**附加思考题**

**功能性需求与质量属性的关系？**

**功能性需求**

功能性需求描述了系统必须执行的具体功能和行为。这些需求通常包括：

* 用户可以执行的操作
* 系统必须支持的业务流程
* 数据处理和存储的能力

**质量属性**

质量属性描述了系统在功能之外的特性和性能标准。这些通常包括：

* **性能**：系统的响应时间和吞吐量
* **可用性**：系统的正常运行时间和故障恢复能力
* **安全性**：系统抵御攻击和保护数据的能力
* **可维护性**：系统的易修改和扩展性
* **可扩展性**：系统处理增加负载的能力
* **可用性**：用户对系统的易用性和学习曲线

**功能性需求与质量属性的关系**

1. **相互影响**：

* 功能性需求的实现方式会直接影响质量属性。例如，选择一种特定的数据存储方案可能会影响系统的性能和可扩展性。
* 质量属性的要求可能会限制功能性需求的实现。例如，为了提高安全性，可能需要增加用户认证步骤，这可能会影响用户体验。

1. **权衡与优化**：

* 在设计系统时，通常需要在不同的质量属性之间进行权衡。例如，提高性能可能会增加系统的复杂性，从而影响可维护性。
* 架构师需要找到一个平衡点，以满足最关键的质量属性，同时实现所有必要的功能性需求。

1. **优先级**：

* 在需求分析阶段，识别和优先考虑关键的质量属性是至关重要的。这将指导架构设计和技术决策。
* 不同的项目会有不同的优先级。例如，金融系统可能会优先考虑安全性，而实时游戏可能更关注性能。

1. **验证与验证**：

* 功能性需求通常通过功能测试进行验证，而质量属性则需要通过性能测试、安全测试等进行验证。
* 确保质量属性得到满足通常需要特定的设计和实现策略，例如使用缓存来提高性能，或使用加密来提高安全性。

因此，功能性需求和质量属性在软件开发中是紧密相连的，架构师需要在设计初期就考虑两者的关系，以确保系统能够满足所有的业务和技术需求。

**质量属性与软件架构的关系？**

**可用性（Availability）**

可用性是指系统正常工作的时间所占的比例。可用性会遇到系统错误、恶意攻击、高负载等问题的影响。可用性面临的主要问题有：

* 物理层失效：比如数据库服务器宕机、停电、网络故障
* 恶意攻击：例如DOS（Deny of Service）攻击
* 软件的设计问题或BUG：比如错误的资源控制锁导致某个资源长期被占用、各种core dump、out of memory、out of stack
* 升级或日常维护

针对这些问题，为了增加可用性，需要考虑:

* 如何设计故障转移（failover），一般可以使用冗余来消除系统中的单点故障。可以是各种冷热备份、分布式系统。
* 如何设计异常处理。异常处理是一个很大的话题，为了支持高可用性，我们应该如何处理异常呢？举个例子，一个网站要处理用户的订单，然而由于数据库服务器的故障，虽然前端的服务一切正常，可是订单无法处理。好的异常处理是把用户的订单请求记录下来，发给人工处理，或者等待数据库恢复后自动处理，并告诉用户订单已经处理，并有可能迟延，请求得到用户的谅解。
* 如何应对不稳定的网络连接。

**互操作性（Interoperability）**

互操作性指的是系统内或者系统之间不同的组件可以有效地进行信息交换，通常是以服务（Service）的形式来进行的。互操作性的关键因素包括通信协议、接口定义、数据格式的定义等等，而标准化是实现互操作性的重要手段。

实现互操作性的主要挑战有以下这些方面：

* 系统内部或者和已有的旧系统（legacy system）之间的数据定义不一致
* 系统的边界模糊，模块之间耦合严重，导致数据冗余
* 缺乏标准，或者各方对标准的实现和认识不一致

为了实现互操作性，我们需要：

* 定义标准的数据格式和语义
* 定义标准的服务接口，并使用基于服务的架构
* 设计高内聚，低耦合的模块以获得最大的灵活性和可重用性

**可维护性（Maintainability）**

可维护性有两个不同的角度，一个是指从软件用户和运维人员的角度，另一个是从软件开发人员的角度。

从用户和运维人员的角度，软件的可维护性是指软件是不是容易安装、升级、打补丁，有了问题是不是容易修复，能不能很容易的获得支持。

从开发人员的角度，软件的可维护性是指软件的架构是不是清楚简单，代码是不是容易阅读，有了问题是不是容易定位错误的原因，有没有可以提供帮助的文档，等等。

软件系统可维护性的主要问题有：

* 模块之间紧耦合导致无法或很难对单独的模块进行修改、替换和升级
* 在高层直接使用底层协议和接口，导致无法替换物理设备实体
* 没有有效的分层和责任的划分，导致一个肿大的模块以及巨大的代码出现在同一个文件甚至函数中
* 没有帮助和设计文档
* 为了兼容旧系统而不得不同时存在两个以上的复杂的代码栈甚至技术不同的实现

随着云原生架构的流行，运维在软件中所占比例也越来越重要，如果增加软件的可维护性是架构设计的重中之重。

**性能（Performance）**

性能也许是软件开发中最被重视的质量属性，也是最特殊的一个。我们通常以系统执行某操作所需要的响应时间（latency）或者在某单位时间所能完成的任务的数量（throughput）来定义性能指标。

性能和其它的质量属性的相关性很高，有一些会对性能产生正面影响，有一些则是负面的。

一般而言，计算机提供了许多的资源，包括CPU、内存、硬盘等等，提高性能的核心就是充分利用这些资源。要保证对资源的使用是正确和有效的。通常提高性能的考虑包括：

* 利用缓存（空间换时间）
* 设计高效的资源共享、多线程、多进程、锁
* 异步
* 减少模块见得信息传递
* 使接口设计传递最小所需的信息
* 增加系统的可伸缩性，是系统能够有效的部署在分布式的资源上

另外我们还需要考虑另一个性能，就是程序员实现功能的性能(开发效率)。随着软件的发展，现在的程序员可以更高效的实现功能需求。一方面编程语言和方法在不断进步，另一方面大量的可重复使用的组件、服务、开源的库使得想在实现同样的功能的时间和需要的开发人员的数量比以前极大的缩小了。

**伸缩性（Scalability）**

伸缩性要求软件系统能够跟着所需处理的工作量相应的伸缩。例如如果计算机是多CPU多核心的，软件是否能够相应的利用到这些计算资源。另一个方面就是软件是不是能够部署到分布式的网络，有效的利用网络中的每一个节点的资源。

有两个方向的伸缩，垂直和水平。

在垂直方向的伸缩（scale up）是指提高单节点的处理能力，比如提高CPU主频和内核数，增大内存，增大磁盘容量等等。

在水平方向的伸缩（scale out）通常是指通过并发和分布的方式来增加节点以提高处理能力。Hadoop就是一个很好地水平伸缩的例子。

设计高伸缩性的软件时，我们可以考虑：

* 避免有状态的服务或组件
* 使用配置文件决定组件的的部署和关系
* 考虑支持数据的分区
* 避免同一层的责任跨越不同的物理实体

在云时代，软件的伸缩性越发重要。

**可测试性（Testability）**

可测试性顾名思义就是指软件是否容易测试。

什么样的软件是不可测试的呢？举个例子来说，一个数据可视化的组件，就是把数据以图表的形式展现出来。有一种数据可视化的类型使用力导向的算法（force-directed）把数据以网络拓扑图的形式展现出来，该算法使用一些随机的参数来模拟节点的初始位置，并通过迭代计算生成最终layout的结果。也就是说每次的layout结果都是不一样的。这样的实现是无法测试的，因为如果测试主要以比对图形为基础，每次测试都会把输出的图形和基准图形进行比较，如果不一致则认为出错或者要修改基准。随机算法虽然从功能上讲并没有错误，但是在这样的测试方法下是无法满足可测试性的要求的。

David Catlett提出了一个SOCK模型可以有效地帮助我们了解可测试性的要素：

* Simple：代码越简单越容易测试。
* Observable：软件系统应该是可观测，无法观测也就无法衡量。
* Control：软件系统应该是可以控制的，尽可能多的把控制权暴露给测试模块。
* Knowledge：测试人员或者模块需要更多地理解被测试模块，理解的越多也就越容易测试（白盒测试）。

当我们设计软件的时候，需要定义哪些质量属性是我们希望实现的，切记，质量属性并非越多越好。一般来说找到最重要的三个来构建软件就好了，而且"鱼和熊掌不能得兼"，各个质量属性之间有可能是互相矛盾或者互相影响的。分布式数据库中的CAP理论就是一个典型的例子。对于一个分布式的计算系统可不能同时满足一致性（Consistency）、可用性（Availability）、分区容忍性（Partition Tolerance）。

**什么是质量属性？**

软件质量属性，也称软件评估属性，是系统架构设计师必须掌握的核心知识点之一，这些质量属性的具体含义是：

1. 性能（Performance）效率指标，是指系统的响应能力，处理任务所需时间或单位时间内的处理量。
2. 可靠性（Reliability）是指软件系统在应用或错误面前，在意外或错误使用的情况下维持软件系统功能特性的基本能力。
3. 容错（Fault-tolerant）出现错误后仍能保证系统系统继续运行，且自行修正错误。
4. 健壮性（Robustness）是指在处理或环境中，系统能够承受压力或变更的能力，错误不对系统产生影响，按既定程序忽略错误。
5. 可用性（Availability）是系统能够正常运行的时间比例。
6. 安全性（Security）是指系统向合法用户提供服务的同时能够阻止非授权用户使用的企图或拒绝服务的能力。
7. 可修改性（Modification）是指能够快速地以较高的性能价格比对系统进行变更的能力。
8. 可维护性（Maintainability）局部修复使故障对架构的负面影响最小化。
9. 可拓展性（Extendibility）因松散耦合更易实现新特征/功能，不影响架构。
10. 可移植性（Portability）适用于多样的环境（硬件平台、语言、操作系统）。
11. 结构重组（Reconstructability）不影响主体进行的灵活配置。
12. 可变性（Changeability）总体架构可变，体系结构经扩充或变更成为新体系结构的能力。
13. 功能性（Functionality）需求的满足程度，是系统所能完成所期望工作的能力。
14. 互操作性（Inter-operation）是指系统与外界或系统与系统之间的相互作用能力，通过可视化或接口方式提供更好的交互操作体验。
15. 易用性（Usability）是衡量用户使用一个软件产品完成指定任务的难易程度。
16. 可测试性（Testability）是指软件发现故障并隔离、定位其故障的能力特性，以及在一定的时间和成本前提下，进行测试设计、测试执行的能力。

**如何理解质量属性？**

1. **明确需求**：

* 与利益相关者沟通，了解他们对系统的期望。
* 确定哪些质量属性最为关键（例如，性能、可用性、安全性等）。

1. **分类质量属性**：

* **性能**：系统的响应时间、吞吐量和资源利用率。
* **可用性**：系统能够正常运行的时间比例。
* **安全性**：保护数据和系统免受未经授权的访问。
* **可维护性**：系统的易修改性和可扩展性。
* **可移植性**：系统在不同环境下运行的能力。
* **可用性**：用户能够有效使用系统的能力。

1. **定义质量属性场景**：

* 描述具体的场景来明确质量属性如何在实践中体现。例如，性能场景可能包括在高负载情况下的响应时间。

1. **评估和权衡**：

* 不同的质量属性可能相互冲突（例如，高性能和高安全性可能需要权衡）。
* 使用架构评估方法（如ATAM）来分析和权衡这些属性。

1. **选择架构风格和模式**：

* 根据关键质量属性选择合适的架构风格（如微服务、事件驱动等）和设计模式（如缓存、负载均衡）。

1. **实施和验证**：

* 实施架构设计，确保它能够满足定义的质量属性。
* 通过测试和监控验证系统是否达到预期的质量属性。

1. **持续改进**：

* 随着需求变化和技术进步，持续评估和改进系统以保持或提升质量属性。

**如何理解架构决策的7个分类？**

1. 结构决策（Structural Decisions）

结构决策关注系统的整体组织和主要组件。这包括：

* 系统的高层次分解
* 模块化策略
* 组件之间的关系和接口定义
* 系统的层次结构

例如，决定采用微服务架构还是单体架构就是一个典型的结构决策。

1. 行为决策（Behavioral Decisions）

行为决策涉及系统的动态方面，包括：

* 系统如何响应外部刺激
* 控制流程
* 数据流
* 状态管理

比如，选择同步或异步通信模式，或者决定使用事件驱动架构，都属于行为决策。

1. 设计元素决策（Design Element Decisions）

这类决策涉及系统中重复出现的设计问题的标准解决方案，通常包括：

* 设计模式的选择和应用
* 框架的使用
* 中间件的选择

例如，决定在系统中使用观察者模式或者选择使用特定的ORM框架。

1. 质量属性决策（Quality Attribute Decisions）

质量属性决策直接影响系统的非功能需求，如：

* 性能
* 安全性
* 可靠性
* 可扩展性
* 可维护性

比如，为了提高系统的可伸缩性而选择使用负载均衡器，或者为了提高安全性而实施多因素认证。

1. 技术栈决策（Technology Stack Decisions）

这类决策涉及系统开发和运行所需的技术选择，包括：

* 编程语言
* 数据库
* 服务器
* 开发工具
* 部署平台

例如，选择使用Java作为主要开发语言，或者决定使用PostgreSQL作为数据库。

1. 交互决策（Interaction Decisions）

交互决策关注系统与外部世界的交互方式，包括：

* 用户界面设计
* API设计
* 与其他系统的集成方式
* 协议选择

比如，决定使用RESTful API还是GraphQL，或者选择如何设计系统的命令行接口。

1. 实施策略决策（Implementation Strategy Decisions）

这类决策涉及如何将架构转化为实际的代码和系统，包括：

* 编码标准
* 测试策略
* 持续集成/持续部署（CI/CD）策略
* 版本控制策略

例如，决定采用测试驱动开发（TDD）方法，或者选择特定的代码审查流程。

理解和应用这7个分类可以帮助我们：

1. 确保在架构设计过程中考虑到所有重要方面。
2. 更好地组织和文档化架构决策。
3. 在团队中更有效地沟通架构决策。
4. 在评审和改进架构时提供一个结构化的框架。