注：本笔记为对于《恰如其分的软件架构》一书重点内容的整理总结，包括但不限于重要知识点整理，自己的反思与评论。

软件架构

一、概述

1.1 分治、知识与抽象

分治 将困难问题分割成小的部分，并且在组合成整体的时候减少每个部分的耦合性（即每个小模块可以分开单独处理）

知识 软件开发者运用先前问题中习得的知识来解决当前问题

抽象 抽象能够精简问题，问题越小越容易理解

1.2 对于Rackspace三种案例的反思

质量属性 不同版本系统展现不同的可修改性（modifiability）、可伸缩性（scalability）、延迟时间（latency），这三个部分一般相互制约，我们需要在其中找到平衡点，实现整体系统的最优

概念模型 作为软件架构师，需要识别并掌握每个系统运用的架构模式，了解每种模式能够满足哪些质量属性，针对领域因素、设计选择及实现细节尽心那个分类整理，并建立联系

抽象与约束 处理好抽象和具象之间的关系，具体问题具体分析的同时不要过度去注意追究细节问题，可能会降低效率

二、软件架构

2.1 何为软件架构

架构和详细设计的区别

> 随着软件系统规模与复杂度的增长，整个系统结构的设计与规格说明书变得更为重要，甚至超过对算法与运算数据结构的选择。系统的结构问题包括：系统的组织，如组件的组合方式；整体的控制结构；用于通信、同步及数据访问的各种协议；针对设计元素的功能分配；设计元素的组合方式；物理分布；伸缩能力及性能；演进的维度；在多个可选设计方案中作出选择。这些都是**软件架构**层面的设计。

定义（源自SEI）：元素、关系及属性

> 计算系统的软件架构是解释该系统所需的结构体的集合，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。

2.2 软件架构为何重要

架构扮演者系统骨架的角色

架构不仅仅指外部可见的主体部分，还指不可见的部分（如约束），列入，锁策略，内存管理策略或者继承第三方组件的技术。

架构影响质量

系统架构不仅要支持所需的功能，同时还能够促进或抑制诸如安全或性能等系统质量。

架构与功能(基本上)是正交的

对系统架构的选择与功能是相互独立的，只要付出努力，所有的架构都能造出各种系统，但是糟糕的架构角色会给功能与质量属性的实现带来障碍。

架构约束程序

为了实现性能或安全等质量要求，可以主动对自己的设计进行约束。

系统不能做什么与系统能做什么同等重要，我们需要明示系统不能做的事情。

优势：

1. 体现判断：通过对设计进行约束，就可以指导其他工程师接受解决方案，无须完整地传递他们所拥有的知识。
2. 促进概念完整性：运用一个始终如一的好主意胜过在各种不必要的方面进行创新
3. 降低复杂度：作为概念完整性的必然结果约束可以化繁为简，从而使得就此构建的系统具有显而易见的基本原则。
4. 理解运行时行为：化繁为简之后，可以更方便地审查源代码，让运行时的行为变得显而易见。

2.3 架构何时重要

五种存在架构高风险的特定案例：

小的解空间 、**高的失败风险**、**难以实现的质量属性**、**全新的领域**、**产品线**（共享通用架构）

2.4 推定架构

推定架构：在特定领域中占据主导地位的架构族，能够很好的处理领域中的常见风险

范例：

操作系统中使用协作进程，以支持系统的长期运行，即使运行其上的软件出现了故障，也能够正常恢复。尽管操作系统在许多方面存在差异，但是几乎所有的操作系统都由内核(kernel)及一组相互协作的系统进程构建而成。通过在各个独立进程中运行任务的方式，就可以将出现在单个任务中的故障与其他任务隔离开来，然后就可以重启该任务，从而使整个系统的功能维持原状。

2.5 如何使用软件架构

2.5.1 架构无关的设计

适用于低风险项目，即使不关注架构也能轻易搭建起来。

缺点：如果开发团队对架构愿景缺乏共识，即便是具有合适架构的系统随时间推移也会变成架构不合适的系统。一旦复杂度增加系统就容易崩溃。

诸如服务总线(service bus)及关系数据库等成熟而又强大的现成连接器(connector)和组件(component)在一定程度上缓解了架构无关的设计方式的弊端。

2.5.2 专注架构的设计

包括**功能范畴**和**质量范畴**

（计算债券利息）（如何扩展到支持成千上万用户规模）

在大型项目中记录架构十分有用，而在小型项目中则无关轻重。

2.5.3 提升架构的设计

当采用提升架构的设计方式时，开发者会以保证系统的**某一目标或属性为目的**去设计架构。

2.6 大型组织中的架构

企业架构：设计一个生态系统，位于其中的每个应用系统都将为整个企业做出自己的贡献。这些内容通常包括系统集成、对跨区域和跨市场的多样性支持及部署环境的标准化。

应用架构：负责单个系统的开发，设计系统的功能。

三、风险驱动模型

工程学的标志就是高效地 利用包括时间在内的各种资源。

3.1 定义

风险驱动模型三个步骤：

(1) 识别风险，并排定优先级；

(2) 选择并运用一组技术；

(3) 评估风险降低的程度。

特点：

以风险或特征为中心 核心要素：将风险放到极为显著的位置

合乎逻辑的理由 不同的开发者察觉到的风险不同，因此会挑选不同的技术，而风险驱动模型却具有一种有用的属性（）产出可供评估的论据。

> 将A、B、C识别为风险，其中B为首要风险。我们会花费时间运用X和Y技术，因为这两项技术有助于降低风险B。我们对由此产生的设计结果进行评审，并一致认为已充分降低了风险B，因此决定按照该设计进行编码。

3.2 你现在采用风险驱动了吗

我们应该能够准确清晰地回答“你的主要失败风险及对应的工程技术是什么”这类问题。

技术选择应该多样化 不同的风险采用不同的技术，一些项目会面临棘手的质量属性需求，因此需要做预先的计划式设计；而另一些项目只是对现有系统的微调，所以失败的风险也就微乎其微。

样本不符 我们应该采用不同的架构和不同的技术来处理面临的不同风险

3.3 风险

风险 = 觉察到的失败概率×觉察到的影响

描述风险 要将可能导致失败的的风险逐个描述为可测试的失败场景，如“当负载达到峰值时，客户体验到的用户界面延迟超过5秒”。

工程风险与项目管理风险

项目管理风险	软件工程风险
“首席开发者出了车祸”	“服务器无法扩展到 1000 名用户的规模”
“不理解客户需求”	“响应消息的解析可能存在缺陷”
“高级副总裁讨厌我们的经理”	“虽然系统现在能够工作，不过无论碰到哪里，系统都有可能散架”

技术类型必须与风险类型相匹配

典型风险

项 目 领 域	典 型 风 险
信息技术	复杂、难以理解的问题
	不确定正在解决真正的问题
	可能挑选不当的现成商业软件
	与现有的晦涩难懂的软件集成
	分散在人们中间的领域知识
	可修改性
系统	性能、可靠性、规模、安全
	并发
	模块的组合方式
Web	安全
	应用的可伸缩性
	开发者的生产力/表达能力

识别风险的方法

不完整的或容易引起误解的质量属性需求是最为常见的风险。可以**召开质量属性研讨会(quality attribute workshops)**，分发基于分类的调查问卷(taxonomy-based questionnaire)，或者采用其他相似方法，从而捕获风险并提供一份失败场景优先级列表。

决定风险的优先级

使用风险矩阵(risk matrices)，如MIL-STD-882D

3.4 技术

方法：

分析与类比模型

类比模型仅支持间接分析，而且通常需要借助领域知识或人类的推理能力。分析模型直接支持计算分析。

视图类型匹配

模块视图类型包括诸如源代码及类等有形的工件；运行时视图类型包括对象等运行时结构；部署视图类型包括服务器机房及硬件等部署元素。

密切相关的技术

某些技术只能用于处理特定的风险，例如，速率单调分析(rate monotonic analysis)主要用于应对可靠性风险，威胁建模(threat modeling)则用于解决安全风险，而队列理论(queuing theory)则可以降低性能风险。

3.5 何时停止（如何把握设计与架构的度）

付出的努力应与风险相称

风险小则付出的努力少。

不甚完整的架构设计

不需要对模型每个细节设计得非常仔细，架构如果能够解决当前的风险了，就可以停止建模。

主观评价

在完成架构后，需要人为主观评价，所涉及的架构是否能客服面临的失败风险。

3.6 计划式设计与演进式设计

演进式设计 系统的设计随着系统实现的增长而增长

敏捷实践中的重构、测试驱动设计及持续集成可以对付各种混乱问题。重构是克服演进式设计中大杂烩问题的主力。重构用解决当前全局问题的设计去替换那些解决旧有局部问题的设计。

计划式设计 项目构建开始前，就非常详细地制订出各种计划

仅对架构做完整的计划，多个团队进行并行开发需要共享架构时也很实用。

最小计划式设计 先做初步的计划式设计，介于演进式设计与计划式设计之间

一般20%的计划式设计和80%的演进式设计。

3.7 软件开发过程

将“风险”作为共享词汇 风险的概念位于工程领域与项目管理领域之间的公共地带。

内嵌的风险 在实践中，某些降低风险的步骤被有意内嵌到软件开发过程之中。

3.8 风险驱动模型的替代方案

风险驱动模型做了两件事：帮助我们决定何时可以停止架构设计，以及引导我们开展各种适当的架构活动。然而，它并不擅长预测在设计上到底该花多长的时间，但它可以帮助我们认识到何时设计业已足够。

代替方案

1. 不做设计：直觉
2. 文档包：建立一整套模型及图表，记录架构
3. 衡量标准：经验数据有助于决定在架构与设计上该投入多少时间。
4. 随机应变：依赖于开发者技能以及经验

五、建模建议

1. 专注于风险
2. 理解你的架构
3. 传播架构技能：让所有开发者都具备架构技能
4. 做出合理的架构决策
5. 避免预先大量设计
6. 避免自顶向下设计
7. 余下的挑战：估算风险，识别风险，排定风险的优先级，评估架构的候选方案，重用模型

5.1传播架构技能与合理决策

5.1.1传播架构技能

团队意义：在多开发者团队中，软件架构知识至关重要，所有开发者都应理解架构，这有助于提高沟通效率，做出合理决策。

避免垄断：架构知识不应被少数人垄断，而应使整个团队都具备架构意识，以实现系统的良好设计和发展。

5.1.2做出合理的架构决策

设计权衡：设计需要权衡，应根据质量属性的优先级进行选择，避免局部优化牺牲整体优先级。

解决分歧：不同开发者对设计方案可能产生分歧，通过透明化决策过程，将分歧转化为工程或需求决策，有助于做出合理选择。

5.2避免设计陷阱

5.2.1避免预先大量设计

BDUF 的问题：预先大量设计可能导致工作重点错误、纸上谈兵和时间浪费，虽然在某些大型或高要求项目中可能是必要的，但应警惕其风险，及时转移到原型或实现阶段。

设计变体：包括设计臻于完美和为建模而建模等变体，这些都可能影响项目的效率和质量。

5.2.2避免自顶向下设计

问题所在：自顶向下的设计可能导致低层设计与实际情况不符，难以重用现有组件，还可能限制开发者的洞察力，应避免这种做法。

5.3余下的挑战

5.3.1估算风险的困难

识别风险：识别风险困难，未预料的风险可能出现，检查清单可帮助分享和保存风险信息，但仍难以全面识别。

排定优先级：排定风险优先级也具有挑战性，预估过高或过低都可能影响项目决策，工程师对风险的看法可能不同，导致主观决策。

5.3.2评估架构候选方案的挑战

评估的难度：评估架构候选方案困难，模型可能无法包含所有细节，选择最佳方案需要权衡各种因素，缺乏明确的参考标准。

动态变化：系统需求和环境的变化可能使评估变得更加复杂，需要不断调整和优化架构设计。

5.3.3模型重用的问题

模型的特点：模型忽略细节，不同模型适用于不同问题，难以完全重用。
具体困难：例如，一个列车调度模型不能作为财务折旧模型重用，组件模型在不同环境中可能无法解决问题，因为它可能没有考虑到特定环境的细节。

5.3.4跨越工程和管理的问题

管理差异：组织管理关注系统功能和质量，而架构设计需要考虑工程和管理因素，两者可能存在冲突，需要进行权衡。
决策影响：例如，系统部署方式的选择可能涉及工程和管理的考虑，需要根据具体情况做出决策。

六、工程师使用模型

6.1模型的必要性

应对复杂问题：工程师在处理大型或复杂的软件系统时，需要借助抽象来构建模型，以更好地理解和解决问题。

简化思维过程：通过抽象，能够将复杂的系统简化为易于理解和处理的模型，帮助工程师在更高层次上把握系统的本质。

6.2模型的作用

提供洞察力：模型可以帮助工程师深入理解系统的内部运作机制，发现潜在的问题和优化点。

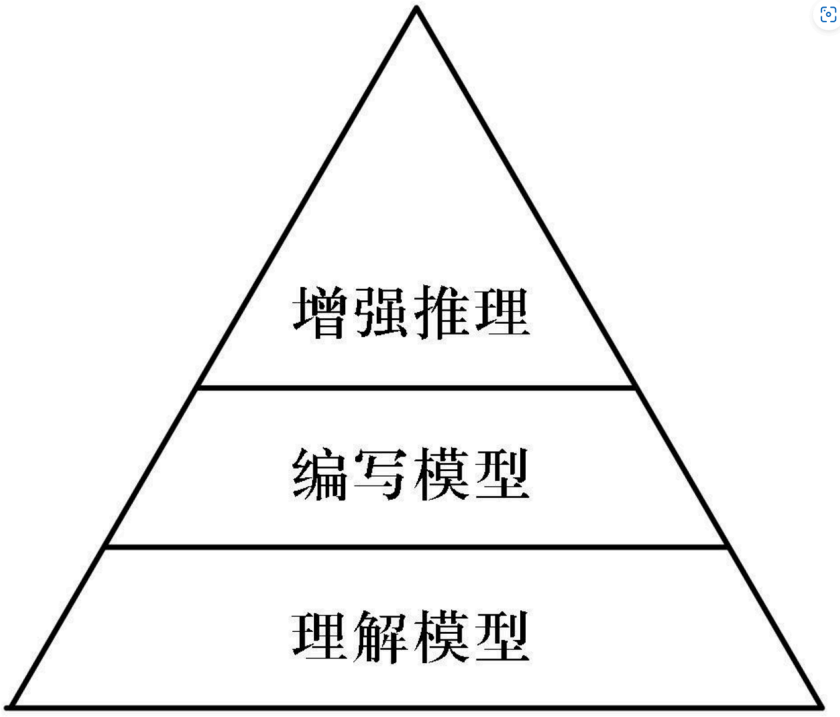
辅助解决问题：利用模型，工程师能够进行预测、分析和评估，从而找到解决问题的最佳方案。

6.3分析系统质量

质量属性评估：通过建立架构模型，可以对系统的质量属性进行分析，如性能、可伸缩性、安全性等。

细节与模型：在分析系统质量时，需要从模型中提取关键信息，同时忽略那些无关紧要的细节，以提高分析的准确性和效率。

6.4模型的特点



6.5建模的原则

明确问题导向：建模前应清晰地明确问题，根据问题的需求和特点来选择合适的模型抽象层次和细节。

选择合适模型：根据问题的性质和复杂性，选择最适合的模型类型，如数学模型、物理模型或架构模型等。

七、规范化结构与模型视图

7.1 规范化模型结构

结构概述：规范化模型结构提供了从抽象到具体的标准组织形式，包括领域模型、设计模型和代码模型三个主要模型，通过指定和细化关系确保模型的一致性和区分不同抽象层次。

模型关系：领域模型描述现实世界的不变事实，设计模型是对系统设计的建模，代码模型则对应系统的源代码。指定关系使不同模型中的相似元素相互对应，细化关系用于关联相同事物的高细节和低细节模型。

7.2 领域模型、设计模型和代码模型

领域模型：表达与系统相关的现实世界的不变事实，如类型、关系和行为，是对领域的抽象描述。

设计模型：主要是在设计者控制下的系统设计模型，包括边界模型和内部模型，是设计承诺的一部分。

代码模型：既是系统的源代码实现，又是对设计的完整承诺，包含了系统的具体实现细节。

7.3 指定与细化关系

指定关系：确保不同模型中相似元素的对应，领域模型中的事实在设计模型中应得到延续，同时设计模型不能违背领域的事实。

细化关系：用于关联边界模型和内部模型，展示相同内容的不同细节层次，细化关系保证了模型的完整性和一致性。

7.4 主模型的视图

视图定义：视图是模型细节的子集显示，包括领域模型、设计模型和代码模型的各种视图，如类型列表、系统上下文图等。

视图一致性：视图应与主模型保持一致，相互之间应保持连贯和准确，以确保对系统的全面理解。

主模型范例：主模型可以是已存在的系统或将要构建的系统，通过视图可以展示系统的不同方面和细节。

7.5 组织模型的其他方式

业务建模：

表7.1　总结不同作者提出的模型，以及如何与本书提到的业务模型、领域模型、设计模型(边界模型和内部模型)和代码模型对应

	业务模型	领域模型	设 计 模 型		代码模型
			边界模型	内部模型	
Bosch			系统环境	组件设计	代码
Cheesman & Daniels		业务概念	类型规格	组件架构	代码
D'Souza (MAp)	业务架构	领域	黑盒	白盒	代码
SEI			需求	架构	代码
Jackson		领域	领域+机器	机器	
RUP	业务建模	业务建模	需求	分析&设计	代码
Syntropy		要素	规格	实现	代码

7.6 UML 的用法

UML 标记：本书使用 UML 提供的标记来描述模型，虽然在某些方面与 UML 标准有所不同，但旨在更清晰地表达模型的结构和关系。

符号表达：通过合理使用 UML 符号，可以更有效地展示模型的元素、关系和行为，提高模型的可读性和可理解性。

7.7 学习总结

规范化模型结构有助于理解和组织软件系统的模型，通过指定和细化关系构建从抽象到具体的模型体系，视图则用于展示模型的细节。

八、邻域模型

8.1 领域与架构的关系

关注与问题：介绍了对领域建模的常见关注和反对意见，如认为已了解领域、领域简单无需建模等，并通过手机联系人列表和用户授权两个故事，说明领域理解不一致可能给系统带来的问题，如查找电话号码困难和系统不兼容等。

避免分析瘫痪：提出避免分析瘫痪的方法，包括明确建模目的、关注风险相关问题、决定模型的深度和广度等。

8.2 信息模型

信息模型是领域模型中最简单、最有价值的部分，包括类型列表、定义和关系，可通过文本或图形（如 UML 类图）表示。在领域建模中使用 UML 模型元素的简化子集，避免过多符号，以方便非开发人员理解。

8.3 导航和不变量

导航：在领域模型中，导航是通过关联来实现的。关联是模型中元素之间的关系，例如在 “个人” 类型和 “联系” 类型之间的关联。通过这种关联，开发人员可以从一个模型元素（如 “个人”）沿着关联移动到另一个相关的元素（如 “联系”），从而遍历整个模型。这种遍历能力有助于开发人员理解模型中不同元素之间的关系和交互方式。

不变量

定义：不变量是在领域模型中用于表示永远为真的断言。它是对模型中元素和关系的约束，确保模型的一致性和准确性。例如，在某个领域中，规定一个人不能有多个相同的联系关系，这就是一个不变量。

表达方式：不变量可以通过文字描述来表达，如 “一个人不能和同一个人建立多次联系关系”，也可以使用对象约束语言（OCL）来精确表达。OCL 是一种用于定义和验证模型约束的语言，它可以在模型中定义各种不变量和约束条件，以确保模型的正确性。

作用：强调不变量和导航在描述领域模型中的重要性，能帮助开发人员更好地理解和定义领域模型。

8.4 快照

图形表示：快照是一种以图形方式展示领域模型实例的手段。它通过可视化的形式呈现出领域模型中具体的实例，这些实例与信息模型中定义的类型是相对应的。例如，在招聘广告和业务网络领域中，如果信息模型定义了 “个人”“联系”“网络” 等类型，那么快照就会展示出这些类型的具体实例，如具体的个人、他们之间的联系以及联系所构成的网络。

8.5 功能场景

事件描述

功能场景主要用于描述导致领域模型状态发生变化的一系列事件。这些事件可以是用户操作、系统内部处理等，它们共同作用于领域模型，使其从一个状态转换到另一个状态。

表示方式

功能场景可以通过文字来详细描述事件的发生顺序和过程，也可以使用图形工具，如 UML 活动图或状态图来直观展示。UML 活动图可以展示事件的流程和各个步骤，状态图则可以展示领域模型在不同事件影响下的状态变化。

重要性

补充描述：功能场景能够补充信息模型的静态描述。信息模型主要关注领域中的静态概念和关系，而功能场景则聚焦于这些概念和关系在运行时的动态变化，展示了领域模型中元素的交互过程和行为。

沟通理解：有助于开发人员与主题专家进行沟通和理解系统功能。主题专家通常对业务流程和系统功能有更深入的了解，通过功能场景，开发人员可以将抽象的业务需求转化为具体的模型描述，与主题专家进行交流，确保系统功能符合业务要求。同时，功能场景也能够帮助开发人员更好地理解系统的复杂性和动态性，从而更好地进行系统设计和开发。

九、设计模型

9.1 设计模型

模型定义

整体概念：设计模型是软件架构中至关重要的组成部分，它是包含所有设计细节的主模型。这意味着设计模型涵盖了系统设计的各个方面，从功能需求到技术实现，从组件结构到系统架构的整体布局。

组织方式：通过视图、封装和嵌套等方式来组织和展示设计信息。视图允许开发人员从不同的角度和关注点来审视设计模型，选择展示特定的细节和信息；封装将系统的实现细节隐藏在内部，只暴露必要的接口，以提高系统的可维护性和可扩展性；嵌套则通过将系统分解为更小的子组件和模块，实现了层次结构的设计，使系统更易于理解和管理。

关联关系：与领域模型和代码模型密切相关。领域模型提供了系统所基于的业务和概念基础，设计模型则将这些领域概念转化为具体的设计决策和实现方案；代码模型是设计模型的具体实现，它体现了设计模型中所定义的结构和行为。

重要性

设计核心：是软件架构设计的核心关注点。在软件开发过程中，大部分的时间和精力都投入到了设计模型的创建和优化中。它决定了系统的整体架构、功能模块的划分、组件之间的通信方式以及系统的质量属性等重要方面。

决策依据：为开发人员提供了决策的依据。通过设计模型，开发人员可以清晰地了解系统的需求和约束，评估不同设计方案的优缺点，从而做出明智的设计决策。这些决策将直接影响系统的实现和质量。

沟通基础：是开发团队内部沟通的重要基础。设计模型的可视化和文档化特性使得开发人员能够更好地理解彼此的设计意图，减少沟通障碍，提高团队协作效率。同时，设计模型也是与其他利益相关者（如客户、项目经理等）进行沟通的重要工具，有助于确保系统的设计符合各方的期望。

9.2 边界模型

模型内容

系统外观：边界模型主要展现系统的外观，包括系统的行为、与外部交互的数据以及系统的质量属性。行为方面，它定义了系统能够执行的操作和响应事件的方式；数据交互方面，它描述了系统与外部系统或用户之间的数据传输格式和协议；质量属性方面，它考虑了系统的性能、可靠性、安全性等非功能需求。

接口承诺：是对系统接口的承诺，而不是对实现细节的承诺。这意味着边界模型只关注系统与外部世界的交互接口，确保这些接口的定义清晰、规范，并且能够满足系统的功能和质量要求。内部实现细节则被隐藏在边界模型的背后，只有在需要深入了解系统实现时才会被揭示。

作用

用户理解：为系统的用户提供了对系统的基本理解。用户通过边界模型可以了解系统的功能和操作方式，知道如何与系统进行交互，从而更好地使用系统。

系统集成：有助于系统的集成和与外部系统的交互。边界模型定义了系统与外部系统的接口规范，使得不同的系统可以通过这些接口进行集成和交互，实现信息共享和业务协同。

设计约束：对系统的设计和实现起到了约束作用。边界模型中的定义和承诺限制了系统内部的设计和实现，确保系统的设计符合外部需求和约束，提高了系统的稳定性和可维护性。

9.3 内部模型

模型特点

细节补充：是设计模型的另一种视图，它显示了边界模型中不予考虑的细节。与边界模型相比，内部模型提供了更深入的系统实现细节，包括组件的内部结构、算法的实现、数据的存储方式等。

细化关系：内部模型是对边界模型的细化，这意味着内部模型中的信息是对边界模型的进一步扩展和深化。边界模型中的承诺和定义在内部模型中必须得到支持和实现，同时内部模型中的细节也不能与边界模型中的定义相冲突。

关系

共同基础：边界模型和内部模型都基于相同的设计模型，它们共同构成了对系统设计的完整描述。边界模型提供了系统的外部视图，而内部模型则提供了系统的内部实现细节，两者相互补充，使开发人员能够全面了解系统的设计。

协同工作：在系统开发过程中，边界模型和内部模型协同工作，共同指导系统的实现。开发人员在设计系统时，首先根据需求创建边界模型，明确系统的外部接口和功能要求；然后，通过细化边界模型，创建内部模型，详细设计系统的内部实现细节。在实现过程中，开发人员始终以边界模型和内部模型为指导，确保系统的实现符合设计要求。

9.4 质量属性

属性介绍

属性定义：质量属性是描述系统外部特性的重要概念，它包括性能、可修改性、安全性、可用性等方面。性能属性关注系统的运行速度、响应时间、吞吐量等；可修改性属性涉及系统的可扩展性、可维护性和可重构性；安全性属性确保系统免受恶意攻击和数据泄露的威胁；可用性属性则关注系统的易用性和用户体验。

正交关系：质量属性与功能在大多数情况下是正交的，这意味着系统可以在不同的质量属性之间进行权衡和优化，而不会影响系统的基本功能。例如，一个系统可以选择在性能和可修改性之间进行权衡，以满足不同的业务需求。

重要性

系统品质：对系统的质量和可用性具有至关重要的影响。一个系统的质量属性直接关系到用户对系统的满意度和系统的长期运行效果。良好的质量属性可以提高系统的可靠性、稳定性和安全性，降低系统的维护成本，提高系统的竞争力。

架构设计：是架构设计的重要考虑因素。架构师在设计系统时，需要根据系统的需求和约束，合理地选择和配置质量属性，以确保系统能够满足用户的期望。同时，架构设计也需要考虑质量属性之间的权衡和协调，以实现系统的整体优化。

9.5 Yinzer 系统的设计之旅

用例和功能场景

用例图：用例图是一种简洁的图形化概览，它展示了系统的功能以及系统与活动者之间的交互关系。通过用例图，开发人员可以快速了解系统的主要功能模块和它们之间的协作方式，以及系统与外部用户和系统的交互接口。

功能场景：功能场景描述了系统在运行过程中一系列事件的发生顺序和系统状态的变化情况。它详细说明了系统如何响应外部事件，如何执行内部操作，以及如何实现系统功能。功能场景是对用例图的补充和细化，它帮助开发人员更深入地理解系统的行为和交互过程。

系统上下文

图的作用：系统上下文图展示了系统与外部系统之间的关系，包括系统与外部系统的交互方式、数据交换格式以及系统所处的运行环境。它提供了系统的宏观视图，帮助开发人员了解系统在整个业务领域中的位置和作用。

细节展示：通过系统上下文图，开发人员可以清晰地看到系统与外部系统之间的连接器和端口，以及它们之间的通信协议。这有助于开发人员理解系统的集成需求和技术实现方案，同时也为系统的安全性和可靠性设计提供了重要依据。

组件、端口和连接器

组件：组件是系统中执行主要计算和数据存储的基本单元。在 Yinzer 系统中，组件的定义和设计直接影响着系统的结构和功能。不同的组件负责不同的功能模块，它们之间通过端口和连接器进行通信和协作。

端口：端口是组件与外部系统进行交互的接口，它定义了组件提供的服务和响应的事件。端口的设计需要考虑到系统的功能需求和外部接口的标准，确保组件能够与其他系统进行有效的交互。

连接器：连接器是组件之间进行通信的桥梁，它定义了组件之间的数据传输方式和通信协议。连接器的选择和设计需要根据系统的性能要求、可靠性需求和可扩展性要求进行综合考虑，以确保系统的稳定性和灵活性。

设计决策、模块和部署

设计决策：设计决策是系统设计过程中的重要环节，它包括系统的架构选择、技术选型、设计原则等方面。设计决策的制定需要考虑到系统的需求、约束和质量属性要求，以确保系统的设计符合预期。

模块：模块是系统实现的基本组织单位，它将相关的代码和功能封装在一起，提供了良好的代码组织和模块化设计。在 Yinzer 系统中，模块的划分和设计需要考虑到系统的功能需求、可维护性和可扩展性要求，以确保系统的易于理解和修改。

部署：部署是将系统部署到实际运行环境中的过程，它包括硬件配置、软件安装、网络配置等方面。部署的设计需要考虑到系统的性能要求、可靠性需求和可维护性要求，以确保系统能够在实际运行环境中稳定运行。

组件装配

装配图展示：组件装配展示了组件、端口和连接器的具体配置和连接关系。通过组件装配图，开发人员可以清晰地看到系统的内部结构和组件之间的协作方式，了解系统是如何实现其功能的。

内部细节：组件装配图不仅展示了系统的外部结构，还揭示了系统的内部实现细节。它包括组件的内部结构、组件之间的通信方式、数据的传输流程等方面。这些内部细节对于开发人员理解系统的工作原理和进行系统维护非常重要。

两级功能场景

细化过程：两级功能场景是对功能场景的进一步细化，它将功能场景中的步骤进一步分解为更小的子步骤，并展示了内部组件之间的协作过程。通过两级功能场景，开发人员可以更深入地理解系统的行为和交互过程，发现系统中可能存在的问题和优化点。

协作关系：两级功能场景强调了内部组件之间的协作关系，它展示了不同组件在实现系统功能过程中的分工和协作。通过分析两级功能场景，开发人员可以更好地理解系统的架构设计和组件之间的交互方式，从而进行系统的优化和改进。

职责、导轨式约束和架构风格

职责分配：职责是指组件或模块在系统中所承担的任务和责任。在 Yinzer 系统的设计中，职责的分配需要考虑到系统的功能需求、组件的能力和约束以及系统的质量属性要求。合理的职责分配可以提高系统的可维护性和可扩展性。

导轨式约束：导轨式约束是一种设计原则，它通过对系统的设计和实现进行约束，确保系统能够满足质量属性要求。在 Yinzer 系统中，导轨式约束可以包括性能约束、可靠性约束、安全性约束等方面。这些约束可以帮助开发人员在设计和实现系统时避免出现错误和风险，确保系统的质量和可靠性。

架构风格：架构风格是一种设计模式，它定义了系统的整体架构和组件之间的交互方式。在 Yinzer 系统中，可以采用分层架构风格、管道 - 过滤器架构风格等不同的架构风格来设计系统。架构风格的选择需要考虑到系统的需求、约束和质量属性要求，以确保系统能够满足用户的期望。

9.6 视图类型

视图类型定义

定义概念：视图类型是一组或一类可以轻松相互对应的视图的集合。视图类型的目的是为了帮助开发人员更好地理解和管理系统的设计信息，通过将系统的设计信息分解为不同的视图类型，可以更清晰地展示系统的不同方面和细节。

对应关系：不同视图类型的视图之间具有一定的对应关系，这意味着它们关注的系统方面和细节是相关的，但又有所不同。例如，模块视图类型关注系统的模块结构和组织方式，运行时视图类型关注系统的运行时行为和组件之间的交互，部署视图类型关注系统的部署环境和硬件配置。

类型介绍

视 图 类 型	视图类型内容举例
模块视图类型	模块、层、依赖、职责(如 CRC)、数据库模式、接口、类、组件类型、连接器类型
运行时视图类型	对象实例、组件实例、连接器实例、行为模型(状态机、场景)、职责(基于实例)
部署视图类型	发布的软件、地理位置、计算节点
Spanning 视图类型	设计权衡(质量属性、业务、其他)、功能场景、质量属性场景

内容特点

详细内容：模块视图类型提供了系统的静态结构信息，包括代码的组织和模块之间的依赖关系；运行时视图类型展示了系统的动态行为和组件之间的交互过程，包括对象的创建、销毁和消息的传递；部署视图类型描述了系统在硬件上的部署情况，包括服务器的配置、网络的拓扑结构和存储设备的分配。

视角差异：不同视图类型从不同的视角展示了系统的设计信息，它们相互补充，共同构成了对系统设计的全面理解。开发人员可以根据自己的需求和关注点选择不同的视图类型来查看系统的设计信息，从而更好地理解和管理系统的设计。

类型差异

难以对应：不同视图类型中的视图在概念和表达方式上存在差异，因此难以直接对应。例如，模块视图类型中的模块和运行时视图类型中的组件在概念上是不同的，它们的表示方式和关注的细节也有所不同。

思考挑战：跨视图类型思考需要开发人员具备较强的系统思维能力和抽象思维能力，能够理解不同视图类型之间的关系和差异，并将它们整合起来形成对系统的整体理解。

思考方法

遵循建议：接受跨视图类型思考的困难是必要的，开发人员应该遵循 Dijkstra 的建议，建立使模块视图类型中的元素在运行时视图类型中更容易想象的风格。这意味着开发人员在设计系统时，应该考虑到系统的运行时行为和组件之间的交互关系，将模块设计与运行时结构相结合，以便更好地理解和管理系统的设计。

选择起点：从合适的视图类型开始思考也是很重要的。开发人员应该根据系统的特点和需求，选择最适合的视图类型作为思考的起点。例如，如果系统的重点是功能实现，那么可以从模块视图类型开始思考；如果系统的重点是运行时性能和可靠性，那么可以从运行时视图类型开始思考。

串联作用

整合信息：功能场景和设计权衡可以跨越视图类型，将不同视图类型中的信息串联起来。功能场景描述了系统的行为和交互过程，它可以通过引用不同视图类型中的元素来展示系统的功能实现；设计权衡则考虑了系统在不同质量属性之间的权衡和取舍，它可以通过比较不同视图类型中的设计方案来做出决策。

理解架构：视图类型串联有助于开发人员更好地理解系统的整体架构和设计。通过将不同视图类型中的信息整合起来，开发人员可以更全面地了解系统的功能、性能、可靠性等方面的要求，并通过设计权衡来做出合理的设计决策，以满足系统的整体需求。

完整性建议

全面考虑：解释了从整体上理解架构的重要性，开发人员应该从每一种视图类型中选取有代表性的视图，以避免视野狭窄。这意味着开发人员不能只关注某一个视图类型，而应该综合考虑系统的各个方面，从不同的角度来理解和分析系统的设计。

避免错误：只有通过全面考虑和综合分析，开发人员才能发现系统设计中可能存在的问题和不足，避免做出片面的决策，从而确保系统的设计质量和可靠性。

9.7 动态架构模型

架构特点

运行时变化：动态架构模型描述了架构在运行时的变化情况，它关注系统在运行过程中组件结构形态的改变和系统行为的动态演化。与静态架构模型相比，动态架构模型更能反映系统的实际运行情况和复杂性。

系统需求：有些系统在运行时需要根据不同的业务需求和环境条件进行动态调整和优化，例如，电子商务系统在高峰时段需要增加服务器资源，以满足用户的访问需求。动态架构模型可以帮助开发人员更好地理解和管理系统的运行时变化，提高系统的适应性和灵活性。

分析困难

工具支持有限：分析动态架构比较困难，目前的工具和分析方法对动态机制的支持有限。动态架构涉及到系统在运行时的复杂行为和交互关系，需要使用专门的工具和技术来进行分析和建模。

理解能力要求高：开发人员需要具备较强的系统思维能力和理解能力，才能够理解和分析动态架构模型。动态架构模型通常比较复杂，需要开发人员深入了解系统的业务需求、架构设计和运行时行为，才能够准确地分析和评估系统的动态特性。

9.8 架构描述语言

语言作用

描述架构：架构描述语言是用于描述架构模型的工具，它提供了一种标准化的方式来表达系统的架构设计。架构描述语言可以帮助开发人员更清晰地表达系统的架构设计意图，提高系统的可理解性和可维护性。

模型表达：不同的架构描述语言对静态架构和动态架构的支持程度不同。一些架构描述语言侧重于静态架构的描述，如 UML；而另一些架构描述语言则更适合描述动态架构，如 Petri 网。开发人员可以根据系统的特点和需求选择合适的架构描述语言来描述系统的架构模型。

选择建议

UML 推荐：推荐使用 UML 来描述架构模型。UML 是一种广泛使用的建模语言，它提供了丰富的图形符号和建模机制，可以方便地描述系统的架构模型。UML 具有强大的表达能力和广泛的应用基础，它可以用于领域模型、设计模型和代码模型的描述，有助于实现模型的一致性和可追溯性。

优势分析：UML 得到了广泛的支持，包括工具支持、开发人员培训和文档标准等方面。使用 UML 可以提高开发人员之间的沟通效率，减少误解和错误，同时也有利于系统的集成和维护。此外，UML 的标准性和开放性也使得它能够与其他工具和技术进行集成，为系统的开发提供了更好的支持。

十、代码模型

10.1 模型 - 代码差异

差异表现

词汇与抽象：架构模型和源代码使用不同词汇，架构模型更抽象，如组件类型在源代码中可能没有直接对应。

设计承诺：架构模型只作部分设计承诺，源代码则要实现完整的功能和质量属性。

内涵 - 外延：架构模型包含内涵式元素，如设计决策和不变量，这些在源代码中难以直接体现。

位　　置	元　　素
架构模型	模块、组件、连接器、端口、组件装配、风格、不变量、职责分配、设计决策、基本原理、协议、质量属性及模型(例如，安全策略、并发模型)
源代码	包、类、方法、变量、函数、过程、语句

管理挑战

- **差异根源**：由于两者的目的和表达方式不同，导致了差异的存在。
- **管理策略**：可以通过机制（如代码自动生成）或人为控制来管理差异，如遵循架构明显的编码风格。

10.2 一致性管理

管理策略概述

多种策略并存：包括忽略分歧、临时建模、只维护概要模型、在特定阶段同步以及定期同步等。

选择依据：取决于工具使用、模型详细级别和项目对分歧的容忍度。

重要性强调：一致性管理对于确保系统的设计和实现一致至关重要，不一致可能导致系统的混乱和错误。

策　　略	描　　述
忽略分歧	使用了过期模型，但是记得曾经做过什么改变
临时建模	模型放在脑子里，需要的时候重建它
概要模型	架构中最基础的部分变化比较少，所以只对这部分进行建模
在里程碑处同步	在迭代结束、发布或其他的里程碑处同步代码和模型
在危机时同步	当出现问题，或者设计评审的时候，同步代码和模型。没有比这个更常见的了
定期同步	成本高，很少见

10.3 架构明显的编码风格

编码原则阐述

代码质量提升：不仅仅是让代码能够运行，更要通过在代码中嵌入架构线索，提高代码的可理解性和可维护性。

设计意图保留：遵循模型嵌入代码原理，将架构设计意图保留在代码中，避免设计意图的丢失。

表达设计意图的方式

- **有意提示**：如使用具有明确含义的方法名，让代码读者能够理解代码的设计意图。
- **按合约设计**：使用前置条件和后置条件等方式，规范代码的行为，确保代码的正确性和可预测性。

10.4 在代码中表达设计意图的模式

模式总体介绍

模式目的：通过在代码中引入特定的模式，为代码读者和维护者提供关于架构的线索，帮助他们理解代码的结构和功能。

模式类型

组件相关模式：包括将组件类型具体化为类，添加端口和连接器实例变量等。

连接器相关模式：将连接器类型具体化为类，明确连接器的职责和工作方式。

其他模式：如端口类型的表达、协议的实现、属性的表示等模式。

10.5 电子邮件处理系统预演

系统架构概述

处理流程：处理电子邮件的系统包括清理原文、标记信息、进行特性分析和分类等阶段，采用管道 - 过滤器架构风格。

架构优势：这种架构风格有助于实现系统的可扩展性和可维护性。

代码体现细节

包结构：包结构反映了系统的模块划分，有助于组织代码和提高代码的可读性。

组件类型：通过 Filter 类等组件类，体现了管道 - 过滤器架构风格，明确了组件的职责和工作方式。

连接器类型：如 Pipe 类，作为连接器的具体实现，负责组件之间的数据传输和通信。

十一、封装和分割

11.1 多层级故事

系统理解与层级结构

系统复杂性与可理解性：大型系统由众多相互作用的部件组成，若设计不当，会使开发人员难以理解。层级嵌套的结构方式能帮助构建多层级故事，使系统更易于理解。

构建多层级故事的指导方针

分级嵌套元素：通过将系统元素分级嵌套，如基本模块、组件和环境元素，创建不同的抽象级别，有助于开发人员逐步理解系统的复杂性。

限制元素数量：每个层级的元素数量应合理，一般在 5 到 50 个之间，过多或过少都不利于理解。

明确元素目的：每个元素都应有明确的目的，这样开发人员才能更好地理解它们在系统中的作用。

确保封装性：元素应被有效封装，隐藏不必要的内部细节，只展示对外的接口，以降低系统的复杂性。

多层级故事的好处与困难

好处

应对规模与复杂性：多层级故事使开发人员能够应对系统的大规模和复杂性，通过将系统分解为多个层级，每个层级关注特定的功能和问题，使开发人员能够更有效地管理和理解系统。

提高可维护性：封装性和层级结构降低了系统的复杂性，使开发人员更容易维护系统。他们可以在不同层级上进行修改和扩展，而不会影响到其他层级的功能。

困难

维护成本：维护多层级故事需要开发人员投入更多的精力和时间，确保各个层级之间的一致性和交互性。

有效封装的挑战：实现有效的封装并不容易，需要开发人员具备良好的设计和编程技能，以确保接口的简洁性和稳定性。

11.2 层级和分割

系统结构与分割策略

分割的概念与作用：分割是将系统分为分散的块，如航天器的有效负载和发射工具，软件的客户端和服务器等。分割有助于理解和管理系统的结构，使系统更容易设计和实现。

无层级系统的例外情况：并非所有系统都能进行层级分解，例如，硬盘在服务器和航天器中的使用情况就不同，在服务器中可以将硬盘作为一个层级封装的部件，但在航天器中，硬盘的影响无法被封装。

自顶向下设计的问题：自顶向下的设计方法并不适用于所有情况，虽然开始时从高层设计入手，但在深入细节时，可能会发现一些细节迫使你改变之前的设计决定。

主分解的影响

主分解的定义：主分解是在系统分解中选择一个主要的关切，如根据主题、功能或其他因素对系统进行分解。主分解会影响系统的结构和设计，不同的主分解可能会导致不同的系统设计方案。

主分解的专横：主分解可能会导致一些问题，如阻碍其他关切的解决，使系统设计变得困难。在进行系统分解时，需要综合考虑各种因素，避免主分解的专横。

11.3 分解策略

分 解 策 略	元　　素
功能	大块的相关功能
典型类型	领域中的重要类型
架构风格	风格中已命名的元素
属性驱动设计	匹配表策略
端口	对应于每一个端口的元素
正交抽象	来自其他领域的元素，例如，数学或图论
智力拼图	现有的元素，加上一些新的元素作为“胶水”

功能分解

分解原则：基于功能来分解系统是一种常见的策略，将相关的功能聚合在一起，形成不同的组件或模块。

Yinzer 系统的例子：在 Yinzer 系统中，可以根据功能将系统分解为网站、数据库、电子邮件、业务网络、招聘广告等组件，或者根据成员和非成员的操作进行横向切分。

典型类型分解

典型类型的定义：识别领域中的典型类型，如联系、广告、用户等，有助于进行职责分配和系统设计。

分解优势：典型类型通常是领域中重要的、独立存在的类型，它们的分解可以使系统的设计更加清晰和合理。

架构风格分解

架构风格的作用：根据架构风格来分解系统，将系统的组件和连接器按照特定的风格进行组织，有助于实现系统的质量属性和设计目标。

例子说明：例如，使用管道 - 过滤器风格构建邮件回复系统，将系统分解为过滤器组件和管道连接器，每个组件和连接器都遵循特定的规则和约束。

质量属性和属性驱动设计

质量属性的重要性：对于大型系统，质量属性的实现至关重要，如可伸缩性、可修改性、安全性等。

属性驱动设计过程：SEI 提出的属性驱动设计过程通过定义质量属性场景，选择适合实现这些质量的模式或设计，来指导系统的设计和实现。

端口分解

端口的作用：每个组件都有端口，端口是组件之间通信的接口，通过端口可以实现组件之间的交互和协作。

分解策略应用：可以为每个端口创建对应的组件，或者将端口作为组件之间的中介，来实现系统的分解和设计。

正交抽象分解

分解概念：将组件的职责转换到一个不同的领域，如使用算法论或其他领域的知识，来进行系统的分解和设计。

潜在优势：正交抽象分解可以利用领域特定的知识和经验，提高系统的性能和可理解性。

智力拼图分解

分解方式：根据已有的子组件、COTS 组件或现有代码，通过组装和整合这些组件来构建系统。

应用场景：当有一些现成的组件或代码可以重用时，智力拼图分解是一种有效的分解策略。

11.4 有效封装

封装的概念与重要性

封装的定义：封装是将系统的实现细节隐藏在内部，只暴露对外的接口，使其他开发人员只能通过接口与系统进行交互。

降低认知压力：有效的封装可以降低其他开发人员的认知压力，使他们能够更容易地理解和使用系统。

封装失败的例子

工作时间表系统的问题：以工作时间表系统为例，说明封装不好的系统会导致接口泄漏抽象，使系统难以理解和维护。

问题的根源：封装失败的原因可能是接口设计不合理，没有隐藏实现细节，或者是对系统的变化和扩展考虑不足。

帕纳斯模块的理念

模块设计原则：帕纳斯模块强调在设计模块时，要隐藏可能变化的细节，确保模块的接口稳定，使模块的实现可以在不影响其他模块的情况下进行修改。

实践意义：帕纳斯模块的理念对于提高系统的可维护性和可扩展性具有重要意义。

判断和风险

封装判断的困难：判断一个封装是否有效是困难的，需要考虑很多因素，如系统的需求、变化的可能性、开发人员的技能和经验等。

风险评估：在进行封装时，需要评估可能带来的风险，如接口变化对其他模块的影响、封装的成本和收益等。

11.5 创建封装接口

抽象数据类型 - 栈的示例

栈的定义与操作：栈是一种简单的数据类型，只允许访问栈的顶部，支持压入、弹出和查看栈顶元素等操作。

抽象数据类型的特点：抽象数据类型通过定义接口和操作，隐藏了实现细节，使开发人员可以在不了解实现的情况下使用数据类型。

把模块和组件当做 ADT

模块和组件的接口设计：类似于抽象数据类型的接口设计，模块和组件的接口应该定义清晰，只暴露必要的操作和属性，隐藏内部实现细节。

设计过程

选择元素：选择模块或组件上的端口，并确定端口上的操作和前后置条件。

绘制快照：通过绘制快照，展示操作对实例状态的影响，帮助理解接口的行为。

构建类型模型：泛化快照对，形成类型模型，确保接口的设计与实现一致。

十二、模型元素

12.1 和部署相关的元素

部署元素概述

硬件与环境：软件运行在硬件上，硬件必须架设在特定的地方，如服务器机房、数据中心等。这些硬件和地理位置被称为环境元素，包括服务器、网络设备等。

部署图与元素表达：部署图显示了模块和组件实例的部署情况，通过环境元素和通信通道来描述软件的运行环境。可以部署的元素包括用户界面的可执行代码、数据库的可执行代码等。

元素属性与约束

属性作用：环境元素、模块及组件上的属性可以用来标示兼容性、性能要求等，为系统的设计和部署提供重要的参考信息。

约束表达：属性可以用来表达对元素的约束，如硬件的内存要求、软件的兼容性要求等，这些约束有助于确保系统在不同的环境中能够正常运行。

12.2 组件

组件定义与类型实例

定义与特点：组件是软件架构中粗粒度的抽象，是系统中执行的主要计算元素和数据存储。组件有类型和实例之分，类似于类和对象的关系，组件类型定义了组件的规范和接口，组件实例则是具体的运行实体。

与模块的比较：组件和模块有一些相似之处，如都包含实现制品，但组件更侧重于运行时的概念，而模块更侧重于实现的组织。组件通常会在运行时进行实例化，并且通过端口和连接器进行通信，而模块在运行时的存在感相对较弱。

组件相关的其他主题

子组件与实现：系统可以由多个组件组成，组件内部可以包含子组件，通过嵌套的方式来组织系统的结构。子组件的实现可以是其他组件、类或函数等。

不确定性和模糊性：在软件架构中，组件的定义和使用存在一些不确定性和模糊性，例如，组件类型和实例的区分、组件的范围和边界等问题，需要开发人员在设计和实现过程中进行谨慎的处理。

基于组件开发：基于组件的开发是一种软件开发模式，通过使用已有的组件来构建系统，可以提高开发效率和软件的质量。但目前基于组件的市场还不够成熟，需要进一步的发展和完善。

12.3 组件装配

组件装配的概念

定义与用途：组件装配是显示组件、端口及连接器实例或类型组成的装配图，用于展示系统的运行时结构和组件之间的交互关系。它是软件架构设计中非常重要的工具，有助于开发人员理解和设计系统的架构。

系统上下文图：系统上下文图是一种特殊的组件装配，它关注正在设计中的系统与外部系统的交互关系，显示了系统的边界和接口。

组件装配的细化

细化过程：组件装配可以进行细化，通过展示组件的内部结构和细节，来深入了解系统的实现。细化过程需要遵循一定的规则，如保持端口的一致性、不改变外部可见的行为等。

细化语义：细化语义决定了在细化过程中可以引入哪些新的细节和变化，开放语义允许引入任意新的项，而封闭语义则限制了可以引入的项目类型，通常会列出不能改变的项目类型。

组件装配的表现力与理解

表现力要求：为了更好地展示系统的架构和组件之间的交互关系，组件装配图需要具有一定的表现力，能够清晰地展示组件、端口、连接器等元素的特征和关系。

理解设计：组件装配图只是理解系统设计的一个起点，还需要结合其他的设计文档和模型，如功能场景、质量属性场景等，来全面理解系统的设计意图和架构。

12.4 连接器

连接器的定义与类型

定义与作用：连接器是两个或更多组件之间运行时的交互通道，它定义了组件之间的通信方式和协议。连接器的类型包括过程调用、事件、管道、共享内存等，不同类型的连接器适用于不同的场景。

重要性强调：连接器在软件架构中起着至关重要的作用，它决定了组件之间的通信方式和交互关系，直接影响了系统的可扩展性、可维护性和性能等方面。

连接器的相关特性

可替换性：连接器和组件一样，都应该具有可替换性，这样可以在不影响系统整体架构的情况下，更换连接器的实现，以满足不同的需求。

选择合适的连接器：在选择连接器时，需要考虑系统的需求、性能要求、可靠性要求等因素，选择最适合的连接器类型。

属性与角色

属性：连接器可以具有各种属性，如性能、安全性、稳定性等，这些属性可以用来描述连接器的特点和适用场景。

角色：连接器的角色是指在与端口连接时所扮演的角色，不同的连接器角色可能会有不同的行为和功能。

连接器的细化与建模

细化过程：连接器可以进行细化，展示其内部的实现细节，如数据转换、协议处理等。细化过程需要遵循一定的规则，确保细化后的模型与原始模型的一致性。

建模灵活性：连接器的建模具有一定的灵活性，需要根据具体的情况选择合适的建模方式，以满足系统的设计需求。

12.5 设计决策

设计决策的重要性

指导设计方向：设计决策是开发人员在系统设计过程中做出的重要决策，它指导着系统的架构设计和实现，决定了系统的功能和质量属性。

决策记录与说明：将设计决策记录下来，并给出相应的说明，有助于其他开发人员理解系统的设计意图和实现原理，同时也有利于系统的维护和演化。

设计决策的表达

决策内容：设计决策包括系统的架构选择、技术选型、设计原则等方面，这些决策都对系统的设计和实现有着重要的影响。

决策说明：设计决策的说明应该包括决策的背景、目标、依据和影响等方面，使其他开发人员能够理解决策的合理性和必要性。

12.6 功能场景

功能场景的定义与作用

定义与特点：功能场景描述了系统的行为，它通过一系列的步骤和事件来展示系统在运行时的功能和交互过程。功能场景与用例类似，但更加具体和详细。

与其他模型的关系：功能场景与其他模型，如领域模型、设计模型等密切相关，它是对这些模型的具体实现和验证。

功能场景的结构与表达

结构要素：功能场景包括目标模型、场景名称、初始状态、活动者及步骤等要素，这些要素共同构成了一个完整的功能场景。

表达形式：功能场景可以用文字描述，也可以用图形化工具，如 UML 时序图来表达。图形化工具可以更加直观地展示系统的行为和交互过程。

功能场景的泛化与两级场景

泛化概念：功能场景可以进行泛化，从具体的场景中抽象出通用的模型，以满足不同的需求。

两级场景：功能场景可以分为两级，一级场景描述系统的整体行为，二级场景则深入到组件内部，展示组件之间的交互过程。

12.7 不变量（约束）

不变量的定义与分类

定义与作用：不变量是对系统的约束，它要求系统在运行时必须满足一定的条件。不变量可以分为静态不变量和动态不变量，静态不变量处理结构，动态不变量处理行为。

约束表达：不变量可以用文字描述，也可以用对象约束语言（OCL）来表达，OCL 是一种用于描述和验证模型约束的语言。

不变量在架构设计中的意义

保证系统质量：不变量可以保证系统的质量，如正确性、可靠性和安全性等。通过对系统的约束，不变量可以防止系统出现错误的行为和状态。

指导设计决策：不变量可以为设计决策提供依据，帮助开发人员做出合理的设计选择。例如，在设计数据库时，可以根据不变量来确定数据库的结构和约束。

12.8 模块

模块的定义与特点

定义与组成：模块是实现制品的集合，如源代码、配置文件、数据库模式定义等。模块可以将相关的代码和资源组织在一起，提供统一的接口和功能。

模块的作用：模块的主要作用是实现代码的模块化和封装，提高代码的可维护性和可扩展性。通过将系统分解为模块，可以更好地管理系统的复杂性。

模块的相关属性与关系

属性表达：模块可以具有各种属性，如编程语言、版本、依赖关系等，这些属性可以用来描述模块的特征和状态。

依赖关系：模块之间可以存在依赖关系，如一个模块依赖于另一个模块的功能或资源。依赖关系的管理是系统设计和开发中的重要环节。

层的概念：分层系统会对模块进行组织，低层为高层提供支持，形成一个层次结构。分层可以提高系统的可维护性和可扩展性，但也需要注意避免层之间的过度耦合。

12.9 端口

端口的定义与作用

定义与功能：端口是组件与外部进行通信的接口，所有进出组件的通信都是通过端口来完成的。端口定义了组件提供的服务和响应的事件，是组件与外部系统进行交互的桥梁。

提供和依赖的端口：端口可以分为提供的端口和依赖的端口，提供的端口用于向外提供服务，依赖的端口用于接收外部的服务请求。

端口的相关特性

多端口类型：为了满足不同的需求，组件可以具有多个端口类型，每个端口类型提供不同的服务和功能。

多个端口实例：一个端口可以有多个实例，不同的实例可以用于不同的场景和目的。

端口与接口：端口和接口有一些相似之处，但也存在一些区别。端口更侧重于运行时的概念，而接口更侧重于设计时的概念。

端口的状态机与协议

状态机表示：端口可以具有状态机，状态机描述了端口在不同状态下的行为和转换关系。状态机可以帮助开发人员更好地理解端口的工作原理和交互过程。

协议定义：端口通常会遵循一定的协议，协议定义了端口之间的通信规则和格式。协议的设计和实现是确保系统正常运行的关键。

12.10 质量属性

质量属性的定义与分类

定义与特点：质量属性是一种超功能需求，它描述了系统在非功能方面的特性，如性能、安全性、可修改性等。质量属性与功能需求是正交的，它们共同决定了系统的质量和价值。

分类方法：质量属性可以按照不同的标准进行分类，如按照视图类型、按照系统特性等。常见的分类方法包括功能质量属性、性能质量属性、安全质量属性等。

质量属性的重要性

对系统的影响：质量属性对系统的质量和性能有着重要的影响，它们直接关系到系统的可用性、可靠性和可维护性等方面。

设计考虑因素：在系统设计过程中，需要充分考虑质量属性的要求，选择合适的架构和设计方案，以确保系统能够满足质量属性的要求。

12.11 质量属性场景

质量属性场景的定义与结构

定义与作用：质量属性场景是对质量属性需求的具体描述，它通过描述系统在特定场景下的行为和响应，来评估系统的质量属性。质量属性场景是 ATAM（架构权衡分析方法）和属性驱动设计过程的基本组成部分。

结构要素：质量属性场景包括来源、触发、环境、制品、响应及响应测量等要素，这些要素共同构成了一个完整的质量属性场景。

质量属性场景的可验证性

验证的重要性：为了确保质量属性场景的有效性，需要对其进行验证。可验证的质量属性场景能够帮助开发人员发现系统在质量属性方面的问题和不足，从而及时进行改进和优化。

验证方法：可验证的质量属性场景应该具有明确的可测量的指标和验证方法，如响应时间、吞吐量、错误率等。开发人员可以通过模拟场景、测试用例等方式来对质量属性场景进行验证。

架构驱动元素

定义与意义：架构驱动元素是对利益相关者至关重要的质量属性场景，它代表了系统设计中需要重点关注和解决的问题。架构驱动元素是从现有的 QA 列表和功能场景中提取出来的，它能够帮助开发人员确定系统的架构方向和设计重点。

选择与应用：在选择架构驱动元素时，需要考虑利益相关者的需求、系统的质量属性要求和实现难度等因素。架构驱动元素可以用于指导系统的架构设计和决策，确保系统能够满足质量属性的要求。

12.12 职责

职责的分配与定义

职责分配原则：在系统设计过程中，需要为系统元素分配职责，确保每个元素都有明确的责任和任务。职责分配应该遵循单一职责原则，即每个元素应该只负责一项特定的任务。

职责的多样性：系统元素的职责可以包括功能职责和质量属性职责，功能职责是指元素在实现系统功能方面的责任，质量属性职责是指元素在保证系统质量属性方面的责任。

意图链与职责关系

意图链的概念：意图链是指从系统的架构驱动元素到具体的系统元素之间的关系链，它描述了系统元素的设计意图和实现过程。意图链能够帮助开发人员更好地理解系统的设计意图和架构原理。

职责的传递与影响：系统元素的职责是相互关联的，一个元素的职责变化可能会影响到其他元素的职责。因此，在进行职责分配和调整时，需要考虑到意图链的影响，确保系统的整体设计和实现不受影响。

12.13 权衡

权衡的概念与意义

权衡的定义：在软件架构设计中，权衡是指在不同的质量属性和设计方案之间进行选择和取舍，以达到最优的系统设计效果。

权衡的重要性：由于系统的资源和约束是有限的，因此在设计过程中需要进行权衡，以确保系统能够在满足各种需求的前提下，实现最优的性能和质量。

权衡的示例与分析

质量属性权衡：在不同的质量属性之间，如性能和可修改性、安全性和可用性等，往往需要进行权衡。开发人员需要根据系统的需求和特点，选择合适的权衡方案。

设计决策权衡：在设计过程中，不同的设计决策也可能会导致不同的权衡结果。例如，在选择架构风格时，需要考虑到架构风格对系统性能、可维护性等方面的影响。

十三、模型关系

13.1 投影（视图）关系

投影的定义与作用

定义：投影是某样东西从特殊角度看上去的样子，在软件架构中，视图是模型细节的已定义子集，显示内容可能经过转换。

作用：帮助应对复杂性和规模问题，减少需要理解的内容，突出模型的特定关切，如速度、气流或可操纵性。

视图间的一致性

挑战：维护多个视图的一致性是难题，可能出现视图冲突，如房屋楼面图中楼梯位置不一致。

解决方式

主模型方法：假定有完整的主模型，视图是主模型的投影，视图不一致说明设计有缺陷。

避免问题：使用视图要有好的理由，避免为还未确定的方案创建视图集。

视图的分类与应用

分类：视图可以根据不同的角度和关注点进行分类，如需求视图、主模型视图、实体投影视图等。

应用：在软件架构中，常见的视图类型包括模块视图、运行时视图和部署视图，不同视图类型关注系统的不同方面。

13.2 分割关系

分割的概念

定义：分割是把一个整体分成几个没有相交的部分，如将草坪分为前院和后院。

与投影的区别：分割要求部分合起来构成整体，且没有交叉，而投影可能会导致信息的丢失或改变。

分割的应用场景

实际例子：在软件系统中，分割可以用于将系统分解为模块、组件或功能模块，以便更好地管理和理解系统的结构。

优势：通过分割，可以降低系统的复杂性，提高系统的可维护性和可扩展性。

13.3 组合关系

组合的定义

定义：组合是将更小的模型创建为更大的模型，与分割相反，组合中的部分不一定是整体的组成部分。

特点：组合可以实现模型的复用和共享，提高系统的设计效率。

组合的应用

实际案例：在软件架构中，组合可以用于将不同功能的模块组合在一起，形成一个更大的系统。

注意事项：在进行组合时，需要注意模型之间的接口和交互，确保组合后的系统能够正常工作。

13.4 分类关系

分类系统的概念

定义：分类系统是用于选取东西并决定它们属于什么类别的系统，具有明确的分类规则和特性。

特点：分类系统应具有明确性、排他性和完整性，确保每个东西都能被正确分类。

分类关系的应用

软件架构中的应用：在软件架构中，分类关系可以用于对系统的元素进行分类，如组件类型、端口类型等，以便更好地管理和理解系统的结构。

注意事项：在进行分类时，需要根据系统的需求和特点，选择合适的分类标准，避免分类过于复杂或简单。

13.5 泛化关系

泛化的定义

定义：泛化是一种类型关系，描述了更通用的类型与更具体的类型之间的关系，如房屋是时尚房屋的泛化。

特点：泛化关系遵循利斯科夫替代原理，即子类型可以用超类型替换，但需要保证系统的正确性和稳定性。

泛化关系的应用

软件架构中的应用：在软件架构中，泛化关系可以用于构建系统的架构模型，如通过泛化关系可以将系统的组件类型组织成一个层次结构，以便更好地理解和管理系统的结构。

注意事项：在使用泛化关系时，需要注意避免过度泛化，导致系统的复杂性增加。

13.6 指定关系

指定关系的定义

定义：指定关系是在两个域之间建立桥梁，标识出现实世界中的东西与模型中对应元素的关系。

作用：指定关系有助于确保模型与现实世界的一致性，避免混淆现实和模型中的概念。

指定关系的应用

实际应用：在软件架构中，指定关系可以用于将系统的需求与模型中的元素进行关联，确保系统的设计符合需求。

注意事项：在使用指定关系时，需要确保指定的关系是准确和明确的，避免产生歧义。

13.7 细化关系

细化关系的定义

定义：细化是同一样东西的高细节和低细节表现形式之间的关系，高细节模型包含了低细节模型的所有信息。

特点：细化关系有助于降低系统的复杂性，使系统更易于理解和管理。

细化关系的应用

模型构建中的应用：在软件架构模型的构建过程中，细化关系可以用于将系统的模型从抽象的高层模型逐步细化到具体的低层模型，以便更好地理解和设计系统。

注意事项：在进行细化时，需要注意保持高细节和低细节模型之间的一致性，避免出现信息丢失或不一致的情况。

13.8 绑定关系

绑定关系的定义

定义：绑定关系是将源模型中的概念与目标模型中的元素进行关联，通过绑定关系可以将模式或风格应用于目标模型。

作用：绑定关系有助于确保系统的架构符合特定的模式或风格要求，提高系统的可维护性和可扩展性。

绑定关系的应用

实际应用：在软件架构中，绑定关系可以用于将系统的架构与特定的架构风格进行绑定，如将分层风格应用于系统的架构设计中。

注意事项：在使用绑定关系时，需要确保绑定的关系是正确和有效的，避免出现错误的绑定。

13.9 依赖关系

依赖关系的定义

定义：依赖关系是指一个模型的改变会导致另一个模型发生变化，如建造房屋的估算价格与原材料价格之间的关系。

特点：依赖关系是软件架构中常见的关系之一，它反映了系统中不同部分之间的相互影响和关联。

依赖关系的应用

软件架构中的应用：在软件架构中，依赖关系可以用于描述系统中不同组件、模块之间的关系，以及系统与外部环境之间的关系。

注意事项：在处理依赖关系时，需要注意避免依赖关系的过度复杂，导致系统的可维护性降低。

十四、架构关系

14.1 架构风格的概念

定义阐述：架构风格是一种在架构层面的模式，它界定了一组可运用的元素类型，涵盖模块、组件、连接器与端口等，并且明确了这些元素间的约束以及关系。例如，规定哪些类型的组件能够相连，以及通过何种连接器相连等。

内涵深度：它不仅仅是简单的结构描述，更是一种对系统架构设计的高层次抽象与规范，影响着整个系统从设计到实现再到维护的全生命周期。

14.2 架构风格的重要性

设计复用价值：为软件架构设计中的常见问题提供了可复用的模板。当面临相似的设计情境时，开发人员可以直接借鉴已有的架构风格，避免重复劳动，加速开发进程，同时减少因重新设计可能引入的错误风险。

沟通理解助力：使得不同开发人员、团队以及利益相关者之间对于系统架构有了统一的理解框架。通过遵循特定的架构风格，各方能够更便捷地沟通系统的结构、行为预期以及功能实现方式，降低沟通成本与误解可能性。

质量保障基石：有助于确保系统具备特定的质量属性。例如，某些架构风格天然地支持高可扩展性，而另一些则侧重于系统的安全性或者性能优化，从而为满足系统的非功能需求提供了有力支持。

14.3 常见架构风格示例与分析

分层架构风格

结构特点：将系统划分为多个层次，各层次职责分明。比如典型的网络协议分层模型，从物理层到应用层，每一层都在为上一层提供服务的同时屏蔽下层的实现细节。

优势与应用场景：提高了系统的可维护性，因为各层的修改相对独立，不会对其他层造成过大影响。适用于功能相对复杂、需要逐步抽象与细化的系统，如企业级应用开发中常见的表示层、业务逻辑层和数据访问层的划分。

管道 - 过滤器架构风格

工作原理：数据如同在管道中流动，依次经过各个过滤器组件进行处理。每个过滤器专注于一项特定的数据处理任务，例如在图像识别系统中，可能有图像采集、图像预处理、特征提取、分类识别等一系列过滤器。

适用范围与效果：在数据处理流程相对固定且顺序性强的场景中表现出色，能够方便地对系统进行扩展与修改，只需添加、删除或替换相应的过滤器即可。可有效提高系统的灵活性与可修改性，尤其在数据处理密集型的应用中应用广泛。

客户端 - 服务器架构风格

角色与交互方式：明确区分客户端与服务器的角色。客户端主要负责与用户的交互操作，收集用户输入并将请求发送给服务器；服务器则专注于处理请求，访问和管理数据资源，并将处理结果返回给客户端。

分布式系统优势：在分布式环境中能够实现资源的集中管理与共享，提高资源利用率。广泛应用于各类网络应用，如 Web 服务、数据库应用等，能够有效支持多用户并发访问与数据的集中存储与处理。

事件驱动架构风格

事件驱动机制：系统的运行由事件的产生与传播来驱动。组件之间通过注册监听特定事件，并在事件发生时做出响应来进行交互。例如在图形用户界面开发中，用户的点击、输入等操作都会触发相应的事件，由对应的事件处理程序进行处理。

响应特性与应用场景：具有很强的灵活性与响应及时性，能够快速适应系统状态的变化与外部事件的触发。适用于对实时性要求较高、系统行为需要根据外部事件动态调整的场景，如实时监控系统、游戏开发等。

14.4 架构风格与系统质量属性的关联

性能影响因素

通信开销与效率：不同架构风格下组件间的通信方式与开销对系统性能有着显著影响。例如，在分布式架构中，如果通信协议设计不合理或者网络拓扑结构复杂，可能导致数据传输延迟增加，从而降低系统的整体性能。

资源分配与利用：架构风格决定了系统资源的分配与利用方式。如某些架构风格可能采用缓存机制或者负载均衡策略来优化资源利用，提高系统的吞吐量与响应速度，满足对性能要求较高的系统需求。

可修改性的架构支撑

模块独立性：一些架构风格强调模块或组件的独立性，如分层架构和管道 - 过滤器架构，使得在修改某个部分时对其他部分的影响最小化，方便系统的升级与维护。

扩展点设计：良好的架构风格会预留扩展点，便于在系统需求发生变化时能够方便地添加新功能或修改现有功能，而无需对整个系统架构进行大规模重构。

安全性架构考量

访问控制机制：部分架构风格内置了访问控制机制，如客户端 - 服务器架构中服务器端可以对客户端的访问请求进行严格的权限验证与过滤，防止非法访问与数据泄露。

数据加密与传输安全：在某些对安全性要求极高的架构风格中，会注重数据在传输过程中的加密处理以及存储时的安全策略，确保系统数据的安全性与完整性。

十五、使用架构模型

15.1 基于属性的架构风格

风格特点与优势

属性驱动设计：基于属性的架构风格以特定的质量属性为核心关注点进行架构设计。例如，以高性能、高可用性或高安全性等质量属性为导向，构建系统的架构框架。

针对性优化：能够针对所需的质量属性进行有针对性的架构元素选择与结构设计。如为实现高可用性，可能采用冗余组件设计、故障转移机制等架构策略，从而有效提升系统在特定质量属性方面的表现。

与其他架构风格对比：与传统通用架构风格不同，它不是从结构或功能的常规分类出发，而是从质量属性需求出发倒推架构设计，更强调对非功能需求的满足与优化。

15.2 基于领域的架构风格

领域相关性体现：基于领域的架构风格紧密围绕特定的业务领域知识与需求构建架构。例如，在金融领域，会依据金融业务的交易流程、风险管理、账户管理等业务特点设计架构，包括专门的金融数据模型、交易处理组件等。

领域知识融合：将领域内的专业知识、规则与架构设计深度融合。使架构不仅符合软件设计的一般性原则，更能准确地反映业务领域的实际运作逻辑，提高系统对业务需求的适应性与支持能力。

可复用性与局限性：在特定领域内具有较高的可复用性，能够快速构建同领域的新系统或扩展现有系统。但由于与特定领域紧密绑定，跨领域应用时可能需要较大的调整与改造。

15.3 基于模式的架构风格

模式应用方式：基于模式的架构风格基于已有的软件设计模式构建架构。例如，采用工厂模式来创建对象实例，或者使用观察者模式实现对象间的事件通知与响应机制，将这些成熟的模式组合运用到架构设计中。

模式优势整合：整合了设计模式的优势，如提高代码的可维护性、增强系统的灵活性与扩展性等。通过遵循既定的模式规范，能够减少架构设计中的错误与不确定性，提高开发效率与系统质量。

模式选择与适配：需要根据系统的具体需求与场景合理选择与适配设计模式。不同的模式适用于不同的问题域，不当的模式选择可能导致系统架构的不合理或性能瓶颈。

15.4 基于参考模型的架构风格

参考模型依据：基于参考模型的架构风格以特定的行业标准或通用参考模型为依据进行架构设计。如在企业资源规划（ERP）系统中，可能依据 SAP 的 ERP 参考模型来构建系统架构，涵盖财务、采购、生产、销售等各个业务模块的架构设计规范。

标准化与规范化：遵循参考模型能够使架构设计符合行业的标准化要求，便于与其他相关系统进行集成与互操作。同时，参考模型提供了一种全面的架构框架，有助于系统的完整性与系统性设计。

灵活性与创新性权衡：在保证遵循参考模型基本框架的同时，需要在系统的灵活性与创新性方面进行权衡。过于严格遵循参考模型可能限制了系统对特殊需求与新兴技术的应用能力，而过度创新又可能导致与行业标准脱节。

15.5 基于视图的架构风格

视图导向设计：基于视图的架构风格以不同的架构视图为核心构建架构。例如，从模块视图关注系统的模块划分与结构组织，从运行时视图考虑系统运行时的组件交互与动态行为，从部署视图着眼于系统在硬件环境中的部署配置。

多视图协同：强调不同视图之间的协同与一致性。通过综合考虑各个视图的需求与特点，构建出全面、协调的系统架构，确保系统在功能、性能、部署等多方面的需求都能得到满足。

视图转换与映射：需要处理好不同视图之间的转换与映射关系。例如，如何从设计模型的模块视图准确地映射到部署视图的硬件节点配置，是保证架构实现的关键环节。

15.6 基于组件的架构风格

组件核心地位：基于组件的架构风格将组件作为架构的核心构建块。组件具有明确的接口与功能定义，通过组件的组合与交互实现系统的整体功能。如在一个电子商务系统中，可能有用户管理组件、商品管理组件、订单处理组件等。

组件复用与替换：强调组件的复用性，成熟的组件可以在不同系统或同一系统的不同部分重复使用，降低开发成本。同时，组件的可替换性使得系统能够方便地进行升级与维护，如替换性能更高的组件或修复有缺陷的组件。

组件间关系管理：注重组件间的关系管理，包括组件之间的接口设计、通信协议、依赖关系等。合理的组件关系设计能够提高系统的稳定性与可扩展性，如采用松耦合的组件关系可减少组件变更对系统的影响。

15.7 基于服务的架构风格

服务概念与特点：基于服务的架构风格以服务为基本单元构建架构。服务是一种自包含、可复用的软件单元，通过网络协议对外提供特定的功能。例如，在一个分布式服务架构中，可能有用户认证服务、数据查询服务、文件上传服务等。

服务组合与编排：强调服务的组合与编排来实现复杂的业务功能。通过将多个服务按照业务流程进行有序组合，形成业务服务链，能够快速响应业务需求的变化与扩展。如在一个在线旅游预订系统中，将酒店预订服务、机票预订服务、旅游套餐定制服务等进行组合编排。

服务治理与质量保障：需要重视服务治理，包括服务的注册与发现、服务的版本管理、服务的监控与度量等。以确保服务的质量与可靠性，如及时发现服务故障并进行切换或修复，保证服务的性能满足业务需求。

15.8 基于云计算的架构风格

云计算特性融入：基于云计算的架构风格充分融入云计算的特性，如弹性计算、按需服务、资源池化等。例如，设计能够根据业务负载自动弹性扩展或收缩的架构，利用云计算平台的资源池灵活分配计算、存储与网络资源。

云服务模式适配：根据云计算的不同服务模式（IaaS、PaaS、SaaS）进行架构适配。在 IaaS 模式下，架构设计更关注底层硬件资源的利用与管理；在 PaaS 模式下，侧重于平台服务的集成与开发环境构建；在 SaaS 模式下，则注重多租户的应用架构设计与数据隔离。

云安全与合规性：注重云安全与合规性要求。由于云计算环境的开放性与共享性，架构设计需要考虑数据安全、隐私保护、网络安全等多方面的安全措施，同时确保符合相关的法律法规与行业标准。

15.9 基于物联网的架构风格

物联网架构要素：基于物联网的架构风格针对物联网的特点进行设计，涵盖感知层、网络层与应用层等架构要素。感知层负责采集物理世界的数据，网络层负责数据的传输与通信，应用层则对数据进行处理与应用，如智能家居系统中的传感器、网络通信模块与智能控制应用的架构设计。

数据处理与传输挑战：面临数据处理与传输的特殊挑战，如海量感知数据的实时处理、低功耗网络通信协议的设计、异构设备的接入与管理等。需要采用相应的架构策略，如数据预处理技术、轻量级网络协议、设备管理平台等。

物联安全与隐私：重视物联安全与隐私保护。由于物联网设备的广泛分布与互联，容易遭受网络攻击与数据泄露风险，架构设计需要包括设备安全认证、数据加密、访问控制等安全机制，保护用户隐私与系统安全。

15.10 基于人工智能的架构风格

人工智能技术集成：基于人工智能的架构风格集成人工智能技术，如机器学习、深度学习、自然语言处理等。例如，在一个智能客服系统中，采用深度学习模型进行语义理解与回答生成，构建能够自动学习与优化的架构。

数据驱动与模型训练：强调数据驱动的架构设计，大量的数据用于训练人工智能模型。架构需要考虑数据的采集、存储、预处理与标注等环节，同时为模型的训练、评估与部署提供良好的环境与支持。

智能交互与决策能力：注重提升系统的智能交互与决策能力。通过人工智能模型的应用，使系统能够与用户进行自然语言交互，根据用户需求与环境信息做出智能决策，如智能推荐系统根据用户历史行为与偏好进行个性化推荐。

15.11 小结

多种风格综述：本章介绍了多种不同类型的架构风格，包括基于属性、领域、模式、参考模型、视图、组件、服务、云计算、物联网、人工智能等架构风格。

风格选择依据：在实际软件架构设计中，需要根据系统的目标、需求、应用场景以及质量属性要求等因素，合理选择合适的架构风格，以构建高效、可靠、可维护的软件系统。不同的架构风格在不同的方面具有各自的优势与局限性，综合运用多种风格或进行风格的创新融合也是未来架构设计的发展方向之一。

十六、结论

16.1 设计模式基础

概念与定义：设计模式是在特定环境下解决常见问题的通用方案，它是对反复出现的设计问题的抽象和总结，具有一定的普遍性和适用性。

目的与价值：旨在提高软件设计的质量和效率，促进软件的可维护性、可扩展性和可复用性。通过使用设计模式，开发人员可以避免重复劳动，快速构建稳定可靠的软件系统。

16.2 创建型模式

单例模式：确保一个类只有一个实例，并提供全局访问点。适用于资源共享、全局配置等场景，如数据库连接池、日志记录器等，可有效控制资源的创建和访问，避免资源浪费和冲突。

工厂模式：定义创建对象的接口，让子类决定实例化的类。将对象的创建和使用分离，提高了代码的灵活性和可扩展性，方便在不同条件下创建不同类型的对象，如根据用户需求创建不同风格的界面组件。

抽象工厂模式：提供创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。适用于创建一组产品族的场景，使得产品的创建和使用更加规范和统一，便于系统的扩展和维护，例如创建不同操作系统下的界面组件族。

建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。用于构建复杂对象时，可以分步构建对象的各个部分，然后组合成完整的对象，如构建具有多种配置选项的复杂产品对象。

原型模式：通过复制现有对象来创建新对象，减少创建对象的成本。适用于创建对象成本较高或对象初始化过程复杂的情况，如创建大量相似的游戏角色对象时，可以先创建一个原型，然后通过复制快速生成其他角色。

16.3 结构型模式

适配器模式：将一个类的接口转换成客户期望的另一个接口，使原本不兼容的类可以一起工作。在系统集成或升级时，用于解决新旧接口不匹配的问题，如将旧系统的接口适配到新系统的接口规范上。

桥接模式：将抽象部分与它的实现部分分离，使它们可以独立变化。适用于处理多维度变化的情况，如不同形状和不同颜色的图形绘制，可以将形状和颜色的变化分别独立处理，提高系统的灵活性。

组合模式：将对象组合成树形结构以表示 “部分 - 整体” 的关系，使得用户对单个对象和组合对象的使用具有一致性。常用于处理具有层次结构的对象，如文件系统中的文件夹和文件的管理，方便对整个目录树进行操作。

装饰器模式：动态地给一个对象添加一些额外的职责，就像给对象穿上一层外衣。在不改变原有对象结构的基础上，增强对象的功能，如给基本的图形绘制对象添加边框、阴影等装饰效果。

外观模式：为子系统中的一组接口提供一个统一的高层接口，使得子系统更容易使用。隐藏了子系统的复杂性，简化了外部对系统的访问，如提供一个统一的数据库操作接口，屏蔽了数据库内部的复杂连接和查询逻辑。

享元模式：运用共享技术有效地支持大量细粒度的对象，减少内存占用。适用于创建大量相似且可共享的对象，如游戏中的棋子对象，如果它们的外观和行为相同，可通过享元模式共享相同的棋子实例，降低内存开销。

16.4 行为型模式

责任链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将请求沿着链传递，直到有一个对象处理它为止，如在工作流系统中，不同的审批环节依次处理审批请求。

命令模式：将一个请求封装为一个对象，从而使不同的请求可以参数化、排队或记录日志等。将请求的发送者和执行者分离，方便对请求进行管理和控制，如在图形界面中，将各种操作命令封装成对象，便于撤销和重做操作。

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。用于处理简单的语言解析问题，如自定义配置文件的解析，将配置文件的语法规则定义并解释执行。

迭代器模式：提供一种方法顺序访问一个聚合对象中的各个元素，而无需暴露该聚合对象的内部表示。方便对集合类对象进行遍历操作，如遍历列表、数组等数据结构，使得遍历操作与数据结构的具体实现分离。

中介者模式：用一个中介对象来封装一系列的对象交互。使得各对象之间的耦合松散，只与中介对象交互，如在多人聊天系统中，通过中介者（服务器）来协调各个用户之间的消息传递。

备忘录模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。以便以后需要时能够恢复对象到先前的状态，如在文本编辑器中保存文档的历史版本，以便撤销操作时恢复。

观察者模式：定义了对象间的一种一对多的依赖关系，当一个对象的状态发生变化时，所有依赖它的对象都得到通知并自动更新。常用于实现事件驱动机制，如在图形界面中，当数据模型发生变化时，通知相关的视图进行更新。

状态模式：允许一个对象在其内部状态改变时改变它的行为。将对象的行为根据不同的状态进行封装，如电梯的运行状态不同时，其操作行为（开门、关门、上升、下降等）也不同。

策略模式：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。使得算法的变化不会影响到使用算法的客户，如在电商系统中，根据不同的促销策略计算商品价格。

模板方法模式：定义一个操作中的算法骨架，而将一些步骤延迟到子类中。使得子类可以在不改变算法结构的基础上，重写某些步骤以实现特定的功能，如在游戏开发中，定义游戏角色的基本行为框架，子类可根据不同角色特点重写部分行为。

16.5 设计模式的应用与整合

应用场景选择：在实际软件设计中，需要根据具体的问题和需求，选择合适的设计模式。不同的设计模式适用于不同的场景，如创建型模式用于对象创建，结构型模式用于处理系统结构，行为型模式用于处理对象间的交互和行为。

模式组合运用：多个设计模式可以组合使用，以解决复杂的软件设计问题。例如，在一个大型企业级应用中，可能同时使用外观模式来简化外部访问，适配器模式来处理系统集成，以及观察者模式来实现数据更新的通知机制等。

模式与架构协同：设计模式与软件架构相互关联，设计模式是架构的微观实现手段，架构为设计模式的应用提供了宏观框架。合理运用设计模式有助于实现软件架构的目标，提高系统的整体质量和性能。