

# React18 SSR + Next.js<sup>13</sup>

刘军 liujun

# 目录

## content



**1 中间件和匹配器**

**2 Layout和生命周期**

**3 网络请求的封装**

**4 编写后端接口**

**5 页面渲染模式**

**6 数据获取和Redux**

**7 项目实战和部署**

# 中间件(middleware)

- Next.js的中间件允许我们去**拦截**客户端发起的请求，例如：API请求、router切换、资源加载、站点图片等。
- 拦截客户端发起的请求等等之后，便可对这些进行：重写、重定向、修改请求响应标头、或响应等操作。
- 使用中间件需按照以下步骤操作：
  - 1.在**根目录**中创建 **middleware.ts** 文件
  - 2.从 middleware.ts 文件中导出一个中间件**middleware**函数（支持 async，并只允许在服务端），会接收两个参数：
    - ✓ req：类型为 **NextRequest**
    - ✓ event：类型为 **NextFetchEvent**
  - 3.通过返回NextResponse对象来实现重定向等功能
    - ✓ next()- 将继续中间件链
    - ✓ redirect()- 将重定向，如：重定向到某个页面
    - ✓ rewrite()- 将重写URL，如：配置反向代理
  - 4.没返回值：页面将按预期加载 和 返回 next() 一样

```
import { NextResponse } from "next/server";
import type { NextRequest } from "next/server";
// run on server
export function middleware(request: NextRequest) {
  // 1. 拦截哪些请求: api、router、assets、站点图标 ....
  console.log(request.nextUrl.pathname);
  return NextResponse.next();
}
```

# 匹配器 (Matcher)

■ 匹配器允许我们过滤中间件以在特定路径上运行，比如：

- ❑ matcher: '/about/:path\*', 意思是匹配以 /about/\* 开头的路径。其中路径开头的: 是修饰符，而 \* 代表0个 或 n个
- ❑ matcher: [ '/about/:path\*', '/dashboard/:path\*' ], 意思是匹配以 /about/\* 和 /dashboard/\* 开头的路径
- ❑ matcher: [ '/((?!api|\_next/static/favicon.ico).\*)' ], 意思是不匹配以 api、\_next、static、favicon.ico 开头的路径

```
export const config = {  
  matcher: '/about/:path*',  
}
```

```
export const config = {  
  matcher: ['/about/:path*', '/dashboard/:path*'],  
}
```

```
export const config = {  
  matcher: [  
    /*  
     * Match all request paths except for the ones starting with:  
     * - api (API routes)  
     * - _next/static (static files)  
     * - favicon.ico (favicon file)  
     */  
    '/((?!api|_next/static/favicon.ico).*)',  
  ],  
}
```

❑ 注意：上面的 path 是占位符，不是固定的。

■ 下面我们通过 **中间件 + 匹配器** 来实现路由的拦截：

□ matcher: `[ '/((?!api|_next/static|favicon.ico).*)' ]`

```
import { NextResponse } from "next/server";
import type { NextRequest, NextFetchEvent } from "next/server";

export function middleware(request: NextRequest, event: NextFetchEvent) {
  console.log("middleware run on server=", request.nextUrl);
  const { origin, pathname } = request.nextUrl;
  if (pathname === "/") {
    // if(如果没有登录) todo ...
    return NextResponse.redirect(new URL("/login", origin));
  } else {
    return NextResponse.next(); // 直接放行
  }
}
```

# 布局组件 (Layout)

- Layout布局是页面的包装器，可以将多个页面的共性东西写到Layout布局中，使用 `props.children` 属性来显示页面内容
  - 例如：每个页面的页眉和页脚组件，这些具有共性的组件我们是可写到一个Layout布局中。
- Layout布局的使用步骤：
  - 1.在components目录下新建 `layout.tsx` 布局组件
  - 2.接着在 `_app.tsx`中通过 `<Layout>` 组件包裹 `<Component>` 组件

```
// components/layout.js

import Navbar from './navbar'
import Footer from './footer'

export default function Layout({ children }) {
  return (
    <>
      <Navbar />
      <main>{children}</main>
      <Footer />
    </>
  )
}
```

```
import Layout from '../components/layout'

export default function MyApp({ Component, pageProps }) {
  return (
    <Layout>
      <Component {...pageProps} />
    </Layout>
  )
}
```

# 嵌套布局 (NestLayout)

- Layout布局可以作为所有页面的容器，也可以给每个页面一个单独的布局也是可以的，并且也可以在布局中嵌套布局。
- 因此，我们可以利用布局再嵌套一个布局来实现二级路由。

```
const Profile: NextPageWithLayout = memo(function () {  
  return <div>Profile</div>;  
});  
  
export default Profile;  
Profile.displayName = "Profile";  
  
Profile.getLayout = function getLayout(page: NextPageWithLayout) {  
  // 布局嵌套布局  
  return (  
    <Layout>  
      <NestLayout>{page}</NestLayout>  
    </Layout>  
  );  
};
```

- Next.js 和 Nuxt3一样，也支持嵌套路由(但是只在app目录下)，也是根据目录结构和文件的名称自动生成。
- 二级路由实现有两种方案：
  - 方案一：使用Layout布局嵌套来实现
  - 方案二：使用Next.js 13版本，新增的app目录（目前 beta 版本）

```
const NextLayout: FC<IProps> = function (props) {  
  return (  
    <div>  
      <div>nest layout</div>  
      <div>  
        <Link href={"/profile"}>  
          <button>profile</button>  
        </Link>  
        <Link href={"/profile/login"}>  
          <button>login</button>  
        </Link>  
        <Link href={"/profile/register"}>  
          <button>register</button>  
        </Link>  
      </div>  
      {props.children}  
    </div>  
  );  
};  
export default NextLayout;
```



# App目录和布局

■ Next.js 13版本，新增的app目录（目前 beta 版本，还是处于实验性阶段，需要在配置开始）

■ 第一步：创建app目录

■ 第二步：创建根HTML布局：layout.tsx

■ 第三步：创建首页：page.tsx

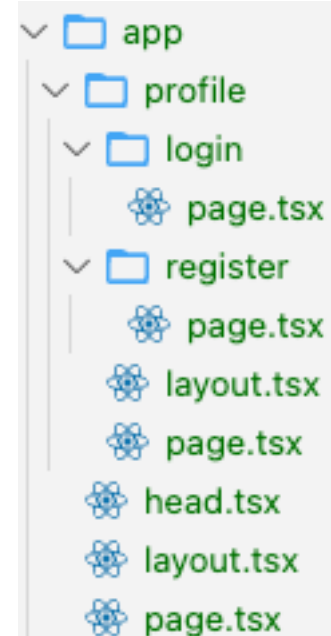
■ 第四步：创建head.tsx，定制head

■ 第五步：创建其它页面和布局

□ page.tsx

□ layout.tsx

```
/** @type {import('next').NextConfig} */  
const nextConfig = {  
  experimental: {  
    appDir: true,  
  },  
};  
  
module.exports = nextConfig;
```



## ■ 客户端渲染

class 组件	Hooks 组件
constructor	useState
getDerivedStateFromProps	useState 里面 update 函数
shouldComponentUpdate	useMemo
render	函数本身
componentDidMount	useEffect
componentDidUpdate	useEffect
componentWillUnmount	useEffect 里面返回的函数
componentDidCatch	无
getDerivedStateFromError	无

## ■ 服务器端渲染

- constructor
- UNSAFE\_componentWillMount
- render -> function component 本身

## ■ 安装依赖库

□ npm i axios -save

## ■ 封装axios步骤

□ 1.定义HYRequest类，并导出

□ 2.在类中定义request、get、post方法

□ 3.在request中使用axios发起网络请求

□ 4.添加TypeScript类型声明

```
class HYRequest {  
  // 1.声明instance的类型  
  instance: AxiosInstance; // 1.声明instance的类型  
  constructor(config: AxiosRequestConfig) {  
    // 2.创建axios实例  
    this.instance = axios.create(config);  
  }  
  // 3.编写request函数，request的T是指定响应结果res.data的类型  
  request<T>(config: AxiosRequestConfig): Promise<T> {  
    return new Promise((resolve, reject) => {  
      this.instance  
        .request<T, AxiosResponse<T>>(config) // 4.调用的instance.request  
        .then((res) => {  
          // 5.将结果resolve返回出去  
          resolve(res.data);  
        })  
        .catch((err) => {  
          reject(err);  
          return err;  
        });  
    });  
  }  
  // 发起get请求，T 代表是响应结果res.data的类型  
  get<T>= any>(url: string, params?: any): Promise<T> {  
    return this.request<T>({ url, params, method: "GET" });  
  }  
  post<T>= any>(url: string, data?: any): Promise<T> {  
    return this.request<T>({ url, data, method: "POST" });  
  }  
}  
export default new HYRequest({  
  baseURL: BASE_URL,  
  timeout: TIME_OUT,
```

- Next.js 提供了编写后端接口的功能（即 API Routes），编写接口可以在 `pages/api` 目录下编写
- 在 `pages/api` 目录下的任何 API Routes 文件都会自动映射到以 `/api/*` 前缀开头接口地址
- 比如：编写一个 `/api/user` 接口
  - 1.在 `pages/api` 目录下新建 `user.ts`
  - 2.接在在该文件中使用 `handler` 函数来定义接口
  - 3.然后就可以用 `fetch` 函数 或 `axios` 轻松调用：`/api/user` 接口了

```
export default function handler(req, res) {  
  if (req.method === 'POST') {  
    // Process a POST request  
  } else {  
    // Handle any other HTTP method  
  }  
}
```

## ■ 认识预渲染

- 默认情况下，Next.js 会 **预渲染** 每个页面，即预先为每个页面生成 HTML 文件，而不是由客户端 JavaScript 来完成。
- 预渲染可以带来更好的性能和 SEO 效果。
- 当浏览器加载一个页面时，页面依赖JS代码将会执行，执行JS代码后会激活页面，使页面具有交互性。(此过程称Hydration)

## ■ Next.js 具有两种形式的预渲染：

- **静态生成（推荐）**：HTML 在 **构建时** 生成，并在每次页面请求（request）时重用。
- **服务器端渲染**：在 **每次页面请求（request）时** 重新生成 HTML页面。

## ■ 提示：出于性能考虑，相对服务器端渲染，更 **推荐** 使用 **静态生成**。

# SSG-静态生成（一）

## ■ 静态生成（也称SSG 或 静态站点生成）

- 如果一个页面使用了 **静态生成**，在 **构建时** 将生成此页面对应的 HTML 文件。这意味着在生产环境中，运行 next build 时将生成该页面对应的 HTML 文件。然后，此 HTML 文件将在每个页面请求时被重用，还可以被 CDN 缓存。
- 在 Next.js 中，你可以静态生成 **带有或不带有数据** 的页面。接下来我们分别看看这两种情况。

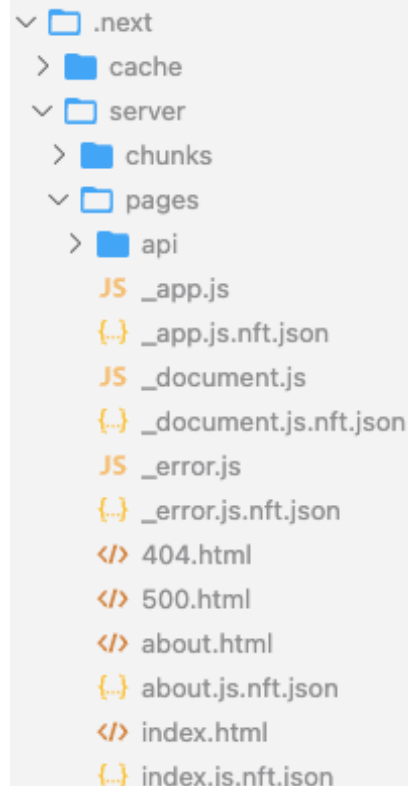
## ■ 生成不带数据的静态页面

- 默认情况下，Next.js 使用 “静态生成” 来预渲染页面但不涉及获取数据。如下例所示：

```
function About() {  
  return <div>About</div>  
}  
  
export default About
```

### □ 请注意：

- ✓ 此页面在预渲染时不需要获取任何外部数据。
- ✓ 在这种情况下，Next.js 只需在构建时为每个页面生成一个 HTML 文件即可。



```
▼ .next  
  > cache  
  ▼ server  
    > chunks  
    ▼ pages  
      > api  
        JS _app.js  
        {} _app.js.nft.json  
        JS _document.js  
        {} _document.js.nft.json  
        JS _error.js  
        {} _error.js.nft.json  
        <> 404.html  
        <> 500.html  
        <> about.html  
        {} about.js.nft.json  
        <> index.html  
        {} index.js.nft.json
```

# SSG-静态生成（二）

## ■ 需要获取数据的静态页面生成

□ 当某些页面需要获取外部数据以进行预渲染，通常有两种情况：

- ✓ 情况一：页面 **内容** 取决于外部数据：使用 Next.js 提供的 `getStaticProps` 函数。
- ✓ 情况二：页面 **paths（路径）** 取决于外部数据：使用 Next.js 提供的 `getStaticPaths` 函数（通常还要同时用 `getStaticProps`）。

## ■ 情况一：页面 **内容** 取决于外部数据

□ 比如，发起网络请求拿到页面**书籍列表**的数据，并展示。

□ 具体的使用步骤是：

- ✓ 1.先在`getStaticProps`函数中借助`axios`获取到数据
- ✓ 2.拿到异步数据之后 `return` 给页面组件
- ✓ 3.页面就可通过`props`拿到数据来渲染页面
- ✓ 4.在`build`时，经过以上步骤，一个静态页面就打包生成

```
export async function getStaticProps(context: any) {  
  // { locales: undefined, locale: undefined, defaultLocale: 'en' }  
  console.log(context);  
  const res = await fetchBooks();  
  return {  
    props: {  
      books: res.data.book,  
    },  
  };  
}
```

# SSG-静态生成 (三)

## ■ 情况二：页面 paths (路径) 取决于外部数据

- 例如，新建一个动态路由页面，然后发起网络请求拿到书本列表，然后每本书的信息都使用单独详情页面显示。
- 简单的理解就是，在build 阶段时，动态拿到n本书，然后根据n书动态生成n个静态详情页面。

// 动态生成 5 个BooksDetail静态页面

```
export const getStaticPaths: GetStaticPaths = async (context) => {  
  const res = await fetchBooks(5);  
  const ids = res.data.book?.map((item: any) => {  
    return {  
      params: { id: item.id + "" }, // id 需要是字符串  
    };  
  });  
  // paths 必须要是一个数组类型, fallback 为 false 表示当路径匹配不到是返回 404  
  return { paths: ids || [], fallback: false };  
};
```

```
export const getStaticProps: GetStaticProps = async (context) => {  
  console.log(context.params?.id);  
  const res = await fetchBookDetail(context.params?.id as string);  
  return {  
    props: {  
      book: res.data.book,  
    },  
  };  
};
```



# 静态生成应用场景

- 建议尽可能使用**静态生成（无论有与没有数据）**，因为静态生成的页面可以构建一次，并可由 CDN 提供服。
- 我们可以为多种类型的页面使用静态生成，包括：
  - 营销页面、官网网站
  - 博客文章、投资组合
  - 电子商务产品列表、帮助和文档
- 如果在用户请求**之前就可以预渲染**页面，那么应该选择静态生成。反之，静态生成就**不合适了**。
- 例如，页面要显示经常更新的数据，并且页面内容会在每次请求时发生变化，这时可以这样选择：
  - 静态生成与**客户端数据获取结合使用**：
    - ✓ 我们可以跳过预呈现页面的某些部分，然后使用客户端 JS 来填充它们，但是客户端渲染是不利于SEO优化的，例如：
      - 在useEffect 中获取数据，在客户端动态渲染页面。
  - **服务器端呈现（也称动态呈现）**：
    - ✓ Next.js 会根据每个请求预呈现一个页面。缺点是稍微慢一点，因为页面无法被 CDN 缓存，但预渲染页面将始终是最新的。

# 服务器端渲染 (SSR)

## ■ 服务器端渲染 (也称SSR 或 动态渲染)

- 如果页面使用的是 **服务器端渲染**，则会在 **每次页面请求时** 重新生成页面的 HTML。
- 要对页面使用服务器端渲染，**你需要 export 一个名为 `getServerSideProps` 的 `async` 函数。**
- 服务器将在每次页面请求时调用此函数。

■ 例如，假设你的某个页面需要预渲染频繁更新的数据（从外部 API 获取）。你就可以编写 `getServerSideProps` 获取该数据并将其传递给 Page，如下所示：

```
export const getServerSideProps: GetServerSideProps = async (context) => {  
  const res = await fetchGoods(context.query?.count as string);  
  return {  
    props: {  
      goods: res.data.goods,  
    },  
  };  
};
```

# 服务器端渲染（注意事项）

- 我们知道getStaticProps和getStaticPaths函数都是在build阶段运行，那么getServerSideProps函数的运行时机是怎么样的呢？
- getServerSideProps运行时机：
  - 首先，getServerSideProps仅在服务器端运行，从不在浏览器上运行。
  - 如果页面使用 getServerSideProps，则：
    - ✓ 当直接通过URL请求此页面时，getServerSideProps在请求时运行，并且此页面将使用返回的 props 进行预渲染
    - ✓ 当通过 Link 或 router切换页面来请求此页面时，Next.js 向服务器发送 API 请求，服务器端会运行getServerSideProps
- 什么时候该使用getServerSideProps？
  - 当页面显示的数据**必须在请求时获取的**，才应使用getServerSideProps 。
    - ✓ 如：页面需要显示**经常更新的数据**，并且页面内容会在每次请求时发生变化。
  - 如过页面使用了getServerSideProps函数，那么该页面将在客户端请求时，会在服务器端渲染，页面默认不会缓存。
  - 如果不需要在客户端每次请求时获取页面数据，那么应该考虑在 **客户端动态渲染 或 getStaticProps**.

# 增量静态再生 (ISR)

- Next.js 除了支持静态生成 和 服务器端渲染，Next.js 还允许在构建网站后创建或更新静态页面。
- 这种模式称为：增量静态再生 (Incremental Static Regeneration)，简称 (ISR)。
- 比如我们继续实现前面的案例：
  - 发起网络请求拿到页面书籍列表的数据，并展示。这次我们使用ISR渲染模式，让服务器端每隔5s重新生成静态书籍列表页面

```
export async function getStaticProps(context: any) {  
  let count = Math.floor(Math.random() * 10) + 1;  
  const res = await fetchBooks(count);  
  return {  
    props: {  
      books: res.data.book,  
    },  
    revalidate: 5, // In seconds  
  };  
}
```

# 客户端渲染 (CSR)

- Next.js 除了支持在服务器端获取数据，同时也是支持在客户端获取数据，并在客户端进行渲染。
- 在客户端获取数据，需要在页面组件或普通组件的 `useEffect` 函数中获取，比如：

```
const Category: FC<IProps> = (props) => {  
  const { datas = [] } = props;  
  
  // 在客户端获取数据  
  useEffect(() => {  
    axios.get("http://codercba.com:9060/juanpi/api/homeInfo").then((res) => {  
      console.log(res);  
    });  
  });  
  
  return (  
    <div className={styles.category}>  
      <div>Category</div>  
    </div>  
  );  
};
```

# Next.js 集成 Redux

## ■ 安装依赖库

❑ `npm i next-redux-wrapper --save`

- ✓ 可以避免在访问服务器端渲染页面时store的重置
- ✓ 该库可以将服务器端redux存的数据，同步一份到客户端上
- ✓ 该库提供了HYDRATE调度操作
  - 当用户访问动态路由或后端渲染的页面时，会执行Hydration来保持两端数据状态一致
  - 比如：每次当用户打开使用了`getStaticProps`或`getServerSideProps`函数生成的页面时，HYDRATE将执行调度操作。

❑ `npm i @reduxjs/toolkit react-redux --save`

# 创建counter模块的reducer

■ 我们先创建counter模块的reducer：通过createSlice创建一个slice。

■ createSlice主要包含如下几个参数：

□ name：用来标记slice的名词

✓ redux-devtool中会显示对应的名词；

□ initialState：第一次初始化时的值；

□ reducers：相当于之前的reducer函数

✓ 对象类型，并且可以添加很多的函数；

✓ 函数类似于 redux 原来reducer中的一个case语句；

□ extraReducer：添加更多额外reducer处理other action

□ createSlice 返回值是一个对象

✓ 对象包含所有的 actions 和 reducer；

```
import { HYDRATE } from "next-redux-wrapper";
const counterSlice = createSlice({
  name: "counter",
  initialState: {
    count: 100,
  },
  reducers: {
    changeCountAction(state, action) {
      console.log("changeCountAction=>", action.type, action.payload);
      state.count += action.payload;
    },
  },
  extraReducers: (builder) => {
    builder.addCase(HYDRATE, (state, action: any) => {
      console.log(state.count); // 当前模块的state
      console.log(action.type); // action types: __NEXT_REDUX_WRAPPER_HYDRATE__
      console.log("HYDRATE", action.payload); // payload 拿到整个rootState
      return {
        ...state,
        ...action.payload.counter, // 更新模块数据
      };
    });
  },
});

// 异步的action
export const fetchHomeDataAction = createAsyncThunk(
  > fetchSubj
```

# store的创建

■ **configureStore**用于创建store对象，常见参数如下：

- ❑ **reducer**，将slice中的reducer可以组成一个对象传入此处；
- ❑ **middleware**：可以使用参数，传入其他的中间件（自行了解）；
- ❑ **devTools**：是否配置devTools工具，默认为true；

```
import { configureStore } from "@reduxjs/toolkit";  
import { createWrapper } from "next-redux-wrapper";  
import counterReducer from "../modules/counter";
```

```
const store = configureStore({  
  reducer: {  
    counter: counterReducer,  
  },  
  devTools: true,  
});  
const makeStore = () => store;
```

```
export const wrapper = createWrapper(makeStore);
```



## ■ 在app.js中将store接入应用：

- **Provider**，内容提供者，给所有的子或孙子组件提供store对象；
- **store**：使用useWrappedStore函数导出的store对象；

```
import { wrapper } from "../store";
export default function App({ Component, ...rest }: AppProp) {
  const { store, props } = wrapper.useWrappedStore(rest);
  return (
    <Provider store={store}>
      <Component {...props.pageProps} />
    </Provider>
  );
}
```

# 开始使用store

■ 在函数式组件中可以使用 react-redux 提供的 Hooks API 连接、操作 store。

□ **useSelector** 允许你使用 selector 函数从 store 中获取数据 (root state) 。

□ **useDispatch** 返回 redux store 的 dispatch 引用。你可以使用它来 dispatch actions。

□ **useStore** 返回一个 store 引用, 和 Provider 组件引用完全一致。

```
const ProfileID: any = memo(function () {  
  /** 1.从redux获取的数据 */  
  const { count2 } = useSelector(  
    (state: any) => ({  
      count2: state.counter.count,  
    } ),  
    shallowEqual  
  );  
  /** 2.发送网络请求 */  
  const dispatch = useDispatch();  
  function handleChangeCount() {  
    dispatch(fetchHomeDataAction(1)); //  
  }  
}
```

```
// 在服务器端使用redux, 服务器端存储状态会触发 hydration, 同步到客户端  
export const getServerSideProps = wrapper.getServerSideProps((store) => {  
  // 下面参数 是客户端请求时传递过来的  
  async ({ req, res, query, params, resolvedUrl, locales }) => {  
    console.log("params=>", params);  
    const id = params?.id;  
    // 1.根据客户端传递的参数, 发起网络请求, 拿到数据存到服务器端的store中, 同时客户端  
    if (typeof id === "string") {  
      await store.dispatch(fetchHomeDataAction(id));  
      console.log("State on server", store.getState());  
    }  
    return {  
      props: {  
        id,  
      },  
    };  
  }  
});
```

# Redux Toolkit异步Action操作

- 在之前的开发中，我们通过redux-thunk中间件让dispatch中可以进行异步操作。
- Redux Toolkit默认已经给我们继承了Thunk相关的功能：createAsyncThunk

```
export const getHomeMultidataAction = createAsyncThunk("home/multidata", async (payload, extraInfo) => {  
  const res = await axios.get("http://123.207.32.32:8000/home/multidata")  
  return res.data.data  
})
```

- 当createAsyncThunk创建出来的action被dispatch时，会存在三种状态：

- ❑ **pending**：action被发出，但是还没有最终的结果；
- ❑ **fulfilled**：获取到最终的结果（有返回值的结果）；
- ❑ **rejected**：执行过程中有错误或者抛出了异常；

- 我们可以在createSlice的extraReducer中监听这些结果：见右图

```
extraReducers: {  
  [getHomeMultidataAction.pending](state, action) {  
    console.log("getHomeMultidataAction pending", action)  
  },  
  [getHomeMultidataAction.fulfilled](state, { payload }) {  
    console.log("getHomeMultidataAction fulfilled", payload)  
    state.banners = payload.banner.list  
    state.recommends = payload.recommend.list  
  },  
  [getHomeMultidataAction.rejected](state, action) {  
  
  }  
}
```

## ■ 网页云音乐-商城

- next-music-web
  - .next
  - assets
  - base-ui
  - components
  - node\_modules
  - pages
  - public
  - service
  - store
  - styles
- .eslintrc.json
- .gitignore
- next-env.d.ts
- next.config.js
- package-lock.json
- package.json
- README.md
- tsconfig.json



# 项目需安装的依赖

## ■ 样式

- ❑ `npm i normalize.css --save`
- ❑ `npm i sass --save`
- ❑ `npm i classnames --save`

## ■ Redux And Toolkit

- ❑ `npm i next-redux-wrapper --save`
- ❑ `npm i @reduxjs/toolkit react-redux --save`

## ■ Axios

- ❑ `npm i axios --save`

## ■ AntDesign

- ❑ `npm i antd --save`
- ❑ `npm i -D @types/antd`

```
"dependencies": {  
  "@reduxjs/toolkit": "^1.9.0",  
  "@types/node": "18.11.9",  
  "@types/react": "18.0.25",  
  "@types/react-dom": "18.0.9",  
  "antd": "^4.24.3",  
  "axios": "^1.1.3",  
  "classnames": "^2.3.2",  
  "eslint": "8.27.0",  
  "eslint-config-next": "13.0.3",  
  "next": "13.0.3",  
  "next-redux-wrapper": "^8.0.0",  
  "normalize.css": "^8.0.1",  
  "react": "18.2.0",  
  "react-dom": "18.2.0",  
  "react-redux": "^8.0.5",  
  "sass": "^1.56.1",  
  "typescript": "4.9.3"  
}
```

# 安装 Ant Design 5

■ Ant Design 官网: <https://ant.design/docs/react/use-in-typescript-cn>

■ Next.js 应用安装Ant Design5

□ 第一步:

✓ npm i antd

✓ npm i -D @types/antd

□ 第二步:

✓ 统一样式风格

□ 第三步:

✓ 就可以开始使用了

■ 注意事项:

□ antd 默认支持基于 ES modules 的 tree shaking

□ 直接引入 `import { Button } from 'antd'` 就会有按需加载的效果

```
import React, { FC } from 'react';
import { Button } from 'antd';
import 'antd/dist/reset.css';
import './App.css';

const App: FC = () => (
  <div className="App">
    <Button type="primary">Button</Button>
  </div>
);

export default App;
```

# 项目打包和部署

## ■ 项目打包

- 执行 `npm run build` (整个项目是部署产物)
- 执行 `npm run start` 本地预览效果

## ■ 使用Node部署

- 运行: `npm run start`
- 指定端口: `PORT=9090 npm run start`

## ■ 使用PM2部署 (推荐)

- 项目根目录运行: `pm2 start npm --name "music-mall" -- run start`
  - ✓ `pm2 start npm` : 在当前目录执行npm
  - ✓ `--name` : 指定应用程序的名称
  - ✓ `--` : 后面所有参数会传递给 npm 程序
- OR: `pm2 start "npm run start" --name music-mall`

All option passed after `--` will be passed as argument to the app:

```
$ pm2 start api.js -- arg1 arg2
```



## ■ PM2 常用命令

# 命名进程

```
pm2 start app.js --name my-api
```

# 显示所有进程状态

```
pm2 list
```

# 停止指定的进程,

```
pm2 stop 0
```

# 停止所有进程

```
pm2 stop all
```

# 重启所有进程

```
pm2 restart all
```

# 重启指定的进程( id )

```
pm2 restart 0
```

# 杀死指定的进程

```
pm2 delete 0
```

# 杀死全部进程

```
pm2 delete all
```

#后台运行pm2, 启动4个app.js, 实现负载均衡

```
pm2 start app.js -i 4
```

```
module.exports = {
  apps: [
    {
      name: "music-mall",
      exec_mode: "fork", // fork 模式, 创建一个 process_child 子进程
      increment_var: "PORT", // 端口自增长
      instances: 3, // 三个实例, 分别运行在: 8888, 8889, 8890 端口
      script: "npm run start",
      env: {
        PORT: 8888,
      },
    },
  ],
};
```

```
module.exports = {
  apps: [
    {
      name: "music-next",
      cwd: "./",
      script: "node_modules/next/dist/bin/next",
      args: "start",
      instances: "max",
      exec_mode: "cluster",
    },
  ],
};
```