19 组件设计原则：组件的边界在哪里？

软件的复杂度和它的规模成**指数**关系，一个复杂度为100的软件系统，如果能拆分成两个互不相关、同等规模的子系统，那么每个子系统的复杂度应该是25，而不是50。软件开发这个行业很久之前就形成了一个共识，应该将复杂的软件系统进行拆分，拆成多个更低复杂度的子系统，子系统还可以继续拆分成更小粒度的组件。也就是说，软件需要进行模块化、组件化设计。

事实上，早在打孔纸带编程时代，程序员们就开始尝试进行软件的组件化设计。那些相对独立，可以被复用的程序被打在纸带卡片上，放在一个盒子里。当某个程序需要复用这个程序组件的时候，就把这一摞纸带卡片从盒子里拿出来，放在要运行的其他纸带的前面或者后面，被光电读卡器一起扫描，一起执行。

其实我们现在的组件开发与复用跟这个也差不多。比如我们用Java开发，会把独立的组件编译成一个一个的jar包，相当于这些组件被封装在一个一个的盒子里。需要复用的时候，程序只需要依赖这些jar包，运行的时候，只需要把这些依赖的jar包放在 `classpath` 路径下，最后被JVM统一装载，一起执行。

现在，稍有规模的软件系统一定被拆分成很多组件。正是因为组件化设计，我们才能开发出复杂的系统。

那么如何进行组件的设计呢？组件的粒度应该多大？如何对组件的功能进行划分？组件的边界又在哪里？

我们之前说过，软件设计的核心目标就是**高内聚、低耦合**。那么今天我们从这两个维度，看组件的设计原则。

组件内聚原则

组件内聚原则主要讨论哪些类应该聚合在同一个组件中，以便组件既能提供相对完整的功能，又不至于太过庞大。在具体设计中，可以遵循以下三个原则。

复用发布等同原则

复用发布等同原则是说，**软件复用的最小粒度应该等同于其发布的最小粒度**。也就是说，如果你希望别人以怎样的粒度复用你的软件，你就应该以怎样的粒度发布你的软件。这其实就是组件的定义了，组件是软件复用和发布的最小粒度软件单元。这个粒度既是复用的粒度，也是发布的粒度。

同时，如果你发布的组件会不断变更，那么你就应该用版本号做好组件的版本管理，以使组件的使用者能够知道自己是否需要升级组件版本，以及是否会出现组件不兼容的情况。因此，组件的版本号应该遵循一些大家都接受的约定。

这里有一个版本号约定建议供你参考，版本号格式：主版本号.次版本号.修订号。比如1.3.12，在这个版本号中，主版本号是1，次版本号是3，修订号是12。主版本号升级，表示组件发生了不向前兼容的重大修订；次版本号升级，表示组件进行了重要的功能修订或者bug修复，但是组件是向前兼容的；修订号升级，表示组件进行了不重要的功能修订或者bug修复。

共同封闭原则

共同封闭原则是说，**我们应该将那些会同时修改，并且为了相同目的而修改的类放到同一个组件中**。而将不会同时修改，并且不会为了相同目的而修改的类放到不同的组件中。

组件的目的虽然是为了复用，然而开发中常常引发问题的，恰恰在于组件本身的可维护性。如果组件在自己的生命周期中必须经历各种变更，那么最好不要涉及其他组件，相关的变更都在同一个组件中。这样，当变更发生的时候，只需要重新发布这个组件就可以了，而不是一大堆组件都受到牵连。

也许将某些类放入这个组件中对于复用是便利的、合理的，但如果组件的复用与维护发生冲突，比如这些类将来的变更和整个组件将来的变更是不同步的，不会由于相同的原因发生变更，那么为了可维护性，应该谨慎考虑，是不是应该将这些类和组件放在一起。

共同复用原则

共同复用原则是说，**不要强迫一个组件的用户依赖他们不需要的东西**。

这个原则一方面是说，我们应该将互相依赖，共同复用的类放在一个组件中。比如说，一个数据结构容器组件，提供数组、Hash表等各种数据结构容器，那么对数据结构遍历的类、排序的类也应该放在这个组件中，以使这个组件中的类共同对外提供服务。

另一方面，这个原则也说明，如果不是被共同依赖的类，就不应该放在同一个组件中。如果不被依赖的类发生变更，就会引起组件变更，进而引起使用组件的程序发生变更。这样就会导致组件的使用者产生不必要的困扰，甚至讨厌使用这样的组件，也造成了组件复用的困难。

其实，以上三个组件内聚原则相互之间也存在一些冲突，比如共同复用原则和共同闭包原则，一个强调易复用，一个强调易维护，而这两者是有冲突的。因此这些原则可以用来指导组件设计时的考量，但要想真正做出正确的设计决策，还需要架构师自己的经验和对场景的理解，对这些原则进行权衡。

组件耦合原则

组件内聚原则讨论的是组件应该包含哪些功能和类，而组件耦合原则讨论组件之间的耦合关系应该如何设计。组件耦合关系设计也应该遵循三个原则。

无循环依赖原则

无循环依赖原则说，**组件依赖关系中不应该出现环**。如果组件A依赖组件B，组件B依赖组件C，组件C又依赖组件A，就形成了循环依赖。

很多时候，循环依赖是在组件的变更过程中逐渐形成的，组件A版本1.0依赖组件B版本1.0，后来组件B升级到1.1，升级的某个功能依赖组件A的1.0版本，于是形成了循环依赖。如果组件设计的边界不清晰，组件开发设计缺乏评审，开发者只关注自己开发的组件，整个项目对组件依赖管理没有统一的规则，很有可能出现循环依赖。

而一旦系统内出现组件循环依赖，系统就会变得非常不稳定。一个微小的bug都可能导致连锁反应，在其他地方出现莫名其妙的问题，有时候甚至什么都没做，头一天还好好的系统，第二天就启动不了了。

在有严重循环依赖的系统内开发代码，整个技术团队就好像在焦油坑里编程，什么也不敢动，也动不了，只有焦躁和沮丧。

稳定依赖原则

稳定依赖原则说，**组件依赖关系必须指向更稳定的方向**。很少有变更的组件是稳定的，也就是说，经常变更的组件是不稳定的。根据稳定依赖原则，不稳定的组件应该依赖稳定的组件，而不是反过来。

反过来说，如果一个组件被更多组件依赖，那么它需要相对是稳定的，因为想要变更一个被很多组件依赖的组件，本身就是一件困难的事。相对应的，如果一个组件依赖了很多的组件，那么它相对也是不稳定的，因为它依赖的任何组件变更，都可能导致自己的变更。

稳定依赖原则通俗地说就是，**组件不应该依赖一个比自己还不稳定的组件**。

稳定抽象原则

稳定抽象原则说，**一个组件的抽象化程度应该与其稳定性程度一致。**也就是说，一个稳定的组件应该是抽象的，而不稳定的组件应该是具体的。

这个原则对具体开发的指导意义就是：如果你设计的组件是具体的、不稳定的，那么可以为这个组件对外提供服务的类设计一组接口，并把这组接口封装在一个专门的组件中，那么这个组件相对就比较抽象、稳定。

在具体实践中，这个抽象接口的组件设计，也应该遵循前面专栏讲到的[依赖倒置原则]。也就是说，抽象的接口组件不应该由低层具体实现组件定义，而应该由高层使用组件定义。高层使用组件依赖接口组件进行编程，而低层实现组件实现接口组件。

Java中的JDBC就是这样一个例子，在JDK中定义JDBC接口组件，这个接口组件位于 `java.sql` 包，我们开发应用程序的时候只需要使用JDBC的接口编程就可以了。而发布应用的时候，我们指定具体的实现组件，可以是MySQL实现的JDBC组件，也可以是Oracle实现的JDBC组件。

小结

组件的边界与依赖关系划分，不仅需要考虑技术问题，也要考虑业务场景问题。易变与稳定，依赖与被依赖，都需要放在业务场景中去考察。有的时候，甚至不只是技术和业务的问题，还需要考虑人的问题，在一个复杂的组织中，组件的依赖与设计需要考虑人的因素，如果组件的功能划分涉及到部门的职责边界，甚至会和公司内的政治关联起来。

因此，公司的技术沉淀与实力，公司的业务情况，部门与团队的人情世故，甚至公司的过往历史，都可能会对组件的设计产生影响。而能够深刻了解这些情况的，通常都是公司的一些“老人”。所以，年龄大的程序员并不一定要和年轻程序员拼技术甚至拼体力，应该发挥自己的所长，去做一些对自己、对公司更有价值的事。

思考题

在稳定抽象原则里，类似JDBC的例子还有很多，你能举几个吗？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

上一页 下一页