基于 Vue 的现代组件库建设


本篇属于组件库建设的开篇，后续的文章视公司内部具体需要再进行扩展。

随着前端业务复杂度的不断提升，社区生态出现了众多如 Vue、React、Angular 等优秀的开源项目，这些项目也在无形中推进了前端组件化的进程。与此同时，面向组件的开发模式也渐渐成为前端的主流。本文将介绍从 0 到 1 搭建组件库的全过程。

你将获得：

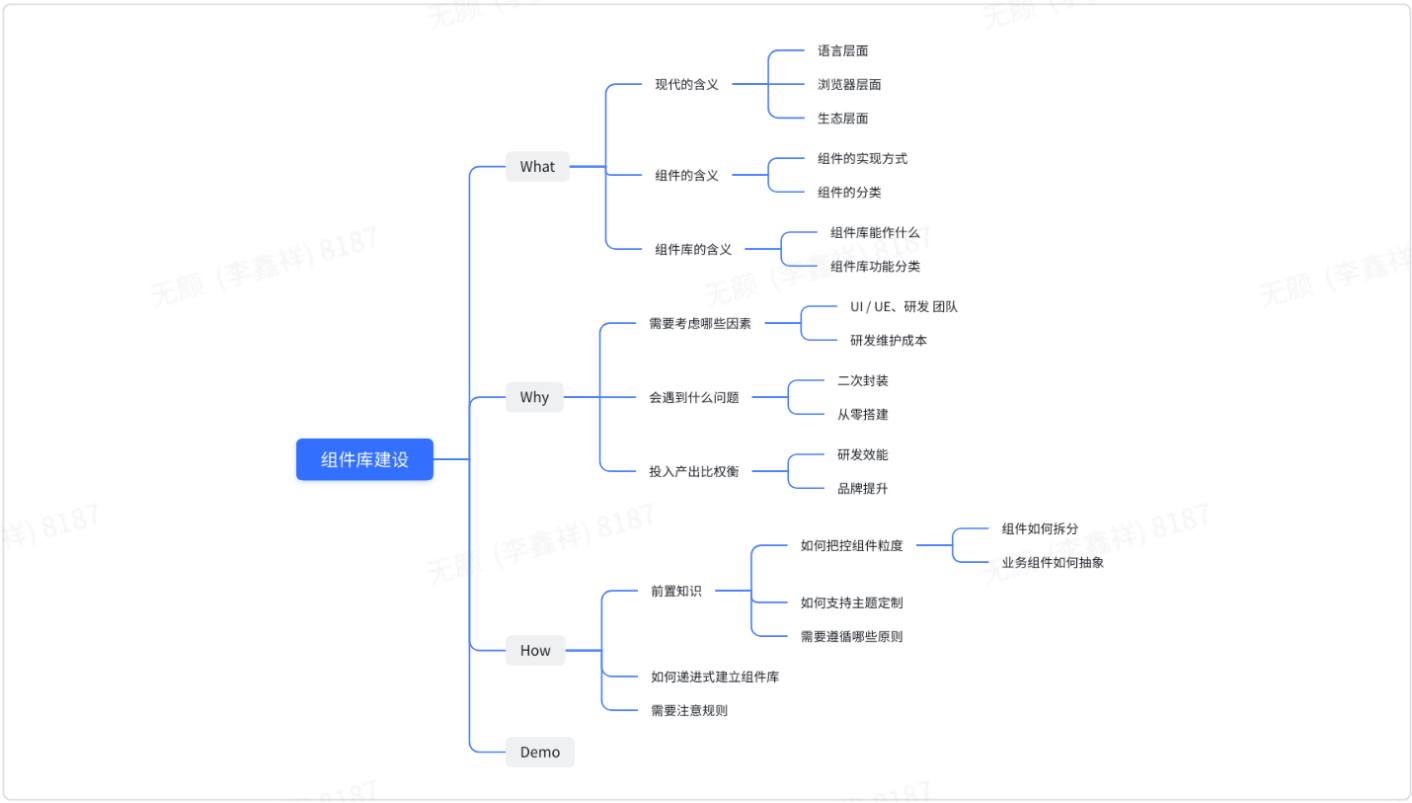
- 全新的前端知识体系（2021年）；
- 对前端组件有更清晰的认识；
- 从零搭建前端组件库的能力；
- 前端项目架构基础能力；

适合人群：

- 爱折腾；
- 想从零搭建前端组件库；
- 不限前端工程师；

注意：本文虽然讲述的是基于 Vue 的现代组件库建设全过程，但整体是从框架无关的角度去描述。并且，文中一些偏 UI 原则的内容，希望读者重点关注。

一、概览



二、什么是现代组件库

本节主要围绕现代组件库来进行说明，分别介绍 “现代” 这个关键词的含义、什么是组件、组件库，以及社区有哪些现成的组件库可供使用。

2.1 现代

> 使用新技术不是为了炫技，而是真的能解决问题，比如提高代码可维护、阅读性，开发效率等。

从 2015 年开始，前端的发展非常迅猛，技术生态也在不断更迭。本文标题中的现代泛指截止 2021 年为止，全新的前端技术生态圈。所谓现代也仅仅指当下，下一阶段的前端也可能发生更大的变化，我们只需要保持拥抱变化的心即可。

无论是从前端内容（HTML）、样式（CSS）、行为（JS、ECMAScript）相关标准的更新，还是各大浏览器厂商对新特性的支持，都给前端带来了与过去不同的研发体验。

下面从语言、生态两个层面提及有哪些新的标准与技术。

语言层面：

- HTML5 提供了更多的语义化标签、**2D/3D 绘图 & 效果** 的特性；
- CSS3 提供了更多的选择器、函数、动画等特性；
- ES 6、7 + 提供了更多数组新方法、可选链、空值合并运算符等特性；

生态层面：

- vite 下一代前端开发与构建工具；
- lerna + monorepo 多包管理方式；
- rollup JS 模块打包器；
- storybook 面向组件开发模式的文档工具；
- ...

此外，除组件库相关的技术需要保持关注外，像 ***Rust*** 、***WASM***、***Flutter*** 等都可以保持关注。

注意：如果对新标准与规范不太熟悉的同学，推荐大家通过 现代 JavaScript 教程、MDN 来重新学习前端相关知识。

2.2 组件

组件是一系列可重用的定制元素（对内容、样式、行为的封装），方便在 web 应用中使用它们。

实现组件化的方式：

- Web Components 一种原生支持方案；
- Vue 组件 基于框架的实现方式；

组件分类：

- 展示组件：纯 UI 组件，通常不包含业务逻辑，如业务接口请求；
- 容器组件：通常包含数据交互，如 Store 数据分发；
- 业务组件：对于业务组件的理解，因人而异，其中一种理解是：具体业务场景下的展示组件。另一种是，封装了具体业务逻辑（如接口请求）的组件。但从解耦的角度来说，需要复用的业务逻辑应封装在非组件级别里；（如：Vue 的 mixins 中、React 的自定义 hook 中）

2.3 组件库

组件库通常是由一系列展示组件构成，以 npm 包的形式供开发者使用，并给予文档支持。另外，组件库需要一整套完整的设计规范指导开发，并有完善的组件新增机制。

理想的组件库应具备以下内容：

- 设计指南与规范；
- 组件文档及使用示例；
- 主题定制工具；
- 原型、高保真视觉稿等设计资源；

从需求层面来说，组件库分为：

- 桌面端；
- 移动端；

组件功能分类：

- Layout 布局；
- Inputs 数据输入；
- Navigation 导航；
- Surfaces 表面展示；
- Feedback 用户反馈；
- Data Display 数据展示；

2.4 社区组件库

对大部分团队来说，其实没有必要从零开发组件库（开发及维护成本大），使用社区现有的组件库即可满足绝大部分业务场景。

以下为 Vue 社区比较优秀的组件库：

名称	优点	缺点	场景
Element UI	支持 vue3	组件定制难度大	PC
vuetifyjs	易于扩展，方便主题定制	国外 UI 风格	PC
vant	轻量、可靠	组件定制难度大	Mobile

对于社区组件库的选择，主要看团队的定制程度多大，但就目前来看，社区能选择的 Vue 组件比较少，且多数定制成本大（更多是简单的二次封装，根据业务场景组合组件），对中后台管理系统的来说，选择 Element UI 居多。移动端场景，选择根据业务来定，比如商城类产品可以选择 Vant。

三、为什么要建设现代组件库

对于一些有品牌定制需求的公司来说，从零搭建组件库的意义最大。否则重复造轮子就只是徒增开发和维护成本。故需要自建组件库的公司往往有非常多不同的 UI 和交互需求（多出现在移动端），同时有成熟的设计、交互、研发团队来支撑组件库的建设。

建设组件库需要考虑：

- 产品对 UI 是否有定制化需求；
- UI / UE 团队是否能支撑组件库建设；
- 研发团队是否支撑组件库的建设；
- 开发和维护成本；

注意：如果团队不满足以上条件，不建议从零开发组件库，但这不代表我们不需要了解如何搭建一个组件库，以及组件库的必备知识。（即使是基于社区现有的组件库二次封装）

3.1 问题

通常，产品对 UI 都是有定制化需求的，只是会出于研发成本的角度考虑，沿用社区较为优秀的资源（比如二次封装 Element UI）。其实对于任何一个产品来说，都应该有整体的设计、交互规范，所谓的设计规范先行就是这个道理。

二次封装造成的问题：

- 研发团队为满足设计要求，各种覆盖原有组件库样式，且不遵循任何规范；
- 设计团队不理解（不赞同）所使用的组件库的设计系统，导致整个设计系统被打破；
- 复合组件没遵循设计规范，全凭研发人员任意组合，甚至连组件的使用场景都是错的；
- 以上操作必然导致主题定制的复杂度增加，另外给主题的维护、组件库的升级带来了致命的问题；

自建组件库的问题：

- 设计侧规范对接、研发成本高；
- 组件库开发人员调配、维护成本高；

- 无法共享社区资源，需自行配套如原型、Sketch 资源来统一产品设计；

3.2 收益

团队足够大，目标足够明确。以上问题都能解决，由此带来的收益如下：

- 极大程度上满足各类业务、主题定制需求；
- 从源码级别把控每一个组件，及时响应需求；
- 团队成长，从组件的使用者变为开发者，更具备了主人翁意识；
- 团队梯队建设，人人都是组件库的贡献者，各自贡献自己的力量；

注意：组件库的建设必须在设计、交互规范的指导下进行。

四、如何进行现代组件库的建设

之前的章节主要介绍了什么是现代组件库、以及建设它的意义是什么。本章，重点会放在如何进行这块内容上。主体内容分为：前置知识，进行步骤、注意事项等。

4.1 前置知识

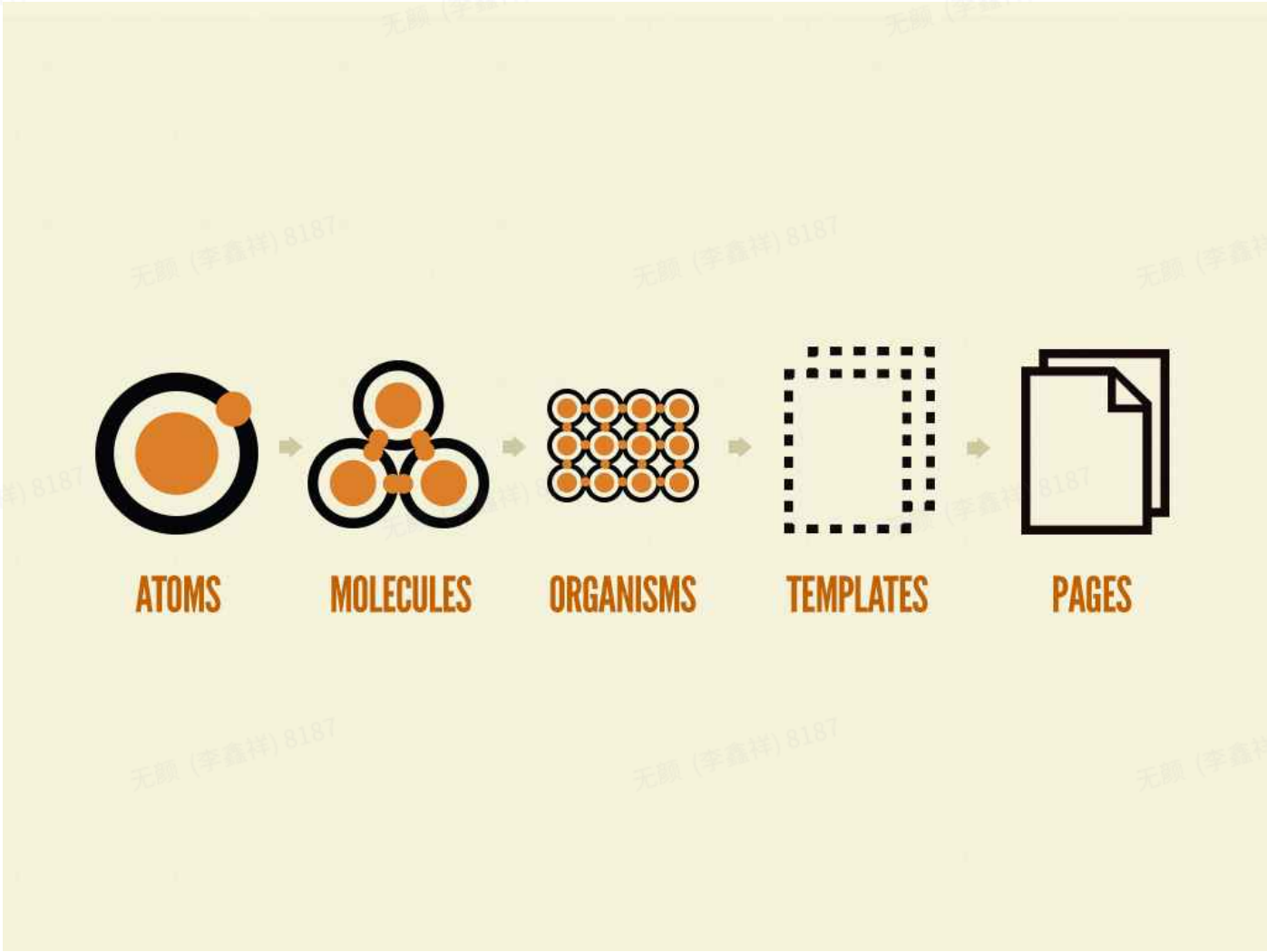
软件开发提倡提前设计，这样能避免开发过程中出现常规问题。此外，提前设计的目的在于从全局的角度去思考整个开发过程，尽可能多地覆盖更多场景以及可能出现的问题并列出解决方案。对于组件库的建设来说，很多同学可能没有想过下面提到的内容，但这些内容对于组件库的建设又是必备的，思考如下。

开发组件需要考虑：

- 如何把控组件粒度；
- 如何支持主题定制；
- 需要遵循哪些原则；

4.1.1 如何把控组件粒度

对于组件粒度，从比较宽泛的角度来说，可以遵循原子设计原则。这个原则理论上是设计师应该了解的，因为它属于网页设计的范畴，但作为合格的前端开发人员，应该与设计师在同一频道上。我们来简单了解一下，先来看一张图：



图中各部分的内容说明如下：

- **原子(Atoms)：**为页面构成的基本元素，例如标签、输入框、文字、颜色等；
- **分子(Molecules)：**由原子构成的简单UI元素，例如按钮；
- **组织(Organisms)：**相对分子而言，较为复杂的构成物，由原子及分子所组成；
- **模板(Templates)：**以页面为基础的架构，将以上元素进行排版；
- **页面(Pages)：**将实际内容（图片、文章等）放置在特定模板内；

也就是说，我们可以把组件按照这个原则进行拆分：原子组件、分子组件、组织组件，而模板、页面对于前端来说通常不属于组件级别。所以，对于网页元素级别的组件来说，遵循该原则就够了。并且业界对此也有比较完备的实践，比如 Material UI。

我对此的建议是：基础组件属于 atoms 级别（对现有网页元素的样式定制），业务组件属于 molecules 级别（对经常出现的业务场景的一种抽象）。在后续的内容中会进行详细说明，以及对应的考虑。

业务组件的开发相对于基础组件来说，难度在于组件的抽象，我们知道，抽象的目的在于代码的**分层**和**复用**，降低项目的复杂度。换句话说，就是我们需要沉淀到组件库的内容。

业务组件抽象的基本原则：

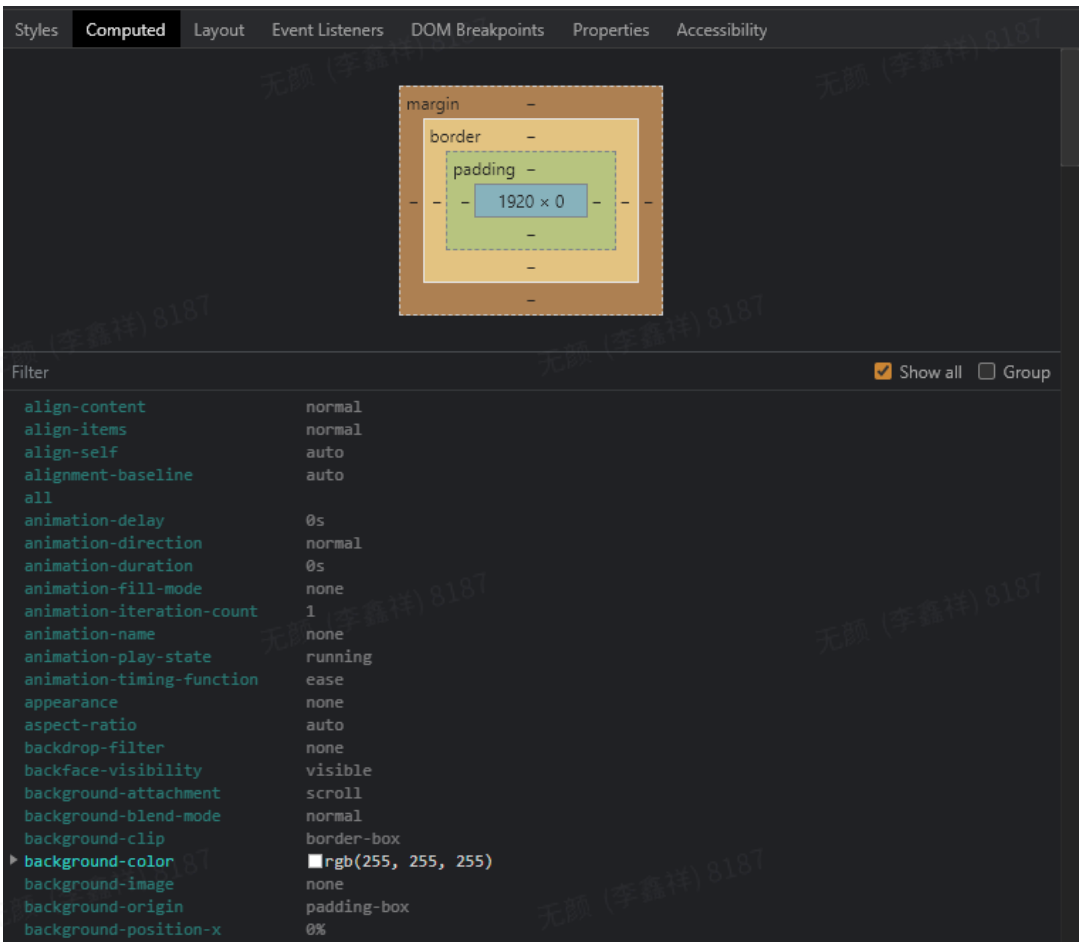
- 单一性：高内聚、低耦合；
- 复用性、通用性：适度、权衡可维护性；
- 职责分离：样式与业务分离；

4.1.2 如何支持主题定制

对于主题定制，首要的是与设计侧一起建立设计系统，设计系统一般包括如下内容：

- 颜色规范
- 文字规范
- 间距规范
- ...

对于前端开发人员来说，通常与 Chrome 控制台中，样式管理 (computed) 相匹配：



期间与设计侧约定的每个细节可以归结为 Design Token，它是设计工程化视角的产物，也是设计与前端之间共用的语言。(这一块的设计理念可参考 Material Design)

实际开发中，通常会与设计师共同约定如下 Design Token (以前端 less 中的变量为例)：

```less
@black: #000;
@white: #fff;
@gray-1: #f7f8fa;
@gray-2: #f2f3f5;
@gray-3: #ebedf0;
@gray-4: #dcdee0;
@gray-5: #c8c9cc;
@gray-6: #969799;
@gray-7: #646566;
@gray-8: #323233;
@red: #ee0a24;
@blue: #1989fa;
@orange: #ff976a;
@orange-dark: #ed6a0c;
@orange-light: #fffbe8;
@green: #07c160;

// Gradient Colors
@gradient-red: linear-gradient(to right, #ff6034, #ee0a24);
@gradient-orange: linear-gradient(to right, #ffd01e, #ff8917);

// Component Colors
@text-color: @gray-8;
@active-color: @gray-2;
@active-opacity: 0.7;
@disabled-opacity: 0.5;
@background-color: @gray-1;
@background-color-light: #fafafa;
```

最终会沉淀出设计与开发共通的 Design Token 资源，后续所有产品设计都是基于这些 Design Token ，后续的主题定制也是基于这个来做的。(多套主题，对应多套 Design Token 规范)

这里顺便提一下，在一些 css in js 的样式管理里面，styled-system 可以作为主题定制非常有力的工具。每一个设计粒度都可以控制的非常好，并且可以应用于所有的组件。(相对 less、scss 的样式解决方案来说， css in js 优势非常明显)

4.1.3 需要遵循哪些原则

对于组件库的开发人员来说，需要遵循如下原则：

- 组件样式中不存在固定值，所有变量均取值于设计侧规范
- 主题定制的粒度需与设计侧保持一致
- 确定一种样式覆盖机制，比如，组件库使用者可以复写 Design Token 改变原有样，或者全局替换 theme 文件来一次性改变主题

对于组件库使用者来说，需要遵循如下原则：

- 需要覆盖的样式均取值于原有组件库的 Design Token 变量

- 需要提前建立样式变量获取机制，编码中不存在样式的固定值
- 如非必要，不要随意打破原有组件库的设计规范，否则将无法全量定制主题

4.2 组件库实战

受限于篇幅，没办法把建设组件库所需要的方方面面都讲清楚。为了节省大家时间，下面直接带着大家一起来完成组件库的搭建。

4.2.1 设立目标

出于面向组件开发为最终目的，本次搭建的组件库应具备如下能力：

- 优质研发体验 (单仓多包管理)
- 标准研发流程 (各种 lint 标配 、代码风格统一工具)
- 交互式文档 (目标是文档自动生成)
- 主题定制
- 质量保证 (组件单测)
- ...

4.2.2 技术选型

对于组件库而言，可能会接触到的技术选型如下：

- 包管理：lerna + yarn
- 文档工具：Storybook
- 编程语言：JavaScript / TypeScript 4.x
- 构建工具：Vue-cli / Vite 2.x
- 前端框架：Vue 3.x
- CSS 预编译：Less / Stylus / Sass
- Git Hook 工具：husky + lint-staged / vue-cli git hook
- 代码规范：EditorConfig + Prettier + ESLint + Airbnb JavaScript Style Guide
- 提交规范：Commitizen + Commitlint + Angular 提交规范
- 版本规范：语义化版本 2.0.0
- 单元测试：vue-test-utils + jest + vue-jest + ts-jest
- 自动部署：GitHub Actions

根据团队自身的情况进行选型，并考虑如下原则：

- 技术栈统一
- 编码规范统一
- 设计规范统一
- 版本规范统一

4.2.3 具体步骤

有了目标以及基本原则和规范，接下来，我们一起来完成组件库的基本搭建。同样的，笔者不会把所有细节以代码的形式贴出来，这样做只会让整篇教程略显冗长。(如果希望达到更好的学习效果，可以结合示例代码来看)

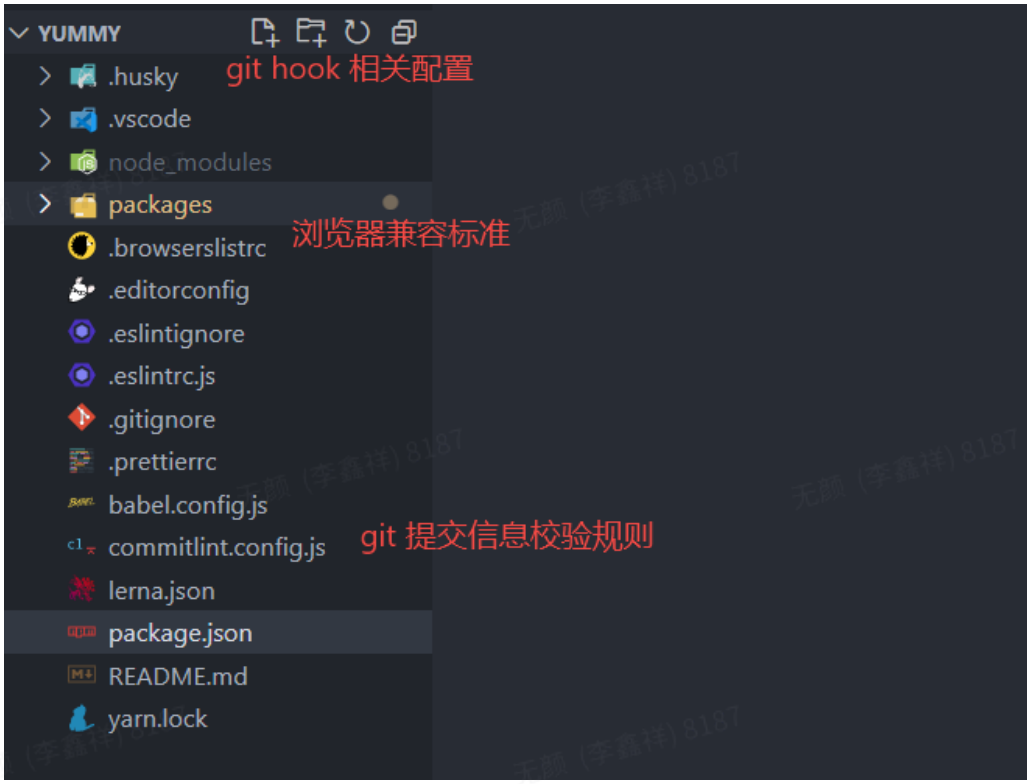
第一步，搭建规范机制 (假设你已经具备一定的项目搭建能力):

Bash

```
# 创建项目

mkdir yummy
cd yummy

# 初始化项目 ( 省略命令执行过程，仅将关键的命令列出 )

npm init # 执行 npm 初始化
lerna init # 执行 lerna 初始化

# 添加各类规范校验机制

vue add eslint # 基于 vue-cli 添加 eslint 参见: https://github.com/vuejs/vue-docs-zh-cn/blob/master/vue-cli-plugin-eslint/REA

yarn add -D husky # 通过 git hook 校验提交信息等
yarn add -D lint-staged # 仅检查 git 暂存内的文件，提高检查效率
yarn add -D prettier # 编码风格统一
yarn add -D commitizen # 标准化 commit 提交信息规范

...

```

这里仅仅讲述一个大概的过程，希望了解更多细节的同学可以通过技术选型中给出的官方文档，了解更多内容。下面直接给出一个具备规范机制标准的组件库有哪些内容。

项目结构：



由于每个团队的规范不相同，这里不展开 eslint、editorconfig、prettier、commitlint 等的配置。总结来说，组件库的规范机制应具备如下能力 (不限于组件库，任何前端项目都应具备)：

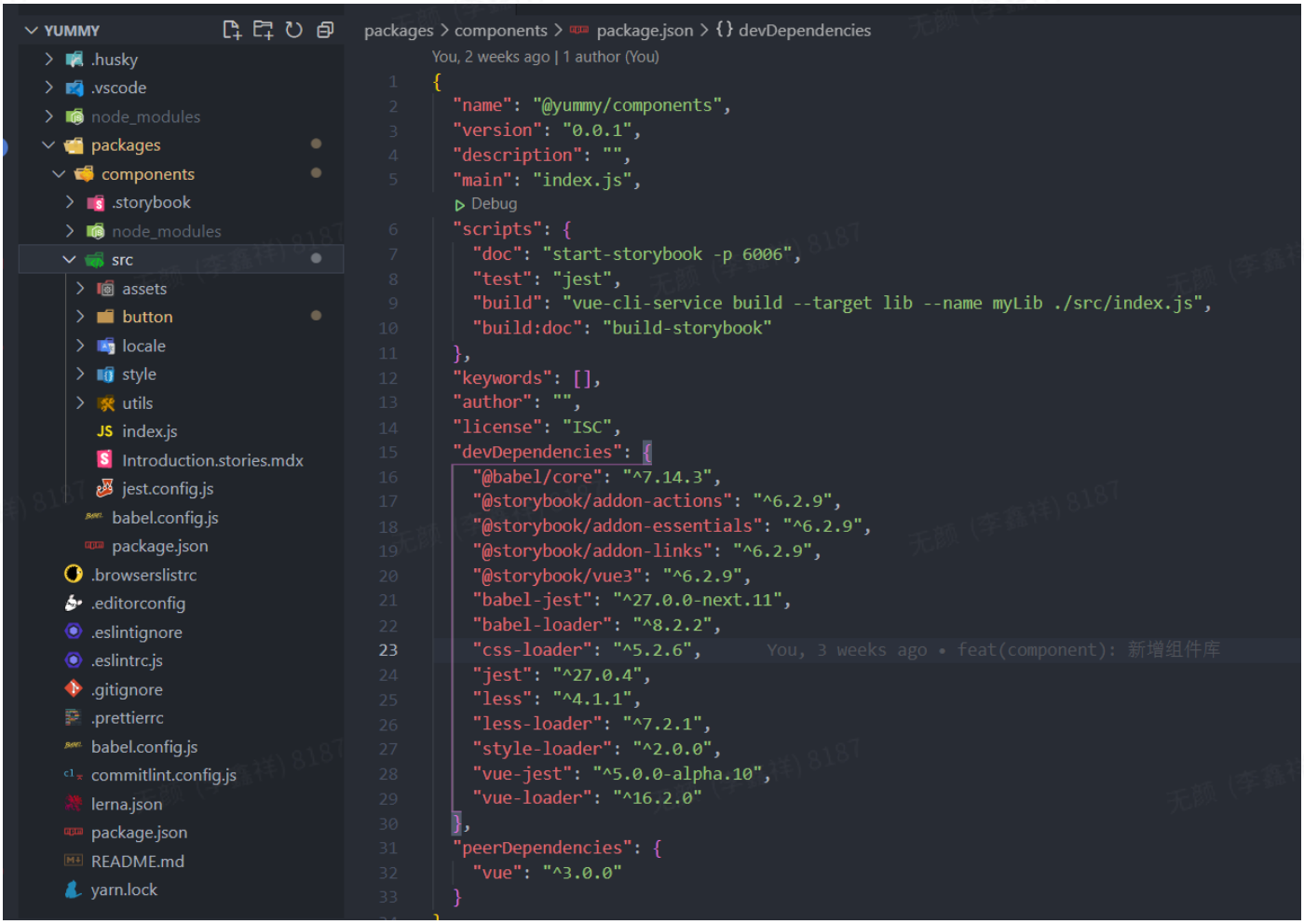
- 提交到 git 仓库的代码符合团队规范 (eslint、代码风格)
- Git 提交信息符合团队规范 (commit lint)
- 兼容目标一致(browserslist)

`package.json` 关键信息如下 (非完整结构，仅作为说明)：

JSON

```json
{
  "private": true, // [ lerna+yarn 配置 ] 防止将非 packages 下的包发布
  "workspaces": [ // [ lerna+yarn 配置 ] 指定对应多包目录
    "packages/*"
  ],
  "scripts": {
    "lint": "vue-cli-service lint **/src/**.{js,vue,jsx}", // 在根目录做所有 packages 的代码检测
    "prepare": "husky install", // husky 必须
    "commit": "cz", // commit 提交模板命令
  },
  "devDependencies": {
    "@commitlint/cli": "^12.1.4",
    "@commitlint/config-conventional": "^12.1.4",
    "@vue/cli-plugin-eslint": "~4.5.0",
    "@vue/cli-service": "~4.5.0",
    "@vue/compiler-sfc": "^3.0.0",
    "@vue/eslint-config-prettier": "^6.0.0", // 通常 eslint 校验应该与 prettier 格式化保持一致
    "commitizen": "^4.2.4",
    "cz-conventional-changelog": "^3.3.0", // 用于自动生成 changelog
    "eslint": "^6.7.2",
    "eslint-plugin-prettier": "^3.3.1", // 通常 eslint 校验应该与 prettier 格式化保持一致
    "eslint-plugin-vue": "^6.2.2",
    "husky": "^6.0.0",
    "lerna": "^4.0.0",
    "lint-staged": "^9.5.0",
    "prettier": "^2.2.1",
    "vue": "^3.0.0"
  },
  "peerDependencies": {
    "vue": "^3.0.0"
  },
  "config": {
    "commitizen": {
      "path": "cz-conventional-changelog"
    }
  }
}
```

第二步，将组件库作为单独的包进行维护 (下图为最终可能的目录结构):



首先，创建 `packages/components` 目录，下面把关键的步骤列一下

初始化项目：

Bash

```
# 初始化 npm
npm init
```

修改 `package.json` 为：

JSON

```json
{
  "name": "@yummy/components", // 多包管理中的项目名称
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "license": "ISC",
  "peerDependencies": {
    "vue": "^3.0.0" // 为统一，与根目录 vue 保持一致
  }
}
```

第二步，添加交互式文档 (storybook)：

Bash

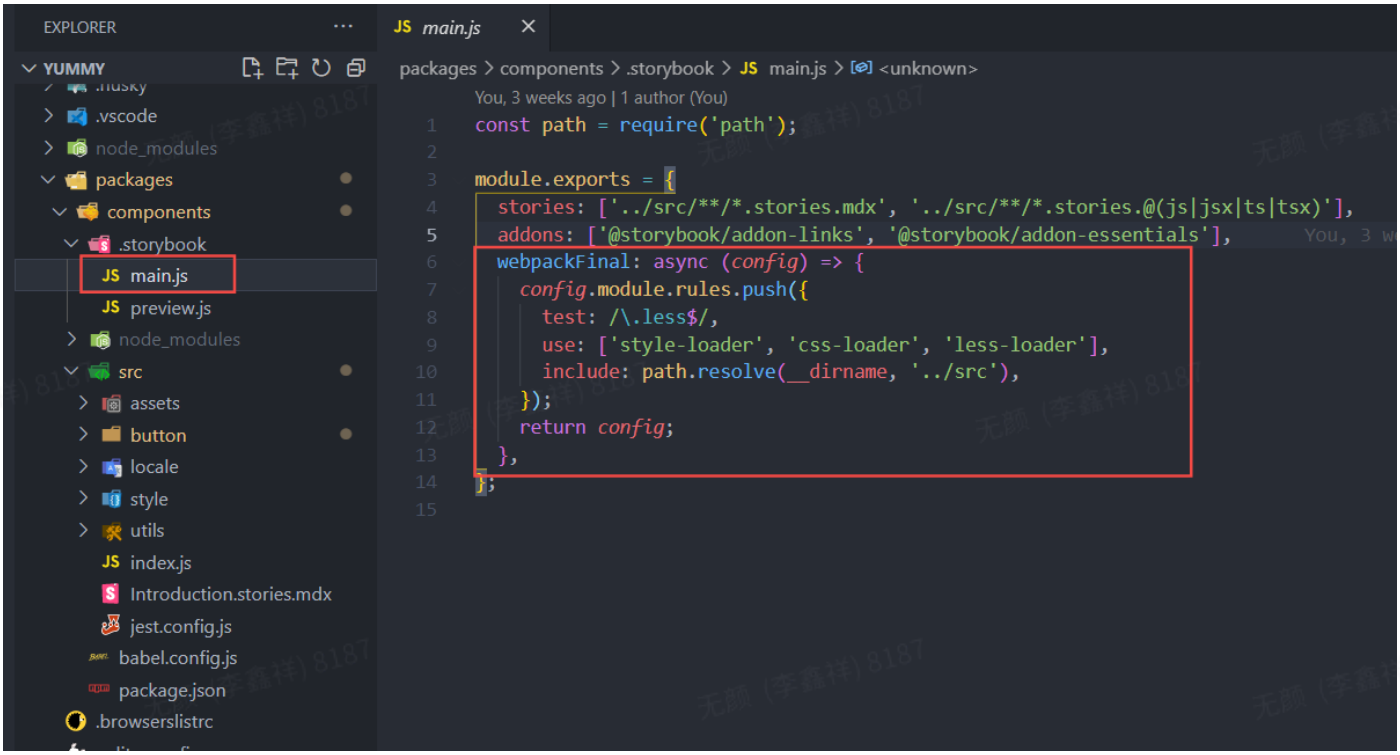
```bash
# 执行 storybook 初始化脚本，并选择 vue 模板：
npx sb init
```

`package.json` 新增修改如下：

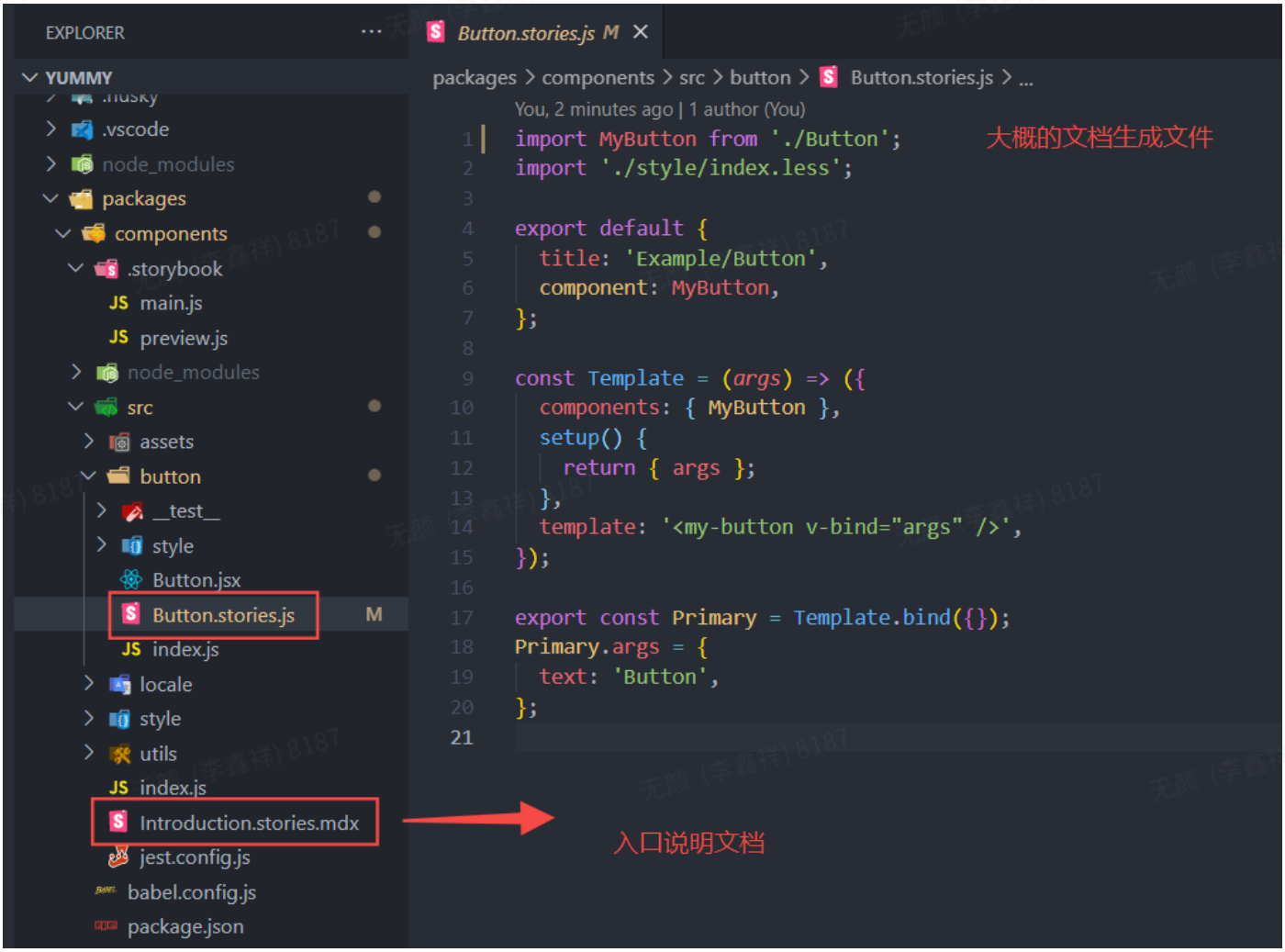
JSON

```json
{
  "scripts": {
    "doc": "start-storybook -p 6006", // 交互式文档启动命令
    "build:doc": "build-storybook" // 打包文档内容
  },
  "devDependencies": {
    "@babel/core": "^7.14.3", // 提供 babel
    "@storybook/addon-actions": "^6.2.9",
    "@storybook/addon-essentials": "^6.2.9",
    "@storybook/addon-links": "^6.2.9",
    "@storybook/vue3": "^6.2.9",
    "babel-loader": "^8.2.2", // 为支持 vue jsx 语法
    "css-loader": "^5.2.6",
    "less": "^4.1.1",
    "less-loader": "^7.2.1",
    "style-loader": "^2.0.0",
  },
}
```

组件库样式解决方案采用的是 less，由于 storybook 初始配置不支持 less，需要添加如下配置：

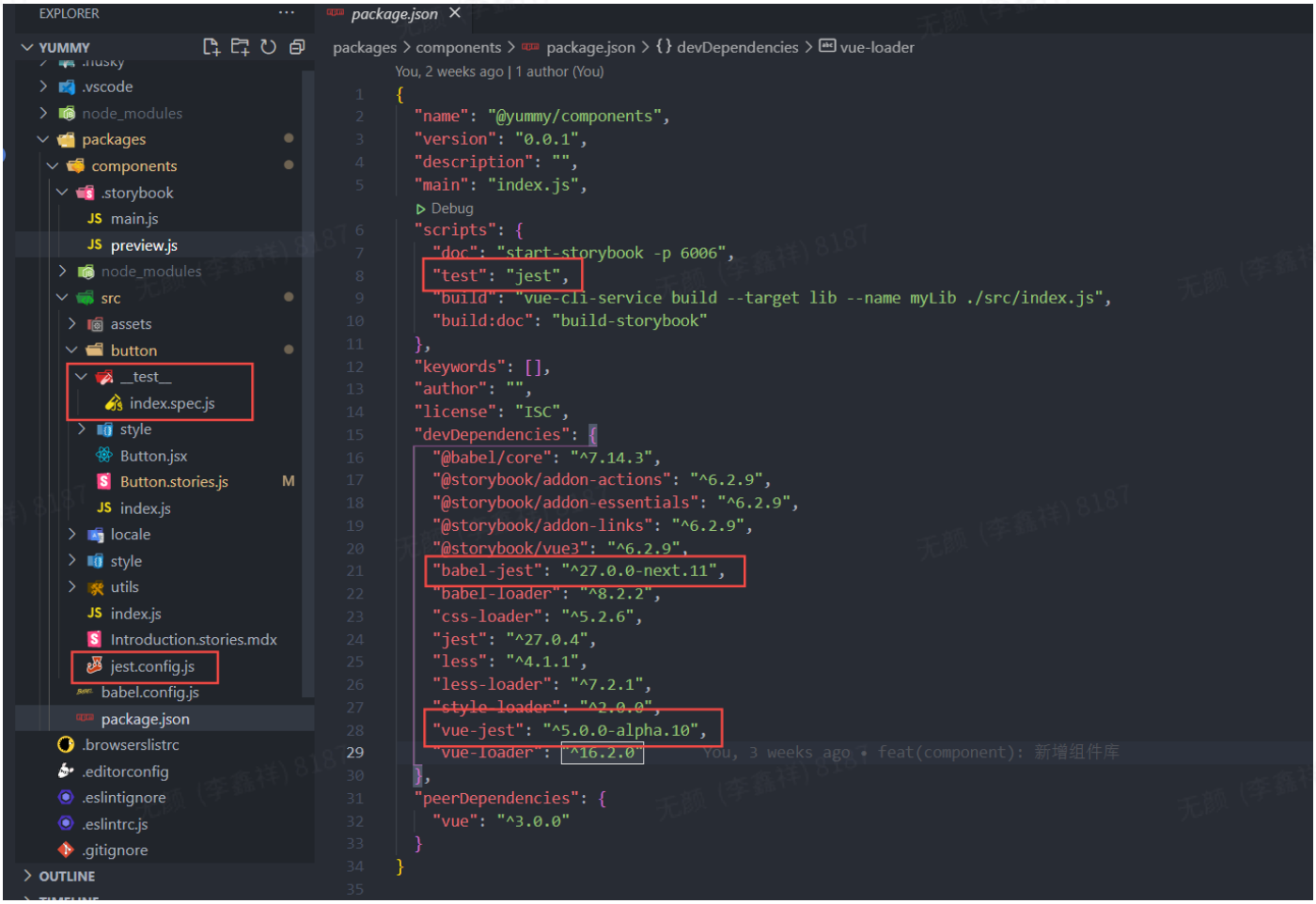


执行 `npx sb init` 命令后，生成的目录含有最基本的示例代码，最终修改后的结果：



第三部、添加单元测试：

同样，为突出重点，这里仅说明一下添加单元测试后的文件结构：



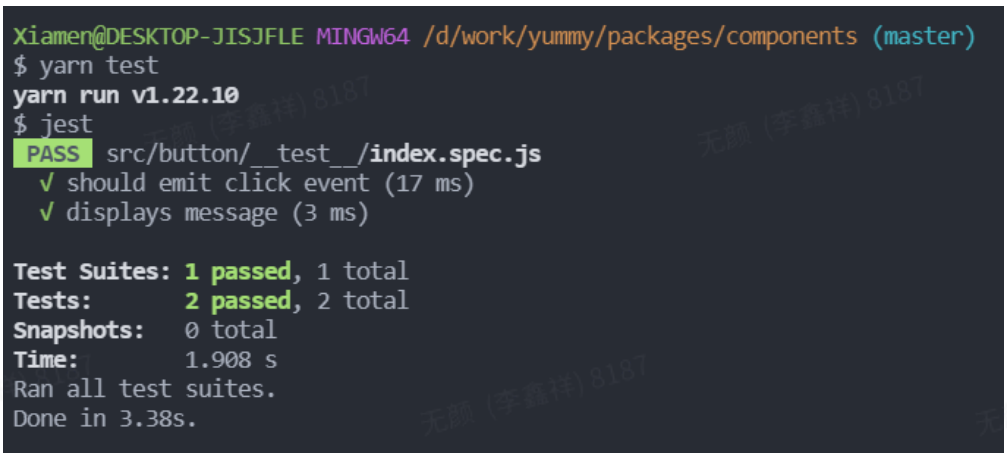
注意：核心依赖 `@vue/test-utils` 也是安装在根目录，目的是让这个单仓多包管理的项目最终依赖的核心包保持一致，如 vue 、vue-service-cli 等。

下面是一个简单的组件的测试示例：

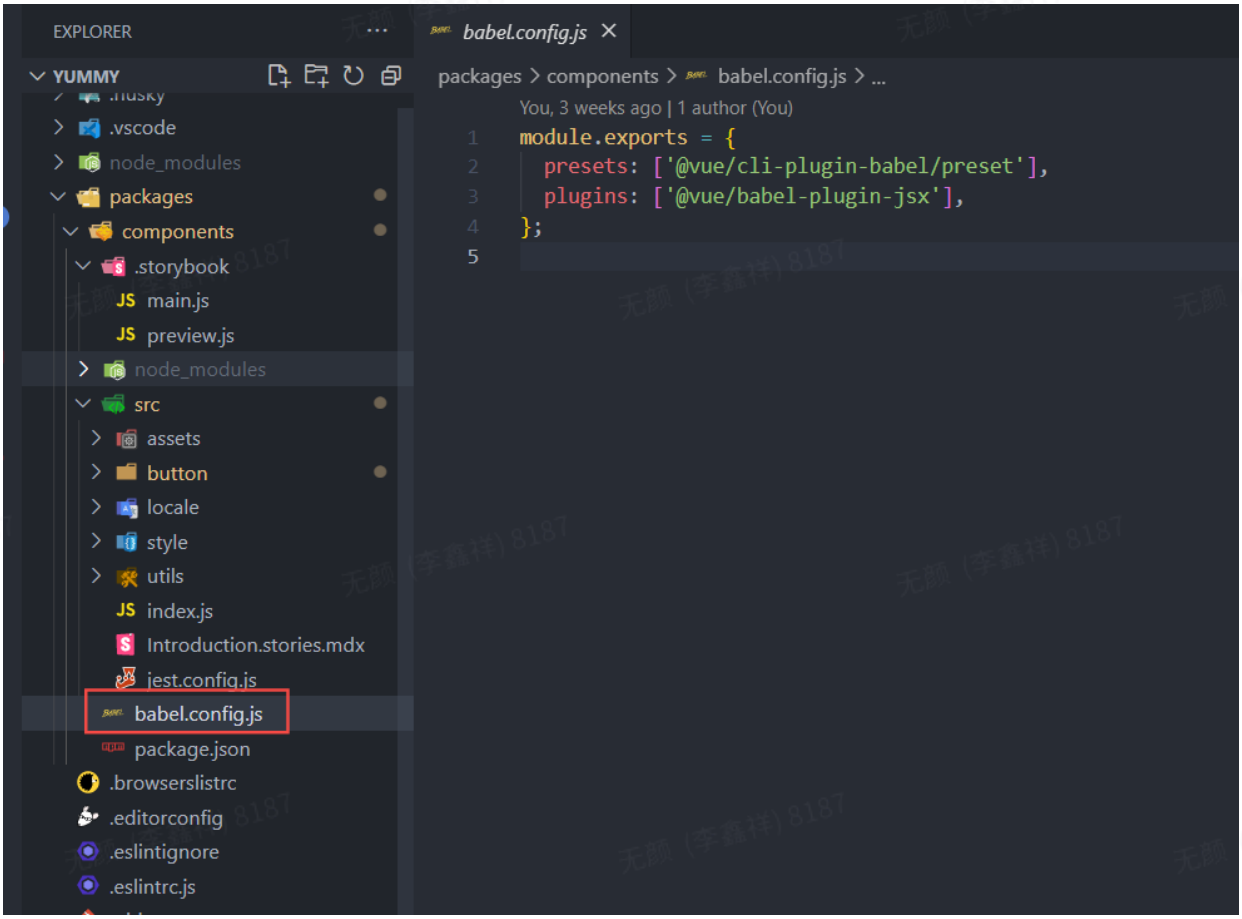
JavaScript

```javascript
/**
 * @jest-environment jsdom
 */

import { mount } from '@vue/test-utils';
import Button from '../Button';

test('should emit click event', () => {
  const wrapper = mount(Button);

  wrapper.trigger('click');
  expect(wrapper.emitted('click').length).toEqual(1);
});

test('displays message', () => {
  const wrapper = mount(Button, {
    propsData: {
      text: 'Hello world',
    },
  });
  expect(wrapper.text()).toContain('Hello world');
});
```

在组件库目录下执行 `yarn test` 的结果：



第四步，添加 JSX 语法支持：



建议是采用 `jsx` 的方式来编写组件，下面是一个示例：

JavaScript

```javascript
import { defineComponent } from 'vue';

export default defineComponent({
  name:'Button',
  props: {},
  emits: ['click'],

  setup(props, { emit}) {
    const renderText = () => {
      if (text) {
        return <span>{text}</span>;
      }
    };

    const onClick = (event) => {
        emit('click', event);
    };

    return () => {
      return (
       <div onClick={onClick} >{renderText()}</div>
      );
    };
  },
});
```

关于 Vue 3.0 组合式 API，再后续文章会结合实际需要给出。

注意：在根目录额外安装 `@vue/cli-plugin-babel/preset` 和 `@vue/babel-plugin-jsx` 用于支持 Vue 的语法。

第五步，添加 Design Token 实现主题定制

组件代码如下：

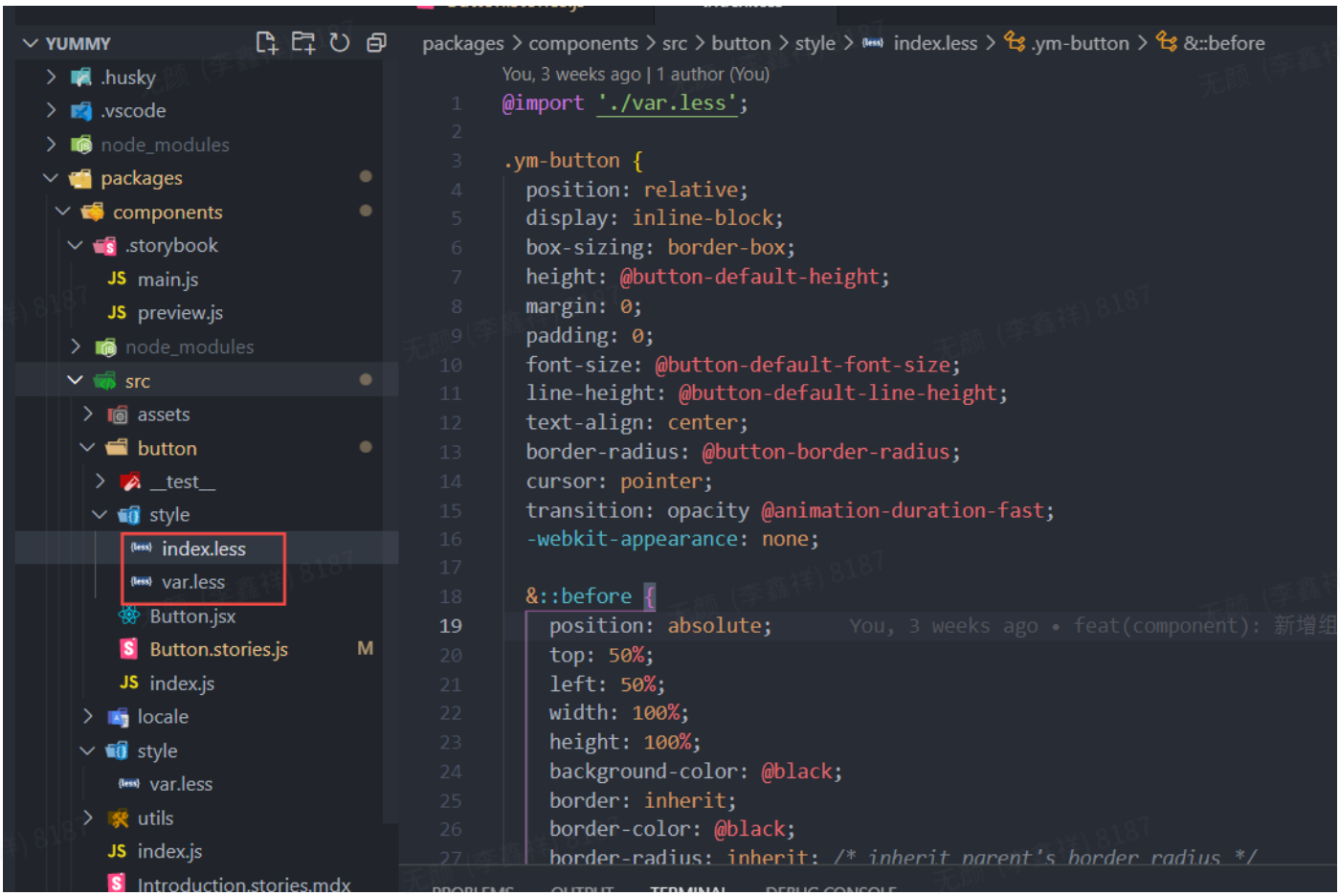
TypeScript

```typescript
import { defineComponent } from 'vue';
import { createNamespace } from '../utils';
import { BORDER_SURROUND } from '../utils/constant';

const [name, bem] = createNamespace('button');
```

```
6
7  export default defineComponent({
8    name,
9    setup(props, { emit, slots }) {
10     const renderText = () => {
11         return <span class={bem('text')}>{text}</span>;
12     };
13
14     const onClick = (event) => {
15       if (props.loading) {
16         event.preventDefault();
17       } else if (!props.disabled) {
18         emit('click', event);
19       }
20     };
21
22     return () => {
23       const { tag, type, size, block, round, plain, square, loading, disabled,
   hairline, nativeType } = props;
24
25       // classes 根据 Props 来动态生成
26       const classes = [
27         bem([
28           type,
29           size,
30           {
31             plain,
32             block,
33             round,
34             square,
35             loading,
36             disabled,
37             hairline,
38           },
39         ]),
40         { [BORDER_SURROUND]: hairline },
41       ];
42
43       return (
44         <tag type={nativeType} class={classes} style={getStyle()} disabled=
   {disabled} onClick={onClick}>
45           <div class={bem('content')}>{renderText()}</div>
46         </tag>
47       );
48     };
49   },
50 });
```

注意：这里采用的 classname 命名规则为 BEM，具体采用哪种规范试团队情况而定。

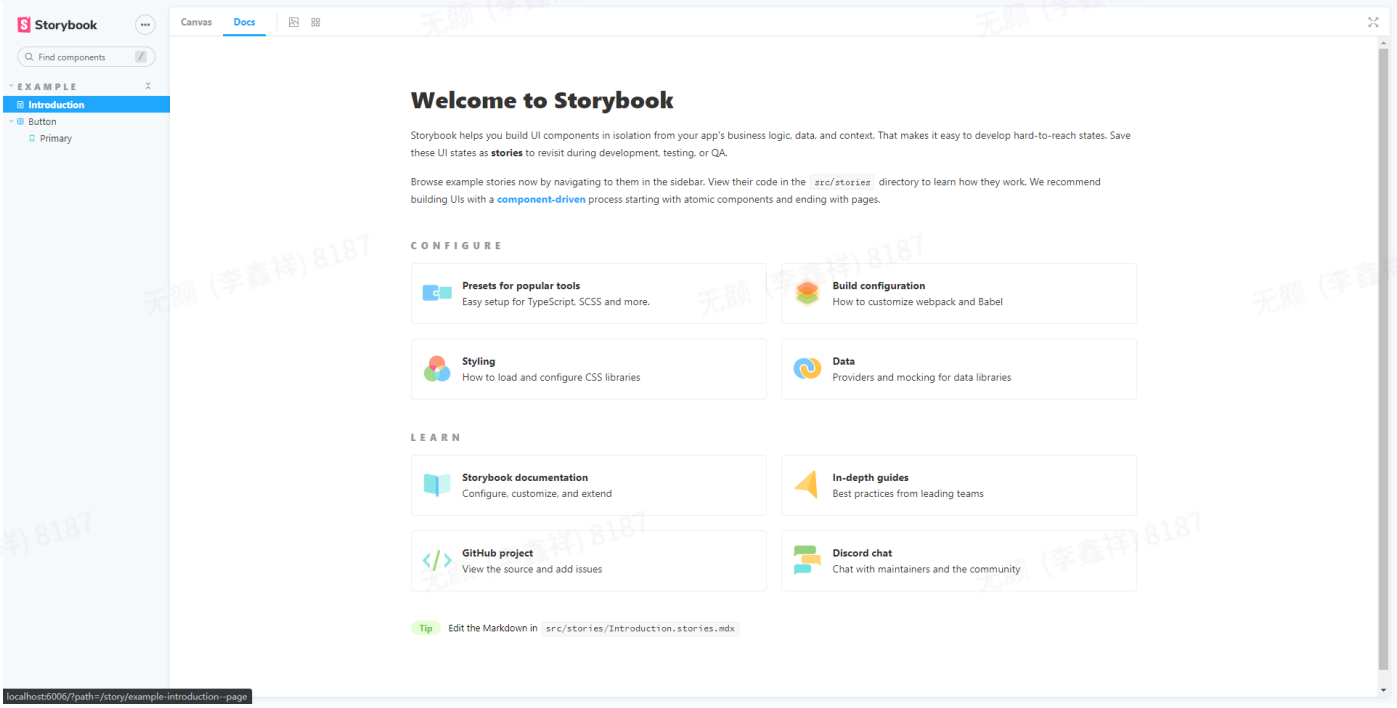
根据主题定制的具体要求，对 less 变量进行合理拆分：



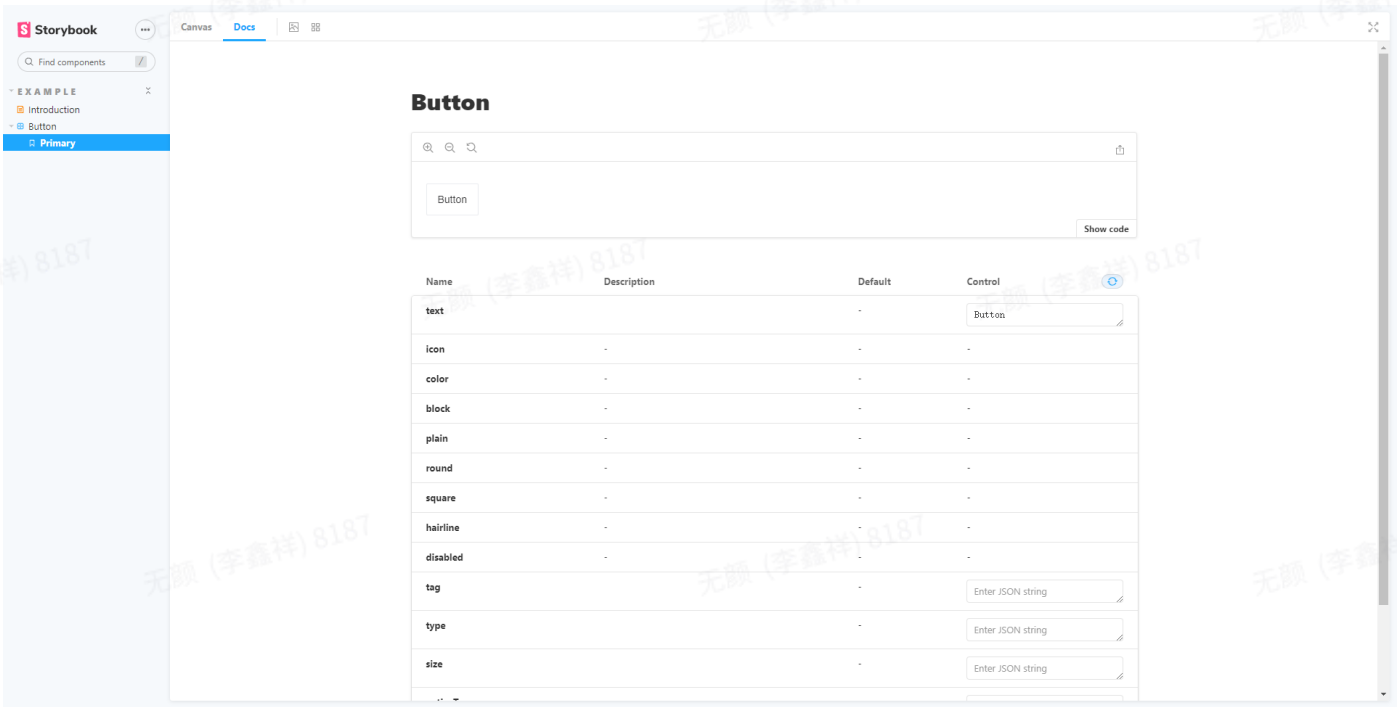
至此一个简单的组件库目录就搭建完成了，虽然省略了一些步骤，但是核心内容都列出来了。读者可以结合源码来进行了解。

4.3 示例

这里是主入口文档说明，可作为各组件分类的快捷说明：



交互式文档，可控制的文档是通过组件内 Props 的 JSdoc 自动生成：



源码地址：

https://git.yummy.tech/common/yummy/-/tree/master/packages/components

五、推荐阅读

- Design Systems Handbook
- Component Driven Development

- Atomic Design
- Component Driven User Interfaces