Angular 开发手册

作者： 刘鸿飞

日期： 2018-07-10

|  |  |  |
| --- | --- | --- |
| 修改人 | 修改内容 | 修改时间 |
| 刘鸿飞 | 添加界面要求 | 2018-08-07 |
| 刘鸿飞 | 添加meta要求 | 2018-11-12 |

# 前后端数据传输

1. 前后端数据传输以Json格式进行传输，return\_code为信息状态码，return\_msg为返回请求信息。错误时，return\_msg内显示错误信息。后端返回结构示例为下：

{

return\_code: 9000,

return-msg: {

List: [],

name: 'alwaysLHF'

}

}

错误时：

{

return\_code: 1010,

return-msg:

'输入信息错误'

}

1. 前端通过http请求获取数据后，必须先对错误码进行验证，然后再进行数据解析，对http请求错误必须进行处理。

example：

this.testService.accountResult().subscribe(

data => {

if (data !== null && data !== undefined) {

if (data[MESSAGE.status] === STATUS.success) {

if (data[MESSAGE.returnMsg].length === 0) {

this.msgs = [];

this.msgs.push({ severity: 'warn', summary: '警告', detail: '该时间段数据为空' });

}

this.accountResultList = data[MESSAGE.returnMsg];

} else {

if (true) {

this.msgs = [];

this.msgs.push({

severity: 'error', summary: '错误', detail: '错误码：' +

data[MESSAGE.status] + '<br/>' + data[MESSAGE.returnMsg]

});

}

}

} else {

this.msgs = [];

this.msgs.push({ severity: 'error', summary: '错误', detail: '系统返回数据不合法' });

}

},

error => {

this.msgs = [];

this.msgs.push({ severity: 'error', summary: '错误', detail: '系统响应失败' });

}

);

} else {

this.msgs = [];

this.msgs.push({ severity: 'warn', summary: '警告', detail: '选取日期区间必须小于等于31天' });

}

1. 后端数据传到前端不能出现null

正确示例：

name: '';

infoList：[];

错误示例：

name: null;

infoList: null;

1. 后端数据类型问题：

在前后端分离的系统，我们对前后端功能进行分工：后端负责业务逻辑的实现和计算。前端只负责后端数据的渲染，不进行任何逻辑或者计算操作。此外，前端由于只支持number数据类型，当后端数据位long时，前端必须使用string类型。

总结：前后端数据类型传输只能使用两种：string 、number。

一般情况下Date等类型禁止使用。number只在小数据情况下使用。

1. 数据格式推荐：

我们在与后端进行交互时，相同功能的数据格式应该保持统一。下面是一些推荐格式：

date1（具体到天）："20180705"

date2（具体到秒）："20180504 12:20:26"

date3（具体到分）："20180607 23:50"

date3（具体到分 12小时制）："20180607 07:50 AM"

# 代码开发要求

## 单一职责

对所有的组件、服务等等应用[*单一职责原则*](https://wikipedia.org/wiki/Single_responsibility_principle)

### 单一规则

**坚持**每个文件只定义一样东西（例如服务或组件）。

**考虑**把文件大小限制在 400 行代码以内。

**为何？**单组件文件非常容易阅读、维护，并能防止在版本控制系统里与团队冲突。

**为何？**单组件文件可以防止一些隐蔽的程序缺陷，当把多个组件合写在同一个文件中时，可能造成共享变量、创建意外的闭包，或者与依赖之间产生意外耦合等情况。

**为何？**单独的组件通常是该文件默认的导出，可以用路由器实现按需加载。

最关键的是，可以增强代码可重用性和阅读性，减少出错的可能性。

### 小函数

**坚持**定义简单函数

**考虑**限制在 75 行之内。

**为何？**简单函数更易于测试，特别是当它们只做一件事，只为一个目的服务时。

**为何？**简单函数促进代码重用。

**为何？**简单函数更易于阅读。

**为何？**简单函数更易于维护。

**为何？**简单函数可避免易在大函数中产生的隐蔽性错误，例如与外界共享变量、创建意外的闭包或与依赖之间产生意外耦合等。

## 总体命名原则

坚持所有符号使用一致的命名规则。

坚持遵循同一个模式来描述符号的特性和类型。推荐的模式为 feature.type.ts。

为何？命名约定提供了一致的方式来查找内容，让你一眼就能找到。 项目的一致性是至关重要的。团队内的一致性也很重要。整个公司的一致性会提供惊人的效率。

为何？命名约定帮助你更快得找到想找的代码，也更容易理解它。

为何？目录名和文件名应该清楚的传递它们的意图。 例如，app/heroes/hero-list.component.ts 包含了一个用来管理英雄列表的组件。

### 使用点和横杠来分隔文件名

**坚持** 在描述性名字中，用横杠来分隔单词。

**坚持**使用点来分隔描述性名字和类型。

**坚持**遵循先描述组件特性，再描述它的类型的模式，对所有组件使用一致的类型命名规则。推荐的模式为 feature.type.ts。

**坚持**使用惯用的后缀来描述类型，包括 \*.service、\*.component、\*.pipe、.module、.directive。 必要时可以创建更多类型名，但必须注意，不要创建太多。

**为何？**类型名字提供一致的方式来快速的识别文件中有什么。

**为何？** 利用编辑器或者 IDE 的模糊搜索功能，可以很容易地找到特定文件。

**为何？** 像 .service 这样的没有简写过的类型名字，描述清楚，毫不含糊。 像 .srv, .svc, 和 .serv 这样的简写可能令人困惑。

**为何？**为自动化任务提供模式匹配。

### 符号名与文件名

**坚持**为所有东西使用一致的命名约定，以它们所代表的东西命名。

**坚持**使用大写驼峰命名法来命名类。符号名匹配它所在的文件名。

**坚持**在符号名后面追加约定的类型后缀（例如 [Component](https://angular.cn/api/core/Component)、[Directive](https://angular.cn/api/core/Directive)、Module、[Pipe](https://angular.cn/api/core/Pipe)、Service）。

**坚持**在符号名后面追加约定的类型后缀（例如 .component.ts、.directive.ts、.module.ts、.pipe.ts、.service.ts）。

**坚持**在文件名后面追加约定的类型后缀（例如 .component.ts、.directive.ts、.module.ts、.pipe.ts、.service.ts）。

**为何？**遵循一致的约定可以快速识别和引用不同类型的资产。

### 服务名

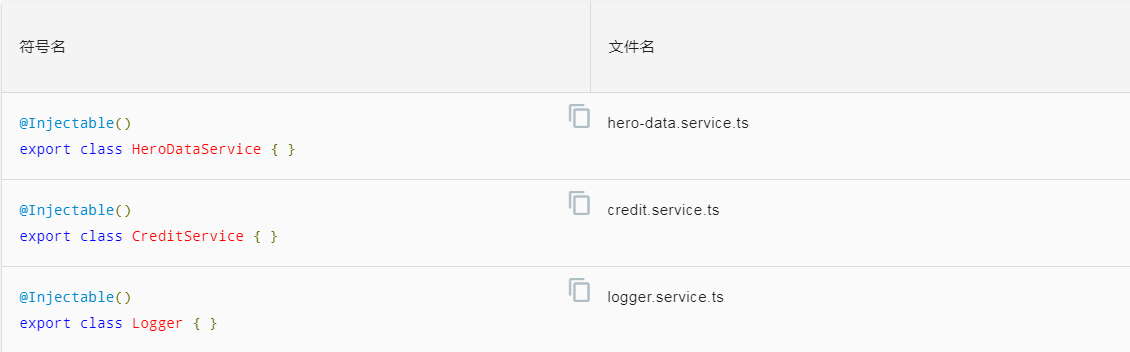
**坚持**使用一致的规则命名服务，以它们的特性来命名。

**坚持**为服务的类名加上 Service 后缀。 例如，获取数据或英雄列表的服务应该命名为 DataService 或 HeroService。

有些词汇显然就是服务，比如那些以“-er”后缀结尾的。比如把记日志的服务命名为 Logger 就比 LoggerService 更好些。需要在你的项目中决定这种特例是否可以接受。 但无论如何，都要尽量保持一致。

**为何？**提供一致的方式来快速识别和引用服务。

**为何？**像 Logger 这样的清楚的服务名不需要后缀。

**为何？**像 Credit 这样的，服务名是名词，需要一个后缀。当不能明显分辨它是服务还是其它东西时，应该添加后缀。

### 把逻辑放到服务里

**坚持**在组件中只包含与视图相关的逻辑。所有其它逻辑都应该放到服务中。

**坚持**把可重用的逻辑放到服务中，保持组件简单，聚焦于它们预期目的。

**为何？**当逻辑被放置到服务里，并以函数的形式暴露时，可以被多个组件重复使用。

**为何？**在单元测试时，服务里的逻辑更容易被隔离。当组件中调用逻辑时，也很容易被模拟。

**为何？**从组件移除依赖并隐藏实施细节。

**为何？**保持组件苗条、精简和聚焦。

### 把表现层逻辑放到组件类里

**坚持**把表现层逻辑放进组件类中，而不要放在模板里。

**为何？**逻辑应该只出现在一个地方（组件类里）而不应分散在两个地方。

**为何？**将组件的表现层逻辑放到组件类而非模板里，可以增强测试性、维护性和重复使用性。

### 服务

#### 服务总是单例的

**坚持**在同一个注入器内，把服务当做单例使用。用它们来共享数据和功能。

**为何？**服务是在特性范围或应用内共享方法的理想载体。

**为何？**服务是共享状态性内存数据的理想载体。

#### 单一职责

**坚持**创建单一职责的服务，用职责封装在它的上下文中。

**坚持**当服务成长到超出单一用途时，创建一个新服务。

**为何？**当服务有多个职责时，它很难被测试。

**为何？**当某个服务有多个职责时，每个注入它的组件或服务都会承担这些职责的全部开销

#### 提供一个服务

坚持在服务的 @[Injectable](https://angular.cn/api/core/Injectable) 装饰器上指定通过应用的根注入器提供服务。

为何？ Angular 注入器是层次化的。

为何？当你在根注入器上提供该服务时，该服务实例在每个需要该服务的类中是共享的。当服务要共享方法或状态时，这是最理想的选择。

为何？当你在服务的 @[Injectable](https://angular.cn/api/core/Injectable) 中注册服务时，CLI 生产环境构建时使用的优化工具可以进行摇树优化，从而移除那些你的应用中从未用过的服务。

为何？当不同的两个组件需要一个服务的不同的实例时，上面的方法这就不理想了。在这种情况下，对于需要崭新和单独服务实例的组件，最好在组件级提供服务。

### 使用 @Injectable() 类装饰器

**坚持**当使用类型作为令牌来注入服务的依赖时，使用 @[Injectable](https://angular.cn/api/core/Injectable)() 类装饰器，而非 @[Inject](https://angular.cn/api/core/Inject)() 参数装饰器。

**为何？** Angular 的 DI 机制会根据服务的构造函数参数的声明类型来解析服务的所有依赖。

**为何？**当服务只接受类型令牌相关的依赖时，比起在每个构造函数参数上使用 @[Inject](https://angular.cn/api/core/Inject)()，@[Injectable](https://angular.cn/api/core/Injectable)() 的语法简洁多了。

### 数据服务

#### 通过服务与 Web 服务器通讯

**坚持**把数据操作和与数据交互的逻辑重构到服务里。

**坚持**让数据服务来负责 XHR 调用、本地储存、内存储存或者其它数据操作。

**为何？**组件的职责是为视图展示或收集信息。它不应该关心如何获取数据，它只需要知道向谁请求数据。把如何获取数据的逻辑移动到数据服务里，简化了组件，让其聚焦于视图。

**为何？**在测试使用数据服务的组件时，可以让数据调用更容易被测试（模拟或者真实）。

**为何？**数据管理的详情，比如头信息、方法、缓存、错误处理和重试逻辑，不是组件和其它的数据消费者应该关心的事情。

数据服务应该封装这些细节。这样，在服务内部修改细节，就不会影响到它的消费者。并且更容易通过实现一个模拟服务来对消费者进行测试。

### 共用原则

所有代码开发时必须考虑可移植性和可维护性：

1. 所有逻辑代码必须符合单一职责原则，代码间耦合性低
2. 当一段代码或一个常量使用次数超过两次时，必须提取出来当做单独服务或常量

### 注释

所有变量常量必须拥有注释

所有方法函数必须拥有注释

# 界面要求

## 渲染引擎

index文件添加如下语句，保证多核浏览器webkit进行渲染

## 唯一标示

所有input、button、table、form等东西必须添加唯一标示以便于测试人员自动化测试。

## 敏感操作

所有敏感操作按钮等（如删除、冻结等点击错误易造成损失的）执行前必须弹出确认模态框，用户确认后再执行

## 颜色风格

界面图标图形颜色应该事先确定可选择选项若干，不够时再统一沟通后添加。颜色配色方案推荐如下：

方案一：



湖蓝：4BACC6

蓝：4F81BD

草绿：9BBB59

方案二：



1：F79646

2：4BACC6

3：8064A2

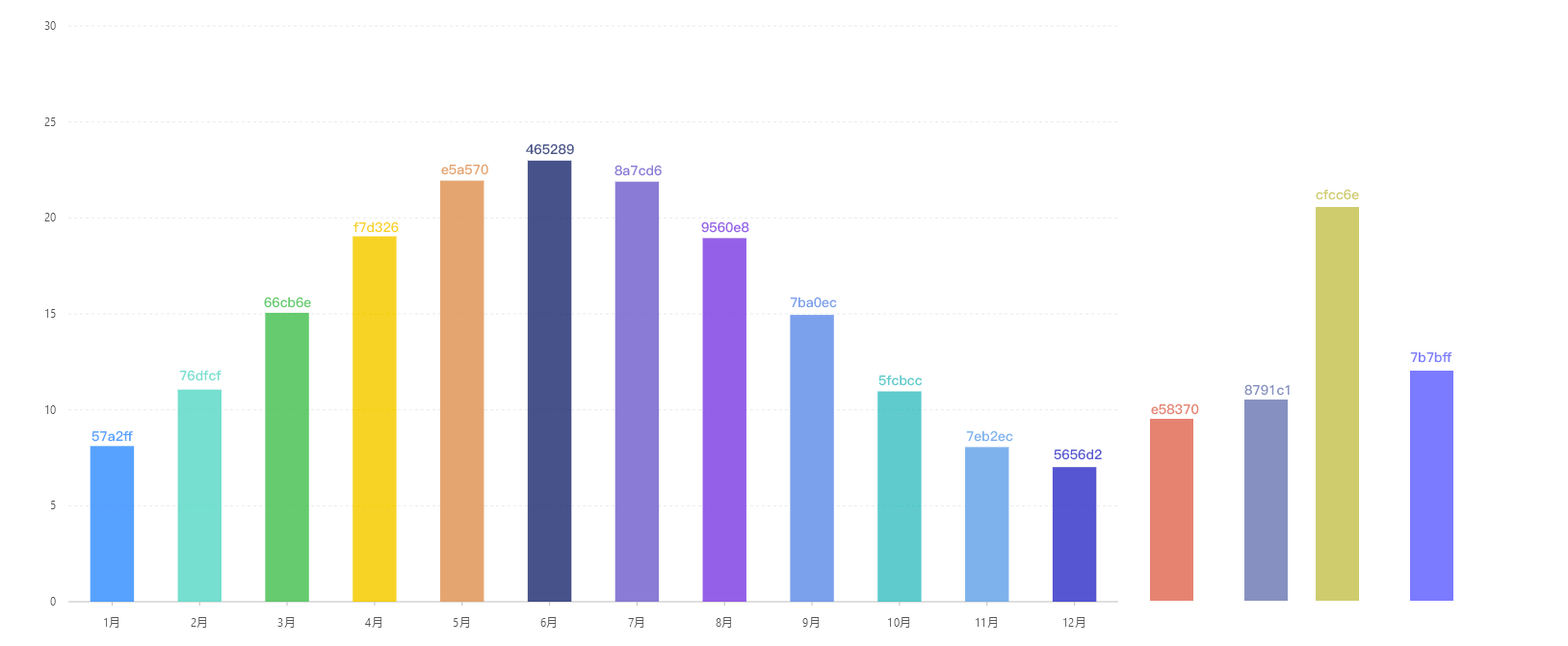
4：B65708

5：276A7C

6：4D3B62

7：F9AB6B

方案三：



方案四：



## 插件要求

同一项目下，为统一风格，如需求相同，则所使用插件必须相同：如输入框、模态框全部使用primeng，不能使用adminLTE或者原生的。

## 界面风格

1. 文字符号

尽量使用中文和中文符号，涉及英文字符的统一进行汉语翻译或者添加说明。

1. 按钮一般使用系统默认蓝色
2. 所有表单使用一同套CSS渲染。表单为一行时，按钮在输入框后面；表单超过一行时，按钮在表单下一行
3. 所有界面应该在宽屏和方屏都正常显示
4. 所有日历等依照中国年月日的显示方式

## 明确需求

为防止反复添加修改，常用组件应该提前明确：

### 输入框：

1. 最大最小长度
2. 字符要求（只输数字等）
3. 是否有提示词
4. 是否有默认值
5. 是否有占位符

### 表格

1. 是否分页
2. 数据为空时如何显示
3. 是否拥有批量操作

### 表单

1. 提交时判断信息是否完整

### 选择框

1. 单选或多选
2. 多选是否批量选择
3. 选项是否后台获取
4. 是否拥有默认选项，没有时选择框显示的文字
5. 统一：去掉输入，只能鼠标点击选择

### 日期选择框

1. 选择范围
2. 选择框显示格式yyyy/mm/dd 或yyyy-mm-dd
3. 统一：用户不能手动输入，只能鼠标点击选择

### 时间显示

1. 24小时制还是12小时制，中间加-还是/。推荐如下：

date1（具体到天）："20180705"

date2（具体到秒）："20180504 12:20:26"

date3（具体到分）："20180607 23:50"

date4（具体到分 12小时制）："20180607 07:50 AM"

date5（具体到天）："2018-07-05"

1. 显示时间是系统获取还是客户端获取