



NEW PLAN

软件高级架构师

· 一 段 新 征 程 ·





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
 - 软件架构复用
 - 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

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。**

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



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。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)



构件技术就是利用某种编程手段，将一些人们所关心的，但又不便于让最终用户去直接操作的细节进行了封装，同时对各种业务逻辑规则进行了实现，用于处理用户的内部操作细节。目前，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**
 - 最底层是对象请求代理ORB,规定了分布对象的定义(接口)和语言映射，实现对象间的通讯和互操作，是分布对象系统中的“软总线”；
 - 在ORB之上定义了很多公共服务，可以提供诸如并发服务、名字服务、事务(交易)服务、安全服务等各种各样的服务；
 - 最上层的公共设施则定义了组件框架，提供可直接为业务对象使用的服务，规定业务对象有效协作所需的协定规则。



构件



1. EJB (Enterprise JavaBeans)

现状：在Java生态中逐渐被轻量级框架（如Spring Boot）替代，但仍是J2EE规范的核心组件

新演进：

- 会话Bean → 演变为Spring的@ Bean和@ Scope注解
- 实体Bean → 被JPA（Hibernate等ORM框架）取代
- 消息驱动Bean → 整合到消息中间件（如Kafka、RabbitMQ）

2. COM/DCOM/COM+

现状：Windows生态中仍有遗留系统使用，但.NET Core跨平台架构成为微软新方向

3. CORBA

现状：金融、电信等传统行业仍有应用，但逐渐被服务网格（Service Mesh）替代

现代替代方案：

- gRPC（Google主导的高性能RPC框架，支持多语言）
- Istio/Envoy（服务间通信治理）



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)

【2022年】以下有关构件特性的描述中，说法不正确的是（）。

- A. 构件是独立部署单元
- B. 构件可作为第三方的组装单元
- C. 构件没有外部的可见状态
- D. 构件作为部署单元，是可拆分的

【2022年】在构件的定义中，（）是一个已命名的一组操作的集合。

- A. 接口
- B. 对象
- C. 函数
- D. 模块

【2021年】以下关于软件构件的叙述中，错误的是(35)

- A. 构件的部署必须能跟它所在的环境及其他构件完全分离
- B. 构件作为一个部署单元是不可拆分的
- C. 在一个特定进程中可能会存在多个特定构件的拷贝
- D. 对于不影响构件功能的某些属性可以对外部可见

【2021年】面向构件的编程目前缺乏完善的方法学支持，构件交互的复杂性带来了很多问题，其中(36)问题会产生数据竞争和死锁现象

- A. 多线程
- B. 异步
- C. 封装
- D. 多语言支持

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。

- 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：单线程控制，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。

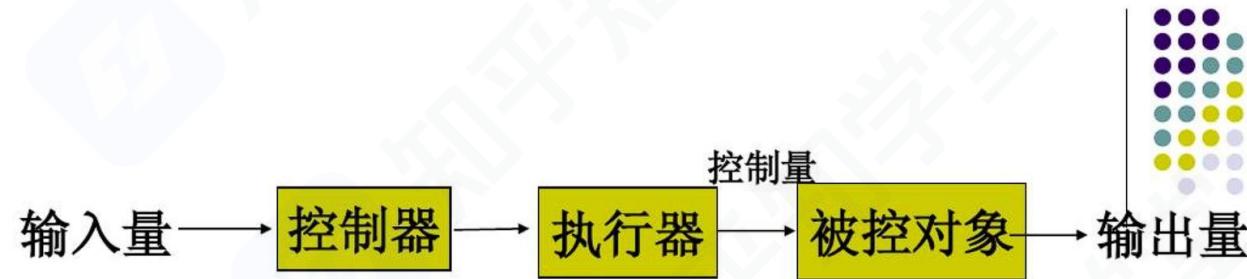
主要优点是为软件复用提供了强大的支持，为构件的维护和演化带来了方便；缺点是构件放弃了对系统计算的控制，只能被动的控制。

- 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 behavior。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。

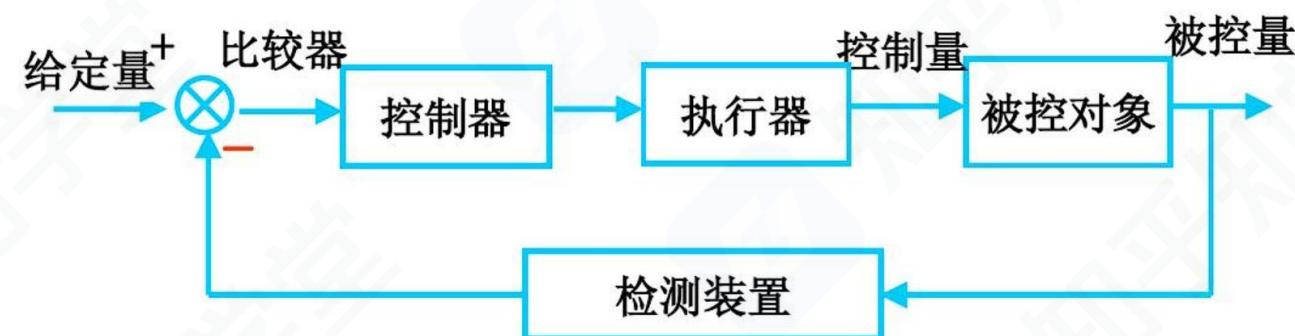
- 仓库风格的架构将数据存储在一个中央仓库或数据库中，各个组件可以从仓库中读取和写入数据。组件之间通过共享数据仓库进行通信和协作。
 - 示例说明：假设一个天气预报系统具有多个数据源，包括气象站、卫星数据和气象传感器。这些数据源将数据写入共享的中央仓库，然后天气预报应用程序可以定期查询仓库以获取最新的气象信息并生成天气预报。
- 黑板风格的架构类似于一个黑板或公告板，多个独立的组件称为"专家"共享一个公共存储区(黑板)，它们可以读取和写入数据。专家根据黑板上的信息进行推断和决策，适用于解决复杂的非结构化问题。
 - 示例说明：假设一个医学诊断系统包括放射科医生、心脏专家和内科医生。每个专家可以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

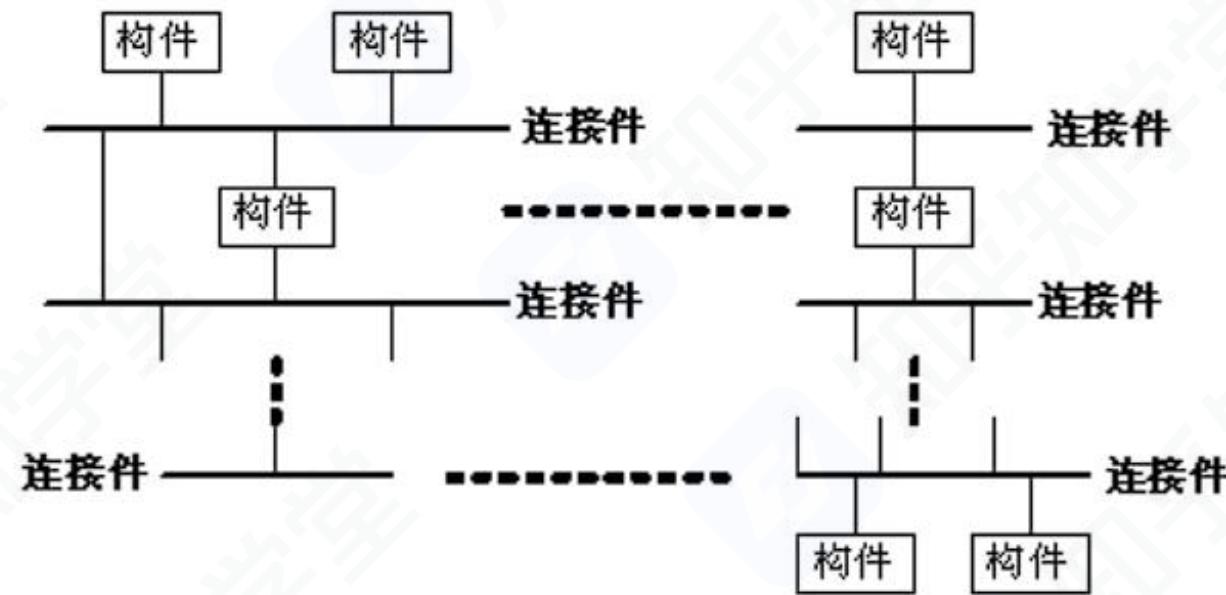


对于闭环控制系统，通常可以用以下的方框图表示：



➤ 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		显示调用, 主程序直接调用子程序。
4	调用/返回-面向对象	无	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。	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	机器人。	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		中央共享数据源, 独立处理单元。
11	仓库超文本	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	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

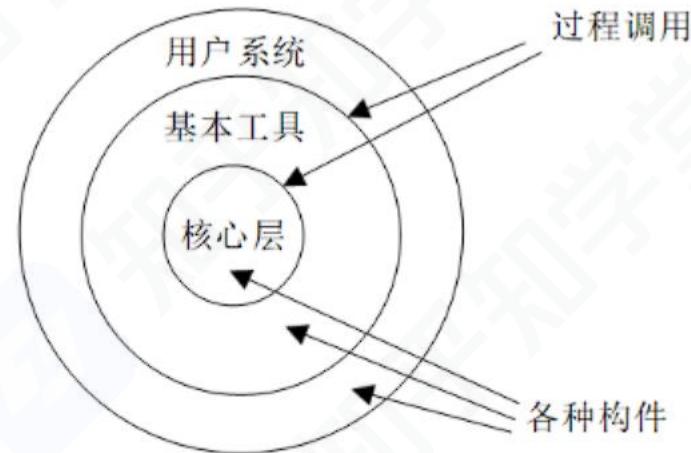
【2022年】48-49.软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式，其中，在批处理风格软件体系结构中，每个处理步骤是一个单独的程序，每一步必须在前一步结束后才能开始，并且数据必须是完整的，以()的方式传递。基于规则的系统包括规则集、规则解释器、规则/数据选择器及()。

- | | | | |
|--------|-------|--------|--------|
| A.迭代 | B.整体 | C.统一格式 | D.递增 |
| A.解释引擎 | B.虚拟机 | C.数据 | D.工作内存 |

【2021年】48.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，按照软件架构风格，物联网系统属于(48)软件架构风格。

- | | | | |
|-------|--------|-------|------|
| A.层次型 | B.事件系统 | C.数据线 | D.C2 |
|-------|--------|-------|------|

【2019】对软件体系结构风格的研究和实践促进了对设计的复用。Garlan和Shaw对经典体系结构风格进行了分类。其中，（ ）属于数据流体系结构风格；（ ）属于虚拟机体系结构风格；而下图描述的属于（ ）体系结构风格。



- | | | | |
|---------|---------|---------|--------|
| A. 面向对象 | B. 事件系统 | C. 规则系统 | D. 批处理 |
| A. 面向对象 | B. 事件系统 | C. 规则系统 | D. 批处理 |
| A. 层次型 | B. 事件系统 | C. 规则系统 | D. 批处理 |



两层C/S架构：客户端和服务器都有处理功能，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架构中的数据从服务器中独立出来了。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可控制。

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。

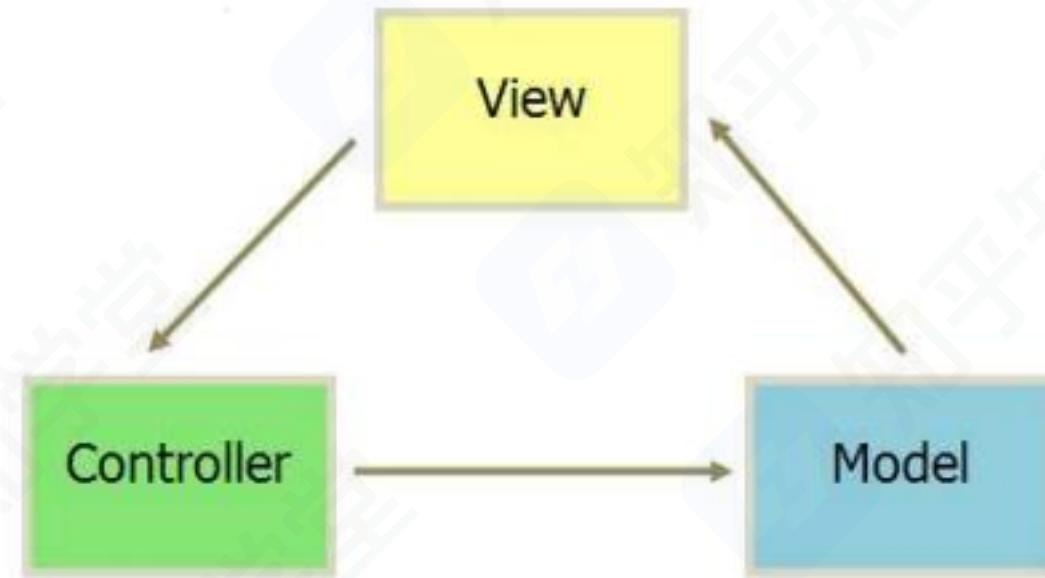
富互联网应用RIA: 弥补三层B/S架构存在的问题，RIA是一种用户接口，比用HTML实现的接口更加健壮，且有可视化内容，本质还是网站模式，其优点如下：

- RIA结合了C/S架构反应速度快、交互性强的优点与B/S架构传播范围广及容易传播的特性；
- RIA简化并改进了B/S架构的用户交互；
- 数据能够被缓存在客户端，从而可以实现一个比基于HTML的响应速度更快且数据往返于服务器的次数更少的用户界面。
- 本质还是O客户端，借助于高速网速实现必要插件在本地的快速缓存，增强页面对动态页面的支持能力，典型的如小程序。

控制器(Controller)：是应用程序中处理用户交互的部分。通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

模型(Model)：是应用程序中用于处理应用程序数据逻辑的部分。通常模型对象负责在数据库中存取数据。模型表示业务数据和业务逻辑。

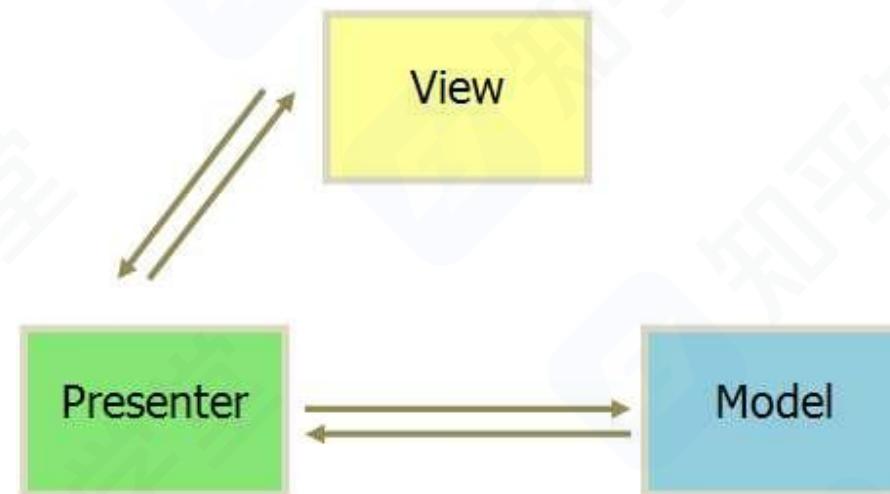
视图(View)：是应用程序中处理数据显示的部分。通常视图是依据模型数据创建的。是用户看到并与之交互的界面。视图向用户显示相关的数据，并能接收用户的输入数据，但是它并不进行任何实际的业务处理：



MVP是把MVC中的Controller换成了Presenter(呈现)，目的就是为了完全切断View跟Model之间的联系，由Presenter充当桥梁，做到View-Model之间通信的完全隔离。

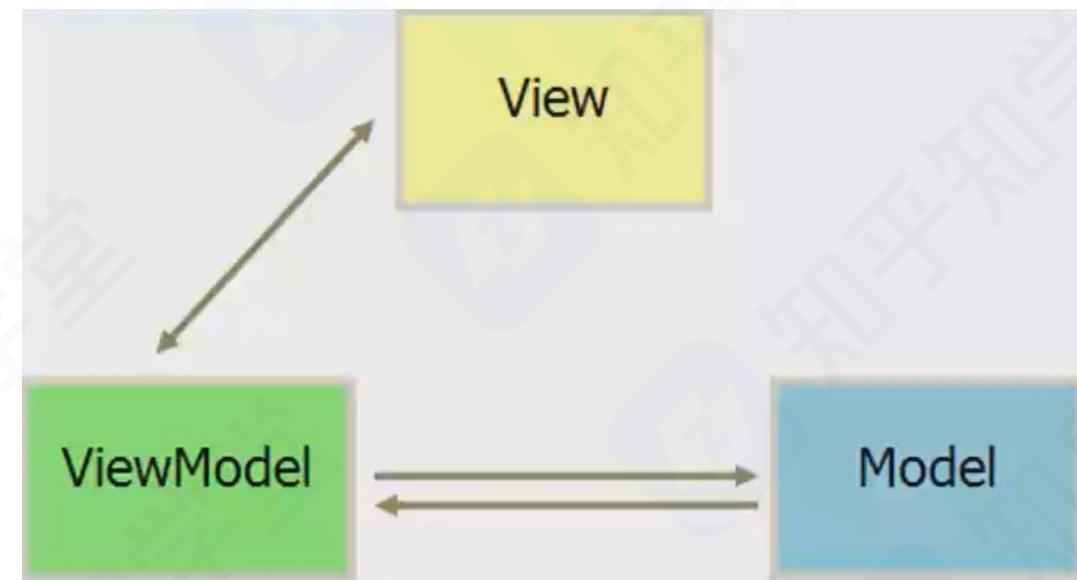
MVP特点：

- M、V、P之间双向通信。
- View与Model不通信，都通过Presenter传递。Presenter完全把Model和View进行了分离，主要的程序逻辑在Presenter里实现。
- View非常薄，不部署任何业务逻辑，称为”被动视图”(PassiveView)，即没有任何主动性，而Presenter非常厚，所有逻辑都部署在那里。
- Presenter与具体的View是没有直接关联的，而是通过定义好的接口进行交互，从而使得在变更View时候可以保持Presenter的不变，这样就可以重用。



MVVM: MVVM (Model-View-ViewModel) 是一种软件架构模式，旨在分离视图 (View) 和模型 (Model)，并通过View和ViewModel的**双向绑定机制**自动同步数据实现高效开发。它的核心思想是将界面逻辑与业务逻辑解耦，提升代码的可维护性和可测试性：

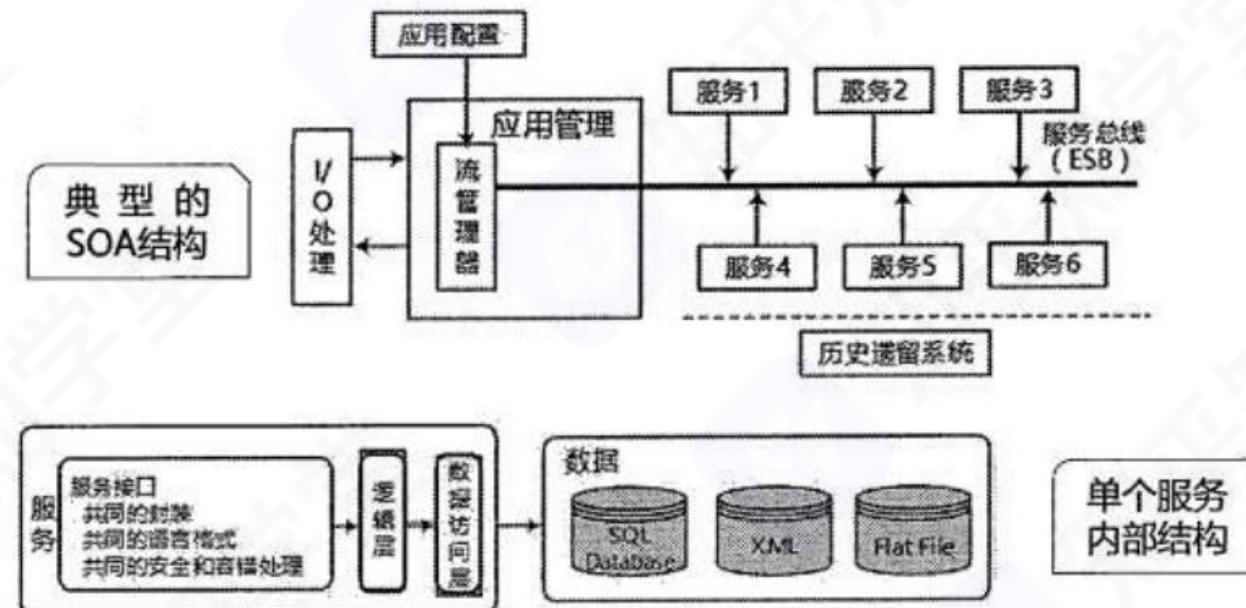
- 低耦合，视图(View)可以独立于Model变化和修改，一个ViewModel可以绑定到不同的”View”上，当View变化的时候 Model可以不变，当Model变化的时候View也可以不变。
- 可重用性，可以把一些视图逻辑放在一个ViewModel里面，让很多view重用这段视图逻辑。
- 独立开发，开发人员可以专注于业务逻辑和数据的开发(ViewModel)，设计人员可以专注于页面设计。
- 可测试，界面向来是比较难于测试的，而现在测试可以针对ViewModel来写。



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。

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：



SOA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规范，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。
- 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。
- 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。
- XML (可扩展标记语言)：是WebService平台中表示数据的基本格式，用于数据交换。

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



面向服务的架构风格



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

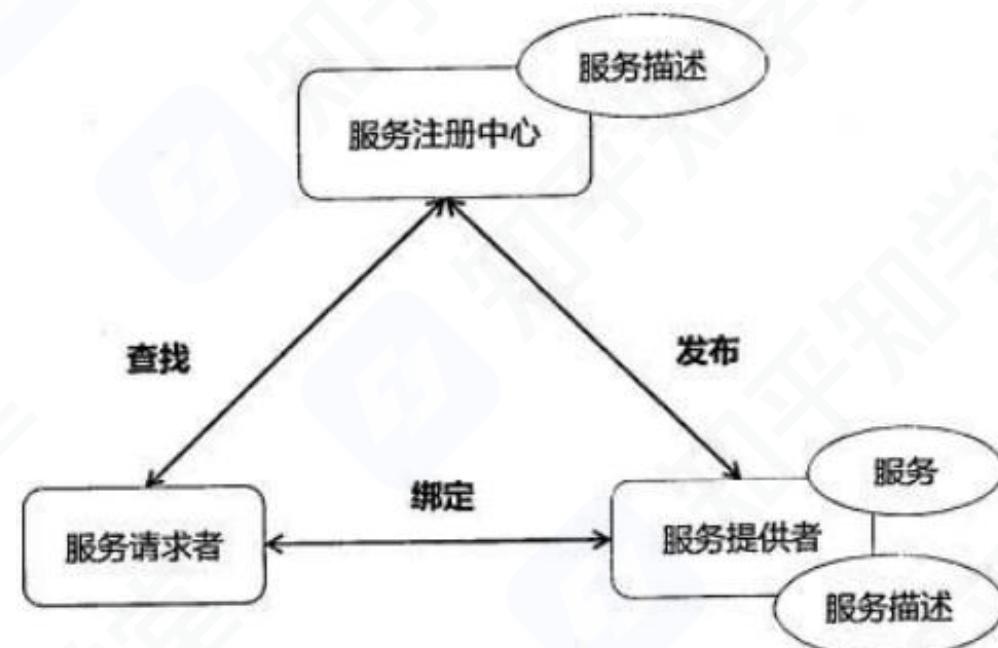
WEB Service: WebService是一种网络服务，它使不同系统之间能够通过网络进行通信和数据交换。WebService基于标准的网络协议（如HTTP和HTTPS），并使用XML、JSON等数据格式，使系统能够跨平台、跨语言通信。WebService可以被视为SOA实现的核心技术之一，它通过服务接口使不同系统之间的数据交换变得更加容易。

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。

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。

在实际的WebService与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。

服务注册表：是SOA架构中的服务目录系统，相当于分布式系统的"服务黄页"。它的核心功能包括：

作用维度	具体功能与实现方式	技术参考
服务注册与发布	服务提供者将服务接口（如WSDL）和访问端点（Endpoint）注册到中心目录，支持UDDI标准协议	UDDI、Consul、Nacos
服务发现与查询	消费者通过关键词（服务名、版本号等）搜索可用服务，获取服务地址和调用方式	REST API查询接口、SOAP请求
版本与依赖管理	支持同一服务的多版本共存（如v1.0和v2.0），管理服务间的依赖关系	语义化版本控制（SemVer）
健康监控与治理	定期检查服务可用性（如心跳检测），自动剔除故障节点，保障服务可靠性	心跳机制、健康检查API
访问控制与安全	管理服务访问权限（如基于角色的访问控制），记录服务调用日志用于审计	OAuth 2.0、RBAC策略

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。



软件产品线：是指一组软件密集型系统，它们共享一个公共的、可管理的特性集，满足某个特定市场或任务的具体需要，是以规定的方式用公共的核心资产集成开发出来的。即围绕核心资产库进行管理、复用、集成新的系统。

- 举例：假设一家汽车导航系统制造公司想要开发多款汽车导航产品，以满足不同市场和客户的需求。他们可以采用软件产品线方法：
- 核心功能：公司首先开发一个通用的核心导航引擎，包括地图数据处理、路线规划、导航指令生成等功能。然后，他们可以基于这个核心引擎，创建不同变种的导航产品。例如：
 - 汽车导航器A：针对高端汽车市场，具有高级语音识别、实时交通信息和豪华界面。
 - 汽车导航器B：针对经济型车型，功能较少，但价格更实惠。
 - 卡车导航系统：针对卡车司机，包括特殊的货物规划和路线优化功能。
 - 定制特性：每个产品变种可以根据特定客户的需求进行定制，添加或移除特定功能或界面元素。

软件架构复用根据复用的时机包括**机会复用**和**系统复用**。机会复用是指开发过程中，只要发现有可复用的资产，就对其进行复用。系统复用是指在开发之前，就要进行规划，以决定哪些需要复用。

可复用的资产包括：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、缺陷消除。
。

复用的基本过程主要包括3个阶段：首先构造/获取可复用的软件资产，其次管理这些资产(把它们放入到构件库)，最后针对特定的需求，从这些资产中选择可复用的部分，以开发满足需求的应用系统。

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。

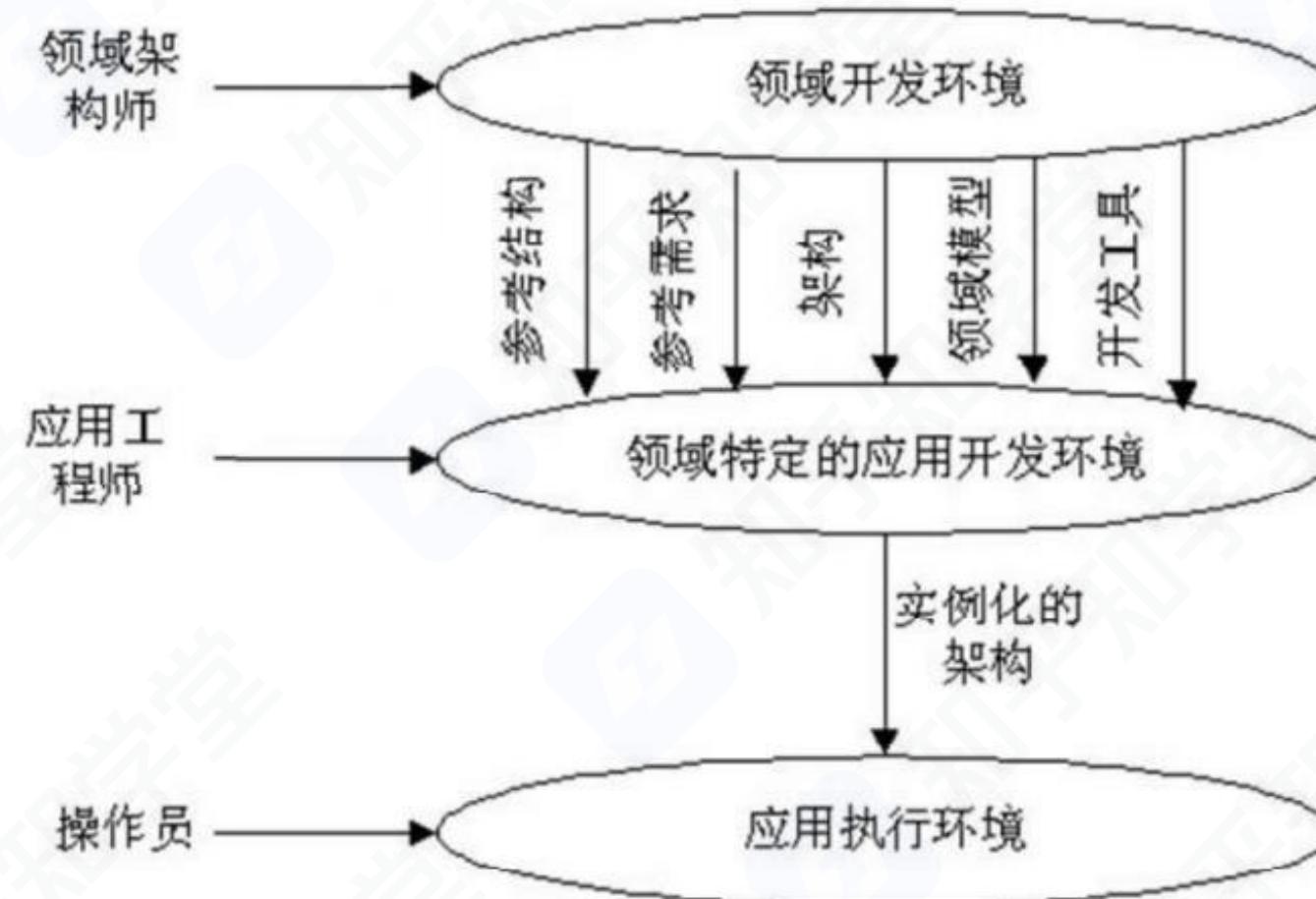
参与DSSA的四种角色人员：领域专家、领域分析人员、领域设计人员和领域实现人员

- 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有经验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的根本架构，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任务。这个层次关注的是系统的实际运行和操作。

DSSA是在一个特定应用领域中为一组应用提供组织结构参考的软件体系结构，参与DSSA的人员可以划分为4种角色，包括领域专家、领域设计人员、领域实现人员和()，其基本活动包括领域分析、领域设计和()。

- | | | | |
|-----------|---------|----------|---------|
| A. 领域测试人员 | B. 领域顾问 | C. 领域分析师 | D. 领域经理 |
| A. 领域建模 | B. 架构设计 | C. 领域实现 | D. 领域评估 |

特定领域软件架构(Domain Specific Software Architecture, DSSA)以一个特定问题领域为对象，形成由领域参考模型，参考需求，()等组成的开发基础架构，支持一个特定领域中多个应用的生成。DSSA的基本活动包括领域分析、领域设计和领域实现。其中领域分析的主要目的是获得()，从而描述领域中系统之间共同的需求，即领域需求；领域设计的主要目标是获得()，从而描述领域模型中表示需求的解决方案；领域实现的主要目标是开发和组织可重用信息，并实现基础软件架构。

- | | | | |
|-------------|-------------|---------------|---------------|
| A. 参考设计 | B. 参考规约 | C. 参考架构 | D. 参考实现 |
| A. 领域边界 | B. 领域信息 | C. 领域对象 | D. 领域模型 |
| A. 特点领域软件需求 | B. 特定领域软件架构 | C. 特定领域软件设计模型 | D. 特定领域软件重用模型 |

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)。

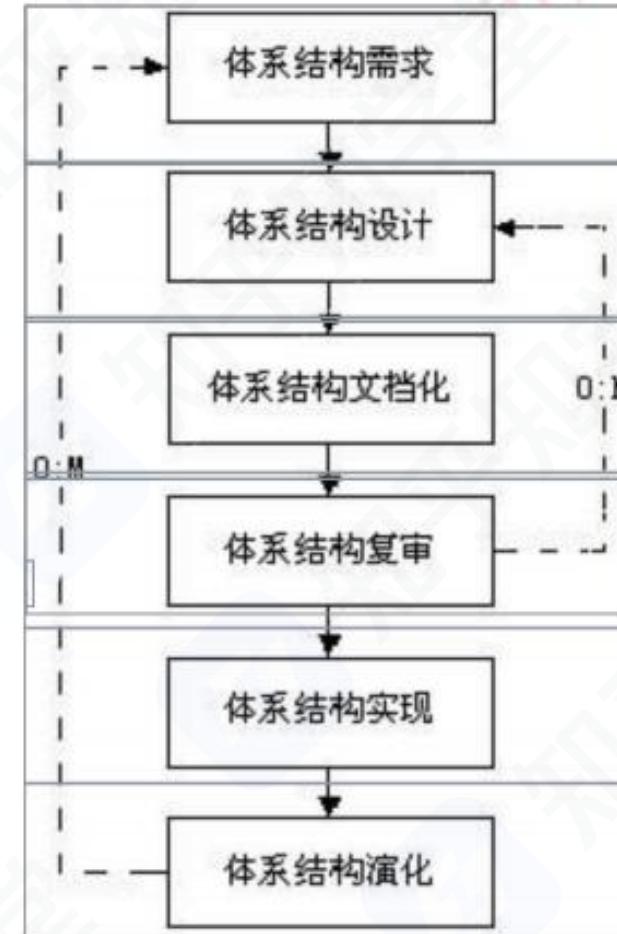
使用ABSD方法，设计活动可以从项目总体功能框架明确就开始，这意味着需求获取和分析还没有完成，就开始了软件设计。

ABSD方法有三个基础：

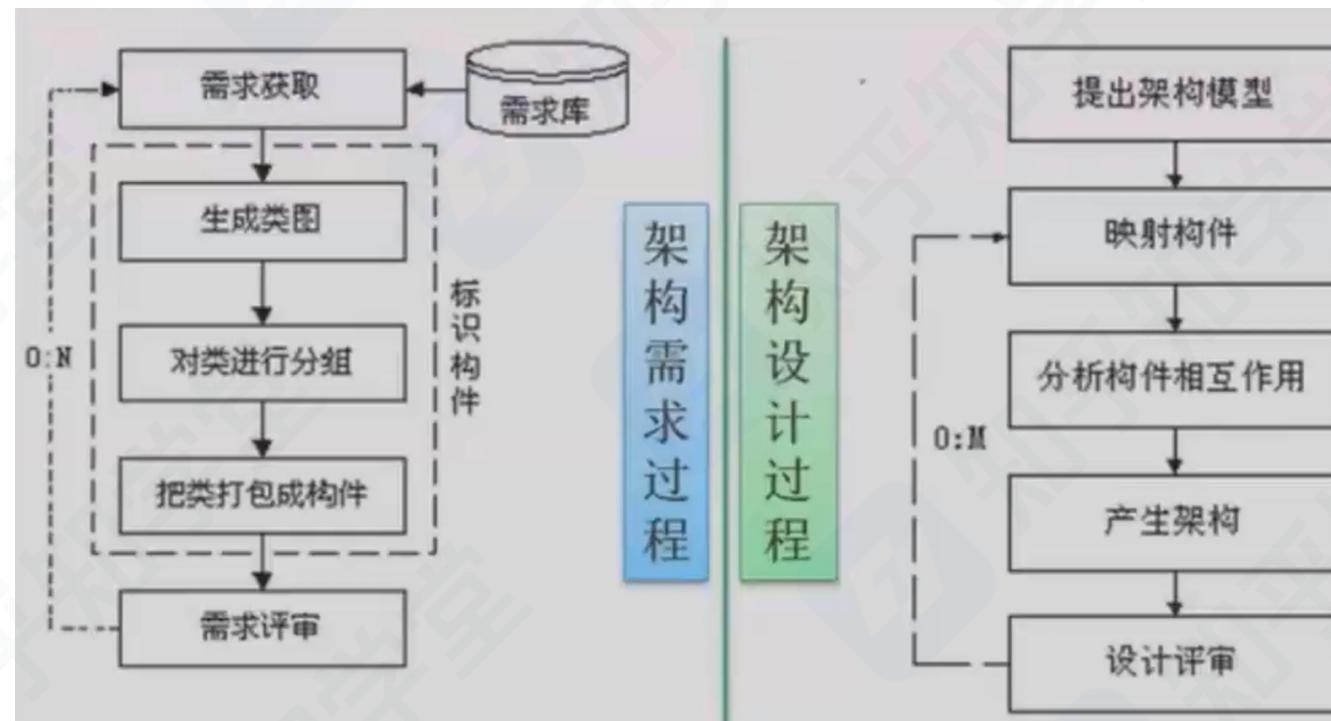
- 第一个基础：是**功能的分解**，使用已有的基于模块的内聚和耦合技术
- 第二个基础：是通过**选择架构风格**来实现质量和业务需求
- 第三个基础：是**软件模板的使用**，软件模板利用了一些软件系统的结构进行复用。

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。

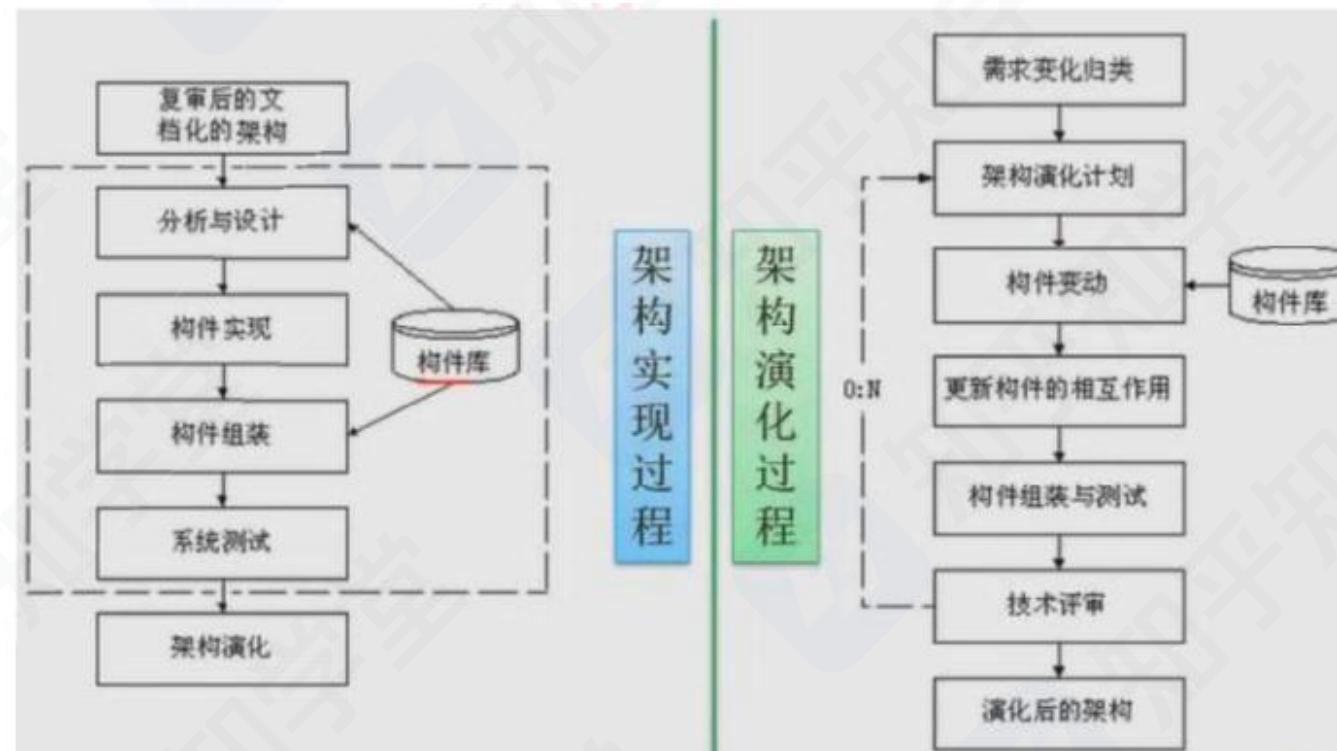
基于架构的软件开发过程可分为下列六步：



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：



在基于体系结构的软件设计方法中，采用()来描述软件架构，采用()来描述功能需求，采用()来描述质量需求。

- | | | | |
|----------|---------|---------|---------|
| A.类图和序列图 | B.视角与视图 | C.构件和类图 | D.构件与功能 |
| B.类图 | B.视角 | C.用例 | D.质量场景 |
| C.连接件 | B.用例 | C.质量场景 | D.质量属性 |

体系结构文档化有助于辅助系统分析人员和程序员去实现体系结构。体系结构文档化过程的主要输出包括()。

- | |
|---------------------------|
| A.结构规格说明、测试体系结构需求的质量设计说明书 |
| B.属性说明书、体系结构描述 |
| C.结构规格说明、软件功能需求说明 |
| D.图体系结构模型、体系结构验证说明 |



02

质量属性和架构评估



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，而可用性强调系统在需要时可供使用。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)：该刺激在某些条件下发生。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。

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。

风险点与非风险点不是以标准专业术语形式出现的，只是一个常规概念，即可能引起风险的因素，可称为风险点。某个做法如果有隐患，有可能导致一些问题，则为风险点；而如果某件事是可行的可接受的，则为非风险点。

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。

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

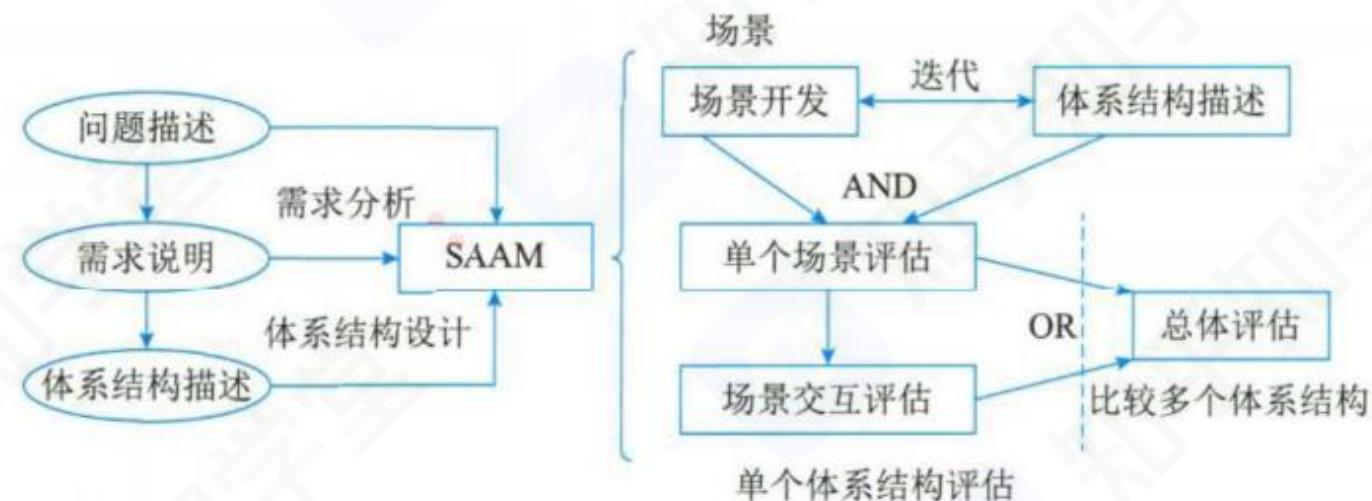
- 举例：对于一个大型数据库管理系统的架构评估，评估团队可以度量数据库查询的平均响应时间、事务处理的吞吐量、数据库表的大小等指标。这些度量可以用于评估系统的性能和可伸缩性。

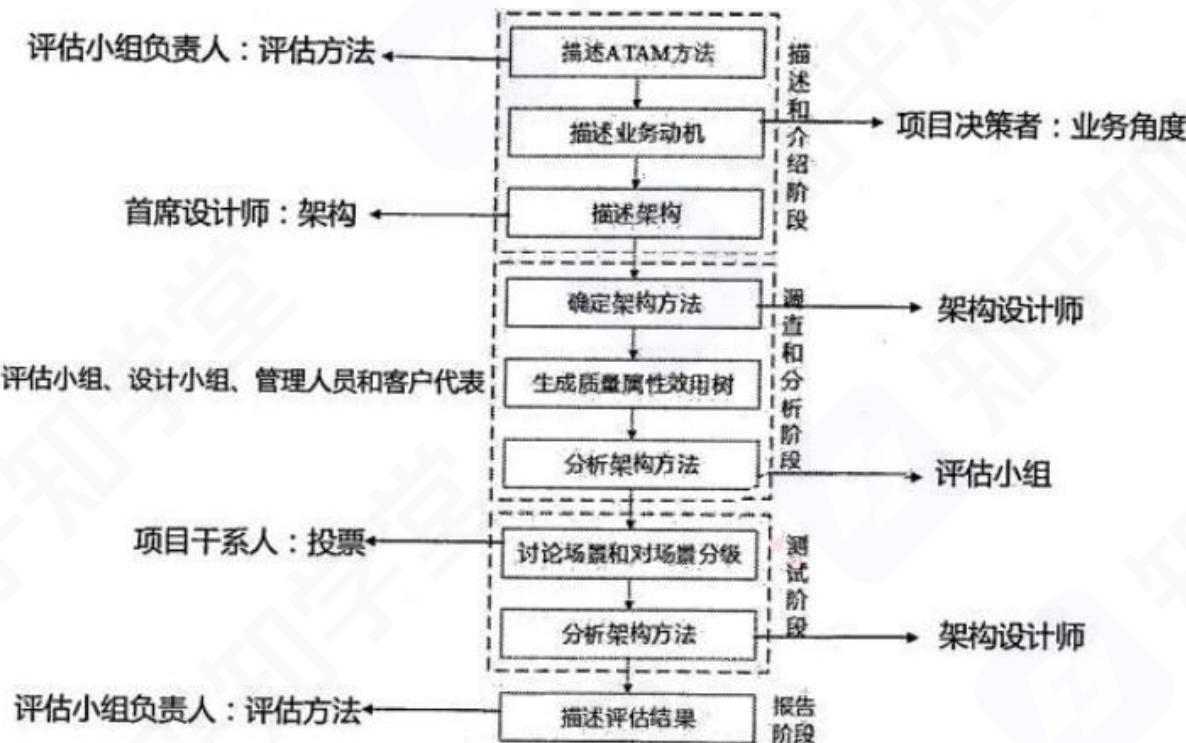
3. 基于场景的方式：主要方法。这是一种系统性的评估方法，涉及定义一系列场景或情境，以测试系统在各种条件下的表现，每个场景描述了一种特定的使用情况，包括输入、预期输出和性能指标。评估人员会模拟这些场景，并评估系统是否满足质量属性需求。

- 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。

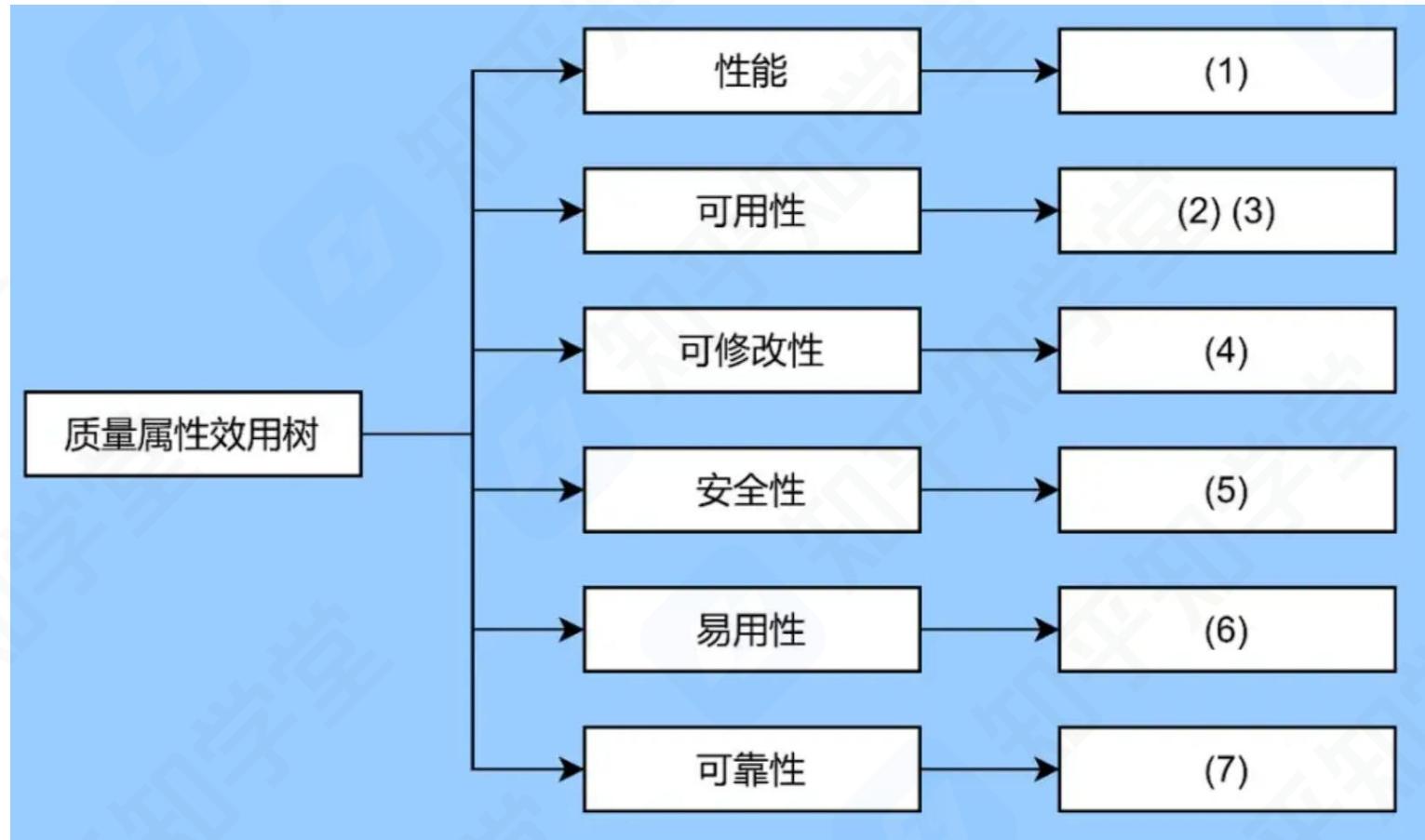




架构权衡分析法ATAM，是一种系统架构评估方法，主要在系统开发之前，针对性能、可用性、安全性和可修改性等质量属性进行评价和折中，让架构师明确如何权衡多个质量目标，参与者有评估小组、项目决策者和其他项目相关人。

ATAM被分为四个主要的活动领域,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。主要针对性能、可用性、安全性和可修改性，在系统开发之前，对这些质量属性进行评价和折中。

在书本的293页有图可以了解。



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)。

某公司欲开发一个在线交易网站，在架构设计阶段，公司的架构师识别出3个核心质量属性场景。其中"网站正常运行时，用户发起的交易请求应该在3秒内完成"主要与(58)质量属性相关，通常可采用(59)架构策略实现该属性："在线交易主站宕机后，能够在3秒内自动切换至备用站点并恢复正常运行"主要与(60)质量属性相关，通常可采用(61)架构策略实现该属性："系统应该具备一定的安全保护措施，从而能够抵挡恶意的入侵破坏行为，并对所有针对网站的攻击行为进行报警和记录"主要与(62)质量属性相关，通常可采用(63)架构策略实现该属性。

- | | | | |
|-------------|-------------|--------|----------|
| (58)A.可用性 | B.性能 | C.易用性 | D.可修改性 |
| (59)A.抽象接口 | B.信息隐藏 | C.主动冗余 | D.资源调度 |
| (60)A.可测试性 | B.易用性 | C.可用性 | D.互操作性 |
| (61)A.记录/回放 | B.操作串行化 | C.心跳 | D.增加计算资源 |
| (62)A.可用性 | B.安全性 | C.可测试性 | D.可修改性 |
| (63)A.追踪审计 | B.Ping/Echo | C.选举 | D.维护现有接口 |



03

软件可靠性

- 本章节在历年考试过程中的分值占比大概是1-2分
- 本章节在新版教材里有一个单独的章节，第九章，也就是从改版之后才开始出现考试题目，基本上考的就是些概念以及常识，没有太多需要重点关注的地方，整体过一遍，记住一些名词和一些简单计算即可
- 被考到知识点有：
 - 软件可靠性基本概念
 - 软件可靠性建模、管理
 - 软件可靠性设计、测试、评价
- 在改版之后的考试中，分别考察的知识点为：
 - 2023年11月：可靠性计算MTTF和MTBF
 - 2024年05月：MTTD平均故障检测时间
 - 2024年11月：N版本程序设计（超纲）

软件可靠性：是软件产品在规定的条件下和规定的时间区间完成规定功能的能力。

软件可靠性和硬件可靠性区别

- 复杂性：软件复杂性比硬件高，大部分失效来自于软件失效。
- 物理退化：硬件失效主要是物理退化所致，软件不存在物理退化。
- 唯一性：软件是唯一的，软件复制不改变软件本身，而任何两个硬件不可能绝对相同。
- 版本更新周期：硬件更新较慢，软件更新较快。

软件可靠性的定量描述：

- 规定时间：自然时间、运行时间、执行时间(占用CPU)。
- 失效概率：软件运行初始时为0,随着时间增加单调递增，不断趋向于1.
- 可靠度：软件系统在规定的条件下、规定的时间内不发生失效的概率。等于1-失效概率。
- 失效强度：单位时间软件系统出现失效的概率。
- 平均失效前时间(MTTF)：平均无故障时间，发生故障前正常运行的时间。
- 平均恢复前时间(MTTR)：平均故障修复时间，发生故障后的修复时间。
- 平均故障间隔时间(MTBF)：失效或维护中所需的平均时间，包括故障时间以及检测和维护设备的时间。
$$MTBF = MTTF + MTTR$$

广义的软件可靠性测试：是指为了最终评价软件系统的可靠性而运用建模、统计、试验、分析和评价等一系列手段对软件系统实施的一种测试。它是针对整个软件的各个方面进行的测试

狭义的软件可靠性测试：是指为了获取可靠性数据，按预先确定的测试用例，在软件的预期使用环境中，对软件实施的一种测试。它是面向缺陷的测试，以用户将要使用的方式来测试软件。它是针对某个功能来设计测试用例的测试

可靠性测试的意义

- 软件失效可能造成灾难性的后果。
- 软件的失效在整个计算机系统失效中的比例较高。
- 软件可靠性技术很不成熟，加剧了软件可靠性问题的重要性。
- 软件可靠性问题是造成费用增长的主要原因之一。
- 软件对生产活动和社会生活的影响越来越大，从而增加了软件可靠性问题在软件工程领域乃至整个计算机工程领域的的重要性。



软件可靠性模型是指：为预计或估算软件的可靠性所建立的可靠性框图和数学模型。即：为可靠性来建立一个模型，方便我们对其进行分析和测试

从技术的角度来看，影响软件可靠性的主要因素包括：

- **运行环境**：软件可靠性受到运行环境的直接影响。例如，一个设计用于Linux操作系统的软件，在Windows操作系统上可能会遇到兼容性问题，导致不可靠性。
- **软件规模**：软件的规模越大，通常越容易引入错误和缺陷，从而降低可靠性。一个操作系统内核的开发相对于一个简单的文本编辑器来说，通常更具挑战性，因为规模更大、复杂性更高。
- **软件内部结构**：软件的设计和架构对可靠性至关重要。合理的软件架构和清晰的模块化设计有助于减少错误的传播和提高维护性。例如，一个精心设计的分层架构的应用程序可能比一个将所有逻辑放在一个庞大函数中的应用程序更可靠。
- **软件的开发方法和开发环境**：开发方法和工具对软件的可靠性有重大影响。采用现代的开发方法，如敏捷开发或持续集成，可以帮助团队更早地发现和解决问题。使用自动化测试工具和代码审查流程也可以提高可靠性。例如，在敏捷开发团队中，频繁的测试和反馈往往有助于提高软件的质量和可靠性。
- **软件的可靠性投入**：这包括在测试、质量控制和质量保证方面投入的资源。投入更多的时间和人力资源来执行全面的测试、代码审查和性能优化通常会提高软件的可靠性。例如，一个投入了大量测试和质量控制工作的开发团队可能会开发出比一个没有经过充分测试的团队更可靠的软件。

一个软件可靠性模型通常(但不是绝对)由以下几部分组成:

- **模型假设**。模型是实际情况的简化或规范化，总要包含若干假设，例如假设用户在购物网站上的行为代表了实际用户的典型行为，包括浏览商品、添加商品到购物车、结账等操作。
- **性能度量**。软件可靠性模型的输出量就是性能度量，如失效强度、残留缺陷数等。失效强度是单位时间内出现问题或错误的平均数量。例如，每小时有多少用户在结账时遇到错误。
- **参数估计方法**。某些可靠性度量的实际值无法直接获得，例如失效强度，可以通过实时监测用户操作并记录错误来估计。例如，每小时记录有多少用户在结账时遇到错误，然后计算失效强度。
- **数据要求**。一个软件可靠性模型要求一定的输入数据，即软件可靠性数据。例如为了进行失效强度的估计，需要实时监测用户操作并记录错误。这可能需要部署监测工具或日志记录系统

软件可靠性管理：为了进一步提高软件的可靠性，人们提出要把**软件可靠性活动贯穿整个软件开发的全过程**

为什么需要可靠性设计：实践证明，保障软件可靠性最有效、最经济、最重要的手段是在**软件设计阶段**采取措施进行可靠性控制。为了从根本上提高软件的可靠性，人们就提出了可靠性设计

可靠性设计其实就是在**常规的软件设计**中，应用各种方法和技术，使程序设计在兼顾用户的功能和性能需求的同时，**全面满足软件的可靠性要求**。

软件可靠性设计原则：

- 软件可靠性设计是**软件设计**的一部分，必须在软件的总体设计框架中使用，并且不能与其他设计原则相冲突。
- 软件可靠性设计在满足提高软件质量要求的前提下，以**提高和保障软件可靠性为最终目标**。
- 软件可靠性设计应**确定软件的可靠性目标**，不能无限扩大化，并且排在功能度、用户需求和开发费用之后考虑。

软件可靠性设计技术主要有**容错设计、检错设计和降低复杂度设计**等技术。

容错：指系统在运行过程中发生一定的硬件故障或软件错误时，仍能保持正常工作而不影响正确结果的一种性能或措施。

容错技术主要有：恢复块设计、N版本程序设计和冗余设计这三种。

- 冗余是指在正常系统运行的基础上加上一定数量的资源，包括信息、时间、硬件和软件，在正常设备或元件出现问题或故障时，可以立即更换冗余的设备或元件，从而保证系统的正常运行。**冗余是容错技术的基础**，通过冗余资源的加入，可以使系统的可靠性得到较大的提高。
- 恢复块设计(动态冗余)：动态冗余又称为主动冗余，是通过在软件中引入恢复块来实现容错。这些恢复块通常是代码块或子程序，用于检测和处理错误情况，以确保系统可以从错误中恢复并继续正常运行。
- N版本程序设计：其设计思想是用**N个具有相同功能的程序同时执行一项计算**，结果通过**多数表决来选择**。是一种通过创建多个相互独立的软件版本来提高可靠性的方法。这些不同版本的软件在相同的输入下执行相同的任务，然后结果进行比对，如果其中一个版本的输出与其他版本不一致，则选择正确的输出。

除此之外，跟容错有关的技术还有：双机容错技术、防卫式程序设计、集群技术，集群技术中又涉及到了负载均衡技术，对于这些技术是有必要做一些基本的了解的，

软件可靠性评价3个过程：选择可靠性模型、收集可靠性数据、可靠性评估和预测。

- 选择可靠性模型：需要选择适合您的软件项目的可靠性模型，这个模型将帮助您评估软件的可靠性。不同的项目可能需要不同的模型，以考虑其特定需求和特点。
- 示例：假设您正在开发一款医院管理系统，您希望评估该系统的可靠性。您可以选择使用“可靠性块图”模型，这是一种常用于评估系统可靠性的模型。然后，您根据系统的结构和功能，创建可靠性块图，标识关键组件和它们之间的依赖关系，以便量化系统的可靠性。
- 收集可靠性数据：一旦选择了适当的可靠性模型，接下来的步骤是收集相关的可靠性数据。这些数据可以包括组件的故障率、失效模式和效应分析（FMEA）等信息，以支持可靠性评估。
- 示例：对于医院管理系统，您可以开始收集以下数据：每个系统组件的故障率和平均失效时间（MTTF）、每个组件的失效模式，例如硬件故障、软件错误等。每个失效模式的严重性评估，以确定其对系统可靠性的影响。
- 可靠性评估和预测：使用所选的可靠性模型和收集的可靠性数据来评估软件的可靠性，并进行可靠性预测。这将帮助您了解软件在实际使用中的可靠性表现，以及是否需要采取进一步的改进措施。
- 示例：使用可靠性块图模型和收集的可靠性数据，您可以评估医院管理系统的整体可靠性。根据组件的故障率和失效模式，您可以预测系统在一定时间内的可用性和可靠性水平。如果评估结果显示某些组件存在潜在的可靠性问题，您可以采取措施来改进这些组件或提供备用方案，以提高系统的可靠性。



04

软件架构演化与维护



- 本章节在历年考试过程中的分值占比大概是0-1分
- 本章节是改版之后新增的内容，之前从没有考过，在这里更像是让我们对软件架构演化通过图例的形式来进行更深的了解，所以老师的建议是把概念过一遍，但是架构演化实例这里还是要好好的理解，方便我们写论文和某些案例分析的简答题部分使用
- 被考到知识点有：
 - 软件架构演化、面向对象架构演化
 - 软件架构演化分类、原则
 - 大型网站架构演化实例
 - 软件架构维护
- 在改版之后的考试中，分别考察的知识点为：
 - 2023年11月：无
 - 2024年05月：架构演化的概念
 - 2024年11月：无



软件架构的演化和维护就是对架构进行修改和完善的过程，目的就是为了使软件能够适应环境的变化而进行的纠错性修改和完善性修改等，是一个不断迭代的过程，直至满足用户需求。

举例：有一家电子商务公司，他们的在线购物平台拥有数百万用户。初始时，他们的平台采用单一的单体架构，所有功能都集成在一个应用程序中。但随着时间推移，业务发展，用户数量增加，公司开始遇到一些问题：

- 性能问题：由于用户数量的增加，平台的性能开始下降，响应时间变得较长。
- 扩展性问题：随着新功能的添加，开发团队发现很难扩展和维护整个应用程序。
- 部署问题：每次发布新版本时，整个应用程序都需要重新部署，导致停机时间和风险增加。

为了解决这些问题，公司决定对软件架构进行演化和维护。他们采取了以下措施：

- 微服务架构：他们将应用程序拆分成多个小型服务，每个服务负责一个特定的功能，如用户管理、购物车、支付等。这使得每个服务可以独立扩展和部署，提高了系统的灵活性和性能。
- 容器化：他们引入了容器技术，如Docker，以便更轻松地管理和部署服务。这减少了部署的停机时间，并提高了可靠性。
- 自动化部署：公司建立了自动化部署管道，以便快速发布新功能和修复bug，而无需手动干预。

通过这些改进，公司成功地演化和维护了他们的软件架构，解决了之前的问题，并为未来的增长和变化做好了准备。这个例子说明了软件架构的演化和维护是一个持续的过程，旨在适应变化的环境和满足用户需求。



面向对象软件架构演化主要分为四种演化：**对象演化、消息演化、复合片段演化和约束演化**

软件架构演化的3种分类方法：

- 按照**软件架构的实现方式和实施粒度**分类：基于过程和函数的演化、面向对象的演化、基于组件的演化和基于架构的演化。
- 按照**研究方法**将软件架构演化方式分为**4类**：
 - 第1类：对演化的支持，如代码模块化的准则、可维护性的指示(如内聚和耦合)、代码重构等，
 - 第2类：版本和工程的管理工具；
 - 第3类：架构变换的形式方法，包括系统结构和行为变换的模型，以及架构演化的重现风格等；
 - 第4类：架构演化的成本收益分析，决定如何增加系统的弹性。
- 针对软件架构的**演化过程是否处于系统运行时期**，可以将软件架构演化分为**静态演化**和**动态演化**。

软件架构的演化时期包括：**设计时演化、运行前演化、有限制运行时演化、运行时演化**。

根据演化过程是否已知可将评估过程分为：演化过程已知的评估（正向）和演化过程未知的评估（逆向）。

演化过程已知的评估其目的在于通过对架构演化过程进行度量，比较架构内部结构上的差异以及由此导致的外部质量属性上的变化，对该演化过程中相关质量属性进行评估。

架构演化评估的执行过程如图所示。图中 A_0 和 A_n 表示一次完整演化前后的相邻版本的软件架构。每经过一次原子演化，即可得到一个架构中间演化版本 A_i 。对每个中间版本架构进行度量，得到架构 A_i 的质量属性度量值 Q_i ， $D(i-1,i)$ 是版本间的质量属性距离。

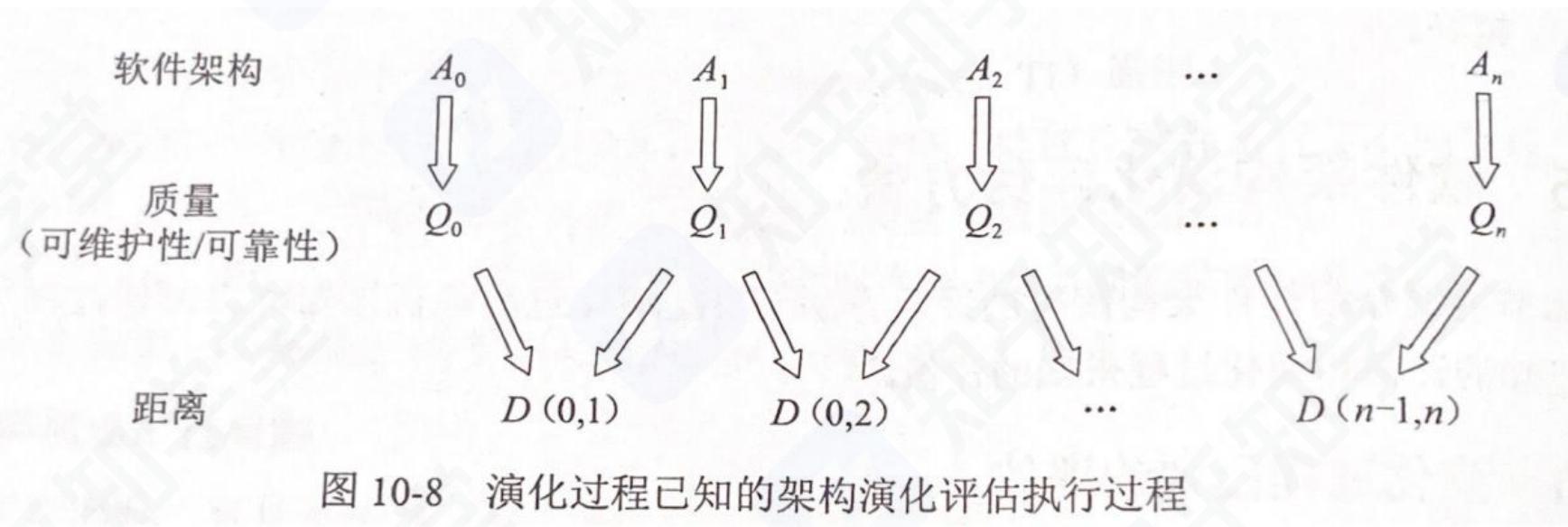


图 10-8 演化过程已知的架构演化评估执行过程

基于度量的架构演化评估方法，其基本思路在于通过对演化前后的软件架构进行度量，比较架构内部结构上的差异以及由此导致的外部质量属性上的变化。其中包括：架构修改影响分析、监控演化过程、分析关键演化过程。

当演化过程未知时，我们无法像演化过程已知时那样追踪架构在演化过程中的每一步变化，只能根据架构演化前后的度量结果逆向推测出架构发生了哪些改变，并分析这些改变与架构相关质量属性的关联关系。

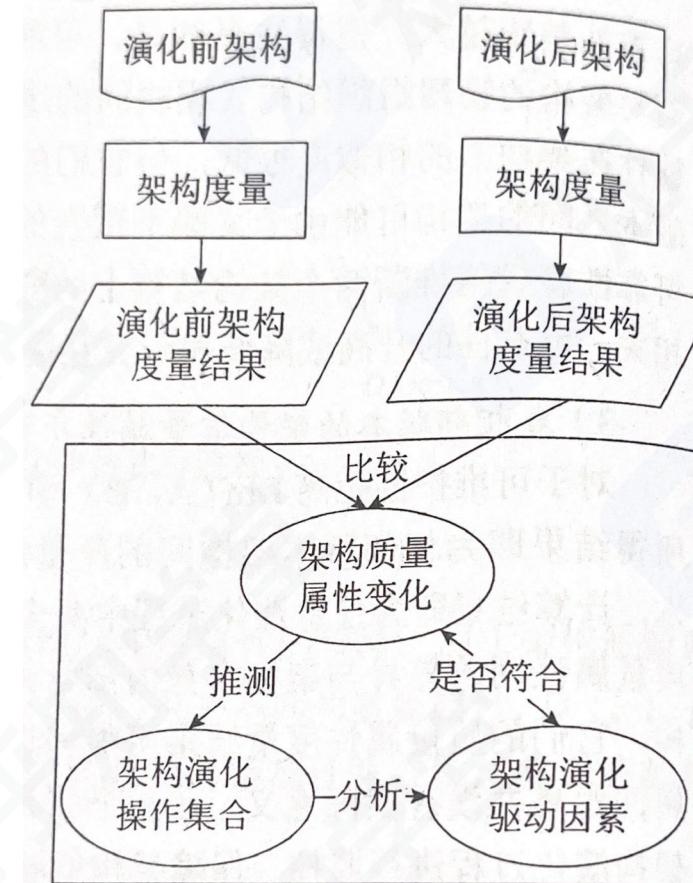


图 10-9 演化过程未知时的架构演化
评估过程示意图



第一阶段：单体架构

特点：所有业务在一个服务器上

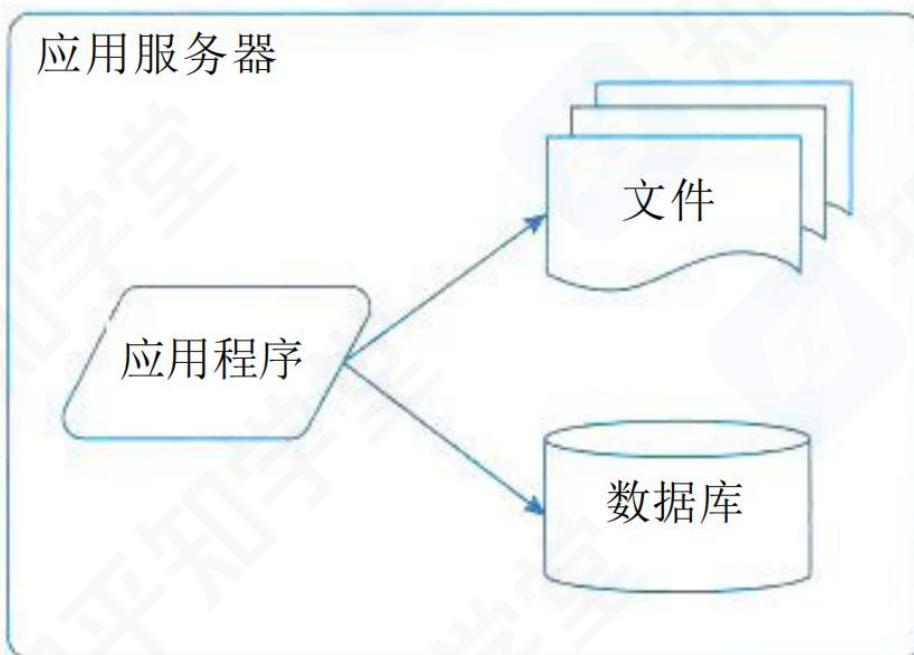


图10-10 第一阶段网站架构

第二阶段：垂直架构

特点：每种业务在一个单独服务器上

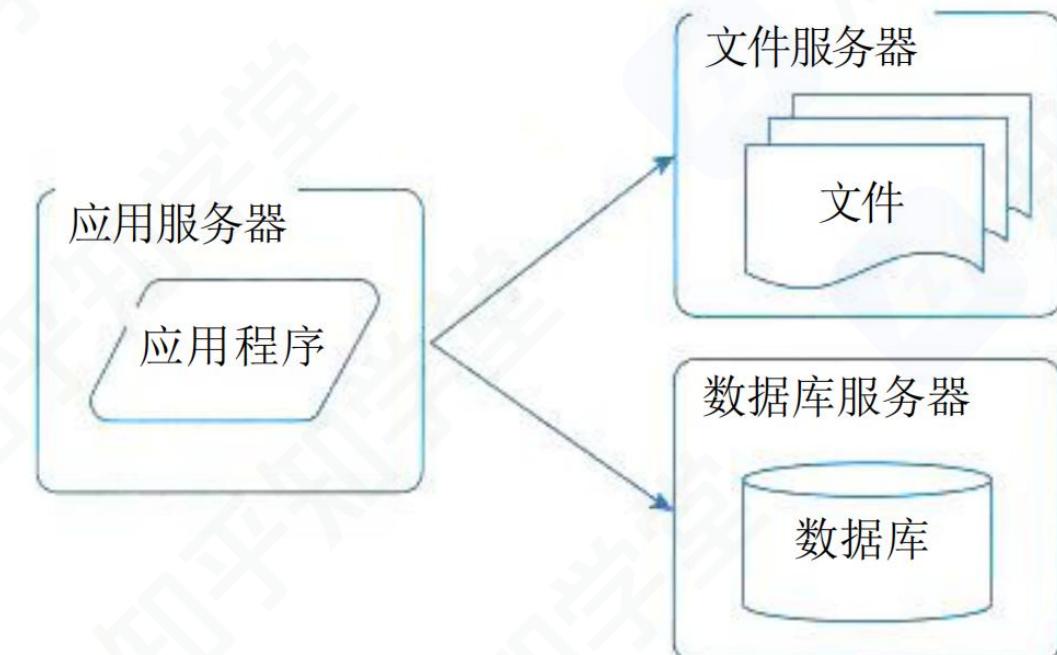


图10-11 第二阶段网站架构

第三阶段：使用缓存改善网站性能
特点：使用缓存提高网站查询效率

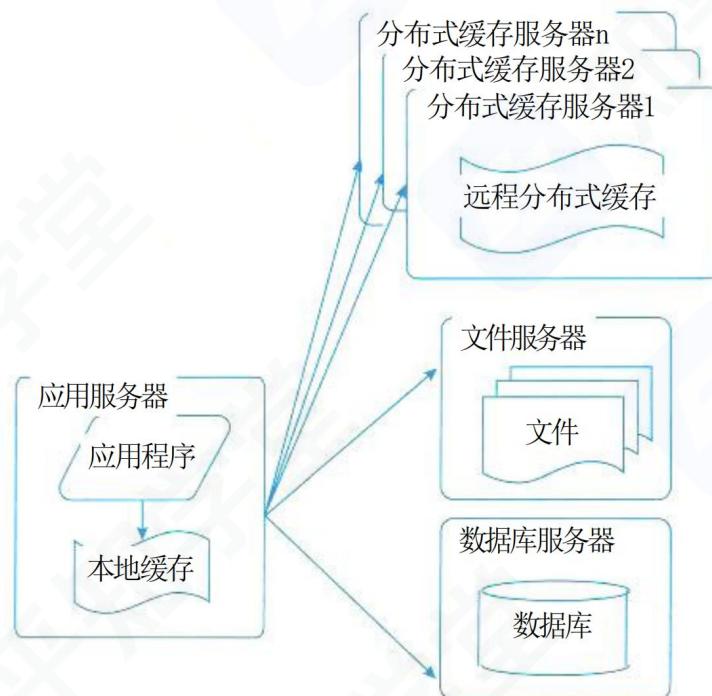


图10-12第三阶段网站架构

第四阶段：使用服务集群改善网站并发处理能力
特点：多台应用服务器来解决访问量过大效率降低的问题

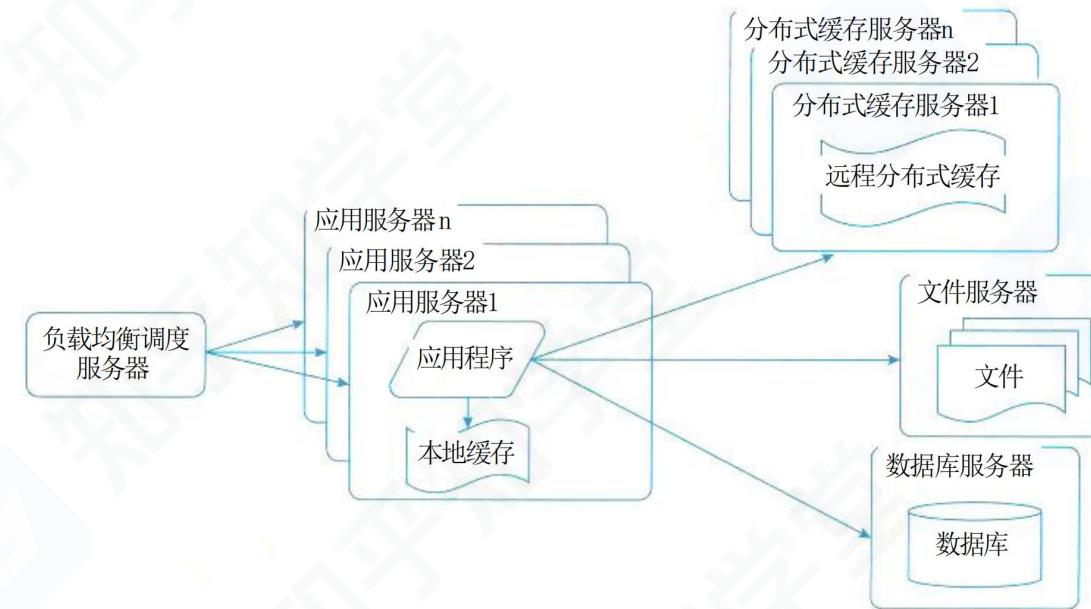


图10-13第四阶段网站架构

第五阶段：数据库读写分离

特点：数据库拆分为主从数据库

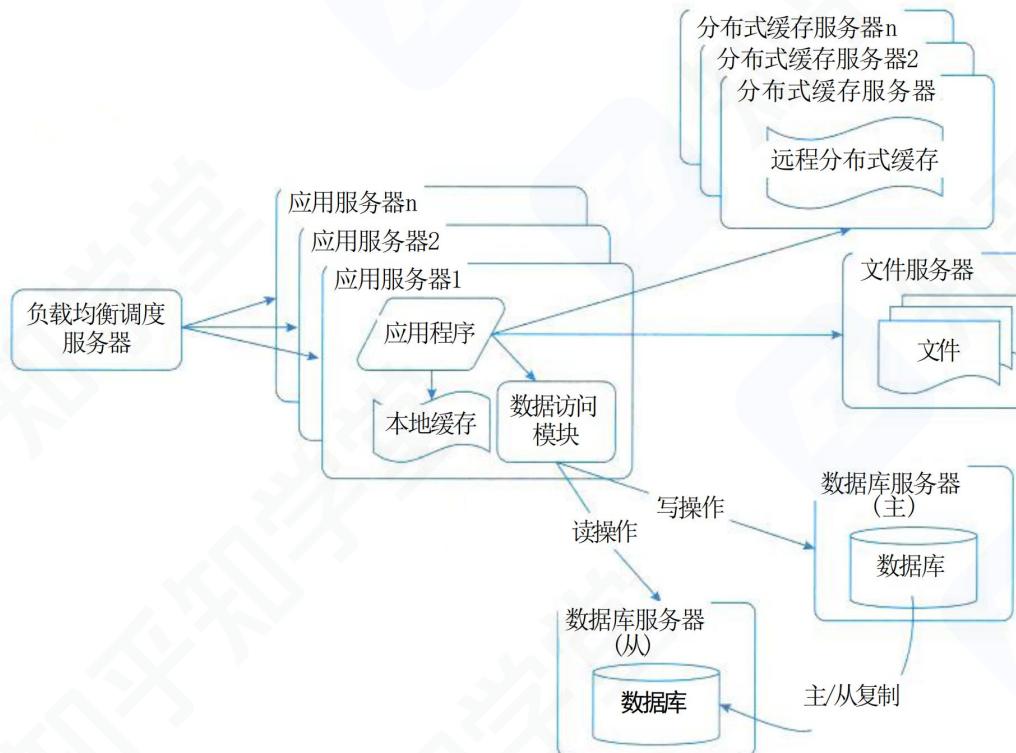


图10-14第五阶段网站架构

第六阶段：使用反向代理和CDN加速网站响应

特点：使用CDN解决不同地域访问效率差异问题

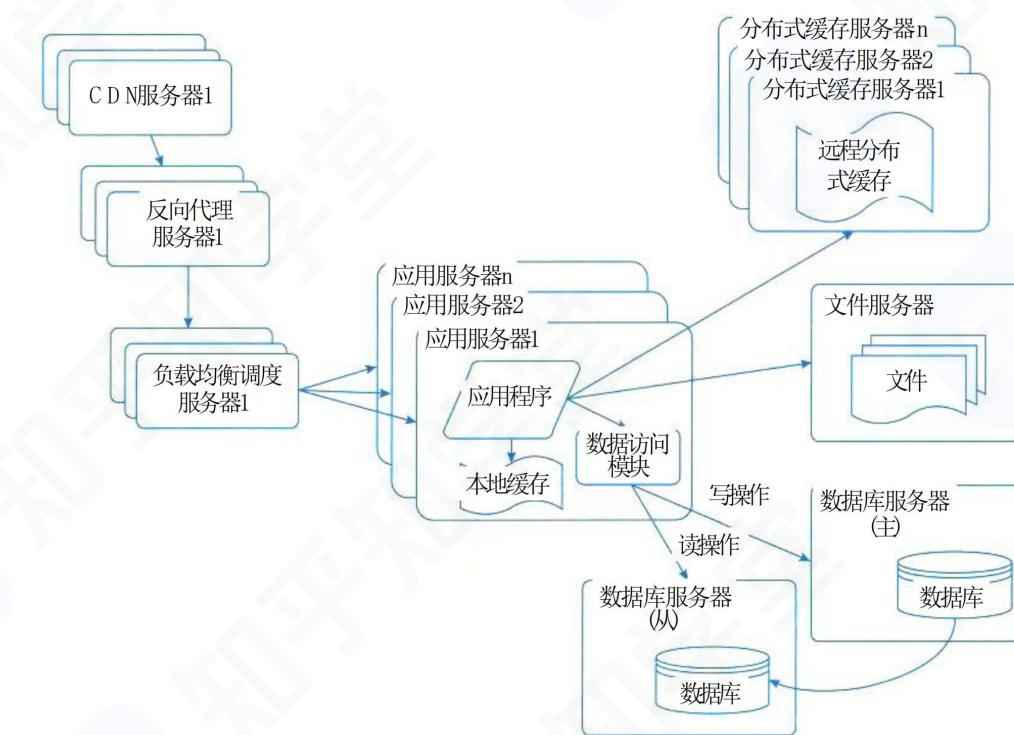


图10-15第六阶段网站架构

第七阶段：使用分布式文件系统和分布式数据库系统

特点：文件服务器和数据库服务器使用分布式来部署分摊访问压力

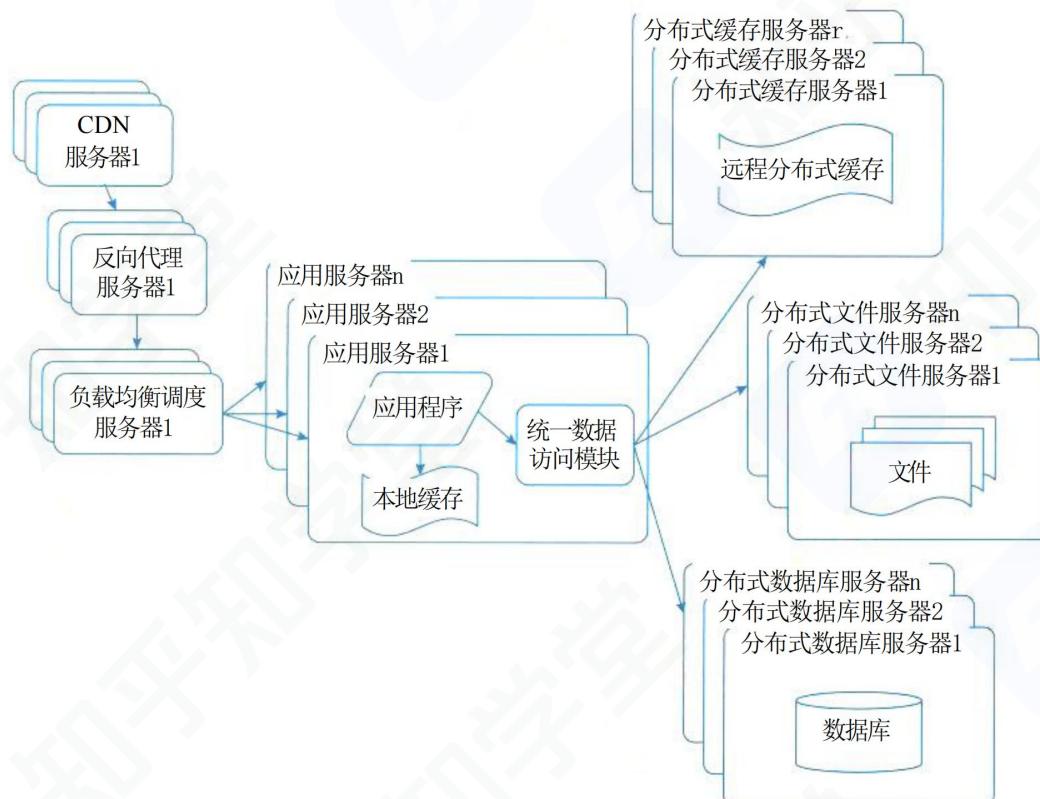


图10-16第七阶段网站架构

第八阶段：使用NOSQL和搜索引擎

特点：使用NOSQL数据库提升访问性能

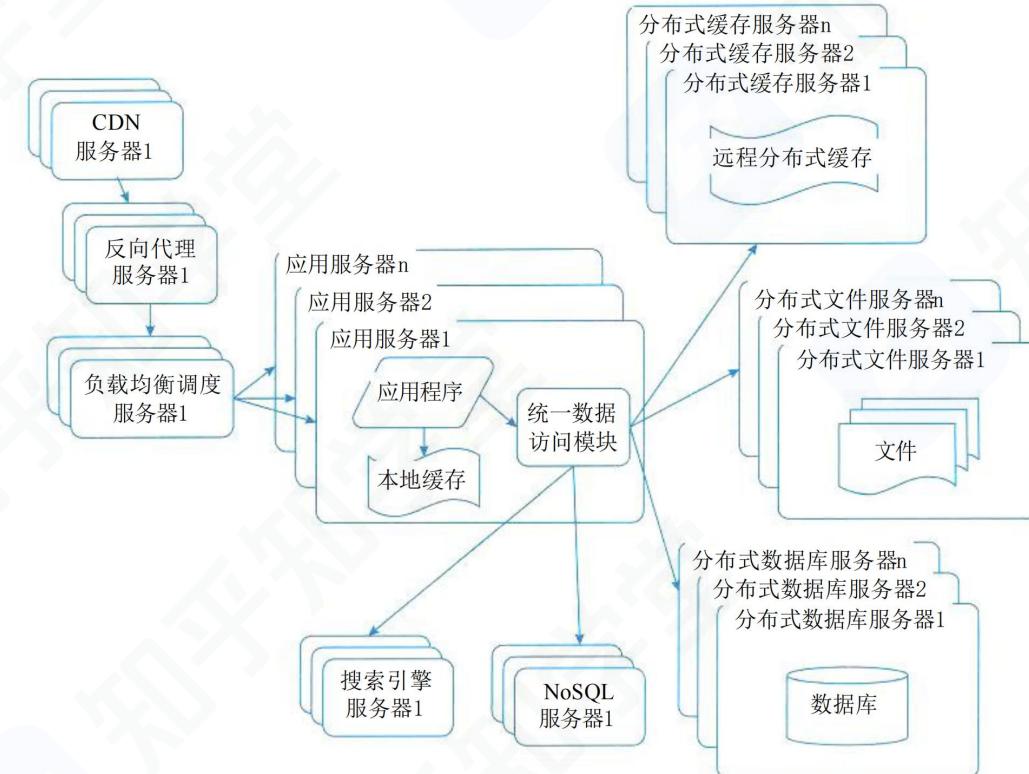


图10-17第八阶段网站架构

第九阶段：业务拆分

特点：把所有业务拆分成微服务的形式，更加灵活方便部署和调整

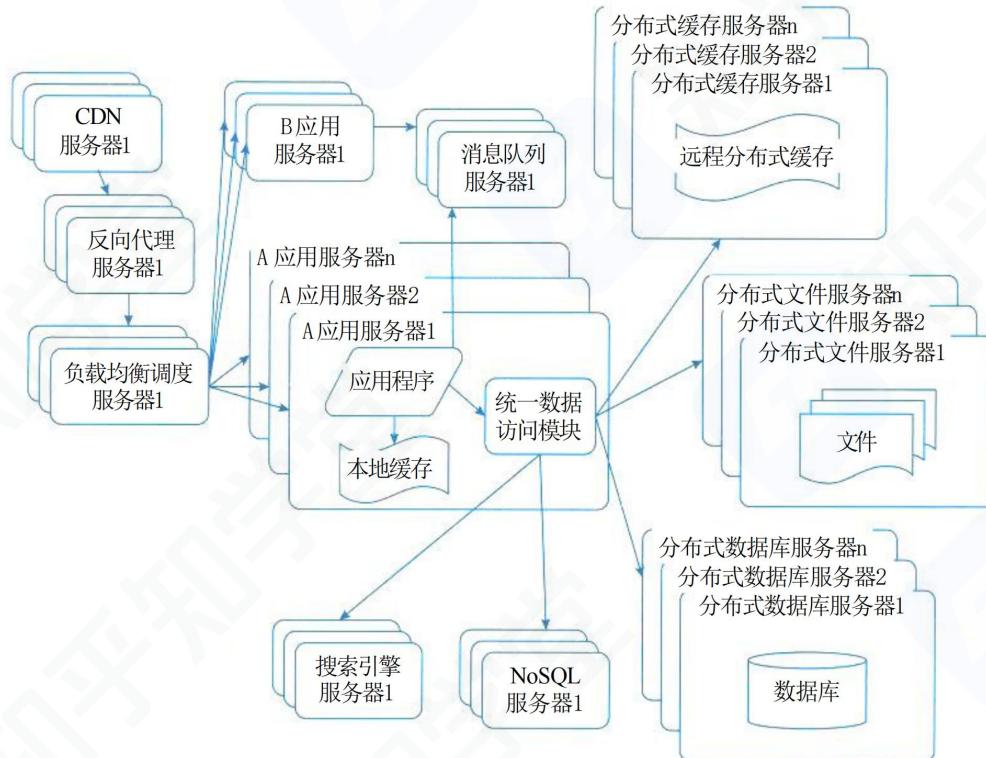


图10-18 第九阶段网站架构

第十阶段：分布式服务

特点：把应用服务器也从集群改为分布式，进一步提升性能

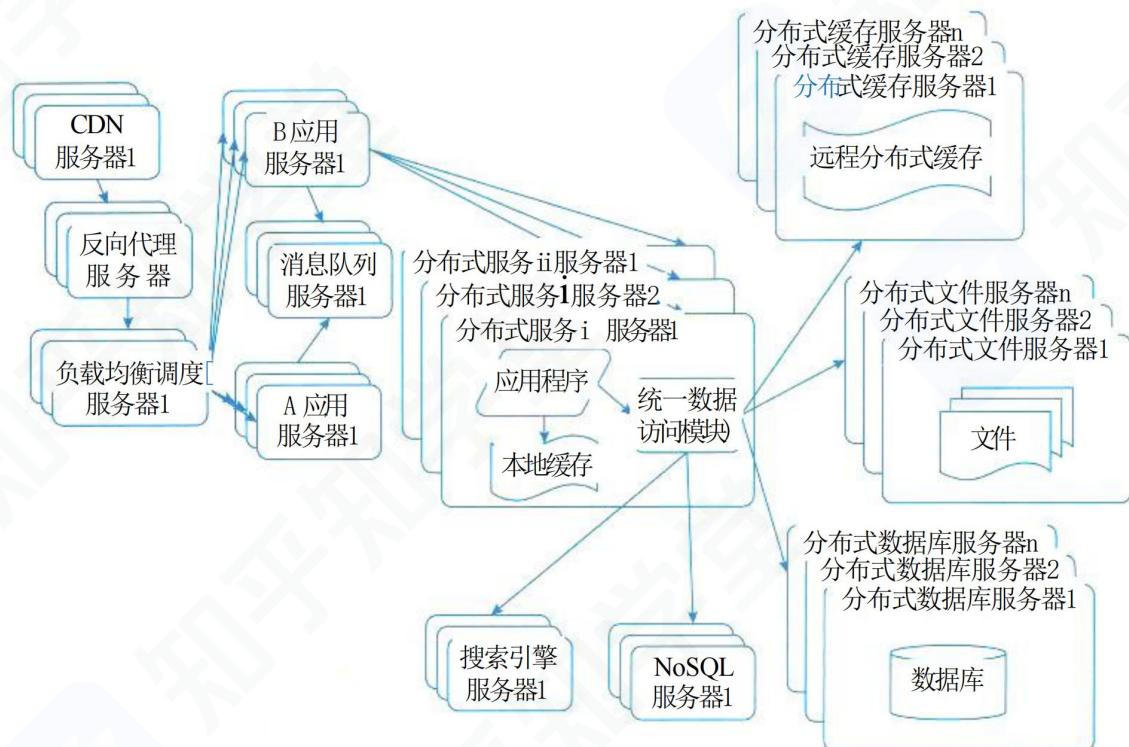


图10-19第十阶段网站架构

软件架构维护过程一般涉及架构知识管理、架构修改管理和架构版本管理。

- 软件架构知识管理是对架构设计中所隐含的决策来源进行文档化表示，进而在架构维护过程中帮助维护人员对架构的修改进行完善的考虑，并能够为其他软件架构的相关活动提供参考。
 - 架构知识的定义：架构知识=架构设计 + 架构设计决策。即需要说明在进行架构设计时采用此种架构的原因。
 - 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，从架构文档等信息来源中捕捉架构知识，进而提供架构的质量属性及其设计依据以进行记录和评价。
- 在软件架构修改管理中，一个主要的做法就是建立一个隔离区域保障该区域中任何修改对其他部分的影响比较小，甚至没有影响。为此，需要明确修改规则、修改类型，以及可能的影响范围和副作用等。
- 软件架构版本管理为软件架构演化的版本演化控制、使用和评价等提供了可靠的依据，并为架构演化量化度量奠定了基础。



练习题



1. 在软件系统的生命周期里，软件的演化速率趋于稳定，如相邻版本的更新率相对稳定。此描述是软件架构演化的()原则。

- A.主体维持 B.系统总体结构优化 C.平滑演化 D.目标一致

2. 软件架构维护过程不包括（ ）。

- A.架构知识管理 B.架构修改管理 C.架构版本管理 D.架构构件管理

3.下列软件架构演化时期，()是在系统设计时规定了演化的具体条件，将系统置于“安全”模式下，演化只发生在某些特定约束满足时，可以进行一些规定好的演化操作。

- A.设计时演化 B.运行前演化 C.有限制运行时演化 D.运行时演化



05

未来信息综合技术



- 本章节在历年考试过程中的分值占比大概是2-3分，但是最近的一次却没考
- 本章节是改版之后新增的内容，之前有考到过，但是是作为超纲的内容，现在单独把这些未来信息技术拎出来作为一章，但是需要注意的是，这一章仍然没有覆盖到所有的前沿技术，比如之前考到的区块链技术在这章就没有呈现，所以老师觉得这一部分很有可能会考到一些不在本章内容里的前沿技术，我们主要是把这一章的内容当看故事一样的过一遍即可
- 被考到知识点有：
 - 信息物理系统技术、人工智能技术
 - 机器人技术、边缘计算
 - 数字孪生体技术、云计算和大数据技术
- 在改版之后的考试中，分别考察的知识点为：
 - 2023年11月：CDN、反向代理、机器学习
 - 2024年05月：云计算、虚拟化、数字孪生
 - 2024年11月：无

信息物理系统(CPS)是控制系统、嵌入式系统的扩展与延伸，其涉及的相关底层理论技术源于对嵌入式技术的应用与提升。**CPS的核心概念是将数字和物理系统融合在一起，以实现更好的协同工作和决策**

CPS通过集成先进的感知、计算、通信、控制等信息技术和自动控制技术，构建了物理空间与信息空间中人、机、物、环境、信息等要素相互映射、适时交互、高效协同的复杂系统，实现系统内资源配置和运行的按需响应、快速迭代、动态优化。

CPS的体系架构

- **单元级CPS**: 是具有不可分割性的CPS最小单元，是具备可感知、可计算、可交互、可延展、自决策功能的CPS最小单元，一个智能部件、一个工业机器人或一个智能机床都可能是一个CPS最小单元。比如智能家居中的智能灯泡
- **系统级CPS**: 多个最小单元(单元级)通过工业网络(如工业现场总线、工业以太网等)，实现更大范围、更宽领域的数据自动流动，实现了多个单元级CPS的互联、互通和互操作，进一步提高制造资源优化配置的广度、深度和精度。包含互联互通、即插即用、边缘网关、数据互操作、协同控制、监视与诊断等功能。比如在汽车制造工厂中，生产线上的各种机器人和设备可以被视为系统级CPS。这些设备通过工业以太网连接，共享生产数据和控制信息。它们可以协同工作，自动调整生产速度和流程以适应需求的变化
- **SoS级**: 多个系统级CPS的有机组合构成SoS级CPS。比如在一个智能城市中，多个系统级CPS (例如交通管理系统、能源管理系统、环境监测系统) 可以组成SoS级CPS。这些系统级CPS共享数据，例如交通流量、能源消耗、空气质量等。集成大数据分析和数据服务，城市可以实时监测和优化交通流量，实现节能减排，提供紧急事件响应等服务。整个城市成为一个智能的、协同工作的系统。

CPS技术体系主要分为：CPS总体技术、CPS支撑技术、CPS核心技术。

- CPS总体技术：就是CPS的顶层设计技术，主要包括：系统架构、异构系统集成、安全技术、试验验证技术等
- CPS支撑技术：就是基于CPS应用的支撑技术，主要包括：智能感知、嵌入式软件、数据库、人机交互、中间件、SDN(软件定义网络)、物联网、大数据等
- CPS核心技术：就是是CPS的基础技术，主要包括：虚实融合控制、智能装备、MBD、数字孪生技术、现场总线、工业以太网、CAX\MES\ERP\PLM\CRM\SCM等

上述技术体系可以分为四大核心技术要素即“一硬”(感知和自动控制)、“一软”(工业软件)、“一网”(工业网络), “一平台”(工业云和智能服务平台)。

- 感知和自动控制是CPS实现的硬件支撑；
- 工业软件固化了CPS计算和数据流程的规则，**是CPS的核心**；
- 工业网络是互联互通和数据传输的网络载体；
- 工业云和智能服务平台是CPS 数据汇聚和支撑上层解决方案的基础，对外提供资源管控和能力服务。



人工智能



人工智能(AI): 利用数字计算机或者数字计算机控制的机器模拟、延伸和扩展人的智能，感知环境、获取知识并使用知识获得最佳结果的理论、方法、技术及应用系统。

人工智能的目标：了解智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器。该领域的研究包括机器人、自然语言处理、计算机视觉和专家系统等。

根据人工智能是否能真正实现推理、思考和解决问题，可以将人工智能分为弱人工智能和强人工智能。

1. 自然语言处理(NLP)。研究实现人与计算机之间用自然语言进行有效通信的各种理论和方法。主要包括机器翻译(从一种自然语言到另外一种自然语言的翻译)、语义理解(利用计算机理解文本篇章内容，并回答相关问题)和问答系统(让计算机像人类一样用自然语言与人交流)等。比如机器翻译是NLP的一个经典应用
2. 计算机视觉。是使用计算机模仿人类视觉系统的科学，让计算机拥有类似人类提取、处理、理解和分析图像以及图像序列的能力，将图像分析任务分解为便于管理的小块任务。比如车牌识别、人脸识别
3. 知识图谱。就是把所有不同种类的信息连接在一起而得到的一个关系网络，提供了从“关系”的角度去分析问题的能力。谷歌知识图谱是一个大规模的知识图谱，它将世界上的各种信息组织成一个关系网络。例如，当您在Google搜索中查找有关某个名人的信息时，知识图谱可以提供与该名人相关的详细信息、事件和关系，以帮助您更好地理解。
4. 人机交互(HCI)。主要研究人和计算机之间的信息交换。比如触摸屏界面是一种常见的人机交互方式
5. 虚拟现实或增强现实(VR/AR)。以计算机为核心的新型视听技术。结合相关科学技术，在一定范围内生成与真实环境在视觉、听觉等方面高度近似的数字化环境
6. 机器学习(ML)。是以数据为基础，通过研究样本数据寻找规律，并根据所得规律对未来数据进行预测。目前，机器学习广泛应用于数据挖掘、计算机视觉、自然语言处理、生物特征识别等领域。比如垃圾邮件过滤器

按照学习模式的不同，机器学习可分为监督学习、无监督学习、半监督学习、强化学习。其中，监督学习需要提供标注的样本集，无监督学习不需要提供标注的样本集，半监督学习需要提供少量标注的样本，而强化学习需要反馈机制。

按照学习方法的不同，机器学习可分为传统机器学习和深度学习。区别在于，传统机器学习的领域特征需要手动完成，且需要大量领域专业知识；深度学习不需要人工特征提取，但需要大量的训练数据集以及强大的GPU服务器来提供算力。

- 传统机器学习从一些观测(训练)样本出发，试图发现不能通过原理分析获得的规律，实现对未来数据行为或趋势的准确预测。在自然语言处理、语音识别、图像识别、信息检索等许多计算机领域获得了广泛应用。
- 深度学习是一种基于多层神经网络并以海量数据作为输入规则的自学习方法，依靠提供给它的大量实际行为数据(训练数据集)，进行参数和规则调整。深度学习更注重特征学习的重要性。

机器学习的常见算法还包括迁移学习、主动学习和演化学习。

- 迁移学习：是指当在某些领域无法取得足够多的数据进行模型训练时，利用另一领域数据获得的关系进行的学习。假设你想构建一个情感分析模型，但在某些特定领域（例如医学）中没有足够的数据来训练模型。但是在一般的互联网上有大量的社交媒体评论和用户反馈的数据，你可以利用这些数据进行情感分析的迁移学习
- 主动学习：通过一定的算法查询最有用的未标记样本，并交由专家进行标记，然后用查询到的样本训练分类模型来提高模型的精度。比如在垃圾邮件过滤器中，主动学习可以用于改善模型的性能
- 演化学习：基于演化算法提供的优化工具设计机器学习算法，针对机器学习任务中存在大量的复杂优化问题，应用于分类、聚类、规则发现、特征选择等机器学习与数据挖掘问题中。例如，在图像分类任务中，演化学习可以自动选择最相关的图像特征，以提高分类准确性。通过不断地进化和选择特征，模型可以逐渐提高性能，找到最佳的特征组合

人工智能目前典型应用：chatgpt、deepseek、豆包、grok

机器人技术已经准备进入4.0时代。所谓机器人4.0时代，就是把云端大脑分布在各个地方，充分利用边缘计算的优势，提供高性价比的服务，把要完成任务的记忆场景的知识和常识很好地组合起来，实现规模化部署。特别强调机器人除了具有感知能力实现智能协作，还应该具有一定的理解和决策能力，进行更加自主的服务。

我们目前的服务机器人大多可以做到物体识别和人脸识别。在机器人4.0时代，我们需要加上更强的自适应能力。

机器人4.0的核心技术

- 云-边-端的无缝协同计算。云-边-端一体的机器人系统是面向大规模机器人的服务平台，信息处理和生成主要在云边-端上分布处理完成。通常情况下，云侧可以提供高性能的计算和知识存储，边缘侧用来进一步处理数据并实现协同和共享。机器人端只用完成实时操作的功能。
- 持续学习与协同学习。希望机器人可以通过少量数据来建立基本的识别能力，然后可以自主地去找到更多的相关数据并进行自动标注。然后用这些自主得到的数据来对自己已有的模型进行重新训练来提高性能。
- 知识图谱。需要更加动态和个性化的知识；需要和机器人的感知与决策能力相结合。
- 场景自适应。主动观察场景内人和物之间的变化，预测可能发生的事件，从而影响之后的行动模式。这个技术的关键问题在于场景预测能力。就是机器人通过对场景内的各种人和物进行细致的观察，结合相关的知识和模型进行分析，并预测之后事件即将发生的时间，改变自己的行为模式。
- 数据安全。既要保证端到端的安全传输，也要保障服务器端的安全存储。



如果按照要求的控制方式分类，机器人可分为**操作机器人、程序机器人、示教再现机器人、智能机器人和综合机器人**。

- 操作机器人。典型代表是在核电站处理放射性物质时远距离进行操作的机器人。
- 程序机器人。可以按预先给定的程序、条件、位置进行作业。比如工业生产线上的焊接机器人
- 示教再现机器人。机器人可以将所教的操作过程自动地记录在磁盘、磁带等存储器中，当需要再现操作时，可重复所教过的动作过程。示教方法有直接示教与遥控示教两种。比如医疗手术机器人，外科医生可以使用机器人执行复杂的手术，并将这些操作记录下来。当需要再次执行相同的手术时，机器人可以重复之前示教的动作过程，以保持手术的准确性和精确性。
- 智能机器人。既可以进行预先设定的动作，还可以按照工作环境的改变而变换动作。比如家用扫地机器人是智能机器人的一个示例。它们可以根据环境的改变自主决策和动作。这些机器人配备了传感器和算法，可以避开障碍物、自动充电并根据不同的房间布局规划清扫路径
- 综合机器人。由操纵机器人、示教再现机器人、智能机器人组合而成的机器人，如火星机器人。整个系统可以看作是由地面指令操纵的操作机器人。

如果按照应用行业来分，机器人可分为**工业机器人、服务机器人和特殊领域机器人**。

边缘计算：是一种分布式计算模型，其中计算资源和数据存储被放置在物理世界的边缘，靠近数据源和终端设备，以降低延迟、提高性能，并更好地满足实时性要求。

边缘计算的主要目标：是将计算功能放置在接近数据源的位置，以便在本地处理数据，减少与远程云计算的通信和响应时间，从而提高效率和实时性。

举例：在智能城市中，交通管理是一个重要的应用场景。边缘计算可以应用于交通信号灯控制系统。传感器和摄像头安装在交通信号灯附近，用于监测交通流量和交通情况。边缘计算设备（如边缘服务器）安装在信号灯控制器旁边，能够实时分析和处理传感器和摄像头捕获的数据。这些设备可以根据实时数据情况智能地调整交通信号的时序，以优化交通流量和减少拥堵。

边缘计算将数据的处理、应用程序的运行甚至一些功能服务的实现，由网络中心下放到网络边缘的节点上。在网络边缘侧的智能网关上就近采集并且处理数据，不需要将大量未处理的原生数据上传到远处的大数据平台。

采用边缘计算的方式，海量数据能够就近处理，大量的设备也能实现高效协同的工作，诸多问题迎刃而解。因此，边缘计算理论上可满足许多行业在敏捷性、实时性、数据优化、应用智能，以及安全与隐私保护等方面的关键需求。

边缘计算的业务本质是云计算在数据中心之外汇聚节点的延伸和演进，主要包括云边缘、边缘云和云化网关三类落地形态；以“边云协同”和“边缘智能”为核心能力发展方向。

云边缘：是指在边缘设备（例如传感器、摄像头、工业机器等）附近部署的**小型计算节点**，通常位于数据源的最近位置。这些节点可以处理、分析和存储数据，并执行本地计算任务。比如智能监控摄像头可以检测异常情况，如交通事故或可疑行为，而无需将所有数据传输到远程云数据中心。这减少了通信延迟，提高了响应速度。

边缘云：是一种**云计算模型**，将云服务部署到靠近数据源和终端设备的边缘位置。边缘云通常包括较大规模的计算资源，可以处理多个边缘设备的数据，比如边缘数据中心

注意：云边缘更侧重于在边缘设备上执行实时计算和响应，而边缘云是将云计算资源推送到离数据源更近的位置，以处理更大规模的边缘计算任务

云化网关：以云化技术与能力重构原有嵌入式**网关系统**（是一种将网络功能虚拟化并将其作为云服务提供的网关），比如工厂自动化控制系统，云化网关可以用于连接和控制各种工厂设备。这些网关设备位于工厂内部，负责与各种传感器、机器和生产设备通信。它们将实时数据传输到本地边缘计算节点，这些节点可以进行实时数据分析和设备控制。同时，云化网关还将数据上传到远程云平台，用于生产计划、维护管理和质量控制。这种方式实现了工厂自动化的高效运营，并满足了实时监控和长期数据分析的需求。

边云协同：边缘计算与云计算各有所长，云计算擅长全局性、非实时、长周期的大数据处理与分析，能够在长周期维护、业务决策支撑等领域发挥优势；边缘计算更适用局部性、实时、短周期数据的处理与分析，能更好地支撑本地业务的实时智能化决策与执行。

边缘计算既靠近执行单元，更是云端所需高价值数据的采集和初步处理单元，可以更好地支撑云端应用，而云计算则是通过大数据分析优化输出的业务规则或模型可以下发到边缘侧，边缘计算基于新的业务规则或模型运行，两者相辅相成，主要包括六种协同：

- 资源协同：边缘节点提供计算、存储、网络、虚拟化等基础设施资源、具有本地资源调度管理能力，同时可与云端协同，接受并执行云端资源调度管理策略，包括边缘节点的设备管理、资源管理以及网络连接管理。
- 数据协同：边缘节点主要负责现场/终端数据的采集，按照规则或数据模型对数据进行初步处理与分析，并将处理结果以及相关数据上传给云端；云端提供海量数据的存储、分析与价值挖掘。
- 智能协同：边缘节点按照AI模型执行推理，实现分布式智能；云端开展AI的集中式模型训练，并将模型下发边缘节点。
- 应用管理协同：边缘节点提供应用部署与运行环境，并对本节点多个应用的生命周期进行管理调度；云端主要提供应用开发、测试环境，以及应用的生命周期管理能力。
- 业务管理协同：边缘节点提供模块化、微服务化的应用/数字孪生/网络等应用实例；云端主要提供按照客户需求实现应用、数字孪生、网络等的业务编排能力。
- 服务协同：边缘节点按照云端策略实现部分ECSaaS服务，通过ECSaaS与云端SaaS的协同实现面向客户的按需SaaS服务；云端主要提供SaaS服务在云端和边缘节点的服务分布策略，以及云端承担的SaaS服务能力。

边缘计算的应用场合(既有中央控制中心，又有分支设备)：智慧园区、安卓云与云游戏、视频监控、工业互联网、智慧医疗。

数字孪生体技术：是跨层级、跨尺度的现实世界和虚拟世界建立沟通的桥梁。指创建一个虚拟的物理对象的数字模型，并使用实时数据对其进行更新，以反映物理对象的真实状态。数字孪生可以用于模拟、预测和优化物理对象的性能。

数字孪生体是现有或将有的物理实体对象的数字模型，通过实测、仿真和数据分析来实时感知、诊断、预物理实体对象的状态，通过优化和指令来调控物理实体对象的行为，通过相关数字模型间的相互学习来进化自身，同时改进利益相关方在物理实体对象生命周期内的决策。

关键技术：

- 建模。建模的目的是将我们对物理世界的理解进行简化和模型化。而数字孪生体的目的或本质是通过数字化和模型化，用信息换能量，以使少的能量消除各种物理实体、特别是复杂系统的不确定性。需求指标、生存期阶段和空间尺度构成了数字孪生体建模技术体系的三维空间。
- 仿真。如果说建模是模型化我们对物理世界或问题的理解，那么仿真就是验证和确认这种理解的正确性和有效性。所以，数字化模型的仿真技术是创建和运行数字孪生体、保证数字孪生体与对应物理实体实现有效闭环的核心技术。仿真将包含了确定性规律和完整机理的模型转化成软件的方式来模拟物理世界的一种技术。只要模型正确，并拥有了完整的输入信息和环境数据，就可以基本准确地反映物理世界的特性和参数。

数字孪生体的应用：

- 制造业：用于产品设计、制造、维护等环节。例如，在产品设计阶段，可以利用数字孪生技术进行虚拟样机测试，以减少设计缺陷；在制造阶段，可以利用数字孪生技术进行生产过程监控，以提高生产效率；在维护阶段，可以利用数字孪生技术进行故障预测，以降低维护成本。
- 能源：用于电网、风力发电等设施的管理。例如，在电网管理中，可以利用数字孪生技术进行电网运行模拟，以提高电网的稳定性和安全性；在风力发电中，可以利用数字孪生技术进行风力发电机组的故障预测，以提高发电效率。
- 医疗：用于手术规划、康复治疗等环节。例如，在手术规划中，可以利用数字孪生技术进行虚拟手术，以提高手术的成功率；在康复治疗中，可以利用数字孪生技术进行康复训练，以提高康复效果。

云计算概念的内涵包含两个方面：**平台和应用**。平台即基础设施，其地位相当于PC上的操作系统，云计算应用程序需要构建在平台之上；云计算应用所需的计算与存储通常在“云端”完成，客户端需要通过互联网访问计算与存储能力。

云计算的服务方式

- **基础设施即服务(IaaS)**。在IaaS模式下，服务提供商将多台服务器组成的“云端”基础设施作为计量服务提供给客户。具体来说，服务提供商将内存、IO设备、存储和计算能力等整合为一个虚拟的资源池，为客户提供所需要的存储资源、虚拟化服务器等服务。比如选择云端的不同配置的服务器
- **平台即服务(PaaS)**。在PaaS模式下，服务提供商将分布式开发环境与平台作为一种服务来提供。这是一种分布式平台服务，厂商提供开发环境、服务器平台、硬件资源等服务给客户，客户在服务提供商平台的基础上定制开发自己的应用程序，并通过其服务器和互联网传递给其他客户。比如在线编程、在线办公软件
- **软件即服务(SaaS)**。在SaaS的服务模式下，服务提供商将应用软件统一部署在云计算平台上，客户根据需要通过互联网向服务提供商订购应用软件服务，服务提供商根据客户所订购软件的数量、时间的长短等因素收费，并且通过标准浏览器向客户提供应用服务。比如百度云盘之类的

三者比较：

- 在灵活性方面，SaaS < PaaS < IaaS 灵活性依次增强。
- 在方便性方面，IaaS < PaaS < SaaS 方便性依次增强。

云计算的部署模式：

- 公有云。在公有云模式下，云基础设施是公开的，可以自由地分配给公众。企业、学术界与政府机构都可以拥有和管理公用云，并实现对公有云的操作。公有云能够以低廉的价格为最终用户提供有吸引力的服务，创造新的业务价值。
- 社区云。在社区云模式下，云基础设施分配给一些社区组织所专有，这些组织共同关注任务、安全需求、政策等信息。云基础设施被社区内的一个或多个组织所拥有、管理及操作。“社区云”是“公有云”范畴内的一个组成部分。
- 私有云。在私有云模式下，云基础服务设施分配给由多种用户组成的单个组织。它可以被这个组织或其他第三方组织所拥有、管理及操作。
- 混合云。混合云是公有云、私有云和社区云的组合。由于安全和控制原因，并非所有的企业信息都能放置在公有云上，因此企业将会使用混合云模式。

大数据是指其大小或复杂性无法通过现有常用的软件工具，以合理的成本并在可接受的时限内对其进行捕获和管理的数据集。这些困难包括数据的收集、存储、搜索、共享、分析和可视化。

大数据的特点：大规模、高速度、多样化、可变性、复杂性等。

大数据分析的分析步骤：

- 数据获取/记录
- 信息抽取/清洗
- 数据集成/聚集/表现
- 数据分析/建模
- 数据解释

大数据的应用领域：制造业、服务业、交通行业、医疗行业。



区块链是一种分布式数据库技术，以区块的形式按时间顺序链接在一起，形成了一个不断增长的、不可篡改的记录链。每个区块包含了一批数据（区块头：包含区块的版本号、时间戳、上一个区块的哈希值等信息。交易信息：包含交易双方、交易时间、交易内容等信息。），这些数据通过密码学（哈希函数）技术连接在一起，形成一个链条。这个技术最初是为了支持比特币这种加密货币而设计的，但现在已经被广泛应用于其他领域。它的核心特性包括：

- **去中心化**：区块链的数据不存储在单一的中心服务器上，而是分布在网络的各个节点上，每个节点都有该链的完整副本。这意味着没有一个中心机构能够控制或篡改整个数据库。
- **不可篡改性**：一旦数据被写入区块链，就几乎不可能被修改或删除。这是因为每个区块都包含了前一个区块的哈希值，形成了一个不可逆的链条结构，任何篡改都会立即被其他节点检测出来。
- **透明性**：区块链中的数据是公开可见的，所有参与者都可以查看和验证数据，从而增加了数据的透明度和可信度。
- **安全性**：区块链使用了加密技术确保数据的安全性，使得交易和信息在传输和存储过程中更加安全可靠。
- **智能合约**：智能合约是基于区块链的自动化合约，能够自动执行合约条款，无需中介机构，提高了交易的效率和可靠性。

区块链的原理：

- **安全性**：区块链的安全性是建立在密码学原理上的，包括哈希函数、非对称加密等技术，确保数据的完整性和不可篡改性。
- **共识机制**：共识机制确保了网络中各个节点对交易记录的一致性认可，从而防止了双重支付等问题。
- **分布式存储**：区块链采用分布式存储方式，数据存储在多个节点上，避免了单点故障和数据丢失的风险。
- **智能合约**：智能合约是一种在区块链上执行的自动化合约，其中包含了预先编写好的代码逻辑，可以自动执行合约条款。

区块链的应用场景：（注重安全和透明）

- **金融**：可以用于构建去中心化金融系统，例如数字货币、交易所等。
- **供应链**：可以用于追踪货物运输过程，提高供应链透明度。
- **医疗**：可以用于存储和管理医疗数据，保障数据安全和隐私。
- **物联网**：可以用于连接和管理物联网设备，实现物联网安全。



练习题

1.CPS技术体系的四大核心技术要求中“一平台”是()。

- A.感知和自动控制
- B.工业软件
- C.工业网络
- D.工业云和智能服务平台

2.人工智能的关键技术包括自然语言处理、计算机视觉、知识图谱、机器学习。机器学习分类中()是利用已标记的有限训练数据集，通过某种学习策略方法建立一个模型，从而实现对新数据/实例标记/映射。

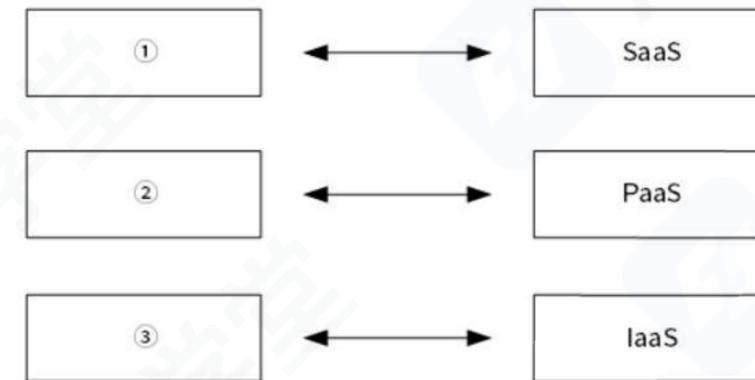
- A.监督学习
- B.无监督学习
- C.半监督学习
- D.强化学习

3.下列哪种场景属于平台即服务 (PaaS) ?

- A. 用户租用云端的虚拟服务器，自主安装操作系统和数据库
- B. 开发者使用在线编程工具开发应用程序，并直接部署到云平台
- C. 企业通过浏览器使用云端提供的客户关系管理 (CRM) 软件
- D. 服务商提供物理服务器的硬件维护和网络带宽

1. 云计算服务体系结构如下图所示，图中①、②、③分别与SaaS、PaaS、IaaS相对应，图中①、②、③应为()。

- A. 应用层、基础设施层、平台层
- B. 应用层、平台层、基础设施层
- C. 平台层、应用层、基础设施层
- D. 平台层、基础设施层、应用层





THE END

功不唐捐，玉汝于成！

• 开启新征程 •

