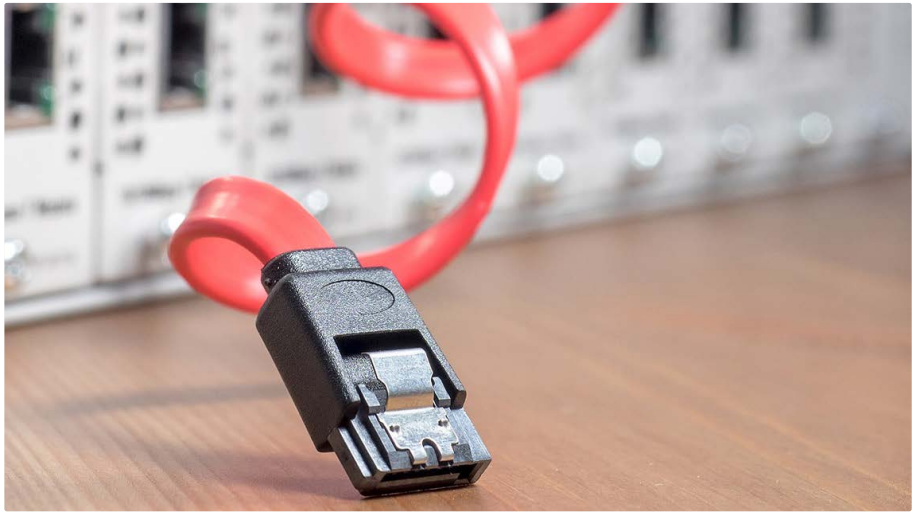
06 | 复杂度来源：可扩展性

2018-05-10 李运华



06 | 复杂度来源：可扩展性
李运华
00:00 / 07:52

复杂度来源前面已经讲了高性能和高可用，今天来聊聊可扩展性。

可扩展性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：正确预测变化、完美封装变化。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。如果新需求能够不改代码甚至少改代码就可以实现，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。

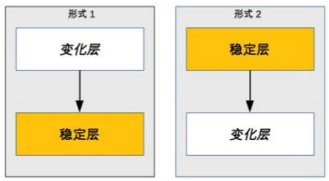
综合分析，预测变化的复杂性在于：

- 不能每个设计点都考虑可扩展性。
- 不能完全不考虑可扩展性。
- 所有的预测都存在出错的可能性。

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，而最终又只能选择一个判断。

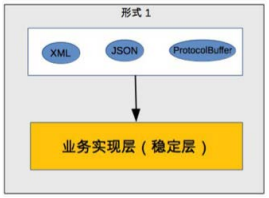
应对变化

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。

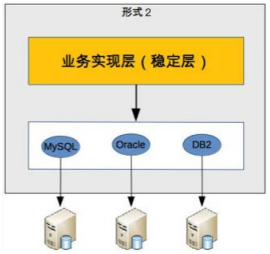


第一种应对变化的常见方案是将“变化”封装在一个“变化层”，将不变的部分封装在一个独立的“稳定层”。

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。例如，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上面图中的“形式1”架构，也就是下面这样。



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，你可以看下面这张图。



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。

1.系统需要拆分出变化层和稳定层

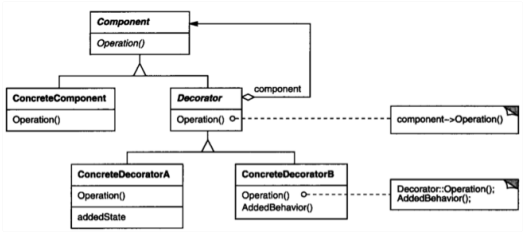
对于哪些属于变化层，哪些属于稳定层，很多时候并不是像前面的示例（不同接口协议或者不同数据库）那样明确，不同的人有不同的理解，导致架构设计评审的时候可能吵翻天。

2.需要设计变化层和稳定层之间的接口

接口设计同样至关重要，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设计不需要太大修改，这是一件很复杂的事情。例如，MySQL的REPLACE INTO和Oracle的MERGE INTO语法和功能有一些差异，那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？相信你看到这里就已经能够大致体会到接口设计的复杂性了。

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”。抽象层是稳定的，实现层可以根据具体业务需要定制开发，当加入新的功能时，只需要增加新的实现，无须修改抽象层。这种方案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。

以设计模式的“装饰者”模式来分析，下面是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：

1.Component和Decorator类。2.Decorator 类继承Component类。3.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，比如可以增加边框、滚动条、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。

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？

欢迎你把答案写到留言区，和我一起讨论。相信经过深度思考的回答，也会让你对知识的理解更加深刻。（编辑乱入：精彩的留言有机会获得丰厚福利哦！）

公号-Java大后端

2018-05-10

今日心得

1 What：什么是架构的可扩展性？

业务需求、运行环境方面的变化都会导致软件系统发生变化，而这种软件系统对上述变化的适应能力就是可扩展性。

可扩展性可以理解为是一种从功能需求方面考虑的软件属性，属性就会存在好坏之分。

按照可扩展性的定义，一个具备良好可扩展性的架构设计应当符合开闭原则：对扩展开放，对修改关闭。衡量一个软件系统具备良好可扩展性主要表现但不限于：（1）软件自身内部方面。在软件系统实现新增的业务功能时，对现有系统功能影响较少，即不需要对现有功能作任何改动或者很少改动。（2）软件外部方面。软件系统本身与其他存在协同关系的外部系统之间存在松耦合关系，软件系统的变化对其他软件系统无影响，其他软件系统和功能不需要进行改动。反之，则是一个可扩展性不好的软件系统。

2 Why：为什么要求架构具备良好的可扩展性？

伴随业务的发展、创新，运行环境的变化，对技术也就提出了更多、更高的要求。能够快速响应上述变化，并最大程度降低对现有系统的影响，是设计可扩展性好的架构的主要目的。

3 How：如何设计可扩展性好的架构？

面向对象思想、设计模式都是为了解决可扩展性的而出现的方法与技术。

设计具备良好可扩展性的系统，有两个思考角度：（1）从业务维度。对业务深入理解，对可预计的业务变化进行预测。（2）从技术维度。利用扩展性好的技术，实现对变化的封装。

(1)在业务维度。对业务深入理解，对业务的发展方向进行预判，也就是不能完全不考虑可扩展性；但是，变化无处不在，在业务看得远一点的同时，需要注意:警惕过度设计；不能每个设计点都考虑可扩展性；所有的预测都存在不正确的可能性。

(2)在技术维度。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。第一种应对变化的常见方案是将"变化"封装在一个"变化层"，将不变的部分封装在一个独立的"稳定层"。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"。

4.在实际工作场景中的解决方案
在实际软件系统架构设计中，常通过以下技术手段实现良好的可扩展性：（1）使用分布式服务(框架)构建可复用的业务平台。（2）使用分布式消息队列降低业务模块间的耦合性。

(1)分布式服务框架
利用分布式服务框架(如Dubbo)可以将业务逻辑实现和可复用组件服务分离开，通过接口降低子系统或模块间的耦合性。新增功能时，可以通过调用可复用的组件实现自身的业务逻辑，而对现有系统没有任何影响。可复用组件升级变更的时候，可以提供多版本服务对应用实现透明升级，对现有应用不会造成影响。

(2) 分布式消息队列
基于生产者-消费者编程模式，利用分布式消息队列(如RabbitMQ)将用户请求、业务请求作为消息发布者将事件构造成消息发布到消息队列，消息的订阅者作为消费者从消息队列中获取消息进行处理。通过这种方式将消息生产和消息处理分离开来，可以透明地增加新的消息生产者任务或者新的消息消费者任务。

曹铮

2018-05-10

我平时工作中更多提醒自己压抑一下想预测各种需求变化的欲望。因为之前总是过度设计。压抑并不是说不去考虑各种变化，而恰恰需要把可能性大的变化点一一罗列出来，分维度打分，维度包括 可能性大小？长期还是短期会变化？如果发生变化，目前的组织和系统要花多大成本适应变化。这些变化正是李老师之前说过的各种复杂度上的变化，比如用户量激增带来的性能要求。此外还包括一个业务功能逻辑上的变化。
在经过上面分析后往往会给出"上中下"策的设计方案，下策一般考虑的变化少，短视，但迅速，修改小，立竿见影。上策一般看重远期，但成本高很高，也很可能预测不中。
最后还要分析，如果决定采用下中策，如果预测的变化发生了，系统修改为中上策的代价有多大，有些代价几乎是无穷大的，比如必须中断服务进行升级。如果代价小，那可以放心采用下策或中策。如果答案是否，可上策当前的代价又真的不可接受，那又要返回头重新分析了
实践发现这个方法挺好用，尤其当有人来咨询架构方案时，往往对给出的结果比较满意

作者回复

2018-05-10

挺实用的方法，架构设计原则部分会讲到

憶海拾貝

2018-05-11

设计模式的核心就是，封装变化，隔离可变性

作者回复

2018-05-11

这是设计模式的核心思想，能理解到这点比背住23个模式更重要

Jaime

2018-05-10

曾经在游戏中使用过工厂模式和状态模式，但交个另外一个人维护了一个月，我回头去看，代码已经没办法入眼[illegible]。虽然说设计模式确实是程序员的基本功，但其实很多程序员也不是很明白设计模式的。对于现在来说，我比较喜欢的做法就是先分层，层与层之间用消息解耦，在层内部的实现我会分模块出来，遵守单一职责原则。同时会积极跟业务部门沟通，预测一下下一步的方向。虽然不是每次都准确，但也大概做到心中有数，对于现在的系统也有个预估。这几年的编程经验给了我一些启发，一定不要过分设计，首先要能正确工作的软件是最重要的。

作者回复

2018-05-11

符合实际情况，分层最有用，代码中用设计模式，如果后面接手的人不懂或者理解不到位，最后改的代码简直没法理解，这样面向过程[illegible]

Mark Yao

2018-05-10

说说我们的消息系统，原始需求是用户操作业务后给用户手机短信推送，设计初衷考虑到可能涉及到多家第三方短信服务商。我们统一定义发送短信接口，定义实时和定时发送方式，在内容形式定义模版接口接受不通形式自定义模版的内容，后来把短信定义为消息中的一种，如微信、短信、站内消息、app push 都为消息，又抽象出来消息接口，消息开发中使用多种设计模式。最后发现就就一直使用短信服务。我困惑地方，扩展性需要在什么时候做，做到什么程度呢？

作者回复

2018-05-10

我的经验供参考：设计的时候考虑可扩展性，但如果评估后发现可扩展性设计的代价太大，那就暂时不做，等到真的有需求时再重构。

过早考虑可扩展性，很多通用性和抽象都是推测的，等到真的要落地了，很可能发现事实并非如此，这就是预测是错误的。

回到你的案例，一般来说，短信本身会考虑可扩展性，例如联通的短信接口和电信的不同，这种可扩展性是要一开始就设计的，但短信和微信，看起来都是消息，实际上差异非常大，可扩展性设计想兼容这两种方式，方案比较复杂，可能看起来有点不伦不类

jw

2018-05-10

应对变化的两个方案：
1，封装变化层和稳定层
2，提炼抽象层和实现层
本质上都是在在将变化和稳定分离

Jesse.zhang

2018-05-10

学习了，扩展性是为了应对未来新的需求变化能快速响应的一种能力，前期架构时需要考虑一定的扩展性，但无需事事考虑，考虑的部分是基于需求分析，以系统重点的业务方向上，其他的业务先开干，之后以代码重构方式考虑扩展，这样一方面可以加快系统落地，其次减少扩展性错误或不完美而带来的工作量。

朋朋

2018-06-03

做游戏 经常变需求玩法 特别是策划做新系统时 但他不能大改旧系统 还得相互适配 因为毕竟是一款游戏 这样 可以把整个系统拆分成多个服务 服务和服务之间用消息队列 一个服务内部用分层的方法 那个层变啦 就修改那个层 那个服务业务变啦就修改那个服务 不用大改整个系统

孙振超

2018-05-26

之前看过一篇介绍架构的博文，里面提到一个观点：衡量架构的好坏是变更的成本。扩展性的好坏很依赖于设计人员对问题的抽象能力，如同文中所描述的把系统分为稳定层和抽象层两部分，就是对问题进行了抽象。具体而言，在设计上经常采用的方法是模板+接口，将具体的业务逻辑抽象为固定的几个步骤，每一个步骤是一个接口，而后根据不同的对应的参数动态选择不同的实现，这样当已有的业务发生变更时，只需要调整相应的逻辑即可，做好关注点分离，面对新增的业务逻辑和形态，添加对应的实现即可，无需修改已有的内容。
另外就是利用动态修改能力（比如java中无需重启服务修改某一个属性的值）来应对业务变动，提升扩展性

作者回复

2018-05-27

将业务逻辑抽象为固定的步骤，适合业务已经比较成熟了，例如nginx将http的处理步骤抽象为大概10个阶段，如果是创新业务，这样做比较难

张玮(大圣)

2018-05-11

看了全部的评论，都在谈构架，架构，其实想说的是在扩展性设计是首先要考虑的是当前的组织和业务体量能承载和所要求的架构，

创业期间，单一，中期分层慢慢过渡，具体到每个子域也是分层慢慢过渡，笼统的说那就是分层，分层，分层，最终还是考量技术人如何界定分层的事

narry

2018-05-10

对于可扩展性，我最常使用的微内核和""流水线+filter"两个模式，微内核将稳定的核心部分逐渐的固定下来，保证系统的稳定，流水线的模式满足了开闭的原则，利于系统的扩展

王念

2018-05-13

先满足功能 再重构框架

作者回复

2018-05-14

这也不行，可能功能还没做完就要重构了

带刺的温柔

2018-05-11

厉害了程序员都想要开发一个完美的灵活可扩展的系统永无止境，而往往陷入过度设计的泥潭，最后累的要死得到的可能是貌似完美符合了扩展性但是非常不好用甚至有点画蛇添足的感觉。觉得扩展性不是一触而就的也不是一成不变的它是一它是不断改进的过程，不变的是满足需求是底线在追求扩展性的路上把风险控制在最低。我在兼容简单与扩展性上我觉得一定的冗余是个不错的选择，老师你觉得呢

作者回复

2018-05-11

非常正确，一定的冗余和浪费，能够大大减少方案的复杂度

轩辕十四

2018-06-26

我感觉中文"可扩展性"其实对应两个概念，extensible(对变化的扩展），以及scalable（对规模的扩展）。分开来讲可能更清楚些

不再犹豫

2018-05-23

代码层面来说我觉得是快速实现，持续重构。建立在对业务充分理解的基础上，按照依赖抽象的原则，提取出因业务变化而导致代码逻辑变化部分。

作者回复

2018-05-23

是的，应用设计模式，设计原则就可以了

aiwen

2018-05-10

讨论问题比较多，解决问题的方案却没提，后续会有相关的问题成熟的常用的解决方案么

作者回复

2018-05-11

有的，你可以详细看看目录

李艳超_Harry

2018-05-10

把一些逻辑脚本化。python grovvy

成功

2018-05-10

适配层用过，也就是今天讲的变化层。

艳姐

2018-07-04

设计模式是代码级别的，跟架构层面的关系，按照层面扩展性例子没说了数据库的优化，能否再举个复杂点的例子更直观

作者回复

2018-07-05

例子是为了说明本质的，本篇在于讲解可扩展性为什么复杂这个问题，提炼了“预测变化”和“应对变化”两个复杂点，你可以结合自己的业务按照这种套路分析一下

Carlos

2018-06-30

李老师专栏真是干货满满呀。我一般是上下班路上听，每篇文章听的不止一两遍。就可扩展性这片，结合老师讲解，我说说自己想法。我认为可扩展性架构里的变化应该分为两种，一种是新增，一种是优化。第一种是架构设计需要考虑的问题，比如系统增加新接入方式，不用改现有代码，只要新增一种实现就能满足，这是好的架构。不好的架构是无法通过新增新实现来完成，而是要修改现有代码逻辑，这样会带来两个问题，一是可能会破坏原有的功能，二是增加原有方式的代码复杂度，为维护增加难度。第二种优化可能主要是功能设计时需要考虑的，现实情况是，产品本身的某个使用习惯，在对客户进行演示交流时，不满足客户要求，他们更习惯另外一种方式，这种情况先是进行引导，引导不成，就要结合客户意愿，进行优化了，这种优化通常是基于原有逻辑的修改。新增实现就没有意义了。

作者回复

2018-07-02

有的业务逻辑，用规则引擎之类的实现，优化也可以做到很容易扩展

chris

2018-06-30

23种设计模式不是死记硬背，生搬乱套的，而是你写的业务代码确实满足了业务不断出现的变化和扩展，即使没用到这23种设计模式，也是可扩展的！

武洪凯

2018-06-26

老师能不能推荐一些构架的书，中文的英文的都可以。课程讲的很好，不过很多细节深扣的话感觉还要继续看书。

作者回复

2018-06-28

深扣细节的话，就直接奔着具体某个系统或者专题，例如kafka或者缓存，这样找书就很容易了

zhou

2018-06-14

老师你讲的看起来有点吃力，能否推荐几本书让我先入个门

作者回复

2018-06-14

专栏已经是我综合自己的经验，结合看了很多架构设计的书籍整出来的，如果专栏看的有点吃力，那看其它书会更吃力。

可以说说你吃力的点，看看能不能帮到你

Chang

2018-06-13

设计不同算法，没办法做到极致！其实设计项目多了发现都是trade off

作者回复

2018-06-13

关键是平衡是一门艺术��

我走我流

2018-06-11

看来设计模式这本书还要翻出来好好学学

作者回复

2018-06-13

架构层面的可扩展和设计模式差别还是比较大的

陈友洲

2018-06-04

话说大神提到的规则引擎是个啥？有哪些使用场景？谢谢

作者回复

2018-06-04

请自行搜索

zhoubo

2018-05-31

华哥，你好，想实现docker部署基础上的项目的自动扩缩容，k8s 又觉得太重，请问有轻量级的技术可选吗？

JOEL

2018-05-21

留个言，本文讲了封装变化以保持扩展的两种方式：分层和抽象。但具体做法却有很多，比如servlet的过滤器，比如从架构上就采用了SOA应该也算，比如有些构建工具maven，采用了预订定义生命周期的方式，有些系统干脆采用了标准化的方式把复杂性隔离到系统外部，做法多种多样，要是能多列举这些最佳实践，也挺有收获。

作者回复

2018-05-22

实践很多，我主张侧重提炼，不然很多人会陷入细节出不来

gevin

2018-05-21

关于可扩展性，我的经验做法是，通常在最初写代码时，按当时对业务需求的认识，对变化做一定的预测（其实就是在动手开发每个业务模块甚至每个功能前，先仔细想一下业务需求，如果发现某个地方可能会变化，就把该变化封装到代码里），并对此在代码里把相应的可扩展性做进去，这样通常能顶住一定的需求变化；然后是针对已做完的功能模块，如果因需求变化需要改动时，一定要再仔细分析需求，在对业务认识所及的范围内，尽量做好预测，并封装变化，做这部分功能的原因是，一个业务功能如果会变化一次，通常也会变化第二次甚至更多次，所以要被改动的功能，就是封装变化的目标。当然，预测变化要有个度，毕竟预测是不准确的

作者回复

2018-05-21

度的把握是一门艺术

古座

2018-05-19

Spi、动态编译。
DDD，对复杂逻辑和扩展性非常友好，但是不好理解程序员学习成本高，按需简化了。
洋葱架构，按职责抽象好几层，比DDD学习成本低。
ESB，异构系统，踩了很多坑……
Akka，raft，gossip自己玩儿的，一直没机会用在项目里……

日光倾城

2018-05-17

前面讲高性能和高可用还觉得很贴近，这个可拓展性就觉得很不好把握了，可能我平常接触的需求比较简单，就只考虑怎么实现，没有考虑扩展性，有时候觉得需求太个性化没啥拓展可言，或者说抽不出啥变化点不变，所以这方面的经验欠缺，后面方代中用到加强这方面的感知

作者回复

2018-05-18

企业级应用，互联网应用，可扩展性都很常见呢，你可以看看《淘宝产品十年事》，里面讲需求的各种变化

路易斯陈凯瑞

2018-05-15

从个人经历来说扩展性更多体现在功能设计阶段，结合业务理解应用设计模式或者一些最佳实践，用的比较多的就是模版方法。

J

2018-05-15

可扩展的技术就是使用了设计模式 ，比如文中提到的装饰者，还有策略模式和模板方法

作者回复

2018-05-16

这是代码层扩展，架构层扩展不能这样做

Sruby

2018-05-15

对于业务流程相同的子类业务，使用抽象类来定义流程，各个业务子类只要实现各自的业务就可以。

张驰

2018-05-14

SPI 动态实现

铭毅天下

2018-05-14

在Elasticsearch开发API选型的时候，使用JEST，最主要的原因：jest对不同es版本都适用。
主要考虑：ES版本更新迭代快，一个稳健的API选型，从可扩展性角度适配了ES版本的变化和更迭。

东方之猪

2018-05-13

项目里灵活运用模板方法，在抽相中定义不变的方法簇，将实现延迟到自类，让各自表现体独自变化，当时就预测这个算法簇不会频繁变化，事实也是预测如此，封装变化，面相抽象而不是实现，瞬间觉得代码水平是面相对象而不是面相过程

作者回复

2018-05-14

那不是过度设计了？

云辉

2018-05-12

扩展性可以理解为最小成本实现变化吧，分层，接口，配置都是思路吧，看谁抽象的功夫深了，不一定要面向对象。

天天向上卡索

2018-05-12

面向接口编程，提取公共部分代码为核心代码，其他的再实现相应接口，设计模式并非空谈，设计模式用的好可以减少后期修改维护的工作量，而设计模式的使用也是要从实际业务场景出发，避免过度设计

作者回复

2018-05-12

设计模式不是空谈，但实践中空谈设计模式和过度使用的不少，关键还理解不到位��

东风

2018-05-12

可扩展性，我觉着是个相当复杂的复杂性。
首先，什么是扩展，在哪里扩展，这就不是容易达成一致的；
其次，为什么是扩展而不是各司其职，也是个好问题；
最近几天在思考是重构还是重写的问题，资源不足又要做就重构，资源充足就重写，看着简单的答案，还是无从下手；
今天为大师兄们留下的小300万行代码打补丁，吐槽，怎么做才能不让小师弟继续吐槽？重构或者重写能解决？
会不会没等小师弟吐槽，回过头来自己就吐槽自己，总结，如果做不到一边生产，一边重构，除了过度设计没有啥重用和复用的捷径

彡工鸟

2018-05-11

这里无法跟作者好好交流:(只能另外留言。其实我想说的是在扩展方面，无论是架构还是代码层面，其实都离不开软件那几大设计原则，而且不用太死记硬背。就像设计模式只是总结出来，而不是某个时间点蹦哒出来。会使用这些只是代表很好的封装了变化。会设计架构肯定能很好用这些原则，反过来是不成立的

老王

2018-05-11

今天学习了，系统要区分"稳定层"和"变化层"。而且变化层也不只是为了扩展吧，变化层可能为需要经常改动的逻辑代码，如果稳定层的代码需要经常改动，那么程序就会bug不断。

李志博

2018-05-11

正打算规则引擎剥离变化的部分

作者回复

2018-05-11

做好踩坑的心理准备��

anchor

2018-05-11

根据不同配送商查询不同数据和打印不同格式面单，通过将将获取数据和打印请求的代码做到可配置到数据库，执行时候动态生成，执行的方式解决

Seven _ dong

2018-05-11

个人觉得可扩展性，是一个高级开发人员的衡量标准，也是迈向架构的第一步，如果说高性能和高可用还可以用机器来扛的话，那么可扩展就真是代码功力了。最近在做组件库，深感设计一个好的组件库的不容易。尤其是考虑到要让使用者用的简单，但是又易于扩展，有的时候就是个悖论。另外老师能不能讲讲在FP下的扩展性怎么做呢?

呵呵

2018-05-10

所谓架构层面的扩展，感觉一般情况下指的是扩容性

作者回复

2018-05-11

扩容性一般称为"可伸缩性"，属于高性能的一部分，就是任务分配的模式带来的效果

feifei

2018-05-10

抽象出业务流，使用责任链来进行实现，可直接实现接口完成扩张，不修改原有代码

乡工鸟

2018-05-10

无论是架构还是代码层面，其实把软件设计那几大原则好好理解，基本上都不是事。剩下的就是实践实践实践。所谓的23个设计模式基本上都能对上

作者回复

2018-05-11

然而我敢打赌即使背住23个设计模式后还是不会做架构设计，我曾经就是这样的

明翼

2018-05-10

不太明白老师的说法，稳定层就是几个接口，变化很小吧

作者回复

2018-05-11

稳定层可以是库，可以是包，可以是接口，可以是中间件，可以是子系统，例如Linux的VFS

波波安

2018-05-10

我们讲的是代码层面的可扩展性，还是架构本身的可扩展性？

作者回复

2018-05-10

可扩展性可以是架构层面的，可以是代码层面的，本质都是一样的，提问用了“代码”中的可扩展性，是考虑到可能比较多的朋友没有做过可扩展性架构设计。

王策

2018-05-10

短信 微信和站内信那个项目可能想做的事情太多，是不是存在违反单一职责原则的问题

作者回复

2018-05-10

单一职责一般用在编码层面，用来指导类设计，用于架构层面的话，很难明确“单一”的粒度。例如，到底“用户管理”（包括登录注册信息管理）是单一职责，还是“用户登录”是单一职责，看起来都可以。

明翼

2018-05-10

设计模式里面的依赖倒置原则，上层不依赖下层，下层也不依赖上层，两者都依赖于抽象，抽象是稳定的，上层和下层都是可扩展的，相当于文章说的一个稳定层抽象，两个变化层...

作者回复

2018-05-10

纠正一下：SOLID原则不是设计模式里面的，而是对象和接口的设计原则。

依赖倒置的难点就在于稳定层的设计，实践中稳定层也难以保证稳定

更多一手资源请添加QQ/微信1182316662