构建大型前端业务项目的一点经验

目前工作中接手的几个项目都是 B端 PC 项目，业务逻辑都比较复杂，并且代码历史较久，在日常的维护中经常会遇到想撂手的技术问题，发现问题、解决问题、避免再次出现同样的问题，既是项目可持续维护的因素之一，也是个人工作经验积累的一个过程

具体、连贯的变量名

在前后端分离的现代化 `web` 开发流程下，相比于以往，前端承担了更多的业务逻辑，尽管存在着 `TypeScript` 等约束工具，但相比于后端语言， `js` 仍具备相当大的灵活性，这就导致了代码一旦复杂，前端代码的排查会更加麻烦

单一变量的层层传递与到处使用是很常见的事情，变量的命名对于追踪变量有着相当大的影响

所以变量名称必须是具体且有实际意义的，不提倡为了追求变量名的精确性而使得变量名称冗长，但模糊而宽泛的变量名同样不可取，这就要求**变量名称即准确又简短**，在某些时候，可能很难做到这一点，*个人倾向是，实在无法做好权衡的前提下，宁愿选择冗长的变量名也不要选择一个模糊宽泛的*

例如， `data` 可以当做是一个变量名，这个变量名用于**临时的局部变量**没啥问题，毕竟你一眼就能看到这个变量所有的使用范围，但如果变量所持有的数据的作用范围较大（例如跨组件）且具备实际业务意义，那么就不太妙了， `data` 可以用作任何数据的变量名，当需要追踪 `data` 的时候，在编辑器里搜索 `data` ，发现到处都是 `data` ，并不是一件美好的事情

一旦确定好了变量名后，最好不要再对其进行重命名，例如想将 `A` 组件里的 `userData` 传递到 `B` 组件中， `B` 组件最好原模原样地接收这个变量名，而不是将其命名为其他的什么名称

有些时候可能就必须要在传递变量的时候进行重命名，例如第三方组件就接收 `data` ，那么你无法传递 `userData` ，这种情况下当然不得不重命名

避免对变量重命名的目的主要是，为了防止在追踪变量的时候因变量名称改变而产生思维上的重新整理，一个变量被重命名了好几次后，追踪到底再回过头来你可能就忘记了自己当初在追踪什么东西，同时对于搜索功能也不太友好，从一而终的连贯变量名可以让你一次搜索就能从追踪的起始位置跳过中间一大堆逻辑直接到终点

必要的选择器

现代化的前端项目基本都会使用 `React` 、 `Vue` 等数据驱动的框架， `UI` 组件一般也都是使用别人封装好的组件库，除非是要写样式，否则 `html` 元素选择器基本上都是可有可无的了，但并不是说就不需要了，最起码的一个好处是，让你想在代码里查找页面上的一个元素时，直接根据选择器名就能精准定位了

页面上有个弹窗展示得不太对，你在浏览器页面里看到这个弹窗元素名叫 `ant-modal-wrap` ，是个第三方的组件所以你代码里根本搜不到这个选择器名；页面上有一句文案有点问题，你在浏览器页面里看到这个文案所在的元素是个没有选择器的 `div` 标签，在目前普遍 `全站div` 的浪潮下，光是定位这个文案到底是哪里的就够花费好一阵子了

所以，这里选择器是起到了一个**索引**的作用，既然是索引，那么同样应该遵守上面变量名相关的规则，即选择器名称应当 即准确又简短

优化应该从一开始就开始

不要提前优化

相信很多人都听过这句话，我曾经将这句话当做是至理名言，但经历的多了之后，目前已经开始有所质疑了

不要提前优化，那么要在什么时候优化？快要 `hold` 不住的时候才优化？迭代了七八十版的时候再优化？团队人员换了一茬又一茬的时候再优化？

当然是可以的，但是那个时候谁来优化，谁来做这种在业务看来毫无产出甚至是可能优化出 `bug` 的吃力不讨好的事情？

一个函数里塞了数十层的 `if...else` ，函数体的代码量超过千行，看着就应该要被优化的，但是这些代码在这里绵延了数年之久，经过了一批又一批不同程序员的修改，承载了不知多少明面上暗地里的业务逻辑，技术上或许好优化，但谁能保证某处优化不会对业务逻辑造成破坏？

没有提前优化，过程中也没有优化，那就完全是没有任何优化了，因而屎山就诞生了

我认为 `不要提前优化` 这句话是产生在一个朝九晚五不加班需求少有充足时间做技术优化的语境之下，这种语境下，这句话是没啥问题的，只是大部分情况下，现实情况根本不符合语境，所以这句话就有问题了

该拆的组件、该提取的公共方法、该规划的目录结构、该引入的代码规范……应该从一开始就形成，**不要等着需要优化的时候才优化，那个时候已经来不及了**

防御性编程

前端最接近用户，意味着前端代码在很大程度上影响用户体验，我们应该尽可能保证页面的稳定运行-哪怕是碰到不符合预期的场景也能让页面平稳落地

这并不需要多么高深技术，只需要付出20%的精力就可以解决 80% 的页面异常情况

例如，设置图片尺寸的时候，考虑这个图片的尺寸可能会出现极端情况-宽高比过大或过小，这个时候应该怎样编写更好的 `css` 属性，还例如，展示一段文案的时候，需要考虑这段文案可能会非常长导致换行甚至换了好几行的情况

js 方面，对于可能抛出错误的代码进行 `try..catch` 处理；从一个变量取值取值的时候，考虑是否可能出现 `Cannot read properties of undefined` 情况，这种错误在js代码中很常见，通过 可选链操作符可轻松解决这个问题。不过我想说的是，不要滥用这个东西，只对你不确定的变量使用这个东西，例如接口的返回值，对于100%确定不会有问题的变量（例如你刚刚定义的、不可能在其他地方被修改的变量）就不要使用了，否则平白增加代码体积不说，还会给其他人造成迷惑

复用（组件、方法）

代码复用是为了提升工作效率，但如果只是为了复用代码而复用，就本末倒置了

通用方法、通用组件鼓励复用，但**业务逻辑、业务组件，慎重复用**

一个常见的例子是，移动端详情页页面和编辑页可能具有大部分重合的逻辑，但类似这种业务属性很强的组件，除非你确信这个组件将来不出现大的改动，否则不要为了贪图眼前的便利而想当然地进行复用

本来为了区分展示态和编辑态，就已经写了一些条件语句了，日后若是出现了已经复用的逻辑必须要按照业务需求进行拆分，甚至是逻辑完全南辕北辙，初期还好，或许还能抢救一下拆分出来，但到了中后期才发现这个问题很可能已经晚了，掺杂了那么多的业务逻辑，你还敢去做拆分吗？那么这个复用组件的代码量必然要被大量的 `if...else` 占领，修改任何一个功能点、排查任何问题都要兼顾两套逻辑，对于维护者来说，这会造成相当大的心智负担，对于项目本身来说，维护的代码将会变得更大

业务代码是千变万化的，原本多个场景下相似的逻辑，很可能随着业务的迭代变得毫无关系，在这种场景下，复用不仅不能提高工作效率，反而还会拖后腿

而对于通用方法和通用组件来说，为了更加彻底地解耦，其应当是**函数式**的，不应当对外部状态产生隐式地修改，也不应当依赖外部可变的变量（例如 `vuex` ）

通用方法最好是纯函数，相同的输入有相同的输出，入参、出参都应当是明确的，让人一眼就看出来需要哪些入参，又会有哪些出参，而不是直接传入一个大的对象，然后在方法体内去一个个查找所需的对象属性

通用组件不应当自作主张修改外部数据，而应该将产生的变化主动抛出去，让上一层组件来明确决定如何使用这个变化

通过配置 alias 引用文件

在多个文件间引用组件/变量是一件司空见惯的事情，很多人可能是为了简单或者习惯使然，倾向于以当前文件为基础，然后通过 `../` 或者 `./` 来将路径定向到目标文件，从而实现对目标文件的引用

想象这样一种场景，项目中有多个文件引入了同一个文件 `btn-group.tsx` ，由于它相对于不同的文件路径可能都是不一样的，所以引入的路径都是不一样的，甚至存在非公共组件被其他组件引用的情况（当初认为这个组件可能不会被其他组件引用，所以没放到公共组件文件夹里，但是后来出现了这种情况）；还可能存在在不同的路径下，存在相同名称的文件，例如 `app/constant/btn-group.tsx` 、 `app/page/home/btn-group.tsx` ，并且由于项目较大，所以这种情况不仅只存在 `btn-group.tsx` 身上，还有好几个文件都面临这样的问题

至此，好像也没啥问题，但当你想修改 `btn-group.tsx` 的时候，问题就来了，你如何确保你的修改不会对所有引用这个文件的文件产生影响？一般情况下就是检查所有引用了这个文件的组件，看下是否产生了预料之外的影响，那么如何找出所有引用了这个文件的组件？

全局搜引用路径？可是大家都用相对引用，引用路径全都不一样无法根据单一的引用路径搜索；根据文件名？可是这个项目中存在多个 `btn-group.tsx` ，你不能保证你搜到的 `btn-group.tsx` 就是你修改的那个 `btn-group.tsx`

当然，你完全可以在通过二次确认来确保你找到的文件都是对的，但这未免是额外的心智负担，在大型项目中，一不留神可能就出现遗漏了

我的建议是，配置 `alias`，然后所有的引用全部以 `alias` 为基准路径进行引用（同一个文件夹下的最好也遵守这个规则），这样一来，对于项目中的任何文件，它的引用路径只可能有一种情况

这么做带来的好处除了搜索文件更清晰之外，还能让引用更加简单，引用文件再也不用一点点地拼接路径了，它的路径就是唯一的，你只需要从其他引用这个文件的地方复制一份即可而不用担心引用的相对路径问题

> 一般的编辑器，例如 `vscode`，是可以直接复制文件相对于项目根路径的相对路径的（`Copy Relative Path`）

依据社区而不是从心

为项目选择设计模式、UI组件库、状态管理库等基础功能的时候，应当选取社区内热度更高的而非根据个人的喜好

你所认为很牛x的设计模式、第三方库等，可能是其他人根本就没听过的，或者其他人根本就不认同的，这只会增加团队之间的协作难度

团队合作项目的代码是用来传承的而不是用来炫技的

抛弃惯性思维

待在舒适区，这是人之本能，因为熟悉，所以上一个项目使用的技术栈在下一个项目里也要继续用

但是，真的合适吗？

上一个项目用了 `mobx`，这个项目里也必须要用吗？上一个项目里将所有的状态数据都放到了状态管理库里，这个项目也要这样做吗？上一个项目没用 `TypeScript`，这一个也不用吗？

可能是不需要的，可能是需要更换的，当然，并不是说就要跟上一个项目反着来，到底怎样最起码要有一个思考的过程，而不是上一个项目就是这样的，所以这一个也要这样

考虑清楚了再写TODO

有意识做优化是个好习惯，但意识得能落到实处

以我的经验看，在多人协作的、业务敏捷迭代的项目中，大多数 `todo` 是无法完成的

大部分 `todo` 都是基于当时的情况做出的考量，当时这个方法可能只有几行， `todo` 要做的时候很简单，但是当时没有做，当过了一段时间再想起来这事的时候，发现那个方法已经变成了几百行了，你还敢动吗?

或者换句话说，你还有完成这个 `todo` 的心思吗？人都是懒惰的，你愿意将原本可以用在打游戏刷视频的时间用在完成这个 `todo` 上吗？看到了别人写的 `todo` ，并且也看明白了，但是你愿意帮别人完成这个 `todo` 吗?

该做的事情应当立即完成，或许因为某些原因无法立即完成,所以你想延后再来做，但是一般情况下，后续再来完成的成本必然大于当下，现在都完成不了，凭什么认为以后就能完成?

好，就算你责任心强，动力十足，但问题是，你如何保证你在昨天或者上周或者上个月写的 `todo` ，还能在百忙之中的今天想起来呢？手头上的事情都忙得焦头烂额，还能想得起来已经过去的事情?

真的需要做的事情，哪怕会让进度延期，只要你理由充分，其他人不可能也没理由去阻止你

小结

很多时候，一些让你能够写出更好的代码建议，实际上对于业务产出是毫无帮助的，哪怕你不遵守这些建议甚至反着来，也不影响你的产出不影响你的绩效，毕竟产出和绩效跟代码写得好不好并没有直接关系，甚至这些所谓的建议有时候还会影响你快速产出，只要我能拿出一个好的产出拿到一个好的绩效，代码写得糙点烂点又有什么关系？以后的事情以后再说呗，搞不好以后维护的人根本不是我

这种心理或许才是常态，毕竟这更加符合现实的利益

但如果你是一位对技术有追求的人，你真的甘心就如此吗？我认为除了现实的考量之外，还应当为自己写下的代码负责