# Architecture

# DNS（域名系统）：庞大的分布式数据库；每天处理数万亿个查询；组织上、物理上分散；“防弹”：可靠性、安全性

大型网站架构

大型网站软件系统特点：⾼并发，⼤流量，⾼可用，海量数据，用户分布⼴泛，⽹络情况复杂，安全环境恶劣，需求快速变更，发布频繁，渐进式发展

⼤型⽹站架构演化的价值观:⼤型⽹站架构技术的核⼼价值是随⽹站所需灵活应对;驱动⼤型⽹站技术发展的主要⼒量是⽹站的业务发展

⽹站架构设计误区:⼀味追随⼤公司的解决⽅案;为了技术⽽技术;企图用技术解决所有的问题

大型网站架构发展历程：**初始阶段**（应用程序、数据库、文件等所有的资源都在一台服务器上）->**应用服务和数据服务分离**（不同特性的服务器承担不同的服务角色，网站的并发处理能力和数据存储空间都得到了很大改善，但I/O还是慢）->**使用缓存改善网站性能**（使用缓存之后，数据访问压力得到有效缓解，但是单一应用服务器能够处请求连接有限，在网站访问高峰期，应用服务器成为整个网站的瓶颈）->**使用应用服务器集群改善网站的并发处理能力**（通过负载均衡调度服务器，可将来自用户浏览器的访问请求分发到应用服务器集群中的任何一台服务器上，服务器的负载压力不再成为整个网站的瓶颈。部分数据库读操作（缓存访问不命中、缓存过期）和全部的写操作需要访问数据库。当负载压力过高时，数据库成为瓶颈。）->**数据库读写分离**（应用服务器在写数据的时候，访问主数据库，主数据库通过主从复制机制将数据更新同步到从数据库。应用服务器在读数据的时候，直接通过从数据库获得数据）->**使用反向代理和CDN加速网站响应**（CDN和反向代理的基本原理都是缓存。加快用户访问速度，减轻后端服务器的负载压力。CDN部署在网络提供商的机房，使用户在请求网络服务时，可以从距离自己最近的网络提供商机房获取数据。反向代理部署在网站的中心机房，当用户请达到中心机房后，反向代理服务器中缓存着用户请求的资源，就直接返回给用户。）->**使用分布式文件系统和分布式数据库系统**->**使用NoSQL和搜索引擎**->**业务拆分**->**分布式服务**

流媒体应用架构

流媒体主要挑战:1.用户规模大：1B2.异构性：不同的用户具有不同的功能（例如，有线与移动设备；带宽丰富与带宽差）3.解决：分布式应用程序架构4.服务器到客户端的带宽会随着时间的推移而变化，网络拥塞程度（内部、接入网、网络核心、视频服务器）也会变化5.数据包丢失，拥塞造成的延迟会延迟播放，或导致视频质量差6.连续播放约束：在客户端视频播放过程中，播放时间必须与原始时间相匹配……但是网络延迟是可变的（抖动），因此需要客户端缓冲区来匹配连续播放约束7.其他挑战：客户端交互性：暂停、快进、回带、跳过视频视频包可能会丢失、重传8.播出缓冲：客户端缓冲和播放延迟：补偿网络增加的延迟，延迟抖动

**CDN**（Content Distribution Network）是一种内容分发网络，它通过在多个地理位置部署服务器来存储内容的副本，从而加速内容的传输和提高可靠性[CDN的目的是尽可能地将内容靠近用户，减少延迟和提高传输速度。这对于视频流服务尤其重要，因为它可以显著提高视频播放的流畅度和减少缓冲时间](https://cloud.baidu.com/article/3107524)。CDN通过将内容分发到接近用户的服务器上来加速内容的传输，CDN：在CDN节点上存储内容的副本，订阅者请求内容，服务提供者返回清单，使用清单，客户端以最高可支持的速率检索内容，如果网络路径拥塞，可能会选择不同的速率或复制

OTT(OVER THE TOP)挑战：从“边缘”应对拥挤的互联网（在哪个CDN节点中放置什么内容？从哪个CDN节点检索内容？以何种速度？）

流化多媒体

服务器：1.将视频文件分成多个块2.每个块以多个不同的比率编码3.不同的速率编码存储在不同的文件

4.文件复制在不同的CDN节点5.清单文件：为不同的块提供url

客户端：1.定期估计服务器到客户端的带宽2.咨询清单，每次请求一个块选择最大的编码率，给定当前的带宽3.可以从不同的服务器节点选择不同的编码率（取决于时间点的可用带宽）4.“智能”客户端：客户端决定何时请求块（这样缓冲区饥饿，或溢出不发生）5.编码率请求（更高质量时更多的带宽）

6.请求块（可以请求服务器离客户端更近的URL或高可用带宽）7.流视频=编码+ DASH +播放缓冲

**DASH**（Dynamic Adaptive Streaming over HTTP）是一种自适应比特率流技术，它允许视频内容提供者将视频编码成多个不同质量的版本，并存储在服务器上。每个版本都有一个URL，用户可以根据自己的网络条件选择不同质量的视频流[1。这样，即使在网络条件不佳的情况下，用户也能获得较好的观看体验，因为DASH会自动调整视频质量以适应当前的网络速度](https://blog.csdn.net/HZRll/article/details/135358089)[2](https://blog.csdn.net/LvGreat/article/details/121252994)。DASH通过提供多个视频质量版本来优化用户的观看体验。

客户端-服务器Client-Server：在客户端-服务器架构中，客户端发出请求，服务器响应许多客户端请求。客户机-服务器捕获专门服务器和客户机的可重用行为，而不指定如何实现服务器和客户端。

点对点Peer-to-Peer：在点对点架构中，所有实体都充当客户端和服务器。虽然很灵活，但它使得很难实现软件来做好任何一个工作。1.没有总一直在线提供服务的服务器2.任意端系统直接通信3.同行从其他同行请求服务，向其他同行提供服务4.自我扩展——新同行带来新的服务能力，和新的服务需求

5.同行间歇连接和改变IP地址6.例子： P2P文件共享（bt），流（KanKan），VoIP（Skype）

Client-Server文件分发时间：**服务器传输**：必须按顺序发送（上载）N个文件副本：发送一份副本的时间：F/us；发送N份副本的时间：NF/us**客户端**：每个客户端必须下载文件副本，dmin=最小客户端下载速率；最短客户端下载时间：F/dmin **分发F的时间到N个客户端使用客户端-服务器方法Dc-s>=max｛NF/us，F/dmin｝**

P2P文件分发时间：**服务器传输**：必须至少上载一个副本：发送一份副本的时间：F/us；**客户端**：每个客户端必须下载文件副本：最短客户端下载时间：F/dmin；**客户端**：as聚合必须下载NF位：最大上传速率（限制最大下载速率）为us+∑ui**使用P2P方法将F分配给N个客户端的时间DP2P>最大｛F/us，F/dmin，NF/（us+∑ui）｝**

Software as a service(SaaS) & Cloud Computing:1.现成软件-软件即服务（SaaS）：**数据安全存储在云中**（丢失设备≠丢失数据；小组可以轻松共享/协作）；**1个软件拷贝，单个硬件/操作系统环境**（对开发人员和用户来说没有兼容性问题；在1%的用户上测试新功能？；升级==部署到云）**持续的客户反馈是可能的，通过快速推出新的更改做出响应**（永久测试版；敏捷的理想匹配？）2.SaaS对基础设施的三大需求：沟通；可扩展性和弹性；可靠性（“24 x 7”）3.集群上的服务：**（大多数）通过商品以太网交换机连接的商品计算机：**比传统服务器更具可扩展性；比传统服务器便宜得多（相当于最大服务器的20倍）；通过大量冗余实现可靠性；1000s服务器的运营商很少（仔细选择相同的硬件/软件；虚拟机监视器和容器化（Docker、Kubernetes）简化了操作）

Architecture Design Principle

Software Design:The process of defining the architecture, components,interfaces, and other characteristics of a system or component and the results of [that] process.定义体系结构、组件的过程，接口和系统的其他特性，或者组成部分；[那个]过程的结果。

为什么软件架构重要：体系结构充当系统的骨架（没有单一的正确架构，但或多或少有适合这份工作的框架）；体系结构影响质量属性（除了支持所需的功能外，系统的体系结构还可以启用或禁用质量。）

设计过程：设计的逻辑：寻找替代方案；设计的形状：层次；过程是风格的决定因素

设计结果/目的：从软件的“全局”视图到定义实现系统所需细节的更窄视图,体系结构;子系统;通信机制;组件;外部、内部和用户接口

架构设计目标/架构复杂性根源Goals of Architecture Design:High Performance高性能High Availability高可用性High Scalability高扩展性（容量，改变）(Capacity,Change)High Heterogeneity高度异质性High Security高安全性Low Cost低成本

Design Principles

KISS(Keep it simple stupid):1.算法，结构，接口，执行，交互2.简化接口3.解耦模块依赖关系4.即插即用5.混搭

Modularity:1.每个模块都简单到可以让人完全理解2.一个模块的变化，并不影响另一个3.一个模块的修改尽量不要修改接口4.一个模块修改好了之后可以和其他新老模块一起工作，这也是现在敏捷开发和持续交付的基础工作。5.将复杂的系统划分为更简单的组件（强内聚，松耦合）6.预期的好处:开发可以更容易地计划;软件增量可以定义和交付;更改可以更容易地适应;测试和调试可以更有效地进行;长期维护可以在没有严重的副作用的情况下进行

Separation of concern（各司其职）:1.许多决策都是密切相关和相互依赖的2.分别集中于问题的不同方面，例如软件质量、开发过程、操作约束。3.分离可能基于时间、质量、视图、大小、细节水平等。

4.横切需求:它们的实现跨越了许多程序组件;当必须对问题进行更改时，这将导致问题——要更改的代码不是本地化的，而是在整个系统的不同位置5.解决复杂问题的系统思维:分解：将复杂问题分解为子问题，小的、简单的、可控的;综合：将子问题的解组合成一个集成的解

Design for change:1.目的:会增加新的功能，可以改变，甚至取消旧的功能;来修复缺陷，加强安全性，或提高性能;来增强用户的使用体验;接受新技术、新平台、新协议、新标准;使系统一起工作，即使它们从未被设计成这样做。2.策略是将可能的变化作为系统设计的一部分3.隔离软件特定部分可能的变化，这样变化将被限制在小部分3.方便修改的软件设计：设计的不是一个程序，而是一系列程序……我们想要利用共性，共享代码，并降低维护成本，尝试预测可能做出的改变，并被展示如何实现当这些预期的变化发生时很容易改变的设计。4.(1)它们可能运行在不同的硬件配置上。(2)它们可以执行相同的功能，但在格式或输入和输出数据上有所不同。(3)由于可用资源的不同，它们在某些数据结构或算法上可能会有所不同。(4)由于输入数据集的大小或某些事件的相对频率的不同，它们可能在某些算法的数据结构上有所不同。(5)有些用户可能只需要其他用户所需要的服务或功能的一个子集。这些‘要求较低’的用户可能会要求他们不必被迫为不需要的功能所消耗的资源付费。”5.“可修改性”与变更的成本有关，指的是软件系统适应变更的难度。

Design for reuse:1.为了避免“重新发明轮子”，不要重复自己，和避免每次写2.软件是昂贵的，软件资产的重用对于提高投资回报率（投资回报）至关重要:降低生产和维护成本;更快地交付;提高质量3.软件可重用资产需求:设计、代码、测试用例;问题解决方案、设计解决方案;工具;过程;经验和最佳实践4.采用成熟的开源组件

软件产品线（或应用程序系列）:1.一组软件密集型系统，共享一套共同的、受管理的特性，满足特定市场或任务的特定需求，并通过规定的（规定的）从一组共同的核心组合方式开发。2.一组具有公共的、特定于领域的架构的相关应用程序3.每次生成新应用程序时都会重复使用公共核心资产库。4.每个应用程序都是核心资产的实例化，以某种方式专门化s

三个关键活动：核心资产开发，产品开发，管理经营:1.反馈循环:核心+定制+管理;产品由核心资产构建;资产被提取，演变，和推广;有预见性的管理，以投资资源，协调新的生产和资产可用性2.迭代:核心资产积累；产品创建；协调管理经营

Architecture Styles(Pattern)：架构设计的主要目的是为了解决软件系统复杂度带来的问题

架构设计三原则：合适优于业界领先，简单优于复杂，演化优于一步到位

性能概述：1.性能是软件系统满足时间要求的能力的。2.它是软件系统普遍存在的软件系统：一切都影响它，从软件本身到所有底层，如操作系统、中间件、硬件、通信网络等。3.在软件工程的许多历史中，性能一直是系统架构的驱动因素。

不同视角下的网站性能：用户视角（用户在浏览器上直观感受到的网站响应速度；1.用户计算机和网站服务器通信的时间；2.网站服务器处理的时间；3.用户计算机浏览器构造请求解析响应数据的时间）开发人员视角（应用程序本身及其相关子系统的性能。1.响应延迟2.系统吞吐量3.并发处理能力4.系统稳定性）维护人员视角（基础设施性能和资源利用率。1.网络运行商的带宽能力2.服务器硬件的配置3.数据中心网络架构4.服务器和网络带宽的资源利用率）

性能指标：1.响应时间2.性能计数器：服务器或是操作系统的数据指标；系统负载（正在被CPU执⾏和等待被执⾏的进程数目总和），对象与线程数，内存使用，，CPU使用，磁盘与⽹络I/O3.并发数：系统能同时处理请求的数量；注册用户数 >> 在线用户数 >> 并发用户数4.吞吐量：单位时间内系统处理的请求数量，体现系统的整体处理能⼒；请求数/秒，页面数/秒，访问⼈数/天，处理的业务数/小时；TPS（每秒事务数），HPS（每秒HTTP请求数），QPS（每秒查询数）

高性能举例：1.高性能关系数据库读写分离：将数据库读写操作分散到不同的节点上2.高性能关系数据库集群业务分库：按照业务模块将数据分散到不同的数据库服务器（一般来说，单台数据库服务器能够支撑 10 万用户量量级的业务，分库能够支撑百万甚至千万用户规模的业务，缺点：操作复杂；代码工作量变大；计算资源变高）；3.高性能短息数据库集群分表：垂直分表适合将表中某些不常用且占了⼤量空间的列拆分出去，但查询次数可能变多，⽔平分表适合表⾏数特别⼤的表 （千万级），范围路由、hash路由、配置路由4.高性能NoSQL：K-V 存储：解决关系数据库无法存储数据结构的问题，以 Redis 为代表；文档数据库：解决关系数据库强 schema 约束的问题，以 MongoDB 为代表；列式数据库：解决关系数据库大数据场景下的 I/O 问题，以 HBase 为代表；全文搜索引擎：解决关系数据库的全文搜索性能问题，以 Elastic Search 为代表；图数据库： 解决图数据的性能问题， 以Neo4j为代表

高性能缓存架构：需要经过复杂运算后得出的数据，存储系统⽆能为⼒；读多写少的数据，存储系统有⼼⽆⼒

缓存系统需要解决的问题

缓存穿透：问题：缓存穿透是指缓存和数据库中都没有的数据，⽽用户不断发起请求，如发起为id为“-1”的数据或id为特别⼤不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压⼒过⼤。解决⽅案：接⼝层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防⽌攻击用户反复用同⼀个id暴⼒攻击

缓存雪崩：问题：当缓存失效（过期）后引起系统性能急剧下降的情况。当缓存过期被清除后，业务系统需要重新⽣成缓存，因此需要再次访问存储系统，再次进⾏运算，这个处理步骤耗时⼏⼗毫秒甚⾄上百毫秒。⽽对于⼀个⾼并发的业务系统来说，⼏百毫秒内可能会接到⼏百上千个请求。由于旧的缓存已经被清除，新的缓存还未⽣成，并且处理这些请求的线程都不知道另外有⼀个线程正在⽣成缓存，因此所有的请求都会去重新⽣成缓存，都会去访问存储系统，从⽽对存储系统造成巨⼤的性能压⼒。这些压⼒又会拖慢整个系统，严重的会造成数据库宕机，从⽽形成⼀系列连锁反应，造成整个系统崩溃。解决⽅案：缓存数据的过期时间设置随机，防⽌同⼀时间⼤量数据过期现象发⽣；如果缓存数据库是分布式部署，将热点数据均匀分布在不同的缓存数据库中；设置热点数据永远不过期

缓存热点：问题：虽然缓存系统本身的性能比较⾼，但对于⼀些特别热点的数据，如果⼤部分甚⾄所有的业务请求都命中同⼀份缓存数据，则这份数据所在的缓存服务器的压⼒也很⼤。解决⽅案：复制多份缓存副本，将请求分散到多个缓存服务器上，减轻缓存热点导致的单台缓存服务器压⼒。缓存的数据是⼀样的，通过在缓存的 key 里面加上编号进⾏区分，每次读缓存时都随机读取其中某份缓存；缓存副本设计有⼀个细节需要注意，就是不同的缓存副本不要设置统⼀的过期时间，否则就会出现所有缓存副本同时⽣成同时失效的情况，从⽽引发缓存雪崩效应。正确的做法是设定⼀个过期时间范围，不同的缓存副本的过期时间是指定范围内的随机值。

使用服务器集群：在⽹站⾼并发访问的场景下，使用负载均衡技术为⼀个应用构建⼀个由多台服务器组成的服务器集群，将并发访问请求分发到多台服务器上处理，避免单⼀服务器因负载压⼒过⼤⽽响应缓慢，使用户请求具有更好的响应延迟特性。

高性能负载均衡：算法：1.轮询：服务器不宕机、服务器与负载均衡器不断连2.加权轮询：服务器性能有差异3.负载最低优先：连接数、请求数、CPU负载、I/O负载，需要采样4.性能最优优先：处理速度，需要采样5.Hash：源地址，ID

WEB前端性能优化:1.浏览器访问优化：减少http请求，使用浏览器缓存，启用压缩，CSS放在页面最上面，JavaScript放在页面最下面，减少Cookie传输2.CDN加速： CDN (Content Distribute Network，内容分发⽹络)，将数据缓存在离用户最近的地⽅，使用户以最快速度获取数据，即所谓⽹络访问第⼀跳3.反向代理：传统代理服务器位于浏览器⼀侧，反向代理位于⽹站机房⼀侧，代理⽹站Web服务器接收HTTP请求。

应用服务器性能优化:1.异步操作2.使用消息队列将调用异步，不使用消息队列，用户的请求数据直接写⼊数据库，在⾼并发的情况下，会对数据库造成巨⼤的压⼒，响应延迟加剧。3.使用消息队列后，用户请求的数据发送给消息队列后立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写⼊数据库。4.消息队列具有很好的削峰作用，通过异步处理，将短时间⾼并发产⽣的事务消息存储在消息队列中，从⽽削平⾼峰期的并发事务。

可用性:是指系统屏蔽或修复故障的能力，以使累积的服务中断期在指定的时间间隔内不超过所要求的值。

容错性:分布式系统即使在低水平的硬件/软件/网络可靠性下也可以保持可用性，容错不应该涉及用户或系统管理员，这是通过恢复（例如，依赖等效服务）和复制组件来实现的

CAP定理：在分布式系统（共享数据的互连节点的集合）中，在写/读对中只能有以下三种保证中的两种：一致性、可用性和分区容差-必须牺牲其中一种（一致性（consistency）：每次读取都会收到最近的写入或错误；可用性（availability）：每个请求都会收到一个（无错误）合理的响应，但不能保证它包含最新的写入；分区容差（partition tolerance）：尽管节点之间的网络丢弃（或延迟）了任意数量的消息，但系统仍能继续运行）：如果我们选择了CA⽽放弃了P，那么当发⽣分区现象时，为了保证C，系统需要禁⽌写⼊；当有写⼊请求时，系统返回 error（例如，当前系统不允许写⼊），这又和A冲突了，因为A要求返回 no error 和 no timeout。因此，分布式系统理论上不可能选择CA 架构，只能选择CP 或者AP 架构。

FMEA(故障模式与影响分析):给出初始的架构设计图->假设架构中某个部件发生故障->分析此故障对系统功能造成的影响->根据分析结果，判断架构是否需要进行优化

高可用存储-双机：主备复制：数据变更频率低；主从复制: 适合写少读多 （1：10，1：100） 的业务；主主复制：适合于那些临时性、可丢失、可覆盖的数据场景

高可用存储-双机-主备切换：状态监测和状态传递（状态：例如机器是否掉电、进程是否存在、响应是否缓慢等）切换时机（什么情况下备机应该升级为主机？是机器掉电后备机才升级，还是主机上的进程不存在就升级，还是主机响应时间超过 2 秒就升级，还是 3 分钟内主机连续重启 3 次就升级等。）切换策略（原来的主机故障恢复后，再次切换，确保原来的主机继续做主机，还是原来的主机故障恢复后自动成为新的备机？）自动程度（切换是完全自动的，还是半自动的？例如，系统判断当前需要切换，但需要人工做最终的确认操作（例如，单击一下“切换”按钮））

高可用储存—数据分区：将数据按照一定的规则进行分区，不同分区分布在不同的地理位置上，每个分区存储一部分数据，通过这种方式来规避地理级别的故障所造成的巨大影响

可扩展性：通过修改和扩展，不断地让软件系统具备更多的功能和特性，满⾜新的需求或者顺应技术发展的趋势

拆: 就是将原本⼤⼀统的系统拆分成多个规模小的部分，扩展时只修改其中⼀部分即可，⽆须整个系统到处都改，通过这种⽅式来减少改动范围，降低改动风险。

面向流程拆分：将整个业务流程拆分为⼏个阶段，每个阶段作为⼀部分。1.**展示层**：负责⽤户⻚⾯设计，不同业务有不同的⻚⾯。例如，登录⻚⾯、注册⻚⾯、信息管理⻚⾯、安全设置⻚⾯等。2.**业务层**：负责具体业务逻辑的处理。例如，登录、注册、信息管理、修改密码等业务。3.**数据层**：负责完成数据访问。例如，增删改查数据库中的数据、记录事件到⽇志⽂件等。4.**存储层**：负责数据的存储。例如，关系型数据库 MySQL、缓存系统 Memcache 等。5.**扩展时大部分情况只需要修改某一层，少部分情况可能修改关联的两层，不会出现所有层都同时要修改。**

面向服务拆分：将系统提供的服务拆分，每个服务作为⼀部分。1.⾯向服务拆分将系统拆分为注册、登录、信息管理、安全设置等服务2.对某个服务扩展，或者要增加新的服务时，只需要扩展相关服务即可，⽆须修改所有的服务。3.如果我们需要在注册服务中增加⼀种“学号注册”功能，则只需要修改“注册服务”和“登录服务”即可，“信息管理服务”和“安全设置”服务⽆须修改。

面向功能拆分：将系统提供的功能拆分，每个功能作为⼀部分。1.每个服务都可以拆分为更多细粒度的功能2.对某个功能扩展，或者要增加新的功能时，只需要扩展相关功能即可，⽆须修改所有的服务。

大小对比：从范围上来看，从⼤到小依次为：流程 > 服务 > 功能

面向流程拆分：分层架构：1.CS:划分的对象是**整个业务系统**，划分的维度是⽤户交互，即将和⽤户交互的部分独⽴为⼀层，⽀撑⽤户交互的后台作为另外⼀层2.**MVC**: 划分的对象是**单个业务⼦系统**，划分的维度是职责，将不同的职责划分到独⽴层，但各层的依赖关系⽐较灵活，MVC框架强制性地把各层的实现功能划分开，各自处理各自的任务，利于解耦合，极⼤地增强了代码的可读性、维护性。M（Model）模型层——与数据库交互的数据模型，进⾏数据相关操作，V（View）视图层——与用户交互的数据区，负责收集展示数据，C（Controller）控制层——收到请求后，调用模型层与数据库交互获取数据，最后将数据返回给视图层（用户）—— 负责接收数据和逻辑处理，当用户通过浏览器发送请求到视图层(view)时，控制层(controller)分发调用模型层（model），进⾏数据库查询（如果查询失败便新建），接下来模型层（model）再将数据库查询到的数据返回给控制层(controller)，控制层(controller)再将其返回给视图层(view)，view层通过web页面把数据信息显示给用户。3.特点：保证各层之间的差异足够清晰，边界足够明显，让人看到架构图后就能看懂整个架构；隔离关注点（separation of concerns），即每个层中的组件只会处理本层的逻辑， 从而能够较好地支撑系统扩展；层层传递，相邻层依赖，一般不跨层跳跃， 从而降低整体系统复杂度

面向服务拆分：SOA(service oriented architecture)、微服务：微服务和SOA交集为服务，且企业服务总线（ESB）完全属于SOA，服务粒度SOA粗微服务细，服务通信SOA重量级ESB而微服务轻量级（Http restful），服务交付SOA慢 微服务快，应用场景SOA企业级而微服务互联网

微服务架构：微服务体系结构风格是一种将单个应用程序开发为一套小服务的方法，每个小服务都在自己的进程中运行，并与轻量级机制进行通信，通常是一个HTTP资源API。这些服务是围绕业务能力构建的，并可通过完全自动化的部署机制进行独立部署。

单体应用与微服务应用的区别：1.单体应用将自己所有功能放在同一进程中，通过在多台服务器上复制自身（单体应用）而扩展；而微服务应用将每个元素功能放在不同的进程中，通过将这些服务在多台服务器上进行分配而扩展（如果有需要就进行复制）2.模块部署：单体将多个模块放在同一进程，而微服务应用的不同模块运行在不同过程中3.数据库部署：单体应用为单个数据库，微服务应用为应用数据库

容器提供了一种以可以在共享操作系统上运行隔离服务的格式打包软件的方法。使软件工作所需的库和设置，具有轻量级、独立、标准、安全的系统，保证软件将始终运行相同的功能

达到目的的方法：实现持续交付/部署：特性：1.通过已发布的接口使用服务作为积木构建块（组件）2.围绕业务能力进行组织3.开发团队完全负责生产过程中的软件4.智能端点和哑管道（dumb pipes）5.对语言的分散控制

微服务细节：1.分散数据库：每个服务都管理自己的数据库，要么是相同数据库技术的不同实例，或者是完全不同的数据库系统——这种方法称为多语种持久性。2.基础设施自动化3.针对故障的设计：微服务团队将希望为每个单独的服务看到复杂的监控和日志记录，如显示启动/停机状态的仪表板以及各种操作和业务相关指标。4.进化设计：将服务分解作为进一步的工具，使应用程序开发人员能够控制应用程序中的更改，而不减缓更改。微服务系统可以有独立的替代性和可升级性。5.四代微服务架构：container orchestration（和谐的结合）；service discovery and fault tolerance；sidecar and service mesh(网络)；serverless architecture

面向功能拆分：微内核架构（Microkernel Architecture），也被称为插件化架构（Plug-in Architecture），是⼀种⾯向功能进⾏拆分的可扩展性架构，通常⽤于实现基于下载安装的应用,Browser,Eclipse，VSCode 等IDE 软件,UNIX 这类操作系统,淘宝 App 客户端软件,保险公司的保险核算逻辑系统，不同的保险品种可以将逻辑封装成插件

可以**组合使用：**以学⽣管理系统为例：整体系统采用面向服务拆分中的“微服务”架构，拆分为“注册服务”“登录服务”“信息管理服务”“安全服务”，每个服务是⼀个独立运⾏的⼦系统，其中的“注册服务”⼦系统本身又是采用面向流程拆分的分层架构。 “登录服务”⼦系统采用的是面向功能拆分的“微内核”架构。