[面向对象六大原则](https://www.cnblogs.com/qifengshi/p/5709594.html)

这是设计模式系列开篇的第一篇文章。也是我学习设计模式过程中的总结。这篇文章主要讲的是面向对象设计中，我们应该遵循的六大原则。只有掌握了这些原则，我们才能更好的理解设计模式。  
我们接下来要介绍以下6个内容。

1. 单一职责原则——SRP
2. 开闭原则——OCP
3. 里式替换原则——LSP
4. 依赖倒置原则——DIP
5. 接口隔离原则——ISP
6. 迪米特原则——LOD

**单一职责原则**

单一职责原则的定义是就一个类而言，应该仅有一个引起他变化的原因。也就是说一个类应该只负责一件事情。如果一个类负责了方法M1,方法M2两个不同的事情，当M1方法发生变化的时候，我们需要修改这个类的M1方法，但是这个时候就有可能导致M2方法不能工作。这个不是我们期待的，但是由于这种设计却很有可能发生。所以这个时候，我们需要把M1方法，M2方法单独分离成两个类。让每个类只专心处理自己的方法。  
单一职责原则的好处如下：

1. 可以降低类的复杂度，一个类只负责一项职责，这样逻辑也简单很多
2. 提高类的可读性，和系统的维护性，因为不会有其他奇怪的方法来干扰我们理解这个类的含义
3. 当发生变化的时候，能将变化的影响降到最小，因为只会在这个类中做出修改。

**开闭原则**

开闭原则和单一职责原则一样，是非常基础而且一般是常识的原则。开闭原则的定义是软件中的对象(类，模块，函数等)应该对于扩展是开放的，但是对于修改是关闭的。  
当需求发生改变的时候，我们需要对代码进行修改，这个时候我们应该尽量去扩展原来的代码，而不是去修改原来的代码，因为这样可能会引起更多的问题。  
这个准则和单一职责原则一样，是一个大家都这样去认为但是又没规定具体该如何去做的一种原则。  
开闭原则我们可以用一种方式来确保他，我们用抽象去构建框架，用实现扩展细节。这样当发生修改的时候，我们就直接用抽象了派生一个具体类去实现修改。

**里氏替换原则**

里氏替换原则是一个非常有用的一个概念。他的定义

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在所有对象o1都替换成o2的时候，程序P的行为都没有发生变化，那么类型T2是类型T1的子类型。

这样说有点复杂，其实有一个简单的定义

所有引用基类的地方必须能够透明地使用其子类的对象。

里氏替换原则通俗的去讲就是：子类可以去扩展父类的功能，但是不能改变父类原有的功能。他包含以下几层意思：

1. 子类可以实现父类的抽象方法，但是不能覆盖父类的非抽象方法。
2. 子类可以增加自己独有的方法。
3. 当子类的方法重载父类的方法时候，方法的形参要比父类的方法的输入参数更加宽松。
4. 当子类的方法实现父类的抽象方法时，方法的返回值要比父类更严格。

里氏替换原则之所以这样要求是因为继承有很多缺点，他虽然是复用代码的一种方法，但同时继承在一定程度上违反了封装。父类的属性和方法对子类都是透明的，子类可以随意修改父类的成员。这也导致了，如果需求变更，子类对父类的方法进行一些复写的时候，其他的子类无法正常工作。所以里氏替换法则被提出来。  
确保程序遵循里氏替换原则可以要求我们的程序建立抽象，通过抽象去建立规范，然后用实现去扩展细节，这个是不是很耳熟，对，里氏替换原则和开闭原则往往是相互依存的。

**依赖倒置原则**

依赖倒置原则指的是一种特殊的解耦方式，使得高层次的模块不应该依赖于低层次的模块的实现细节的目的，依赖模块被颠倒了。  
这也是一个让人难懂的定义，他可以简单来说就是

高层模块不应该依赖底层模块，两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

在Java 中抽象指的是接口或者抽象类，两者皆不能实例化。而细节就是实现类，也就是实现了接口或者继承了抽象类的类。他是可以被实例化的。高层模块指的是调用端，底层模块是具体的实现类。在Java中，依赖倒置原则是指模块间的依赖是通过抽象来发生的，实现类之间不发生直接的依赖关系，其依赖关系是通过接口是来实现的。这就是俗称的面向接口编程。  
我们下面有一个例子来讲述这个问题。这个例子是工人用锤子来修理东西。我们的代码如下：

public class Hammer {

public String function(){

return "用锤子修理东西";

}

}

public class Worker {

public void fix(Hammer hammer){

System.out.println("工人" + hammer.function());

}

public static void main(String[] args) {

new Worker().fix(new Hammer());

}

}

这个是一个很简单的例子，但是如果我们要新增加一个功能，工人用 螺丝刀来修理东西，在这个类，我们发现是很难做的。因为我们Worker类依赖于一个具体的实现类Hammer。所以我们用到面向接口编程的思想，改成如下的代码：

public interface Tools {

public String function();

}

然后我们的Worker是通过这个接口来于其他细节类进行依赖。代码如下：

public class Worker {

public void fix(Tools tool){

System.out.println("工人" + tool.function());

}

public static void main(String[] args) {

new Worker().fix(new Hammer());

new Worker().fix(new Screwdriver());

}

}

我们的Hammer类与Screwdriver类实现这个接口

public class Hammer implements Tools{

public String function(){

return "用锤子修理东西";

}

}

public class Screwdriver implements Tools{

@Override

public String function() {

return "用螺丝刀修理东西";

}

}

这样，通过面向接口编程，我们的代码就有了很高的扩展性，降低了代码之间的耦合度，提高了系统的稳定性。

**接口隔离原则**

接口隔离原则的定义是

客户端不应该依赖他不需要的接口

换一种说法就是类间的依赖关系应该建立在最小的接口上。这样说好像更难懂。我们通过一个例子来说明。我们知道在Java中一个具体类实现了一个接口，那必然就要实现接口中的所有方法。如果我们有一个类A和类B通过接口I来依赖，类B是对类A依赖的实现，这个接口I有5个方法。但是类A与类B只通过方法1,2,3依赖，然后类C与类D通过接口I来依赖，类D是对类C依赖的实现但是他们却是通过方法1,4,5依赖。那么是必在实现接口的时候，类B就要有实现他不需要的方法4和方法5 而类D就要实现他不需要的方法2，和方法3。这简直就是一个灾难的设计。  
所以我们需要对接口进行拆分，就是把接口分成满足依赖关系的最小接口，类B与类D不需要去实现与他们无关接口方法。比如在这个例子中，我们可以把接口拆成3个，第一个是仅仅由方法1的接口，第二个接口是包含2,3方法的，第三个接口是包含4,5方法的。  
这样，我们的设计就满足了接口隔离原则。  
以上这些设计思想用英文的第一个字母可以组成SOLID ，满足这个5个原则的程序也被称为满足了SOLID准则。

**迪米特原则**

迪米特原则也被称为最小知识原则，他的定义

一个对象应该对其他对象保持最小的了解。

因为类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大，所以这也是我们提倡的软件编程的总的原则：低耦合，高内聚。  
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。

这里我们可以用一个现实生活中的例子来讲解一下。比如我们需要一张CD,我们可能去音像店去问老板有没有我们需要的那张CD，老板说现在没有，等有的时候你们来拿就行了。在这里我们不需要关心老板是从哪里，怎么获得的那张CD，我们只和老板（直接朋友）沟通，至于老板从他的朋友那里通过何种条件得到的CD，我们不关心，我们不和老板的朋友（陌生人）进行通信，这个就是迪米特的一个应用。说白了，就是一种中介的方式。我们通过老板这个中介来和真正提供CD的人发生联系。

**总结**

到这里，面向对象的六大原则，就写完了。我们看出来，这些原则其实都是应对不断改变的需求。每当需求变化的时候，我们利用这些原则来使我们的代码改动量最小，而且所造成的影响也是最小的。但是我们在看这些原则的时候，我们会发现很多原则并没有提供一种公式化的结论，而即使提供了公式化的结论的原则也只是建议去这样做。这是因为，这些设计原则本来就是从很多实际的代码中提取出来的，他是一个经验化的结论。怎么去用它，用好他，就要依靠设计者的经验。否则一味者去使用设计原则可能会使代码出现过度设计的情况。大多数的原则都是通过提取出抽象和接口来实现，如果发生过度的设计，就会出现很多抽象类和接口，增加了系统的复杂度。让本来很小的项目变得很庞大，当然这也是Java的特性（任何的小项目都会做成中型的项目）。