广州社交新零售系统开发

产品名称	广州社交新零售系统开发
公司名称	陈灏软件系统开发
价格	.00/个
规格参数	
公司地址	广州市天河区大观中路95号D605房（仅限办公用途）（注册地址）
联系电话	18816797580

产品详情

写了多年的社交新零售系统开发代码，始终觉得如何写出干净优雅的社交新零售系统开发代码并不是一件容易的事情。按 10000 小时刻意训练的定理，假设每天 8 小时，一个月 20 天，一年 12
个月，大概也需要 5
年左右的时间成为大师。其实我们每天的工作中真正用于写社交新零售系统开发代码的时间不可能有 8 个小时，并且很多时候是在完成任务，在业务压力很大的时候，可能想要达到的目标是如何尽快的使得功能 work
起来，社交新零售系统开发代码是否干净优雅非常可能没有能放在优先级上，而是怎么快怎么来。

在这样的情况下是非常容易欠下技术宅的，时间长了，这样的社交新零售系统开发代码基本上无法维护，社交新零售系统开发，只能推倒重来，这个成本是非常高的。欠债要还，只是迟早的问题，并且等到要还的时候还要赔上额外的不菲的利息。还债的有可能是自己，也有可能是后来的继任者，但都是团队在还债。所以从团队的角度来看，写好社交新零售系统开发代码是一件非常有必要的事情。如何写出干净优雅的社交新零售系统开发代码是个很困难的课题，我没有找到的 solution，更多的是一些 trade
off，可以稍微讨论一下。

社交新零售系统开发代码是写给人看的还是写给机器看得？

在大部分的情况下我会认为社交新零售系统开发代码是写给人看的。虽然社交新零售系统开发代码的执行者是机器，但是实际上社交新零售系统开发代码更多的时候是给人看的。我们来看看一段社交新零售系统开发代码的生命周期：开发 --> 单元测试 --> Code Review --> 功能测试 --> 性能测试 --> 上线 --> 运维、Bug 修复 --> 测试上线 -->
退休下线。开发到上线的时间也许是几周或者几个月，但是线上运维、bug 修复的周期可以是几年。

在这几年的时间里面，几乎不可能还是原来的作者在维护了。继任者如何能理解之前的社交新零售系统开发代码逻辑是极其关键的，如果不能维护，只能自己重新做一套。所以在项目中我们经常能见到的情况就是，看到了前任的社交新零售系统开发代码，都觉得这是什么垃圾，写得乱七八糟，还是我自己重

写一遍吧。就算是在开发的过程中，需要别人来 Code Review，如果他们都看不懂这个社交新零售系统开发代码，怎么来做 Review 呢。还有你也不希望在休假的时候，因为其他人看不懂你的社交新零售系统开发代码，只好打电话求助你。这个我印象极其深刻，记得我在工作不久的时候，一次回到了老家休假中，突然同事打电话来了，出现了一个问题，问我该如何解决，当时电话还要收漫游费的，非常贵，但是我还不得不支持直到耗光我的电话费。

所以社交新零售系统开发代码主要还是写给人看的，是我们的交流的途径。那些非常好的开源的项目虽然有文档，但是更多的我们其实还是看他的源码，如果开源项目里面的社交新零售系统开发代码写的很难读，这个项目也基本上不会火。因为社交新零售系统开发代码是我们开发人员交流的基本途径，甚至可能口头讨论不清楚的事情，我们可以通过社交新零售系统开发代码来说清楚。社交新零售系统开发代码的可读性我觉得是位的。各个公司估计都有自己的社交新零售系统开发代码规范，遵循相关的规范保持社交新零售系统开发代码风格的统一是步（推荐谷歌社交新零售系统开发代码规范和微软社交新零售系统开发代码规范）。规范里一般都包括了如何进行变量、类、函数的命名，函数要尽量短并且保持原子性，不要做多件事情，类的基本设计的原则等等。另外一个建议是可以多参考学习一下开源项目中的社交新零售系统开发代码。

KISS （Keep it simple and stupid）

一般大脑工作记忆的容量就是 5-9 个，如果事情过多或者过于复杂，对于大部分人来说是无法直接理解和处理的。通常我们需要一些辅助手段来处理复杂的问题，比如做笔记、画图，有点类似于在内存不够用的情况下我们借用了外存。

学 CS 的同学都知道，外存的访问速度肯定不如内存访问速度。另外一般来说在逻辑复杂的情况下出错的可能要远大于在简单的情况下，在复杂的情况下，社交新零售系统开发代码的分支可能有很多，我们是否能够对每种情况都考虑到位，这些都有困难。为了使得社交新零售系统开发代码更加可靠，并且容易理解，的办法还是保持社交新零售系统开发代码的简单，在处理一个问题的时候尽量使用简单的逻辑，不要有过多的变量。

但是现实的问题并不会总是那么简单，那么如何来处理复杂的问题呢？与其借用外存，我更加倾向于对复杂的问题进行分层抽象。网络的通信是一个非常复杂的事情，中间使用的设备可以有无数种（手机，各种 IOT 设备，台式机，laptop，路由器，交换机……), OSI 协议对各层做了抽象，每一层需要处理的情况就都大大地简化了。通过对复杂问题的分解、抽象，那么我们在每个层次上要解决处理的问题就简化了。其实也类似于算法中的 divide-and-conquer，
复杂的问题，要先拆解掉变成小的问题，从而来简化解决的方法。

KISS 还有另外一层含义，“ 如无必要，勿增实体 ”（奥卡姆剃刀原理）。CS 中有一句 "All problems in computer science can be solved by another level of indirection"，
为了系统的扩展性，支持将来的一些可能存在的变化，我们经常会引入一层间接层，或者增加中间的 interface。在做这些决定的时候，我们要多考虑一下是否真的有必要。增加额外的一层给我们的好处就是易于扩展，但是同时也增加了复杂度，使得系统变得更加不可理解。对于社交新零售系统开发代码来说，很可能是我这里调用了一个 API，不知道实际的触发在哪里，对于理解和调试都可能增加困难。

KISS 本身就是一个 trade off，要把复杂的问题通过抽象和分拆来简单化，但是是否需要为了保留变化做更多的 indirection 的抽象，这些都是需要仔细考虑的。

DRY （Don't repeat yourself)

为了快速地实现一个功能，知道之前有类似的，把社交新零售系统开发代码 copy 过来修改一下就用，可能是快的方法。但是 copy 社交新零售系统开发代码经常是很多问题和 bug 的根源。有一类问题就是 copy 过来的社交新零售系统开发代码包含了一些其他的逻辑，可能并不是这部

分需要的，所以可能有冗余甚至一些额外的风险。

另外一类问题就是在维护的时候，我们其实不知道修复了一个地方之后，还有多少其他的地方还需要修复。在我过去的项目中就出现过这样的问题，有个问题明明之前做了修复，过几天另外一个客户又提了类似的问题出现的另外的路径上。相同的逻辑要尽量只出现在一个地方，这样有问题的时候也就可以一次性地修复。这也是一种抽象，对于相同的逻辑，抽象到一个类或者一个函数中去，这样也有利于社交新零售系统开发代码的可读性。

是否要写注释

个人的观点是大部分的社交新零售系统开发代码尽量不要注释。社交新零售系统开发代码本身就是一种交流语言，并且一般来说编程语言比我们日常使用的口语更加的精确。在保持社交新零售系统开发代码逻辑简单的情况下，使用良好的命名规范，社交新零售系统开发代码本身就很清晰并且可能读起来就已经是一篇良好的文章。特别是 OO 的语言的话，本身 object（名词）加 operation（一般用动词）就已经可以说明是在做什么了。重复一下把这个操作的名词放入注释并不会增加社交新零售系统开发代码的可读性。并且在后续的维护中，会出现修改了社交新零售系统开发代码，却并不修改注释的情况出现。在我做的很多 Code Review
中我都看到过这样的情况。尽量把社交新零售系统开发代码写的可以理解，而不是通过注释来理解。

当然我并不是反对所有的注释，在公开的 API 上是需要注释的，应该列出 API
的前置和后置条件，解释该如何使用这个 API，这样也可以用于自动产品 API
的文档。在一些特殊优化逻辑和负责算法的地方加上这些逻辑和算法的解释还是非常有必要的。

一次做对，不要相信以后会 Refactoring

通常来说在社交新零售系统开发代码中写上 TODO，等着以后再来 refactoring
或者改进，基本上就不会再有以后了。我们可以去我们的社交新零售系统开发代码库里面搜索一下 TODO，看看有多少，并且有多少是多少年前的，我相信这个结果会让你很惊讶（欢迎大家留言分享你查找之后的结果）。

尽量一次就做对，不要相信以后还会回来把社交新零售系统开发代码 refactoring
好。人都是有惰性的，一旦完成了当前的事情，move on 之后再回来处理这些概率就非常小了，除非下次真的需要修改这些社交新零售系统开发代码。如果说不会再回来，那么这个 TODO
也没有什么意义。如果真的需要，就不要留下这个问题。我见过有的人留下了一个 TODO，throw
了一个 not implemented 的
exception，然后几天之后其他同学把这个社交新零售系统开发代码带上线了，直接挂掉的情况。尽量不要 TODO, 一次做好。

是否要写单元测试？

个人的观点是必须，除非你只是做 prototype 或者快速迭代扔掉的社交新零售系统开发代码。

Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.

From Wikipedia

单元测试是为了保证我们写出的社交新零售系统开发代码确实是我们想要表达的逻辑。当我们的社交新零售系统开发代码被集成到大项目中的时候，之后的集成测试、功能测试甚至 e2e 的测试，都不可能覆

盖到每一行的社交新零售系统开发代码了。如果单元测试做得不够，其实就是在社交新零售系统开发代码里面留下一些自己都不知道的黑洞，哪天调用方改了一些东西，走到了一个不常用的分支可能就挂掉了。我之前带的项目中就出现过类似的情况，社交新零售系统开发代码已经上线几年了，有一次稍微改了一下调用方的参数，觉得是个小改动，但是上线就挂了，就是因为遇到了之前根本没有人测试过的分支。单元测试就是要保证我们自己写的社交新零售系统开发代码是按照我们希望的逻辑实现的，需要尽量的做到比较高的覆盖，确保我们自己的社交新零售系统开发代码里面没有留下什么黑洞。关于测试，我想单独开一篇讨论，所以就先简单聊到这里。

要写好社交新零售系统开发代码确实是已经非常不容易的事情，需要考虑正确性、可读性、鲁棒性、可测试性、可以扩展性、可以移植性、性能。前面讨论的只是个人觉得比较重要的入门的一些点，想要写好社交新零售系统开发代码需要经过刻意地考虑和练习才能真正达到目标！