# 面向对象设计原则 · 设计模式Java版

https://quanke.gitbooks.io/design-pattern-java/content/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E8%AE%BE%E8%AE%A1%E5%8E%9F%E5%88%99.html

面向对象设计原则

对于面向对象软件系统的设计而言，在支持可维护性的同时，提高系统的可复用性是一个至关重要的问题，如何同时提高一个软件系统的可维护性和可复用性是面向对象设计需要解决的核心问题之一。在面向对象设计中，可维护性的复用是以设计原则为基础的。每一个原则都蕴含一些面向对象设计的思想，可以从不同的角度提升一个软件结构的设计水平。

面向对象设计原则为支持可维护性复用而诞生，这些原则蕴含在很多设计模式中，它们是从许多设计方案中总结出的指导性原则。面向对象设计原则也是我们用于评价一个设计模式的使用效果的重要指标之一，在设计模式的学习中，大家经常会看到诸如“XXX模式符合XXX原则”、“XXX模式违反了XXX原则”这样的语句。

最常见的7种面向对象设计原则如下表所示： 表1 7种常用的面向对象设计原则

|  |  |  |
| --- | --- | --- |
| 设计原则名称 | 定 义 | 使用频率 |
| 单一职责原则 (Single Responsibility Principle, SRP) | 一个类只负责一个功能领域中的相应职责 | ★★★★☆ |
| 开闭原则 (Open-Closed Principle, OCP) | 软件实体应对扩展开放，而对修改关闭 | ★★★★★ |
| 里氏代换原则 (Liskov Substitution Principle, LSP) | 所有引用基类对象的地方能够透明地使用其子类的对象 | ★★★★★ |
| 依赖倒转原则 (Dependence Inversion Principle, DIP) | 抽象不应该依赖于细节，细节应该依赖于抽象 | ★★★★★ |
| 接口隔离原则 (Interface Segregation Principle, ISP) | 使用多个专门的接口，而不使用单一的总接口 | ★★☆☆☆ |
| 合成复用原则 (Composite Reuse Principle, CRP) | 尽量使用对象组合，而不是继承来达到复用的目的 | ★★★★☆ |
| 迪米特法则 (Law of Demeter, LoD) | 一个软件实体应当尽可能少地与其他实体发生相互作用 | ★★★☆☆ |

# 谈谈面向对象设计(OOD)原则 - 简书

http://www.jianshu.com/p/e378025920f8

拥有一把锤子未必能成为建筑师

最近在项目开发过程中碰到了一些问题，发现在每波迭代开发过程中，经常需要去修改之前的代码，虽然出现这样的情形很正常，新的需求必然会带来新的功能新的设计，导致之前的代码受到影响。记得看过一个笑话：

“杀一个程序员不需要用枪，改三次需求就可以了”

其实需求设计是一个方面，另外我们作为设计开发人员有时候也需要去反省，反省一下代码的设计是否合理，为什么新功能的在原有代码上扩展会那么难，为什么我们的代码这么不稳定，牵一发而动全身？    
我觉得能成为一名程序员，至少不会是一个笨的人，要完成一个功能，总能想办法实现（不然早被开除啦～），但实现的方法思路却有好有坏，不过我认为思路可以被引导，软件开发不是才刚开始，它已经存在一段时间，我们可以吸收前人的一些经验教训来提高自己，比如GOF的《设计模式：可复用面向对象软件的基础》，帮我们总结了很多问题的解决思路。这段时间也花了点时间学习面向对象设计的一些思想，也谈谈自己的一些理解。    
提到设计模式，我想很多人都看过这块的一些书籍，不过不知道会不会有跟我一样的困惑：看的时候都理解，但是实际开发的时候却无法融入，后面慢慢就忘记了。尴尬，可能我们只是看到了某个模式的表面，而隐藏在模式后面的一些“真理”却没有去挖掘，这个模式是要解决什么问题？其实模式设计的背后都是为了遵循某种设计原则。

“比设计模式更重要的是设计原则”

面相对象设计的概念大家也都知道，它的设计目标就是希望软件系统能做到以下几点：

* 可扩展：新特性能够很容易的添加到现有系统中，不会影响原本的东西
* 可修改：当修改某一部分的代码时，不会影响到其它不相关的部分
* 可替代：将系统中某部分的代码用其它有相同接口的类替换时，不会影响到现有系统

这几个可以用来检测我们的软件系统是不是设计得合理，而如何设计出易于维护和扩展的软件系统是有设计原则可以遵循指导的，[Robert C. Martin](https://en.wikipedia.org/wiki/Robert_Cecil_Martin)提出了面相对象设计的五个基本原则（SOLID）:

* S－单一职责原则
* O－开放关闭原则
* L－里氏替换原则
* I－接口隔离原则
* D－依赖倒置原则

我们在进行面相对象设计的时候应该牢记这几个原则，这能让你成为更优秀的设计开发人员－－－至少你的代码不会那么烂，下面来简单了解一下这几个原则。

单一职责原则：Single Responsibility Principle

一个类有且仅有一个职责，只有一个引起它变化的原因。

简单来说一个类只做好一件事就行，不去管跟自己不相干的，狗拿耗子多管闲事，其核心就是解耦以及高内聚。这个原则看着很简单，我们在写代码的时候即便不知道这个原则也会往这个方向靠拢，写出功能相对单一的类，不过这个原则很容易违背，因为可能由于某种原因，原来功能单一的类需要被细化成颗粒更小的职责1跟职责2，所以在每次迭代过程中可能需要重新梳理重构之前编写的代码，将不同的职责封装到不同的类或者模块中。    
举个栗子：

@interface DataTransfer : NSObject

-(void)upload:(NSData \*)data; //上传数据

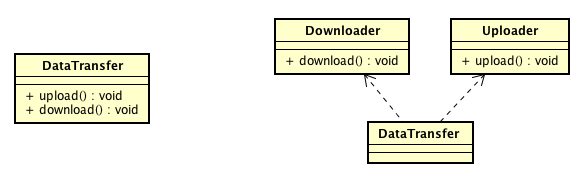
-(void)download(NSString\*)url; //根据URL下载东西

@end

DataTransfer包含上传跟下载功能，仔细考虑可以发现这相当于实现了两个功能，一个负责上传的相关逻辑，另一个负责下载的逻辑，而这个两个功能相对对立，当有一个功能改变的时候，比如我们之前是使用AFNetworking，现在想换成其它第三方或者nsurlconnection来实现上传跟下载：

* 上传方式变更，导致DataTransfer变更
* 下载方式变更，导致 DataTransfer变更

这就违反了单一职责的原则，所以需要将不同的功能拆解成两个不同的类，来负责各自的职责，不过这个拆的粒度可能因人而已，有时候并不需要拆的过细，不要成了为设计而设计。



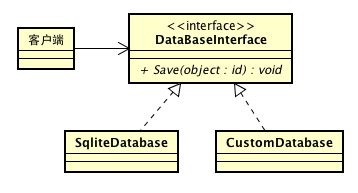
单一职责

在我们项目中经常看到很多违反这条原则的代码，而且违反的比较明显，许多类都是丰富功能的超级集合，整个类变得臃肿难以理解，这时候就需要我们有意识地去重构了。

开放关闭原则：Open Closed Principle

开闭原则的定义是说一个软件实体如类，模块和函数应该对扩展开放，而对修改关闭，具体来说就是你应该通过扩展来实现变化，而不是通过修改原有的代码来实现变化，该原则是面相对象设计最基本的原则。    
之前说过在项目中每当需求需改的时候经常需要对代码有很大的改动，很大程度上就是因为我们对这个原则理解的不够透彻。    
开闭原则的关键在于抽象，我们需要抽象出那些不会变化或者基本不变的东西，这部分东西相对稳定，这也就是对修改关闭的地方（这并不意味着不可以再修改），而对于那些容易变化的部分我们也对其封装，但是这部分是可以动态修改的，这也就是对扩展开发的地方，比如设计模式中的策略模式和模板模式就是在实现这个原则（现在应该对模式有更感性的认识了吧～）。

举个例子：我们需要保存对象到数据库当中，其中有个类似save()的保存方法，这部分应该是不变的，接口相对稳定，而具体保存的实现却有可能不同，我们现在可能是保存在Sqlite数据库中，假如以后如果想保存到一个自己实现的数据库中时，我们只需要实现一个拥有同样接口的扩展类添加进去即可，这就是对扩展开放，不会对之前的代码造成任何影响，就可以实现保存到新数据库的功能，保证了系统的稳定性。



开闭原则

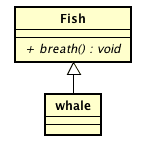
实现开闭原则的指导思想就是：

* 抽象出相对稳定的接口，这部分应该不改动或者很少改动
* 封装变化

不过在软件开发过程中，要一开始就完全按照开闭原则来可能比较困难，更多的情况是在不断的迭代重构过程中去改进，在可预见的变化范围内去做设计。

里氏替代原则：Liskov Substitution Principle

该原则的定义：所有引用基类的地方必须能透明地使用其子类的对象。简单来说，所有使用基类代码的地方，如果换成子类对象的时候还能够正常运行，则满足这个原则，否则就是继承关系有问题，应该废除两者的继承关系，这个原则可以用来判断我们的对象继承关系是否合理。    
比如有一个鲸鱼的类，我们让鲸鱼继承于鱼类，然后鱼类有个呼吸的功能：



鲸鱼继承自鱼类

然后在水里的时候，鱼能够进行呼吸：

if(isInwater){

//在水中了，开始呼吸

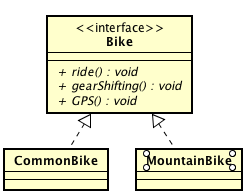
fish.breath();

}

当我们把鲸鱼这个子对象替换原来的基类鱼对象，鲸鱼在水里开始呼吸，这时问题就出现了，鲸鱼是哺乳动物，在水里呼吸是没法呼吸的，一直在水里就GG思密达了，所以这违反了该原则，我们就可以判断鲸鱼继承于鱼类不合理，需要去重新设计。    
通常在设计的时候，我们都会优先采用组合而不是继承，因为继承虽然减少了代码，提高了代码的重用性，但是父类跟子类会有很强的耦合性，破坏了封装。

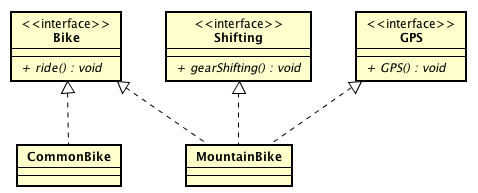
接口隔离原则：Interface Segregation Principle

该原则的定义：不能强迫用户去依赖那些他们不使用的接口。简单来说就是客户端需要什么接口，就提供给它什么样的接口，其它多余的接口就不要提供，不要让接口变得臃肿，否则当对象一个没有使用的方法被改变了，这个对象也将会受到影响。接口的设计应该遵循最小接口原则，其实这也是高内聚的一种表现，换句话说，使用多个功能单一、高内聚的接口总比使用一个庞大的接口要好。    
举个简单的例子：比如我们有个自行车接口，这个接口包含了很多方法，包括GPS定位，以及换挡的方法



不满足ISP原则

然后我们发现即便普通的自行车也需要实现GPS定位以及换挡的功能，显然这违背了接口隔离的原则。遵循接口最小化的原则，我们重新设计：



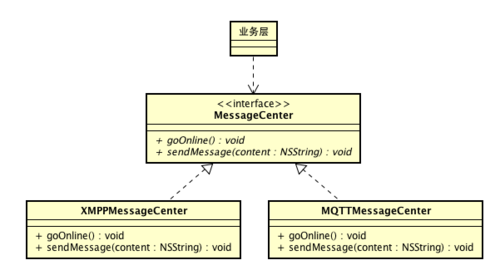
满足ISP原则

这样一来每个接口的功能相对单一，使用多个专门的接口比使用一个总的接口要好，假如我们的山地车没有没有GPS定位的功能，我们不去继承实现对应的接口即可，在iOS开发中有很多这样的例子，比如UITalbleView的代理有两个不同的接口，UITableViewDataSource专门负责需要显示的内容，UITableViewDelegate专门负责一些view的自定义显示，然后我们会继承多个接口，这就满足了ISP原则。

@interface ViewController () <UITableViewDataSource,UITableViewDelegate,OtherProtocol>

依赖倒置原则：Dependence Inversion Principle

该原则的定义：高层模块不应该依赖低层模块，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。其实这就是我们经常说的“针对接口编程”，这里的接口就是抽象，我们应该依赖接口，而不是依赖具体的实现来编程。    
如你在Sqlite数据库的基础上开发一套新的数据库系统AWEDatabase，这时候Sqlite相当于底层模块，而你的AWEDatabase就属于高层模块；而从AWEDatabase开发使用者来看，他的业务层就相当于高层模块，而AWEDatabase就变成底层模块了，所以模块的高低应该是从开发者当前的角度来看的，不过DIP原则从不同角度来看它都适合且需要被遵守。假如我们高层模块直接依赖于底层模块，带来的后果是每次底层模块改动，高层模块就会受到影响，整个系统就变得不稳定，这也违反了开放关闭原则。    
通常我们会通过引入中间层的方式来解决这个问题，这个中间层相当于一个抽象接口层，高层模块和底层模块都依赖于这个中间层来交互，这样只要中间抽象层保持不变，底层模块改变不会影响到高层模块，这就满足了开放关闭原则；而且假如高层模块跟底层模块同时处于开发阶段，这样有了中间抽象层之后，每个模块都可以针对这个抽象层的接口同时开发，高层模块就不需要等到底层模块开发完毕才能继续了。    
比如在我们项目中有涉及IM的功能，现在这个IM模块采用的是XMPP协议来实现，客户端通过这个模块来实现消息的收发，但是假如后面我们想要换成其它协议，比如MQTT等，针对接口编程的话就可以让我们很轻松的实现模块替换：



针对接口编程

@protocol MessageDelegate <NSObject>

@required

-(void)goOnline;

-(void)sendMessage:(NSString\*)content;

@end

//xmpp实现

@interface XMPPMessageCenter <MessageDelegate>

@end

//MQTT实现

@interface MQTTMessageCenter <MessageDelegate>

@end

//业务层

@interface BussinessLayer

//使用遵循MessageDelegate协议的对象，针对接口编程，以后替换也很方便

@property(nonatomic,strong)id<MessageDelegate> messageCenter;

@end

当我们在进行面向对象设计的时候应该充分考虑上面这几个原则，一开始可能设计并不完美，不过可以在重构的过程中不断完善。但其实很多人都跳过了设计这个环节，拿到一个模块直接动手编写代码，更不用说去思考设计了，项目中也有很多这样的例子。当然对于简单的模块或许不用什么设计，不过假如模块相对复杂的话，能够在动手写代码之前好好设计思考一下，养成这个习惯，肯定会对编写出可读性、稳定性以及可扩展性较高的代码有帮助。

最关键的软件开发工具是受过良好设计原则训练的思维。

文／树下的老男孩（简书作者）  
原文链接：http://www.jianshu.com/p/e378025920f8  
著作权归作者所有，转载请联系作者获得授权，并标注“简书作者”。

# 一些软件设计的原则 | | 酷 壳 - CoolShell

http://coolshell.cn/articles/4535.html

以前本站向大家介绍过一些软件开发的原则，比如[优质代码的十诫](http://coolshell.cn/articles/1007.html)和[Unix传奇(下篇)](http://coolshell.cn/articles/2324.html)中所以说的UNIX的设计原则。相信大家从中能够从中学了解到一些设计原理方面的知识，正如我在《[再谈“我是怎么招聘程序”](http://coolshell.cn/articles/4506.html)》中所说的，一个好的程序员通常由其操作技能、知识水平，经验层力和能力四个方面组成。在这里想和大家说说设计中的一些原则，我认为这些东西属于长期经验总结出来的知识。这些原则，每一个程序员都应该了解。但是请不要教条主义，在使用的时候还是要多多考虑实际情况。其实，**下面这些原则，不单单只是软件开发，可以推广到其它生产活动中，甚至我们的生活中**。

Don’t Repeat Yourself (DRY)

DRY 是一个最简单的法则，也是最容易被理解的。但它也可能是最难被应用的（因为要做到这样，我们需要在泛型设计上做相当的努力，这并不是一件容易的事）。它意味着，当我们在两个或多个地方的时候发现一些相似的代码的时候，我们需要把他们的共性抽象出来形一个唯一的新方法，并且改变现有的地方的代码让他们以一些合适的参数调用这个新的方法。

**参考**：<http://en.wikipedia.org/wiki/Don%27t_repeat_yourself>

Keep It Simple, Stupid (KISS)

KISS原则在设计上可能最被推崇的，在家装设计，界面设计 ，操作设计上，复杂的东西越来越被众人所BS了，而简单的东西越来越被人所认可，比如[这些UI的设计](http://coolshell.cn/articles/1907.html)和我们[中国网页](http://coolshell.cn/articles/3605.html)（尤其是[新浪的网页](http://coolshell.cn/articles/3872.html)）者是负面的例子。“宜家”（IKEA）简约、效率的家居设计、生产思路；“微软”（Microsoft）“所见即所得”的理念；“谷歌”（Google)简约、直接的商业风格，无一例外的遵循了“kiss”原则，也正是“kiss”原则，成就了这些看似神奇的商业经典。而苹果公司的iPhone/iPad将这个原则实践到了极至。

把一个事情搞复杂是一件简单的事，但要把一个复杂的事变简单，这是一件复杂的事。

**参考**：<http://en.wikipedia.org/wiki/KISS_principle>

Program to an interface, not an implementation

这是设计模式中最根本的哲学，注重接口，而不是实现，依赖接口，而不是实现。接口是抽象是稳定的，实现则是多种多样的。以后面我们会面向对象的SOLID原则中会提到我们的依赖倒置原则，就是这个原则的的另一种样子。还有一条原则叫 **Composition over inheritance**（喜欢组合而不是继承），这两条是那23个经典设计模式中的设计原则。

Command-Query Separation (CQS)  – 命令-查询分离原则

* 查询：当一个方法返回一个值来回应一个问题的时候，它就具有查询的性质；
* 命令：当一个方法要改变对象的状态的时候，它就具有命令的性质；

通常，一个方法可能是纯的Command模式或者是纯的Query模式，或者是两者的混合体。在设计接口时，如果可能，应该尽量使接口单一化，保证方法的行为严格的是命令或者是查询，这样查询方法不会改变对象的状态，没有副作用，而会改变对象的状态的方法不可能有返回值。也就是说：如果我们要问一个问题，那么就不应该影响到它的答案。实际应用，要视具体情况而定，语义的清晰性和使用的简单性之间需要权衡。将Command和Query功能合并入一个方法，方便了客户的使用，但是，降低了清晰性，而且，可能不便于基于断言的程序设计并且需要一个变量来保存查询结果。

在系统设计中，很多系统也是以这样原则设计的，查询的功能和命令功能的系统分离，这样有则于系统性能，也有利于系统的安全性。

**参考**：<http://en.wikipedia.org/wiki/Command-query_separation>

You Ain’t Gonna Need It (YAGNI)

这个原则简而言之为——只考虑和设计必须的功能，避免过度设计。只实现目前需要的功能，在以后您需要更多功能时，可以再进行添加。

* 如无必要，勿增复杂性。
* 软件开发先是一场沟通博弈。

以前本站有一篇关于[过度重构的文章](http://coolshell.cn/articles/3005.html)，这个示例就是这个原则的反例。而，WebSphere的设计者就[表示过他过度设计了这个产品](http://www.bbc.co.uk/news/business-11944966)。我们的程序员或是架构师在设计系统的时候，会考虑很多扩展性的东西，导致在架构与设计方面使用了大量折衷，最后导致项目失败。这是个令人感到讽刺的教训，因为本来希望尽可能延长项目的生命周期，结果反而缩短了生命周期。

**参考**：<http://en.wikipedia.org/wiki/You_Ain%27t_Gonna_Need_It>

Law of Demeter – 迪米特法则

迪米特法则(Law of Demeter)，又称“最少知识原则”（Principle of Least Knowledge），其来源于1987年荷兰大学的一个叫做Demeter的项目。Craig Larman把Law of Demeter又称作“不要和陌生人说话”。在《程序员修炼之道》中讲LoD的那一章叫作“解耦合与迪米特法则”。关于迪米特法则有一些很形象的比喻：

* 如果你想让你的狗跑的话，你会对狗狗说还是对四条狗腿说？
* 如果你去店里买东西，你会把钱交给店员，还是会把钱包交给店员让他自己拿？

和狗的四肢说话？让店员自己从钱包里拿钱？这听起来有点荒唐，不过在我们的代码里这几乎是见怪不怪的事情了。

对于LoD，正式的表述如下：

*对于对象 ‘O’ 中一个方法’M’，M应该只能够访问以下对象中的方法：*

1. *对象O；*
2. *与O直接相关的Component Object；*
3. *由方法M创建或者实例化的对象；*
4. *作为方法M的参数的对象。*

在《Clean Code》一书中，有一段Apache framework中的一段违反了LoD的代码：

final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();

这么长的一串对其它对象的细节，以及细节的细节，细节的细节的细节……的调用，增加了耦合，使得代码结构复杂、僵化，难以扩展和维护。

在《重构》一书中的代码的环味道中有一种叫做“Feature Envy”(依恋情结），形象的描述了一种违反了LoC的情况。Feature Envy就是说一个对象对其它对象的内容更有兴趣，也就是说老是羡慕别的对象的成员、结构或者功能，大老远的调用人家的东西。这样的结构显然是不合理的。我们的程序应该写得比较“害羞”。不能像前面例子中的那个不把自己当外人的店员一样，拿过客人的钱包自己把钱拿出来。“害羞”的程序只和自己最近的朋友交谈。这种情况下应该调整程序的结构，让那个对象自己拥有它羡慕的feature，或者使用合理的设计模式（例如Facade和Mediator）。

**参考**：<http://en.wikipedia.org/wiki/Principle_of_Least_Knowledge>

面向对象的S.O.L.I.D 原则

一般来说这是面向对象的五大设计原则，但是，我觉得这些原则可适用于所有的软件开发。

**Single Responsibility Principle (SRP) – 职责单一原则**

关于单一职责原则，其核心的思想是：**一个类，只做一件事，并把这件事做好，其只有一个引起它变化的原因**。单一职责原则可以看作是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。职责过多，可能引起它变化的原因就越多，这将导致职责依赖，相互之间就产生影响，从而极大的损伤其内聚性和耦合度。单一职责，通常意味着单一的功能，因此不要为一个模块实现过多的功能点，以保证实体只有一个引起它变化的原因。

* Unix/Linux是这一原则的完美体现者。各个程序都独立负责一个单一的事。
* Windows是这一原则的反面示例。几乎所有的程序都交织耦合在一起。

**Open/Closed Principle (OCP) – 开闭原则**

关于开发封闭原则，其核心的思想是：模块是可扩展的，而不可修改的。也就是说，**对扩展是开放的，而对修改是封闭的**。

* 对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。
* 对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

对于面向对象来说，需要你依赖抽象，而不是实现，23个经典设计模式中的“策略模式”就是这个实现。对于非面向对象编程，一些API需要你传入一个你可以扩展的函数，比如我们的C 语言的qsort()允许你提供一个“比较器”，STL中的容器类的内存分配，ACE中的多线程的各种锁。对于软件方面，浏览器的各种插件属于这个原则的实践。

**Liskov substitution principle (LSP) – 里氏代换原则**

软件工程大师Robert C. Martin把里氏代换原则最终简化为一句话：“Subtypes must be substitutable for their base types”。也就是，子类必须能够替换成它们的基类。即：子类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。另外，不应该在代码中出现if/else之类对子类类型进行判断的条件。里氏替换原则LSP是使代码符合开闭原则的一个重要保证。正是由于子类型的可替换性才使得父类型的模块在无需修改的情况下就可以扩展。

这么说来，似乎有点教条化，我非常建议大家看看这个原则个两个最经典的案例——“正方形不是长方形”和“鸵鸟不是鸟”。通过这两个案例，你会明白《墨子 小取》中说的 ——“娣，美人也，爱娣，非爱美人也….盗，人也；恶盗，非恶人也。”——妹妹虽然是美人，但喜欢妹妹并不代表喜欢美人。盗贼是人，但讨厌盗贼也并不代表就讨厌人类。**这个原则让你考虑的不是语义上对象的间的关系，而是实际需求的环境**。

在很多情况下，在设计初期我们类之间的关系不是很明确，LSP则给了我们一个判断和设计类之间关系的基准：需不需要继承，以及怎样设计继承关系。

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

接口隔离原则意思是把功能实现在接口中，而不是类中，使用多个专门的接口比使用单一的总接口要好。

举个例子，我们对电脑有不同的使用方式，比如：写作，通讯，看电影，打游戏，上网，编程，计算，数据等，如果我们把这些功能都声明在电脑的抽类里面，那么，我们的上网本，PC机，服务器，笔记本的实现类都要实现所有的这些接口，这就显得太复杂了。所以，我们可以把其这些功能接口隔离开来，比如：工作学习接口，编程开发接口，上网娱乐接口，计算和数据服务接口，这样，我们的不同功能的电脑就可以有所选择地继承这些接口。

这个原则可以提升我们“搭积木式”的软件开发。对于设计来说，Java中的各种Event Listener和Adapter，对于软件开发来说，不同的用户权限有不同的功能，不同的版本有不同的功能，都是这个原则的应用。

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

高层模块不应该依赖于低层模块的实现，而是依赖于高层抽象。

举个例子，墙面的开关不应该依赖于电灯的开关实现，而是应该依赖于一个抽象的开关的标准接口，这样，当我们扩展程序的时候，我们的开关同样可以控制其它不同的灯，甚至不同的电器。也就是说，电灯和其它电器继承并实现我们的标准开关接口，而我们的开关产商就可不需要关于其要控制什么样的设备，只需要关心那个标准的开关标准。这就是依赖倒置原则。

这就好像浏览器并不依赖于后面的web服务器，其只依赖于HTTP协议。这个原则实在是太重要了，社会的分工化，标准化都是这个设计原则的体现。

**参考**：<http://en.wikipedia.org/wiki/Solid_(object-oriented_design)>

Common Closure Principle（CCP）– 共同封闭原则

一个包中所有的类应该对同一种类型的变化关闭。一个变化影响一个包，便影响了包中所有的类。一个更简短的说法是：一起修改的类，应该组合在一起（同一个包里）。如果必须修改应用程序里的代码，我们希望所有的修改都发生在一个包里（修改关闭），而不是遍布在很多包里。CCP原则就是把因为某个同样的原因而需要修改的所有类组合进一个包里。如果2个类从物理上或者从概念上联系得非常紧密，它们通常一起发生改变，那么它们应该属于同一个包。

CCP延伸了开闭原则（OCP）的“关闭”概念，当因为某个原因需要修改时，把需要修改的范围限制在一个最小范围内的包里。

**参考**：<http://c2.com/cgi/wiki?CommonClosurePrinciple>

Common Reuse Principle (CRP) – 共同重用原则

包的所有类被一起重用。如果你重用了其中的一个类，就重用全部。换个说法是，没有被一起重用的类不应该被组合在一起。CRP原则帮助我们决定哪些类应该被放到同一个包里。依赖一个包就是依赖这个包所包含的一切。当一个包发生了改变，并发布新的版本，使用这个包的所有用户都必须在新的包环境下验证他们的工作，即使被他们使用的部分没有发生任何改变。因为如果包中包含有未被使用的类，即使用户不关心该类是否改变，但用户还是不得不升级该包并对原来的功能加以重新测试。

CCP则让系统的维护者受益。CCP让包尽可能大（CCP原则加入功能相关的类），CRP则让包尽可能小（CRP原则剔除不使用的类）。它们的出发点不一样，但不相互冲突。

**参考**：<http://c2.com/cgi/wiki?CommonReusePrinciple>

Hollywood Principle – 好莱坞原则

好莱坞原则就是一句话——“don’t call us, we’ll call you.”。意思是，好莱坞的经纪人们不希望你去联系他们，而是他们会在需要的时候来联系你。也就是说，所有的组件都是被动的，所有的组件初始化和调用都由容器负责。组件处在一个容器当中，由容器负责管理。

简单的来讲，就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。这也就是所谓“控制反转”的概念所在：

1. 不创建对象，而是描述创建对象的方式。
2. 在代码中，对象与服务没有直接联系，而是容器负责将这些联系在一起。

控制权由应用代码中转到了外部容器，控制权的转移，是所谓反转。

好莱坞原则就是IoC（Inversion of Control）或DI（Dependency Injection ）的基础原则。这个原则很像依赖倒置原则，依赖接口，而不是实例，但是这个原则要解决的是怎么把这个实例传入调用类中？你可能把其声明成成员，你可以通过构造函数，你可以通过函数参数。但是 IoC可以让你通过配置文件，一个由Service Container 读取的配置文件来产生实际配置的类。但是程序也有可能变得不易读了，程序的性能也有可能还会下降。

**参考**：

* <http://en.wikipedia.org/wiki/Hollywood_Principle>
* <http://en.wikipedia.org/wiki/Inversion_of_Control>

High Cohesion & Low/Loose coupling & – 高内聚， 低耦合

这个原则是UNIX操作系统设计的经典原则，把模块间的耦合降到最低，而努力让一个模块做到精益求精。

* 内聚：一个模块内各个元素彼此结合的紧密程度
* 耦合：一个软件结构内不同模块之间互连程度的度量

内聚意味着重用和独立，耦合意味着多米诺效应牵一发动全身。

**参考**：

* [http://en.wikipedia.org/wiki/Coupling\_(computer\_science)](http://en.wikipedia.org/wiki/Coupling_%28computer_science%29)
* [http://en.wikipedia.org/wiki/Cohesion\_(computer\_science)](http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29)

Convention over Configuration（CoC）– 惯例优于配置原则

简单点说，就是将一些公认的配置方式和信息作为内部缺省的规则来使用。例如，Hibernate的映射文件，如果约定字段名和类属性一致的话，基本上就可以不要这个配置文件了。你的应用只需要指定不convention的信息即可，从而减少了大量convention而又不得不花时间和精力啰里啰嗦的东东。配置文件很多时候相当的影响开发效率。

Rails 中很少有配置文件（但不是没有，数据库连接就是一个配置文件），Rails 的fans号称期开发效率是 java 开发的 10 倍，估计就是这个原因。Maven也使用了CoC原则，当你执行mvn -compile命令的时候，不需要指源文件放在什么地方，而编译以后的class文件放置在什么地方也没有指定，这就是CoC原则。

**参考**：<http://en.wikipedia.org/wiki/Convention_over_Configuration>

Separation of Concerns (SoC) – 关注点分离

SoC 是计算机科学中最重要的努力目标之一。这个原则，就是在软件开发中，通过各种手段，将问题的各个关注点分开。如果一个问题能分解为独立且较小的问题，就是相对较易解决的。问题太过于复杂，要解决问题需要关注的点太多，而程序员的能力是有限的，不能同时关注于问题的各个方面。正如程序员的记忆力相对于计算机知识来说那么有限一样，程序员解决问题的能力相对于要解决的问题的复杂性也是一样的非常有限。在我们分析问题的时候，如果我们把所有的东西混在一起讨论，那么就只会有一个结果——乱。

我记得在上一家公司有一个项目，讨论就讨论了1年多，项目本来不复杂，但是没有使用SoC，全部的东西混为一谈，再加上一堆程序员注入了各种不同的观点和想法，整个项目一下子就失控了。最后，本来一个1年的项目做了3年。

实现关注点分离的方法主要有两种，一种是标准化，另一种是抽象与包装。标准化就是制定一套标准，让使用者都遵守它，将人们的行为统一起来，这样使用标准的人就不用担心别人会有很多种不同的实现，使自己的程序不能和别人的配合。Java EE就是一个标准的大集合。每个开发者只需要关注于标准本身和他所在做的事情就行了。就像是开发镙丝钉的人只专注于开发镙丝钉就行了，而不用关注镙帽是怎么生产的，反正镙帽和镙丝钉按标来就一定能合得上。不断地把程序的某些部分抽像差包装起来，也是实现关注点分离的好方法。一旦一个函数被抽像出来并实现了，那么使用函数的人就不用关心这个函数是如何实现的，同样的，一旦一个类被抽像并实现了，类的使用者也不用再关注于这个类的内部是如何实现的。诸如组件，分层，面向服务，等等这些概念都是在不同的层次上做抽像和包装，以使得使用者不用关心它的内部实现细节。

说白了还是“高内聚，低耦合”。

**参考**：<http://sulong.me/archives/99>

Design by Contract (DbC) – 契约式设计

DbC的核心思想是对软件系统中的元素之间相互合作以及“责任”与“义务”的比喻。这种比喻从商业活动中“客户”与“供应商”达成“契约”而得来。例如：

* 供应商必须提供某种产品（责任），并且他有权期望客户已经付款（权利）。
* 客户必须付款（责任），并且有权得到产品（权利）。
* 契约双方必须履行那些对所有契约都有效的责任，如法律和规定等。

同样的，如果在程序设计中一个模块提供了某种功能，那么它要：

* 期望所有调用它的客户模块都保证一定的进入条件：这就是模块的先验条件（客户的义务和供应商的权利，这样它就不用去处理不满足先验条件的情况）。
* 保证退出时给出特定的属性：这就是模块的后验条件——（供应商的义务，显然也是客户的权利）。
* 在进入时假定，并在退出时保持一些特定的属性：不变式。

契约就是这些权利和义务的正式形式。我们可以用“三个问题”来总结DbC，并且作为设计者要经常问：

* 它期望的是什么？
* 它要保证的是什么？
* 它要保持的是什么？

根据Bertrand Meyer氏提出的DBC概念的描述，对于类的一个方法，都有一个前提条件以及一个后续条件，前提条件说明方法接受什么样的参数数据等，只有前提条件得到满足时，这个方法才能被调用；同时后续条件用来说明这个方法完成时的状态，如果一个方法的执行会导致这个方法的后续条件不成立，那么这个方法也不应该正常返回。

现在把前提条件以及后续条件应用到继承子类中，子类方法应该满足：

1. 前提条件不强于基类．
2. 后续条件不弱于基类．

换句话说，通过基类的接口调用一个对象时，用户只知道基类前提条件以及后续条件。因此继承类不得要求用户提供比基类方法要求的更强的前提条件，亦即，继承类方法必须接受任何基类方法能接受的任何条件（参数）。同样，继承类必须顺从基类的所有后续条件，亦即，继承类方法的行为和输出不得违反由基类建立起来的任何约束，不能让用户对继承类方法的输出感到困惑。

这样，我们就有了基于契约的LSP，基于契约的LSP是LSP的一种强化。

**参考**：<http://en.wikipedia.org/wiki/Design_by_contract>

Acyclic Dependencies Principle (ADP) – 无环依赖原则

包之间的依赖结构必须是一个直接的无环图形，也就是说，在依赖结构中不允许出现环（循环依赖）。如果包的依赖形成了环状结构，怎么样打破这种循环依赖呢？有2种方法可以打破这种循环依赖关系：第一种方法是创建新的包，如果A、B、C形成环路依赖，那么把这些共同类抽出来放在一个新的包D里。这样就把C依赖A变成了C依赖D以及A依赖D，从而打破了循环依赖关系。第二种方法是使用DIP（依赖倒置原则）和ISP（接口分隔原则）设计原则。

无环依赖原则（ADP）为我们解决包之间的关系耦合问题。在设计模块时，不能有循环依赖。

**参考**：<http://c2.com/cgi/wiki?AcyclicDependenciesPrinciple>

————————————————————————————

上面这些原则可能有些学院派，也可能太为理论，我在这里说的也比较模糊和简单，这里只是给大家一个概貌，如果想要了解更多的东西，大家可以多google一下。

不过这些原则看上去都不难，但是要用好却并不那么容易。要能把这些原则用得好用得精，而不教条，我的经验如下：（我以为这是一个理论到应用的过程）

1. 你可以先粗浅或是表面地知道这些原则。
2. 但不要急着马上就使用。
3. 在工作学习中观察和总结别人或自己的设计。
4. 再回过头来了回顾一下这些原则，相信你会有一些自己的心得。
5. 有适度地去实践一下。
6. Goto第 3步。

我相信可能还会有其实一些原则，欢迎大家提供。

（全文完）

# 面向对象设计七大原则 - sunflower627 - 博客园

http://www.cnblogs.com/sunflower627/p/4718702.html

[**SunnyAmy**](http://www.cnblogs.com/sunflower627/)

[面向对象设计七大原则](http://www.cnblogs.com/sunflower627/p/4718702.html)

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

每一个类应该专注于做一件事情。

**2. 里氏替换原则（Liskov Substitution Principle）**

超类存在的地方，子类是可以替换的。

**3. 依赖倒置原则（Dependence Inversion Principle）**

实现尽量依赖抽象，不依赖具体实现。

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

应当为客户端提供尽可能小的单独的接口，而不是提供大的总的接口。

**5. 迪米特法则（Law Of Demeter）**

又叫最少知识原则，一个软件实体应当尽可能少的与其他实体发生相互作用。

**6. 开闭原则（Open Close Principle）**

面向扩展开放，面向修改关闭。

**7. 组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）**

尽量使用合成/聚合达到复用，尽量少用继承。原则： 一个类中有另一个类的对象。

**细则**

**单一职责原则（Single Responsibility Principle）**

因为：

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；提高类的可读性，提高系统的可维护性；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，都适用单一职责原则。

所以：

从大局上看Android中的Paint和Canvas等类都遵守单一职责原则，Paint和Canvas各司其职。

**里氏替换原则（Liskov Substitution Principle）**

因为：

里氏替换原则告诉我们，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。里氏替换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

所以：

使用里氏替换原则时需要注意，子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。尽量把父类设计为抽象类或者接口，让子类继承父类或实现父接口，并实现在父类中声明的方法，运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类来实现。

从大局看Java的多态就属于这个原则。

**依赖倒置原则（Dependence Inversion Principle）**

因为：

具体依赖抽象，上层依赖下层。假设B是较A低的模块，但B需要使用到A的功能，这个时候，B不应当直接使用A中的具体类；而应当由B定义一抽象接口，并由A来实现这个抽象接口，B只使用这个抽象接口；这样就达到了依赖倒置的目的，B也解除了对A的依赖，反过来是A依赖于B定义的抽象接口。通过上层模块难以避免依赖下层模块，假如B也直接依赖A的实现，那么就可能造成循环依赖。

所以：

采用依赖倒置原则可以减少类间的耦合性，提高系统的稳定性，减少并行开发引起的风险，提高代码的可读性和可维护性。

从大局看Java的多态就属于这个原则。

**接口隔离原则（Interface Segregation Principle）**

因为：

提供尽可能小的单独接口，而不要提供大的总接口。暴露行为让后面的实现类知道的越少越好。譬如类ProgramMonkey通过接口CodeInterface依赖类CodeC，类ProgramMaster通过接口CodeInterface依赖类CodeAndroid，如果接口CodeInterface对于类ProgramMonkey和类CodeC来说不是最小接口，则类CodeC和类CodeAndroid必须去实现他们不需要的方法。将臃肿的接口CodeInterface拆分为独立的几个接口，类ProgramMonkey和类ProgramMaster分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

所以：

建立单一接口，不要建立庞大的接口，尽量细化接口，接口中的方法尽量少。也就是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。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约定，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。

从大局来说Java的接口可以实现多继承就是接口隔离原则的基础保障。

**迪米特法则（Law Of Demeter）**

因为：

类与类之间的关系越密切，耦合度也就越来越大，只有尽量降低类与类之间的耦合才符合设计模式；对于被依赖的类来说，无论逻辑多复杂都要尽量封装在类的内部；每个对象都会与其他对象有耦合关系，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耦合依赖，而出现在局部变量中的类则不是直接耦合依赖，也就是说，不是直接耦合依赖的类最好不要作为局部变量的形式出现在类的内部。

所以：

一个对象对另一个对象知道的越少越好，即一个软件实体应当尽可能少的与其他实体发生相互作用，在一个类里能少用多少其他类就少用多少，尤其是局部变量的依赖类，能省略尽量省略。同时如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一方法的话，可以通过第三者转发这个调用。

从大局来说Android App开发中的多Fragment与依赖的Activity间交互通信遵守了这一法则。

**开闭原则（Open Close Principle）**

因为：

开放封闭原则主要体现在对扩展开放、对修改封闭，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。软件需求总是变化的，世界上没有一个软件的是不变的，因此对软件设计人员来说，必须在不需要对原有系统进行修改的情况下，实现灵活的系统扩展。

所以：

可以通过Template Method模式和Strategy模式进行重构，实现对修改封闭，对扩展开放的设计思路。   
封装变化，是实现开放封闭原则的重要手段，对于经常发生变化的状态，一般将其封装为一个抽象，拒绝滥用抽象，只将经常变化的部分进行抽象。

**组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）**

因为：

其实整个设计模式就是在讲如何类与类之间的组合/聚合。在一个新的对象里面通过关联关系（包括组合关系和聚合关系）使用一些已有的对象，使之成为新对象的一部分，新对象通过委派调用已有对象的方法达到复用其已有功能的目的。也就是，要尽量使用类的合成复用，尽量不要使用继承。

如果为了复用，便使用继承的方式将两个不相干的类联系在一起，违反里氏代换原则，哪是生搬硬套，忽略了继承了缺点。继承复用破坏数据封装性，将基类的实现细节全部暴露给了派生类，基类的内部细节常常对派生类是透明的，白箱复用；虽然简单，但不安全，不能在程序的运行过程中随便改变；基类的实现发生了改变，派生类的实现也不得不改变；从基类继承而来的派生类是静态的，不可能在运行时间内发生改变，因此没有足够的灵活性。

所以：

组合/聚合复用原则可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

**总结**

至此，七大基本原则介绍完毕，很空洞，需要联系Java与android代码去仔细体会琢磨。

# JAVA设计模式学习2——面向对象设计原则 - alaric's blog - ITeye技术网站

http://alaric.iteye.com/blog/1722198

软件设计的核心是提高软件的可复用性和可维护性。通常一个软件之所以可复用性和可扩展性差的原因在于设计过于僵硬，过于脆弱，复用率低，粘度过高等原因导致的，这时候需要想办法提高可扩展性，灵活性和可插入性，从而提高软件的可复用性和可维护性。一般可维护性和可复用性不能同时能达到目的，只有遵循一定的设计原则，设计出来的系统才能同时满足可复用性和可维护性。面向对象设计原则主要有如下几条：

1、“开闭”原则（Open-Closed Principle）简称OCP，讲的是一个软件实体应该对扩展开放对修改关闭。

2、里氏代换原则（Liskov Substitution Principle）简称LSP，讲的是任何父类出现的地方都可以被子类代替。

3、依赖倒转原则（Dependency INversion Principle）简称DIP，讲的是要依赖于抽象不要依赖于实现。

4、接口隔离原则（Interface Segregation Principle）简称ISP，讲的是为客户端提供尽可能小的单独的接口，而不是提供大的总接口。

5、组合/聚合服用原则（Composition/Aggregation Principle）简称CARP，讲的是要尽可能使用组合，聚合来达到复用目的而不是利用继承。

6、迪米特法则（Law of Demeter）简称LoD，讲的是一个软件实体应当与尽可能少的其他软件实体发生相互作用。

为什么要在讲设计模式前讲设计原则，是因为设计模式是面向对象设计原则的具体指导，所以有了理论和设计指导我们就可以进入设计模式学习了，设计模式大家常说的有23中，其实现实中要多的多，大概分为三类：创建模式，结构模式和行为模式。

**设计模式系列目录：**

[JAVA设计模式学习1——统一建模语言UML](http://alaric.iteye.com/blog/1722197)

[JAVA设计模式学习2——面向对象设计原则](http://alaric.iteye.com/blog/1722198)

[JAVA设计模式学习3——设计模式概述](http://alaric.iteye.com/blog/1722199)

[JAVA设计模式学习4——简单工厂模式](http://alaric.iteye.com/blog/1908619)

[JAVA设计模式学习5——工厂方法模式](http://alaric.iteye.com/blog/1908876)

[JAVA设计模式学习6——抽象工厂模式](http://alaric.iteye.com/blog/1908902)

[JAVA设计模式学习7——单例模式](http://alaric.iteye.com/blog/1909007)

[JAVA设计模式学习8——建造模式](http://alaric.iteye.com/blog/1909867)

[JAVA设计模式学习9——原型模式](http://alaric.iteye.com/blog/1909960)

[JAVA设计模式学习10——组合模式](http://alaric.iteye.com/blog/1910919)

[JAVA设计模式学习11——适配器模式](http://alaric.iteye.com/blog/1911161)

[JAVA设计模式学习12——装饰器模式](http://alaric.iteye.com/blog/1911980)

[JAVA设计模式学习13——代理模式](http://alaric.iteye.com/blog/1913048)

[JAVA设计模式学习14——享元模式](http://alaric.iteye.com/blog/1914498)

[JAVA设计模式学习15——门面模式](http://alaric.iteye.com/blog/1915854)

[JAVA设计模式学习16——桥梁模式](http://alaric.iteye.com/blog/1918381)

[JAVA设计模式学习17——策略模式](http://alaric.iteye.com/blog/1920714)

[JAVA设计模式学习18——模版方法模式](http://alaric.iteye.com/blog/1922820)

[JAVA设计模式学习19——观察者模式](http://alaric.iteye.com/blog/1924169)

[JAVA设计模式学习20——责任链模式](http://alaric.iteye.com/blog/1926447)

[JAVA设计模式学习21——命令模式](http://alaric.iteye.com/blog/1927894)

[JAVA设计模式学习22——备忘录模式](http://alaric.iteye.com/blog/1931253)

[JAVA设计模式学习23——状态模式](http://alaric.iteye.com/blog/1938400)

[JAVA设计模式学习24——访问者模式](http://alaric.iteye.com/blog/1942517)

[JAVA设计模式学习25——迭代器模式](http://alaric.iteye.com/blog/1978409)