ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

对象的扩展

上一章 下一章

对象（object）是 JavaScript 最重要的数据结构。ES6 对它进行了重大升级，本章介绍数据结构本身的改变，下一章介绍 `Object` 对象的新增方法。

---

1. 属性的简洁表示法

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

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

// 等同于
const baz = {foo: foo};
```

上面代码中，变量 `foo` 直接写在大括号里面。这时，属性名就是变量名，属性值就是变量值。下面是另一个例子。

```
function f(x, y) {
  return {x, y};
}

// 等同于

function f(x, y) {
  return {x: x, y: y};
}

f(1, 2) // Object {x: 1, y: 2}
```

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!";
  }
};
```

下面是一个实际的例子。

```
let birth = '2000/01/01';

const Person = {

  name: '张三',

  //等同于birth: birth
  birth,

  // 等同于hello: function ()...
  hello() { console.log('我的名字是', this.name); }

};
```

这种写法用于函数的返回值，将会非常方便。

```
function getPoint() {
  const x = 1;
  const y = 10;
  return {x, y};
}

getPoint()
// {x:1, y:10}
```

CommonJS 模块输出一组变量，就非常合适使用简洁写法。

```
let ms = {};

function getItem (key) {
  return key in ms ? ms[key] : null;
}

function setItem (key, value) {
  ms[key] = value;
}

function clear () {
  ms = {};
}

module.exports = { getItem, setItem, clear };
// 等同于
module.exports = {
  getItem: getItem,
  setItem: setItem,
  clear: clear
};
```

属性的赋值器（setter）和取值器（getter），事实上也是采用这种写法。

```
const cart = {
  _wheels: 4,

  get wheels () {
    return this._wheels;
  },

  set wheels (value) {
    if (value < this._wheels) {
      throw new Error('数值太小了！');
    }
```

```
    this._wheels = value;
  }
}
```

简洁写法在打印对象时也很有用。

```
let user = {
  name: 'test'
};

let foo = {
  bar: 'baz'
};

console.log(user, foo)
// {name: "test"} {bar: "baz"}
console.log({user, foo})
// {user: {name: "test"}, foo: {bar: "baz"}}
```

上面代码中，`console.log` 直接输出 `user` 和 `foo` 两个对象时，就是两组键值对，可能会混淆。把它们放在大括号里面输出，就变成了对象的简洁表示法，每组键值对前面会打印对象名，这样就比较清晰了。

注意，简写的对象方法不能用作构造函数，会报错。

```
const obj = {
  f() {
    this.foo = 'bar';
  }
};

new obj.f() // 报错
```

上面代码中，`f` 是一个简写的对象方法，所以 `obj.f` 不能当作构造函数使用。

---

2. 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 = {
  [propKey]: true,
  ['a' + 'bc']: 123
};
```

下面是另一个例子。

```
let lastWord = 'last word';

const a = {
  'first word': 'hello',
  [lastWord]: 'world'
};

a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

表达式还可以用于定义方法名。

```
let obj = {
  ['h' + 'ello']() {
    return 'hi';
  }
};

obj.hello() // hi
```

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'};
```

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串 `[object Object]`，这一点要特别小心。

```
const keyA = {a: 1};
const keyB = {b: 2};

const myObject = {
  [keyA]: 'valueA',
  [keyB]: 'valueB'
};

myObject // Object {[object Object]: "valueB"}
```

上面代码中，`[keyA]` 和 `[keyB]` 得到的都是 `[object Object]`，所以 `[keyB]` 会把 `[keyA]` 覆盖掉，而 `myObject` 最后只有一个 `[object Object]` 属性。

3. 方法的 name 属性

函数的 `name` 属性，返回函数名。对象方法也是函数，因此也有 `name` 属性。

```
const person = {
  sayName() {
    console.log('hello!');
  },
};

person.sayName.name   // "sayName"
```

上面代码中，方法的 `name` 属性返回函数名（即方法名）。

如果对象的方法使用了取值函数（`getter`）和存值函数（`setter`），则 `name` 属性不是在该方法上面，而是该方法的属性的描述对象的 `get` 和 `set` 属性上面，返回值是方法名前加上 `get` 和 `set`。

```
const obj = {
  get foo() {},
  set foo(x) {}
};

obj.foo.name
// TypeError: Cannot read property 'name' of undefined

const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');

descriptor.get.name // "get foo"
descriptor.set.name // "set foo"
```

有两种特殊情况：`bind` 方法创造的函数，`name` 属性返回 `bound` 加上原函数的名字；`Function` 构造函数创造的函数，`name` 属性返回 `anonymous`。

```
(new Function()).name // "anonymous"

var doSomething = function() {
  // ...
};
doSomething.bind().name // "bound doSomething"
```

如果对象的方法是一个 Symbol 值，那么 `name` 属性返回的是这个 Symbol 值的描述。

```
const key1 = Symbol('description');
const key2 = Symbol();
let obj = {
  [key1]() {},
  [key2]() {},
};
obj[key1].name // "[description]"
obj[key2].name // ""
```

上面代码中，`key1` 对应的 Symbol 值有描述，`key2` 没有。

---

4. 属性的可枚举性和遍历

上一章　　下一章

可枚举性

对象的每个属性都有一个描述对象（Descriptor），用来控制该属性的行为。`Object.getOwnPropertyDescriptor` 方法可以获取该属性的描述对象。

```
let obj = { foo: 123 };
Object.getOwnPropertyDescriptor(obj, 'foo')
//  {
//    value: 123,
//    writable: true,
//    enumerable: true,
//    configurable: true
//  }
```

描述对象的 `enumerable` 属性，称为"可枚举性"，如果该属性为 `false`，就表示某些操作会忽略当前属性。

目前，有四个操作会忽略 `enumerable` 为 `false` 的属性。

- `for...in` 循环：只遍历对象自身的和继承的可枚举的属性。
- `Object.keys()`：返回对象自身的所有可枚举的属性的键名。
- `JSON.stringify()`：只串行化对象自身的可枚举的属性。
- `Object.assign()`： 忽略 `enumerable` 为 `false` 的属性，只拷贝对象自身的可枚举的属性。

这四个操作之中，前三个是 ES5 就有的，最后一个 `Object.assign()` 是 ES6 新增的。其中，只有 `for...in` 会返回继承的属性，其他三个方法都会忽略继承的属性，只处理对象自身的属性。实际上，引入"可枚举"（`enumerable`）这个概念的最初目的，就是让某些属性可以规避掉 `for...in` 操作，不然所有内部属性和方法都会被遍历到。比如，对象原型的 `toString` 方法，以及数组的 `length` 属性，就通过"可枚举性"，从而避免被 `for...in` 遍历到。

```
Object.getOwnPropertyDescriptor(Object.prototype, 'toString').enumerable
// false

Object.getOwnPropertyDescriptor([], 'length').enumerable
// false
```

上面代码中，`toString` 和 `length` 属性的 `enumerable` 都是 `false`，因此 `for...in` 不会遍历到这两个继承自原型的属性。

另外，ES6 规定，所有 Class 的原型的方法都是不可枚举的。

```
Object.getOwnPropertyDescriptor(class {foo() {}}.prototype, 'foo').enumerable
// false
```

总的来说，操作中引入继承的属性会让问题复杂化，大多数时候，我们只关心对象自身的属性。所以，尽量不要用 `for...in` 循环，而用 `Object.keys()` 代替。

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。

---

5. super 关键字

我们知道， `this` 关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字 `super` ，指向当前对象的原型对象。

```
const proto = {
  foo: 'hello'
};

const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};

Object.setPrototypeOf(obj, proto);
obj.find() // "hello"
```

上面代码中，对象 `obj.find()` 方法之中，通过 `super.foo` 引用了原型对象 `proto` 的 `foo` 属性。

注意， `super` 关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。

```
// 报错
const obj = {
  foo: super.foo
}
```

```
// 报错
const obj = {
  foo: () => super.foo
}

// 报错
const obj = {
  foo: function () {
    return super.foo
  }
}
```

上面三种 `super` 的用法都会报错，因为对于 JavaScript 引擎来说，这里的 `super` 都没有用在对象的方法之中。第一种写法是 `super` 用在属性里面，第二种和第三种写法是 `super` 用在一个函数里面，然后赋值给 `foo` 属性。目前，只有对象方法的简写法可以让 JavaScript 引擎确认，定义的是对象的方法。

JavaScript 引擎内部， `super.foo` 等同于 `Object.getPrototypeOf(this).foo` （属性）或 `Object.getPrototypeOf(this).foo.call(this)` （方法）。

```
const proto = {
  x: 'hello',
  foo() {
    console.log(this.x);
  },
};

const obj = {
  x: 'world',
  foo() {
    super.foo();
  }
}

Object.setPrototypeOf(obj, proto);

obj.foo() // "world"
```

上面代码中， `super.foo` 指向原型对象 `proto` 的 `foo` 方法，但是绑定的 `this` 却还是当前对象 `obj` ，因此输出的就是 `world` 。

---

6. 对象的扩展运算符

《数组的扩展》一章中，已经介绍过扩展运算符（ `...` ）。ES2018 将这个运算符引入了对象。

解构赋值

对象的解构赋值用于从一个对象取值，相当于将目标对象自身的所有可遍历的（enumerable）、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 { ...z } = null; // 运行时错误
let { ...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 } = someObject; // 句法错误
let { x, ...y, ...z } = someObject; // 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。

```
let obj = { a: { b: 1 } };
let { ...x } = obj;
obj.a.b = 2;
x.a.b // 2
```

上面代码中， x 是解构赋值所在的对象，拷贝了对象 obj 的 a 属性。 a 属性引用了一个对象，修改这个对象的值，会影响到解构赋值对它的引用。

另外，扩展运算符的解构赋值，不能复制继承自原型对象的属性。

```
let o1 = { a: 1 };
let o2 = { b: 2 };
o2.__proto__ = o1;
let { ...o3 } = o2;
o3 // { b: 2 }
o3.a // undefined
```

上面代码中，对象 o3 复制了 o2 ，但是只复制了 o2 自身的属性，没有复制它的原型对象 o1 的属性。

下面是另一个例子。

```
const o = Object.create({ x: 1, y: 2 });
o.z = 3;

let { x, ...newObj } = o;
let { y, z } = newObj;
x // 1
y // undefined
z // 3
```

上面代码中，变量 x 是单纯的解构赋值，所以可以读取对象 o 继承的属性；变量 y 和 z 是扩展运算符的解构赋值，只能读取对象 o 自身的属性，所以变量 z 可以赋值成功，变量 y 取不到值。ES6 规定，变量声明语句之中，如果使用解构赋值，扩展运算符后面必须是一个变量名，而不能是一个解构赋值表达式，所以上面代码引入了中间变量 newObj ，如果写成下面这样会报错。

```
let { x, ...{ y, z } } = o;
// SyntaxError: ... must be followed by an identifier in declaration contexts
```

解构赋值的一个用处，是扩展某个函数的参数，引入其他操作。

```
function baseFunction({ a, b }) {
  // ...
}
function wrapperFunction({ x, y, ...restConfig }) {
  // 使用 x 和 y 参数进行操作
  // 其余参数传给原始函数
  return baseFunction(restConfig);
}
```

上面代码中，原始函数 `baseFunction` 接受 `a` 和 `b` 作为参数，函数 `wrapperFunction` 在 `baseFunction` 的基础上进行了扩展，能够接受多余的参数，并且保留原始函数的行为。

扩展运算符

对象的扩展运算符（`...`）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 }
```

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

```
let foo = { ...['a', 'b', 'c'] };
foo
// {0: "a", 1: "b", 2: "c"}
```

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 }
```

如果扩展运算符后面不是对象，则会自动将其转为对象。

```
// 等同于 {...Object(1)}
{...1} // {}
```

上面代码中，扩展运算符后面是整数 `1`，会自动转为数值的包装对象 `Number{1}`。由于该对象没有自身属性，所以返回一个空对象。

下面的例子都是类似的道理。

```
// 等同于 {...Object(true)}
{...true} // {}

// 等同于 {...Object(undefined)}
{...undefined} // {}

// 等同于 {...Object(null)}
{...null} // {}
```

但是，如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象，因此返回的不是空对象。

```
{...'hello'}
// {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}
```

对象的扩展运算符等同于使用 `Object.assign()` 方法。

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

上面的例子只是拷贝了对象实例的属性，如果想完整克隆一个对象，还拷贝对象原型的属性，可以采用下面的写法。

```
// 写法一
const clone1 = {
  __proto__: Object.getPrototypeOf(obj),
  ...obj
};

// 写法二
const clone2 = Object.assign(
  Object.create(Object.getPrototypeOf(obj)),
  obj
);

// 写法三
const clone3 = Object.create(
  Object.getPrototypeOf(obj),
  Object.getOwnPropertyDescriptors(obj)
)
```

上面代码中，写法一的 `__proto__` 属性在非浏览器的环境不一定部署，因此推荐使用写法二和写法三。

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。

这用来修改现有对象部分的属性就很方便了。

```
let newVersion = {
  ...previousVersion,
  name: 'New Name' // Override the name property
};
```

上面代码中， `newVersion` 对象自定义了 `name` 属性，其他属性全部复制自 `previousVersion` 对象。

如果把自定义属性放在扩展运算符前面，就变成了设置新对象的默认属性值。

```
let aWithDefaults = { x: 1, y: 2, ...a };
// 等同于
```

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

与数组的扩展运算符一样，对象的扩展运算符后面可以跟表达式。

```
const obj = {
  ...(x > 1 ? {a: 1} : {}),
  b: 2,
};
```

扩展运算符的参数对象之中，如果有取值函数 `get`，这个函数是会执行的。

```
// 并不会抛出错误，因为 x 属性只是被定义，但没执行
let aWithXGetter = {
  ...a,
  get x() {
    throw new Error('not throw yet');
  }
};

// 会抛出错误，因为 x 属性被执行了
let runtimeError = {
  ...a,
  ...{
    get x() {
      throw new Error('throw now');
    }
  }
};
```

---

7. 链判断运算符

编程实务中，如果读取对象内部的某个属性，往往需要判断一下该对象是否存在。比如，要读取 `message.body.user.firstName`，安全的写法是写成下面这样。

```
const firstName = (message
  && message.body
  && message.body.user
  && message.body.user.firstName) || 'default';
```

或者使用三元运算符 `?:`，判断一个对象是否存在。

```
const fooInput = myForm.querySelector('input[name=foo]')
const fooValue = fooInput ? fooInput.value : undefined
```

这样的层层判断非常麻烦，因此 ES2020 引入了“链判断运算符”（optional chaining operator） `?.`，简化上面的写法。

```
const firstName = message?.body?.user?.firstName || 'default';
const fooValue = myForm.querySelector('input[name=foo]')?.value
```

上面代码使用了 `?.` 运算符，直接在链式调用的时候判断，左侧的对象是否为 `null` 或 `undefined`。如果是的，就不再往下运算，而是返回 `undefined`。

链判断运算符有三种用法。

- `obj?.prop` // 对象属性
- `obj?.[expr]` // 同上
- `func?.(...args)` // 函数或对象方法的调用

下面是判断对象方法是否存在，如果存在就立即执行的例子。

```
iterator.return?.()
```

上面代码中， `iterator.return` 如果有定义，就会调用该方法，否则直接返回 `undefined` 。

对于那些可能没有实现的方法，这个运算符尤其有用。

```
if (myForm.checkValidity?.() === false) {
  // 表单校验失败
  return;
}
```

上面代码中，老式浏览器的表单可能没有 `checkValidity` 这个方法，这时 `?.` 运算符就会返回 `undefined` ，判断语句就变成了 `undefined === false` ，所以就会跳过下面的代码。

下面是这个运算符常见的使用形式，以及不使用该运算符时的等价形式。

```
a?.b
// 等同于
a == null ? undefined : a.b

a?.[x]
// 等同于
a == null ? undefined : a[x]

a?.b()
// 等同于
a == null ? undefined : a.b()

a?.()
// 等同于
a == null ? undefined : a()
```

上面代码中，特别注意后两种形式，如果 `a?.b()` 里面的 `a.b` 不是函数，不可调用，那么 `a?.b()` 是会报错的。 `a?.()` 也是如此，如果 `a` 不是 `null` 或 `undefined` ，但也不是函数，那么 `a?.()` 会报错。

使用这个运算符，有几个注意点。

（1）短路机制

```
a?.[++x]
// 等同于
a == null ? undefined : a[++x]
```

上面代码中，如果 `a` 是 `undefined` 或 `null` ，那么 `x` 不会进行递增运算。也就是说，链判断运算符一旦为真，右侧的表达式就不再求值。

（2）delete 运算符

```
delete a?.b
// 等同于
```

```
a == null ? undefined : delete a.b
```

上面代码中，如果 a 是 undefined 或 null ，会直接返回 undefined ，而不会进行 delete 运算。

（3）括号的影响

如果属性链有圆括号，链判断运算符对圆括号外部没有影响，只对圆括号内部有影响。

```
(a?.b).c
// 等价于
(a == null ? undefined : a.b).c
```

上面代码中， ?. 对圆括号外部没有影响，不管 a 对象是否存在，圆括号后面的 .c 总是会执行。

一般来说，使用 ?. 运算符的场合，不应该使用圆括号。

（4）报错场合

以下写法是禁止的，会报错。

```
// 构造函数
new a?.()
new a?.b()

// 链判断运算符的右侧有模板字符串
a?.`{b}`
a?.b`{c}`

// 链判断运算符的左侧是 super
super?.()
super?.foo

// 链运算符用于赋值运算符左侧
a?.b = c
```

（5）右侧不得为十进制数值

为了保证兼容以前的代码，允许 foo?.3:0 被解析成 foo ? .3 : 0 ，因此规定如果 ?. 后面紧跟一个十进制数字，那么 ?. 不再被看成是一个完整的运算符，而会按照三元运算符进行处理，也就是说，那个小数点会归属于后面的十进制数字，形成一个小数。

---

8. Null 判断运算符

读取对象属性的时候，如果某个属性的值是 null 或 undefined ，有时候需要为它们指定默认值。常见做法是通过 || 运算符指定默认值。

```
const headerText = response.settings.headerText || 'Hello, world!';
const animationDuration = response.settings.animationDuration || 300;
const showSplashScreen = response.settings.showSplashScreen || true;
```

上面的三行代码都通过 || 运算符指定默认值，但是这样写是错的。开发者的原意是，只要属性的值为 null 或 undefined ，默认值就会生效，但是属性的值如果为空字符串或 false 或 0 ，默认值也会生效。

为了避免这种情况，ES2020 引入了一个新的 Null 判断运算符 ?? 。它的行为类似 || ，但是只有运算符左侧的值为 null 或 undefined 时，才会返回右侧的值。

```
const headerText = response.settings.headerText ?? 'Hello, world!';
const animationDuration = response.settings.animationDuration ?? 300;
const showSplashScreen = response.settings.showSplashScreen ?? true;
```

上面代码中，默认值只有在属性值为 `null` 或 `undefined` 时，才会生效。

这个运算符的一个目的，就是跟链判断运算符 `?.` 配合使用，为 `null` 或 `undefined` 的值设置默认值。

```
const animationDuration = response.settings?.animationDuration ?? 300;
```

上面代码中， `response.settings` 如果是 `null` 或 `undefined` ，就会返回默认值300。

这个运算符很适合判断函数参数是否赋值。

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

上面代码判断 `props` 参数的 `enabled` 属性是否赋值，等同于下面的写法。

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

`??` 有一个运算优先级问题，它与 `&&` 和 `||` 的优先级孰高孰低。现在的规则是，如果多个逻辑运算符一起使用，必须用括号表明优先级，否则会报错。

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

上面四个表达式都会报错，必须加入表明优先级的括号。

```
(lhs && middle) ?? rhs;
lhs && (middle ?? rhs);

(lhs ?? middle) && rhs;
lhs ?? (middle && rhs);

(lhs || middle) ?? rhs;
lhs || (middle ?? rhs);

(lhs ?? middle) || rhs;
lhs ?? (middle || rhs);
```

留言