一定要优雅，高端前端程序员都应该具备的基本素养

为了能让更多人认识到这一点，站在前端的角度上，我在仔细拜读了项目中的那些暗藏玄机的代码后，决定写下此文，由于本人功力尚浅，且之前一直走在错误的道路上，所以本文在真正的高手看来可能有些班门弄斧，在此献丑了 

用 TypeScript，但不完全用

`TypeScript` 大行其道，在每个团队中，总有那么些个宵小之辈想尽一切办法在项目里引入 `ts` ，这种行为严重阻碍了屎山的成长速度，但同是打工人我们也不好阻止，不过就算如此，也无法阻止我们行使正义

众所周知， `TypeScript` 别名 `AnyScript` ，很显然，这就是 `TypeScript` 创始人 `Anders Hejlsberg` 给我们留下的暗示，我们有理由相信 `AnyScript` 才是他真正的目的

ts 复制代码

```ts
const list: any = []
const obj: any = {}
const a: any = 1
```

引入了 `ts` 的项目，由于是在原可运行代码的基础上额外添加了类型注释，所以代码体积毫无疑问会增大，有调查显示，可能会增加 `30%` 的代码量，如果充分发挥 `AnyScript` 的宗旨，意味着你很轻松地就让代码增加了 `30%` 毫无用处但也挑不出啥毛病的代码，这些代码甚至还会增加项目的编译时间（毕竟增加了 `ts` 校验和移除的成本嘛）

你不仅能让自己写的代码用上 `AnyScript` ，甚至还可以给那些支持 `ts` 的第三方框架/库一个大嘴巴子

ts 复制代码

```ts
export default defineComponent({
  props: {
    // 现在 data 是 any 类型的啦
    data: {
      type: Number as PropType<any>,
    },
  },
  setup(_, { emit }) {
    // 现在 props 是 any 类型的啦
```

```
    const props: any = _
    ...
  }
})
```

当然了，全屏 any 可能还是有点明显了，所以你可以适当地给部分变量加上具体类型，但是加上类型不意味着必须要正确使用

ts 复制代码

```ts
const obj: number[] = []
// ...
// 虽然 obj 是个 number[]，但为了实现业务，就得塞入一些不是 number 的类型，我也不想的啊是不是
// 至于编辑器会划红线报错？那是小问题，不用管它，别人一打开这个项目就是满屏的红线，想想就激动
obj.push('2')
obj.push([3])
```

命名应该更自由

命名一直是个困扰很多程序员的问题，究其原因，我们总想给变量找个能够很好表达意思的名称，这样一来代码的可阅读性就高了，但现在我们知道，这并不是件好事，所以我们应该放纵自我，既摆脱了命名困难症，又加速了屎山的堆积进度

ts 复制代码

```ts
const a1 = {}
const a2 = {}
const a3 = 2
const p = 1
```

我必须强调一点，命名不仅是变量命名，还包含**文件名、类名、组件名**等，这些都是我们可以发挥的地方，例如类名

html 复制代码

```html
<div class="box">
  <div class="box1"></div>
  <div class="box2"></div>
<div>
<div class="box3"></div>
```

乍一看似乎没啥毛病，要说有毛病似乎也不值当单独挑出来说，没错，要的就是这个效果，让人单看一段代码不好说什么，但是如果积少成多，整个项目都是 box 呢？全局搜索都给你废

了！如果你某些组件再一不小心没用 `scoped` 呢？稍不留意就不知道把什么组件的样式给改了，想想就美得很

关于 `css` 我还想多说一点，鉴于其灵活性，我们还可以做得更多，总有人说什么 `BEM` 不 `BEM` 的，他们敢用我们就敢写这样的代码

scss 复制代码

```scss
&-card {
  &-btn {
    &_link {
      &--right {
      }
    }
    &-nodata {
      &_link {
        &--replay {
          &--create {}
        }
      }
    }
  }
  &-desc {}
}
```

好了，现在请在几百行（关于这一点下一节会说到）这种格式的代码里找出类名 `.xxx__item_current.mod-xxx__link` 对应的样式吧

代码一定要长

屎山一定是够高够深的，这就要求我们的代码应该是够长够多的

大到一个文件的长度，小到一个类、一个函数，甚至是一个 `if` 的条件体，都是我们自由发挥的好地方。

什么单文件最好不超过 `400` 行，什么一个函数不超过 `100` 行，简直就是毒瘤，

所以这就要求我们要具备将十行代码就能解决的事情写成一百行的能力，最好能给人一种多即是少的感觉

ts 复制代码

```ts
data === 1
  ? 'img'
  : data === 2
    ? 'video'
    : data === 3
      ? 'text'
      : data === 4
        ? 'picture'
        : data === 5
          ? 'miniApp'
```

三元表达式可以优雅地表达逻辑，像诗一样，虽然这段代码看起来比较多，但逻辑就是这么多，我还专门用了三元表达式优化，不能怪我是不是？什么 `map` 映射枚举优化听都没听过

你也可以选择其他一些比较容易实现的思路，例如，多写一些废话

ts 复制代码

```ts
if (a > 10) {
  // 虽然下面几个 if 中对于 a 的判断毫无用处，但不仔细看谁能看出来呢？看出来了也不好说什么，毕竟也没啥错
  // 除此之外，多级 if 嵌套也是堆屎山的一个小技巧，什么提前 return 不是太明白
  if (a > 5) {
    if (a > 3 && b) {

    }
  }
  if (a > 4) {

  }
}
```

除此之外，你还可以写一些中规中矩的方法，但重点在于这些方法根本就没用到，这种发挥的地方就更多了，简直就是扩充代码体积的利器，毕竟单看这些方法没啥毛病，但谁能想到根本

就用不到呢？就算有人怀疑了，但你猜他敢随便从运行得好好的业务项目里删掉一些没啥错的代码吗？

一次编写，到处引用

大家应该都知道，一个公共的方法或变量最好放在公共文件夹中，这样方便引用，提升聚合度，本来这种做法是会极大阻碍屎山进程的，但俗话说事在人为，哪有什么绝对正确的东西，不过看是谁在当搅屎棍罢了

例如在一个项目中，分为好几个页面，首页和详情页，我在写详情页的时候，发现有一个变量在首页已经存在了，一般做法是把这个变量提升到公共文件中去，这样将来如果涉及到各自修改的话，能做到心中有数，但我偏不，我TM直接在详情页里引用首页的变量

我不仅在业务组件里引用，甚至还可以在公共组件里引用，一般做法是业务组件引用公共组件/方法/变量，嘿，我偏要公共组件引用业务组件组件/方法/变量

这样导致的后果就是，一个本该私有的变量/方法，被不知道多少个页面组件、模块组件、公共文件所引用，万一需要修改，处理不好的话直接塌方，任何一个参与改动的人都战战兢兢如履薄冰，要么就是复制粘贴重写一份，要么就是要写很多没少意义的兼容逻辑，无论选择哪种，都将成为屎山的忠实贡献者

组件、方法多多滴耦合

为了避免其他人复用我的方法或组件，那么在写方法或组件的时候，一定要尽可能耦合，提升复用的门槛

例如明明可以通过 `Props` 传参解决的事情，我偏要从全局状态里取，例如 `vuex` ，独一份的全局数据，想传参就得改 `store` 数据，但你猜你改的时候会不会影响到其他某个页面某个组件的正常使用呢？如果你用了，那你就可能导致意料之外的问题，如果你不用你就得自己重写一个组件

组件不需要传参？没关系，我直接把组件的内部变量给挂到全局状态上去，虽然这些内部变量确实只有某一个组件在用，但我挂到全局状态也没啥错啊是不是

嘿，明明一个组件就能解决的事情，现在有了俩，后面还可能有仨，这代码量不就上来了吗？

方法也是如此，明明可以抽取参数，遵循函数式编程理念，我偏要跟外部变量产生关联

ts 复制代码

```ts
// 首先这个命名就很契合上面说的自由命名法
function fn1() {
  // ...
  // fn1 的逻辑比较长，且解决的是通用问题，
  // 但 myObj 偏偏是一个外部变量，这下看你怎么复用
  window.myObj.name = 'otherName'
  window.myObj.children.push({ id: window.myObj.children.length })
  // ...
}
```

翻译翻译，什么叫 mutable

翻译翻译?

好，我翻译下

> ~~Immutable Data 就是一旦创建，就不能再被更改的数据~~
>
> Mutable Data 就是一旦创建，就能随时被更改的数据

得益于 javascript 的灵活性， un-immutable 大大滴有市场，借助这把利器，我们可以让数据的流动如空气般无处不在，如何使用 Mutable Data 相信很多初学者早已无师自通，我就不多说了，只希望你们莫忘初心

难的是，能在使用这把利器的同时，还能驾驭好它

如何做到？我们以 vue 为例的话，那么只需要三个单词： Watch 、 Watch 还是踏马的 Watch ！

js 复制代码

```js
export default {
  watch: {
    '$route.name': fuction(){
        this.handleChange()
    },
    'userName': fuction(){
        this.handleChange()
    },
    'dateTime': fuction(){
        this.handleChange()
    },
    'color': fuction(){
```

```
        this.handleChange()
    },
    'queryKey': fuction(){
        this.handleChange()
    },
  },
  methods: {
    handleChange() {
      // ...
    }
  }
}
```

当然，如果你用 `vue-ts` 的话，就更省心了

ts 复制代码

```ts
@Watch('$route.name')
@Watch('userName')
@Watch('dateTime')
@Watch('color')
@Watch('queryKey')
handleChange() {
  // ...
}
```

至于 `react`，无非是换成 `useEffect`，总之本质上都是踏马的 `watch`，能 `mutable` 的绝不 `immutable`，能 `watch` 的绝不主动调用！

注意，千万不要加 `debounce`，想象一下，`watch` 的所有字段可能会同时触发，而 `handleChange` 恰好又是一个耗性能的方法……

魔术字符串是个好东西

实际上，据我观察，排除掉某些居心不轨的人之外，大部分人还是比较喜欢写魔术字符串的，这让我很欣慰，看着满屏的不知道从哪里冒出来也不知道代表着什么的硬编码字符串，让人很有安全感

ts 复制代码

```ts
if (a === 'prepare') {
  const data = localStorage.getItem('HOME-show_guide')
  // ...
} else if (a === 'head' && b === 'repeating-error') {
  switch(c) {
    case 'pic':
```

```
      // ...
      break
    case 'inDrawer':
      // ...
      break
  }
}
```

基于此，我们还可以做得更多，比如用变量拼接魔术字符串， debug 的时候直接废掉全局搜索

ts 复制代码

```ts
if (a === query.name + '_head') {

}
```

大家都是中国人，为什么不试试汉字呢?

ts 复制代码

```ts
if (data === '正常') {

} else if (data === '错误') {

} else if (data === '通过') {

}
```

一般人认为魔术字符串就是写死的字符串，但其实还可以玩得更魔幻一点

ts 复制代码

```ts
enum EventType {
  Move,
  Skip,
  Batch
}
```

枚举是个好东西，可以避免直接书写无意义的字符，但前提是你真的是按照正常人思维来使用的 正常人是怎么用的呢？例如，我们有一段 vue 逻辑就是用判断这个枚举值的

ts 复制代码

```ts
handleEvent(value: EventType) {
    if (value === EventType.Move) {
      // ...
    } else if (value === EventType.Skip) {
      // ...
    } else if (value === EventType.Batch) {
      // ...
```

```
    }
}
```

看着没啥问题，挺好的，但在模板里，诶，我嫌麻烦，我这样写

vue 复制代码

```
<template>
  <div @click="handleEvent(2)">确定</div>
<template>
```

`2` 是什么？写这段代码的人那肯定知道啊，就是 `EventType.Batch` 的默认值，但是我不跟别人说我也不写注释，后面人看到 `EventType` 的定义只会想到这是个枚举定义，正常人是不可能想到这些枚举的默认值居然还会以魔术字符串的形式被用到

那么想一下，某一天，一个倒霉蛋想改这段代码，他想在 `EventType` 中新增一个枚举，为了以防万一改到其他位置，他搜遍了整个项目确认了每个项目中用到 `EventType` 的地方，发现自己的改动不会有任何问题，于是他在 `EventType` 的开头加上了自己的内容

ts 复制代码

```ts
enum EventType {
  Clown,
  Move,
  Skip,
  Batch
}
```

好了兄弟们，`<div @click="handleEvent(2)">确定</div>` 这段逻辑不声不响地完蛋了！

轮子就得自己造才舒心

众所周知，造轮子可以显著提升我们程序员的技术水平，另外由于轮子我们已经自己造了，所以减少了对社区的依赖，同时又增加了项目体积，有力地推动了屎山的成长进程，可以说是一鱼两吃了

例如我们可能经常在项目中使用到时间格式化的方法，一般人都是直接引入 `dayjs` 完事，太肤浅了，我们应该自己实现，例如，将字符串格式日期格式化为时间戳

ts 复制代码

```ts
function format(str1: any, str2: any) {
  const num1 = new Date(str1).getTime()
  const num2 = new Date(str2).getTime()
```

```
  return (num2 - num1) / 1000
}
```

多么精简多么优雅，至于你说的什么格式校验什么 `safari` 下日期字符串的特殊处理，等遇到了再说嘛，就算是 `dayjs` 不也是经过了多次 `fixbug` 才走到今天的嘛，多一些宽松和耐心好不好啦

如果你觉得仅仅是 `dayjs` 这种小打小闹难以让你充分发挥，你甚至可以造个 `vuex` ， `vue` 官网上写明了 `eventBus` 可以充当全局状态管理的，所以我们完全可以自己来嘛，这里就不举例了，这是自由发挥的地方，就不局限大家的思路了

借助社区的力量-轮子还是别人的好

考虑到大家都只是混口饭吃而已，凡事都造轮子未免有些强人所难，所以我们可以尝试走向另外一个极端——凡事都用轮子解决

判断某个变量是字符串还是对象，kind-of拿来吧你；获取某个对象的 `key` ，object-keys拿来吧你；获取屏幕尺寸，vue-screen-size拿来吧你……等等，就不一一列举了，需要大家自己去发现

先甭管实际场景是不是真的需要这些库，也甭管是不是杀鸡用牛刀，要是大家听都没听过的轮子那就更好了，这样才能彰显你的见多识广，总之能解决问题的轮子就是好问题，

在此我得特别提点一下 `lodash` ，这可是解决很多问题的利器，但是别下载错了，得是commonjs版本的那个，量大管饱还正宗，es module版本是不行滴，太小家子气

ts 复制代码

```
import _ from 'lodash'
```

多尝试不同的方式来解决相同的问题

世界上的路有很多，很多路都能通往同一个目的地，但大多数人庸庸碌碌，只知道沿着前人的脚步，没有自己的思想，别人说啥就是啥，这种行为对于我们程序员这种高端的职业来说，坏处很大，任何一个有远大理想的程序员都应该避免

落到实际上来，就是尝试使用不同的技术和方案解决相同的问题

- 搞个 `css` 模块化方案，什么 `BEM` 、 `OOCSS` 、 `CSS Modules` 、 `CSS-in-JS` 都在项目里引入，紧跟潮流扩展视野
- `vue` 项目只用 `template` ？逊啦你， `render` 渲染搞起来
- 之前看过什么前端依赖注入什么反射的文章，虽然对于绝大多数业务项目而言都是水土不服，但问题不大，能跑起来就行，引入引入
- 还有那什么 `rxjs` ，人家都说好，虽然我也不知道好在哪里，但胜在门槛高一般人搞不清楚所以得试试
- `Pinia` 是个好东西，什么，我们项目里已经有 `vuex` 了？ `out` 啦，人家官网说了 `vue2` 也可以用，我们一定要试试，紧跟社区潮流嘛，一个项目里有两套状态管理有什么值得大惊小怪的！

做好自己，莫管他人闲事

看过一个小故事，有人问一个年纪很大的老爷爷的长寿秘诀是什么，老爷爷说是从来不管闲事

这个故事对我们程序员来说也很有启发，写好你自己的代码，不要去关心别人能不能看得懂，不要去关心别人是不是会掉进你写的坑里

ts 复制代码

```ts
mounted() {
  setTimeout(() => {
    const width = this.$refs.box.offsetWidth
    const itemWidth = 50
    // ...
  }, 200)
}
```

例如对于上述代码，为什么要在 `mounted` 里写个 `setTimeout` 呢？为什么这个 `setTimeout` 的时间是 `200` 呢？可能是因为 `box` 这个元素大概会在 `mounted` 之后的 `200ms` 左右接口返回数据就有内容了，就可以测量其宽度进行其他一系列的逻辑了，至于有没有可能因为网络等原因超过 `200ms` 还是没有内容呢？这些不需要关心，你只要保证在你开发的时候 `200ms` 这个时间是没问题的就行了； `itemWidth` 代表另外一个元素的宽度，在你写代码的时候，这个元素宽度就是 `50` ，所以没必要即时测量，你直接写死了，至于后面其他人会不会改变这个元素的宽度导致你这里不准了，这就不是你要考虑的事情了，你开发的时候确实没问题，其他人搞出来问题其他人负责就行，管你啥事呢？

代码自解释

高端的程序员，往往采用最朴素的编码方式，高手从来不写注释，因为他们写的代码都是自解释的，什么叫自解释？就是你看代码就跟看注释一样，所以不需要注释

我觉得很有道理，代码都在那里搁着了，逻辑写得清清楚楚，为啥还要写注释呢，直接看代码不就行了吗？

乍一看，似乎这一条有点阻碍堆屎山的进程，实则不然

一堆注定要被迭代无数版、被无数人修改、传承多年的代码，其必定是逻辑错综复杂，难免存在一些不可名状的让人说不清道不明的逻辑，没有注释的加成，这些逻辑大概率要永远成为黑洞了，所有人看到都得绕着走，相当于是围绕着这些黑洞额外搭起了一套逻辑，这代码体积和复杂度不就上来了吗？

如果你实在手痒，倒也可以写点注释，我这里透露一个既能让你写写注释过过瘾又能为堆屎山加一把力的方法，那就是：在注释里撒谎！

没错，谁说注释只能写对的？我理解不够，所以注释写得不太对有什么奇怪的吗？我又没保证注释一定是对的，也没逼着你看注释，所以你看注释结果被注释误导写了个bug，这凭啥怪我啊

ts 复制代码

```ts
// 计算 data 是否可用
//（实际上，这个方法的作用是计算 data 是否 不可用）
function isDisabledData(data: any) {
  // ...
}
```

上述这个例子只能说是小试牛刀，毕竟多调试一下很容易被发现的，但就算被发现了，大家也只会觉得你只是个小粗心鬼罢了，怎么好责怪你呢，这也算是给其他人的一个小惊喜了，况且，万一真有人不管不顾就信了，那你就赚大了

编译问题坚决不改

为了阻碍屎山的成长速度，有些阴险的家伙总想在各种层面上加以限制，例如加各种 `lint` ，在编译的时候，命令行中就会告诉你你哪些地方没有按照规则来，但大部分是 `waring` 级别的，即你不改项目也能正常运行，这就是我们的突破点了。

尽管按照你的想法去写代码， `lint` 的事情不要去管， `waring` 报错就当没看到，又不是不能用？在这种情况下，如果有人不小心弄了个 `error` 级别的错误，他面对的就是从好几屏的

`warning` 中找他的那个 `error` 的场景了，这就相当于是提前跟屎山来了一次面对面的拥抱

根据破窗理论，这种行为将会影响到越来越多的人，大家都将心照不宣地视 `warning` 于无物（从好几屏的 `warning` 中找到自己的那个实在是太麻烦了），所谓的 `lint` 就成了笑话

小结

一座历久弥香的屎山，必定是需要经过时间的沉淀和无数人的操练才能最终成型，这需要我们所有人的努力，多年之后，当你看到你曾经参与堆砌的屎山中道崩殂轰然倒塌的时候，你就算是真的领悟了我们程序员所掌控的恐怖实力！🐶