## 《恰如其分的软件架构》读书笔记

《恰如其分的软件架构》旨在提供软件架构的基本概念和实用建议。书中强调风险驱动的架构设计，提倡根据风险调整架构工作的程度，以降低风险并平衡成本与收益。作者通过案例和建议展示如何管理架构建设和范围，突出了架构的民主化和积累陈述性知识的重要性。

**第一章：概述**

第1章为读者提供了软件架构的基本概念，强调了在软件系统的规模和复杂度不断增长的背景下，软件架构的重要性和作用。通过案例分析，展示了不同架构对系统功能和质量属性的影响，以及在设计过程中需要进行的质量属性权衡。同时，介绍了风险驱动的软件架构方法，为后续章节深入讨论软件架构的理论和实践奠定了基础。本章提出的内容总结如下：

1. 软件系统的增长与复杂性：

软件系统规模和复杂度的增长带来了挑战。

开发者需要新的方法来应对这些挑战，包括分治、知识、抽象等策略。

2. 分治、知识、抽象：

分治：将大问题分解为小问题，便于管理和解决。

知识：利用过去的经验来解决新问题。

抽象：通过隐藏细节简化问题，便于理解和处理。

3. 软件架构的重要性

软件架构作为解决复杂性和规模问题的关键工具。

帮助开发者分割系统、提供设计知识和理解软件的抽象。

4. 软件架构案例分析

通过比较三个不同版本的邮件日志查询系统，展示了架构对功能和质量属性的影响。

质量属性如可修改性、可伸缩性和延迟时间在不同架构中的表现差异。

5. 质量属性的权衡

提升某一质量属性可能会牺牲其他属性，需要在设计时做出权衡。

6. 概念模型

软件架构专家通过形成概念模型来更有效地理解和推理软件系统。

7. 抽象与约束

通过施加约束简化系统推理，如限制作业的副作用以便于管理和并行处理。

8. 视角转换

开发者需要接受新的架构抽象，并在适当时候放弃旧的抽象。

9. 架构师的角色

区分架构师的工作头衔、构建过程和工程制品。

10. 风险驱动的软件架构

基于风险的方法来决定软件架构的工作量，强调风险与努力之间的关联。

11. 敏捷开发者的架构

在敏捷开发中，软件架构的角色，包括恰如其分的架构和概念模型的重要性。

作者强调，尽管没有一劳永逸的解决方案（即“银弹”），但通过这些方法，开发者可以更好地分解系统、提供知识和利用抽象来揭示问题的本质。软件架构作为一种“武器”，能够帮助开发者有效地处理软件系统的复杂性和规模问题，提供设计优秀软件所需的知识和抽象，从而构建出更大规模、更复杂的系统。软件架构不是要限制创造力，而是要使开发者能够在更大的规模和复杂度上发挥创造力。

**第一部分**

本书的第一部分主要介绍了软件架构的基本概念、原理以及风险驱动方法。

**第二章：软件架构**

第二章《软件架构》深入探讨了软件架构的概念、重要性以及它在软件开发中的作用，强调了软件架构在确保软件系统成功中的中心作用，并提供了对架构概念的深入理解。该章的重点总结如下：

**软件架构的定义**

软件架构是软件系统的结构体集合，包括软件元素、元素之间的关系以及元素和关系的属性。

架构关注的是设计的宏观部分，如模块和模块间的连接方式，而详细设计则涵盖其他方面。

**架构与详细设计的区别**

架构通常涉及系统的组织、组件的组合方式、控制结构、通信协议、功能分配、物理分布等宏观层面的问题。

详细设计则更关注于具体的实现细节，如算法的选择、数据结构的实现等。

**架构的重要性**

系统骨架：架构为软件系统提供了支撑其功能和质量属性的基础结构。

影响质量属性：不同的架构对系统的性能、安全性、可修改性等质量属性有不同的影响。

与功能正交：架构与系统功能可以独立选择，但必须相互匹配以确保系统的整体效能。

系统约束：架构通过施加约束来确保系统能够达到预期的质量属性。

**架构何时重要**

在面对大规模或高复杂度的系统、高失败风险、难以实现的质量属性、全新领域或产品线架构时，架构的重要性尤为突出。

**推定架构**

推定架构是指在特定领域中占据主导地位的架构族，如IT系统中的三层架构。

推定架构通常能够很好地处理领域中的常见风险，因此被广泛采用。

**如何运用软件架构**

开发者可以采取三种对待软件架构的方式：忽视架构、欣然接受架构、提升架构。

专注架构的设计要求开发者审慎地选择软件架构，以满足软件的目标，包括功能和质量属性。

提升架构的设计则更进一步，要求架构能够保证系统的某一目标或属性。

**第三章：风险驱动模型**

第三章深入探讨了如何通过识别和应对软件项目中的风险来指导架构设计，强调了在软件开发过程中，如何通过风险驱动的方法来决定架构工作的程度和使用的技术，以及如何在计划式设计与演进式设计之间找到平衡。通过这种方法，开发者可以更有效地利用资源，降低项目失败的风险，并构建出成功的软件系统。以下是该章的重点总结：

**风险驱动模型是什么？**

核心理念：软件架构的努力应与项目中的风险成正比，即在风险最高的地方投入更多的设计和分析工作。

三个步骤：识别风险并排定优先级、选择并运用相关技术、评估风险降低的程度。

**风险驱动模型的实践**

风险识别：开发者需要识别出项目中的主要风险，这些风险可能是技术难题、管理问题或外部因素。

技术选择：根据识别出的风险，选择能够有效降低这些风险的技术和方法。

风险评估：在采用技术后，评估风险是否已经得到充分缓解，决定是否可以开始或继续编码工作。

**风险与技术的关系**

风险描述：风险应描述得足够具体，以便可以进行测试和验证。

工程风险与管理风险：区分工程风险（与产品分析、设计和实现有关）和管理风险（与进度、资源分配有关）。

技术的选择与应用

技术多样性：不同项目面临不同的风险，因此需要运用不同的技术。

最优技术集：选择最佳的技术集合来降低优先级最高的风险。

无法根除的工程风险：由于时间和资源的限制，不可能完全消除所有工程风险。

**何时停止架构工作**

付出的努力应与风险相称：只在风险最高的领域进行深入的架构设计。

不甚完整的架构设计：架构模型可能不完整，只针对识别出的风险进行设计。

主观评价：最终需要主观判断架构设计是否已经足够充分地降低了风险。

**计划式设计与演进式设计**

计划式设计：在项目开始前详细制订计划。

演进式设计：设计随着系统实现的增长而增长。

最小计划式设计：结合计划式和演进式设计，先进行少量的预先设计，然后通过演进式设计来处理需求变化。

**第四章：实例：家庭媒体播放器**

第四章通过一个具体的案例——家庭影院媒体播放器系统，展示了如何应用风险驱动模型进行软件架构设计。

**案例背景**

系统描述：家庭媒体播放器是一个能够在电视和音响中播放音乐、视频和图像的计算机系统，可以连接本地硬盘或互联网媒体源。

系统特点：除了基本播放功能，还支持音乐和图片的幻灯片播放、视频播放、元数据收集和第三方扩展。

**面临的挑战**

团队沟通：随着项目的扩展，新加入的异地开发者需要理解系统的设计和架构，以避免架构侵蚀。

COTS组件集成：需要将第三方商业组件集成到系统中，以支持多平台运行。

元数据一致性：需要确保内部元数据表示与互联网上的元数据兼容，以便第三方库可以扩展系统。

**风险驱动方法的应用**

识别风险：识别了与团队沟通、组件集成和元数据一致性相关的风险。

选择技术：针对每个风险，选择了相应的架构和设计技术来降低风险。

评估风险降低：评估所选技术是否充分降低了风险，并决定是否可以开始编码。

**具体实施步骤**

模块模型：创建模块模型帮助新团队成员理解系统的结构和组件依赖。

质量属性与设计决策：讨论了质量属性的优先级和设计决策，如用户界面响应、播放平滑度、可靠性等。

运行时模型：描述了运行时组件和连接器，以及它们如何协作。

功能场景：通过功能场景展示了指令和播放视频的执行轨迹。

**反思**

沟通与文档：强调了与团队沟通项目设计的重要性，并讨论了如何通过文档降低风险。

模型与代码的差距：认识到模型与代码之间的差距，并意识到需要覆盖主要模型和视图类型来有效沟通设计意图。

**第五章：建模建议**

第五章为读者提供了关于软件架构建模的实用建议和深入见解，提供了关于如何有效使用架构模型以降低项目风险的深入讨论，并强调了在架构设计中考虑风险、质量属性和合理决策的重要性。同时，也指出了在实际应用中可能遇到的挑战，并提供了相应的建议和解决方案。

**专注于风险**

风险驱动设计：强调了在架构设计中专注于风险的重要性，建议开发者根据项目中的风险来选择合适的架构技术和投入相应的时间。

**理解你的架构**

架构知识内在化：鼓励开发者将架构的知识和抽象内在化，以便在构建新系统或分析现有系统时能够运用这些理解。

架构与质量属性：讨论了架构师如何理解架构选择对系统质量属性的影响，并如何评估这些选择的合理性。

**传播架构技能**

团队架构意识：提倡在团队中传播架构技能，使得每个成员都能理解整个架构，提高沟通效率，并作出可靠的决策。

架构知识共享：指出了集中或囤积架构知识的问题，并强调了开发者拥有架构意识的重要性。

**作出合理的架构决策**

权衡与优先级：讨论了如何在不同质量属性之间进行合理的权衡，并确保架构决策与项目的质量属性优先级相匹配。

透明化决策过程：建议通过模板表达设计方案的理由，使得决策过程更加透明，便于解决团队内的分歧。

**避免预先大量设计（BDUF）**

BDUF的风险：讨论了预先大量设计的风险，包括工作内容可能并非问题所在、纸上谈兵和瞎子点灯。

BDUF的适用场景：指出了在某些情况下，如大型项目或对质量要求苛刻的项目，BDUF可能是最佳选择。

**避免自顶向下设计**

自顶向下设计的问题：指出了自顶向下设计可能忽视底层模式和重用现有组件的困难。

自底向上的洞察力：强调了从细节到抽象层面都需要具备的洞察力，并鼓励利用这种能力。

**余下的挑战**

风险识别与评估：讨论了识别和评估风险的挑战，以及如何使用风险驱动模型来改进其他方法。

架构候选方案评估：描述了评估不同架构候选方案的困难，以及如何根据不完全的信息作出决策。

**特性和风险：一个故事**

风险与特性的比较：通过一个故事比较了基于特性驱动的开发和风险驱动模型，强调了在设计中考虑风险的重要性。

**第二部分**

**第六章 工程师使用模型**

第六章深入探讨了模型在工程实践中的应用，特别是在软件架构领域中的重要性和作用，强调了模型在软件工程中的核心作用，特别是在处理复杂系统时。模型不仅帮助工程师理解和设计系统，还提供了一种沟通和记录架构决策的方式。通过模型，工程师能够更好地预测系统行为，评估不同设计方案，并最终构建出更高质量的软件。

**模型的必要性**

规模与复杂度：随着软件系统的规模和复杂度增加，工程师需要依赖抽象模型来理解和解决问题。

抽象的力量：抽象模型提供了深入洞察问题本质的能力，并为解决这些问题提供了手段。

**模型的功能**

洞察力与杠杆效应：模型帮助工程师识别潜在的冲突、瓶颈和性能问题，从而在设计和决策中获得优势。

分析系统质量：通过模型，工程师可以分析系统的质量属性，如性能、可伸缩性和安全性。

**模型的构建与使用**

模型的简化：有效的模型会忽略不相关细节，专注于对解决问题至关重要的方面。

模型的增强推理：模型不仅是理解系统的工具，也是增强推理和预测系统行为的手段。

**模型的类型与应用**

领域模型：描述了与系统相关的现实世界的不变事实。

设计模型：描述了构建系统的设计决策和机制。

代码模型：描述了系统的源代码实现。

**模型之间的关系**

指定与细化：指定关系确保模型之间的一致性，细化关系则展示了模型在不同抽象层次上的细节。

**模型的视图**

视图的定义：视图是模型的一个子集，它展示了模型的特定方面，使得复杂模型更易于管理和理解。

视图的一致性：所有视图都应与主模型保持一致，确保在模型更新时视图也相应更新。

**模型的实际应用**

问题解决：工程师使用模型来解决实际问题，从定义问题到找到解决方案，再到实施和验证。

模型的迭代：在实践中，模型是迭代发展的，随着对问题理解的深入，模型也会不断进化。

**第七章 软件架构的概念模型**

第七章深入探讨了软件架构的概念模型，包括其组成、用途以及如何通过模型来理解和分析软件系统。

**规范化模型结构**

模型范围：介绍了从抽象到具体的三个主要模型：领域模型、设计模型和代码模型。

指定与细化关系：解释了如何使用指定（designation）和细化（refinement）关系来确保模型之间的一致性和区分不同抽象层次。

**领域模型**

不变事实：领域模型描述了与系统相关的现实世界的不变事实，如类型定义、关系和行为。

信息模型：讨论了如何通过类型列表和定义来构建信息模型，以及如何图形化地表达这些模型。

**设计模型**

系统设计：设计模型描述了构建系统的设计决策和机制，包括组件、连接器和架构风格。

视图：介绍了如何使用视图来展示设计模型的特定方面，以便更好地理解和分析。

**代码模型**

源代码实现：代码模型关注于系统的源代码实现，包括代码的结构和组织。

模型与代码的关系：讨论了代码模型如何与设计模型相联系，以及如何确保代码实现与设计承诺的一致性。

**主模型与视图**

主模型概念：强调了所有视图都应与主模型保持一致，以及如何通过视图来展示主模型的特定细节。

视图一致性：讨论了视图之间的一致性，以及如何确保视图更新与主模型的变更同步。

**组织模型的其他方式**

业务建模：讨论了业务建模与领域建模的关系，以及为什么本书选择专注于领域模型。

UML的用法：探讨了统一建模语言（UML）在架构建模中的应用，以及如何根据需要调整UML的使用。

**小结**

模型的重要性：总结了模型在理解和构建软件架构中的关键作用，以及如何通过模型来降低项目风险。

**第八章 领域模型**

第八章深入探讨了领域模型的概念、构建方法以及它在软件开发和架构设计中的作用。

**领域模型的定义和作用**

概念解释：领域模型是对现实世界中与软件系统相关的部分的抽象表示，它包括了系统中的关键概念、实体、关系和规则。

目的：领域模型的目的是捕捉和表达业务领域的本质，为软件开发提供一个共同的理解基础。

**领域模型的组成部分**

信息模型：包括类型（types）列表和定义，描述了领域中的核心概念和它们之间的关系。

导航和不变量：领域模型中的导航（navigation）指的是在类型之间移动的能力，而不变量（invariants）则是必须始终为真的条件或规则。

快照和场景：快照（snapshots）是领域模型中某一时刻的状态图，场景（scenarios）则描述了状态随时间变化的一系列事件。

**构建领域模型**

方法和技巧：介绍了如何通过信息模型、快照和场景来构建领域模型，以及如何使用这些模型来探索和验证业务规则。

实例化：通过具体实例来展示领域模型中的类型和关系，帮助更好地理解和沟通领域知识。

**领域模型与软件架构的关系**

架构决策：领域模型为软件架构提供了信息基础，帮助架构师做出更符合业务需求的决策。

风险管理：通过领域模型可以识别和解决潜在的业务规则冲突，减少项目失败的风险。

**领域模型的应用**

与业务专家的沟通：领域模型作为一种语言，帮助开发人员与业务专家进行有效沟通，确保软件设计符合业务需求。

迭代和演化：领域模型是迭代发展的，随着对业务领域的深入理解，模型也会不断演化和完善。

**小结**

重要性：强调了领域模型在软件开发中的重要性，它不仅是业务知识和规则的载体，也是软件开发团队之间沟通的桥梁。

**第九章**

第九章专注于软件架构中的设计与设计模型的构建，它解释了设计模型是什么、如何创建以及如何使用这些模型来表达软件系统的设计。

**设计模型的基本概念**

设计模型定义：设计模型是对软件系统的设计进行建模的蓝图，包括系统如何操作领域内的实体和概念。

与领域模型的关系：设计模型基于领域模型中的事实，并将这些事实映射到系统的具体实现中。

**设计模型的组成部分**

视图和视图类型：设计模型可以通过不同的视图来展现，每个视图关注模型的不同方面，如组件视图、部署视图等。

封装和边界：设计模型使用封装来隐藏实现细节，只展示系统的接口和边界模型，即系统的外部可见部分。

**设计模型的详细元素**

组件和连接器：设计模型中的组件代表系统中的主要计算元素，而连接器则表示组件之间的交互路径。

端口和职责：端口是组件上用于交互的点，职责则描述了组件的功能和责任。

**设计模型的构建和使用**

构建设计模型：介绍了如何构建设计模型，包括确定组件、定义连接器、划分端口和分配职责。

使用设计模型：讨论了如何使用设计模型来指导系统的开发和维护，以及如何通过模型来沟通设计决策。

**设计模型与质量属性**

质量属性场景：描述了如何使用设计模型来分析和确保系统的质量属性，如性能、安全性和可维护性。

架构驱动：探讨了质量属性如何驱动架构决策，以及如何通过设计模型来满足这些驱动因素。

**设计决策和权衡**

设计决策：讨论了在设计过程中需要做出的关键决策，以及如何记录和传达这些决策。

权衡：分析了在不同质量属性之间进行权衡的常见情况，以及如何在设计中反映这些权衡。

**设计模型的实例化**

组件装配：展示了如何将设计模型中的组件和连接器实例化，以反映系统的实际运行时结构。

功能场景：通过具体的场景来说明设计模型中的组件如何协作完成特定的功能。

**第十章**

第十章深入探讨了软件架构中的代码模型，即系统的源代码实现，以及它与架构模型之间的关系。

**模型与代码的差异**

词汇和抽象差异：架构模型和源代码使用不同的词汇和抽象级别来描述系统的不同方面。

设计承诺：架构模型提供了部分承诺，而源代码必须提供完全的承诺，包括系统的可执行性。

内涵与外延：架构模型包含内涵和外延特性的元素，而代码主要包含外延特性。

**模型-代码差异的管理**

机制和人为控制：讨论了通过机制（如模型驱动工程）和人为控制（如开发人员的理解）来管理模型和代码之间的差异。

同步策略：介绍了不同的策略来保持架构模型和源代码之间的同步，包括忽略分歧、临时建模、概要模型、里程碑同步等。

**架构明显的编码风格**

嵌入架构线索：提倡在源代码中嵌入关于系统架构模型的线索，以降低丢失设计意图的风险。

模型嵌入代码原理：在代码中表达模型有助于理解和演化系统。

**在代码中表达设计意图**

有意提示和按合约设计：讨论了如何通过方法命名、前置条件和后置条件等手段在代码中表达设计意图。

软机制和硬机制：区分了依赖于人为解释的软机制和可以用机器检查的硬机制。

**模型嵌入代码原理的应用**

代码中的领域模型：讨论了如何在代码中映射领域模型，以及这种做法的好处和挑战。

技术债和分歧：探讨了代码与模型之间的分歧，以及如何通过减少分歧来管理技术债。

**表达架构设计意图的模式**

组件类型、连接器类型和端口类型：介绍了如何在代码中具体化这些架构概念，以提高代码的可读性和可维护性。

协议和属性：讨论了如何在代码中表达架构协议和属性，以及使用注解和命名约定的方法。

**第十一章**

第十一章探讨了软件架构中封装和分割的重要性，以及如何通过有效的模块化和组件化来提高软件的可理解性和可维护性。

**多层级故事**

系统结构化：讨论了如何通过层级嵌套的方式来结构化系统，使得每个层级都能讲述部件如何交互的故事。

认知负担：分析了多层级故事如何帮助开发人员应对大规模和复杂性，同时指出了维护这种结构化所带来的成本。

**层级和分割**

分割概念：介绍了分割系统为分散块的方法，以及如何通过层级嵌套来组织系统。

主分解：讨论了在系统组织中选择主关切的重要性，以及这种选择如何影响其他关切的解决。

**分解策略**

功能、典型类型、架构风格：探讨了不同的分解策略，包括基于功能、典型类型和架构风格的分解。

质量属性和属性驱动设计：介绍了如何基于质量属性来驱动模块的递归设计。

**有效封装**

封装的重要性：讨论了封装如何降低复杂性，以及有效封装和低效封装的区别。

帕纳斯模块：介绍了David Parnas关于创建稳定且有效封装模块的方法。

**创建封装接口**

抽象数据类型（ADT）：通过栈的例子解释了ADT的概念，并讨论了如何将模块和组件视为ADT。

接口描述：讨论了如何创建包含操作签名、前置和后置条件的接口描述，以及如何通过类型模型来理解操作。

**模型元素**

核心元素集：描述了制作架构模型需要用到的元素词汇，如模块、组件、连接器、端口等。

**小结**

封装和分割的选择：总结了封装和分割对系统质量的影响，以及如何通过有效的封装来降低认知压力。

**第十二章**

第十二章深入探讨了构建软件架构模型时使用的关键元素，这些元素是理解和表达软件架构的基础。

**模型元素概览**

核心元素集：介绍了构成架构模型的基本元素，包括模块、组件、连接器、端口、角色、质量属性、原理、环境元素、场景、不变量、权衡及风格等。

**部署相关元素**

环境元素和通信通道：讨论了软件如何在硬件上运行以及硬件如何部署在物理位置，包括节点、连接以及环境元素的概念。

**组件**

组件的定义和类型：组件作为系统中的主要计算元素和数据存储，讨论了组件类型与实例的区别，以及组件与模块的关系。

子组件和实现：探讨了组件的嵌套性质，即子组件的使用，以及组件的实现细节。

组件建模中的不确定性和含糊性：讨论了在组件建模中常见的误解和不确定性。

**组件装配**

组件和连接器图：描述了如何通过组件装配图展示组件、端口和连接器的实例或类型，以及它们如何组成更大的系统。

系统上下文图：介绍了系统上下文图的作用，展示了系统作为一个组件实例与外部系统的关系。

**连接器**

连接器的定义和重要性：连接器作为组件间通信的通道，讨论了连接器在架构中的重要性和作用。

连接器的类型和属性：介绍了连接器的不同类型，如过程调用、事件、管道等，以及连接器的属性，如性能和安全性。

**设计决策**

关键性设计决策：讨论了在软件架构中关键性设计决策的作用，以及如何记录和传达这些决策。

**功能场景**

功能场景的描述：功能场景描述了系统行为，展示了系统元素如何随时间变化和交互。

**不变量（约束）**

不变量的概念：不变量是对系统的限制，要求系统必须怎么做或严禁做什么，包括静态不变量和动态不变量。

**模块**

模块的定义和属性：模块是实现制品的集合，讨论了模块的属性和它们如何提供或依赖服务。

**端口**

端口的作用：所有进出组件的通信都是通过端口进行的，讨论了提供的和依赖的端口，以及端口的多实例问题。

**质量属性**

质量属性的理解：质量属性是超功能需求，讨论了它们与功能正交的性质，以及如何在架构中分析和满足这些属性。

**第十三章**

第十三章《模型关系》深入探讨了软件架构中模型之间的关系，这些关系有助于我们更精确地理解和表达架构元素之间的相互作用。以下是该章节的主要内容总结：

**模型关系的重要性**

理解关系：阐述了理解模型关系对于补充架构概念模型的重要性，以及它们在发现模型中潜在缺陷中的作用。

**九种模型关系**

投影（视图）关系：描述了如何通过投影来展示模型的特定细节，以及如何维护视图间的一致性。

分割关系：解释了如何将整体模型分割成不相交的部分，以及分割的特性和应用。

组合关系：讨论了如何通过组合更小的模型来创建更大的模型，以及组合与分割的对比。

分类关系：介绍了如何使用分类系统对模型元素进行归类，以及分类系统的三个理想特性。

泛化关系：描述了类型如何归入另一个类型，以及泛化和子类型的概念。

指定关系：讨论了如何在不同域之间建立桥梁，例如现实世界与问题域模型之间。

细化关系：解释了同一样东西高细节和低细节表现形式之间的关系，以及细化的不同语义。

绑定关系：讨论了如何将模型元素与模式或风格的占位符相关联。

依赖关系：描述了一个模型的改变如何影响另一个模型。

**模型关系的实践应用**

使用关系：通过一个房屋和车库的系统例子，展示了如何将这些关系综合起来使用，以及它们在实际模型中的应用。

**模型关系的挑战与优势**

挑战：讨论了使用模型关系时面临的挑战，如细节的缺失可能导致设计问题。

优势：强调了模型关系在降低复杂性和缩小规模方面的优势，使得架构问题更易于处理。

第十四章

**约束**

尽管使用约束或风格可能会增加设计的复杂性，但它们对于确保软件系统的质量和可维护性是至关重要的。通过使用风格，可以提高沟通效率、促进设计重用，并确保系统的质量属性。

**模式和风格的目的**

模式和风格用于解释和沟通设计过程中的概念，以及提供设计能力，如管道-过滤器风格允许过滤器的重新配置。

权衡的例子：在管道-过滤器风格中，理论上过滤器应独立并通过管道通信，但实际中第一个和/或最后一个过滤器可能会违反这一约束，影响风格的某些特性。在客户端-服务器风格中，柏拉图式风格要求服务器不知道客户端的存在以实现解耦，但实际中服务器可能会主动推送数据给客户端，影响解耦性。

架构风格和模式在实际应用中的体验式版本可能与理想化的柏拉图式版本有所不同，这些差异需要在设计和沟通中得到妥善处理。

**模式对风格**

强调了区分架构模式和架构风格的重要性，并指出模式通常针对设计中较小的层面，可以在一个设计中出现多种，而系统通常只有一个主导的架构风格。同时，也提到了在某些情况下，架构风格和模式之间的界限可能变得模糊，特别是在大型系统中，原先的架构风格可能成为更大系统中的一个架构模式。建议不必过分纠结于术语的分类，可以将架构模式和架构风格视为同义词。

**风格目录**



**分层**：分层架构风格以层和使用关系为核心元素，通过约束提升系统的可修改性、可移植性和可重用性。存在变体如跳层和共享层，影响层定义和质量属性。柏拉图式分层风格与实际体验式风格有差异，但即使在违反约束的情况下，分层仍有助于模块化和功能分组。回调机制允许层间安全通信，保持解耦。

**大泥球：**大泥球风格特征为缺乏清晰结构和坏结构的遗留，表现为随意的信息共享和全局数据结构滥用。这种风格通常起源于模块视图类型的混乱，并逐渐扩散至其他视图类型。由于缺乏概念完整性和一致性，导致技术债务累积，系统可维护性和可扩展性差。尽管被视为反面模式，大泥球风格在某些情况下反映了软件工程中的“刚刚好”策略，即并非所有系统都需要过度设计。一旦系统陷入大泥球状态，某些开发人员可能因其复杂性而获得安全感和威信，导致问题持续恶化，难以重构。

管道-过滤器：管道-过滤器架构风格通过管道传输数据至过滤器进行处理，类似于化学处理厂的液体流动。这种风格的关键特征是持续、增量式地处理数据，与批量处理风格不同。它由管道、过滤器、读端口和写端口组成，要求过滤器独立且增量式处理数据。管道-过滤器风格支持网络的延迟组合、可修改性、再配置性和重用性，适合并行处理，但不太适合交互式应用。实现时需考虑停止条件和管道缓冲限制。

**批量顺序处理：**该架构风格通过将数据从一个阶段增量式地传输到下一个阶段进行处理。与管道-过滤器风格不同，每个阶段处理完所有任务后才将输出写入下一阶段，数据通常以文件形式存储在磁盘上。该架构中的每个阶段独立处理输入数据，不依赖于前一个阶段。阶段之间通过简单的连接器（如磁盘文件）传递数据，通常是线性顺序排列的。批量顺序处理架构具有较高的可修改性，因为阶段之间独立，但由于缺乏并行执行，系统的并行处理机会较少，且输出可能为空或不完整，影响可用性。然而，该架构也因其简单性和较高的吞吐量而在某些应用中具有优势。

**以模型为中心：**该架构风格通过一个中心模型（数据仓库）与多个视图和控制器组件交互，视图和控制器独立工作，仅通过模型交换数据。该架构提高了系统的可修改性和可扩展性，因为组件之间耦合度低，便于独立修改或添加。模型集中管理状态，便于持久化，并支持并发执行。常见于如MVC和观察者模式等设计模式，适用于不确定未来系统配置的场景。

**分发-订阅：**该架构风格通过事件总线将事件的发布者和订阅者解耦。发布者发布事件，但不关心谁接收或如何处理；订阅者订阅事件，但不关心事件的发布者。事件总线作为连接器，负责递送事件并支持多个订阅者。该架构提高了系统的可维护性和可扩展性，因为可以在不改变发布者的情况下，添加新的订阅者或发布者。然而，事件总线引入了性能开销，且事件的生产者和消费者之间的交互被间接化，可能影响系统的性能。

**客户端-服务器：**架构风格中，客户端发起请求，服务器响应。客户端知道服务器的身份并向其请求服务，而服务器在接收到客户端请求前并不知道客户端的身份。该风格支持同步或异步连接，且通信可以是有状态或无状态的。变体包括允许服务器在建立连接后向客户端推送更新（如IMAP协议），以及多层架构，其中多个客户端-服务器实例形成层次结构，每层有专门功能。该架构提高了系统的可维护性和可扩展性，因为服务器端的更改不需要影响客户端，便于集中控制和业务规则的修改。与模型为中心的风格类似，客户端-服务器架构也强调客户端与服务器的分离，但不限制客户端之间的交互。

**对等：**对等架构风格中，所有节点是对等的，既可以作为客户端也可以作为服务器，形成一个无层级的对称网络。每个节点既能发起请求，也能提供服务，节点间的通信不受分层限制。该架构强调减少不对称性，支持灵活的连接和断开。对等系统具有较高的可用性和弹性，因为即使部分节点离线，系统仍能保持运行。对等网络能够高度扩展，支持大规模节点（如BitTorrent、Skype），且不依赖于单点或集中基础设施。其主要优点包括可伸缩性、可扩展性和系统容错性，尽管网络孤岛可能削弱这些优点。

**Map-reduce：**Map-reduce架构适用于大数据处理，将计算分布到多台机器上，提升性能和可伸缩性。数据拆分后由Map Worker处理并生成中间结果，Reduce Worker合并结果。开发者只需编写map和reduce函数，框架处理并行计算和容错。优点：可伸缩性——支持从单机到大规模集群。可用性——机器故障时可恢复。简化开发——开发者只需关注单个任务的实现。性能受数据局部性影响，通常与批量处理结合使用。

**镜像：**两台硬件并行运行，确保高可用性。若一台宕机，另一台继续工作。尽管可靠性降低，但可用性显著提升。

**支架：**商用服务器垂直排列，接入同一网络。支架内的计算机通信更快，适合高带宽需求的场景。

**服务器农场：**多个计算机集中在一个房间内，通常由多个支架组成。它提供大规模资源，易于扩展，常用于支持无状态应用，如Web服务器和中间层。

**第十五章**

**理想的模型具有的特性**

充分精确：模型应根据需要选择合适的精度，避免过度精细或过于粗略的模型。精确度的选择取决于模型要解决的问题，过度精确可能带来不必要的成本。

准确性：模型应与现实世界一致，并保持内部一致性，避免矛盾和错误。准确性包括与现实世界的一致性、内部一致性、引用完整性和可证伪性。

预言能力：模型应能预测系统的未来行为，帮助发现潜在问题（如死锁或不兼容）。有预测能力的模型能够降低风险，并增强分析能力。

促进理解：模型应通过排除不必要的细节和集中注意力来提高可理解性。使用合适的格式和符号，避免复杂或多余的细节，使模型对目标读者更加友好。

细节一致性：模型应保持细节的一致性，不同部分不应有不同的细节级别，避免混淆。细化应根据需要进行，但要明确说明细节的选择理由。

单主题视图：每个模型视图应集中于单一主题，避免在同一视图中展示过多无关信息，从而简化分析和理解。

经济性：模型应根据风险进行投资，不应过度投入。应选择成本最低、能有效解决问题的模型，从而在解决风险的同时节省资源。

**视图创建和使用中的挑战**

特定视图产生窄焦点：每个视图专注于一个特定的焦点，可能导致对整个系统的理解变得困难。视图之间的整合和全局视角可能丧失。

问题落在视图之间：某些问题可能跨越多个视图的边界。例如，内燃机设计中的热与压力之间的权衡可能无法在单一视图中体现。

视图之间的不一致：多个视图可能存在不一致，导致不同视图所展示的信息与主模型不符，需要追溯并解决这些不一致。

**避免的反面模式**

喜欢的视图：开发人员可能沉迷于自己偏爱的视图类型，忽视其是否适合当前问题。应根据问题选择合适的视图，而不是依赖单一视图。

一张图定了所有规则：尝试在一张图中展示所有信息往往导致图表复杂且难以理解。不同的视图类型应分别呈现不同的关注点，避免图表过于拥挤。

**改善视图质量**

1. 功能场景串联视图

功能场景通过将不同视图中的元素组合成一个连贯的整体，帮助读者理解系统。场景可以跨视图类型使用，通过结构化的写作方式，使得每个步骤清晰描述目标模型的变化，并且每个步骤的发起者、活动和引用都需要明确。

2. 让功能场景动起来

让功能场景“动起来”意味着开发人员在心中模拟场景的每一步，检查每一步对模型的影响。这有助于识别不一致和遗漏，确保模型的完整性和合理性。通过这种“心理演练”，开发人员可以验证模型中的通信、状态变化及变体等问题。

3. 编写活动规范

活动规范通过明确前置条件和后置条件，将不同视图串联起来。它帮助开发人员限定模型的细节，只包含场景所要求的内容，从而提高模型的聚焦和精确度。虽然编写活动规范需要时间，但它有助于提高建模能力，并确保模型的一致性和完整性。

**提高图的质量**

1. 在图上加图例

每张图都应该有图例，尤其是在使用标准符号（如UML）时。图例有助于理解符号的含义，避免读者混淆。它还促使设计者更加仔细地选择和使用符号，并确保每个元素的意义明确。图例不仅提供了符号的定义，还能表达设计者的思路和质量目标。

2. 不要画连接器上的箭头

避免在连接器上使用箭头，避免过度简化图的表达。箭头可能会引发误解，特别是当连接方向不清晰时。通过使用不同的端口和连接器图形来表达细节，并结合文字描述，可以更准确地传达信息，而非依赖箭头。

**架构分析技术精要**

人类分析能力：人脑在架构分析中尤为关键，标准视图有效，但领域特定视图更有帮助。避免复杂和装饰性设计。

非正式分析技术：质量属性讨论会——识别和优先排序风险。架构检查单——确保已考虑已知风险。

架构评审与ATAM：专家评审与架构权衡，发现设计缺陷与风险。

正式分析技术：通过工具分析架构模型，需将模型转化为工具能理解的格式，验证断言并确保模型准确性。

特定分析：安全、可靠性与性能分析：识别漏洞、避免单点失败、评估系统性能。

**几种可能的架构不匹配**

1、对组件的假设：基础设施，谁有控制权，数据使用。

2、对连接器的假设：协议，传输数据的结构。

3、对系统的假设：系统的拓扑结构，组件是否出现。

4、对建造过程的假设：初始化的次序。

**模型抽象与选择**

构建系统模型时，必须在细节与抽象之间做出选择。对于API模型，虽然它具体且可测试，能直接与源代码对照并检测问题，但它可能规模庞大且难以维护，且容易忽视架构的整体视角。相较之下，更抽象的模型在处理大规模系统时更具可操作性，尤其是解决性能或吞吐量等架构问题时。构建模型时，应根据要回答的问题选择合适的抽象级别，以确保模型既简洁又高效。

**构建现有系统的架构模型**

在构建现有系统的架构模型时，关键是明确模型的目的和要回答的问题。建模可以帮助更好地理解系统、评估架构变更对质量属性的影响、为外部团队撰写文档，或检查系统与参考架构的兼容性。为了确保有效性，应限制模型的范围，只聚焦于解答特定问题。

然而，建模现有系统时可能面临挑战，尤其是当系统设计不清晰或杂乱时。模型能够揭示系统的复杂性，但如果设计混乱，建模可能只会暴露出系统的依赖和通信路径。因此，预先确定模型的目标和细节级别是至关重要的。

**第十六章**

**架构建模的挑战**

架构建模面临多个挑战，主要集中在架构抽象的适用性、建模技术以及模型的有效性三个方面。

**架构抽象的适用性**

架构抽象与编程语言抽象难以完全对齐，尤其是在现有代码和框架中，可能会遇到实现障碍。对于旧系统，常常需要将其视为一个大模块集合，而不是深入建模其内部细节。

面向对象和非面向对象语言的差异增加了抽象转换的难度，且框架的动态特性更难精确建模。

**建模技术**

停止建模的时机：决定何时停止对功能建模是一个挑战，过度建模会浪费资源，且有机会成本。建模应根据风险和需求深入，而不是过度关注细节。

动态架构：动态架构系统在运行时的结构变化难以建模，现有工具和符号尚不完备。

视图一致性：多视图建模需要确保不同视图之间的一致性，手工维护视图一致性可能会导致错误或遗漏。

横向关切：设计思想（如并发）可能横跨多个组件和抽象，处理这些问题常常复杂且挑战性大。

**有效建模**

提升细节：选择适当的细节来提升架构模型的精确度是一个难题。过多或过少的细节都会影响模型的有效性，且没有明确的指南来选择最佳细节。

预言能力：架构模型在预见系统问题时具有挑战性，特别是对性能等方面的预言，精确建模往往代价高昂。

**聚焦质量属性**

软件架构应优先考虑质量属性（如性能、安全性、可修改性），因为架构决定了这些特性的可实现性。虽然任何架构都能支持功能，精心设计的架构才确保质量需求得到满足。改变架构代价高昂，因此早期考虑质量属性能避免后期的高成本。高风险场景下，应采用架构为中心的设计方法，将质量属性融入架构中，确保功能实现的同时满足质量需求。

解决问题，而不是仅仅对它们建模

通过建模来解决复杂性和规模带来的问题。这是一条围绕着代偿图的长路，尽管它不能直接解决问题，但能给你以帮助。

**使用导轨一样的约束**

架构约束可以引导系统朝预期方向发展，这种方法称为以架构为中心的设计。通过明确的约束，可以确保系统满足特定目标，如可伸缩性、可靠性等。例如，Rackspace公司选择map-reduce架构以提高可伸缩性，家庭媒体播放器系统通过隔离不可靠组件保证系统稳定性。引入约束需要权衡，可能会增加复杂性或降低效率，但它帮助系统更好地实现目标并减少不确定性。

约束不仅能控制系统方向，还能作为分析工具，例如Android通过约束应用访问系统服务提高安全性。在选择引入约束时，需从无约束开始，明确目标，权衡实现路径，最终决定适当的约束来保证质量和特性。

**使用标准架构抽象**

在开发大型系统时，标准化的架构抽象有助于开发人员之间的有效沟通。尽管编程语言处理对象、类等细节很有效，但当涉及到系统架构时，术语和符号往往不统一，导致沟通困难。缺乏共同的架构抽象，开发人员可能难以进行有效交流，甚至根本无法沟通。

架构抽象不仅有助于沟通，还能帮助开发人员从整体视角思考系统设计，识别潜在问题。没有这些抽象，设计的全貌可能无法呈现，问题也不易显现。因此，开发人员应学会使用架构抽象，它们像工具箱中的新工具，帮助应对日益复杂的系统设计挑战。