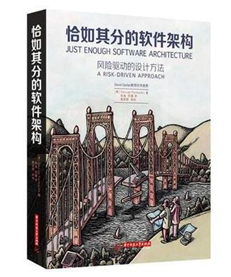
**《恰如其分的软件架构》读书笔记**

建筑与环境学院 张泽 2021151470032



这学期我选择了《恰如其分的软件架构》这本书来阅读，这本书是由 George和Fairbanks 共同编著，致力于通过风险驱动的方法进行软件架构开发。但是这本书在读起来有点文邹邹的感觉，有些地方有点晦涩难懂。书中提倡按风险驱动的架构设计，是值得借鉴的设计方法。

本书描述了一种恰如其分的软件架构设计方法。作者建议根据项目面临的风险来调整架构设计的成本，并从多个视角阐述了软件架构的建模过程和方法，包括用例模型、概念模型、域模型、设计模型和代码模型等。本书不仅介绍方法，而且还对方法和概念进行了归类和阐述，将软件架构设计融入开发实践中，与敏捷开发方法有机地结合在一起，比较适合程序员来阅读。

通过阅读本书，从用以下几个方面来进行总结

**一、什么是软件架构？**

软件架构就是系统设计，以及它对诸如性能、安全和可修改性等系统所产生的影响。软件架构抉择很重要，架构是系统骨骼，直接影响质量属性，并约束整个系统。从某个角度来说，架构与功能可以互相组合，只是不同组合有不同表现效果而已。利用模型和抽象概念去构建和解释系统架构。架构的质量直接影响到软件的性能、可维护性和可扩展性。书中强调，一个好的架构能够使软件系统更加健壮，更容易适应变化，同时也能降低开发和维护的成本。

**二、软件架构为什么重要？**

软件架构变得重要的时机通常是在系统复杂性上升时。随着系统的规模不断扩大，涉及到的组件、功能模块、以及业务需求愈加复杂，单纯的代码设计和实现已经无法保证系统的长期健康和可持续发展。架构作为高层次的设计方案，帮助团队从整体上把控系统的方向，并且为后期的扩展、维护、以及可能的技术栈变化提供清晰的框架和指导。

由于问题复杂，很难设计出好的解决方案，因此架构就显得尤为重要， 它不仅是解决复杂问题的手段，更是架构师应对不确定性和变化的武器。在面对复杂的业务需求和技术限制时，架构需要帮助开发团队在多变的环境中找到最优解，并且为系统的可维护性、可扩展性、可靠性等提供保障。

随着软件系统的演进，架构的重要性进一步凸显。尤其是在以下几个场景中，架构变得至关重要：

1. 业务需求频繁变化：在需求不断变化的环境中，架构提供了灵活性和适应性。设计合理的架构能够帮助团队快速响应需求的变化，同时避免过度开发和冗余功能。
2. 系统规模增大：随着系统的复杂度增加，单纯的代码层次和模块划分变得无法满足开发和维护的需求。架构有助于将复杂的系统拆解成更易管理和维护的模块或组件。
3. 团队协作：随着开发团队规模的扩大，良好的架构能够帮助不同的团队成员在保持一致性的同时，有效协作。架构设计往往是各个团队之间沟通和协作的基础。

**三、风险驱动模型**

在设计架构时，需要问这么几个问题：

1. 我的风险是什么？
2. 用于降低这些风险的技术是什么？
3. 风险是否化解，可以开始编码了吗？

风险驱动模型归纳三个步骤：

1、识别风险，并排定优先级

2、选择并运用一组技术降低风险

3、评估风险降低的程度

上述问题的提出，是为了帮助架构师在系统设计初期就能够识别潜在的风险，并采取适当的措施来应对它们，从而降低系统开发和运行中的不确定性和失败概率，项目领域的风险如下图，风险驱动模型可归纳为以下三个步骤：



图1：项目领域典型的风险

1. 识别风险，并排定优先级

在架构设计的初期，首先要识别可能影响项目成功的各种风险。风险可以来自多个方面，例如技术难题、架构复杂性、团队能力、需求变更、时间约束等。一旦识别出这些风险，需要根据它们的影响程度和发生概率对其进行排序，优先处理那些可能造成严重后果的风险。

1. 选择并运用一组技术降低风险

在确定了风险后，接下来需要评估可用的技术或方法来降低这些风险。例如，面对性能风险时，可以选择使用负载均衡、缓存、分布式架构等技术；对于可扩展性风险，可以通过模块化设计、微服务架构等手段来降低系统耦合，提升灵活性。选择合适的技术不仅可以缓解已识别的风险，还能为后续的开发工作打下坚实的基础。

1. 评估风险降低的程度

风险降低的程度是衡量架构设计是否成功的重要标准。在选择并实施了相应的技术措施后，需要评估这些措施是否有效地减轻了风险。可以通过原型验证、性能测试、系统可维护性分析等手段，来检查这些技术是否真正达到了预期的效果。如果风险没有得到有效控制或缓解，可能需要重新审视架构设计，进一步优化或选择其他方案。

**四、技术选型指导原则**

1、首先，当面临一个要解决的问题，而在其他情况下又遇到需要证明的问题，技术决策应该与具体需求相匹配

2、其次，某些问题可以通过类比模型解决，而其他问题，借助分析模型解决，此时需要分辨不同模型之间的差异

3、再次，采用特定类型的模型，才能有效分析特定问题

4、最后，某些技术之间存在着密切的关系，分析他们之间关系也很重要

要求解的问题和要证明的问题：要求解的问题，找到答案就可以了；要证明的问题，需要找到所有情况为真，这比求解问题难的多。

书中有一个举例：“数据库能否保存多大一百个字符的名称？”它属于要求解的问题，为此问题编写测试用例就很容易验证出来，“该系统是否一直符合该框架的应用程序编程接口？” 这就是需要证明的问题。尽管可以做多种测试，但，仍然会有某些情况被忽略了。

**五、如何把握设计与架构的度？**

有人说架构就是一种平衡，在现阶段下，“利益”的平衡和取舍。架构设计越复杂，那么，实现所花费的时间就越多，而需求变化又极快，所以必须平衡需求与架构之间冲突。不需要识别出所有风险后，包括现在和以后的风险才开始架构，只要架构能满足现在业务发展，并能应付未来1到2年的发展即可。

5.1 演进式设计

从历史过往来看，演进式架构饱受争议，因为局部而又不协调的设计决策会导致混乱，从而创造出一个大杂烩，既难维护，又难以进一步演进。然而，随着敏捷实践中的重构、测试驱动设计以及持续集成可以对付各种混乱问题。重构（对代码进改进）清除了不协调的局部设计，测试驱动设计确保对系统更改不会导致系统丢失或破坏现有功能，持续集成则为整个团队提供了统一代码库。

因为有上述种种实践，一些人认为可以不用做计划式设计。其中，重构是克服演进式设计中大杂烩问题的主力。但重构缺陷是，它并没有为架构规模的转换提供指导。

6.2 计划式设计

演进式设计对立面就是计划式设计。它的总体思路是，在项目构建前，就非常详细制定各种计划。有时，它被称为预先大量设计（BDUF）。

6.3 最小计划式设计#

最小计划式设计 （little design up front，Martin，2009）。这是一种介于演进式设计和计划式设计之间的一种设计方法。在计划式设计与演进式设计之间取得平衡是可能的。一种方法是做一些初步的计划式设计，确保架构可以处理一些最大风险。在初始计划式设计完成后，未来的需求变化往往通过局部设计去处理，或采用演技式设计，前提是重构、测试驱动设计和持续集成等实践在项目中已经开展起来。

**五、总结**

读完《恰如其分的软件架构》后，我对软件架构的设计与实现有了更深刻的理解。这本书不仅是对软件架构的全面指南，更是一本关于如何在复杂的技术和业务需求中找到恰当平衡的智慧之作。书中通过大量的实际案例和理论分析，帮助读者在架构设计中做出合适的决策，并强调架构选择必须与业务目标和团队能力相匹配，而非盲目追求技术的先进性或复杂性。

在现代软件开发中，架构设计不仅仅是技术层面的选择，更多的是一项战略性决策。作者通过细致的分析，展示了不同架构设计模式在特定场景下的优劣势，以及如何根据实际需求和环境做出合理的权衡。例如，书中讨论的“适度设计”理念，提醒我们避免过度设计和过度抽象，强调架构要“恰如其分”，既能满足当前需求，又能具备一定的灵活性，适应未来的变化。

对于架构师和开发人员来说，这本书提供了非常实用的工具和思维框架，帮助我们在面对日益复杂的系统时，不仅关注技术细节，更要从全局角度把控系统的演化方向。作者通过对各种架构风格的介绍，帮助读者理解如何在微服务、事件驱动、领域驱动设计等多种架构风格中选择最适合的解决方案。

总的来说，《恰如其分的软件架构》不仅是一本技术书籍，更是一种系统思维和设计哲学的体现。它通过深入浅出的方式，传达了架构设计中的核心原则，并为读者提供了清晰的决策框架。无论是初入架构领域的从业者，还是经验丰富的架构师，这本书都能为我们提供宝贵的指导，帮助我们在复杂的软件工程中做出更加明智和有效的决策。