ECMAScript 2015
从入门到出家

新一代的 javascript 即 ECMAScript 2015(也称为 ECMAScript6 或 ES6)给我们带来很多令人意想不到的功能

技术基础部

Web 前端工程师

谢忠阳

shinexie

概要

ECMAScript 2015（曾用名：ECMAScript 6、ES6）是 JavaScript 语言的最新标准，已经在 2015 年 6 月正式发布。

ES2015 的目标，是使得 JavaScript 语言可以用来编写大型的复杂的应用程序，成为企业级开发语言。

1996 年 11 月 ECMAScript 1.0 版发布。

1998 年 06 月 ECMAScript 2.0 版发布。

1999 年 12 月 ECMAScript 3.0 版发布，成为 JavaScript 的通行标准，得到了广泛支持。

2007 年 10 月 ~~ECMAScript 4.0~~ 版草案发布，对 3.0 版做了大幅升级，预计次年 8 月发布正式版本。草案发布后，由于 4.0 版的目标过于激进，各方对于是否通过这个标准，发生了严重分歧。

2008 年 07 月 ECMAScript 3.1 版发布。

2009 年 12 月 ECMAScript 5.0 版正式发布。Harmony 项目则一分为二，一些较为可行的设想定名为 JavaScript.next 继续开发，后来演变成 ECMAScript 6；一些不是很成熟的设想，则被视为 JavaScript.next.next，在更远的将来再考虑推出。

2011 年 06 月 ECMAscript 5.1 版发布。

2013 年 03 月 ECMAScript 6 草案冻结，不再添加新功能。新的功能设想将被放到 ECMAScript 7。

2013 年 12 月 ECMAScript 6 草案发布。然后是 12 个月的讨论期，听取各方反馈。

2015 年 06 月 ECMAScript 6 发布正式版本，并更名为 **ESMAS2015**。

详细英文介绍：http://www.ecma-international.org/ecma-262/6.0/

ECMAScript7 可能包括的功能有：

Object.observe 用来监听对象（以及数组）的变化。一旦监听对象发生变化，就会触发回调函数。

Async 函数 在 Promise 和 Generator 函数基础上，提出的异步操作解决方案。

Multi-Threading 多线程支持。

Traits 它将是“类”功能（class）的一个替代。

其他可能包括的功能还有：更精确的数值计算、改善的内存回收、增强的跨站点安全、类型化的更贴近硬件的低级别操作、国际化支持（Internationalization Support）、更多的数据结构等等。

第 一 讲　let 和 const 命令

ES6 提出了两个新的声明变量的命令：let 和 const。其中，let 完全可以取代 var，因为两者语义相同，而且 let 没有副作用。

Let 命令

基本用法

ES6 新增了 let 命令，用来声明变量。它的用法类似于 var，但是所声明的变量，只在 let 命令所在的代码块内有效。

```
{
    let a = 10;
    var b = 1;
}
alert(b);// 1
alert(a); // ReferenceError: a is not defined.
```

let 只在变量所在的代码块有效，特别适合 for 循环，也可减少一些闭包的使用。

不存在变量提升

let 不像 var 那样，会发生“变量提升”现象。

```
function fun() {
    console.log(foo); // ReferenceError
    let foo = 2;
}
```

上面代码在声明 foo 之前，就使用这个变量，结果会抛出一个错误。

```
var tmp = 123;
if (true) {
    tmp = 'abc'; // ReferenceError
    let tmp;
}
```

上面代码中，存在全局变量 tmp，但是块级作用域内 let 又声明了一个局部变量 tmp，导致后者绑定这个块级作用域，所以在 let 声明变量前，对 tmp 赋值会报错。

ES6 明确规定，如果区块中存在 let 和 const 命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些命令，就会报错。

总之，在代码块内，使用 let 命令声明变量之前，该变量都是不可用的。这在语法上，称为**“暂时性死区”**（temporal dead zone，简称 TDZ）。

不允许重复声明

let 不允许在相同作用域内，重复声明同一个变量。

```
{
    let a = 10;// 报错
    var a = 1; // 报错
}
{
    let a = 10;
    let a = 1;// 报错
}
```

块级作用域

let 实际上为 JavaScript 新增了块级作用域。

```
function f1() {
    let n = 5;
    if (true) {
        let n = 10;//只有在此块内才可访问此变量
    }
    console.log(n); // 5
}
```

上面的函数有两个代码块，都声明了变量 n，运行后输出 5。这表示外层代码块不受内层代码块的影响。如果使用 var 定义变量 n，最后输出的值就是 10。

块级作用域的出现，实际上使得获得广泛应用的立即执行匿名函数（IIFE）不再必要了。

```
(function () {
    var tmp = 2;//匿名函数写法
}());
{
    let tmp = 2;// 块级作用域写法
}
```

另外，ES6 也规定，函数本身的作用域，在其所在的块级作用域之内。

```
function fun() { alert('弹我说明是 ES6!'); }
(function () {
    if(false) {
        function fun() { alert('弹我说明是 ES5!'); }
    }
    fun();
}());
```

Const 命令

const 也用来声明变量，但是声明的是常量。一旦声明，常量的值就不能改变。const 的作用域与 let

命令相同：只在声明所在的块级作用域内有效，同样变量不提升。const 命令只指向变量所在的地址，引用地址不能重新赋值。

```
const foo = {};
foo = {}; // SyntaxError: invalid assignment to const foo

var a = 1;
const b = a;
a = 2;//TypeError: redeclaration of const b

const C1 = {};
C1.a = 1;
document.write(C1.a); // 1
C1 = {};  // 报错  重新赋值，地址改变

//冻结对象，此时前面用不用 const 都是一个效果
const C2 = Object.freeze({});
C2.a = 1; //Error,对象不可扩展
document.write(C2.a);
```

全局对象的属性

全局对象是最顶层的对象，在浏览器环境指的是 window 对象，在 Node.js 指的是 global 对象。在 JavaScript 语言中，所有全局变量都是全局对象的属性。

ES6 规定，var 命令和 function 命令声明的全局变量，属于全局对象的属性；let 命令、const 命令、class 命令声明的全局变量，不属于全局对象的属性。

```
let a = 1;
// 如果在 node 环境，可以写成 global.a
console.log(window.a);// undefined
```

第二讲 destructuring 变量的解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

数组的解构赋值

```
var [a, b, c] = [1, 2, 3];//等价 var a = 1,b = 2,c = 3;
let [foo, [[bar], baz]] = [1, [[2], 3]];
console.log(foo,bar,baz);//1  2  3

let [,,third] = ["foo", "bar", "baz"];
console.log(third);// "baz"
```

```
let [head, ...tail] = [1, 2, 3, 4];//rest 变量解构
console.log(tail); // [2, 3, 4]

let [x, y] = [1, 2, 3];//不完全解构
console.log(x,y);// 1  2
```

如果解构不成功，变量的值就等于 undefined。

```
var [foo] = [];
var [foo] = 1;
var [foo] = false;
var [foo] = NaN;
var [bar, foo] = [1];
```

以上几种情况都属于解构不成功，foo 的值都会等于 undefined。这是因为原始类型的值，会自动转为对象，比如数值 1 转为 new Number(1)，从而导致 foo 取到 undefined。

如果对 undefined 或 null 进行解构，会报错。

```
var [foo] = undefined;//TypeError: undefined has no properties
var [foo] = null;//TypeError: null has no properties
```

这是因为解构只能用于数组或对象。其他原始类型的值都可以转为相应的对象，但是 undefined 和 null 不能转为对象，因此报错。

解构赋值允许指定默认值。

```
var [foo = true] = []; // foo = true
var [x, y = 'b'] = ['a']; // x='a', y='b'
var [x, y = 'b'] = ['a', undefined]; // x='a', y='b'
```

对于 Set 结构及某种数据结构具有 Iterator 接口的都可以使用解构赋值。

字符串的解构赋值

由于 JavaScript 引擎内部，某些场合时，字符串会被转为类似数组的对象。因此，字符串也可以解构赋值。

```
const [a, b, c, d, e] = 'hello';
console.log(a);// h
console.log(b);// e
console.log(c);// l
console.log(d);// l
console.log(e);// o
```

类似数组的对象都有一个 length 属性，因此还可以对这个属性结构赋值。

```
let {length : i} = 'hello';
console.log(i);//5
let {length : l} = [];
console.log(l);//0
```

对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

```
var { foo, bar } = { foo: "aaa", bar: "bbb" };
console.log(foo);// "aaa"
console.log(bar);// "bbb"
```

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

```
var { b, a } = { a: "aaa", b: "bbb" };
console.log(a);// aaa
console.log(b);// bbb

var { baz } = { foo: "aaa", bar: "bbb" };
console.log(baz);// undefined
```

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于 undefined。

如果变量名与属性名不一致，必须写成下面这样。

```
let { first: f, last: l } = { first: 'hello', last: 'world' };
console.log(f);//'hello'
console.log(l);//'world'
```

和数组一样，解构也可以用于嵌套结构的对象。

```
var { p: [x, { y }] } = { p: [ "Hello",{ y: "World" } ] };
console.log(x);// "Hello"
console.log(x);// "World"
```

对象的解构也可以指定默认值。

```
var { x = 3 } = {}; // x = 3
```

对象解构可以与函数参数的默认值一起使用。

```
function fun({x = 0, y = 0} = {}) {
    return [x, y];
}
fun({x: 3, y: 8}); // [3, 8]
fun({x: 3}); // [3, 0]
fun({}); // [0, 0]
fun(); // [0, 0]
```

上面代码中，函数 fun 的参数是一个对象，通过对这个对象进行解构，得到变量 x 和 y 的值。如果解构失败，x 和 y 等于默认值。

注意，指定函数参数的默认值时，不能采用下面的写法。

```
function fun({x, y} = { x: 0, y: 0 }) {
    return [x, y];
```

```
}
fun({x: 3, y: 8}); // [3, 8]
fun({x: 3}); // [3, undefined]
fun({}); // [undefined, undefined]
fun(); // [0, 0]
```

上面代码是为函数 fun 的参数指定默认值，而不是为变量 x 和 y 指定默认值，所以会得到与前一种写法不同的结果。

如果要将一个已经声明的变量用于解构赋值，必须非常小心。

```
var x;
{x} = {x:1};// SyntaxError: syntax error
```

上面代码的写法会报错，因为 JavaScript 引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免 JavaScript 将其解释为代码块，才能解决这个问题。

```
// 正确的写法
({x}) = {x:1};
// 或者
({x} = {x:1});
```

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

```
let { log, sin, cos,random } = Math;
console.log(random());//0.8954327846785
```

用途

（1）交换变量的值

```
[x, y] = [y, x];//上面代码交换变量 x 和 y 的值，这样的写法不仅简洁，而且易读，语义非常清晰。
```

（2）从函数返回多个值

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

```
function example1() {
    return [1, 2, 3];
}
var [a, b, c] = example1();

function example2() {
    return {foo: 1,bar: 2};
}
var { foo, bar } = example2();
```

（3）函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { }
```

```
f([1, 2, 3]);

// 参数是一组无次序的值
function fu({x, y, z}) { }
fu({x:1, z:2, y:3});
```

（4）提取 JSON 数据

解构赋值对提取 JSON 对象中的数据，尤其有用。

```
var jsonData = {
    id: 42,
    status: "OK",
    data: [867, 5309]
};
let { id, status, data: number } = jsonData;
console.log(id, status, number);
// 42, OK, [867, 5309]
```

（5）函数参数的默认值

```
function f({async = true, foo = 1}) {}
```

指定参数的默认值，就避免了在函数体内部再写 var foo = config.foo || 'default foo';这样的语句。

（6）遍历 Map 结构

任何部署了 Iterator 接口的对象，都可以用 for...of 循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。

```
var map = new Map();
map.set('first', 'hello');
map.set('second', 'world');
for (let [key, value] of map) {
    console.log(key + " is " + value);
}

for (let [key] of map) {
    // 获取键名
}

// 获取键值
for (let [,value] of map) {
    // 获取键值
}
```

（7）输入模块的指定方法

加载模块时，往往需要指定输入那些方法。解构赋值使得输入语句非常清晰。

```
let obj = require("something"); //普通写法
obj.a();
obj.b();
```

```
let { a, b } = require("something");//解构变量写法
a();
b();
```

第三讲 String 字符串的扩展

String.codePointAt()

ES6 加强了对 Unicode 的支持，并且扩展了字符串对象。JavaScript 内部，字符以 UTF-16 的格式储存，每个字符固定为 2 个字节。对于那些需要 4 个字节储存的字符(Unicode 码点大于 0xFFFF 的字符)，JavaScript 会认为它们是两个字符。

```
var s = "吉";
s.length // 2
s.charAt(0) // ''
s.charAt(1) // ''
s.charCodeAt(0) // 55362
s.charCodeAt(1) // 57271
```

上面代码中，汉字“吉”的码点是 0x20BB7，UTF-16 编码为 0xD842 0xDFB7（十进制为 55362 57271），需要 4 个字节储存。对于这种 4 个字节的字符，JavaScript 不能正确处理，字符串长度会误判为 2，而且 charAt 方法无法读取字 符，charCodeAt 方法只能分别返回前两个字节和后两个字节的值。

ES6 提供了 codePointAt 方法，能够正确处理 4 个字节储存的字符，返回一个字符的码点。

```
var s = "吉a";
s.codePointAt(0) // 134071
s.codePointAt(1) // 57271
s.charCodeAt(2) // 97
```

codePointAt 方法的参数，是字符在字符串中的位置（从 0 开始）。上面代码中，JavaScript 将“吉a”视为三个字 符，codePointAt 方法在第一个字符上，正确地识别了“吉”，返回了它的十进制码点 134071（即十六进制的 20BB7）。在第二个字符（即 “吉”的后两个字节）和第三个字符“a”上，codePointAt 方法的结果与 charCodeAt 方法相同。

总之，codePointAt 方法会正确返回四字节的 UTF-16 字符的码点。对于那些两个字节储存的常规字符，它的返回结果与 charCodeAt 方法相同。

String.fromCodePoint()

ES5 提供 String.fromCharCode 方法，用于从码点返回对应字符，但是这个方法不能识别辅助平面的字符（编号大于 0xFFFF）。

```
String.fromCharCode(0x20BB7);

// "ஷ" 返回码点 U+0BB7 对应的字符，而不是码点 U+20BB7 对应的字符

String.fromCodePoint(0x20BB7);
// "吉" ES6fromCodePoint 正确返回码点对应的字符
```

String.At()

ES5 提供 String.prototype.charAt 方法，返回字符串给定位置的字符。该方法不能识别码点大于 0xFFFF 的字符。

```
'吉'.charAt(0);
// '\uD842'
'吉'.at(0);
// '吉'
```

字符的 Unicode 表示法

JavaScript 允许采用“\uxxxx”形式表示一个字符，其中“xxxx”表示字符的码点。但是，这种表示法只限于\u0000——\uFFFF 之间的字符。超出这个范围的字符，必须用两个双字节的形式表达。

```
"\u0061";  // "a"
"\uD842\uDFB7";   // "吉"
"\u20BB7"    // "  7"
```

上面代码表示，如果直接在“\u”后面跟上超过 0xFFFF 的数值（比如\u20BB7），JavaScript 会理解成“\u20BB+7”。由于\u20BB 是一个不可打印字符，所以只会显示一个空格，后面跟着一个 7。

ES6 对这一点做出了改进，只要将码点放入大括号，就能正确解读该字符。

```
"\u{20BB7}" ;   // "吉"
"\u{41}\u{42}\u{43}" ;   // "ABC"
```

正则表示式的 u 修饰符

ES6 对正则表达式添加了 u 修饰符，用来正确处理大于\uFFFF 的 Unicode 字符。

```
/^.$/.test("吉"); // false
/^.$/u.test("吉"); // true

/\u{61}/.test('a'); // false
/\u{61}/u.test('a'); // true
/\u{20BB7}/u.test('吉'); // true

/吉{2}/.test('吉吉'); // false
/吉{2}/u.test('吉吉'); // true
```

```
/^\S$/.test('吉'); // false
/^\S$/u.test('吉');// true

//\u004B 与\u212A 都是大写的 K
/[a-z]/i.test('\u212A'); // false
/[a-z]/iu.test('\u212A'); // true
```

Normalize()

为了表示语调和重音符号，Unicode 提供了两种方法。一种是直接提供带重音符号的字符，比如Ǒ（\u01D1）。另一种是提供合成符号 （combining character），即原字符与重音符号的合成，两个字符合成一个字符，比如 O（\u004F）和 ˇ （\u030C）合成Ǒ（\u004F \u030C）。

```
'\u01D1'==='\u004F\u030C' //false
'\u01D1'.length; // 1
'\u004F\u030C'.length; // 2

'\u01D1'.normalize() === '\u004F\u030C'.normalize();    // true
```

normalize 方法可以接受四个参数:

- NFC　　默认参数，表示“标准等价合成”（Normalization Form Canonical Composition），返回多个简单字符的合成字符。所谓“标准等价”指的是视觉和语义上的等价。
- NFD　　表示“标准等价分解”（Normalization Form Canonical Decomposition），即在标准等价的前提下，返回合成字符分解的多个简单字符。
- NFKC　　表示“兼容等价合成”（Normalization Form Compatibility Composition），返回合成字符。所谓“兼容等价”指的是语义上存在等价，但视觉上不等价，比如“囍”和“喜喜”。
- NFKD　　表示“兼容等价分解”（Normalization Form Compatibility Decomposition），即在兼容等价的前提下，返回合成字符分解的多个简单字符。

Includes(),startsWith(),endsWith()

传统上，JavaScript 只有 indexOf 方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6 又提供了三种新方法。

- includes()　　返回布尔值，表示是否找到了参数字符串；
- startsWith()　　返回布尔值，表示参数字符串是否在源字符串的头部；
- endsWith()　　返回布尔值，表示参数字符串是否在源字符串的尾部；

```
var s = "Hello world!";
s.startsWith("Hello"); // true
s.endsWith("!"); // true
s.includes("o"); // true
```

```
//这三个方法都支持第二个参数，表示开始搜索的位置，从左边 1 开始算起。
s.startsWith("world", 6); // true  表示第 6 个字符以后的字符
s.endsWith("Hello", 5); // true  表示第 1 和第 5 个字符之间的字符
s.includes("Hello", 5); // false  表示第 5 个字符以后的字符
```

String.repeat()

repeat()返回一个新字符串，表示将原字符串重复 n 次。

```
"x".repeat(3); // "xxx"
"hello".repeat(2); // "hellohello"
```

正则表达式的 y 修饰符

除了 u 修饰符，ES6 还为正则表达式添加了 y 修饰符，叫做**“粘连”**（sticky）修饰符。它的作用与 g 修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始，不同之处在于 g 修饰符只要剩余位置中存在匹配就可，而 y 修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

```
var str = "aaa_aa_a";
str.match(/a+/g); // ["aaa","aa","a"]
str.match(/a+/y); // ["aaa"]  第二次匹配等同于 "_aa_a".match(/^a+/);所以没有结果
```

上面代码有两个正则表达式，一个使用 g 修饰符，另一个使用 y 修饰符。这两个正则表达式第一次匹配后都是剩余字符串“_aa_a”。由于 g 修饰没有位置要求，所以仍能匹配出两次结果，而 y 修饰符要求匹配必须从头部开始，所以返回 null。y 修饰符的设计本意，就是让头部匹配的标志^在全局匹配中都有效。

```
var r = /hello\d/y;
r.sticky // true    表示是否设置了 y 修饰符
```

Regexp.escape()

必须使用反斜杠对其中的特殊字符转义，才能用来作为一个正则匹配的模式。已经有提议将这个需求标准化，作为 Regexp.escape()，放入 ES7。

```
RegExp.escape("(*.*)"); // "\(\*\.\*\)"
//等同于以下函数
function escapeRegExp(str) {
    return str.replace(/[\-\[\]\/\{\}\(\)\*\+\?\.\\\^\$\|]/g, "\\$&");
}
```

模板字符串

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，

也可以用来定义多行字符串，或者在字符串中嵌入变量。

```
`In JavaScript is a line-feed.`;// 普通字符串

`In JavaScript this is
not legal.`;// 多行字符串

var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`;// 字符串中嵌入变量

var x = 1,y = 2;
console.log(`${x} + ${y} = ${x+y}`);//字符串内仍可数学计算

function fn() {
    return "Hello World";
}
console.log(`foo ${fn()} bar`); // 字符串内可调用函数

var msg = `Hello, ${place}`; // throws error
```

标签模板

模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。

```
var a = 5;
var b = 10;
function tag(arr, v1, v2) {
    console.log(arr[0]);//Hello\n
    console.log(arr.raw[0]);//Hello\\n
    console.log(arr[1]);// world
    console.log(v1);//15 也就是${a + b}
    console.log(v2);//50 也就是${a * b}
}
tag `Hello\n ${ a + b } world ${ a * b}`;//等同于 tag(['Hello\n ', ' world ', ''], 15, 50)
```

如上函数 tag，当一个模板调用此函数时，第一个变量是一个数组。这个数组是以模板里的变量作为分割线分割模板得出的数组，而所有变量则作为参数从第二位依次传入函数。

其中 arr 数组还有一个 raw 属性，其作用是转义字符串。

```
function temp(arr,...arrs) {
    for(var i = 0,str = '';i < arr.length; i++){
        str += arr[i] + (arrs[i]||"");// 或 str += arr.raw[i] + (arrs[i]||'')
    }
    return str;
}
```

上面这个例子展示了，如何将各个参数按照原来的位置拼合回去。

模板字符串并不能取代 Mustache 之类的模板函数，因为没有条件判断和循环处理功能，但是通过标签函数，你可以自己添加这些功能。

```
var libraryHtml = hashTemplate`
<ul>
#for book in ${myBooks}
        <li><i>#{book.title}</i> by #{book.author}</li>
#end
</ul>`;
```

除此之外，你甚至可以使用标签模板，在 JavaScript 语言之中嵌入其他语言。

```
java`
class HelloWorldApp {
    public static void main(String[] args) {
    System.out.println("Hello World!"); // Display the string.
}
}
`

HelloWorldApp.main();
```

String.raw()

String.raw 方法，往往用来充当模板字符串的处理函数，返回字符串被转义前的原始格式。

```
String.raw`Hi\n${2+3}!`;   // "Hi\\n5!"
String.raw`Hi\u000A!`;   // 'Hi\\u000A!'
```

String.raw 方法也可以正常的函数形式使用。这时，它的第一个参数，应该是一个具有 raw 属性的对象，且 raw 属性的值应该是一个数组。

```
String.raw({ raw: 'test' }, 0, 1, 2);  // 't0e1s2t'
// 等同于
String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2);
```

第四讲 Number 数值的扩展

二进制和八进制表示法

ES6 提供了二进制和八进制数值的新的写法，分别用前缀 0b 和 0o 表示。八进制用 0o 前缀表示的方法，将要取代已经在 ES5 中被逐步淘汰的加前缀 0 的写法。

```
0b111110111 === 503 // true
0o767 === 503 // true
```

Number.isFinite(), Number.isNaN()

ES6 在 Number 对象上，新提供了 Number.isFinite()和 Number.isNaN()两个方法，用来检查 Infinite 和 NaN 这两个特殊值。

Number.isFinite()用来检查一个数值是否是有限数字（finity）。

```
Number.isFinite(15); // true
Number.isFinite(0.8); // true
Number.isFinite(NaN); // false
Number.isFinite(Infinity); // false
Number.isFinite(-Infinity); // false
Number.isFinite("foo"); // false
Number.isFinite("15"); // false  注意没有隐性转换数值
Number.isFinite(true); // false
```

Number.isNaN()用来检查一个值是否为 NaN。

```
Number.isNaN(NaN); // true
Number.isNaN(15); // false
Number.isNaN("15"); // false
Number.isNaN(true); // false
```

它们与传统的全局方法 isFinite()和 isNaN()的区别在于，传统方法先调用 Number()将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，非数值一律返回 false。

```
isFinite(25); // true
isFinite("25"); // true
Number.isFinite(25); // true
Number.isFinite("25"); // false

isNaN(NaN); // true
isNaN("NaN"); // true
Number.isNaN(NaN); // true
Number.isNaN("NaN"); // false
```

Number.parseInt(), Number.parseFloat()

ES6 将全局方法 parseInt()和 parseFloat()，移植到 Number 对象上面，行为完全保持不变。

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

Number.isInteger()和安全整数

Number.isInteger()用来判断一个值是否为整数。需要注意的是，在 JavaScript 内部，整数和浮点数是同样的储存方法，所以 3 和 3.0 被视为同一个值。

```
Number.isInteger(25) ; // true
Number.isInteger(25.0) ; // true
Number.isInteger(25.1) ; // false
Number.isInteger("15") ; // false
Number.isInteger(true) ; // false
```

JavaScript 能够准确表示的整数范围在-2^53 and 2^53 之间。ES6 引入了 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 这两个常量，用来表示这个 范围的上下限。Number.isSafeInteger()则是用来判断一个整数是否落在这个范围之内。

```
var inside = Number.MAX_SAFE_INTEGER;
var outside = inside + 1;

Number.isInteger(inside); // true
Number.isSafeInteger(inside); // true

Number.isInteger(outside); // true
Number.isSafeInteger(outside); // false
```

Math 对象的扩展

```
//Math.trunc 方法用于去除一个数的小数部分，返回整数部分。
Math.trunc(4.1) ; // 4
Math.trunc(4.9) ; // 4
Math.trunc(-4.1) ; // -4
Math.trunc(-4.9) ; // -4

//Math.sign 方法用来判断一个数到底是正数、负数、还是零。如果参数为正数，返回+1；参数为负数，返回-1；参数为 0，返回 0；参数为 NaN，返回 NaN。
Math.sign(-5) ; // -1
Math.sign(5) ; // +1
Math.sign(0) ; // +0
Math.sign(-0) ; // -0
Math.sign(NaN) ; // NaN
```

ES6 在 Math 对象上还提供了许多新的数学方法

- Math.acosh(x)　　返回 x 的反双曲余弦（inverse hyperbolic cosine）
- Math.asinh(x)　　返回 x 的反双曲正弦（inverse hyperbolic sine）
- Math.atanh(x)　　返回 x 的反双曲正切（inverse hyperbolic tangent）
- Math.cbrt(x)　　返回 x 的立方根
- Math.clz32(x)　　返回 x 的 32 位二进制整数表示形式的前导 0 的个数
- Math.cosh(x)　　返回 x 的双曲余弦（hyperbolic cosine）
- Math.expm1(x)　　返回 e^x - 1
- Math.fround(x)　　返回 x 的单精度浮点数形式
- Math.hypot(...values)　返回所有参数的平方和的平方根
- Math.imul(x, y)　　返回两个参数以 32 位整数形式相乘的结果

- Math.log1p(x)　　返回 1 + x 的自然对数
- Math.log10(x)　　返回以 10 为底的 x 的对数
- Math.log2(x)　　返回以 2 为底的 x 的对数
- Math.tanh(x)　　返回 x 的双曲正切（hyperbolic tangent）

第五讲　Array 数组的扩展

Array.from()

Array.from 方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）。

```
let ps = document.querySelectorAll('p');
Array.from(ps).forEach(function (p) {
    console.log(p);
});

//Array.from 方法可以将函数的 arguments 对象，转为数组。
function foo() {
    var args = Array.from( arguments );
}

//任何有 length 属性的对象，都可以通过 Array.from 方法转为数组
Array.from({ 0: "a", 1: "b", 2: "c", length: 3 });   // [ "a", "b" , "c" ]

//对于还没有部署该方法的浏览器，可以用 Array.prototyp.slice 方法替代。
const toArray = (() =>
    Array.from ? Array.from : obj => [].slice.call(obj);
)();

//Array.from()还可以接受第二个参数，作用类似于数组的 map 方法，用来对每个元素进行处理
Array.from(arrayLike, x => x * x);
// 等同于
Array.from(arrayLike).map(x => x * x);

//下面的例子将数组中布尔值为 false 的成员转为 0
Array.from([1, , 2, , 3], (n) => n || 0);  // [1, 0, 2, 0, 3]

//Array.from()的一个应用是,将字符串转为数组,然后返回字符串的长度,这样可以避免 JavaScript 将大于\uFFFF 的 Unicode 字符，算作两个字符的 bug
function countSymbols(string) {
    return Array.from(string).length;
}
```

Array.of()

Array.of 方法用于将一组值，转换为数组。这个方法的主要目的，是弥补数组构造函数 Array()的不足。因为参数个数的不同，会导致 Array()的行为有差异。

```
Array.of(3, 11, 8); // [3,11,8]
Array.of(3); // [3]
Array.of(3).length; // 1

//Array.of 方法可以用下面的代码模拟实现。
function ArrayOf(){
    return [].slice.call(arguments);
}
```

数组实例的 find()和 findIndex()

数组实例的 find 和 findIndex 方法，都用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为 true 的成员，然后 find 返回该成员或 undefined，findexIndex 返回该成员的位置或-1。

```
[1, 4, -5, 10].find(function(value, index, arr) {
    return value < 0;
}); //-5
[1, 5, 10, 15].findIndex(function(value, index, arr) {
    return value > 9;
}); // 2
//indexOf 方法无法识别数组的 NaN 成员，但是 findIndex 方法可以借助 Object.is 方法做到
[NaN].indexOf(NaN);// -1
[NaN].findIndex(y => Object.is(NaN, y));    // 0
```

数组实例的 fill()

fill()使用给定值，填充一个数组。fill()还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

```
['a', 'b', 'c'].fill(7); // [7, 7, 7]
new Array(3).fill(7);    // [7, 7, 7]
['a', 'b', 'c'].fill(7, 1, 2);  // ['a', 7, 'c']
```

数组实例的 entries()，keys()和 values()

ES6 提供三个新的方法——entries()，keys()和 values()——用于遍历数组。它们都返回一个遍历器，

可以用for...of循环进 行遍历,唯一的区别是keys()是对键名的遍历、values()是对键值的遍历,entries()是对键值对的遍历。

```
for (let index of ['a', 'b'].keys()) {
    console.log(index);// 0 ,  1
}
for (let elem of ['a', 'b'].values()) {
    console.log(elem);// 'a', 'b'
}
for (let [index, elem] of ['a', 'b'].entries()) {
    console.log(index, elem); // 0 'a', 1 'b'
}
```

数组实例的 includes()

Array.prototype.includes 方法返回一个布尔值，表示某个数组是否包含给定的值。该方法属于 ES7。该方法的第二个参数表示搜索的起始位置，默认为 0。此功能暂可使用 Array.some 顶代。

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, NaN].includes(NaN,1); // true
Array.prototype.includes = function(value,index){
    return this.slice(index || 0).some(function(v){
        return v === value;
    })
}
```

数组推导

ES6 提供简洁写法，允许直接通过现有数组生成新数组，这被称为**数组推导**（array comprehension）。for...of 后面还可以附加 if 语句，用来设定循环的限制条件。

```
var a1 = [1, 2, 3, 4];
var a2 = [for (i of a1) i * 2]; // [2, 4, 6, 8]

var years = [ 1954, 1974, 1990, 2006, 2010, 2014 ];
[for (year of years) if (year > 2000) year]; // [ 2006, 2010, 2014 ]

[for (year of years) if (year > 2000) if(year < 2010) year]; // [ 2006]
[for (year of years) if (year > 2000 && year < 2010) year];  // [ 2006] 等同上
```

数组推导可以替代 map 和 filter 方法。在一个数组推导中，还可以使用多个 for...of 结构，构成多重循环。

```
[for (i of [1, 2, 3]) i * i];
[1, 2, 3].map(function (i) { return i * i });// 等同上
```

```
[for (i of [1,4,2,3,-8]) if (i < 3) i];
[1,4,2,3,-8].filter(function(i) { return i < 3 });// 等同上

var a1 = ["x1", "y1"];
var a2 = ["x2", "y2"];
var a3 = ["x3", "y3"];
[for (s of a1) for (w of a2) for (r of a3) s + w + r];
//["x1x2x3", "x1x2y3", "x1y2x3", "x1y2y3", "y1x2x3", "y1x2y3", "y1y2x3", "y1y2y3"]
```

由于字符串可以视为数组，因此字符串也可以直接用于数组推导。

```
[for (c of 'abcde') if (/[aeiou]/.test(c)) c].join(''); // 'ae'
[for (c of 'abcde') c+'0'].join(''); // 'a0b0c0d0e0'
```

上面代码使用了数组推导，对字符串进行处理。数组推导需要注意的地方是，新数组会立即在内存中生成。这时，如果原数组是一个很大的数组，将会非常耗费内存。

Array.observe()，Array.unobserve()

这两个方法用于**监听**（取消监听）数组的变化，指定回调函数。

它们的用法与 Object.observe 和 Object.unobserve 方法完全一致，也属于 ES7 的一部分，请参阅后继章节。唯一的区别是，对象可监听的变化一共有六种，而数组只有四种：add、update、delete、splice（数组的 length 属性发生变化）。

第六讲　Object 对象的扩展

属性的简洁表示法

ES6 允许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。

```
var Person = {
    name: '传统形式',
    birth,//等同于 birth: birth
    hello() { console.log('我的名字是', this.name); }// 等同于 hello: function ()...
}
var x = 1, y = 10;
var o = {x, y};//等同于{x:1,y:10}
```

属性名的表达式

JavaScript 语言定义对象的属性，有两种方法。ES6 允许字面量定义对象时，用方法二（表达式）作为对象的属性名，即把表达式放在方括号内。

```
// ES5 传统定义属性方法
obj.foo = true;
obj['f'+'oo'] = true;

let propKey = 'foo';
let f = 'fun';
let obj = {
    [propKey]: true,//ES6 属性名表达式
    ['a'+'bc']: 123,
    [f](){
        console.log('表达式还可定义函数');
    }
};
```

Object.is()

Object.is()用来比较两个值是否严格相等。它与严格比较运算符（===）的行为基本一致，不同之处只有两个：一是+0 不等于-0，二是 NaN 等于自身。

```
+0 === -0; //true
NaN === NaN; // false

Object.is(+0, -0); // false
Object.is(NaN, NaN); // true
```

Object.assign()

Object.assign 方法用来将源对象（source）的所有可枚举属性，复制到目标对象（target）。它至少需要两个对象作为参数，第一个参数是目标对象，后面的参数都是源对象。只要有一个参数不是对象，就会抛出 TypeError 错误。

```
var target = { a: 1,b: 1 };
var source1 = { b: 2,c: 2 };
var source2 = { c: 3 };
Object.assign(target, source1, source2);// target = {a:1, b:2, c:3}
```

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

Assign 方法作用很多：

```
// (1) 为对象添加属性
class Point {
    constructor(x, y) {
        Object.assign(this, {x, y});//将 x 属性和 y 属性添加到 Point 类的对象实例
    }
}
```

```
// (2) 为对象添加方法
Object.assign(SomeClass.prototype, {
    someMethod(arg1, arg2) {},
    anotherMethod() {}
});
// 等同于下面的写法
SomeClass.prototype.someMethod = function (arg1, arg2) {};
SomeClass.prototype.anotherMethod = function () {};

// (3) 克隆对象
function clone(origin) {
    return Object.assign({}, origin);//只能克隆原始对象自身的值，不能克隆它继承的值
}
function clones(origin) {
    let originProto = Object.getPrototypeOf(origin);
    return Object.assign(Object.create(originProto), origin);//同时克隆它的继承值
}

// (4) 合并多个对象
const merge = (target, ...sources) => Object.assign(target, ...sources);

// (5) 为属性指定默认值
const DEFAULTS = { logLevel: 0, outputFormat: 'html'};
function processContent(options) {
    let options = Object.assign({}, DEFAULTS, options);
}
```

__proto__，Object.setPrototypeOf()，Object.getPrototypeOf()

__proto__属性，用来读取或设置当前对象的 prototype 对象。该属性一度被正式写入 ES6 草案，但后来又被移除。目前，所有浏览器（包括 IE11）都部署了这个属性。

```
// es6 的写法
var obj = {
    __proto__: someOtherObj,
    method: function() {}
};
// es5 的写法
var obj = Object.create(someOtherObj);
obj.method = function() {}
```

Object.setPrototypeOf 方法的作用与__proto__相同，用来设置一个对象的 prototype 对象。它是 ES6 正式推荐的设置原型对象的方法。

```
// 格式
Object.setPrototypeOf(object, prototype);
```

```
// 用法
var o = Object.setPrototypeOf({}, null);
//该方法等同于下面的函数
function(obj, proto) {
    obj.__proto__ = proto;
    return obj;
}
```

Object.getPrototypeOf()该方法与 setPrototypeOf 方法配套，用于读取一个对象的 prototype 对象。

```
Object.getPrototypeOf(obj);//返回 obj 的 prototype
```

Symbol 概述

在 ES5 中，对象的属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法，新方法的名字有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入 Symbol 的原因。

ES6 引入了一种新的原始数据类型 Symbol，表示独一无二的 ID。它通过 Symbol 函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

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

上面代码中，变量 s 就是一个独一无二的 ID。typeof 运算符的结果，表明变量 s 是 Symbol 数据类型，而不是字符串之类的其他类型。

注意，Symbol 函数前不能使用 new 命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。也就是说，由于 Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

Symbol 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述。

```
var mySymbol = Symbol('Test');
console.log(mySymbol.length);// 1
console.log(Symbol.prototype);// Symbol{}
console.log(mySymbol.name);// Test
```

Symbol 函数的参数只是表示对当前 Symbol 类型的值的描述，因此相同参数的 Symbol 函数的返回值是不相等的。

```
Symbol() === Symbol(); // false
Symbol("foo") === Symbol("foo"); // false 注意它每次都创建一个新的符号

//Symbol 类型的值不能与其他类型的值进行运算，会报错。
var sym = Symbol('My symbol');
"your symbol is " + sym;    // TypeError: can't convert symbol to string
```

```
`your symbol is ${sym}`      // TypeError: can't convert symbol to string

//但是，Symbol 类型的值可以转为字符串。
String(sym);   // 'Symbol(My symbol)'
sym.toString();    // 'Symbol(My symbol)'
```

作为属性名的 Symbol

Symbol 类型作为标识符，用于对象的属性名时，保证了属性名之间不会发生冲突。如果一个对象由多个模块构成，这样就不会出现同名的属性，也就防止了键值被不小心改写或覆盖。Symbol 类型还可以用于定义一组常量，防止它们的值发生冲突。

```
var mySymbol = Symbol();
var a = {};
a[mySymbol] = 'Hello!';// 第一种写法
var a = {[mySymbol]: 123};// 第二种写法
Object.defineProperty(a, mySymbol, { value: 'Hello!' });// 第三种写法
// 以上写法都得到同样结果
console.log(a[mySymbol]); // "Hello!"

console.log(a.mySymbol); // 错误的写法，你懂的

//神奇的效果就在这里，如果 a、b 是其他任意字符串结果就不一样了。
var a = Symbol();
var b = Symbol();
var obj = {
        [a]:"a",
        [b]:"b"
};
console.log(obj[a] === obj[b]);// false
console.log(obj);// Object{} 而且是不可枚举哟
```

Symbol.for()，Symbol.keyFor()

Symbol.for 方法在全局环境中搜索指定 key 的 Symbol 值，如果存在就返回这个 Symbol 值，否则就新建一个指定 key 的 Symbol 值并返回。Symbol.for()与 Symbol()这两种写法，都会生成新的 Symbol。它们的区别是，前者会被登记在全局环境中供搜索，后者不会。Symbol.for()不会每次调用就返回一个新的 Symbol 类型的值，而是会先检查跟定的 key 是否已经存在，如果不存在才会新建一个值。比如，如果你调用 Symbol.for("cat")30 次，每次都会返回同一个 Symbol 值，但是调用 Symbol("cat")30 次，会返回 30 个不同的 Symbol 值。

```
Symbol.for("bar") === Symbol.for("bar");  // true
Symbol("bar") === Symbol("bar");  // false
```

Symbol.keyFor 方法返回一个已登记的 Symbol 类型值的 key。

```
var s1 = Symbol.for("foo");//已登记
Symbol.keyFor(s1); // "foo"
var s2 = Symbol("foo");//未登记所以找不到
Symbol.keyFor(s2); // undefined
```

属性名的遍历

Symbol 作为属性名，该属性不会出现在 for...in 循环中，也不会被 Object.keys()、Object.getOwnPropertyNames()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。

Object.getOwnPropertySymbols 方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。

```
var obj = {};
var a = Symbol('a');
var b = Symbol.for('b');
obj[a] = 'Hello';
obj[b] = 'World';
Object.getOwnPropertySymbols(obj); // [Symbol(a), Symbol(b)]
```

另一个新的 API，Reflect.ownKeys 方法可以返回所有类型的键名，包括常规键名和 Symbol 键名。

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

内置的 Symbol 值

除了定义自己使用的 Symbol 值以外，ES6 还提供一些内置的 Symbol 值，指向语言内部使用的方法。

（1）Symbol.hasInstance

该值指向对象的内部方法@@hasInstance（两个@表示这是内部方法，外部无法直接调用，下同），该对象使用 instanceof 运算符时，会调用这个方法，判断该对象是否为某个构造函数的实例。

（2）Symbol.isConcatSpreadable

该值指向对象的内部方法@@isConcatSpreadable，该对象使用 Array.prototype.concat()时，会调用这个方法，返回一个布尔值，表示该对象是否可以扩展成数组。

（3）Symbol.isRegExp

该值指向对象的内部方法@@isRegExp，该对象被用作正则表达式时，会调用这个方法，返回一个布尔值，表示该对象是否为一个正则对象。

（4）Symbol.match

该值作为属性名时，返回对象的正则表达式形式。当执行 str.match(myObject)时，如果该属性存在，会先查看 myObject[Symbol.match]属性是否存在。

（5）Symbol.iterator

该值指向对象的内部方法@@iterator，该对象进行 for...of 循环时，会调用这个方法，返回该对象的默认遍历器，后继有介绍。

（6）Symbol.toPrimitive

该值指向对象的内部方法@@toPrimitive，该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。

（7）Symbol.toStringTag

该值指向对象的内部属性@@toStringTag，在该对象上调用 Object.prototype.toString()时，会返回这个属性，它是一个字符串，表示该对象的字符串形式。

（8）Symbol.unscopables

该值指向对象的内部属性@@unscopables，返回一个数组，成员为该对象使用 with 关键字时，会被 with 环境排除的那些属性值。

Proxy

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），即对编程语言进行编程。

Proxy 可以理解成在目标对象之前，架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作。

ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例。

```
var proxy = new Proxy({}, {
    get: function(target, property) {
        return 35;
    }
});
proxy.time; // 35
proxy.name; // 35
proxy.title; // 35
```

Proxy 实例也可以作为其他对象的原型对象。

```
var proxy = new Proxy({}, {
    get: function(target, property) {
        return 35;
    }
});
let obj = Object.create(proxy);
obj.time; // 35  访问了原型链
```

Proxy(target, handler)，这里的 handler 有如下的方法：

- get(target, propKey, receiver)：拦截对象属性的读取，比如 proxy.foo 和 proxy['foo']，返回类型不限。最后一个参数 receiver 可选，当 target 对象设置了 propKey 属性的 get 函数时，receiver 对象会绑定 get 函数的 this 对象。
- set(target, propKey, value, receiver)：拦截对象属性的设置，比如 proxy.foo = v 或 proxy['foo'] = v，返回一个布尔值。
- has(target, propKey)：拦截 propKey in proxy 的操作，返回一个布尔值。
- deleteProperty(target, propKey) ：拦截 delete proxy[propKey]的操作，返回一个布尔值。
- enumerate(target)：拦截 for (var x in proxy)，返回一个遍历器。
- hasOwn(target, propKey)：拦截 proxy.hasOwnProperty('foo')，返回一个布尔值。
- ownKeys(target)：拦截 Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回对象所有自身的属性，而 Object.keys()仅返回对象可遍历的属性。
- getOwnPropertyDescriptor(target, propKey) ：拦截 Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
- defineProperty(target, propKey, propDesc)：拦截 Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
- preventExtensions(target)：拦截 Object.preventExtensions(proxy)，返回一个布尔值。
- getPrototypeOf(target) ：拦截 Object.getPrototypeOf(proxy)，返回一个对象。
- isExtensible(target)：拦截 Object.isExtensible(proxy)，返回一个布尔值。
- setPrototypeOf(target, proto)：拦截 Object.setPrototypeOf(proxy, proto)，返回一个布尔值。

如果目标对象是函数，那么还有两种额外操作可以拦截：

- apply(target, object, args)：拦截 Proxy 实例作为函数调用的操作，比如 proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
- construct(target, args, proxy)：拦截 Proxy 实例作为构造函数调用的操作，比如 new proxy(...args)。

get 方法用于拦截某个属性的读取操作。上文已经有一个例子，下面是另一个拦截读取操作的例子。

```
var person = {
    name: "张三"
};
var proxy = new Proxy(person, {
    get: function(target, property) {
        if (property in target) {
```

```
            return target[property];
        } else {
            throw new ReferenceError("Property \"" + property + "\" does not exist.");
        }
    }
});
proxy.name; // "张三"
proxy.age; // 抛出一个错误
```

set 方法用来拦截某个属性的赋值操作。假定 Person 对象有一个 age 属性，该属性应该是一个不大于 200 的整数，那么可以使用 Proxy 对象保证 age 的属性值符合要求。

```
let validator = {
    set: function(obj, prop, value) {
        if (prop === 'age') {
            if (!Number.isInteger(value)) {
                throw new TypeError('The age is not an integer');
            }
            if (value > 200) {
                throw new RangeError('The age seems invalid');
            }
        }
        // 对于 age 以外的属性，直接保存
        obj[prop] = value;
    }
};
let person = new Proxy({}, validator);
person.age = 100;
person.age; // 100
person.age = 'young'; // 报错
person.age = 300; // 报错
```

apply 方法拦截函数的调用、call 和 apply 操作。

```
var p = new Proxy(function() {}, {
    apply: function(target, thisArg, argumentsList) {
        console.log("stop: " + argumentsList.join(", "));
        return argumentsList[0] + argumentsList[1] + argumentsList[2];
    }
});
console.log(p(1, 2, 3)); // "stop: 1, 2, 3"     6
```

ownKeys 方法用来拦截 Object.keys()操作。

```
let target = {};
let handler = {
    ownKeys(target) {
        return ['hello', 'world'];
    }
};
```

```
let proxy = new Proxy(target, handler);
Object.keys(proxy);  // [ 'hello', 'world' ]
```

Proxy.revocable 方法返回一个可取消的 Proxy 实例。

```
var revocable = Proxy.revocable({}, {
    get(target, name) {
        return "[[" + name + "]]";
    }
});
var proxy = revocable.proxy; // 代理
proxy.foo;              // "[[foo]]"
revocable.revoke();     // 执行撤销方法
proxy.foo;              // TypeError
proxy.foo = 1;          // 同样 TypeError
delete proxy.foo;       // 还是 TypeError
typeof proxy;           // "object", 因为 typeof 不属于可代理操作
```

Object.observe()，Object.unobserve()

Object.observe 方法用来监听对象（以及数组）的变化。一旦监听对象发生变化，就会触发回调函数。

```
var obj = {};
function observer(changes){
    changes.forEach(function(change) {
        console.log('发生变动的属性：' + change.name);
        console.log('变动前的值：' + change.oldValue);
        console.log('变动后的值：' + change.object[change.name]);
        console.log('变动类型：' + change.type);
    });
}
Object.observe(obj, observer);
obj.a = 2;
//发生变动的属性：a
//变动前的值：undefined
//变动后的值：2
//变动类型：add
```

参照上面代码，Object.observe 方法指定的回调函数，接受一个数组（changes）作为参数。该数组的成员与对象的变化一一对应，也就是 说，对象发生多少个变化，该数组就有多少个成员。每个成员是一个对象，它的 name 属性表示发生变化源对象的属性名，oldValue 属性表示发生变化前的值，object 属性指向变动后的源对象，type 属性表示变化的种类。

Object.observe 方法目前共支持监听六种变化。

- Add　　添加属性
- Update　　属性值的变化

- Delete　　删除属性
- setPrototype　　设置原型
- Reconfigure　　属性的 attributes 对象发生变化
- preventExtensions　　对象被禁止扩展（当一个对象变得不可扩展时，也就不必再监听了）

Object.observe 方法还可以接受第三个参数，用来指定监听的事件种类。

```
Object.observe(o, observer, ['delete']);//只在发生 delete 事件时，才会调用回调函数。
Object.unobserve(o, observer);//取消监听
```

注意 Object.observe 和 Object.unobserve 这两个方法不属于 ES6，而是属于 ES7 的一部分。不过 Chrome 浏览器从 33 版起就已经支持。

第七讲　Set 和 Map 数据结构

Set

集合（Set）对象允许你存储任意类型的唯一值（**不能重复**），无论是原始值还是对象引用。

```
var a = new Set([1,2,2,3,4,4,5]);
a.size;//5
for(let i of a){
    console.log(i);//1,2,3,4,5
}
```

向 Set 加入值的时候，不会发生类型转换，所以 5 和“5”是两个不同的值。Set 内部判断两个值是否不同，使用的算法类似于精确相等运算符（===），唯一的例外是 NaN 等于自身。这意味着，两个对象总是不相等的。

```
let set = new Set();
set.add({});
set.add({});
set.size; // 2
```

Set 属性和方法

Set 结构有以下属性：

- Set.prototype.constructor　　构造函数，默认就是 Set 函数。
- Set.prototype.size　　返回 Set 的成员总数。

Set 数据结构有以下方法：

- add(value)　　添加某个值，返回 Set 结构本身。
- delete(value)　　删除某个值，返回一个布尔值，表示删除是否成功。

- has(value)　　返回一个布尔值，表示该值是否为 Set 的成员。
- clear()　　清除所有成员，没有返回值。

```
var s = new Set();
s.add(1).add(2).add(2); // 注意 2 被加入了两次
s.size; // 2
s.has(1); // true
s.has(2); // true
s.has(3); // false
s.delete(2);
s.has(2); // false
function dedupe(array) {
    return Array.from(new Set(array));//数组去重新方法
    //或 return [...new Set(array)];
}
```

Set 遍历操作

Set 结构有一个 values 方法，返回一个遍历器。Set 结构的默认遍历器就是它的 values 方法。这意味着，可以省略 values 方法，直接用 for...of 循环遍历 Set。

```
let set = new Set(['red', 'green', 'blue']);
for ( let item of set.values() ){
    console.log(item);
}
//等同于
for (let x of set) {
    console.log(x);
}

Set.prototype[Symbol.iterator] === Set.prototype.values;  // true
```

为了与 Map 结构保持一致，Set 结构也有 keys 和 entries、forEach、filter 等方法。

```
let set = new Set(['red', 'green', 'blue']);
for ( let item of set.keys() ){
    console.log(item);
}
for ( let [key, value] of set.entries() ){
    console.log(key, value);
}
// 用 forEach 迭代
set.forEach(function(value) {
    return value + "end"; //与其他不同 Set 作 forEach 迭代时不会改变本身的值
});

//因此使用 Set，可以很容易地实现并集（Union）和交集（Intersect）。
```

```
let a = new Set([1,2,3]);
let b = new Set([4,3,2]);
let union = new Set([...a, ...b]);//并集
let intersect = new Set([...a].filter(x => b.has(x)));//交集
```

WeakSet

一个 WeakSet 对象是一个无序的集合,可以用它来存储任意的对象值,并且对这些对象值保持弱引用。

WeakSet 结构与 Set 类似，也是不重复的值的集合，但是它们有两个区别：

- WeakSet 对象中只能存放对象值,不能存放原始值,而 Set 对象是原始值或对象值都可以。
- WeakSet 对象中存储的对象值都是被弱引用的，如果没有其他的变量或属性引用这个对象值，则这个对象值会被当成垃圾回收掉。正因为这样，WeakSet 对象是无法被枚举的，没有办法拿到它包含的所有元素。同时 WeakSet 没有 size 属性，也不可遍历的。

WeakSet 结构有以下三个方法：

- WeakSet.prototype.add(value) 向 WeakSet 实例添加一个新成员。
- WeakSet.prototype.delete(value) 清除 WeakSet 实例的指定成员。
- WeakSet.prototype.clear() 清空 WeakSet 实例的所有成员。
- WeakSet.prototype.has(value) 返回一个布尔值，表示某个值是否在 WeakSet 实例之中。

```
var ws = new WeakSet();
var obj = {};
var foo = {};
ws.add(window);
ws.add(obj);
ws.has(window); // true
ws.has(foo);    // false, 对象 foo 并没有被添加进 ws 中
ws.delete(window); // 从集合中删除 window 对象
ws.has(window);    // false, window 对象已经被删除了
ws.clear(); // 清空整个 WeakSet 对象
```

WeakSet 的一个用处，是储存 DOM 节点，而不用担心这些节点从文档移除时，会引发**内存泄漏**。

Map

Map 对象就是简单的键/值映射，其中键和值可以是任意值(原始值或对象值)。

在判断两个值是否为同一个键的时候，使用的并不是===运算符，而是使用了一种称之为"same-value"的内部算法，该算法很特殊，对于 Map 对象来说，+0(按照以往的经验与 -0 是严格相等的)和-0 是两个不同的键。而 NaN 在作为 Map 对象的键时和另外一个 NaN 是一个相同的键(尽管 NaN !== NaN)。

Map 的方法

- myMap.get(key)　　返回键 key 关联的值,如果该键不存在则返回 undefined ；
- myMap.set(key, value)　　设置键 key 在 myMap 中的值为 value.返回 undefined;
- myMap.has(key)　　返回一个布尔值,表明键 key 是否存在于 myMap 中;
- myMap.delete(key)　　删除键 key 及对应的值. 在这之后, myMap.has(key)将返回 false;
- myMap.clear()　　清空 myMap 中的所有键值对;

```
var myMap = new Map();
var keyObj = {},keyFunc = function () {},keyString = "a string",n = NaN;
// 添加键
myMap.set(keyString, "和键'a string'关联的值");
myMap.set(keyObj, "和键 keyObj 关联的值");
myMap.set(keyFunc, "和键 keyFunc 关联的值");
myMap.set(n, "和键 NaN 关联的值");
myMap.size; // 4
// 读取值
myMap.get(keyString);    // "和键'a string'关联的值"
myMap.get(keyObj);       // "和键 keyObj 关联的值"
myMap.get(keyFunc);      // "和键 keyFunc 关联的值"
myMap.get("a string");   // "和键'a string'关联的值"
                         // 因为 keyString === 'a string'
myMap.get({});           // undefined, 因为 keyObj !== {}
myMap.get(function() {}); // undefined, 因为 keyFunc !== function () {}
myMap.get(NaN); // 和键 NaN 关联的值, 虽然 NaN !==NaN

myMap.set(0, "正零");
myMap.set(-0, "负零");
0 === -0; // true
myMap.get(-0); // "负零"
myMap.get(0);  // "正零"
```

Map 原生提供三个遍历器：

- keys()　　返回键名的遍历器;
- values()　　返回键值的遍历器;
- entries()　　返回所有成员的遍历器;

```
let map = new Map([
    ['F', 'no'],
    ['T',  'yes']
]);
for (let key of map.keys()) {
    console.log(key);
}
for (let value of map.values()) {
    console.log(value);
}
for (let item of map.entries()) {
```

```
    console.log(item[0], item[1]);
}
// 或者
for (let [key, value] of map.entries()) {
    console.log(key, value);
}
// 等同于使用 map.entries()
for (let [key, value] of map) {
    console.log(key, value);
}
map[Symbol.iterator] === map.entries;    // true
```

Object 和 Map 的区别：

- 一个对象通常都有自己的原型，所以一个对象总有一个“prototype”键。不过，现在可以使用 map = Object.create(null)来创建一个没有原型的对象。
- 一个对象的键只能是**字符串**，但一个 Map 的键可以是任意值。
- 你可以很容易的通过 size 属性得到一个 Map 的键值对个数。
- 对象迭代时无法指定顺序，而 Map 是按添加的顺序迭代。

WeakMap

WeakMap 结构与 Map 结构基本类似，唯一的区别是它只接受对象作为键名（null 除外），不接受原始类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。

WeakMap 的设计目的在于，键名是对象的弱引用（垃圾回收机制不将该引用考虑在内），所以其所对应的对象可能会被自动回收。当对象被回收后 WeakMap 自动移除对应的键值对。典型应用是，一个对应 DOM 元素的 WeakMap 结构，当某个 DOM 元素被清除，其所对应的 WeakMap 记录就会自动被移除。基本上，WeakMap 的专用场合就是它的键所对应的对象，可能会在将来消失。WeakMap 结构有助于**防止内存泄漏**。

```
var wm = new WeakMap();
var element = document.querySelector(".element");

wm.set(element, "Original");
wm.get(element); // "Original"

element.parentNode.removeChild(element);
element = null;
wm.get(element); // undefined
```

上面代码中，变量 wm 是一个 WeakMap 实例，我们将一个 DOM 节点 element 作为键名，然后销毁这个节点，element 对应的键就自动消失了，不存在内在泄漏风险，再引用这个键名就返回 undefined。

WeakMap 与 Map 在 API 上的区别主要是两个，一是没有遍历操作（即没有 key()、values()和 entries()方法），也没有 size 属性；二是无法清空，即不支持 clear 方法。这与 WeakMap 的键不被计入引用、被垃

圾回收机制忽略有关。因此，WeakMap 只有四个方法 可用：get()、set()、has()、delete()。

第八讲　iterator 遍历器

概念

JavaScript 原有的数据结构，主要是数组（Array）和对象（Object），ES6 又添加了 Map 和 Set，用户还可以组合使用它们，定义自己的数据结构。这就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它属于一种接口规格，任何数据结构只要部署这个接口，就可以完成遍历操作，即依次处理该结构的所有成员。它的作用有两个，一是为各种数据结构，提供一个统一的、简便的接口，二是使得数据结构的成员能够按某种次序排列。在 ES6 中，遍历操作特指 for...of 循环，即 Iterator 接口主要供 for...of 消费。

遍历器的遍历过程是这样的：它提供了一个指针，默认指向当前数据结构的起始位置。也就是说，遍历器返回一个内部指针。第一次调用遍历器的 next 方法，可以将指针指向到第一个成员，第二次调用 next 方法，就指向第二个成员，直至指向数据结构的结束位置。每一次调用，都会返回当前成员的信息，具体来说，就是返回一个包含 value 和 done 两个属性的对象。其中，value 属性是当前成员的值，done 属性是一个布尔值，表示遍历是否结束。

```
var a = { x: 10,y: 20};
var iter = Iterator(a);
console.log(iter.next()); // ["x", 10]
console.log(iter.next()); // ["y", 20]
console.log(iter.next()); // throws StopIteration

//下面是一个模拟 next 方法返回值的例子
function makeIterator(array){
    var nextIndex = 0;
    return {
        next: function(){
            return nextIndex < array.length ?
            {value: array[nextIndex++], done: false} :
            {value: undefined, done: true};
        }
    }
}
var it = makeIterator(['a', 'b']);
it.next(); // { value: "a", done: false }
it.next(); // { value: "b", done: false }
it.next(); // { value: undefined, done: true }
```

在 ES6 中，有些数据结构原生提供遍历器（比如数组），即不用任何处理，就可以被 for...of 循环遍历，有些就不行（比如对象）。原因在于，这些数据结构部署了 System.iterator 属性（详见下文）。凡是部署了 System.iterator 属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器。

默认的 Iterator 接口

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，即 for...of 循环。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。

ES6 规定，默认的 Iterator 接口部署在数据结构的 Symbol.iterator 属性，或者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）。也就是说，调用 Symbol.iterator 方法，就会得到当前数据结构的默认遍历器。Symbol.iterator 本身是一个表达式，返回 Symbol 对象的 iterator 属性，这是一个预定义好的、类型为 Symbol 的特殊值，所以要放在方括号内。

在 ES6 中，有三类数据结构原生具备 Iterator 接口：**数组、某些类似数组的对象**、Set 和 Map 结构。

```
let arr = ['a', 'b', 'c'];
let iter = arr[Symbol.iterator]();
iter.next(); // { value: 'a', done: false }
iter.next(); // { value: 'b', done: false }
iter.next(); // { value: 'c', done: false }
iter.next(); // { value: undefined, done: true }
```

上面代码中，变量 arr 是一个数组，原生就具有遍历器接口，部署在 arr 的 Symbol.iterator 属性上面。所以，调用这个属性就得到遍历器。

上面提到，原生就部署 iterator 接口的数据结构有三类，对于这三类数据结构，不用自己写遍历器，for...of 循环会自动遍历它们。除此之外，其他数据结构（主要是对象）的 Iterator 接口，都需要自己在 Symbol.iterator 属性上面部署，这样才会被 for...of 循环遍历。

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。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作 Map 结构使用，ES5 没有 Map 结构，而 ES6 原生提供了。

一个对象如果要有可被 for...of 循环调用的 Iterator 接口，就必须在 Symbol.iterator 的属性上部署遍历器方法（原型链上的对象具有该方法也可）。

```
//为对象添加 Iterator 接口的例子
let obj = {
    data: [ 'hello', 'world' ],[Symbol.iterator]() {
    const self = this;
    let index = 0;
```

```
    return {
        next() {
            if (index < self.data.length) {
                return {value: self.data[index++],done: false};
            } else {
                return { value: undefined, done: true };
            }
        }
    }
    }
};
var a = obj[Symbol.iterator]();
console.log(a.next());
console.log(a.next());
console.log(a.next());
```

对于类似数组的对象（存在数值键名和 length 属性），部署 Iterator 接口，有一个简便方法，就是 Symbol.iterator 方法直接引用数值的 Iterator 接口。

```
NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
```

有了遍历器接口，数据结构就可以用 for...of 循环遍历，也可以使用 while 循环遍历。

```
var iter = ITERABLE[Symbol.iterator]();
var result = iter.next();
while (!result.done) {
    var x = result.value;
    result = iter.next();
}
```

调用默认 iterator 接口的场合

（1）解构赋值

对数组和 Set 结构进行解构赋值时，会默认调用 iterator 接口。

```
let set = new Set().add('a').add('b').add('c');
let [x,y] = set;    // x='a'; y='b'

let [first, ...rest] = set;   // first='a'; rest=['b','c'];
```

（2）扩展运算符

扩展运算符（...）也会调用默认的 iterator 接口。

```
var str = 'hello';
[...str]; //  ['h','e','l','l','o']

let arr = ['b', 'c'];
['a', ...arr, 'd'];    // ['a', 'b', 'c', 'd']
```

（3）其他场合

以下场合也会用到默认的 iterator 接口。

- yield*
- Array.from()
- Map(), Set(), WeakMap(), WeakSet()
- Promise.all(), Promise.race()

原生具备 iterator 接口的数据结构

ES6 对数组提供 entries()、keys()和 values()三个方法，就是返回三个遍历器。

```
var arr = [1, 5, 7];
var arrEntries = arr.entries();
arrEntries.toString();   // "[object Array Iterator]"
arrEntries === arrEntries[Symbol.iterator]();    // true
```

字符串是一个类似数组的对象，也原生具有 Iterator 接口。

```
var someString = "hi";
typeof someString[Symbol.iterator];   // "function"

var iterator = someString[Symbol.iterator]();
iterator.next();  // { value: "h", done: false }
iterator.next();  // { value: "i", done: false }
iterator.next();  // { value: undefined, done: true }
```

可以覆盖原生的 Symbol.iterator 方法，达到修改遍历器行为的目的。

```
var str = new String("hi");
[...str]; // ["h", "i"]
str[Symbol.iterator] = function() {
    return {
        _first:true,
        next: function() {
            if (this._first) {
                this._first = false;
                return { value: "bye", done: false };
            } else {
                return { done: true };
            }
        }
    };
};
[...str] // ["bye"]
```

Iterator 接口与 Generator 函数

部署 Symbol.iterator 方法的最简单实现，还要使用下一节才讲的 Generator 函数。

```
var myIterable = {};
myIterable[Symbol.iterator] = function* () {
    yield 1;
    yield 2;
    yield 3;
}
[...myIterable]; // [1, 2, 3]

// 或者采用下面的简洁写法
let obj = {
    * [Symbol.iterator]() {
        yield 'hello';
        yield 'world';
    }
}
for (let x of obj) {
    console.log(x); //hello \n   world
}
```

return()，throw()

遍历器除了具有 next 方法（必备），还可以具有 return 方法和 throw 方法（可选）。for...of 循环如果提前退出（通常是因为出错，或者有 break 语句或 continue 语句），就会调用 return 方法。如果一个对象在完成遍历前，需要清理或释放资源，就可以部署 return 方法。throw 方法主要是配合 Generator 函数使用，一般的遍历器用不到这个方法。请见 Generator 函数讲解。

For...of 循环

ES6 借鉴 C++、Java、C#和 Python 语言，引入了 for...of 循环，作为遍历所有数据结构的统一的方法。一个数据结构只要部署了 Symbol.iterator 方法，就被视为具有 iterator 接口，就可以用 for...of 循环遍历它的成员。也就是说 for...of 循环内部调用的是数据结构的 Symbol.iterator 方法。

for...of 循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如 arguments 对象、DOM NodeList 对象）、后文的 Generator 对象，以及字符串。

数组原生具备 iterator 接口，for...of 循环本质上就是调用这个接口产生的遍历器，可以代替数组实例的 forEach 方法。JavaScript 原有的 for...in 循环，只能获得对象的键名，而 ES6 提供 for...of 循环，允许遍历获得键值。

```
const arr = ['red', 'green', 'blue'];
let iterator  = arr[Symbol.iterator]();
for(let v of arr) {
```

```
    console.log(v); // red green blue
}
//等同于
for(let v of iterator) {
    console.log(v); // red green blue
}
```

Set 和 Map 结构也原生具有 Iterator 接口，可以直接使用 for...of 循环。

```
var engines = Set(["Gecko", "Trident", "Webkit", "Webkit"]);
for (var e of engines) {
    console.log(e);
}

var maps = new Map();
maps.set("edition", 6);
maps.set("committee", "TC39");
maps.set("standard", "ECMA-262");
for (var [name, value] of maps) {
    console.log(name + ": " + value);
}
```

值得注意的地方有两个，首先，遍历的顺序是按照各个成员被添加进数据结构的顺序。其次，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。

for...of 遍历还可用于字符串（且能识别 32 位 UTF-16 字符）DOM NodeList 对象、arguments 等类似数组对象。

```
for (let p of document.querySelectorAll("p")) { }
for (let p of arguments) { }
for (let x of 'a\uD83D\uDC0A') {
    console.log(x);
    // 'a'
    // '\uD83D\uDC0A'
}

//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
let arrayLike = { length: 2, 0: 'a', 1: 'b' };
for (let x of arrayLike) {
    console.log(x);// 报错
}
for (let x of Array.from(arrayLike)) {
    console.log(x);// 正确
}
```

对象不能直接使用 for...of，会报错，必须部署了 iterator 接口后才能使用。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，或者将数组的 Symbol.iterator 属性，

直接赋值给其他对象的 Symbol.iterator 属性，或者使用 Generator 函数将对象重新包装一下。

```
var es6 = { edition: 6, committee: "TC39", standard: "ECMA-262"};
for (e of es6) {
    console.log(e);// TypeError: es6 is not iterable
}
for (var key of Object.keys(someObject)) {
    console.log(key + ": " + someObject[key]);
}
jQuery.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];//给 jquery 赋值 iterator
//Generator 函数包装
function* entries(obj) {
    for (let key of Object.keys(obj)) {
        yield [key, obj[key]];
    }
}
for (let [key, value] of entries(obj)) {
    console.log(key, "->", value);
}
```

总结：for...of 提供了遍历数据结构统一接口，不同于 forEach 方法，它可以与 break、continue 和 return 配合使用，它有 for...in 一样的简洁，但却没有 for...in 那些的缺点。for...in 的缺点有会遍历原型链上的键，任意顺序遍历键名，和遍历数组时以字符串表示键名，不适合遍历数组。

第九讲　function 函数的扩展

函数参数的默认值

```
function log(x, y) {
    y = y || 'World';//ES5 的方法
    y = arguments[1] !== (void 0) ? arguments[1] : 'World'; //traceur 转译方法
    y = arguments.length <= 1 || arguments[1] === undefined ? 'World' : arguments[1]; //babel 转译方法
    console.log(x, y);
}
function log(x, y = 'World') {
    console.log(x, y);//ES6 的方法
}

fetch(url, { method = 'GET' } = {}){
    //如果不含第二个参数，则默认值为一个空对象；如果包含第二个参数，则它的 method 属性默认值为 GET
    console.log(method);
}
```

可见如果传入 undefined 或不传，将触发该参数的默认值，null 和 false 和空值等都没有这个效果。指定了默认值以后，函数的 length 属性，将返回没有指定默认值的参数个数。也就是说，指定了默认值后，

length 属性将失真。

```
(function(a){}).length; // 1
(function(a = 5){}).length; // 0
(function(a, b, c = 5){}).length; // 2
```

函数参数的默认值具有作用域，且参数在函数体内不能再使用 let 或 const 再定义。参数默认值事以与解构赋值联合起来使用。

```
function foo({x, y = 5}) {
    console.log(x, y);
}
foo({}); // undefined, 5
foo({x: 1}); // 1, 5
foo({x: 1, y: 2}); // 1, 2
```

rest 参数

ES6 引入 rest 参数（...变量名），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。

```
// arguments 变量的写法
const sortNumbers = () =>   Array.prototype.slice.call(arguments).sort();

// rest 参数的写法
const sortNumbers = (...numbers) => numbers.sort();
```

剩余参数和 arguments 对象之间的区别主要有三个：

- 剩余参数只包含那些没有对应形参的实参，而 arguments 对象包含了传给函数的所有实参。
- argument 对象不是一个真实的数组，而剩余参数是真实的 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，比如 sort、 map、forEach、pop 等。
- arguments 对象对象还有一些附加的属性（比如 callee 属性）。

```
//注意，rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。
function f(a, ...b, c) {
    // ... 报错
}

//函数的 length 属性，不包括 rest 参数，类似默认参数。
(function(a) {}).length;  // 1
(function(...a) {}).length;  // 0
(function(a, ...b) {}).length;  // 1
```

扩展运算符

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参

数序列。该运算符主要用于函数调用。

```
//参数使用扩展运算符的例子
Array.push(...items);//加入数组
var args = [0, 1];
f(-1, ...args, 2, ...[3]);//函数调用 可以避免使用 apply
new Date(...[2015, 1, 1]);//传参
Math.max(...[14, 3, 77]);//传参
const [first, ...rest] = [1, 2, 3, 4, 5];//解构赋值  但只能放在参数的最后一位，否则会报错。
//扩展运算符还可以将一个数值或任何类似数组的对象扩展成数组
[...5];  // [0, 1, 2, 3, 4, 5]
[..."hello"];  // [ "h", "e", "l", "l", "o" ]
[...document.querySelectorAll('div')];
[...map.keys()];
//还有 Generator 函数等等
```

箭头函数

ES6 允许使用“箭头”（=>）定义函数。

```
var f = function(v) {return v; };//传统函数
var f = (v) => v; //ES6 同上函数
var sum = function(num1, num2) { return num1 + num2;}
var sum = (num1, num2) => num1 + num2;// 等同于上面函数

var getTempItem = id => ({ id: id, name: "Temp" });//如果返回的是对象需要外面加上括号
var full = ({ first, last }) => first + ' ' + last;//箭头函数可以与变量解构结合使用
const numbers = (...nums) => nums;
numbers(1, 2, 3, 4, 5); // [1,2,3,4,5]
```

箭头函数有几个使用注意点：

- 函数体内的 this 对象，绑定定义时所在的对象，而不是使用时所在的对象。
- 不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误。
- 不可以使用 arguments 对象，该对象在函数体内不存在。

上面三点中，第一点尤其值得注意。this 对象的指向是可变的，但是在箭头函数中，它是固定的。下面的代码是一个例子，将 this 对象绑定定义时所在的对象。

```
var handler = {
    id: "123456",
    init: function() {
        document.addEventListener("click",
            event => this.doSomething(event.type), false);//注意这里的 this
    },
    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
```

```
    }
};
```

由于 this 在箭头函数中被绑定，所以不能用 call()、apply()、bind()这些方法去改变 this 的指向。

长期以来，JavaScript 语言的 this 对象一直是一个令人头痛的问题，在对象方法中使用 this，必须非常小心。箭头函数绑定 this，很大程度上解决了这个困扰。

函数绑定

箭头函数可以绑定 this 对象，大大减少了显式绑定 this 对象的写法（call、apply、bind）。但是，箭头函数并不适用于所有场合，所以 ES7 提出了“**函数绑定**”（function bind）运算符，用来取代 call、apply、bind 调用。虽然该语法还是 ES7 的一个提案，但是 Babel 转码器已经支持。

函数绑定运算符是并排的两个双引号（::），双引号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象，作为上下文环境（即 this 对象），绑定到右边的函数上面。

```
let log = ::console.log;  // 简写  原是 let log = console::console.log;
var log = console.log.bind(console);// 同上

foo::bar;
bar.call(foo);// 同上

foo::bar(...arguments);
bar.apply(foo, arguments);// 同上
```

尾调用优化

尾调用（Tail Call）是函数式编程的一个重要概念，就是指某个函数的最后一步是调用另一个函数。

```
function f(x){
    return g(x) + 1;//不属于尾调用
}
function f(x){
    let y = g(x);
    return y;   //不属于尾调用
}
function f(x){
    return g(x);//属于尾调用
}
```

尾调用之所以特殊就是它在函数体的最后一步位置。

我们知道，函数调用会在内存形成一个“**调用记录**”，又称“调用帧”（call frame），保存调用位置和内部变量等信息。如果在函数 A 的内部调用函数 B，那么在 A 的调用帧上方，还会形成一个 B 的调用帧。等到 B 运行结束，将结果返回到 A，B 的调用帧才会消失。如果函数 B 内部还调用函数 C，那就还有一个 C 的

调用帧，以此类推。所有的调用帧，就形成一个“调用栈”（call stack）。

尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。

```
function f() {
    let m = 1;
    let n = 2;
    return g(m + n);
}
f();
// 等同于
function f() {
    return g(3);
}
f();
// 等同于
g(3);
```

上面代码中，如果函数 g 不是尾调用，函数 f 就需要保存内部变量 m 和 n 的值、g 的调用位置等信息。但由于调用 g 之后，函数 f 就结束了，所以执行到最后一步，完全可以删除 f(x) 的调用帧，只保留 g(3) 的调用帧。

这就叫做“**尾调用优化**”（Tail call optimization），即只保留内层函数的调用帧。如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”的意义。

尾递归

函数调用自身，称为**递归**。如果尾调用自身，就称为**尾递归**。

递归非常耗费内存，因为需要同时保存成千上万个调用帧，很容易发生“**栈溢出**”错误（stack overflow）但对于尾递归来说，由于只存在一个调用帧，所以永远不会发生“栈溢出”错误。“尾调用优化”对递归操作意义重大，所以一些函数式编程语言将其写入了语言规格。ES6 也是如此，第一次明确规定，所有 ECMAScript 的实现，都必须部署“尾调用优化”。这就是说，在 ES6 中，只要使用尾递归，就不会发生栈溢出，相对节省内存。

```
//阶乘函数，计算 n 的阶乘，最多需要保存 n 个调用记录，复杂度 O(n)
function factorial(n) {
    if (n === 1) return 1;
    return n * factorial(n - 1);
}
factorial(5); // 120
```

```
//如果改写成尾递归，只保留一个调用记录，复杂度 O(1)
function factorial(n, total) {
    if (n === 1) return total;
    return factorial(n - 1, n * total);
}
factorial(5,1); // 120
```

递归函数改写

尾递归的实现，往往需要改写递归函数，确保最后一步只调用自身。做到这一点的方法，就是把所有用到的内部变量改写成函数的参数。

```
//如上阶乘函数改写成如下
function tailFactorial(n, total) {
    if (n === 1) return total;
    return tailFactorial(n - 1, n * total);
}
function factorial(n) {
    return tailFactorial(n, 1);
}
factorial(5); // 120
```

函数式编程有一个概念，叫做**柯里化**（currying），意思是将多参数的函数转换成单参数的形式。这里也可以使用柯里化。

```
function currying(fn, n) {
    return function (m) {
        return fn.call(this, m, n);
    };
}
function tailFactorial(n, total) {
    if (n === 1) return total;
    return tailFactorial(n - 1, n * total);
}
const factorial = currying(tailFactorial, 1);

factorial(5); // 120
```

总结一下，递归本质上是一种循环操作。纯粹的函数式编程语言没有循环操作命令，所有的循环都用递归实现，这就是为什么尾递归对这些语言极其重要。对于其他支持“尾调用优化”的语言（比如 Lua，ES6），只需要知道循环可以用递归代替，而一旦使用递归，就最好使用尾递归。

第十讲　Class 和 Module

Class 基本语法

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 class 关键字，可以定义类。

```
//定义类
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return '('+this.x+', '+this.y+')';
    }
}
```

上面代码定义了一个“类”，可以看到里面有一个 constructor 方法，这就是构造方法，而 this 关键字则代表实例对象。

Constructor 方法

constructor 方法是类的默认方法，通过 new 命令生成对象实例时，自动调用该方法。

```
var point = new Point(2, 3);//实例对象

class Point {} //name 属性
Point.name // "Point"

//class 表达式  与函数一样，Class 也可以使用表达式的形式定义。
const MyClass = class Me {
    getClassName() {
        return Me.name;
    }
};
```

Class 的继承

Class 之间可以通过 extends 关键字实现继承。子类会继承父类的属性和方法。

```
class Point {
    constructor(x, y) {
        this.x = x;
```

```
        this.y = y;
    }
}
class ColorPoint extends Point {
    constructor(x, y, color) {
        this.color = color; // ReferenceError
        super(x, y);
        this.color = color; // 正确
    }
}
```

上面代码中，子类的 constructor 方法没有调用 super 之前，就使用 this 关键字，结果报错，而放在 super 方法之后就是正确的。

注意：ColorPoint 继承了父类 Point，但是它的构造函数必须调用 super 方法。下面是生成子类实例的代码。

```
let cp = new ColorPoint(25, 8, 'green');
cp instanceof ColorPoint // true
cp instanceof Point // true
```

class 的取值函数（getter）和存值函数（setter）

在 Class 内部可以使用 get 和 set 关键字，对某个属性设置存值函数和取值函数。

```
class MyClass {
    get prop() {
        return 'getter';
    }
    set prop(value) {
        document.write('setter: '+ value);
    }
}
let inst = new MyClass();
inst.prop = 123;  // setter: 123
inst.prop;   // 'getter'
```

上面代码中，prop 属性有对应的存值函数和取值函数，因此赋值和读取行为都被自定义了。

Class 的 Generator 方法

如果某个方法之前加上星号（*），就表示该方法是一个 Generator 函数。

```
class Foo {
    constructor(...args) {
        this.args = args;
    }
```

```
    * [Symbol.iterator]() {
        for (let arg of this.args) {
            yield arg;
        }
    }
}
for (let x of new Foo('hello', 'world')) {
    document.write(x);
}
// hello
// world
```

上面代码中，Foo 类的 Symbol.iterator 方法前有一个星号，表示该方法是一个 Generator 函数。Symbol.iterator 方法返回一个 Foo 类的默认遍历器，for...of 循环会自动调用这个遍历器。

Class 的静态方法

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上 static 关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“**静态方法**”。

```
class Foo {
    static classMethod() {
        return 'hello';
    }
}
Foo.classMethod() // 'hello'
var foo = new Foo();
foo.classMethod()
// TypeError: undefined is not a function
```

上面代码中，Foo 类的 classMethod 方法前有 static 关键字，表明该方法是一个静态方法，可以直接在 Foo 类上调用（Foo.classMethod()），而不是在 Foo 类的实例上调用。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。

父类的静态方法，可以被子类继承。

```
class Foo {
    static classMethod() {
        return 'hello';
    }
}
class Bar extends Foo {}
Bar.classMethod(); // 'hello'
```

上面代码中，父类 Foo 有一个静态方法，子类 Bar 可以调用这个方法。

静态方法也是可以从 super 对象上调用的。

```
class Foo {
    static classMethod() {
        return 'hello';
    }
}
class Bar extends Foo {
    static classMethod() {
        return super.classMethod() + ', too';
    }
}
Bar.classMethod();
```

new.target 属性

new 是从构造函数生成实例的命令。ES6 为 new 命令引入了一个 new.target 属性，（在构造函数中）返回 new 命令作用于的那个构造函数。如果构造函数不是通过 new 命令调用的，new.target 会返回 undefined，因此这个属性可以用来确定构造函数是怎么调用的。

```
function Person(name) {
    if (new.target !== undefined) {
        this.name = name;
    } else {
        throw new Error('必须使用 new 生成实例');
    }
}
// 另一种写法
function Person(name) {
    if (new.target === Person) {
        this.name = name;
    } else {
        throw new Error('必须使用 new 生成实例');
    }
}
var person = new Person('张三'); // 正确
var notAPerson = Person.call(person, '张三');  // 报错
```

上面代码确保构造函数只能通过 new 命令调用。

- Class 内部调用 new.target，返回当前 Class
- 子类继承父类时 new.target 会返回子类

修饰器-类的修饰

修饰器（Decorator）是一个表达式，用来修改类的行为。这是 ES7 的一个提案，目前 Babel 转码器已

经支持。

修饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，修饰器能在编译阶段运行代码。

```
function testable(target) {
    target.isTestable = true;
}
@testable
class MyTestableClass {}
console.log(MyTestableClass.isTestable) // true
```

上面代码中，@testable 就是一个修饰器。它修改了 MyTestableClass 这个类的行为，为它加上了静态属性 isTestable。

基本上，修饰器的行为就是下面这样。

```
@decorator
class A {}

// 等同于
class A {}
A = decorator(A) || A;
```

也就是说，修饰器本质上就是能在编译时执行的函数。

修饰器函数可以接受三个参数，依次是**目标函数、属性名**和**该属性的描述对象**。后两个参数可省略。上面代码中，testable 函数的参数 target，就是所要修饰的对象。如果希望修饰器的行为能够根据目标对象的不同而不同，就要在外面再封装一层函数。

```
function testable(isTestable) {
    return function(target) {
        target.isTestable = isTestable;
    }
}

@testable(true) class MyTestableClass () {}
document.write(MyTestableClass.isTestable) // true

@testable(false) class MyClass () {}
document.write(MyClass.isTestable) // false
```

如果想要为类的实例添加方法，可以在修饰器函数中，为目标类的 prototype 属性添加方法。

Export 命令

模块功能主要由两个命令构成：export 和 import。

- export 命令用于用户自定义模块，规定对外接口；

- import 命令用于输入其他模块提供的功能，同时创造命名空间（namespace），防止函数名冲突。

ES6 允许将独立的 JS 文件作为模块，允许一个 JavaScript 脚本文件调用另一个脚本文件。

现有 profile.js 文件，保存了用户信息。ES6 将其视为一个模块，里面用 export 命令对外部输出了三个变量。

```
// profile.js
var firstName = 'Michael';
var lastName = 'Jackson';
var year = 1958;

export {firstName, lastName, year};
```

Import 命令

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）。

```
// main.js
import {firstName, lastName, year} from './profile';

function sfirsetHeader(element) {
    element.textContent = firstName + ' ' + lastName;
}
```

上面代码属于另一个文件 main.js，import 命令就用于加载 profile.js 文件，并从中输入变量。import 命令接受一个对象（用大括号表示），里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块（profile.js）对外接口的名称相同。

如果想为输入的变量重新取一个名字，import 语句中要使用 as 关键字，将输入的变量重命名。

```
import { lastName as surname } from './profile';
```

ES6 支持多重加载，即所加载的模块中又加载其他模块。

模块的整体输入

export 命令除了输出变量，还可以输出方法或类（class）。下面是一个 circle.js 文件，它输出两个方法 area 和 circumference。

```
// circle.js
export function area(radius) {
    return Math.PI * radius * radius;
}
export function circumference(radius) {
    return 2 * Math.PI * radius;
```

```
}

/*然后，main.js 输入 circlek.js 模块*/

// main.js
import { area, circumference } from 'circle';
document.write("圆面积：" + area(4));
document.write("圆周长：" + circumference(14));

/*上面写法是逐一指定要输入的方法。另一种写法是整体输入*/

import * as circle from 'circle';
document.write("圆面积：" + circle.area(4));
document.write("圆周长：" + circle.circumference(14));
```

module 命令

module 命令可以取代 import 语句，达到整体输入模块的作用。

```
// main.js
module circle from 'circle';
document.write("圆面积：" + circle.area(4));
document.write("圆周长：" + circle.circumference(14));
```

module 命令后面跟一个变量，表示输入的模块定义在该变量上。

Export default 命令

为加载模块指定默认输出，使用 export default 命令。

```
// export-default.js
export default function () {
  document.write('foo');
}
```

上面代码是一个模块文件 export-default.js，它的默认输出是一个函数。其他模块加载该模块时，import 命令可以为该匿名函数指定任意名字。

```
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

上面代码的 import 命令，可以用任意名称指向 export-default.js 输出的方法。需要注意的是，这时 import 命令后面，不使用大括号。

模块的继承

模块之间也可以继承。假设有一个 circleplus 模块，继承了 circle 模块。

```
// circleplus.js
export * from 'circle';
export var e = 2.71828182846;
export default function(x) {
    return Math.exp(x);
}
```

上面代码中的“export *”，表示输出 circle 模块的所有属性和方法，export default 命令定义模块的默认方法。

这时，也可以将 circle 的属性或方法，改名后再输出。

```
// circleplus.js
export { area as circleArea } from 'circle';
```

上面代码表示，只输出 circle 模块的 area 方法，且将其改名为 circleArea。

加载上面模块的写法如下。

```
// main.js

module math from "circleplus";
import exp from "circleplus";
document.write(exp(math.pi));
```

上面代码中的"import exp"表示，将 circleplus 模块的默认方法加载为 exp 方法。

第十一讲　Generator 函数

简介

所谓 Generator，有多种理解角度。首先，可以把它理解成一个函数的内部状态的遍历器，每调用一次，函数的内部状态发生一次改变（可以理解成发生某些事件）。ES6 引入 Generator 函数，作用就是可以完全控制函数的内部状态的变化，依次遍历这些状态。

在形式上，Generator 是一个普通函数，但是有两个特征。一是，function 命令与函数名之间有一个星号；二是函数体内部使用 yield 语句，定义遍历器的每个成员，即不同的内部状态（yield 语句在英语里的意思就是“产出”）。

```
function* helloWorldGenerator() {
    yield 'hello';
    yield 'world';
    return 'ending';
```

```
}
var hw = helloWorldGenerator();
hw.next();   // { value: 'hello', done: false }
hw.next();   // { value: 'world', done: false }
hw.next();   // { value: 'ending', done: true }
hw.next();   // { value: undefined, done: true }
```

上面代码定义了一个 Generator 函数 helloWorldGenerator，它的遍历器有两个成员“hello”和“world”。调用这个函数，就会得到遍历器。

当调用 Generator 函数的时候，该函数并不执行，而是返回一个遍历器（可以理解成暂停执行）。以后，每次调用这个遍历器的 next 方法，就从函数体的头部或者上一次停下来的地方开始执行（可以理解成恢复执行），直到遇到下一个 yield 语句为止。也就是说，next 方法就是在遍历 yield 语句定义的内部状态。

Generator 函数使用 iterator 接口，每次调用 next 方法的返回值，就是一个标准的 iterator 返回值：有着 value 和 done 两个属性的对象。其中，value 是 yield 语句后面那个表达式的值，done 是一个布尔值，表示是否遍历结束。

上面说过，任意一个对象的 Symbol.iterator 属性，等于该对象的遍历器函数，即调用该函数会返回该对象的一个遍历器。遍历器本身也是一个对象，它的 Symbol.iterator 属性执行后就返回自身。

```
function* gen(){
    // some code
}
var g = gen();
g[Symbol.iterator]() === g ;    // true
```

由于 Generator 函数返回的遍历器，只有调用 next 方法才会遍历下一个成员，所以其实提供了一种可以暂停执行的函数。yield 语句就是暂停标志，next 方法遇到 yield，就会暂停执行后面的操作，并将紧跟在 yield 后面的那个表达式的值，作为返回对象的 value 属性的值。当下一次调用 next 方法时，再继续往下执行，直到遇到下一个 yield 语句。如果没有再遇到新的 yield 语句，就一直运行到函数结束，将 return 语句后面的表达式的值，作为 value 属性的值，如果该函数没有 return 语句，则 value 属性的值为 undefined。另一方面，由于 yield 后面的表达式，直到调用 next 方法时才会执行，因此等于为 JavaScript 提供了手动的“**惰性求值**”（Lazy Evaluation）的语法功能。

注意：yield 语句不能用在普通函数中，否则会报错！

Next 方法的参数

yield 语句本身没有返回值，或者说总是返回 undefined。next 方法可以带一个参数，该参数就会被当作上一个 yield 语句的返回值。

```
function* f() {
    for(var i=0; true; i++) {
```

```
        var reset = yield i;
        if(reset) { i = -1; }
    }
}
var g = f();
g.next(); // { value: 0, done: false }
g.next(); // { value: 1, done: false }
g.next(true); // { value: 0, done: false }
```

上面代码先定义了一个可以无限运行的 Generator 函数 f，如果 next 方法没有参数，每次运行到 yield 语句，变量 reset 的值总是 undefined。当 next 方法带一个参数 true 时，当前的变量 reset 就被重置为这个参数（即 true），因此 i 会等于-1，下一轮循环就会从 -1 开始递增。

这个功能有很重要的语法意义。Generator 函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过 next 方法的参数，就有办法在 Generator 函数开始运行之后，继续向函数体内部注入值。也就是说，可以在 Generator 函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。

For...of 循环

for...of 循环可以自动遍历 Generator 函数，且此时不再需要调用 next 方法。

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
    return 6;
}
for (let v of foo()) {
    console.log(v);// 1 2 3 4 5
}
```

上面代码使用 for...of 循环，依次显示 5 个 yield 语句的值。这里需要注意，一旦 next 方法的返回对象的 done 属性为 true，for...of 循环就会中止，且不包含该返回对象，所以上面代码的 return 语句返回的 6，不包括在 for...of 循环之中。

下面是一个利用 generator 函数和 for...of 循环，实现**斐波那契数列**的例子。

```
function* fibonacci() {
    let [prev, curr] = [0, 1];
    for (;;) {
        [prev, curr] = [curr, prev + curr];
        yield curr;
    }
}
```

```
for (let n of fibonacci()) {
    if (n > 1000) break;
    console.log(n);
}
```

throw 方法

Generator 函数还有一个特点，它可以在函数体外抛出错误，然后在函数体内捕获。

```
var g = function* () {
    while (true) {
        try {
            yield;
        } catch (e) {
            if (e != 'a') throw e;
            console.log('内部捕获', e);
        }
    }
}
var i = g();
i.next();
try {
    i.throw('a');
    i.throw('b');
} catch (e) {
    console.log('外部捕获', e);
}
// 内部捕获 a
// 外部捕获 b
```

上面代码中，遍历器 i 连续抛出两个错误。第一个错误被 Generator 函数体内的 catch 捕获，然后 Generator 函数执行完成，于是第二个错误被函数体外的 catch 捕获。

一旦 Generator 执行过程中抛出错误，就不会再执行下去了。如果此后还调用 next 方法，将返回一个 value 属性等于 undefined、done 属性等于 true 的对象，即 JavaScript 引擎认为这个 Generator 已经运行结束了。

Generator.prototype.return()

Generator 函数返回的遍历器对象，还有一个 return 方法，可以返回给定的值，并且终结遍历 Generator 函数。

```
function* gen(){
    yield 1;
    yield 2;
```

```
    yield 3;
}
var g = gen();
g.next();        // { value: 1, done: false }
g.return("foo"); // { value: "foo", done: true }
g.next();        // { value: undefined, done: true }
```

上面代码中，遍历器对象 g 调用 return 方法后，返回值的 value 属性就是 return 方法的参数 foo。并且，Generator 函数的遍历就终止了，返回值的 done 属性为 true，以后再调用 next 方法，done 属性总是返回 true。

如果 return 方法调用时，不提供参数，则返回值的 vaule 属性为 undefined。

如果 Generator 函数内部有 try...finally 代码块，那么 return 方法会推迟到 finally 代码块执行完再执行。

```
function* numbers () {
    yield 1;
    try {
        yield 2;
        yield 3;
    } finally {
        yield 4;
        yield 5;
    }
    yield 6;
}
var g = numbers();
g.next();    // { done: false, value: 1 }
g.next();    // { done: false, value: 2 }
g.return(7); // { done: false, value: 4 }
g.next();    // { done: false, value: 5 }
g.next();    // { done: true, value: 7 }
```

上面代码中，调用 return 方法后，就开始执行 finally 代码块，然后等到 finally 代码块执行完，再执行 return 方法。

Yield*语句

如果 yield 命令后面跟的是一个遍历器，需要在 yield 命令后面加上星号，表明它返回的是一个遍历器。这被称为 yield*语句。

```
let delegatedIterator = (function* () {
    yield 'Hello!';
    yield 'Bye!';
}());
let delegatingIterator = (function* () {
```

```
    yield 'Greetings!';
    yield* delegatedIterator;
    yield 'Ok, bye.';
}());
for(let value of delegatingIterator) {
    console.log(value);
}
// "Greetings!
// "Hello!"
// "Bye!"
// "Ok, bye."
```

作为对象属性的 Generator 函数

如果一个对象的属性是 Generator 函数，可以简写成下面的形式。

```
let obj = {
    * myGeneratorMethod() {}
}
//等同于
let obj = {
    myGeneratorMethod: function* () {}
}
```

Generator 函数推导

ES7 在数组推导的基础上，提出了 Generator 函数推导（Generator comprehension）。

```
let generator = function* () {
    for (let i = 0; i < 6; i++) {
        yield i;
    }
}
let squared = ( for (n of generator()) n * n );
// 等同于
let squared = Array.from(generator()).map(n => n * n);
console.log(...squared);    // 0 1 4 9 16 25
```

“**推导**”这种语法结构，在 ES6 只能用于数组，ES7 将其推广到了 Generator 函数。for...of 循环会自动调用遍历器的 next 方法，将返回值的 value 属性作为数组的一个成员。

Generator 函数推导是对数组结构的一种模拟，它的最大优点是惰性求值，即直到真正用到时才会求值，这样可以保证效率。请看下面的例子。

```
let bigArray = new Array(100000);
for (let i = 0; i < 100000; i++) {
    bigArray[i] = i;
```

```
}

let first = bigArray.map(n => n * n)[0];
console.log(first);
```

上面例子遍历一个大数组，但是在真正遍历之前，这个数组已经生成了，占用了系统资源。如果改用 Generator 函数推导，就能避免这一点。下面代码只在用到时，才会生成一个大数组。

```
let bigGenerator = function* () {
    for (let i = 0; i < 100000; i++) {
        yield i;
    }
}

let squared = ( for (n of bigGenerator()) n * n );
console.log(squared.next());
```

Generator 是实现状态机的最佳结构。比如，下面的 clock 函数就是一个状态机。

```
var ticking = true;
var clock = function() {
    if (ticking)
        console.log('Tick!');
    else
        console.log('Tock!');
    ticking = !ticking; //或装逼写法 console.log((ticking = !ticking) ? 'Tock!':'Tick!');
}
```

上面代码的 clock 函数一共有两种状态（Tick 和 Tock），每运行一次，就改变一次状态。这个函数如果用 Generator 实现，就是下面这样。

```
var clock = function*(_) {
    while (true) {
        yield _;
        console.log('Tick!');
        yield _;
        console.log('Tock!');
    }
};
```

上面的 Generator 实现与 ES5 实现对比，可以看到少了用来保存状态的外部变量 ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。Generator 之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。

应用

Generator 可以暂停函数执行，返回任意表达式的值。这种特点使得 Generator 有多种应用场景。

（1）异步操作的同步化表达

Generator 函数的暂停执行的效果，意味着可以把异步操作写在 yield 语句里面，等到调用 next 方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在 yield 语句下面，反正要等到调用 next 方法时再执行。所以，Generator 函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

```
//加载 UI 界面
function* loadUI() {
    showLoadingScreen();
    yield loadUIDataAsynchronously();
    hideLoadingScreen();
}
var loader = loadUI();
loader.next();// 加载 UI
loader.next();// 卸载 UI

//使用 yield 语句可以手动逐行读取文件
function* numbers() {
    let file = new FileReader("numbers.txt");
    try {
        while(!file.eof) {
            yield parseInt(file.readLine(), 10);
        }
    } finally {
        file.close();
    }
}
```

（2）控制流管理

如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});
```

Generator 函数可以进一步改善代码运行流程。

```
function* longRunningTask() {
    try {
        var value1 = yield step1();
        var value2 = yield step2(value1);
        var value3 = yield step3(value2);
        var value4 = yield step4(value3);
```

```
        // Do something with value4
    } catch (e) {
        // Handle any error from step1 through step4
    }
}
//然后，使用一个函数，按次序自动执行所有步骤。
scheduler(longRunningTask());
function scheduler(task) {
    setTimeout(function() {
        var taskObj = task.next(task.value);
        // 如果 Generator 函数未结束，就继续调用
        if (!taskObj.done) {
            task.value = taskObj.value;
            scheduler(task);
        }
    }, 0);
}
```

多个任务按顺序一个接一个执行时，yield 语句可以按顺序排列。多个任务需要并列执行时（比如只有 A 任务和 B 任务都执行完，才能执行 C 任务），可以采用数组的写法。

```
function* parallelDownloads() {
    let [text1,text2] = yield [  taskA(), taskB()   ];
    console.log(text1, text2);
}
```

上面代码中，yield 语句的参数是一个数组，成员就是两个任务 taskA 和 taskB，只有等这两个任务都完成了，才会接着执行下面的语句。

（3）部署 iterator 接口

利用 Generator 函数，可以在任意对象上部署 iterator 接口。

```
function* iterEntries(obj) {
    let keys = Object.keys(obj);
    for (let i = 0; i < keys.length; i ++) {
        let key = keys[i];
        yield [key, obj[key]];
    }
}
let myObj = { foo: 3, bar: 7 };
for (let [key, value] of iterEntries(myObj)) {
    console.log(key, value);
}
// foo 3
// bar 7
```

上述代码中，myObj 是一个普通对象，通过 iterEntries 函数，就有了 iterator 接口。也就是说，可以在任意对象上部署 next 方法。

下面是一个对数组部署 Iterator 接口的例子，尽管数组原生具有这个接口。

```
function* makeSimpleGenerator(array){
    var nextIndex = 0;
    while(nextIndex < array.length){
        yield array[nextIndex++];
    }
}
var gen = makeSimpleGenerator(['yo', 'ya']);
gen.next().value; // 'yo'
gen.next().value; // 'ya'
gen.next().done;  // true
```

（4）作为数据结构

Generator 可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为 Generator 函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。

```
function *doStuff() {
    yield fs.readFile.bind(null, 'hello.txt');
    yield fs.readFile.bind(null, 'world.txt');
    yield fs.readFile.bind(null, 'and-such.txt');
}
//上面代码就是依次返回三个函数，但是由于使用了 Generator 函数，导致可以像处理数组那样，处理这三个返回的函数
for (task of doStuff()) {
    // task 是一个函数，可以像回调函数那样使用它
}
//实际上，如果用 ES5 表达，完全可以用数组模拟 Generator 的这种用法
function doStuff() {
    return [
        fs.readFile.bind(null, 'hello.txt'),
        fs.readFile.bind(null, 'world.txt'),
        fs.readFile.bind(null, 'and-such.txt')
    ];
}
//上面的函数，可以用一模一样的 for...of 循环处理！两相一比较，就不难看出 Generator 使得数据或者操作，具备了类似数组的接口
```

第十二讲　Promise 对象

基本用法

ES6 原生提供了 Promise 对象。所谓 Promise 对象，就是代表了某个未来才会知道结果的事件（通常是一个异步操作），并且这个事件提供统一的 API，可供进一步处理。

有了 Promise 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise 对象提供的接口，使得控制异步操作更加容易。

ES6 的 Promise 对象是一个构造函数，用来生成 Promise 实例。一个 Promise 处于以下四种状态之一：

- Pending　　初始状态，非 fulfilled 或 rejected；
- Resolved　　成功的操作，或称 Fulfilled；
- Rejected　　失败的操作；
- Settled　　Promise 已被 fulfilled 或 rejected，且不是 pending。

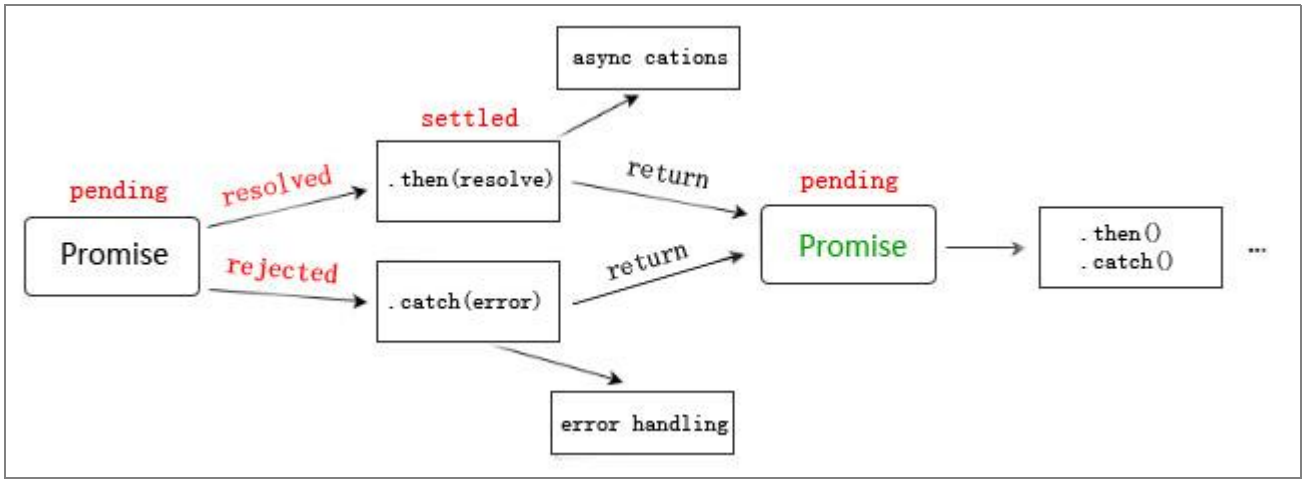
```
var promise = new Promise(function(resolve, reject) {
    if (/* 异步操作成功 */){
        resolve(value);
    } else {
        reject(error);
    }
});

promise.then(function(value) {
    // success
}, function(value) {
    // failure
});

getJSON("/posts.json").then(function(json) {
    return json.post;
});
```

上面代码中 Promise 构造函数接受一个函数作为参数，该函数的两个参数分别是 resolve 方法和 reject 方法。如果异步操作成功，则用 resolve 方法将 Promise 对象的状态，从“未完成”变为“成功”（即从 pending 变为 resolved）；如果异步操作失败，则用 reject 方法将 Promise 对象的状态，从“未完成”变为“失败”

（即从 pending 变为 rejected）。



Promise 实例生成以后，可以用 then 方法分别指定 resolve 方法和 reject 方法的回调函数。

Promise 缺点：

- 首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消；
- 其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部；
- 第三，当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

Promise.prototype.then()

Promise 实例具有 then 方法，也就是说，then 方法是定义在原型对象 Promise.prototype 上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过 then 方法的第一个参数是 Resolved 状态的回调函数，第二个参数（可选）是 Rejected 状态的回调函数。

```
getJSON("/post/1.json").then(
    post => getJSON(post.commentURL)
).then(
    comments => console.log("Resolved: ", comments),
    err => console.log("Rejected: ", err)
);
```

then 方法返回的是一个新的 Promise 实例（注意，不是原来那个 Promise 实例）。因此可以采用链式写法，即 then 方法后面再调用另一个 then 方法。

上面的代码使用 then 方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

采用链式的 then，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个 Promise 对象（即有异步操作），这时后一个回调函数，就会等待该 Promise 对象的状态发生变化，才会被调用。

Promise.prototype.catch()

Promise.prototype.catch 方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数。

```
getJSON("/posts.json").then(function(posts) {
    // ...
}).catch(function(error) {
    // 处理前一个回调函数运行时发生的错误
    document.write('发生错误！', error);
});
```

getJSON 方法返回一个 Promise 对象，如果该对象状态变为 Resolved，则会调用 then 方法指定的回调函数；如果异步操作抛出错误，状态就会变为 Rejected，就会调用 catch 方法指定的回调函数，处理这个错误。

```
var promise = new Promise(function(resolve, reject) {
    throw new Error('test')
});
promise.catch(function(error) { document.write(error) });
// Error: test
```

上面代码中，Promise 抛出一个错误，就被 catch 方法指定的回调函数捕获。

Promise 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个 catch 语句捕获。

```
getJSON("/post/1.json").then(function(post) {
    return getJSON(post.commentURL);
}).then(function(comments) {
    // some code
}).catch(function(error) {
    // 处理前面三个 Promise 产生的错误
});
```

上面代码中，一共有三个 Promise 对象：一个由 getJSON 产生，两个由 then 产生。它们之中任何一个抛出的错误，都会被最后一个 catch 捕获。

Promise.all()

Promise.all 方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

```
var p = Promise.all([p1,p2,p3]);
```

上面代码中，Promise.all 方法接受一个数组作为参数，p1、p2、p3 都是 Promise 对象的实例(Promise.all 方法的参数不一定是数组，但是必须具有 iterator 接口，且返回的每个成员都是 Promise 实例）。p 的状态由 p1、p2、p3 决定，分成两种情况：

- 只有 p1、p2、p3 的状态都变成 fulfilled，p 的状态才会变成 fulfilled，此时 p1、p2、p3 的返回值组成一个数组，传递给 p 的回调函数。
- 只要 p1、p2、p3 之中有一个被 rejected，p 的状态就变成 rejected，此时第一个被 reject 的实例的返回值，会传递给 p 的回调函数。

下面是一个具体的例子。

```
// 生成一个 Promise 对象的数组
var promises = [2, 3, 5, 7, 11, 13].map(function(id){
    return getJSON("/post/" + id + ".json");
});
Promise.all(promises).then(function(posts) {
    // ...
}).catch(function(reason){
    // ...
});
```

Promise.race()

Promise.race 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

```
var p = Promise.race([p1,p2,p3]);
```

上面代码中，只要 p1、p2、p3 之中有一个实例率先改变状态，p 的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给 p 的回调函数。

如果 Promise.all 方法和 Promise.race 方法的参数，不是 Promise 实例，就会先调用下面讲到的 Promise.resolve 方法，将参数转为 Promise 实例，再进一步处理。

Promise.resolve()

有时需要将现有对象转为 Promise 对象，Promise.resolve 方法就起到这个作用。

如果 Promise.resolve 方法的参数，不是具有 then 方法的对象（又称 thenable 对象），则返回一个新的 Promise 对象，且它的状态为 Resolved。

```
var p = Promise.resolve('Hello');
p.then(function (s){
    document.write(s);// Hello
});
```

由于字符串 Hello 不属于异步操作（判断方法是它不是具有 then 方法的对象），返回 Promise 实例的状态从一生成就是 Resolved，所以回调函数会立即执行。Promise.resolve 方法的参数是可选的。

Promise.reject()

Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为 rejected。Promise.reject 方法的参数 reason，会被传递给实例的回调函数。

```
var p = Promise.reject('出错了');
p.then(null, function (s){
    document.write(s)
});
```

上面代码生成一个 Promise 对象的实例 p，状态为 rejected，回调函数会立即执行。

Generator 函数与 Promise 的结合

使用 Generator 函数管理流程，遇到异步操作的时候，通常返回一个 Promise 对象。

```
function getFoo () {
    return new Promise(function (resolve, reject){
        resolve('foo');
    });
}
var g = function* () {
    try {
        var foo = yield getFoo();
        document.write(foo);
    } catch (e) {
        document.write(e);
    }
};
function run (generator) {
    var it = generator();
    function go(result) {
        if (result.done) return result.value;
        return result.value.then(function (value) {
            return go(it.next(value));
        }, function (error) {
            return go(it.throw(error));
        });
    }
    go(it.next());
}
run(g);
```

上面代码的 Generator 函数 g 之中，有一个异步操作 getFoo，它返回的就是一个 Promise 对象。函数 run 用来处理这个 Promise 对象，并调用下一个 next 方法。

Async 函数

async 函数与 Promise、Generator 函数一样，是用来取代回调函数、解决异步操作的一种方法。

async 函数，就是下面这样。

```
var asyncReadFile = async function (){
    var f1 = await readFile('/etc/fstab');
    var f2 = await readFile('/etc/shells');
    document.write(f1.toString());
    document.write(f2.toString());
};
```

async 函数对 Generator 函数的改进，体现在以下三点：

- 内置执行器　　Generator 函数的执行必须靠执行器，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。
- 更好的语义　　async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。
- 更广的适用性　　co 函数库约定，yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

ES6 中已经将 await 列为保留字，在 ES5 中仍可使用 await 作为变量名。await 命令只能用在 async 函数之中，如果用在普通函数，就会报错。await 命令后面的 Promise 对象，运行结果可能是 rejected，所以最好把 await 命令放在 try...catch 代码块中。

```
async function myFunction() {
    try {
        await somethingThatReturnsAPromise();
    } catch (err) {
        console.log(err);
    }
}

// 另一种写法
async function myFunction() {
    await somethingThatReturnsAPromise().catch(function (err){
        console.log(err);
    })
}
```

后记

ES6 转译器：

Traceur：google 出品基于 node.js 的 ES6 转译器，支持 ES6 在线使用

Babel：基于 node.js 的 ES6 转译器

ES6-shim:兼容包 https://github.com/paulmillr/es6-shim

Jsdc:国产货 https://github.com/army8735/jsdc

在线 ES6 转译器：

Traceur：http://google.github.io/traceur-compiler/demo/repl.html

Babel: https://babeljs.io/repl/

马上使用 ES6：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>es6</title>
    <!-- 加载 Traceur 编译器 -->
    <script src="http://google.github.io/traceur-compiler/bin/traceur.js" type="text/javascript"></script>
    <!-- 将 Traceur 编译器用于网页 -->
    <script src="http://google.github.io/traceur-compiler/src/bootstrap.js" type="text/javascript"></script>
</head>
<body>
<script type="text/javascript">
    traceur.options.experimental = true;//打开实验选项，否则有些特性可能编译不成功
</script>
<script type="module">
    //注意，script 标签的 type 属性的值是 module(或者 traceur)，而不是 text/javascript
    class Calc {
        constructor(){
            console.log('Calc constructor');
        }
        add(a, b){ return a + b; }
    }
    var c = new Calc();
    console.log(c.add(4,5));
</script>
</body>
</html>
```

浏览器支持 ES6 查阅：http://kangax.github.io/compat-table/es6/