二

06 复杂度来源：可扩展性

你好，我是华仔。复杂度来源前面已经讲了高性能和高可用，今天我们来聊聊可扩展性。

可扩展性是指，系统为了应对将来需求变化而提供的一种扩展能力，当有新的需求出现时，系统不需要或者仅需要少量修改就可以支持，无须整个系统重构或者重建。

由于软件系统固有的多变性，新的需求总会不断提出来，因此可扩展性显得尤其重要。在软件开发领域，面向对象思想的提出，就是为了解决可扩展性带来的问题；后来的设计模式，更是将可扩展性做到了极致。得益于设计模式的巨大影响力，几乎所有的技术人员对于可扩展性都特别重视。

设计具备良好可扩展性的系统，有两个基本条件：

但要达成这两个条件，本身也是一件复杂的事情，我来具体分析一下。

预测变化

软件系统与硬件或者建筑相比，有一个很大的差异：软件系统在发布后，还可以不断地修改和演进。

这就意味着**不断有新的需求需要实现**。

如果新需求能够少改代码甚至不改代码就可以实现，那当然是皆大欢喜的，否则来一个需求就要求系统大改一次，成本会非常高，程序员心里也不爽（改来改去），产品经理也不爽（做得那么慢），老板也不爽（那么多人就只能干这么点事）。

因此作为架构师，我们总是试图去预测所有的变化，然后设计完美的方案来应对。当下一次需求真正来临时，架构师可以自豪地说：“这个我当时已经预测到了，架构已经完美地支持，只需要一两天工作量就可以了！”

然而理想是美好的，现实却是复杂的。有一句谚语：“唯一不变的是变化。”如果按照这个标准去衡量，架构师每个设计方案都要考虑可扩展性，例如：

架构师准备设计一个简单的后台管理系统，当架构师考虑用 MySQL 存储数据时，是否要考虑后续需要用 Oracle 来存储？

当架构师设计用 HTTP 做接口协议时，是否要考虑要不要支持 ProtocolBuffer？

甚至更离谱一点，架构师是否要考虑 VR 技术对架构的影响从而提前做好可扩展性？

如果每个点都考虑可扩展性，架构师会不堪重负，架构设计也会异常庞大且最终无法落地。但架构师也不能完全不做预测，否则可能系统刚上线，马上来新的需求就需要重构，这同样意味着前期很多投入的工作量也白费了。

同时，“预测”这个词，本身就暗示了不可能每次预测都是准确的。如果预测的事情出错，我们期望中的需求迟迟不来，甚至被明确否定，那么基于预测做的架构设计就没什么作用，投入的工作量也就白费了。

综合分析，预测变化的复杂性在于：

对于架构师来说，如何把握预测的程度和提升预测结果的准确性，是一件很复杂的事情，而且没有通用的标准可以简单套上去，更多是靠自己的经验、直觉。所以架构设计评审的时候，经常会出现两个设计师对某个判断争得面红耳赤的情况，原因就在于没有明确标准，不同的人理解和判断有偏差，而最终又只能选择其中一个判断。

2 年法则

那么我们设计架构的时候要怎么办呢？根据以往的职业经历和思考，我提炼出一个“2 年法则”供你参考：**只预测 2 年内的可能变化，不要试图预测 5 年甚至 10 年后的变化。**

当然，你可能会有疑问：为什么一定是 2 年呢？有的行业变化快，有的行业变化慢，不应该是按照行业特点来选择具体的预测周期吗？

理论上来说确实如此，但实际操作的时候你会发现，如果你要给出一个让大家都信服的行业预测周期，其实是很难的。

我之所以说要预测 2 年，是因为变化快的行业，你能够预测 2 年已经足够了；而变化慢的行业，本身就变化慢，预测本身的意义不大，预测 5 年和预测 2 年的结果是差不多的。所以“2 年法则”在大部分场景下都是适用的。

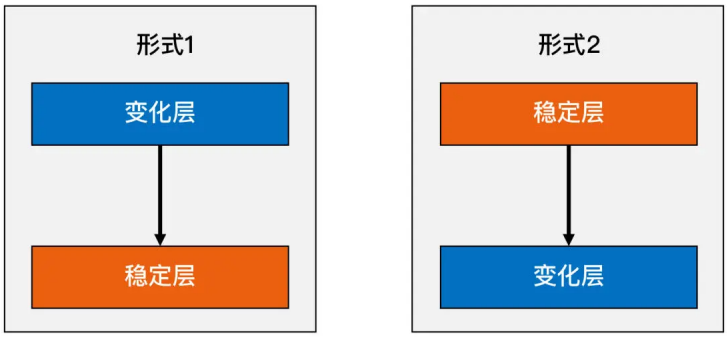
应对变化

假设架构师经验非常丰富，目光非常敏锐，看问题非常准，所有的变化都能准确预测，

是否意味着可扩展性就很容易实现了呢？也没那么理想！因为预测变化是一回事，采取什么方案来应对变化，又是另外一个复杂的事情。即使预测很准确，如果方案不合适，则系统扩展一样很麻烦。

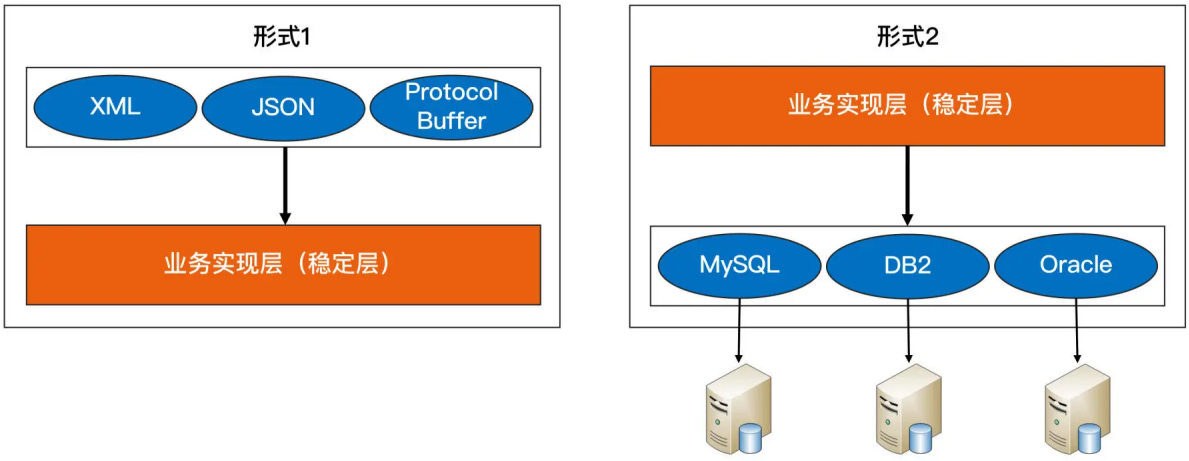
方案一：提炼出“变化层”和“稳定层”

第一种应对变化的常见方案是：**将不变的部分封装在一个独立的“稳定层”，将“变化”封装在一个“变化层”**（也叫“适配层”）。这种方案的核心思想是通过变化层来**隔离变化**。



无论是变化层依赖稳定层，还是稳定层依赖变化层都是可以的，需要根据具体业务情况来设计。

如果系统需要支持 XML、JSON、ProtocolBuffer 三种接入方式，那么最终的架构就是“形式 1”架构；如果系统需要支持 MySQL、Oracle、DB2 数据库存储，那么最终的架构就变成了“形式 2”的架构了。



无论采取哪种形式，通过剥离变化层和稳定层的方式应对变化，都会带来两个主要的复

杂性相关的问题。

对于哪些属于变化层，哪些属于稳定层，很多时候并不是像前面的示例（不同接口协议或者不同数据库）那样明确，不同的人有不同的理解，导致架构设计评审的时候可能吵翻天。

对于稳定层来说，接口肯定是越稳定越好；但对于变化层来说，在有差异的多个实现方式中找出共同点，并且还要保证当加入新的功能时，原有的接口不需要太大修改，这是一件很复杂的事情，所以接口设计同样至关重要。

例如，MySQL 的 REPLACE INTO 和 Oracle 的 MERGE INTO 语法和功能有一些差异，那么存储层如何向稳定层提供数据访问接口呢？是采取 MySQL 的方式，还是采取 Oracle 的方式，还是自适应判断？如果再考虑 DB2 的情况呢？

看到这里，相信你已经能够大致体会到接口设计的复杂性了。

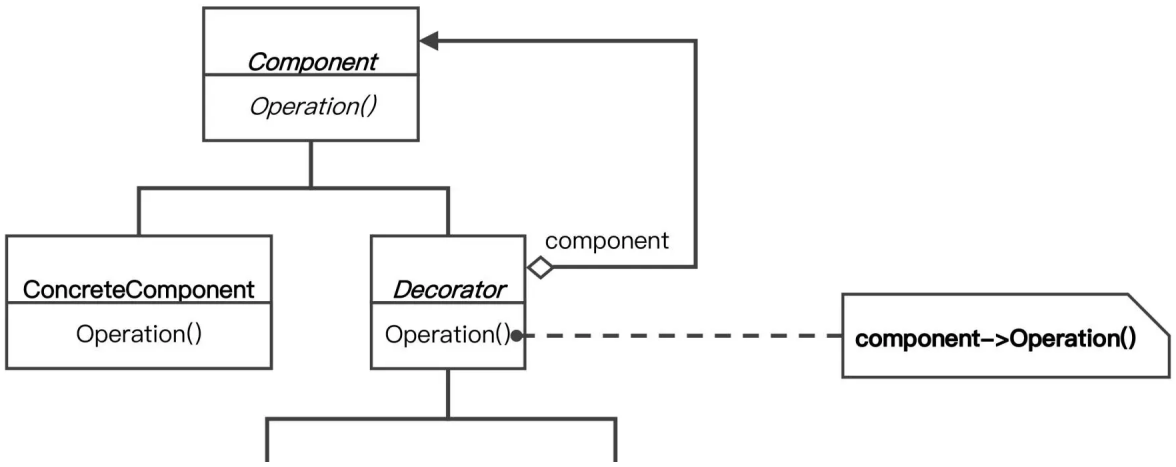
方案二：提炼出“抽象层”和“实现层”

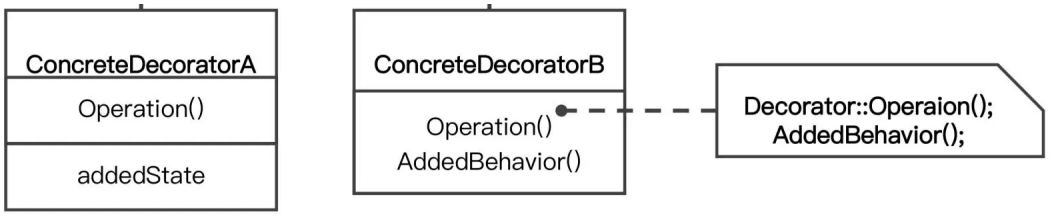
第二种常见的应对变化的方案是：**提炼出一个“抽象层”和一个“实现层”**。如果说方案一的核心思想是通过变化层来隔离变化，那么方案二的核心思想就是通过实现层来**封装变化**。

因为抽象层的接口是稳定的不变的，我们可以基于抽象层的接口来实现统一的处理规则，而实现层可以根据具体业务需求定制开发不同的实现细节，所以当加入新的功能时，只要遵循处理规则然后修改实现层，增加新的实现细节就可以了，无须修改抽象层。

方案二典型的实践就是设计模式和规则引擎。考虑到绝大部分技术人员对设计模式都非常熟悉，我以设计模式为例来说明这种方案的复杂性。

下面是设计模式的“装饰者”模式的类关系图。





图中的 Component 和 Decorator 就是抽象出来的规则，这个规则包括几部分：

Decorator 类继承 Component 类。

Decorator 类聚合了 Component 类。

这个规则一旦抽象出来后就固定了，不能轻易修改。例如，把规则 3 去掉，就无法实现装饰者模式的目的了。

装饰者模式相比传统的继承来实现功能，确实灵活很多。例如，《设计模式》中装饰者模式的样例“TextView”类的实现，用了装饰者之后，能够灵活地给 TextView 增加额外更多功能，包括可以增加边框、滚动条和背景图片等。这些功能上的组合不影响规则，只需要按照规则实现即可。

但装饰者模式相对普通的类实现模式，明显要复杂多了。本来一个函数或者一个类就能搞定的事情，现在要拆分成多个类，而且多个类之间必须按照装饰者模式来设计和调用。

规则引擎和设计模式类似，都是通过灵活的设计来达到可扩展的目的，但“灵活的设计”本身就是一件复杂的事情，不说别的，光是把 23 种设计模式全部理解和备注，都是一件很困难的事情。

1 写 2 抄 3 重构原则

那么，我们在实际工作中具体如何来应对变化呢？Martin Fowler 在他的经典书籍《重构》中给出一个“Rule of three”的原则，原文是“Three Strikes And You Refactor”，中文一般翻译为“事不过三，三则重构”。

而我将其翻译为“1 写 2 抄 3 重构”，也就是说你不要一开始就考虑复杂的可扩展性应对方法，而是等到第三次遇到类似的实现的时候再来重构，重构的时候采取隔离或者封装的方案。

举个最简单的例子，假设你们的创新业务要对接第三方钱包，按照这个原则，就可以这样做：

1 写：最开始你们选择了微信钱包对接，此时不需要考虑太多可扩展性，直接快速对照

微信支付的 API 对接即可，因为业务是否能做起来还不确定。

2 抄：后来你们发现业务发展不错，决定要接入支付宝，此时还是可以不考虑可扩展，直接把原来微信支付接入的代码拷贝过来，然后对照支付宝的 API，快速修改上线。

3 重构：因为业务发展不错，为了方便更多用户，你们决定接入银联云闪付，此时就需要考虑重构，参考设计模式的模板方法和策略模式将支付对接的功能进行封装。

小结

今天我从预测变化和应对变化这两个设计可扩展性系统的条件，以及它们实现起来本身的复杂性，为你讲了复杂度来源之一的可扩展性，希望对你有所帮助。

这就是今天的全部内容，留一道思考题给你吧。你在具体代码中使用过哪些可扩展的技术？最终的效果如何？

上一页 下一页