# 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---》

## 第三章：基础工具

“工欲善其事，必先利其器”，作为开发者，要不断的收集精良的基础工具，储备职业生涯中的装备 。工具越好，同时知道怎么更好的使用它，效率会得到很大的提升，所以工具会放大我们的才能。

#### 1.纯文本的威力

纯文本是将知识（需求、设计、实现、测试、文档等）持久的存储下来的最佳格式。常常以手工或者编程的方式进行操作，纯文本存储有如下理由：为防备老化而加保险（时间戳等）、杠杆效应（应用不限于当前，广泛用于计算机领域）、易于测试。可以成为异构环境中公共标准，很多中间件，如redis的aof、注册中心zk、hbase等等，都依赖文本文件记录或者存储数据。

#### 2.shell游戏

不要局限于图形工具的方便，熟悉shell，会大大提高生产率，我们现在的环境部署已经是一个sh脚本执行所有运维组件（环境、storm、Kafka、mq，db，java程序等等），有时发布市场包也会仿照测试的sh脚本写点简单的，的确是按照既定轨迹执行命令，方便且精确，避免误操作。我们每天都会操作linux，有必要花时间掌握sh关键知识，掌握sh应用知识。

#### 3.加强编辑能力

游刃有余的使用编辑器，就每天都使用的idea举例，每次看到有的同事不用拿鼠标就用快捷键编程找代码，就感觉比每次从键盘上移到鼠标上，再从鼠标上移动到键盘上要快一点；换角度看每次看到鼠标点点点到不知道哪里的同事，就在后面有点着急了，这还真的体现老司机和新司机的区别。Idea快键那么多，还要加上其他工具使用技巧，很难记住，先开始我们要发掘一种新特性，会有不适应慢点，要尽快把它内化成为一种肌肉记忆，能做到这点的就是不断重复。

#### 版本控制

VCS:对团队协作、环境部署流水线、问题追踪、团队交流帮助太大了；其他书中介绍都已知晓。

#### 调试

我们的系统总会有bug，没有软件是完美的。调试是定位找到并解决bug的重要手段和途径。树立正确的调试心态：去解决问题，而不是责备；不要恐慌。

调试策略：我们要收集bug出现的边界条件，在修改代码前先让bug能复现，复现步骤越多，修复的难度就越大。

陌生之地：“错误的结果”当成了代码错误，学会用笔和纸去梳理脉络；输入值的敏感度，才会使程序崩溃，还可以用二分法细分，缩小范围定位；

二分法：输出日志及跟踪信息（有用的），找个橡皮鸭诉说代码逻辑（没有用过），排除法（实用，排除过下发中代码问题还是设备问题），“select”没出问题（变更一个东西，影响其他问题，导致不能使用，那么这个变动就有间接或者直接的责任）。

让人吃惊的元素：总会有这样的问题出现，开发和测试环境没有问题，可是上了预生产、生产环境，总会推翻之前的测试结果，用户就会用他的方式帮你找到bug，有的是并发有的是环境，有的是特定的场景等等。这样的问题往往让开发和测试感到惊讶，所以很多互联网企业在软件第一次上线都会先内部发布，再灰度发布试部分运营一周的时间，待充分消化这些吃惊的元素，才考虑大规模发布。

#### 文本处理

书中建议我们学习一种文本处理语言，如python或rudy。以后可以安排学点python。

#### 7.工程日记：

最后建议我们有意识的手写记录工程日记，其实我们有时会有有道云笔记记录一些重点，没必要原原本本的记录能快速处理完或者不重要的日记。其实我们写周报就是在写工程日记，只记录关键。

## 第四章：务实的偏执

既然没有人能写出完美的代码，我们应该在编码过程中，强迫怀疑错误就隐藏在自己的代码的某处，并且行之有效的建立起防御机制。

#### 契约式设计

契约规定了我和对方的责任和权利，文档化及对主张进行校验是契约式设计的核心。

DBC的理解：不能满足契约的就是个bug，为什么这么说？如果调用者满足了例程（函数或者方法接口执行体）的所有前置条件（参数校验，需求数据，必须为真的是什么等），则例程应保证在完成时所有的后置条件（例程保证做的是什么？意味着能得出这样的结论，想要的需求结果）和不变式（类始终保证该条件下为真）都为真。

DBC与测试驱动开发（特定的单元测试，已经属于防御性编程了），DBC还是有存在的5点价值。

DBC的设计技术：断言（spring源码中大量用到），通常会检查到第三方和你的边界处最多的问题，效果是：在问题发生的地方尽早崩溃，能让问题和诊断问题更加容易。

语义不变式：不可违背的需求（如一次付款，不能扣我两次钱）。

动态契约与代理：nginx的负载均衡算法，的确可以用代理为我们化解访问资源分配带来的危机。

#### 死掉的程序不会说慌

尽快检查出问题好处之一是，可以更早崩溃，该抛出异常阻断的时候要阻断，事务一致性的处理有必要一定要加，数据同步问题一定要考虑分布式事务的有效方式实现。

#### 断言式编程

使用断言去预防不可能的事情；例如检查参数非空、算法检查等

断言的副作用：调试本身改变了被调试系统的原有行为。

保持断言常开：不要完全依赖测试通过的结果来去掉断言可能发生的事情。而且断言的精心设计，还能找到一些难以复制的生产bug，从而发现得到非常稳定的防弹级软件。

#### 如何保持资源的平衡

资源使用遵循可预测的模式：分配、使用、释放；避免没有计划的使用资源。

有始有终，对释放的资源负责；在局部行动：缩小范围；

嵌套分配：

1. 释放资源与分配资源顺序相反。
2. 代码的不同位置，都分配了同一组资源，就始终以相同的顺序分配他们，可减少死锁可能。

对象与异常；保持平衡与异常：这里在下发时就遇到了，提前抛出的异常，导致后面的对象为null，使用时报错；当无法保持资源平衡时，注意层级结构方式释放或者管理资源。检查平衡可用jvm工具检查是否存在内存泄露、

#### 不要冲出前灯的范围

经常晚上打球回来在流芳路那里刮车子地盘，夜晚开太快，有坑的位置，灯光没有提前照射到，导致没有提前反映避坑。

小步前进—由始至终：有时我也会采用这种方式去推进工作，先深思熟虑考虑一些小的需求实现，把不确定的先圈定影响范围，边推进向前，边不断调整。任何大到需要去占卜的猜测性需求，一般往后面压压，讨论和问清楚再动手设计。第二还得注意设计代码时，不能耦合那种需求，要可替换或者可弥补性的设计方式。

《黑天鹅：如何应对不可知的未来》早些年看过这本神作，重大事件都来自备受关注、难以预测、罕见且超出预期的事件。最后要避免黑天鹅的发生唯有时刻注意事务的变化细节，避免占卜。