性能分析2021

- Q1目标和规划
  - 1、启动性能
    - 1）、目标：提升大盘FMP到1450ms，FMP提升80ms
    - 2）、项目：
      - （1）、落地的项目/现状/收益/目标
      - （2）、新启动的项目
  - 2、小程序运行时性能
    - 1）、目标
    - 2）、项目
      - （1）、落地中的项目
      - （3）、新启动项目
  - 3、NAview性能
    - 1）、目标：FMP优于webview 100ms+以上(现优于80ms)
    - 2）、项目
  - 4、动态扩展机制
    - 1）、目标
    - 2）项目
  - 5、数据报表建设
- 一，启动性能
  - 1、启动时序图（暂未更新）
  - 1、启动的主要环节和性能
    - 1）、资源加载和执行
      - （1）、逻辑层资源加载和执行
      - （2）、渲染层资源加载和执行

- 2）、数据获取：
  - （1）、initData获取会阻塞首次渲染
  - （2）、核心请求耗时仍是影响首次渲染性能的主要因素
- 3）、首屏渲染
  - （1）、核心渲染

- 2、多运行时下启动性能
  - 1）、广义swanjs下，core要关注更多运行时下的启动性能
    - （1）、web化性能
- 3、编译
  - 1）、编译与运行时存在耦合，可以改进的地方较多
  - 2）、编译产物健康度与性能
  - 3）、优化包编译灰度放量
  - 4、动态库、插件、动态库扩展机制
  - 1）、动态库对启动性能影响明显；
    - （1）、动态库的引入会显著增加包体积；
    - （2）、动态库会导致资源加载与解析的耗时增加;
    - （3）、动态库的加载机制完善
  - 2）、插件的不确定性；
  - 3）动态扩展机制
    - （1）、动态扩展机制，可以从技术机制上避免后续动态性带来的性能问题
    - （2）、动态扩展机制，给性能优化带来的新的方向
- 5、包体积与分包
  - 1）、分包推广因副作用受限，需从编译粒度角度来解决；
    - （1）、分包的副作用
    - （2）、分包与编译优化
- 6、开发者与优化推广
  - 1）、后续落地及开发中的很多优化手段，会需要开发者实施；
  - 2）、秒开的大面积落地，需要开发者做更多事情

- 二，分场景性能
  - 1、视频

- 2、二跳
- 3、benchmark
- 1、性能优化一览图
- 2、性能小组规划

Q1目标和规划

1、启动性能

1）、目标：提升大盘FMP到1450ms，FMP提升80ms

2）、项目：

（1）、落地的项目/现状/收益/目标

项目	现状	预计大盘FMP收益(全量或者TOP30接入后)	Q1期望目标	协同方现状	运营协同和开发者配合	风险	当前项目卡点

自定义组件拆分	B端针对小程序包暂未放量，其它方ready	50ms	全量落地（线上覆盖90%流量小程序应用此优化）	**编译**：工具已经全量发版 **B端**：TODO是否可以先两个包全量，针对3.270.x及以上swanjs应用此优化包 **C端**：审核包下发bug已经修复，ready **swanjs**：已全量 **客户端**：12.7版本（swanjs 3.270.x）修复自定义路由bug	开发者小程序有上线即可 （了解到目前top小程序发包较频繁，无需推动）	需尽快放量发现问题	B端
app.js拆分	待推动top小程序使用新编译包配合验证 待编译包全量上线	80ms（考虑到preload已经推全）	落地TOP30小程序	**swanjs**：已全量 **编译**：提供给三家使用（装馨家、健康、爱企查）本周或下周一提包 **B端**：TODO 1、实验阶段可以用两个包方案进行 2、全量需要三个包方案ready	开发者提包上线	B端三个包方案落地	B端
slave 预加载项目	待swanjs上线	30ms	全量落地	**swanjs**：3.270.x待放量 **客户端**：待swanjs放量50%时开ab开关，验证该功能	无需开发者配合	未经过线上验证，需放量发现问题	swanjs放量

预执行（onPrefetch）	待端实验全量	100ms	1、落地2家TOP小程序（风行视频、爱奇艺），验证FMP收益，达到可推广状态； 2、推动top3小程序接入	**swanjs**：已全量 **客户端**：放量中 5% **开发者接入**： 爱奇艺 开发中，风行视频 已提包依赖客户端放量	开发者改造成本较大，需专人配合开发者接入 需运营协同	开发者改造成本高，整体机制未经过线上一家小程序验证，风险较高	端放量（当前风行视频负收益，可能影响放量） 开发者接入
1. san 升级至使用 san.spa.dev.js 版本（有性能收益，需要考虑对工具影响）吕雷 2. 预加载阶段生成基础组件 aNode（启动性能有收益）吕雷 3. 去除基础组件的 getComponentType（对benchmark基础组件有收益）宁总 4. 优化自定义组件生成，主要是改 for 循环和扩展运算符 方哥 5. page 提前 aNode 解析 晓倩 6. 生成必渲染自定义组件的 aNode 云竹	待放量	线下测试100ms	上线并放量				

（2）、新启动的项目

项目	现状	预计大盘FMP收益(全量或者TOP30接入后)	Q1目标	需协同方	运营协同和开发者配合	风险	
slaveReady项目	swanjs已回滚 要重新开发评审	20ms	全量落地	swanjs：重新评审开发 客户端：按照新方案开发	无需开发者配合	看是否依赖跨平台做长期方案，还是用短期方案先上线	因客户端ios webview销毁问题重新评审设计
自定义组件css优化	方案中	30ms	完成整体开发上线，落地一家top小程序验证收益	编译 B端	开发者重新上线小程序	1、需要 cover 的 case 比较多，且不易在灰度过程中发现问题（样式问题） 2、B端，编译包放量机制；	
逻辑层多小程序代码预充优化	方案中	30ms	开发、测试和上线。全量			多小程序代码和模块隔离风险	
~~端能力调用缓存~~	~~调研中~~	~~20ms~~	~~方案、开发、测试和上线。全量~~			耗时的能力涉及屏幕等信息，客户端已经做过一轮缓存，前端缓存再横竖屏切换过程中可能出问题 结论：加到性能白屏书里，作为性能优化建议让开发者对同步能力调用进行缓存，同步百科开发者进行耗时能力缓存	
APack二期	待开发	30ms	1、基础组件apack（anode提前生成项目上线后收益不大） 2、固化swanjs，去掉实验； 3、编译去掉template，同编译优化包下发；	编译 B 端	开发者重新上线小程序【优化包】	1、一期Bug修复，12.7放量，apack放量； 2、B端，编译包放量机制；	

appjs拆分后二跳资源提前加载	方案中		提升二跳页的fmp				
benchmark指标review 更真实反应swanjs性能			对benmark现有指标进行review和调整 重新跑基线				

2、小程序运行时性能

1）、目标

（1）、Benchmark对比竞品Gap保持40%+

整体	百度-现状(a)	目标	微信(b)	GAP-现状 (a-b)/b	目标
core	32996	32500	52789	-37.49%	-40%
component	44514	44000	72812	-38.86%	-40%
api	1643	1600	2687	-38.84%	-40%
总	79153	78000	128288	-38.30%	-40%

（2）、分场景性能

- 视频启播时间
- 二跳

2）、项目

（1）、落地中的项目

场景	项目	现状	预计大盘收益(全量或者TOP30接入后)	Q1目标

视频	通过onPrefetch和预取端能力提升视频起播时间		起播时间：	
二跳	slave 预加载项目【启动中包含】	放量中 5%	二跳FMP：	

（3）、新启动项目

场景	项目	现状	预计大盘收益(全量或者TOP30接入后)	Q1目标
二跳	逻辑层二跳资源预先加载	方案中	二跳FMP：	
二跳	基于initData二跳预先渲染	方案中	二跳FMP：	
Benchmark	自定义组件运行时优化【启动中包含】	方案中	Benchmark：	

3、NAview性能

1）、目标：FMP优于webview 100ms+以上(现优于80ms)

2）、项目

项目	现状	预估收益	Q1目标
updateview优化（减少调用）	未开始	15ms	方案、开发、测试、上线
节点创建过程优化	未开始	20ms	方案、开发、测试、上线
batchRender优化（减少调用）	未开始		
事件机制优化（减少端调用）	未开始		

4、动态扩展机制

1）、目标

1、完善整体方案，多端开始；

2、落地；

2）项目

项目	现状	预估收益	Q1目标	成本
动态扩展机制方案方案评审	1、四端方案，初步评审完毕； 2、产品侧没有相关业务诉求； 3、TP场景收益不明确；		1、确认最终扩展机制的规范和方案；	五端改造
TP-度小店小程序模板依赖下发复用	1、方案沟通中； 2、目前预下载（preload）率为55%；	待TP PM同步		度小店预估接入成本为 15天，认为改造成本太大；
动态库-互动组件转依赖下发				

5、数据报表建设

showx性能报表规划

报表项	简介	规划	
细分时长	对FMP进行细分，方便观察某项优化的效果。	1、【推动+开发】在Android下，资源加载相关的时长可以进一步细分。内核已提供api获取某个文件的加载时长的细分指标，后续可以上报到这个报表上。（具体细节可和王宁同步） 2、完善FMP指标的真实度	

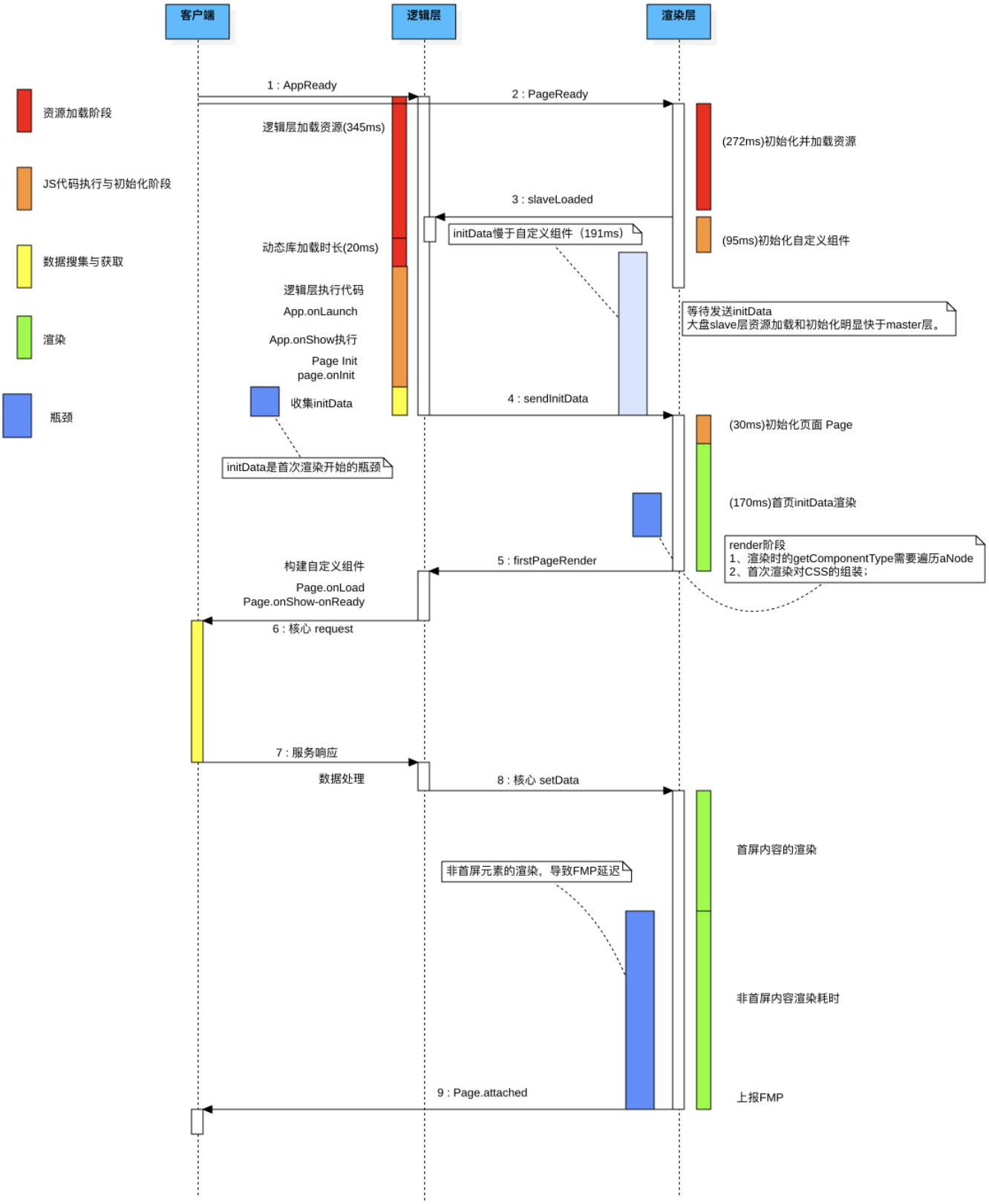
多跳转场数据	展现路由跳转场景的性能数据	1、【推动】数据量太少，需推动数据侧RD将数据源从ubc切换为ceres（12.7铺量后数据侧可以做切换） 2、【推动】不能筛选swanjs，需推动客户端上报数据时带上swanjs版本（12.7已加）	
API时长	在框架层对所有API的调起→完成的时间进行线上观察	针对启动阶段的API调用进行数据统计 正对长耗时API进行监控	

一，启动性能

1、启动时序图（暂未更新）

FMP大盘：1600ms

FMP目标：1000ms


客户端
逻辑层
渲染层
资源加载阶段
JS代码执行与初始化阶段
数据搜集与获取
渲染
瓶颈
1 : AppReady
2 : PageReady
逻辑层加载资源(345ms)
(272ms)初始化并加载资源
3 : slaveLoaded
initData慢于自定义组件（191ms）
(95ms)初始化自定义组件
动态库加载时长(20ms)
逻辑层执行代码
App.onLaunch
App.onShow执行
Page Init
page.onInit
等待发送initData
大盘slave层资源加载和初始化明显快于master层。
收集initData
4 : sendInitData
(30ms)初始化页面 Page
initData是首次渲染开始的瓶颈
(170ms)首页initData渲染
render阶段
1、渲染时的getComponentType需要遍历aNode
2、首次渲染对CSS的组装；
构建自定义组件
5 : firstPageRender
Page.onLoad
Page.onShow-onReady
6 : 核心 request
7 : 服务响应
数据处理
8 : 核心 setData
首屏内容的渲染
非首屏元素的渲染，导致FMP延迟
非首屏内容渲染耗时
9 : Page.attached
上报FMP

启动性能时序分析图

1、启动的主要环节和性能

1）、资源加载和执行

（1）、逻辑层资源加载和执行

逻辑层资源加载、解析、执行、初始化等。主要是通过_naSwan.include将逻辑层所需资源加载到js执行环境中（v8/jsc），并进行解析；

- 现状
  - 框架AppReady下发到App.onLaunch开始执行，主要为逻辑层代码、动态库资源加载、解析时间，合计270ms 。
  - 大盘逻辑层代码加载、解析耗时为 220ms
  - 逻辑层代码加载完毕到appLaunch开始执行，为50ms;
- 近期变化（最近三个月）
  - 逻辑层资源加载和执行，从410ms降低到270ms，主要是由于逻辑层代码preload带来的提升；
  - App.onLaunch结束到Page.onInit开始执行，从137ms增加到了168ms；？
- 进行中的项目
  - appJS拆分
  - 热启动&二跳预先加载
- 后续优化方向和思路
  - 多小程序代码预充
    - 背景：目前preload在feed场景下的命中率为80%，搜索场景下的命中率为60，可以通过**多小程序代码预充**进一步提升命中率
    - 目标：目前初步估计多小程序代码预充，可以使
      - feed命中率提升到90%
      - 搜索命中率提升到80%
  - JS代码异步加载
    - 背景：在目前的preload命中率下，仍有大量的场景需要在启动时，加载逻辑层代码。但是有些场景appJs拆分，覆盖不了，比如动态库，非首屏的自定义组件

        - 目标：实现非首屏的资源，可以异步加载，例如活动组件-底bar，在很多场景下，不要求其立即展示，可以在FMP展示之后再加载互动组件；
    - 端能力调用优化
        - 背景：端能力调用会通客户端通信，多次调用时也会有一定的耗时
        - 目标：通过在框架层实现一套缓存机制，对一些端能力调用进行缓存

- 总结：
    - 逻辑层资源加载，在preload && appJS拆分后，加载耗时大幅减少，优化空间会进一步减少

（2）、渲染层资源加载和执行

渲染层资源加载、解析、执行、初始化等，主要是通过DOM操作将资源挂在到webview容器上，包含JavaScript，CSS（目前自定义组件的CSS以JS的形式存在）。

- 现状
    - 模版和CSS加载时长：85ms
    - SJS加载时长：37ms
    - 自定义组件加载时长：210ms
    - 自定义组件初始化：100ms
- 进行中的项目
    - 渲染层预先加载
- 近期变化
    - 各项数据没有明显变化，自定义组件没有明显变化
- 后续优化方向和思路
    - 背景：渲染层目前页面和自定义组件代码会全部转换为apack，及绝大部分资源都可以使用JSON来表达；
    - 目标：通过将渲染层的资源转换为JSON的数据格式，利用对内核处理静态类型（JSON）比处理动态脚本（JS）快的特性，提升加载和解析效率

2）、数据获取：

（1）、initData获取会阻塞首次渲染

收集页面、自定义组件渲染所需的初始化数据，initData获取会阻塞首次渲染

- 近期变化（最近三个月）
  - 等待initData发送时长从25ms增加到了50ms；
  - initData慢于自定义组件时长，从210ms降到了130ms；

（2）、核心请求耗时仍是影响首次渲染性能的主要因素

- 现状
  - 99%以上的小程序，FMP的渲染依赖核心数据的获取。
  - 基于initData首次渲染触发的FMP占比不到1%；（没有准确数据，目前是根据众测感受得出）
- 进行中的项目
  - onPrefetch生命周期
    - 背景 ：支持onPrefetch生命周期，使开发者可以在该生命周期中发送request请求，提前获取核心数据；
    - 目标：使核心request提前200ms左右
    - 风险点：该生命周期落地问题较大

3）、首屏渲染

降低渲染阶段的消耗，将与渲染无关的逻辑尽可能前置；渲染阶段的优化，是后面优化的重点，重点为按需渲染和组件优化；

（1）、核心渲染

- 现状
  - 大盘前端渲染总耗时：800ms；
  - 核心渲染内容超过一屏，是导致渲染时长较高的主要原因；
  - 自定义组件过度使用会导致性能退化
  - 通过对核心渲染进行优化，使FMP数据显著提升的案例
    - 宝宝知道在业务层对核心页面进行分屏渲染，使FMP性能提升350ms；
    - 知道通过对问答页进行首屏渲染优化，FMP 性能提升了500ms;

- 进行中的项目
  - 推动业务方进行分屏按需渲染
  - 通过setData实现按需渲染
- 后续项目及思路
  - 统计线上各个小程序，FMP触发时渲染内容信息，分析所渲染内容和首屏内容差异性
  - 可推动内核实现content-visibility API，从框架层面实现按需或者使按需的实现成本降低；【需联合内核侧，收益非常大】
  - 自定义组件的CSS的处理逻辑【难度大，需联合编译侧，收益明显。目前微姐在跟进】
    - 自定义组件在初始化阶段从js数组转为字符串，在渲染阶段需动态将css标签插入dom；
    - 可考虑在编译阶段，将自定义组件的css直接生成css文件，作用域等问题可考虑通过scoped等形式来处理
  - 非首屏元素的异步渲染（影子组件），可配合异步加载
  - 预渲染，在prefetch等基础上，提前进行页面的渲染

2、多运行时下启动性能

1）、广义swanjs下，core要关注更多运行时下的启动性能

（1）、web化性能

- 资源加载及资源拆分粒度对web化的FCP的影响很大；
  - web化无预加载机制；
  - web化下资源下载有网络耗时，对资源文件大小很敏感；

（2）、NAview性能

- NAview的落地对性能要求极高
- NAview的性能，主要集中在渲染阶段

3、编译

性能&&编译项目

编译和运行时需要完全解耦。

1）、编译与运行时存在耦合，可以改进的地方较多

（1）、在构建中对模块进行粒度拆分优化，需要修改加载逻辑；

- 逻辑层app.js拆分基本解耦，动态库、插件还可进一步考虑；
- 渲染层，自定义组件拆分后也已经解耦，渲染层还需y要考虑一个更完整的方案，来包含渲染层的所有资源；

（2）、运行时存在编译注入的逻辑；

- 动态库的工厂函数

（3）、运行时有些处理可移到编译层去处理

- slave层自定义组件的CSS组装过程
    1. 目前自定义组件的css是用js数组来表示的，而不是css文件。会导致js加载解析时间增加。；
    2. 在初始化阶段，需要动态将样式从数组转为字符串；
    3. 在渲染阶段，动态将样式插入dom的操作，会导致styleObject的生成比较晚，进而影响ayout tree的生成；

2）、编译产物健康度与性能

（1）、编译产物缺少二次检查机制，导致代码包产物存在优化点

- 大体积的图片资源；
- 无效的图片资源，如CSS从未引用过的图片，可以给出提示；
- 无效的文件，如package-lock.json；

（2）、开发者无法了解到编译产物状态，使开发者取少一些优化信息

- 提供类似于bundle analysis的分析报告，使开发者对包产物有一定的了解；
- 按照文件分类，目录给出产物体积大小信息，使开发者对各类资源管理、各个代码目录的大小有一定认知；
- 根据编译产物的现状，给出优化建议，比如
    - app.js文件过大；
    - 分包设置是否合理：基于分包配置和核心页面；

3）、优化包编译灰度放量

（1）、目前因优化包灰度放量机制缺陷，阻碍了编译优化项目的落地：

- app.js拆分未落地；
- 自定义组件拆分【因B端问题，已经全部终止优化包放量】；

（2）、编译侧流程，导致部分优化编译项目无法落地

- template编译优化

4、动态库、插件、动态库扩展机制

1）、动态库对启动性能影响明显；

【模块】动态库

（1）、动态库的引入会显著增加包体积；

- 互动组件产物大小超过100k;
- 动态库资源的跨小程序共享，可以避免每个小程序的包都包含相同的动态库文件【联合下载】；

（2）、动态库会导致资源加载与解析的耗时增加;

- 按需加载（已实现）能避免特定页面加载该页面不需要的动态库；
- 动态或者异步加载机制，可以使开发者动态控制动态库的加载时机；

（3）、动态库的加载机制完善

- 联合下载和动态加载的配合，可以大幅降低动态库对启动加载带来的影响；

2）、插件的不确定性；

- 插件的产品形态还比较模糊；

3）动态扩展机制

动态扩展机制五端汇总

（1）、动态扩展机制，可以从技术机制上避免后续动态性带来的性能问题

- 动态库会导致包体积增加
- 动态库会导致页面资源加载耗时增加，且无法异步

（2）、动态扩展机制，给性能优化带来的新的方向

- 优于现有分包方案的拆分构建策略
- 可以实现跨小程序资源的缓存贡献
- 可以实现小程序包的增量下发
- 可以提供异步加载能力

5、包体积与分包

包体积优化，可以减少包下载时间，且分包后的代码的加载和解析时间将会减少。

1）、分包推广因副作用受限，需从编译粒度角度来解决；

（1）、分包的副作用

- 普通分包、独立分包需要分场景；
- 较多开发者反馈分包后性能未提升，甚至退化；

（2）、分包与编译优化

- 分包解决的是按需加载问题；
- 分包的依赖拆分应同编译构建协同；

6、开发者与优化推广

1）、后续落地及开发中的很多优化手段，会需要开发者实施；

- 预取生命周期，同onInit一样依赖开发者接入；

2）、秒开的大面积落地，需要开发者做更多事情

- 从实践来看，如果将现有手段（包含落地中）全部运用，已经可以很接近秒开启动了，但这需要开发者深入理解性能优化的原理、手段等知识；

二，分场景性能

1、视频

主要疼点为视频起播时间

主要方案为，利用onPrefetch生命周期提前获取视频地址

2、二跳

二跳时的逻辑包含：

- 如果为分包，且本地没有包资源，需要下载包资源；
- 需要加载逻辑层代码（app.js拆分后），渲染层代码（自定义组件和CSS）；
- 搜集并发送initData，进行首次渲染

现状：

- 目前大盘二跳FMP数据报表建设未完善

进行中的项目：

- 渲染层二跳资源预先加载，提前下载分包代码和进行渲染层资源加载
- 逻辑层二跳资源预先加载

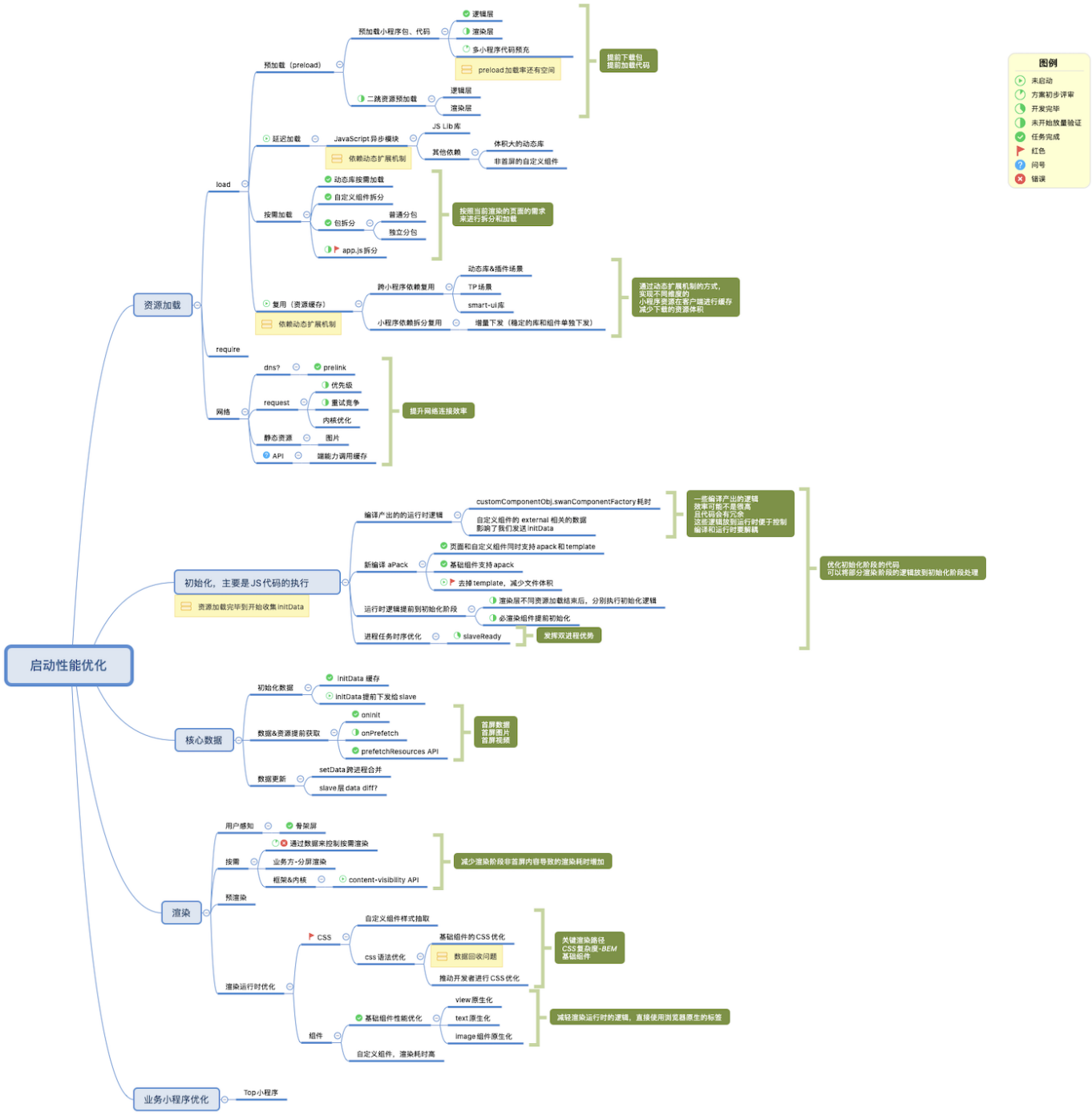
后续优化方向

- 基于initData的，二跳预先渲染

3、benchmark

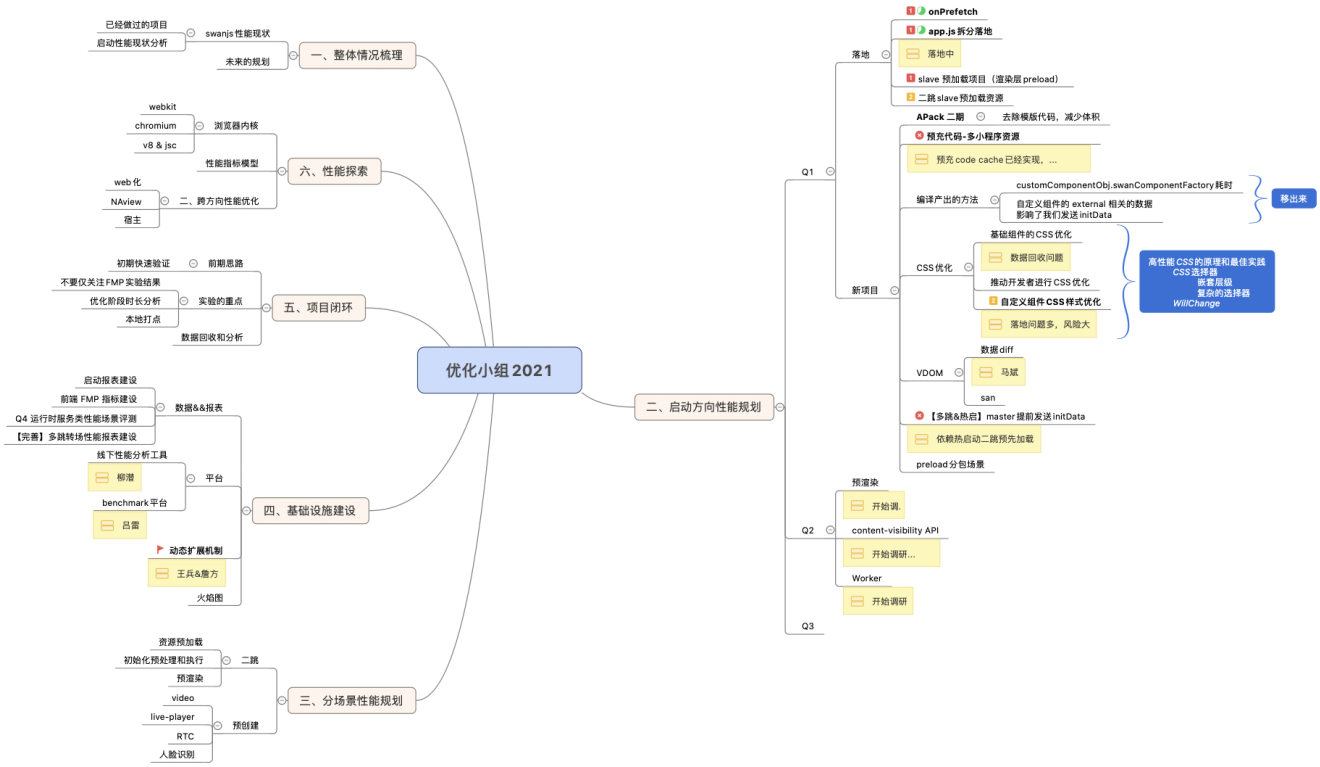
附件：

1、性能优化一览图


图例
未启动
方案初步评审
开发完毕
未开始放量验证
任务完成
红色
问号
错误
启动性能优化
资源加载
load
预加载（preload）
预加载小程序包、代码
逻辑层
渲染层
多小程序代码预充
preload加载率还有空间
提前下载包
提前加载代码
二跳资源预加载
逻辑层
渲染层
延迟加载
JavaScript异步模块
JS Lib库
其他依赖
体积大的动态库
非首屏的自定义组件
依赖动态扩展机制
按需加载
动态库按需加载
自定义组件拆分
包拆分
普通分包
独立分包
app.js拆分
按照当前渲染的页面的需求
来进行拆分和加载
复用（资源缓存）
依赖动态扩展机制
跨小程序依赖复用
动态库&插件场景
TP场景
smart-ui库
小程序依赖拆分复用
增量下发（稳定的库和组件单独下发）
通过动态扩展机制的方式，
实现不同维度的
小程序资源在客户端进行缓存
减少下载的资源体积
require
网络
dns?
prelink
request
优先级
重试竞争
内核优化
静态资源
图片
API
端能力调用缓存
提升网络连接效率
初始化，主要是JS代码的执行
资源加载完毕到开始收集initData
编译产出的运行时逻辑
customComponentObj.swanComponentFactory耗时
自定义组件的 external 相关的数据
影响了我们发送 initData
一些编译产出的逻辑
效率可能不是很高
且代码会有冗余
这些逻辑放到运行时便于控制
编译和运行时要解耦
新编译 aPack
页面和自定义组件同时支持 apack 和 template
基础组件支持 apack
去掉template，减少文件体积
运行时逻辑提前到初始化阶段
渲染层不同资源加载结束后，分别执行初始化逻辑
必渲染组件提前初始化
进程任务时序优化
slaveReady
发挥双进程优势
优化初始化阶段的代码
可以将部分渲染阶段的逻辑放到初始化阶段处理
核心数据
初始化数据
initData 缓存
initData 提前下发给 slave
数据&资源提前获取
oninit
onPrefetch
prefetchResources API
首屏数据
首屏图片
首屏视频
数据更新
setData 跨进程合并
slave 层 data diff?
渲染
用户感知
骨架屏
按需
通过数据来控制按需渲染
业务方-分屏渲染
框架&内核
content-visibility API
减少渲染阶段非首屏内容导致的渲染耗时增加
预渲染
渲染运行时优化
CSS
自定义组件样式抽取
css 语法优化
基础组件的CSS优化
数据回收问题
推动开发者进行CSS优化
关键渲染路径
CSS复杂度-BEM
基础组件
组件
基础组件性能优化
view 原生化
text 原生化
image 组件原生化
自定义组件，渲染耗时高
减轻渲染运行时的逻辑，直接使用浏览器原生的标签
业务小程序优化
Top小程序

2、性能小组规划


□□□□2021.xmind


优化小组 2021
一、整体情况梳理
已经做过的项目
启动性能现状分析
swanjs 性能现状
未来的规划
六、性能探索
浏览器内核
webkit
chromium
v8 & jsc
性能指标模型
二、跨方向性能优化
web 化
NAview
宿主
五、项目闭环
前期思路
初期快速验证
实验的重点
不要仅关注 FMP 实验结果
优化阶段时长分析
本地打点
数据回收和分析
四、基础设施建设
数据&&报表
启动报表建设
前端 FMP 指标建设
Q4 运行时服务类性能场景评测
【完善】多跳转场性能报表建设
平台
线下性能分析工具
柳潜
benchmark 平台
吕雷
动态扩展机制
王兵&詹方
火焰图
三、分场景性能规划
二跳
资源预加载
初始化预处理和执行
预渲染
预创建
video
live-player
RTC
人脸识别
二、启动方向性能规划
Q1
落地
onPrefetch
app.js 拆分落地
落地中
slave 预加载项目（渲染层 preload）
二跳 slave 预加载资源
新项目
APack 二期
去除模版代码，减少体积
预充代码-多小程序资源
预充 code cache 已经实现，...
编译产出的方法
customComponentObj.swanComponentFactory 耗时
自定义组件的 external 相关的数据
影响了我们发送 initData
移出来
CSS 优化
基础组件的 CSS 优化
数据回收问题
推动开发者进行 CSS 优化
自定义组件 CSS 样式优化
落地问题多，风险大
高性能 CSS 的原理和最佳实践
CSS 选择器
嵌套层级
复杂的选择器
WillChange
VDOM
数据 diff
马斌
san
【多跳&热启】master 提前发送 initData
依赖热启动二跳预先加载
preload 分包场景
Q2
预渲染
开始调.
content-visibility API
开始调研...
Worker
开始调研
Q3