**《恰如其分的软件架构》读书笔记**

周佳霖 2022141461189

为了对软件架构的选择有更加深刻的了解，我按照老师的要求阅读了《恰如其分的软件架构》这本书，在阅读过程中对架构设计的目标、方法和实践都有了更加深刻的理解。

《恰如其分的软件架构——风险驱动的设计方法》是一本面向软件开发人员和架构师的经典著作，聚焦于如何在项目开发中以风险为导向，合理规划和设计软件架构。本书的核心思想是：**架构设计不必追求“完美”或“复杂”，而是应当根据实际项目的风险制定恰到好处的方案**。围绕着这个思想，作者一步步结合案例讲解了如何以管理项目中的关键风险为目标来进行架构设计。这让我对软件架构有了更加深刻的思考。下面是阅读过程中我对于书本内容受到的一些启发和思考。

**1 怎么理解软件架构**

根据卡内基·梅隆大学软件工程研究所对于软件架构的定义：计算系统的软件架构是解释给系统所需的结构体集合，其中包括：软件元素、元素之间的相互关系，以及二者各自的属性。通俗的来理解，软件架构就是系统设计，以及它对性能、安全和可修改性等系统所产生的影响。因此，软件架构的选择就非常重要，将直接影响系统的质量属性。

说到软件架构的重要性，关键在于其可以影响整个系统，所以选择合适的软件架构，才可以降低风险，规避失败。其重要性可以总结为以下四点：

* 架构扮演系统骨架的角色。
* 架构影响质量属性。
* 架构与功能基本上是正交的。
* 架构是对系统的约束。

因此，我们需要正确的选择软件架构，选择能够最大程度上满足质量属性

的软件架构。

在一些特定的时候，软件架构的选择会变得尤为重要。比如，让问题非常复杂，很难设计出很好的解决方案时，选择合适的架构非常重要；当风险很高时，选择合适的架构非常重要；当设计解决全新领域的问题时，选择合适的架构非常重要；等等。

那么，进行软件架构设计的方式有哪些呢？主要是以下四个方式：

* 不预先设计，在开发者编码的过程中进行同步的设计。
* 分配设计的时间比例。
* 先进行详细的架构设计，形成详尽的架构文档。
* 随机应变。

**2 风险驱动完成架构选择**

首先，需要理解风险驱动模型，其分为以下三步：

1. 识别风险，对风险排定优先级。

从需求开始，找到难以实现的内容，不完整或令人误解的质量属性需求。同时结合工作经验，找到一些领域内的典型风险。最后把风险进行优先级排序。

1. 选择并应用一组技术降低风险

在识别了风险之后，针对风险，运用期望的技术来降低这些风险。但是，任

何特定技术都擅长降低某种风险，而对于其他风险却未必。有些风险可以通过多种技术去缓解，而有些风险甚至需要发明新的技术去解决。因此，判断使用什么技术也是非常困难且重要的。

1. 评估风险降低的程度

综上，如何在风险驱动下进行技术选型呢？首先，技术决策应该和具体需求

相匹配；其次，有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；再次，采用特定类型的模型，分析特定的问题；最后，考虑技术之间的关系。

**3 把握设计和架构之间的度**

架构设计是一种平衡的艺术，需要在当前需求和未来发展之间权衡取舍。复杂的架构设计会增加实现时间，而需求变化又快，因此架构设计应满足当前业务发展需求，并能支持未来1到2年的发展即可，无需识别所有当前和未来的风险。为了实现这一目标，可以采用**演进式设计**和**计划式设计**的结合策略。

演进式设计通过重构、测试驱动设计和持续集成等敏捷实践来应对局部设计不协调的问题。重构可以改进代码质量，测试驱动设计确保系统功能不被破坏，持续集成则为团队提供统一的代码库。然而，演进式设计可能难以应对架构规模的转换，而**计划式设计**则强调在项目开始前制定详细的架构计划，但这种预先设计容易因为需求变化而变得过时或不灵活。为此，**最小计划式设计**提供了一种折中方法，即在项目初期做必要的计划式设计以应对最大风险，然后结合演进式设计来处理后续的需求变化。这种方法依赖于敏捷实践的支持，如重构和持续集成，以确保系统的灵活性和可维护性。

此外，理解开发过程中的变化，如从传统开发模型到 DevOps 模型的过渡，也是架构设计的重要考虑点。尽管风险驱动模型能帮助决定架构设计的深度和时间，但它在准确预测设计所需时间方面仍有局限性，需要架构师根据项目具体情况灵活调整设计策略。

**4 建模**

规模越大的系统，复杂度越高，越需要进行建模。建立架构模型是一种理解和解决棘手问题的好方法，因为它可以去掉无关的细节，使得你能够将注意力放在主要部分以及相关关系上，做出预测，评估候选方案。

建模可以从领域模型、设计模型、代码模型等方面来入手进行建模。规范化模型结构顶部是抽象层次最高的模型（领域），底部模型则代表具体（代码）。指定关系和细化关系能确保模型一致性，又使得他们区分不同的抽象层次。

* 领域模型：

领域模型表达了与系统相关的现实世界的不变事实。

* 设计模型：

需要构建的系统不仅会显示在领域模型中，还会在设计模型中体现。

设计模式由递归嵌套的边界模型和内部模型组成。边界模型涉及公共接口，内部模型介绍了内部设计。

* 代码模型：

系统的源代码实现。

**5 架构风格总结**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 视图类型 | 元素和关系 | 约束/导轨 | 质量提升 |
| 分层 | 模块 | 层，使用关系，回调通道 | 只能使用相邻的下一层 | 可修改性、可移植性、可重用性 |
| 大泥球 | 模块 | 无 | 无 | 无，反而有很多损害 |
| 管道-过滤器 | 运行时 | 管道连接器，过滤组件、读写端口 | 独立的过滤器，增量式处理 | 可再配置性（可修改性）、可重用性 |
| 批量顺序处理 | 运行时 | 阶段（步骤）、作业（批量） | 独立的阶段，非增量式处理 | 可重用性、可修改性 |
| 以模型为中心（共享数据） | 运行时 | 模型、视图、控制器组件，更新和通知端口 | 视图和控制器只能通过模型进行交互 | 可修改性、扩展性、并行 |
| 分发-订阅 | 运行时 | 分发和订阅端口，事件总线连接 | 事件生产者和消费者都是耦合松散的，即忘者 | 可维护性、可扩展性 |
| 客户端-服务器和 N 层 | 运行时 | 客户端和服务器组件，请求-响应连接器 | 不对称关系，服务器独立 | 可维护性、可扩展性，现有代码集成 |
| 对等 | 运行时 | 对等组件，请求-响应连接器 | 平等的对等关系，所有节点既是服务端又是客户端 | 可用性、系统弹性、可伸缩性、可扩展性 |
| Map-Reduce | 运行时和部署 | Master、map 和 reduce workers，本地和全局文件系统连接 | 可拆分的数据集供 map 和 reduce function 使用，分配和并行 | 可伸缩性、性能、可用性 |
| 镜像、农场和支架 | 部署 | 多样化 | 多样化 | 多样化、性能、可用性 |

通过这本书，我对架构设计有了更加深刻的理解。我明白了，架构应当从项目需求和风险出发，而不是盲目选择流行的技术栈，有时候“高大上”不一定是最好的。同时，架构设计是一个迭代的过程，而非“一次性完成”，在后续的项目中，应更加注重架构的可扩展性和调整空间，而不是试图“一步到位”。

而本书的核心思想——“恰如其分”，不仅是对架构设计的要求，也是对开发团队时间、资源的尊重。书中曾多次提到过这样的思想，不同规模的项目、不同阶段的需求，决定了架构设计的深度和复杂度。对于一个小型的 MVP 项目，过于复杂的架构设计显然是不必要的，可能会拖慢开发进程。而对于一个面向亿万用户的大型分布式系统，则需要更全面的架构设计。这让我在读完本书后更加理解“恰如其分”的理念：架构设计应服务于项目，而不是成为项目的负担。

总的来说，《恰如其分的软件架构》是一本实用性极强的书，让我对软件架构，特别是风险驱动思想的软件架构学到了很多。这本书让我重新思考了架构设计的意义，也让我意识到，架构师的任务不是追求“完美”，而是找到“刚好足够”的解决方案。未来，我会尝试将书中的方法应用到实际项目中，期待能在实践中更加深入体会如何寻找恰如其分的软件架构。