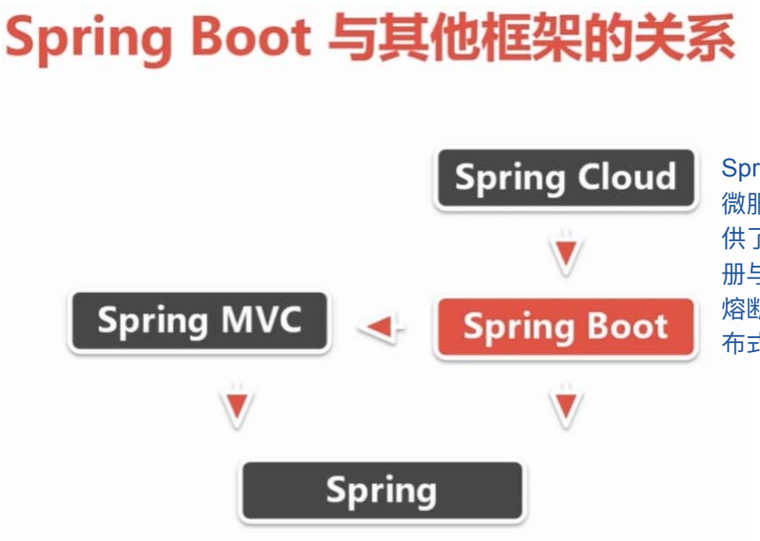
微服务学习笔记

1. Spring boot 与 Spring Cloud

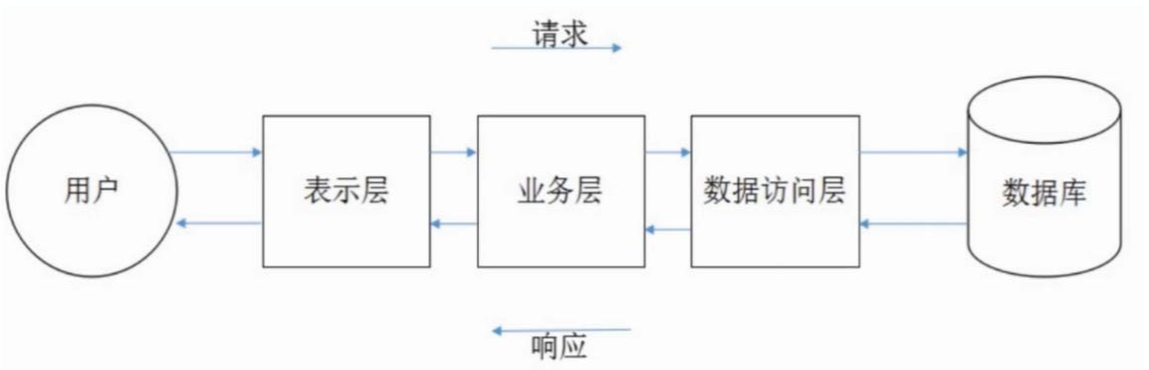


Spring boot 是 Spring 的一套快速配置脚手架，可以基于spring boot 快速开发单个微服务，是Spring的引导，用于启动Spring，使得Spring的学习和使用变得快速无痛。不仅适合替换原有的工程结构，更适合微服务开发。

Spring Cloud基于Spring Boot，为微服务体系开发中的架构问题，提供了一整套的解决方案——服务注册与发现，服务消费，服务保护与熔断，网关，分布式调用追踪，分布式配置管理等。

1. 单体架构

1.1 单体架构 架构图



1.2 单体架构 优缺点:

2. 微服务

2.1 微服务定义

In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

1. 多个高内聚低耦合的小程序构成
2. 独立进程，独立部署（多台服务器）
3. 服务间采取轻量级通信（Restful【基于 http】, RPC【基于 tcp】,异步消息【mq】）
4. 可以采用不同语言（技术栈）
5. 可以采用不同的数据库存储

2.2 微服务的优势

1. 易于开发与维护 （容易理解）（启动调试较快，提升开发效率）

2. 独立部署（部署上线不需要协调其他的服务，不用等到凌晨才部署上线）

3. 伸缩性强（横向、纵向、z 轴扩展）

- 每个子服务都可以在横向和纵向上进行扩展
- 每个服务都可以按照硬件资源的需求进行独立扩容

4. 与组织团队相匹配

不同的团队负责不同子服务，eg: 推荐模块->算法团队；用户模块->开发团队

5. 技术异构性

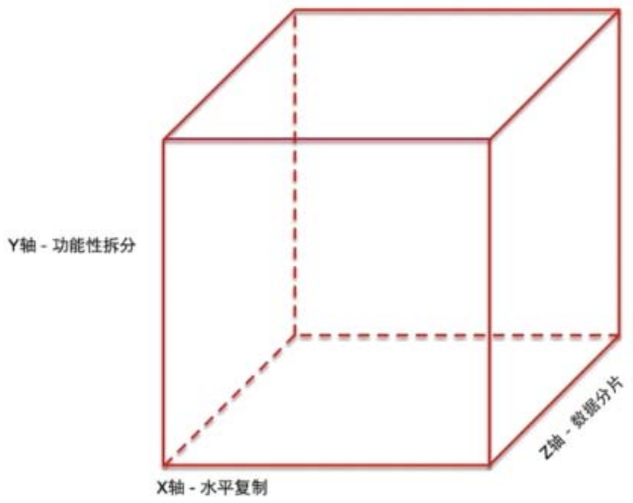
长连接推送服务：go 语言

搜索推荐：elastic search 作为存储和搜索引擎

用户服务：redis 作缓存

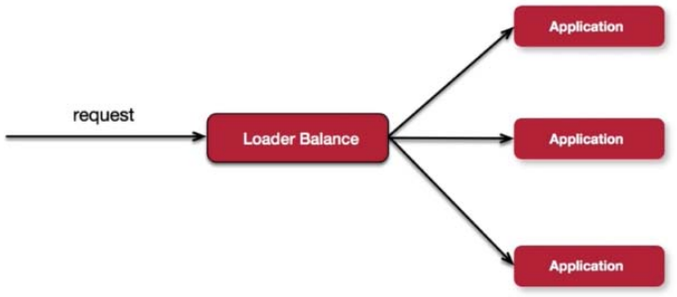
2.3 微服务的拆分

1. 伸缩立方

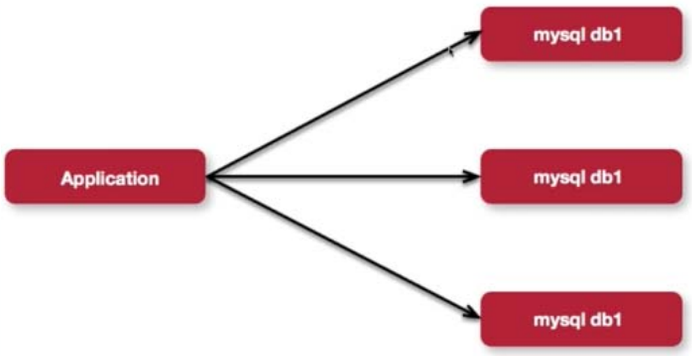


Z轴：分库、分表
Y轴：解决表与表之间的id竞争
x轴：application copy

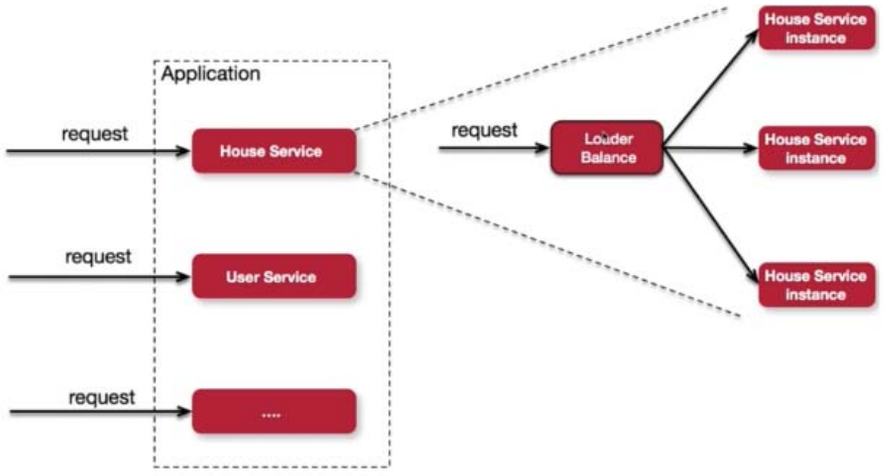
- X 轴拓展：Application 通过复制，部署到多个服务器上，负载均衡设备来分发请求



- Z 轴扩展-数据分片



- Y 轴扩展-功能性拆分



可以按照服务对硬件资源不同的需求进行升级
Eg：House Service 可以是 CPU 密集型的机器也可以是内存密集型的机器

微服务架构举例：

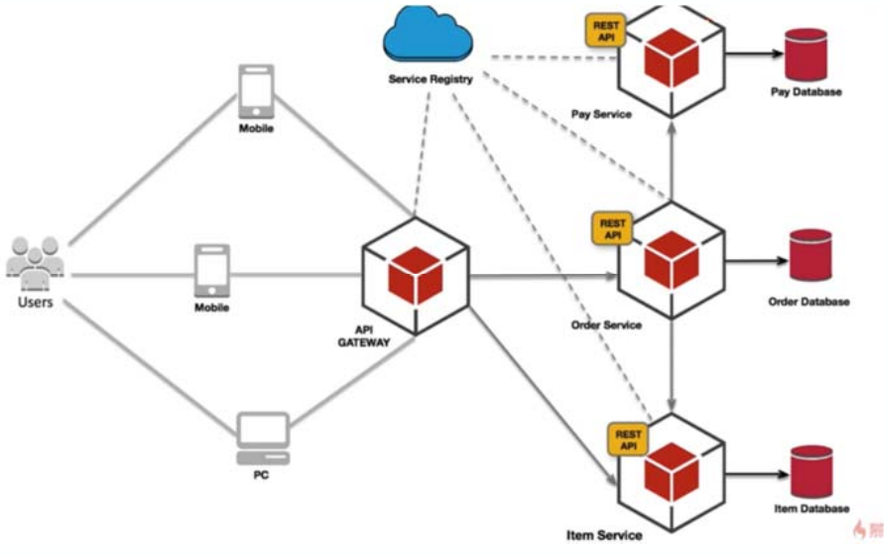


Figure 1 微服务架构图举例 1

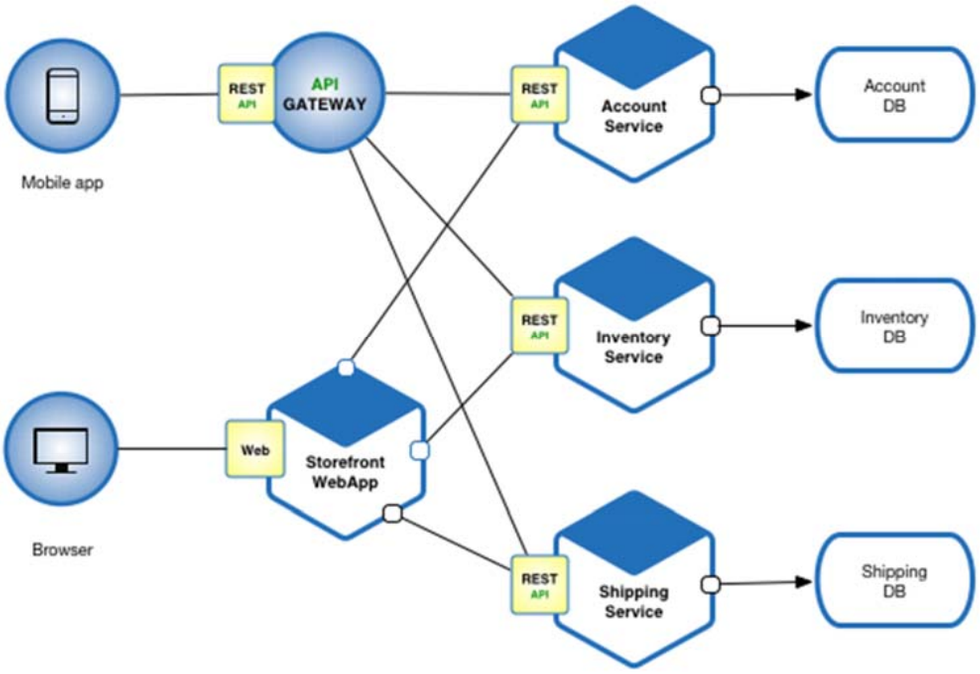


Figure 2 微服务架构图举例 2

2.4 微服务面临的挑战

1. 微服务拆分

- 服务拆分原则：领域模型、组织架构、单一职责
- 微服务拥有独立的数据库（现部署在同一服务器上，后续再隔离服务器）
- 确定服务边界：高内聚、低耦合（公开的 api 通信，接口隐藏内部的细节）、

2. 数据一致性

nosql(=not only sql)：泛指非关系型数据库,不保证关系数据的ACID特性,eg:HBase

- 分布式事务无法应用到微服务中：延时性高，nosql 数据库不支持；
  应该使用最终一致性代替强一致性
- 事务性的操作应该尽量放在同一服务中

补充

- *强一致性：系统中的某个数据被成功更新后，后续任何对该数据的读取操作都将得到更新后的值；*
- *弱一致性：系统中的某个数据被更新后，后续对该数据的读取操作可能得到更新后的值，也可能是更改前的值。但经过"不一致时间窗口"这段时间后，后续对该数据的读取都是更新后的值；*
- *最终一致性：是弱一致性的特殊形式，存储系统保证在没有新的更新的条件下，最终所有的访问都是最后更新的值。*

3. 服务通信

- 通信方案：RPC vs REST vs 异步消息
- 服务注册与发现 （Spring Cloud Eureka）
- 负载均衡

4. 服务网关 （内网与外网的边界）

- API Gateway: 身份认证、安全防御、路由服务、流量控制、日志统计
- 聚合服务：为前端服务的后端（聚合来自不同模块的数据返回给前端）

5．高可观察

- 集中监控：注册中心、服务进程、流量日志、服务状态都需要监控
- 日志聚合及检索（共享库）
- 分布式追踪

服务降级：当整个微服务架构整体的负载超出了预设的上限阈值或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，我们可以将一些 **不重要** 或 **不紧急** 的服务或任务进行服务的 **延迟使用** 或 **暂停使用**。

6. 可靠性

- 流量控制、超时控制
- 舱壁隔离（每一个服务独立的线程池）、熔断机制（服务调用出错在一定时间内达到一定的次数，自动关闭对该服务的开关，改为返回错误，或者转为降级方法）
- 服务降级、幂等重试（多次操作和一次操作的结果是一样的，保证了重试不会）

微服务关注全景图：

服务降级分类：

超时降级：主要配置好超时时间和超时重试次数和机制，并使用异步机制探测恢复情况

失败次数降级：主要是一些不稳定的API，当失败调用次数达到一定阀值自动降级，同样要使用异步机制探测回复情况

故障降级：如要调用的远程服务挂掉了（网络故障、DNS故障、HTTP服务返回错误的状态码和RPC服务抛出异常），则可以直接降级

限流降级：当触发了限流超额时，可以使用暂时屏蔽的方式来进行短暂的屏蔽