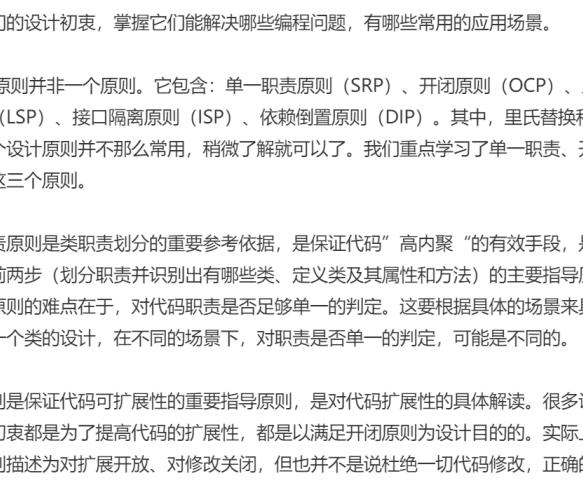
99 | 总结回顾：在实际软件开发中常用的设计思想、原则和模式

王争 2020-06-19

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。