www.vip.com

以Cloud Native 打造大型电商系统

赵守忠

唯品会
vip.com
一家专门做特卖的网站

简介

个人简介：赵守忠，唯品会云计算高级架构师。有6年云计算相关经验，在IAAS，PAAS，电商系统云化有较为丰富的经验和心得。擅长IAAS的异构计算设备、异构存储设备、异构网络设备、异构虚拟化；擅长PAAS的CI，CD，App生命周期管理，云服务等业务。专注Openstack、Mesos 、Docker等领域。

演讲题目：以Cloud Native打造大型电商系统
演讲内容简介：Cloud Native基本概念介绍、Cloud Native在电商的优秀实践、基于Cloud Native的电商架构等。

大纲

- Cloud Native概念
- Cloud Native+电商优秀实践
- 微服务
- 云服务
- 典型云化电商架构
- 架构演进案例
- Q&A

Cloud Native 概念

议题背景：Cloud Native表面看起来比较容易理解，但是细思好像又有些模糊不清：它用来解决什么问题？它是一个新技术还是一个新的方法？什么样的APP符合“云原生”的呢？等等。

Cloud Native定义： Cloud Native是Matt Stine提出的一个概念，它是一个思想的集合，包括DevOps、持续交付(Continuous Delivery)、微服务(MicroServices)、敏捷基础设施(Agile Infrastructure)、康威定律(Conways Law)等，以及根据商业能力对公司进行重组。

可以说，Cloud Native即包含技术（微服务，敏捷基础设施），也包含管理（DevOps，持续交付，康威定律，重组等）。Cloud Native也可以说是一系列Cloud技术、企业管理方法的集合。

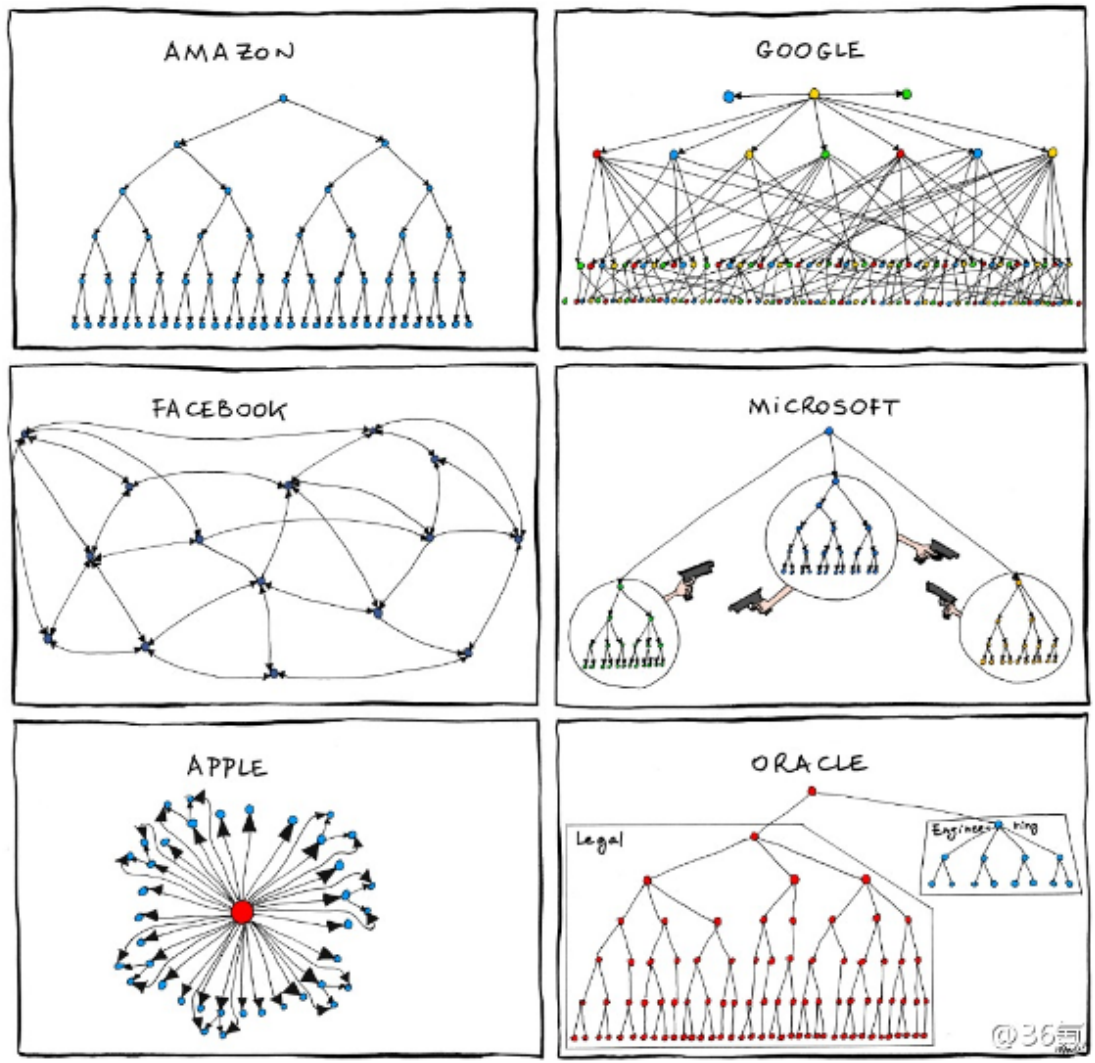
Cloud Native的价值：企业采用基于Cloud Native的技术和管理方法，可以更好的把业务迁移到云平台，从而享受云的高效和按需资源能力。

下面重点讲解Cloud Native的几个常用概念

Cloud Native 概念

Cloud Native的五个层面：

- （1）康威定律：业务云化推行，从某种意义上讲也是一种变革。既然是变革，必然会涉及组织的各个层面，开发、质量、运维等等都会涉及。康威定律则准确的描述了系统架构和组织的关系：组织决定系统架构！
- 一个云系统最终长成什么样子，则完全是企业的组织结构决定的，是组织内部、组织之间的沟通结构。
- 如果要想得到一个合理的Cloud架构，仅从技术入手是不够的，还需要从组织架构入手，才真正有效。

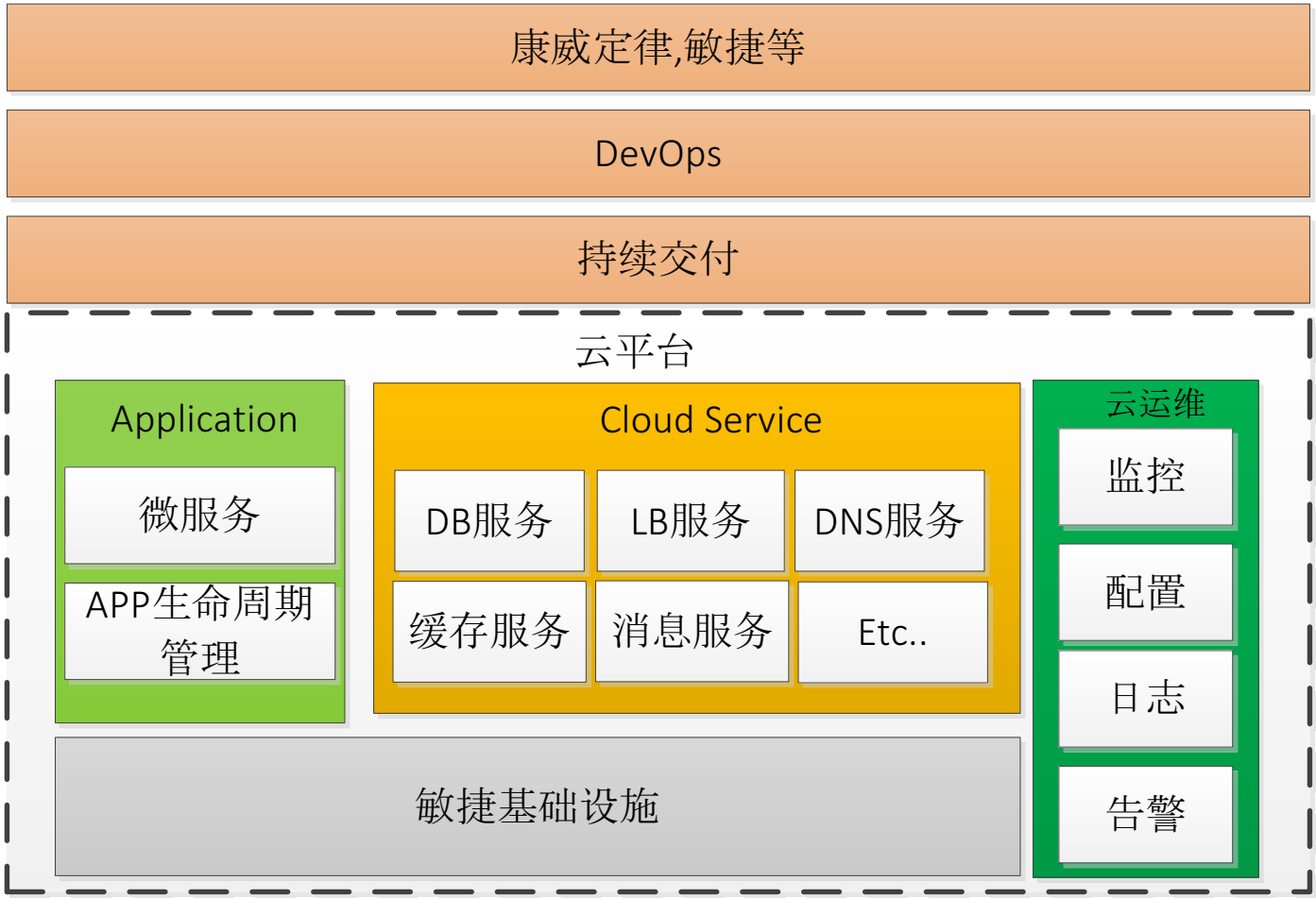


Cloud Native 概念

- （2）DevOps：（英文Development和Operations的组合）是一组过程、方法与系统的统称，用于促进开发（应用程序/软件工程）、运维和质量保障（QA）部门之间的沟通、协作与整合。它的出现是由于软件行业日益清晰地认识到：为了按时交付软件产品和服务，开发和运维必须紧密合作。
- （3）持续交付(Continuous Delivery)：是一系列的开发实践方法，用来确保让代码能够快速、安全的部署到产品环境中，它通过将每一次改动都提交到一个模拟产品环境中，使用严格的自动化测试，确保业务应用和服务能符合预期。因为使用完全的自动化过程来把每个变更自动的提交到测试环境中，所以当业务开发完成时，你有信心只需要按一次按钮就能将应用安全的部署到产品环境中。持续交付可以采用：CI(持续集成)、代码检查、UT(单元测试)，持续部署等方式，打通开发、测试、生产的各个环节，持续的增量的交付产品。
- （4）微服务(MicroServices)：微服务首先是一个服务，其次微服务具备如下特点：颗粒比较少；采用轻量级通信框架；可以独立部署；去中心化；有明确的业务边界等。
- 微服务可以采用Docker、LXC等技术手段实现。
- （5）敏捷基础设施(Agile Infrastructure)：提供弹性、按需的计算、存储、网络资源能力。可以通过Openstack、KVM、Ceph、OVS等技术手段实现。

Cloud Native+电商优秀实践

- Cloud Natvie是一系列技术和管理实践的集合，它的具体范围，业界是没有明确定义的，这给我们留下不少扩展空间：即Cloud Native可以和不同的行业结合，实践出不同的优秀实践。在电商这个特定领域，又有哪些优秀实践呢？下面这个图给予一个示例。（备注：这个图仅是个人思考所得，非标准和官方定义。）

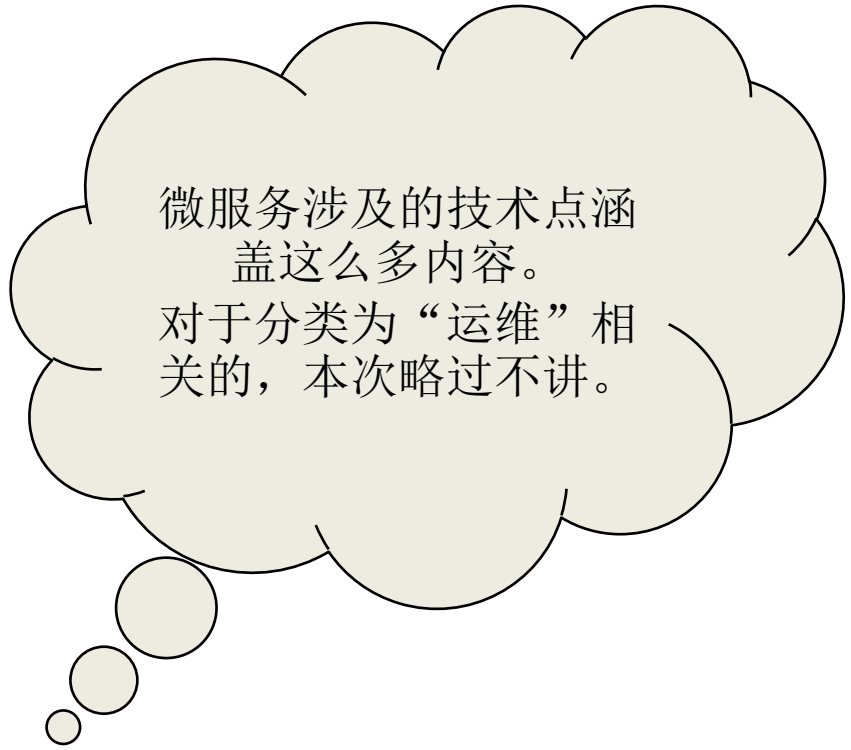


Cloud Native+电商优秀实践

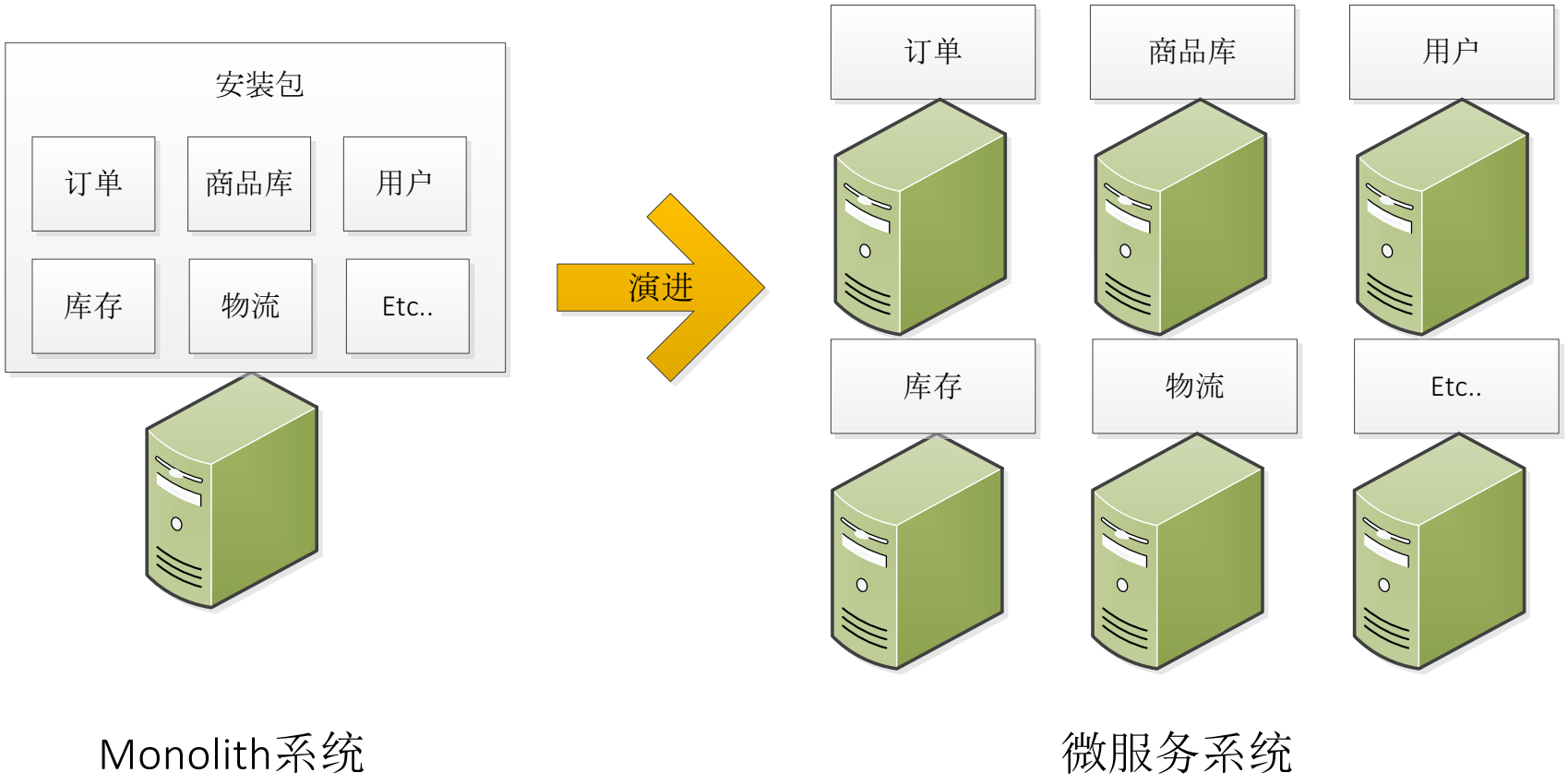
- 敏捷基础设施：
  - 这部分是整个Cloud Native的基础，对应于IAAS(Infrastructure as a Service)部分。备注：这部分内容比较多，本次不重点讲。
- Application/Cloud Service
  - Application：
    - APP生命周期管理：实现对应用的生命周期管理，包括应用的编译、打包、UT、部署、升级、伸缩、状态监控、销毁等多个活动。
    - 微服务：微服务主要是针对应用的。微服务涉及应用的架构设计、部署框架、高并发、部署、升级、监控等各个环节。本次重点来讲解。
  - 云服务：
    - 涉及DB，缓存，消息、DNS、LB等多个服务。这些服务在支撑应用面对大容量，高并发方面发挥着重要作用。本次重点讲解。
  - Application和云服务对应于PAAS(Platform as a Service)部分。
- 云运维：
  - 包括日志、告警、监控、配置等内容。备注：这部分内容比较多，本次不重点讲。
- 管理相关：
  - 涉及康威定律，敏捷，DevOps，持续交付等内容。这些都属于管理层面的优秀实践，当然这些优秀实践底层还是需要Cloud技术支撑的。但是更强调在组织层面如何推动Cloud Native。备注：这部分内容比较多，本次不重点讲。

微服务

分类	技术点名称
概念	Monolith和微服务
	SOA和微服务
架构	微服务如何对接老系统
	微服务注册和发现
	微服务路由/LB
	微服务流控
	微服务隔离
	微服务配置中心
	API网关/边缘服务
规范	12因子
运维	微服务部署
	微服务升级
	微服务监控
	微服务日志
	微服务依赖
	微服务伸缩
	微服务高可用性



Monolith和微服务



Monolith系统

微服务系统

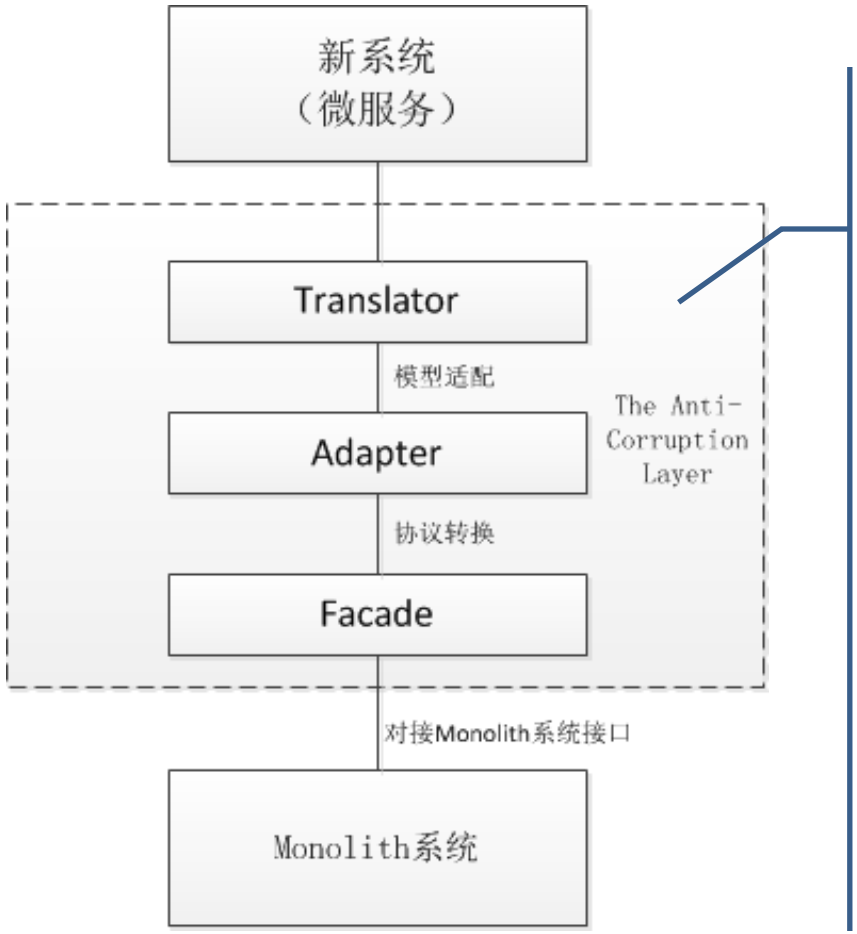
Monolith系统中，各组件不能独立运行，必须一起打包发布。扩容时，需要整体扩容。
微服务系统中，各组件可以独立发布，独立运行。扩容时，可以根据需要单个组件扩容。

SOA和微服务

- You should instead think of microservices as a specific approach for SOA in the same way that XP or Scrum are specific approaches for Agile software development.----来自《Building Microservices》
- 这是我目前认为描述他们关系最为贴切的一句话。
- 传统的SOA，一个关键点是ESB（企业服务总线）。但是ESB在很多企业应用并不成功，这也搞坏了SOA的名声。
- SOA面向服务的架构设计思想是没有问题的，微服务也符合这种思想。但是相对于传统概念的SOA，微服务更强调“微”，以“微”概念试图和传统SOA做区分。

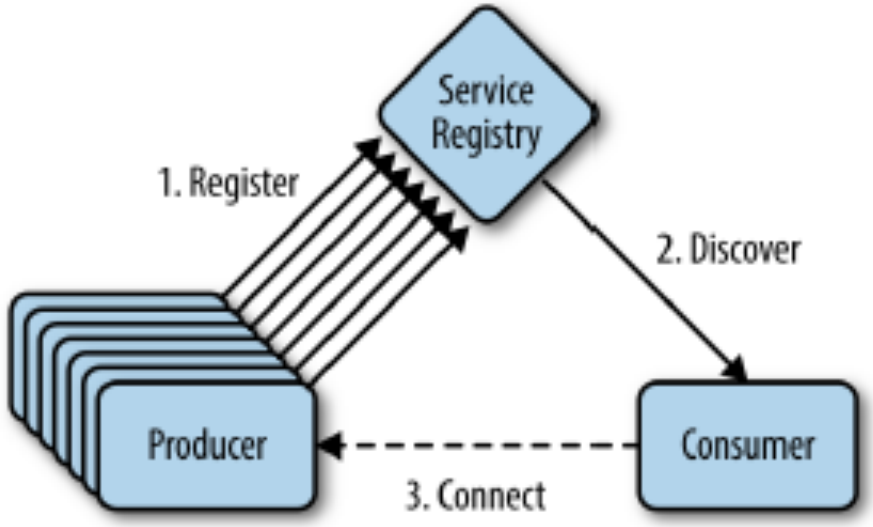
微服务如何对接老系统

- 直接对原有系统进行微服务改造，是比较困难的，几乎是不现实的。比较合理的方法是新系统采用微服务开发，对老系统进行服务封装，系统架构如下所示：



Monolith系统：对应企业老的系统。
The Anti-Corruption Layer：反腐层，这层完成对老系统的桥接，并阻止老系统的腐烂蔓延。它包含三部分：
（1）Facade：简化对老系统接口的对接。
（2）Adapter：Request，Response 请求协议适配
（3）Translator：领域模型适配，转换微服务模型和老系统模型。
新系统（微服务）：新系统开发采用微服务架构。

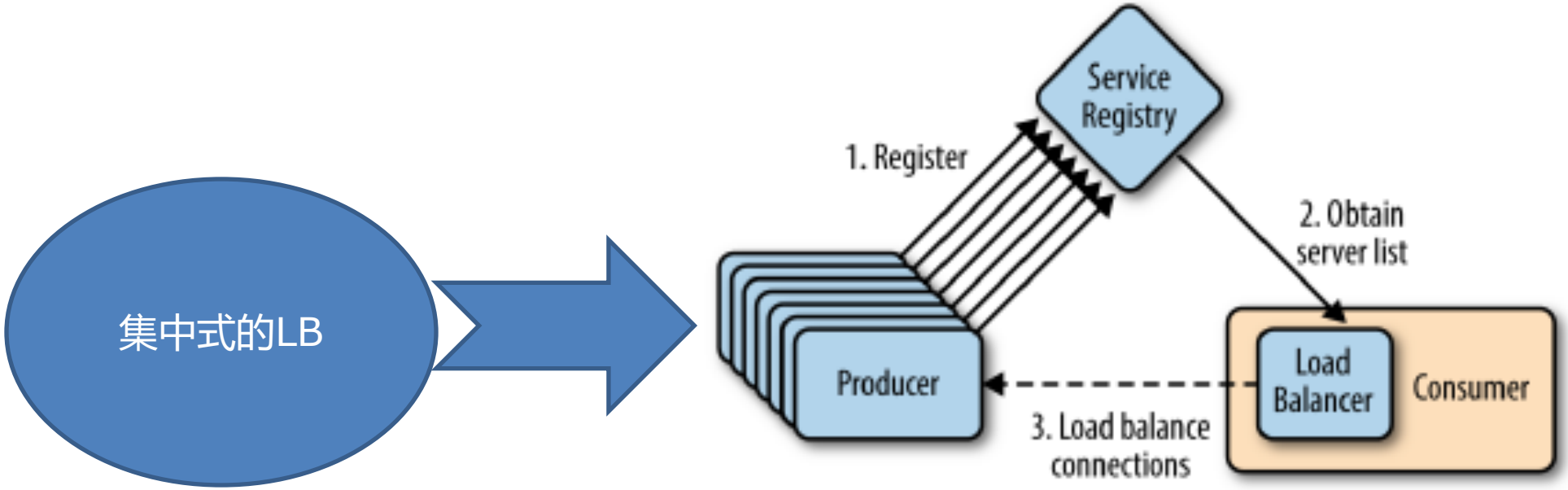
微服务注册和发现



这个是传统的服务注册和发现架构图。服务注册方式，常见的包括DNS，基于zookeeper的服务注册方案等等。

备注：Consumer和Producer之间一般还有个LB

微服务路由和LB



集中的LB一旦垮掉，将会影响整个系统。

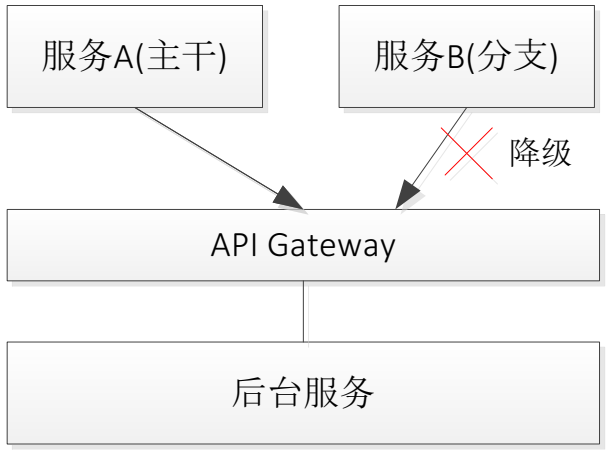
分布式LB

每个Client端有单独的LB服务（可以内置代码库，也可以单独进程）。

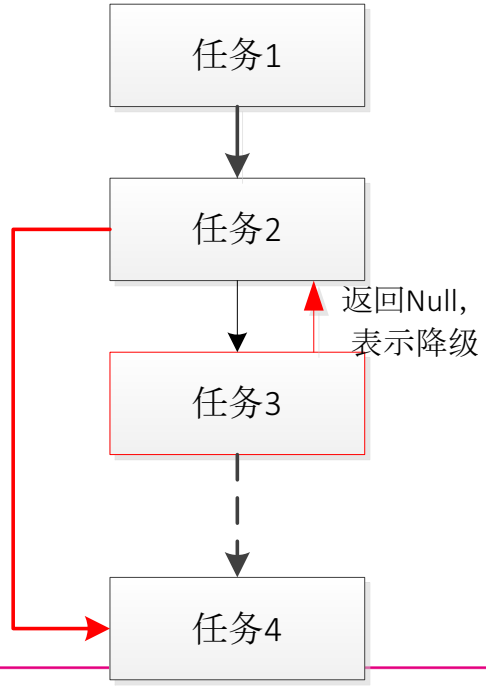
Client从服务注册中心拿到service 列表后，根据一定的路由算法，直接和服务通信。

微服务流控

- 微服务流控包含如下几点
  - 限流：每个系统都有最大的处理能力，一旦请求超过了系统最大处理能力，必须进行限流，否则会压垮整个系统。限流方法包括：
    - 设定最大访问请求速率，对于超出的部分则丢弃。
    - 按比例丢弃请求，比如一次性随机丢失20%的请求。
  - 降级：服务降级指人为的关闭一些不重要服务，或者把一些不重要的处理步骤旁路。降级的目的是防止非核心服务和流程占用的带宽和资源影响核心服务/主流程，从而保证核心服务，核心流程有充足的带宽和资源。
  - 熔断：流量过大时，会导致某些服务不可用。这时必须进行服务熔断，确保主流程可以继续进行。

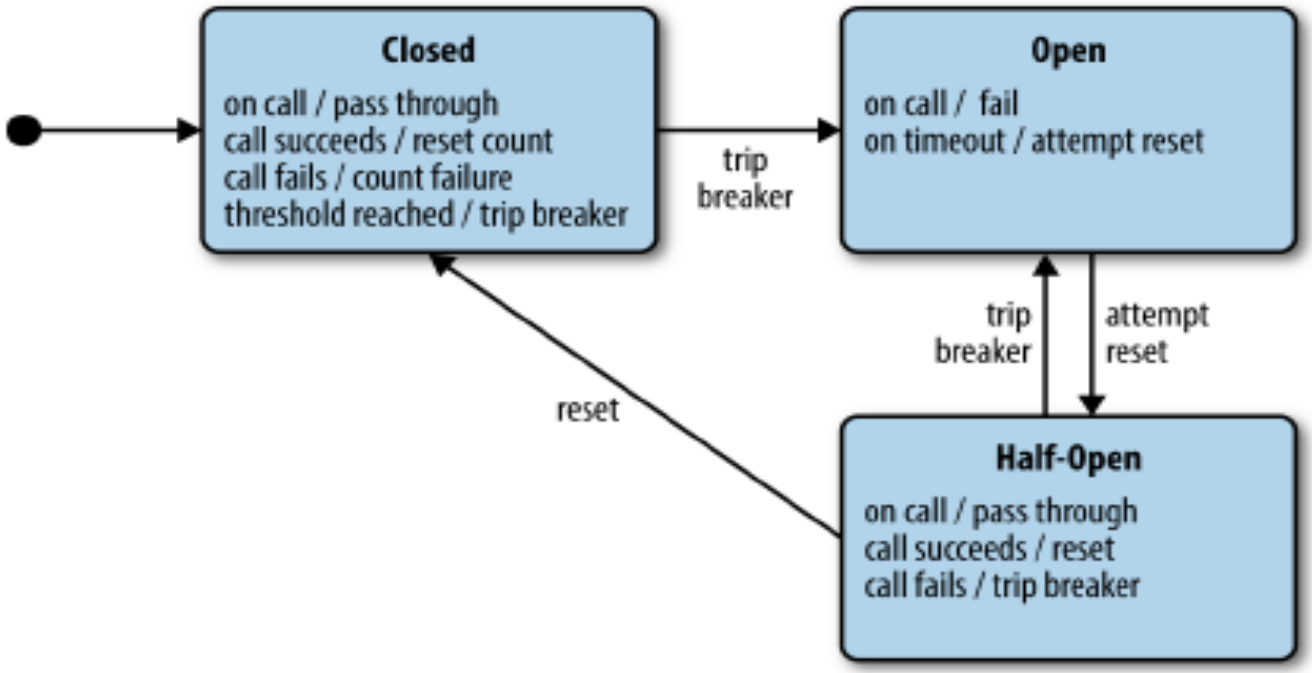


服务B在高并发时，有可能和服务A抢占资源，导致服务A失败。



任务3属于一个大流程的非关键步奏，但是有一定的资源消耗。高并发时可以关闭，确保主流程正常运行。

微服务流控-熔断



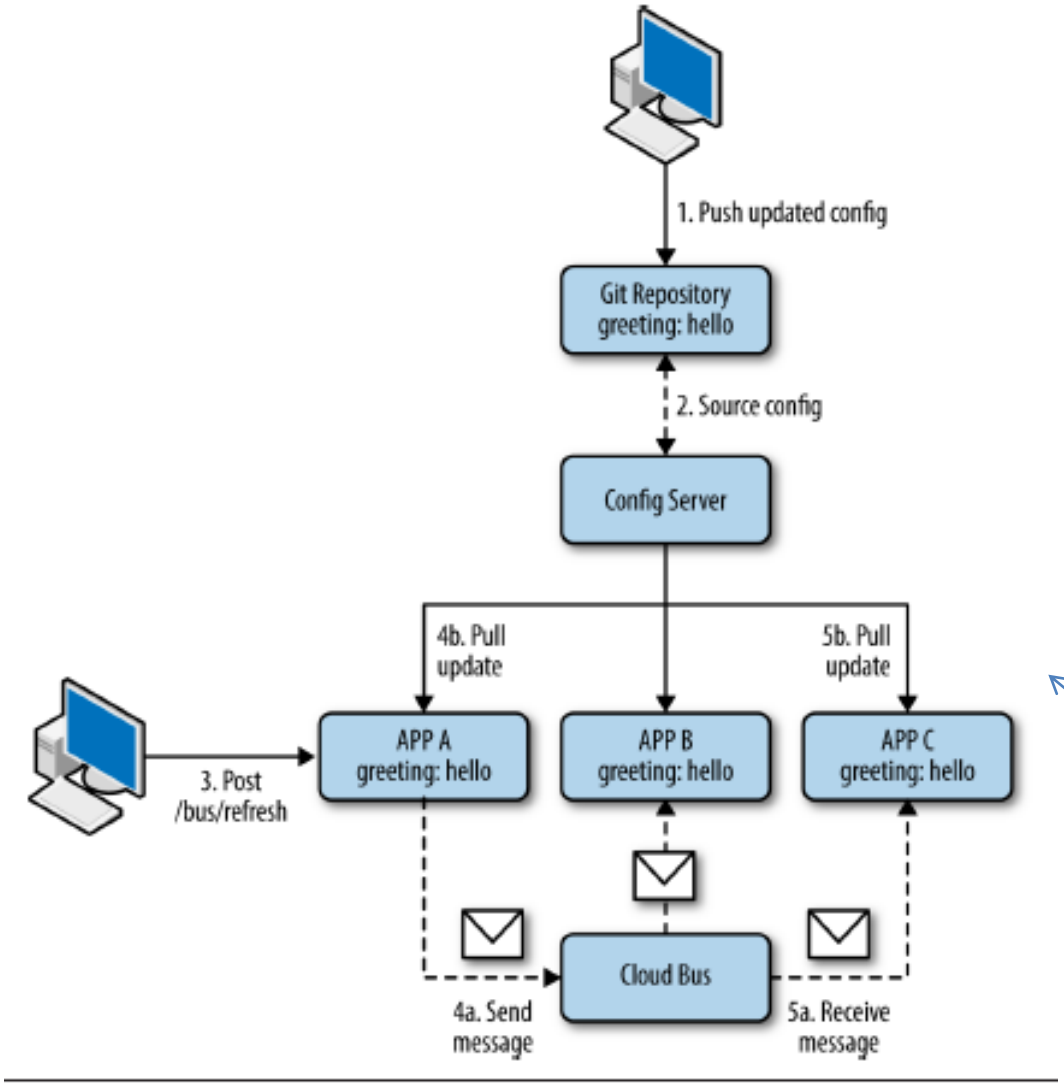
电路熔断器（Circuit Breaker）：该模式的原理类似于电路熔断器，如果电路发生短路，熔断器能够主动熔断电路，以避免灾难性损失

（1）正常状态下，电路处于关闭状态(Closed)，调用是直接传递给依赖服务的；
（2）如果调用出错，则进入失败计数状态；
（3）失败计数达到一定阈值后，进入熔断状态(Open)，这时的调用总是返回失败；
（4）累计一段时间以后，保护器会尝试进入半熔断状态(Half-Open)
（5）处于Harf-Open状态时，调用先被传递给依赖的服务，如果成功，则重置电路状态为“Closed”，否则把电路状态置为“Open”

微服务隔离

- 舱壁(Bulkheads)模式:
  - 该模式像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响 。
- 这种模式比较常见的思路为：
  - A：采用微服务是首选，比如docker。Docker是进程隔离的，单个docker失效不会影响其他docker容器。
  - B：把大的并行处理工作，由多个线程池来负荷分担。

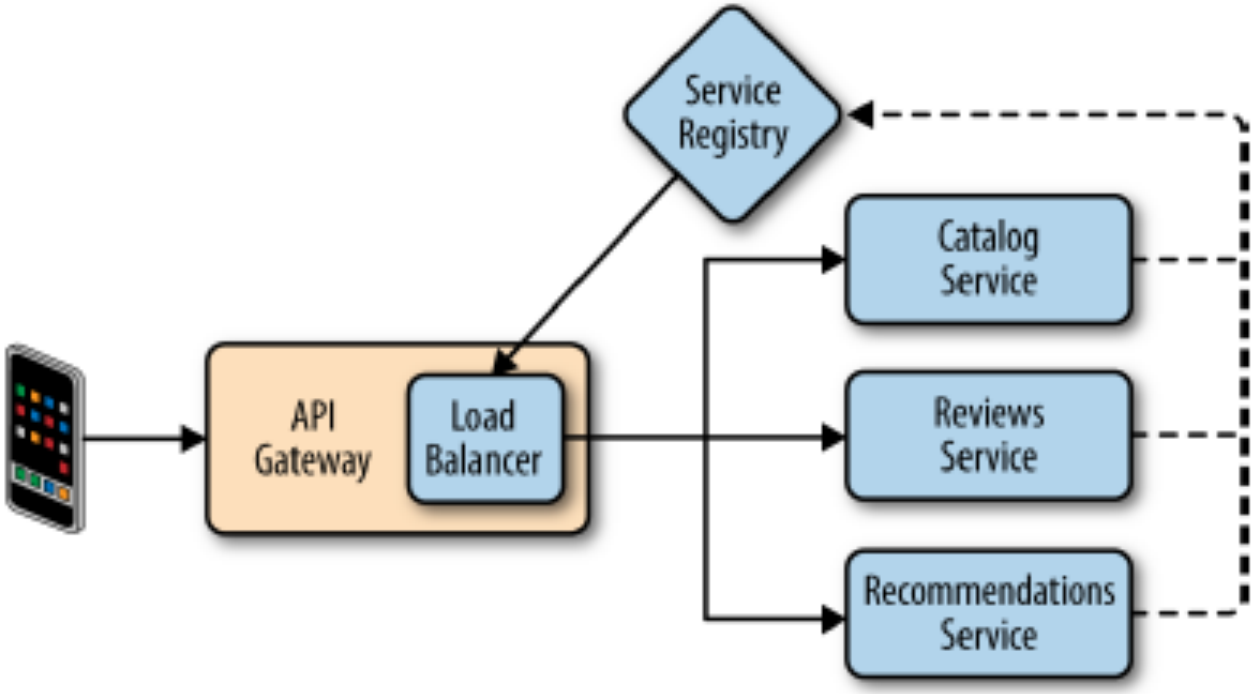
微服务配置中心



- 微服务的配置是和具体环境相关的，开发测试环境，staging环境，线上环境的配置是不同。这些不同的配置需要统一纳入配置中心。
- 配置中心需要满足分布式能力
- 配置中心需要支持版本化管理：支持配置信息版本化控制，可审计，安全，配置更新不需要重启。
- 这部分可以参考Spring Cloud Config Server，Spring Cloud Bus。

1：用户提交配置更新
2：配置server更新
3：对APP A发起Refresh更新配置操作
4a：发送消息到Bus
4b：Pull 更新的配置
5a：APP C接收消息
5b：Pull更新的配置
6：其他节点同步更新

API网关/边缘服务



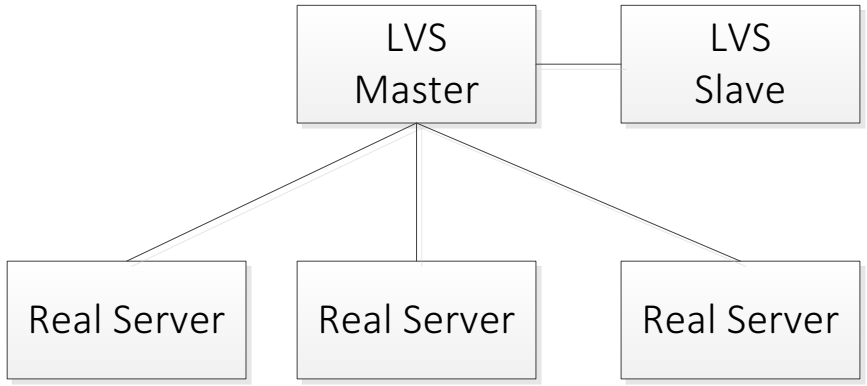
API Gateway:
(1)对设备侧(PC,Mobile等)提供简化的单一服务接口；
(2)它内部聚合后台几十甚至上百微服务；
(3)必要时进行协议转换(比如Http转为AMQP)
价值：它的主要作用是简化设备侧开发的复杂度，减少微服务网络调用数量和网络延迟问题。

12因子(Twelve-Factor)

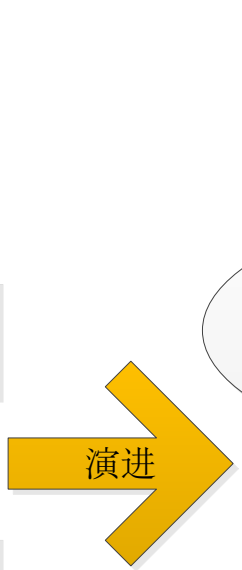
- 12因子(Twelve-Factor)：
  - 基准代码(Codebase)：代码必须纳入配置库统一管理
  - 依赖(Dependencies)：显式的声明对其他服务的依赖，比如通过Maven, Bundler, NPM等。
  - 配置(Config)：对于不同环境（开发/staging/生产等）的参数配置，是通过环境变量的方式进行注入。
  - 后台服务(Backing services)：对于DB，缓存等后台服务，是作为附加资源，可以独立的Bind/Unbind。
  - 编译/发布/运行(Build, Release, Run)：Build，Release，Run这三个阶段要清晰的定义和分开。
  - 无状态进程(Processes)：App的进程是无状态的，任何状态信息都存储到Backing services（DB，缓存等）。
  - 端口绑定(Port binding)：App是自包含的，所有对外服务通过Port Binding暴露，比如通过Http。
  - 并发(Concurrency)：App可以水平的Scaling。
  - 快速启动终止 (Disposability)：App进程可以被安全的、快速的关闭和重启。
  - 环境一致性(Dev/prod parity):尽可能的保持开发、staging，线上环境的一致性。
  - 日志(Logs)：把日志作为事件流，不管理日志文件，通过一个集中的服务，由执行环境去收集、聚合、索引、分析日志事件。
  - 管理进程(Admin processes):在相同环境下，后台管理任务作为一次性进程运行.

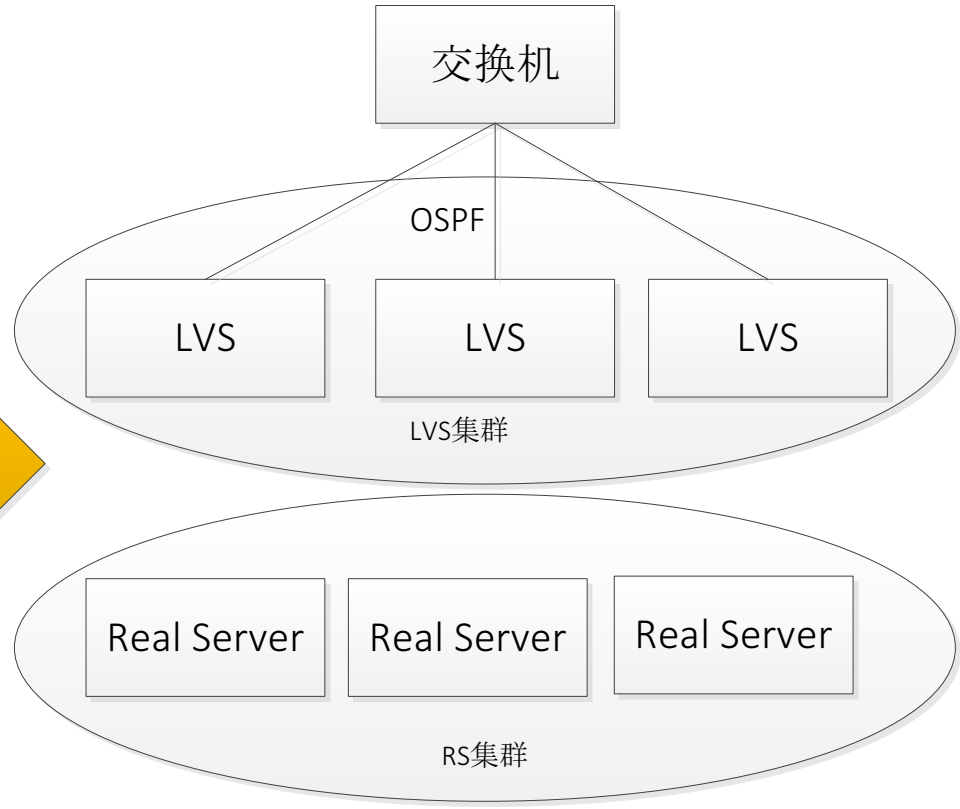
云服务--LB

- 负载均衡可以由硬件提供，比如F5；也可以采用软件方式，比如LVS。我们就以LVS举例说明。



LVS采用主备模式，可以满足大部分的业务场景。

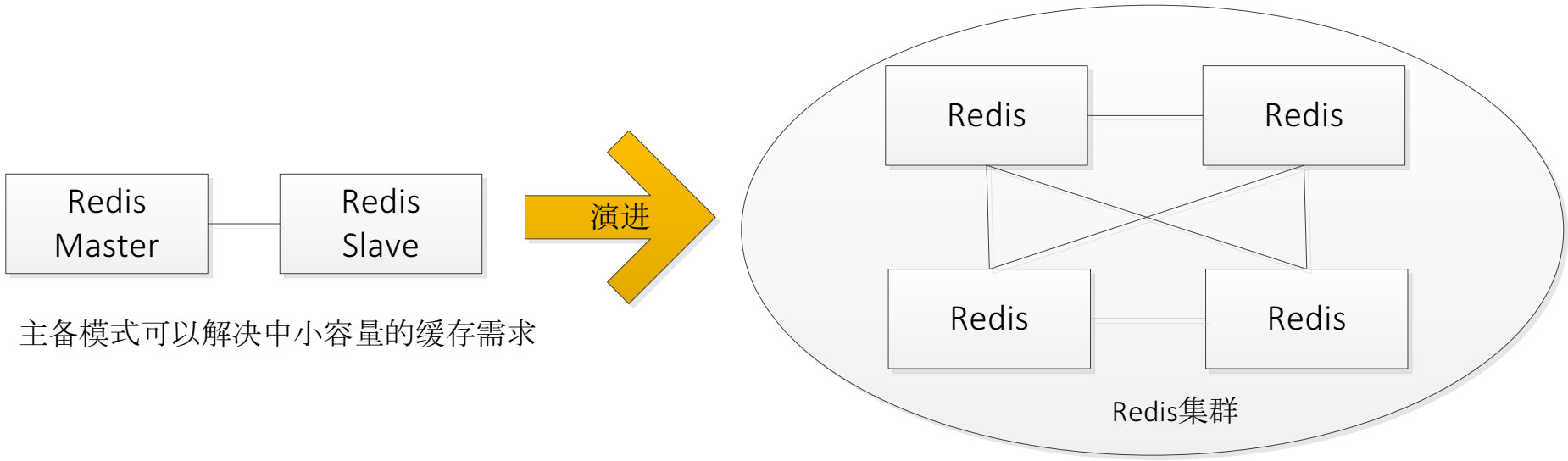




LVS采用集群模式，适合于流量非常大的场景。

云服务--缓存

- 缓存用的比较多的是Redis和Memcached。以Redis为例。

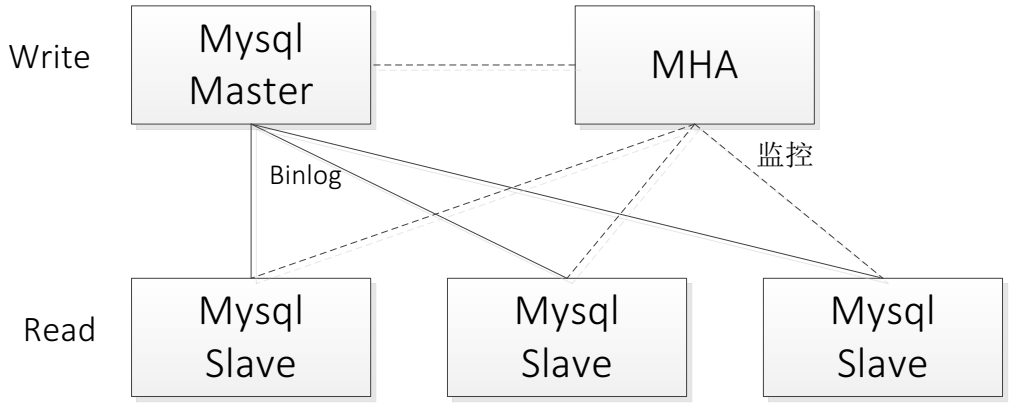


主备模式可以解决中小容量的缓存需求

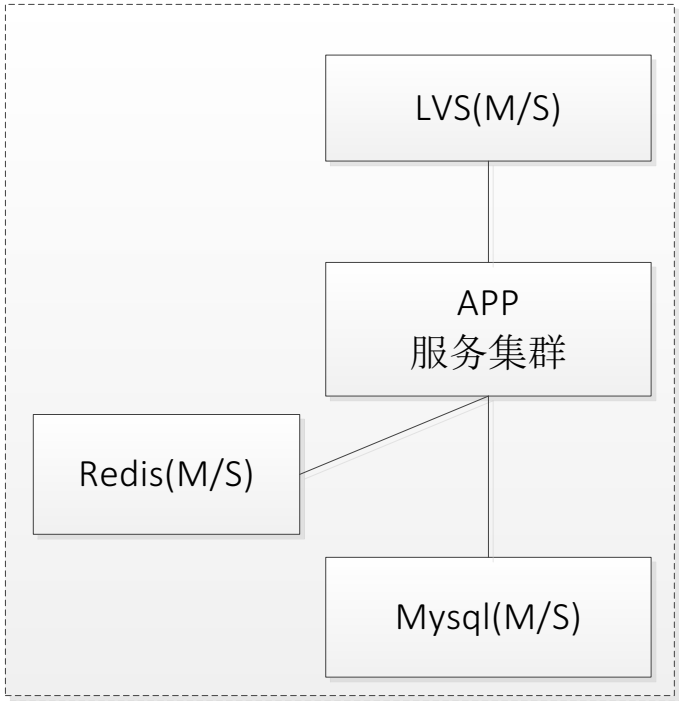
集群模式可以支持更大的缓存需求

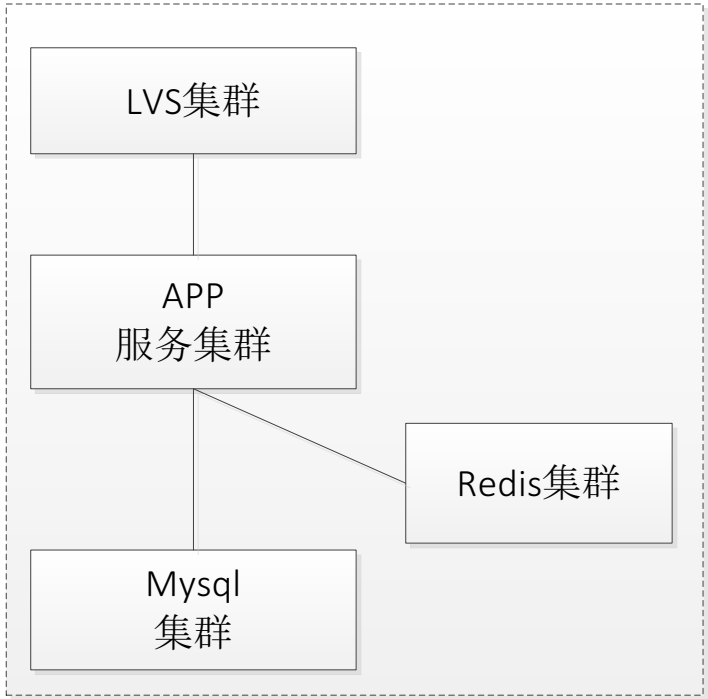
云服务--DB

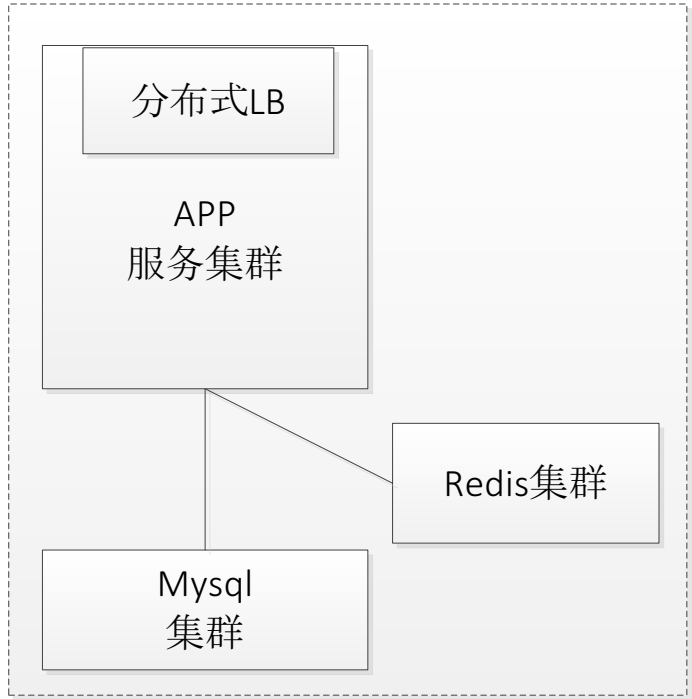
- 数据库部分重点讲Mysql。
- 随着容量的增大，DB部分采取的方案不外乎：
  - 拆库：把一个数据库拆分为多个
  - 拆表：如果单个表的数量太大，则把该表拆分为多个表。
  - 读写分类：主库完成写操作，从库完成读操作。
  - 如上处理完成，形成大的DB集群



典型云化电商架构

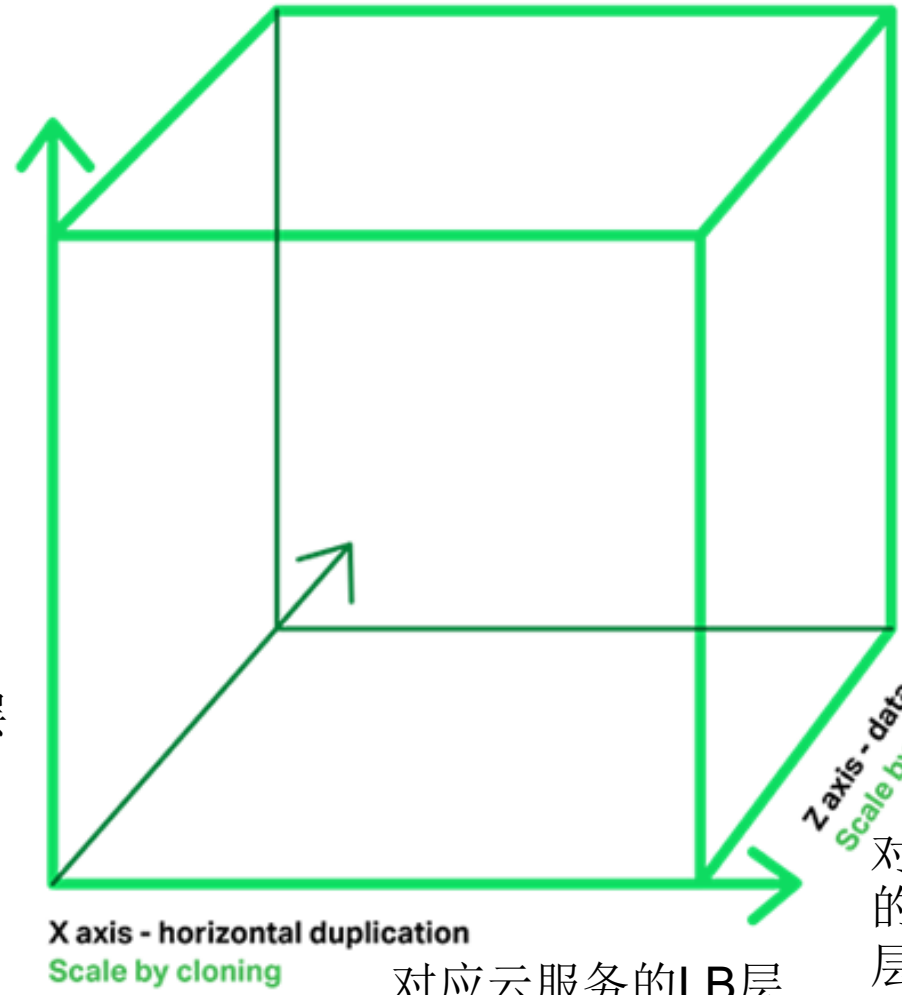






- 从架构上来看，主要有四层：LB层，APP服务层(微服务)，缓存层，DB层，存在三种典型模式：
  - 主从模式
  - 集群模式
  - 分布式模式
- 当然，很多网站架构处于两者之间混合的状态。
- 如何演进到云化架构呢，后面内容会给个实例

架构演进案例--系统三维扩展模型



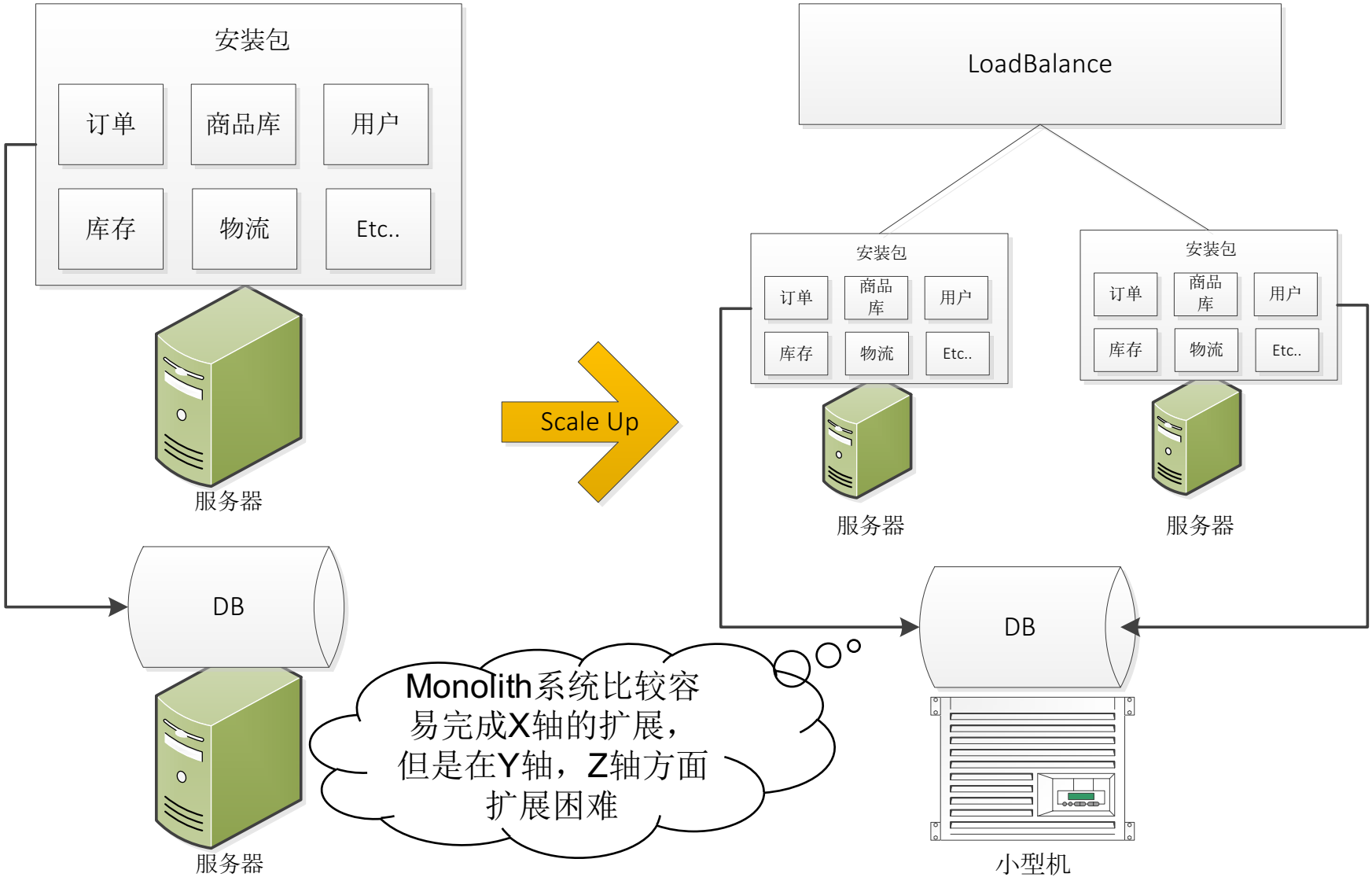
对应微服务层

对应云服务的LB层

对应云服务的缓存/DB层

这个是经典的三维扩展模型：
X轴通过LB进行扩展，对应云服务的LB部分；
Y轴通过功能的解耦和拆分完成，对应微服务部分；
Z轴完成数据的拆分，对应云服务的缓存，DB部分。
云平台需要支撑这三维的扩展能力，从而可以支撑大容量，高并发的业务场景。

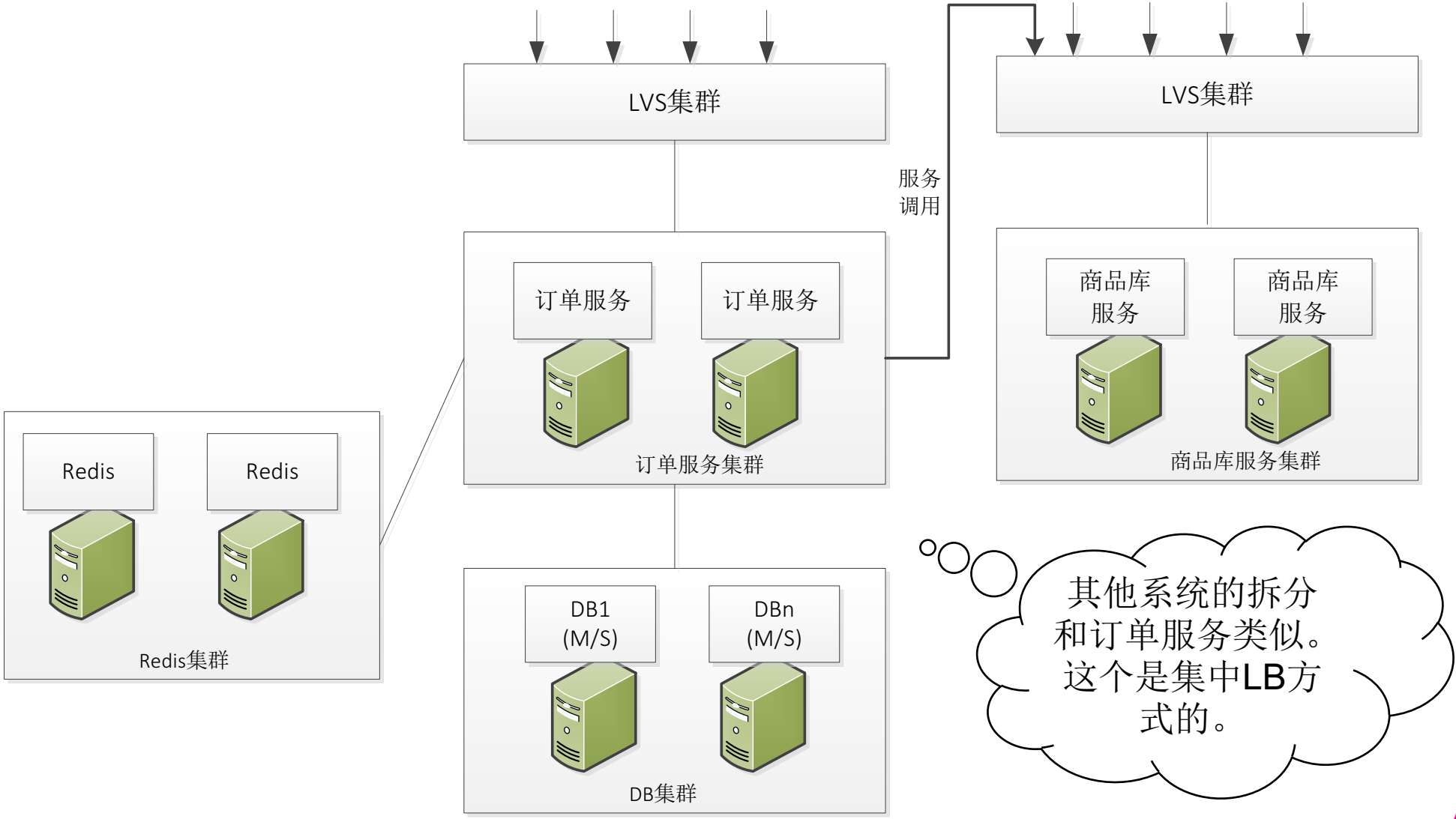
架构演进案例--Monolith系统演进



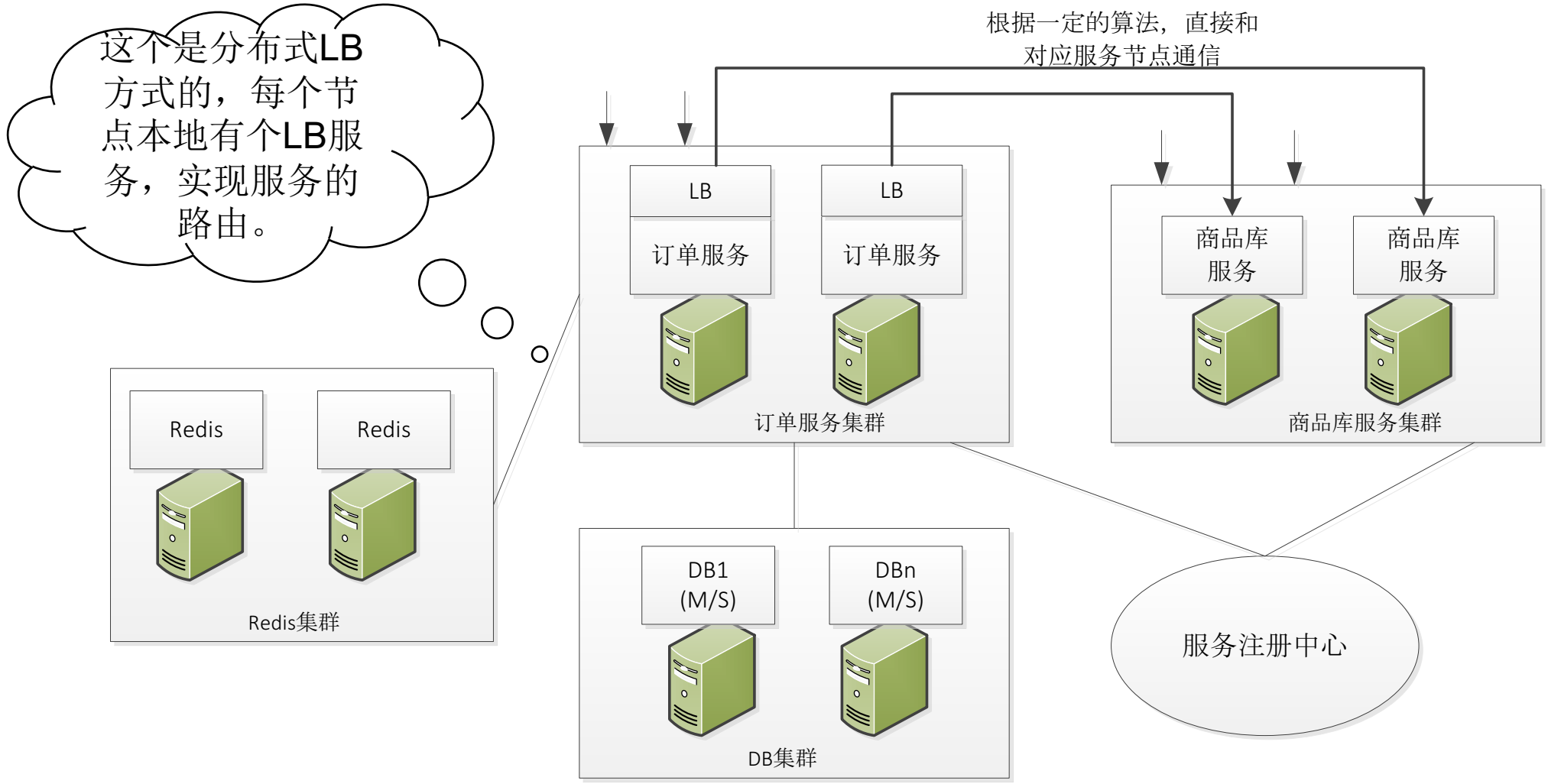
架构演进案例--微服务演进

- 随着用户并发量的增加，需要进一步从X轴，Y轴，Z轴进一步拆分。
  - Y轴拆分：以业务为单位进行功能拆分，比如电商系统可以拆分为订单服务、用户服务、库存服务等。每个服务可以根据需要在进一步拆分为更细力度的服务。一般的大型网站可以拆分出几千个这样的服务。
  - X轴拆分：单一的集中的LB必然无法支撑，会拆分出多个LB。比如一个服务或者多个服务对应一个LB（或者采用无中心的分布式LB方案）。
  - Z轴拆分：Z轴拆分也是根据Y轴功能的拆分而对应拆分，比如一个服务对应一个或者多个DB/缓存集群。

架构演进案例--微服务演进



架构演进案例--微服务演进



总结

- 总结
  - 前面讲解了Cloud Native，微服务，云服务等，讲解三维系统扩展模型。
  - 这些内容都是构建在云平台这个能力基础上，如果没有云平台这个基础能力支撑，是没有办法愉快的玩耍的。
  - 云平台
    - 对接物理机、网络、存储，弹性提供虚拟机、docker、云盘等弹性资源。
    - 支持对APP生命周期管理
    - 提供微服务支撑平台
    - 提供云服务能力（对云服务接入、部署、配置、监控等实现自动化）
    - 提供云运维能力，实现对APP，云服务的统一日志、监控、告警、配置等。
    - 对持续交付提供技术支撑和框架支持，实现高效的版本发布和管理。
  - 有了这些基础能力的支撑，才能支撑高并发、大容量的电商系统。
  - 当然，除了技术平台支撑外，还需要有Cloud化的管理理念，包括康威定律，Devops，持续发布等。如果不能在组织和管理层面上对Cloud Native提供支持，是无法真正推行Cloud Native的。

后记

准备时间有限，难免有遗漏。欢迎大家指正。

QQ：2834646170

微信号：zhaoshouzhong_shx

谢谢

Thank you

www.VIP.com