**《软件架构实践》读书笔记**

在当今快速发展的技术领域，软件架构的重要性日益凸显，它不仅关系到软件产品的质量，还直接影响到产品的可维护性、可扩展性和适应变化的能力。《软件架构实践》一书为我们提供了深入理解软件架构的宝贵资源，它不仅涵盖了软件架构的基本概念和重要性，还详细讨论了质量属性、架构解决方案、可扩展的架构实践，以及架构与组织之间的关系。这本书一共有六个部分共计26章，每一章都深入浅出地讲解了关于软甲架构不同方面的知识。我主要阅读了这本书的第3章：理解质量属性、第4章：可用性、第5章：可部署性以及第8章：可修改性，并记录了一些学到的知识和个人的感悟，

**第3章 理解质量属性**

许多因素决定了系统架构中必须提供的质量。这些质量比功能更重要，功能是系统能力、服务和行为的基本陈述。尽管功能和其他质量密切相关，但功能通常在开发计划中占据主要地位。然而，这种偏好是短视的。系统经常被重新设计，不是因为它们在功能上存在缺陷（替换成的通常在功能上是相同的），而是因为它们难以维护、移植或扩展；或者它们运行太慢了；或者它们已被黑客入侵。

质量属性（QA）是可度量或可测试的系统属性，用于指示系统（在除了系统基本功能之外）满足利益相关者需求的程度。可以将质量属性视为：在沿着利益相关者感兴趣的某个维度上衡量产品的“效用”。

**1.功能性**

功能性是系统完成预期工作的能力。在所有需求中，功能性与架构的关系最奇怪。

首先，功能性并不决定架构。也就是说，给定一组必需的功能，可以创建无尽的的架构来满足该功能。至少，可以通过多种方式划分功能，并将子功能分配给不同的架构元素。

事实上，如果功能是唯一重要的事情，根本不需要将系统划分为多个部分：一个没有内部结构的单一块就可以了。相反，我们将系统设计为协作的架构元素（模块、层、类、服务、数据库、应用程序、线程、对等节点、层等）的结构化集合，以使它们易于理解并支持各种其他用途。

虽然功能性独立于任何特定结构，但它是通过为架构元素分配职责来实现的。这个过程产生了最基本的一个架构结构——模块分解。

尽管职责可以任意分配给某个模块，但当其他质量属性很重要时，软件架构会限制这种分配。例如，系统经常（或可能总是）被分割，以便几个人可以合作地构建它们。架构师对功能性的兴趣在于它如何与其他质量交互并约束其他质量。

**2.质量属性考量**

正如系统的功能不会在没有适当考虑质量属性的情况下独立存在一样，质量属性也不会独立存在；它们与系统的功能有关。如果功能需求是“当用户按下绿色按钮时，将显示”选项“对话框”，则性能 QA 注解可能会描述对话框的显示速度；可用性 QA 注解可能会描述此功能允许失败的频率以及修复速度；易用性 QA 注解可能会描述学习此功能的难易程度。

1. 为属性提供的定义不可测试。说一个系统是“可修改的”是没有意义的。每个系统对于一组变更都是可修改的，并且对于另一组变更则是不可修改的。在这方面，其他质量属性也是类似的：系统在某些缺陷方面可能是健壮的，而对于其他的缺陷可能是脆弱的，等等。
2. 讨论通常集中在某个特定问题属于哪种质量上。对系统的拒绝服务攻击是可用性层面的、性能层面的、安全性层面还是易用性层面的？所有四个属性社区都将声称对拒绝服务攻击具有“所有权”。在某种程度上，所有这些都是正确的。但是，这种关于分类的争论并不能帮助作为架构师的我们，去理解和创建架构性解决方案来实际管理关注的属性。
3. 每个属性社区都开发了自己的词汇表。性能社区有“事件”到达系统，安全社区有“攻击”到达系统，可用性社区有“错误”到达，易用性社区有“用户输入”。所有这些实际上可能指的是相同的事件，但它们使用不同的术语来描述。

**3.指定质量属性需求：质量属性场景**

质量属性场景有六个部分：

1. 触发事件（stimulus）：我们使用术语“触发事件”来描述到达系统或项目的事件。触发事件可能是：对性能社区来说的事件，对易用性社区来说的用户操作，或者对安全社区来说的攻击，等等。我们使用相同的术语来描述待开发质量的激励行为。因此，对可修改性的触发事件是对修改的请求；可测试性的触发事件是开发单元的完成。
2. 触发源（Stimulus source）：触发事件必须有一个来源——它必须来自某个地方。某个实体（人类、计算机系统或任何其他参与者）必须产生触发事件。触发事件的来源可能会影响系统如何处理它。来自受信任用户的请求不会受到与不受信任用户的请求相同的审查。
3. 响应（Response）：响应是由于触发事件的到来而发生的活动。响应是架构师承诺满足的。它由系统（运行时质量）或开发人员（开发时质量）响应触发事件时应执行的职责组成。例如，在性能方案中，事件到达（触发事件），系统应处理该事件并生成响应。在可修改性方案中，修改请求到达（触发事件），开发人员应实现修改（没有副作用），然后测试和部署修改。
4. 响应测量（Response measure）：当响应发生时，它应该以某种方式可测量，以便场景可以测试，也就是说，这样我们就可以确定架构师是否实现了它。对于性能，这可能是对延迟或吞吐量的测量；对于可修改性，可能是进行、测试和部署修改所需的人工或时间。
5. 环境（Environment）： 环境是场景发生的一组条件。这通常是指运行时状态：系统可能处于过载状态或正常运行，或处于某种其他相关状态。对于许多系统，“正常”操作可以指多种模式之一。对于这些类型的系统，环境应指定系统在哪种模式下执行。但环境也可以指系统根本不运行的状态：当它处于开发、测试、刷新数据或两次运行之间为电池充电时。环境为场景的其余部分设置上下文。例如，在代码被冻结以进行发布后到达的修改请求的处理方式，可能与冻结之前到达的请求不同。组件的连续第五次失效的处理方式可能与该组件的第一次失效不同。
6. 工件（Artifact）：触发事件到达某个目标。这通常仅指系统或项目本身，但如果可能的话，越精确会越有帮助。工件可以是系统的集合、整个系统或系统的一个或多个部分。失效或变更请求可能只影响系统的一小部分。数据存储中的失效的处理方式可能与元数据存储中的失效不同。对用户界面的修改可能比对中间件的修改响应的更快。

**第4章 可用性**

可用性是指软件的一种特性。这是一个广义的概念，包括通常所说的可靠性（尽管它可能包括其他考虑因素，如定期维护导致的停机时间）。可用性建立在可靠性的概念之上，增加了恢复的概念，即当系统发生故障时，它会自行修复，修复可以通过各种方式完成。

可用性还包括系统屏蔽或修复故障的能力，使其不会成为失效，从而确保累积服务中断期在指定的时间间隔内不会超过所需值。该定义包含可靠性、鲁棒性和任何其他涉及不可接受失效概念的质量属性的概念。

失效是指系统与其规范的偏差，这种偏差在外部可见。确定失效已经发生需要环境中的一些外部观察者。

失效的原因称为故障（fault）。故障可以是所考虑系统的内部故障，也可以是外部故障。故障发生和失效发生之间的中间状态称为错误。故障可以预防、容忍、消除或预测。通过这些操作，系统对故障具有“弹性”。我们关注的领域包括如何检测系统故障、系统故障发生的频率、故障发生时会发生什么、允许系统停止运行多长时间、故障或失效何时可以安全发生、如何预防故障或失效以及失效发生时需要什么样的通知。

可用性与信息安全性密切相关，但与其有明显不同。拒绝服务攻击被明确设计为使系统失败，即，使其不可用。可用性也与性能密切相关，因为可能很难判断系统何时发生故障，何时只是反应极其缓慢。最后，可用性与安全性密切相关，安全性涉及防止系统进入危险状态，并在进入危险状态时恢复或限制损坏。

构建高可用性容错系统中要求最高的任务之一是了解运行过程中可能出现的失效的性质。一旦理解了这些，就可以在系统中设计缓解策略。

**可用性策略**

当系统不再提供与其规范一致的服务并且系统参与者可以观察到此失效时，就会发生失效。故障（或故障组合）有可能导致失效。反过来，可用性策略旨在使系统能够防止或忍受系统故障，以便系统提供的服务始终符合其规范。 可用性策略具有以下三个目的之一：故障检测、故障恢复或故障预防。这些策略通常由软件基础设施（如中间件包）提供，

**1.故障检测（Detect Faults）**

在任何系统对故障采取措施之前，必须检测或预测故障的存在。此类别中的策略包括：

1. 监控（Monitor）：此组件用于监控系统其他各个部分的运行状况状态：处理器、进程、I/O、内存等。系统监控可以检测网络或其他共享资源中的失效或拥塞，例如拒绝服务攻击。它使用此类别中的其他策略来协调软件，以检测失灵的组件。例如，系统监控可以启动自检（self-tests），或者成为检测错误的时间戳（timestamps）或丢失心跳（heartbeats）的组件。
2. ping/echo：在这种策略中，异步请求/响应消息对在节点之间交换；它用于确定通过关联网络路径的可达性和往返延迟。此外，echo指示 ping 的组件处于活动状态。ping 通常由系统监控发送。Ping/echo需要设置时间阈值；此阈值告诉 ping 组件在认为 ping 组件失败（“超时”）之前等待echo的时间。ping/echo 的标准实现可用于通过互联网协议 （IP） 互连的节点。
3. 心跳（Heartbeat）： 此故障检测机制采用系统监控和被监控进程之间的定期消息交换。检测信号的一个特殊情况是，被监控的进程定期重置其监控中的看门狗计时器，以防止其过期，从而发出故障信号。对于关注可伸缩性的系统，可以通过将检测信号消息搭载到正在交换的其他控制消息上来减少传输和处理开销。检测信号和 ping/echo 之间的区别在于谁负责启动运行状况检查，监控还是组件本身。
4. 时间戳（Timestamp）： 此策略用于检测不正确的事件序列，主要是在分布式消息传递系统中。可以通过在事件发生后立即将本地时钟的状态分配给事件来建立事件的时间戳。序列号也可用于此目的，因为分布式系统中的时间戳在不同处理器之间可能不一致。
5. 状态监控（Condition monitoring）：这种策略涉及检查进程或设备中的条件，或验证设计过程中的假设。通过监控条件，此策略可防止系统产生错误行为。计算校验和是这种策略的一个常见示例。但是，监控本身必须简单（理想情况下，可证明是正确的），以确保它不会引入新的软件错误。

**2.故障恢复（Recover from Faults ）**

从故障中恢复（Recover from faults） 策略被细化为准备和修复策略以及重新引入策略。后者关注的是将故障（但已修复）的组件重新引入正常运行。

1. 冗余备件（Redundant spare）：此策略是指在主组件发生故障时，一个或多个重复组件可以介入并接管工作的配置。此策略是热备盘、温备盘和冷备盘模式的核心，它们的主要区别在于备份组件在接管时的最新程度。
2. 回滚（Rollback）：回滚允许系统在检测到失效时恢复到先前已知的良好状态（称为“回滚线”）— 回滚时间。一旦达到良好状态，就可以继续执行。此策略通常与事务策略和冗余备件策略结合使用，以便在发生回滚后，失败组件的备用版本将提升为活动状态。回滚取决于以前良好状态（检查点）的副本可用于正在回滚的组件。检查点可以存储在固定位置并定期更新，也可以在处理过程中方便或重要的时间（例如在复杂操作完成时）进行更新。
3. 异常处理（Exception handling）：一旦检测到异常，系统将以某种方式处理它。它能做的最简单的事情就是崩溃——但是，当然，从可用性、易用性、可测试性和简单的良好意愿的角度来看，这是一个糟糕的主意。还有更多富有成效的可能性。用于异常处理的机制在很大程度上取决于所使用的编程环境，范围从简单的函数返回代码（错误代码）到使用异常类，这些异常类包含有助于故障关联的信息，例如异常的名称、异常的来源和异常的原因。然后，软件可以使用此信息来屏蔽或修复故障。
4. 软件升级（Software upgrade）：此策略的目标是以不影响服务的方式实现对可执行代码映像的服务中升级。策略包括以下内容：
5. 功能补丁（Function patch）：这种补丁用于过程编程，它使用增量链接器/加载器将更新的软件功能存储到目标内存的预分配段中。新版本的软件功能将使用已弃用功能的入口点和出口点。
6. 类补丁（Class patch）：这种升级适用于执行面向对象代码的目标，其中类定义包括一个后门机制，该机制允许运行时添加成员数据和函数。
7. 无中断服务中软件升级 （Hitless in-service software upgrade，ISSU）。这利用冗余备件策略来实现对软件和相关架构的不影响服务的升级。

**3.故障预防（Prevent Faults）**

与其检测故障然后尝试从中恢复，不如你的系统首先可以防止它们发生呢？虽然听起来好像需要某种程度的千里眼，但事实证明，在许多情况下，可以做到这一点。

1. 从服务中删除（Removal from service）：此策略是指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。例如，系统的某个组件可能会停止服务并重置以清除潜在故障（例如内存泄漏、碎片或未受保护的缓存中的软错误），然后再累积故障达到影响服务的级别，从而导致系统失效。这种策略的其他术语是软件复兴（software rejuvenation）和治疗性重启（therapeutic reboot）。 如果您每晚重新启动计算机，则您正在练习从服务中删除。
2. 事务（Transactions）：面向高可用性服务的系统利用事务语义来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，这些属性统称为“ACID 属性”。交易策略最常见的实现是“两阶段提交”（2PC）协议。此策略可防止由两个进程尝试同时更新同一数据项而导致的竞争条件。
3. 预测模型（Predictive model）：当与监控结合使用时，预测模型用于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。监控的运行指标用于预测故障的发生；示例包括会话建立率（在 HTTP 服务器中）、阈值交叉（监控某些受约束的共享资源的高水位线和低水位线）、进程状态统计信息（例如，服务中、服务中断、维护中、空闲）和消息队列长度统计信息。
4. 异常预防（Exception prevention）：此策略是指用于防止系统异常发生的技术。前面讨论了异常类的使用，它允许系统透明地从系统异常中恢复。异常预防的其他示例包括纠错代码（用于电信）、抽象数据类型（如智能指针）以及使用包装器来防止故障（如悬空指针或信号量访问冲突）。智能指针通过对指针进行边界检查来防止异常，并确保在没有数据引用资源时自动取消分配资源，从而避免资源泄漏。
5. 增加能力集（Increase competence set）： 程序的能力集是它“有能力”运行的状态集。例如，分母为零的状态超出了大多数除法计划的能力集。当一个组件引发异常时，它表明它发现自己超出了其能力范围；从本质上讲，它不知道该怎么做，正在认输。提高组件的能力集意味着将其设计为处理更多情况（故障），作为其正常运行的一部分。例如，假定其有权访问共享资源的组件在发现访问被阻止时可能会引发异常。另一个组件可能只是等待访问或立即返回，并指示它将在下次具有访问权限时自行完成其操作。在此示例中，第二个组件比第一个组件具有更大的能力集。

**第5章 可部署性**

**1.可部署性的定义与重要性**

可部署性是指软件能够在可预测和可接受的时间及工作量内部署到生产环境中，并在新部署不符合规范时能够快速回滚。随着虚拟化和云基础架构的普及，架构师的职责之一是确保部署过程高效、可预测，以降低系统风险。

**2.持续部署与持续交付**

持续部署是代码通过所有自动化测试后自动部署到生产环境的实践，而持续交付则是自动化将代码发布到存储库，然后由运维团队部署到生产环境。持续部署依赖于精心设计的测试自动化，意味着开发人员对应用的更改在编写后几分钟内就能生效，前提是通过了自动化测试。

**3.部署管道的自动化**

部署管道是从代码签入版本控制系统到部署应用程序的一系列工具和活动的自动化序列。这些工具自动测试新提交的代码，测试集成代码的功能，并测试应用程序在负载下的性能、安全性和许可证合规性等问题。自动化不仅加快了部署速度，还减少了人为错误。

**4.容器化、服务网格与云原生架构**

在云原生技术体系下，容器化成为部署应用的首选，Kubernetes是首选的容器编排和调度系统。服务网格从简化服务治理能力的需求中诞生，核心概念是把服务治理能力从开发者的代码中抽象出来，放到一个单独的sidecar代理中实现。这使得开发者只需关注业务代码，而运维人员操作sidecar实现服务治理。

**5.可部署性挑战与机遇**

随着软件系统复杂性增加，可部署性面临更多挑战，如服务和组件分布在不同地理位置和硬件上。架构师必须考虑如何在不影响现有服务的情况下部署新版本，并在出现问题时快速恢复服务。这些挑战带来了对新工具和技术的需求，如容器化、服务网格和云原生架构。

**6.持续部署的实践**

持续部署要求开发团队频繁地将代码集成到共享仓库中，并通过自动化构建和测试来验证。持续部署的实践包括版本控制、自动化构建、自动化测试、部署到测试环境、反馈循环以及部署到生产环境。

**7.DevOps实践**

DevOps是开发和运营的结合，旨在缩短将更改提交到系统与将更改放入正常生产之间的时间，同时确保高质量。实施DevOps是流程改进工作，包含文化和组织元素，以及对工具和架构设计的依赖。

**8.自动化测试**

自动化测试是提高软件质量、降低开发成本的重要手段。它涵盖了软件开发的各个阶段，从单元测试到集成测试。单元测试针对软件中的最小可测试单元进行测试，而集成测试关注模块间的交互和协作。

**9.结论**

可部署性是现代软件开发中的关键属性，它影响着软件的发布频率和效率。通过采用持续部署、DevOps实践和适当的可部署性策略，组织能够提高软件交付的速度和质量，从而在竞争激烈的市场中保持竞争力。架构师必须评估和实施适当的可部署性策略，以确保软件可以高效、可靠地部署和回滚。

**第8章 可修改性**

**1.可修改性的概念与重要性**

可修改性是软件架构中的一个关键质量属性，它涉及到系统对变化的适应能力。在软件系统的生命周期中，变化是不可避免的，它可能包括添加新功能、修复缺陷、增强安全性、提高性能或改善用户体验。可修改性的核心目标是降低这些变更的成本和风险。

**2.可修改性的关键问题**

1. **可以变更什么？** 系统的任何方面都可能变化，包括功能、平台、环境、质量属性和容量。
2. **变更的可能性有多大？** 架构师必须决定哪些变更是可能的，因此哪些将得到支持。
3. **何时进行变更以及谁进行变更？** 变更可以在实现期间、编译期间、生成期间、配置设置期间或执行期间进行，由开发人员、最终用户或系统管理员执行。
4. **变更的成本是多少？** 引入变更机制的成本和使用这些机制进行修改的成本。

**3.可修改性与软件生命周期**

可修改性与软件生命周期紧密相关，因为变更成本通常与变更在生命周期中发生的位置有关。变更越晚，成本越高。因此，架构设计需要考虑如何延迟变更的绑定时间，以减少成本。

**4.可修改性的策略**

可修改性的策略旨在控制变更的复杂性、时间和成本。这些策略包括增加内聚、降低耦合、减少模块大小和延迟绑定时间。

**5.增加内聚**

* **拆分模块**：将包含不紧密职责的模块重构为更紧凑的模块，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类似职责集中在一起，以减少变更的影响范围。

**6.降低耦合**

* **封装**：隔离受审查的功能并为其引入显式接口。
* **使用中介**：防止模块之间耦合过紧。
* **抽象公共服务**：限制模块之间的依赖关系，减少耦合。
* **限制依赖关系**：通过限制模块的可见性和授权来实现。

**7.延迟绑定**

* **参数化**：通过参数化引入灵活性，延迟值的绑定。
* **组件替换**：在生成脚本或文件中进行组件替换。
* **编译时参数化**：在编译时绑定值。
* **切面**：在部署、启动时或初始化时绑定值。
* **配置时绑定**：使用资源文件。
* **运行时绑定**：通过发现、解释参数、共享存储库和多态性。

**8.可修改性模式**

可修改性模式将系统划分为模块，以支持可移植性、可修改性和重用性。以下是一些常用的模式：

**9.客户端-服务器模式**

* **好处**：动态建立连接，客户端之间无耦合，客户端数量可扩展，客户端和服务器可以独立演讲，公共服务可在多个客户端间共享。
* **权衡**：通信可能因网络拥塞而延迟，必须为信息安全性和维护完整性做出特殊规定。

**10.插件（微内核）模式**

* **好处**：提供受控机制扩展核心产品，插件可由不同团队开发，插件可以独立于微内核演讲。
* **权衡**：可能更容易引入安全漏洞和隐私威胁。

**11.分层模式**

* **好处**：可以更改较低层中的软件而不会影响上层，较低层可以在不同应用程序中重用，任何团队都必须了解的接口数量减少。
* **权衡**：如果设计不正确，可能会妨碍为上层提供所需的底层抽象，通常会给系统增加性能损失。

**12.发布-订阅模式**

* **好处**：发布者和订阅者独立且松散耦合，可以轻松更改系统行为，可以轻松记录事件以允许记录和回放。
* **权衡**：可能对性能产生负面影响，组件无法确定接收已发布消息需要多长时间，可能对同步系统的确定性产生负面影响。

**13.可修改性的实际应用**

在实际应用中，可修改性要求架构师在设计阶段就预测未来可能的变化，并为之做好准备。这可能涉及到引入额外的复杂性，但长远来看，这种设计可以显著降低未来的维护成本。例如，通过引入插件架构，系统可以在不停止服务的情况下添加新功能。通过实现服务的发现和绑定，系统可以在运行时动态地适应环境变化。

**14.可修改性与架构决策**

架构决策对可修改性有着深远的影响。例如，选择一个高度耦合的架构可能会在短期内加快开发速度，但会限制未来的变更能力。相反，一个松耦合的架构可能需要更多的初始工作，但它提供了更好的灵活性和可维护性。架构师必须在这些权衡之间做出明智的决策，以满足项目的需求。

**15.结论**

可修改性是软件架构设计中的核心质量属性，它影响着软件系统的长期维护和演化能力。通过采用适当的可修改性策略和模式，架构师可以设计出易于变更和适应新需求的系统。这不仅减少了变更的成本，还提高了系统的灵活性和竞争力。架构师必须在设计时考虑到系统的可修改性，以确保系统能够适应未来的变化，无论是技术进步、市场需求还是业务需求的变化。