# 程序员修炼之道总结及读后感

### 第一章：务实的哲学

#### 务实的重要性：

务实的态度一直在生活和工作被多数人所称赞的一种良好品质，这种务实的态度也关系到自己的职业生涯的发展的进展顺利与否；作为程序员有的时候会想，能否在以前某些方面上在优化、养成什么样的编码习惯、设计到什么地步才具有未来市场需求的扩展性？问题是没有明确的方向和方法去做得更好。读完第一章就知道编码务实的重要性和思考通向务实编码的方式方法。

记得18年在杭州参加某互联网技术交流会上，有位台下听众问台上的阿里大佬：“一直在公司写业务crud，怎样才能提升自己技术能力？”，我想当时台下200多人中有大部分曾经或现在遇到的情况基本类似，除非不写代码，听完大佬的愤愤的回答，就知道这是一种务实心态调整和自身在公司定位的问题。就引用他的一句反问：“业务crud怎么就不能锻炼自己的技术了，阿里的也大部分时间也在每天curd，写业务体现出技术能力的深浅”，我想有这样问题的可能不是不务实，而是不知道怎么去务实自己的编码，从而学习中提升和展现出自己的设计编码价值。下面介绍几种书中我认为重要的需要务实的点：

* 1. 团队信任

“做一个靠谱的人”，开发中团队协作，预研技术等，不仅要获得领导的支持，在需要多方配合时，更是需要同事的支持，在他们眼里，是否足够放心省心的让我去完成任务，值得关注和思考的点。

* 1. 承担责任

遇到问题或者做事不要找借口搪塞敷衍过去，真诚的接受同事或领导指出的问题，当然自己发现及时纠正更好；随后思考问题为什么会出现，缺少某一个环节的把关、缺少某些知识、资源分配存在问题等等，针对性的预防或解决问题，这就是责任心的体现。

* 1. 软件的熵

技术债和软件腐败的问题几乎每个公司做软件开发都会存在，要做到遇到有（糟糕的设计、错误的决定、低劣的代码，多余的代码等）就立马清除，“不要放任破窗”，预防沿着不合理继续前进，结果就是整个项目都在业务扩展能力上衰退，最终难以维护。

核心业务需从地基开始就要花时间设计好，多花点时间在设计上，后期开发扩展才会轻松很多，也就从根基上阻止了软件熵的无序叠加。这里还体现了另外一点：好的编码习惯会让自己省心，特别是要不断网上加业务功能的代码，做到省心又省时。

* 1. 石头做的汤和煮熟的青蛙

这两则寓言故事，让我意识到功能不应该注意力集中在一个点上，要全局考虑，多关注个人以外发生的事情；有公共属性的方法要抽离成全局使用。从另外一个角度看，士兵初始需求就是要煮一锅汤，随着食材的丰富，变成了一大锅丰盛的大餐，软件或者早期需求也许并不丰富，但能把握住需求主要脉络，开发代码时注意基线的规范把控，后期在不偏离主脉络规范的情况添砖加瓦也不是什么难事，最终也能开发出完美的功能软件。

* 1. 够好的软件即可

有的需求需要用户去参与权衡是否做成这样的，这多数产品经理会去画出动静结合的原型图，很少让开发去实现初稿在去改，开发动手开发了，在由于需求变更去修改，就多个方面的成本飙升，确认需求的最好方式还是在产品经理调研需求的阶段，避免需求不明，导致后面埋坑。

开发人员会根据既定需求，去保证时间周期和交付质量。

* 1. 知识组合

“投资知识，收益最佳”这是真理；管理知识组合的诀窍是一开始就这样做，并养成习惯。加入有助于学习的社区群体、每年要求系统性的读技术性和非技术性的书籍书本，学习热门的新技术和新的解决方案。

* 1. 交流

交流，除了工作上交流业务实现之外很少去参加技术沙龙之类的活动。工作上的，其实我们大多数研发都不善于表达自己的想法，缺少语言组织能力，说的不够精准或时机不对等问题；所以大多数情况我们规定了要有文档或者邮件写明白，足够工作中使用了。这些肯定远远不够，不满足未来的发展条件。

### 第二章：务实的方法

#### 优秀设计的精髓

谈起怎么设计优秀的软件，现在很多人都能说出一二来，常见的则是遵循“高类聚、低耦合”、接口或者类设计具有单一职责、微服务化、api依赖方式、rpc服务的互相调用、数据字典、命名需全英文的规范的方式（词能达意）等等，最终的目的都是为了设计出优秀的软件，使之更容易应对未来的变更；

###### 1.1 ETC是一种价值观念，不是一条规则

一种设计意识对于很多大大小小公司来说推行起来是困难的，长期保持下去就更难了。举例：如刚成立两百人开发团队，大伙都来自全国各地的不同年龄层次不同领域的java开发，有着自己对编程的不同理解和行为风格，在团队协作上存在这样或那样的壁垒；怎么办？他说这样对，你说这样对；唯一的办法就是统一标准（借鉴阿里巴巴编程规范、国外的编程规范），刚开始领导推行的时候实施效果明显，到了后面领导没盯着的时候，就开始恢复以前的模样，出现方法上千行，不按照三层架构的各自职责写代码，控制层写业务，冗余代码、重复代码、不抽离成接口公共使用等等问题又开始出现，为什么？因为公共的方法要定义公共属性进行抽离，需要设计，需要不断重构原来方法和类，这种繁琐又会出bug的事会只会让开发先暂避，久而久之还是回到了以前的糟糕设计的样子。书中提到了ETC是一种价值观念，不是一条规则。这似乎可以根治编码设计陋习。

我理解的是：不是一条领导让我们遵守的规则，而是一种编码时时刻刻漂浮在意识思维之上的理念，这种理念微妙的将我们开发时推向正确的方向。这种意识形态会成为我们在设计编码前、完成编码单元测试时、修复bug和需求变更时习惯性的出现，并问自己这样做是让系统更难还是更易。

#### 2 DRY——邪恶的重复

形成第一个意识习惯避免重复：“在一个系统中，没一处知识都必须单一、明确、权威的表达。”书中强调避免重复不仅仅限于编码，便举了一些例子：文档重复、数据中的DRY违规、表征重复、内部api重复、外部api重复、数据源引起的重复、开发人员间的重复。最终要努力达到的方向是，孕育出一个更容易找到和复用已有事物的环境，而不是自己编写。如果复用不容易，人们就不会这么做。如果未能复用，就有重复知识的风险。

代码重复是研发过程中最常见的，用编码规范一扫就会发现很多，要么是数千行代码都在一个方法中不做分离导致的，要么是好的算法或工具类没有养成抽出到公共的plugin中公用导致重复写，重复出bug，经常出现改一个业务点，需要很多地方一起改，害怕改遗漏了一处导致给自己埋坑出问题，到了不熟悉它的业务还真不敢动它的难处，难道这就是标题的意思？有以下几种方法对研发有帮助，总结如下：

①公共工具的api保存到maven的中央仓库，以便不同团队共享。

②类的设计需遵循单一职责，一个方法函数最好只做一件事，尽量不超过80行，当然能写出jdk源码两行代码一个方法就完事了，且粗细粒度掌握刚好，那你们都是我膜拜的大神。

③开发人员间的重复要不要重视？不同开发不同功能模块间，工具类重复、数据保存多次等不易发现，项目知识管理员定期了解各自的需求和实现，或者rpc调用解决数据重复问题。

#### 3 正交性

正如上节中的设计需遵循单一职责，这里正交性再次强调独立性的设计会给自己的程序解耦，总结如下：

“对于两个或多个事务，其中一个改变不影响其他任何一个，则这些事务是正交的。在良好的设计系统中，数据库相交应该和用户界面正交。”

而非正交，就像新手驾驶直升机，只知道每步操作的作用和顺序，实际操作时却被每个步骤牵扯的紧密弄得焦头烂额，正如有时我们的系统，变更时牵一发动全身。

正交的好处：①避免重复造轮子，促进重用；分而治之的架构思维，将变更限制在局部，为单元测试和定位问题等等提高了总的生产力。②分而治之的模块分层思想，将病变部分隔离，不影响大局，提高系统的健壮性、可测试性和可变性，大大减少系统风险。

如何设计，还是分而治之的架构思维，如模块化、基于组件、基于构件和分层；规划好了之后要问自己两点：如一个特定功能背后的需求发生显著改变，有多少模块会受到影响？是否依赖了哪些我无法控制的东西？

灵活使用开源工具包（apache/spring/alibaba等），编码时注意解耦、避免全局变量的线程安全问题、避免相似函数时参考策略模式；养成强迫症般的质疑代码的习惯，重构成为常事。

#### 4 可逆性

代码的可逆取决于早期的需求主脉络，那么架构的可逆将取决什么呢？架构可逆灵活性取决于早期的架构视图构建时是否考虑进去了，从而选择或者兼顾什么架构风格。

#### 5 曳光弹

这种使用场景一般出现在客户需求模糊或者需求多的琳琅满目，使人不知道从哪里开始才好，这时需要搭一个简易框子和简易流程，用曳光弹的方式侦察和情报收集工作，然后用螺旋模型+快速开发模型的迭代方式不断填充已经明确的需求的集成开发工作。

#### 6 原型和标签

与曳光弹相反，原型和标签是注重细节。第一次知道原型设计是为了学习经验，价值不在于产生代码，而在于吸取教训。

原型的细节注意点：正确性、完整性、健壮性、格式。

还可以制作架构原型，在特定领域边界内考虑架构原型，列出一些便签和索引卡即可。

不要把原型中的一次性代码用于产品中，它是用来早期识别出潜在问题点，并及时纠正，这种纠正成本低，比开发之后纠正要舒服的多。

#### 7 领域语言

“语言的界限，即是一个人的界限。”说明它是特定领域而存在的范畴，在这个领域内，有专门的术语或者语言表达，如何将内部语言（领域）转成外部可用可理解的语言，《领域驱动设计》书中就把应用领域的形成需要领域专家收集和翻译成程序员能读懂的术语，针对这样一项人工翻译的工程也是成本巨大，一般大型的企业级ERP由于业务的复杂性会去DDD设计好。

#### 8 估算

多精确才够：任何数量的估算，适用于答案的单位来反映想要传达的精确性。

估算从何而来：看看周围有类似的问题的人，是如何解决的，差不多能借鉴到他人经验。

理解在问什么：精确度的问题；问题域的范围问题。

对系统建模：简单的建模，却能换来精确性。粗略的思维模型框架会给估算引入不准确性，这是不可避免的。

把模型分解成组件：获取到模型，分解成组件，描述组件间的如何交互的数学规则。

确定每个参数的值：在分解的基础进一步确认分解后的参数值

计算答案：估算答案不会唯一，就算错也错的有价值

记录估算能力：无论估算是准确的还不准确的，都不要忘了找到原因，进行总结。

估算项目进度：“粉刷导弹”告诉我们，估算是由一系列的预案构成。“吃掉大象”是一种围绕这关键几点，有时也称螺旋+快速迭代

----》检查需求----》分析风险-----》设计、实现、集成---》和用户一起验证---》