ArchSummit 全球架构师峰会 深圳站 2016

滴滴出行业务系统的架构升级

关于 · 我

- 2015 年 5 月加入滴滴，第一个项目即负责公司级重构项目的技术架构设计
- 曾在微软和百度工作，后自主创业五年有余，深耕于游戏行业
- 熟悉前后端各种工程类技术栈，对各种新技术持续保持关注
- 喜欢对事物进行抽象，寻找其中优雅简洁的本质

大纲

- 挑战在哪里？
- 现状是什么？
- 该如何入手？
- 客户端怎么拆？
- Web App 怎么拆？
- 服务端 API 怎么拆？
- 效果怎么样？
- 如何避免重蹈覆辙？

滴滴出行

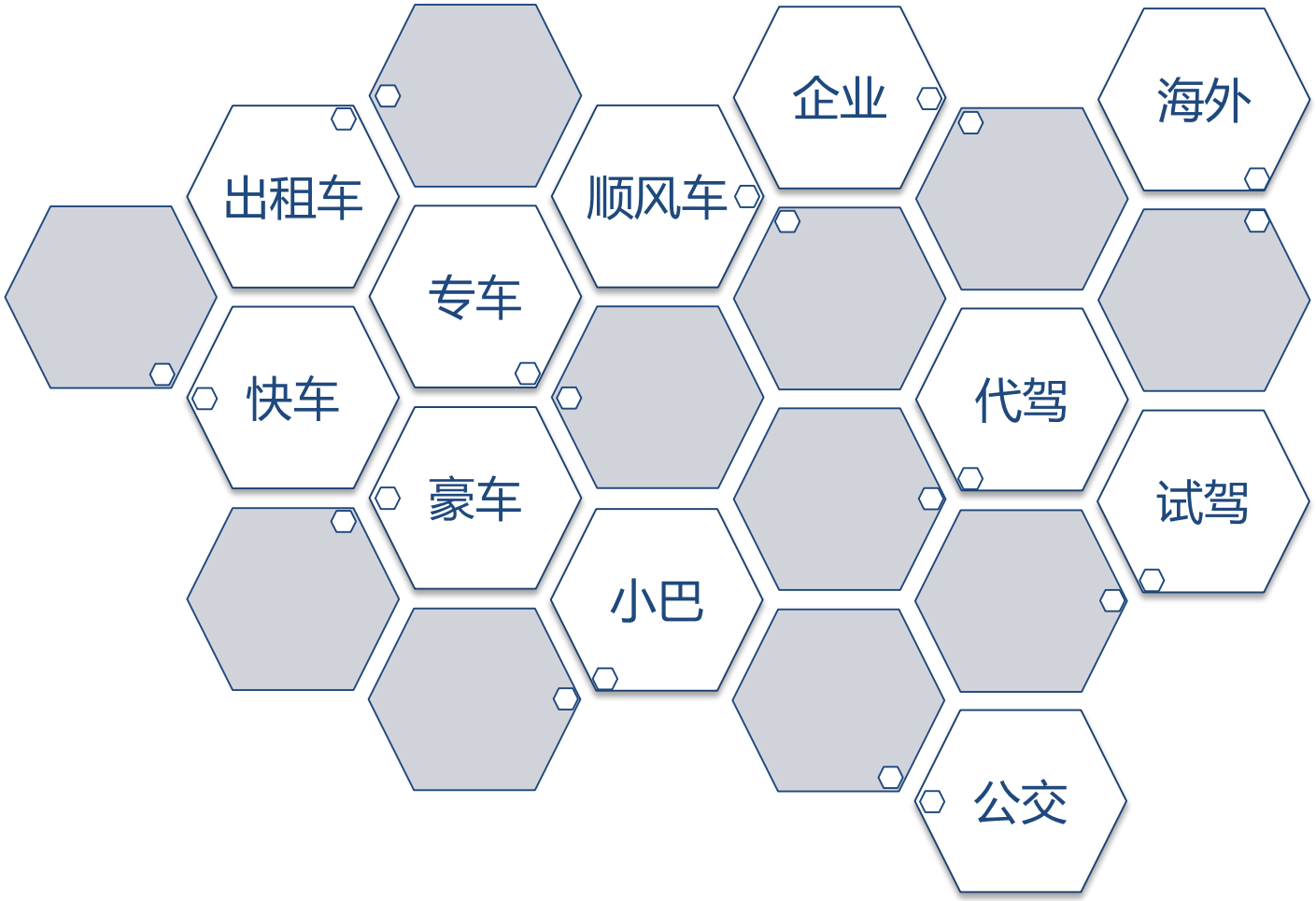
滴滴一下美好出行

挑战在哪里？

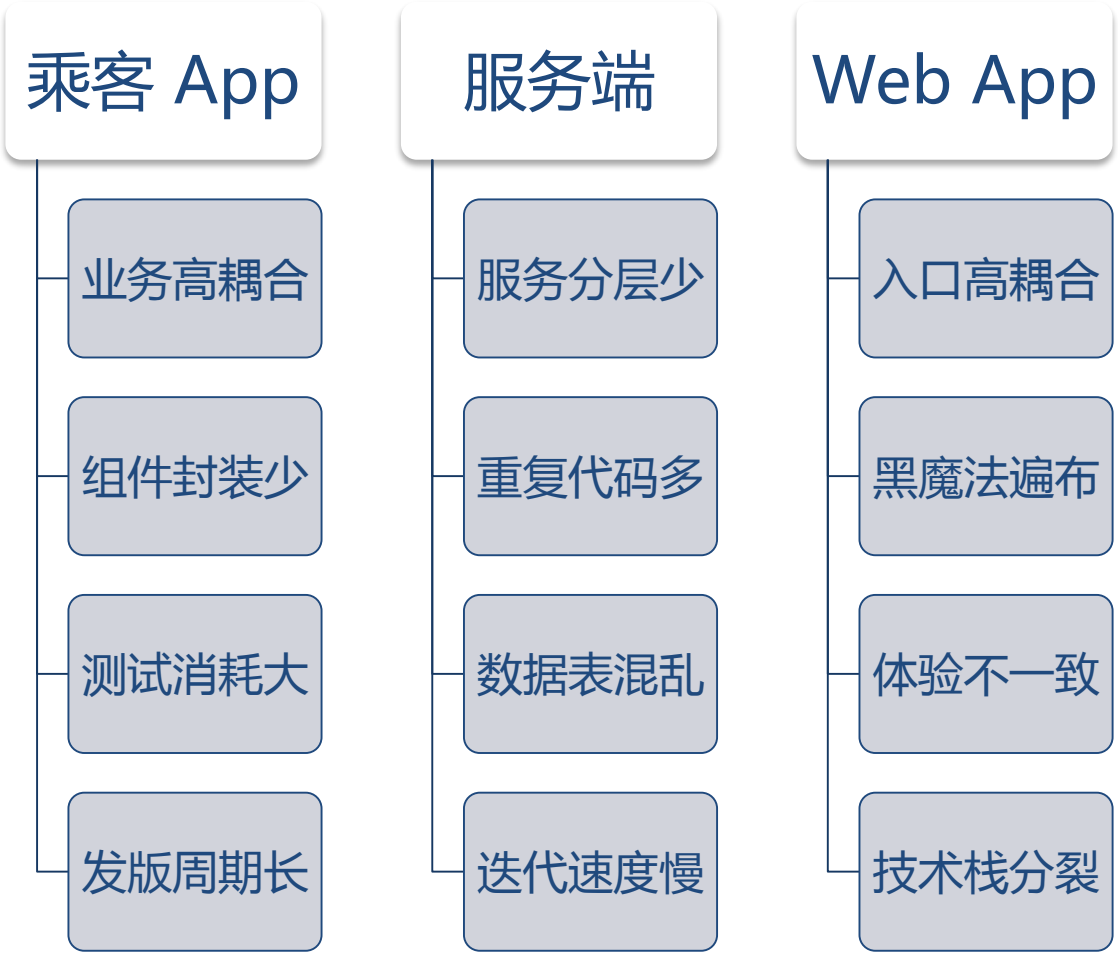
公司规模指数级增长


2016/6：累计融资超过 100 亿美元
2015/9：更名“滴滴出行”，全面出击
2015/2：滴滴和快的合并
2014/1：D 轮 7 亿美元及补贴大战
2012/12：A 轮 300 万美元
2012/6：公司成立

业务数量爆炸性增长

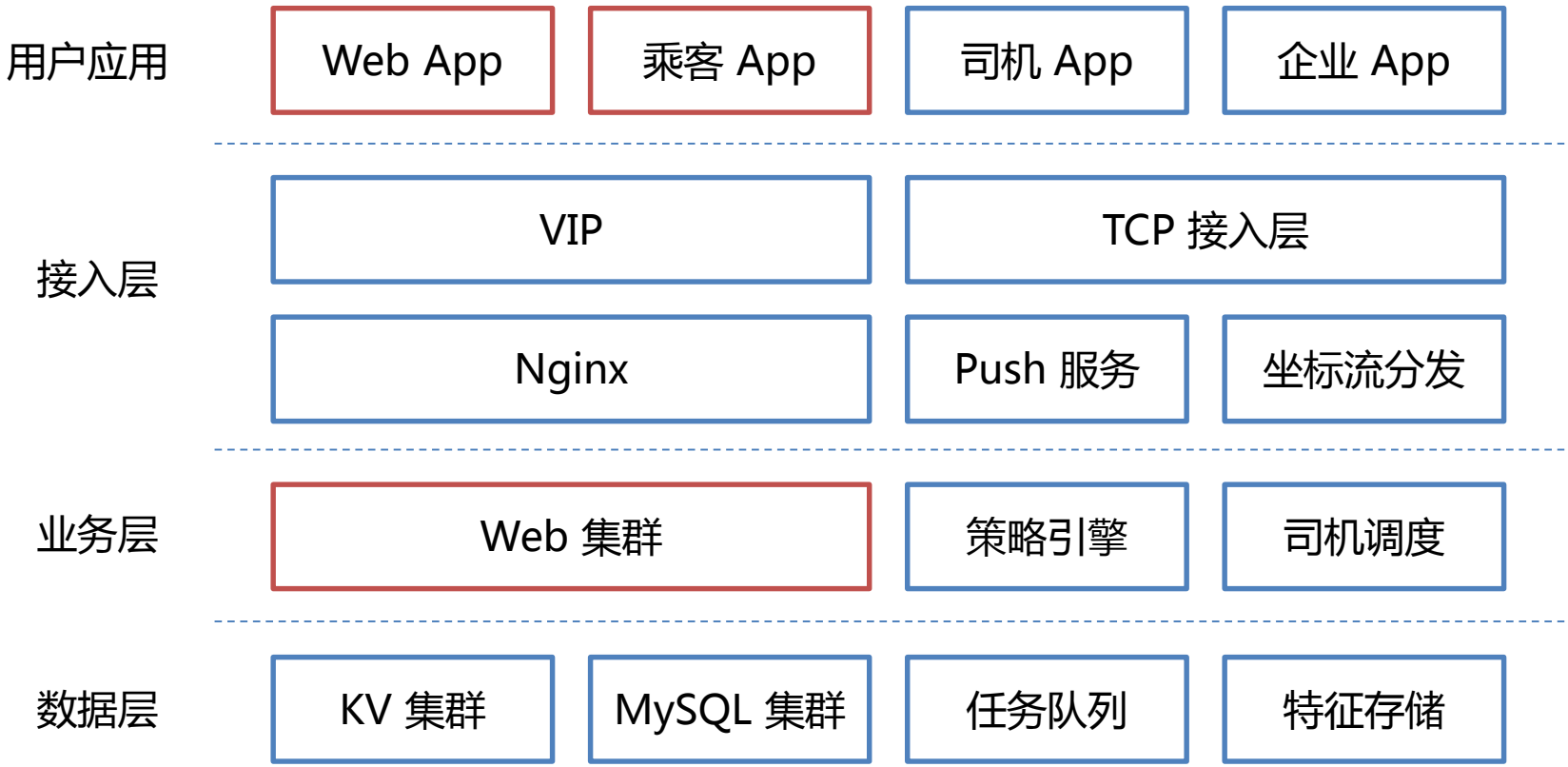


积压大量难以解决的问题



现状是什么？

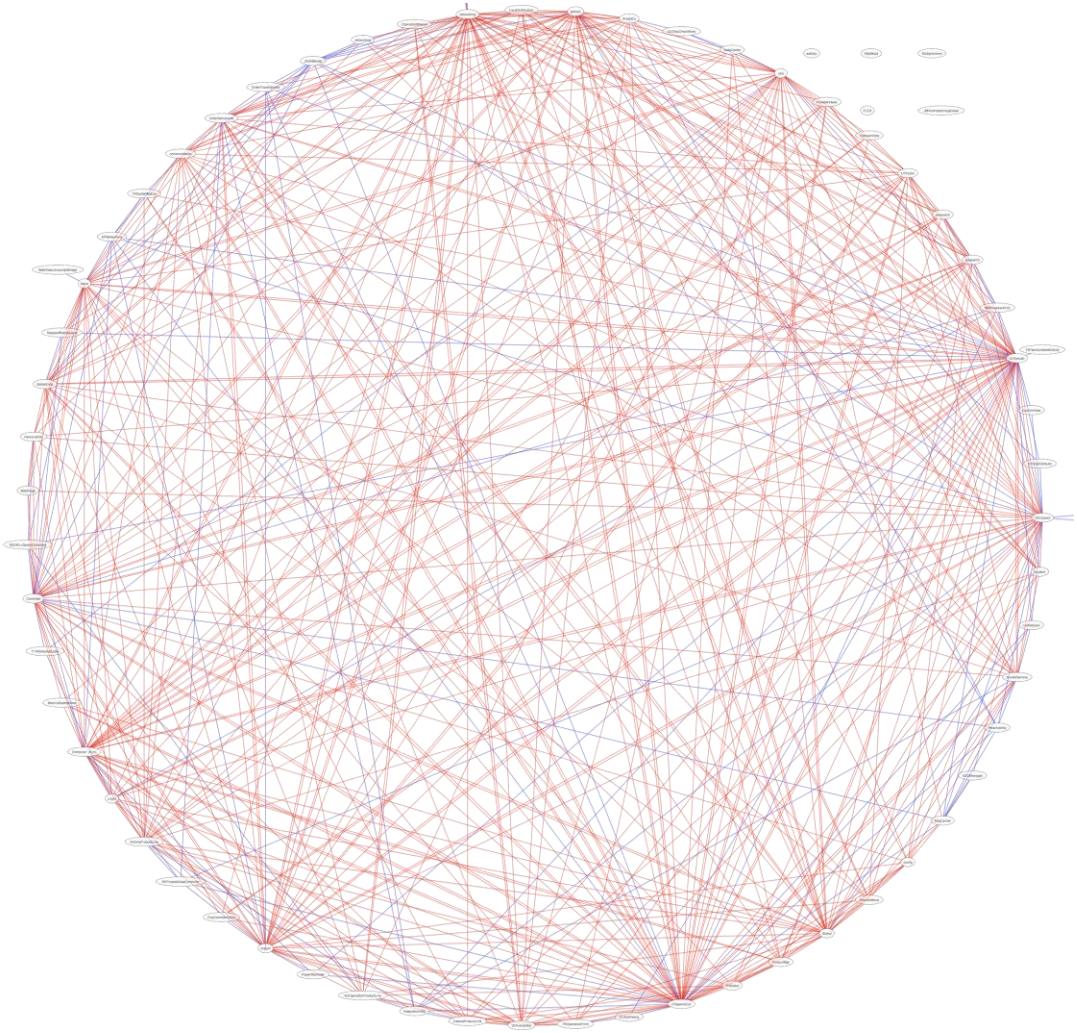
2015 年下半年系统架构



2015 年下半年乘客 App 结构

- 基本情况
  - iOS 曾经历过一次大规模重构，已经沉淀出一些公共组件，但是依赖关系过于混乱
  - Android 一直野蛮生长，没有公共框架和特别的设计

- iOS 模块依赖分析
  - 分析 #import 和 #include
  - 一个物理目录算作一个模块
  - 存在循环依赖，标为红色
  - 单向依赖，标为蓝色



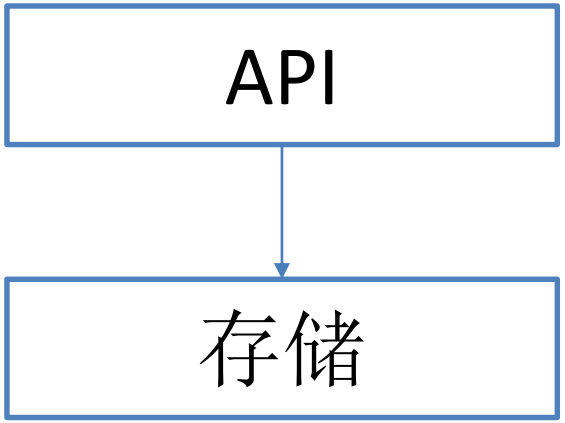
2015 年下半年 Web App 结构

- 基本情况
  - 由于历史原因，所有业务入口耦合在出租车代码中，业务修改入口需要更改出租车的仓库
  - 发送订单之后页面跳转到各个业务
  - 前端代码没有模块化，仅做了简单的代码合并
  - 所有渠道使用同一份代码，充斥着黑魔法
  - 没有公共组件，也没有机制来沉淀组件

重构前的 Web App 首页

2015 年下半年 API 结构

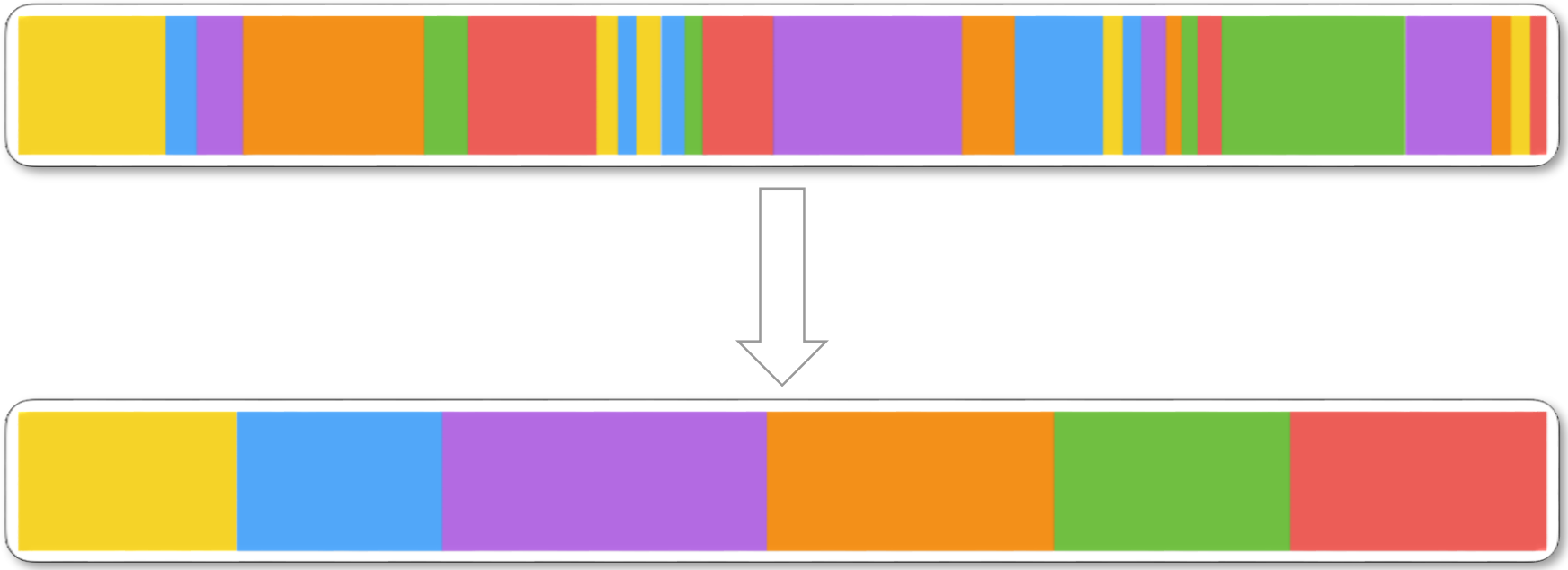
- 基本情况
  - API 层仅存在业务线级别的划分
  - 业务内缺乏模块划分，所有 API 混合在一个仓库中
  - API 和后台通过数据库直接共享信息，没有 model 封装
  - API 的日志没有统一规范，监控、大数据分析、反作弊等不统一
  - API 函数长度惊人，且存在大量重复代码
  - 快车 API 项目总代码量达到百万行



该从何入手？

重构思路

将类似的模块归类及合并，再逐步优化



从前到后、从粗到细

- v0.5：2015 年 Q3/Q4
  - 乘客 App 代码按业务拆分
  - Web App 首页代码按业务拆分
  - 提供各种方便组件下沉的构建流程和开发工具
- v1.0：2016 年 Q1/Q2
  - API 服务化改造
  - 平台服务下沉

核心关注点

代码治理 + 模块下沉

关键动作

按照产品逻辑，重新划分模块

将不同模块拆分到不同仓库之中

消除模块间的循环依赖

独立的模块开发、测试、上线流程

提供模块下沉所需要的工具和流程

全新的基于模块的构建系统

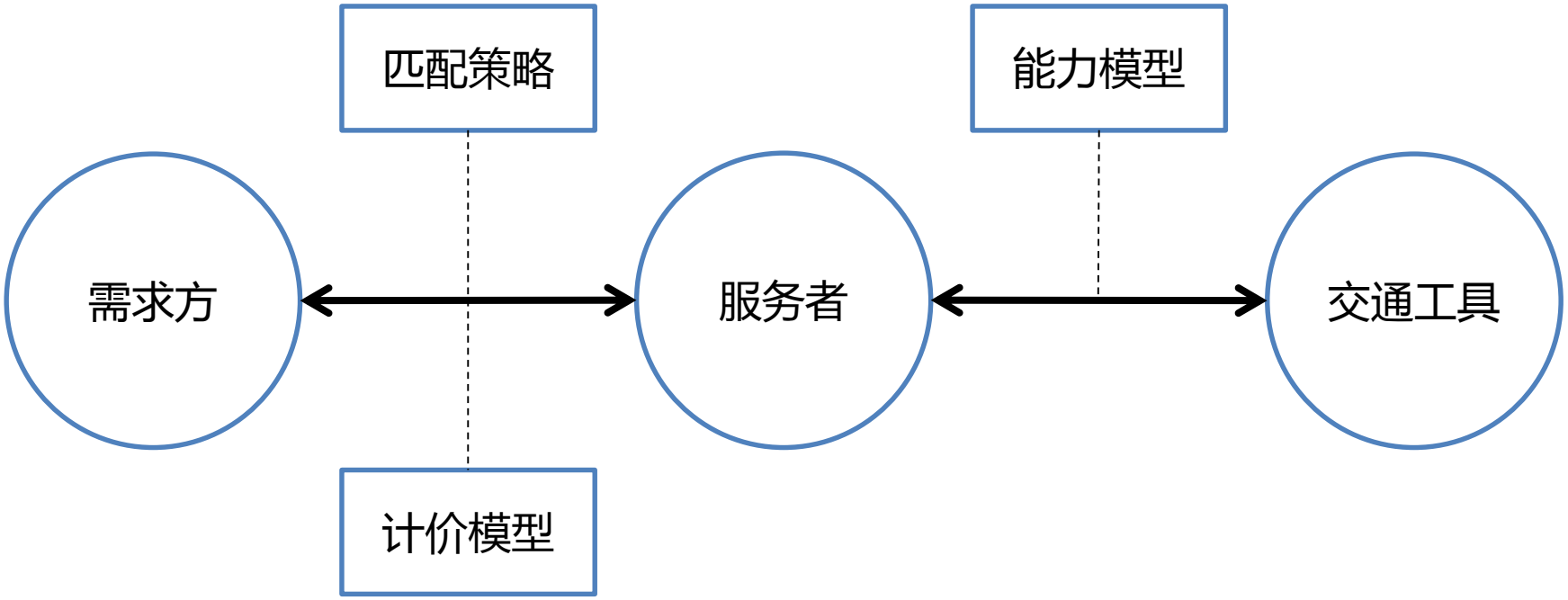
控制模块下沉所需成本

临时的全公司需求管控

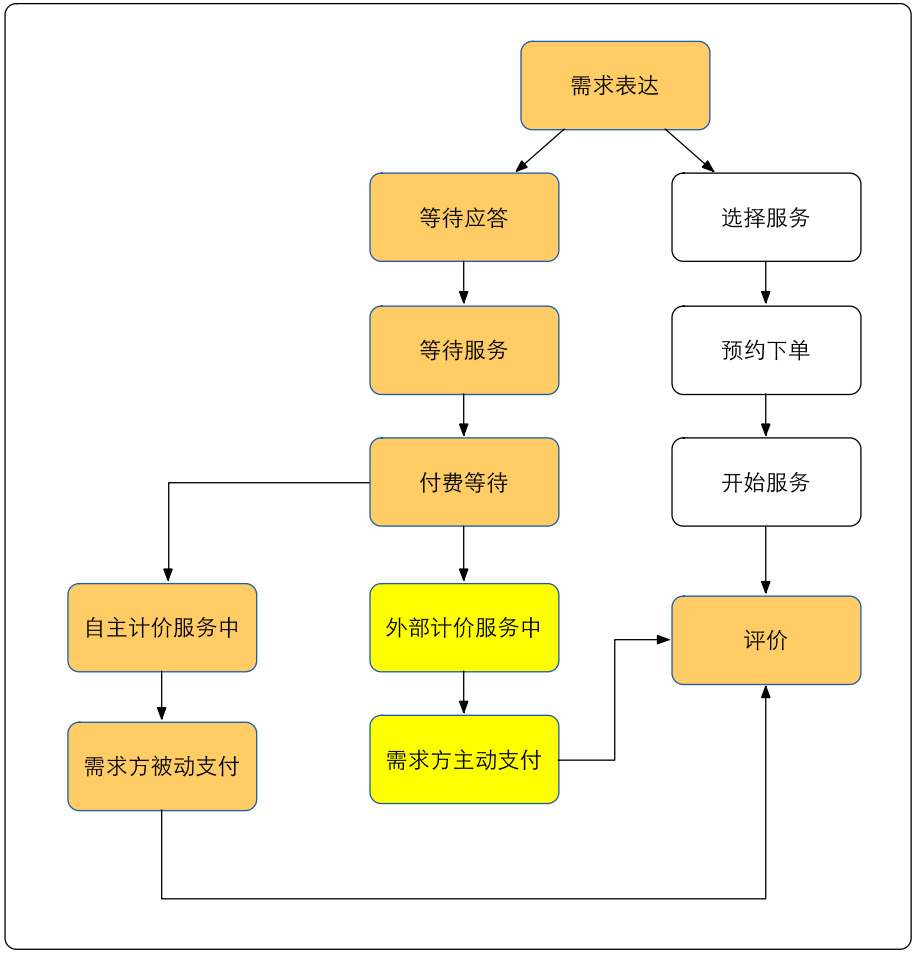
核心设计思想

	无状态	异步化
客户端	互相独立无依赖的界面流程	异步获取共享数据
Web App	业务互不干扰的并发加载	异步获取共享数据
服务端	易于扩容的无状态服务单元	用订阅 / 发布模型解耦

业务模型的理想形态



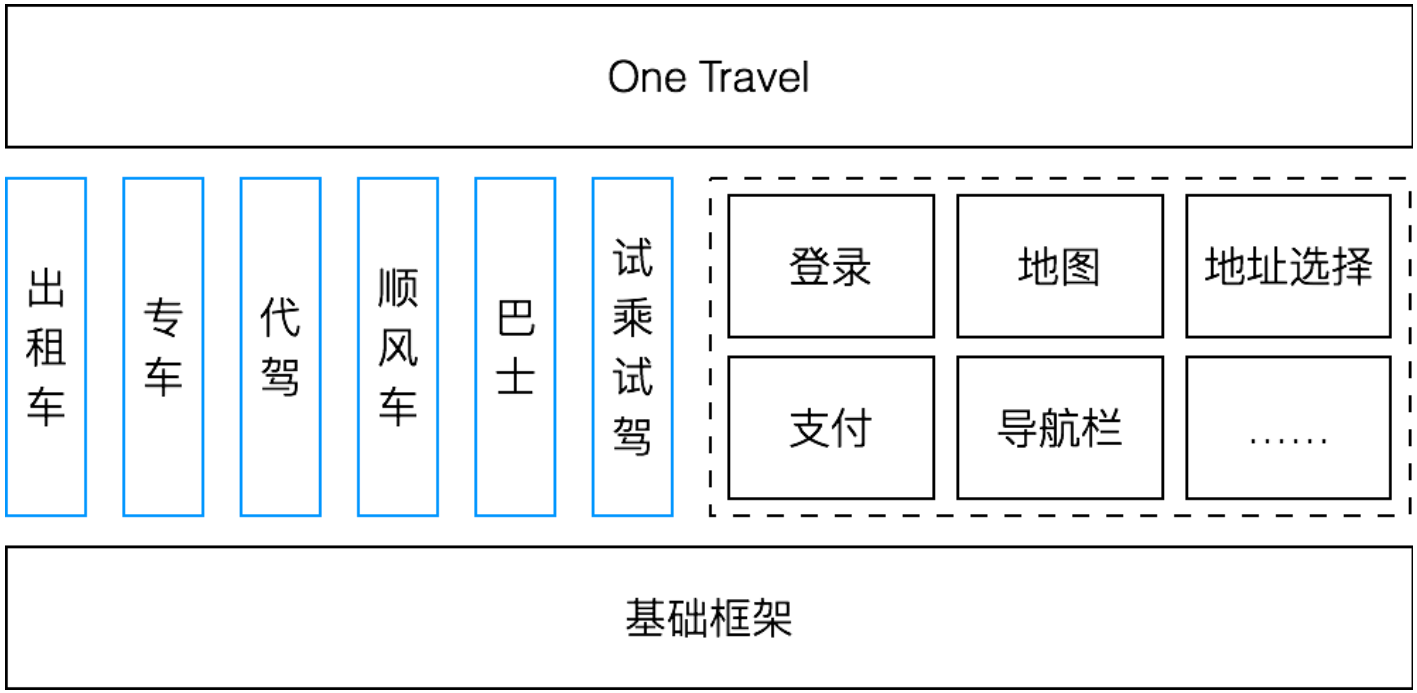
高度可配置的出行工具


需求表达
等待应答
选择服务
等待服务
预约下单
付费等待
开始服务
自主计价服务中
外部计价服务中
评价
需求方被动支付
需求方主动支付

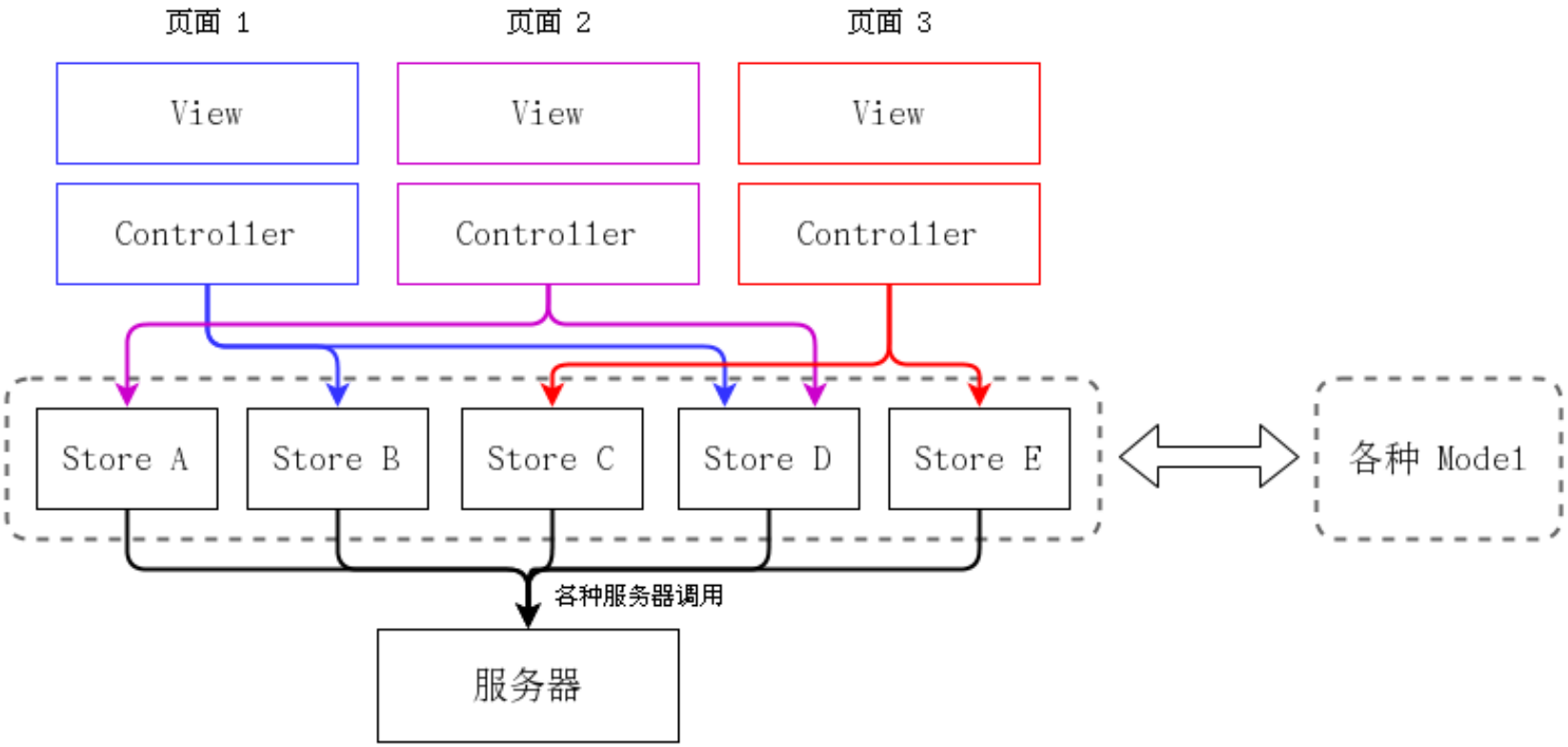
客户端怎么拆？

乘客 App 方案

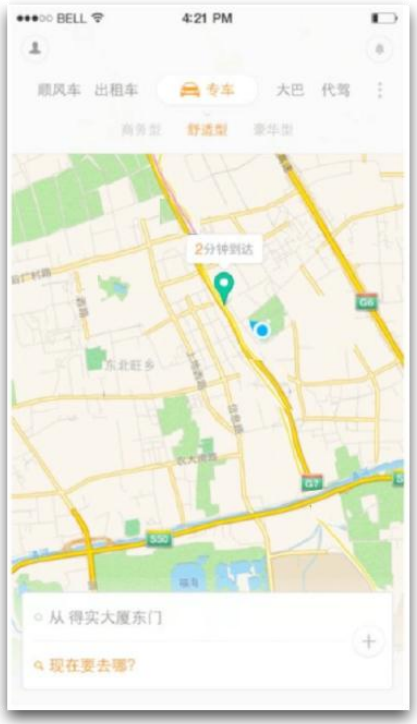
- 所有业务都作为独立仓库从主工程抽离出来，可单独运行
- iOS 采用 Cocoapods；Android 采用 gradle + maven
- 暂时不关注业务内部的代码质量问题，由业务自行演进，只通过制定一些规则来引导



乘客 App 跨页面解耦

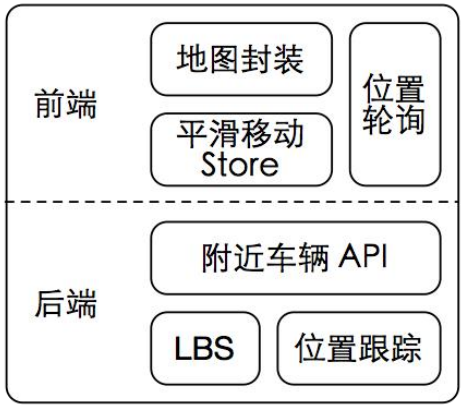

页面 1
页面 2
页面 3
View
View
View
Controller
Controller
Controller
Store A
Store B
Store C
Store D
Store E
各种 Model
各种服务器调用
服务器

乘客 App 组件化

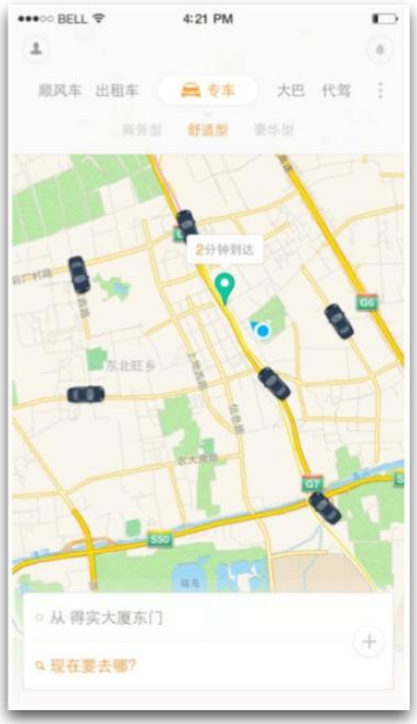


+

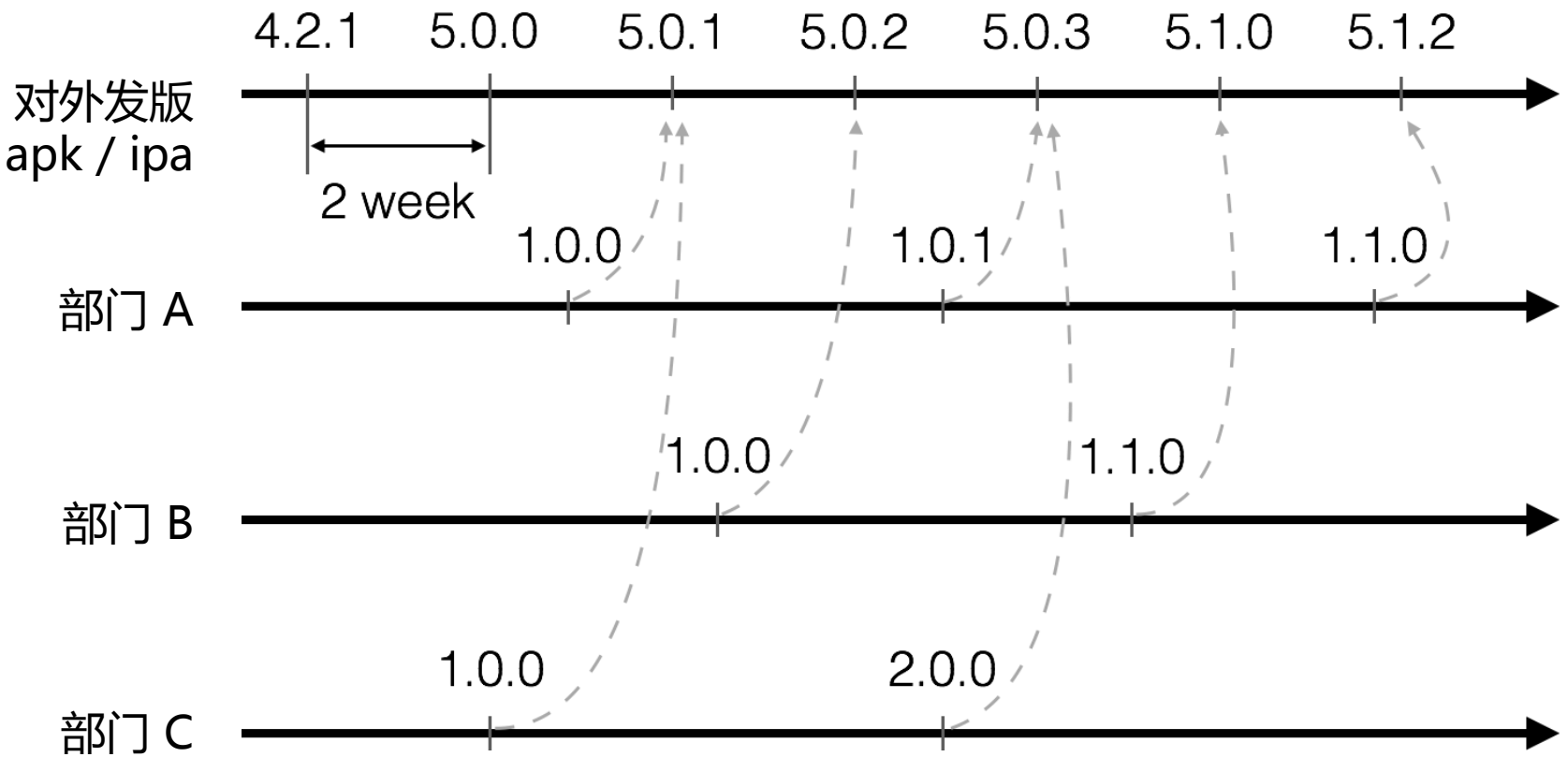
平滑移动组件



=



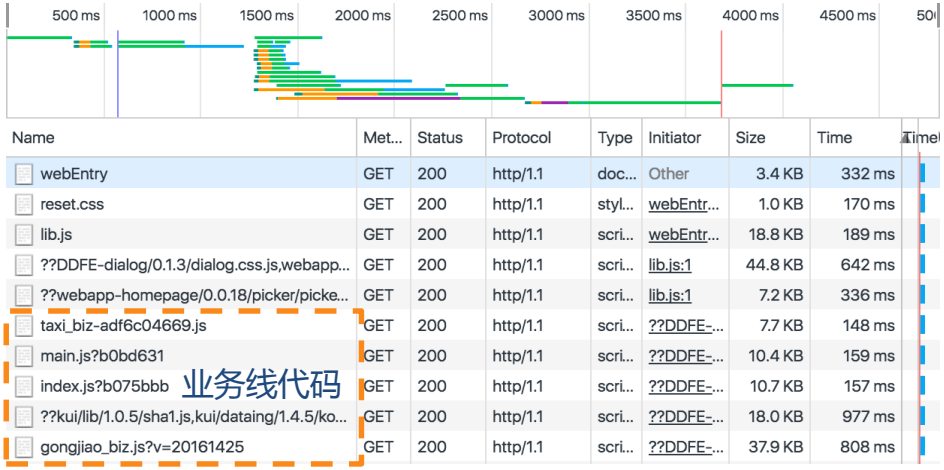
乘客 App 业务集成方案



Web App 怎么拆?

Web App 方案

- 实现一个新的业务框架，提供入口调用规范
- 用 scrat / webpack 管理组件依赖
- 动态加载业务代码，支持离线缓存和分级灰度
- 控制每个业务代码加载时间和未捕获异常，避免造成全局性灾难



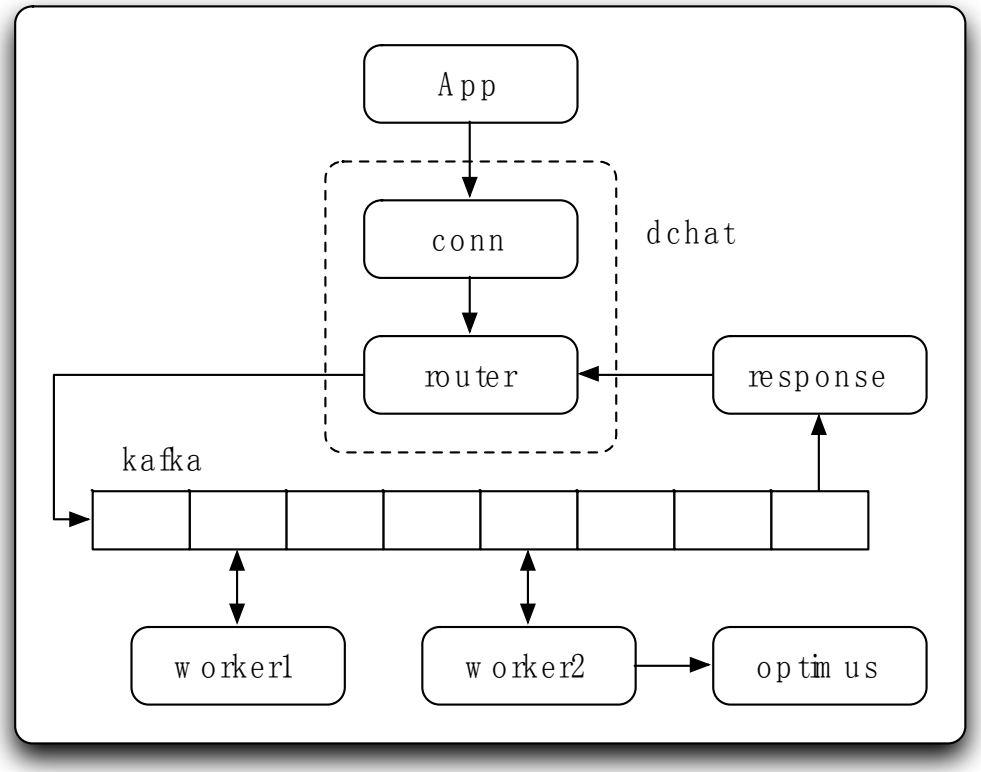
Web App 方案

- 实现一系列公共组件，提供贴近客户端的交互体验
  - 字体和颜色规范
  - 按钮
  - 导航栏
  - 弹层
  - 地址列表
  - 时间选择器
  - 进度条
  - ……

服务端 API 怎么拆？

服务器 API 拆分 Plan A：可拼接的业务组件

- 2015 年方案
  - 将业务按照抽象的可复用的组件进行拆分
  - 基于长连接的应答式可靠安全的自定义通信协议
  - 基于“消息”的订阅 / 发布模式，每一个 worker 都是一个最小业务单元，可通过配置服务随意拼接 worker 顺序、插入新 worker、复制流量
  - 统一的订单系统 optimus
  - 统一的账号、支付、分单引擎等核心组件



Plan A 的优缺点

- 优点
  - 函数式编程思想，类似 erlang 的 actor 模型，每个 worker 都是逻辑十分单纯的功能，极高的逻辑内聚性
  - Worker 的上下游可方便的通过配置修改或添加
  - 所有请求可追踪、重放、复制、进行实时大数据分析
  - 天然支持灰度上线、动态容量伸缩、有损服务
- 缺点
  - 方案抽象度过高，超出团队可接受水平
  - 遗留系统代码完全不可复用，整体开发量巨大
  - 服务端框架全部自研，缺乏必要的工具链支持

最终半途而废……

服务器 API 拆分 Plan B：

- 2016 年方案
  - 回归现实，先从遗留代码拆分开始，以面向对象思路进行拆分
  - 保持接口不变、数据存储不变，将一次函数调用变成多次 RPC 调用
  - 小步快跑，多次分城市灰度上线

- 缺点？

效果怎么样？

乘客 App

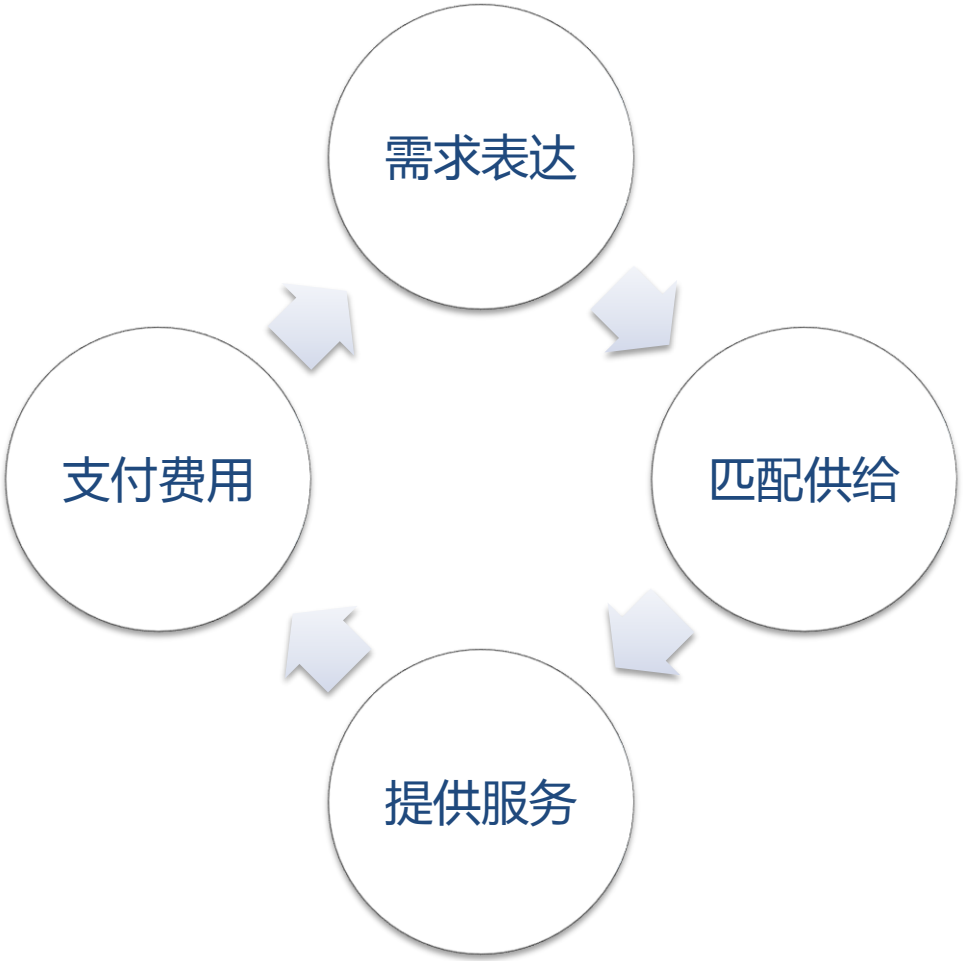
	重构前	重构后
业务开发	业务间代码互相影响，经常因为业务一个修改全局 build break	业务独立开发测试，互不干扰
组件化	基本靠代码拷贝，无法组件化	每个组件都可做到独立仓库管理，独立更新周期
构建速度	改动代码经常会造成全部重新编译，一次编译 30 分钟	任何修改对编译速度的影响都非常有限，能极快的完成编译
迭代速度	基本两周一个版本，半年内曾两次因为业务间互相等待造成重大延期	稳定两周一个版本，很有节奏感
Crash 率	> 1%	< 0.3%

Web App

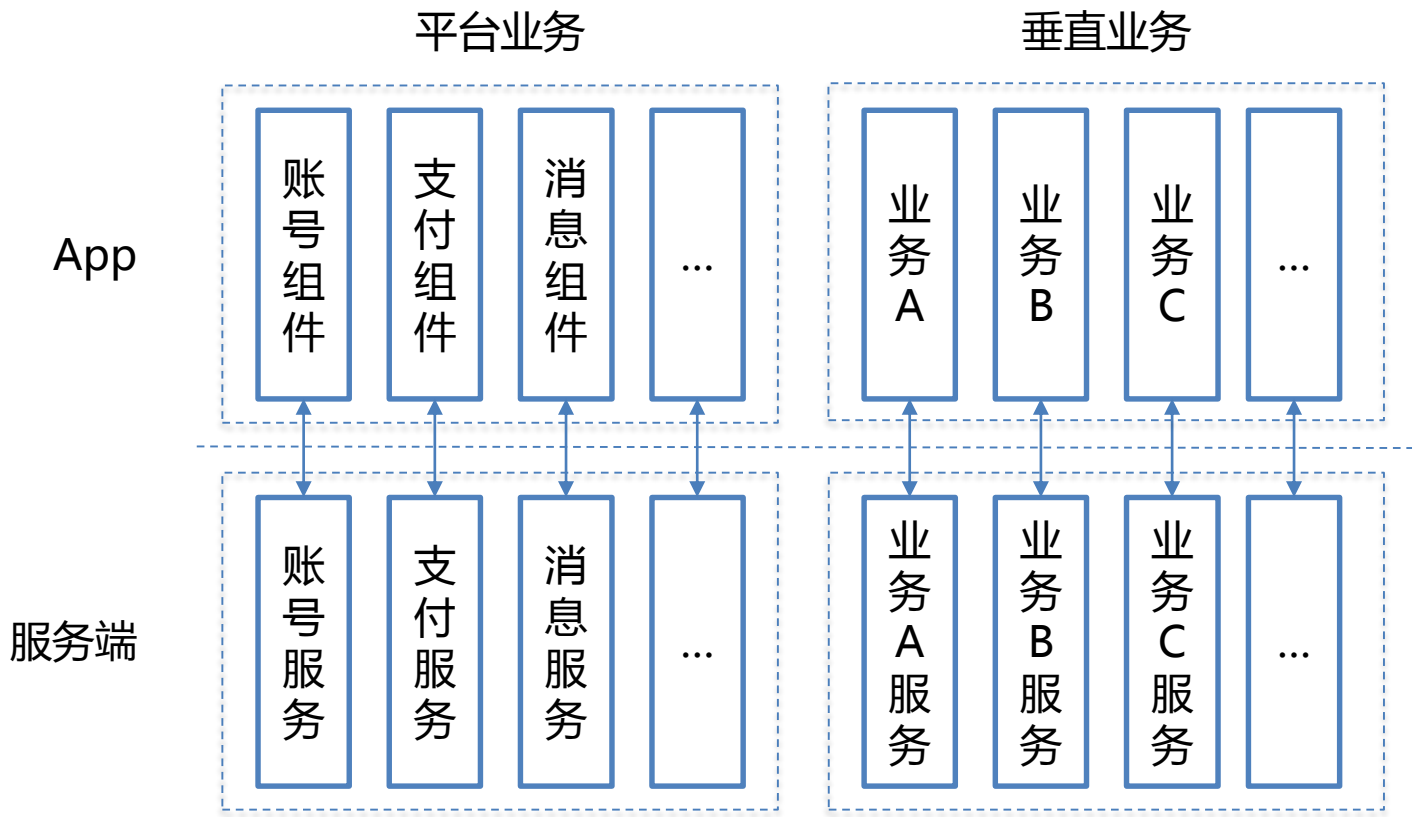
	重构前	重构后
业务开发	业务的首页功能开发依赖于出租车团队排期	业务独立开发测试上线，互不干扰
组件化	基本靠代码拷贝，无法组件化	每个组件都可做到独立仓库管理，独立更新周期
加载速度	弱网环境下，first rendering 时间长	全部资源离线缓存按需加载，first rendering 时间大幅提升
用户体验	跟乘客 App 相比过于简单的地址选择、时间选择、导航栏等控件	极度贴近原生乘客 App 的各种控件体验

如何避免重蹈覆辙？

重新思考业务模型：抽象、抽象、再抽象



形成从客户端到服务端的 Feature Team



提供一个标准化组件平台

Thanks!