从重新认识前端渲染开始，小红书的前端性能监控及优化实践

李季骏

小红书社区前端工程师

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

自我介绍

2016 年底加入小红书至今，目前任职于社区前端组，负责了包括小红书前端工程化、服务端同构、混合开发容器层建设（WebView、React Native）等方向。

对于跨平台混合开发场景有丰富的实践，并在效率、性能等方向累积了一定的工程化解决方案经验。

目录

- App WebView 开发模式给前端开发者提出的挑战，传统的（已有的） Web 渲染性能衡量方式的不切实用
- 核心性能体验指标的梳理和工程化的采集/监控方案
- 基于关键渲染帧的过程拆解，和多个场景下的实践
- 借助 App Webview 的容器能力下的探索
- 一点工程化的思考

衡量 Web 渲染性能前，几个需要被解答的问题

- 提炼影响用户决策的感官指标
  - 渐进式的渲染过程（骨架屏、首屏）
- 跨实现的统一衡量手段
  - client-side-render 与 server-side-render 的横向比较
- End-to-End 的采集方式
  - 采集来自于用户的真实体验
- 拆解影响感官指标的过程因素，尝试归因并优化
  - 找寻关键过程的关键影响因子（网络、机型）
  - 单节点的优化和验证能力

提炼影响用户决策的感官指标

The Metric	The Experience	
First Paint (FP) / First Contentful Paint (FCP)	Is it happening?	Did the navigation start successfully? Has the server responded?
First Meaningful Paint (FMP)	Is it useful?	Has enough content rendered that users can engage with it?
Time to Interactive (TTI)	Is it usable?	Can users interact with the page, or is it still busy loading?
Long Tasks (technically the absence of long tasks)	Is it delightful?	Are the interactions smooth and natural, free of lag and jank?

https://developers.google.com/web/fundamentals/performance/user-centric-performance-metrics#_1

传统的（已有的） Web 渲染性能衡量方式的不切实用

“

它们能帮你分析你的网页可能存在的症结；

它们不太了解你的网页在不同设备的运行情况；

甚至不能告诉你你在完成一项优化工作后给用户带来了多少价值；

它们不应该成为你开展优化工作的起点或终点；

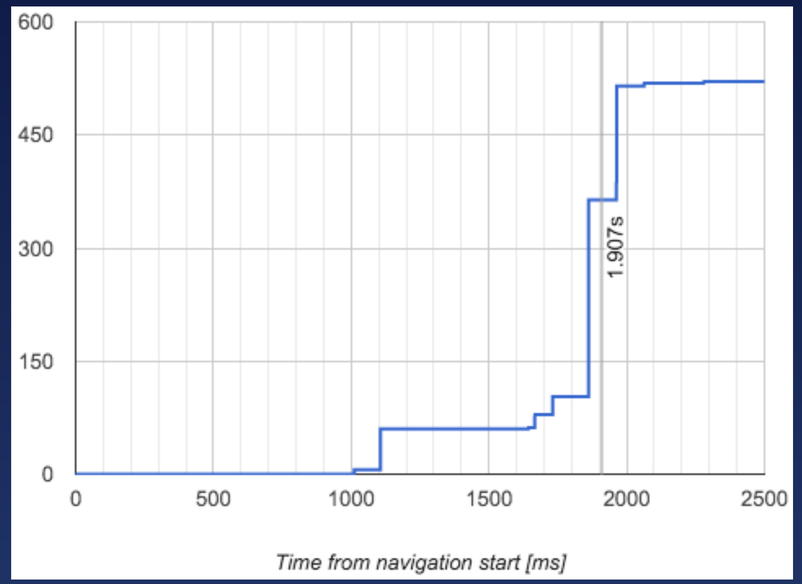
”

—— lighthouse, Web Page Test

在真实用户的设备上衡量这些指标

- 基于 JS 的统计实现原理

基于 Mutation 观察每一次渲染帧，并根据适用的算法计算出 FCP，FMP 等关键渲染帧 timing




```
// Paint-timing collector
observer = new MutationObserver(records => {
  records.forEach(record => {
    if (record.type === 'childList') {
      recordNodes(record.addedNodes)
    }
  })

  pushObservedPoints()
})

observer.observe(document)
```

事实上，这段 code 还能另你重新认识和理解你的应用是如何被渲染的，尤其是当 MVVM 的 Web 编程思想大行其道

工程化的采集/监控方案

采集 → 处理 → 消费

JS SDK

Collector

Emitter

Translator

Visualization

Frontend Engineering, then we have data

透过数据被量化放大的问题

client-side-render 和 server-side-render 的 tradeoff 2500ms / 1750ms

client-side-render

PWA、可以充分调动本地缓存的能力
渐进式的首屏体验

首屏渲染依赖 JS 执行完成
渲染时序处理不当页面可能出现多次跳闪

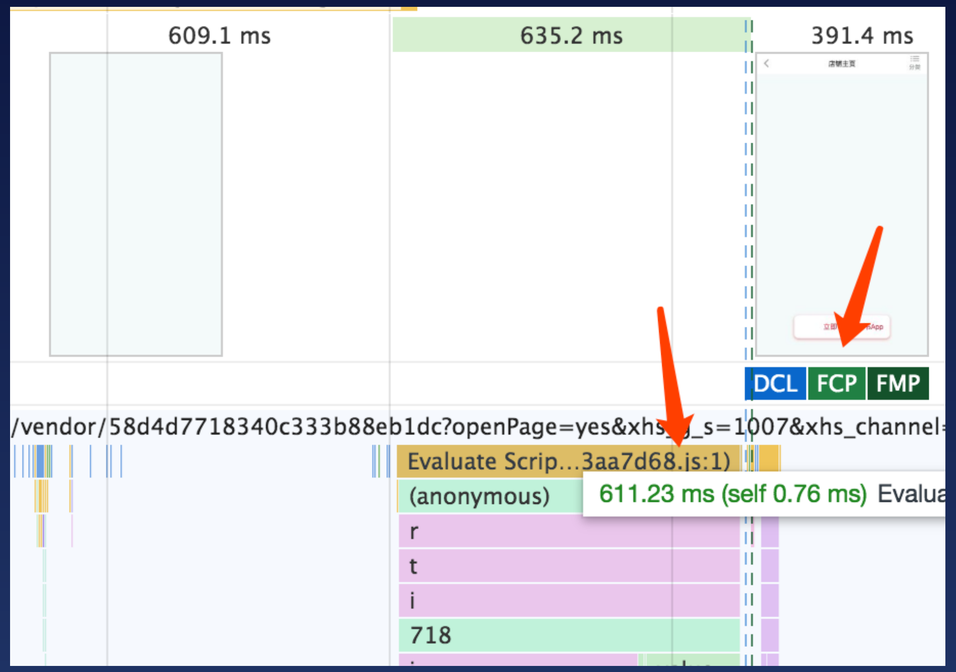
server-side-render

首屏更快

HTML 无法缓存
网络故障时，用户可能面对长时间的白屏

巨大的 JavaScript 启动开销

比起服务端渲染/客户端渲染，可以将启动成本是均摊到整个生命周期，是前端渲染与之比较下难以逾越的鸿沟的本质




```
// Example Component
create() {
  http.get('foo').then(res => {
    this.foo = res.foo
  })
}
```

巨大的 JavaScript 启动开销

- dynamic import

```
// code-splitting
import('foo').then(foo => {
  // do something
})

// dynamic import
const importFoo = () => (
  Promise.resolve(require('foo').default)
)
importFoo().then(foo => {
  // do something
})
```

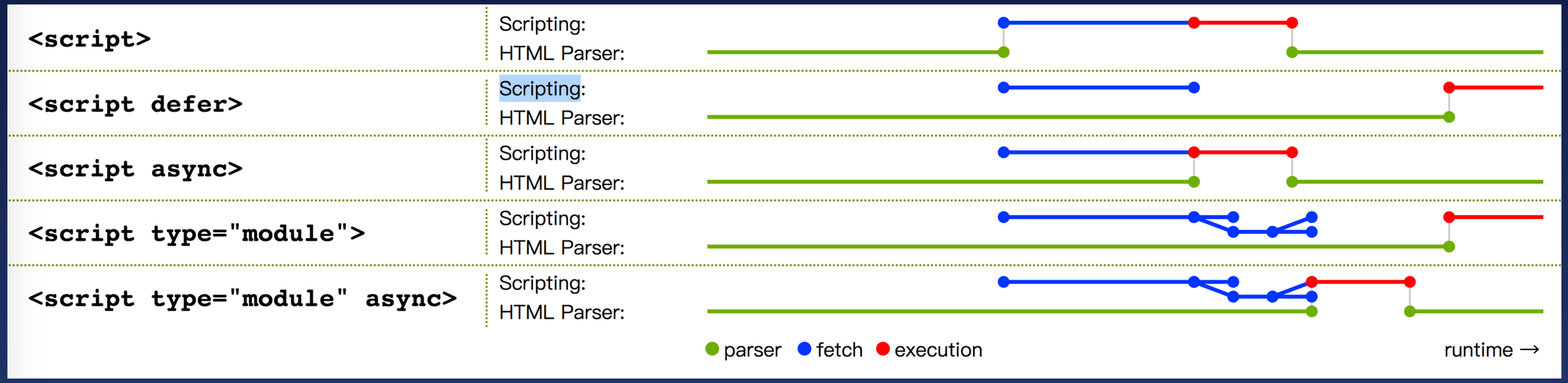
- redundant packages

Redundant-Package	Suggestion	Location
@xhs/ozone-bridge	^2.12.2 -> 2.14.2	@xhs/cube-plugin@2.0.1
@xhs/ozone-detector	^3.1.2 -> 3.2.0	@xhs/cube-plugin@2.0.1
@xhs/logger	^2.1.0 -> 2.2.1	@xhs/launcher@1.4.0
@xhs/ozone-bridge	^2.12.1 -> 2.14.2	@xhs/launcher@1.4.0
@xhs/ozone-detector	^3.1.1 -> 3.2.0	@xhs/launcher@1.4.0
@xhs/ozone-download	^1.1.0 -> 1.3.1	@xhs/launcher@1.4.0
@xhs/logger	0.1.22 -> 2.2.1	@xhs/launcher-plugin-ui

透过数据被量化放大的问题

PWA 就可以实现解决“秒开零白屏”的效果么，哪些因素干扰了骨架图的出现？600ms

- 理解关键渲染路径
- 理解浏览器绘制

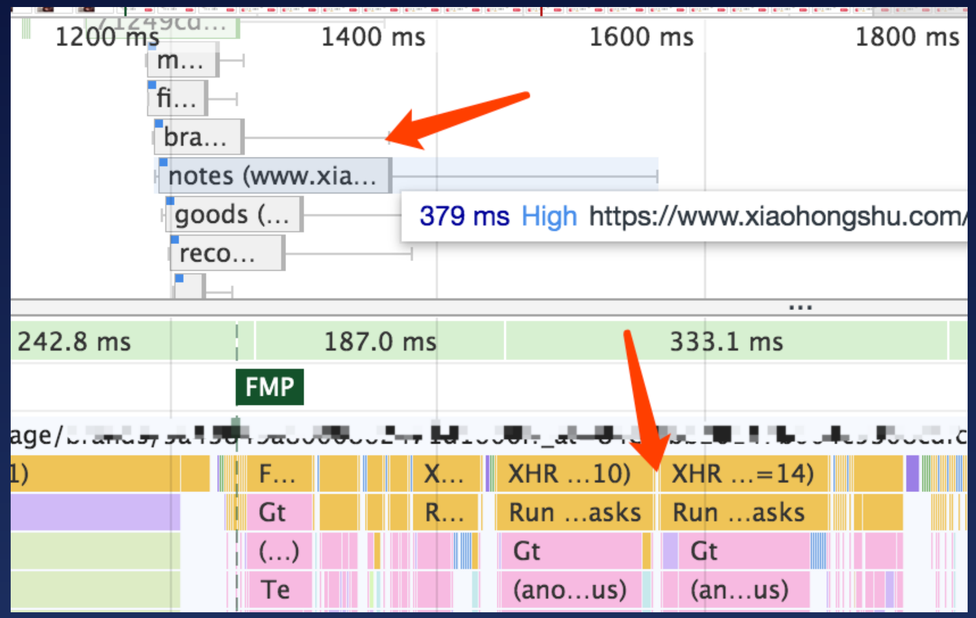


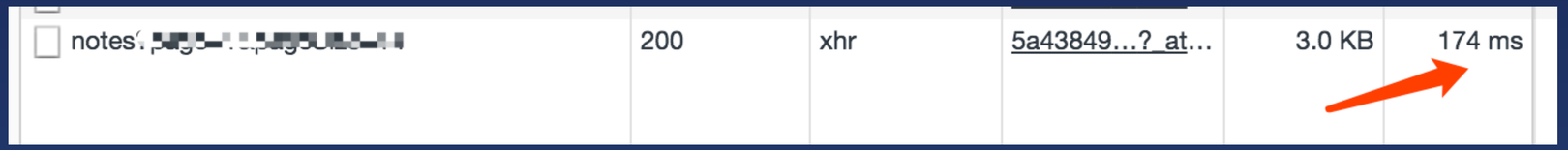
调整脚本加载策略

“defer 的脚本被完全缓存时，并没有遵守规范等待解析结束，反而阻塞了解析与渲染。”

透过数据被量化放大的问题

渲染时序的混乱，渲染不稳定　1350ms ~ 3150ms





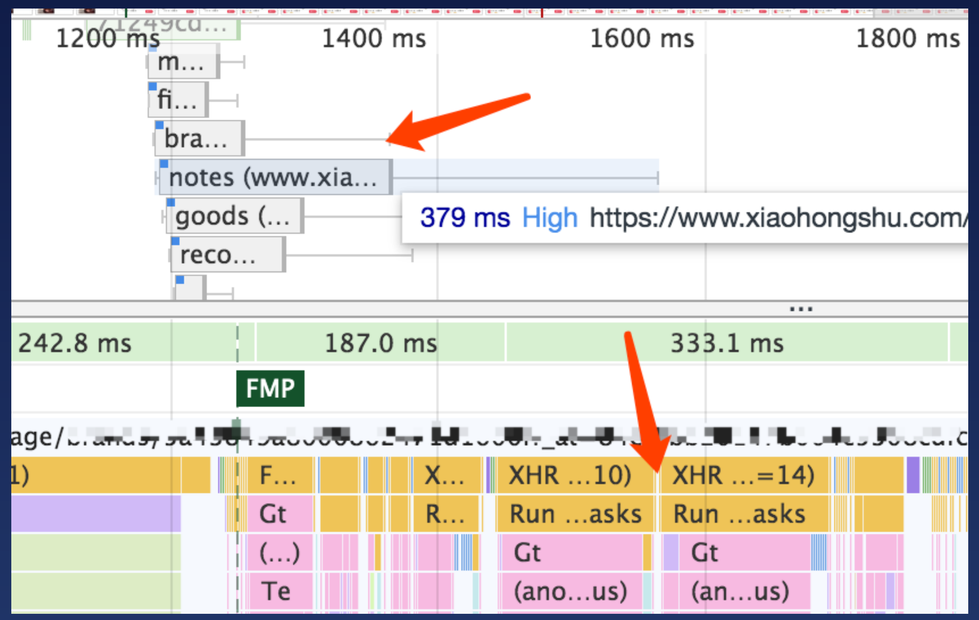
理解 Vue 的渲染触发机制

batcher (watcher)

Register dom-update task which cost large calculation on dom diff by a microtask.

One event loop

update dom



渲染时序完全由网络决定

```
// Main component
mounted() {
  this.fetchNotes().then(notes => {
    this.notes = notes
  })
  this.fetchTags().then(tags => {
    this.tags = tags
  })
  this.fetchGallery().then(gallery => {
    this.gallery = gallery
  })
}
```

```
// Main component
mounted() {
  Promise.all([
    this.fetchNotes(),
    this.fetchTags(),
    this.fetchGallery(),
  ]).then([notes, tags, gallery] => {
    this.notes = notes
    this.tags = tags
    this.gallery = gallery
  })
}
```

通过 Vuex 控制数据流，从而控制渲染时序

```
// Vuex Getters
function notes(state) {
  return state.Note.notes
}

function tags(state) {
  return state.Note.ready ? state.Tag.tags : []
}

function gallery(state) {
  return state.Note.ready && state.Tag.ready ? state.gallery.pics : []
}
```

小红书借助 App Webview 的容器能力下的探索

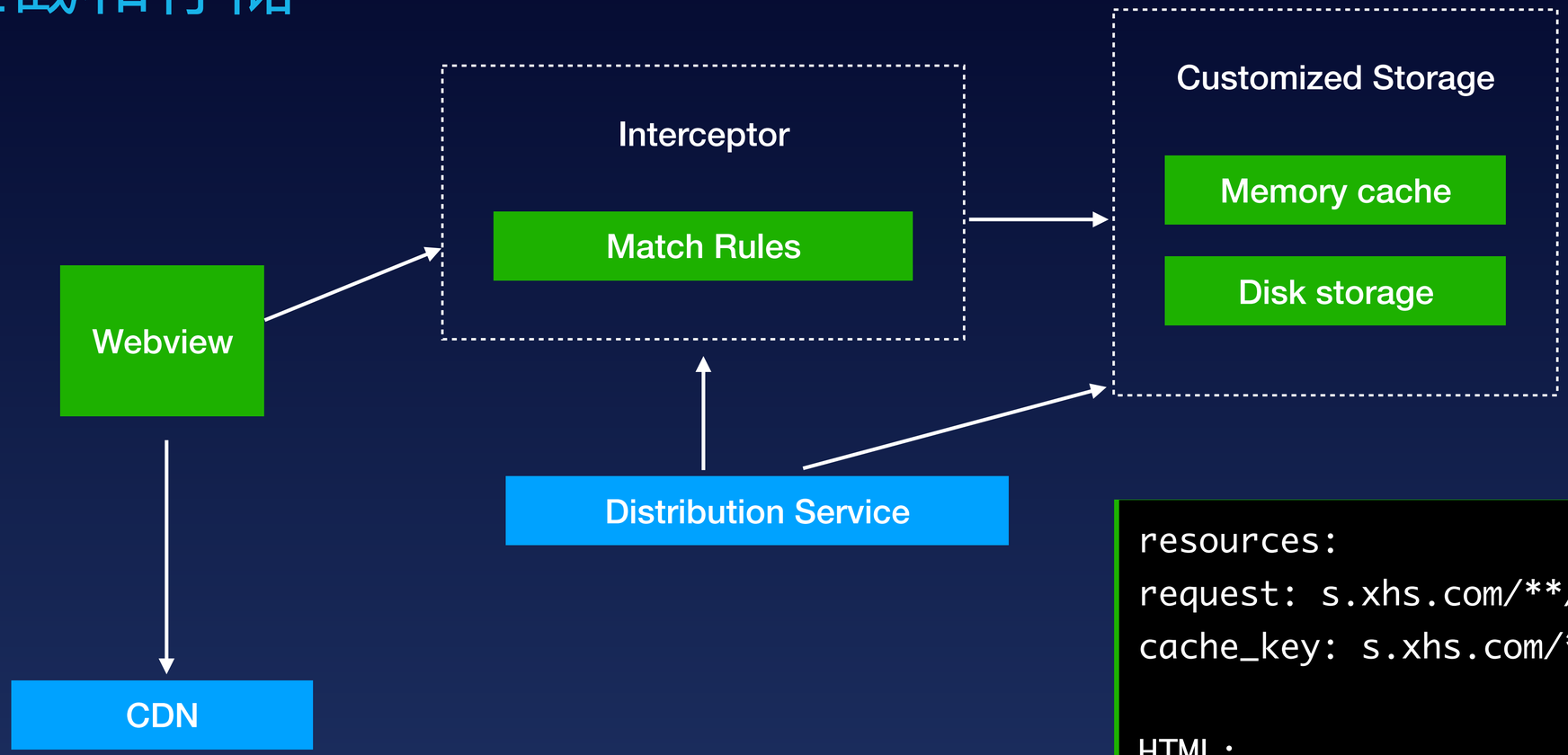
- 并行关键渲染路径上的部分过程
- 空间换时间

资源内置

如何拦截？如何存储？如何更新（下发）？

	如何拦截	如何存储	如何更新（下发）
PWA	service worker：基于浏览器能力的网络层拦截	Browser Storage	基于 service worker 的策略配置（前端可控）
Bowl	改变资源的引用方法，基于 JS library 能力	Browser Storage	基于业务实现的策略配置（前端可控）
小程序	基于 Native 能力	Native App Storage	完全由平台策略决定
Target：高定制化，低侵入的资源内置设施；简单高效的资源更新（下发）策略			

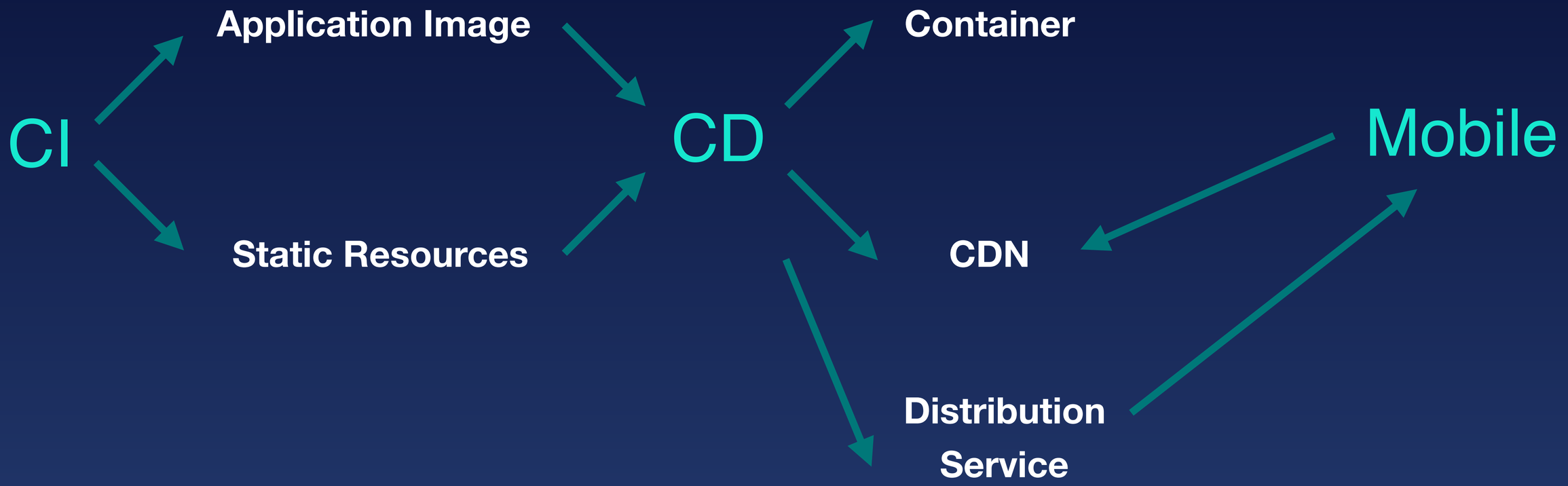
拦截和存储




```
resources:
request: s.xhs.com/**/main.175cae.js
cache_key: s.xhs.com/**/main.175cae.js

HTML:
request: www.xhs.com/item/1000123
cache_key: s.xhs.com/**/index.6045eb9.html
```

资源更新（下发）


CI
Application Image
Static Resources
CD
Container
CDN
Distribution
Service
Mobile

数据预传、数据预取

一些在 Native 开发生态下成熟的解决方案在混合开发场景下的应用

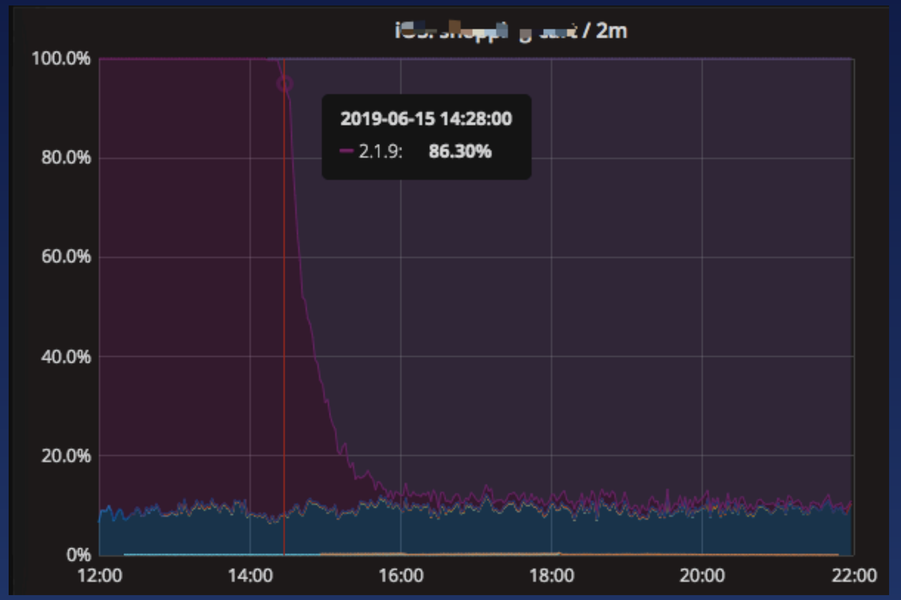
- 数据预传（强依赖入口页）
  - 数据携带的协议
  - 数据使用的 API
  - 底层 Cache 的打通（图片）

- 数据预取（目标页自治）
  - 数据使用的 API
  - 预取规则的下发

充分借助客户端能力，前后端混合渲染方案

一点工程化的思考

- 数据能力
  - 采集 -> 处理 -> 可视化（监控/实验）
- 优化解决方案
  - 持续集成/交付
  - 对传统 web 应用分发形式的打破（分发、容灾）



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

THANKS

GMTC
全球大前端技术大会