1. Umi是什么

Umi，中文可发音为**乌米**，是可扩展的企业级前端应用框架。Umi 以路由为基础的，同时支持配置式路由和约定式路由，保证路由的功能完备，并以此进行功能扩展。然后配以生命周期完善的插件体系，覆盖从源码到构建产物的每个生命周期，支持各种功能扩展和业务需求<https://umijs.org/zh-CN/docs>

* 🎉 **可扩展**，Umi 实现了完整的生命周期，并使其插件化，Umi 内部功能也全由插件完成。此外还支持插件和插件集，以满足功能和垂直域的分层需求。
* 📦 **开箱即用**，Umi 内置了路由、构建、部署、测试等，仅需一个依赖即可上手开发。并且还提供针对 React 的集成插件集，内涵丰富的功能，可满足日常 80% 的开发需求。
* 🐠 **企业级**，经蚂蚁内部 3000+ 项目以及阿里、优酷、网易、飞猪、口碑等公司项目的验证，值得信赖。
* 🚀 **大量自研**，包含微前端、组件打包、文档工具、请求库、hooks 库、数据流等，满足日常项目的周边需求。
* 🌴 **完备路由**，同时支持配置式路由和约定式路由，同时保持功能的完备性，比如动态路由、嵌套路由、权限路由等等。
* 🚄 **面向未来**，在满足需求的同时，我们也不会停止对新技术的探索。比如 dll 提速、modern mode、webpack@5、自动化 external、bundler less 等等。

1. 什么时候不用umi

React版本在16.8及以上

Node版本不低于node10

Ie版本不低于ie8

有很强的webpack自定义需求

需要选择不同的路由方案

1.项目做大的时候，开发调试的启动和热更新时间会变得很长。  
2.大应用下，网站打开很慢，有没有办法基于路由做到按需加载。  
3.dva的model每次都要手写载入，能否一开始就同项目初始化好？

使用乌米，即可解决以上问题，并且还能提供如下优势：

* 开箱即用，内置 react、react-router 等
* 类 next.js 且功能完备的路由约定，同时支持配置的路由方式
* 完善的插件体系，覆盖从源码到构建产物的每个生命周期
* 一键兼容到 IE9
* 完善的 TypeScript 支持
* 与 dva 数据流的深入融合

1. 为什么不是create-react-app

create-react-app 是基于 webpack 的打包层方案，包含 build、dev、lint 等，他在打包层把体验做到了极致，但是不包含路由，不是框架，也不支持配置。所以，如果大家想基于他修改部分配置，或者希望在打包层之外也做技术收敛时，就会遇到困难。

1. Next.js

next.js 是个很好的选择，Umi 很多功能是参考 next.js 做的。要说有哪些地方不如 Umi，我觉得可能是不够贴近业务，不够接地气。比如 antd、dva 的深度整合，比如国际化、权限、数据流、配置式路由、补丁方案、自动化 external 方面等等一线开发者才会遇到的问题。

1. 快速上手

创建目录

$ mkdir myapp && cd myapp

# 安装依赖

$ yarn add umi

# 创建页面

$ npx umi g page index --typescript –less

# 启动开发

$ npx umi dev

6.umi3项目初始化

## 环境准备

首先得有 [node](https://link.juejin.cn?target=https%3A%2F%2Fnodejs.org%2Fen%2F)，并确保 node 版本是 10.13 或以上。

推荐使用 yarn 管理 npm 依赖

本项目使用的版本为 node v14.17.5 yarn 1.22.15

## 脚手架

桌面新建umi3文件夹, 用vscode打开, 打开vscode终端,

执行 yarn create @umijs/umi-app 创建项目

安装依赖 yarn

启动项目 yarn start

# 配置 prettier,eslint, stylelint

umi 维护了一个 prettier，eslint，stylelint 的配置文件合集 [umi-fabric](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fumijs%2Ffabric)

yarn add @umijs/fabric -D

根目录新建下面三个文件,删除.prettierrc文件

.eslintrc.js、.prettierrc.js、.stylelintrc.js

**配置如下**

//.eslintrc.js 配置

module.exports = {

extends: [require.resolve('@umijs/fabric/dist/eslint')],

// in antd-design-pro

globals: {

ANT\_DESIGN\_PRO\_ONLY\_DO\_NOT\_USE\_IN\_YOUR\_PRODUCTION: true,

page: true,

},

rules: {

// your rules

'prefer-const': 0,

},

};

//.prettierrc.js 配置

const fabric = require('@umijs/fabric')

module.exports = {

...fabric.prettier,

semi: false,

}

//.stylelintrc.js 配置

const fabric = require('@umijs/fabric')

module.exports = {

...fabric.stylelint,

}

根目录新建eslint忽略文件 .eslintignore

.eslintrc.js

node\_modules

在package.json 里面的lint-staged 新增 "eslint --fix"

最后你的 vscode 要安装这三个同名扩展插件，这时候分别去更改 js、less 文件，会发现已经有风格校验了。

验证

修改src/paegs文件夹下的index.tsx文件,新增一个a变量,有eslint错误提示,说明eslint生效了

然后再单独提交index.tsx这一个文件

会提示有错误,无法提交，说明pre-commmit 钩子生效

## 保存时自动格式化代码

在vscode设置 文本编辑器的格式化里面 勾选Format on Save

## 我的eslint，或者prettier 不生效？

去到终端里的输出，找到eslint或者prettier 看他们的输出日志，是否正常。如果有报错，根据报错信息处理问题

检查步骤:

* 确保安装umi-fabric
* 检查配置文件是否存在
* vscode 得eslint 和prettier 插件是否下载
* 确认输出日志，是否有报错

## pre-commit时的lint-staged 不生效

在package.json中 我们配置了如下得代码

意思是 在代码commit之前 执行prettier格式化代码和eslint fix 如果你在提交代码的时候没有生效，请执行

yarn install --force

复制代码

执行这个命令重新拉取依赖

## 不生效的原因？

自己刚开始也是各种google，查看文档，也没有找出原因，最后在umi2的一个issue里面，自己找到了答案。

原因在于我们初始化git仓库的顺序，如果我们先初始化git仓库 然后再创建项目，再拉取依赖。是没有任何问题的。

如果我们先创建了umi项目，拉去依赖，最后初始化git，提交代码到git仓库，当我们拉去依赖时， 这是就还没有.git 就没有生成相关的pre-commit，所以就没有生效。所以这时我们就只需要在重新拉取下依赖就可以了。

# 配置css初始化代码

为什么要初始化css

建站老手都知道，这是为了考虑到浏览器的兼容问题，其实不同浏览器对有些标签的默认值是不同的，如果没对CSS初始化往往会出现浏览器之间的页面差异。当然，初始化样式会对SEO有一定的影响，但鱼和熊掌不可兼得，但力求影响最小的情况下初始化。

最简单的初始化方法就是： \* {padding: 0; margin: 0;} 。有很多人也是这样写的。这确实很简单，但有人就会感到疑问：\*号这样一个通用符在编写代码的时候是快，但如果网站很大，CSS样式表文件很大，这样写的话，他会把所有的标签都初始化一遍，这样就大大的加强了网站运行的负载，会使网站加载的时候需要很长一段时间。

CSS初始化是指重设浏览器的样式。不同的浏览器默认的样式可能不尽相同，所以开发时的第一件事可能就是如何把它们统一。如果没对CSS初始化往往会出现浏览器之间的页面差异。每次新开发网站或新网页时候通过初始化CSS样式的属性，为我们将用到的CSS或html标签更加方便准确，使得我们开发网页内容时更加方便简洁，同时减少CSS代码量，节约网页下载时间。

Umi 中约定 src/global.css 为全局样式，如果存在此文件，会被自动引入到入口文件最前面。

src下面新建global.css,代码如下

body,

ol,

ul,

h1,

h2,

h3,

h4,

h5,

h6,

p,

th,

td,

dl,

dd,

form,

fieldset,

legend,

input,

textarea,

select,

figure,

figcaption {

margin: 0;

padding: 0;

}

li {

list-style-type: none;

}

a {

text-decoration: none;

}

a:hover {

text-decoration: underline;

}

img {

border: none;

}

input {

outline: none;

}

# 配置文件

Umi 在 .umirc.ts 或 config/config.ts 中配置项目和插件，支持 es6。

如果项目的配置不复杂，推荐在 .umirc.ts 中写配置； 如果项目的配置比较复杂，可以将配置写在 config/config.ts 中，并把配置的一部分拆分出去，比如路由配置可以拆分成单独的 routes.ts

推荐两种配置方式二选一，.umirc.ts 优先级更高。

我们采用config的方式,删除.umirc.ts,根目录新建config文件夹, 在里面新建config.ts

默认内容如下

import { defineConfig } from 'umi';

export default defineConfig({

nodeModulesTransform: {

type: 'none',

},

routes: [

{ path: '/', component: '@/pages/index' },

],

fastRefresh: {},

});

# 多环境多配置文件

可以通过环境变量 UMI\_ENV 区分不同环境来指定配置。

为了兼容性,可借助三方工具 [cross-env](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fkentcdodds%2Fcross-env)来设置环境变量

yarn add cross-env --dev

复制代码

在package.json中的script中

"start": "cross-env UMI\_ENV=dev umi dev",

"start:test": "cross-env UMI\_ENV=test umi dev",

"start:prd": "cross-env UMI\_ENV=prd umi dev",

"build": "cross-env UMI\_ENV=dev umi build",

"build:test": "cross-env UMI\_ENV=test umi build",

"build:prd": "cross-env UMI\_ENV=prd umi build",

复制代码

然后再config文件夹下 新建

config.dev.ts,config.test.ts,config.prd.ts

代表开发环境,测试环境,生产环境的配置文件.

config.dev.ts

import { defineConfig } from 'umi';

export default defineConfig({

define: {

CurrentEnvironment: 'dev',

},

});

复制代码

config.test.ts

import { defineConfig } from 'umi';

export default defineConfig({

define: {

CurrentEnvironment: 'test',

},

});

复制代码

config.prd.ts

import { defineConfig } from 'umi';

export default defineConfig({

define: {

CurrentEnvironment: 'prd',

},

});

复制代码

CurrentEnvironment 变量代表当前的环境，后面根据不同的环境配置不同的请求地址会用到

[define](https://link.juejin.cn?target=https%3A%2F%2Fumijs.org%2Fzh-CN%2Fconfig%23define) 用于提供给代码中可用的变量,定义的变量可以全局拿到

这时 执行 yarn start:prd,然后去到pages的index.tsx打印CurrentEnvironment.

这时需要去到根目录的 typings.d.ts 添加

// 声明当前的环境

declare const CurrentEnvironment: 'dev' | 'test' | 'prd';

复制代码

然后报错消失 控制台打印如下

这时 重新执行yanr start:test 控制台打印如下

环境变量和多环境多配置 成功

\*\*注意点\*\*:

config.ts作为配置文件时，记得删除.umirc.ts 不然config.ts不会生效。

# 自定义环境变量

如果我们想自定义一个环境变量,REACT\_APP\_ENV. 同样我们可以在package.json里面设置

然后我们要这样拿到这个变量呢?

首先 我们要在config.ts 的 define 配置

define: {

REACT\_APP\_ENV: process.env.REACT\_APP\_ENV,

},

复制代码

然后再在根目录的 typings.d.ts 定义

declare const REACT\_APP\_ENV: string;

复制代码

这样就可以以在全局中拿到和使用 REACT\_APP\_ENV这个环境变量了.

可以在任意组件 直接打印

console.log('自定义环境变量', REACT\_APP\_ENV);

复制代码

# 系统自带的环境变量

[官方提供的环境变量](https://link.juejin.cn?target=https%3A%2F%2Fumijs.org%2Fzh-CN%2Fdocs%2Fenv-variables%23%25E7%258E%25AF%25E5%25A2%2583%25E5%258F%2598%25E9%2587%258F%25E5%2588%2597%25E8%25A1%25A8)

怎么使用?

在根目录新建.env 环境变量配置文件

然后写入

PORT=3000 // 表示启动的端口号为3000

COMPRESS = none // 不压缩 CSS 和 JS

复制代码

还有一些环境变量 不能配在 .env 中，只能在命令行里添加

比如 FORK\_TS\_CHECKER 默认不开启 TypeScript 类型检查，值为 1 时启用。

"start": "cross-env FORK\_TS\_CHECKER=1 UMI\_ENV=dev umi dev",

复制代码

# 请求的封装

src文件夹下新建 request文件夹 新建request.ts

request.ts

/\*\*

\* 网络请求工具 封装umi-request

\* 更详细的 api 文档: https://github.com/umijs/umi-request

\*/

import { extend } from 'umi-request';

import type { RequestOptionsInit } from 'umi-request';

import { notification } from 'antd';

// codeMessage仅供参考 具体根据和后端协商,在详细定义.

const codeMessage = {

200: '服务器成功返回请求的数据。',

400: '发出的请求有错误，服务器没有进行新建或修改数据的操作。',

500: '服务器发生错误，请检查服务器。',

};

type mapCode = 200 | 400 | 500;

/\*\*

\* 错误异常处理程序

\*/

const errorHandler = (error: { response: Response }): Response => {

const { response } = error;

if (response && response.status) {

let errorText = codeMessage[response.status as mapCode] || response.statusText;

const { status, url } = response;

response

?.clone()

?.json()

?.then((res) => {

// 后端返回错误信息,就用后端传回的

errorText = res.msg ? res.msg : errorText;

notification.error({

message: `请求错误 ${status}: ${url}`,

description: errorText,

});

});

} else if (!response) {

notification.error({

description: '您的网络发生异常，无法连接服务器',

message: '网络异常',

});

}

return response;

};

/\*\*

\* 配置request请求时的默认参数

\*/

const request = extend({

errorHandler, // 默认错误处理

credentials: 'include', // 默认请求是否带上cookie

});

// 根据不同的开发环境,配置请求前缀

interface ApiPrefix {

dev: string;

test: string;

prd: string;

}

const apiPreFix: ApiPrefix = {

dev: 'http://120.55.193.14:3030/',

test: 'http://120.55.193.14:3030/',

prd: 'http://120.55.193.14:3030/',

};

// request拦截器, 携带token,以及根据环境,配置不同的请求前缀

request.interceptors.request.use((url: string, options: RequestOptionsInit) => {

// 不携带token的请求数组

let notCarryTokenArr: string[] = [];

if (notCarryTokenArr.includes(url)) {

return {

url: `${apiPreFix[CurrentEnvironment]}${url}`,

options,

};

}

// 给每个请求带上token

let token = localStorage.getItem('tokens') || '';

let headers = {

Authorization: `Bearer ${token}`,

};

return {

url: `${apiPreFix[CurrentEnvironment]}${url}`,

options: { ...options, interceptors: true, headers },

};

});

/\*\*

\* @url 请求的url

\* @parameter 上传的参数

\*/

// 封装的get,post.put,delete请求

const get = async (url: string, parameter?: Record<string, unknown>): Promise<any> => {

try {

const res = await request(url, { method: 'get', params: parameter });

return res;

} catch (error) {

console.error(error);

}

};

const deletes = async (url: string, parameter?: Record<string, unknown>): Promise<any> => {

try {

const res = await request(url, { method: 'delete', params: parameter });

return res;

} catch (error) {

console.error(error);

}

};

const post = async (url: string, parameter?: Record<string, unknown>): Promise<any> => {

try {

const res = await request(url, { method: 'post', data: parameter });

return res;

} catch (error) {

console.error(error);

}

};

const put = async (url: string, parameter?: Record<string, unknown>): Promise<any> => {

try {

const res = await request(url, { method: 'put', data: parameter });

return res;

} catch (error) {

console.error(error);

}

};

export default {

get,

post,

put,

deletes,

};

复制代码

这里封装了umi-request,统一处理了接口错误,请求拦截器携带token等.最后在配合useRequest 非常的好用.

# umi中使用dva

## 介绍

包含以下功能，

* **内置 dva**，默认版本是 ^2.6.0-beta.20，如果项目中有依赖，会优先使用项目中依赖的版本。
* **约定式的 model 组织方式**，不用手动注册 model
* **文件名即 namespace**，model 内如果没有声明 namespace，会以文件名作为 namespace
* **内置 dva-loading**，直接 connect loading 字段使用即可
* **支持 immer**，通过配置 immer 开启

## 约定式的 model 组织方式

符合以下规则的文件会被认为是 model 文件，

* src/models 下的文件
* src/pages 下，子目录中 models 目录下的文件
* src/pages 下，所有 model.ts 文件(不区分任何字母大小写)

## 实际使用

比如在src下新建 models文件夹,里面新建test.ts

test.ts

import type { Effect, Reducer, Subscription } from 'umi'; // 映入umi 定义好的ts类型

import axios from '../request/request'; // 引入封装好的网络请求

// state 接口

export interface TextModelState {

name?: string;

testData?: string;

}

// test model接口

export interface TextModelType {

namespace: 'testModel';

state: TextModelState;

effects: {

query: Effect;

};

reducers: {

save: Reducer<TextModelState>;

msg: Reducer<TextModelState>;

};

subscriptions?: { setup: Subscription };

}

const IndexModel: TextModelType = {

namespace: 'testModel',

state: {

name: '初始名字',

testData: '初始testData',

},

effects: {

\*query(action, { call, put }) {

const getDataTest = async () => {

const data = await axios.get('test');

return data;

};

let testData = yield call(getDataTest);

yield put({

type: 'msg',

data: { testData: testData?.msg },

});

},

},

reducers: {

save(state) {

return {

...state,

name: 'jimmy',

};

},

msg(state, action) {

return {

...state,

testData: action?.data?.testData,

testData2: action?.data?.testData2,

};

},

},

};

export default IndexModel;

复制代码

在src/pages下的index.tsx中使用

index.tsx

import type { Effect, Reducer, Subscription } from 'umi'; // 引入umi 定义好的ts类型

import axios from '../request/request'; // 引入封装好的网络请求

// state 接口

export interface TextModelState {

name?: string;

testData?: string;

}

// test model接口

export interface TextModelType {

namespace: 'testModel';

state: TextModelState;

effects: {

query: Effect;

};

reducers: {

save: Reducer<TextModelState>;

msg: Reducer<TextModelState>;

};

subscriptions?: { setup: Subscription };

}

const IndexModel: TextModelType = {

namespace: 'testModel',

state: {

name: '初始名字',

testData: '初始testData',

},

effects: {

\*query(action, { call, put }) {

const getDataTest = async () => {

const data = await axios.get('test');

return data;

};

let testData = yield call(getDataTest);

yield put({

type: 'msg',

data: { testData: testData?.msg },

});

},

},

reducers: {

save(state) {

return {

...state,

name: 'jimmy',

};

},

msg(state, action) {

return {

...state,

testData: action?.data?.testData,

testData2: action?.data?.testData2,

};

},

},

};

export default IndexModel;

复制代码

# mfsu

启用 mfsu 后，热启动得到 \*\*10 倍\*\* 提升；热更新提升 \*\*50%\*\*  以上！

如何启用

在 config/config.ts 中添加 mfsu:{}

# 项目源代码

[请点击我](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fchenglianjie%2Fcommom-umi-template)

和两个小伙伴一起,会根据实际运用中出现的问题或者没有考虑完善的地方,持续的更新迭代.如有问题,欢迎提Issue或者在评论区留言

# FAQ

## umi 不是内部或外部命令

解决办法

执行 yarn global bin 拿到 bin 路径。然后把这个路径添加到环境变量里面的系统变量的path里面

如果还是不行，执行

yarn global add umi

复制代码

**如遇到更多问题，请查考**

[官方FAQ](https://link.juejin.cn?target=https%3A%2F%2Fumijs.org%2Fzh-CN%2Fdocs%2Ffaq%23test)

[官方仓库的issue](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fumijs%2Fumi%2Fissues)

大家还是要把官方文档看两遍哦,一些基础,简单的知识本文章没有涉及.

作者：Jimmy\_kiwi  
链接：https://juejin.cn/post/7021358536504393741  
来源：稀土掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。