**《软件架构实践》读书笔记**

# 什么是软件架构？

### 软件架构的定义：

软件架构是推理系统所需的一组结构，包括软件元素、它们之间的关系以及两者的属性。

### 软件架构之于软件的重要性：

架构作为业务目标与具体系统之间的桥梁，帮助实现业务目标。

架构决策不仅仅是早期的，也包括后期的，且不仅限于“重大”或“重要”的决策。

### 架构结构的分类：

（1）组件和连接器结构：

关注点：系统的运行时行为，包括组件如何交互以执行系统功能。

元素：主要的计算单元，可以是服务、对等方、客户端、服务器等。

连接器：组件之间的通信机制，如调用返回、进程同步操作、管道等。

主要问题：

确定主要的执行组件及其运行时交互；识别主要的共享数据存储；分析系统的可并行性和结构变化能力。

（2）模块结构：

关注点：系统的实现单元，即模块，以及它们如何分配计算职责。

元素：代码或数据单元，如类、包、层等。

关系：使用、泛化（继承）、“是xx的一部分”等。

主要问题：

确定每个模块的主要功能职责；分析模块间的依赖关系；评估系统可修改性的影响。

（3）分配结构：

关注点：软件元素如何映射到非软件结构，如硬件、团队和文件系统。

元素：软件元素与非软件元素，如处理器、开发团队等。

关系：分配和迁移。

主要问题：

软件元素在哪个处理器上执行；软件元素在开发、测试和构建过程中的存储位置；软件元素分配给哪些开发团队。

### 架构视图：

架构视图是对架构结构的一种表示，它聚焦于系统的特定方面，以支持特定的分析或决策过程。每个视图都是从不同的视角捕捉系统的一个或多个架构结构，视图包括：

逻辑视图：关注系统的功能性和数据处理方面，包括数据流和处理过程。

开发视图：关注系统的模块化，包括软件的组织和分配给不同开发团队的部分。

物理视图：关注系统的物理实例和分布，包括硬件、节点和通信链接

其中又分为几种视图类型：

模块视图：

展示了系统的模块化结构，包括模块的职责和它们之间的关系，用于理解系统的组织和如何进行开发和维护。

组件和连接器视图（C&C View）：

描述了系统的运行时组件和它们之间的交互方式，用于分析系统的动态行为，如性能和可用性。

分配视图：

展示了软件元素如何在硬件和网络中分布，用于理解系统的物理部署和资源利用。

行为视图：

描述了系统组件的行为和交互，通常使用状态图或活动图来表示，用于分析系统的动态方面，如并发性和同步。

数据视图：

专注于数据存储、数据流和数据结构，用于理解系统的持久性和数据完整性。

部署视图：

展示了系统的物理部署，包括硬件、节点和通信路径，用于分析系统的物理架构和部署问题。

架构视图在软件架构中至关重要，因为它们提供了多维度的系统理解，支持对关键质量属性的深入分析，促进了团队间的有效沟通与协作，指导了架构的设计和实现，记录了关键决策，支持系统的评估和验证，并帮助团队适应系统的变化和演进，从而提高了项目的成功率和构建出更高质量、更可靠的软件系统。

# 第二章 为什么软件架构很重要？

### 软件架构重要的十三个理由

驱动质量属性：架构对系统的质量属性起着决定性作用，是实现系统关键质量属性（如性能、可靠性、可维护性）的基石。

变更论证与管理：架构决策允许对系统的变更进行论证和管理，对系统的演进至关重要。

系统质量预测：通过对架构的分析，可以及早预测系统的质量，从而避免后期的质量问题和相关成本损失。

沟通工具：记录的架构增强了利益相关者之间的沟通。

早期设计决策：架构体现了系统最早期的设计决策，对后续开发有巨大影响。

约束实现：架构定义了对后续实现的约束，确保符合架构规定。

组织结构影响：架构决定了组织结构，反之亦然。

增量开发基础：架构为增量开发提供了基础。

成本和进度估算：架构是进行成本和进度估算的关键工件。

可转移、可重用的模型：架构可以创建为可转移、可重用的模型，成为产品线的核心。

组件集成关注：基于架构的开发将注意力集中在组件的集成上，有利于保持项目的整体性与一致性。

设计复杂性降低：架构通过限制设计备选方案，降低设计和系统复杂性。

培训新成员：架构可以成为培训新团队成员的基础。

我的个人理解：

架构给整个项目的各个方面提供了一个公用的参考框架，让各个开发部门共同协作，同时又对项目的整个生命周期产生影响，是一个长期且公用的架构，所以至关重要。

具体的理由将在后续章节解释。

# 第三章 理解质量属性

### 质量属性的定义

质量属性（QA）是可度量或可测试的系统属性，用于指示系统满足利益相关者需求的程度，除了系统的基本功能。质量属性是系统除了基本功能之外的重要特征，它们决定了系统是否能够在实际应用中满足用户和利益相关者的期望。

### 相比较于功能性

功能性描述系统完成预期工作的能力，而质量属性关注系统如何实现这些功能。功能性需求通常在开发计划中占据主导地位，质量属性也同样重要，因为它们影响系统的可维护性、移植性、扩展性和性能。

### 质量属性场景

（1）触发事件（Stimulus）：

触发事件是指引起系统响应的动作或条件。它可以是用户的输入、系统的内部状态变化、外部系统的交互等。

（2）触发源（Stimulus Source）：

触发源是触发事件的发起者。了解触发源有助于系统决定如何响应事件，因为不同的源可能需要不同的处理策略。

（3）环境（Environment）：

环境描述了场景发生的上下文条件，包括系统的状态、外部条件、时间等因素，这些条件影响系统的行为。

（4）工件（Artifact）：

工件是指触发事件影响的系统部分。它可以是整个系统、系统的一部分或系统中的特定组件。

（5）响应（Response）：

响应是系统对触发事件的直接反应。这个反应应该能够被明确描述和测量，以便于测试和验证。

（6）响应测量（Response Measure）：

响应测量是指对系统响应的量化度量，它可以是时间、资源消耗、成功率等，用于评估系统是否满足质量属性的要求。

### 质量属性实现方法

架构策略是一系列设计决策，它们专注于优化软件架构中的特定质量属性。这些策略针对单个属性，如性能或可维护性，提供具体的技术或方法来指导设计，以确保系统的行为和性能满足既定的质量标准。

策略的实施涉及对系统设计的细致调整，例如通过改变数据访问模式来提升响应速度，或者采用设计模式来增强系统的灵活性和可扩展性。由于策略专注于单一属性，它们可能会对系统中的其他质量属性产生连锁反应，因此实施时需要进行细致的权衡。

架构模式则提供了一种更为全面和经过验证的解决方案，用于解决在特定设计环境中经常出现的问题。这些模式描述了一组元素的角色、职责、关系以及它们的协作方式，通常涵盖了多个设计决策，并且能够对多个质量属性产生影响。

模式的应用可以帮助架构师避免从头开始设计，而是借鉴已经证明有效的架构解决方案。由于模式可能包含多个策略，它们在解决质量属性问题时往往需要在不同属性间进行权衡，以实现整体的架构平衡。

具体实施步骤：

（1）识别需求：首先，需要识别和定义系统的质量属性需求，这通常通过质量属性场景来完成。

（2）选择策略和模式：基于识别的需求，选择适当的策略和模式来实现这些质量属性。

（3）设计决策：做出具体的设计决策，这些决策将支持所选策略和模式的实施。

（4）权衡：在实施过程中，需要在不同的质量属性之间进行权衡，因为优化一个属性可能会对另一个属性产生负面影响。

（5）验证和测试：最后，通过测试和验证来确保实施的策略和模式达到了预期的效果，并满足质量属性的要求。

# 第四章 可用性

### 可用性的定义

可用性是软件的一种特性，指系统在需要时能够执行其任务的能力，是一个广义的概念，包括可靠性、鲁棒性，以及系统发生故障时的自我修复能力。可用性涉及到系统如何屏蔽或修复故障，确保服务中断时间在指定时间间隔内不超过预定值，可用性与信息安全性、性能和安全性密切相关。

常用的公式是：

可用性=

其中MTBF是平均失效间时间，MTTR是平均修复时间,计划的停机时间不计入可用性计算。

### 可用性策略

（1）故障检测策略：故障检测策略使系统能够识别出故障的存在，常见的策略包括：

监控（Monitor）：监控系统各个部分的运行状况，如处理器、进程、I/O、内存等。

Ping/Echo：通过在节点间交换异步请求/响应消息对，确定网络路径的可达性和往返延迟。

心跳（Heartbeat）：通过系统监控和被监控进程之间的定期消息交换来检测故障。

时间戳（Timestamp）：用于检测分布式系统中不正确的事件序列。

状态监控（Condition Monitoring）：检查进程或设备中的条件，或验证设计过程中的假设。

健全性检查（Sanity Checking）：检查组件的特定操作或输出的有效性或合理性。

投票（Voting）：比较来自多个来源的计算结果，决定使用哪些结果。

异常检测（Exception Detection）：检测改变正常执行流程的系统状况。

自检（Self-test）：组件运行程序测试自身是否正确操作。

（2）故障恢复策略：故障恢复策略关注于从故障中恢复，包括准备和修复策略以及重新引入策略，有：

冗余备件（Redundant Spare）：在主组件发生故障时，备用组件介入并接管工作。

回滚（Rollback）：系统恢复到先前已知的良好状态。

异常处理（Exception Handling）：处理检测到的异常，可能包括报告、纠正或屏蔽故障。

软件升级（Software Upgrade）：以不影响服务的方式实现对可执行代码的服务中升级。

重试（Retry）：假定导致失效的故障是暂时性的，并且重试操作可能会导致成功。

忽略错误行为（Ignore Faulty Behavior）：忽略从特定来源发送的虚假消息。

优雅降级（Graceful Degradation）：在组件失效时保持最关键的系统功能，放弃不太重要的功能。

重新配置（Reconfiguration）：通过将职责重新分配给保持正常运行的资源或组件，从失效中恢复。

（3）故障预防策略：故障预防策略旨在防止故障的发生。

从服务中删除（Removal from Service）：暂时将系统组件置于停止服务状态，以减轻潜在的系统失效。

事务（Transactions）：确保分布式组件之间交换的异步消息是原子的、一致的、隔离的和持久的。

预测模型（Predictive Model）：监控系统进程的运行健康状态，以确保系统在其标称操作参数内运行，并在系统接近临界阈值时采取纠正措施。

异常预防（Exception Prevention）：防止系统异常发生的技术。

增加能力集（Increase Competence Set）：提高组件的能力集，使其能够处理更多情况。

### 可用性模式

主动冗余（热备）：所有节点并行接收和处理相同输入，允许冗余备件与活动节点保持同步状态。

被动冗余（温备）：只有保护组的活动成员处理输入流量，冗余备件维护的状态仅与保护组中主动节点的状态松散耦合。

备用（冷备）：冗余备件在发生故障切换之前保持停止服务，需要较长时间来启动和同步状态。

三重模块冗余（TMR）：三个组件做同样的事情，投票逻辑检测三个输出状态之间的任何不一致。

断路开关：防止调用者尝试无数次，等待永远不会出现的响应，当认为系统正在处理故障时，它会中断无休止的重试循环。

可用性不仅仅是软件质量的一个方面，它是系统设计中不可或缺的核心要素，这要求架构师在设计初期就全面考虑系统的鲁棒性、故障恢复能力和预防措施，以确保系统在各种情况下都能稳定运行，这对于创建既可靠又用户友好的软件解决方案至关重要。

# 第五章 可部署性

### 可部署性的定义

可部署性是指软件能够在可预测和可接受的时间以及工作量内部署，并且如果新部署不符合规范，可以同样高效地回滚。

### 环境

（1）开发环境

开发环境是单个模块开发的起点，通常涉及单元测试以确保代码的正确性。这个环境模拟了真实世界的条件，但更受控，允许开发者在不影响主系统的情况下进行实验和测试。

（2）集成环境

一旦代码通过单元测试并经过审查，它会被提交到版本控制系统，并在集成环境中触发构建活动。在这里，持续集成服务器会编译新代码，并与服务的其他部分的最新兼容代码版本一起构建，形成可执行映像。集成测试确保各个模块能够协同工作，为后续的部署打下基础。

（3）暂存环境

暂存环境承担着测试整个系统质量的任务，包括性能测试、安全测试和许可证合规性检查。这个环境模拟了生产环境的运行条件，以验证新部署的功能和性能是否符合预期。对于嵌入式系统，暂存环境可能包括物理环境模拟器，为系统提供合成输入。

（4）生产环境

经过暂存环境的严格测试，应用程序最终会部署到生产环境。在这里，服务将受到密切监控，以确保其质量和性能满足业务需求。生产环境是软件生命周期的最终阶段，也是用户实际交互的地方，因此，它的稳定性和性能对业务至关重要。

每个环境承担不同的测试和验证任务，从开发环境中的单元测试到集成环境中的集成测试，再到暂存环境中的全面质量测试，最终到生产环境中的实际使用监控。这种分层的环境策略有助于逐步揭示和解决软件在不同阶段可能遇到的问题，从而降低风险并提高部署的成功率。

### 可部署性策略

（1）扩展推出（Canary Releases）：这种策略通过逐步推出新版本给一小部分用户，从而减少新部署的潜在风险。它允许团队在全面部署前验证新版本的性能和稳定性。

（2）回滚（Rollback）：当新部署出现问题时，能够迅速回滚到之前的稳定版本是至关重要的。这一策略要求系统能够追踪所有变更，并能够快速恢复到先前的状态。

（3）编写部署命令脚本（Scripted Deployments）：为了确保部署的一致性和可重复性，自动化的部署脚本是必不可少的。这些脚本记录了部署过程中的每一步，减少了人为错误，并提高了部署效率。

（4）管理服务交互（Managing Service Interactions）：在多版本服务并存的情况下，需要确保不同版本的服务能够正确交互。这可能涉及到版本协商、数据兼容性和接口适配等问题。

（5）包依赖项（Packaging Dependencies）：确保服务及其所有依赖项一起部署，可以避免运行时依赖冲突。这通常通过容器化、虚拟机或服务网格等技术实现。

（6）功能切换（Feature Toggling）：在新功能部署后，可能需要根据反馈或性能指标动态开启或禁用这些功能。功能切换允许在不重新部署服务的情况下控制功能的可用性。

### 可部署性功能

微服务架构（Microservices Architecture）：通过将系统分解为一系列小的、独立的服务，每个服务都可以独立部署和扩展，从而提高了系统的灵活性和可维护性。

蓝色/绿色部署（Blue/Green Deployment）：这种模式通过维护两个生产环境（蓝色和绿色）来实现无缝部署。当新版本准备就绪时，可以快速切换流量，从而减少部署中断。

滚动升级（Rolling Upgrades）：在这种模式下，服务的实例逐个更新，而不是一次性替换所有实例。这有助于维持服务的连续性，并允许在部署过程中监控新版本的健康状况。

# 第六章 能效性

### 能效性的定义

能效性是系统在设计和运行过程中对能源消耗进行管理和优化的能力。这一概念不仅关注技术层面的能源利用效率，也涵盖了环境影响和成本效益的考量。能效性直接关系到减少温室气体排放、降低运营成本、提高系统的可持续性、增强设备电池寿命和用户体验，以及满足法规遵从性。

### 能效性的策略

（1）资源监控

计量：通过传感器基础设施实时收集计算资源的能耗数据，从整个数据中心的能耗到单个服务器或硬盘驱动器的能耗。

静态分类：在无法实时收集数据的情况下，通过编目计算资源及其已知的能量特征来估计能耗，这些特征可以来源于基准测试或制造商规格。

动态分类：当静态模型不足以应对负载或环境条件的变化时，使用动态模型根据瞬态条件（如工作负载）来估算能耗。

（2）资源分配

减少使用：在需求减少时停用资源，如降低硬盘转速、关闭CPU或服务器，或将虚拟机整合到较少的物理服务器上。

发现服务：将服务请求与服务提供者匹配，考虑能源信息，允许请求者根据能源特征选择服务提供者。

调度资源：根据任务限制和优先级，动态地在计算资源之间分配任务，选择能效更高或能源成本更低的资源。

（3）减少资源需求

管理事件到达：通过控制事件的到达率来减少资源需求。

限制事件响应：对事件响应进行限制，可能涉及优先级排序或延迟某些响应。

减少计算开销：优化算法和代码以减少计算资源的使用。

提高资源使用效率：通过更有效地使用资源来减少能源消耗。

### 能效性的模式

（1）传感器融合：传感器融合模式利用多个传感器的数据来减少能耗。在移动应用和物联网系统中，低功耗传感器的数据可以用来推断是否需要从高功耗传感器收集数据，从而减少高能耗设备的使用。

好处：智能地减少高能耗设备的使用，降低整体能耗。

权衡：需要前期的复杂性来处理和比较多个传感器的数据，且可能牺牲一些数据质量。

（2）终止异常任务：此模式监控并终止异常耗电的应用程序，特别是在移动系统中，以防止它们无意中消耗过多能源。

好处：提供了一种管理未知能耗属性应用能耗的“故障安全”选项。

权衡：增加了系统监控的开销，可能与用户意图相悖。

（3）电源监控：电源监控模式通过监视和管理系统设备，减少它们处于活动状态的时间，智能地节省功耗。

好处：在不影响最终用户体验的情况下节省能源。

权衡：设备关闭后再启动可能会增加延迟，增加了系统设计的前期复杂性。

# 第七章 可集成性

### 可集成性的定义

可集成性是指软件组件或单元能够被有效地整合到一个系统中的能力，它涉及到的不仅是技术层面的集成，还包括了集成过程中的成本和技术风险。随着系统复杂性的增加，组件间的集成变得越来越频繁和必要，架构师必须在设计时就考虑到如何降低集成的难度和风险，确保系统的可扩展性和灵活性，同时控制未来集成工作的成本。

### 可集成性的评估方法

评估可集成性涉及以下几个关键方面：

（1）接口分析：检查组件间的接口，识别它们之间的语法兼容性、数据交换格式、通信协议和互操作性标准。

（2）依赖关系映射：确定组件间的依赖关系，包括直接和间接依赖，以及这些依赖关系如何影响系统的其他部分。

（3）差异度量：评估组件间在数据语义、行为模式和时间同步上的差异，这些差异可能导致集成问题。

（4）风险评估：基于接口的复杂性和依赖关系的差异，预测集成过程中可能遇到的风险和技术挑战。

（5）成本估算：根据差异度量和风险评估，估算集成活动可能涉及的成本，包括人力、时间和资源消耗。

（6）策略制定：基于评估结果，制定策略来降低集成的复杂性、风险和成本，这些策略可能包括组件重构、接口标准化和中间件引入。

### 可集成性的策略

（1）限制依赖关系：为了降低集成的复杂性和风险，限制系统组件之间的依赖关系至关重要。这可以通过以下方式实现：

封装：通过引入显式接口，减少对组件内部实现的依赖，从而降低更改的传播风险。

使用中介：中介组件可以打破直接依赖，减少组件间的耦合，如发布-订阅系统、数据库或服务发现机制。

限制通信路径：通过设计限制组件间的直接通信，例如在面向服务的架构中强制所有通信通过企业服务总线。

遵守标准：采用广泛认可的行业标准，减少因平台差异导致的集成问题。

抽象公共服务：为相似服务提供统一的抽象接口，隐藏具体实现的细节。

（2）适配：适配策略涉及调整组件以满足集成需求，包括：

发现服务：提供服务目录，帮助组件在运行时动态发现和绑定到其他服务。

剪裁接口：在不修改组件本身的情况下，通过添加或隐藏接口元素来适配不同组件的接口。

配置行为：允许组件在构建、初始化或运行时根据配置改变行为，以适应不同的集成场景。

（3）协调：协调策略关注如何管理和控制组件间的交互：

业务流程：使用工作流引擎和业务流程管理来协调服务的调用，减少服务间的直接依赖。

管理资源：通过资源管理器控制对关键资源的访问，确保资源在组件间公平分配和有效管理。

### 可集成性的模式

（1）包装器模式：包装器模式涉及创建一个封装特定组件的包装器，该包装器提供了一个与现有系统兼容的接口，同时隐藏了被包装组件的实现细节。这种模式允许在不修改组件本身的情况下，将其集成到不同的上下文中。

好处：能够将不兼容的组件集成到系统中，提供了灵活性和隔离性。

权衡：可能会引入额外的开销，因为所有的交互都需要通过包装器进行。

（2）桥接模式：桥接模式用于将抽象与实现解耦，使得两者可以独立变化。这种模式允许系统在运行时选择和替换组件的具体实现，而不影响使用组件的客户端代码。

好处：提供了灵活性，使得系统能够适应变化，尤其是在组件实现需要频繁更新或替换时。

权衡：增加了系统的复杂性，因为需要管理抽象和实现之间的映射关系。

（3）中介模式：中介模式引入一个中间组件，该组件负责协调多个客户端和服务器之间的通信。这种模式可以减少组件间的直接依赖，使得系统更加模块化。

好处：降低了组件间的耦合度，简化了组件间的通信。

权衡：可能增加了系统的复杂性，并可能影响性能，因为所有通信都需要通过中介进行。

(4)面向服务的架构（SOA）模式:SOA模式是一种将系统构建为服务集合的架构风格，其中服务是自包含的业务逻辑单元，可以通过网络进行访问。SOA支持服务的发现、绑定和执行。

好处：促进了服务的重用和系统的灵活性，使得新服务可以轻松集成到现有的系统中。

权衡：需要遵循一套标准和协议，可能会增加开发的复杂性。

(5)动态发现模式:动态发现模式允许系统在运行时动态地发现和绑定服务。这种模式支持服务的即插即用，使得系统能够灵活地适应服务的可用性变化。

好处：提高了系统的适应性和灵活性，尤其是在服务频繁变化的环境中。

权衡：需要额外的机制来管理和维护服务的注册和发现过程。

个人感悟：软件组件的集成能力对于项目的成败至关重要。架构师必须提前规划，以确保系统的可扩展性和灵活性，同时降低未来集成工作的风险和成本。这要求我们在设计时就考虑到组件之间的接口和依赖关系，以及如何通过策略和模式来优化这些关系。

# 第八章 可修改性

### 可修改性的定义

可修改性关注于降低软件系统进行更改的成本和风险，涉及对系统功能、平台、环境、质量属性、容量等方面的变更。大多数软件系统变更不仅发生在添加新功能时，也发生在修复缺陷、增强安全性、提高性能等方面。因此可修改性是软件架构中一个至关重要的质量属性。

### 可修改性策略

（1）增加内聚：增加内聚力的策略涉及重新分配模块间的职责，以降低单个变更影响多个模块的可能性：

拆分模块：如果一个模块包含不紧密相关的职责，拆分这样的模块可以降低未来变更的平均成本。

重新分配职责：将分布在不同模块中的相似职责集中在一起，以减少因变更而需要修改的模块数量。

（2）降低耦合：降低耦合的策略旨在减少模块间的依赖关系，从而降低变更的传播风险：

封装：通过引入显式接口，减少对模块内部实现的依赖，从而降低因变更而影响其他模块的可能性。

使用中介：中介组件可以打破模块间的直接依赖，减少耦合度。

抽象公共服务：为类似服务提供统一的抽象接口，减少模块间的直接交互。

限制依赖关系：限制模块间的可见性和授权，控制模块间的交互。

（3）延迟绑定：延迟绑定的策略关注于将值的绑定推迟到软件生命周期的更晚阶段，以降低变更的成本：

参数化：通过参数化函数，增加灵活性，允许在不修改代码的情况下进行变更。

组件替换：在生成脚本或文件中使用组件替换，以推迟绑定决策。

编译时参数化：允许在编译时根据参数值生成不同的代码版本。

切面：在不同的生命周期阶段应用横切关注点，如日志记录或事务管理。

配置时绑定：使用配置文件或参数设置来绑定值。

资源文件：在部署、启动或初始化时绑定资源文件中的值。

运行时绑定：包括发现服务、解释参数、共享存储库和多态性等策略，允许在运行时进行变更。

### 可修改性模式

（1）客户端-服务器模式：客户端-服务器模式由服务器组成，这些服务器向多个分布式客户端提供服务。这种模式支持动态连接、客户端之间的松散耦合，以及客户端和服务器的独立演进。服务器可以无状态地处理客户端请求，或者维护与客户端相关的状态信息。

好处：降低了服务器和客户端之间的耦合度，便于扩展和维护。

权衡：可能增加网络通信的复杂性和性能开销。

（2）插件（微内核）模式：插件模式涉及一个提供核心功能的核心组件和一系列可以通过固定接口添加功能的专用插件。这种模式允许系统的功能在不修改核心组件的情况下进行扩展。

好处：提供了一种受控的扩展机制，允许独立开发和部署插件。

权衡：增加了系统的复杂性，可能影响安全性和性能。

（3）分层模式：分层模式将系统划分为层，每层提供一组内聚的服务，并且层之间的交互受到严格控制。这种模式支持关注点分离，使得系统的不同部分可以独立开发和维护。

好处：提高了系统的模块化程度，便于重用和维护。

权衡：可能增加系统的性能开销，并且需要仔细设计以避免层之间的过度耦合。

（4）发布-订阅模式：发布-订阅模式是一种组件通过异步消息进行通信的架构模式。发布者发送消息，而订阅者根据兴趣接收这些消息。这种模式支持组件之间的松散耦合和灵活的交互。

好处：解耦了消息的生产者和消费者，便于系统扩展和变化。

权衡：可能影响消息传递的性能和可靠性，并且需要有效的机制来管理订阅和消息分发。

个人感悟：软件系统的灵活性和适应性很重要，可修改性涉及到项目管理、团队协作和长期规划，需要架构师在一开始就做好准备。

# 第九章 性能

### 性能的定义和场景

性能是指软件系统满足时限要求的能力，涉及系统对事件的响应时间，包括处理时间和阻塞时间，在计算机操作需要显著时间的场景中尤为重要。即使没有明确的性能要求，用户也会期望系统能够及时响应，由于性能问题通常在系统构建后被发现，所以架构时期的预设要很重要。

### 性能的策略

（1）控制资源需求：控制资源需求的策略旨在减少系统处理事件所需的资源量，从而提高性能：

管理工作请求：通过服务级别协议（SLA）限制进入系统的请求数量，确保系统不会过载。

管理采样率：降低从传感器接收数据的速率或每秒处理的视频帧数，以减少处理需求。

限制事件响应：对事件的处理速率进行限制，确保系统不会因处理过多事件而性能下降。

确定事件优先级：根据事件的重要性分配优先级，确保高优先级事件得到快速响应。

减少计算开销：优化算法和代码，减少处理每个事件所需的工作量。

限定执行时间：对算法的执行时间进行限制，以保证系统能够及时响应。

提高资源使用效率：通过优化算法和代码，提高资源的使用效率。

（2）管理资源：管理资源的策略关注于如何更有效地利用现有资源，以提高系统性能：

增加资源：通过增加处理器速度、内存容量或网络带宽来提升性能。

引入并发：通过多线程或分布式处理来并行处理请求，减少阻塞时间。

维护计算的多个副本：通过复制服务或服务器来减少资源竞争。

维护数据的多个副本：通过数据复制和缓存来减少数据访问的竞争。

绑定队列大小：控制排队的最大数量，以限制处理请求所需的资源。

调度资源：选择合适的调度策略，以确保资源得到合理分配和使用。

### 性能的模式

（1）服务网格模式：服务网格模式应用于微服务架构，通过在每个微服务旁边部署代理（挎斗）来处理服务间通信、监控和信息安全等横切关注点。这种模式允许业务逻辑与通用功能分离，简化了服务间的通信和管理。

好处：简化了微服务间的通信和监控，提高了性能，因为挎斗可以减少网络延迟。

权衡：增加了系统的复杂性，每个代理都会消耗一定的资源。

（2）负载均衡器模式：负载均衡器模式使用中介来分配客户端请求到服务器池中的不同实例，以平衡负载并提高系统的可用性和响应性。

好处：提高了系统的吞吐量和响应性，同时隐藏了服务器的失效。

权衡：负载均衡器本身可能成为性能瓶颈或单点故障。

（3）限流模式：限流模式通过控制对特定资源或服务的访问速率来管理工作请求，通常由中介（限流器）来监控和限制请求流量。

好处：防止系统过载，确保服务在高需求下仍能正常运行。

权衡：可能导致请求丢失或延迟，需要合理配置以避免性能问题。

(4)Map-Reduce模式:Map-Reduce模式专门用于处理大型数据集的分布式和并行排序，通过映射和化简两个阶段来执行数据分析。

好处：通过并行处理提高了对大数据集分析的效率。

权衡：对于小数据集或无法并行处理的数据集，Map-Reduce模式可能引入不必要的开销。

个人感悟：性能直接影响用户体验和业务成果，要提前考虑性能要求并施加适合的策略。

# 第十章 安全性

### 安全性的定义

安全性涉及系统避免进入导致或导致其环境中的参与者遭受损害、受伤或失去生命的状态的能力，包括检测这些不安全状态并从中恢复，以防止由此产生的伤害。所以安全性策略可以大致分为：不安全状态避免、不安全状态检测或不安全状态修正，通用场景包括数据源、触发事件、环境、工件、响应和响应度量等。

### 安全性的策略

（1）不安全状态避免：避免不安全状态的策略旨在预防系统进入可能导致伤害或损害的状态：

替代（Substitution）：使用更安全的保护机制（通常基于硬件）来替代具有潜在危险的软件功能。

预测模型（Predictive Modeling）：通过监控系统状态预测潜在问题，确保系统在其标称操作参数内运行，并提供早期预警。

（2）不安全状态检测：检测不安全状态的策略用于识别系统何时进入不安全状态：

超时（Timeouts）：确定组件操作是否满足时限约束，以便在组件未能满足其时限时引发异常。

时间戳（Timestamps）：在分布式系统中，用于检测不正确的事件序列。

条件监测（Condition Monitoring）：检查进程或设备中的条件，或验证设计期间的假设。

健全性检查（Sanity Checking）：检查特定操作结果或组件输入输出的有效性或合理性。

比较（Comparison）：通过比较多个同步或复制元素产生的输出来检测不安全状态。

（3）抑制：抑制策略旨在限制与已进入的不安全状态相关的危害：

冗余（Redundancy）：使用多个组件来提高系统的容错能力，包括副本、功能冗余和分析冗余。

限制后果（Consequence Limitation）：通过中止、降级和屏蔽等策略来限制不安全状态可能导致的不良影响。

屏障（Barriers）：通过防火墙和互锁等策略来阻止问题传播。

（4）恢复：恢复策略用于将系统恢复到安全状态：

回滚（Rollback）：在检测到故障时恢复到先前已知的良好状态。

修复状态（State Repair）：修复错误状态，然后继续执行。

重新配置（Reconfiguration）：在发生失效时，通过将逻辑架构重新映射到正常运行的资源上来恢复资源。

### 安全性的模式

（1）传感器冗余模式：传感器冗余模式通过复制关键传感器来增强系统的安全性，确保即使某个传感器发生故障，系统仍然能够依赖其他传感器继续运行。这种模式可以防止因单个传感器的故障而导致整个系统失效。

好处：提高了系统对传感器故障的容错能力，增加了安全性。

权衡：增加了系统的成本和复杂性，因为需要处理多个传感器的输入。

（2）监视器-执行器模式：监视器-执行器模式涉及两个软件组件：一个执行器控制器负责计算，另一个监视器负责在执行器控制器的输出被发送到物理执行器之前检查其合理性。这种模式通过分离计算和验证步骤来增强安全性。

好处：增加了一个额外的检查层，有助于捕捉和防止潜在的错误。

权衡：需要额外的开发和维护工作，可能会增加系统的响应时间。

（3）分离安全模式：分离安全模式通过将系统分为安全关键部分和非安全关键部分来降低系统认证的成本和复杂性。这种模式允许对系统的安全关键部分进行更严格的测试和认证。

好处：减少了系统认证的工作量和成本，因为只有安全关键部分需要经过严格认证。

权衡：需要在系统架构中明确划分安全和非安全部分，这可能影响系统的整体设计。

# 第十一章 信息安全性

### 信息安全性的定义

信息安全性是衡量系统保护数据和信息免受未经授权的访问，同时仍向授权的人员和系统提供访问权限的能力，涉及保密性、完整性和可用性（CIA）三个基本特征。攻击可以采取多种形式，包括未经授权访问数据、修改数据或拒绝合法用户服务，因此信息安全性尤为重要。

### 信息安全性的策略

（1）检测攻击：检测攻击的策略旨在识别和预警潜在的安全威胁：

入侵检测：通过比较网络流量或服务请求模式与已知的恶意行为模式，检测可能的入侵行为。

拒绝服务攻击检测：识别网络流量模式或特征，与已知的拒绝服务（DoS）攻击特征进行比较。

验证消息完整性：使用校验和或哈希值等技术确保消息、资源文件和配置文件的完整性。

检测消息传递异常：通过检查消息传递的时间和连接/断开连接的异常数量，检测潜在的中间人攻击。

（2）抵抗攻击：抵抗攻击的策略旨在防止未授权访问和操纵数据：

识别参与者：通过用户ID、IP地址或其他标识符识别系统的外部输入来源。

验证参与者：通过密码、数字证书、双因素身份验证等方法确保参与者身份的真实性。

授权参与者：确保经过身份验证的参与者有权访问和修改数据或服务。

限制访问：通过限制资源的访问点数量或控制流量类型，减少系统的攻击面。

限制暴露：减少可以通过单个接入点访问的数据或服务量，以降低损害影响。

加密数据：对数据和通信应用加密，以保护数据的保密性。

分离实体：通过物理分离、虚拟机或空气间隙等手段，限制攻击的范围。

验证输入：对系统接收的输入进行清理和检查，以抵御SQL注入和跨站脚本攻击等。

更改凭据设置：强制用户定期更改安全设置，以防止攻击者利用默认或已知的设置。

（3）对攻击做出反应：对攻击做出反应的策略涉及在检测到攻击时采取的措施：

撤销访问权限：在认为攻击正在进行时，限制对敏感资源的访问，即使是合法用户也不例外。

限制登录：在检测到多次登录失败尝试时，限制从特定计算机的访问。

通知参与者：在检测到攻击时，通知操作员、其他人员或合作系统，以便采取行动。

（4）从攻击中恢复：从攻击中恢复的策略旨在恢复服务和数据的完整性：

审计：记录用户和系统的行为及其影响，以帮助追踪攻击者并识别攻击者。

不可否认性：确保消息的发送者不能否认发送过消息，接收者也不能否认收到过消息，通常通过数字签名和可信第三方认证实现。

### 信息安全性的模式

（1）拦截验证器模式：拦截验证器模式在消息的源和目标之间插入一个软件元素（拦截器），负责验证消息的完整性和安全性。这个模式通常用于实现验证消息完整性的策略，但也可以包括检测入侵和检测拒绝服务等策略。

好处：提供了一个集中点来检查所有传入和传出的消息，增强了对消息完整性和安全性的控制。

权衡：引入了额外的处理步骤，可能会影响系统的性能，并且需要定期更新以应对新的威胁。

（2）入侵防御系统模式：入侵防御系统（IPS）模式使用一个独立的组件来识别和分析任何可疑活动，并在活动被视为恶意时进行阻止和报告。这些系统寻找整体使用中的可疑模式，而不仅仅是异常消息。

好处：能够实时监控和响应潜在的安全威胁，提供对已知攻击模式的防御。

权衡：需要不断更新模式数据库以保持有效性，可能会产生性能开销，并且可能不完全适合特定应用的需求。

（3）分离模式：分离模式通过物理分离、虚拟机或空气间隙等手段，将系统的不同部分隔离开来，以限制攻击的范围和影响。

好处：减少了攻击者横向移动的能力，增强了对敏感数据和系统的保护。

权衡：可能会增加系统的复杂性和管理成本，并且可能影响系统组件之间的互操作性。

（4）分布式责任模式：分布式责任模式将安全责任分散到系统的多个部分，以确保没有单一的故障点。

好处：提高了系统的弹性，即使部分系统受到攻击，其他部分仍能继续运行。

权衡：需要更复杂的协调和通信机制，可能会增加系统的管理和维护难度。

个人感悟：在数字化时代，信息安全不仅是技术问题，也是法律和道德问题，架构师必须将安全性放在无时无刻都在思考的地方。

# 第十二章 可测试性

### 可测试性的定义

可测试性是指软件系统通过测试来展示其故障的难易程度，如果一个系统能够容易地“暴露”其故障，那么它就是可测试的，可测试性不仅关乎发现故障，还包括复制错误和缩小错误可能原因的范围。可测试性的策略旨在提高测试的效率和效果。

### 可测试性的策略

（1）控制和监视系统状态：这些策略关注于如何控制软件组件的输入和如何监视其输出及内部状态，以便进行有效的测试。

专用接口：为测试目的提供专门的接口，以便能够控制组件的内部状态和捕获其变量值。

记录/回放：记录系统的状态和行为，以便在测试中回放和重现故障。

本地化状态存储：将系统的状态信息集中存储，以便于测试时的控制和观察。

抽象数据源：通过抽象化数据源，使得在测试中可以轻松替换为测试数据。

沙盒：在隔离的环境中运行系统或其部分，以进行测试而不影响生产环境。

可执行断言：在代码中嵌入断言，以自动检查程序状态是否符合预期。

（2）限制复杂性：这些策略旨在减少系统设计的复杂性，从而降低测试的难度。

限制结构复杂性：通过简化组件之间的依赖关系，减少循环依赖，封装外部依赖，降低系统的耦合度。

限制不确定性：识别并减少系统中的不确定性来源，如不受控的并行性，以提高测试的可预测性。

### 可测试性的模式

（1）依赖注入模式：依赖注入模式通过将组件的依赖关系从组件内部转移到外部，使得测试时可以轻松替换这些依赖项。这种模式利用控制反转，允许测试时注入测试用的模拟对象或桩。

好处：简化了组件的测试，因为可以控制组件的依赖项，更容易模拟和监控组件的行为。

权衡：可能增加系统的复杂性，因为需要管理依赖关系的注入机制。

（2）策略模式：策略模式允许在运行时更换算法或行为。通过定义一系列算法，并使它们在运行时可互换，可以在测试中轻松切换到测试专用的策略。

好处：提高了代码的灵活性，使得测试不同的行为或算法变得更加容易。

权衡：可能会增加系统的复杂性，因为需要维护多个策略实现。

（3）拦截过滤器模式：拦截过滤器模式在请求到达最终处理组件之前，允许对请求进行预处理和后处理。这种模式可以用来插入测试相关的逻辑，如日志记录、监控或验证。

好处：提供了一种灵活的方式来插入测试逻辑，而不需要修改组件的内部代码。

权衡：可能会影响系统的性能，尤其是在处理大量请求时。

# 第十三章 易用性

### 易用性的定义

易用性关注的是用户完成期望任务的难易程度以及系统提供的用户支持类型，包括学习系统特性、高效使用系统、最小化用户错误的影响、使系统适应用户需求以及增强信心和满意度，是提高系统质量（或用户对质量的感知）以及最终用户满意度的最经济、最简便的方法之一

### 易用性的策略

（1）支持用户主导：这些策略着重于增强用户在与系统交互中的主动性：

“取消”策略：系统必须能够响应用户的取消命令，及时终止正在进行的操作，并释放相关资源。

“撤销”策略：系统应允许用户撤销最近的操作，恢复到之前的状态。这可能涉及到保存状态快照或记录可逆操作。

“暂停/恢复”策略：对于长时间运行的任务，系统应提供暂停和恢复功能，以便用户可以根据需要暂时中断任务。

“聚合”策略：系统应能够将多个对象或操作聚合为一组，以便用户可以一次性对整个组执行操作，减少重复劳动。

（2）支持系统主导：这些策略关注于系统如何主动为用户提供支持：

“维护任务模型”：系统应维护一个任务模型，以便理解用户的目标和上下文，从而提供适时的帮助和建议。

“维护用户模型”：系统应构建和维护一个用户模型，以适应不同用户的需求和偏好，提供个性化的体验。

“维护系统模型”：系统应有一个自我模型，用于预测和解释系统行为，并向用户提供适当的反馈。

### 易用性的模式

（1）模型-视图-控制器（MVC）模式：MVC模式是一种广泛应用的易用性模式，它通过将应用程序的数据（模型）、用户界面（视图）和控制逻辑（控制器）分离，来实现关注点分离。

好处：MVC模式使得用户界面的更改不影响业务逻辑，从而提高了系统的可维护性和可扩展性。它允许多个视图共享同一个模型，便于实现不同设备或平台的适配。

权衡：对于复杂的用户界面，MVC模式可能会增加系统的复杂性，因为数据和逻辑需要在多个组件之间同步。

（2）观察者模式：观察者模式允许对象（观察者）订阅并响应另一个对象（主题）的状态变化，从而实现状态的自动更新。

好处：这种模式简化了组件之间的通信，使得视图组件可以实时反映模型的变化，提高了系统的响应性和交互性。

权衡：如果管理不当，观察者模式可能会导致内存泄漏和性能问题，因为观察者需要正确注册和注销。

（3）备忘录模式：备忘录模式提供了一种保存和恢复对象状态的机制，常用于实现撤销功能。

好处：通过保存状态的快照，备忘录模式使得用户可以轻松撤销操作，提高了操作的可逆性和系统的易用性。

权衡：备忘录模式可能会增加系统的存储负担，因为需要保存额外的状态信息，特别是在处理大量数据时。

个人感悟：易用性牵扯到用户的心理和行为，易用性好的系统可以显著提高用户的满意度和效率，易用性差的系统则可能导致用户的挫败感和效率低下。

# 第十四章 其他质量属性

除了性能、可用性和安全性等之外的质量属性，这些多样化的质量属性同样对软件系统的整体效能和用户满意度有着重要影响。

可构建性：这一属性衡量了架构对于快速和高效开发过程的适应性，影响着将架构转化为满足所有需求的工作产品的成本和时间。

概念完整性：反映了架构设计的一致性，有助于提高架构的可理解性，并在实现和维护过程中减少混乱，增加可预测性。

开发可分发性：随着全球分布式团队开发的普及，开发可分发性成为了一个关键的质量属性，它要求系统设计能够最小化团队间的协调需求，降低模块间的耦合度，以适应不同团队的工作。

# 第十五章 软件接口

### 接口的概念

接口是软件架构中的核心概念，定义了元素如何相遇、交互、通信和协调，接口不仅仅是代码层面的抽象，还代表了元素之间的契约和交互的边界。

同样的，一个实体可以有多个接口，以服务于不同类别的参与者。

一个接口的资源有两个基本属性：

资源语法指的是资源的签名，它包括资源的名称、参数的名称和数据类型等信息，这些信息对于编写正确使用资源的程序是必要的。

资源语义描述了调用资源的结果，包括数据的赋值、状态的变化、发出的事件或消息、未来行为的改变以及人类可观察到的结果。

同时接口的资源由以下几部分组成：

操作是传输控制和数据的机制，通常也会返回结果，并且可能失败。接口设计需要明确如何检测错误。

事件通常是异步的，可以是传入的，表示接收到的消息或流元素的到达；也可以是传出的，用于通知监听者内部发生的事件。

属性是接口的元数据，例如访问权限、度量单位或格式假设，它们影响操作的行为和元素的状态。

随着软件的发展，接口也会演进，接口演进的技术包括弃用、版本控制和扩展等。

### 接口交互方式

（1）远程过程调用（RPC）：远程过程调用（RPC）模拟了命令式语言中的过程调用，允许程序员编写调用远程过程的代码，就像调用本地过程一样。RPC的核心优势在于它的透明性，使得网络中的位置差异对调用者来说是不可见的。调用被转换为发送到远程元素的消息，结果再通过消息回调用者。RPC经历了多次演变，最新的版本如gRPC，支持二进制传输参数、异步操作、认证、双向流和流量控制等特性，使用HTTP 2.0作为传输层。

（2）表述性状态转移（REST）：REST是一种网络服务协议，基于Web使用的原始协议。REST的核心是一组约束，这些约束定义了元素之间交互的方式：

统一接口：所有交互都使用相同的形式，通常是HTTP，资源通过URI指定。

客户端-服务器：客户端发起请求，服务器提供资源。

无状态：每个请求从客户端到服务器必须包含所有必要的信息，以便服务器理解请求并给出响应。

可缓存：响应必须定义它们是否可以被缓存。

分层系统：通信可能通过多个层次进行，例如代理和网关。

按需代码（可选）：服务器可以按需将执行代码（如JavaScript）发送给客户端。

REST通常与HTTP协议一起使用，HTTP命令与CRUD操作相对应，例如POST对应创建，GET对应读取，PUT对应更新/替换，PATCH对应更新/修改，DELETE对应删除。

# 第十六章 虚拟化

### 引言

在20世纪60年代，计算领域面临的挑战是如何在一台物理机器上让多个独立应用程序共享资源，如内存、磁盘、I/O通道和用户输入设备。由于资源无法共享，导致计算机的利用效率低下，大多数应用程序只使用了大约10%的可用资源。虚拟机和容器的出现正是为了解决这一共享问题，它们的目标是将应用程序相互隔离，同时允许资源共享。

### 共享资源

我们通常关心共享的有四种资源：中央处理器（CPU）、内存、磁盘存储和网络连接。处理器共享通过线程调度机制实现，而虚拟内存技术则通过内存管理硬件将进程的地址空间划分为页面，并在物理内存和辅助存储之间交换页面。磁盘共享和隔离通过磁盘控制器实现，网络隔离则是通过IP地址和端口号来识别消息。

### 虚拟机

虚拟机允许在一台物理计算机中执行多个模拟的或虚拟的计算机。物理计算机被称为“主机”，虚拟机被称为“客户机”，而虚拟机管理程序则是虚拟机的操作系统。虚拟机管理程序直接在物理计算机硬件上运行，通常被称为“裸机”或“类型1”虚拟机管理程序。虚拟机管理程序执行两个主要功能：管理在每个虚拟机中运行的代码和管理虚拟机本身。虚拟机的启动方式与裸机物理机的启动方式相同，都是通过引导加载程序将操作系统代码从磁盘读入内存。虚拟机提供了一个CPU、内存、I/O设备和一个网络连接，从虚拟机内部的操作系统和软件服务的角度来看，似乎软件是在裸机物理机内部执行。虚拟机对架构师有两个主要影响：性能和关注点分离。虚拟化允许架构师将运行时资源视为商品，将供应和部署决策推迟到另一个人或组织。

### 容器

容器是一种在减少镜像传输时间和启动时间的同时保持虚拟化大部分优势的机制。容器被打包成可执行的容器镜像进行传输，容器运行时引擎充当一个虚拟化的操作系统。与虚拟机不同，容器共享相同的操作系统内核，操作系统可以加载到裸机物理机或虚拟机上。

容器的分配和启动速度比虚拟机快得多，这使得它们适合于动态分配和释放的场景。容器映像的大小很小，只包括支持我们想要运行的服务所必需的那些程序和库。容器运行时引擎的接口已由开放容器倡议标准化，这使得由一个供应商的软件包创建的容器能够在另一个供应商提供的容器运行时引擎上执行。

虚拟机对物理硬件进行虚拟化，而容器则对操作系统进行虚拟化；虚拟机可以运行任何操作系统，而容器目前仅限于Linux、Windows或iOS；虚拟机中的服务通过操作系统功能启动、停止和暂停，而容器中的服务通过容器运行时引擎功能启动和停止。

# 第十七章 云计算和分布式计算

### 云基础

云服务提供商运营的数据中心拥有数以万计的物理设备，规模的限制因素是其消耗的电力以及设备产生的热量。云数据中心由一排排机架组成，高速网络交换机连接着这些机架。云提供商将其数据中心划分为“区域”，云区域既是一种逻辑结构，也是一种物理结构，有助于减少访问服务的网络延迟并遵守某些监管限制，其分为公有云、私有云和混合云：

公有云由云服务提供商拥有和提供，向任何同意服务条款并且能够为使用服务付费的人提供基础设施服务。

私有云由一个组织拥有并运营，仅供该组织的成员使用。

混合云方法是一种混合模式，其中一些工作负载在私有云中运行，其他工作负载在公有云中运行。

### 云的拓展

（1）分布式计算和负载均衡器

负载均衡器解决了单个服务实例收到太多请求的问题，通过拥有该服务的多个实例并在它们之间分配请求。负载均衡器可以是独立系统，也可以与其他功能捆绑在一起，必须非常高效，因为它位于从客户端到服务的每条消息的路径上。

（2）分布式系统中的状态管理

当一个服务能够同时处理多个客户端请求时，状态管理就变得很重要。状态可以存储在每个服务实例中（有状态的服务），每个客户端中（无状态的服务），或者在服务和客户端之外的数据库中持久化（无状态的服务）。

（3）分布式系统中的时间协调

确切知道当前时间看似是一项简单的任务，但实际上并非易事。网络上两个不同设备的时钟读数肯定会有所不同。网络时间协议（NTP）用于在通过局域网或广域网连接的不同设备之间同步时间。

（4）分布式系统中的数据协调

解决分布式机器之间共享资源的问题涉及复杂的分布式协调算法。这些算法的设计正确性非常复杂，而且由于编程语言和网络接口语义的微妙之处，即使实现一个已被证明的算法也很困难。

# 第十八章 移动系统

### 移动系统的特性

（1）能源：移动系统的能源管理是一个关键的架构问题，因为它们通常依赖于有限的电池供电。这与其他系统的主要区别在于能源的有限性和不可再生性，尤其是在移动设备如智能手机和笔记本电脑中。

监测电源：移动系统需要能够监测其能源来源的状态，尤其是在电池供电的设备中。这包括了解电池的剩余电量，以便在电量低时通知用户，并采取节能措施。

控制能源使用：移动系统必须能够控制能源消耗，这可能涉及降低功耗或在必要时关闭非关键功能。例如，减少屏幕亮度或降低处理器性能可以延长电池寿命。

容忍断电：移动系统应能够妥善处理突然断电的情况，包括在电源恢复后能够快速重启，恢复到正常运行状态。

（2）网络连接性：移动系统的网络连接性是其核心功能之一，因为它们需要在移动过程中与其他设备和网络保持通信。

支持的通信接口数量：移动系统可能需要支持多种通信协议，但应仅包含必要的接口以优化功耗和空间。

动态选择合适的协议：系统应能够根据当前环境和条件动态选择最合适的通信协议。

可修改性：考虑到通信协议的快速发展，系统应设计为能够轻松更新和替换通信相关的组件。

（3）传感器和执行器：移动系统通常配备有多种传感器和执行器，这些是它们与物理世界交互的主要方式。

创建环境的准确表示：系统必须能够根据传感器数据创建准确的环境表示，并据此做出响应。

传感器数据和执行器命令的安全性和隐私性：确保传感器数据的安全性和隐私性，特别是在涉及个人数据时。

降级操作：在传感器失效或无法读取时，系统应能够进入降级模式，以确保安全运行。

（4）资源：移动系统的资源受限于其物理尺寸、重量和成本。

资源与性能的权衡：在资源有限的情况下，架构师必须在性能和资源消耗之间做出权衡。

功能与资源的分配：决定哪些功能应该分配给哪些资源，以及如何在系统的不同部分之间共享资源。

（5）生命周期：移动系统的生命周期包括硬件选择、测试、部署更新和日志记录，这些都与固定系统有所不同。

硬件选择的影响：硬件选择对软件架构的影响，尤其是在早期阶段。

测试的复杂性：移动系统的测试可能更加复杂，需要考虑多种设备和网络条件。

更新的部署：确保系统能够安全、高效地部署更新，同时保持数据一致性和系统安全性。

日志记录的重要性：在移动系统中，日志记录对于事件调查和系统分析至关重要。

# 第十九章 架构重要需求

### 获取重要需求

1. 从需求文档中收集架构重要需求（ASRs）

需求文档提供了系统预期行为和功能的初步理解，架构师需要从这些文档中提取出可能对系统架构产生重大影响的关键信息。

使用情况：用户角色、系统模式、国际化和语言差异。

时间方面：及时性和元素协调。

外部元素：与外部系统、协议、传感器或执行器、中间件的交互。

网络方面：网络属性和配置，包括安全属性。

编排方面：处理步骤和信息流。

安全属性：用户角色、权限和认证。

数据方面：持久性和时效性。

资源方面：时间、并发性、内存占用、调度、多用户、多活动、设备、能源使用、软资源和可扩展性需求。

项目管理方面：团队组建计划、技能组合、培训和团队协调。

硬件选择方面：处理器、处理器系列、处理器的演进。

1. 通过访谈利益相关者收集架构重要需求

当需求文档不足以提供足够的信息或者在项目初期质量属性需求尚未明确时，架构师需要直接与利益相关者沟通

准备：确定关键利益相关者，并制定访谈提纲。

执行：进行面对面或远程访谈，记录关键信息。

分析：从访谈中提取需求，特别是那些可能影响架构设计的需求。

反馈：将收集到的需求反馈给利益相关者，以确认理解和准确性。

1. 通过理解业务目标收集架构重要需求

业务目标是推动系统建设的根本原因，直接影响架构的设计和实现，架构师需理解这些目标，以确保架构能够支持组织的愿景和战略。

1. 在效用树中捕获架构重要需求

定义效用：以系统的总体价值或目标作为效用树的根节点。

细化质量属性：从效用出发，列出系统需要展现的主要质量属性。

记录具体需求：在每个质量属性下，记录该质量属性的具体细化内容和具体的架构重要需求。

# 第二十章 设计架构

### 属性驱动设计

（1）评审输入：在开始设计之前，首先要确保所有架构驱动因素都是可用的并且是准确的。这些因素包括设计目的、功能需求、质量属性场景、约束条件和架构关注点。这一步是确保设计过程有坚实基础的关键。

（2）建立迭代目标：每次设计迭代都应该有一个明确的目标，这个目标通常与满足一部分架构驱动因素相关。明确的目标有助于集中设计活动，并确保设计决策与这些目标保持一致。

（3）选择系统元素：基于驱动因素，选择一个或多个系统元素进行优化。这些元素可能是模块、组件或其他架构元素，它们的选择将直接影响架构的质量和效率。

（4）选择设计概念：识别和选择可以满足特定驱动因素的设计概念，如设计模式、参考架构或外部组件。这一步骤需要评估不同的设计选项，并选择最适合当前设计目标的概念。

（5）实例化架构元素：将选定的设计概念实例化为具体的架构元素，并为这些元素分配职责。这包括定义元素的属性和接口，以及它们之间的关系。

（6）绘制视图草图并记录设计决策：在设计过程中，创建视图草图以可视化架构结构，并记录下关键的设计决策和基本原理。这些记录对于后续的分析和沟通至关重要。

（7）分析当前设计：对当前迭代的设计进行分析，确保它满足了迭代目标和设计目的。这一步可能涉及技术审查、风险评估和质量属性的验证。

# 第二十一章 评估架构

### 评估架构的定义

“架构评估”是确定架构在多大程度上适合其预期目的的过程。架构对系统和软件工程项目的成功极其重要，因此停下来确保你正在设计的架构能够提供所有预期的功能是有意义的。

这就是评估的作用，它基于对各种替代方案的分析，并利用一些成熟的方法分析架构。每一个架构都伴随着风险，架构评估的输出包括识别架构中的风险部分，评估就像一份保险。

### 关键评估活动

（1）理解架构和需求：评估活动的第一步是确保所有参与者对当前架构和业务需求有深入的理解。这通常涉及对架构文档的审查、架构师的演示，以及与项目团队的讨论。

（2）确定评估驱动因素：评估团队需要确定将指导评估的驱动因素。这些驱动因素通常基于架构重要需求（ASRs），特别是那些与质量属性相关的需求。这些需求可能已经文档化，或者需要在评估过程中确定。

（3）分析架构对需求的满足程度：对于每个已识别的质量属性场景，评估团队需要分析架构是否能够满足这些场景。这包括评估架构的实际实现以及预测的架构行为。评估团队会提出问题，以验证架构是否能够满足场景，并识别可能的风险或问题。

（4）记录潜在问题：在分析过程中，评估团队会记录所有潜在的问题和风险。这些问题可能涉及架构的不足、设计决策的潜在影响，或对特定质量属性的担忧。这些问题将作为后续讨论和决策的基础。

（5）评估决策的重要性和风险：评估活动还包括对决策的重要性和潜在风险的评估。这涉及到对架构决策的影响进行量化，以及对替代方案的考虑。评估团队需要确定哪些决策对架构的成功至关重要，以及这些决策可能带来的风险。

（6）制定风险缓解策略：一旦识别出潜在的问题和风险，评估团队需要与项目团队合作，制定缓解策略。这可能包括设计变更、额外的测试或监控措施，以及对项目计划的调整。

（7）沟通评估结果：评估活动的最后一步是将评估结果传达给所有相关利益相关者。这包括准备评估报告、展示风险和问题，以及提出建议的行动方案。沟通的目的是确保所有利益相关者都了解评估发现，并同意采取的后续行动。

### 参与评估人员

（1）内部架构师：内部架构师对组织的业务目标和技术环境通常有深刻的理解。他们可以在项目的关键阶段进行评估，帮助识别和解决架构问题。由于他们参与项目的日常工作，内部架构师可以在设计过程中（如属性驱动设计过程）进行持续的评估。

（2）同行评审团队：同行评审可以由项目团队成员或组织内的其他专业人员组成。这种评估方式的优势在于，它能够利用团队成员的集体智慧和不同视角。同行评审可以在项目的特定阶段进行，例如在完成重要的设计里程碑后。

（3）外部专家：外部专家或咨询公司可以提供客观的评估视角，他们可能带来新的见解和最佳实践。外部评估人员不受组织内部偏见的影响，可能更敢于提出敏感问题。他们的专业知识和经验可以帮助识别内部团队可能忽视的风险和问题。

（4）跨职能团队：跨职能团队由具有不同专业背景的成员组成，包括开发人员、测试人员、运维人员和业务分析师。这种多元化的团队能够从多个角度评估架构，确保评估结果全面且平衡。

（5）利益相关者代表：利益相关者代表参与评估可以确保评估结果符合实际的业务需求和用户期望。他们可以提供关于系统如何满足业务目标和用户需求的宝贵见解。

（6）评估工具和自动化：除了人工评估外，还可以使用自动化工具来辅助评估过程。这些工具可以分析架构文档、代码库和配置，以识别潜在的风险和问题。

### 架构权衡分析方法（ATAM）

ATAM的关键特点：

多利益相关者的参与：ATAM要求项目决策者、架构师和架构利益相关者共同参与，确保评估结果能够全面反映各方的关注点和需求。

以质量属性为中心：ATAM将质量属性需求作为评估的出发点，通过构建质量属性效用树和场景，明确架构需要满足的具体目标。

结构化的方法：ATAM提供了一套清晰的步骤和模板，指导评估团队系统地分析架构，确保评估过程的一致性和可重复性。

风险识别：ATAM着重于识别和分析架构决策对质量属性的影响，从而揭示可能的风险和权衡点。

ATAM评估活动分布在四个阶段：

合作与准备阶段：在这个阶段，评估团队与项目关键决策者共同制定评估的细节，包括评估的时间表、参与人员、后勤安排等。

评估阶段：评估团队与项目决策者会面，开始信息收集和分析。在这个阶段，评估团队将使用一系列结构化的访谈和分析活动来理解架构和业务目标。

利益相关者参与阶段：在这个阶段，更广泛的利益相关者群体参与进来，提供他们对架构的见解和质量属性需求。

跟进阶段：评估团队生成并交付最终报告，总结评估过程和结果，包括识别的风险、非风险和权衡点。

ATAM的产出包括：

明确的业务目标：评估过程中明确阐述的业务目标，为项目提供了清晰的方向。

质量属性场景：以质量属性场景形式表达的已确定优先级的质量属性需求，为评估架构提供了具体的依据。

风险和非风险的识别：通过评估活动识别的架构风险和非风险点，为项目的风险管理提供了支持。

风险主题：从识别的风险中提炼出的风险主题，揭示了架构甚至整个项目中的系统性弱点。

架构决策与质量需求的映射：将架构决策与质量属性需求相映射，为理解架构设计提供了透明度。

# 第二十二章 记录架构

### 架构文档

架构文档是沟通架构决策、设计和意图的关键工具，对于确保软件系统的成功至关重要，便于架构师与开发团队、管理层、维护人员以及其他利益相关者之间互相沟通，架构文档的受众包括开发人员、测试人员、、维护人员、项目经理、最终用户等。

### 符号表示

（1）非正式符号：非正式符号指的是使用通用绘图和编辑工具，根据特定系统的视觉惯例绘制的视图。这些视图通常是图形化的，如方框和线条图，它们易于创建和理解，但可能缺乏严格的语义定义，因此不适合进行形式化分析。

（2）半正式符号：半正式符号采用标准化的图形符号来表达视图，这些符号规定了图形元素和构造规则，但并不提供完整的语义处理。UML（统一建模语言）及其系统工程辅助工具SysML（系统建模语言）是半正式符号的典型例子。这些符号允许进行基本的语法分析，但不能完全满足语义分析的需求。

（3）正式符号：正式符号使用具有精确语义（通常基于数学）的符号来描述视图。这种符号允许对语法和语义进行形式化分析，从而支持自动化工具进行架构分析和代码生成。架构描述语言（ADL）是正式符号的一种，它们为架构表示提供图形词汇和底层语义，有时专门为特定的架构视图设计。

# 第二十三章 管理架构债务

### 确定是否存在架构债务问题

（1）识别架构债务的迹象

频繁的变更：如果某些组件或模块频繁地需要变更，这可能表明它们与系统的其他部分过于耦合，或者设计不够灵活。

高错误率：如果特定部分的代码经常出现错误，这可能意味着这些部分的设计存在问题，难以维护。

低开发效率：如果开发团队在实现新功能或修改现有功能时进展缓慢，这可能是因为架构缺乏必要的抽象或模块化。

技术债务的传闻：团队成员可能会报告感觉到架构中存在问题，即使这些问题尚未通过定量数据得到证实。

（2）使用设计结构矩阵（DSM）识别债务

静态依赖关系：通过静态代码分析工具，可以识别组件之间的直接依赖关系，这些关系在DSM中表示为结构性链接。

演化依赖关系：通过分析版本控制系统中的变更历史，可以识别随时间共同变化的组件，这些关系在DSM中表示为演化链接。

（3）分析DSM以识别问题区域

稀疏性与密集性：一个健康的架构在DSM中应该表现出稀疏性，意味着大多数组件之间的依赖关系较少。相反，一个密集的DSM可能表明存在架构债务。

下三角矩阵：理想的DSM应该是下三角的，表明组件主要依赖于较低层次的组件，而不是相反。如果DSM中有大量的条目出现在对角线上方，这可能表明存在循环依赖或其他架构问题。

（4）结合定性和定量分析

定性分析：通过与架构师和开发人员的讨论，可以收集关于架构问题的主观见解。

定量分析：通过分析问题跟踪数据和版本控制历史，可以收集关于变更频率和错误率的客观数据。

# 第二十四章 架构师在项目中的角色

### 给架构师的建议

（1）与项目经理保持良好关系：了解项目经理的任务和关注点，并作为架构师支持这些任务和关注点。

（2）理解项目管理知识领域：在PMBOK列出的知识领域中，架构师应扮演积极角色，提供专业意见和支持。

（3）增量式架构发布：与利益相关者合作，确定发布节奏和每个项目增量的内容。确保第一个架构增量包括模块分解和使用视图，以及初步的C&C视图。

（4）早期关注质量属性：利用影响力确保早期发布处理系统最具挑战性的质量属性要求，避免开发周期后期出现架构意外。

（5）适应敏捷开发：在敏捷项目中，架构师应实现适当的敏捷性，通过执行属性驱动设计的迭代来开始项目，支持项目的增量发展。

（6）平衡敏捷原则：在敏捷原则中找到与架构实践相符合的点，对于与架构相冲突的原则保持警惕。

（7）分布式开发的协调：在分布式开发中，重视团队职责分配和模块依赖关系的管理，最小化全球分布式团队所拥有的模块之间的依赖关系。

（8）重视文档作用：在分布式开发中，文档成为协调团队的重要手段，确保文档编写良好、组织有序且得到恰当传播。

（9）适应远程工作趋势：随着远程工作和分布式开发的普及，架构师需要适应新的工作模式，确保沟通和协调的有效性。

（10）持续沟通与反馈：确保开发人员之间的沟通和协调，征求关于进度、问题和风险的反馈，监督文档工作。

# 第二十五章 架构能力

### 架构师的个人能力

（1）职责

技术职责：

设计、评估和记录架构。

处理和改进现有系统。

管理需求和评估技术。

非技术职责：

支持项目管理。

管理团队和支持组织业务。

提供技术领导力和团队建设。

（2）技能

沟通技巧：包括对外和对内的沟通能力。

人际交往能力：涉及团队合作和冲突解决。

工作技能：包括领导力、工作负载管理和企业环境中的卓越表现。

（3）知识

计算机科学知识：架构概念、软件工程、设计和编程知识。

技术和平台了解：特定技术和平台的一般知识。

组织背景和管理知识：领域知识、行业知识、业务知识和领导管理技术。

（4）组织架构能力

人员相关：招聘、职业发展、认证和绩效评估。

流程相关：建立架构实践、明确职责、建立审查委员会和架构里程碑。

技术相关：维护可复用架构的存储库和提供架构工具资源。