1. 面向对象
   1. 面向对象的四大特性：封装、抽象、继承、多态
   2. 面向对象编程与面向过程编程的区别和联系
   3. 面向对象分析、面向对象设计、面向对象编程
   4. 接口和抽象类的区别以及各自的应用场景
   5. 基于接口而非实现编程的设计思想
   6. 多用组合少用继承的设计思想
   7. 面向过程的贫血模型和面向对象的充血模型
2. 设计原则

2.1 SOLID 原则

* SRP 单一职责原则
* OCP 开闭原则
* LSP 里式替换原则
* ISP 接口隔离原则
* DIP 依赖倒置原则

2.2 DRY 原则、KISS 原则、YAGNI 原则、LOD 法则

1. 设计模式

3.1 创建型

常用的有：单例模式、工厂模式（工厂方法和抽象工厂）、建造者模式。

不常用的有：原型模式。

3.2 结构型

常用的有：代理模式、桥接模式、装饰者模式、适配器模式。

不常用的有：门面模式、组合模式、享元模式。

3.3 行为型

常用的有：观察者模式、模板模式、策略模式、职责链模式、迭代器模式、状态模式。

不常用的有：访问者模式、备忘录模式、命令模式、解释器模式、中介模式。

1. 编程规范

编程规范主要解决的是代码的可读性问题。编码规范相对于设计原则、设计模式，更加具体、更加偏重代码细节。即便你可能对设计原则不熟悉、对设计模式不了解，但你最起码要掌握基本的编码规范，比如，如何给变量、类、函数命名，如何写代码注释，函数不宜过长、参数不能过多等等。对于编码规范，考虑到很多书籍已经讲得很好了（比如《重构》《代码大全》《代码整洁之道》等）。而且，每条编码规范都非常简单、非常明确，比较偏向于记忆，你只要照着来做可以。它不像设计原则，需要融入很多个人的理解和思考。所以，在这个专栏中，我并没有花太多的篇幅来讲解所有的编码规范，而是总结了我认为的最能改善代码质量的 20 条规范。如果你暂时没有时间去看那些经典的书籍，看我这些就够了。除此之外，专栏并没有将编码规范单独作为一个模块来讲解，而是跟重构放到了一起。之所以这样做，那是因为我把重构分为大重构和小重构两种类型，而小重构利用的知识基本上就是编码规范。除了编码规范，我们还会介绍一些代码的坏味道，让你知道什么样的代码是不符合规范的，应该如何优化。参照编码规范，你可以写出可读性好的代码；参照代码的坏味道，你可以找出代码存在的可读性问题。

1. 代码重构

在软件开发中，只要软件在不停地迭代，就没有一劳永逸的设计。随着需求的变化，代码的不停堆砌，原有的设计必定会存在这样那样的问题。针对这些问题，我们就需要进行代码重构。重构是软件开发中非常重要的一个环节。持续重构是保持代码质量不下降的有效手段，能有效避免代码腐化到无可救药的地步。而重构的工具就是我们前面罗列的那些面向对象设计思想、设计原则、设计模式、编码规范。实际上，设计思想、设计原则、设计模式一个最重要的应用场景就是在重构的时候。我们前面讲过，虽然使用设计模式可以提高代码的可扩展性，但过度不恰当地使用，也会增加代码的复杂度，影响代码的可读性。在开发初期，除非特别必须，我们一定不要过度设计，应用复杂的设计模式。而是当代码出现问题的时候，我们再针对问题，应用原则和模式进行重构。这样就能有效避免前期的过度设计。对于重构这部分内容，你需要掌握以下几个知识点：重构的目的（why）、对象（what）、时机（when）、方法（how）；保证重构不出错的技术手段：单元测试和代码的可测试性；两种不同规模的重构：大重构（大规模高层次）和小重构（小规模低层次）。希望你学完这部分内容之后，不仅仅是掌握一些重构技巧、套路，更重要的是建立持续重构意识，把重构当作开发的一部分，融入到日常的开发中。

1. 五者之间的联系

面向对象编程因为其具有丰富的特性（封装、抽象、继承、多态），可以实现很多复杂的设计思路，是很多设计原则、设计模式等编码实现的基础。设计原则是指导我们代码设计的一些经验总结，对于某些场景下，是否应该应用某种设计模式，具有指导意义。比如，“开闭原则”是很多设计模式（策略、模板等）的指导原则。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。应用设计模式的主要目的是提高代码的可扩展性。从抽象程度上来讲，设计原则比设计模式更抽象。设计模式更加具体、更加可执行。编程规范主要解决的是代码的可读性问题。编码规范相对于设计原则、设计模式，更加具体、更加偏重代码细节、更加能落地。持续的小重构依赖的理论基础主要就是编程规范。重构作为保持代码质量不下降的有效手段，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。实际上，面向对象、设计原则、设计模式、编程规范、代码重构，这五者都是保持或者提高代码质量的方法论，本质上都是服务于编写高质量代码这一件事的。当我们追本逐源，看清这个本质之后，很多事情怎么做就清楚了，很多选择怎么选也清楚了。比如，在某个场景下，该不该用这个设计模式，那就看能不能提高代码的可扩展性；要不要重构，那就看重代码是否存在可读、可维护问题等。

