学会了ES6，就不会写出那样的代码

2018 年 04 月 09 日

声明变量的新姿势

用let不用var

ES6之前我们用var声明一个变量，但是它有很多弊病：

- 因为没有块级作用域，很容易声明全局变量
- 变量提升
- 可以重复声明 还记得这道面试题吗?

```
var a = [];
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 10
a[7](); // 10
a[8](); // 10
a[9](); // 10
```

所以，你现在还有什么理由不用let?

有时候const比let更好

const和let的唯一区别就是，const不可以被更改，所以当声明变量的时候，尤其是在声明容易被更改的全局变量的时候，尽量使用const。

- 更好的代码语义化，一眼看到就是常量。
- 另一个原因是因为JavaScript 编译器对const的优化要比let好，多使用const，有利于提高程序的运行效率。
- 所有的函数都应该设置为常量。

动态字符串

不要使用“双引号”，一律用单引号或反引号

```
// low
const a = "foobar";
const b = 'foo' + a + 'bar';

// best
const a = 'foobar';
const b = `foo${a}bar`;
const c = 'foobar';
```

解构赋值的骚操作

变量赋值

在用到数组成员对变量赋值时，尽量使用解构赋值。

```
const arr = [1, 2, 3, 4];

// low
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

函数传对象

函数的参数如果是对象的成员，优先使用解构赋值。

```
// low
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;
}

// good
function getFullName({ firstName, lastName }) {
}
```

如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。这样便于以后添加返回值，以及更改返回值的顺序。

```
// low
function processInput(input) {
  return [left, right, top, bottom];
}

// good
function processInput(input) {
  return { left, right, top, bottom };
}

const { left, right } = processInput(input);
```

关于对象的细节

逗号

单行定义的对象结尾不要逗号：

```
// low
const a = { k1: v1, k2: v2, };

// good
const a = { k1: v1, k2: v2 };
```

多行定义的对象要保留逗号：

```
// low
const b = {
  k1: v1,
  k2: v2
};

// good
const b = {
  k1: v1,
  k2: v2,
};
```

一次性初始化完全

不要声明之后又给对象添加新属性：

```
// low
const a = {};
a.x = 3;

// good
const a = { x: null };
a.x = 3;
```

如果一定非要加请使用Object.assign：

```
const a = {};
Object.assign(a, { x: 3 });
```

如果对象的属性名是动态的，可以在创造对象的时候，使用属性表达式定义：

```
// low
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

再简洁一点

在定义对象时，能简洁表达尽量简洁表达：

```
var ref = 'some value';
```

```
// low
const atom = {
  ref: ref,

  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  ref,

  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

数组

...

使用扩展运算符（...）拷贝数组：

```
// 还在用for i 你就太low了
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// cool !
const itemsCopy = [...items];
```

不要跟我提类数组

用 Array.from 方法，将类似数组的对象转为数组：

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

函数

箭头函数=>

立即执行函数可以写成箭头函数的形式：

```
(() => {
  console.log('Welcome to the Internet.');
})();
```

尽量写箭头函数使你的代码看起来简洁优雅：

```
// low
[1, 2, 3].map(function (x) {
  return x * x;
});
```

```
// cool !
[1, 2, 3].map(x => x * x);
```

不要把布尔值直接传入函数

```
// low
function divide(a, b, option = false ) {
}

// good
function divide(a, b, { option = false } = {}) {
}
```

别再用arguments（类数组）了！

使用 rest 运算符（...）代替，rest 运算符可以提供一个真正的数组。

```
// low
```

```
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

传参时试试设置默认值?

```
// low
function handleThings(opts) {
  opts = opts || {};
}

// good
function handleThings(opts = {}) {
  // ...
}
```

Object? Map!

简单的键值对优先Map

如果只是简单的key: value结构，建议优先使用Map，因为Map提供方便的遍历机制。

```
let map = new Map(arr);
// 遍历key值
for (let key of map.keys()) {
  console.log(key);
}
// 遍历value值
for (let value of map.values()) {
  console.log(value);
}
// 遍历key和value值
for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
```

更加简洁直观class语法

```
// low
function Queue(contents = []) {
  this._queue = [...contents];
}
Queue.prototype.pop = function() {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}

// good
class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

模块化

引入模块

使用import取代require，因为Module是Javascript模块的标准写法。

```
// bad
const moduleA = require('moduleA');
const func1 = moduleA.func1;
const func2 = moduleA.func2;

// good
import { func1, func2 } from 'moduleA';
```

输出模块

使用export输出变量，拒绝module.exports:

```
import React from 'react';

class Breadcrumbs extends React.Component {
  render() {
    return <nav />;
  }
};

export default Breadcrumbs;
```

- 输出单个值，使用export default
- 输出多个值，使用export
- export default与普通的export不要同时使用

编码规范

- 模块输出一个函数，首字母应该小写：

```
function getData() {
}

export default getData;
```

- 模块输出一个对象，首字母应该大写

```
const Person = {
  someCode: {
  }
};

export default Person ;
```