antd-plus-ui

一个基于 antd4.24.10 的二次封装组件库

项目初始化

```
// 拉取项目代码
git clone https://github.com/feihu1024/antd-plus-ui.git

// 安装依赖
cd antd-plus-ui
npm install

// 启动项目
npm start
```

项目结构

```
antd-plus-ui
├─ .dumi                          # api站点相关目录，例如全局样式、全局脚本、站点主题、自定义组件等
|   ├─ theme                      # 自定义主题
|   └─ global.less          # 全局样式
├─ .husky                         # 代码提交相关配置
├─ docker                         # docker部署相关文件
|   └─ nginx.conf           # nginx配置
├─ scripts                        # 全局脚本目录
|   ├─ clean-tmp.js               # 清除.dumi目录下自动生成的tmp目录
|   └─ source-replace-loader.js   # 用于解决第三方主题antd版本冲突的webpack loader
├─ docs                           # 文档目录，适用于普通文档生成路由，详见dumi官方介绍
|   ├─ components       # 组件库页面的文档
|   ├─ guide.zh-CN.md       # 指南页面的文档
|   └─ index.zh-CN.md       # 首页页面的文档
├─ public                         # 站点的静态资源目录
├─ src                            # 组件目录
|   ├─ foo                     # foo组件
|       ......                  # 其他更多组件
|   └─ index.ts           # 组件导出配置
├─ .dumirc.ts                     # dumi 的配置文件
├─ .fatherrc.ts                   # father 的配置文件，用于组件库打包
├─ .eslintrc.js                   # eslint插件配置
├─ .prettierrc.js       # prettier插件配置
├─ .release-it.js       # release-it配置，用于管理自动发布流程
├─ .stylelintrc         # stylelint插件配置
└─ tsconfig.json        # ts相关配置
```

组件开发指南

组件的基本结构

```
🗀src   # 组件目录
├ 🗀DCascader      # 组件根目录
|  ├ 🗎index.less      # 组件样式文件
|  ├ 🗎index.tsx       # 组件入口文件
|  ├ 🗎index.zh-CN.md  # 组件文档
|  └ 🗀demos           # 组件示例目录
|     ├ 🗎basicDemo.tsx
|     ├ 🗎loadChildrenDemo.tsx
|     └ 🗎loadingDemo.tsx
```

一个组件的基本结构如上图所示，为保持组件风格一致，开发时应当尽量遵循以下规则:

- 组件根目录以驼峰命名
- 编写组件时应尽可能提供相应的 ts 类型定义
- 组件样式如果较为复杂，应当进行合理拆分，并添加一个 styles 目录对其进行统一管理
- 组件样式请尽量不要使用 `模块化` 引入方式，否则使用者很难进行样式覆盖
- 组件示例应当统一放置在 demos 目录下，示例中尽量不要存放体积过大的静态文件，如果有，应当在 public/mock 目录下新建相应的文件进行存放

组件的编写

```
import React, { forwardRef } from 'react';
import { Cascader } from 'antd';

import './index.less'; // 引入样式
// import styles from './index.modules.less'; 不推荐的形式，会导致使用者难以覆盖样式

export type DCascaderProps = any // ts类型定义

// 组件主体结构
function InternalCascader(props: DCascaderProps, ref: React.Ref<CascaderRef>) {
    return <div>DCascader组件</div>
}

// 组件导出
const DCascader = forwardRef(InternalCascader);
export default DCascader;
```

组件的导出

所有组件都需要在 src/index.tsx 下进行导出，导出时应当导出组件必要的入口文件及相关的 ts 类型定义

```
export type { DCascaderProps } from './DCascader';
export { default as DCascader } from './DCascader';
```

组件示例的编写

```
import { useEffect, useState } from 'react';
// 使用当前组件库的别名路径而不是相对路径
import { DCascader } from 'antd-plus-ui'; // 正确示例
// import DCascader from '../index';  错误示例
const getRegionData = () => {
  return new Promise<{ provinceList: any[]; cityList: any[]; countyList: any[]
}>((resolve) => {
```

```
    async function exec() {
      // 体积较大的静态资源文件从public/mock下引入
      const bodyProvince = await
fetch('/mock/dcascader/china_region_province.json');
      const provinceList = await bodyProvince.json();
      const bodyCity = await fetch('/mock/dcascader/china_region_city.json');
      const cityList = await bodyCity.json();
      const bodyCounty = await
fetch('/mock/dcascader/china_region_county.json');
      const countyList = await bodyCounty.json();
      resolve({ provinceList, cityList, countyList });
    }
    exec();
  });
};

export default function BasicDemo() {
  const [regionData, setRegionData] = useState<{
    provinceList: any[];
    cityList: any[];
    countyList: any[];
  }>({ provinceList: [], cityList: [], countyList: [] });

  const getOptionsAsync = (value, option): Promise<Array<{ value: string; label:
string }>> => {
    return new Promise((resolve) => {
      const { provinceList, cityList, countyList } = regionData;
      let options;
      if (option) {
        const listMap = { province: cityList, city: countyList };
        const codeMap = { province: 'provinceCode', city: 'cityCode' };
        const { level, code } = option;
        const list = listMap[level]?.filter((item) => item[codeMap[level]] ===
code);
        options = list?.map((item) => ({
          ...item,
          value: item.code,
          label: item.name,
          isLeaf: item.level === 'county',
        }));
      } else {
        options = provinceList.map((item) => ({
          ...item,
          label: item.name,
          value: item.code,
          isLeaf: false,
        }));
      }

      resolve(options);
    });
  };

  const onChange = (values, options) => {
    console.log(values, options);
  };

  useEffect(() => {
```

```
    getRegionData().then((res) => setRegionData(res));
  }, []);

  return <DCascader options={getOptionsAsync} showSearch onChange={onChange} />;
}
```

组件文档的编写

组件文档使用 markdown 格式编写， 所有 Markdown 配置均以 FrontMatter 的形式配置在 Markdown 文件顶端，具体用法请参照 dumi 官方介绍， 对于开发的组件，应至少包含以下结构：

- 组件介绍（一个二级标题,包含组件的基本说明）
- 组件示例（若干二级标题，每个标题代表一个示例）
- API 文档（一个二级标题，命名为 API，以表格形式列出所有可用的 api）

```
---
title: DCascader
description: 基于 antd 4.24.10 Cascader 的二次封装组件
tocDepth: 2
nav:
  title: 组件
  path: /components
group:
  title: 表单
---

## 组件特性

- options、loadData 均支持传入异步函数，在 From 表单组件中使用更方便
- 加载选项列表时可以显示加载中效果
- 本地搜索时默认匹配 label 字段
- 文本框与下拉面板同宽

## 基础用法

<code src="./demos/basicDemo.tsx"  title="示例标题" description="示例说明"></code>

## 动态加载子级列表

<code src="./demos/loadChildrenDemo.tsx" title="示例标题" description="示例说明">
</code>

## 显示加载中

<code src="./demos/loadingDemo.tsx" title="示例标题" description="示例说明"></code>

## API

| 参数  | 说明     | 类型      | 默认值      | 版本      |
|  :-- | :--      | :--      | :----      | :---    |
| api1 | api1说明 | api1类型 | api1默认值 | api1版本 |
| api2 | api2说明 | api2类型 | api2默认值 | api2版本 |
| api3 | api3说明 | api3类型 | api3默认值 | api3版本 |
```

代码风格及提交规范

vscode 插件

为保证团队风格的一致，推荐在编辑器中安装以下插件，项目中已包含相应插件的配置，如无必要，请勿修改

插件名称	插件说明
ESLint	js 代码检查工具
Stylelint	css 样式检查工具
Prettier	代码格式化工具
GitLens	代码提交插件

代码检查与格式化

由于项目已经包含 husky，每次提交前都会自动进行 lint 检查及代码格式化，如果检查不通过则会拒绝提交。代码提交时应当确保已经消除了所有 lint 错误，并尽量处理 lint 警告。lint 检查也可以手动执行，命令如下：

```
npm run lint        // 同时检查js与css

npm run lint:es     // 只检查js

npm run lint:css    // 只检查css

npm run prettier    // 执行代码格式化

npm run doctor      // 执行依赖检查
```

git 提交

本项目使用 husky 方案来规范 代码的提交,提交代码时请尽量遵循以下约定

提交类型	类型说明
feat	新增功能
fix	bug 修复
docs	文档更新
style	不影响程序逻辑的代码修改(修改空白字符，补全缺失的分号等)
refactor	重构代码(既没有新增功能，也没有修复 bug)
perf	性能优化
test	新增测试用例或是更新现有测试
build	主要目的是修改项目构建系统(例如 glup，webpack，rollup 的配置等)的提交
ci	主要目的是修改项目集成流程(例如 Travis，Jenkins，GitLab CI，Circle 等)的提交
chore	不属于以上类型的其他类型(日常事务)
revert	回滚某个更早之前的提交

```
git commit -m <type>[optional scope]: <description> // 提交格式

git commit -m "fix: 修复bug"  // type后的冒号和空格不可省略，description不能以大写字母开头
```

详细规则请前往commitlint官方文档查看

调试与发布

在测试项目中调试

1. 在当前项目中通过 link 命令将当前包链接到全局

   ```
   cd antd-plus-ui
   npm link         // 将当前包链接到全局
   // npm unlink    调试完成后解除链接
   ```

2. 在测试项目中链接当前包

   ```
   cd test-project
   npm link @pointcloud/pcloud-components
   // npm unlink --no-save package && npm install  调试完成后解除链接,恢复依赖
   ```

发布项目

1. 前置工作
   - 执行 `npm whoami` 查看当前用户是否已经登录，如果未登录则执行 `npm login`
   - 检查 `package.json` 中的 NPM 包名及 `publishConfig` 是否符合预期
2. release 发布

   使用 release-it 可以根据 release-it 配置 自动完成发布前的准备工作，包括：更新版本号、生成 tag 、更新 changelog、git 提交及推送远程仓库、npm 发布等

```
npm run release
```