Clean Code

细节之中自有天地，整洁成就卓越代码

王先淦 2020.6.1

大纲

- 整洁代码
- 有意义的命名
- 函数
- 注释
- 格式

整洁的代码

为什么需要整洁的代码

- 混乱的代码导致团队生产力下降
- 对于繁琐的系统架构设计理解困难，不清楚什么样的修改符合设计意图，什么样的修改违背设计意图
- 混乱的代码会导致代码更加混乱
- 恶性循环...最后到无法维护

整洁的代码

- “我们都曾经瞟一眼自己亲手造成的混乱，决定弃之而不顾，走向新一天。我们都曾经看到自己的烂程序居然能运行，然后断言能运行的烂程序总比什么都没有强。我们都曾经说过有朝一日再回头清理。当然，在那些日子里，我们都没听过勒布朗法则：稍后等于永不”
- “光把代码写好可不够。必须时时保持代码整洁。我们都见过代码随时间流逝而腐坏。我们应当更积极地阻止腐坏的发生。借用美国童子军的一条简单的军规，应用到我们的专业领域：让营地比你来时更干净。”

有意义的命名

命名要有意义

- 名副其实。通过名称我们知道有关变量的大多数信息（是什么，做什么，怎么用）。
- 无歧义。（比如 少儿互动题 和 互动题 ）
- 简洁易懂，简洁是建立在易懂的基础上的。

```
a(b: string[], c: string[]) {
    for (let d = 0; d < b.length; d++) {
        c[d] = b[d];
    }
}
```

```
copyChars(char1: string[], char2: string[]) {
    for (let i = 0; i < char1.length; i++) {
        char2[i] = char1[i];
    }
}
```

有意义的命名

有意义的区分

- 同一作用范围内两样东西不能重名
- 如果名称相异，那其意思也应该不同
- 以数字系列命名是完全没有意义的

```
copyChars(char1: string[], char2: string[]) {
    for (let i = 0; i < char1.length; i++) {
        char2[i] = char1[i];
    }
}
```

```
copyChars(source: string[], destination: string[]) {
    for (let i = 0; i < source.length; i++) {
        destination[i] = source[i];
    }
}
```

有意义的命名

其他

- 不要把类型名字加进变量名里面（增加修改的难度）
- 命名取名规则使得程序更易读，驼峰法：取名的时候，名字里如果有多个单词，则后面的单词的首字母要大写。
- 要习惯把常量抽象成const变量，使得以后的修改方便且不容易出错。
- 类名大多数应该是名词。
- 方法名大多数应该是动词。

函数

基本规范

- 短小
- 还要更短小
- 函数应该只做一个事情（要判断函数是否只做了一件事，有一个办法是看在你编的函数里，你能否再拆出一个函数）
- 函数抽象层级，自顶向下规则

函数

- https://gerrit.zhenguanyu.com/c/tutor-focus-ng-common/+/694955/2/projects/tfnc-lib/src/lib/toast/toast.service.ts

函数

函数参数

- 参数名，最好和函数名有联系。这样可以，大大减轻记忆参数的负担
- 禁止把布尔值传入函数。因为这样做，意味着，宣称本函数不只做一件事：ture时做一件事，false时则在做另外一件事。
- 函数参数的个数尽可能少！最理想的参数数量是零。这个规则，使得用函数更简单，还使得测试函数得到了方便，因为输入的参数少了，困难麻烦就少了。
- 如果参数有多个的话，可能需要将参数封装成对象。
- 应该避免使用输出参数（比如 setStudentPhase(student, phase) 来调用显然没有 student.setPhase(phase) 舒服）。

函数

- switch 函数没有办法变得很简洁，可以采取工厂模式，将 switch 埋藏在较低的抽象层级

注释

别给糟糕的代码加注释——重新写吧。

- 注释是弥补我们在用代码表达意图时遭遇的失败。
- 若编程语言足够有表达力，或者我们长于用这些语言来表达意图，就不那么需要注释。
- 什么也比不上放置良好的注释来得有用。什么也不会比乱七八糟的注释更有本事搞乱一个模块。什么也不会比陈旧、提供错误信息的注释更有破坏性。

注释

- 注释存在的时间越久，就离其所描述的代码越远。
- 代码在变动，然而注释并不总是随之变动。
- 如果你发现自己需要写注释，再想想看是否有办法翻盘。

```
onSelectQuestions(questionIds: number[]) {
    // 进入设置分值环节
    const questionMarks = questionIds.map( callbackfn: () => null);
    if (Array.isArray(this.setScoreModal.questionMarks)) {
        for (let i = 0; i < questionMarks.length && i < this.setScoreModal.questionMarks.length; i += 1) {
            questionMarks[i] = this.setScoreModal.questionMarks[i];
        }
    }
    this.setScoreModal = {
        show: true,
        questionIds,
        questionMarks,
        individual: false,
    };
}
```

```
onSelectQuestions(questionIds: number[]) {
    this.setQuestionsScore(questionIds);
}

private setQuestionsScore(questionIds: number[]) {
    const questionMarks = this.getQuestionMarks(questionIds);
    this.showSetScoreModal(questionIds, questionMarks);
}

private getQuestionMarks(questionIds: number[]) {
    const questionMarks = questionIds.map( callbackfn: () => null);
    if (Array.isArray(this.setScoreModal.questionMarks)) {
        for (let i = 0; i < questionMarks.length && i < this.setScoreModal.questionMarks.length; i += 1) {
            questionMarks[i] = this.setScoreModal.questionMarks[i];
        }
    }
    return questionMarks;
}

private showSetScoreModal(questionIds: number[], questionMarks) {
    this.setScoreModal = {
        show: true,
        questionIds,
        questionMarks,
        individual: false,
    };
}
```

注释

好的注释

- 提供信息的注释。例如解释某个抽象方法的返回值，规定参数的顺序和个数。
- 对意图的解释。使别人更清楚一段复杂代码是在干什么。
- 法律信息。在每个源文件开头，写上版权时间等法律信息。

注释

常见的注释

- 阐释。把一些难懂的参数和返回值的意义翻译成某种可读形式。
- 警示。比如说有一个函数，测试它将会花上很多时间，我们将写上警示。
- 放大重要性。就是可以用来放大某处(看起来不合理)的重要性。
- TODO，大部分ide会识别定位TODO注释。如果有写TODO的习惯，要定期查看。

注释

坏注释

- 注释掉的代码。注释后的代码会严重误导以后别人阅读这段代码，在早已有版本控制系统的时代，系统会将之前版本的代码记录，而无需我们用注释来标记，所以被注释掉的代码应该全部删掉。
- 为了解释复杂语句的注释。当一行语句很复杂的时候，我们通常会写注释。但实际上规范的写法是，一个语句只做一件简单的事，我们应该重构原来的复杂语句，改成几行简单语句，从而代替注释。
- 喃喃自语。
- 多余的注释解代码。
- 误导性注释。有的时候注释缺少一些信息，会误导程序员，使得其他人简单的调用某个函数。

格式

好的代码格式，意味着代码的整洁和对细节的关注

- 如果是在团队中工作，则团队应该一致同意采用一套简单的格式规则，所有成员都要遵守，并且贯彻。
- 格式关乎沟通，而沟通是专业开发者的头等大事。
- 或许你认为让代码能工作才是专业开发者的头等大事。但是实际上，修改和维护代码才是开发者花时间花得最多的地方。只有拥有良好的代码格式，代码的可读性才会增加，这对日后修改和维护产生深远影响。

格式

垂直格式

- 类的属性变量应该全部在类的顶部声明，而不是东一个西一个，使人很难找到。
- 最上面的代码应该是最抽象的，底部细节应该在下面实现。这样就能像报纸文章一样，最重要的概念在最前面，底部细节最后才会出来。
- 概念相关的代码应该放到一起，相关性越强，彼此之间的距离就该越短。

格式

水平格式

- 一行的上限是120个字符。短代码行，利于理解，所以应该尽力保持代码行短小(在30个字符以内)

格式

缩进

- if，while，函数后面都应该加上缩进。违反缩进规则的代码，通常可读性极差

格式

团队规则

- 每个程序员都有自己喜欢的格式规则，但如果在一个团队中工作，就必须是团队说了算。
- 一组开发者应当认同一种格式风格，启动项目之前制定一套编码风格，所花时间很短，却能为为以后阅读他人代码、团队合作提供了巨大的便捷。

参考资料

- 代码整洁之道 clean code
- https://www.cnblogs.com/edisonchou/p/edc_clean_code_notes.html
- https://www.jianshu.com/c/42cb0f684c6c
- https://www.cnblogs.com/lyy-2016/p/6118040.html
- 知乎