编写高质量的代码

1. 面向对象
2. 什么是面向对象编程？**面向对象编程是一种编程范式或编程风格**。它以类或对象作为组织代码的基本单元，并将封装、抽象、继承、多态四个特性，作为代码设计和实现的基石 。
3. 什么是面向对象编程语言？**面向对象编程语言是支持类或对象的语法机制**，并有现成的语法机制，能方便地实现面向对象编程四大特性（封装、抽象、继承、多态）的编程语言。
4. 如何判定一个编程语言是否是面向对象编程语言？如果按照严格的的定义，需要有现成的语法支持**类、对象、四大特性**才能叫作面向对象编程语言。如果放宽要求的话，只要某种编程语言支持类、对象语法机制，那基本上就可以说这种编程语言是面向对象编程语言了，不一定非得要求具有所有的四大特性。
5. 面向对象编程和面向对象编程语言之间有何关系？面向对象编程一般使用面向对象编程语言来进行，但是，不用面向对象编程语言，我们照样可以进行面向对象编程。反过来讲，即便我们使用面向对象编程语言，写出来的代码也不一定是面向对象编程风格的，也有可能是面向过程编程风格的。
6. 什么是面向对象分析和面向对象设计？简单点讲，**面向对象分析就是要搞清楚做什么，面向对象设计就是要搞清楚怎么做**。两个阶段最终的产出是类的设计，包括程序被拆解为哪些类，每个类有哪些属性方法、类与类之间如何交互等等。
   1. 面向对象的四大特性：封装、抽象、继承、多态

**封装：**封装也叫作信息隐藏或者数据访问保护。类通过暴露有限的访问接口，授权外部仅能通过类提供的方式（或者叫函数）来访问内部信息或者数据。要实现这个功能需要依赖编程语言的访问权限控制。（Java 语言中的private、public 等关键字）。这就好比，如果一个冰箱有很多按钮，你就要研究很长时间，还不一定能操作正确。相反，如果只有几个必要的按钮，比如开、停、调节温度，你一眼就能知道该如何来操作，而且操作出错的概率也会降低很多。

**抽象：**封装主要讲的是如何隐藏信息、保护数据，而抽象讲的是如何隐藏方法的具体实现，让调用者只需要关心方法提供了哪些功能，并不需要知道这些功能是如何实现的。常用编程语言提供的接口类（比如 Java 中的 interface 关键字语法）或者抽象类（比如 Java 中的 abstract 关键字语法）这两种语法机制，来实现抽象这一特性。在面对复杂系统的时候，人脑能承受的信息复杂程度是有限的，所以我们必须忽略掉一些非关键性的实现细节。而抽象作为一种只关注功能点不关注实现的设计思路，正好帮我们的大脑过滤掉许多非必要的信息。很多设计原则都体现了抽象这种设计思想，比如基于接口而非实现编程、开闭原则（对扩展开放、对修改关闭）、代码解耦（降低代码的耦合性）等。我们在定义（或者叫命名）类的方法的时候，也要有抽象思维，不要在方法定义中，暴露太多的实现细节，以保证在某个时间点需要改变方法的实现逻辑的时候，不用去修改其定义。

**继承：**继承是用来表示类之间的 is-a 关系，比如猫是一种哺乳动物。从继承关系上来讲，继承可以分为两种模式，单继承和多继承。单继承表示一个子类只继承一个父类，多继承表示一个子类可以继承多个父类，比如猫既是哺乳动物，又是爬行动物。继承最大的一个好处就是代码复用。假如两个类有一些相同的属性和方法，我们就可以将这些相同的部分，抽取到父类中，让两个子类继承父类。我们通过继承来关联两个类，反应真实世界中的这种关系，非常符合人类的认知，而且，从设计的角度来说，也有一种结构美感。不过，过度使用继承，继承层次过深过复杂，就会导致代码可读性、可维护性变差。子类和父类高度耦合，修改父类的代码，会直接影响到子类。

**多态：**多态是指，子类可以替换父类，在实际的代码运行过程中，调用子类的方法实现。为了实现多态需要以下几种语法机制，第一个语法机制是编程语言要支持父类对象可以引用子类对象；第二个语法机制是编程语言要支持继承；第三个语法机制是编程语言要支持子类可以重写（override）父类中的方法。也就是说，只要两个类具有相同的方法，就可以实现多态，并不要求两个类之间有任何关系，这就是所谓的 duck-typing，是一些动态语言所特有的语法机制。而像 Java 这样的静态语言，通过继承实现多态特性，必须要求两个类之间有继承关系，通过接口实现多态特性，类必须实现对应的接口。多态特性能提高代码的可扩展性和复用性。多态也是很多设计模式、设计原则、编程技巧的代码实现基础，比如策略模式、基于接口而非实现编程、依赖倒置原则、里式替换原则、利用多态去掉冗长的 if-else 语句等等。

## 封装

What：隐藏信息，保护数据访问。

How：暴露有限接口和属性，需要编程语言提供访问控制的语法。

Why：提高代码可维护性；降低接口复杂度，提高类的易用性。

##抽象

What: 隐藏具体实现，使用者只需关心功能，无需关心实现。

How: 通过接口类或者抽象类实现，特殊语法机制非必须。

Why: 提高代码的扩展性、维护性；降低复杂度，减少细节负担。

##继承

What: 表示 is-a 关系，分为单继承和多继承。

How: 需要编程语言提供特殊语法机制。例如 Java 的 “extends”，C++ 的 “:” 。

Why: 解决代码复用问题。

##多态

What: 子类替换父类，在运行时调用子类的实现。

How: 需要编程语言提供特殊的语法机制。比如继承、接口类、duck-typing。

Why: 提高代码扩展性和复用性。

3W 模型的关键在于 Why，没有 Why，其它两个就没有存在的意义。从四大特性可以看出，面向对象的终极目的只有一个：可维护性。易扩展、易复用，降低复杂度等等都属于可维护性的实现方式。

* 1. 面向对象编程与面向过程编程的区别和联系

不要用面向对象语言编写面向过程风格的代码，一般有以下几种场景：

1. 滥用getter、setter方法

通过访问权限控制，隐藏内部数据，外部仅能通过类提供的有限的接口访问、修改内部数据，getter要避免集合容器内部被修改的风险。Collections.unmodifiableList方法，让 getter 方法返回一个不可被修改的UnmodifiableList集合容器。

1. 滥用全局变量和全局方法

常见的全局变量有单例类对象、静态成员变量、常量等，常见的全局方法有静态方法。静态成员变量归属于类上的数据，被所有的实例化对象所共享，也相当于一定程度上的全局变量。静态方法将方法与数据分离，破坏了封装特性，是典型的面向过程风格。Constants类太大容易引起编译时间过长，依赖的类都要重新编译。好的解决方案是：第一种是将 Constants 类拆解为功能更加单一的多个类，比如跟 MySQL 配置相关的常量，我们放到MysqlConstants类中；跟 Redis 配置相关的常量，我们放到RedisConstants类中。并不单独地设计 Constants 常量类，而是哪个类用到了某个常量，我们就把这个常量定义到这个类中。代替Utils类的方法是：第一种采用继承；第二种是定义不包含任何属性的静态方法类。

1. 定义数据和方法分离的类

传统的 MVC 结构分为 Model 层、Controller 层、View 层这三层。不过，在做前后端分离之后，三层结构在后端开发中，会稍微有些调整，被分为 Controller 层、Service 层、Repository 层。Controller 层负责暴露接口给前端调用，Service 层负责核心业务逻辑，Repository 层负责数据读写。而在每一层中，我们又会定义相应的 VO（View Object）、BO（Business Object）、Entity。一般情况下，VO、BO、Entity 中只会定义数据，不会定义方法，所有操作这些数据的业务逻辑都定义在对应的 Controller 类、Service 类、Repository 类中。这就是典型的面向过程的编程风格。实际上，这种开发模式叫作基于贫血模型的开发模式，也是我们现在非常常用的一种 Web 项目的开发模式。

在面向对象编程中，为什么容易写出面向过程风格的代码？

在生活中，你去完成一个任务，你一般都会思考，应该先做什么、后做什么，如何一步一步地顺序执行一系列操作，最后完成整个任务。面向过程编程风格恰恰符合人的这种流程化思维方式。而面向对象编程风格正好相反。它是一种自底向上的思考方式。它不是先去按照执行流程来分解任务，而是将任务翻译成一个一个的小的模块（也就是类），设计类之间的交互，最后按照流程将类组装起来，完成整个任务。

“贫血模型”的开发模式为什么会流行？

1）实现简单。Object仅仅作为传递数据的媒介，不用考虑过多的设计方面，将核心业务逻辑放到service层，用Hibernate之类的框架一套，完美解决任务。

2）上手快。使用贫血模式开发的web项目，新来的程序员看看代码就能“照猫画虎”干活了，不需要多高的技术水平。所以很多程序员干了几年，仅仅就会写CURD。

3）一些技术鼓励使用贫血模型。例如J2EE Entity Beans，Hibernate等。

* 1. 面向对象分析、面向对象设计、面向对象编程
  2. 接口和抽象类的区别以及各自的应用场景

抽象类的特性

1. 抽象类不允许被实例化，只能被继承；
2. 抽象类可以包含属性和方法；
3. 子类继承抽象类，必须实现抽象类中的所有抽象方法；

接口的特性

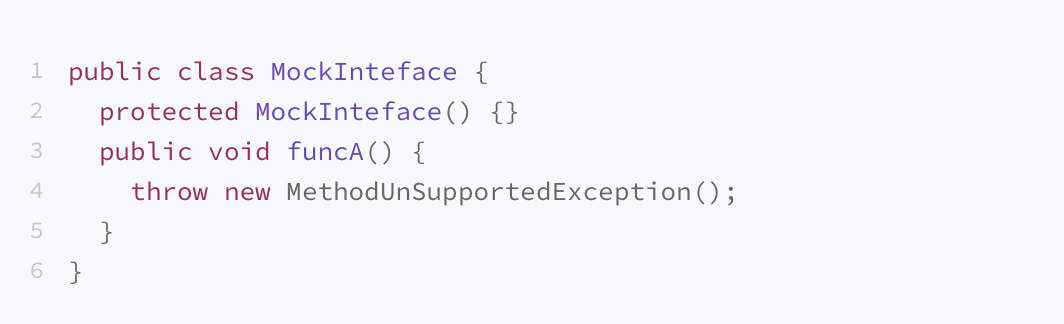
1. 接口不能包含属性（也就是成员变量）；
2. 接口只能声明方法，方法不能包含代码实现；
3. 类实现接口的时候，必须实现接口中声明的所有方法；

抽象类与接口的区别

从语法特性上对比，这两者有比较大的区别，比如抽象类中可以定义属性、方法的实现，而接口中不能定义属性，方法也不能包含代码实现等等。除了语法特性，从设计的角度，两者也有比较大的区别。抽象类实际上就是类，只不过是一种特殊的类，这种类不能被实例化为对象，只能被子类继承。我们知道，继承关系是一种 is-a 的关系，那抽象类既然属于类，也表示一种 is-a 的关系。相对于抽象类的 is-a 关系来说，接口表示一种 has-a 关系，表示具有某些功能。对于接口，有一个更加形象的叫法，那就是协议（contract）。

抽象类解决代码复用，采用多态实现优雅的代码复用；接口是为了实现解耦。





**如何用普通类模拟接口**我们可以让类中的方法抛出MethodUnSupportedException异常，来模拟不包含实现的接口，并且能强迫子类在继承这个父类的时候，都去主动实现父类的方法，否则就会在运行时抛出异常。那又如何避免这个类被实例化呢？实际上很简单，我们只需要将这个类的构造函数声明为 protected 访问权限就可以了。

**什么时候用接口、什么时候用抽象类**如果我们要表示一种 is-a 的关系，并且是为了解决代码复用的问题，我们就用抽象类；如果我们要表示一种 has-a 关系，并且是为了解决抽象而非代码复用的问题，那我们就可以使用接口。从类的继承层次上来看，抽象类是一种自下而上的设计思路，先有子类的代码重复，然后再抽象成上层的父类（也就是抽象类）。而接口正好相反，它是一种自上而下的设计思路。我们在编程的时候，一般都是先设计接口，再去考虑具体的实现。

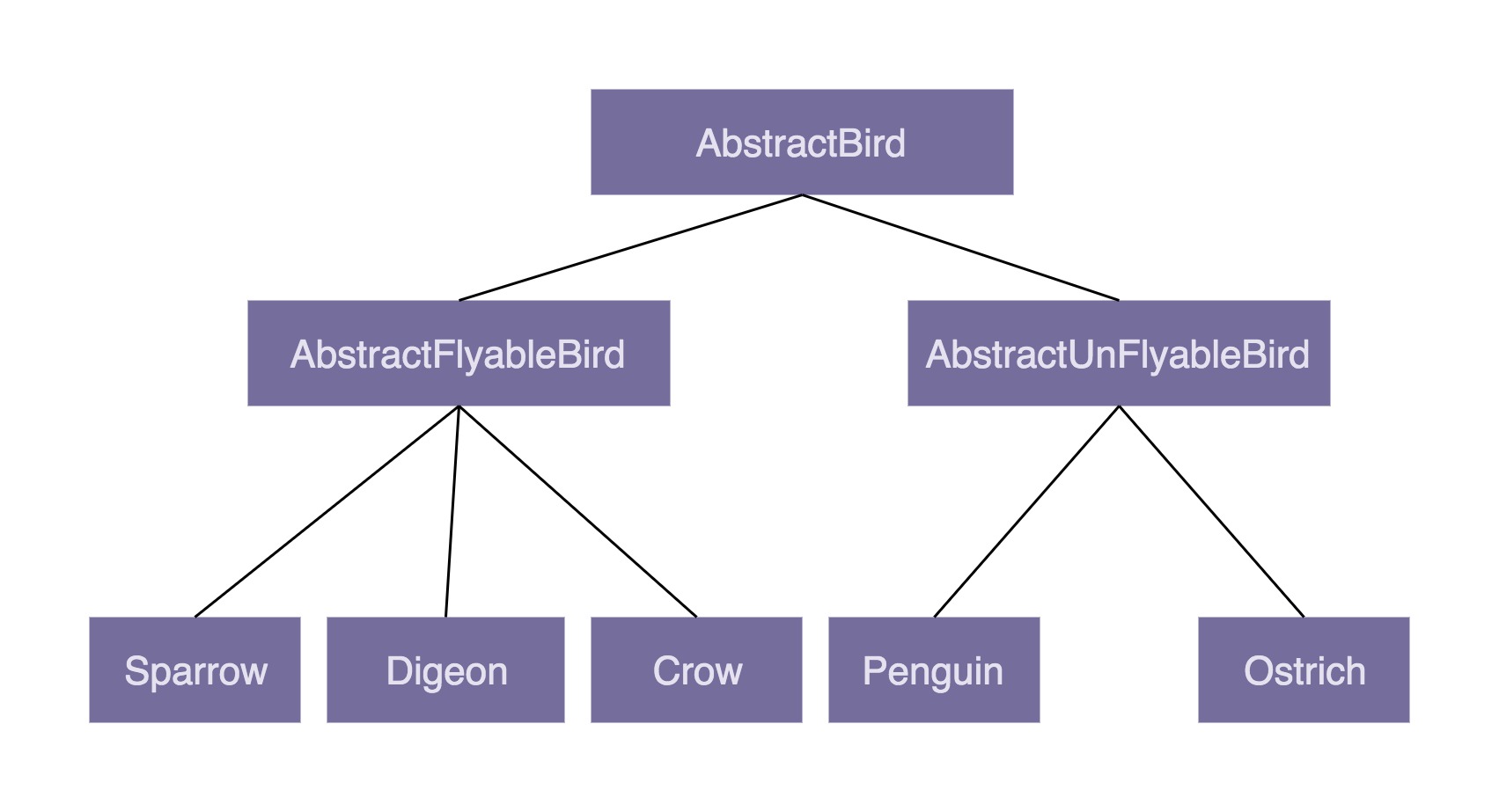
* 1. 基于接口而非实现编程的设计思想

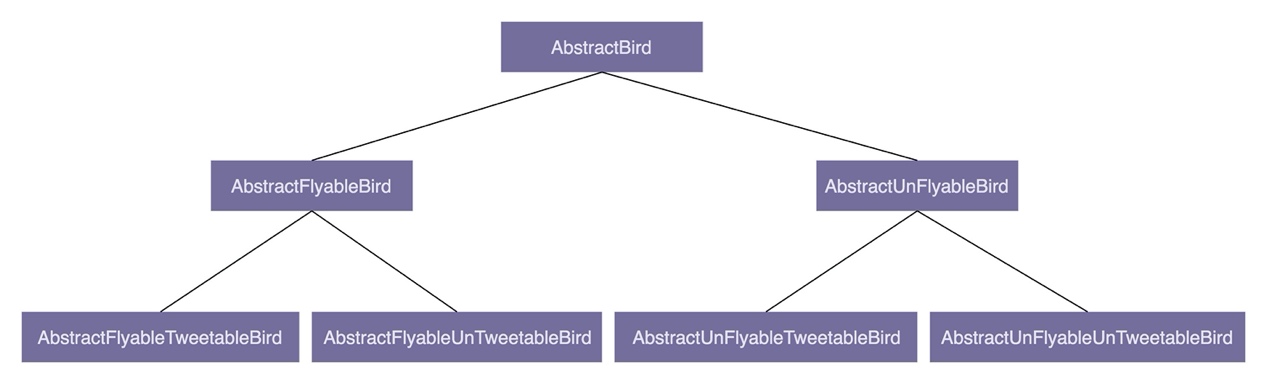
1. 函数的命名不能暴露任何实现细节。
2. 封装具体的实现细节。
3. 为实现类定义抽象的接口。

是否需要为每个类定义接口？这条原则的设计初衷是，将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。上游系统面向接口而非实现编程，不依赖不稳定的实现细节，这样当实现发生变化的时候，上游系统的代码基本上不需要做改动，以此来降低代码间的耦合性，提高代码的扩展性。从这个设计初衷上来看，如果在我们的业务场景中，某个功能只有一种实现方式，未来也不可能被其他实现方式替换，那我们就没有必要为其设计接口，也没有必要基于接口编程，直接使用实现类就可以了。除此之外，越是不稳定的系统，我们越是要在代码的扩展性、维护性上下功夫。相反，如果某个系统特别稳定，在开发完之后，基本上不需要做维护，那我们就没有必要为其扩展性，投入不必要的开发时间。

* 1. 多用组合少用继承的设计思想

**在面向对象编程中，有一条非常经典的设计原则，那就是：组合优于继承，多用组合少用继承。**





继承最大的问题就在于：继承层次过深、继承关系过于复杂会影响到代码的可读性和可维护性。实际上，我们可以利用组合（composition）、接口、委托（delegation）三个技术手段，一块儿来解决刚刚继承存在的问题。

我们前面讲到接口的时候说过，接口表示具有某种行为特性。针对“会飞”这样一个行为特性，我们可以定义一个 Flyable 接口，只让会飞的鸟去实现这个接口。对于会叫、会下蛋这些行为特性，我们可以类似地定义 Tweetable 接口、EggLayable 接口。我们将这个设计思路翻译成 Java 代码的话，就是下面这个样子：



不过，我们知道，接口只声明方法，不定义实现。也就是说，每个会下蛋的鸟都要实现一遍 layEgg() 方法，并且实现逻辑是一样的，这就会导致代码重复的问题。那这个问题又该如何解决呢？我们可以针对三个接口再定义三个实现类，它们分别是：实现了 fly() 方法的 FlyAbility 类、实现了 tweet() 方法的 TweetAbility 类、实现了 layEgg() 方法的 EggLayAbility 类。然后，通过组合和委托技术来消除代码重复。具体的代码实现如下所示：



**如果类之间的继承结构稳定（不会轻易改变），继承层次比较浅（比如，最多有两层继承关系），继承关系不复杂，我们就可以大胆地使用继承。反之，系统越不稳定，继承层次很深，继承关系复杂，我们就尽量使用组合来替代继承。**除此之外，还有一些设计模式会固定使用继承或者组合。比如，装饰者模式（decorator pattern）、策略模式（strategy pattern）、组合模式（composite pattern）等都使用了组合关系，而模板模式（template pattern）使用了继承关系。

还有一些特殊的场景要求我们必须使用继承。如果你不能改变一个函数的入参类型，而入参又非接口，为了支持多态，只能采用继承来实现。比如下面这样一段代码，其中 FeignClient 是一个外部类，我们没有权限去修改这部分代码，但是我们希望能重写这个类在运行时执行的 encode() 函数。这个时候，我们只能采用继承来实现了。



* 1. 面向过程的贫血模型和面向对象的充血模型

1. 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。

