**《软件架构实践》读书笔记**

**第一章：软件架构概述**

**1.1 软件架构的定义**

软件架构是系统草图的比喻，它描述了系统的重要结构、行为和属性，以及这些元素之间的相互关系。这个定义强调了架构的两个关键方面：系统的结构和系统的行为。结构指的是系统的组件以及这些组件之间的关系，而行为则描述了系统如何响应外部刺激。属性则涉及到系统的质量，如性能、可靠性、可用性和安全性。

**1.2 架构的重要性**

架构的重要性在于它对系统的质量属性有着决定性的影响。架构决策会直接影响到系统的可维护性、可扩展性、性能和安全性等关键属性。一个好的架构可以使得系统更容易理解和维护，同时也能够更好地适应变化。相反，一个糟糕的架构可能会导致系统难以维护和扩展，甚至可能因为架构的缺陷而导致系统失败。

**1.3 架构视图**

架构视图是描述架构的不同方式，它们提供了对系统的不同视角，以满足不同利益相关者的需求。常见的架构视图包括：

* **逻辑视图**：关注系统的功能性，包括系统的组件和它们之间的交互。
* **开发视图**：关注系统的实现，包括软件的模块化和组件化。
* **物理视图**：关注系统的部署，包括硬件配置和软件在硬件上的分布。
* **进程视图**：关注系统的并发性，包括系统的线程、进程和它们的交互。
* **场景视图**：关注系统的行为，包括系统如何响应特定的用户交互或事件。

**1.4 架构视图的作用**

架构视图的作用在于它们能够帮助不同的利益相关者理解系统的不同方面。例如，开发人员可能更关注开发视图，以了解如何构建和维护系统；而系统管理员可能更关注物理视图，以了解如何在硬件上部署和维护系统。架构视图也有助于沟通架构决策，使得团队成员和利益相关者能够更好地理解架构的设计意图和潜在的权衡。

**1.5 架构框架**

架构框架是一套工具和方法，用于支持架构的分析、设计和记录。常见的架构框架包括：

* **Zachman框架**：提供了一个用于描述系统各个方面的矩阵。
* **C4模型**：专注于软件架构的层次化视图，从系统的整体视图到系统的细节视图。
* **TOGAF**：提供了一套用于企业架构开发的方法和工具。

**1.6 架构框架的作用**

架构框架的作用在于它们提供了一套标准化的方法来分析和设计架构，使得架构师能够更加系统和一致地进行工作。框架还能够帮助架构师识别和记录架构的关键元素，以及这些元素之间的关系。

**1.7 架构与需求**

架构与需求紧密相关，架构设计需要满足系统的需求，同时也需要考虑需求的变化。架构师需要理解系统的需求，包括功能性需求和非功能性需求，并在架构设计中考虑这些需求。需求的变化可能会导致架构的调整，因此架构设计需要具有一定的灵活性和可扩展性。

**1.8 架构与设计**

架构与设计是密切相关的，架构是设计的高层次抽象，而设计则是架构的具体实现。架构提供了设计的指导原则和约束条件，而设计则需要遵循架构的指导原则和约束条件。架构师需要在架构设计和详细设计之间进行平衡，以确保系统的一致性和可维护性。

**1.9 架构与实现**

架构与实现之间的关系是复杂的。架构定义了系统的高层次结构和行为，而实现则是架构的具体化。实现需要遵循架构的指导原则和约束条件，同时也需要考虑技术的选择和实现的效率。架构师需要确保实现与架构保持一致，同时也需要关注实现的质量和性能。

**1.10 架构的演进**

架构不是一成不变的，它需要随着系统的发展和环境的变化而演进。架构的演进可能涉及到架构的调整和优化，以适应新的需求和技术。架构师需要监控架构的演进，并在必要时进行调整，以确保系统的长期成功和可维护性。

**第二章：什么是软件架构**

**2.1 软件架构概念的澄清**

软件架构是某个软件或计算机系统的一个或多个结构，它们由软件元素、这些元素的外部可见属性以及这些元素之间的关系组成。这里的“外部可见属性”指的是其他元素对该元素的假设，例如它所提供的服务、性能特征、错误处理、共享资源的使用等。架构定义了软件元素，并包含了关于各元素应如何彼此相关的信息。架构是系统的抽象，去除了不影响它们如何使用、其他元素如何使用以及如何与其他元素关联或交互的细节。

**2.2 架构的重要性**

软件架构的重要性主要体现在以下几个方面：

1. **抑制或实现系统的质量属性**：架构可以抑制或实现系统的关键质量属性，如性能、可靠性和安全性。
2. **变更的推理和管理**：架构中的决策使你能够在系统演进时对变更进行推理和管理。
3. **质量的早期预测**：对架构的分析能够及早预测系统的质量。
4. **增强利益相关者之间的沟通**：有文档记录的架构可增强利益相关者之间的沟通。
5. **最早的、最根本的设计决策**：架构承载着最早的、因而也是最根本的、最难改变的设计决策。
6. **对后续实现的约束**：架构定义了对后续实现的一组约束。
7. **组织结构的决定因素**：架构决定组织的结构，反之亦然。
8. **增量开发的基础**：架构可以为增量开发提供基础。
9. **成本和进度推理的关键工件**：架构是使架构师和项目经理能够对成本和进度进行推理的关键工件。
10. **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，从而构成产品线的核心。
11. **组件组装的焦点**：基于架构的开发将注意力集中在组件的组装上，而不仅仅是它们的创建上。
12. **引导开发人员的创造力**：通过限制设计选择，架构引导开发人员的创造力，降低设计和系统复杂性。
13. **培训新团队成员的基础**：架构可以成为培训新团队成员的基础。

**2.3 架构作为沟通工具**

架构作为一种沟通工具，可以帮助不同背景和专业知识的利益相关者理解系统的结构和行为。这种沟通是至关重要的，因为它确保了所有涉众对系统有共同的理解，从而可以有效地协商、达成共识或相互沟通。

**2.4 架构与设计选择**

架构限制了设计选择的词汇，通过自愿将我们自己限制在相对较少的元素选择及其交互方式中，可以获得一些好处。这样做最小化了系统的设计复杂性，增强了重用性，使得设计更规则、更简单，更容易理解和沟通，并带来更可靠的可预测结果。

**2.5 架构作为培训基础**

架构，包括元素如何相互作用以执行所需行为的描述，可以作为新的项目成员对系统的首次介绍。这强化了架构作为所有利益相关者的共同参考点的观点，支持和鼓励沟通。

**第三章：质量属性**

**3.1 质量属性的定义**

质量属性是系统的关键特性，它们描述了系统在实际运行和使用中的表现。这些属性不仅包括功能性需求，还包括非功能性需求，如系统的性能、可靠性、可用性、安全性和可维护性等。

**3.2 质量属性的分类**

质量属性可以分为几个类别：

1. **性能属性**：如响应时间、吞吐量和资源利用率。
2. **可靠性属性**：如故障发生的概率、恢复时间和系统的平均无故障时间。
3. **可用性属性**：如系统的可访问性和可恢复性。
4. **安全性属性**：如系统的保密性、完整性和不可抵赖性。
5. **可维护性和可扩展性属性**：如系统的可测试性、可修改性和可扩展性。

**3.3 质量属性的重要性**

质量属性对于软件系统的成功至关重要。它们不仅影响用户的满意度，还影响系统的长期可维护性和业务的竞争力。例如，一个高性能的系统可以处理大量的用户请求，而一个高可用性的系统可以在出现故障时快速恢复服务。

**3.4 质量属性的权衡**

不同的质量属性之间可能存在冲突，架构师需要在这些属性之间做出权衡。例如，提高系统的安全性可能会牺牲一些性能，或者增加系统的复杂性可能会影响可维护性。架构师需要根据项目的具体需求和业务目标来平衡这些权衡。

**3.5 质量属性的度量**

度量是理解和改进质量属性的关键。架构师需要定义和使用适当的度量标准来评估质量属性，这些度量标准可以是定量的，也可以是定性的。例如，响应时间可以通过毫秒来度量，而系统的可用性可以通过百分比来表示。

**3.6 质量属性的分析**

质量属性分析是评估系统是否满足质量属性要求的过程。这通常涉及到使用特定的技术和工具，如模拟、原型和测试。通过质量属性分析，架构师可以识别潜在的问题，并提出改进措施。

**3.7 质量属性的优化**

优化是提高质量属性的过程。这可能涉及到改进系统的设计、选择更好的技术或调整系统配置。优化的目标是在满足业务需求的同时，最大化质量属性。

**3.8 质量属性的文档记录**

记录质量属性和相关的度量标准是沟通和理解系统要求的重要部分。架构文档应该包括质量属性的需求、度量标准和优化策略。

**第四章：架构决策**

**4.1 架构决策的定义**

架构决策是指在软件架构过程中做出的关于系统结构和行为的决策。这些决策通常涉及系统的组织方式、组件的选择、技术的使用以及质量属性的权衡。架构决策不仅影响系统的技术实现，还影响系统的功能和非功能需求。

**4.2 架构决策的重要性**

架构决策至关重要，因为它们为系统的开发和演进设定了基调。它们是系统设计中最根本的决策，通常也是最难改变的。架构决策的质量直接影响到系统的可维护性、可扩展性和性能。

**4.3 架构决策的过程**

架构决策是一个迭代和协作的过程，涉及多个利益相关者。这个过程通常包括以下几个步骤：

1. **识别问题**：确定需要做出架构决策的问题或需求。
2. **探索选项**：研究不同的解决方案和选项。
3. **评估权衡**：评估每个选项的利弊和潜在的权衡。
4. **做出决策**：基于评估结果做出最终决策。
5. **记录决策**：记录决策的原因、选项和结果，以便未来的参考和沟通。

**4.4 架构决策的影响**

架构决策对系统的多个方面有影响，包括：

* **技术选择**：决策会影响技术栈和工具的选择。
* **系统设计**：决策会影响系统的模块化、组件化和服务化。
* **质量属性**：决策会影响系统的性能、安全性和可维护性等质量属性。
* **项目成本和进度**：决策会影响项目的成本和进度。
* **团队结构**：决策可能会影响团队的组织结构和工作流程。

**4.5 架构决策的记录**

记录架构决策是确保项目成功的关键。记录应包括决策的背景、选项、理由、影响和结果。这有助于团队成员理解决策的依据，也便于未来的回顾和分析。

**4.6 架构决策的沟通**

沟通是架构决策过程中的一个重要环节。架构师需要与项目经理、开发人员、测试人员和其他利益相关者沟通决策的内容和影响。有效的沟通可以帮助团队成员理解决策的重要性，并确保决策得到正确实施。

**4.7 架构决策的验证**

验证架构决策是确保决策正确性和有效性的过程。这可能涉及到原型开发、模拟、测试和其他验证活动。验证可以帮助识别潜在的问题，并提供改进决策的机会。

**4.8 架构决策的演进**

随着项目的发展和环境的变化，架构决策可能需要演进和调整。架构师需要监控系统的发展，评估决策的长期影响，并在必要时进行调整。

**第五章：实现质量属性**

**5.1 战术介绍**

战术是指那些能够影响质量属性响应控制的设计决策。它们是实现质量属性的基本设计决策，可以使得一个设计具有可移植性、高性能或可集成性等特性。战术的实施对于满足特定的质量属性需求至关重要。

**5.2 可用性的战术**

可用性的战术主要用于阻止错误发展成为故障，或者至少能够将错误的影响限制在一定范围内，从而使修复变为可能。这些战术包括：

1. **错误检测**：包括命令/响应、心跳、异常等战术。
2. **错误恢复**：包括表决、主动冗余、被动冗余、备件、shadow操作、状态再同步、检查点/回滚等战术。
3. **错误预防**：包括从服务器中删除、事务、进程监控等战术。

**5.3 可修改性的战术**

可修改性的战术目标是控制实现、测试和部署变更的时间和成本。这些战术包括：

1. **局部化修改**：维持语义的唯一性、预期期望的变更、泛化该模块、限制可能的选择。
2. **防止连锁反应**：信息隐藏、维持现有的接口、添加接口、限制通信路径。
3. **推迟绑定时间**：运行时注册、配置文件、多态、组件更换、遵守已定义的协议。

**5.4 确定关键质量与关键功能**

软件需求包括功能需求、质量属性和约束。确定关键质量与关键功能是十分重要的。架构设计之初，如果制定了错误的质量属性目标（包括遗漏重要的质量属性），那么开发的过程可能会很痛苦。在“需求结构化”的基础上，“确定关键质量”着重完成如下两项任务：

1. 根据系统所在领域的特点及系统规模等因素，确定架构设计重点支持哪些质量属性（例如重点支持高性能、可扩展性）。
2. 分析上述质量属性之间的制约关系，第一时间指定权衡折衷的具体策略（例如明确高性能是第一位的，可扩展性与高性能相矛盾时应照顾高性能要求，是否引入支持可扩展性的设计须经架构组评审）。

**5.5 确定关键质量的五大原则**

1. 分类合适+必要扩充。
2. 考虑多方涉众。
3. 检查性思维。
4. 识别矛盾+划定优先级。
5. 严格程度符合领域与规模特点。

**5.6 确定关键功能的规则**

确定关键功能子集的规则包括：

1. 核心功能。
2. 必做功能。
3. 高风险功能。
4. 独特功能（覆盖了上述3类功能没有涉及的职责）。

**第六章：衡量和治理架构特性**

**6.1 架构特性的定义**

架构特性通常指系统的关键质量属性，如性能、可靠性、可用性和可扩展性。这些特性不仅影响系统的操作表现，还与系统的业务目标紧密相关。架构特性的定义需要清晰和具体，以便团队能够围绕架构创建一种无处不在的语言，并鼓励客观定义。

**6.2 架构特性的衡量**

衡量架构特性是确保系统满足业务目标的关键步骤。许多架构特性都有直接的业务衡量标准，如性能或可扩展性。这些衡量标准可以帮助团队理解系统在实际运行中的表现，并根据这些数据做出改进决策。

**6.3 架构特性的治理**

架构治理是确保架构特性得到有效管理和控制的过程。这包括定义架构原则、制定架构决策和监督架构的实施。架构治理有助于确保系统的架构特性与业务目标保持一致，并支持系统的长期成功。

**6.4 架构决策与质量需求的映射**

架构决策可以根据它们支持或阻碍的驱动因素来解释。对于在评估过程中检查的每个质量属性场景，确定并捕获那些有助于实现该场景的架构决策。这些决策可以作为这些决策的理由陈述。

**6.5 敏感点和权衡点的识别**

敏感点是对质量属性响应有显著影响的架构决策。当两个或多个质量属性响应对同一架构决策敏感，但其中一个得到改善而另一个恶化时，就会出现权衡。识别这些敏感点和权衡点对于做出平衡的架构决策至关重要。

**6.6 风险主题的识别**

评估团队在分析完成时检查所有已发现的风险，以寻找总体主题，这些主题可以确定架构甚至架构过程和团队中的系统性弱点。如果不加以处理，这些风险主题将威胁项目的业务目标。

**6.7 架构特性的业务措施**

许多架构特性都有明显的直接衡量标准，如性能或可扩展性。这些架构特性也提供了许多细微的解释，这取决于团队的目标。例如，是测量平均响应时间还是最大响应时间？业务措施有助于将架构特性与具体的业务目标联系起来。

**第七章：软件架构设计**

**7.1 架构设计的重要性**

软件架构设计是降低成本、改进质量、按时和按需交付产品的关键因素。它能够满足系统的品质要求，使受益人达成一致的目标，支持计划编制过程，对系统开发具有指导性，有效管理复杂性，为复用奠定基础，降低维护费用，并支持冲突分析。

**7.2 基于架构的软件开发方法**

基于架构的软件开发方法（ABSD）是由架构驱动的，即由构成架构的商业、质量和功能需求的组合驱动的。ABSD方法允许设计活动从项目总体功能框架明确就开始，这意味着需求抽取和分析还没有完成时，就开始了软件设计。ABSD方法是递归的，且迭代的每一个步骤都是清晰定义的，这有助于降低体系结构设计的随意性。

**7.3 属性驱动的设计（ADD）**

属性驱动的设计（ADD）是一种设计架构的方法，它包括分解、分配、实例化三个步骤。在分解过程中，建立软件必须满足的质量属性，在每个阶段选择战术和架构模式来满足属性场景，然后对功能进行分配，最后实例化。

**7.4 架构视图和质量目标**

架构设计是一个迭代过程，如果要开发的系统能够从已有的系统中导出大部分，则可以使用已有系统的设计过程。不同的视图被用来表达与质量目标有关的信息。

**7.5 体系结构需求**

体系结构需求用来激发和调整设计决策，不同的视图被用来表达与质量目标有关的信息。体系结构需求阶段包括标识构件、架构需求评审等步骤。

**7.6 体系结构设计**

体系结构设计过程包括提出软件体系结构模型、把已标识的构件映射到软件体系结构中、分析构件之间的相互作用、产生软件体系结构、设计评审等步骤。

**7.7 体系结构文档化**

体系结构文档化是将体系结构进行文档化的过程，它是系统设计与开发人员的通信媒介，是为验证体系结构设计和提炼或修改这些设计所执行预先分析的基础。

**7.8 软件架构风格**

软件架构风格是描述某一特定应用领域中系统组织方式的惯用模式。体系结构风格定义一个系统家族，即一个体系结构定义一个词汇表和一组约束。

**7.9 软件架构复用**

软件架构复用可以减少开发工作、减少开发时间以及降低开发成本，提高生产力。不仅如此，它还可以提高产品质量使其具有更好的互操作性。同时，软件架构复用会使产品维护变得更加简单。

**7.10 特定领域软件体系结构（DSSA）**

特定领域软件体系结构的主要目的是在一组相关的应用中共享软件体系结构。DSSA在一个特定应用领域中为一组应用提供组织结构参考的标准软件体系结构。

**第八章：软件架构设计**

**8.1 软件架构设计的定义**

软件架构设计，也称为软件架构，是指刻画软件系统的构成要素及其之间的逻辑关联。它包括软件系统的组织结构、组件及联系（组件间以及组件和环境之间）以及指导设计和系统演进的原则。

**8.2 软件架构设计的三要素**

软件架构设计的三要素包括构件（component）、连接件（connector）和约束（constraint）。构件是软件系统的物理模块，连接件表示构件之间的连接和交互关系，约束则是构件中的元素应满足的条件以及构件组装成更大模块时应满足的条件。

**8.3 架构视图**

软件架构设计通常采用多视图建模，如“4+1”视图模型，包括逻辑视图、开发视图、物理视图和运行视图，以及场景视图。每个视图关注不同的架构方面，如逻辑视图关注功能，物理视图关注部署，开发视图关注模块组织和管理。

**8.4 架构风格**

软件架构风格是描述某一特定应用领域中系统组织方式的惯用模式。常见的架构风格包括层次风格、管道/过滤器风格、信息库风格、C/S风格、MVC风格、SOA风格等。

**8.5 架构设计的内容**

架构设计内容包括确定软件系统的整体架构，包括构件的划分、连接件的选择以及它们之间的组织方式。此外，还包括模块设计、用户界面设计、数据库设计以及数据结构和算法设计。

**8.6 架构设计的重要性**

架构设计是软件开发的蓝图，其重要性体现在确保软件的稳定性、可扩展性和可维护性，促进团队之间的协作和知识共享。一个好的架构能够界定问题与核心生命周期，精准打击业务痛点，同时激发团队潜能，实现利益分配与权责对等。

**8.7 架构演化**

在改变原有结构之前，开发组织必须制订一个周密的体系结构演化计划，作为后续演化开发工作的指南。这包括修改、增加或删除构件，更新构件的相互作用，构件组装与测试，以及技术评审。

**8.8 架构文档化**

架构文档化是将体系结构进行文档化的过程，它是系统设计与开发人员的通信媒介，是为验证体系结构设计和提炼或修改这些设计所执行预先分析的基础。文档的主要输出结果是体系结构规格说明和测试体系结构需求的质量设计说明书。

**第九章：软件架构复用与维护**

**9.1 软件架构复用**

软件架构复用是指在不同软件项目中使用已有的架构元素和设计。这种复用可以显著提高开发效率，减少开发成本，并提升软件质量。软件架构复用可以分为机会复用和系统复用两种类型：

1. **机会复用**：在开发过程中，一旦发现有可复用的资产，就对其进行复用。
2. **系统复用**：在开发之前，就要进行规划，以决定哪些需要复用。

**9.2 软件架构复用的基本过程**

软件架构复用的基本过程包括三个阶段：

1. **构造可复用资产**：首先需要构造恰当的、可复用的资产，并且这些资产必须是可靠的、可被广泛使用的、易于理解和修改的。
2. **管理可复用资产**：构件库（Component Library）是支持软件复用的必要设施，它负责存储和管理可复用构件。
3. **使用可复用资产**：通过获取需求，检索复用资产库，获取可复用资产，并定制这些可复用资产：修改、扩展、配置等，最后将它们组装与集成，形成最终系统。

**9.3 软件架构维护**

软件架构维护是软件开发和维护过程中的一个重点制品，是软件需求和设计、实现之间的桥梁。软件架构的维护与演化密不可分，维护需要对软件架构的演化过程进行追踪和控制，以保障软件架构的演化过程能够满足需求。

**9.4 软件架构知识管理**

软件架构知识管理侧重于软件开发和实现过程所涉及的架构静态演化，在架构文档等信息来源中捕捉架构知识，提供架构的质量属性及其设计依据进行记录和评价。

**9.5 构件及其复用**

构件是指软件系统中可以明确辨识的构成成分。而可复用构件是指具有相对独立的功能和可复用价值的构件。构件是一个组成单元，具有约定规范的接口及明确的依赖环境。

**9.6 商用构件标准规范**

商用构件标准规范包括CORBA、J2EE和DNA 2000等，它们定义了构件的交互和集成方式。

**9.7 应用系统簇与构件系统**

构件系统只在开发单位内部使用，而应用系统提供给外部客户。与应用系统相比，构件系统具有通用性、可复用性。一个构件系统是能提供一系列可复用特性的系统产品。

**9.8 基于复用开发的组织结构**

与传统的开发组织结构不同，它需要有一部分用于开发可复用资产的资源，这部分资源应同具体引用系统的开发资源分开，以确保不被占用。

**9.9 产品线与系统演化**

实质上是用架构技术构建产品线，并在此基础上借用复用技术持续演化，不断的推出新产品，满足市场追求产品升级换代的需求。

**第十章：软件架构与信息安全性**

**10.1 信息安全与信息系统安全**

信息安全是一个广泛和抽象的概念，涵盖了人工和自动信息处理的安全、网络化和非网络化的信息安全。信息系统安全则更具体，是确保信息系统结构安全、与信息系统相关的元素安全、以及与此相关的各种安全技术、安全服务和安全管理的总和。

**10.2 信息系统安全管理**

安全管理是理解应该保护什么和为什么需要保护这两个概念之间的桥梁。它包括计划、组织、指挥、协调和控制等活动，以确保信息安全。

**10.3 信息安全技术基础知识**

信息安全的五个基本要素包括机密性、完整性、可用性、可控性和可审查性。网络安全漏洞、安全威胁和安全措施也是本节讨论的重点。

**10.4 信息加解密技术**

本节介绍了对称加密算法和非对称加密算法，包括DES、3DES、IDEA、RSA、Elgamal和ECC等。

**10.5 数字签名技术**

数字签名技术涉及信息摘要和单向散列函数，用于确保信息完整性和防篡改。常用的消息摘要算法有MD5、SHA等。

**10.6 计算机信息系统安全保护等级**

信息系统安全保护等级从用户自主保护级到访问验证保护级，涉及不同层次的安全需求和保护措施。

**10.7 应用程序安全加固**

第十章还讨论了应用程序安全加固，包括信息安全技术体系与威胁模型、操作系统安全加固、运行环境安全加固等内容。

**10.8 软件工程与信息安全**

第十章还涉及软件工程的概述，包括软件开发生命周期、系统软件文档、PDCA模型、软件系统工具和软件设计活动。

**10.9 逆向工程**

逆向工程是分析程序，力图在比源码更高成次抽象层次上建立程序的表示过程，是设计的恢复过程。

**第十一章：信息安全性**

**11.1 信息安全性的通用场景**

本节描述了信息安全性的通用场景，包括保密性、完整性和可用性。系统需要确保能够维护这些安全属性，保护数据或服务免受未经授权的访问，确保数据不被未经授权的操作，以及确保处理各方得到保证和参与。

**11.2 隐私**

与安全密切相关的问题是隐私。隐私问题近年来变得越来越重要，欧盟通过《通用数据保护条例》（GDPR）将其纳入法律。隐私的实现是关于限制对信息的访问，特别是个人身份信息（PII）。架构师可能需要验证PII是否对不需要访问PII的开发团队成员隐藏。

**11.3 抵抗攻击**

本节讨论了多种抵抗攻击的方法，包括识别参与者、分离实体、验证输入和更改凭据设置。这些方法有助于限制攻击的范围和影响，确保系统的安全性。

**11.4 对攻击做出反应**

对于潜在的攻击，系统需要有应对策略，包括撤销访问权限、限制登录和通知参与者。这些策略有助于减轻攻击的影响，并确保系统的快速恢复。

**11.5 从攻击中恢复**

一旦系统检测到并尝试抵抗攻击，它就需要恢复。恢复的一部分是恢复服务，可能包括保留其他服务器或网络连接。审计和不可否认性策略也在此阶段发挥作用，帮助追踪攻击者的行为并识别攻击者。

**11.6 信息安全的模式**

本节介绍了两种广为人知的安全模式：拦截验证器和入侵防御系统。拦截验证器在消息的源和目标之间插入一个软件元素，而入侵防御系统（IPS）是一个独立组件，其主要目的是识别和分析任何可疑活动。

**11.7 扩展阅读**

本节提供了关于信息安全性的扩展阅读建议，包括编码、组织流程和技术流程。这些资源有助于更深入地理解信息安全性的各个方面。

**第十二章：软件架构复用与维护**

**12.1 软件架构复用**

软件架构复用是指在不同的软件项目中使用已有的架构元素和设计。这种复用可以显著提高开发效率，减少开发成本，并提升软件质量。软件架构复用可以分为机会复用和系统复用两种类型：

1. **机会复用**：在开发过程中，一旦发现有可复用的资产，就对其进行复用。
2. **系统复用**：在开发之前，就要进行规划，以决定哪些需要复用。

软件架构复用的原因包括减少开发工作、减少开发时间以及降低开发成本，提高生产力、提高产品质量使其具有更好的互操作性，使产品维护变得更加简单。

软件架构复用的对象及形式非常广泛，包括需求、架构设计、元素、建模与分析、测试、项目规划、过程、方法和工具、人员、样本系统以及缺陷消除。

**12.2 软件架构复用的基本过程**

软件架构复用的基本过程包括三个阶段：

1. **获得可复用的软件资产**：首先需要构造恰当的、可复用的资产，并且这些资产必须是可靠的、可被广泛使用的、易于理解和修改的。
2. **管理可复用资产**：该阶段最重要的是构件库（Component Library），由于对可复用构件进行存储和管理，它是支持软件复用的必要设施。
3. **使用可复用资产**：通过获取需求，检索复用资产库，获取可复用资产，并定制这些可复用资产：修改、扩展、配置等，最后将它们组装与集成，形成最终系统。

**12.3 软件架构维护**

软件架构维护是软件开发和维护过程中的一个重点制品，是软件需求和设计、实现之间的桥梁。软件架构的维护与演化密不可分，维护需要对软件架构的演化过程进行追踪和控制，以保障软件架构的演化过程能够满足需求。

软件架构维护过程包括架构知识管理、架构修改管理和架构版本管理等内容：

1. **架构知识管理**：侧重于软件开发和实现过程所涉及的架构静态演化，在架构文档等信息来源中捕捉架构知识，提供架构的质量属性及其设计依据进行记录和评价。
2. **架构修改管理**：主要是建立一个隔离区域，保障该区域中任何修改对其他部分影响最小。
3. **架构版本管理**：涉及软件架构随时间变化的版本控制和管理。

**第十三章：软件架构复用与维护**

**13.1 软件架构复用**

软件架构复用是指在开发新的软件系统时，重复利用已有的、经过验证的软件架构，以提高开发效率、降低风险、保持一致性，并促进软件系统的模块化和标准化。软件架构复用是软件工程领域的一项重要实践，有助于实现快速开发、节约成本、提升软件质量和可维护性。

**13.1.1 架构组件复用**

软件架构通常由一系列抽象层次较高的组件构成，如模块、子系统、框架、设计模式等。这些组件可以是具体的代码实现，也可以是设计规范、接口定义、配置文件等非执行形式。复用这些组件意味着在新系统开发时直接采用或适当修改已有的架构组件，而非从零开始设计。

**13.1.2 架构风格与原则复用**

除了具体的架构组件，软件架构的风格、原则、最佳实践等也可以在新项目中复用。这些高级指导原则有助于构建出结构清晰、易于维护、符合业界标准的软件系统。

**13.1.3 促进软件复用的措施**

为了有效进行软件架构复用，需要采取以下措施：

* 建立企业级架构资产库：收集、整理并维护企业的架构组件、框架、设计模式、接口规范、最佳实践等资源，形成可供开发团队查阅、选用的资产库。
* 强化知识共享与培训：定期开展内部技术分享会、工作坊等活动，提升团队对现有架构资产的理解和应用能力。提供必要的培训，使开发人员熟悉复用的原则、方法和工具。
* 实施架构评审：在项目启动阶段及关键里程碑点进行架构评审，确保新系统的设计遵循已有的架构原则和最佳实践，合理利用现有架构资产。
* 持续优化与更新架构资产：随着技术的发展和业务的变化，及时更新和优化架构资产库，淘汰过时的组件，引入新的框架和技术，保持资产的先进性和适用性。

**13.2 软件架构维护**

软件架构维护是软件开发和维护过程中的一个重点制品，是软件需求和设计、实现之间的桥梁。软件架构的维护与演化密不可分，维护需要对软件架构的演化过程进行追踪和控制，以保障软件架构的演化过程能够满足需求。

**13.2.1 软件架构知识管理**

软件架构知识管理侧重于软件开发和实现过程所涉及的架构静态演化，在架构文档等信息来源中捕捉架构知识，提供架构的质量属性及其设计依据进行记录和评价。

**13.2.2 构件及其复用**

构件是指软件系统中可以明确辨识的构成成分。而可复用构件是指具有相对独立的功能和可复用价值的构件。构件是一个组成单元，具有约定规范的接口及明确的依赖环境。

**13.2.3 商用构件标准规范**

商用构件标准规范包括CORBA、J2EE和DNA 2000等，它们定义了构件的交互和集成方式。

**13.2.4 应用系统簇与构件系统**

构件系统只在开发单位内部使用，而应用系统提供给外部客户。与应用系统相比，构件系统具有通用性、可复用性。一个构件系统是能提供一系列可复用特性的系统产品。

**13.2.5 基于复用开发的组织结构**

与传统的开发组织结构不同，它需要有一部分用于开发可复用资产的资源，这部分资源应同具体引用系统的开发资源分开，以确保不被占用。

**13.2.6 产品线与系统演化**

实质上是用架构技术构建产品线，并在此基础上借用复用技术持续演化，不断的推出新产品，满足市场追求产品升级换代的需求。