《[设计模式之美](https://time.geekbang.org/column/intro/250)》是极客时间上的一个代码学习系列，在学习之后特在此做记录和总结。

## 一、设计原则

**1）单一职责原则（Single Responsibility Principle，SRP）**

  一个类或者模块只负责完成一个职责（或者功能），模块可看作比类更加粗粒度的代码块，模块中包含多个类，多个类组成一个模块。

**2）开闭原则（Open Closed Principle，OCP）**

  添加一个新的功能，在已有代码基础上扩展代码（新增模块、类、方法等），而非修改已有代码（修改模块、类、方法等）。

**3）里式替换原则（Liskov Substitution Principle，LSP）**

  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。

**4）接口隔离原则（Interface Segregation Principle，ISP）**

  接口的调用者或使用者不应该强迫依赖它不需要的接口。

**5）依赖倒置原则（Dependency Inversion Principle，DIP）**

  高层模块不要依赖低层模块。高层模块和低层模块应该通过抽象来互相依赖。抽象不要依赖具体实现细节，具体实现细节依赖抽象。

**6）控制反转（Inversion Of Control，IOC）**

  “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可以通过框架来控制。流程的控制权从程序员“反转”到了框架。

**7）依赖注入（Dependency Injection，DI）**

  不通过 new() 的方式在类内部创建依赖类对象，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）给类使用。

**8）基于接口而非实现编程**

  设计初衷是将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。

**9）KISS**

  尽量保持简单。代码足够简单，也就意味着很容易读懂，bug 比较难隐藏。原则讲的是“如何做”的问题。

**10）YAGNI（You Ain’t Gonna Need It）**

  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。其核心思想就是：不要做过度设计。原则说的是“要不要做”的问题。

**11）DRY（Don’t Repeat Yourself）**

  三种典型的代码重复情况：实现逻辑重复、功能语义重复和代码执行重复。

**12）迪米特法则（Law of Demeter，LOD）**

  也叫最小知识原则（The Least Knowledge Principle），就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。

## 二、规范与重构

**1）可测试性**

  针对代码编写单元测试的难易程度。通过依赖注入，在编写单元测试的时候，可以通过 mock 的方法依赖外部服务。

**2）解耦**

1. 封装与抽象，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。
2. 中间层，简化模块或类之间的依赖关系，并能起到过渡的作用，让开发和重构同步进行，不互相干扰。
3. 模块化，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。
4. 其他设计思想和原则，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。

**3）编码规范**

1. 命名以能准确达意为目标，利用上下文简化命名，命名要可读、可搜索。
2. 注释的内容主要包含这样四个方面：做什么、为什么、怎么做、怎么用。对一些边界条件、特殊情况进行说明，以及对函数输入、输出、异常进行说明。
3. 类、函数规模不要超过一个显示屏的垂直高度。
4. 一行代码最长不能超过 IDE 显示的宽度。
5. 善用空行分割单元块，在类的成员变量与函数之间、静态成员变量与普通成员变量之间、各函数之间、甚至各成员变量之间，界限更加明确。
6. 缩进，Java 语言倾向于两格缩进，PHP 语言倾向于四格缩进。
7. 大括号是否要另起一行，PHP 程序员喜欢另起一行，Java 程序员喜欢跟上一条语句放到一起。
8. 类中成员的排列顺序，在 Java 类文件中，先要书写类所属的包名，然后再罗列 import 引入的依赖类。
9. 把代码分割成更小的单元块，要有模块化和抽象思维，善于将大块的复杂逻辑提炼成类或者函数，屏蔽掉细节，让阅读代码的人不至于迷失在细节中。
10. 避免函数参数过多，考虑函数是否职责单一，是否能通过拆分成多个函数的方式来减少参数。将函数的参数封装成对象。
11. 勿用函数参数来控制逻辑，不要在函数中使用布尔类型的标识参数来控制内部逻辑。
12. 函数设计要职责单一，相对于类和模块，函数的粒度比较小，代码行数少，所以在应用单一职责原则的时候，没有像应用到类或者模块那样模棱两可，能多单一就多单一。
13. 移除过深的嵌套层次，嵌套最好不超过两层，超过两层之后就要思考一下是否可以减少嵌套。
14. 学会使用解释性变量，常量取代魔法数字。使用解释性变量来解释复杂表达式。

**4）代码质量问题清单**

  下面 7 个是通用的关注点，可以作为一些常规检查项，套用在任何代码的重构上。 1. 目录设置是否合理、模块划分是否清晰、代码结构是否满足“高内聚、松耦合”？ 2. 是否遵循经典的设计原则和设计思想（SOLID、DRY、KISS、YAGNI、LOD 等）？ 3. 设计模式是否应用得当？是否有过度设计？ 4. 代码是否容易扩展？如果要添加新功能，是否容易实现？ 5. 代码是否可以复用？是否可以复用已有的项目代码或类库？是否有重复造轮子？ 6. 代码是否容易测试？单元测试是否全面覆盖了各种正常和异常的情况？ 7. 代码是否易读？是否符合编码规范（比如命名和注释是否恰当、代码风格是否一致等）？

  还要关注代码实现是否满足业务本身特有的功能和非功能需求。一些比较共性的关注点如下所示： 1. 代码是否实现了预期的业务需求？ 2. 逻辑是否正确？是否处理了各种异常情况？ 3. 日志打印是否得当？是否方便 debug 排查问题？ 4. 接口是否易用？是否支持幂等、事务等？ 5. 代码是否存在并发问题？是否线程安全？ 6. 性能是否有优化空间，比如，SQL、算法是否可以优化？ 7. 是否有安全漏洞？比如输入输出校验是否全面？

## 三、创建型设计模式

**1）单例模式（Singleton Design Pattern）**

  一个类只允许创建一个对象（或者实例），那这个类就是一个单例类。用来创建全局唯一的对象。

**2）工厂模式**

  用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。 1. 简单工厂（Simple Factory）利用 if 分支创建对象。 2. 工厂方法（Factory Method）定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。

**3）建造者模式（Builder Design Pattern）**

  建造者模式是用来创建复杂对象，可以通过设置不同的可选参数，“定制化”地创建不同的对象。

**4）原型模式（Prototype Design Pattern）**

  基于原型来创建对象的方式。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。

## 四、结构型设计模式

**1）代理模式（Proxy Design Pattern）**

  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。

**2）桥接模式（Bridge Design Pattern）**

  对于这个模式有两种不同的理解方式。 1. 将抽象和实现解耦，让它们可以独立变化。 2. 一个类存在两个（或多个）独立变化的维度，通过组合的方式，让这两个（或多个）维度可以独立进行扩展。

**3）装饰器模式（Decorator Design Pattern）**

  相对于简单的组合关系，有两个比较特殊的地方。 1. 装饰器类和原始类继承同样的父类，这样可以对原始类“嵌套”多个装饰器类。 2. 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。

  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。

  代理模式偏重业务无关，高度抽象和稳定性较高的场景。装饰器模式偏重业务相关，定制化诉求高，改动较频繁的场景。

**4）适配器模式（Adapter Design Pattern）**

  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。

  适配器模式可以看作一种“补偿模式”，用来补救设计上的缺陷。应用这种模式算是“无奈之举”。

  代理、桥接、装饰器和适配器都可以称为 Wrapper 模式，也就是通过 Wrapper 类二次封装原始类。 1. 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。 2. 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。 3. 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。 4. 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。

**5）门面模式（Facade Design Pattern）**

  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。子系统（subsystem）既可以是一个完整的系统，也可以是更细粒度的类或者模块。

  与适配器模式的区别： 1. 适配器模式是做接口转换，解决的是原接口和目标接口不匹配的问题。在代码结构上主要是继承加组合。 2. 门面模式做接口整合，解决的是多接口调用带来的问题。在代码结构上主要是封装。

**6）组合模式（Composite Design Pattern）**

  组合模式是将一组对象组织成树形结构，以表示一种“部分 - 整体”的层次结构。组合让客户端（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。

  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，业务需求可以通过在树上的递归遍历算法来实现。

**7）享元模式（Flyweight Design Pattern）**

  意图是复用对象，节省内存，前提是享元对象是不可变对象。

  如果这些重复的对象是不可变对象，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。

  不可变对象不能暴露任何 set() 等修改内部状态的方法。

## 五、行为型设计模式

**1）观察者模式（Observer Design Pattern）**

  也叫发布订阅模式（Publish-Subscribe Design Pattern），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。

**2）模板模式（Template Method Design Pattern）**

  可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。

**3）策略模式（Strategy Design Pattern）**

  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。

  策略模式解耦的是策略的定义、创建、使用这三部分。让每个部分都不至于过于复杂、代码量过多。

**4）职责链模式（Chain Of Responsibility Design Pattern）**

  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。

**5）状态模式**

  状态模式一般用来实现状态机，而状态机常用在游戏、工作流引擎等系统开发中。

  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。

**6）迭代器模式（Iterator Design Pattern）**

  也叫游标模式（Cursor Design Pattern）用来遍历集合对象。

  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。

  迭代器中需要定义 hasNext()、currentItem()、next() 三个最基本的方法。

**7）访问者者模式（Visitor Design Pattern）**

  允许一个或者多个操作应用到一组对象上，解耦操作和对象本身。

**8）备忘录模式（Memento Design Pattern）**

  也叫快照（Snapshot）模式，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。

**9）命令模式（Command Design Pattern）**

  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。

  在大部分编程语言中，函数没法作为参数传递给其他函数，也没法赋值给变量。借助命令模式，可以将函数封装成对象。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。

  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。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。

**10）解释器模式（Interpreter Design Pattern）**

  只在一些特定的领域会被用到，比如编译器、规则引擎、正则表达式。它能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。

**11）中介模式（Mediator Design Pattern）**

  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。

  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）。