前言

1. 什么是编码规范?

编码规范就是指导如何编写和组织代码的一系列标准。通过阅读这些编码规范，你可以知道各个公司的前端开发人员是如何编写代码的。

2. 我们为什么需要编码规范?

一个主要原因是：每个人都以不同的方式编写代码。我可能喜欢以某种方式做某件事，而且你可能喜欢以不同的方式去做。如果我们每个人都只在我们自己的代码上工作，这样并没有什么问题。但是，如果你有一个 10个，100个甚至1000个开发人员的团队，都在同一个代码库上工作，会发生什么呢？事情变得非常糟糕。 编码规范可以使新开发人员快速掌握代码，然后编写出其他开发人员可以快速轻松理解的代码！

3. 为什么选择Airbnb编码规范

ESlint主要提供3种预安装包：**Google标准**、**Airbnb标准**和**Standard标准**。这3个标准里，**Google标准**就是**Google**公司的标准，**Airbnb标准**是著名的独角兽公司**Airbnb**的前端编码规范，该项目是**github**上很受欢迎的一个开源项目，目前获得了80000多个**star**。**Standard**就是一些前端工程师自定的标准。目前来看，公认的最好的标准是**Airbnb标准**（互联网发展日新月异，永远是年轻人颠覆老年人，连**Google**都老了）。

这里我们选用了**Airbnb标准**里面的**JavaScript规范**和**React规范**。在上面两个规范的基础上我们删除了部分规则，也修改了部分规则，生成了如下的规范。

Airbnb JavaScript 风格指南

一、强制性规范

1. 引用

1.1 所有的赋值都用`const`，避免使用`var`。

eslint: `prefer-const`, `no-const-assign`

> Why? 因为这个确保你不会改变你的初始值，重复引用会导致bug和代码难以理解

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

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

1.2 如果你一定要对参数重新赋值，那就用`let`，而不是`var`。

eslint: `no-var`

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

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

5. 字符串

5.3 永远不要在字符串上使用`eval()`，它会打开太多的漏洞。

eslint: `no-eval`

5.4 不要转义字符串中不必要转义的字符。

eslint: `no-useless-escape`

```
  // bad
  const foo = '\'this\' \i\s \"quoted\"';
  // good
  const foo = '\'this\' is "quoted"';
  const foo = `my name is '${name}'`;
```

6. 函数

6.2 用圆括号包裹立即调用函数表达式 (IIFE)。

eslint: `wrap-iife`

```
// 立即调用函数表达式 (IIFE)
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

6.6 隔开函数签名，括号两边用空格隔开。

eslint: `space-before-function-paren` `space-before-blocks`

> 为什么？这样做有益代码的一致性，添加或删除函数名时不需要添加或删除空格。

```
// bad
const f = function(){};
const g = function (){};
const h = function() {};
// good
const x = function () {};
const y = function a() {};
```

7. 箭头函数

7.4 避免使用比较运算符(`< =`, `>=`)时，混淆箭头函数语法(`=>`)。

eslint: `no-confusing-arrow`

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

7.5 在隐式return中强制约束函数体的位置， 就写在箭头后面。

eslint: `implicit-arrow-linebreak`

```
// bad
(foo) =>
  bar;

(foo) =>
  (bar);

// good
(foo) => bar;
(foo) => (bar);
(foo) => (
   bar
)
```

8. 类&构造函数

8.2 避免重复类成员。

eslint: `no-dupe-class-members`

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

9. 模块

9.1 一个地方只在一个路径中 import(导入) 。

eslint: `no-duplicate-imports`

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

9.2 不要 export(导出) 可变绑定。

eslint: `import/no-mutable-exports`

> 为什么？ 一般应该避免可变性，特别是在导出可变绑定时。虽然一些特殊情况下，可能需要这种技术，但是一般而言，只应该导出常量引用。

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

9.4 将所有 `import` 导入放在非导入语句的上面。

eslint: `import/first`

```
// bad
import foo from 'foo';
foo.init();
import bar from 'bar';
// good
import foo from 'foo';
import bar from 'bar';
foo.init();
```

10. 迭代器 和 生成器

10.2 如果您必须使用 generators (生成器)，请确保它们的函数签名恰当的间隔。

eslint: `generator-star-spacing`

> 为什么？ `function` 和 `*` 都是同一概念关键字的组成部分 - `*` 不是 `function` 的修饰符，`function*` 是一个独特的构造，与`function`不同。

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

```
// bad
function*foo() {
    // ...
}
// bad
function *foo() {
    // ...
}
// very bad
function*foo() {
    // ...
}
// very bad
const wat = function*() {
    // ...
};
// good
function* foo() {
    // ...
}
// good
const foo = function* () {
    // ...
};
```

12. 变量

12.1 总是使用 `const` 或 `let` 来声明变量。 不这样做会导致产生全局变量。 我们希望避免污染全局命名空间。

eslint: `no-undef` `prefer-const`

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

12.2 使用 `const` 或 `let` 声明每个变量。

eslint: `one-var`

> 为什么？ 以这种方式添加新的变量声明更容易，你永远不必担心是否需要将 `,` 换成 `;`，或引入标点符号差异。您也可以在调试器中遍历每个声明，而不是一次跳过所有的变量。

```
// bad
const items = getItems(),
      goSportsTeam = true,
      dragonball = 'z';
// bad
// (与上面的比较，并尝试找出错误)
const items = getItems(),
      goSportsTeam = true;
dragonball = 'z';
// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

12.3 变量不要链式赋值。

eslint: `no-multi-assign`

> 为什么？ 链接变量赋值会创建隐式全局变量。

```
// bad
(function example() {
    // JavaScript 将其解析为
    // let a = ( b = ( c = 1 ) );
    // let关键字只适用于变量a;
    // 变量b和c变成了全局变量。
    let a = b = c = 1;
}());
console.log(a); // 抛出 ReferenceError (引用错误)
console.log(b); // 1
console.log(c); // 1

// good
(function example() {
    let a = 1;
    let b = a;
    let c = a;
}());
console.log(a); // 抛出 ReferenceError (引用错误)
console.log(b); // 抛出 ReferenceError (引用错误)
console.log(c); // 抛出 ReferenceError (引用错误)
// 同样适用于 `const`
```

13. 比较运算符 & 等号

13.3 三元表达式不应该嵌套，通常写成单行表达式。

eslint: `no-nested-ternary`

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

// 拆分成2个分离的三元表达式
const maybeNull = value1 > value2 ? 'baz' : null;
// better
const foo = maybe1 > maybe2
    ? 'bar'
    : maybeNull;
// best
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

13.4 避免不必要的三元表达式语句。

eslint: `no-unneeded-ternary`

```
// 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;
```

13.5 当运算符混合在一个语句中时，请将其放在括号内。混合算术运算符时，不要将 `**` 和 `%` 与 `+`，`-`，`*`，`/` 混合在一起。

eslint: `no-mixed-operators`

> 为什么？ 这可以提高可读性，并清晰展现开发者的意图。

```
// bad
const foo = a && b < 0 || c > 0 || d + 1 === 0;
// bad
const bar = a ** b - 5 % d;
// bad
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;
```

14. 代码块

14.1 使用大括号包裹所有的多行代码块。

eslint: `nonblock-statement-body-position`

```
// bad
if (test)
    return false;
// good
if (test) return false;
// good
if (test) {
    return false;
}
// bad
function foo() { return false; }
// good
function bar() {
    return false;
}
```

14.2 如果通过 `if` 和 `else` 使用多行代码块，把 `else` 放在 `if` 代码块闭合括号的同一行。

eslint: `brace-style`

```
// bad
if (test) {
    thing1();
    thing2();}
else {
    thing3();
}
// good
if (test) {
    thing1();
    thing2();
} else {
    thing3();
}
```

15. 注释

15.1 所有注释符和注释内容用一个空格隔开，让它更容易阅读。

eslint: `spaced-comment`

```
// bad

//is current tabconst active = true;

// good

// is current tabconst active = true;

// bad

/**
    *make() returns a new element
    *based on the passed-in tag name
    */
function make(tag) {
    // ...
    return element;
}

// good

/**
    * make() returns a new element
    * based on the passed-in tag name
    */
function make(tag) {
    // ...
    return element;
}
```

16. 空白

16.1 使用 4 个空格作为缩进。

eslint: `indent`

```
// good
function foo() {
????let name;
}
// bad
function bar() {
?let name;
}
// bad
function baz() {
??let name;
}
```

16.2 在大括号前放置 1 个空格。

eslint: `space-before-blocks`

```
// bad
function test(){
    console.log('test');
}
// good
function test() {
    console.log('test');
}
// bad
dog.set('attr',{
    age: '1 year',
    breed: 'Bernese Mountain Dog'
});
// good
dog.set('attr', {
    age: '1 year',
    breed: 'Bernese Mountain Dog'
});
```

16.3 在控制语句（`if`、`while` 等）的小括号前放一个空格。在函数调用及声明中，不在函数的参数列表前加空格。

eslint: `keyword-spacing`

```
// bad
if(isJedi) {
    fight ();
}
// good
if (isJedi) {
    fight();
```

```
}
// bad
function fight () {
    console.log ('Swooosh!');
}
// good
function fight() {
    console.log('Swooosh!');
}
```

16.4 使用空格把运算符隔开。

eslint: `space-infix-ops`

```
// bad
const x=y+5;
// good
const x = y + 5;
```

16.9 不要在圆括号内加空格。

eslint: `space-in-parens`

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

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

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

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

16.10 不要在中括号内添加空格。

eslint: `array-bracket-spacing`

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

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

16.11 在大括号内添加空格。

eslint: `object-curly-spacing`

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

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

16.12 避免有超过100个字符（包括空格）的代码行。

eslint: `max-len`

```
// bad
const foo = jsonData && jsonData.foo && jsonData.foo.bar &&
jsonData.foo.bar.baz && jsonData.foo.bar.baz.quux &&
jsonData.foo.bar.baz.quux.xyzzy;

// bad
$.ajax({ method: 'POST', url: 'https://airbnb.com/', data: { name: 'John' }
}).done(() => console.log('Congratulations!')).fail(() => console.log('You
have failed this city.'));

// good
const foo = jsonData
  && jsonData.foo
  && jsonData.foo.bar
  && jsonData.foo.bar.baz
  && jsonData.foo.bar.baz.quux
  && jsonData.foo.bar.baz.quux.xyzzy;

// good
$.ajax({
  method: 'POST',
  url: 'https://airbnb.com/',
  data: { name: 'John' },
})
  .done(() => console.log('Congratulations!'))
  .fail(() => console.log('You have failed this city.'));
```

16.13 作为语句的花括号内也要加空格 —— `{` 后和 `}` 前都需要空格。

eslint: `block-spacing`

```
// bad
function foo() {return true;}
if (foo) { bar = 0;}

// good
function foo() { return true; }
if (foo) { bar = 0; }
```

16.14 `,` 前不要空格，`,` 后需要空格。

eslint: `comma-spacing`

```
// bad
var foo = 1,bar = 2;
var arr = [1 , 2];

// good
var foo = 1, bar = 2;
var arr = [1, 2];
```

16.15 计算属性内要空格。参考上述花括号和中括号的规则。

eslint: `computed-property-spacing`

```
// bad
obj[foo ]
obj[ 'foo']
var x = {[ b ]: a}
obj[foo[ bar ]]

// good
obj[foo]
obj['foo']
var x = { [b]: a }
obj[foo[bar]]
```

16.16 调用函数时，函数名和小括号之间不要空格。

eslint: `func-call-spacing`

```
// bad
func ();

func
();

// good
func();
```

16.17 在对象的字面量属性中， `key` `value` 之间要有空格。

eslint: key-spacing

```
// bad
var obj = { "foo" : 42 };
var obj2 = { "foo":42 };

// good
var obj = { "foo": 42 };
```

17. 逗号

17.1 行开头处不要实用使用逗号。

eslint: comma-style

```
// bad
const story = [
    once
  , upon
  , aTime
];

// good
const story = [
  once,
  upon,
  aTime,
];

// bad
const hero = {
    firstName: 'Ada'
  , lastName: 'Lovelace'
  , birthYear: 1815
  , superPower: 'computers'
};
```

```
// good
const hero = {
  firstName: 'Ada',
  lastName: 'Lovelace',
  birthYear: 1815,
  superPower: 'computers',
};
```

18. 分号

18.1 当然要使用封号

eslint: `semi`

> 为什么？ 当 JavaScript 遇到没有分号的换行符时，它使用一组称为自动分号插入的规则来确定是否应该将换行符视为语句的结尾，并且（顾名思义）如果被这样认为的话，在换行符前面自动插入一个分号。ASI（自动分号插入）包含了一些稀奇古怪的的行为，不过，如果 JavaScript 错误地解释了你的换行符，你的代码将会被中断执行。随着新功能成为 JavaScript 的一部分，这些规则将变得更加复杂。明确地结束你的语句并配置你的 linter 来捕获缺少的分号，将有助于防止遇到问题。

```
// bad - raises exception
const luke = {}
const leia = {}
[luke, leia].forEach((jedi) => jedi.father = 'vader')

// bad - raises exception
const reaction = "No! That’s impossible!"
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}())

// bad - returns `undefined` instead of the value on the next line - always
happens when `return` is on a line by itself because of ASI!
function foo() {
  return
    'search your feelings, you know it to be foo'
}

// good
const luke = {};
const leia = {};
[luke, leia].forEach((jedi) => {
  jedi.father = 'vader';
});

// good
const reaction = "No! That’s impossible!";
```

```
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}());

// good
function foo() {
  return 'search your feelings, you know it to be foo';
}
```

20. 命名规则

20.1 避免使用单字母名称。使你的命名具有描述性。

eslint: `id-length`

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

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

20.2 当命名对象，函数和实例时使用驼峰式命名。

eslint: `camelcase`

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

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

20.3 当命名构造函数或类的时候使用 PascalCase 式命名，（注：即单词首字母大写）。

eslint: `new-cap`

```
// 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',
});
```

Airbnb React/JSX 编码规范

一、强行性规范

1. Class vs `React.createClass` vs stateless

1.1 如果你要用 state refs， 最好用 `class extends React.Component` 而不是 `React.createClass`

eslint: `react/prefer-es6-class` `react/prefer-stateless-function`

```
// bad
const Listing = React.createClass({
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
});

// good
class Listing extends React.Component {
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
}
```

如果你没有使用 state、 refs ， 最好用正常函数(不是箭头函数)而不是 class：

```
// bad
```

```
class Listing extends React.Component {
  render() {
    return <div>{this.props.hello}</div>;
  }
}

// bad (不鼓励依赖函数名推断----relying on function name inference is
discouraged)
const Listing = ({ hello }) => (
  <div>{hello}</div>
);

// good
function Listing({ hello }) {
  return <div>{hello}</div>;
}
```

2. 命名(Naming)

2.1 文件名: 用大驼峰作为文件名。参数命名: React 组件用大驼峰，组件的实例用小驼峰。

eslint: `react/jsx-pascal-case`

3. 对齐(Alignment)

3.1 对 JSX 语法使用这些对齐风格。

eslint: `react/jsx-closing-bracket-location` `react/jsx-closing-tag-location`

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// 如果能放在一行，也可以用单行表示
<Foo bar="bar" />

// Foo 里面的标签正常缩进
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
```

```
</Foo>

// bad
{showButton &&
  <Button />
}

// bad
{
  showButton &&
    <Button />
}

// good
{showButton && (
  <Button />
)}

// good
{showButton && <Button />}
```

4. 空格

4.1 在自闭和标签内空一格。

eslint: `no-multi-spaces`, `react/jsx-tag-spacing`

```
// bad
<Foo/>

// very bad
<Foo                  />

// bad
<Foo
 />

// good
<Foo />
```

4.2 JSX 里的大括号不要空格。

eslint: `react/jsx-curly-spacing`

```
// bad
<Foo bar={ baz } />

// good
<Foo bar={baz} />
```

5 属性

5.1 如果 prop 的值是 true 可以忽略这个值，直接写 prop 名就可以。

eslint: `react/jsx-boolean-value`

```
// bad
<Foo
  hidden={true}
/>

// good
<Foo
  hidden
/>

// good
<Foo hidden />
```

6 引用

6.1推荐用 ref callback 函数。

eslint: `react/no-string-refs`

```
// bad
<Foo
  ref="myRef"
/>

// good
<Foo
  ref={(ref) => { this.myRef = ref; }}
/>
```

7 括号

7.1 当 JSX 标签有多行时，用圆括号包起来。

eslint: `react/jsx-wrap-multilines`

8 标签

8.1 当没有子元素时，最好用自闭合标签。

eslint: `react/self-closing-comp`

```
// bad
<Foo variant="stuff"></Foo>

// good
<Foo variant="stuff" />
```

8.2 如果你的组件有多行属性，用他的闭合标签单独作为结束行。

eslint: `react/jsx-closing-bracket-location`

```
// bad
<Foo
  bar="bar"
  baz="baz" />

// good
<Foo
  bar="bar"
  baz="baz"
/>
```

9 方法

9.1 在构造函数里绑定事件处理函数。

eslint: `react/jsx-no-bind`

> Why? render 函数中的绑定调用在每次 render 的时候都会创建一个新的函数。

```
// bad
class extends React.Component {
  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv.bind(this)} />;
  }
}

// good
class extends React.Component {
```

```
  constructor(props) {
    super(props);

    this.onClickDiv = this.onClickDiv.bind(this);
  }

  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />;
  }
}
```

9.2 确保你的 `render` 函数有返回值。

eslint: `react/require-render-return`

```
// bad
render() {
  (<div />);
}

// good
render() {
  return (<div />);
}
```

参考文献

airbnb官方的JavaScript规范

备注

文档中如果有读不懂的地方，建议去读airbnb官方的JavaScript规范的英文原版。