XDC 2021 XITU DEVELOPERS CONFERENCE

2021 XITU DEVELOPERS CONFERENCE//
稀土开发者大会
技 术 未 来 · 向 新 而 生

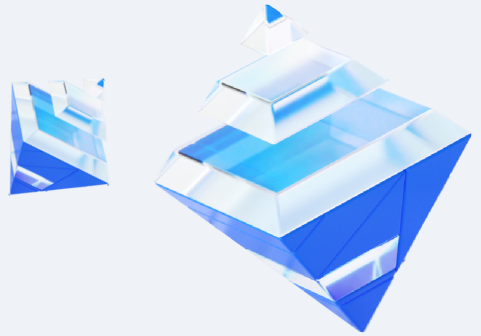
# 前端性能优化实践


王梓童

美团买菜iOS工程师


# 王梓童

- 2016年加入美团
- 深度参与移动端性能优化工作

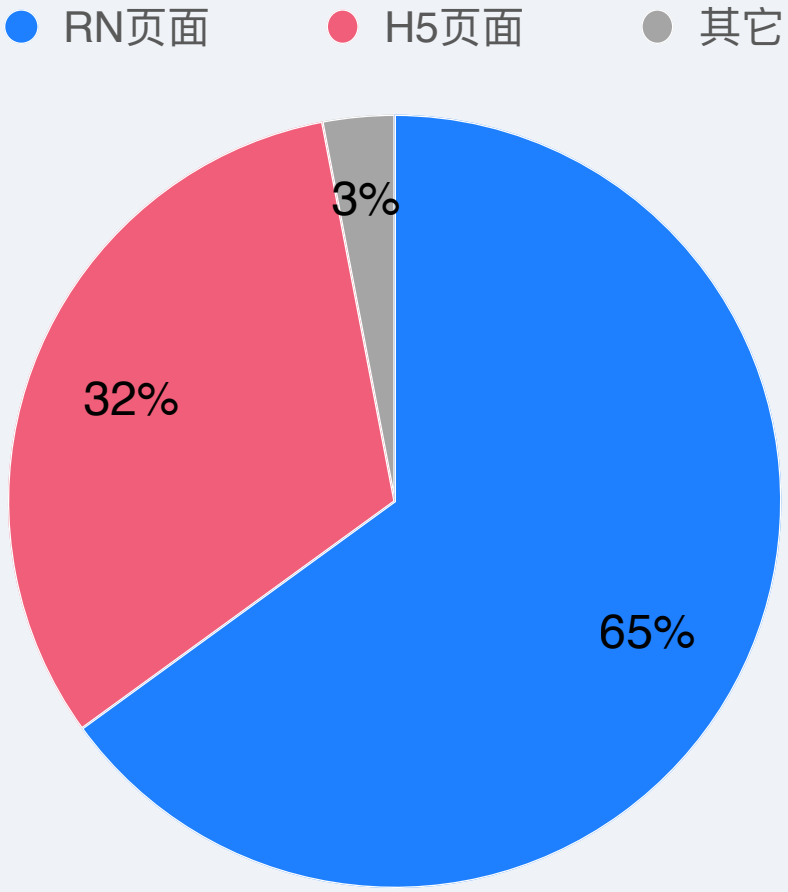


# Contents

## 目　录



# 业务场景及技术选型




包大小

效率

灵活

主流程：RN

营销页：H5

# 为什么做首屏优化

Pinterest减少页面加载时长40%, 搜索和注册数提高了15%

BBC页面加载时长每增加1秒，用户流失10%

DoubleClick发现如果移动网站加载时长超过3秒，53%的用户会放弃

性能 —正反馈→ 业务

# 指标衡量

Google在RAIL性能评估模型中提出，为了持续吸引用户，在 1000 毫秒以内呈现交互内容

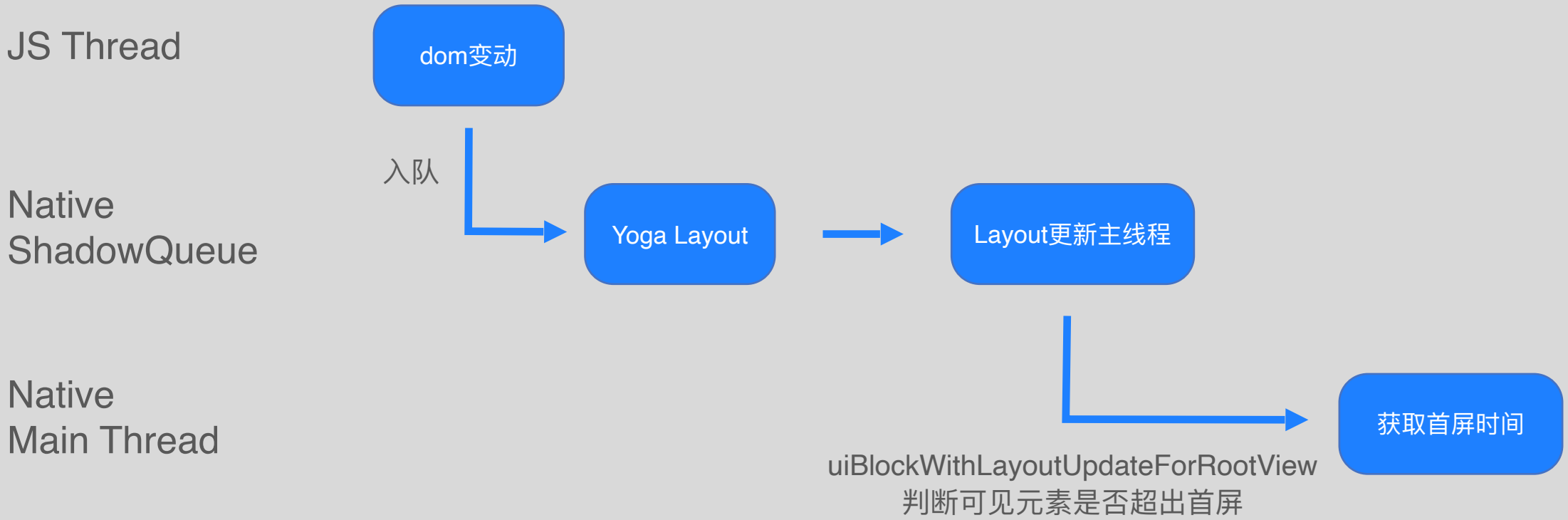
同时建议将“首屏渲染时间”的终点，视为主角元素呈现在屏幕上的时刻

如何定义主角元素呈现在屏幕上？ → 可见元素超出屏幕的时刻

# 指标衡量

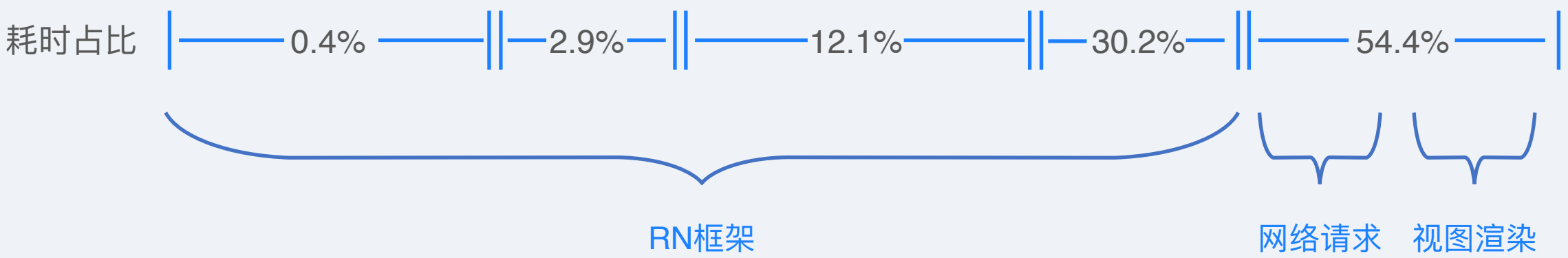

可见元素超出屏幕

# 指标获取

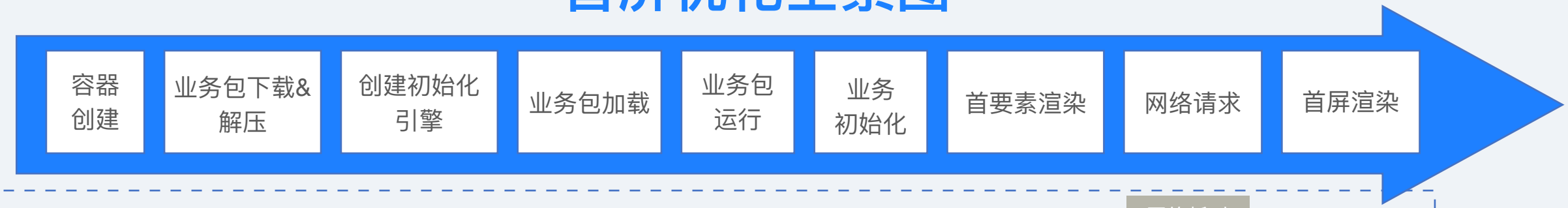

JS Thread
Native
ShadowQueue
Native
Main Thread
dom变动
入队
Yoga Layout
Layout更新主线程
获取首屏时间
uiBlockWithLayoutUpdateForRootView
判断可见元素是否超出首屏

# 优化思路分析


容器创建
业务包下载&解压
本地有包时跳过
创建初始化引擎
有初始化引擎时跳过
业务包加载
引擎复用时跳过
业务包运行
业务初始化
首要素渲染
网络请求
首屏渲染


耗时占比
0.4%
2.9%
12.1%
30.2%
54.4%
RN框架
网络请求
视图渲染

# 首屏优化全景图




影响因素

下载时机
网络状况
SDK初始化
预加载时机
包大小
index大小
桥耗时
网络耗时
UI复杂度
JS执行速度

优化策略

包预置/预下载
预初始化
分包懒加载
包内懒加载
分步加载
分步渲染
业务预热
数据带入
请求前置
引擎复用
静态化
预加载
引擎升级
CodeCache

度量工具

RN秒开SDK

监控运维

日报
播报
上线卡控

分析工具

RN秒开诊断工具
RN Debugger
Node Tree

# 重点优化措施

RN框架优化

网络请求优化

视图渲染优化

# RN框架优化

**RN框架优化**

预加载

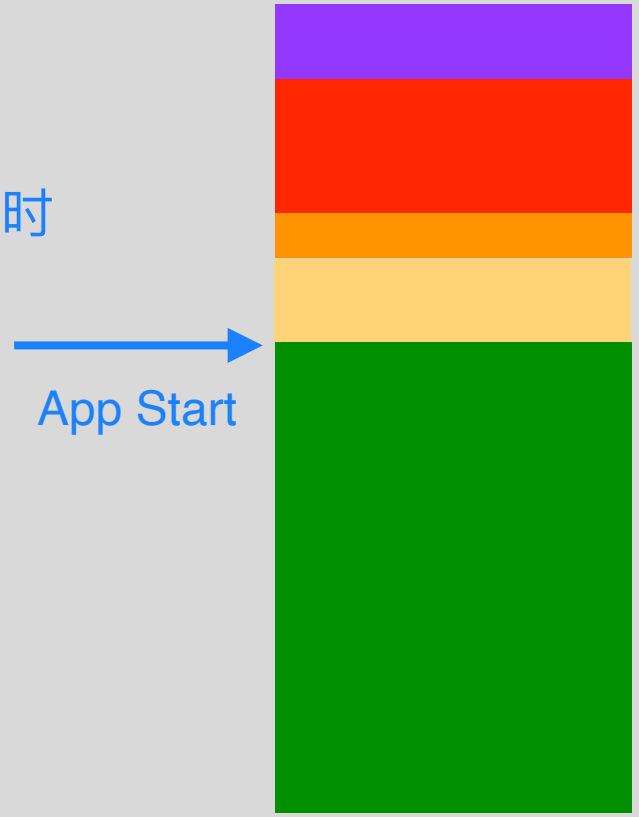
CodeCache

bundle体积优化

懒加载

# 预加载

RN启动过程各阶段耗时

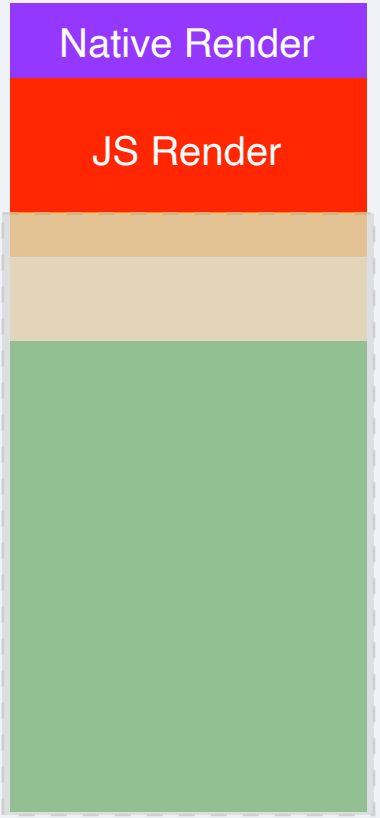



- Native Render
- JS Render
- Fetch
- Before Fetch
- JS Init + Require
- Native Init

# 预加载


**阶段一**

提前加载基础bundle，
业务bundle



# 预加载

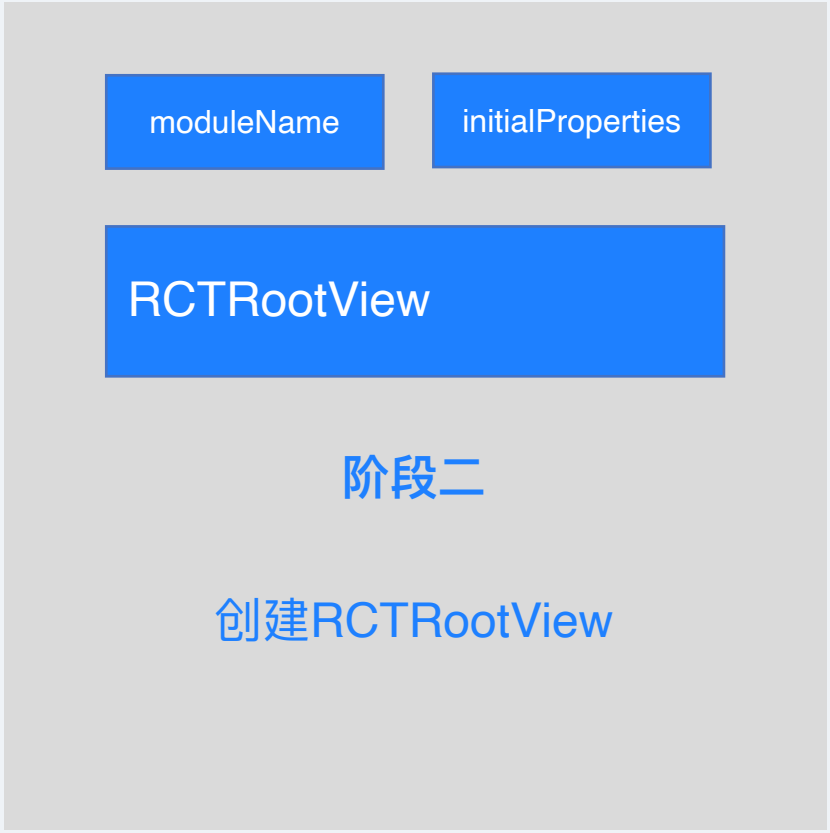

**阶段一**

提前加载基础bundle，业务bundle

控制同时进行预加载的数量，减少对内存的占用

结合业务场景，用户画像智能预加载，提高预加载命中率

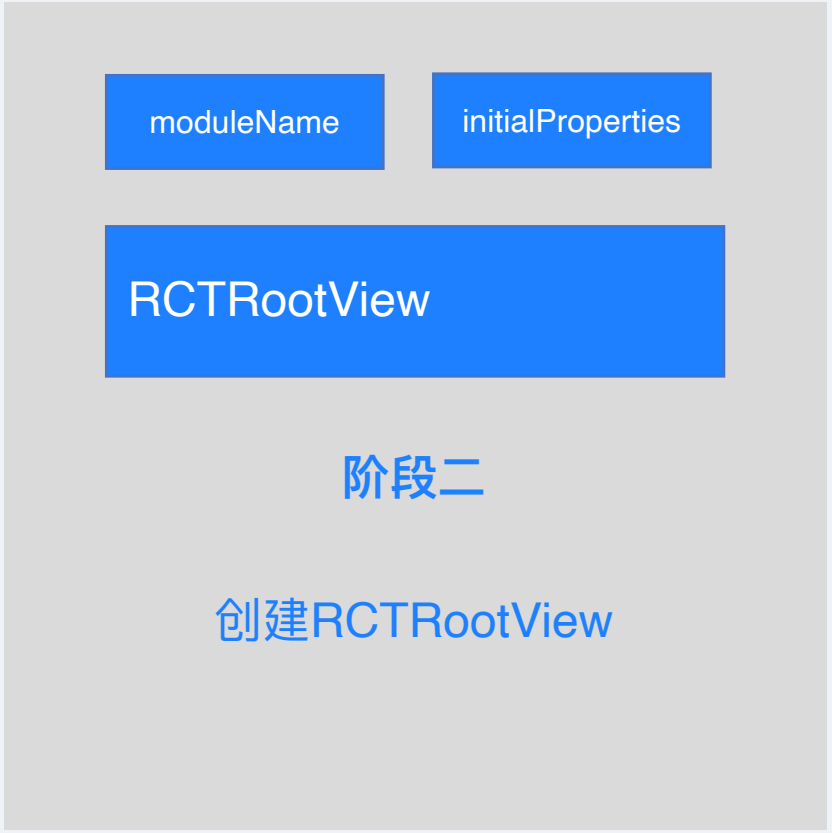
# 预加载



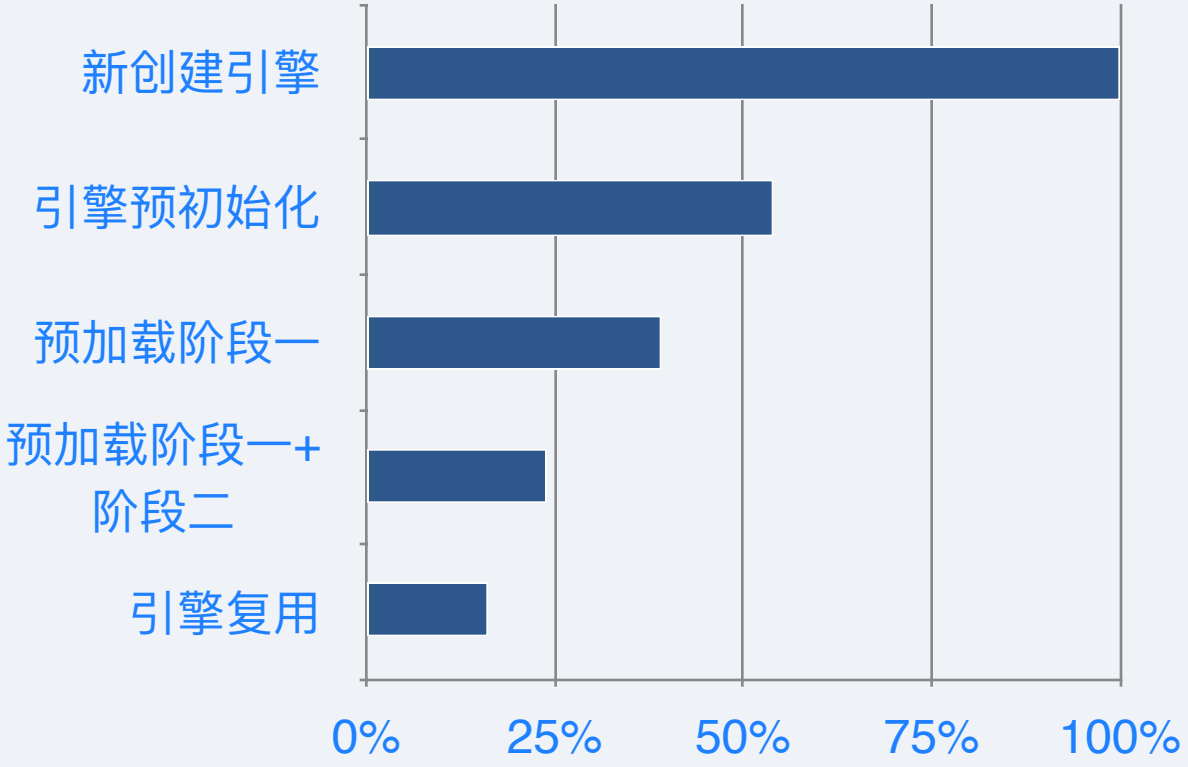

执行预加载阶段二会短时间占用主线程资源，影响上游页面的加载时间，需要综合谨慎地评估它的性能影响

需要修改业务JS代码进行部分屏蔽操作，比如说屏蔽掉后台请求和数据埋点，避免影响后台接口请求量和数据埋点准确性。

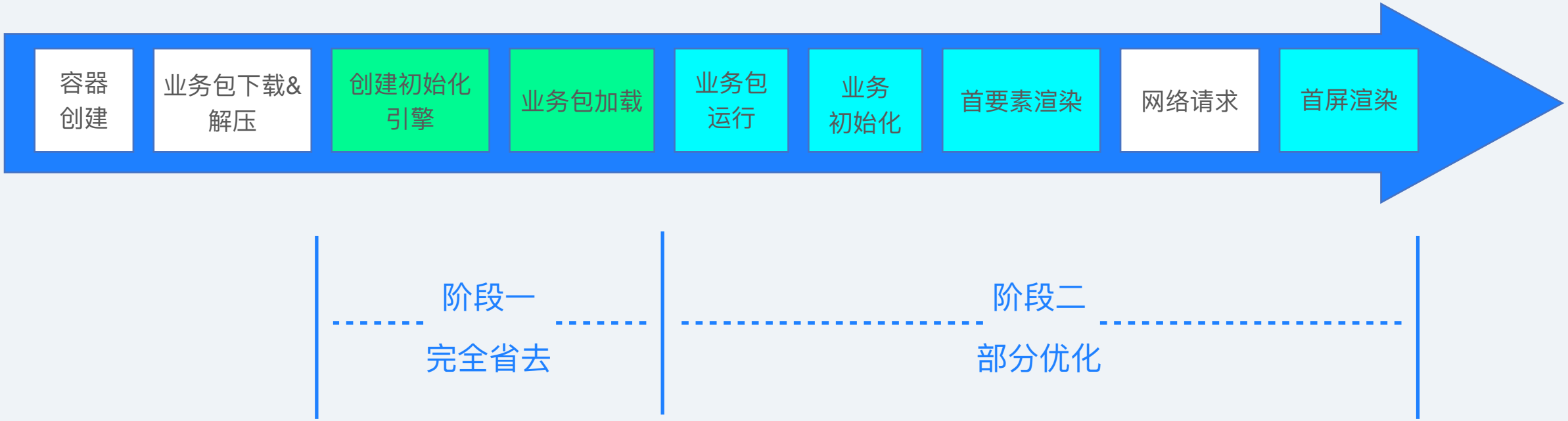
# 预加载




不同引擎加载策略下FCP时间（以新创建引擎为基准）

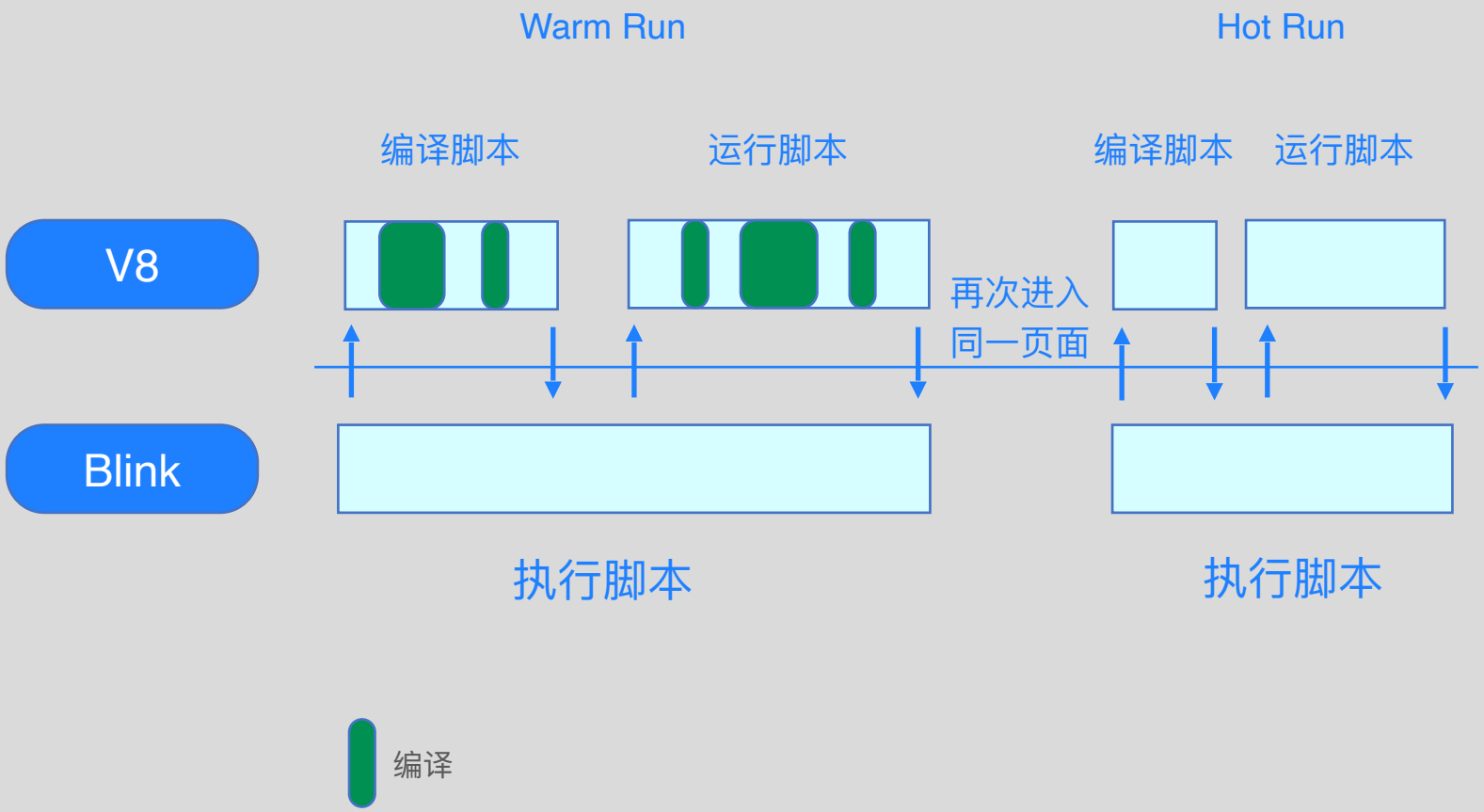


# 预加载

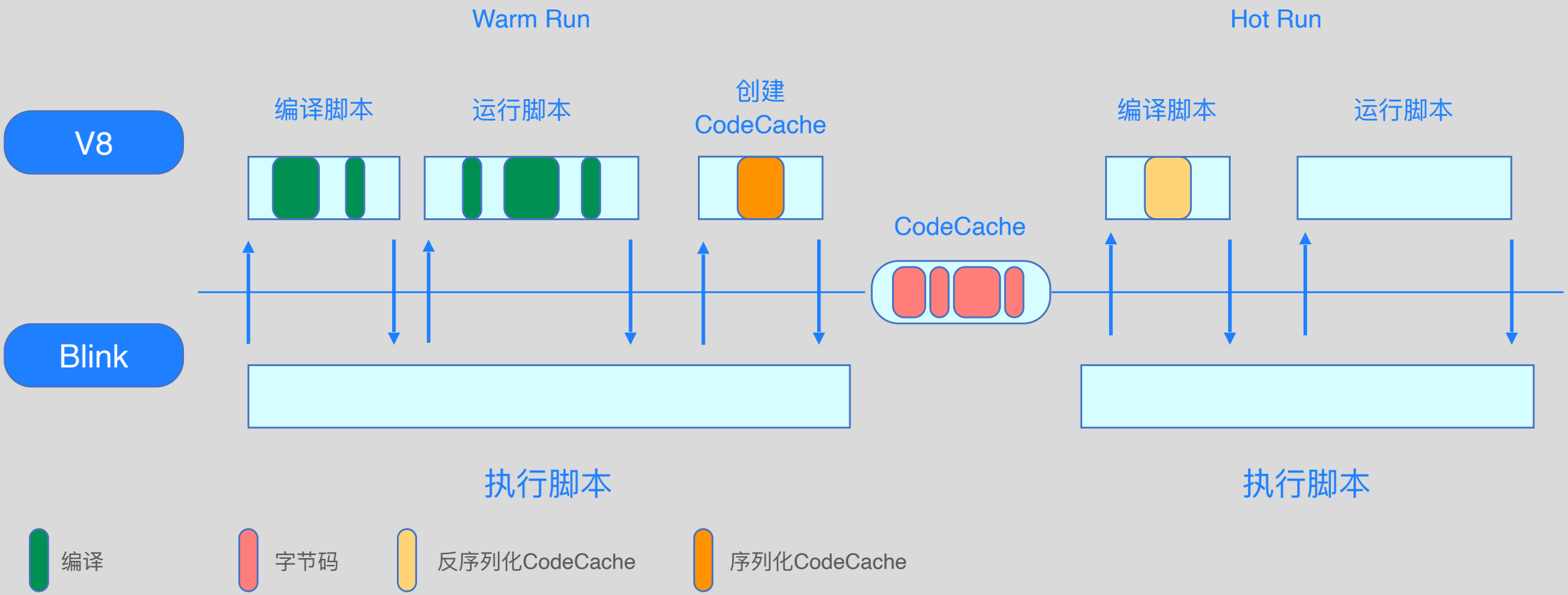

容器创建
业务包下载&解压
创建初始化引擎
业务包加载
业务包运行
业务初始化
首要素渲染
网络请求
首屏渲染
阶段一
完全省去
阶段二
部分优化

# CodeCache

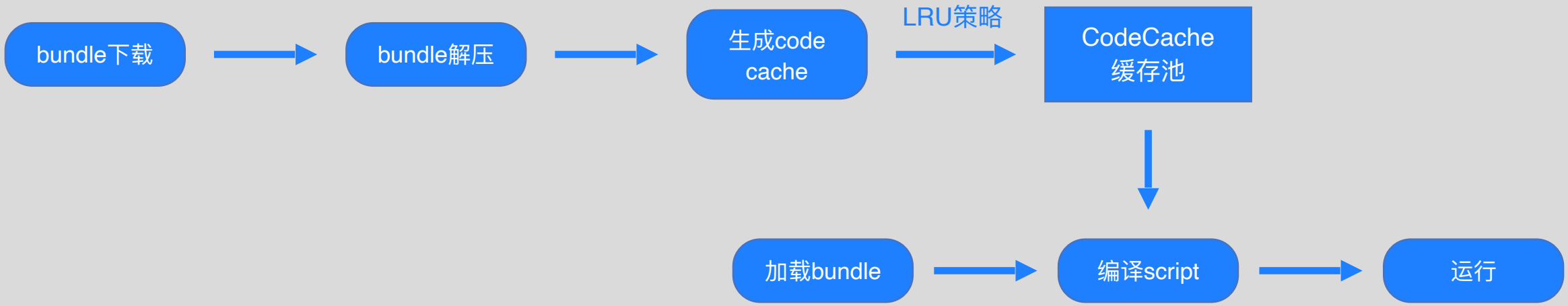
引擎复用时，JS引擎可以复用同一脚本的编译结果，从而跳过编译过程。但当引擎被销毁时，编译结果也随之消失。



# CodeCache


Warm Run
Hot Run
V8
Blink
编译脚本
运行脚本
创建
CodeCache
CodeCache
编译脚本
运行脚本
执行脚本
执行脚本
编译
字节码
反序列化CodeCache
序列化CodeCache

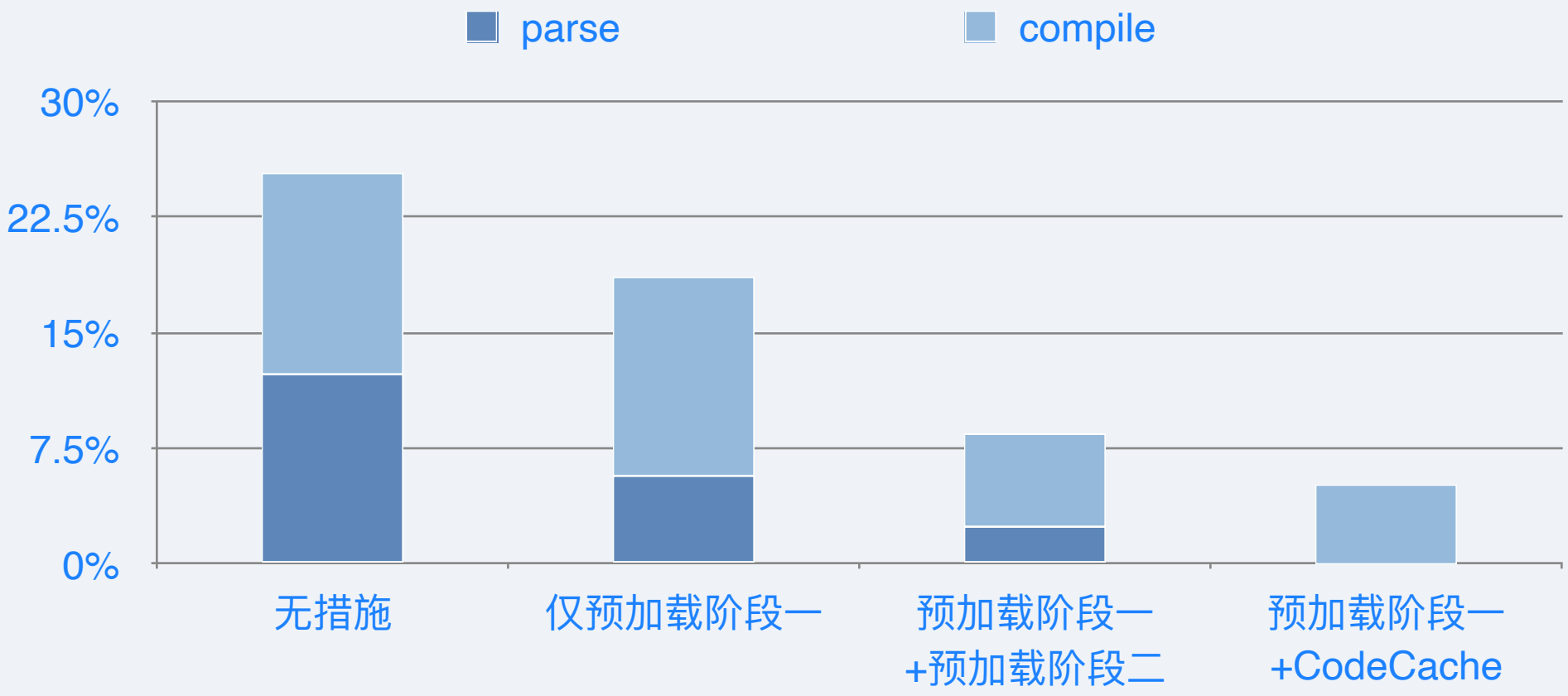
# CodeCache




CodeCache体积大小约为原bundle大小的1.8倍
使用LRU策略控制CodeCache体积大小

# CodeCache

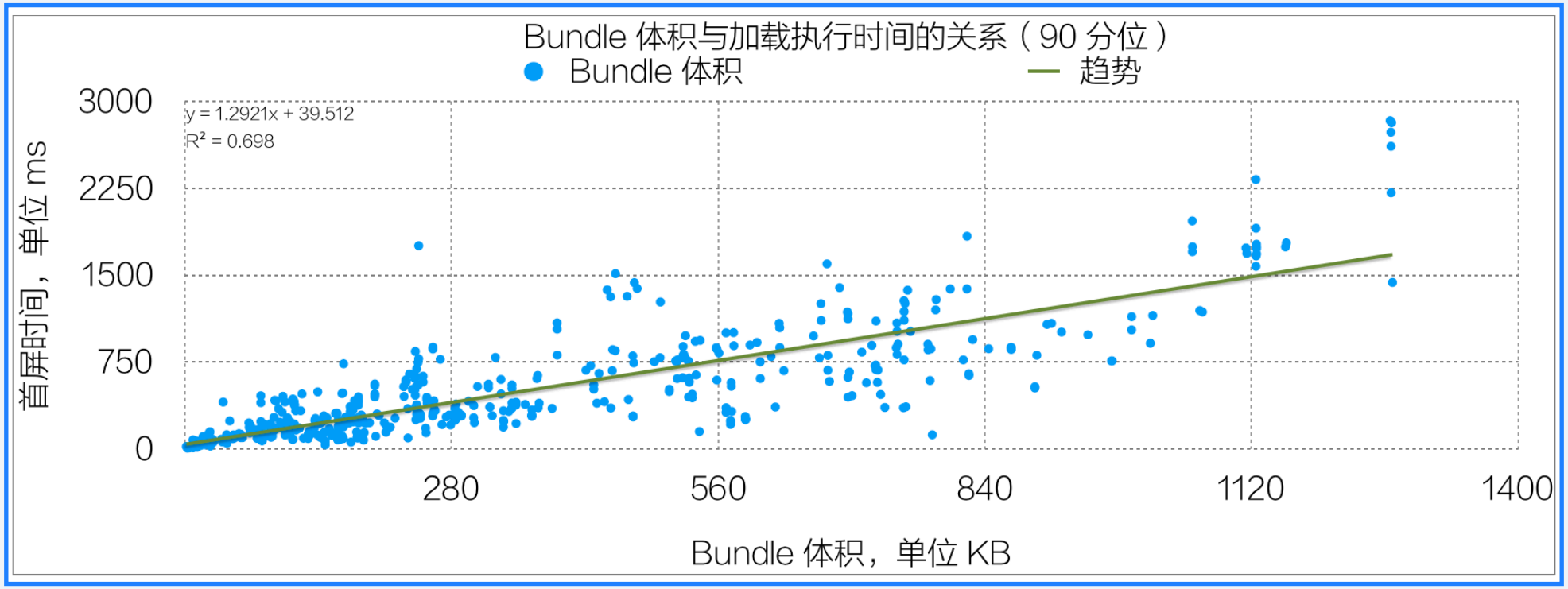
不同优化措施前2s加载过程中parse，
compile耗时占比



# CodeCache


容器创建
业务包下载&解压
创建初始化引擎
业务包加载
业务包运行
业务初始化
首要素渲染
网络请求
首屏渲染
阶段一
完全省去
CodeCache
部分优化

# bundle体积优化


Bundle 体积与加载执行时间的关系（90 分位）
Bundle 体积
趋势
y = 1.2921x + 39.512
R² = 0.698
3000
2250
1500
750
0
首屏时间，单位 ms
280
560
840
1120
1400
Bundle 体积，单位 KB

# bundle体积优化

Tree Shaking

- 裁剪Dead Code

按需加载打包

- 抽象公共模块，提高代码复用率
- 重构基础工具类库，支持按需加载打包

# 懒加载

RAMbundle

InlineRequire

# 懒加载

Index.ts

InlineRequire

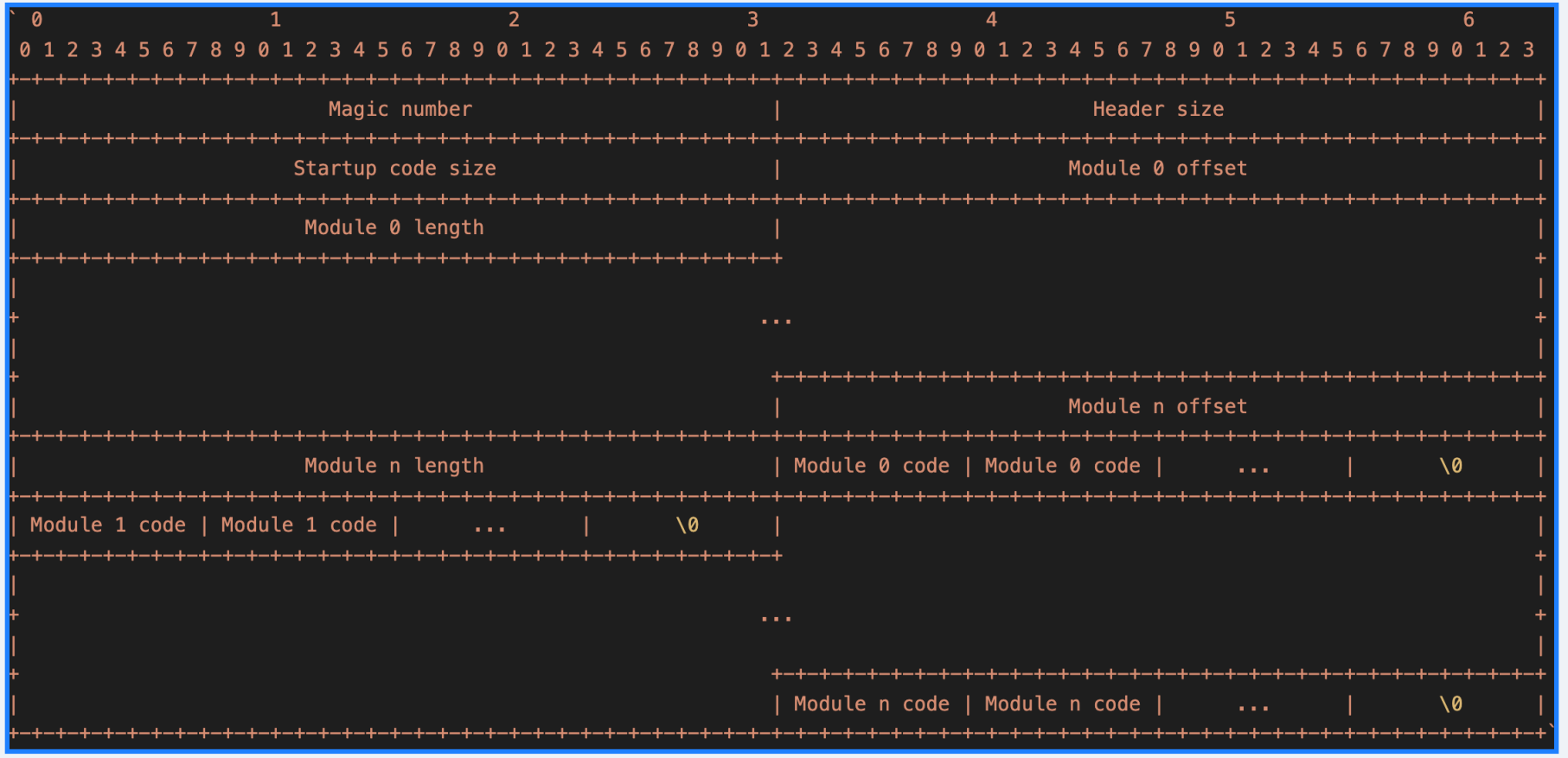
```
setTimeout(() => {
  console.log("时间到");
  const a = require("./a.ts").default;
  console.log(a);
}, 10000);
console.log("首屏加载完成");
```

non-InlineRequire

```
import a from "./a";
setTimeout(() => {
  console.log("时间到");
  console.log(a);
}, 10000);
console.log("首屏加载完成");
```

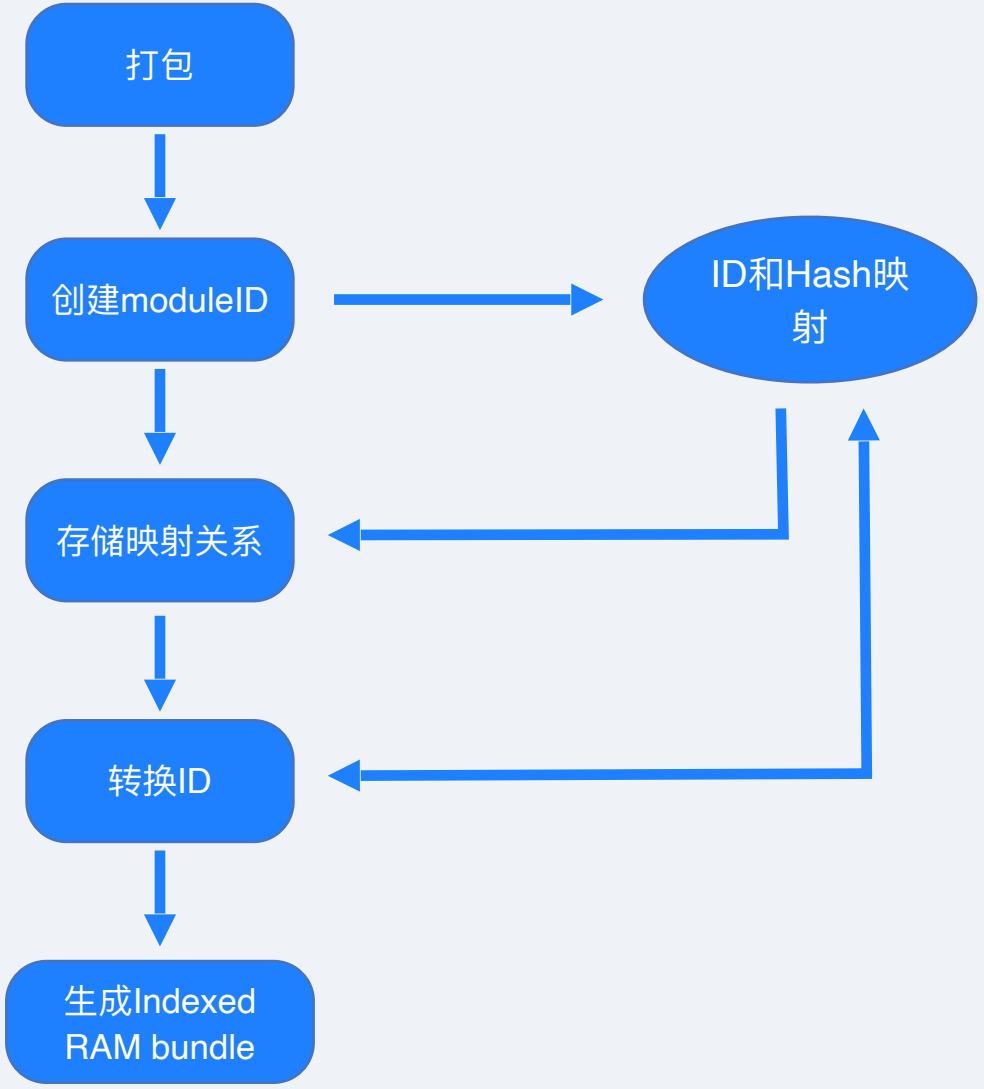
# 懒加载

## Indexed RAM bundle

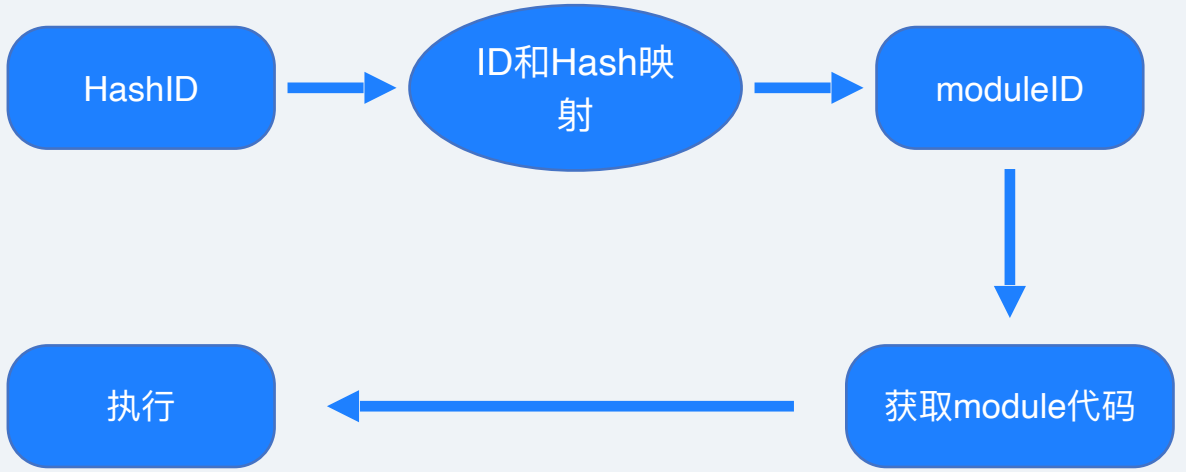


# 懒加载

## Indexed RAM bundle生成步骤




## 代码执行步骤



# 懒加载

VeryExpensive.js

```
import React, { Component } from 'react';
import { Text } from 'react-native';
// ... import some very expensive modules

// You may want to log at the file level to verify when this is happening
console.log('VeryExpensive component loaded');

export default class VeryExpensive extends Component {
  // lots and lots of code
  render() {
    return <Text>Very Expensive Component</Text>;
  }
}
```

假定这是一个逻辑非常重的组件

# 懒加载

OptimizedVeryExpensive.js

```
import React, { Component } from 'react';
import { TouchableOpacity, View, Text } from 'react-native';

let VeryExpensive = null;

export default class Optimized extends Component {
  state = { needsExpensive: false };

  didPress = () => {
    if (VeryExpensive == null) {
      VeryExpensive = require('./VeryExpensive').default;
    }

    this.setState(() => ({
      needsExpensive: true,
    }));
  };

  render() {
    return (
      <View style={{ marginTop: 20 }}>
        <TouchableOpacity onPress={this.didPress}>
          <Text>Load</Text>
        </TouchableOpacity>
        {this.state.needsExpensive ? <VeryExpensive /> : null}
      </View>
    );
  }
}
```

# 懒加载

进一步优化，获取首屏所需js不进行懒加载

```
const modules = require.getModules();
const moduleIds = Object.keys(modules);
const loadedModuleNames = moduleIds
  .filter(moduleId => modules[moduleId].isInitialized) // 已经初始化的模块
  .map(moduleId => modules[moduleId].verboseName);
const waitingModuleNames = moduleIds
  .filter(moduleId => !modules[moduleId].isInitialized) // 未初始化的模块
  .map(moduleId => modules[moduleId].verboseName);

//
console.log(
  '已经加载的模块数量:',
  loadedModuleNames.length,
  '等待的模块数量:',
  waitingModuleNames.length
);

// 将输出的内容放到 scripts/modulePaths.js 文件中。
console.log(`module.exports = ${JSON.stringify(loadedModuleNames.sort())};`);
```

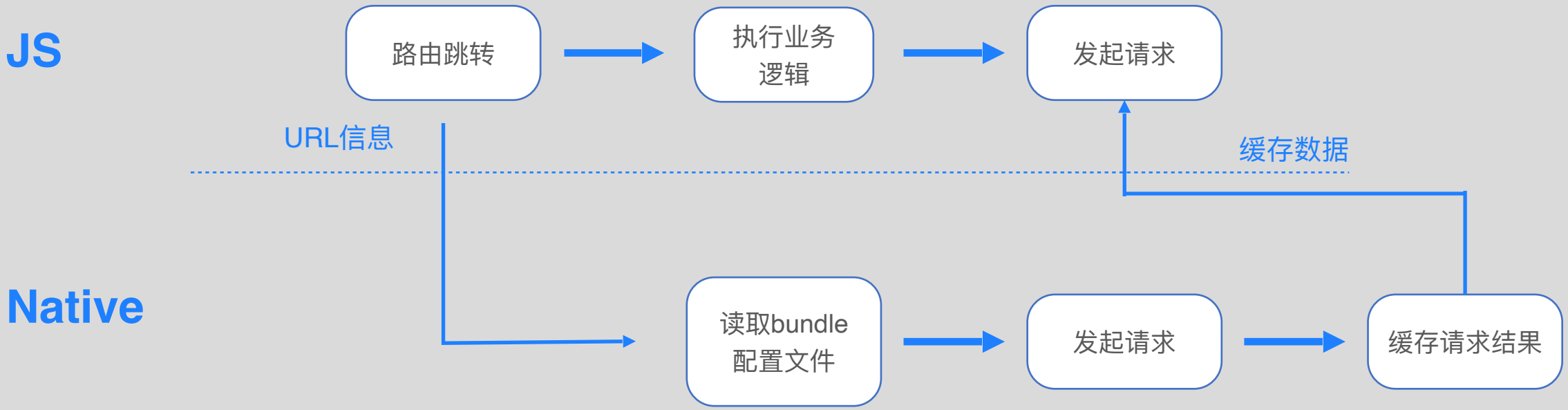
# Bundle体积优化&懒加载

# 网络请求优化

# 网络请求优化


T2时间是否可以被消除?

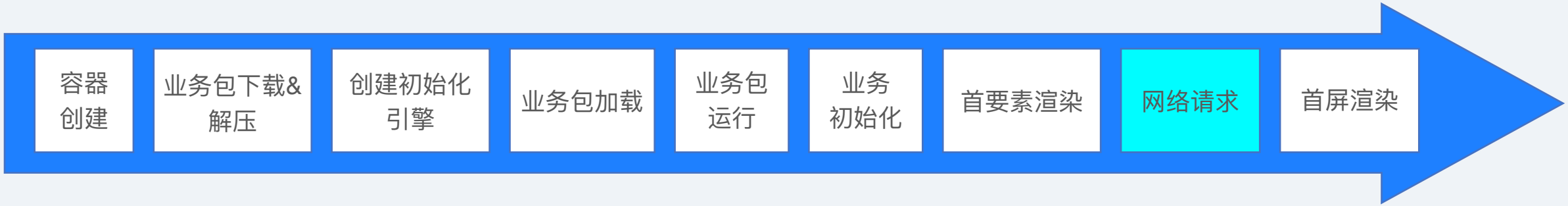
# 请求前置


JS
路由跳转
执行业务逻辑
发起请求
URL信息
缓存数据
Native
读取bundle配置文件
发起请求
缓存请求结果

# 网络请求优化


T1
T2’
趋于0
T3
JS
容器创建
业务包下载&解压
创建初始化引擎
业务包加载
业务包运行
业务初始化
首要素渲染
网络请求
首屏渲染
Native
网络请求
T2

# 网络请求优化




请求前置

部分或全部优化

# 视图渲染优化

分步渲染

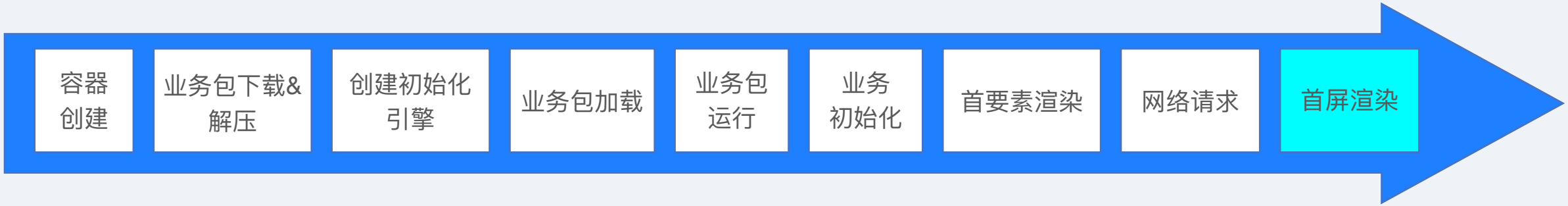
# 分步渲染

```
render () {
    ...
    <Priority
        priority={0}
    >
        <ComponentA />
    </Priority>

    <Priority
        priority={1}
    >
        <ComponentB />
    </Priority>

     <Priority
        priority={2}
    >
        <ComponentC />
    </Priority>
    ...
}
```

设定模块优先级，优先渲染首屏模块

可配合懒加载一起使用

# 视图渲染优化

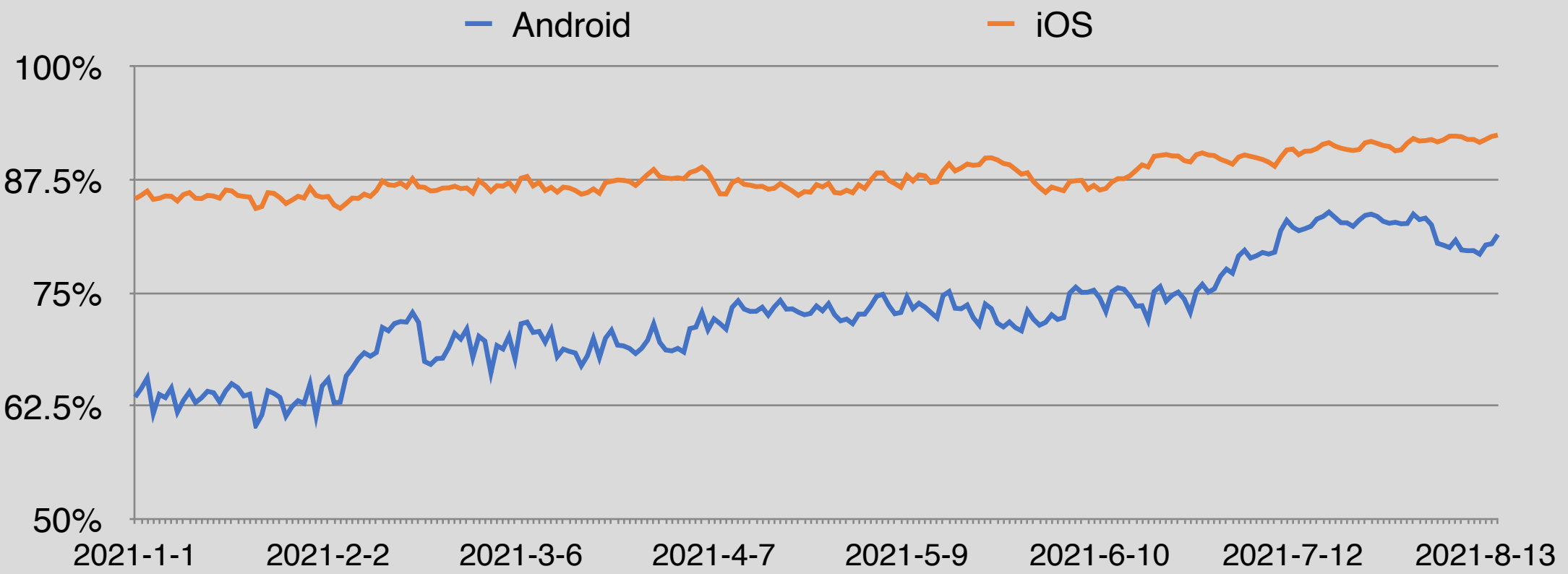



分步渲染

部分优化

# 优化效果

首屏时间小于1s的频率



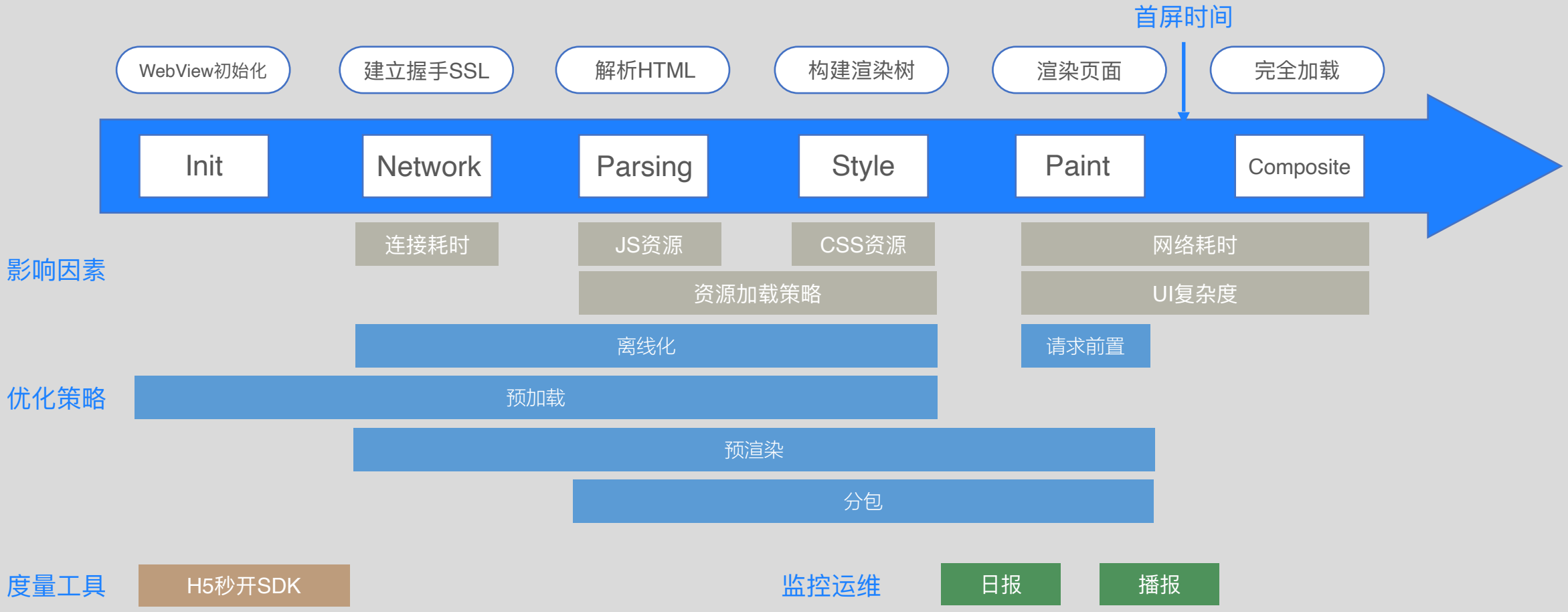
# 优化手段汇总

| 优化分类 | 方案 | 预期收益 | 适合场景 | 备注 | 业务改造成本 |
| --- | --- | --- | --- | --- | --- |
| RN框架优化 | 预加载阶段一 | 400ms | 所有但需配置 | 注意控制内存占用 | 无 |
| RN框架优化 | 预加载阶段一+预加载阶段二 | 600ms | 所有但需配置 | 副作用较大，尽量不使用 | 较大，需屏蔽请求和埋点 |
| RN框架优化 | 预加载阶段一CodeCache | 600ms | 所有但需配置 | 注意磁盘占用 | 无 |
| RN框架优化 | 包体积缩减 | 300ms | 所有 | 建议控制包大小在1M以内 | 无 |
| RN框架优化 | 懒加载 | 500ms | 非首屏元素过多 | 适合详情页，首页 | 中等，需修改业务代码写法 |
| 网络请求优化 | 请求前置 | 300ms | 有请求耗时时 | | 无 |
| 视图渲染优化 | 分步渲染 | 200ms | 非首屏元素过多 | 适合详情页，首页 | 中等，需修改业务代码写法 |

# H5优化实践

## 如何获得首屏时间?

# 优化思路分析


首屏时间
WebView初始化
建立握手SSL
解析HTML
构建渲染树
渲染页面
完全加载
Init
Network
Parsing
Style
Paint
Composite
影响因素
连接耗时
JS资源
CSS资源
网络耗时
资源加载策略
UI复杂度
优化策略
离线化
请求前置
预加载
预渲染
分包
度量工具
H5秒开SDK
监控运维
日报
播报

# 重点优化措施

资源优化

网络请求优化

渲染优化

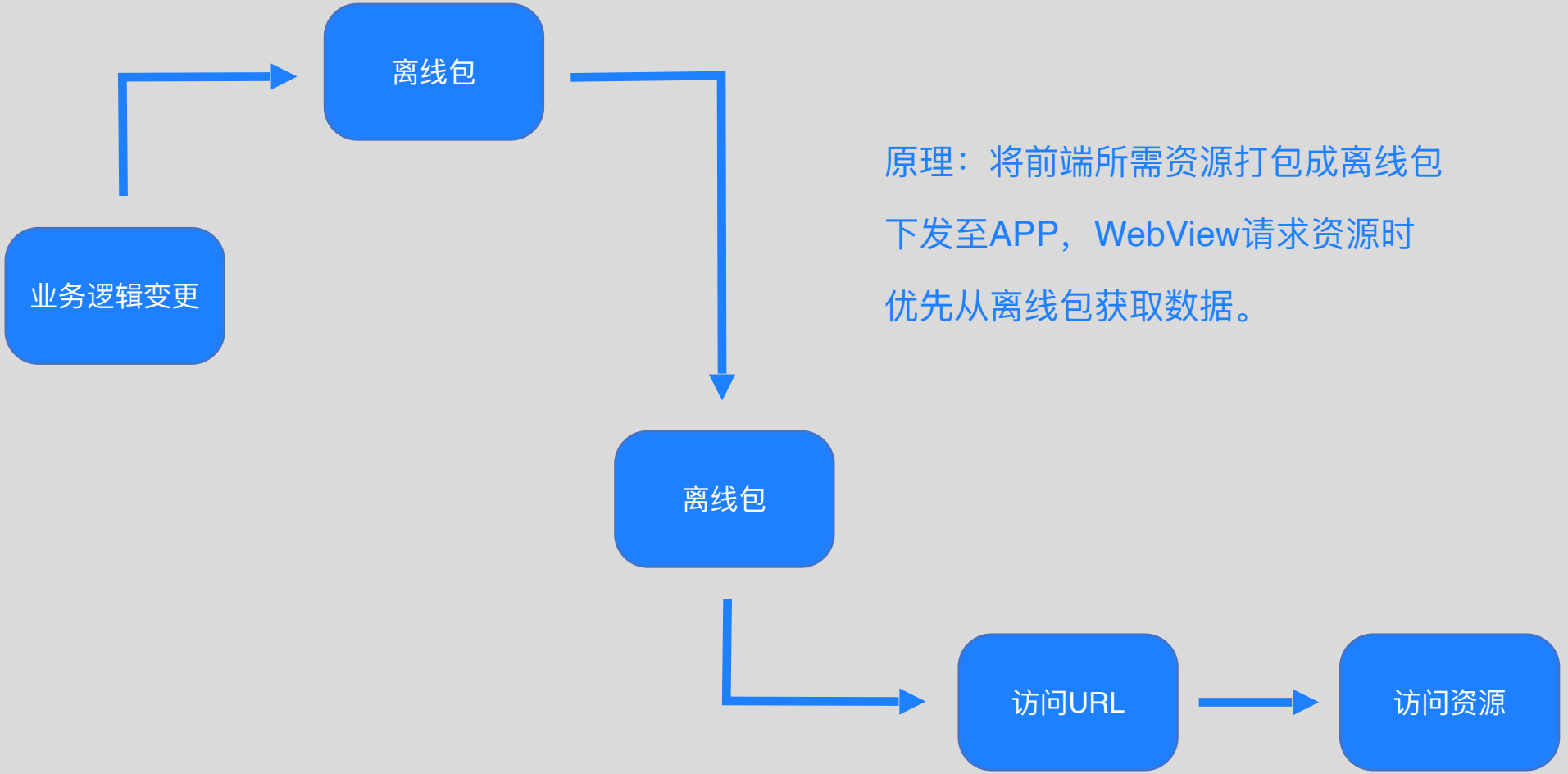
# 离线化

离线平台

开发者

离线SDK

WebView




原理：将前端所需资源打包成离线包

下发至APP，WebView请求资源时

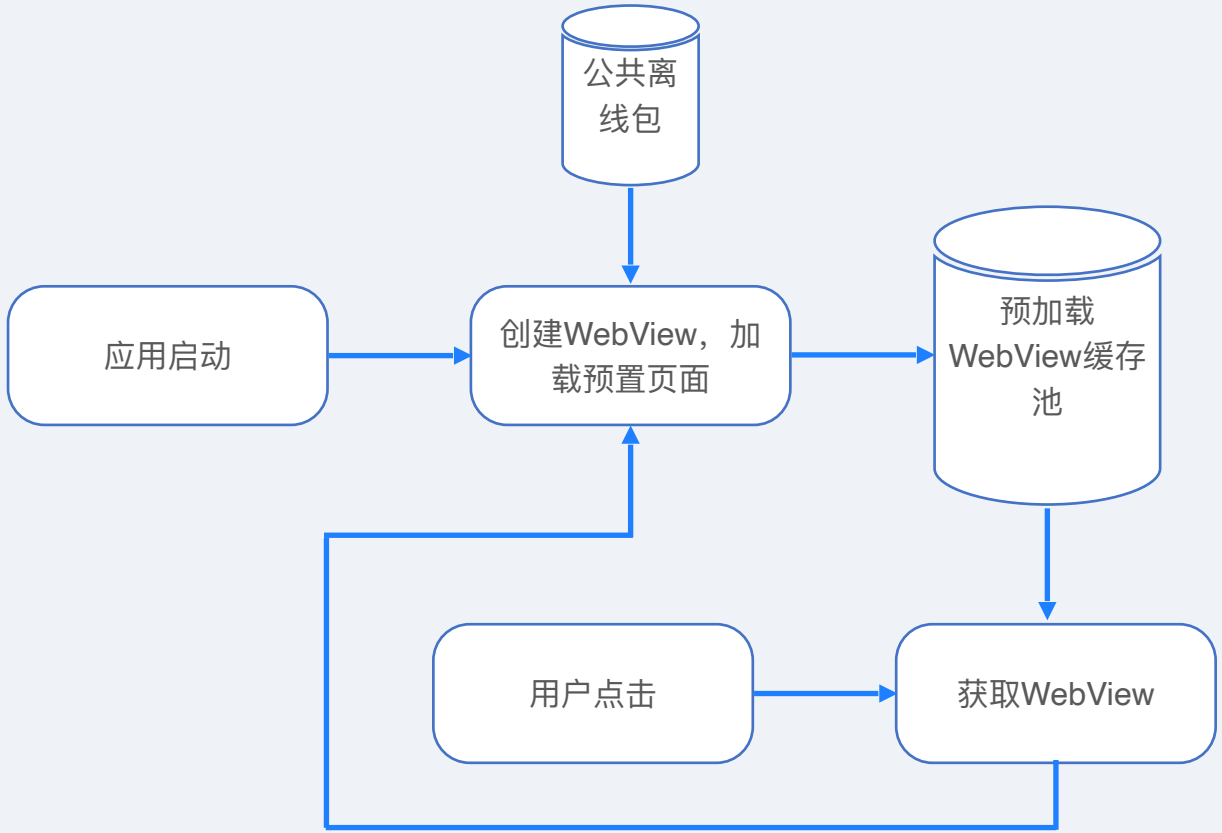
优先从离线包获取数据。

# 离线化


WebView初始化
建立握手SSL
解析HTML
构建渲染树
渲染页面
完全加载
Init
Network
Parsing
Style
Paint
Composite

# 预加载

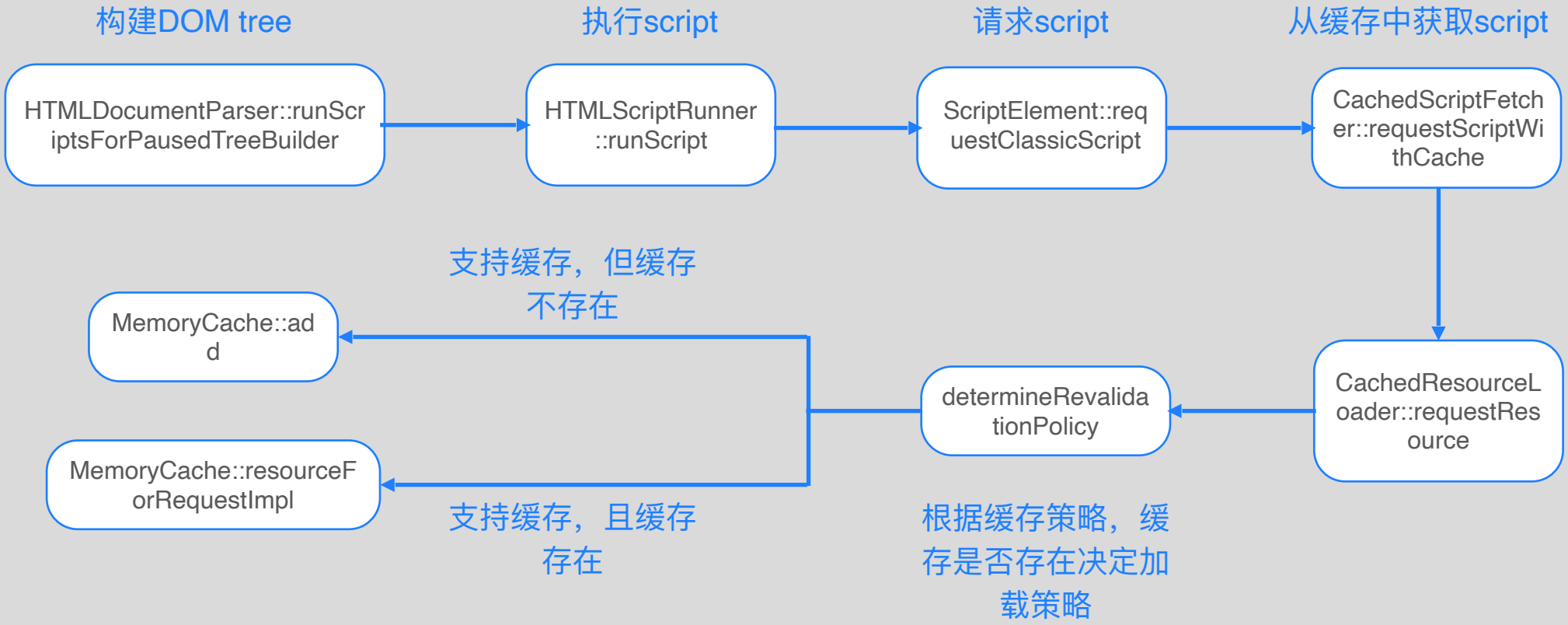
预创建WebView，并加载预置页面，该页面使用了公共资源。用户点击进入web页面时，直接使用预创建的WebView加载业务页面，从而达到复用资源，减少资源请求、解析耗时的目的。




3s后重新创建预加载WebView

# 预加载

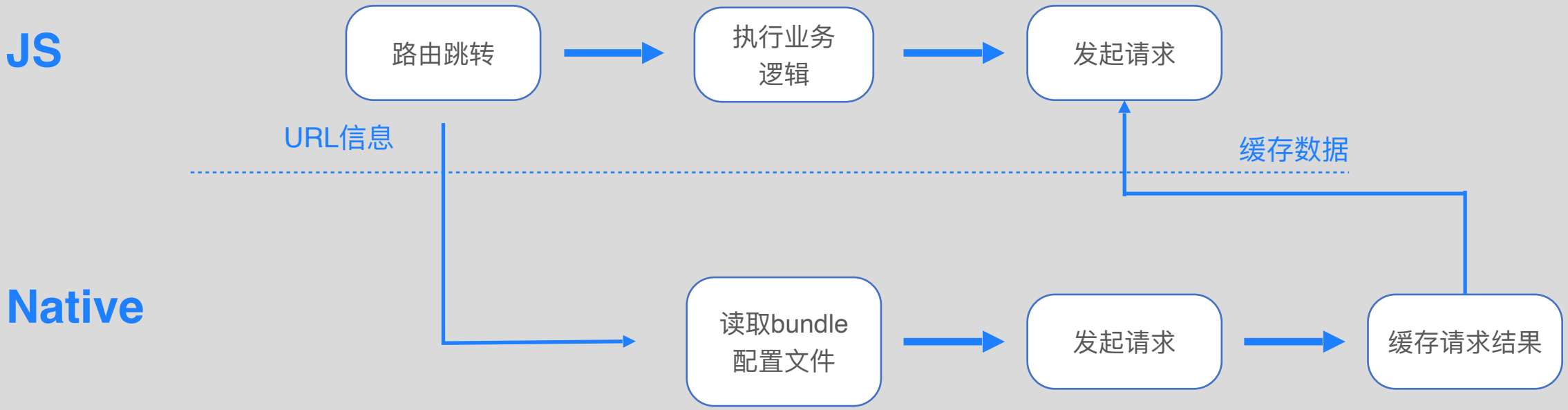
预加载为什么可以复用资源?



# 预加载


WebView初始化
建立握手SSL
解析HTML
构建渲染树
渲染页面
完全加载
Init
Network
Parsing
Style
Paint
Composite

# 请求前置

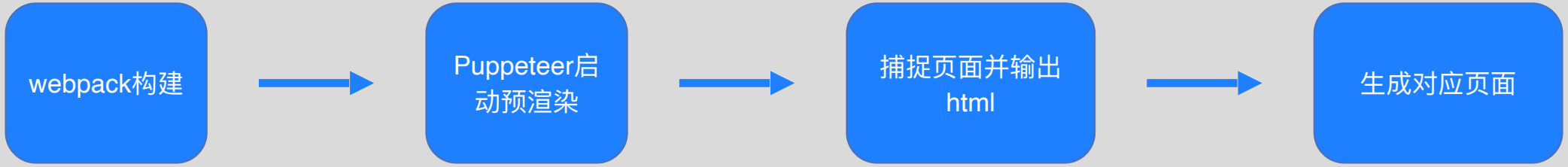

JS
路由跳转
执行业务逻辑
发起请求
URL信息
缓存数据
Native
读取bundle配置文件
发起请求
缓存请求结果

# 请求前置

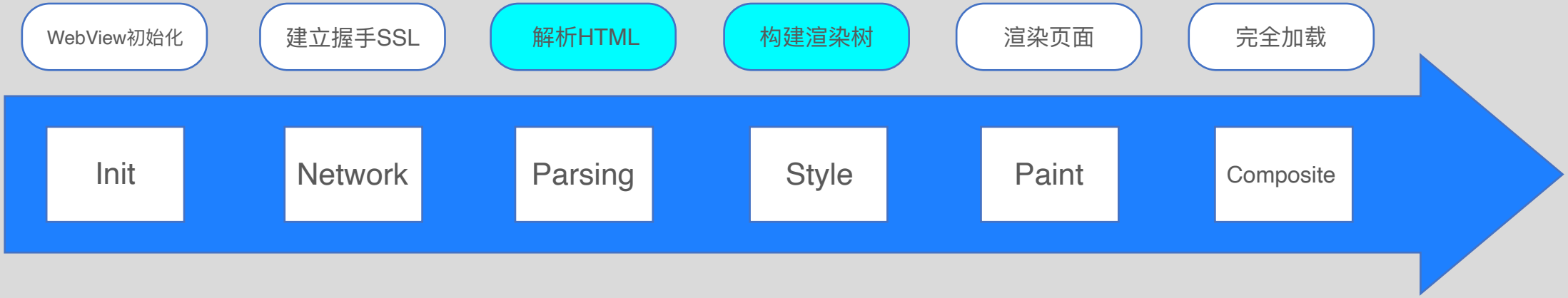

WebView初始化
建立握手SSL
解析HTML
构建渲染树
渲染页面
完全加载
Init
Network
Parsing
Style
Paint
Composite

# 预渲染

在webpack打包结束并生成文件后（after-emit hook），会启动一个server模拟网站的运行，用puppeteer访问指定的页面route，得到相应的html结构，并将结果输出到指定目录。

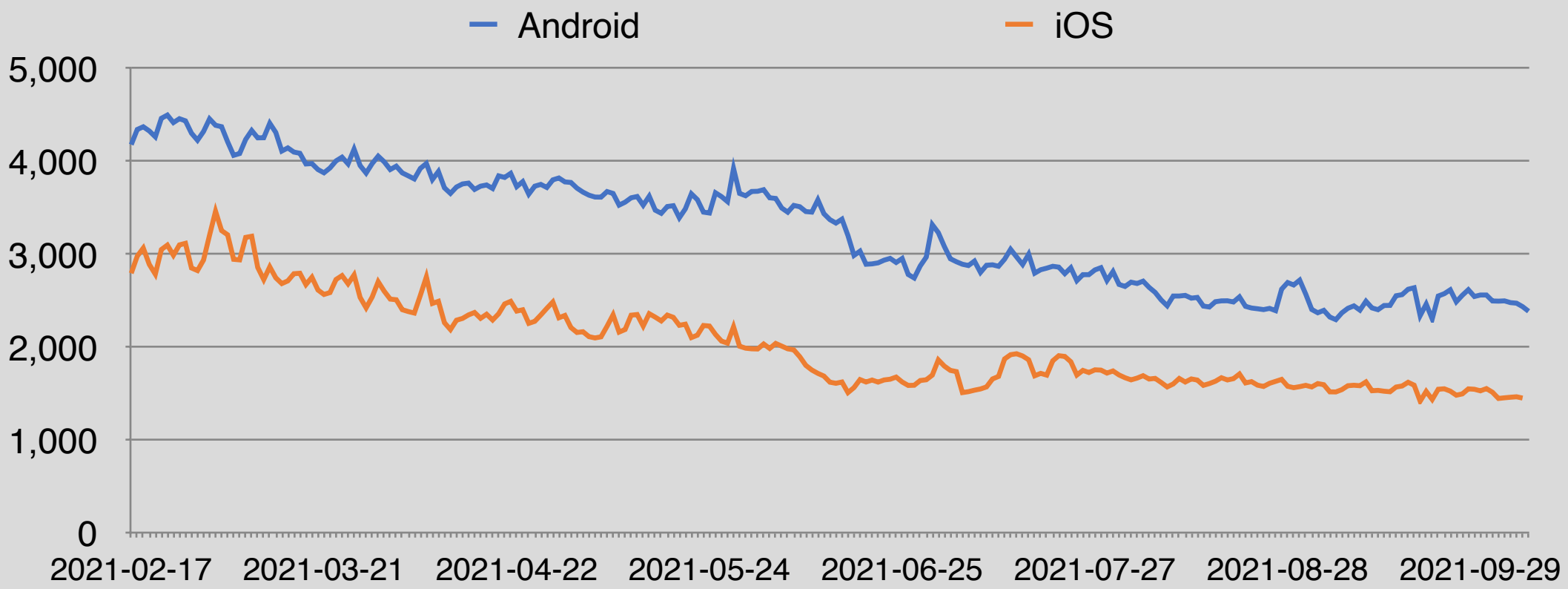


# 预渲染


WebView初始化
建立握手SSL
解析HTML
构建渲染树
渲染页面
完全加载
Init
Network
Parsing
Style
Paint
Composite

# 优化效果

TP90首屏时间



# 优化手段汇总

| 优化分类 | 方案 | 预期收益 | 适合场景 | 备注 | 业务改造成本 |
| --- | --- | --- | --- | --- | --- |
| 资源优化 | 离线化 | 600ms | 资源较多，网速较差 | 需保证较高离线包命中率 | 无 |
| 资源优化 | 预加载 | 300ms | 资源较多，网速较差 | 考虑内存影响 | 无 |
| 网络请求优化 | 请求前置 | 400ms | 有请求耗时时 | | 无 |
| 视图渲染优化 | 预渲染 | 200ms | 静态页面 | | 无 |

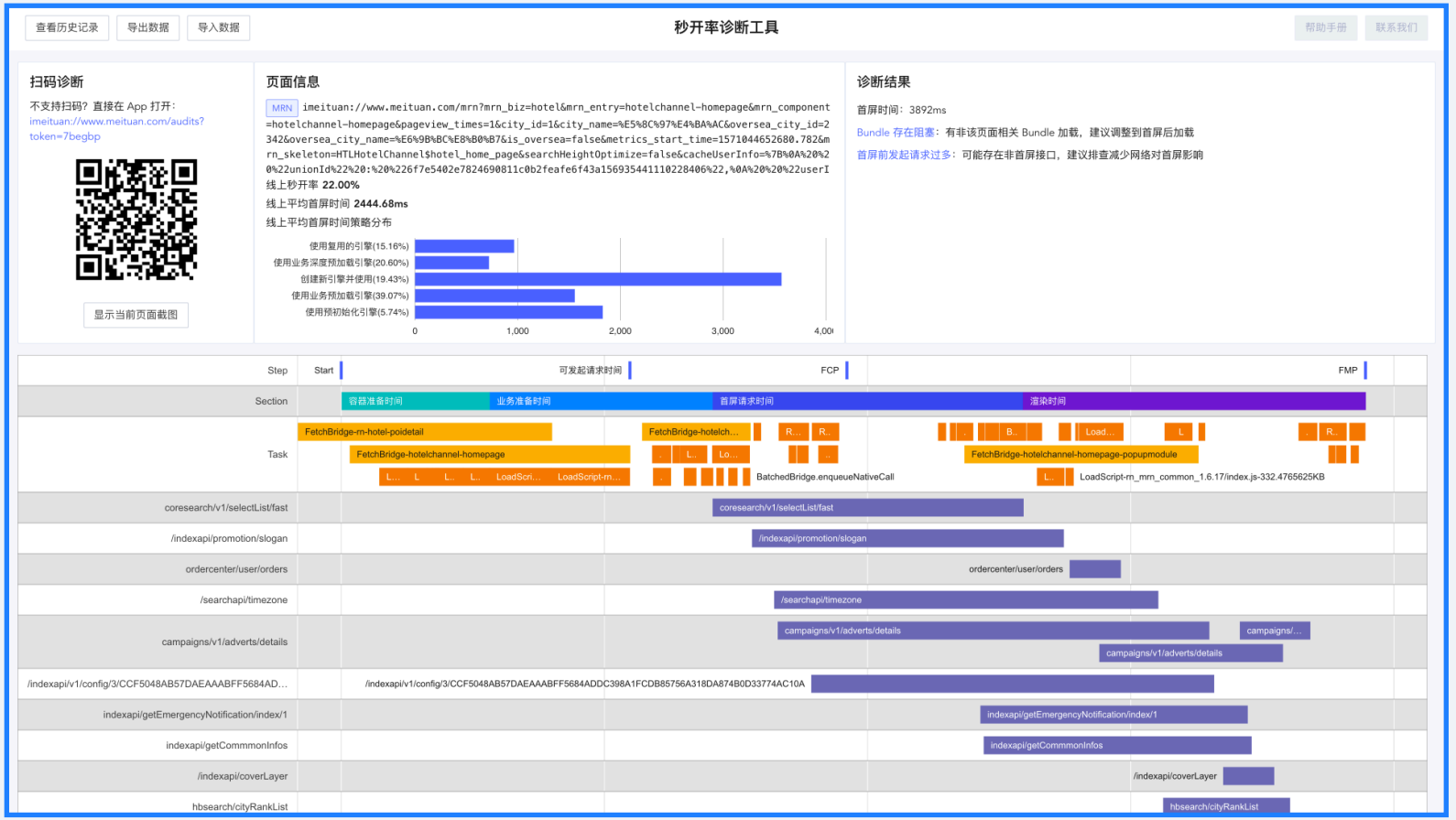
# 指标运维

指标排查

指标监控运营

# 指标排查

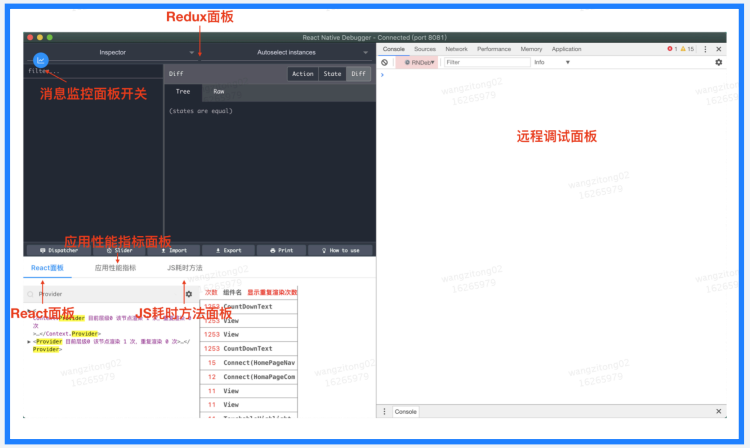
## 秒开诊断工具

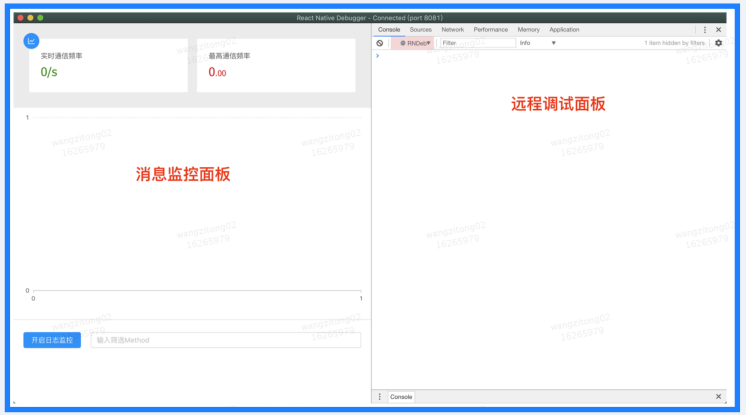



- 页面线上性能情况
- 阶段耗时
- 请求数据耗时
- 诊断结果

# 指标排查

## RN debugger



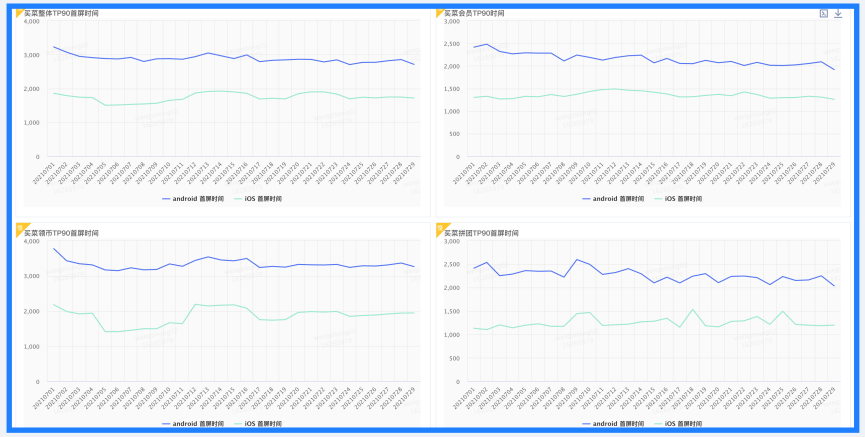




- 渲染
  - render次数
  - 冗余render
- 响应
  - 消息响应及时性
  - 绘图指令延迟
- 负载
  - CPU
  - 内存
  - 流量
- 卡顿
  - 帧率
  - 耗时方法
  - 耗时action

# 指标监控运营

## 日报

## 大盘




## 上线前性能测试卡控

- 集成阶段
- 集成期代码集成
- 打包 && 上线（提测）
- 开发自测(Release)
- 性能测试(蘑菇云)

# 未来展望

## Snapshot

As of the release of V8 v4.3 two months ago, embedders can utilize snapshotting to skip over the startup time incurred by such an initialization. The test case for this feature shows how this API works.

To create a snapshot, we can call `v8::V8::CreateSnapshotDataBlob` with the to-be-embedded script as a null-terminated C string. After creating a new context, this script is compiled and executed. In our example, we create two custom startup snapshots, each of which define functions on top of what JavaScript already has built in.

We can then use `v8::Isolate::CreateParams` to configure a newly-created isolate so that it initializes contexts from a custom startup snapshot. Contexts created in that isolate are exact copies of the one from which we took a snapshot. The functions defined in the snapshot are available without having to define them again.

- 保留RN引擎的堆对象，加速RN引擎创建

# 未来展望

## Sparkplug

With our current two-compiler model, we can't tier up to optimised code much faster; we can (and are) working on making the optimisation faster, but at some point you can only get faster by removing optimisation passes, which reduces peak performance. Even worse, we can't really start optimising earlier, because we won't have stable object shape feedback yet.

Enter Sparkplug: our new non-optimising JavaScript compiler we're releasing with V8 v9.1, which nestles between the Ignition interpreter and the TurboFan optimising compiler.


The new compiler pipeline

- 使用Sparkplug，加速JS编译

性能优化框架建设

将目前的性能优化能力插件化，根据页面类型自动启用

XDC 2021 XITU DEVELOPERS CONFERENCE

2021 XITU DEVELOPERS CONFERENCE//
稀土开发者大会
技 术 未 来 · 向 新 而 生

# THANKS