I.格式说明

```强调部分```

bad: 不恰当的使用方式

good: 良好的使用方式

best: 最好的使用方式

II.代码规范

Types

1. 使用const运算符定义常量，避免用var声明
   ```这个确保了你不会重新给变量赋值， 这种情况下会导致复杂的代码中难以定位错误```

   bad: var a = 1;

   good: const a = 1;

2. 如果有重新赋值的需要，用let运算符取代var
   ```let是一个块级作用域的描述符```

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

   good:
   ```
   let count = 1;
       if (true) {
           count += 1;
       }
   ```

3. ```let 和 const描述符字义的变量/常量都是块级作用域的```

References

1. 使用字面量的方式定义对象或数组

   bad:
   ```
   var obj = new Object();
   var array = new Array();
   ```
   ```

```
good:
var obj = {};
var array = [];
```

Objects

1. 不要用保留字作为对象的key，在IE8下这种情况很可能不生效

```
bad:
var person = {
    default: 'john',
    private: '1234',
};

good:
var person = {
    default: 'john',
    secretCode: '1234',
};
```

2. 使用对象方法的简写形式

```
bad:
const counter = {
    value: 1,
    addValue: function (value) {
        return counter.value + value;
    },
};

good:
const counter = {
    value: 1,
    addValue(value) {
         return counter.value + value;
    },
};
```

3. 同上， 声明对象属性时使用缩写的形式
```这样更简洁并且更能表达变量的含义```

```
bad:
const myProperty = 'propertyValue';
const obj = {
    myProperty: myProperty,
};

good:
const myProperty = 'propertyValue';
```
```

```
const obj = {
    myProperty, // 这个逗号保留
};
```

4. 只在属性名是不合法的标识符时使用引号

```
bad:
const badCase = {
    'foo': 3,
    'bar': 4,
    'data-src': 5,
};

good:
const badCase = {
    foo: 3,
    bar: 4,
    'data-src': 5,
};
```

5. 不要直接调用Object.prototype的方法，缓存常用的方法，例如hasOwnProperty, propertyIsEnumerable, isPrototypeOf
```{hasOwnPropertype:false}, null对象就会报错```

```
bad:
console.log(object.hasOwnProperty(key));

good:
console.log(Object.hasOwnProperty.call(object, key));


best:
const has = Object.prototype.hasOwnProperty;
console.log(has.call(object, key));
```

Arrays

1. 使用字面量的形式命名

```
bad:
const listItems = new Array();

good:
const listItems = [];
```

2. 使用push方法添加数据

```
const stack = [];

bad:
```
```

```
stack[stack.length] = 'blablabla';
```

```
good:
stack.push('blablabla');
```

3. 使用剩余参数形式进行数组拷贝

```
bad:
const len = items.length;
const copiedArray = [];
let i;
for (i = 0; i < len; i++) {
    copiedArray[i] = items[i];
}
```

```
good:
const copiedArray = [...items]; // 这种拷贝方式实际上做的是一次浅拷贝
```

4. 用Array.from转化类数组对象

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

5. 在数组的回调方法中使用return语句

```
good:
[1, 2, 3].map((x) => {
    const y = x + 1;
    return x * y;
});
```

```
good:
[1, 2, 3].map(x => x + 1);
```

```
bad:
const flat = {};
[[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {
    const flattern = memo.concat(item);
    flat[index] = flatten;
});
```

```
good:
const flat = {};
[[0, 1], [2, 3], [4, 5]].reduce((memo, item, index) => {
    const flattern = memo.concat(item);
    flat[index] = flatten;
    return flatten;
});
```

```
bad:
inbox.filter((msg) => {
    const {subject, author} = msg;
    if (subject === 'Nodejs') {
        return author === 'Piaoling';
    } else {
        return false';
    }
});

good:
inbox.filter((msg) => {
    const {subject, author} = msg;
    if (subject === 'Nodejs') {
       return author === 'Piaoling';
    }

    return false;
});
```

Destructuring

1. 当要获取对象的多个属性时使用对象的析构解析

```
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. 使用数组的析构解析

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

bad:
const first = arr[0];
const second = arr[1];
```

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

3. 使用对象的析构解析取代数组的析构解析返回多个值，这样可以不用关注返回的顺序

```
bad:
function processInput (input) {
  // 函数内部的处理逻辑
  return [left, right, top, bottom];
}
```

这样取的时候比较麻烦，要关注顺序：`const [left, __, top] = processInput(input);`

```
good:
function processInput (input) {
  // 函数内部的处理逻辑
  return {left, right, top, bottom};
}
```

这样取的时候就可以用 `const {left, top} = processInput(input);`

Strings

1. 使用单引号声明string类型的变量

```
bad: const name = "John Doe";

good: const name = 'John Doe';
```

2. 超过100个字符的stirng变量要使用字符串连接的方式进行赋值

3. 如果过度使用， 过长的字符串会影响性能问题

```
bad: const errorMessage =
'blablablablablablablablablablablablablablablablablablablablablablablablablablablablablablablablablablablablabla';

bad: const errorMessage = 'blablablablablablablablablabla\
blablablablablablablablablablabla\
blablablablablablablablablablabla';

good: const errorMessage = 'blablablablablablablabla' +
                           'blablablablablablablabla' +
                           'blablablabla';
```

4. 如果要动态生成字符串，使用模板字符串代替字符串的拼接

```
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}?`;
}
```

5. ```不要对string使用eval函数，存在有太多漏洞```

6. 不要在不必要的时候在字符串中使用转义字符，反斜杠会导致不良好的可读性

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

good:
const foo = '\'this\' is "quoted"';
const foo = `'this' is "quoted"`;
```

Functions

1. 使用函数声明替代函数表达式，理由如下：
   a. 更容易识别函数的调用栈
   b. 通过函数声明方式定义函数，整个函数体会被提升，而函数表达式的方式只有引用的变量会被提升

```
bad:
const foo = function (){
};

good:
function foo() {
}
```

2. 用括号包裹立即执行函数

```
(function () {
  console.log('IIFE has been executed successfully');
})();
```

```在可以使用module的地方，基本上可以不使用IIFE```

3. 不能在一个非函数块内声明函数(if, while等)，通过变量赋值来替代

4. 不要使用'arguments'命名一个变量，它会覆盖每一个函数作用域内的arguments对象

```
bad:
function nope(name, options, arguments) {
}

good:
function yup(name, options, args) {
}
```

5. 不要使用arguments， 可以选择使用剩余参数语法...进行替代，好处是既可以明确指定你想获取的参数，还可以使用真实的数组

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

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

6. 使用函数参数默认值的语法替代改变函数参数

```
bad:
function handleClicks(opts) {
    opts = opts || {};
    // ... stuff ...
}

bad:
function handleClick(opts) {
    if (opts === void 0) {
        opts = {};
    }
    // ... stuff ...
```
```

```
}

good:
function handleClick(opts = {}) {
    // ... stuff ...
}
```

7. 总是把含默认值的参数放在最后

```
bad:
function handleClick(opts = {}, name) {
    // ... stuff ...
}

good:
function handleClick(name, opts = {}) {
    // ... stuff ...
}
```

8. 不要使用函数构造器创造新函数，因为这个方式会类似于用eval方法执行，导致漏洞的产生

```
bad:
var add = new Function('a', 'b', 'return a + b');

bad:
var subtract = Function('a', 'b', 'return a - b');
```

9. 合理地添加/使用空格

```
bad:
const f = function(){};
const g = function (){};
const h = function() {};

good:
const x = function () {};
const y = function a() {};
```

10. 不要改变入参， 这个可能会在原调用处导致副作用

```
bad:
function f(obj) {
    obj.key = 1;
}

good:
function f(obj) {
    const has = Object.prototype.hasOwn;
    const key = has.call(obj, 'key') ? obj.key : 1;
```

```
}
```

11. 不要重新对入参赋值

```
bad:
function f1(a) {
    a = 1;
}

bad:
function f2(a) {
    if (!a) { a = 1;}
}

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

good:
function f4(a = 1) {
}
```

Arrow Functions

1. 当一定要使用函数表达式时， 如传递一个匿名函数，尽量使用箭头函数

```
bad:
[1, 2, 3].map(function (x) {
    const y = x + 1;
    return x * y;
});

good:
[1, 2, 3].map((x) => {
    const y = x + 1;
    return x * y;
});
```

2. 如果只有一个函数，可以忽略括号，并使用隐式`return`，否则保留括号并使用`return`

```
bad:
[1, 2, 3].map(number => {
    const nextNumber = number + 1;
    `A string containing the ${nextNumber}.`;
});

good:
```

```
[1, 2, 3].map(number => {
    const nextNumber = number + 1;
    return `A string containing the ${nextNumber}`;
});
```

3. 针对箭头函数中只有一个参数的函数， 如果忽略了括号，那么返回的语句体也应该忽略掉

```
bad:
[1, 2, 3].map((x) => x * x);

good:
[1, 2, 3].map(x => x * x);
```

4. 在使用比较运算符(<=, >=)时应避免能带来疑惑的箭头函数语法

```
bad:
const itemHeight = item => item.height > 256 ? item.largeSize
: item.smallSize;

good:
const itemHeight = (item) => {return item.height > 256 ?
item.largeSize : item.smallSize;}
```

Classes & Constructors

1. 避免直接使用prototype，应使用class关键字

```
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;
    }
}
```

2. 使用extends关键字代替继承

```
bad:
const inherits = require('inherits');
function PeekableQueue (contents) {
    Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function () {
    return this._queue[0];
};

good:
class PeekableQueue extends Queue {
    peek () {
        return this._queue[0];
    }
}
```

3. 可以自己在类中重写toString方法，确保其正常工作且不会产生副作用

```
class Plane {
    constructor(options = {}) {
        this.name = options.name || 'no name';
    }

    getName() {
         return this.name;
    }

    toString() {
         return `${this.getName()}'s plane`;
    }
}
```

4. 使用 import / export来导入/导出模块

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

5. 不要把所有模块全部导入

```
bad:
import * as Yougoods from './Yougoods';

good:
import Yougoods from './Yougoods';
```

6. 在同一路径中导入所有模块

```
bad:
import foo from 'foo';
import {mod1, mod2} from 'foo';

good:
import foo, {mod1, mod2} from 'foo';

good:
import foo, {
    named1,
    named2,
} from 'foo';
```

7. 只有一个模块在进行导出时， 加上default关键字

```
export default function foo() {};
```

Iterators & Generators

1. 不要使用迭代器， 推荐使用map, reduce来替代类似for-of的循环

```
const numbers = [1, 2, 3, 4, 5];

bad:
let sum = 0;
for (let sum of numbers) {
    sum += num;
}

good:
let sum = 0;
numbers.forEach(num => sum += num);

best:
const sum = numbers.reduce((total, number) => total + num, 0);
```

2. 建议暂时不要使用generator, 因为它们暂时不能很好地被ES5支持

3. 如果一定要使用generator， 注意函数签名被合适地分隔开

```
bad:
function * foo() {
}
const bar = function * () {
};
const baz = function *() {
};
const quux = function*() {
```

```
};
function*foo() {
};

good:
function* foo() {
}
const foo = function* () {
};
```

Properties

1. 使用"."运算符来获取属性

```
const person = {
    name: 'John',
    age: 28,
};

bad:
const personName = person['name'];

good:
const personName = person.name;
```

2. 当属性名是变量时使用"[]"运算符来获取属性

```
const person = {
    name: 'John',
    age: 28,
};

function getProp(prop) {
    return person[prop];
}

const personName = getProp('name');
```

Variables

1. 使用const声明变量， 不要进行污染全局变量的赋值

```
bad:
superMan = new SuperMan();

good:
const superMan = new SuperMan();
```

2. 使用const来对每一个变量进行声明
```不要用逗号运算符进行一次性地声明，容易出错```

```
bad:
const items = getItems(),
      goShopping = true;

good:
const items = getItems();
const goShopping = true;
```

3. 为所有用const和let进行声明的变量进行良好地分组

```
bad:
let i, len, dragonball,
    items = getItems(),
    goShopping = true;

bad:
let i;
const items = getItems();
let dragonball;
const goShopping = true;
let len;

good:
const goShopping = true;
const items = getItems();
let dragonball;
let i;
let length;
```

4. 在需要的时候再分配变量，但是要确保在合理的位置进行分配

```
bad:
function checkName(hasName) {
    const name = getName(); // 这里没必要先调用

    if (hasName === 'test') return false; // 只有一句的可以不用括
号

    if (name === 'test') {
        this.setName('');
        return false;
    }

    return name;
}
```
```

```
good:
function checkName(hasName) {
    if (hasName === 'test') return false;

    const name = getName();

    if (name === 'test') {
        this.setName('');
        return false;
    }

    return name;
}
```

Hoisting（变量提升）

1. var 变量会提升到其作用域的顶部，但是赋值部分却不会
   ```const 和 let在使用上有可有会造成TDZ(Temporal Dead Zones), typeof运算符不再安全（要着重注意）```
   下面是几个例子：

```
function example() {
    console.log(varThatNotDefined); // throws a Reference
Error
}

function example() {
    console.log(varThatNotAssigned);  // undefined
    var varThatNotAssigned = true;
}

function example() {
    let varThatNotAssigned;
    console.log(varThatNotAssigned); // undefined
    varThatNotAssigned = true;
}

function example() {
    console.log(varThatNotAssigned); // throws a Reference
Error
    console.log(typeof varThatNotAssigned); // throws a
Reference Error
    const varThatNotAssigned = true;
}
```

2. 匿名函数表达式会提升变量的声明，但函数的分配不会

```
function example() {
    console.log(anonymous); // undefined
```
```

```
    anonymous(); // throws a TypeError

    var anonymous = function () {
         console.log('anonymous function expression');
    };
}
```

3. 命名的函数表达式提升变量的声明，但函数名和函数体都不会

```
function example() {
    console.log(named); // undefined

    named(); // => TypeError named is not a function

    superMan(); // ReferenceError superMan is not defined

    var named = function superMan() {
        console.log('I\'m a superman');
    };
}

// 即使是匿名函数的函数名相同，也是一样
function example() {
    console.log(named); // undefined

    named(); // TypeError named is not a function

    var named = function named() {
        console.log('named');
    };
}
```

4. 函数声明会提升函数名和函数体

```
function example() {
    superMan(); // flying

    function superMan() {
        console.log('flying');
    }
}
```

Comparison Operators && Equality

1. 总是使用 `===` 和 `!==` 来进行比较

2. 条件表达式，如`if`，进行比较时会强制使用`ToBoolean`这个抽象方法，并且会遵循以下一些规则

a. Object对象等价于true
b. undefined等价于false
c. null等价于false
d. Boolean对象等价于其对应的boolean值
e. Number对象中，除了+0，-0和NaN等价于false，其它都是true
f. String对象中如果是空串''等价于false，其它都是等价于true

```
if ([0] && []) {
    // true
    // 数组也是对象，即使是一个空数组，if判断时都会等价于true
}
```

3. 比较时使用简写形式

```
bad:
if (name !== '') {
    // do stuff
}

good:
if (name) {
    // do stuff
}

bad:
if (collection.length > 0) {
    // do stuff
}

good:
if (collection) {
    // do stuff
}
```

4. 在使用switch语句时，如果有使用声明声明的地方， 用'{}'进行包裹

```
bad:
switch (foo) {
    case 1:
        let x = 1;
        break;
    case 2:
        const y = 2;
        break;
    default:
        class C {}
}

good:
```

```
switch (foo) {
    case 1: {
        let x = 1;
        break;
    }
    case 2: {
        const y = 2;
        break;
    }
    default: {
        class C {}
    }
}
```

5. 三目运算符应该在一行内书写

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

good:
const maybeNull = value1 > value2 ? 'baz' : null;
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

6. 语法块

```
bad:
if (test)
    return false;

good:
if (test) return false;

good:
if (test) {
    return false;
}


bad:
if (test) {
    doThings();
    doOtherThings();
}
else {
    doAnotherThings();
}

good:
if (test) {
```

```
    doThings();
    doOtherThings();
} else {
    doAnotherThings();
}
```

Comments

1. 用/** ... */作多行注释，sublime插件推荐使用DocBlockr，注释要及时更新
2. 使用//作为单行注释，并新起一行放置在注释对象的顶部，注释前留一行
3. 在注释时加上一些前缀起到辅助的提示效果，如FIXME, TODO

```
class Compiler extends BaseCompiler {
    constructor () {
        super();

        // FIXME: shouldn't use a global var here
        lexer = new Lexer();

        // TODO: template should be loaded from a config file
        template = `It\'s a {{PageName}} page`;
    }
}
```

Spaces

1. 使用tab键，并设置为2个空格，缩进采用2个tab
2. 在前花括号前加上一个空格

```
bad:
function test(){
}

good:
function test() {
}

bad:
dog.set('attr',{
    age: '1 year',
    breed: 'Super dog',
});

good:
dog.set('attr', {
    age: '1 year',
```

```
    breed: 'Super dog',
});
```

3. 在括号前加入一个空格，但如果是参数列表前的括号，不需要加空格

```
bad:
function fight () {
}

if(isHero) {
    fight ();
}

good:
function fight() {
}

if(isHero) {
    fight();
}
```

4. 在运算符前后设置一个空格

```
bad:
const x=y+5;

good:
const x = y + 5;
```

5. 在使用jQuery或其它lib时， 如果有比较长的链式操作,应使用多行，并适当缩进

```
bad:
$('#selector').
    find('.active').
        highlight().
        end().
    find('.open').
        updateCount();


good:
$('#selector')
    .find('.active')
        .highlight()
        .end()
    .find('.open')
        .updateCount();
```

6. 在一个语法块后留一个空行

```
bad:
if (foo) {
    return bar;
}
return baz;

good:
if (foo) {
    return bar;
}

return baz;

bad:
const obj = {
    foo() {
    },
    bar() {
    },
};
return obj;


good:
const obj = {
    foo() {
    },

    bar() {
    },
};

return obj;
```

7. 不要用空行填充你的代码

```
bad:
if (baz) {

    console.log(qux);
} else {
    console.log(foo);

}

good:
if (baz) {
    console.log(qux);
} else {
    console.log(foo);
```

```
}
```

8. 不要在括号内加空格

```
bad:
function bar( foo ) {
    return foo;
}

good:
function bar(foo) {
    return foo;
}


bad:
if ( foo ) {
    console.log(foo);
}

good:
if (foo) {
    console.log(foo);
}
```

9. 同样地，不要在方括号内加空格

```
bad:
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);

good:
const foo = [1, 2, 3];
console.log(foo(0));
```

10. 花括号内的话，要适当添加空格

```
bad:
const foo = {clark: 'kent'};


good:
const foo = { clark: 'kent' };
```

Commas

1. 不要有前导逗号

```
bad:
const userStory = [
```

```
    once
  , upon
  , aTime
];

good:
const userStory = [
    once,
    upon,
    aTime,
];

bad:
const hero = {
    firstName: 'John'
  , lastName: 'Doe'
  , birthYear: 1815
  , superPower: 'computers'
};

good:
const hero = {
    firstName: 'John',
    lastName: 'Doe',
    birthYear: 1815,
    superPower: 'computers',
};
```

2. 注意以上的后继逗号要保留，这个在做git diff的时候好处很多

Semicolons

1. 分号一定要保留

Type Casting & Coercion

1. String

```
// this.reviewScore = 9;
bad:
const totalScore = this.reviewScore + ''; // 会先执行
this.reviewScore.valueOf()

bad:
const totalScore = this.reviewScore.toString(); // 不保证一定返
回一个string
```

```
good:
const totalScore = String(this.reviewScore);
```

2. Numbers

```
const inputValue = '4';

bad:
const val = new Number(inputValue);

bad:
const val = +inputValue;

bad:
const val = inputValue >> 0;

bad:
const val = parseInt(inputValue);

good:
const val = Number(inputValue);

good:
const val = parseInt(inputValue, 10);
```

3. 如果在代码中因为想提升效率而使用 >> 来取代parseInt的话，请加上注释说明清楚

```
/**
* parseInt 太慢了，这里需要做一个大数据IP去重算法要用到位运算
* 位运算快得多了， 测试结果在这 http://blabla.com/#test_result.html
*/
const val = inputValue >> 0;
```

4. 使用位移运算的时候要小心，因为Numbers底层用的是64bits的实现，而位移运算总会返回一个 32-bit 的整数，最大的32位整数是 2,147,483,647

```
2147483647 >> 0 // => 2147483647
2147483648 >> 0 // => -2147483648
2147483649 >> 0 // => -2147483647
```

5. Boolean

```
const age = 0;

bad:
const hasAge = new Boolean(age);

good:
```

```
const hasAge = Boolean(age);

best:
const hasAge = !!age;
```

Conventions

1. 避免无意义的命名，如单个名字的函数命名

```
bad:
function q() {
    // ... do stuff ...
}

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

2. 使用驼峰式命名

```
bad:
const OBJEctssssss = {};
const this_is_my_object = {};
function c() {}

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

3. 类的定义，构造函数，单例对象，函数库的使用上，名字的首字母大写

4. 不要使用前导或后继的下划线

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

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

5. 尽量少地用箭头函数保存this引用，使用箭头函数进行取代

```
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);
    };
}
```

Accessors

1. getters/setters

```
bad:
class Dragon {
    get age() {
        // ...
    }

    set age() {
        // ...
    }
}

good:
class Dragon {
    getAge() {
        // ...
    }

    setAge() {
        // ...
    }
}
```

2. 如果一个属性或方法是boolean，用isVal()，hasVal()的形式

```
bad:
if (!dragon.age()) {
    return false;
```

```
}

good:
if (!dragon.hasAge()) {
    return false;
}
```

jQuery

1. 如果是jQuery对象， 加上$前缀

```
bad:

const sidebar = $('.sidebar');

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

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

2. 缓存jQuery的查找

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

    // ...stuff...

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

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

    // ...stuff...

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

3. 使用parent > child的形式去查找对象，可以提升效率(这块需要了解jQuery选择器的实现原理)