

页面加载性能优化 + Maiev平台介绍 + 排行榜

页面实践

本文启发自 [Webapp 性能优化指导手册](#)。读完此文后受益匪浅，建议阅读。

页面加载耗时很大一部分时间耗费在了网络请求资源上。其中**请求**和**资源**是两个优化的方向。

通用请求资源优化措施：

缩短请求链路

- 将资源部署在CDN上
- 预请求 preload & prefetch
 - preload **关键**资源，在页面渲染前请求。常见的关键资源有样式文件、字体，来避免闪烁。
 - Prefetch 在页面**空闲**时请求资源。常用于预请求spa页面路由切换后的资源。（注意是当前页面内需要的资源。与之相对的是dns-prefetch & preconnect，用于加速稍后可能点击的第三方链接的页面加载）
- 缓存资源（service worker、storage、html cache）

关于html cache的一个重要的优化点是保证文件内容无变化时可以复用缓存。

 - webpack配置：使用contenthash命名文件；runtime单独成chunk；配置ModuleID不随build改变。
 - 将不变的内容和会经常变化的内容单独打包成各自文件，分别缓存。
- 开启Gecko离线化

端内开发，可开启Gecko离线化，资源直接下发到本地，由客户端拦截请求并拉取本地资源。

减少资源本身的大小：

对于单个资源的请求来讲，不考虑其他影响因素，资源大小越小，请求完成越快。

- 使用最新版本的webpack

通常来讲，webpack版本越高，则其打包效果越好。从webpack4迁移到webpack5，可看官方的总结：<https://webpack.js.org/migrate/5/>。

TODO: 迁移成本验证

TODO: 对于使用webpack-chain的项目, 迁移是否会遇到痛苦?

- 打包时使用各种资源的Minify插件来减少体积（如terser、css-minimizer-webpack-plugin等）

- 开启http压缩

常见的压缩协议有gzip和brotli，大多数场景下brotli的压缩比更好。

待确认：目前公司的公有cdn域名尚不支持brotli，我们是否可改懂车帝的私有cdn域名开启brotli支持？ -- 懂车帝私有CDN已开启brotli支持。

- 使用合适的文件压缩

如图片在适合场景下使用webp。

- 替换包 / 缩减包体积：

如echarts包，如果正常引入，体积庞大。此时有两个思路

- a. 如果只需要一个简单的制图而不需要其他功能，可考虑使用更轻量的第三方包（甚至手撸canvas）。
- b. 某些包如echarts有按需引入的设计。<https://echarts.apache.org/handbook/zh/basics/import/>

另外，按需引入包不仅减少网络加载时间，也能减少js执行时间。尤其是如果要引入的包有很多初始化的逻辑但却没用到相关的功能时，这部分执行时间可以被节约。

- 不必要的包不要引入

webpack中有两项配置可以删除unused code，一是usedExports，二是sideEffects。

- useExports: 利用terser在statements层面评估其是否有副作用。缺点：要执行到深层模块，且评估不一定准确（比如遇到HOC时），可通过/*#__PURE__*/指令解决一部分问题。
- SideEffects：在module层面起作用，简单清晰。缺点：大部分第三方包未配置。

具体措施：

1. 我们自身的仓库，应合理配置sideEffects来减少本身的bundle体积。
2. 当import了没使用的包时，只有当包支持es6语法、且其packagejson正确设置了sideeffect，才可在我们项目的webpack treeshaking成功删除掉。但是大部分包没有正确设置或没有支持es6，这要求我们要避免引入没使用的包。（这里我通过eslint的no-unused-vars规则来找到并删除这些import，但是还有一些import了但没有赋值给变量的，目前没找到好的解决办法，只能在开发时留心）。

- 按需引入polyfill

有些polyfill是为了解决兼容性，按需引入可节约体积。

这里必须使用if语句+dynamic import才能实现按需引入，这样打出来的包是async chunk。若是单纯的if语句+ require，chunk为initial而非async。initial表示同步加载，即页面进入后就会请求。async表示是由dynamic import引入的，逻辑执行到懒加载时才有可能请求（这里可再加webpackPrefetch或webpackPreload指令来提前请求时机）。

• Modern build

一些现代浏览器已经开启了部分es6语法的支持，而常用的babel config targets默认会兼容低版本浏览器。这既增加了代码体积，也会浪费部分性能（因为原生api的性能要好过polyfill）。

我们可以在构建时分别创建两个babel config，一个是现代的配置（编译产物为modern.js），另一个是兼容的配置（编译产物为legacy.js）。然后在html里同时引入 `<script type="module" src="modern.js">` 和 `<script nomodule src="legacy.js">`。支持module的浏览器（现代浏览器）会加载前者，老浏览器则会加载后者。

• 解决包冗余（同包不同版本的重复依赖问题）

由于npm的工作原理，当不同的包依赖相同包的不同版本时（或者依赖要求1.x这种范围版本），node_modules里会出现不同版本的同一个包，称之为duplication。

若不处理，首先有可能引起线上的问题：比如某个包的A版本有问题，在B版本修复了，但因为重复依赖没有删除，A版本代码还在项目中；再次有些包是单例模式，如react，项目是无法同时存在不同版本的react的。另外冗余出现，重复引入了包，增加了体积。

解决办法：若使用npm管理包，则运行npm dedupe。

NPM包一般遵循semver语义，即【MAJOR.MINOR.PATCH】。

- 不同MAJOR的重复依赖不可删除。
- 同MAJOR的重复依赖不一定能删除，原因是并非所有包的开发者都遵循semver，即使是同MAJOR，也有可能发生breaking changes，所以测试回归是必须。（注：MAJOR是0时，也不遵循semver，dedupe时不会清除重复依赖。）

【注】发现一个npm奇怪的特性会导致dedupe命令失败（使用的npm6，原理待查明）：npm受安装顺序影响，有时理论上可以被deduped的dependency graph，却没有成功。解法：确保node_modules根目录下的该包版本高于其他包所依赖的该包版本。

将大的请求拆解为多小请求：

对于请求的资源数量来讲，由于我们的服务器已开启对http2的支持，其“多路复用”的特性致使，请求的文件数量多少不会对请求的总时间有较大影响。

- 第三方包（node_modules下的包）单独打包。（这么做的另一理由是便于缓存，上面有提到）
- css代码从js文件中拆出成单独css文件。

- Url-loader等 limit可设置成false/0。

延迟加载（时间轴方向上，先请求必须代码，之后需要时再请求其他代码）。

可显著减少页面初次进入加载时间（若使用不当，会导致需要的信息加载延后，因此需酌情优化）。

- 使用dynamic import: () => import('./lazy-module')。
- 可结合webpack prefetch使用，来优化等待时间。

除资源请求，我们还需要优化同步脚本执行时间

许多页面跑分指标和同步脚本执行时长有关，尤其是长任务(long task)。通过火焰图，找到long task，然后找其中self time长的任务，逐项分析解决。

利用Maiev平台来分析页面性能：

为什么使用maiev

1. 针对打包产物进行信息聚合，有专项的统计与分析
2. 类似lighthouse的跑分

（这个平台后续会加上对gecko离线化的支持，目前还没有。因此如果需要真实的数据，则使用hybrid分析平台）。

3. 易用的性能分析火焰图

- 提供同类型任务聚合。
- 火焰图相较chrome performance更易用，追查问题方便。
- map文件一般不上传至cdn，chrome的performance无法定位任务所在的源码位置。

如何使用maiev

1. 创建项目

<https://maiev.bytedance.net/> 点击create project 填写项目信息。示例：

Create Project



SCM Repo *

motor/fe/pc

Owners *



yuminghao.izyc (X) (+)

Department *

懂车帝-研发-前端

Byte Tree *

| 懂车帝 | 前端 | PC站

OK

Cancel

2. 配置Bundle

使maiev可以收集分析打包产物。

安装、引入maiev的webpack插件，以在构建时将产物和其他所需信息上传至maiev。

Shell

```
1 npm install @perfkit/size-limit-webpack -D
```

JavaScript

```
1 const { SizeLimitPlugin } = require("@perfkit/size-limit-webpack")
2
3 module.exports = {
4   // ...
5   plugins: [ new SizeLimitPlugin() ]
6   // ...
7 }
```

如上配置后，每次scm构建时，打包统计信息便会自动上传至maiev -> bundle模块。

若想本地调试，可在命令行传入以下环境变量： BUILD_REPO_PATH=code.byted.org/motor-fe/autorank（改成你的scm仓库地址） BUILD_REPO_NAME=motor/fe/autorank（改成你的scm仓库名） BUILD_VERSION=local-x.x.x

其中BUILD_VERSION xxx 可任意上传(若不设置 默认0.0.0 平台没有正确处理 会merge之前的构建)

3. 配置Lab

跑性能分 类似lighthouse。

- 在settings里配置pages，增加想测试的页面url信息。可选设备类型(在Profile里配置，之后应该会支持离线化等)和环境请求头更改(在Environment里配置)。
- 【可选】可在byted-motor流水线配置新增atom，maiev_scan，以每次运行流水线时触发跑分
- 可手动触发跑分/take a snap。注意此时scm版本默认是0.0.0，可手动修改。
- maiev平台在

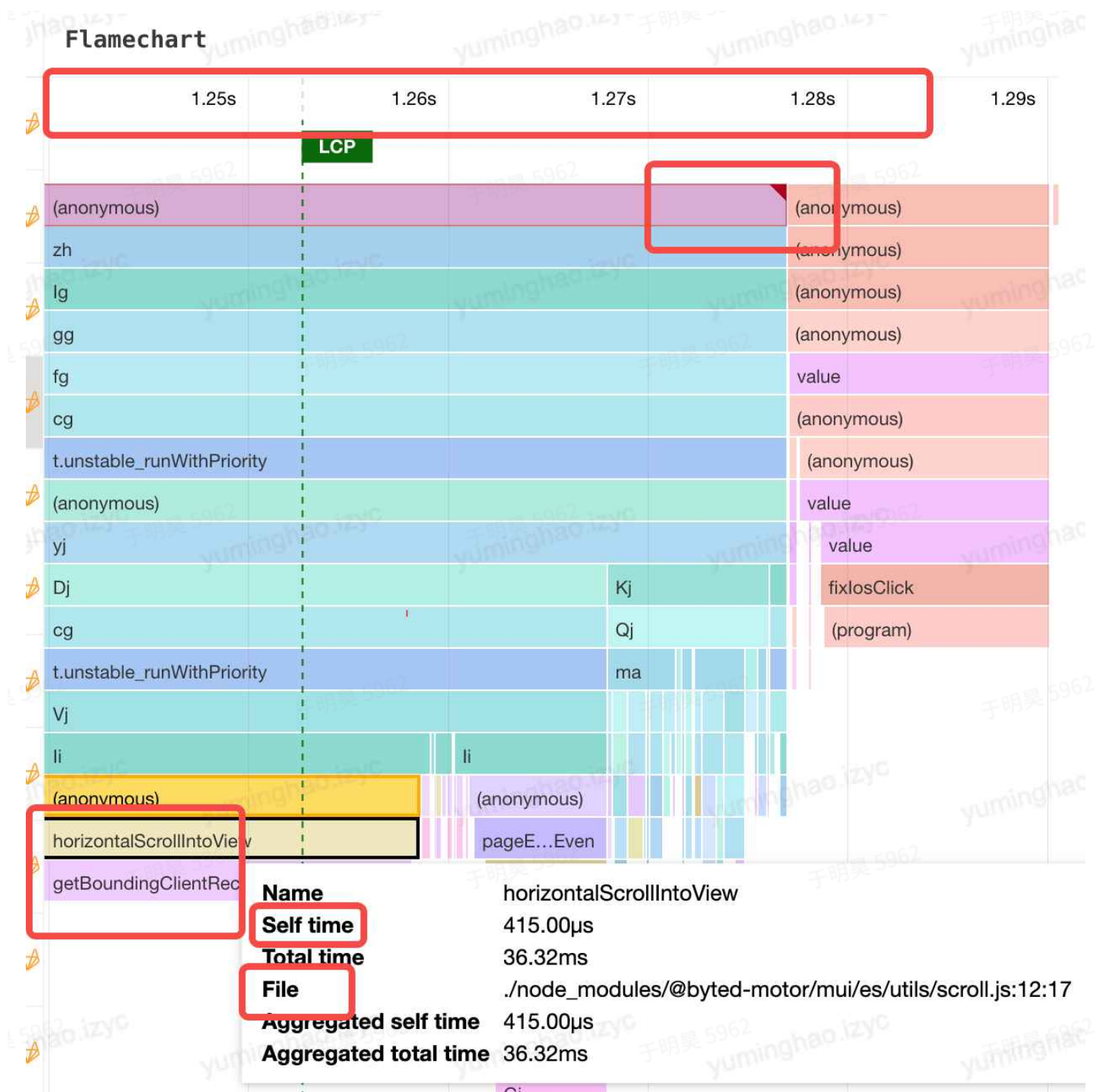
4. 配置source

火焰图、源码定位。

当有**scm版本相同的**bundle和lab同时存在时，会自动触发source任务的运行。

火焰图分析方法

图中有用信息



- 横轴是时间线
- 竖轴从上到下是任务和其调用栈
- 右上角有红色三角形的是长任务
- 鼠标停留在任务上 可看到
 - File: 源代码位置信息。前提是要正确的配置sourcemap。注意，若项目内使用了babel-loader等，要禁用掉它们的sourcemap，以防冲突。
 - Self time: 等于total time - 调用其他子任务的时间。如 function a() {b();ownTask;c()}, 则a的 self time即为ownTask的执行时间。

分析思路





1. 首先，找到所有长任务。长任务指总耗时50ms以上的任务，一般都是页面的瓶颈，需要逐个击破。
2. 对于每个长任务，找到其self time值较大的子任务。
3. 根据file，找到子任务所在的源代码位置，对耗时的产生进行分析，并解决。
4. 有些没有子任务的任务（多为系统任务），需要结合network去搞清楚任务是在做什么。比如是否是在渲染、排版等。以此来帮助我们分析页面的渲染过程。

结合排行榜页面，使用火焰图找到短板：

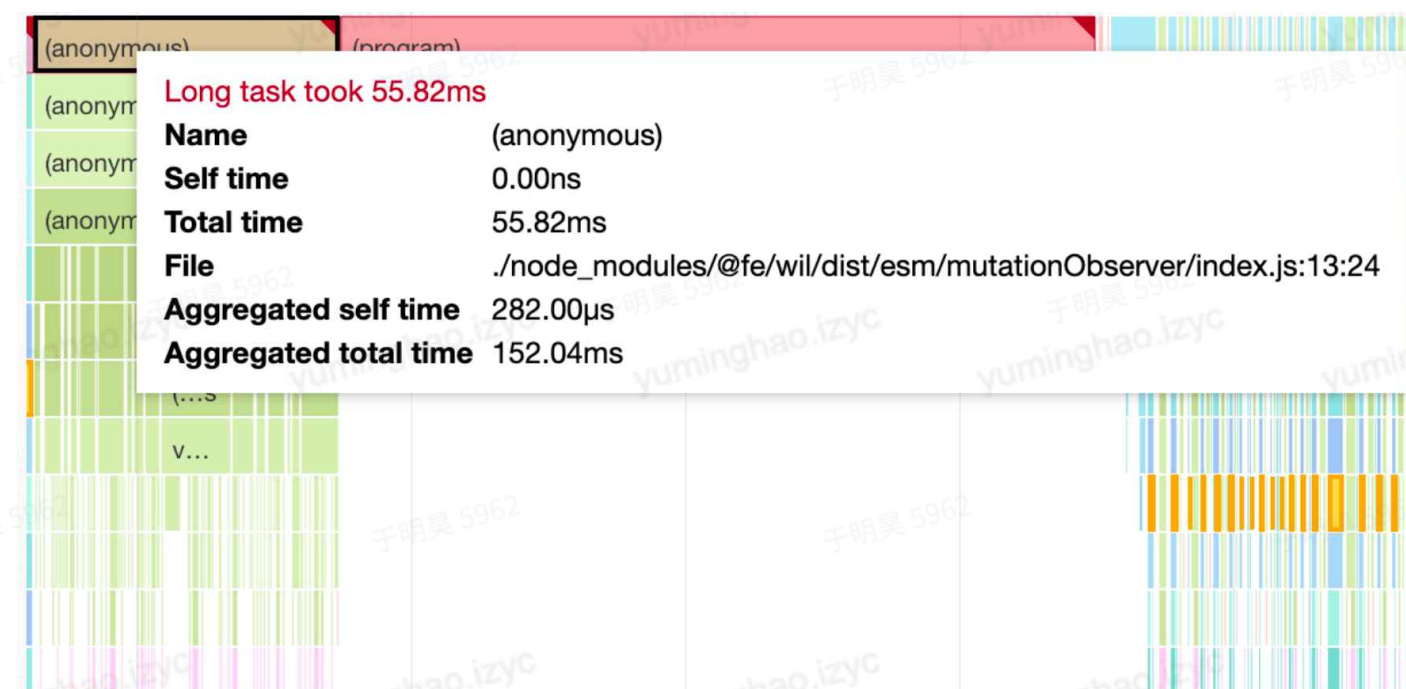
Maiev 链接：<https://maiev.bytedance.net/projects/3608/source?issueId=246111&page=1&reportId=855982&version=1.0.0.545>

1. wil

分析：这里通过file名可看出，wil使用mutationobserver去监听dom并上报埋点，在页面初始化时有耗时操作。从source左侧可看到，wil总耗时116ms，但分布在页面加载初期的大约只有不到50ms。

next	(anonymous)	../../src/mutationObserver/index.ts:15:29	59.04ms			SCM
next	(anonymous)	../../src/mutationObserver/index.ts:15:29	57.42ms			SCM

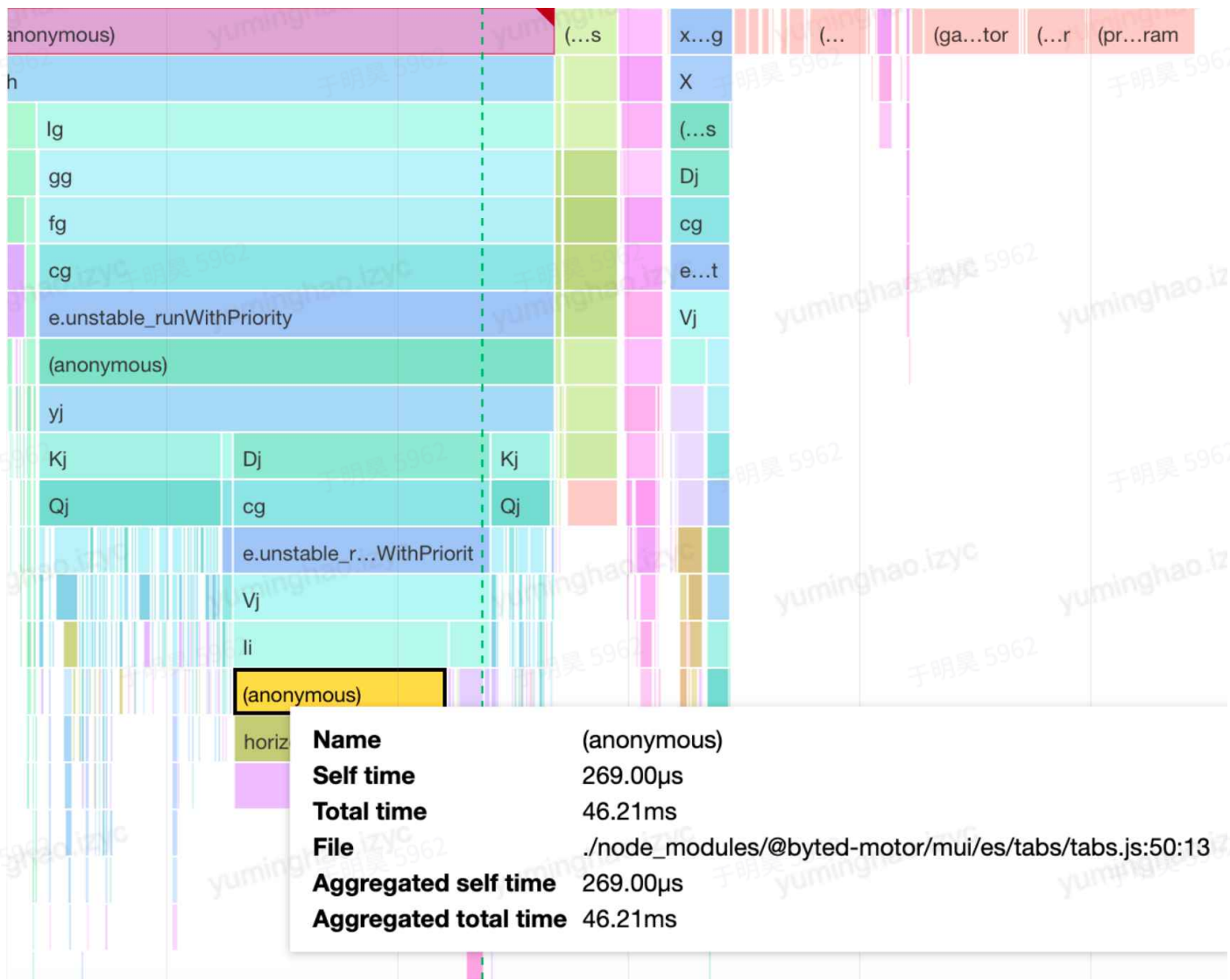
解：找包作者去为了这点性能提升去改动包不现实。排行榜页面埋点也不复杂。如有追求极致性能的需要，可手动通过byted-motor/tea埋点。但是从易用性上考虑，wil仍有保留的价值。



2. mui-tabs。 (46.21ms)

分析：可以看到是@byted-mui/tabs组件有耗时操作。根据file找到源码位置，可看到是组件在初始化时使用getBoundingClientRect去计算了位置，导致了耗时。这在tab索引为0时是不必要的（大部分情况初进入页面时都是0）。

解：可对索引为0的情况特殊处理，避免不必要的位置计算。我们可对MUI贡献代码来解决。这里顺带一提，这种情况下适用npm link来方便同时调试两个仓库（这里是排行榜和MUI）。



3. @bridege/general (≈240ms)

分析：该包是负责jsb初始化和调用的。从jsb的实现原理说，需要js进行一些特殊行为来让客户端接收信息。如console，如这里用到的createIframe。从火焰图可以猜测，general包在性能上还有优化空间。但是bridge不是我们团队维护，可能推动成本会很高。

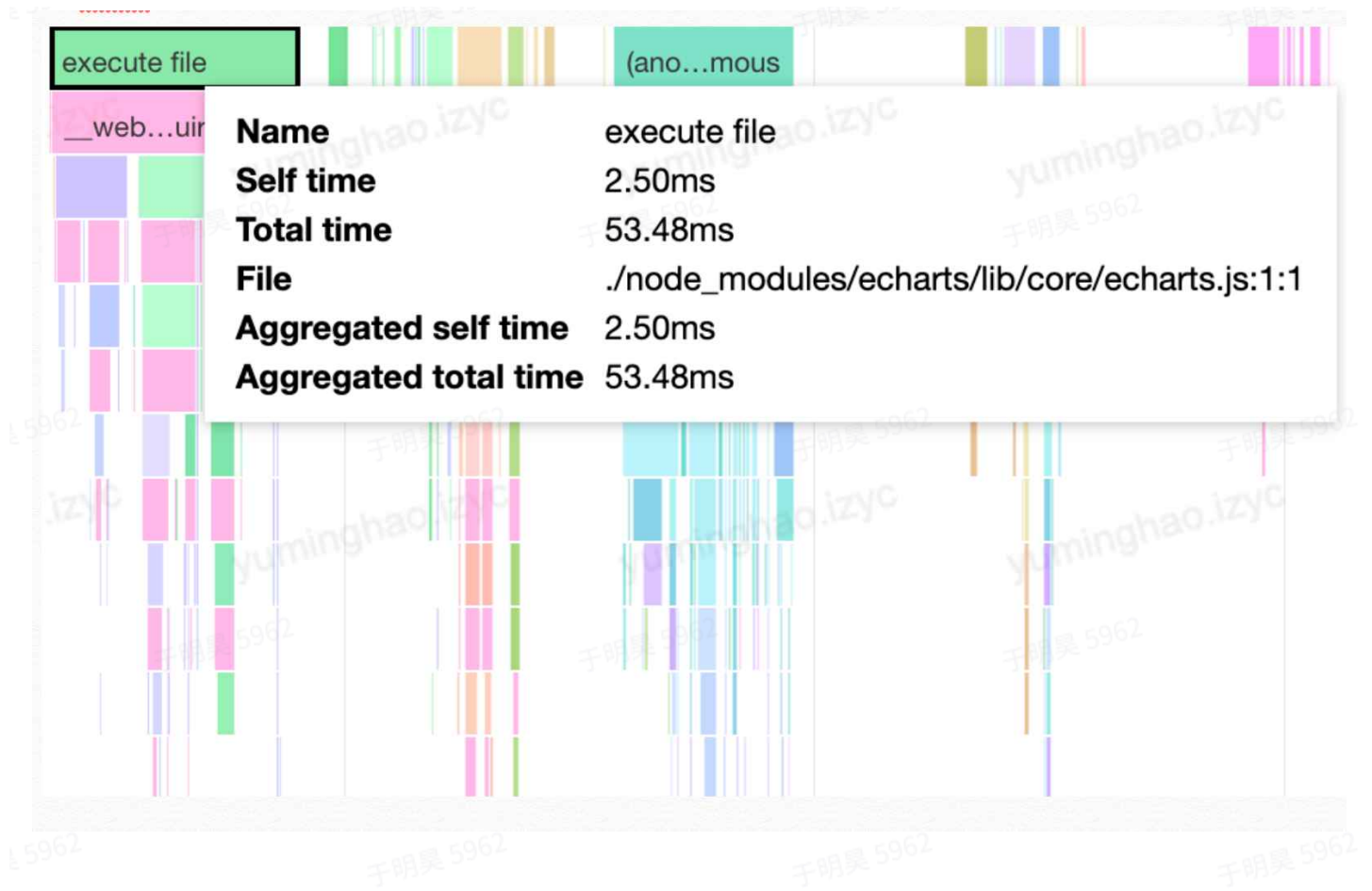
解：业务测无计可施，只能推动bridge团队解决。



4. Echarts (57.96ms)

分析：可以看到echarts有很大的耗时，因为echarts制图需要消耗系统资源。但是echarts制图和页面渲染在排行榜页面无直接关系，我们可将制图逻辑置于页面渲染后。

解：使用dynamic import，懒加载echarts。在用户点击“销量趋势”后再启动制图；按需引入echarts包，减少内部执行时间，加速请求。



效果展示

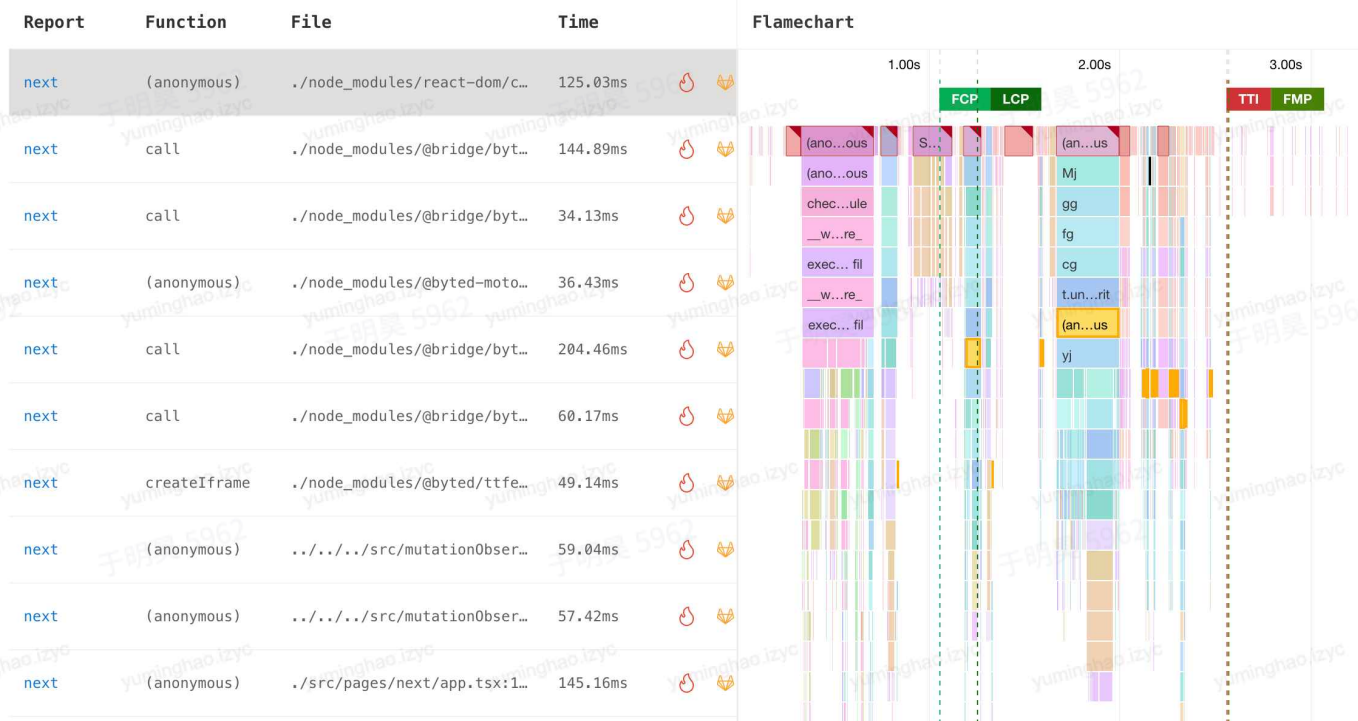
优化前

<https://maiev.bytedance.net/projects/3608/source?issueId=357026&reportId=1219260&version=1.0.0.628>

Projects > motor/fe/autorank ★ > Source

version: 1.0.0.628

Install Vscode E



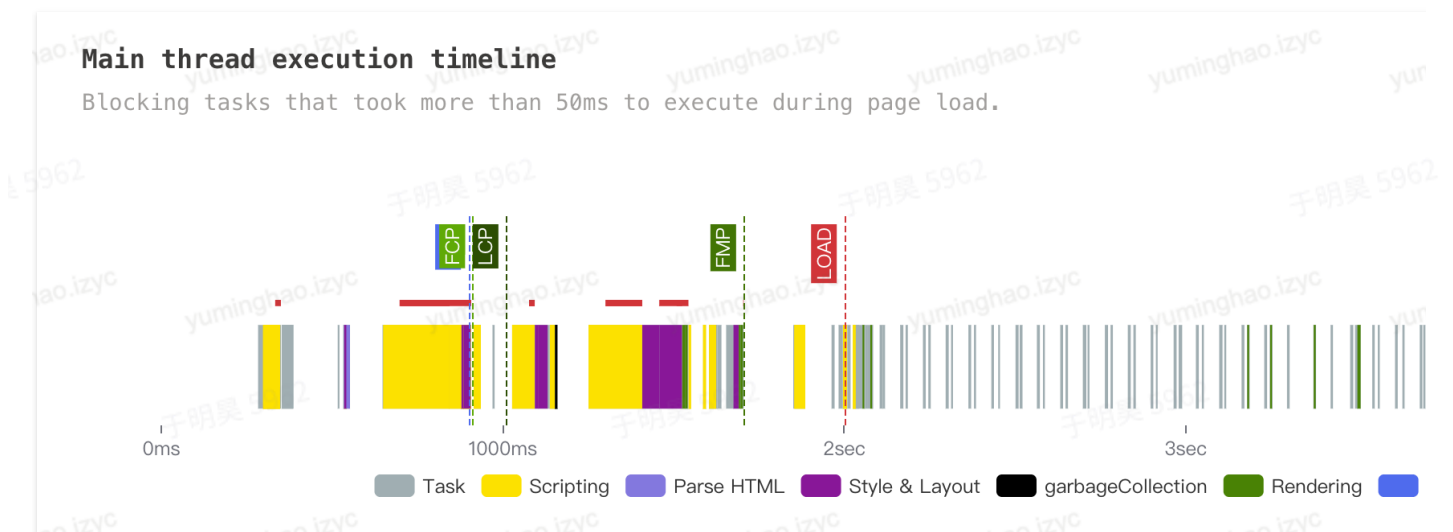
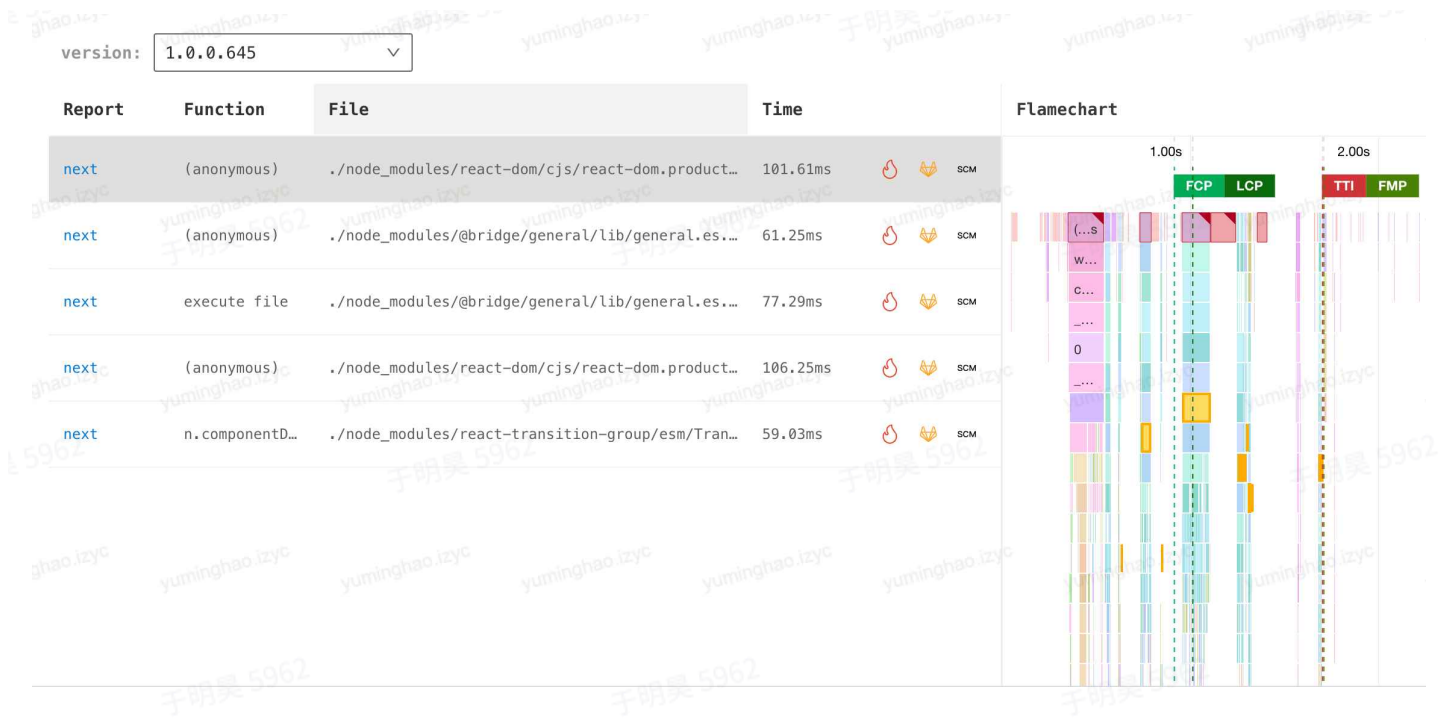
Main thread execution timeline

Blocking tasks that took more than 50ms to execute during page load.



优化后

<https://maiev.bytedance.net/projects/3608/source?issueId=362501&reportId=1232538&version=1.0.0.645>



效果对比：

整体看可以看到优化效果还是十分明显的。首先长任务已经只剩react-dom相关的任务和@bridge/general。其次FCP、LCP、FMP均有数百毫秒的提升。

代码展示

1. Webpack config

```
JavaScript

1 // 基于webpack4
2 // 使用webpack-chain
3 // 为节约空间，只展示关键部分。
```

```

4 // 以下配置 保证内容不变则文件名不变, 可复用缓存
5 config
6   .output
7   .path(relative('../dist'))
8   .filename('js/[name].[contenthash].js')
9 config.optimization.runtimeChunk('single')
10 config
11   .plugin('hash')
12   .use(webpack.HashedModuleIdsPlugin) // webpack5不需要该插件 而应设置 moduleIds:
    'deterministic'.
13   .end()
14
15 // 以下配置保证node_modules中的不同包单独打包。
16 config.optimization.splitChunks({
17   chunks: 'all',
18   maxInitialRequests: Infinity,
19   minSize: 0,
20   cacheGroups: {
21     vendor: {
22       test(module) {
23         return (
24           /[\\/]node_modules[\\/]/.test(module.resource) &&
25           !/@byted-motor/.test(module.resource)
26         )
27       },
28       name(module) {
29         // get the name. E.g. node_modules/packageName/not/this/part.js
30         // or node_modules/packageName
31         const packageName = module.context.match(
32           /[\\/]node_modules[\\/](.*?)([\\/]|$)/
33         )[1]
34
35         // npm package names are URL-safe, but some servers don't like @
36         symbols
37         return `npm.${packageName.replace('@', '')}`
38       },
39     },
40     bytedMotors: {
41       test(module) {
42         return (
43           /[\\/]node_modules[\\/]/.test(module.resource) &&
44           @byted-motor/.test(module.resource)
45         )
46       },
47       name(module) {

```



```

48     const packageName = module.context.match(
49         /[\\/]node_modules[\\/]@byted-motor[\\/](.*?)([\\/]|$)/
50     )[1]
51     return `npm.byted-motor.${packageName.replace('@', '')}`
52 },
53 },
54 },
55 })
56
57 // 以下配置保证 需要prefetch的图片（预先置于images/prefetch 文件夹下）文件名带上
    prefetch标识
58 config.module
59     .rule('image')
60     .test(/\. (png|jpe?g|gif|svg|webp)$/i)
61     .exclude
62     .add(path.resolve(__dirname, '../src/assets/images/prefetch/'))
63     .end()
64     .use('url')
65     .loader('url-loader')
66     .options({limit: false})
67 config.module
68     .rule('prefetch-image')
69     .test(/\. (png|jpe?g|gif|svg|webp)$/i)
70     .include.add(path.resolve(__dirname, '../src/assets/images/prefetch/'))
71     .end()
72     .use('file')
73     .loader('file-loader')
74     // 这里使用file-loader 是为了用其name能力。
75     .options({
76         name: 'images/prefetch.[hash].[ext]',
77     })

```

2. Prefetch & preload

其中Assets是在webpack node API中暴露的

JavaScript

```
1 // build.js
2 let assets
3 webpack(conf, (err, stats) => {assets = stats.toJson})
4
5 // index.pug
6 // preload css, fonts.
7 mixin preload(modernAssets, legacyAssets, modernBuild)
8   each item in ((modernBuild ? modernAssets : legacyAssets) || [])
9     if item.endsWith('.css')
10       link(as='style' rel='preload' href=item)/
11     if item.endsWith('.woff2')
12       link(as='font' type="font/woff2" rel='preload' crossorigin href=item)/
13     if item.endsWith('.woff')
14       link(as='font' type='font/woff' rel='preload' crossorigin href=item)/
15
16 // prefetch those shown after toggling tabs.
17 mixin prefetch(modernAssets, legacyAssets, modernBuild)
18   each item in ((modernBuild ? modernAssets : legacyAssets) || [])
19     if item.indexOf('images/prefetch') > -1
20       link(as='image' rel='prefetch' href=item)/
21     if item.endsWith('.js')
22       link(as='script' rel='prefetch' crossorigin='anonymous' href=item)/
```

3. Lazy-load echarts

其中用到了/*webpackPrefetch: true*/来进一步优化加载速度

JavaScript

```
1 import React, { Suspense, lazy } from 'react'
2
3 const CardChartsPromise = import(/* webpackPrefetch: true */ './card-charts')
4 const CardCharts = lazy(() => CardChartsPromise)
5 // <Emprt /> is whatever fallback you want.
6 return <Suspense fallback={<Emprt />}>
7   <CardCharts
8     seriesID={seriesId}
9     tabIndex={tabIndex}
10    rankDataType={rankDataType}
11    rankModel={rankModel}
12    month={month}
13    setTitle={setTitle}
14  />
15 </Suspense>
```

4. 按需引入echarts包

JavaScript

```
1 import ReactECharts from 'echarts-for-react/esm/core'
2 import { LineChart, BarChart } from 'echarts/charts'
3 import * as echarts from 'echarts/core'
4
5 import {
6   GridComponent,
7   TooltipComponent,
8   TitleComponent,
9 } from 'echarts/components'
10 import {
11   CanvasRenderer,
12 } from 'echarts/renderers'
13
14 // Register the required components
15 echarts.use([
16   TitleComponent,
17   TooltipComponent,
18   GridComponent,
19   LineChart,
20   BarChart,
21   CanvasRenderer,
22 ])
```

5. 按需引入polyfill

JavaScript

```
1 // 原来:
2
3 import 'intersection-observer'
4 import 'unfetch/polyfill'
5
6 // 修改后
7
8 // common-packages.ts
9 async function polyfillAsync() {
10   if (!('fetch' in window)) {
11     await import('unfetch/polyfill')
12   }
13   if (!('IntersectionObserver' in window)) {
14     await import('intersection-observer')
15   }
16 }
17 export default polyfillAsync
18
19 // index.ts
20 import polyfillAsync from 'Utils/common-packages'
21 polyfillAsync().finally(() => render())
```

6. Mui-tabs optimization

对activeIndex为0的tab 不需要滚动 因而不需要计算位置。

```
90 +   const activeIndex = tabChildren.findIndex((e, i) => {
91 +     if (isValidElement(e)) return (e.props.value || i) === value
92 +     return e === value
93 +   })
94 +   if (scrollStrategy === 'edge' && activeIndex > 0) {
95     horizontalScrollIntoView(
96       containerEl,
97       currentSelectedChild,
98       !scrollSmoothly,
99       scrollDistance
100     )
101   } else if (scrollStrategy === 'center') {
```

7. Wil

代码不贴了。为了说明问题，这里改成不再使用Wil，改用组件库@byted-motor/tea包发埋点，用useinview去实现show埋点。从代码易用性的角度考虑，实际项目不准备弃用wil。

8. Npm dedupe

JavaScript

```
1 // 可在 Maiev Bundle中直观地找到包冗余。先运行
2 npm dedupe
3
4 // 若没解决冲突 则出现上面说的那种情况：“发现一npm奇怪的特性会导致dedupe命令失败（原理
  待查明）：npm受安装顺序影响，有时理论上可以被deduped的dependency graph，却没有成功。解
  法：确保node_modules根目录下的该包版本高于其他包所依赖的该包版本”
5 // 可按上述尝试解决
6
7 // 另：听说用pnpm可避免包冗余的问题
```

另外，前端数据大多通过请求后端接口而来。提早请求时机以更早的拿到数据并渲染，对于用户体验是有帮助的。排行榜页面采用缓存配置项的做法（请求数据的入参依赖于配置项，而配置项优化前也是走后端接口的），来提早数据的请求时机。代码靠近业务逻辑，这里也不贴了。