JavaScript规范

类型

1. **基本类型: 你可以直接获取到基本类型的值**

    a. string
    b. number
    c. boolean
    d. null
    e. undefined
    f. symbol

JavaScript 复制代码

```JavaScript
const foo=1;
let bar=foo;
bar=9;
console.log(foo,bar); // => 1,9
```

2. **复杂类型: 复杂类型赋值是获取到他的引用的值。 相当于传引用**

    a. object
    b. array
    c. function

JavaScript 复制代码

```JavaScript
const foo = [1,2];
const bar = foo;
bar[0] = 9;
console.log(foo[0],bar[0]); // =>9,9
```

引用

1. **所有的赋值都用const，避免使用var.** eslint: prefer-const, no-const-assign

```JavaScript
// bad
var a = 1;
var b = 2;

// good
const a = 1;
const b = 2;
```

说明：确保初始值不被改变，避免重复引用导致bug，让代码更容易理解

2. 如果一定要对参数重新赋值，请使用let，而不是var. eslint: no-var

```JavaScript
// bad
var count = 1;
if (true) {
  count += 1;
}

// good, use the let.
let count = 1;
if (true) {
  count += 1;
}
```

说明：let是块级作用域，而var是函数级作用域

3. let、const都是块级作用域

```JavaScript
// const 和 let 都只存在于它定义的那个块级作用域
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

对象

1. 使用字面值创建对象. eslint: no-new-object

JavaScript | 复制代码

```javascript
// bad
const item = new Object();

// good
const item = {};
```

2. 当创建一个带有动态属性名的对象时，用计算后属性名

JavaScript | 复制代码

```javascript
function getKey(k) {
  return `a key named ${k}`;
}

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

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

说明：这样可以让创建对象的属性只出现在一个地方，而不是多处出现

3. 用属性值缩写. eslint: object-shorthand

```javascript
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

说明：让代码更简洁，可读性更强

4. 将所有缩写放在对象声明的开始

```javascript
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

说明：这样也是为了更方便的知道有哪些属性用了缩写

5. 对象浅拷贝时，更推荐使用扩展运算符[就是...运算符]，而不是Object.assign。获取对象指定的几个属性时，用对象的rest解构运算符[也是...运算符]更好

JavaScript 复制代码

```
// very bad
const original = { a: 1, b: 2 };
const copy = Object.assign(original, { c: 3 }); // this mutates `original` ಠ_ಠ
delete copy.a; // so does this

// bad
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }

// good es6扩展运算符 ...
const original = { a: 1, b: 2 };
// 浅拷贝
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }

// rest 赋值运算符
const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

数组

1. 用字面量赋值。 eslint: no-array-constructor

JavaScript 复制代码

```
// bad
const items = new Array();

// good
const items = [];
```

2. 用Array#push 代替直接向数组中添加一个值。

JavaScript 复制代码

```
const someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

3. 用扩展运算符做数组浅拷贝，类似上面的对象浅拷贝

JavaScript 复制代码

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i += 1) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

4. 用 Array.from 去将一个类数组对象转成一个数组

JavaScript 复制代码

```
const arrLike = { 0: 'foo', 1: 'bar', 2: 'baz', length: 3 };

// bad
const arr = Array.prototype.slice.call(arrLike);

// good
const arr = Array.from(arrLike);
```

5. 用 Array.from 而不是 ... 运算符去做map遍历。 因为这样可以避免创建一个临时数组

JavaScript 复制代码

```
// bad
const baz = [...foo].map(bar);

// good
const baz = Array.from(foo, bar);
```

解构

1. 用对象的解构赋值来获取和使用对象某个或多个属性值

JavaScript 复制代码

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

  return `${firstName} ${lastName}`;
}

// good
function getFullName(user) {
  const { firstName, lastName } = user;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

2. 用数组解构

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

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

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

3. 多个返回值用对象的解构，而不是数据解构

```JavaScript
// bad
function processInput(input) {
  // 然后就是见证奇迹的时刻
  return [left, right, top, bottom];
}

// 调用者需要想一想返回值的顺序
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // oops， 奇迹又发生了
  return { left, right, top, bottom };
}

// 调用者只需要选择他想用的值就好了
const { left, top } = processInput(input);
```

说明：你可以在后期添加新的属性或者变换变量的顺序而不会打破原有的调用

字符串

1. 用字符串模板而不是字符串拼接来组织可编程字符串

JavaScript 复制代码

```javascript
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// bad
function sayHi(name) {
  return `How are you, ${ name }?`;
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

说明：模板字符串更具可读性、语法简洁

2. 永远不要在字符串中用eval()，他就是潘多拉盒子

3. 不要使用不必要的转义字符

JavaScript 复制代码

```javascript
// bad
const foo = '\'this\' \i\s \"quoted\"';

// good
const foo = '\'this\' is "quoted"';

//best
const foo = `my name is '${name}'`;
```

说明：反斜线可读性差，所以他们只在必须使用时才出现

函数

1. 不要用arguments命名参数。他的优先级高于每个函数作用域自带的 arguments 对象， 这会导致函数自带的 arguments 值被覆盖

JavaScript 复制代码

```
// bad
function foo(name, options, arguments) {
  // ...
}

// good
function foo(name, options, args) {
  // ...
}
```

2. 不要使用arguments，用rest语法...代替

JavaScript 复制代码

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

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

说明：...明确你想用那个参数。而且rest参数是真数组，而不是类似数组的arguments

3. 用默认参数语法而不是在函数里对参数重新赋值

JavaScript 复制代码

```
// really bad
function handleThings(opts) {
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

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

4. 把默认参数赋值放在最后

JavaScript 复制代码

```
// bad
function handleThings(opts = {}, name) {
  // ...
}

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

5. 不要改参数

JavaScript 复制代码

```javascript
// bad
function f1(obj) {
  obj.key = 1;
};

// good
function f2(obj) {
  const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key : 1;
};
```

说明：操作参数对象对原始调用者会导致意想不到的副作用。 就是不要改参数的数据结构，保留参数原始值和数据结构

6. 不要对参数重新赋值

JavaScript 复制代码

```javascript
// bad
function f1(a) {
  a = 1;
  // ...
}

function f2(a) {
  if (!a) { a = 1; }
  // ...
}

// good
function f3(a) {
  const b = a || 1;
  // ...
}

function f4(a = 1) {
  // ...
}
```

箭头函数

略

类

1. 常用class，避免直接操作prototype

JavaScript | 复制代码

```javascript
// bad
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;
  }
}
```

说明：class语法更简洁更易理解

2. 如果没有具体说明，类有默认的构造方法。一个空的构造函数或只是代表父类的构造函数是不需要写的

JavaScript | 复制代码

```javascript
// bad
class Jedi {
  constructor() {}

  getName() {
    return this.name;
  }
}

// bad
class Rey extends Jedi {
  // 这种构造函数是不需要写的
  constructor(...args) {
    super(...args);
  }
}

// good
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
    this.name = 'Rey';
  }
}
```

3. 避免重复类成员

JavaScript 复制代码

```
// bad
class Foo {
  bar() { return 1; }
  bar() { return 2; }
}

// good
class Foo {
  bar() { return 1; }
}

// good
class Foo {
  bar() { return 2; }
}
```

说明：重复类成员会默默的执行最后一个 —— 重复本身也是一个bug

模块

1. 用(import/export) 模块而不是无标准的模块系统

JavaScript 复制代码

```
// bad
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;

// best
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

2. 不要用import通配符， 就是 * 这种方式

```JavaScript
// bad
import * as AirbnbStyleGuide from './AirbnbStyleGuide';

// good
import AirbnbStyleGuide from './AirbnbStyleGuide';
```

3. 不要导出可变的东西

```JavaScript
// bad
let foo = 3;
export { foo }

// good
const foo = 3;
export { foo }
```

4. 在一个单一导出模块里，用 export default

```JavaScript
// bad
export function foo() {}

// good
export default function foo() {}
```

说明：鼓励使用更多文件，每个文件只做一件事情并导出，这样可读性和可维护性更好

5. import 放在其他所有语句之前

变量

1. 用const或let声明变量，避免污染全局命名空间

JavaScript 复制代码

```
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

2. const放一起，let放一起

JavaScript 复制代码

```
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

比较操作

1. 三元表达式不应该嵌套，通常是单行表达式

JavaScript 复制代码

```
// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;

// better
const maybeNull = value1 > value2 ? 'baz' : null;

const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull;

// best
const maybeNull = value1 > value2 ? 'baz' : null;

const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

2. 避免不需要的三元表达式

JavaScript 复制代码

```
// bad
const foo = a ? a : b;
const bar = c ? true : false;
const baz = c ? false : true;

// good
const foo = a || b;
const bar = !!c;
const baz = !c;
```

3. 用圆括号来混合多个操作符

JavaScript | 复制代码

```javascript
// bad
const foo = a && b < 0 || c > 0 || d + 1 === 0;

// bad
const bar = a ** b - 5 % d;

// bad
// 别人会陷入(a || b) && c 的迷惑中
if (a || b && c) {
  return d;
}

// good
const foo = (a && b < 0) || c > 0 || (d + 1 === 0);

// good
const bar = (a ** b) - (5 % d);

// good
if (a || (b && c)) {
  return d;
}

// good
const bar = a + b / c * d;
```

命名

1. **避免用一个字母命名，让你的命名可描述**

```javascript
// bad
function q() {
  // ...
}

// good
function query() {
  // ...
}
```

2. 用小驼峰式命名你的对象、函数、实例

```javascript
// bad
const OBJEcttssssss = {};
const this_is_my_object = {};
function c() {}

// good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

3. 用大驼峰式命名类

JavaScript 复制代码

```
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}

const good = new User({
  name: 'yup',
});
```

4. 不要用前置或后置下划线

JavaScript 复制代码

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';
this._firstName = 'Panda';

// good
this.firstName = 'Panda';
```

说明：JavaScript 没有私有属性或私有方法的概念

5. 不要保存引用this， 用箭头函数或函数绑定——Function#bind.

JavaScript | 复制代码

```javascript
// bad
function foo() {
  const self = this;
  return function () {
    console.log(self);
  };
}

// bad
function foo() {
  const that = this;
  return function () {
    console.log(that);
  };
}

// good
function foo() {
  return () => {
    console.log(this);
  };
}
```

6. export default导出模块A，则这个文件名也叫A.*， import 时候的参数也叫A，大小写完全一致

JavaScript | 复制代码

```javascript
// file 1 contents
class CheckBox {
  // ...
}
export default CheckBox;

// file 2 contents
export default function fortyTwo() { return 42; }

// file 3 contents
export default function insideDirectory() {}

// in some other file
// bad
import CheckBox from './checkBox'; // PascalCase import/export, camelCase filename
import FortyTwo from './FortyTwo'; // PascalCase import/filename, camelCase export
import InsideDirectory from './InsideDirectory'; // PascalCase import/filename, camelCase export

// bad
import CheckBox from './check_box'; // PascalCase import/export, snake_case filename
import forty_two from './forty_two'; // snake_case import/filename, camelCase export
import inside_directory from './inside_directory'; // snake_case import, camelCase export
import index from './inside_directory/index'; // requiring the index file explicitly
import insideDirectory from './insideDirectory/index'; // requiring the index file explicitly

// good
import CheckBox from './CheckBox'; // PascalCase export/import/filename
import fortyTwo from './fortyTwo'; // camelCase export/import/filename
import insideDirectory from './insideDirectory'; // camelCase export/import/directory name/implicit "index"
// ^ supports both insideDirectory.js and insideDirectory/index.js
```

7. 当你export-default一个函数时，函数名用小驼峰，文件名需要和函数名一致

JavaScript 复制代码

```
function makeStyleGuide() {
  // ...
}

export default makeStyleGuide;
```

8. 当你export一个结构体/类/单例/函数库/对象 时用大驼峰

JavaScript 复制代码

```
const AirbnbStyleGuide = {
  es6: {
  }
};

export default AirbnbStyleGuide;
```

9. 简称和缩写应该全部大写或全部小写（不要怕长）

JavaScript 复制代码

```
// bad
import SmsContainer from './containers/SmsContainer';

// bad
const HttpRequests = [
  // ...
];

// good
import SMSContainer from './containers/SMSContainer';

// good
const HTTPRequests = [
  // ...
];

// best
import TextMessageContainer from './containers/TextMessageContainer';

// best
const Requests = [
  // ...
];
```

jQuery

1. jQuery对象用$变量表示

JavaScript 复制代码

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');

// good
const $sidebarBtn = $('.sidebar-btn');
```

2. 暂存jQuery查找

JavaScript | 复制代码

```javascript
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

3. DOM查找用层叠式$('.sidebar ul') 或 父节点 > 子节点 $('.sidebar > ul')

4. 用jQuery对象查询作用域的find方法查询

JavaScript | 复制代码

```javascript
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```