概念

ECMAScript

1996 年 11 月，JavaScript 的创造者将 JavaScript 提交给国际标准化组织 ECMA，希望这种语言能够成为国际标准。

次年，ECMA 发布 262 号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript，这个版本就是 1.0 版。

ECMAScript 和 JavaScript：前者是后者的规范，后者是前者的一种实现。

ES6 (ECMAScript 6)

2015 年 6 月正式发布 ECMAScript 6.0 版本，命名为《ECMAScript 2015 标准》(简称 ES2015)。之后每年的 6 月份发布一个更新版本(6.1 版,6.2 版……)，

ES6 泛指 ES2015、ES2016 这些新版本。

Babel 转码器

虽然各大浏览器的最新版本对 ES6 的支持度越来越高，但考虑到现有环境对 ES6 的支持有限，我们可以用 ES6 的方式编写程序，再用转码器将 ES6 的语法转为 ES5 去执行。Babel 就是这样一个被广泛使用的 ES6 转码器(ES6 -> ES5)。

1. **配置文件.babelrc(存放至项目根目录下)**

```
{
  "presets": [es2015],
  "plugins": []
}
```

安装 ES5 转码规则

```
$ npm install --save-dev babel-preset-es2015
```

2. **命令行转码 babel-cli**

安装 babel-cli 工具

```
$ npm install --global babel-cli
```

转码

```
$ babel src -d lib
```

3. **浏览器环境**

从 Babel 6.0 开始，不再直接提供浏览器版本，而是要用构建工具构建出来。如果你没有或不想使用构建工具，可以通过安装 5.x 版本的 babel-core 模块获取。

```
$ npm install babel-core@5
```

在网页中调用

```
<script src="node_modules/babel-core/browser.js"></script>
```

目录

---

let 和 const

let

JavaScript 没有块级作用域的概念.。
JavaScript 的作用域：全局，局部(函数块)

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

1. let 命令用法类似于 var，但所声明的变量只在 let 所在的代码块中有效 (let 实际上为 JavaScript 新增了块级作用域)。

2. 同一作用域，不允许重复声明，
   但内层作用域可以覆盖外层同名定义。

```
{
   let a = 10;
```

```
  {
    let a = 20;
  }
}
```

3. 顶层对象的属性:

```
var a = 10;
let b = 10;

window.a      // 10
window.b      // undefined
```

原本，全局变量等同于顶层对象的属性。

ES6，let、const、class 声明的全局变量，不属于顶层对象的属性。从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。

const

```
Const PI = 3.1415926;
```

1. 声明一个只读的常量。一旦声明，常量的值就不能改变。
2. 作用域与 let 命令相同：只在声明所在的块级作用域内有效。

注意

新的规范下，我们基本使用 let 取代 var 来声明变量，用 const 来声明只读的常量。

变量的解构赋值

从数组和对象中提取值，对变量进行赋值

数组

1. **基本用法**

```
let [ , , , a, b] = [5, 6, 7, 8];     //a = 8, b = undefined

let [c, [d]] = [1, 2];      //报错
```

数组解构赋值时，等号前后结构必须一致

2. **默认值**

```
let [x, y = 'b'] = ['a'];    // x='a', y='b'
```

3. **注意**

ES6 内部使用严格相等运算符（===），判断一个位置是否有值。所以，如果一个数组成员不严格等于 undefined，默认值是不会生效的。

```
var [x = 1] = [undefined];
// x = 1

var [x = 1] = [null];
// x = null
```

对象

1. **基本用法**

```
let { first, insert, second} = { first: [], second: {}}
//first = [], insert = undefined, second = {}
```

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。

如果变量名与属性名不一致，必须写成下面这样：

```
var { foo: baz } = { foo: 'aaa', bar: 'bbb' };
// baz : "aaa"
```

2. **默认值**

默认值生效的条件是，对象的属性值严格等于 undefined。

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

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

3. **注意**

对已经声明的变量用于解构赋值：

当 let { x } = { x: 1} 时，是重新声明了 x 变量，并进行赋值。

错误的写法：{x} = {x: 1};

JS 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：({x} = {x: 1});

模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的。

```
$('#result').append(
    'There are <b>' + basket.count + '</b> '
    + 'items in your basket, ' + '<em>' +
    basket.onSale + '</em> are on sale!'
);
```

ES6：

```
$('#result').append(`
    There are <b>${basket.count}</b> items
    in your basket, <em>${basket.onSale}</em>
    are on sale!
`);
```

模板字符串（template string）是增强版的字符串。

1. **用反引号 ` 标识**
   字符串中需要使用反引号，则前面要用反斜杠转义。
2. **可以用来定义多行字符串**
   所有的空格和缩进都会被保留在输出之中。
3. **${变量}**
   可以在字符串中嵌入变量，括号内部可以放入任意的 JavaScript 表达式，可以进行运算，引用对象属性，以及调用函数。

数组的扩展

Array.from()

将类数组对象转为真正的数组

```
let arrayLike = {
   '0': 'a',
   '1': 'b',
   '2': 'c',
   length: 3
};

// ES5 的写法
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']
```

```
// ES6 的写法
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

实际应用中，常见的类似数组的对象是 DOM 操作返回的 NodeList 集合，以及函数内部的 arguments 对象。Array.from 都可以将它们转为真正的数组。

Array.from()实际有三个参数：

a. 类数组对象

b. map 函数：对每个元素进行处理，将处理后的值放入返回的数组

c. 来绑定 map 函数中的 this 关键字

```
Array.from(arrayLike, function(item, index){
  //对每个元素进行处理
}, this)
```

数组的空位

空位不是 undefined

[, , ,]该数组中有 4 个空位，与[undefined, undefined, undefined, undefined]不同。

1. **ES5 对空位的处理**
   - forEach(), filter(), every() 和 some()都会跳过空位。
   - map()会跳过空位，但会保留这个值。
   - join()和 toString()会将空位视为 undefined，而 undefined 和 null 会被处理成空字符串。

2. **ES6 对空位的处理**

   明确将空位转为 undefined。

   如：Array.from 方法会将数组的空位，转为 undefined，也就是说，这个方法不会忽略空位。

```
Array.from(['a',,'b'])        // [ "a", undefined, "b" ]

['a',,'b'].map(item => item)        //[ "a", , "b" ]
```

函数的扩展

函数参数的默认值

1. **基本用法**

```
function func(x = 0, y = 0) {
   console.log('x=',x,'y=',y);
}
func(5); //x= 5 y= 0
```

2. 结合解构赋值

函数参数的解构赋值其实就是对数组、对象解构赋值的应用。

```
function add([x, y], {i, j = '默认值'}) {
   //x = 1, y = [3, 4]
   //i = {}, j = '默认值'
}
add([1, [3, 4] ], { i: {}, y: 5})
```

rest 参数

```
function add(param1, param2, ...values) {
   //param1 = 1,
   //param2 = 2,
   //param1 = [3, 4, 5]
}
add(1,2,3,4,5);
```

- 将多余的参数放入数组 values 中。
- rest 参数必须是最后一个参数。
- 用剩余参数代替 arguments。
- 仅用于形参中！

扩展运算符... (spread)

相当于 rest 参数的逆运算，将数组转为逗号分隔的参数序列。

```
console.log(1, ...[2, 3, 4], 5)
// 1 2 3 4 5
```

可代替数组的 apply 方法将数组拆分传入：

```
// ES5 的写法
function f(x, y, z) {
   // ...
}
var args = [0, 1, 2];
f.apply(null, args);
```

```
// ES6 的写法
function f(x, y, z) {
  // ...
}
var args = [0, 1, 2];
f(...args);
```

严格模式

当函数参数使用默认值 or 解构赋值 or rest 参数 or 扩展运算符时，函数内部就不能'use strict'，否则会报错。（原因：执行顺序）

解决方式：

- 全局使用'use strict'
- 利用立即执行函数：

```
const func = (
  'use strict'
  return function(value = 42){
    //内部函数可使用默认值 or 解构赋值 or rest 参数 or 扩展运算符
  }
)()
```

箭头函数 () => {}

1. **基本用法**

```
var func = v => y //  等同于 var func = function(v) { return v; }
```

多于一个参数 或 没有参数时，要用`()`将参数包裹起来。

代码块部分多余一条语句时，要用`{}`将代码块包裹起来。

```
var func = (a, b) => { a += 1, b += 2; return a+b; }
```

2. **注意点**

this 对象的指向是可变的，但是在箭头函数中，它是固定的。

this 指向固定化，实际原因是箭头函数没有自己的 this，故内部的 this 就是外层代码的 this。

箭头函数可以让 setTimeout 里面的 this，绑定**定义时**所在的作用域，而不是指向运行时所在的作用域。

```
function Timer() {
  this.s1 = 0;
  this.s2 = 0;
  setInterval(() => this.s1++, 1000);
  //  箭头函数

  setInterval(function () { this.s2++;   }, 1000);
```

```
  // 普通函数 s2 指向全局的变量 s2
}
var timer = new Timer();
setTimeout(() => console.log('s1: ', timer.s1), 3100); // s1: 3
setTimeout(() => console.log('s2: ', timer.s2), 3100); // s2: 0
```

---

class

JavaScript 语言的传统方法是通过构造函数，定义并生成新对象。

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
};

var p = new Point(1, 2);
```

ES6 提供了更接近传统语言(C++,Java)的写法，引入了 Class（类）这个概念，作为对象的模板。

1. **ES5 的构造函数 Point，对应 ES6 的 Point 类的构造方法(constructor)。**
2. **class 的方法(toString)。**

   定义“类”的方法的时候，前面不需要加上 function 这个关键字。

   方法之间不需要逗号分隔，加了会报错。

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

var p = new Point(1, 2);
```

继承

```
class ColorPoint extends Point {
  //ColorPoint 类通过 extends 关键字，继承了 Point 类的所有属性和方法
  constructor(x, y, color) {
    this.color = color;        // 报错，此时子类还未继承父类的 this 对象

    super(x, y);         // 通过 super(x,y)调用父类的 constructor(x, y)

    this.color = color;       //正确
  }

  toString() {
    return this.color + ' ' + super.toString(); // 调用父类的 toString()
  }
}
```

注意：

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。

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'
```

父类的静态方法，可以被子类继承。

Module

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。

模块功能主要由两个命令构成：export 和 import。

1. **输出：**

   如果希望外部能够读取模块内部的某个变量，就必须使用 export 关键字暴露该接口。

```
export function multiply(x, y) {
  return x 1.y;
```

```
};

export function plus(x, y) {
    return x + y;
};

export default function subtractive(x, y) {
    // export default  默认输出，函数名无效
    return x - y;
};
```

2. **引入：**

import 引入其他模块提供的功能。

```
import calc from './calculation' //  导入默认的输出
import {multiply, plus} //  导入 multiply, plus
```

总结

ES6 的支持度目前越来越高，内容也在逐步完善中。
以上内容只是 ES6 中，非常基础、非常小的一部分，有什么问题欢迎指出和探讨。
同时建议学习的同学，还是要抽出时间系统地掌握 ES6 的语法。