# ES6 新特性

# 作者：李振超

# 一、ES6简介

历时将近6年的时间来制定的新 ECMAScript 标准 ECMAScript 6（亦称 ECMAScript Harmony，简称 ES6）终于在 2015 年 6 月正式发布。自从上一个标准版本 ES5 在 2009 年发布以后，ES6 就一直以**新语法**、**新特性**的优越性吸引著众多 JavaScript 开发者，驱使他们积极尝鲜。

由于ES6是在2015年发布的，所以也叫ES2015。

以后ESCMAScript标准一年一更新，统一使用年份命名：ES2016、ES2017、....

> 下面开始介绍ES6常用的一些新特性：

# 二、块级作用域绑定

> 在ES5之前，不存在块级作用域，在编程的时候很多时候会带来很多的不便，ES6新增了块级作用域，补足了这方面的缺陷。

块级声明指的是该声明的变量无法被代码块外部访问。块作用域，又被称为词法作用域（lexical scopes），可以在如下的条件下创建：

- 函数内部
- 在代码块（即 { }）内部

块级作用域是很多类C语言的工作机制，ECMAScript 6 引入块级声明的目的是增强 JavaScript 的灵活性，同时又能与其它编程语言保持一致。

## 2.1 let声明

> 使用let声明变量的语法和使用var声明的语法是一样的。**但是let声明的变量的作用域会限制在当前的代码块中。这是let与var的最大区别。**

```
<script type="text/javascript">
    let a = 10;
    if(a > 5){
        console.log(b); //用let声明的变量没有声明提前这一特性，所以此处也访问不到（报错）
        let b = 20;
        console.log(b);
    }
    console.log(b); //由于b是在if块中使用let声明的，所以此处无法访问到。（报错）
</script>
```

> 注意：

1. 用 let 声明的变量具有块级作用域，只能在声明的块中访问，在块外面无法访问
2. 用let声明的变量也没有声明提前这一特性。
3. 在同一个块中，let声明的变量也不能重复声明。
4. 在声明变量的时候尽量使用let，慢慢的抛弃var

## 2.2 const声明(Constant Declarations)

> 在 ES6 使用const来声明的变量称之为常量。这意味着它们不能再次被赋值。由于这个原因，所有的 const 声明的变量都必须在声明处初始化。
>
> const声明的常量和let变量一样也是具有块级作用域的特性。

```
<script type="text/javascript">
    var a = 20;
    if (true) {
        const b = 20;
        b = 30;  //错误！常量不能重新赋值
        const c; //错误！  常量声明的同时必须赋值。
    }
</script>
```

> 注意：

1. const的特性除了声明的是常量为，其他与let一样。
2. 在let和const声明前的这段区域称之为暂存性死区（**The Temporal Dead Zone** —TDZ)。
3. 使用let和const声明的变量和常量不再是window的属性。 也就是说通过window.a是无法访问到的。

## 2.3 循环中的块级绑定

> 使用var声明的循环变量在循环结束后仍然可以访问到。 使用let声明的循环变量，在循环结束之后会立即销毁。

```
<script type="text/javascript">
    for(let i = 0; i < 3; i++){ // 循环结束之后会立即销毁 i
        console.log(i);
    }
    console.log(i);  //此处无法访问到 i 。
</script>
```

## 2.4 循环中的函数

> 看下面的代码，是输出10个10，而不是0，1，2，...

```
<script type="text/javascript">
    var funcs = [];
    for (var i = 0; i < 10; i++) {
        funcs.push(function () {
            console.log(i);
        });
    }
    funcs.forEach(function (func) {
        func();     // 输出 "10" 共10次
    });
</script>
```

> 解决办法需要使用函数的自执行特性。

```
var funcs = [];
for (var i = 0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
        }
    }(i)));
}
funcs.forEach(function(func) {
    func();     // 输出 0, 1, 2 ... 9
});
```

**如果使用let声明变量，则完全可以避免前面的问题。 这是ES6规范中专门定义的特性。在for ... in和for ... of循环中也适用**

```
<script type="text/javascript">
    var funcs = [];
    for (let i = 0; i < 10; i++) {
        funcs.push(function () {
            console.log(i);
        });
    }
    funcs.forEach(function (func) {
        func();     // 输出 0, 1, 2 ... 9
    })
</script>
```

> 说明：

1. let 声明使得每次迭代都会创建一个变量 i，所以循环内部创建的函数会获得各自的变量 i 的拷贝。每份拷贝都会在每次迭代的开始被创建并被赋值。

# 三、函数的新增特性

## 3.1 带默认参数的函数

> JavaScript函数的最大的一个特点就是在传递参数的时候，参数的个数不受限制的。为了健壮性考虑，一般在函数内部需要做一些默认值的处理。

```
function makeRequest(url, timeout, callback) {
    timeout = timeout || 2000;
    callback = callback || function() {};
}
```

其实上面的默认值方法有个bug：当timeout是0的时候也会当做假值来处理，从而给赋值默认值2000.

> ES6从语言层面面上增加了 **默认值的** 支持。看下面的代码：

```
//这个函数如果只传入第一个参数，后面两个不传入，则会使用默认值。如果后面两个也传入了参数，则不会使用默认值。
function makeRequest(url, timeout = 2000, callback = function() {}) {

    // 其余代码

}
```

## 3.2 默认参数对 arguments 对象的影响

> 在非严格模式下，arguments总是能反映出命名参数的变化。看下面的代码：

```
<script type="text/javascript">
    function foo(a, b) {
        //非严格模式
        console.log(arguments[0] === a); //true
        console.log(arguments[1] === b); //true
        a = 10;
        b = 20;
        console.log(arguments[0] === a); //true
        console.log(arguments[1] === b); //true
    }
    foo(1, 2);
</script>
```

> **在ES5的严格模式下**，arguments只反映参数的初始值，而不再反映命名参数的变化！

```
<script type="text/javascript">

    function foo(a, b) {
        //严格模式
        "use strict"
        console.log(arguments[0] === a); //true
        console.log(arguments[1] === b); //true
        a = 10;
        b = 20;
        console.log(arguments[0] === a); //false。  修改a的值不会影响到arguments[0]的值
        console.log(arguments[1] === b); //false
    }
    foo(1, 2);
</script>
```

> 当使用ES6参数默认值的时候，不管是否是在严格模式下，都和ES5的严格模式相同。看下面的代码：

```
<script type="text/javascript">

    function foo(a, b = 30) {
        console.log(arguments[0] === a); //true
        console.log(arguments[1] === b); //true
        a = 10;
        b = 20;
        console.log(arguments[0]  === a); //false。  由于b使用了默认值。虽然a没有使用默认值，但是仍然表现的和严格模式一样。
        console.log(arguments[1] === b); //false。  b使用了默认值，所以表现的和严格模式一样。
    }
    foo(1, 2);
</script>
```

> 注意：如果这样调用foo(1),则 a == 1， b == 30， arguments[0] == 1, arguments[1] == undefined。也就是说默认值并不会赋值给arguments参数。

## 3.3 默认参数表达式 (Default Parameter Expressions)

> 参数的默认值，也可以是一个表达式或者函数调用等。看下面的代码

```
<script type="text/javascript">
    function getValue() {
        return 5;
    }

    function add(first, second = getValue()) { //表示使用getValue这个函数的返回值作为second的默认值。
        return first + second;
    }

    console.log(add(1, 1));     // 2.  调用add函数的时候，传入了第二个参数，则以传入的参数为准。
    console.log(add(1));        // 6。 调用add函数的时候，没有传入第二个参数，则会调用getValue函数。
</script>
```

> 有一点需要要注意：getValue()只会在调用add且不传入第二个参数的时候才会去调用。不是在解析阶段调用的。

```
<script type="text/javascript">
    let value = 5;
    function getValue() {
        return value++;
    }

    function add(first, second = getValue()) {  //
        return first + second;
    }

    console.log(add(1, 1));     // 2
    console.log(add(1));        // 6。
    console.log(add(1));        // 7
    console.log(add(1));        // 8
</script>
```

> 由于默认值可以表达式，所以我们甚至可以使用前面的参数作为后面参数的默认值。

```
function add(first, second = first) {  // 使用第一个参数作为第二个参数的默认值
        return first + second;
 }
```

> 注意：可以把前面的参数作为后面参数的默认值，但是不能把后面的参数作为第一个参数的默认值。这可以前面说的let和const的暂存性死区一个意思。

```
function add(first = second, second)) {  // 这种写法是错误的

        return first + second;
}
```

## 3.4 未命名参数问题

> Javascript并不限制传入的参数的数量。在调用函数的时候，传入的实参的个数超过形参的个数的时候，超过的部分就成为了未命名参数。在ES5之前，我们一般可以通过arguments对象来获取到未命名参数的值。但是略显繁琐。

```
<script type="text/javascript">
    function foo(a) {
        console.log(a);
        console.log(arguments[1])  //取得传入的多余的参数。
    }
    foo(2, 3);
</script>
```

> ES6，提供了一种更加优雅处理未命名参数的问题：**剩余参数( Rest Parameters )**
>
> 语法：function a(a, ... b){ }

> 剩余参数使用三个点( ... )和变量名来表示。

```
<script type="text/javascript">
    function foo(a, ...b) {
        console.log(a);
        console.log(b instanceof Array);  //true  .多余的参数都被放入了b中。b其实就是一个数组。
    }
    foo(2, 3, 4, 6);
</script>
```

> 注意：

1. 函数最多只能有一个剩余参数b。而且这个剩余参数必须位于参数列表的最后位置。
2. 虽然有了剩余参数，但是arguments仍然存在，但是arguments完全无视了剩余参数的存在。
3. 剩余参数是在函数声明的时候出现的。

## 3.5 函数中的扩展运算符

> 例如:Math中的max函数可以返回任意多个参数中的最大值。但是如果这些参数在一个数组中，则没有办法直接传入。以前通用的做法是使用applay方法。
>
> 看下面的代码：

```
<script type="text/javascript">
    let values = [25, 50, 75, 100]
    console.log(Math.max.apply(Math, values));  // 100
</script>
```

> 上面这种方法虽然可行，但是总是不是那么直观。
>
> 使用ES6提供的扩展运算符可以很容易的解决这个问题。在数组前加前缀 ... (三个点)。

```
<script type="text/javascript">
    let values = [25, 50, 75, 100]
    console.log(Math.max(...values));  //使用扩展运算符。相当于拆解了数组了。
    console.log(Math.max(...values, 200));  //也可以使用扩展运算符和参数的混用，则这个时候就有 5 个数参与比较了。
</script>
```

> **注意：剩余参数和扩展运算符都是 使用三个点作为前缀。但是他们使用的位置是不一样的。**
>
> 1. 剩余参数是用在函数的声明的时候的参数列表中，而且必须在参数列表的后面
> 2. 扩展运算符是用在函数调用的时候作为实参来传递的，在实参中的位置没有限制。

# 四、全新的函数：箭头函数（=>）->

> ECMAScript 6 最有意思的部分之一就是箭头函数。正如其名，箭头函数由“箭头”（=>）这种新的语法来定义。
>
> 其实在别的语言中早就有了这种语法结构，不过他们叫拉姆达表达式。

# 4.1 箭头函数语法

> 基本语法如下：

```
(形参列表)=>{
  //函数体
}
```

---

> 箭头函数可以赋值给变量，也可以像匿名函数一样直接作为参数传递。

- 示例1：

```
<script type="text/javascript">
    var sum = (num1, num2) =>{
        return num1 + num2;
    }
    console.log(sum(3, 4));
    //前面的箭头函数等同于下面的传统函数
    var add = function (num1, num2) {
        return num1 + num2;
    }
    console.log(add(2, 4))
</script>
```

---

> 如果函数体内只有一行代码，则包裹函数体的 **大括号** ({ })完全可以省略。如果有return，return关键字也可以省略。
>
> 如果函数体内有多条语句，则 {} 不能省略。

- 示例2：

```
<script type="text/javascript">
    var sum = (num1, num2) => num1 + num2;
    console.log(sum(5, 4));
    //前面的箭头函数等同于下面的传统函数
    var add = function (num1, num2) {
        return num1 + num2;
    }
    console.log(add(2, 4));

    //如果这一行代码是没有返回值的，则方法的返回自也是undefined
    var foo = (num1, num2) => console.log("aaa");
    console.log(foo(3,4));  //这个地方的返回值就是undefined
</script>
```

> 如果箭头函数只有一个参数，则包裹参数的小括号可以省略。其余情况下都不可以省略。**当然如果不传入参数也不可以省略**

- 示例3：

```
<script type="text/javascript">
    var foo = a=> a+3; //因为只有一个参数，所以()可以省略
    console.log(foo(4)); // 7
</script>
```

> 如果想直接返回一个js对象，而且还不想添加传统的大括号和return，则必须给整个对象添加一个**小括号 ()**

- 示例4：

```
<script type="text/javascript">
    var foo = ()=>({name:"lisi", age:30});
    console.log(foo());
    //等同于下面的;
    var foo1 = ()=>{
        return {
            name:"lisi",
            age : 30
        };
    }
</script>
```

## 4.2 使用箭头函数实现函数自执行

```
<script type="text/javascript">
    var person = (name => {
            return {
                name: name,
                age: 30
            }
        }
    )("zs");
    console.log(person);
</script>
```

## 4.3 箭头函数中无this绑定(No this Binding)

> 在ES5之前this的绑定是个比较麻烦的问题，稍不注意就达不到自己想要的效果。因为this的绑定和定义位置无关，只和调用方式有关。
>
> **在箭头函数中则没有这样的问题，在箭头函数中，this和定义时的作用域相关，不用考虑调用方式**
>
> 箭头函数没有 this 绑定，意味着 this 只能通过查找作用域链来确定。**如果箭头函数被另一个不包含箭头函数的函数囊括，那么 this 的值和该函数中的 this 相等，否则 this 的值为 window。**

```
<script type="text/javascript">
    var PageHandler = {
        id: "123456",
        init: function () {
            document.addEventListener("click",
                event => this.doSomething(event.type), false); // 在此处this的和init函数内的this相同。
        },

        doSomething: function (type) {
            console.log("Handling " + type + " for " + this.id);
        }
    };
    PageHandler.init();
</script>
```

看下面的一段代码：

```
<script type="text/javascript">

    var p = {
        foo:()=>console.log(this)   //此处this为window
    }
    p.foo();  //输出为 window对象。    并不是我想要的。所以在定义对象的方法的时候应该避免使用箭头函数。
//箭头函数一般用在传递参数，或者在函数内部声明函数的时候使用。
</script>
```

> 说明：

1. 箭头函数作为一个使用完就扔的函数，不能作为构造函数使用。也就是不能使用new 的方式来使用箭头函数。
2. 由于箭头函数中的this与函数的作用域相关，所以不能使用call、apply、bind来重新绑定this。但是虽然this不能重新绑定，但是还是可以使用call和apply方法去执行箭头函数的。

## 4.4 无arguments绑定

> 虽然箭头函数没有自己的arguments对象，但是在箭头函数内部还是可以使用它外部函数的arguments对象的。

```
<script type="text/javascript">
    function foo() {
        //这里的arguments是foo函数的arguments对象。箭头函数自己是没有 arguments 对象的。
        return ()=>arguments[0]; //箭头函数的返回值是foo函数的第一个参数
    }
    var arrow = foo(4, 5);
    console.log(arrow()); // 4
</script>
```

# 五、对象功能的扩展

> 在JavaScript中，几乎所有的类型都是对象，所以使用好对象，对提示JavaScript的性能很重要。
>
> **ECMAScript 6 给对象的各个方面，从简单的语法扩展到操作与交互，都做了改进。**

## 5.1 对象类别

> ECMAScript 6 规范明确定义了每种对象类别。理解该术语对于从整体上认识该门语言显得十分重要。对象类别包括：

- 普通对象（ordinary object）拥有 JavaScript 对象所有的默认行为。
- 特异对象（exotic object）的某些内部行为和默认的有所差异。
- 标准对象（standard object）是 ECMAScript 6 中定义的对象，例如 Array, Date 等，它们既可能是普通也可能是特异对象。

- 内置对象（built-in object）指 JavaScript 执行环境开始运行时已存在的对象。标准对象均为内置对象。

## 5.2 对象字面量的语法扩展

### 5.2.1 简写的属性初始化

```
<script type="text/javascript">
    function createPerson(name, age) {
        //返回一个对象：属性名和参数名相同。
        return {
            name:name,
            age:age
        }
    }
    console.log(createPerson("lisi", 30)); // {name:"lisi", age:30}
    //在ES6中，上面的写法可以简化成如下形式

</script>
```

> 在ES6中，上面的写法可以简化成如下的形式：

```
<script type="text/javascript">
    function createPerson(name, age) {
        //返回一个对象：属性名和参数名相同。
        return {
            name,  //当对象属性名和本地变量名相同时，可以省略冒号和值
            age
        }
    }
    console.log(createPerson("lisi", 30)); // {name:"lisi", age:30}
</script>
```

*当对象字面量中的属性只有属性名的时候，JavaScript 引擎会在该作用域内寻找是否有和属性同名的变量。在本例中，本地变量 name 的值被赋给了对象字面量中的 name 属性。*

*该项扩展使得对象字面量的初始化变得简明的同时也消除了命名错误。对象属性被同名变量赋值在 JavaScript 中是一种普遍的编程模式，所以这项扩展的添加非常受欢迎。*

### 5.2.2 简写的方法声明

```
<script type="text/javascript">
    var person = {
        name:'lisi',
        sayHell:function () {
            console.log("我的名字是: " + this.name);
        }
    }
    person.sayHell()
</script>
```

> 在ES6中，上面的写法可以简化成如下的形式：

```
<script type="text/javascript">
    var person = {
        name:'李四',
        sayHell() {
            console.log("我的名字是: " + this.name);
        }
    }
    person.sayHell()
</script>
```

*省略了冒号和function看起来更简洁*

## 5.2.3 在字面量中动态计算属性名

> 在ES5之前，如果属性名是个变量或者需要动态计算，则只能通过 对象.[变量名] 的方式去访问。而且这种动态计算属性名的方式 **在字面量中** 是无法使用的。

```
<script type="text/javascript">
    var p = {
        name : '李四',
        age : 20
    }
    var attName = 'name';
    console.log(p[attName]) //这里 attName表示的是一个变量名。
</script>
```

> 而下面的方式使用时没有办法访问到attName这个变量的。

```
<script type="text/javascript">
    var attName = 'name';
    var p = {
        attName : '李四',  // 这里的attName是属性名，相当于各级p定义了属性名叫attName的属性。
        age : 20
    }
    console.log(p[attName])  // undefined
</script>
```

> 在ES6中，把属性名用[ ]括起来，则括号中就可以引用提前定义的变量。

```
<script type="text/javascript">
    var attName = 'name';
    var p = {
        [attName] : '李四',  // 引用了变量attName。相当于添加了一个属性名为name的属性
        age : 20
    }
    console.log(p[attName])  // 李四
</script>
```

## 5.3 新增的方法

> ECMAScript 从第五版开始避免在 Object.prototype 上添加新的全局函数或方法，转而去考虑具体的对象类型如数组）应该有什么方法。当某些方法不适合这些具体类型时就将它们添加到全局 Object 上 。
>
> ECMAScript 6 在全局 Object 上添加了几个新的方法来轻松地完成一些特定任务。

### 5.3.1 Object.is()

> 在 JavaSciprt 中当你想比较两个值时，你极有可能使用比较操作符（）或严格比较操作符（=）。许多开发者为了避免在比较的过程中发生强制类型转换，更倾向于后者。但即使是严格等于操作符，它也不是万能的。例如，它认为 +0 和 -0 是相等的，虽然它们在 JavaScript 引擎中表示的方式不同。同样 NaN === NaN 会返回 false，所以必须使用 isNaN() 函数才能判断 NaN 。

> ECMAScript 6 引入了 Object.is() 方法来补偿严格等于操作符怪异行为的过失。该函数接受两个参数并在它们相等的返回 true 。只有两者在类型和值都相同的情况下才会判为相等。如下所示：

```
console.log(+0 == -0);              // true
console.log(+0 === -0);             // true
console.log(Object.is(+0, -0));     // false

console.log(NaN == NaN);            // false
console.log(NaN === NaN);           // false
console.log(Object.is(NaN, NaN));   // true

console.log(5 == 5);                // true
console.log(5 == "5");              // true
console.log(5 === 5);               // true
console.log(5 === "5");             // false
console.log(Object.is(5, 5));       // true
console.log(Object.is(5, "5"));     // false
```

*很多情况下 Object.is() 的表现和 = 是相同的。它们之间的区别是前者* ***认为 +0 和 -0 不相等而 NaN 和 NaN 则是相同的****。不过弃用后者是完全没有必要的。何时选择 Object.is() 与 == 或 = 取决于代码的实际情况。*

## 5.3.2 Object.assign()

> 使用assign主要是为了简化对象的混入（mixin）。混入是指的在一个对象中引用另一个对象的属性或方法。
>
> assing可以把一个对象的属性和访问完整的转copy到另外一个对象中。

```
<script type="text/javascript">
    var p = {
        name : "lisi",
        age : 20,
        friends : ['张三', '李四']
    }
    var p1 = {};
    Object.assign(p1, p); //则p1中就有了与p相同的属性和方法.  p1是接受者，p是提供者
    console.log(p1);
    //这种copy是浅copy，也就是说如果属性值是对象的话，只是copy的对象的地址值(引用)
    console.log(p1.friends == p.friends);  //true   p1和p的friends同事指向了同一个数组。
    p.friends.push("王五");
    console.log(p1.friends); //['张三', '李四', '王五']
</script>
```

> assign方法可以接受任意多的提供者。意味着后面提供者的同名属性和覆盖前面提供者的属性值。

```
<script type="text/javascript">
    var p = {
        name : "lisi",
        age : 20,
        friends : ['张三', '李四']
    }
    var p1 = {
        name : 'zs',
    }
    var p2 = {};
    Object.assign(p2, p, p1); //p和p1都是提供者
    console.log(p2.name); // zs
</script>
```

# 六、字符串功能的增强

## 6.1 查找子字符串

> 在以前在字符串中查找字符串的时候，都是使用indexOf方法。
>
> ES6新增了三个方法来查找字符串。

- includes() 方法会在给定文本存在于字符串中的任意位置时返回 true，否则返回 false 。
- startsWith() 方法会在给定文本出现在字符串开头时返回 true，否则返回 false 。
- endsWith() 方法会在给定文本出现在字符串末尾时返回 true，否则返回 false 。

> 每个方法都接收两个参数：需要搜索的文本和可选的起始索引值。
>
> 当提供第二个参数后，includes() 和 startsWith() 会以该索引为起始点进行匹配，而 endsWith() 则是字符串搜索的结束位置。
>
> 若第二个参数未提供，includes() 和 startsWith() 会从字符串的起始中开始检索，endsWith() 则是从字符串的末尾。
>
> 实际上，第二个参数减少了需要检索的字符串的总量。以下是使用这些方法的演示：

```
var msg = "Hello world!";

console.log(msg.startsWith("Hello"));       // true
console.log(msg.endsWith("!"));             // true
console.log(msg.includes("o"));             // true

console.log(msg.startsWith("o"));           // false
console.log(msg.endsWith("world!"));        // true
console.log(msg.includes("x"));             // false

console.log(msg.startsWith("o", 4));        // true
console.log(msg.endsWith("o", 8));          // true
console.log(msg.includes("o", 8));          // false
```

## 6.2 repeat方法

> ECMAScript 6 还向字符串添加了 repeat() 方法，它接受一个数字参数作为字符串的重复次数。该方法返回一个重复包含初始字符串的新字符串，重复次数等于参数。例如：

```
console.log("x".repeat(3));          // "xxx"
console.log("hello".repeat(2));      // "hellohello"
console.log("abc".repeat(4));        // "abcabcabcabc"
```

## 6.3 字符串模板字面量

> 模板字面量是 ECMAScript 6 针对 JavaScript 直到 ECMAScript 5 依然缺失的如下功能的回应：
>
> - **多行字符串** 针对多行字符串的形式概念（formal concept）。
> - **基本的字符串格式化** 将字符串中的变量置换为值的能力。
> - **转义 HTML** 能将字符串进行转义并使其安全地插入到 HTML 的能力。
>
> 模板字面量以一种全新的表现形式解决了这些问题而不需要向 JavaScript 已有的字符串添加额外的功能。

### 6.3.1 基本语法

> 使用一对反引号 ``(tab正上方的按键)来表示模板字面量。

```
let message = `Hello world!`;   //使用模板字面量创建了一个字符串

console.log(message);               // "Hello world!"
console.log(typeof message);        // "string"
console.log(message.length);        // 12
```

> 注意：如果模板字符串中使用到了反引号，则应该转义。但是单双引号不需要转义

### 6.3.2 多行字符串

> 在ES5之前JavaScript是不支持多行字符串的。（但是在以前的版本中有一个大家都认为是bug的方式可以写出多行字符串，就是在尾部添加一个反斜杠 \）

```
<body>
<script type="text/javascript">
    var s = "abc \
    aaaaaa";
    console.log(s); //但是输出的结果中不包括换行
</script>
</body>
```

> 但是在ES6中字符串的模板字面量轻松的解决了多行字符串的问题，而且没有任何新的语法

```
<script type="text/javascript">
    var s = `abc
    aaaaa
    dsalfja
    dfadfja`;
    console.log(s);
</script>
```

> 但是要注意： 反引号中的所有空格和缩进都是有效字符。

## 6.3.3 字符串置换

> 置换允许你将 JavaScript 表达式嵌入到模板字面量中并将其结果作为输出字符串中的一部分。
>
> 语法：${变量名、表达式、任意运算、方法调用等}
>
> 可以嵌入任何有效的JavaScript代码

```
<script type="text/javascript">
    var name = "李四";
    var msg = `欢迎你${name}同学`;
    console.log(msg)
</script>
```

## 6.3.4 模板标签

### 6.3.4.1 什么是模板标签

> 模板字面量真正的强大之处来源于模板标签。一个模板标签可以被转换为模板字面量并作为最终值返回。标签在模板的头部，即左 ` 字符之前指定，如下所示：

```
let message = myTag`Hello world`;
```

> 在上面的代码中，myTag就是模板标签。
>
> myTag其实是一个函数，这个函数会被调用来处理这个模板字符串。

### 6.3.4.2 定义模板标签

> 一个标签仅代表一个函数，他接受需要处理的模板字面量。
>
> 标签分别接收模板字面量中的片段，且必须将它们组合以得出结果。
>
> 函数的首个参数为包含普通 JavaScript 字符串的数组。余下的参数为每次置换的对应值。
>
> 标签函数一般使用剩余参数来定义，以便轻松地处理数据。如下：

```
<script type="text/javascript">
    let name = '张三',
        age = 20,
        message = show`我来给大家介绍${name}的年龄是${age}.`;

    /*
        应该定义一个函数show：
        参数1：一个字符串数组。在本例中包含三个元素。
            0:"我来给大家介绍"
            1:"的年龄是"
            2:"."
        参数2和参数3：表示需要置换的字符串的值。
     */
    function show(stringArr, value1, value2) {
        console.log(stringArr); //
        console.log(value1);  // 张三
        console.log(value2);  // 20
        return "abc";
    }
    console.log(message); //abc
</script>
```

> 为了简化书写，一般把Value1和Value2写成剩余字符串的形式

```
function show(stringArr, ...values){

}
```

# 七、解构

## 7.1 解构的实用

> 在 ECMAScript 5 或更早的版本中，从对象或数组中获取特定的数据并赋值给本地变量需要书写很多并且相似的代码。例如：

```
let options = {
        repeat: true,
        save: false
   };

// 从对象中提取数据

let repeat = options.repeat,
    save = options.save;
```

> 这段代码反复地提取在 options 上存储地属性值并将它们传递给同名的本地变量。虽然这些看起来不是那么复杂，不过想象一下如果你的一大批变量有着相同的需求，你就只能一个一个地赋值。而且，如果你需要从对象内部嵌套的结构来查找想要的数据，你极有可能为了一小块数据而访问了整个数据结构。

> 这也是 ECMAScript 6 给对象和数组添加解构的原因。当你想要把数据结构分解为更小的部分时，从这些部分中提取数据会更容易些。很多语言都能使用精简的语法来实现解构操作。ECMAScript 6 解构的实际语法或许你已经非常熟悉：对象和数组字面量。

# 7.2 对象解构

## 7.2.1 对象解构的基本形式

> 对象结构的语法就是在赋值语句的左侧使用类似对象字面量的结构。

```
let node = {
        type: "Identifier",
        name: "foo"
    };
//这里就相当于声明了两个变量：  type = node.type;  name:node.name
let { type, name } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

> 在上面的结构中必须要初始化。否则会出现语法错误。

```
// 语法错误!
var { type, name };

// 语法错误!
let { type, name };

// 语法错误!
const { type, name };
```

## 7.2.2 解构赋值表达式

> 如果声明的变量想改变他们的值，也可以使用解构表达式。

```
<script type="text/javascript">
    let node = {
      type: "Identifier",
      name: "foo"
    },
    type = "Literal",
    name = 5;

  //注意：此处必须要在圆括号内才能使用解构表达式
  ({type, name} = node);

  console.log(type);      // "Identifier"
  console.log(name);      // "foo""
</script>
```

### 7.2.3 对象解构时的默认值

> 如果赋值号右边的对象中没有与左边变量同名的属性，则左边的变量会是 undefined

```
let node = {
        type: "Identifier",
        name: "foo"
    };
//因为node中没有叫value的属性，所以valued的值将会是undefined
let { type, name, value } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // undefined
```

> 不过我们也可以手动指定他的默认值。（这个和函数的参数默认值很像）

```
<script type="text/javascript">
    let node = {
        type: "Identifier",
        name: "foo"
    };
    //手动添加value的默认值为3
    let { type, name, value = 3} = node;

    console.log(type);      // "Identifier"
    console.log(name);      // "foo"
    console.log(value);     // 3
</script>
```

### 7.2.4 赋值给不同的变量名

> 在前面的操作中，都是把对象的属性值，赋值给同名变量。
>
> 其实也可以赋值给不同名的变量。

```
<script type="text/javascript">
    let node = {
        type: "Identifier",
        name: "foo"
    };
    // localType才是要定义的新的变量。  type是node的属性
    let {type: localType, name: localName} = node;

    console.log(localType);     // "Identifier"
    console.log(localName);     // "foo"
</script>
```

> 注意：冒号后面才是要定义的新的变量，这个和我们的对象字面量不太一样！
>
> 这个地方也可以使用默认值。

```
let node = {
        type: "Identifier"
    };

let { type: localType, name: localName = "bar" } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "bar"
```

# 7.3 数组解构

## 7.3.1 数组解构基本语法

> 数据解构的语法和对象解构看起来类似，只是将对象字面量替换成了数组字面量，而且解构操作的是数组内部的位置（索引）而不是对象中的命名属性，例如：

```
let colors = [ "red", "green", "blue" ];
let [ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

> 如果只想取数组中的某一项，则可以不用命名。

```
let colors = [ "red", "green", "blue" ];
//只取数组中的第三项。
let [ , , thirdColor ] = colors;

console.log(thirdColor);        // "blue"
```

### 7.3.2 解构表达式

> 你可以想要赋值的情况下使用数组的解构赋值表达式，但是和对象解构不同，没必要将它们包含在圆括号中，例如：

```
let colors = [ "red", "green", "blue" ],
    firstColor = "black",
    secondColor = "purple";

[ firstColor, secondColor ] = colors;  //可以不用加括号。当然添加也不犯法

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

> 数组解构表达式有一个很常用的地方，就是交换两个变量的值。在以前一般定义一个第三方变量进行交换，例如下面的代码：

```
<script type="text/javascript">
    let a = 3,
        b = 4,
        temp;
    temp = a;
    a = b;
    b = temp;
    console.log(a);
    console.log(b)
</script>
```

> 那么在ES6中完全可以抛弃第三方变量这种方式，使用我们的数组解构表达式

```
<script type="text/javascript">
    let a = 3,
        b = 4;
    //左侧和前面的案例是一样的，右侧是一个新创建的数组字面量。
    [a, b] = [b, a];
    console.log(a);
    console.log(b)
</script>
```

# 八、新的基本类型：Symbol

> 以前我们有5种基本数据类型：Number、String、Boolean、Null、Undefined
>
> ES6新增了一种新的数据类型：Symbol

> 在ES5之前我们都没办法创建私有变量，只能想办法去封装。symbol 来创建私有成员，这也是 JavaScript 开发者长久以来期待的一项特性。

# 8.1 创建Symbol

> Symbol在基本数据类型中是比较特别的。我们以前的都可以用字面量去创建基本数据类型的数据，但是Symbol却不可以使用字面量的是形式去创建。

> 我们可以使用symbol全局函数来创建Symbol。

```
<script type="text/javascript">
    let firstName = Symbol();   //创建一个Symbol
    let person = {};

    person[firstName] = "张三";
    console.log(person[firstName]);     // "张三"
</script>
```

说明：上面的代码中，firstName 作为 symbol 类型被创建并赋值给 person 对象以作其属性。每次访问这个属性时必须使用该 symbol 。

> 在创建Symbol的时候，也可以传入字符串，这个字符串也仅仅是在调试输出的时候方便，实际没有啥用处。

```
<script type="text/javascript">
    var s1 = Symbol("abc");
    var s2 = Symbol("abc");
    console.log(s1 == s2); //false
</script>
```

> 注意：任意两个Symbol都不会相等，即使创建他们的时候使用了相同的参数。

# 8.2 识别Symbol

> 既然 symbol 是基础类型，你可以使用 typeof 操作符来判断变量是否为 symbol 。ECMAScript 6 拓展了 typeof 使其操作 symbol 时返回 "symbol"。例如：

```
let symbol = Symbol();
console.log(typeof symbol);         // "symbol"
```

# 8.3 Symbol作为属性名

由于每一个Symbol值都是不相等的，这意味着Symbol值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

```
var mySymbol = Symbol();

// 第一种写法

var a = {};

a[mySymbol] = 'Hello!';

// 第二种写法

var a = {

    [mySymbol]: 'Hello!'
}
```

> 以上两种写法都是相同的结果

注意：

1. symbol作为对象的属性的时候，只能使用 [ ] 去访问，不能使用点去访问。
2. symbol作为对象的属性名使用的时候，该属性还是公开属性，不是私有属性。但是这个时候使用for... in和for...of时无法遍历到这个symbol属性的。

## 8.4 Symbol属性名的遍历

Symbol 作为属性名，该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。但是，它也不是私有属性，有一个Object.getOwnPropertySymbols方法，可以获取指定对象的所有 Symbol 属性名。

> 看下面的代码

```
<script type="text/javascript">
    var obj = {};
    var a = Symbol('a');
    var b = Symbol('b');

    obj[a] = 'Hello';
    obj[b] = 'World';
    // 返回obj对象所有Symbol类型的属性名组成的数组。
    var objectSymbols = Object.getOwnPropertySymbols(obj);
    console.log(objectSymbols)  //[Symbol(a), Symbol(b)]
</script>
```

> 看下面的代码

```
var obj = {};

var foo = Symbol("foo");
obj[foo] = "lisi";
for (var i in obj) {
  console.log(i); // 无输出 。    因为遍历不到Symbol型的属性
}

Object.getOwnPropertyNames(obj);// []   只能拿到非Symbol类型的属性


Object.getOwnPropertySymbols(obj) //[Symbol(foo)]
```

> 还有一个新API可以拿到所有类型的属性，包括常规和Symbol型的。
>
> Reflect.ownKeys

```
let obj = {
  [Symbol('my_key')]: 1,
  enum: 2,
  nonEnum: 3
};

Reflect.ownKeys(obj);//  ["enum", "nonEnum", Symbol(my_key)]
```

说明：

1. 由于以 Symbol 值作为名称的属性，不会被常规方法遍历得到。我们可以利用这个特性，为对象定义一些非私有的、但又希望只用于内部的方法。

# 8.5 Symbol.for(字符串)和Symbol.keyFor(symbol类型的值)

> 一、Symbol.for(字符串参数)：在全局环境中搜索 以该字符串作为参数的Symbol值，如果搜到则返回这个sybol，如果搜不到则创建一个Symbol，并把它注册在全局环境中。

```
    <script type="text/javascript">
        //第一次搜不到，则新创建一个返回，并在全局环境(window)中注册
        var a = Symbol.for("foo");
         //第二次搜到上次创建的
        var b = Symbol.for("foo");
        console.log(a === b);  //因为两次搜到的是同一个Symbol，所以此处是true
    </script>
```

> Symbol.for()和Symbol()都可以创建Symbol类型的数据。
>
> 二者区别：

> 1. Symbol.for()对同样的字符串，每次得到结果肯定是一样的。因为都是从全局环境中搜索。
> 2. Symbol()则不会有搜索的过程，每次都是一个全新的不同的symbol，而且也不会向全局环境中注册。

> 看下面的代码

```
<script type="text/javascript">
   var a = Symbol("foo");
   var b = Symbol.for("foo");
   console.log(a == b); //false
</script>
```

> 二、Symbol.keyFor(symbol):返回一个已经注册的symbol的"key"。

```
<script type="text/javascript">
    var a = Symbol("foo");
    var b = Symbol.for("foo");
    console.log(Symbol.keyFor(a)); // undefined.   因为a没有想全局环境中登记，所以是undefinded
    console.log(Symbol.keyFor(b)); // foo
</script>
```

# 九、Set数据结构

JavaScript 在绝大部分历史时期内只有一种集合类型，那就是数组。数组在 JavaScript 中的使用方式和其它语言很相似，但是其它集合类型的缺乏导致数组也经常被当作队列（queues）和栈（stacks）来使用。

因为数组的索引只能是数字类型，当开发者觉得非数字类型的索引是必要的时候会使用非数组对象。这项用法促进了以非类数组对象为基础的 set 和 map 集合类型的实现。

> Set是类似数组的一种结构，可以存储数据，与数组的区别主要是 **Set中的元素不能重复，而数组中的元素可以重复。**
>
> 一句话总结：***Set类型是一个包含无重复元素的有序列表***

## 9.1 创建Set和并添加元素

> Set本身是一个构造函数。

```
<script type="text/javascript">
    //创建Set数据结构对象。
    var s = new Set();
    //调用set对象的add方法，向set中添加元素
    s.add("a");
    s.add("c");
    s.add("b");
     //set的size属性可以获取set中元素的个数
    console.log(s.size)
</script>
```

## 9.2 Set中不能添加重复元素

```
<script type="text/javascript">
    var s = new Set();
    s.add("a");
    s.add("c");
    s.add("b");
    s.add("a");  //重复，所以添加失败。注意这个地方并不会保存。
    console.log(s.size); // 长度是3
</script>
```

> 看下面的代码：

```
<script type="text/javascript">
    var s = new Set();
    s.add(5);
    s.add("5");
    console.log(s.size); // 长度是2
</script>
```

在上面的代码中，数字5和字符串5都会添加成功。为什么呢?

> Set是使用什么机制来判断两个元素是否相等的呢?
>
> **是通过我们前面说过的 Object.is(a, b) 来判断两个元素是否相等。**
>
> 回忆一下：这个方法除了 +0和-0、NaN和NaN认为相等，其余和三个 === 是完全一样的。

```
<script type="text/javascript">
    var s = new Set();
    s.add(+0);
    s.add(-0);  //重复添加不进去
    s.add(NaN);
    s.add(NaN); //重复添加不进去
    s.add([]);
    s.add([]);  //两个空数组不相等，所以可以添加进去
    s.add({});
    s.add({});  // 两个空对象也不重复，所以也可以添加进去
    console.log(s.size); // 长度是6
</script>
```

## 9.3 使用数组初始化Set

```
<script type="text/javascript">
    //使用数组中的元素来初始化Set，当然碰到重复的也不会添加进去。
    var s = new Set([2, 3, 2, 2, 4]);
    console.log(s.size)
</script>
```

## 9.4 判断一个值是否在Set中

> 使用Set的 has() 方法可以判断一个值是否在这个set中。

```
<script type="text/javascript">
    let set = new Set();
    set.add(5);
    set.add("5");

    console.log(set.has(5));    // true
    console.log(set.has(6));    // false
</script>
```

## 9.5 移除Set中的元素

> delete(要删除的值) ：删除单个值
>
> clear()：清空所有的值

```
<script type="text/javascript">
    let set = new Set();
    set.add(5);
    set.add("5");

    console.log(set.has(5));    // true

    set.delete(5);

    console.log(set.has(5));    // false
    console.log(set.size);      // 1

    set.clear();

    console.log(set.has("5"));  // false
    console.log(set.size);      // 0
</script>
```

## 9.6 遍历Set

> 数组有个方法forEach可以遍历数组。
>
> 1. Set也有forEach可以遍历Set。
>
> 使用Set的forEach遍历时的回调函数有三个参数：
>
> function (value, key, ownerSet){
>
> }
>
> 参数1：遍历到的元素的值
>
> 参数2：对set集合来说，参数2的值和参数1的值是完全一样的。
>
> 参数3：这个 set 自己

```
<script type="text/javascript">
    let set = new Set(["a", "c", "b", 9]);
    set.forEach(function (v, k, s) {
        console.log(v + "   " + (v === k) + "  " + (s === set));   // 永远是true
    })

</script>
```

> 2. for...of也可以遍历set。

```
for(var v of set){
    console.log(v)
}
```

## 9.7 将Set转换为数组

> 将数组转换为Set相当容易，你只需要在创建Set集合时把数组作为参数传递进去即可。

> **把Set转换为数组使用前面讲到的扩展运算符也很容易**

```
<script type="text/javascript">
    let set = new Set([1, 2, 3, 3, 3, 4, 5]),
        arr = [...set];  //使用扩展运算符。那么新的数组中已经没有了重复元素。注意，此处对set并没有什么影响

    console.log(arr);             // [1,2,3,4,5]
</script>
```

> 这种情况在需要去数组中重复元素的时候非常好用。

```
<script type="text/javascript">
    function eliminateDuplicates(items) {
        return [...new Set(items)];
    }
    let numbers = [1, 2, 3, 3, 3, 4, 5, 5, 2, 1, 1],
        //返回的是新的没有重复元素的数组。
        noDuplicates = eliminateDuplicates(numbers);
    console.log(noDuplicates);      // [1,2,3,4,5]
</script>
```

---

***Set提供了处理一系列值的方式，不过如果想给这些值添加一些附加数据则显得力不从心，所以又提供了一种新的数据结构：Map***

# 十、Map数据结构

ECMAScript 6 中的 map 类型包含一组有序的键值对，其中键和值可以是任何类型。

键的比较结果由 ~~Object.is~~() 来决定，所以你可以同时使用 5 和 "5" 做为键来存储，因为它们是不同的类型。

这和使用对象属性做为值的方法大相径庭，因为 **对象的属性会被强制转换为字符串类型**。

## 10.1 创建Map对象和Map的基本的存取操作

> 1. Map创建也是使用Map构造函数
> 2. 向Map存储键值对使用set(key, value);方法
> 3. 可以使用get(key),来获取指定key对应的value

```
<script type="text/javascript">
    var map = new Map();
    map.set("a", "lisi");
    map.set("b", "zhangsan");
    map.set("b", "zhangsan222");  // 第二次添加，新的value会替换掉旧的
    console.log(map.get("a"));
    console.log(map.get("b"));   //zhangsan222
     console.log(map.get("c")); //undefined.如果key不存在，则返回undefined
     console.log(map.size); //2
</script>
```

## 10.2 Map与Set类似的3个方法

- has(key) - 判断给定的 key 是否在 map 中存在
- delete(key) - 移除 map 中的 key 及对应的值
- clear() - 移除 map 中所有的键值对

## 10.3 初始化Map

> 创建Map的时候也可以像Set一样传入数组。但是传入的数组中必须有两个元素，这个两个元素分别是一个数组。
>
> 也就是传入的实际是一个二维数组！

```
<script type="text/javascript">
  //map接受一个二维数组
    var map = new Map([
      //每一个数组中，第一个是是map的可以，第二个是map的value。如果只有第一个，则值是undefined
        ["name", "lisi"],
        ["age", 20],
        ["sex", "nan"]
    ]);
    console.log(map.size);
    console.log(map.get("name"))
</script>
```

## 10.4 Map的forEach方法

```
<script type="text/javascript">
    var map = new Map([
        ["name", "李四"],
        ["age", 20],
        ["sex", "nan"]
    ]);
    /*
        回调函数有函数:
        参数1: 键值对的value
        参数2: 键值对的key
        参数3: map对象本身
     */
    map.forEach(function (value, key, ownMap) {
        console.log(`key=${key} ,vlue=${value}`);
        console.log(this);
    })
</script>
```

# 十一、迭代器和for...of循环

## 11.1 循环问题

```
var colors = ["red", "green", "blue"];

for (var i = 0, len = colors.length; i < len; i++) {
    console.log(colors[i]);
}
```

> 上面的代码写起来简单，但是实际使用的过程中，我们需要自己去控制变量，如果有嵌套的情况下，还要控制多个变量，很容易出错。
>
> 迭代器就是为了解决这个问题的。

## 11.2 什么是迭代器

1. 迭代器是一个对象
2. 迭代器提供一个方法next() 这个方式总是能够返回迭代到的对象。
3. next返回的对象中，应该有两个属性：done 是一个boolean值。 value：具体的数据

迭代器只是带有特殊接口(方法)的对象。所有迭代器对象都带有 next() 方法并返回一个包含两个属性的结果对象。这些属性分别是 value 和 done，前者代表下一个位置的值，后者在没有更多值可供迭代的时候为 true 。迭代器带有一个内部指针，来指向集合中某个值的位置。当 next() 方法调用后，指针下一位置的值会被返回。

若你在末尾的值被返回之后继续调用 next()，那么返回的 done 属性值为 true，value 的值则由迭代器设定。该值并不属于数据集，而是专门为数据关联的附加信息，如若该信息并未指定则返回 undefined 。迭代器返回的值和函数返回值有些类似，因为两者都是返回给调用者信息的最终手段。

> 我们可以用ES5之前的知识手动创建一个迭代器：

```
function createIterator(items) {
    var i = 0;
    return {
        next: function() {
            var done = (i >= items.length);
            var value = !done ? items[i++] : undefined;
            return {
                done: done,
                value: value
            };

        }
    };
}
//创建一个可以在指定数组上面迭代的迭代器对象。
var iterator = createIterator([1, 2, 3]);

console.log(iterator.next());          // "{ value: 1, done: false }"
console.log(iterator.next());          // "{ value: 2, done: false }"
console.log(iterator.next());          // "{ value: 3, done: false }"
console.log(iterator.next());          // "{ value: undefined, done: true }"

// for all further calls
console.log(iterator.next());          // "{ value: undefined, done: true }"
```

从以上的示例来看，根据 ECMAScript 6 规范模拟实现的迭代器还是有些复杂。

幸运的是，ECMAScript 6 还提供了生成器，使得迭代器对象的创建容易了许多。

## 11.3 生成器函数

> **生成器函数就是返回迭代器的函数！**
>
> 生成器函数由 function 关键字和之后的星号（*）标识，同时还能使用新的 yield 关键字。

> 看下面代码：

```
<script type="text/javascript">
    //生成器函数。  注意中间的 * 不能丢
    function * createIterator() {
        //每个yield的后面的值表示我们迭代到的值。   yield也定义了我们迭代的顺序。
        yield 3;
        yield 4;
        yield 2;
    }
    var it = createIterator();
    console.log(it.next().value);   // 2
    console.log(it.next().value);   // 4
    console.log(it.next().value);   // 2
    console.log(it.next().value);  //undefined
</script>
```

> **迭代器函数也是函数，所以他可以像正常的函数一样调用，但是生成器函数会自动返回一个迭代器对象。**
>
> 每调用一次迭代器的next方法，如果碰到yield都会返回一个迭代到的一个对象，然后停止执行，直到下次调用next方法，会从上次停止的地方继续执行。

```
//这个迭代器函数返回的迭代器可以迭代传入的数组中的所有元素。
function *createIterator(items) {
    for (let i = 0; i < items.length; i++) {
      //每调用一次next，碰到yild程序就会停止，并返回迭代到的对象 {value : items[i], done : true}
        yield items[i];
    }
}

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// 进一步调用
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

> 注意：

1. yield 关键字只能 **直接用在生成器内部** 。在其它地方甚至是生成器内部的函数中使用都会抛出语法错误。

## 11.4 生成器函数表达式

> 你可以使用函数表达式来创建生成器，只需在 function 关键字和圆括号之间添加星号（*）。例如：

```
let createIterator = function *(items) {
    for (let i = 0; i < items.length; i++) {
        yield items[i];
    }
};

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next());           // "{ value: 2, done: false }"
console.log(iterator.next());           // "{ value: 3, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"

// 进一步调用
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

> 注意：无法使用箭头函数来创建生成器。

## 11.5 可迭代类型和for-of迭代循环

> 迭代器的主要工作就是迭代数据，但是不是所有的数据都是可以迭代的。

与迭代器紧密相关的是，可迭代类型是指那些包含 Symbol.iterator 属性（方法）的对象。

该 symbol 类型定义了返回迭代器的函数。在 ECMAScript 6 中，所有的集合对象（数组，set 和 map）与字符串都是可迭代类型，因此它们都有默认的迭代器。可迭代类型是为了 ECMAScript6 新添加的 **for-of** 循环而设计的。

换句话说，默认情况下只有 **数组、set、Map和字符串**才可以使用迭代器去迭代。 (也就可以使用for...of了)

for...of循环只迭代出来的元素，根本不管索引！不管索引！不管索引！重要的问题重复三遍！

> 使用 for...of 迭代数组：

```
<script type="text/javascript">
    var arr = ["a", "c", "b", "d"];
    for(var item of arr){
        console.log(item)
    }

</script>
```

> 使用 for...of 迭代Set:

```
<script type="text/javascript">
    var set = new Set(["a", "c", "b", "d"]);
    for(var item of set){
        console.log(item)
    }

</script>
```

> 使用 for...of 迭代Map：

```
<script type="text/javascript">
    var map = new Map([["name", "lisi"],["sex", "男"],["age", 20]]);
    map.set("aaa", "bbb")
    for(var item of map){
        console.log(item);  //注意：这里迭代到的是由key和value组成的数组。
    }
</script>
```

> 使用for ... of迭代字符串

```
<script type="text/javascript">
    var s = "abcd";
    for(let c of s){
        console.log(c)
    }
</script>
```

> 注意：for...of 只能迭代可以迭代的对象，对于非可迭代对象使用for...of会抛出异常

说明：以数组为例。

for-of 循环首先会调用 values 数组的 Symbol.iterator 方法来获取迭代器（Symbol.iterator 方法由幕后的 JavaScript 引擎调用）。之后再调用 iterator.next() 并将结果对象中的 value 属性值，即 1，2，3，依次赋给 num 变量。当检测到结果对象中的 done 为 true，循环会退出，所以 num 不会被赋值为 undefined 。

如果你只想简单的迭代数组或集合中的元素，那么 for-of 循环比 for 要更好。for-of 一般不容易出错，因为要追踪的条件更少。所以还是把 for 循环留给复杂控制条件的需求吧。

## 11.6 访问可迭代类型的默认迭代器

> Symbol.iterator是可迭代类型的一个方法，调用这个方法就可以获取到他的默认迭代器。

```
<script type="text/javascript">
    let s = "abcd";
    let it = s[Symbol.iterator]();  //调用字符串的Symbol.iterator方法
    console.log(it.next());  //返回迭代器迭代到的第一个对象
</script>
```

> 因为Symbol可以返回一个对象的默认迭代器，所以我们可以使用它来判断一个对象是否可迭代

```
<script type="text/javascript">
    function isIterable(object) {
        return typeof object[Symbol.iterator] === "function";
    }

    console.log(isIterable([1, 2, 3]));     // true
    console.log(isIterable("Hello"));       // true
    console.log(isIterable(new Map()));     // true
    console.log(isIterable(new Set()));     // true
    console.log(isIterable({"name":"李四"})); // false。普通对象不可迭代
</script>
```

## 11.7 自定义可迭代类型

> 开发者自定义的对象默认是不可迭代类型，但是你可以为它们创建 Symbol.iterator 属性并指定一个生成器来使这个对象可迭代。例如：

```
let collection = {
    items: [],
    *[Symbol.iterator]() {
        for (let item of this.items) {
            yield item;
        }
    }

};

collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}
```

# 十二、类

> 和大多数面向对象的语言（object-oriented programming language）不同，JavaScript 在诞生之初并不支持使用类和传统的类继承并作为主要的定义方式来创建相似或关联的对象。
>
> 这很令开发者困惑，而且在早于 ECMAScript 1 到 ECMAScript 5 这段时期，很多库都创建了一些实用工具（utility）来让 JavaScript 从表层上支持类。
>
> 尽管一些 JavaScript 开发者强烈主张该语言不需要类，但由于大量的库都对类做了实现，ECMAScript 6 也顺势将其引入。

# 12.1 ES5之前的模拟的类

在 ECMAScript 5 或更早的版本中，JavaScript 没有类。和类这个概念及行为最接近的是创建一个构造函数并在构造函数的原型上添加方法，这种实现也被称为自定义的类型创建，例如：

```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
};

let person = new PersonType("Nicholas");
person.sayName();   // 输出 "Nicholas"

console.log(person instanceof PersonType);  // true
console.log(person instanceof Object);      // true
```

> 说明：

前面的PersonType我们以前一直叫做构造函数，其实他就是一个类型，因为他确实表示了一种类型。

# 12.2 ES6中基本的类声明

> 在ES6直接借鉴其他语言，引入了类的概念。所以再实现上面那种模拟 的类就容易了很多。

```
//class关键字必须是小写。    后面就是跟的类名
class PersonClass {
    // 等效于 PersonType 构造函数。
    constructor(name) {  //这个表示类的构造函数。constuctor也是关键字必须小写。
        this.name = name;  //创建属性。   也叫当前类型的自有属性。
    }
    // 等效于 PersonType.prototype.sayName.   这里的sayName使用了我们前面的简写的方式。
    sayName() {
        console.log(this.name);
    }
}
let person = new PersonClass("Nicholas");
person.sayName();   // 输出 "Nicholas"

console.log(person instanceof PersonClass);     // true
console.log(person instanceof Object);          // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName);  // "function"
```

> 说明：

1. 自有属性：属性只出现在实例而不是原型上，而且只能由构造函数和方法来创建。在本例中，name 就是自有属性。我建议 **尽可能的将所有自有属性创建在构造函数中**，这样当查找属性时可以做到一目了然。
2. 类声明只是上例中自定义类型的语法糖。PersonClass 声明实际上创建了一个行为和 constructor 方法相同的构造函数，这也是 typeof PersonClass 返回 "function" 的原因。sayName() 在本例中作为 PersonClass.prototype 的方法，和上个示例中 sayName() 和 PersonType.prototype 关系一致。这些相似度允许你混合使用自定义类型和类而不需要纠结使用方式。

***虽然类和以前的使用构造函数+原型的方式很像，但是还是有一些不太相同的地方，而且要牢记***

1. 类声明和函数定义不同，**类的声明是不会被提升的**。类声明的行为和 let 比较相似，所以当执行流作用到类声明之前类会存在于暂存性死区（temporal dead zone）内。
2. 类声明中的代码自动运行在严格模式下，同时没有任何办法可以手动切换到非严格模式。
3. 所有的方法都是不可枚举的（non-enumerable），这和自定义类型相比是个显著的差异，因为后者需要使用 Object.defineProperty() 才能定义不可枚举的方法。
4. 所有的方法都不能使用 new 来调用，因为它们没有内部方法 [[Construct]]。
5. 不使用 new 来调用类构造函数会抛出错误。也就是 **必须使用new 类()** 的方式使用
6. 试图在类的方法内部重写类名的行为会抛出错误。（因为在类的内部，类名是作为一个常量存在的）

## 12.2 匿名类表达式

> 函数有函数表达式，类也有类表达式。
>
> 类表达式的功能和前面的类的声明是一样的。

```
let PersonClass = class {

    // 等效于 PersonType 构造函数
    constructor(name) {
        this.name = name;
    }

    // 等效于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};

let person = new PersonClass("Nicholas");
person.sayName();   // 输出 "Nicholas"

console.log(person instanceof PersonClass);     // true
console.log(person instanceof Object);          // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName);  // "function"
```

## 12.3 具名类表达式

```

let PersonClass = class PersonClass2{

    // 等效于 PersonType 构造函数
    constructor(name) {
        this.name = name;
    }

    // 等效于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};
```

> 注意：具名类表达式中PersonClass2这个类名只能在类的内部访问到，在外面是访问不到的.

## 12.4 作为一等公民的类型

> 在JavaScript中，函数是作为一等公民存在的。(也叫一等函数)。
>
> 类也是一等公民。

1. 类可以作为参数传递

```
function createObject(classDef) {
    return new classDef();
}

let obj = createObject(class {

    sayHi() {
        console.log("Hi!");
    }
});

obj.sayHi();        // "Hi!"
```

2. 立即调用类构造函数，创建单例

```
let person = new class {

    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }

}("Nicholas");

person.sayName();       // "Nicholas"
```

## 12.5 动态计算类成员的命名

> 类的成员，也可以像我们前面的对象的属性一样可以动态计算.( 使用[ ] 来计算)

```
let methodName = "sayName";
class PersonClass {
    constructor(name) {
        this.name = name;
    }

    [methodName]() {
        console.log(this.name);
    }
}
let me = new PersonClass("Nicholas");
me.sayName();           // "Nicholas"
```

## 12.6 静态成员

> 在ES5中，我们可以直接给构造函数添加属性或方法来模拟静态成员。

```
function PersonType(name) {
    this.name = name;
}
// 静态方法。  直接添加到构造方法上。  (其实是把构造函数当做一个普通的对象来用。)
PersonType.create = function(name) {
    return new PersonType(name);
};
// 实例方法
PersonType.prototype.sayName = function() {
    console.log(this.name);
};
var person = PersonType.create("Nicholas");
```

> 在上面的create方法在其他语言中一般都是作为静态方法来使用的。

**下面高能，请注意：**

ECMAScript 6 的类通过在方法之前使用正式的 ***static*** 关键字简化了静态方法的创建。例如，下例中的类和上例相比是等效的：

```
class PersonClass {

    // 等效于 PersonType 构造函数
    constructor(name) {
        this.name = name;
    }

    // 等效于 PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }

    // 等效于 PersonType.create。
    static create(name) {
        return new PersonClass(name);
    }
}

let person = PersonClass.create("Nicholas");
```

> 注意：静态成员通过实例对象不能访问，只能通过类名访问！！！
>
> 通过和ES5模拟静态方法的例子你应该知道为啥了吧

## 12.7 ES6中的继承

> 在ES6之前要完成继承，需要写很多的代码。看下面的继承的例子：

```
<script type="text/javascript">
    function Father(name) {
        this.name = name;
    }
    Father.prototype.sayName = function () {
        console.log(this.name);
    }

    function Son(name,age) {
        Father.call(this, name);
        this.age = age;
    }
    Son.prototype = new Father();
    Son.prototype.constructor = Son;
    Son.prototype.sayAge = function () {
        console.log(this.age);
    }

    var son1 = new Son("儿子", 20);
    son1.sayAge();  //20
    son1.sayName(); //儿子

</script>
```

### 12.7.1 继承的基本写法

> 如果在ES6通过类的方式完成继承就简单了很多。
>
> 需要用到一个新的关键字：extends

```
<script type="text/javascript">
    class Father{
        constructor(name){
            this.name = name;
        }
        sayName(){
            console.log(this.name);
        }
    }
    class Son extends Father{  //extents后面跟表示要继承的类型
        constructor(name, age){
            super(name);  //相当于以前的：Father.call(this, name);
            this.age = age;
        }
        //子类独有的方法
        sayAge(){
            console.log(this.age);
        }
    }

    var son1 = new Son("李四", 30);
    son1.sayAge();
    son1.sayName();
     console.log(son1 instanceof Son);  // true
     console.log(son1 instanceof Father);  //true

</script>
```

> 这种继承方法，和我们前面提到的构造函数+原型的继承方式本质是一样的。但是写起来更简单，可读性也更好。

***关于super的使用，有几点需要注意：***

1. 你只能在派生类中使用 super()，否则（没有使用 extends 的类或函数中使用）一个错误会被抛出。
2. 你必须在构造函数的起始位置调用 super()，因为它会初始化 this。任何在 super() 之前访问 this 的行为都会造成错误。也即是说super()必须放在构造函数的首行。
3. 在类构造函数中，唯一能避免调用 super() 的办法是返回一个对象。

## 12.7.2　在子类中屏蔽父类的方法

> 如果在子类中声明与父类中的同名的方法，则会覆盖父类的方法。(这种情况在其他语言中称之为 方法的覆写、重写 )

```
<script type="text/javascript">
    class Father{
        constructor(name){
            this.name = name;
        }
        sayName(){
            console.log(this.name);
        }
    }
    class Son extends Father{  //extents后面跟表示要继承的类型
        constructor(name, age){
            super(name);  //相当于以前的：Father.call(this, name);
            this.age = age;
        }
        //子类独有的方法
        sayAge(){
            console.log(this.age);
        }
        //子类中的方法会屏蔽到父类中的同名方法。
        sayName(){
            super.syaName();  //调用被覆盖的父类中的方法。
            console.log("我是子类的方法，我屏蔽了父类：" + name);
        }
    }

    var son1 = new Son("李四", 30);
    son1.sayAge();
    son1.sayName();
</script>
```

> 如果在子类中又确实需要调用父类中被覆盖的方法，可以通过super.方法()来完成。
>
> 注意：
>
> 1. 如果是调用构造方法，则super不要加点，而且必须是在子类构造方法的第一行调用父类的构造方法
> 2. 普通方法调用需要使用super.父类的方法() 来调用。

## 12.7.3 静态方法也可以继承

```
<script type="text/javascript">
   class Father{
       static foo(){
           console.log("我是父类的静态方法");
       }
   }
   class Son extends Father{

   }
   Son.foo(); //子类也继承了父类的静态方法。  这种方式调用和直接通过父类名调用时一样的。

</script>
```

# 十三、Babel

> 到目前2017年为止，也不是所有的浏览器都支持ES6的特性。
>
> 所以我们需要把ES6转换成ES5的代码，就要用到所谓的转码器。Babel就是目前使用最广泛的把ES6代码转换成ES5及以前代码的转码器。
>
> 有了babel我们就可以放心的使用ES6的最新的语法，而不用担心浏览器不支持了！！！

## 13.1 安装Babel

> 首先要保证电脑上已经安装了npm
>
> 使用如下命令就可以安装babel安装(我们这里使用的是全局安装):

```
npm install -g babel-cli
```

## 13.2 在当前的项目根目录下创建一个package.json文件

> 在当前当前项目的根目录下创建一个package.json文件
>
> 文件内容如下:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "scripts": {
    "build": "babel src -d lib"
  },
  "devDependencies": {
    "babel-cli": "^6.0.0"
  }
}
```

> 注意：
>
> 1. src表示的是放js源码的文件夹
> 2. lib表示的是将来放生成的es5的代码文件夹。

## 13.3 在当前项目跟目录下创建一个 .babelrc文件

> 注意：创建这个文件的时候不需要文件名，只需要扩展名即可。
>
> 文件内容如下：

```
{
  "presets": ["env"]
}
```

## 13.4 安装babel-preset-env

> 用babel-preset-env可以把es6的代码转换成es5代码
>
> 命令如下：

```
npm install babel-preset-env --save-dev
```

## 13.5 在当前项目根目录下创建一个目录 src

> 这个目录的名字随意，但是一定要和前面的package.json中保持一致。

> 在该目录下写es6 js文件代码。

## 13.6 把es6代码转换成es5代码

> 使用下面的命令即可完成：

```
npm run build
```

> 生成文件会自动放入lib目录下。

# 十四、Moudle

> JavaScript 采用 “共享一切” 的代码加载方式是该语言中最令人迷惑且容易出错的方面之一。
>
> 其它语言使用包（package）的概念来定义代码的作用范围，然而在 ECMAScript 6 之前，每个 JavaScript 文件中定义的内容都由全局作用域共享。
>
> 当 web 应用变得复杂并需要书写更多的 JavaScript 代码时，上述加载方式会出现命名冲突或安全方面的问题。
>
> ECMAScript 6 的目标之一就是解决作用域的问题并将 JavaScript 应用中的代码整理得更有条理，于是模块应运而生。

***很不幸的是：目前，所有的浏览器都还不能支持ES6的模块。只能通过第三方的工具转成ES5的代码***

## 14.1 什么是模块

模块是指采取不同于现有加载方式的 JavaScript 文件（与 script 这种传统的加载模式相对）。这种方式很有必要，因为它和 script 使用不同的语义：

1. 模块中的代码自动运行在严格模式下，并无任何办法修改为非严格模式。
2. 模块中的顶级（top level）变量不会被添加到全局作用域中。它们只存在于各自的模块中的顶级作用域。
3. 模块顶级作用域中的 this 为 undefined 。
4. 模块不允许存在 HTML 式的注释（JavaScript 历史悠久的遗留特性）。
5. 模块必须输出可被模块外部代码使用的相关内容。
6. 一个模块可以引入另外的模块。

## 14.2 导出模块

可以使用 export 关键字来对外暴露模块中的部分代码。

一般情况下，可以在任何变量，函数或类声明之前添加这个关键字来输出它们，

> 看下面的代码:
>
> 声明一个文件：a.js 代码如下

```
// 输出变量
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

// 输出函数
export function sum(num1, num2) {
    return num1 + num1;
}

// 输出类
export class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
}

// 该函数没有使用export关键字    所以该函数是模块私有的。也就是说只能在当前文件访问，出了这个文件就访问不到
function subtract(num1, num2) {
    return num1 - num2;
}

// 定义一个函数...
function multiply(num1, num2) {
    return num1 * num2;
}

// 可以把这个函数的引用导出。   和导出函数是一样的。
export { multiply };
```

> 注意：在上面的代码中，除了exprot关键字，其他和我们以前的代码没有任何不同。

## 14.3 引入模块

> 一旦有了导出内容的模块，则可以在另一个模块中使用import关键字来获取他们。
>
> 引入模块的语法：

```
import { identifier1, identifier2 } from "./a.js";
```

import 之后的花括号表示从模块中引入的绑定。from 关键字表示从哪个模块引入这些绑定。模块由一个包含模块路径的字符串表示（称为模块指示符，module sepcifier）。浏览器中的