Web前端的知识之旅哟——ES6-let与const(四十三)

终于写到ES6了！

ES6在ES5的基础上增加了许多新功能和新用法，现在js更加倾向于面向对象的语言了，而且用过es6的写法可以大大简化代码。

不过很遗憾的是只有当前的一些高版本的浏览器才支持ES6，因此我们需要ES6的转码器babel，利用nodejs的npm功能就可以下载，具体的配置情况大家参考一下网上吧~

那么下面进入正题

let命令和const命令

let命令和var很类似，也是用来声明变量的，但是所声明的变量，只有在let命令所在的代码块内才有效，这也是es6最大的特点之一：增加了**块级作用域**。

正是因为块级作用域的存在，我们在es5中需要用闭包解决的东西，现在可以直接用let来解决了，比如我们经典的点击弹出的问题，以前需要用闭包去解决，现在我们只需要在for循环里面声明i的时候用let命令就可以了。

同时let不存在变量提升的问题，不会像var那样将变量提升到作用域的顶部，let声明的变量一定要在let声明之后使用，否则会报错。

暂时性死区：在let命令声明变量之前，该变量都是不可用的。只要会计作用域之内存在let，它所声明的变量就绑定到了这个区域之内，不收外部影响。

比如说我们有一个全局的demo变量，在一个块之内又用let声明了一个demo变量，然后调用这个demo，那么不管声明在调用的前面还是后面，都不会去调用那个全局的demo变量。

暂时性死区的本质就是：只要进入当前作用域，所要使用的变量就已经存在了，但是不可以获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

es6明确规定：如果区块中有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性作用域，凡是在声明之前就使用这些变量就会报错。

let a = b= 123;

这也是一个比较隐蔽的死区，可能不太容易发现。

同时typeof也变得不太安全了。

es6中的暂时性死区和无变量提升，主要是为了减少运行时的错误，防止在变量声明之前就使用这个变量，从而导致意料之外的错误。

不允许重复声明：在同一个作用域之内，let和const都不允许重复声明同一个变量，包括变量和形参的名字相同的情况。

const命令

const声明一个只读的常量，一旦声明，常量的值就不允许改变了。

比如const PI = 3.14这样的常量。

const声明的变量不能被修改，这意味着我们只能在声明的同时就赋值，不能先声明后赋值。

不过const对于一个引用值来说有点特殊，const只是保证变量指向的地址不变，当我们const声明一个对象或者数组的时候：

const obj = {};

虽然我们不能改变obj的指向，但是我们可以正常地对obj的属性进行增删改查的操作。

obj.name = ‘ScarLett’;

诸如此类。

块级作用域

let实际上为js新增加了块级作用域。

es6允许块级作用域的任意嵌套：

{{{{let a = 1;}}}}

这种写法是没有问题的。

内层作用域可以定义一个和外层作用域的同名变量，外层作用域无法读取内层作用域的变量。

块级作用域的出现，使得es5中的立即执行函数不在有太多的必要了（仅限解决闭包的问题的时候）。

虽然es6中允许了函数在作用域块内部声明，但是考虑到环境导致的行为差异太大，应该避免在块级作用域内部声明函数，如果确实需要，应该写生函数表达式，而不是函数声明语句。

（ES5规定，函数只能在顶层作用域，也就是全局韩静，和函数作用域之中声明，不能再块级作用域中声明。不过大多数浏览器都没有遵守这个规定……）

顶层对象的属性

这里的顶层对象指的就是我们的window这一级别的对象。

在es5中，我们在全局环境声明一个变量，相当于把这个变量挂载到了window对象上面：

var a = 1 <=> window.a = 1;

这就导致了变成人员在写的时候经常会不知不觉地创建了全局变量，这样很不利于模块化变成。

在es6中，var和function 命令声明的全局变量，依然是window的属性，但是let、const、class声明的全局变量，不再是window的属性了。

那么这次ES6的开始就介绍到这里哟~

Web前端的知识之旅哟——ES6-解构赋值(四十四)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-15

* 围观(208)

* [评论(0)](http://www.flyingliu.com/2016/11/15/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-%e8%a7%a3%e6%9e%84%e8%b5%8b%e5%80%bc%e5%9b%9b%e5%8d%81%e5%9b%9b/#comments)

数组的解构赋值

es6允许按照一定的模式，从数组和对象中提取值，对变量进行赋值，这被称之为解构赋值。

比如在es5中：

2. var a = 1, b = 2, c = 3;

JavaScript

在es6中就可以通过一个数组来简写：

2. let [a,b,c] = [1,2,3];

JavaScript

这样也是可以的：

2. let [a,[b],c] = [1,[2],3];

JavaScript

只要按照对应的位置进行赋值，我们可以任意嵌套。

2. let [,,c] = [1,2,3] *// 3*
3. let [head, ...tail] = [1,2,3,4];
4. head => 1 tail =>[2,3,4]

JavaScript

如果解构失败，变量的值就是undefined

2. let [a,b,...c] = [1];
3. *//a=>1 b=>undefined c=>[]*

JavaScript

还有一种情况是不完全解构，即左边的变量数量个数小于右边的值的个数，这样的情况也是可以解构成功的，按照变量对应的位置进行解构即可。

如果等号的右边不是数组（或者严格来说，不是可遍历的结构），那么将会报错。

2. let [foo] = 1;
3. let [foo] = false;
4. let [foo] = NaN;
5. let [foo] = undefined;
6. let [foo] = null;
7. let [foo] = {};

JavaScript

这些都会报错。

解构赋值对于var、let、const命令都适用。

解构赋值与允许执行默认值

2. let [a,b=2] = [1];

JavaScript

只有数组的值严格等于undefined，才会被赋予默认值，否则即使是null和false也会被正常赋值。

2. let [x=1] = [undefined]; *// x = 1;*
3. let [x=1] = [null]; *// x = null;*

JavaScript

如果我们把这个默认值给设置成了一个函数，即使我们在默认值那里执行了这个函数：

2. function f() {
3. console.log('aaa');
4. }
6. let [x = f()] = [1];

JavaScript

如果x被成功解构赋值了，没有赋值成默认值，那么这个函数就不会被执行，因为根本js根本不会走那里的代码，因此看上去我们执行了f函数，但是其实并没有，所以不会打印aaa的。

默认值可以引用解构赋值的其他变量，但是前提是该变量必须已经声明。

2. let [x = 1, y = x] = []; *// x=1; y=1*
3. let [x = 1, y = x] = [2]; *// x=2; y=2*
4. let [x = 1, y = x] = [1, 2]; *// x=1; y=2*
5. let [x = y, y = 1] = []; *// ReferenceError*

JavaScript

最后一个之所以会报错，是因为x用到默认值y的时候，y还没有被声明。

对象的解构赋值

解构赋值不仅仅可以用于数组，也可以用于对象

2. var {a,b} = {a:123, b:'234'};
3. *// a = 123;*
4. *// b = '234';*

JavaScript

对象的解构赋值和数组的解构赋值的区别在于：数组的元素是按照次序排序的，因此它们赋值也是按照顺序赋值的，但是对象的属性是没有次序的，变量必须与属性同名，才会被赋值。

2. var {a,b} = {b:'234',a:123};
3. *// a = 123;*
4. *// b = '234';*

JavaScript

而与数组的解构赋值一样，如果我们的变量在对象中没有找到同名的属性的话，那么这个变量的值就是undefined。

对象的解构赋值的内部机制是：先找到同名的属性，然后把这个属性值赋予给对应的变量。

也就是说，这样的写法也是可以被赋值的：

2. {a:demo} = {a:123};
3. *//demo 123*

JavaScript

现在将a的属性值123拿出来赋值给变量a，但是变量a的值是demo，因此就是把demo的值变成了123。

注意这里demo才是真正的变量，a只是一个匹配模式，模式是不会被赋值的。

这样就完成了即使没有同名属性可以成功赋值的任务。

同时对象的解构赋值也只是可以有默认值的，条件和数组的一样。

这里需要注意的是：

如果我们要给一个已经声明的变量进行解构赋值的话，可能会这样写：

2. let x;
3. {x} = {x:1}

JavaScript

但是由于es6中块级作用域的存在，js引擎会将{}理解成一个代码块，从而导致语法错误。

我们要在外面加一层括号：

2. let x;
3. ({x} = {x:1})

JavaScript

这样就可以正确地识别了。

由于数组本质也是特殊的对象，因此可以对数组进行对象属性的解构：

2. var arr = [1, 2, 3];
3. var {0 : first, [arr.length - 1] : last} = arr;
4. first *// 1*
5. last *// 3*

JavaScript

字符串也可以被解构赋值，和数组非常类似。

2. const [a, b, c, d, e] = 'hello';
3. a *// "h"*
4. b *// "e"*
5. c *// "l"*
6. d *// "l"*
7. e *// "o"*

JavaScript

由于类数组和对象都有length属性，我们还可以专门对这个属性进行解构赋值：

2. let {length : len} = 'hello';
3. len *// 5*

JavaScript

解构赋值的时候，如果等号右边是数组和布尔值，则会有限转化成对象。

2. let {toString: s} = 123;
3. s === Number.prototype.toString *// true*
5. let {toString: s} = true;
6. s === Boolean.prototype.toString *// true*

JavaScript

由于undefined和null无法转化为对象，所以对他们进行解构赋值都会报错。

函数参数的解构赋值

函数的参数也是可以进行解构赋值的

2. function add([x, y]){
3. return x + y;
4. }
6. add([1, 2]); *// 3*

JavaScript

add函数的参数在传入数组的一瞬间就被解构赋值成了x、y两个变量了，因此在函数内部使用的是x、y变量而不是数组。

同时参数的解构赋值也可以有默认值，当我们不传入实参的时候就会赋值为默认值。

2. function move({x = 0, y = 0} = {}) {
3. return [x, y];
4. }
6. move({x: 3, y: 8}); *// [3, 8]*
7. move({x: 3}); *// [3, 0]*
8. move({}); *// [0, 0]*
9. move(); *// [0, 0]*

JavaScript

但是要注意这种写法和下面这一种是不一样的：

2. function move({x, y} = { x: 0, y: 0 }) {
3. return [x, y];
4. }
5. move({x: 3, y: 8}); *// [3, 8]*
6. move({x: 3}); *// [3, undefined]*
7. move({}); *// [undefined, undefined]*
8. move(); *// [0, 0]*

JavaScript

二者的主要区别在于默认值的写法上，第一种是给x、y变量指定默认值，而第二种是给move函数指定默认值而不是x、y变量。

这一点一定要注意！！

还有一点：圆括号可能会对解构赋值带来影响，因此能不用圆括号就不用。

解构赋值的用途

1.交换变量

2. [x, y] = [y, x];

JavaScript

2.函数可以有多个返回值

2. return [1, 2, 3];

JavaScript

然后可以通过解构赋值给多个变量赋值

3.函数参数的定义

解构赋值可以方便的把形参和实参对应起来，只需要形参是一个对象即可，不需要是按照固定的顺序写实参，就像我们在es5中用的json参数一样。

2. function f({x, y, z}) { ... }
3. f({z: 3, y: 2, x: 1});

JavaScript

4.提取JSON数据

2. let { id, status, data: number } = json;

JavaScript

不仅仅是json，只要是个对象我们都可以把属性值快速的提取出来。

5.函数参数的默认值

6.遍历Map解构

map可以用for…of循环遍历，解构赋值可以方便获得键/值。

2. for (let [key, value] of map) {
3. console.log(key + " is " + value);
4. }

JavaScript

而且可以只获取键名或者键值

2. *// 获取键名*
3. for (let [key] of map) {
4. *// ...*
5. }
6. *// 获取键值*
7. for (let [,value] of map) {
8. *// ...*
9. }

JavaScript

7.配合requireJS

2. const {src} = require('source');

JavaScript

这样可以直接获取source模块中的src变量。

当然也可以配合es6中的import来使用。

Web前端的知识之旅哟——ES6-字符串的扩展(四十五)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-15

* 围观(256)

* [评论(0)](http://www.flyingliu.com/2016/11/15/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-%e5%ad%97%e7%ac%a6%e4%b8%b2%e7%9a%84%e6%89%a9%e5%b1%95%e5%9b%9b%e5%8d%81%e4%ba%94/#comments)

字符串的扩展

1.字符的Unicode表示

在es6中，js允许使用“\uxxxx”的形式来表示一个字符，其中xxxx叫做这个字符的码点。

\u0061 -> a

不过xxxx的范围只能是0000-ffff，超出这个范围的字符就需要用两个双字节来表示了。

“\uD842\uDFB7”这两个码点表示了一个汉字

如果我们\u后面跟的十六进制数字超过了ffff，那么js默认会变成xxxx+x的形式来进行解析，这就和我们预期的字符有差别了，不过我们可以通过给码点加上大括号来让js正确解析字符。

“\u{20BB7}”

Unicode表示的字符和正常写出来的字符是完全一样的。

正式由于有了字符的Unicode表示的方法，因此衍生出了字符串的很多和码点有关的方法。

2.codePointAt()

js内部，字符都是utf-16形式储存的，每个字符固定两个字节，即一个Unicode码点。

对于两个Unicode码点储存的字符，js默认是四个字节。

2. var s = "𠮷";
3. s.length *// 2*

JavaScript

对于这种4个字节的字符，js不能正确处理，字符长度会被误判为2，而且charAt方法无法读取整个字符，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，可以正确处理4个字节的字符，返回一个字符的码点。

2. var s = "𠮷a";
3. s.codePointAt(0) *// 134071*

JavaScript

但是这里要注意的是，js依然将“𠮷”这个字认为是4个字节的，也就是说s这个字符串的长度为3，后面的字符a是在第三位上的，codePointAt（1）的解构和charCodeAt的结果是一样的，我们要查看后面的a需要codePointAt（2）。

codePointAt方法返回的码点是十进制的，如果需要十六进制的可以用toString方法转一下。

2. var s = "𠮷a";
3. s.codePointAt(0).toString(16) *// "20bb7"*

JavaScript

虽然这个方法解决了我们的四字节的问题，但是还是有缺陷的，我们的四字节依然被认为是两个长度，在我们现在的观念中a应该是下标为1的，长度应该一共是2，但是现在的情况依然是长度为3，必须查看下标2才能查看到字符a。

这个时候我们可以用前面提到过的for…of循环，它可以正确识别32位的四字节字符。

2. var s = '𠮷a';
3. for (let ch of s) {
4. console.log(ch.codePointAt(0).toString(16));
5. }
6. *// 20bb7*
7. *// 61*

JavaScript

最后，charPointAt()方法是检测一个字符是否是四字节的最简单的方法。

3.String.fromCodePoint()

注意这里是String上的静态方法。

在es5中，我们有String.fromCharCode可以配合ASCII码使用来检测是哪一个字符，但是只能是双字节的字符，现在碰到四字节的字符就会出现问题。

这个时候我们就需要用es6的fromCodePoint方法了。

2. String.fromCodePoint(0x20BB7)
3. *// "𠮷"*

JavaScript

4.at()

es5中我们有一个charAt方法可以判断字符串中第n为是什么字符，但是也会因为4字节字符的存在导致出现问题。

es6的提案中，有一个at()方法可以用来判断四字节字符。

不过现在这个方法还在提案中，并没有被真正实现。

5.includes()、startsWith()、endsWith()

es5中，我们只有indexOf方法，可以判断一个字符串是否包含我们需要查找的字符串。

es6中又添加了三个新方法：

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

2. const s = 'Hello world!';
3. s.startsWith('Hello') *// true*
4. s.endsWith('!') *// true*
5. s.includes('o') *// true*

JavaScript

这三个方法都有两个参数，第一个参数是我们要查询的参数字符串，第二个参数是可选参数，表示开始查询的位置。

2. const s = 'Hello world!';
3. s.startsWith('world', 6) *// true*
4. s.endsWith('Hello', 5) *// true*
5. s.includes('Hello', 6) *// false*

JavaScript

6.repeat()

将字符串重复n次，参数是正整数

如果参数是小数，会被自动向下取整

如果是负数或者无穷大，会报错

如果是0到-1之间的小数，则会等同于0

NaN等同于0

如果参数是字符串，会先转化成数字，如果可以就按照数字来进行，否则按照NaN处理。

7.模板字符串

在es5中，我们要输出模板通常是把很多字符串用“+”连接起来，很不方便。

2. $('#result').append(
3. 'There are **' + basket.count + '** ' +
4. 'items in your basket, ' +
5. '*' + basket.onSale +*
6. *'* are on sale!'
7. );

JavaScript

在es6中引入了模板字符串，可以有效地解决这个问题。

我们只需要用反引号“`”即可。

其中，如果是js中的变量，则放到${}里面即可。

2. $('#result').append(`
3. There are **${basket.count}** items
4. in your basket, *${basket.onSale}*
5. are on sale!
6. `);

JavaScript

这个东西配合react使用尤其好用，而且形式也和react很像。

不过我们用es6的模板字符串会将我们前后多余的空格也算进去，所以最好在模板字符串后面加上.trim()，用来把前后多余的空格都去掉。

模板字符串也是可以调用函数的，只要放在${}中间就可以了，这个${}相当于在字符串中插入了js作用空间。

8.标签字符串

如果把模板字符串写在了函数后面，那么这个模板字符串就会被当做是函数的参数来调用函数了。

2. alert`123`
3. *// 等同于*
4. alert(123)

JavaScript

如果模板字符串还有变量的话，那就会先处理变量变成参数，然后再调用函数。

Web前端的知识之旅哟——ES6-数值和数组的扩展(四十六)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-15

* 围观(339)

* [评论(1)](http://www.flyingliu.com/2016/11/15/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-%e6%95%b0%e5%80%bc%e5%92%8c%e6%95%b0%e7%bb%84%e7%9a%84%e6%89%a9%e5%b1%95%e5%9b%9b%e5%8d%81%e4%ba%94/#comments)

数值的扩展

es6中，在Number对象上，新增了Number.isFinite()和Number.isNaN()方法。

Number.isFinite()是检测一个数值是否是有限的，如果是无限的则返回false。

Number.isNaN()是检测非数的，如果是NaN则返回true。

在es5中，isNaN方法是全局window上的一个方法，在es6中Number对象也有了。

不止是isNaN方法，还有许多以前的全局方法都复制了一份到Number对象上。

Number.parseInt();

Number.parseFloat();

Number.isInteger(); 这个方法可以判断一个数值是否是整数。

然而在js内部，整数和浮点数都是同样的储存方法，所以1和1.0都会被认为是整数，他们两个是同一个值。

Number.EPSILON

这是es6在Number对象上新增的一个极小的常量，大约是2.220446049250313e-16。

引入这个极小常量的目的是方便浮点数的计算，设置一个误差范围。

我们都知道浮点数的计算是不精确的，我们只要认为这个误差能够小于Number.EPSILON，就可以认定这个结果是正确的，因此，Number.EPSILON的实质是一个可以接受的误差范围。

Math对象也扩展了很多方法，但是使用频率不是很高，大家有兴趣可以了解一下。

数组的扩展

1.**Array.from()**

这个方法用于将类数组和可比遍历的对象（包括es6的Set和Map数据结构）转化成真正的数组。

2. let arrayLike = {
3. '0': 'a',
4. '1': 'b',
5. '2': 'c',
6. length: 3
7. };
8. *// ES6的写法*
9. let arr2 = Array.from(arrayLike); *// ['a', 'b', 'c']*

JavaScript

实际上比较常见的是操作DOM的类数组，我们可以通过这个方法直接转化成真正的数组。

Array.from()还接受第二个参数，作用类似数组的map方法，用来对每个元素进行处理，然后将处理后的值放入到返回的数组中。

2. Array.from(arr, x => x \* x);
3. let spans = document.querySelectorAll('span.name');
4. let names2 = Array.from(spans, s => s.textContent)

JavaScript

这样就很方便地把每一个span的文本内容放入到了数组里面。

Array.from()的另一个应用是，将字符串转化成数组，然后返回字符串的长度。

因为它能够正确处理各种Unicode字符，可以避免Javascrit将大于\uFFFF的Unicode字符算成两个字符的bug。

2. function countSymbols(string) {
3. return Array.from(string).length;
4. }

JavaScript

2.**Array.of()**

Array.of()方法用于将一组数值转化成数组。

2. Array.of(3, 11, 8) *// [3,11,8]*
3. Array.of(3) *// [3]*
4. Array.of(3).length *// 1*

JavaScript

这个方法的主要目的是弥补数组构造函数Array()的不足。在数组的构造函数中，如果只有一个值，那么这个值被认作为是数组的长度而不是数组的第一个值。

在Array.of中就不存在这种因为参数不同而导致的重载的情况，它的行为非常统一，可以替代new Array()。

3.数组实例的**copyWithin()**方法

这个方法是在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回该数组。

它接受三个参数：

target（必选）：从该位置开始替换数据。

start（可选）：从该位置开始读取数据，默认为0。如果是负值，表示倒数。

end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

2. [1, 2, 3, 4, 5].copyWithin(0, 3)
3. *// [4, 5, 3, 4, 5]*

JavaScript

从下标为3的地方开始读取，读到最后，然后从数组的第一位开始替换。

4.数组实例的**find()**和**findIndex()**方法

数组实例的find方法，用于找到第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员以此执行该回调函数，直到找出第一个返回值true的成员，然后返回该成员。

如果没有符合条件的成员，则返回undefined。

2. [1, 4, -5, 10].find((n) => n < 0)
3. *// -5*

JavaScript

find方法的回调函数可以接受三个参数，以此为当前的值、当前的位置和原数组。

2. [1, 5, 10, 15].find(function(value, index, arr) {
3. return value > 9;
4. }) *// 10*

JavaScript

数组实例的findIndex方法的用法和find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

2. [1, 5, 10, 15].findIndex(function(value, index, arr) {
3. return value > 9;
4. }) *// 2*

JavaScript

折两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

**并且这两个方法都可以发现NaN，弥补了数组的IndexOf方法的不足。**

2. [NaN].indexOf(NaN)
3. *// -1*

JavaScript

在es5中的indexOf是无法找到NaN的。

5.数组实例的**fill()**方法

fill方法使用给定值，填充一个数组，一般用于数组的初始化。

2. new Array(3).fill(7)
3. *// [7, 7, 7]*

JavaScript

fill方法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。

2. ['a', 'b', 'c'].fill(7, 1, 2)
3. *// ['a', 7, 'c']*

JavaScript

6.数组实例的**entries()、key()、values()**方法

这三个方法都用于遍历数组，唯一的区别在于：

keys是对键名的遍历

values是对值的遍历

entries是对键值对的遍历

7.数组实例的**includes()**方法

这个方法可以判断数组是否包含给定的值，有则返回true，否则返回false。

该方法的第二个参数表示搜索的起始位置，默认是0，如果第二个参数为负数，则表示倒数的位置，如果它大于数组长度，则会重置为从0开始。

**includes也可以对NaN进行判断。**

8.**数组的空位**

in操作符可以检测数组的第n为是否是空位

2. 0 in [undefined, undefined, undefined] *// true*
3. 0 in [, , ,] *// false*

JavaScript

注意undefined不是空位。

es6的方法和es5的不一样，es6明确将空位转为undefined，而不像es5里面有的方法跳过空位进行处理。

Web前端的知识之旅哟——ES6-函数的扩展(四十七)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-16

* 围观(270)

* [评论(0)](http://www.flyingliu.com/2016/11/16/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-%e5%87%bd%e6%95%b0%e7%9a%84%e6%89%a9%e5%b1%95%e5%9b%9b%e5%8d%81%e4%b8%83/#comments)

ES6扩展了很多函数的东西，也简化了函数的写法。

函数的扩展

1.**默认值**

在es5中，我们为了处理没有传参数造成的错误，一般会采用手动设置初始值的写法：

2. function demo(x) {
3. var x = x || 'xxx';
4. }

JavaScript

但是这种写法会有一种情况，那就是如果我们传递进去的是一个false值，这样它就不会是我们传递的值而是变成了默认值。

在es6中，函数的参数被允许设置初始值，写在参数定义后面即可

2. function demo(x = 0) {
3. *// code*
4. }

JavaScript

这样的好处就是，一个是解决了false的bug，另一个是很简洁，读代码的人一眼就可以看出来哪些参数是可选的，哪些参数比必选的。

另一种默认值的方式就是和我们前面提到的解构赋值配合使用的，这里就不过多说明了。

参数默认值的位置

通常情况下，有默认值的参数一般往后放，这样可以容易看出来那些事可省略参数，而且调用函数传递参数的时候也会方便很多。

2.函数的length属性

我们给函数参数指定了默认值之后，函数的length属性将会返回没有指定默认值的参数的个数，也就是说，指定了默认值之后，length属性将会失真。

2. (function (a) {}).length *// 1*
3. (function (a = 5) {}).length *// 0*
4. (function (a, b, c = 5) {}).length *// 2*

JavaScript

这是因为length属性的含义是，该函数预期被传入的参数的个数。某个参数被指定默认值之后，预期传入的参数个数就不包括这个参数了。

3.rest参数

es6中引入了rest参数，形式为“…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。

2. function add(...values) {
3. let sum = 0;
4. for (var val of values) {
5. sum += val;
6. }
7. return sum;
8. }
9. add(2, 5, 3) *// 10*

JavaScript

这一段代码是一个求和函数，利用rest参数，可以向该函数传递任意数量的参数。

注意：rest参数之后就不能再有其他参数了，即只能是最后一个参数，否则会报错。

2. *// 报错*
3. function f(a, ...b, c) {
4. *// ...*
5. }

JavaScript

这里有一点比较特殊：函数的length属性，不包括rest参数。

2. (function(a) {}).length *// 1*
3. (function(...a) {}).length *// 0*
4. (function(a, ...b) {}).length *// 1*

JavaScript

4.扩展运算符

这里说的扩展运算符其实就是三个点“…”，它比较像上面rest的逆运算，**它将一个数组分化成用逗号分隔的参数序列**。

2. console.log(...[1, 2, 3])
3. *// 1 2 3*

JavaScript

当然扩展运算符主要是配合rest参数在函数里面使用的：

2. function push(array, ...items) {
3. array.push(...items);
4. }

JavaScript

这样rest参数会把剩余的参数都放到一个数组里面，扩展运算符可以把这个参数数组重新分解成一个个的参数。

这样就可以利用扩展运算符替代apply方法了。

扩展运算符的应用：

①.合并数组

es5中连接数组需要用concat，在es6中直接用扩展运算符就可以了

2. *// ES5*
3. [1, 2].concat(more)
4. *// ES6*
5. [1, 2, ...more]

JavaScript

②.与解构赋值配合使用

2. const [first, ...rest] = [1, 2, 3, 4, 5];
3. first *// 1*
4. rest *// [2, 3, 4, 5]*

JavaScript

不过这里和函数参数一样，扩展运算符只能放在参数的最后一位，否则会报错。

③.字符串

扩展运算符还可以将字符串转为真正的数组。

2. [...'hello']
3. *// [ "h", "e", "l", "l", "o" ]*

JavaScript

上面的写法有一个好处：可以正确识别32的Unicode字符。

2. 'x\uD83D\uDE80y'.length *// 4*
3. [...'x\uD83D\uDE80y'].length *// 3*

JavaScript

④.实现了Itreator接口的对象，都可转变成真正的数组，比如Map和Set结构。

5.严格模式

在es5中，函数内部我们可以通过写一句“use strict”来声明使用严格模式，但是在es6中做了一点点修改。

只要函数参数使用了默认值、解构赋值或者扩展运算符，那么这个函数的内部就不能显式地设定为严格模式，否则会报错。

这样规定的原因是：函数内部的严格模式，同时适用于函数体代码和函数参数代码，但是函数执行的时候，会先执行函数参数代码，然后再执行函数体代码，这样就有一个不合理的地方，只有在函数体代码里面，我们才可以知道这个函数是否是严格模式，但是函数参数代码却先与函数体代码执行。

我们如果要使用严格模式，有两种方法：

①.设置全局严格模式

②.把函数包在一个没有参数的立即执行函数里面，将立即执行函数设定为严格模式

2. const doSomething = (function () {
3. 'use strict';
4. return function(value = 42) {
5. return value;
6. };
7. }());

JavaScript

6.name属性

函数的name属性可以返回这个函数的函数名（感觉并没有什么卵用）

在es5中匿名函数赋值给一个变量，然后查看这个变量的name会返回一个空字符串，但是es6中可以返回这个变量名。

2. var func1 = function () {};
4. *// ES5*
5. func1.name *// ""*
7. *// ES6*
8. func1.name *// "func1"*

JavaScript

如果将一个具体的命名函数赋值给一个变量，则es5和es6的name属性都会返回这个命名函数原本的名字。

2. const bar = function baz() {};
4. *// ES5*
5. bar.name *// "baz"*
7. *// ES6*
8. bar.name *// "baz"*

JavaScript

Function构造函数返回的函数实例，name属性的值为“anonymous”。

bind返回的函数，name属性值会加上“bound”前缀。

7.**箭头函数**

es6的函数正是因为箭头函数的存在，所以书写起来非常方便。

**es6允许我们使用“箭头” (=>)来定义函数。**

2. var f = v => v;
3. *// 等同于：*
4. var f = function(v) {
5. return v;
6. };

JavaScript

是不是非常方便！

箭头前面的是参数，后面的是返回值。

如果箭头函数不需要参数或者需要多个参数，就使用一个圆括号代表参数部分。

2. var f = () => 5;
3. *// 等同于*
4. var f = function () { return 5 };
6. var sum = (num1, num2) => num1 + num2;
7. *// 等同于*
8. var sum = function(num1, num2) {
9. return num1 + num2;
10. };

JavaScript

如果函数代码块部分不仅仅是一个返回语句，还有其他的很多条语句的话，就需要使用大括号将他们括起来，并且使用return语句来返回。

2. var f = (a, b, c) => {
3. let d = a + b;
4. return d \* c;
5. }

JavaScript

由于大括号会被解释成代码块，所以箭头函数如果要直接返回一个对象，必须在对象外面加上括号。

2. var getTempItem = id => ({ id: id, name: "ScarLett" });

JavaScript

箭头函数可以与变量解构赋值配合使用：

2. const full = ({ first, last }) => first + ' ' + last;
4. *// 等同于*
5. function full(person) {
6. return person.first + ' ' + person.last;
7. }

JavaScript

箭头函数还可以配合rest参数使用：

2. const numbers = (...nums) => nums;
3. numbers(1, 2, 3, 4, 5)
4. *// [1,2,3,4,5]*

JavaScript

使用箭头函数有几个需要注意的地方：

①.**函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象，这个是和es5很不一样的地方。**

②.不可以当做构造函数，也就是说，不能够使用new 命令，否则会报错。

③.不可以使用arguments对象，该对象在函数体内不存在。如果需要，可以使用rest参数来代替。

④.不可以使用yield命令，因此箭头函数不能用作Generator函数。

上面四点中，第一点尤其需要注意：es5中是谁调用函数，函数内部的this就指向谁，es6的箭头函数是函数在哪里声明，函数里的this就指向哪里。

2. var id = 'global';
4. var f = () => {
5. setTimeout( () => {
6. console.log(id);
7. } , 100)
9. }
11. f.call({"id": "local"});
12. *// local*

JavaScript

在正常的函数里面，应该是打印global的，因为在setTimeout执行的时候，实在全局环境下执行的，此时的this指向的是window，因此打印global。

但是在箭头函数里面，this的指向在f函数定义的时候就固定的指向了我们传入的那个对象，因此不论setTimeout

在哪里调用，都是打印的local。

箭头函数可以让this指向固定化，这种特性很有利于封装回调函数。

这里说明一下原理：

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。也正是因为箭头函数没有自己的this，所以它不能当做构造函数。

其实箭头函数放在正常的代码中应该是这个样子的：

2. function foo() {
3. var \_this = this;
4. setTimeout(function () {
5. console.log('id:', \_this.id);
6. }, 100);
7. }

JavaScript

由于箭头函数没有自己的this，所以当然也就不能使用call()、apply()、bind()这些改变this指向的方法。

除了this，一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。

Web前端的知识之旅哟——ES6-对象的扩展(四十八)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-16

* 围观(240)

* [评论(0)](http://www.flyingliu.com/2016/11/16/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-%e5%af%b9%e8%b1%a1%e7%9a%84%e6%89%a9%e5%b1%95%e5%9b%9b%e5%8d%81%e5%85%ab/#comments)

对象的扩展

1.属性的简洁表示法

es6中，如果属性名和属性值是一样的，那么可以只写属性名，不写属性值，这个时候属性值就是属性名所代表的变量。

2. function f(x, y) {
3. return {x, y};
4. }
6. *// 等同于*
8. function f(x, y) {
9. return {x: x, y: y};
10. }

JavaScript

2.属性名表达式

我们知道给对象添加属性有两种方式：一种是点的方式，另一种是方括号的方式。

2. *// 方法一*
3. obj.foo = true;
5. *// 方法二*
6. obj['a' + 'bc'] = 123;

JavaScript

在es5中，我们使用字面量方式定义对象，只有一种写法：

2. var obj = {
3. foo: true,
4. abc: 123
5. };

JavaScript

在es6中，允许字面量定义对象时，用方括号作为对象的属性名，然后里面写一个变量或者表达式：

2. let propKey = 'foo';
4. let obj = {
5. [propKey]: true,
6. ['a' + 'bc']: 123
7. };

JavaScript

表达式还可以用来定义方法名：

2. let obj = {
3. ['h'+'ello']() {
4. return 'hi';
5. }
6. };

JavaScript

注意，属性简洁表示法和属性名表达式不能同时使用，否则会报错。

3.Object.is

这个方法是比较两个值是否相等的。

在es5中，我们只有两个比较运算符：==和===

但是他们都有缺点：

==会自动进行类型转换，===的NaN不等于自身，以及+0等于-0.

在es6中，我们可以通过Object.is来进行比较，和严格相等的比较基本一样，但是他有两个不同之处：一个是+0不等于-0，一个是NaN等于自身。

2. +0 === -0 *//true*
3. NaN === NaN *// false*
5. Object.is(+0, -0) *// false*
6. Object.is(NaN, NaN) *// true*

JavaScript

4.**Object.assign**

这个方法是一个很重要的方法。

它用于对象的合并，将源对象的所有可枚举属性，复制到目标对象中。

Object.assgin方法的第一个参数是目标对象，后面的参数都是源对象。

2. var target = { a: 1 };
4. var source1 = { b: 2 };
5. var source2 = { c: 3 };
7. Object.assign(target, source1, source2);
8. target *// {a:1, b:2, c:3}e>*

JavaScript

如果有同名的属性，后面对象的属性会覆盖前面对象的属性。

如果只有一个参数，Object.assign会直接返回该参数。

2. var obj = {a: 1};
3. Object.assign(obj) === obj *// true*

JavaScript

如果该参数不是对象，则会先转换成对象，然后返回。

由于undefined和null无法转成对象，所以如果他们作为**目标对象参数**，就会报错。

但是如果**不是目标对象而是源对象参数**是null或者undefined，那么会直接跳过，并不会报错。

注意：Object.assign方法实行的是**浅层克隆**，并不是深层克隆。也就是说，如果对象的属性是一个引用值的话，那么目标对象拷贝得到的是这个引用值的引用。

用途：

①.为对象添加属性

②.为对象添加方法

③.克隆对象

④.合并多个对象

⑤.为属性指定默认值

5.属性的遍历

①.for…in

自身和继承的可枚举属性

②.Object.keys

自身的所有可枚举属性

③.Object.getOwnPropertyNames

自身的所有属性，包括可枚举的和不可枚举的

④.Object.getOwnPropertySymbols

自身的所有Symbol属性

⑤.Reflect.ownKeys

自身的所有属性，不管是属性名还是Symbol或者字符串，也不管是否可枚举

以上的5中遍历对象属性的方法，都是遵守同样的属性遍历的次序规则。

首先遍历所有属性名为数值的属性，按照数字排序。

其次遍历所有属性名为字符串的属性，按照生成时间排序。

最后遍历所有属性名为Symbol的属性，按照生成时间排序。

7.Object.values()、Object.entries()

Object.value()返回一个数组，是自身所有可遍历属性的键值。

Object.entries()返回一个数组，是自身所有可遍历属性的键值对。

8.对象的解构赋值

这还是es7的一个提案，不过babel转码器已经实现了该功能。

2. let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
3. x *// 1*
4. y *// 2*
5. z *// { a: 3, b: 4 }*

JavaScript

解构赋值不会拷贝继承自圆形对象的属性的。

9.对象配合扩展运算符

也可以实现和Object.assign一样的克隆功能

2. let z = { a: 3, b: 4 };
3. let n = { ...z };
4. n *// { a: 3, b: 4 }*

JavaScript

扩展运算符可以用于合并两个对象。

2. let ab = { ...a, ...b };
3. *// 等同于*
4. let ab = Object.assign({}, a, b);

JavaScript

如果用户自定义属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

Web前端的知识之旅哟——ES6-Set与Map结构(四十⑨)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-16

* 围观(230)

* [评论(0)](http://www.flyingliu.com/2016/11/16/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-set%e4%b8%8emap%e7%bb%93%e6%9e%84%e5%9b%9b%e5%8d%81%e2%91%a8/#comments)

Set和Map是es6新增的两个数据结构

Set

Set和数组很相似，但是它的成员的值都是唯一的，没有重复值。

2. var items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
3. items *// 1,2,3,4,5*

JavaScript

Set实例的方法分为了两大类：操作方法和遍历方法。

操作方法：

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。

遍历方法：

keys()：返回键名的遍历器

values()：返回键值的遍历器

entries()：返回键值对的遍历器

forEach()：使用回调函数遍历每个成员

2.WeakSet

WeakSet结构和Set很相似，主要有两个区别：

1.WeakSet的成员只能是对象，不能是其他类型的值。

2.WeakSet中的对象都是若引用，即垃圾回收机制不会考虑WeakSet对该对象的引用，也就是说，如果有其他对象都不在引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑它是否在WeakSet之中。这个特点意味着，无法引用WeakSet的成员，因此WeakSet是不可以遍历的。

Map

Map主要是解决传统es5中的键值对的键名只能是字符串的问题而引入的一种结构 。

在Map中，键值名不限于字符串，可以使各种类型的值。

2. var m = new Map();
3. var o = {p: 'Hello World'};
5. m.set(o, 'content')
6. m.get(o) *// "content"*
8. m.has(o) *// true*
9. m.delete(o) *// true*
10. m.has(o) *// false*

JavaScript

Map实例的属性：

1.size

size属性返回Map结构的成员总数。

2.set(key, value)

set方法设置key所对应的键值，然后返回整个Map结构。如果key已经存在了，那么键值会被更新，否则生成新的键值对。

set方法返回的是Map本身，因此可以使用链式写法：

2. let map = new Map()
3. .set(1, 'a')
4. .set(2, 'b')
5. .set(3, 'c');

JavaScript

3.get(key)

get方法读取key对应的键值，如果找不到则返回undefined。

4.has(key)

has方法返回一个布尔值，表示某个键是否在Map数据结构中。

5.delete(key)

delete方法删除某个键，返回true。如果删除失败，返回false。

6.clear()

clear方法清除所有成员，没有返回值

遍历方法

Map原生提供三个遍历器生成函数和一个遍历方法。

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。

forEach()：遍历Map的所有成员。

需要特别注意的是，Map的遍历顺序就是插入顺序。

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），不接受其他类型的值作为键名，而且键名所指向的对象，不计入垃圾回收 机制。

Web前端的知识之旅哟——ES6-Promise(五十)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-16

* 围观(363)

* [评论(0)](http://www.flyingliu.com/2016/11/16/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-promise%e4%ba%94%e5%8d%81/#comments)

Promise

**一、Promise含义**

Promise是异步编程的一种解决方案，所谓的Promise，简单来说就是一个容器，里面保存着未来才会结束的事件的结果。

Promise对象有以下两个特点：

1.对象的状态不受外界影响。Promise对象代表一种异步操作，有三种状态：Pending(进行中)、Resolved(已完成)、Rejected(已失败)。只有异步操作的结果可以改变状态，其他的任何操作都不能改变状态。

2.一旦状态改变了，就不会再变了，任何时候都可以得到这个结果。Promise对象的状态只有两种可能：Pending->Resolved或者Pending->Rejected。只要这两种情况发生了，状态就不会再改变了，并且会一直保持这个结果。这与事件监听不同，事件的特点是，不同时间监听，得到的结果都是不同的。

当然Promise也有一些缺点：

首先无法取消Promise，一旦创建他就会立即执行，中途无法取消。

其次，如果还不设置回调函数，Promise内部跑出的错误，不会反映到外部。

最后，当处于Pending状态时，无法得知目前进展到哪一个阶段了。

**二、基本用法**

Promise对象是一个构造函数，用来生成Promise实例的。

2. var promise = new Promise(function(resolve, reject) {
3. *// ... some code*
5. if (*/\* 异步操作成功 \*/*){
6. resolve(value);
7. } else {
8. reject(error);
9. }
10. });

JavaScript

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和reject。他们是两个函数，由js引擎提供，不需要自己部署。

resolve函数的作用是：

将Promise对象的状态从“未成功”变成“成功”（Pending->Resolved），在异步操作成功时调用，并将一步操作的结果，作为参数传递出去。

reject函数的作用是：

将Promise对象的状态从“未成功”变成“失败”（Pending->Rejected），在异步操作失败时候调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定Resolved状态和Reject状态的回调函数。

2. promise.then(function(value) {
3. *// success*
4. }, function(error) {
5. *// failure*
6. });

JavaScript

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为Resolved时调用，第二个回调函数是Promise对象的状态变成Rejected时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数。

这里我们就可以用Promise来异步加载图片：

2. function loadImageAsync(url) {
3. return new Promise(function(resolve, reject) {
4. var image = new Image();
6. image.onload = function() {
7. resolve(image);
8. };
10. image.onerror = function() {
11. reject(new Error('Could not load image at ' + url));
12. };
14. image.src = url;
15. });
16. }

JavaScript

只有图片加载成功之后才调用resolve函数。

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。

2. getJSON("/posts.json").then(function(json) {
3. return json.post;
4. }).then(function(post) {
5. *// ...*
6. });

JavaScript

这一段代码中使用了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中。

**三、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。

2. getJSON("/posts.json").then(function(posts) {
3. *// ...*
4. }).catch(function(error) {
5. *// 处理 getJSON 和 前一个回调函数运行时发生的错误*
6. console.log('发生错误！', error);
7. });

JavaScript

**四、Promise.all()**

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

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

JavaScript

当p1p2p3的状态都是resolved的时候，p的状态才是resolved，否则只要有一个是rejected，那么p的状态就时rejected。

**五、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，用法和all是一样的。

它与all的不同在于：race是只要有一个状态改变了，p的状态就会改变，并且变成和第一个状态改变之后的一样的状态。

**六、Promise.resolve()**

有时候需要将现有的对象转为Promise对象，Promise.resolve方法就起到这个作用。

2. Promise.resolve('foo')
3. *// 等价于*
4. new Promise(resolve => resolve('foo'))

JavaScript

Promise.resolve方法的参数分为四种情况：  
1.参数是一个Promise实例

如果参数是Promise实例，那么Promise.resolve将不做任何修改、原封不动地返 回这个实例。

2.参数是一个thenable对象

thenable对象指的是具有then方法的对象，比如下面这个对象。

2. let thenable = {
3. then: function(resolve, reject) {
4. resolve(42);
5. }
6. };

JavaScript

Promise.resolve方法会将这个对象转为Promise对象，然后立即执行thenable对象的then方法。

3.参数不是具有then方法的对象，或根本就不是对象

如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve 方法返回一个新的Promise对象，状态为Resolved。

2. var p = Promise.resolve('Hello');
4. p.then(function (s){
5. console.log(s)
6. });
7. *// Hello*

JavaScript

Promise.resolve方法允许调用时不带参数，直接返回一个Resolved状态的Promise对象。

所以，如果希望得到一个Promise对象，比较方便的方法就是直接调用Promise.resolve方法。

**七、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。它的参数用法与Promise.resolve方法完全一样。

**八、其他两个附加方法**

1.done()方法

Promise对象的回调链，不管以then方法或catch方法结尾，要是最后一个方法抛出错误，都有可能无法捕捉到（因为Promise内部的错误不会冒泡到全局）。因此，我们可以提供一个done方法，总是处于 回调链的尾端，保证抛出任何可能出现的错误。

2. asyncFunc()
3. .then(f1)
4. .catch(r1)
5. .then(f2)
6. .done();

JavaScript

2.finally()方法

finally方法用于指定不管Promise对象最后状态如何，都会执行的操作。它与done方法最大的区别就在于，它接受一个普通的回调函数作为参数，该参数不管怎样都必须执行。

类似try catch finally里面的finally。

Web前端的知识之旅哟——ES6-Class(五十一)

* 分类：[JAVASCRIPT](http://www.flyingliu.com/category/javascript/)

* 发表：2016-11-16

* 围观(300)

* [评论(0)](http://www.flyingliu.com/2016/11/16/web%e5%89%8d%e7%ab%af%e7%9a%84%e7%9f%a5%e8%af%86%e4%b9%8b%e6%97%85%e5%93%9f-es6-class%e4%ba%94%e5%8d%81%e4%b8%80/#comments)

这一篇就是ES6的最后一篇了，同时也是我认为ES6中最重要的改变之一。

Class

**一、class介绍**

传统js中，我们声明一个构造函数都是这样的：

2. function Point(x, y) {
3. this.x = x;
4. this.y = y;
5. }
7. Point.prototype.toString = function () {
8. return '(' + this.x + ', ' + this.y + ')';
9. };
11. var p = new Point(1, 2);

JavaScript

但是这样的形式和传统的面向对象语言（c++、java）区别很大，容易让新学习这门语言的程序员感到困惑，因此es6提供了更接近传统语言的写法，引入了class（类）的概念，作为对象的模板。

2. *//定义类*
3. class Point {
4. constructor(x, y) {
5. this.x = x;
6. this.y = y;
7. }
9. toString() {
10. return '(' + this.x + ', ' + this.y + ')';
11. }
12. }

JavaScript

这样就完成了对es5中构造函数的改造了。

构造函数的prototype属性，在es6中用Object.assign来写也会方便很多：

2. Object.assign(Point.prototype, {
3. toString(){},
4. toValue(){}
5. });

JavaScript

这里需要注意的是，用class来声明的类里面的方法，都是不可枚举的。

**1.constructor方法**

constructor方法是类的默认方法，通过new命令在生成对象实例 时，自动调用该方法。一个类必须有constructor方法，如果没有显示定义，一个空的constructor会被默认添加。

生成类的实例对象的写法，与ES5完全一样，也是使用new命令。如果忘记加上new，像函数那样调用class，将会报错。

2. *// 报错*
3. var point = Point(2, 3);
5. *// 正确*
6. var point = new Point(2, 3);

JavaScript

**2.不存在变量提升**

在class内部，不存在变量提升

**3.class表达式**

与函数一样，类也可以用表达式的形式定义。

2. const MyClass = class Me {
3. getClassName() {
4. return Me.name;
5. }
6. };

JavaScript

上面代码使用表达式定义了一个类。需要注意的是，这个类的名字是MyClass，而不是Me，Me只在class内部代码可用，指代当前类。

如果类的内部没有用到的话，可以省略Me，也就是可以写成我们的匿名函数表达式的形式。

采用class表达式，可以写出立即执行的class。

2. let person = new class {
3. constructor(name) {
4. this.name = name;
5. }
7. sayName() {
8. console.log(this.name);
9. }
10. }('张三');
12. person.sayName(); *// "张三"*

JavaScript

上面的代码中，person是一个立即执行的类的实例。

**4.this指向**

类的方法内部如果含有this，它默认指向类的实例，但是我们在使用的时候可能会出错，因为我们用了这个类里面的一个方法，但是this却不是指向的这个类而是指向了window，从而导致了报错。

一个比较简单的解决办法是，在构造函数中绑定this，这样就不会找不到方法了。

2. class Logger {
3. constructor() {
4. this.printName = this.printName.bind(this);
5. }
7. *// ...*
8. }

JavaScript

上面这一段代码中，如果我们没有绑定this的话，我们直接调用Logger的printName方法，由于this指向了window，会报出找不到printName的错误。

另一种解决办法是使用箭头函数：

2. class Logger {
3. constructor() {
4. this.printName = (name = 'there') => {
5. this.print(`Hello ${name}`);
6. };
7. }
9. *// ...*
10. }

JavaScript

在前面的箭头函数中我们可以知道，箭头函数内部的this在函数定义的同时就确定了，因此也就不存在this的指向问题了。

**二、class的继承**

class之间可以通过extends关键字实现继承，这比es5的通过原型链的继承要清晰和方便很多。

2. class son extends father {}

JavaScript

这里的son类就是通过extends继承了father类的所有属性和方法。

我们可以在子类里面通过super方法调用父类的constructor方法。

2. class son extends father {
3. constructor(x, y, color) {
4. super(x, y); *// 调用父类的constructor(x, y)*
5. this.color = color;
6. }
8. toString() {
9. return this.color + ' ' + super.toString(); *// 调用父类的toString()*
10. }
11. }

JavaScript

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

所以这里必须要注意：只有调用了super方法之后，才可以使用this关键字。

类的prototype属性和\_\_prototype\_\_属性

1.子类的\_\_proto\_\_属性，表示构造函数的继承，总是指向父类。

2.子类prototype属性的\_\_proto\_\_属性，表示方法的继承，总是指向父类的  
prototype属性。

Extends的继承目标

除了继承父类之外，还有三种特殊情况

1.子类继承Object类。

class A extends Object {}

这种情况下，A其实就是构造函数Object的复制，A的实例就是Object的实例。

2.不存在任何继承

class A {}

这种情况下，A作为一个基类，就是一个普通的函数，所以直接继承Function.prototype。但是，A调用后返回一个空对象，所以A.prototype.\_\_proto\_\_指向构造函数Object的prototype属性，而不是A的prototype。

3.子类继承null

class A extends null {}

这种情况和第二种情况很像，A也是一个普通函数，所以直接继承Function.prototype。但是A调用后返回的对象不继承任何方法，所以它的\_\_proto\_\_指向Function.prototype。

Object.getPrototypeOf()

Object.getPrototypeOf方法可以用来用来从子类上获取父类。

super关键字

super这个关键字，有两种用法，含义不同。

1.作为函数调用时（即super(…args)），super代表父类的构造函数。

2.作为对象调用时（即super.prop或super.method()），super代表父类。注意，此时super既可以引用父类实例的属性和方法，也可以引用父类的静态方法。

**三、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”。

2. class Foo {
3. static classMethod() {
4. return 'hello';
5. }
6. }
8. Foo.classMethod() *// 'hello'*

JavaScript

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。

2. class Foo {
3. static classMethod() {
4. return 'hello';
5. }
6. }
8. class Bar extends Foo {
9. }
10. Bar.classMethod(); *// 'hello'*

JavaScript

静态方法也是可以从super对象上调用的。

2. class Foo {
3. static classMethod() {
4. return 'hello';
5. }
6. }
8. class Bar extends Foo {
9. static classMethod() {
10. return super.classMethod() + ', too';
11. }
12. }
14. Bar.classMethod();