ES6

ECMAScript 和 JavaScript 的关系

前者是后者的规格，后者是前者的一种实现

ES6是什么

ES6全称ECMAScript 6，泛指5.1 版以后的 JavaScript 标准

顶层对象

JavaScript中存在一个提供全局环境（即全局作用域）的顶层对象，所有代码都是在这个环境中运行

在浏览器中（Worker除外），顶层对象是`window`，在其他环境下没有`window`

`globalThis`总是指向顶层对象

尾逗号

函数定义或调用时，最后一个参数后允许有一个逗号：

```
function move(
  x,
  y,
) { /* … */ }
move(
  114514,
  1919810,
)
```

数组和对象的最后一个元素或属性后允许有一个逗号：

```
['p2p', '下崽器',]

{
  'Content-Type': 'application/msgpack',
  Authorization: 'Bearer KFCCRAZYTHURSDAYVIVO50',
}
```

变量/常量

变量提升与重复声明

使用`var`会出现“变量提升”现象，即变量可以在声明之前使用，值为`undefined`，`let`、`const`和`class`则不存在这种情况：

```
console.log(foo) // undefined
var foo = 2

console.log(bar) // ReferenceError: bar is not defined
let bar = 2
```

`let`、`const`和`class`均不允许重复声明

```
function fxxk() {
  let a = 114
  var a = 514 // SyntaxError: Identifier 'a' has already been declared
}

function fxxk(b) {
  let a = 666
  let a = 666 // SyntaxError: Identifier 'a' has already been declared
  let b // 与参数重名，报错
  {
    let b // 在新的作用域内，正确
  }
}
```

常量

常量就是声明后不能重新赋值的变量

```
const PI = Math.acos(-1)
PI = 3.14 // TypeError: Assignment to constant variable
PI -= .1 // 同上
```

const声明必须赋值，不能留到以后赋值

```
const gusha // SyntaxError: Missing initializer in const declaration
```

const保证的，并不是变量的值不得改动，而是变量名所指的数据不得改动

- 对于值类型数据（数值、BigInt、字符串、布尔值、Symbol）：const等同于常量
- 对于引用类型数据：const变量所指对象内容是可变的，但是不能使const变量指向另一个对象

```
const blacklist = []
blacklist.push('vovi', 'oqqo') // OK
blacklist.length = 0  // OK
blacklist = ['@bishi']   // TypeError: Assignment to constant variable
```

代码块与块级作用域

代码块（大括号）用于创建块级作用域，外层代码块不受内层代码块的影响，内层作用域可以定义外层作用域的同名变量

```
let n = 1
if (n) {
  let n = 0 // 这个n和上面的n不是同一变量
  console.log(n) // 0
}
console.log(n) // 1
```

代码块可任意嵌套，每层都是一个单独的作用域

```
{{
  {let banner = '油门一响，爹妈白养'}
  console.log(banner) // ReferenceError: banner is not defined
}}
```

应该避免在块级作用域内使用`function`声明函数，例如：

```
function f() { console.log('乌鸦坐飞机'); }
(() => {
  if (0) {
    // 重复声明一次函数f
    function f() { console.log('妈妈生的') }
  }
  f()
})()
// Uncaught TypeError: f is not a function
```

上面代码中，`function`声明的函数行为类似`var`，被提升到函数作用域内，值为`undefined`，所以出现了与预期不一致的结果

在块级作用域内声明函数应该写成赋值语句的形式，对于不变的的函数应该设置为常量

```
{
  let seconds = 30 * 30 * 24 * 3600
  const getTrainingTime = function () {
    return seconds
  }
}
```

for循环的括号是个单独的作用域

```
for (let i = 0; i < 3; i++) {
  let i = 3
  console.log(i) // 这个i和上面的i是两个变量
}
console.log(i) // ReferenceError: i is not defined
```

暂时性死区

只要块级作用域内存在`let`、`const`、`class`声明，它所声明的变量就不再受外部的影响

```
var s = 123
if (1) {
  s = 'a' // ReferenceError
  let s
}
```

上面代码中在`let`声明变量之前该变量都是不可用的，这个区域称为“暂时性死区”

顶层对象的属性

`let`、`const`、`class`声明的全局变量，不属于顶层对象的属性：

```
var a = 1
globalThis.a // 1
let b = 1
globalThis.b // undefined
```

常量

所有的函数都应该设置为常量

语句扩展

```
try {
  // …
} catch (e) {
  // …
}
```

如果用不到参数`e`，可省略catch的参数：

```
try {
  // …
} catch {
  // …
}
```

可遍历对象

内置的可遍历对象有

- 数组
- 字符串
- Set、Map
- TypedArray系列

`for...of`循环用于遍历可遍历对象，例如

```
const arr = ['老六', '老七', '老八']
for(let v of arr)
  console.log(v) // 老六 老七 老八
```

解构赋值

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，称为解构，例如

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

等价于

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

多余的值会被赋为`undefined`，例如

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

以上两种情况`foo`的值均为`undefined`

默认值

解构结果为`undefined`时，会赋为默认值，如：

```
let [foo = true] = []
foo // true
let [x, y = 'b'] = ['a'] // x='a', y='b'
let [x, y = 'b'] = ['a', null] // x='a', y='b'
```

如果默认值为表达式，那么这个表达式是惰性求值的

```
function f() { console.log('foo') }
let [x = f()] = [1]
```

上面代码中，因为x能取到非`undefined`的值，所以函数f根本不会执行

默认值可以引用解构赋值的其他变量

```
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: y is not defined
```

上面最后一个表达式报错是因为`x`用`y`做默认值时，`y`还没有声明

可遍历对象的解构赋值

以`[`开头的解构赋值，等号右边必须为可遍历对象，例如：

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

```
baz // 3
let [ , , third] = ["妈见打", "封妈谱", "撒背您"]
third // "撒背您"
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 // []
let [x, y, ...z = 0] = ['a'] // SyntaxError: Invalid destructuring assignment
target
let [x, y] = [1, 2, 3]
x // 1
y // 2
let [a, [b], d] = [1, [2, 3], 4]
a // 1
b // 2
d // 4
```

如果等号右边不是可遍历对象，那么将会报错

```
let [a, b] = {} // TypeError: {} is not iterable
```

扩展运算符用于解构赋值时只能位于最后，否则报错：

```
const [...head, tail] = [1, 2, 3] // Uncaught SyntaxError: Rest element must be
last element
const [first, ...middle, last] =  [1, 2, 3] // Uncaught SyntaxError: Rest
element must be last element
```

对象的解构赋值

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，变量的取值由属性名决定

```
let { foo: baz } = { foo: 'a', bar: 'b' }
baz // "a"
let obj = { first: 'hello', last: 'world' }
let { first: f, last: l } = obj
f // 'hello'
l // 'world'
```

若变量名与属性名相同，可以简写：

```
// 等价于 let { bar: bar, foo: foo } = { foo: 'a', bar: 'b' }
let { bar, foo } = { foo: 'a', bar: 'b' }
foo // "a"
bar // "b"
let { baz } = { foo: 'a', bar: 'b' }
baz // undefined
```

对象解构也可以用于嵌套的对象：

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

若解构模式是嵌套的对象，且子对象所在的父属性不存在，会报错

```
let {foo: {bar}} = {baz: 'baz'} // TypeError: Cannot read properties of
undefined (reading 'bar')
```

对象解构赋值的规则是，只要等号右边的值不是对象，就先将其转为对象。`undefined`和`null`无法转为对象，对它们进行解构赋值都会报错

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
```

扩展运算符用于对象解构时只能位于最后

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

函数参数的解构赋值

函数参数也可以使用解构赋值：

```
[[1, 2], [3, 4]].map(([a, b]) => a + b)
// [3, 7]
```

带默认值参数解构：

```
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]
```

上面是为变量`x`和`y`指定默认值

下面是为函数`move`的参数指定默认值，会得到不同结果

```
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]
```

注意事项

解构赋值前不一定需要`let`、`const`或`var`，例如交换变量的值：

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

但如果解构赋值语句以`{`开头会被当成一个代码块，导致语法错误：

```
{a: b} = obj // SyntaxError: Unexpected token '='
```

需加上`()`：

```
({a: b} = obj)
```

解构赋值允许等号左边为`{}`

```
({} = [true, false])
({} = 'a')
({} = [])
```

上面的表达式虽然毫无意义，但可以执行

数值的扩展

允许使用`_`作为分隔符，对于较长的数特别有用：

```
1_919_810.114_514
```

注意：

1. 不能为数值的首位和末位

2. 不能有多个分隔符连在一起
3. 小数点的前后不能有分隔符
4. 科学计数法中，表示指数的`e`或`E`前后不能有分隔符
5. 分隔符不能紧跟着进制的前缀`0b`、`0o`、`0x`（不分大小写）

下面三个将字符串转成数值的函数，不支持数值分隔符

- Number
- parseInt
- parseFloat
  例如：

```
Number('123_456') // NaN
parseInt('123_456') // 123
```

BigInt

BigInt是用于精确表示大整数的类型

```
const a = 2172141653n
const b = 15346349309n
// BigInt 可以保持精度
a * b // 33334444555566667777n
// 普通整数无法保持精度
Number(a) * Number(b) // 33334444555566670000

1234 // 普通整数
1234n // BigInt
// BigInt 的运算
1n + 2n // 3n
```

BigInt 与普通整数是两种值：

```
42n === 42 // false

typeof 42n // 'bigint'
```

BigInt 不能与普通数值进行混合运算

```
1n + 1.3 // 报错
```

可以使用负号（`-`），但是不能使用正号（`+`），只能用`Number()`显式转换为普通数值

```
-42n // 正确
+42n // 报错

Number(1n) // 1
```

BigInt 与字符串混合运算时，会先转为字符串，再进行运算

取反运算符（`!`）可将 BigInt 转为布尔值

```
'' + 123n // "123"

!0n // true
!1n // false
```

几乎所有的数值运算符都可以用在 BigInt，但是有两个例外：

1. 无符号的右移位运算符 `>>>`（对BigInt无意义，请用右移运算符 `>>` 代替）
2. 一元的正运算符 `+`

除法运算 `/` 会舍去小数部分：

```
9n / 5n // 1n
```

新增运算符

指数运算符

指数运算符是右结合，而不是常见的左结合，并且有相应的组合赋值运算符：

```
2 ** 3 ** 2 // 512，相当于 2 ** (3 ** 2)

let a = 1.5
a **= 2 // 同 a = a ** a;
let b = 4
b **= 0.5 // 同 b = Math.sqrt(b);
```

链判断运算符

`a?.b` 相当于 `a == null ? undefined : a.b`，这里的 `b` 还可以是 `[x]`、`()`、`f()`，例如：

- `a?.[x]`
- `a?.()`
- `a?.f()`

`?.` 右侧不得紧邻数字，如 `foo?.3:0` 会被解析为 `foo ? .3 : 0`

?? 运算符

`??` 类似 `||`，但是只有运算符左侧的值为 `null` 或 `undefined` 时，才会返回右侧的值

这个运算符很适合判断函数参数是否赋值，例如：

```
function Component(props) {
  const enable = props.enabled ?? true
  // …
}
```

当 `props.enabled` 不为 `null` 时等同于

```
function Component(props) {
  const {
    enabled: enable = true,
  } = props
  // …
}
```

`??`与其他逻辑运算符一起使用必须用括号表明优先级，否则报错

```
// SyntaxError
lhs && middle ?? rhs
lhs ?? middle && rhs
lhs || middle ?? rhs
lhs ?? middle || rhs
```

`??`运算符有对应的组合赋值运算符`??=`，`a ??= b`相当于`a = a ?? b`

数组的扩展

扩展运算符是三个点（`...`）。是 rest 参数的逆运算，将一个可遍历对象转为用逗号分隔的参数序列，例如：

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

扩展运算符后面还可以是表达式，若扩展一个空的可遍历对象（如空数组或空串），则无任何效果

```
const arr = [
  ...(x > 0 ? ['上单'] : []),
  '蒙鼓族',
]
```

数组的浅拷贝

```
const a = [1, 2];
const b = a;
b[0] = 2;
a // [2, 2]
```

上面代码中，`a`与`b`均指向同一个数组，因为数组是一种对象类型，修改`b`，相当于修改`a`，若要修改`b`的同时不影响`a`，可以使用浅拷贝：

```
const a = [1, 2]
// 写法一
const b = [...a]
// 写法二
const [...b] = a
```

对象的浅拷贝

与数组的浅拷贝差不多

```
let a = { x: 114, y: 514 }
let b = { ...a } // 或 let { ...b } = a
b.x = 666
a.x // 114
```

合并数组

```
const x = ['a', 'b']
const y = ['c']
const z = ['d', 'e']

[...x, ...y, ...z] // [ 'a', 'b', 'c', 'd', 'e' ]
```

字符串转为字符数组

```
[...'ikun'] // ["i", "k", "u", "n"]

// 不能正确识别四个字节的 Unicode 字符
'x\uD83D\uDE80y'.length // 4
// 可正确识别四个字节的 Unicode 字符
[...'x\uD83D\uDE80y'].length // 3
```

Array.from与Array.of

`Array.from(x)`等价于`[...x]`

`Array.of`用于将一组值，转换为数组：

```
Array.of(11, 45, 14) // [11, 45, 14]
Array.of(3) // [3]
Array.of(3).length // 1
```

这个方法用于弥补`Array()`的不足，`Array()`的行为会因参数个数的不同而变化

数组的空位

空位不是`undefined`，`undefined`作为数组元素时下标是存在的。空位表示该下标不存在，in运算符可以说明这一点：

```
0 in [undefined, undefined] // true
0 in [, ,] // false
```

扩展运算符（`...`）会将空位转为undefined，`copyWithin()`会连空位一起拷贝

```
[...['a',,'b']] // ["a", undefined, "b"]
[,'a','b',,].copyWithin(2,0) // [,"a",,"a"]
```

对数组使用`map`方法遍历会跳过空位，其他函数会把空位当成`undefined`处理

排序稳定性

排序稳定性是排序算法的重要属性，指的是排序关键字相同的项目，排序前后的顺序不变

假设有一个姓和名的列表，要求按照“姓氏为主要关键字，名字为次要关键字”进行排序。开发者可能会先按名字排序，再按姓氏进行排序。如果排序算法是稳定的，这样就可以达到“先姓氏，后名字”的排序效果。如果是不稳定的，就不行

Array.prototype.sort的默认排序算法是稳定的

字符串的扩展

模板字符串

模板字符串可包含换行符，也可以嵌入变量，用反引号（`）表示

```
console.log(`第一行
第二行`)
// 字符串中嵌入变量
let name = "伤害", replacement = "要是"
`伤害都是${name}打的，要是换成${replacement}，早就打完了`
```

大括号内部可以放入任意表达式，若表达式的值不是字符串，将转为字符串

```
`${x} + ${y * 2} = ${x + y * 2}` // "1 + 4 = 5"
```

CodePoint

一句话，尽量用`String.fromCodePoint`和`codePointAt`代替`String.fromCharCode`和`charCodeAt`

箭头函数

```
function f() { return this }
const g = () => this
f.call({a: 1}) // {a: 1}
g.call({a: 1})
```

箭头函数内的`this`对象就是定义时的`this`对象，并且不能改变

对象的扩展

对象属性的简写

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

对象方法的简写

```
const o = {
  say() {
    console.log("Hello!")
  }
}
// 等同于
const o = {
  say: function() {
    console.log("Hello!")
  }
}
```

对象方法中的`this`指向对象本身：

```
const o = {
  value: 0,
  print() {
    console.log(this.value)
  }
}
o.value = 1
o.print() // 1
```

属性名表达式

属性名可以是表达式，此时实际属性名为表达式结果转换成字符串后的值

```
let key = 'foo';
let obj = {
  [key]: true,
  [`s${key}`]: 6,

  // 定义方法
  ['h' + 'ello']() {
    return 'hi'
  }
}
```

属性名表达式不能与简写同时使用：

```
// 正确
const suzhi = 666
const ikun = { [suzhi]: 666 }
// 错误
const suzhi = 666
const ikun = { [suzhi] } // SyntaxError: Unexpected token '['
```

Symbol

Symbol表示独一无二的值。它是一种基本数据类型，不是对象

```
let s = Symbol()
typeof s // "symbol"
```

`Symbol`函数前不能使用`new`，否则会报错

`Symbol`函数可以接受一个字符串作为参数，如果不是字符串则将其转为字符串，表示对`Symbol`实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分

Symbol 可以显式转为字符串

```
let s1 = Symbol('毙肾客')
let s2 = Symbol('并夕夕')
s1 // Symbol(毙肾客)
s2 // Symbol(并夕夕)
s1.toString() // "Symbol(毙肾客)"，等同于String(s1)
s2.toString() // "Symbol(并夕夕)"
```

Symbol 不能与其他类型的值进行运算

```
let s = Symbol('77');
'磨磨唧唧，必出' + s // TypeError: can't convert symbol to string
```

Symbol.for

```
Symbol() == Symbol() // false
Symbol.for('毙肾客') == Symbol.for('毙肾客') // true
Symbol.for() == Symbol.for('undefined') // true
```

Set

集合是一个数据结构，类似于数组，但没有重复的值

- `size`返回集合大小
- `add`方法用于向集合添加值，会忽略重复的值
- `delete`方法用于删除值，返回是否成功删除
- `has`方法用来判断值是否在集合中
- `clear`方法用于清空集合

```
const s = new Set()
[5,3,2,3,1,3,2,3].forEach(x => s.add(x))
```

`Set`函数可以接受一个可遍历对象，因此上述代码等效于

```
const s = new Set([5,3,2,3,1,3,2,3])
```

Set的一些应用

```
[...new Set(arr)] // 去除数组`arr`的重复成员
[...new Set('老师，菜菜，捞捞')].join('') // 去除字符串中的重复字符
```

Map

Map是一个数据结构，允许将任意值当作键，普通的对象只允许字符串和Symbol作为键

Map内部使用`Object.is`来比较键

```
const m = new Map()
const o = () => '银盒护卫队'
m.set(o, '抬棺七人众')
m.get(o) // "抬棺七人众"
m.has(o) // true
m.delete(o) // true
m.has(o) // false

m.set(-0, '种门')
m.get(+0) // '种门'
m.set(true, '张三克星')
m.set('true', '歪比洼卜')
m.get(true) // '张三克星'
m.set(NaN, 'SbF₅')
m.get(NaN) // 'SbF₅'
```

Map本身也是一个对象，因此能当普通对象用，虽然以下写法不会报错，但这是错误的用法

```
m[true] = '东风快递'
```

Set和Map的遍历

Set和Map均按照插入顺序遍历

```
for (let i of s)
  console.log(i)
```

类

类可以看成是函数声明的语法糖，类名本身是一个构造函数

```
class Point {
  constructor(x, y) {
    this.x = x
    this.y = y
  }

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

  // 静态方法
  static test() {
    console.log(this)
  }
}
```

```
typeof Point // "function"
Point === Point.prototype.constructor // true
```

调用类的构造函数必须加`new`，否则报错

类的方法中的`this`指向类的实例（对象），静态方法中的`this`指向类（函数）