应用架构设计

2016年05月14日 10:57:28

阅读数：6334

如何实现大型网站架构设计的负载均衡- http://blog.csdn.net/t4i2b10X4c22nF6A/article/details/79062766 大型网站负载均衡的利器：全局负载均衡系统（GSLB）；内容缓存系统（CDN）；服务器负载均衡系统（SLB） 服务器负载均衡系统的常见调度算法： 轮询（Round Robin）；加权轮询（Weighted Round Robin）；最少连接（Least Connections）；加权最少连接（Weighted Least Connections）

无架构，不系统，架构是大型系统的关键。从形上看，架构是系统的骨架，支撑和链接各个部分；从神上看，架构是系统的灵魂，深刻体现业务本质。

架构可细分为业务架构、应用架构、技术架构，业务架构是战略，应用架构是战术，技术架构是装备。其中应用架构承上启下，一方面承接业务架构的落地，另一方面影响技术选型。

如何针对当前需求，选择合适的应用架构，如何面向未来，保证架构平滑过渡，这个是软件开发者，特别是架构师，都需要深入思考的问题。本文基于作者在大型互联网系统的实践和思考，和大家一起探讨应用架构的选型。

本文主要内容包括：

- 应用架构本质
- 单体式
- 分布式
- SOA架构
- SOA落地方式
- 应用架构进化

应用架构本质

应用作为独立可部署的单元，为系统划分了明确的边界，深刻影响系统功能组织、代码开发、部署和运维等各方面，应用架构定义系统有哪些应用、以及应用之间如何分工和合作。

分有两种方式，一种是水平分，按照功能处理顺序划分应用，比如把系统分为web前端/中间服务/后台任务，这是面向业务深度的划分。另一种是垂直分，按照不同的业务类型划分应用，比如进销存系统可以划分为三个独立的应用，这是面向业务广度的划分。

应用的合反映应用之间如何协作，共同完成复杂的业务case，主要体现在应用之间的通讯机制和数据格式，通讯机制可以是同步调用/异步消息/共享DB访问等，数据格式可以是文本/XML/JSON/二进制等。

应用的分偏向于业务，反映业务架构，应用的合偏向于技术，影响技术架构。分降低了业务复杂度，系统更有序，合增加了技术复杂度，系统更无序。

应用架构的本质是通过系统拆分，平衡业务和技术复杂性，保证系统形散神不散。

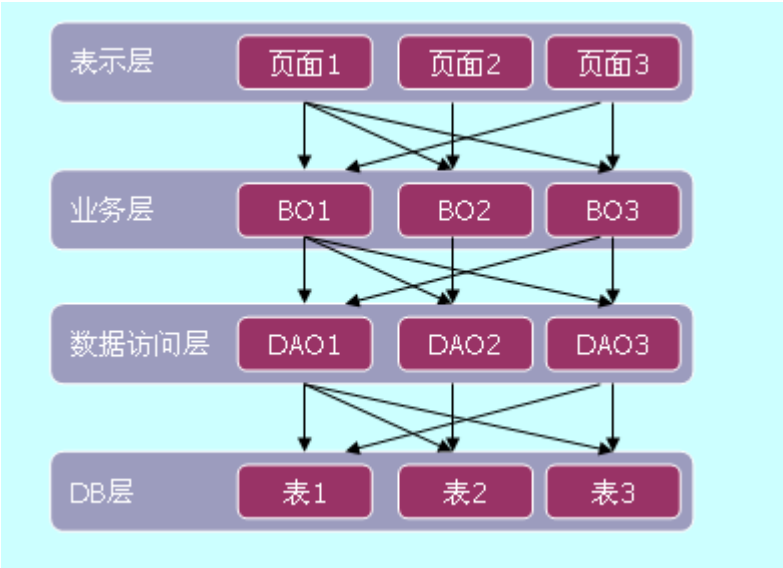
系统采用什么样的应用架构，受业务复杂性影响，包括企业发展阶段和业务特点；同时受技术复杂性影响，包括IT技术发展阶段和内部技术人员水平。业务复杂性(包括业务量大)必然带来技术复杂性，应用架构目标是解决业务复杂性的同时，避免技术太复杂，确保业务架构落地。

常见的应用架构有多种，下面根据系统拆分方式，以及如何平衡业务与技术的角度进行分析，讨论各自的适用性，给企业应用架构选型提供参考。

单体式应用

1. 架构模型

系统只有一个应用，相应地，代码放在一个工程里管理；打包成一个应用；部署在一台机器；在一个DB里存储数据。单体式应用的架构如下图所示：



单体式应用采用分层架构，按照调用顺序，从上到下一般为表示层、业务层、数据访问层、DB层，表示层负责用户体验，业务层负责业务逻辑，数据访问层负责DB层的数据存取。

单体应用在水平方向上，上下层之间职责划分清晰；但垂直方向上缺乏清晰的边界，上下层模块之间是多对多的依赖关系，比如业务模块1 (图中BO1)可能调用数据层所有模块DAO1~3， DAO1也可能被业务层所有模块BO1~3调用。

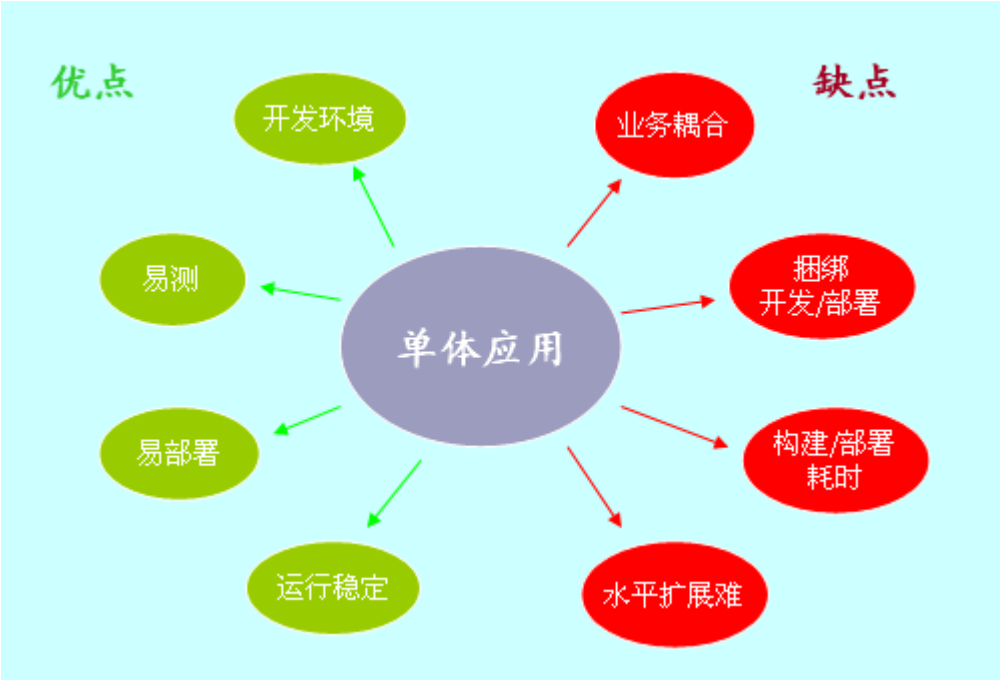
单体应用通过水平分层，降低了业务复杂性；同时模块之间是进程内部调用，技术实现简单。

但单体应用对系统的切分不彻底，只有水平切分，并且是逻辑上，因此适合业务比较单一，但深度上比较复杂的系统，比如TCP/IP网络通讯，从应用层/传输层/网络层/链路层，层层推进，类似这样的系统可以方便地增加水平层次去适配。

对于广度上复杂的业务，由于缺乏垂直切分，强行把不同业务绑定在一起，整个系统神散形不散，带来一系列问题。比如OTA网站包含机票/酒店/旅游等多个垂直业务板块，每块都比较独立，就不适合放在一起开发维护。

1. 优缺点

单体式应用的优点和缺点都很鲜明，如下图所示。



单体式应用的优点明显：

- 现有IDE都是集成开发环境，非常适合单体式应用，开发、编译、调试一站式搞定。
  一个应用包含所有功能，容易测试和部署。
- 运行在一个物理节点，环境单一，运行稳定，故障恢复简单。

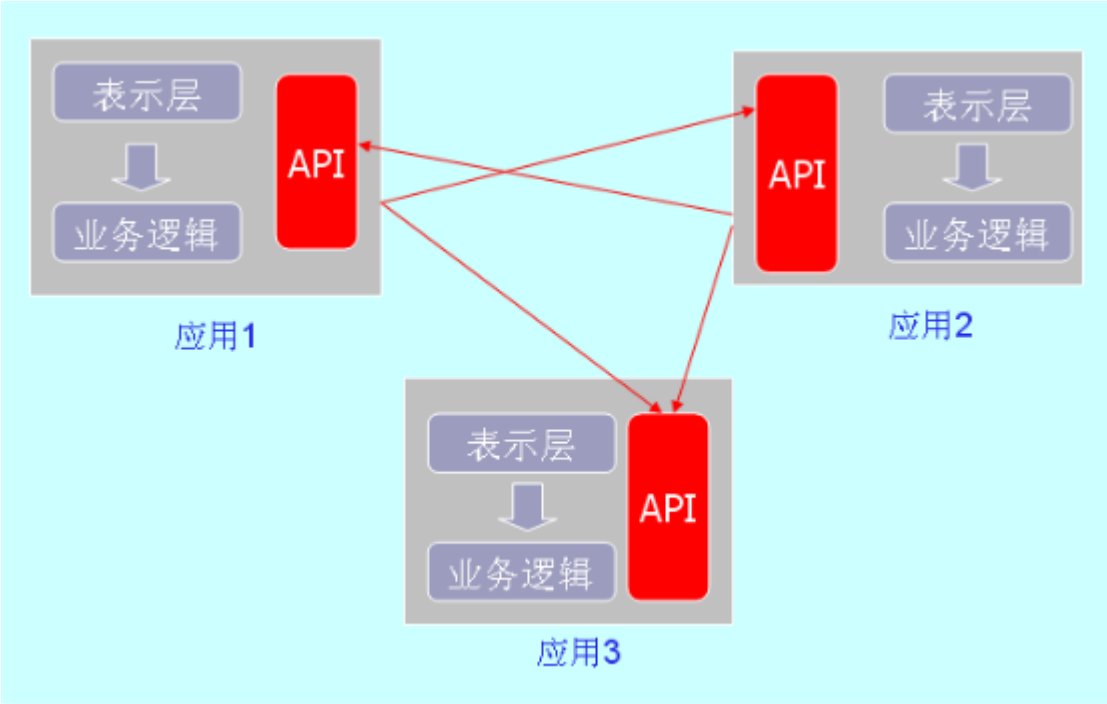
单体式应用的缺点也明显：

- 业务边界模糊，模块职责不清晰，当系统逐渐变大，代码依赖复杂，难以维护。
- 所有人同时在一个工程上开发，容易发生代码修改冲突，依赖复杂导致项目协调困难，并且局部修改影响不可知，需要全覆盖测试，需要重新部署，难以支持大团队并行开发。
- 当系统很大时，编译和部署耗时。
- 应用水平扩展难，一方面状态在应用内部管理，无法透明路由；另一方面，不同模块对资源需求差异大，当业务量增大时，一视同仁地为所有模块增加机器导致硬件浪费。

作者之前曾在一家大型电商公司工作，当时整个网站是一个单体应用，有数百万行代码，有专门的团队负责代码合并，有专门的团队负责编译脚本开发，有一套复杂的火车模型协调不同团队，整套流程体系很精密很复杂，但这何尝不是单体应用的无奈和代价。

分布式架构应用

1. 架构模型



在分布式应用架构中，应用相互独立，每个应用代码独立开发，独立部署，应用通过有限的API接口互相关联。API接口属于应用一部分，一般和表示层处于同一层次，两者共享业务逻辑层，API和表示层采用同样的web端技术，通讯协议一般使用HTTP，数据格式是JSON，应用集成方式比较简化。

分布式架构首先对系统按照业务进行垂直切分，对广度上复杂的业务实现物理解耦，应用内部还是水平切分，对深度上复杂的业务实现逻辑解耦。分布式架构也可以解决业务量大的问题，对于某些高并发/大流量系统，把系统切分为不同应用，针对资源需求特点（比如CPU/IO/存储密集型），通过加强硬件配置满足不同应用的需求，避免一刀切方式带来的资源浪费。

技术上，API采用标准的HTTP/JSON进行通讯，调用双方实现难度都不大，但是API 一般是“裸奔”的，在系统层面，调用依赖关系不透明，调用可靠性缺乏保障，因此只适用应用之间依赖链路少，调用量不大的系统，即应用之间耦合确实够松的系统。

1. 优缺点

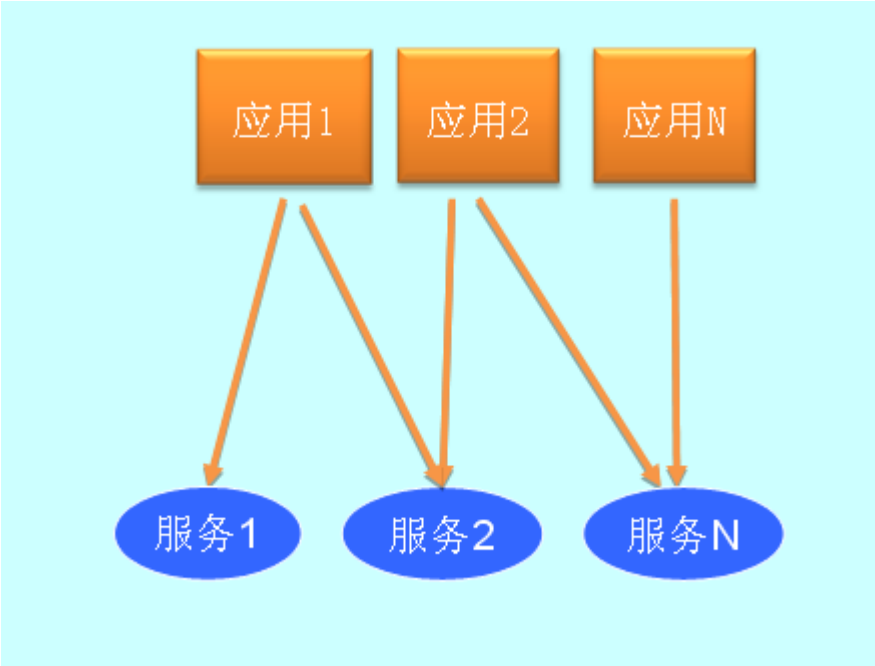
分布式架构每个应用内部高内聚，独立开发、测试和部署，支持开发敏捷；同时应用之间松耦合，业务边界清晰，业务依赖明确，支持大项目并行开发，实现业务敏捷。

在分布式架构中，应用的表示层和API没有物理分离，需要同时满足自身业务需求和关联业务需求，相互影响，比如API接口会随着外部应用的需求经常变化，这会导致整个应用重新部署。

运行时，API以HTTP/REST方式通过网络对外提供接口，其通信可靠性和数据的封装性相对于进程内调用比较差，如果没有一些可靠的技术机制，如路由保障/失败重试/监控等， 裸奔的API方式将严重影响系统整体可用性。

SOA架构

1. 架构模型



广义上，SOA也是分布式应用架构一种，但内涵不同。

这里有两种类型“应用”，应用1~N是前端应用，面向用户，服务1~N是service，只提供业务逻辑和数据。这些应用都是独立部署，前端应用不再通过API直接关联，而是通过后端服务共享业务逻辑。

此外相对于“裸奔”的API，SOA架构提供配套的服务治理，包括服务注册、服务路由、服务授权、服务降级、服务监控等等。这些功能通过专门的中间件支持，有中心化和去中心化两种方式，具体技术实现机制和适用场景，网上有很多专门介绍，这里就不展开了。

SOA架构在分布式架构垂直切分的基础上，进一步把原来单体应用的业务逻辑层独立成service，做到物理上的彻底分离。

每个service专注于特定职责，实现系统核心业务逻辑，service之间通过互相调用，可以完成复杂业务逻辑，解决业务深度上的问题；同时service面向众多的应用，以共享的方式支持逻辑复用。所以，SOA架构既体现业务的分，又体现业务的合，更多地从业务整体上考虑系统拆分。

相比分布式应用架构，基于SOA的系统有大量的service应用，整个系统基于服务调用，所以对服务依赖的透明性和服务调用的可靠性提出很高要求，需要专门的SOA框架支持，还需要配套的监控体系和自动化的运维系统支持，技术复杂性高，SOA架构可以集中体现一个企业的IT技术能力。

1. 优缺点

SOA架构优缺点如下图所示：

相比较普通API方式，SOA架构更进一步：

1)每个service都是浓缩的精华，聚焦某方面核心业务，同时以复用的方式供整个系统共享。 2)服务作为独立的应用，独立部署，接口清晰，很容易做自动化测试和部署。 3)服务是无状态的，很容易做水平扩展；通过容器虚拟化技术，实现故障隔离和资源高效利用，业务量大的时候，加机器即可。 4)基于SOA的系统可以根据服务运行情况，灵活调控服务资源，包括服务上下架、服务升降级等，使系统真正具备可运营的能力。

当然天下没有免费的午餐，SOA也带来了额外复杂性和弊端：

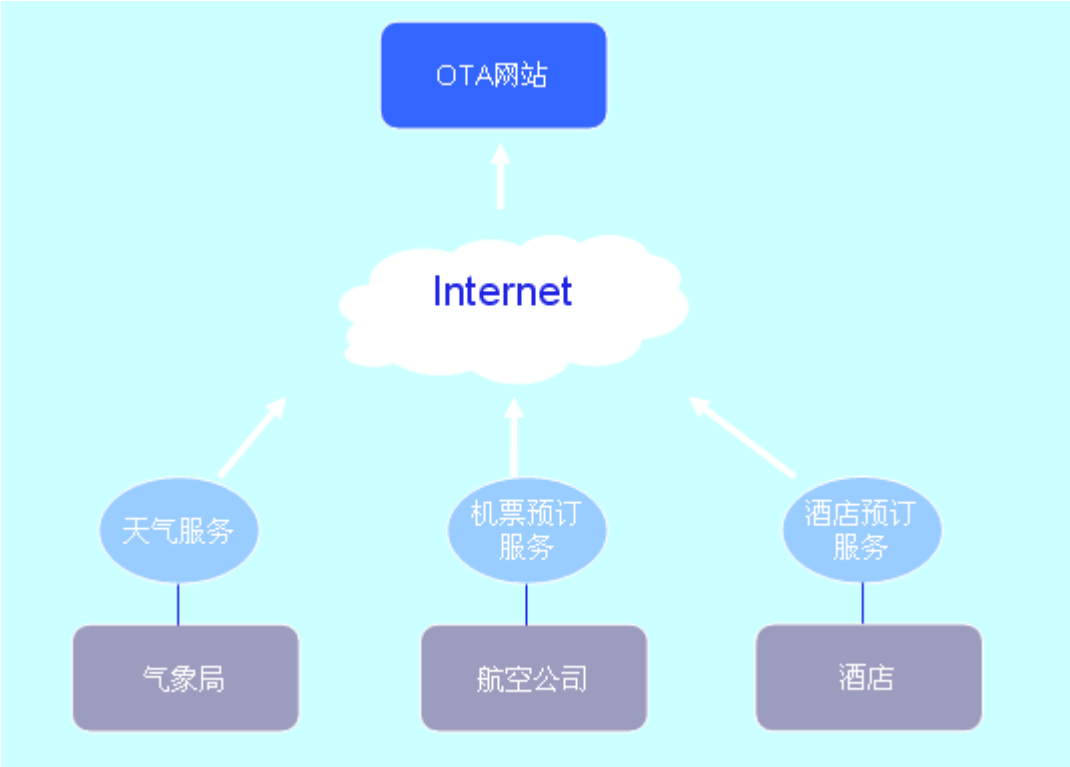
1)系统依赖复杂，给开发/测试/部署带来一系列挑战。 2)端到端的调用链路长，可靠性降低，依赖网络状况、服务框架及具体service的质量。 3)分布式数据一致性和分布式事务支持困难，一般通过最终一致性简化解决。 4)端到端的测试和排障复杂，SOA对运维提出更高要求。

SOA落地方式

在实践中，SOA架构不断深入发展，具体落地形式也多种多样。

1)面向外部SOA

SOA的前身是web service，web service初衷是企业之间通过Internet进行互联，如下图所示：

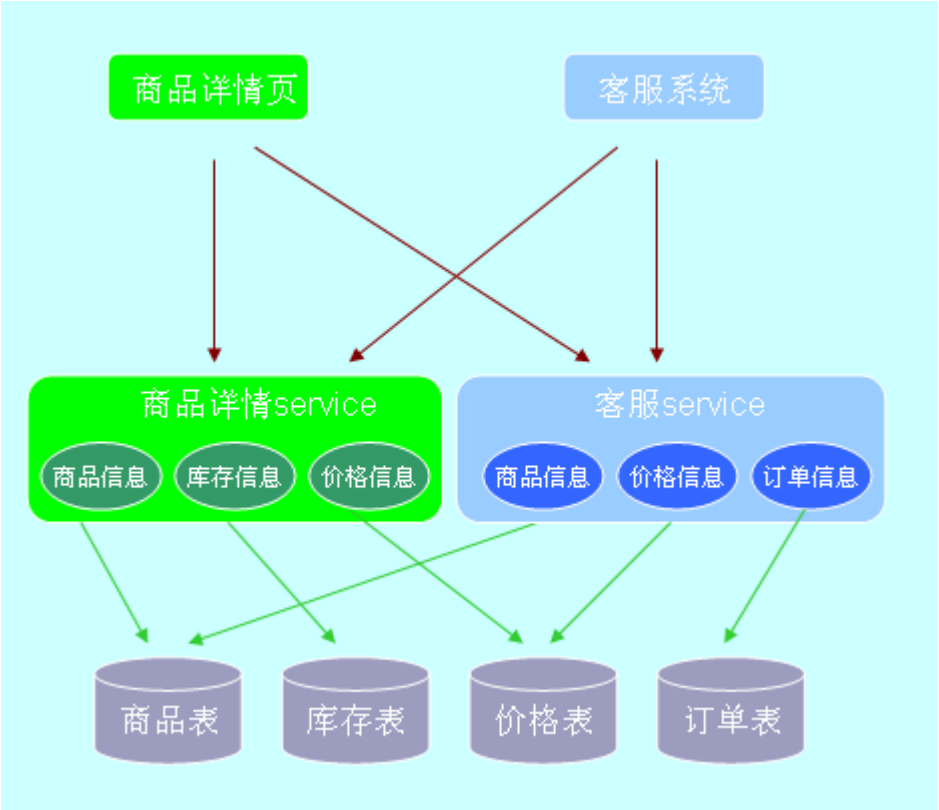


每个公司把自己的优势资源通过web service发布，如图中天气服务/机票服务/酒店预订服务来自不同公司，其他企业可以直接调用服务或整合多个服务，实现企业间资源共享。

2)面向应用SOA

面向应用SOA把原单体应用里的业务逻辑层剥离出来，作为单独的服务对外提供。 举一个电商的例子，这里有两个应用，顾客使用的商品详情页，展示商品的信息、商品库存，商品价格；内部客服人员使用的客服系统，根据顾客来电要求，修改订单，同时也需要获取商品的基本信息、价格信息等。

经过SOA改造后，应用架构如下图所示：

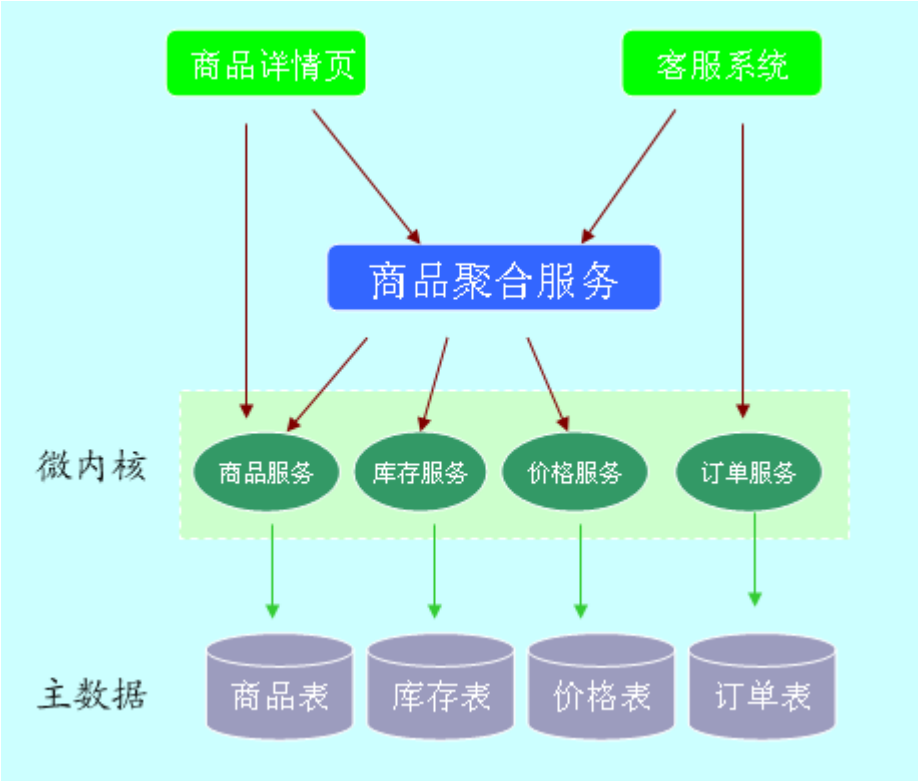


这里的service实现底层数据对于前端页面的透明化，表示层和业务逻辑各自独立维护，同时全部业务逻辑对其他应用开放，新应用可以自由整合来自多个服务的接口，快速支持业务创新。

面向应用的SOA架构对系统进行物理上的水平切分，结果是表示层单飞，逻辑层对外全开放。但每个service是原系统业务逻辑的封装，接口设计面向原应用的业务case，如果其它应用的需求有差异，则自己创建service访问底层数据。如此导致service职责不够聚焦，类似的接口分散化，同时底层数据表被多方访问，数据模型修改困难，数据一致性难以保障。

最终系统整体依赖复杂，容易形成网状结构，修改时，往往牵一发动全身。

3)微内核SOA 每个企业都有自己的核心数据，比如对于电商系统来说，用户/商品/订单/库存/价格都是核心数据，称之为主数据。微内核SOA聚焦各类主数据，封装相关表的所有访问，架构示意如下：



每个服务独占式地封装对应主数据表的访问，这些服务构成系统的基础服务，一起组成系统的微内核，供所有上层应用共享。

微内核服务是原子服务，接口粒度比较细，可以在其上构造聚合服务，为上层应用提供粗粒度服务。可以是信息聚合，比如图中商品聚合服务整合商品的基本信息/库存/价格；也可以是流程聚合，比如下单接口，调用来自多个服务的接口，共同完成复杂的下单操作。

这里服务是分层次的，聚合服务是上层，基础服务是底层，依赖规则如下：

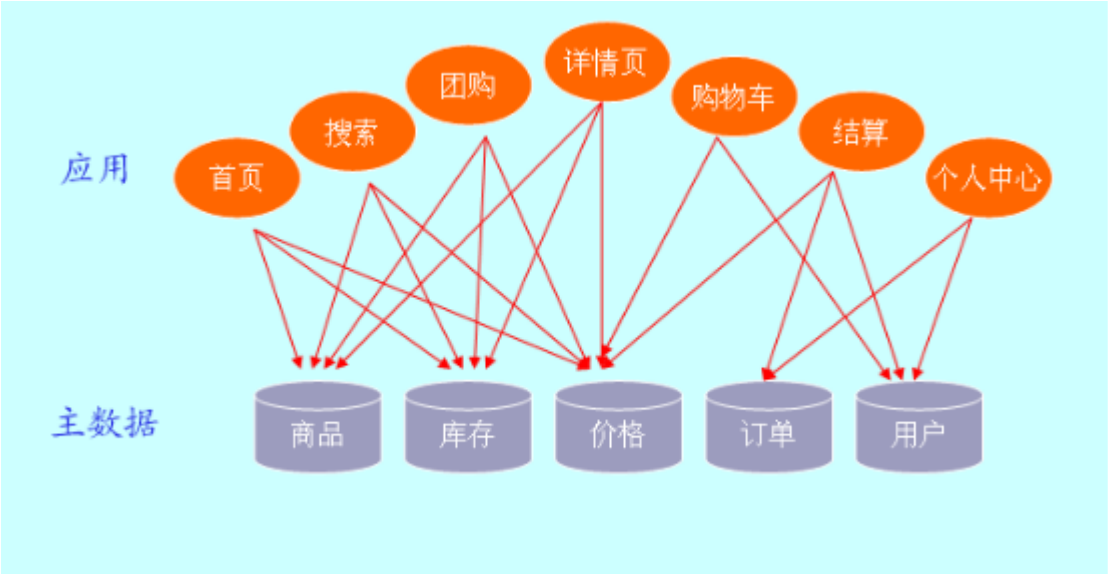
- 上层服务可以调用同层服务和基础服务
- 基础服务是原子服务，不可相互调用
- 前端应用可调用聚合服务和跨层调用基础服务

微内核的微表示服务数量有限，接口粒度细；内核表示这些基础服务处于调用底层，负责核心数据和业务，这和操作系统的内核概念上类似，主数据相当于核心的硬件，如cpu/内存/外存等，微内核的各个基础服务分别负责这些核心资源的管理，屏蔽底层的复杂性，对上层应用提供统一入口和透明化访问。

最近微服务很火，其特征是职责单一、接口粒度细、轻量级通讯协议等，微内核SOA架构有微服务的形，同时有业务内核的神，是架构形散神不散思想的很好体现，这个在淘宝、京东、一号店等大型电商系统都已有丰富实践。

4)方式比较

面向企业外部SOA，业务场景有特殊性，不深入分析，这里主要比较面向应用SOA和微内核SOA的区别，一个大型B2C电商系统，应用和主数据是多对多依赖关系，如下图所示：



面向应用的服务从特定应用出发，整合应用对相关数据的访问需求；微内核服务从特定主数据为中心，收敛各个业务对数据的访问需求。

在面向应用的服务设计下，数据表的访问入口是发散的，来自多个应用，这带来一系列弊端： 1)数据模型碎片化 每个应用都会基于自己的需求，往表里加字段，很多电商的商品表/订单表有多达200个字段，都是野蛮增长，缺少控制的结果。 2)数据模型修改困难 类似的访问需求散布在多个服务，缺乏整合，同时表schema修改会影响很多服务和应用。 3)连接资源利用率低 多个服务直连数据库，并且每个服务会尽可能多地配置连接数，在应用数量多，业务并发量大的情况下，往往导致数据库连接数不够。

微内核SOA通过收敛对主数据的访问，保证数据模型一致性、优化接口和有效利用数据库连接资源。同时通过服务分层，简化系统依赖关系。更为重要的是，微内核服务保证了业务模型的一致性，比如电商系统的商品体系，包含单品/系列商品/组合商品/搭售商品/虚拟商品等一系列复杂概念，这些复杂逻辑在基础商品服务里处理，对上层业务透明一致。

如果业务模式简单，应用数量少，特定主数据的访问绝大多数(比如说80%)来自某个应用，则服务设计以应用为中心是可以的，不利影响比较小。

对于大型互联网系统，业务广度和深度都复杂，但无论多复杂的系统，主数据类型都是有限的，可以通过聚焦有限的基础业务，以此支持无限的应用业务，结果是底层业务模型稳定，上层业务可以灵活扩展。

面向应用的服务设计是SOA初级阶段，从具体业务着手，自底向上，难度小；微内核服务设计是SOA高级阶段，从全局着手，对业务和数据模型高度抽象，自顶向下，难度大。

值得注意的是，在提供基础服务同时，每个应用也可以创建自己需要的服务（但主数据的访问必须通过基础服务），所以微内核的服务和面向应用的服务可以有机结合在一起，当业务应用变得很多，并且不断增长，可以考虑逐步往基础服务过渡，整合特定主数据有关的业务逻辑。

顺便提一下，应用架构会影响组织架构，如果采用面向应用的服务设计，具体service一般由相关应用的团队负责；如果是微内核的服务设计，一般由单独的共享服务部门负责所有基础服务开发，和各个业务研发部门并列，保证设计的中立性和需求响应的及时性。

应用架构的进化

软件是人类活动的虚拟，业务架构是生产活动的体现，应用架构是具体分工合作关系的体现。 单体应用类似原始氏族时代，氏族内部有简单分工，氏族之间没有联系；分布式架构类似封建社会，每个家庭自给自足，家庭之间有少量交换关系；SOA架构类似工业时代，企业提供各种成品服务，我为人人，人人为我，相互依赖。微内核的SOA架构类似后工业时代，有些企业聚焦提供水电煤等基础设施服务，其他企业在之上提供生活服务，依赖有层次。

业务架构是生产力，应用架构是生产关系，技术架构是生产工具。业务架构决定应用架构，应用架构需要适配业务架构，并随着业务架构不断进化，同时应用架构依托技术架构最终落地。

企业一开始业务比较简单，比如进销存，此时面向内部用户，提供简单的信息管理系统（MIS），支持数据增删改查即可，单体应用可以满足要求。

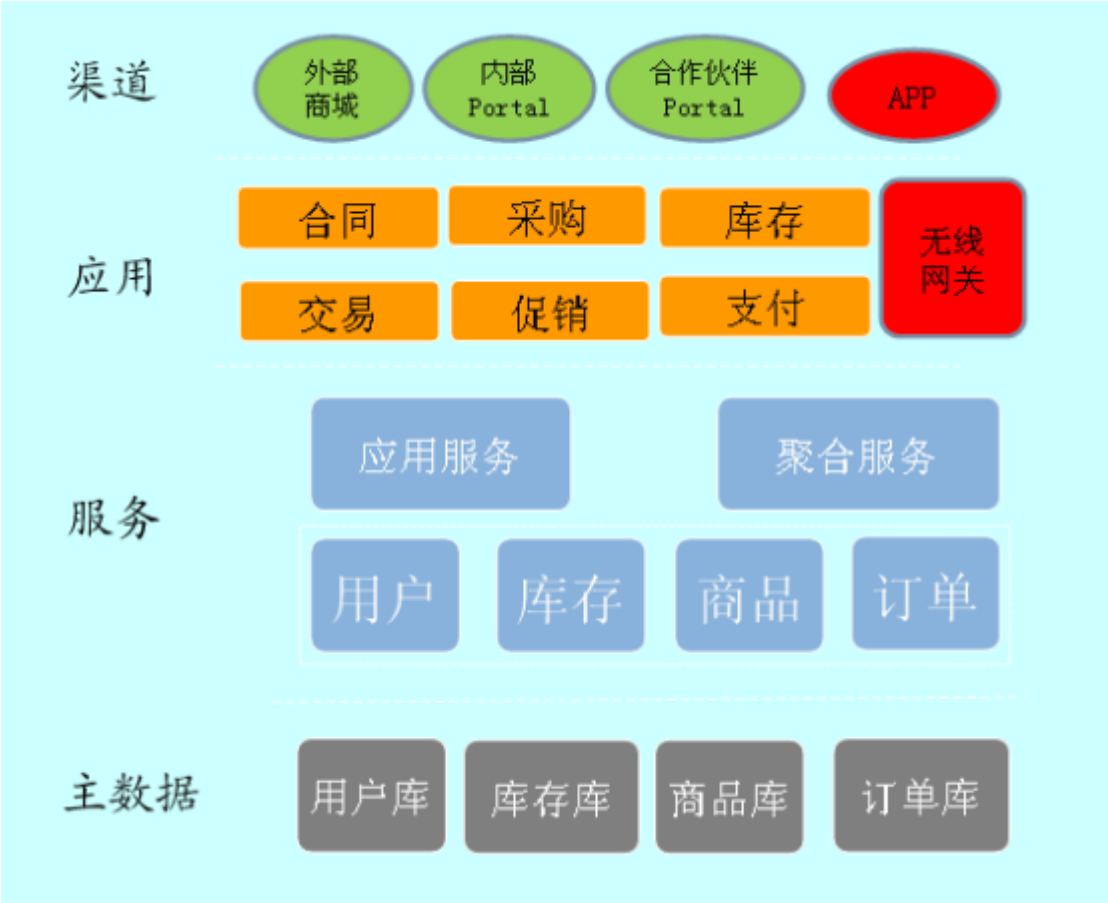
随着业务深入，进销存每块业务都变复杂，同时新增客户关系管理，以更好支持营销，业务的深度和广度都增加，这时需要对系统按照业务拆分，变成一个分布式系统。

更进一步，企业转向互联网+战略，拓展在线交易，线上系统和内部系统业务类似，没必要重做一套，此时把内部系统的逻辑做服务化改造，同时供线上线下系统使用，变成一个简单的SOA架构。

紧接着业务模式越来越复杂，订单、商品、库存、价格每块玩法都很深入，比如价格区分会员等级，访问渠道（无线还是PC），销售方式（团购还是普通）等，还有大量的价格促销，这些规则很复杂，容易相互冲突，需要把分散到各个业务的价格逻辑进行统一管理，以基础价格服务的方式透明地提供给上层应用，变成一个微内核的SOA架构。

同时不管是企业内部用户，还是外部顾客所需要的功能，都由很多细分的应用提供支持，需要提供portal，集成相关应用，为不同用户提供统一视图，顶层变成一个AOA的架构（application orientated architecture）。

随着业务和系统不断进化，最后一个比较完善的大型互联网应用架构如下图所示：



最终整个系统化整为零，形神兼备，支持积木式拼装，支持开发敏捷和业务敏捷。

应用架构，需要站在业务和技术中间，在正确的时间点做正确的架构选择，保证系统有序进化。