

02

OPEN ORIENTED

凹凸实验室

ES6(一)

暖暖

let和const
变量的解构赋值
字符串的扩展
数值的扩展
正则的扩展
数组的扩展
函数的扩展
对象的扩展

let和const

	var	let	const
作用	用来声明变量	用来声明变量	用来声明常量
变量提升	✓	✗	✗
块级作用域	✗	✓	✓
暂时性死区	✗	✓	✓
同一作用域重复声明	✓	✗	✗
全局对象的属性	✓	✗	✗

	var	let	const
作用	用来声明变量	用来声明变量	用来声明常量
变量提升	✓	✗	✗
块级作用域	✗	✓	✓
暂时性死区	✗	✓	✓
同一作用域重复声明	✓	✗	✗
全局对象的属性	✓	✗	✗

- const一旦声明常量，必须立即初始化。
- const一旦声明常量，值就不能改变。
- 对于复合类型的变量，const的变量指向数据所在的地址，const命令只是保证变量名指向的地址不变，并不保证该地址的数据不变，所以依然可以为其添加新属性。

	var	let	const
作用	用来声明变量	用来声明变量	用来声明常量
变量提升	✓	✗	✗
块级作用域	✗	✓	✓
暂时性死区	✗	✓	✓
同一作用域重复声明	✓	✗	✗
全局对象的属性	✓	✗	✗

- const一旦声明常量，必须立即初始化。
- const一旦声明常量，值就不能改变。
- 对于复合类型的变量，const的变量指向数据所在的地址，const命令只是保证变量名指向的地址不变，并不保证该地址的数据不变，所以依然可以为其添加新属性。

```
// 复合类型
const aaa = [];
aaa.push("Hello"); // 可执行
aaa.length = 2
```

```
// 最终aaa的结果为 ["Hello", undefined]
```

	var	let	const
作用	用来声明变量	用来声明变量	用来声明常量
变量提升	✓	✗	✗
块级作用域	✗	✓	✓
暂时性死区	✗	✓	✓
同一作用域重复声明	✓	✗	✗
全局对象的属性	✓	✗	✗

暂时性死区（temporal dead zone）：只要块级作用域内存在let命令，它所声明的变量就不再受外部的影响。

	var	let	const
作用	用来声明变量	用来声明变量	用来声明常量
变量提升	✓	✗	✗
块级作用域	✗	✓	✓
暂时性死区	✗	✓	✓
同一作用域重复声明	✓	✗	✗
全局对象的属性	✓	✗	✗

暂时性死区（temporal dead zone）：只要块级作用域内存在let命令，它所声明的变量就不再受外部的影响。

```
var tmp = 123;

if (true) {
  // TDZ开始
  tmp = 'abc'; // ReferenceError, 变量tmp的“死区”
  typeof tmp; // ReferenceError, 用到该变量就报错, typeof操作也不例外
  let tmp; // TDZ结束
}
```


	var	let	const
作用	用来声明变量	用来声明变量	用来声明常量
变量提升	✓	✗	✗
块级作用域	✗	✓	✓
暂时性死区	✗	✓	✓
同一作用域重复声明	✓	✗	✗
全局对象的属性	✓	✗	✗

暂时性死区（temporal dead zone）：只要块级作用域内存在let命令，它所声明的变量就不再受外部的影响。

```
var tmp = 123;

if (true) {
  // TDZ开始
  tmp = 'abc'; // ReferenceError, 变量tmp的“死区”
  typeof tmp; // ReferenceError, 用到该变量就报错, typeof操作也不例外
  let tmp; // TDZ结束
}

// 执行结果:
// Uncaught ReferenceError: tmp is not defined
```

变量的解构赋值

解构（Destructuring）： 允许按照一定模式，从数组或对象等等中提取值，对变量进行赋值。

	数组的解构赋值	对象的解构赋值
基本用法	<pre>// 等同于 var a = 1,b = 2,c = 3; var [a, b, c] = [1, 2, 3];</pre>	<pre>// var foo = "aaa"; // var bar = undefined; var { bar, foo } = { foo: "aaa", bazzz: "bbb" };</pre>
备注	数组的元素变量取值由它的位置决定	<p>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。</p> <p>如果变量名与属性名不一致，则这么写，此时变量的声明和赋值是一体的：</p> <pre>// var f = "aaa",b="bob"; var { foo: f, bar: b } = { foo: "aaa", bar: "bbb" };</pre>

	字符串的解构赋值	数值和布尔值的解构赋值	函数参数的解构赋值
基本用法	<pre>// 字符串被转换成类似数组的对象 let { a, b } = 'hi'; console.log(a,b); // h,i</pre>	<pre>let {toString: s} = 123; s === Number.prototype.toString // true</pre>	<pre>function add([x, y]){ return x + y; } add([1, 2]) // 3</pre>
备注	<pre>// 数组的对象都有一个length属性，因此还可以对这个属性解构赋值 let {length:len} = 'hello'; console.log(len); //5</pre>	<p>解构赋值的规则是，只要等号右边的值不是对象，就先将其转为对象。 不能转为对象，则报错，如：</p> <pre>let [foo] = 1; let [foo] = false; let [foo] = NaN; let [foo] = undefined; let [foo] = null; let [foo] = {};</pre>	

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

```
// 报错，因为foo这时等于undefined，再取子属性就报错。  
var {foo: {bar}} = {baz: 'baz'}
```

将一个已经声明的变量用于解构赋值，JavaScript引擎会将{x}理解成一个代码块，从而导致错误。

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

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

字符串的扩展

	codePointAt()	String.fromCodePoint()	for...of	at()
对比函数	charCodeAt()	String.fromCharCode	传统的for循环	charAt()
栗子	<pre>// 将“吉a”视为三个字符，a是第三个字符 var s = '吉a'; // 返回十六进制 console.log(s.codePointAt(0).toString(16)) // 20bb7 console.log(s.charCodeAt(0).toString(16)) // d842</pre>	<pre>// 返回true String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x \uD83D\uDE80y'</pre>	<pre>var text = String.fromCodePoint(0x20BB7) ; for (let i of text) { console.log(i); } // "吉"</pre>	需要垫片库实现
相同与不同	<p>相同：两者对比，作用基本一致。</p> <p>不同：ES6可以正确识别32位的UTF-16字符</p>			

	includes()	startsWith()	endsWith()	repeat()	padStart(), padEnd()
作用	返回布尔值，表示是否找到了参数字符串	返回布尔值，表示参数字符串是否在源字符串的头部。	返回布尔值，表示参数字符串是否在源字符串的尾部。	返回一个新字符串，表示将原字符串重复n次。 参数： 小数，则取整； 负数或者Infinity则会报错； 0到-1之间的小数，则等同于0。字符串，则会先转换成数字； NaN等同于0。	字符串补全长度的功能。 第一个参数用来指定字符串的**最小长度**，第二个参数是用来补全的字符串，省略第二个参数，则会用空格补全长度。
	第二个参数，表示开始搜索的位置。				
栗子	<pre>var s = 'Hello world!'; s.startsWith('world', 6) // true s.endsWith('Hello', 5) // true s.includes('Hello', 6) // false s.includes('Hello') // true</pre>			<pre>x'.repeat(3) // "xxx"</pre>	<pre>'12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"</pre>

模板字符串

- 模版字符串用反引号(`)来标识，支持多行字符串，保留所有的空格和缩进。
- 模版字符串需要替换的变量需要写在\${}之中。

```
$('#result').html(  
  `    <p>${result.num}</p>  
  </div>`  
)
```

标签模版

- 函数调用的一种特殊形式。“标签”指的就是函数的标识名，紧跟在后面的模板字符串就是它的参数。
- 第一个参数是一个数组，后面的其它参数都是模版字符串各个变量被替换后的值。

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

```
function tag(s, v1, v2) {  
  return `${s.join(',')},${v1},${v2}`;  
}
```

```
tag`Hello ${ a + b } world ${ a * b }`;  
// 返回"Hello , world ,,15,50"
```

正则的扩展

- RegExp构造函数
- 所有与正则相关的方法，全都定义在RegExp对象上
- 修饰符：u、i、y
- 属性：sticky、flags
- 后行断言

数值的扩展

ES6提供了二进制和八进制数值的新的写法，分别用前缀0b**（或**0B**）和**0o**（或**0O**）表示。**

```
Number('0b111') // 7
```

```
Number('0o10') // 8
```

	Number.isFinite(), Number.isNaN()	Number.parseInt(), Number.parseFloat()
作用	传统的全局方法isFinite()和isNaN()的区别在于：这两个新方法只对数值有效，非数值一律返回false。	与parseInt()和parseFloat(), 行为不变
	移植到Number对象上面，逐步减少全局性方法，使得语言逐步模块化。	

	Number.isInteger()	Number.EPSILON	安全整数	Number.isSafeInteger()
基本用法	判断是否是整数	一个极小的常量Number.EPSILON，目的在于为浮点数计算，设置一个误差范围。如果这个误差能够小于Number.EPSILON，我们就可以认为得到了正确结果。	安全整数的范围是在-2^53到2^53之间（不含两个端点）。 Number.MAX_SAFE_INTEGER Number.MIN_SAFE_INTEGER 这两个常量，用来表示这个范围的上下限。	用来判断一个整数是否落在安全整数范围之内
栗子	// 整数和浮点数是同样的储存方法，所以被视为同一个值 Number.isInteger(25) // true Number.isInteger(25.0) // true	Number.EPSILON === 2.220446049250313e-16 // 返回true	Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1 // true	

- Math对象的扩展：
新增了17个与数学相关的方法，包括对数方法、三角函数等
- 指数运算符：`**=`

```
let b = 3;  
b **= 3;  
// 等同于 b = b * b * b; 最后b为27  
b **= 4;  
// 等同于 b = b * b * b * b;
```

数组的扩展

	Array.from()	Array.of()	copyWithin()	fill()
作用	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。	返回参数值组成的数组； 目的在于弥补数组构造函数Array()的参数个数的不同，会导致Array()的行为有差异。	将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。	填充数组元素
栗子	// 返回 ['h', 'e', 'l', 'l', 'o'] Array.from('hello') // 返回['jack', 'jack'] Array.from({ length: 2 }, () => 'jack')	// 行为一致 Array.of(1) // [1] Array.of(1, 2) // [1, 2] // 参数不同，行为不同 Array(3) // [, ,] Array(3, 11, 8) // [3, 11, 8]	// 将3号位复制到0号位 [1, 2, 3, 4, 5].copyWithin(0, 3, 4) // [4, 2, 3, 4, 5]	// fill方法从1号位开始，向原数组填充7，到3号位之前结束。 ['a', 'b', 'c'].fill(7, 1, 3) // ['a', 7, 7]

	find()和findIndex()	entries(), keys()和values()	includes()
作用	<p>find(): 用于找出第一个符合条件的数组成员。</p> <p>findIndex(): 返回第一个符合条件的数组成员的位置。</p>	<p>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。</p>	<p>返回一个布尔值, 表示某个数组是否包含给定的值</p>
栗子	<pre>// 找出数组中第一个小于0的成员。 [1, 5, 10, 15].find(function(value, index, arr) { return value > 9; }) // 10</pre>	<pre>for (let [index, elem] of ['a', 'b'].entries()) { console.log(index, elem); } // 0 "a" // 1 "b"</pre>	<pre>[1, 2, NaN].includes(NaN); // true // indexOf对NaN的误判 [NaN].indexOf(NaN) // 返回-1</pre>

函数的扩展

函数参数的默认值

```
function log(x, y = 'World',z) {  
  console.log(x, y, z);  
}
```

```
log('Hello') // Hello World undefined  
log('Hello', '') // Hello undefined
```

```
log('Hello', ,2) // 报错，语法错误  
log('Hello',undefined,2) // Hello World 2
```


函数参数的默认值

```
function log(x, y = 'World',z) {  
  console.log(x, y, z);  
}
```

```
log('Hello') // Hello World undefined  
log('Hello', '') // Hello undefined
```

```
log('Hello', ,2) // 报错, 语法错误  
log('Hello',undefined,2) // Hello World 2
```

```
// 函数参数的默认值与解构赋值的默认值  
function move({x=1, y=2} = { x: 0, y: 0 }) {  
  return [x, y];  
}
```

```
// y的解构赋值的默认值生效  
move({x: 3}); // [3, 2]
```

```
// x和y解构赋值的默认值生效, 覆盖了函数默认值  
move({}); // [1, 2]
```

```
// 没有参数时, 函数参数默认值生效  
move(); // [0,0]
```

rest参数

写法: ...变量名

作用: 用于获取函数的多余参数

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)

```
// 报错
function f(a, ...b, c) {
  // ...
}
```

//这里的arr相当于arguments, 但是arr是一个真正的数组

```
function getSum(...arr){
  let sum = 0;
  for ( var i of arr ){
    sum += i;
  }
  return sum;
}
```

```
let sum1 = getSum(1,2,3,4);
console.log( sum1 ); //10
```

扩展运算符

写法: ...

作用: **rest**参数的逆运算, 将一个数组转为用逗号分隔的参数序列。

```
function add(x, y) {  
  return x + y;  
}
```

```
var numbers = [4, 38];  
add(...numbers) // 42
```

扩展运算符

写法: ...

作用: **rest**参数的逆运算, 将一个数组转为用逗号分隔的参数序列。

- 替代数组的apply方法: `Math.max(...[14, 3, 77])`

- 合并数组:

```
var arr1 = ['a', 'b'];  
var arr2 = ['c'];  
[...arr1, ...arr2] // [ 'a', 'b', 'c' ]
```

- 将字符串转为真正的数组

```
[...'hello'] // [ "h", "e", "l", "l", "o" ]
```

- 任何Iterator接口的对象, 都可以用扩展运算符转为真正的数组。

```
var nodeList = document.querySelectorAll('div');  
var array = [...nodeList];
```

函数的length属性

length属性的含义是，该函数预期传入的参数个数。
因此指定了默认值、**rest**参数后，**length**属性将失真。

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

```
function log(x, y, z = 'World') {  
  console.log(x, y, z);  
}  
log.length // 2
```

```
function log(x, y, ...z) {  
  console.log(x, y, z);  
}  
log.length // 2
```

函数的name属性

- 将一个匿名函数赋值给一个变量，ES5的name属性，会返回空字符串，而ES6的name属性会返回实际的函数名。
- Function构造函数返回的函数实例，name属性值为“anonymous”
- bind返回的函数，name属性值会加上“bound ”前缀。

```
function foo() {};  
foo.bind({}).name // "bound foo"
```

箭头函数

包含了箭头 (=>) 的函数

- this指向的固定化：箭头函数没有自己的this，因此它的this就是外层代码块的this。因为没有this，所以也就不能用作构造函数。
- 除了this，以下三个变量在箭头函数之中也是不存在的，指向外层函数的对应变量：
arguments、super、new.target
- 不可以使用yield命令，因此箭头函数不能用作Generator函数。

```
// this指向的固定化
function foo() {
  console.log(this);
  setTimeout( () => {
    console.log(this);
  },100);
}
```

```
foo() // window window
foo.call({a: 'a'}) // Object {a: "a"} Object {a: "a"}
```


对象的扩展

属性的简洁表示法

```
var o = {  
  method() { // 等同于 method: function() {  
    return "Hello!";  
  }  
};
```

属性的简洁表示法

```
var o = {  
  method() { // 等同于 method: function() {  
    return "Hello!";  
  }  
};
```

属性名表达式

ES6允许字面量定义对象时，把表达式放在方括号内；还可以用于定义方法名。

```
let obj = {  
  [propKey]: true,  
  ['a' + 'bc']: 123  
};
```

属性的简洁表示法

```
var o = {  
  method() { // 等同于 method: function() {  
    return "Hello!";  
  }  
};
```

属性名表达式

ES6允许字面量定义对象时，把表达式放在方括号内；还可以用于定义方法名。

```
let obj = {  
  [propKey]: true,  
  ['a' + 'bc']: 123  
};
```

注意，属性名表达式与简洁表示法，不能同时使用!!!

方法的name属性

对象方法也是函数，因此也有name属性。

- 使用了取值函数，则会在方法名前加上get；
- 存值函数，方法名的前面会加上set；
- bind方法创造的函数，name属性返回“bound”加上原函数的名字；
- Function构造函数创造的函数，name属性返回“anonymous”；
- 对象的方法是一个Symbol值，那么name属性返回的是这个Symbol值的描述。

	Object.is(值一,值二)	Object.assign()	Object.values(), Object.entries()
基本用法	<p>用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。</p> <p>不同之处只有两个：Object.is的行为是+0不等于-0，NaN等于自身。</p>	<p>Object.assign方法用于对象的合并，将源对象（source）的所有可枚举自有属性，可覆盖性地复制到目标对象（target）；属于浅拷贝</p> <p>如果target不能转成对象，就会抛出TypeError错误；source不能转为对象则直接忽略这个参数</p>	<p>Object.values 返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。会过滤属性名为Symbol值的属性。</p> <p>Object.entries 返回一个键值对数组。该方法的行为与Object.values基本一致。</p>
栗子	<pre>+0 === -0 //true NaN === NaN // false Object.is(+0, -0) // false Object.is(NaN, NaN) // true</pre>	<pre>// 栗子1 var target = { a: 1, b: 1 }; var source1 = { b: 2, c: 2 }; Object.assign(target, source1); target // {a:1, b:2, c:2} // 栗子2 // String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"} Object.assign('abc');</pre>	<pre>var obj = { foo: 'bar', baz: 42 }; Object.values(obj) // ["bar", 42] Object.entries(obj) // [["foo", "bar"], ["baz", 42]]</pre>

	Object.setPrototypeOf(), Object.getPrototypeOf()	Object.getOwnPropertyDescriptors()
基本用法	Object.setPrototypeOf()（写操作）、 Object.getPrototypeOf()（读操作）、 Object.create()（生成操作）代替__proto__内部属性， 用来读取或设置当前对象的prototype对象。	ES7提案：Object.getOwnPropertyDescriptors(obj)， 返回指定对象所有自身属性（非继承属性）的描述对象。 该方法的提出目的，主要是为了解决Object.assign()无法正确拷贝get属性和set属性的问题。
栗子	Object.setPrototypeOf(object, prototype) Object.getPrototypeOf(obj);	const obj = Object.create(prot, Object.getOwnPropertyDescriptors({ foo: 123, }));

<http://es6.ruanyifeng.com/>

02

OPEN ORIENTED

凹凸实验室

THANKS FOR LISTENING

王彩暖