ES6新特性

ECMAScript 包含了什么

ECMA (一个类似 W3C 的标准化组织) 是 JavaScript 语言的标准化组织。JavaScript 正是基于 ECMAScript 标准的实现。ECMAScript 定义了：
语法 - 解析规则、关键词、语句、声明、操作符等。
类型 - boolean, number, string, object 等。
原型与继承
内建对象和函数的标准库 - JSON, Math, 数组方法以及对象的自省方法等。
ECMScript 并没有定义任何与 HTML，CSS 或者 Web APIs 相关的内容，例如 DOM (Document Object Model)。这些内容都由其他独立的规范定义。ECMAScript 囊括了 JavaScript 语言的所有方面，不仅仅局限于浏览器，也包含了非浏览器环境，例如 Node.js。
新标准
ES6 是对语言的一次重大升级。同时，现有的 JavaScript 代码仍能继续运行。ES6 在设计的时候就保证了与现有代码的最大兼容性。事实上，很多浏览器已经支持了一些 ES6的特性，并不断努力实现剩余的部分。这就意味着包含 ES6 特性的 JavaScript 代码已经可以在这些实现了 ES6 特性的浏览器中运行了！

ES6新特性

- 变量声明let与const

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;
}

a // ReferenceError: a is not defined.
b // 1
```

上面代码在代码块之中，分别用`let`和`var`声明了两个变量。然后在代码块之外调用这两个变量，结果`let`声明的变量报错，`var`声明的变量返回了正确的值。这表明，`let`声明的变量只在它所在的代码块有效。

`for`循环的计数器，就很合适使用`let`命令。

```
for (let i = 0; i < 10; i++) {
  // ...
}

console.log(i);
// ReferenceError: i is not defined
```

上面代码中，计数器`i`只在`for`循环体内有效，在循环体外引用就会报错。

下面的代码如果使用`var`，最后输出的是`10`。

```
var a = [];
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 10
```

上面代码中，变量`i`是`var`命令声明的，在全局范围内都有效，所以全局只有一个变量`i`。每一次循环，变量`i`的值都会发生改变，而循环内被赋给数组`a`的函数内部的`console.log(i)`，里面的`i`指向的就是全局的`i`。也就是说，所有数组`a`的成员里面的`i`，指向的都是同一个`i`，导致运行时输出的是最后一轮的`i`的值，也就是 10。

如果使用`let`，声明的变量仅在块级作用域内有效，最后输出的是 6。

```
var a = [];
for (let i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 6
```

上面代码中，变量`i`是`let`声明的，当前的`i`只在本轮循环有效，所以每一次循环的`i`其实都是一个新的变量，所以最后输出的是`6`。你可能会问，如果每一轮循环的变量`i`都是重新声明的，那它怎么知道上一轮循环的值，从而计算出本轮循环的值？这是因为 JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量`i`时，就在上一轮循环的基础上进行计算。

另外，`for`循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
for (let i = 0; i < 3; i++) {
  let i = 'abc';
  console.log(i);
}
// abc
// abc
// abc
```

上面代码正确运行，输出了 3 次`abc`。这表明函数内部的变量`i`与循环变量`i`不在同一个作用域，有各自单独的作用域（同一个作用域不可使用`let`重复声明同一个变量）。

不存在变量提升

`var`命令会发生“变量提升”现象，即变量可以在声明之前使用，值为`undefined`。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，`let`命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

```
// var 的情况
console.log(foo); // 输出undefined
var foo = 2;

// let 的情况
console.log(bar); // 报错ReferenceError
let bar = 2;
```

上面代码中，变量`foo`用`var`命令声明，会发生变量提升，即脚本开始运行时，变量`foo`已经存在了，但是没有值，所以会输出`undefined`。变量`bar`用`let`命令声明，不会发生变量提升。这表示在声明它之前，变量`bar`是不存在的，这时如果用到它，就会抛出一个错误。

暂时性死区

只要块级作用域内存在`let`命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

```
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变量，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）。

```
if (true) {
  // TDZ开始
  tmp = 'abc'; // ReferenceError
  console.log(tmp); // ReferenceError

  let tmp; // TDZ结束
  console.log(tmp); // undefined

  tmp = 123;
  console.log(tmp); // 123
}
```

上面代码中，在`let`命令声明变量`tmp`之前，都属于变量`tmp`的“死区”。

“暂时性死区”也意味着`typeof`不再是一个百分之百安全的操作。

```
typeof x; // ReferenceError
let x;
```

上面代码中，变量`x`使用`let`命令声明，所以在声明之前，都属于`x`的“死区”，只要用到该变量就会报错。因此，`typeof`运行时就会抛出一个`ReferenceError`。

作为比较，如果一个变量根本没有被声明，使用`typeof`反而不会报错。

```
typeof undeclared_variable // "undefined"
```

上面代码中，`undeclared_variable`是一个不存在的变量名，结果返回“undefined”。所以，在没有`let`之前，`typeof`运算符是百分之百安全的，永远不会报错。现在这一点不成立了。这样的设计是为了让大家养成良好的编程习惯，变量一定要在声明之后使用，否则就报错。

有些“死区”比较隐蔽，不太容易发现。

```
function bar(x = y, y = 2) {
  return [x, y];
}

bar(); // 报错
```

上面代码中，调用`bar`函数之所以报错（某些实现可能不报错），是因为参数`x`默认值等于另一个参数`y`，而此时`y`还没有声明，属于“死区”。如果`y`的默认值是`x`，就不会报错，因为此时`x`已经声明了。

```
function bar(x = 2, y = x) {
  return [x, y];
}
bar(); // [2, 2]
```

另外，下面的代码也会报错，与`var`的行为不同。

```
// 不报错
var x = x;

// 报错
let x = x;
// ReferenceError: x is not defined
```

上面代码报错，也是因为暂时性死区。使用`let`声明变量时，只要变量在还没有声明完成前使用，就会报错。上面这行就属于这个情况，在变量`x`的声明语句还没有执行完成前，就去取`x`的值，导致报错”x 未定义“。

ES6 规定暂时性死区和`let`、`const`语句不出现变量提升，主要是为了减少运行时错误，防止在变量声明前就使用这个变量，从而导致意料之外的行为。这样的错误在 ES5 是很常见的，现在有了这种规定，避免此类错误就很容易了。

总之，暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，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。

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。

```
// 报错
function func() {
  let a = 10;
  var a = 1;
}

// 报错
function func() {
  let a = 10;
  let a = 1;
}
```

因此，不能在函数内部重新声明参数。

```
function func(arg) {
  let arg;
}
func() // 报错

function func(arg) {
  {
    let arg;
  }
}
func() // 不报错
```

块级作用域

为什么需要块级作用域?

ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。

第一种场景，内层变量可能会覆盖外层变量。

```
var tmp = new Date();

function f() {
  console.log(tmp);
  if (false) {
    var tmp = 'hello world';
  }
}

f(); // undefined
```

上面代码的原意是，`if`代码块的外部使用外层的`tmp`变量，内部使用内层的`tmp`变量。但是，函数`f`执行后，输出结果为`undefined`，原因在于变量提升，导致内层的`tmp`变量覆盖了外层的`tmp`变量。

第二种场景，用来计数的循环变量泄露为全局变量。

```
var s = 'hello';

for (var i = 0; i < s.length; i++) {
  console.log(s[i]);
}

console.log(i); // 5
```

上面代码中，变量`i`只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

ES6 的块级作用域

`let`实际上为 JavaScript 新增了块级作用域。

```
function f1() {
  let n = 5;
  if (true) {
    let n = 10;
  }
  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。如果两次都使用`var`定义变量`n`，最后输出的值才是 10。

ES6 允许块级作用域的任意嵌套。

```
{{{{
  {let insane = 'Hello World'}
  console.log(insane); // 报错
}}}};
```

上面代码使用了一个五层的块级作用域，每一层都是一个单独的作用域。第四层作用域无法读取第五层作用域的内部变量。

内层作用域可以定义外层作用域的同名变量。

```
{{{{
  let insane = 'Hello World';
  {let insane = 'Hello World'}
}}}};
```

块级作用域的出现，实际上使得获得广泛应用的匿名立即执行函数表达式（匿名 IIFE）不再必要了。

```
// IIFE 写法
(function () {
  var tmp = ...;
  ...
}());

// 块级作用域写法
{
  let tmp = ...;
  ...
}
```

const 命令

基本用法

`const`声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const PI = 3.1415;
PI // 3.1415

PI = 3;
// TypeError: Assignment to constant variable.
```

上面代码表明改变常量的值会报错。

`const`声明的变量不得改变值，这意味着，`const`一旦声明变量，就必须立即初始化，不能留到以后赋值。

```
const foo;
// SyntaxError: Missing initializer in const declaration
```

上面代码表示，对于`const`来说，只声明不赋值，就会报错。

`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。

```
if (true) {
  const MAX = 5;
}

MAX // Uncaught ReferenceError: MAX is not defined
```

`const`命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

```
if (true) {
  console.log(MAX); // ReferenceError
  const MAX = 5;
}
```

上面代码在常量`MAX`声明之前就调用，结果报错。

`const`声明的常量，也与`let`一样不可重复声明。

```
var message = "Hello!";
let age = 25;

// 以下两行都会报错
const message = "Goodbye!";
const age = 30;
```

- 变量的解构赋值
  - 数组解构赋值
  - 对象解构赋值
  - 字符串解构赋值

数组的解构赋值

基本用法

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）。

以前，为变量赋值，只能直接指定值。

```
let a = 1;
let b = 2;
let c = 3;
```

ES6 允许写成下面这样。

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

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
foo // 1
bar // 2
baz // 3

let [ , , third] = ["foo", "bar", "baz"];
third // "baz"

let [x, , y] = [1, 2, 3];
x // 1
y // 3

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

let [x, y, ...z] = ['a'];
x // "a"
y // undefined
z // []
```

如果解构不成功，变量的值就等于`undefined`。

```
let [foo] = [];
let [bar, foo] = [1];
```

以上两种情况都属于解构不成功，`foo`的值都会等于`undefined`。

另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

```
let [x, y] = [1, 2, 3];
x // 1
y // 2

let [a, [b], d] = [1, [2, 3], 4];
a // 1
b // 2
d // 4
```

上面两个例子，都属于不完全解构，但是可以成功。

如果等号的右边不是数组（或者严格地说，不是可遍历的结构，参见《Iterator》一章），那么将会报错。

```
// 报错
let [foo] = 1;
let [foo] = false;
let [foo] = NaN;
let [foo] = undefined;
let [foo] = null;
let [foo] = {};
```

上面的语句都会报错，因为等号右边的值，要么转为对象以后不具备 Iterator 接口（前五个表达式），要么本身就不具备 Iterator 接口（最后一个表达式）。

对于 Set 结构，也可以使用数组的解构赋值。

```
let [x, y, z] = new Set(['a', 'b', 'c']);
x // "a"
```

事实上，只要某种数据结构具有 Iterator 接口，都可以采用数组形式的解构赋值。

```
function* fibs() {
  let a = 0;
  let b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

let [first, second, third, fourth, fifth, sixth] = fibs();
sixth // 5
```

上面代码中，`fibs`是一个 Generator 函数（参见《Generator 函数》一章），原生具有 Iterator 接口。解构赋值会依次从这个接口获取值。

对象的解构赋值

简介

解构不仅可以用于数组，还可以用于对象。

```
let { foo, bar } = { foo: 'aaa', bar: 'bbb' };
foo // "aaa"
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。

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

let { baz } = { foo: 'aaa', bar: 'bbb' };
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`。

如果解构失败，变量的值等于`undefined`。

```
let {foo} = {bar: 'baz'};
foo // undefined
```

上面代码中，等号右边的对象没有`foo`属性，所以变量`foo`取不到值，所以等于`undefined`。

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。

```
// 例一
let { log, sin, cos } = Math;

// 例二
const { log } = console;
log('hello') // hello
```

上面代码的例一将`Math`对象的对数、正弦、余弦三个方法，赋值到对应的变量上，使用起来就会方便很多。例二将`console.log`赋值到`log`变量。

如果变量名与属性名不一致，必须写成下面这样。

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

let obj = { first: 'hello', last: 'world' };
let { first: f, last: l } = obj;
f // 'hello'
l // 'world'
```

这实际上说明，对象的解构赋值是下面形式的简写（参见《对象的扩展》一章）。

```
let { foo: foo, bar: bar } = { foo: 'aaa', bar: 'bbb' };
```

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

```
let { foo: baz } = { foo: 'aaa', bar: 'bbb' };
baz // "aaa"
foo // error: foo is not defined
```

上面代码中，`foo`是匹配的模式，`baz`才是变量。真正被赋值的是变量`baz`，而不是模式`foo`。

与数组一样，解构也可以用于嵌套结构的对象。

```
let obj = {
  p: [
    'Hello',
    { y: 'World' }
  ]
};

let { p: [x, { y }] } = obj;
x // "Hello"
y // "World"
```

注意，这时`p`是模式，不是变量，因此不会被赋值。如果`p`也要作为变量赋值，可以写成下面这样。

```
let obj = {
  p: [
    'Hello',
    { y: 'World' }
  ]
};

let { p, p: [x, { y }] } = obj;
x // "Hello"
y // "World"
p // ["Hello", {y: "World"}]
```

下面是另一个例子。

```
const node = {
  loc: {
    start: {
      line: 1,
      column: 5
    }
  }
};

let { loc, loc: { start }, loc: { start: { line }} } = node;
line // 1
loc  // Object {start: Object}
start // Object {line: 1, column: 5}
```

上面代码有三次解构赋值，分别是对`loc`、`start`、`line`三个属性的解构赋值。注意，最后一次对`line`属性的解构赋值之中，只有`line`是变量，`loc`和`start`都是模式，不是变量。

下面是嵌套赋值的例子。

```
let obj = {};
let arr = [];

({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });

obj // {prop:123}
arr // [true]
```

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
let {foo: {bar}} = {baz: 'baz'};
```

上面代码中，等号左边对象的`foo`属性，对应一个子对象。该子对象的`bar`属性，解构时会报错。原因很简单，因为`foo`这时等于`undefined`，再取子属性就会报错。

注意，对象的解构赋值可以取到继承的属性。

```
const obj1 = {};
const obj2 = { foo: 'bar' };
Object.setPrototypeOf(obj1, obj2);

const { foo } = obj1;
foo // "bar"
```

上面代码中，对象`obj1`的原型对象是`obj2`。`foo`属性不是`obj1`自身的属性，而是继承自`obj2`的属性，解构赋值可以取到这个属性。

字符串的解构赋值

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

```
const [a, b, c, d, e] = 'hello';
a // "h"
b // "e"
c // "l"
d // "l"
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解构赋值。

```
let {length : len} = 'hello';
len // 5
```

- 字符串扩展
  - includes()　判断字符串中是否包含指定的字符（有：true，无：false）
  - startsWith() 判断字符串是否以特定的字符开始（有：true，无：false）
  - endsWith()　判断字符串是否以特定的字符结束（有：true，无：false）
  - 模板字符串　``反引号 表示模板，模板中的内容可以有格式，通过${}填充数据
  - ## 模板字符串

    传统的 JavaScript 语言，输出模板通常是这样写的（下面使用了 jQuery 的方法）。

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

    上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

    ```
    $('#result').append(`
      There are <b>${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，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。

```
// 普通字符串
`In JavaScript '\n' is a line-feed.`

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

console.log(`string text line 1
string text line 2`);

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

上面代码中的模板字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

```
let greeting = `\`Yo\` World!`;
```

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

```
$('#list').html(`
<ul>
  <li>first</li>
  <li>second</li>
</ul>
`);
```

上面代码中，所有模板字符串的空格和换行，都是被保留的，比如`<ul>`标签前面会有一个换行。如果你不想要这个换行，可以使用`trim`方法消除它。

```
$('#list').html(`
<ul>
  <li>first</li>
  <li>second</li>
</ul>
`.trim());
```

模板字符串中嵌入变量，需要将变量名写在`${}`之中。

- 函数扩展
  - 参数默认值

函数参数的默认值

基本用法

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

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

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

上面代码检查函数`log`的参数`y`有没有赋值，如果没有，则指定默认值为`World`。这种写法的缺点在于，如果参数`y`赋值了，但是对应的布尔值为`false`，则该赋值不起作用。就像上面代码的最后一行，参数`y`等于空字符，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数`y`是否被赋值，如果没有，再等于默认值。

```
if (typeof y === 'undefined') {
  y = 'World';
}
```

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

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

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

可以看到，ES6 的写法比 ES5 简洁许多，而且非常自然。下面是另一个例子。

```
function Point(x = 0, y = 0) {
  this.x = x;
  this.y = y;
}

const p = new Point();
p // { x: 0, y: 0 }
```

- 参数解构赋值

与解构赋值默认值结合使用

参数默认值可以与解构赋值的默认值，结合起来使用。

```
function foo({x, y = 5}) {
  console.log(x, y);
}

foo({}) // undefined 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
foo() // TypeError: Cannot read property 'x' of undefined
```

上面代码只使用了对象的解构赋值默认值，没有使用函数参数的默认值。只有当函数`foo`的参数是一个对象时，变量`x`和`y`才会通过解构赋值生成。如果函数`foo`调用时没提供参数，变量`x`和`y`就不会生成，从而报错。通过提供函数参数的默认值，就可以避免这种情况。

- rest函数

rest 参数

ES6 引入 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。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。

```
function add(...values) {
  let sum = 0;

  for (var val of values) {
    sum += val;
  }

  return sum;
}

add(2, 5, 3) // 10
```

上面代码的`add`函数是一个求和函数，利用 rest 参数，可以向该函数传入任意数目的参数。

- 扩展运算符

```
//扩展运算符 ...
    function test5(a, b, c, d) {
        console.log(a + b + c + d);
    }
    // test5(1,2,3,4);
    let arr = [1, 2, 3, 4];
    //test5.apply(null,arr); //apply方法重用 第一个参数是对象，第二个参数是数组
    test5(...arr);
```

- 箭头函数

箭头函数

基本用法

ES6 允许使用“箭头”（`=>`）定义函数。

```
var f = v => v;

// 等同于
var f = function (v) {
  return v;
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

```
var f = () => 5;
// 等同于
var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;
// 等同于
var sum = function(num1, num2) {
  return num1 + num2;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用`return`语句返回。

```
var sum = (num1, num2) => { return num1 + num2; }
```

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错。

```
// 报错
let getTempItem = id => { id: id, name: "Temp" };

// 不报错
let getTempItem = id => ({ id: id, name: "Temp" });
```

下面是一种特殊情况，虽然可以运行，但会得到错误的结果。

```
let foo = () => { a: 1 };
foo() // undefined
```

上面代码中，原始意图是返回一个对象`{ a: 1 }`，但是由于引擎认为大括号是代码块，所以执行了一行语句`a: 1`。这时，`a`可以被解释为语句的标签，因此实际执行的语句是`1;`，然后函数就结束了，没有返回值。

如果箭头函数只有一行语句，且不需要返回值，可以采用下面的写法，就不用写大括号了。

- for of 循环允许你遍历 Arrays（数组）, Strings（字符串）, Sets（集合）等可迭代的数据结构等

```
const iterable = ['mini', 'mani', 'mo'];

    for (const value of iterable) {
        console.log(value);
    }

    var str = "hello";
    for (const value of str) {
        console.log(value);
    }
    // 不可以遍历对象
    // var obj = {
    //     name: "张三",
    //     sex: "男"
    // }
    // for (const value of obj) {
    //     console.log(value);
    // }
```

- Set集合

Set

基本用法

ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

`Set`本身是一个构造函数，用来生成 Set 数据结构。

```
const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

for (let i of s) {
  console.log(i);
}
// 2 3 5 4
```

上面代码通过`add()`方法向 Set 结构加入成员，结果表明 Set 结构不会添加重复的值。

`Set`函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化。

```
// 例一
const set = new Set([1, 2, 3, 4, 4]);
[...set]
// [1, 2, 3, 4]

// 例二
const items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
items.size // 5

// 例三
const set = new Set(document.querySelectorAll('div'));
set.size // 56

// 类似于
const set = new Set();
document
 .querySelectorAll('div')
 .forEach(div => set.add(div));
set.size // 56
```

上面代码中，例一和例二都是`Set`函数接受数组作为参数，例三是接受类似数组的对象作为参数。

- 类与继承

类的由来

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。

基本上，ES6 的`class`可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的`class`写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用 ES6 的`class`改写，就是下面这样。

```
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。

类的继承

Class 可以通过`extends`关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

```
class Point {
}

class ColorPoint extends Point {
}
```

上面代码定义了一个`ColorPoint`类，该类通过`extends`关键字，继承了`Point`类的所有属性和方法。但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个`Point`类。下面，我们在`ColorPoint`内部加上代码。

```
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y); // 调用父类的constructor(x, y)
    this.color = color;
  }

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

上面代码中，`constructor`方法和`toString`方法之中，都出现了`super`关键字，它在这里表示父类的构造函数，用来新建父类的`this`对象。

子类必须在`constructor`方法中调用`super`方法，否则新建实例时会报错。这是因为子类自己的`this`对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，加上子类自己的实例属性和方法。如果不调用`super`方法，子类就得不到`this`对象。