概述

关于

前端代码规范是基于W3C等官方文档，并结合团队日常业务需求以及团队在日常开发过程中总结提炼出的经验而制定。
旨在增强团队开发协作、提高代码质量和打造开发基石的编码规范。

TSA前端代码现状

TSA前端代码，引用深蓝平台1.3.5版本，采用的框架是Angularjs，UI采用semantic(semantic基于Jquery)+深蓝平台(深蓝平台是对semantic的二次封装和通用组件的封装)，js工具库采用lodash，接口请求模拟采用mockjs，样式采用的stylus，打包工具webpack

```
框架：Angularjs
UI：深蓝平台+Semantic
js库：lodash
请求模拟：mockjs
Style：stylus
打包工具：webpack
```

TSA前端框架和UI改进

TSA框架由Angularjs转为React，引入了dev控件库作为新的UI框架，保留原有代码的基础上进行的更新，所以老代码的维护还是采用以前的形式，新功能的开发，均采用新框架和新UI。

特殊符号含义

本文档与bfc的前端规范文档进行整合，特殊的规范将以如下符号标识：

```
-bfc：bfc规范文档提出的规范
-tsap：bfc和tsa的规范文档所共有的规范
!：重点规范，不遵守可能造成代码错误的规范
```

以下规范在持续更新中：

- 命名规范
- HTML规范
- CSS规范
- JavaScript规范

命名规范

由历史原因及个人习惯引起的命名不统一，导致不同成员在维护同一页面时，效率低下，迭代、维护成本极高，故统一命名标准。

目录命名

目录文件夹命名以见名知意为原则，区分不同功能，减少文件夹的深度嵌套，降低单个文件的代码量，因为冗长的代码可读性低。

```
build：打包配置文件夹
config：全局配置文件夹
dist：打包完成文件夹
js：控制器文件夹
public：公共文件夹
  assets：静态资源文件夹
    iconfont：图标文件夹，建议采用svg格式的图标，只用引用一个js文件
    images：图片文件夹
    styles：样式文件夹
  business：项目指令文件夹
  common：公用服务及指令文件夹
    directives：  指令文件夹
    filters：过滤器文件夹
    services：服务文件夹
  config：配置文件夹
  mock：接口代理文件夹
  componet：组件文件夹，建议将弹窗等可复用的功能拆分为组件
  store：存储文件夹
  util：工具文件夹
  visualize：图表文件夹
views：HTML文件夹
```

图片命名

分功能命名文件夹，如login，多个英文名之间用中划线-间隔，如login-bg

HTML/CSS文件命名

确保文件命名总是以字母开头而不是数字，且字母一律小写，以下划线连接且不带其他标点符号，如：

```
<!-- HTML -->
jdc.html
jdc_list.html
jdc_detail.html

<!-- SASS -->
jdc.scss
jdc_list.scss
jdc_detail.scss
```

HTML规范

HTML遵循浏览器解析正确，可读性强，便于维护的原则

代码规范

元素及标签闭合

HTML元素共有以下5种：

- 空元素：area、base、br、col、command、embed、hr、img、input、keygen、link、meta、param、source、track、wbr。
- 原始文本元素：script、style。
- RCDATA元素：textarea、title。
- 外来元素：来自MathML命名空间和SVG命名空间的元素。
- 常规元素：其他HTML允许的元素都称为常规元素。

五类元素的区别如下：

1. 空元素，不能容纳任何内容（因为它们没有闭合标签，没有内容能够放在开始标签和闭合标签中间）。
2. 原始文本元素，可以容纳文本。
3. RCDATA元素，可以容纳文本和字符引用。
4. 外部元素，可以容纳文本、字符引用、CDATA段、其他元素和注释。
5. 基本元素，可以容纳文本、字符引用、其他元素和注释。

元素标签的闭合应遵循以下原则：

- 原始文本元素、RCDATA元素以及常规元素都有一个开始标签来表示开始，一个结束标签来表示结束。
- 某些元素的开始和结束标签是可以省略的，如果规定标签不能被省略，那么就绝对不能省略它。
- 空元素只有一个开始标签，且不能为空元素设置结束标签。
- 外来元素可以有一个开始标签和配对的结束标签，或者只有一个自闭合的开始标签，且后者情况下该元素不能有结束标签。

为了能让浏览器更好的解析代码以及能让代码具有更好的可读性，有如下约定：

- 所有具有开始标签和结束标签的元素都要写上起止标签，某些允许省略开始标签或和束标签的元素亦都要写上。
- 空元素标签都不加 “/” 字符。

推荐：

```
<div>
    <h1>我是h1标题</h1>
    <p>我是一段文字，我有始有终，浏览器能正确解析</p>
</div>

<br>
```

不推荐：

```
<div>
    <h1>我是h1标题</h1>
    <p>我是一段文字，我有始无终，浏览器亦能正确解析
</div>

<br/>
```

书写风格

HTML代码大小写

HTML标签名、类名、标签属性和大部分属性值统一用小写，自定义组件用中划线"-"分割。
推荐：

```
<div class="demo"></div>
<nav-bar nav-data="data"></nav-bar>
```

不推荐：

```
<div class="DEMO"></div>

<DIV CLASS="DEMO"></DIV>

<navBar navData="data"></navBar>
```

HTML文本、CDATA、JavaScript、meta标签某些属性等内容可大小写混合。

```
<!-- 优先使用 IE 最新版本和 Chrome Frame -->
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1"/>

<!-- HTML文本内容 -->
<h1>I AM WHAT I AM </h1>

<!-- JavaScript 内容 -->
<script type="text/javascript">
        var demoName = 'demoName';
        ...
</script>

<!-- CDATA 内容 -->
<script type="text/javascript"><![CDATA[
...
]]></script>
```

元素属性

元素属性值使用双引号语法。

特殊字符引用

在 HTML 中不能使用小于号 “<” 和大于号 “>”特殊字符，浏览器会将它们作为标签解析，若要正确显示，在 HTML 源代码中使用字符实体。
推荐：

```
<a href="#">more&gt;&gt;</a>
```

不推荐：

```
<a href="#">more>></a>
```

代码缩进

统一使用四个空格进行代码缩进，使得各编辑器表现一致（各编辑器有相关配置）。

代码嵌套

元素嵌套规范，每个块状元素独立一行，内联元素可选。
推荐：

```
<div>
    <h1></h1>
    <p></p>
</div>
<p><span></span><span></span></p>
```

不推荐：

```
<div>
    <h1></h1><p></p>
</div>
<p>
    <span></span>
    <span></span>
</p>
```

段落元素与标题元素只能嵌套内联元素。
推荐：

```
<h1><span></span></h1>
<p><span></span><span></span></p>
```

不推荐：

```
<h1><div></div></h1>
<p><div></div><div></div></p>
```

CSS规范

CSS遵循可读性强，便于维护的原则

代码规范

样式表编码!

样式文件必须写上 @charset 规则，并且一定要在样式文件的第一行首个字符位置开始写，编码名用“UTF-8”。

```
@charset "UTF-8";
```

代码风格

代码格式化

样式书写一般有两种：一种是紧凑格式 (Compact)， 一种是展开格式（Expanded），统一使用展开格式书写样式

```
/* 紧凑 */
.nav{ display: block;width: 50px;}


/* 展开 */
.nav {
    display: block;
    width: 50px;
}
```

代码大小写-tsap

样式选择器，属性名，属性值关键字全部使用小写字母书写，并使用“-”连接。

```
/* 不推荐 */
.navList{
        DISPLAY:BLOCK;
}

/* 推荐 */
.nav-list {
        display:block;
}
```

常用方式-bfc

CSS样式常用的为三种方式：内联（行间）、内嵌、外链

```
<!-- 推荐 外链接式 -->
<head>
  <link rel="stylesheet" type="text/css" href="css/style.css" />
</head>

<!-- 内嵌式 适用于小页面 -->
<head>
  <style type="text/css">
    html,body{
                margin: 0;
                padding: 0;
        }
  </style>
</head>

 <!-- 不推荐 行内样式，优先级太高、HTML不纯净 -->
 <p style="color: red;">我是P段落</p>

 <!-- 不推荐 导入式，相比link太慢 -->
<head>
  <style type="text/css">
    @import url("css/style.css");
  </style>
</head>
```

- 推荐使用类选择器，少用ID选择器!
  ID在一个页面中的唯一性导致了如果以ID为选择器来写CSS，就无法重用。

选择器-tsap

- 尽量少用通用选择器 *
- 不使用 ID 选择器
- 不使用无具体语义定义的标签选择器
- 尽量不使用!important

```
/* 推荐 */
.nav {}
.nav li {}
.nav li p{}

/* 不推荐 */
*{}
#nav {}
.nav div{}
```

代码缩进

统一使用四个空格进行代码缩进，使得各编辑器表现一致（各编辑器有相关配置）。

```
.nav {
    width: 100%;
    height: 100%;
}
```

分号-tsap

每个属性声明末尾都要加分号；

```
.nav {
    width: 100%;
    height: 100%;
}
```

代码易读性

- 左括号与类名之间一个空格，冒号与属性值之间一个空格
- 逗号分隔的取值，逗号之后一个空格
- 为单个css选择器或新申明开启新行
- 颜色值 rgb() rgba() hsl() hsla() rect() 中不需有空格，且取值不要带有不必要的 0
- 属性值十六进制数值能用简写的尽量用简写
- 不要为 0 指明单位-tsap

```
/* 不推荐 */
.element {
background-color: rgba(25, 25, 25, 0.6);
    transition: linear 0.8s;
}

/* 推荐 */
.element {
    background-color: rgba(25, 25, 25, .6);
        transition: linear .8s;
}

/* 不推荐 */
.element {
        margin: 0rem;
        border-width: 0px;
}

/* 推荐 */
.element {
        margin: 0;
        border-width: 0;
}
```

属性值引号

css属性值需要用到引号时，统一使用单引号

```
/* 推荐 */
.nav {
        font-family: 'Hiragino Sans GB';
}

/* 不推荐 */
.nav {
        font-family: "Hiragino Sans GB";
}
```

属性书写顺序-tsap

建议遵循以下顺序：

1. 布局定位属性：display / position / float / clear / visibility / overflow
2. 自身属性：width / height / margin / padding / border / background
3. 文本属性：color / font / text-decoration / text-align / vertical-align / white- space / break-word
4. 其他属性（CSS3）：content / cursor / border-radius / box-shadow / text-shadow / background:linear-gradient …

5. CSS3 浏览器私有前缀在前，标准前缀在后

```
.nav {
    display: block;
    position: relative;
    float: left;
    width: 100px;
    height: 100px;
    margin: 0 10px;
    padding: 20px 0;
    font-family: Arial, 'Helvetica Neue', Helvetica, sans-serif;
    color: #333;
    background: rgba(0,0,0,.5);
    -webkit-border-radius: 10px;
    -moz-border-radius: 10px;
    -o-border-radius: 10px;
    -ms-border-radius: 10px;
    border-radius: 10px;
}
```

换行-bfc

对于组合的选择器，各选择器单独占一行，利于阅读和修改

```
/* 不推荐 */
.item, .item-secondary, .item[type=text] {
  padding: 15px;
  margin: 0px 0px 15px;
}
/* 如果需要中途修改选择器，可能会有漏改 */
.menu .item, .menu .item-secondary, .item[type=text] {
  padding: 15px;
  margin: 0px 0px 15px;
}

/* 推荐 */
.menu .item,
.menu .item-secondary,
.menu .item[type="text"] {
  padding: 15px;
  margin-bottom: 15px;
}
```

对于只包含一条声明的样式，为了易读性和便于快速编辑，可以将语句放在同一行。花括号{}内两侧都需要一个空格。

```
/* 临近的样式花括号可对齐 */
.icon                  { background-position: 0 0; }
.icon-home             { background-position: 0 -20px; }
.icon-account   { background-position: 0 -40px; }
```

简写形式的属性声明-bfc

需要显示地设置所有值的情况下，尽量使用简写形式的属性声明。如果只需要指定部分属性，不建议使用简写形式。过度使用简写形式会导致代码混乱，并且会对属性带来不必要的覆盖，或许会引起意外的副作用。
[padding，margin，background，border，border-radius]

```
/* 简写形式设置所有属性 */
.element {
        margin: 0 0 10px;
        border: 1px solid #333;
        background: #00FF00 url(bgimage.gif) no-repeat fixed top;
        border-radius: 3px 3px 0 0;
}

/* 设置部分属性 */
.element {
        margin-bottom: 10px;
        border-color: #666;
        border-top-left-radius: 3px;
        border-top-right-radius: 3px;
        background-image: url("image.jpg");
}
```

后代选择器命名-bfc

后代选择器不需要完整表现结构树层级，尽量简短。
通过使用后代选择器的方法，你不需要考虑他的命名是否已被使用，因为他只在当前模块或元件中生效，同样的样式名可以在不同的模块或元件中重复使用，互不干扰；在多人协作或者分模块协作的时候效果尤为明显
注：后代选择器不要单独使用，因为污染的可能性较大；

```
/* 不推荐完整的结构树层级 */
.m-list { margin: 0; }
.m-list .m-list-itm { margin: 1px; }
.m-list .m-list-cnt { margin-left: 100px; }

/* 推荐 简洁写法，这里的.itm和.cnt只在.m-list中有效 */
.m-list { margin: 0; }
.m-list .itm { margin: 1px; }
.m-list .cnt { margin-left: 100px; }
```

媒体查询

尽量将媒体查询的规则靠近与他们相关的规则，不要将他们一起放到一个独立的样式文件中，或者丢在文档的最底部，这样做只会让大家以后更容易忘记他们

```
.element {
    ...
}
.element-avatar{
    ...
}

/* 推荐 媒体查询尽量靠近与其相关的规则 */
@media (min-width: 480px) {
    .element {
        ...
    }
    .element-avatar {
        ...
    }
}
```

注释规范-tsap

单行注释

- 注释以字符 /* 开始，以字符 */ 结束
- 注释不能嵌套

```
/* 这是一行单行注释 */
.element {
    ...
}
```

模块注释

注释内容第一个字符和最后一个字符都是一个空格字符，/* 与 模块信息描述占一行，多个横线分隔符-与*/占一行，行与行之间相隔两行。
推荐：

```
/* Module A
-------------------------------------------------------------- */
.mod_a {}


/* Module B
-------------------------------------------------------------- */
.mod_b {}
```

不推荐：

```
/* Module A ------------------------------------------------- */
.mod_a {}
/* Module B ------------------------------------------------- */
.mod_b {}
```

文件信息注释

在样式文件编码声明 @charset 语句下面注明页面名称、作者、创建日期等信息。

```
@charset "UTF-8";
/**
 * @desc File Info
 * @author Author Name
 * @date 2015-10-10
 */
```

JavaScript规范

js规范采用ES6，由于兼容浏览器版本问题--谷歌最低v52，尽量避免使用过高版本的ES规范并且做好向下兼容，遵循最大限度的可复用，可读性强，便于维护的原则。

语言规范

变量声明-tsap

const 和 let 都是块级作用域，var 是函数级作用域，变量声明必须加上关键字
使用规范：

- 声明变量使用const和let，不推荐使用var，避免变量提升等问题，更规范更安全
- 常量声明，使用const
- 变量声明，使用let
- 将所有的 const 和 let 分组

```
// 不推荐
let a;
const max;
let c;
const min;
let e;

// 推荐
const MAX;
const MIN;
let a;
let c;
let e;
```

块内函数声明-bfc

不要在块内声明一个函数，虽然很多 JS 引擎都支持块内声明函数, 但它不属于 ECMAScript 规范 (见 ECMA-262, 第13和14条). 各个浏览器糟糕的实现相互不兼容, 有些也与未来 ECMAScript 草案相违背. ECMAScript 只允许在脚本的根语句或函数中声明函数.

```
/* 不推荐 不要在块内声明一个函数 */
if (x) {
  function foo() {}
}
```

如果确实需要在块中定义函数, 建议使用函数表达式来初始化变量:

```
if (x) {
  let foo = function() {};
}
```

对象

- 使用对象方法的简写方式

```
// 不推荐
let item = {
  value: 1,

  addValue: function (val) {
    return item.value + val;
  }
};

// 推荐
let item = {
  value: 1,

  addValue(val) {
    return item.value + val;
  }
};
```

- 使用对象属性值的简写方式

```
let job = 'FrontEnd';

// 不推荐
let item = {
  job: job,
};

// 推荐
let item = {
  job,
};
```

- 对象属性值的简写方式要和声明式的方式分组

```
let job = 'FrontEnd';
let department = 'TSA';

// 不推荐
let item = {
  sex: 'male',
  job,
  age: 25,
  department,
};

// 推荐
let item = {
  job,
  department,
  sex: 'male',
  age: 25,
};
```

数组

- 使用拓展运算符 ... 复制数组（注意：浅拷贝）

```
// 不推荐
let items = [];
let itemsCopy = [];
let len = items.length;
let i;

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

// 推荐
itemsCopy = [...items];
```

解构赋值

- 当需要使用对象的多个属性时，请使用解构赋值

```
// 不推荐
function getFullName (user) {
  let firstName = user.firstName;
  let lastName = user.lastName;

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

// 推荐
function getFullName (user) {
  let { firstName, lastName } = user;

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

// 推荐better
function getFullName ({ firstName, lastName }) {
  return `${firstName} ${lastName}`
}
```

- 当需要使用数组的多个值时，请同样使用解构赋值

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

// 不推荐
let first = arr[0];
let second = arr[1];

// 推荐
let [first, second] = arr;
```

- 函数需要回传多个值时，请使用对象的解构，而不是数组的解构

```
// 不推荐
function doSomething () {
  return [top, right, bottom, left];
}

// 如果是数组解构，那么在调用时就需要考虑数据的顺序
const [top, xx, xxx, left] = doSomething();

// 推荐
function doSomething () {
  return { top, right, bottom, left };
}

// 此时不需要考虑数据的顺序
const { top, left } = doSomething()
```

字符串

- 字符串统一使用单引号的形式 "

```
// 不推荐
const department = "TSA";

// 推荐
const department = 'TSA';
```

- 字符串太长的时候，请不要使用字符串连接符换行 \，而是使用 +

```
const str = '全流量分析 全流量分析 全流量分析 ' +
  '全流量分析 全流量分析 全流量分析 ' +
  '全流量分析 全流量分析 全流量分析 ';
```

- 程序化生成字符串时，请使用模板字符串

```
const test = 'test';

// 不推荐
const str = ['a', 'b', test].join();

// 推荐
const str = 'a' + 'b' + test;

// 推荐
const str = `ab${test}`;
```

模块

- 使用标准的 ES6 模块语法 import 和 export

```
// 不推荐
let util = require('./util');
module.exports = util;

// 推荐
import Util from './util';
export default Util;

// 推荐better
import { Util } from './util';  // 对象解构
export default Util
```

- 不要使用 import 的通配符 *，这样可以确保你只有一个默认的 export

```
// 不推荐
import * as Util from './util';

// 推荐
import Util from './util';
```

分号-tsap

- 语句末尾需要添加分号，以确保代码被编译器的正常解析。

```
/* if,function,for,while,switch等代码块后不需要分号 */
if (x) {
        // code
}       // 末尾无需分号

function foo() {
        // code
}       // 末尾无需分号

for (init; condition; update) {
        // code
}       // 末尾无需分号

/* 并不是所有{}后面都需要省略分号 */
var foo = function() {}; // 函数表达式初始化变量，末尾添加分号
var oMenu = {
        title: 'xx',
        name: 'xxx',
        cnt: 10
}; // 对象声明和初始化，末尾添加分号
```

禁用方法

- eval()：能够将字符串解析为js语句，可能受到xss攻击，尽量避免使用
- with() {}： 扩展上下文的作用域，造成作用域污染，尽量避免使用

代码规范

统一团队的编码规范，有助于代码的维护。

单行代码块

- 单行代码块中推荐加入空格，便于阅读和维护

```
// 不推荐
a ? ((a > b) ? fn1() : fn2()) : fn1();
// 推荐
a ? ( (a > b) ? fn1() : fn2() ) : fn1();
```

大括号风格-tsap

在编程过程中，大括号风格与缩进风格紧密联系，用来描述大括号相对代码块位置的方法有很多。在 JavaScript 中，主要有三种风格，如下：

1. One True Brace Style

```
if (foo) {
  bar();
} else {
  baz();
}
```

2. Stroustrup

```
if (foo) {
  bar();
}
else {
  baz();
}
```

3. Allman

```
if (foo)
{
  bar();
}
else
{
  baz();
}
```

- 推荐使用One True Brace Style 风格
- 不推荐省略if和else后面的代码块符号{}。当阅读或扩展程序时，可能会引起意想不到的错误。

```
/* 不推荐省略if和else后面的代码块符号 */
if (err) return err.msg;

if (score > 50)
        setFail();
else
        setSuccess();

/* 如果扩展代码可能会导致意料之外的错误，比如: */
if (err) system.log(err.code);return err.msg;  // err.msg始终会执行

/* 推荐 总是使用{} */
if (err) {
        return err.msg;
}

if (score > 50) {
        setFail();
} else {
        setSuccess();
}

/* 如果确实希望代码保持在一行，应该这样 */
if (err) { return err.msg; }
```

变量命名

- 当命名变量时，主流分为驼峰式命名和下划线命名，推荐使用驼峰式命名。

细分约定-bfc：

- 所有变量名应是有意义的英文，不要使用拼音或单个字母+数字；
- 变量命名采用小驼峰法(第一个单词首字母小写)；
- 常量所有单词大写，并且每个单词间加下划线；
- 类命名必须是大驼峰法(所有单词第一个字母均大写)；
- 类的私有变量属性成员， 建议使用混合式命名，并在前面加下划线。

```
/* 推荐 */
var sHtml = '<div>动态HTML</div>'; // 变量见名知意，小驼峰
const PI = 3.14;                    // 常量大写
function Car() {                    // 类名大驼峰
        var _maxSpeed = 200;          // 私有变量属性，下划线开头
        this.start = function() {}
        this.stop = function() {}
}

/* 不推荐 */
var a1 = '<div>动态HTML</div>';    // 变量名字母+数字，无意义
var pi = 3.14;                      // 常量名小写，不易辨识
function car() {                    // 类名全小写，不易辨识
        var maxSpeed = 200;           // 私有变量，不易辨识
        this.start = function() {}
        this.stop = function() {}
}
```

拖尾逗号

使用拖尾逗号的好处是，简化了对象和数组添加或删除元素，我们只需要修改新增的行即可，并不会增加差异化的代码行数。

```
let foo = {
  name: 'foo',
  age: '22',
}
```

逗号风格

- 逗号分隔列表时， 推荐将逗号放置在当前行的末尾。

```
// 不推荐
let foo = 1
,
bar = 2;

let foo = 1
, bar = 2;

let foo = ['name'
          , 'age']
// 推荐
let foo = 1,
    bar = 2;

let foo = ['name',
            'age'];
```

逗号空格

- 为了提高代码的可读性，推荐在逗号后面使用空格，逗号前面不加空格。

```
// 不推荐
let foo = 1,bar = 2;
let foo = 1 , bar = 2;
let foo = 1 ,bar = 2;
// 推荐
let foo = 1, bar = 2;
```

缩进

- 约定使用 空格 来缩进，而且缩进使用两个空格。
- 通过配置 .editorconfig ，将 Tab 自动转换为空格。

对象字面量的键值缩进

- 约定对象字面量的键和值之间不能存在空格，且要求对象字面量的冒号和值之间存在一个空格。

```
// 不推荐
let obj = { 'foo' : 'haha' };
// 推荐
let obj = { 'foo': 'haha' };
```

构造函数首字母大写

- 约定构造函数的首字母要大小，以此来区分构造函数和普通函数。

```
// 不推荐
let fooItem = new foo();
// 推荐
let fooItem = new Foo();
```

链式赋值

- 链式赋值容易造成代码的可读性差，所以团队约定禁止使用链式赋值

```
// 不推荐
let a = b = c = 1;
// 推荐
let a = 1;
let b = 1;
let c = 1;
```

代码块空格

- 推荐代码块前要添加空格。

```
// 不推荐
if (a){
  b();
}

function a (){}
// 推荐
if (a) {
  b();
}

function a () {}
```

操作符的空格

- 推荐操作符前后都需要添加空格。

```
// 不推荐
let sum = 1+2;
// 推荐
let sum = 1 + 2;
```

注释-tsap

添加合理的注释能便于维护，方便代码review和重构

- 特殊逻辑的语句，添加单行注释
- 单独一行：//(双斜线)与文字之间保留一个空格
- 代码后面添加注释：//(双斜线)与代码之间保留两个个空格，并且//(双斜线)与文字之间保留一个空格

```
/* 推荐 单行注释 */
// 以下为初始化代码：1)单独在一行
var maxCount = 10;  // 设置最大量；2)在代码后面注释
setTitle();         // 设置标题；3) 与上面的注释对齐
setName();          // 设置名称；3) 临近的多处注释推荐对齐
```

- 方法前书写方法注释，描述参数和返回值

```
 /**
  * @description: 方法描述
  * @param {type} 参数名
  * @return:
  */
```

- js文件前添加注释，描述作者和创建时间

```
/*
 * @Description: 项目描述
 * @Author: 作者名
 * @Date: 创建时间
 */
```

方法和文件的注释，推荐使用idea的自定义注释代码块，方便快捷，戳这里查看

过长的单行予以换行-bfc

换行应选择在操作符和标点符号之后

```
/* 推荐 逻辑运算符在换行首，更利于代码阅读 */
if (oUser.nAge < 30
    && oUser.bIsChecked === true
    || oUser.sName === 'admin') {
    // code
}
```

循环的使用-bfc

在循环中，尽量使用变量先获取到相关数值，在放入循环中进行判断，否则非常影响性能。

```
/* 不推荐 每次循环都要重新计算length值 */
for (var i = 0; i < $('#id').children().length; i++) {
    // code
}

/* 推荐 先获取len再使用 */
for (var i = 0, len = $('#id').children().length; i < len; i++) {
    // code
}
```

代码间隔-bfc

- 功能模块之间，间隔一行就够了，不需要多行空白。
- 控制器间最多间隔两行

```
/* 不推荐 功能模块之间使用过多换行，太随意 */
// 功能模块1
init();
setTitle();
setName();



// 功能模块2
createElement();
getDate();
getData();

/* 推荐 模块间使用一行空白，必要时添加注释说明 */
// 功能模块1
init();
setTitle();
setName();

// 功能模块2
createElement();
getDate();
getData();
```

代码结构化-bfc

以Angular的controller代码为示例，展示常见的代码结构

```
/* 不推荐 代码零散 */
app.controller('appPage', function($scope){
        $scope.add = function() {
                // code
        }
        $scope.edit = function() {
                // code
        }
        $scope.delete = function() {
                // code
        }

        $scope.submitLoading = false;
        $scope.dataLoading = false;

        $scope.getParam = function() {
                // code
        }
        $scope.getData = function() {
                // code
        }

        // code
});
```

以功能模块来对函数进行归类

```
/* 推荐  */
app.controller('appPage', function($scope){
        // 菜单栏model层
        $scope.menuModel = {
                loading: false,
                add: function() {
                        // code
                },
                edit: function() {
                        // code
                },
                delete: function() {
                        // code
                }
        }

        // 数据列表model层
        $scope.mainModel = {
                loading: false,
                getParam = function() {
                        // code
                },
                getData: function() {
                        // code
                }
        }

        // code
});
```

另外：

1. 在HTML里直接使用的变量，须在controller里声明和初始化；
2. 代码封装，逻辑处理->service，功能组件->directive（比如时间组件、上传控件、图表绘制），或两者结合。