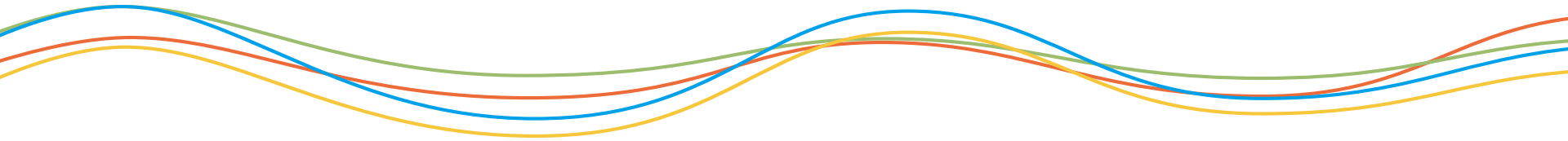
瓜子服务架构的变迁和性能优化

纪鹏程@瓜子二手车



自我介绍

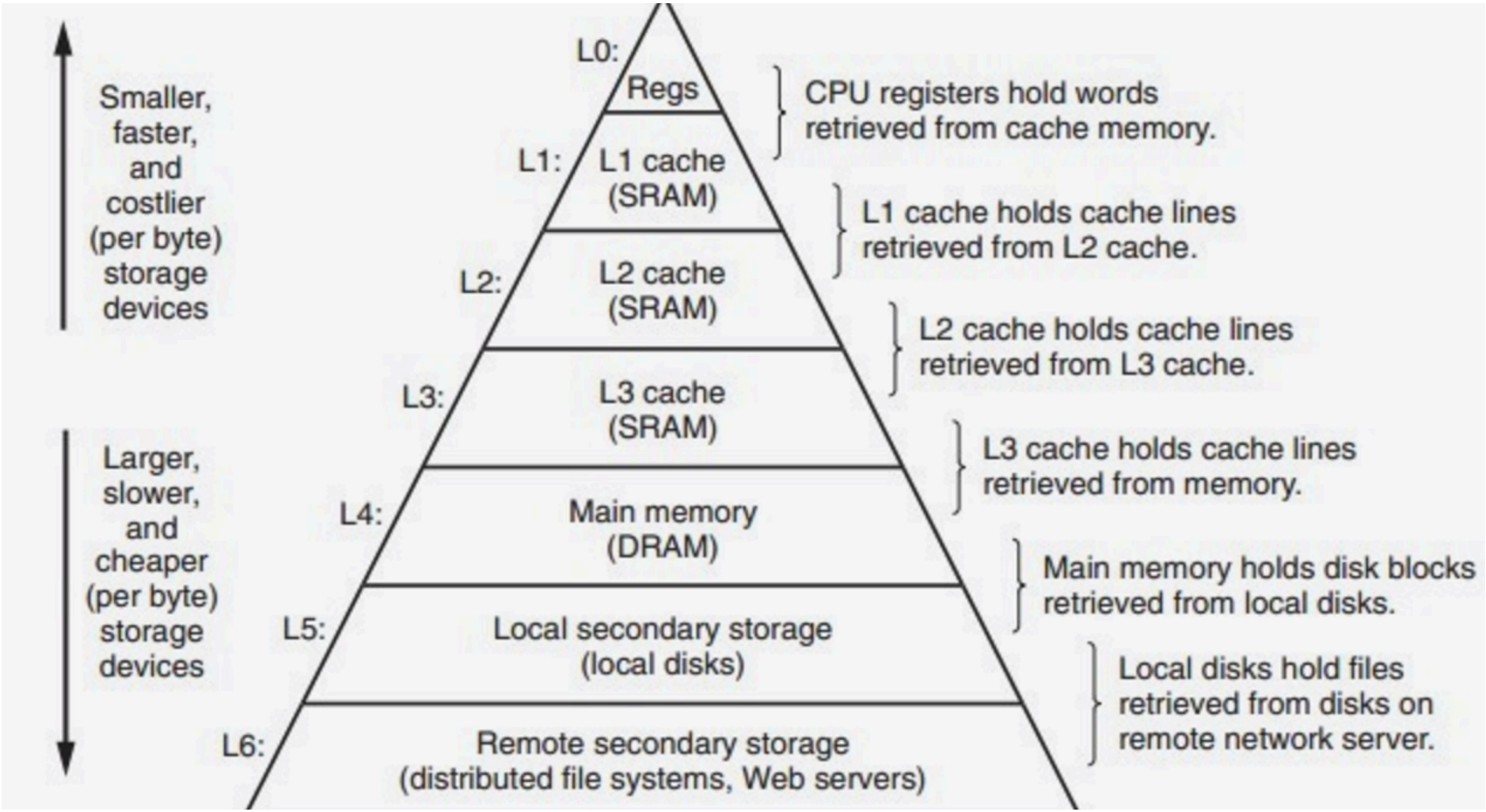
- 2000.9-2004.7 南开大学
- 2004.9-2007.7 中科院软件所
- 2007.7-2011.5 百度-贴吧
- 2011.5-2015.9 赶集
- 2015.9-now 瓜子二手车

演讲大纲

- Web性能的优化思路
- 瓜子Web架构的优化过程
- 微服务化的挑战

Web性能的优化思路

性能的层级


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
CPU registers hold words retrieved from cache memory.
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from L2 cache.
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from L3 cache.
L3:
L3 cache (SRAM)
L3 cache holds cache lines retrieved from memory.
L4:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L5:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network server.
L6:
Remote secondary storage (distributed file systems, Web servers)

速度的量级

•L1 cache reference	0.5ns
•L2 cache reference	7ns
•Read 1M sequentially from memory	0.25ms
•Disk seek	8~10ms
•Send 1M bytes over 1Gbps network	10ms
•Read 1MB sequentially from disk	20~25ms

系统性能的木桶定律

IO介绍

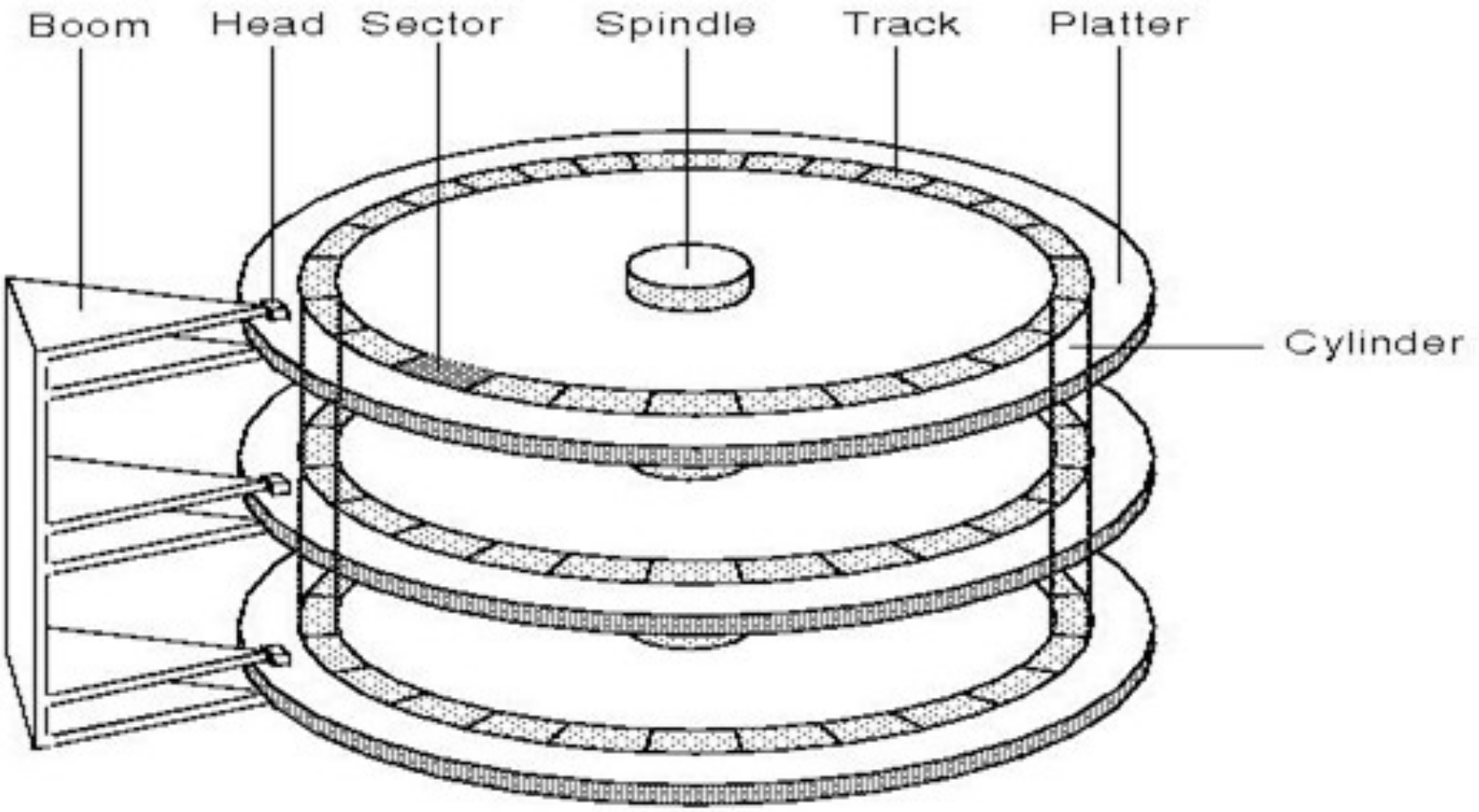
- 媒介：　　磁带、磁盘、SSD
- 读写方式：　　顺序读写、随机读写
- 加速设计：　　缓存、预读、回写
- 系统缓存：　　IO buffer、Direct IO

“内存是新的硬盘，硬盘是新的磁带”

-Jim Gray

磁带

磁盘的物理结构



IO的优化

- 相关联的数据物理相邻，减少IO的读写次数
- 提高顺序读写的比例
- 举例：贴吧帖子、LevelDB

数据库的优化

- 减少表的join操作
- 一定程度上反范式设计
- 索引量的折中，兼顾读和写
- KV存储系统的使用
- 慢查询的控制

网络层的优化

- C10K问题
- select、epoll（ET、FT）
- TCP、UDP、HTTP、WebSocket
- 长连接、短连接
- 同步、异步
- 阻塞、非阻塞

前端优化

- 合并请求
- 域名拆分
- 模块化&按需加载
- 开启Gzip和HTTP/2
- 开启KeepAlive
- Minify

编程语言的选择

- 性能
- 生态的完备性
- 复杂度
- 领域相关
- 多进程 / 线程、协程

性能监控的方法

- 代码打点
  - 优点：简单、高效，对系统性能损失小
  - 缺点：每次都需要修改代码、不够全面
- 增加性能插件
  - 优点：一次部署，终身受用
  - 缺点：系统性能损耗
- 使用APM产品
  - 优点：使用简单，部署快，全面
  - 缺点：需要一定费用，也有一定的损耗

总体原则

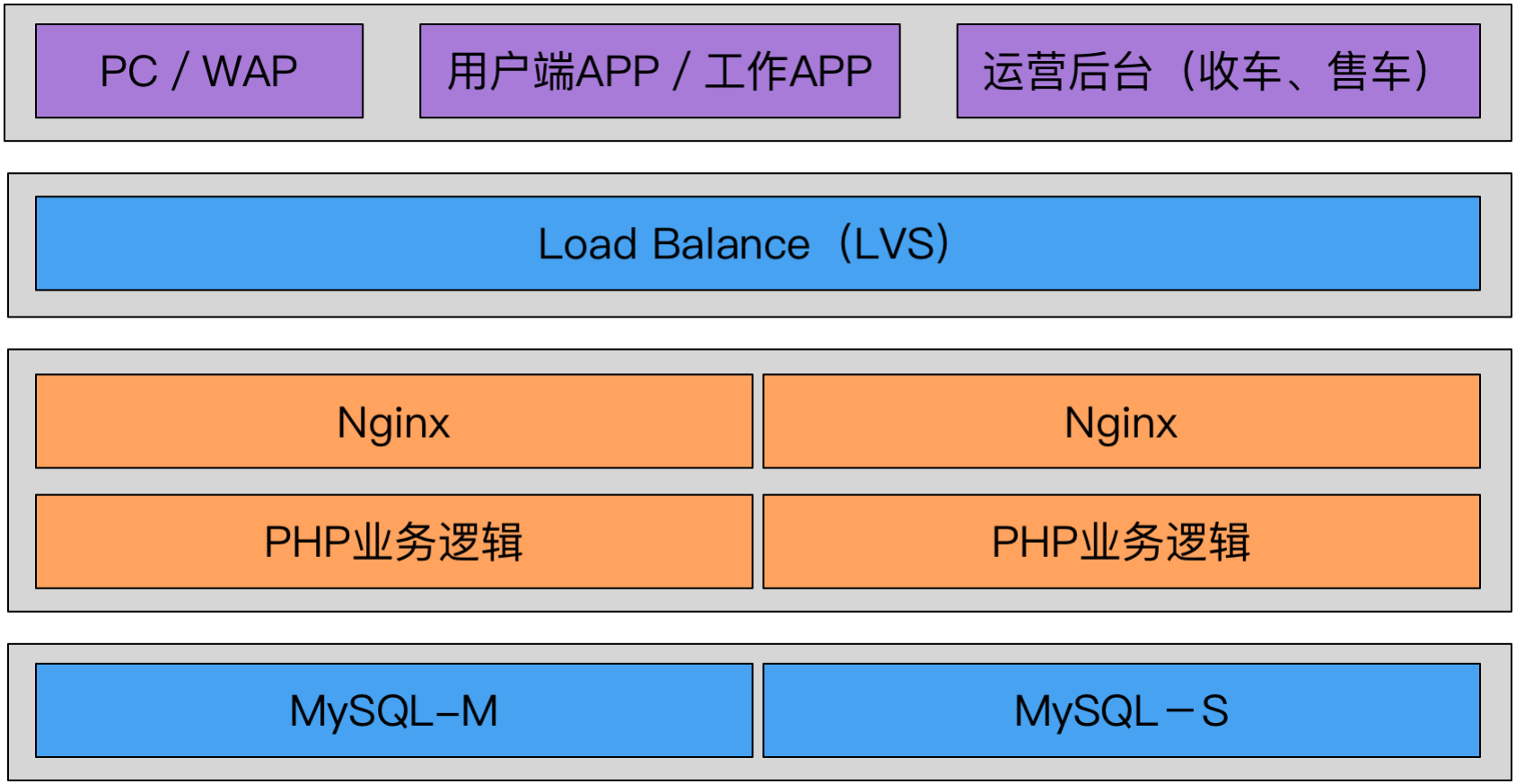
- 寻找合适的性能监控方法
- 找出最短的短板
- 甄别短板类型（IO密集、网络、计算密集）
- 针对性解决

前期抓大放小，后期精细处理

瓜子WEB架构的优化过程

质量、及时交付、性能

$V_{0.1}$的架构


PC / WAP
用户端APP / 工作APP
运营后台（收车、售车）
Load Balance（LVS）
Nginx
Nginx
PHP业务逻辑
PHP业务逻辑
MySQL-M
MySQL－S

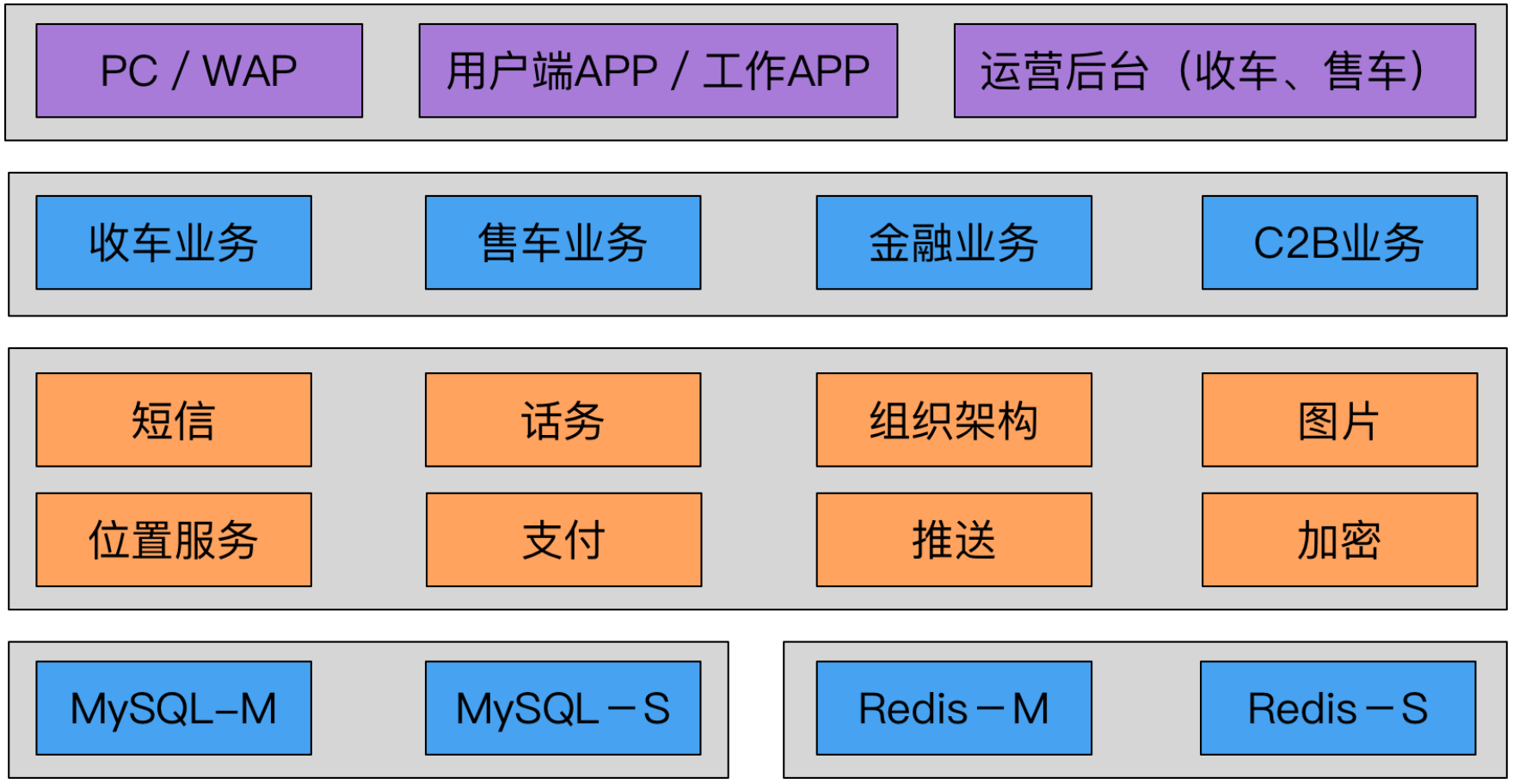
$V_{0.1}$的启示

- “过早优化是一切罪恶的根源”
- 主要精力集中在业务上，快速迭代
- 尽量采用第三方服务
- 不加应用级cache
- 消除架构中的“单点”
- 数据安全性
- 数据字典的确立

$V_{0.1}$的问题

- 代码可维护性差，新人介入成本太高
- 耦合太紧密，微小的错误也会带来整体的crash
- 全部业务都在一个DB实例上，性能扩展性极低
- 日志缺乏和规范不统一，定位问题困难
- 系统、应用、业务级别的监控缺乏

$V_{0.5}$的架构


PC / WAP
用户端APP / 工作APP
运营后台（收车、售车）
收车业务
售车业务
金融业务
C2B业务
短信
话务
组织架构
图片
位置服务
支付
推送
加密
MySQL-M
MySQL－S
Redis－M
Redis－S

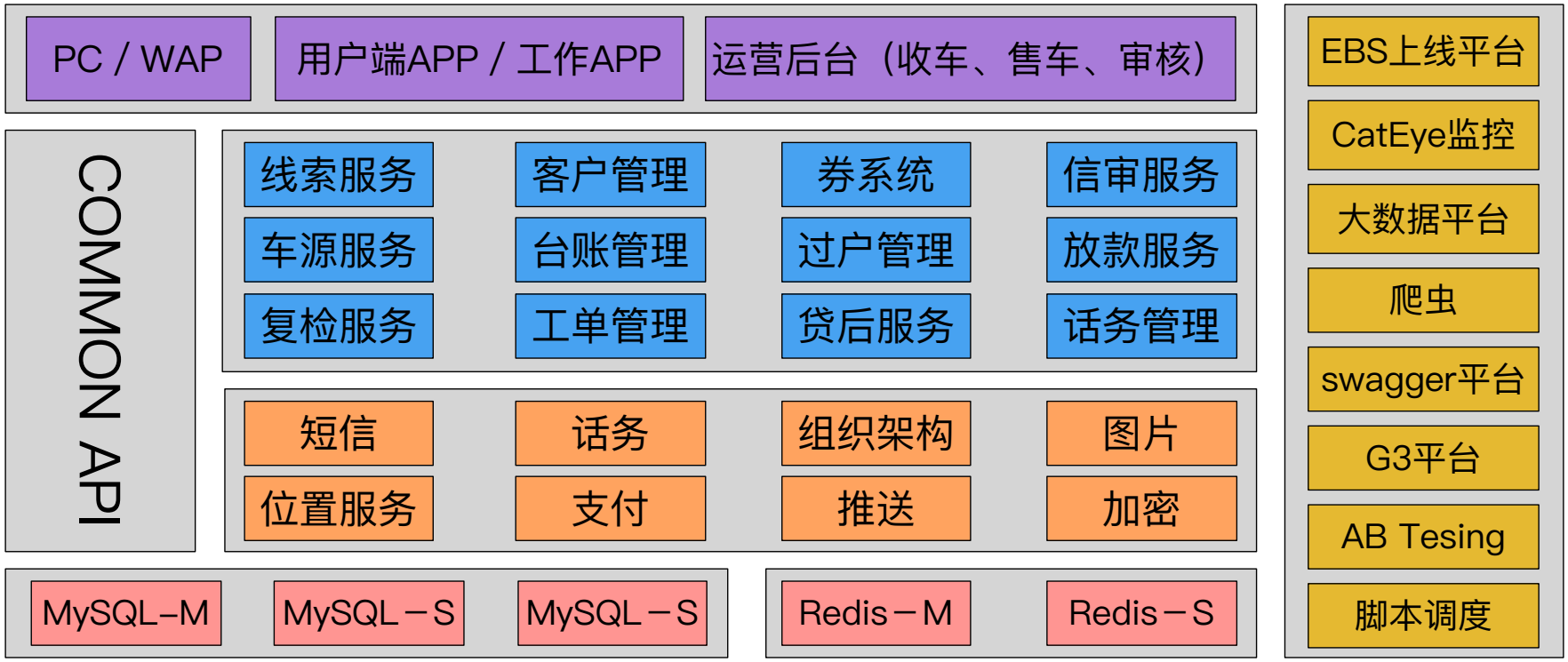
$V_{0.5}$的启示

- 业务级别的代码拆分
- DB上按照业务分实例
- 统一日志规范
- Open-falcon的应用
- 基于Twenproxy的redis集群
- 简单逻辑直接走openresty

$V_{0.5}$的问题

- 业务内部拆分不够细，耦合依然比较重
- 代码分层混乱，越级、跨级调用多
- 业务间接口调用方式千差万别
- 接口的可用性和性能监控不够
- 上线工具不完善，不支持回滚
- 测试以黑盒为主

$V_{1.0}$的架构


PC / WAP
用户端APP / 工作APP
运营后台（收车、售车、审核）
COMMON API
线索服务
客户管理
券系统
信审服务
车源服务
台账管理
过户管理
放款服务
复检服务
工单管理
贷后服务
话务管理
短信
话务
组织架构
图片
位置服务
支付
推送
加密
MySQL-M
MySQL－S
MySQL－S
Redis－M
Redis－S
EBS上线平台
CatEye监控
大数据平台
爬虫
swagger平台
G3平台
AB Tesing
脚本调度

$V_{1.0}$的启示

- 业务内拆分更细粒度的服务
- 统一接口规范
- RPC的调用，串行 -> 并行
- 计算密集型改由go服务来实现
- 代码层分级，禁止跨级调用
- 第三方服务增加主备和自动切换机制

单测

- 单测的重要性
- 持续集成的关键
- 先从核心代码开始，逐步提高覆盖率
- 用好的方法让单测更简单
- 要体现在每日构建中

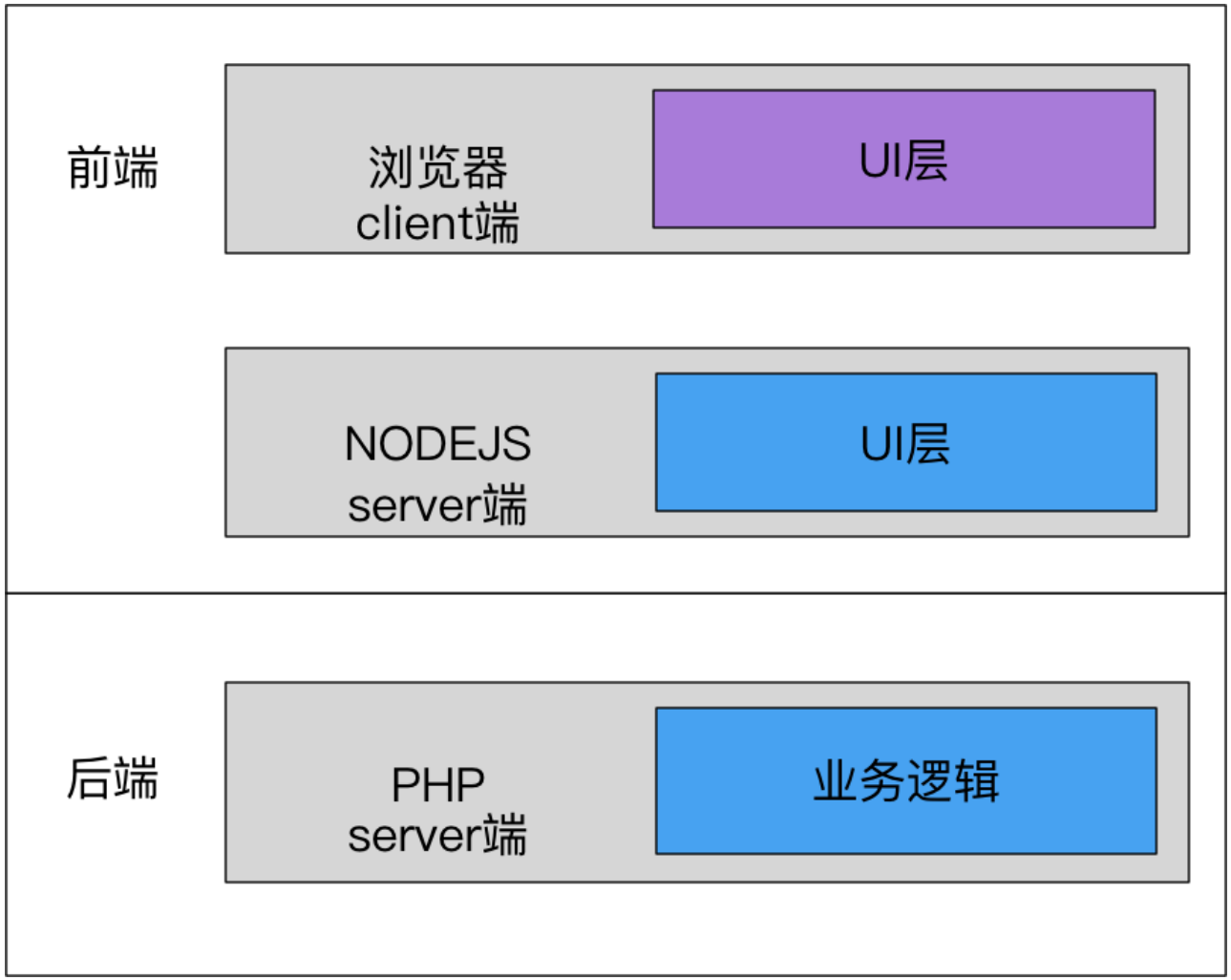
接口

- 接口的重要性与日俱增
- 用swagger集中统一管理
- 接口测试的case管理
- 可用性及性能监控

代码风格

- 设计模式
- 代码规范 - (eg:PSR1、PSR2)
- 使用git hook，提交前code sniffer准入

前后端分离



微服务化的挑战

架构的复杂度

- 服务注册、服务发现、熔断机制等基础组件
- 架构和部署上带来的复杂度
- 需要很强的OpDev团队
- 根据组织的复杂度来决定服务化的粒度

！架构上的分工来弥补性能的损失和复杂度的提高

注意点

- 本地调用和远程调用问题
- 服务的粒度问题
- 数据库的问题

Q & A