中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

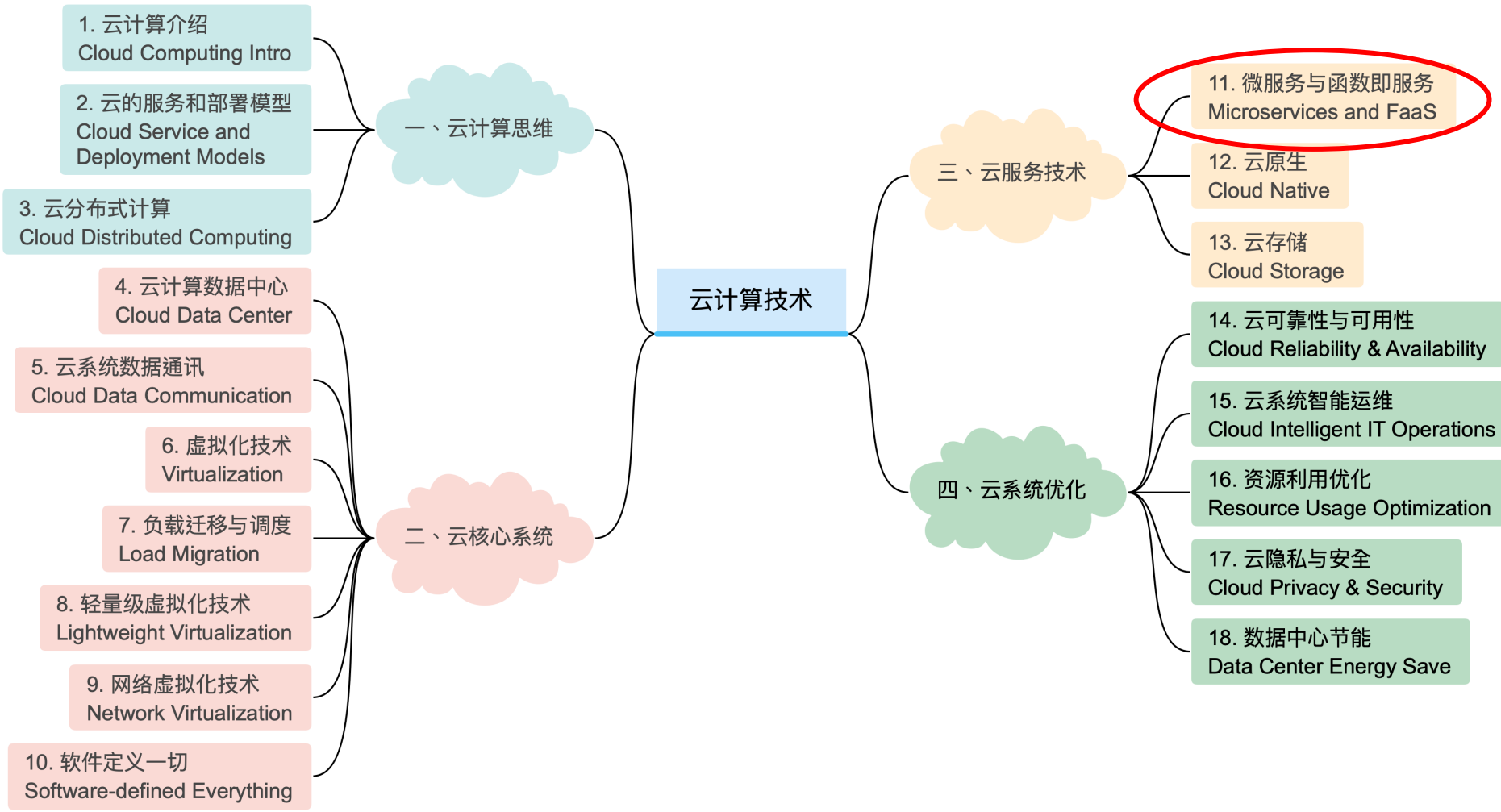
Lecture 11：微服务架构

SSE316：云计算技术
Cloud Computing Technologies

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn


云计算技术
一、云计算思维
1. 云计算介绍
Cloud Computing Intro
2. 云的服务和部署模型
Cloud Service and
Deployment Models
3. 云分布式计算
Cloud Distributed Computing
二、云核心系统
4. 云计算数据中心
Cloud Data Center
5. 云系统数据通讯
Cloud Data Communication
6. 虚拟化技术
Virtualization
7. 负载迁移与调度
Load Migration
8. 轻量级虚拟化技术
Lightweight Virtualization
9. 网络虚拟化技术
Network Virtualization
10. 软件定义一切
Software-defined Everything
三、云服务技术
11. 微服务与函数即服务
Microservices and FaaS
12. 云原生
Cloud Native
13. 云存储
Cloud Storage
四、云系统优化
14. 云可靠性与可用性
Cloud Reliability & Availability
15. 云系统智能运维
Cloud Intelligent IT Operations
16. 资源利用优化
Resource Usage Optimization
17. 云隐私与安全
Cloud Privacy & Security
18. 数据中心节能
Data Center Energy Save

Today’s topics

- 微服务的概念
- 微服务架构评估
- 云计算中间件
- 无服务器计算

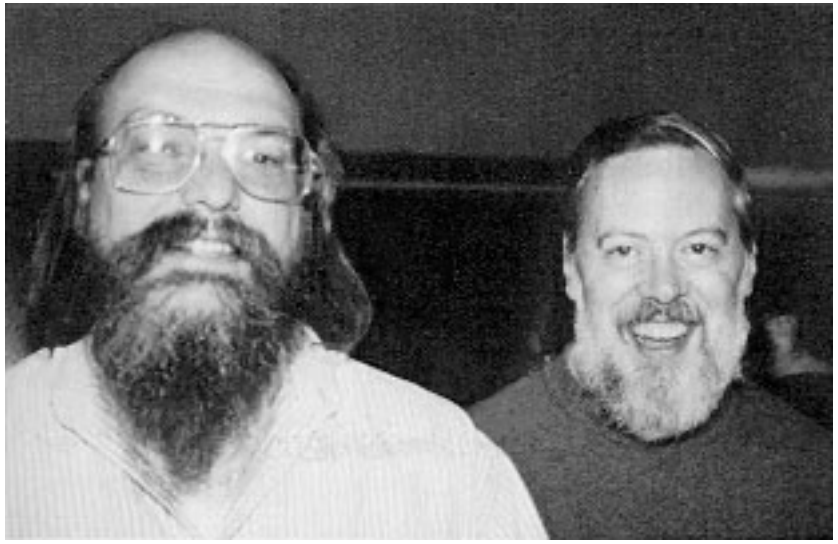
Part I

为什么需要微服务?

先从一个电商系统的成长故事开始：系统不是一开始就需要微服务，而是在复杂性超过单体承载能力时，微服务才有意义。

Unix 设计哲学

- The UNIX philosophy: (from Wikipedia [1])
  - Emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and repurposed by developers other than its creators
  - Favors composability as opposed to **monolithic design**
  - “Do one thing and do it well”



Ken Thompson 和 Dennis Ritchie，
The Unix philosophy 的主要倡导者

[1] https://en.wikipedia.org/wiki/Unix_philosophy

从 Unix 设计哲学到微服务

1 单一职责

2 组合优先

3 可演进

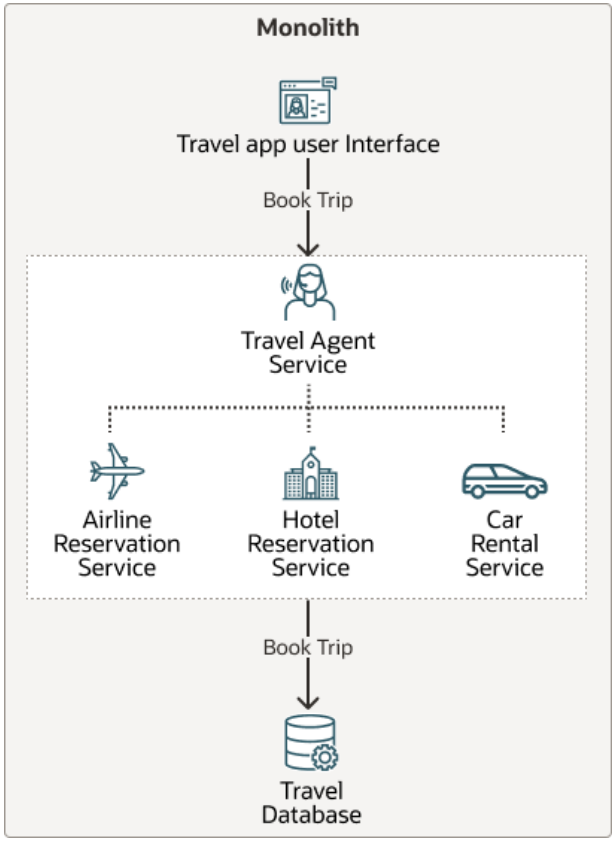
把复杂系统拆成小而清晰的组件，再通过稳定接口组合起来。

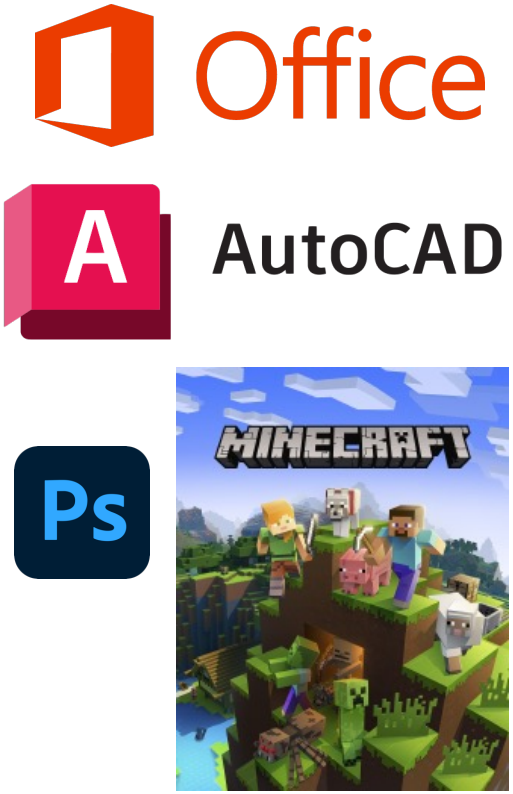
Unix philosophy 给微服务留下的直觉：单一职责、可组合、可替换。

单体软件架构 Monolithic architecture

- 单体软件架构
  - 一种传统的软件架构模式
  - 所有功能模块和组件都集成在**单个应用程序或单个可执行文件**中

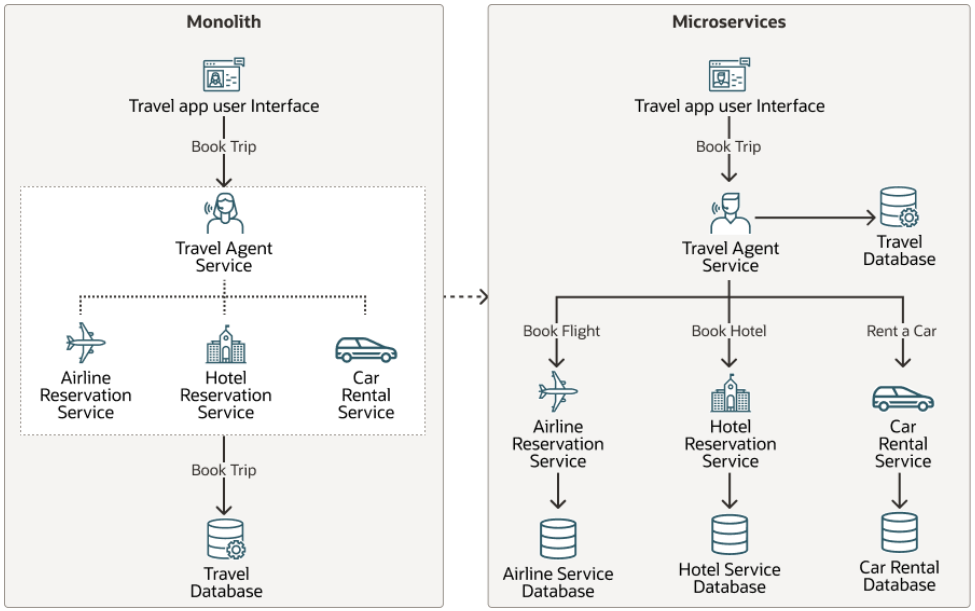
- 适用场景
  - 小型项目
  - 快速原型开发
  - 资源有限的场景
  - 桌面应用
  - …

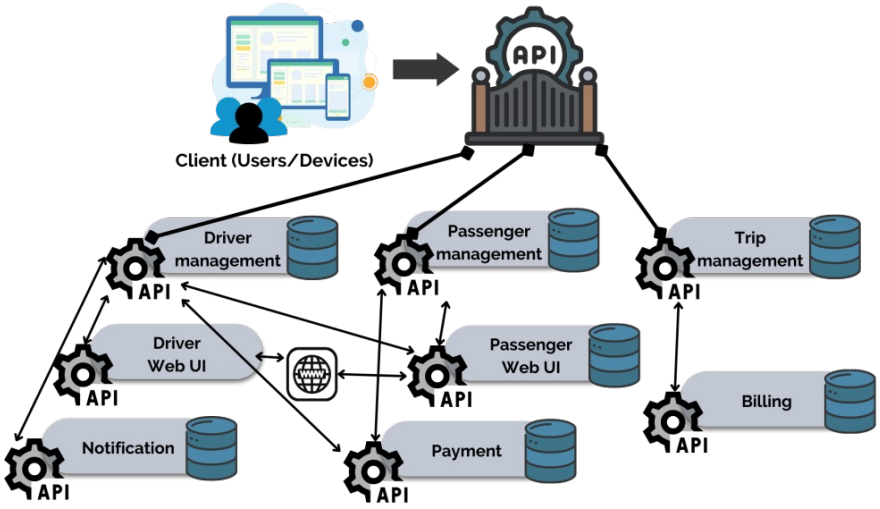




微服务架构

- Microservices (from Wikipedia [2])
  - Microservices are a **software development principle** that arranges an application as **a collection of loosely coupled services**
- Application
  - A complex service provided to the user (e.g., e-commerce portal)





[2] https://en.wikipedia.org/wiki/Microservices

一家电商从 3 张表起步

1 起步阶段

- 一个代码库
- 一个数据库
- 一个团队
- 一天上线一次

2 增长阶段

- 订单、库存、支付耦合
- 促销活动导致流量尖峰
- 多人同时改同一模块

3 平台阶段

- 团队变多
- 业务线变多
- 发布窗口变小
- 故障影响面变大

微服务的出场条件：不是“技术新”，而是“组织、业务、发布、故障隔离”同时给系统施压。

单体应用不是坏架构：它只是会长大

+ 单体的优点

- 开发简单：一个工程就能启动
- 事务简单：一个数据库内完成
- 调试简单：调用链短、部署少

! 单体的边界

- 模块耦合逐渐变强
- 小改动也要整体回归
- 局部扩容很困难

? 判断标准

- 业务是否快速变化
- 团队是否互相阻塞
- 故障是否经常全站扩散

微服务不是为了否定单体，而是为了管理已经超出单体舒适区的复杂性。

单体的典型痛点：牵一发而动全身

1. **构建慢** 改一行库存逻辑，也要重新构建整个应用
2. **发布难** 支付模块修复 bug，必须等待所有模块一起发布
3. **扩容粗** 只有商品浏览流量高，却要扩容整个应用
4. **故障大** 通知模块线程池耗尽，拖慢订单下单
5. **边界糊** 谁能改订单状态？接口和数据库都能改

微服务尝试把这些痛点从“全局问题”拆成“局部可控的问题”

解决的不是代码行数，而是变化速度

模块	变化频率	风险特点	更适合的治理方式
商品展示	高	促销活动多，前端/接口常改	可快速独立发布
订单核心	中	状态机复杂，强业务一致性	接口契约稳定，重点测试
支付	低	合规、安全和外部通道依赖	小心变更，强审计
通知	高	模板、渠道、触达策略常变	异步化，故障隔离

当不同模块的生命周期差异很大时，把它们放在同一个发布单元里，会让整个系统按最慢的节奏前进。

云环境放大了微服务的价值

云原生平台提供的基础能力

镜像构建 — 容器运行 — 服务复制 — 流量调度 — 自动恢复

HPA 弹性伸缩

- 只扩订单服务或商品服务
- 成本随热点变化而变化

ISO 故障隔离

- 一个服务异常不应拖垮全站
- 用限流、熔断、降级保护核心链路

CI 自动化交付

- 每个服务独立构建和发布
- 用流水线、灰度、回滚降低风险

这个系统现在适合拆成微服务吗？

一个课程项目只有 3 名同学、2 周开发周期、功能还在探索中，是否应该一开始就做微服务？

A 应该：微服务更先进，架构看起来更专业

B 不应该：先用清晰模块化单体更稳妥

C 折中：先拆 10 个服务，再慢慢补自动化

推荐答案：B。微服务有额外的通信、部署、测试和运维成本；没有业务复杂性和团队规模时，模块化单体通常更合适。

Part II

微服务的核心思想

微服务不是“把应用切碎”，而是让每个服务围绕业务能力独立演进，并明确契约协作。

什么是微服务？一句话定义

微服务是一组围绕业务能力构建、可独立部署、通过轻量通信协作的小型服务。

关键词：业务能力 · 独立部署 · 明确接口 · 自治团队

S 小

小到能被一个团队理解、修改、测试和发布。

A 自治

服务内部的实现、数据和发布节奏尽量独立。

API 协作

服务之间通过 API、消息和契约协同完成业务。

经典定义：松耦合 + 有边界 + 可独立演进

“由松耦合元素组成的面向服务架构，每个元素都有清晰的 bounded context。”

Bounded Context

语义边界清楚，同一个词在不同服务里可以有不同含义。

轻量通信

HTTP、RPC、消息等机制承载协作，接口比内部实现更重要。

独立交付

每个服务能独立开发、测试、部署和回滚。

数据责任

服务拥有自己的核心数据和业务规则，避免随意共享表。

这 4 个特征比“服务数量”更能判断一个系统是不是健康的微服务架构。

服务边界：按业务能力，而不是按技术层

口不要把 Controller、Service、DAO 各拆成服务

X 按技术层拆分：看起来规整，但耦合仍在

- user-controller-service
- order-service-layer
- payment-dao-service
- 跨服务调用像远程函数调用

OK 按业务能力拆分：边界更接近真实职责

- 用户服务：身份与地址
- 订单服务：订单生命周期
- 库存服务：库存预留与扣减
- 支付服务：支付请求与回调

微服务的边界应该回答：“这个业务能力由谁负责？谁拥有规则和数据？”

独立部署：微服务最硬的标准之一

- 每个微服务应该像一个独立产品一样，被单独修改、测试、发布和回滚

理想的订单服务变更流程

OK 独立部署意味着

- 服务有自己的构建产物
- 服务有自己的发布流水线
- 失败时可以独立回滚

! 不意味着

- 没有任何依赖
- 可以随意破坏 API
- 完全不需要集成测试

REQ 关键前提

- 接口契约稳定
- 自动化测试可靠
- 运行平台能承接发布

自治团队：架构也是组织协作方式

口微服务不只是技术架构，也会改变团队协作方式

按职能分组

需求流过多个团队，排队和交接成本高；一个功能可能横跨多个部门。

按业务能力分组

订单小队：从需求到上线

支付小队：从需求到上线

商品小队：从需求到上线

服务边界和团队边界互相影响。

去中心化数据管理：服务拥有自己的数据

- 微服务不止是把代码拆成多个服务就可以了，如果所有服务还在直接读写同一个数据库，系统本质上仍然强耦合
- 因为真正的业务规则，往往藏在数据和表结构里

X 共享数据库的问题

- 绕过 API 直接改表，业务规则失控
- 表结构变更会影响多个服务
- 数据库成为事实上的单体核心
- 权限、安全和审计边界模糊

OK 服务拥有数据的含义

- 订单服务管理订单表
- 库存服务管理库存账本
- 支付服务管理支付流水
- 跨服务通过 API 或事件交换数据

实践中可以先做到“逻辑 ownership 清晰”，再逐步做到物理数据库隔离。

API 契约：服务之间的“课程表”

订单服务 API

POST /orders	创建订单
GET /orders/{id}	查询订单
POST /orders/{id}/pay	发起支付
POST /orders/{id}/cancel	取消订单

{} 输入输出

字段含义、必填项、错误码要稳定。

v 兼容演进

新增字段通常安全，删除/改语义很危险。

T 契约测试

消费者和提供者都要验证接口假设。

微服务不是免费的：复杂性从代码转到系统

收益	代价
服务可独立发布	部署对象变多，流水线和环境管理更复杂
局部扩容更灵活	服务之间通过网络调用，网络调用和序列化带来延迟与失败
团队自治更强	跨服务需求需要契约、版本和协调
故障隔离更好	需要熔断、限流、追踪和告警能力
技术栈可局部演进	标准化、治理和知识共享更重要

一句话：微服务把“模块内复杂性”换成了“分布式系统复杂性”

服务越小越好吗？

L 太粗

- 边界不清
- 发布仍然互相阻塞
- 局部扩容困难

OK 合适

- 业务能力完整
- 团队能独立负责
- 数据 ownership 清楚

S 太细

- 一次业务跨越太多服务
- 网络调用成本高
- 排查故障困难

经验法则：一个服务要小到可理解，大到能表达完整业务能力。

以下哪个更像微服务边界？

要拆一个电商系统，下列拆法哪一个更符合“围绕业务能力”的思想？

A Controller 服务、Service 服务、DAO 服务

B 订单服务、库存服务、支付服务、通知服务

C MySQL 服务、Redis 服务、Nginx 服务

推荐答案：B。微服务优先按业务能力拆，而不是按技术层或基础设施组件拆。

Part III

如何拆分微服务系统?

以“电商下单”为主线，学习从业务流程、数据 ownership、变化频率和团队职责中找服务边界。

案例对象：一次电商下单到底发生了什么？

登录用户 → 浏览商品 → 提交订单 → 预留库存 → 发起支付 → 确认订单 → 发送通知

服务拆分不是先画技术组件，而是先理解业务链路中的职责和约束。

U 用户视角

- 我能不能买？
- 要付多少钱？
- 订单是否成功？

B 业务视角

- 库存是否足够？
- 支付是否到账？
- 失败后如何恢复？

S 系统视角

- 哪些服务参与？
- 哪些调用必须同步？
- 哪些信息可以异步？

从业务能力识别服务候选

用 用户服务

账号、地址、会员身份

商 商品服务

商品信息、价格展示

订 订单服务

订单生命周期与状态机

库 库存服务

库存扣减、预留、释放

支 支付服务

支付请求、回调、退款

通 通知服务

短信、邮件、站内信

候选服务不是最终答案：还要继续用数据、事务、调用频率和团队能力来验证。

领域驱动设计的直觉：Bounded Context

同一个词在不同上下文里含义不同：边界清楚，接口才不会越来越含糊。

商品上下文

商品=可展示、可搜索、可定价的销售对象

库存上下文

商品=库存账本中的 SKU 与仓位数量

订单上下文

商品=订单明细中的快照，不随商品改名而改变

支付上下文

订单=支付单关联的业务单号与金额

SKU: Stock Keeping Unit (库存最小单位)

拆分维度：四个问题帮你找到边界

D 谁拥有数据？

- 核心表由哪个服务维护
- 其他服务是否只读副本或通过 API 访问

VAR 谁经常变化？

- 变化频率类似的模块更容易放在一起
- 变化频率差异大时更适合拆开

= 谁必须强一致？

- 强一致边界内尽量少跨服务
- 跨服务一致性通常用事件与补偿

T 第四个问题：谁能独立负责？

服务边界最终要落到团队、代码仓库、发布流水线和运行责任上。

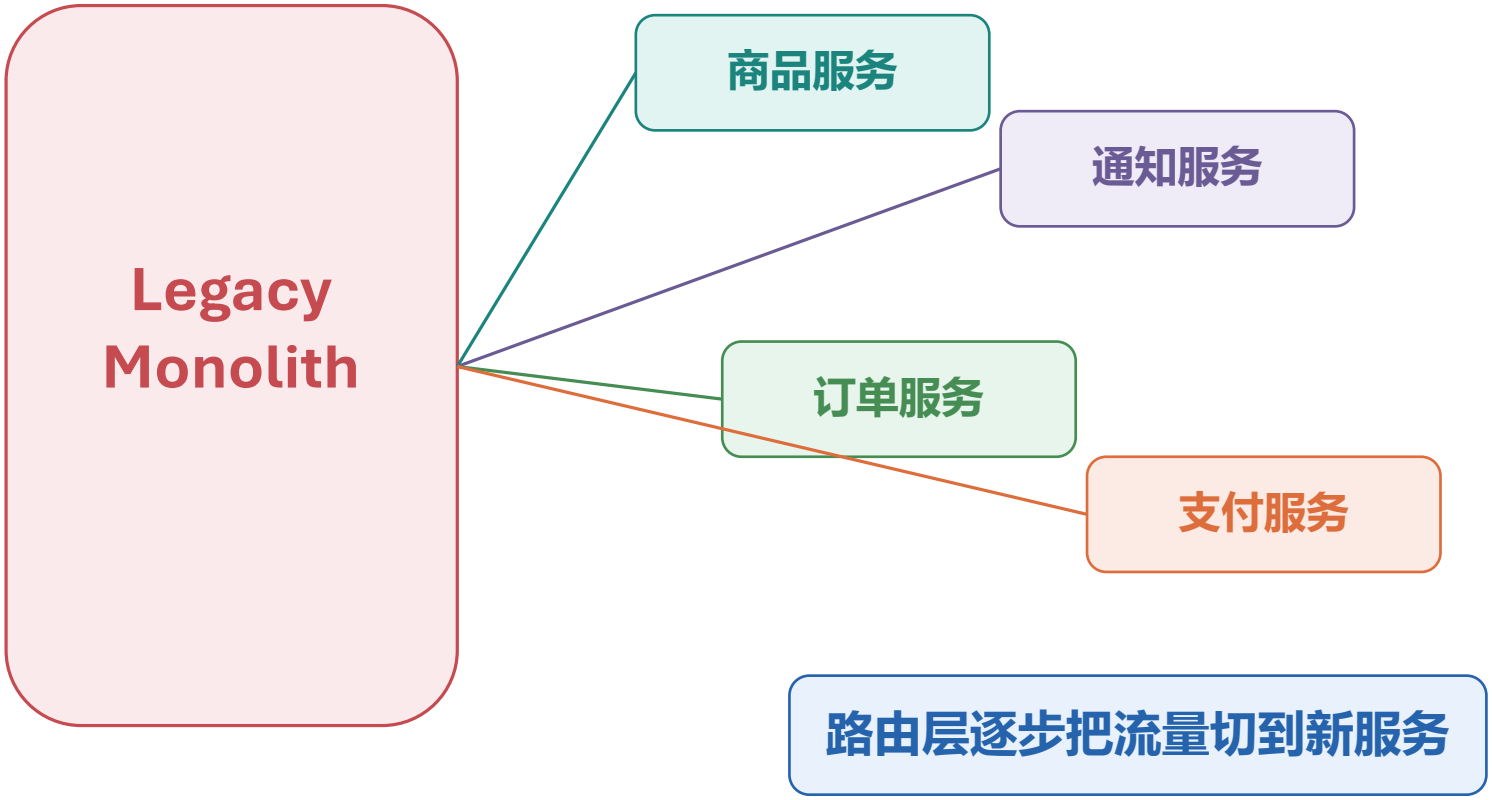
从单体到微服务：推荐的演进路径

1. 先把单体内部模块边界整理清楚，禁止随意跨模块访问数据。
2. 抽离变化频率高、边界清晰、风险相对可控的外围能力。
3. 通过 API 或事件建立契约，让新服务承担真实流量。
4. 逐步迁移数据 ownership，保留回滚路径和双写/同步策略。
5. 补齐自动化测试、部署、监控和告警，再扩大拆分范围。

微服务化更像“搬家”，不是“炸掉旧楼再建新楼”。

Strangler Fig Pattern：用新服务包围旧系统

- 一种低风险迁移方式



好处：一次迁移一个能力，旧系统仍可运行，风险更可控。

反模式：分布式单体

LOOK 看起来像微服务

- 有很多独立进程
- 有 HTTP 或 RPC 调用
- 每个服务都有独立仓库
- 画架构图很漂亮

! 实际上仍是单体

- 所有服务必须同一时间发布
- 一个服务改字段，所有服务跟着改
- 共享同一个数据库核心表
- 一次请求必须串行经过十几个服务

微服务的目标是降低协作阻塞；如果拆完之后更难发布、更难定位，说明边界可能错了。

订单和库存要不要拆成两个服务？

下单时必须预留库存。订单团队和库存团队发布节奏不同，库存还被门店系统复用。更合理的方案是？

A 放在一个服务里，避免任何跨服务通信

B 拆成订单服务和库存服务，通过 API/事件协作

C 拆成 20 个库存字段级服务，粒度越细越好

推荐答案：B。库存是独立业务能力且被多个场景复用，但需要设计预留、释放、幂等和补偿逻辑。

服务边界的 5 条检查线

1. 这个服务是否代表一个清晰的业务能力，而不是技术层？
2. 这个服务是否能拥有自己的核心数据与业务规则？
3. 它是否能在大多数情况下独立开发、测试和部署？
4. 跨服务调用是否可接受，失败时是否有降级或补偿策略？
5. 团队是否有能力承担这个服务的运行责任与监控告警？

基于微服务交互的架构评估

口微服务之间复杂的交互

- 单个服务可能非常小且简单，但服务之间的**交互可能非常复杂**
- 即**复杂性从服务内部转移到了架构级别**，即服务之间的交互

Trace Analysis Based Microservice Architecture Measurement

Xin Peng*† Fudan University China

Chenxi Zhang* Fudan University China

Zhongyuan Zhao* Fudan University China

Akasaka Isami* Fudan University China

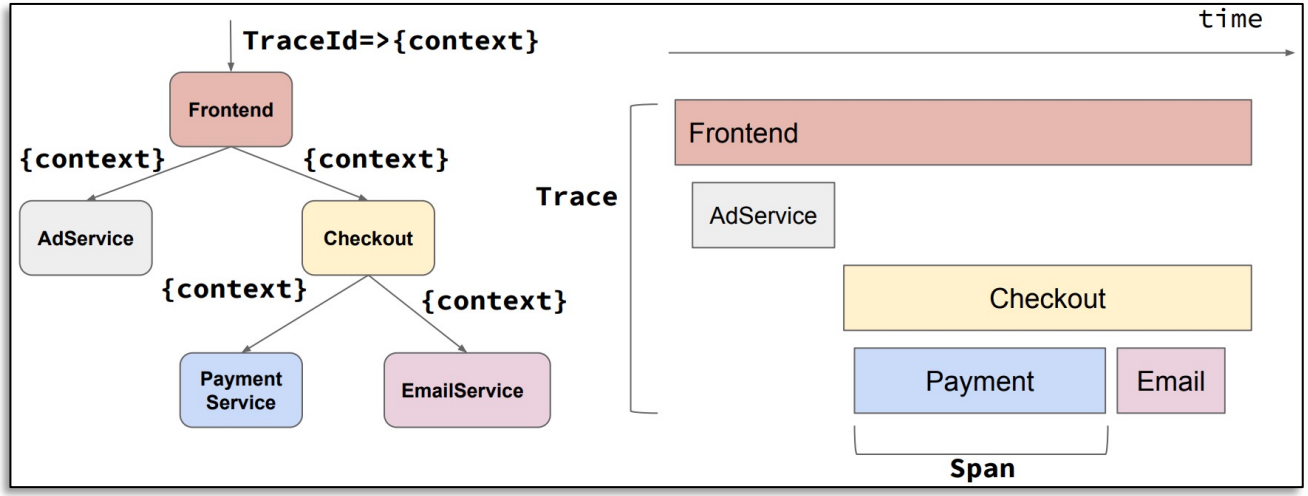
Xiaofeng Guo Alibaba Group China

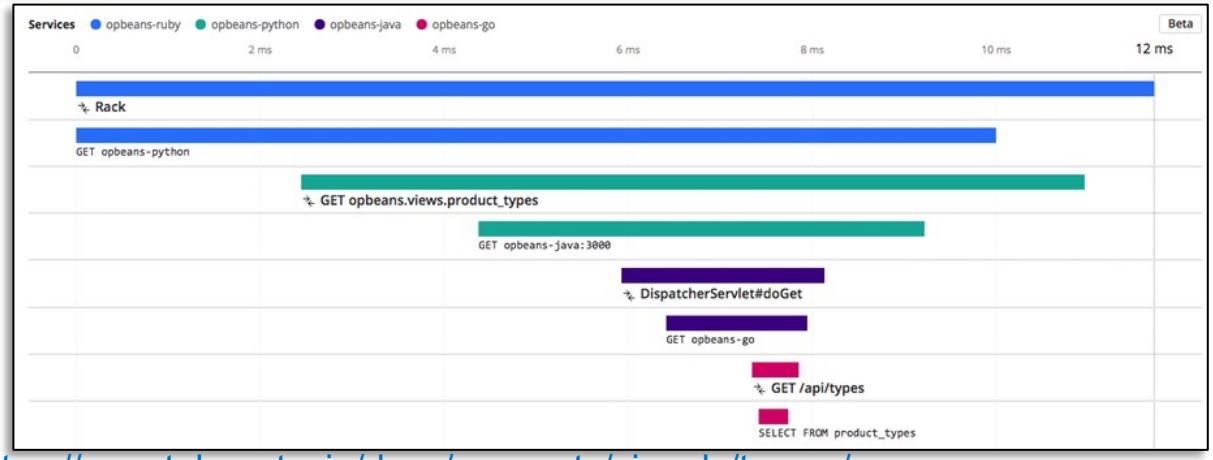
Yunna Cui* Fudan University China

提出一个用于微服务架构度量的 **trace 数据模型**，该模型能够对**请求的执行过程**以及**接口和服务之间的交互**进行细粒度分析

分布式追踪 Distributed tracing

Trace 记录一个请求从接收到处理并最终响应的完整路径，包括经过的所有服务和这些服务的交互过程






```
{
  "name": "hello",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```


https://opentelemetry.io/docs/concepts/signals/traces/

为什么要评估微服务架构？

1. **服务过小**　两个服务总是一起调用、一起修改，拆分可能过度。
2. **重复服务**　功能相似、数据相似、调用者相似，却由不同团队维护。
3. **职责混乱**　一个服务承担多个不相关任务，接口目标分散。
4. **循环依赖**　A 调 B，B 又调 A，发布和理解都变困难。
5. **调用链过长**　一次请求穿过太多服务，性能与可用性都受影响。

架构评估的目标不是给系统打分，而是发现下一步应该合并、拆分还是改造调用链。

评估只问两个核心问题

IND

问题 1：服务是否足够独立?

- 接口目标是否集中?
- 是否经常和同一批服务一起变化?
- 是否和其他服务访问同一类数据?
- 是否需要合并过小服务或拆分混乱职责?

LIC

问题 2：调用链是否过于复杂?

- 一次请求经过多少服务?
- 是否存在循环调用或重复调用?
- 故障是否容易沿链路扩散?
- 是否能用批处理、缓存或异步事件简化?

三组指标的直觉

指标组	它在看什么	架构提示
内聚性 Cohesion	一个服务的接口是否围绕同一目标、访问相似数据、被相似调用者使用	低内聚通常提示职责混乱，可能需要拆分
耦合性 Coupling	两个服务是否经常一起出现、共享数据属性、被同一上游或下游关联	高耦合通常提示过度拆分或重复服务，可能需要合并
调用链复杂性	一次请求链路有多长，是否重复调用、循环调用、跨越太多服务	链路过长通常提示性能、可用性和理解成本风险

下单系统如何做健康检查？

观察到的现象	可能的问题	下一步动作
订单服务每次都同步调用通知服务，通知服务故障会拖慢下单	调用链不该把外围动作放进核心链路	改成 OrderCreated 事件异步通知
库存预留和库存查询拆成两个服务，但总是一起发布、一起扩容	服务过小，高耦合	考虑合并为库存服务的不同接口
商品服务既管理价格、图片，又管理库存账本	职责混乱，数据规则混在一起	把商品展示与库存账本分开
订单详情页需要串行查 8 个服务	查询链路复杂，用户体验受影响	用 BFF/API 组合或读模型优化

看到这条下单 trace，你会怎么改？

checkout -> order -> stock -> product -> stock -> payment -> notify，其中 stock 被重复调用，notify 同步阻塞主链路。

A 优先把 notify 改成异步事件，并检查 stock 重复调用是否可批量化

B 把所有服务合并回一个单体，避免任何网络调用

C 继续拆分 stock，让每个字段都有一个服务

推荐答案：A。主链路先保核心交易；重复调用优先考虑批量化、缓存或重构调用层次，而不是盲目合并或继续拆碎。

Part IV

微服务如何通信?

微服务不是消除依赖，而是把依赖显式化。通信设计决定了系统的延迟、可用性和故障传播方式。

服务通信的两种基本节奏

SYNC 同步调用

- 调用方等待结果
- 适合需要立即回答的查询或校验
- 失败会直接影响用户链路
- 常见：REST、gRPC

ASY 异步消息

- 调用方发送事件后继续执行
- 适合通知、积分、日志、数据同步
- 削峰填谷，但要处理重复和延迟
- 常见：Kafka、RabbitMQ

选择通信方式时先问：用户是否必须立即知道结果？失败是否应该阻塞主链路？

REST 和 RPC：两种远程调用的基本思想

REST **REST：围绕资源**

- URL 表示资源：/orders/123
- HTTP 动词表示操作：GET/POST/PUT/DELETE
- 强调无状态、统一接口、可读性
- 常见载荷：JSON

REST 像是在操作一组业务资源

RPC **RPC：围绕动作**

- 把远程服务当成本地函数调用
- 方法名表示动作：CreateOrder()
- 强调明确契约、参数和返回值
- 常见实现：gRPC、Thrift、Dubbo

RPC 像是在调用一个远程函数

在微服务里，它们都是服务间通信方式；选择重点是可读性、契约强度、性能和生态。

REST 与 gRPC：不是谁更高级，而是谁更适合

维度	REST/HTTP+JSON	gRPC/Protobuf
可读性	人类容易调试，浏览器/工具支持好	二进制协议，可读性依赖工具
契约	可用 OpenAPI 描述	IDL 强契约，生成客户端代码
性能	足够通用，开销略高	高性能，适合内部高频调用
生态	开放 API、前端调用友好	服务内部调用、流式传输友好
课堂记忆	像发一封明文邮件	像使用约定格式的内部专线

API Gateway：微服务系统的前门

- 微服务系统对外的统一入口，客户端不直接访问各个微服务，而是先访问网关
- 隐藏内部服务结构，降低客户端复杂度，也便于统一治理外部流量

服务发现：服务实例每天都在变化

- 服务发现是“通过服务名动态找到可用服务实例”的机制
- 由于服务实例会频繁启动、下线、扩容和故障迁移，调用方不能写死 IP 地址，而应通过注册中心查询当前可用实例，并配合负载均衡完成请求转发

1 服务注册

实例启动后把地址、端口和健康状态登记到注册中心。

2 服务发现

调用方通过服务名查询可用实例，而不是写死 IP。

3 负载均衡

在多个实例之间分配请求，避开不健康节点。

调用时看到的是服务名，不是固定机器

消息中间件的三个作用

- 消息中间件是服务之间传递消息和事件的通用基础设施
- 它把发送方和接收方隔开：上游服务只负责发布事件，下游服务按自己的节奏消费消息

DEC 应用解耦

- 发送方只发布事件
- 消费者可以增减
- 下游短暂不可用不阻塞主链路

ASY 异步提速

- 先完成核心交易
- 外围动作稍后处理
- 用户等待时间更短

BUF 削峰填谷

- 高峰堆积消息
- 下游按能力消费
- 把瞬时压力变成可处理队列

注意：消息队列不是万能胶，重复消息、顺序、延迟和补偿都要被认真设计。

消息队列：给系统一个缓冲带

- 消息队列是服务之间的异步通信缓冲区
- 订单服务把“订单已创建”这类事件写入队列，下游服务再按自己的处理能力消费消息

BUF 削峰填谷

- 高峰先写入队列
- 消费者按能力慢慢处理

DEC 解耦依赖

- 订单不必知道所有下游
- 新增消费者不改订单服务

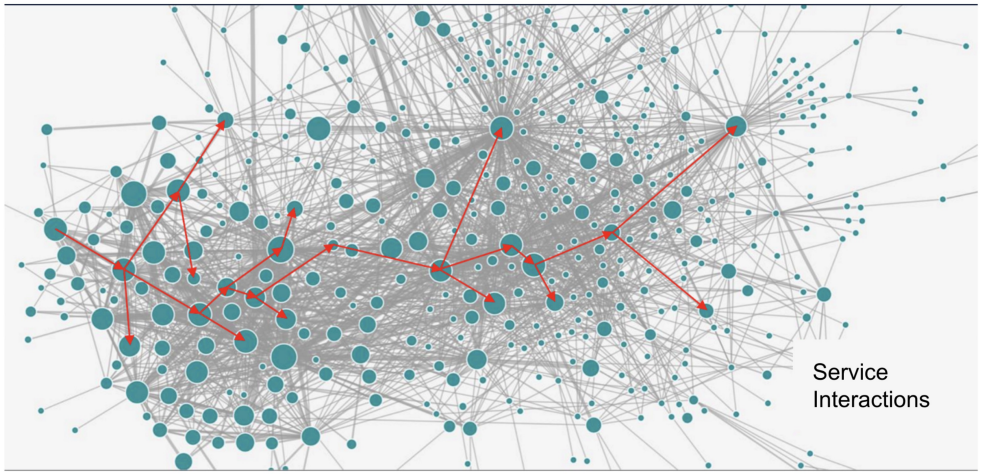
! 新问题

- 消息重复
- 消息延迟
- 顺序与补偿

调用链越长，可用性越脆弱

一次下单的同步链路

网关 — 订单 — 库存 — 支付 — 优惠券 — 风控



Complex microservice
RPC call graph at Uber

如果每个服务可用性都是 99.9%，6 个串行依赖的整体可用性约为 99.4%。

服务越多，单点看起来更可靠，但链路整体更需要超时、重试、熔断、降级和异步化。

超时、重试、熔断：远程调用三件套

T Timeout

- 不要无限等待
- 超时时间要小于用户可接受等待
- 下游慢时及时释放资源

R Retry

- 只重试可安全重试的请求
- 指数退避 + 抖动
- 必须考虑幂等

CB Circuit Breaker

- 连续失败后快速失败
- 给下游恢复时间
- 配合降级结果

这些机制的共同目标：让故障停在局部，不要沿调用链扩散。

版本兼容：接口不是你一个人的事

1. 新增字段通常兼容，删除字段或改变字段含义通常不兼容。
2. 先让服务端兼容新旧格式，再逐步升级客户端。
3. 接口废弃需要公告、指标观察和明确下线窗口。
4. 关键接口使用契约测试，避免“我以为你不会改”。

记忆点：微服务发布像接力赛，不能把接力棒突然换成另一种形状。

同步还是异步？

用户下单成功后，系统要发送短信、增加积分、写运营报表。这些动作应该怎样处理？

A 全部同步完成后再返回用户

B 下单核心链路同步，通知/积分/报表用事件异步处理

C 全部异步，用户不用知道订单是否创建成功

推荐答案：B。核心业务结果要及时反馈，外围动作可以异步，降低主链路延迟和故障传播。

Part V

数据与一致性

微服务最难的部分通常不是拆代码，而是拆数据之后如何保持业务正确。

每个服务一个数据库？先理解原则

OK 原则

- 服务拥有自己的数据模型和业务规则
- 其他服务不能直接改它的表
- 跨服务数据通过 API、事件或只读视图同步

PATH 落地路径

- 早期可共享数据库但分 schema/权限
- 成熟后逐步物理拆库
- 拆库前先解决数据同步和回滚策略

真正危险的不是“同一台数据库”，而是“谁都能直接改谁的数据”。

分布式事务为什么难?

一次下单涉及多个独立资源

订单 DB — 库存 DB — 支付通道 — 消息队列

! 部分成功

库存预留成功，但支付失败怎么办?

? 网络不确定

请求超时，是失败了，还是成功但响应丢了?

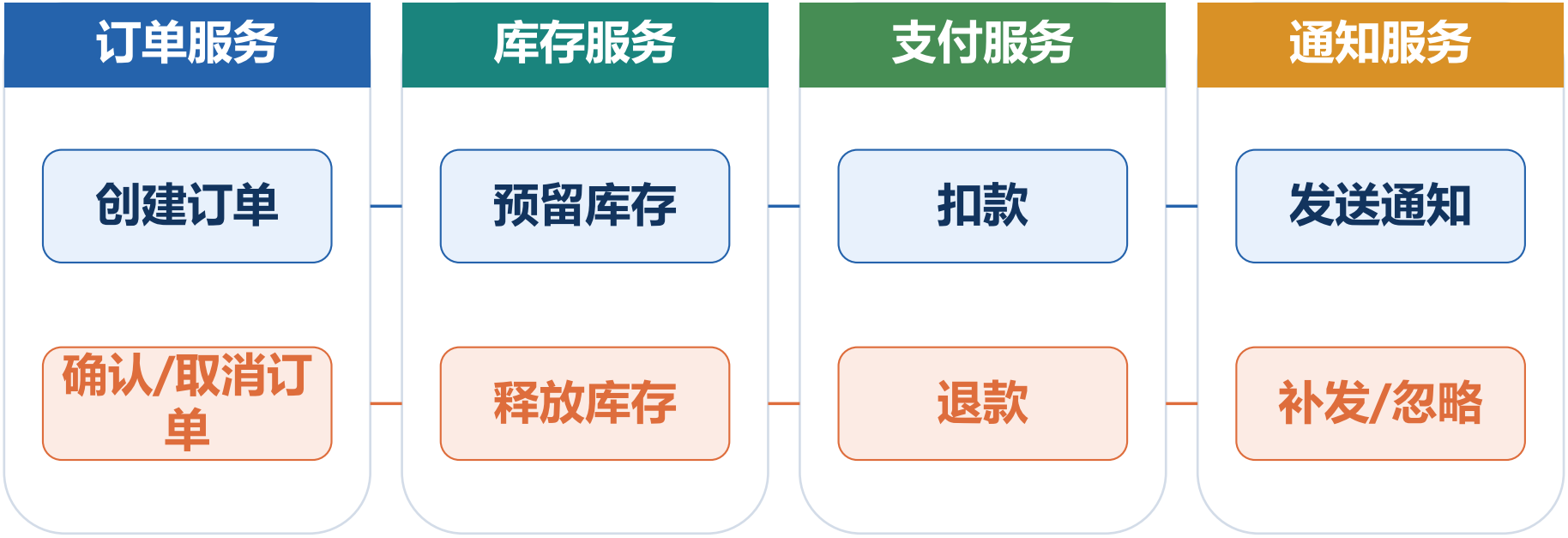
LOCK 锁范围大

长事务会拖慢服务并降低可用性。

微服务更常见的选择：避免跨服务强事务，用业务补偿和最终一致性。

Sage：把大事务拆成一串可补偿的小事务

- 前面某一步已经成功了，但后面失败了，就用一个反向业务动作把状态修回来



核心思想：用一串本地事务 + 补偿动作，替代一个横跨所有服务的大事务。

最终一致性：不是不一致，而是经过过程达到一致

□系统允许在短时间内处于中间状态，但经过事件、重试、补偿、同步之后，最终会达到业务上可接受的一致状态

下单成功后，积分可能几秒后到账；通知可能稍后到达；报表可能分钟级更新。

0s 立即一致

- 支付金额
- 订单状态关键流转
- 库存预留结果

s 秒级一致

- 通知发送
- 积分增加
- 优惠券状态同步

m 分钟级一致

- 运营报表
- 搜索索引
- 推荐特征

幂等性：重复请求不能造成重复业务效果

问题：支付回调发了两次，订单会不会被确认两次？库存会不会扣两次？

手段	直觉解释
业务唯一键	同一个 order_id + payment_id 只能处理一次
状态机校验	订单已支付时再次收到支付成功事件，直接忽略
去重表/日志	记录已经消费过的消息 ID

幂等让重试变得安全，是微服务可靠性的地基。

事件驱动：让状态变化被看见

- 微服务之间不要总是互相“命令对方做事”，也可以把自己的状态变化发布成事件，让其他服务自己决定怎么响应
- 比如**订单服务**创建订单后，不一定直接调用所有下游服务，而是发布一个事件

N 事件命名

- 用过去式：OrderCreated
- 避免命令式：CreateOrder

E 事件内容

- 带业务 ID 与时间
- 携带足够上下文，但不过度暴露内部表结构

C 事件消费

- 可能重复消费
- 可能延迟到达
- 消费者要幂等

支付成功回调超时了怎么办？

支付平台调用支付服务回调接口超时，于是稍后重试。支付服务应该如何设计？

A 拒绝第二次回调，避免麻烦

B 用 payment_id/order_id 做幂等，重复回调返回成功

C 每次回调都重新扣库存

推荐答案：B。外部系统重试很常见，内部处理必须幂等；重复消息不应造成重复业务效果。

该不该微服务化？一个判断框架

OK 适合拆
- 业务边界清晰
- 团队规模较大
- 发布互相阻塞

GO 先补课
- 模块化单体
- 契约测试
- 数据 ownership

! 暂缓拆
- 业务还在探索
- 边界不清
- 团队运行能力不足

G 拆完要评估
- 内聚性
- 耦合性
- 调用链复杂性

中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn