# ES2015

ECMAScript 6（简称ES6）是于2015年6月正式发布的JavaScript语言的标准，正式名为ECMAScript 2015（ES2015）。它的目标是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言

## 扩展运算符

扩展运算符，也就是展开语法(Spread syntax), 可以在函数调用/数组构造时, 将数组表达式或者string在语法层面展开；还可以在构造字面量对象时, 将对象表达式按key-value的方式展开。(译者注: 字面量一般指 [1, 2, 3] 或者 {name: "mdn"} 这种简洁的构造方式)

<**script**>  
 **var *arr1*** = [1, 2, 3];  
 **var *arr2*** = [4, 5, 6];  
  
 *// var arr3 = arr.concat(arr2);* **var *arr3*** = [...***arr1***, ...***arr2***, 5, 98];  
 ***console***.log(***arr3***);  
  
 **var *str*** = **'123456'** *// var str2 = str.split('');* **var *str2*** = [...***str***, **'hello'**];  
 ***console***.log(***str2***);  
  
 **var *obj*** = {  
 **a**: 1,  
 **b**: 2,  
 **c**: 3  
 }  
 **var *obj2*** = { ...***obj***, **d**: **'4'** };  
 ***console***.log(***obj2***);  
</**script**>

求一组数字的最大值：不支持直接把数组作为Math.max的参数，所以使用扩展运算符：

<**script**>  
 **var *arr*** = [1,22,31,5,56]  
 ***console***.log(***Math***.max(...***arr***));  
</**script**>

剩余语法(Rest syntax) 看起来和展开语法完全相同，不同点在于, 剩余参数用于解构数组和对象。从某种意义上说，剩余语法与展开语法是相反的：展开语法将数组展开为其中的各个元素，而剩余语法则是将多个元素收集起来并“凝聚”为单个元素。

## 解构赋值

ES6解构赋值语法：用来将数组中的值或对象中的属性取出来区分为不同变量

### 对象的解构赋值

<**script**>  
 **let *obj*** = {  
 **name**: **'小明'**,  
 **age**: 33  
 };  
  
 ***console***.log(***obj***.**name**, ***obj***.**age**);  
  
 **let** { ***name***, ***age*** } = ***obj***;  
 ***console***.log(***name***, ***age***);  
</**script**>

注意花括号中必须是属性名，不能是自己命名的变量，不能是：let {a,b}=obj  
也就是说键名和变量名是一致的

如果需要重命名，也就是设置别名，使用冒号：

<**script**>  
 **let *obj*** = {  
 **name**: **'小明'**,  
 **age**: 33  
 };  
  
 **let** { **name**: ***str*** } = ***obj***;  
 ***console***.log(***str***);  
</**script**>

设置默认值，使用=号：

<**script**>  
 **let *obj*** = { **name**: **'张三'**, **age**: 19 , **c**: **undefined** }  
 **let** { ***c*** } = ***obj***;  
 ***console***.log(***c***); *// undefined*</**script**>

<**script**>  
 **let *obj*** = { **name**: **'张三'**, **age**: 19 , **c**: 100 }  
 **let** { ***c*** } = ***obj***;  
 ***console***.log(***c***); *// 100*</**script**>

<**script**>  
 **let *obj*** = { **name**: **'张三'**, **age**: 19 , **c**: **undefined** }  
 **let** { ***c*** = 100 } = ***obj***;  
 ***console***.log(***c***); *// 100*</**script**>

<**script**>**let *obj2*** = {  
 **name**: **'小明'**,  
 **age**: 33,  
 **d**: **false** };  
 **let** { ***d*** = 100 } = ***obj2***;  
 ***console***.log(***d***); *// false*</**script**>

------------

同时使用别名和默认值：

<**script**>  
 **let *obj*** = { **name**: **'张三'**, **age**: 19 , }  
 **let** { **gender**:***g***=**'girl'** } = ***obj***;  
 ***console***.log(***g***); *// girl*</**script**>

<**script**>  
 **let *obj*** = { **name**: **'张三'**, **age**: 19 , **gender**:**'boy'** }  
 **let** { **gender**:***g***=**'girl'** } = ***obj***;  
 ***console***.log(***g***); *// 'boy'*</**script**>

支持剩余参数：

<**script**>  
 **let *obj*** = {  
 **name**: **'xiaoming'**,  
 **age**: 25,  
 **sex**: **'man'**,  
 **height**: 170  
 }  
 **let** { ***name***, ***age***, ...***r*** } = ***obj***;  
 ***console***.log(***name***, ***age***, ***r***);  
</**script**>

### 数组的解构赋值

<**script**>  
 **var *arr*** = [**'张三'**,**'李四'**,**'王五'**,**'赵六'**];  
 **var** [***a***,***b***,***c***] = ***arr***;  
 ***console***.log(***a***,***b***,***c***); *// 张三 李四 王五*</**script**>

数组的解构赋值是严格按照顺序的，不想取的值，可以留空占位：  
支持设置默认值

<**script**>  
 **var *arr*** = [**'张三'**,**'李四'**,**'王五'**,**'赵六'**];  
 ***arr***[0] = **undefined**;  
 **var** [***a***=**'hello'**,***b***, ,***d***] = ***arr***; *//如果第0个不存在，则取默认值* ***console***.log(***a***,***b***,***d***); *// hello 李四 赵六*</**script**>

支持剩余参数

<**script**>  
 **var *arr*** = [**'张三'**,**'李四'**,**'王五'**,**'赵六'**];  
 **var** [***a***,***b***,...***r***] = ***arr***;  
 ***console***.log(***a***,***b***,***r***); *// 张三 李四 (2) ["王五", "赵六"]*</**script**>

## 对象的新增

### 新写法

<**script**>  
  
 *// ES5对象写法：* **var *a*** = 1;  
 **var *objes5*** = {  
 **a**: ***a***,  
 fn: **function**(){  
 ***console***.log(**'hello es5'**);  
 }  
 };  
  
 **var *objes6*** = {  
 ***a***,  
 fn(){  
 ***console***.log(**'hello es6'**);  
 }  
 }  
  
 ***console***.log(***objes5***);  
 ***objes5***.fn();  
  
 ***console***.log(***objes6***);  
 ***objes6***.fn();  
</**script**>

### 对象的键名可以是变量

<**script**>  
  
 *// ES6对象的键名可以是变量，注意放到中括号里面：* **var *attrname*** = **'width'**;  
 **var *attrobj*** = {};  
 **var *attrfn*** = **'fn'**;  
 **var *obj*** = {  
 [***attrname***]: 100, *// 支持* [***attrobj***]: **"hello"**, *// 不支持对象作为键名* [***attrfn***](){ *// 支持* ***console***.log(**'hello es6'**);  
 }  
 };  
  
 ***console***.log(***obj***);  
 ***obj***.fn();  
</**script**>

### Object.is()方法

Object.is() 方法判断两个值是否为同一个值。

<**script**>  
 ***console***.log(1 == **'1'**); *// true* ***console***.log(***NaN*** === ***NaN***); *// false* ***console***.log(***Object***.is(1,1)); *// true* ***console***.log(***Object***.is(1,**'1'**)); *// false* ***console***.log(***Object***.is(***NaN***, ***NaN***)); *// true* ***console***.log(***Object***.is({},{})); *// false* **var *obj*** = {};  
 ***console***.log(***Object***.is(***obj***,***obj***)); *// true*</**script**>

### Object.assign()

Object.assign() 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。  
如果目标对象中的属性具有相同的键，则属性将被源对象中的属性覆盖。后面的源对象的属性将类似地覆盖前面的源对象的属性。

<**script**>  
 **const *target*** = { **a**: 1, **b**: 2 };  
 **const *source*** = { **b**: 4, **c**: 5 };  
  
 **const *returnedTarget*** = ***Object***.assign(***target***, ***source***);  
  
 ***console***.log(***target***);  
 *// expected output: Object { a: 1, b: 4, c: 5 }* ***console***.log(***returnedTarget***);  
 *// expected output: Object { a: 1, b: 4, c: 5 }*</**script**>

### 示例

传统写法：

<**script**>  
 **function** *move*(obj) {  
 **var** defaultObj = {  
 **ease**: **'linear'**,  
 **duration**: 2000  
 };  
 **var** para = {  
 **ease**: obj.**ease** || defaultObj.**ease**,  
 **duration**: obj.**duration** || defaultObj.**duration** };  
 ***console***.log(para);  
 }  
 *move*({  
 **ease**: **'easeIn'**,  
 *// duration: 1000* })  
</**script**>

ES6写法：

<**script**>  
 **function** *move*(obj) {  
 **var** defaultObj = {  
 **ease**: **'linear'**,  
 **duration**: 2000  
 };  
 **var** para = {};  
 ***Object***.assign(para, defaultObj, obj,)  
 ***console***.log(para);  
 }  
 *move*({  
 **ease**: **'easeIn'**,  
 *// duration: 1000* })  
</**script**>

## Symbol()

数据类型 “symbol” 是一种原始数据类型，这个类型的值可以用来创建匿名的对象属性。

在 JavaScript 运行时环境中，一个符号类型值可以通过调用函数 Symbol() 创建，这个函数动态地生成了一个匿名，唯一的值。Symbol类型唯一合理的用法是用变量存储 symbol的值，然后使用存储的值创建对象属性。

内置函数 “Symbol()” 是一个不完整的类，当作为函数调用时会返回一个 symbol 值，试图通过语法 “new Symbol()” 作为构造函数调用时会抛出一个错误。

### "Symbol" 值表示唯一的标识符

let s1 = Symbol()

由于Symbol是一种基础数据类型，所以当我们使用typeof去检查它的类型的时候，它会返回一个属于自己的类型symbol，而不是什么string、object之类的：

typeof s1 // 'symbol'

ES6给我们带来一种全新的数据类型：Symbol => 解决对象的属性名冲突

我们定义两个symbol类型的变量sm1，sm2，然后用全等符号===进行比较，得到的是false。也就是他们都是独一无二的值，并不相等。

每个Symbol实例都是唯一的。因此，当你比较两个Symbol实例的时候，将总会返回false：

let s1 = Symbol()  
let s2 = Symbol()  
let s3 = Symbol()

s1 === s2 // false  
s2 === s3 // false

### 描述信息

两个不一样的值，控制台输出的一样，这样无疑给我们开发调试带来一定的不便，你也可以在调用Symbol()函数时传入一个可选的字符串参数，相当于给你创建的Symbol实例一个描述信息：

let s2 = Symbol('another symbol')

<**script**>  
 **let *sm1*** = ***Symbol***(**'sm1'**);  
 **let *sm2*** = ***Symbol***(**'sm2'**);  
 ***console***.log(***sm1*** === ***sm2***);  
 ***console***.log(***sm1***,***sm2***);  
</**script**>

用字符串sm1和sm2作为参数，结果打印出来的变量sm1和sm2就是Symbol(sm1)和Symbol(sm2)，等于加上了描述，很容易区分出来。

需要注意的是，即使参数一样，描述一样，得到的两个值也是不相等的

Symbol 保证是唯一的。即使我们创建了许多具有相同描述的Symbol，它们的值也是不同。描述只是一个标签，不影响任何东西。

### Symbol 不会被自动转换为字符串

let id = Symbol("id");

alert(id); // 报错：无法将 Symbol 值转换为字符串。

如果我们真的想显示一个 Symbol，我们需要在它上面调用 .toString()，如下所示：

let id = Symbol("id");

alert(id.toString()); // Symbol(id)，现在它有效了

或者获取 symbol.description属性，只显示描述（description）：

let id = Symbol("id");

alert(id.description); // id

### 字面量中的 Symbol

如果我们要在对象字面量 {...} 中使用 Symbol，则需要使用方括号把它括起来。

就像这样：

<**script**>  
 **let *id*** = ***Symbol***(**"id"**);  
 **let *user*** = {  
 **name**: **"John"**,  
 [***id***]: 123 *// 而不是 "id"：123* };  
 ***console***.log(***user***.**id**); *// undefined* ***console***.log(***user***[***id***]); *// 123*</**script**>

把一个symbol类型的值作为对象的属性名的时候，一定要用中括号[ ]，不能用点运算符，因为用点运算符的话，会导致javascript把后面的属性名为理解为一个字符串类型，而不是symbol类型。

当symbol值作为对象的属性名的时候，不能用点运算符获取对应的值。

### 其他类型的属性键被强制为字符串

根据规范，对象的属性键只能是字符串类型或者 Symbol 类型。

我们只能在对象中使用字符串或 symbol 作为键，其它类型会被转换为字符串。

例如，在作为属性键使用时，数字 0变成了字符串"0"：

<**script**>  
 **let *obj*** = {  
 0: **"test"** *// 和 "0": "test" 一样* };  
  
 *// 两个 alert 都访问相同的属性（Number 0 被转换为字符串 "0"）  
 alert*( ***obj***[**"0"**] ); *// test  
 alert*( ***obj***[0] ); *// test（同一个属性）*</**script**>

### 把symbol作为键名

使用普通的字符串作为键名容易被修改掉：

<**script**>  
 **let *obj*** = {  
 **a**:1,  
 say(){  
 ***console***.log(**'hello1'**);  
 }  
 }  
  
 *// obj.a = 2;  
 // obj.say = () => {  
 // console.log('hello2');  
 // }  
 // console.log(obj);  
 // obj.say();* **let *a*** = ***Symbol***(**'a'**)  
 *alert*(***a***)  
 ***obj***[***a***] = 2  
  
 **let *say*** = ***Symbol***(**'sayfn'**);  
 ***obj***[***say***] = () => {  
 ***console***.log(**'hello2'**);  
 }  
  
 ***console***.log(***obj***[***a***]);  
 ***obj***[***say***]();  
  
  
 *// 注意普通字符串作为键名的访问写法：  
 // console.log(obj.a);  
 // console.log(obj['a']);*</**script**>

### 属性名的遍历

要取得对象上所有可枚举的实例属性，可以使用ECMAScript5的Object.keys()方法。这个方法接收一个对象作为参数，返回一个包含所有可枚举属性的字符串数组。

当symbol类型的值作为属性名的时候，Symbol 属性不参与for...in，也不会被Object.keys( )获取到。我们来看案例：

如果我们硬是想要获取symbol类型的属性怎么办？我们可以用一个方法：Object.getOwnPropertySymbols( )，它会找到symbol类型的属性并且返回一个数组，数组的成员就是symbol类型的属性值，

这样的话，获取字符串类型的属性和获取symbol类型的属性要分开两种不同的方式来获取，难免有有时候会很不方便，有木有什么办法让我们一次性获取所有类型的属性，不管它是字符串类型还是symbol类型呢？

有的，我们可以用Reflect.ownKeys( )方法实现：

<**script**>  
 **let *name*** = ***Symbol***(**"name"**);  
 **let *person*** = {  
 [***name***]: **'张三'**,  
 **age**: 12  
 };  
 **for** (**const *key* in *person***) {  
 ***console***.log(***key***); *// age* }  
 ***console***.log(***Object***.keys(***person***)); *// ["age"]  
  
 // 使用ES6的方法遍历：* ***console***.log(***Object***.getOwnPropertySymbols(***person***)); *// [Symbol(name)]* ***console***.log(***Reflect***.*ownKeys*(***person***)); *// ["age", Symbol(name)]*</**script**>

我们将对象person传入Reflect.ownKeys( )函数中，函数就会给我们返回一个数组，数组的内容便是对象的属性，包括symbol类型和字符串类型。

相反，Object.assign会同时复制字符串和 symbol 属性：

<**script**>  
 **let *id*** = ***Symbol***(**"id"**);  
 **let *user*** = {  
 [***id***]: 123  
 };  
  
 **let *clone*** = ***Object***.assign({}, ***user***);  
  
 *alert*( ***clone***[***id***] ); *// 123*</**script**>

### symbol本身无法运算和拼接

symbol类型本身无法运算和字符串拼接，但是symbol的键值可以：

<**script**>  
 **var *obj*** = {  
 **a**:1  
 }  
  
 **var *a*** = ***Symbol***(**'a'**)  
 ***obj***[***a***] = 100  
 **var *b*** = ***Symbol***(**'b'**)  
 ***obj***[***b***] = **'bbb'  
  
 *console***.log(***a*** + 1); *// 报错* ***console***.log(***a*** + **'hello'**); *// 报错* ***console***.log(***obj***[***a***]+1);  
 ***console***.log(***obj***[***b***] + **'hello'**);  
  
 *// 注意普通字符串作为键名的访问写法：* ***console***.log(***obj***.**a**);  
 ***console***.log(***obj***[**'a'**]);  
</**script**>

### Symbol()用来代替魔法数字

如下案例：我们以前都是使用+ -等字符串表示的，那么平均数用什么字符呢：你可能会使用 ！ ，但是过了一段时间你就不知道！表示什么了

<**script**>**const *JIA*** = ***Symbol***(**'+'**);  
 **const *JIAN*** = ***Symbol***(**'-'**);  
 **const *CHENG*** = ***Symbol***(**'\*'**);  
 **const *CHU*** = ***Symbol***(**'/'**);  
 **const *AVG*** = ***Symbol***(**'AVG'**);  
  
 ***console***.log(*calc*(***CHENG***, 10, 20));  
  
 **function** *calc*(type, a, b) {  
 **switch** (type) {  
 **case *JIA***:  
 **return** a + b  
 **break**;  
 **case *JIAN***:  
 **return** a - b  
 **break**;  
 **case *CHENG***:  
 **return** a \* b  
 **break**;  
 **case *CHU***:  
 **return** a / b  
 **break**;  
 **case *AVG***:  
 **return** (a + b) / 2  
 }  
 }  
  
</**script**>

## Set对象

ES6提供了新的对象类型Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

Set对象是值的集合，你可以按照插入的顺序迭代它的元素。 Set中的元素只会出现一次，即 Set 中的元素是唯一的。

new Set() 构造函数可以接受一个数组（或类似数组的对象）作为参数，用来初始化。

<**script**>  
 **let** s = **new** Set([**'a'**, **'b'**, **'c'**])  
 console.log(s);  
 console.log(s.size);  
 console.log(**typeof** s);  
 console.log(s.constructor);  
</**script**>

### add(value)方法

添加某个值，返回的是set对象本身，所以可以继续调用add方法

<**script**>  
 **let *s*** = **new *Set***([**'a'**, **'b'**, **'c'**])  
 ***s***.add(1).add(2)  
 ***s***.add(**'a'**) *// 无效，值唯一，不能重复添加* ***s***.add(***NaN***).add(***NaN***); *// 可以检测到多个NaN，不会重复添加* ***console***.log(***s***);  
</**script**>

### delete(value)方法

删除某个值，返回一个布尔值，表示删除是否成功。

<**script**>  
 **let *s*** = **new *Set***([**'a'**, **'b'**, **'c'**])  
 **let *res*** = ***s***.delete(**'b'**)  
 **let *res2*** = ***s***.delete(**'z'**)  
 ***console***.log(***s***, ***res***, ***res2***);  
</**script**>

### has(value)方法

返回一个布尔值，表示该值是否为Set的成员。

console.log(s.has(2)); // true

### clear()方法

清除所有成员，没有返回值

s.clear(); // 清空所有

### 遍历set对象

可以使用forEach()方法遍历set对象

<**script**>  
 **let *s*** = **new *Set***([**'a'**, **'b'**, **'c'**])  
 ***s***.forEach((item,index,set) => {  
 ***console***.log(item,index,set); *// 结果：键和值是一样的* });  
</**script**>

### key()方法

与values()方法相同，返回一个新的迭代器对象，该对象包含Set对象中的按插入顺序排列的所有元素的值。

把set结构当中所有项按照插入顺序打印出来

<**script**>  
 **let** s = **new** Set([**'a'**, **'b'**, **'c'**])  
 **let** keys = s.keys();  
 *// console.log(keys); // 这个实例上有个next()方法* console.log(keys.next());  
 console.log(keys.next());  
 console.log(keys.next());  
 console.log(keys.next());  
</**script**>

### values()方法

返回一个新的迭代器对象，该对象包含Set对象中的按插入顺序排列的所有元素的值。

<**script**>  
 **let *s*** = **new *Set***([**'a'**, **'b'**, **'c'**])  
 **let *values*** = ***s***.values();  
 *// console.log(values); // 这个实例上有个next()方法* ***console***.log(***values***.next());  
 ***console***.log(***values***.next());  
 ***console***.log(***values***.next());  
 ***console***.log(***values***.next());  
</**script**>

因为set对象的key和value值是一样的，所以keys()方法和values()方法返回的结果是一样的。

### entries()方法

返回一个新的迭代器对象，该对象包含Set对象中的按插入顺序排列的所有元素的值的[value, value]数组。为了使这个方法和Map对象保持相似， 每个值的键和值相等。

<**script**>  
 **let *s*** = **new *Set***([**'a'**, **'b'**, **'c'**])  
 **let *entries*** = ***s***.entries();  
 *// console.log(entries); // 这个实例上有个next()方法* ***console***.log(***entries***.next());  
 ***console***.log(***entries***.next());  
 ***console***.log(***entries***.next());  
 ***console***.log(***entries***.next());  
</**script**>

### 数组去重

利用set对象值唯一的特点，可以数组去重：

<**script**>  
 **let *arr*** = [11,22,11,32,22,4]  
 **let *s*** = **new *Set***(***arr***)  
 ***console***.log(***s***);  
</**script**>

## Map对象

它是键值对的集合，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。

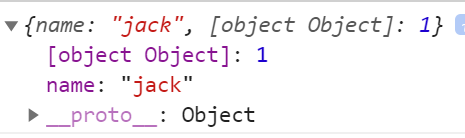
实例属性和方法：size、set、get、has、delete、clear

遍历方法：forEach（）

### 把一个对象作为另一个对象的key

<**script**>  
 **let** obj1 = {  
 name:**'jack'** };  
 **let** obj2 = {  
 name: **"rose"** };  
 *// 把obj2作为obj1的key* obj1[obj2] = 1;  
 console.log(obj1);  
 console.log(obj2.toString()); *// '[object Object]'*</**script**>

结果：键名变成了字符串：[object Object] ，因为内部调用了 toString()方法：



可以使用map对象来解决这个问题：

<**script**>  
 **let *map*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***console***.log(***map***); *// {"name" => "小明"}* ***console***.log(***map***.**size**); *// 1* ***console***.log(***map***.get(**'name'**)); *// "小明"* ***map***.set(**"age"**,100)  
 ***console***.log(***map***); *// {"name" => "小明", "age" => 100}* **var *obj*** = {**name**:**'张三'**}  
 ***map***.set(***obj***, 1) *// 把对象作为键* ***console***.log(***map***);  
</**script**>

### delete()方法

删除某个值，返回一个布尔值，表示删除是否成功。

<**script**>  
 **let *map*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***map***.set(**"age"**,100)  
 **var *res*** = ***map***.delete(**'name'**)  
 ***console***.log(***map***,***res***);  
</**script**>

### has()方法

返回一个布尔值，表示该值是否为Map对象的成员。

<**script**>  
 **let *map*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***map***.set(**"age"**,100)  
 ***console***.log(***map***.has(**'age'**));  
</**script**>

### clear()方法

<**script**>  
 **let *map*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***map***.set(**"age"**,100)  
 ***map***.clear()  
 ***console***.log(***map***);  
</**script**>

### 遍历

<**script**>  
 ***console***.log(***window***);  
 **let *map*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***map***.set(**"age"**,100)  
 ***map***.forEach((value,key,m) => {  
 ***console***.log(value,key,m);  
 });  
</**script**>

### keys()方法

<**script**>  
 **let *m*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***m***.set(**"age"**,100)  
 **var *ms*** = ***m***.keys();  
 ***console***.log(***ms***.next());  
 ***console***.log(***ms***.next());  
 ***console***.log(***ms***.next());  
</**script**>

### values()方法

<**script**>  
 **let *m*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***m***.set(**"age"**,100)  
 **var *res*** = ***m***.values();  
 ***console***.log(***res***.next());  
 ***console***.log(***res***.next());  
 ***console***.log(***res***.next());  
</**script**>

### entries()方法

<**script**>  
 **let *m*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 ***m***.set(**"age"**,100)  
 **var *res*** = ***m***.entries();  
 ***console***.log(***res***.next());  
 ***console***.log(***res***.next());  
 ***console***.log(***res***.next());  
</**script**>

### 注意

<**script**>  
 **let *m*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 **var *obj*** = {};  
 ***m***.set({},1)  
 ***m***.set({},2)  
 ***console***.log(***m***); *// 有3对  
 // 不能使用get方法获取，因为get中的{}是一个新对象，不是指set对象中的那个对象  
 // 两个对象的地址不一样* ***console***.log(***m***.get({})); *// undefined*</**script**>

<**script**>  
 **let *m*** = **new *Map***([ [**'name'**,**'小明'**] ]);  
 **var *obj*** = {};  
 ***m***.set(***obj***,1)  
 ***m***.set(***obj***,2)  
 ***console***.log(***m***); *// 有2对, 下面的obj把上面的覆盖了  
 // 可以使用get方法获取，因为get中的obj就是指向set对象中的那个对象  
 // 两个对象的地址一样* ***console***.log(***m***.get(***obj***)); *// undefined*</**script**>

## 遍历接口

遍历接口也就是遍历器 iterator

iterator是一种接口，为不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费。

Iterator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，当使用for...of循环遍历某种数据结构时，该循环会自动去寻找 Iterator 接口。一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。

ES6中提供了一些具备原生iterator接口的数据结构(包括Array、Map、Set、String、函数的arguments对象、NodeList对象。不包括Object)

对象（Object）之所以没有默认部署 Iterator 接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。

字符串是一个类似数组的对象，也原生具有 Iterator 接口。并不是所有类似数组的对象都具有 Iterator 接口，一个简便的解决方法，就是使用Array.from方法将其转为数组。

如果想要用for...of来循环对象，一种解决方法是，使用Object.keys方法将对象的键名生成一个数组，然后遍历这个数组。（Object.keys(obj) 把对象的属性放到一个数组中，Object.values(obj)把对象的属性值放到一个数组中， Object.entries(obj) 把对象的键值放到一个数组中 ）

<**script**>  
 **for** (**var *key* of *Object***.keys(obj)) {  
 ***console***.log(***key*** + **': '** + obj[***key***]);  
 }  
</**script**>

使用for...of支持直接解构：

<**script**>  
 **let *obj*** = {  
 **a**:1,  
 **b**:2  
 };  
 **for** (**const** [***key***,***value***] **of *Object***.entries(***obj***)) {  
 ***console***.log(***key***,***value***);  
 }  
</**script**>

JavaScript 原有的for...in循环，只能获得对象的键名，不能直接获取键值。ES6 提供for...of循环，允许遍历获得键值。for...in循环读取键名，for...of循环读取键值。如果要通过for...of循环，获取数组的索引，可以借助数组实例的entries方法和keys方法（所以之前看到的很多博客上说for...of遍历拿不到数组的索引是不正确的）。同样，如果想要通过for...in循环获得对象的键值也是有办法的，如下面的代码。

<**script**>  
 **let *es6*** = {  
 **name**:**"Clarence\_W"**,  
 **age**:25,  
 **job**:**"coder"** }  
  
 **for**(**let *item* in *es6***) {  
 ***console***.log(***item***,***es6***[***item***]);  
 }  
  
 *//name Clarence\_W  
 //age 25  
 //job coder*</**script**>

说说for...in的几个缺点：

1、如果用for...in循环数组的话，返回的是索引值，并且索引值是字符串类型的数字；

2、for...in循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。

3、在一些情况下，遍历顺序有可能不是按照实际数组的内部顺序；

所以综上几个缺点，for...in可以遍历数组，但是最好不要用它遍历数组，它更适合遍历对象。

吧啦吧啦整了这么多好像都是在说for...of有多好，对比起来确实比之前的循环有优势，比如说forEach不能跳出循环，for...of则可以与break、continue和return配合使用。

### 查看遍历接口

看一个数据上面是否被部署了遍历接口，有部署接口就可以使用keys ， values， entries进行遍历

<**script**>  
 **var *arr*** = [**'a'**,**'b'**,**'c'**]  
 ***console***.log(***arr***[***Symbol***.**iterator**]);  
  
 **var *k*** = str.keys();  
 ***console***.log(***k***.next());  
 ***console***.log(***k***.next());  
 ***console***.log(***k***.next());  
 ***console***.log(***k***.next());  
</**script**>

### 哪些对象部署了iterator接口？

数组，类数组，Set和Map对象

类似数组的东西也有iterator接口，也可以使用for...of遍历

<**ul**>  
 <**li**>111</**li**>  
 <**li**>111</**li**>  
 <**li**>111</**li**>  
</**ul**>  
<**script**>  
 **var *li*** = ***document***.querySelectorAll(**'li'**);  
 ***console***.log(***li***[***Symbol***.**iterator**]);  
 **for** (**const *attr* of *li***) {  
 ***console***.log(***attr***);  
 }  
</**script**>

## for...of

如果部署了iterator遍历接口，就可以使用for...of遍历

for...of语句在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环，调用自定义迭代钩子，并为每个不同属性的值执行语句

语法：

for (variable of iterable) {

//statements

}

variable：在每次迭代中，将不同属性的值分配给变量。  
iterable：被迭代枚举其属性的对象。

<**script**>  
 **var *arr*** = [**'a'**, **'b'**, **'c'**]  
 **for** (**const *attr* of *arr***) {  
 ***console***.log(***attr***);  
 }  
</**script**>

### 迭代数组

<**script**>  
 **let *iterable*** = [10, 20, 30];  
  
 **for** (**let *value* of *iterable***) {  
 ***value*** += 1;  
 ***console***.log(***value***);  
 }  
</**script**>

结果：11,21,31

### 迭代字符串

<**script**>  
 **let *iterable*** = **"boo"**;  
  
 **for** (**let *value* of *iterable***) {  
 ***console***.log(***value***);  
 }  
</**script**>

### 迭代Set

使用for...of，Set 结构遍历时，返回的是一个值，而 Map 结构遍历时，返回的是一个数组，该数组的两个成员分别为当前 Map 成员的键名和键值。

<**script**>  
 **let *iterable*** = **new *Set***([1, 1, 2, 2, 3, 3]);  
  
 **for** (**let *value* of *iterable***) {  
 ***console***.log(***value***);  
 }  
 *// 1  
 // 2  
 // 3*</**script**>

### 迭代Map

<**script**>  
 **let *iterable*** = **new *Map***([[**"a"**, 1], [**"b"**, 2], [**"c"**, 3]]);  
  
 **for** (**let *entry* of *iterable***) {  
 ***console***.log(***entry***);  
 }  
 *// ["a", 1]  
 // ["b", 2]  
 // ["c", 3]* **for** (**let** [***key***, ***value***] **of *iterable***) {  
 ***console***.log(***value***);  
 }  
 *// 1  
 // 2  
 // 3*</**script**>

### 迭代arguments

<**script**>  
 (**function** () {  
 **for** (**let** argument **of** arguments) {  
 ***console***.log(argument);  
 }  
 })(1, 2, 3);  
  
 *// 1  
 // 2  
 // 3*</**script**>

### 迭代DOM集合

<**script**>  
**let *articleParagraphs*** = ***document***.querySelectorAll(**"article > p"**);  
  
 **for** (**let *paragraph* of *articleParagraphs***) {  
 ***paragraph***.**classList**.add(**"read"**);  
 }  
</**script**>

### for...of与for...in的区别

两者都是遍历的方法，前者是遍历键名，后者是遍历键值。

看一下两者的定义：

for...in 遍历当前对象的所有可枚举属性(包括自有属性，从原型继承的属性)，  
for...of 遍历当前可遍历(iterable)对象拥有的可迭代元素(iterator)，是ES6提供的新方法

以下示例显示了与Array一起使用时，for...of循环和for...in循环之间的区别。

<**script**>  
 *// 从原型继承的属性* ***Object***.**prototype**.objCustom = **function** () { };  
 ***Array***.**prototype**.arrCustom = **function** () { };  
 *// 自定义属性* **let *iterable*** = [3, 5, 7];  
 ***iterable***.**foo** = **'hello'**;  
 **for** (**let *i* in *iterable***) {  
 ***console***.log(***i***); *// 0, 1, 2, "foo", "arrCustom", "objCustom"* }  
 **for** (**let *i* in *iterable***) {  
 **if** (***iterable***.hasOwnProperty(***i***)) {  
 ***console***.log(***i***); *// 0, 1, 2, "foo"* }  
 }  
 **for** (**let *i* of *iterable***) {  
 ***console***.log(***i***); *// 3, 5, 7* }  
</**script**>

# ES6模块化

一个.js文件就是一个模块

ES6之前已经出现了js模块加载的方案，最主要的是CommonJS和AMD规范。  
commonjs主要应用于服务器，实现同步加载，如nodejs。  
AMD规范应用于浏览器，如requirejs，为异步加载。  
同时还有CMD规范，为同步加载方案如seaJS。

ES6在语言规格的层面上，实现了模块功能，而且实现得相当简单，完全可以取代现有的CommonJS和AMD规范，成为浏览器和服务器通用的模块解决方案。

ES6模块主要有两个功能：export和import  
export用于对外输出本模块变量的接口  
import用于在一个模块中加载另一个含有export接口的模块。

也就是说使用export命令定义了模块的对外接口以后，其他JS文件就可以通过import命令加载这个模块（文件）。

## 对于导入

只有export default导出形式才不需要加大括号。其他导出形式一定要放在大括号里面，不然会报错

## 使用export导出

### 直接导出声明的变量或函数

b.js实现导出

**export let *myName*** = **"张三"**;  
*// export let sayHi = (value) => {  
// console.log("hello：" + value);  
// };  
// 当然也可以写成函数声明：***export function** *sayHi*(value) {  
 ***console***.log(**"hello："** + value);  
};

导入：

**import** { myName, sayHi } **from './b.js'**;  
***console***.log(myName);  
sayHi(**'你好'**);

在html页面中引入a.js

<script src="a.js" type="module"></script>

这里要特别注意：script的标签必须为type="module" ，而且必须在服务器模式下打开页面

### 把变量放到一个对象中导出

即使只导出一个变量，也必须放到大括号里面，否则报错

**let *myName*** = **"张三"**;  
**let** *sayHi* = () => {  
 ***console***.log(**"我的名字是："** + ***myName***);  
};  
**export** { ***myName***, *sayHi* }

导入：

导入的变量一定要放在大括号里面，不然会报错，{}括号内的变量来自于b.js文件export出的变量标识符。只有export default形式才不需要加大括号。import的时候必须带上.js的后缀名；

**import** { myName, sayHi } **from './b.js'**;  
***console***.log(myName);  
sayHi();

在页面中引入a.js

<script src="a.js" type="module"></script>

### 可以在导出时重新起个名字

**let *myName*** = **"张三丰"**;  
**let *age*** = 16;  
**export** { ***myName*** as newName , ***age*** }

**import** { newName, age } **from './b.js'**;  
console.log(newName, age);

### 也可以在导入变量时重新起个名字

这样可以避免和当前模块中的变量冲突

**let *myName*** = **"张三丰"**;  
**let *age*** = 16;  
**export** { ***myName*** , ***age*** }

**import** { myName, age **as** newAge } **from './b.js'**;  
***console***.log(myName, newAge);

## 使用export default导出

可以使用export default命令，为模块指定默认输出，这样就不需要知道所要加载模块的变量名。

一个文件内最多只能有一个export default。

b.js实现导出

**let *myName*** = **"张三丰"**;  
**export default *myName***

myName不能加大括号，加了以后导出就变成了：{myName: "张三丰"}

其实此处相当于为sex变量值"boy"起了一个系统默认的变量名default，自然default只能有一个值，所以一个文件内不能有多个export default。

a.js实现导入

本质上，a.js文件的export default输出一个叫做default的变量，然后import时允许你为它取任意变量名且不能用大括号包含

**import** anyname **from './b.js'**;  
**import** any **from './b.js'**;  
***console***.log(anyname,any);

直接导出一个对象

**export default** {  
 **myName**:**'孙悟空'**,  
 say: **function**(){  
 ***console***.log(**'我就是'** + **this**.**myName**);  
 }  
}

**import** obj **from './b.js'**;  
***console***.log(obj.myName);  
obj.say();

## 导入整个模块的内容

这将myModule插入当前作用域，其中包含来自位于/modules/my-module.js文件中导出的所有接口。

语法：import \* as myModule from '/modules/my-module.js';

导出：

**let *myName*** = **"张三丰"**;  
**let *age*** = 16;  
**export** { ***myName*** , ***age*** }  
**export default function** *fn*(){  
 ***console***.log(**'啦啦啦，我是默认导出的哦~'**);  
}

导入：

**import** \* **as** tool **from './b.js'**;  
***console***.log(tool.myName, tool.age);  
***console***.log(tool.**default**);  
tool.**default**();

## 导入默认和指定值

导入 默认导出值和指定导出值：

导出：

**let *myName*** = **"张三丰"**;  
**let *age*** = 16;  
**export** { ***myName*** , ***age*** }  
**export default** 1000

导入：num是默认导出值，大括号里是指定导出值，两者用逗号隔开：

**import** num, {myName, age} **from './b.js'**;  
***console***.log(myName, age);  
***console***.log(num);

## 只导入模块

导出：

**let *myName*** = **"张三丰"**;  
**let *age*** = 16;  
**export** { ***myName*** , ***age*** }  
**export default** 1000  
  
***console***.log(**'我是b.js模块'**);

只导入模块，不导入模块里面的值：

**import './b.js'**;

注意：如果导入多次（把import './b.js'多复制几次），也只执行一次 ：console.log('我是b.js模块');

## 注意

### this指向

模块中的this指向 undefined

### 值的修改

**let *num*** = 3000;  
**export** { ***num*** }

修改num的值就报错：

**import** { num } **from './b.js'**;  
console.log(num);  
num = 1000;

解决方法：

**let *num*** = 3000;  
**function** *add*() {  
 ***num*** += 100;  
}  
**export** { ***num***, *add* }

通过调用add函数实现修改num的值：

**import** { num, add } **from './b.js'**;  
add();  
***console***.log(num);

此时如果还有其他模块导入了b.js模块，那么num就是被修改后的值

### 模块引入默认加了defer

a.js里面有打印console.log(456);

<**script src="a.js" type="module"**></**script**>  
<**script**>  
 ***console***.log(123);  
</**script**>

结果：先打印123 ，后打印 456。

所以说一旦使用了模块，相当于默认加了个defer属性：

<script src="a.js" type="module" defer></script>

### 模块中的变量是私有的

a.js模块中有let num = 1000;

<**script src="a.js" type="module"**></**script**>  
<**script**>  
 ***console***.log(num);  
</**script**>

结果报错，num is not defined，因此模块中的变量需要导出export，导入import操作才能使用。

## defer

在内联js上写defer没有效果

defer：在外链js上使用，会在其他的js，dom节点加载完成之后再加载。多个defer的执行按照先后次序。

1.js  
let num = 1

2.js  
console.log(num);

如果把2.js先引入，就会报错，找不到num，解决方法就是defer：

<script src="2.js" defer></script>  
<script src="1.js"></script>