设计模式三原则

发表于 2022-08-28 | 更新于 2023-04-06 | 设计模式

| 字数总计: 2.3k | 阅读时长: 6 分钟 | 阅读量: 2713 | 评论数: 0

配套视频课程已更新完毕，大家可通过以下两种方式观看视频讲解：

关注公众号： 爱编程的大丙 ，或者进入 大丙课堂 学习。

苏丙榅

合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。

我们在进行程序设计的时候，要尽可能地保证程序的可扩展性、可维护性和可读性，所以需要使用一些设计模式，这些设计模式都遵循了以下三个原则，下面来依次为大家介绍。

单一职责原则

C++ 面向对象三大特性之一的封装指的就是将单一事物抽象出来组合成一个类，所以我们在设计类的时候每个类中处理的是单一事物而不是某些事物的集合。

设计模式中所谓的单一职责原则，就是对一个类而言，应该仅有一个引起它变化的原因，其实就是将这个类所承担的职责单一化（就跟海贼王中的能力者一样，每个人只能吃一颗恶魔果实，拥有某一种能力【黑胡子这个 Bug 除外】）。

如果一个类承担的职责过多，就等于把这些职责耦合到了一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致设计变得脆弱，当变化发生时，设计会遭受到意想不到的破坏。

目录

最新文章

偷袭白胡子的这个男人被白胡子视为自己的儿子，他叫 斯库亚德 ，本来是一起去救 艾斯 的【此时的他是一个单一职责的类】，后来被 赤犬 挑拨离间想到了自己的过去，并萌生了别的想法【这个类被追加了一些其他的职责】，最终背叛并刺穿了白胡子的身体【这个类没能完成开始时的预期任务，就此废掉了】。由此可见，让一个类承担太多的职责绝非好事。

软件设计真正要做的事情就是，发现根据需求发现职责，并把这些职责进行分离，添加新的类，给当前类减负，越是这样项目才越容易维护。

开放封闭原则

CMake 保姆级教程（下）
2023-03-15

CMake 保姆级教程（上）
2023-03-06

访问者模式 - 再见，香波地群岛
2022-09-22

模板方法模式 - 和平主义者
2022-09-21

状态模式 - 文斯莫克・山治
2022-09-20

开放 – 封闭原则说的是软件实体（类、模块、函数等）可以扩展，但是不可以修改。也就是说 对于扩展是开放的，对于修改是封闭的 。

该原则是程序设计的一种理想模式，在很多情况下无法做到完全的封闭。但是作为设计人员，应该能够对自己设计的模块在哪些位置产生何种变化了然于胸，因此 需要在这些位置创建抽象类来隔离以后发生的这些同类变化（其实就是对多态的应用，创建新的子类并重写父类虚函数，用以更新处理动作）。

> 此处的抽象类，其实并不等价与 C++ 中完全意义上是抽象类（需要有纯虚函数），这里所说的抽象类只需要包含虚函数（纯虚函或非纯虚函数）能够实现多态即可。

草帽团船长路飞从出海到现在一共召集了 9 个伙伴，这些伙伴在船上的职责都是不一样的，有 音乐家 、 船工 、 舵手 、 航海士 、 剑士 、 考古学家 、 狙击手 、 厨师 、 船医 ，作为船长没有要求自己学习这些船员的技能【对自己来说是封闭的】，而是提出了伙伴的概念【这就是一个可变的抽象】，最终找到了优秀的伙伴加入【对外是开放的，每个伙伴都是这个抽象的具体实现，但他们的技能又有所不同】，事实证明这样做是对的，如果反其道而行之，不仅违背了开放封闭原则，也违背了单一职责原则。

开放 – 封闭原则是面向对象设计的核心所在，这样可以给我们设计出的程序带来巨大的好处，使其可维护性、可扩展性、可复用性、灵活性更好。

依赖倒转原则

关于依赖倒转原则，对应的是两条非常抽象的描述：

1. 高层模块不应该依赖低层模块，两个都应该依赖抽象。
2. 抽象不应该依赖细节，细节应该依赖抽象。

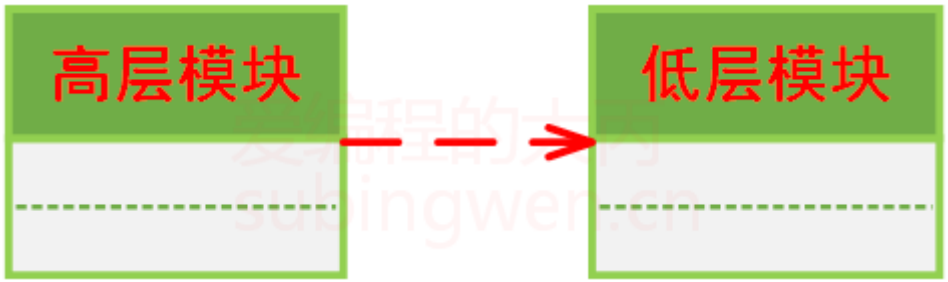
先用人话解释一下这两句话中的一些抽象概念：

- 高层模块：可以理解为上层应用，就是业务层的实现
- 低层模块：可以理解为底层接口，比如封装好的 API、动态库等
- 抽象：指的就是抽象类或者接口，在 C++ 中没有接口，只有抽象类

先举一个高层模块依赖低层模块的例子：

> *大聪明的项目组接了一个新项目，低层使用的是 `MySql` 的数据库接口，高层基于这套接口对数据库表进行了添删查改，实现了对业务层数据的处理。而后由于某些原因，要存储到数据库的数据量暴增，所以更换了 `Oracle` 数据库，由于低层的数据库接口变了，高层代码的数据库操作部分是直接调用了低层的接口，因此也需要进行对应的修改，无法实现对高层代码的直接复用，大聪明欲哭无泪。*

通过上面的例子可以得知，当依赖的低层模块变了就会牵一发而动全身，如果这样设计项目架构，对于程序猿来说，其工作量无疑是很重的。



如果要搞明白这个案例的解决方案以及抽象和细节之间的依赖关系，需要先了解另一个原则 — 里氏代换原则 。

里氏代换原则

所谓的里氏代换原则就是子类类型必须能够替换掉它们的父类类型。

关于这个原理的应用其实也很常见，比如在 Qt 中，所有窗口类型的类的构造函数都有一个 `QWidget*` 类型的参数（`QWidget` 类是所有窗口的基类），通过这个参数指定当前窗口的父对

象。虽然参数是窗口类的基类类型，但是我们在给其指定实参的大多数时候，指定的都是子类的对象，其实也就是相当于使用子类类型替换掉了它们的父类类型。

这个原则的要满足的第一个条件就是 继承 ， 其次还要求 子类继承的所有父类的属性和方法对于子类来说都是合理的。 关于这个是否合理下面举个栗子：

比如，对于哺乳动物来说都是胎生，但是有一种特殊的存在就是鸭嘴兽，它虽然是哺乳动物，但是是卵生。

如果我们设计了两个类：哺乳动物类和鸭嘴兽类，此时能够让鸭嘴兽类继承哺乳动物类吗？答案肯定是否定的，因为如果我们这么做了，鸭嘴兽就继承了胎生属性，这个属性和它自身的情况是不匹配的。如果想要遵循里氏代换原则，我们就不能让着两个类有继承关系。

如果我们创建了其它 的胎生的哺乳动物类，那么它们是可以继承哺乳动物这个类的， 在实际应用中就可以使用子类替换掉父类，同时功能也不会受到影响，父类实现了复用，子类也能在父类的基础上增加新的行为，这个就是里氏代换原则 。

上面在讲 依赖倒转原则 的时候说过， 抽象不应该依赖细节，细节应该依赖抽象。 也就意味着我们应该对细节进行封装，在 C++ 中就是将其放到一个抽象类中（C++ 中没有接口，不能像 Java 一样封装成接口），每个细节就相当于上面例子中的哺乳动物的一个特性，这样一来这个抽象的哺乳动物类就成了项目架构中高层和低层的桥梁，将二者整合到一起。

- 抽象类中提供的接口是固定不变的
- 低层模块是抽象类的子类，继承了抽象类的接口，并且可以重写这些接口的行为
- 高层模块想要实现某些功能，调用的是抽象类中的函数接口，并且是通过抽象类的父类指针引用其子类的实例对象（用子类类型替换父类类型），这样就实现了多态。

基于依赖倒转原则将项目的结构换成上图的这种模式之后，低层模块发生变化，对应高层模块是没有任何影响的，这样程序猿的工作量降低了，代码也更容易维护（说白了，依赖倒转原则就是对多

态的典型应用）。

设计模式

打赏

相关推荐

评论

昵称 邮箱 网址(http://)

来都来了, 说点什么吧...

提交

来发评论吧 ~

Powered By Valine
v1.5.1