

NEW PLAN

软考高级架构师

• 一段新征程 •





软件架构的演化和定义



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。



本质上讲，软件架构的演化就是软件整体结构的演化，演化过程涵盖软件架构的全生命周期，包括软件架构需求的获取、软件架构建模、软件架构文档、软件架构实现以及软件架构维护等阶段。



软件架构演化的重要性体现在：一是架构是整个系统的骨架，是软件系统具备诸多好的特性的保障；二是软件架构作为软件蓝图为人们宏观管控软件系统的整体复杂性和变化性提供了一条有效途径。



软件架构的演化能降低软件演化的成本的原因：

- 对系统的软件架构进行的形式化、可视化表示提高了软件的可构造性，便于软件演化。
- 软件架构设计方案涵盖的整体结构信息、配置信息、约束信息等有助于开发人员充分考虑未来可能出现的演化问题、演化情况和演化环境。
- 架构设计时对系统组件之间的耦合描述有助于软件系统的动态调整。



软件架构的定义包含组件、连接件、约束三大要素，这类软件架构演化主要关注的就是这三者之间的添加、修改和删除等。



软件架构的演化和定义



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



面向对象软件架构演化主要分为以下四种演化：对象演化、消息演化、复合片段演化和约束演化



对象演化：在顺序图中，组件的实体是对象，会对架构设计的动态行为产生影响的演化只包括Add Object (AO)和Delete Object(DO)两种。

- AO表示在顺序图中添加一个新的对象。这种演化一般是在系统需要添加新的对象来实现某种新的功能，或需要将现有对象的某个功能独立以增加架构灵活性的时候发生。
- DO删除顺序图中现有的一个对象。这种演化一般在系统需要移除某个现有的功能，或需要合并某些对象及其功能来降低架构的复杂度的时候发生。



消息演化：将消息演化分为AddMessage(AM)、DeleteMessage (DM)、SwapMessageOrder (SMO)、OverturnMessage (OM)、ChangeMessageModule (CMM)5种。

M增添一条新的消息，产生在对象之间需要增加新的交互行为的时候。

- DM删除当前的一条消息，产生在需要移除某个交互行为的时候，是AM的逆向演化。
- SMO交换两条消息的时间顺序，发生在需要改变两个交互行为之间关系的时候。
- OM反转消息的发送对象与接收对象，发生在需要修改某个交互行为本身的时候。
- CMM改变消息的发送或接收对象，发生在需要修改某个交互行为本身的时候。



面向对象软件架构演化



复合片段演化：复合片段是对象交互关系的控制流描述，表示可能发生在不同场合的交互，与消息同属于连接件范畴。复合片段的演化分为AddFragment (AF)、Deletefragment(DF)、FragmentTypeChange (FTC)和FragmentConditionChange (FCC)。

- FCC改变复合片段内部执行的条件，发生在改变当前控制流的执行条件时。自动机中与控制流执行条件相对应的转移包括两个，一个是符合条件时的转移，另一个是不符合条件时的转移，因此每次发生FCC 演化时会同时修改这两个转移的触发事件。
- AF在某几条消息上新增复合片段，发生在需要增添新的控制流时。复合片段所产生的分支是不同类型的。
- DF删除某个现有的复合片段，发生在需要移除当前某段控制流时。DF 与AF互为逆向演化过程。FTC改变复合片段的类型，发生在需要改变某段控制流时。类型演化意味着交互流程的改变，一般伴随着条件、内部执行序列的同时演化，可以视为复合片段的删除与添加的组合。



约束演化：顺序图中的约束信息以文字描述的方式存储于对象或消息中，约束演化就是直接对约束信息进行添加和删除。

- AC(Add Constraint)直接添加新的约束信息，会对架构设计产生直接的影响，需要判断当前设计是否满足新添加的约束要求。
- DC>Delete Constraint)直接移除某条约束信息，发生在去除某些不必要条件的时候，一般而言架构设计均会满足演化后的约束



软件架构演化方式的分类



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:
 - 第一类: 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
 - 第二类: 版本和工程的管理工具
 - 第三类: 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
 - 第四类: 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



软件架构演化方式的分类



软件架构静态演化主要是在设计时演化以及运行前演化。与此相对应的维护方法有3类：**更正性维护、适应性维护和完**
善性维护。



软件的静态演化一般包括如下5个步骤。

- **软件理解**：查阅软件文档，分析软件架构，识别系统组成元素及其之间的相互关系，提取系统的抽象表示形式。
- **需求变更分析**：静态演化往往是由于用户需求变化、系统运行出错和运行环境发生改变等原因所引起的，需要找出新的软件需求与原有的差异。
- **演化计划**：分析原系统，确定演化范围和成本，选择合适的演化计划。
- **系统重构**：根据演化计划对系统进行重构，使之适应当前的需求。
- **系统测试**：对演化后的系统进行测试，查找其中的错误和不足之处。



软件架构演化方式的分类



一次完整软件架构演化过程可以看作经过一系列原子演化操作组合而成。所谓原子演化操作是指基于UML模型表示的软件架构，在逻辑语义上粒度最小的架构修改操作。每经过一次原子演化操作，架构会形成一个演化中间版本。



架构演化的可维护性度量基于组件图表示的软件架构，在较高层次上评估架构的某个原子修改操作对整个架构所产生的影响。这些原子修改操作包括增加/删除模块间的依赖、增加/删除模块间的接口、增加/删除模块、拆分/聚合模块等。



架构演化的可靠性评估基于用例图、部署图和顺序图，分析在架构模块的交互过程中某个原子演化操作对交互场景可靠程度的影响。这些原子修改操作包括增加/删除消息、增加/删除交互对象、增加/删除/修改消息片段、增加/删除用例执行、增加/删除角色等。



软件架构演化方式的分类



动态演化是在系统运行期间的演化，需要在不停止系统功能的情况下完成演化，较之静态演化更加困难。具体发生在**有限制的运行时演化**和**运行时演化阶段**。



架构的动态演化主要来自两类需求：

- **软件内部执行所导致的体系结构改变**，例如，许多服务器端软件会在客户请求到达时创建新的组件来响应用户需求，假设有一个在线游戏服务器，它需要动态地创建新游戏房间以响应不同的玩家请求。当玩家请求加入一个新的游戏房间时，服务器会根据请求创建一个新的游戏房间
- **软件系统外部的请求对软件进行的重配置**，例如，操作系统在升级时无须重新启动，在运行过程中就完成对体系结构的修改。在操作系统中，内核是负责管理硬件资源和执行系统级任务的核心组件。当需要进行内核更新时，通常不需要重新启动整个计算机。相反，操作系统可以在运行时加载新的内核模块，然后逐渐切换到新内核，同时继续运行用户应用程序。这允许操作系统在不中断正在运行的应用程序的情况下对其体系结构进行修改和更新，提供了更高的可用性和服务连续性



软件的动态性分为3个级别：

- **交互动态性**，要求数据在固定的结构下动态交互；
- **结构动态性**，允许对结构进行修改，通常的形式是组件和连接件实例的添加和删除，这种动态性是研究和应用的主流；
- **架构动态性**，允许软件架构的基本构造的变动，即结构可以被重定义，如新的组件类型的定义。



软件架构演化方式的分类



根据所修改的内容不同，软件的动态演化主要包括以下4个方面。

- **属性改名：**目前所有的ADL都支持对非功能属性的分析和规约，而在运行过程中，用户可能会对这些指标进行重新定义，比如服务响应时间改为响应时间
- **行为变化：**在运行过程中，用户需求变化或系统自身服务质量的调节都将引发软件行为的变化。比如一个在线社交媒体平台，最初允许用户发布文字和图片。然后，用户需求发生变化，要求支持视频发布。
- **拓扑结构改变：**如增删组件，增删连接件，改变组件与连接件之间的关联关系等。比如一个分布式系统中有多个服务节点，它们相互连接以共同处理请求。随着系统的扩展，需要增加新的服务节点，同时删除一些不再需要的旧节点
- **风格变化：**一般软件演化后其架构风格应当保持不变，如果非要改变软件的架构风格，也只能将**架构风格变为其衍生风格**，比如一个企业级应用程序最初采用了两层客户端/服务器（C/S）架构，其中客户端直接与数据库交互。然后，由于需求的变化或性能问题，决定将架构改变为三层C/S架构，其中增加了应用服务器层来处理业务逻辑



软件架构演化方式的分类



目前，实现软件架构动态演化的技术主要有两种：采用动态软件架构(DSA)和进行动态重配置(DR)。

- DSA是指在运行时刻会发生变化的系统框架结构，允许在运行过程中通过框架结构的动态演化实现对架构的修改
- DR从组件和连接件的配置入手，允许在运行过程中增删组件，增删连接件，修改连接关系等操作



实现软件架构动态演化的基本原理是使DSA在可运行应用系统中以一类有状态、有行为、可操作的实体显式地表示出来，并且被整个运行环境共享，作为整个系统运行的依据。也就是说，运行时刻体系结构相关信息的改变可用来触发、驱动系统自身的动态调整。



系统必须提供SA动态演化的一些相关功能：保存当前软件架构信息的功能、设置监控机制监视系统有无需求变化、保证演化操作原子性。



DSA实施动态演化大体遵循以下4步：①捕捉并分析需求变化；②获取或生成体系结构演化策略；③根据步骤2得到的演化策略，选择适当的演化策略并实施演化；④演化后的评估与检测。



基于软件动态重配置的软件架构动态演化主要是指在**软件部署之后对配置信息的修改**，常常被用于系统动态升级时需要进行的配置信息修改。一般来说，动态重配置可能涉及的修改有：

- ①简单任的相关实现修改;② workflow实例任务的添加和删除：③组合任务流程中的个体修改：④任务输入来源的添加和删除：⑤任务输入来源的优先级修改：⑥组合任务输出目标的添加和删除：⑦组合任务输出目标的优先级修改等。



动态重配置模式：主从模式、中央控制模式、客户端/服务器模式、分布式控制模式。



软件架构演化原则

- 1.演化成本控制原则：保证成本可控
- 2.进度可控原则：保证进度可控
- 3.风险可控原则：保证风险可控
- 4.主体维持原则：保证软件系统主体行为稳定。
- 5.系统总体结构优化原则：保证演化之后的软件系统整体结构（布局）更加合理。
- 6.平滑演化原则：保证软件的演化速率趋于稳定，如相邻版本的更新率相对固定
- 7.目标一致原则：架构演化的阶段目标和最终目标要一致。
- 8.模块独立演化原则：保证要演化的模块相对独立，不要影响到其他模块
- 9.影响可控原则：软件中一个模块如果发生变更，其给其他模块带来的影响要在可控范围内
- 10.复杂性可控原则：演化必须要控制架构的复杂性，从而进一步保障软件的复杂性在可控范围内。
- 11.有利于重构原则：架构演化要遵循有利于重构原则，使得演化之后的软件架构更便于重构。
- 12.有利于重用原则：架构演化最好能维持，甚至提高整体架构的可重用性。
- 13.设计原则遵从性原则：架构演化最好不能与架构设计原则冲突
- 14.适应新技术原则：软件要独立于特定的技术手段，这样才能够让软件运行于不同平台。
- 15.环境适应性原则：架构演化后的软件版本能够比较容易适应新的硬件环境与软件环境
- 16.标准依从性原则：架构演化不会违背相关质量标准(国际标准、国家标准、行业标准、企业标准等)
- 17.质量向好原则：通过演化使得所关注的某个质量指标或某些质量指标的综合效果变得更好或者更满意，例如可靠性提高了。
- 18.适应新需求原则：架构演化之后要很容易适应新的需求变更



软件架构演化评估方法



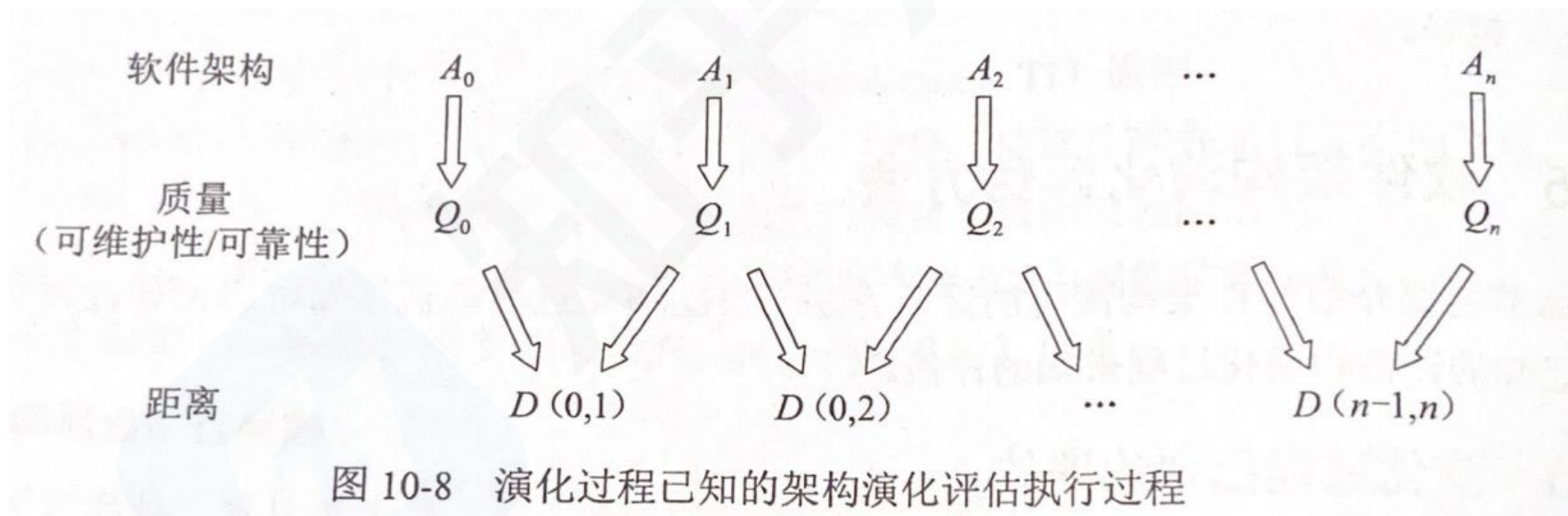
根据演化过程是否已知可将评估过程分为：**演化过程已知的评估（正向）**和**演化过程未知的评估（逆向）**。



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。





软件架构演化评估方法

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。具体包括：**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。

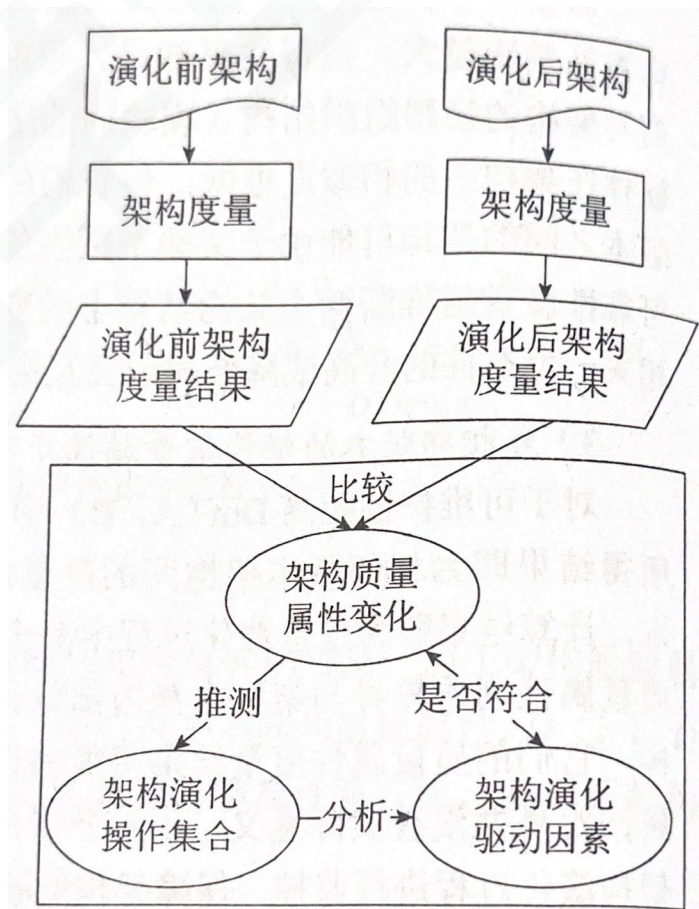


图 10-9 演化过程未知时的架构演化评估过程示意图



大型网站架构演化

第一阶段：单体架构

特点：所有业务在一个服务器上

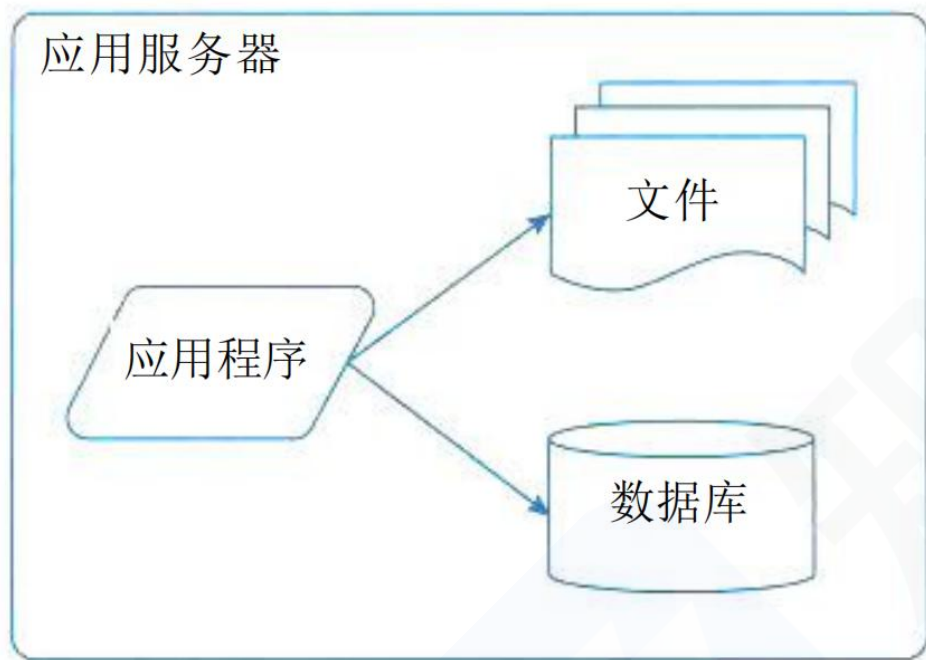


图10-10 第一阶段网站架构

第二阶段：垂直架构

特点：每种业务在一个单独服务器上

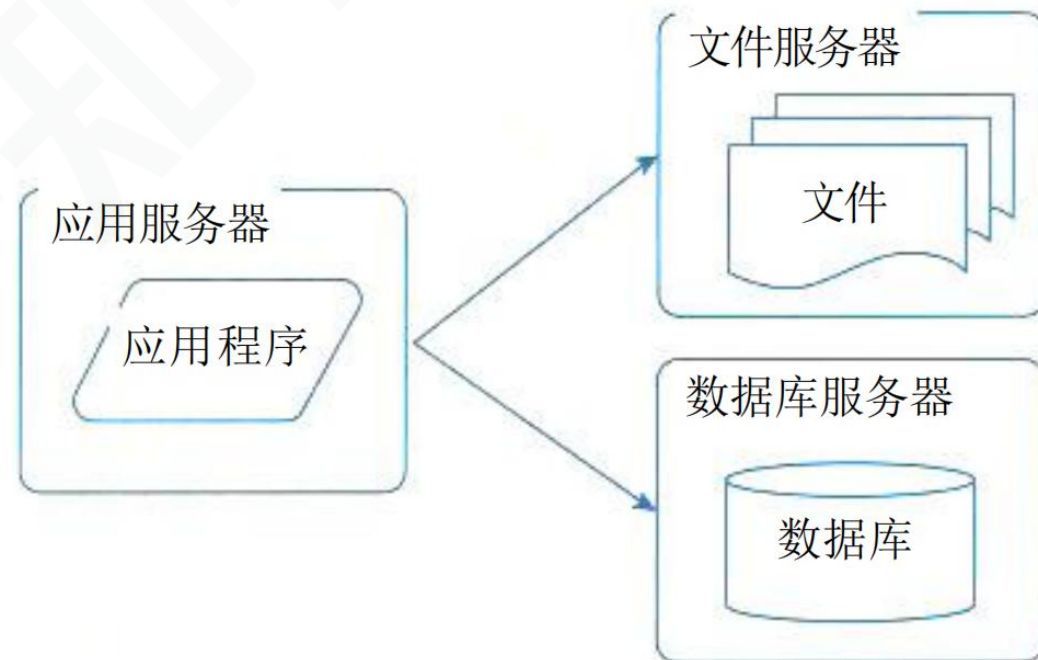


图10-11 第二阶段网站架构



大型网站架构演化

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

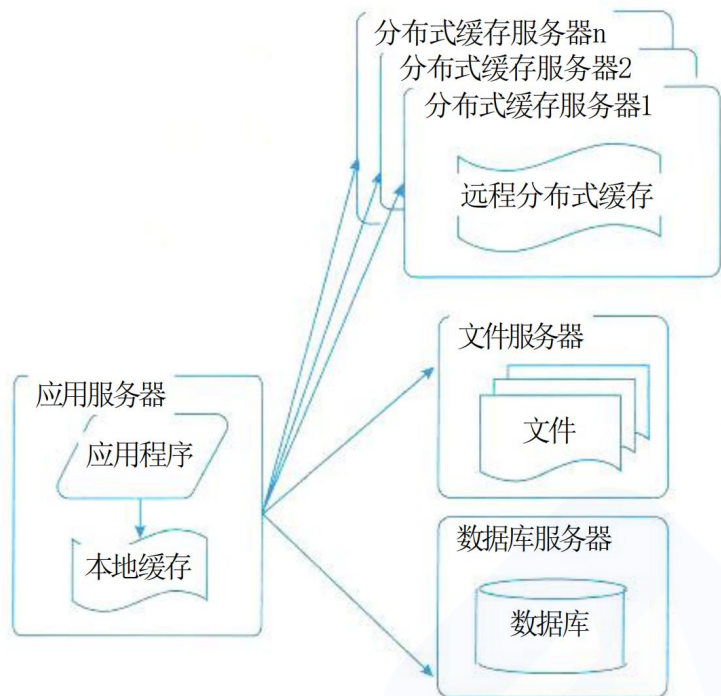


图10-12第三阶段网站架构

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

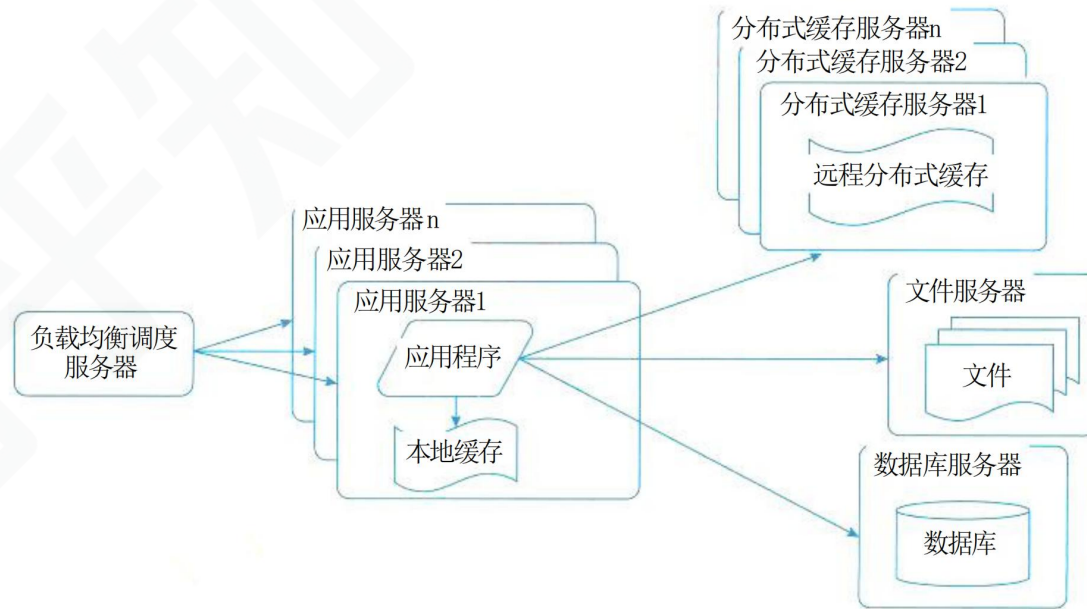


图10-13第四阶段网站架构



大型网站架构演化

第五阶段：数据库读写分离

特点：数据库拆分为主从数据库

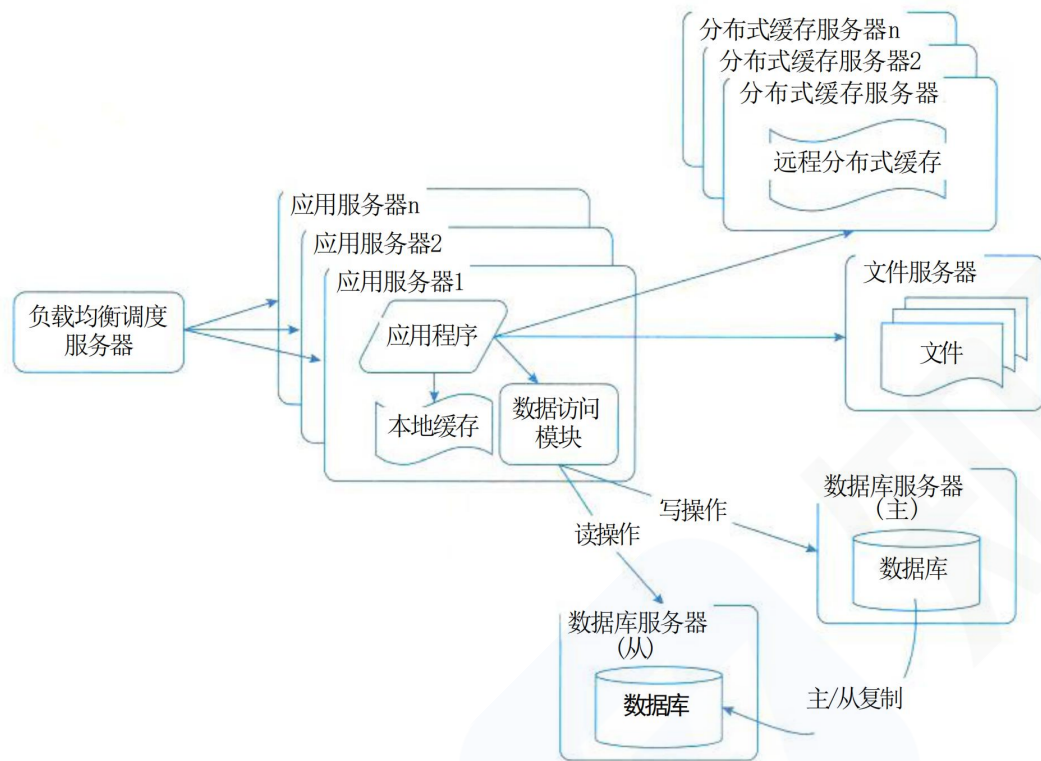


图10-14第五阶段网站架构

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

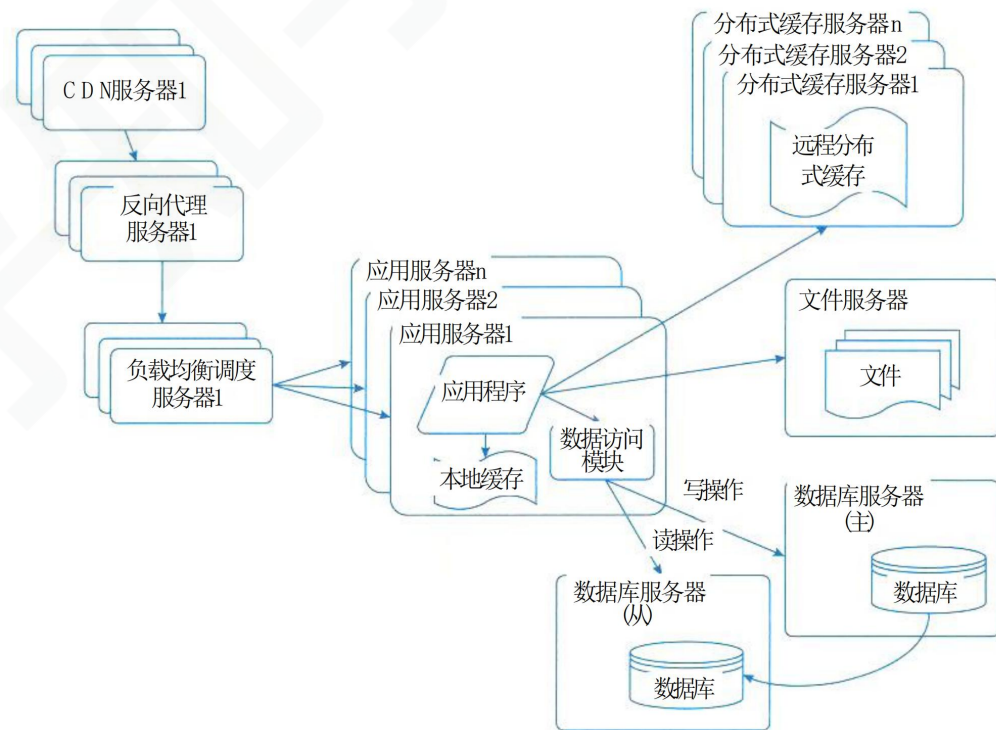


图10-15第六阶段网站架构

大型网站架构演化

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

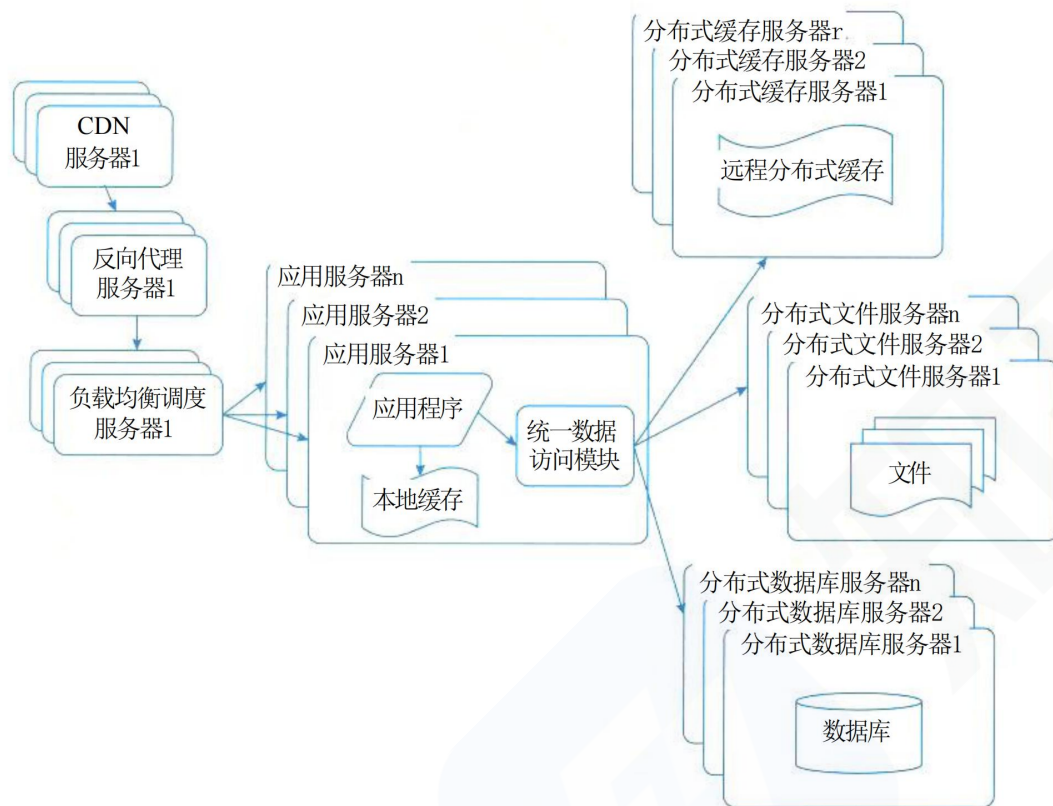


图10-16第七阶段网站架构

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

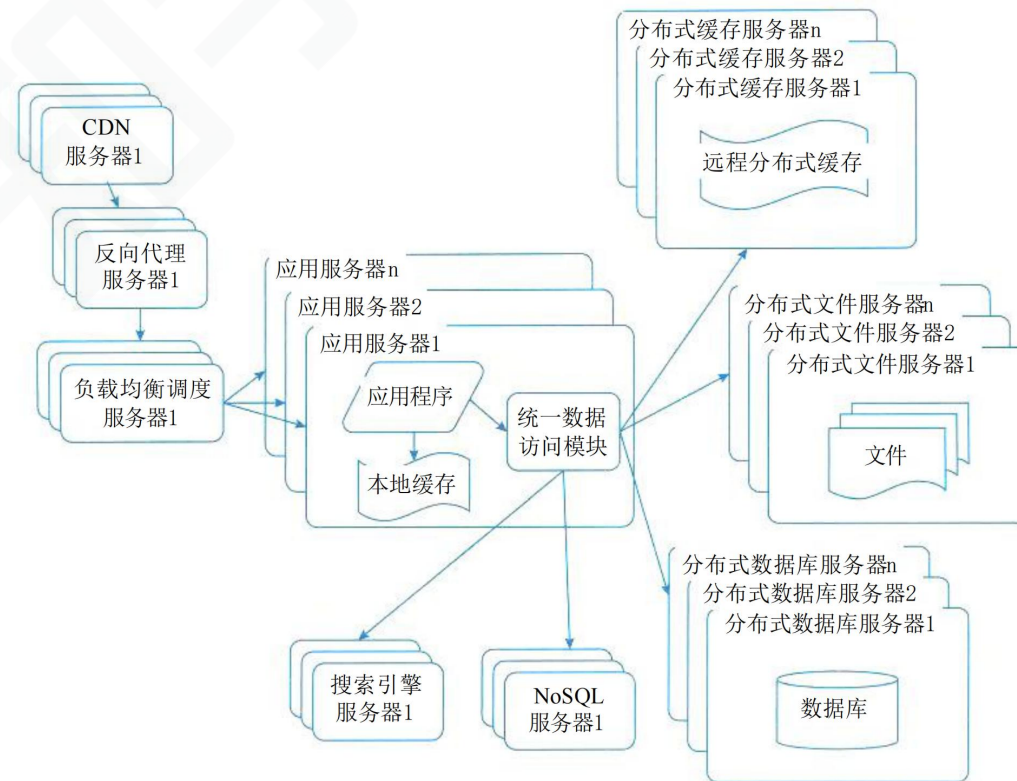


图10-17第八阶段网站架构



大型网站架构演化

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

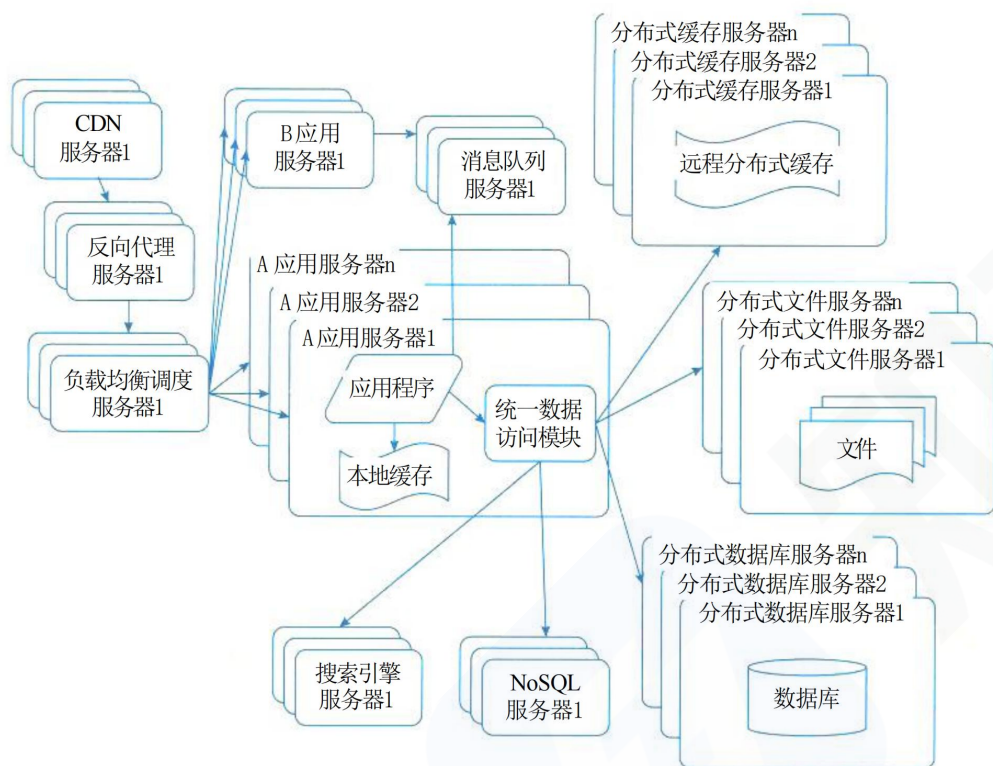


图10-18 第九阶段网站架构

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

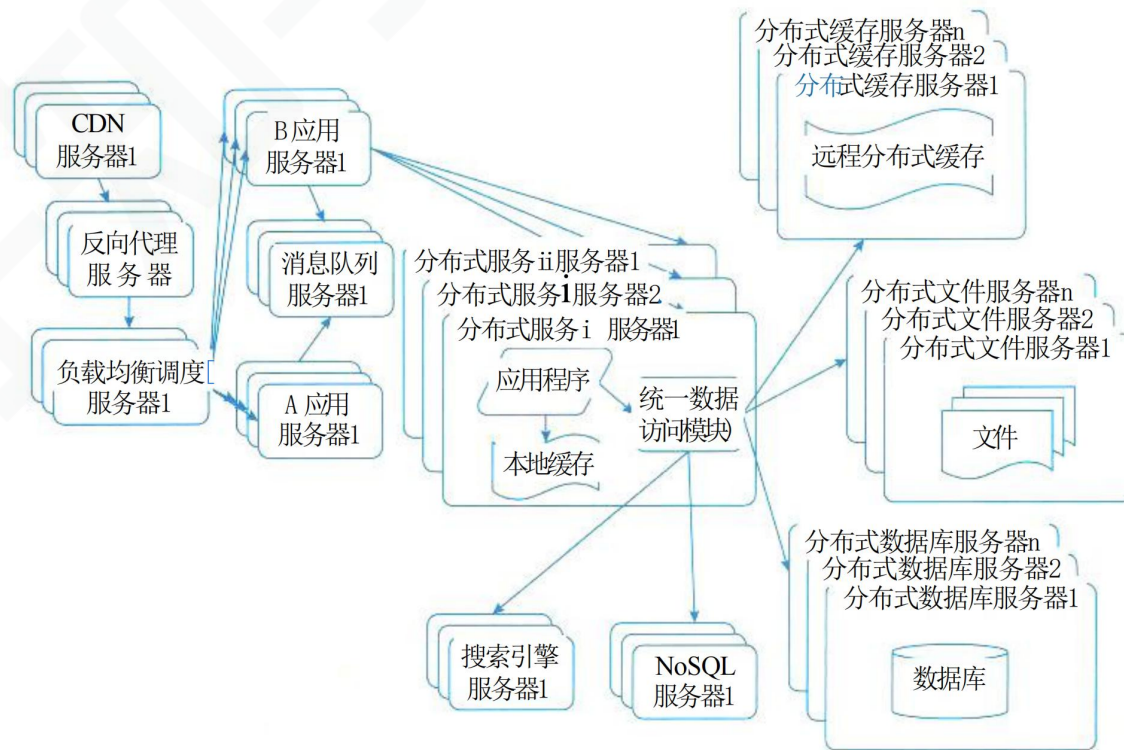


图10-19 第十阶段网站架构



软件架构维护



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。

T H E E N D

功不唐捐，玉汝于成！

• 开 启 新 征 程 •

