系统通道通识课程

讲　师：谢志远、吴小君

课程开发：谢志远、毛灵、吴小君、马杰星

课程目标

- 认识与了解B端（M端）系统的特点
- 学习与了解系统通道所需的基础知识
- 通过实例讲解如何做系统开发

适用人群

- 非系统通道了解系统通道覆盖系统的特点和系统开发所需的基础知识
- 系统通道校招、社招新同学了解系统通道的全貌和专业知识及技能要求

课程目录

B端（M端）系统介绍

销售

商家

司机

员工

B端（M端）系统介绍

系统定义

B端全称是Business即商家（泛指企业）的产品，通常是企业或商家，为工作或商业目的而使用的系统型软件、工具或平台。例如：大象、开店宝、阿波罗等。

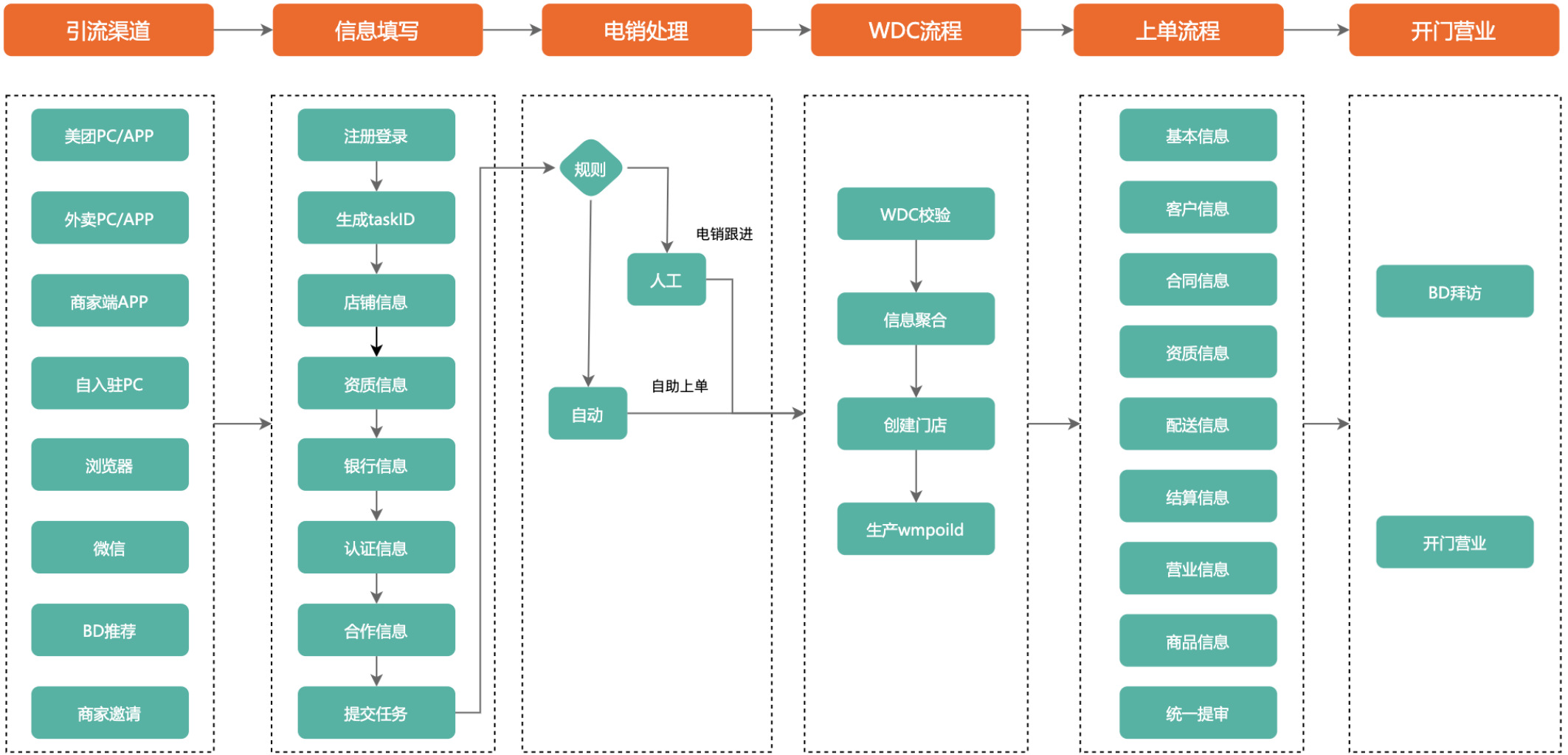
系统用户

企业员工、商分、运营、销售、商家、合作商、骑手、司机、客服 …

系统特点

线下流程信息化、系统化，业务流转效率更高，成本更低。效率优于体验。

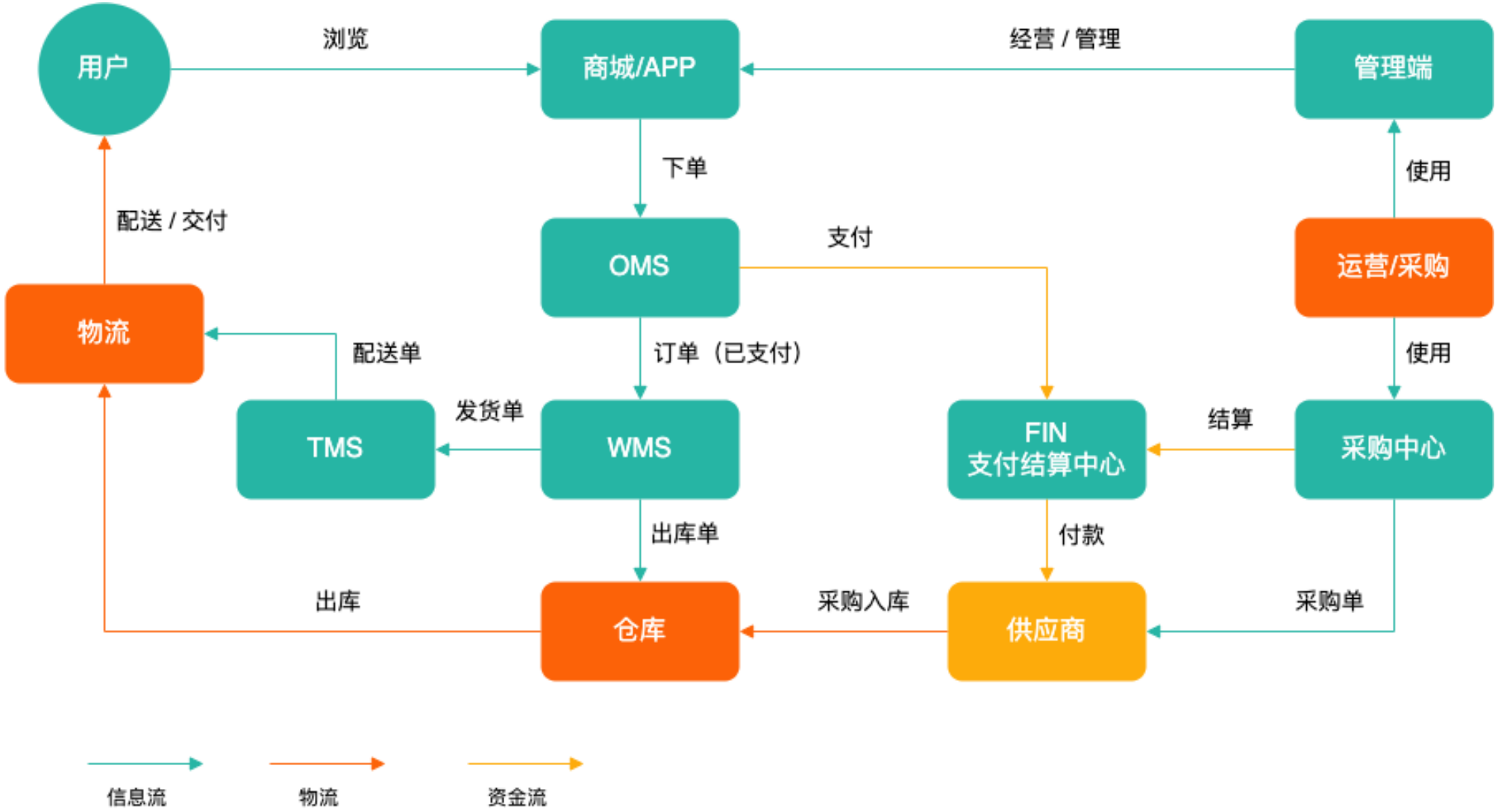
常见业务流程 – 外卖商家入驻流程



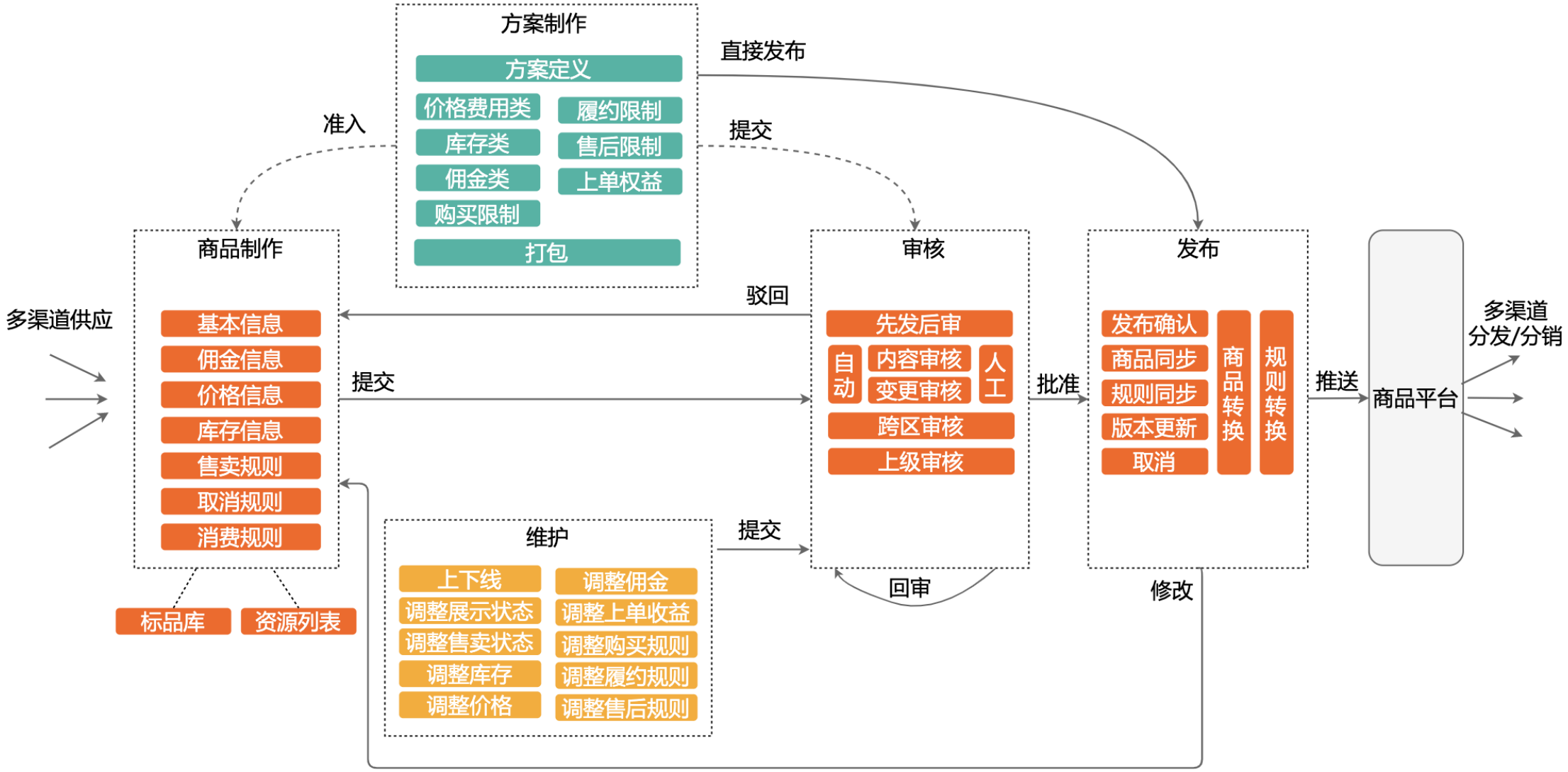
常见业务流程 – 销售流程


售前系统
新建门店
销售新建
商家新建
POI部门
运营分配门店
战区管理
销售跟进门店
商机平台
门店管理
拜访管理
轮转管理
售中系统
商户入驻
营业执照
实名认证
银行卡信息
签合作协议
纸质协议
电子协议
产品售卖
方案管理
上单管理
售后系统
业绩查看
首页
仪表盘
商户续约
维护提醒
销售数据
奖金计算
指标体系
销售业绩

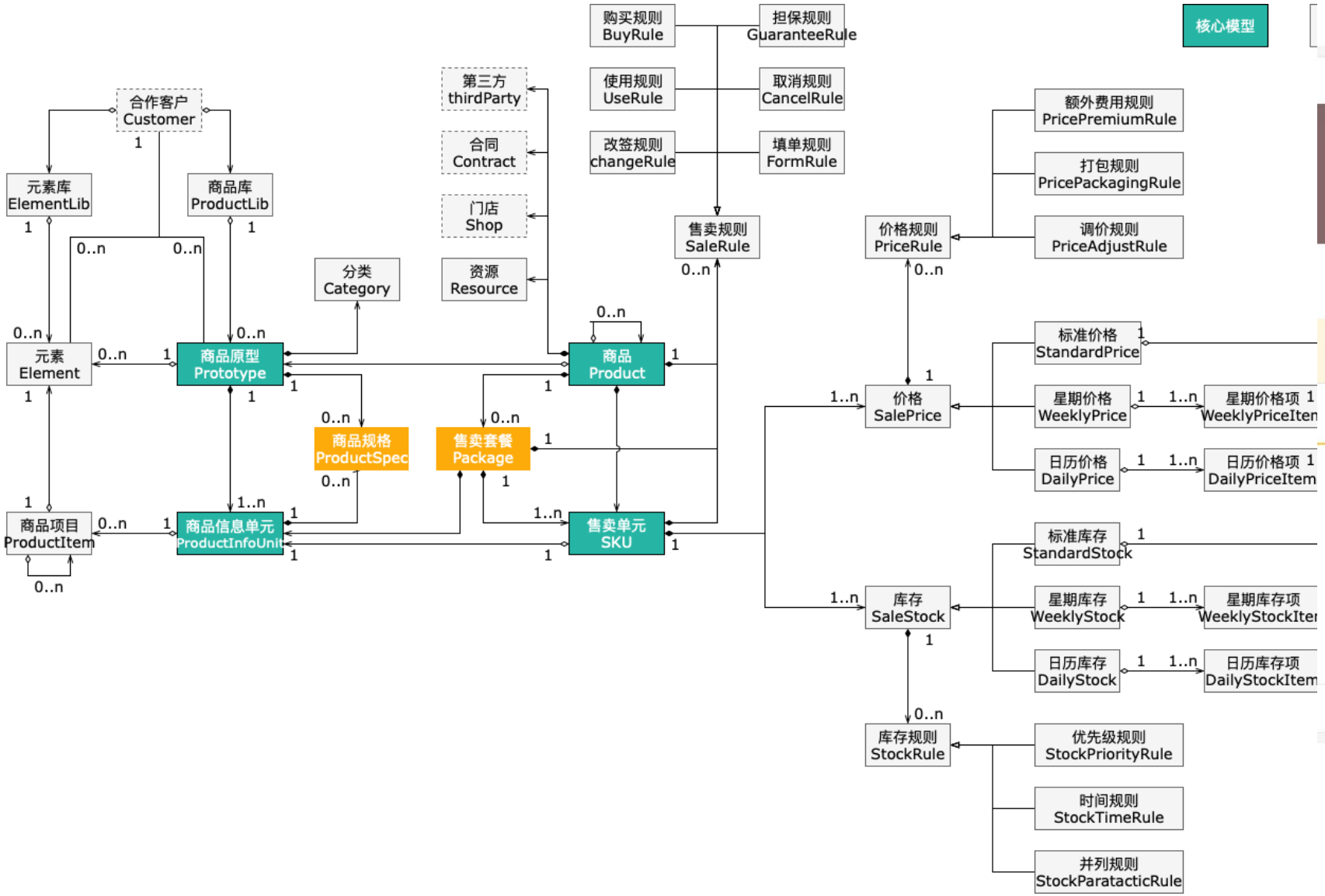
常见业务流程 – 快驴下单业务流程



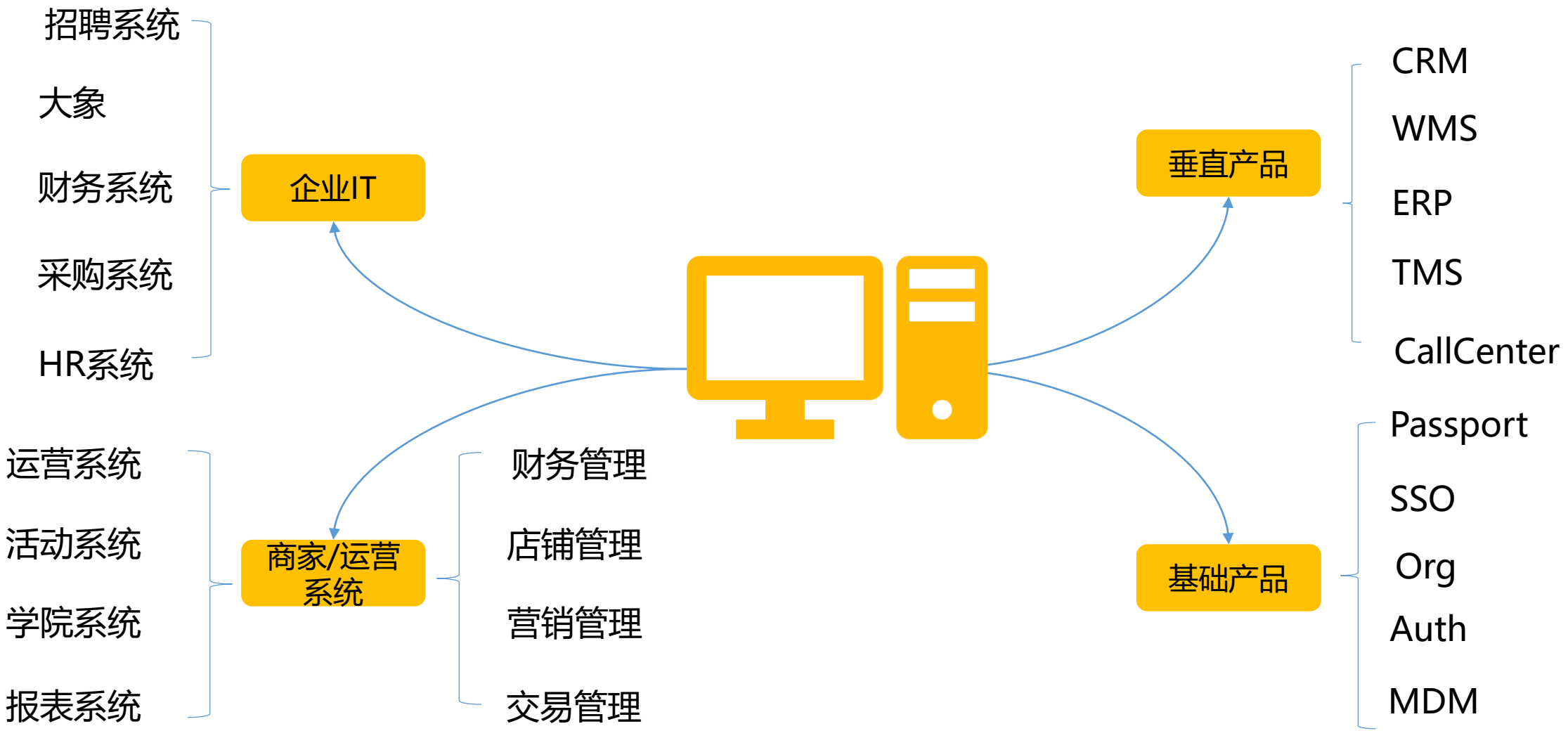
常见业务流程 - 到店供应链上单流程



常见业务流程 – 到综商品模型

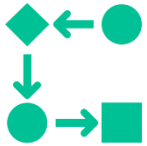


B端（M端）产品大图


招聘系统
大象
财务系统
采购系统
HR系统
企业IT
垂直产品
CRM
WMS
ERP
TMS
CallCenter
运营系统
活动系统
学院系统
报表系统
商家/运营系统
财务管理
店铺管理
营销管理
交易管理
基础产品
Passport
SSO
Org
Auth
MDM

系统开发的特点

使用角色多：CRM系统为例：销售、销售主管、战区长官、Bu Head、运营人员等看到的数据以及操作界面都不完全相同。



业务流程长：以外卖商家入驻为例，包括渠道引流、信息填报、销售处理、WDC处理、上单处理、开门营业等流程，要管理好状态的流转和流程的可配。

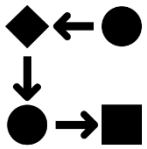
策略变化快：销售策略每月的变化需要系统灵活支持；商家评分模型随时可能调整需要系统当天生效。

模型复杂：商品模型包含的商品、价格、库存、售卖等复杂实体，需要支持将来可能新出现（或已有变更）的商品定义、定价和售卖。

系统开发VS后台开发

系统开发要求：可扩展、可复用、可维护


模型通用

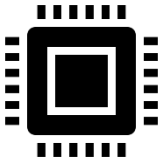

流程复用


灵活配置


质量可靠

VS


高并发


高性能


高可用


可扩展

业务复杂性高 → 提效、降本

技术复杂性高 → 保质、降本

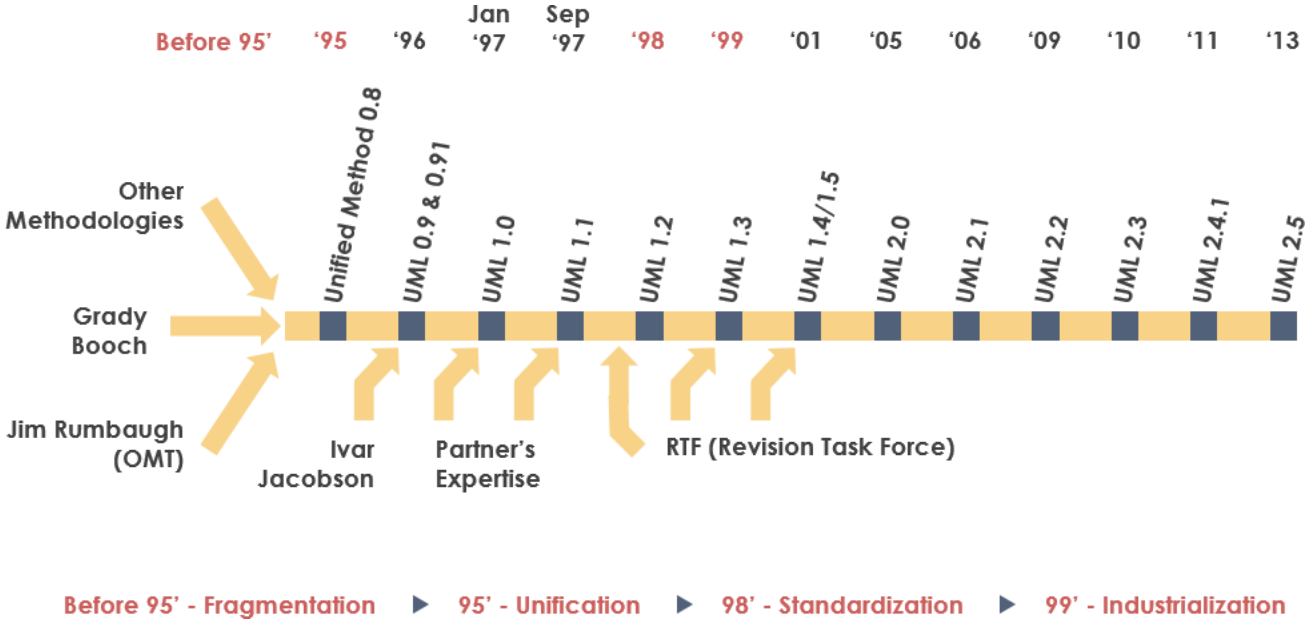
课程目录

UML 简介

是什么

UML（Unified Modeling Language）

是面向对象软件的**标准化建模语言**



解决什么问题

- 表达和阐述系统的结构或行为
- 让系统设计图像化
- 提供构建系统的模板
- 有助于理解复杂系统

音乐 vs 五线谱 / 数学 vs 公式 / 软件需求模型 vs UML

UML 模型

- 功能模型

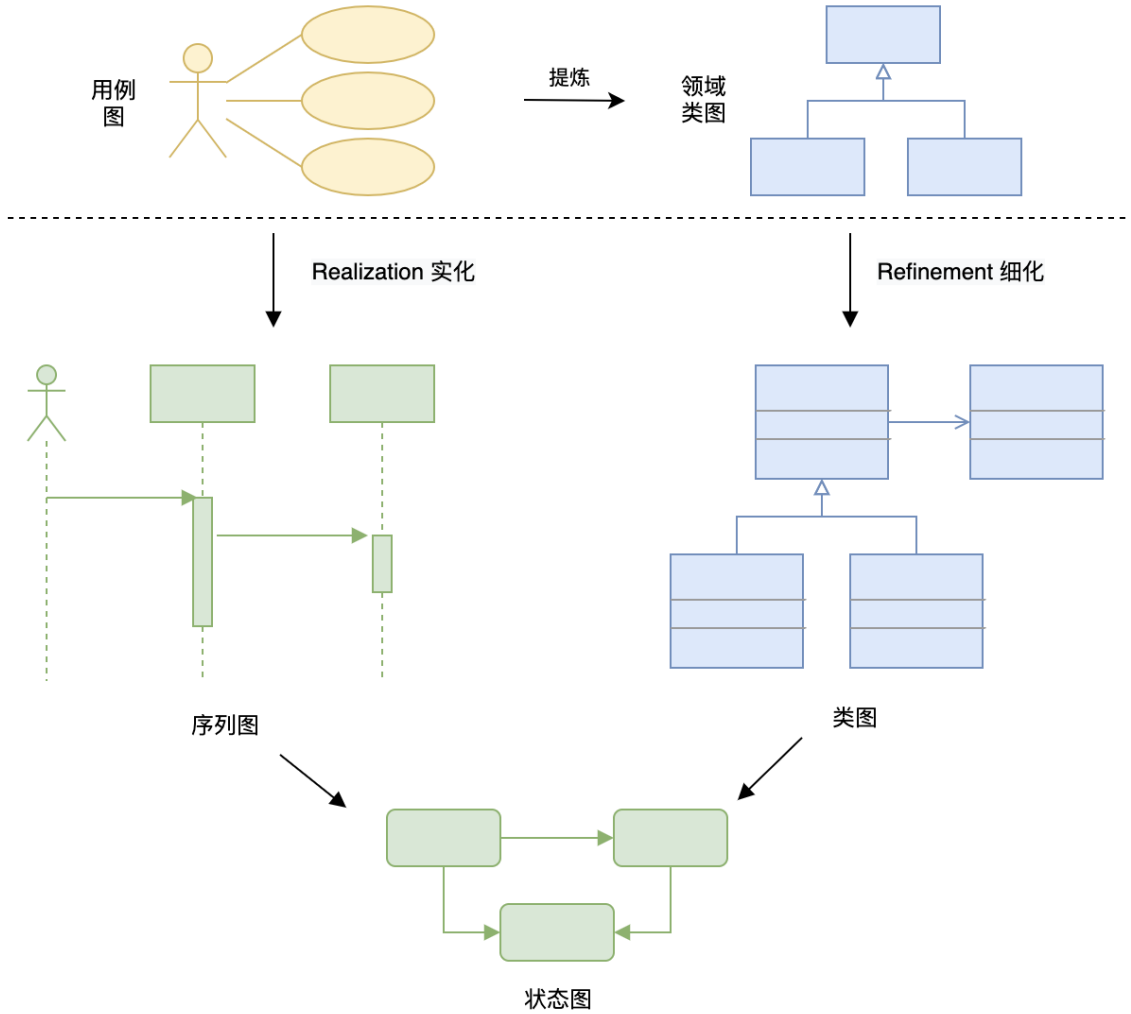
  展示系统的外部功能，包括用例图

- 对象模型

  展示系统的内部结构，包括类图、对象图等

- 动态模型

  展示系统的内部行为，包括序列图，活动图，状态图等



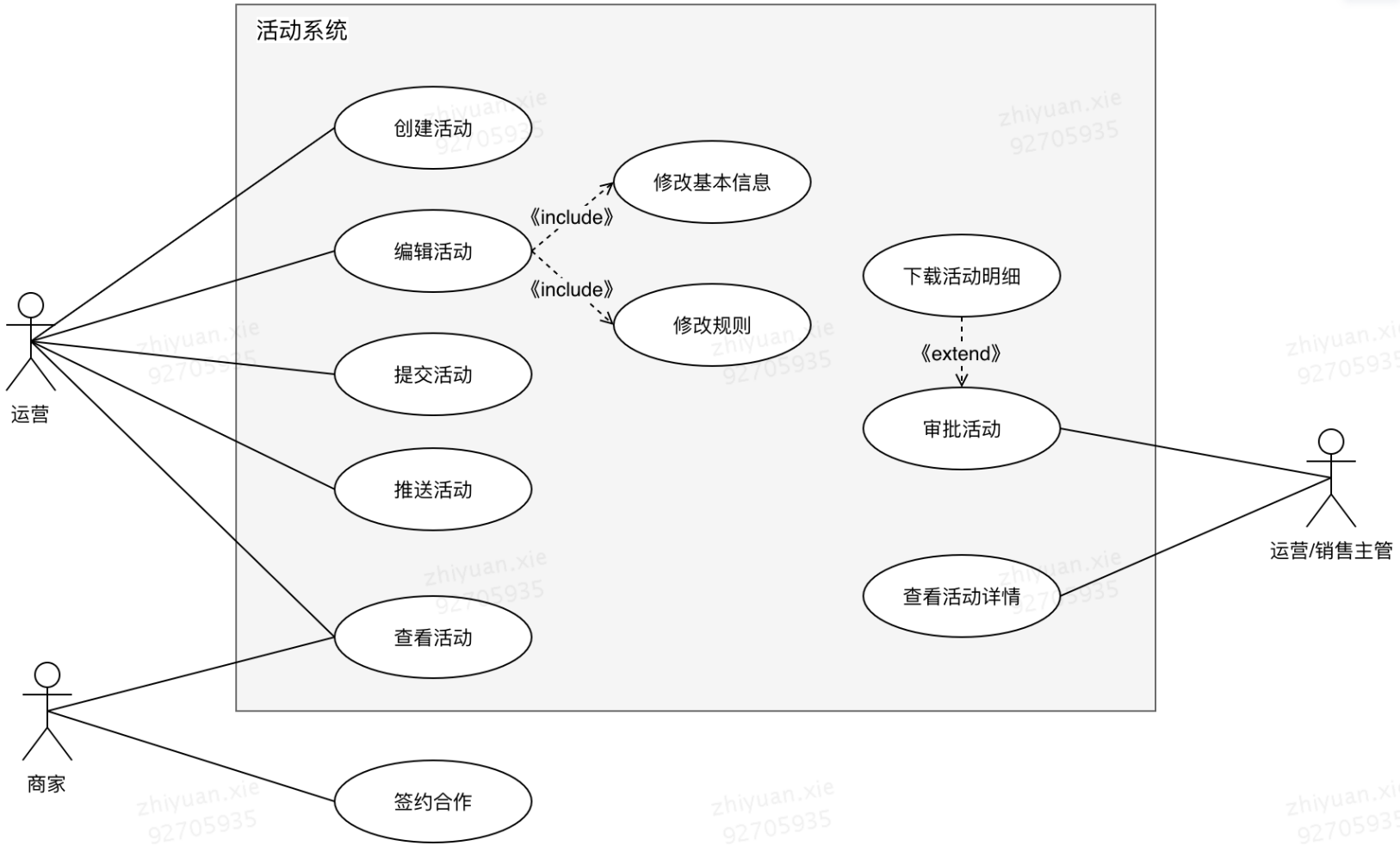
模型独立于图形，图形是模型信息的表达方式

UML对象模型 - 用例图

用例图是外部用户（被称为参与者）所能观察到的系统功能的模型图

要素

- 参与者
- 用例
- 边界
- 关系



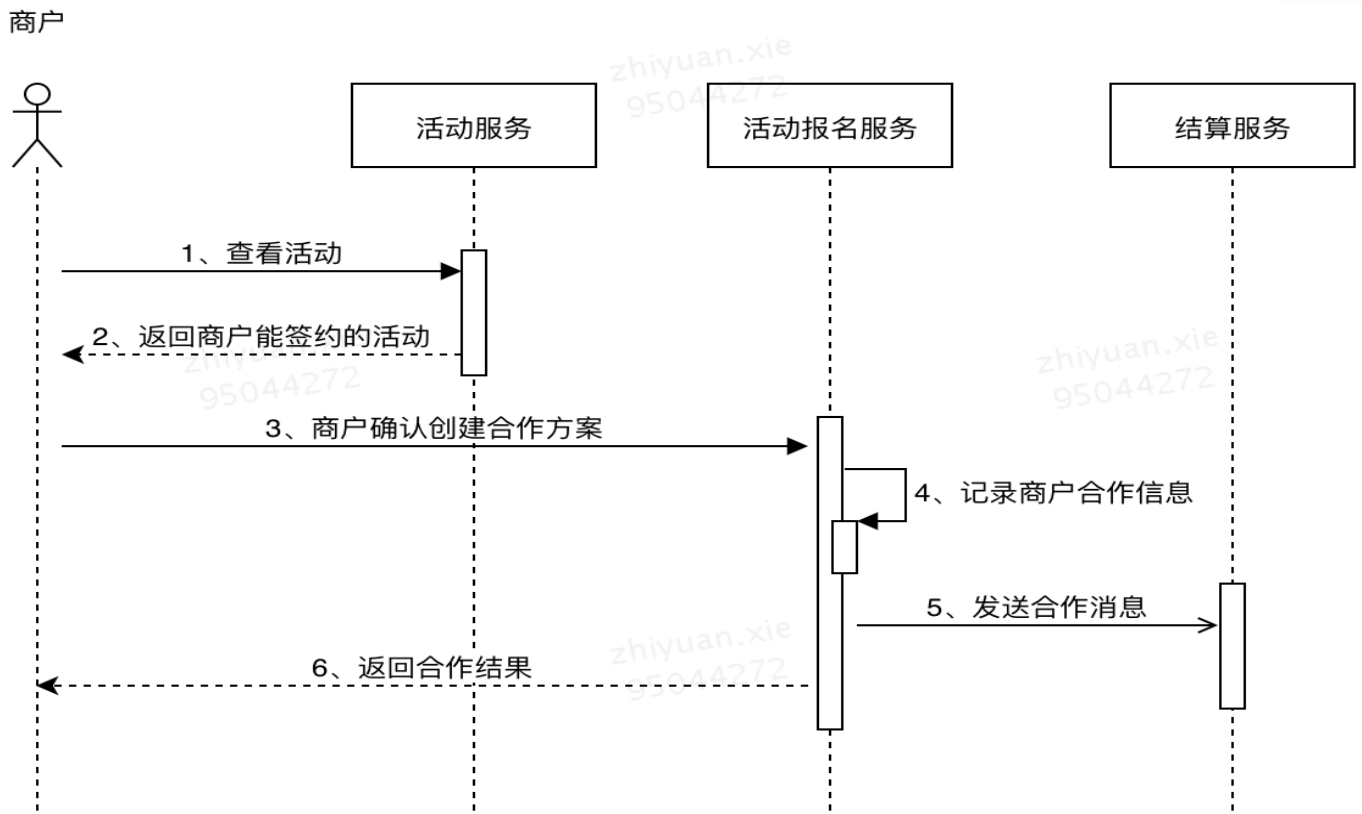
UML用例图示例

UML 动态模型 - 序列图

序列图是一种行为图，用于描述消息在对象生命线上按照约定顺序执行的交互行为

要素

- 角色
- 对象
- 生命线
- 消息
- 激活期



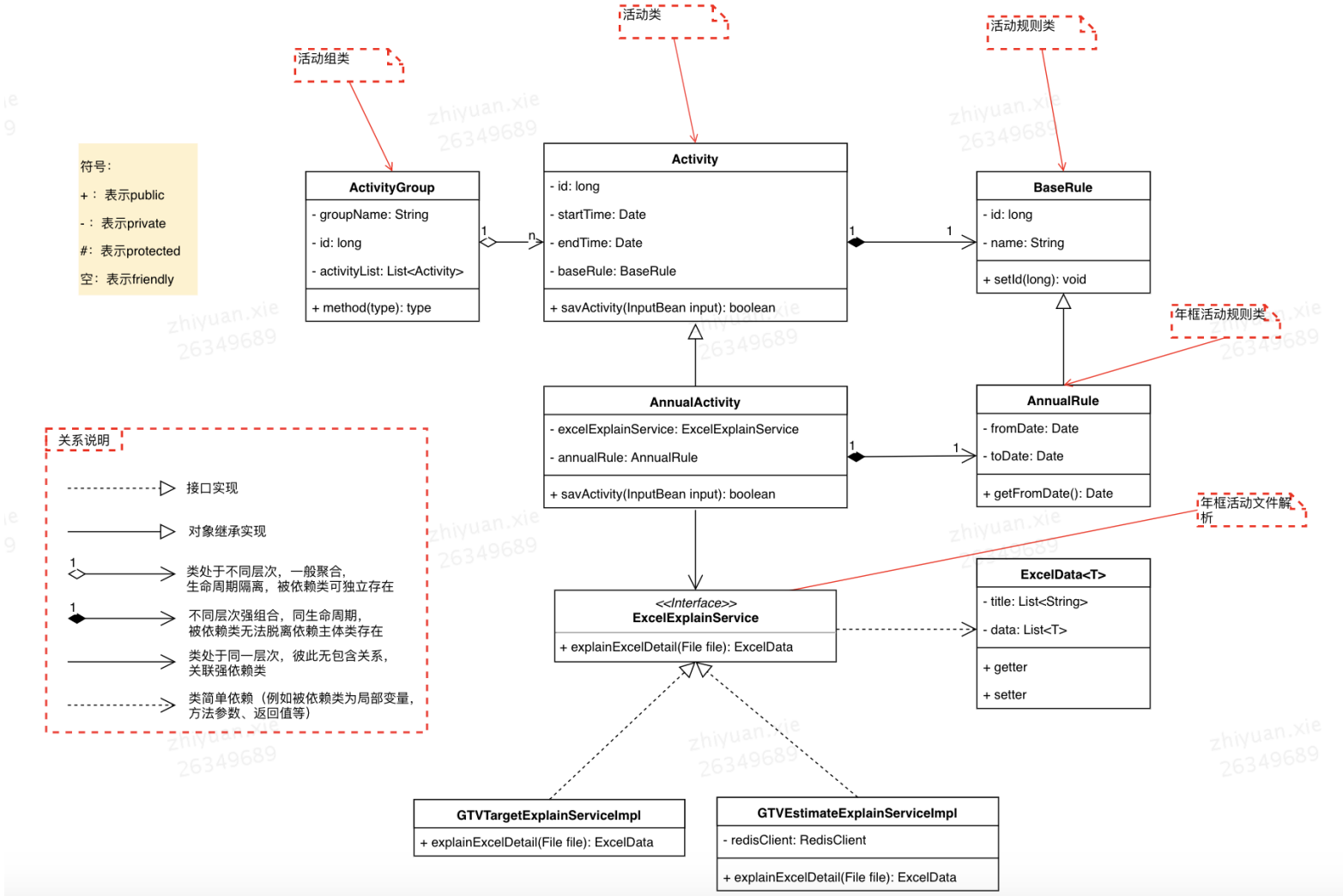
UML序列图示例

UML对象模型 – 类图

类图是一种静态结构图，用于描述系统的类集合，类的属性和类之间的关系

要素

- 类名
- 属性
- 操作
- 关系

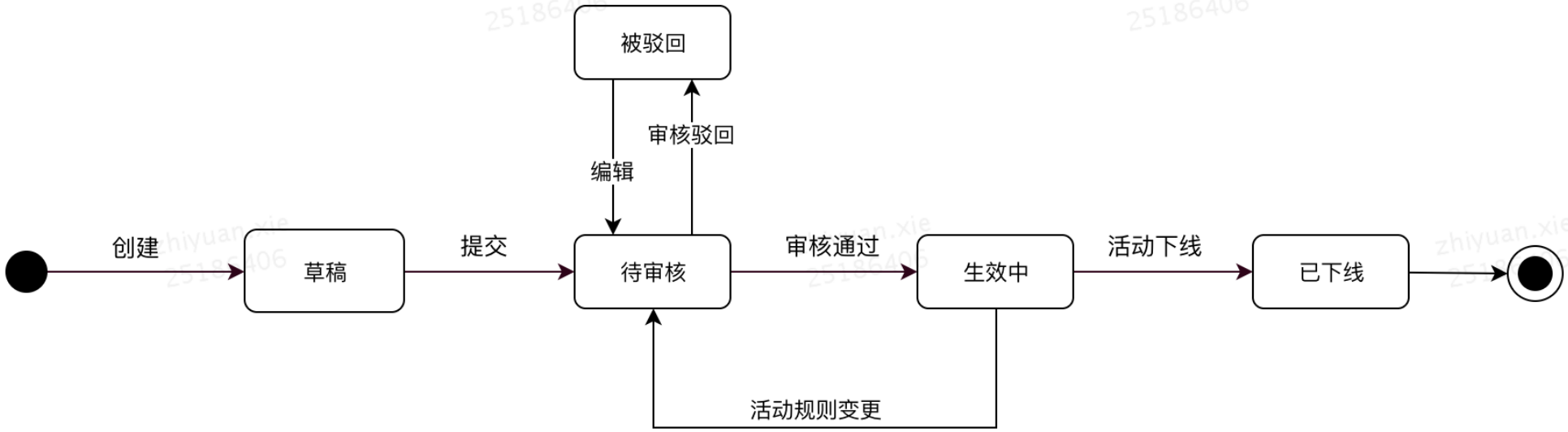


UML类图示例

UML 对象模型 - 状态图

状态图是描述一个实体基于事件反应的动态行为，显示了该实体如何根据当前所处的状态对不同的事件做出反应。

活动状态图：

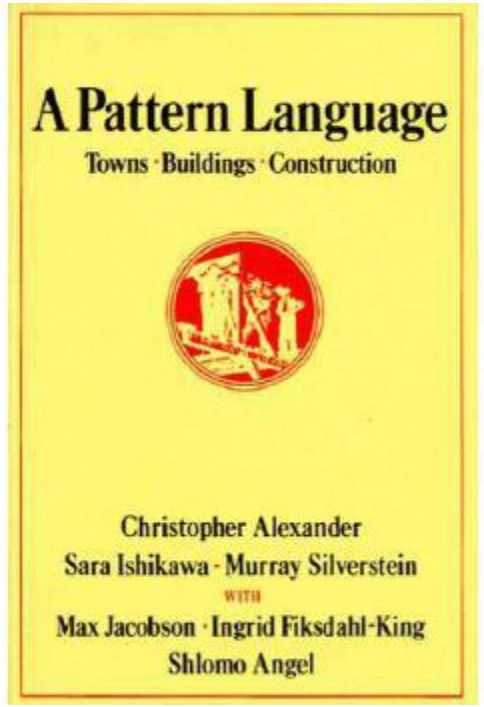


UML类图示例

课程目录

设计模式简介

- **Christopher Alexander**
  - 模式(Pattern)之父
  - 美国加利佛尼亚大学环境结构中心研究所所长
- **何为设计模式**
  - 设计就是某个问题的解决方案
  - 如果某个解决方案对某类问题都很有用，我们就把它总结出来——形成了设计模式



面向对象、设计原则、设计模式

面向对象

- 丰富的特性，可以实现很多复杂的设计，是很多设计原则、设计模式实现的基础

设计原则

- 指导软件设计的经验总结，对于某些场景下，是否应该应用某种设计模式，具有指导意义

设计模式

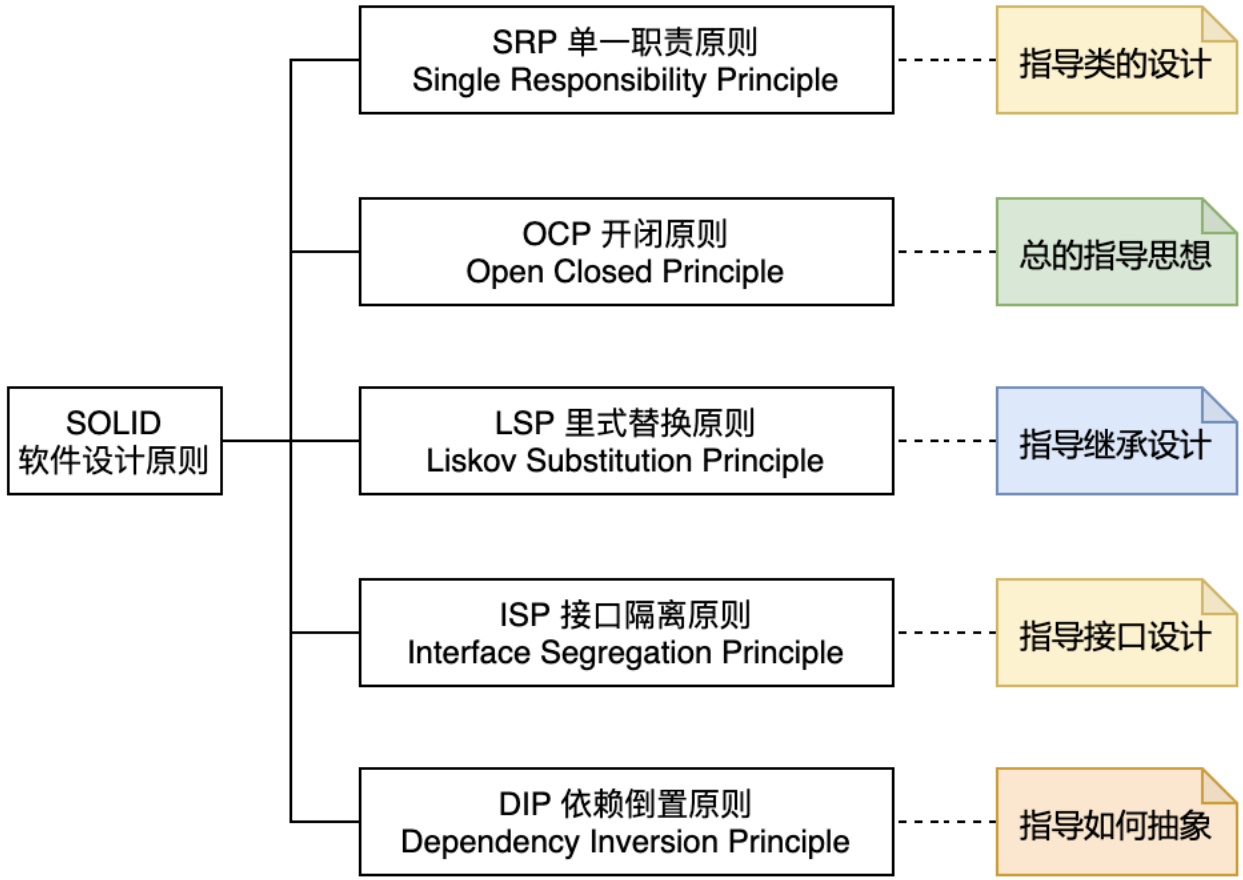
- 针对软件开发中经常遇到的一些设计问题，总结出来的一套解决方案或者设计思路，比设计原则更具体、可执行

设计模式原则 - SOLID

设计原则是数十年软件工程经验、大量软件开发人员和研究人员思想和著作的成果，有助于消除不良设计，构建出最好的设计

不良设计问题：

- 僵化：设计难以改变
- 脆弱：设计易于破坏
- 死板：设计难以重用

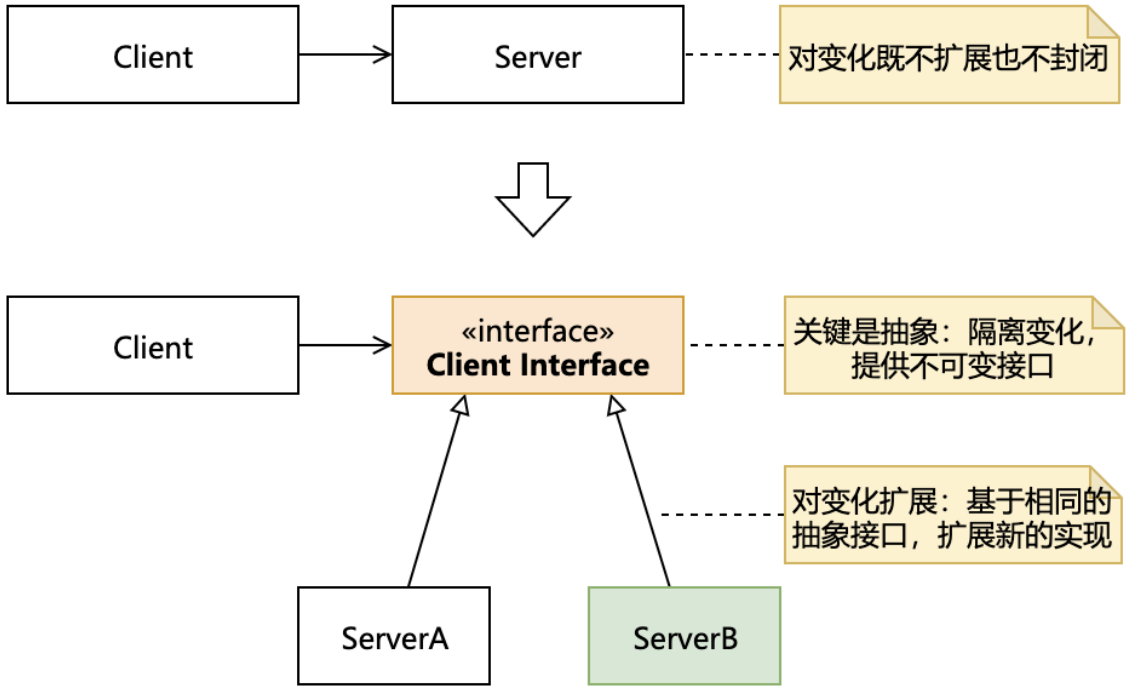


设计模式原则 - 开闭原则(OCP)

软件实体（模块、类、方法等）应该是对扩展开放、对修改关闭。它是软件设计中最重要的原则之一，强调软件应通过扩展来实现变化，而不是通过修改已有的代码，是构建可维护性和可重用性代码的基础。

具体实例

- 模板模式
- 策略模式
- 责任链模式



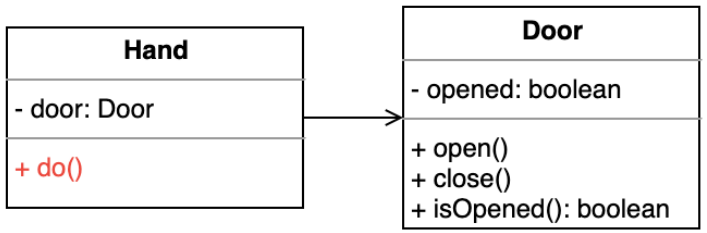
开闭原则示例图（策略模式）

经典的23种设计模式

大部分是为了解决扩展性问题

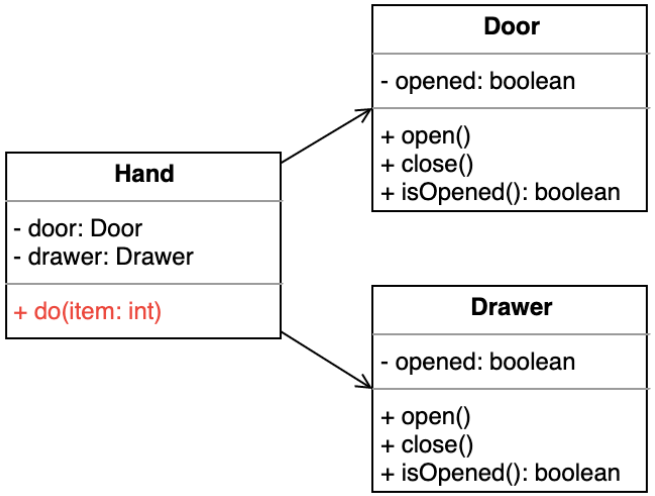
设计模式原则 - 开闭原则(OCP) - 示例

程序模拟用手开关门



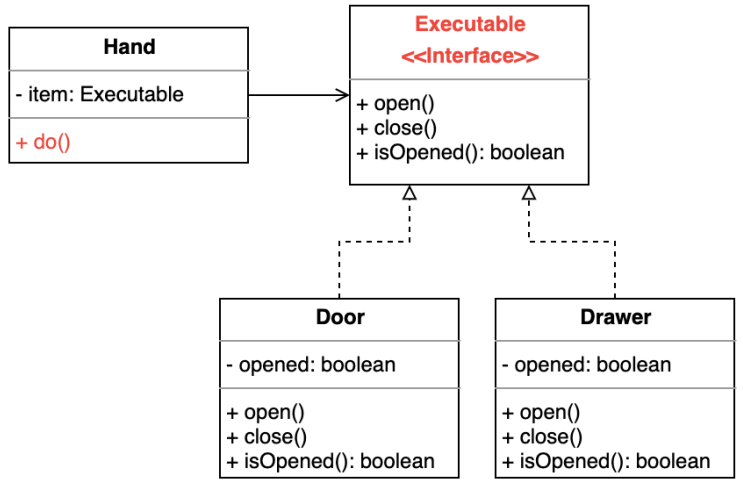

```
public class Hand {
    public Door door;
    void do() {
        if (door.isOpen())
            door.close();
        else
            door.open();
    }
}
```

新需求：用手关抽屉、冰箱 ...




```
public void do(int item){
    switch (item){
        case 1:
            if (door.isOpen())
                door.close();
            ...
        case 2:
            if (drawer.isOpen())
                drawer.close();
            ...
    }
}
```

符合OCP的设计

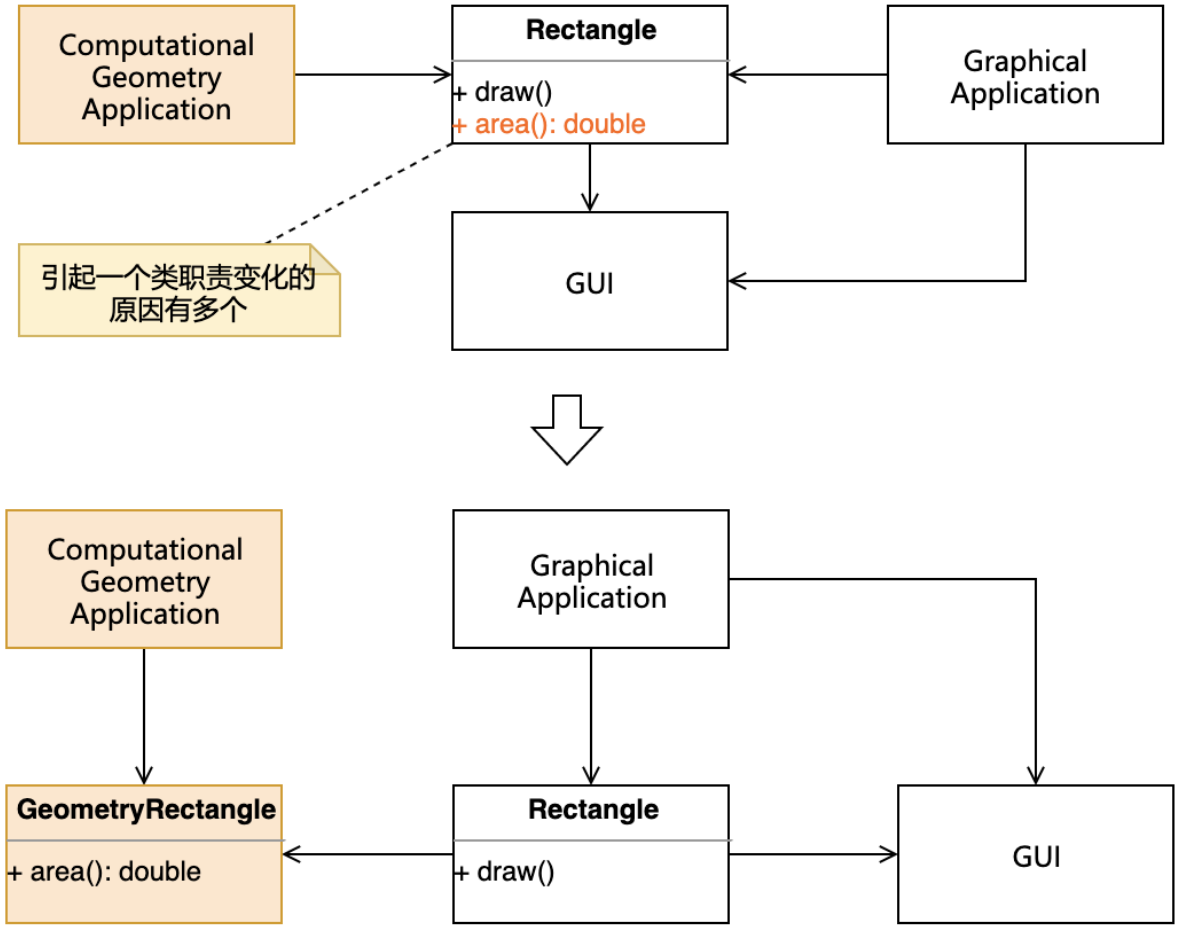



```
public class Hand {
    public Excutable item;
    void do() {
        if (item.isOpen())
            item.close();
        else
            item.open();
    }
}
```

设计模式原则 - 单一职责原则(SRP)

一个类或者模块只负责完成一个职责（功能）

- SRP（Single Responsibility Principle）是为了实现代码高内聚，提高代码的复用性、可读性、可维护性
- 一个类承担的职责过多和耦合，会削弱其完成其他职责的能力，这种耦合会导致脆弱的设计，变化发生时设计遭到意想不到的破坏



单一职责原则示例图

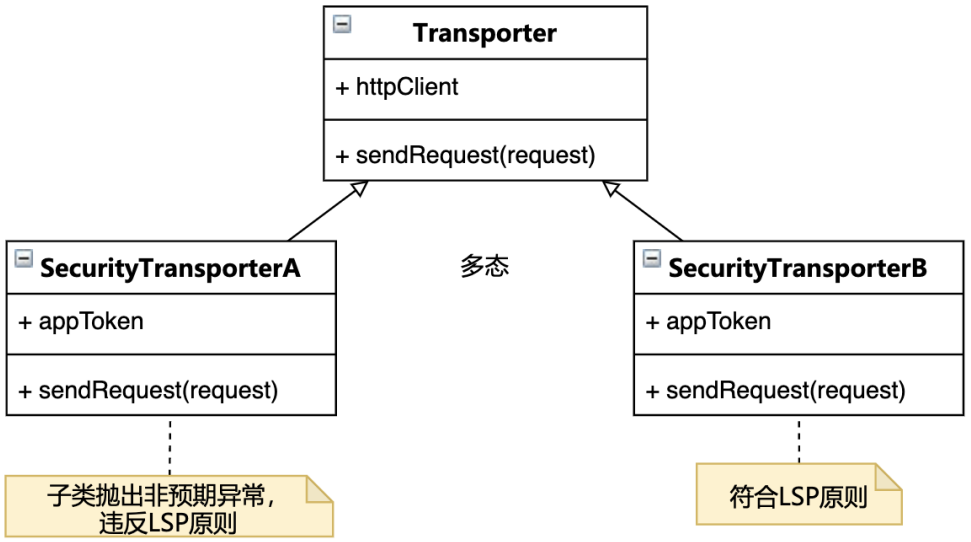
设计模式原则 – 里氏替换原则(LSP)

- 子类对象能够替换程序中父类对象出现的任何地方，并且保证原来程序的逻辑行为不变及正确性不被破坏

- OCP背后的主要机制是抽象和多态，支持其关键机制之一是继承，而LSP保证了继承不破坏OCP

典型违反LSP原则的例子：

- 子类违背父类方法声明要实现的功能
- 子类违背父类对输入、输出、异常的约定




```
@Override
public Response sendRequest(Request request){
    if(StringUitls.isBlank(appToken)){
        throw new NoAuthorizationRuntimeException();
    }
    return super.sendRequest(request);
}
```

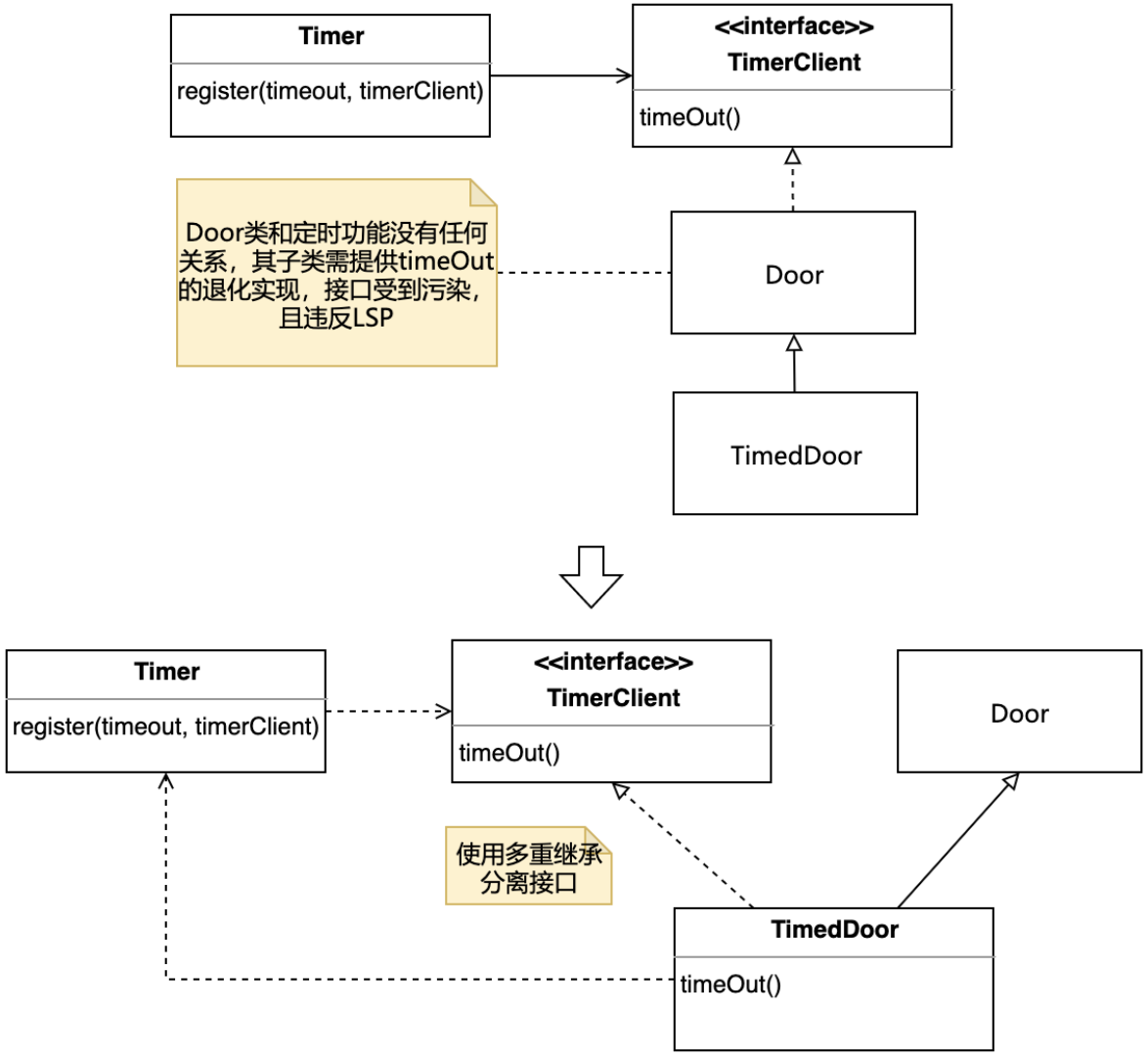
```
@Override
public Response sendRequest(Request request){
    if(!StringUitls.isBlank(appToken)){
        request.addPayload("appToken", appToken);
    }
    return super.sendRequest(request);
}
```

里式替换原则反例示例图

设计模式原则 - 接口隔离原则(ISP)

- 客户端不应该被强迫依赖它不需要的接口。

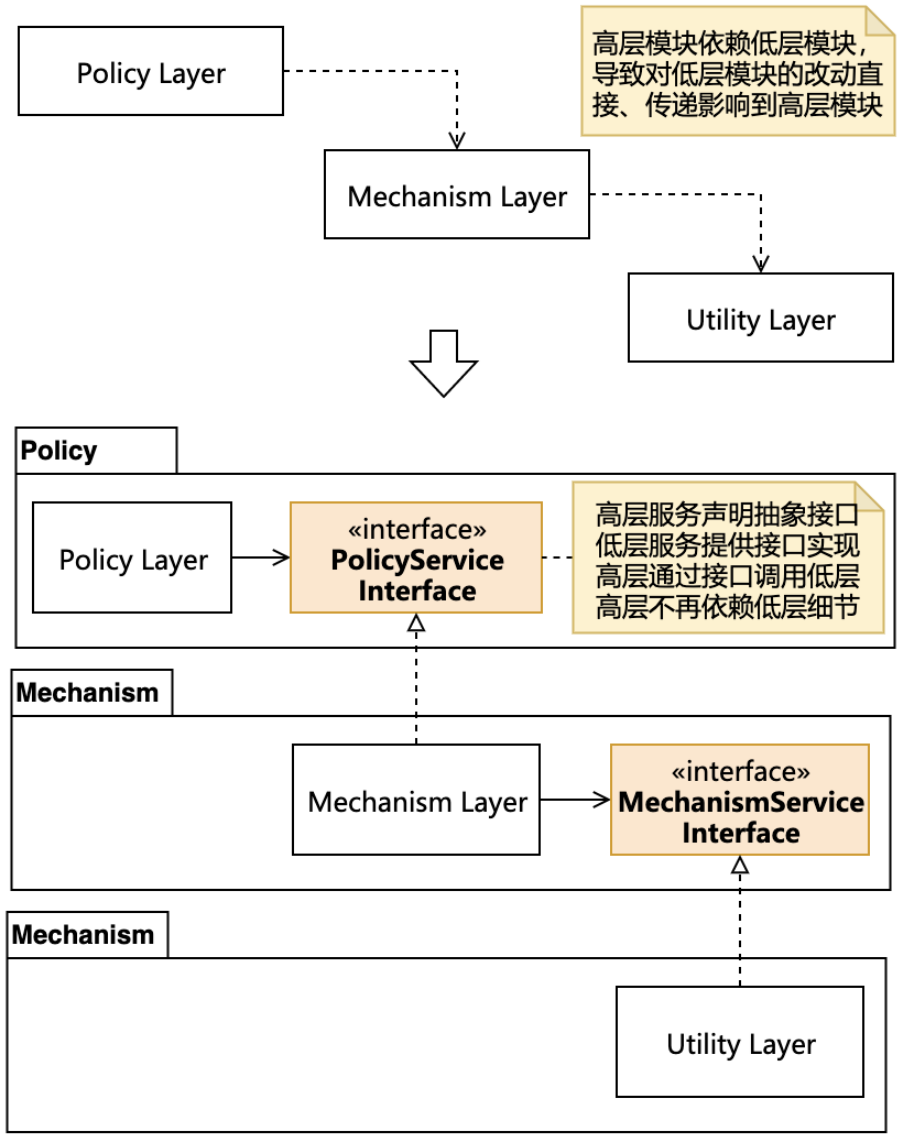
- 接口隔离原则提供了一种判断接口的职责是否单一的标准，即通过调用者如何使用接口来间接地判定：如果调用者只使用部分接口或接口的部分功能，那接口的设计就不够职责单一。



接口隔离原则示例图

设计模式原则 – 依赖倒置原则(DIP)

- 高层模块不应该依赖低层模块。二者应该都依赖于抽象
- 所有结构良好的面向对象架构都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供了一组内聚的服务



依赖倒置原则示例图

设计模式带来的好处

A

松耦合

松耦合系统设计，一方面能够适应灵活的系统变化，另一方面当系统内部结构变化时不会影响到其他系统模块。

B

面向对象特性

面向对象方式抽象概念，一方面隐藏内部复杂性，另一方面借助对象的封装能力能够保障边界不被突破。

C

改善设计

促进基于模式的开发，方便进行重构，可以确保开发正确的代码，并降低在设计或实现中出现的错误的可能。

课程目录

为什么需要软件架构

随着软件系统规模的增加，计算相关的算法和数据结构不再构成主要的设计问题；当系统由许多部分组成时，整个系统的组织，也就是所说的“软件架构”，导致了一系列新的设计问题。

---《An Introduction to Software Architecture》

20世纪60~70年代

第一次软件危机

逻辑复杂问题

面向过程
C、Pascal语言诞生

20世纪80~90年代

第二次软件危机

扩展复杂问题

面向对象
C++、Java等语言诞生

2000年以后

互联网的兴起

规模复杂问题

面向服务
领域驱动

主要内容介绍

A 分层架构

经典分层及架构演进

六边形架构

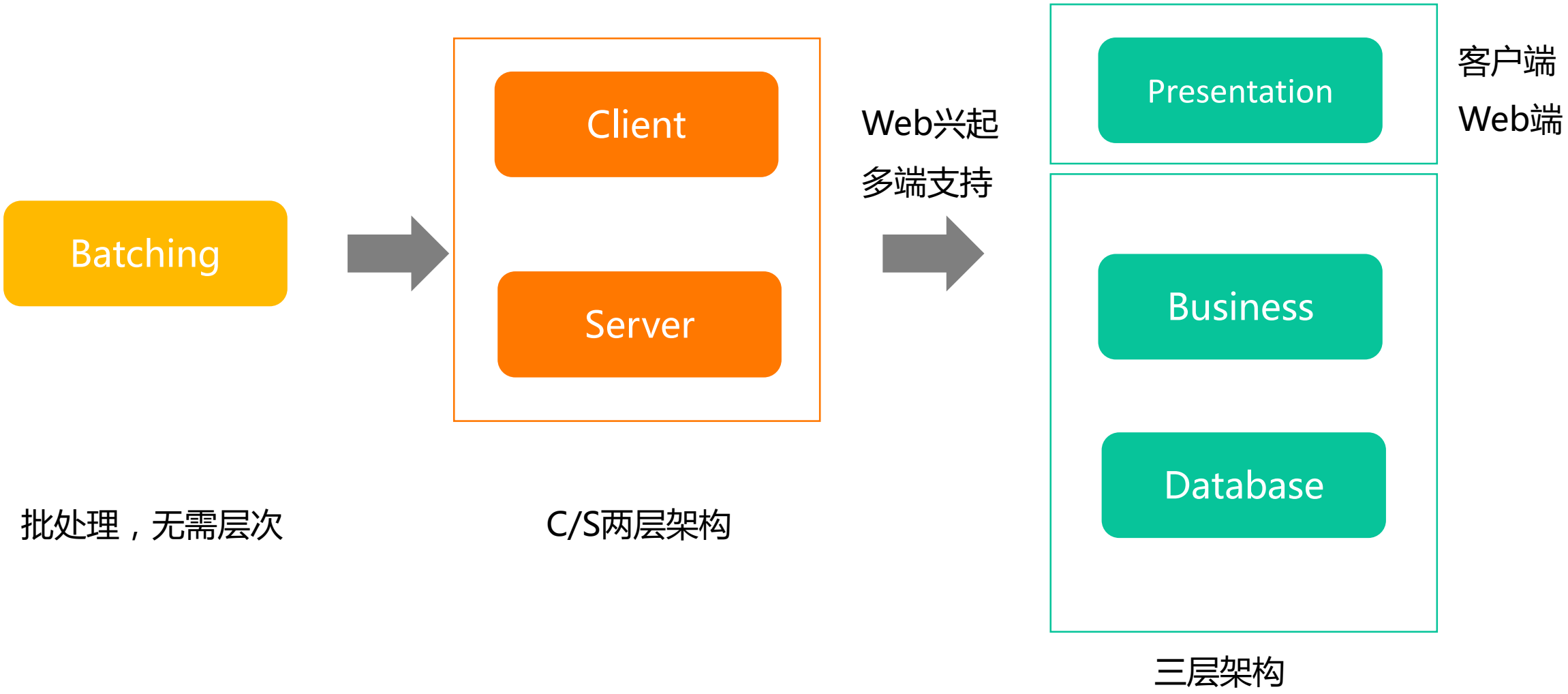
B 微内核架构

分离变与不变的部分

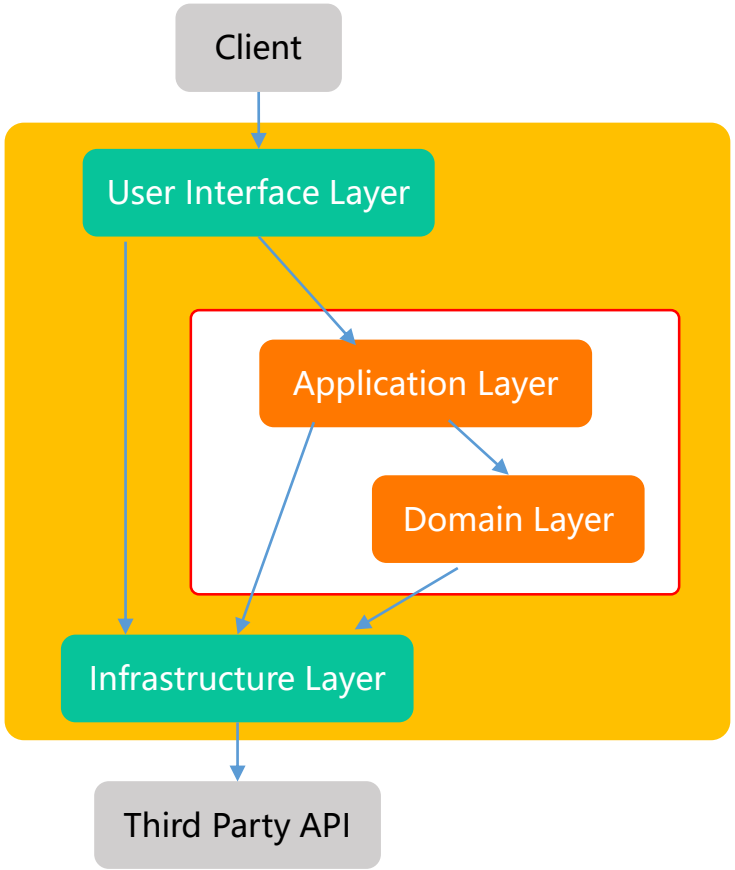
C 混合架构

各种架构的组合应用

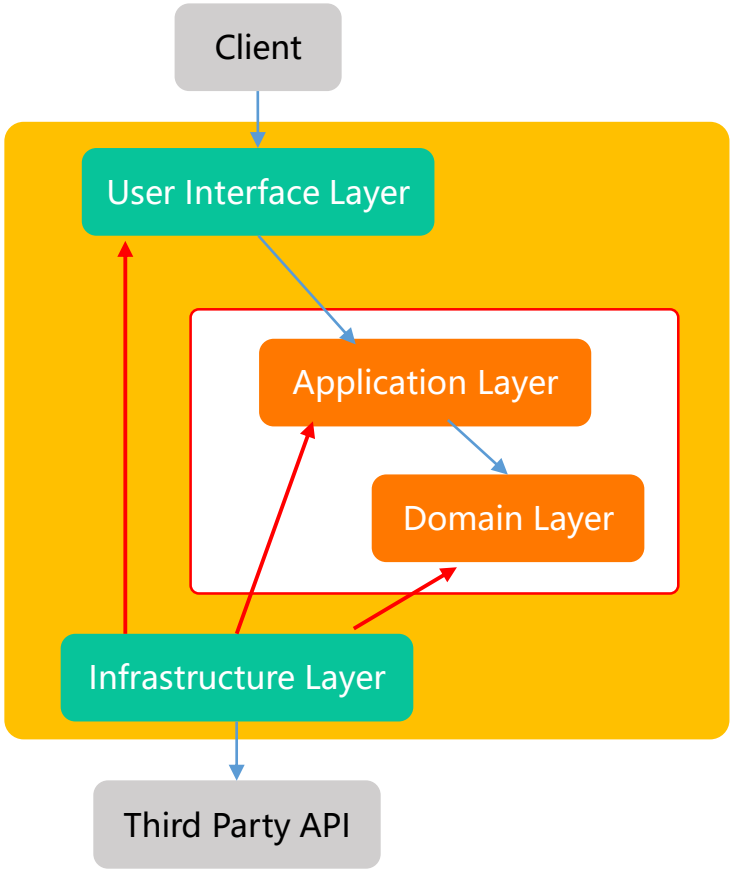
可扩展系统架构 – 经典分层演化



可扩展系统架构 – 经典的四层架构

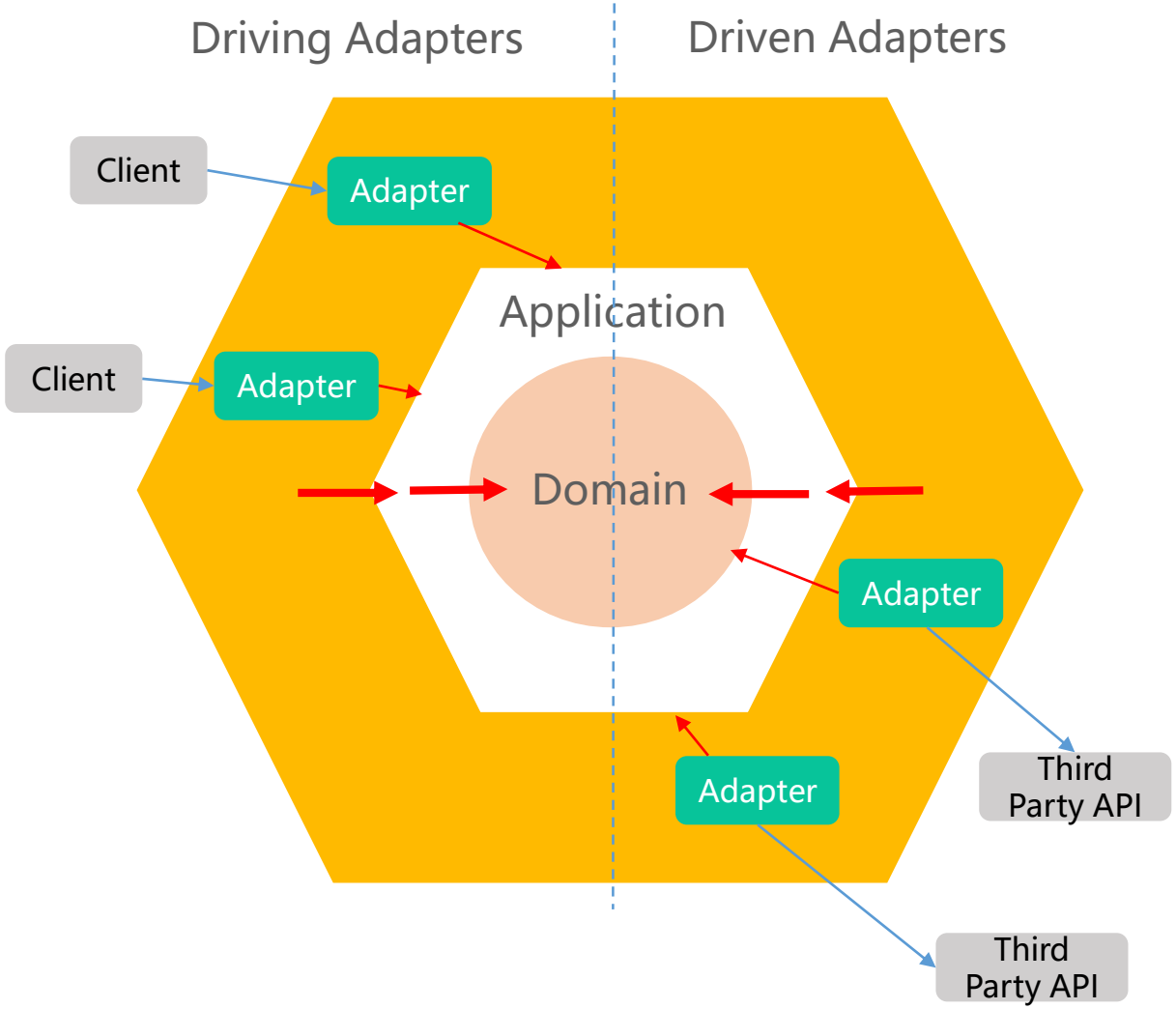


经典四层架构



分层架构+DIP

可扩展系统架构 – 六边形架构（端口适配器架构）



- 隔离核心逻辑与外部关注点
- 依赖向内收缩
- 促进可测试性和持续演进

分层架构的优缺点

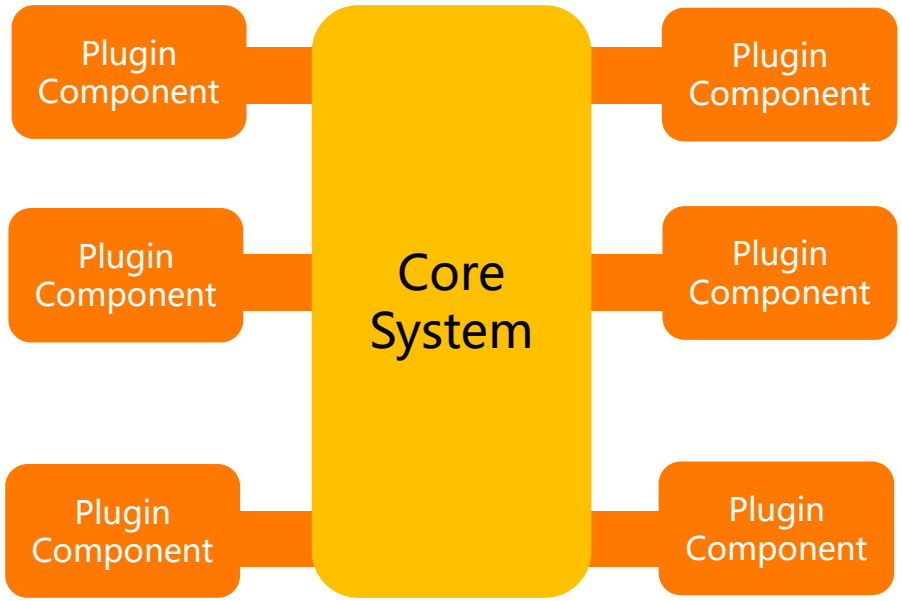
A 优点

- **分离关注**：
  - 开发人员可以只关注整个结构中的某一层
- **松散耦合**：
  - 可以很容易用新的实现替换原有层次实现
  - 降低层与层之间的依赖
- **逻辑复用**：利于各层逻辑的复用
- **标准定义**：有利于标准化

B 缺点

- **性能降低**：不同层次间调用耗时
- **级联修改**：如果表现层新增功能，则对应的业务层及持久层可能需要新增代码

可扩展系统架构 – 微内核架构



内核系统

- 插件管理
- 插件加载
- 插件通信
- 业务隔离

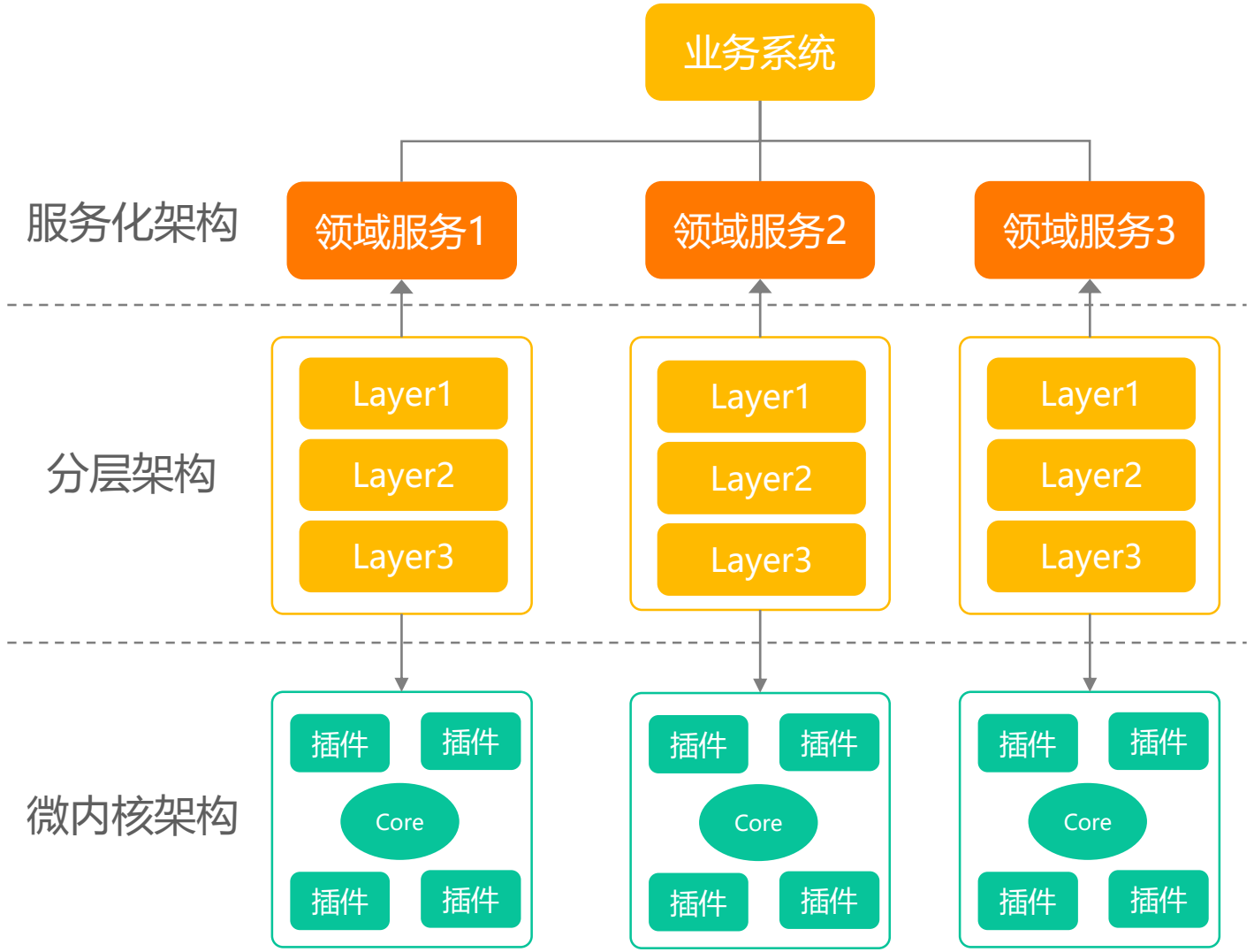
插件模块

- 封装变化的业务逻辑
- 支持快速灵活扩展

微内核（插件化）架构：隔离易变与不变

可扩展系统架构

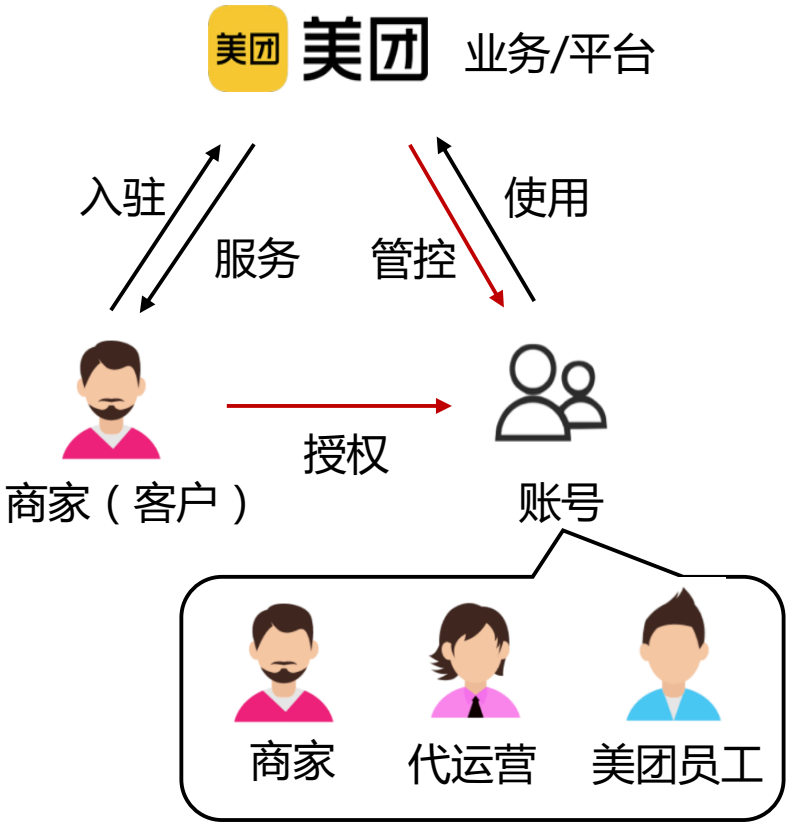
架构拆分



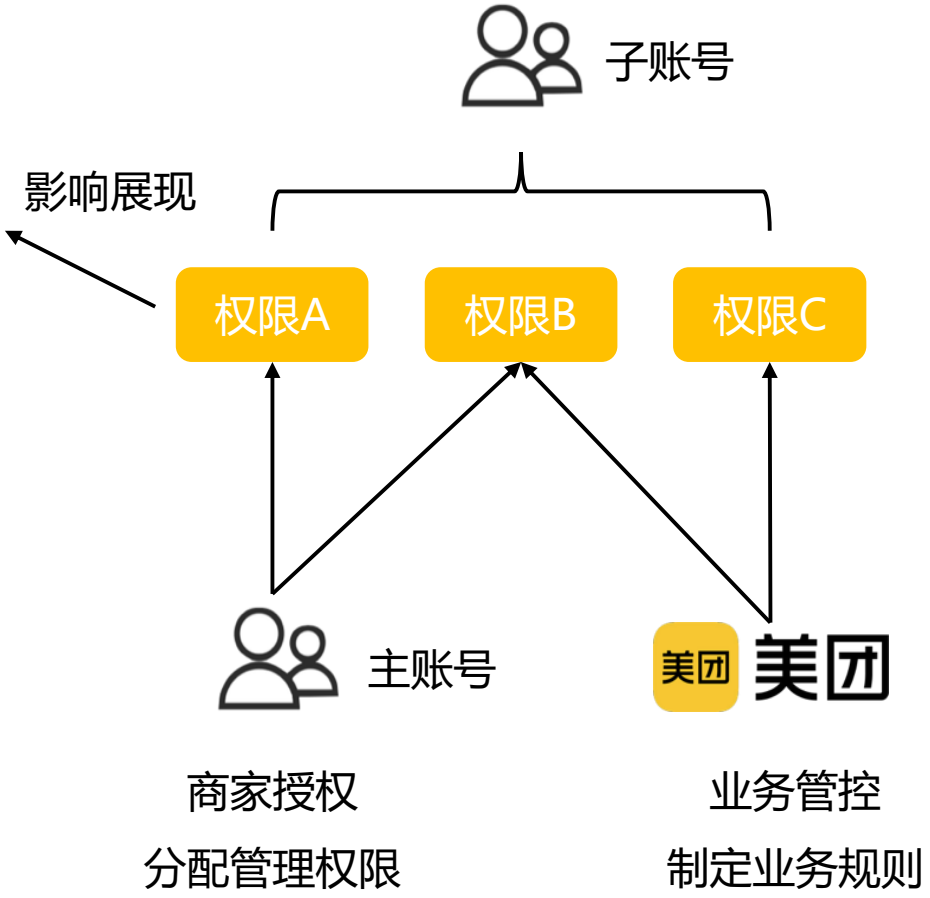
- 业务维度：纵向拆分
- 技术维度：横向拆分
- 功能维度：变化隔离

课程目录

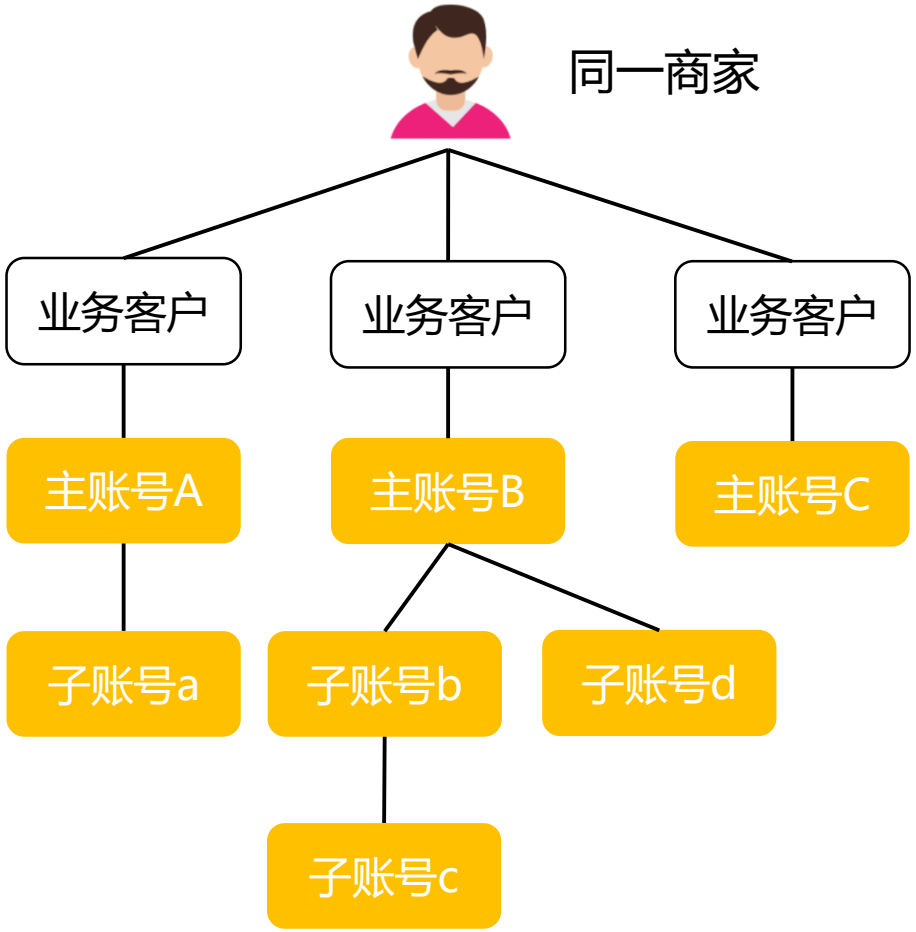
商家账号权限业务


美团
业务/平台
入驻
服务
管控
使用
授权
商家（客户）
账号
商家
代运营
美团员工

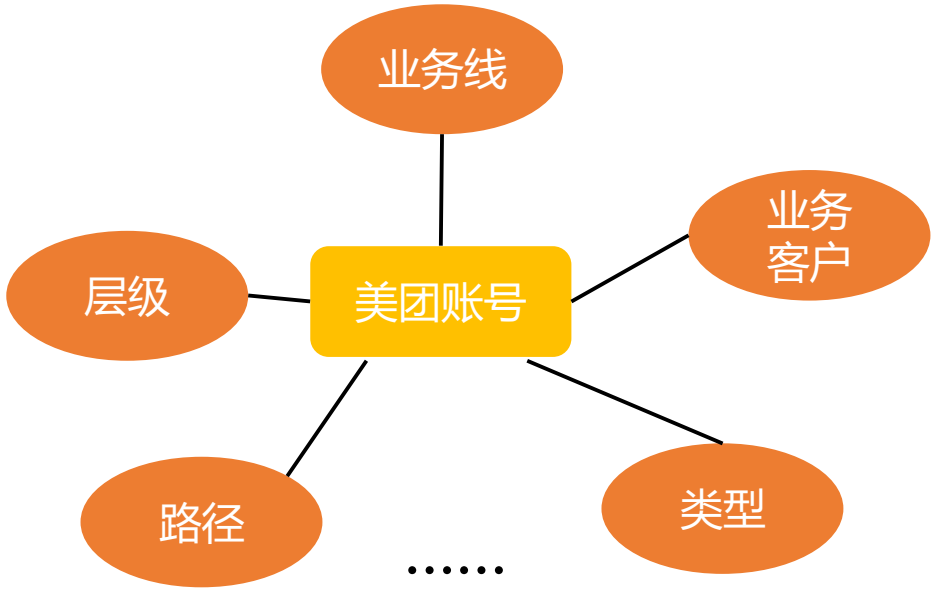

纽约客酒仙桥店
核码验证
扫码验证
买单收款
订单管理
¥241.50 用户支付成功，下午02:18
收银机数据
12345.67
今日实收
234
12/25
3
富保鸡丁
美团点评线上数据
今天
昨天
近7天
2473.52
3473.52
团购
542.52
657.28
智能支付
185.44
256.37
点餐
185.44
256.37


子账号
影响展现
权限A
权限B
权限C
主账号
美团
商家授权
分配管理权限
业务管控
制定业务规则

账号权限模型 – 账号层级



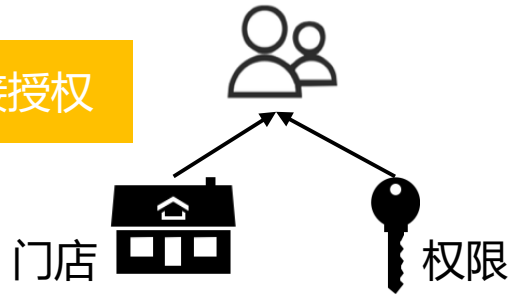
抽象



- 支持业务线隔离
- 账号树形结构满足业务不同层级诉求

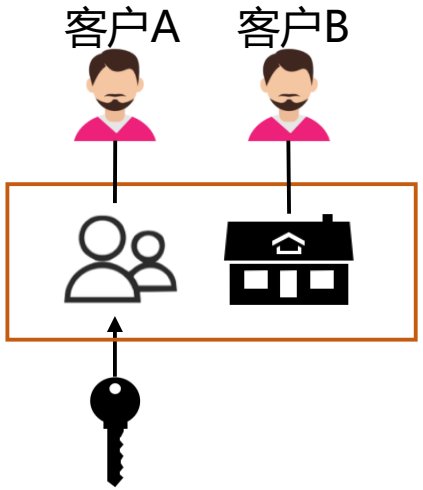
账号权限模型 – 授权

直接授权



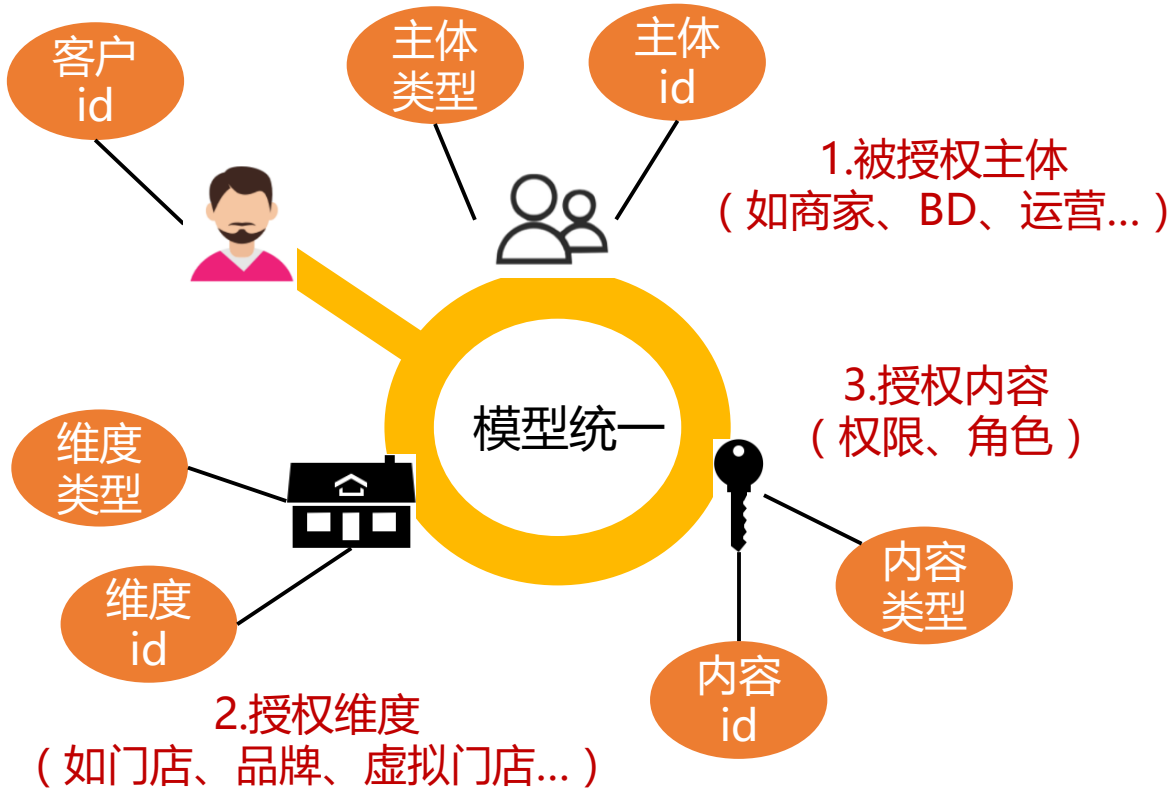
指定维度授权

跨客户授权



抽象

授权方式多样：需要抽象授权模型，避免模型爆炸

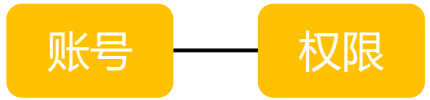


抽象出授权三元组 + 跨客户

账号权限模型 – 鉴权-访问控制模型

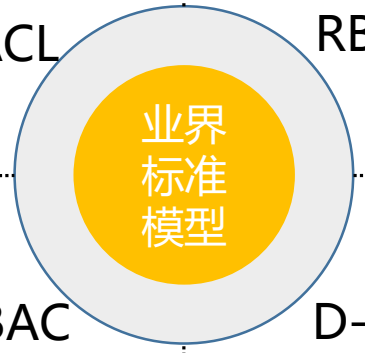
基于权限：直接关联权限

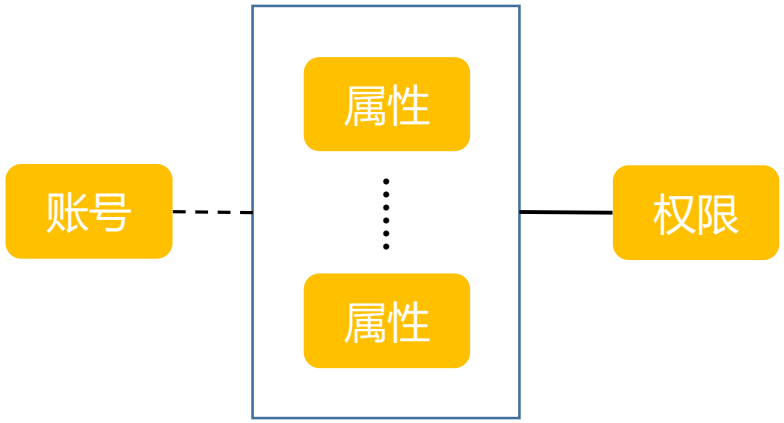
例：账号A直接拥有财务管理权限



基于角色：通过角色间接拥有权限

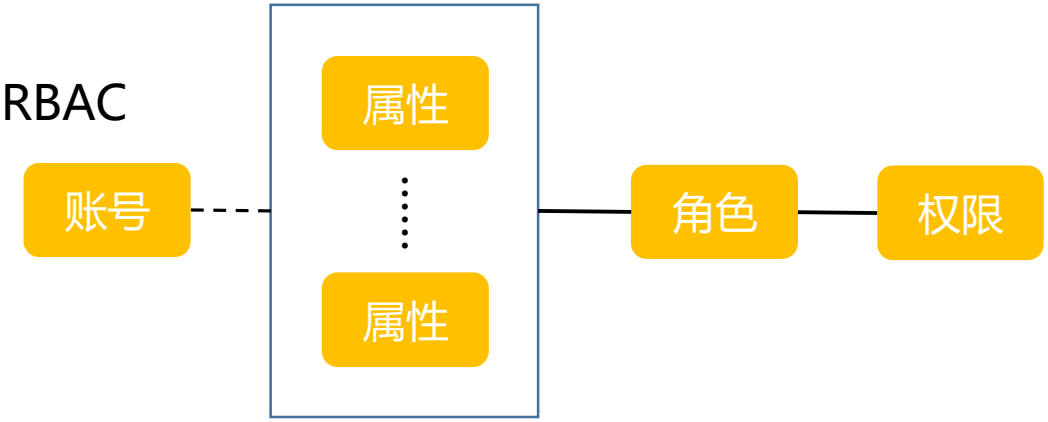
例：账号B是店长角色，而店长角色拥有财务管理权限





基于属性：满足属性条件拥有权限

例：门店在北京并且有在线套餐拥有套餐管理权限



动态角色：满足属性条件拥有角色，间接拥有权限

例：账号D拥有交易管理角色，该角色如满足管理门店在北京并且有在线套餐则拥有套餐管理权限

账号权限模型 – 鉴权 – 访问控制模型

业务实际情况：

简单场景：单模型

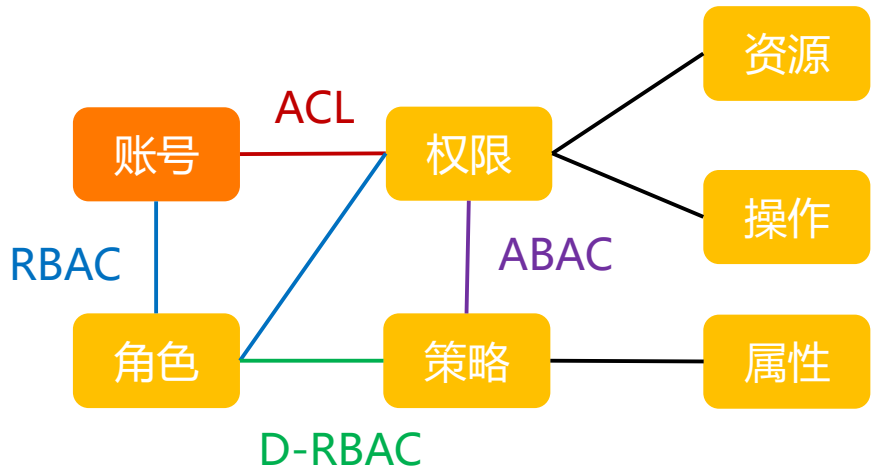
- ACL
- RBAC

复杂场景：模型组合

- ACL ∩ ABAC
- RBAC ∩ ABAC
- RBAC ∩ D-RBAC

抽象

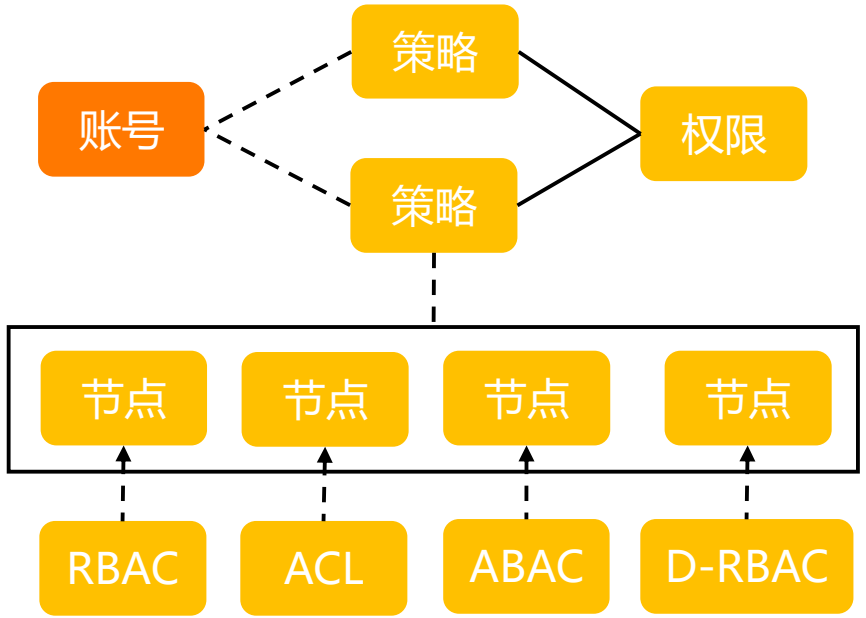
模型统一：



支持全部访问控制模型，支持模型组合

抽象

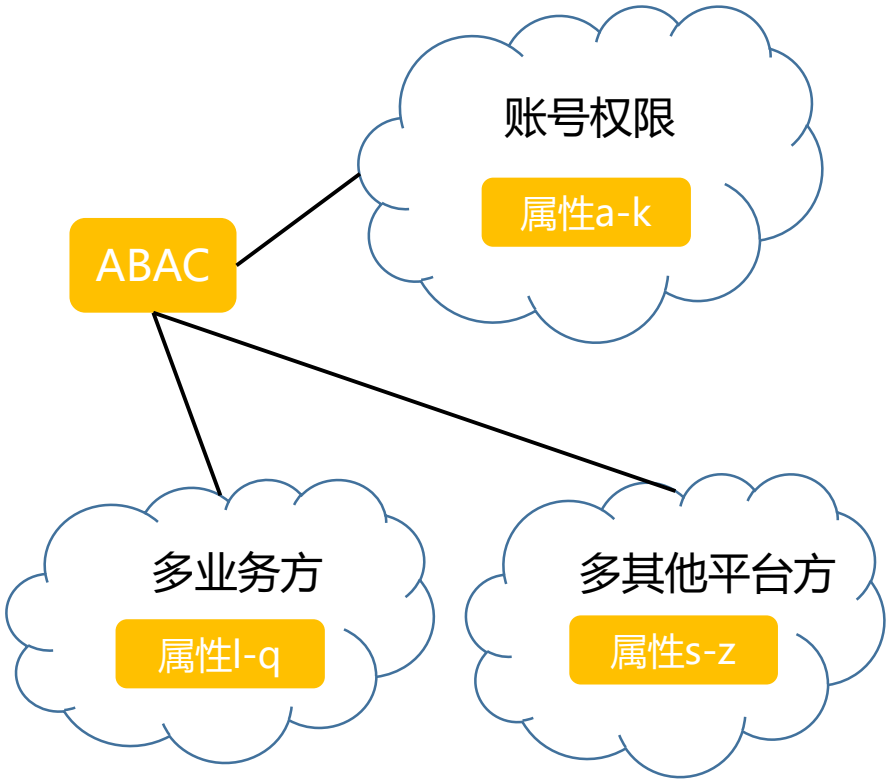
1个权限会有1个或多个策略



结果策略：

- 短路策略：满足继续执行，不满足拒绝
- 拒绝策略：满足拒绝，不满足继续执行
- 通过策略：满足通过，不满足继续执行

账号权限模型 – 属性

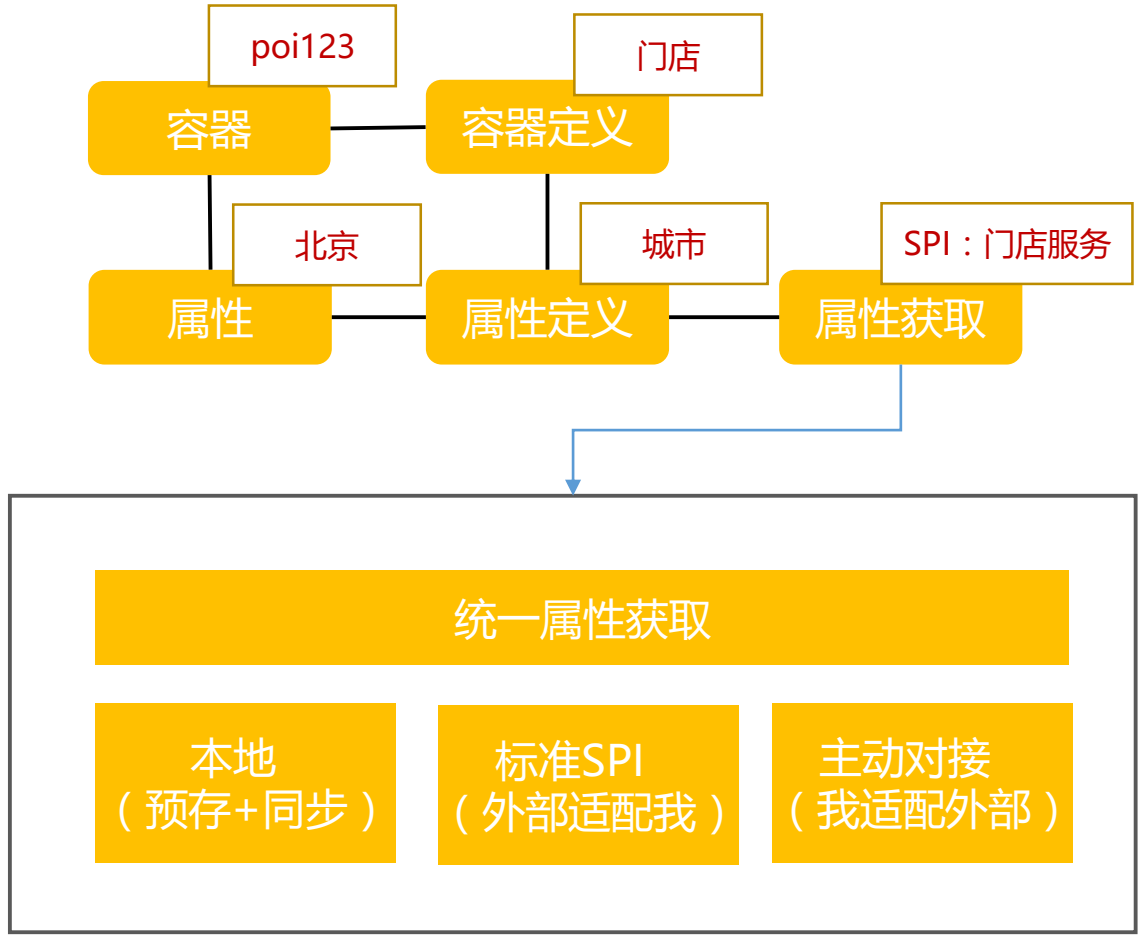


- 业务属性多样：避免模型爆炸
- 属性维护方多样：避免硬编码

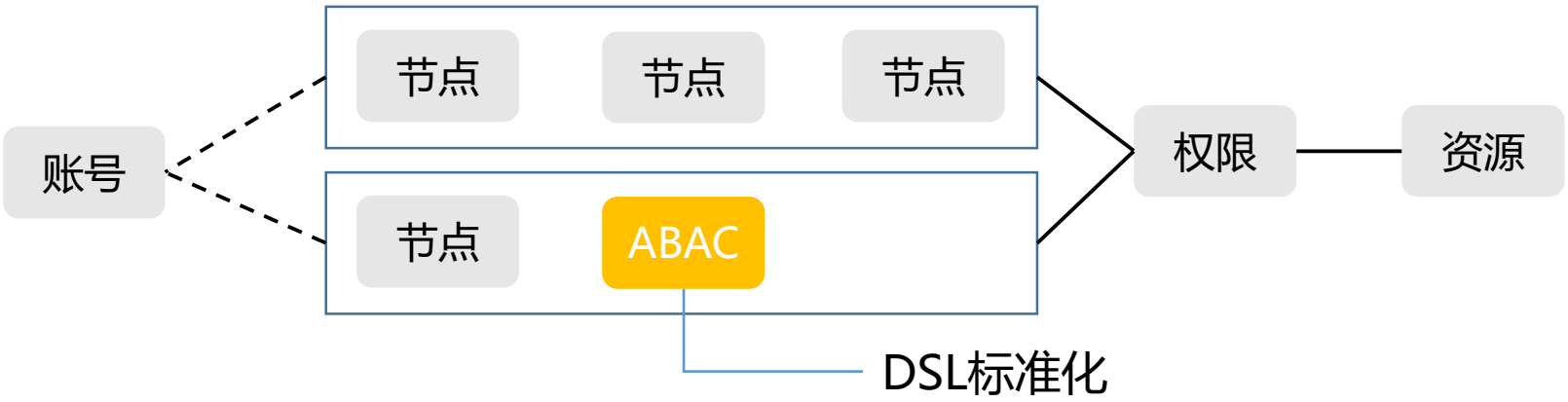
抽象

模型统一抽象

每个属性都有一个所属容器，如账号、客户、门店等。以门店123的城市属性为例。



账号权限模型 – 鉴权如何使用属性



实现配置化

属性获取

获取协议标准化：容器+属性+容器实例

"city_id":"poi/city_id/poi_id=${this:context:poiId}"

容器　属性　容器实例

属性判断

条件判断标准化：表达式+组合模式

```
"condition": { "expressions": [{
    "leftStatement": "${this:node-1:city_id}",
    "method": "NumberEquals",
    "rightStatement": 2
    }],
    mode": "AND
}
```

属性值　判断　预期值　组合模式

账号权限架构 – 整体功能分层与模块划分

接入层：

- 聚焦业务接入效率

能力层（核心）：

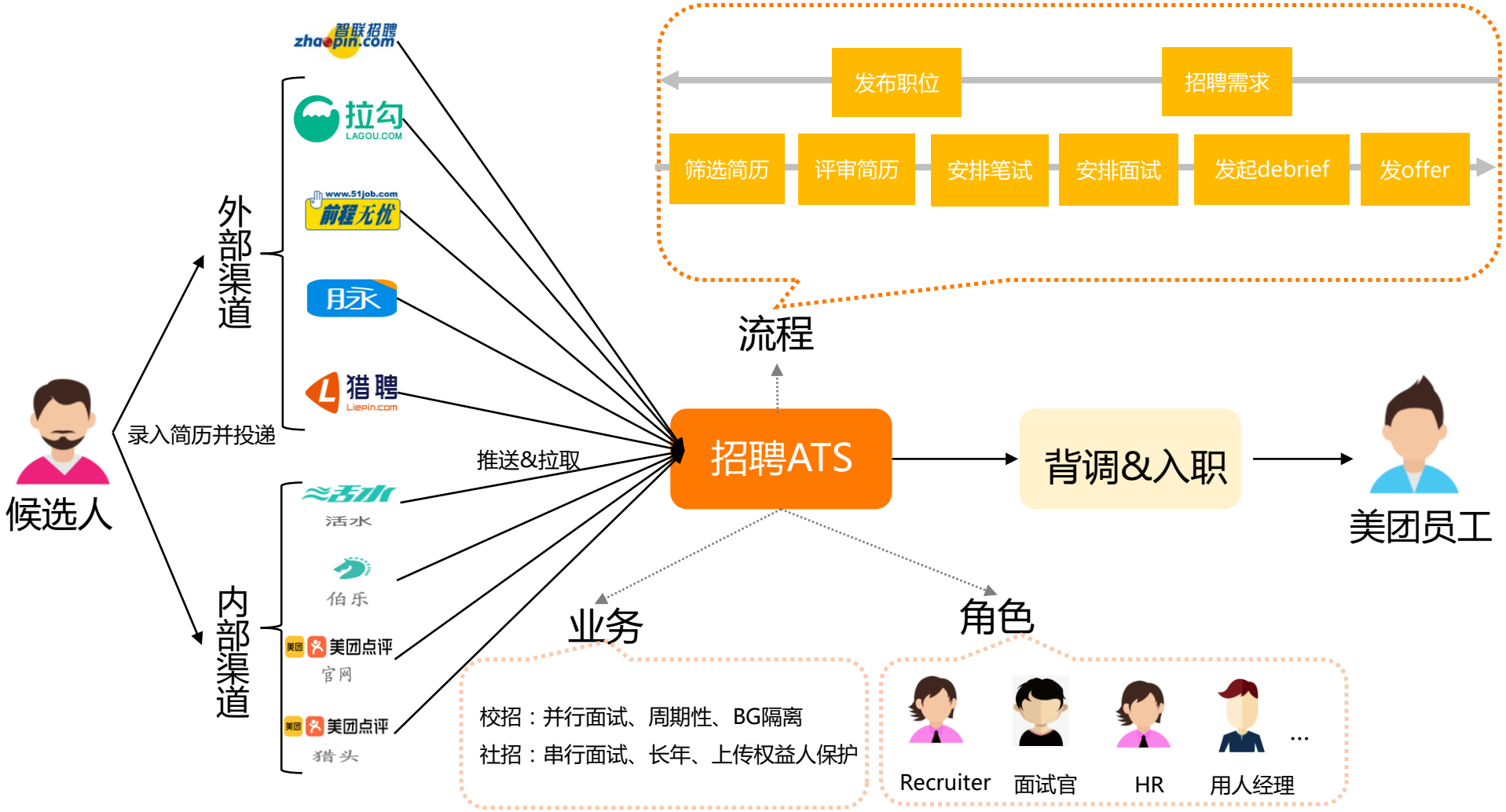
- 聚焦领域能力
- 聚焦领域边界与协作

基础层（支撑）：

- 聚焦数据规范与丰富
- 聚焦数据打通与跨业务线

课程目录

招聘业务

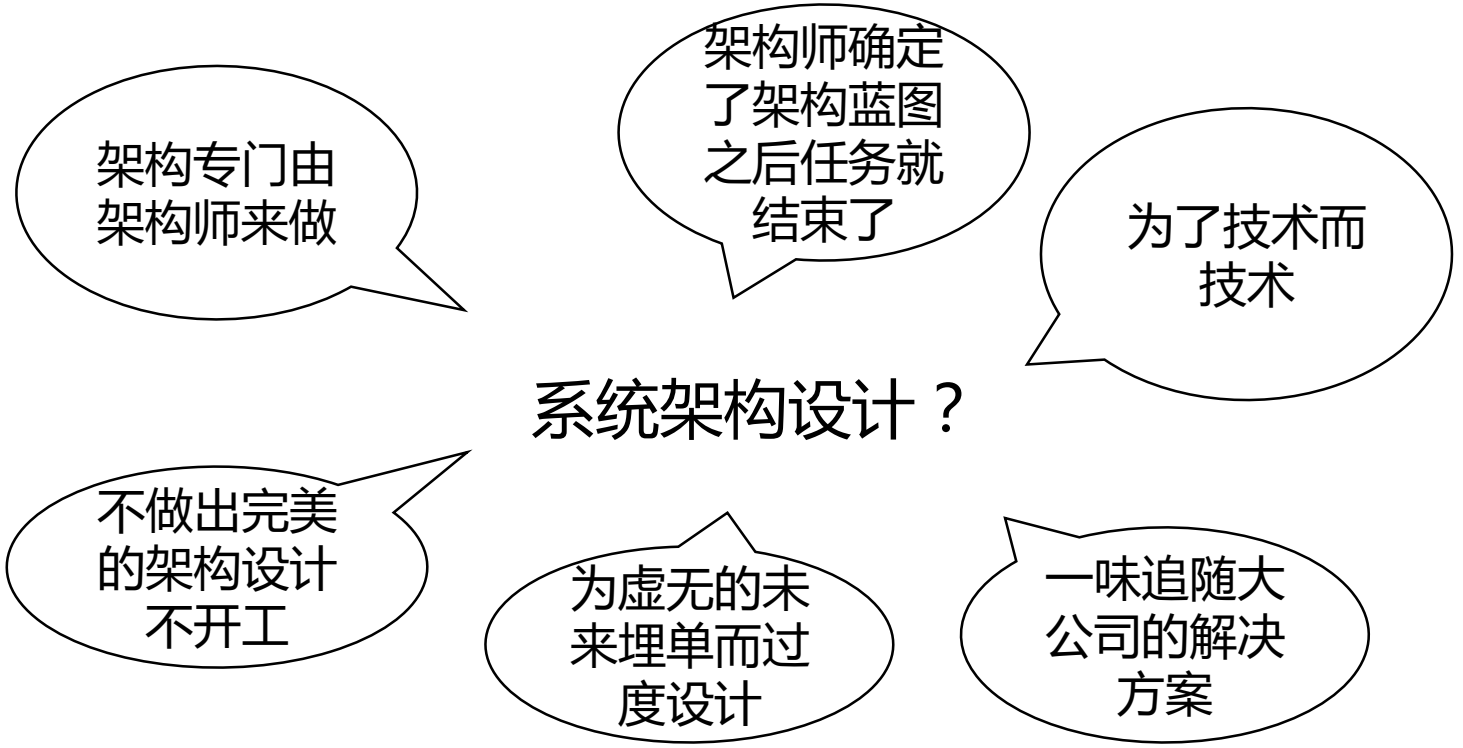


- 流程链路长
  - 提需求/发职位
  - 筛简历/做评审
  - 面试/笔试/Debrief
  - offer审批/入职
- 业务规则复杂
  - 对接渠道多
  - 岗位类型多
  - 校招/社招差异大
- 角色多
  - 对内9个角色
  - 对外3个角色

复杂业务系统常规应对策略 – 系统架构设计

业务系统复杂度不断提升

系统架构设计越发重要



遵循“合适、简洁、可演进”的原则

开发成本与收益比

架构设计选型

分层架构

核心概念是管理依赖，隔离每层关注点。如使用测试驱动开发会有更好的健壮性。

优势：每层关注点隔离，健壮性好

缺点：容易导致代码库难以维护

事件驱动架构

分布式异步架构模式，用于创建可伸缩的应用程序。

优势：自适应能力强（优于微服务架构）

缺点：实现相对复杂，事务难管理

微服务架构

业务逻辑和处理流程的服务组件

可伸缩、易于部署和交付的独立部署单元。

优势：异构连接和松耦合

缺点：团队沟通过载，有额外的网络开销和性能损耗

响应式架构

事件驱动架构的升级，系统对用户的请求即时做出响应，依赖异步的消息传递，淘宝主推。

优势：耦合低，跨模块好用，消息模式集成其他语言比较轻松

缺点：实现复杂度更高

插件架构

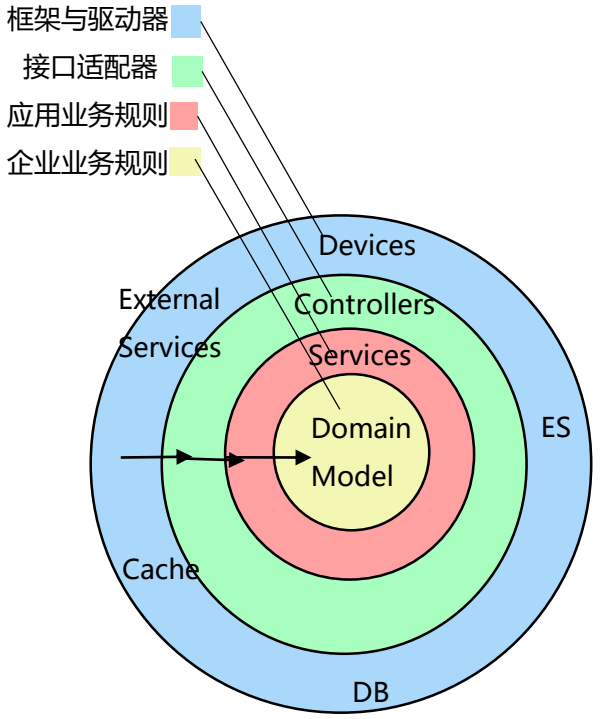
由核心系统和插件模块组成，通常包含最小的业务逻辑，

优势：插件批次独立，易解耦。

缺点：设计难度大，扩展性考虑复杂

。 。 。 。 。 。

招聘系统架构演进 – 1.0（分层架构）



分层架构

框架和驱动器:

负责对接外部资源，不属于系统开发的范畴

接口适配器:

用于打通应用业务逻辑与外层的框架和驱动器，实现逻辑的适配以访问外部资源

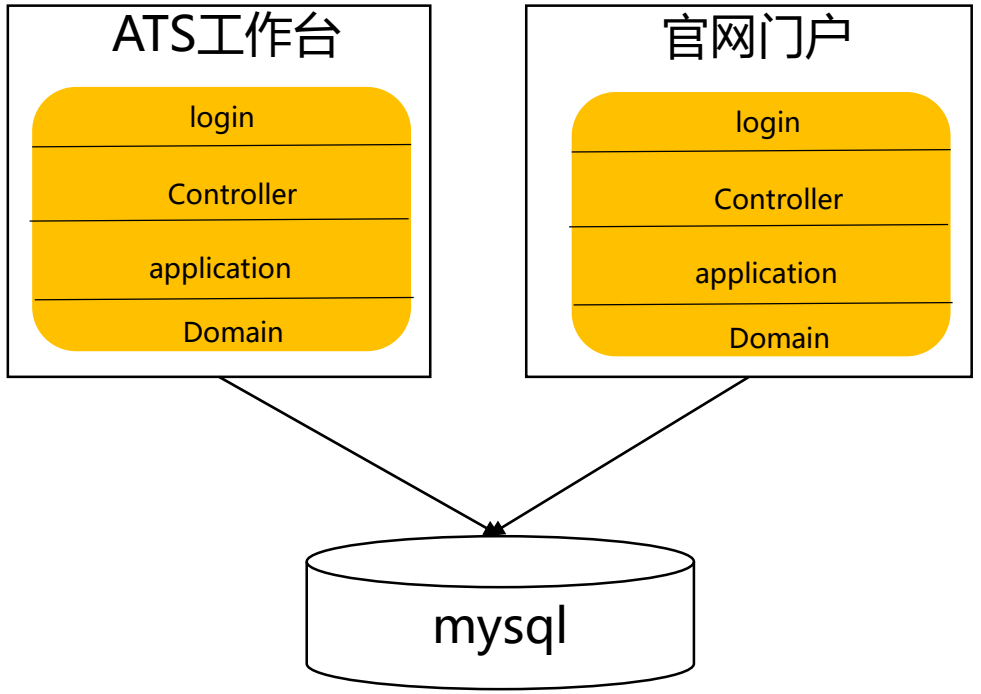
应用业务规则：

是打通内部业务与外部资源的一个通道，因为提供了输出端口与输入端口，但它对外的接口展示的其实是应用逻辑

企业业务规则:

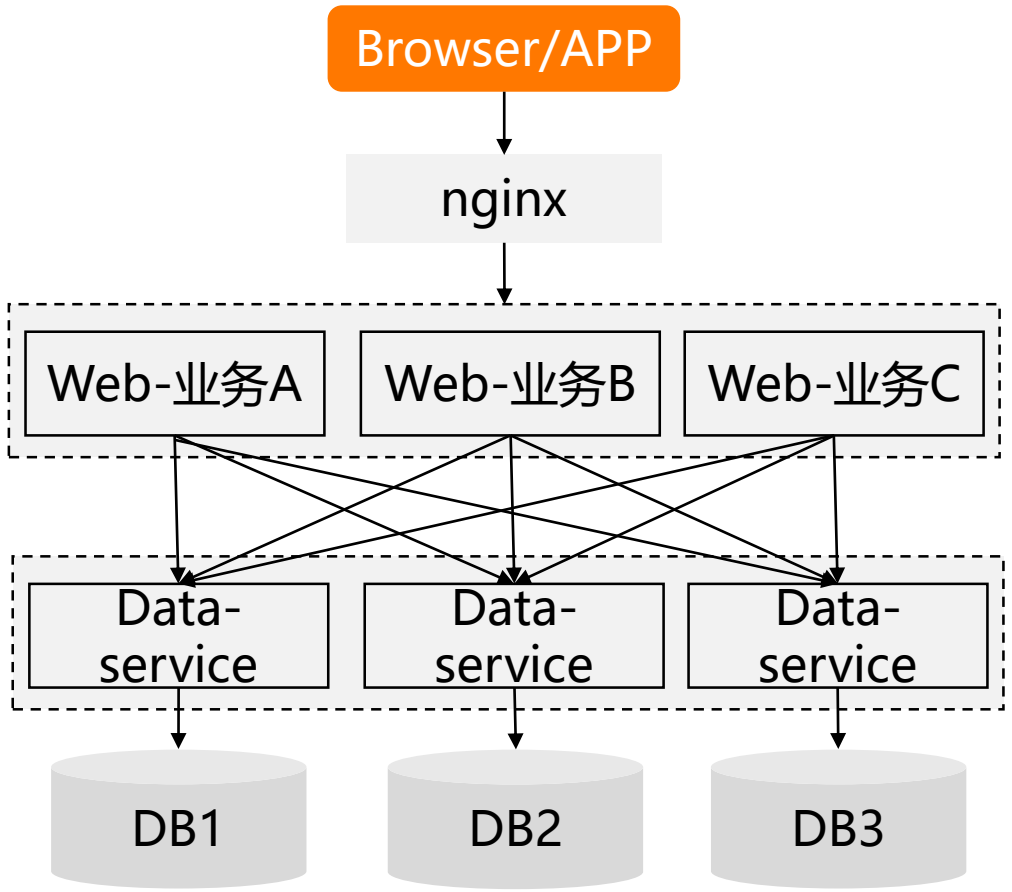
面向业务的领域模型

应用



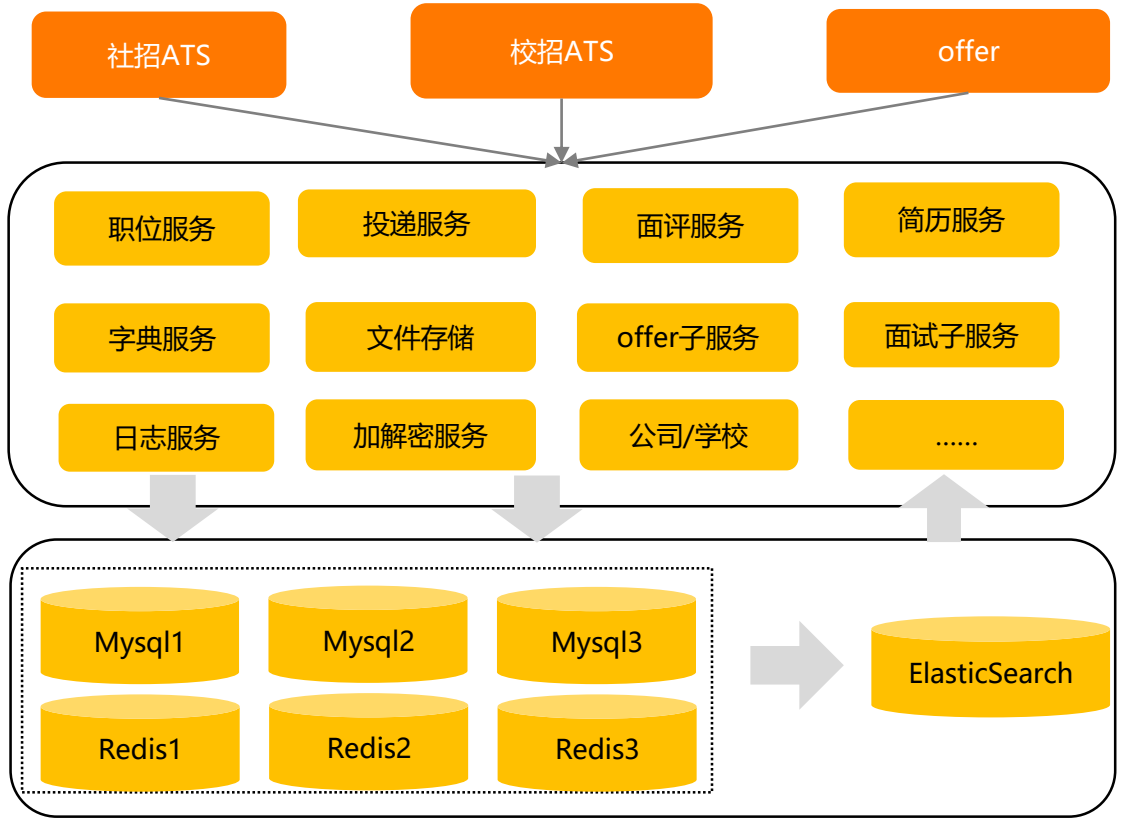
解决痛点	1.0架构选择	效果	问题
无系统可用 从0到1，快速上线	分层架构 单库	完成初期探索， 快速支持业务需求	随着需求的不断新增，代码复用率低，系统已经很臃肿

招聘系统架构演进 – 2.0（微服务架构）



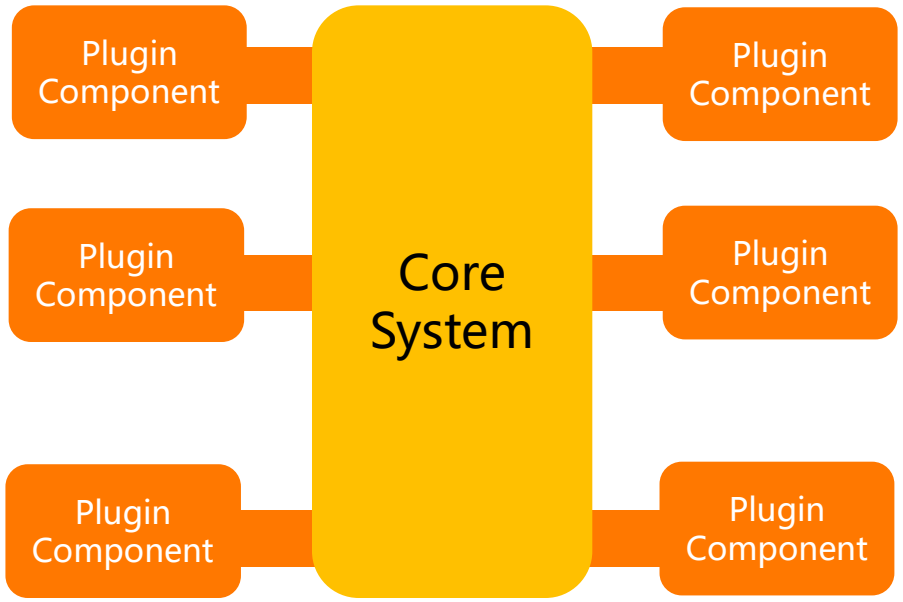
微服务架构

应用



解决痛点	2.0架构升级	效果	问题
社招/校招融合	微服务架构	投递成功率：98%	微服务数量太多，
场景复杂，性能低下	分库/读写分离	性能：300ms	开发效率降低

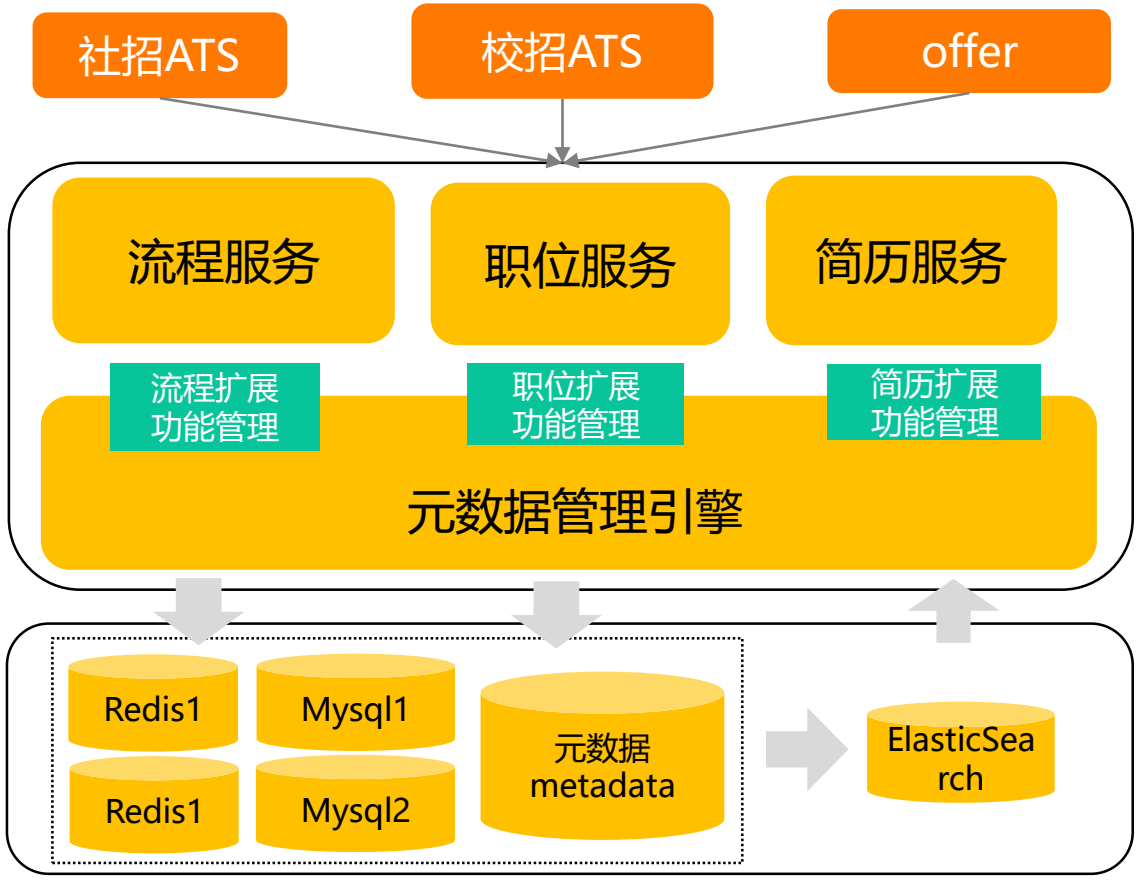
招聘系统架构演进 – 3.0（微服务+插件架构）



插件架构

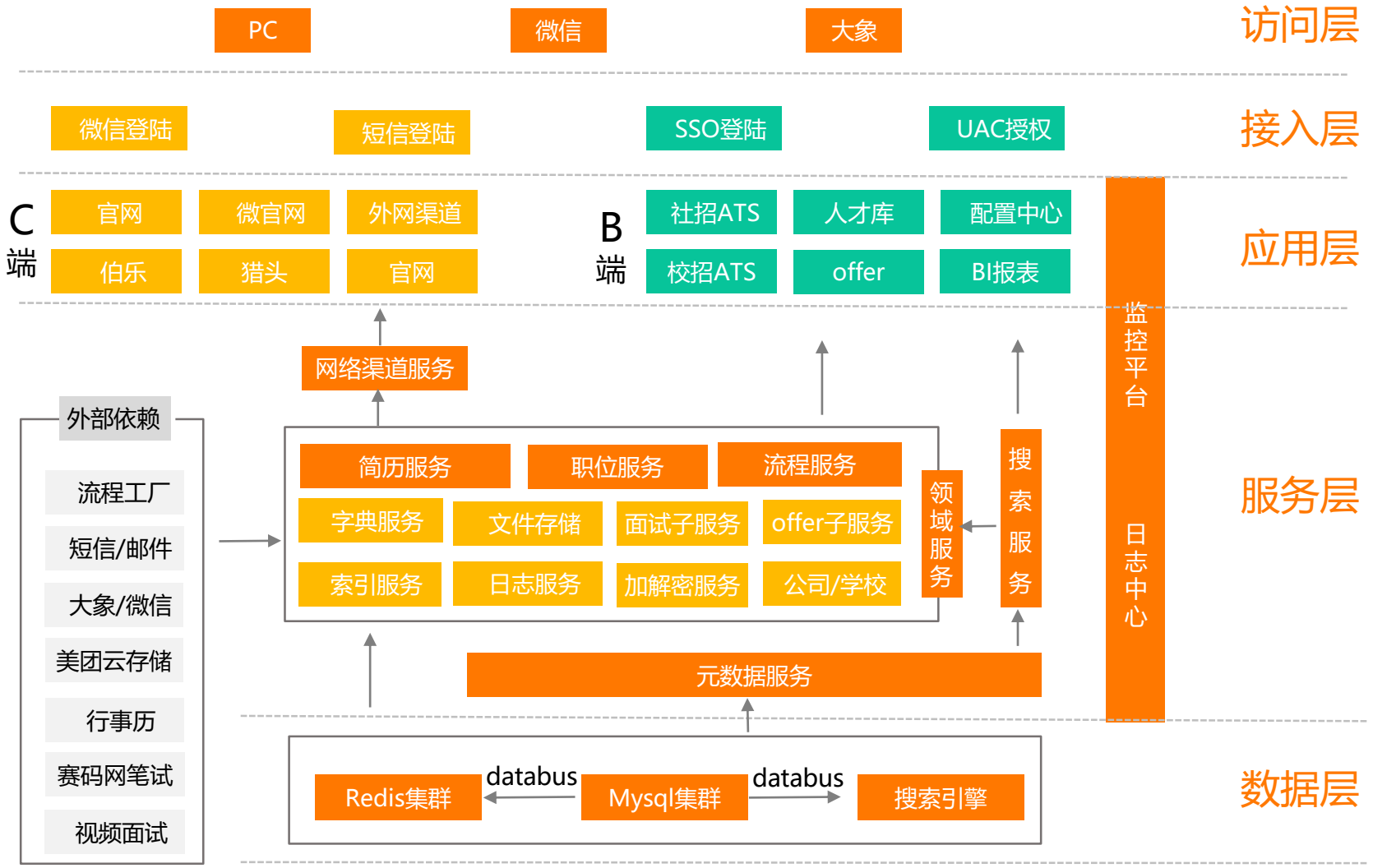
（拆分出不变和易变）

应用



解决痛点	3.0架构升级	效果	问题
用户越来越多，不同用户的需求满足效率低下	微服务架构+插件架构	60%的定制需求可配置 开发时长：10PD → 0	平台化标准功能，用户体验不好

招聘系统 – 整体架构图


PC
微信
大象
访问层
聚焦用户体验
微信登陆
短信登陆
SSO登陆
UAC授权
接入层
聚焦业务接入效率
C端
官网
微官网
外网渠道
伯乐
猎头
官网
B端
社招ATS
人才库
配置中心
校招ATS
offer
BI报表
应用层
聚焦权限控制及服务协作
监控平台
日志中心
网络渠道服务
外部依赖
流程工厂
短信/邮件
大象/微信
美团云存储
行事历
赛码网笔试
视频面试
简历服务
职位服务
流程服务
字典服务
文件存储
面试子服务
offer子服务
索引服务
日志服务
加解密服务
公司/学校
领域服务
搜索服务
元数据服务
服务层
聚焦服务边界
聚焦服务的复用
聚焦业务规则的处理
Redis集群
databus
Mysql集群
databus
搜索引擎
数据层
聚焦数据读写性能
聚焦数据一致性

系统通道相关推荐

1、 《设计模式》
ISBN:9787111618331

2、《企业应用架构模式》
ISBN：9787111303930

3、 《重构 改善既有代码的设计》
ISBN：9787115508652

4、 《领域驱动设计》
ISBN:9787115376756

1、《领域驱动设计基础》
https://mit.sankuai.com/activity/index.html#/13588

2、《企业架构设计》
https://ipu.sankuai.com/app/course/detail/157869540_157822739

互联网+大学 | 技术学院

Internet Plus University

谢谢！