ES6

数组扩展

- 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。

- 扩展运算符与正常的函数参数可以结合使用，非常灵活。
  `function f(v, w, x, y, z) { }`
  `const args = [0, 1];`
  `f(-1, ...args, 2, ...[3]);`
- 扩展运算符后面还可以放置表达式。
  `const arr = [`
  `...(x > 0 ? ['a'] : []),`
  `'b',`
  `];`
- 如果扩展运算符后面是一个空数组，则不产生任何效果。
  `[...[], 1]`
  `// [1]`
- 替代函数的 apply 方法
  - 下面是扩展运算符取代apply方法的一个实际的例子，应用Math.max方法，简化求出一个数组最大元素的写法。
    `// ES5 的写法`
    `Math.max.apply(null, [14, 3, 77])`
    `// ES6 的写法`
    `Math.max(...[14, 3, 77])`
    `// 等同于`
    `Math.max(14, 3, 77);`

- 通过push函数，将一个数组添加到另一个数组的尾部。

```
// ES5的 写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
Array.prototype.push.apply(arr1, arr2);
// ES6 的写法
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1.push(...arr2);
```

扩展运算符的应用

（1）**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];
// 写法一
const a2 = [...a1];
// 写法二
const [...a2] = a1;
```

（2）**合并数组**

扩展运算符提供了数组合并的新写法。

```
// ES5 的合并数组
arr1.concat(arr2, arr3);
// ES6 的合并数组
[...arr1, ...arr2, ...arr3]
```

这两种方法都是浅拷贝，使用的时候需要注意。

（3）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。

（4）字符串

扩展运算符还可以将字符串转为真正的数组。

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

（5）实现了 Iterator 接口的对象

任何 Iterator 接口的对象（参阅 Iterator 一章），都可以用扩展运算符转为真正的数组。

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

上面代码中，querySelectorAll方法返回的是一个nodeList对象。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因在于NodeList对象实现了

Iterator 。

2.Array.from()

- 用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）。
  ```
  // 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还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。
  ```
  Array.from(arrayLike, x => x * x);
  // 等同于
  Array.from(arrayLike).map(x => x * x);
  Array.from([1, 2, 3], (x) => x * x)
  // [1, 4, 9]
  ```

3.Array.of()

- 用于将一组值，转换为数组。
  ```
  Array.of(3, 11, 8) // [3,11,8]
  Array.of(3) // [3]
  Array.of(3).length // 1
  ```
- Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。

4.数组实例的 copyWithin()

- 数组实例的copyWithin方法，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），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。
  ```
  Array.prototype.copyWithin(target, start = 0, end = this.length)
  ```
  它接受三个参数。
  - target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
  - start（可选）：从该位置开始读取数据，默认为 0。如果为负值，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。
  - 这三个参数都应该是数值，如果不是，会自动转为数值。

```
// 将3号位复制到0号位
[1, 2, 3, 4, 5].copyWithin(0, 3, 4)
// [4, 2, 3, 4, 5]
```

```
// -2相当于3号位，-1相当于4号位
[1, 2, 3, 4, 5].copyWithin(0, -2, -1)
// [4, 2, 3, 4, 5]

// 将3号位复制到0号位
[].copyWithin.call({length: 5, 3: 1}, 0, 3)
// {0: 1, 3: 1, length: 5}

// 将2号位到数组结束，复制到0号位
let i32a = new Int32Array([1, 2, 3, 4, 5]);
i32a.copyWithin(0, 2);
// Int32Array [3, 4, 5, 4, 5]

// 对于没有部署 TypedArray 的 copyWithin 方法的平台
// 需要采用下面的写法
[].copyWithin.call(new Int32Array([1, 2, 3, 4, 5]), 0, 3, 4);
// Int32Array [4, 2, 3, 4, 5]
```

5.数组实例的 find() 和 findIndex()

- 数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为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。

`[1, 4, -5, 10].find((n) => n < 0)`
`// -5`
上面代码找出数组中第一个小于 0 的成员。
`[1, 5, 10, 15].find(function(value, index, arr) {`
`return value > 9;`
`}) // 10`
上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。
数组实例的findIndex方法的用法与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。
`[1, 5, 10, 15].findIndex(function(value, index, arr) {`
`return value > 9;`
`}) // 2`
这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。
`function f(v){`
`return v > this.age;`
`}`
`let person = {name: 'John', age: 20};`
`[10, 12, 26, 15].find(f, person); // 26`
上面的代码中，find函数接收了第二个参数person对象，回调函数中的this对象指向person

对象。
另外，这两个方法都可以发现NaN，弥补了数组的indexOf方法的不足。

```
[NaN].indexOf(NaN)
// -1
[NaN].findIndex(y => Object.is(NaN, y))
// 0
```

上面代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。

变量的结构赋值

- 可以从数组中提取值，按照对应位置，对变量赋值。
  let [a, b, c] = [1, 2, 3];
- 模式匹配：

  ```
  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 // []
  ```

  解构不成功，变量的值就等于undefined。
- 不完全解构:

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

对象的解构赋值

let { foo, bar } = { foo: “aaa”, bar: “bbb” };
foo // “aaa”
bar // “bbb”
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。

- 变量名与属性名不一致，必须写成:

```
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'
```

对象的解构赋值的内部机制:

- 1.是先找到同名属性
- 2.再赋给对应的变量,
  真正被赋值的是后者

嵌套赋值

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

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

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

字符串的解构赋值

- 字符串转换成类数组对象:
  const [a, b, c, d, e] = ‘hello’;
  a // “h”
  b // “e”
  c // “l”
  d // “l”
  e // “o”
- 属性解构赋值:
  let {length : len} = ‘hello’;
  len // 5

数值和布尔值的解构赋值

- 等号右边是数值和布尔值，先转为对象:
  let {toString: s} = 123;
  s = Number.prototype.toString // true
  let {toString: s} = true;
  s = Boolean.prototype.toString // true

函数参数的解构赋值

```
function add([x, y]){
return x + y;
}
add([1, 2]); // 3
```

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。

不能使用圆括号的情况

（1）变量声明语句
（2）函数参数
（3）赋值语句的模式

结构赋值的用途

（1）交换变量的值
let x = 1;
let y = 2;
[x, y] = [y, x];
（2）从函数返回多个值

- 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();
```

（3）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});`

（4）提取 JSON 数据:
解构赋值对提取 JSON 对象中的数据，尤其有用。

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

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

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

（5）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
};
```

（6）遍历 Map 结构
任何部署了 Iterator 接口的对象，都可以用for…of循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。

```
const map = new Map();
map.set('first', 'hello');
map.set('second', 'world');

for (let [key, value] of map) {
  console.log(key + " is " + value);
}
// first is hello
// second is world
```

只获取键名或键值，可以写成:

```
// 获取键名
for (let [key] of map) {
  // ...
}

// 获取键值
for (let [,value] of map) {
  // ...
}
```

（7）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");
```

函数扩展

- 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
```

- 与解构赋值默认值结合使用

```
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
```

- 函数的 length 属性
  指定默认值以后，函数的length属性，指定默认值后，length属性将失真。

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

- 作用域
  设置了参数默认值，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，不会出现。

```
var x = 1;
function f(x, y = x) {
  console.log(y);
}
f(2) // 2
```

- rest 参数
  rest 参数（ 形式为…变量名 ），用于获取函数多余参数，不需使用arguments对象。
  rest 参数搭配的变量是一个数组，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
```

注意: `rest 参数之后不能再有其他参数（只能是最后一个参数），否则会报错。`

函数的length属性，不包括 rest 参数。

- name 属性
  function foo() {}
  foo.name // “foo”

箭头函数

```
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; }
```

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:
（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。

Class 的基本语法

- Class（类）：作为对象的模板。通过class关键字，可以定义类。

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

- 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。
- 类的实例对象

```
class Point {
  // ...
}
// 报错
var point = Point(2, 3);
// 正确
var point = new Point(2, 3);
```

- Class 表达式

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

- 不存在变量提升
  要先定义再使用

- 私有方法和私有属性：

```
class Widget {
  // 公有方法
  foo (baz) {
    this._bar(baz);
  }
  // 私有方法
  _bar(baz) {
    return this.snaf = baz;
  }
}
```

- this 的指向
  类的方法内部如果含有this，它默认指向类的实例。
- Class 的静态方法
  在方法前加上static关键字:表示该方法不会被实例继承，而是直接通过类来调用，称为“静态方法”。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}
```

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。

类的 prototype 属性和proto属性

Class 作为构造函数的语法糖，同时有prototype属性和**proto**属性，因此同时存在两条继承链。
（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。