**[面向对象设计的SOLID原则](http://www.cnblogs.com/shanyou/archive/2009/09/21/1570716.html)**

正如牛顿三大定律在经典力学中的位置一样，“开-闭”原则（Open-Closed Principle）是面向对象的可

复用设计（Object Oriented Design或OOD）的基石。其他设计原则（里氏代换原则、依赖倒转原则、合成/聚合复用原则、迪米特法则、接口隔离原则）是实现“开-闭”原则的手段和工具。

S.O.L.I.D是面向对象设计和编程(OOD&OOP)中几个重要编码原则(Programming Principle)的首字母缩写。

|  |  |  |
| --- | --- | --- |
| **SRP** | [The Single Responsibility Principle](http://www.objectmentor.com/resources/articles/srp.pdf) | 单一责任原则 |
| **OCP** | [The Open Closed Principle](http://www.objectmentor.com/resources/articles/ocp.pdf) | 开放封闭原则 |
| **LSP** | [The Liskov Substitution Principle](http://www.objectmentor.com/resources/articles/lsp.pdf) | 里氏替换原则 |
| **DIP** | [The Dependency Inversion Principle](http://www.objectmentor.com/resources/articles/dip.pdf) | 依赖倒置原则 |
| **ISP** | [The Interface Segregation Principle](http://www.objectmentor.com/resources/articles/isp.pdf) | 接口分离原则 |

**单一责任原则：**   
当需要修改某个类的时候原因有且只有一个（THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE）。换句话说就是让一个类只做一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。   
 [](http://images.cnblogs.com/cnblogs_com/shanyou/WindowsLiveWriter/SOLID_6402/SingleResponsibilityPrinciple2_71060858_2.jpg)

从面向对象角度解释这个原则为："引起类变化的因素永远不要多于一个。" 或者说 "一个类有且仅有一个职责"。这似乎不太好理解，特别是"引起类变化的因素永远不要多于一个。"这句话更是有点虚，让人有点摸不着头脑。

      我们通常都说“低耦合，高内聚”。在我看来，这里的"单一职责"就是我们通常所说的“高内聚”，即一个类只完成它应该完成的职责，不能推诿责任，也不可越殂代疱，不能成为无所不能的上帝类。如果你的团队中实施宽松的“代码集体所有权”，在编码的过程中出现许多人同时修改（维护）同一个类的现象，而且成员之间的沟通不够及时，主动和畅通的话，那么时间一长，就很可能出现“承担过多职责”的上帝类。这时，提炼基类/接口和提炼类重构将能帮助我们消除或减轻这种设计臭味。

在所有的SOLID 原则中，这是大多数开发人员感到最能完全理解的一条。严格来说，这也可能是违反最频繁的一条原则了。

**开放封闭原则**

Software entities should be open for extension, but closed for modification.

软件实体应该是可扩展，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。这个原则是诸多面向对象编程原则中最抽象、最难理解的一个。

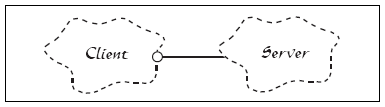
[](http://images.cnblogs.com/cnblogs_com/shanyou/WindowsLiveWriter/SOLID_6402/OpenClosedPrinciple2_2C596E17_2.jpg)

从面向对象设计角度看，这个原则可以这么理解："软件实体(类,模块,函数等等)应当对扩展开放，对修改闭合。" 通俗来讲，它意味着你（或者类的客户）应当能在不修改一个类的前提下扩展这个类的行为。在OOD里，对扩展开放意味着类或模块的行为能够改变，在需求变化时我们能以新的，不同的方式让模块改变，或者在新的应用中满足需求。

        也就是说，对扩展是开放的，而对修改是封闭的。我们通常都说：向系统中增加功能时应该只是添加新代码，而应该尽量少的修改原代码。在我看来，这就是遵循开放封闭原则所能带来的效果。曾经在网上看到过这样一句话“哪里变化，封装哪里”。这其实就是说，我们要将系统中可能变化的地方封装起来，即对修改封闭。同时，为了应对系统需求（功能）的扩展，需要抽象！

       这里抽象是关键。《设计模式》中的state模式和strategy模式是这个原则的最好体现。

举一个例子：



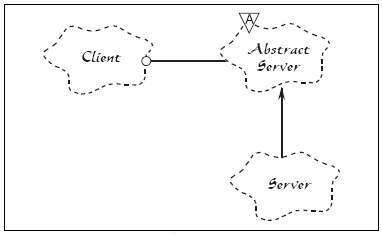
违反了开放封闭原则的类结构图。

客户端代码直接面向服务器端的具体实现编程，缺乏灵活性。这样如果服务器因为某些原因被其他服务器替换了，那么客户端调用服务器的代码也必须做相应的修改或替换。这其实就是”面向实现编程“的设计臭味！

那么，如何修改才能得到正确灵活的设计？

答案是：抽象！为服务器端的代码（类型）抽象出一个抽象基类（定义一组完成服务职责的最小接口）。

下面是正确的设计：



遵循开放封闭原则的类结构图。

       基本上，你抽象的东西是你系统的核心内容，如果你抽象得好，很可能增加一个新的服务器类型（扩展）只需要添加新类型（继承自AbstractServer即可）。因此代码要尽可能以抽象(这里的AbstractServer)为依据，这会允许你扩展抽象事物，定义一个新的实现而不需要修改任何客户端代码。即”面向接口编程，不要面向实现编程“！

于是，问题变为：“那么，如果我需要修改这个基类的工作方式，那应当怎么做呢？”OCP 的另一部分中给出这一答案；基类应当开放，可进行扩充。在这里，扩充是指创建一个由此基类继承而来的派生类，它可以扩充或重载基类功能，以提供使用者所需要的特定功能。这样，使用者就能使用类的修改版本，而不会影响到类的其他使用者。使用者还可以在将来更轻松地使用基类的升级版本，因为他们不用担心丢失自己的修改内容。

所以解决问题的关键在于抽象化，抽象化是面向对象设计的第一个核心本质

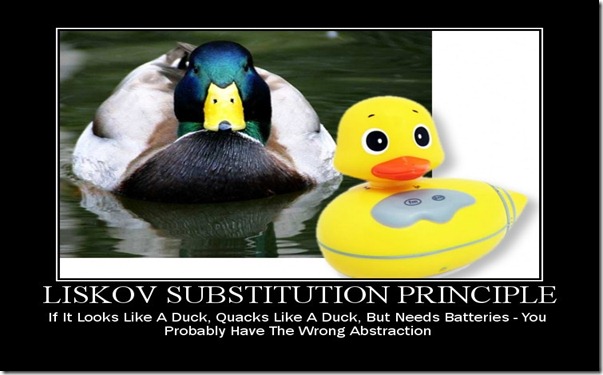
**里氏替换原则**

[里氏代换原则](http://baike.baidu.com/view/1638361.htm)（Liskov Substitution Principle）

[里氏代换原则](http://baike.baidu.com/view/1638361.htm)是由麻省理工学院（MIT）计算机科学实验室的Liskov女士，在1987年的OOPSLA大会上发表的一篇文章《Data Abstraction and Hierarchy》里面提出来的，主要阐述了有关继承的一些原则，也就是什么时候应该使用继承，什么时候不应该使用继承，以及其中的蕴涵的原理。2002年，软件工程大师Robert C. Martin，出版了一本《Agile Software Development Principles Patterns and Practices》，在文中他把[里氏代换原则](http://baike.baidu.com/view/1638361.htm" \t "_blank)最终简化为一句话："Subtypes must be substitutable for their base types"，也就是说，子类必须能够替换成它们的基类。

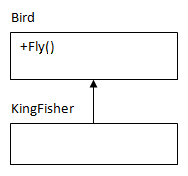
我们把[里氏代换原则](http://baike.baidu.com/view/1638361.htm" \t "_blank)解释得更完整一些：在一个软件系统中，子类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。子类也能够在基类的基础上增加新的行为。

[里氏代换原则](http://baike.baidu.com/view/1638361.htm" \t "_blank)是对开闭原则的补充，它讲的是基类和子类的关系。只有当这种关系存在时，里氏代换关系才存在。

当一个子类的实例应该能够替换任何其超类的实例时，它们之间才具有is-A关系   
[](http://images.cnblogs.com/cnblogs_com/shanyou/WindowsLiveWriter/SOLID_6402/LiskovSubtitutionPrinciple_52BB5162_2.jpg)

 Liskov's 替换原则意思是："子类型必须能够替换它们的基类型。"或者换个说法："使用基类引用的地方必须能使用继承类的对象而不必知道它。" 这个原则正是保证继承能够被正确使用的前提。通常我们都说，“优先使用组合（委托）而不是继承”或者说“只有在确定是 is-a 的关系时才能使用继承”，因为继承经常导致”紧耦合“的设计。  
    在基本的面向对象原则里，"继承"通常是"is a"的关系。如果"Developer" 是一个"SoftwareProfessional",那么"Developer"类应当继承"SoftwareProfessional"类。在类设计中"Is a"关系非常重要，但它容易冲昏头脑，导致使用错误的继承造成错误设计。

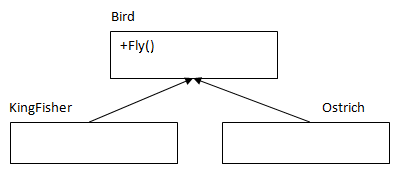
看一个最最经典的例子：



遵循Liskov替换原则的类结构图。

注：这里，KingFisher（翠鸟）类扩展了Bird基类，并继承了Fly()方法，这没有问题。

但是下面这个类结构图就存在设计上的问题：



违反Liskov替换原则的类结构图。

Ostrich(鸵鸟)是一种鸟，这毋庸置疑，并从Bird类继承，这从概念上说没有问题。但是鸵鸟它能飞吗？不能，那么这个设计就违反了LSP。因为在使用Bird的地方不一定能用Ostrich代替。所以，即使在现实中看起来没问题，在类设计中，Ostrich不应该从Bird类继承，这里应该从Bird中分离一个不会飞的类NoFlyBrid，Ostrich应该继承这个不会飞的鸟类NoFlyBrid。

为什么LSP如此重要？

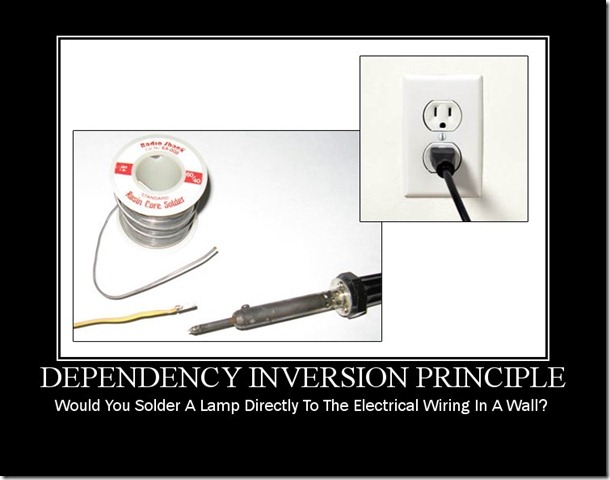
如果没有LSP，类继承就会混乱；如果子类作为一个参数传递给方法，将会出现未知行为；

如果没有LSP，适用与基类的单元测试将不能成功用于测试子类；

Composite模式，Proxy模式，Strategy模式 实现了Liskov原则

LSP讲的是基类和子类的关系。只有当这种关系存在时，里氏代换关系才存在。如果两个具体的类A，B之间的关系违反了LSP的设计，(假设是从B到A的继承关系)那么根据具体的情况可以在下面的两种重构方案中选择一种。 创建一个新的抽象类C，作为两个具体类的超类，将A，B的共同行为移动到C中来解决问题。 从B到A的继承关系改为委派关系。

在进行设计的时候，我们尽量从抽象类继承，而不是从具体类继承。如果从继承等级树来看，所有[叶子节点](http://baike.baidu.com/view/544292.htm" \t "_blank)应当是具体类，而所有的树枝节点应当是抽象类或者接口。当然这个只是一个一般性的指导原则，使用的时候还要具体情况具体分析。

**依赖倒置原则**1. 高层模块不应该依赖于低层模块，二者都应该依赖于抽象   
2. 抽象不应该依赖于细节，细节应该依赖于抽象   
[](http://images.cnblogs.com/cnblogs_com/shanyou/WindowsLiveWriter/SOLID_6402/DependencyInversionPrinciple_0278F9E2_2.jpg)

在完美世界里，应用程序的组件之间没有耦合关系或绑定关系。开发人员也能够改变

自己希望改变的任何东西，而不需要担心在应用程序的其他地方出现缺陷，或者“不希望

存在的负面影响”。令人悲伤的是，我们并不是生活在完美世界里。因此，组件需要相互

绑定在一起，或者在某一点耦合，以构成实际应用程序。

依赖倒置原则(Dependency Inversion Principle，DIP)规定：代码应当取决于抽象概念，

而不是具体实现；这些抽象不应当依赖于细节；而细节应当依赖于抽象。类可能依赖于其

他类来执行其工作。但是，它们不应当依赖于该类的特定具体实现，而应当是它的抽象。也就是说，

服务不知道(或不关心)正在使用哪个具体的数据访问组件——只有它的抽象或代

码契约(或接口)支持那些用于保存和检索员工所需要的方法。

显然，这一概念会大大提高系统的灵活性。如果类只关心它们用于支持特定契约而不

是特定类型的组件，就可以快速而轻松地修改这些低级服务的功能，同时最大限度地降低

对系统其余部分的影响

**接口分离原则**   
不能强迫用户去依赖那些他们不使用的接口。换句话说，使用多个专门的接口比使用单一的总接口总要好。

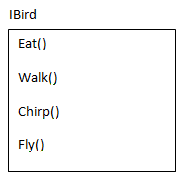
[](http://images.cnblogs.com/cnblogs_com/shanyou/WindowsLiveWriter/SOLID_6402/InterfaceSegregationPrinciple_60216468_2.jpg)

这个原则的意思是"客户端不应该被迫依赖于它们不用的接口。" 也就是说，一个接口或者类应该拥有尽可能少的行为（那么，什么叫尽可能少？就是少到恰好能完成它自身的职责），这也是保证“软件系统模块的粒度尽可能少，以达到高度可重用的目的。

     接口包含太多的方法会降低其可用性，像这种包含了无用方法的"胖接口"会增加类之间的耦合。如果一个类想实现该接口,那么它需要实现所有的方法,尽管有些对它来说可能完全没用，所以这样做会在系统中引入不必要的复杂度，降低代码的可维护性或鲁棒性。

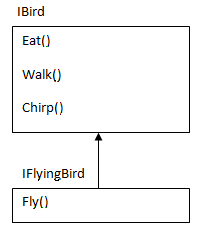
     接口分离原则确保实现的接口有它们共同的职责,它们是明确的,易理解的,可复用的.

下面这个例子充分的说明了”接口应该仅包含必要的方法，而不该包含其它的“。如果一个接口包含了过多的方法，应该通过分离接口将其拆分。



这是一个违反接口分离原则的胖接口。

注意到IBird接口包含很多鸟类的行为,包括Fly()行为.现在如果一个Bird类(如Ostrich)实现了这个接口，那么它需要实现不必要的Fly()行为(Ostrich不会飞)。因此，这个"胖接口"应该拆分成两个不同的接口，IBird和IFlyingBird, 而IFlyingBird继承自IBird。如下图所示：



这样的话，重用将变得非常灵活：如果一种鸟不会飞(如Ostrich)，那它实现IBird接口。如果一种鸟会飞(如KingFisher)，那么它实现IFlyingBird。

因此，如果我们想要获得可重用的方案，就应当遵循接口分离原则，把接口定义成仅包含必要的部分，以便在任何需要该接口功能的地方复用这个接口。

**六、合成复用原则（CRP）（优先利用组合而非继承）**

合成复用原则（Composite Reuse Principle,CRP），即优先使用委托而不是继承来重用已用功能（代码）。循序这一原则通常也是避免触犯里氏替换原则所要求的。

通过合成/聚合的优缺点

优点：

1) 新对象存取成分对象的唯一方法是通过成分对象的接口。

2) 这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的。

3) 这种复用支持包装。

4) 这种复用所需的依赖较少。

5) 每一个新的类可以将焦点集中在一个任务上。

6) 这种复用可以在运行时间内动态进行，新对象可以动态的引用与成分对象类型相同的对象。

7) 作为复用手段可以应用到几乎任何环境中去。

缺点:就是系统中会有较多的对象需要管理。

通过继承来进行复用的优缺点

优点： 新的实现较为容易，因为超类的大部分功能可以通过继承的关系自动进入子类。修改和扩展继承而来的实现较为容易。

缺点： 继承复用破坏包装，因为继承将超类的实现细节暴露给子类。由于超类的内部细节常常是对于子类透明的，

所以这种复用是透明的复用，又称“白箱”复用。如果超类发生改变，那么子类的实现也不得不发生改变。

从超类继承而来的实现是静态的，不可能在运行时间内发生改变，没有足够的灵活性。继承只能在有限的环境中使用。

**七、迪米特法则（LoD / LKP）**

迪米特法则（Law Of Demeter）又称最小知识原则（Least Knowledge Principle, LKP）。意思是一个对象应当对其它对象有尽量好的了解，即应该保持对象间有尽量少的相互作用是，使得对象（类）具有好的独立性，可测试性，也就易于维护。

关于“迪米特法则”的其它表述还有：只与你的朋友们通信，不要与“陌生人”说话。

设计模式中的Facade模式和Mediator模式就是使用了这一原则，降低模块间的耦合。

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。

**八、封装变化**

在软件领域中唯一不变的就是“Change”，因此封装你认为或猜测未来将发生变化的代码。OOPS设计模式的优点在于易于测试和维护封装的代码。如果你使用Java编码，可以默认私有化变量和方法，并逐步增加访问权限，比如从private到protected和not public。有几种Java设计模式也使用封装，比如Factory设计模式是封装“对象创建”，其灵活性使得之后引进新代码不会对现有的代码造成影响。

**九、DRY（Don't repeat yourself）原则**

即不要写重复的代码，而是用“abstraction”类来抽象公有的东西。如果你需要多次用到一个硬编码值，那么可以设为公共常量；假设你要在两个以上的地方使用一个代码块，那么可以将它设为一个独立的方法。设计的原则就是要易于维护，使用公共代码来实现两个不同的功能，其实就是近似地把这两个功能永远捆绑到了一起。

**九、委托原则**

该原则最典型的例子是Java中的equals() 和hashCode() 方法。为了平等地比较两个对象，我们用类本身而不是客户端类来做比较。这个设计原则的好处是没有重复的代码，而且很容易对其进行修改。