**一些软件设计的原则**

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，**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使用父类型的模块在无需修改的情况下就可以扩展（使用不同的子类型）**。

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。