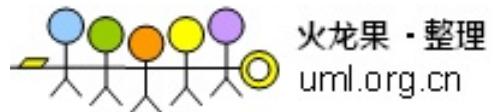


JavaScript 规范 (ES6)



Airbnb 的 [ES5 规范](#)写的非常好，现在添加了 ES6 的部分。

类型

- 原始类型：值传递

- `string`
- `number`
- `boolean`
- `null`
- `undefined`

```
1 const foo = 1;
2 let bar = foo;
3
4 bar = 9;
5
6 console.log(foo, bar);
```

- 复杂类型：引用传递

- `object`
- `array`
- `function`

```
1 const foo = [1, 2];
2 const bar = foo;
3
4 bar[0] = 9;
5
6 console.log(foo[0], bar[0]);
```

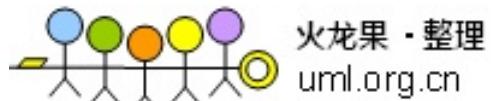
引用

- 为引用使用 `const` 关键字，而不是 `var`

这样确保你不能修改引用类型，否则可能会导致一些 bug 或难以理解的代码。

```
1
2 var a = 1;
3 var b = 2;
4
5
6 const a = 1;
7 const b = 2;
```

- 如果你必须修改引用，使用 `let` 代替 `var`



因为 `let` 是块作用域的，而 `var` 是函数作用域。

```

1  var count = 1;
2  if (true) {
3      count += 1;
4  }
5
6
7
8  let count = 1;
9  if (true) {
10     count += 1;
11 }
```

- `let` 和 `const` 都是块作用域的

对象

- 使用对象字面量创建对象

```

1  var item = new Object();
2
3
4
5  var item = {};
```

- 不要使用[保留字](#)（reserved words）作为键，否则在 IE8 下将出错，issue

```

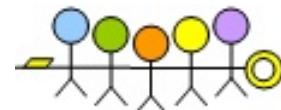
1 // bad
2 var superman = {
3     class: 'superhero',
4     default: { clark: 'kent' },
5     private: true
6 };
7
8 // good
9 var superman = {
10    klass: 'superhero',
11    defaults: { clark: 'kent' },
12    hidden: true
13};
```

- 使用易读的同义词代替保留字

```

1 // bad
2 const superman = {
3     class: 'alien'
4 };
5
6 // bad
7 const superman = {
8     klass: 'alien'
9 };
10
```

```
11 // good
12 const superman = {
13   type: 'alien'
14 };
```



- 创建对象时使用计算的属性名，而不要在创建对象后使用对象的动态特性

这样可以在同一个位置定义对象的所有属性。

```
1 function getKey(k) {
2   return `a key named ${k}`;
3 }
4
5
6   const obj = {
7     id: 5,
8     name: 'San Francisco'
9   };
10  obj[getKey('enabled')] = true;
11
12
13  const obj = {
14    id: 5,
15    name: 'San Francisco',
16    [getKey('enabled')]: true
17 };
```

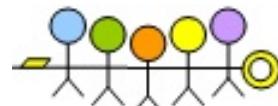
- 使用定义对象方法的简短形式

```
1 const atom = {
2   value: 1,
3
4   addValue: function (value) {
5     return atom.value + value;
6   }
7 };
8
9
10 const atom = {
11   value: 1,
12
13   addValue(value) {
14     return atom.value + value;
15   }
16 };
17 };
```

- 使用定义对象属性的简短形式

书写起来更加简单，并且可以自描述。

```
1 const lukeSkywalker = 'Luke Skywalker';
2
3
4   const obj = {
5     lukeSkywalker: lukeSkywalker
6   };
7 }
```



```
8
9   const obj = {
10    lukeSkywalker
11  };
```

- 将所有简写的属性写在对象定义的最顶部

这样可以更加方便地知道哪些属性使用了简短形式。

```
1  const anakinSkywalker = 'Anakin Skywalker';
2  const lukeSkywalker = 'Luke Skywalker';
3
4
5  const obj = {
6    episodeOne: 1,
7    twoJedisWalkIntoACantina: 2,
8    lukeSkywalker,
9    episodeThree: 3,
10   mayTheFourth: 4,
11   anakinSkywalker
12 };
13
14
15  const obj = {
16    lukeSkywalker,
17    anakinSkywalker,
18    episodeOne: 1,
19    twoJedisWalkIntoACantina: 2,
20    episodeThree: 3,
21    mayTheFourth: 4
22 };
```

数组

- 使用字面量语法创建数组

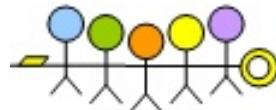
```
1
2  const items = new Array();
3
4
5  const items = [];
```

- 如果你不知道数组的长度，使用 push

```
1  const someStack = [];
2
3
4
5  someStack[someStack.length] = 'abracadabra';
6
7
8  someStack.push('abracadabra');
```

- 使用 ... 来拷贝数组

```
1
2  const len = items.length;
```



```

3 const itemsCopy = [];
4 let i;
5
6 for (i = 0; i < len; i++) {
7   itemsCopy[i] = items[i];
8 }
9
10
11 const itemsCopy = [...items];

```

- 使用 `Array.from` 将类数组对象转换为数组

```

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

```

解构 Destructuring

- 访问或使用对象的多个属性时请使用对象的解构赋值

解构赋值避免了为这些属性创建临时变量或对象。

```

1
2 function getFullName(user) {
3   const firstName = user.firstName;
4   const lastName = user.lastName;
5
6   return `${firstName} ${lastName}`;
7 }
8
9
10 function getFullName(obj) {
11   const { firstName, lastName } = obj;
12   return `${firstName} ${lastName}`;
13 }
14
15
16 function getFullName({ firstName, lastName }) {
17   return `${firstName} ${lastName}`;
18 }

```

- 使用数组解构赋值

```

1 const arr = [1, 2, 3, 4];
2
3
4 const first = arr[0];
5 const second = arr[1];
6
7
8 const [first, second] = arr;

```

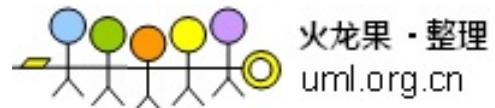
- 函数有多个返回值时使用对象解构，而不是数组解构

这样你就可以随时添加新的返回值或任意改变返回值的顺序，而不会导致调用失败。

```

1  function processInput(input) {
2      return [left, right, top, bottom];
3  }
4
5
6
7  const [left, __, top] = processInput(input);
8
9
10 function processInput(input) {
11     return { left, right, top, bottom };
12 }
13
14
15
16 const { left, right } = processInput(input);

```



字符串

- 使用单引号 ''

```

1 var name = "Bob Parr";
2
3
4
5 var name = 'Bob Parr';
6
7
8 var fullName = "Bob " + this.lastName;
9
10
11 var fullName = 'Bob ' + this.lastName;

```

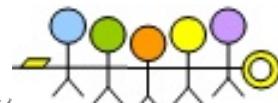
- 超过80个字符的字符串应该使用字符串连接换行
- 注：如果过度使用长字符串连接可能会影响性能。[jsPerf & Discussion](#)

```

1 var errorMessage = 'This is a super long error that was thrown because of
2 Batman. When you stop to think about how Batman had anything to do with
3 this, you would get nowhere fast.';
4
5
6 var errorMessage = 'This is a super long error that \
7 was thrown because of Batman. \
8 When you stop to think about \
9 how Batman had anything to do \
10 with this, you would get nowhere \
11 fast.';
12
13
14
15 var errorMessage = 'This is a super long error that ' +
16   'was thrown because of Batman.' +
17   'When you stop to think about ' +
18   'how Batman had anything to do ' +
19   'with this, you would get nowhere ' +
20   'fast.';

```

- 编程构建字符串时，使用字符串模板而不是字符串连接



模板给你一个可读的字符串，简洁的语法与适当的换行和`

```

1 function sayHi(name) {
2   return 'How are you, ' + name + '?';
3 }
4
5
6
7 function sayHi(name) {
8   return ['How are you, ', name, '?'].join();
9 }
10
11
12 function sayHi(name) {
13   return `How are you, ${name}?`;
14 }
```

函数

- 使用函数声明而不是函数表达式

函数声明拥有函数名，在调用栈中更加容易识别。并且，函数声明会整体提升，而函数表达式只会提升变量本身。这条规则也可以这样描述，始终使用箭头函数来代替函数表达式。

```

1 const foo = function () {
2 };
3
4
5
6 function foo() {
7 }
```

- 函数表达式

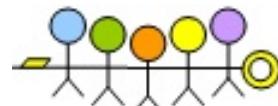
```

1 (( ) => {
2   console.log('Welcome to the Internet. Please follow me.');
3 })();
```

- 绝对不要在一个非函数块（if, while, 等等）里声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但是它们解析不同
- 注：ECMA-262 把块定义为一组语句，函数声明不是一个语句。阅读 [ECMA-262](#) 对这个问题的说明

```

1 if (currentUser) {
2   function test() {
3     console.log('Nope.');
4   }
5 }
6
7 }
```



```
8
9 if (currentUser) {
10   var test = function test() {
11     console.log('Yup.');
12   };
13 }
```

- 绝对不要把参数命名为 `arguments`, 这将会覆盖函数作用域内传过来的 `arguments` 对象

```
1
2 function nope(name, options, arguments) {
3
4 }
5
6
7 function yup(name, options, args) {
8
9 }
```

- 永远不要使用 `arguments`, 使用 `...` 操作符来代替

`...` 操作符可以明确指定你需要哪些参数, 并且得到的是一个真实的数组, 而不是 `arguments` 这样的类数组对象。

```
1
2 function concatenateAll() {
3   const args = Array.prototype.slice.call(arguments);
4   return args.join('');
5 }
6
7
8 function concatenateAll(...args) {
9   return args.join('');
10 }
```

- 使用函数参数默认值语法, 而不是修改函数的实参

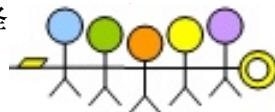
箭头函数 Arrow Functions

- 当必须使用函数表达式时（例如传递一个匿名函数时）, 请使用箭头函数

箭头函数提供了更简洁的语法, 并且箭头函数中 `this` 对象的指向是不变的, `this` 对象绑定定义时所在的对象, 这通常是我们想要的。如果该函数的逻辑非常复杂, 请将该函数提取为一个函数声明。

```
1
2 [1, 2, 3].map(function (x) {
3   return x * x;
4 });
5
6
7 [1, 2, 3].map((x) => {
8   return x * x
9 });
```

- 总是用括号包裹参数，省略括号只适用于单个参数，并且还降



```

1 [1, 2, 3].map(x => x * x);
2
3 [1, 2, 3].map((x) => x * x);
4
5

```

构造函数

- 总是使用 `class` 关键字，避免直接修改 `prototype`

`class` 语法更简洁，也更易理解。

```

1 function Queue(contents = []) {
2     this._queue = [...contents];
3 }
4 Queue.prototype.pop = function() {
5     const value = this._queue[0];
6     this._queue.splice(0, 1);
7     return value;
8 }
9
10
11
12
13 class Queue {
14     constructor(contents = []) {
15         this._queue = [...contents];
16     }
17     pop() {
18         const value = this._queue[0];
19         this._queue.splice(0, 1);
20         return value;
21     }
22 }

```

- 使用 `extends` 关键字来继承

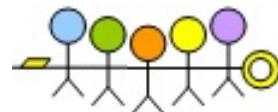
这是一个内置的继承方式，并且不会破坏 `instanceof` 原型检查。

```

1
2     const inherits = require('inherits');
3     function PeekableQueue(contents) {
4         Queue.apply(this, contents);
5     }
6     inherits(PeekableQueue, Queue);
7     PeekableQueue.prototype.peek = function() {
8         return this._queue[0];
9     }
10
11
12     class PeekableQueue extends Queue {
13         peek() {
14             return this._queue[0];
15         }
16     }

```

- 在方法中返回 `this` 以方便链式调用



```

1  Jedi.prototype.jump = function() {
2      this.jumping = true;
3      return true;
4  };
5
6
7  Jedi.prototype.setHeight = function(height) {
8      this.height = height;
9  };
10
11 const luke = new Jedi();
12 luke.jump();
13 luke.setHeight(20);
14
15
16 class Jedi {
17     jump() {
18         this.jumping = true;
19         return this;
20     }
21
22     setHeight(height) {
23         this.height = height;
24         return this;
25     }
26 }
27
28 const luke = new Jedi();
29
30 luke.jump()
31     .setHeight(20);

```

- 可以写一个自定义的`toString()`方法，但是确保它工作正常并且不会有副作用

```

1  class Jedi {
2     constructor(options = {}) {
3         this.name = options.name || 'no name';
4     }
5
6     getName() {
7         return this.name;
8     }
9
10    toString() {
11        return `Jedi - ${this.getName()}`;
12    }
13 }

```

模块

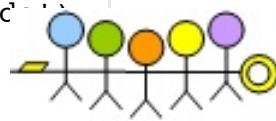
- 总是在非标准的模块系统中使用标准的 `import` 和 `export` 语法，我们总是可以将标准的模块语法转换成支持特定模块加载器的语法。

模块是未来的趋势，那么我们为何不在现在就开始使用。

```

2 const AirbnbStyleGuide = require('./AirbnbStyleGuide');
3 module.exports = AirbnbStyleGuide.es6;
4
5
6 import AirbnbStyleGuide from './AirbnbStyleGuide';
7 export default AirbnbStyleGuide.es6;
8
9
10 import { es6 } from './AirbnbStyleGuide';
11 export default es6;

```



- 不要使用通配符 * 的 import

这样确保了只有一个默认的 export 项

```

1 import * as AirbnbStyleGuide from './AirbnbStyleGuide';
2
3
4 import AirbnbStyleGuide from './AirbnbStyleGuide';
5

```

- 不要直接从一个 import 上 export

虽然一行代码看起来更简洁，但是有一个明确的 import 和一个明确的 export 使得代码行为更加明确。

```

1
2
3 export default { es6 } from './airbnbStyleGuide';
4
5
6
7 import { es6 } from './AirbnbStyleGuide';
8 export default es6;

```

Iterators 和 Generators

- 不要使用迭代器（Iterators）。优先使用 JavaScript 中 map 和 reduce 这类高阶函数来代替 for-of 循环

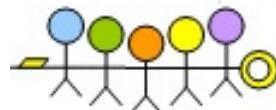
处理纯函数的返回值更加容易并且没有副作用

```

1 const numbers = [1, 2, 3, 4, 5];
2
3
4 let sum = 0;
5 for (let num of numbers) {
6     sum += num;
7 }
8
9 sum === 15;
10
11
12 let sum = 0;
13 numbers.forEach((num) => sum += num);
14 sum === 15;

```

```
15  
16     const sum = numbers.reduce((total, num) => total  
17     sum === 15;
```



- 不要使用 generators

它们不太容易转换为 ES5 的语法。

属性

- 使用点 . 操作符来访问属性

```
1 const luke = {  
2   jedi: true,  
3   age: 28  
4 };  
5  
6 const isJedi = luke['jedi'];  
7  
8 const isJedi = luke.jedi;
```

- 当使用变量访问属性时使用中括号 []

```
1 var luke = {  
2   jedi: true,  
3   age: 28  
4 };  
5  
6 function getProp(prop) {  
7   return luke[prop];  
8 }  
9  
10 var isJedi = getProp('jedi');
```

变量

- 总是使用 const 来声明变量，否则将生成全局变量，我们应该避免污染全局命名空间

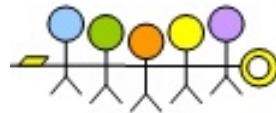
```
1 superPower = new SuperPower();  
2  
3  
4 const superPower = new SuperPower();  
5
```

- 为每个变量都使用 const 关键字声明

这种方式更加容易添加新变量，并且不必担忧将 `const` 错误写成 `const`；而导致生成全局变量。

```
1 const items = getItems(),  
2
```

```
3     goSportsTeam = true,  
4     dragonball = 'z';  
5  
6  
7  
8 const items = getItems(),  
9     goSportsTeam = true;  
10    dragonball = 'z';  
11  
12  
13 const items = getItems();  
14 const goSportsTeam = true;  
15 const dragonball = 'z';
```



- 将所有 `const` 变量放在一起，然后将所有 `let` 变量放在一起

```
1 let i, len, dragonball,  
2     items = getItems(),  
3     goSportsTeam = true;  
4  
5  
6  
7 let i;  
8 let items = getItems();  
9 let dragonball;  
10 let goSportsTeam = true;  
11 let len;  
12  
13  
14 const goSportsTeam = true;  
15 const items = getItems();  
16 let dragonball;  
17 let i;  
18 let length;
```

- 在必要的时候声明变量，并且将其放在合适的位置

`let` 和 `const` 是块级作用域的，而不是函数作用域。

```
1 function() {  
2     test();  
3     console.log('doing stuff...');  
4  
5  
6  
7  
8     const name = getName();  
9  
10    if (name === 'test') {  
11        return false;  
12    }  
13  
14    return name;  
15 }  
16  
17  
18 function() {  
19     const name = getName();  
20 }
```



```
21     if (!arguments.length) {
22         return false;
23     }
24
25     return true;
26 }
27
28
29 function() {
30     if (!arguments.length) {
31         return false;
32     }
33
34     const name = getName();
35
36     return true;
37 }
```

变量提升 Hoisting

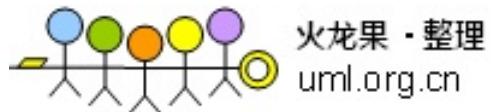
- 通过 `var` 声明的变量将被提升到作用域的顶部，但他们的赋值不会被提升。通过 `const` 和 `let` 声明的变量不存在变量提升，这里有一个新概念，称为“[暂时性死区（Temporal Dead Zones \(TDZ\)）](#)”。有必要理解 `typeof` 不再是一个百分之百安全的操作。
- 匿名函数表达式提升了对应的变量名，但赋值过程没有被提升

```
1 function example() {
2     console.log(anonymous);
3
4     anonymous();
5
6     let anonymous = function() {
7         console.log('anonymous function expression');
8     };
9 }
```

- 命名的函数表达式提升了对应的变量名，函数名和函数体没有被提升

```
1 function example() {
2     console.log(named);
3
4     named();
5
6     superPower();
7
8     var named = function superPower() {
9         console.log('Flying');
10    };
11 }
12
13
14
15 function example() {
16     console.log(named);
17
18     named();
19
20     var named = function named() {
```

```
21     console.log('named');
22 }
23 }
```



- 函数声明将被提升

```
1 function example() {
2   superPower();
3
4   function superPower() {
5     console.log('Flying');
6   }
7 }
```

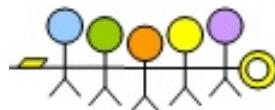
- 更多细节可以参考 [Ben Cherry 的 JavaScript Scoping & Hoisting](#)

比较运算符和等号

- 使用 `==` 和 `!=` 而不是 `=` 和 `!=`
- 比较运算通过 `ToBoolean` 强制转换并遵循一下规则：
 - `Object - true`
 - `Undefined - false`
 - `Null - false`
 - `Booleans - 被转换为对应的值`
 - `Number - 值为 +0, -0, NaN 时为 false, 否则为 true`
 - `String - 空字符串 '' 为 false, 否则为 true`
- 使用快捷方式

```
1 if (name !== '') {
2
3 }
4
5
6
7 if (name) {
8
9 }
10
11
12 if (collection.length > 0) {
13
14 }
15
16
17 if (collection.length) {
18
19 }
```

- 更多细节请阅读 [Truth Equality and JavaScript](#)



- 给所有多行的块使用大括号

```

1  if (test)
2      return false;
3
4
5
6  if (test) return false;
7
8
9  if (test) {
10     return false;
11 }
12
13
14 function() { return false; }
15
16
17 function() {
18     return false;
19 }
```

- 使用 `if...else` 这样的多行块时，请将 `else` 和 `if` 的结束括号放在同一行

```

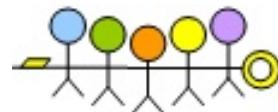
1  if (test) {
2      thing1();
3      thing2();
4  }
5  else {
6      thing3();
7  }
8
9
10
11 if (test) {
12     thing1();
13     thing2();
14 } else {
15     thing3();
16 }
```

注释

- 使用 `/** ... */` 进行多行注释，包括描述，指定类型以及参数值和返回值

```

1
2
3
4
5
6
7  function make(tag) {
8
9
10    return element;
```



```
12 }
13
14
15
16 * make() returns a new element
17 * based on the passed in tag name
18 *
19 * @param <String> tag
20 * @return <Element> element
21 */
22 function make(tag) {
23
24
25
26     return element;
27 }
```

- 使用 // 进行单行注释，将注释放在被注释对象的上面，并在注释之前保留一个空行

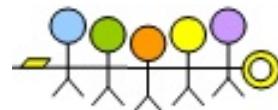
```
1 const active = true;
2
3
4
5
6 const active = true;
7
8
9 function getType() {
10     console.log('fetching type...');
11
12     const type = this._type || 'no type';
13
14     return type;
15 }
16
17
18 function getType() {
19     console.log('fetching type...');
20
21     const type = this._type || 'no type';
22
23     return type;
24 }
25 }
```

- 使用 // FIXME: 来注释一个问题

```
1 function Calculator() {
2
3
4     total = 0;
5
6     return this;
7 }
```

- 使用 // TODO: 来注释一个问题的解决方案

```
1 function Calculator() {
2 }
```



```
3     this.total = 0;
4
5     return this;
6 }
7 }
```

空白

- 将 tab 设置为 2 个空格缩进

```
1
2 function() {
3   ...const name;
4 }
5
6
7 function() {
8   ·const name;
9 }
10
11
12 function() {
13   ..const name;
14 }
```

- 前大括号前放置一个空格

```
1
2 function test(){
3   console.log('test');
4 }
5
6
7 function test() {
8   console.log('test');
9 }
10
11
12 dog.set('attr',{
13   age: '1 year',
14   breed: 'Bernese Mountain Dog'
15 });
16
17
18 dog.set('attr', {
19   age: '1 year',
20   breed: 'Bernese Mountain Dog'
21 });
```

- 运算符之间用空格分隔

```
1
2 const x=y+5;
3
4
5 const x = y + 5;
```

- 文件末尾使用单个换行符



```
1 (function(global) {  
2   })(this);
```

```
1 (function(global) {  
2   })(this);  
3   ↵  
4   ↵
```

```
1 (function(global) {  
2   })(this);  
3   ↵  
4   ↵
```

- 方法链式调用时保持适当的缩进，并且使用前置的`.`来表示该行是一个方法调用，而不是一个新语句

```
1  $('#items').find('.selected').highlight().end().find('.open').updateCount();  
2  
3  $('#items').  
4    find('selected').  
5      highlight().  
6      end().  
7    find('.open').  
8      updateCount();  
9  
10 $('#items')  
11   .find('.selected')  
12     .highlight()  
13     .end()  
14   .find('.open')  
15     .updateCount();  
16  
17  
18  
19 const leds =  
20 stage.selectAll('.led').data(data).enter().append('svg:svg').class('led',  
21 true)  
22   .attr('width', (radius + margin) * 2).append('svg:g')  
23   .attr('transform', 'translate(' + (radius + margin) + ',' + (radius +  
24 margin) + ')')  
25   .call(tron.led);  
26  
27  
28 const leds = stage.selectAll('.led')  
29   .data(data)  
30   .enter().append('svg:svg')  
31     .class('led', true)  
32     .attr('width', (radius + margin) * 2)  
33     .append('svg:g')  
34       .attr('transform', 'translate(' + (radius + margin) + ',' + (radius +  
margin) + ')')  
35       .call(tron.led);
```

- 在语句块之后和下一语句之前都保持一个空行



```
1 if (foo) {  
2     return bar;  
3 }  
4 return baz;  
5  
6  
7 if (foo) {  
8     return bar;  
9 }  
10  
11 return baz;  
12  
13  
14 const obj = {  
15     foo: function() {  
16         },  
17         bar: function() {  
18             }  
19     };  
20     return obj;  
21  
22  
23 const obj = {  
24     foo: function() {  
25         },  
26         bar: function() {  
27             }  
28     };  
29     return obj;  
30  
31  
32 return obj;
```

逗号

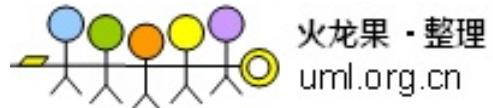
- 不要将逗号放前面

```
1 const story = [  
2     once  
3     , upon  
4     , aTime  
5 ];  
6  
7  
9 const story = [  
10    once,  
11    upon,  
12    aTime  
13 ];  
14  
15  
16 const hero = {  
17     firstName: 'Bob'  
18     , lastName: 'Parr'  
19     , heroName: 'Mr. Incredible'  
20     , superPower: 'strength'  
21 };  
22  
23
```

```

24 const hero = {
25   firstName: 'Bob',
26   lastName: 'Parr',
27   heroName: 'Mr. Incredibile',
28   superPower: 'strength'
29 };

```



- 不要添加多余的逗号，否则将在 IE6/7 和 IE9 的怪异模式下导致错误。同时，某些 ES3 的实现会计算多数组的长度，这在 ES5 中有[澄清](#)

```

1
2   const hero = {
3     firstName: 'Kevin',
4     lastName: 'Flynn',
5   };
6
7   const heroes = [
8     'Batman',
9     'Superman',
10 ];
11
12
13   const hero = {
14     firstName: 'Kevin',
15     lastName: 'Flynn'
16   };
17
18   const heroes = [
19     'Batman',
20     'Superman'
21 ];

```

分号

- 句末一定要添加分号

```

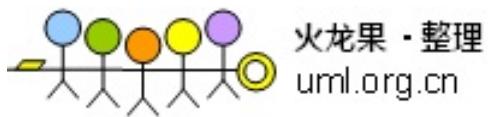
1
2   (function() {
3     const name = 'Skywalker'
4     return name
5   })()
6
7
8   (() => {
9     const name = 'Skywalker';
10    return name;
11  })();
12
13
14 ;(() => {
15   const name = 'Skywalker';
16   return name;
17 })();

```

类型转换

- 在语句的开始执行类型转换

- 字符串：



```
1
2
3
4 const totalScore = this.reviewScore + '';
5
6
7 const totalScore = String(this.reviewScore);
```

- 对数字使用 parseInt 并且总是带上类型转换的基数

```
1 const inputValue = '4';
2
3
4 const val = new Number(inputValue);
5
6
7 const val = +inputValue;
8
9
10 const val = inputValue >> 0;
11
12
13 const val = parseInt(inputValue);
14
15
16 const val = Number(inputValue);
17
18
19 const val = parseInt(inputValue, 10);
```

- 不管是出于一些奇特的原因，还是

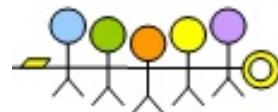
parseInt 是一个瓶颈而需要位运算来解决某些性能问题，请为你的代码注释为什么要这样做

```
1
2
3 * parseInt was the reason my code was slow.
4 * Bitshifting the String to coerce it to a
5 * Number made it a lot faster.
6 */
7 const val = inputValue >> 0;
```

- 注意：使用位移运算时要特别小心。Number 在 JavaScript 中表示为 64 位的值，但位移运算总是返回一个 32 位的整数（source），对大于 32 位的整数进行位移运算会导致意外的结果（讨论）。32 位最大整数为 2,147,483,647：

- 布尔值

```
1 var age = 0;
2
3
4 var hasAge = new Boolean(age);
5
```



```
6 var hasAge = Boolean(age);  
7  
8  
9  
10 var hasAge = !!age;
```

命名约定

- 避免单个字符名，让你的变量名有描述意义

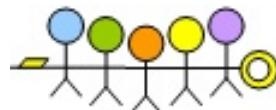
```
1 function q() {  
2  
3 }  
4  
5 function query() {  
6  
7 }  
8  
9 }
```

- 命名对象、函数和实例时使用小驼峰命名规则

```
1 var OBJEcttsssss = {};  
2 var this_is_my_object = {};  
3 var this-is-my-object = {};  
4 function c() {};  
5 var u = new user({  
6   name: 'Bob Parr'  
7 });  
8  
9  
10  
11 var thisIsMyObject = {};  
12 function thisIsMyFunction() {};  
13 var user = new User({  
14   name: 'Bob Parr'  
15 });
```

- 命名构造函数或类时使用大驼峰命名规则

```
1 function user(options) {  
2   this.name = options.name;  
3 }  
4  
5 const bad = new user({  
6   name: 'nope'  
7 });  
8  
9  
10  
11 class User {  
12   constructor(options) {  
13     this.name = options.name;  
14   }  
15 }  
16  
17 const good = new User({  
18   name: 'yup'
```



- 命名私有属性时前面加个下划线 _

```

1  this.__firstName__ = 'Panda';
2  this.firstName_ = 'Panda';
3
4
5
6  this._firstName = 'Panda';

```

- 保存对 this 的引用时使用 _this

```

1
2  function() {
3      var self = this;
4      return function() {
5          console.log(self);
6      };
7 }
8
9
10 function() {
11     var that = this;
12     return function() {
13         console.log(that);
14     };
15 }
16
17
18 function() {
19     var _this = this;
20     return function() {
21         console.log(_this);
22     };
23 }

```

- 导出单一一个类时，确保你的文件名就是你的类名

```

1
2 class CheckBox {
3
4 }
5 module.exports = CheckBox;
6
7
8
9 const CheckBox = require('./checkBox');
10
11
12 const CheckBox = require('./check_box');
13
14
15 const CheckBox = require('./CheckBox');

```

- 导出一个默认小驼峰命名的函数时，文件名应该就是导出的方法名

```

1 function makeStyleGuide() {

```

```
2 }
3
4 export default makeStyleGuide;
```



- 导出单例、函数库或裸对象时，使用大驼峰命名规则

```
1 const AirbnbStyleGuide = {
2   es6: {
3     }
4   };
5
6 export default AirbnbStyleGuide;
```

访问器

- 属性的访问器函数不是必须的
- 如果你确实有存取器函数的话使用 `getVal()` 和 `setVal('hello')`

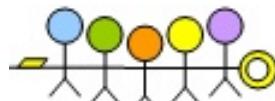
```
1 dragon.age();
2
3
4 dragon.getAge();
5
6
7 dragon.age(25);
8
9
10 dragon.setAge(25);
```

- 如果属性是布尔值，使用 `isValid()` 或 `hasVal()`

```
1 if (!dragon.age()) {
2   return false;
3 }
4
5
6
7 if (!dragon.hasAge()) {
8   return false;
9 }
```

- 可以创建`get()`和`set()`函数，但是要保持一致性

```
1 function Jedi(options) {
2   options || (options = {});
3   var lightsaber = options.lightsaber || 'blue';
4   this.set('lightsaber', lightsaber);
5 }
6
7 Jedi.prototype.set = function(key, val) {
8   this[key] = val;
9 };
10
11 Jedi.prototype.get = function(key) {
12   return this[key];
13 };
```



- 当给事件附加数据时，传入一个哈希而不是原始值，这可以让事件数据里而不用找出并更新那个事件的事件处理器

```

1 $(this).trigger('listingUpdated', listing.id);
2 ...
3
4
5
6 $(this).on('listingUpdated', function(e, listingId) {
7
8 });

```

```

1 $(this).trigger('listingUpdated', { listingId : listing.id });
2 ...
3
4
5
6 $(this).on('listingUpdated', function(e, data) {
7
8 });

```

jQuery

- 为 jQuery 对象命名时添加 \$ 前缀

```

1 const sidebar = $('.sidebar');
2
3
4
5 const $sidebar = $('.sidebar');

```

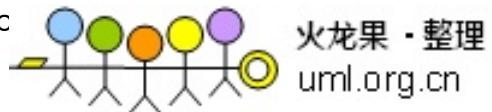
- 缓存 jQuery 的查询结果

```

1
2 function setSidebar() {
3     $('.sidebar').hide();
4
5
6
7     $('.sidebar').css({
8         'background-color': 'pink'
9     });
10 }
11
12
13 function setSidebar() {
14     const $sidebar = $('.sidebar');
15     $sidebar.hide();
16
17
18
19     $sidebar.css({
20         'background-color': 'pink'
21     });
22 }

```

- 对DOM查询使用级联的 `$('.sidebar ul')` 或 `$('.sideb`
- 在指定作用域进行查询时使用 `find`



```

1  $('ul', '.sidebar').hide();
2
3
4
5  $('.sidebar').find('ul').hide();
6
7
8  $('.sidebar ul').hide();
9
10
11 $('.sidebar > ul').hide();
12
13
14 $sidebar.find('ul').hide();

```

ECMAScript 5 兼容性

- 参考 Kangax 的 [ES5 compatibility table](#)

ECMAScript 6 新特性

下面是本文涉及到的 ES6 新特性：

性能

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

资源

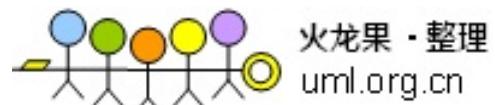
Read This

- [Annotated ECMAScript 5.1](#)

工具

- Code Style Linters
 - [JSHint - Airbnb Style jshintrc](#)
 - [JSCS - Airbnb Style Preset](#)

其它规范



火龙果 · 整理
uml.org.cn

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

其它风格

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on Github - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

更多文章

- Understanding JavaScript Closures - Angus Croll
- Basic JavaScript for the impatient programmer - Dr. Axel Rauschmayer
- You Might Not Need jQuery - Zack Bloom & Adam Schwartz
- ES6 Features - Luke Hoban
- Frontend Guidelines - Benjamin De Cock

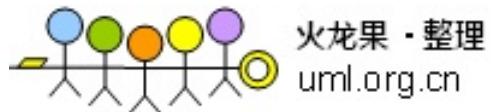
书籍

- JavaScript: The Good Parts - Douglas Crockford
- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch
- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- Human JavaScript - Henrik Joreteg
- Superhero.js - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
- JSBooks - Julien Bouquillon
- Third Party JavaScript - Ben Vinegar and Anton Kovalyov
- Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript - David Herman

播客

- DailyJS

- JavaScript Weekly
- JavaScript, JavaScript...
- Bocoup Weblog
- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- Dustin Diaz
- nettuts



火龙果 · 整理
uml.org.cn

Podcasts

- JavaScript Jabber