# 面向对象七大原则

## 1.开闭原则（Open Close Principle）

**定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。**

开放-封闭原则的意思就是说，你设计的时候，时刻要考虑，尽量让这个类是足够好，写好了就不要去修改了，如果新需求来，我们增加一些类就完事了，原来的代码能不动则不动。这个原则有两个特性，一个是说“对于扩展是开放的”，另一个是说“对于更改是封闭的”。面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。这就是“开放-封闭原则”的精神所在

绝对的修改关闭是不可能的，无论模块是多么的‘封闭‘，都会存在一些无法对之封闭的变化，既然不可能完全封闭，设计人员必须对于他设计的模块应该对哪种变化封闭做出选择。他必须先猜测出最有可能发生的变化种类，然后构造抽象来隔离那些变化。在我们最初编写代码时，假设变化不会发生，当变化发生时，我们就创建抽象来隔离以后发生同类的变化。

简单的用一句经典的话来说：过去的事已成历史，是不可修改的，因为时光不可倒流，但现在或明天计划做什么，是可以自己决定(即扩展)的。

## 2.单一职责原则（Single Responsibility Principle）

**定义：一个类只负责一项职责，应该仅有一个引起它变化的原因**

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责P被分化为粒度更细的职责P1和P2。

遵循单一职责原的优点有：

1.可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；

2.提高类的可读性，提高系统的可维护性；

3.变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都需要遵循这一重要原则。

## 3.里氏代换原则（Liskov Substitution Principle）

对继承的概念有清晰的了解

**定义1：子类型必须能够替换掉它们的父类型。**

**定义2：如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。**

描述：一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别，也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化

比如，在生物学分类上，企鹅是一种鸟，但在编程世界里，企鹅却不能继承鸟。在面向对象设计时，子类拥有父类所有非private的行为和属性，鸟会飞，但企鹅不会飞，所以企鹅不能继承鸟类。

只有当子类可以替换掉父类，软件单位的功能不受影响时，父类才能真正被复用，而子类也能够在父类的基础上增加新的行为，正是有里氏代换原则，使得继承复用成为了可能。正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展，不然还谈什么扩展开放，修改关闭呢。

## 4.依赖倒转原则（Dependence Inversion Principle）

对抽象和接口的概念有基本的了解

**定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。即针对接口编程，不要针对实现编程**

依赖倒转其实就是谁也不要依靠谁，除了约定的接口，大家都可以灵活自如。依赖倒转可以说是面向对象设计的标志，用哪种语言来编写程序不重要，如果编写时考虑的都是如何针对抽象编程而不是针对细节编程，即程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程化的设计了。如果设计的各个部件或类相互依赖，这样就是耦合度高，难以维护和扩展，这也就体现不出面向对象的好处了。

依赖倒转原则，好比一个团队，有需求组，开发组，测试组，开发组和测试组都是面对同样的需求后，做自己相应的工作，而不应该是测试组按照开发组理解的需求去做测试用例，也就是说开发组和测试组都是直接面向需求组工作，大家的目的是一样的，保证产品按时上线，需求是不依赖于开发和测试的。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。

总之，依赖倒置原则就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

## 5.接口隔离原则（Interface Segregation Principle）

这个对接口的使用要求很高，可以不必了解

**定义：类之间的依赖关系应该建立在最小的接口上**

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉的接口隔离原则跟单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

## 6.迪米特法则（Law Of Demeter）

这个是高阶要求，可以不必了解

**定义：一个软件实体应当尽可能少的与其他实体发生相互作用**

迪米特法则其根本思想，是强调了类之间的松耦合，类之间的耦合越弱,越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成影响，也就是说，信息的隐藏促进了软件的复用。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

一句话总结就是：一个对象应该对其他对象保持最少的了解。

## 7.合成/聚合复用原则

这个对类之间的关系要掌握得非常清楚，可以不必了解

**定义：尽量使用合成/聚合，而不是通过继承达到复用的目的**

该原则就是说要尽量的使用合成和聚合，而不是继承关系达到复用的目的。在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分：新的对象通过向这些对象的委派达到复用已有功能的目的。

参考资料

<https://blog.csdn.net/u011288271/article/details/52497602>

<http://www.cnblogs.com/throwable/p/9315318.html>

<http://www.ibcibc.com/thread-16-1-1.html>