



NEW PLAN

软件高级架构师

• 一段新征程 •





01 架构基础知识 (构件)

02 架构风格 (架构风格的表+层次架构风格+面向服务架构风格)

03 特定领域软件架构 (DSSA)

04 基于架构的软件开发 (ABSD)

05 质量属性

06 架构评估



01

架构设计基础知识



- 本章节和下一章的质量属性架构评估一起是系统架构这门课的重点中的重点，选择题就占据了**20-25分**，案例分析题和论文都有涉及到，所以本章节全部内容都是重点，除去老师强调的每年必考的题之外，其他的地方在有时间空余的情况下，尽可能全面的过一遍，有条件去看书是最好的。
- 被考到知识点有：
 - 软件架构概念，软件架构设计与生命周期、基于架构的软件开发方法；基于架构的软件设计**ABSD**
 - 软件架构风格：数据流、调用/返回、以数据为中心、虚拟机、独立构件
 - 特定领域软件架构**DSSA**
 - 软件质量属性、敏感点、风险点
 - 系统架构评估：架构权衡分析**ATAM**、基于场景的架构分析方法**SAAM**、成本效益分析**CBAM**
- 在改版之后的考试中，分别考察的知识点为：
 - **2023年11月**：机会复用和系统复用、静态架构评估、质量属性、架构定义、质量属性效用树的结构、**DSSA**、**ABSD**、质量属性场景刺激和度量、批处理管理过滤器风格对比、构件接口、构件没有外部可见状态、适应性和装配性构件、构件检索、构件管理步骤、构件可组装性和可部署性、层次式架构。
 - **2024年05月**：架构演化和构件、黑板风格、管道过滤器风格、层次架构风格、架构风格判断、事件驱动风格、物联网三层、**SOA**的**UDDI**协议、构件、**DSSA**领域模型、构件组装、质量属性、六要素之响应、效用树、**SAAM**、**ATAM**、性能参数和设计策略、可靠性**MTTD**、健壮性、架构定义（视角与视图）、**ADL**
 - **2024年11月**：架构**4+1**视图、**ATAM**、效用树、架构风格、敏感点和权衡点、**ABSD**、**EAI**的集成顺序、**EAI**、架构评估、架构演化、**DSSA**、质量属性、架构复用
 - **2025年05月**：层次架构、架构风格（4分）、软件质量、**WebService**、质量属性（3分）--->总计考了10分，下篇八大架构也考了3分，分别是**SOA**、**ESB**和安全架构
 - **2025年11月**：质量属性、权衡点、**DSSA**、**MVP**模型、**ABSD**、**C/S**架构、进程通信、规则系统(架构风格)、可移植性、**CBAM**、微服务、**Hofmeister**视图(架构)、头脑风暴质量属性(质量属性)、解释器、质量属性效用树--->总计考了16分，其中4分考的书本上很细的点，下篇八大架构考了2分（大数据架构和云原生）
 - **2026年05月**：运行期质量属性、架构风格（5分）、**ABSD**（2分）、**CS**架构、**DSSA**（2分）、架构评估方法（3分）、质量属性（4）、**lambda**（2分）、访问控制矩阵 **ACM**（安全架构）、微服务--->总计考了22分，3分是下篇八大架构



软件架构的定义：软件架构(Software Architecture)或称软件体系结构，是指系统的一个或者多个结构，**这些结构包括软件的构件（可能是程序模块、类或者是中间件）、构件的外部可见属性及其之间的相互关系。**体系结构的设计包括数据库设计和软件结构设计，后者主要关注软件构件的结构、属性和交互作用，并通过多种视图全面描述。

简单理解软件架构：软件架构指从需求分析到软件设计之间的过渡过程。**只要软件架构设计好了，整个软件就不会出现坍塌性的错误，即不会崩溃。**

软件架构的详细解释：软件架构为软件系统提供了一个结构、行为和属性的高级抽象，由**构件的描述、构件的相互作用(连接件)、指导构件集成的模式以及这些模式的约束组成。**

总结：在软件中，架构决策包括如何组织代码、模块和组件，如何处理数据流、用户界面和业务逻辑。好的软件架构能够确保软件具有良好的性能、可扩展性、可维护性和安全性，就像一个精心设计的大楼能够提供舒适和便捷的居住环境一样。所以，**软件架构就是软件的总体设计方案，它决定了软件如何组织和工作，规定了软件系统的整体结构和各个部分之间的关系，以满足用户需求和业务目标。好的架构是构建可靠软件的基础。**

软件架构就是：规划一座数字城市的功能区（构件），设计每个区对外的服务窗口（接口），铺设连接全城的基础设施网络（关系），并用多套城市蓝图（视图）让决策者、建设者、运营者都能看懂——最终让这座城市能容纳百万人口、应对突发灾害、持续迭代升级而不崩盘。

软件架构要素	城市类比	核心含义
软件构件	软件构件有支付模块、用户模块、订单模块，而城市里面也有对应的比如公安局、港口、图书馆、高铁站	每个构件都是一个复杂的子系统内部自成体系，对外提供特定服务。
外部可见属性	金融街对外提供：存款、贷款、转账服务（你不需要知道金库在哪、风控系统怎么跑） 港口对外提供：货物装卸、报关、航运对接（你不需要知道起重机型号、调度算法）	只暴露必要的功能接口，隐藏内部实现细节
相互关系	系统的网关、API总线就相当于城市中的地铁、公交网络，系统中的消息队列就相当于城市中的公路、输电网	构件之间的连接方式、通信机制、依赖管理 关键设计：老城区（遗留系统）和新城区（微服务）怎么打通，高峰期（双11）怎么限流、分流？ 某个区停电（服务宕机）怎么保证其他区正常运转？
多种视图	城市里面有总体规划图、交通规划图、应急防灾图等，对应系统中也有逻辑视图、进程视图	针对不同干系人（开发、运维、管理、用户）展示不同维度的设计



1. 软件架构设计贯穿于软件开发生命周期的各个阶段。软件架构设计并非只在开发初期进行，而是贯穿于软件开发生命周期的各个阶段，包括需求分析、设计、实现、测试、部署和维护等。
2. 需求分析阶段。需求分析和SA设计面临的是不同的对象：一个是问题空间；另一个是解空间。从软件需求模型向SA模型的转换主要关注两个问题：如何根据需求模型构建SA模型。如何保证模型转换的可追踪性。简单来说，该阶段关注的是如何将用户需求转换为软件架构模型，并确保模型的可追踪性。
3. 设计阶段。是SA研究关注的最早和最多的阶段，这一阶段的SA研究主要包括：SA模型的描述、SA模型的设计与分析方法，以及对SA设计经验的总结与复用等。有关SA模型描述的研究分为3个层次：SA的基本概念(构件和连接件)、体系结构描述语言ADL、SA模型的多视图表示。简单来说，该阶段关注的是软件架构模型的描述、设计与分析方法，以及设计经验的总结与复用。
4. 实现阶段。最初SA研究往往只关注较高层次的系统设计、描述和验证。为了有效实现SA设计向实现的转换，实现阶段的体系结构研究表现在对开发过程的支持、开发语言和构件的选择以及相关测试技术。简单来说，该阶段关注的是如何将软件架构设计转换为代码，以及如何进行测试。

5. 构件组装阶段。在SA设计模型的指导下，可复用构件的组装可以在较高层次上实现系统，并能够提高系统实现的效率。在构件组装的过程中，SA设计模型起到了系统蓝图的作用。简单来说，该阶段关注的是如何在架构设计模型的指导下，进行可复用构件的组装，提高系统实现效率，并解决组装过程中的相关问题。
6. 部署阶段。提供高层的体系结构视图来描述部署阶段的软硬件模型，以及基于SA模型可以分析部署方案的质量属性，从而选择合理的部署方案。简单来说，该阶段关注的是如何根据软件架构模型进行部署，并分析部署方案的质量属性。
7. 后开发阶段。是指软件部署安装之后的阶段。这一阶段的SA研究主要围绕维护、演化、复用等方面来进行。典型的研究方向包括动态软件体系结构、体系结构恢复与重建等。简单来说，该阶段关注的是如何根据软件架构模型进行维护和演化

阶段	作用和意义
需求分析阶段	有利于各阶段参与者的交流，也易于维护各阶段的可追踪性
设计阶段	关注的最早和最多的阶段
实现阶段	有效实现从软件架构设计向实现的转换
构件组装阶段	可复用构件组装的设计能够提高系统实现的效率
部署阶段	组织和展示部署阶段的软硬件架构、评估分析部署方案
后开发阶段	主要围绕维护、演化、复用进行

架构阶段	城市建设阶段	核心任务	具体类比
贯穿始终	城市规划贯穿项目全周期	架构思维不丢	从立项到城市衰老，总规师团队全程参与，不是画完图就撤
需求分析	城市立项调研	问题空间→解空间	市民诉求→规划方案，调研市民要什么（通勤？就业？环境？）→确定建"金融中心"还是"宜居新城"，需求可追溯（为什么建这个区）
设计阶段	总体规划编制	描述、分析、复用	画总规图（多视图：用地、交通、市政）、用规划术语（ADL）、借鉴其他城市经验（设计模式复用）
实现阶段	施工建设	蓝图→实体	选建材（开发语言）、按图施工（编码）、工程监理（测试验收）
构件组装	模块化建造	复用组装	用预制构件（装配式建筑）、标准接口（门窗规格）、快速搭楼（提高效率）
部署阶段	配套设施落地	软硬件部署	水电网接入（服务器上架）、交通压力测试（负载测试）、选最优方案（A/B部署对比）
后开发阶段	城市运营治理	维护、演化、复用	老旧小区改造（架构重构）、城市大脑升级（动态架构）、从废墟考古复原规划（架构恢复）

在架构设计中，**构件(Component)**是指系统的重要部分，它们是软件系统中可独立部署、可复用、具有明确接口和功能的模块化单元，可能是一个类、库、服务等。构件通常用来划分系统的不同功能或责任，以便更容易管理、维护和扩展整个系统。我们可以将构件视为乐高积木——每个积木（构件）有标准接口（凹凸结构），可通过组装（接口适配）构建复杂系统（模型）。

构件的核心特征：

特征	说明
可复用性	构件设计时需解耦，不依赖具体业务场景，可在不同系统中重复使用。
标准化接口	提供明确的输入/输出接口（API或协议），隐藏内部实现细节（黑盒化）。
独立性	可独立开发、测试、部署，通过接口与其他构件交互。
可组装性	支持通过配置或代码调用与其他构件组合，形成完整系统功能。

构件与对象的区别：

特性	构件	对象
部署单元	独立部署单元	实例单元
状态可见性	无外部可见状态	可能具有外部可见状态
复用粒度	粗粒度（功能模块）	细粒度（类/方法）

构件的组装方式:

技术类型	特点	适用场景
基于功能组装	通过子程序调用实现功能模块集成	传统分层架构（如MVC）
基于数据组装	围绕核心数据结构构建框架（如Jackson方法）	数据密集型系统（ETL流程）
面向对象组装	通过继承/多态实现复用（如Java类库）	OOP系统（Spring框架）

组装过程中的关键点:

- 接口匹配: 确保构件接口的输入/输出兼容性（如协议、数据格式）
- 语境适配: 调整构件对运行环境的依赖（如数据库连接配置）
- 异常处理: 解决组装失配问题（如通信协议不一致、数据模型冲突）

构件组装是指构件相互直接集成或是用专门编写的‘胶水代码’将它们整合在一起来创建一个系统或另一个构件的过程。常见的组装构件有以下**3种**组装方式。

- **顺序组装**。通过按顺序调用已经存在的构件，可以用两个已经存在的构件来创建一个新的构件。如上一个构件输出作为下一个构件的输入。类似于接力赛（一棒接一棒）
- **层次组装**。这种情况发生在一个构件直接调用自另一个构件所提供的服务时。被调用的构件为调用的构件提供所需的服务。二者之间接口匹配兼容。类似于上下级（领导派活给下属）
- **叠加组装**。这种情况发生在两个或两个以上构件放在一起创建一个新构件的时候。这个新构件合并了原构件的功能，从而对外提供了新的接口。外部应用可以通过新接口来调用原有构件的接口，而原有构件不互相依赖，也不互相调用。这种组装类型适合于构件是程序单元或者构件是服务的情况。类似于拼盘菜（各菜独立摆盘，共用一个大盘子端上桌）

构件组装的三种不兼容问题(通过编写适配器解决):

- 参数不兼容。接口每一侧的操作有相同的名字，但参数类型或参数个数不相同。
- 操作不兼容。提供接口和请求接口的操作名不同。
- 操作不完备。一个构件的提供接口是另一个构件请求接口的一个子集，或者相反。



国际上常用的**构件标准**主要有**三大流派**。

- EJB(Enterprise Java Bean)规范由Sun公司制定，有三种类型的EJB：
 - **会话Bean(Session Bean)**: 用于管理会话和业务逻辑，有不同的类型适应不同的需求
 - **实体Bean(Entity Bean)**: 用于与持久化数据交互，将对象映射到数据库表。
 - **消息驱动Bean(Message-driven Bean)**: 用于异步消息处理，响应来自消息队列的消息
- COM、DCOM、COM+: COM是微软公司的。DCOM是COM的进一步扩展，具有位置独立性和语言无关性。COM+并不是COM的新版本，是COM的新发展或是更高层次的应用
- CORBA
 - CORBA标准主要分为三个层次: **对象请求代理、公共对象服务和公共设施**
 - CORBA (Common Object Request Broker Architecture, 公共对象请求代理体系结构) 是由 OMG (Object Management Group, 对象管理组织) 制定的一套分布式计算标准, 核心目标是解决不同平台、不同编程语言开发的软件组件之间的“互操作问题”——简单说, 就是让用 Java 写的程序能调用 C++ 写的程序, 让 Windows 上的组件能和 Linux 上的组件协作, 而不用关心对方的“出身”(位置、语言、操作系统等)。
 - CORBA 本质是一套“跨平台、跨语言的分布式对象互操作标准”, 通过 ORB (软总线)、公共对象服务 (通用工具)、公共设施 (业务框架) 三层结构, 让不同技术栈的软件组件能像“本地对象”一样协作, 解决了异构系统的“沟通难题”。虽然随着 Web 服务 (如 SOAP、REST) 的兴起, CORBA 的应用场景越来越少了。

对比维度	CORBA	DCOM	EJB
标准化组织	OMG（开放联盟）	Microsoft（专有）	Sun/Oracle（专有）
平台支持	跨平台（Unix/Linux/Windows）	主要是Windows	所有支持Java的平台
语言支持	多语言（C++、Java、Python等）	多语言（C++、VB、Java等）	仅Java
核心协议	IIOP	ORPC（基于DCE/RPC）	JRMP（Java远程消息协议）
市场定位	大型企业异构集成	Windows企业应用	Java EE企业应用
当前状态	legacy系统维护	逐步被.NET取代	被Spring等轻量级框架替代



构件

1. 微服务架构标准

核心规范：RESTful接口描述标准（替代CORBA IDL）

技术栈：Spring Cloud Alibaba、Quarkus、Micronaut

2. 云原生构件标准

容器镜像标准（Docker/Containerd兼容）

运行时规范（Kubernetes CRI标准）

3. 跨平台互操作标准

GraphQL：替代传统SOAP/WSDL，实现前后端解耦

Apache Arrow：内存数据交换格式（跨语言/组件高效通信）

传统标准	现代替代方案
CORBA/DCOM/ RMI	RESTful API、gRPC、消息 队列（Kafka/RabbitMQ）
EJB	Spring Boot、微服务架构
重量级应用服 务器	容器化 （Docker/Kubernetes）、 Serverless

软件架构风格的概念：是描述某一特定应用领域中系统组织方式的**惯用模式**。架构风格定义一个系统家族，即一个**架构定义**、一个**词汇表**和一组**约束**。架构定义描述了系统的整体结构和组织方式。词汇表中包含一些构件和连接件类型，而这组约束指出系统是如何将这些构件和连接件组合起来的。简单来说，**软件体系结构风格就是一个模板**，它规定了特定应用领域中软件系统应该如何构建。

软件架构风格的作用：反映了领域中众多系统所共有的**结构和语义特性**，并指导如何将各个模块和子系统**有效地组织成一个完整的系统**。对软件架构风格的研究和实践促进对设计的重用，一些经过实践证实的解决方案也可以可靠地用于解决新的问题。

架构设计的一个核心问题是能否达到架构级的软件复用，强调对架构设计的重用。

架构风格定义了用于描述系统的术语表和一组指导构建系统的规则。

简单来说，如果说架构设计是如何规划一座城市的话，那么架构风格是选择建城的具体方式——是像古代城池一样严格分层？像现代联邦一样服务自治？还是像物流网络一样事件驱动？不同流派没有绝对好坏，只有适不适合当前业务规模、团队能力和质量需求。

- 数据流风格：面向数据流，按照一定的顺序从前向后执行程序，代表的风格有批处理序列、管道-过滤器。
- 调用/返回风格：构件之间存在互相调用的关系，一般是显式的调用，代表的风格有主程序/子程序、面向对象、层次结构、客户端服务器。
- 独立构件风格：构件之间是互相独立的，不存在显式的调用关系，而是通过某个事件触发、异步的方式来执行，代表的风格有进程通信、事件驱动系统(隐式调用)。
- 虚拟机风格：自定义了一套规则供使用者使用，使用者基于这个规则来开发构件，能够跨平台适配，代表的风格有解释器、基于规则的系统。
- 以数据为中心风格(数据共享风格、仓库风格)：以数据为中心，所有的操作都是围绕建立的数据中心进行的，代表的风格有数据库系统、仓库系统、黑板系统。



- 批处理序列：构件为一系列固定顺序的计算单元，多件事情同步顺序执行，构件之间只通过数据传递交互。
每个处理步骤是一个独立的程序，每一步必须在其前一步结束后才能开始，数据必须是完整的，以整体的方式传递。比如批量图像处理：一次性处理大量图像，例如调整大小、添加水印或转换格式。
- 管道-过滤器：每个构件都有一组输入和输出，构件读取输入的数据流，经过内部处理，产生输出数据流。
前一个构件的输出作为后一个构件的输入，前后数据流关联。过滤器就是构件，连接件就是管道。比如文本处理管道：在文本分析中，可以将文本处理划分为分词、词干提取、情感分析等多个阶段，每个阶段是一个过滤器。

早期编译器就是采用的这种架构，要一步一步处理的，均可考虑此架构风格。

二者区别：批处理风格通常一次性处理大量数据，离线执行，适用于批量处理任务。管道过滤器风格将任务分解为多个阶段，每个阶段逐个处理数据，通常是实时或近实时执行，适用于可组合和可扩展的任务。



- 主程序/子程序：多线程控制，把问题划分为若干个处理步骤，构件即为主程序和子程序，子程序通常可合成为模块。过程调用作为交互机制，充当连接件的角色。
- 面向对象：对象是抽象数据类型的实例。连接件即使对象间交互的方式，对象是通过函数和过程的调用来交互的。
- 层次结构：构件组成一个层次结构，连接件通过决定层间如何交互的协议来定义。每层为上一层提供服务，使用下一层的服务，只能见到与自己邻接的层。修改某一层，最多影响其相邻的两层(通常只能影响上层)。优点是可以将一个复杂问题分解成一个增量步骤序列的实现。缺点是并不是每个系统都可以很容易的划分为分层的模式，并且因为进行层次调用，会影响效率。
- 进程通信：构件是独立的**进程**，连接件是**消息传递**。构件通常是命名过程，消息传递的方式可以是点对点、异步或同步方式，以及远程过程(方法)调用等。进程通信涉及不同的进程或线程之间的通信和数据共享。这些进程可以运行在同一台计算机上，也可以分布在不同的计算机上。
 - 举例：一个典型的示例是一个多线程的文件下载器，其中一个线程负责下载文件，另一个线程负责监视下载进度。这两个线程需要通过进程通信来共享下载状态信息，以便监视线程可以显示下载进度。



事件驱动系统(隐式调用): 构件不直接调用一个过程, 而是触发或广播一个或多个事件。构件中的过程在一个或多个事件中注册, 当某个事件被触发时, 系统自动调用在这个事件中注册的所有过程。

- 举例: 一个典型的示例是图形用户界面(GUI)应用程序。用户的操作(如点击按钮、键盘输入)会生成事件, 应用程序的控件或组件会注册事件处理程序来响应这些事件。

主要优点是软件复用提供了强大的支持, 为构件的维护和演化带来了方便; 缺点是构件放弃了对系统计算的控制, 只能被动的控制。



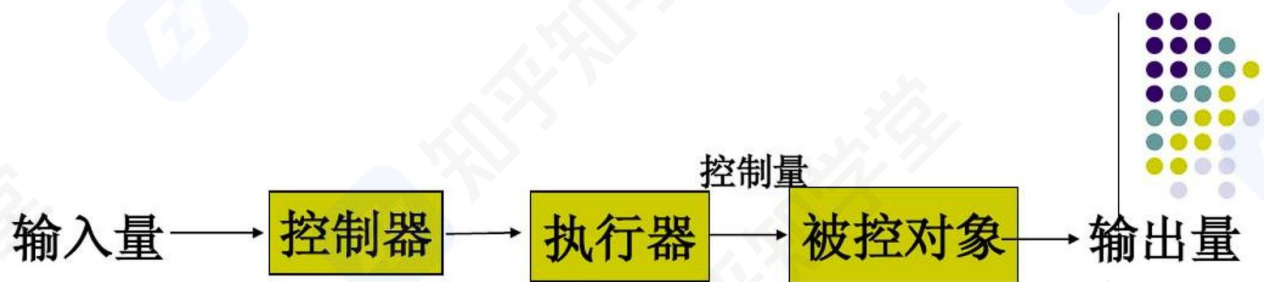
- 解释器：通常包括一个完成解释工作的解释引擎、一个包含将被解释的代码的存储区、一个记录解释引擎当前工作状态的数据结构，以及一个记录源代码被解释执行的进度的数据结构。涉及解析和执行一系列指令或命令，通常通过解释器来实现。解释器将文本或代码解析成可执行的操作。缺点是执行效率低。
 - 举例：编程语言的解释器是最常见的示例。例如，Python解释器会解释和执行Python编写的代码。用户输入一个数学表达式，解释器会将其解析并执行计算，然后返回结果。例如，用户输入"2 + 3"，解释器会计算并返回"5"。
- 基于规则的系统：包括规则集、规则解释器、规则/数据选择器和工作内存，一般用在人工智能领域和DSS(决策支持系统)中。适用于根据一组事先定义的规则或条件来控制系统的行为。这种风格通常用于实现灵活的业务规则和决策逻辑。
 - 举例：银行的贷款审批系统。系统使用基于规则的风格，根据客户的信用评分、贷款金额和其他因素来应用一组贷款批准规则。如果满足规则的条件，系统将自动批准或拒绝贷款申请。



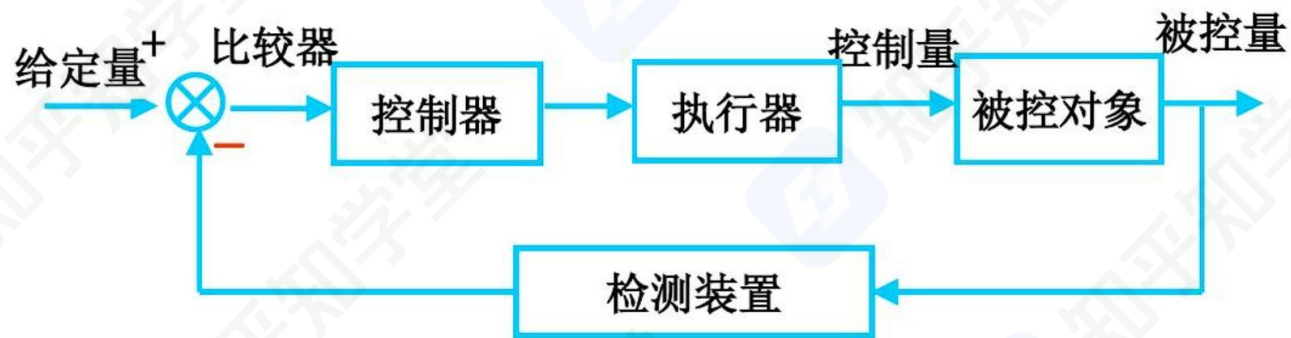
- 仓库风格的架构将数据存储在一个中央仓库或数据库中，各个组件可以从仓库中读取和写入数据。组件之间通过共享数据仓库进行通信和协作。
 - 示例说明：假设一个天气预报系统具有多个数据源，包括气象站、卫星数据和气象传感器。这些数据源将数据写入共享的中央仓库，然后天气预报应用程序可以定期查询仓库以获取最新的气象信息并生成天气预报。
- 黑板风格的架构类似于一个黑板或公告板，多个独立的组件称为"专家"共享一个公共存储区(黑板)，它们可以读取和写入数据。专家根据黑板上的信息进行推断和决策，适用于解决复杂的非结构化问题。
 - 示例说明：假设一个医学诊断系统包括放射科医生、心脏专家和内科医生。每个专家可以根据患者的病历信息(例如X光片、心电图和实验室报告)向黑板提交自己的诊断。黑板负责集成各个专家的诊断，并生成最终的诊断结果。

总之，**仓库风格强调数据的集中存储和共享，组件通过访问共享的仓库来交互**。而黑板风格侧重于多个独立的组件共享一个中央知识存储区，根据共享的信息进行**推断和决策**。这两种风格在处理复杂问题和协作方面都具有一定的优势。

- 闭环控制：当软件被用来操作一个物理系统时，软件与硬件之间可以粗略的表示为一个反馈循环，这个反馈循环通过接受一定的输入，确定一系列的输出，最终使环境达到一个新的状态，适合于嵌入式系统，涉及连续的动作与状态。比如开空调，不会一调到某个温度就马上能到该温度，是逐渐接收室内的温度来变化输出空调的冷气；

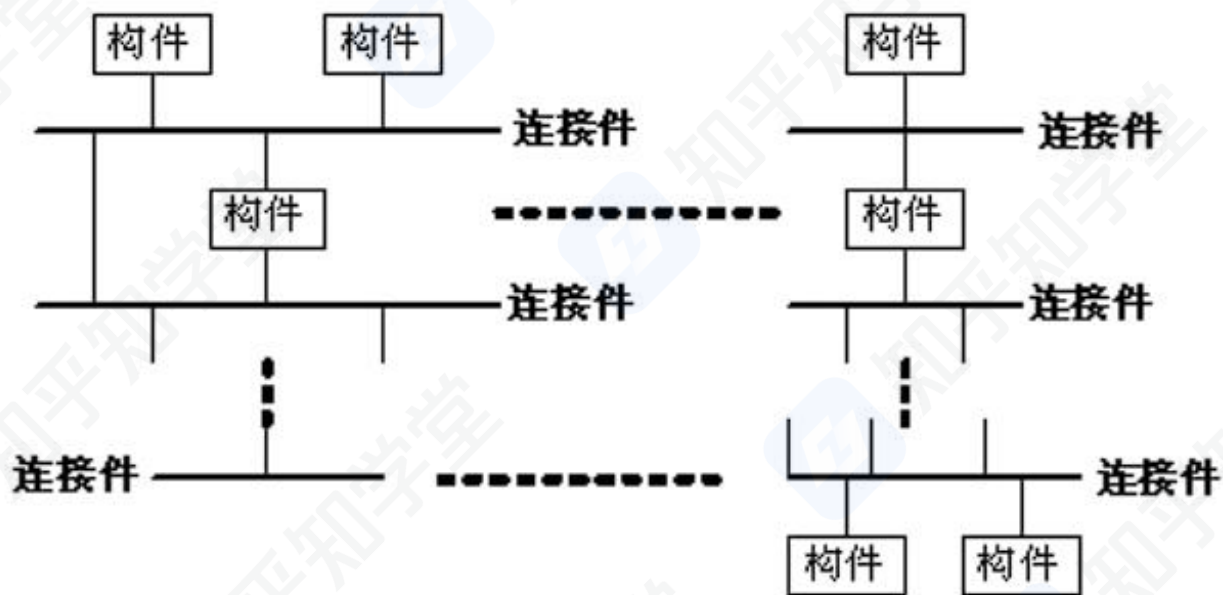


对于闭环控制系统，通常可以用以下的方框图表示：



➤ **C2**体系结构风格可以概括为：通过连接件绑定在一起的按照一组规则运作的并行构件网络。**C2**风格中的系统组织规则如下：

- 系统中的构件和连接件都有一个顶部和一个底部；
- 构件的顶部应连接到某连接件的底部，构件的底部则应连接到某连接件的顶部，而构件与构件之间的直接连接是不允许的；
- 一个连接件可以和任意数目的其它构件和连接件连接；
- 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。



序号	架构风格名	常考关键字及实例	简介
1	数据流-批处理	传统编译器，每个阶段产生的结果作为下一个阶段的输入，区别在于整体。	一个接一个，以整体为单位。
2	数据流-管道-过滤器		一个接一个，前一个输出是后一个输入。
3	调用/返回-主程序/子程序	无	显示调用，主程序直接调用子程序。
4	调用/返回-面向对象		对象是构件，通过对象调用封装的方法和属性。
5	调用/返回-层次结构		分层，每层最多影响其上下两层，有调用关系。
6	独立构件-进程通信		进程间独立的消息传递，同步异步。
7	独立构件-事件驱动(隐式调用)	事件触发推动动作，如程序语言的语法高亮、语法错误提示。	不直接调用，通过事件驱动。
8	虚拟机-解释器	自定义流程，按流程执行，规则随时改变，灵活定义，业务灵活组合。 机器人。	解释自定义的规则，解释引擎、存储区、数据结构。
9	虚拟机-规则系统		规则集、规则解释器、选择器和工作内存，用于DSS和人工智能、专家系统。
10	仓库数据库	现代编译器的集成开发环境IDE,以数据为中心。 又称为数据共享风格	中央共享数据源，独立处理单元。
11	仓库超文本		网状链接，多用于互联网。
12	仓库-黑板		语音识别、知识推理等问题复杂、解空间很大、求解过程不确定的这一类软件系统，黑板、知识源、控制。
13	闭环过程控制	汽车巡航定速，空调温度调节，设定参数，并不断调整。	发出控制命令并接受反馈，循环往复达到平衡。
14	C2风格	构件和连接件、顶部和底部。	通过连接件绑定在一起按照一组规则运作的并行构件网络

两层C/S架构：客户端和服务端都有处理功能，并且功能层集成在客户端里，客户端既要处理展示又要处理功能，所以也叫胖客户端，现在已经不常用，原因有：开发成本较高、客户端程序设计复杂、信息内容和形式单一、用户界面风格不一、软件移植困难、软件维护和升级困难、新技术不能轻易应用、安全性问题、服务器端压力大难以复用。

三层C/S架构：将处理功能独立出来，表示层和数据层都变得简单。表示层在客户机上，功能层在应用服务器上，数据层在数据库服务器上。即将两层C/S架构中的功能层从客户端中独立出来了。其优点下面四点：

- 各层在逻辑上保持相对独立，整个系统的逻辑结构更为清晰，能提高系统和软件的可维护性和可扩展性；
- 允许灵活有效的选用相应的平台和硬件系统，具有良好的可升级性和开放性；
- 各层可以并行开发，各层也可以选择各自最适合的开发语言；
- 功能层有效的隔离表示层与数据层，为严格的安全管理奠定了坚实的基础，整个系统的管理层次也更加合理和控制。

三层B/S架构： 是三层C/S架构的变种，将客户端变为用户客户端上的浏览器，将应用服务器变为网络上的WEB服务器，又称为0客户端架构，虽然不用开发客户端，但有很多缺点：

- 使用浏览器作为客户端的话安全性难以控制；
- 在数据查询等响应速度上，要远远低于C/S架构，因为C/S架构有部分数据存储在本地；
- 数据提交一般以页面为单位，数据的动态交互性不强。

混合架构风格：

- 内外有别模型：企业内部使用C/S,外部人员访问使用B/S。
- 查改有别模型：采用B/S查询，采用C/S修改。
- 注意：混合架构实现困难，且成本高。

总结：

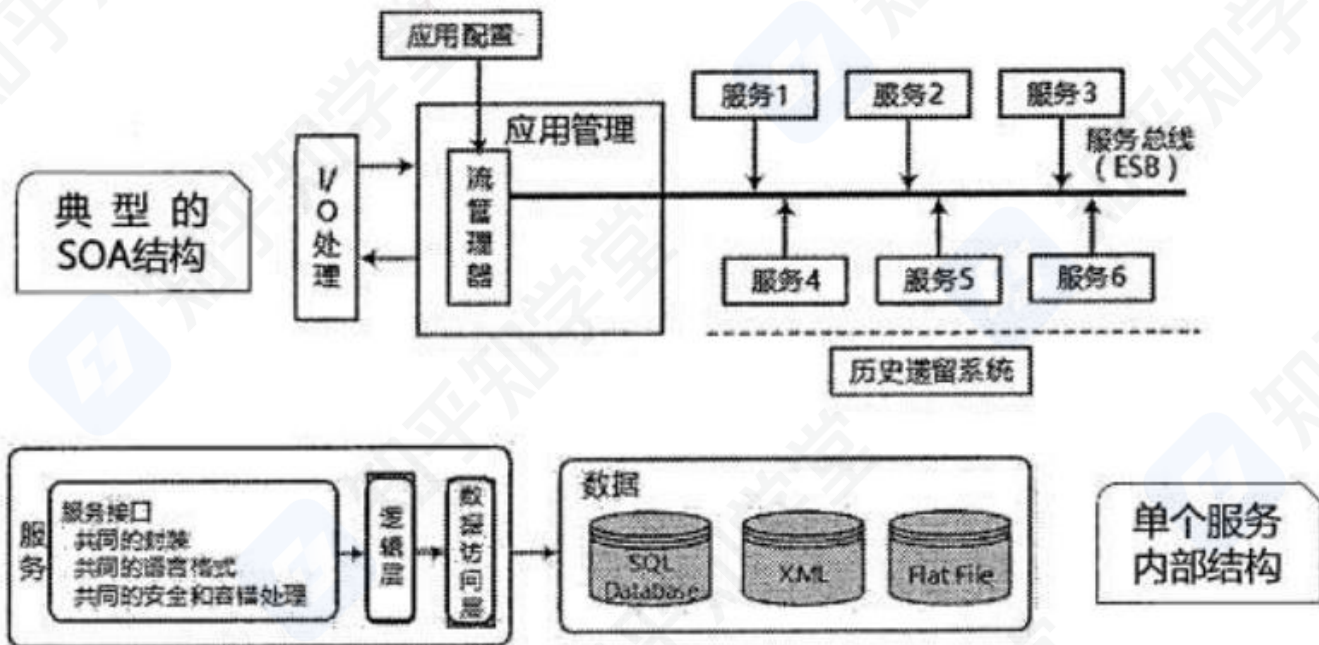
- 选择两层架构：适合业务简单、性能敏感、资源有限的场景（如小型工具类软件）。
- 选择三层架构：适合需要安全隔离、业务逻辑复用、长期维护的中大型系统（如电商平台、ERP系统）。其缺点可通过技术手段缓解，例如：
 - 使用高性能RPC框架（如gRPC）降低通信开销；
 - 通过容器化（Docker/K8s）简化部署和扩展；
 - 采用分布式缓存（Redis）减少数据库压力。

面向服务的架构风格

SOA是一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通信，不涉及底层编程接口和通信模型，类似于企业部门之间通过规范流程协作。

在SOA中，服务是一种为了满足某项业务需求的操作、规则等的逻辑组合，它包含一系列有序活动的交互，为实现用户目标提供支持。

SOA并不仅仅是一种开发方法，还具有管理上的优点，管理员可直接管理开发人员所构建的相同服务。多个服务通过企业服务总线(ESB)提出服务请求，由应用管理来进行处理，如下：



简单来说，SOA就类似于中国的政府服务中心

核心思想就是把分散的政府部门（系统功能）包装成"服务窗口"，通过统一的"政务大厅"（ESB总线）对外提供标准化服务，市民（应用）不需要知道具体找哪个部门，只需在窗口提交申请。

在面向服务的架构（SOA）中，有多种实现技术和方法，其中包括Web Service、服务注册表和企业服务总线（ESB）

- Web Service：利用标准化的Web协议实现服务的调用
- 服务注册表：用于服务的发现和查找；
- 企业服务总线（ESB）：作为服务间通信的中间件，提供消息路由和转换功能

WEB Service: WebService是一种网络服务，它使不同系统之间能够通过网络进行通信和数据交换。WebService基于标准的网络协议（如HTTP和HTTPS），并使用XML、JSON等数据格式，使系统能够跨平台、跨语言通信。WebService可以被视为SOA实现的核心技术之一，它通过服务接口使不同系统之间的数据交换变得更加容易。

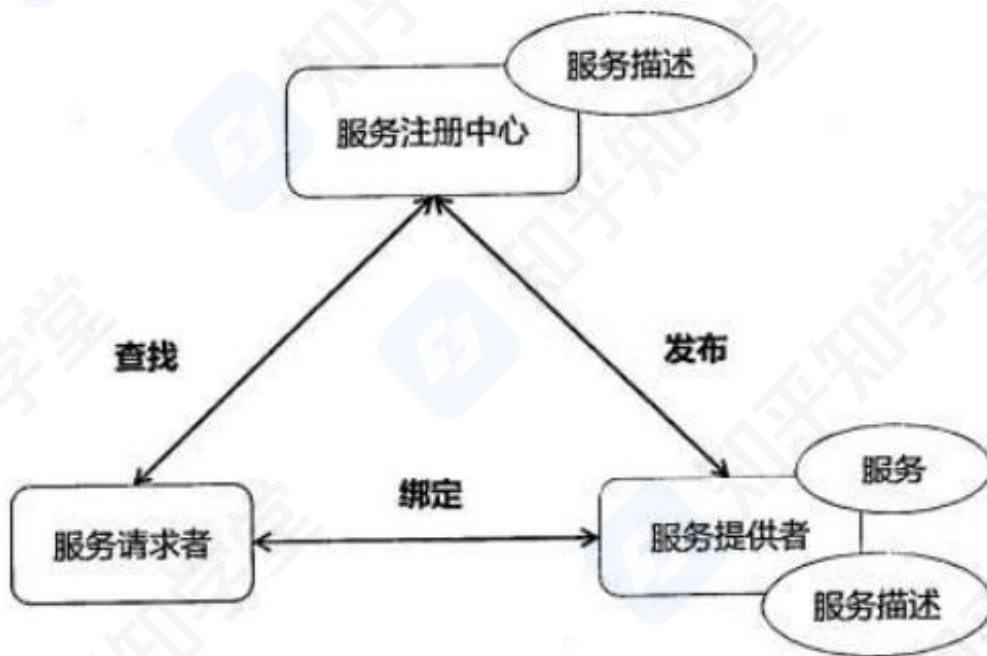
WebService主要由以下三个核心组件组成：

- 服务提供者（Service Provider）：提供具体服务的实体，负责发布服务并处理服务请求。
- 服务请求者（Service Consumer）：使用服务的实体，通过网络请求访问服务。
- 服务注册中心（Service Registry）：管理服务的注册和查找信息，服务提供者在服务注册中心注册服务，服务请求者可以从注册中心获取服务的访问地址。

面向服务的架构风格

WebService的工作流程通常包含以下步骤：

- 服务注册：服务提供者将服务的WSDL描述文档注册到服务注册中心。
- 服务查找：服务请求者通过查询服务注册中心，获取所需服务的WSDL文档。
- 服务调用：服务请求者根据WSDL文档描述的信息，构造请求消息，发送到服务提供者，进行了请求绑定。
- 服务响应：服务提供者处理请求并返回响应消息给服务请求者。



WebService中应用的关键技术如下表:

- UDDI: 是一套基于WEB的、分布式的、为WebService提供的、信息注册中心的**实现标准**, 同时也包含一组使企业能将自身提供的WebService注册, 以使别的企业能够发现的访问协议的实现标准, 用于WEB服务注册**统一描述、发现及集成**。就是一个**服务登记系统**, 让服务提供者发布服务, 让消费者查找服务。就像企业要在黄页上登记才能被客户找到一样。但在微服务时代, 这种"集中式黄页"已被更轻量的"服务发现机制" (如**Eureka**) 取代, 不过其核心思想【服务注册与发现】仍然是分布式系统的基石
- WSDL (Web Service描述语言): 将Web服务描述定义为一组服务访问点, 客户端可以通过这些服务访问点对包含面向文档信息或面向过程调用的服务进行访问(类似远程调用), 用于描述服务。**就类似于政务大厅的办事指南**
- SOAP(简单对象访问协议): 是用于交换XML编码信息的轻量级协议, 用于传递信息。**就类似于我们去办理护照, 不管哪个城市都是统一的格式**
- 速记口诀: **WSDL = 服务说明书** (说明书告诉你怎么用这个服务)、**SOAP = 信封格式** (信封规定信怎么写、怎么寄)、**UDDI = 黄页电话簿** (电话簿帮你找到服务在哪)

功能	协议
发现服务	UDDI DISCO
描述服务	WSDL XML Schema
消息格式层	SOAP REST
编码格式层	XML (DOM, SAX)
传输协议层	HTTP、TCP/IP、SMTP等



WebService的实现主要有两种协议标准：**基于SOAP的WebService**和**基于REST的WebService**。

- SOAP (Simple Object Access Protocol) 是一种基于XML的消息传递协议，通常用于实现企业级WebService。SOAP消息通常在HTTP协议的请求和响应体中传输，也支持其他传输协议（如SMTP）。SOAP协议规范包括消息格式、服务描述（WSDL）和消息传输的安全性。
- REST (Representational State Transfer) 是一种轻量级的WebService架构风格。它基于HTTP协议，使用URL表示资源，使用HTTP方法（GET、POST、PUT、DELETE）进行资源的操作。RESTful服务通过JSON或XML格式传输数据，更加简洁高效，广泛应用于互联网服务和轻量级应用。

举例：以一个电商系统为例，电商系统包含多个子系统，如用户管理系统、商品管理系统、订单系统、支付系统等。基于SOA架构，电商系统可以将这些子系统设计为独立的服务，并通过WebService进行通信。

- 用户管理服务：提供用户注册、登录、信息更新等功能。
- 商品管理服务：提供商品的增删查改等功能。
- 订单服务：提供订单的创建、查询、更新等功能。
- 支付服务：提供支付相关的功能，如支付处理、退款等。

每个服务都实现为独立的WebService，通过SOAP或REST API提供接口。每个服务的接口规范描述在WSDL文件中，供服务请求者调用，用户下单时，系统调用订单服务创建订单，同时调用支付服务处理支付。订单服务通过服务注册中心获取支付服务的接口地址，并向支付服务发起请求。支付完成后，支付服务返回结果给订单服务，订单状态更新。

企业服务总线ESB：简单来说是一根管道，用来连接各个服务节点。**ESB的存在是为了集成基于不同协议的不同服务，ESB做了消息的转化、解释以及路由的工作，以此来让不同的服务互联互通。**

包括：客户端(服务请求者)、基础架构服务(中间件)、核心集成服务(提供服务)。

ESB特点：

- SOA的一种实现方式，ESB在面向服务的架构中起到的是总线作用，将各种服务进行连接与整合；
- 描述服务的元数据和服务注册管理；
- 在服务请求者和提供者之间传递数据，以及对这些数据进行转换的能力，并支持由实践中总结出来的一些模式如同步模式、异步模式等；
- 发现、路由、匹配和选择的能力，以支持服务之间的动态交互，解耦服务请求者和服务提供者。高级一些的能力，包括对安全的支持、服务质量保证、可管理性和负载平衡等。





场景	技术特征	类比
企业应用集成(EAI)	需要把50个老系统（工商、税务、社保、公积金）串起来办一件事	多部门并联审批
跨系统业务流程	北京提交申请，自动流转 to 上海办理，全程透明追踪	跨省通办
严格治理要求	每一步操作留痕，符合审计要求，不能随意变更流程	涉密业务
服务重用	办护照、办驾照、办银行卡都要查身份证，复用同一个"身份认证服务"	身份证办理



面向服务的架构风格



在实际的SOA实现中，会遇到以下几个挑战：

1. 性能问题：由于WebService需要序列化和反序列化数据，尤其是在SOAP协议下，数据量较大，传输效率低，导致性能下降。可以通过以下方式优化：

- 使用RESTful API进行轻量级服务。
- 使用缓存机制减轻负载。

2. 安全问题：WebService的数据传输存在安全风险，尤其是在互联网环境下。解决方案包括：

- 使用HTTPS加密数据传输。
- 实现身份认证和访问控制（如OAuth、JWT等）。

3. 复杂数据的处理：在需要处理复杂数据结构时，可以选择SOAP协议，结合WSDL和XML Schema来支持复杂数据类型。



2025年下：综合知识



27.下列关于MVP (Model-View-Presenter) 模式的说法错误的是 ()

A.View可以直接访问model

B.MVP常用在移动端

C.M表示Model 数据实体, V表示View展示数据, P表示Presenter处理数据

D.一个Presenter可以对应多个页面

33.客户端服务器是属于 () 架构风格。

A.层次

B.独立构件

C.黑板

D.数据流

48、一个解释器通常包括完成解释工作的解释引擎，一个包含被解释的代码存储区，（ ）和一个记录源代码被解释执行进度的数据结构。

- A.解释器引擎的内部状态的数据结构
- B.工作日志存储
- C.保存代码的存储区状态
- D.规则引擎

58、智能家居涉及到各种传感器的使用，需要根据温度、湿度、用户的使用习惯和预定状态动态调整各项参数，最适合的是（ ）

- A. 规则系统
- B. 解释器
- C. 黑板风格
- D. 管道-过滤器

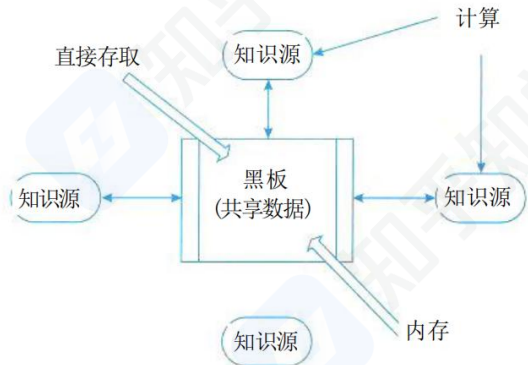


图7-14黑板体系结构风格

对于特定应用问题，黑板系统可通过选取各种黑板、知识源和控制模块的构件来设计，也可以利用预先定制的黑板体系结构的编程环境。黑板系统的传统应用是信号处理领域，如语音识别和模式识别。另一应用是松耦合代理数据共享存取。

7.3.5虚拟机体系结构风格

虚拟机体系结构风格的基本思想是人为构建一个运行环境，在这个环境之上，可以解析与运行自定义的一些语言，这样来增加架构的灵活性。虚拟机体系结构风格主要包括解释器风格和规则系统风格。

1. 解释器体系结构风格

一个解释器通常包括完成解释工作的解释引擎，一个包含将被解释的代码的存储区，一个记录解释引擎当前工作状态的数据结构，以及一个记录源代码被解释执行进度的数据结构。

具有解释器风格（见图7-15）的软件中含有一个虚拟机，可以仿真硬件的执行过程和一些关键应用。解释器通常被用来建立一种虚拟机以弥合程序语义与硬件语义之间的差异。其缺点是执行效率较低。典型的例子是专家系统。



2025年下：综合知识



知乎知学堂

65、下列关于微服务架构的描述，错误的是（）。

- A.必须部署在统一环境，否则无法相互调用
- B.服务间通过API（如REST、gRPC）通信
- C.每个服务聚焦单一业务功能
- D.支持独立部署和扩展



2026年上：综合知识



10、黑板架构中调用多个(问题1)构件进行数据处理和计算的。

- A.知识源
- B.黑板
- C.控制器
- D.数据源

17、在两层C/S架构中，胖客户端通常负责()

- A.仅数据管理
- B.仅用户界面展示
- C.用户界面交互和业务逻辑处理
- D.仅消息队列转发



43、在软件架构风格中，强调数据按顺序流经一系列处理构件、每个构件完成一类转换处理的是()

- A. 分层风格
- B. 管道-过滤器风格
- C. 事件驱动风格
- D. 仓库风格

51、在子程序-返回架构风格中，最典型的连接机制是()

- A. 过程调用
- B. 事件广播
- C. 发布订阅
- D. 共享黑板

55、在进程通信架构风格中，构件通常是独立进程，连接件通常是()

- A. 过程调用、消息传递或远程调用机制
- B. 类继承和对象实例化
- C. 表单与报表模板
- D. 黑板与控制器

2026年上：综合知识

61、关于隐式调用或事件驱动风格的表述，错误的是()

- A.事件触发者通常不知道会由谁处理
- B.一个构件既可能触发事件，也可能处理事件
- C.事件触发后一定能预先确定完整处理流程
- D.该风格有利于松耦合扩展

64、A微服务调用B微服务时，若B的接口只是偶尔出现响应延迟，为避免调用线程长时间阻塞，最直接合适的措施是()

- A.熔断
- B.限流
- C.服务降级
- D.设置超时时间

DSSA(Domain Specific Software Architecture,DSSA): 就是专用于一类特定类型的任务(领域)的、在整个领域中能有效地使用的、为成功构造应用系统限定了标准的组合结构的软件构件的集合。它旨在**满足该领域的独特需求和约束**。这种架构通常通过针对特定问题领域的专业知识和最佳实践来优化软件系统的设计, 以提供更高的性能、可维护性和可扩展性。DSSA 的特征: **领域性、普遍性、抽象性、可复用性**

- 例如: 在医疗保健领域, 电子病历系统是一个常见的应用, 用于管理患者的医疗记录。为了满足医疗保健行业的特殊需求, 可以使用DSSA来设计和构建这样的系统。通过采用DSSA, 这家医疗保健软件公司能够开发出适用于医疗保健领域的高度定制化且符合行业标准的电子病历系统。这就是特定领域的软件架构在医疗保健领域的一个示例。不同领域的DSSA可以根据其特殊需求进行定制化

DSSA就是一个特定的问题领域中支持一组应用的**领域模型、参考需求、参考架构**等组成的开发基础, 其目标就是支持在一个特定领域中多个应用的生成。

- **垂直域:** 在一个特定领域中的通用的软件架构, 是一个完整的架构。例如电子病历系统、医院信息系统或医学影像分析系统
- **水平域:** 在多个不同的特定领域之间的相同的部分的小工具(如购物和教育都有收费系统, 收费系统即是水平域)。例如网络安全通用架构等

DSSA的三个基本活动：领域分析、领域设计和领域实现

领域分析：这个阶段的主要目标是**获得领域模型(领域需求)**。识别信息源(需求)，即整个领域工程过程中信息的来源，可能的信息源包括现存系统、技术文献、问题域和系统开发的专家、用户调查和市场分析、领域演化的历史记录等，在此基础上就可以分析领域中系统的需求，确定哪些需求是领域中的系统广泛共享的，从而建立领域模型。

- 比如在医疗系统中，领域分析阶段，团队收集了与医院管理相关的信息源，包括医疗领域的法规、患者需求、医院流程和现有系统。通过与医院管理员、医生和护士的讨论以及研究医疗保健法规，他们确定了系统的需求，如患者信息记录、医生排班、药物管理等。这些需求构成了领域模型，也就是医院信息管理领域的需求。

领域设计：这个阶段的目标是**获得DSSA**。DSSA描述在领域模型中表示的需求的解决方案，它不是单个系统的表示，而是能够适应领域中多个系统的需求的一个高层次的设计。建立了领域模型之后，就可以派生出满足这些被建模的领域需求DSSA。

- 举例：在领域设计阶段，基于领域模型，团队提供了医院信息管理系统的高层次设计。这个设计不是一个具体的应用程序，而是一个通用的架构。它包括模块化组件，如患者信息管理模块、医生排班模块、药物管理模块等。这些模块设计成可扩展和可重用的，以便满足不同医院的需求。这个领域设计能够适应医院信息管理领域中多个系统的需求。



领域实现：这个阶段的主要目标是依据领域模型和**DSSA**开发和组织可重用信息。这些可重用信息可能是从现有系统中提取得到，也可能需要通过新的开发得到。

- 举例：在领域实现阶段，团队根据领域模型和领域设计来开发具体的医院信息管理系统。他们实现了患者信息管理模块，包括患者信息录入、查看和编辑功能。同时，他们也开发了医生排班模块，以及药物管理模块，确保这些模块符合领域设计的要求。这些模块的开发是基于领域模型和DSSA的指导原则，以确保系统的可维护性和可重用性。

领域分析用于确定需求，领域设计用于提供通用架构，而领域实现用于将该架构转化为具体的应用程序模块。这个过程有助于确保系统能够满足特定领域的需求，并具备可维护和可重用的特性。

参与DSSA的四种角色人员：**领域专家、领域分析人员、领域设计人员和领域实现人员**

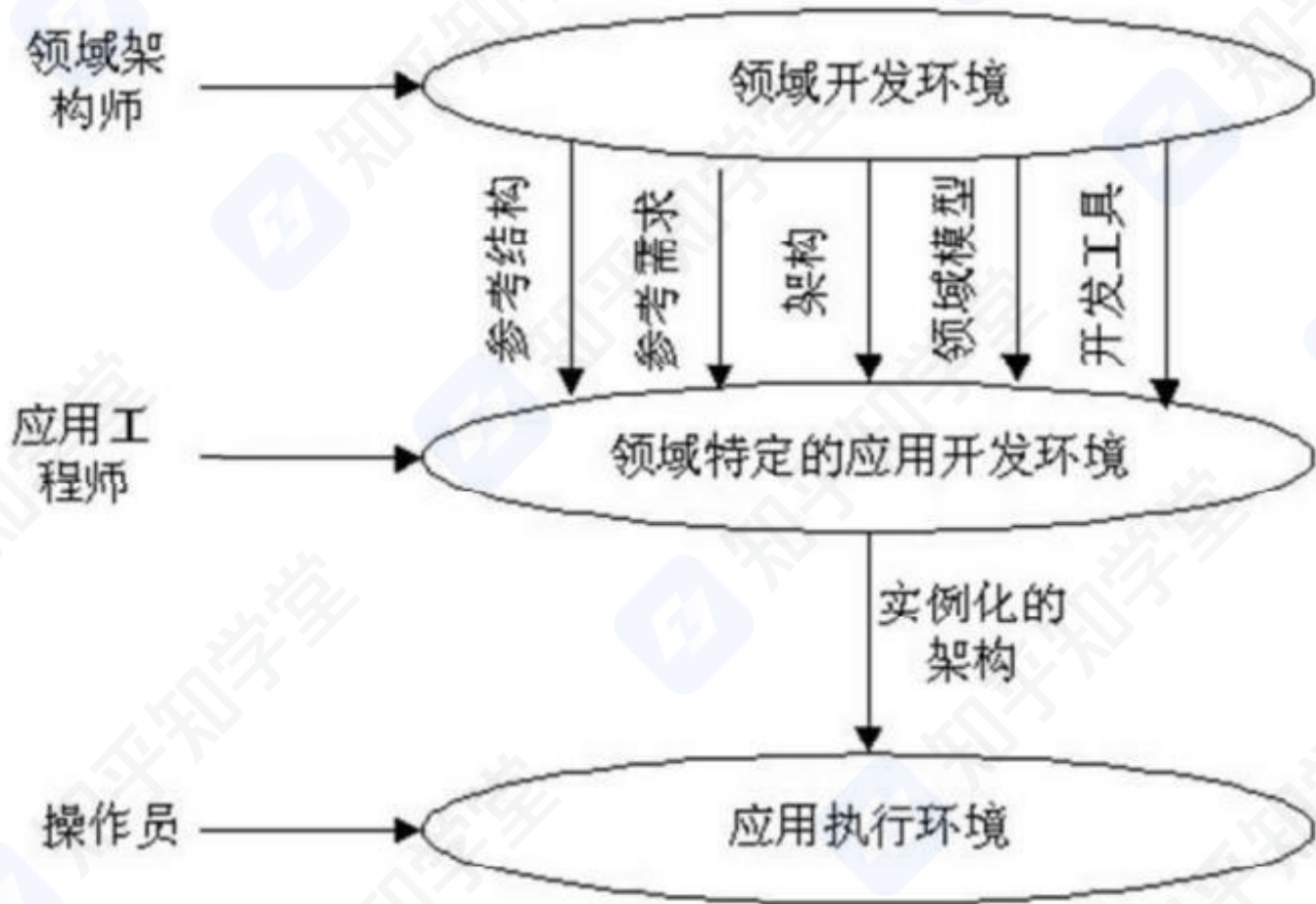
- 领域专家：包括该领域中系统的有经验的用户、从事该领域中系统的需求分析、设计、实现以及项目管理的有经验的软件工程师等。提供关于领域中系统的需求规约和实现的知识，帮助组织规范的、一致的领域字典，帮助选择样本系统作为领域工程的依据，复审领域模型、DSSA等领域工程产品，等等。
- 领域分析人员：由具有知识工程背景的经验丰富的系统分析员来担任。控制整个领域分析过程，进行知识获取，将获取的知识组织到领域模型中。
- 领域设计人员：由有经验的软件设计人员来担任。根据领域模型和现有系统开发出DSSA，并对DSSA的准确性和一致性进行验证。
- 领域实现人员：由有经验的程序设计人员来担任。根据领域模型和DSSA，开发构件。

建立DSSA的过程(实现的角度):

1. 定义领域范围: 领域中的应用要满足用户一系列的需求。比如我们正在开发一个在线教育平台领域, 其领域范围: 涵盖课程管理、学生管理、教师管理、在线考试等功能。
2. 定义领域特定的元素: 建立领域的字典, 归纳领域中的术语, 识别出领域中相同和不相同的元素。比如学生、分数、老师等元素就是在线教育平台领域的特定元素。
3. 定义领域特定的设计和实现需求的约束: 识别领域中的所有约束, 这些约束对领域的设计和实现会造成什么后果。比如我要求系统能够支持同时在线 10,000 名学生, 并且用户数据需加密存储, 防止信息泄露。
4. 定义领域模型和架构: 设计通用架构, 并描述其组件。比如领域模型有个模型叫做课程模型, 该模型实现的架构是 MVC, 其组件描述是处理课程的发布、修改、删除
5. 产生、搜集可复用的产品单元: 为DSSA增加复用构件, 使可用于新的系统。比如用户认证模块、课程模块都可以在相同领域的系统中复用

以上过程是并发的、递归的、反复的、螺旋型的。

三层次系统模型：领域开发环境、领域特定的应用开发环境、应用执行环境



领域开发环境：在领域开发环境中，领域架构师负责制定医院信息管理系统核心架构，以及定义了系统的参考结构、参考需求、架构、领域模型和开发工具。这个环境的任务是为特定领域(医院信息管理)创建通用的框架，以满足多个医院信息系统的需求。

- 例如，领域架构师可能决定使用分布式数据库系统以确保数据的可扩展性和安全性，还可能定义患者信息管理、医生排班和药物管理等领域需求。

领域特定的应用开发环境：在领域特定的应用开发环境中，应用工程师根据具体的医院信息管理系统需求，将核心架构实例化为一个具体的应用程序。这个环境会根据领域开发环境提供的参考结构和模型，来创建医院A、医院B等具体医院信息管理系统的实例。

- 例如，对于医院A，应用工程师会根据核心架构和领域模型来定制患者信息管理模块，以适应医院A的需求。同样，对于医院B，应用工程师也会根据同一核心架构，但可能进行不同的定制以满足医院B的特定需求。

应用执行环境：应用执行环境是医院信息管理系统最终运行的地方，由操作员负责实际使用和维护。在这个环境中，已经实例化的医院信息管理系统被部署和运行。

- 例如，医院A的操作员使用医院A的实例化系统来记录和管理患者信息，医院B的操作员使用医院B的实例化系统来执行相同的任务。这个层次关注的是系统的实际运行和操作。



2025年下：综合知识



知乎知学堂

47、DSSA领域分析阶段产物是（）

- A. 领域模型
- B. 领域知识
- C. 特定领域
- D. 领域样本



2026年上：综合知识



18、在DSSA建立过程中，产生领域字典或领域术语表的阶段通常是()

- A.定义领域参考架构
- B.定义领域特定元素
- C.进行领域实现编码
- D.进行系统切换部署

49、DSSA的建立过程通常具有并发、递归和反复特征，因此更接近()式过程。

- A.瀑布
- B.螺旋
- C.V模型
- D.喷泉

基于架构的软件开发(Architecturally Based Software Development, ABSD)是一种软件开发方法，**强调在开发过程中首先定义系统的体系结构，然后根据这个体系结构来实现系统。它有助于确保系统的结构和设计与业务需求保持一致**

假设一家电子商务公司决定开发一个全新的在线购物网站，他们采用基于架构的软件开发方法：

- 定义系统架构：首先，开发团队会定义系统的体系结构，包括用户界面层、业务逻辑层和数据存储层。这些层次将被明确定义，以确保开发过程中每个组件的职责清晰明确。
- ABSD的关键决策：在系统架构阶段，团队会做出一些关键的架构决策，例如选择使用哪种技术堆栈、如何处理用户身份验证、如何处理库存管理、如何处理支付等。这些决策是ABSD的核心，它们会在整个开发过程中起到指导作用。
- 系统实施：一旦系统架构被定义和核心决策被制定，开发团队会开始实施系统。他们会根据架构中的不同层次来编写代码，确保各个组件按照系统设计进行开发。
- 持续维护和演化：随着时间的推移，业务需求可能会发生变化，但由于系统的基本架构已经定义，团队可以相对容易地进行扩展和修改，而不必重新设计整个系统。

基于架构的软件开发方法强调了在软件开发过程中先关注系统的结构和核心决策，以确保最终的系统能够满足业务需求并具有良好的扩展性和维护性。**这个方法有助于降低项目失败的风险，因为它强调了在开发之前做出关键决策的重要性。**



ABSD方法是架构驱动，强调由业务、质量和功能需求的组合驱动架构设计。它强调采用视角和视图来描述软件架构，采用用例和质量属性场景来描述需求。进一步来说，用例描述的是功能需求，质量属性场景描述的是质量需求(或侧重于非功能需求)。

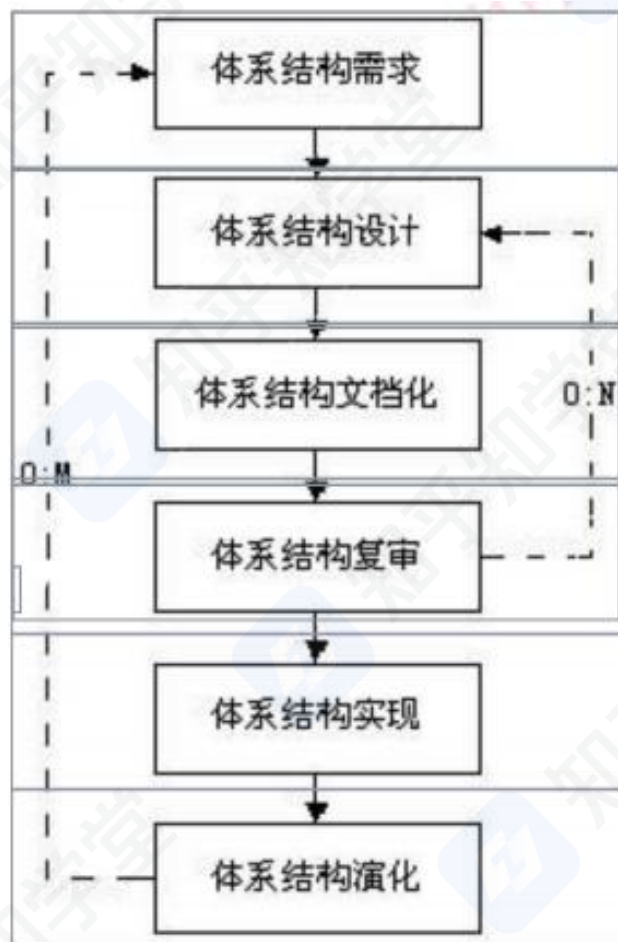
使用ABSD方法，设计活动可以从项目总体功能框架明确就开始，这意味着需求获取和分析还没有完成，就开始了软件设计。

ABSD方法有三个基础：

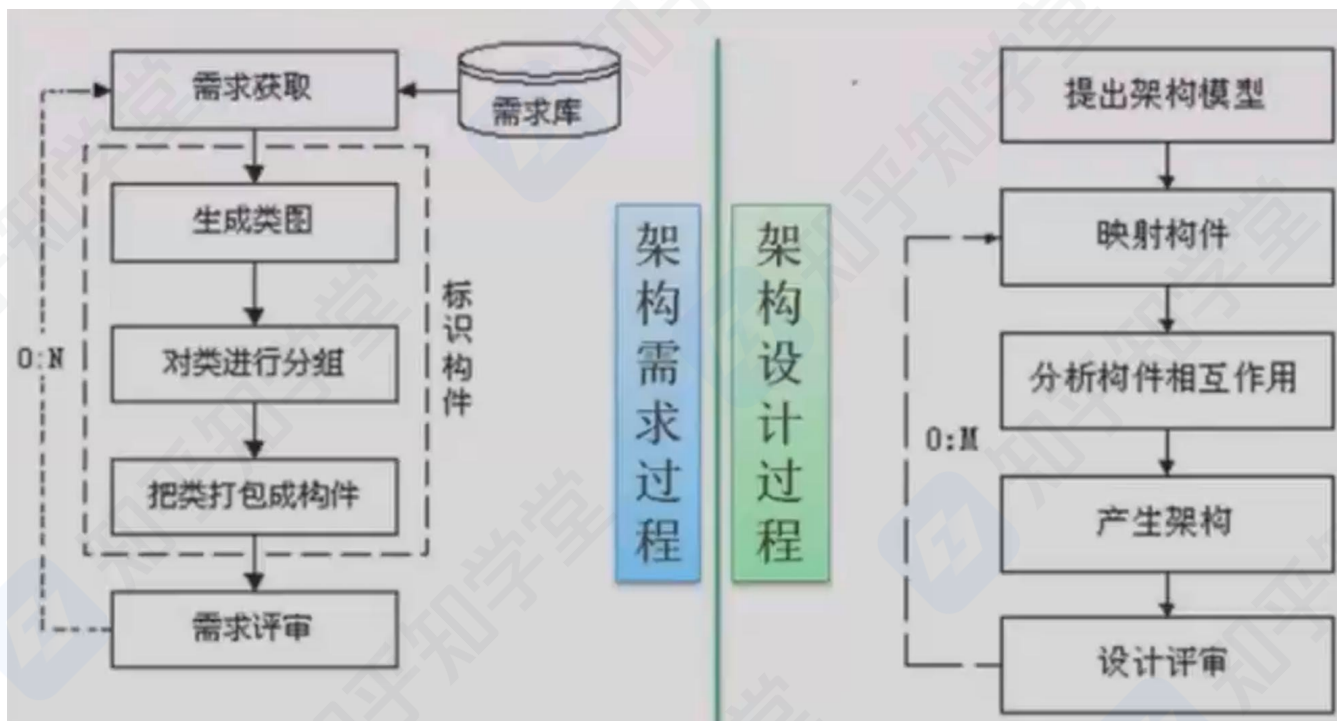
- 第一个基础：是功能的分解，使用已有的基于模块的内聚和耦合技术
- 第二个基础：是通过选择架构风格来实现质量和业务需求
- 第三个基础：是软件模板的使用，软件模板利用了一些软件系统的结构进行复用。

ABSD方法是递归的，且迭代的每一个步骤都是清晰定义的。因此，不管设计是否完成，架构总是清晰的，有助于降低架构设计的随意性。

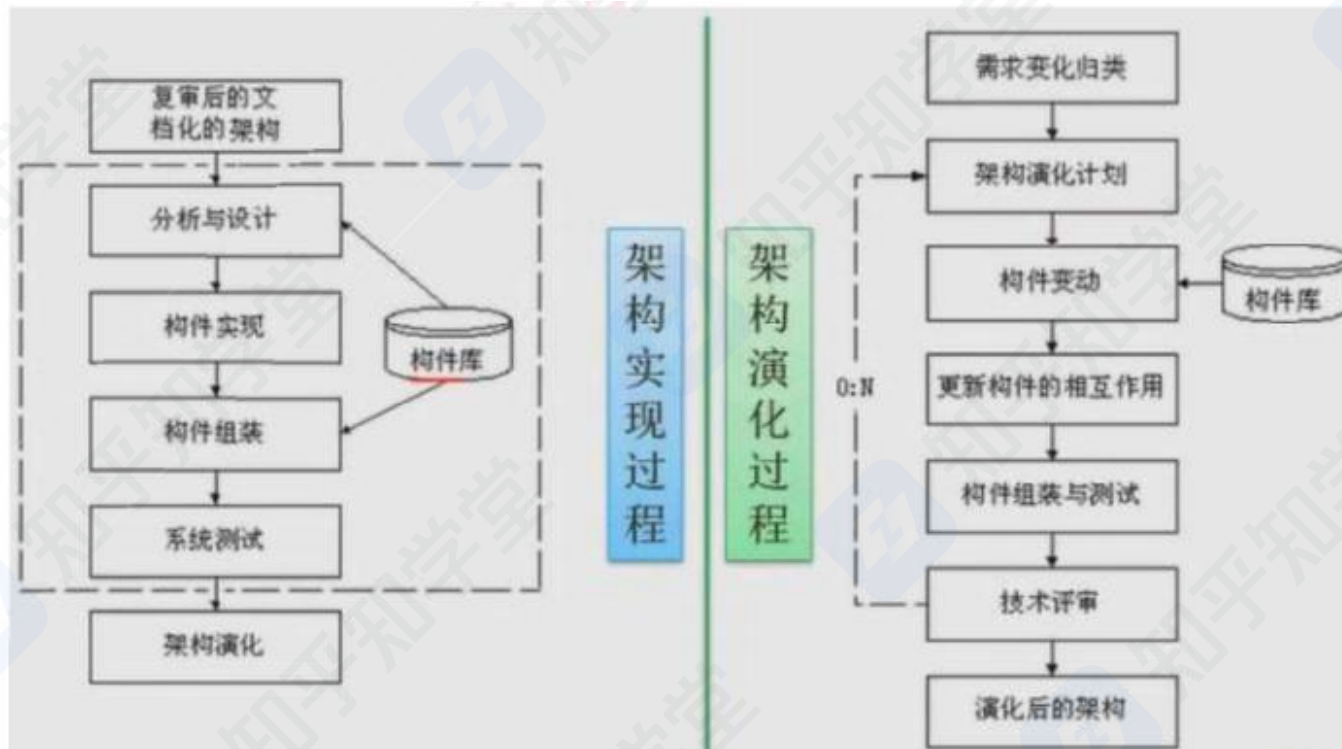
基于架构的软件开发过程可分为下列六步：



1. 架构需求：重在掌握标识构件的三步，如下左图。
2. 架构设计：将需求阶段的标识构件映射成构件，进行分析，如下右图。
3. 架构(体系结构)文档化：主要产出两种文档，即**架构(体系结构)规格说明**，**测试架构(体系结构)需求的质量设计说明书**。文档是至关重要的，是所有人员通信的手段，关系开发的成败。



4. 架构复审：由外部人员(独立于开发组织之外的人，如用户代表和领域专家等)参加的复审，复审架构是否满足需求，质量问题，构件划分合理性等。若复审不过，则返回架构设计阶段进行重新设计、文档化，再复审。
5. 架构实现：用实体来显示出架构。实现构件，构件组装成系统，如下左图：
6. 架构演化：对架构进行改变，按需求增删构件，使架构可复用，如下右图：



基础信息对比

维度	CBSD	ABSD
中文名称	基于构件的软件开发	基于架构的软件开发
核心目标	通过预制构件快速组装系统	通过架构设计保障系统质量与扩展性
关注焦点	构件的复用性与集成方式	系统整体结构与关键质量属性（性能、安全、可靠性等）
典型应用场景	企业级应用快速开发（如ERP、CRM）	复杂系统开发（如自动驾驶系统、分布式交易平台）

关键技术对比

技术特征	CBSD	ABSD
核心组件	构件库（如Docker镜像仓库）、集成框架（Spring Boot）	架构描述语言（ADL）、架构评估工具（ATAM）
复用单元	业务级构件（如支付模块、用户管理组件）	架构模式（如微服务、分层架构）
质量保障	依赖构件的单元测试	基于架构的代码生成、形式化验证
变更影响	构件替换可能导致接口冲突	架构变更需重新评估质量属性



2025年下：综合知识



知乎知学堂

28.ABSD方法的三个核心不包括（）

- A.对系统进行功能分解
- B.采用架构风格实现质量属性和商业需求
- C.采用软件模板设计软件结构
- D.项目管理



2026年上：综合知识



知乎知学堂

13.ABSD方法经过递归细化后，最终可产生()

- A.用户手册和运行手册
- B.构件和类
- C.需求基线和测试用例
- D.数据仓库和数据集市



02

质量属性和架构评估



软件系统的质量属性

可以将软件系统的质量属性分为**开发期质量属性**和**运行期质量属性**2个部分。

1. 开发期质量属性主要指在软件开发阶段所关注的质量属性，这个阶段关注人群主要是开发者，包含**6**个方面。

- 易理解性：指设计被开发人员理解的难易程度。
- 可扩展性：软件因适应新需求或需求变化而增加新功能的能力，也称为灵活性。
- 可重用性：指重用软件系统或某一部分的难易程度。
- 可测试性：对软件测试以证明其满足需求规范的难易程度。
- 可维护性：当需要修改缺陷、增加功能、提高质量属性时，识别修改点并实施修改的难易程度。
- 可移植性：将软件系统从一个运行环境转移到另一个不同的运行环境的难易程度。

2. 运行期质量属性主要指在软件运行阶段所关注的质量属性，这个阶段关注人群主要是用户，包含**7**个方面。

- 性能：性能是指软件系统及时提供相应服务的能力，如速度、吞吐量和容量等的要求。
- 安全性：指软件系统同时兼顾向合法用户提供服务，以及阻止非授权使用的能力。
- 可伸缩性：指当用户数和数据量增加时，软件系统维持高服务质量的能力。例如，通过增加服务器来提高能力。
- 互操作性：指本软件系统与其他系统交换数据和相互调用服务的难易程度。
- 可靠性：软件系统在一定的时间内持续无故障运行的能力。
- 可用性：指系统在一定时间内正常工作的时间所占的比例。可用性会受到系统错误，恶意攻击，高负载等问题的影响。
- 鲁棒性：是指软件系统在非正常情况(用户进行了非法操作、相关软硬件系统发生了故障)下仍能够正常运行的能力，也称健壮性或容错性。



1. 性能：指系统的响应能力，即要经过多长时间才能对某个事件做出响应，或者在某段时间内系统所能处理的事件的个数。如响应时间、吞吐量。
 - 设计策略：优先级队列、增加计算资源、减少计算开销、引入并发机制、采用资源调度等。
2. 可靠性：是指系统在一段时间内保持正常运行而不发生故障的能力。它强调了系统的稳定性和可靠性，通常是通过衡量系统在一段时间内发生故障的概率来评估的。一个可靠性高的系统意味着它很少出现故障，用户可以信任它的稳定性。可靠性有一些指标需要了解。如MTTF(平均故障时间)、MTBF(平均故障间隔时间)、MTTR(平均故障修复时间)。
 - 设计策略：心跳、Ping/Echo、冗余、选举。
3. 可用性：是指系统在需要的时候可供使用的能力，即系统处于可操作状态的时间比例。可用性通常通过计算系统在一定时间内可操作的百分比来评估。一个高可用性的系统意味着它在大部分时间内都是可用的，用户可以随时访问和使用它。
 - 设计策略：心跳、Ping/Echo、冗余、选举。



可靠性和可用性的区别：假设有两家云存储服务提供商：Provider A 和 Provider B。

- Provider A 的服务在过去一年中从未发生过故障，用户可以随时访问和上传文件。这表明 Provider A 的系统具有很高的可靠性，因为它极少出现故障。
 - Provider B 的服务在过去一年中总共停机了5小时，但它提供了冗余备份和快速恢复机制，因此用户在服务中断后很快就可以继续使用。这表明 Provider B 的系统具有很高的可用性，因为它在大部分时间内都是可操作的，尽管它发生了一些故障。
 - 在这个示例中，Provider A 的系统更可靠，因为它几乎没有发生故障。Provider B 的系统更可用，因为它尽管发生了一些故障，但它能够快速恢复，以确保用户可以继续使用。可靠性强调系统不发生故障，而可用性强调系统需要时可供使用。两者都是衡量系统性能和质量的重要标准，但它们关注的方面略有不同。
4. 安全性：是指系统在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。如保密性、完整性、不可抵赖性、可控性。
- 设计策略：入侵检测、用户认证、用户授权、追踪审计。



5. 可修改性：指能够快速的以**较高的性价比对系统进行变更的能力**。通常以某些具体的变更为基准，通过考察这些变更的代价衡量。
 - 设计策略：接口-实现分类、抽象、信息隐藏（是不是感觉有点像结构化开发和面向对象开发的设计原则）
6. 功能性：**是系统所能完成所期望的工作的能力**。一项任务的完成需要系统中许多或大多数构件的相互协作。
7. 可变性（可扩展）：**指体系结构经扩充或变更而成为新体系结构的能力**。这种新体系结构应该符合预先定义的规则，在某些具体方面不同于原有的体系结构。当要将某个体系结构作为一系列相关产品的基础时，可变性是很重要的。
8. 互操作性：作为系统组成部分的软件不是独立存在的，经常与其他系统或自身环境相互作用。为了支持互操作性，软件体系结构必须为外部可视的功能特性和数据结构提供精心设计的软件入口。**程序和用其他编程语言编写的软件系统的交互作用就是互操作性的问题，也影响应用的软件体系结构。**
9. 易用性：**它关注的是软件系统的用户界面和交互设计，以确保用户能够轻松、高效地使用系统，并感到满意。**易用性不仅关乎用户界面的外观，还包括用户体验、交互流程和用户学习曲线等方面

质量属性分类	核心定义与特征	关键子属性/策略
性能 (Performance)	系统响应能力，通过单位时间处理事务数或单个事务耗时衡量。重点优化资源管理和调度策略。	-响应时间 -吞吐量 -资源优化策略（并发、缓存、负载均衡）
可靠性 (Reliability)	系统在错误或意外情况下维持功能的能力，通过MTTF/MTBF量化。重点关注错误处理机制。	-容错：错误发生时自动修复（如冗余机制） -健壮性：错误时安全终止（如异常处理）
可用性 (Availability)	系统正常运行时间比例，通过故障间隔时间和恢复速度衡量。依赖故障检测与恢复策略。	-心跳检测 -冗余切换（主从库、灾备机房） -状态同步（检查点/回滚）
安全性 (Security)	阻止非授权访问并保障数据安全，涵盖机密性、完整性、可控性等维度。需平衡安全与性能。	-身份认证（OAuth/JWT） -访问控制（RBAC） -数据加密（TLS/SSL） -审计追踪
可修改性 (Modifiability)	以低成本高效修改系统的能力，包括维护、扩展和重组。需通过高内聚低耦合设计实现。	-可维护性：局部化修改（模块化） -可扩展性：松耦合接口（插件化） -结构重组：动态配置（微服务） -可移植性：环境无关性
功能性 (Functionality)	系统完成预期任务的能力，需通过架构设计协调多组件协作。	-接口标准化（OpenAPI） -服务编排（工作流引擎）
可变性 (Changeability)	架构适应未来变更的能力，支持产品线开发或多版本演进。	-配置驱动设计（Feature Toggle） -抽象核心逻辑（领域驱动设计）
互操作性 (Interoperability)	系统与其他系统交换数据或服务的能力，依赖标准化接口协议。	-协议兼容（HTTP/REST、gRPC） -数据格式统一（JSON Schema、Protobuf）

质量属性场景是一种用于描述系统如何满足特定质量属性需求的情境或情景。它由6部分组成：

- 刺激源(谁)：这是某个生成该刺激的实体(人、计算机系统或者任何其他刺激器)。
- 刺激(做什么)：该刺激是当刺激到达系统时需要考虑的条件。
- 环境(在什么样的环境下)：该刺激在某些条件内发生。当激励发生时，系统可能处于过载、运行或者其他情况。
- 制品(对哪个功能)：某个制品被激励。这可能是整个系统，也可能是系统的一部分。
- 响应(得到什么反馈)：该响应是在激励到达后所采取的行动。
- 响应度量(对反馈进行度量)：当响应发生时，应当能够以某种方式对其进行度量，以对需求进行测试。

可修改性质量属性场景描述实例：

场景要素	可能的情况
刺激源	最终用户、开发人员、系统管理员
刺激	希望增加、删除、修改、改变功能、质量属性、容量等
环境	系统设计时、编译时、构建时、运行时
制品	系统用户界面、平台、环境或与目标系统交互的系统
响应	查找架构中需要修改的位置，进行修改且不会影响其他功能，对所做的修改进行测试，部署所做的修改
响应度量	根据所影响元素的数量度量的成本、努力、资金；该修改对其他功能或质量属性所造成影响的程度

举例：假设我们正在设计一个电子商务网站，其中一个关键的质量属性是性能，希望确保网站在高负载时仍能快速响应

场景：高负载购物日：

描述：在每年的假期购物季节（如圣诞节），我们预期会有大量的用户访问我们的网站，进行在线购物。在高负载购物日，我们预计用户流量将增加至平时的**10倍**。

- 刺激源（谁）：刺激源是购物网站的用户，这些用户在特定的购物季节（例如圣诞节）涌入网站，导致高负载情况。
- 刺激（做什么）：刺激是用户访问购物网站，浏览产品、添加商品到购物车以及进行结算等在线购物活动。
- 环境（在什么样的环境下）：环境是购物网站在特定的日期范围内，即12月24日至12月25日，而且用户流量显著增加，购物活动频繁进行。
- 制品（对哪个功能）：制品是购物网站的整个系统，特别是与用户交互的部分，包括网站的前端和后端。
- 响应（得到什么反馈）：响应是购物网站采取的一系列行动，以确保在高负载购物日仍能够提供快速响应和正常运行的服务。这包括使用负载均衡、缓存静态内容、垂直扩展和水平扩展等措施。
- 响应度量（对反馈进行度量）：响应度量是在高负载购物日期间对系统性能进行度量的过程。它包括测量网页加载时间、服务器资源利用率、响应时间、错误率等指标，以确保系统在性能方面达到了预期目标。



2025年下：综合知识



59、一个软件可以在不同架构、操作系统、编译器、环境运行、编译，这体现了软件的（ ）

- A. 可变性
- B. 可移植性
- C. 互操作性
- D. 互操作性



2、下列质量属性中，全部属于运行期质量属性的是()

- A.安全性、可修改性、性能
- B.安全性、可用性、性能
- C.可测试性、可用性、可维护性
- D.可扩展性、可移植性、可维护性

58、系统的学习曲线和用户操作效率，主要体现的软件质量属性是()

- A.可用性
- B.易用性
- C.可靠性
- D.可维护性

69.在运行中的系统上人为关闭一台节点或服务器，观察业务是否仍可继续提供服务，这种测试场景主要是在验证系统的()

- A.容错性
- B.易用性
- C.可测试性
- D.可修改性



2026年上：综合知识



70、统一支付系统，要求增加支付宝的方式，且在5人/天之内完成，体现的软件质量属性是()

- A.可修改性
- B.可靠性
- C.兼容性
- D.性能

系统架构评估中的重要概念

敏感点：是指为了实现某一种特定的质量属性，一个或多个构件所具有的特性。

权衡点：是影响多个质量属性的特性，是多个质量属性的敏感点。

风险承担者：该架构所涉及到的利益相关人

软件架构评估在架构设计之后，系统设计之前，因此与设计、实现、测试都没有关系。评估的目的是为了评估所采用的架构是否能解决软件系统需求，但不是单纯的确定是否满足需求，有时候还需要针对质量属性进行评估架构是否能满足。

概念	核心判断标准（关键问题）	典型特征
敏感点	调整设计细节后，是否导致单个质量属性显著变化？	对“单一质量属性”敏感
权衡点	调整设计细节后，是否出现多个质量属性“此消彼长”？	多质量属性的“相互制约”
风险点	设计是否存在潜在隐患，可能导致系统故障或需求不达标？	存在“不确定性”或“隐患”
非风险点	设计是否成熟稳定，几乎不会引发问题（如技术验证充分、符合标准）？	“低不确定性”“高稳定性”

风险承担者	职责	所关心的问题
系统生产者		
软件系统架构师	负责软件架构的质量需求间进行权衡的人	对其他风险承担者提出的质量需求的折中和调停
开发人员	设计人员或程序员	架构描述的清晰与完整、各部分的内聚性与受限耦合、清楚的交互机制
维护人员	系统初次部署完成后对系统进行更改的人	可维护性，确定出某个更改发生后必须对系统中哪些地方进行改动的能力
集成人员	负责构件集成和组装的开发人员	与上同
测试人员	负责系统测试的开发人员	集成、一致的错误处理协议，受限的构件耦合、构件的高内聚性、概念完整性
标准专家	负责所开发软件必须满足的标准细节的开发人员	对所关心问题的分离、可修改性和互操作性



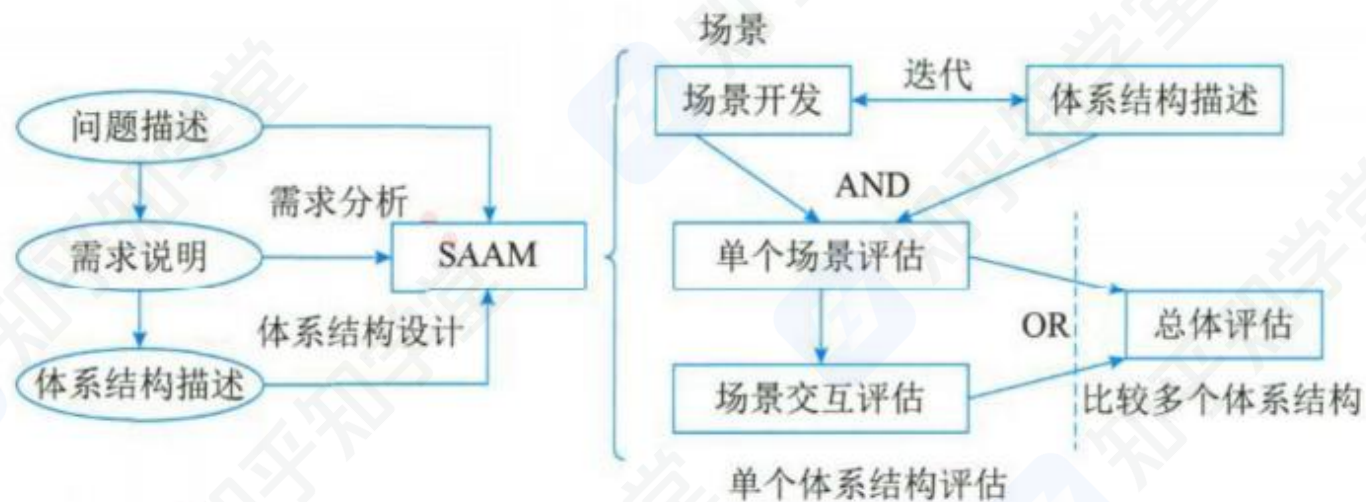
- 1. 基于调查问卷(检查表)的方式：**类似于需求获取中的问卷调查方式，只不过是架构方面的问卷，这种方式要求评估人员对领域和架构具有一定的了解，因为他们需要能够理解并回答与架构相关的问题。这种方式通常用于获取主观反馈和意见，以便改进架构。
 - 举例：对于一个大型电子商务平台的架构评估，评估团队可以向开发团队发送架构调查问卷，询问他们对性能、可维护性和安全性等方面的看法。问题可能包括：“您认为当前系统的性能是否满足需求？”或者“您认为系统的安全措施是否足够？”
- 2. 基于度量的方式：**制定一些定量指标来度量架构，如代码行数、内存使用、响应时间等，来评估系统的各个方面。这种方式要求评估人员对架构的技术细节和度量标准有一定了解
 - 举例：对于一个大型数据库管理系统的架构评估，评估团队可以度量数据库查询的平均响应时间、事务处理的吞吐量、数据库表的大小等指标。这些度量可以用于评估系统的性能和可伸缩性。



- 3. 基于场景的方式：**主要方法。这是一种系统性的评估方法，涉及定义一系列场景或情境，以测试系统在各种条件下的表现，每个场景描述了一种特定的使用情况，包括输入、预期输出和性能指标。评估人员会模拟这些场景，并评估系统是否满足质量属性需求。
- 实施过程：首先要确定应用领域的功能和软件架构的结构之间的映射，然后要设计用于体现待评估质量属性的场景(即4+1视图中的场景)，最后分析软件架构对场景的支持程度。要求评估人员即对领域熟悉，也对架构熟悉。
 - 从三个方面对场景进行设计：刺激(事件)、环境(事件发生的环境)和响应(架构响应刺激的过程)。
 - 举例：对于一个网络视频流媒体服务的架构评估，评估团队可以定义场景，如“同时1000名用户观看高清视频”或“在网络拥塞时仍能提供无缝的视频流”。然后，他们会模拟这些场景，测试系统的性能、可用性和容错性。

基于场景的架构分析方法SAAM: SAAM是一种非功能质量属性的架构分析方法, **是最早形成文档并得到广泛应用的软件架构分析方法。**

- 特定目标。SAAM的目标是对描述应用程序属性的文档, 验证基本的架构假设和原则。
- 质量属性。这一方法的基本特点是**把任何形式的质量属性都具体化为场景, 但可修改性是SAAM分析的主要质量属性。**
- 架构描述。SAAM用于架构的最后版本, 但早于详细设计。架构的描述形式应当被所有参与者理解。
- 功能、结构和分配被定义为描述架构的3个主要方面。
- 方法活动。SAAM的主要输入是**问题描述、需求声明和架构描述。**下图描绘了SAAM分析活动的相关输入及评估过程。包括5个步骤, 即**场景开发、架构描述、单个场景评估、场景交互和总体评估。**



系统架构评估方法



289-304页详细描述了ATAM的阶段过程

架构权衡分析法ATAM，是一种系统架构评估方法，主要在系统开发之前，针对性能、可用性、安全性和可修改性等质量属性进行评价和折中，让架构师明确如何权衡多个质量目标，参与者有评估小组、项目决策者和其他项目相关人。

ATAM被分为四个主要的阶段,分别是场景和需求收集、体系结构视图和场景实现、属性模型构造和分析以及架构评审与折中。整个评估过程强调以属性(质量属性)作为架构评估的核心概念。

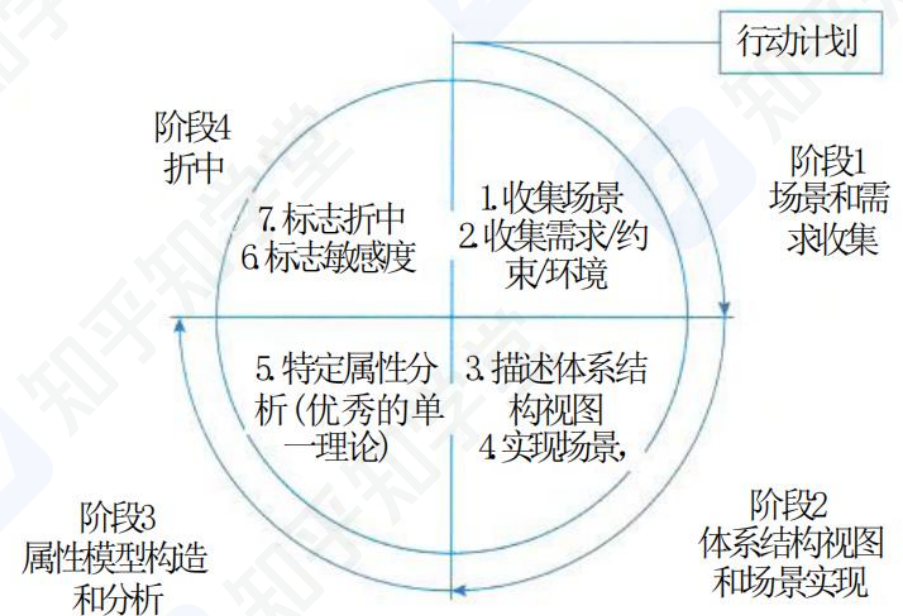
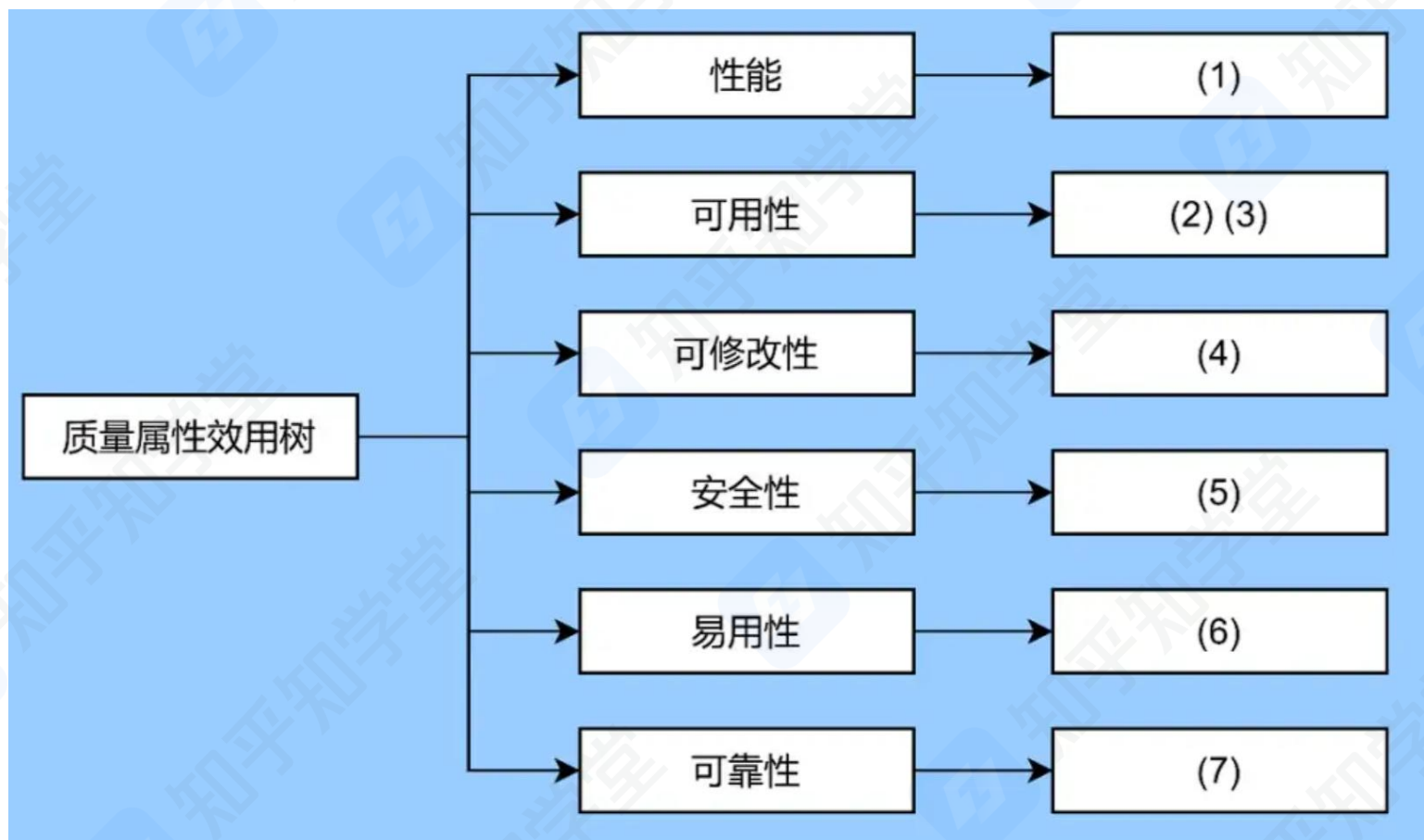
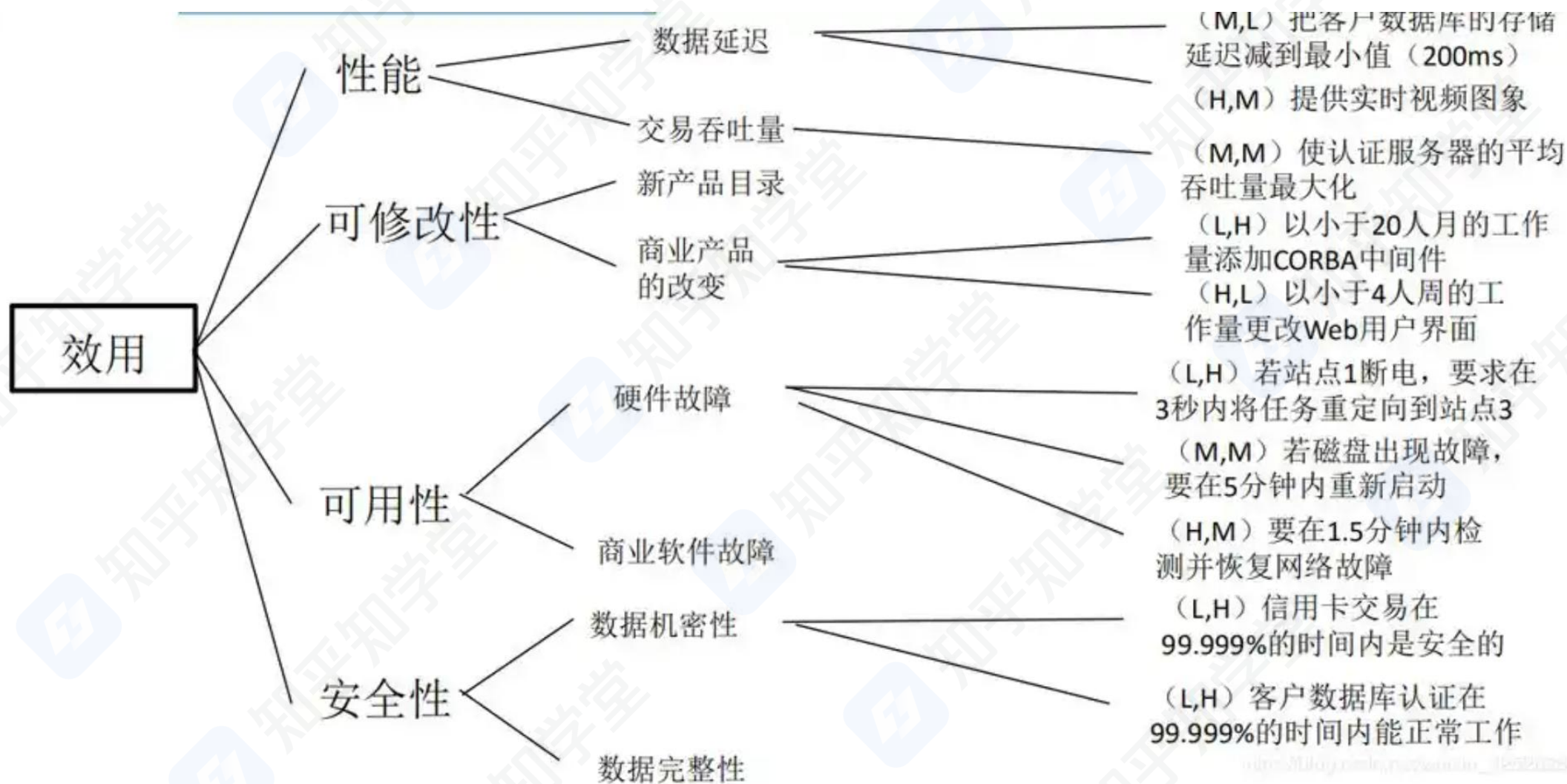


图8-2 ATAM分析评估过程

在书本的293页可以详细了解。





ATAM的阶段解释（见书本289-304页）：

1. 描述和介绍阶段：

- 目标： 此阶段的目标是定义评估的范围和目标，确定要评估的软件架构，明确要优化的质量属性以及介绍ATAM方法的步骤和原则。
- 活动： 在这个阶段，评估团队会与项目干系人一起定义评估的目标，确定评估的软件架构，并收集架构文档和相关信息。团队还会介绍ATAM方法的步骤，以确保所有参与者了解评估的过程。
- 示例： 对于电子商务网站的架构评估，评估团队会与项目干系人合作，确定评估的目标是提高性能和安全性。他们收集有关网站架构的文档，如架构图和设计文档。

2. 调查和分析阶段：

- 目标： 此阶段的目标是确定架构方法，分析架构并评估其对质量属性的影响，同时识别潜在的问题和权衡决策。
- 活动： 在这个阶段，评估团队会定义架构方法，分析不同的架构设计，生成质量属性效应树以表示不同决策对质量属性的影响。他们还会识别潜在的问题和决策权衡。
- 产出： 在此阶段，**产出质量属性效应树**（Quality Attribute Utility Tree），用于表示不同架构决策对质量属性的影响以及它们之间的权衡关系。
- 示例： 对于电子商务网站的架构评估，评估团队会定义不同的架构决策，如引入缓存或增加服务器资源。他们生成质量属性效应树，以分析这些决策对性能和安全性的影响。

3. 测试阶段:

- 目标: 测试阶段旨在验证架构是否满足质量属性需求, 以及在不同情况下的性能和行为。
- 活动: 团队创建测试用例来模拟质量属性场景, 包括性能测试、安全性测试等。他们运行这些测试用例, 测量系统的性能和行为, 并记录测试结果。在此阶段, 评估团队讨论各种质量属性场景, 对它们进行分级, 以确定哪些场景对系统的关键性最高。团队还会分析不同的架构方法, 以确定哪种方法最有可能满足关键场景的需求。最后, 项目干系人会对不同的架构方法和场景分级进行投票, 以帮助团队确定最佳的架构方案。
- 示例: 评估团队可能会讨论电子商务网站的性能、可伸缩性和安全性场景, 分级它们的重要性, 并分析引入缓存或增加服务器资源等不同架构方法的影响。项目干系人会进行投票, 以选择最合适的架构方法。

4. 报告阶段:

- 目标: 报告阶段的目标是总结评估的结果、提供改进建议, 并为决策者提供决策依据。
- 活动: 团队生成ATAM评估报告, 其中包括评估的发现、性能数据、可能的改进建议以及权衡决策。报告应该清晰地传达关键信息, 以便决策者可以做出明智的架构决策。
- 示例: 电子商务网站的评估报告可能包括性能测试结果、安全性评估、建议的架构改进, 以及与质量属性场景相关的权衡决策。报告将提供给项目管理团队, 以指导后续的架构决策和改进。

成本效益分析法CBAM: 用来对架构建立的成本来进行设计和建模，让决策者根据投资收益率来选择合适的架构，可以看做对ATAM的补充，在ATAM确定质量合理的基础上，再对效益进行分析。

有下列步骤:

- 整理场景(确定场景，并确定优先级，选择三分之一优先级最高的场景进行分析);
- 对场景进行细化(对每个场景详细分析，确定最好、最坏的情况);
- 确定场景的优先级(项目干系人对场景投票，根据投票结果确定优先级);
- 分配效用(对场景响应级别确定效用表，建立策略、场景、响应级别的表格);
- 形成“策略-场景-响应级别的对应关系”;
- 确定期望的质量属性响应级别的效用(根据效用表确定所对应的具体场景的效用表);
- 计算各架构策略的总收益;
- 根据受成本限制影响的投资报酬率选择架构策略(估算成本，用上一步的收益减去成本，得出收益，并选择收益最高的架构策略)。

50、质量属性效用树按优先级排序，沿用的两个维度是“每个场景对系统成功的重要性”以及（）

- A 从架构师角度来难的难易程度
- B 从架构师角度来难的效益
- C 从项目干系人来看的成本
- D 从干系人来看的重要性

294

系统架构设计师教程(第2版)

2) 质量属性效用树生成

效用树以“效用”作为根结点，质量属性构成效用树的辅助级别。这些属性位于表8-9中的第3列。在每个质量属性中都会包含特定的质量属性说明，以提供对方案更精确的描述。后者形成了实用程序树中的叶节点。效用树沿着两个维度进行优先顺序：**每个场景对系统成功的重要性**以及对此场景实现(从架构师的角度来看)所带来的难易程度的估计。效用树中的优先级排名为高(H)、中(M)和低(L)。

62、CBAM选择架构的核心依据是（）。

- A.项目干系人需求优先级
- B.投资回报 (ROI)
- C.项目干系人角度的实现成本
- D.系统架构师角度的实现难度



2026年上：综合知识



41、最早形成文档并得到广泛应用的软件架构评估方法是()

- A.SAAM
- B.ATAM
- C.CBAM
- D.ARID

58、关于软件架构评估的说法，错误的是()

- A.最终用户或其代表不应该参与主流架构评估活动
- B.敏感点一定只影响一个构件的特性
- C.系统购买者不一定需要参与架构评估
- D.架构评估关注质量属性场景与权衡分析

68、在软件质量属性评估中，下列通常不属于可用性度量指标的是()

- A.MTBF
- B.MTTR
- C.可用率
- D.故障发生的概率



T H E E N D

功不唐捐，玉汝于成！

• 开 启 新 征 程 •

