设计模式
之美

王争（@小争哥）◎ 著

前 Google 工程师

- 23 种设计模式与范式实战精讲
- 200+ 真实案例分析设计与实现
- 顶尖互联网公司的编程经验分享
- 应对设计模式面试的思路与技巧

前言

什么是设计模式

设计模式讲的是如何写出可扩展、可读、可维护的高质量代码，所以，它们跟平时的编码会有直接的关系，也会直接影响到你的开发能力。

为什么要学习设计模式

1. 应对面试中的设计模式相关问题。
2. 告别写被人吐槽的烂代码。我见过太多的烂代码，比如命名不规范、类设计不合理、分层不清晰、没有模块化概念、代码结构混乱、高度耦合等等。这样的代码维护起来非常费劲，添加或者修改一个功能，常常会牵一发而动全身，让你无从下手，恨不得将全部的代码删掉重写！
3. 提高复杂代码的设计和开发能力。
4. 让读源码、学框架事半功倍。
5. 为你的职场发展做铺垫

如何评价代码质量的高低?

仔细看前面罗列的所有代码质量评价标准，你会发现，有些词语过于笼统、抽象，比较偏向对于整体的描述，比如优雅、好、坏、整洁、清晰等；有些过于细节、偏重方法论，比如模块化、高内聚低耦合、文档详尽、分层清晰等；有些可能并不仅仅局限于编码，跟架构设计等也有关系，比如可伸缩性、可用性、稳定性等。

为了做到有的放矢、有重点地学习，我挑选了其中几个最常用的、最重要的评价标准，来详细讲解，其中就包括：可维护性、可读性、可扩展性、灵活性、简洁性（简单、复杂）、可复用性、可测试性。接下来，我们逐一讲解一下。

代码质量的评价有很强的主观性，描述代码质量的词汇也有很多，比如可读性、可维护性、灵活、优雅、简洁等，这些词汇是从不同的维度去评价代码质量的。它们之间有互相作用，并不是独立的，比如，代码的可读性好、可扩展性好就意味着代码的可维护性好。代码质量高低是一个综合各种因素得到的结论。我们并不能通过单一的维度去评价一段代码的好坏。

1. 可维护性（maintainability），落实到编码开发，所谓的“维护”无外乎就是修改 bug、修改老的代码、添加新的代码之类的工作。所谓“代码易维护”就是指，在不破坏原有代码设计、不引入新的 bug 的情况下，能够快速地修改或者添加代码。所谓“代码不易维护”就是指，修改或者添加代码需要冒着极大的引入新 bug 的风险，并且需要花费很长的时间才能完成。
2. 可读性（readability），我们在编写代码的时候，时刻要考虑到代码是否易读、易理解。除此之外，代码的可读性在非常大程度上会影响代码的可维护性。毕竟，不管是修改 bug，还是修改添加功能代码，我们首先要做的事情就是读懂代码。代码读不大懂，就很有可能因为考虑不周全，而引入新的 bug。我们需要看代码是否符合编码规范、命名是否达意、注释是否详尽、函数是否长短合适、模块划分是否清晰、是否符合高内聚低耦合等等。你应该也能感觉到，从正面上，我们很难给出一个覆盖所有评价指标的列表。这也是我们无法量化可读性的原因。
3. 可扩展性（extensibility），代码的可扩展性表示，我们在不修改或少量修改原有代码的情况下，通过扩展的方式添加新的功能代码。说直白点就是，代码预留了一些功能扩展点，你可以把新功能代码，直接插到扩展点上，而不需要因为要添加一个功能而大动干戈，改动大量的原始代码。

4. 灵活性（flexibility），从刚刚举的场景来看，如果一段代码易扩展、易复用或者易用，我们都可以称这段代码写得比较灵活。所以，灵活这个词的含义非常宽泛，很多场景下都可以使用。
5. 简洁性（simplicity），有一条非常著名的设计原则，你一定听过，那就是 KISS 原则：“Keep It Simple，Stupid”。这个原则说的意思就是，尽量保持代码简单。代码简单、逻辑清晰，也就意味着易读、易维护。我们在编写代码的时候，往往也会把简单、清晰放到首位。不过，很多编程经验不足的程序员会觉得，简单的代码没有技术含量，喜欢在项目中引入一些复杂的设计模式，觉得这样才能体现自己的技术水平。实际上，思从深而行从简，真正的高手能云淡风轻地用最简单的方法解决最复杂的问题。这也是一个编程老手跟编程新手的本质区别之一。
6. 可复用性，代码的可复用性可以简单地理解为，尽量减少重复代码的编写，复用已有的代码。
7. 可测试性，相对于前面六个评价标准，代码的可测试性是一个相对较少被提及，但又非常重要的代码质量评价标准。代码可测试性的好坏，能从侧面上非常准确地反应代码质量的好坏。代码的可测试性差，比较难写单元测试，那基本上就能说明代码设计得有问题。

最常用的评价标准有哪几个?

最常用到几个评判代码质量的标准是：可维护性、可读性、可扩展性、灵活性、简洁性、可复用性、可测试性。其中，可维护性、可读性、可扩展性又是提到最多的、最重要的三个评价标准。

面向对象、设计原则、设计模式、编程规范、重构，这五者有何关系?

- 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。

极客时间

面向对象

编程范式

1. 面向过程
2. 面向对象
3. 函数式编程

什么是面向对象编程?

面向对象编程的英文缩写是 OOP，全称是 Object Oriented Programming。对应地，面向对象编程语言的英文缩写是 OOPL，全称是 Object Oriented Programming Language。

面向对象编程中有两个非常重要、非常基础的概念，那就是类（class）和对象（object）。这两个概念最早出现在 1960 年，在 Simula 这种编程语言中第一次使用。而面向对象编程这个概念第一次被使用是在 Smalltalk 这种编程语言中。Smalltalk 被认为是第一个真正意义上的面向对象编程语言。

如果非得给出一个定义的话，我觉得可以用下面两句话来概括

- 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 。
- 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。

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 。

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。

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。

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。

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。

面向对象四大特性

封装（Encapsulation）

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来访问内部信息或者数据。它需要编程语言提供权限访问控制语法来支持，例如 Java 中的 private、protected、public 关键字。封装特性存在的意义，一方面是保护数据不被随意修改，提高代码的可维护性；另一方面是仅暴露有限的必要接口，提高类的易用性。

抽象（Abstraction）

封装主要讲如何隐藏信息、保护数据，那抽象就是讲如何隐藏方法的具体实现，让使用者只需要关心方法提供了哪些功能，不需要知道这些功能是如何实现的。抽象可以通过接口类或者抽象类来实现，但也并不需要特殊的语法机制来支持。抽象存在的意义，一方面是提高代码的可扩展性、维护性，修改实现不需要改变定义，减少代码的改动范围；另一方面，它也是处理复杂系统的有效手段，能有效地过滤掉不必要关注的信息。

继承（Inheritance）

继承是用来表示类之间的 is-a 关系，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。为了实现继承这个特性，编程语言需要提供特殊的语法机制来支持。继承主要是用来解决代码复用的问题。

多态（Polymorphism）

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。多态这种特性也需要编程语言提供特殊的语法机制来实现，比如继承、接口类、duck-typing。多态可以提高代码的扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础。

Clean Architecture 里面指出多态是函数指针的一种应用。并用getchar()举了例子。然后用多态的实现引出了“依赖反转”的例子。

面向对象编程相比面向过程编程有哪些优势?

- 对于大规模复杂程序的开发，程序的处理流程并非单一的一条主线，而是错综复杂的网状结构。面向对象编程比起面向过程编程，更能应对这种复杂类型的程序开发。
- 面向对象编程相比面向过程编程，具有更加丰富的特性（封装、抽象、继承、多态）。利用这些特性编写出来的代码，更加易扩展、易复用、易维护。
- 从编程语言跟机器打交道的方式的演进规律中，我们可以总结出：面向对象编程语言比起面向过程编程语言，更加人性化、更加高级、更加智能。

Constants、Utils 类设计问题

定义一个如此大而全的 Constants 类，并不是一种很好的设计思路。为什么这么说呢？原因主要有以下几点。

- 首先，这样的设计会影响代码的可维护性。如果参与开发同一个项目的工程师有很多，在开发过程中，可能都要涉及修改这个类，比如往这个类里添加常量，那这个类就会变得越来越大，成百上千行都有可能，查找修改某个常量也会变得比较费时，而且还会增加提交代码冲突的概率。
- 其次，这样的设计还会增加代码的编译时间。当 Constants 类中包含很多常量定义的时候，依赖这个类的代码就会很多。那每次修改 Constants 类，都会导致依赖它的类文件重新编译，因此会浪费很多不必要的编译时间。不要小看编译花费的时间，对于一个非常大的工程项目来说，编译一次项目花费的时间可能是几分钟，甚至几十分钟。而我们在开发过程中，每次运行单元测试，都会触发一次编译的过程，这个编译时间就有可能会影响到我们的开发效率。
- 最后，这样的设计还会影响代码的复用性。如果我们要在另一个项目中，复用本项目开发的某个类，而这个类又依赖 Constants 类。即便这个类只依赖 Constants 类中的一小部分常量，我们仍然需要把整个 Constants 类也一并引入，也就引入了很多无关的常量到新的项目中。

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。当然，还有一种我个人觉得更好的设计思路，那就是并不单独地设计 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。比如，RedisConfig 类用到了 Redis 配置相关的常量，那我们就直接将这些常量定义在 RedisConfig 中，这样也提高了类设计的内聚性和代码的复用性。

实际上，Utils 类的出现是基于这样一个问题背景：如果我们有两个类 A 和 B，它们要用到一块相同的功能逻辑，为了避免代码重复，我们不应该在两个类中，将这个相同的功能逻辑，重复地实现两遍。这个时候我们该怎么办呢？

我们在讲面向对象特性的时候，讲过继承可以实现代码复用。利用继承特性，我们把相同的属性和方法，抽取出来，定义到父类中。子类复用父类中的属性和方法，达到代码复用的目的。但是，有的时候，从业务含义上，A 类和 B 类并不一定具有继承关系，比如 Crawler 类和 PageAnalyzer 类，它们都用到了 URL 拼接和分割的功能，但并不具有继承关系（既不是父子关系，也不是兄弟关系）。仅仅为了代码复用，生硬地抽象出一个父类出来，会影响到代码的可读性。如果不熟悉背后设计思路的同事，发现 Crawler 类和 PageAnalyzer 类继承同一个父类，而父类中定义的却是 URL 相关的操作，会觉得这个代码写得莫名其妙，理解不了。

实际上，只包含静态方法不包含任何属性的 Utils 类，是彻彻底底的面向过程的编程风格。但这并不是说，我们就要杜绝使用 Utils 类了。实际上，从刚刚讲的 Utils 类存在的目的来看，它在软件开发中还是挺有用的，能解决代码复用问题。所以，这里并不是说完全不能用 Utils 类，而是说，要尽量避免滥用，不要不加思考地随意去定义 Utils 类。

在定义 Utils 类之前，你要问一下自己，你真的需要单独定义这样一个 Utils 类吗？是否可以把 Utils 类中的某些方法定义到其他类中呢？如果在回答完这些问题之后，你还是觉得确实有必要去定义这样一个 Utils 类，那就大胆地去定义它吧。因为即便在面向对象编程中，我们也并不是完全排斥面向过程风格的代码。只要它能为我们写出好的代码贡献力量，我们就可以适度地去使用。

除此之外，类比 Constants 类的设计，我们设计 Utils 类的时候，最好也能细化一下，针对不同的功能，设计不同的 Utils 类，比如 FileUtils、IOUtils、StringUtils、UrlUtils 等，不要设计一个过于大而全的 Utils 类。

抽象类和接口

抽象类

- 抽象类不允许被实例化，只能被继承。也就是说，你不能 new 一个抽象类的对象出来（Logger logger = new Logger(...); 会报编译错误）。
- 抽象类可以包含属性和方法。方法既可以包含代码实现（比如 Logger 中的 log() 方法），也可以不包含代码实现（比如 Logger 中的 doLog() 方法）。不包含代码实现的方法叫作抽象方法。
- 子类继承抽象类，必须实现抽象类中的所有抽象方法。对应到例子代码中就是，所有继承 Logger 抽象类的子类，都必须重写 doLog() 方法。

接口特性

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

前面我们讲了抽象类和接口的定义，以及各自的语法特性。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）。

抽象类和接口能解决什么编程问题

抽象类更多的是为了代码复用，而接口就更侧重于解耦。接口是对行为的一种抽象，**相当于一组协议或者契约**，你可以联想类比一下 API 接口。调用者只需要关注抽象的接口，不需要了解具体的实现，具体的实现代码对调用者透明。接口实现了约定和实现相分离，可以降低代码间的耦合性，提高代码的可扩展性。

抽象类是对成员变量和方法的抽象，是一种 is-a 关系，是为了解决代码复用问题。接口仅仅是对方法的抽象，是一种 has-a 关系，表示具有某一组行为特性，是为了解决解耦问题，隔离接口和具体的实现，提高代码的扩展性。

抽象类和接口的应用场景区别

什么时候该用抽象类？什么时候该用接口？实际上，判断的标准很简单。如果要表示一种 is-a 的关系，并且是为了解决代码复用问题，我们就用抽象类；如果要表示一种 has-a 关系，并且是为了解决抽象而非代码复用问题，那我们就用接口。

基于接口而非实现编程

1. 实际上，“基于接口而非实现编程”这条原则的另一个表述方式，是“基于抽象而非实现编程”。后者的表述方式其实更能体现这条原则的设计初衷。在软件开发中，最大的挑战之一就是需求的不断变化，这也是考验代码设计好坏的一个标准。**越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性，越能应对未来的需求变化。好的代码设计，不仅能应对当下的需求，而且在将来需求发生变化的时候，仍然能够在不破坏原有代码设计的情况下灵活应对**。而抽象就是提高代码扩展性、灵活性、可维护性最有效的手段之一。
2. 我们在定义接口的时候，一方面，命名要足够通用，不能包含跟具体实现相关的字眼；另一方面，与特定实现有关的方法不要定义在接口中。
3. “基于接口而非实现编程”这条原则，不仅仅可以指导非常细节的编程开发，还能指导更加上层的架构设计、系统设计等。比如，服务端与客户端之间的“接口”设计、类库的“接口”设计。

是否需要为每个类定义接口?

做任何事情都要讲求一个“度”，过度使用这条原则，非得给每个类都定义接口，接口满天飞，也会导致不必要的开发负担。至于什么时候，该为某个类定义接口，实现基于接口的编程，什么时候不需要定义接口，直接使用实现类编程，我们做权衡的根本依据，还是要回归到设计原则诞生的初衷上来。只要搞清楚了这条原则是为了解决什么样的问题而产生的，你就会发现，很多之前模棱两可的问题，都会变得豁然开朗。

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。

为什么不推荐使用继承?

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。为什么不推荐使用继承？组合相比继承有哪些优势？如何判断该用组合还是继承？今天，我们就围绕着这三个问题，来详细讲解一下这条设计原则。

继承是面向对象的四大特性之一，用来表示类之间的 is-a 关系，可以解决代码复用的问题。虽然继承有诸多作用，但继承层次过深、过复杂，也会影响到代码的可维护性。在这种情况下，我们应该尽量少用，甚至不用继承。

组合相比继承有哪些优势?

我们知道继承主要有三个作用：表示 is-a 关系，支持多态特性，代码复用。而这三个作用都可以通过其他技术手段来达成。比如 is-a 关系，我们可以通过组合和接口的 has-a 关系来替代；多态特性我们可以利用接口来实现；代码复用我们可以通过组合和委托来实现。所以，从理论上讲，通过组合、接口、委托三个技术手段，我们完全可以替换掉继承，在项目中不用或者少用继承关系，特别是一些复杂的继承关系。

如何判断该用组合还是继承?

尽管我们鼓励多用组合少用继承，但组合也并不是完美的，继承也并非一无是处。从上面的例子来看，继承改写成组合意味着要做更细粒度的类的拆分。这也就意味着，我们要定义更多的类和接口。类和接口的增多也就或多或少地增加代码的复杂程度和维护成本。所以，在实际的项目开发中，我们还是要根据具体的情况，来具体选择该用继承还是组合。

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。

尽管有些人说，要杜绝继承，100% 用组合代替继承，但是我的观点没那么极端！之所以“多用组合少用继承”这个口号喊得这么响，只是因为，长期以来，我们过度使用继承。还是那句话，组合并不完美，继承也不是一无是处。只要我们控制好它们的副作用、发挥它们各自的优势，在不同的场合下，恰当地选择使用继承还是组合，这才是我们所追求的境界。

设计原则

SOLID 原则

单一职责（Single Responsibility Principle）

单一职责（SRP）：Single Responsibility Principle，一个类只负责完成一个职责或者功能。不要设计大而全的类，要设计粒度小、功能单一的类。单一职责原则是为了实现代码高内聚、低耦合，提高代码的复用性、可读性、可维护性。

```
public class UserInfo {
  private long userId;
  private String username;
  private String email;
  private String telephone;
  private long createTime;
  private long lastLoginTime;
  private String avatarUrl;
  private String provinceOfAddress; // 省
```

```
  private String cityOfAddress; // 市
  private String regionOfAddress; // 区
  private String detailedAddress; // 详细地址
  // ...省略其他属性和方法...
}
```

对于这个问题，有两种不同的观点。一种观点是，UserInfo 类包含的都是跟用户相关的信息，所有的属性和方法都隶属于用户这样一个业务模型，满足单一职责原则；另一种观点是，地址信息在 UserInfo 类中，所占的比重比较高，可以继续拆分成独立的 UserAddress 类，UserInfo 只保留除 Address 之外的其他信息，拆分之后的两个类的职责更加单一。

哪种观点更对呢？实际上，要从中做出选择，我们不能脱离具体的应用场景。如果在这个社交产品中，用户的地址信息跟其他信息一样，只是单纯地用来展示，那 UserInfo 现在的设计就是合理的。但是，如果这个社交产品发展得比较好，之后又在产品中添加了电商的模块，用户的地址信息还会用在电商物流中，那我们最好将地址信息从 UserInfo 中拆分出来，独立成用户物流信息（或者叫地址信息、收货信息等）。

从刚刚这个例子，我们可以总结出，不同的应用场景、不同阶段的需求背景下，对同一个类的职责是否单一的判定，可能都是不一样的。在某种应用场景或者当下的需求背景下，一个类的设计可能已经满足单一职责原则了，但如果换个应用场景或着在未来的某个需求背景下，可能就不满足了，需要继续拆分成粒度更细的类。

综上所述，评价一个类的职责是否足够单一，我们并没有一个非常明确的、可以量化的标准，可以说，这是件非常主观、仁者见仁智者见智的事情。实际上，在真正的软件开发中，我们也没必要过于未雨绸缪，过度设计。所以，我们可以先写一个粗粒度的类，满足业务需求。随着业务的发展，如果粗粒度的类越来越庞大，代码越来越多，这个时候，我们就可以将这个粗粒度的类，拆分成几个更细粒度的类。这就是所谓的持续重构（后面的章节中我们会讲到）。

听到这里，你可能会说，这个原则如此含糊不清、模棱两可，到底该如何拿捏才好啊？我这里还有一些小技巧，能够很好地帮你，从侧面上判定一个类的职责是否够单一。而且，我个人觉得，下面这几条判断原则，比起很主观地去思考类是否职责单一，要更有指导意义、更具有可执行性：

- 类中的代码行数、函数或属性过多，会影响代码的可读性和可维护性，我们就需要考虑对类进行拆分；
- 类依赖的其他类过多，或者依赖类的其他类过多，不符合高内聚、低耦合的设计思想，我们就需要考虑对类进行拆分；
- 私有方法过多，我们就要考虑能否将私有方法独立到新的类中，设置为 public 方法，供更多的类使用，从而提高代码的复用性；
- 比较难给类起一个合适名字，很难用一个业务名词概括，或者只能用一些笼统的 Manager、Context 之类的词语来命名，这就说明类的职责定义得可能不够清晰；
- 类中大量的方法都是集中操作类中的某几个属性，比如，在 UserInfo 例子中，如果一半的方法都是在操作 address 信息，那就可以考虑将这几个属性和对应的方法拆分出来。

类的职责是否设计得越单一越好?

单一职责原则通过避免设计大而全的类，避免将不相关的功能耦合在一起，来提高类的内聚性。同时，类职责单一，类依赖的和被依赖的其他类也会变少，减少了代码的耦合性，以此来实现代码的高内聚、低耦合。但是，如果拆分得过细，实际上会适得其反，反倒会降低内聚性，也会影响代码的可维护性。

开闭原则（Open Closed Principle）

开闭原则（OCP）：Open Closed Principle，对扩展开放，对修改关闭。添加一个新的功能，应该是通过在已有代码基础上扩展代码（新增模块、类、方法、属性等），而非修改已有代码（修改模块、类、方法、属性等）的方式来完成。关于定义，我们有两点要注意。第一点是，开闭原则并不是说完全杜绝修改，而是以最小的修改代码的代价来完成新功能的开发。第二点是，同样的代码改动，在粗代码粒度下，可能被认定为“修改”；在细代码粒度下，可能又被认定为“扩展”。

而且，我们要认识到，添加一个新功能，不可能任何模块、类、方法的代码都不“修改”，这个是做不到的。类需要创建、组装、并且做一些初始化操作，才能构建成可运行的的程序，这部分代码的修改是在所难免的。**我们要做的是尽量让修改操作更集中、更少、更上层，尽量让最核心、最复杂的那部分逻辑代码满足开闭原则。**

前面我们提到，写出支持“对扩展开放、对修改关闭”的代码的关键是预留扩展点。那问题是如何才能识别出所有可能的扩展点呢？

如果你开发的是一个业务导向的系统，比如金融系统、电商系统、物流系统等，要想识别出尽可能多的扩展点，就要对业务有足够的了解，能够知道当下以及未来可能要支持的业务需求。如果你开发的是跟业务无关的、通用的、偏底层的系统，比如，框架、组件、类库，你需要了解“它们会被如何使用？今后你打算添加哪些功能？使用者未来会有哪些更多的功能需求？”等问题。

不过，有一句话说得好，“唯一不变的只有变化本身”。即便我们对业务、对系统有足够的了解，那也不可能识别出所有的扩展点，即便你能识别出所有的扩展点，为这些地方都预留扩展点，这样做的成本也是不可接受的。我们没必要为一些遥远的、不一定发生的需求去提前买单，做过度设计。

最合理的做法是，对于一些比较确定的、短期内可能就会扩展，或者需求改动对代码结构影响比较大的情况，或者实现成本不高的扩展点，在编写代码的时候之后，我们就可以事先做些扩展性设计。但对于一些不确定未来是否要支持的需求，或者实现起来比较复杂的扩展点，我们可以等到有需求驱动的时候，再通过重构代码的方式来支持扩展的需求。

而且，开闭原则也并不是免费的。有些情况下，代码的扩展性会跟可读性相冲突。比如，我们之前举的 Alert 告警的例子。为了更好地支持扩展性，我们对代码进行了重构，重构之后的代码要比之前的代码复杂很多，理解起来也更加有难度。很多时候，我们都需要在扩展性和可读性之间做权衡。在某些场景下，代码的扩展性很重要，我们就可以适当地牺牲一些代码的可读性；在另一些场景下，代码的可读性更加重要，那我们就适当地牺牲一些代码的可扩展性。

1. 如何理解“对扩展开放、对修改关闭”？

添加一个新的功能，应该是通过在已有代码基础上扩展代码（新增模块、类、方法、属性等），而非修改已有代码（修改模块、类、方法、属性等）的方式来完成。关于定义，我们有两点要注意。第一点是，开闭原则并不是说完全杜绝修改，而是以最小的修改代码的代价来完成新功能的开发。第二点是，同样的代码改动，在粗代码粒度下，可能被认定为“修改”；在细代码粒度下，可能又被认定为“扩展”。

1. 如何做到“对扩展开放、修改关闭”？

我们要时刻具备扩展意识、抽象意识、封装意识。在写代码的时候，我们要多花点时间思考一下，这段代码未来可能有哪些需求变更，如何设计代码结构，事先留好扩展点，以便在未来需求变更的时候，在不改动代码整体结构、做到最小代码改动的情况下，将新的代码灵活地插入到扩展点上。

很多设计原则、设计思想、设计模式，都是以提高代码的扩展性为最终目的的。特别是 23 种经典设计模式，大部分都是为了解决代码的扩展性问题而总结出来的，都是以开闭原则为指导原则的。最常用来提高代码扩展性的方法有：多态、依赖注入、基于接口而非实现编程，以及大部分的设计模式（比如，装饰、策略、模板、职责链、状态）。

里式替换（Liskov Substitution Principle）

里式替换（LSP）：子类对象（object of subtype/derived class）能够替换程序（program）中父类对象（object of base/parent class）出现的任何地方，并且保证原来程序的逻辑行为（behavior）不变及正确性不被破坏。举例： 是拿父类的单元测试去验证子类的代码。如果某些单元测试运行失败，就有可能说明，子类的设计实现没有完全地遵守父类的约定，子类有可能违背了里式替换原则。

里式替换原则是用来指导，继承关系中子类该如何设计的一个原则。理解里式替换原则，最核心的就是理解“design by contract，按照协议来设计”这几个字。父类定义了函数的“约定”（或者叫协议），那子类可以改变函数的内部实现逻辑，但不能改变函数原有的“约定”。这里的约定包括：函数声明要实现的功能；对输入、输出、异常的约定；甚至包括注释中所罗列的任何特殊说明。

理解这个原则，我们还要弄明白里式替换原则跟多态的区别。虽然从定义描述和代码实现上来看，多态和里式替换有点类似，但它们关注的角度是不一样的。多态是面向对象编程的一大特性，也是面向对象编程语言的一种语法。它是一种代码实现的思路。而里式替换是一种设计原则，用来指导继承关系中子类该如何设计，子类的设计要保证在替换父类的时候，不改变原有程序的逻辑及不破坏原有程序的正确性。

接口隔离原则（Interface Segregation Principle）

接口隔离原则（ISP）：调用方不应该被强迫依赖它不需要的接口。举例： Config 接口拆分为Updater 和 Viewer 两个接口

```
public interface Updater {
  void update();
}

public interface Viewer {
  String outputInPlainText();
  Map<String, String> output();
}

public class RedisConfig implemets Updater, Viewer {
  //...省略其他属性和方法...
  @Override
  public void update() { //... }
  @Override
  public String outputInPlainText() { //... }
  @Override
  public Map<String, String> output() { //...}
}

public class KafkaConfig implements Updater {
  //...省略其他属性和方法...
  @Override
  public void update() { //... }
}

public class MysqlConfig implements Viewer {
  //...省略其他属性和方法...
  @Override
```

```
  public String outputInPlainText() { //... }
  @Override
  public Map<String, String> output() { //...}
}

public class SimpleHttpServer {
  private String host;
  private int port;
  private Map<String, List<Viewer>> viewers = new HashMap<>();

  public SimpleHttpServer(String host, int port) {//...}

  public void addViewers(String urlDirectory, Viewer viewer) {
    if (!viewers.containsKey(urlDirectory)) {
      viewers.put(urlDirectory, new ArrayList<Viewer>());
    }
    this.viewers.get(urlDirectory).add(viewer);
  }

  public void run() { //... }
}

public class Application {
    ConfigSource configSource = new ZookeeperConfigSource();
    public static final RedisConfig redisConfig = new RedisConfig(configSource);
    public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
    public static final MySqlConfig mysqlConfig = new MySqlConfig(configSource);

    public static void main(String[] args) {
        ScheduledUpdater redisConfigUpdater =
            new ScheduledUpdater(redisConfig, 300, 300);
        redisConfigUpdater.run();

        ScheduledUpdater kafkaConfigUpdater =
            new ScheduledUpdater(kafkaConfig, 60, 60);
        redisConfigUpdater.run();

        SimpleHttpServer simpleHttpServer = new SimpleHttpServer(“127.0.0.1”, 2389);
        simpleHttpServer.addViewer("/config", redisConfig);
        simpleHttpServer.addViewer("/config", mysqlConfig);
        simpleHttpServer.run();
    }
}
```

依赖反转原则（Dependency Inversion Principle）

依赖反转原则（DIP）： Dependency Inversion Principle 高层模块（high-level modules）不要依赖低层模块（low-level）。高层模块和低层模块应该通过抽象（abstractions）来互相依赖。除此之外，抽象（abstractions）不要依赖具体实现细节（details），具体实现细节（details）依赖抽象（abstractions）。举例Tomcat和Java WebApp，两者都依赖同一个“抽象”，也就是 Servlet 规范。

1. 控制反转

控制反转（IOC）: Inversion Of Control，框架提供了一个可扩展的代码骨架，用来组装对象、管理整个执行流程。程序员利用框架进行开发的时候，只需要往预留的扩展点上，添加跟自己业务相关的代码，就可以利用框架来驱动整个程序流程的执行。

实际上，控制反转是一个比较笼统的设计思想，并不是一种具体的实现方法，一般用来指导框架层面的设计。这里所说的“控制”指的是对程序执行流程的控制，而“反转”指的是在没有使用框架之前，程序员自己控制整个程序的执行。在使用框架之后，整个程序的执行流程通过框架来控制。流程的控制权从程序员“反转”给了框架。

1. 依赖注入

依赖注入（DI）: Dependency Injection 依赖注入的方式来将依赖的类对象传递进来，这样就提高了代码的扩展性，我们可以灵活地替换依赖的类

依赖注入和控制反转恰恰相反，它是一种具体的编码技巧。我们不通过 new 的方式在类内部创建依赖类的对象，而是将依赖的类对象在外部创建好之后，通过构造函数、函数参数等方式传递（或注入）给类来使用。

1. 依赖注入框架

我们通过依赖注入框架提供的扩展点，简单配置一下所有需要的类及其类与类之间依赖关系，就可以实现由框架来自动创建对象、管理对象的生命周期、依赖注入等原本需要程序员来做的事情。

1. 依赖反转原则

依赖反转原则也叫作依赖倒置原则。这条原则跟控制反转有点类似，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。抽象不要依赖具体实现细节，具体实现细节依赖抽象。

KISS原则

KISS 原则是保持代码可读和可维护的重要手段。KISS 原则中的“简单”并不是以代码行数来考量的。代码行数越少并不代表代码越简单，我们还要考虑逻辑复杂度、实现难度、代码的可读性等。而且，本身就复杂的问题，用复杂的方法解决，并不违背 KISS 原则。除此之外，同样的代码，在某个业务场景下满足 KISS 原则，换一个应用场景可能就不满足了。

对于如何写出满足 KISS 原则的代码，我还总结了下面几条指导原则：

1. 不要使用同事可能不懂的技术来实现代码。比如前面例子中的正则表达式，还有一些编程语言中过于高级的语法等。
2. 不要重复造轮子，要善于使用已经有的工具类库。经验证明，自己去实现这些类库，出 bug 的概率会更高，维护的成本也比较高。
3. 不要过度优化。不要过度使用一些奇技淫巧（比如，位运算代替算术运算、复杂的条件语句代替 if-else、使用一些过于底层的函数等）来优化代码，牺牲代码的可读性。

YAGNI原则

YAGNI 原则的英文全称是：You Ain’t Gonna Need It。直译就是：你不会需要它。这条原则也算是万金油了。当用在软件开发中的时候，它的意思是：不要去设计当前用不到的功能；不要去编写当前用不到的代码。实际上，这条原则的核心思想就是：不要做过度设计。

再比如，我们不要在项目中提前引入不需要依赖的开发包。对于 Java 程序员来说，我们经常使用 Maven 或者 Gradle 来管理依赖的类库（library）。我发现，有些同事为了避免开发中 library 包缺失而频繁地修改 Maven 或者 Gradle 配置文件，提前往项目里引入大量常用的 library 包。实际上，这样的做法也是违背 YAGNI 原则的。

DRY 原则（Don’t Repeat Yourself）

我们今天讲了三种代码重复的情况：实现逻辑重复、功能语义重复、代码执行重复。实现逻辑重复，但功能语义不重复的代码，并不违反 DRY 原则。实现逻辑不重复，但功能语义重复的代码，也算是违反 DRY 原则。除此之外，代码执行重复也算是违反 DRY 原则。

代码复用性

减少代码耦合

对于高度耦合的代码，当我们希望复用其中的一个功能，想把这个功能的代码抽取出来成为一个独立的模块、类或者函数的时候，往往会发现牵一发而动全身。移动一点代码，就要牵连到很多其他相关的代码。所以，高度耦合的代码会影响到代码的复用性，我们要尽量减少代码耦合。

满足单一职责原则

我们前面讲过，如果职责不够单一，模块、类设计得大而全，那依赖它的代码或者它依赖的代码就会比较多，进而增加了代码的耦合。根据上一点，也就会影响到代码的复用性。相反，越细粒度的代码，代码的通用性会越好，越容易被复用。

模块化

这里的“模块”，不单单指一组类构成的模块，还可以理解为单个类、函数。我们要善于将功能独立的代码，封装成模块。独立的模块就像一块一块的积木，更加容易复用，可以直接拿来搭建更加复杂的系统。

业务与非业务逻辑分离

越是跟业务无关的代码越是容易复用，越是针对特定业务的代码越难复用。所以，为了复用跟业务无关的代码，我们将业务和非业务逻辑代码分离，抽取成一些通用的框架、类库、组件等。

通用代码下沉

从分层的角度来看，越底层的代码越通用、会被越多的模块调用，越应该设计得足够可复用。一般情况下，在代码分层之后，为了避免交叉调用导致调用关系混乱，我们只允许上层代码调用下层代码及同层代码之间的调用，杜绝下层代码调用上层代码。所以，通用的代码我们尽量下沉到更下层。

继承、多态、抽象、封装

在讲面向对象特性的时候，我们讲到，利用继承，可以将公共的代码抽取到父类，子类复用父类的属性和方法。利用多态，我们可以动态地替换一段代码的部分逻辑，让这段代码可复用。除此之外，抽象和封装，从更加广义的层面、而非狭义的面向对象特性的层面来理解的话，越抽象、越不依赖具体的实现，越容易复用。代码封装成模块，隐藏可变的细节、暴露不变的接口，就越容易复用。

应用模板等设计模式

一些设计模式，也能提高代码的复用性。比如，模板模式利用了多态来实现，可以灵活地替换其中的部分代码，整个流程模板代码可复用。关于应用设计模式提高代码复用性这一部分，我们留在后面慢慢来讲解。

迪米特法则/最小知识原则

迪米特法则的英文翻译是：Law of Demeter，缩写是 LOD。也叫作最小知识原则，英文翻译为：The Least Knowledge Principle。

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（也就是定义中的“有限知识”）。

如何理解“高内聚、松耦合”?

“高内聚、松耦合”是一个非常重要的设计思想，能够有效提高代码的可读性和可维护性，缩小功能改动导致的代码改动范围。“高内聚”用来指导类本身的设计，“松耦合”用来指导类与类之间依赖关系的设计。

所谓高内聚，就是指相近的功能应该放到同一个类中，不相近的功能不要放到同一类中。相近的功能往往会被同时修改，放到同一个类中，修改会比较集中。所谓松耦合指的是，在代码中，类与类之间的依赖关系简单清晰。即使两个类有依赖关系，一个类的代码改动也不会或者很少导致依赖类的代码改动。

如何理解“迪米特法则”?

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。迪米特法则是希望减少类之间的耦合，让类越独立越好。每个类都应该少了解系统的其他部分。一旦发生变化，需要了解这一变化的类就会比较少。

为什么要分 MVC 三层开发?

分层能起到代码复用的作用

同一个 Repository 可能会被多个 Service 来调用，同一个 Service 可能会被多个 Controller 调用。比如，UserService 中的 getUserById() 接口封装了通过 ID 获取用户信息的逻辑，这部分逻辑可能会被 UserController 和 AdminController 等多个 Controller 使用。如果没有 Service 层，每个 Controller 都要重复实现这部分逻辑，显然会违反 DRY 原则。

分层能起到隔离变化的作用

分层体现了一种抽象和封装的设计思想。比如，Repository 层封装了对数据库访问的操作，提供了抽象的数据访问接口。基于接口而非实现编程的设计思想，Service 层使用 Repository 层提供的接口，并不关心其底层依赖的是哪种具体的数据库。当我们需要替换数据库的时候，比如从 MySQL 到 Oracle，从 Oracle 到 Redis，只需要改动 Repository 层的代码，Service 层的代码完全不需要修改。

除此之外，Controller、Service、Repository 三层代码的稳定程度不同、引起变化的原因不同，所以分成三层来组织代码，能有效地隔离变化。比如，Repository 层基于数据库表，而数据库表改动的可能性很小，所以 Repository 层的代码最稳定，而 Controller 层提供适配给外部使用的接口，代码经常会变动。分层之后，Controller 层中代码的频繁改动并不会影响到稳定的 Repository 层。

分层能起到隔离关注点的作用

Repository 层只关注数据的读写。Service 层只关注业务逻辑，不关注数据的来源。Controller 层只关注与外界打交道，数据校验、封装、格式转换，并不关心业务逻辑。三层之间的关注点不同，分层之后，职责分明，更加符合单一职责原则，代码的内聚性更好。

分层能提高代码的可测试性

后面讲单元测试的时候，我们会讲到，单元测试不依赖不可控的外部组件，比如数据库。分层之后，Repsitory 层的代码通过依赖注入的方式供 Service 层使用，当要测试包含核心业务逻辑的 Service 层代码的时候，我们可以用 mock 的数据源替代真实的数据库，注入到 Service 层代码中。代码的可测试性和单元测试我们后面会讲到，这里你稍微了解即可。

分层能应对系统的复杂性

所有的代码都放到一个类中，那这个类的代码就会因为需求的迭代而无限膨胀。我们知道，当一个类或一个函数的代码过多之后，可读性、可维护性就会变差。那我们就要想办法拆分。拆分有垂直和水平两个方向。水平方向基于业务来做拆分，就是模块化；垂直方向基于流程来做拆分，就是这里说的分层。

还是那句话，不管是分层、模块化，还是 OOP、DDD，以及各种设计模式、原则和思想，都是为了应对复杂系统，应对系统的复杂性。对于简单系统来说，其实是发挥不了作用的，就是俗话说的“杀鸡焉用牛刀”。

还是那句话，不管是分层、模块化，还是 OOP、DDD，以及各种设计模式、原则和思想，都是为了应对复杂系统，应对系统的复杂性。对于简单系统来说，其实是发挥不了作用的，就是俗话说的“杀鸡焉用牛刀”。

BO、VO、Entity 存在的意义是什么?

在前面的章节中，我们提到，针对 Controller、Service、Repository 三层，每层都会定义相应的数据对象，它们分别是 VO（View Object）、BO（Business Object）、Entity，例如 UserVo、UserBo、UserEntity。在实际的开发中，VO、BO、Entity 可能存在大量的重复字段，甚至三者包含的字段完全一样。在开发的过程中，我们经常需要重复定义三个几乎一样的类，显然是一种重复劳动。

相对于每层定义各自的数据对象来说，是不是定义一个公共的数据对象更好些呢?

实际上，我更加推荐每层都定义各自的数据对象这种设计思路，主要有以下 3 个方面的原因。

- VO、BO、Entity 并非完全一样。比如，我们可以在 UserEntity、UserBo 中定义 Password 字段，但显然不能在 UserVo 中定义 Password 字段，否则就会将用户的密码暴露出去。
- VO、BO、Entity 三个类虽然代码重复，但功能语义不重复，从职责上讲是不一样的。所以，也并不能算违背 DRY 原则。在前面讲到 DRY 原则的时候，针对这种情况，如果合并为同一个类，那也会存在后期因为需求的变化而需要再拆分的问题。
- 为了尽量减少每层之间的耦合，把职责边界划分明确，每层都会维护自己的数据对象，层与层之间通过接口交互。数据从下一层传递到上一层的时候，将下一层的数据对象转化成上一层的数据对象，再继续处理。虽然这样的设计稍微有些繁琐，每层都需要定义各自的数据对象，需要做数据对象之间的转化，但是分层清晰。对于非常大的项目来说，结构清晰是第一位的！

既然 VO、BO、Entity 不能合并，那如何解决代码重复的问题呢？

从设计的角度来说，VO、BO、Entity 的设计思路并不违反 DRY 原则，为了分层清晰、减少耦合，多维护几个类的成本也并不是不能接受的。但是，如果你真的有代码洁癖，对于代码重复的问题，我们也有一些办法来解决。
我们前面讲到，继承可以解决代码重复问题。我们可以将公共的字段定义在父类中，让 VO、BO、Entity 都继承这个父类，各自只定义特有的字段。因为这里的继承层次很浅，也不复杂，所以使用继承并不会影响代码的可读性和可维护性。后期如果因为业务的需要，有些字段需要从父类移动到子类，或者从子类提取到父类，代码改起来也并不复杂。
前面在讲“多用组合，少用继承”设计思想的时候，我们提到，组合也可以解决代码重复的问题，所以，这里我们还可以将公共的字段抽取到公共的类中，VO、BO、Entity 通过组合关系来复用这个类的代码。

代码重复问题解决了，那不同分层之间的数据对象该如何互相转化呢？

当下一层的数据通过接口调用传递到上一层之后，我们需要将它转化成上一层对应的数据对象类型。比如，Service 层从 Repository 层获取的 Entity 之后，将其转化成 BO，再继续业务逻辑的处理。所以，整个开发的过程会涉及“Entity 到 BO”和“BO 到 VO”这两种转化。

VO、BO、Entity 都是基于贫血模型的，而且为了兼容框架或开发库（比如 MyBatis、Dozer、BeanUtils），我们还需要定义每个字段的 set 方法。这些都违背 OOP 的封装特性，会导致数据被随意修改。那到底该怎么办好呢？

前面我们也提到过，Entity 和 VO 的生命周期是有限的，都仅限在本层范围内。而对应的 Repository 层和 Controller 层也都不包含太多业务逻辑，所以也不会有太多代码随意修改数据，即便设计成贫血、定义每个字段的 set 方法，相对来说也是安全的。

不过，Service 层包含比较多的业务逻辑代码，所以 BO 就存在被任意修改的风险了。但是，设计的问题本身就没有最优解，只有权衡。为了使用方便，我们只能做一些妥协，放弃 BO 的封装特性，由程序员自己来负责这些数据对象的不被错误使用。

规范与重构

为什么要重构

首先，重构是时刻保证代码质量的一个极其有效的手段，不至于让代码腐化到无可救药的地步。项目在演进，代码不停地在堆砌。如果没有人为代码的质量负责任，代码总是会往越来越混乱的方向演进。当混乱到一定程度之后，量变引起质变，项目的维护成本已经高过重新开发一套新代码的成本，想要再去重构，已经没有人能做到了。

其次，优秀的代码或架构不是一开始就能完全设计好的，就像优秀的公司和产品也都是迭代出来的。我们无法 100% 遇见未来的需求，也没有足够的精力、时间、资源为遥远的未来买单，所以，随着系统的演进，重构代码也是不可避免的。

重构是避免过度设计的有效手段。在我们维护代码的过程中，真正遇到问题的时候，再对代码进行重构，能有效避免前期投入太多时间做过度的设计，做到有的放矢。

除此之外，重构对一个工程师本身技术的成长也有重要的意义。

从前面我给出的重构的定义来看，重构实际上是对我们学习的经典设计思想、设计原则、设计模式、编程规范的一种应用。重构实际上就是将这些理论知识，应用到实践的一个很好的场景，能够锻炼我们熟练使用这些理论知识的能力。除此之外，平时堆砌业务逻辑，你可能总觉得没啥成长，而将一个比较烂的代码重构成一个比较好的代码，会让你很有成就感。

重构的对象

根据重构的规模，我们可以笼统地分为大规模高层次重构（以下简称为“大型重构”）和小规模低层次的重构（以下简称为“小型重构”）。

大型重构指的是对顶层代码设计的重构，包括：系统、模块、代码结构、类与类之间的关系等的重构，重构的手段有：分层、模块化、解耦、抽象可复用组件等等。这类重构的工具就是我们学习过的那些设计思想、原则和模式。这类重构涉及的代码改动会比较多，影响面会比较大，所以难度也较大，耗时会比较长，引入 bug 的风险也会相对比较大。

小型重构指的是对代码细节的重构，主要是针对类、函数、变量等代码级别的重构，比如规范命名、规范注释、消除超大类或函数、提取重复代码等等。小型重构更多的是利用我们能后面要讲到的编码规范。这类重构要修改的地方比较集中，比较简单，可操作性较强，耗时会比较短，引入 bug 的风险相对来说也会比较小。你只需要熟练掌握各种编码规范，就可以做到得心应手。

什么时候重构

搞清楚了为什么重构，到底重构什么，我们再来看一下，什么时候重构？是代码烂到一定程度之后才去重构吗？当然不是。因为当代码真的烂到出现“开发效率低，招了很多人，天天加班，出活却不多，线上 bug 频发，领导发飙，中层束手无策，工程师抱怨不断，查找 bug 困难”的时候，基本上重构也无法解决问题了。

我个人比较反对，平时不注重代码质量，堆砌烂代码，实在维护不了了就大刀阔斧地重构、甚至重写的行为。有时候项目代码太多了，重构很难做得彻底，最后又搞出来一个“四不像的怪物”，这就更麻烦了！所以，寄希望于在代码烂到一定程度之后，集中重构解决所有问题是不现实的，我们必须探索一条可持续、可演进的方式。

所以，我特别提倡的重构策略是持续重构。这也是我在工作中特别喜欢干的事情。平时没有事情的时候，你可以看看项目中有哪些写得不够好的、可以优化的代码，主动去重构一下。或者，在修改、添加某个功能代码的时候，你也可以顺手把不符合编码规范、不好的设计重构一下。总之，就像把单元测试、Code Review 作为开发的一部分，我们如果能把持续重构也作为开发的一部分，成为一种开发习惯，对项目、对自己都会很有好处。

尽管我们说重构能力很重要，但持续重构意识更重要。我们要正确地看待代码质量和重构这件事情。技术在更新、需求在变化、人员在流动，代码质量总会在下降，代码总会存在不完美，重构就会持续在进行。时刻具有持续重构意识，才能避免开发初期就过度设计，避免代码维护的过程中质量的下降。而那些看到别人代码有点瑕疵就一顿乱骂，或者花尽心思去构思一个完美设计的人，往往都是因为没有树立正确的代码质量观，没有持续重构意识。

如何重构

前面我们讲到，按照重构的规模，重构可以笼统地分为大型重构和小型重构。对于这两种不同规模的重构，我们要区别对待。

对于大型重构来说，因为涉及的模块、代码会比较多，如果项目代码质量又比较差，耦合比较严重，往往会牵一发而动全身，本来觉得一天就能完成的重构，你会发现越改越多、越改越乱，没一两个礼拜都搞不定。而新的业务开发又与重构相冲突，最后只能半途而废，revert 掉所有的改动，很失落地又去堆砌烂代码了。

在进行大型重构的时候，我们要提前做好完善的重构计划，有条不紊地分阶段来进行。每个阶段完成一小部分代码的重构，然后提交、测试、运行，发现没有问题之后，再继续进行下一阶段的重构，保证代码仓库中的代码一直处于可运行、逻辑正确的状态。每个阶段，我们都要控制好重构影响到的代码范围，考虑好如何兼容老的代码逻辑，必要的时候还需要写一些兼容过渡代码。只有这样，我们才能让每一阶段的重构都不至于耗时太长（最好一天就能完成），不至于与新的功能开发相冲突。

大规模高层次的重构一定是有组织、有计划，并且非常谨慎的，需要有经验、熟悉业务的资深同事来主导。而小规模低层次的重构，因为影响范围小，改动耗时短，所以，只要你愿意并且有时间，随时都可以去做。实际上，除了人工去发现低层次的质量问题，我们还可以借助很多成熟的静态代码分析工具（比如 CheckStyle、FindBugs、PMD），来自动发现代码中的问题，然后针对性地进行重构优化。

对于重构这件事情，资深的工程师、项目 leader 要负起责任来，没事就重构一下代码，时刻保证代码质量处在一个良好的状态。否则，一旦出现“破窗效应”，一个人往里堆了一些烂代码，之后就会有更多的人往里堆更烂的代码。毕竟往项目里堆砌烂代码的成本太低了。不过，保持代码质量最好的方法还是打造一种好的技术氛围，以此来驱动大家主动去关注代码质量，持续重构代码。

解耦为什么重要

软件设计与开发最重要的工作之一就是应对复杂性。人处理复杂性的能力是有限的。过于复杂的代码往往在可读性、可维护性上都不友好。那如何来控制代码的复杂性呢？手段有很多，我个人认为，最关键的就是解耦，保证代码松耦合、高内聚。如果说重构是保证代码质量不至于腐化到无可救药地步的有效手段，那么利用解耦的方法对代码重构，就是保证代码不至于复杂到无法控制的有效手段。

代码是否需要“解耦”?

间接的衡量标准有很多，前面我们讲到了一些，比如，看修改代码会不会牵一发而动全身。除此之外，还有一个直接的衡量标准，也是我在阅读源码的时候经常会用到的，那就是把模块与模块之间、类与类之间的依赖关系画出来，根据依赖关系图的复杂性来判断是否需要解耦重构。

如果依赖关系复杂、混乱，那从代码结构上来讲，可读性和可维护性肯定不是太好，那我们就需要考虑是否可以通过解耦的方法，让依赖关系变得清晰、简单。当然，这种判断还是有比较强的主观色彩，但是可以作为一种参考和梳理依赖的手段，配合间接的衡量标准一块来使用。

如何给代码解耦

封装与抽象

封装和抽象作为两个非常通用的设计思想，可以应用在很多设计场景中，比如系统、模块、lib、组件、接口、类等等的设计。封装和抽象可以有效地隐藏实现的复杂性，隔离实现的易变性，给依赖的模块提供稳定且易用的抽象接口。

比如，Unix 系统提供的 open() 文件操作函数，我们用起来非常简单，但是底层实现却非常复杂，涉及权限控制、并发控制、物理存储等等。我们通过将其封装成一个抽象的 open() 函数，能够有效控制代码复杂性的蔓延，将复杂性封装在局部代码中。除此之外，因为 open() 函数基于抽象而非具体的实现来定义，所以我们在改动 open() 函数的底层实现的时候，并不需要改动依赖它的上层代码，也符合我们前面提到的“高内聚、松耦合”代码的评判标准。

中间层

引入中间层能简化模块或类之间的依赖关系。下面这张图是引入中间层前后的依赖关系对比图。在引入数据存储中间层之前，A、B、C 三个模块都要依赖内存一级缓存、Redis 二级缓存、DB 持久化存储三个模块。在引入中间层之后，三个模块只需要依赖数据存储一个模块即可。从图上可以看出，中间层的引入明显地简化了依赖关系，让代码结构更加清晰。

除此之外，我们在进行重构的时候，引入中间层可以起到过渡的作用，能够让开发和重构同步进行，不互相干扰。比如，某个接口设计得有问题，我们需要修改它的定义，同时，所有调用这个接口的代码都要做相应的改动。如果新开发的代码也用到这个接口，那开发就跟重构冲突了。为了让重构能小步快跑，我们可以分下面四个阶段来完成接口的修改。

> 第一阶段：引入一个中间层，包裹老的接口，提供新的接口定义。
> 第二阶段：新开发的代码依赖中间层提供的新接口。
> 第三阶段：将依赖老接口的代码改为调用新接口。
> 第四阶段：确保所有的代码都调用新接口之后，删除掉老的接口。

模块化

模块化是构建复杂系统常用的手段。不仅在软件行业，在建筑、机械制造等行业，这个手段也非常有用。对于一个大型复杂系统来说，没有人能掌控所有的细节。之所以我们能搭建出如此复杂的系统，并且能维护得了，最主要的原因就是将系统划分成各个独立的模块，让不同的人负责不同的模块，这样即便在不了解全部细节的情况下，管理者也能协调各个模块，让整个系统有效运转。

聚焦到软件开发上面，很多大型软件（比如 Windows）之所以能做到几百、上千人有条不紊地协作开发，也归功于模块化做得好。不同的模块之间通过 API 来进行通信，每个模块之间耦合很小，每个小的团队聚焦于一个独立的高内聚模块来开发，最终像搭积木一样将各个模块组装起来，构建成一个超级复杂的系统。

我们再聚焦到代码层面。合理地划分模块能有效地解耦代码，提高代码的可读性和可维护性。所以，我们在开发代码的时候，一定要有模块化意识，将每个模块都当作一个独立的 lib 一样来开发，只提供封装了内部实现细节的接口给其他模块使用，这样可以减少不同模块之间的耦合度。

实际上，从刚刚的讲解中我们也可以发现，模块化的思想无处不在，像 SOA、微服务、lib 库、系统内模块划分，甚至是类、函数的设计，都体现了模块化思想。如果追本溯源，模块化思想更加本质的东西就是分而治之。

其他设计思想和原则

“高内聚、松耦合”是一个非常重要的设计思想，能够有效提高代码的可读性和可维护性，缩小功能改动导致的代码改动范围。实际上，在前面的章节中，我们已经多次提到过这个设计思想。很多设计原则都以实现代码的“高内聚、松耦合”为目的。我们来一块总结回顾一下都有哪些原则。

- 单一职责原则
  我们前面提到，内聚性和耦合性并非独立的。高内聚会让代码更加松耦合，而实现高内聚的重要指导原则就是单一职责原则。模块或者类的职责设计得单一，而不是大而全，那依赖它的类和它依赖的类就会比较少，代码耦合也就相应的降低了。

- 基于接口而非实现编程
  基于接口而非实现编程能通过接口这样一个中间层，隔离变化和具体的实现。这样做的好处是，在有依赖关系的两个模块或类之间，一个模块或者类的改动，不会影响到另一个模块或类。实际上，这就相当于将一种强依赖关系（强耦合）解耦为了弱依赖关系（弱耦合）。
- 依赖注入
  跟基于接口而非实现编程思想类似，依赖注入也是将代码之间的强耦合变为弱耦合。尽管依赖注入无法将本应该有依赖关系的两个类，解耦为没有依赖关系，但可以让耦合关系没那么紧密，容易做到插拔替换。
- 多用组合少用继承
  我们知道，继承是一种强依赖关系，父类与子类高度耦合，且这种耦合关系非常脆弱，牵一发而动全身，父类的每一次改动都会影响所有的子类。相反，组合关系是一种弱依赖关系，这种关系更加灵活，所以，对于继承结构比较复杂的代码，利用组合来替换继承，也是一种解耦的有效手段。
- 迪米特法则
  迪米特法则讲的是，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。从定义上，我们明显可以看出，这条原则的目的就是为了实现代码的松耦合。

除了上面讲到的这些设计思想和原则之外，还有一些设计模式也是为了解耦依赖，比如观察者模式，有关这一部分的内容，我们留在设计模式模块中慢慢讲解。

快速改善代码质量的20条编程规范

关于命名

1. 命名的关键是能准确达意。对于不同作用域的命名，我们可以适当地选择不同的长度。
2. 我们可以借助类的信息来简化属性、函数的命名，利用函数的信息来简化函数参数的命名。
3. 命名要可读、可搜索。不要使用生僻的、不好读的英文单词来命名。命名要符合项目的统一规范，也不要用些反直觉的命名。
4. 接口有两种命名方式：一种是在接口中带前缀“I”；另一种是在接口的实现类中带后缀“Impl”。对于抽象类的命名，也有两种方式，一种是带上前缀“Abstract”，一种是不带前缀。这两种命名方式都可以，关键是要在项目中统一。

关于注释

1. 注释的内容主要包含这样三个方面：做什么、为什么、怎么做。对于一些复杂的类和接口，我们可能还需要写明“如何用”。
2. 类和函数一定要写注释，而且要写得尽可能全面详细。函数内部的注释要相对少一些，一般都是靠好的命名、提炼函数、解释性变量、总结性注释来提高代码可读性。
3. 关于代码风格
4. 函数、类多大才合适？函数的代码行数不要超过一屏幕的大小，比如 50 行。类的大小限制比较难确定。
5. 一行代码多长最合适？最好不要超过 IDE 的显示宽度。当然，也不能太小，否则会导致很多稍微长点的语句被折成两行，也会影响到代码的整洁，不利于阅读。
6. 善用空行分割单元块。对于比较长的函数，为了让逻辑更加清晰，可以使用空行来分割各个代码块。
7. 四格缩进还是两格缩进？我个人比较推荐使用两格缩进，这样可以节省空间，尤其是在代码嵌套层次比较深的情况下。不管是用两格缩进还是四格缩进，一定不要用 tab 键缩进。
8. 大括号是否要另起一行？将大括号放到跟上一条语句同一行，可以节省代码行数。但是将大括号另起新的一行的方式，左右括号可以垂直对齐，哪些代码属于哪一个代码块，更加一目了然。
9. 类中成员怎么排列？在 Google Java 编程规范中，依赖类按照字母序从小到大排列。类中先写成员变量后写函数。成员变量之间或函数之间，先写静态成员变量或函数，后写普通变量或函数，并且按照作用域大小依次排列。

关于编码技巧

1. 将复杂的逻辑提炼拆分成函数和类。
2. 通过拆分成多个函数或将参数封装为对象的方式，来处理参数过多的情况。
3. 函数中不要使用参数来做代码执行逻辑的控制。
4. 函数设计要职责单一。
5. 移除过深的嵌套层次，方法包括：去掉多余的 if 或 else 语句，使用 continue、break、return 关键字提前退出嵌套，调整执行顺序来减少嵌套，将部分嵌套逻辑抽象成函数。
6. 用字面常量取代魔法数。
7. 用解释性变量来解释复杂表达式，以此提高代码可读性。

统一编码规范

- 除了这三节讲到的比较细节的知识点之外，最后，还有一条非常重要的，那就是，项目、团队，甚至公司，一定要制定统一的编码规范，并且通过 Code Review 督促执行，这对提高代码质量有立竿见影的效果。

如何发现代码质量问题

如何发现代码质量问题 · 业务需求checklist

- 代码是否实现了预期的业务需求?
- 逻辑是否正确? 是否处理了各种异常情况?
- 日志打印是否得当? 是否方便debug排查问题?
- 接口是否易用? 是否支持幂等、事务等?
- 代码是否存在并发问题? 是否线程安全?
- 性能是否有优化空间，比如，SQL、算法是否可以优化?
- 是否有安全漏洞? 比如，输入输出校验是否全面?

极客时间

关于重构思考

1. 即便是非常简单的需求，不同水平的人写出来的代码，差别可能会很大。我们要对代码质量有所追求，不能只是凑活能用就好。花点心思写一段高质量的代码，比写 100 段凑活能用的代码，对你的代码能力提高更有帮助。
2. 知其然知其所以然，了解优秀代码设计的演变过程，比学习优秀设计本身更有价值。知道为什么这么做，比单纯地知道怎么做更重要，这样可以避免你过度使用设计模式、思想和原则。
3. 设计思想、原则、模式本身并没有太多“高大上”的东西，都是一些简单的道理，而且知识点也并不多，关键还是锻炼具体代码具体分析的能力，把知识点恰当地用在项目中。
4. 我经常讲，高手之间的竞争都是在细节。大的架构设计、分层、分模块思路实际上都差不多。没有项目是靠一些不为人知的设计来取胜的，即便有，很快也能被学习过去。所以，关键还是看代码细节处理得够不够好。这些细节的差别累积起来，会让代码质量有质的差别。所以，要想提高代码质量，还是要在细节处下功夫。

函数出错应该返回啥?

1. 返回错误码C ： 语言没有异常这样的语法机制，返回错误码便是最常用的出错处理方式。而 Java、Python 等比较新的编程语言中，大部分情况下，我们都用异常来处理函数出错的情况，极少会用到错误码。
2. 返回 NULL 值：在多数编程语言中，我们用 NULL 来表示“不存在”这种语义。对于查找函数来说，数据不存在并非一种异常情况，是一种正常行为，所以返回表示不存在语义的 NULL 值比返回异常更加合理。
3. 返回空对象返回： NULL 值有各种弊端，对此有一个比较经典的应对策略，那就是应用空对象设计模式。当函数返回的数据是字符串类型或者集合类型的时候，我们可以用空字符串或空集合替代 NULL 值，来表示不存在的情况。这样，我们在使用函数的时候，就可以不用做 NULL 值判断。
4. 抛出异常对象：尽管前面讲了很多函数出错的返回数据类型，但是，最常用的函数出错处理方式是抛出异常。异常有两种类型：受检异常和非受检异常。

总之，是否往上继续抛出，要看上层代码是否关心这个异常。关心就将它抛出，否则就直接吞掉。是否需要包装成新的异常抛出，看上层代码是否能理解这个异常、是否业务相关。如果能理解、业务相关就可以直接抛出，否则就封装成新的异常抛出。

代码质量、面向对象、设计模式、规范与重构总结

代码质量评判标准

极客时间

面向对象

二、面向对象
意义：提高代码的扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础
面向对象VS面向过程
更能应对这种复杂类型的程序开发
具有更加丰富的特性（封装、抽象、继承、多态）
更加人性化、更加高级、更加智能
面向对象分析、设计与编程
面向对象分析：搞清楚做什么，产出详细的需求描述
面向对象设计：搞清楚怎么做，将需求描述转化为具体的类
划分职责进而识别出有哪些类
定义类及其属性和方法
定义类与类之间的交互关系
将类组装起来并提供执行入口
面向对象编程：将分析和设计的的结果翻译成代码
接口VS抽象类
如果要表示一种is-a的关系，并且是为了解决代码复用问题，我们就用抽象类
如果要表示 一种has-a关系，并且是为了解决抽象而非代码复用问题，那我们就用接口
基于接口而非实现编程
1.将接口和实现相分离，封装不稳定的实现，暴露稳定的接口
2.即“基于抽象而非实现编程”。抽象是提高代码扩展性、灵活性、可维护性最有效的手段之一
多用组合少用继承
为什么不推荐使用继承?
继承层次过深、过复杂，会影响到代码的可维护性
组合相比继承有哪些优势?
继承的3个作用
表示is-a关系
支持多态特性
代码复用
组合能解决层次过深、过复杂的继承关系影响代码可维护性的问题
如何判断该用组合还是继承?
1.类之间的继承结构稳定，层次比较浅，关系不复杂，我们使用继承。反之，用组合
2.鼓励多用组合少用继承，但组合也并不是完美的，继承也并非一无是处
贫血模型VS充血模型
基于贫血模型的传统开发模式：面向过程
适合业务不复杂的系统开发
基于充血模型的DDD开发模式：面向对象
适合业务复杂的系统开发
两者主要区别在Service层，Controller层和Repository层的代码基本上相同

极客时间

设计模式

SOLID原则：SRP单一职责原则
概念：一个类只负责完成一个职责或者功能
意义
提高类的内聚性
实现代码的高内聚、松耦合
不满足的5种情况
1.类中的代码行数、函数或者属性过多
2.类依赖的其他类过多或者依赖类的其他类过多
3.私有方法过多
4.比较难给类起一个合适的名字
5.类中大量的方法都是集中操作类中的某几个属性
SOLID原则：OCP开闭原则
如何理解?
开闭原则并不是说完全杜绝修改，而是以最小的修改代码的代价来完成新功能的开发
同样的代码改动，在粗代码粒度下，可能被认定为“修改”；在细代码粒度下，可能又被认定为“扩展”
如何做到?
时刻具备扩展意识、抽象意识、封装意识
常用来提高代码扩展性的方法
多态
依赖注入
基于接口而非实现编程
大部分设计模式：装饰、策略、模板、职责链、状态
SOLID原则：LSP里式替换原则
概念：子类对象能够替换程序中父类对象出现的任何地方，并且保证原来程序的逻辑行为不变及正确性不被破坏
核心：“design by contract，按照协议来设计”。父类定义了函数的“约定（或协议）”，子类可以改变函数的内部实现逻辑，但不能改变函数的原有的“约定”
里式替换原则VS多态
多态是面向对象编程的一大特性，是面向对象编程语言的一种语法，是一种代码实现思路

三、设计原则
SOLID原则：ISP接口隔离原则
概念：客户端不应该强迫依赖它不需要的接口。其中的“客户端”，可以理解为接口的调用者或者使用者
核心：“接口”的3种不同的理解
1.把“接口”理解为一组接口集合
2.把“接口”理解为单个API接口或函数
3.把“接口”理解为OOP中的接口
单一职责原则VS接口隔离原则
单一职责原则针对的是模块、类、接口的设计
接口隔离原则提供了一种判断接口的职责是否单一的标准：通过调用者如何使用接口来间接地判定
SOLID原则：DIP依赖倒置原则
控制反转：一个比较笼统的设计思想，并不是一种具体的实现方法
依赖注入：和控制反转恰恰相反，是一种具体的编码技巧
依赖注入框架：通过依赖注入框架提供的扩展点，简单配置一下所有需要的类及其类与类之间依赖关系，就可以实现由框架来自动创建对象、管理对象的生命周期、依赖注入等原本需要程序员来做的事情
依赖反转原则：也叫作依赖倒置原则。跟控制反转有点类似，主要用来指导框架层面的设计
KISS、YAGNI原则
KISS原则
概念：尽量保持简单
意义：保持代码可读和可维护的重要手段
如何写出满足KISS原则的代码？
不要使用同事可能不懂的技术来实现代码
不要重复造轮子，善于使用已经有的工具类库
不要过度优化
YAGNI原则
概念：不要去设计当前用不到的功能，不要去编写当前用不到的代码
核心：不要做过度设计
KISS VS YAGNI
KISS原则：“如何做”、尽量保持简单
YAGNI原则：“要不要做”、当前不需要的就不要做
DRY原则
概念：不要写重复的代码
3种代码重复的情况
实现逻辑重复
功能语义重复
代码执行重复
提高代码复用性的一些手段
减少代码耦合
满足单一职责原则
模块化
业务与非业务逻辑分离
通用代码下沉
继承、多态、抽象、封装
应用模板等设计模式
有复用意识
LOD原则
高内聚、松耦合
高内聚：指导类本身的设计。松耦合：指导类与类之间依赖关系的设计
高内聚：相近的功能应该放到同一个类中，不相近的功能不要放到同一类中
松耦合：即使两个类有依赖关系，一个类的代码改动也不会或者很少导致依赖类的代码改动
迪米特法则
概念：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
意义：减少类之间的耦合，让类越独立越好
极客时间

规范与重构

重构概述
为什么重构？
保持代码质量持续处于一个可控状态，不至于腐化到无可救药地步
重构什么？
大规模高层次的重构
内容：对代码分层、模块化、解耦、梳理类之间的交互关系、抽象复用组件
理论基础：设计思想、原则、模式
小规模低层次的重构
内容：规范命名、注释、修正函数参数过多、消除超大类、提取重复代码等编程细节问题，主要是针对类、函数级别的重构
理论基础：编码规范
什么时候重构？
建立持续重构意识，把重构作为开发必不可少的部分融入到开发中
如何重构？
大规模高层次的重构
难度比较大，需要组织、有计划地进行，分阶段地小步快跑
小规模低层次的重构
影响范围小，改动耗时短，随时随地都可以做
什么是单元测试？
概念：代码层面的测试，用于测试“自己”编写的代码的逻辑的正确性
“单元”：一般是类或函数，而不是模块或者系统
为什么要写单元测试？
能有效地发现代码中的Bug、代码设计上的问题
写单元测试的过程本身就是代码重构的过程
单元测试是对集成测试的有力补充，有利于快速熟悉代码，是TDD可落地执行的折中方案
概念：针对代码设计覆盖各种输入、异常、边界条件的测试用例，并将其翻译成代码的过程

四、规范与重构

- 单元测试
  - 如何编写单元测试？
    - 方法：可以利用一些测试框架来简化测试代码的编写
    - 5个正确的编写认知
      - 1.编写单元测试尽管繁琐，但并不是太耗时
      - 2.可以稍微放低单元测试的质量要求
      - 3.覆盖率作为衡量单元测试好坏的唯一标准是不合理的
      - 4.写单元测试一般不需要了解代码的实现逻辑
      - 5.单元测试框架无法测试多半是代码的可测试性不好
  - 单元测试为何难落地执行？
    - 写单元测试本身比较繁琐，技术挑战不大，很多程序员不愿意去写
    - 国内研发比较偏向快糙猛，容易因为开发进度紧，导致单元测试的执行虎头蛇尾
    - 没有建立对单元测试正确的认识，觉得可有可无，单靠督促很难执行得很好
- 代码的可测试性
  - 什么是代码的可测试性？
    - 针对代码编写单元测试的难易程度
  - 依赖注入是编写可测试性代码的最有效手段
    - 依赖注入可以通过mock的方法将不可控的依赖变得可控
    - 除了mock方式，还可以利用二次封装来解决某些代码行为不可控的情况
  - 常见的5种Anti-Patterns
    - 1.代码中包含未决行为逻辑
    - 2.滥用可变全局变量
    - 3.滥用静态方法
    - 4.使用复杂的继承关系
    - 5.高度耦合的代码
- 大型重构：解耦
  - “解耦”为何如此重要？
    - 保证代码松耦合、高内聚，是控制代码复杂度的有效手段
    - 代码高内聚、松耦合，也就是代码结构清晰、分层、模块化合理、依赖关系简单、模块或类之间的耦合小，代码整体的质量就不会差
  - 代码是否需要“解耦”？
    - 间接的衡量标准
      - 改动一个模块或类的代码受影响的模块或类的是否有很多
      - 改动一个模块或者类的代码，依赖的模块或者类是否需要改动
      - 代码的可测试性是否好
    - 直接的衡量标准
      - 把模块与模块之间及其类与类之间的依赖关系画出来，根据依赖关系图的复杂性来判断
  - 如何给代码“解耦”？
    - 封装与抽象
    - 中间层
    - 模块化
    - 其他设计思想与原则
      - 单一职责原则
      - 基于接口而非实现编程
      - 依赖注入
      - 多用组合少用继承
      - 迪米特法则
    - 设计模式
- 小型重构：编码规范（20条）
  - 命名与注释
    - 1.命名的关键是能准确达意
    - 2.借助类的信息来简化属性、函数的命名，利用函数的信息来简化函数参数的命名
    - 3.命名要可读、可搜索，不要使用生僻的、不好读的英文单词
    - 4.接口的2种命名方式：在接口中带前缀“I”；在接口的实现类中带后缀“Impl”。抽象类的2种命名方式：带上前缀“Abstract”；不带前缀
    - 5.注释的内容：做什么、为什么、怎么做。复杂的类和接口，还要写明“如何用”
    - 6.类和函数一定要写注释，而且要写得尽可能全面详细
  - 代码风格
    - 7.函数的代码行数不要超过一屏幕的大小，比如50行
    - 8.一行代码最好不要超过IDE的显示宽度
    - 9.善用空行分割单元块
    - 10.推荐两格缩进，节省空间。一定不要用tab键缩进
    - 11.将大括号跟上一条语句同一行，可以节省代码行数。另起新的一行，结构清晰
    - 12.在Google Java编程规范中，依赖类按照字母序从小到大排列。类中先写成员变量后写函数。成员变量之间或函数之间，先写静态成员变量或函数，后写普通变量或函数，并且按照作用域大小依次排列
  - 编程技巧
    - 13.将复杂的逻辑提炼拆分成函数和类
    - 14.通过拆分成多个函数或将参数封装为对象的方式，来处理参数过多
    - 15.函数中不要使用参数来做代码执行逻辑的控制
    - 16.函数设计要职责单一
    - 17.移除过深的嵌套层次
    - 18.用字面常量取代魔法数
    - 19.用解释性变量来解释复杂表达式
  - 20.团队或项目要制定统一的编码规范，code review督促执行

设计模式与范式：创建型

单例设计模式（Singleton Design Pattern）

单例的定义

单例设计模式（Singleton Design Pattern）理解起来非常简单。一个类只允许创建一个对象（或者叫实例），那这个类就是一个单例类，这种设计模式就叫作单例设计模式，简称单例模式。

单例的用处

从业务概念上，有些数据在系统中只应该保存一份，就比较适合设计为单例类。比如，系统的配置信息类。除此之外，我们还可以使用单例解决资源访问冲突的问题。

单例存在哪些问题

- 单例对 OOP 特性的支持不友好
- 单例会隐藏类之间的依赖关系
- 单例对代码的扩展性不友好
- 单例对代码的可测试性不友好
- 单例不支持有参数的构造函数

单例有什么替代解决方案?

为了保证全局唯一，除了使用单例，我们还可以用静态方法来实现。不过，静态方法这种实现思路，并不能解决我们之前提到的问题。如果要完全解决这些问题，我们可能要从根上，寻找其他方式来实现全局唯一类了。比如，通过工厂模式、IOC 容器（比如 Spring IOC 容器）来保证，由程序员自己来保证（自己在编写代码的时候自己保证不要创建两个类对象）。

有人把单例当作反模式，主张杜绝在项目中使用。我个人觉得这有点极端。模式没有对错，关键看你怎么用。如果单例类并没有后续扩展的需求，并且不依赖外部系统，那设计成单例类就没有太大问题。对于一些全局的类，我们在其他地方 new 的话，还要在类之间传来传去，不如直接做成单例类，使用起来简洁方便。

工厂模式（Factory Design Pattern）

简单工场（Simple Factory）

实现一

```
public class RuleConfigSource {
  public RuleConfig load(String ruleConfigFilePath) {
    String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
    IRuleConfigParser parser = RuleConfigParserFactory.createParser(ruleConfigFileExtension);
    if (parser == null) {
      throw new InvalidRuleConfigException(
              "Rule config file format is not supported: " + ruleConfigFilePath);
    }

    String configText = "";
    //从ruleConfigFilePath文件中读取配置文本到configText中
```

```
    RuleConfig ruleConfig = parser.parse(configText);
    return ruleConfig;
  }

  private String getFileExtension(String filePath) {
    //...解析文件名获取扩展名，比如rule.json，返回json
    return "json";
  }
}

public class RuleConfigParserFactory {
  public static IRuleConfigParser createParser(String configFormat) {
    IRuleConfigParser parser = null;
    if ("json".equalsIgnoreCase(configFormat)) {
      parser = new JsonRuleConfigParser();
    } else if ("xml".equalsIgnoreCase(configFormat)) {
      parser = new XmlRuleConfigParser();
    } else if ("yaml".equalsIgnoreCase(configFormat)) {
      parser = new YamlRuleConfigParser();
    } else if ("properties".equalsIgnoreCase(configFormat)) {
      parser = new PropertiesRuleConfigParser();
    }
    return parser;
  }
}
```

实现二

```
public class RuleConfigParserFactory {
  private static final Map<String, RuleConfigParser> cachedParsers = new HashMap<>();

  static {
    cachedParsers.put("json", new JsonRuleConfigParser());
    cachedParsers.put("xml", new XmlRuleConfigParser());
    cachedParsers.put("yaml", new YamlRuleConfigParser());
    cachedParsers.put("properties", new PropertiesRuleConfigParser());
  }

  public static IRuleConfigParser createParser(String configFormat) {
    if (configFormat == null || configFormat.isEmpty()) {
      return null;//返回null还是IllegalArgumentException全凭你自己说了算
    }
    IRuleConfigParser parser = cachedParsers.get(configFormat.toLowerCase());
    return parser;
  }
}
```

工厂方法模式

```
public interface IRuleConfigParserFactory {
  IRuleConfigParser createParser();
}

public class JsonRuleConfigParserFactory implements IRuleConfigParserFactory {
  @Override
  public IRuleConfigParser createParser() {
    return new JsonRuleConfigParser();
  }
}

public class XmlRuleConfigParserFactory implements IRuleConfigParserFactory {
  @Override
  public IRuleConfigParser createParser() {
    return new XmlRuleConfigParser();
  }
}

public class YamlRuleConfigParserFactory implements IRuleConfigParserFactory {
  @Override
  public IRuleConfigParser createParser() {
    return new YamlRuleConfigParser();
  }
}

public class PropertiesRuleConfigParserFactory implements IRuleConfigParserFactory {
  @Override
  public IRuleConfigParser createParser() {
    return new PropertiesRuleConfigParser();
  }
}


public class RuleConfigSource {
  public RuleConfig load(String ruleConfigFilePath) {
    String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

    IRuleConfigParserFactory parserFactory =
RuleConfigParserFactoryMap.getParserFactory(ruleConfigFileExtension);
    if (parserFactory == null) {
      throw new InvalidRuleConfigException("Rule config file format is not supported: " +
ruleConfigFilePath);
    }
    IRuleConfigParser parser = parserFactory.createParser();

    String configText = "";
    //从ruleConfigFilePath文件中读取配置文本到configText中
```

```
    RuleConfig ruleConfig = parser.parse(configText);
    return ruleConfig;
  }

  private String getFileExtension(String filePath) {
    //...解析文件名获取扩展名，比如rule.json，返回json
    return "json";
  }
}

//因为工厂类只包含方法，不包含成员变量，完全可以复用，
//不需要每次都创建新的工厂类对象，所以，简单工厂模式的第二种实现思路更加合适。
public class RuleConfigParserFactoryMap { //工厂的工厂
  private static final Map<String, IRuleConfigParserFactory> cachedFactories = new HashMap<>();
那什么时候该用工厂方法模式，而非简单工厂模式呢？       static {
    cachedFactories.put("json", new JsonRuleConfigParserFactory());
    cachedFactories.put("xml", new XmlRuleConfigParserFactory());
    cachedFactories.put("yaml", new YamlRuleConfigParserFactory());
    cachedFactories.put("properties", new PropertiesRuleConfigParserFactory());
  }

  public static IRuleConfigParserFactory getParserFactory(String type) {
    if (type == null || type.isEmpty()) {
      return null;
    }
    IRuleConfigParserFactory parserFactory = cachedFactories.get(type.toLowerCase());
    return parserFactory;
  }
}
```

那什么时候该用工厂方法模式，而非简单工厂模式呢？

我们前面提到，之所以将某个代码块剥离出来，独立为函数或者类，原因是这个代码块的逻辑过于复杂，剥离之后能让代码更加清晰，更加可读、可维护。但是，如果代码块本身并不复杂，就几行代码而已，我们完全没必要将它拆分成单独的函数或者类。

基于这个设计思想，当对象的创建逻辑比较复杂，不只是简单的 new 一下就可以，而是要组合其他类对象，做各种初始化操作的时候，我们推荐使用工厂方法模式，将复杂的创建逻辑拆分到多个工厂类中，让每个工厂类都不至于过于复杂。而使用简单工厂模式，将所有的创建逻辑都放到一个工厂类中，会导致这个工厂类变得很复杂。

抽象工厂（Abstract Factory）

在简单工厂和工厂方法中，类只有一种分类方式。比如，在规则配置解析那个例子中，解析器类只会根据配置文件格式（Json、Xml、Yaml.....）来分类。但是，如果类有两种分类方式，比如，我们既可以按照配置文件格式来分类，也可以按照解析的对象（Rule 规则配置还是 System 系统配置）来分类，那就会对应下面这 8 个 parser 类。

```
针对规则配置的解析器：基于接口IRuleConfigParser
JsonRuleConfigParser
XmlRuleConfigParser
YamlRuleConfigParser
PropertiesRuleConfigParser

针对系统配置的解析器：基于接口ISystemConfigParser
JsonSystemConfigParser
XmlSystemConfigParser
YamlSystemConfigParser
PropertiesSystemConfigParser
```

针对这种特殊的场景，如果还是继续用工厂方法来实现的话，我们要针对每个 parser 都编写一个工厂类，也就是要编写 8 个工厂类。如果我们未来还需要增加针对业务配置的解析器（比如 IBizConfigParser），那就要再对应地增加 4 个工厂类。而我们知道，过多的类也会让系统难维护。这个问题该怎么解决呢？

抽象工厂就是针对这种非常特殊的场景而诞生的。我们可以让一个工厂负责创建多个不同类型的对象（IRuleConfigParser、ISystemConfigParser 等），而不是只创建一种 parser 对象。这样就可以有效地减少工厂类的个数。具体的代码实现如下所示：

```
public interface IConfigParserFactory {
  IRuleConfigParser createRuleParser();
  ISystemConfigParser createSystemParser();
  //此处可以扩展新的parser类型，比如IBizConfigParser
}

public class JsonConfigParserFactory implements IConfigParserFactory {
  @Override
  public IRuleConfigParser createRuleParser() {
    return new JsonRuleConfigParser();
  }

  @Override
  public ISystemConfigParser createSystemParser() {
    return new JsonSystemConfigParser();
  }
}

public class XmlConfigParserFactory implements IConfigParserFactory {
  @Override
  public IRuleConfigParser createRuleParser() {
    return new XmlRuleConfigParser();
  }

  @Override
  public ISystemConfigParser createSystemParser() {
    return new XmlSystemConfigParser();
  }
```

```
}

// 省略YamlConfigParserFactory和PropertiesConfigParserFactory代码
```

工厂模式和 DI 容器有何区别?

实际上，DI 容器底层最基本的设计思路就是基于工厂模式的。DI 容器相当于一个大的工厂类，负责在程序启动的时候，根据配置（要创建哪些类对象，每个类对象的创建需要依赖哪些其他类对象）事先创建好对象。当应用程序需要使用某个类对象的时候，直接从容器中获取即可。正是因为它持有一堆对象，所以这个框架才被称为“容器”。

DI 容器相对于我们上节课讲的工厂模式的例子来说，它处理的是更大的对象创建工程。上节课讲的工厂模式中，一个工厂类只负责某个类对象或者某一组相关类对象（继承自同一抽象类或者接口的子类）的创建，而 DI 容器负责的是整个应用中所有类对象的创建。

除此之外，DI 容器负责的事情要比单纯的工厂模式要多。比如，它还包括配置的解析、对象生命周期的管理。接下来，我们就详细讲讲，一个简单的 DI 容器应该包含哪些核心功能。

DI 容器的核心功能有哪些?

总结一下，一个简单的 DI 容器的核心功能一般有三个：配置解析、对象创建和对象生命周期管理。

构建者模式（Builder pattern）

```
public class ResourcePoolConfig {
  private String name;
  private int maxTotal;
  private int maxIdle;
  private int minIdle;

  private ResourcePoolConfig(Builder builder) {
    this.name = builder.name;
    this.maxTotal = builder.maxTotal;
    this.maxIdle = builder.maxIdle;
    this.minIdle = builder.minIdle;
  }
  //...省略getter方法...

  //我们将Builder类设计成了ResourcePoolConfig的内部类。
  //我们也可以将Builder类设计成独立的非内部类ResourcePoolConfigBuilder。
  public static class Builder {
    private static final int DEFAULT_MAX_TOTAL = 8;
    private static final int DEFAULT_MAX_IDLE = 8;
    private static final int DEFAULT_MIN_IDLE = 0;

    private String name;
    private int maxTotal = DEFAULT_MAX_TOTAL;
    private int maxIdle = DEFAULT_MAX_IDLE;
    private int minIdle = DEFAULT_MIN_IDLE;

    public ResourcePoolConfig build() {
```

```
      // 校验逻辑放到这里来做，包括必填项校验、依赖关系校验、约束条件校验等
      if (StringUtils.isBlank(name)) {
        throw new IllegalArgumentException("...");
      }
      if (maxIdle > maxTotal) {
        throw new IllegalArgumentException("...");
      }
      if (minIdle > maxTotal || minIdle > maxIdle) {
        throw new IllegalArgumentException("...");
      }

      return new ResourcePoolConfig(this);
    }

    public Builder setName(String name) {
      if (StringUtils.isBlank(name)) {
        throw new IllegalArgumentException("...");
      }
      this.name = name;
      return this;
    }

    public Builder setMaxTotal(int maxTotal) {
      if (maxTotal <= 0) {
        throw new IllegalArgumentException("...");
      }
      this.maxTotal = maxTotal;
      return this;
    }

    public Builder setMaxIdle(int maxIdle) {
      if (maxIdle < 0) {
        throw new IllegalArgumentException("...");
      }
      this.maxIdle = maxIdle;
      return this;
    }

    public Builder setMinIdle(int minIdle) {
      if (minIdle < 0) {
        throw new IllegalArgumentException("...");
      }
      this.minIdle = minIdle;
      return this;
    }
  }
}

// 这段代码会抛出IllegalArgumentException，因为minIdle>maxIdle
```

```
ResourcePoolConfig config = new ResourcePoolConfig.Builder()
        .setName("dbconnectionpool")
        .setMaxTotal(16)
        .setMaxIdle(10)
        .setMinIdle(12)
        .build();
```

原型模式(Prototype Pattern)

什么是原型模式?

如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同），在这种情况下，我们可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式，来创建新对象，以达到节省创建时间的目的。这种基于原型来创建对象的方式就叫作原型设计模式，简称原型模式。

原型模式的两种实现方法

原型模式有两种实现方法，深拷贝和浅拷贝。浅拷贝只会复制对象中基本数据类型数据和引用对象的内存地址，不会递归地复制引用对象，以及引用对象的引用对象……而深拷贝得到的是一份完完全全独立的对象。所以，深拷贝比起浅拷贝来说，更加耗时，更加耗内存空间。

如果要拷贝的对象是不可变对象，浅拷贝共享不可变对象是没问题的，但对于可变对象来说，浅拷贝得到的对象和原始对象会共享部分数据，就有可能出现数据被修改的风险，也就变得复杂多了。除非像我们今天实战中举的那个例子，需要从数据库中加载 10 万条数据并构建散列表索引，操作非常耗时，这种情况下比较推荐使用浅拷贝，否则，没有充分的理由，不要为了一点点的性能提升而使用浅拷贝。

设计模式与范式：结构型

代理模式（Proxy Design Pattern）

代理模式（Proxy Design Pattern）的原理和代码实现都不难掌握。它在不改变原始类（或叫被代理类）代码的情况下，通过引入代理类来给原始类附加功能。我们通过一个简单的例子来解释一下这段话。

```
public interface IUserController {
  UserVo login(String telephone, String password);
  UserVo register(String telephone, String password);
}

public class UserController implements IUserController {
  //...省略其他属性和方法...

  @Override
  public UserVo login(String telephone, String password) {
    //...省略login逻辑...
    //...返回UserVo数据...
  }
```

```
  @Override
  public UserVo register(String telephone, String password) {
    //...省略register逻辑...
    //...返回UserVo数据...
  }
}

public class UserControllerProxy implements IUserController {
  private MetricsCollector metricsCollector;
  private UserController userController;

  public UserControllerProxy(UserController userController) {
    this.userController = userController;
    this.metricsCollector = new MetricsCollector();
  }

  @Override
  public UserVo login(String telephone, String password) {
    long startTimestamp = System.currentTimeMillis();

    // 委托
    UserVo userVo = userController.login(telephone, password);

    long endTimeStamp = System.currentTimeMillis();
    long responseTime = endTimeStamp - startTimestamp;
    RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
    metricsCollector.recordRequest(requestInfo);

    return userVo;
  }

  @Override
  public UserVo register(String telephone, String password) {
    long startTimestamp = System.currentTimeMillis();

    UserVo userVo = userController.register(telephone, password);

    long endTimeStamp = System.currentTimeMillis();
    long responseTime = endTimeStamp - startTimestamp;
    RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
    metricsCollector.recordRequest(requestInfo);

    return userVo;
  }
}

//UserControllerProxy使用举例
//因为原始类和代理类实现相同的接口，是基于接口而非实现编程
```

```
//将UserController类对象替换为UserControllerProxy类对象，不需要改动太多代码
IUserController userController = new UserControllerProxy(new UserController());
```

动态代理（Dynamic Proxy）

实际上，Spring AOP 底层的实现原理就是基于动态代理。用户配置好需要给哪些类创建代理，并定义好在执行原始类的业务代码前后执行哪些附加功能。Spring 为这些类创建动态代理对象，并在 JVM 中替代原始类对象。原本在代码中执行的原始类的方法，被换作执行代理类的方法，也就实现了给原始类添加附加功能的目的。

代理模式的应用场景

1. 业务系统的非功能性需求开发

代理模式最常用的一个应用场景就是，在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。我们将这些附加功能与业务功能解耦，放到代理类中统一处理，让程序员只需要关注业务方面的开发。实际上，前面举的搜集接口请求信息的例子，就是这个应用场景的一个典型例子。

如果你熟悉 Java 语言和 Spring 开发框架，这部分工作都是可以在 Spring AOP 切面中完成的。前面我们也提到，Spring AOP 底层的实现原理就是基于动态代理。

1. 代理模式在 RPC、缓存中的应用

实际上，RPC 框架也可以看作一种代理模式，GoF 的《设计模式》一书中把它称作远程代理。通过远程代理，将网络通信、数据编解码等细节隐藏起来。客户端在使用 RPC 服务的时候，就像使用本地函数一样，无需了解跟服务器交互的细节。除此之外，RPC 服务的开发者也只需要开发业务逻辑，就像开发本地使用的函数一样，不需要关注跟客户端的交互细节。

桥接模式（Bridge Design Pattern）

今天，我们再学习另外一种结构型模式：桥接模式。桥接模式的代码实现非常简单，但是理解起来稍微有点难度，并且应用场景也比较局限，所以，相当于代理模式来说，桥接模式在实际的项目中并没有那么常用，你只需要简单了解，见到能认识就可以，并不是我们学习的重点。

桥接模式，也叫作桥梁模式，英文是 Bridge Design Pattern。这个模式可以说是 23 种设计模式中最难理解的模式之一了。我查阅了比较多的书籍和资料之后发现，对于这个模式有两种不同的理解方式。

当然，这其中“最纯正”的理解方式，当属 GoF 的《设计模式》一书中对桥接模式的定义。毕竟，这 23 种经典的设计模式，最初就是由这本书总结出来的。在 GoF 的《设计模式》一书中，桥接模式是这么定义的：“Decouple an abstraction from its implementation so that the two can vary independently。”翻译成中文就是：“将抽象和实现解耦，让它们可以独立变化。”

关于桥接模式，很多书籍、资料中，还有另外一种理解方式：“一个类存在两个（或多个）独立变化的维度，我们通过组合的方式，让这两个（或多个）维度可以独立进行扩展。”通过组合关系来替代继承关系，避免继承层次的指数级爆炸。这种理解方式非常类似于，我们之前讲过的“组合优于继承”设计原则，所以，这里我就不多解释了。我们重点看下 GoF 的理解方式。

```
Class.forName("com.mysql.jdbc.Driver");//加载及注册JDBC驱动程序
String url = "jdbc:mysql://localhost:3306/sample_db?user=root&password=your_password";
Connection con = DriverManager.getConnection(url);
Statement stmt = con.createStatement();
String query = "select * from test";
ResultSet rs=stmt.executeQuery(query);
while(rs.next()) {
  rs.getString(1);
  rs.getInt(2);
}
```

如果我们想要把 MySQL 数据库换成 Oracle 数据库，只需要把第一行代码中的 com.mysql.jdbc.Driver 换成 oracle.jdbc.driver.OracleDriver 就可以了。当然，也有更灵活的实现方式，我们可以把需要加载的 Driver 类写到配置文件中，当程序启动的时候，自动从配置文件中加载，这样在切换数据库的时候，我们都不需要修改代码，只需要修改配置文件就可以了。

装饰器模式 （Decorator Pattern）

装饰器模式主要解决继承关系过于复杂的问题，通过组合来替代继承。它主要的作用是给原始类添加增强功能。这也是判断是否该用装饰器模式的一个重要的依据。除此之外，装饰器模式还有一个特点，那就是可以对原始类嵌套使用多个装饰器。为了满足这个应用场景，在设计的时候，装饰器类需要跟原始类继承相同的抽象类或者接口。

第一个比较特殊的地方是：装饰器类和原始类继承同样的父类，这样我们可以对原始类“嵌套”多个装饰器类。比如，下面这样一段代码，我们对 FileInputStream 嵌套了两个装饰器类：BufferedInputStream 和 DataInputStream，让它既支持缓存读取，又支持按照基本数据类型来读取数据。

```
InputStream in = new FileInputStream("/user/wangzheng/test.txt");
InputStream bin = new BufferedInputStream(in);
DataInputStream din = new DataInputStream(bin);
int data = din.readInt();
```

第二个比较特殊的地方是：装饰器类是对功能的增强，这也是装饰器模式应用场景的一个重要特点。实际上，符合“组合关系”这种代码结构的设计模式有很多，比如之前讲过的代理模式、桥接模式，还有现在的装饰器模式。尽管它们的代码结构很相似，但是每种设计模式的意图是不同的。就拿比较相似的代理模式和装饰器模式来说吧，代理模式中，代理类附加的是跟原始类无关的功能，而在装饰器模式中，装饰器类附加的是跟原始类相关的增强功能。

```
// 代理模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
  void f();
}
public class A impelements IA {
  public void f() { //... }
}
public class AProxy implements IA {
  private IA a;
  public AProxy(IA a) {
    this.a = a;
  }
```

```
  public void f() {
    // 新添加的代理逻辑
    a.f();
    // 新添加的代理逻辑
  }
}

// 装饰器模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
  void f();
}
public class A implements IA {
  public void f() { //... }
}
public class ADecorator implements IA {
  private IA a;
  public ADecorator(IA a) {
    this.a = a;
  }

  public void f() {
    // 功能增强代码
    a.f();
    // 功能增强代码
  }
}
```

Python的装饰器模式

```
import functools

def log_with_param(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print('call %s():' % func.__name__)
            print('args = {}'.format(*args))
            print('log_param = {}'.format(text))
            return func(*args, **kwargs)

        return wrapper

    return decorator

@log_with_param("param")
def test_with_param(p):
    print(test_with_param.__name__)
```

适配器模式 （Adapter Design Pattern）

适配器模式的英文翻译是 Adapter Design Pattern。顾名思义，这个模式就是用来做适配的，它将不兼容的接口转换为可兼容的接口，让原本由于接口不兼容而不能一起工作的类可以一起工作。对于这个模式，有一个经常被拿来解释它的例子，就是 USB 转接头充当适配器，把两种不兼容的接口，通过转接变得可以一起工作。

原理很简单，我们再来看下它的代码实现。适配器模式有两种实现方式：类适配器和对象适配器。其中，类适配器使用继承关系来实现，对象适配器使用组合关系来实现。具体的代码实现如下所示。其中，ITarget 表示要转化成的接口定义。Adaptee 是一组不兼容 ITarget 接口定义的接口，Adaptor 将 Adaptee 转化成一组符合 ITarget 接口定义的接口。

```
// 类适配器：基于继承
public interface ITarget {
  void f1();
  void f2();
  void fc();
}

public class Adaptee {
  public void fa() { //... }
  public void fb() { //... }
  public void fc() { //... }
}

public class Adaptor extends Adaptee implements ITarget {
  public void f1() {
    super.fa();
  }

  public void f2() {
    //...重新实现f2()...
  }

  // 这里fc()不需要实现，直接继承自Adaptee，这是跟对象适配器最大的不同点
}

// 对象适配器：基于组合
public interface ITarget {
  void f1();
  void f2();
  void fc();
}
```

```
public class Adaptee {
  public void fa() { //... }
  public void fb() { //... }
  public void fc() { //... }
}

public class Adaptor implements ITarget {
  private Adaptee adaptee;

  public Adaptor(Adaptee adaptee) {
    this.adaptee = adaptee;
  }

  public void f1() {
    adaptee.fa(); //委托给Adaptee
  }

  public void f2() {
    //...重新实现f2()...
  }

  public void fc() {
    adaptee.fc();
  }
}
```

针对这两种实现方式，在实际的开发中，到底该如何选择使用哪一种呢？判断的标准主要有两个，一个是 Adaptee 接口的个数，另一个是 Adaptee 和 ITarget 的契合程度。

- 如果 Adaptee 接口并不多，那两种实现方式都可以。
- 如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都相同，那我们推荐使用类适配器，因为 Adaptor 复用父类 Adaptee 的接口，比起对象适配器的实现方式，Adaptor 的代码量要少一些。
- 如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都不相同，那我们推荐使用对象适配器，因为组合结构相对于继承更加灵活。

适配器模式应用场景总结

前面我们反复提到，适配器模式的应用场景是“接口不兼容”。那在实际的开发中，什么情况下才会出现接口不兼容呢？我建议你先自己思考一下这个问题，然后再来看我下面的总结 。

1. 封装有缺陷的接口设计

假设我们依赖的外部系统在接口设计方面有缺陷（比如包含大量静态方法），引入之后会影响到我们自身代码的可测试性。为了隔离设计上的缺陷，我们希望对外部系统提供的接口进行二次封装，抽象出更好的接口设计，这个时候就可以使用适配器模式了。

```
public class CD { //这个类来自外部sdk，我们无权修改它的代码
  //...
  public static void staticFunction1() { //... }
```

```
  public void uglyNamingFunction2() { //... }

  public void tooManyParamsFunction3(int paramA, int paramB, ...) { //... }

   public void lowPerformanceFunction4() { //... }
}

// 使用适配器模式进行重构
public class ITarget {
  void function1();
  void function2();
  void fucntion3(ParamsWrapperDefinition paramsWrapper);
  void function4();
  //...
}
// 注意：适配器类的命名不一定非得末尾带Adaptor
public class CDAdaptor extends CD implements ITarget {
  //...
  public void function1() {
     super.staticFunction1();
  }

  public void function2() {
    super.uglyNamingFucntion2();
  }

  public void function3(ParamsWrapperDefinition paramsWrapper) {
     super.tooManyParamsFunction3(paramsWrapper.getParamA(), ...);
  }

  public void function4() {
    //...reimplement it...
  }
}
```

1. 统一多个类的接口设计

某个功能的实现依赖多个外部系统（或者说类）。通过适配器模式，将它们的接口适配为统一的接口定义，然后我们就可以使用多态的特性来复用代码逻辑。具体我还是举个例子来解释一下。

假设我们的系统要对用户输入的文本内容做敏感词过滤，为了提高过滤的召回率，我们引入了多款第三方敏感词过滤系统，依次对用户输入的内容进行过滤，过滤掉尽可能多的敏感词。但是，每个系统提供的过滤接口都是不同的。这就意味着我们没法复用一套逻辑来调用各个系统。这个时候，我们就可以使用适配器模式，将所有系统的接口适配为统一的接口定义，这样我们可以复用调用敏感词过滤的代码。

```
public class ASensitiveWordsFilter { // A敏感词过滤系统提供的接口
  //text是原始文本，函数输出用***替换敏感词之后的文本
  public String filterSexyWords(String text) {
    // ...
```

```
  }

  public String filterPoliticalWords(String text) {
    // ...
  }
}

public class BSensitiveWordsFilter  { // B敏感词过滤系统提供的接口
  public String filter(String text) {
    //...
  }
}

public class CSensitiveWordsFilter { // C敏感词过滤系统提供的接口
  public String filter(String text, String mask) {
    //...
  }
}

// 未使用适配器模式之前的代码：代码的可测试性、扩展性不好
public class RiskManagement {
  private ASensitiveWordsFilter aFilter = new ASensitiveWordsFilter();
  private BSensitiveWordsFilter bFilter = new BSensitiveWordsFilter();
  private CSensitiveWordsFilter cFilter = new CSensitiveWordsFilter();

  public String filterSensitiveWords(String text) {
    String maskedText = aFilter.filterSexyWords(text);
    maskedText = aFilter.filterPoliticalWords(maskedText);
    maskedText = bFilter.filter(maskedText);
    maskedText = cFilter.filter(maskedText, "***");
    return maskedText;
  }
}

// 使用适配器模式进行改造
public interface ISensitiveWordsFilter { // 统一接口定义
  String filter(String text);
}

public class ASensitiveWordsFilterAdaptor implements ISensitiveWordsFilter {
  private ASensitiveWordsFilter aFilter;
  public String filter(String text) {
    String maskedText = aFilter.filterSexyWords(text);
    maskedText = aFilter.filterPoliticalWords(maskedText);
    return maskedText;
  }
}
//...省略BSensitiveWordsFilterAdaptor、CSensitiveWordsFilterAdaptor...
```

```
// 扩展性更好，更加符合开闭原则，如果添加一个新的敏感词过滤系统，
// 这个类完全不需要改动；而且基于接口而非实现编程，代码的可测试性更好。
public class RiskManagement {
  private List<ISensitiveWordsFilter> filters = new ArrayList<>();

  public void addSensitiveWordsFilter(ISensitiveWordsFilter filter) {
    filters.add(filter);
  }

  public String filterSensitiveWords(String text) {
    String maskedText = text;
    for (ISensitiveWordsFilter filter : filters) {
      maskedText = filter.filter(maskedText);
    }
    return maskedText;
  }
}
```

1. 替换依赖的外部系统

当我们把项目中依赖的一个外部系统替换为另一个外部系统的时候，利用适配器模式，可以减少对代码的改动。具体的代码示例如下所示：

```
  // 外部系统A
  public interface IA {
    //...
    void fa();
  }
  public class A implements IA {
    //...
    public void fa() { //... }
  }
  // 在我们的项目中，外部系统A的使用示例
  public class Demo {
    private IA a;
    public Demo(IA a) {
      this.a = a;
    }
    //...
  }
  Demo d = new Demo(new A());

  // 将外部系统A替换成外部系统B
  public class BAdaptor implemnts IA {
    private B b;
    public BAdaptor(B b) {
      this.b= b;
    }
    public void fa() {
```

```
      //...
      b.fb();
    }
  }
  // 借助BAdaptor，Demo的代码中，调用IA接口的地方都无需改动，
  // 只需要将BAdaptor如下注入到Demo即可。
  Demo d = new Demo(new BAdaptor(new B()));
```

1. 兼容老版本接口

在做版本升级的时候，对于一些要废弃的接口，我们不直接将其删除，而是暂时保留，并且标注为 deprecated，并将内部实现逻辑委托为新的接口实现。这样做的好处是，让使用它的项目有个过渡期，而不是强制进行代码修改。这也可以粗略地看作适配器模式的一个应用场景。同样，我还是通过一个例子，来进一步解释一下。

JDK1.0 中包含一个遍历集合容器的类 Enumeration。JDK2.0 对这个类进行了重构，将它改名为 Iterator 类，并且对它的代码实现做了优化。但是考虑到如果将 Enumeration 直接从 JDK2.0 中删除，那使用 JDK1.0 的项目如果切换到 JDK2.0，代码就会编译不通过。为了避免这种情况的发生，我们必须把项目中所有使用到 Enumeration 的地方，都修改为使用 Iterator 才行。

单独一个项目做 Enumeration 到 Iterator 的替换，勉强还能接受。但是，使用 Java 开发的项目太多了，一次 JDK 的升级，导致所有的项目不做代码修改就会编译报错，这显然是不合理的。这就是我们经常所说的不兼容升级。为了做到兼容使用低版本 JDK 的老代码，我们可以暂时保留 Enumeration 类，并将其实现替换为直接调用 Itertor。代码示例如下所示：

```
  public class Collections {
    public static Emueration emumeration(final Collection c) {
      return new Enumeration() {
        Iterator i = c.iterator();

        public boolean hasMoreElments() {
          return i.hashNext();
        }

        public Object nextElement() {
          return i.next():
        }
      }
    }
  }
```

1. 适配不同格式的数据

前面我们讲到，适配器模式主要用于接口的适配，实际上，它还可以用在不同格式的数据之间的适配。比如，把从不同征信系统拉取的不同格式的征信数据，统一为相同的格式，以方便存储和使用。再比如，Java 中的 Arrays.asList() 也可以看作一种数据适配器，将数组类型的数据转化为集合容器类型。

List stooges = Arrays.asList(“Larry”, “Moe”, “Curly”);

代理、桥接、装饰器、适配器 4 种设计模式的区别

代理、桥接、装饰器、适配器，这 4 种模式是比较常用的结构型设计模式。它们的代码结构非常相似。笼统来说，它们都可以称为 Wrapper 模式，也就是通过 Wrapper 类二次封装原始类。

尽管代码结构相似，但这 4 种设计模式的用意完全不同，也就是说要解决的问题、应用场景不同，这也是它们的主要区别。这里我就简单说一下它们之间的区别。

代理模式：代理模式在不改变原始类接口的条件下，为原始类定义一个代理类，主要目的是控制访问，而非加强功能，这是它跟装饰器模式最大的不同。

桥接模式：桥接模式的目的是将接口部分和实现部分分离，从而让它们可以较为容易、也相对独立地加以改变。

装饰器模式：装饰者模式在不改变原始类接口的情况下，对原始类功能进行增强，并且支持多个装饰器的嵌套使用。

适配器模式：适配器模式是一种事后的补救策略。适配器提供跟原始类不同的接口，而代理模式、装饰器模式提供的都是跟原始类相同的接口。

门面模式（Facade Design Pattern）

门面模式，也叫外观模式，英文全称是 Facade Design Pattern。在 GoF 的《设计模式》一书中，门面模式是这样定义的：

> Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

翻译成中文就是：门面模式为子系统提供一组统一的接口，定义一组高层接口让子系统更易用。

我们知道，类、模块、系统之间的“通信”，一般都是通过接口调用来完成的。接口设计的好坏，直接影响到类、模块、系统是否好用。所以，我们要多花点心思在接口设计上。我经常说，完成接口设计，就相当于完成了一半的开发任务。只要接口设计得好，那代码就差不到哪里去。

接口粒度设计得太大，太小都不好。太大会导致接口不可复用，太小会导致接口不易用。在实际的开发中，接口的可复用性和易用性需要“微妙”的权衡。针对这个问题，我的一个基本的处理原则是，尽量保持接口的可复用性，但针对特殊情况，允许提供冗余的门面接口，来提供更易用的接口。

门面模式的应用场景举例

解决易用性问题

门面模式可以用来封装系统的底层实现，隐藏系统的复杂性，提供一组更加简单易用、更高层的接口。比如，Linux 系统调用函数就可以看作一种“门面”。它是 Linux 操作系统暴露给开发者的一组“特殊”的编程接口，它封装了底层更基础的 Linux 内核调用。再比如，Linux 的 Shell 命令，实际上也可以看作一种门面模式的应用。它继续封装系统调用，提供更加友好、简单的命令，让我们可以直接通过执行命令来跟操作系统交互。

我们前面也多次讲过，设计原则、思想、模式很多都是相通的，是同一个道理不同角度的表述。实际上，从隐藏实现复杂性，提供更易用接口这个意图来看，门面模式有点类似之前讲到的迪米特法则（最少知识原则）和接口隔离原则：两个有交互的系统，只暴露有限的必要的接口。除此之外，门面模式还有点类似之前提到封装、抽象的设计思想，提供更抽象的接口，封装底层实现细节。

解决性能问题

关于利用门面模式解决性能问题这一点，刚刚我们已经讲过了。我们通过将多个接口调用替换为一个门面接口调用，减少网络通信成本，提高 App 客户端的响应速度。所以，关于这点，我就不再举例说明了。我们来讨论一下这样一个问题：从代码实现的角度来看，该如何组织门面接口和非门面接口？

如果门面接口不多，我们完全可以将它跟非门面接口放到一块，也不需要特殊标记，当作普通接口来用即可。如果门面接口很多，我们可以在已有的接口之上，再重新抽象出一层，专门放置门面接口，从类、包的命名上跟原来的接口层做区分。如果门面接口特别多，并且很多都是跨多个子系统的，我们可以将门面接口放到一个新的子系统中。

解决分布式事务问题

关于利用门面模式来解决分布式事务问题，我们通过一个例子来解释一下。

在一个金融系统中，有两个业务领域模型，用户和钱包。这两个业务领域模型都对外暴露了一系列接口，比如用户的增删改查接口、钱包的增删改查接口。假设有这样一个业务场景：在用户注册的时候，我们不仅会创建用户（在数据库 User 表中），还会给用户创建一个钱包（在数据库的 Wallet 表中）。

对于这样一个简单的业务需求，我们可以通过依次调用用户的创建接口和钱包的创建接口来完成。但是，用户注册需要支持事务，也就是说，创建用户和钱包的两个操作，要么都成功，要么都失败，不能一个成功、一个失败。

要支持两个接口调用在一个事务中执行，是比较难实现的，这涉及分布式事务问题。虽然我们可以通过引入分布式事务框架或者事后补偿的机制来解决，但代码实现都比较复杂。而最简单的解决方案是，利用数据库事务或者 Spring 框架提供的事务（如果是 Java 语言的话），在一个事务中，执行创建用户和创建钱包这两个 SQL 操作。这就要求两个 SQL 操作要在一个接口中完成，所以，我们可以借鉴门面模式的思想，再设计一个包裹这两个操作的新接口，让新接口在一个事务中执行两个 SQL 操作。

组合模式（Composite Design Pattern）

组合模式跟我们之前讲的面向对象设计中的“组合关系（通过组合来组装两个类）”，完全是两码事。这里讲的“组合模式”，主要是用来处理树形结构数据。这里的“数据”，你可以简单理解为一组对象集合，待会我们会详细讲解。

正因为其应用场景的特殊性，数据必须能表示成树形结构，这也导致了这种模式在实际的项目开发中并不那么常用。但是，一旦数据满足树形结构，应用这种模式就能发挥很大的作用，能让代码变得非常简洁。

在 GoF 的《设计模式》一书中，组合模式是这样定义的：

> Compose objects into tree structure to represent part-whole hierarchies.Composite lets client treat individual objects and compositions of objects uniformly.

翻译成中文就是：将一组对象组织（Compose）成树形结构，以表示一种“部分 - 整体”的层次结构。组合让客户端（在很多设计模式书籍中，“客户端”代指代码的使用者。）可以统一单个对象和组合对象的处理逻辑。

组合模式的设计思路，与其说是一种设计模式，**倒不如说是对业务场景的一种数据结构和算法的抽象**。其中，数据可以表示成树这种数据结构，业务需求可以通过在树上的递归遍历算法来实现。

说白了就是 一个树有很多节点，这个节点可以是多个类型的组合（比如文件系统，节点可以是目录，也可以是文件）。

享元模式（Flyweight Design Pattern）

所谓“享元”，顾名思义就是被共享的单元。享元模式的意图是复用对象，节省内存，前提是享元对象是不可变对象。

具体来讲，当一个系统中存在大量重复对象的时候，如果这些重复的对象是不可变对象，我们就可以利用享元模式将对象设计成享元，在内存中只保留一份实例，供多处代码引用。这样可以减少内存中对象的数量，起到节省内存的目的。实际上，不仅仅相同对象可以设计成享元，对于相似对象，我们也可以将这些对象中相同的部分（字段）提取出来，设计成享元，让这些大量相似对象引用这些享元。

这里我稍微解释一下，定义中的“不可变对象”指的是，一旦通过构造函数初始化完成之后，它的状态（对象的成员变量或者属性）就不会再被修改了。所以，不可变对象不能暴露任何 set() 等修改内部状态的方法。之所以要求享元是不可变对象，那是因为它会被多处代码共享使用，避免一处代码对享元进行了修改，影响到其他使用它的代码。

在上面的代码实现中，我们利用工厂类来缓存 ChessPieceUnit 信息（也就是 id、text、color）。通过工厂类获取到的 ChessPieceUnit 就是享元。所有的 ChessBoard 对象共享这 30 个 ChessPieceUnit 对象（因为象棋中只有 30 个棋子）。在使用享元模式之前，记录 1 万个棋局，我们要创建 30 万（30*1 万）个棋子的 ChessPieceUnit 对象。利用享元模式，我们只需要创建 30 个享元对象供所有棋局共享使用即可，大大节省了内存。

那享元模式的原理讲完了，我们来总结一下它的代码结构。实际上，它的代码实现非常简单，主要是通过工厂模式，在工厂类中，通过一个 Map 来缓存已经创建过的享元对象，来达到复用的目的。

享元模式 vs 单例、缓存、对象池

我们先来看享元模式跟单例的区别

在单例模式中，一个类只能创建一个对象，而在享元模式中，一个类可以创建多个对象，每个对象被多处代码引用共享。实际上，享元模式有点类似于之前讲到的单例的变体：多例。

我们前面也多次提到，区别两种设计模式，不能光看代码实现，而是要看设计意图，也就是要解决的问题。尽管从代码实现上来看，享元模式和多例有很多相似之处，但从设计意图上来看，它们是完全不同的。应用享元模式是为了对象复用，节省内存，而应用多例模式是为了限制对象的个数。

我们再来看享元模式跟缓存的区别

在享元模式的实现中，我们通过工厂类来“缓存”已经创建好的对象。这里的“缓存”实际上是“存储”的意思，跟我们平时所说的“数据库缓存”“CPU 缓存”“MemCache 缓存”是两回事。我们平时所讲的缓存，主要是为了提高访问效率，而非复用。

最后我们来看享元模式跟对象池的区别

对象池、连接池（比如数据库连接池）、线程池等也是为了复用，那它们跟享元模式有什么区别呢?

你可能对连接池、线程池比较熟悉，对对象池比较陌生，所以，这里我简单解释一下对象池。像 C++ 这样的编程语言，内存的管理是由程序员负责的。为了避免频繁地进行对象创建和释放导致内存碎片，我们可以预先申请一片连续的内存空间，也就是这里说的对象池。每次创建对象时，我们从对象池中直接取出一个空闲对象来使用，对象使用完成之后，再放回到对象池中以供后续复用，而非直接释放掉。

虽然对象池、连接池、线程池、享元模式都是为了复用，但是，如果我们再细致地抠一抠“复用”这个字眼的话，对象池、连接池、线程池等池化技术中的“复用”和享元模式中的“复用”实际上是不同的概念。

池化技术中的“复用”可以理解为“重复使用”，主要目的是节省时间（比如从数据库池中取一个连接，不需要重新创建）。在任意时刻，每一个对象、连接、线程，并不会被多处使用，而是被一个使用者独占，当使用完成之后，放回到池中，再由其他使用者重复利用。享元模式中的“复用”可以理解为“共享使用”，在整个生命周期中，都是被所有使用者共享的，主要目的是节省空间。

实际上，享元模式对 JVM 的垃圾回收并不友好。因为享元工厂类一直保存了对享元对象的引用，这就导致享元对象在没有任何代码使用的情况下，也并不会被 JVM 垃圾回收机制自动回收掉。因此，在某些情况下，如果对象的生命周期很短，也不会被密集使用，利用享元模式反倒可能会浪费更多的内存。所以，除非经过线上验证，利用享元模式真的可以大大节省内存，否则，就不要过度使用这个模式，为了一点点内存的节省而引入一个复杂的设计模式，得不偿失啊。

设计模式与范式：行为型

我们常把 23 种经典的设计模式分为三类：创建型、结构型、行为型。前面我们已经学习了创建型和结构型，从今天起，我们开始学习行为型设计模式。我们知道，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“类或对象的组合或组装”问题，那行为型设计模式主要解决的就是“类或对象之间的交互”问题。

前面我们已经学习了很多设计模式，不知道你有没有发现，实际上，设计模式要干的事情就是解耦。创建型模式是将创建和使用代码解耦，结构型模式是将不同功能代码解耦，行为型模式是将不同的行为代码解耦，具体到观察者模式，它是将观察者和被观察者代码解耦。借助设计模式，我们利用更好的代码结构，将一大坨代码拆分成职责更单一的小类，让其满足开闭原则、高内聚松耦合等特性，以此来控制和应对代码的复杂性，提高代码的可扩展性。

观察者模式（Observer Design Pattern）

观察者模式（Observer Design Pattern）也被称为发布订阅模式（Publish-Subscribe Design Pattern）。在 GoF 的《设计模式》一书中，它的定义是这样的：

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

翻译成中文就是：在对象之间定义一个一对多的依赖，当一个对象状态改变的时候，所有依赖的对象都会自动收到通知。

一般情况下，被依赖的对象叫作被观察者（Observable），依赖的对象叫作观察者（Observer）。不过，在实际的项目开发中，这两种对象的称呼是比较灵活的，有各种不同的叫法，比如：Subject-Observer、Publisher-Subscriber、Producer-Consumer、EventEmitter-EventListener、Dispatcher-Listener。不管怎么称呼，只要应用场景符合刚刚给出的定义，都可以看作观察者模式。

```
public interface Subject {
  void registerObserver(Observer observer);
  void removeObserver(Observer observer);
  void notifyObservers(Message message);
}

public interface Observer {
  void update(Message message);
}

public class ConcreteSubject implements Subject {
  private List<Observer> observers = new ArrayList<Observer>();
```

```
  @Override
  public void registerObserver(Observer observer) {
    observers.add(observer);
  }

  @Override
  public void removeObserver(Observer observer) {
    observers.remove(observer);
  }

  @Override
  public void notifyObservers(Message message) {
    for (Observer observer : observers) {
      observer.update(message);
    }
  }

}

public class ConcreteObserverOne implements Observer {
  @Override
  public void update(Message message) {
    //TODO: 获取消息通知，执行自己的逻辑...
    System.out.println("ConcreteObserverOne is notified.");
  }
}

public class ConcreteObserverTwo implements Observer {
  @Override
  public void update(Message message) {
    //TODO: 获取消息通知，执行自己的逻辑...
    System.out.println("ConcreteObserverTwo is notified.");
  }
}

public class Demo {
  public static void main(String[] args) {
    ConcreteSubject subject = new ConcreteSubject();
    subject.registerObserver(new ConcreteObserverOne());
    subject.registerObserver(new ConcreteObserverTwo());
    subject.notifyObservers(new Message());
  }
}
```

观察者模式的应用场景

观察者模式的应用场景非常广泛，小到代码层面的解耦，大到架构层面的系统解耦，再或者一些产品的设计思路，都有这种模式的影子，比如，邮件订阅、RSS Feeds，本质上都是观察者模式。不同的应用场景和需求下，这个模式也有截然不同的实现方式，有同步阻塞的实现方式，也有异步非阻塞的实现方式；有进程内的实现方式，也有跨进程的实现方式。

模板模式

模板代码，流程固定，动态的内容，用抽象方式给业务方自己去实现。

用处：复用、控制翻转IOC

模板模式，全称是模板方法设计模式，英文是 Template Method Design Pattern。在 GoF 的《设计模式》一书中，它是这么定义的：

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

翻译成中文就是：模板方法模式在一个方法中定义一个算法骨架，并将某些步骤推迟到子类中实现。模板方法模式可以让子类在不改变算法整体结构的情况下，重新定义算法中的某些步骤。

这里的“算法”，我们可以理解为广义上的“业务逻辑”，并不特指数据结构和算法中的“算法”。这里的算法骨架就是“模板”，包含算法骨架的方法就是“模板方法”，这也是模板方法模式名字的由来。

模板模式作用一：复用

开篇的时候，我们讲到模板模式有两大作用：复用和扩展。我们先来看它的第一个作用：复用。

模板模式把一个算法中不变的流程抽象到父类的模板方法 templateMethod() 中，将可变的部分 method1()、method2() 留给子类 ContreteClass1 和 ContreteClass2 来实现。所有的子类都可以复用父类中模板方法定义的流程代码。我们通过两个小例子来更直观地体会一下。

1.Java InputStream

Java IO 类库中，有很多类的设计用到了模板模式，比如 InputStream、OutputStream、Reader、Writer。我们拿 InputStream 来举例说明一下。

我把 InputStream 部分相关代码贴在了下面。在代码中，read() 函数是一个模板方法，定义了读取数据的整个流程，并且暴露了一个可以由子类来定制的抽象方法。不过这个方法也被命名为了 read()，只是参数跟模板方法不同。

```
public abstract class InputStream implements Closeable {
  //...省略其他代码...

  public int read(byte b[], int off, int len) throws IOException {
    if (b == null) {
      throw new NullPointerException();
    } else if (off < 0 || len < 0 || len > b.length - off) {
      throw new IndexOutOfBoundsException();
    } else if (len == 0) {
```

```
      return 0;
    }

    int c = read();
    if (c == -1) {
      return -1;
    }
    b[off] = (byte)c;

    int i = 1;
    try {
      for (; i < len ; i++) {
        c = read();
        if (c == -1) {
          break;
        }
        b[off + i] = (byte)c;
      }
    } catch (IOException ee) {
    }
    return i;
  }

  public abstract int read() throws IOException;
}

public class ByteArrayInputStream extends InputStream {
  //...省略其他代码...

  @Override
  public synchronized int read() {
    return (pos < count) ? (buf[pos++] & 0xff) : -1;
  }
}
```

2.Java AbstractList

在 Java AbstractList 类中，addAll() 函数可以看作模板方法，add() 是子类需要重写的方法，尽管没有声明为 abstract 的，但函数实现直接抛出了 UnsupportedOperationException 异常。前提是，如果子类不重写是不能使用的。

```
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    boolean modified = false;
    for (E e : c) {
        add(index++, e);
        modified = true;
    }
    return modified;
}

public void add(int index, E element) {
    throw new UnsupportedOperationException();
}
```

模板模式作用二：扩展

模板模式的第二大作用的是扩展。这里所说的扩展，并不是指代码的扩展性，而是指框架的扩展性，有点类似我们之前讲到的控制反转，你可以结合第 19 节来一块理解。基于这个作用，模板模式常用在框架的开发中，让框架用户可以在不修改框架源码的情况下，定制化框架的功能。我们通过 Junit TestCase、Java Servlet 两个例子来解释一下。

1.Java Servlet

对于 Java Web 项目开发来说，常用的开发框架是 SpringMVC。利用它，我们只需要关注业务代码的编写，底层的原理几乎不会涉及。但是，如果我们抛开这些高级框架来开发 Web 项目，必然会用到 Servlet。实际上，使用比较底层的 Servlet 来开发 Web 项目也不难。我们只需要定义一个继承 HttpServlet 的类，并且重写其中的 doGet() 或 doPost() 方法，来分别处理 get 和 post 请求。

2.JUnit TestCase

跟 Java Servlet 类似，JUnit 框架也通过模板模式提供了一些功能扩展点（setUp()、tearDown() 等），让框架用户可以在这些扩展点上扩展功能。

在使用 JUnit 测试框架来编写单元测试的时候，我们编写的测试类都要继承框架提供的 TestCase 类。在 TestCase 类中，runBare() 函数是模板方法，它定义了执行测试用例的整体流程：先执行 setUp() 做些准备工作，然后执行 runTest() 运行真正的测试代码，最后执行 tearDown() 做扫尾工作。

回调（Callback）

复用和扩展是模板模式的两大作用，实际上，还有另外一个技术概念，也能起到跟模板模式相同的作用，那就是回调（Callback）。今天我们今天就来看一下，回调的原理、实现和应用，以及它跟模板模式的区别和联系。

相对于普通的函数调用来说，回调是一种双向调用关系。A 类事先注册某个函数 F 到 B 类，A 类在调用 B 类的 P 函数的时候，B 类反过来调用 A 类注册给它的 F 函数。这里的 F 函数就是“回调函数”。A 调用 B，B 反过来又调用 A，这种调用机制就叫作“回调”。

A 类如何将回调函数传递给 B 类呢？不同的编程语言，有不同的实现方法。C 语言可以使用函数指针，Java 则需要使用包裹了回调函数的类对象，我们简称为回调对象。这里我用 Java 语言举例说明一下。代码如下所示：

```
public interface ICallback {
  void methodToCallback();
```

```
}

public class BClass {
  public void process(ICallback callback) {
    //...
    callback.methodToCallback();
    //...
  }
}

public class AClass {
  public static void main(String[] args) {
    BClass b = new BClass();
    b.process(new ICallback() { //回调对象
      @Override
      public void methodToCallback() {
        System.out.println("Call back me.");
      }
    });
  }
}
```

应用举例一：JdbcTemplate
应用举例二：setClickListener()
应用举例三：addShutdownHook()

模板模式 VS 回调

从应用场景上来看，同步回调跟模板模式几乎一致。它们都是在一个大的算法骨架中，自由替换其中的某个步骤，起到代码复用和扩展的目的。而异步回调跟模板模式有较大差别，更像是观察者模式。

从代码实现上来看，回调和模板模式完全不同。回调基于组合关系来实现，把一个对象传递给另一个对象，是一种对象之间的关系；模板模式基于继承关系来实现，子类重写父类的抽象方法，是一种类之间的关系。

前面我们也讲到，组合优于继承。实际上，这里也不例外。在代码实现上，回调相对于模板模式会更加灵活，主要体现在下面几点。

- 像 Java 这种只支持单继承的语言，基于模板模式编写的子类，已经继承了一个父类，不再具有继承的能力。
- 回调可以使用匿名类来创建回调对象，可以不用事先定义类；而模板模式针对不同的实现都要定义不同的子类。
- 如果某个类中定义了多个模板方法，每个方法都有对应的抽象方法，那即便我们只用到其中的一个模板方法，子类也必须实现所有的抽象方法。而回调就更加灵活，我们只需要往用到的模板方法中注入回调对象即可。

策略模式（Strategy Design Pattern）

策略模式，英文全称是 Strategy Design Pattern。在 GoF 的《设计模式》一书中，它是这样定义的：

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

翻译成中文就是：定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）。

我们知道，工厂模式是解耦对象的创建和使用，观察者模式是解耦观察者和被观察者。策略模式跟两者类似，也能起到解耦的作用，不过，它解耦的是策略的定义、创建、使用这三部分。接下来，我就详细讲讲一个完整的策略模式应该包含的这三个部分。

策略的定义

策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。因为所有的策略类都实现相同的接口，所以，客户端代码基于接口而非实现编程，可以灵活地替换不同的策略。示例代码如下所示：

```
public interface Strategy {
  void algorithmInterface();
}

public class ConcreteStrategyA implements Strategy {
  @Override
  public void  algorithmInterface() {
    //具体的算法...
  }
}

public class ConcreteStrategyB implements Strategy {
  @Override
  public void  algorithmInterface() {
    //具体的算法...
  }
}
```

策略的创建

因为策略模式会包含一组策略，在使用它们的时候，一般会通过类型（type）来判断创建哪个策略来使用。为了封装创建逻辑，我们需要对客户端代码屏蔽创建细节。我们可以把根据 type 创建策略的逻辑抽离出来，放到工厂类中。示例代码如下所示：

```
public class StrategyFactory {
  private static final Map<String, Strategy> strategies = new HashMap<>();

  static {
    strategies.put("A", new ConcreteStrategyA());
    strategies.put("B", new ConcreteStrategyB());
  }
```

```
  public static Strategy getStrategy(String type) {
    if (type == null || type.isEmpty()) {
      throw new IllegalArgumentException("type should not be empty.");
    }
    return strategies.get(type);
  }
}
```

一般来讲，如果策略类是无状态的，不包含成员变量，只是纯粹的算法实现，这样的策略对象是可以被共享使用的，不需要在每次调用 getStrategy() 的时候，都创建一个新的策略对象。针对这种情况，我们可以使用上面这种工厂类的实现方式，事先创建好每个策略对象，缓存到工厂类中，用的时候直接返回。

相反，如果策略类是有状态的，根据业务场景的需要，我们希望每次从工厂方法中，获得的都是新创建的策略对象，而不是缓存好可共享的策略对象，那我们就需要按照如下方式来实现策略工厂类。

```
public class StrategyFactory {
  public static Strategy getStrategy(String type) {
    if (type == null || type.isEmpty()) {
      throw new IllegalArgumentException("type should not be empty.");
    }

    if (type.equals("A")) {
      return new ConcreteStrategyA();
    } else if (type.equals("B")) {
      return new ConcreteStrategyB();
    }

    return null;
  }
}
```

策略的使用

这里的“运行时动态”指的是，我们事先并不知道会使用哪个策略，而是在程序运行期间，根据配置、用户输入、计算结果等这些不确定因素，动态决定使用哪种策略。接下来，我们通过一个例子来解释一下。

```
// 策略接口：EvictionStrategy
// 策略类：LruEvictionStrategy、FifoEvictionStrategy、LfuEvictionStrategy...
// 策略工厂：EvictionStrategyFactory

public class UserCache {
  private Map<String, User> cacheData = new HashMap<>();
  private EvictionStrategy eviction;

  public UserCache(EvictionStrategy eviction) {
    this.eviction = eviction;
  }
```

```
  //...
}

// 运行时动态确定，根据配置文件的配置决定使用哪种策略
public class Application {
  public static void main(String[] args) throws Exception {
    EvictionStrategy evictionStrategy = null;
    Properties props = new Properties();
    props.load(new FileInputStream("./config.properties"));
    String type = props.getProperty("eviction_type");
    evictionStrategy = EvictionStrategyFactory.getEvictionStrategy(type);
    UserCache userCache = new UserCache(evictionStrategy);
    //...
  }
}

// 非运行时动态确定，在代码中指定使用哪种策略
public class Application {
  public static void main(String[] args) {
    //...
    EvictionStrategy evictionStrategy = new LruEvictionStrategy();
    UserCache userCache = new UserCache(evictionStrategy);
    //...
  }
}
```

职责链模式 （Chain Of Responsibility Design Pattern）

感觉跟框架的中间件有点类似。

职责链模式的英文翻译是 Chain Of Responsibility Design Pattern。在 GoF 的《设计模式》中，它是这么定义的：

> Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

翻译成中文就是：将请求的发送和接收解耦，让多个接收对象都有机会处理这个请求。将这些接收对象串成一条链，并沿着这条链传递这个请求，直到链上的某个接收对象能够处理它为止。

首先，我们来看，职责链模式如何应对代码的复杂性。

将大块代码逻辑拆分成函数，将大类拆分成小类，是应对代码复杂性的常用方法。应用职责链模式，我们把各个敏感词过滤函数继续拆分出来，设计成独立的类，进一步简化了 SensitiveWordFilter 类，让 SensitiveWordFilter 类的代码不会过多，过复杂。

其次，我们再来看，职责链模式如何让代码满足开闭原则，提高代码的扩展性。

当我们要扩展新的过滤算法的时候，比如，我们还需要过滤特殊符号，按照非职责链模式的代码实现方式，我们需要修改 SensitiveWordFilter 的代码，违反开闭原则。不过，这样的修改还算比较集中，也是可以接受的。而职责链模式的实现方式更加优雅，只需要新添加一个 Filter 类，并且通过 addFilter() 函数将它添加到 FilterChain 中即可，其他代码完全不需要修改。

不过，你可能会说，即便使用职责链模式来实现，当添加新的过滤算法的时候，还是要修改客户端代码（ApplicationDemo），这样做也没有完全符合开闭原则。

实际上，细化一下的话，我们可以把上面的代码分成两类：框架代码和客户端代码。其中，ApplicationDemo 属于客户端代码，也就是使用框架的代码。除 ApplicationDemo 之外的代码属于敏感词过滤框架代码。

假设敏感词过滤框架并不是我们开发维护的，而是我们引入的一个第三方框架，我们要扩展一个新的过滤算法，不可能直接去修改框架的源码。这个时候，利用职责链模式就能达到开篇所说的，在不修改框架源码的情况下，基于职责链模式提供的扩展点，来扩展新的功能。换句话说，我们在框架这个代码范围内实现了开闭原则。

除此之外，利用职责链模式相对于不用职责链的实现方式，还有一个好处，那就是配置过滤算法更加灵活，可以只选择使用某几个过滤算法。

状态模式（）

什么是有限状态机?

有限状态机，英文翻译是 Finite State Machine，缩写为 FSM，简称为状态机。状态机有 3 个组成部分：状态（State）、事件（Event）、动作（Action）。其中，事件也称为转移条件（Transition Condition）。事件触发状态的转移及动作的执行。不过，动作不是必须的，也可能只转移状态，不执行任何动作。

状态机实现方式一：分支逻辑法

```
if (currentState.equals(State.SUPER)) {
  this.currentState = State.SMALL;
  this.score -= 100;
  return;
}
```

状态机实现方式二：查表法

实际上，上面这种实现方法有点类似 hard code，对于复杂的状态机来说不适用，而状态机的第二种实现方式查表法，就更加合适了。接下来，我们就一块儿来看下，如何利用查表法来补全骨架代码。

实际上，除了用状态转移图来表示之外，状态机还可以用二维表来表示，如下所示。在这个二维表中，第一维表示当前状态，第二维表示事件，值表示当前状态经过事件之后，转移到的新状态及其执行的动作。

	E1(Got MushRoom)	E2(Got Cape)	E3(Got Fire Flower)	E4(Met Monster)
Small	Super/+100	Cape/+200	Fire/+300	/
Super	/	Cape/+200	Fire/+300	Small/-100
Cape	/	/	/	Small/-200
Fire	/	/	/	Small/-300
备注：表中的斜杠表示不存在这种状态转移。				

极客时间

相对于分支逻辑的实现方式，查表法的代码实现更加清晰，可读性和可维护性更好。当修改状态机时，我们只需要修改 transitionTable 和 actionTable 两个二维数组即可。实际上，如果我们把这两个二维数组存储在配置文件中，当需要修改状态机时，我们甚至可以不修改任何代码，只需要修改配置文件就可以了。

状态机实现方式三：状态模式

在查表法的代码实现中，事件触发的动作只是简单的积分加减，所以，我们用一个 int 类型的二维数组 actionTable 就能表示，二维数组中的值表示积分的加减值。但是，如果要执行的动作并非这么简单，而是一系列复杂的逻辑操作（比如加减积分、写数据库，还有可能发送消息通知等等），我们就没法用如此简单的二维数组来表示了。这也就是说，查表法的实现方式有一定局限性。

虽然分支逻辑的实现方式不存在这个问题，但它又存在前面讲到的其他问题，比如分支判断逻辑较多，导致代码可读性和可维护性不好等。实际上，针对分支逻辑法存在的问题，我们可以使用状态模式来解决。

状态模式通过将事件触发的状态转移和动作执行，拆分到不同的状态类中，来避免分支判断逻辑。我们还是结合代码来理解这句话。

利用状态模式，我们来补全 MarioStateMachine 类，补全后的代码如下所示。

其中，IMario 是状态的接口，定义了所有的事件。SmallMario、SuperMario、CapeMario、FireMario 是 IMario 接口的实现类，分别对应状态机中的 4 个状态。原来所有的状态转移和动作执行的代码逻辑，都集中在 MarioStateMachine 类中，现在，这些代码逻辑被分散到了这 4 个状态类中。

```
public interface IMario {
  State getName();
  void obtainMushRoom(MarioStateMachine stateMachine);
  void obtainCape(MarioStateMachine stateMachine);
  void obtainFireFlower(MarioStateMachine stateMachine);
  void meetMonster(MarioStateMachine stateMachine);
}

public class SmallMario implements IMario {
  private static final SmallMario instance = new SmallMario();
  private SmallMario() {}
  public static SmallMario getInstance() {
    return instance;
```

```
  }

  @Override
  public State getName() {
    return State.SMALL;
  }

  @Override
  public void obtainMushRoom(MarioStateMachine stateMachine) {
    stateMachine.setCurrentState(SuperMario.getInstance());
    stateMachine.setScore(stateMachine.getScore() + 100);
  }

  @Override
  public void obtainCape(MarioStateMachine stateMachine) {
    stateMachine.setCurrentState(CapeMario.getInstance());
    stateMachine.setScore(stateMachine.getScore() + 200);
  }

  @Override
  public void obtainFireFlower(MarioStateMachine stateMachine) {
    stateMachine.setCurrentState(FireMario.getInstance());
    stateMachine.setScore(stateMachine.getScore() + 300);
  }

  @Override
  public void meetMonster(MarioStateMachine stateMachine) {
    // do nothing...
  }
}

// 省略SuperMario、CapeMario、FireMario类...

public class MarioStateMachine {
  private int score;
  private IMario currentState;

  public MarioStateMachine() {
    this.score = 0;
    this.currentState = SmallMario.getInstance();
  }

  public void obtainMushRoom() {
    this.currentState.obtainMushRoom(this);
  }

  public void obtainCape() {
    this.currentState.obtainCape(this);
  }
```

```
  public void obtainFireFlower() {
    this.currentState.obtainFireFlower(this);
  }

  public void meetMonster() {
    this.currentState.meetMonster(this);
  }

  public int getScore() {
    return this.score;
  }

  public State getCurrentState() {
    return this.currentState.getName();
  }

  public void setScore(int score) {
    this.score = score;
  }

  public void setCurrentState(IMario currentState) {
    this.currentState = currentState;
  }
}
```

实际上，像游戏这种比较复杂的状态机，包含的状态比较多，我优先推荐使用查表法，而状态模式会引入非常多的状态类，会导致代码比较难维护。相反，像电商下单、外卖下单这种类型的状态机，它们的状态并不多，状态转移也比较简单，但事件触发执行的动作包含的业务逻辑可能会比较复杂，所以，更加推荐使用状态模式来实现。

迭代器模式（Iterator Design Pattern）

迭代器模式（Iterator Design Pattern），也叫作游标模式（Cursor Design Pattern）。

在开篇中我们讲到，它用来遍历集合对象。这里说的“集合对象”也可以叫“容器”“聚合对象”，实际上就是包含一组对象的对象，比如数组、链表、树、图、跳表。迭代器模式将集合对象的遍历操作从集合类中拆分出来，放到迭代器类中，让两者的职责更加单一。

迭代器是用来遍历容器的，所以，一个完整的迭代器模式一般会涉及容器和容器迭代器两部分内容。为了达到基于接口而非实现编程的目的，容器又包含容器接口、容器实现类，迭代器又包含迭代器接口、迭代器实现类。

```
// 接口定义方式一
public interface Iterator<E> {
  boolean hasNext();
  void next();
  E currentItem();
}
```

迭代时候增删数据怎么办?

在通过迭代器来遍历集合元素的同时，增加或者删除集合中的元素，有可能会导致某个元素被重复遍历或遍历不到。不过，并不是所有情况下都会遍历出错，有的时候也可以正常遍历，所以，这种行为称为结果不可预期行为或者未决行为。实际上，“不可预期”比直接出错更加可怕，有的时候运行正确，有的时候运行错误，一些隐藏很深、很难 debug 的 bug 就是这么产生的。

有两种比较干脆利索的解决方案，来避免出现这种不可预期的运行结果。一种是遍历的时候不允许增删元素，另一种是增删元素之后让遍历报错。第一种解决方案比较难实现，因为很难确定迭代器使用结束的时间点。第二种解决方案更加合理。Java 语言就是采用的这种解决方案。增删元素之后，我们选择 fail-fast 解决方式，让遍历操作直接抛出运行时异常。

像 Java 语言，迭代器类中除了前面提到的几个最基本的方法之外，还定义了一个 remove() 方法，能够在遍历集合的同时，安全地删除集合中的元素。

快照模式迭代器

理解这个问题最关键的是理解“快照”两个字。所谓“快照”，指我们为容器创建迭代器的时候，相当于给容器拍了一张快照（Snapshot）。之后即便我们增删容器中的元素，快照中的元素并不会做相应的改动。而迭代器遍历的对象是快照而非容器，这样就避免了在使用迭代器遍历的过程中，增删容器中的元素，导致的不可预期的结果或者报错。

解决方案一

我们先来看最简单的一种解决办法。在迭代器类中定义一个成员变量 snapshot 来存储快照。每当创建迭代器的时候，都拷贝一份容器中的元素到快照中，后续的遍历操作都基于这个迭代器自己持有的快照来进行。

解决方案二

我们可以在容器中，为每个元素保存两个时间戳，一个是添加时间戳 addTimestamp，一个是删除时间戳 delTimestamp。当元素被加入到集合中的时候，我们将 addTimestamp 设置为当前时间，将 delTimestamp 设置成最大长整型值（Long.MAX_VALUE）。当元素被删除时，我们将 delTimestamp 更新为当前时间，表示已经被删除。

同时，每个迭代器也保存一个迭代器创建时间戳 snapshotTimestamp，也就是迭代器对应的快照的创建时间戳。当使用迭代器来遍历容器的时候，只有满足 addTimestamp

如果元素的 addTimestamp>snapshotTimestamp，说明元素在创建了迭代器之后才加入的，不属于这个迭代器的快照；如果元素的 delTimestamp

访问者模式(Visitor Pattern)

前面我们讲到，大部分设计模式的原理和实现都很简单，不过也有例外，比如今天要讲的访问者模式。它可以算是 23 种经典设计模式中最难理解的几个之一。因为它难理解、难实现，应用它会导致代码的可读性、可维护性变差，所以，访问者模式在实际的软件开发中很少被用到，在没有特别必要的情况下，建议你不要使用访问者模式。

```
public abstract class ResourceFile {
  protected String filePath;
  public ResourceFile(String filePath) {
    this.filePath = filePath;
  }
  abstract public void accept(Visitor vistor);
```

```
}

public class PdfFile extends ResourceFile {
  public PdfFile(String filePath) {
    super(filePath);
  }

  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }

  //...
}
//...PPTFile、WordFile跟PdfFile类似，这里就省略了...

public interface Visitor {
  void visit(PdfFile pdfFile);
  void visit(PPTFile pdfFile);
  void visit(WordFile pdfFile);
}

public class Extractor implements Visitor {
  @Override
  public void visit(PPTFile pptFile) {
    //...
    System.out.println("Extract PPT.");
  }

  @Override
  public void visit(PdfFile pdfFile) {
    //...
    System.out.println("Extract PDF.");
  }

  @Override
  public void visit(WordFile wordFile) {
    //...
    System.out.println("Extract WORD.");
  }
}

public class Compressor implements Visitor {
  @Override
  public void visit(PPTFile pptFile) {
    //...
    System.out.println("Compress PPT.");
  }
```

```
  @Override
  public void visit(PdfFile pdfFile) {
    //...
    System.out.println("Compress PDF.");
  }

  @Override
  public void visit(WordFile wordFile) {
    //...
    System.out.println("Compress WORD.");
  }

}

public class ToolApplication {
  public static void main(String[] args) {
    Extractor extractor = new Extractor();
    List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
    for (ResourceFile resourceFile : resourceFiles) {
      resourceFile.accept(extractor);
    }

    Compressor compressor = new Compressor();
    for(ResourceFile resourceFile : resourceFiles) {
      resourceFile.accept(compressor);
    }
  }

  private static List<ResourceFile> listAllResourceFiles(String resourceDirectory) {
    List<ResourceFile> resourceFiles = new ArrayList<>();
    //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
    resourceFiles.add(new PdfFile("a.pdf"));
    resourceFiles.add(new WordFile("b.word"));
    resourceFiles.add(new PPTFile("c.ppt"));
    return resourceFiles;
  }
}
```

访问者模式允许一个或者多个操作应用到一组对象上，设计意图是解耦操作和对象本身，保持类职责单一、满足开闭原则以及应对代码的复杂性。

对于访问者模式，学习的主要难点在代码实现。而代码实现比较复杂的主要原因是，函数重载在大部分面向对象编程语言中是静态绑定的。也就是说，调用类的哪个重载函数，是在编译期间，由参数的声明类型决定的，而非运行时，根据参数的实际类型决定的。

正是因为代码实现难理解，所以，在项目中应用这种模式，会导致代码的可读性比较差。如果你的同事不了解这种设计模式，可能就会读不懂、维护不了你写的代码。所以，除非不得已，不要使用这种模式。

Single Dispatch 和 Double Dispatch

如何理解“Dispatch”这个单词呢？ 在面向对象编程语言中，我们可以把方法调用理解为一种消息传递，也就是“Dispatch”。一个对象调用另一个对象的方法，就相当于给它发送一条消息。这条消息起码要包含对象名、方法名、方法参数。

所谓 Single Dispatch，指的是执行哪个对象的方法，根据对象的运行时类型来决定；执行对象的哪个方法，根据方法参数的编译时类型来决定。所谓 Double Dispatch，指的是执行哪个对象的方法，根据对象的运行时类型来决定；执行对象的哪个方法，根据方法参数的运行时类型来决定。

具体到编程语言的语法机制，Single Dispatch 和 Double Dispatch 跟多态和函数重载直接相关。当前主流的面向对象编程语言（比如，Java、C++、C#）都只支持 Single Dispatch，不支持 Double Dispatch。

```
public class ParentClass {
  public void f() {
    System.out.println("I am ParentClass's f().");
  }
}

public class ChildClass extends ParentClass {
  public void f() {
    System.out.println("I am ChildClass's f().");
  }
}

public class SingleDispatchClass {
  public void polymorphismFunction(ParentClass p) {
    p.f();
  }

  public void overloadFunction(ParentClass p) {
    System.out.println("I am overloadFunction(ParentClass p).");
  }

  public void overloadFunction(ChildClass c) {
    System.out.println("I am overloadFunction(ChildClass c).");
  }
}

public class DemoMain {
  public static void main(String[] args) {
    SingleDispatchClass demo = new SingleDispatchClass();
    ParentClass p = new ChildClass();
    demo.polymorphismFunction(p);//执行哪个对象的方法，由对象的实际类型决定
    demo.overloadFunction(p);//执行对象的哪个方法，由参数对象的声明类型决定
  }
}

//代码执行结果：
I am ChildClass's f().
```

```
I am overloadFunction(ParentClass p).
```

Java 支持多态特性，代码可以在运行时获得对象的实际类型（也就是前面提到的运行时类型），然后根据实际类型决定调用哪个方法。尽管 Java 支持函数重载，但 Java 设计的函数重载的语法规则是，并不是在运行时，根据传递进函数的参数的实际类型，来决定调用哪个重载函数，而是在编译时，根据传递进函数的参数的声明类型（也就是前面提到的编译时类型），来决定调用哪个重载函数。也就是说，具体执行哪个对象的哪个方法，只跟对象的运行时类型有关，跟参数的运行时类型无关。所以，Java 语言只支持 Single Dispatch。

备忘录模式（Memento Design Pattern）

备忘录模式，也叫快照（Snapshot）模式，英文翻译是 Memento Design Pattern。在 GoF 的《设计模式》一书中，备忘录模式是这么定义的：

> Captures and externalizes an object’s internal state so that it can be restored later, all without violating encapsulation.

翻译成中文就是：在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。

备忘录模式也叫快照模式，具体来说，就是在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。这个模式的定义表达了两部分内容：一部分是，存储副本以便后期恢复；另一部分是，要在不违背封装原则的前提下，进行对象的备份和恢复。

备忘录模式的应用场景也比较明确和有限，主要是用来防丢失、撤销、恢复等。它跟平时我们常说的“备份”很相似。两者的主要区别在于，备忘录模式更侧重于代码的设计和实现，备份更侧重架构设计或产品设计。

对于大对象的备份来说，备份占用的存储空间会比较大，备份和恢复的耗时会比较长。针对这个问题，不同的业务场景有不同的处理方式。比如，只备份必要的恢复信息，结合最新的数据来恢复；再比如，全量备份和增量备份相结合，低频全量备份，高频增量备份，两者结合来做恢复。

命令模式（Command Design Pattern）

命令模式的英文翻译是 Command Design Pattern。在 GoF 的《设计模式》一书中，它是这么定义的：

> The command pattern encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

命令模式将请求（命令）封装为一个对象，这样可以使用不同的请求参数化其他对象（将不同请求依赖注入到其他对象），并且能够支持请求（命令）的排队执行、记录日志、撤销等（附加控制）功能。

落实到编码实现，命令模式用的最核心的实现手段，是将函数封装成对象。我们知道，C 语言支持函数指针，我们可以把函数当作变量传递来传递去。但是，在大部分编程语言中，函数没法儿作为参数传递给其他函数，也没法儿赋值给变量。借助命令模式，我们可以将函数封装成对象。具体来说就是，设计一个包含这个函数的类，实例化一个对象传来传去，这样就可以实现把函数像对象一样使用。从实现的角度来说，它类似我们之前讲过的回调。

当我们把函数封装成对象之后，对象就可以存储下来，方便控制执行。所以，命令模式的主要作用和应用场景，是用来控制命令的执行，比如，异步、延迟、排队执行命令、撤销重做命令、存储命令、给命令记录日志等等，这才是命令模式能发挥独一无二作用的地方。

命令模式 VS 策略模式

实际上，每个设计模式都应该由两部分组成：第一部分是应用场景，即这个模式可以解决哪类问题；第二部分是解决方案，即这个模式的设计思路和具体的代码实现。不过，代码实现并不是模式必须包含的。如果你单纯地只关注解决方案这一部分，甚至只关注代码实现，就会产生大部分模式看起来都很相似的错觉。

之前讲策略模式的时候，我们有讲到，策略模式包含策略的定义、创建和使用三部分，从代码结构上来，它非常像工厂模式。它们的区别在于，策略模式侧重“策略”或“算法”这个特定的应用场景，用来解决根据运行时状态从一组策略中选择不同策略的问题，而工厂模式侧重封装对象的创建过程，这里的对象没有任何业务场景的限定，可以是策略，但也可以是其他东西。从设计意图上来，这两个模式完全是两回事儿。

在策略模式中，不同的策略具有相同的目的、不同的实现、互相之间可以替换。比如，BubbleSort、SelectionSort 都是为了实现排序的，只不过一个是用冒泡排序算法来实现的，另一个是用选择排序算法来实现的。而在命令模式中，不同的命令具有不同的目的，对应不同的处理逻辑，并且互相之间不可替换。

解释器模式（Interpreter Design Pattern）

解释器模式的英文翻译是 Interpreter Design Pattern。在 GoF 的《设计模式》一书中，它是这样定义的：

> Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

翻译成中文就是：解释器模式为某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法。

解释器模式为某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法。实际上，这里的“语言”不仅仅指我们平时说的中、英、日、法等各种语言。从广义上来讲，只要是能承载信息的载体，我们都可以称之为“语言”，比如，古代的结绳记事、盲文、哑语、摩斯密码等。

解释器模式的代码实现比较灵活，没有固定的模板。我们前面说过，应用设计模式主要是应对代码的复杂性，解释器模式也不例外。它的代码实现的核心思想，就是将语法解析的工作拆分到各个小类中，以此来避免大而全的解析类。一般的做法是，将语法规则拆分一些小的独立的单元，然后对每个单元进行解析，最终合并为对整个语法规则的解析。

解释器模式实战举例

接下来，我们再来看一个更加接近实战的例子，也就是咱们今天标题中的问题：如何实现一个自定义接口告警规则功能?

在我们平时的项目开发中，监控系统非常重要，它可以时刻监控业务系统的运行情况，及时将异常报告给开发者。比如，如果每分钟接口出错数超过 100，监控系统就通过短信、微信、邮件等方式发送告警给开发者。

一般来讲，监控系统支持开发者自定义告警规则，比如我们可以用下面这样一个表达式，来表示一个告警规则，它表达的意思是：每分钟 API 总出错数超过 100 或者每分钟 API 总调用数超过 10000 就触发告警。

```
public interface Expression {
  boolean interpret(Map<String, Long> stats);
}

public class GreaterExpression implements Expression {
  private String key;
  private long value;
```

```
  public GreaterExpression(String strExpression) {
    String[] elements = strExpression.trim().split("\\s+");
    if (elements.length != 3 || !elements[1].trim().equals(">")) {
      throw new RuntimeException("Expression is invalid: " + strExpression);
    }
    this.key = elements[0].trim();
    this.value = Long.parseLong(elements[2].trim());
  }

  public GreaterExpression(String key, long value) {
    this.key = key;
    this.value = value;
  }

  @Override
  public boolean interpret(Map<String, Long> stats) {
    if (!stats.containsKey(key)) {
      return false;
    }
    long statValue = stats.get(key);
    return statValue > value;
  }
}

// LessExpression/EqualExpression跟GreaterExpression代码类似，这里就省略了

public class AndExpression implements Expression {
  private List<Expression> expressions = new ArrayList<>();

  public AndExpression(String strAndExpression) {
    String[] strExpressions = strAndExpression.split("&&");
    for (String strExpr : strExpressions) {
      if (strExpr.contains(">")) {
        expressions.add(new GreaterExpression(strExpr));
      } else if (strExpr.contains("<")) {
        expressions.add(new LessExpression(strExpr));
      } else if (strExpr.contains("==")) {
        expressions.add(new EqualExpression(strExpr));
      } else {
        throw new RuntimeException("Expression is invalid: " + strAndExpression);
      }
    }
  }

  public AndExpression(List<Expression> expressions) {
    this.expressions.addAll(expressions);
  }
```

```
  @Override
  public boolean interpret(Map<String, Long> stats) {
    for (Expression expr : expressions) {
      if (!expr.interpret(stats)) {
        return false;
      }
    }
    return true;
  }

}

public class OrExpression implements Expression {
  private List<Expression> expressions = new ArrayList<>();

  public OrExpression(String strOrExpression) {
    String[] andExpressions = strOrExpression.split("\\|\\|");
    for (String andExpr : andExpressions) {
      expressions.add(new AndExpression(andExpr));
    }
  }

  public OrExpression(List<Expression> expressions) {
    this.expressions.addAll(expressions);
  }

  @Override
  public boolean interpret(Map<String, Long> stats) {
    for (Expression expr : expressions) {
      if (expr.interpret(stats)) {
        return true;
      }
    }
    return false;
  }
}

public class AlertRuleInterpreter {
  private Expression expression;

  public AlertRuleInterpreter(String ruleExpression) {
    this.expression = new OrExpression(ruleExpression);
  }

  public boolean interpret(Map<String, Long> stats) {
    return expression.interpret(stats);
  }
}
```

中介模式(Mediator Design Pattern)

中介模式。跟前面刚刚讲过的命令模式、解释器模式类似，中介模式也属于不怎么常用的模式，应用场景比较特殊、有限，但是，跟它俩不同的是，中介模式理解起来并不难，代码实现也非常简单，学习难度要小很多。

中介模式的英文翻译是 Mediator Design Pattern。在 GoF 中的《设计模式》一书中，它是这样定义的：

> Mediator pattern defines a separate (mediator) object that encapsulates the interaction between a set of objects and the objects delegate their interaction to a mediator object instead of interacting with each other directly.

翻译成中文就是：中介模式定义了一个单独的（中介）对象，来封装一组对象之间的交互。将这组对象之间的交互委派给与中介对象交互，来避免对象之间的直接交互。

实际上，中介模式的设计思想跟中间层很像，通过引入中介这个中间层，将一组对象之间的交互关系（或者说依赖关系）从多对多（网状关系）转换为一对多（星状关系）。原来一个对象要跟 n 个对象交互，现在只需要跟一个中介对象交互，从而最小化对象之间的交互关系，降低了代码的复杂度，提高了代码的可读性和可维护性。

这里我画了一张对象交互关系的对比图。其中，右边的交互图是利用中介模式对左边交互关系优化之后的结果，从图中我们可以很直观地看出，右边的交互关系更加清晰、简洁。

提到中介模式，有一个比较经典的例子不得不说，那就是航空管制。

为了让飞机在飞行的时候互不干扰，每架飞机都需要知道其他飞机每时每刻的位置，这就需要时刻跟其他飞机通信。飞机通信形成的通信网络就会无比复杂。这个时候，我们通过引入“塔台”这样一个中介，让每架飞机只跟塔台来通信，发送自己的位置给塔台，由塔台来负责每架飞机的航线调度。这样就大大简化了通信网络。

从代码中我们可以看出，原本业务逻辑会分散在各个控件中，现在都集中到了中介类中。实际上，这样做既有好处，也有坏处。好处是简化了控件之间的交互，坏处是中介类有可能会变成大而复杂的“上帝类”（God Class）。所以，在使用中介模式的时候，我们要根据实际的情况，平衡对象之间交互的复杂度和中介类本身的复杂度。

中介模式 VS 观察者模式

前面讲观察者模式的时候，我们讲到，观察者模式有多种实现方式。虽然经典的实现方式没法彻底解耦观察者和被观察者，观察者需要注册到被观察者中，被观察者状态更新需要调用观察者的 update() 方法。但是，在跨进程的实现方式中，我们可以利用消息队列实现彻底解耦，观察者和被观察者都只需要跟消息队列交互，观察者完全不知道被观察者的存在，被观察者也完全不知道观察者的存在。

我们前面提到，中介模式也是为了解耦对象之间的交互，所有的参与者都只与中介进行交互。而观察者模式中的消息队列，就有点类似中介模式中的“中介”，观察者模式的中观察者和被观察者，就有点类似中介模式中的“参与者”。那问题来了：中介模式和观察者模式的区别在哪里呢？什么时候选择使用中介模式？什么时候选择使用观察者模式呢？

在观察者模式中，尽管一个参与者既可以是观察者，同时也可以是被观察者，但是，大部分情况下，交互关系往往都是单向的，一个参与者要么是观察者，要么是被观察者，不会兼具两种身份。也就是说，在观察者模式的应用场景中，参与者之间的交互关系比较有条理。

而中介模式正好相反。只有当参与者之间的交互关系错综复杂，维护成本很高的时候，我们才考虑使用中介模式。毕竟，中介模式的应用会带来一定的副作用，前面也讲到，它有可能会产生大而复杂的上帝类。除此之外，如果一个参与者状态的改变，其他参与者执行的操作有一定先后顺序的要求，这个时候，中介模式就可以利用中介类，通过先后调用不同参与者的方法，来实现顺序的控制，而观察者模式是无法实现这样的顺序要求的。

Unix 设计模式应用

封装与抽象

在 Unix、Linux 系统中，有一句经典的话，“Everything is a file”，翻译成中文就是“一切皆文件”。这句话的意思就是，在 Unix、Linux 系统中，很多东西都被抽象成“文件”这样一个概念，比如 Socket、驱动、硬盘、系统信息等。它们使用文件系统的路径作为统一的命名空间（namespace），使用统一的 read、write 标准函数来访问。

比如，我们要查看 CPU 的信息，在 Linux 系统中，我们只需要使用 Vim、Gedit 等编辑器或者 cat 命令，像打开其他文件一样，打开 /proc/cpuinfo，就能查看到相应的信息。除此之外，我们还可以通过查看 /proc/uptime 文件，了解系统运行了多久，查看 /proc/version 了解系统的内核版本等。

实际上，“一切皆文件”就体现了封装和抽象的设计思想。

封装了不同类型设备的访问细节，抽象为统一的文件访问方式，更高层的代码就能基于统一的访问方式，来访问底层不同类型的设备。这样做的好处是，隔离底层设备访问的复杂性。统一的访问方式能够简化上层代码的编写，并且代码更容易复用。

除此之外，抽象和封装还能有效控制代码复杂性的蔓延，将复杂性封装在局部代码中，隔离实现的易变性，提供简单、统一的访问接口，让其他模块来使用，其他模块基于抽象的接口而非具体的实现编程，代码会更加稳定。

分层与模块化

前面我们也提到，模块化是构建复杂系统的常用手段。

对于像 Unix 这样的复杂系统，没有人能掌控所有的细节。之所以我们能开发出如此复杂的系统，并且能维护得了，最主要的原因就是将系统划分成各个独立的模块，比如进程调度、进程通信、内存管理、虚拟文件系统、网络接口等模块。不同的模块之间通过接口来进行通信，模块之间耦合很小，每个小的团队聚焦于一个独立的高内聚模块来开发，最终像搭积木一样，将各个模块组装起来，构建成一个超级复杂的系统。

除此之外，Unix、Linux 等大型系统之所以能做到几百、上千人有条不紊地协作开发，也归功于模块化做得好。不同的团队负责不同的模块开发，这样即便在不了解全部细节的情况下，管理者也能协调各个模块，让整个系统有效运转。

我们常说，计算机领域的任何问题都可以通过增加一个间接的中间层来解决，这本身就体现了分层的重要性。比如，Unix 系统也是基于分层开发的，它可以大致上分为三层，分别是内核、系统调用、应用层。每一层都对上层封装实现细节，暴露抽象的接口来调用。而且，任意一层都可以被重新实现，不会影响到其他层的代码。

面对复杂系统的开发，我们要善于应用分层技术，把容易复用、跟具体业务关系不大的代码，尽量下沉到下层，把容易变动、跟具体业务强相关的代码，尽量上移到上层。

基于接口通信

刚刚我们讲了分层、模块化，那不同的层之间、不同的模块之间，是如何通信的呢？一般来讲都是通过接口调用。在设计模块（module）或者层（layer）要暴露的接口的时候，我们要学会隐藏实现，接口从命名到定义都要抽象一些，尽量少涉及具体的实现细节。

比如，Unix 系统提供的 open() 文件操作函数，底层实现非常复杂，涉及权限控制、并发控制、物理存储，但我们用起来却非常简单。除此之外，因为 open() 函数基于抽象而非具体的实现来定义，所以我们在改动 open() 函数的底层实现的时候，并不需要改动依赖它的上层代码。

高内聚、松耦合

高内聚、松耦合是一个比较通用的设计思想，内聚性好、耦合少的代码，能让我们在修改或者阅读代码的时候，聚集到在一个小范围的模块或者类中，不需要了解太多其他模块或类的代码，让我们的焦点不至于太发散，也就降低了阅读和修改代码的难度。而且，因为依赖关系简单，耦合小，修改代码不会牵一发而动全身，代码改动比较集中，引入 bug 的风险也就减少了很多。

实际上，刚刚讲到的很多方法，比如封装、抽象、分层、模块化、基于接口通信，都能有效地实现代码的高内聚、松耦合。反过来，代码的高内聚、松耦合，也就意味着，抽象、封装做到比较到位、代码结构清晰、分层和模块化合理、依赖关系简单，那代码整体的质量就不会太差。即便某个具体的类或者模块设计得不怎么合理，代码质量不怎么高，影响的范围也是非常有限的。我们可以聚焦于这个模块或者类做相应的小型重构。而相对于代码结构的调整，这种改动范围比较集中的小型重构的难度就小多了。

为扩展而设计

越是复杂项目，越要在前期设计上多花点时间。提前思考项目中未来可能会有哪些功能需要扩展，提前预留好扩展点，以便在未来需求变更的时候，在不改动代码整体结构的情况下，轻松地添加新功能。

做到代码可扩展，需要代码满足开闭原则。特别是像 Unix 这样的开源项目，有 n 多人参与开发，任何人都可以提交代码到代码库中。代码满足开闭原则，基于扩展而非修改来添加新功能，最小化、集中化代码改动，避免新代码影响到老代码，降低引入 bug 的风险。

除了满足开闭原则，做到代码可扩展，我们前面也提到很多方法，比如封装和抽象，基于接口编程等。识别出代码可变部分和不可变部分，将可变部分封装起来，隔离变化，提供抽象化的不可变接口，供上层系统使用。当具体的实现发生变化的时候，我们只需要基于相同的抽象接口，扩展一个新的实现，替换掉老的实现即可，上游系统的代码几乎不需要修改。

KISS 首要原则

简单清晰、可读性好，是任何大型软件开发要遵循的首要原则。只要可读性好，即便扩展性不好，顶多就是多花点时间、多改动几行代码的事情。但是，如果可读性不好，连看都看不懂，那就不是多花时间可以解决得了的了。如果你对现有代码的逻辑似懂非懂，抱着尝试的心态去修改代码，引入 bug 的可能性就会很大。

不管是自己还是团队，在参与大型项目开发的时候，要尽量避免过度设计、过早优化，在扩展性和可读性有冲突的时候，或者在两者之间权衡，模棱两可的时候，应该选择遵循 KISS 原则，首选可读性。

最小惊奇原则

《Unix 编程艺术》一书中提到一个 Unix 的经典设计原则，叫“最小惊奇原则”，英文是“The Least Surprise Principle”。实际上，这个原则等同于“遵守开发规范”，意思是，在做设计或者编码的时候要遵守统一的开发规范，避免反直觉的设计。实际上，关于这一点，我们在前面的编码规范部分也讲到过。

遵从统一的编码规范，所有的代码都像一个人写出来的，能有效地减少阅读干扰。在大型软件开发中，参与开发的人员很多，如果每个人都按照自己的编码习惯来写代码，那整个项目的代码风格就会千奇百怪，这个类是这种编码风格，另一个类又是另外一种风格。在阅读的时候，我们要不停地切换去适应不同的编码风格，可读性就变差了。所以，对于大型项目的开发来说，我们要特别重视遵守统一的开发规范。

如何提高代码质量

吹毛求疵般地执行编码规范

严格执行代码规范，可以使一个项目乃至整个公司的代码具有完全统一的风格，就像同一个人编写的。而且，命名良好的变量、函数、类和注释，也可以提高代码的可读性。编码规范不难掌握，关键是要严格执行。在 Code Review 时，我们一定要严格要求，看到不符合规范的代码，一定要指出并要求修改。

但是，据我了解，实际情况往往事与愿违。虽然大家都知道优秀的代码规范是怎样的，但在具体写代码的过程中，执行得却不好。我觉得，这种情况产生的主要原因还是不够重视。很多人会觉得，一个变量或者函数命名成啥样，关系并不大。所以命名时不推敲，注释也不写，Code Review 的时候也都一副事不关己的心态，觉得没必要太抠细节。日积月累，项目代码就会变得越来越差。所以我这里还是要强调一下，细节决定成败，代码规范的严格执行极为关键。

编写高质量的单元测试

单元测试是最容易执行且对提高代码质量见效最快的方法之一。高质量的单元测试不仅仅要求测试覆盖率要高，还要求测试的全面性，除了测试正常逻辑的执行之外，还要重点、全面地测试异常下的执行情况。毕竟代码出问题的地方大部分都发生在异常、边界条件下。

对于大型复杂项目，集成测试、黑盒测试都很难测试全面，因为组合爆炸，穷举所有测试用例的成本很高，几乎是不可能的。单元测试就是很好的补充。它可以在类、函数这些细粒度的代码层面，保证代码运行无误。底层细粒度的代码 bug 少了，组合起来构建而成的整个系统的 bug 也就相应的减少了。

不流于形式的 Code Review

如果说很多工程师对单元测试不怎么重视，那对 Code Review 就是不怎么接受。我之前跟一些同行聊起 Code Review 的时候，很多人的反应是，这玩意儿不可能很好地执行，形式大于效果，纯粹是浪费时间。是的，即便 Code Review 做得再流畅，也是要花时间的。所以，在业务开发任务繁重的时候，Code Review 往往会流于形式、虎头蛇尾，效果确实不怎么好。

但我们并不能因此就否定 Code Review 本身的价值。在 Google、Facebook 等外企中，Code Review 应用得非常成功，已经成为了开发流程中不可或缺的一部分。所以，要想真正发挥 Code Review 的作用，关键还是要执行到位，不能流于形式。

开发未动、文档先行

对大部分工程师来说，编写技术文档是件挺让人“反感”的事情。一般来讲，在开发某个系统或者重要模块或者功能之前，我们应该先写技术文档，然后，发送给同组或者相关同事审查，在审查没有问题的情况下再开发。这样能够保证事先达成共识，开发出来的东西不至于走样。而且，当开发完成之后，进行 Code Review 的时候，代码审查者通过阅读开发文档，也可以快速理解代码。

除此之外，对于团队和公司来讲，文档是重要的财富。对新人熟悉代码或任务的交接等，技术文档很有帮助。而且，作为一个规范化的技术团队，技术文档是一种摒弃作坊式开发和个人英雄主义的有效方法，是保证团队有效协作的途径。

持续重构、重构、重构

我个人比较反对平时不注重代码质量，堆砌烂代码，实在维护不了了就大刀阔斧地重构甚至重写。有的时候，因为项目代码太多，重构很难做到彻底，最后又搞出来一个四不像的怪物，这就更麻烦了！

优秀的代码或架构不是一开始就能设计好的，就像优秀的公司或产品也都是迭代出来的。我们无法 100% 预见未来的需求，也没有足够的精力、时间、资源为遥远的未来买单。所以，随着系统的演进，重构是不可避免的。

虽然我刚刚说不支持大刀阔斧、推倒重来式的大重构，但持续的小重构我还是比较提倡的。它也是时刻保证代码质量、防止代码腐化的有效手段。换句话说，不要等到问题堆得太多了再去解决，要时刻有人对代码整体质量负责任，平时没事就改改代码。千万不要觉得重构代码就是浪费时间，不务正业！

特别是一些业务开发团队，有时候为了快速完成一个业务需求，只追求速度，到处 hard code，在完全不考虑非功能性需求、代码质量的情况下，堆砌烂代码。实际上，这种情况还是比较常见的。不过没关系，等你有时间了，一定要记着重构，不然烂代码越堆越多，总有一天代码会变得无法维护。

对项目与团队进行拆分

我们知道，团队人比较少，比如十几个人的时候，代码量不多，不超过 10 万行，怎么开发、怎么管理都没问题，大家互相都比较了解彼此做的东西。即便代码质量太差了，我们大不了把它重写一遍。但是，对于一个大型项目来说，参与开发的人员会比较多，代码量很大，有几十万、甚至几百万行代码，有几十、甚至几百号人同时开发维护，那研发管理就变得极其重要。

面对大型复杂项目，我们不仅仅需要对代码进行拆分，还需要对研发团队进行拆分。上一节课我们讲到了一些代码拆分的方法，比如模块化、分层等。同理，我们也可以把大团队拆成几个小团队。每个小团队对应负责一个小的项目（模块、微服务等），这样每个团队负责的项目包含的代码都不至于很多，也不至于出现代码质量太差无法维护的情况。