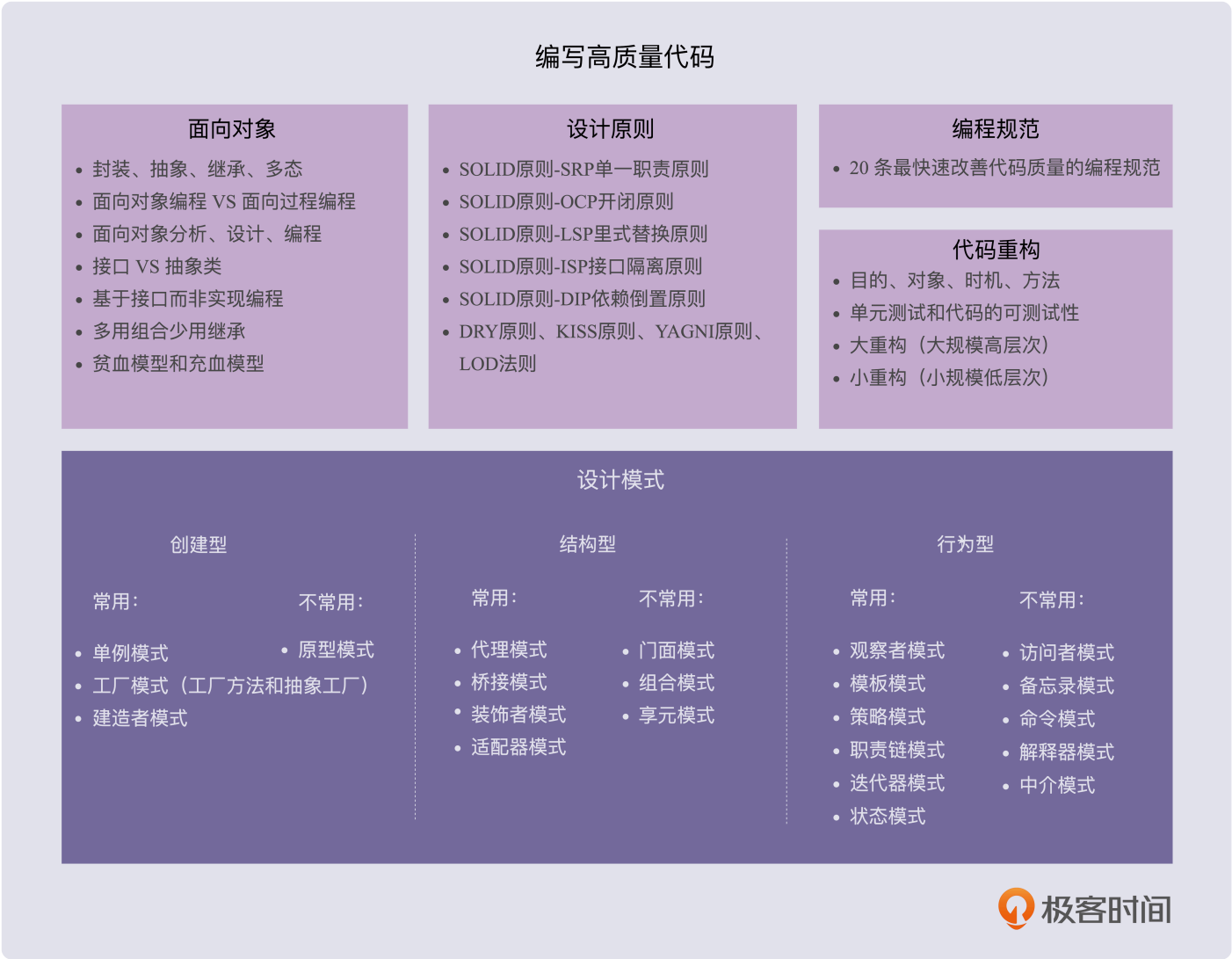
99 | 总结回顾：在实际软件开发中常用的设计思想、原则和模式

王争 · 设计模式之美

到今天为止，理论部分和实战部分都已经讲完了，整个专栏也接近尾声了。我这里用两节课的时间，带你一块复习一下前面学到的知识点。跟前面的讲解相对应，这两节课分别是针对理论部分和实战部分进行回顾总结。

今天，我先来带你回顾一下整个专栏的知识体系。我们整个专栏围绕着编写高质量代码展开，涵盖了代码设计的方方面面，主要包括面向对象、设计原则、编码规范、重构技巧、设计模式这五个部分。我们就从这五个方面，带你一块把之前学过的知识点串一遍。



话不多说，让我们正式开始今天的学习吧！

面向对象

相对于面向过程、函数式编程，面向对象是现在最主流的编程范式。纯面向过程的编程方法，现在已经不多见了，而新的函数式编程，因为它的应用场景比较局限，所以大多作为面向对象编程的一种补充，用在科学计算、大数据处理等特殊领域。

它提供了丰富的特性，比如封装、抽象、继承、多态，有助于实现复杂的设计思路，是很多设计原则、设计模式等编程实现的基础。

在面向对象这一部分，我们要重点掌握面向对象的四大特性：封装、抽象、继承、多态，以及面向对象编程与面向过程编程的区别。需要特别注意的是，在平时的面向对象编程开发中，我们要避免编写出面向过程风格的代码。

除此之外，我们还重点学习了面向对象分析（OOA）、设计（OOD）、编程（OOP）。其中，面向对象分析就是需求分析，面向对象设计是代码层面的设计，输出的设计结果是类。面向对象编程就是将设计的结果翻译成代码的过程。

在专栏中，我们重点讲解了面向对象设计这一部分。我们可以把面向对象设计分为四个环节：划分职责并识别出有哪些类、定义类及其属性和方法、定义类之间的交互关系、组装类并提供执行入口。我们通过几个项目案例，带你实战了一下设计过程，希望你能面对开发需求的时候，不会无从下手，做到有章可循，按照我们的给出的步骤，有条不紊地完成设计。

在面向对象这一部分，我们还额外讲到了两个设计思想：基于接口而非实现的设计思想、多用组合少用继承的设计思想。这两个设计思想虽然简单，但非常实用，应用它们能让代码更加灵活，更加容易扩展，所以，这两个设计思想几乎贯穿在我们整个专栏的代码实现中。

设计原则

在专栏的最开始，我们总结了一套评判代码质量的标准，比如可读性、可维护性、可扩展性、复用性等，这是从代码整体质量的角度来评判的。但是，落实到具体的细节，我们往往从是否符合设计原则，来对代码设计进行评判。比如，我们说这段代码的可扩展性比较差，主要原因是违背了开闭原则。这也就是说，相对于可读性、可维护性、可扩展性等代码整体质量的评判标准，设计原则更加具体，能够更加明确地指出代码存在的问题。

在专栏中，我们重点讲解了一些经典的设计原则，大部分都耳熟能详。它们分别是 SOLID 原则、DRY 原则、KISS 原则、YAGNI 原则、LOD 原则。这些原则的定义描述都很简单，看似都很好理解，但也都比较抽象，比较难落地指导具体的编程。所以，学习的重点是透彻理解它们的设计初衷，掌握它们能解决哪些编程问题，有哪些常用的应用场景。

SOLID 原则并非一个原则。它包含：单一职责原则（SRP）、开闭原则（OCP）、里氏替换原则（LSP）、接口隔离原则（ISP）、依赖倒置原则（DIP）。其中，里氏替换和接口隔离这两个设计原则并不那么常用，稍微了解就可以了。我们重点学习了单一职责、开闭、依赖倒置这三个原则。

单一职责原则是类职责划分的重要参考依据，是保证代码”高内聚“的有效手段，是面向对象设计前两步（划分职责并识别出有哪些类、定义类及其属性和方法）的主要指导原则。单一职责

原则的难点在于，对代码职责是否足够单一的判定。这要根据具体的场景来具体分析。同一个类的设计，在不同的场景下，对职责是否单一的判定，可能是不同的。

开闭原则是保证代码可扩展性的重要指导原则，是对代码扩展性的具体解读。很多设计模式诞生的初衷都是为了提高代码的扩展性，都是以满足开闭原则为设计目的的。实际上，尽管开闭原则描述为对扩展开放、对修改关闭，但也并不是说杜绝一切代码修改，正确的理解是以最小化修改代价来完成新功能的添加。实际上，在平时的开发中，我们要时刻思考，目前的设计在以后应对新功能扩展的时候，是否能做到不需要大的代码修改（比如调整代码结构）就能完成。

依赖倒置原则主要用来指导框架层面的设计。高层模块不依赖低层模块，它们共同依赖同一个抽象。深挖一下的话，我们要把它跟控制反转、依赖注入、依赖注入框架做区分。实际上，比依赖倒置原则更加常用的是依赖注入。它用来指导如何编写可测试性代码，换句话说，编写可测试代码的诀窍就是应用依赖注入。

KISS、YAGNI 可以说是两个万金油原则，小到代码、大到架构、产品，很多场景都能套用这两条原则。其中，YAGNI 原则表示暂时不需要的就不要做，KISS 原则表示要做就要尽量保持简单。跟单一职责原则类似，掌握这两个原则的难点也是在于，对代码是否符合 KISS、YAGNI 原则的判定。这也需要根据具体的场景来具体分析，在某个时间点、某个场景下，某段代码符合 KISS、YAGNI 原则，换个时间点、换个场景，可能就不符合了。

DRY 原则主要是提醒你不要写重复的代码，这个倒是不难掌握。LOD 原则又叫最小知道原则，不该有直接依赖关系的类之间，不要有依赖；有依赖关系的类之间，尽量只依赖必要的接口。如果说单一职责原则是为了实现“高内聚”， 那这个原则就是为了实现“松耦合”。

编码规范

编码规范很重要，特别是对于初入职、开发经验不多的程序员，遵从好的编码规范，能让你写出来的代码至少不会太烂。而且，编码规范都比较具体，不像设计原则、模式、思想那样，比较抽象，需要融入很多个人的理解和思考，需要根据具体的场景具体分析，所以，它落地执行起来更加容易。

虽然我们讲了很多设计思想、原则、模式，但是，大部分代码都不需要用到这么复杂的设计，即便用到，可能也就只是用到极个别的知识点，而且用的也不会很频繁。但是，编码规范就不一样了。编码规范影响到你写的每个类、函数、变量。你编写每行代码的时候都要思考是否符合编码规范。

除此之外，编程规范主要解决代码的可读性问题。我个人觉得，在编写代码的时候，我们要把可读性放到首位。只有在代码可读性比较好的情况下，我们再去考虑代码的扩展性、灵活性等。一般来说，一个可读性比较好的代码，对它修改、扩展、重构都不是难事，因为这些工作的前提都是先读懂代码。

不过，专栏中只是总结了一些最常用的、最能明显改善代码质量的编码规范，更进一步的学习你可以参考《重构》《代码大全》《代码整洁之道》等书籍，或者参看你公司内部的编码规范。

重构技巧

重构作为保持代码质量不腐化的有效手段，利用的就是面向对象、设计原则、设计模式、编码规范这些理论。在重构的过程中，我们用代码质量评判标准来评判代码的整体质量，然后对照设计原则来发现代码存在的具体问题，最后用设计模式或者编码规范对存在的问题进行改善。

持续重构除了能保证代码质量不腐化之外，还能有效避免过度设计。有了持续重构意识，我们就不会因为担心设计不足而过度设计。我们先按照最简单的思路来设计，然后在后续的开发过程中逐步迭代重构。

在专栏中，我们还对重构进行了粗略的分类，分为大规模高层次的重构和小规模低层次的重构。不管哪种重构，保证重构不出错，除了熟悉代码之外，还有就是完善的单元测试。

设计模式

如果说设计原则相当于编程心法，那设计模式相当于具体的招式。设计模式是针对软件开发中经常遇到的一些设计问题，总结出来的一套解决方案或者设计思路。我们用设计原则来评判代码设计哪里有问题，然后再通过具体的设计模式来改善。相对于其他部分来讲，设计模式是最

容易学习的，但也是最容易被滥用的，所以，我们在第 75 讲中还专门讲了如何避免过度设计。

经典的设计模式有 23 种，分三种类型：创建型、结构型和行为型。其中，创建型设计模式主要解决“对象的创建”问题，结构型设计模式主要解决“类或对象的组合”问题，行为型设计模式主要解决“类或对象之间的交互”问题。

虽然专栏中讲到的设计模式有很多种，但常用的并不多，主要有：单例、工厂、建造者、代理、装饰器、适配器、观察者、模板、策略、职责链、迭代器这 11 种，所以，你只要集中精力，把这 11 种搞明白就可以了，剩下的那 12 种稍微了解，混个眼熟，等到真正用到的时候，再深入地去研究学习就可以了。

课堂讨论

很多人反映学了就忘，对于上面的这些知识点，你记住了百分之多少呢？你是怎么克服学了就忘的问题的呢？

欢迎留言和我分享你的想法，如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

本文深入探讨了软件开发中的设计思想、原则和模式。首先强调了面向对象编程的重要性，介绍了面向对象的四大特性以及面向对象分析、设计和编程的重要性。其次，重点讲解了一些经典的设计原则，包括SOLID原则、DRY原则、KISS原则、YAGNI原则、LOD原则，以及它们的应用场景和指导作用。最后，强调了编码规范对代码可读性的重要性，指出可读性是编写高质量代码的首要条件。此外，文章还涉及了重构技巧和设计模式的内容，强调了持续重构的重要性以及设计模式在解决软件设计问题中的作用。总的来说，本文通过理论和实战两方面的回顾，帮助读者系统地了解了软件开发中的设计思想、原则和模式，为读者提供了全面的知识体系和实用的编程指导。

全部留言 (43)

最新　精选

，
2020-06-19

重复了多遍,不停地学不停地忘,这一年看了两本代码整洁之道,大部分都忘了,看了一本架构整洁之道,抽象的概念,原则,计算架构混乱度的公式,也都忘了,只记住了架构师的职责,架构师该做什么,看了一本深入理解java虚拟机三遍,还总结了脑图,就这样也还是忘得差不多了,c++因为工作中用到,看了一半c++primer,现在也都忘完了,effective java看了一遍,做了总结,还是忘完了,算法,撸了两遍数据结构与算法之美,刷了100多道题,现在也记不住多少了,mysql 45讲刷了一半多,也都忘得差不多了,许式伟的架构课刷了一遍,也就记住一些架构师做事的方式.....

大部分知识,只要"知道"就够了,关键时刻能想起他来,再去研究适不适合当前场景.想把细节都捯饬清楚不现实,而且容易忘

所以我的做法是忘就忘了,忘了说明他不重要,毕竟很多内容,平时工作都是凭直觉去思考就能想到,譬如重构一段老代码,或者起一个新模块,跟同事了解完前因后果,需求分析,自然而然就有了解决方案,设计原则,模式也就在不经意之中应用上了,根本不会在一开始就想"我要用某个设计模式"这种问题

然后,知识学到和会用,真的是两码事,前一阵做过一个udp客户端,向服务器发包和收包,我的做法就是发一次再收一次,然后这个模块就经常出现卡死,问题产生的原因很简单,就是udp丢包了,我一直就知道这件事情,选tcp和udp的时候还想过两者各自的优点和缺点,tcp信息传输稳定,但是需要考虑断点重连,udp不需要管理连接,但是需要防止丢包,即便想到这一点,代码依旧没有考虑到这个问题,把发包和收包给串行了,导致请求阻塞,然后给好几个同事看,大家也愣是没找到问题.这件事情让我感触很深,这个知识,你知道他,和你做事情的时候能用上他,真的就是两码事,所以我在总结出一个成长路线,一个对编程不了解的人新入行时,多搬砖成长的快,搬砖熟练到一定程度,补习基础成长的快,补到一定程度,知识开始盈余,实战不足,这时应该多搬砖,所以一个快速提升的途径应该是搬砖->学习->搬砖->学习.无限循环,搬砖的时候不要怕出错,搬砖是对学习成果的检验,学习时不要钻牛角尖,学习是为了学以致用,而不是"秀"给别人看,一定要钻研细节的话就等搬砖用到了,或者面试时再去钻研,忘掉的知识,感觉重要就复习,感觉不重要就忘掉好了

作者回复: 可以看看我写的这篇文章：
https://mp.weixin.qq.com/s/uKkQMlWTtAmvsEYZCxvZeg

共 8 条评论>

40

悟光
2020-06-19

记住了一半吧，但是和原来写的代码做对比发现代码质量真的有非常大的进步。我从争哥专栏一开始就学习了，这期间犯过这么几点错误。
1、看完面向对象的时候设计的时候感觉醍醐灌顶，看完文章两遍之后就觉得掌握了，但是在遇到新的需求做需求分析，设计类的时候还是感觉脑容量不够，有点不知所措，无从下手的感

觉。后面反思发现是想一开始就做比较完美的设计和类划分，想到的每个方案其实都有很明显的缺陷，这种矛盾的感觉导致烦躁和自我怀疑。
2、看完编码规范和重构之后，给变量起名的时候也过分纠结了，某种程度上降低效率。后面反思觉得完全可以用注释来提升可读性，没必要在细节上耗费太多精力。
3、看设计模式的时候，某这些瞬间感觉看到新世界，再联想到看过的一些优秀的源码激动的开始鼓掌🤣🤣，心里就起了强烈的实践一下的渴望于是在很小的功能里也用上设计模式😂😂，，导致有同事看我代码的时候有点费劲，尽管也激发了同事学习的兴趣，但是实际上也是一种孤独设计。
对第一种采取：先摈弃空想，有思路就动手画线框图，用大白话写文档，有个初出版，然后在反复揣摩争哥的例子，然后再去改，如此三四遍，开始写代码先完成需求。过一周再回头看看专栏，在做设计，在修改代码。在这个过程中很明显感受到自己对设计原则为什么那么定义有更真切直观的感受，并且感受到对业务的熟悉程度也影响设计的好坏，是循序渐进的过程。
2、编码规范和设计模式的学习，更多的看优秀框架，对框架的一部分觉得很感兴趣就在草稿纸上抄写主要的逻辑（个人觉得手写更慢，能留给大脑更多删减思考优秀优秀代码的精妙），并且对同一部分强迫自己看两三遍，因为一遍有一遍的发现新想法和对优秀代码钦佩的体验。这种感觉加深了记忆。
总结：1、找到自己想详细了解的部分反复看，并且隔段时间在作总结。2、通过手写加深细节记忆，提升理解，找到学习的动力。3、还是反复看和做总结。

作者回复: 👍

15

旅途
2020-11-29

我是用的主动回想 我在网上看 主动回想的学习方式 优于重复阅读和复习

作者回复: 嗯嗯 ❖❖❖❖❖❖

2

张细敏
2020-06-19

老师，后续有课程讲解aop吗?

作者回复: 你去我公众号里找找😂“小争哥”

2

Winon
2020-07-04

想请教老师关于模板方法模式的疑问，用这种设计模式是否也是一种面向过程做法？是否对于充血模型实适用性情况相对少一点呢？对面向过程的交易流程、步骤适用性情况比较多一点呢？

作者回复: 好像也没有，不用太纠结于这个。。。😂

1

Jxin
2020-06-19

记住大半。比较蠢，我是靠持续重构，眼里不容刺的把自己碰到的每行代码捋顺眼入的门。另外整理知识分享，教会别人，在准备的时候有时能理解得更透测。在解答别人问题时，有时能有眼前一亮新的理解。

总结，较真死磕，用心分享。

共 2 条评论>

66

岁月神偷
2020-06-19

对于一个重复的知识点，学了就忘的根本原因还是没有彻底弄明白，遇到这类情况，我想提出两个观点，一家之言，仅供参照。第一，对于难啃的知识，要学会迂回作战，在遇到非常大的困难时往往我们会陷入一种死磕到底的状态，问题当然是要解决的，但面对强敌，学会迂回也不失为一种战术，一个知识点非常难懂，很可能是这个知识点涵盖的其他知识点你没有掌握，由于对其他知识点的掌握不足，影响了自己的理解。所以，平稳心态继续学习，在某个时间点回过头来看的时候，有的难题真的会茅塞顿开。第二，实践是检验真理的唯一标准，如果学来的知识点一直无法实际运用，你很难说自己完全弄明白了，很多坑是要在实战过程中踩出来的，纸上谈兵的书生和经历过战场的老兵哪个更靠谱一些呢，所以不能一直浮在理论层面，一定要想法设法在工作实践中加以运用，以此来加深自己对知识的理解。当一个人真正搞明白一个知识点的时候，不存在记不记得住之说，同一句话可以有很多种说法，换一种说法他一样能够把这个知识点说明白。

共 1 条评论>

20

强哥
2020-06-21

“教”是最好的“学”，总结、实践并传授其他人。

5

俊辉

2020-06-28

边看边在 ProcessOn 写脑图，减少脑袋即时记忆负担，而且知识架构一目了然。学习、复习效率高了很多。

4

buggetout

2020-06-19

1个月把课看到这儿了，编码能力和设计能力大有提高。

4