WEB CONF
<腾 讯 前 端 技 术 大 会/>

极致SSR

构建高效率高性能Web同构应用

段隆贤

个人介绍

段隆贤 lucien

腾讯视频Web高级工程师

腾讯视频从2016年开始整站接入Node.js服务， 2017年开始自研Vue语法编译器并在线上作为直出，同构的模板引擎广泛使用。

腾讯视频在Node.js SSR(服务器端渲染: Server-Side Rendering)， Serverless， 分块传输， 同构等技术有多个成熟的案例。

目录

为什么需要SSR

1. 更好的搜索引擎优化(SEO: Search Engine Optimization)

2. 更快的首次有效绘制(FMP: First Meaningful Paint)

什么是极致的SSR

1. 更短的首次有效绘制时间(FMP: First Meaningful Paint)
2. 更短的可交互时间 (TTI : Time to Interactive)
3. 有更好的交互体验，给用户更好的动画过渡和预期

并且： 开发效率高， 开发简单

SSR的现状

对比预期， 现状是什么?

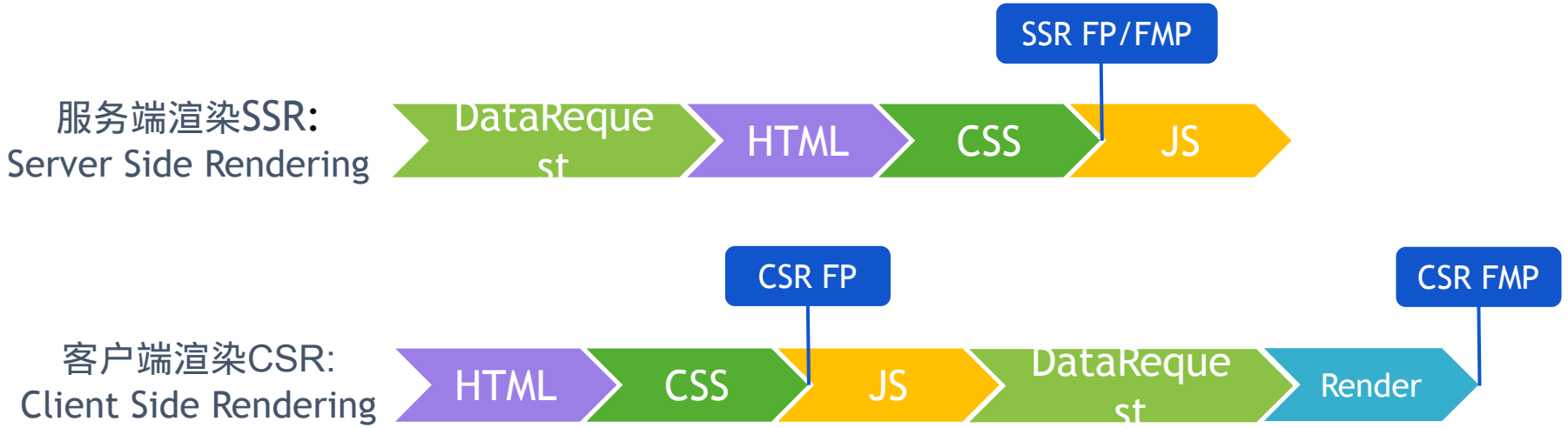
1. 一般SSR, 首屏(FMP)依赖页面所有接口， 首屏不一定快，同时分块传输有额外的工作量

2. 开发效率低: 传统的SSR, 需要操作DOM, 开发效率低， 难维护，同构页面可响应时间（TTI）长

3. SSR页面切换无法渐进式加载， 页面切换时不能定义过渡动画

SSR现状: 首次绘制慢

一般SSR首次绘制(FP: First Paint)慢, 不能渐进式加载



HTTP1.1 分块传输可以完美的解决这个问题

分块传输

分块传输是什么

RFC2616: The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an OPTIONAL trailer containing entity-header fields.

说人话： 不需要等待页面所有的接口返回， 页面头部接口响应即可响应页面， FP(首字渲染 first paint)和FCP(首次内容渲染: first contentful paint)更快

在HTTP协议的展现形式:

HTTP1.1: Transfer-Encoding: chunked

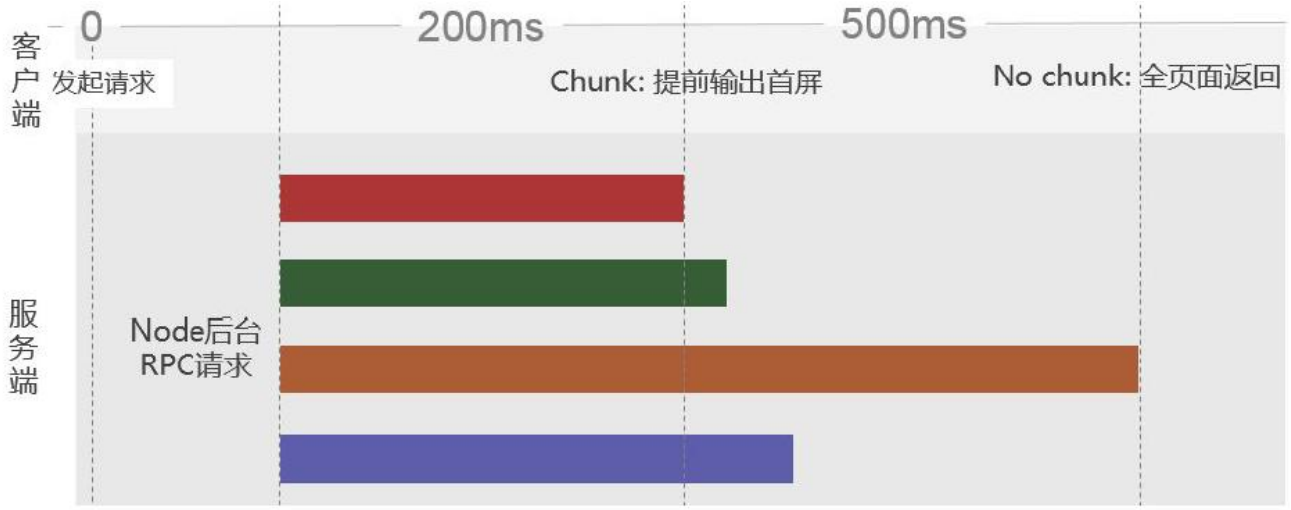
HTTP2: 数据帧

分块传输

SSR为什么需要分块传输

利用分块返回可以更快加载页面首屏

- 不分块返回：需等待所有后台数据ready再发给浏览器
- 分块返回：第一时间返回页面头部，提前加载css，再分块渲染页面



分块传输

SSR为什么需要分块传输

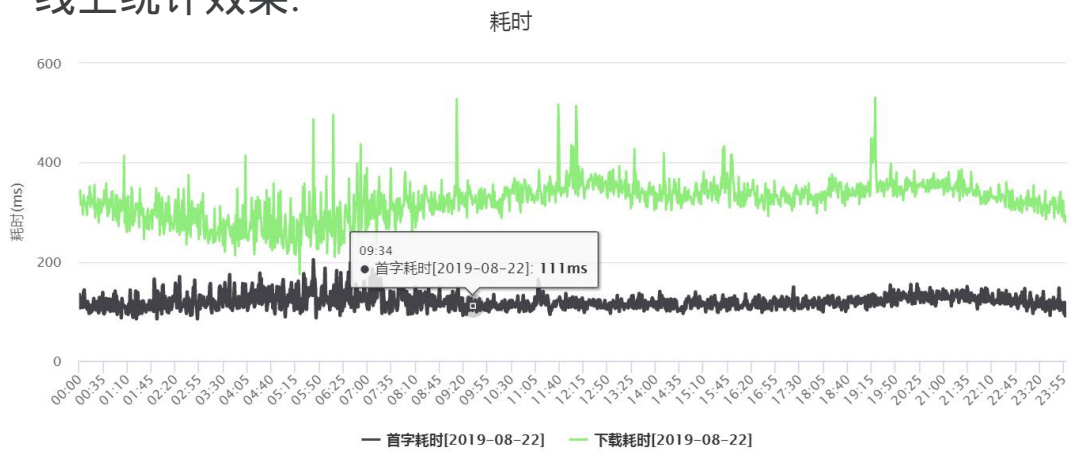
腾讯视频首页的分块加载

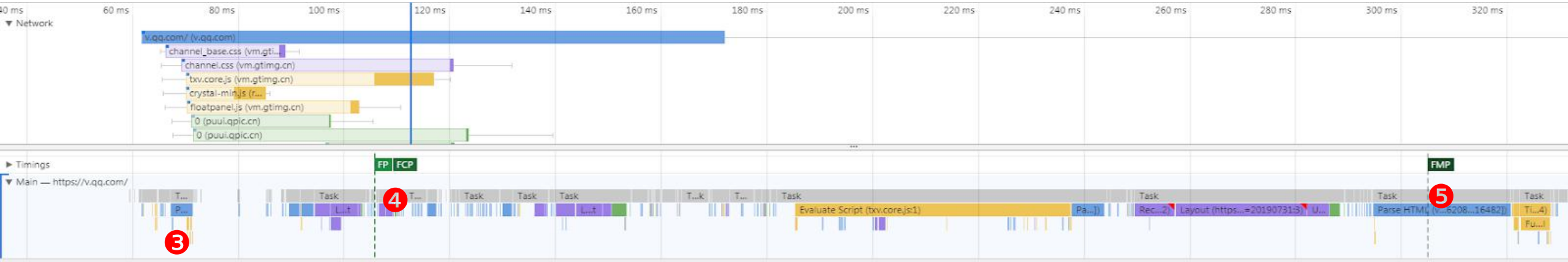
高度11007px（1080P下10屏）

关键指标:

1. 首字耗时47ms
2. 下载耗时252ms
3. 70ms时开始解析html，并请求css
4. 浏览器首次渲染110ms
5. 首屏渲染310ms

线上统计效果:





分块传输: 自动化

当前分块方案的问题及优化方案

问题: 分块增加开发成本

1. 人力手工拆分模板
2. 找到模板间的异步数据依赖并维护数据间的依赖链
3. 需要描述模板与数据之间的关系

方案: 把脏活累活自动化

1. 程序分析模板的异步数据， 自动拆分模板
2. 根据模板上下的依赖关系，自动收集数据依赖
3. 自动把局部模板和数据关联

自动化分块传输

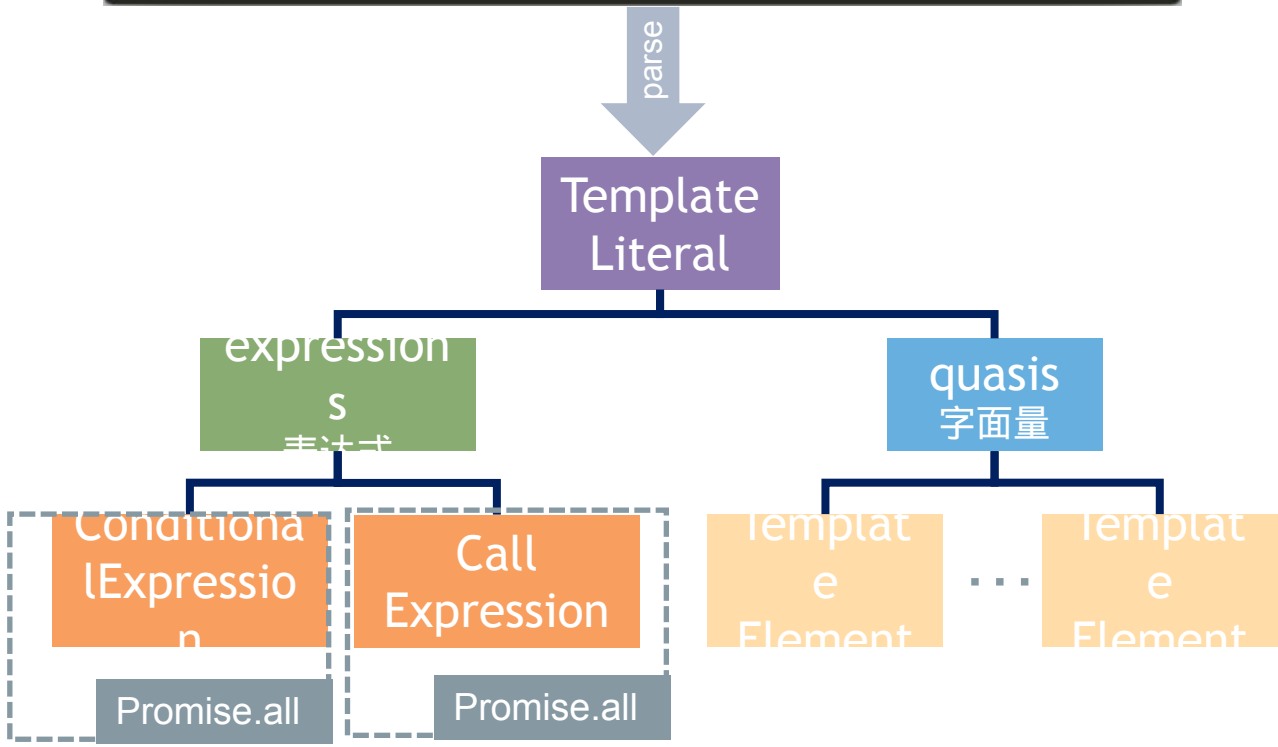
方案: 语法树分析

因ES6 Template的表达式， 有子程序或模板

故需分析局部模板和数据依赖:

1. 查找第一层的模板解析表达式(${})节点
2. 分析并收集该节点下所有的与数据相关的标志符
3. 用Promise.all包含该节点并重命名标志符

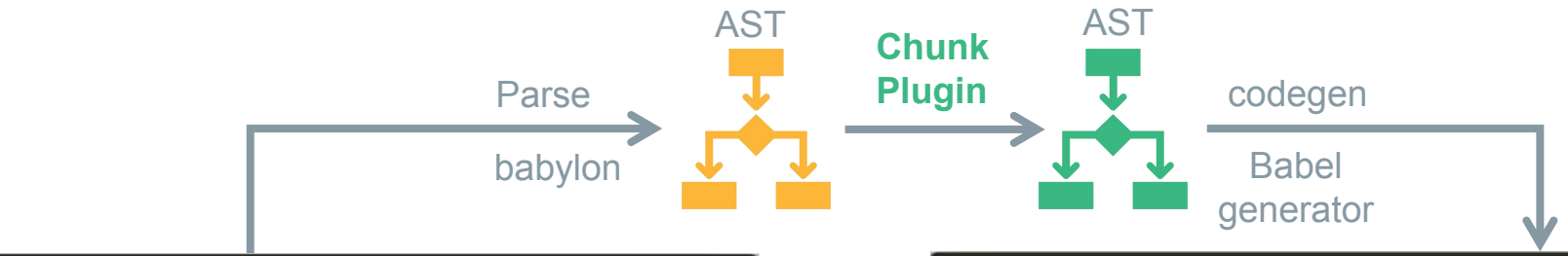
```
function render({header, list, hasJoin}) {
  return `
  <div>${hasJoin ? header.title : "加入" }</div>
  <ul>${list.map(
    item => hasJoin ? `<li>${item.name}</li>` : '')
    .join('')}
  </ul>`
}
```



语法树处理异步数据示意图

自动化分块传输

分析数据与局部模板的依赖




```
function render({header, list, hasJoin}) {
  return `
  <div>${hasJoin ? header.title : "加入" }</div>
  <ul>${list.map(
    item => hasJoin ? `<li>${item.name}</li>` : '')
    .join('')}
  </ul>`
}
```

编译前

```
function render({ header, list, hasJoin }) {
  return `
  <div>${Promise.all([hasJoin, header])
    .then(([_hasJoin, _header]) => _hasJoin ? _header.title : "加入")}</div>
  <ul>${Promise.all([list, hasJoin])
    .then(([_list, _hasJoin2]) =>
          _list.map(item => _hasJoin2 ? `<li>${item.name}</li>` : '')
          .join(''))}
  </ul>`;
}
```

编译后

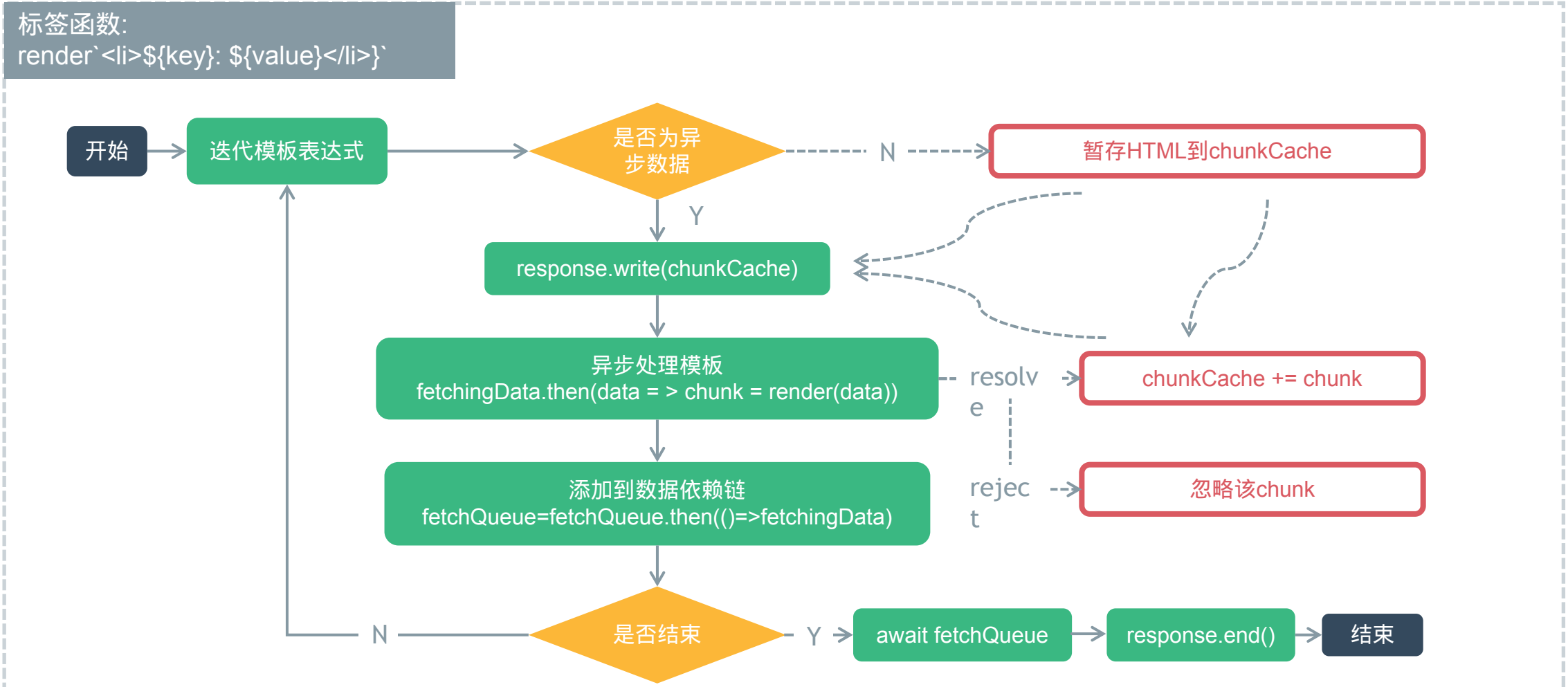
数据依赖收集流程图

ES6 Template不能解析 Promise等异步数据？

自动化分块传输

标签函数内处理异步数据

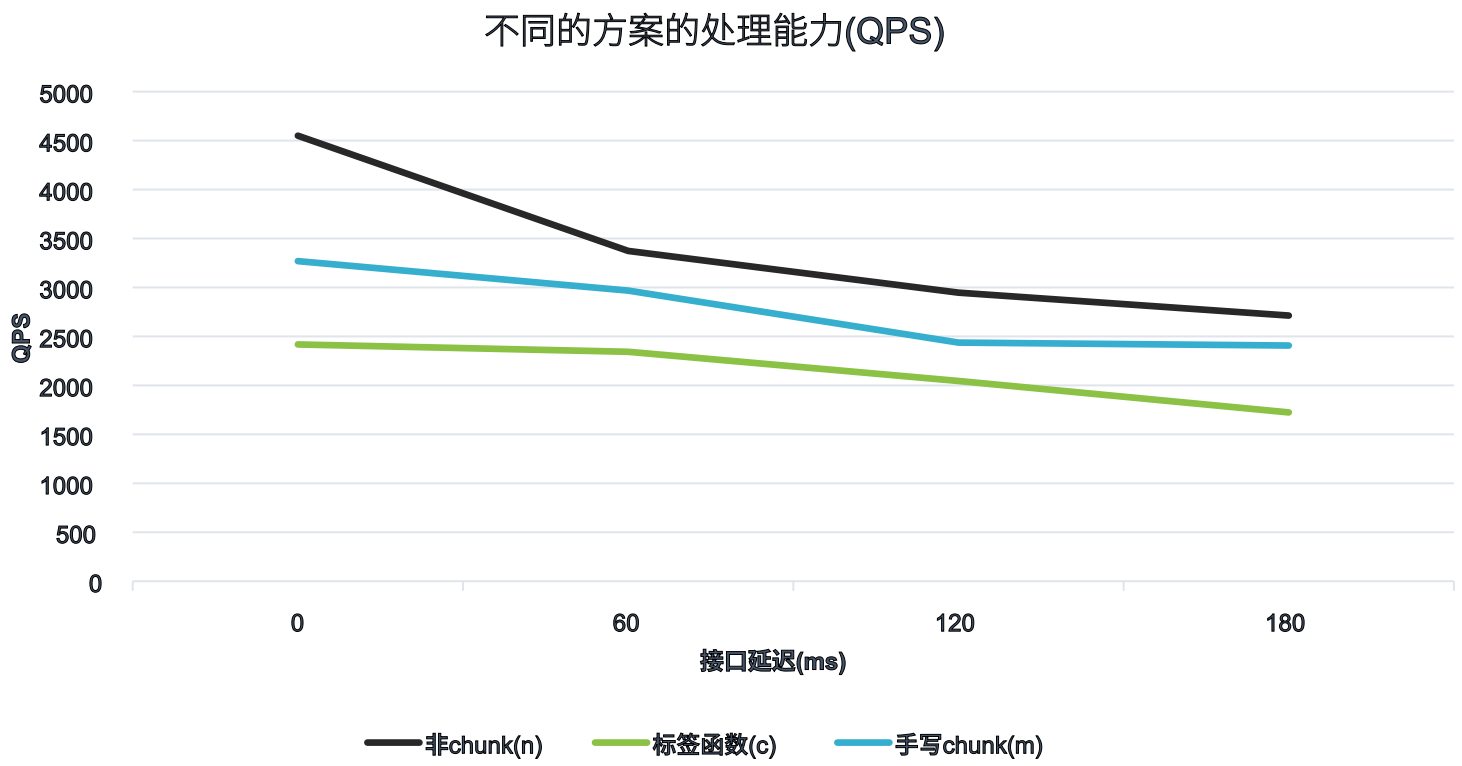
标签函数:
render`<li>${key}: ${value}</li>}`



用标签函数支持异步数据解析: 1. 缓存输出， 直至下一个异步数据前 write到客户端。 2 构建依赖链

自动化分块传输

标签函数方案的性能



标签函数的方案比手写分块的性能低16~28%

新目标：性能媲美手写分块

* 采样对象有8个异步请求，8个分块

自动化分块传输

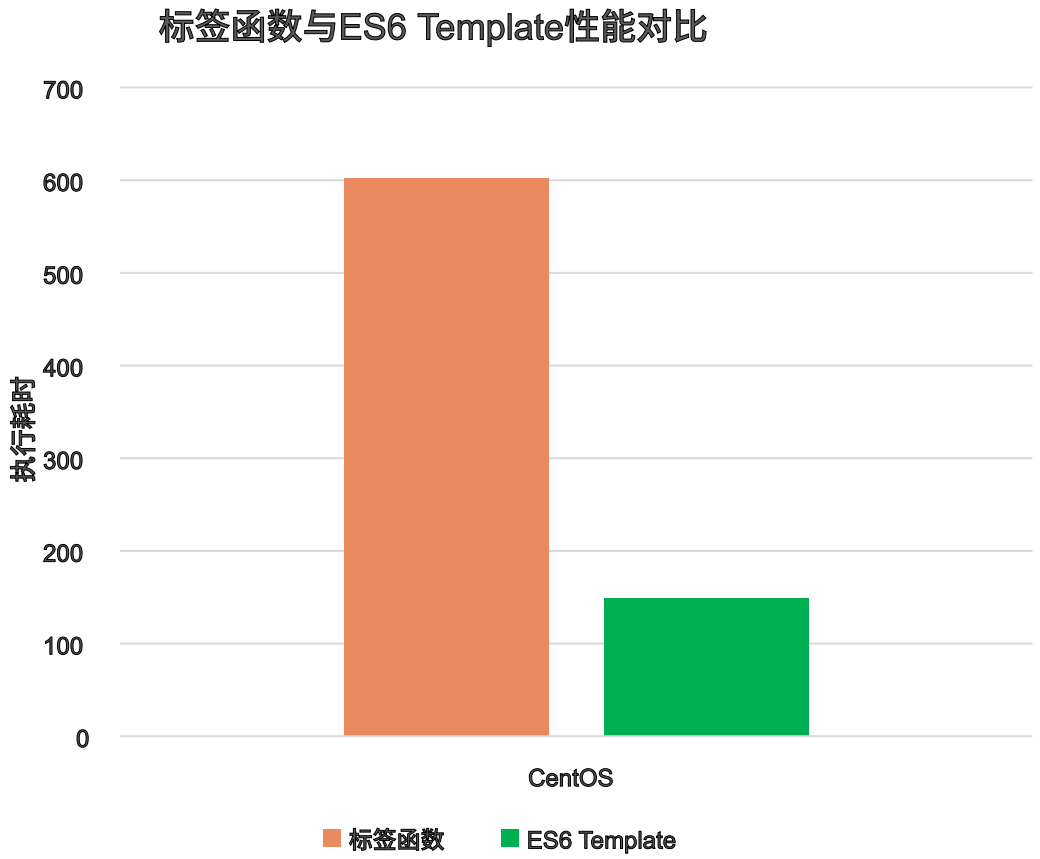
分析性能问题

有性能消耗：

1. 构建Promsie链
2. 字符串的拷贝(chunkCache += chunk)
3. 标签函数

在CentOS中纯ES6 template的性能是标签函数的4倍

上面都是在运行时不变， 编译时可以确定的条件， 所有都可以在编译时计算

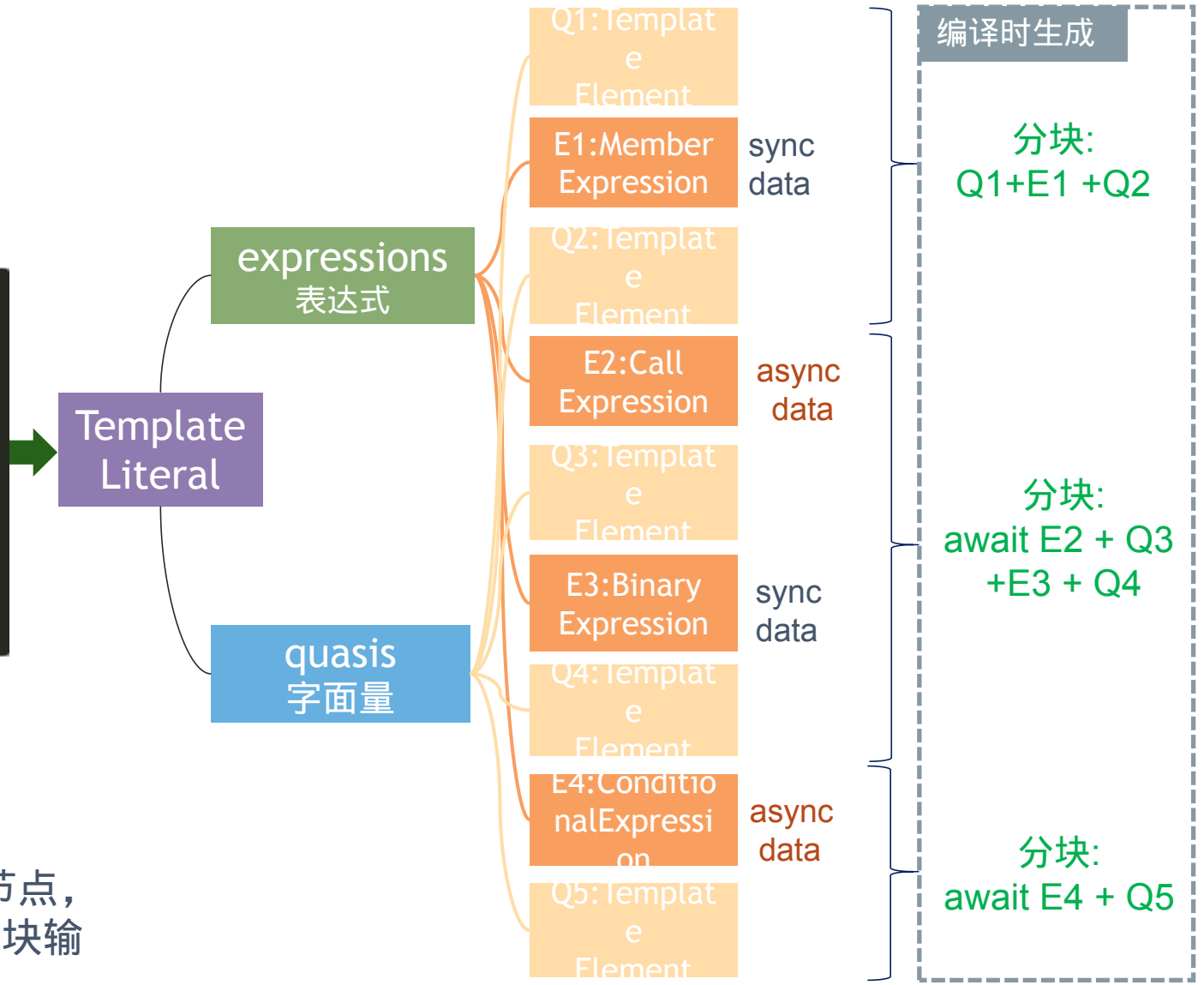


* 5个表达式的模板执行100k次的性能， 越低性能越好

自动化分块传输

解决性能问题: 编译时计算

```
function render({asyncData1, asyncData2},{syncData1, syncData2}) {
  return `
  <div>${syncData1.title}</div>
  <ul>${asyncData2.map(E2
    item => hasJoin ? `<li>${item.name}</li>` : '')
    .join('')}
  </ul> Q3
  E3
  <section>${syncData2 + "_suffix"}</section> Q4
  ${asyncData2 ? asyncData2 : "backup"}`
}
```



模板语法树分块示意图

运算放在编译时:

分析Templateiteral(在语法树上表示模板字符串)节点，用异步的表达式分割模板， 转字符串拼接， 分块输出

自动化分块传输

编译前后对比

编译篇

```
async function render({ data1, data2, data3, data4, data5, has }, { query }, res, onerror)
{
  res.write(('\n<!DOCTYPE html>\n<html lang="en">\n<head>\n  <meta name="viewport"
content="width=device-width, initial-scale=1.0">\n
<title>Chunk</title>\n</head>\n<body>\n  <h1 id="' + query.id + '">'));
  res.write((await data1.then((_data) => data1)) + '</h1>\n  ')
  res.write((await data2.then((_data2) => _data2 ? _data2 : "")) + '\n  ')
  res.write((await data3.then((_data3) => _data3.title)) + '\n  ')
  res.write((await data4.then((_data4) => _data4.list.map(i => `<li>${i}</li>`).join(''))))
+ '\n  ')
  res.write((await Promise.all([has, data5]).then(([_has, _data5]) => _has ? _data5 : ''))
+ '\n</body>\n</html>')
  res.end();
}
```

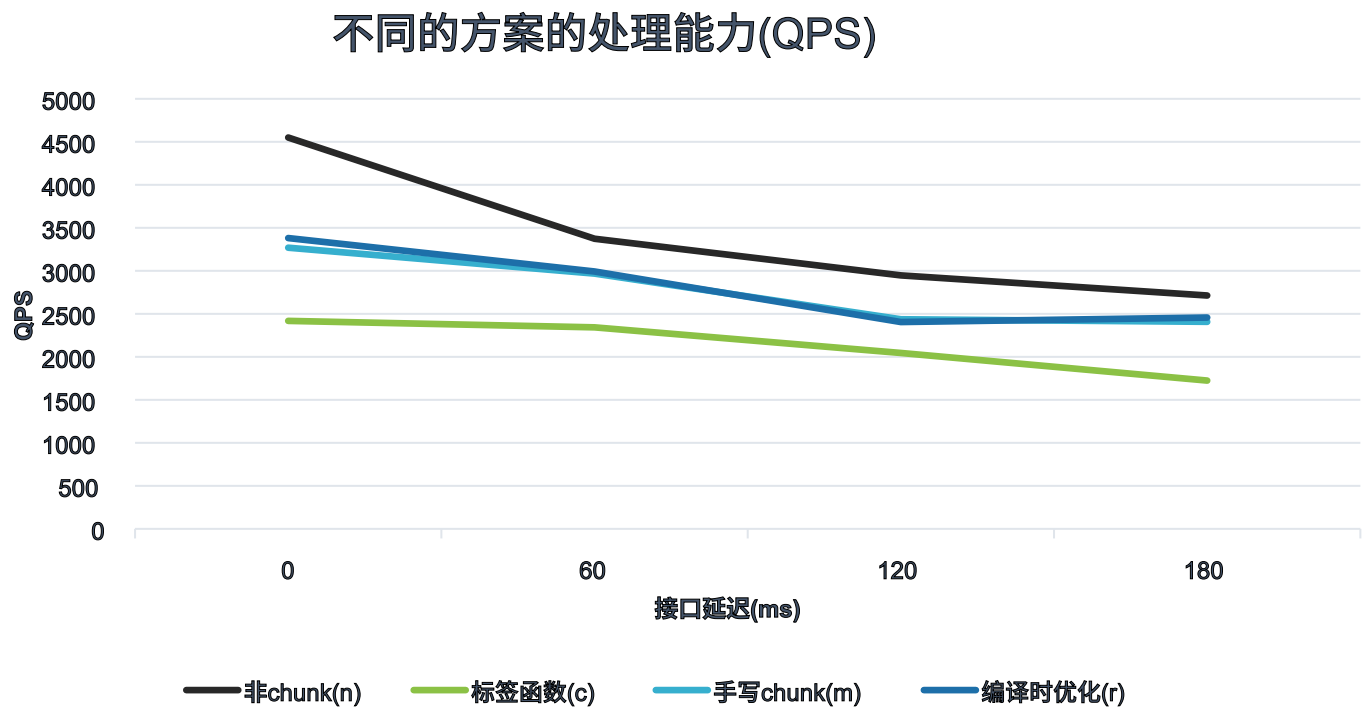
编译时计算：

1. 构建Promsie依赖链(转await)
2. 字符串的拷贝(转字面量)
3. 标签函数(转拼接)

* 为了展示方便， 编译后去除了异常兜底的逻辑

自动化分块传输

编译时优化后的性能：手写分块一样的性能



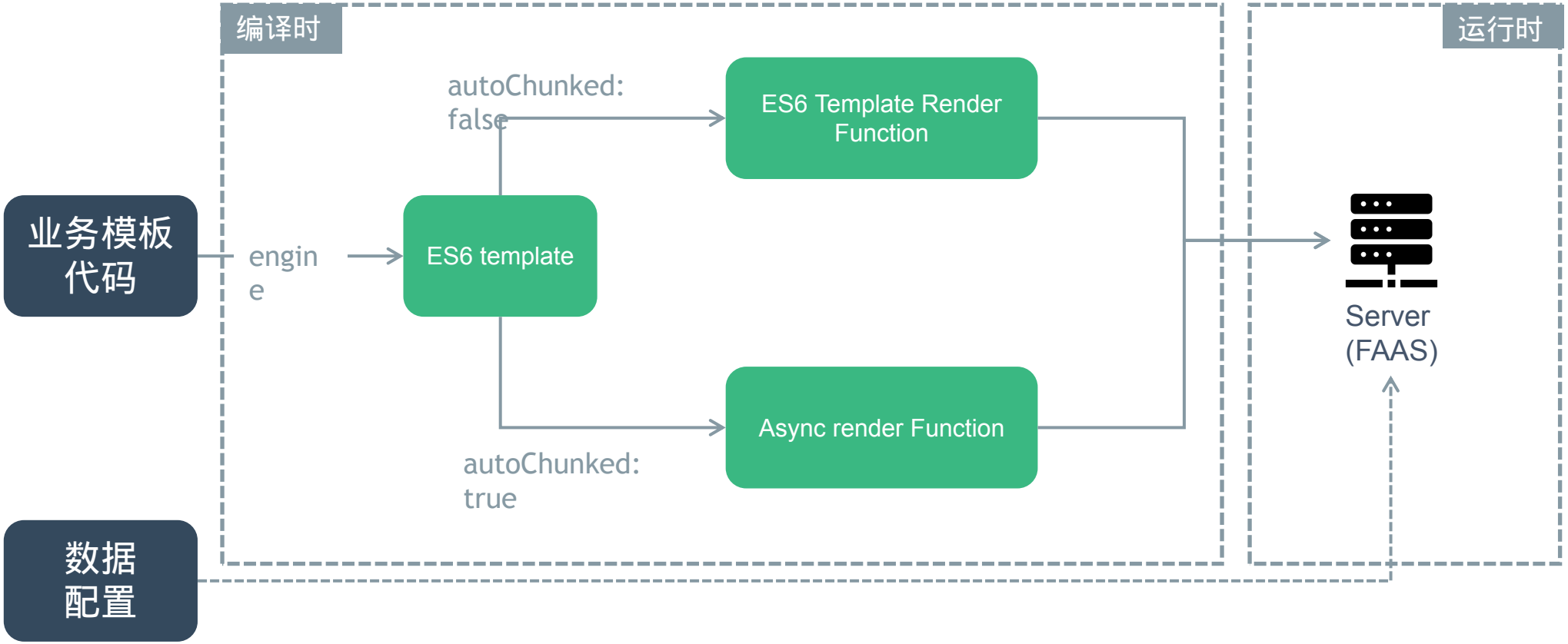
通过把运行时的计算放到编译时，
可以达到和手写分块一样的性能

在接口的平均响应时间为60ms时，
相比标签函数QPS可以提升26%

* 采样对象有8个异步请求，8个分块

自动化分块传输

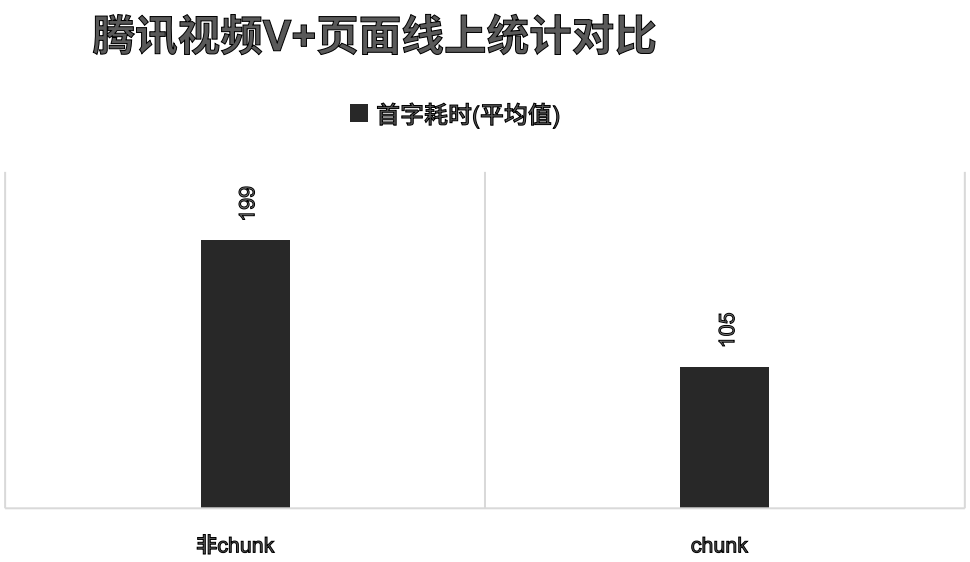
业务无关， 自动分块: 非分块一样的开发效率



编译时开关， 一键切换分块/非分块, 和普通非分块页面一样的开发效率， 业务开发无感知

自动化分块传输

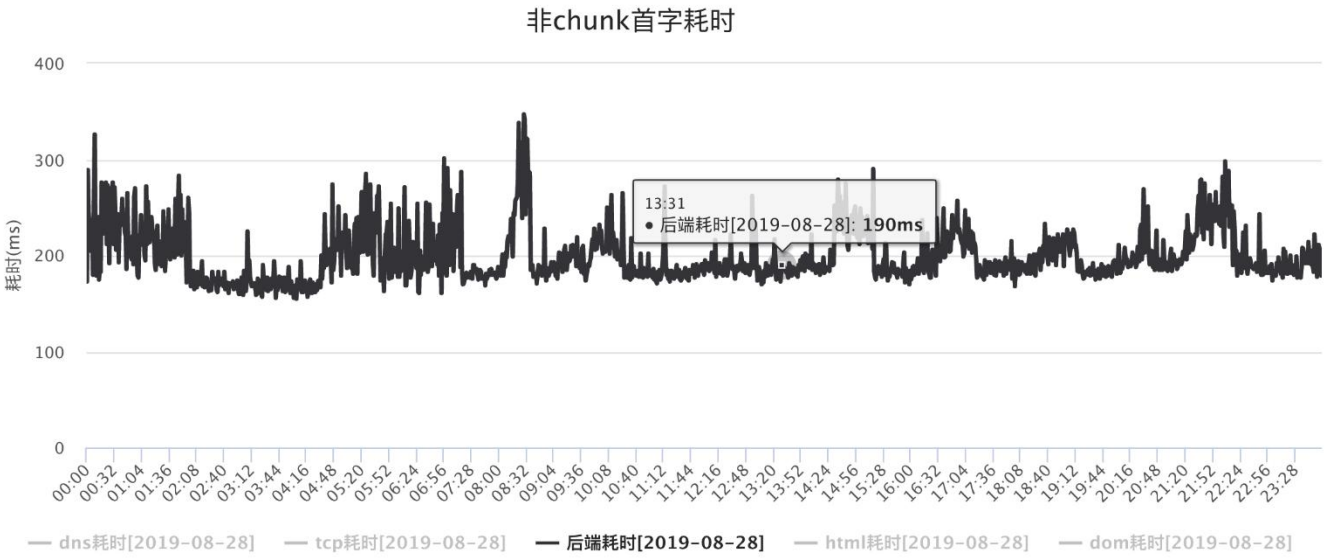
效果对比

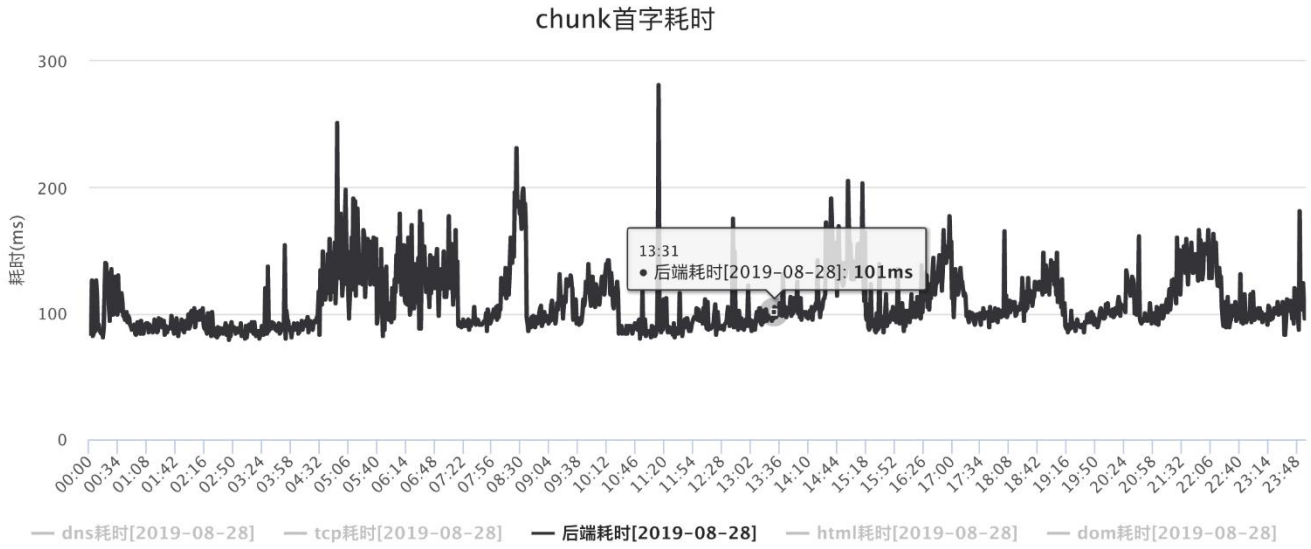


线上的环境较复杂， 首字(TTFB)耗时

分块相对非分块提高了47%

线上实时效果对比:





* 后端耗时 = responseStart - requestSta

自动化分块传输

效果对比

优化前

1. 人力手工拆分模板

2. 运行时维护异步数据间的依赖链

3. 需要描述模板与数据之间的关系。

优化后

1. 自动管理异常和依赖：自动兜底数据请求失败，超时等异常， 分析模板数据依赖
2. 编译时处理依赖链性能无忧： 和手写原始的分块一样的性能, 比业务在运行时封装的分块性能更好
3. 一键切换分块/非分块：编译时开关， 一键切换分块/非分块，普通非分块页面一样的开发效率，业务无感知

自动化分块传输

小结

一句话说方案： 由模板的语法树， 分析代码的上下文，分析数据和模板间的依赖， 用异步数据分割模板， 分块逐步输出

优化奥义：工作量大的部分的自动化， 有性能损失且编译时确定的运算放在编译时

SSR现状: 效率低

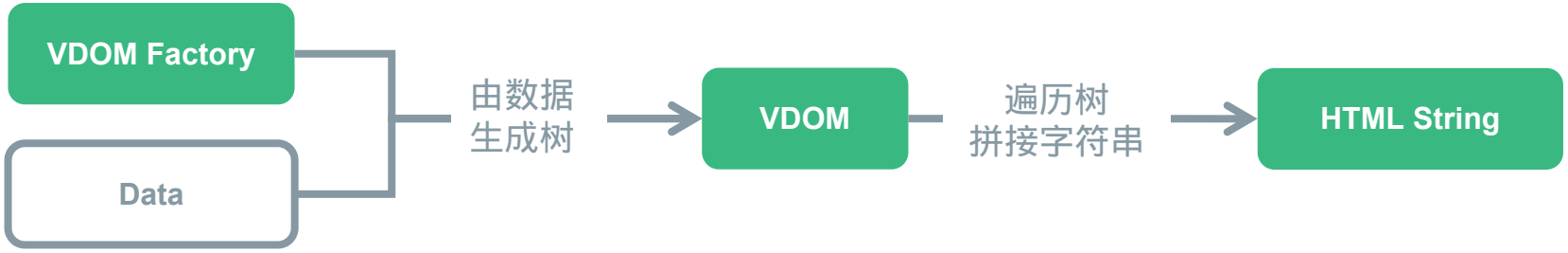
1. **开发效率低:** 传统的SSR, 需要操作DOM, 开发效率低， 难维护

2. **同构的可响应时间长(TTI):** 由于同构需要客户端激活DOM, 可响应时间长

3. **同构服务端性能低:** HTTP服务端无状态无需状态管理， 也无需虚拟DOM

SSR现状: 同构的SSR性能低

字符串的拼接是SSR最简单粗暴有效的方式



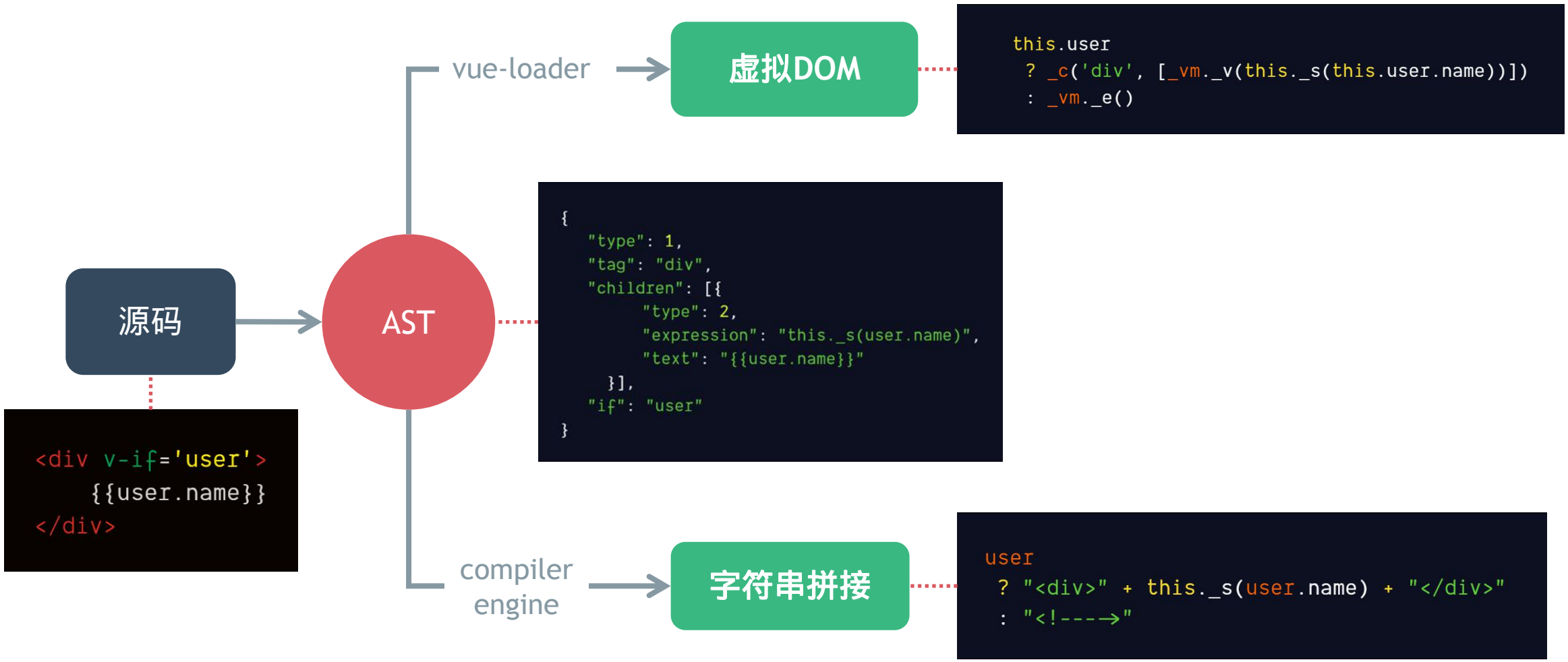
基于VDOM的SSR流程图

由于需要生成树再遍历树基于VDOM的SSR有比较大的性能损耗

Vue编译时优化

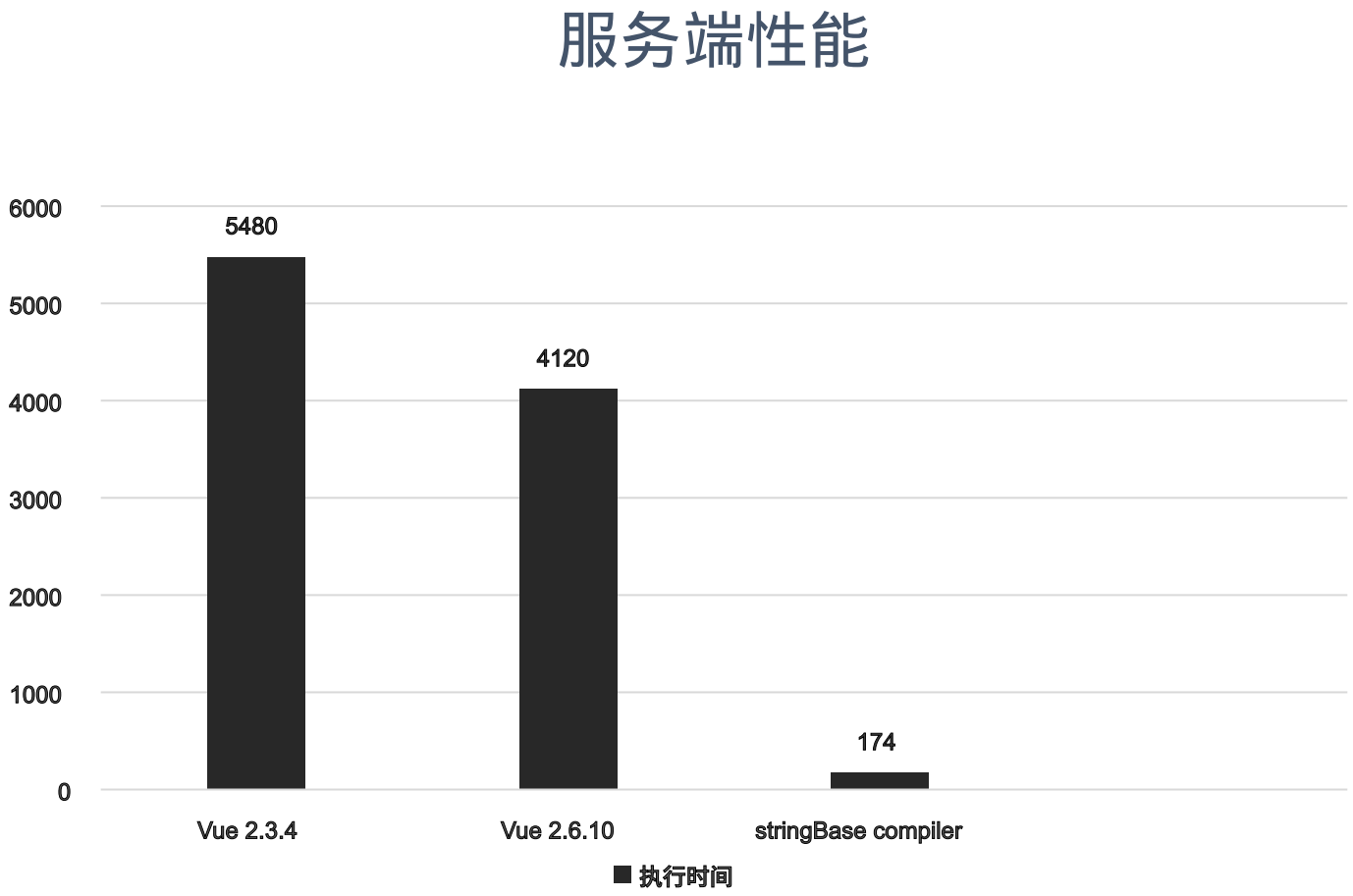
编译时优化思路

自研Vue编译引擎：在编译时把Vue语法转化为字符串拼接



Vue编译时优化

编译引擎效果



比Vue基于VDOM的官方SSR方案快了20+倍(Vue2.3.4快30+倍)

Vue编译时优化

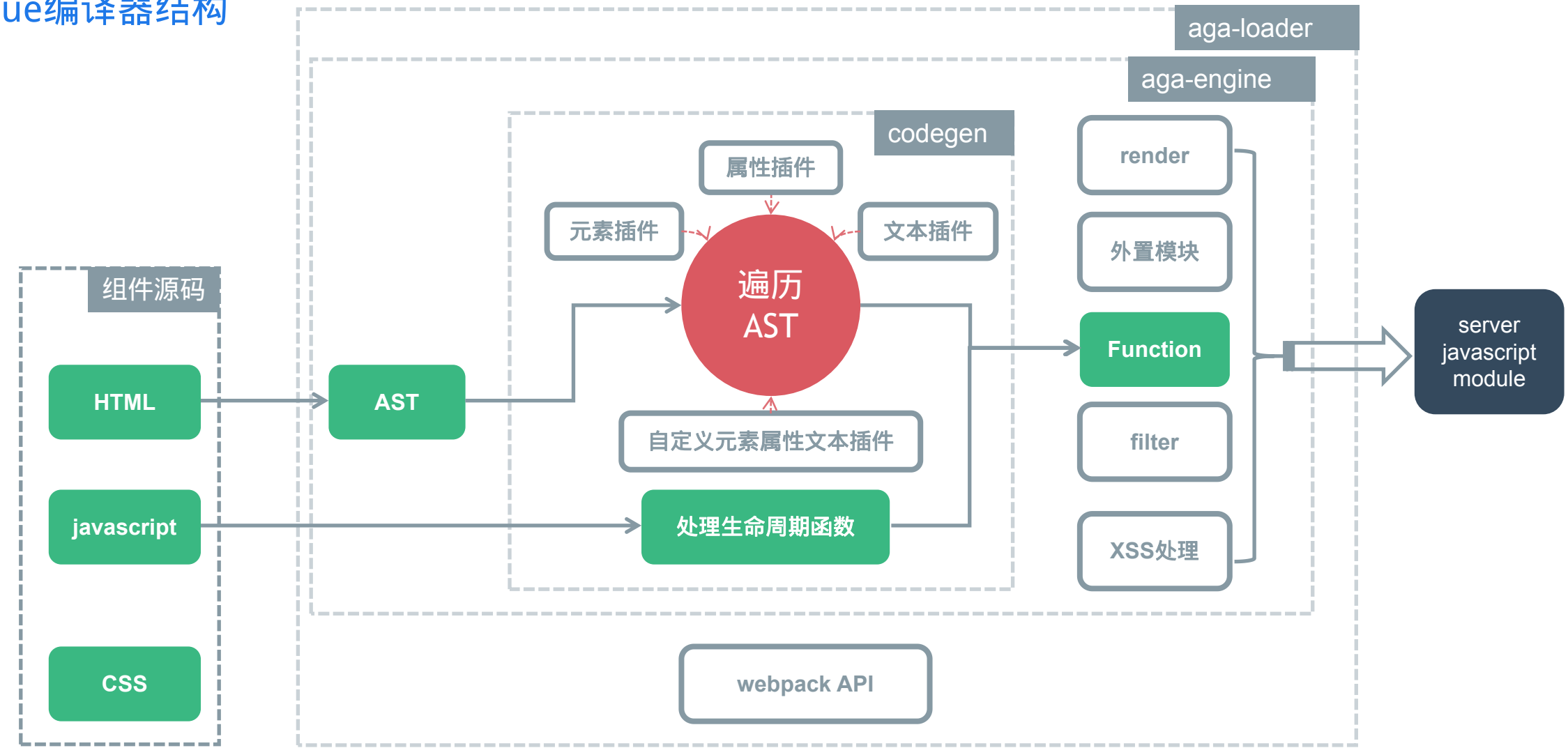
小结

一句话说方案： 由Vue语法树，在编译时生成线性字符串的拼接， 无需构造和遍历VDOM树形性能更高.

优化奥义： HTTP传输是字符流， HTTP协议的内容部份(HTTP2.0数据帧)都可以用字符串表达。字符串的拼接是SSR最简单高效的方式！除了字符串拼接， 其它计算放在编译时

高效开发

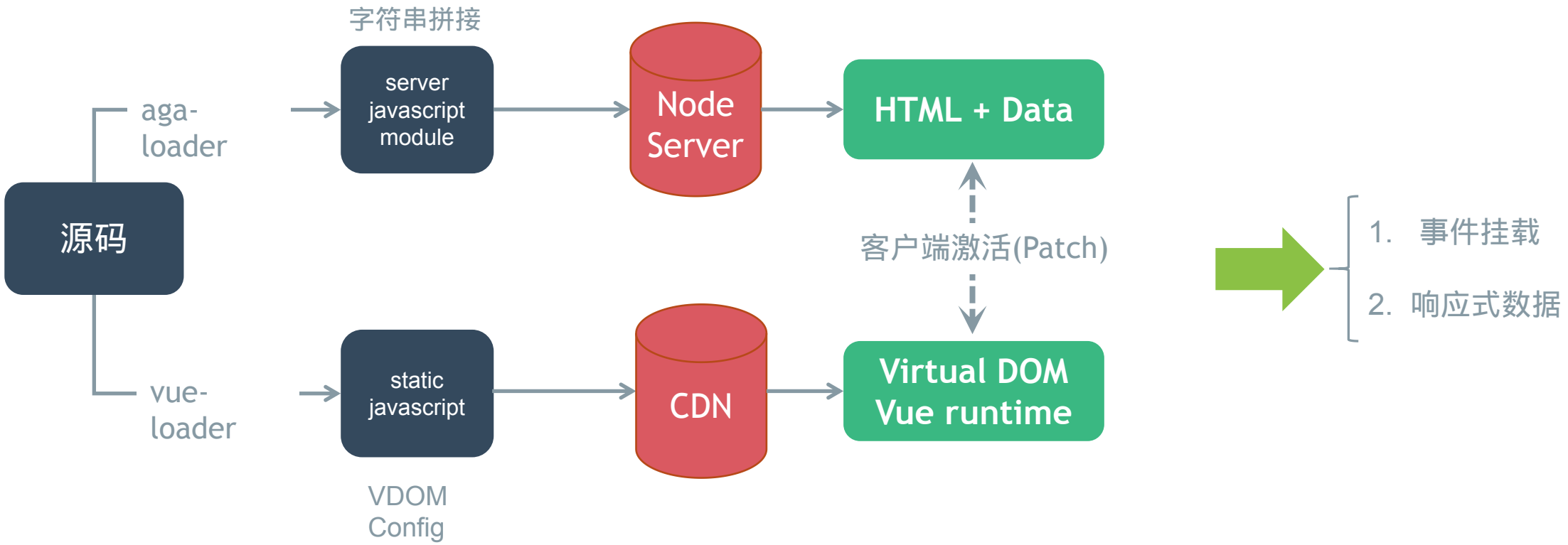
Vue编译器结构



engine处理流程图

高效开发

同构渲染流程



同构流程示意图

高效开发

效果

1. 编译器覆盖了Vue 90%+的语法，统一了前后端语法

2. 目标语法是ES6 Template或字符串拼接， 性能高， 对开发透明， 支持的Serverless服务结合。

3. 相对当前的同构方案提供了一种简单的Vue同构方案，不用担心性能问题， 统一服务端的模板语法。

目前编译器在腾讯内部已应用在100+个页面。

高效开发

不同渲染模式的特性

渲染模式	首次渲染FP	首屏FMP	可交互TTI	开发效率
前端渲染	★★★★	★	★	★★★★
直出	★★★★	★★★☆	★★★★	★
页面同构	★★★★	★★★☆	★	★★★

同构的可交互时间相比直出多了Patch时间可交互时间TTI比较长, 如何优化？

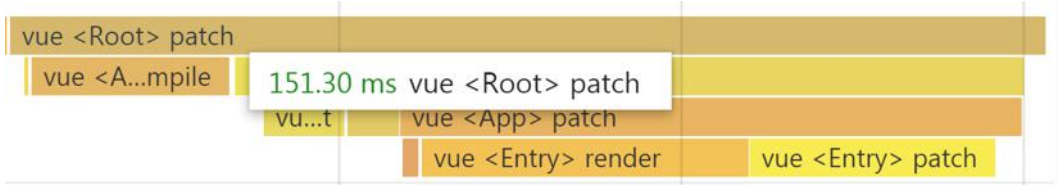
高效开发

同构的问题

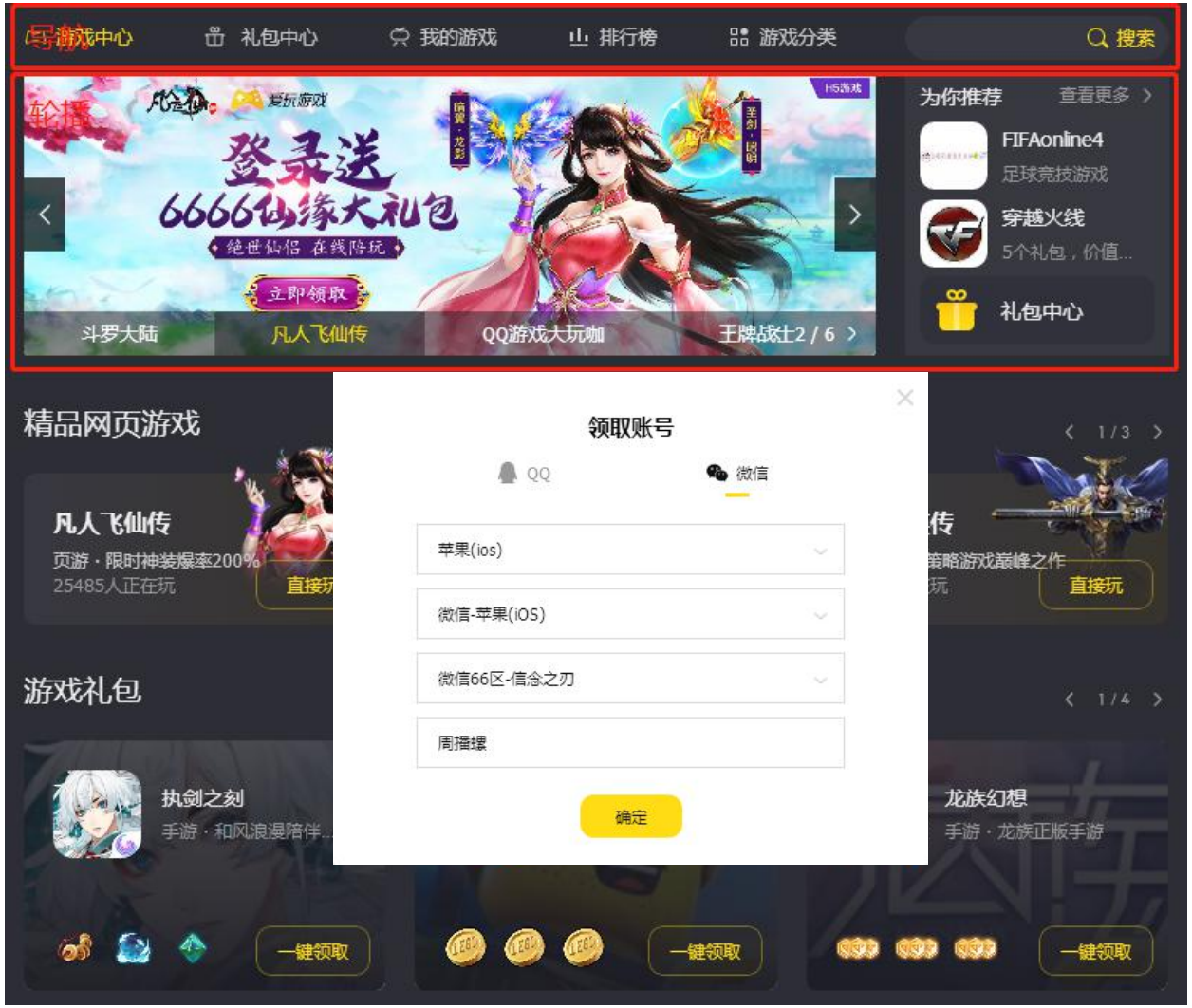
并不是所有的模块适合用同构

1. 有些模板就是静态的数据和简单的交互

2. 同构在客户端有激活的计算成本，激活的时间和DOM量正相关



客户端激活的成本



高效开发

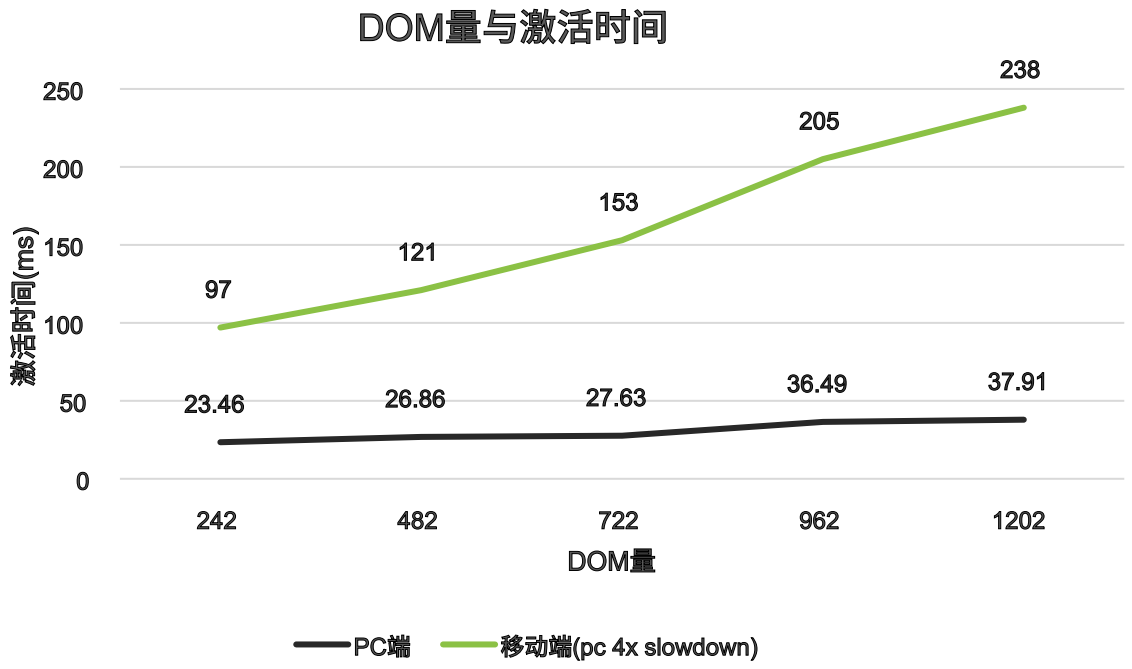
局部同构的方案和性能

方案:

把页面交互多的部分用同构， 其它直出

实现: 同一页面只激活同构的DOM，其它DOM和直出完全一样， 不激活。

性能对比:



降低不必要的同构DOM量， 可以大大提升移动端激活性能

采样硬件：i7 16G

高效开发

渲染模式及适应的场景

渲染模式	首次渲染FP	首屏FMP	可交互TTI	开发效率	适用场景
前端渲染	★★★★	★	★	★★★★	需要快速上线， 表单比较多， 不关心首屏和SEO
直出	★★★★	★★★★	★★★★	★	SEO,首屏快, 业务简单，交互较少
全页面同构	★★★★	★★★★	★	★★★	SEO， 首屏快, 交互复杂
局部同构	★★★★	★★★★	★★★	★★★	SEO, 局部交互多，性能相比全同构更好

业务开发可根据场景特点在同一项目的不同页面选择各自适合渲染模式。

同构的TTI优化

小结

一句话说方案： 前端激活的时间和激活的DOM量正相关， 减少不必要的激活, 可以加快可响应时间(TTI)

优化奥义：如果减少每个单位的计算量有困难， 那么就减少那些不必要的计算， 减少总量

SSR现状： 不能增量更新，定义切换动画

SSR页面切换无法渐进式加载， 不能定义页面切换动画

前端路由&服务端直出

无刷新切换：直出单页应用运用前端路由管理模块，

页面无刷新切换的页面状态， 可自定义切换动效。

一句话： 直出单页应用即有直出首屏快，SEO友好的优点， 又有前端渲染页面间切换白屏时间短，可定义动效的优点。

构建高效率高性能Web同构应用

总结

性能优化

1. Vue编译引擎，运行时字符串拼接性能佳， 统一前后台语法;

2. 前端路由管理模块，页面无刷新的增量更新，自定义页面切换动画

效率优化

1. 自动化分块传输， 编译时分析优化让分块和非分块工作量一样， 比一般的封装分块性能更好， 支持一键切换

2. 支持同构、局部同构等多种渲染模式， 声明式开发效率更高

WEB CONF
<腾 讯 前 端 技 术 大 会/>

极致SSR

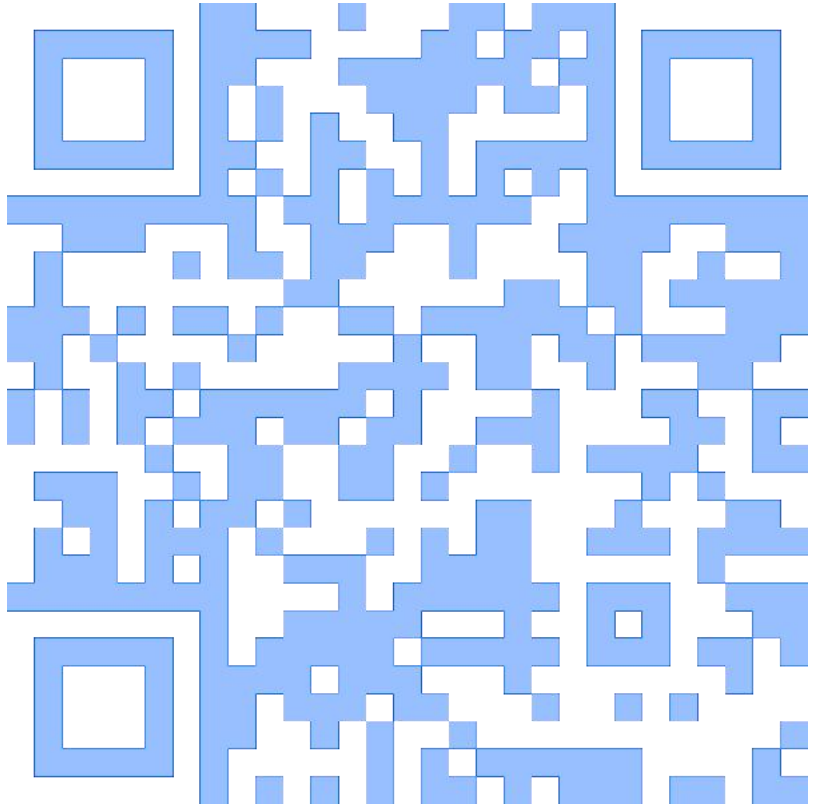
编译优化，写的更少，运行更好

Compile-time optimization: write less, run better

段隆贤

扩展

宏观架构方面：腾讯视频Node.js服务如何支撑高并发



TWEB CONF

<腾 讯 前 端 技 术 大 会/>

<Thanks/>

主办方 Organizer Tencent TWeb


Q/A