去哪儿网客户端无埋点监控与性能优化实践

孟超

去哪儿网公共产品部大前端负责人

Geekbang. 极客邦科技 InfoQ

极客邦科技
会议推荐2019

5月

QCon 北京
全球软件开发大会
大会：5月6–8日
培训：5月9–10日

QCon 广州
全球软件开发大会
培训：5月25–26日
大会：5月27–28日

6月

GTLC GLOBAL TECH LEADERSHIP CONFERENCE 上海
技术领导力峰会
时间：6月14–15日

GMTC 北京
全球大前端技术大会
大会：6月20–21日
培训：6月22–23日

7月

ArchSummit 深圳
全球架构师峰会
大会：7月12–13日
培训：7月14–15日

10月

QCon 上海
全球软件开发大会
大会：10月17–19日
培训：10月20–21日

11月

GMTC 深圳
全球大前端技术大会
大会：11月8–9日
培训：11月10–11日

AiCon 北京
全球人工智能与机器学习大会
大会：11月21–22日
培训：11月23–24日

12月

ArchSummit 北京
全球架构师峰会
大会：12月6–7日
培训：12月8–9日

Geekbang 极客邦科技 | TGO 鲲鹏會

TGO鲲鹏会

汇聚全球科技领导者的高端社群

全球12大城市

850+高端科技领导者

使命 Mission

为社会输送更多优秀的科技领导者

愿景 Vision

构建全球领先的有技术背景优秀人才的学习成长平台

扫描二维码，了解更多内容

自我介绍

2014 年加入去哪儿网

主要从事客户端混合开发框架相关的工作。先后负责参与去哪儿网 Hybrid 框架、QRN 框架（基于 React Native）、热更新系统、客户端监控、性能优化等相关工作。

目前专注于客户端性能优化和 APM 监控的一体化解决方案

目录

去哪儿网客户端性能监控面临的挑战

Qunar App 的现状

宫格：

RN 有 8 个

Hybrid 有 5 个

Native 有 2 个

小宫格：

9 个入口全是 Hybrid

Tab 全是RN

工作中遇到的困难

简单工作流：

问题发现难 → 缺乏报警机制

问题定位慢 → 统计信息不全

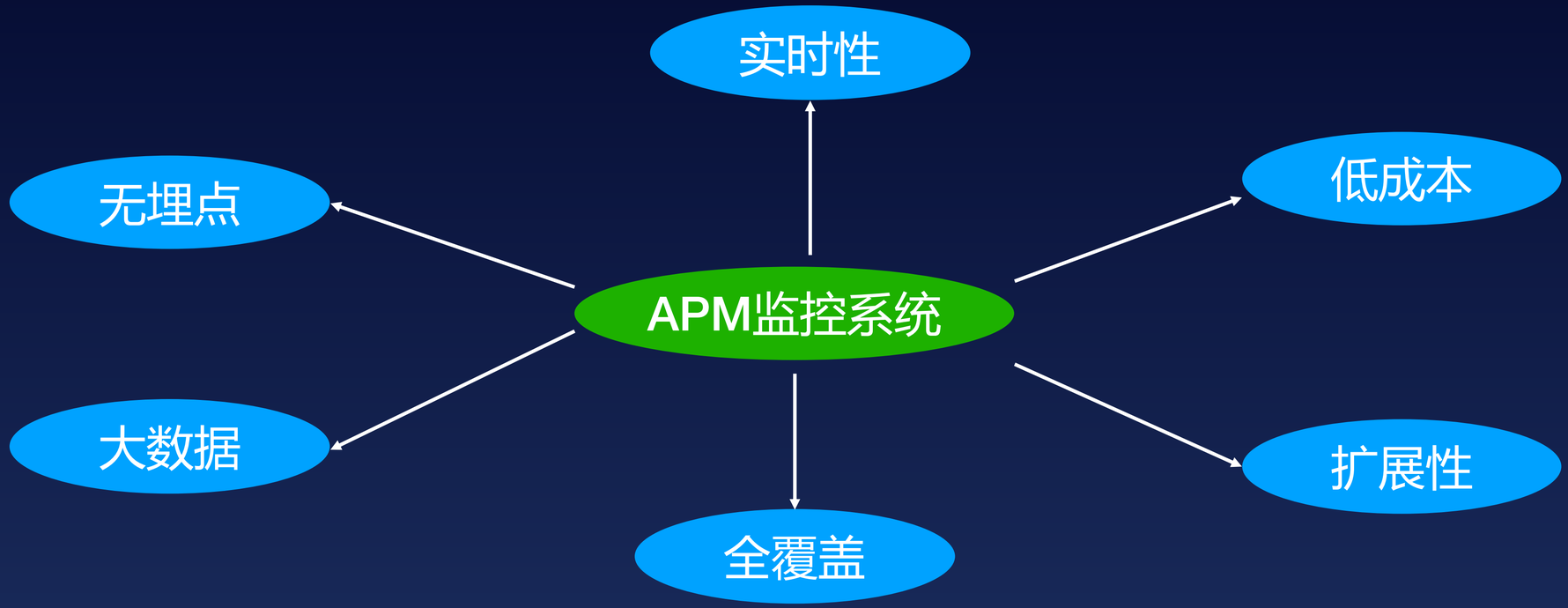
性能方面的担忧

性能：Native > RN > Hybrid

大量采用 RN 和 Hybrid 需要考虑的问题：

- 复杂度高的业务，性能会不会成为瓶颈?
- 发版频率高，如何快速发现性能问题?
- 如何评估优化效果?

需要一个APM 监控系统



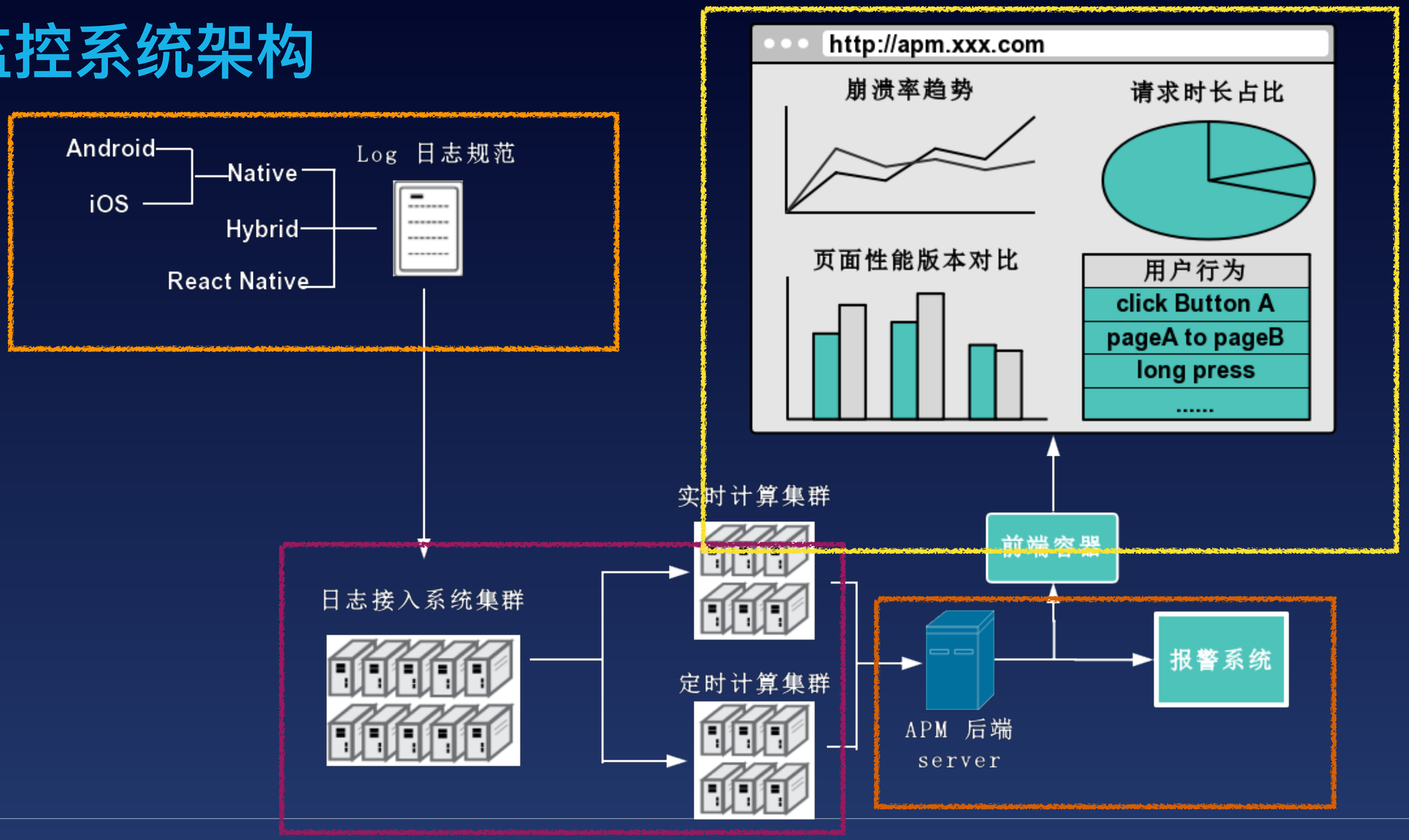
要追求完美吗？

NO，够好就行

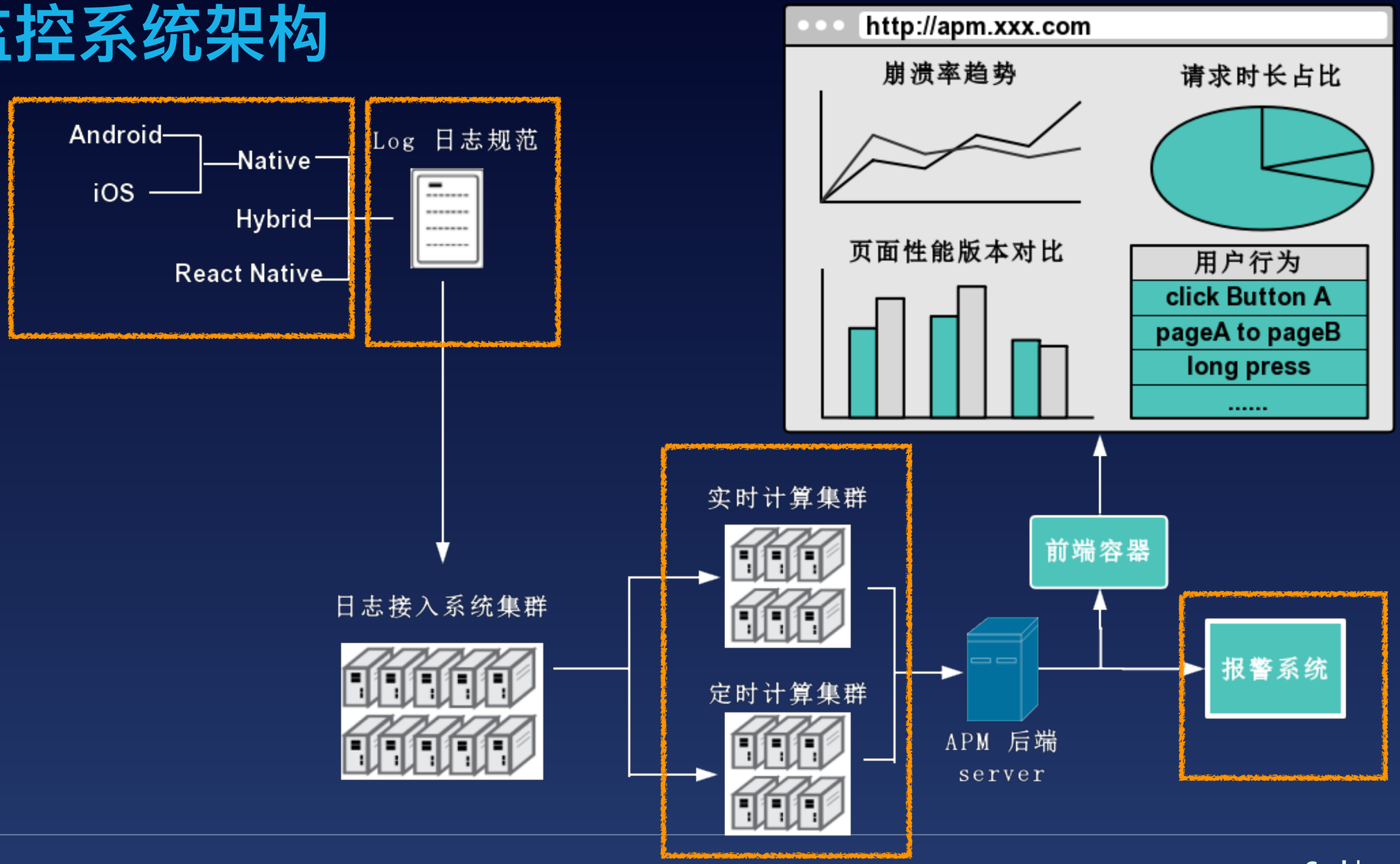
- 覆盖关键性能指标、多维度分析
- 异常快速预警

去哪儿网客户端性能监控体系建设

监控系统架构

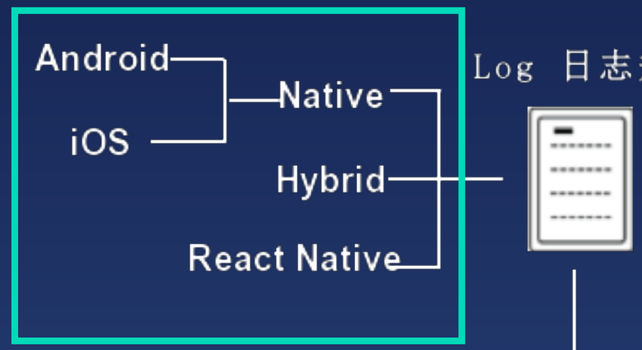

Android
iOS
Native
Hybrid
React Native
Log 日志规范
http://apm.xxx.com
崩溃率趋势
请求时长占比
页面性能版本对比
用户行为
click Button A
pageA to pageB
long press
.....
实时计算集群
日志接入系统集群
定时计算集群
前端容器
APM 后端
server
报警系统

监控系统架构


Android
iOS
Native
Hybrid
React Native
Log 日志规范
http://apm.xxx.com
崩溃率趋势
请求时长占比
页面性能版本对比
用户行为
click Button A
pageA to pageB
long press
.....
日志接入系统集群
实时计算集群
定时计算集群
APM 后端
server
前端容器
报警系统

要统计的指标

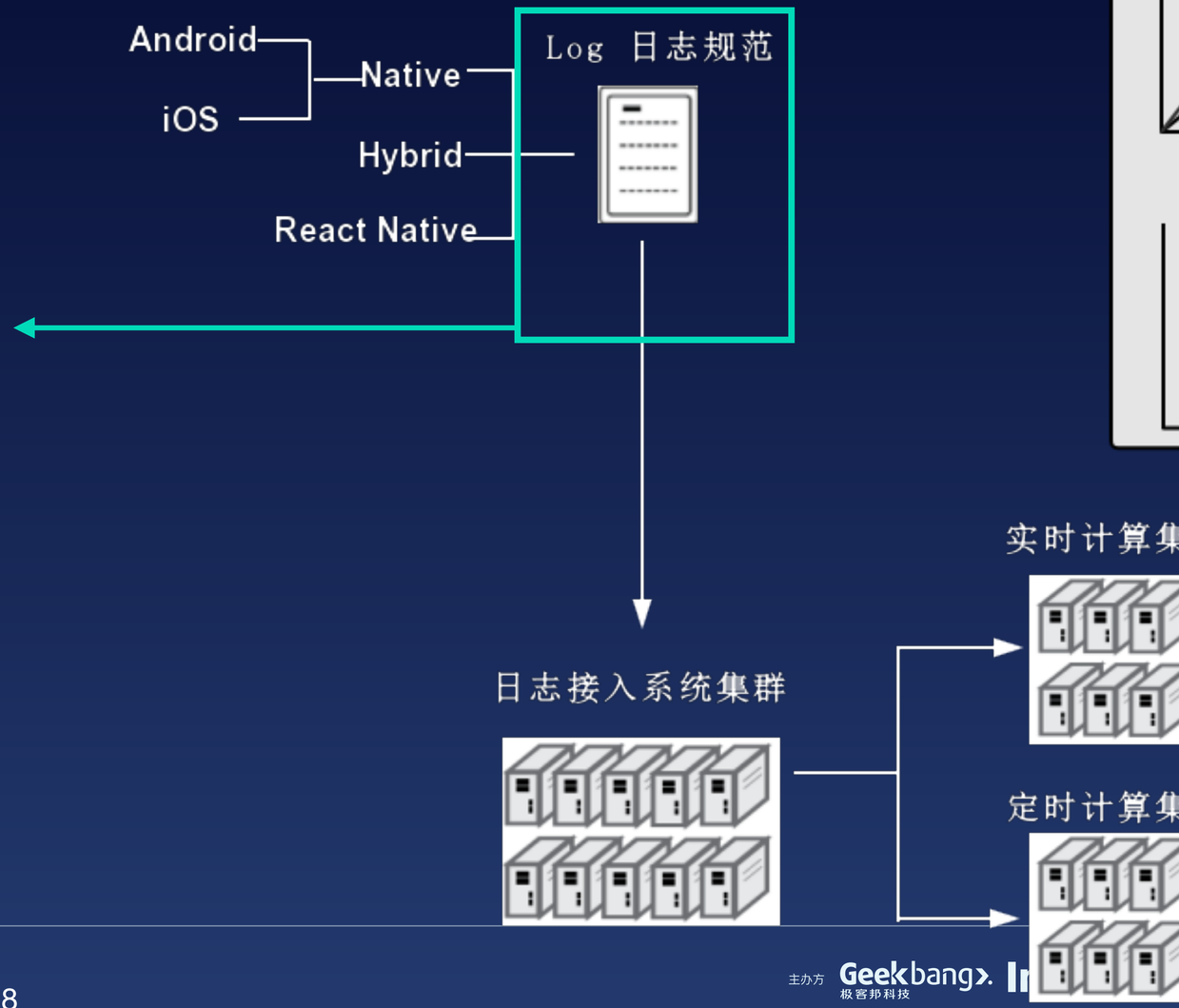
网络请求	成功率 状态码 流量 网络响应时间 DNS解析时间 SSL握手时间 首包时间
卡顿	界面帧率 卡顿堆栈
崩溃	启动崩溃 崩溃率 崩溃堆栈 脚本错误
abort率	内存过高等原因被系统杀死
页面性能	页面创建时间 页面加载完成时间
其它	内存 CPU使用率 APP 启动时间 电量



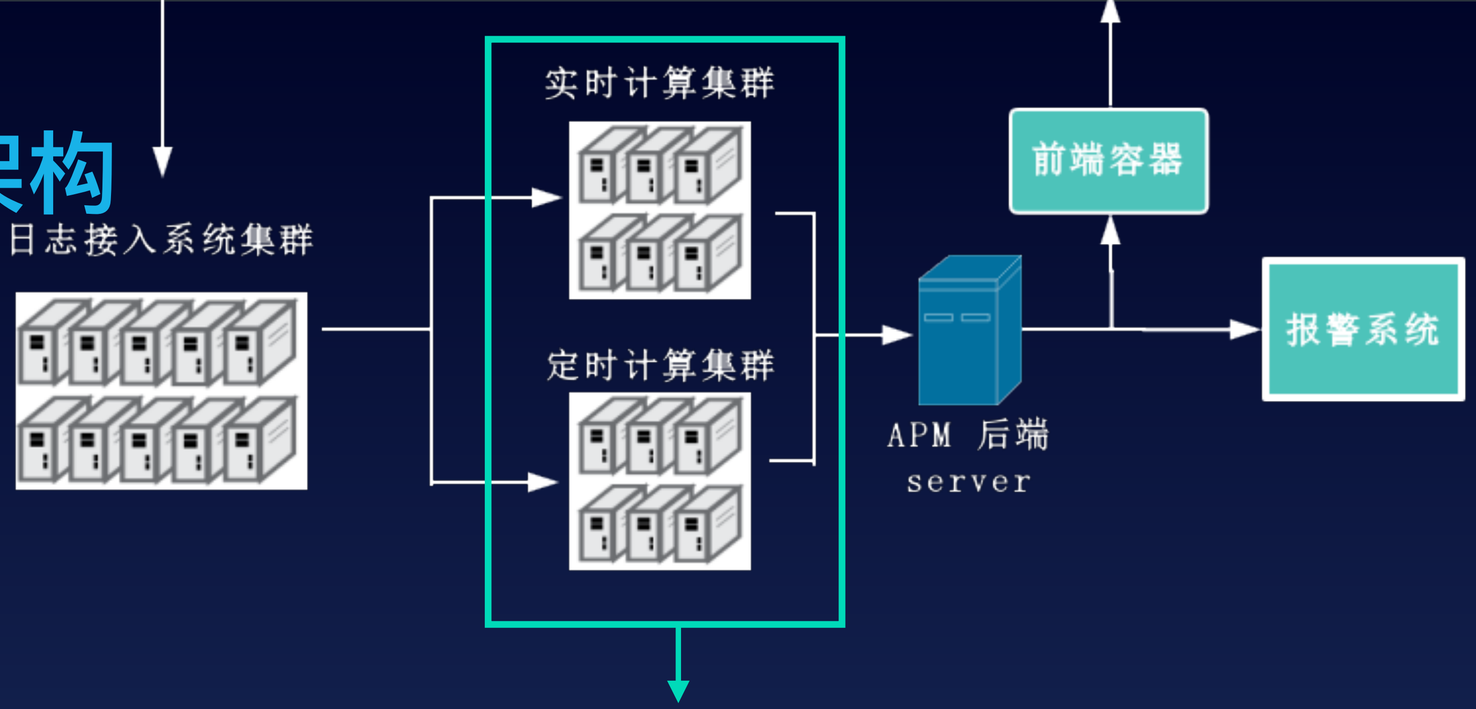
监控系统架构

日志规范 ： 时间戳*Common 参数*Business 参数

```
{
    "platform": "ios",
    "logType": "apm",
    "business":[{
        "action": hyNet/rnNet/iosNet/adrNet...
        "extra": 字符串类型，非必传。主要用于附加调试信息
    }]
    "common":{
          "appVersion": app 版本
          "appID": app id
          "channelID": 渠道 id
          "userID": 用户 id
          "osVersion": 系统版本
          "model": 机型
          "loc": 位置信息
          "mno": 运营商信息
          "timestamp": 时间戳
          "extra": {}扩展字段
    }
}
```



监控系统架构



定时计算（一小时，一天）

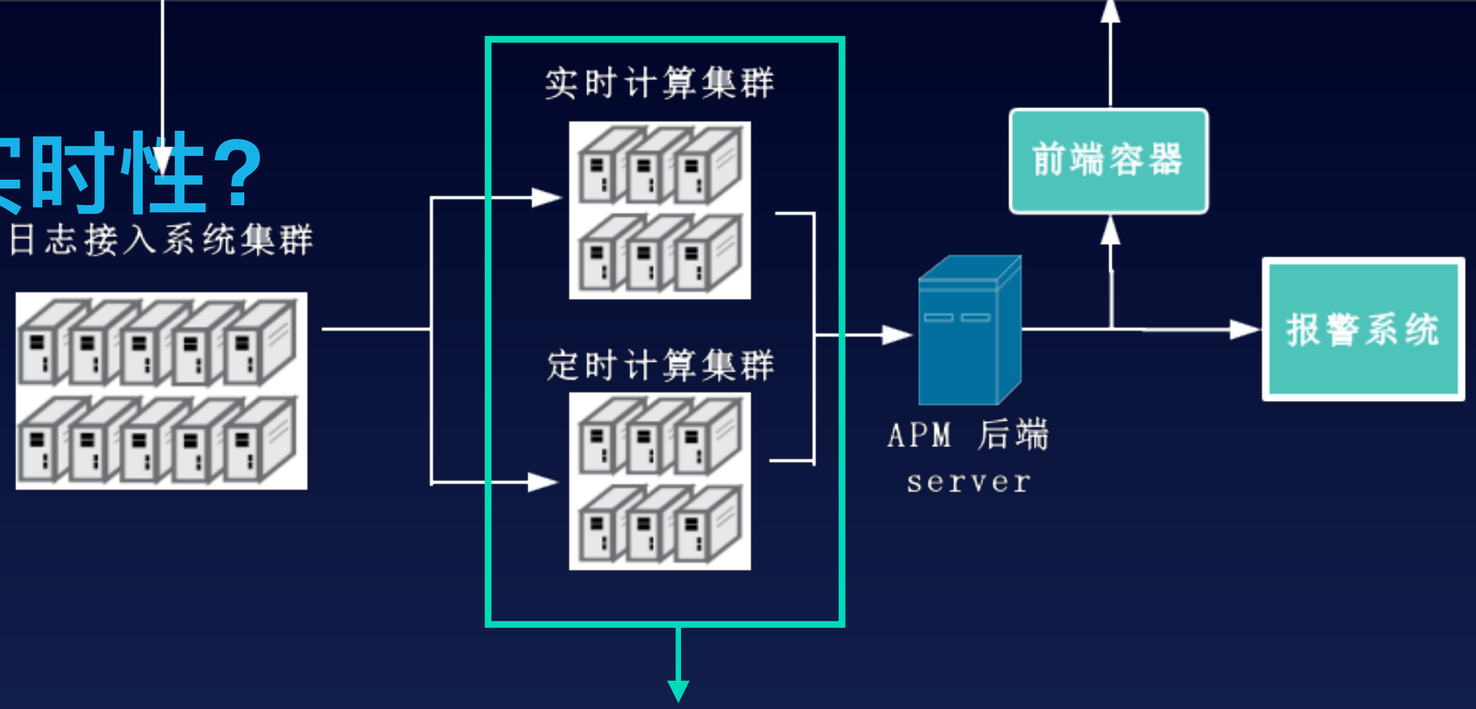
log server → kafka → 入Hive库 → 定时任务 → mysql/es

实时计算

log server → kafka → ES 集群

怎么保证实时性?

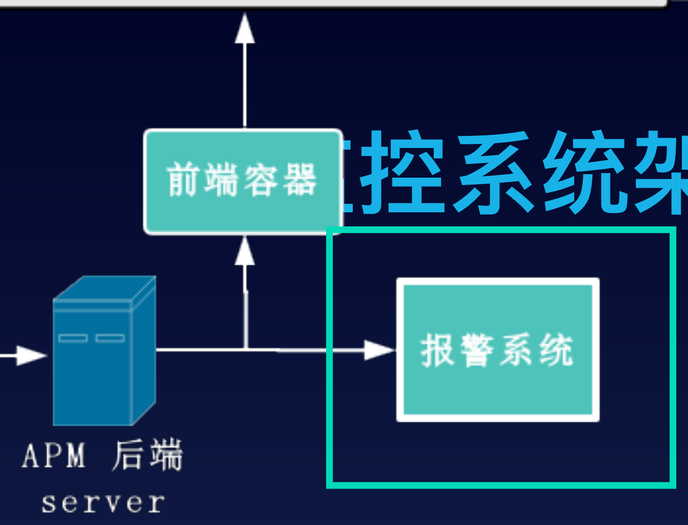
怎么保证实时性?



日志上传的实时性:

- 普通日志累计十条上传
- 崩溃日志第一时间上传

控系统架构

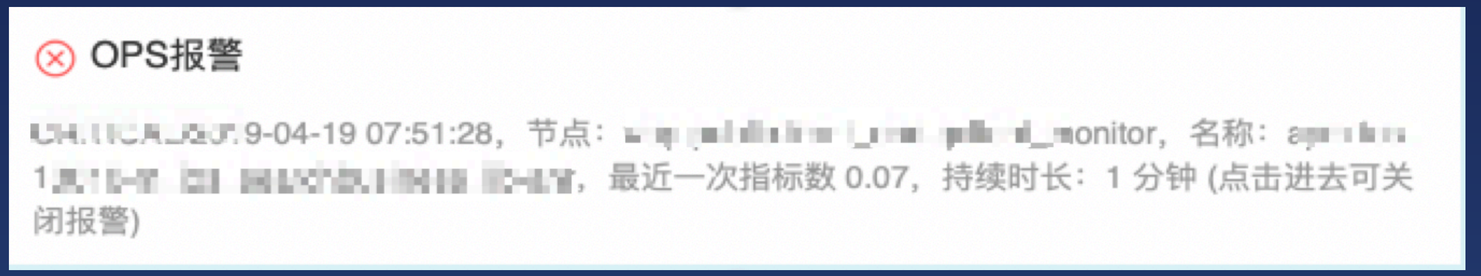


设置报警规则

3%10m>10

次数　10分钟　每秒影响用户数

触发报警



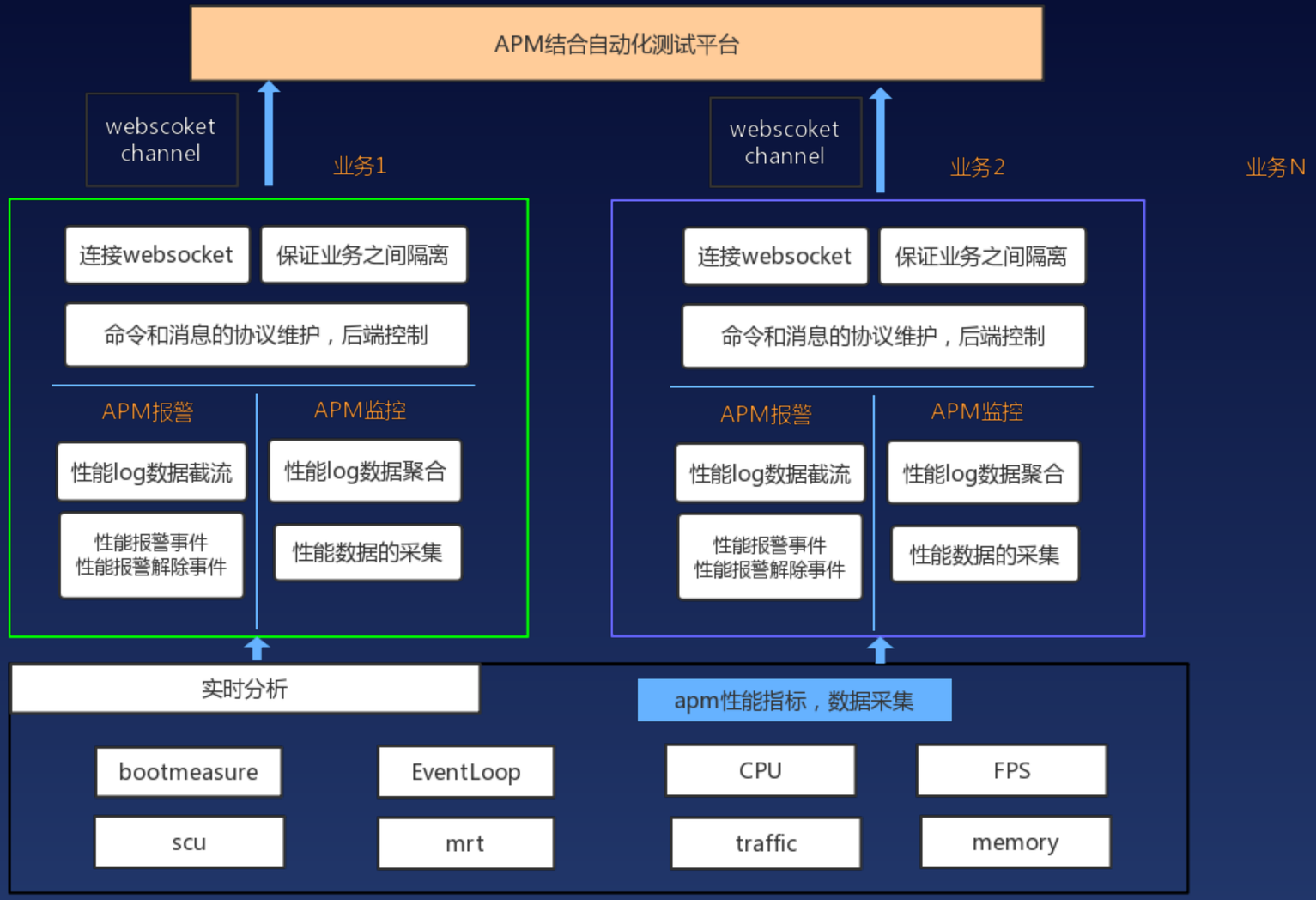
到 apm 平台查看细节，解决问题

监控指标区分环境

- 影响性能，且浪费用户流量
- 不能快速定位到问题具体出在哪里
- 如果能在 beat 环境提前发现问题，能减少损失

在区分环境方面做了哪些事情

1. 结合 QA 自动化测试系统，获得更多Beta数据



在区分环境方面做了哪些事情

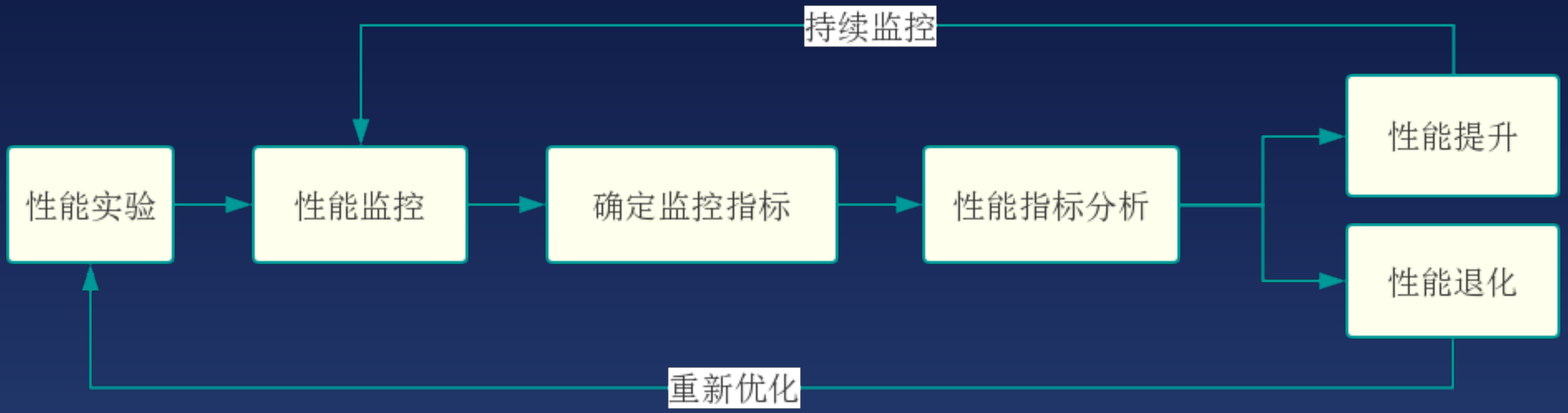
1. 结合 QA 自动化测试系统，获得更多Beta 数据
2. 日志区分 beta 和线上
3. 收紧指标上报条件（比如卡顿上报由线上 5s 改为 3s）
4. 电量，内存，帧率只在 beta环境收集

日志互通，形成闭环

日志互通的关键：uid、时间戳、PageName

例子：崩溃报警->崩溃堆栈->uid->用户行为

形成闭环，持续提供优化建议



重新认识无埋点

- 减少沟通、提高效率
- 优先用无埋点
- 侵入公共代码实现也属于无埋点
- 数据收集无差别（BadCase 较多），需要更强的数据处理能力
- 不能盲信，要和主动埋点结合起来使用

性能优化实践


网络请求
成功率
状态码
流量
网络响应时间
DNS解析时间
SSL握手时间
首包时间
卡顿
界面帧率
卡顿堆栈
崩溃
启动崩溃
崩溃率
崩溃堆栈
脚本错误
abort率
内存过高等原因被系统杀死
页面性能
页面创建时间
页面加载完成时间
其它
内存
CPU使用率
APP 启动时间
电量

页面首屏性能优化实践

确定要关注的指标


页面加载完成时间
Native
RN
Hybrid
用户可见时间
Native 页面生命周期完成
页面创建时间
Native
首屏网络请求时间
Native
Native 渲染时间
Native
页面创建时间
RN
RN 框架 js 加载时间
RN
RN 业务js 加载时间
RN
首屏网络请求时间
RN
RN 渲染时间
RN
页面创建时间
Hybrid
Hybrid 框架初始化时间
Hybrid
h5 加载完成（dom ready）
Hybrid
包含加载 js 前的一些代码执行时间

页面加载完成时间埋点

经历了三个阶段：

1、通过 hook loadview 的显示/隐藏

2、主动埋点

3、检测页面特征，统一埋点

通过 hook loadview 的显示/隐藏

原理：loadView 关闭，则页面渲染完成

↓

通过 AOP 编程思想，替换 loadview 的 show 和hide 方法，

可以间接统计页面渲染时长

通过 hook loadview 的显示/隐藏

静态内容已经渲染完

局部还未完成，正在 loading

陷入坑里无法自拔

- 工作量大：需要适配多种 loading（ loadview 改了就用不了）
- 统计不准：有些页面多次调用 loadview 的 show 和 hide
- 覆盖不全：有些页面没有 loadview

通过 hook loadview 的显示/隐藏

数据靠谱吗?

哪些靠谱, 哪些不靠谱，怎么区分?

值得欣慰的是
hybrid 框架是公共提供的，可以保证 Hybrid 项目靠谱

主动埋点

key	必填	type	语义	建议埋点位置
clickTime	是	客户端时间戳	点击tab	点击行为发生时获取（入口埋点或schema透传）
bizStart	是	客户端时间戳	业务页面初始化	adr:oncreate首行 iOS:init首行 rn:业务js首行
didLoad	是	客户端时间戳	页面加载耗时	adr:onResume首行 iOS:willAppear首行 rn:didMount尾行
onshow	是	客户端时间戳	首次渲染完成	adr:ToWindow iOS:didAppear尾行 rn:onshow首行(接收onshow广播，执行onshow回调)
reqStart	是	客户端时间戳	请求开始	第一个请求前一行（影响首屏/页面渲染的请求，由业务定义）
reqEnd	是	客户端时间戳	请求结束	所有相应的回调判断所有请求是否相应，全部为true则打点（影响首屏/页面渲染的请求，由业务定义）
willUpdate	是	客户端时间戳	开始带数据渲染	第一个请求返回时打点，开始执行update（基于页面的实现业务自定义）
didUpdate	是	客户端时间戳	首屏数据渲染完成	最后一个请求返回时完成执行update时打点（首屏/页面所有模块的渲染完成时打点，由业务定义）

指标	指标名称（英文）	计算方式	含义
用户感知时间	userPerceptionCost	didUpdate - clickTime	从 用户点击 到 首屏有效数据渲染完成（页面可见且可操作）
页面间跳转时间	pageJumpCost	bizStart - clickTime	从 用户点击 到 业务页面初始化（点击对应的逻辑执行、schema分发、框架加载）
首次渲染时间	firstRenderCost	onShow - clickTime	从 用户点击 到 首次渲染完成（使用兜底数据或缓存数据渲染得到的页面，可见）
有效数据渲染时间	effectiveRenderCost	didUpdate - willUpdate	从 有效数据返回 到 首屏有效数据渲染完成 （首屏多个模块渲染的耗时）
索取数据时间	reqDataCost	reqEnd - reqStart	从 客户端请求开始 到 首屏所有数据全部返回

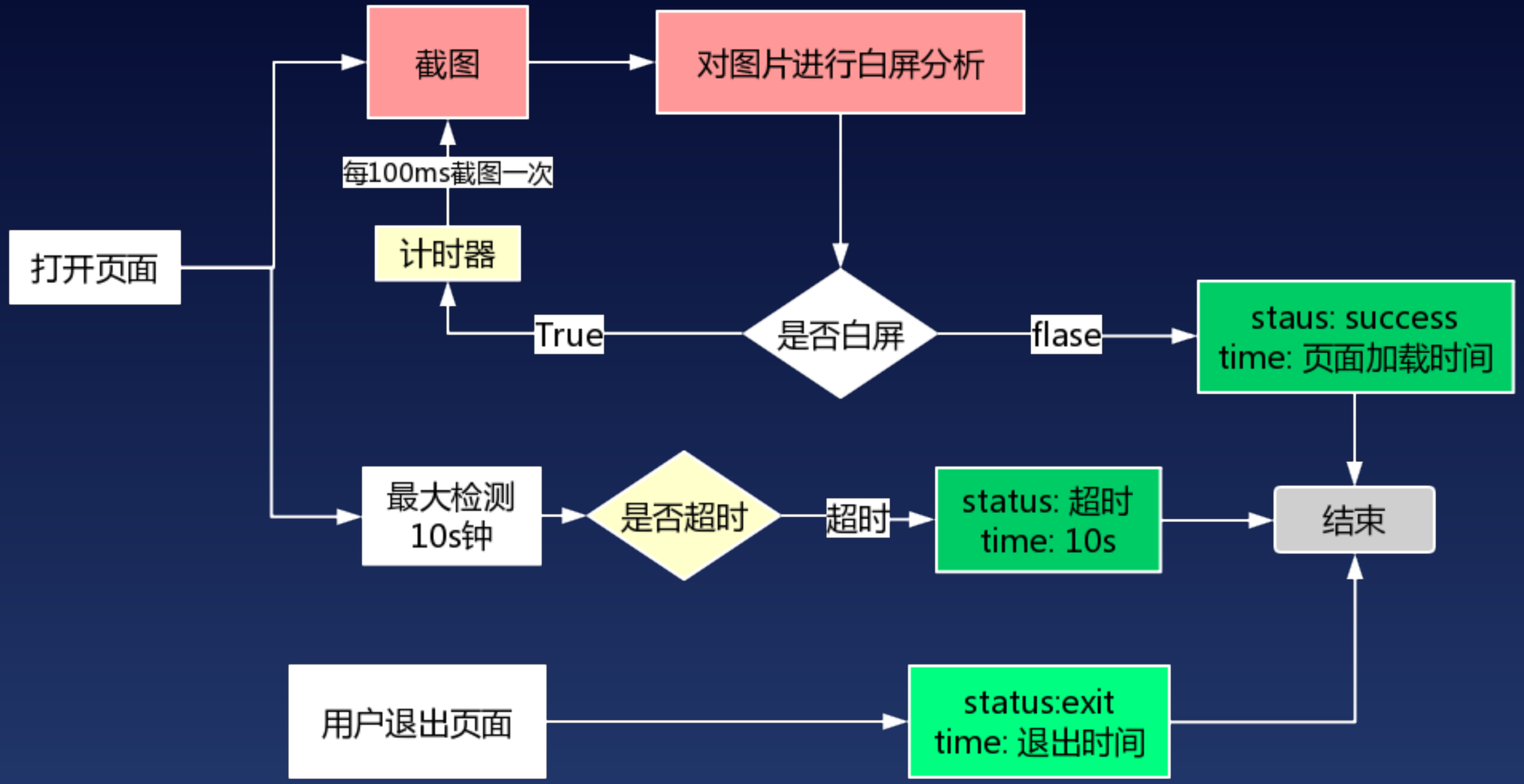
主动埋点

联系各业务负责人，推了一个月，最终完成了任务

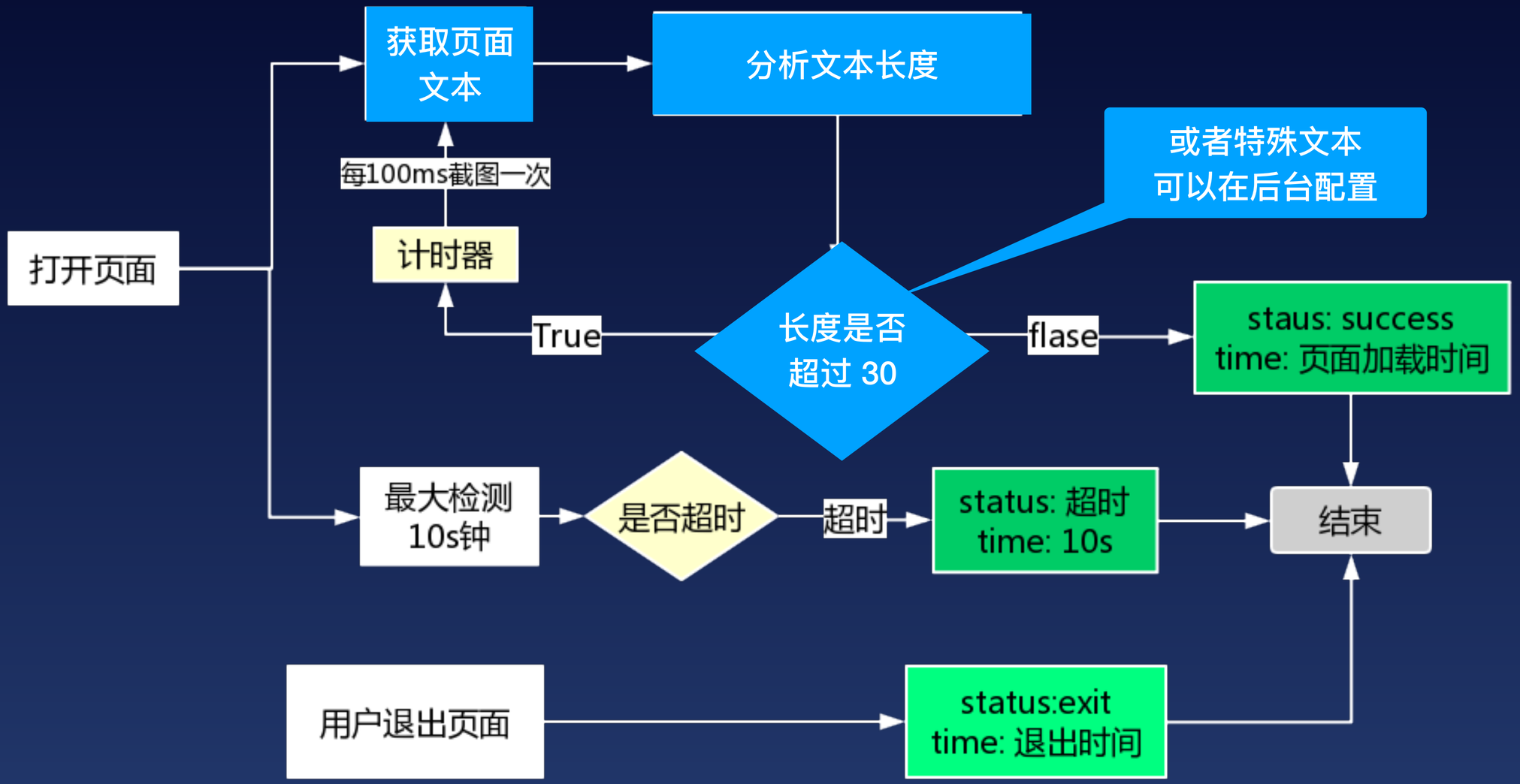
↓ 但是

发现需要再加一个资源版本号参数，有利于按资源版本分析

通过截图像素点分析的方式统计页面时长

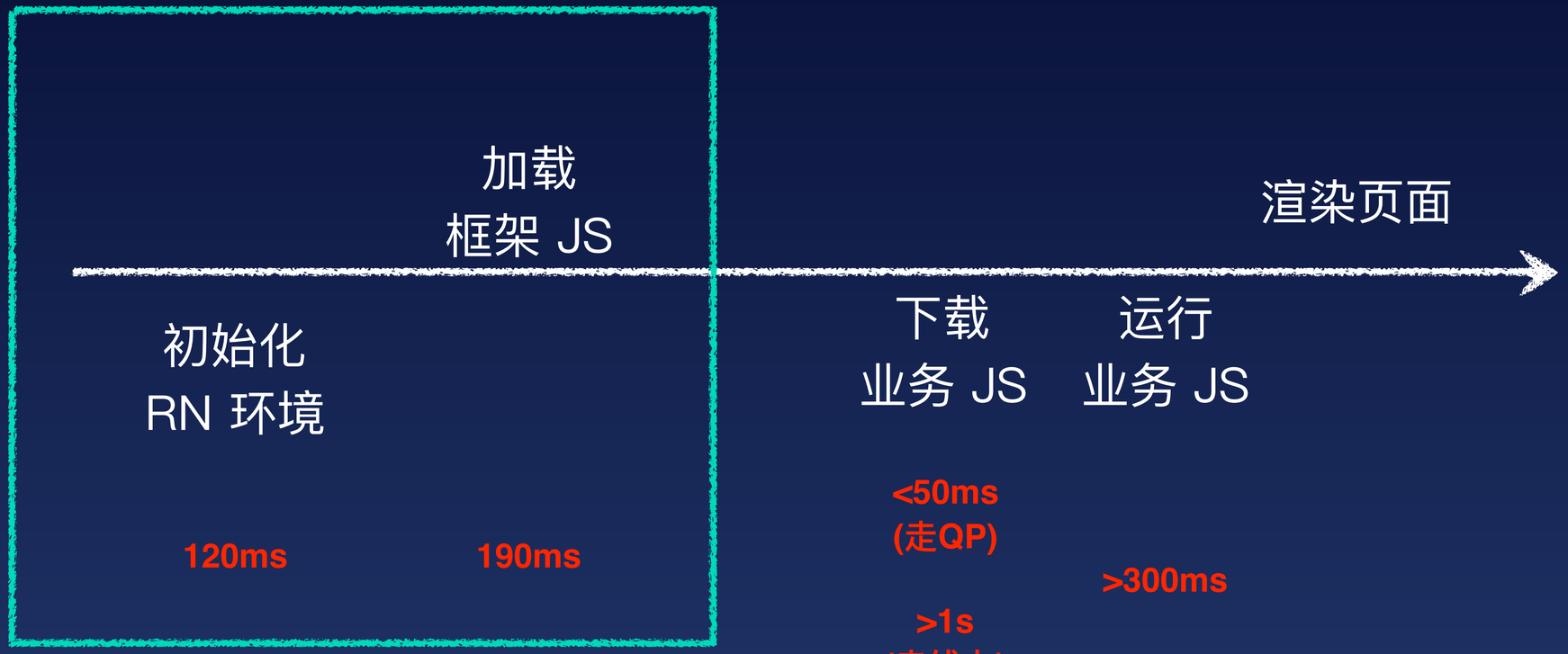


通过文本长度的方式统计页面时长

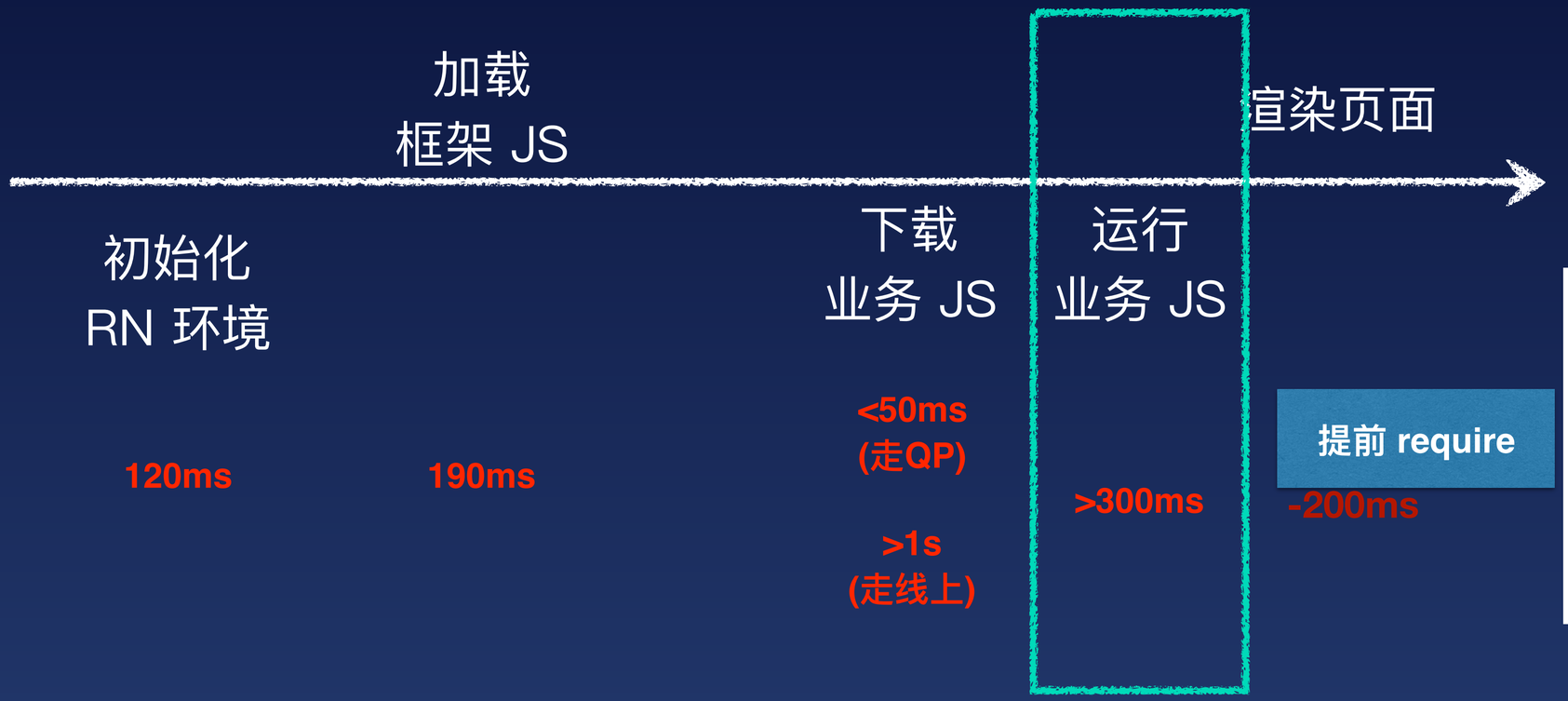


RN 启动优化

预加载：将减少310ms



RN 启动优化




```
import {
  View,
  Text,
  Image,
  TextInput,
  ScrollView,
  ListView,
  StyleSheet
} from 'react-native';
```

RN 启动优化

异步初始化

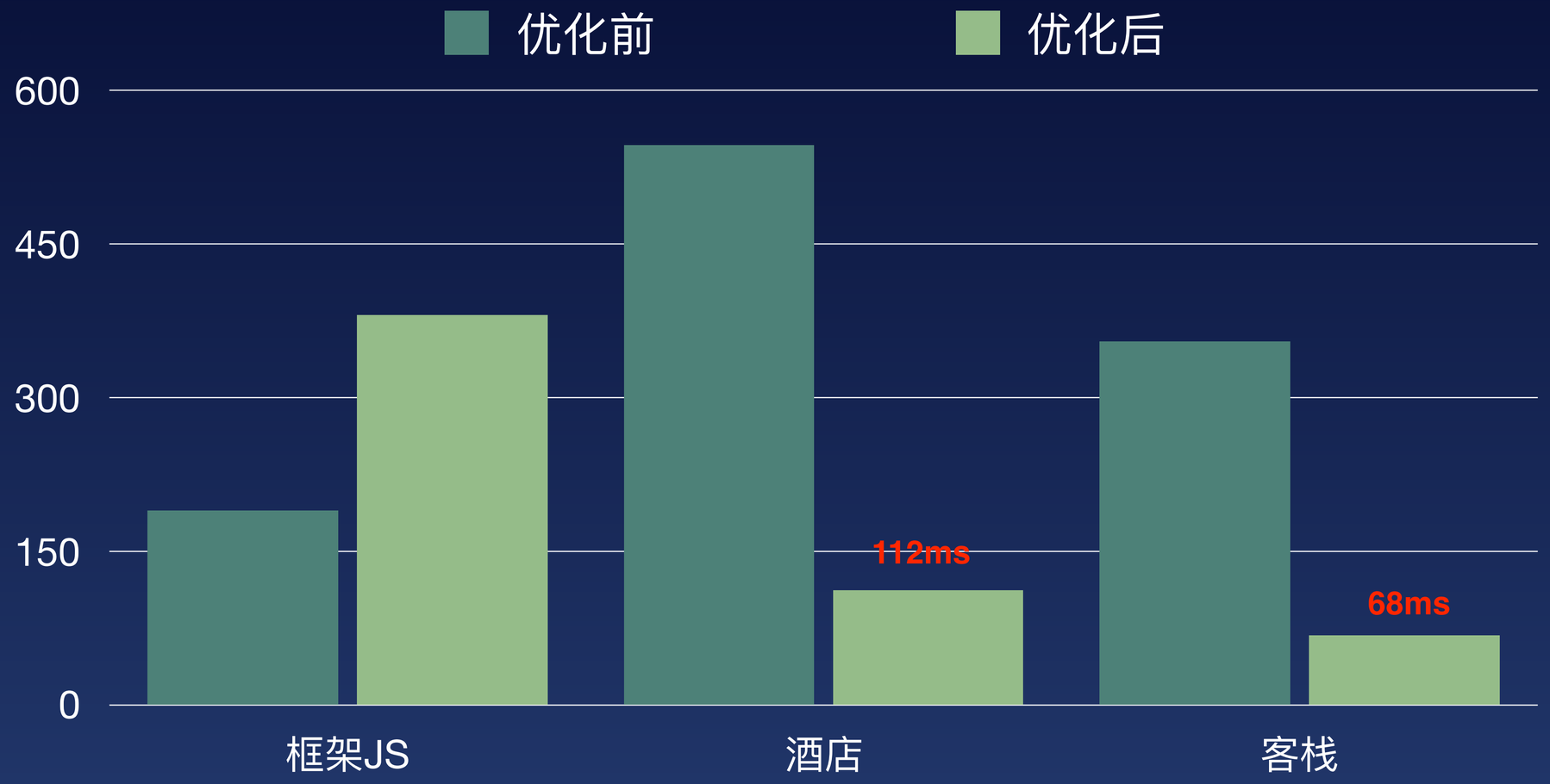
```
AppRegistry.registerComponent('DetailImage', () => {
    const DetailImage  = require('./src/containers/DetailImage/
    DetailImage');

    class DetailImageRoot extends Component {
        render() {
            return (
                <Provider store={store}>
                    <DetailImage {...this.props}/>
                </Provider>
            )
        }
    }

    return DetailImageRoot;
});
```

运行业务 JS 时
不初始化页面

	优化前	优化后	业务JS大小
20个页面	546ms	112ms	940KB
1个页面	82ms		249KB

```
AppRegistry.runApplication('DetailImage', {
  initialProps:{}
});
```

优化效果对比



页面首屏性能优化

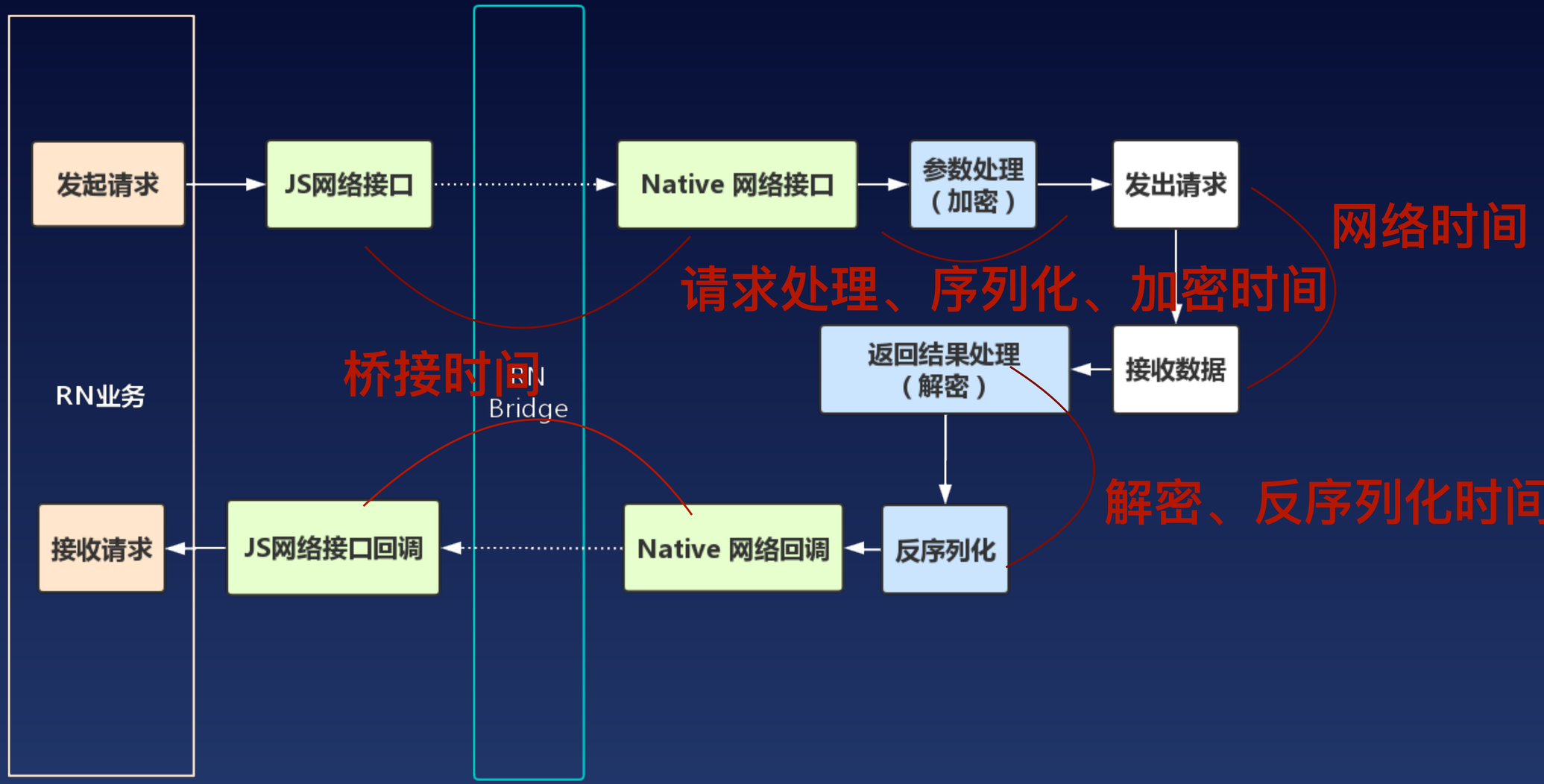
通过监控指标，定合理的目标。

在实现目标的过程中会很多优化手段，就不一一展开了

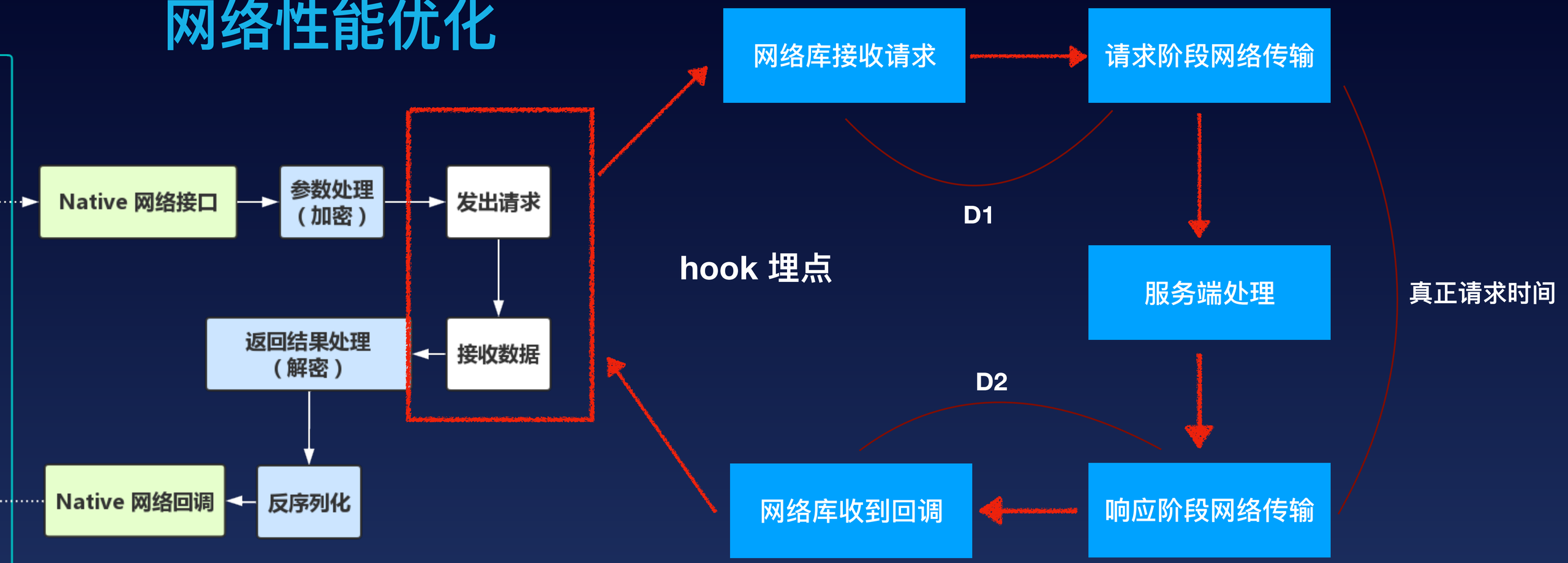
- 模块初始化串行改并行
- 避免重复渲染
- 减少首屏对网络的依赖
- 使用缓存
- 预渲染

网络性能优化实践

网络性能优化


发起请求
JS网络接口
Native 网络接口
参数处理
（加密）
发出请求
网络时间
请求处理、序列化、加密时间
返回结果处理
（解密）
接收数据
桥接时间
RN业务
RN
Bridge
解密、反序列化时间
接收请求
JS网络接口回调
Native 网络回调
反序列化

网络性能优化

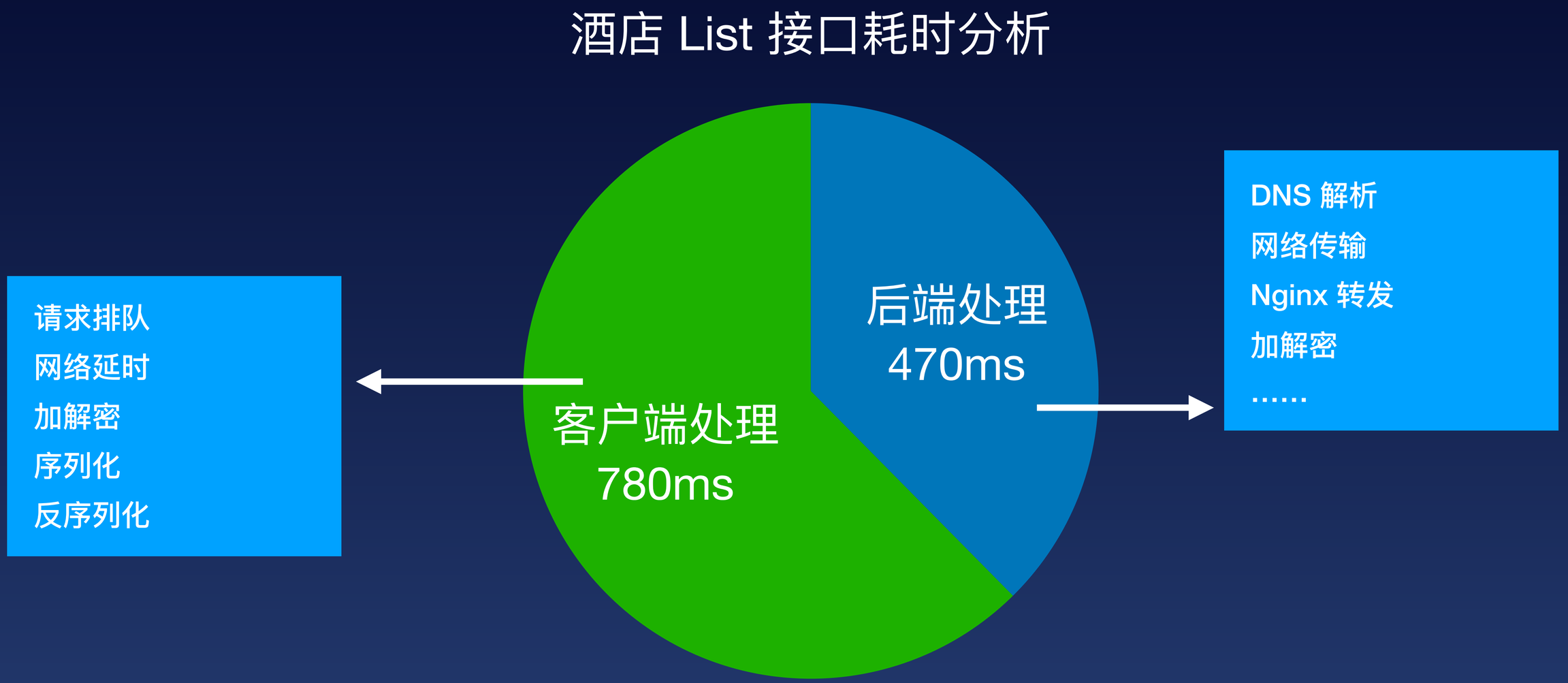


用户感受到的时间 = D1 + 真正请求时间 + D2

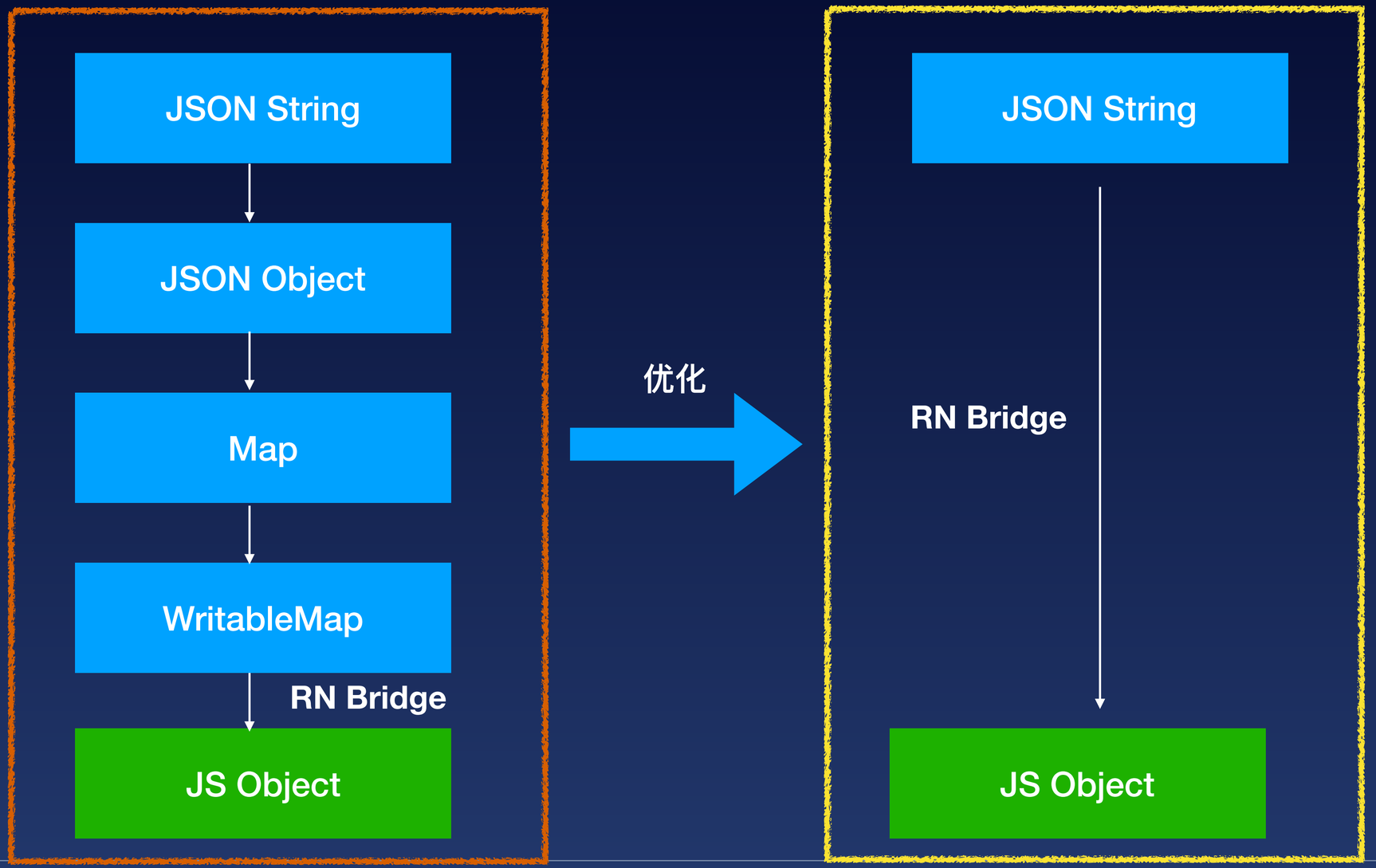
当 CPU 很忙时，D1、D2 会变大，比如 App 启动时，经过统计请求时间会比真正网络请求时间大 1/3 左右

CPU 不忙的时候，也差不多比真正请求时间多 10%左右

网络性能优化


酒店 List 接口耗时分析
后端处理
470ms
客户端处理
780ms
请求排队
网络延时
加解密
序列化
反序列化
DNS 解析
网络传输
Nginx 转发
加解密
……

优化反序列化方式



优化效果：

1M 的 response Data

减少 150ms 左右

网络性能优化

1. 请求、响应数据 Size 优化

- 删除无用的字段
- 精简、合并字段

2. 拆分接口

优化效果

Size 方面：

45KB ——> 5KB

时间方面：

Adr 减少 600ms

iOS 减少 400ms

网络性能优化

3.请求串行改并行

- 解耦数据，并行请求
- 组件渲染顺序优化

优化效果

Adr 减少 900ms

iOS 减少 600ms

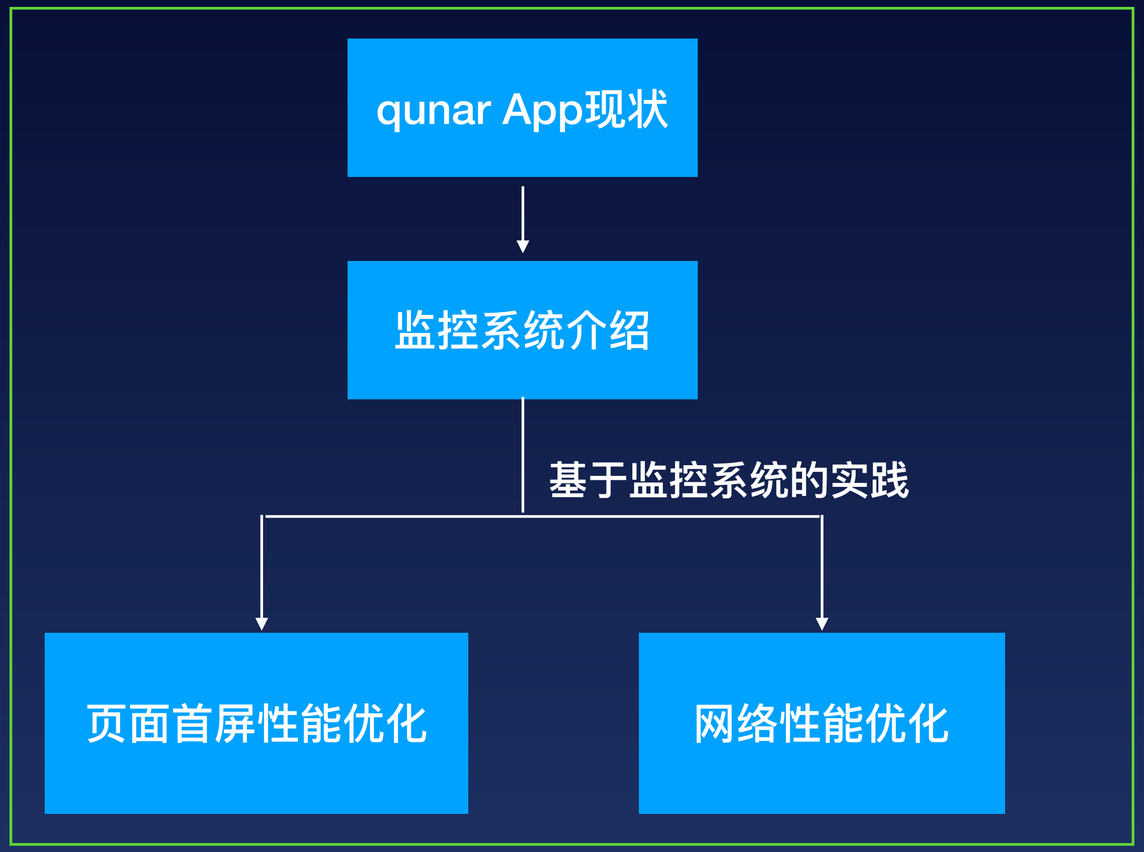
网络性能优化

4.其它网络性能优化方式

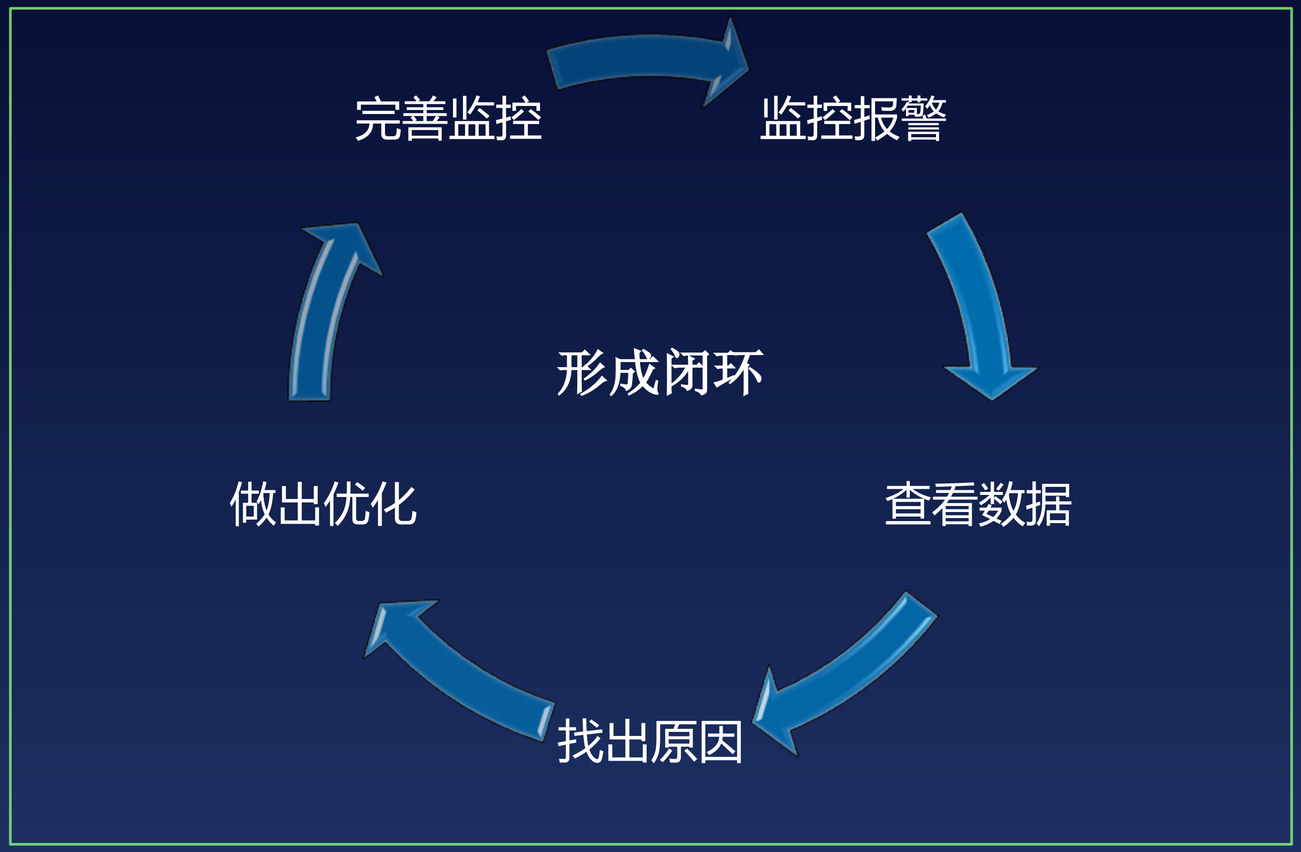
- 预请求
- 数据缓存
- 使用 HTTP/2
- 本地 DNS
- 服务端、客户端一起优化

总结与展望

总结与展望


qunar App现状
监控系统介绍
基于监控系统的实践
页面首屏性能优化
网络性能优化

总结


完善监控
监控报警
形成闭环
查看数据
找出原因
做出优化

展望

Geekbang 极客邦科技 | InfoQ 极客传媒

InfoQ官网 全新改版上线

促进软件开发领域知识与创新的传播

关注InfoQ网站
第一时间浏览原创IT新闻资讯

免费下载迷你书
阅读一线开发者的技术干货

极客大学

前端训练营

用3个月时间，彻底学透前端开发必备技能

- 线下线上混合式学习
- 名师手把手教学
- 一线大厂项目实操
- 毕业即享内推服务

了解详情

讲师：程劭非（winter）

前手机淘宝前端负责人

THANKS

GMTC
全球大前端技术大会