微服务

- 定义
  - 将大的系统分解，构建逻辑上独立的小系统。每个微服务都有自己的业务逻辑和数据库，对一个微服务的修改不会影响其他微服务。微服务间通信使用轻量级的协议，如：http, REST,消息协议。
- 原则
  - 功能完整性，职责单一性原则
  - 围绕业务构建微服务。从技术上来说，微服务不能局限于某个技术栈或者后端存储，可以非常灵活，以便于解决业务问题
  - 谁构建，谁负责
    - 让开发参与到软件的运维中，更好的理解自己的软件是如何被使用的。
  - 基础设施的自动化
    - 微服务必须是独立可部署的，应该可以打包所有依赖，甚至运行环境
  - 设计考虑失败
    - 开发前就要考虑好，微服务失败对于整个用户体验的影响
    - 能够快速检测到失败，如果可能，自动恢复
    - 微服务对实时监控投入更多的注意力，检测系统指标和业务指标
  - 迭代演进，并非一蹴而就
    - 当演进到一定阶段，对微服务的开发，部署，测试，运维等成本都比较低时，就是一个好的微服务
    - 模块分解是否合理的一个关键指标是：是否模块可以独立替换或者更新，而不影响使用者
  - API版本兼容性优先考虑
- 好处
  - 为每个微服务选择最适合的架构和技术，更新架构和技术也更容易
  - 降低企业创新的成本
  - 可以有选择的做扩展

- 微服务是个有机体，可以自由增加功能，最小化对已有服务的影响
- 微服务打包了运行环境，使得多版本并存成为可能
- 围绕微服务构建小的，专注的，敏捷开发团队

- 实践
  - 对业务进行拆分
    - 横向拆分。按照不同的业务域进行拆分，例如：订单，商品，库存
    - 纵向拆分。把一个业务功能中的不同模块与组件进行拆分。例如：把公共组件拆分成原子服务，下沉到底层，形成独立的原子服务层。
  - 做好微服务的分层
    - 梳理和抽取核心应用，公共应用，作为独立的服务下沉到核心和公共能力层，逐渐形成稳定的服务中心，使得前端应用更快的响应市场需求
  - 服务的自动注册和发现
  - 接口先行，语言中立
- 治理策略
  - 服务的注册和发现
    - 解决的问题
      - 服务越来越多，服务URL配置管理越来越复杂
  - 软复杂均衡和容错
    - 解决的问题
      - 硬件负载均衡器的压力越来越大
  - 服务的依赖关系
    - 解决的问题
      - **服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系**
    - 解决方法
      - 对应用分层，比如：核心数据层，业务应用层，不允许低层向高层依赖，以免出现循环依赖
      - 在注册中心定义架构体系，列明有哪些层的定义，每个服务暴露和引用时，都必须声明自己属于哪一层，这样注册中心更容易发现架构的腐化现象
  - 服务的负责人和文档
    - 解决的问题
      - 服务多了，沟通成本上升，调某个服务失败找谁，服务的参数有什么约定

- 安全问题
  - 解决的问题
    - 慢慢地一些敏感数据也开始服务化了，安全问题开始变的重要，谁能调用服务，怎么授权
  - 解决方法
    - 服务需要一个密码，访问带着密码。服务提供者生成令牌，将令牌告诉注册中心，由注册中心决定是否告知消费者。这样就能在注册中心做比较复杂的授权功能。
- 流量控制
  - 解决的问题
    - 避免一个消费者将整个服务给拖垮
  - 解决方法
    - 服务提供者监测到流量超标时，拒绝部分请求，进行自我保护
    - 消费者上线前跟提供者约定SLA，消费者承诺每天的调用量，请求的数据量，提供者承诺响应时间，出错率，将SLA记录在监控中心，定时与监控数据做对比，超标则报警
- 路由
  - 解决的问题
    - 保证重要应用的服务质量
  - 解决方法
    - 应用分离
      - consumer.application = foo => provider.host = 1,2,3 consumer.application != foo => provider.host = 5,6
    - 读写分离
      - method.name = find*,get* => provider.host = 1,2,3 method.name != find*,get* => provider.host = 5,6
- 自动测试
  - 解决问题
    - **服务上线后，需要验证服务是否可用，但因防火墙的限制，线下是不能访问线上服务的，不得不先写好一个测试Main，然后放到线上去执行，非常麻烦，并且容易忘记验证。**
  - 解决方法

- 所以线上需要有一个自动运行的验证程序，用户只需在界面上填上要验证的服务方法，以及参数值和期望的返回值，当有一个服务提供者上线时，将自动运行该用例，并将运行结果发邮件通知负责人。

- 服务路由
  - 本地短路策略。优先调用JVM内服务提供者，其次是同一台主机上的，最后是跨网络调用
- 上线审批，下线通知
  - 解决的问题
    - 因为暴露服务很简单，服务的上线越来越随意，有时候负责服务化的架构师都不知道有人上线了某个服务，使得线上服务鱼龙混杂，甚至出现重复的服务，而服务下线比上线还困难。
  - 解决方法
    - 需要一个新服务上线审批流程，必须经过服务化的架构师审批过了，才可以上线。 而服务下线时，应先标识为过时，然后通知调用方尽快修改调用，直到没有人调此服务，才能下线。
- 兼容性
  - 解决问题
    - 因服务接口设计的经验一直在慢慢的积累过程中，很多接口并不能一促而蹴，在修改的过程中，如何保证兼容性，怎么判断是否兼容？另外，更深层次的，业务行为兼容吗？
  - 解决方法
    - 可以根据使用的协议类型，分析接口及领域模型的变更是否兼容，比如：对比加减字段，方法签名等。
      而业务上，可能需要基于自动回归测试用例，形成Technology Compatibility Kit (TCK)，确保兼容升级。
- 容错和降级
  - 解决问题
    - 随着服务的不停升级，总有些意想不到的事情产生，比如Cache写错了导致内存溢出，每次核心服务一挂，影响一大片，人心惶惶
  - 解决方法
    - 应用间声明依赖强度，哪些功能是强依赖，哪些功能是弱依赖，基于依赖强度计算影响面，定期测试复查，加强关键路径上服务的优化和容错，清理不该在关键路径上的服务
    - 提供容错的Mock数据，Mock数据可以在注册中心动态下发，当服务不可用时，使用Mock数据
    - 前端页面的Portal也应该应用降级，当Portal获取不到数据时，直接隐藏，或者替换成其他模块展示，并提供开关，可人工干预是否显示
- 服务的编排

    - 解决问题
        - 当已经有很多小服务，可能需要组合多个小服务的大服务，为此不得不包装一个新服务，新服务业务逻辑少，却带来很大的开发工作量
    - 解决方法
        - 需要一个服务编排引擎，内置简单的流程引擎，只需要用XML或者DSL声明如何聚合服务，注册中心可以直接下发给消费者执行聚合逻辑，或者部署通用的编排服务器，所有请求由编排服务器转发
- 服务资源调度
    - 解决问题
        - 并不是所有服务的访问量都大，很多的服务都只有一丁点访问量，却需要部署两台提供服务的机器，进行HA互备，如何减少浪费的机器。
        - **机器总是的闲时和忙时，或者冗余机器防灾，如何提高机器的利用率?**
    - 解决方法
        - 此时可能需要让服务容器支持在一台机器上部署多个应用，可以用多JVM隔离，也可以用ClassLoader隔离。
        - 即然已经可以自动部署了，那根据监控数据，就可以实现资源调度，根据应用的压力情况，自动添加机器并部署。
- 容量规划
    - 解决问题
        - 服务调用量越来越大，服务的容量问题就暴露出来了，这个服务需要多少机器支撑，什么时候该加机器
    - 解决方法
        - 第一步，将每天的调用量和响应时间统计出来，作为容量规划的参考指标
        - 第二部，要动态可以调整权重，在线上将某一台机器的权重一直加大，直到达到响应时间的阈值，记录此时的访问量，再用访问量乘以机器数反推总容量