ES6入门

ES6 既是一个历史名词，也是一个泛指，含义是 5.1 版以后的 JavaScript 的下一代标准，涵盖了 ES2015、ES2016、ES2017 等等.

let和const命令

let命令

基本用法

let和var一样都是用来声明变量,但是它所声明的变量,只在let命令所在的代码块内有效.从ES6开始,JS拥有了块级作用域,下面看一下,var与let的区别.

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

for循环的计数器,适合使用let命令

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

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

计数器i只在for循环体内有效，在循环体外引用就会报错。

与var 进行对比

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

console.log(i);// 10
```

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.

不存在变量提升

var命令会出现变量提升,即变量可以在声明之前使用,值为undefined。 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;
```

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。

foo使用var声明,会发生变量提升,在运行console.log时,变量foo已经存在了.运行时的顺序是

```
var foo;
console.log(foo);
foo = 2;
```

另外需要说明的是:

- 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）。

“暂时性死区”也意味着typeof不再是一个百分之百安全的操作.因此我们一定要养成良好的编程习惯,变量一定要在声明之后使用,否则就会报错.

- let 不允许在相同作用域内,重复声明同一个变量.

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。外层作用域无法读取内层作用域的变量

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

内层作用域可以定义外层作用域的同名变量(作用域链:内层可以访问到外层的定义变量)。

```
{{{{
  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立即执行函数表达式（IIFE）不再必要了。

- ES6 的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使用大括号，就会报错。
- 尽量避免在块级作用域内声明函数.(ES5规定函数必须在顶层作用域和函数作用域之中声明)

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

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

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

const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针是固定的，至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

```
const foo = {};

// 为 foo 添加一个属性，可以成功
foo.prop = 123;
foo.prop // 123

// 将 foo 指向另一个对象，就会报错
foo = {}; // TypeError: "foo" is read-only
```

常量foo储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把foo指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

如果想将对象冻结,应该使用Object.freeze方法.

变量的解构赋值

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. 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];
```

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
```

默认值

解构赋值允许指定默认值

```
let [foo = true] = [];
foo // true

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

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。

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

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

如果一个数组成员是null，默认值就不会生效，因为null不严格等于undefined。

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

```
function f() {
  console.log('aaa');
}

let [x = f()] = [1];
```

上面代码中，因为x能取到值，所以函数f根本不会执行。上面的代码其实等价于下面的代码。

```
let x;
if ([1][0] === undefined) {
  x = f();
} else {
  x = [1][0];
}
```

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

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

上面最后一个表达式之所以会报错，是因为x用到默认值y时，y还没有声明。

对象的解构赋值

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

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

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"}]
```

对象的解构也可以指定默认值

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

var {x, y = 5} = {x: 1};
x // 1
y // 5

var {x: y = 3} = {};
y // 3

var {x: y = 3} = {x: 5};
y // 5

var { message: msg = 'Something went wrong' } = {};
msg // "Something went wrong"
```

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
```

如果x属性等于null，就不严格相等于undefined，导致默认值不会生效。

如果解构失败，变量的值等于undefined。

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

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，请看下面的代码。

由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。

```
let arr = [1, 2, 3];
let {0 : first, [arr.length - 1] : last} = arr;
first // 1
last // 3
```

函数参数的解构赋值

函数的参数也可以使用解构赋值。

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

add([1, 2]); // 3
```

上面代码中，函数add的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量x和y。对于函数内部的代码来说，它们能感受到的参数就是x和y。

函数参数的解构也可以使用默认值。

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

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

上面代码中，函数move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，下面的写法会得到不一样的结果。

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

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

上面代码是为函数move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。

用途

交换变量的值

```
let x = 1;
let y = 2;

[x, y] = [y, x];
```

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。

```
// 返回一个数组
function example() {
  return [1, 2, 3];
}
let [a, b, c] = example();

// 返回一个对象

function example() {
  return {
    foo: 1,
    bar: 2
  };
}
let { foo, bar } = example();
```

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

提取JSON数据

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data: number } = jsonData;

console.log(id, status, number);
// 42, "OK", [867, 5309]
```

函数参数的默认值

```
jQuery.ajax = function (url, {
  async = true,
  beforeSend = function () {},
  cache = true,
  complete = function () {},
  crossDomain = false,
  global = true,
  // ... more config
}) {
  // ... do stuff
};
```

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。

输入模块的指定方法

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

```
const { SourceMapConsumer, SourceNode } = require("source-map");
```

函数的扩展

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
```

除了简洁，ES6 的写法还有两个好处：首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行.

参数变量是默认声明的，所以不能用let或const再次声明。

```
function foo(x = 5) {
  let x = 1; // error
  const x = 2; // error
}
```

上面代码中，参数变量x是默认声明的，在函数体中，不能用let或const再次声明，否则会报错。

使用参数默认值时，函数不能有同名参数。

```
// 不报错
function foo(x, x, y) {
  // ...
}

// 报错
function foo(x, x, y = 1) {
  // ...
}
// SyntaxError: Duplicate parameter name not allowed in this context
```

另外，一个容易忽略的地方是，参数默认值不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默认值是惰性求值的。

```
let x = 99;
function foo(p = x + 1) {
  console.log(p);
}

foo() // 100

x = 100;
foo() // 101
```

上面代码中，参数p的默认值是x + 1。这时，每次调用函数foo，都会重新计算x + 1，而不是默认p等于 100。

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。

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

foo() // undefined 5
```

上面代码指定，如果没有提供参数，函数foo的参数默认为一个空对象。

作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域（context）。等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的。

```
let x = 1;

function f(y = x) {
  let x = 2;
  console.log(y);
}

f() // 1
```

上面代码中，函数f调用时，参数y = x形成一个单独的作用域。这个作用域里面，变量x本身没有定义，所以指向外层的全局变量x。函数调用时，函数体内部的局部变量x影响不到默认值变量x。

```
function f(y = x) {
  let x = 2;
  console.log(y);
}

f() // ReferenceError: x is not defined
```

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。

```
// arguments变量的写法
function sortNumbers() {
  return Array.prototype.slice.call(arguments).sort();
}

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

上面代码的两种写法，比较后可以发现，rest 参数的写法更自然也更简洁。

arguments对象不是数组，而是一个类似数组的对象。所以为了使用数组的方法，必须使用Array.prototype.slice.call先将其转为数组。rest 参数就不存在这个问题，它就是一个真正的数组，数组特有的方法都可以使用

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。

箭头函数

基本用法

ES6 允许使用“箭头”（=>）定义函数。

```
var f = v => v;
```

上面的箭头函数等同于：

```
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" });
```

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。

```
let fn = () => void doesNotReturn();
```

箭头函数可以与变量解构结合使用。

```
const full = ({ first, last }) => first + ' ' + last;

// 等同于
function full(person) {
  return person.first + ' ' + person.last;
}
```

使用注意点

箭头函数有几个使用注意点。

（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。

（2）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。

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

（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。

```

function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

var id = 21;

foo.call({ id: 42 });
```

上面代码中，setTimeout的参数是一个箭头函数，这个箭头函数的定义生效是在foo函数生成时，而它的真正执行要等到 100 毫秒后。如果是普通函数，执行时this应该指向全局对象window，这时应该输出21。但是，箭头函数导致this总是指向函数定义生效时所在的对象（本例是{id: 42}），所以输出的是42。

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。

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。下面是一个例子，DOM 事件的回调函数封装在一个对象里面。

```
var handler = {
  id: '123456',

  init: function() {
    document.addEventListener('click',
      event => this.doSomething(event.type), false);
  },

  doSomething: function(type) {
    console.log('Handling ' + type  + ' for ' + this.id);
  }
};
```

上面代码的init方法中，使用了箭头函数，这导致这个箭头函数里面的this，总是指向handler对象。否则，回调函数运行时，this.doSomething这一行会报错，因为此时this指向document对象。

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。正是因为它没有this，所以也就不能用作构造函数。

所以，箭头函数转成 ES5 的代码如下。

```
// ES6
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

// ES5
function foo() {
  var _this = this;

  setTimeout(function () {
    console.log('id:', _this.id);
  }, 100);
}
```

上面代码中，转换后的 ES5 版本清楚地说明了，箭头函数里面根本没有自己的this，而是引用外层的this。

由于箭头函数没有自己的this，所以当然也就不能用call()、apply()、bind()这些方法去改变this的指向。

数组的扩展

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。

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

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

[...document.querySelectorAll('div')]
// [<div>, <div>, <div>]
```

该运算符主要用于函数调用。

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

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

add(...numbers)这两行，是函数的调用，使用了扩展运算符。该运算符将一个数组，变为参数序列。

替代数组的apply方法

由于扩展运算符可以展开数组，所以不再需要apply方法，将数组转为函数的参数了。

```
// 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) {
  // ...
}
let args = [0, 1, 2];
f(...args);
```

扩展运算符的应用

复制数组

数组是复合的数据类型，直接复制的话，只是复制了指向底层数据结构的指针，而不是克隆一个全新的数组。

```
const a1 = [1, 2];
const a2 = a1;

a2[0] = 2;
a1 // [2, 2]
```

上面代码中，a2并不是a1的克隆，而是指向同一份数据的另一个指针。修改a2，会直接导致a1的变化。

ES5 只能用变通方法来复制数组。

```
const a1 = [1, 2];
const a2 = a1.concat();

a2[0] = 2;
a1 // [1, 2]
```

上面代码中，a1会返回原数组的克隆，再修改a2就不会对a1产生影响。

扩展运算符提供了复制数组的简便写法。

```
const a1 = [1, 2];
// 写法一
const a2 = [...a1];
// 写法二
const [...a2] = a1;
上面的两种写法，a2都是a1的克隆。
```

合并数组

扩展运算符提供了数组合并的新写法。

```
// ES5
[1, 2].concat(more)
// ES6
[1, 2, ...more]

var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];

// ES5的合并数组
arr1.concat(arr2, arr3);
// [ 'a', 'b', 'c', 'd', 'e' ]

// ES6的合并数组
[...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于生成数组。

```
// ES5
a = list[0], rest = list.slice(1)
// ES6
[a, ...rest] = list
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...butLast, last] = [1, 2, 3, 4, 5];
// 报错

const [first, ...middle, last] = [1, 2, 3, 4, 5];
// 报错
```

对象的扩展

属性的简洁表示法

```
const foo = 'bar';
const baz = {foo};
baz // {foo: "bar"}

// 等同于
const baz = {foo: foo};
```

上面代码表明，ES6 允许在对象之中，直接写变量。这时，属性名为变量名, 属性值为变量的值。

除了属性简写，方法也可以简写。

```
const o = {
  method() {
    return "Hello!";
  }
};

// 等同于

const o = {
  method: function() {
    return "Hello!";
  }
};
```

属性名表达式

JavaScript 定义对象的属性，有两种方法。

```
// 方法一
obj.foo = true;

// 方法二
obj['a' + 'bc'] = 123;
```

上面代码的方法一是直接用标识符作为属性名，方法二是用表达式作为属性名，这时要将表达式放在方括号之内。

但是，如果使用字面量方式定义对象（使用大括号），在 ES5 中只能使用方法一（标识符）定义属性。

```
var obj = {
  foo: true,
  abc: 123
};
```

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。

```
let propKey = 'foo';

let obj = {


};
```

注意，属性名表达式与简洁表示法，不能同时使用，会报错。

```
// 报错
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] };

// 正确
const foo = 'bar';
const baz = { [foo]: 'abc'};
```

Object.aasign()

基本用法

Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

```
const target = { a: 1 };

const source1 = { b: 2 };
const source2 = { c: 3 };

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

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

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。

```
const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

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

如果只有一个参数，Object.assign会直接返回该参数。

```
const obj = {a: 1};
Object.assign(obj) === obj // true
```

如果该参数不是对象，则会先转成对象，然后返回。

```
typeof Object.assign(2) // "object"
```

由于undefined和null无法转成对象，所以如果它们作为参数，就会报错。

```
Object.assign(undefined) // 报错
Object.assign(null) // 报错
```

Object.assign拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（enumerable: false）。

```
Object.assign({b: 'c'},
  Object.defineProperty({}, 'invisible', {
    enumerable: false,
    value: 'hello'
  })
)
// { b: 'c' }
```

上面代码中，Object.assign要拷贝的对象只有一个不可枚举属性invisible，这个属性并没有被拷贝进去。

注意点

浅拷贝

（1）浅拷贝

Object.assign方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

```
const obj1 = {a: {b: 1}};
const obj2 = Object.assign({}, obj1);

obj1.a.b = 2;
obj2.a.b // 2
上面代码中，源对象obj1的a属性的值是一个对象，Object.assign拷贝得到的是这个对象的引用。这个对象的任何变化，都会反映到目
```

2）同名属性的替换

对于这种嵌套的对象，一旦遇到同名属性，Object.assign的处理方法是替换，而不是添加。

```
const target = { a: { b: 'c', d: 'e' } }
const source = { a: { b: 'hello' } }
Object.assign(target, source)
// { a: { b: 'hello' } }
```

上面代码中，target对象的a属性被source对象的a属性整个替换掉了，而不会得到{ a: { b: 'hello', d: 'e' } }的结果。这通常不是开发者想要的，需要特别小心。

一些函数库提供Object.assign的定制版本（比如 Lodash 的_.defaultsDeep方法），可以得到深拷贝的合并。

（3）数组的处理

Object.assign可以用来处理数组，但是会把数组视为对象。

```
Object.assign([1, 2, 3], [4, 5])
// [4, 5, 3]
```

上面代码中，Object.assign把数组视为属性名为 0、1、2 的对象，因此源数组的 0 号属性4覆盖了目标数组的 0 号属性1。

（4）取值函数的处理

Object.assign只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。

const source = { get foo() { return 1 } }; const target = {};

Object.assign(target, source) // { foo: 1 } 上面代码中，source对象的foo属性是一个取值函数，Object.assign不会复制这个取值函数，只会拿到值以后，将这个值复制过去。

常见用途

Object.assign方法有很多用处。

（1）为对象添加属性

```
class Point {
  constructor(x, y) {
    Object.assign(this, {x, y});
  }
}
```

上面方法通过Object.assign方法，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 () {
  ···
};
```

上面代码使用了对象属性的简洁表示法，直接将两个函数放在大括号中，再使用assign方法添加到SomeClass.prototype之中。

（3）克隆对象

```
function clone(origin) {
  return Object.assign({}, origin);
}
```

上面代码将原始对象拷贝到一个空对象，就得到了原始对象的克隆。

不过，采用这种方法克隆，只能克隆原始对象自身的值，不能克隆它继承的值。如果想要保持继承链，可以采用下面的代码。

```
function clone(origin) {
  let originProto = Object.getPrototypeOf(origin);
  return Object.assign(Object.create(originProto), origin);
}
```

（4）合并多个对象

将多个对象合并到某个对象。

```
const merge =
  (target, ...sources) => Object.assign(target, ...sources);
```

如果希望合并后返回一个新对象，可以改写上面函数，对一个空对象合并。

```
const merge =
  (...sources) => Object.assign({}, ...sources);
```

（5）为属性指定默认值

```
const DEFAULTS = {
  logLevel: 0,
  outputFormat: 'html'
};

function processContent(options) {
  options = Object.assign({}, DEFAULTS, options);
  console.log(options);
  // ...
}
```

上面代码中，DEFAULTS对象是默认值，options对象是用户提供的参数。Object.assign方法将DEFAULTS和options合并成一个新对象，如果两者有同名属性，则option的属性值会覆盖DEFAULTS的属性值。

注意，由于存在浅拷贝的问题，DEFAULTS对象和options对象的所有属性的值，最好都是简单类型，不要指向另一个对象。否则，DEFAULTS对象的该属性很可能不起作用。

属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

（1）for...in

for...in循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）。

（2）Object.keys(obj)

Object.keys返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

（3）Object.getOwnPropertyNames(obj)

Object.getOwnPropertyNames返回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名。

（4）Object.getOwnPropertySymbols(obj)

Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有 Symbol 属性的键名。

（5）Reflect.ownKeys(obj)

Reflect.ownKeys返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

以上的 5 种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

首先遍历所有数值键，按照数值升序排列。 其次遍历所有字符串键，按照加入时间升序排列。 最后遍历所有 Symbol 键，按照加入时间升序排列。

```
Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })
// ['2', '10', 'b', 'a', Symbol()]
```

上面代码中，Reflect.ownKeys方法返回一个数组，包含了参数对象的所有属性。这个数组的属性次序是这样的，首先是数值属性2和10，其次是字符串属性b和a，最后是 Symbol 属性。

对象的扩展运算符

解构赋值

对象的解构赋值用于从一个对象取值，相当于将所有可遍历的、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

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

上面代码中，变量z是解构赋值所在的对象。它获取等号右边的所有尚未读取的键（a和b），将它们连同值一起拷贝过来。

由于解构赋值要求等号右边是一个对象，所以如果等号右边是undefined或null，就会报错，因为它们无法转为对象。

```
let { x, y, ...z } = null; // 运行时错误
let { x, y, ...z } = undefined; // 运行时错误
```

解构赋值必须是最后一个参数，否则会报错。

```
let { ...x, y, z } = obj; // 句法错误
let { x, ...y, ...z } = obj; // 句法错误
```

上面代码中，解构赋值不是最后一个参数，所以会报错。

注意，解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。

扩展运算符

扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

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

扩展运算符可以用于合并两个对象。

```
let ab = { ...a, ...b };
// 等同于
let ab = Object.assign({}, a, b);
```

如果用户自定义的属性，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。

```
let aWithOverrides = { ...a, x: 1, y: 2 };
// 等同于
let aWithOverrides = { ...a, ...{ x: 1, y: 2 } };
// 等同于
let x = 1, y = 2, aWithOverrides = { ...a, x, y };
// 等同于
let aWithOverrides = Object.assign({}, a, { x: 1, y: 2 });
```

上面代码中，a对象的x属性和y属性，拷贝到新对象后会被覆盖掉。

如果扩展运算符后面是一个空对象，则没有任何效果。

```
{...{}, a: 1}
// { a: 1 }
```

如果扩展运算符的参数是null或undefined，这两个值会被忽略，不会报错。

```
let emptyObject = { ...null, ...undefined }; // 不报错
```

Promise对象

Promise含义

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

Promise对象有以下两个特点。

（1）对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为resolved（已定型）。如果改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

注意，为了行文方便，本章后面的resolved统一只指fulfilled状态，不包含rejected状态。

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对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。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）。

基本用法

ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。

下面代码创造了一个Promise实例。

```
const promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

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。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，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rejected状态的回调函数。

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

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对象的状态变为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。

Promise 新建后就会立即执行。

```
let promise = new Promise(function(resolve, reject) {
  console.log('Promise');
  resolve();
});

promise.then(function() {
  console.log('resolved.');
});

console.log('Hi!');

// Promise
// Hi!
// resolved
```

上面代码中，Promise 新建后立即执行，所以首先输出的是Promise。然后，then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以resolved最后输出。

如果调用resolve函数和reject函数时带有参数，那么它们的参数会被传递给回调函数。reject函数的参数通常是Error对象的实例，表示抛出的错误；resolve函数的参数除了正常的值以外，还可能是另一个 Promise 实例，比如像下面这样。

```
const p1 = new Promise(function (resolve, reject) {
  // ...
});

const p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
})
```

上面代码中，p1和p2都是 Promise 的实例，但是p2的resolve方法将p1作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时p1的状态就会传递给p2，也就是说，p1的状态决定了p2的状态。如果p1的状态是pending，那么p2的回调函数就会等待p1的状态改变；如果p1的状态已经是resolved或者rejected，那么p2的回调函数将会立刻执行。

注意，调用resolve或reject并不会终结 Promise 的参数函数的执行。

```
new Promise((resolve, reject) => {
  resolve(1);
  console.log(2);
}).then(r => {
  console.log(r);
});
// 2
// 1
```

上面代码中，调用resolve(1)以后，后面的console.log(2)还是会执行，并且会首先打印出来。这是因为立即 resolved 的 Promise 是在本轮事件循环的末尾执行，总是晚于本轮循环的同步任务。

一般来说，调用resolve或reject以后，Promise 的使命就完成了，后继操作应该放到then方法里面，而不应该直接写在resolve或reject的后面。所以，最好在它们前面加上return语句，这样就不会有意外。

```
new Promise((resolve, reject) => {
  return resolve(1);
  // 后面的语句不会执行
  console.log(2);
})
```

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。

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。

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

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。

```
getJSON("/post/1.json").then(function(post) {
  return getJSON(post.commentURL);
}).then(function funcA(comments) {
  console.log("resolved: ", comments);
}, function funcB(err){
  console.log("rejected: ", err);
});
```

上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为resolved，就调用funcA，如果状态变为rejected，就调用funcB。

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 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

上面代码中，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。另外，then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。

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。

一般来说，不要在then方法里面定义 Reject 状态的回调函数（即then的第二个参数），总是使用catch方法。

```
// bad
promise
  .then(function(data) {
    // success
  }, function(err) {
    // error
  });

// good
promise
  .then(function(data) { //cb
    // success
  })
  .catch(function(err) {
    // error
  });
```

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面then方法执行中的错误，也更接近同步的写法（try/catch）。因此，建议总是使用catch方法，而不使用then方法的第二个参数。

Class的基本语法

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。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

```
class Point {
}

// 等同于
class Point {
  constructor() {}
}
```

上面代码中，定义了一个空的类Point，JavaScript 引擎会自动为它添加一个空的constructor方法。

```
class Foo {
  constructor() {
    return Object.create(null);
  }
}

new Foo() instanceof Foo
// false
```

上面代码中，constructor函数返回一个全新的对象，结果导致实例对象不是Foo类的实例。

类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。

```
class Foo {
  constructor() {
    return Object.create(null);
  }
}

Foo()
// TypeError: Class constructor Foo cannot be invoked without 'new'
```

类的实例对象

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

```
class Point {
  // ...
}

// 报错
var point = Point(2, 3);

// 正确
var point = new Point(2, 3);
```

与 ES5 一样，实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

```
//定义类
class Point {

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }

}

var point = new Point(2, 3);

point.toString() // (2, 3)

point.hasOwnProperty('x') // true
point.hasOwnProperty('y') // true
point.hasOwnProperty('toString') // false
point.__proto__.hasOwnProperty('toString') // true
```

上面代码中，x和y都是实例对象point自身的属性（因为定义在this变量上），所以hasOwnProperty方法返回true，而toString是原型对象的属性（因为定义在Point类上），所以hasOwnProperty方法返回false。这些都与 ES5 的行为保持一致。

与 ES5 一样，类的所有实例共享一个原型对象。

Class表达式

与函数一样，类也可以使用表达式的形式定义。

```
const MyClass = class Me {
  getClassName() {
    return Me.name;
  }
};
```

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 的内部代码可用，指代当前类。

如果类的内部没用到的话，可以省略Me，也就是可以写成下面的形式。

```
const MyClass = class { /* ... */ };
采用 Class 表达式，可以写出立即执行的 Class。

let person = new class {
  constructor(name) {
    this.name = name;
  }

  sayName() {
    console.log(this.name);
  }
}('张三');

person.sayName(); // "张三"
```

上面代码中，person是一个立即执行的类的实例。

不存在的变量提升

类不存在变量提升（hoist），这一点与 ES5 完全不同。

```
new Foo(); // ReferenceError
class Foo {}
```

上面代码中，Foo类使用在前，定义在后，这样会报错，因为 ES6 不会把类的声明提升到代码头部。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

{ let Foo = class {}; class Bar extends Foo { } } 上面的代码不会报错，因为Bar继承Foo的时候，Foo已经有定义了。但是，如果存在class的提升，上面代码就会报错，因为class会被提升到代码头部，而let命令是不提升的，所以导致Bar继承Foo的时候，Foo还没有定义。

私有方法

私有方法是常见需求，但 ES6 不提供，只能通过变通方法模拟实现。

一种做法是在命名上加以区别。

```
class Widget {

  // 公有方法
  foo (baz) {
    this._bar(baz);
  }

  // 私有方法
  _bar(baz) {
    return this.snaf = baz;
  }

  // ...
}
```

上面代码中，_bar方法前面的下划线，表示这是一个只限于内部使用的私有方法。但是，这种命名是不保险的，在类的外部，还是可以调用到这个方法。

另一种方法就是索性将私有方法移出模块，因为模块内部的所有方法都是对外可见的。

```
class Widget {
  foo (baz) {
    bar.call(this, baz);
  }

  // ...
}

function bar(baz) {
  return this.snaf = baz;
}
```

上面代码中，foo是公有方法，内部调用了bar.call(this, baz)。这使得bar实际上成为了当前模块的私有方法。

还有一种方法是利用Symbol值的唯一性，将私有方法的名字命名为一个Symbol值。

```
const bar = Symbol('bar');
const snaf = Symbol('snaf');

export default class myClass{

  // 公有方法
  foo(baz) {
    this[bar](baz);
  }

  // 私有方法
  [bar](baz) {
    return this[snaf] = baz;
  }

  // ...
};
```

上面代码中，bar和snaf都是Symbol值，导致第三方无法获取到它们，因此达到了私有方法和私有属性的效果。

this的指向

类的方法内部如果含有this，它默认指向类的实例。但是，必须非常小心，一旦单独使用该方法，很可能报错。

```
class Logger {
  printName(name = 'there') {
    this.print(`Hello ${name}`);
  }

  print(text) {
    console.log(text);
  }
}

const logger = new Logger();
const { printName } = logger;
printName(); // TypeError: Cannot read property 'print' of undefined
```

上面代码中，printName方法中的this，默认指向Logger类的实例。但是，如果将这个方法提取出来单独使用，this会指向该方法运行时所在的环境，因为找不到print方法而导致报错。

一个比较简单的解决方法是，在构造方法中绑定this，这样就不会找不到print方法了。

```
class Logger {
  constructor() {
    this.printName = this.printName.bind(this);
  }

  // ...
}
```

另一种解决方法是使用箭头函数。

```
class Logger {
  constructor() {
    this.printName = (name = 'there') => {
      this.print(`Hello ${name}`);
    };
  }

  // ...
}
```

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: foo.classMethod 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。

注意，如果静态方法包含this关键字，这个this指的是类，而不是实例。

```
class Foo {
  static bar () {
    this.baz();
  }
  static baz () {
    console.log('hello');
  }
  baz () {
    console.log('world');
  }
}

Foo.bar() // hello
```

上面代码中，静态方法bar调用了this.baz，这里的this指的是Foo类，而不是Foo的实例，等同于调用Foo.baz。另外，从这个例子还可以看出，静态方法可以与非静态方法重名。

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() // "hello, too"
```

Class 的静态属性和实例属性

静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。

```
class Foo {
}

Foo.prop = 1;
Foo.prop // 1
```

上面的写法为Foo类定义了一个静态属性prop。

类的实例属性

类的实例属性可以用等式，写入类的定义之中。

```
class MyClass {
  myProp = 42;

  constructor() {
    console.log(this.myProp); // 42
  }
}
```

上面代码中，myProp就是MyClass的实例属性。在MyClass的实例上，可以读取这个属性。

以前，我们定义实例属性，只能写在类的constructor方法里面。

```
class ReactCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
}
```

上面代码中，构造方法constructor里面，定义了this.state属性。

有了新的写法以后，可以不在constructor方法里面定义。

```
class ReactCounter extends React.Component {
  state = {
    count: 0
  };
}
```

类的静态属性

类的静态属性只要在上面的实例属性写法前面，加上static关键字就可以了。

class MyClass { static myStaticProp = 42;

constructor() { console.log(MyClass.myStaticProp); // 42 } }

Class的继承

简介

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。

ES5 的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6 的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

如果子类没有定义constructor方法，这个方法会被默认添加. 也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

最后，父类的静态方法，也会被子类继承。

```
class A {
  static hello() {
    console.log('hello world');
  }
}

class B extends A {
}

B.hello()  // hello world
```

上面代码中，hello()是A类的静态方法，B继承A，也继承了A的静态方法。

Object.getPrototypeOf()

Object.getPrototypeOf方法可以用来从子类上获取父类。

```
Object.getPrototypeOf(ColorPoint) === Point
// true
```

因此，可以使用这个方法判断，一个类是否继承了另一个类。

super关键字

super这个关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

第一种情况，super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数。

```
class A {}

class B extends A {
  constructor() {
    super();
  }
}
```

上面代码中，子类B的构造函数之中的super()，代表调用父类的构造函数。这是必须的，否则 JavaScript 引擎会报错。

注意，super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B，因此super()在这里相当于A.prototype.constructor.call(this)。作为函数时，super()只能用在子类的构造函数之中，用在其他地方就会报错。

第二种情况，super作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

```
class A {
  p() {
    return 2;
  }
}

class B extends A {
  constructor() {
    super();
    console.log(super.p()); // 2
  }
}

let b = new B();
```

上面代码中，子类B当中的super.p()，就是将super当作一个对象使用。这时，super在普通方法之中，指向A.prototype，所以super.p()就相当于A.prototype.p()。

这里需要注意，由于super指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过super调用的。

```
class A {
  constructor() {
    this.p = 2;
  }
}

class B extends A {
  get m() {
    return super.p;
  }
}

let b = new B();
b.m // undefined
```

上面代码中，p是父类A实例的属性，super.p就引用不到它。

如果属性定义在父类的原型对象上，super就可以取到。

```
class A {}
A.prototype.x = 2;

class B extends A {
  constructor() {
    super();
    console.log(super.x) // 2
  }
}

let b = new B();
```

上面代码中，属性x是定义在A.prototype上面的，所以super.x可以取到它的值。

extends 的继承目标

extends关键字后面可以跟多种类型的值。

```
class B extends A {
}
```

上面代码的A，只要是一个有prototype属性的函数，就能被B继承。由于函数都有prototype属性（除了Function.prototype函数），因此A可以是任意函数。

下面，讨论三种特殊情况。

第一种特殊情况，子类继承Object类。

```
class A extends Object {
}

A.__proto__ === Object // true
A.prototype.__proto__ === Object.prototype // true
```

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。

第二种特殊情况，不存在任何继承。

```
class A {
}

A.__proto__ === Function.prototype // true
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，A作为一个基类（即不存在任何继承），就是一个普通函数，所以直接继承Function.prototype。但是，A调用后返回一个空对象（即Object实例），所以A.prototype.**proto**指向构造函数（Object）的prototype属性。

第三种特殊情况，子类继承null。

```
class A extends null {
}

A.__proto__ === Function.prototype // true
A.prototype.__proto__ === undefined // true
```

这种情况与第二种情况非常像。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，即实质上执行了下面的代码。

```
class C extends null {
  constructor() { return Object.create(null); }
}
```

Mixin 模式的实现

Mixin 指的是多个对象合成一个新的对象，新对象具有各个组成成员的接口。它的最简单实现如下。

```
const a = {
  a: 'a'
};
const b = {
  b: 'b'
};
const c = {...a, ...b}; // {a: 'a', b: 'b'}
```

上面代码中，c对象是a对象和b对象的合成，具有两者的接口。

下面是一个更完备的实现，将多个类的接口“混入”（mix in）另一个类。

```
function mix(...mixins) {
  class Mix {}

  for (let mixin of mixins) {
    copyProperties(Mix, mixin); // 拷贝实例属性
    copyProperties(Mix.prototype, mixin.prototype); // 拷贝原型属性
  }

  return Mix;
}

function copyProperties(target, source) {
  for (let key of Reflect.ownKeys(source)) {
    if ( key !== "constructor"
      && key !== "prototype"
      && key !== "name"
    ) {
      let desc = Object.getOwnPropertyDescriptor(source, key);
      Object.defineProperty(target, key, desc);
    }
  }
}
```

上面代码的mix函数，可以将多个对象合成为一个类。使用的时候，只要继承这个类即可。

```
class DistributedEdit extends mix(Loggable, Serializable) {
  // ...
}
```

Module语法

概述

ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

```
// CommonJS模块
let { stat, exists, readFile } = require('fs');

// 等同于
let _fs = require('fs');
let stat = _fs.stat;
let exists = _fs.exists;
let readfile = _fs.readfile;
```

上面代码的实质是整体加载fs模块（即加载fs的所有方法），生成一个对象（_fs），然后再从这个对象上面读取 3 个方法。这种加载称为“运行时加载”，因为只有运行时才能得到这个对象，导致完全没办法在编译时做“静态优化”。

ES6 模块不是对象，而是通过export命令显式指定输出的代码，再通过import命令输入。

```
// ES6模块
import { stat, exists, readFile } from 'fs';
```

上面代码的实质是从fs模块加载 3 个方法，其他方法不加载。这种加载称为“编译时加载”或者静态加载，即 ES6 可以在编译时就完成模块加载，效率要比 CommonJS 模块的加载方式高。当然，这也导致了没法引用 ES6 模块本身，因为它不是对象。

除了静态加载带来的各种好处，ES6 模块还有以下好处。

- 不再需要UMD模块格式了，将来服务器和浏览器都会支持 ES6 模块格式。目前，通过各种工具库，其实已经做到了这一点。
- 将来浏览器的新 API 就能用模块格式提供，不再必须做成全局变量或者navigator对象的属性。
- 不再需要对象作为命名空间（比如Math对象），未来这些功能可以通过模块提供。

严格模式

ES6 的模块自动采用严格模式，不管你有没有在模块头部加上"use strict";。

严格模式主要有以下限制:

1. 变量必须声明后再使用
2. 函数的参数不能有同名属性，否则报错
3. 不能使用with语句
4. 不能对只读属性赋值，否则报错
5. 不能使用前缀 0 表示八进制数，否则报错
6. 不能删除不可删除的属性，否则报错
7. 不能删除变量delete prop，会报错，只能删除属性delete global[prop]
8. eval不会在它的外层作用域引入变量
9. eval和arguments不能被重新赋值
10. arguments不会自动反映函数参数的变化
11. 不能使用arguments.callee
12. 不能使用arguments.caller
13. 禁止this指向全局对象
14. 不能使用fn.caller和fn.arguments获取函数调用的堆栈
15. 增加了保留字（比如protected、static和interface）

其中，尤其需要注意this的限制。ES6 模块之中，顶层的this指向undefined，即不应该在顶层代码使用this。

export命令

模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的

功能。

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。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。下面是一个 JS 文件，里面使用export命令输出变量。

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

上面代码是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。

export的写法，除了像上面这样，还有另外一种。

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

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

上面代码在export命令后面，使用大括号指定所要输出的一组变量。它与前一种写法（直接放置在var语句前）是等价的，但是应该优先考虑使用这种写法。因为这样就可以在脚本尾部，一眼看清楚输出了哪些变量。

export命令除了输出变量，还可以输出函数或类（class）。

```
export function multiply(x, y) {
  return x * y;
};
```

上面代码对外输出一个函数multiply。

通常情况下，export输出的变量就是本来的名字，但是可以使用as关键字重命名。

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

上面代码使用as关键字，重命名了函数v1和v2的对外接口。重命名后，v2可以用不同的名字输出两次。

需要特别注意的是，export命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

```
// 报错
export 1;

// 报错
var m = 1;
export m;
```

上面两种写法都会报错，因为没有提供对外的接口。第一种写法直接输出 1，第二种写法通过变量m，还是直接输出 1。1只是一个值，不是接口。正确的写法是下面这样。

```
// 写法一
export var m = 1;

// 写法二
var m = 1;
export {m};

// 写法三
var n = 1;
export {n as m};
```

上面三种写法都是正确的，规定了对外的接口m。其他脚本可以通过这个接口，取到值1。它们的实质是，在接口名与模块内部变量之间，建立了一一对应的关系。

同样的，function和class的输出，也必须遵守这样的写法。

```
// 报错
function f() {}
export f;

// 正确
export function f() {};

// 正确
function f() {}
export {f};
```

另外，export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

```
export var foo = 'bar';
setTimeout(() => foo = 'baz', 500);
```

上面代码输出变量foo，值为bar，500 毫秒之后变成baz。

最后，export命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的import命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了 ES6 模块的设计初衷。

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。

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

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

上面代码的import命令，用于加载profile.js文件，并从中输入变量。import命令接受一对大括号，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命令要使用as关键字，将输入的变量重命名。

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

import后面的from指定模块文件的位置，可以是相对路径，也可以是绝对路径，.js后缀可以省略。如果只是模块名，不带有路径，那么必须有配置文件，告诉 JavaScript 引擎该模块的位置。

```
import {myMethod} from 'util';
```

上面代码中，util是模块文件名，由于不带有路径，必须通过配置，告诉引擎怎么取到这个模块。

注意，import命令具有提升效果，会提升到整个模块的头部，首先执行。

```
foo();

import { foo } from 'my_module';
```

上面的代码不会报错，因为import的执行早于foo的调用。这种行为的本质是，import命令是编译阶段执行的，在代码运行之前。

由于import是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

```
// 报错
import { 'f' + 'oo' } from 'my_module';

// 报错
let module = 'my_module';
import { foo } from module;

// 报错
if (x === 1) {
  import { foo } from 'module1';
} else {
  import { foo } from 'module2';
}
```

上面三种写法都会报错，因为它们用到了表达式、变量和if结构。在静态分析阶段，这些语法都是没法得到值的。

最后，import语句会执行所加载的模块，因此可以有下面的写法。

```
import 'lodash';
```

上面代码仅仅执行lodash模块，但是不输入任何值。

如果多次重复执行同一句import语句，那么只会执行一次，而不会执行多次。

```
import 'lodash';
import 'lodash';
```

上面代码加载了两次lodash，但是只会执行一次。

```
import { foo } from 'my_module';
import { bar } from 'my_module';

// 等同于
import { foo, bar } from 'my_module';
```

上面代码中，虽然foo和bar在两个语句中加载，但是它们对应的是同一个my_module实例。也就是说，import语句是 Singleton 模式。

目前阶段，通过 Babel 转码，CommonJS 模块的require命令和 ES6 模块的import命令，可以写在同一个模块里面，但是最好不要这样做。因为import在静态解析阶段执行，所以它是一个模块之中最早执行的。

模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（*）指定一个对象，所有输出值都加载在这个对象上面。

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;
}
```

现在，加载这个模块。

```
// main.js

import { area, circumference } from './circle';

console.log('圆面积: ' + area(4));
console.log('圆周长: ' + circumference(14));
```

上面写法是逐一指定要加载的方法，整体加载的写法如下。

```
import * as circle from './circle';

console.log('圆面积: ' + circle.area(4));
console.log('圆周长: ' + circle.circumference(14));
```

注意，模块整体加载所在的那个对象（上例是circle），应该是可以静态分析的，所以不允许运行时改变。下面的写法都是不允许的。

```
import * as circle from './circle';

// 下面两行都是不允许的
circle.foo = 'hello';
circle.area = function () {};
```

export default 命令

从前面的例子可以看出，使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('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，这时就不需要知道原模块输出的函数名。需要注意的是，这时import命令后面，不使用大括号。

export default命令用在非匿名函数前，也是可以的。

```
// export-default.js
export default function foo() {
  console.log('foo');
}

// 或者写成

function foo() {
  console.log('foo');
}

export default foo;
```

上面代码中，foo函数的函数名foo，在模块外部是无效的。加载的时候，视同匿名函数加载。

下面比较一下默认输出和正常输出。

```
// 第一组
export default function crc32() { // 输出
  // ...
}

import crc32 from 'crc32'; // 输入

// 第二组
export function crc32() { // 输出
  // ...
};

import {crc32} from 'crc32'; // 输入
```

上面代码的两组写法，第一组是使用export default时，对应的import语句不需要使用大括号；第二组是不使用export default时，对应的import语句需要使用大括号。

export default命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此export default命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能对应一个方法。

本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效

的。

```
// modules.js
function add(x, y) {
  return x * y;
}
export {add as default};
// 等同于
// export default add;

// app.js
import { default as foo } from 'modules';
// 等同于
// import foo from 'modules';
```

正是因为export default命令其实只是输出一个叫做default的变量，所以它后面不能跟变量声明语句。

```
// 正确
export var a = 1;

// 正确
var a = 1;
export default a;

// 错误
export default var a = 1;
```

上面代码中，export default a的含义是将变量a的值赋给变量default。所以，最后一种写法会报错。

同样地，因为export default本质是将该命令后面的值，赋给default变量以后再默认，所以直接将一个值写在export default之后。

```
// 正确
export default 42;

// 报错
export 42;
```

上面代码中，后一句报错是因为没有指定对外的接口，而前一句指定外对接口为default。

有了export default命令，输入模块时就非常直观了，以输入 lodash 模块为例。

```
import _ from 'lodash';
```

如果想在一条import语句中，同时输入默认方法和其他接口，可以写成下面这样。

```
import _, { each, each as forEach } from 'lodash';
```

对应上面代码的export语句如下。

```
export default function (obj) {
  // ...
}

export function each(obj, iterator, context) {
  // ...
}

export { each as forEach };
```

上面代码的最后一行的意思是，暴露出forEach接口，默认指向each接口，即forEach和each指向同一个方法。

export default也可以用来输出类。

```
// MyClass.js
export default class { ... }

// main.js
import MyClass from 'MyClass';
let o = new MyClass();
```

更多ES6知识点请参考: http://es6.ruanyifeng.com/