ECMAScript 6 入门

作者：阮一峰

目录

其他

对象的扩展

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。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;
  }
}
```

注意，简洁写法的属性名总是字符串，这会导致一些看上去比较奇怪的结果。

```
const obj = {
  class () {}
};

// 等同于

var obj = {
  'class': function() {}
};
```

上面代码中，`class` 是字符串，所以不会因为它属于关键字，而导致语法解析报错。

如果某个方法的值是一个 Generator 函数，前面需要加上星号。

```
const obj = {
  * m() {
    yield 'hello world';
  }
};
```

---

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

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 { 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 } = 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');
    }
  }
};
```

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

yunnysunny • 2 years ago

可惜Null传导运算符现在还不支持

23 ^ | v • Reply • Share ›

常小波 ➚ yunnysunny • 2 years ago

Swift里面倒是用的溜溜的，有？和！

^ | v • Reply • Share ›

Xiaodong Guo • 5 years ago

ES6允许表达式作为对象的属性名，在写法上要把表达式放在大括号内。

应该是中括号吧？

2 ^ | v • Reply • Share ›

ruanyf Mod ➚ Xiaodong Guo • 5 years ago

谢谢支出，已经更正了。

1 ^ | v • Reply • Share ›

Frank Fang ➚ Xiaodong Guo • 5 years ago

我也觉得大括号有一些歧义，一般我喜欢说「圆括号」「方括号」和「花括号」 :)

1 ^ | v • Reply • Share ›

neller • 3 months ago

偶遇珠峰培训

see more

1 ^ | v • Reply • Share ›

Randy Hsu • 5 months ago

阮老师，为什么Object.assign() 拷贝第一层属性是深复制，第二层却是浅复制

1 ^ | v • Reply • Share ›

ruanyf Mod ➚ Randy Hsu • 4 months ago

如果属性是对象，拷贝的都是引用。

1 ^ | v • Reply • Share ›

Troy.Edward • 3 months ago

（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);
}
```

阮老师，这样实现的clone是否应该是浅克隆，如果是的话，最好标准说明一下。

^ | v • Reply • Share ›

Linhh • 3 months ago

阮老师，Object.defineProperty(obj, prop, descriptor) 算不算定义对象属性的第三种方法吖？

^ | v • Reply • Share ›

ghr • 6 months ago

let a1={a:3}
let {...a2}=a1
a1.a=4
a2的a值依旧是3

^ | v • Reply • Share ›

1364386878 • 7 months ago

同名属性的替换中 lodash的 defaultsDeep 并没有很好的合并 而是存在即跳过
官方示例：

```
_.defaultsDeep({ 'a': { 'b': 2 } }, { 'a': { 'b': 1, 'c': 3 } });
// => { 'a': { 'b': 2, 'c': 3 } }
```

^ | v • Reply • Share ›

zhengmao xu • 10 months ago

留言板膜拜各位大佬 ^_-

^ | v • Reply • Share ›

Ouyang Chao • a year ago

可惜optional chaining【现在】还不支持， 都2017年底了～

^ | v • Reply • Share ›

李鑫 • a year ago

对象扩展符 取值函数get throws 拼错了

^ | v • Reply • Share ›

ruanyf Mod ➚ 李鑫 • a year ago

谢谢指出，改正了。

^ | v • Reply • Share ›

Zoe Zhang • a year ago

不好意思老师，这两种写法和那句“简洁写法的属性名总是字符串”让我有点懵，如果可以麻烦老师指导下，谢谢

^ | v • Reply • Share ›

绫绮飒 • 2 years ago

// 如果 a 是 null 或 undefined, 返回 undefined
// 否则返回 a?.b.c().d --------->话说这里是不是多了个问号
a?.b.c().d

^ | v • Reply • Share ›

ruanyf Mod ➚ 绫绮飒 • 2 years ago

谢谢指出，我改过来了。

^ | v • Reply • Share ›

绫绮飒 • 2 years ago

直接函数对象传递的引用是没错，直接修改options 我跑了一下感觉有问题。options此刻应该是内部的临时变量，改变临时变量的地址达不到修改options原始值的目的吧。 用return就没问题

另外请教一下：“由于存在深拷贝的问题，DEFAULTS对象和options对象的所有属性的值，都只能是简单类型，而不能指向另一个对象。”这句话应该如何理解

Reply • Share ›

ruanyf Mod ➚ 绫绮飒 • 2 years ago

这里的原意就是在函数内部使用 options，你可以刷新一下，我更新了文章，这部分写得更清楚了。

Reply • Share ›

绫绮飒 ➚ ruanyf • 2 years ago

嗯 我懂了 谢谢老师

Reply • Share ›

huazi • 2 years ago

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

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


这个在我本地直接报这个错误

Reply • Share ›

ruanyf Mod ➚ huazi • 2 years ago

对象的扩展运算符，现在都不支持，必须通过 Babel 转码。

Reply • Share ›

zjy • 2 years ago

var o = Object.create({ x: 1, y: 2 });

var o = Object.create({ x: 1, y: 2 });

o.z = 3;

let { x, ...{ y, z } } = o;

x // 1

y // undefined

z // 3
这个例子报错了，是为什么呢

see more

• Reply • Share ›

ruanyf Mod ➚ zjy • 2 years ago
对象的扩展运算符，只拷贝对象自身的属性，不会拷贝继承的属性。
• Reply • Share ›

niki571 • 2 years ago
var obj = Object.create({}, {p: {value: 42}});
Object.values(obj) // [ ]
上面代码中，Object.create方法的第二个参数添加的对象属性（属性p），如果不显式声明，默认是不可遍历的，因为p是继承的属性，而不是对象自身的属性。Object.values不会返回这个属性。

阮老师，这边p属性不返回不是因为他是继承属性，而是他enumerable:false，改成true就遍历出来了。p属性就是自身属性。
var obj = Object.create({}, {p: {value: 42,enumerable:true}});
Object.values(obj) // [42]
• Reply • Share ›

ruanyf Mod ➚ niki571 • 2 years ago
谢谢指出，已经改正。
• Reply • Share ›

hanzhijue • 2 years ago
Object.observe 为什么会被标准移除呢？
• Reply • Share ›

ruanyf Mod ➚ hanzhijue • 2 years ago
Object.keys() 是 ES5 引入的，当时就没包含在小标题里面。刚才想了想，把它加入标题了。
• Reply • Share ›

hanzhijue ➚ ruanyf • 2 years ago

抱歉抱歉，我也是发现Object.keys() 是 ES5 引入的，所以后来把留言给改了。之前确实是想确认一下是不是小标题9.漏写了Object.keys() 。

Reply • Share ›

ocxers • 2 years ago

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

function processContent(options) {
let options = Object.assign({}, DEFAULTS, options);
}
```

阮老师，这里的let应该去掉，否则会报错: Identifier 'options' has already been declared

Reply • Share ›

ruanyf Mod → ocxers • 2 years ago

谢谢指出，已经改正。

Reply • Share ›

黄欢 • 2 years ago

JavaScript的语法这么设计，以后真是一个人一个风格，浪费很多时间啊

Reply • Share ›

icewind • 3 years ago

这些简写真是语言的噩梦，我真怕以后读别人的最基本的代码都读不懂了

Reply • Share ›

DelphinWU • 3 years ago

阮老师，

```
// 非Generator函数的版本
function entries(obj) {
let arr = [];
for (key of Object.keys(obj)) {
arr.push([key, obj[key]]);
}
return arr;
}
```

此处for循环中貌似缺了一个let吧？

Reply • Share ›

ruanyf Mod → DelphinWU • 3 years ago

谢谢指出，已经改正。

Reply • Share ›

饕餮 • 3 years ago

```
// 非Generator函数的版本
function entries(obj) {
```

return (for (key of Object.keys(obj)) [key, obj[key]]);
}
这段话在执行时会报错

Reply • Share ›

ruanyf Mod ➚ 饕餮 • 3 years ago
谢谢指出，已经改正。

Reply • Share ›

samhwang1990 • 3 years ago
阮老师，Reflect.enumerate() 这个方法已经obsolete 了：
https://developer.mozilla.o...

Reply • Share ›

ruanyf Mod ➚ samhwang1990 • 3 years ago
谢谢指出，删掉了这个方法。

Reply • Share ›

Jun Lang • 3 years ago
mark

Reply • Share ›

maicss ke • 3 years ago
Object.getOwnPropertyDescriptors方法配合Object.defineProperties方法，拷贝属性的set和get方法时，在node v6.2.0 里：
1， 没有getOwnPropertyDescriptors方法，只有一个getOwnPropertyDescriptor方法。
2，改成getOwnPropertyDescriptor方法之后，报错：TypeError: cannot convert undefined or null to object.

谢谢大神的书，在此拜谢。

Reply • Share ›

ruanyf Mod ➚ maicss ke • 3 years ago
这是ES7的方法，还没列入标准，Node不支持啊。

Reply • Share ›

haoju zheng • 3 years ago
下面的代码会有错误， options 已经声明过了

为属性指定默认值
const DEFAULTS = {

logLevel: 0,

outputFormat: 'html'

};

function processContent(options) {

let options = Object.assign({}, DEFAULTS, options);

}

Reply • Share ›

Jack • 3 years ago

"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。只要有一个参数不是对象，就会抛出TypeError错误" 和 "如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同"
有点矛盾，觉得第一句话可以改一改

Reply • Share ›

ruanyf Mod ➚ Jack • 3 years ago

谢谢指出，我会尽快改正。上次修改的的时候，我忘记把前面一起改掉了。

Reply • Share ›

Jack • 3 years ago

你好，请教下+0 -0的区别是什么？实际用途是什么？

Reply • Share ›

ruanyf Mod ➚ Jack • 3 years ago

基本没有区别，就是JavaScript对这两个值有区分。

http://javascript.ruanyifen...

Reply • Share ›

Jack ➚ ruanyf • 3 years ago

thanks

Reply • Share ›

Load more comments

ALSO ON **ECMASCRIPT 6 入门**

异步操作

84 comments • 4 years ago

Qiuleo —

数值的扩展

36 comments • 5 years ago

Heekei Zhuang — Math.trunc 的 polyfill：~~4.1 === 4 // true~~3.9 === 3 // true~~-3.9 === -3 // true……

编程风格

51 comments • 4 years ago

Tom Bian — mark

Module

98 comments • 4 years ago

Don — @ruanyfruanyf