



NEW PLAN

软件高级架构师

• 一段新征程 •

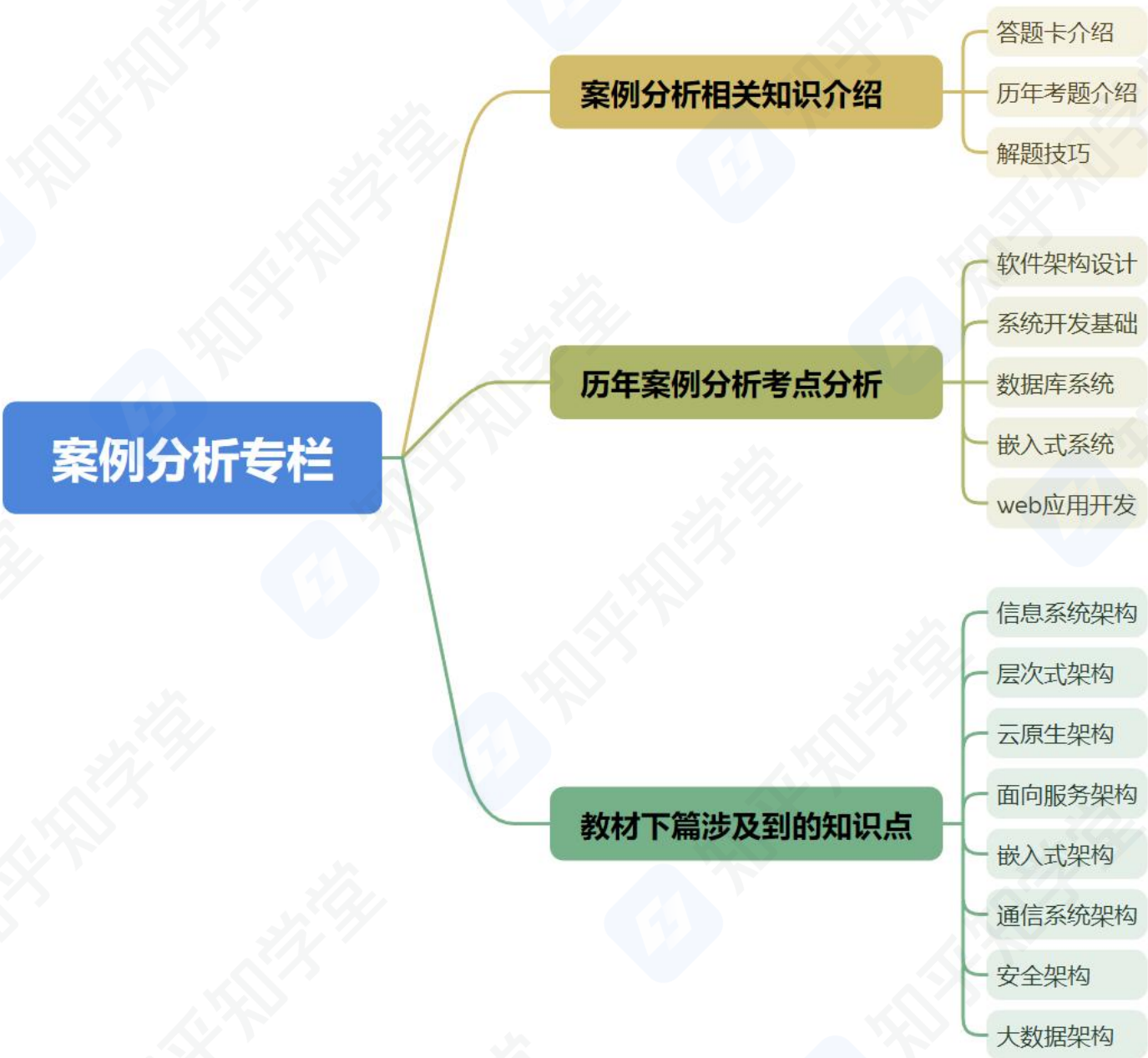




大纲介绍



知乎知学堂



1.下篇八大架构我们重点讨论除了**层次架构、云原生架构、面向服务架构以及安全架构**，其他的信息系统架构、通信系统架构、大数据架构和嵌入式架构同学们只需要过一遍视频了解一些基本的理论知识即可。

2.这些架构里面主要包含两方面的内容：知识点、架构案例图，对于知识点部分，现在有可能会出现在选择题中，但是比重不大，大概**1-3分**，老师已经把重点地方保留了，至于架构案例图，老师觉得不太可能会考到（虽然有一次考试直接考了大数据架构的原题），同学们稍微留意下即可；

3.八大架构的简单总结：

- 信息系统架构：了解一些基本概念即可
- **层次架构**：重点，不管是相关概念以及图例都要进行掌握
- **云原生架构**：重点，主要是有助于理解这个架构从而方便我们去书写对应的论文，虽然也有架构图，但是重要程度次之
- 通信系统架构：了解一些基本概念即可
- **面向服务架构**：重点，很多同学不理解什么是**SOA**，以及搞不清楚它和微服务的区别，所以这里主要是了解**SOA**架构所涉及的特点以及实现方式，方便出到选择题会进行选择，以及防止出现论文，这里没有架构图
- **安全架构**：次重点，了解一些安全的常识以应对选择题里面出现的安全超纲题目，里面也提到了**2个**架构图，但是都是在层次架构的基础上引入安全的管理和涉及理念，老师认为不大可能单独考安全架构的案例分析，但是完全有可能在考其他案例分析题的时候加入安全架构的知识来作为单独的一问
- 大数据架构：熟悉，之前已经连续**2次**考试都考到了大数据架构，分别是考了原图的案例分析以及考了论文，所以案例分析再考概率已经不高了，论文考到数据相关的题目并且涉及到大数据的可能还是有的，所以我们只需要掌握大数据架构的一些常识理论即可
- 嵌入式架构：了解一些基本概念和鸿蒙系统的架构图即可



01

层次架构

表现层和业务层框架设计

- 软件层次式体系结构是**最通用**的架构，也被叫作**N层架构模式**。大部分的应用会分成**表现层(或称为展示层)**、**中间层(或称为业务层)**、**数据访问层(或称为持久层)**和数据层。
- 表现层动态生成设计：基于XML的界面管理技术可实现灵活的**界面配置(静态)**、**界面动态生成和界面定制(动态)**。其思路是用XML生成配置文件及界面所需的元数据，按不同需求生成界面元素及软件界面。
- 业务框架位于系统架构的中间层，是实现系统功能的核心组件。采用容器的形式，便于系统功能的开发、代码重用和管理。下图便是在吸收了SOA思想之后的一个三层体系结构的简图。
- 在业务容器中，业务逻辑是按照Domain Model—Service—Control思想来实现的。
 - Domain Model: 是领域层业务对象，它仅仅包含**业务相关的属性**。
 - Service: 是业务过程实现的组成部分，是应用程序的不同功能单元，通过在这些服务之间定义良好的接口和契约联系起来。
 - Control: 服务控制器，是服务之间的纽带，不同服务之间的切换就是通过它来实现的。

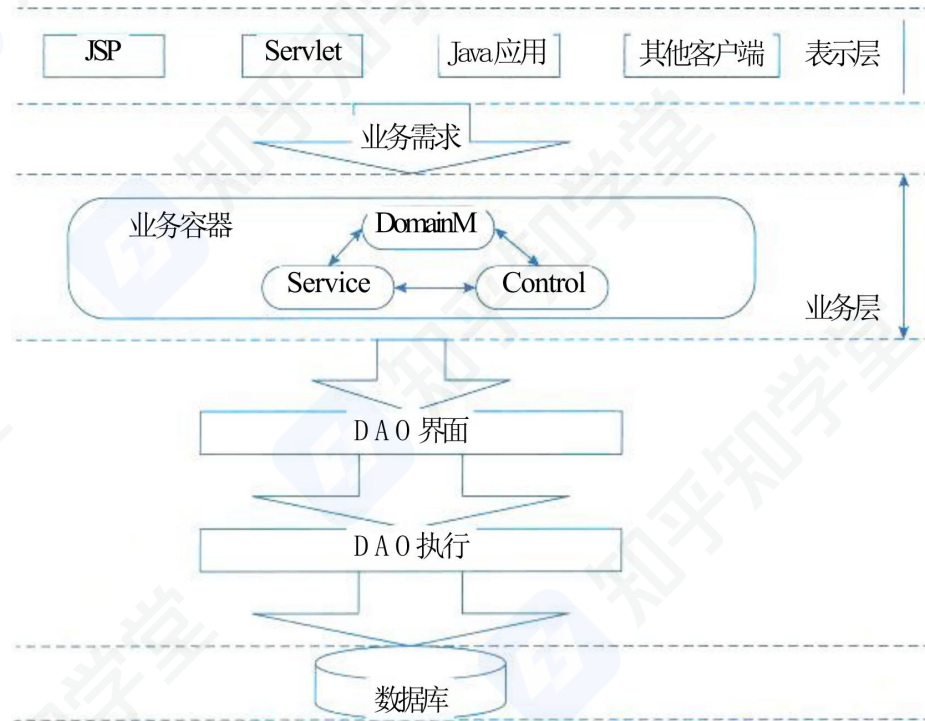


图13-10业务框架在整个系统架构中的位置

5种数据访问模式:

- **在线访问**: 会占用一个数据库连接, 读取数据, 每个数据库操作都会通过这个连接不断地与后台的数据源进行交互。比如使用pl/sql或者navicat连接数据库
- **Data Access Object**: 是标准J2EE设计模式之一, 开发人员常常用这种模式将底层数据访问操作与高层业务逻辑分离开。
- **Data Transfer Object**: 是经典EJB设计模式之一。DTO本身是这样一组对象或是数据的容器, 它需要跨不同的进程或是网络的边界来传输数据。这类对象本身应该不包含具体的业务逻辑, 并且通常这些对象内部只能进行一些诸如内部一致性检查和基本验证之类的方法, 而且这些方法最好不要再调用其他的对象行为。
- **离线数据模式**是**以数据为中心**, 数据从数据源获取之后, 将按照某种预定义的结构存放在系统中, 成为应用的中心。离线, 对数据的各种操作独立于各种与后台数据源之间的连接或是事务
- **对象/关系映射**(Object/Relation Mapping, O/R Mapping): 大多数应用中的数据都是依据关系模型存储在关系型数据库中; 而很多应用程序中的数据在开发或是运行时则是以对象的形式组织起来的。那么, 对象/关系映射就提供了这样一种工具或是平台, 能够帮助将**应用程序中的数据转换成关系型数据库中的记录;或是将关系数据库中的记录转换成应用程序中代码便于操作的对象**

- **工厂模式**在数据库访问层的应用：首先**定义一个操纵数据库的接口DataAccess**，然后根据**数据库的不同，由类工厂决定实例化哪个类**。因为DataAccess的具体实现类有一些共同的方法，所以先从DataAccess实现一个抽象AbstractDataAccess类，包含一些公用方法。然后，分别为SQL Server、Oracle和MySQL数据库编写三个数据访问的具体实现类。现在已经完成了所要的功能，下面需要创建一个Factory类，来实现自动数据库切换的管理。这个类很简单，主要的功能就是根据数据库类型，返回适当的数据库操纵类。
- **事务处理设计**：JavaBean中使用**JDBC**方式进行事务处理：在JDBC中，打开一个连接对象Connection时，默认是auto-commit模式，每个SQL语句都被当作一个事务，即每次执行一个语句，都会自动地得到事务确认。为了能将多个SQL语句组合成一个事务，要将auto-commit模式屏蔽掉。在auto-commit模式屏蔽掉之后，如果不调用commit()方法，SQL语句不会得到事务确认。在最近一次commit()方法调用之后的所有SQL会在方法commit()调用时得到确认。
- **连接对象管理设计（数据库连接池）**：通过**资源池**解决资源频繁分配、释放所造成的问题。有了这个连接池，下面就可以提供一套自定义的分配、释放策略。当客户请求数据库连接时，首先看连接池中是否有未分配出去的连接。如果存在空闲连接则把连接分配给客户，并标记该连接为已分配。若连接池中没空闲连接，就在已经分配出去的连接中，寻找一个合适的连接给客户，此时该连接在多个客户间复用。当客户释放数据库连接时，可以根据该连接是否被复用，进行不同的处理。如果连接没有使用者，就放入到连接池中，而不是被关闭。

物联网可以分为三个层次，**底层是用来感知数据的感知层**，即利用传感器、二维码、RFID等设备随时随地获取物体的信息。**第二层是数据传输处理的网络层**，即通过各种传感网络与互联网的融合，将对象当前的信息实时准确地传递出去。**第三层则是与行业需求结合的应用层**，即通过智能计算、云计算等将对象进行智能化控制。

- 感知层：用于识别物体、采集信息。感知层包括二维码标签和识读者、RFID标签和读写器、摄像头、GPS、传感器、M2M终端、传感器网关等，主要功能是识别对象、采集信息，与人体结构中皮肤和五官的作用类似。**感知层解决的是人类世界和物理世界的数据获取问题。**
- 网络层：用于传递信息和处理信息。网络层包括通信网与互联网的融合网络、网络管理中心、信息中心和智能处理中心等。网络层将感知层获取的信息进行传递和处理，类似于人体结构中的神经中枢和大脑。**网络层解决的是传输和预处理感知层所获得数据的问题。**
- 应用层：实现广泛智能化。应用层是物联网与行业专业技术的深度融合，结合行业需求实现行业智能化，这类似于人们的社会分工。物联网应用层利用经过分析处理的感知数据，为用户提供丰富的特定服务。**应用层解决的是信息处理和人机交互的问题。**

层次式架构案例分析

基于物联网架构的电子小票服务系统
采用感知层、网络层和应用层的3层物联网体系架构模型：





02

云原生架构

- 云原生架构是**基于云原生技术**的一组架构原则和设计模式的集合，旨在将**云应用中的非业务代码部分进行最大化的剥离**，从而让云设施接管应用中原有的大量非功能特性(如弹性、韧性、安全、可观测性、灰度等)，使业务不再有非功能性业务中断困扰的同时，具备轻量、敏捷、高度自动化的特点。
- 云原生的代码通常包括三部分：**业务代码、三方软件、处理非功能特性的代码**。从业务代码中剥离大量非功能性特性(不会是所有，比如易用性还不能剥离)到IaaS(基础设施即服务)和PaaS(平台即服务)中。
- 具备云原生架构的应用可以**最大程度利用云服务和提升软件交付能力，进一步加快软件开发**。其特点包括：**代码结构发生巨大变化、非功能性特性大量委托、高度自动化的软件交付**。
- 云原生架构原则
 - **服务化原则**：拆分为微服务架构、小服务架构，分别迭代。
 - **弹性原则**：系统的部署规模可以随着业务量的变化而自动伸缩。
 - **可观测原则**：通过日志、链路跟踪和度量等手段。
 - **韧性原则**：当软件所依赖的软硬件组件出现各种异常时，软件表现出来的抵御能力。
 - **所有过程自动化原则**：一方面标准化企业内部的软件交付过程，另一方面在标准化的基础上进行自动化，通过配置数据自描述和面向终态的交付过程。
 - **零信任原则**：默认情况下不应该信任网络内部和外部的任何人/设备/系统，需要基于认证和授权重构访问控制的信任基础，以身份为中心。
 - **架构持续演进原则**：云原生架构本身也必须是一个具备持续演进能力的架构。

主要架构模式

- **服务化架构模式**：典型模式是**微服务**和**小服务**模式。通过服务化架构，把**代码模块关系和部署关系进行分离**，每个接口可以部署不同数量的实例，单独扩缩容，从而使得整体的部署更经济。
- **Mesh化架构模式**：把**中间件框架(如RPC、缓存、异步消息等)**从业务进程中分离，让中间件**SDK**与业务代码进一步解耦，从而使得中间件升级对业务进程没有影响。分离后在业务进程中只保留很“薄”的**Client**部分。
- **Serverless模式**：将“部署”这个动作从运维中“收走”，使开发者不用关心应用运行地点、操作系统、网络配置、**CPU**性能等，也就是把应用的整个运行都委托给云。
- **存储计算分离模式**：在云环境中，推荐把各类暂态数据(如**session**)、结构化和非结构化持久数据都采用云服务来保存，从而实现存储计算分离。
- **分布式事务模式**：大颗粒度的业务需要访问多个微服务，必然带来分布式事务问题，否则数据就会出现不一致。架构师需要根据不同的场景选择合适的分布式事务模式。
- **可观测架构**：可观测架构包括**Logging**、**Tracing**、**Metrics** 三个方面，其中**Logging**提供多个级别的详细信息跟踪，由应用开发者主动提供;**Tracing**提供一个请求从前端到后端的完整调用链路跟踪对于分布式场景尤其有用;**Metrics**则提供对系统量化的多维度度量。
- **事件驱动架构**：本质上是一种应用/组件间的集成架构模式。可用于服务解耦、增强服务韧性、数据变化通知等场景中。

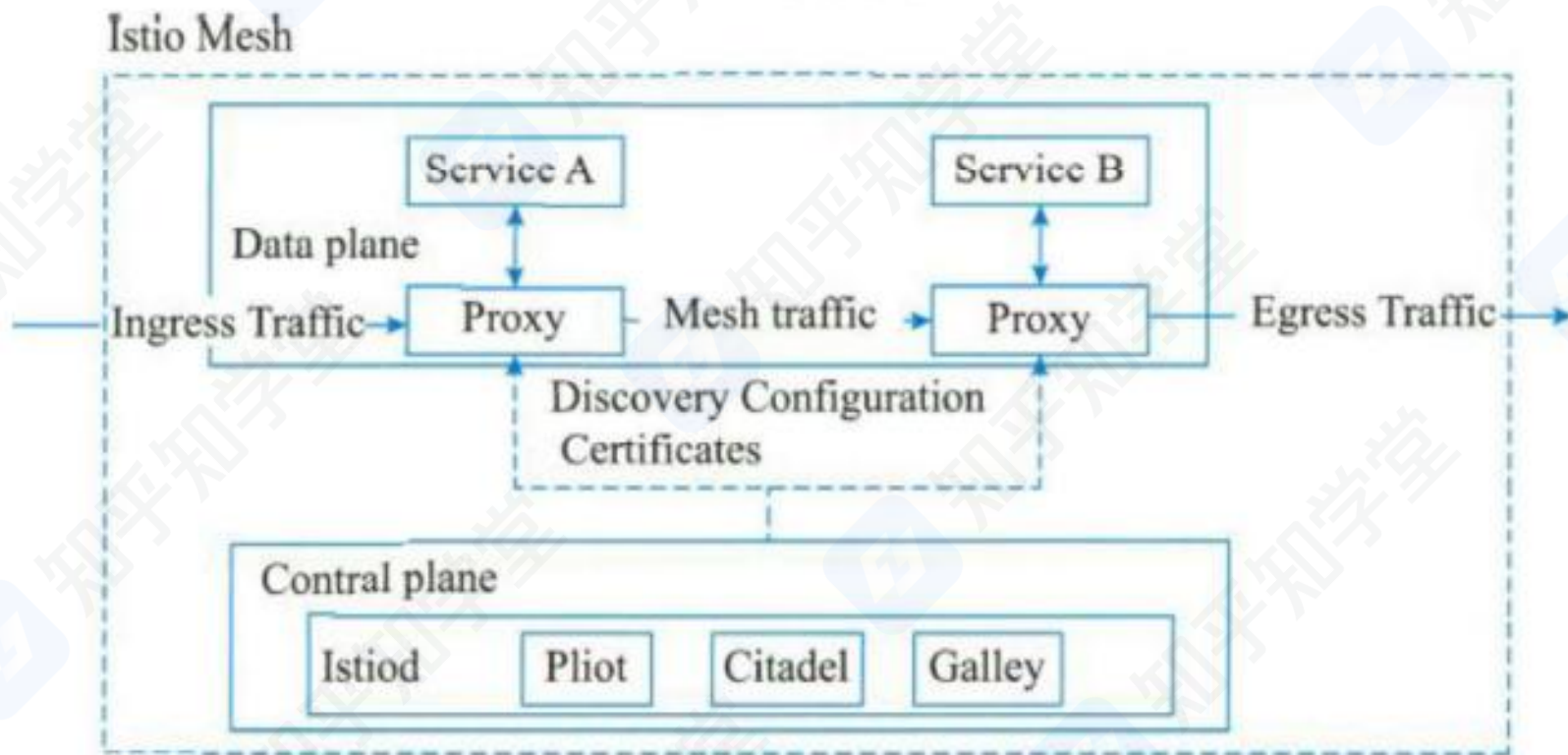
- 容器技术：容器作为标准化软件单元，它将应用及其所有依赖项打包，使应用不再受环境限制，在不同计算环境间快速、可靠地运行。通过容器技术，企业可以充分发挥云计算弹性优势，降低运维成本。
- Kubernetes已经成为容器编排的事实标准，被广泛用于自动部署，扩展和管理容器化应用。Kubernetes提供了分布式应用管理的核心能力，包括：资源调度、应用部署与管理、自动修复、服务发现与负载均衡、弹性伸缩、声明式API、可扩展性架构、可移植性。
- 云原生微服务：微服务模式将后端单体应用拆分为松耦合的多个子应用，每个子应用负责一组子功能。这些子应用称为“微服务”，多个“微服务”共同形成了一个物理独立但逻辑完整的分布式微服务体系。这些微服务相对独立，通过解耦研发、测试与部署流程，提高整体迭代效率。
- 微服务设计约束：
 - 微服务个体约束：功能在业务域划分上应是相互独立的，低耦合、单一职责。
 - 微服务与微服务之间的横向关系：主要从微服务的可发现性和可交互性处理服务间的横向关系，一般需要服务注册中心。
 - 微服务与数据层之间的纵向约束：在微服务领域，提供数据存储隔离原则，即数据是微服务的私有资产，对于该数据的访问都必须通过当前微服务提供的API来访问。
 - 全局视角下的微服务分布式约束：故障发现时效性和根因精确性始终是开发运维人员的核心诉求。

主要微服务技术:

- **Apache Dubbo**作为源自阿里巴巴的一款开源高性能**RPC**框架, 特性包括基于透明接口的**RPC**、智能负载均衡、自动服务注册和发现、可扩展性高、运行时流量路由与可视化的服务治理。
- **Spring Cloud**作为开发者的主要微服务选择之一, 为开发者提供了分布式系统需要的配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性**Token**、全局锁、决策竞选、分布式会话与集群状态管理等能力和开发工具。
- **Eclipse MicroProfile**作为**Java** 微服务开发的基础编程模型, 它致力于定义企业**Java**微服务规范, **MicroProfile**提供指标、**API**文档、运行状况检查、容错与分布式跟踪等能力, 使用它创建的云原生微服务可以自由地部署在任何地方, 包括服务网格架构。
- **Tars**是腾讯将其内部使用的微服务框架, 包含一整套开发框架与管理平台, 兼顾多语言、易用性.高性能与服务治理, 理念是让开发更聚焦业务逻辑, 让运维更高效。
- **SOFAStack**是由蚂蚁金服开源的一套用于快速构建金融级分布式架构的中间件, 也是在金融场景里的最佳实践。
- **DAPR**(分布式应用运行时)是微软新推出的一种可移植的、无服务器的、事件驱动的运行时, 它使开发人员可以轻松构建弹性, 无状态和有状态微服务, 这些服务运行在云和边缘上, 并包含多种语言和开发框架。

- **无服务器技术(Serverless)**因为屏蔽了服务器的各种运维复杂度，让开发人员可以将**更多精力用于业务逻辑设计与实现**，而逐渐成为云原生主流技术之一。
- **Serverless**计算包含以下特征：
 - **全托管的计算服务**，客户只需要编写代码构建应用，无需关注同质化的、负担繁重的基于服务器等基础设施的开发、运维、安全、高可用等工作；
 - **通用性**，结合云BaaSAPI的能力，能够支撑云上所有重要类型的应用；
 - 自动弹性伸缩，让用户无需为资源使用提前进行容量规划；
 - **按量计费**，让企业使用成本得有效降低，无需为闲置资源付费。
- **函数计算(FaaS)**是**Serverless**中最具代表性的产品形态。通过把应用逻辑拆分多个函数，每个函数都通过事件驱动的方式触发执行。
- 无服务器技术关注点：计算资源弹性调度、负载均衡和流控、安全性。

- 服务网格(ServiceMesh)是分布式应用在微服务软件架构之上发展起来的新技术，旨在将那些微服务间的连接、安全、流量控制和可观测等通用功能下沉为平台基础设施，实现应用与平台基础设施的解耦。这个解耦意味着开发者无需关注微服务相关治理问题而聚焦于业务逻辑本身，提升应用开发效率并加速业务探索和创新。
- 在这张架构图中，服务A调用服务B的所有请求，都被其下的服务代理截获，代理服务A完成到服务B的服务发现、熔断、限流等策略，而这些策略的总控是在控制平面(ControlPlane)上配置。



某旅行公司云原生改造：某公司主体由两个公司合并之后，出现了以下2个问题：

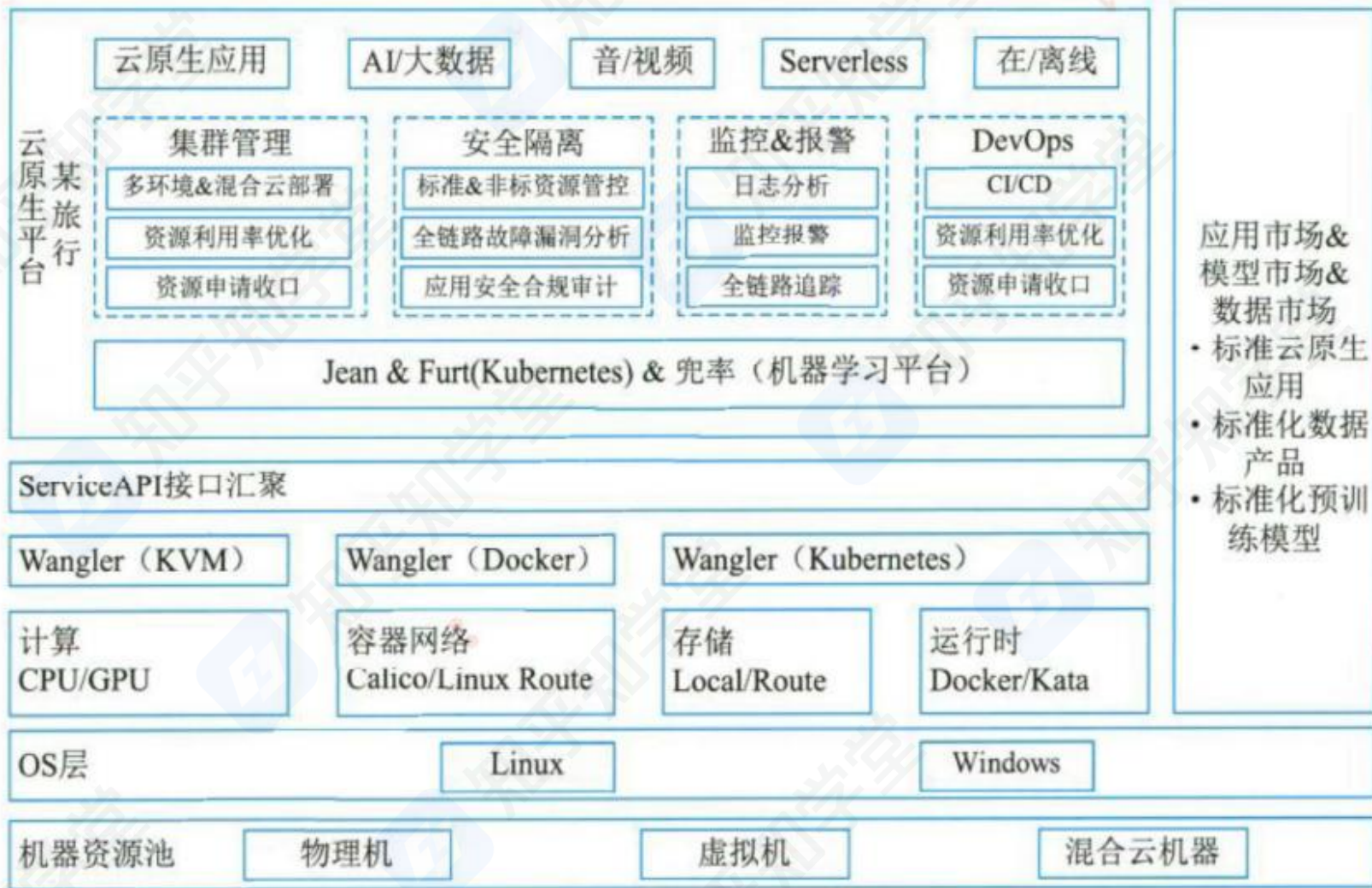
- 技术体系不同，需要整合为一体
- 节假日高并发流量

- 改造第一阶段，某旅行技术团队为了提升集群资源利用率，降低资源使用成本。利用云原生思维重构部分技术体系，将多套旧有系统合并、收拢到一套以云原生应用为核心的私有云平台上，同时将IDC、物理网络、虚拟网络、计算资源、存储资源等通过IaaS、PaaS等，实现虚拟化封装、切割再投产的自动化流程。随着服务器集群规模的扩大，部分机器开始频繁出现故障。此时，保障服务稳定性成了第二阶段改造的首要任务。
- 第二阶段基于公有云、私有云和离线专属云集群等新型动态计算环境，某旅行公司的技术团队帮助业务构建和运行具有弹性的云原生应用，促进业务团队开始使用声明式API，同时通过不可变基础设施、服务网格和容器服务，来构建容错性好、易于管理和观察的应用系统，并结合平台可靠的自动化恢复、弹性计算来完成整个服务稳定性的提升。
- 第三阶段通过基础组件、服务的云原生改造、服务依赖梳理和定义等方式，使应用不再需要考虑底层资源、机房、运行时间和供应商等因素。此外，还利用标准的云原生应用模型，实现了服务的跨地域、跨云自动化灾备、自动部署，并向云原生场景下的DevOps演进。架构图如下：

云原生架构案例分析

从最下层到最上层，各层作用如下：

- 1.机器资源池：提供物理机、虚拟机、混合云机器等基础硬件资源，是架构的“硬件基石”。
- 2.OS 层：提供 Linux、Windows 等操作系统环境，衔接硬件与上层软件。
- 3.基础资源层：封装计算（CPU/GPU）、容器网络、存储、运行时等核心资源，为上层提供“计算、网络、存储、容器运行”能力。
- 4.Wangler 资源管理层：统一管理 KVM（虚拟化）、Docker（容器）、Kubernetes（编排）等技术，简化虚拟机、容器的生命周期调度。
- 5.ServiceAPI 接口汇聚层：将下层资源与能力封装为标准化 API，向上提供统一调用入口，屏蔽底层复杂度。
- 6.基础平台层：依托 Kubernetes 和机器学习平台，提供云原生编排、AI / 大数据支撑能力，是业务的“技术底座”。
- 7.平台能力层：通过集群管理、安全隔离、监控报警、DevOps 四大能力，保障上层业务的稳定、高效运行。
- 8.应用场景层：直接支撑云原生应用、AI / 大数据、音视频、Serverless、在 / 离线等各类业务场景。



云原生架构案例分析

1. 混合云环境

提供测试、压测、生产、公有云等多类部署环境，作为系统的基础资源载体，支持不同阶段（开发→测试→生产）与不同云形态（私有 + 公有）的应用部署。

2. 容器云平台

围绕资源、集群、应用、镜像提供全生命周期管理，并集成安全、日志、监控告警能力，是云原生架构的核心底座，为上层应用提供容器化部署、调度与运维支撑。

3. DevOps

覆盖需求到运维全流程，通过自动化与跨团队协作，加速应用从“需求”到“上线运维”的生命周期，保障持续交付效率。

4. 中间件服务

提供RabbitMQ（消息队列）、Kafka（分布式消息）、Redis（缓存）、Nginx（反向代理 / 负载均衡）等通用中间件，为上层服务提供消息通信、数据缓存、流量路由等基础能力。

5. 微服务治理组件

包含服务发现、消息中心、调度中心、配置中心，解决微服务架构下的“服务注册发现、消息路由、任务调度、配置统一管理”等治理问题，保障微服务间的高效协同。

6. 业务组件

承载用户、车辆、订单、能量等核心业务逻辑，是直接面向业务场景的功能模块，为用户提供具体的业务价值（如用户管理、订单交易等）。

7. 网关

作为系统的统一入口，负责请求路由、流量控制、认证授权等，屏蔽内部架构复杂度，对外提供简洁、安全的访问接口。





03

面向服务架构

■ 粒度不同:

- 整体上来说, SOA 的服务粒度要粗一些, 而微服务的服务粒度要细一些。例如, 对一个大型企业来说, “员工管理系统”就是一个 SOA 架构中的服务; 而如果采用微服务架构, 则“员工管理系统”会被拆分为更多的服务, 比如“员工信息管理”、“员工考勤管理”、“员工假期管理”和“员工福利管理”等更多服务。

■ 通信模式:

- SOA 采用了 ESB 作为服务间通信的关键组件, 负责服务定义、服务路由、消息转换、消息传递, 总体上是重量级的实现。
- 微服务推荐使用统一的协议和格式, 例如, RESTful 协议、RPC 协议, 无须 ESB 这样的重量级实现。

■ 服务交付:

- SOA 对服务的交付并没有特殊要求, 因为 SOA 更多考虑的是兼容已有的系统;
- 微服务的架构理念要求“快速交付”, 相应地要求采取自动化测试、持续集成、自动化部署等敏捷开发相关的最佳实践。如果没有这些基础能力支撑, 微服务规模一旦变大 (例如, 超过 20 个微服务), 整体就难以达到快速交付的要求, 这也是很多企业在实行微服务时踩过的一个明显的坑, 就是系统拆分为微服务后, 部署的成本呈指数上升。

■ 应用场景:

- SOA 更加适合于庞大、复杂、异构的企业级系统, 这也是 SOA 诞生的背景。这类系统的典型特征就是很多系统已经发展多年, 采用不同的企业级技术, 有的是内部开发的, 有的是外部购买的, 无法完全推倒重来或者进行大规模的优化和重构。因为成本和影响太大, 只能采用兼容的方式进行处理, 而承担兼容任务的就是 ESB。
- 微服务更加适合于快速、轻量级、基于 Web 的互联网系统, 这类系统业务变化快, 需要快速尝试、快速交付; 同时基本都是基于 Web, 虽然开发技术可能差异很大 (例如, Java、C++、.NET 等), 但对外接口基本都是提供 HTTP RESTful 风格的接口, 无须考虑在接口层进行类似 SOA 的 ESB 那样的处理。

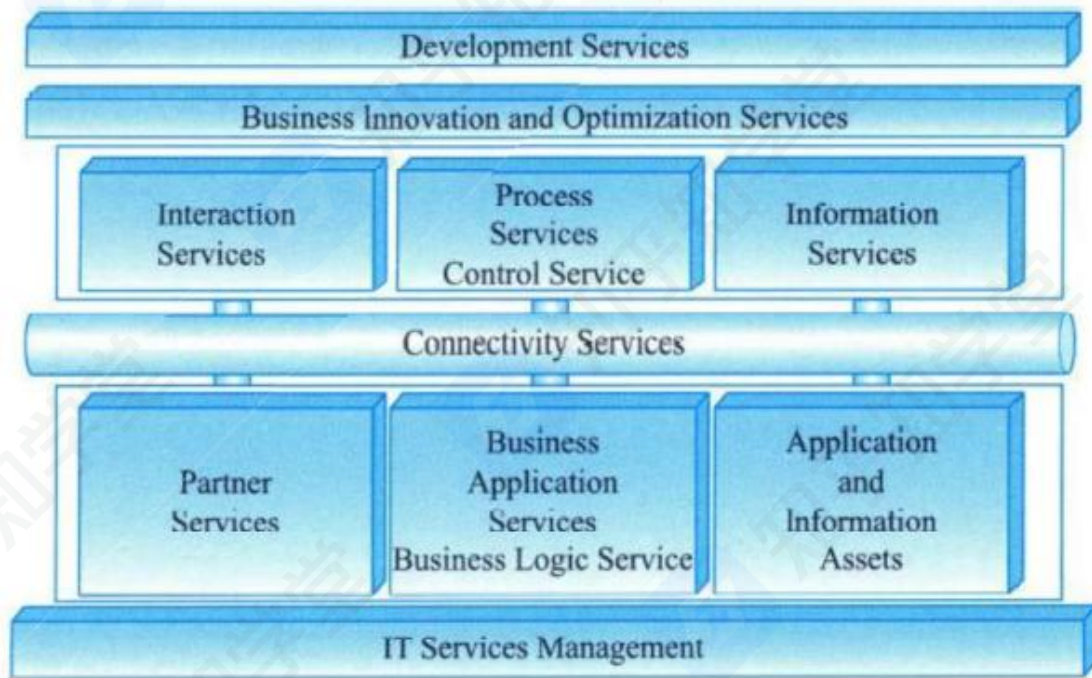
■ 因此, 我们可以看到, SOA 和微服务本质上是两种不同的架构设计理念, 只是在“服务”这个点上有交集而已。

■ SOA 和微服务是两种不同理念的架构模式, 并不存在孰优孰劣, 只是应用场景不同而已。我们介绍 SOA 时候提到其产生历史背景是因为企业的 IT 服务系统庞大而又复杂, 改造成本很高, 但业务上又要求其互通, 因此才会提出 SOA 这种解决方案。如果我们将微服务的架构模式生搬硬套到企业级 IT 服务系统中, 这些 IT 服务系统的改造成本可能远远超出实施 SOA 的成本。

对比维度	SOA	微服务
服务粒度	粗	细
服务通信	重量级，ESB	轻量级，例如 HTTP RESTful
服务交付	慢	快
应用场景	企业级	互联网

SOA的参考架构

- 典型的以服务为中心的企业集成架构如下图所示，采用“**关注点分离**”的方法规划企业集成中的各种架构元素，同时从服务视角规划每种架构元素提供的服务，以及服务如何被组合在一起完成某种类型的集成。可划分为六大类：
- **业务逻辑服务 (Business Logic Service)**: 包括用于实现业务逻辑的服务和执行业务逻辑的能力，其中包括业务应用服务 (Business Application Service)、业务伙伴服务 (PartnerService) 以及应用和信息资产 (Application and Information asset)。
- **控制服务 (Control Service)**: 包括实现人 (People)、流程 (Process) 和信息 (Information) 集成的服务，以及执行这些集成逻辑的能力。
- **连接服务 (Connectivity Service)**: 通过提供企业服务总线提供分布在各种架构元素中服务间的连接性。
- **业务创新和优化服务 (Business Innovation and Optimization Service)**: 用于监控业务系统运行时服务的业务性能，并通过及时了解到的业务性能和变化，采取措施适应变化的市场。
- **开发服务 (Development Service)**: 贯彻整个软件开发生命周期的开发平台，从需求分析，到建模、设计、开发、测试和维护等全面的工具支持。
- **IT服务管理 (IT Service Management)**: 支持业务系统运行的各种基础设施管理能力或服务，如安全服务、目录服务、系统管理和资源虚拟化。



- 连接服务：通过企业服务总线(ESB)来实现，它的基本特征和能力包括：
 - 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。
- 业务逻辑服务
 - 整合已有应用(应用和信息访问服务)：实现对已有应用和信息的集成，主要有两类访问服务：可接入服务、事件发现服务。
 - 整合新开发的应用(业务应用服务)：实现新应用集成，主要有三类业务应用服务：组件服务(可重用)、核心服务（运行时）、接口服务。
 - 整合客户和业务伙伴(B2C/B2B，伙伴服务)：提供与企业外部的B2B的集成能力，包括：社区服务、文档服务、协议服务。

■ 控制服务:

- **数据整合**(信息服务): 提供集成数据的能力, 目前主要包括如下集中信息服务: 联邦服务(不同类型数据聚合)、复制服务(远程数据本地访问)、转换服务(格式转换)、搜索服务。
- **流程整合**(流程服务): 完成业务流程集成, 包括: 编排服务(预定义流程顺序)、事务服务(保证ACID)、人工服务(人工活动集成到流程中)。
- **用户访问整合**(交互服务): 实现用户访问集成, 包括: 交付服务(运行时交互框架)、体验服务、资源服务(运行时交互组件的管理)。

■ 开发服务: 开发环境和工具中为不同开发者的角色提供的功能被称为开发服务。根据开发过程中开发者角色和职责的不同, 有如下4类服务: 建模服务、设计服务、实现服务、测试服务。

■ 业务创新和优化: 以业务性能管理(BPM)技术为核心提供业务事件发布、收集和关键业务指标监控能力。包括以下服务:

- 公共事件框架服务: 通过一个公共事件框架提供IT和业务事件的激发、存储和分类等。
- 采集服务: 通过基于策略的过滤和相关性分析检测感兴趣的服务。
- 监控服务: 通过事件与监控上下文间的映射, 计算和管理业务流程的关键性能指标。

■ IT服务管理: 为业务流程和服务提供安全、高效和健康的运行环境, 包括: 安全和目录服务、系统管理和虚拟化服务。

■ Web服务最基本的协议包括**UDDI**、**WSDL**和**SOAP**，通过它们，可以提供直接而又简单的Web Service支持,如图所示。

- **UDDI**: UDDI 是用于服务的注册和发现。它允许提供者将其 Web 服务描述注册到 UDDI 注册中心，以便其他用户可以发现并访问这些服务。
- 示例：假设有一个电子商务网站，它提供了一组 Web 服务，包括产品搜索、购物车管理和订单处理。这些服务的描述信息可以被注册到 UDDI 注册中心，以便其他电子商务网站或应用程序可以查找并使用这些服务。
- **WSDL**: WSDL 是用于描述服务的接口和操作。它提供了一种标准方式来定义服务的输入、输出以及如何与服务进行交互。
- 示例：假设一个天气预报 Web 服务，它提供了获取特定城市天气信息的功能。WSDL 文件将定义该服务的接口，包括输入参数（城市名称）和输出（天气信息）。其他开发人员可以使用这个 WSDL 文件来了解如何调用该服务。
- **SOAP**: SOAP 是一种用于在不同计算机之间进行通信的协议(在不同服务之间进行消息交换)。它允许 Web 服务之间的消息交换，并提供了一种标准的、跨平台的通信机制。一般来说是一个XML文件
- 示例：考虑一个在线支付系统，它需要与信用卡验证 Web 服务进行通信以验证用户的支付信息。在此过程中，SOAP 协议可以用于创建包含用户支付信息的信息，并将其发送到信用卡验证服务。该服务将使用相同的 SOAP 协议返回验证结果。



- **REST (Representational State Transfer)** 是一种用于构建分布式系统的架构风格和设计原则，它强调使用现有的 Web 标准和协议，并通过资源的状态表示来实现通信。REST 常用于构建 Web 服务和 API。以下是 REST 规范的核心原则：
 - **资源**：在 REST 中，一切都是资源，如网页、图片、文件等。每个资源都有一个唯一的地址 (URL) 来标识它。
 - **表现层**：资源的状态以某种格式表示，通常是 JSON 或 XML。例如，一篇博客文章可以以 JSON 形式表示其标题和内容。
 - **HTTP 方法**：HTTP 方法 (如 GET、POST、PUT、DELETE) 用于对资源进行操作。GET 用于获取资源，POST 用于创建资源，PUT 用于更新资源，DELETE 用于删除资源。
 - **无状态**：REST 是无状态的，每个请求都包含足够的信息，服务器不需要保存客户端的状态。
 - **客户端-服务器**：客户端负责请求资源，服务器负责提供资源。这种分离性使系统更容易扩展和维护。
 - **统一接口**：REST 使用统一的接口规则，使客户端和服务端能够彼此理解。
- **举例**：假设你使用 REST 架构构建一个博客系统：
 - 每篇博客文章是一个资源，每篇文章都有一个唯一的 URL。
 - 博客文章的信息以 JSON 形式表示，包括标题、内容、作者等。
 - 你可以使用 HTTP GET 请求来获取文章，HTTP POST 请求来创建新文章，HTTP PUT 请求来更新文章，HTTP DELETE 请求来删除文章。
 - 服务器不需要保存客户端的状态，每个请求都是独立的。
 - 客户端通过 HTTP 请求获取博客文章，服务器通过 HTTP 响应提供博客文章。
- 只要遵循 REST 设计思想同时满足设计约束的一类架构设计或者应用程序的统称，这一类都可以称之为 RESTFull。

■ SOA的设计原则

- **无状态**。调用服务的时候不用考虑到它还需要其他的状态及数据，以避免服务请求者依赖于服务提供者的状态。
- **单一实例**。每个服务都只提供单一的功能，避免功能冗余。
- **明确定义的接口**。使用者依赖服务规约调用服务，所以服务定义必须长时间稳定，一旦公布，不能随意更改;服务的定义应尽可能明确，减少使用者的不适当使用;不要让使用者看到服务内部的私有数据。
- **自包含和模块化**。服务封装了那些在业务上稳定、重复出现的活动和组件，实现服务的功能实体是完全独立自主的，独立进行部署、版本控制、自我管理和恢复。
- **粗粒度**。服务数量不应该太多，依靠消息交互而不是远程过程调用(RPC)，通常消息量比较大，但是服务之间的交互频度较低。
- **服务之间的松耦合性**。服务使用者看到的是服务的接口，其位置、实现技术和当前状态等对使用者是不可见的，服务私有数据对服务使用者是不可见的。
- **重用能力**。服务应该是可以重用的。
- **互操作性、兼容和策略声明**。为了确保服务规约的全面和明确，策略成为一个越来越重要的方面。这可以是技术相关的内容，例如一个服务对安全性方面的要求;也可以是跟业务有关的语义方面的内容，例如需要满足的费用或者服务级别方面的要求，这些策略对于服务在交互时是非常重要的。

■ 服务注册表模式，支持如下SOA治理功能：

- 服务注册：应用开发者，也叫服务提供者，向注册表公布他们的功能。
- 服务位置：也就是服务应用开发者，帮助他们查询注册服务，寻找符合自身要求的服务。
- 服务绑定：服务的消费者利用检索到的服务合同来开发代码，开发的代码将与注册的服务绑定、调用注册的服务以及与他们实现互动。

■ 企业服务总线模式，由中间件技术实现的支持面向服务架构的基础软件平台，支持异构环境中的服务以基于消息和事件驱动模式的交互，并且具有适当的服务质量和可管理性。

- 一个典型的在ESB环境中组件之间的交互过程是：首先由服务请求者触发一次交互过程，产生一个服务请求消息，并将该消息按照ESB的要求标准化，然后标准化的消息被发送给服务总线。ESB根据请求消息中的服务名或者接口名进行目的组件查找，将消息转发至目的组件，并最终将处理结果逆向返回给服务请求者。这种交互过程不再是点对点的直接交互模式，而是由事件驱动的消息交互模式。

■ ESB的核心功能如下。

- 提供位置透明性的消息路由和寻址服务。
- 提供服务注册和命名的管理功能。
- 支持多种消息传递范型(如请求/响应、发布/订阅等)。
- 支持多种可以广泛使用的传输协议。
- 支持多种数据格式及其相互转换。
- 提供日志和监控功能。



04

安全架构

- 对于信息系统来说，威胁可以是对物理环境、通信链路、网络系统、操作系统、应用系统以及管理系统等方面。
 - 物理安全威胁：是指对系统所用设备的威胁，如自然灾害、电源故障等；
 - 通信链路安全威胁：是指在传输线路上安装窃听装置或对通信链路进行干扰；
 - 网络安全威胁：是指通过技术手段窃取互联网信息，对网络形成严重的安全威胁；
 - 操作系统安全威胁：是指对系统平台中的软件或硬件芯片中植入威胁，如“木马”和“陷阱门”、BIOS的万能密码；
 - 应用系统安全威胁：是指对于网络服务或用户业务系统安全的威胁；
 - 管理系统安全威胁：是指由于人员管理上疏忽而引发人为的安全漏洞，如人为的通过拷贝、拍照、抄录等手段盗取计算机信息。
- 具体来讲，常见的安全威胁有以下几种。
 - (1)信息泄露：信息被泄露或透露给某个非授权的实体。
 - (2)破坏信息的完整性：数据被非授权地进行增删、修改或破坏而受到损失。
 - (3)拒绝服务：对信息或其他资源的合法访问被无条件地阻止。
 - (4)非法使用(非授权访问)：某一资源被某个非授权的人或以非授权的方式使用。
 - (5)窃听：用各种可能的合法或非法的手段窃取系统中的信息资源和敏感信息。

(6)业务流分析：通过对系统进行**长期监听**，利用统计分析方法对诸如通信频度、通信的信息流向、通信总量的变化等态势进行研究，从而发现有价值的信息和规律。

(7)假冒：通过欺骗通信系统(或用户)达到非法用户冒充成为合法用户，或者特权小的用户冒充成为特权大的用户的目的。**黑客大多是采用假冒进行攻击。**

(8)旁路控制：攻击者利用系统的安全缺陷或安全性上的脆弱之处获得非授权的权利或特权。

(9)授权侵犯：被授权以某一目的使用某一系统或资源的某个人，却将此权限用于其他非授权的目的，也称作“内部攻击”。

(10)特洛伊木马：软件中含有一个察觉不出的或者无害的程序段，当它被执行时，会破坏用户的安全。

(11)陷阱门(后门)：在某个系统或某个部件中设置了“机关”，使得当提供特定的输入数据时，允许违反安全策略。

(12)抵赖：这是一种来自用户的攻击。例如：否认自己曾经发布过的某条消息、伪造一份对方来信等。

(13)重放：所截获的某次合法的通信数据备份，出于非法的目的而被重新发送。

(14)计算机病毒：所谓计算机病毒，是一种在计算机系统运行过程中能够实现传染和侵害的功能程序。一种病毒通常含有两个功能：一种功能是对其他程序产生“感染”；另外一种或者是引发损坏功能或者是一种植入攻击的能力。

(15)人员渎职：一个授权的人为了钱或利益、或由于粗心，将信息泄露给一个非授权的人。

(16)媒体废弃：信息被从废弃的磁盘或打印过的存储介质中获得。

(17)物理侵入：侵入者通过绕过物理控制而获得对系统的访问。

(18)窃取：重要的安全物品，如令牌或身份卡被盗。

(19)业务欺骗：某一伪系统或系统部件欺骗合法的用户或系统自愿地放弃敏感信息。

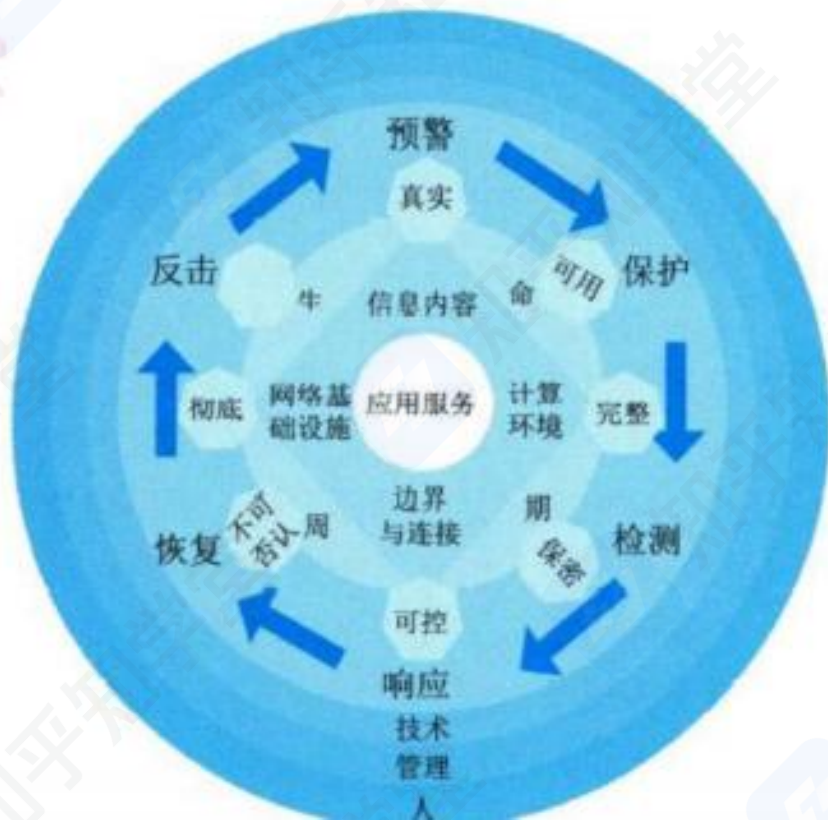
■ 安全架构是架构面向安全性方向上的一种细分，通常的产品安全架构、安全技术体系架构和审计架构可组成三道安全防线。

- 产品安全架构：构建产品安全质量属性的主要组成部分以及它们之间的关系。产品安全架构的目标是如何在不依赖外部防御系统的情况下，从源头打造自身安全的产品。
- 安全技术体系架构：构建安全技术体系的主要组成部分以及它们之间的关系。安全技术体系架构的任务是构建通用的安全技术基础设施，包括安全基础设施、安全工具和技术、安全组件与支持系统等，系统性地增强各产品的安全防御能力。
- 审计架构：独立的审计部门或其所能提供的风险发现能力，审计的范围主要包括安全风险在内的所有风险。

- 信息系统的安全目标是**控制和管理主体(含用户和进程)对客体(含数据和程序)的访问**。
- 安全模型是准确地描述安全的重要方面及其与系统行为的关系，安全策略是从安全角度为系统整体和构成它的组件提出基本的目标。**安全模型提供了实现目标应该做什么，不应该做什么，具有实践指导意义，它给出了策略的形式**。
- 信息系统安全体系主要是由**技术体系(技术)、组织机构体系(人)和管理体系(制度)**三部分共同构成的。
 - 技术体系是**全面提供信息系统安全保护的技术保障系统**，该体系由物理安全技术和系统安全技术两大类构成。
 - 组织体系是信息系统的**组织保障系统**，由机构、岗位和人事三个模块构成。
 - 管理体系由**法律管理、制度管理和培训管理**三部分组成。

信息安全整体架构设计

- WPDRRC(Waring/Protect/Detect/React/Restore/Counterattack)是我国针对美国提出的PDRR的基础上新增了预警和反击功能的信息安全模型，它有6个环节和3大要素。
- 6个环节包括：预警、保护、检测、响应、恢复和反击，它们具有较强的时序性和动态性，能够较好地反映出信息系统安全保障体系的预警能力、保护能力、检测能力、响应能力、恢复能力和反击能力。
- 3大要素包括：人员、策略和技术。人员是核心，策略是桥梁，技术是保证，落实在WPDRRC 的6个环节的各个方面，将安全策预警略变为安全现实。



■ 信息系统安全设计重点考虑两个方面：**系统安全保障体系**和**信息安全体系架构**。

1.系统安全保障体系：是由**安全服务、协议层次和系统单元**等三个层面组成，且每个层都涵盖了安全管理的内容。系统安全保障体系设计工作主要考虑以下几点：

- **安全区域策略的确定**：根据安全区域的划分，主管部门应制定针对性的安全策略，比如涉及到金钱和隐私信息的区域的安全级别更高，涉及到外部人员的区域安全级别低一些。
- **统一配置和管理防病毒系统**：主管部门应当建立整体防御策略，以实现统一的配置和管理。
- **网络安全管理**：加强网络安全管理，制定有关规章制度。

2.信息安全体系架构：具体在**安全控制系统**，我们可以从下面5个方面开展分析和设计工作。

- **物理安全**：保证计算机信息系统各种设备的物理安全是保障整个网络系统安全的前提。包括：环境安全、设备安全、媒体安全等。
- **系统安全**：主要是指对信息系统组成中各个部件的安全要求。系统安全是系统整体安全的基础。它主要包括：网络结构安全、操作系统安全和应用系统安全。
- **网络安全**：是整个安全解决方案的关键。它主要包括：访问控制、通信保密、入侵检测、网络安全扫描系统和防病毒等。
- **应用安全**：主要是指多个用户使用网络系统时，对共享资源和信息存储操作所带来的安全问题。它主要包括资源共享和信息存储两个方面。
- **安全管理**：主要体现在三个方面。其一是制定健全的安全管理体制;其二是构建安全管理平台;其三是增强人员的安全防范意识。

信息安全整体架构设计-面向企业的安全控制系统安全架构

1. 数据源

提供系统最基础的数据来源，涵盖网络、主机、存储、数据库、应用系统、机房环境、资产、文档等，是所有数据的“源头”。

2. 数据处理层

负责采集数据（自动采集或人工录入），涵盖监控数据、管理数据、系统数据、文档数据、文档数据，为上层功能提供数据支撑。

3. 功能层

系统的核心能力层，分为三大模块：

- 系统可用性监控：通过告警管理、拓扑/网络/应用等多维度监控、故障关联分析，保障系统能稳定运行。
- 服务支持：以“服务台”为核心，开展事件、请求、问题、变更、配置、知识库管理，为运维、管理等工作提供流程化服务支撑。
- 系统安全性监控：通过安全事件处理流程、实时监控、威胁追溯、安全域管理、风险管理、安全事件关联分析，保障系统安全。
- （另有“辅助工具”增强上述功能）

4. 展现层

将功能层的能力可视化/交互化呈现，包括总控室（全局视角监控）、运维门户网站（运维人员操作入口）、客户管理、报表管理、决策支持（为管理提供数据决策）、资产管理等，方便不同角色查看和使用。

5. 用户接入层

支持运维人员、管理人员、客户通过 WEB、PDA、即时通信、电话等方式接入系统，是用户与系统交互的“入口”。



数据库系统的安全设计

- **数据库完整性**是指数据库中数据的正确性和相容性。数据库完整性由各种各样的**完整性约束**来保证，因此可以说数据库完整性设计就是数据库完整性约束的设计。数据库完整性约束可以通过**DBMS**或应用程序来实现，基于**DBMS**的完整性约束作为模式的一部分存入数据库中。
- 在实施数据库完整性设计时，需要把握以下基本原则：
 - 根据数据库完整性约束的类型确定其实现的系统层次和方式，并提前考虑对系统性能的影响。一般情况下，静态约束应尽量包含在数据库模式中，而动态约束由应用程序实现。比如在购物车中，库存的数量是一个静态约束，应该包含在数据库模式中，以确保不会出现负库存。另一方面，用户的购物车中的商品数量则是一个动态约束，因为它根据用户的操作动态变化，应该由应用程序来实现。
 - 实体完整性约束、引用完整性约束是关系数据库最重要的完整性约束，在不影响系统关键性能的前提下需尽量应用。用一定的时间和空间来换取系统的易用性是值得的。比如在银行的数据库中，账户余额是一个非常重要的数据。应该应用实体完整性约束来确保账户余额不会出现负数，从而保护客户的资金
 - 要**慎用目前主流DBMS都支持的触发器功能**，一方面由于触发器的性能开销较大;另一方面，触发器的多级触发难以控制，容易发生错误，非用不可时，最好使用**Before**型语句级触发器。
 - 在需求分析阶段就必须制定完整性约束的命名规范，尽量使用有意义的英文单词、缩写词、表名、列名及下画线等组合，使其易于识别和记忆。
 - 要根据业务规则对数据库完整性进行细致的测试，以尽早排除隐含的完整性约束间的冲突和对性能的影响。
 - 要有专职的数据库设计小组，自始至终负责数据库的分析、设计、测试、实施及早期维护。
 - 应采用合适的**CASE**工具来降低数据库设计各阶段的工作量。



系统架构的脆弱性分析



- 从技术角度而言，漏洞的来源主要有以下几个方面：软件设计时的瑕疵、软件实现中的弱点、软件本身的瑕疵、系统和网络的错误配置。
- 软件脆弱性有其自身的特点，主要包括4个方面：
 - 脆弱性是软件系统中隐藏的一个弱点，本身不会引起危害，但被利用后会产生严重的安全后果；
 - 在软件开发过程中，自觉或不自觉引入的逻辑错误是大多数脆弱性的根本来源；
 - 与具体的系统环境密切相关，系统环境的任何差异都有可能产生不同的脆弱性问题；
 - 旧的脆弱性得到修补或纠正的同时可能引入新的脆弱性，因此脆弱性问题会长期存在。
- 软件脆弱性的生命周期：即每一种脆弱性都有其引入原因；在一种脆弱性引入之后，它会产生某种破坏效果，从而破坏系统的完整性或者可用性；针对已有的每一种脆弱性，人们可能会提出一些修补措施，在实施这些修补措施之后脆弱性将消失。



■ 软件脆弱性分析可从三个方面考虑:

- 分析**软件故障现象**，分析故障的技术本质、总结脆弱性模式;
- 分析**软件开发**，发现安全管理和技术的薄弱环节，提高软件安全性;
- 分析**软件使用**，发现其脆弱性，采取相应措施，避免脆弱性转化为安全故障。

■ 软件脆弱性分析首先要明确分析对象，脆弱性分析对象可以分为两类：**脆弱性数据**和**软件系统**。由于软件本身具有自身的性质和特点，针对软件的脆弱性分析，我们也需要考虑软件本身的各种特点。主要考虑**软件结构**和**实现技术**两个方面。

典型**软件架构**的脆弱性分析

1.**分层架构**的脆弱性主要表现在两个方面:

- **层间**的脆弱性。一旦某个底层发生错误，那么整个程序将会无法正常运行。
- **层间通信**的脆弱性。将系统隔离为多个相对独立的层，这就要求在层与层之间引入通信机制。本来“直来直去”的操作现在要层层传递，势必造成**性能下降**。

2.**C/S架构**的脆弱性主要表现在以下几个方面:

- **客户端软件**的脆弱性。因为在用户计算机上安装了客户端软件，所以这个系统就面临着程序被分析、数据被截取的安全隐患。
- **网络开放性**的脆弱性。目前很多传统的**C/S**系统还是采用二层结构，也就是说所有客户端直接读取服务器端中的数据，在客户端包括了数据的用户名，密码等致命的信息，这样会给系统带来安全隐患。
- **网络协议**的脆弱性。

3.B/S 架构的脆弱性主要表现在：系统如果使用HTTP协议，B/S架构相对C/S架构而言**更容易被病毒入侵**，虽然最新的HTTP协议在安全性方面有所提升，但还是弱于C/S。

4.**事件驱动架构**的脆弱性主要表现在：

- **组件**的脆弱性。组件削弱了自身对系统的控制能力，一个组件触发事件，并不能确定响应该事件的其他组件及各组建的执行顺序。
- **组件间交换数据**的脆弱性。组件不能很好地解决数据交换问题，事件触发时，一个组件有可能需要将参数传递给另一个组件，而数据量很大的时候，如何有效传递是一个脆弱性问题。
- **组件间逻辑关系**的脆弱性。事件架构使系统中各组件的逻辑关系变得更加复杂。
- 事件驱动**容易进入死循环**，这是由编程逻辑决定的。
- **高并发**的脆弱性。虽然事件驱动可实现有效利用CPU资源，但是存在高并发事件处理造成的系统响应问题，而且，高并发容易导致系统数据不正确、丢失数据等现象。
- **固定流程**的脆弱性。因为事件驱动的可响应流程基本都是固定的，如果操作不当，容易引发安全问题。

5. MVC架构的脆弱性主要表现在:

- MVC架构的**复杂性**带来脆弱性。MVC架构增加了系统结构和实现的复杂性。比如说一个简单的界面，如果严格遵循MVC方式，使得模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率。
- **视图与控制器间紧密连接**的脆弱性。视图与控制器是相互分离但确是联系紧密的部件，没有控制器的存在，视图应用是很有限的。反之亦然，这样就妨碍了它们的独立重用。
- **视图对模型数据的低效率访问**的脆弱性。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问也将损害操作性能。

6. 微内核架构的脆弱性主要表现在:

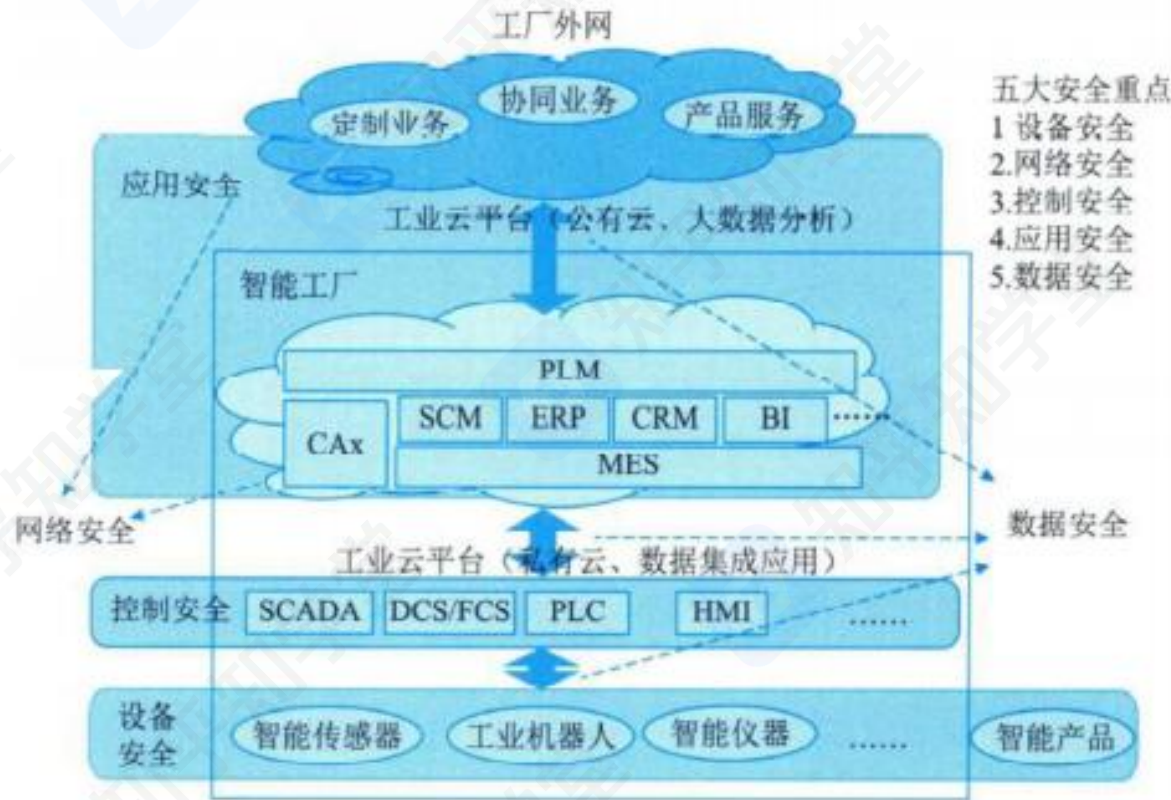
- 微内核架构**难以进行良好的整体化优化**。由于微内核系统的核心态只实现了最基本的系统操作，这样内核以外的外部程序之间的独立运行使得系统难以进行良好的整体优化。
- 微内核系统的进程间通信开销也较单一内核系统要大得多。从整体上看，在当前硬件条件下，微内核在效率上的损失小于其在结构上获得的收益。
- 通信损失率高。微内核把系统分为各个小的功能块，从而降低了设计难度，系统的维护与修改也容易，但通信带来的效率损失是一个问题。

7. 微服务架构的脆弱性主要表现在:

- 开发人员需要处理**分布式系统的复杂结构**。
- 开发人员要**设计服务之间的通信机制**，通过写代码来处理消息传递中速度过慢或者不可用等局部实效问题。
- **服务管理的复杂性**，在生产环境中要管理多个不同的服务实例，这意味着开发团队需要全局统筹。

安全架构设计案例分析

- 跨区域的安全生产管理是大型集团企业面临的主要生产问题。大型企业希望通过云计算平台实现异地的设计、生产、制造、管理和数据处理等，并确保企业内部生产的安全、保密和数据的完整。
- 混合云架构往往被大型企业所接受。混合云融合了公有云和私有云，是近年来云计算的主要模式和发展方向。我们知道私有云主要是面向企业用户，出于安全考虑，企业更愿意将数据存放在私有云中，但是同时又希望可以获得公有云的计算资源，在这种情况下混合云被越来越多地采用，它将公有云和私有云进行混合和匹配，以获得最佳的效果，这种个性化的解决方案，达到了既省钱又安全的目的。
- 下图给出了大型企业采用混合云技术的安全生产管理系统的架构，企业由**多个跨区域的智能工厂和公司总部组成**，公司总部负责相关业务的管理、协调和统计分析，而**每个智能工厂负责智能产品的设计与生产制造**。智能工厂内部采用私有云实现产品设计、数据共享和生产集成等，公司总部与智能工厂间采用公有云实现智能工厂间、智能工厂与公司总部间的业务管理、协调和统计分析等。
- 整个安全生产管理系统架构由四层组成：
 - 设备层：主要是指用于智能工厂生产产品所需的相关设备；
 - 控制层：主要是指智能工厂生产产品所需要建立的一套自动控制系统，控制智能设备完成生产工作；
 - 设计/管理层：是指智能工厂各种开发、业务控制和数据管理功能的集合，实现数据集成与应用；
 - 应用层：主要是指在云计算平台上进行信息处理，有两个核心功能，一是“数据”，二是“应用”。





05

Redis数据库



✓ Redis常见场景详解.pdf



✓ Redis的基础理论知识.pdf



T H E E N D

功不唐捐，玉汝于成！

• 开 启 新 征 程 •

