

百度前端代码规范

书栈(BookStack.CN)

目 录

致谢

README

Javascript编码规范

Javascript编码规范 - ESNext补充篇

HTML编码规范

CSS编码规范

Less编码规范

E-JSON数据传输标准

模块和加载器规范

包结构规范

项目目录结构规范

图表库标准

react编码规范

致谢

当前文档《百度前端代码规范》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建, 生成于 2018-07-04。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能, 以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理, 书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候, 发现文档内容有不恰当的地方, 请向我们反馈, 让我们共同携手, 将知识准确、高效且有效地传递给每一个人。

同时, 如果您在日常工作、生活和学习中遇到有价值有营养的知识文档, 欢迎分享到 书栈(BookStack.CN), 为知识的传承献上您的一份力量!

如果当前文档生成时间太久, 请到 书栈(BookStack.CN) 获取最新的文档, 以跟上知识更新换代的步伐。

文档地址: <http://www.bookstack.cn/books/ecomfe-spec>

书栈官网: <http://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享, 让知识传承更久远! 感谢知识的创造者, 感谢知识的分享者, 也感谢每一位阅读到此处的读者, 因为我们都将成为知识的传承者。

README

This repository contains the specifications.

- [Javascript编码规范](#) [1.3]
- [Javascript编码规范 - ESNext补充篇](#) [draft]
- [HTML编码规范](#) [1.2]
- [CSS编码规范](#) [1.2]
- [Less编码规范](#) [1.1]
- [E-JSON数据传输标准](#) [1.0]
- [模块和加载器规范](#) [1.1]
- [包结构规范](#) [1.1]
- [项目目录结构规范](#) [1.1]
- [图表库标准](#) [1.0]
- [react编码规范](#) [draft]

Lint and fix tool: [FECS](#)

来源(书栈小编注)

<https://github.com/ecomfe/spec>

Javascript编码规范

JavaScript编码规范

1 前言

JavaScript 在百度一直有着广泛的应用，特别是在浏览器端的行为管理。本文档的目标是使 JavaScript 代码风格保持一致，容易被理解和被维护。

虽然本文档是针对 JavaScript 设计的，但是在使用各种 JavaScript 的预编译语言时(如 TypeScript 等)时，适用的部分也应尽量遵循本文档的约定。

2 代码风格

2.1 文件

[建议] JavaScript 文件使用无 `BOM` 的 `UTF-8` 编码。

解释：

UTF-8 编码具有更广泛的适应性。BOM 在使用程序或工具处理文件时可能造成不必要的干扰。

[建议] 在文件结尾处，保留一个空行。

2.2 结构

2.2.1 缩进

[强制] 使用 `4` 个空格做为一个缩进层级，不允许使用 `2` 个空格 或 `tab` 字符。

[强制] `switch` 下的 `case` 和 `default` 必须增加一个缩进层级。

示例：

```
1. // good
2. switch (variable) {
3.
4.     case '1':
5.         // do...
6.         break;
7.
```

```
8.     case '2':
9.         // do...
10.        break;
11.
12.    default:
13.        // do...
14.
15. }
16.
17. // bad
18. switch (variable) {
19.
20. case '1':
21.     // do...
22.     break;
23.
24. case '2':
25.     // do...
26.     break;
27.
28. default:
29.     // do...
30.
31. }
```

2.2.2 空格

[强制] 二元运算符两侧必须有一个空格，一元运算符与操作对象之间不允许有空格。

示例：

```
1. var a = !arr.length;
2. a++;
3. a = b + c;
```

[强制] 用作代码块起始的左花括号 `{` 前必须有一个空格。

示例：

```
1. // good
2. if (condition) {
3. }
4.
```

```
5. while (condition) {
6. }
7.
8. function funcName() {
9. }
10.
11. // bad
12. if (condition){
13. }
14.
15. while (condition){
16. }
17.
18. function funcName(){
19. }
```

[强制] `if / else / for / while / function / switch / do / try / catch / finally` 关键字后，必须有一个空格。

示例：

```
1. // good
2. if (condition) {
3. }
4.
5. while (condition) {
6. }
7.
8. (function () {
9. })();
10.
11. // bad
12. if(condition) {
13. }
14.
15. while(condition) {
16. }
17.
18. (function() {
19. })();
```

[强制] 在对象创建时，属性中的 `:` 之后必须有空格，`:` 之前不允许有空格。

示例：

```
1. // good
2. var obj = {
3.     a: 1,
4.     b: 2,
5.     c: 3
6. };
7.
8. // bad
9. var obj = {
10.     a : 1,
11.     b:2,
12.     c :3
13. };
```

[强制] 函数声明、具名函数表达式、函数调用中，函数名和 `(` 之间不允许有空格。

示例：

```
1. // good
2. function funcName() {
3. }
4.
5. var funcName = function funcName() {
6. };
7.
8. funcName();
9.
10. // bad
11. function funcName () {
12. }
13.
14. var funcName = function funcName () {
15. };
16.
17. funcName ();
```

[强制] `,` 和 `;` 前不允许有空格。如果不位于行尾，`,` 和 `;` 后必须跟一个空格。

示例：

```
1. // good
```



```

2.  callFunc(a, b);
3.
4.  // bad
5.  callFunc(a , b) ;

```

[强制] 在函数调用、函数声明、括号表达式、属性访问、等语句中，`()` 和 `[]` 内紧贴括号部分不允许有空格。

`if / for / while / switch / catch`

示例：

```

1.  // good
2.
3.  callFunc(param1, param2, param3);
4.
5.  save(this.list[this.indexes[i]]);
6.
7.  needIncream && (variable += increment);
8.
9.  if (num > list.length) {
10. }
11.
12. while (len--) {
13. }
14.
15.
16. // bad
17.
18. callFunc( param1, param2, param3 );
19.
20. save( this.list[ this.indexes[ i ] ] );
21.
22. needIncrement && ( variable += increment );
23.
24. if ( num > list.length ) {
25. }
26.
27. while ( len-- ) {
28. }

```

[强制] 单行声明的数组与对象，如果包含元素，`{}` 和 `[]` 内紧贴括号部分不允许包含空格。

解释：

声明包含元素的数组与对象，只有当内部元素的形式较为简单时，才允许写在一行。元素复杂的情况，还是应该换行书写。

示例：

```
1. // good
2. var arr1 = [];
3. var arr2 = [1, 2, 3];
4. var obj1 = {};
5. var obj2 = {name: 'obj'};
6. var obj3 = {
7.     name: 'obj',
8.     age: 20,
9.     sex: 1
10. };
11.
12. // bad
13. var arr1 = [ ];
14. var arr2 = [ 1, 2, 3 ];
15. var obj1 = { };
16. var obj2 = { name: 'obj' };
17. var obj3 = {name: 'obj', age: 20, sex: 1};
```

[强制] 行尾不得有多余的空格。

2.2.3 换行

[强制] 每个独立语句结束后必须换行。

[强制] 每行不得超过 120 个字符。

解释：

超长的不可分割的代码允许例外，比如复杂的正则表达式。长字符串不在例外之列。

[强制] 运算符处换行时，运算符必须在新行的行首。

示例：

```
1. // good
2. if (user.isAuthenticated()
3.     && user.isInRole('admin')
4.     && user.hasAuthority('add-admin')
5.     || user.hasAuthority('delete-admin')
6. ) {
```

```
7.      // Code
8.  }
9.
10. var result = number1 + number2 + number3
11.     + number4 + number5;
12.
13.
14. // bad
15. if (user.isAuthenticated() &&
16.     user.isInRole('admin') &&
17.     user.hasAuthority('add-admin') ||
18.     user.hasAuthority('delete-admin')) {
19.     // Code
20. }
21.
22. var result = number1 + number2 + number3 +
23.     number4 + number5;
```

[强制] 在函数声明、函数表达式、函数调用、对象创建、数组创建、`for` 语句等场景中，不允许在
`,` 或 `;` 前换行。

示例：

```
1.  // good
2.  var obj = {
3.      a: 1,
4.      b: 2,
5.      c: 3
6.  };
7.
8.  foo(
9.      aVeryVeryLongArgument,
10.     anotherVeryLongArgument,
11.     callback
12. );
13.
14.
15. // bad
16. var obj = {
17.     a: 1
18.     , b: 2
19.     , c: 3
20. };
```

```
21.  
22.  foo(  
23.      aVeryVeryLongArgument  
24.      , anotherVeryLongArgument  
25.      , callback  
26.  );
```

[建议] 不同行为或逻辑的语句集，使用空行隔开，更易阅读。

示例：

```
1.  // 仅为按逻辑换行的示例，不代表setStyle的最优实现  
2.  function setStyle(element, property, value) {  
3.      if (element == null) {  
4.          return;  
5.      }  
6.  
7.      element.style[property] = value;  
8.  }
```

[建议] 在语句的行长度超过 120 时，根据逻辑条件合理缩进。

示例：

```
1.  // 较复杂的逻辑条件组合，将每个条件独立一行，逻辑运算符放置在行首进行分隔，或将部分逻辑按逻辑  
   // 组合进行分隔。  
2.  // 建议最终将右括号 ) 与左大括号 { 放在独立一行，保证与 `if` 内语句块能容易视觉辨识。  
3.  if (user.isAuthenticated()  
4.      && user.isInRole('admin')  
5.      && user.hasAuthority('add-admin')  
6.      || user.hasAuthority('delete-admin')  
7.  ) {  
8.      // Code  
9.  }  
10.  
11. // 按一定长度截断字符串，并使用 + 运算符进行连接。  
12. // 分隔字符串尽量按语义进行，如不要在一个完整的名词中间断开。  
13. // 特别的，对于 HTML 片段的拼接，通过缩进，保持和 HTML 相同的结构。  
14. var html = '' // 此处用一个空字符串，以便整个 HTML 片段都在新行严格对齐  
15.     + '<article>'  
16.     + '    <h1>Title here</h1>'  
17.     + '    <p>This is a paragraph</p>'  
18.     + '    <footer>Complete</footer>'
```

```
19.     + '</article>';
20.
21. // 也可使用数组来进行拼接, 相对 `` 更容易调整缩进。
22. var html = [
23.     '<article>',
24.     '<h1>Title here</h1>',
25.     '<p>This is a paragraph</p>',
26.     '<footer>Complete</footer>',
27.     '</article>'
28. ];
29. html = html.join('');
30.
31. // 当参数过多时, 将每个参数独立写在一行上, 并将结束的右括号 ) 独立一行。
32. // 所有参数必须增加一个缩进。
33. foo(
34.     aVeryVeryLongArgument,
35.     anotherVeryLongArgument,
36.     callback
37. );
38.
39. // 也可以按逻辑对参数进行组合。
40. // 最经典的是 baidu.format 函数, 调用时将参数分为“模板”和“数据”两块
41. baidu.format(
42.     dateFormatTemplate,
43.     year, month, date, hour, minute, second
44. );
45.
46. // 当函数调用时, 如果有一个或以上参数跨越多行, 应当每一个参数独立一行。
47. // 这通常出现在匿名函数或者对象初始化等作为参数时, 如 `setTimeout` 函数等。
48. setTimeout(
49.     function () {
50.         alert('hello');
51.     },
52.     200
53. );
54.
55. order.data.read(
56.     'id=' + me.model.id,
57.     function (data) {
58.         me.attchToModel(data.result);
59.         callback();
60.     },
```

```
61.     300
62. );
63.
64. // 链式调用较长时采用缩进进行调整。
65. $('#items')
66.     .find('.selected')
67.     .highlight()
68.     .end();
69.
70. // 三元运算符由3部分组成，因此其换行应当根据每个部分的长度不同，形成不同的情况。
71. var result = thisIsAVeryVeryLongCondition
72.     ? resultA : resultB;
73.
74. var result = condition
75.     ? thisIsAVeryVeryLongResult
76.     : resultB;
77.
78. // 数组和对象初始化的混用，严格按照每个对象的 `{` 和结束 `}` 在独立一行的风格书写。
79. var array = [
80.     {
81.         // ...
82.     },
83.     {
84.         // ...
85.     }
86. ];
```

[建议] 对于 `if...else...`、`try...catch...finally` 等语句，推荐使用在 `}` 号后添加一个换行的风格，使代码层次结构更清晰，阅读性更好。

示例：

```
1. if (condition) {
2.     // some statements;
3. }
4. else {
5.     // some statements;
6. }
7.
8. try {
9.     // some statements;
10. }
11. catch (ex) {
```

```
12.      // some statements;
13.  }
```

2.2.4 语句

[强制] 不得省略语句结束的分号。

[强制] 在 `if / else / for / do / while` 语句中，即使只有一行，也不得省略块 `{...}`。

示例：

```
1.  // good
2.  if (condition) {
3.      callFunc();
4.  }
5.
6.  // bad
7.  if (condition) callFunc();
8.  if (condition)
9.      callFunc();
```

[强制] 函数定义结束不允许添加分号。

示例：

```
1.  // good
2.  function funcName() {
3.  }
4.
5.  // bad
6.  function funcName() {
7.  };
8.
9.  // 如果是函数表达式，分号是不允许省略的。
10. var funcName = function () {
11. };
```

[强制] `IIFE` 必须在函数表达式外添加 `(`，非 `IIFE` 不得在函数表达式外添加 `(`。

解释：

IIFE = Immediately-Invoked Function Expression.

额外的 `(` 能够让代码在阅读的一开始就能判断函数是否立即被调用，进而明白接下来代码的用途。而

不是一直拖到底部才恍然大悟。

示例：

```
1.  // good
2.  var task = (function () {
3.      // Code
4.      return result;
5.  })();
6.
7.  var func = function () {
8.  };
9.
10.
11. // bad
12. var task = function () {
13.     // Code
14.     return result;
15. }();
16.
17. var func = (function () {
18. });
```

2.3 命名

[强制] **变量** 使用 **Camel命名法**。

示例：

```
1. var loadingModules = {};
```

[强制] **常量** 使用 **全部字母大写，单词间下划线分隔** 的命名方式。

示例：

```
1. var HTML_ENTITY = {};
```

[强制] **函数** 使用 **Camel命名法**。

示例：

```
1. function stringFormat(source) {
2. }
```


[强制] 函数的 **参数** 使用 **Camel命名法**。

示例：

```
1. function hear(theBells) {  
2. }
```

[强制] **类** 使用 **Pascal命名法**。

示例：

```
1. function TextNode(options) {  
2. }
```

[强制] 类的 **方法** / **属性** 使用 **Camel命名法**。

示例：

```
1. function TextNode(value, engine) {  
2.     this.value = value;  
3.     this.engine = engine;  
4. }  
5.  
6. TextNode.prototype.clone = function () {  
7.     return this;  
8. };
```

[强制] **枚举变量** 使用 **Pascal命名法**，**枚举的属性** 使用 **全部字母大写，单词间下划线分隔** 的命名方式。

示例：

```
1. var TargetState = {  
2.     READING: 1,  
3.     READED: 2,  
4.     APPLIED: 3,  
5.     READY: 4  
6. };
```

[强制] **命名空间** 使用 **Camel命名法**。

示例：

```
1. equipments.heavyWeapons = {};
```

[强制] 由多个单词组成的缩写词，在命名中，根据当前命名法和出现的位置，所有字母的大小写与首字母的大小写保持一致。

示例：

```
1. function XMLParser() {  
2. }  
3.  
4. function insertHTML(element, html) {  
5. }  
6.  
7. var httpRequest = new HTTPRequest();
```

[强制] **类名** 使用 **名词**。

示例：

```
1. function Engine(options) {  
2. }
```

[建议] **函数名** 使用 **动宾短语**。

示例：

```
1. function getStyle(element) {  
2. }
```

[建议] **boolean** 类型的变量使用 **is** 或 **has** 开头。

示例：

```
1. var isReady = false;  
2. var hasMoreCommands = false;
```

[建议] **Promise对象** 用 **动宾短语的进行时** 表达。

示例：

```
1. var loadingData = ajax.get('url');  
2. loadingData.then(callback);
```

2.4 注释

2.4.1 单行注释

[强制] 必须独占一行。 `//` 后跟一个空格，缩进与下一行被注释说明的代码一致。

2.4.2 多行注释

[建议] 避免使用 `/*...*/` 这样的多行注释。有多行注释内容时，使用多个单行注释。

2.4.3 文档化注释

[强制] 为了便于代码阅读和自文档化，以下内容必须包含以 `/**...*/` 形式的块注释中。

解释：

1. 文件
2. namespace
3. 类
4. 函数或方法
5. 类属性
6. 事件
7. 全局变量
8. 常量
9. AMD 模块

[强制] 文档注释前必须空一行。

[建议] 自文档化的文档说明 what，而不是 how。

2.4.4 类型定义

[强制] 类型定义都是以 `{` 开始，以 `}` 结束。

解释：

常用类型如：`{string}`，`{number}`，`{boolean}`，`{Object}`，`{Function}`，`{RegExp}`，`{Array}`，`{Date}`。

类型不仅局限于内置的类型，也可以是自定义的类型。比如定义了一个类 `Developer`，就可以使用它来定义一个参数和返回值的类型。

[强制] 对于基本类型 `{string}`，`{number}`，`{boolean}`，首字母必须小写。

类型定义	语法示例	解释
String	<code>{string}</code>	—
Number	<code>{number}</code>	—
Boolean	<code>{boolean}</code>	—

Object	{Object}	—
Function	{Function}	—
RegExp	{RegExp}	—
Array	{Array}	—
Date	{Date}	—
单一类型集合	{Array.<string>}	string 类型的数组
多类型	{(number boolean)}	可能是 number 类型，也可能是 boolean 类型
允许为null	{?number}	可能是 number，也可能是 null
不允许为null	{!Object}	Object 类型，但不是 null
Function类型	{function(number, boolean)}	函数，形参类型
Function带返回值	{function(number, boolean):string}	函数，形参，返回值类型
Promise	Promise.<resolveType, rejectType>	Promise，成功返回的数据类型，失败返回的错误类型
参数可选	@param {string=} name	可选参数，=为类型后缀
可变参数	@param {...number} args	变长参数，...为类型前缀
任意类型	{*}	任意类型
可选任意类型	@param {*=} name	可选参数，类型不限
可变任意类型	@param {...*} args	变长参数，类型不限

2.4.5 文件注释

[强制] 文件顶部必须包含文件注释，用 `@file` 标识文件说明。

示例：

```

1.  /**
2.   * @file Describe the file
3.   */

```

[建议] 文件注释中可以用 `@author` 标识开发者信息。

解释：

开发者信息能够体现开发人员对文件的贡献，并且能够让遇到问题或希望了解相关信息的人找到维护人。通常情况文件在被创建时标识的是创建者。随着项目的进展，越来越多的人加入，参与这个文件的开发，新的作者应该被加入 `@author` 标识。

`@author` 标识具有多人时，原则是按照 `责任` 进行排序。通常的说就是如果有问题，就是找第一个人应该比找第二个人有效。比如文件的创建者由于各种原因，模块移交给了其他人或其他团队，后来因为新增需求，其他人在新增代码时，添加 `@author` 标识应该把自己的名字添加在创建人的前

面。

`@author` 中的名字不允许被删除。任何劳动成果都应该被尊重。

业务项目中，一个文件可能被多人频繁修改，并且每个人的维护时间都可能不会很长，不建议为文件增加 `@author` 标识。通过版本控制系统追踪变更，按业务逻辑单元确定模块的维护责任人，通过文档与wiki跟踪和查询，是更好的责任管理方式。

对于业务逻辑无关的技术型基础项目，特别是开源的公共项目，应使用 `@author` 标识。

示例：

```
1. /**
2.  * @file Describe the file
3.  * @author author-name(mail-name@domain.com)
4.  *      author-name2(mail-name2@domain.com)
5.  */
```

2.4.6 命名空间注释

[建议] 命名空间使用 `@namespace` 标识。

示例：

```
1. /**
2.  * @namespace
3.  */
4. var util = {};
```

2.4.7 类注释

[建议] 使用 `@class` 标记类或构造函数。

解释：

对于使用对象 `constructor` 属性来定义的构造函数，可以使用 `@constructor` 来标记。

示例：

```
1. /**
2.  * 描述
3.  *
4.  * @class
5.  */
```

```

6.  function Developer() {
7.      // constructor body
8.  }

```

[建议] 使用 `@extends` 标记类的继承信息。

示例：

```

1.  /**
2.   * 描述
3.   *
4.   * @class
5.   * @extends Developer
6.   */
7.  function Fronteer() {
8.      Developer.call(this);
9.      // constructor body
10. }
11. util.inherits(Fronteer, Developer);

```

[强制] 使用包装方式扩展类成员时， 必须通过 `@lends` 进行重新指向。

解释：

没有 `@lends` 标记将无法为该类生成包含扩展类成员的文档。

示例：

```

1.  /**
2.   * 类描述
3.   *
4.   * @class
5.   * @extends Developer
6.   */
7.  function Fronteer() {
8.      Developer.call(this);
9.      // constructor body
10. }
11.
12. util.extend(
13.     Fronteer.prototype,
14.     /** @lends Fronteer.prototype */{
15.         getLevel: function () {

```

```

16.          // TODO
17.      }
18.  }
19. );

```

[强制] 类的属性或方法等成员信息不是 `public` 的，应使用 `@protected` 或 `@private` 标识可访问性。

解释：

生成的文档中将有可访问性的标记，避免用户直接使用非 `public` 的属性或方法。

示例：

```

1.  /**
2.   * 类描述
3.   *
4.   * @class
5.   * @extends Developer
6.   */
7.  var Fronteer = function () {
8.      Developer.call(this);
9.
10.     /**
11.      * 属性描述
12.      *
13.      * @type {string}
14.      * @private
15.      */
16.     this.level = 'T12';
17.
18.     // constructor body
19. };
20. util.inherits(Fronteer, Developer);
21.
22. /**
23.  * 方法描述
24.  *
25.  * @private
26.  * @return {string} 返回值描述
27.  */
28. Fronteer.prototype.getLevel = function () {
29. };

```

2.4.8 函数/方法注释

[强制] 函数/方法注释必须包含函数说明，有参数和返回值时必须使用注释标识。

解释：

当 `return` 关键字仅作退出函数/方法使用时，无须对返回值作注释标识。

[强制] 参数和返回值注释必须包含类型信息，且不允许省略参数的说明。

[建议] 当函数是内部函数，外部不可访问时，可以使用 `@inner` 标识。

示例：

```

1.  /**
2.   * 函数描述
3.   *
4.   * @param {string} p1 参数1的说明
5.   * @param {string} p2 参数2的说明，比较长
6.   *      那就换行了.
7.   * @param {number=} p3 参数3的说明（可选）
8.   * @return {Object} 返回值描述
9.   */
10. function foo(p1, p2, p3) {
11.     var p3 = p3 || 10;
12.     return {
13.         p1: p1,
14.         p2: p2,
15.         p3: p3
16.     };
17. }
```

[强制] 对 Object 中各项的描述， 必须使用 `@param` 标识。

示例：

```

1.  /**
2.   * 函数描述
3.   *
4.   * @param {Object} option 参数描述
5.   * @param {string} option.url option项描述
6.   * @param {string=} option.method option项描述，可选参数
7.   */
8. function foo(option) {
9.     // TODO
```



```
10. }
```

[建议] 重写父类方法时，应当添加 `@override` 标识。如果重写的形参个数、类型、顺序和返回值类型均未发生变化，可省略 `@param`、`@return`，仅用 `@override` 标识，否则仍应作完整注释。

解释：

简而言之，当子类重写的方法能直接套用父类的方法注释时可省略对参数与返回值的注释。

2.4.9 事件注释

[强制] 必须使用 `@event` 标识事件，事件参数的标识与方法描述的参数标识相同。

示例：

```
1. /**
2.  * 值变更时触发
3.  *
4.  * @event Select#change
5.  * @param {Object} e e描述
6.  * @param {string} e.before before描述
7.  * @param {string} e.after after描述
8.  */
9. this.fire(
10.     'change',
11.     {
12.         before: 'foo',
13.         after: 'bar'
14.     }
15. );
```

[强制] 在会广播事件的函数前使用 `@fires` 标识广播的事件，在广播事件代码前使用 `@event` 标识事件。

[建议] 对于事件对象的注释，使用 `@param` 标识，生成文档时可读性更好。

示例：

```
1. /**
2.  * 点击处理
3.  *
4.  * @fires Select#change
5.  * @private
6.  */
```

```

7.  Select.prototype.clickHandler = function () {
8.
9.      /**
10.       * 值变更时触发
11.       *
12.       * @event Select#change
13.       * @param {Object} e e描述
14.       * @param {string} e.before before描述
15.       * @param {string} e.after after描述
16.       */
17.      this.fire(
18.          'change',
19.          {
20.              before: 'foo',
21.              after: 'bar'
22.          }
23.      );
24.  };

```

2.4.10 常量注释

[强制] 常量必须使用 `@const` 标记，并包含说明和类型信息。

示例：

```

1.  /**
2.   * 常量说明
3.   *
4.   * @const
5.   * @type {string}
6.   */
7.  var REQUEST_URL = 'myurl.do';

```

2.4.11 复杂类型注释

[建议] 对于类型未定义的复杂结构的注释，可以使用 `@typedef` 标识来定义。

示例：

```

1.  // `namespaceA~` 可以换成其它 namepaths 前缀，目的是为了生成文档中能显示 `@typedef` 定义的类型和链接。
2.  /**
3.   * 服务器

```

```

4.  *
5.  * @typedef {Object} namespaceA~Server
6.  * @property {string} host 主机
7.  * @property {number} port 端口
8.  */
9.
10. /**
11.  * 服务器列表
12.  *
13.  * @type {Array.<namespaceA~Server>}
14.  */
15. var servers = [
16.     {
17.         host: '1.2.3.4',
18.         port: 8080
19.     },
20.     {
21.         host: '1.2.3.5',
22.         port: 8081
23.     }
24. ];

```

2.4.12 AMD 模块注释

[强制] AMD 模块使用 `@module` 或 `@exports` 标识。

解释：

`@exports` 与 `@module` 都可以用来标识模块，区别在于 `@module` 可以省略模块名称。而只使用 `@exports` 时在 `namepaths` 中可以省略 `module:` 前缀。

示例：

```

1. define(
2.     function (require) {
3.
4.         /**
5.          * foo description
6.          *
7.          * @exports Foo
8.          */
9.         var foo = {
10.             // TODO

```

```
11.     };
12.
13.     /**
14.      * baz description
15.      *
16.      * @return {boolean} return description
17.      */
18.     foo.baz = function () {
19.         // TODO
20.     };
21.
22.     return foo;
23.
24. }
25. );
```

也可以在 exports 变量前使用 @module 标识:

```
1.  define(
2.      function (require) {
3.
4.          /**
5.           * module description.
6.           *
7.           * @module foo
8.           */
9.          var exports = {};
10.
11.
12.          /**
13.           * bar description
14.           *
15.           */
16.          exports.bar = function () {
17.              // TODO
18.          };
19.
20.          return exports;
21.      }
22. );
```

如果直接使用 factory 的 exports 参数, 还可以:

```

1.  /**
2.   * module description.
3.   *
4.   * @module
5.   */
6.  define(
7.      function (require, exports) {
8.
9.          /**
10.           * bar description
11.           *
12.           */
13.          exports.bar = function () {
14.              // TODO
15.          };
16.          return exports;
17.      }
18.  );

```

[强制] 对于已使用 `@module` 标识为 AMD模块 的引用, 在 `namepaths` 中必须增加 `module:` 作前缀。

解释:

`namepaths` 没有 `module:` 前缀时, 生成的文档中将无法正确生成链接。

示例:

```

1.  /**
2.   * 点击处理
3.   *
4.   * @fires module:Select#change
5.   * @private
6.   */
7.  Select.prototype.clickHandler = function () {
8.      /**
9.       * 值变更时触发
10.       *
11.       * @event module:Select#change
12.       * @param {Object} e e描述
13.       * @param {string} e.before before描述
14.       * @param {string} e.after after描述
15.       */

```

```

16.     this.fire(
17.         'change',
18.         {
19.             before: 'foo',
20.             after: 'bar'
21.         }
22.     );
23. };

```

[建议] 对于类定义的模块，可以使用 `@alias` 标识构造函数。

示例：

```

1.  /**
2.   * A module representing a jacket.
3.   * @module jacket
4.   */
5.  define(
6.      function () {
7.
8.          /**
9.           * @class
10.          * @alias module:jacket
11.          */
12.          var Jacket = function () {
13.
14.
15.          };
16.
17.          return Jacket;
18.      }
19.  );

```

[建议] 多模块定义时，可以使用 `@exports` 标识各个模块。

示例：

```

1.  // one module
2.  define('html/utills',
3.      /**
4.       * Utility functions to ease working with DOM elements.
5.       * @exports html/utills
6.       */
7.      function () {
8.          var exports = {

```

```

9.         };
10.
11.         return exports;
12.     }
13. );
14.
15. // another module
16. define('tag',
17.     /** @exports tag */
18.     function () {
19.         var exports = {
20.         };
21.
22.         return exports;
23.     }
24. );

```

[建议] 对于 exports 为 Object 的模块，可以使用 `@namespace` 标识。

解释：

使用 `@namespace` 而不是 `@module` 或 `@exports` 时，对模块的引用可以省略 `module:` 前缀。

[建议] 对于 exports 为类名的模块，使用 `@class` 和 `@exports` 标识。

示例：

```

1.
2. // 只使用 @class Bar 时，类方法和属性都必须增加 @name Bar#methodName 来标识，与
   // @exports 配合可以免除这一麻烦，并且在引用时可以省去 module: 前缀。
3. // 另外需要注意类名需要使用 var 定义的方式。
4.
5. /**
6.  * Bar description
7.  *
8.  * @see foo
9.  * @exports Bar
10.  * @class
11.  */
12. var Bar = function () {
13.     // TODO
14. };
15.
16. /**

```

```
17.  * baz description
18.  *
19.  * @return {(string|Array)} return description
20.  */
21. Bar.prototype.baz = function () {
22.    // TODO
23. };
```

2.4.13 细节注释

对于内部实现、不容易理解的逻辑说明、摘要信息等，我们可能需要编写细节注释。

[建议] 细节注释遵循单行注释的格式。说明必须换行时，每行是一个单行注释的起始。

示例：

```
1. function foo(p1, p2, opt_p3) {
2.    // 这里对具体内部逻辑进行说明
3.    // 说明太长需要换行
4.    for (...) {
5.        ....
6.    }
7. }
```

[强制] 有时我们会使用一些特殊标记进行说明。特殊标记必须使用单行注释的形式。下面列举了一些常用标记：

解释：

1. TODO：有功能待实现。此时需要对将要实现的功能进行简单说明。
2. FIXME：该处代码运行没问题，但可能由于时间赶或者其他原因，需要修正。此时需要对如何修正进行简单说明。
3. HACK：为修正某些问题而写的不太好或者使用了某些诡异手段的代码。此时需要对思路或诡异手段进行描述。
4. XXX：该处存在陷阱。此时需要对陷阱进行描述。

3 语言特性

3.1 变量

[强制] 变量、函数在使用前必须先定义。

解释：

不通过 `var` 定义变量将导致变量污染全局环境。

示例：

```
1. // good
2. var name = 'MyName';
3.
4. // bad
5. name = 'MyName';
```

原则上不建议使用全局变量，对于已有的全局变量或第三方框架引入的全局变量，需要根据检查工具的语法标识。

示例：

```
1. /* globals jQuery */
2. var element = jQuery('#element-id');
```

[强制] 每个 `var` 只能声明一个变量。

解释：

一个 `var` 声明多个变量，容易导致较长的行长度，并且在修改时容易造成逗号和分号的混淆。

示例：

```
1. // good
2. var hangModules = [];
3. var missModules = [];
4. var visited = {};
5.
6. // bad
7. var hangModules = [],
8.     missModules = [],
9.     visited = {};
```

[强制] 变量必须 `即用即声明`，不得在函数或其它形式的代码块起始位置统一声明所有变量。

解释：

变量声明与使用的距离越远，出现的跨度越大，代码的阅读与维护成本越高。虽然JavaScript的变量是函数作用域，还是应该根据编程中的意图，缩小变量出现的距离空间。

示例:

```
1.  // good
2.  function kv2List(source) {
3.      var list = [];
4.
5.      for (var key in source) {
6.          if (source.hasOwnProperty(key)) {
7.              var item = {
8.                  k: key,
9.                  v: source[key]
10.             };
11.
12.             list.push(item);
13.         }
14.     }
15.
16.     return list;
17. }
18.
19. // bad
20. function kv2List(source) {
21.     var list = [];
22.     var key;
23.     var item;
24.
25.     for (key in source) {
26.         if (source.hasOwnProperty(key)) {
27.             item = {
28.                 k: key,
29.                 v: source[key]
30.             };
31.
32.             list.push(item);
33.         }
34.     }
35.
36.     return list;
37. }
```

3.2 条件

[强制] 在 Equality Expression 中使用类型严格的 `===`。仅当判断 `null` 或 `undefined` 时，允许使用 `== null`。

解释：

使用 `===` 可以避免等于判断中隐式的类型转换。

示例：

```
1. // good
2. if (age === 30) {
3.     // .....
4. }
5.
6. // bad
7. if (age == 30) {
8.     // .....
9. }
```

[建议] 尽可能使用简洁的表达式。

示例：

```
1. // 字符串为空
2.
3. // good
4. if (!name) {
5.     // .....
6. }
7.
8. // bad
9. if (name === '') {
10.    // .....
11. }
```

```
1. // 字符串非空
2.
3. // good
4. if (name) {
5.     // .....
6. }
7.
```

```
8.  // bad
9.  if (name !== '') {
10.    // .....
11. }
```

```
1.  // 数组非空
2.
3.  // good
4.  if (collection.length) {
5.    // .....
6.  }
7.
8.  // bad
9.  if (collection.length > 0) {
10.    // .....
11. }
```

```
1.  // 布尔不成立
2.
3.  // good
4.  if (!notTrue) {
5.    // .....
6.  }
7.
8.  // bad
9.  if (notTrue === false) {
10.    // .....
11. }
```

```
1.  // null 或 undefined
2.
3.  // good
4.  if (noValue == null) {
5.    // .....
6.  }
7.
8.  // bad
9.  if (noValue === null || typeof noValue === 'undefined') {
10.    // .....
11. }
```

[建议] 按执行频率排列分支的顺序。

解释：

按执行频率排列分支的顺序好处是：

1. 阅读的人容易找到最常见的情况，增加可读性。
2. 提高执行效率。

[建议] 对于相同变量或表达式的多值条件，用 `switch` 代替 `if`。

示例：

```
1. // good
2. switch (typeof variable) {
3.     case 'object':
4.         // .....
5.         break;
6.     case 'number':
7.     case 'boolean':
8.     case 'string':
9.         // .....
10.        break;
11. }
12.
13. // bad
14. var type = typeof variable;
15. if (type === 'object') {
16.     // .....
17. }
18. else if (type === 'number' || type === 'boolean' || type === 'string') {
19.     // .....
20. }
```

[建议] 如果函数或全局中的 `else` 块后没有任何语句，可以删除 `else`。

示例：

```
1. // good
2. function getName() {
3.     if (name) {
4.         return name;
5.     }
6. }
```

```
7.     return 'unnamed';
8. }
9.
10. // bad
11. function getName() {
12.     if (name) {
13.         return name;
14.     }
15.     else {
16.         return 'unnamed';
17.     }
18. }
```

3.3 循环

[建议] 不要在循环体中包含函数表达式，事先将函数提取到循环体外。

解释：

循环体中的函数表达式，运行过程中会生成循环次数个函数对象。

示例：

```
1. // good
2. function clicker() {
3.     // .....
4. }
5.
6. for (var i = 0, len = elements.length; i < len; i++) {
7.     var element = elements[i];
8.     addListener(element, 'click', clicker);
9. }
10.
11.
12. // bad
13. for (var i = 0, len = elements.length; i < len; i++) {
14.     var element = elements[i];
15.     addListener(element, 'click', function () {});
16. }
```

[建议] 对循环内多次使用的不变值，在循环外用变量缓存。

示例：

```
1. // good
2. var width = wrap.offsetWidth + 'px';
3. for (var i = 0, len = elements.length; i < len; i++) {
4.     var element = elements[i];
5.     element.style.width = width;
6.     // .....
7. }
8.
9.
10. // bad
11. for (var i = 0, len = elements.length; i < len; i++) {
12.     var element = elements[i];
13.     element.style.width = wrap.offsetWidth + 'px';
14.     // .....
15. }
```

[建议] 对有序集合进行遍历时，缓存 `length`。

解释：

虽然现代浏览器都对数组长度进行了缓存，但对于一些宿主对象和老旧浏览器的数组对象，在每次 `length` 访问时会动态计算元素个数，此时缓存 `length` 能有效提高程序性能。

示例：

```
1. for (var i = 0, len = elements.length; i < len; i++) {
2.     var element = elements[i];
3.     // .....
4. }
```

[建议] 对有序集合进行顺序无关的遍历时，使用逆序遍历。

解释：

逆序遍历可以节省变量，代码比较优化。

示例：

```
1. var len = elements.length;
2. while (len--) {
3.     var element = elements[len];
4.     // .....
5. }
```

3.4 类型

3.4.1 类型检测

[建议] 类型检测优先使用 `typeof`。对象类型检测使用 `instanceof`。`null` 或 `undefined` 的检测使用 `== null`。

示例：

```
1. // string
2. typeof variable === 'string'
3.
4. // number
5. typeof variable === 'number'
6.
7. // boolean
8. typeof variable === 'boolean'
9.
10. // Function
11. typeof variable === 'function'
12.
13. // Object
14. typeof variable === 'object'
15.
16. // RegExp
17. variable instanceof RegExp
18.
19. // Array
20. variable instanceof Array
21.
22. // null
23. variable === null
24.
25. // null or undefined
26. variable == null
27.
28. // undefined
29. typeof variable === 'undefined'
```

3.4.2 类型转换

[建议] 转换成 `string` 时，使用 `+ ''`。

示例：

```
1. // good
2. num + '';
3.
4. // bad
5. new String(num);
6. num.toString();
7. String(num);
```

[建议] 转换成 `number` 时，通常使用 `+`。

示例：

```
1. // good
2. +str;
3.
4. // bad
5. Number(str);
```

[建议] `string` 转换成 `number`，要转换的字符串结尾包含非数字并期望忽略时，使用 `parseInt`。

示例：

```
1. var width = '200px';
2. parseInt(width, 10);
```

[强制] 使用 `parseInt` 时，必须指定进制。

示例：

```
1. // good
2. parseInt(str, 10);
3.
4. // bad
5. parseInt(str);
```

[建议] 转换成 `boolean` 时，使用 `!!`。

示例：

```
1. var num = 3.14;
```

```
2. !!num;
```

[建议] `number` 去除小数点, 使用 `Math.floor` / `Math.round` / `Math.ceil`, 不使用 `parseInt`。

示例:

```
1. // good
2. var num = 3.14;
3. Math.ceil(num);
4.
5. // bad
6. var num = 3.14;
7. parseInt(num, 10);
```

3.5 字符串

[强制] 字符串开头和结束使用单引号 `'`。

解释:

1. 输入单引号不需要按住 `shift`, 方便输入。
2. 实际使用中, 字符串经常用来拼接 HTML。为方便 HTML 中包含双引号而不需要转义写法。

示例:

```
1. var str = '我是一个字符串';
2. var html = '<div class="cls">拼接HTML可以省去双引号转义</div>';
```

[建议] 使用 `数组` 或 `+` 拼接字符串。

解释:

1. 使用 `+` 拼接字符串, 如果拼接的全部是 `StringLiteral`, 压缩工具可以对其进行自动合并的优化。所以, 静态字符串建议使用 `+` 拼接。
2. 在现代浏览器下, 使用 `+` 拼接字符串, 性能较数组的方式要高。
3. 如需要兼顾老旧浏览器, 应尽量使用数组拼接字符串。

示例:

```
1. // 使用数组拼接字符串
2. var str = [
3.     // 推荐换行开始并缩进开始第一个字符串, 对齐代码, 方便阅读.
4.     '<ul>',
```

```

5.         '<li>第一项</li>',
6.         '<li>第二项</li>',
7.     '</ul>'
8. ].join('');
9.
10. // 使用 `` 拼接字符串
11. var str2 = '' // 建议第一个为空字符串，第二个换行开始并缩进开始，对齐代码，方便阅读
12.     + '<ul>',
13.     + '<li>第一项</li>',
14.     + '<li>第二项</li>',
15.     + '</ul>';

```

[建议] 使用字符串拼接的方式生成HTML，需要根据语境进行合理的转义。

解释：

在 `JavaScript` 中拼接，并且最终将输出到页面中的字符串，需要进行合理转义，以防止安全漏洞。下面的示例代码为场景说明，不能直接运行。

示例：

```

1. // HTML 转义
2. var str = '<p>' + htmlEncode(content) + '</p>';
3.
4. // HTML 转义
5. var str = '<input type="text" value="' + htmlEncode(value) + '">';
6.
7. // URL 转义
8. var str = '<a href="/?key=' + htmlEncode(urlEncode(value)) + '">link</a>';
9.
10. // JavaScript字符串 转义 + HTML 转义
11. var str = '<button onclick="check(\' ' + htmlEncode(strLiteral(name)) + '\')">提交</button>';

```

[建议] 复杂的数据到视图字符串的转换过程，选用一种模板引擎。

解释：

使用模板引擎有如下好处：

1. 在开发过程中专注于数据，将视图生成的过程由另外一个层级维护，使程序逻辑结构更清晰。
2. 优秀的模板引擎，通过模板编译技术和高质量的编译产物，能获得比手工拼接字符串更高的性能。
3. 模板引擎能方便的对动态数据进行相应的转义，部分模板引擎默认进行HTML转义，安全性更好。

- artTemplate: 体积较小, 在所有环境下性能高, 语法灵活。
- dot.js: 体积小, 在现代浏览器下性能高, 语法灵活。
- etpl: 体积较小, 在所有环境下性能高, 模板复用性高, 语法灵活。
- handlebars: 体积大, 在所有环境下性能高, 扩展性高。
- hogon: 体积小, 在现代浏览器下性能高。
- nunjucks: 体积较大, 性能一般, 模板复用性高。

3.6 对象

[强制] 使用对象字面量 `{}` 创建新 `Object`。

示例:

```
1. // good
2. var obj = {};
3.
4. // bad
5. var obj = new Object();
```

[建议] 对象创建时, 如果一个对象的所有 `属性` 均可以不添加引号, 建议所有 `属性` 不添加引号。

示例:

```
1. var info = {
2.     name: 'someone',
3.     age: 28
4. };
```

[建议] 对象创建时, 如果任何一个 `属性` 需要添加引号, 则所有 `属性` 建议添加 `'`。

解释:

如果属性不符合 `Identifier` 和 `NumberLiteral` 的形式, 就需要以 `StringLiteral` 的形式提供。

示例:

```
1. // good
2. var info = {
3.     'name': 'someone',
4.     'age': 28,
5.     'more-info': '...'
```

```
6.  };
7.
8.  // bad
9.  var info = {
10.     name: 'someone',
11.     age: 28,
12.     'more-info': '...'
13.  };

```

[强制] 不允许修改和扩展任何原生对象和宿主对象的原型。

示例：

```
1.  // 以下行为绝对禁止
2.  String.prototype.trim = function () {
3.  };

```

[建议] 属性访问时，尽量使用 `.`。

解释：

属性名符合 Identifier 的要求，就可以通过 `.` 来访问，否则就只能通过 `[expr]` 方式访问。

通常在 JavaScript 中声明的对象，属性命名是使用 Camel 命名法，用 `.` 来访问更清晰简洁。部分特殊的属性（比如来自后端的 JSON ），可能采用不寻常的命名方式，可以通过 `[expr]` 方式访问。

示例：

```
1.  info.age;
2.  info['more-info'];

```

[建议] `for in` 遍历对象时，使用 `hasOwnProperty` 过滤掉原型中的属性。

示例：

```
1.  var newInfo = {};
2.  for (var key in info) {
3.      if (info.hasOwnProperty(key)) {
4.          newInfo[key] = info[key];
5.      }
6.  }

```

3.7 数组

[强制] 使用数组字面量 `[]` 创建新数组，除非想要创建的是指定长度的数组。

示例：

```
1. // good
2. var arr = [];
3.
4. // bad
5. var arr = new Array();
```

[强制] 遍历数组不使用 `for in`。

解释：

数组对象可能存在数字以外的属性，这种情况下 `for in` 不会得到正确结果。

示例：

```
1. var arr = ['a', 'b', 'c'];
2.
3. // 这里仅作演示，实际中应使用 Object 类型
4. arr.other = 'other things';
5.
6. // 正确的遍历方式
7. for (var i = 0, len = arr.length; i < len; i++) {
8.     console.log(i);
9. }
10.
11. // 错误的遍历方式
12. for (var i in arr) {
13.     console.log(i);
14. }
```

[建议] 不因为性能的原因自己实现数组排序功能，尽量使用数组的 `sort` 方法。

解释：

自己实现的常规排序算法，在性能上并不优于数组默认的 `sort` 方法。以下两种场景可以自己实现排序：

1. 需要稳定的排序算法，达到严格一致的排序结果。
2. 数据特点鲜明，适合使用桶排。

[建议] 清空数组使用 `.length = 0`。

3.8 函数

3.8.1 函数长度

[建议] 一个函数的长度控制在 `50` 行以内。

解释：

将过多的逻辑单元混在一个大函数中，易导致难以维护。一个清晰易懂的函数应该完成单一的逻辑单元。复杂的操作应进一步抽取，通过函数的调用来体现流程。

特定算法等不可分割的逻辑允许例外。

示例：

```
1. function syncViewStateOnUserAction() {
2.     if (x.checked) {
3.         y.checked = true;
4.         z.value = '';
5.     }
6.     else {
7.         y.checked = false;
8.     }
9.
10.    if (a.value) {
11.        warning.innerText = '';
12.        submitButton.disabled = false;
13.    }
14.    else {
15.        warning.innerText = 'Please enter it';
16.        submitButton.disabled = true;
17.    }
18. }
19.
20. // 直接阅读该函数会难以明确其主线逻辑，因此下方是一种更合理的表达方式：
21.
22. function syncViewStateOnUserAction() {
23.     syncXStateToView();
24.     checkAAvailability();
25. }
26.
27. function syncXStateToView() {
```

```

28.     y.checked = x.checked;
29.
30.     if (x.checked) {
31.         z.value = '';
32.     }
33. }
34.
35. function checkAAvailability() {
36.     if (a.value) {
37.         clearWarnignForA();
38.     }
39.     else {
40.         displayWarningForAMissing();
41.     }
42. }

```

3.8.2 参数设计

[建议] 一个函数的参数控制在 **6** 个以内。

解释：

除去不定长参数以外，函数具备不同逻辑意义的参数建议控制在 **6** 个以内，过多参数会导致维护难度增大。

某些情况下，如使用 AMD Loader 的 **require** 加载多个模块时，其 **callback** 可能会存在较多参数，因此对函数参数的个数不做强制限制。

[建议] 通过 **options** 参数传递非数据输入型参数。

解释：

有些函数的参数并不是作为算法的输入，而是对算法的某些分支条件判断之用，此类参数建议通过一个 **options** 参数传递。

如下函数：

```

1.  /**
2.   * 移除某个元素
3.   *
4.   * @param {Node} element 需要移除的元素
5.   * @param {boolean} removeEventListeners 是否同时将所有注册在元素上的事件移除
6.   */
7.  function removeElement(element, removeEventListeners) {

```



```

8.     element.parent.removeChild(element);
9.
10.    if (removeEventListeners) {
11.        element.clearEventListeners();
12.    }
13. }

```

可以转换为下面的签名：

```

1.  /**
2.   * 移除某个元素
3.   *
4.   * @param {Node} element 需要移除的元素
5.   * @param {Object} options 相关的逻辑配置
6.   * @param {boolean} options.removeEventListeners 是否同时将所有注册在元素上的事件移除
7.   */
8.  function removeElement(element, options) {
9.      element.parent.removeChild(element);
10.
11.      if (options.removeEventListeners) {
12.          element.clearEventListeners();
13.      }
14.  }

```

这种模式有几个显著的优势：

- `boolean` 型的配置项具备名称，从调用的代码上更易理解其表达的逻辑意义。
- 当配置项有增长时，无需无休止地增加参数个数，不会出现 `removeElement(element, true, false, false, 3)` 这样难以理解的调用代码。
- 当部分配置参数可选时，多个参数的形式非常难处理重载逻辑，而使用一个 `options` 对象只需判断属性是否存在，实现得以简化。

3.8.3 闭包

[建议] 在适当的时候将闭包内大对象置为 `null`。

解释：

在 JavaScript 中，无需特别的关键词就可以使用闭包，一个函数可以任意访问在其定义的作用域外的变量。需要注意的是，函数的作用域是静态的，即在定义时决定，与调用的时机和方式没有任何关系。

闭包会阻止一些变量的垃圾回收，对于较老旧的 JavaScript 引擎，可能导致外部所有变量均无法回

收。

首先一个较为明确的结论是，以下内容会影响到闭包内变量的回收：

- 嵌套的函数中是否有使用该变量。
- 嵌套的函数中是否有 直接调用 `eval`。
- 是否使用了 `with` 表达式。

Chakra、V8 和 SpiderMonkey 将受以上因素的影响，表现出不尽相同又较为相似的回收策略，而 JScript.dll 和 Carakan 则完全没有这方面的优化，会完整保留整个 `LexicalEnvironment` 中的所有变量绑定，造成一定的内存消耗。

由于对闭包内变量有回收优化策略的 Chakra、V8 和 SpiderMonkey 引擎的行为较为相似，因此可以总结如下，当返回一个函数 `fn` 时：

1. 如果 `fn` 的 `[[Scope]]` 是 `ObjectEnvironment` (with 表达式生成 `ObjectEnvironment`，函数和 `catch` 表达式生成 `DeclarativeEnvironment`)，则：
 - i. 如果是 V8 引擎，则退出全过程。
 - ii. 如果是 SpiderMonkey，则处理该 `ObjectEnvironment` 的外层 `LexicalEnvironment`。
2. 获取当前 `LexicalEnvironment` 下的所有类型为 `Function` 的对象，对于每一个 `Function` 对象，分析其 `FunctionBody`：
 - i. 如果 `FunctionBody` 中含有 直接调用 `eval`，则退出全过程。
 - ii. 否则得到所有的 `Identifier`。
 - iii. 对于每一个 `Identifier`，设其为 `name`，根据查找变量引用的规则，从 `LexicalEnvironment` 中找出名称为 `name` 的绑定 `binding`。
 - iv. 对 `binding` 添加 `notSwap` 属性，其值为 `true`。
3. 检查当前 `LexicalEnvironment` 中的每一个变量绑定，如果该绑定有 `notSwap` 属性且值为 `true`，则：
 - i. 如果是 V8 引擎，删除该绑定。
 - ii. 如果是 SpiderMonkey，将该绑定的值设为 `undefined`，将删除 `notSwap` 属性。

对于 Chakra 引擎，暂无法得知是按 V8 的模式还是按 SpiderMonkey 的模式进行。

如果有 非常庞大 的对象，且预计会在 老旧的引擎 中执行，则使用闭包时，注意将闭包不需要的对象置为空引用。

[建议] 使用 `IIFE` 避免 `Lift 效应`。

解释：

在引用函数外部变量时，函数执行时外部变量的值由运行时决定而非定义时，最典型的场景如下：

```
1. var tasks = [];
```

```

2.  for (var i = 0; i < 5; i++) {
3.      tasks[tasks.length] = function () {
4.          console.log('Current cursor is at ' + i);
5.      };
6.  }
7.
8.  var len = tasks.length;
9.  while (len--) {
10.      tasks[len]();
11.  }

```

以上代码对 `tasks` 中的函数的执行均会输出 `Current cursor is at 5`，往往不符合预期。

此现象称为 **Lift 效应**。解决的方式是通过额外加上一层闭包函数，将需要的外部变量作为参数传递来解除变量的绑定关系：

```

1.  var tasks = [];
2.  for (var i = 0; i < 5; i++) {
3.      // 注意有一层额外的闭包
4.      tasks[tasks.length] = (function (i) {
5.          return function () {
6.              console.log('Current cursor is at ' + i);
7.          };
8.      })(i);
9.  }
10.
11. var len = tasks.length;
12. while (len--) {
13.     tasks[len]();
14. }

```

3.8.4 空函数

[建议] 空函数不使用 `new Function()` 的形式。

示例：

```
1. var emptyFunction = function () {};
```

[建议] 对于性能有高要求的场合，建议存在一个空函数的常量，供多处使用共享。

示例：

```

1.  var EMPTY_FUNCTION = function () {};
2.
3.  function MyClass() {
4.  }
5.
6.  MyClass.prototype.abstractMethod = EMPTY_FUNCTION;
7.  MyClass.prototype.hooks.before = EMPTY_FUNCTION;
8.  MyClass.prototype.hooks.after = EMPTY_FUNCTION;

```

3.9 面向对象

[强制] 类的继承方案，实现时需要修正 `constructor`。

解释：

通常使用其他 library 的类继承方案都会进行 `constructor` 修正。如果是自己实现的类继承方案，需要进行 `constructor` 修正。

示例：

```

1.  /**
2.   * 构建类之间的继承关系
3.   *
4.   * @param {Function} subClass 子类函数
5.   * @param {Function} superClass 父类函数
6.   */
7.  function inherits(subClass, superClass) {
8.      var F = new Function();
9.      F.prototype = superClass.prototype;
10.     subClass.prototype = new F();
11.     subClass.prototype.constructor = subClass;
12. }

```

[建议] 声明类时，保证 `constructor` 的正确性。

示例：

```

1.  function Animal(name) {
2.      this.name = name;
3.  }
4.
5.  // 直接prototype等于对象时，需要修正constructor
6.  Animal.prototype = {

```

```
7.     constructor: Animal,
8.
9.     jump: function () {
10.         alert('animal ' + this.name + ' jump');
11.     }
12. };
13.
14. // 这种方式扩展prototype则无需理会constructor
15. Animal.prototype.jump = function () {
16.     alert('animal ' + this.name + ' jump');
17. };
```

[建议] 属性在构造函数中声明，方法在原型中声明。

解释：

原型对象的成员被所有实例共享，能节约内存占用。所以编码时我们应该遵守这样的原则：原型对象包含程序不会修改的成员，如方法函数或配置项。

```
1. function TextNode(value, engine) {
2.     this.value = value;
3.     this.engine = engine;
4. }
5.
6. TextNode.prototype.clone = function () {
7.     return this;
8. };
```

[强制] 自定义事件的事件名 必须全小写。

解释：

在 JavaScript 广泛应用的浏览器环境，绝大多数 DOM 事件名称都是全小写的。为了遵循大多数 JavaScript 开发者的习惯，在设计自定义事件时，事件名也应该全小写。

[强制] 自定义事件只能有一个 event 参数。如果事件需要传递较多信息，应仔细设计事件对象。

解释：

一个事件对象的好处有：

1. 顺序无关，避免事件监听者需要记忆参数顺序。
2. 每个事件信息都可以根据需要提供或者不提供，更自由。
3. 扩展方便，未来添加事件信息时，无需考虑会破坏监听器参数形式而无法向后兼容。

[建议] 设计自定义事件时，应考虑禁止默认行为。

解释：

常见禁止默认行为的方式有两种：

1. 事件监听函数中 `return false`。
2. 事件对象中包含禁止默认行为的方法，如 `preventDefault`。

3.10 动态特性

3.10.1 eval

[强制] 避免使用直接 `eval` 函数。

解释：

直接 `eval`，指的是以函数方式调用 `eval` 的调用方法。直接 `eval` 调用执行代码的作用域为本地作用域，应当避免。

如果有特殊情况需要使用直接 `eval`，需在代码中用详细的注释说明为何必须使用直接 `eval`，不能使用其它动态执行代码的方式，同时需要其他资深工程师进行 Code Review。

[建议] 尽量避免使用 `eval` 函数。

3.10.2 动态执行代码

[建议] 使用 `new Function` 执行动态代码。

解释：

通过 `new Function` 生成的函数作用域是全局使用域，不会影响当前的本地作用域。如果有动态代码执行的需求，建议使用 `new Function`。

示例：

```
1. var handler = new Function('x', 'y', 'return x + y;');
2. var result = handler($('#x').val(), $('#y').val());
```

3.10.3 with

[建议] 尽量不要使用 `with`。

解释：

使用 `with` 可能会增加代码的复杂度，不利于阅读和管理；也会对性能有影响。大多数使用 `with` 的场景都能使用其他方式较好的替代。所以，尽量不要使用 `with`。

3.10.4 delete

[建议] 减少 `delete` 的使用。

解释：

如果没有特别的需求，减少或避免使用 `delete`。`delete` 的使用会破坏部分 JavaScript 引擎的性能优化。

[建议] 处理 `delete` 可能产生的异常。

解释：

对于有被遍历需求，且值 `null` 被认为具有业务逻辑意义的值的对象，移除某个属性必须使用 `delete` 操作。

在严格模式或 IE 下使用 `delete` 时，不能被删除的属性会抛出异常，因此在不确定属性是否可以删除的情况下，建议添加 `try-catch` 块。

示例：

```
1. try {  
2.     delete o.x;  
3. }  
4. catch (deleteError) {  
5.     o.x = null;  
6. }
```

3.10.5 对象属性

[建议] 避免修改外部传入的对象。

解释：

JavaScript 因其脚本语言的动态特性，当一个对象未被 `seal` 或 `freeze` 时，可以任意添加、删除、修改属性值。

但是随意地对 非自身控制的对象 进行修改，很容易造成代码在不可预知的情况下出现问题。因此，设计良好的组件、函数应该避免对外部传入的对象的修改。

下面代码的 `selectNode` 方法修改了由外部传入的 `datasource` 对象。如果 `datasource` 用在其它场合（如另一个 `Tree` 实例）下，会造成状态的混乱。

```
1. function Tree(datasource) {  
2.     this.datasource = datasource;
```

```

3.  }
4.
5.  Tree.prototype.selectNode = function (id) {
6.      // 从datasource中找出节点对象
7.      var node = this.findNode(id);
8.      if (node) {
9.          node.selected = true;
10.         this.flushView();
11.     }
12. };

```

对于此类场景，需要使用额外的对象来维护，使用由自身控制，不与外部产生任何交互的 **selectedNodeIndex** 对象来维护节点的选中状态，不对 **datasource** 作任何修改。

```

1.  function Tree(datasource) {
2.      this.datasource = datasource;
3.      this.selectedNodeIndex = {};
4.  }
5.
6.  Tree.prototype.selectNode = function (id) {
7.
8.      // 从datasource中找出节点对象
9.      var node = this.findNode(id);
10.
11.      if (node) {
12.          this.selectedNodeIndex[id] = true;
13.          this.flushView();
14.      }
15.
16.  };

```

除此之外，也可以通过 `deepClone` 等手段将自身维护的对象与外部传入的分离，保证不会相互影响。

[建议] 具备强类型的设计。

解释：

- 如果一个属性被设计为 `boolean` 类型，则不要使用 `1` 或 `0` 作为其值。对于标识性的属性，如对代码体积有严格要求，可以从一开始就设计为 `number` 类型且将 `0` 作为否定值。
- 从 DOM 中取出的值通常为 `string` 类型，如果有对象或函数的接收类型为 `number` 类型，提前作好转换，而不是期望对象、函数可以处理多类型的值。

4 浏览器环境

4.1 模块化

4.1.1 AMD

[强制] 使用 `AMD` 作为模块定义。

解释：

AMD 作为由社区认可的模块定义形式，提供多种重载提供灵活的使用方式，并且绝大多数优秀的 Library 都支持 AMD，适合作为规范。

目前，比较成熟的 AMD Loader 有：

- 官方实现的 `requirejs`
- 百度自己实现的 `esl`

[强制] 模块 `id` 必须符合标准。

解释：

模块 `id` 必须符合以下约束条件：

1. 类型为 `string`，并且是由 `/` 分割的一系列 `terms` 来组成。例如：`this/is/a/module`。
2. `term` 应该符合 `[a-zA-Z0-9_-]+` 规则。
3. 不应该有 `.js` 后缀。
4. 跟文件的路径保持一致。

4.1.2 define

[建议] 定义模块时不要指明 `id` 和 `dependencies`。

解释：

在 AMD 的设计思想里，模块名称是和所在路径相关的，匿名的模块更利于封包和迁移。模块依赖应在模块定义内部通过 `local require` 引用。

所以，推荐使用 `define(factory)` 的形式进行模块定义。

示例：

```
1. define(  
2.     function (require) {  
3.     }  
4. );
```

[建议] 使用 `return` 来返回模块定义。

解释：

使用 `return` 可以减少 `factory` 接收的参数（不需要接收 `exports` 和 `module`），在没有 `AMD Loader` 的场景下也更容易进行简单的处理来伪造一个 `Loader`。

示例：

```
1. define(  
2.     function (require) {  
3.         var exports = {};  
4.  
5.         // ...  
6.  
7.         return exports;  
8.     }  
9. );
```

4.1.3 require

[强制] 全局运行环境中，`require` 必须以 `async require` 形式调用。

解释：

模块的加载过程是异步的，同步调用并无法保证得到正确的结果。

示例：

```
1. // good  
2. require(['foo'], function (foo) {  
3. });  
4.  
5. // bad  
6. var foo = require('foo');
```

[强制] 模块定义中只允许使用 `local require`，不允许使用 `global require`。

解释：

1. 在模块定义中使用 `global require`，对封装性是一种破坏。
2. 在 `AMD` 里，`global require` 是可以被重命名的。并且 `Loader` 甚至没有全局的 `require` 变量，而是用 `Loader` 名称做为 `global require`。模块定义不应该依赖使用的

Loader。

[强制] Package 在实现时，内部模块的 `require` 必须使用 `relative id`。

解释：

对于任何可能通过 发布-引入 的形式复用的第三方库、框架、包，开发者所定义的名称不代表使用者使用的名称。因此不要基于任何名称的假设。在实现源码中，`require` 自身的其它模块时使用 `relative id`。

示例：

```
1. define(
2.     function (require) {
3.         var util = require('./util');
4.     }
5. );
```

[建议] 不会被调用的依赖模块，在 `factory` 开始处统一 `require`。

解释：

有些模块是依赖的模块，但不会在模块实现中被直接调用，最为典型的是 `css` / `js` / `tpl` 等 Plugin 所引入的外部内容。此类内容建议放在模块定义最开始处统一引用。

示例：

```
1. define(
2.     function (require) {
3.         require('css!foo.css');
4.         require('tpl!bar.tpl.html');
5.
6.         // ...
7.     }
8. );
```

4.2 DOM

4.2.1 元素获取

[建议] 对于单个元素，尽可能使用 `document.getElementById` 获取，避免使用 `document.all`。

[建议] 对于多个元素的集合，尽可能使用 `context.getElementsByTagName` 获取。其中 `context` 可以为 `document` 或其他元素。指定 `tagName` 参数为 `*` 可以获得所有子元素。

[建议] 遍历元素集合时，尽量缓存集合长度。如需多次操作同一集合，则应将集合转为数组。

解释：

原生获取元素集合的结果并不直接引用 DOM 元素，而是对索引进行读取，所以 DOM 结构的改变会实时反映到结果中。

示例：

```
1. <div></div>
2. <span></span>
3.
4. <script>
5. var elements = document.getElementsByTagName('*');
6.
7. // 显示为 DIV
8. alert(elements[0].tagName);
9.
10. var div = elements[0];
11. var p = document.createElement('p');
12. document.body.insertBefore(p, div);
13.
14. // 显示为 P
15. alert(elements[0].tagName);
16. </script>
```

[建议] 获取元素的直接子元素时使用 `children`。避免使用 `childNodes`，除非预期是需要包含文本、注释和属性类型的节点。

4.2.2 样式获取

[建议] 获取元素实际样式信息时，应使用 `getComputedStyle` 或 `currentStyle`。

解释：

通过 `style` 只能获得内联定义或通过 JavaScript 直接设置的样式。通过 CSS class 设置的元素样式无法直接通过 `style` 获取。

4.2.3 样式设置

[建议] 尽可能通过为元素添加预定义的 `className` 来改变元素样式，避免直接操作 `style` 设置。

[强制] 通过 `style` 对象设置元素样式时，对于带单位非 0 值的属性，不允许省略单位。

解释：

除了 IE，标准浏览器会忽略不规范的属性值，导致兼容性问题。

4.2.4 DOM 操作

[建议] 操作 `DOM` 时，尽量减少页面 `reflow`。

解释：

页面 `reflow` 是非常耗时的行为，非常容易导致性能瓶颈。下面一些场景会触发浏览器的`reflow`：

- DOM元素的添加、修改（内容）、删除。
- 应用新的样式或者修改任何影响元素布局的属性。
- Resize浏览器窗口、滚动页面。
- 读取元素的某些属性（`offsetLeft`、`offsetTop`、`offsetHeight`、`offsetWidth`、`scrollTop/Left/Width/Height`、`clientTop/Left/Width/Height`、`getComputedStyle()`、`currentStyle`(in IE)）。

[建议] 尽量减少 `DOM` 操作。

解释：

DOM 操作也是非常耗时的一种操作，减少 DOM 操作有助于提高性能。举一个简单的例子，构建一个列表。我们可以用两种方式：

1. 在循环体中 `createElement` 并 `append` 到父元素中。
2. 在循环体中拼接 HTML 字符串，循环结束后写父元素的 `innerHTML`。

第一种方法看起来比较标准，但是每次循环都会对 DOM 进行操作，性能极低。在这里推荐使用第二种方法。

4.2.5 DOM 事件

[建议] 优先使用 `addEventListener / attachEvent` 绑定事件，避免直接在 HTML 属性中或 DOM 的 `expando` 属性绑定事件处理。

解释：

`expando` 属性绑定事件容易导致互相覆盖。

[建议] 使用 `addEventListener` 时第三个参数使用 `false`。

解释：

标准浏览器中的 `addEventListener` 可以通过第三个参数指定两种时间触发模型：冒泡和捕获。而 IE 的 `attachEvent` 仅支持冒泡的事件触发。所以为了保持一致性，通常 `addEventListener` 的第三个参数都为 `false`。

[建议] 在没有事件自动管理的框架支持下，应持有监听器函数的引用，在适当时候（元素释放、页面卸载等）移除添加的监听器。

Javascript编码规范 - ESNext补充篇

JavaScript 编码规范 - ESNext 补充篇（草案）

1 前言

随着 ECMAScript 的不断发展，越来越多更新的语言特性将被使用，给应用的开发带来方便。本文档的目标是使 ECMAScript 新特性的代码风格保持一致，并给予一些实践建议。

本文档仅包含新特性部分。基础部分请遵循 [JavaScript Style Guide](#)。

由于 ECMAScript 依然在快速的不断发展，本文档也将可能随时保持更新。更新内容主要涉及对新增的语言特性的格式规范化、实践指导，引擎与编译器环境变化的使用指导。

虽然本文档是针对 ECMAScript 设计的，但是在使用各种基于 ECMAScript 扩展的语言时(如 JSX、TypeScript 等)，适用的部分也应尽量遵循本文档的约定。

2 代码风格

2.1 文件

[建议] ESNext 语法的 JavaScript 文件使用 `.js` 扩展名。

[强制] 当文件无法使用 `.js` 扩展名时，使用 `.es` 扩展名。

解释：

某些应用开发时，可能同时包含 ES 5和 ESNext 文件，运行环境仅支持 ES5，ESNext 文件需要经过预编译。部分场景下，编译工具的选择可能需要通过扩展名区分，需要重新定义ESNext文件的扩展名。此时，ESNext 文件必须使用 `.es` 扩展名。

但是，更推荐使用其他条件作为是否需要编译的区分：

1. 基于文件内容。
2. 不同类型文件放在不同目录下。

2.2 结构

2.2.1 缩进

[建议] 使用多行模板字符串时遵循缩进原则。当空行与空白字符敏感时，不使用多行模板字符串。

解释：

4 空格为一个缩进，换行后添加一层缩进。将起始和结束的 ``` 符号单独放一行，有助于生成 HTML 时的标签对齐。

为避免破坏缩进的统一，当空行与空白字符敏感时，建议使用 `多个模板字符串` 或 `普通字符串` 进行连接运算，也可使用数组 `join` 生成字符串。

示例：

```

1.  // good
2.  function foo() {
3.      let html = `
4.          <div>
5.              <p></p>
6.              <p></p>
7.          </div>
8.      `;
9.  }
10.
11. // Good
12. function greeting(name) {
13.     return 'Hello, \n'
14.         + `${name.firstName} ${name.lastName}`;
15. }
16.
17. // Bad
18. function greeting(name) {
19.     return `Hello,
20. ${name.firstName} ${name.lastName}`;
21. }
```

2.2.2 空格

[强制] 使用 `generator` 时，`*` 前面不允许有空格，`*` 后面必须有一个空格。

示例：

```

1.  // good
2.  function* caller() {
3.      yield 'a';
4.      yield* callee();
5.      yield 'd';
```



```
6.  }
7.
8.  // bad
9.  function * caller() {
10.    yield 'a';
11.    yield *callee();
12.    yield 'd';
13. }
```

2.2.3 语句

[强制] 类声明结束不允许添加分号。

解释：

与函数声明保持一致。

[强制] 类成员定义中，方法定义后不允许添加分号，成员属性定义后必须添加分号。

解释：

成员属性是当前 **Stage 0** 的标准，如果使用的话，则定义后加上分号。

示例：

```
1.  // good
2.  class Foo {
3.    foo = 3;
4.
5.    bar() {
6.
7.    }
8.  }
9.
10. // bad
11. class Foo {
12.   foo = 3
13.
14.   bar() {
15.
16.   }
17. }
```

[强制] `export` 语句后，不允许出现表示空语句的分号。

解释：

`export` 关键字不影响后续语句类型。

示例：

```

1.  // good
2.  export function foo() {
3.  }
4.
5.  export default function bar() {
6.  }
7.
8.
9.  // bad
10. export function foo() {
11. };
12.
13. export default function bar() {
14. };

```

[强制] 属性装饰器后，可以不加分号的场景，不允许加分号。

解释：

只有一种场景是必须加分号的：当属性 `key` 是 `computed property key` 时，其装饰器必须加分号，否则修饰 `key` 的 `[]` 会做为之前表达式的 `property accessor`。

上面描述的场景，装饰器后需要加分号。其余场景下的属性装饰器后不允许加分号。

示例：

```

1.  // good
2.  class Foo {
3.      @log('INFO')
4.      bar() {
5.
6.      }
7.
8.      @log('INFO');
9.      ['bar' + 2]() {
10.
11.      }
12. }

```

```

13.
14.  // bad
15.  class Foo {
16.      @log('INFO');
17.      bar() {
18.
19.      }
20.
21.      @log('INFO')
22.      ['bar' + 2]() {
23.
24.      }
25.  }

```

[强制] 箭头函数的参数只有一个，并且不包含解构时，参数部分的括号必须省略。

示例：

```

1.  // good
2.  list.map(item => item * 2);
3.
4.  // good
5.  let fetchName = async id => {
6.      let user = await request(`users/${id}`);
7.      return user.fullName;
8.  };
9.
10. // bad
11. list.map((item) => item * 2);
12.
13. // bad
14. let fetchName = async (id) => {
15.     let user = await request(`users/${id}`);
16.     return user.fullName;
17. };

```

[建议] 箭头函数的函数体只有一个单行表达式语句，且作为返回值时，省略 `{}` 和 `return`。

如果单个表达式过长，可以使用 `()` 进行包裹。

示例：

```

1.  // good

```

```

2. list.map(item => item * 2);
3.
4. let foo = () => (
5.     condition
6.     ? returnValueA()
7.     : returnValueB()
8. );
9.
10. // bad
11. list.map(item => {
12.     return item * 2;
13. });

```

[建议] 箭头函数的函数体只有一个 `Object Literal`，且作为返回值时，使用 `()` 包裹。

示例：

```

1. // good
2. list.map(item => ({name: item[0], email: item[1]}));

```

[强制] 解构多个变量时，如果超过行长度限制，每个解构的变量必须单独一行。

解释：

太多的变量解构会让一行的代码非常长，极有可能超过单行长度控制，使代码可读性下降。

示例：

```

1. // good
2. let {
3.     name: personName,
4.     email: personEmail,
5.     sex: personSex,
6.     age: personAge
7. } = person;
8.
9. // bad
10. let {name: personName, email: personEmail,
11.     sex: personSex, age: personAge
12. } = person;

```

3 语言特性

3.1 变量

[强制] 使用 `let` 和 `const` 定义变量，不使用 `var`。

解释：

使用 `let` 和 `const` 定义时，变量作用域范围更明确。

示例：

```
1. // good
2. for (let i = 0; i < 10; i++) {
3.
4. }
5.
6. // bad
7. for (var i = 0; i < 10; i++) {
8.
9. }
```

3.2 解构

[强制] 不要使用3层及以上的解构。

解释：

过多层次的解构会让代码变得难以阅读。

示例：

```
1. // bad
2. let {documentElement: {firstElementChild: {nextSibling}}} = window;
```

[建议] 使用解构减少中间变量。

解释：

常见场景如变量值交换，可能产生中间变量。这种场景推荐使用解构。

示例：

```
1. // good
2. [x, y] = [y, x];
3.
```

```
4. // bad
5. let temp = x;
6. x = y;
7. y = temp;
```

[强制] 仅定义一个变量时不允许使用解构。

解释：

在这种场景下，使用解构将降低代码可读性。

示例：

```
1. // good
2. let len = myString.length;
3.
4. // bad
5. let {length: len} = myString;
```

[强制] 如果不节省编写时产生的中间变量，解构表达式 `=` 号右边不允许是 `ObjectLiteral` 和 `ArrayLiteral`。

解释：

在这种场景下，使用解构将降低代码可读性，通常也并无收益。

示例：

```
1. // good
2. let {first: firstName, last: lastName} = person;
3. let one = 1;
4. let two = 2;
5.
6. // bad
7. let [one, two] = [1, 2];
```

[强制] 使用剩余运算符时，剩余运算符之前的所有元素必需具名。

解释：

剩余运算符之前的元素省略名称可能带来较大的程序阅读障碍。如果仅仅为了取数组后几项，请使用 `slice` 方法。

示例：

```
1. // good
2. let [one, two, ...anyOther] = myArray;
3. let other = myArray.slice(3);
4.
5. // bad
6. let [,,, ...other] = myArray;
```

3.3 模板字符串

[强制] 字符串内变量替换时，不要使用 2 次及以上的函数调用。

解释：

在变量替换符内有太多的函数调用等复杂语法会导致可读性下降。

示例：

```
1. // good
2. let fullName = getFullName(getFirstName(), getLastName());
3. let s = `Hello ${fullName}`;
4.
5. // bad
6. let s = `Hello ${getFullName(getFirstName(), getLastName())}`;
```

3.4 函数

[建议] 使用变量默认语法代替基于条件判断的默认值声明。

解释：

添加默认值有助于引擎的优化，在未来 strong mode 下也会有更好的效果。

示例：

```
1. // good
2. function foo(text = 'hello') {
3. }
4.
5. // bad
6. function foo(text) {
7.     text = text || 'hello';
```

```
8. }
```

[强制] 不要使用 `arguments` 对象，应使用 `...args` 代替。

解释：

在未来 `strong mode` 下 `arguments` 将被禁用。

示例：

```
1. // good
2. function foo(...args) {
3.     console.log(args.join(''));
4. }
5.
6. // bad
7. function foo() {
8.     console.log([].join.call(arguments));
9. }
```

3.5 箭头函数

[强制] 一个函数被设计为需要 `call` 和 `apply` 的时候，不能是箭头函数。

解释：

箭头函数会强制绑定当前环境下的 `this`。

3.6 对象

[建议] 定义对象时，如果所有键均指向同名变量，则所有键都使用缩写；如果有一个键无法指向同名变量，则所有键都不使用缩写。

解释：

目的在于保持所有键值对声明的一致性。

```
1. // good
2. let foo = {x, y, z};
3.
4. let foo2 = {
5.     x: 1,
6.     y: 2,
```



```
7.      z: z
8.  };
9.
10.
11.  // bad
12.  let foo = {
13.      x: x,
14.      y: y,
15.      z: z
16.  };
17.
18.  let foo2 = {
19.      x: 1,
20.      y: 2,
21.      z
22.  };
```

[强制] 定义方法时使用 `MethodDefinition` 语法，不使用 `PropertyName: FunctionExpression` 语法。

解释：

`MethodDefinition` 语法更清晰简洁。

示例：

```
1.  // good
2.  let foo = {
3.      bar(x, y) {
4.          return x + y;
5.      }
6.  };
7.
8.  // bad
9.  let foo = {
10.     bar: function (x, y) {
11.         return x + y;
12.     }
13.  };
```

[建议] 使用 `Object.keys` 或 `Object.entries` 进行对象遍历。

解释：

不建议使用 `for .. in` 进行对象的遍历，以避免遗漏 `hasOwnProperty` 产生的错误。

示例：

```
1. // good
2. for (let key of Object.keys(foo)) {
3.     let value = foo[key];
4. }
5.
6. // good
7. for (let [key, value] of Object.entries(foo)) {
8.     // ...
9. }
```

[建议] 定义对象的方法不应使用箭头函数。

解释：

箭头函数将 `this` 绑定到当前环境，在 `obj.method()` 调用时容易导致不期待的 `this`。
除非明确需要绑定 `this`，否则不应使用箭头函数。

示例：

```
1. // good
2. let foo = {
3.     bar(x, y) {
4.         return x + y;
5.     }
6. };
7.
8. // bad
9. let foo = {
10.     bar: (x, y) => x + y
11. };
```

[建议] 尽量使用计算属性键在一个完整的字面量中完整地定义一个对象，避免对象定义后直接增加对象属性。

解释：

在一个完整的字面量中声明所有的键值，而不需要将代码分散开来，有助于提升代码可读性。

示例：

```
1. // good
2. const MY_KEY = 'bar';
3. let foo = {
4.     [MY_KEY + 'Hash']: 123
5. };
6.
7. // bad
8. const MY_KEY = 'bar';
9. let foo = {};
10. foo[MY_KEY + 'Hash'] = 123;
```

3.7 类

[强制] 使用 `class` 关键字定义一个类。

解释：

直接使用 `class` 定义类更清晰。不要再使用 `function` 和 `prototype` 形式的定义。

```
1. // good
2. class TextNode {
3.     constructor(value, engine) {
4.         this.value = value;
5.         this.engine = engine;
6.     }
7.
8.     clone() {
9.         return this;
10.    }
11. }
12.
13. // bad
14. function TextNode(value, engine) {
15.     this.value = value;
16.     this.engine = engine;
17. }
18.
19. TextNode.prototype.clone = function () {
20.     return this;
21. };
```

[强制] 使用 `super` 访问父类成员，而非父类的 `prototype`。

解释：

使用 `super` 和 `super.foo` 可以快速访问父类成员，而不必硬编码父类模块而导致修改和维护的不便，同时更节省代码。

```

1.  // good
2.  class TextNode extends Node {
3.      constructor(value, engine) {
4.          super(value);
5.          this.engine = engine;
6.      }
7.
8.      setNodeValue(value) {
9.          super.setNodeValue(value);
10.         this.textContent = value;
11.     }
12. }
13.
14. // bad
15. class TextNode extends Node {
16.     constructor(value, engine) {
17.         Node.apply(this, arguments);
18.         this.engine = engine;
19.     }
20.
21.     setNodeValue(value) {
22.         Node.prototype.setNodeValue.call(this, value);
23.         this.textContent = value;
24.     }
25. }
```

3.8 模块

[强制] `export` 与内容定义放在一起。

解释：

何处声明要导出的东西，就在何处使用 `export` 关键字，不在声明后再统一导出。

示例：

```
1. // good
2. export function foo() {
3. }
4.
5. export const bar = 3;
6.
7.
8. // bad
9. function foo() {
10. }
11.
12. const bar = 3;
13.
14. export {foo};
15. export {bar};
```

[建议] 相互之间无关联的内容使用命名导出。

解释：

举个例子，工具对象中的各个方法，相互之间并没有强关联，通常外部会选择几个使用，则应该使用命名导出。

简而言之，当一个模块只扮演命名空间的作用时，使用命名导出。

[强制] 所有 `import` 语句写在模块开始处。

示例：

```
1. // good
2. import {bar} from './bar';
3.
4. function foo() {
5.     bar.work();
6. }
7.
8. // bad
9. function foo() {
10.     import {bar} from './bar';
11.     bar.work();
12. }
```

3.9 集合

[建议] 对数组进行连接操作时，使用数组展开语法。

解释：

用数组展开代替 `concat` 方法，数组展开对 `Iterable` 有更好的兼容性。

示例：

```
1. // good
2. let foo = [...foo, newValue];
3. let bar = [...bar, ...newValues];
4.
5. // bad
6. let foo = foo.concat(newValue);
7. let bar = bar.concat(newValues);
```

[建议] 不要使用数组展开进行数组的复制操作。

解释：

使用数组展开语法进行复制，代码可读性较差。推荐使用 `Array.from` 方法进行复制操作。

示例：

```
1. // good
2. let otherArr = Array.from(arr);
3.
4. // bad
5. let otherArr = [...arr];
```

[建议] 尽可能使用 `for .. of` 进行遍历。

解释：

使用 `for .. of` 可以更好地接受任何的 `Iterable` 对象，如 `Map#values` 生成的迭代器，使得方法的通用性更强。

以下情况除外：

1. 遍历确实成为了性能瓶颈，需要使用原生 `for` 循环提升性能。
2. 需要遍历过程中的索引。

[强制] 当键值有可能不是字符串时，必须使用 `Map`；当元素有可能不是字符串时，必须使用 `Set`。

解释：

使用普通 `Object`，对非字符串类型的 `key`，需要自己实现序列化。并且运行过程中的对象变化难以通知 `Object`。

[建议] 需要一个不可重复的集合时，应使用 `Set`。

解释：

不要使用 `{foo: true}` 这样的普通 `Object`。

示例：

```
1. // good
2. let members = new Set(['one', 'two', 'three']);
3.
4. // bad
5. let members = {
6.   one: true,
7.   two: true,
8.   three: true
9. };
```

[建议] 当需要遍历功能时，使用 `Map` 和 `Set`。

解释：

`Map` 和 `Set` 是可遍历对象，能够方便地使用 `for...of` 遍历。不要使用使用普通 `Object`。

示例：

```
1. // good
2. let membersAge = new Map([
3.   ['one', 10],
4.   ['two', 20],
5.   ['three', 30]
6. ]);
7. for (let [key, value] of map) {
8. }
9.
```

```
10. // bad
11. let membersAge = {
12.     one: 10,
13.     two: 20,
14.     three: 30
15. };
16. for (let key in membersAge) {
17.     if (membersAge.hasOwnProperty(key)) {
18.         let value = membersAge[key];
19.     }
20. }
```

[建议] 程序运行过程中有添加或移除元素的操作时，使用 `Map` 和 `Set`。

解释：

使用 `Map` 和 `Set`，程序的可理解性更好；普通 `Object` 的语义更倾向于表达固定的结构。

示例：

```
1. // good
2. let membersAge = new Map();
3. membersAge.set('one', 10);
4. membersAge.set('two', 20);
5. membersAge.set('three', 30);
6. membersAge.delete('one');
7.
8. // bad
9. let membersAge = {};
10. membersAge.one = 10;
11. membersAge.two = 20;
12. membersAge.three = 30;
13. delete membersAge['one'];
```

3.10 异步

[强制] 回调函数的嵌套不得超过3层。

解释：

深层次的回调函数的嵌套会让代码变得难以阅读。

示例：


```
1. // bad
2. getUser(userId, function (user) {
3.     validateUser(user, function (isValid) {
4.         if (isValid) {
5.             saveReport(report, user, function () {
6.                 notice('Saved!');
7.             });
8.         }
9.     });
10. });
```

[建议] 使用 `Promise` 代替 `callback`。

解释：

相比 `callback`，使用 `Promise` 能够使复杂异步过程的代码更清晰。

示例：

```
1. // good
2. let user;
3. getUser(userId)
4.     .then(function (userObj) {
5.         user = userObj;
6.         return validateUser(user);
7.     })
8.     .then(function (isValid) {
9.         if (isValid) {
10.            return saveReport(report, user);
11.        }
12.
13.        return Promise.reject('Invalid!');
14.    })
15.    .then(
16.        function () {
17.            notice('Saved!');
18.        },
19.        function (message) {
20.            notice(message);
21.        }
22.    );
```

[强制] 使用标准的 `Promise` API。

解释：

1. 不允许使用非标准的 `Promise` API，如 `jQuery` 的 `Deferred`、`Q.js` 的 `defer` 等。
2. 不允许使用非标准的 `Promise` 扩展 API，如 `bluebird` 的 `Promise.any` 等。

使用标准的 `Promise` API，当运行环境都支持时，可以把 `Promise Lib` 直接去掉。

[强制] 不允许直接扩展 `Promise` 对象的 `prototype`。

解释：

理由和 不允许修改和扩展任何原生对象和宿主对象的原型 是一样的。如果想要使用更方便，可以用 `utility` 函数的形式。

[强制] 不得为了编写的方便，将可以并行的IO过程串行化。

解释：

并行 IO 消耗时间约等于 IO 时间最大的那个过程，串行的话消耗时间将是所有过程的时间之和。

示例：

```
1. requestData().then(function (data) {
2.     renderTags(data.tags);
3.     renderArticles(data.articles);
4. });
5.
6. // good
7. async function requestData() {
8.     const [tags, articles] = await Promise.all([
9.         requestTags(),
10.        requestArticles()
11.    ]);
12.    return {tags, articles};
13. }
14.
15. // bad
16. async function requestData() {
17.     let tags = await requestTags();
18.     let articles = await requestArticles();
19. }
```

```
20.     return Promise.resolve({tags, articles});
21. }
```

[建议] 使用 `async/await` 代替 `generator` + `co`。

解释：

使用语言自身的能力可以使代码更清晰，也无需引入 `co` 库。

示例：

```
1.  addReport(report, userId).then(
2.    function () {
3.      notice('Saved!');
4.    },
5.    function (message) {
6.      notice(message);
7.    }
8.  );
9.
10. // good
11. async function addReport(report, userId) {
12.   let user = await getUser(userId);
13.   let isValid = await validateUser(user);
14.
15.   if (isValid) {
16.     let savePromise = saveReport(report, user);
17.     return savePromise();
18.   }
19.
20.   return Promise.reject('Invalid');
21. }
22.
23. // bad
24. function addReport(report, userId) {
25.   return co(function* () {
26.     let user = yield getUser(userId);
27.     let isValid = yield validateUser(user);
28.
29.     if (isValid) {
30.       let savePromise = saveReport(report, user);
31.       return savePromise();
32.     }

```

```
33.  
34.         return Promise.reject('Invalid');  
35.     });  
36. }
```

4 环境

4.1 运行环境

[建议] 持续跟进与关注运行环境对语言特性的支持程度。

解释：

[查看环境对语言特性的支持程度](#)

ES 标准的制定还在不断进行中，各种环境对语言特性的支持也日新月异。了解项目中用到了哪些 ESNext 的特性，了解项目的运行环境，并持续跟进这些特性在运行环境中的支持程度是很有必要的。这意味着：

1. 如果有任何一个运行环境（比如 chrome）支持了项目里用到的所有特性，你可以在开发时抛弃预编译。
2. 如果所有环境都支持了某一特性（比如 Promise），你可以抛弃相关的 shim，或无需在预编译时进行转换。
3. 如果所有环境都支持了项目里用到的所有特性，你可以完全抛弃预编译。

无论如何，在选择预编译工具时，你都需要清晰的知道你现阶段将在项目里使用哪些语言特性，然后了解预编译工具对语言特性的支持程度，做出选择。

[强制] 在运行环境中没有 `Promise` 时，将 `Promise` 的实现 `shim` 到 `global` 中。

解释：

当前运行环境下没有 `Promise` 时，可以引入 `shim` 的扩展。如果自己实现，需要实现在 `global` 下，并且与标准 API 保持一致。

这样，未来运行环境支持时，可以随时把 `Promise` 扩展直接扔掉，而应用代码无需任何修改。

4.2 预编译

[建议] 使用 `babel` 做为预编译工具时，建议使用 `5.x` 版本。

解释：

由于 `babel` 最新的 `6` 暂时还不稳定，建议暂时使用 `5.x`。不同的产品，对于浏览器支持的情况不同，使用 `babel` 的时候，需要设置的参数也有一些区别。下面在示例中给出一些建议的参数。

示例：

```
1. # 建议的参数
2. --loose all --modules amd --blacklist strict
3.
4. # 如果需要使用 es7.classProperties、es7.decorators 等一些特性，需要额外的 --stage 0 参数
5. --loose all --modules amd --blacklist strict --stage 0
```

[建议] 使用 `babel` 做为预编译工具时，通过 `external-helpers` 减少生成文件的大小。

解释：

当 `babel` 在转换代码的过程中发现需要一些特性时，会在该文件头部生成对应的 `helper` 代码。默认情况下，对于每一个经由 `babel` 处理的文件，均会在文件头部生成对应需要的辅助函数，多份文件辅助函数存在重复，占用了不必要的代码体积。

因此推荐打开 `externalHelpers: true` 选项，使 `babel` 在转换后内容中不写入 `helper` 相关的代码，而是使用一个外部的 `.js` 统一提供所有的 `helper`。对于 `external-helpers` 的使用，可以有两种方式：

1. 默认方式：需要通过 `<script>` 自行引入 `babel-polyfill.js` 和 `babel-external-helpers.js`。
2. 定制方式：自己实现 `babel-runtime`。

示例：

```
1. # 默认方式
2. --loose all --modules amd --external-helpers
3. # `babelHelpers` 的代码可以通过执行 `babel-external-helpers -t var` 得到所有相关API的实现
4.
5. # 定制方式
6. --loose all --modules amd --optional runtime
```

[建议] 使用 `TypeScript` 做为预编译工具时，建议使用 `1.6+` 版本。

解释：

`TypeScript` 1.6 之后，基本摒弃了之前的与 ESNext 相冲突的地方。目前 `TypeScript` 的思路就是遵循标准，将 stage 已经足够成熟的功能纳入，并提供静态类型和类型检查，所以其在 stage 0/1 的支持上不如 `babel`。另外，`TypeScript` 不能指定关闭某个 transform，但其编译速度比 `babel` 更高。

`TypeScript` 的常用参数在下面给出了示例。

示例：

```
1. --module amd --target ES3
2. --module commonjs --target ES6
```

[建议] 使用 `TypeScript` 做为预编译工具时，不使用 `tsc` 命令。

解释：

`TypeScript` 提供的 `tsc` 命令只支持后缀名 `.ts`、`.tsx`、`.d.ts` 的文件编译，对于 JavaScript 来说，保持后缀名为 `.js` 是原则，本文档的 [文件](#) 章节也有所要求。

如果要使用 `TypeScript` 做为预编译工具，可基于其 [Compiler API](#) 开发自己的预编译工具。如果你是 FIS 用户，可以使用 [FIS TypeScript 插件](#)。

[建议] 生成的代码在浏览器环境运行时，应生成 AMD 模块化代码。

解释：

AMD 在浏览器环境应用较为成熟。

[建议] 浏览器端项目中如果 ESNext 代码和 ES3/5 代码混合，不要使用 `TypeScript` 做为预编译工具。

解释：

`TypeScript` 产生的 module 代码使用 `exports.default` 导出默认的 export，但是没有直接为 `module.exports` 赋值，导致在另外一个普通文件中使用 `require('moduleName')` 是拿不到东西的。

需要使用 `TypeScript` 的话，建议整个项目所有文件都是 ESNext module 的，采用混合的 module 容易出现不可预期的结果。

[建议] AMD/CommonJS 模块依赖 ESNext 模块时，AMD/CommonJS 模块对 default export 的 require 需要改造。

解释：

ESNext 模块经过编译后，named export 会挂载在 exports 对象上，default export 也会挂载在 exports 对象上名称为 default 的属性。同时 exports 对象会包含一个值为 true 的 __esModule 属性。那么问题来了，当 AMD/CommonJS 模块依赖了 ESNext 模块时，require 期望拿到的是 exports.default，但你实际上拿到的是 exports。

所以，老的 AMD/CommonJS 模块依赖了 default export 的 ESNext 模块时，对 default export 的 require 需要改造成 `require('name').default`。

另外，如果是 ESNext 模块之间的互相依赖，transpiler 会通过加入中间对象和引入 interop 方法，所以不会产生这个问题。

HTML编码规范

HTML编码规范

1 前言

HTML 作为描述网页结构的超文本标记语言，在百度一直有着广泛的应用。本文档的目标是使 HTML 代码风格保持一致，容易被理解和被维护。

2 代码风格

2.1 缩进与换行

[强制] 使用 `4` 个空格做为一个缩进层级，不允许使用 `2` 个空格 或 `tab` 字符。

解释：

对于非 HTML 标签之间的缩进，比如 `script` 或 `style` 标签内容缩进，与 `script` 或 `style` 标签的缩进同级。

示例：

```
1. <style>
2. /* 样式内容的第一级缩进与所属的 style 标签对齐 */
3. ul {
4.     padding: 0;
5. }
6. </style>
7. <ul>
8.     <li>first</li>
9.     <li>second</li>
10. </ul>
11. <script>
12. // 脚本代码的第一级缩进与所属的 script 标签对齐
13. require(['app'], function (app) {
14.     app.init();
15. });
16. </script>
```


[建议] 每行不得超过 120 个字符。

解释：

过长的代码不容易阅读与维护。但是考虑到 HTML 的特殊性，不做硬性要求。

2.2 命名

[强制] `class` 必须单词全字母小写，单词间以 `-` 分隔。

[强制] `class` 必须代表相应模块或部件的内容或功能，不得以样式信息进行命名。

示例：

```
1. <!-- good -->
2. <div class="sidebar"></div>
3.
4. <!-- bad -->
5. <div class="left"></div>
```

[强制] 元素 `id` 必须保证页面唯一。

解释：

同一个页面中，不同的元素包含相同的 `id`，不符合 `id` 的属性含义。并且使用 `document.getElementById` 时可能导致难以追查的问题。

[建议] `id` 建议单词全字母小写，单词间以 `-` 分隔。同项目必须保持风格一致。

[建议] `id`、`class` 命名，在避免冲突并描述清楚的前提下尽可能短。

示例：

```
1. <!-- good -->
2. <div id="nav"></div>
3. <!-- bad -->
4. <div id="navigation"></div>
5.
6. <!-- good -->
7. <p class="comment"></p>
8. <!-- bad -->
```

```
9.  <p class="com"></p>
10.
11. <!-- good -->
12. <span class="author"></span>
13. <!-- bad -->
14. <span class="red"></span>
```

[强制] 禁止为了 `hook 脚本`，创建无样式信息的 `class`。

解释：

不允许 `class` 只用于让 JavaScript 选择某些元素，`class` 应该具有明确的语义和样式。否则容易导致 CSS class 泛滥。

使用 `id`、属性选择作为 hook 是更好的方式。

[强制] 同一页面，应避免使用相同的 `name` 与 `id`。

解释：

IE 浏览器会混淆元素的 `id` 和 `name` 属性，`document.getElementById` 可能获得不期望的元素。所以在对元素的 `id` 与 `name` 属性的命名需要非常小心。

一个比较好的实践是，为 `id` 和 `name` 使用不同的命名法。

示例：

```
1. <input name="foo">
2. <div id="foo"></div>
3. <script>
4. // IE6 将显示 INPUT
5. alert(document.getElementById('foo').tagName);
6. </script>
7. `
```

2.3 标签

[强制] 标签名必须使用小写字母。

示例：

```
1. <!-- good -->
2. <p>Hello StyleGuide!</p>
```

```
3.  
4. <!-- bad -->  
5. <P>Hello StyleGuide!</P>
```

[强制] 对于无需自闭合的标签，不允许自闭合。

解释：

常见无需自闭合标签有 `input`、`br`、`img`、`hr` 等。

示例：

```
1. <!-- good -->  
2. <input type="text" name="title">  
3.  
4. <!-- bad -->  
5. <input type="text" name="title" />
```

[强制] 对 `HTML5` 中规定允许省略的闭合标签，不允许省略闭合标签。

解释：

对代码体积要求非常严苛的场景，可以例外。比如：第三方页面使用的投放系统。

示例：

```
1. <!-- good -->  
2. <ul>  
3.     <li>first</li>  
4.     <li>second</li>  
5. </ul>  
6.  
7. <!-- bad -->  
8. <ul>  
9.     <li>first  
10.    <li>second  
11. </ul>
```

[强制] 标签使用必须符合标签嵌套规则。

解释：

比如 `div` 不得置于 `p` 中，`tbody` 必须置于 `table` 中。

详细的标签嵌套规则参见[HTML DTD](#)中的 **Elements** 定义部分。

[建议] HTML 标签的使用应该遵循标签的语义。

解释：

下面是常见标签语义

- p - 段落
- h1, h2, h3, h4, h5, h6 - 层级标题
- strong, em - 强调
- ins - 插入
- del - 删除
- abbr - 缩写
- code - 代码标识
- cite - 引述来源作品的标题
- q - 引用
- blockquote - 一段或长篇引用
- ul - 无序列表
- ol - 有序列表
- dl, dt, dd - 定义列表

示例：

```
1. <!-- good -->
2. <p>Esprima serves as an important <strong>building block</strong> for some
   JavaScript language tools.</p>
3.
4. <!-- bad -->
5. <div>Esprima serves as an important <span class="strong">building block</span>
   for some JavaScript language tools.</div>
```

[建议] 在 CSS 可以实现相同需求的情况下不得使用表格进行布局。

解释：

在兼容性允许的情况下应尽量保持语义正确性。对网格对齐和拉伸性有严格要求的场景允许例外，如多列复杂表单。

[建议] 标签的使用应尽量简洁，减少不必要的标签。

示例：

```
1. <!-- good -->
2. 
3.
4. <!-- bad -->
5. <span class="avatar">
6.     
7. </span>
```

2.4 属性

[强制] 属性名必须使用小写字母。

示例：

```
1. <!-- good -->
2. <table cellpadding="0">...</table>
3.
4. <!-- bad -->
5. <table cellSpacing="0">...</table>
```

[强制] 属性值必须用双引号包围。

解释：

不允许使用单引号，不允许不使用引号。

示例：

```
1. <!-- good -->
2. <script src="es1.js"></script>
3.
4. <!-- bad -->
5. <script src='es1.js'></script>
6. <script src=es1.js></script>
```

[建议] 布尔类型的属性，建议不添加属性值。

示例：

```
1. <input type="text" disabled>
2. <input type="checkbox" value="1" checked>
```

[建议] 自定义属性建议以 `xxx-` 为前缀，推荐使用 `data-`。

解释：

使用前缀有助于区分自定义属性和标准定义的属性。

示例：

```
1. <ol data-ui-type="Select"></ol>
```

3 通用

3.1 DOCTYPE

[强制] 使用 `HTML5` 的 `doctype` 来启用标准模式，建议使用大写的 `DOCTYPE`。

示例：

```
1. <!DOCTYPE html>
```

[建议] 启用 IE Edge 模式。

示例：

```
1. <meta http-equiv="X-UA-Compatible" content="IE=Edge">
```

[建议] 在 `html` 标签上设置正确的 `lang` 属性。

解释：

有助于提高页面的可访问性，如：让语音合成工具确定其所应该采用的发音，令翻译工具确定其翻译语言等。

示例：

```
1. <html lang="zh-CN">
```

3.2 编码

[强制] 页面必须使用精简形式，明确指定字符编码。指定字符编码的 `meta` 必

须是 `head` 的第一个直接子元素。

解释：

见 [HTML5 Charset能用吗](#) 一文。

示例：

```
1. <html>
2.     <head>
3.         <meta charset="UTF-8">
4.         .....
5.     </head>
6.     <body>
7.         .....
8.     </body>
9. </html>
```

[建议] `HTML` 文件使用无 `BOM` 的 `UTF-8` 编码。

解释：

`UTF-8` 编码具有更广泛的适应性。`BOM` 在使用程序或工具处理文件时可能造成不必要的干扰。

3.3 CSS 和 JavaScript 引入

[强制] 引入 `CSS` 时必须指明 `rel="stylesheet"`。

示例：

```
1. <link rel="stylesheet" href="page.css">
```

[建议] 引入 `CSS` 和 `JavaScript` 时无须指明 `type` 属性。

解释：

`text/css` 和 `text/javascript` 是 `type` 的默认值。

[建议] 展现定义放置于外部 `CSS` 中，行为定义放置于外部 `JavaScript` 中。

解释：

结构-样式-行为的代码分离，对于提高代码的可阅读性和维护性都有好处。

[建议] 在 `head` 中引入页面需要的所有 `CSS` 资源。

解释：

在页面渲染的过程中，新的CSS可能导致元素的样式重新计算和绘制，页面闪烁。

[建议] `JavaScript` 应当放在页面末尾，或采用异步加载。

解释：

将 `script` 放在页面中间将阻断页面的渲染。出于性能方面的考虑，如非必要，请遵守此条建议。

示例：

```
1. <body>
2.     <!-- a lot of elements -->
3.     <script src="init-behavior.js"></script>
4. </body>
```

[建议] 移动环境或只针对现代浏览器设计的 Web 应用，如果引用外部资源的 `URL` 协议部分与页面相同，建议省略协议前缀。

解释：

使用 `protocol-relative URL` 引入 CSS，在 `IE7/8` 下，会发两次请求。是否使用 `protocol-relative URL` 应充分考虑页面针对的环境。

示例：

```
1. <script src="//s1.bdstatic.com/cache/static/jquery-1.10.2.min_f2fb5194.js">
   </script>
```

4 head

4.1 title

[强制] 页面必须包含 `title` 标签声明标题。

[强制] `title` 必须作为 `head` 的直接子元素，并紧随 `charset` 声明之后。

解释：

`title` 中如果包含 ASCII 之外的字符，浏览器需要知道字符编码类型才能进行解码，否则可能导

致乱码。

示例：

```
1. <head>
2.     <meta charset="UTF-8">
3.     <title>页面标题</title>
4. </head>
```

4.2 favicon

[强制] 保证 `favicon` 可访问。

解释：

在未指定 `favicon` 时，大多数浏览器会请求 Web Server 根目录下的 `favicon.ico`。为了保证 `favicon` 可访问，避免 404，必须遵循以下两种方法之一：

1. 在 Web Server 根目录放置 `favicon.ico` 文件。
2. 使用 `link` 指定 `favicon`。

示例：

```
1. <link rel="shortcut icon" href="path/to/favicon.ico">
```

4.3 viewport

[建议] 若页面欲对移动设备友好，需指定页面的 `viewport`。

解释：

`viewport meta tag` 可以设置可视区域的宽度和初始缩放大小，避免在移动设备上出现页面展示不正常。

比如，在页面宽度小于 `980px` 时，若需 iOS 设备友好，应当设置 `viewport` 的 `width` 值来适应你的页面宽度。同时因为不同移动设备分辨率不同，在设置时，应当使用 `device-width` 和 `device-height` 变量。

另外，为了使 `viewport` 正常工作，在页面内容样式布局设计上也要做相应调整，如避免绝对定位等。关于 `viewport` 的更多介绍，可以参见 [Safari Web Content Guide](#)的介绍

5 图片

[强制] 禁止 `img` 的 `src` 取值为空。延迟加载的图片也要增加默认的 `src`。

解释：

`src` 取值为空，会导致部分浏览器重新加载一次当前页面，参考：<https://developer.yahoo.com/performance/rules.html#emptysrc>

[建议] 避免为 `img` 添加不必要的 `title` 属性。

解释：

多余的 `title` 影响看图体验，并且增加了页面尺寸。

[建议] 为重要图片添加 `alt` 属性。

解释：

可以提高图片加载失败时的用户体验。

[建议] 添加 `width` 和 `height` 属性，以避免页面抖动。

[建议] 有下载需求的图片采用 `img` 标签实现，无下载需求的图片采用 CSS 背景图实现。

解释：

1. 产品 logo、用户头像、用户产生的图片等有潜在下载需求的图片，以 `img` 形式实现，能方便用户下载。
2. 无下载需求的图片，比如：icon、背景、代码使用的图片等，尽可能采用 CSS 背景图实现。

6 表单

6.1 控件标题

[强制] 有文本标题的控件必须使用 `label` 标签将其与其标题相关联。

解释：

有两种方式：

1. 将控件置于 `label` 内。
2. `label` 的 `for` 属性指向控件的 `id`。

推荐使用第一种，减少不必要的 `id`。如果 DOM 结构不允许直接嵌套，则应使用第二种。

示例：

```
1. <label><input type="checkbox" name="confirm" value="on"> 我已确认上述条款</label>
2.
3. <label for="username">用户名 :</label> <input type="text" name="username"
   id="username">
```

6.2 按钮

[强制] 使用 `button` 元素时必须指明 `type` 属性值。

解释：

`button` 元素的默认 `type` 为 `submit`，如果被置于 `form` 元素中，点击后将导致表单提交。为显示区分其作用方便理解，必须给出 `type` 属性。

示例：

```
1. <button type="submit">提交</button>
2. <button type="button">取消</button>
```

[建议] 尽量不要使用按钮类元素的 `name` 属性。

解释：

由于浏览器兼容性问题，使用按钮的 `name` 属性会带来许多难以发现的问题。具体情况可参考[此文](#)。

6.3 可访问性（A11Y）

[建议] 负责主要功能的按钮在 DOM 中的顺序应靠前。

解释：

负责主要功能的按钮应相对靠前，以提高可访问性。如果在 CSS 中指定了 `float: right` 则可能导致视觉上主按钮在前，而 DOM 中主按钮靠后的情况。

示例：

```
1. <!-- good -->
2. <style>
```

```
3.  .buttons .button-group {
4.      float: right;
5.  }
6.  </style>
7.
8.  <div class="buttons">
9.      <div class="button-group">
10.         <button type="submit">提交</button>
11.         <button type="button">取消</button>
12.     </div>
13. </div>
14.
15. <!-- bad -->
16. <style>
17. .buttons button {
18.     float: right;
19. }
20. </style>
21.
22. <div class="buttons">
23.     <button type="button">取消</button>
24.     <button type="submit">提交</button>
25. </div>
```

[建议] 当使用 JavaScript 进行表单提交时，如果条件允许，应使原生提交功能正常工作。

解释：

当浏览器 JS 运行错误或关闭 JS 时，提交功能将无法工作。如果正确指定了 `form` 元素的 `action` 属性和表单控件的 `name` 属性时，提交仍可继续进行。

示例：

```
1. <form action="/login" method="post">
2.     <p><input name="username" type="text" placeholder="用户名"></p>
3.     <p><input name="password" type="password" placeholder="密码"></p>
4. </form>
```

[建议] 在针对移动设备开发的页面时，根据内容类型指定输入框的 `type` 属性。

解释：

根据内容类型指定输入框类型，能获得友好的输入体验。

示例：

```
1. <input type="date">
```

7 多媒体

[建议] 当在现代浏览器中使用 `audio` 以及 `video` 标签来播放音频、视频时，应当注意格式。

解释：

音频应尽可能覆盖到如下格式：

- MP3
- WAV
- Ogg

视频应尽可能覆盖到如下格式：

- MP4
- WebM
- Ogg

[建议] 在支持 `HTML5` 的浏览器中优先使用 `audio` 和 `video` 标签来定义音视频元素。

[建议] 使用退化到插件的方式来对多浏览器进行支持。

示例：

```
1. <audio controls>
2.   <source src="audio.mp3" type="audio/mpeg">
3.   <source src="audio.ogg" type="audio/ogg">
4.   <object width="100" height="50" data="audio.mp3">
5.     <embed width="100" height="50" src="audio.swf">
6.   </object>
7. </audio>
8.
9. <video width="100" height="50" controls>
```

```
10.     <source src="video.mp4" type="video/mp4">
11.     <source src="video.ogv" type="video/ogg">
12.     <object width="100" height="50" data="video.mp4">
13.         <embed width="100" height="50" src="video.swf">
14.     </object>
15. </video>
```

[建议] 只在必要的时候开启音视频的自动播放。

[建议] 在 `object` 标签内部提供指示浏览器不支持该标签的说明。

示例：

```
1. <object width="100" height="50" data="something.swf">DO NOT SUPPORT THIS
   TAG</object>
```

8 模板中的 HTML

[建议] 模板代码的缩进优先保证 HTML 代码的缩进规则。

示例：

```
1. <!-- good -->
2. {if $display == true}
3. <div>
4.     <ul>
5.         {foreach $item_list as $item}
6.             <li>{$item.name}</li>
7.         {/foreach}
8.     </ul>
9. </div>
10. {/if}
11.
12. <!-- bad -->
13. {if $display == true}
14.     <div>
15.         <ul>
16.             {foreach $item_list as $item}
17.                 <li>{$item.name}</li>
18.             {/foreach}
19.         </ul>
```

```

20.     </div>
21.  {/if}

```

[建议] 模板代码应以保证 HTML 单个标签语法的正确性为基本原则。

示例：

```

1.  <!-- good -->
2.  <li class="{if $item.type_id == $current_type}focus{/if}">{ $item.type_name }
    </li>
3.
4.  <!-- bad -->
5.  <li {if $item.type_id == $current_type} class="focus"{/if}>{ $item.type_name }
    </li>

```

[建议] 在循环处理模板数据构造表格时，若要求每行输出固定的个数，建议先将数据分组，之后再循环输出。

示例：

```

1.  <!-- good -->
2.  <table>
3.      {foreach $item_list as $item_group}
4.      <tr>
5.          {foreach $item_group as $item}
6.          <td>{ $item.name }</td>
7.          {/foreach}
8.      </tr>
9.      {/foreach}
10. </table>
11.
12. <!-- bad -->
13. <table>
14. <tr>
15.     {foreach $item_list as $item}
16.     <td>{ $item.name }</td>
17.     {if $item@iteration is div by 5}
18.     </tr>
19.     <tr>
20.         {/if}
21.     {/foreach}
22. </tr>

```

```
23. </table>
```


CSS编码规范

CSS编码规范

1 前言

CSS 作为网页样式的描述语言，在百度一直有着广泛的应用。本文档的目标是使 CSS 代码风格保持一致，容易被理解和被维护。

虽然本文档是针对 CSS 设计的，但是在使用各种 CSS 的预编译器(如 less、sass、stylus 等)时，适用的部分也应尽量遵循本文档的约定。

2 代码风格

2.1 文件

[建议] CSS 文件使用无 BOM 的 UTF-8 编码。

解释：

UTF-8 编码具有更广泛的适应性。BOM 在使用程序或工具处理文件时可能造成不必要的干扰。

2.2 缩进

[强制] 使用 4 个空格做为一个缩进层级，不允许使用 2 个空格 或 tab 字符。

示例：

```
1. .selector {  
2.     margin: 0;  
3.     padding: 0;  
4. }
```

2.3 空格

[强制] 选择器 与 { 之间必须包含空格。

示例：

```
1. .selector {
2. }
```

[强制] **属性名** 与之后的 **:** 之间不允许包含空格, **:** 与 **属性值** 之间必须包含空格。

示例:

```
1. margin: 0;
```

[强制] **列表型属性值** 书写在单行时, **,** 后必须跟一个空格。

示例:

```
1. font-family: Arial, sans-serif;
```

2.4 行长度

[强制] 每行不得超过 **120** 个字符, 除非单行不可分割。

解释:

常见不可分割的场景为URL超长。

[建议] 对于超长的样式, 在样式值的 **空格** 处或 **,** 后换行, 建议按逻辑分组。

示例:

```
1. /* 不同属性值按逻辑分组 */
2. background:
3.     transparent url(aVeryVeryVeryLongUrlIsPlacedHere)
4.     no-repeat 0 0;
5.
6. /* 可重复多次的属性, 每次重复一行 */
7. background-image:
8.     url(aVeryVeryVeryLongUrlIsPlacedHere)
9.     url(anotherVeryVeryVeryLongUrlIsPlacedHere);
10.
11. /* 类似函数的属性值可以根据函数调用的缩进进行 */
12. background-image: -webkit-gradient(
13.     linear,
```



```
14.     left bottom,  
15.     left top,  
16.     color-stop(0.04, rgb(88,94,124)),  
17.     color-stop(0.52, rgb(115,123,162))  
18. );
```

2.5 选择器

[强制] 当一个 rule 包含多个 selector 时，每个选择器声明必须独占一行。

示例：

```
1.  /* good */  
2.  .post,  
3.  .page,  
4.  .comment {  
5.      line-height: 1.5;  
6.  }  
7.  
8.  /* bad */  
9.  .post, .page, .comment {  
10.      line-height: 1.5;  
11.  }
```

[强制] 、、 选择器的两边各保留一个空格。

示例：

```
1.  /* good */  
2.  main > nav {  
3.      padding: 10px;  
4.  }  
5.  
6.  label + input {  
7.      margin-left: 5px;  
8.  }  
9.  
10. input:checked ~ button {  
11.     background-color: #69C;  
12. }  
13.
```

```
14. /* bad */
15. main>nav {
16.     padding: 10px;
17. }
18.
19. label+input {
20.     margin-left: 5px;
21. }
22.
23. input:checked~button {
24.     background-color: #69C;
25. }
```

[强制] 属性选择器中的值必须用双引号包围。

解释：

不允许使用单引号，不允许不使用引号。

示例：

```
1. /* good */
2. article[character="juliet"] {
3.     voice-family: "Vivien Leigh", victoria, female;
4. }
5.
6. /* bad */
7. article[character='juliet'] {
8.     voice-family: "Vivien Leigh", victoria, female;
9. }
```

2.6 属性

[强制] 属性定义必须另起一行。

示例：

```
1. /* good */
2. .selector {
3.     margin: 0;
4.     padding: 0;
5. }
```

```
6.  
7.  /* bad */  
8.  .selector { margin: 0; padding: 0; }
```

[强制] 属性定义后必须以分号结尾。

示例：

```
1.  /* good */  
2.  .selector {  
3.      margin: 0;  
4.  }  
5.  
6.  /* bad */  
7.  .selector {  
8.      margin: 0  
9.  }
```

3 通用

3.1 选择器

[强制] 如无必要，不得为 `id`、`class` 选择器添加类型选择器进行限定。

解释：

在性能和维护性上，都有一定的影响。

示例：

```
1.  /* good */  
2.  #error,  
3.  .danger-message {  
4.      font-color: #c00;  
5.  }  
6.  
7.  /* bad */  
8.  dialog#error,  
9.  p.danger-message {  
10.     font-color: #c00;  
11. }
```

[建议] 选择器的嵌套层级应不大于 **3** 级，位置靠后的限定条件应尽可能精确。

示例：

```
1. /* good */
2. #username input {}
3. .comment .avatar {}
4.
5. /* bad */
6. .page .header .login #username input {}
7. .comment div * {}
```

3.2 属性缩写

[建议] 在可以使用缩写的情况下，尽量使用属性缩写。

示例：

```
1. /* good */
2. .post {
3.     font: 12px/1.5 arial, sans-serif;
4. }
5.
6. /* bad */
7. .post {
8.     font-family: arial, sans-serif;
9.     font-size: 12px;
10.    line-height: 1.5;
11. }
```

[建议] 使用 **border** / **margin** / **padding** 等缩写时，应注意隐含值对实际数值的影响，确实需要设置多个方向的值时才使用缩写。

解释：

border / **margin** / **padding** 等缩写会同时设置多个属性的值，容易覆盖不需要覆盖的设定。如某些方向需要继承其他声明的值，则应该分开设置。

示例：

```
1. /* centering <article class="page"> horizontally and highlight featured ones */
```

```

2.  article {
3.     margin: 5px;
4.     border: 1px solid #999;
5. }
6.
7. /* good */
8. .page {
9.     margin-right: auto;
10.    margin-left: auto;
11. }
12.
13. .featured {
14.     border-color: #69c;
15. }
16.
17. /* bad */
18. .page {
19.     margin: 5px auto; /* introducing redundancy */
20. }
21.
22. .featured {
23.     border: 1px solid #69c; /* introducing redundancy */
24. }

```

3.3 属性书写顺序

[建议] 同一 rule set 下的属性在书写时，应按功能进行分组，并以 **Formatting Model**（布局方式、位置） > **Box Model**（尺寸） > **Typographic**（文本相关） > **Visual**（视觉效果） 的顺序书写，以提高代码的可读性。

解释：

- Formatting Model 相关属性包括：position / top / right / bottom / left / float / display / overflow 等
- Box Model 相关属性包括：border / margin / padding / width / height 等
- Typographic 相关属性包括：font / line-height / text-align / word-wrap 等
- Visual 相关属性包括：background / color / transition / list-style 等

另外，如果包含 content 属性，应放在最前面。

示例：

```
1. .sidebar {
2.     /* formatting model: positioning schemes / offsets / z-indexes / display /
   ... */
3.     position: absolute;
4.     top: 50px;
5.     left: 0;
6.     overflow-x: hidden;
7.
8.     /* box model: sizes / margins / paddings / borders / ... */
9.     width: 200px;
10.    padding: 5px;
11.    border: 1px solid #ddd;
12.
13.    /* typographic: font / aligns / text styles / ... */
14.    font-size: 14px;
15.    line-height: 20px;
16.
17.    /* visual: colors / shadows / gradients / ... */
18.    background: #f5f5f5;
19.    color: #333;
20.    -webkit-transition: color 1s;
21.    -moz-transition: color 1s;
22.    transition: color 1s;
23. }
```

3.4 清除浮动

[建议] 当元素需要撑起高度以包含内部的浮动元素时，通过对伪类设置或触发 **BFC** 的方式进行 **clearfix**。尽量不使用增加空标签的方式。

clear

解释：

触发 BFC 的方式很多，常见的有：

- float 非 none
- position 非 static
- overflow 非 visible

如希望使用更小副作用的清除浮动方法，参见 [A new micro clearfix hack](#) 一文。

另需注意，对已经触发 BFC 的元素不需要再进行 clearfix。

3.5 !important

[建议] 尽量不使用 `!important` 声明。

[建议] 当需要强制指定样式且不允许任何场景覆盖时，通过标签内联和 `!important` 定义样式。

解释：

必须注意的是，仅在设计上 `确实不允许任何其它场景覆盖样式` 时，才使用内联的 `!important` 样式。通常在第三方环境的应用中使用这种方案。下面的 `z-index` 章节是其中一个特殊场景的典型样例。

3.6 z-index

[建议] 将 `z-index` 进行分层，对文档流外绝对定位元素的视觉层级关系进行管理。

解释：

同层的多个元素，如多个由用户输入触发的 Dialog，在该层级内使用相同的 `z-index` 或递增 `z-index`。

建议每层包含100个 `z-index` 来容纳足够的元素，如果每层元素较多，可以调整这个数值。

[建议] 在可控环境下，期望显示在最上层的元素，`z-index` 指定为 `999999`。

解释：

可控环境分成两种，一种是自身产品线环境；还有一种是可能会被其他产品线引用，但是不会被外部第三方的产品引用。

不建议取值为 `2147483647`。以便于自身产品线被其他产品线引用时，当遇到层级覆盖冲突的情况，留出向上调整的空间。

[建议] 在第三方环境下，期望显示在最上层的元素，通过标签内联和 `!important`，将 `z-index` 指定为 `2147483647`。

解释：

第三方环境对于开发者来说完全不可控。在第三方环境下的元素，为了保证元素不被其页面其他样式定义覆盖，需要采用此做法。

4 值与单位

4.1 文本

[强制] 文本内容必须用双引号包围。

解释：

文本类型的内容可能在选择器、属性值等内容中。

示例：

```
1.  /* good */
2.  html[lang|"zh"] q:before {
3.      font-family: "Microsoft YaHei", sans-serif;
4.      content: "";
5.  }
6.
7.  html[lang|"zh"] q:after {
8.      font-family: "Microsoft YaHei", sans-serif;
9.      content: "";
10. }
11.
12. /* bad */
13. html[lang|=zh] q:before {
14.     font-family: 'Microsoft YaHei', sans-serif;
15.     content: '';
16. }
17.
18. html[lang|=zh] q:after {
19.     font-family: "Microsoft YaHei", sans-serif;
20.     content: "";
21. }
```

4.2 数值

[强制] 当数值为 0 - 1 之间的小数时，省略整数部分的 0。

示例：

```
1.  /* good */
2.  panel {
3.      opacity: .8;
4.  }
```

```
5.  
6.  /* bad */  
7.  panel {  
8.      opacity: 0.8;  
9.  }
```

4.3 url()

[强制] `url()` 函数中的路径不加引号。

示例：

```
1.  body {  
2.      background: url(bg.png);  
3.  }
```

[建议] `url()` 函数中的绝对路径可省去协议名。

示例：

```
1.  body {  
2.      background: url(//baidu.com/img/bg.png) no-repeat 0 0;  
3.  }
```

4.4 长度

[强制] 长度为 `0` 时须省略单位。（也只有长度单位可省）

示例：

```
1.  /* good */  
2.  body {  
3.      padding: 0 5px;  
4.  }  
5.  
6.  /* bad */  
7.  body {  
8.      padding: 0px 5px;  
9.  }
```

4.5 颜色

[强制] RGB颜色值必须使用十六进制记号形式 `#rrggbb`。不允许使用 `rgb()`。

解释：

带有alpha的颜色信息可以使用 `rgba()`。使用 `rgba()` 时每个逗号后必须保留一个空格。

示例：

```
1.  /* good */
2.  .success {
3.      box-shadow: 0 0 2px rgba(0, 128, 0, .3);
4.      border-color: #008000;
5.  }
6.
7.  /* bad */
8.  .success {
9.      box-shadow: 0 0 2px rgba(0,128,0,.3);
10.     border-color: rgb(0, 128, 0);
11. }
```

[强制] 颜色值可以缩写时，必须使用缩写形式。

示例：

```
1.  /* good */
2.  .success {
3.      background-color: #aca;
4.  }
5.
6.  /* bad */
7.  .success {
8.      background-color: #aaccaa;
9.  }
```

[强制] 颜色值不允许使用命名色值。

示例：

```
1.  /* good */
2.  .success {
3.      color: #90ee90;
4.  }
5.
```

```
6.  /* bad */
7.  .success {
8.      color: lightgreen;
9.  }
```

[建议] 颜色值中的英文字符采用小写。如不用小写也需要保证同一项目内保持大小写一致。

示例：

```
1.  /* good */
2.  .success {
3.      background-color: #aca;
4.      color: #90ee90;
5.  }
6.
7.  /* good */
8.  .success {
9.      background-color: #ACA;
10.     color: #90EE90;
11. }
12.
13. /* bad */
14. .success {
15.     background-color: #ACA;
16.     color: #90ee90;
17. }
```

4.6 2D 位置

[强制] 必须同时给出水平和垂直方向的位置。

解释：

2D 位置初始值为 `0% 0%`，但在只有一个方向的值时，另一个方向的值会被解析为 `center`。为避免理解上的困扰，应同时给出两个方向的值。[background-position属性值的定义](#)

示例：

```
1.  /* good */
2.  body {
3.      background-position: center top; /* 50% 0% */
```

```
4.  }
5.
6.  /* bad */
7.  body {
8.      background-position: top; /* 50% 0% */
9.  }
```

5 文本编排

5.1 字体族

[强制] `font-family` 属性中的字体族名称应使用字体的英文 `Family Name`，其中如有空格，须放置在引号中。

解释：

所谓英文 Family Name，为字体文件的一个元数据，常见名称如下：

字体	操作系统	Family Name
宋体（中易宋体）	Windows	SimSun
黑体（中易黑体）	Windows	SimHei
微软雅黑	Windows	Microsoft YaHei
微软正黑	Windows	Microsoft JhengHei
华文黑体	Mac/iOS	STHeiti
冬青黑体	Mac/iOS	Hiragino Sans GB
文泉驿正黑	Linux	WenQuanYi Zen Hei
文泉驿微米黑	Linux	WenQuanYi Micro Hei

示例：

```
1. h1 {
2.     font-family: "Microsoft YaHei";
3. }
```

[强制] `font-family` 按「西文字体在前、中文字体在后」、「效果佳（质量高/更能满足需求）的字体在前、效果一般的字体在后」的顺序编写，最后必须指定一个通用字体族(`serif` / `sans-serif`)。

解释：

更详细说明可参考[本文](#)。

示例：

```
1. /* Display according to platform */
2. .article {
3.     font-family: Arial, sans-serif;
4. }
5.
6. /* Specific for most platforms */
7. h1 {
8.     font-family: "Helvetica Neue", Arial, "Hiragino Sans GB", "WenQuanYi Micro
9.     Hei", "Microsoft YaHei", sans-serif;
10. }
```

[强制] `font-family` 不区分大小写，但在同一个项目中，同样的大小写必须统一。

Family Name

示例：

```
1. /* good */
2. body {
3.     font-family: Arial, sans-serif;
4. }
5.
6. h1 {
7.     font-family: Arial, "Microsoft YaHei", sans-serif;
8. }
9.
10. /* bad */
11. body {
12.     font-family: arial, sans-serif;
13. }
14.
15. h1 {
16.     font-family: Arial, "Microsoft YaHei", sans-serif;
17. }
```

5.2 字号

[强制] 需要在 Windows 平台显示的中文内容，其字号应不小于 `12px`。

解释：

由于 Windows 的字体渲染机制，小于 `12px` 的文字显示效果极差、难以辨认。

5.3 字体风格

[建议] 需要在 Windows 平台显示的中文内容，不要使用除 `normal` 外的 `font-style`。其他平台也应慎用。

解释：

由于中文字体没有 `italic` 风格的实现，所有浏览器下都会 fallback 到 `oblique` 实现（自动拟合为斜体），小字号下（特别是 Windows 下会在小字号下使用点阵字体的情况下）显示效果差，造成阅读困难。

5.4 字重

[强制] `font-weight` 属性必须使用数值方式描述。

解释：

CSS 的字重分 100 - 900 共九档，但目前受字体本身质量和浏览器的限制，实际上支持 `400` 和 `700` 两档，分别等价于关键词 `normal` 和 `bold`。

浏览器本身使用一系列[启发式规则](#)来进行匹配，在 `<700` 时一般匹配字体的 Regular 字重，`>=700` 时匹配 Bold 字重。

但已有浏览器开始支持 `=600` 时匹配 Semibold 字重（见[此表](#)），故使用数值描述增加了灵活性，也更简短。

示例：

```
1. /* good */
2. h1 {
3.     font-weight: 700;
4. }
5.
6. /* bad */
7. h1 {
8.     font-weight: bold;
9. }
```

5.5 行高

[建议] `line-height` 在定义文本段落时，应使用数值。

解释：

将 `line-height` 设置为数值，浏览器会基于当前元素设置的 `font-size` 进行再次计算。在不同字号的文本段落组合中，能达到较为舒适的行间间隔效果，避免在每个设置了 `font-size` 都需要设置 `line-height`。

当 `line-height` 用于控制垂直居中时，还是应该设置成与容器高度一致。

示例：

```
1. .container {  
2.     line-height: 1.5;  
3. }
```

6 变换与动画

[强制] 使用 `transition` 时应指定 `transition-property`。

示例：

```
1. /* good */  
2. .box {  
3.     transition: color 1s, border-color 1s;  
4. }  
5.  
6. /* bad */  
7. .box {  
8.     transition: all 1s;  
9. }
```

[建议] 尽可能在浏览器能高效实现的属性上添加过渡和动画。

解释：

见[本文](#)，在可能的情况下应选择这样四种变换：

- `transform: translate(npx, npx);`
- `transform: scale(n);`
- `transform: rotate(ndeg);`
- `opacity: 0..1;`

典型的，可以使用 `translate` 来代替 `left` 作为动画属性。

示例：

```
1.  /* good */
2.  .box {
3.      transition: transform 1s;
4.  }
5.  .box:hover {
6.      transform: translate(20px); /* move right for 20px */
7.  }
8.
9.  /* bad */
10. .box {
11.     left: 0;
12.     transition: left 1s;
13. }
14. .box:hover {
15.     left: 20px; /* move right for 20px */
16. }
```

7 响应式

[强制] `Media Query` 不得单独编排，必须与相关的规则一起定义。

示例：

```
1.  /* Good */
2.  /* header styles */
3.  @media (...) {
4.      /* header styles */
5.  }
6.
7.  /* main styles */
8.  @media (...) {
9.      /* main styles */
10. }
11.
12. /* footer styles */
13. @media (...) {
14.     /* footer styles */
```

```
15. }
16.
17.
18. /* Bad */
19. /* header styles */
20. /* main styles */
21. /* footer styles */
22.
23. @media (...) {
24.     /* header styles */
25.     /* main styles */
26.     /* footer styles */
27. }
```

[强制] Media Query 如果有多个逗号分隔的条件时，应将每个条件放在单独一行中。

示例：

```
1. @media
2. (-webkit-min-device-pixel-ratio: 2), /* Webkit-based browsers */
3. (min--moz-device-pixel-ratio: 2),    /* Older Firefox browsers (prior to
   Firefox 16) */
4. (min-resolution: 2dppx),            /* The standard way */
5. (min-resolution: 192dpi) {          /* dppx fallback */
6.     /* Retina-specific stuff here */
7. }
```

[建议] 尽可能给出在高分辨率设备（Retina）下效果更佳样式。

8 兼容性

8.1 属性前缀

[强制] 带私有前缀的属性由长到短排列，按冒号位置对齐。

解释：

标准属性放在最后，按冒号对齐方便阅读，也便于在编辑器内进行多行编辑。

示例：

```
1. .box {  
2.     -webkit-box-sizing: border-box;  
3.     -moz-box-sizing: border-box;  
4.     box-sizing: border-box;  
5. }
```

8.2 Hack

[建议] 需要添加 `hack` 时应尽可能考虑是否可以采用其他方式解决。

解释：

如果能通过合理的 HTML 结构或使用其他的 CSS 定义达到理想的样式，则不应该使用 hack 手段解决问题。通常 hack 会导致维护成本的增加。

[建议] 尽量使用 `选择器 hack` 处理兼容性，而非 `属性 hack`。

解释：

尽量使用符合 CSS 语法的 selector hack，可以避免一些第三方库无法识别 hack 语法的问题。

示例：

```
1. /* IE 7 */  
2. *:first-child + html #header {  
3.     margin-top: 3px;  
4.     padding: 5px;  
5. }  
6.  
7. /* IE 6 */  
8. * html #header {  
9.     margin-top: 5px;  
10.    padding: 4px;  
11. }
```

[建议] 尽量使用简单的 `属性 hack`。

示例：

```
1. .box {  
2.     _display: inline; /* fix double margin */  
3.     float: left;  
4.     margin-left: 20px;
```

```
5.  }  
6.  
7.  .container {  
8.      overflow: hidden;  
9.      *zoom: 1; /* triggering hasLayout */  
10. }
```

8.3 Expression

[强制] 禁止使用 `Expression`。

Less编码规范

Less 编码规范（1.1）

简介

该文档主要的设计目标是提高 Less 文档的团队一致性与可维护性。

Less 代码的基本规范和原则与 [CSS 编码规范](#) 保持一致。

编撰

erik、顾轶灵、黄后锦、李玉北、赵雷。

本文档由 [商业运营体系前端技术组](#) 审校发布。

要求

在本文档中，使用的关键字会以中文+括号包含的关键字英文表示：必须（MUST）。关键字“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，and “OPTIONAL”被定义在rfc2119中。

编码

使用UTF-8编码。不得（MUST NOT）包含BOM信息。

代码组织

代码必须（MUST）按如下形式按顺序组织：

1. `@import`
2. 变量声明
3. 样式声明

```
1. // ✓  
2. @import "est/all.less";  
3.
```

```
4. @default-text-color: #333;  
5.  
6. .page {  
7.     width: 960px;  
8.     margin: 0 auto;  
9. }
```

@import

语句

`@import` 语句引用的文件必须（MUST）写在一对引号内，`.less` 后缀不得（MUST NOT）省略（与引入 CSS 文件时的路径格式一致）。引号使用 `'` 和 `"` 均可，但在同一项目内必须（MUST）统一。

```
1. // x  
2. @import 'est/all';  
3. @import "my/mixins.less";  
4.  
5. // ✓  
6. @import "est/all.less";  
7. @import "my/mixins.less";
```

空格

属性、变量

选择器和 `{` 之间必须（MUST）保留一个空格。

属性名后的冒号（`:`）与属性值之间必须（MUST）保留一个空格，冒号前不得（MUST NOT）保留空格。

定义变量时冒号（`:`）与变量值之间必须（MUST）保留一个空格，冒号前不得（MUST NOT）保留空格。

在用逗号（`,`）分隔的列表（Less 函数参数列表、以 `,` 分隔的属性值等）中，逗号后必须（MUST）保留一个空格，逗号前不得（MUST NOT）保留空格。







```
1. // x  
2. .box{  
3.     @w:50px;
```

```

4.     @h :30px;
5.     width:@w;
6.     height :@h;
7.     color: rgba(255,255,255,.3);
8.     transition: width 1s,height 3s;
9. }
10.
11. // ✓
12. .box {
13.     @w: 50px;
14.     @h: 30px;
15.     width: @w;
16.     height: @h;
17.     transition: width 1s, height 3s;
18. }

```

运算



 /  /  /  四个运算符两侧必须（MUST）保留一个空格。 /  两侧的操作数必须（MUST）有相同的单位，如果其中一个是变量，另一个数值必须（MUST）书写单位。

```

1. // ✗
2. @a: 200px;
3. @b: (@a+100)*2;
4.
5. // ✓
6. @a: 200px;
7. @b: (@a + 100px) * 2;

```

混入 (Mixin)

Mixin 和后面的空格之间不得（MUST NOT）包含空格。在给 mixin 传递参数时，在参数分隔符（ / ）后必须（MUST）保留一个空格：

```

1. // ✗
2. .box {
3.     .size(30px,20px);
4.     .clearfix ();
5. }
6.
7. // ✓

```



```
8.  .box {  
9.    .size(30px, 20px);  
10.   .clearfix();  
11. }
```

选择器

当多个选择器共享一个声明块时，每个选择器声明必须（MUST）独占一行。

```
1.  // ✗  
2.  h1, h2, h3 {  
3.    font-weight: 700;  
4.  }  
5.  
6.  // ✓  
7.  h1,  
8.  h2,  
9.  h3 {  
10.   font-weight: 700;  
11. }
```

Class 命名不得以样式信息进行描述，如 `.float-right`、`text-red` 等。

省略与缩写

缩写

多个属性定义可以使用缩写时，尽量（SHOULD）使用缩写。缩写更清晰字节数更少。常见缩写有 `margin`、`border`、`padding`、`font`、`list-style` 等。在书写时必须（MUST）考量缩写展开后是否有不需要覆盖的属性内容被修改，从而带来副作用。

数值

对于处于 `(0, 1)` 范围内的数值，小数点前的 `0` 可以（MAY）省略，同一项目中必须（MUST）保持一致。

```
1.  // ✗  
2.  transition-duration: 0.5s, .7s;  
3.
```

```

4. // ✓
5. transition-duration: .5s, .7s;

```

0 值

当属性值为 0 时，必须（MUST）省略可省的单位（长度单位如 `px`、`em`，不包括时间、角度等如 `s`、`deg`）。

```

1. // ✗
2. margin-top: 0px;
3.
4. // ✓
5. margin-top: 0;

```

颜色

颜色定义必须（MUST）使用 `#rrggbb` 格式定义，并在可能时尽量（SHOULD）缩写为 `#rgb` 形式，且避免直接使用颜色名称与 `rgb()` 表达式。

```

1. // ✗
2. border-color: red;
3. color: rgb(254, 254, 254);
4.
5. // ✓
6. border-color: #f00;
7. color: #fefefe;

```

私有属性前缀

同一属性有不同私有前缀的，尽量（SHOULD）按前缀长度降序书写，标准形式必须（MUST）写在最后。且这一组属性以第一条的位置为准，尽量（SHOULD）按冒号的位置对齐。

```

1. // ✓
2. .box {
3.     -webkit-transform: rotate(30deg);
4.     -moz-transform: rotate(30deg);
5.     -ms-transform: rotate(30deg);
6.     -o-transform: rotate(30deg);
7.     transform: rotate(30deg);
8. }

```

其他

可以（MAY）在无其他更好解决办法时使用 CSS hack，并且尽量（SHOULD）使用简单的属性名 hack 如 `_zoom`、`*margin`。

可以（MAY）但谨慎使用 IE 滤镜。需要注意的是，IE 滤镜中图片的 URL 是以页面路径作为相对目录，而不是 CSS 文件路径。

嵌套和缩进

必须（MUST）采用 4 个空格为一次缩进，不得（MUST NOT）采用 TAB 作为缩进。

嵌套的声明块前必须（MUST）增加一次缩进，有多个声明块共享命名空间时尽量（SHOULD）嵌套书写，避免选择器的重复。

但是需注意的是，尽量（SHOULD）仅在必须区分上下文时才引入嵌套关系（在嵌套书写前先考虑如果不能嵌套，会如何书写选择器）。

```
1.  // ✗
2.  .main .title {
3.    font-weight: 700;
4.  }
5.
6.  .main .content {
7.    line-height: 1.5;
8.  }
9.
10. .main {
11.   .warning {
12.     font-weight: 700;
13.   }
14.
15.   .comment-form {
16.     #comment:invalid {
17.       color: red;
18.     }
19.   }
20. }
21.
22.  // ✓
23. .main {
24.   .title {
```

```
25.         font-weight: 700;
26.     }
27.
28.     .content {
29.         line-height: 1.5;
30.     }
31.
32.     .warning {
33.         font-weight: 700;
34.     }
35. }
36.
37. #comment:invalid {
38.     color: red;
39. }
```

变量

Less 的变量值总是以同一作用域下最后一个同名变量为准，务必注意后面的设定会覆盖所有之前的设定。

变量命名必须（MUST）采用 `@foo-bar` 形式，不得（MUST NOT）使用 `@fooBar` 形式。

```
1. // ✗
2. @sidebarWidth: 200px;
3. @width:800px;
4.
5. // ✓
6. @sidebar-width: 200px;
7. @width: 800px;
```

继承

使用继承时，如果在声明块内书写 `:extend` 语句，必须（MUST）写在开头：

```
1. // ✗
2. .sub {
3.     color: red;
4.     &:extend(.mod all);
```

```
5.  }  
6.  
7.  // ✓  
8.  .sub {  
9.      &:extend(.mod all);  
10.     color: red;  
11. }
```

混入 (Mixin)

在定义 mixin 时，如果 mixin 名称不是一个需要使用的 className，必须 (MUST) 加上括号，否则即使不被调用也会输出到 CSS 中。

```
1.  // ✗  
2.  .big-text {  
3.      font-size: 2em;  
4.  }  
5.  
6.  h3 {  
7.      .big-text;  
8.  }  
9.  
10. // ✓  
11. .big-text() {  
12.     font-size: 2em;  
13. }  
14.  
15. h3 {  
16.     .big-text();  
17. }
```

如果混入的是本身不输出内容 mixin，必须 (MUST) 在 mixin 后添加括号 (即使不传参数)，以区分这是否是一个 className (修改以后是否会影响 HTML)。

```
1.  // ✗  
2.  .box {  
3.      .clearfix;  
4.      .size (20px);  
5.  }  
6.
```

```
7.  // ✓
8.  .box {
9.      .clearfix();
10.     .size(20px);
11. }
```

Mixin 的参数分隔符使用 `,` 和 `;` 均可，但在同一项目中必须（MUST）保持一致。

命名空间

变量和 mixin 在命名时必须（MUST）遵循如下原则：

- 一个项目只能引入一个无命名前缀的基础样式库（如 est）
- 业务代码和其他被引入的样式代码中，变量和 mixin 必须有项目或库的前缀

字符串

在进行字符串转义时，使用 `~""` 表达式与 `e()` 函数均可，但在同一项目中必须（MUST）保持一致。

字符串两侧的引号必须（MUST）使用 `"`。

JS 表达式

可以（MAY）使用 JS 表达式（`~``）生成属性值或变量，其中包含的字符串两侧的引号尽量（SHOULD）使用单引号（`'`）。

注释

单行注释尽量（SHOULD）使用 `//` 方式。

```
1.  // Hide everything
2.  * {
3.      display: none;
4.  }
```

E-JSON数据传输标准

E - JSON数据传输标准

简介

E - JSON的设计目标是使业务系统向浏览器端传递的JSON数据保持一致，容易被理解和处理，并兼顾传输的数据量。E - JSON依托于http协议（rfc2616）与JSON数据交换格式（rfc4627）。

编撰

erik, 欧阳先伟

评审

曹特磊, 蓝晓斌, 李铮, 林攀辉, 童遥, 王志寿, 严俊羿

要求

在本文档中，使用的关键字会以中文+括号包含的关键字英文表示：必须(MUST)。关键字“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，and “OPTIONAL”被定义在rfc2119中。

JSON数据类型

JSON (JavaScript Object Notation) 是一种轻量级，基于文本，语言无关的数据交换格式。其包括了基本数据类型4种和复合数据类型2种，共6种数据类型。在下面章节中，JSON数据类型的表示法为JSON+空格+数据类型，如：JSON Array。

传输的数据，包括对象属性以及数组成员， 必须(MUST) 是6种JSON数据类型之一。 杜绝(MUST NOT) 使用function、Date等js对象类型。

基本数据类型

- Number可以表示整数和浮点数。
- Boolean可以表示真假，值为true或false。
- String表示一个字符串。
- Null通常用于表示空对象。

“true”和true，这两个数据代表的是不同的数据类型。非字符串类型数据输出时一定要(MUST NOT) 为两端加上双引号，否则可能产生不希望的后果(如if中判断“false”的结果是true)。其他容易产生错误的例子如：0和“0”等。

复合数据类型

Object是无序的集合，以键值对的方式保持数据。一个Object中包含零到多个name/value的数据，数据间以逗号(,)分隔。name为String类型，value可以是任意类型的数据。

Object的最后一个元素之后一定不要(MUST NOT) 加上分隔符的逗号，否则可能导致解析出错。

Array(数组)为多个值的有序集合，数组元素间以逗号(,)分隔。

http响应头

status

http响应的status 必须(MUST) 为200。通常JSON数据被用于通过XMLHttpRequest对象访问，通过javascript进行处理。返回错误的状态码可能导致错误不被响应，数据不被处理。

Content-Type

Content-Type字段定义了响应体的类型。一般情况下，浏览器会根据该类型对内容进行正确的处理。对于传输JSON数据的响应，Content-Type 推荐(RECOMMENDED) 设置为“text/javascript”或“text/plain”。避免(MUST NOT) 将Context-Type设置为text/html，否则可能导致安全问题。

Content-Type中可以指定字符集。通常 需要(SHOULD) 明确指定一个字符集。如果是通过XMLHttpRequest请求的数据，并且字符编码为UTF-8时，可以不指定字符集。

Context-Type示例

```
1. text/javascript; charset=UTF-8
```

数据字段

返回的数据包含在http响应体中。数据 必须(MUST) 是一个JSON Object。该Object可能包含3个字段：status, statusInfo, data。

status

status字段 必须(*MUST*) 是一个不小于0的JSON Number整数，表示请求的状态。这个字段 可以(*SHOULD*) 被省略，省略时和为0时表示同一含义。

0：表示server端理解了请求，成功处理并返回。

非0：表示发生错误。 可以(*SHOULD*) 根据错误类型扩展错误码。

一个成功请求的status字段

```
1. {  
2.     "status": 0,  
3.     "data": "hello world!"  
4. }
```

statusInfo

statusInfo字段 通常(*SHOULD*) 是一个JSON String或JSON Object，表示除了请求状态外server端想要对status做出的说明，使client端能够获取更多信息进行后续处理。这个字段是 可选的(*OPTIONAL*)。下面的两个例子中，statusInfo字段的信息都可以用于client端程序的后续处理，但是粒度和处理方式会有不同。

client端参数错误的statusInfo

简单说明的statusInfo：

```
1. {  
2.     "status": 1,  
3.     "statusInfo": "参数错误"  
4. }
```

具有更多信息的statusInfo：

```
1. {  
2.     "status": 1,  
3.     "statusInfo": {  
4.         "text": "参数错误",  
5.         "parameters": {  
6.             "email": "电子邮件格式不正确"  
7.         }  
8.     }  
9. }
```

data

data字段可以是除JSON `Null`之外的任意JSON类型，表示请求返回的数据主体。这个字段是 可选的 (*OPTIONAL*) 。数据主体data包含了在请求成功时有意义的的数据。

一个查询姓名请求的返回数据

```
1. {  
2.     "status": 0,  
3.     "data": "Lily"  
4. }
```

数据场景

本章为常见数据场景定义了通用的标准数据格式，用于数据传输和使用。额外地，本章为部分可能大数据量传输的数据场景定义了变通数据格式。变通数据格式可在数据解析阶段转换成标准数据格式。

变通数据格式 必须(*MUST*) 是一个JSON Object，其中 必须(*MUST*) 包含e-type属性和data属性。e-type属性标识数据类型，便于对数据进行解析；data属性包含变通后的数据。变通数据 可以 (*MAY*) 包含其他的属性，标识数据的其他扩展信息。

变通数据格式的e-type属性定义了table值。e-type属性可以使用者扩展其他属性值，扩展的属性值 必须(*MUST*) 以“项目缩写-名称”命名，如“fc-list”，自主解析。

日期类型

日期类型不属于JSON数据类型。对于日期类型，我们 必须(*MUST*) 使用JSON String来表示。为了让日期能够更容易的被显示和被解析，对于日期我们 应当(*SHOULD*) 使用更适合internet的格式，遵循rfc3339。

数据场景：日期

```
1. {  
2.     "status": 0,  
3.     "data": "2010-10-10"  
4. }
```

记录

记录代表二维表中的一行，通常用于表示某个具体事务抽象的属性。标准记录数据 必须(*MUST*) 为一个JSON Object，记录的主键命名 必须(*MUST*) 为“id”。单条记录数据不包含变通数据格式。

数据场景：记录

```

1.  {
2.      "id": 250,
3.      "name": "erik",
4.      "sex": 1,
5.      "age": 18
6.  }
```

二维表

二维表类型标识为table，是关系模型的主要数据结构。二维表结构具有变通数据格式。标准二维表数据 必须(MUST) 以一维JSON Array形式表示，JSON Array中每一项是一个JSON Object，代表一条记录。JSON Object的每个成员代表一个字段。每条记录的主键命名 必须(MUST) 为"id"。

在标准二维表中，字段名在每条记录中都被传输，会造成额外的数据量传输。这个问题会随着记录数的增大会更加突出。为了减少传输数据量，变通格式使用二维JSON Array传输数据，扩展fields属性用于字段说明。fields字段为JSON Array。

数据场景：标准二维表

```

1.  [
2.      {
3.          "id": 250,
4.          "name": "erik",
5.          "sex": 1,
6.          "age": 18
7.      },
8.      {
9.          "id": 251,
10.         "name": "欧阳先伟",
11.         "sex": 1,
12.         "age": 28
13.      }
14.  ]
```

数据场景：变通二维表

```

1.  {
2.      "e-type": "table",
3.      "fields": ["id", "name", "sex", "age"],
```

```

4.     "data": [
5.         [250, "erik", 1, 18],
6.         [251, "欧阳先伟", 1, 28]
7.     ]
8. }

```

数据页

数据页是列表数据常用的数据方式，可能通过查询或翻页获得数据。数据页是二维表数据的包装，包含列表数据本身更多的信息。

数据页 必须(*MUST*) 是一个JSON Object，其中 必须(*MUST*) 包含的属性为data。data是一个二维表。数据页可以包括一些 可选(*OPTIONAL*) 的属性，表示当前数据页的信息。下表列举了数据页的可选属性。

数据页可选属性

- {Number} page - 当前页码，计数 必须(*MUST*) 为不小于0的整数，从0开始。
- {Number} pageSize - 每页显示条数， 必须(*MUST*) 大于0。
- {Number} total - 列表总记录数， 必须(*MUST*) 为不小于0的整数。表示当前条件下所有记录的数目，非本页的记录数。
- {String} orderBy - 列表排序规则。多个排序规则之间以逗号分割(,)；正序或倒序以asc或desc表示，与字段名之间以一个空格间隔。例："id desc,name asc"
- {String} keyword - 列表所属的搜索关键字。
- {Object} condition - 列表所属的搜索条件集合。属性中可以包含或不包含keyword字段，如果不包含， 建议(*RECOMMENDED*) 在解析的时候附加搜索关键字keyword条件。

数据场景：数据页

```

1.  {
2.     "page": 0,
3.     "pageSize": 30,
4.     "keyword": "",
5.     "data": [
6.         {
7.             "id": 250,
8.             "name": "erik",
9.             "sex": 1,
10.            "age": 18
11.        },
12.        {

```

```
13.         "id": 251,  
14.         "name": "欧阳先伟",  
15.         "sex": 1,  
16.         "age": 28  
17.     }  
18. ]  
19. }
```

键/值对象

对于在一个JSON Object中表示键/值：

- 键的属性名 必须(*MUST*) 为name， 杜绝(*MUST NOT*) 使用key或k
- 值的属性名 必须(*MUST*) 为value， 杜绝(*MUST NOT*) 使用v。

数据场景：键/值对象

```
1. {  
2.     "name": "BMW",  
3.     "value": 1  
4. }
```

键/值有序集合

键/值有序集合表示对事务或逻辑类型的抽象与分类。常见的应用场景有单选复选框集合，下拉菜单等。

标准的键/值有序集合是一个JSON Array，集合中的每一项是一个JSON Object。项 必须(*MUST*) 包含name和value属性。 可以(*MAY*) 通过其他的属性修饰每一项的特殊信息，如selected。

数据场景：键/值有序集合

```
1. [  
2.     {  
3.         "name": "BMW",  
4.         "value": 1  
5.     },  
6.     {  
7.         "name": "Benz",  
8.         "value": 2,  
9.         "selected": true  
10.    }  
]
```

```
11.  ]
```

树

树形数据用于表示层叠的数据结构。树型数据 必须(*MUST*) 是一个JSON Object，代表树型数据的根节点。下面是标准定义的可选节点列表，不在列表中的属性 可以(*SHOULD*) 自行扩展。

树型数据结构的可选节点属性

- {Number|String} id - 节点的唯一标识。
- {String} text - 名称或用于显示的字符串。
- {Array} children - 子节点列表。

数据场景：树型数据

```
1.  {
2.      "id": 1,
3.      "text": "中国",
4.      "children": [
5.          {
6.              "id": 10,
7.              "text": "北京",
8.              "children": [
9.                  {
10.                      "id": 100,
11.                      "text": "东城区"
12.                  },
13.                  {
14.                      "id": 101,
15.                      "text": "西城区"
16.                  },
17.                  {
18.                      "id": 102,
19.                      "text": "海淀区"
20.                  }
21.                  .....
22.              ]
23.          },
24.          {
25.              "id": 31,
26.              "text": "海南",
27.              "children": [
```

```
28.         {
29.             "id": 600,
30.             "text": "海口"
31.         },
32.         {
33.             "id": 601,
34.             "text": "三亚"
35.         },
36.         {
37.             "id": 602,
38.             "text": "五指山"
39.         }
40.         .....
41.     ]
42. }
43. ....
44. ]
45. }
```

引用

- [RFC 2119] “Key words for use in RFCs to Indicate Requirement Levels”
- [RFC 4627] “The application/json Media Type for JavaScript Object Notation (JSON)”
- [RFC 2616] “Hypertext Transfer Protocol”
- [RFC 3339] “Date and Time on the Internet: Timestamps”

模块和加载器规范

模块和加载器规范

简介

该文档主要的设计目标是定义前端代码的模块规范，便于开发资源的共享和复用。该文档在 `amdjs` 规范的基础上，进行了更细粒度的规范化。

编撰

李玉北、erik、黄后锦、王杨、张立理、赵雷、陈新乐、顾轶灵、林志峰、刘恺华。

本文档由 `商业运营体系前端技术组` 审校发布。

要求

在本文档中，使用的关键字会以中文+括号包含的关键字英文表示： 必须(MUST)。关键字“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，and “OPTIONAL”被定义在rfc2119中。

模块定义

模块定义 必须(MUST) 采用如下的方式：

```
1. define( factory );
```

推荐采用 `define(factory)` 的方式进行 `模块定义`。使用匿名 `moduleId`，从而保证开发中模块与路径相关联，有利于模块的管理与整体迁移。

SHOULD NOT使用如下的方式：

```
1. define( moduleId, deps, factory );
```

moduleId

`moduleId` 的格式应该符合 `amdjs` 中的约束条件。

1. `moduleId` 的类型应该是 `string`，并且是由 `/` 分割的一些 `term` 来组成。例

如: `this/is/a/moduleId`。

2. `term` 应该符合 `[a-zA-Z0-9_]+` 这个规则。
3. `moduleId` 不应该有 `.js` 后缀。
4. `moduleId` 应该跟文件的路径保持一致。

`moduleId` 在实际使用 (如 `require`) 的时候, 又可以分为如下几种类型:

1. `relative moduleId`: 是以 `./` 或者 `../` 开头的 `moduleId`。例如: `./foo`, `../../bar`。
2. `top-level moduleId`: 除上面两种之外的 `moduleId`。例如 `foo`, `bar/a`, `bar/b`。

在模块定义的时候, `define` 的第一个参数如果是 `moduleId`, 必须(MUST) 是 `top-level moduleId`, 不允许(MUST NOT) 是 `relative moduleId`。

factory

AMD风格与CommonJS风格

模块的 `factory` 有两种风格, `AMD推荐的风格` 和 `CommonJS的风格`。`AMD推荐的风格` 通过返回一个对象做为模块对象, `CommonJS的风格` 通过对 `module.exports` 或 `exports` 的属性 赋值来达到暴露模块对象的目的。

建议(SHOULD) 使用 `AMD推荐的风格`, 其更符合Web应用的习惯, 对模块的数据类型也便于管理。

```

1.  // AMD推荐的风格
2.  define( function( require ) {
3.      return {
4.          method: function () {
5.              var foo = require("./foo/bar");
6.              // blabla...
7.          }
8.      };
9.  });
10.
11. // CommonJS的风格
12. define( function( require, exports, module ) {
13.     module.exports = {
14.         method: function () {
15.             var foo = require("./foo/bar");
16.             // blabla...
17.         }
18.     };
19. });

```

参数

模块的 `factory` 默认有三个参数，分别是 `require` , `exports` , `module` 。

```
1. define( function( require, exports, module ) {  
2.     // blabla...  
3. });
```

使用 `AMD推荐风格` 时, `exports` 和 `module` 参数可以省略。

```
1. define( function( require ) {  
2.     // blabla...  
3. });
```

开发者 不允许(*MUST NOT*) 修改 `require` , `exports` , `module` 参数的形参名称。下面就是错误的用法:

```
1. define( function( req, exp, mod ) {  
2.     // blablabla...  
3. });
```

类型

`factory` 可以是任何类型，一般来说常见的就是三种类型 `function` , `string` , `object` 。当 `factory` 不是 `function` 时，将直接做为模块对象。

```
1. // src/foo.js  
2. define( "hello world. I'm {name}" );  
3.  
4. // src/bar.js  
5. define( {"name": "fe"} );
```

上面这两种写法等价于:

```
1. // src/foo.js  
2. define( function(require) {  
3.     return "hello world. I'm {name}";  
4. });  
5.  
6. // src/bar.js
```

```

7.  define( function(require) {
8.      return {"name": "fe"};
9.  } );

```

require

`require` 这个函数的参数是 `moduleId`，通过调用 `require` 我们就可以引入其他的模块。`require` 有两种形式：

```

1.  require( {string} moduleId );
2.  require( {Array} moduleIdList, {Function} callback );

```

`require` 存在 `local require` 和 `global require` 的区别。

在 `factory` 内部的 `require` 是 `local require`，如果 `require` 参数中的 `moduleId` 的类型是 `relative moduleId`，那么相对的是当前 `模块id`。

在全局作用域下面调用的 `require` 是 `global require`，`global require` 不支持 `relative moduleId`。

```

1.  // src/foo.js
2.  define( function( require ) {
3.      var bar = require("./bar"); // local require
4.  });
5.
6.  // src/main.js
7.  // global require
8.  require( ['foo', 'bar'], function ( foo, bar ) {
9.      // blablalbla...
10. });

```

exports

`exports` 是使用 `CommonJS风格` 定义模块时，用来公开当前模块对外提供的API的。另外也可以忽略 `exports` 参数，直接在 `factory` 里面返回自己想公开的API。例如下面三种写法功能是一样的：

```

1.  define( function( require, exports, module ) {
2.      exports.name = "foo";
3.  });
4.
5.  define( function( require, exports, module ) {

```

```

6.     return { "name" : "foo" };
7. });
8.
9. define( function( require, exports, module ) {
10.     module.exports.name = "foo";
11. });

```

`module` 是当前模块的一些信息，一般不会用到。其中 `module.exports === exports`。

dependencies

模块和模块的依赖关系需要通过 `require` 函数调用来保证。

```

1. // src/js/ui/Button.js
2. define( function( require, exports, module ) {
3.     require("css!../../css/ui/Button.css");
4.     require("tpl!../../tpl/ui/Button.tpl.html");
5.
6.     var Control = require("ui/Control");
7.
8.     /**
9.      * @constructor
10.     * @extends {Control}
11.     */
12.     function Button() {
13.         Control.call(this);
14.
15.         var foo = require("./foo");
16.         foo.bar();
17.     }
18.     baidu.inherits(Button, Control);
19.
20.     ...
21.
22.     // exports = Button;
23.     // return Button;
24. });

```

具体实现的时候是通过正则表达式分析 `factory` 的函数体来识别出来的。因此为了保证识别的正确率，请尽量

避免在函数体内定义 `require` 变量或者 `require` 属性。例如不要这么做：

```

1. var require = function(){};
2. var a = {require:function(){};};
3. a.require("./foo");
4. require("./bar");

```

模块加载器配置

AMD Loader 应该支持如下的配置，更新配置的时候，写法如下：

```

1. <script src="{amdloader.js}"></script>
2. <script>
3.   require.config({
4.     ....
5.   });
6. </script>

```

baseUrl

类型应该是 **string**。在 **ID-to-path** 的阶段，会以 **baseUrl** 作为根目录来计算。如果没有配置的话，就默认以当前页面所在的目录为 **baseUrl**。

如果 **baseUrl** 的值是 **relative**，那么相对的是当前页面，而不是 **AMD Loader** 所在的位置。

paths

类型应该是 **Object.<string, string>**。它维护的是 **moduleId** 前缀到路径的映射规则。这个对象中的 **key** 应该是 **moduleId** 的前缀，**value** 如果是一个相对路径的话，那么相对的是 **baseUrl**。当然也可以是绝对路径的话，例

如：**/this/is/a/path**，**//www.google.com/this/is/a/path**。

```

1. {
2.   baseUrl: '/fe/code/path',
3.   paths: {
4.     'ui': 'esui/v1.0/ui',
5.     'ui/Panel': 'esui/v1.2/ui/Panel',
6.     'tangram': 'third_party/tangram/v1.0',
7.     'themes': '//www.baidu.com/css/styles/blue'
8.   }
9. }

```

在 **ID-to-path** 的阶段，如果 **模块** 或者 **资源** 是以 **ui**，**ui/Panel**，**tangram** 开头的

话，那么就会去配置指定的地方去加载。例如：

- `ui/Button` => `/fe/code/path/esui/v1.0/ui/Button.js`
- `ui/Panel` => `/fe/code/path/esui/v1.2/ui/Panel.js`
- `js!tangram` => `/fe/code/path/third_party/tangram/v1.0/tangram.js`
- `css!themes/base` => `//www.baidu.com/css/styles/blue/base.css`

另外，需要支持为插件指定不同的 `paths`，语法如下：

```
1. {
2.   baseUrl: '/fe/code/path',
3.   paths: {
4.     'css!': '//www.baidu.com/css/styles/blue',
5.     'css!foo': 'bar',
6.     'js!': '//www.google.com/js/gcl',
7.     'js!foo': 'bar'
8.   }
9. }
```

模块加载器插件

该文档不限定使用何种 `AMD Loader`，但是一个 `AMD Loader` 应该支持至少三种插件（`css`，`js`，`tpl`）才能满足我们的业务需求。

插件语法

```
1. [Plugin Module ID]![resource ID]
```

`Plugin Module Id` 是插件的 `moduleId`，例如 `css`，`js`，`tpl` 等等。`!` 是分割符。

`resource ID` 是 `资源Id`，可以是 `top-level` 或者 `relative`。如果 `resource ID` 是 `relative`，那么相对的是当前 `模块的Id`，而不是当前 `模块Url`。例如：

```
1. // src/Button.js
2. define( function( require, exports, module ){
3.   require( "css!./css/Button.css" );
4.   require( "css!base.css" );
5.   require( "tpl!./tpl/Button.tpl.html" );
6. });
```

如果当前模块的路径是 `${root}/src/ui/Button.js`，那么该模块依赖

的 `Button.css` 和 `Button.tpl.html` 的路径就应该分别是 `${root}/src/css/ui/Button.css`，`${root}/src/tpl/Button.tpl.html`；该模块依赖的 `base.css` 的路径应该是 `${baseUrl}/base.css`。

css插件

参考上面的示例。如果 `resource ID` 省略后缀名的话，默认是 `.css`；如果有后缀名，以具体的后缀名为准。例如：`.less`。

js插件

用来加载不符合该文档规范的js文件，例如 `jquery`，`tangram` 等等。例如：

```
1. // src/js/ui/Button.js
2. define( function( require, exports, module ) {
3.     require( "js!jquery" );
4.     require( "js!./tangram" );
5. });
```

tpl插件

如果项目需要前端模板，需要通过tpl插件加载。tpl插件由模板引擎提供方实现。插件的语法应该跟上述 `js`，`css` 插件的语法保持一致，例如：

```
1. require( "tpl!./foo.tpl.html" );
```

FAQ

为什么不能采用`define(moduleId, deps, factory)`来定义模块？

`define(moduleId, deps, factory)` 这种写法，很容易出现很长的deps，影响代码的风格。

```
1. define(
2.     "module/id",
3.     [
4.         "module/a",
5.         "module/b",
6.         "module/c"
7.     ],
8.     function ( require ) {
```

```
9.      // blabla...
10.    }
11.  );
```

构建工具对代码进行处理和编译时，允许将代码编译成这种风格，明确硬依赖。

相对于模块的Id和相对于模块Url有什么区别？

还是看 [erik的解释吧](#)

包结构规范

包结构规范（1.1）

简介

该文档主要的设计目标是商业体系 **前端** 资源分包进行约定规范，使开发资源容易被共享和复用。

编撰

李玉北、erik、黄后锦、王杨、张立理、赵雷。

本文档由 **商业运营体系前端技术组** 审校发布。

要求

在本文档中，使用的关键字会以中文+括号包含的关键字英文表示：必须(MUST)。关键字“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，and “OPTIONAL”被定义在rfc2119中。

规范说明约定

以下规范文档中，以 **`${root}`** 表示包的根目录。

包开发说明

包定义

包 是实现某个独立功能，有复用价值的代码集。在具体的实现过程中， **必须(MUST)** 按照**模块和加载器规范**来开发和管理模块。

模块定义

包 中的模块定义时 **必须(MUST)** 采用匿名id **`define(factory)`** 进行定义， **不允许(MUST NOT)** 使用 **`define(moduleId, factory)`**。

依赖管理

包 中的模块对其他模块的依赖分成两种：**内部模块依赖** 和 **外部包依赖**。下面是关于模块依赖的管理说明：

1. 对 **内部模块依赖** 的情况，必须(MUST) 保证内部模块id与路径的对应关系。require依赖引用必须(MUST) 使用 **relative id**，不允许(MUST NOT) 使用 **top-level id**。
2. 对 **外部包依赖** 的情况，require依赖引用 必须(MUST) 使用 **top-level id**。

开发时，我们通常会做一些测试用例或示例，此时需要通过AMD Loader将当前包粘合到页面环境，并使其可运行。这时我们需要遵守一些规则：

1. 对 **内部模块依赖**，AMD Loader配置 推荐(RECOMMENDED) 通过 **packages** 将 **location** 配置到 **\${root}** 下的 **src** 目录，不允许(MUST NOT) 通过 **paths** 进行路径映射。
2. 对 **外部包依赖**，请参照[项目目录结构规范](#)将相关依赖包导入，并且 必须(MUST) 通过 **packages** 项配置AMD Loader。

```

1. // 示例：ER package的test配置
2. require.config({
3.     packages: [
4.         {
5.             name: 'er',
6.             location: '../src',
7.             main: 'main'
8.         },
9.         {
10.            name: 'mini-event',
11.            location: '../dep/mini-event/1.0.0/src',
12.            main: 'main'
13.        },
14.        {
15.            name: 'etpl',
16.            location: '../dep/etpl/2.0.2/src',
17.            main: 'main'
18.        }
19.    ]
20. });

```

资源

允许(SHALL) 包含如下类型的资源：

脚本，样式以及样式相关图片，直接引用图片，html，模板，文档，测试套件。

包描述文件

包描述文件 必须(MUST) 置于 **\${root}** 下，命名为package.json， 必须(MUST) 是一个UTF-8

编码的严格JSON格式的文本文件。

必选字段

- **name** : 包名。 必须(MUST) 为由camel命名法产生的字母组成的字符串。
- **version** : 版本号。版本号 必须(MUST) 为字符串, 需要符合SemVer的格式约定。
- **maintainers** : 维护者列表。该字段 必须(MUST) 是一个数组, 数组中每项 必须(MUST) 包含维护者的名称字段" name "与电子邮件字段" email "。

可选字段

- **main** : 模块名, 用来说明当前 包 的入口文件。如果包名为 **foo** , 那么执行 `require("foo")` 的时候, 返回的内容就是当前模块 **exports** 的内容。
- **description** : 描述信息。 必须(MUST) 为字符串。
- **dependencies** : 依赖声明。该字段 必须(MUST) 是一个 **JSON Object** , 其中 **key** 为依赖的包名, **value** 为版本号, 支持如下的格式:
 - **version**
 - **>version**
 - **>=version**
 - **<version**
 - **<=version**
 - ***** : 任意版本
- **devDependencies** : 类似 **dependencies** 的角色, 但不是当前 包 必须的, 只是在开发和调试的时候一些依赖的内容。
- **contributors** : 贡献者列表。该字段 必须(MUST) 是一个数组, 数组中每项 必须(MUST) 包含维护者的名称字段 **name** 与电子邮件字段 **email** 。
- **homepage** : 该字段 必须(MUST) 为URL格式的字符串。

示例

```
1.  {
2.      "name": "zrender",
3.      "version": "0.0.1",
4.      "maintainers": [
5.          {"name": "foo", "email": "foo@baidu.com"},
6.          {"name": "bar", "email": "bar@baidu.com", "url":
7.              "http://www.baidu.com/bar"}
8.      ],
9.      "dependencies": {
10.          "foo": "0.0.1",
11.          "bar": "*"
12.      }
13.  }
```

```
11.         },
12.         "devDependencies": {
13.             "uglify-js": "*"
14.         }
15.     }
```

包目录结构

示例

```
1.  ${root}/
2.    package.json
3.    README.md
4.    src/
5.      css/[?]
6.      img/
7.      main.js
8.    dep/
9.      foo/
10.     bar/
11.     jquery/
12.     tangram/
13.     mustache/
14.     ...
15.   test/
16.   doc/
```

package.json

`package.json` 必须(*MUST*) 直接放在包顶层目录 `${root}` 中。

src

按照通常的理解，一个 `包项目` 不应该特别复杂。下面是一个简单的例子，详细的目录划分方法和原则请参考 [项目目录结构规范](#)。

```
1.  ${root}/
2.    src/
3.      main.js
4.      Control.js
```

```
5.      Button.js
6.      css/
7.          button.css
8.  package.json
9.  README.md
```

dep

用来存放 `dependencies package` 的代码， 不允许 (*MUST NOT*) 放置任何其它的内容。

参考资料

1. <https://npmjs.org/doc/json.html>
2. <http://twitter.github.com/bower/>
3. <http://yeoman.io/>
4. <http://semver.org/>

项目目录结构规范

项目目录结构规范

简介

该文档主要的设计目标是项目开发的目录结构保持一致，使容易理解并方便构建与管理。

编撰

李玉北、erik、黄后锦、王杨、张立理、赵雷、陈新乐、刘恺华。

本文档由 **商业运营体系前端技术组** 审校发布。

要求

在本文档中，使用的关键字会以中文+括号包含的关键字英文表示：必须(MUST)。关键字“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，and “OPTIONAL”被定义在rfc2119中。

规范说明约定

以下规范文档中：

1. **项目** 包含但不限于 **业务项目** 和 **包项目**。
2. **`${root}`** 表示 **项目** 的根目录。

资源分类

资源 分成两大类：

1. **源代码资源**：指开发者编写的源代码，包括 **js**、**html**、**css**、**template** 等。
2. **内容资源**：指希望做为内容提供给访问者的资源，包括 **图片**、**字体**、**flash**、**pdf** 等。

目录命名原则

1. 简洁。有习惯性缩写的单词 必须(MUST) 采用容易理解的缩写。如：源代码目录使用 **src**，不使用 **source**。下面是更多例子：
 - i. **img**：图片。不允许(MUST NOT) 使用 **image**、**images**、**imgs** 等。

- ii. `js` : javascript脚本。 不允许(MUST NOT) 使用 `script`、`scripts` 等。
 - iii. `css` : 样式表。 不允许(MUST NOT) 使用 `style`、`styles` 等。
 - iv. `swf` : flash。 不允许(MUST NOT) 使用 `flash` 等。
 - v. `src` : 源文件目录。 不允许(MUST NOT) 使用 `source` 等。
 - vi. `dep` : 引入的第三方依赖包目录。 不允许(MUST NOT) 使用 `lib`、`library`、`dependency` 等。
2. 不允许(MUST NOT) 使用复数形式。如: `imgs`、`docs` 是不被允许的。

目录划分

`${root}`目录结构划分

在`${root}`下, 目录结构 必须(MUST) 按照 `职能` 进行划分, 不允许(MUST NOT) 将 `资源类型` 或 `业务逻辑` 划分的目录直接置于`${root}`下。

常用的目录有 `src`、`doc`、`dep`、`test` 等。详细请参考[一级目录详细说明](#)

```
1. ${root}/
2.   src/
3.   test/
4.   doc/
5.   dep/
6.   ...
```

业务项目目录结构划分

`业务项目` 的`${root}`目录结构划分遵循[\\${root}目录结构划分](#)。

项目代号

业务项目 可以(SHOULD) 为项目起一个代号名称。代号名称 必须(MUST) 为一个单词, 不宜过长。
例: 北斗的项目代号为 `triones`, 哥伦布的项目代号为 `clb`, 百度锦囊的项目代号为 `jn`。项目代号有利于区分不同项目, 为未来项目之间的重用留下扩展的后路。

在项目开发时, 通常会使用如下[加载器配置](#), 将项目代号指向 `src`。

```
1. {
2.   baseUrl: '${docroot}',
3.   paths: {
4.     'triones': 'src'
5.   }
```

```
6. }
```

根据业务逻辑划分src目录结构

业务项目 的 **src** 目录内，绝大多数情况 应当(*SHOULD*) 根据 **业务逻辑** 划分目录结构。划分出的子目录（比如例子中的 **biz1**）我们称为 **业务目录**。

src 下 必须(*MUST*) 只包含 **业务目录** 与 **common** 目录。**业务公共资源** 必须(*MUST*) 命名为 **common**。**common** 目录做为 **业务公共资源** 的目录，也视如 **业务目录**。

```
1. ${root}/
2.   src/
3.     common/
4.     biz1/
5.       subbiz1/
6.       subbiz2/
7.     biz2/
```

较小规模的 **业务项目**（如投放端），**src** 目录允许视如 **业务目录**，直接按照**业务目录划分原则**划分目录结构。

```
1. ${root}/
2.   src/
3.     foo.js
```

业务目录划分原则

- JS资源** 不允许(*MUST NOT*) 按 **资源类型** 划分目录， 必须(*MUST*) 按 **业务逻辑** 划分目录。**JS资源** 应直接置于 **业务目录** 下。即：**业务目录** 下不允许出现 **js** 目录。
- 除 **JS资源** 外的 **源文件资源**，当资源数量较多时，为方便管理， 允许(*SHOULD*) 按 **资源类型** 划分目录。即：**业务目录** 下允许出现 **css**、**tpl** 目录。
- 内容资源** 允许(*SHOULD*) 按 **资源类型** 划分目录。即：**业务目录** 下允许出现 **img**、**swf**、**font** 目录。
- 业务目录** 中，如果文件太多不好管理，需要划分子目录时，也 必须(*MUST*) 继续遵守根据 **业务逻辑** 划分的原则，划分子业务。如：下面例子中的 **subbiz1**。

通常，对于一个 **业务目录**， 鼓励(*SHOULD*) 将业务相关的 **源文件资源** 都直接置于 **业务目录** 下。

```
1. biz1/
2.   img/
```



```
3.      add_button.png
4.      add.js
5.      add.tpl.html
6.      add.css
```

[业务目录](#) 下 [源文件资源](#) 数量较多时，我们第一直觉应该是：是否业务划分不够细？是否应该划分子业务，建立子业务目录？

```
1.  biz2/
2.      subbiz1/
3.          list.js
4.          list.tpl.html
5.          list.css
6.      subbiz2/
```

遇到确实是一个业务整体，无法划分子业务时， 允许(MAY) 将非 [JS资源](#) 按 [资源类型](#) 划分目录进行管理。

```
1.  biz1/
2.      css/
3.          add.css
4.          edit.css
5.          remove.css
6.      img/
7.          add_button.png
8.      tpl/
9.          add.html
10.         edit.html
11.         remove.html
12.     add.js
13.     edit.js
14.     remove.js
```

[源文件资源](#) 和 [内容资源](#) 请参考[资源分类](#)章节，常用 [资源目录](#) 请参考[资源目录](#)章节，常用 [业务目录](#) 请参考[业务目录](#)章节。

业务项目目录划分示例

```
1.  ${root}/
2.      src/
3.          common/
```

```
4.          img/
5.          sprites.png
6.          logo.png
7.          conf.js
8.          layout.css
9.      biz1/
10.         img/
11.         add_button.png
12.         add.js
13.         add.tpl.html
14.         add.less
15.     biz2/
16.         subbiz1/
17.             list.js
18.             list.tpl.html
19.             list.css
20.         subbiz2/
21.     dep/
22.     er/
23.         src/
24.         test/
25.     esui/
26.         src/
27.         test/
28.     test/
29.     doc/
30.     index.html
31.     main.html
32.     .....
```

包项目目录结构划分

包项目 的 $\${root}$ 目录结构划分遵循 $\${root}$ 目录结构划分。

包项目src目录结构划分

包 是实现某个独立功能，有复用价值的代码集。按照通常的理解，一个 **包项目** 不应该特别复杂。

所以，**包** 可视如一个不太复杂的 **业务**，其 **src** 下的划分原则与 **业务项目** 的**业务目录划分原则**保持一致。

```
1.  $\${root}/$ 
```

```
2.      src/  
3.          css/  
4.              img/  
5.                  sprites.png  
6.                  table.css  
7.                  button.css  
8.                  select.css  
9.      main.js  
10.     Control.js  
11.     InputControl.js  
12.     Button.js  
13.     Table.js  
14.     Select.js  
15.     test/  
16.     doc/  
17.     package.json  
18.     ...
```

常用目录

一级目录

直接置于 `${root}` 下的目录称作 **一级目录**。一级目录 *必须(MUST)* 具有某种 **职能** 属性。

除了下面列举的一些常见目录之外，`${root}` 下面也可以放置一些跟项目发布相关的文件，例如 `build.sh`，`build.xml`，`Makefile`，`Gruntfile` 等等。

src

src 目录用于存放开发时源文件，发布时 *必须(MUST)* 被删除。

dep

dep 目录用于存放 **项目** 引入依赖的第三方包。该目录下的内容通过平台工具管理，项目开发人员 *不允许(MUST NOT)* 更改 **dep** 目录下第三方包的任何内容。

当项目需要修改引入的第三方代码时，第三方包应将源码直接置于 `${root}/src` 目录下，规则见该目录下的规定。

更多关于 **包** 的内容请参考 [包结构规范](#)

tool

tool 目录用于存放开发时或构建阶段使用的工具。该目录在发布时 必须(*MUST*) 被删除。

test

test 目录用于存放测试用例以及开发阶段的模拟数据。该目录在发布时 必须(*MUST*) 被删除。

doc

doc 目录用于存放项目文档。项目文档可能是开发者维护的文档，也可能是通过工具生成的文档。

entry

entry 目录用于存放项目的 **页面入口文件**，通常是上线后可被直接访问的静态页面。

RIA项目 通常会包含较少的 **页面入口文件**，常见的是 **main.html**，这些文件 可以(*SHOULD*) 直接放在 **\${root}** 目录下。

```
1. ${root}/
2.     src/
3.         common/
4.             conf.js
5.         card/
6.         gold/
7.         message/
8.     index.html
9.     main.html
10.     .....
```

多页面项目 通常 **页面入口文件** 较多， 可以(*SHOULD*) 统一放在 **entry** 目录中，按 **业务逻辑** 命名。

```
1. ${root}/
2.     src/
3.         common/
4.             conf.js
5.         card/
6.         gold/
7.         message/
8.     entry/
9.         card.html
10.        gold.html
11.        message.html
12.        .....
```

项目在发布的时候，构建工具可以 `页面入口文件` 为入口进行分析和编译。

`RIA项目` 经过构建工具编译后，目录结构可能如下：

```
1. output/  
2.     asset/  
3.         js/  
4.         css/  
5.         tpl/  
6.         img/  
7.     index.html  
8.     main.html
```

`多页面项目` 经过构建工具编译后，目录结构可能如下：

```
1. output/  
2.     card/  
3.         asset/  
4.             js/  
5.             css/  
6.             img/  
7.         index.html  
8.     gold/  
9.         asset/  
10.            js/  
11.            css/  
12.            img/  
13.        index.html
```

asset

`asset` 目录用于存放用于 `线上访问` 的静态资源。

通常构建工具会对 `src` 目录和 `dep` 目录下的资源进行分析、合并与压缩等，生成到 `asset` 目录下。所以该目录尽量避免手工管理。下面是一个构建工具生成后的 `asset` 目录示例：

```
1. ${root}/  
2.     asset/  
3.         js/  
4.             loader.js  
5.             build.js  
6.         css/
```

```

7.          common.css
8.          img/
9.          tpl/
10.         build.tpl.html
11.        img/
12.        ...

```

资源目录

按 **资源** 类型命名的目录称作 **资源目录**。**资源目录** 不允许(*MUST NOT*) 直接置于`${root}`下。

js

js 目录可用于存放 **js** 资源文件（包含可编译成 **js** 的 **coffeescript** 等语言）。**js** 文件后缀名 必须(*MUST*) 为 `.js`，**coffeescript**文件 后缀名 必须(*MUST*) 为 `.coffee`。

js 目录内 必须(*MUST*) 存放 **js** 资源文件，但 **js** 资源文件不一定 (*MAY NOT*) 存放于 **js** 目录下：

1. 对于 **src** 目录，**js** 资源文件 不允许(*MUST NOT*) 存放于 **js** 目录下。
2. 对于 **asset** 目录，**js** 资源文件 可以(*SHOULD*) 存放于 **js** 目录下，视构建行为决定。
3. 对于其他 **一级目录** 内，**js** 资源文件 可以(*SHOULD*) 不存放于 **js** 目录下。

CSS

css 目录可用于存放 **css资源文件**（包含 **less**，**sass** 等动态样式表语言）。**css** 文件后缀名 必须(*MUST*) 为 `.css`，**less** 文件后缀名 必须(*MUST*) 为 `.less`。

css 目录内 必须(*MUST*) 存放 **css** 资源文件，但 **css** 资源文件不一定 (*MAY NOT*) 存放于 **css** 目录下：

1. 对于 **src** 目录，**css** 资源文件 可以(*SHOULD*) 存放于 **业务目录** 下，也 可以(*SHOULD*) 存放于 **css** 目录下。
2. 对于 **asset** 目录，**css** 资源文件 可以(*SHOULD*) 存放于 **css** 目录下，视构建行为决定。
3. 对于其他 **一级目录** 内，**css** 资源文件 可以(*SHOULD*) 不存放于 **css** 目录下。

关于css引用图片的位置说明，请参考章节。

img

img 目录可用于存放 **图片资源文件**。包括 **页面直接引用** 的图片与 **css引用** 图片。常见的图片资源有 **gif/jpg/png/svg/bmp** 等。

对于 **css** 引用的图片， 必须(*MUST*) 放在 `./img` 目录下，`.` 代表当前 **css** 资源所在的目

录。

对于 `页面直接引用` 的图片：

1. 被多页面引用的图片 应该(*SHOULD*) 放在 `${root}/src/common/img` 目录下。
2. 单一页面引用的图片 应该(*SHOULD*) 放在 `./img` 目录下，`.` 代表当前页面所在的目录。

tpl

`tpl` 目录可用于存放 `template` 资源文件。`template` 资源文件后缀名 可以(*SHOULD*) 为 `.html` 或 `.tpl`。

通常，对于 `RIA` 系统，`template` 资源文件采用 `.html` 后缀使其能够被 `xhr` 加载。

font

`font` 目录可用于存放字体资源文件。常见的字体资源有 `tff/woff/svg` 等。

swf

`swf` 目录可用于存放 `flash` 资源文件。`flash` 资源文件 不允许(*MUST NOT*) 置于 `img` 目录中。

业务目录

common

`common` 目录为业务公共目录，用于存放业务项目的业务公共文件。所以，根据 `业务逻辑` 划分目录结构时，业务逻辑命名 不允许(*MUST NOT*) 为 `common`。

FAQ

为啥biz下面没资源类型目录了？

如果在 `biz` 下继续划分 `资源目录`，代码的结构可能就是这样子了：

```
1. ${root}/
2.   src/
3.     biz1/
4.       js/
5.         list.js
```

当我们需要使用 `list.js` 的时候，必须写如下的代码：`require("../biz1/js/list")`，但是从

逻辑上说，更合理的写法应该是 `require("../biz1/list")`。因此我们不推荐在 `biz` 下面对源代码资源划分目录。

图表库标准

百度图表库标准（1.0）

简介

图表在各种类型的产品中都有应用，本文档主要的设计目标是规范前端图表库的标准图表类型、接口、数据格式及样式设置，使之容易被理解、使用和维护。

同时，希望通过这次标准化，推动创建出可用的标准图表库，使之能更快捷地应用到各个项目中。

编撰

林志峰、赵庶、erik、刘阳、杨冬

本文档由 [百度Flash组](#) 与 [商业运营体系前端技术组](#) 联合审校发布。

要求

在本文档中，使用的关键字会以中文+括号包含的关键字英文表示： 必须(*MUST*)。关键字“MUST”，“MUST NOT”，“REQUIRED”，“SHALL”，“SHALL NOT”，“SHOULD”，“SHOULD NOT”，“RECOMMENDED”，“MAY”，and “OPTIONAL”被定义在rfc2119中。

名词解析

基本名词

名词	描述

chart	是指一个完整的图表，如折线图，饼图等“基本”图表类型或由基本图表组合而成的“混搭”图表，可能包括坐标轴、图例等
axis	直角坐标系中的一个坐标轴，坐标轴可分为类目轴和数值轴
xAxis	直角坐标系中的横轴，通常并默认为类目轴
yAxis	直角坐标系中的纵轴，通常并默认为数值轴
grid	直角坐标系中除坐标轴外的绘图网格
tooltip	提示框
legend	图例
series	数据系列

图表名词

名词	描述
line	折线图，堆积折线图，面积图，堆积面积图
bar	柱形图，堆积柱形图，条形图，堆积条形图
scatter	散点图，气泡图
pie	饼图，圆环图
radar	雷达图，填充雷达图

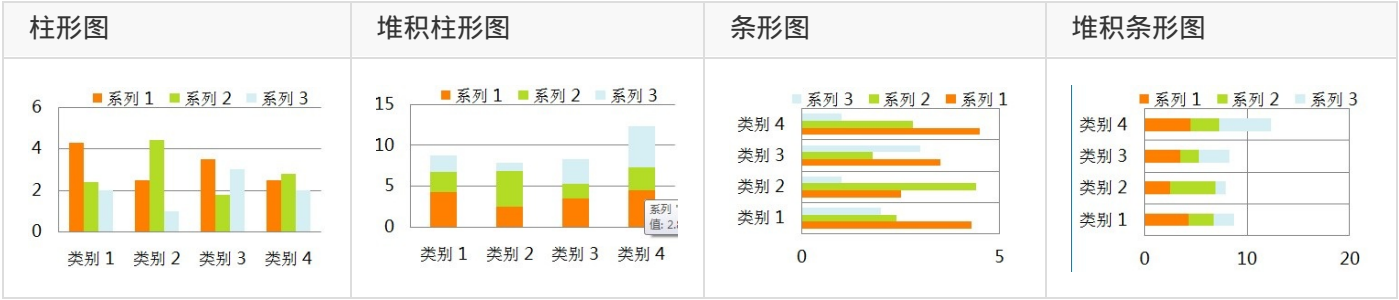
图表类型

图表库标准包含单图表类型的标准图表以及多图表类型混合的混搭图表：

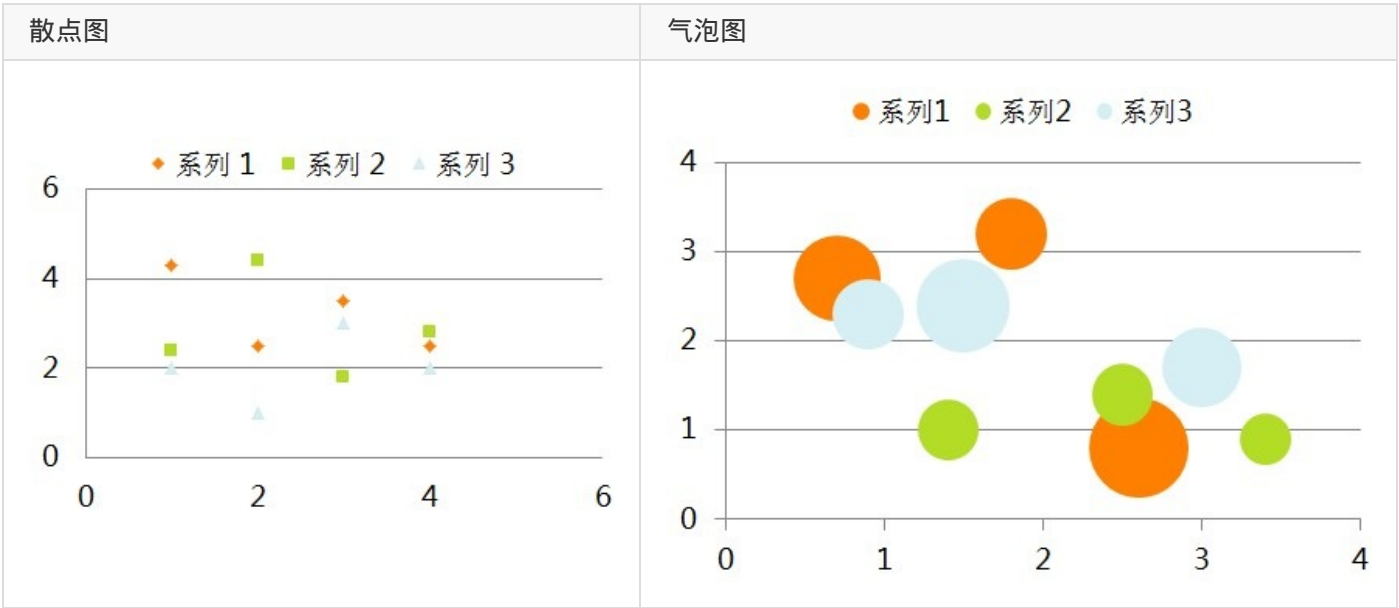




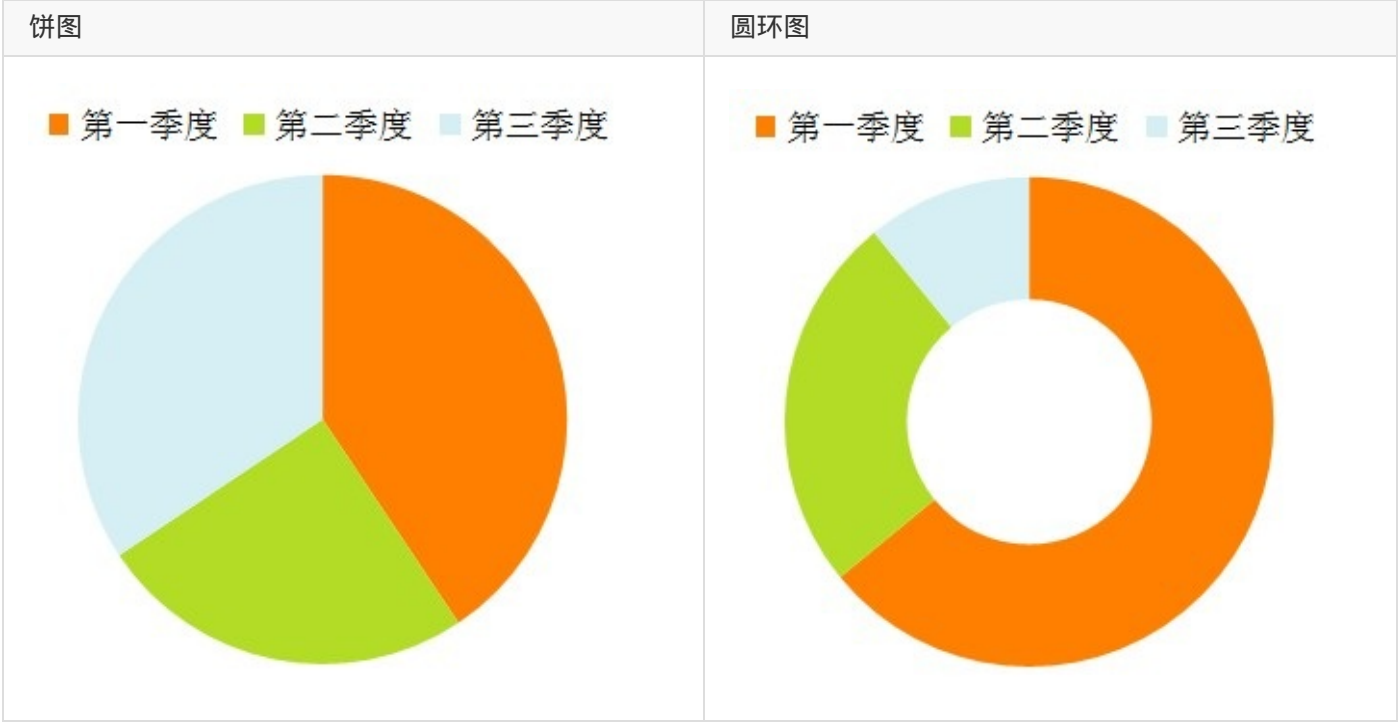
bar



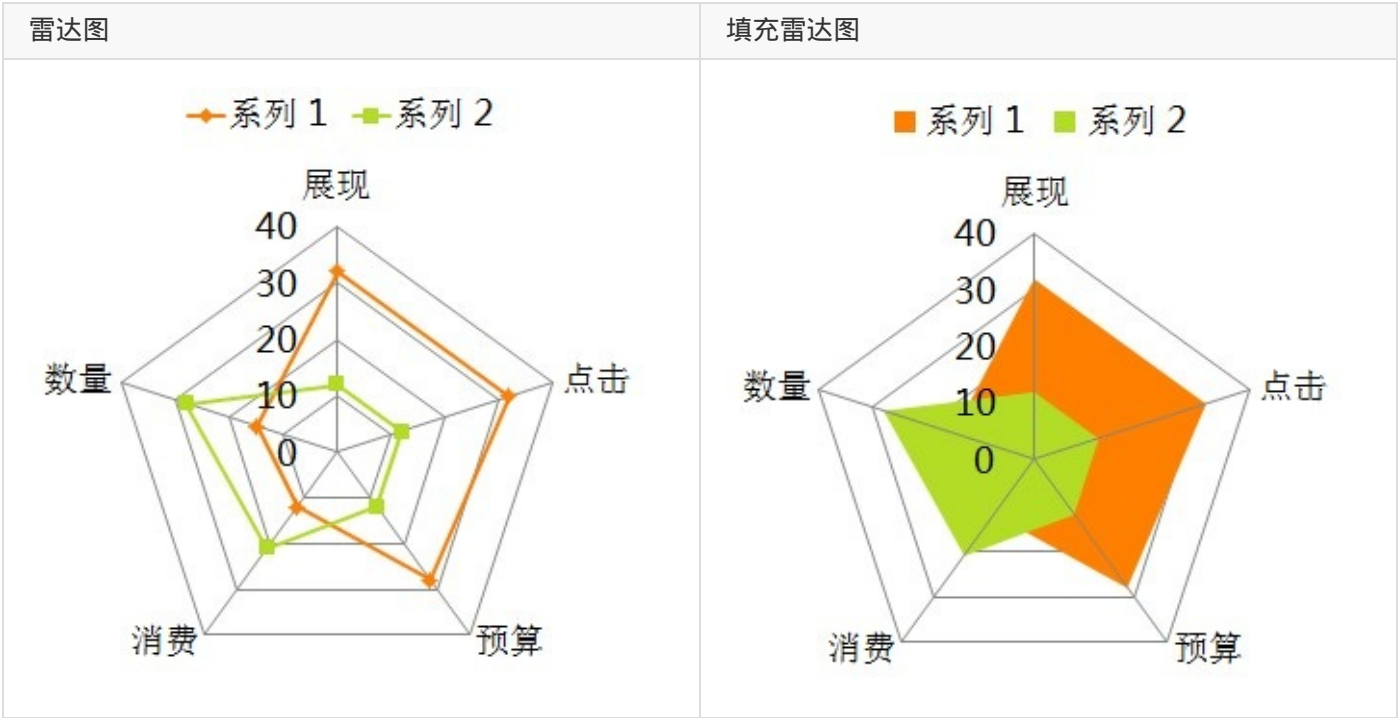
scatter



pie



radar



初始化

图表库实现 必须(MUST) 为多实例的，实例选项 应当(SHOULD) 在新建时传入，同时 可选(OPTIONAL) 的在实例新建后通过实例方法setOption(见方法) 传入，两种初始化方式最终产出效果 必须(MUST) 是等价的，即如下两组代码产出效果相同。

```
1. //初始化实例时传入选项
2. var myChart = new echarts(option);
3.
4. //初始化实例选项为空，通过实例方法传入选项
5. var myChart = new echarts();
6. myChart.setOption(option);
```

同时，在实例中任何个性化选项 不得(MUST NOT) 影响其他已存在或未来生成的实例。

方法

名称	描述
{void} setOption({Object} option)	万能接口，配置图表实例任何可配置选项（详见 option ），多次调用时option选项 必须(MUST) 是合并（merge）的
{void} setSeries({Array} series)	数据接口，驱动图表生成的数据内容（详见 series ），效果 应当(SHOULD) 等同调用setOption({series:{...}})
{void} on({string} eventName, {Function} eventListener)	事件绑定， 必须(MUST) 支持事件有：click, hover
{void} un({string} eventName, {Function} eventListener)	事件解绑定
{void} showLoading({Object} loadingOption)	过渡控制（详见 loadingOption ），显示loading（读取中）
{void} hideLoading({void})	过渡控制，隐藏loading（读取中）
{void} clear({void})	清空绘画内容，清空后实例可用
{void} dispose({void})	释放图表实例，释放后实例不再可用

选项

option

图表选项，包含图表实例任何可配置选项

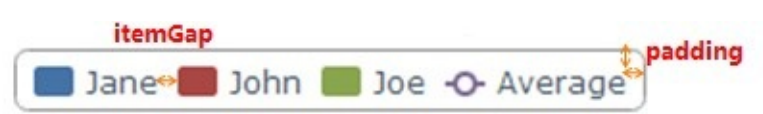
名称	描述
{Array} color	数值系列的颜色列表，默认为null则采用内置颜色，可配数组，eg: [' #87cefa ', ' rgba(123,123,123,0.5) ', '...'], 当系列数量个数比颜色列表长度大时将循环选取
{Object} legend	图例（详见 legend ），每个图表最多仅有一个图例，混搭图表共享
{Object} tooltip	提示框（详见 tooltip ），鼠标悬浮交互时的信息提示
{Object} grid	直角坐标系内绘图网格（详见 grid ）

{Array} xAxis	直角坐标系中横轴数组（详见 xAxis ），数组中每一项代表一条横轴坐标轴，标准（1.0）中规定最多同时存在2条横轴
{Array} yAxis	直角坐标系中纵轴数组（详见 yAxis ），数组中每一项代表一条纵轴坐标轴，标准（1.0）中规定最多同时存在2条纵轴
{Array} series	驱动图表生成的数据内容（详见 series ），数组中每一项代表一个系列的特殊选项及数据

legend

图例，每个图表最多仅有一个图例

名称	默认值	描述
{string} orient	'horizontal'	布局方式，默认为水平布局，可选为：'horizontal' 'vertical'
{string} {number} x	'center'	水平安放位置，默认为全图居中，可选为：'center' 'left' 'right' {number} (x坐标，单位px)
{string} {number} y	'top'	垂直安放位置，默认为全图顶端，可选为：'top' 'bottom' 'center' {number} (y坐标，单位px)
{string} backgroundColor	'#fff'	图例背景颜色
{string} borderColor	'#333'	图例边框颜色
{number} borderRadius	4	图例边框圆角，单位px，默认为4
{number} borderWidth	0	图例边框线宽，单位px，默认为0（无边框）
{number} {Array} padding	5	图例内边距，单位px，默认各方向内边距为5，接受数组分别设定上右下左边距，同css
{number} itemGap	10	各个item之间的间隔，单位px，默认为10，横向布局时为水平间隔，纵向布局时为纵向间隔
{string} itemRender	null	默认item渲染器（详见 series 中itemRender描述）
{Object} itemStyle	null	默认item渲染样式（详见 series 中itemStyle描述）
{Array} data	null	图例内容（详见 legend.data ，数组中每一项代表一个item



legend.data

图例内容数组，数组中每一项代表一个item，数组项可为{Object}，可以完整指定一个图例item的内容：

```

1.  [
2.    {
3.      name: 'xxx',           //图例名称
4.      itemRender: 'yyy',    //item渲染器，详见series中itemRender描述
5.      itemStyle: {...}      //item渲染样式，详见series中itemStyle
      描述
6.    }
7.    /*, {...}*/
8.  ]

```

当不指定itemRender或itemStyle时，则会根据name值索引series中同名name所用的itemRender或itemStyle，如果上述两项都不指定则数组项可退化为{string}，即

```

1.  [
2.    {
3.      name: 'xxx'
4.    },
5.    {
6.      name: 'zzz'
7.    }
8.  ]

```

和

```

1.  ['xxx', 'zzz']

```

是等价的。

当不指定itemRender或itemStyle，同时根据name值索引不到series中有同名name时，则会使用legend中itemRender或itemStyle，如果此时legend.itemRender或legend.itemStyle不存在则该项图例item将不被显示

tooltip

提示框, 鼠标悬浮交互时的信息提示


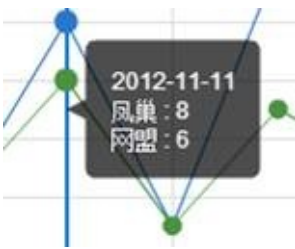
名称	默认值	描述
{string} trigger	'item'	触发类型，默认数据触发，见下图，可选为：'item' 'axis'
{boolean} show	true	显示策略，可选为：true（显示） false（隐藏）
{string} {Function} formatter	null	内容格式器：{string}（Template） {Function}，见表格下方
{string} backgroundColor	'rgba(0,0,0,0.7)'	提示背景颜色，默认为透明度为0.7的黑色
{string} borderColor	'#333'	提示边框颜色
{number} borderRadius	4	提示边框圆角，单位px，默认为4
{number} borderWidth	0	提示边框线宽，单位px，默认为0（无边框）
{number} {Array} padding	5	提示内边距，单位px，默认各方向内边距为5，接受数组分别设定上右下左边距，同css
{Object} textStyle	null	文本样式（详见 textStyle ）

内容格式器formatter：

- {string}，模板（Template），其变量为：

- {a} | {a0}
- {b} | {b0}
- {c} | {c0}
- {d} | {d0} （部分图表类型无此项）
- 多值下则存在多套{a1}, {b1}, {c1}, {d1}, {a2}, {b2}, {c2}, {d2}, ...
- 其中变量a、b、c在不同图表类型下代表数据含义为：
 - 折线（面积）图、柱状（条形）图、散点图：a（系列名称），b（横轴值），c（纵轴值），d（无）
 - 气泡图：a（系列名称），b（横轴值），c（纵轴值），d（数值）
 - 饼图、雷达图：a（系列名称），b（数据项名称），c（数值），d（百分比）
- {Function}, 传递参数为数组，数组项同模板变量：
 - [[a, b, c, d], [a1, b1, c1, d1], ...]

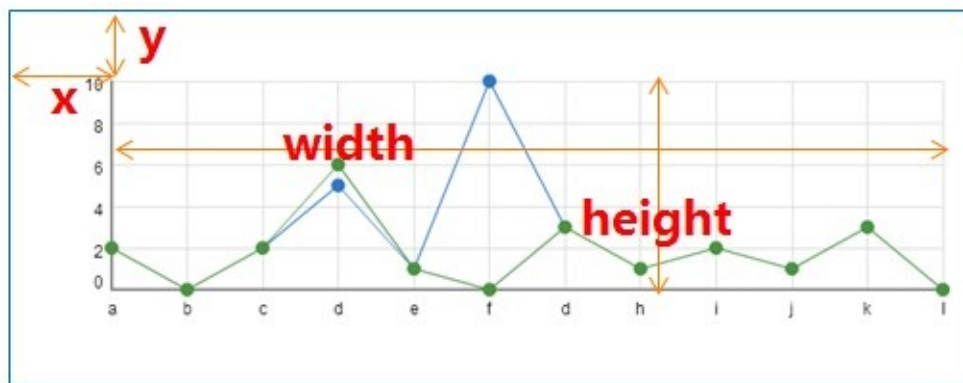
触发类型

item触发	axis触发
	

grid

直角坐标系内绘图网格

名称	默认值	描述
{number} x	80	直角坐标系内绘图网格起始横坐标，数值单位px
{number} y	40	直角坐标系内绘图网格起始纵坐标，数值单位px
{number} width	600	直角坐标系内绘图网格宽度，数值单位px
{number} height	250	直角坐标系内绘图网格高度，数值单位px



xAxis

直角坐标系中横轴数组，数组中每一项代表一条横轴坐标轴。

标准（1.0）中规定最多同时存在2条横轴，单条横轴时可指定安放于`grid`的底部（默认）或顶部，2条同时存在时则默认第一条安放于底部，第二天安放于顶部。

坐标轴有两种类型，类目型和数值型（区别详见`axis`），横轴通常为类目型，但条形图时则横轴为数值型，散点图时则横纵均为数值型，具体参数详见`axis`。

yAxis

直角坐标系中纵轴数组，数组中每一项代表一条纵轴坐标轴。

标准（1.0）中规定最多同时存在2条纵轴，单条纵轴时可指定安放于`grid`的左侧（默认）或右侧，2条同时存在时则默认第一条安放于左侧，第二天安放于右侧。

坐标轴有两种类型，类目型和数值型（区别详见`axis`），纵轴通常为数值型，但条形图时则纵轴为类目型，具体参数详见`axis`。

axis

坐标轴有两种类型，类目型和数值型，他们的区别在于：

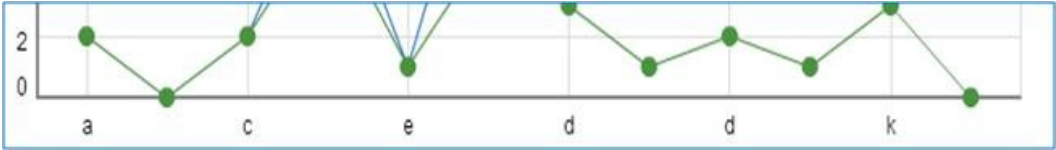
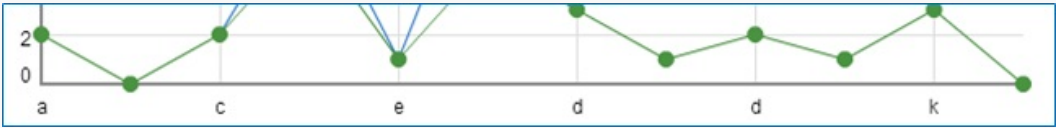
- 类目型：需要指定类目列表，坐标轴内有且仅有这些指定类目坐标
- 数值型：需要指定数值区间，坐标轴内包含数值区间内容全部坐标

下面是坐标轴的全部选项，其中个别选项仅在类目型或数值型时有效，请注意适用类型：

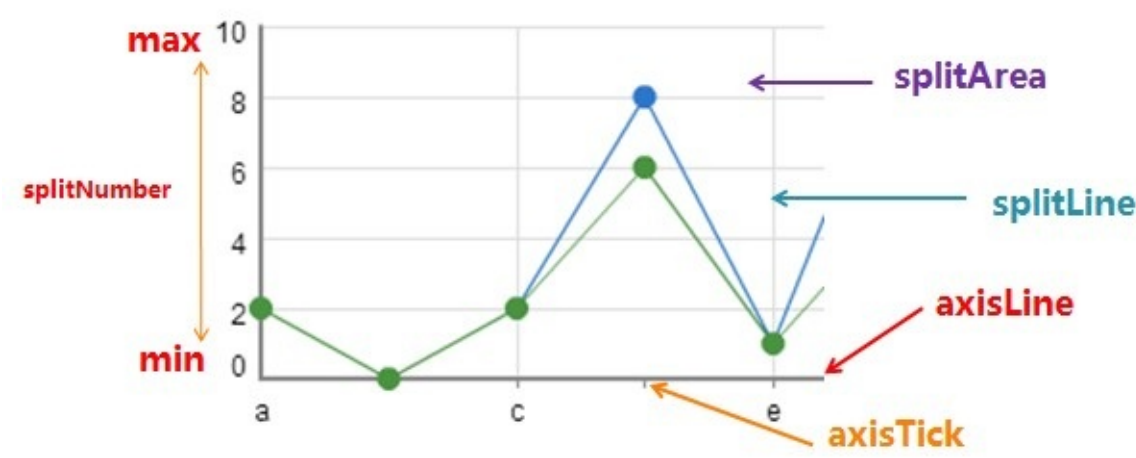
名称	默认值	适用类型	描述
{string} type	'category' 'value'	通用	坐标轴类型，横轴默认为类目型'category'，纵轴默认为数值型'value'
{string} position	'bottom' 'left'	通用	坐标轴类型，横轴默认为类目型'bottom'，纵轴默认为数值型'left'，可选为：'bottom' 'top' 'left' 'right'
{boolean}	true	类目	类目起始和结束两端空白策略，见下图，默认为true留空，false

		型	
{Array} boundaryGap	[0, 0]	数值型	数值轴两端空白策略，数组内数值代表百分比，[原始数据最小值与最终最小值之间的差额，原始数据最大值与最终最大值之间的差额]
{number} min	null	数值型	指定的最小值，eg：0，默认无，会自动根据具体数值调整，指定后将忽略boundaryGap[0]
{number} max	null	数值型	指定的最大值，eg：100，默认无，会自动根据具体数值调整，指定后将忽略boundaryGap[1]
{number} precision	0	数值型	小数精度，默认为0，无小数点
{number} power	100	数值型	整数精度，默认为100，个位和百位为0
{number} splitNumber	5	数值型	分割段数，默认为5
{Object} axisLine	{show : true}	通用	坐标轴线，默认显示，属性show控制显示与否，属性lineStyle（详见 lineStyle ）控制线条样式
{Object} axisTick	{show : false}	通用	坐标轴小标记，默认不显示，属性show控制显示与否，属性length控制线长，属性lineStyle（详见 lineStyle ）控制线条样式
{Object} axisLabel	{show : true}	通用	坐标轴文本标签，详见 axis.axisLabel
{Object} splitLine	{show : true}	通用	分隔线，默认显示，属性show控制显示与否，属性lineStyle（详见 lineStyle ）控制线条样式
{Object} splitArea	{show : false}	通用	分隔区域，默认不显示，属性show控制显示与否，属性areaStyle（详见 areaStyle ）控制区域样式
{Array} data	[]	类目型	类目列表，同时也是label内容，详见 axis.data

boundaryGap端空白策略

设置	效果
boundaryGap: true	
boundaryGap: false	

axis属性说明



axis.axisLabel

坐标轴文本标签选项

名称	默认值	适用类型	描述
{boolean} show	true	通用	是否显示，默认为true，设为false后下面都没意义了
{string} {number} interval	'auto'	类目型	标签显示挑选间隔，默认为'auto'，可选为：'auto'（自动隐藏显示不下的） 0（全部显示） {number}（用户指定选择间隔）
{number} rotate	0	通用	标签旋转角度，默认为0，不旋转，正直为逆时针，负值为顺时针，可选为：-90 ~ 90
{number} margin	2	通用	坐标轴文本标签与坐标轴的间距，默认为2
{string} {Function} formatter	null	通用	间隔名称格式器：{string}（Template） {Function}
{Object} textStyle	null	通用	文本样式（详见 textStyle ）

间隔名称格式器formatter：

- {string}，模板（Template），其变量为：
 - {value}：内容或值
- {Function}，传递参数同模板变量：
 - eg: function(value){return “星期” + “日一二三四五六”.charAt(value);}

axis.data

类目型坐标轴文本标签数组，指定label内容。

数组项通常为文本，如：

```
1. ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', ..., 'Dec']
```

当需要对个别标签进行个性化定义时，数组项可用对象，如：

```
1. [  
2.     'Jav', 'Feb', 'Mar',  
3.     {
```

被作为模板变量值或参数传入

```
5.         textStyle:{           //详见textStyle
6.             color : 'red'
7.             ...
8.         },
9.     },
10.     'May', '...'
11. ]
```

series

驱动图表生成的数据内容，数组中每一项代表一个系列的特殊选项及数据，其中个别选项仅在部分图表类型中有效，请注意适用类型：

名称	默认值	适用类型	描述
{string} name	''	通用	系列名称
{Object} tooltip	null	通用	提示框样式，仅对本系列有效，如不设则用 option.tooltip (详见 tooltip)，鼠标悬浮交互时的信息提示
{string} type	'line'	通用	图表类型，必要参数！可选为：折线图'line' 散点图'scatter' 柱状图'bar' 饼图'pie' 雷达图'radar'
{string} itemRender	null	通用	图形项渲染器
{Object} itemStyle	null	通用	图形样式 (详见 itemStyle)
{string} stack	null	折线图， 柱状图， 散点图	组合名称，做多组数据的堆积图时使用，eg: stack:'group1'，则series数组中stack值等于'group1'的数据做堆积计算
{number} xAxisIndex	0	折线图， 柱状图， 散点图	xAxis 坐标轴数组的索引，指定该系列数据所用的横坐标轴
{number} yAxisIndex	0	折线图， 柱状图， 散点图	yAxis 坐标轴数组的索引，指定该系列数据所用的纵坐标轴
{number} barMinHeight	20	柱状图	柱条最小高度，防止某item的值过小而影响交互
{number} barWidth	40	柱状图	柱条宽度
{Array} center	null	饼图	圆心坐标，默认无（为自适应居中）
{number} {Array} radius	100	饼图	半径，传数组实现环形图，[内半径，外半径]
{number} startAngle	0	饼图	开始角度，

minAngle	5	饼图	最小角度，避免显示过小
{Array} data	[]	通用	数据（详见 series.data ）

series.data

系列中的内容数组，折线图以及柱状图中当前数组长度 必须(MUST) 等于所使用类目轴文本标签数组 [axis.data](#)的长度，并且他们间是一一对应的，。

数组项通常为数值，如：

```
1.  [12, 34, 56, ..., 10, 23]
```

当某类目对应数据不存在（‘不存在’ != 0）时，可用‘-’表示，无数据在折线图中表现为折线在该点断开，在柱状图中表现为该点无柱形，如：

```
1.  [12, '-', 56, ..., 10, 23]
```

当需要对个别内容进行个性化定义时，数组项可用对象，如：

```
1.  [
2.      12, 34,
3.      {
4.          value : 56,
5.          tooltip:{},          //自定义特殊tooltip, 仅对该item有效, 详见tooltip
6.          itemRender:{},      //自定义特殊itemRender, 仅对该item有效, 同itemRender
7.          itemStyle:{        //自定义特殊itemStyle, 仅对该item有效, 详见
            itemStyle
8.        },
9.        ..., 10, 23
10. ]
```

特别的，当图表类型为scatter（散点图或气泡图）时，其数值设置比较特殊，他的横纵坐标轴都可能为数值型，并且气泡图时需要指定气泡大小，所以scatter型图表 应当(SHOULD) 设置为：

```
1.  [
2.      {
3.          value : [10, 25, 5]    //[xValue, yValue, rValue], 数组内依次为横值, 纵
            值, 大小
4.        },
5.        ...,
6.        {
7.          value : [30, 128, 15], //同上
```



```

8.      tooltip: {},                //自定义特殊tooltip, 仅对该item有效, 详见tooltip
9.      renderItem: {},            //自定义特殊itemRender, 仅对该item有效, 同
      renderItem
10.     itemStyle: {}              //自定义特殊itemStyle, 仅对该item有效, 详见
      itemStyle
11.   }
12. ]

```

再特别的, 当图表类型为饼图时, 需要说明每部分数据的名称name, 所以 应当(*SHOULD*) 设置为:

```

1.  [
2.    {
3.      value : 12,
4.      name : 'apple'           //每部分数据的名称
5.    },
6.    ...,
7.    {
8.      value : 23,              //同上
9.      name : 'orange'         //同上
10.     tooltip: {},             //自定义特殊tooltip, 仅对该item有效, 详见tooltip
11.     renderItem: {},          //自定义特殊itemRender, 仅对该item有效, 同
      renderItem
12.     itemStyle: {}            //自定义特殊itemStyle, 仅对该item有效, 详见
      itemStyle
13.   }
14. ]

```

itemStyle

图形样式, 可设置图表内图形的默认样式和强调样式(悬浮悬浮时样式):

```

1.  itemStyle: {
2.    normal: {
3.      ...
4.    },
5.    emphasis: {
6.      ...
7.    }
8.  }

```

其中normal和emphasis属性为对象, 其包含:

名称	默认值	适用类型	描述
{string} color	图表各异	通用	颜色
{Object} lineStyle	图表各异	折线图	线条样式，详见 lineStyle
{Object} areaStyle	图表各异	堆积折线图，柱状图，饼图，填充雷达图	区域样式，详见 areaStyle
{Object} label	{show: true, position:'outer'}	饼图	饼图标签，默认悬浮时显示
{Object} labelLine	{show: true}	饼图	饼图标签视觉引导线，当饼图标签位置（label.position）为'outer'时有效

其中饼图标签label属性为对象，其包含：

名称	默认值	描述
{boolean} show	true	标签显示策略，可选为：true（显示） false（隐藏）
{string} position	'outer'	标签显示位置，可选为：'outer'（外部） 'inner'（内部）
{Object} textStyle	null	标签的文本样式（详见 textStyle ）

其中饼图标签视觉引导线labelLine属性为对象，其包含：

名称	默认值	描述
{boolean} show	true	饼图标签视觉引导线显示策略，可选为：true（显示） false（隐藏）
{number} length	50	线长
{Object} lineStyle	各异	线条样式，详见 lineStyle

通过有效设置itemStyle的正常normal和emphasis选项可实现个性化的显示策略，比如希望饼图文字标签默认隐藏，并在鼠标悬浮时通过一条红色的视觉引导线显示在饼图外部区域，可以如下设置：

```
1.  itemStyle: {
2.      normal: {
3.          label: {
4.              show: false
5.          }
6.          labelLine: {
7.              show: false
8.          }
9.      },
10.     emphasis: {
11.         label: {
12.             show: true,
13.             position: 'outer'
14.         }
15.         labelLine: {
16.             show: true,
17.             lineStyle: {
18.                 color: 'red'
19.             }
20.         }
21.     }
22. }
```

lineStyle

线条（线段）样式

名称	描述
{string} color	颜色
{string} type	线条样式，可选为：‘solid’ ‘dotted’ ‘dashed’
{number} width	线宽

areaStyle

区域填充样式

名称	描述
{string} color	颜色
{string} type	填充样式，标准（1.0）目前仅支持'default'(实填充)

textStyle

文字样式

名称	描述
{string} color	颜色
{string} decoration	修饰，可选为：'none' 'underline'
{string} align	对齐方式，可选为：'left' 'right' 'center' 'justify'
{string} fontFamily	字体系列
{string} fontSize	字号
{string} fontStyle	样式，可选为：'normal' 'italic'
{string} fontWeight	粗细，可选为：'normal' 'bold' 'bolder' 'lighter'

loadingOption

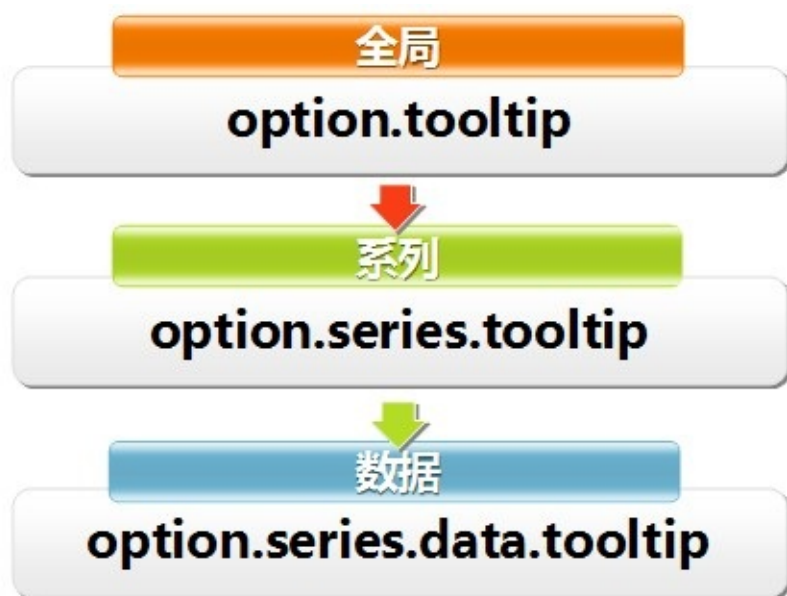
过渡显示，loading（读取中）的选项

名称	默认值	描述
{string} text	''	显示话术
{string} {number} x	'center'	水平安放位置，默认为全图居中，可选为：'center' 'left' 'right' {number} (x坐标，单位px)
{string} {number} y	'center'	垂直安放位置，默认为全图居中，可选为：'center' 'bottom' 'top' {number} (y坐标，单位px)
{Object} textStyle	null	显示话术的文本样式（详见 textStyle ）

多级控制设计

简单的说，你可以通过这三级满足不同level的定制和个性化需求：

- 通过 `option.*` 设置全局统一配置；
- 通过 `option.series.*` 设置特定系列特殊配置，其优先级高于 `option` 内的同名配置；
- 通过 `option.series.data.*` 设置特定数据项的特殊配置，最高优先级；



附录：一个直观的事例

```

1. // 图表实例化-----
2. var myChart = new echarts();
3.
4. // 图表使用-----
5. var option = {
6.     // 图例配置
7.     legend: {
8.         padding: 5, // 图例内边距，单位px，默认上下左右
           内边距为5
9.         itemGap: 10, // Legend各个item之间的间隔，横向
           布局时为水平间隔，纵向布局时为纵向间隔
10.        data: ['ios', 'android']
11.    },
12.
13.    // 气泡提示配置
14.    tooltip: {
15.        trigger: 'item', // 触发类型，默认数据触发，可选
           为:'axis'
  
```

```

16.     },
17.
18.     // 直角坐标系内绘图网格
19.     grid: {
20.         width: 500, // 直角坐标系内绘图网格宽度，数值单
           位px，并不包括坐标label
21.         height: 300 // 直角坐标系内绘图网格高度，数值单
           位px，并不包括坐标label
22.     },
23.
24.     // 直角坐标系中横轴数组
25.     xAxis: [
26.         {
27.             type: 'category', // 坐标轴类型，横轴默认为类目轴，数
           值轴则参考yAxis说明
28.             data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
           'Sep', 'Oct', 'Nov', 'Dec']
29.         }
30.     ],
31.
32.     // 直角坐标系中纵轴数组
33.     yAxis: [
34.         {
35.             type: 'value', // 坐标轴类型，纵轴默认为数值轴，类
           目轴则参考xAxis说明
36.             boundaryGap: [0.1, 0.1], // 坐标轴两端空白策略，数组内数值代
           表百分比
37.             splitNumber: 4 // 数值轴用，分割段数，默认为5
38.         }
39.     ]
40. };
41. myChart.setOption(option);
42.
43. //ajax getting data.....
44.
45. // 过渡-----
46. myChart.showLoading({
47.     text: '正在努力的读取数据中...', //loading话术
48.     x: 'center', //水平安放位置，默认为 'center', 可选
           为: 'left' || 'right' || Number可指定x坐标
49.     y: 'center' //垂直安放位置，默认为'center', 可选为:
           'top' || bottom' || Number可指定y坐标

```

```
50. });
51.
52. //ajax return
53. myChart.hideLoading();
54.
55. // 数据数组, data from ajax
56. var series = [
57.     {
58.         name: 'ios', // 系列名称
59.         type: 'line', // 图表类型, 折线图line、散点图
60.         data: [112, 23, 45, 56, 233, 343, 454, 89, 343, 123, 45, 123]
61.     },
62.     {
63.         name: 'android', // 系列名称
64.         type: 'line', // 图表类型, 折线图line、散点图
65.         data: [45, 123, 145, 526, 233, 343, 44, 829, 33, 123, 45, 13]
66.     }
67. ];
68.
69. myChart.setSeries(series);
70.
71. // 图表清空-----
72. myChart.clear();
73.
74. // 图表释放-----
75. myChart.dispose();
```

react编码规范

React规范

文件组织

- [强制]同一目录下不得拥有同名的 `.js` 和 `.jsx` 文件。

在使用模块导入时，倾向于不添加后缀，如果存在同名但不同后缀的文件，构建工具将无法决定哪一个是需要引入的模块。

- [强制]组件文件使用一致的 `.js` 或 `.jsx` 后缀。

所有组件文件的后缀名从 `.js` 或 `.jsx` 中任选其一。

不应在项目中出现部分组件为 `.js` 文件，部分为 `.jsx` 的情况。

- [强制]每一个文件以 `export default` 的形式暴露一个组件。

允许一个文件中存在多个不同的组件，但仅允许通过 `export default` 暴露一个组件，其它组件均定义为内部组件。

- [强制]每个存放组件的目录使用一个 `index.js` 以命名导出的形式暴露所有组件。

同目录内的组件相互引用使用 `import Foo from './Foo';` 进行。

引用其它目录的组件使用 `import {Foo} from '../component';` 进行。

建议使用VSCode的export-index插件等插件自动生成 `index.js` 的内容。

命名规则

- [强制]组件名为PascalCase。

包括函数组件，名称均为PascalCase。

- [强制]组件名称与文件名称保持相同。

同时组件名称应当能体现出组件的功能，以便通过观察文件名即确定使用哪一个组件。

- [强制]高阶组件使用camelCase命名。

高阶组件事实上并非一个组件，而是一个“生成组件类型”的函数，因此遵守JavaScript函数命

名的规范，使用camelCase命名。

- [强制]使用 `onXxx` 形式作为 `props` 中用于回调的属性名称。

使用统一的命名规则用以区分 `props` 中回调和非回调部分的属性，在JSX上可以清晰地看到一个组件向上和向下的逻辑交互。

对于不用于回调的函数类型的属性，使用动词作为属性名称。

```
1. // onClick作为回调以on开头，renderText非回调函数则使用动词
2. let Label = ({onClick, renderText}) => <span onClick={onClick}>
    {renderText()}</span>;
```

- [建议]使用 `withXxx` 或 `xxxable` 形式的词作为高阶组件的名称。

高阶组件是为组件添加行为和功能的函数，因此使用如上形式的词有助于对其功能进行理解。

- [建议]作为组件方法的事件处理函数以具备业务含义的词作为名称，不使用 `onXxx` 形式命名。

```
1. // Good
2. class Form {
3.   @autobind
4.   collectAndSubmitData() {
5.     let data = {
6.       name: this.state.name,
7.       age: this.state.age
8.     };
9.     this.props.onSubmit(data);
10.  }
11.
12.   @autobind
13.   syncName() {
14.     // ...
15.   }
16.
17.   @autobind
18.   syncAge() {
19.     // ...
20.   }
21.
22.   render() {
23.     return (
24.       <div>
```

```

25.             <label>姓名:<input type="text" onChange={this.syncName}
                /></label>
26.             <label>年龄:<input type="number" onChange={this.syncAge}
                /></label>
27.             <button type="button" onClick={this.collectAndSubmit}>提
                交</button>
28.         </div>
29.     );
30. }
31. }

```

组件声明

- [强制]使用ES Class声明组件，禁止使用 `React.createClass`。

`React v15.5.0`已经弃用了 `React.createClass` 函数。

```

1.  // Bad
2.  let Message = React.createClass({
3.      render() {
4.          return <span>{this.state.message}</span>;
5.      }
6.  });
7.
8.  // Good
9.  class Message extends PureComponent {
10.      render() {
11.          return <span>{this.state.message}</span>;
12.      }
13.  }

```

- [强制]不使用 `state` 的组件声明为函数组件。

函数组件在React中有着特殊的地位，在将来也有可能得到更多的内部优化。

```

1.  // Bad
2.  class NextNumber {
3.      render() {
4.          return <span>{this.props.value + 1}</span>
5.      }
6.  }
7.

```

```
8.    // Good
9.    let NextNumber = ({value}) => <span>{value + 1}</span>;
```

- [强制]所有组件均需声明 `propTypes`。

`propTypes` 在提升组件健壮性的同时，也是一种类似组件的文档的存在，有助于代码的阅读和理解。

- [强制]对于所有非 `isRequired` 的属性，在 `defaultProps` 中声明对应的值。

声明初始值有助于对组件初始状态的理解，也可以减少 `propTypes` 对类型进行校验产生的开销。

对于初始没有值的属性，应当声明初始值为 `null` 而非 `undefined`。

- [强制]如无必要，使用静态属性语法声明 `propTypes`、`contextTypes`、`defaultProps` 和 `state`。

仅当初始 `state` 需要从 `props` 计算得到的时候，才将 `state` 的声明放在构造函数中，其它情况下均使用静态属性声明进行。

- [强制]依照规定顺序编排组件中的方法和属性。

按照以下顺序编排组件中的方法和属性：

- i. `static displayName`
- ii. `static propTypes`
- iii. `static contextTypes`
- iv. `static defaultProps`
- v. `static state`
- vi. 其它静态的属性
- vii. 用于事件处理并且以属性的方式 (`onClick = e => {...}`) 声明的方法
- viii. 其它实例属性
- ix. `constructor`
- x. `getChildContext`
- xi. `componentWillMount`
- xii. `componentDidMount`
- xiii. `shouldComponentUpdate`
- xiv. `componentWillUpdate`
- xv. `componentDidUpdate`
- xvi. `componentWillUnmount`
- xvii. 事件处理方法
- xviii. 其它方法

xix. `render`

其中 `shouldComponentUpdate` 和 `render` 是一个组件最容易被阅读的函数，因此放在最下方有助于快速定位。

- [建议]无需显式引入React对象。

使用JSX隐式地依赖当前环境下有 `React` 这一对象，但在源码上并没有显式使用，这种情况下添加 `import React from 'react';` 会造成一个没有使用的变量存在。

使用**babel-plugin-react-require**插件可以很好地解决这一问题，因此无需显式地编写 `import React from 'react';` 这一语句。

- [建议]使用箭头函数声明函数组件。

箭头函数具备更简洁的语法（无需 `function` 关键字），且可以在仅有一个语句时省去 `return` 造成的额外缩进。

- [建议]高阶组件返回新的组件类型时，添加 `displayName` 属性。

同时在 `displayName` 上声明高阶组件的存在。

```

1.  // Good
2.  let asPureComponent = Component => {
3.    let componentName = Component.displayName || Component.name ||
      'UnknownComponent';
4.    return class extends PureComponent {
5.      static displayName = `asPure(${componentName})`
6.
7.      render() {
8.        return <Component {...this.props} />;
9.      }
10.    };
11.  };

```

组件实现

- [强制]除顶层或路由级组件以外，所有组件均在概念上实现为纯组件（Pure Component）。

本条规则并非要求组件继承自 `PureComponent`，“概念上的纯组件”的意思为一个组件在 `props` 和 `state` 没有变化（`shallowEqual`）的情况下，渲染的结果应保持一致，即 `shouldComponentUpdate` 应当返回 `false`。

一个典型的非纯组件是使用了随机数或日期等函数：

```
1. let RandomNumber = () => <span>{Math.random()}</span>;
2. let Clock = () => <span>{Date.time()}</span>;
```

非纯组件具备向上的“传染性”，即一个包含非纯组件的组件也必须是非纯组件，依次沿组件树结构向上。由于非纯组件无法通过 `shouldComponentUpdate` 优化渲染性能且具备传染性，因此要避免在非顶层或路由组件中使用。

如果需要在组件树的某个节点使用随机数、日期等非纯的数据，应当由顶层组件生成这个值并通过 `props` 传递下来。对于使用Redux等应用状态管理的系统，可以在应用状态中存放相关值（如Redux使用Action Creator生成这些值并通过Action和reducer更新到store中）。

- [强制] 禁止为继承自 `PureComponent` 的组件编写 `shouldComponentUpdate` 实现。

参考[React的相关Issue](#)，在React的实现中，`PureComponent` 并不直接实现 `shouldComponentUpdate`，而是添加一个 `isReactPureComponent` 的标记，由 `CompositeComponent` 通过识别这个标记实现相关的逻辑。因此在 `PureComponent` 上自定义 `shouldComponentUpdate` 并无法享受 `super.shouldComponentUpdate` 的逻辑复用，也会使得这个继承关系失去意义。

- [强制] 为非继承自 `PureComponent` 的纯组件实现 `shouldComponentUpdate` 方法。

`shouldComponentUpdate` 方法在React的性能中扮演着至关重要的角色，纯组件必定能通过 `props` 和 `state` 的变化来决定是否进行渲染，因此如果组件为纯组件且不继承 `shouldComponentUpdate`，则应当有自己的 `shouldComponentUpdate` 实现来减少不必要的渲染。

- [建议] 为函数组件添加 `PureComponent` 能力。

函数组件并非一定是纯组件，因此其 `shouldComponentUpdate` 的实现为 `return true;`，这可能导致额外的无意义渲染，因此推荐使用高阶组件为其添加 `shouldComponentUpdate` 的相关逻辑。

推荐使用[react-pure-stateless-component](#)库实现这一功能。

- [建议] 使用 `@autobind` 进行事件处理方法与 `this` 的绑定。

由于 `PureComponent` 使用 `shallowEqual` 进行是否渲染的判断，如果在JSX中使用 `bind` 或箭头函数绑定 `this` 会造成子组件每次获取的函数都是一个新的引用，这破坏了 `shouldComponentUpdate` 的逻辑，引入了无意义的重复渲染，因此需要在 `render` 调用之前就將事件处理方法与 `this` 绑定，在每次 `render`` 调用中获取同样的引用。

当前比较流行的事前绑定 `this` 的方法有2种，其一使用类属性的语法：

```

1.   class Foo {
2.       onClick = e => {
3.           // ...
4.       }
5.   };

```

其二使用 `@autobind` 的装饰器：

```

1.   class Foo {
2.       @autobind
3.       onClick(e) {
4.           // ...
5.       }
6.   }

```

使用类属性语法虽然可以避免引入一个 `autobind` 的实现，但存在一定的缺陷：

- i. 对于新手不容易理解函数内的 `this` 的定义。
- ii. 无法在函数是使用其它的装饰器（如 `memoize`、`deprecated` 或检验相关的逻辑等）。

因此，推荐使用 `@autobind` 装饰器实现 `this` 的事先绑定，推荐使用 `core-decorators` 库提供的相关装饰器实现。

JSX

- [强制] 没有子节点的非DOM组件使用自闭合语法。

对于DOM节点，按照HTML编码规范相关规则进行闭合，其中 `void element` 使用自闭合语法。

```

1.   // Bad
2.   <Foo></Foo>
3.
4.   // Good
5.   <Foo />

```

- [强制] 保持起始和结束标签在同一层缩进。

对于标签前面有其它语句（如 `return` 的情况，使用括号进行换行和缩进）。

```

1.   // Bad

```

```

2.   class Message {
3.       render() {
4.           return <div>
5.               <span>Hello World</span>
6.           </div>;
7.       }
8.   }
9.
10.  // Good
11.  class Message {
12.      render() {
13.          return (
14.              <div>
15.                  <span>Hello World</span>
16.              </div>;
17.          );
18.      }
19.  }

```

对于直接 `return` 的函数组件，可以直接使用括号而省去大括号和 `return` 关键字：

```

1.  let Message = () => (
2.      <div>
3.          <span>Hello World</span>
4.      </div>
5.  );

```

- [强制]对于多属性需要换行，从第一个属性开始，每个属性一行。

```

1.  // 没有子节点
2.  <SomeComponent
3.      longProp={longProp}
4.      anotherLongProp={anotherLongProp}
5.  />
6.
7.  // 有子节点
8.  <SomeComponent
9.      longProp={longProp}
10.     anotherLongProp={anotherLongProp}
11.  >
12.     <SomeChild />
13.     <SomeChild />

```

```
14.    </SomeComponent>
```

- [强制]以字符串字面量作为值的属性使用双引号 (`"`)，在其它类型表达式中的字符串使用单引号 (`'`)。

```
1.    // Bad
2.    <Foo bar='bar' />
3.    <Foo style={{width: "20px"}} />
4.
5.    // Good
6.    <Foo bar="bar" />
7.    <Foo style={{width: '20px'}} />
```

- [强制]自闭合标签的 `</>` 前添加一个空格。

```
1.    // Bad
2.    <Foo bar="bar"/>
3.    <Foo bar="bar" />
4.
5.    // Good
6.    <Foo bar="bar" />
```

- [强制]对于值为 `true` 的属性，省去值部分。

```
1.    // Bad
2.    <Foo visible={true} />
3.
4.    // Good
5.    <Foo visible />
```

- [强制]对于需要使用 `key` 的场合，提供一个唯一标识作为 `key` 属性的值，禁止使用可能会变化的属性（如索引）。

`key` 属性是React在进行列表更新时的重要属性，如该属性会发生变化，渲染的性能和正确性都无法得到保证。

```
1.    // Bad
2.    {list.map((item, index) => <Foo key={index} {...item} />)}
3.
4.    // Good
5.    {list.map(item => <Foo key={item.id} {...item} />)}
```

- [建议]避免在JSX的属性值中直接使用对象和函数表达式。

`PureComponent` 使用 `shallowEqual` 对 `props` 和 `state` 进行比较来决定是否需要渲染，而在JSX的属性值中使用对象、函数表达式会造成每一次的对象引用不同，从而 `shallowEqual` 会返回 `false`，导致不必要的渲染。

```

1.  ```javascript
2.  // Bad
3.  class WarnButton {
4.      alertMessage(message) {
5.          alert(message);
6.      }
7.
8.      render() {
9.          return <button type="button" onClick={() =>
10.             this.alertMessage(this.props.message)}>提示</button>
11.      }
12.
13.  // Good
14.  class WarnButton {
15.      @autobind
16.      alertMessage() {
17.          alert(this.props.message);
18.      }
19.
20.      render() {
21.          return <button type="button" onClick={this.alertMessage}>提示</button>
22.      }
23.  }
24.  ```

```

- [建议]将JSX的层级控制在3层以内。

JSX提供了基于组件的便携的复用形式，因此可以通过将结构中的一部分封装为一个函数组件来很好地拆分大型复杂的结构。层次过深的结构会带来过多缩进、可读性下降等缺点。如同控制函数内代码行数和分支层级一样，对JSX的层级进行控制可以有效提升代码的可维护性。

```

1.  // Bad
2.  let List = ({items}) => (
3.      <ul>
4.          {

```

```
5.         items.map(item => (  
6.             <li>  
7.                 <header>  
8.                     <h3>{item.title}</h3>  
9.                     <span>{item.subtitle}</span>  
10.                 </header>  
11.                 <section>{item.content}</section>  
12.                 <footer>  
13.                     <span>{item.author}</span>@<time>{item.postTime}  
14.                 </time>  
15.                 </footer>  
16.             </li>  
17.         ))  
18.     }  
19. </ul>  
20. );  
21. // Good  
22. let Header = ({title, subtitle}) => (  
23.     <header>  
24.         <h3>{title}</h3>  
25.         <span>{subtitle}</span>  
26.     </header>  
27. );  
28.  
29. let Content = ({content}) => <section>{content}</section>;  
30.  
31. let Footer = ({author, postTime}) => (  
32.     <footer>  
33.         <span>{author}</span>@<time>{postTime}</time>  
34.     </footer>  
35. );  
36.  
37. let Item = item => (  
38.     <div>  
39.         <Header {...item} />  
40.         <Content {...item} />  
41.         <Footer {...item} />  
42.     </div>  
43. );  
44.  
45. let List = ({items}) => (  

```

```
46.      <ul>
47.          {items.map(Item)}
48.      </ul>
49.  );
```