'xiaoming',
    'xiaoning',
    'xiaoqian',
    'knight'
],
[Symbol.iterator]() {
    //索引变量
    let index = 0;
    //
    let _this = this;
    return {
        next: function () {
            if (index < _this.stus.length) {
                const result = { value: _this.stus[index], done: false
                    //下标自增
                    index++;
                    //返回结果
                    return result;
                }else{
                    return {value: undefined, done: true};
                }
            }
        }
}
```

```
//遍历这个对象
for(let v of banji){
    console.log(v);
}

// banji.stus.forEach()
```

		Filter	Default
xiaoming	迭代器自定义遍		
xiaoning	迭代器自定义遍		
xiaotian	迭代器自定义遍		
knight	迭代器自定义遍		

生成器

生成器其实是一种特殊的函数

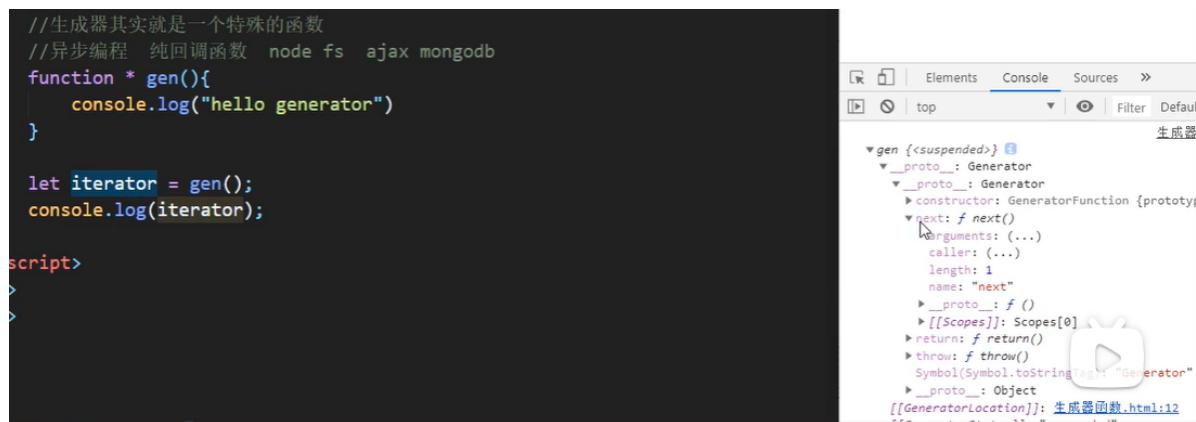
用来进行异步编程

声明与调用

生成器在声明上面和执行上面都跟一般的函数不一样

声明时，**function * 函数名()**，需要在function和函数名之间加一个***星号**，*两遍哪怕没有空格也行，

执行时直接执行函数并不会得到函数执行结果，如图



```
//生成器其实就是一个特殊的函数
//异步编程 纯回调函数 node fs ajax mongodb
function * gen(){
  console.log("hello generator")
}

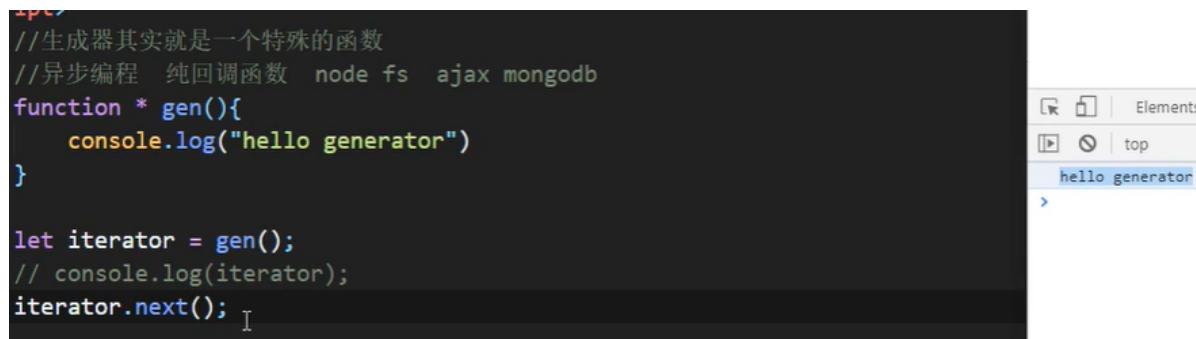
let iterator = gen();
console.log(iterator);

script>
>
>
```

The screenshot shows a browser's developer tools console. On the left, there is a code editor window with the above JavaScript code. On the right, the browser's developer tools interface is shown, specifically the 'Console' tab. It displays the output of the code: 'gen' is defined as a Generator object, and its properties like 'next', 'constructor', and 'Symbol(Symbol.toStringTag): "Generator"' are listed.

打印出来这个函数，发现是一个迭代器对象，对象里面有一个**next方法**

只有执行这个迭代器对象中的next方法后，才会得到函数执行结果



```
//生成器其实就是一个特殊的函数
//异步编程 纯回调函数 node fs ajax mongodb
function * gen(){
  console.log("hello generator")
}

let iterator = gen();
// console.log(iterator);
iterator.next(); I
```

The screenshot shows a browser's developer tools console. On the left, the same generator function code is shown. On the right, the console output shows the result of calling 'next()' on the iterator object. The output is 'Object { value: undefined, done: true}', indicating that the generator has completed its execution.

生成器中还有一个**yield语句**，yield语句用来**分割生成器函数**

如下图。三个yield语句将生成器分为四个部分

每执行一次next方法，生成器函数就往下执行一步

```
//生成器其实就是一个特殊的函数
//异步编程 纯回调函数 node fs ajax mongodb
//函数代码的分隔符
function * gen(){
    console.log(111);
    yield '一只没有耳朵';
    console.log(222);
    yield '一只没有尾部';
    console.log(333);
    yield '真奇怪';
    console.log(444);
}

let iterator = gen();
iterator.next();
iterator.next();
```



也可以用for...of来进行遍历

```
function * gen(){
    console.log(111);
    yield '一只没有耳朵';
    console.log(222);
    yield '一只没有尾部';
    console.log(333);
    yield '真奇怪';
    console.log(444);
}

// let iterator = gen();
// iterator.next();
// iterator.next();
// iterator.next();
// iterator.next();

//遍历
for(let v of gen()){
    console.log(v);
}
```



next方法也有返回值，就是之前讲过的value, done

value就是yield后后面的值

```
//生成器其实就是一个特殊的函数
//异步编程 纯回调函数 node fs ajax mongodb
//函数代码的分隔符
function * gen(){
    // console.log(111);
    yield '一只没有耳朵';
    // console.log(222);
    yield '一只没有尾部';
    // console.log(333);
    yield '真奇怪';
    // console.log(444);
}

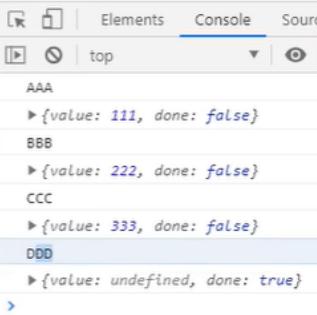
let iterator = gen();
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```



参数传递

```
function * gen(arg){
    console.log(arg);
    let one = yield 111;
    console.log(one);
    let two = yield 222;
    console.log(two);
    let three = yield 333;
    console.log(three);
}

//执行获取迭代器对象
let iterator = gen('AAA');
console.log(iterator.next());
//next方法可以传入实参
console.log(iterator.next('BBB'));
console.log(iterator.next('CCC'));
console.log(iterator.next('DDD'));
```



生成器还可以进行赋值

yield也可以进行赋值，只不过yield的赋值是在调用next方法时，往next中进行赋值

如图，**第二个next方法**中进行赋值，传入到迭代器中**第一个yield**中，作为第一个yield的**返回结果**。可以console.log查看

以此类推，**第三个next方法**传入的参数作为**第二个yield语句的返回结果**

生成器函数实例

用来解决异步编程中**回调地狱**的现象：异步操作中还嵌套着好几层异步操作（在promise中讲过）

回调地狱：

```
// 异步编程 文件操作 网络操作.ajax, request 数据库操作
// 1s 后控制台输出 111 2s后输出 222 3s后输出 333
// 回调地狱
setTimeout(() => {
    console.log(111);
    setTimeout(() => {
        console.log(222);
        setTimeout(() => {
            console.log(333);
        }, 3000);
    }, 2000);
}, 1000);
```

例子：

首先模拟三个异步操作，将三个异步操作函数在外部定义，然后将三个异步函数放在生成器中的
yield语句后面

然后可以调用生成器函数，并且执行一次next方法

```
//调用生成器函数
let iterator = gen();
iterator.next()
```

将执行的结果停留到第一次yield语句后，第一次异步操作执行结束后就不动了

然后将几个异步操作连起来，将三个next方法，放到三个异步函数中去。完整代码如下

```
function one(){
    setTimeout(()=>{
        console.log(111);
        iterator.next();
    },1000)
}

function two(){
    setTimeout(()=>{
        console.log(222);
        iterator.next();
    },2000)
}

function three(){
    setTimeout(()=>{
        console.log(333);
        iterator.next();
    },3000)
}

function * gen(){
    yield one();
    yield two();
    yield three();
}

//调用生成器函数
let iterator = gen();
iterator.next();
```

这样整体代码结构就比较清楚了

实例2：

```

function getUsers(){
    setTimeout(() => {
        let data = '用户数据';
        iterator.next(data);
    },1000)
}

function getOrders(){
    setTimeout(() => {
        let data = '订单数据';
        iterator.next(data);
    },1000)
}

function getGoods(){
    setTimeout(() => {
        let data = '商品数据';
        iterator.next(data);
    },1000)
}

function * gen(){
    let users = yield getUsers();
    console.log(users);
    let orders = yield getOrders();
    console.log(orders);
    let goods = yield getGoods();
    console.log(goods);
}

// 调用生成器函数
let iterator = gen();
iterator.next();

```

CODE 404, [HTML_LIVE_UNKNOWNS_ONE_SCHEME](#)

用户数据

[练习4.html](#):

订单数据

[练习4.html](#):

商品数据

[练习4.html](#):

之前讲的**第二个next方法**中传入的参数是**第一个yield语句返回结果**（第一个next在调用生成器函数就使用了），并且第二个next方法放在了第一个异步函数中，以此类推，

在生成器中yield返回结果使用变量进行接收，可以打印来

这样next与异步函数直接的关系比之前讲的更加直观

哪一个异步函数中next传入的参数，就是这个异步函数所在的yield语句返回值

Promise

使用方法之前详细讲过，这里不再叙述

```
//实例化 Promise 对象
const p = new Promise(function(resolve, reject){
    setTimeout(function(){
        //
        // let data = '数据库中的用户数据';
        //resolve
        // resolve(data);

        let err = '数据读取失败';
        reject(err);
    }, 1000);
});

//调用 promise 对象的 then 方法
p.then(function(value){
    console.log(value);
}, function(reason){
    console.error(reason);
})
```

例子：利用node.js中的fs模块读取文件，读取文件本身就是一个异步操作

```
//1. 引入 fs 模块
const fs = require('fs');
```

```
//3. 使用 Promise 封装
const p = new Promise(function(resolve, reject){
    fs.readFile("./resources/为学.mda", (err, data)=>{
        //判断如果失败
        if(err) reject(err);
        //如果成功
        resolve(data);
    });
});

p.then(function(value){
    console.log(value.toString());
}, function(reason){
    console.log("读取失败!!");
});

```

一个**ajax数据请求**的真实例子：（只不过这里没有成功，因为发生了跨域问题）

```

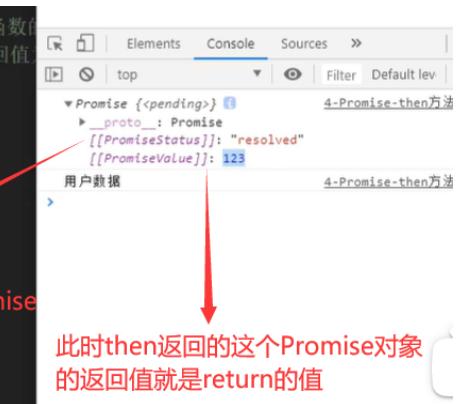
const p = new Promise((resolve, reject) =>{
    // 1. 创建对象
    const xhr = new XMLHttpRequest();
    // 2. 初始化
    xhr.open("GET", "http://api.apiopen.top/getJ");
    // 3. 发送
    xhr.send()
    // 4. 绑定事件，处理响应结果
    xhr.onreadystatechange = function(){
        // 判断整个过程是否进入了第四个阶段，也就是最后一个阶段
        if (xhr.readyState === 4){
            // 判断相应状态码200-299
            if (xhr.status >= 200 && xhr.status < 300){
                // 表示成功
                resolve(xhr.response);
            }else{
                reject(xhr.status)
            }
        }
    }
})
// 指定回调
p.then(function(value){
    console.log(value);
},function(reason){
    console.error(reason);
})

```

Promise的then方法

调用then方法**then方法的返回结果是Promise对象，Promise对象状态由回调函数的执行结果决定**

①如果回调函数中返回的结果是**非 promise类型的属性**，数字、字符串等等，**then方法的返回结果状态为成功，返回值为这个Promise对象的成功值**



The screenshot shows a browser's developer tools console. On the left, there is some JavaScript code demonstrating the then method. On the right, the console output shows a Promise object with its state changing from 'pending' to 'resolved' with a value of '123'. A red arrow points from the explanatory text below to the 'resolved' state in the console output.

```

// 调用 then 方法 then方法的返回结果是 Promise 对象，对象状态由回调函数的
// 1. 如果回调函数中返回的结果是 非 promise 类型的属性，状态为成功，返回值
// 为这个Promise对象的成功值

const result = p.then(value => {
    console.log(value);
    return 123;
}, reason=>{
    console.warn(reason);
});

console.log(result);

```

回调函数返回的结果为
数字，则then返回的这个Promise
对象状态为成功

此时then返回的这个Promise对象
的返回值就是return的值

②如果then的回调函数中return返回的是一个Promise对象

则**返回的这个Promise对象状态**就决定了**then返回的Promise对象状态**

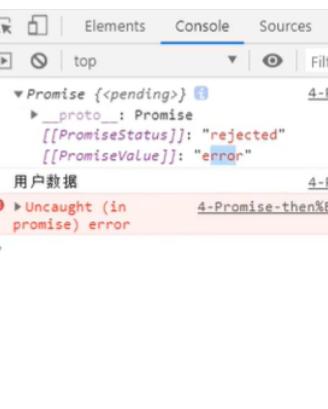
并且**return返回的Promise对象成功/失败的值就是then返回的Promise对象成功/失败的值**

```
const result = p.then(value => {
  console.log(value);
  //1. 非 promise 类型的属性
  // return 'iloveyou';
  //2. 是 promise 对象
  return new Promise((resolve, reject)=>{
    resolve('ok');
  });
}, reason=>{
  console.warn(reason);
});

console.log(result);
```



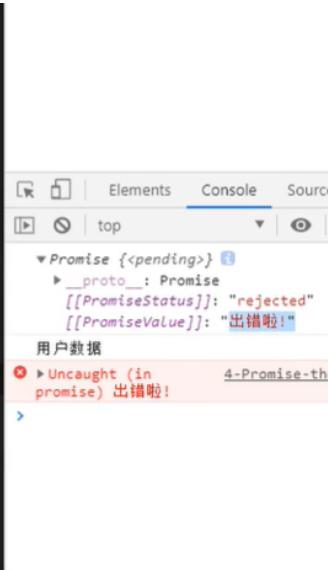
```
const result = p.then(value => {
  console.log(value);
  //1. 非 promise 类型的属性
  // return 'iloveyou';
  //2. 是 promise 对象
  return new Promise((resolve, reject)=>[
    // resolve('ok');
    reject('error');
  ]);
}, reason=>{
  console.warn(reason);
});
```



③如果then函数中的回调函数中直接抛出错误，则then返回的Promise对象状态为失败，失败的值就是抛出错误的值

```
const result = p.then(value => [
  console.log(value);
  //1. 非 promise 类型的属性
  // return 'iloveyou';
  //2. 是 promise 对象
  // return new Promise((resolve, reject)=>{
  //   // resolve('ok');
  //   // reject('error');
  // });
  //3. 抛出错误
  // throw new Error('出错啦!');
  throw '出错啦!';
], reason=>{
  console.warn(reason);
});

console.log(result);
```



因为then方法返回的是Promise对象由这种特性，所以可以在then中链式调用异步操作

catch

catch是用来捕捉错误的

使用catch可以做到**只指定一个回调函数**，并且这个回调函数是执行错误的回调函数，可以免于指定成功的回调函数

```
const p = new Promise((resolve, reject)=>{
    setTimeout(()=>{
        //设置 p 对象的状态为失败，并设置失败的值
        reject("出错啦!");
    }, 1000)
});

// p.then(function(value){}, function(reason){
//     console.error(reason);
// });

p.catch(function(reason){
    console.warn(reason);
});
```

Set

ES6提供了新的数据结构**Set (集合)**。它类似于数组，但成员的值都是唯一的，

集合实现了 Iterator 接口，所以可以使用**扩展运算符**和**for...of**进行遍历，

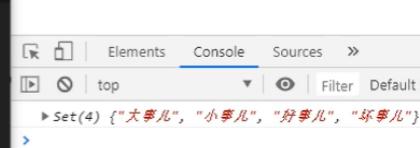
集合的属性和方法：

- 1) **size** 返回集合的元素个数
- 2) **add** 增加一个新元素，返回当前集合
- 3) **delete** 删除元素，返回 boolean 值
- 4) **has** 检测集合中是否包含某个元素，返回 boolean 值

通过new Set()构造函数来创建一个set

同时可以在创建set的时候附初始值，初始值是一个可迭代数据，一般的是一个数组，并且会**自动去重**

```
//声明一个 set
let s = new Set();
let s2 = new Set(['大事儿','小事儿','好事儿','坏事儿','小事儿']);
console.log(s2);
```



set的一些方法

size: 得到set中数组长度

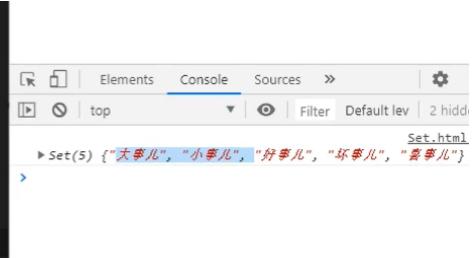
```
//声明一个 set
let s = new Set();
let s2 = new Set(['大事儿','小事儿','好事儿','坏事儿','小事儿']);

//元素个数
console.log(s2.size);
// console.log(s2);
```

add: 往数组中添加一个元素

```
//声明一个 set
let s = new Set();
let s2 = new Set(['大事儿','小事儿','好事儿','坏事儿','小事儿']);

//元素个数
// console.log(s2.size);
//添加新的元素
s2.add('喜事儿');
console.log(s2);
```



has: 检测集合中有没有相应的元素，存在的话就返回一个true，不存在true

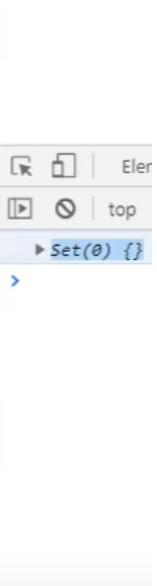
```
//声明一个 set
let s = new Set();
let s2 = new Set(['大事儿','小事儿','好事儿','坏事儿','小事儿']);

//元素个数
// console.log(s2.size);
//添加新的元素
// s2.add('喜事儿');
//删除元素
// s2.delete('坏事儿');
//检测
console.log(s2.has('好事儿'));
```

clear: 清空集合

```
//声明一个 set
let s = new Set();
let s2 = new Set(['大事儿','小事儿','好事儿','坏事儿','小事儿']);

//元素个数
// console.log(s2.size);
//添加新的元素
// s2.add('喜事儿');
//删除元素
// s2.delete('坏事儿');
//检测
// console.log(s2.has('糟心事'));
//清空
s2.clear();
console.log(s2);
```



set实践

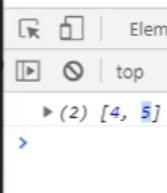
数组去重

```
let arr = [1,2,3,4,5,4,3,2,1];
//1. 数组去重
let result = [...new Set(arr)];
console.log(result);
```

取交集

这里着重使用了集合的知识，其实也可以使用数组的知识解决

```
//2. 交集
let arr2 = [4,5,6,5,6];
let result = [...new Set(arr)].filter(item => {
  let s2 = new Set(arr2); // 4 5 6
  if(s2.has(item)){
    return true;
  }else{
    return false;
  }
});
console.log(result);
```



简化

```
let result = [...new Set(arr)].filter(item => new Set(arr2).has(item));
console.log(result);
```

求并集

```
let union = [...new Set([...arr, ...arr2])];
console.log(union);
```

求差集

差集跟交集操作相反，在filter中选择后一个数组不存在的值

```
//4. 差集
let diff = [...new Set(arr)].filter(item => !(new Set(arr2).has(item)));
console.log(diff);
```

这两个数组谁在前面谁在后面，对于结果有影响

map

ES6提供了Map数据结构。它**类似于对象**，也是键值对的集合。

但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

Map也实现了 Iterator 接口，所以可以使用 **扩展运算符** 和 **for...of** 进行遍历，

- 1) **size** 返回 Map 的元素个数
- 2) **set** 增加一个新元素，返回当前 Map
- 3) **get** 返回键名对象的键值
- 4) **has** 检测 Map 中是否包含某个元素，返回 boolean 值
- 5) **clear** 清空集合，返回 undefined

创建方法：通过 new Map() 构造函数来创建

Map 的属性和方法：

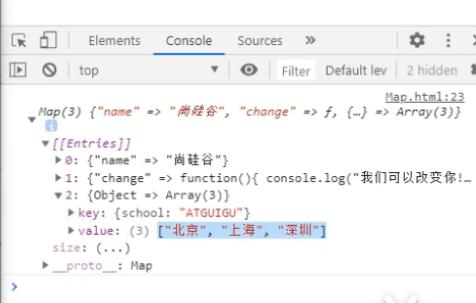
set: 添加元素，此方法有两个参数，即键值对，键的类型可以是各种类型

```
// 声明 Map
let m = new Map();

// 添加元素
m.set('name', '尚硅谷');
m.set('change', function(){
    console.log("我们可以改变你!!!");
});

let key = {
    school : 'ATGUIGU'
};
m.set(key, ['北京', '上海', '深圳']);

console.log(m);
```



size: 返回 Map 元素的个数

```
// size
console.log(m.size);
```

delete: 删除某个元素，参数是删除元素的键名

```
// 删除
m.delete('name');
```

get: 获取，参数是元素的键名

```
// 获取
console.log(m.get('change'));
```

clear: 清空，清空 Map 中所有的元素

for...of 遍历，遍历的结果是一个个数组，每个数组有两个元素，分别是键、值

```
// 清空
m.clear();
```

```
//遍历
for(let v of m){
    console.log(v);
}
```

```
▶ (2) ["name", "尚硅谷"]
▶ (2) ["change", f]
▶ (2) [{...}, Array(3)]
```

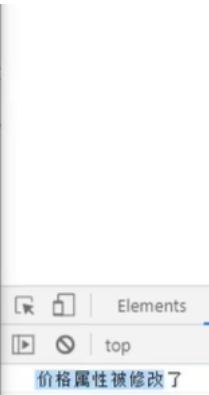
class中的setter和getter的设置

get:

get方法对属性进行绑定，当在一个属性前使用get时，**将属性写成函数形式，属性值通过返回值来定义**，此函数中含可以定义一些操作，当此**属性被读取**后，就会执行此函数中的操作

set:

在一个属性前面使用set方法，同样将属性写成函数的样子，但是**必须为函数传入一个形参**，可以**不用返回值**，在**修改属性值**的时候，就会执行set函数中的操作



```
// get 和 set
class Phone{
    get price(){
        console.log("价格属性被读取了");
        return 'iloveyou';
    }

    set price(newVal){
        console.log('价格属性被修改了');
    }
}

//实例化对象
let s = new Phone();

// console.log(s.price);
s.price = 'free';
...
```

使用场景：

get可以用在动态变化的数据上，比如求平均值、总和，数据在变化，这时候使用get可以动态的读取实时的数据

set可以用在可以在修改属性时添加一些控制和判断，比如对数值类型的限定，在set函数中可以做判断，符合要求可以赋值，不符合就不赋值

数值方法的扩展

Number. EPSILON

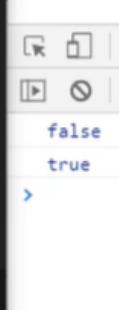
Number. EPSILON是 javascript表示的**最小精度**

EPSILON属性的值接近于 $2.228468492563136808472633361816E-16$

我们斗志到编程语言中 $0.1+0.2$ 不等于 0.3

可以利用Number. EPSILON来编写以下函数，两值相差小于Number. EPSILON，就认为相等

```
function equal(a, b){  
    if(Math.abs(a-b) < Number.EPSILON){  
        return true;  
    }else{  
        return false;  
    }  
}  
  
console.log(0.1 + 0.2 === 0.3); //  
console.log(equal(0.1 + 0.2, 0.3))
```



false
true
>

二进制和八进制

二进制以0b开头

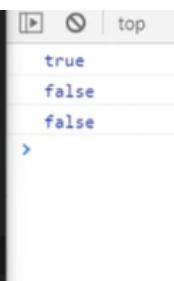
八进制以0o开头

十六进制之前就有，以0x开头

Number.isFinite

Number.isFinite检测一个数值是否是有限数

```
// let x = 0xff;  
// console.log(x);  
  
// 2. Number.isFinite 检测一个数值是否为有限数  
console.log(Number.isFinite(100));  
console.log(Number.isFinite(100/0));  
console.log(Number.isFinite(Infinity));
```



true
false
false
>

ps:js中的Infinity属性

Infinity(无穷大)在JS中是一个特殊的数字，它的特性是：它比任何有限的数字都大，如果不知道 Infinity，我们在一些运算操作遇到时，就会觉得很有意思。

现在我们来看看JS中的Infinity属性，了解用例并解决一些常见的陷阱。

1. Infinity(无穷)的定义

无穷可以分为两种，正无穷和负无穷，JS中对应的表示方式为： $+Infinity$ (或者 ∞) 和 $-Infinity$ 。

这意味着Infinity和-Infinity（小于任何有限数的数字）都是number类型的特殊值：

```
typeof Infinity; // => 'number'  
typeof -Infinity; // => 'number'
```

Infinity 是全局对象的属性：

```
window.Infinity; // => Infinity
```

另外，Number函数也有两个属性来表示正负无穷大：

```
Number.POSITIVE_INFINITY; // => Infinity  
Number.NEGATIVE_INFINITY; // => -Infinity
```

2. Infinity 的特性

Infinity比任何有限数都大。

举几个例子 Look Look:

```
Infinity > 100;           // => true  
Infinity > Number.MAX_SAFE_INTEGER; // => true  
Infinity > Number.MAX_VALUE;      // => true
```

Infinity 在加法、乘法和除法等算术运算中用作操作数时会产生有趣的效果：

```
Infinity + 1;    // => Infinity  
Infinity + Infinity; // => Infinity  
Infinity * 2;    // => Infinity  
Infinity * Infinity; // => Infinity  
Infinity / 2;    // => Infinity
```

一些Infinity 的运算得到有限的数：

```
10 / Infinity; // => 0
```

一个有限的数除以0得到 Infinity 结果：

```
2 / 0; // => Infinity
```

对无穷数进行概念上不正确的运算会得到NaN。例如，不能除以无限数，也无法确定无限数是奇数还是偶数：

```
> Infinity / Infinity; // => NaN  
>  
> Infinity % 2; // => NaN
```

2.1 负无穷

负无穷小于任何有限数。

将-Infinity 与一些有限数字进行比较：

```
-Infinity < 100; // => true  
  
-Infinity < -Number.MAX_SAFE_INTEGER; // => true  
  
-Infinity < -Number.MAX_VALUE; // => true
```

同时，负无穷小于正无穷：

```
-Infinity < Infinity; // => true
```

当使用不同操作符操作数时，也可能会得到负无穷：

```
Infinity * -1; // => -Infinity  
  
Infinity / -2; // => -Infinity  
  
-2 / 0; // => -Infinity
```

3. 判断无穷

幸运的是，Infinity等于相同符号的Infinity：

```
Infinity === Infinity; // => true  
  
-Infinity === -Infinity; // => true
```

但前面的符号不一样就不相等，就也很好理解：

```
Infinity === -Infinity; // => false
```

JSt有一个特殊的函数Number.isFinite(value)，用于检查提供的值是否有限数：

```
Number.isFinite(Infinity); // => false  
Number.isFinite(-Infinity); // => false  
Number.isFinite(999); // => true
```

\4. 无穷的的使用情况 当我们需要初始化涉及数字比较的计算时，无穷值就非常方便。例如，在数组中搜索最小值时：

```
function findMin(array) {  
  let min = Infinity;  
  for (const item of array) {  
    min = Math.min(min, item);  
  }  
  return min;  
}  
  
findMin([5, 2, 1, 4]); // => 1
```

min变量使用Infinity初始化。在第一次for()迭代中，最小值成为第一项。

5. Infinity 的一些坑

我们很可能不会经常使用Infinity值。但是，值得知道何时会出现Infinity值。

5.1. 解析数据

假设JS使用一个输入(POST请求、输入字段的值等)来解析一个数字。在简单的情况下，它会工作得很好：

```
parseFloat('10.5'); // => 10.5  
  
parseFloat('ZZZ'); // => NaN
```

这里需要小心的，parseFloat()将'Infinity'字符串解析为实际的Infinity数：

```
parseFloat('Infinity'); // => Infinity
```

另一个是使用parseInt()来解析整数，它无法将'Infinity'识别为整数：

```
parseInt('10', 10); // => 10  
  
parseInt('Infinity', 10); // => NaN
```

5.2 JSON 序列化

JSON.stringify()将Infinity数字序列化为null。

```
const worker = {  
    salary: Infinity  
};  
  
JSON.stringify(worker); // => '{ "salary": null }'
```

salary 属性值为Infinity但是当字符串化为JSON时，"salary"值将变为null。

5.3 最大数溢出

Number.MAX_VALUE是JS中最大的浮点数。

为了使用甚至大于Number.MAX_VALUE的数字，JS将该数字转换为Infinity：

```
2 * Number.MAX_VALUE; // => Infinity  
Math.pow(10, 1000); // => Infinity
```

5.4 Math函数

JS中Math命名空间的某些函数可以返回Infinity：

```
const numbers = [1, 2];  
const empty = [];  
Math.max(...numbers); // => 2  
Math.max(...empty); // => -Infinity  
Math.min(...numbers); // => 1  
Math.min(...empty); // => Infinity
```

在不带参数的情况下调用Math.max()时，返回-Infinity，而Math.min()则相应地返回Infinity。如果尝试确定一个空数组的最大值或最小值，那结果后面人感到意外。

总结

JS中的Infinity表示无穷数的概念。任何有限数均小于Infinity，而任何有限数均大于-Infinity。

比较JS中的无穷值很容易：Infinity === Infinity 为 true。特殊的函数Number.isFinite()确定提供的参数是否是一个有限的数字。

在涉及数字比较的算法时，可以使用Infinite初始化变量，用例是寻找数组的最小值。

解析来自输入的数字时，必须小心Infinity：Number('Infinity')，parseFloat('Infinity')返回实际的Infinity。当使用JSON.stringify()序列化时，Infinity变为null。

Number.isNaN()

Number.isNaN()检测一个数值是否为NaN

在es5中isNaN作为一个单独的函数，在es6中作为Number的一个方法

```
console.log(Number.isNaN(123));
```

Number.parseInt()和Number.parseFloat()

Number.parseInt Number.parseFloat字符串转整数

parseInt和parseFloat在es5里面也是单独的方法，现在作为Number的方法

```
// console.log(Number.isNaN(123));  
  
//4. Number.parseInt Number.parseFloat字符串转整数  
console.log(Number.parseInt('5211314love'));  
console.log(Number.parseFloat('3.1415926神奇'));
```

5211314
3.1415926

Number.isInteger()

判断一个数是否为整数

```
//4. Number.parseInt Number.parseFloat字符串转整数  
// console.log(Number.parseInt('5211314love'));  
// console.log(Number.parseFloat('3.1415926神奇'));  
  
//5. Number.isInteger 判断一个数是否为整数  
console.log(Number.isInteger(5));  
console.log(Number.isInteger(2.5));
```

true
false

Math.trunc()

将小数部分抹掉

```
//6. Math.trunc 将数字的小数部分抹掉  
console.log(Math.trunc(3.5));
```

Math.sign()

检测一个数是正数、负数、零

```
console.log(Math.sign(100));  
console.log(Math.sign(0));  
console.log(Math.sign(-20000));
```

正数返回1

负数返回-1

零返回0

对象方法的扩展

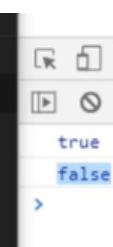
Object.is()

判断两个值是否相等，参数是要判断的两个值，相等返回true，不相等返回false

跟等号很像，但是有区别

Object.is判断两个NaN，结果是true，而等于号为false

```
// console.log(true).boolean  
console.log(Object.is(NaN, NaN)); // ===  
console.log(NaN === NaN); // !==  
// 2. Object.assign 对象的合并  
  
// 3. Object.setPrototypeOf Object.getPrototypeOf
```



Object.assign()

之前讲过这里不再讲

Object.setPrototypeOf()

设置原型对象，两个参数，后面的对象设置为前面对象的原型

```
// 3. Object.setPrototypeOf 设置原型对象 Object.getPrototypeOf  
const school = {  
    name: '尚硅谷'  
}  
const cities = [  
    xiaoqu: ['北京', '上海', '深圳']  
]  
  
Object.setPrototypeOf(school, cities);  
console.log(school);
```



The expanded object 'school' shows the following structure:

- {name: "尚硅谷"}
 - name: "尚硅谷"
 - __proto__:
 - xiaoqu: ["北京", "上海", "深圳"]
 - __proto__:
 - constructor: f Object()
 - hasOwnProperty: f hasOwnProperty
 - isPrototypeOf: f isPrototypeOf()
 - propertyIsEnumerable: f property.
 - toLocaleString: f toLocaleString
 - toString: f toString()
 - valueOf: f valueOf()
 - __defineGetter__: f __defineGett
 - __defineSetter__: f __defineSett
 - __lookupGetter__: f __LookupGett

Object.getPrototypeOf()

获取对象的原型对象

```
console.log(Object.getPrototypeOf(school));
```

模块化

1.25.2. 模块化规范产品

ES6 之前的模块化规范有：

- 1) CommonJS => NodeJS、Browserify
- 2) AMD => requireJS
- 3) CMD => seaJS

基本的已经讲过了，这里只介绍一点

在导入的模块多的时候，直接新建一个js文件，一般都是**app.js入口文件**，在app.js文件中将所有文件中都会用到的模块统一引入

然后在**入口index.html文件**中引入app.js文件即可

通过**script标签**，**type必须是module**

```
<script src="./src/js/app.js" type="module"></script>
```

babel对es6模块化代码的转换

上面写的es6模块化语法并不能对所有的浏览器实现兼容

这时候需要babel来对es6转化到es5

babel可以通过打包工具browserify或者webpack来进行下载

下面例子时使用browserify来下载的依赖（之前使用过webpack）

例子只是一个简单地示例，手动实现的文件的转换

然后进行打包上传

```
sky
<!--
  1. 安装工具 babel-cli babel-preset-env browserify(webpack)
  2. npx babel src/js -d dist/js
  3. 打包 npx browserify dist/js/app.js -o dist/bundle.js
-->
```