链滴

设计原则

作者：DongXiaokai0819
原文链接：https://ld246.com/article/1572699895477
来源网站：链滴

<h2 id="设计原则">设计原则</h2>
本文章只是介绍与讲解各种设计原则的基本概念，以便以后复习使用。具体的原则在代码中的使
我打算在以后整理各种设计模式的时候，再讲讲。
<h3 id="可维护性与可复用性">可维护性与可复用性</h3>
<h4 id="具有可维护性的设计目标">具有可维护性的设计目标</h4>
一个好的系统设计应该有如下的性质：

可扩展性：新的性能可以很容易的加入现有的系统中，而又不会对该系统的其它模块造成影响。 <
li>
灵活性：代码修改不会波及其他的模块。
可插入性：可以很容易的将一个类用另一个有同样接口的类代替。

<h4 id="系统的可复用性">系统的可复用性</h4>
系统复用的优点：

复用可以提高生产效率
复用可以提高软件质量
复用可以改善系统的可维护性。

<h5 id="传统复用的方式">传统复用的方式</h5>

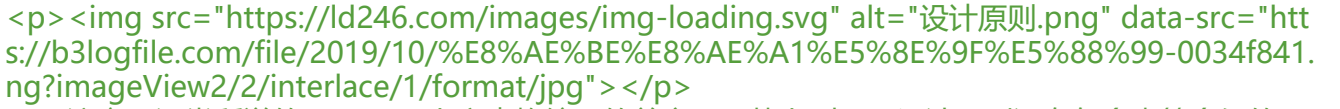
代码的剪贴复用，这种方式在具体实施时，要冒着产生错误的风险。
算法的复用
各种数据结构的复用

<h5 id="面向对象设计的复用">面向对象设计的复用</h5>
面向对象的语言中，数据的抽象化、继承、封装和多态性等语言特性使得一个系统可在更高的层
上提供可复用性。

数据的抽象化和继承关系使得概念和定义可以复用
多态性使得实现和应用可以复用
抽象化和封装可以保持和促进系统的可维护性

复用的焦点，就不再集中在函数和算法等具体实现细节上，而是几种在最重要的含有宏观商业逻
的抽象层次上。这并不是说具体实现细节的复用不再重要。
<h4 id="可维护性复用">可维护性复用</h4>
在面向对象的设计中，可维护性复用是以设计原则和设计模式为基础的。

开闭原则、里氏替换原则和组合/聚合复用原则 可以在提高系统可复用性的同时，提高系统的可
展性。
开闭原则、迪米特法则、接口隔离原则 可以在提高系统可复用性的同时，提高系统的灵活性。 </l
>
开闭原则、里氏替换原则、组合/聚合原则、依赖倒置原则 可以在提高系统可复用性的同时， 提
系统的可插入性。

<h3 id="设计原则-">设计原则</h3>


注意，通常所说的 SOLID（上方表格缩写的首字母，从上到下）设计原则没有包含本篇介绍的
米特法则，而只有其他五项。
<h4 id="开闭原则">开闭原则</h4>
软件，如类、模块和函数方法等，应该 <code>对扩展开放，对修改关闭</code>
其它设计原则都是开闭原则的手段和工具，是附属于开闭原则的

解读：

用抽象构建框架，用实现扩展细节。不以改动原有类的方式来实现新需求，而是应该以实现事先象出来的接口（或具体类继承抽象类）的方式来实现。
这个抽象层预见了所有可能的扩展，在任何情况下都不改变。
应考虑设计中什么可能会发生变化。（注：考虑的不是什么会导致设计改变，而是允许什么发生化而不让这一变化导致重新设计）

优点：开闭原则的优点在于可以在不改动原有代码的前提下给程序扩展功能。增加了程序的可扩性，同时也降低了程序的维护成本。
<h4 id="单一职责原则">单一职责原则</h4>
就是说功能要单一，一个对象应该只包含单一的职责，并且该职责被完整封装在一个类中。或者说，就一个类而言，应该只有一个引起它变化的原因。

一个类（或者模块、方法）承担的职责越多，它被复用的可能性越小。
单一职责原则是实现高内聚、低耦合的指导方针。

类的职责主要包括俩个方面：

数据职责，通过属性来体现
行为职责，通过方法来体现

解读：
类职责的变化往往就是导致类变化的原因。也就是说如果一个类具有多种职责，就会有多种导致这个变化的原因，从而导致这个类的维护变得困难。往往在软件开发中，随着需求的不断增加，可能会给来的类添加一些本来不属于它的一些职责，从而违反了单一职责原则。如果我们发现当前类的职责不仅有一个，就应该将本来不属于该类真正的职责分离出去。不仅仅是类，函数也要遵循单一职责原则即一个函数制作一件事情。如果发现一个函数里面有不同的任务，则需要将不同的任务以另一个函数形式分离出去。
优点：如果类与方法的职责划分的很清晰，不但可以提高代码的可读性，更实际性地更降低了程出错的风险，因为清晰的代码会让 bug 无处藏身，也有利于 bug 的追踪，也就是降低了程序的维护本。
<h4 id="里氏替换原则">里氏替换原则</h4>
就是说在使用基类的的地方可以任意使用其子类，能保证子类完美替换基类。

里氏替换原则是继承复用的基础
反过来的替换则不成立，即如果一个软件使用的是一个子类，那么它不一定适用于父类。

解读：

只要父类能出现的地方子类就能出现。反之，父类则未必能胜任。
在继承体系中，子类中可以增加自己特有的方法，也可以实现父类的抽象方法，但是不能重写父的非抽象方法，否则该继承关系就不是一个正确的继承关系。

优点：增强程序的健壮性，即使增加了子类，原有的子类还可以继续运行。
<h5 id="如果违反了里氏替换原则怎么办-">如果违反了里氏替换原则怎么办?</h5>
如果有俩个具有继承关系的类 A 和 B 违反了里氏替换原则，就要取消继承关系，可采用以下方：

创建一个新的抽象类 C，作为俩个具体类的父类，讲 A 和 B 的共同行为移动到 C 中，从而解决 A 和 B 行为不完全一致的问题
将 A 和 B 的继承关系改写为组合/聚合关系

<h4 id="依赖倒置原则">依赖倒置原则</h4>
依赖倒置原则是面向对象设计的核心原则
抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。</
>

高层模块不应该依赖底层模块，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

解读：
高层模块就是调用端，低层模块就是具体实现类。抽象就是指接口或抽象类。细节就是实现类。
要达到上述要求，一个具体类应当只实现抽象类或 Java 接口中声明过的方法，而不应该给出多余的
法
通俗点来说就是依赖倒置原则的本质就是通过抽象（接口或者抽象类）使得各类或者模块的实现彼此
立，互不影响，实现类之间不发生直接的依赖关系，实现模块间的松耦合。
优点：可减少类之间的耦合性，提高系统的可维护性，减少并行开发引起的风险，提高代码的可
性。
<h5 id="耦合关系">耦合关系</h5>
在面向对象的系统里，俩个类之间有零耦合、具体耦合和抽象耦合三种类型的耦合关系。

零耦合：指俩个类之间没有耦合关系
具体耦合：指在俩个具体类之间的耦合，一个类对另一个具体类的直接引用
抽象耦合：指一个具体类和一个抽象类/Java 接口之间的耦合，有最大的灵活性

依赖倒置原则要求客户端依赖于抽象耦合
<h4 id="接口隔离原则">接口隔离原则</h4>
准确而恰当地划分角色以及角色所对应的接口，是面向对象设计的一个重要的组成部
。
在这里有俩种解释：

客户端不应该依赖哪些它不需要的接口。
另一种就是，一旦一个接口太大、太笨重，则需要将它分割成一些更细小的接口，使用该接口的
户端仅需要知道与之相关的方法即可。

<blockquote>
接口隔离原则是指使用多个专门的接口比使用单一的总接口要好。接口仅仅提供客户端需要的行
，即所需的方法，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不
提供大的总接口。
从客户端的角度看，接口隔离原则是指一个类对另一个的依赖性应当建立在最小的接口上。
</blockquote>
解读：
使用接口隔离原则拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，
在满足高内聚的前提下，接口中的方法越少越好。
总之，应当将多个不同的角色交给不同的接口，而不应当都交给同一个接口。
不要将看上去差不多的甚至是没有关系的接口合并，这样会形成国与臃肿的大接口，称为接口的污染

优点：避免同一个接口里面包含不同类职责的方法，接口责任划分更加明确，符合高内聚低耦合
思想。
<h4 id="迪米特法则">迪米特法则</h4>
核心观念：类间解耦
迪米特法则又称为最少知识原则，是指一个对象应当对其他对象有尽可能少的了解。
迪米特法则可以表述为只与你直接的朋友通信；不要跟“陌生人”说话
在迪米特法则中，对于一个对象，其朋友包括以下几类：

当前对象本身（this）

以参数形式传入到当前对象方法中的对象
当前对象的成员对象
如果当前对象的成员成员对象是一个集合，那么集合中的元素也都是朋友
当前对象所创建的对象

任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”
<blockquote>
简单的说，迪米特法则就是指一个软件实体应当尽可能少地与其他实体发生相互作用。这样的话当一个模块修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制它要求限制软件实体之间通信的宽度和深度。
</blockquote>
迪米特法则可分为侠义法则和广义法则：
<h5 id="侠义迪米特法则">侠义迪米特法则</h5>
指如果俩个类不是必须要彼此直接通信，那么这俩个类就不应当发生直接的相互作用。
如果其中的一个类需要调用另一个类（陌生人）的某一个方法，可通过第三者（朋友）转发这个调用
但第三者需要额外增加方法。
优：可降低类之间的耦合
劣：会在系统中造出大量的小方法，散落在系统的各个角落
<h5 id="广义迪米特法则">广义迪米特法则</h5>
指对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。一个系统的规越大，信息的隐藏就越重要。
一个设计得好的模块应该就自己的内部数据和与实现有关的细节隐藏起来，并提供给外界的 API 和自的实现分隔开。
这样，模块与模块之间只通过彼此的 API 相互通信，而不理会模块内部的工作细节。这就是面向对象封装特性。
<h5 id="注意">注意</h5>
将迪米特法则运用到系统设计，特别是类的设计时，要注意以下几点：

在类的划分上，应当使创建的类之间的耦合为弱耦合，有利于复用。一个弱耦合中的类被修改不对有关系的类造成影响。
在类的设计上，应尽量将一个类设计成不可变类。
在类的设计上，应尽量降低一个类的访问权限。
在类的设计上，应尽量降低成员的访问权限。

<h4 id="组合-聚合复用原则">组合/聚合复用原则</h4>
要尽量使用组合/聚合，而相应的减少继承的使用
首先介绍以下组合与聚合的概念

组合：是一种强的拥有关系，体现了严格的部分和整体的关系，部分和整体的生命周期一样。</l
>
聚合：表示一种弱的拥有关系或者整体与部分的关系，体现的是 A 对象可以包含 B 对象，但 B 象不是 A 对象的一部分。

<blockquote>
组合/聚合复用原则会使类的继承层次保持较小的规模，避免成为不可控制的庞然大物。
</blockquote>
<h5 id="组合-聚合复用-与-继承复用">组合/聚合复用 与 继承复用</h5>
<h6 id="继承复用">继承复用</h6>
里氏替换原则是继承复用的基础。

优点：父类的大部分功能可以通过继承关系自动进入子类，所以新的实现比较容易，修改或扩展承而来的实现也比较容易
缺点：

从父类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性
典型的白箱复用，父类的内部细节对子类是透明的，继承将父类的实现细节暴露给子类，继承破包装
子类与父类之间有紧密的依赖关系

<h6 id="组合-聚合复用">组合/聚合复用</h6>
组合或者聚合是将已有的对象纳入新对象中，使之成为新对象的一部分，因此新对象可以调用已对象的功能。

优点：

耦合度相对较低
可以在运行时动态进行
黑箱复用，已有对象的内部细节对新对象不可见
作为复用的手段，几乎可以应用到任何环境中

缺点：使用组合/聚合复用的话，需要管理较多的对象

<h6 id="Has-A-与-Is-A">Has-A 与 Is-A</h6>
Has-A ： 表示某一个角色具有某一项责任，代表一个类是另一个类的一个角色，而不是另一个类特殊种类。组合/聚合复用就是 Has-A。
Is-A ： 是严格分类学意义上的定义，意思是一个类是另一个类的一种。继承复用是 Is-A 关系。
<blockquote>
根据里氏替换原则，如果俩个类的关系是 Has-A 关系而不是 Is-A 关系，那么这俩个类一定违反氏替换原则。
只有俩个类满足里氏替换原则，才能使 Is-A 关系。
</blockquote>
<h5 id="参考">参考</h5>

《软件体系结构与设计》
https://www.jianshu.com/p/807bc228dbc
