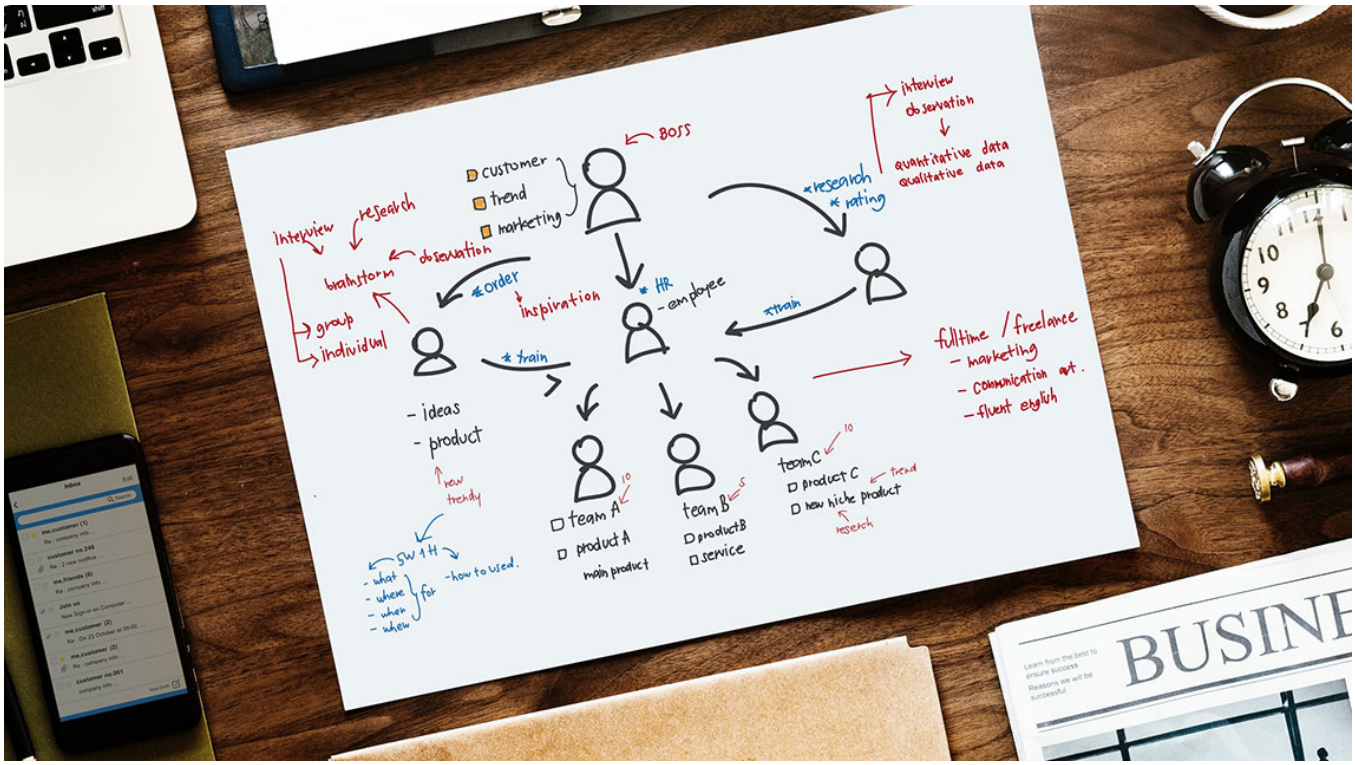
程序员练级攻略（2018）：软件设计

2018-06-21 陈皓，杨爽



程序员练级攻略（2018）：软件设计
朗读人：柴巍　24′49″ | 8.53M

学习软件设计的方法、理念、范式和模式，是让你从一个程序员通向工程师的必备技能。如果你不懂这些设计方法，那么你将无法做出优质的软件。这就好像写作文一样，文章人人都能写，但是能写得有条理，有章法，有血有肉，就不简单了。软件开发也一样，实现功能，做出来并不难，但是要做漂亮，做优雅，就非常不容易了。

Linus 说过，这世界程序员之所有高下之分，最大的区别就是程序员的“品味”不一样。有品位的程序员和没有品位的程序员所写出来的代码，所做出来的软件，差距非常大，而且价值也差别很大。所以，对于我们每一个程序员，如果你想成为软件工程师、设计师或架构师，软件设计是你必须用心学习的事。

然而，软件设计这个事，并不是一朝一夕就能学会的，也不是别人能把你教会的，很多东西需要你自己用实践、用时间、用错误、用教训、用痛苦才能真正体会其中的精髓的。所以，除了学习理论知识外，你还需要大量的工程实践，然后每过一段时间就把这些设计的东西重新回炉一下。

你会发现这些软件设计的东西，就像饮茶一样，一开始是苦的，然后慢慢回甘，最终你会喝出真正的滋味。

要学好这些软件开发和设计的方法，你真的需要磨练和苦行，反复咀嚼，反复推敲，在实践和理论中螺旋式地学习，才能真正掌握。 所以，你需要有足够的耐心和恒心。

编程范式

学习编程范式可以让你明白编程的本质和各种语言的编程方式。虽然很多程序员都忽略了这个事，但是其实是非常非常重要的事。因此，我推荐以下一些资料，帮助你系统化地学习和理解。

- 一个是我在极客时间写的《编程范式游记》系列文章，目录如下。
    - 编程范式游记（1）- 起源
    - 编程范式游记（2）- 泛型编程
    - 编程范式游记（3）- 类型系统和泛型的本质
    - 编程范式游记（4）- 函数式编程
    - 编程范式游记（5）- 修饰器模式
    - 编程范式游记（6）- 面向对象编程
    - 编程范式游记（7）- 基于原型的编程范式
    - 编程范式游记（8）- Go 语言的委托模式
    - 编程范式游记（9）- 编程的本质
    - 编程范式游记（10）- 逻辑编程范式
    - 编程范式游记（11）- 程序世界里的编程范式
- Wikipedia: Programming paradigm ，维基百科上有一个编程范式的页面，顺着这个页面看下去，你可以看到很多很多有用的和编程相关的知识。这些东西对你的编程技能的提高会非常非常有帮助。
- Six programming paradigms that will change how you think about coding，中文翻译版为 六个编程范型将改变你对编程的看法。这篇文章讲了默认支持并发（Concurrent by default）、依赖类型（Dependent types）、连接性语言（Concatenative languages）、声明式编程（Declarative programming）、符号式编程（Symbolic programming）、基于知识的编程（Knowledge-based programming）等六种不太常见的编程范式，并结合了一些你没怎么听说过的语言来分别进行讲述。

  比如在讲 Concatenative languages 时，以 Forth、cat 和 joy 三种语言为例讲述这一编程范式背后的思想——语言中的所有内容都是一个函数，用于将数据推送到堆栈或从堆栈弹出数据；程序几乎完全通过功能组合来构建（concatenation is composition）。作者认为，这些编程范式背后的思想十分有魅力，能够改变对编程的思考。我看完此文，对此也深信不

疑。虽然这些语言和编程范式不常用到，但确实能在思想层面给予人很大的启发。这也是我推荐此文的目的。

- Programming Paradigms for Dummies: What Every Programmer Should Know ，这篇文章的作者彼得·范·罗伊（Peter Van Roy）是比利时鲁汶大学的计算机科学教师。他在这篇文章里分析了编程语言在历史上的演进，有哪些典型的、值得研究的案例，里面体现了哪些值得学习的范式。

  比如，在分布式编程领域，他提到了 Erlang、E、Distributed Oz 和 Didactic Oz 这四种编程语言。虽然它们都是分布式编程语言，但各有特色，各自解决了不同的问题。通过这篇文章能学到不少在设计编程语言时要考虑的问题，让你重新审视自己所使用的编程语言应该怎样用才能用好，有什么局限性，这些局限性能否被克服等。

- 斯坦福大学公开课：编程范式，这是一门比较基础且很详细的课程，适合学习编程语言的初学者。它通过讲述 C、C++、并发编程、Scheme、Python 这 5 门语言，介绍了它们各自不同的编程范式。以 C 语言为例，它解释了 C 语言的基本要素，如指针、内存分配、堆、C 风格的字符串等，并解释了为什么 C 语言会在泛型编程、多态等方面有局限性。通过学习这门课程，你会对一些常用的编程范式有所了解。

一些软件设计的相关原则

- Don't Repeat Yourself (DRY) ，DRY 是一个最简单的法则，也是最容易被理解的。但它也可能是最难被应用的（因为要做到这样，我们需要在泛型设计上做相当的努力，这并不是一件容易的事）。它意味着，当在两个或多个地方发现一些相似的代码的时候，我们需要把它们的共性抽象出来形成一个唯一的新方法，并且改变现有地方的代码让它们以一些合适的参数调用这个新的方法。

- Keep It Simple, Stupid(KISS) ，KISS 原则在设计上可能最被推崇，在家装设计、界面设计和操作设计上，复杂的东西越来越被众人所鄙视了，而简单的东西越来越被人所认可。宜家（IKEA）简约、高效的家居设计和生产思路；微软（Microsoft）“所见即所得”的理念；谷歌（Google）简约、直接的商业风格，无一例外地遵循了“KISS”原则。也正是“KISS”原则，成就了这些看似神奇的商业经典。而苹果公司的 iPhone 和 iPad 将这个原则实践到了极至。

- Program to an interface, not an implementation，这是设计模式中最根本的哲学，注重接口，而不是实现，依赖接口，而不是实现。接口是抽象是稳定的，实现则是多种多样的。在面向对象的 S.O.L.I.D 原则中会提到我们的依赖倒置原则，就是这个原则的另一种样子。还有一条原则叫 Composition over inheritance（喜欢组合而不是继承），这两条是那 23 个经典设计模式中的设计原则。

- You Ain't Gonna Need It (YAGNI) ，这个原则简而言之为——只考虑和设计必须的功能，避免过度设计。只实现目前需要的功能，在以后你需要更多功能时，可以再进行添加。如无必要，勿增复杂性。软件开发是一场 trade-off 的博弈。

- Law of Demeter，迪米特法则 (Law of Demeter)，又称“最少知识原则”（Principle of Least Knowledge），其来源于 1987 年荷兰大学的一个叫做 Demeter 的项目。克雷格·拉尔曼（Craig Larman）把 Law of Demeter 又称作“不要和陌生人说话”。在《程序员修炼之道》中讲 LoD 的那一章将其叫作“解耦合与迪米特法则”。

  关于迪米特法则有一些很形象的比喻：1) 如果你想让你的狗跑的话，你会对狗狗说还是对四条狗腿说？2) 如果你去店里买东西，你会把钱交给店员，还是会把钱包交给店员让他自己拿？和狗的四肢说话？让店员自己从钱包里拿钱？这听起来有点儿荒唐，不过在我们的代码里这几乎是见怪不怪的事情了。对于 LoD，正式的表述如下：

  对于对象 'O' 中一个方法 'M'，M 应该只能够访问以下对象中的方法：

  1. 对象 O；
  2. 与 O 直接相关的 Component Object；
  3. 由方法 M 创建或者实例化的对象；
  4. 作为方法 M 的参数的对象。

- 面向对象的 S.O.L.I.D 原则：

  - SRP（Single Responsibility Principle）- 职责单一原则。关于单一职责原则，其核心的思想是：一个类，只做一件事，并把这件事做好，其只有一个引起它变化的原因。单一职责原则可以看作是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。

    职责过多，可能引起它变化的原因就越多，这将导致职责依赖，相互之间就产生影响，从而极大地损伤其内聚性和耦合度。单一职责，通常意味着单一的功能，因此不要为一个模块实现过多的功能点，以保证实体只有一个引起它变化的原因。

  - OCP（Open/Closed Principle）- 开闭原则。关于开发封闭原则，其核心的思想是：模块是可扩展的，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

  - LSP（Liskov substitution principle）- 里氏代换原则。软件工程大师罗伯特·马丁（Robert C. Martin）把里氏代换原则最终简化为一句话：“Subtypes must be substitutable for their base types”。也就是，子类必须能够替换成它们的基类。即子

类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。另外，不应该在代码中出现 if/else 之类对子类类型进行判断的条件。里氏替换原则 LSP 是使代码符合开闭原则的一个重要保证。正是由于子类型的可替换性才使得父类型的模块在无需修改的情况下就可以扩展。

- ISP（Interface Segregation Principle ）- 接口隔离原则。接口隔离原则的意思是把功能实现在接口中，而不是类中，使用多个专门的接口比使用单一的总接口要好。举个例子，我们对电脑有不同的使用方式，比如：写作、通讯、看电影、打游戏、上网、编程、计算和数据存储等。

  如果我们把这些功能都声明在电脑的抽象类里面，那么，我们的上网本、PC 机、服务器和笔记本的实现类都要实现所有的这些接口，这就显得太复杂了。所以，我们可以把这些功能接口隔离开来，如工作学习接口、编程开发接口、上网娱乐接口、计算和数据服务接口，这样，我们的不同功能的电脑就可以有所选择地继承这些接口。

- DIP（Dependency Inversion Principle）- 依赖倒置原则。高层模块不应该依赖于低层模块的实现，而是依赖于高层抽象。举个例子，墙面的开关不应该依赖于电灯的开关实现，而是应该依赖于一个抽象的开关的标准接口。这样，当我们扩展程序的时候，开关同样可以控制其它不同的灯，甚至不同的电器。也就是说，电灯和其它电器继承并实现我们的标准开关接口，而开关厂商就可以不需要关于其要控制什么样的设备，只需要关心那个标准的开关标准。这就是依赖倒置原则。

- CCP（Common Closure Principle）- 共同封闭原则，一个包中所有的类应该对同一种类型的变化关闭。一个变化影响一个包，便影响了包中所有的类。一个更简短的说法是：一起修改的类，应该组合在一起（同一个包里）。如果必须修改应用程序里的代码，那么我们希望所有的修改都发生在一个包里（修改关闭），而不是遍布在很多包里。

  CCP 原则就是把因为某个同样的原因而需要修改的所有类组合进一个包里。如果两个类从物理上或者从概念上联系得非常紧密，它们通常一起发生改变，那么它们应该属于同一个包。CCP 延伸了开闭原则（OCP）的“关闭”概念，当因为某个原因需要修改时，把需要修改的范围限制在一个最小范围内的包里。

- CRP（Common Reuse Principle）- 共同重用原则 ，包的所有类被一起重用。如果你重用了其中的一个类，就重用全部。换个说法是，没有被一起重用的类不应该组合在一起。CRP 原则帮助我们决定哪些类应该被放到同一个包里。依赖一个包就是依赖这个包所包含的一切。

  当一个包发生了改变，并发布新的版本，使用这个包的所有用户都必须在新的包环境下验证他们的工作，即使被他们使用的部分没有发生任何改变。因为如果包中包含未被使用的类，即使用户不关心该类是否改变，但用户还是不得不升级该包并对原来的功能加以重新测试。

CCP 则让系统的维护者受益。CCP 让包尽可能大（CCP 原则加入功能相关的类），CRP 则让包尽可能小（CRP 原则剔除不使用的类）。它们的出发点不一样，但不相互冲突。

- 好莱坞原则 - Hollywood Principle ，好莱坞原则就是一句话——“don't call us, we'll call you.” 。意思是，好莱坞的经纪人不希望你去联系他们，而是他们会在需要的时候来联系你。也就是说，所有的组件都是被动的，所有的组件初始化和调用都由容器负责。

  简单来讲，就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。这也就是所谓“控制反转”的概念所在：1) 不创建对象，而是描述创建对象的方式。2）在代码中，对象与服务没有直接联系，而是容器负责将这些联系在一起。控制权由应用代码中转到了外部容器，控制权的转移，是所谓反转。好莱坞原则就是IoC（Inversion of Control） 或 DI（Dependency Injection）的基础原则。

- 高内聚， 低耦合 & - High Cohesion & Low/Loose coupling)，这个原则是 UNIX 操作系统设计的经典原则，把模块间的耦合降到最低，而努力让一个模块做到精益求精。内聚，指一个模块内各个元素彼此结合的紧密程度；耦合指一个软件结构内不同模块之间互连程度的度量。内聚意味着重用和独立，耦合意味着多米诺效应牵一发动全身。对于面向对象来说，你也可以看看马萨诸塞州戈登学院的面向对象课中的这一节讲义High Cohesion and Low Coupling。

- CoC（Convention over Configuration）- 惯例优于配置原则 ，简单点说，就是将一些公认的配置方式和信息作为内部缺省的规则来使用。例如，Hibernate 的映射文件，如果约定字段名和类属性一致的话，基本上就可以不要这个配置文件了。你的应用只需要指定不 convention 的信息即可，从而减少了大量 convention 而又不得不花时间和精力啰里啰嗦的东东。

  配置文件在很多时候相当影响开发效率。Rails 中很少有配置文件（但不是没有，数据库连接就是一个配置文件）。Rails 的 fans 号称其开发效率是 Java 开发的 10 倍，估计就是这个原因。Maven 也使用了 CoC 原则，当你执行 `mvn -compile` 命令的时候，不需要指定源文件放在什么地方，而编译以后的 class 文件放置在什么地方也没有指定，这就是 CoC 原则。

- SoC (Separation of Concerns) - 关注点分离 ，SoC 是计算机科学中最重要的努力目标之一。这个原则，就是在软件开发中，通过各种手段，将问题的各个关注点分开。如果一个问题能分解为独立且较小的问题，就是相对较易解决的。问题太过于复杂，要解决问题需要关注的点太多，而程序员的能力是有限的，不能同时关注于问题的各个方面。

  正如程序员的记忆力相对于计算机知识来说那么有限一样，程序员解决问题的能力相对于要解决的问题的复杂性也是一样的非常有限。在我们分析问题的时候，如果我们把所有的东西混在一起讨论，那么就只会有一个结果——乱。实现关注点分离的方法主要有两种，一种是标准化，另一种是抽象与包装。标准化就是制定一套标准，让使用者都遵守它，将人们的行

为统一起来，这样使用标准的人就不用担心别人会有很多种不同的实现，使自己的程序不能和别人的配合。

就像是开发镙丝钉的人只专注于开发镙丝钉就行了，而不用关注镙帽是怎么生产的，反正镙帽和镙丝钉按照标准来就一定能合得上。不断地把程序的某些部分抽象并包装起来，也是实现关注点分离的好方法。一旦一个函数被抽象出来并实现了，那么使用函数的人就不用关心这个函数是如何实现的。同样的，一旦一个类被抽象并实现了，类的使用者也不用再关注于这个类的内部是如何实现的。诸如组件、分层、面向服务等这些概念都是在不同的层次上做抽象和包装，以使得使用者不用关心它的内部实现细节。

- DbC（Design by Contract）- 契约式设计 ，DbC 的核心思想是对软件系统中的元素之间相互合作以及“责任”与“义务”的比喻。这种比喻从商业活动中“客户”与“供应商”达成“契约”而得来。如果在程序设计中一个模块提供了某种功能，那么它要：
    - 期望所有调用它的客户模块都保证一定的进入条件：这就是模块的先验条件（客户的义务和供应商的权利，这样它就不用去处理不满足先验条件的情况）。
    - 保证退出时给出特定的属性：这就是模块的后验条件（供应商的义务，显然也是客户的权利）。
    - 在进入时假定，并在退出时保持一些特定的属性：不变式。
- ADP（Acyclic Dependencies Principle）- 无环依赖原则 ，包（或服务）之间的依赖结构必须是一个直接的无环图形，也就是说，在依赖结构中不允许出现环（循环依赖）。如果包的依赖形成了环状结构，怎么样打破这种循环依赖呢？

  有两种方法可以打破这种循环依赖关系：第一种方法是创建新的包，如果 A、B、C 形成环路依赖，那么把这些共同类抽出来放在一个新的包 D 里。这样就把 C 依赖 A 变成了 C 依赖 D 以及 A 依赖 D，从而打破了循环依赖关系。第二种方法是使用 DIP（依赖倒置原则）和 ISP（接口分隔原则）设计原则。无环依赖原则（ADP）为我们解决包之间的关系耦合问题。在设计模块时，不能有循环依赖。

一些软件设计的读物

- 《领域驱动设计》 ，本书是领域驱动设计方面的经典之作。全书围绕着设计和开发实践，结合若干真实的项目案例，向读者阐述如何在真实的软件开发中应用领域驱动设计。书中给出了领域驱动设计的系统化方法，并将人们普遍接受的一些实践综合到一起，融入了作者的见解和经验，展现了一些可扩展的设计新实践、已验证过的技术以及便于应对复杂领域的软件项目开发的基本原则。

- 《UNIX 编程艺术》，这本书主要介绍了 Unix 系统领域中的设计和开发哲学、思想文化体系、原则与经验，由公认的 Unix 编程大师、开源运动领袖人物之一埃里克·雷蒙德（Eric S. Raymond）倾力多年写作而成。包括 Unix 设计者在内的多位领域专家也为本书贡献了宝贵的内容。本书内容涉及社群文化、软件开发设计与实现，覆盖面广、内容深邃，完全展现了作者极其深厚的经验积累和领域智慧。

- 《Clean Architecture》，如果你读过《Clean Code》和《The Clean Coder》这两本书。你就能猜得到这种 Clean 系列一定也是出自“Bob 大叔”之手。没错，就是 Bob 大叔的心血之作。除了这个网站，《Clean Architecture》也是一本书，这是一本很不错的架构类图书。对软件架构的元素、方法等讲得很清楚。示例都比较简单，并带一些软件变化历史的讲述，很开阔视野。

- The Twelve-Factor App，如今，软件通常会作为一种服务来交付，它们被称为网络应用程序，或软件即服务（SaaS）。12-Factor 为构建 SaaS 应用提供了方法论，这也是架构师必读的文章。（中译版）这篇文章在业内的影响力很大，必读！

- Avoid Over Engineering，有时候，我们会过渡设计我们的系统，过度设计会把我们带到另外一个复杂度上，所以，我们需要一些工程上的平衡。这篇文章是一篇非常不错地告诉你什么是过度设计的文章。

- Instagram Engineering's 3 rules to a scalable cloud application architecture，Instagram 工程的三个黄金法则：1）使用稳定可靠的技术（迎接新的技术）；2）不要重新发明轮子；3）Keep it very simple。我觉得这三条很不错。其实，Amazon 也有两条工程法则，一个是自动化，一个是简化。

- How To Design A Good API and Why it Matters - Joshua Bloch，Google 的一个分享，关于如何设计好一个 API。

- 关于 Restful API 的设计，你可以学习并借鉴一下下面这些文章。

  - Best Practices for Designing a Pragmatic RESTful API
  - Ideal REST API design
  - HTTP API Design Guide
  - Microsoft REST API Guidelines
  - IBM Watson REST API Guidelines
  - Zalando RESTful API and Event Scheme Guidelines

- The Problem With Logging，一篇关于程序打日志的短文，可以让你知道一些可能以往不知道的打日志需要注意的问题。

- Concurrent Programming for Scalable Web Architectures ，这是一本在线的免费书，教你如何架构一个可扩展的高性能的网站。其中谈到了一些不错的设计方法和知识。

小结

好了，总结一下今天分享的内容。我认为，“品位”不同，是各层次程序员之间最大的区别，这也决定了他们所做出来的软件的质量和价值。因此，我特意撰写了软件设计这一篇章，帮助那些想成长为软件工程师、设计师或架构师的程序员，提高软件设计的品位，进而实现自己的目标。

虽然很多程序员都忽略了对编程范式的学习，但我觉得学习编程范式其实是非常非常重要的事，能够明白编程的本质和各种语言的编程方式。为此，我推荐了好几份学习资料，帮助你系统化地学习和理解。随后我介绍了 DRY- 避免重复原则、KISS- 简单原则、迪米特法则（又称“最少知识原则”）、 面向对象的 S.O.L.I.D 原则等多个经典的软件设计原则。

最后，我精选并推荐了软件设计方面的学习资料，如《领域驱动设计》、《UNIX 编程艺术》和《Clean Architecture》等必读好书，以及如何构建 SaaS，如何避免过度设计，如何设计 API，如何用程序打日志等方面的资料。

希望这些内容对你有帮助。从下一篇文章开始，我们将进入《程序员练级攻略（2018）》的第五个篇章——高手成长篇。敬请期待。

下面是《程序员练级攻略（2018）》系列文章的目录（持续更新中）。

精选留言

颇忒妥 3

强烈推荐领域驱动设计，要看原版

2018-06-21

刘強 1

最值得付费专栏。

2018-06-22

Lee 1

耗子叔，有个阅读顺序的问题一直想请教您。以前是按您酷壳上的练级攻略学习的，现在的工作c++程序员，您介绍的有些书是看过的，但是从入门启蒙篇里开始的每篇里都有很多没看过的书，我是应该按您每篇里的书读完后再读下一篇里的，还是说可以先着重读某一篇里的。比如，系统篇里的操作系统和网络的书我读过一些，但是入门篇里的Python只学过一些笨办法学Python的教程，html/css只在w3cschool上学过一些，并没有把启蒙篇里的学完，这时的学习顺序怎么安排优先级，比如可能会面临换工作的可能。

2018-06-21

SMTCode 0

感觉手握大量武林秘籍，怎奈无神功护体，不得其精要。总喜欢幻想一招制敌，怎奈真正的修行要经过多少个春夏秋冬。一路走来，能坚持下来的人了了，但坚持下来的都成了高手和传说。不写了，孩醒了，看娃～

2018-06-22

SMTCode 0

字字珠玑，针针见血，博学多才，厚积薄发。对于入行晚，又没有耗叔这般超级大脑，还要陪伴宝宝的我来说，优先把编程语言这种工具用起来，给老婆和孩儿一个基本的经济保障。但要想用好工具，需要把耗叔推荐的系统知识、编程语言、编程规约穿插起来，而且要经过大量实战才能融会贯通。接触过耗叔推荐的部分书籍，要想坚持看下去真心难。耗叔的毅力和领悟能力不是一般的高。以前我都是以操作系统为中心划分知识方向，现在觉得耗叔的划分（计算+存储+网络）更加科学。耗叔推荐的四种编程语言，都很难一蹴而就，应该从自己擅长的部分入手，逐步扩大。我想把计算机技术作为一个终身的奋斗目标，从这角度出发，系统基本功越早学越好，何时学都不晚。耗叔成功给我们穿了一条线，能少走很多弯路。至于每个人能领略到何种程度，最终能达到何种高度，最后拼的还是脑袋+耐力。面对这么多维度的知识，该如何平衡各部分的时间，也是因人而异的，我也没有想好。不过千里之行，始于足下。行动起来，逐个突破，纵使达不到耗叔的高度，也能让自己站到更高的层次，用耐力、有耐心去打磨自己。我觉得耗叔应该很“冷”，因为高处不胜寒，孤独是多么的寂寞。耗叔有我们，不孤独。

2018-06-22

湖心亭看雪 0

关于DDD，我也推荐一本落地的好书吧，implementing domain-driven design，实现领域驱动设计。

2018-06-21

Michael 0

耗哥问个题外话，对于加班文化怎么看，感觉很多加班并不能提升多少技术，很多都是拼体力

耗哥怎么看待加班呢

2018-06-21

赵亮 0

avoid over engineering 链接打不开，是被墙了么？

2018-06-21

*********26 0

我的水平不够，仅做参考，膜拜大佬

2018-06-21

gevin 0

太经典了！

把我这些年掌握的编程原则和经验都串起来了，还补充了我的各个短板

感谢耗子哥！

2018-06-21

夜行观星 0

做好打持久战的准备

2018-06-21

Abyssal 0

每期必看！谢谢皓叔指明方向！

2018-06-21

孙悟空 0

编程范式是不是可以理解为设计模式昵

2018-06-21

云学 0

曾经花了一年的时间学习和实践clean code，对作者提到的那些原则有一些体会，收获很大

2018-06-21

Geek.Kwok 0

与和“世界杯”一样豪门的知识盛宴，超级赞

2018-06-21

长不胖的Garfield 0

信息量太大了！这一篇文章够看好久

2018-06-21

zzz 0

耗子叔你好，有一个很久之前就在的想法和顾虑：从csdn 到酷壳，再到这里。你分享和推荐的很多知识，经典，但是好多知识点，需要日复一日，甚至好几个月才能学完，那么，问题来了，比如，你一个月前推荐的东西，我一个月后才学完，就比如这个程序员系列，基本学完您推荐的书后，得好几年，两年后还能看到极客时间您的专栏的文章吗？或者您的酷壳和csdn 会一直都在吗？我用不用把你的所有博客和文章记录到我的笔记，以防丢失呢？这个工程量有点大。嘿嘿😁

2018-06-21