# 前端代码规范

**规范信息**

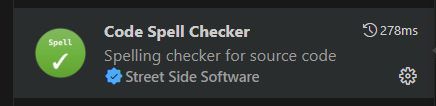
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态 | 版本 | 日期 | 负责人 | 批准 |
| 0.1 | 草稿 | 2023-12-12 | 张雨馨、陈雪英 | 李腾飞 |

**更改历史**

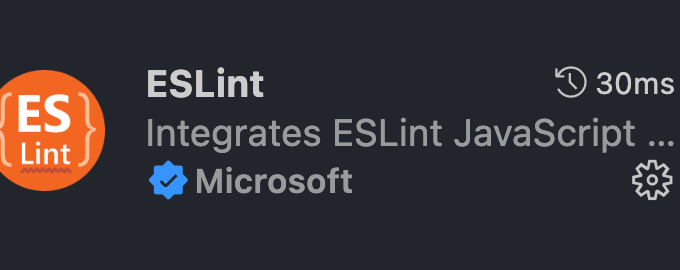
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 状态 | 日期 | 负责人 | 更改原因 |
| 0.1 | 草稿 | 2023-12-12 | 张雨馨、陈雪英 | 根据Angular风格指南以及各大厂代码规范起草 |

**一、VScode插件使用建议**

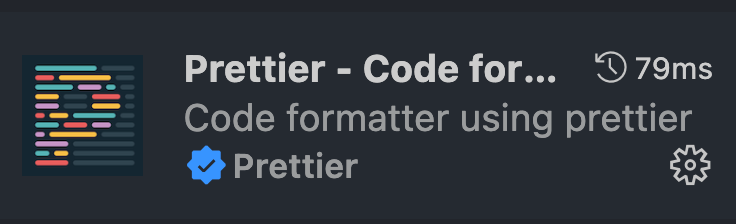
**1【强制】安装Code Spell Checker英文拼写检查插件**



**2【强制】安装ESLint，配置文件统一提供**



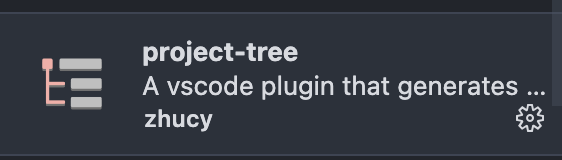
**3【强制】安装Prettier，配置文件统一提供**



**4【强制】本地历史插件，防止误删代码**



**5【建议】 项目结构树，可以快速生成**



**二、前端命名规范**

**目录规范**

|  |
| --- |
| 应用工程目录  ├─ src  │ ├─ app  │ │ ├─ app-routing.module.ts  │ │ ├─ app.component.css  │ │ ├─ app.component.spec.ts  │ │ ├─ app.component.ts  │ │ ├─ app.module.ts  │ │ ├─ implementation  │ │ │ ├─ programs  │ │ │ │ ├─ demo //模块根目录  │ │ │ │ │ ├─ components //公共组件  │ │ │ │ │ │ ├─ demo1-component  │ │ │ │ │ │ ├─ demo2-component  │ │ │ │ │ ├─ model //实体  │ │ │ │ │ ├─ pages //作业组件文件夹  │ │ │ │ │ │ ├─ index  │ │ │ │ │ │ │ ├─ demo-index.component.html  │ │ │ │ │ │ │ ├─ demo-index.component.less  │ │ │ │ │ │ │ ├─ demo-index.component.spec.ts  │ │ │ │ │ │ │ └─ demo-index.component.ts  │ │ │ │ │ │ ├─ query  │ │ │ │ │ ├─ repository  │ │ │ │ │ │ └─ demo.repository.ts  │ │ │ │ │ └─ service  │ │ │ │ │ └─ demo.service.ts  │ │ │ │ │ ├─ demo-routing.module.ts  │ │ │ │ │ ├─ demo.component.html  │ │ │ │ │ ├─ demo.component.less  │ │ │ │ │ ├─ demo.component.spec.ts  │ │ │ │ │ ├─ demo.component.ts  │ │ │ │ │ ├─ demo.module.ts  │ │ │ │ ├─ custom //客制类文件  │ │ │ │ │ ├─ custom-kc-detail //客制页面  │ │ │ │ ├─ demo //内部演示类文件  │ │ │ │ │ ├─ demo-online-edit //内部演示页面  │ │ │ ├─ shared //共享module  │ │ │ │ ├─ repository  │ │ │ │ │ ├─ common.repository.ts //公共服务  │ │ │ │ │ └─ modal.repository.ts  │ │ │ │ ├─ service  │ │ │ │ │ ├─ common.service.ts //公共接口请求  │ │ │ │ │ └─ modal.service.ts //通用组件service  │ │ │ │ ├─ shared.module.ts  │ │ │ │ └─ sharing  │ │ │ │ ├─ modal //通用组件  │ │ │ │ │ ├─ modal.component.html  │ │ │ │ │ ├─ modal.component.less  │ │ │ │ │ ├─ modal.component.spec.ts  │ │ │ │ │ └─ modal.component.ts  │ │ │ │ ├─ sharing-routing.module.ts  │ │ │ │ ├─ sharing.component.html  │ │ │ │ ├─ sharing.component.ts  │ │ │ │ └─ sharing.module.ts  │ ├─ assets  │ │ ├─ api.dev.json // 链接各个平台配置地址  │ │ ├─ api.json  │ │ ├─ i18n  │ │ │ ├─ en\_US  │ │ │ │ ├─ default.json // 所有国际化文字写在此文件  │ │ │ ├─ zh\_CN  │ │ │ │ ├─ default.json // 所有国际化文字写在此文件  │ │ │ └─ zh\_TW  │ │ │ ├─ default.json // 所有国际化文字写在此文件  │ │ ├─ iconfont // 图标库  │ │ └─ img // 图片存放处  ├─ angular.json // angular配置json  └─ package.json // 打包引入依赖包入口 |

1【强制】必须在programs文件夹下面创建页面

2【强制】必须在严格按照以上目录创建目录层级

**目录的命名**

【强制】全部采用小写方式， 以中划线分隔，有复数结构时，要采用复数命名法， 缩写不用复数

正例：components下面有多个文件，所以要加s,写成复数形式

反例：img是缩写不用加s, 不能写成imgs

**工程目录下模块开发根目录的命名**

【强制】使用模块的名称命名，不需要做任何修改，模块的名称根据音译使用中划线"-"连接

正例：knowledge，recycle-bin

反例：KNOWLEDGE，recycle\_bin

**客制/公共组件命名**

根据页面的类型，在相应的文件夹下面创建文件，必须用ng g c 命令创建组件,严禁手动创建,定制组件名称统一使用小写形式，以中划线分隔。

|  |  |
| --- | --- |
| **类型** | **规则** |
| 通用页面 | 以上目录所示，以模块功能设定 |
| 客制demo | custom-客户名称-页面名称 |
| 演示demo | demo-页面名称 |

正例：ng g c knowledge/knowledge-detail

ng g c custom/custom-workreport-detail

反例：直接在目录下手动创建

**文件的命名**

【强制】采用小写方式， 以中划线分隔

正例：render-dom.ts、render-dom.js、render-dom.html

反例：renderDom.ts、renderDom.js、renderDom.html

**三、TS代码规范**

**命名规范**

**1【强制】变量、函数名称使用小驼峰命名；组件传值命名建议。**

（1）语义化，见名知意：动词+名词

（2）变量：数组：dataList；禁止用如temp等语义不明的名称

（3）函数名称：getData()：获取数据；setData()：设置数据；onChangeData()：改变数据；onClickName()：点击事件； handleData()：处理数据；

（4）不要给output属性添加on前缀，为了区别事件本身【参照angular官网代码规范指南】：

比如：bad：@Output() onSaveForm = new EventEmitter<IPerson>()

（5） 如果该属性很难构造出默认值，请使用 ? 来把该属性显式标记为可选的；

如果字段是必填的,建议使用 ! ，并设置默认值。

@Input() name?:string;

@Input() id!:string='123';

示例

|  |
| --- |
| TypeScript // good case let loadSuccess = false; function getPersonInfo(){};  // bad case let LoadSuccess = false; function GetPersonInfo(){}; |

**2【强制】类、接口、类型别名、命名空间、枚举使用大驼峰命名**

* 接口：IXXX；以I开头；
* 枚举：常量使用大写

export enum SolutionState {

PUBLISHE= 0,

OFFL = 1,

}

示例

|  |
| --- |
| TypeScript // good case class AppComponent {} interface IPersonInfo {} type UserInfo = ; namespace SomeNameSpaceName {}  // bad case class appComponent {} interface aPersonInfo {} type userInfo = ; namespace someNameSpaceName {} |

**3【强制】常量、枚举属性使用全字母大写，单词下划线分割的命名方式**

示例

|  |
| --- |
| TypeScript // good case let ZERO = 0; // 枚举默认以0开始，若需将其以指定数字开始，可添加＝号 let PersonNationality = {  CHINA,  USA,  UK,  ABC\_DFG }  // bad case let Zero = 0; // 枚举默认以0开始，若需将其以指定数字开始，可添加＝号 let personNationality = {  china,  usa,  uk,  abc } |

**4【建议】善用标签说明：**

TODO/FIXME/NOTE/WIP

|  |
| --- |
| TypeScript // TODO 待办 // FIXME 修复这个bug // NOTE 注意事项 // WIP 任务在开发中 |

**5【建议】注释：**

复杂方法善用多行注释标明，简单方法、变量因见名知意，起到注释的作用

TODO/FIXME/NOTE/WIP

|  |
| --- |
| TypeScript  // good case  /\*\*     \* 打开选择弹框     \* @param type：类型 1-问题内容 2-答案 3-关键字段     \* @returns     \*/    openSelectModel(type: number) {} |

**格式规范**

**1【强制】空格**

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

|  |
| --- |
| JavaScript // good case let a = !arr.length; a++; a = b + c;  // bad case let a =!arr.length; a ++; a =b+ c; |

**2【强制】字符串应采用单引号的形式**

示例

|  |
| --- |
| Plain Text let hello: string = '你好'; |

**3【强制】不得省略语句结束的分号**

|  |
| --- |
| TypeScript // good case const a = 10; // bad case const a = 10 |

**4【强制】花括号总应在上一行尾行**

|  |
| --- |
| Go // good case  function sayHello(){ }  // bad case function sayHello() { } |

**5【强制】换行，每行不得超过140个字符,超过后可读性会变差**

**6【建议】组件中，类型定义应放在第一位**

|  |
| --- |
| TypeScript @Component({  selector: 'app-root',  template: '<router-outlet></router-outlet>',  styleUrls: ['./app.component.css'], }) export class AppComponent {  loading = false;  constructor() {} } |

**编码规范**

**通用**

**1【强制】避免使用any类型，相反，创建一个接口/类或使用unknown类型**

|  |
| --- |
| TypeScript  // good case  dmcUrl: string;  // bad case  dmcUrl: any;    // good case  temp[i] as unknown as number  // bad case  temp[i] as any    // good case  interface IProgramInfoParams {  /\*\* 定制页路径 \*/  sourceRoot: string;  /\*\* 页面type \*/  type: string;  /\*\* json路径 \*/  jsonPath: string;  /\*\* 定制页说明 \*/  description?: string; }  // bad case IProgramInfoParams: any; |

**2【强制】魔法数字应使用枚举代替**

|  |
| --- |
| Go // good case let PersonNationality = {  CHINA = 1,  USA = 2,  UK = 3,  ABC\_DFG = 4 } // bad case let a = 1; let b = 2; let c = 3; |

**3【强制】使用undefined代替null**

|  |
| --- |
| TypeScript // good case //类组件全局变量 description; // 局部变量 let description = undefined;  //bad case description = null; |

**4【强制】变量不应存在undefined的情况**

|  |
| --- |
| TypeScript // good case // names永远存在、至少为空数组 obj!.names.forEach(() => {})  // bad case // names可能不存在 obj.names.forEach(() => {}) |

**5【建议】将复杂函数尽可能拆分成小函数**

**6【建议】判断应尽量简单增加可读性**

**7【建议】抽离公共组件/服务/指令/方法**

**8【建议】尽可能使用===代替==**

**9【建议】少用迭代器forin forof**

**10 【建议】尽量避免代码多层嵌套（超过3层）及回调地狱**

**11 【建议】发到正式区的代码，避免存在console，禁止存在debugger**

**类**

**1【强制】变量/方法等按需取用**

在声明变量/方法时使用数据访问说明符(public/private),不同语言的默认访问修饰符都是不一样的, Java 的默认访问修饰符是包级别, Typescript 的默认访问修饰符是public, 始终指定 访问修饰符可以避免意外暴露接口

**2 【强制】使用rxjs进行订阅后应适时取消订阅，解决内存泄露的问题**

|  |
| --- |
| TypeScript // good case valueChanges.pipe(takeUntil(this.destroy$))  .subscribe(() => {});   // bad case valueChanges.subscribe(() => {}); |

**3【强制】使用const/let代替var**

|  |
| --- |
| TypeScript // good case const cat = 'tom'; let mouse = 'jerry'; // bad case var mouse = 'jerry'; |

**4【强制】避免全局变量，优先使用局部变量**

**5【建议】导入时按需引入，减少不必要的引入**

示例：

|  |
| --- |
| Go // good case import { Subject } from 'rxjs'; // bad case import \* from 'moment-timezone'; |

**6【建议】类组件成员顺序**

（1）静态成员=>实例成员

（2）属性=>方法

（3）公开=>保护=>私有

示例

|  |
| --- |
| JavaScript // good case class MyComponent extends Component {  // 静态属性  public static props = {};  // 实例属性  public age = {};  // 私有属性  private show = false;    // 静态方法  public static getMyName(){}  // 公开实例方法  public commontTest(){}  // 私有实例方法  private handleClick(){} }  // bad case class MyComponent extends Component {  private handleClick(){}  public static getMyName(){}  private show = false;   public age = {};  public commontTest(){}  public static props = {}; } |

**接口定义**

**1【建议】在声明变量或接口属性时，针对可选属性使用ts可选运算符**

|  |
| --- |
| TypeScript interface PersonInfo {  name: string;  age: number;  height?: number; } |

**2【建议】不要一味的导出类型/函数(export)等，除非确定需要在多个组件之间进行共享**

**模块**

**1【建议】模块导入顺序和分组：**

当文件越来越大时，import语句就会变成一团乱麻，此时我们按以下方法进行整理

（1）npm模块

（2）src相对导入模块

（3）相对路径

每个分组之间使用空行分割，分组内按照模块导入路径以字母顺序进行排序：

|  |
| --- |
| Go import A from "a"; import B from "b";  import MyComponent from "@/component/MyComponent"; import Redirect from "@/components/Redirect";  import foo from "../../foo"; import bar from "../bar"; import bar from "./bar"; |

**三、LESS规范**

**代码组织**

【强制】代码必须按如下形式按顺序组织

|  |
| --- |
| JSON @import 变量声明 样式声明 |

【强制】@import语句引用的文件必须在一对引号内，必须带.less后缀，引号使用' 或者""都可以，但是在同一个项目内必须保持一致。

**属性、变量**

【强制】属性名后的冒号（:）与属性值之间必须保留一个空格，冒号前不得保留空格

【强制】定义变量时冒号（:）与变量值之间必须保留一个空格，冒号前不得保留空格

正例：

|  |
| --- |
| CSS @a: 100px; color: #aaa; |

反例：

|  |
| --- |
| CSS @a:100px; color:#aaa; |

**运算**

【强制】+-\*/四个运算符两侧必须保留一个空格，+-两侧的操作数必须有相同的单位，如果其中一个是变量，另一个必须书写单位。

【建议】在编写样式的时候尽量少使用运算，大量的计算会很消耗性能

正例：

|  |
| --- |
| CSS @a: 200px; @b: (@a + 100px) \* 2; |

反例：

|  |
| --- |
| CSS @a: 200px; @b: (@a+100)\*2; |

**ng-deep伪类**

【强制】使用::ng-deep之前带上:host选择器。如果::ng-deep组合器在:host伪类之外使用，改样式就会污染其他组件，使用::ng-deep，避免使用/deep/

正例：

|  |
| --- |
| CSS :host ::ng-deep app-ci-demo h3 {  font-size: 18px; } |

反例：直接在最外层修改

|  |
| --- |
| CSS ::ng-deep h3 {  font-size: 18px; } |

**选择器**

【建议】当多个选择器共享一个声明块时，每个选择器声明必须独占一行

正例：

|  |
| --- |
| CSS h1, h2, h3{  font-weight:700; } |

反例：

|  |
| --- |
| CSS h1,h2,h3{  font-weight:700; } |

**颜色**

【建议】颜色定义需采用#rrggbb的格式（十六进制），并在可能时尽量缩写为#rgb形式，且避免直接使用颜色名称与rgb()表达式。

正例：

|  |
| --- |
| CSS color: #333 |

反例：

|  |
| --- |
| CSS color:rgb(255,255,255) |

**嵌套和缩进**

嵌套的作用：为了让层级关系更加清晰

(1)【建议】采用4个空格为一次缩进，不得采用tab作为缩进

(2)【建议】嵌套的声明块必须增加一次缩进，有多个声明块共享命名空间时尽量嵌套书写

(3)【强制】避免不必要的嵌套。这是因为虽然你可以使用嵌套，但是并不意味着应该使用嵌套。只有在必须将样式限制在父元素内（也就是后代选择器），并且存在多个需要嵌套的元素时才使用嵌套,不然会导致样式的权重越来越重，也不易于代码的阅读

(4)【强制】避免超过4层的嵌套

如下层级结构：

|  |
| --- |
| HTML <div class="list-box">  <ul class="list-ul-box">  <li class="list-ul-li-box">  <span class="list-ul-li-span-box"></span>   </li>  </ul> </div> |

正例：保证最外层的class命名的唯一性，对模块进行拆分

|  |
| --- |
| CSS .list-box{  font-size:14px;  .list-ul-box{  width:600px;  margin:30px auto;  padding:0px;  list-style:none;  }  .list-ul-li-box{  height:30px;  line-height:30px;  background-color:pink;  margin-bottom:5px;  }  .list-ul-li-span-box{  float:left;  font-size:12px;  } } |

反例：

|  |
| --- |
| CSS .list-box{  font-size:14px;  .li-ul-box{  width:600px;  margin:30px auto;  padding:0px;  list-style:none;  .list-ul-li-box{  height:30px;  line-height:30px;  background-color:pink;  margin-bottom:5px;  .list-ul-li-span-box{  float:left;  font-size:12px;  }  }   } }  less最后编码还是得转换成css，上面的代码转换之后变成如下代码，可读性不强，而且继承太多。  .list-box {  font-size: 14px; } .list-box .li-ul-box {  width: 600px;  margin: 30px auto;  padding: 0px;  list-style: none; } .list-box .li-ul-box .list-ul-li-box {  height: 30px;  line-height: 30px;  background-color: pink;  margin-bottom: 5px; } .list-box .li-ul-box .list-ul-li-box .list-ul-li-span-box {  float: left;  font-size: 12px; } |

**【建议】css样式顺序问题**

1. 布局定位属性：display / position / float / clear / visibility / overflow（建议 display 第一个写，毕竟关系到模式）
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

正例：

|  |
| --- |
| CSS .box {  display: flex;  width:30px;  height: 30px;  margin-bottom: 5px;  line-height: 30px;  background-color: pink; } |

反例：

|  |
| --- |
| CSS .box {  height: 30px;  line-height: 30px;  background-color: pink;  margin-bottom: 5px;  display: flex;  width:30px; } |

**继承**

【建议】在使用继承的时候，在声明块内书写 :extend 语句，必须写在开头

|  |
| --- |
| CSS .nav{  &:extend(.line);  background:blue; } .line{  color:#333; } |

**less文件中JS 表达式**

【建议】可以使用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使用单引号（'）

正例：

|  |
| --- |
| CSS @str: "hello"; @var:~`'@{str}'.toUpperCase()+'!'`; |

反例：

|  |
| --- |
| CSS @str: "hello"; @var:~`"@{str}".toUpperCase()+"!"`; |

**私有属性前缀**

【建议】同一属性有不同的私有前缀，尽量按照前缀长度的降序书写，标准形式必须写在后面，尽量按照冒号的位置对齐。

正例：

|  |
| --- |
| CSS .box {  -webkit-transform:rotate(3deg);  -moz-transform:rotate(-3deg);  -ms-transform:rotate(-3deg);  transform:rotate(-3deg); } |

反例：

|  |
| --- |
| CSS .box {  -webkit-transform:rotate(3deg);  -moz-transform:rotate(-3deg);  -ms-transform:rotate(-3deg);  transform:rotate(-3deg); } |

**四、GIT提交规范**

**背景**

Git是目前世界上最先进的分布式版本控制系统，在我们平时的项目开发中已经广泛使用。而当我们使用Git提交代码时，都需要写Commit Message提交说明才能够正常提交。

*git commit -m "提交"*

然而，我们平时在编写提交说明时，通常会直接填写如"fix"或"bug"等不规范的说明，不规范的提交说明很难让人明白这次代码提交究竟是为了什么。而在工作中，一份清晰简明规范的Commit Message能让后续代码审查、信息查找、版本回退都更加高效可靠。

**提交规范**

那么，什么样的提交说明才能符合规范的说明呢？不同的团队会制定不同的规范，当然，我们也可以直接使用目前业界流行的规范，比如[commitlint](https://github.com/conventional-changelog/commitlint/#what-is-commitlint)。接下来将会对该提交规范进行介绍。

**提交格式**

*type(scope?): subject*

*#type代表提交类型，scope代表本次提交范围，subject表明提交主题*

**提交类型**

1. feat: 任务的开发
2. fix: bug的修复
3. refactor：代码重构，没有加新功能或者修复bug
4. style： 样式修复
5. script： 脚本的研发
6. chore: 改变构建流程、或者增加依赖库、工具等 (package.json的改动需写明chore)
7. doc: readme等文档类的开发
8. ci: 更改CI配置文件和脚本
9. test: 单元测试的修改

**作用域**

scope中编号则为禅道上的编号，用来追溯对应的开发任务和bug信息，以此让后续流程变得更加高效可靠

当我们在任务中需修复bug时，则scope可写成feat(fix 666): 修复批量赋值某bug，这样的形式，以表明是在修复任务中的bug

**提交主题**

主题（subject）描述是简短的一句话，简单说明此次提交的内容。

真实的例子是这样的：

*chore(chore 999): git添加提交信息校验钩子*

*feat(fix 666): 修复批量赋值某bug*

*feat(feat 666): 开发定制按钮功能*

**强制校验提交信息**

虽然有了规范，但是还是无法保证每个人都能够遵守相应的规范，因此就需要使用一些工具来保证大家都能够提交符合规范的Commit Message。那我们可以使用Husky + Commintlint 打造规范的Git检查工作流，确保我们的提交信息只有符合规范才能提交到代码仓库。

**Husky**

[husky](https://github.com/typicode/husky)是常见的git hook工具，使用husky可以挂载Git钩子，当我们本地进行git commit或git push等操作前，能够执行其它一些操作，比如进行ESLint、commit-msg检查，如果不通过，就不允许commit或push。

**CommitLint**

commitlint通过结合husky一起使用，可以在开发者进行commit前就对Commit Message进行检查，只有符合规范，才能够进行commit。