

OSPP 2022 项目申请书

Apache APISIX Dashboard V3

项目摘要

Apache APISIX 即将迎来 V3 版本，而 Apache APISIX Dashboard 也需要对 V3 版本进行适配，借此机会，需要将 Dashboard 的前端部分进行重构，以解决历史问题，并带来更好的体验。

提案人：房森 wirth.fang@foxmail.com

导师：杨陶 iskyex@outlook.com

项目摘要

项目概览

产出介绍

提案人做过的同类产品原型

简单讨论原型的实现（登陆/页面）

登陆页面实现

内容页面实现

原型项目目录

项目目标及技术实现梳理

如何增加状态管理

如何增加e2e测试覆盖

如何对Dashboard性能进行优化[9]

如何解决有关前端issue

开发进度表

社区联络期（June 15 - June 30）

编码阶段一（July 1 - July 15）

编码阶段二（July 15 - Aug 30）

编码阶段三（Sep 1 - Sep 23）

缓冲时间（7 days）

关于我

申请信息

Self-introduction

参考资料

项目概览

产出介绍

使用 TypeScript 作为主要的编程语言


- ☐ 梳理项目结构，制定合理的重构计划
- ☐ 完成重构方案中路由与页面的重构工作：
 - ☐ 增加状态管理
 - ☐ 增加e2e测试覆盖
 - ☐ 对Dashboard性能进行优化
 - ☐ 以ts代替有关js代码段
- ☐ 解决项目中现存的 issue
- ☐ 以ts代替有关js代码段

提案人做过的同类产品原型

我对更新Dashboard V3的经验来源于我之前在字节跳动青训营时作为组长完成的基于React.js的后台管理系统（[FangSen9000/team1730: 字节青训营第二届元气满满小白队项目 \(github.com\)](https://github.com/FangSen9000/team1730)）它与 APISIX Dashboard十分相似，我从中掌握熟练了JavaScript/TypeScript, HTML, CSS, React的使用，我也有Docker, APISIX的背景知识。完全符合项目所需知识需要。

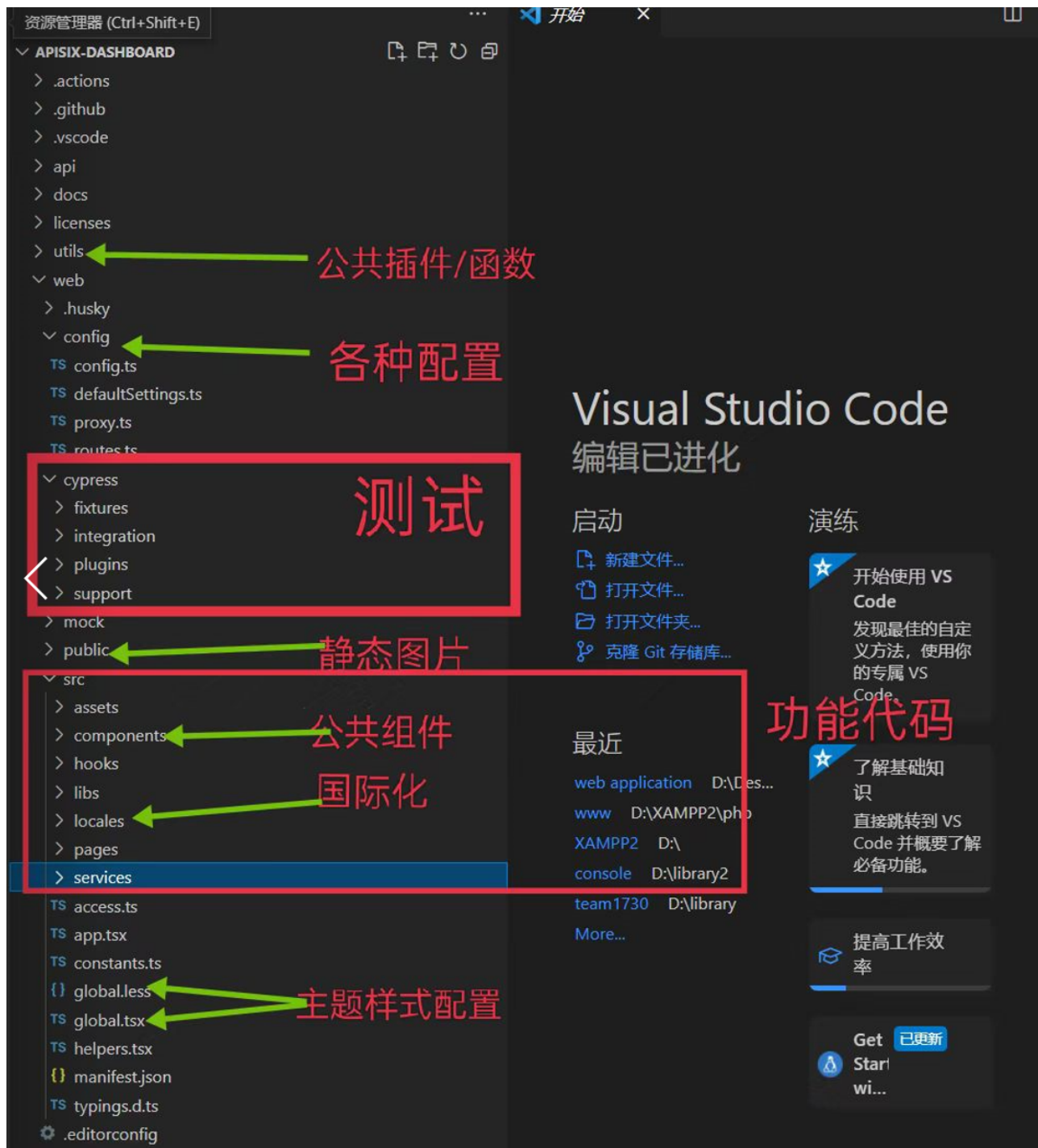
我使用过Antdpro V4和Antdpro V5，对他们web结构变化有经验。他们的文件结构和内容也很相似，所以我如鱼得水。下图是Apache APISIX Dashboard的登录页面和部分page和我写的火山引擎后台管理系统登录页面和部分page（它是以AntdPro V5为基础的React.js后台管理系统，且用TypeScript所写，我完成了三个以上的page和商品模块的jsx, utils, component, service文件的编写，对以ts 重构 APISIX Dashboard有充分的信心和完美的匹配度。

简单讨论原型的实现（登陆/页面）

 附件:简单讨论类似Dashboard项目原型的实现（登陆/页面）

项目目标及技术实现梳理

如果要对一个前端面板进行二次开发，首先要做的是读懂他的源码，我使用Docker克隆了Dashboard的 Git 存储库创建好了环境，对Apache Dashboard的文件目录理解如下图：



在分析了Dashboard代码之后，我初步制定了五个目标：

- ☐ 增加状态管理
- ☐ 增加e2e测试覆盖

- ☐ 对Dashboard性能进行优化
- ☐ 以ts代替有关js代码段
- ☐ 解决有关issue

如何增加状态管理

状态管理分为服务端和浏览器端，它将浏览器与web服务器之间多次交互当做一个整体来处理，并且将多次交互所产生的数据(即状态)保存下来。[3]

- 方式一 将状态保存在浏览器端，通常使用Cookie技术。
- 方式二 将状态保存在服务器端，通常使用Session技术。

根据我的分析，src/components中有很多嵌套，尤其是plugin文件夹中的文件，这种情况不利于Dashboard的长远发展，不利于未来的维护。因此，我打算使用最流行的状态管理库如Redux或原生的Context来为Dashboard添加状态管理：

Redux介绍

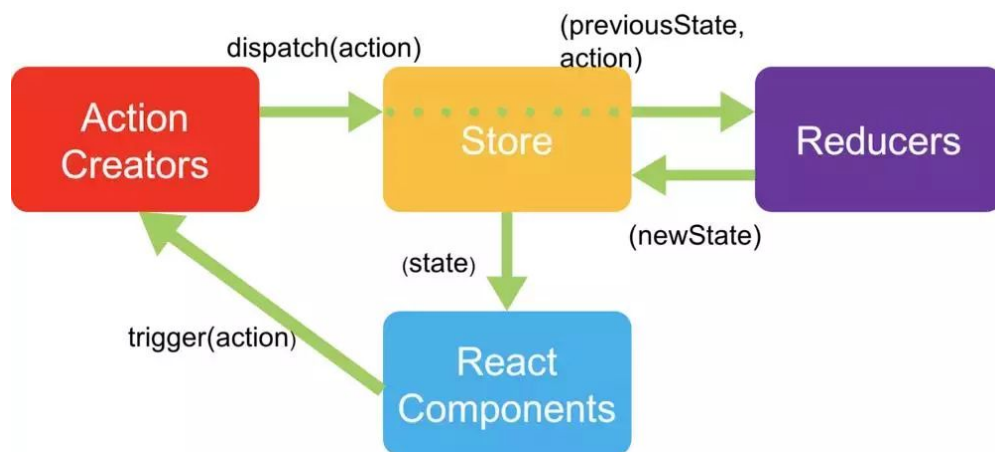
Redux 是整个 React 生态系统中最早，最成功的状态管理库之一。我已经在许多项目中使用过 Redux，如今它依然很强大。开始使用Redux前，先安装依赖：

```
npm install @reduxjs-toolkit react-redux
```

要使用 Redux，您需要创建和配置以下内容：

1. A store
2. Reducers
3. A provider

为了帮助解释所有这些工作原理，我在实现 Redux 中的 Notes app 的代码中做了注释，具体可以看这个Demo：[FangSen9000/demo-dashboard-Redux](https://github.com/FangSen9000/demo-dashboard-Redux)



Context介绍

要创建和使用 context ，请直接从React导入钩子。它具备Redux的状态管理架构，而无需大量的配置代码，也无需外部依赖。

React发布了Context作为内置功能，允许我们创建全局的状态。只要几行就可以配置成功。使用useReducer hook，可以模拟redux的模式，使用Context可以访问程序中任何位置的全局状态，只需要在根节点使用provider包裹即可。下面是它的工作原理：

```
/* 1. Import the context hooks */
import React, { useState, createContext, useContext } from 'react';

/* 2. Create a piece of context */
const NotesContext = createContext();

/* 3. Set the context using a provider */
<NotesContext.Provider value={{ notes: ['note1', 'note2'] }}>
  <App />
</NotesContext.Provider>

/* 4. Use the context */
const { notes } = useContext(NotesContext);
```

完整代码可以看这个Demo: [FangSen9000/demo-dashboard-Context \(github.com\)](https://github.com/FangSen9000/demo-dashboard-Context)

Mobx介绍

XState 试图解决现代UI复杂性的问题，并且依赖于有限状态机的思想和实现。XState 是由 David Khouishid[10]，创建的，自发布以来，网上就有很多很多关于它的讨论。

在体验上，它的实现方式是与其他库截然不同的。它的复杂性比其他任何一种都要高，但是关于状态如何工作的思维模型确实很 cool 而且对于提高能力很有帮助，当我看过一些 demo app之后，让我思考了很多...总的来说它有以下优点：

1. mobx可以使用observable定义一些需要被观察的状态
2. 组件视图的观察可以引用mobx-react的observer, `import {observer} from 'mobx-react'`;
3. 当通过action或者computed改变被观察的数据后，组件中所引入这些数据的地方也会随着改变。
4. 一般可以使用类组件和装饰器来简化代码，提高开发效率。

XState介绍

MobX 具有可观察者和观察者的概念，然而可观察的API有所改变，那就是不必指定希望被观察的每个项，而是可以使用 `makeAutoObservable` 来为你处理所有事情。总的来说它有以下优点：

它主要将使用的数据存储在 `context` 中。在这里，我们有一个 `notes` 列表 和一个 `input` 输入框。有两种操作，一种用于创建 `note` (`CREATE_NOTE`)，另一种用于设置 `input` (`CHANGE`)。具体实现可以看官方的一个例子：

[FangSen9000/demo-dashboard-XState](https://github.com/FangSen9000/demo-dashboard-XState)

如何增加e2e测试覆盖

APISIX Dashboard采用的是Cypress测试框架[\[4\]](#)，它的文件目录如下：

```
- /cypress
  - /fixtures (mock 数据)
    - example.json
  - /integration (测试文件)
    - /examples (一般格式为 *.spec.js, 可支持 .jsx/.coffee/.cjsx)
  - /plugins (用于配置安装的 插件, task 系统)
    - index.js
  - /support (用于调整自定义选项)
    - commands.js
    - index.js
  - /screenshots (默认截屏文件夹)
```

解决将路由测试迁移到 e2enew [#2411](#)

对于这些任务，我想着手改进它，它只有一部分指标合格，覆盖范围还是有点小

```
@@          Coverage Diff          @@
##          master    #2411    +/-    ##
=====
+ Coverage   51.61%    71.26%    +19.64%
=====
Files        189        58       -131
Lines       7441       4020     -3421
Branches     828         0      -828
```

```
=====
```

- Hits	3841	2865	-976
+ Misses	3316	849	-2467
- Partial	284	306	+22

解决迁移旧的后端e2e测试[#2197](#)&&使用Ginkgo重写所有e2e测试[#1500](#)

当前项目中有两组不同的后端端到端代码，每组使用不同的实现，并且一些测试用例重复且令人困惑。我将会使用ginkgo将所有测试用例迁移到实现中，并统一其中的代码编写。最终目标是提高功能测试覆盖率，并修复测试过程中可能发生的各种问题，以提高一次性通过率。我会编写其中一组用例。[\[5\]](#)[\[6\]](#)

- 将旧的测试用例逐个迁移到ginkgo实现。
- 删除旧的测试用例。
- 组织测试用例并创建规范以指导应如何编写测试代码。
- 编写规范文档以供参考。

如何对Dashboard性能进行优化[\[9\]](#)

按需引入

我暂未发现Dashboard有相关操作，当我们在做react项目的时候，会用到antd之类的ui库，值得思考的一件事是，如果我们只是用到了antd中的个别组件，比如<Button />, 就要把整个样式库引进来，打包就会发现，体积因为引入了整个样式大了很多。我们可以通过.babelrc实现按需引入。

```
[ "import", {
  "libraryName":
    "antd",
  "libraryDirectory": "es",
  "style": true
} ]
```

路由懒加载，路由监听器

react路由懒加载，是由 dynamic异步加载组件总结出来的，针对大型项目有很多页面，在配置路由的时候，如果没有对路由进行处理，一次性会加载大量路由，这对页面初始化很不友好，会延长页面初始化时间，所以想着用asyncRouter来按需加载页面路由。

我发现Dashboard使用的仍是传统路由，我准备使用react路由懒加载（基于import 函数路由懒加载）众所周知，import 执行会返回一个Promise作为异步加载的手段。我们可以利用这点来实现react异步加载路由。具体可以看代码：[FangSen9000/demo-dashboard-Routes \(github.com\)](https://github.com/FangSen9000/demo-dashboard-Routes)

受控性组件颗粒化

它的目的是避免因自身的渲染更新或是副作用带来的全局重新渲染。可控性组件和非可控性的区别就是dom元素值是否与受到react数据状态state控制。一旦由react的state控制数据状态，比如input输入框的值，就会造成这样一个场景，为了使input值实时变化，会不断setState，就会不断触发render函数，如果父组件内容简单还好，如果父组件比较复杂，会造成牵一发而动全身。简单来说就是减少组件耦合程度。

可考虑的更多方法

事件节流和防抖，使用纯组件，使用 React.memo 进行组件记忆，使用 shouldComponentUpdate生命周期事件，不要使用内联函数，在 Constructor 的早期绑定函数，在 Constructor 的早期绑定函数，不要在 render 方法中导出数据，为组件创建错误边界等。

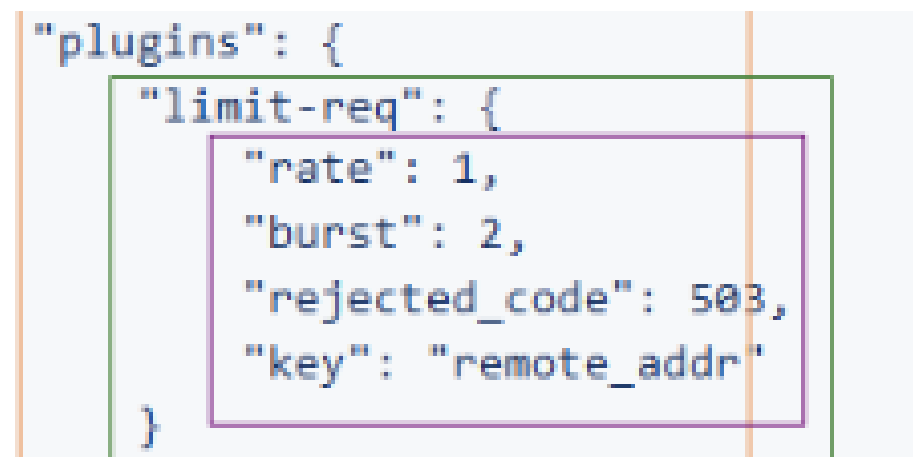
如何解决有关前端issue

解决仪表盘 Docker 实现使用优化[#2446](#)

使用多实例时每个实例生成并持有一个jwt令牌，将导致验证冲突，不符合apisix本身的高可用性解决方法：优化文档以强调仪表盘实例数量不超过1。[\[8\]](#)

为插件编辑器添加提示[#1363](#)

我们需要清楚地告诉用户应该填充数据的哪一部分。



解决方案，在相应模态框添加预览的相关提示内容。

开发进度表

社区联络期 (June 15 - June 30)

这个阶段的任务是深入了解项目，并可能在此过程中解决一些可能会干扰进度的因素与预案。对所需的技术做一些研究，与社区开发人员和导师交谈，并改变一些技术步骤或计划。

编码阶段一 (July 1 - July 15)

编码阶段一有2周的时间，并在阶段结束后进行评估计划质量与风险。第一阶段的目标是

- ☐ 梳理项目结构，制定合理的更细节的重构计划

编码阶段二 (July 15 - Aug 30)

编码第二阶段有6周时间，阶段结束后进行可能的中期评估，有三个初步目标，至于其执行先后顺序有待商酌。

- ☐ 增加状态管理
- ☐ 增加e2e测试覆盖
- ☐ 解决有关issue

编码阶段三 (Sep 1 - Sep 23)

编码第二阶段有3周时间，如果任务进行顺利，我们将修复问题（issue）和对Dashboard性能进行优化。我们注意到社区中的一些问题与前端有关，也就是我们的项目控制台。我们可以考虑完成它们。

- ☐ 以ts代替有关js代码段
- ☐ 对Dashboard性能进行优化

一旦实现了本个阶段的目标，我们会检查代码并修复前面代码中的任何已知bug。编写代码应该被文档化。然后我们的工作基本完成了，然后是最后的评估。

缓冲时间 (7 days)

有9天的缓冲时间，以防前几周发生的事情没有按计划进行。我认为一些工作可能会在两周内完成，所以我们实际上会进展得更快，并拥有更多的容错能力。

参考资料

- [1] [使用AntdPro创建基于React的管理后台：对最新版本antd V5.0后台管理鉴权/权限的研究](#)
- [2] [使用AntdPro创建基于React的管理后台中 < 火山引擎后台管理 > 商品模块的实现：代码结构简析](#)
- [3] [状态管理_学习笔记_滨海之君的博客-CSDN](#)
- [4] [为什么选择赛普拉斯？|赛普拉斯文档 \(cypress.io\)](#)
- [5] [Cypress简介_张弛Terry的博客-CSDN](#)
- [6] [Cypress End-to-End Testing视频分析-YouTube](#)
- [7] [JWT –生成和验证令牌–示例_dnc8371的博客-CSDN](#)
- [8] [React性能优化技巧 weixin 43844392的博客-CSDN](#)
- [9] [Umijs/umi-request: 网络请求库，基于 fetch 封装, 兼具 fetch 与 axios 的特点](#)
- [10]  附件：简单讨论类似Dashboard项目原型的实现（登陆/页面）
- [11] [附件：demo-dashboard: 这是我为APISIX Dashboard项目存放Demo的地方\(github.com\)](#)