

技术方案 · 架构选型 · 设计评审

系统设计：技术方案与架构设计

wxquare · 原分享 2024-06-12

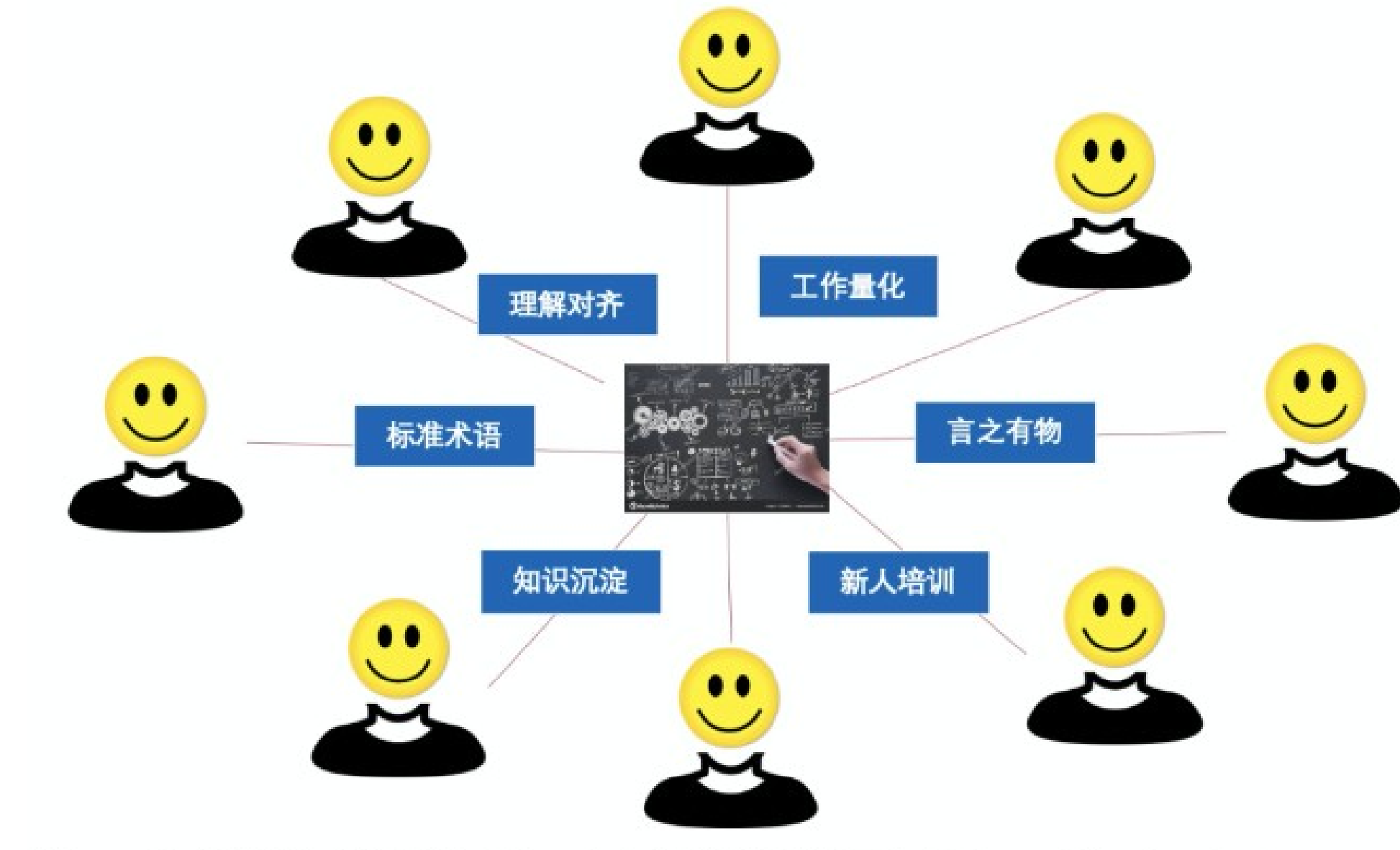
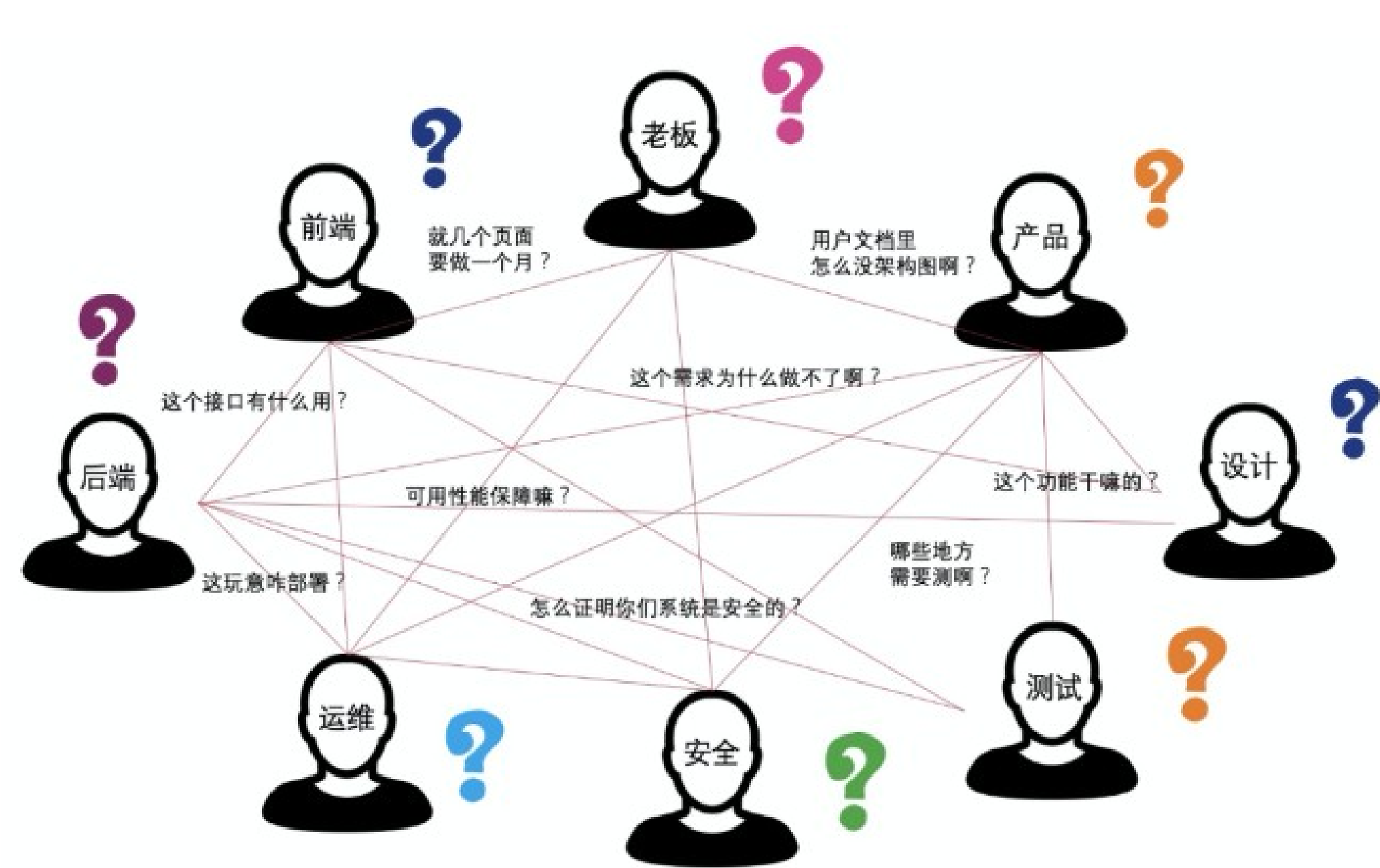
前言：课前说明

1. 主要涉及后端系统设计，不涉及前端开发和算法等
2. 不会探讨非常庞大的业务架构设计，而是聚焦于日常工作中常见的技术方案设计
3. 主要参考《数据密集型应用》以及 github 相关项目
4. 尽量减少‘八股文’，而是探讨平时做技术方案的问题
5. 建议 5 年以下的学习，也希望大佬给一些建议

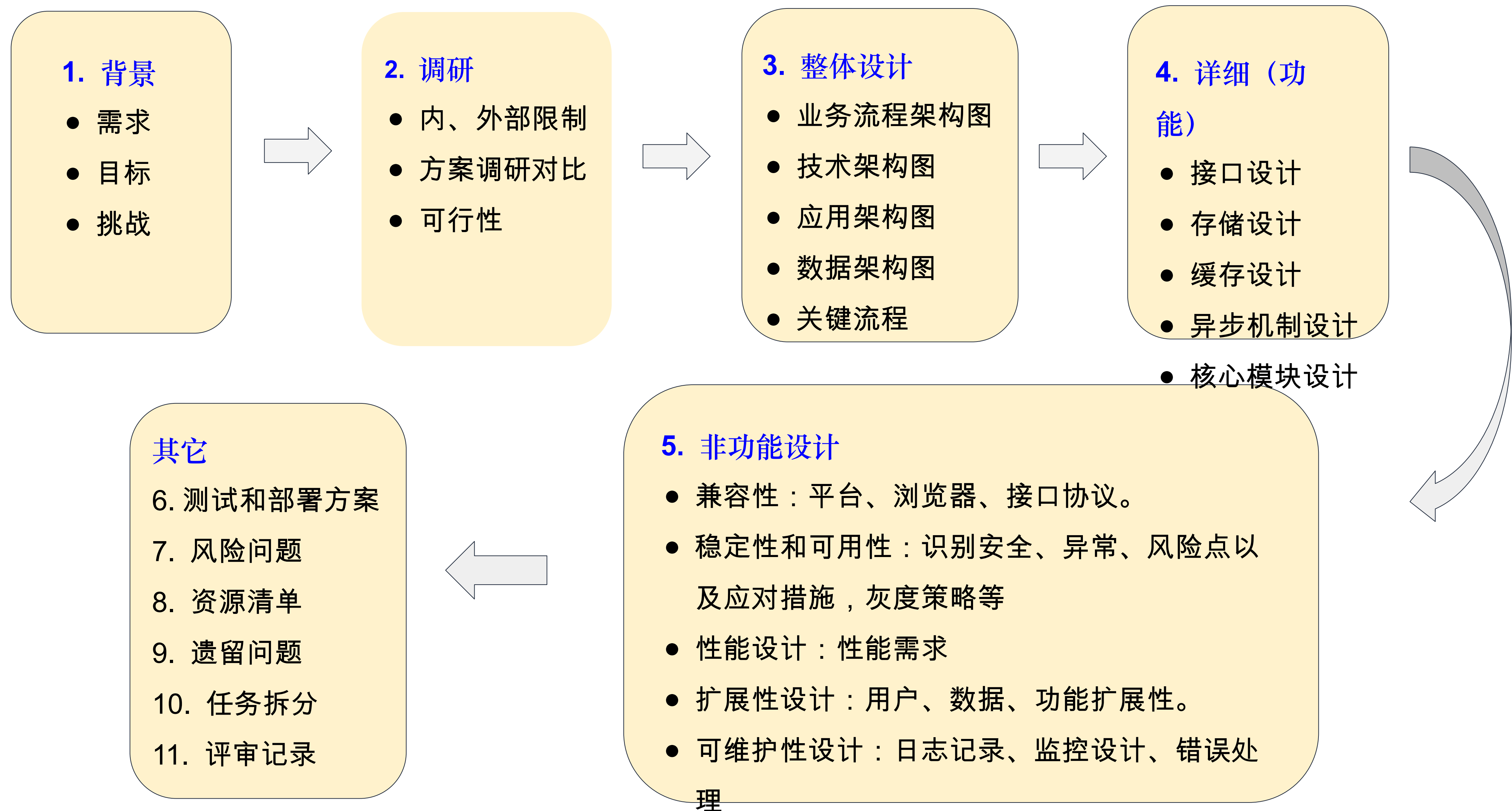


- <https://github.com/Vonng/ddia>
- <https://github.com/leewaiho/Clean-Architecture-zh>
- <https://github.com/donnemartin/system-design-primer>

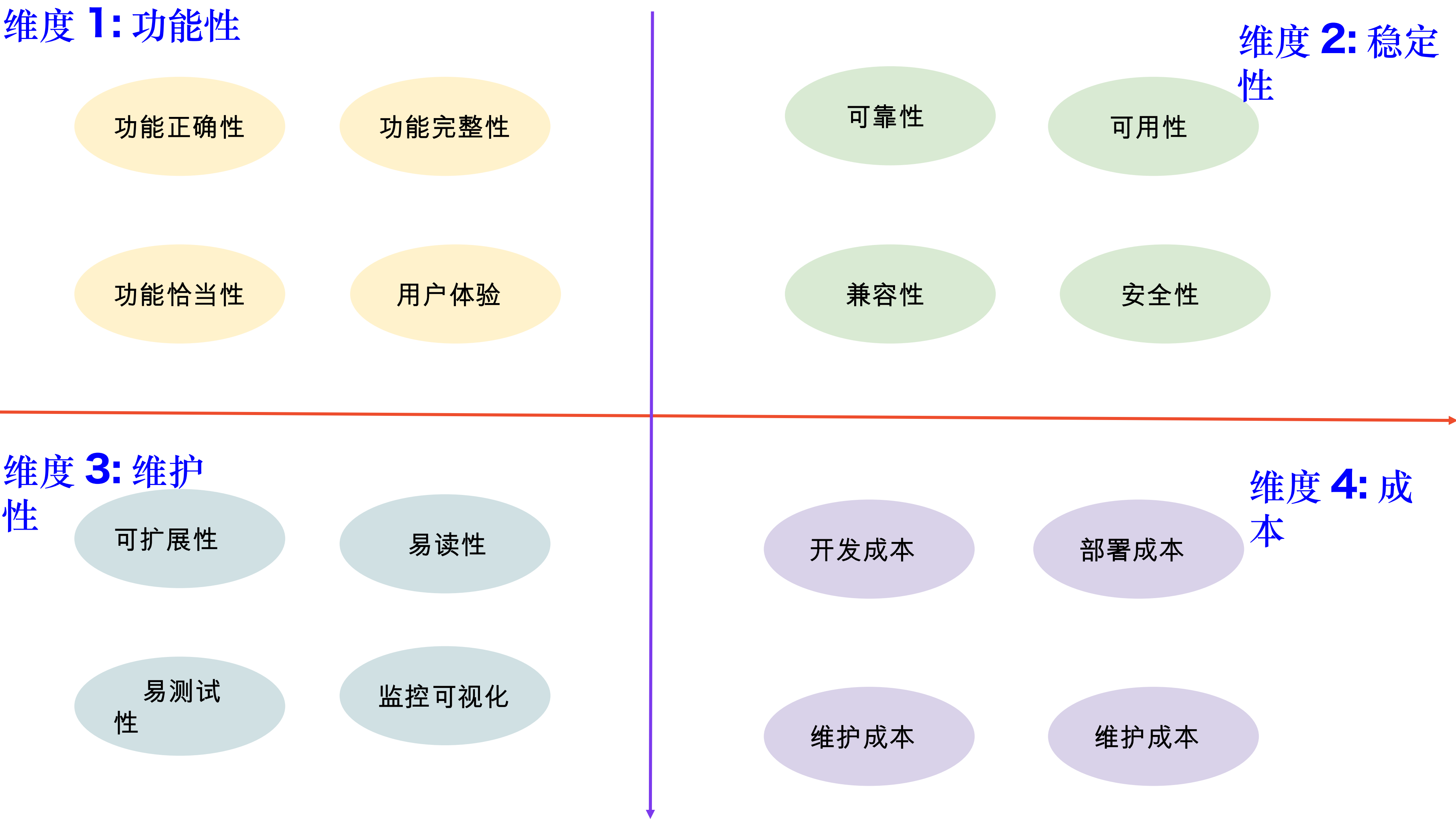
前言：为什么要做技术方案设计？



前言： 技术设计应包含哪些内容？



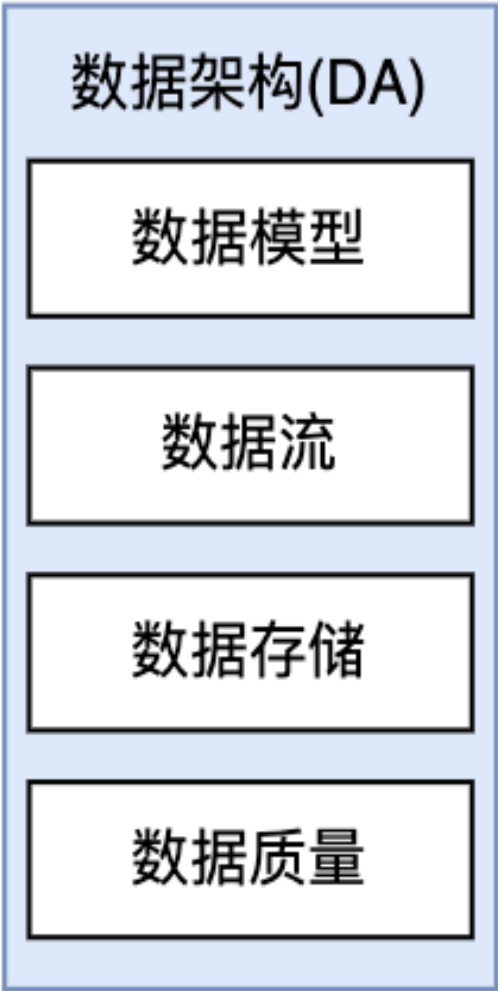
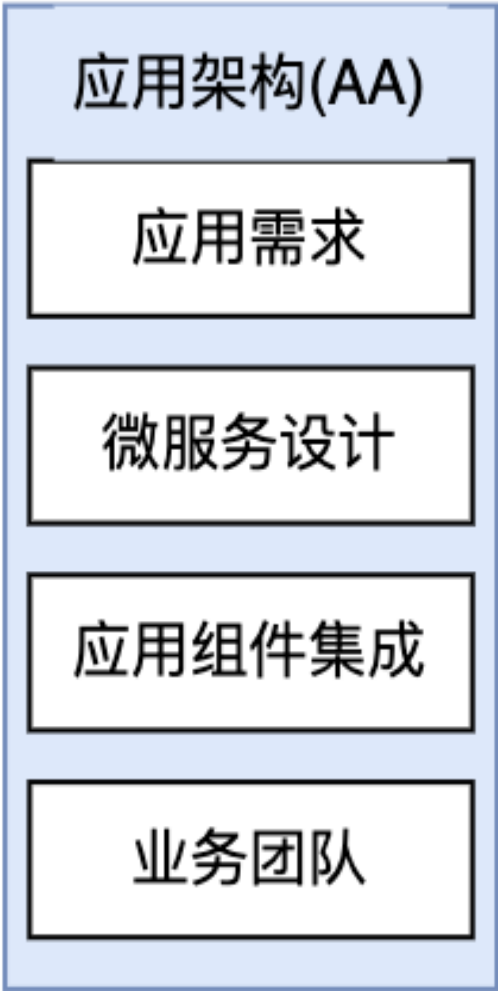
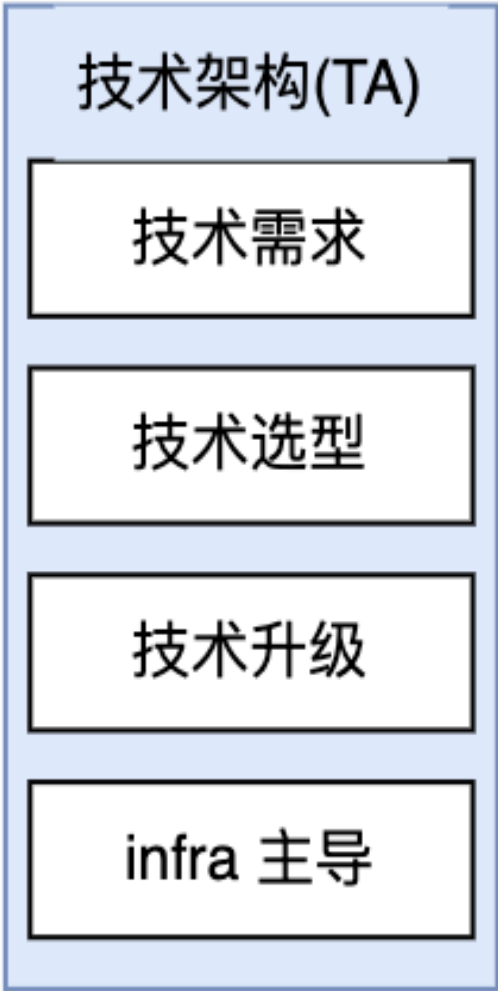
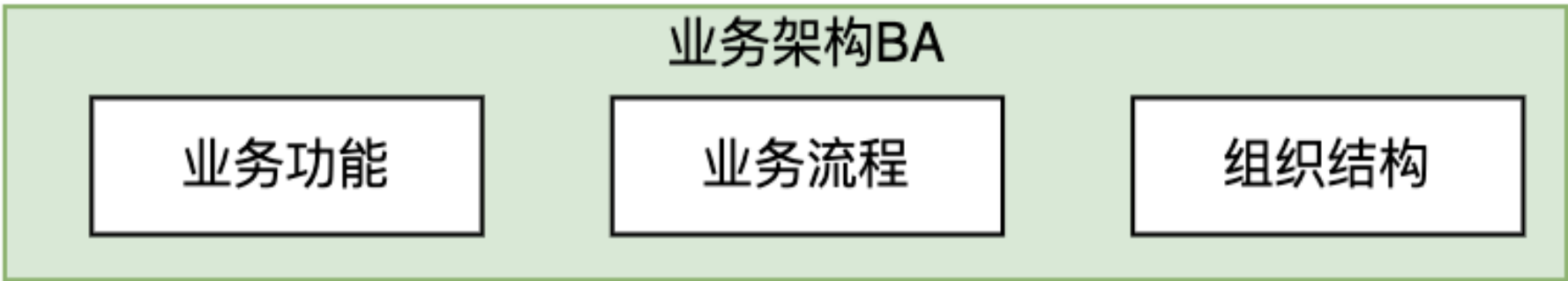
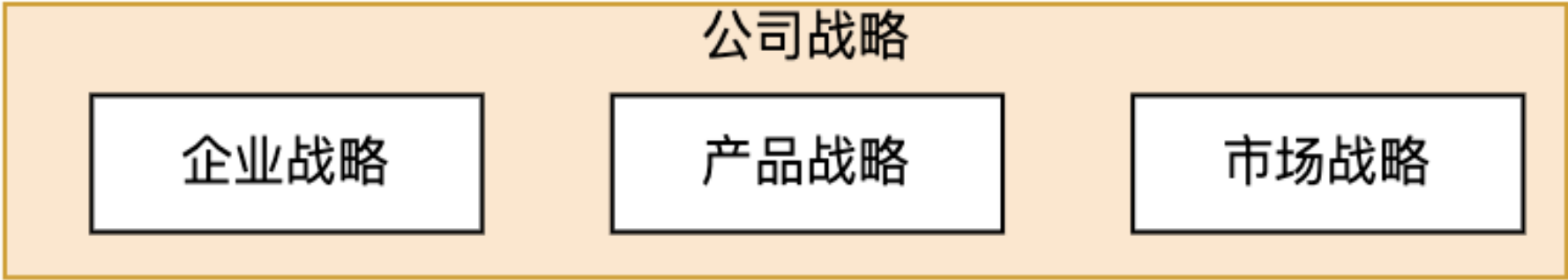
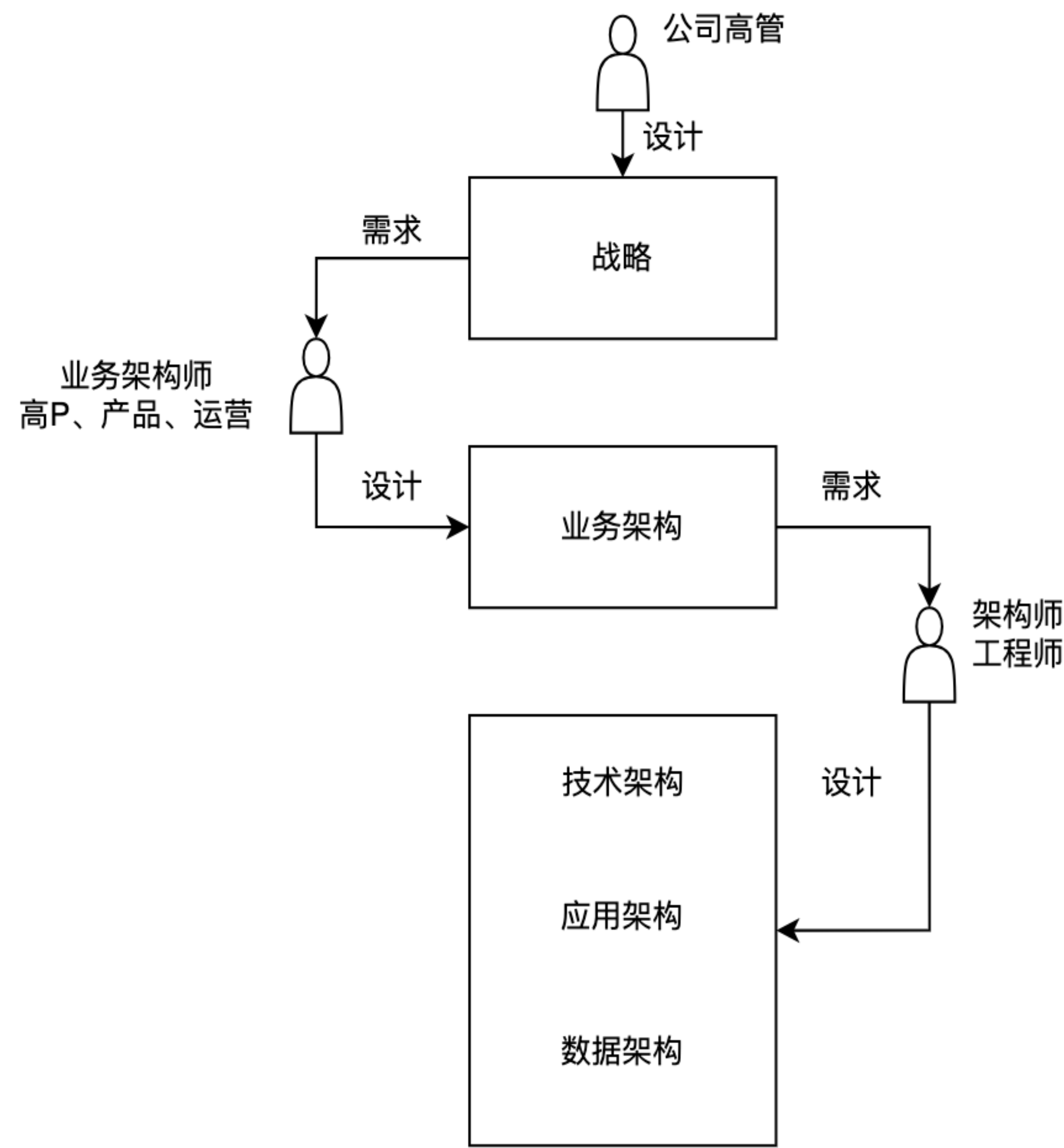
前言：怎么评价技术设计的质量？



目录

- ❑ 1. 整体性设计
- ❑ 2. 接口设计
- ❑ 3. 存储设计
- ❑ 4. 缓存设计
- ❑ 5. 异步设计
- ❑ 6. 非功能性设计

整体设计：什么是架构？ EA+4A



整体设计：业务架构（BA）

定义：

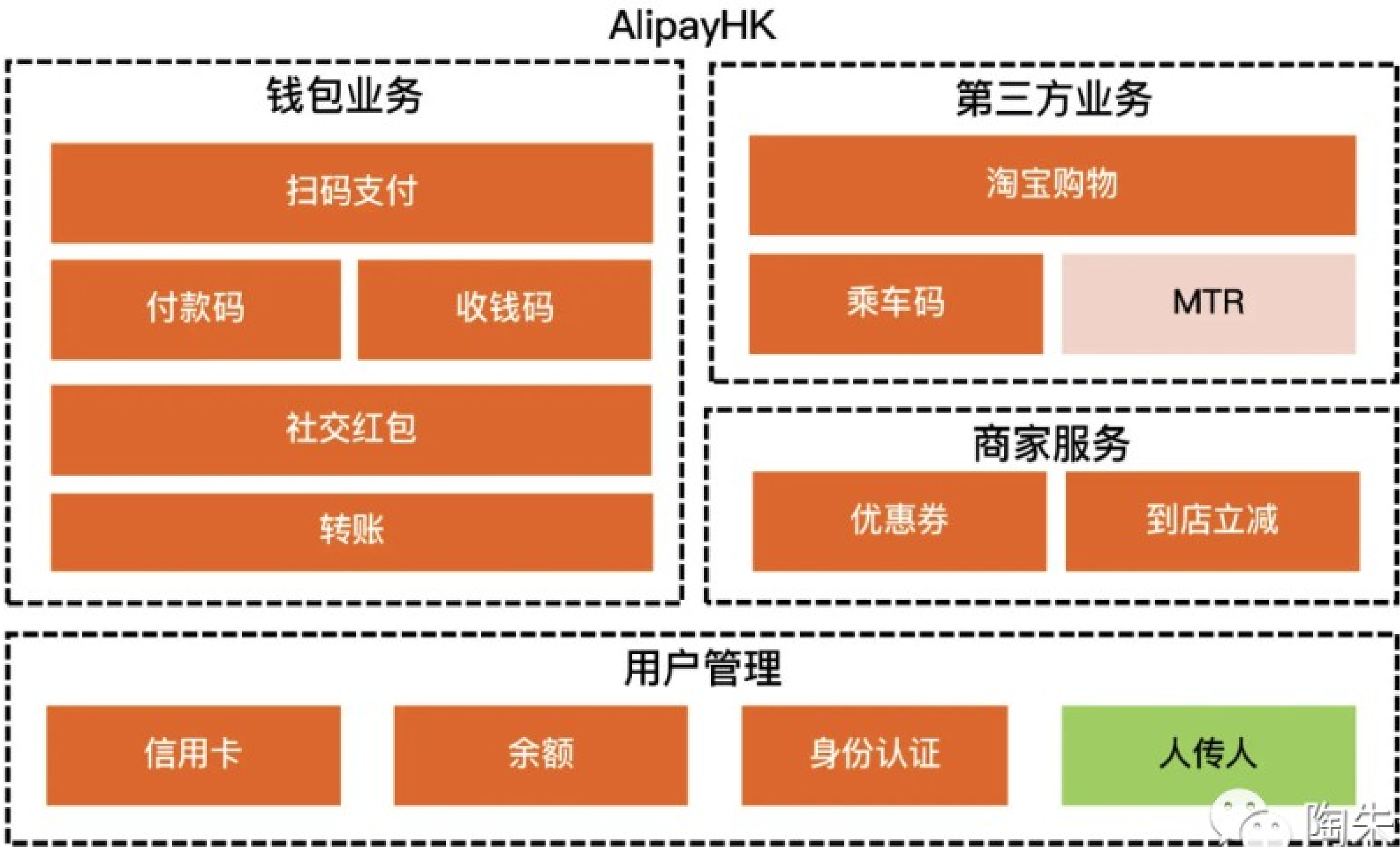
描述系统对用户提供的什么业务功能
核心是：流程、组织、模块、边界

使用场景：

- 1. 产品人员规划业务
- 2. 给高 P 汇报工作
- 3. 新员工培训

画图技巧：

- 1. 通过不同颜色标识业务状态
- 2. 业务分组管理

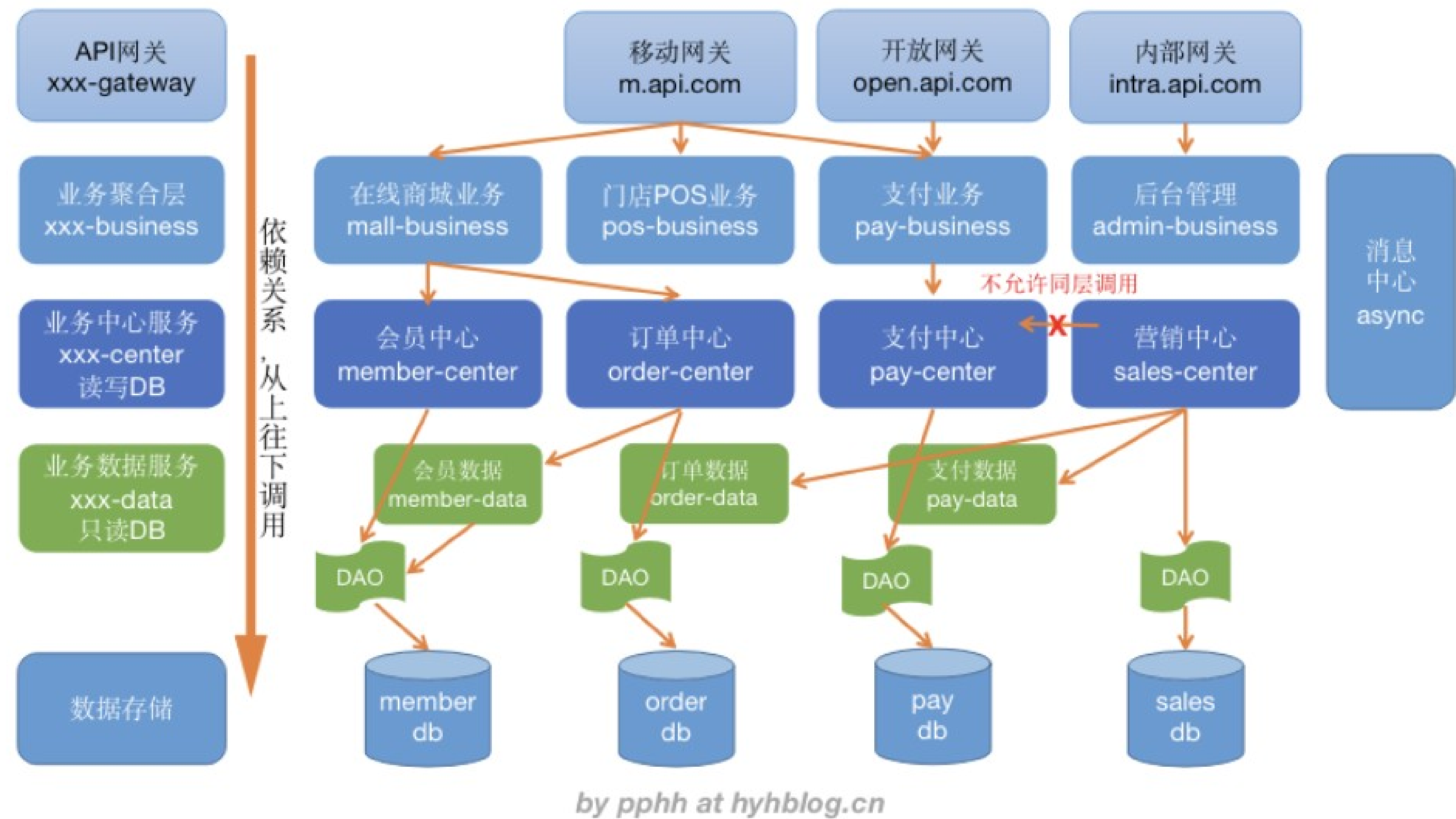


整体设计：应用架构（AA）

定义：
描述列系统中各服务之间的交互关系

使用场景：
1. 开发、测试、SRE
2. 新员工培训

画图技巧：
1. 分层管理
2. 连线



- 1. 服务调用关系从上到下、禁止同层调用
- 2. 应用名遵从格式规范、快速识别逻辑层次

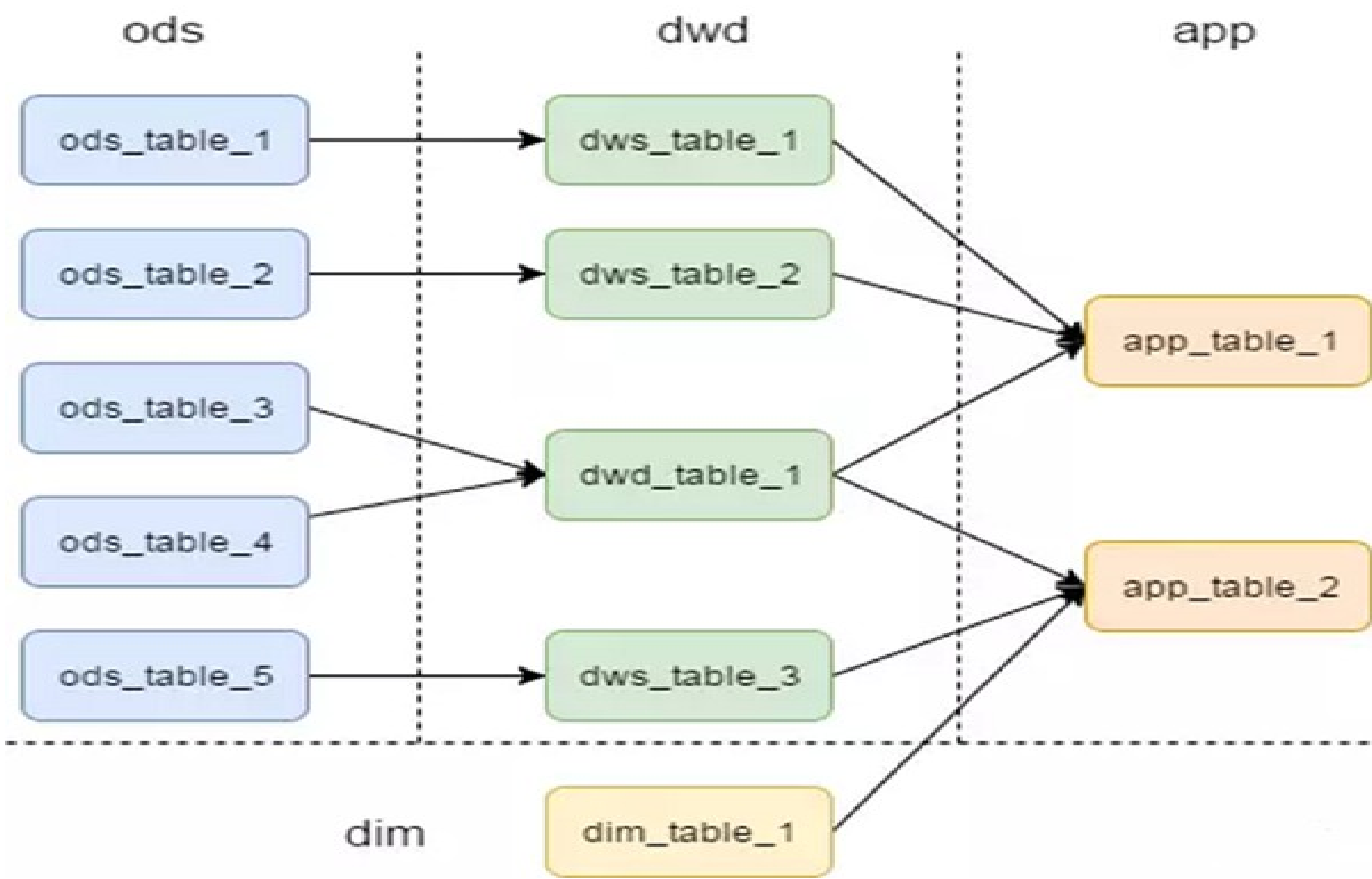
整体设计：数据架构（DA）

定义：

它涉及数据的组织、存储、处理和管理。用于开发、运营人员、新员工培训。

分层管理：

- ODS 层：原始数据，如从订单交易数据、用户数据、物流数据。
- DWD 层：将 ODS 层的数据进行清洗、去重和基本转换，详细的订单事实表、用户事实表、物流事实表等。
- DIM 层：存储产品维度表、时间维度表、用户维度表，提供详细描述性信息。
- APP 层：根据业务需求，将 DWD 层的数据进行汇总和加工，生成销售报表



整体设计：技术架构（TA）

定义：

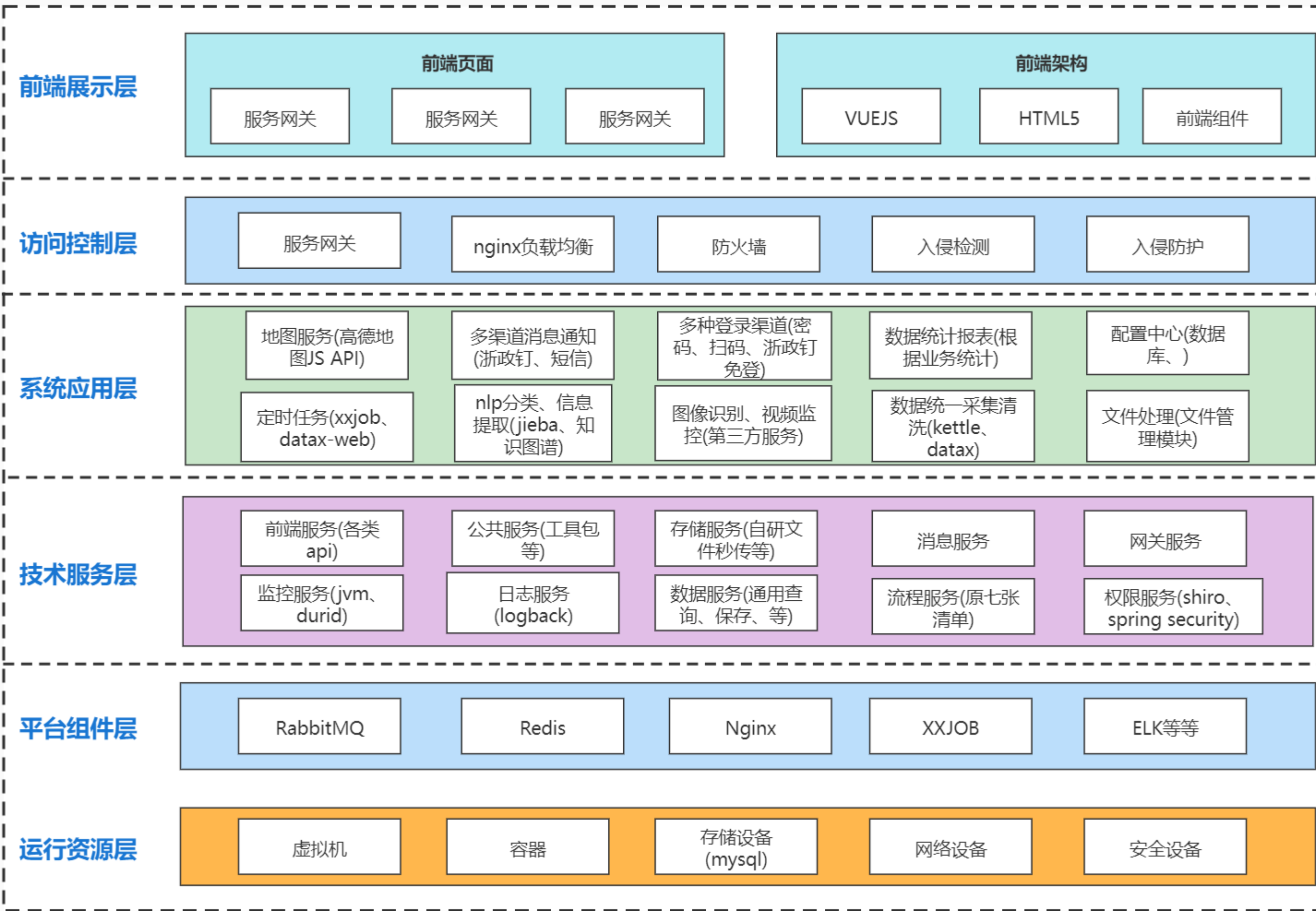
展现一个系统的技术选型、组件、模块其相互关系的图示

使用场景：

- 1. 为开发团队提供明确的参考
- 2. 运维与优化
- 3. 新员工培训

画图技巧：

- 1. 分层次展示



CSDN @技术写轮

整体设计：整体设计的原则

	隔离/拆分	抽象	复用	集成	治理
业务BA	1. 业务域拆分 2. 商品/交易/履约/支付 3. 核心和非核心		1. 基础业务下沉 2. 交易、用户	1. 业务域异步集成	1. 业务边界
应用AA	1. 按照业务域拆分 2. 模块拆分 3. 在线离线拆分 4. 核心和非核心	1. 服务抽象	1. 基础服务复用	1. 易变依赖稳定 2. 非核心依赖核心	1. 服务治理 2. SLA、水平扩展 3. 限流/降级/熔断
数据DA	1. 续写分离 2. 业务域分离 3. 冷热分离	1. 数据库抽象 2. 逻辑数据库		1. 数据库要通过服务访问	1. 备份 2. 数据质量
技术TA	1. 功能,非功能性 2. 域名、子域	1. 容器抽象	1. 框架、组件 2. 代码		1. 灰度、回滚 2. 灾备、监控

目录

- ❑ 1. 整体性设计
- ❑ 2. 接口设计
- ❑ 3. 存储设计
- ❑ 4. 缓存设计
- ❑ 5. 异步设计
- ❑ 6. 非功能性设计

接口设计：尽量使用 REST 风格的 API

URI 设计

- **Base URL:** `https://example.domain.com/api`
- 分组：`group` 用于对 API 进行分组，通常表示功能模块或业务领域。
- 版本：`version` 用于版本控制，如 `v1`, `v2`，便于 API 的演进和向后兼容。
- 资源：`resource` 表示具体的资源，如 `users`, `orders`

资源命名

区分单复数资源清晰直观。

- `/api/v1/users` (订单集合)
- `/api/v1/user`

使用 HTTP 方法来表示操作。

- 获取所有用户：`GET /api/v1/users`
- 获取特定用户：`GET /api/v1/users/{id}`
- 创建新用户：`POST /api/v1/users`
- 更新用户：`PUT /api/v1/users/{id}`
- 删除用户：`DELETE /api/v1/users/{id}`

其它规则

- 应使用连字符 (-) 来提高 URI 的可读性
- 路径中避免使用动词，用名词
- 不得在 URI 中使用下划线 (_)
- URI 路径中全都使用小写字母

接口设计：统一错误码风格

1. 始终返回 **200 http status code**，**Body** 中包含错误信息

如：Facebook API 的错误 Code 设计，始终返回 200 http status code：

```
{
  "error": {
    "message": "Syntax error \"Field picture specified more than once. This is only possible before version 2.1\" at character 23: id,name,picture,picture",
    "type": "OAuthException",
    "code": 2500,
    "fbtrace_id": "xxxxxxxxxxxx"
  }
}
```

2. 返回合适的 **HTTP Code**，并在 **Body** 中返回详尽的错误信息

HTTP/1.1 400

```
{
  "code": 100101,
  "message": "Database error",
  "reference":
  "https://github.com/xx/tree/master/docs/guide/faq/xxxx"
}
```

业务错误码

1. 不同部分代表不同的服务
2. 不同部分代表不同的模块
3. 不同的品类

例如：**1000100**

- 模块：**1000**

- 详细错误码：**100**

使用常用的 httpstatus code

- 200 - 表示请求成功执行
- 400 - 表示客户端出问题
- 500 - 表示服务端出问题

接口设计：幂等性设计（All write API）

幂等性保证多次调用对资源的影响是一致的

幂等性

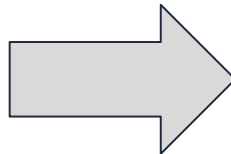
- 1. 网络不稳定，接口超时重试
- 2. 用户错误操作带来的重试
- 3. MQ 消息重复消费

业务场景：

- 1. 用户商品上传
- 2. 订单创建、支付等
- 3. 消费 MQ，异步发货

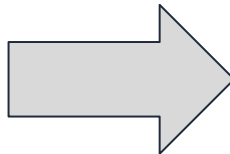
源头 控制请求

- 提交动作按钮置灰
- 及时重定向页面
- 仅作为辅助方案



过程中 过滤重复请求

- 幂等键：请求参数、设计 token
- 幂等表、去重表



根本上 过滤重复请求

- 唯一约束、唯一键
- 乐观锁、状态机，version
- 悲观锁

接口设计：接口时序图

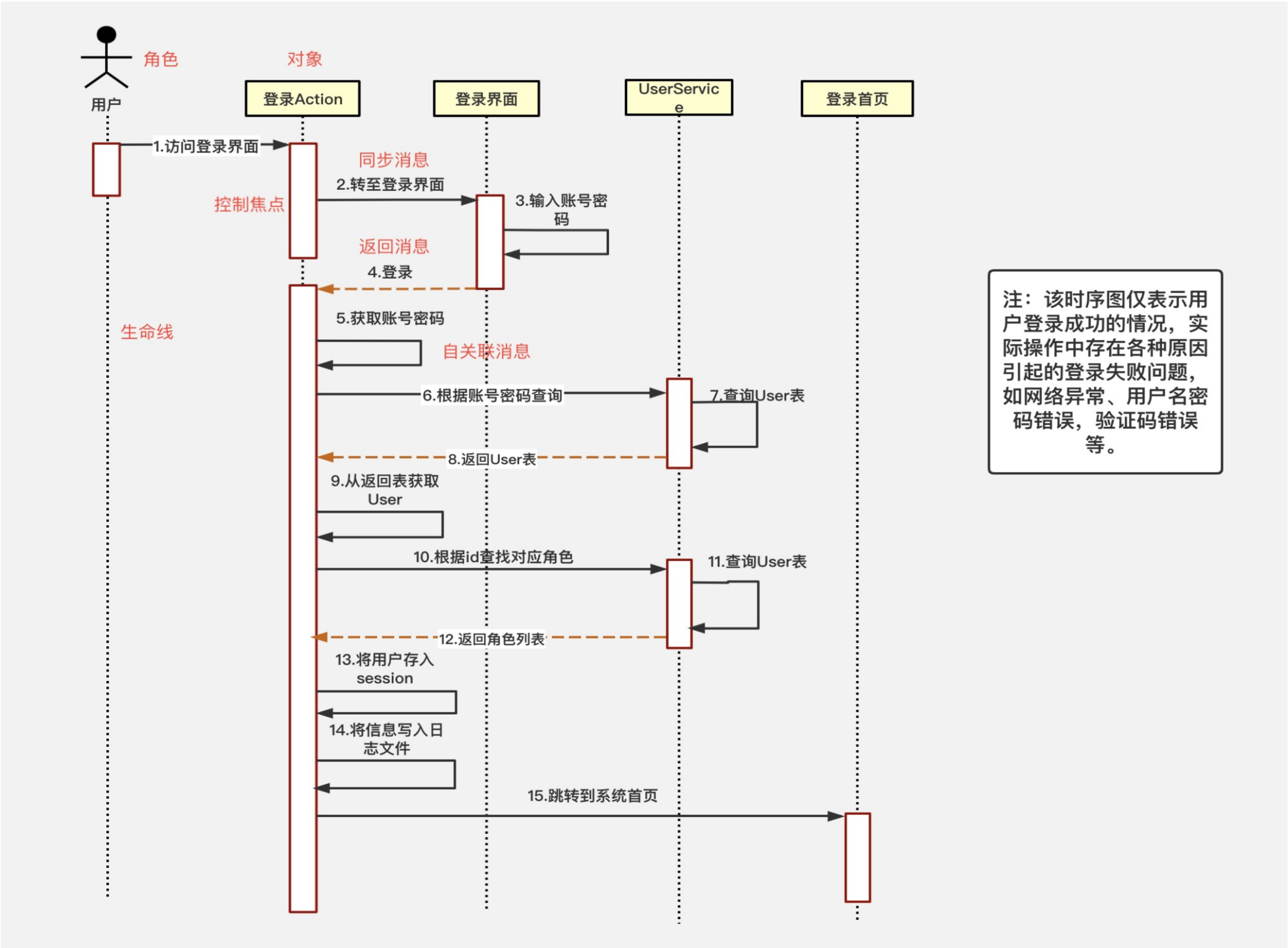
定义：

接口时序图 (Sequence Diagram) 是一种 UML 图，用于展示对象之间的交互以及这些交互按时间顺序发生的顺序。时序图能够帮助开发者理解和设计系统中各个组件之间的通信过程。

画图工具：

PlantUML：一个开源工具，通过简单的文本描述来生成 UML 图，包括时序图

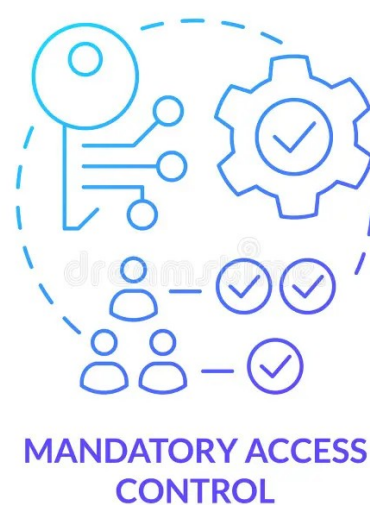
Draw.io：一个免费的在线图表绘制工具，易于使用。



接口设计：安全性设计

访问控制

- 用户鉴权
- 用户授权
- appkey
- 黑白名单机制



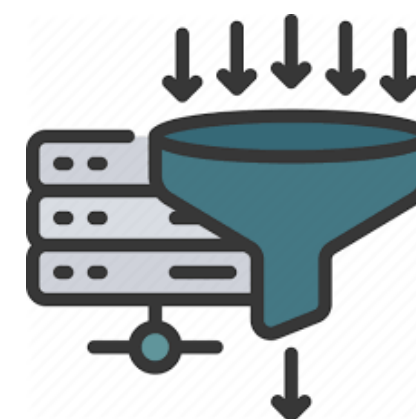
数据保护

- 输入参数校验
- 输出敏感信息
- 加密、https



异常处理

- 限流策略
- 熔断策略
- 降级策略
- 隔离
- 超时重试



可观测性

- 监控告警
- 审计日志
- tracing



目录

- ❑ 1. 整体性设计
- ❑ 2. 接口设计
- ❑ 3. 存储设计
- ❑ 4. 缓存设计
- ❑ 5. 异步设计
- ❑ 6. 非功能性设计

存储设计： 如何选择合适的存储组件

SQL

✓ACID

✓SQL

✗水平扩展性（数据量）

✗可用性（相对）

NoSQL

✗ACID

✗SQL

✓水平扩展性（数据量）

✓可用性（相对）

NewSQL

✓ACID

✓SQL

✓水平扩展性（数据量）

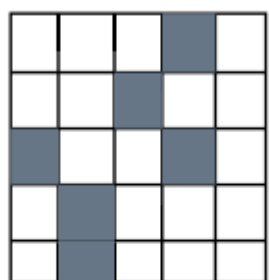
✓可用性（相对）



Relational



Document



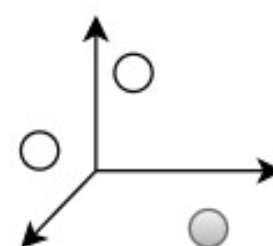
Column-Family



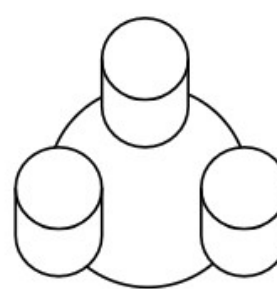
Time-series



Graph



Vector DB



Distributed SQL

常用于构建交易系统,ERP, CRM 财务系统等。如 MySQL PostgreSQL

用于存储、检索和管理半结构化数, MongoDB, Elastic Search

用于列很多、稀疏数据。例如广告、金融数据分析。 Hbase, Cassandra

用于存储和查询和时间序列相关的数据。监控、物联网。例如 Prometheus, InfluxDB

在社交网络、推荐引擎和欺诈检测等。如 Neo4j, Neptune

用于存储、查询和操作高维向量数据,主要用于 AI、NLP、推荐等。如 Milvus, FAISS。

结合 RDBMS 的 ACID 和 NoSQL 的扩展性和性能的新型数据库系统。主要应用大型金融、电子商务系统。如 TiDB、Google Spanner 等

存储设计：关系型数据建模（data modeling）

概念模型：

- 描述业务需求，业务规则
- 组成部分：描述关键实体、属性、关系

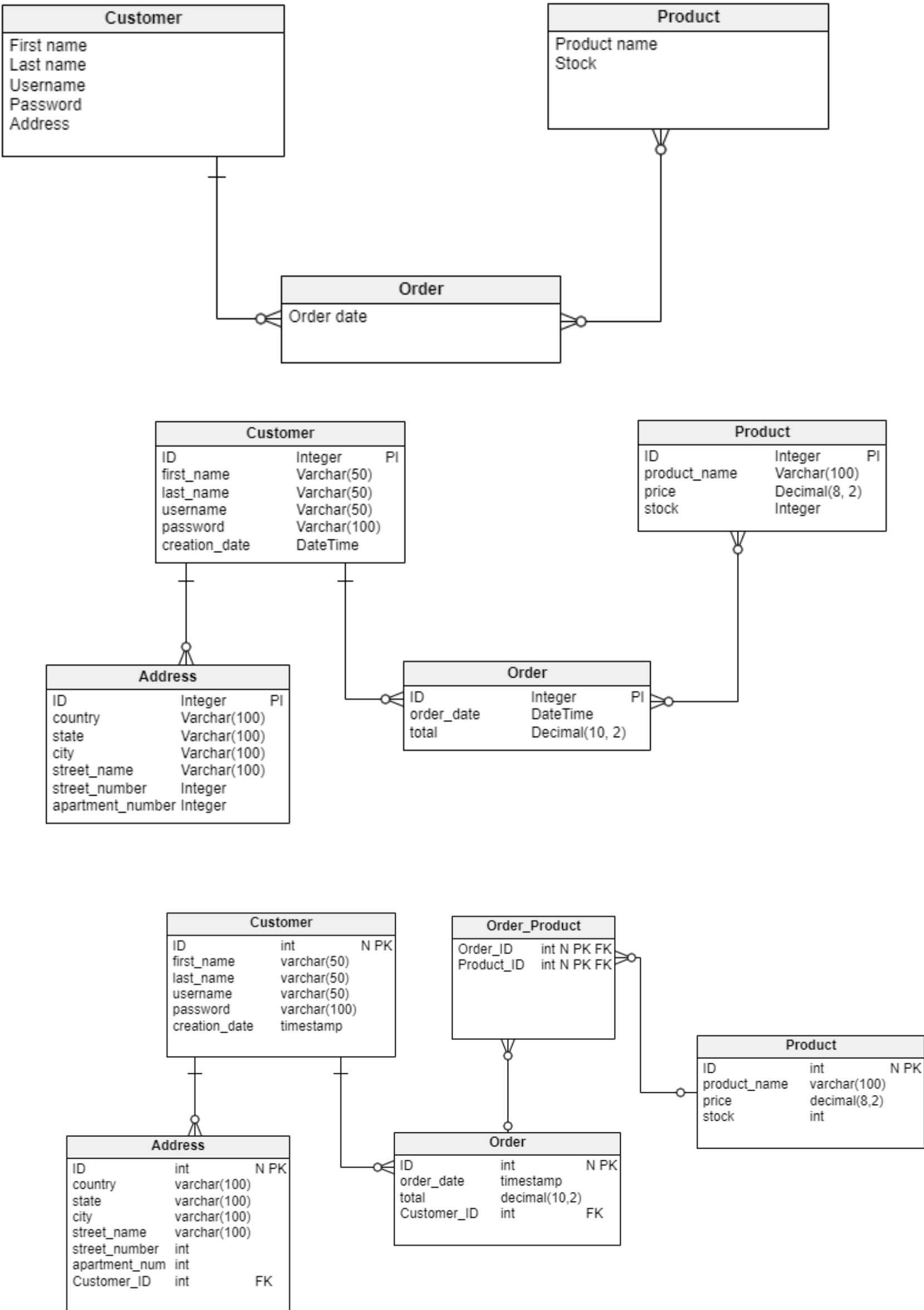
逻辑模型：

- 具体化概念模型
- 规范化、3NF
- 组成：表、数据类型

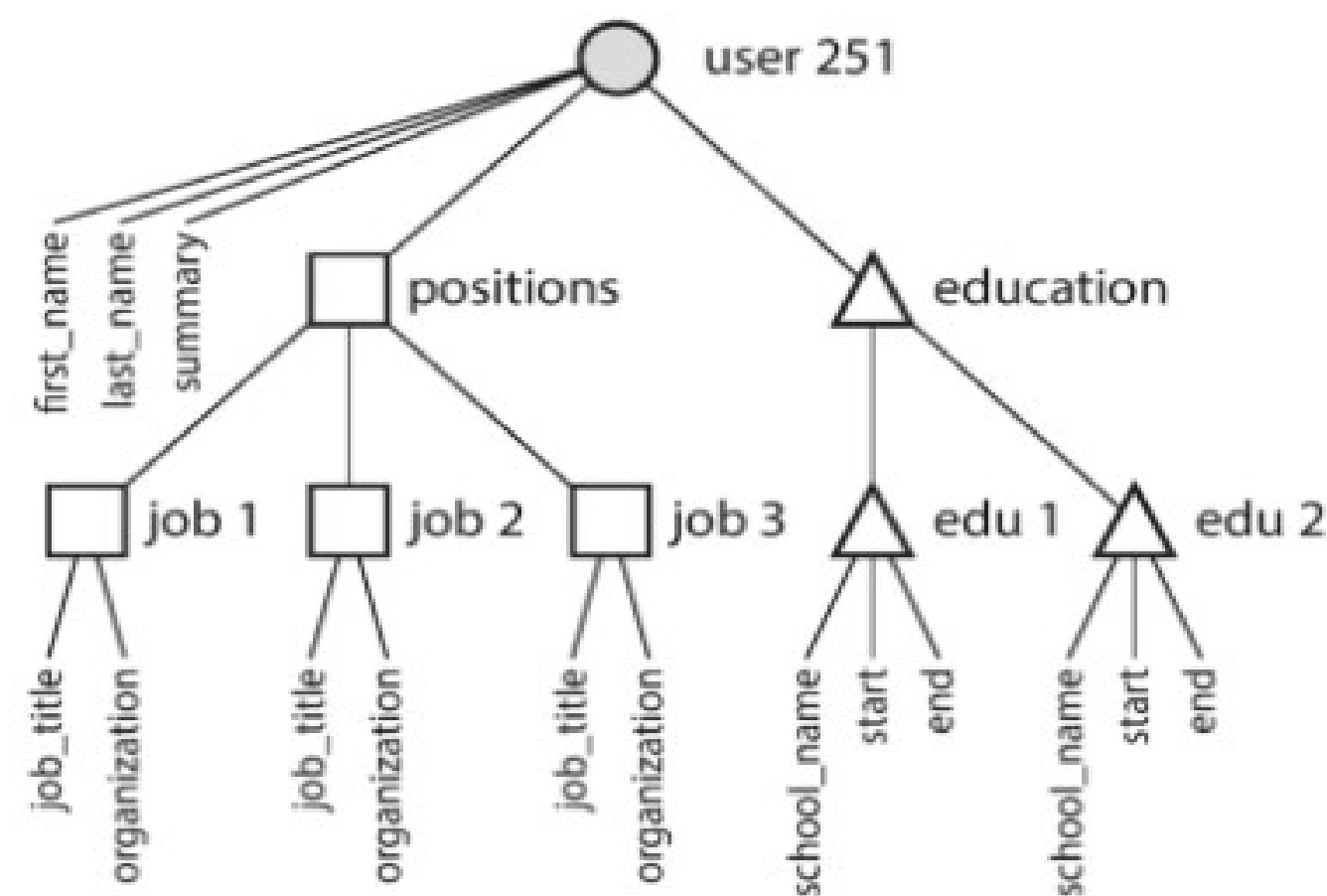
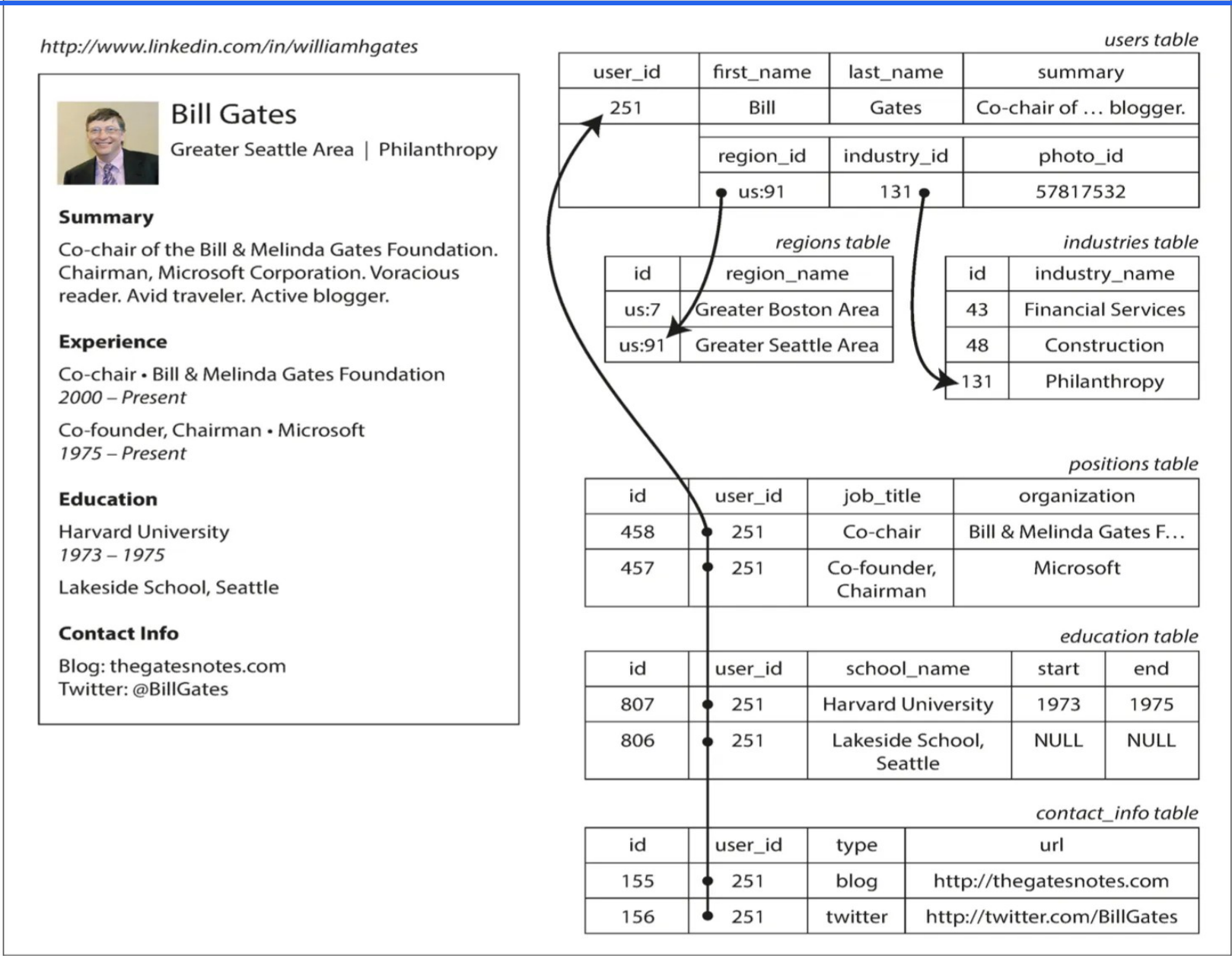
物理模型：

- 存储细节，分区 sharding，分表
- 存储引擎、列格式、编码方式
- 主键、外键、约束条件、索引

逐级深化



存储设计：关系型 vs 文档型



	文档模型	关系模型
事务支持	不支持	可以事务性，支持 ACID 属性
数据灵活性	弱 schema，支持动态增加字段	强 schema，修改 代价很大
对应关系 一对多，多对一	天然的一对多 树形嵌套关系	通过外键 Join
代码简化	数据具有文档结构则文档模型天然合适，	主键，索引，条件过滤
Join 支持	对 Join 支持的不太好	支持
访问局部性	一次性访问整个文档	分散在多个表中

存储设计：关系型 vs 列示存储

业务模型

	ad_id1	ad_id2	ad_id3	...
000001	0.8		0.5	
000002		0.9		
000003	0.1		0.6	
....				

关系存储

userid	adid	data
000001	ad_id1	0.8
000003	ad_id1	0.1
000002	ad_id2	0.9
000001	ad_id3	0.5
000003	ad_id3	0.6

列示存储

rowkey	qualifier	data
000001	ad_id1	0.8
	ad_id3	0.5
000002	ad_id2	0.9
000003	ad_id1	0.1
	ad_id3	0.6

什么情况考虑列存储？

- 列是不固定的
- 列更新较多，不是行更新
- 行数据稀疏
- 数据量很大

	数据模型	扩展性 (数据量)	数据模型 灵活性	适用场景
关系型存储	表格	复杂	差	复杂查询
列示存储	表格 (按照列存储)	高度水平扩展	列不固定	简单模型、数据量大

存储设计：数据分片 sharding 策略

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

什么时候考虑分表、分库、分实例?

- 单实例无法存储数据
- 单实例无法支持读写请求
- 业务隔离、分库
- 单表数量 >500w (经验值)
- 垂直 sharding , 水平 sharding

	映射关系 维护难度	节点增删数据迁移	数据动态平衡
hash 取模	简单	数据迁移大	无
范围 range based	难	可以控制数据迁移的范围	基于数据快 是可以做到 mongodb
一致性 hash	中	需要迁移的数据较少，所有节点都能贡献部分数据	可以

存储设计：副本策略

副本策略用于提高可用性和性能

1. 单领导者，1 个或者多个 slave

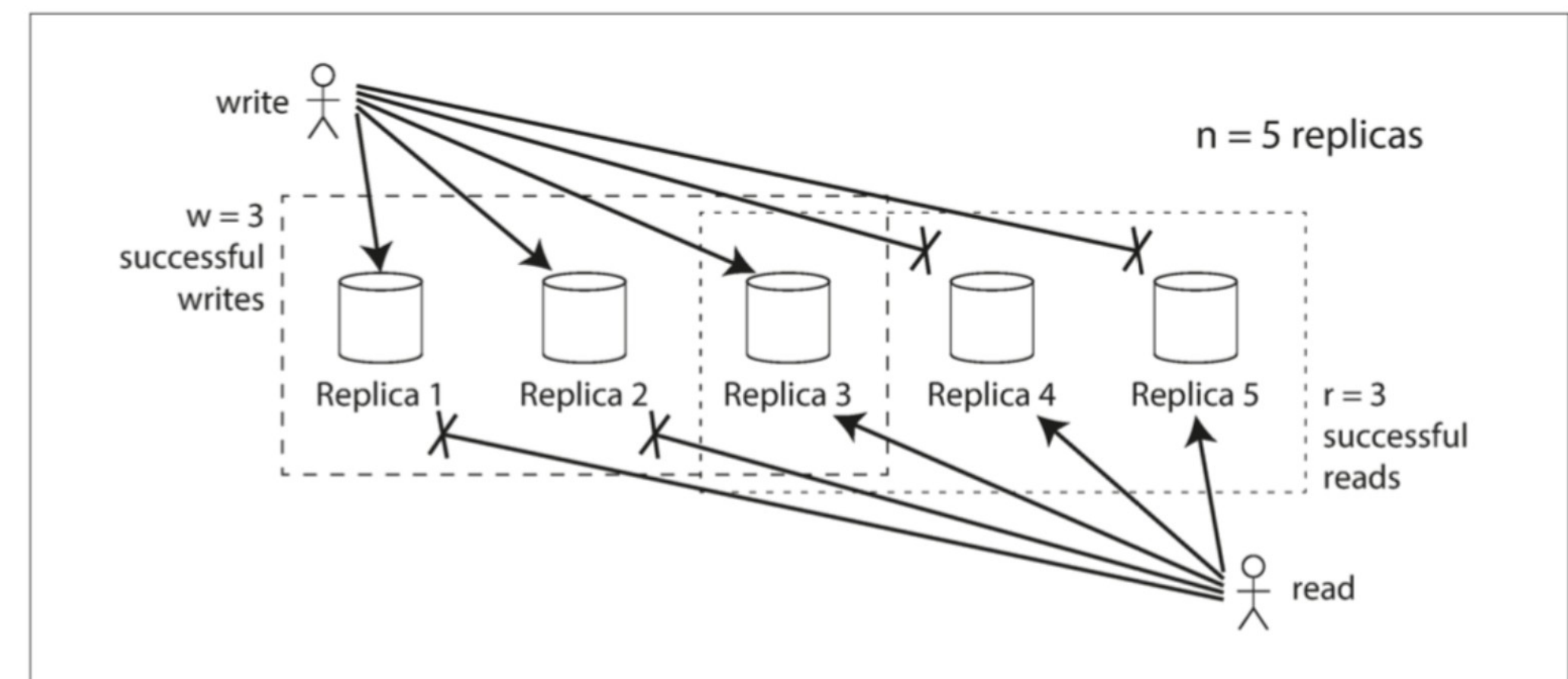
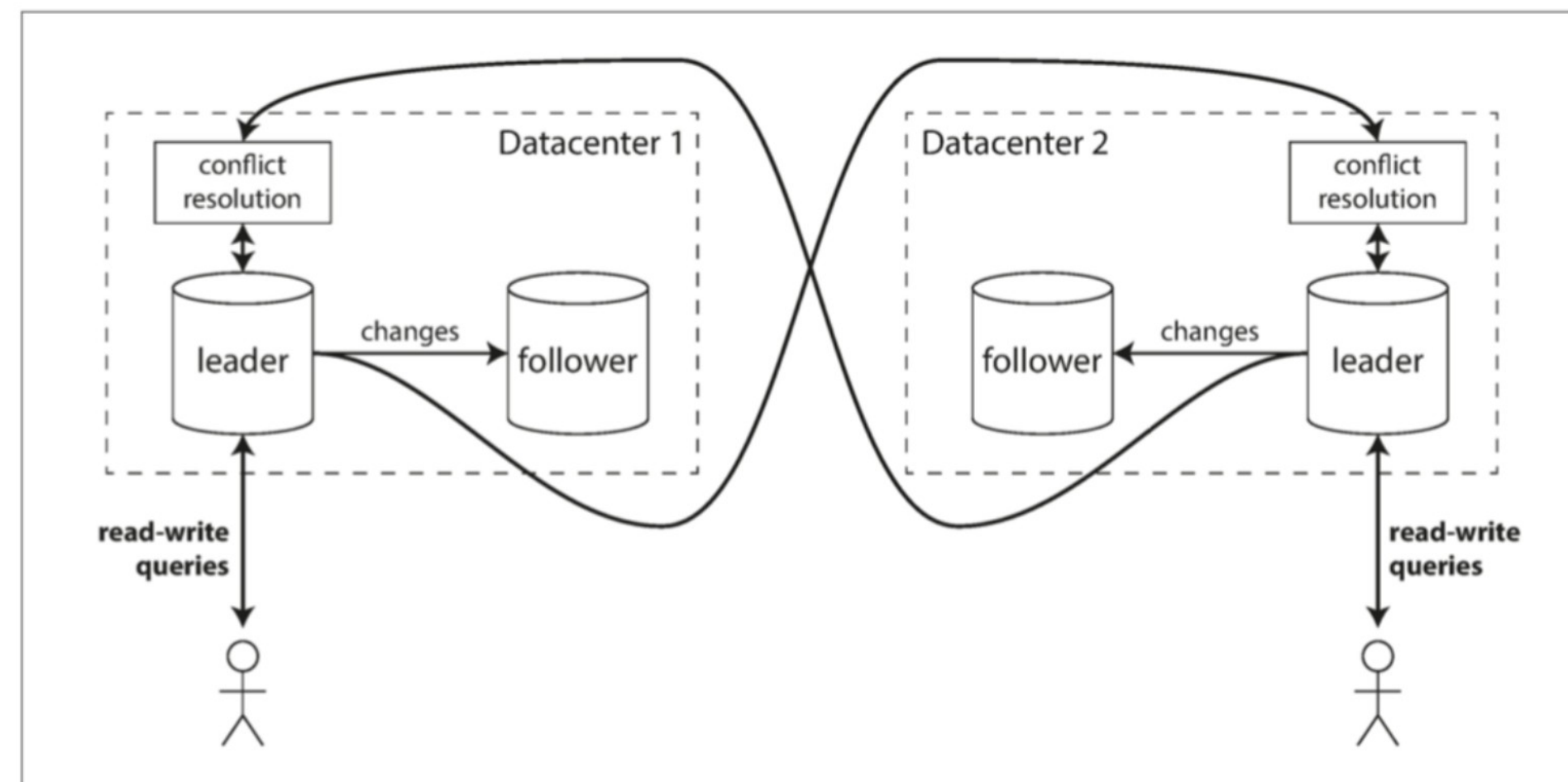
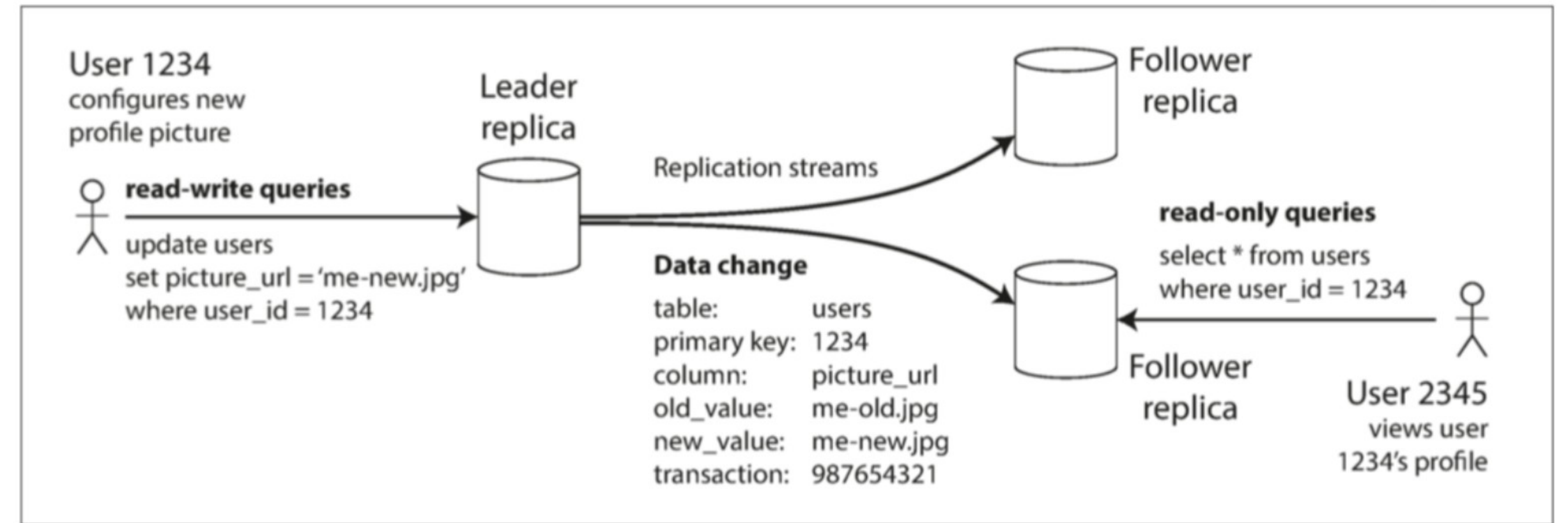
- 异步复制、半同步复制、同步复制
- 常见故障，主库失效、从库失效、复制的延时的影响

2. 多主复制

- 用于多数据中心多活、离线客户端等场景
- 避免冲突、冲突合并、解决冲突

3. 无主模式

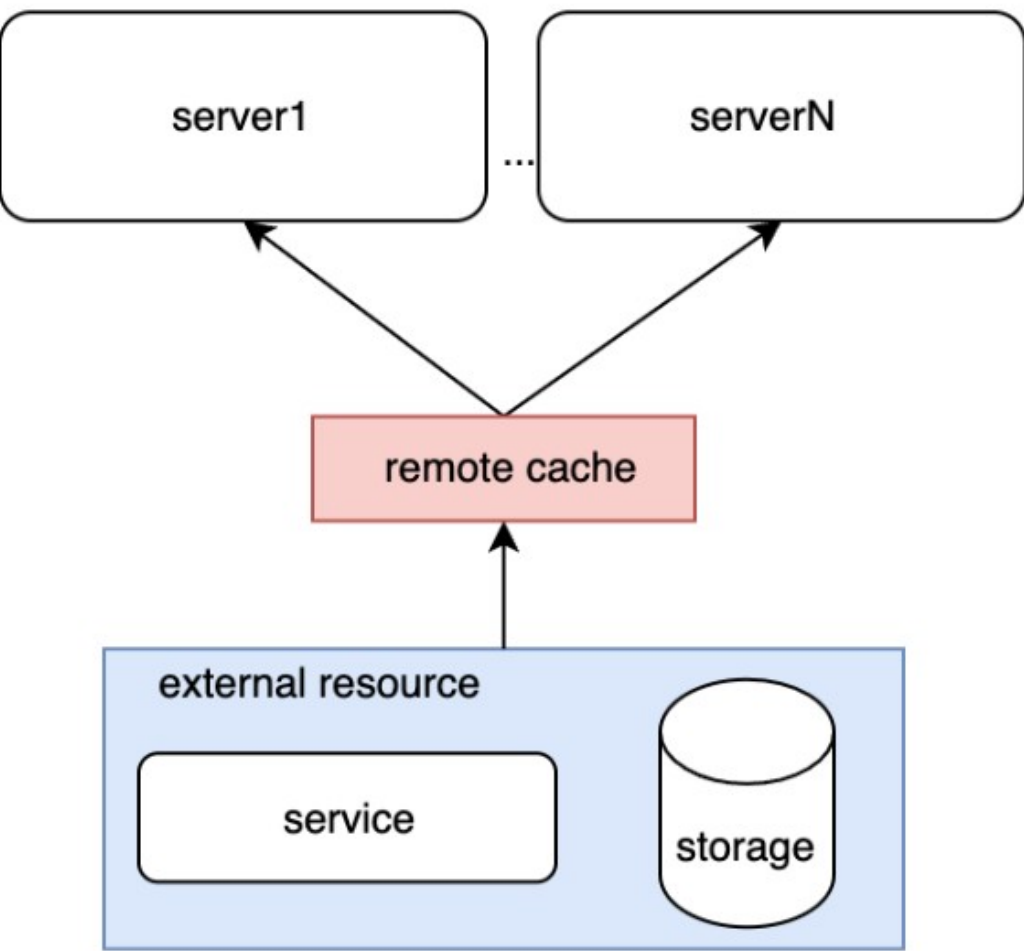
- 允许任何副本接受写请求，客户端将每个写入发送到几个节点，通过配置法定人数判决数据一致性，例如 dynamo



目录

- ❑ 1. 整体性设计
- ❑ 2. 接口设计
- ❑ 3. 存储设计
- ❑ 4. 缓存设计
- ❑ 5. 异步设计
- ❑ 6. 非功能性设计

缓存设计 - Remote、Local、Multilevel Cache



优点:

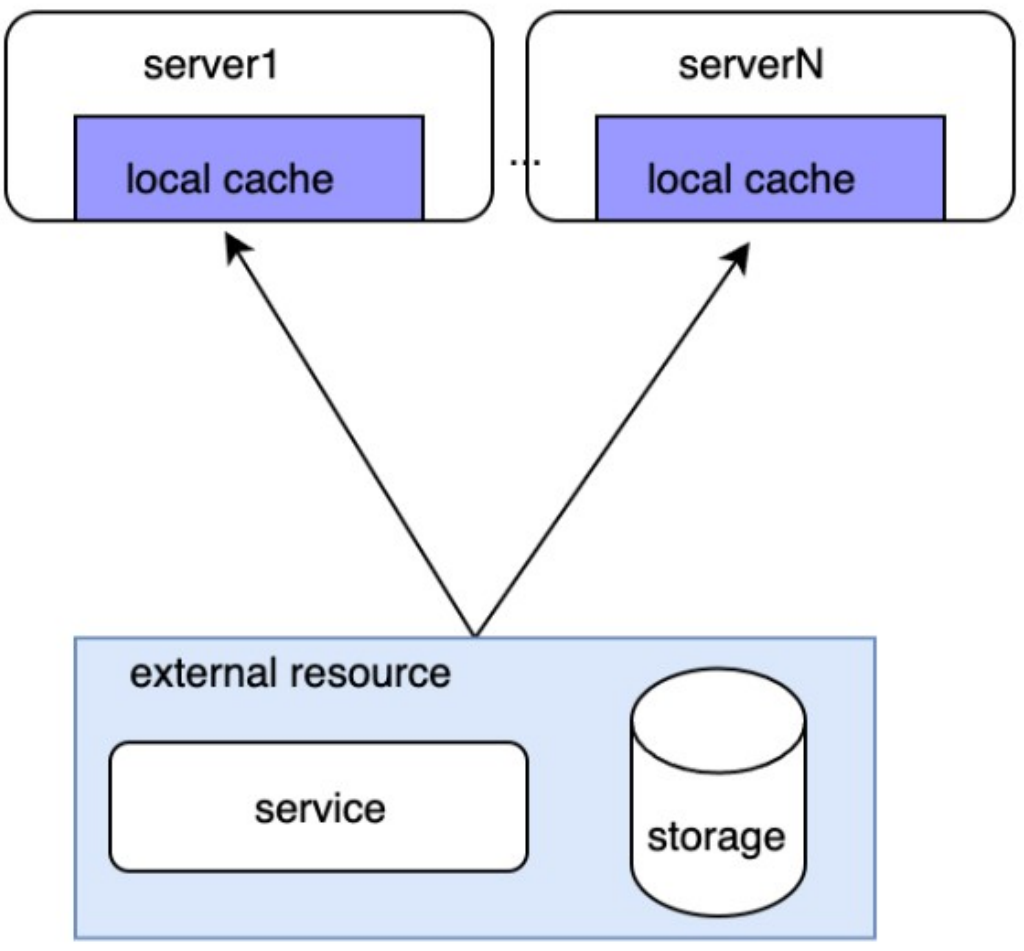
- 多个实例共享缓存，一致性好
- 简单，缓存服务器扩展

缺点:

- 网络延迟

使用场景:

- 缓存最应优先考虑的场景



优点:

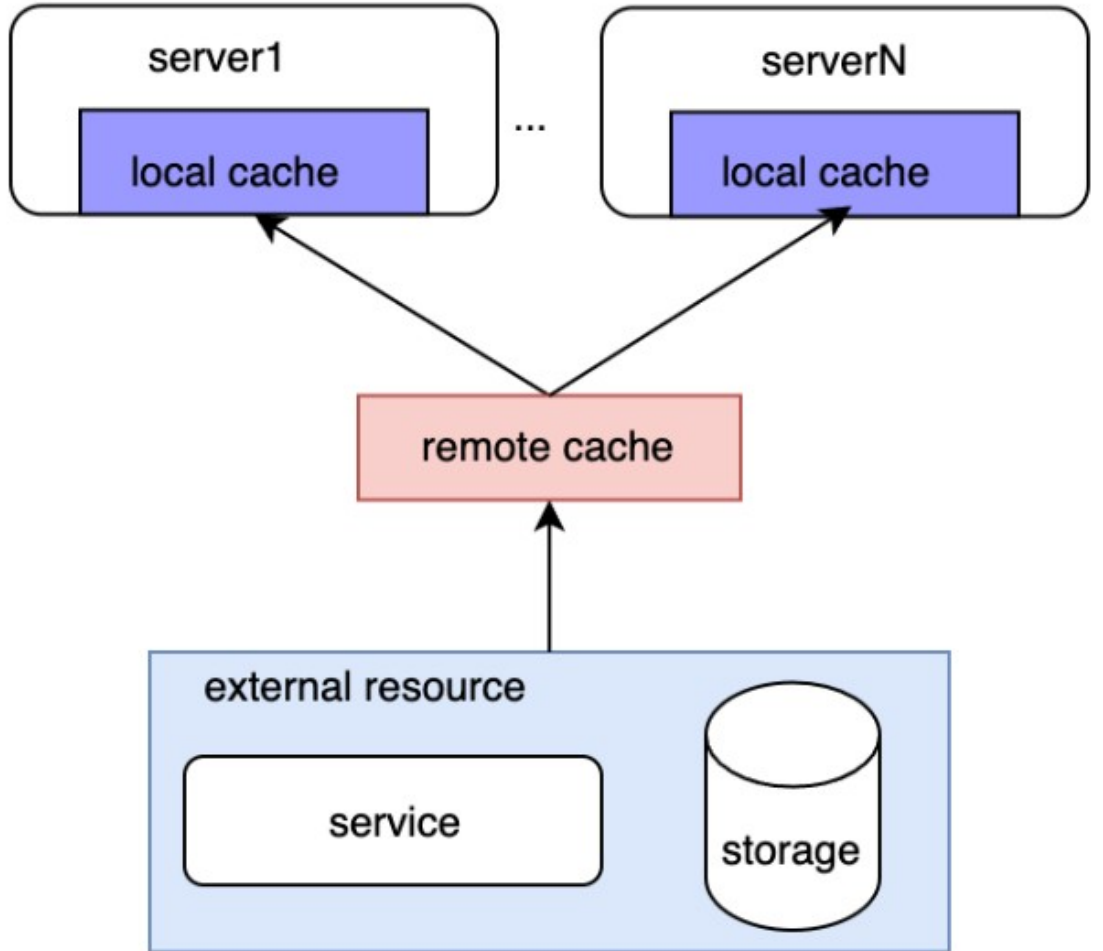
- 极低的访问延时

缺点:

- 资源使用大、单实例的内存容量
- 不同实例之间数据不完全一致

使用场景:

- 访问量极高、延时要求极高
- 数据读多，写（更新）少



优点:

- 结合了 local 和 remote 的优点

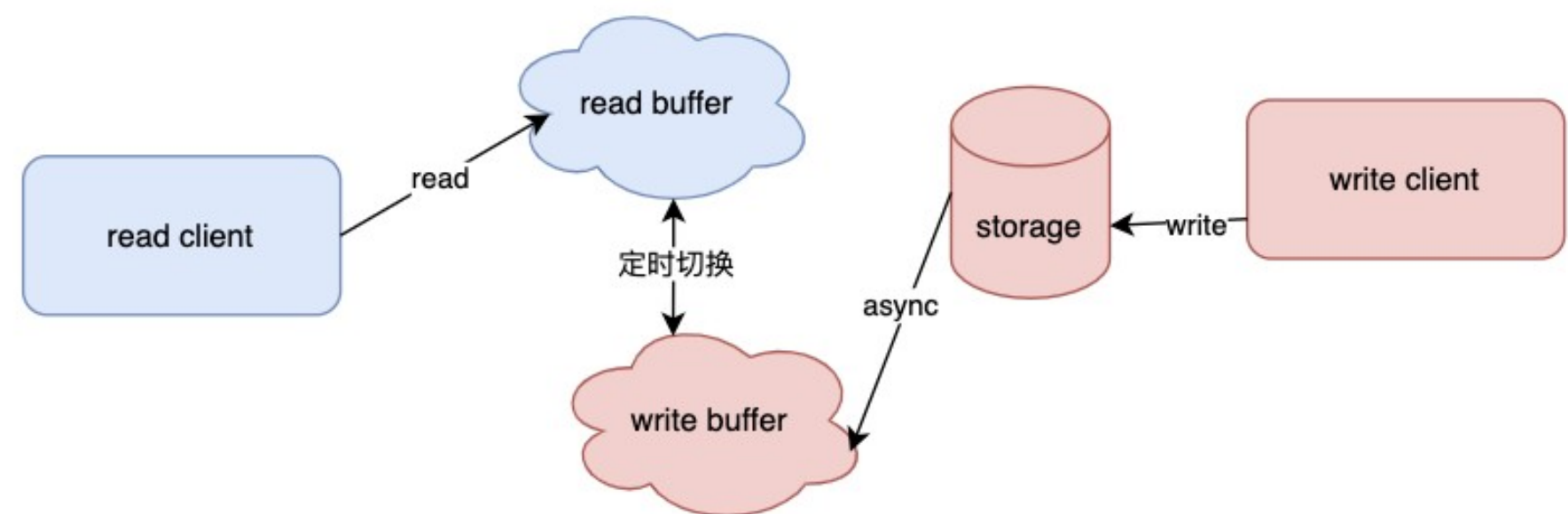
缺点:

- 复杂，数据更新策略复杂，增加了数据不一致性

使用场景:

- 数据量大，存在明显的热点数据，例如新闻 feed 流

缓存设计 - 双 buffer vs LRU



优点：

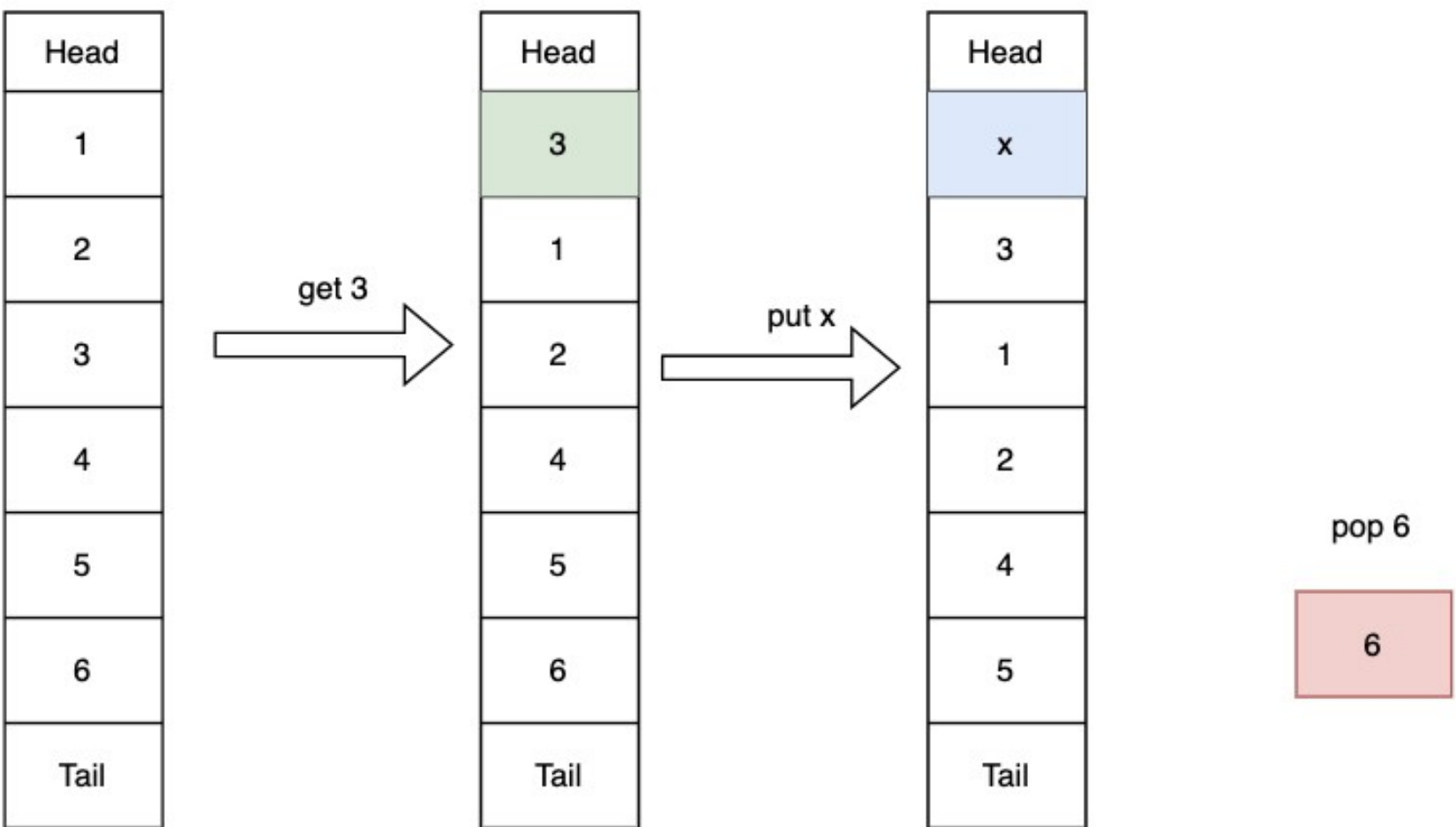
- 通过提前准备数据，减少处理时的延迟
- 读写分离，性能好

缺点：

- 两份内存开销

使用场景：

低延时，数据量不太大，对数据一致性要求不高的场景



优点：

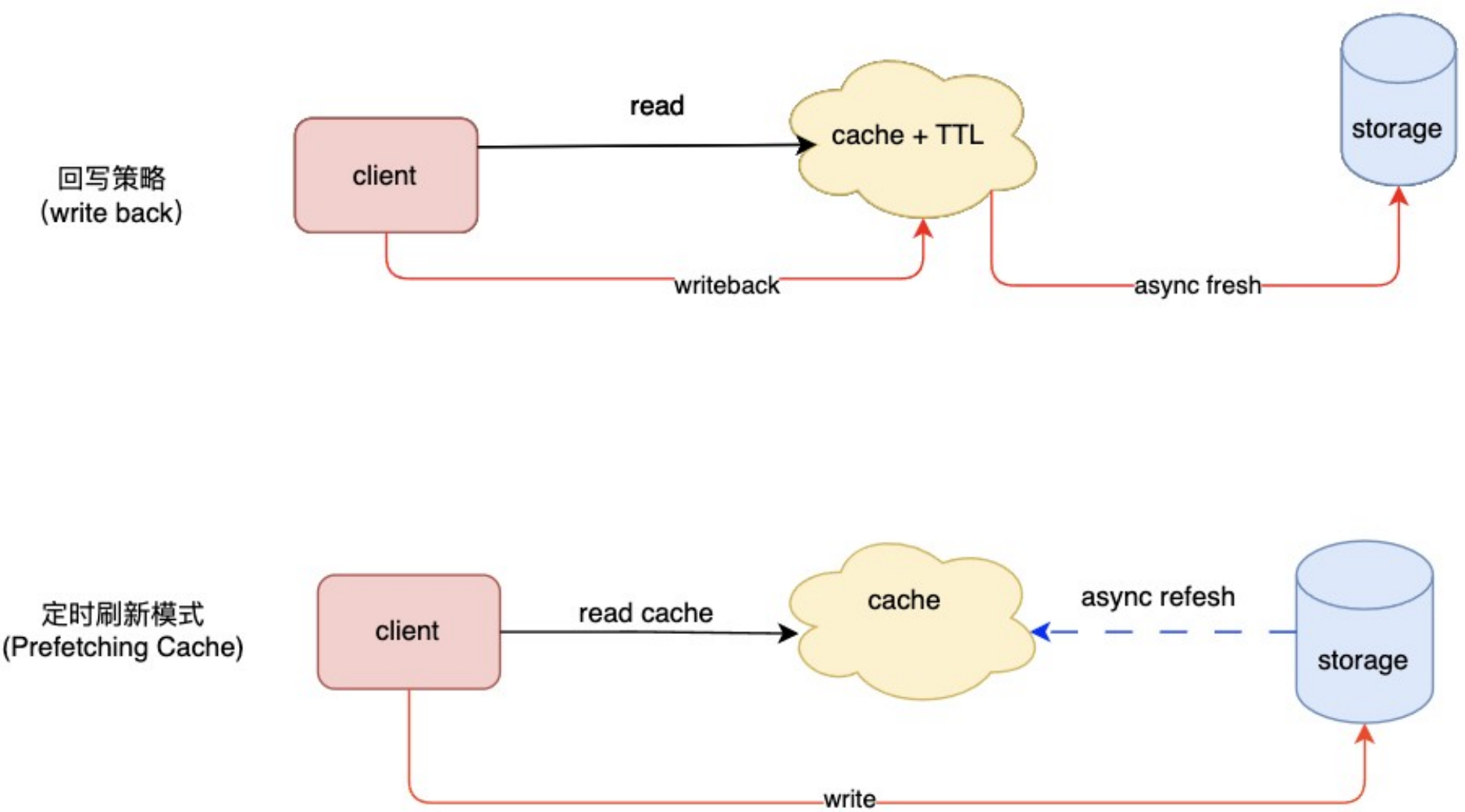
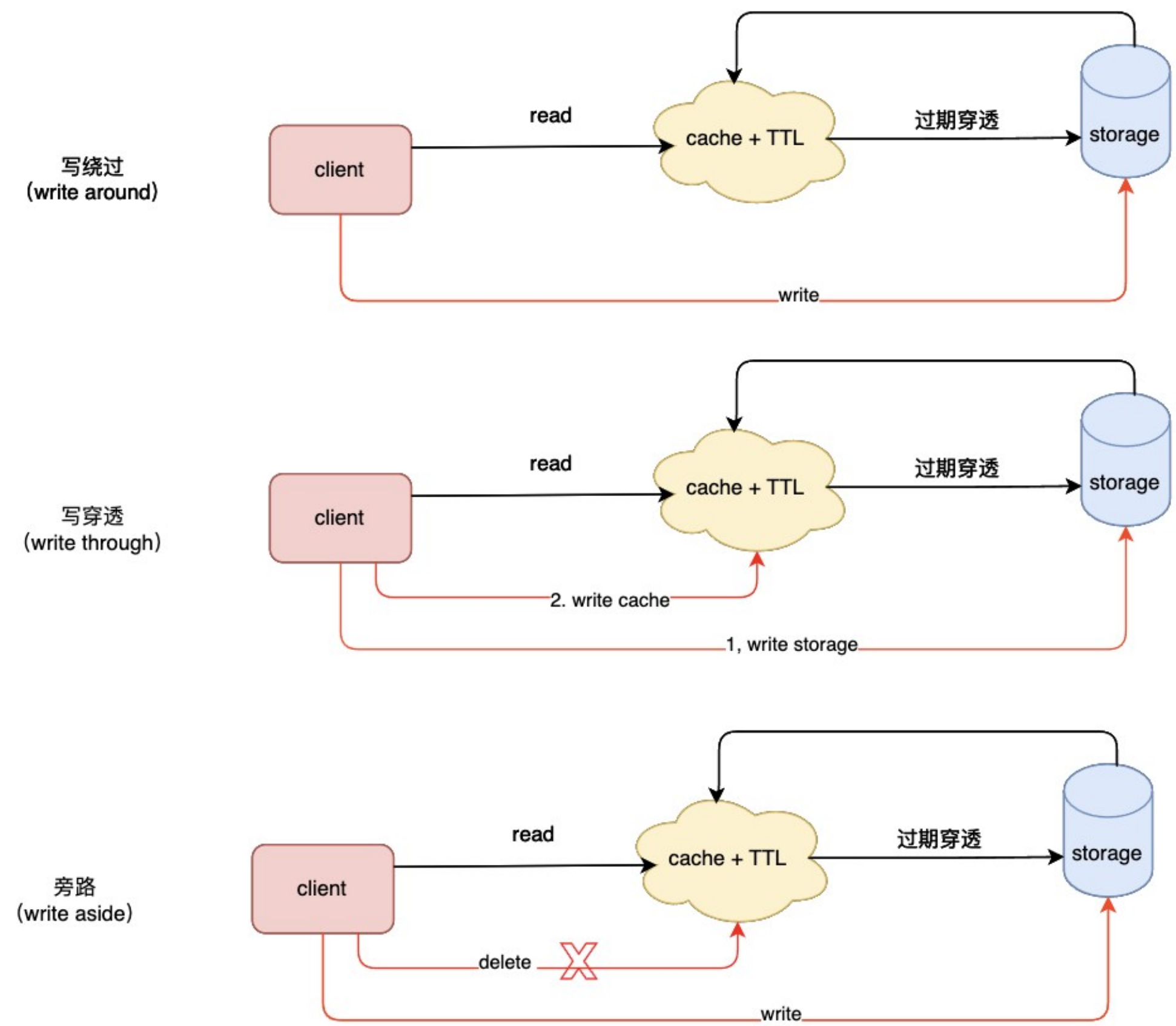
- 内存开销小

缺点：

- 并发写加锁，影响读
- 缓存命中率受 size 和热点数据影响

场景：数据量大，比较明显的热点数据的场景

缓存设计 - 读写模式的选择



- 写请求能不能直接更新存储？
- 写请求更新存储后，要不要更新 cache? update 还是 delete
- 读请求要不要回源到存储？
- TTL 设置多少？

缓存设计 - 安全性设计

1. 资源评估指标监控

- kv size 、 key 、 存储量预估
- 读写峰值预估
- 命中率监控
- TTL 过期 (分钟级别、小时级别)
- 设计具有唯一性且可读性的 key

3. 后端存储保护

- 缓存击穿、缓存雪崩、缓存穿透
- 访问存储限频率、分布式锁
- TTL+random(x)
- 缓存记录不存在 key 、 bloomfilter 等

2. 热 key 和大 key 问题?

Hot Key :

- 创建多个副本，分散不同的节点上
- 本地缓存

Big Key:

- 缓存结构是否合理
- 数据压缩、数据拆分

4. 缓存一致性

- 业务对缓存与存储的不一致的容忍度
- 合理设置缓存的失效时间
- 缓存刷新机制，双写、失败、并发控制等

目录

- ❑ 1. 整体性设计
- ❑ 2. 接口设计
- ❑ 3. 存储设计
- ❑ 4. 缓存设计
- ❑ 5. 异步设计
- ❑ 6. 非功能性设计

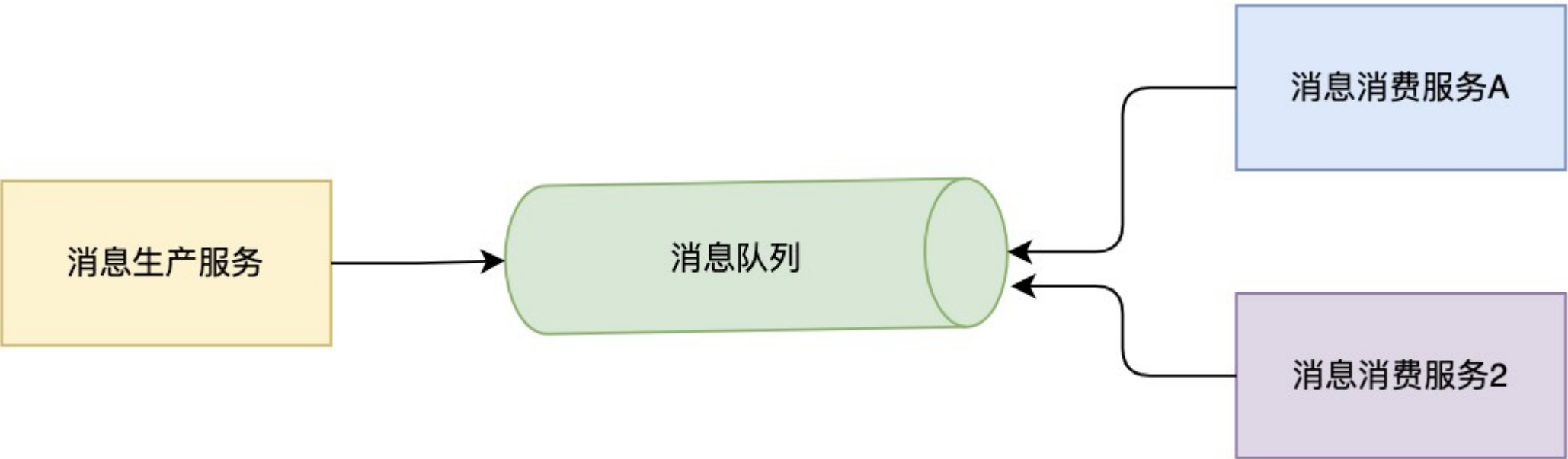
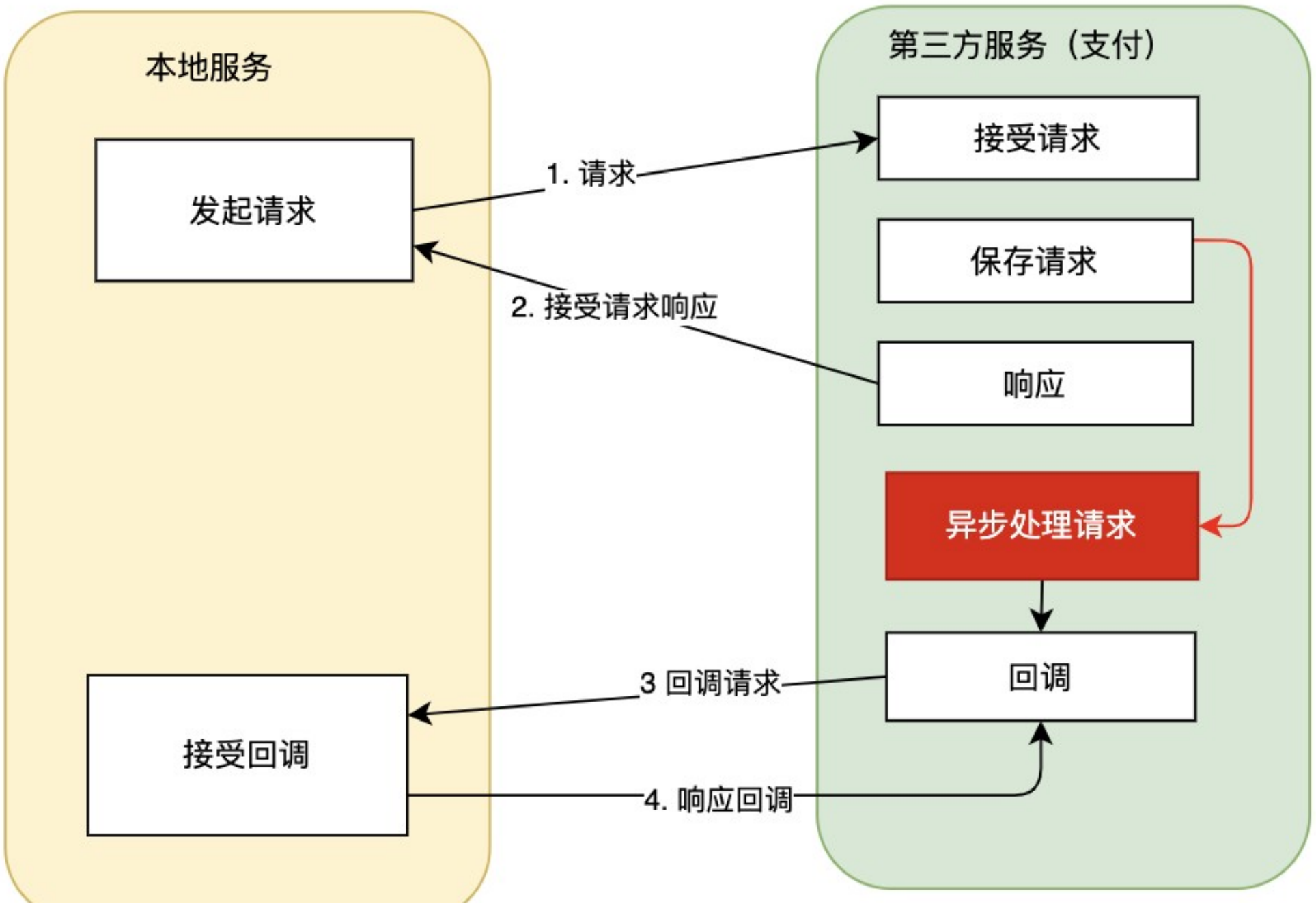
异步机制设计：异步使用场景

异步机制的优势：

- 提高系统吞吐量
- 平衡流量、消峰
- 实现服务间的松耦合和高可靠性
- 失败隔离，自我修复

使用场景：

1. 第三方服务、异步回调
 - 异步处理耗时、不需要即时处理的业务
 - 系统中容错和重试
2. 发布和多个订阅（一对多）
 - 事件驱动架构
 - 数据流式处理和实时计算
4. 定时任务调度、延时任务



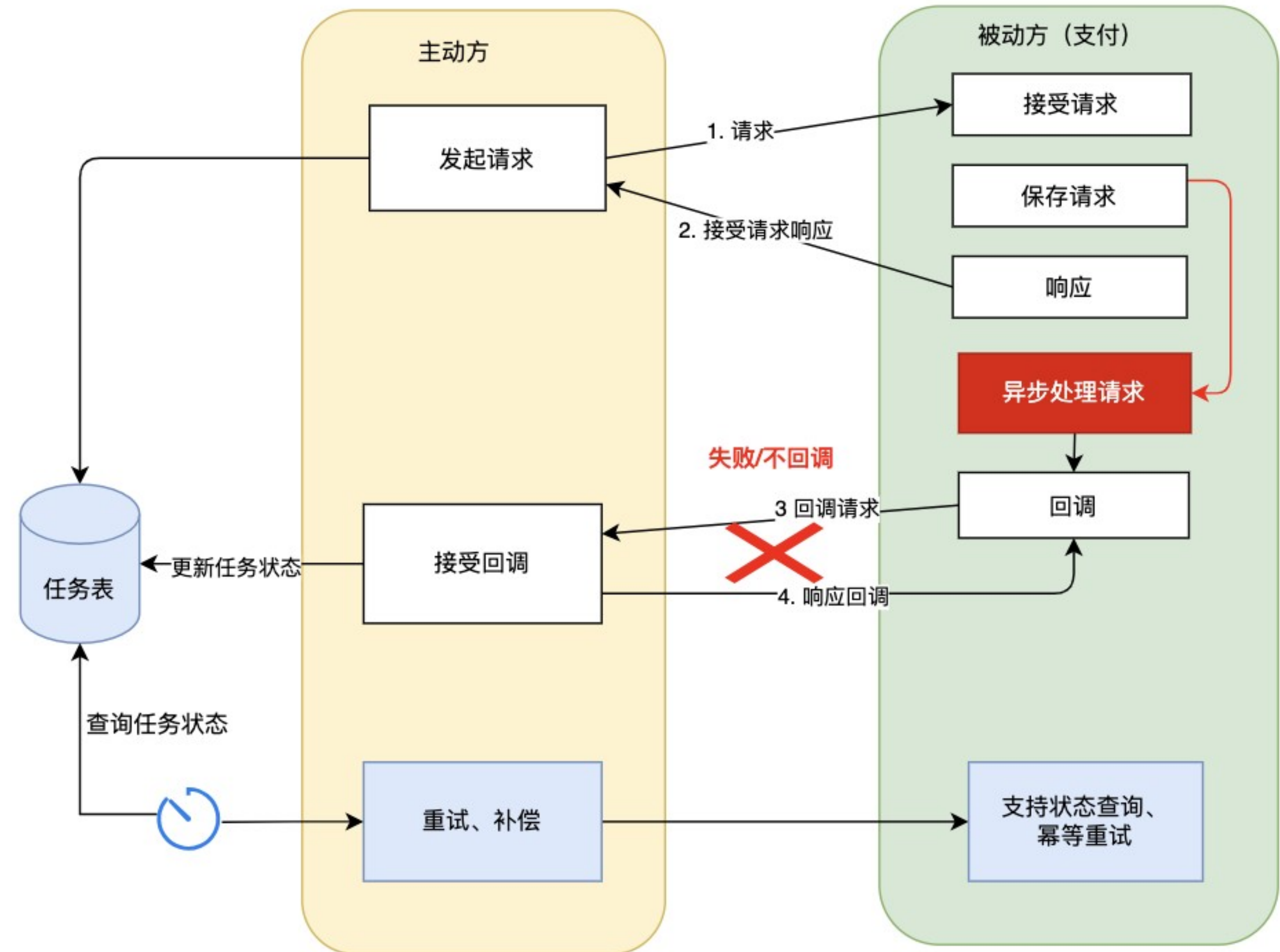
异步机制设计：任务重试补偿机制

异步机制的缺点：

- 异步操作中的错误和异常处理更加复杂，未捕获的异常可能导致任务失败或系统不稳定
- 丢消息、消息重复消费、消息乱序
- 调试、监控更加复杂

解决措施：

- 任务重试和补偿、幂等重试、最终一致性
- kafka 分区有序性、序列号等
- lag 异常监控
- 全局异常处理机制，记录日志和监控错误，通常需要统一的组件或者基类



目录

- ❑ 1. 整体性设计
- ❑ 2. 接口设计
- ❑ 3. 存储设计
- ❑ 4. 缓存设计
- ❑ 5. 异步设计
- ❑ 6. 非功能性设计

非功能性需求和权衡

用户角度：

- 安全性 (security)
- 易用性 (usability)
- 可用性 (availability)
- 性能 (performance)
- 用户体验 (user Experience)

业务角度：

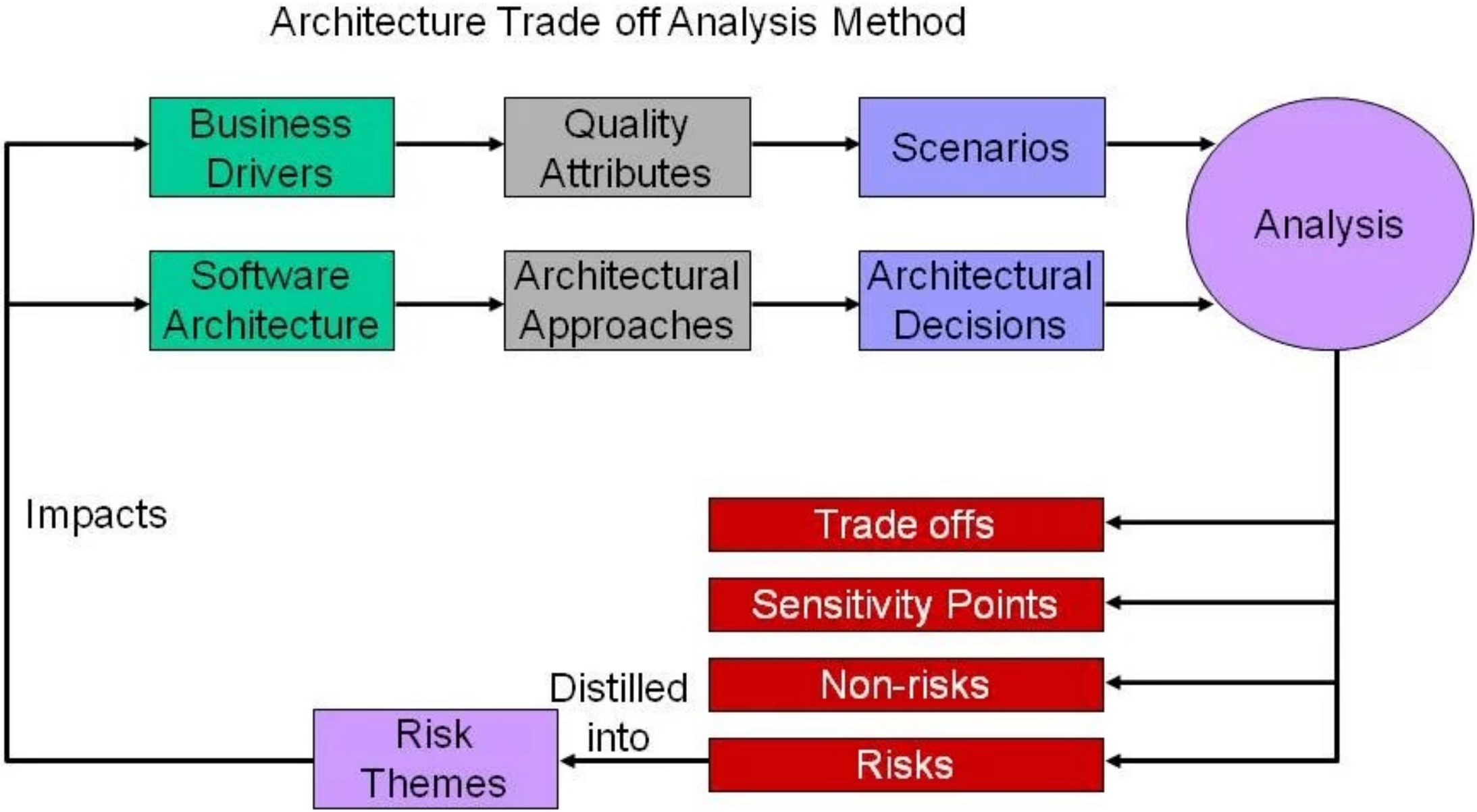
- 成本 (cost)
- 效率，时间
(time)
- 资源使用率

开发者角度：

- 可维护性 (Maintainability)
- 可扩展性 (scalability)，灵活性
- 可测试性 (Testability)
- 代码质量



非功能性需求： Trade-offs 分析方法 ATAM



资料来源

定义：以场景集合的形式描述质量要求，将架构决策映射到质量要求。输出一组已确定的敏感点 ensitivity 和权衡点 tradeoff 、一些风险 risks 和非风险 on-risks 、风险主题。

第 2 层：质量属性	第 3 层：质量属性求精	第 4 层：质量属性场景	重要性	难度	累加和
可维护性	M1：对一个子系统的更改不要要求改动其他子系统	M1.1：部署下一个科学数据服务器版本，实现对当前版本的更新。升级应该在 8 小时以内完成，并且不应影响其他子系统及查找、浏览或预定功能的使用。	30	30	60
	M2：独立地回退子系统的部署	M2.1：使科学数据服务器从 M1 回退	20	20	40
可操作性	O10：系统应该能在 20 分钟内根据用户类型、数据类型、介质类型、目的地或用户对 1000 项定单重新设置优先级	O10.1：积压任务管理-- 在系统连续 24 小时不能正常工作后，应在 30 分钟内为所积压的任务设置优先级，以保证各任务能够按优先级得以处理且正常的操作能够得以维持，不会在恢复正常操作状态后	30	20	50

参考学习

- [互联网系统设计原则](#)

谢谢阅读

配套文章：wxquare.github.io