第一章微服务架构介绍

（Spring Cloud 初级）

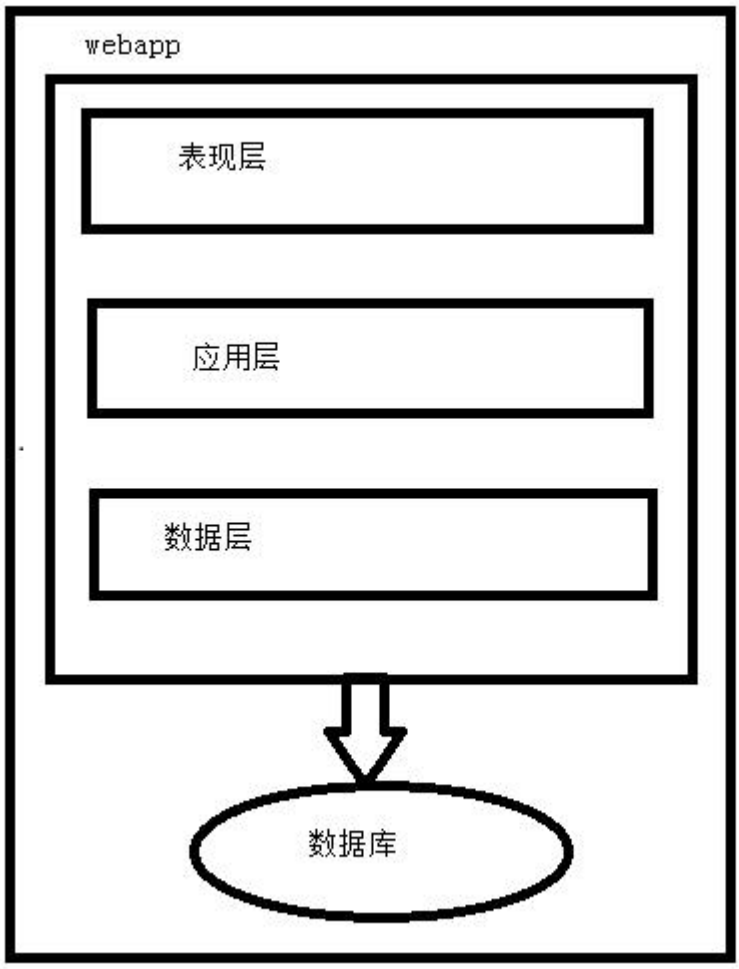
一、 单体架构

单体架构也称之为单体系统或者是单体应用。就是一种把系统中所有的功能、模块耦合在一个应用中的架构方式。

1 单体架构特点

1.1打包成一个独立的单元(导成一个唯一的 jar 包或者是 war 包)

1.2会一个进程的方式来运行



MVC架构

2 单体架构的优点、缺点

2.1优点

2.1.1项目易于管理

2.1.2部署简单

2.2缺点

2.2.1测试成本高

2.2.2可伸缩性差

2.2.3可靠性差

2.2.4迭代困难

2.2.5跨语言程度差

2.2.6团队协作难

二、 微服务架构

1 什么是微服务

微服务是一种架构风格。一个大型的复杂软件应用，由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好的完成该任务。

2 架构风格

项目的一种设计模式。

2.1常见的架构风格

2.1.1客户端与服务端的

2.1.2基于组件模型的架构（EJB）

2.1.3分层架构(MVC)

2.1.4面向服务架构(SOA)

3 微服务特点：

3.1 系统是由多个服务构成

3.2 每个服务可以单独独立部署

3.3 每个服务之间是松耦合的。服务内部是高内聚的，外部是低耦合的。高内聚就是每个服务只关注完成一个功能。

4 微服务的优点、缺点

4.1 优点

4.1.1 测试容易

4.1.2 可伸缩性强

4.1.3 可靠性强

4.1.4 跨语言程度会更加灵活

4.1.5 团队协作容易

4.1.6 系统迭代容易

4.2缺点

4.2.1运维成本过高，部署数量较多

4.2.2接口兼容多版本

4.2.3分布式系统的复杂性

4.2.4分布式事务

三、 MVC、RPC、SOA、微服务架构之间的区别

1 MVC 架构

其实 MVC 架构就是一个单体架构。

代表技术：Struts2、SpringMVC、Spring、Mybatis 等等。

2 RPC 架构

RPC(Remote Procedure Call)：远程过程调用。他一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

代表技术：Thrift、Hessian 等等

3 SOA 架构

SOA(Service oriented Architecture):面向服务架构

ESB(Enterparise Servce Bus):企业服务总线，服务中介。主要是提供了一个服务于服务之间的交互。

ESB 包含的功能如：负载均衡，流量控制，加密处理，服务的监控，异常处理，监控告急等等。

代表技术：Mule、WSO2

4 微服务架构

微服务就是一个轻量级的服务治理方案。

代表技术：SpringCloud、dubbo 等等

四、 如何设计微服务以及设计原则

1) AKF 拆分原则
2) 前后端分离原则
3) 无状态服务
4) RestFul 的通信风格

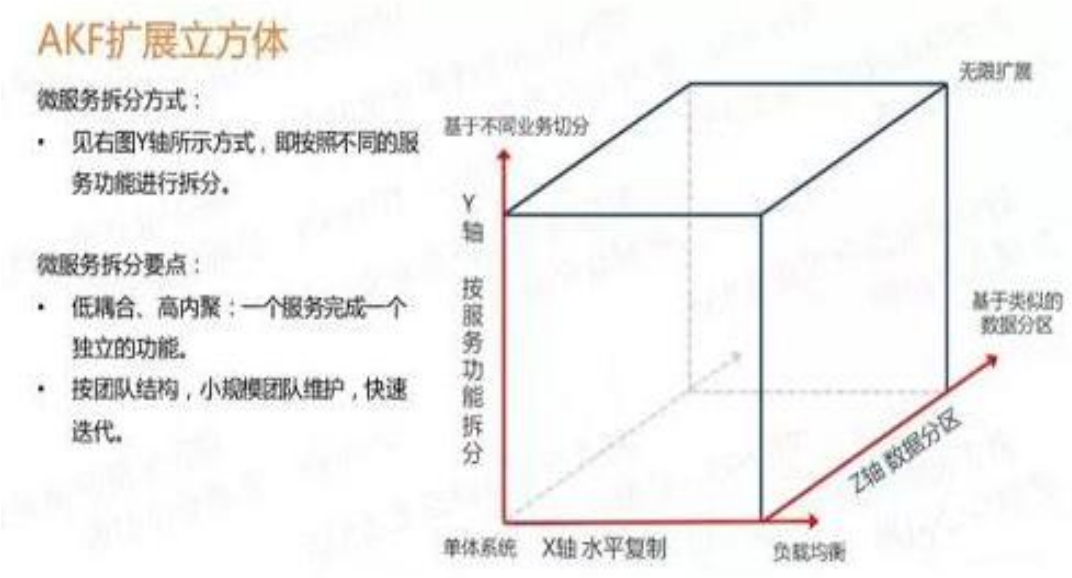
1 AKF 拆分原则

业界对于可扩展的系统架构设计有一个朴素的理念,就是：

通过加机器就可以解决容量和可用性问题。(如果一台不行那就两台)。

我是个段子：（世界上没有什么事是一顿烧烤不能解决的。如果有，那就两顿。）

这一理念在“云计算”概念疯狂流行的今天，得到了广泛的认可！对于一个规模迅速增长的系统而言，容量和性能问题当然是首当其冲的。但是随着时间的向前，系统规模的增长，除了面对性能与容量的问题外，还需要面对功能与模块数量上的增长带来的系统复杂性问题以及业务的变化带来的提供差异化服务问题。而许多系统，在架构设计时并未充分考虑到这些问题，导致系统的重构成为常态，从而影响业务交付能力，还浪费人力财力！对此，《可扩展的艺术》一书提出了一个更加系统的可扩展模型——**AKF 可扩展立方**（Scalability Cube）。这个立方体中沿着三个坐标轴设置分别为：X、Y、Z。



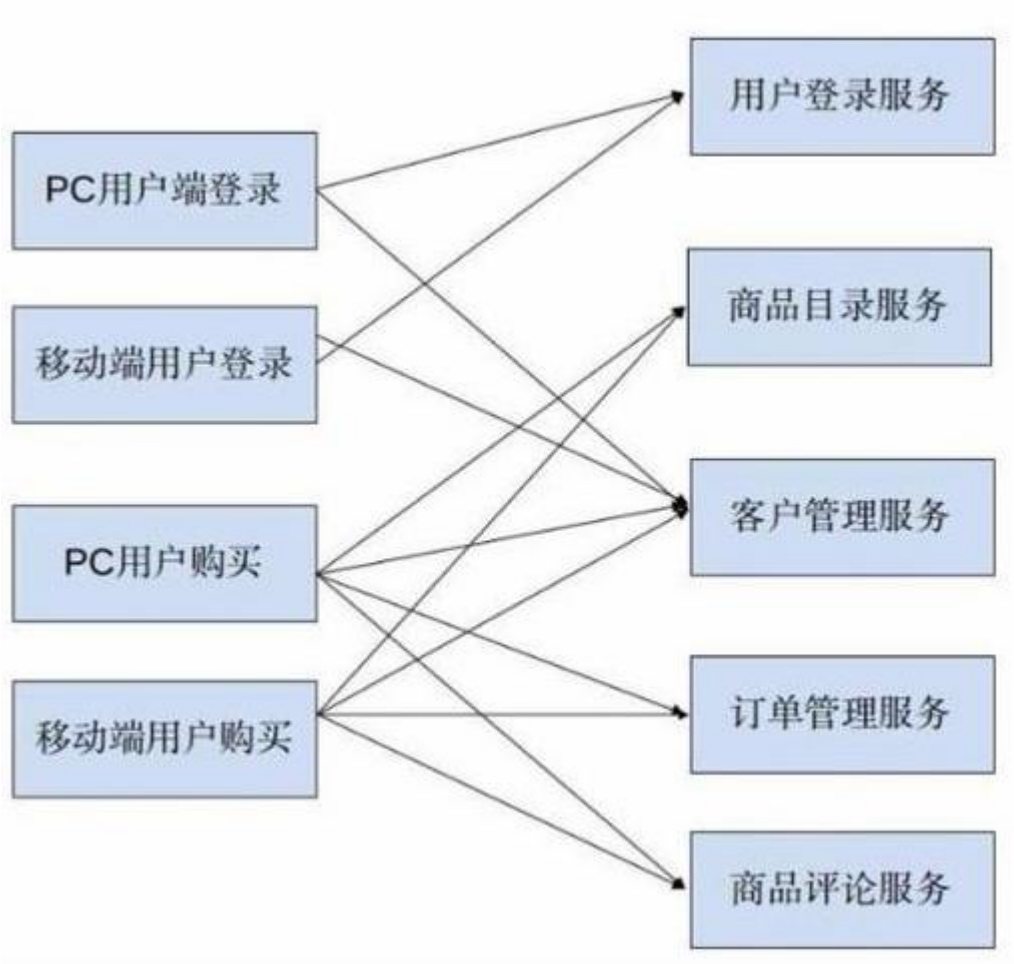
Y 轴（功能） —— 关注应用中功能划分，基于不同的业务拆分

X 轴（水平扩展） —— 关注水平扩展，也就是”加机器解决问题”

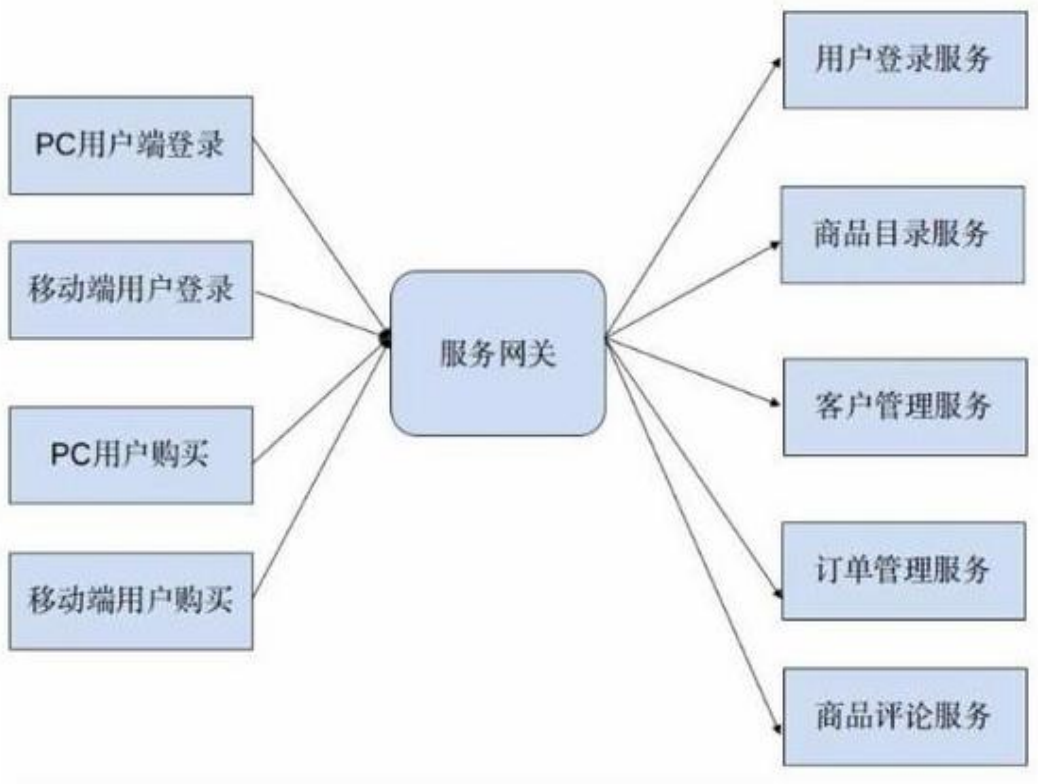
Z 轴（数据分区） —— 关注服务和数据的优先级划分，如按地域划分

1.1 Y 轴(功能)

Y 轴扩展会将庞大的整体应用拆分为多个服务。每个服务实现一组相关的功能，如订单管理、客户管理等。在工程上常见的方案是 **服务化架构(SOA)** 。比如对于一个电子商务平台，我们可以拆分成不同的服务，组成下面这样的架构：



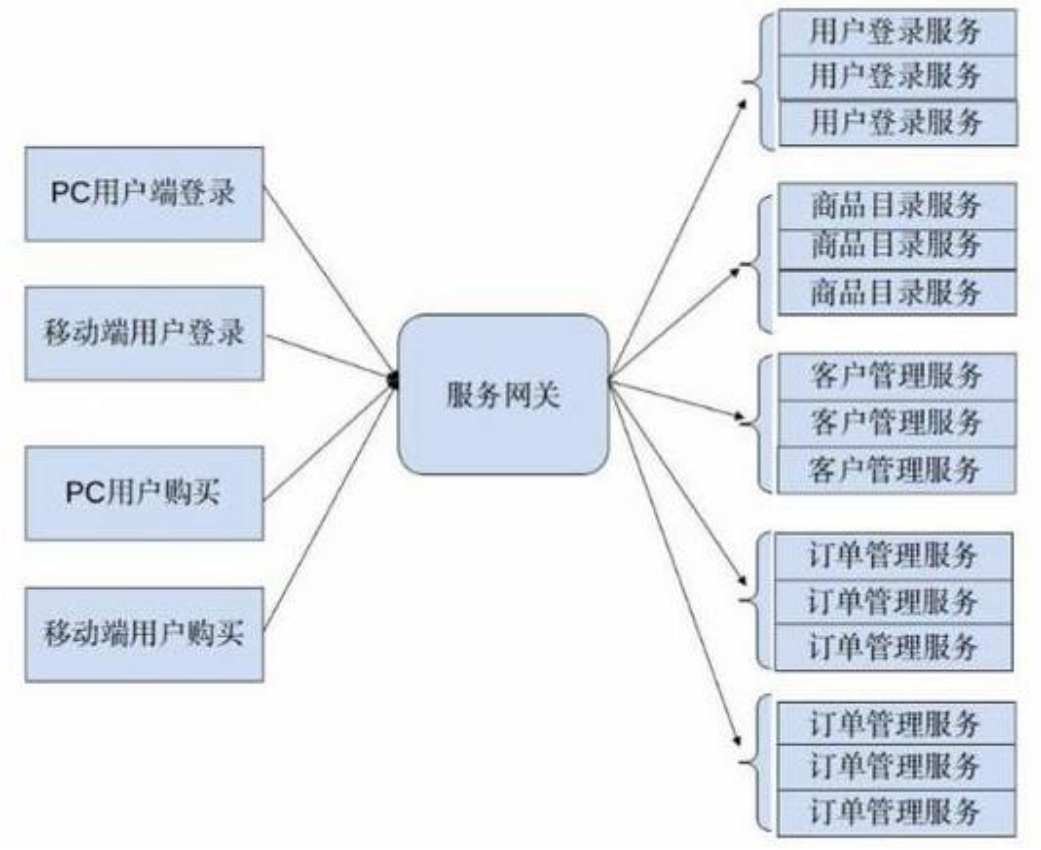
但通过观察上图容易发现，当服务数量增多时，服务调用关系变得复杂。为系统添加一个新功能，要调用的服务数也变得不可控，由此引发了服务管理上的混乱。所以，一般情况下，需要采用服务注册的机制形成服务网关来进行服务治理。系统的架构将变成下图所示：



1.2 X 轴(水平扩展)

X 轴扩展与我们前面朴素理念是一致的，通过绝对平等地复制服务与数据，以解决容量和可用性的问题。其实就是将微服务运行多个实例，做集群加负载均衡的模式。

为了提升单个服务的可用性和容量， **对每一个服务进行 X 轴扩展划分** 。



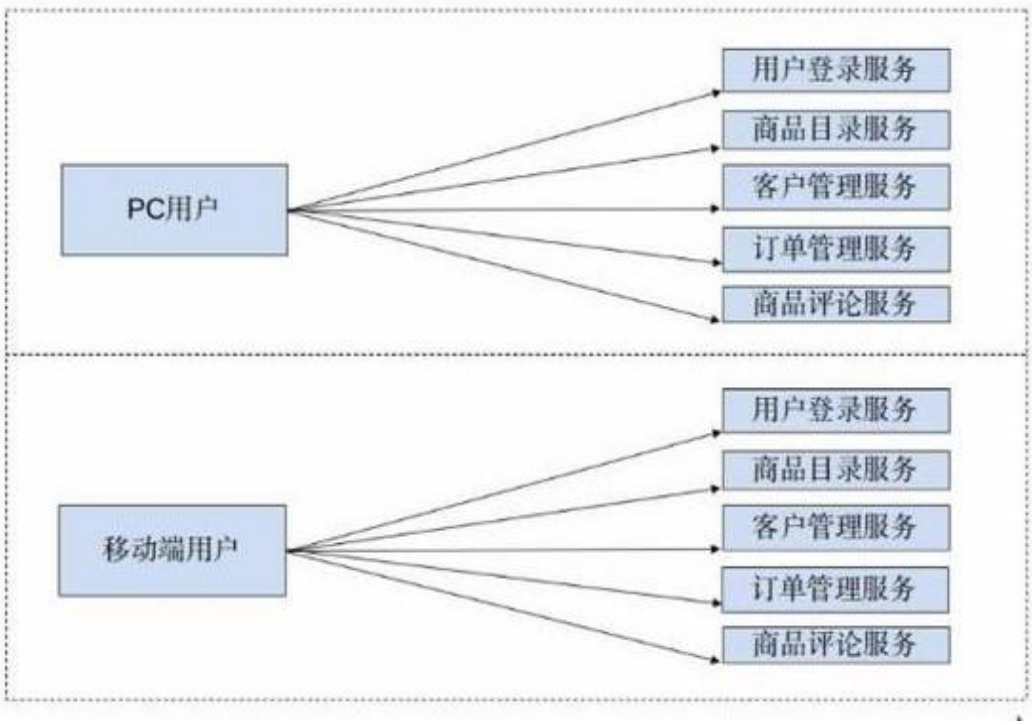
1.3 Z 轴(数据分区)

Z 轴扩展通常是指基于请求者或用户独特的需求，进行系统划分，并使得划分出来的子系统是相互隔离但又是完整的。以生产汽车的工厂来举例：福特公司为了发展在中国的业务，或者利用中国的廉价劳动力，在中国建立一个完整的子工厂，与美国工厂一样，负责完整的汽车生产。这就是一种 Z 轴扩展。

1.3.1工程领域常见的 Z 轴扩展有以下两种方案：

1.3.1.1 单元化架构

在分布式服务设计领域，一个单元（Cell）就是满足某个分区所有业务操作的自包含闭环。如上面我们说到的 Y 轴扩展的 SOA 架构，客户端对服务端节点的选择一般是随机的，但是，如果在此加上 Z 轴扩展，那服务节点的选择将不再是随机的了，而是每个单元自成一体。如下图：

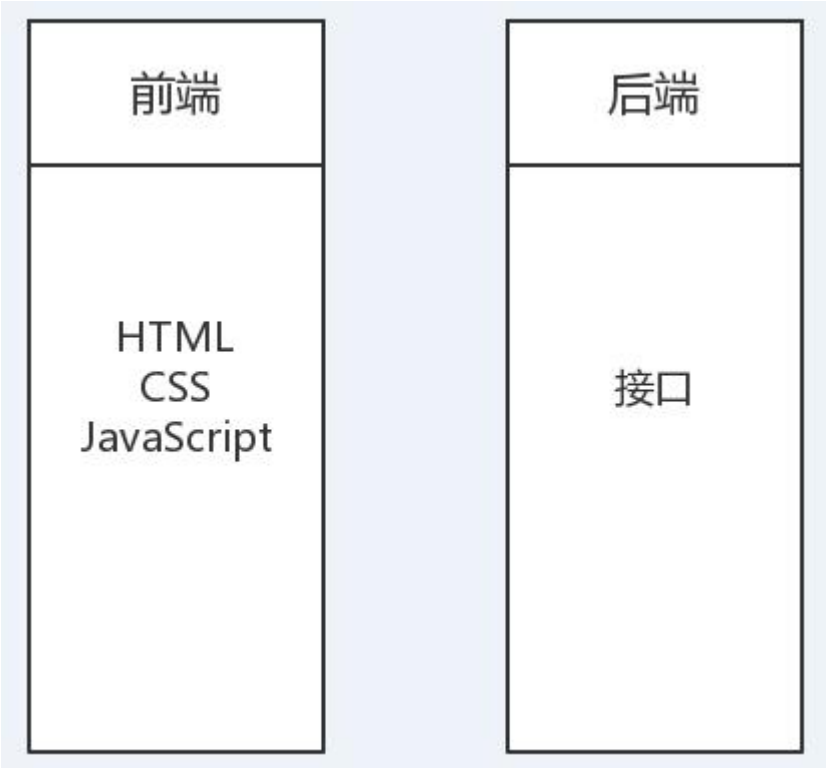


1.3.1.2 数据分区

为了性能数据安全上的考虑，我们将一个完整的数据集按一定的维度划分出不同的子集。 一个分区（Shard），就是是整体数据集的一个子集。比如用尾号来划分用户，那同样尾号的那部分用户就可以认为是一个分区。数据分区为一般包括以下几种数据划分的方式：

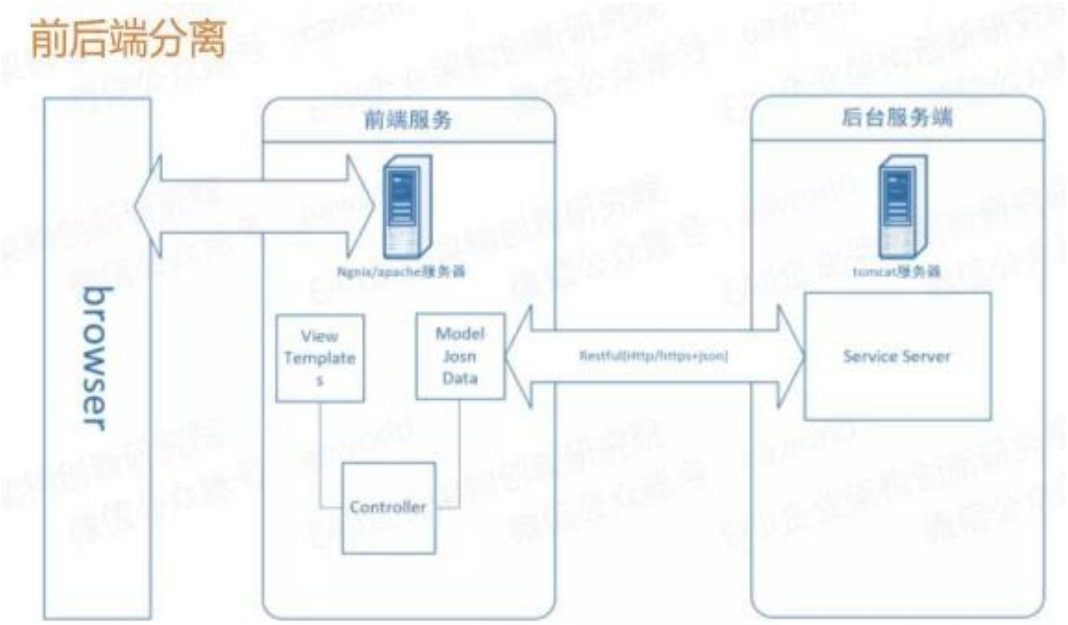
数据类型（如：业务类型）
数据范围（如：时间段，用户 ID）
数据热度（如：用户活跃度，商品热度）
按读写分（如：商品描述，商品库存）

2 前后端分离原则



何为前后端分离？前后端本来不就分离么？这要从尴尬的 jsp 讲起。分工精细化从来都是蛋糕做大的原则，多个领域工程师最好在不需要接触其他领域知识的情况下合作，才可能使效率越来越高，维护也会变得简单。jsp 的模板技术融合了 html 和 java 代码，使得传统 MVC 开发中的前后端在这里如胶似漆，前端做好页面，后端转成模板，发现问题再找前端，前端又看不懂 java 代码.....前后端分离的目的就是将这尴尬局面打破。

前后端分离原则，简单来讲就是前端和后端的代码分离，我们推荐的模式是最好采用物理分离的方式部署，进一步促使更彻底的分离。如果继续直接使用服务端模板技术，如：jsp，把 java、js、html、css 都堆到一个页面里，稍微复杂一点的页面就无法维护了。



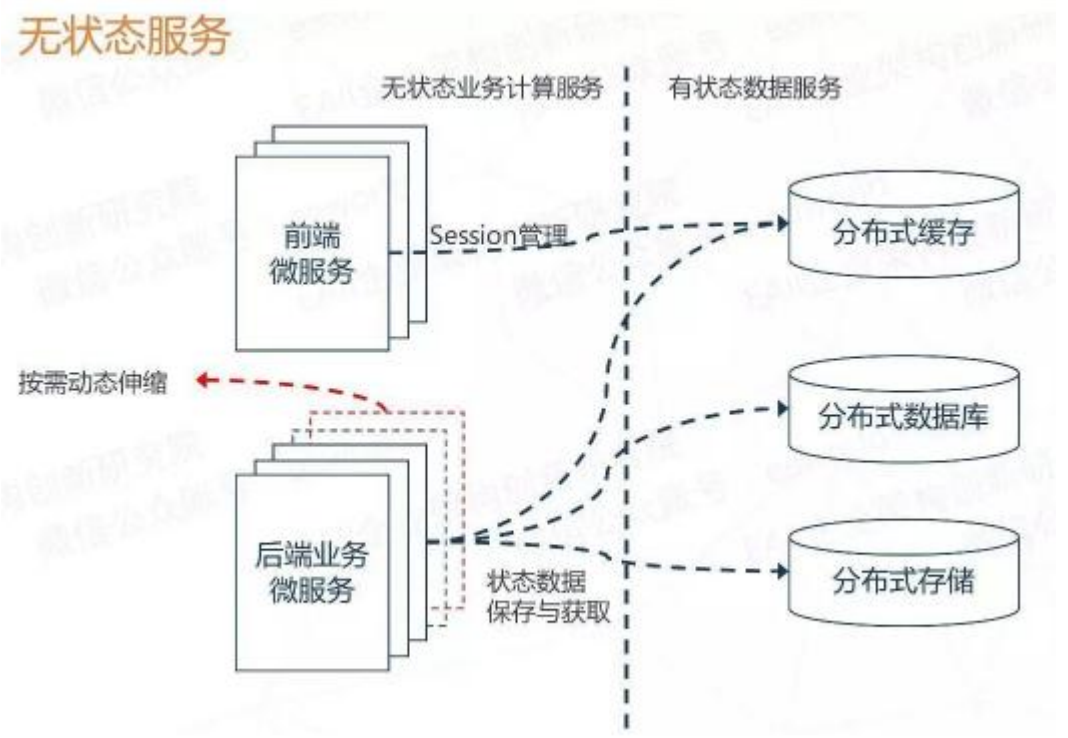
这种分离方式有几个好处：

1) 前后端技术分离，可以由各自的专家来对各自的领域进行优化，这样前段的用户体验优化效果更好。

2) 分离模式下，前后端交互界面更清晰，就剩下了接口模型，后端的接口简洁明了，更容易维护。

3) 前端多渠道集成场景更容易实现，后端服务无需变更，采用统一的数据和模型，可以支持多个前端：例如：微信 h5 前端、PC 前端、安卓前端、IOS 前端。

3 无状态服务

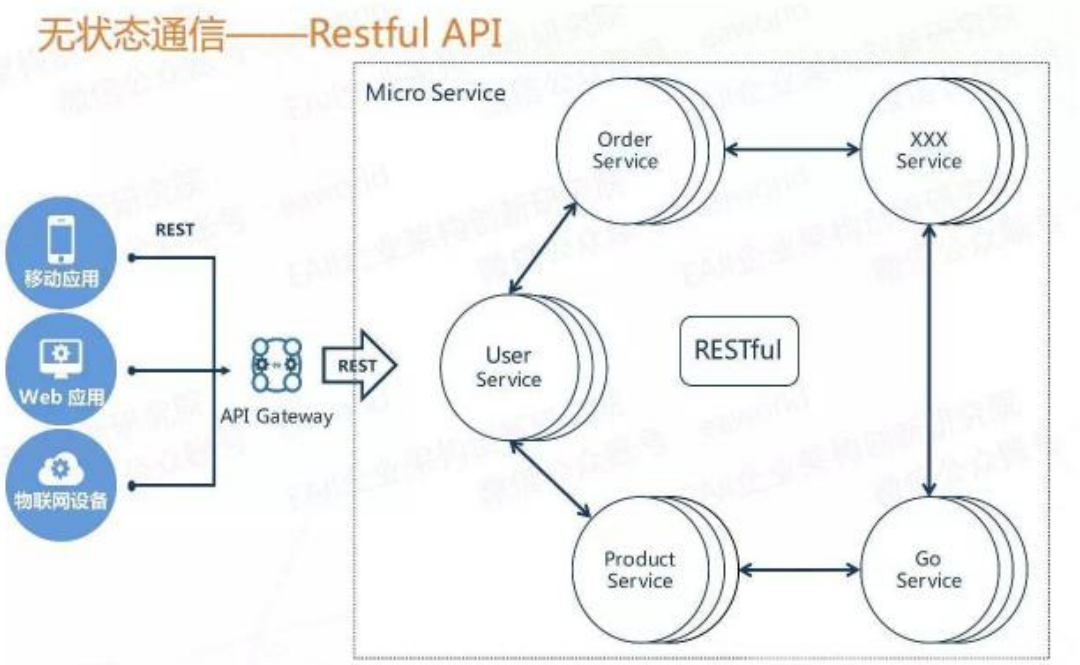


对于无状态服务，首先说一下什么是状态：如果一个数据需要被多个服务共享，才能完成一笔交易，那么这个数据被称为状态。进而依赖这个“状态”数据的服务被称为有状态服务，反之称为无状态服务。

那么这个无状态服务原则并不是说在微服务架构里就不允许存在状态，表达的真实意思是要把有状态的业务服务改变为无状态的计算类服务，那么状态数据也就相应的迁移到对应的“有状态数据服务”中。

场景说明：例如我们以前在本地内存中建立的数据缓存、Session 缓存，到现在的微服务架构中就应该把这些数据迁移到分布式缓存中存储，让业务服务变成一个无状态的计算节点。迁移后，就可以做到按需动态伸缩，微服务应用在运行时动态增删节点，就不再需要考虑缓存数据如何同步的问题。

4 RestFul 的通讯风格



作为一个原则来讲本来应该是个“无状态通信原则”，在这里我们直接推荐一个实践优选的 Restful 通信风格 ，因为他有很多好处：

1）无状态协议 HTTP，具备先天优势，扩展能力很强。例如需要安全加密，有现成的成熟方案 HTTPS 即可。

2）JSON 报文序列化，轻量简单，人与机器均可读，学习成本低，搜索引擎友好。

3) 语言无关，各大热门语言都提供成熟的 Restful API 框架，相对其他的一些 RPC 框架生态更完善。