

Vue3 + Nuxt3 (核心)

刘军 liujun

目录

content



1 嵌套路由和中间件

2 Nuxt自定义布局

3 页面渲染模式

4 Nuxt3 插件开发

5 Lifecycle Hooks

6 获取数据、API接口

7 useState 和 Pinia

嵌套路由

■ Nuxt 和 Vue一样，也是支持嵌套路由的，只不过在Nuxt中，**嵌套路由也是根据目录结构和文件的名称自动生成。**

■ 编写嵌套路由步骤：

- 1.创建一个**一级路由**，如：parent.vue
- 2.创建一个与一级路由同名同级的文件夹，如： parent
- 3.在parent文件夹下，创建一个嵌套的**二级路由**
 - ✓ 如： parent/child.vue， 则为一个二级路由页面
 - ✓ 如： parent/index.vue 则为**二级路由默认的页面**
- 4.需要在parent.vue中添加 NuxtPage 路由占位

```
> -| pages/  
> ---| parent/  
> -----| child.vue  
> ---| parent.vue
```

```
[  
  {  
    path: '/parent',  
    component: '~/pages/parent.vue',  
    name: 'parent',  
    children: [  
      {  
        path: 'child',  
        component: '~/pages/parent/child.vue',  
        name: 'parent-child'  
      }  
    ]  
  }  
]
```

路由中间件 (middleware)

■ Nuxt 提供了一个可定制的 **路由中间件**，用来**监听路由的导航**，包括：**局部和全局监听**（支持再服务器和客户端执行）

■ 路由中间件分为三种：

□ 匿名（或内联）路由中间件

✓ 在页面中使用 **definePageMeta** 函数定义，可监听局部路由。当注册多个中间件时，会按照注册顺序来执行。

□ 命名路由中间件

✓ 在 **middleware 目录**下定义，并会自动加载中间件。**命名规范 kebab-case**

□ 全局路由中间件（优先级比前面的高，支持两端）

✓ 在**middleware 目录**中，**需带.global后缀的文件**，每次路由更改都会自动运行。

```
export default defineNuxtRouteMiddleware((to, from) => {
  // isAuthenticated() is an example method verifying if
  if (isAuthenticated() === false) {
    return navigateTo('/login')
  }
})
```

```
<script setup>
definePageMeta({
  // define middleware as a function
  middleware: [
    function (to, from) {
      const auth = useState('auth')

      if (!auth.value.authenticated) {
        return navigateTo('/login')
      }

      return navigateTo('/checkout')
    }
  ],
  // ... or a string
  middleware: 'auth'

  // ... or multiple strings
  middleware: ['auth', 'another-named-middleware']
})
</script>
```

路由验证(validate)

■ Nuxt支持对每个页面路由进行验证，我们可以通过definePageMeta中的validate属性来对路由进行验证。

□ validate属性接受一个回调函数，回调函数中以 route 作为参数

□ 回调函数的返回值支持：

✓ 返回 bool 值来确定是否放行路由

➢ true 放行路由

➢ false 默认重定向到内置的 404 页面

✓ 返回对象

➢ { statusCode: 401 } // 返回自定义的 401 页面，验证失败

■ 路由验证失败，可以自定义错误页面：

□ 在项目根目录（不是pages目录）新建 error.vue

```
<script setup>
definePageMeta({
  validate: async (route) => {
    const nuxtApp = useNuxtApp()
    // Check if the id is made up of digits
    return /^d+$/.test(route.params.id)
  }
})
</script>
```

```
<template>
  <button @click="handleError">Clear errors</button>
</template>

<script setup>
const props = defineProps({
  error: Object
})

const handleError = () => clearError({ redirect: '/' })
</script>
```

布局 (Layout)

- Layout布局是**页面的包装器**，可以将多个**页面共性东西**抽取到 Layout布局 中。
 - 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布局是使用**<slot>组件**来显示**页面中的**内容
- Layout布局有两种使用方式：
 - 方式一：默认布局
 - ✓ 在layouts目录下新建默认的布局组件，比如：**layouts/default.vue**
 - ✓ 然后在app.vue中通过**<NuxtLayout>**内置组件来使用
 - 方式二：自定义布局 (Custom Layout)
 - ✓ 继续在layouts文件夹下新建 Layout 布局组件，比如: layouts/custom-layout.vue
 - ✓ 然后在app.vue中给<NuxtLayout>内置组件 **指定name属性** 的值为：custom-layout
 - 也支持在页面中使用 **definePageMeta** 宏函数来指定 layout 布局

layouts/default.vue

```
<template>
  <div>
    <AppHeader />
    <slot />
    <AppFooter />
  </div>
</template>
```

渲染模式

■ 浏览器 和 服务器都可以解释 JavaScript 代码，都可以将 Vue.js 组件呈现为 HTML 元素。此过程称为渲染。

□ 在客户端将 Vue.js 组件呈现为 HTML 元素，称为：客户端渲染模式

□ 在服务器将 Vue.js 组件呈现为 HTML 元素，称为：服务器渲染模式

■ 而Nuxt3是支持多种渲染模式，比如：

□ 客户端渲染模式（CSR）： 只需将 `ssr` 设置为 `false`

□ 服务器端渲染模式（SSR）： 只需将 `ssr` 设置为 `true`

□ 混合渲染模式（SSR | CSR | SSG | SWR）： 需在 `routeRules` 根据每个路由动态配置渲染模式（beta版本）

```
export default defineNuxtConfig({
  routeRules: {
    // Static page generated on-demand, revalidates in background
    '/blog/**': { swr: true },
    // Static page generated on-demand once
    '/articles/**': { static: true },
    // Render these routes with SPA
    '/admin/**': { ssr: false },
  }
})
```

Nuxt 3 starting from rc.12 comes with the public beta for route rules and hybrid rendering support. Using route rules you can define rules for a group of nuxt routes, change rendering mode or assign a cache strategy based on route! Nuxt server will automatically register corresponding middleware and wrap routes with cache handlers using nitro caching layer. Whenever possible, route rules will be automatically applied to the deployment platform's native rules (currently Netlify and Vercel are supported).

`static` and `swr` - `static` enables a single (on-demand) build; `swr` enables a static build, that lasts for a configurable TTL. (currently enables full incremental static generation on Netlify, with Vercel coming soon)

Nuxt插件 (Plugins)

■ Nuxt3支持自定义插件进行扩展，创建插件有两种方式：

□ 方式一：直接使用 `useNuxtApp()` 中的 `provide(name, value)` 方法直接创建，比如：可在App.vue中创建

✓ `useNuxtApp` 提供了访问 Nuxt 共享运行时上下文的方法和属性（两端可用）：`provide`、`hooks`、`callhook`、`vueApp`等

□ 方式二：在 `plugins` 目录中创建插件（推荐）

✓ 顶级和子目录index文件写的插件会在创建Vue应用程序时会自动加载和注册

✓ `.server` 或 `.client` 后缀名插件，可区分服务器端或客户端，用时需区分环境

■ 在 `plugins` 目录中创建插件

□ 1.在 `plugins` 目录下创建 `plugins/price.ts` 插件

□ 2.接着 `defineNuxtPlugin` 函数创建插件，参数是一个回调函数

□ 3.然后在组件中使用 `useNuxtApp()` 来拿到插件中的方法

■ 注意事项：

□ 插件注册顺序可以通过在文件名前加上一个数字来控制插件注册的顺序

✓ 比如：`plugins/1.price.ts`、`plugins/2.string.ts`、`plugins/3.date.ts`

```
export default defineNuxtPlugin(() => {
  return {
    provide: {
      hello: (msg: string) => `Hello ${msg}!`
    }
  }
})
```

```
<script setup lang="ts">
// alternatively, you can also
const { $hello } = useNuxtApp()
</script>
```

App Lifecycle Hooks

■ 监听App的生命周期的Hooks:

- App Hooks 主要可由 [Nuxt 插件](#) 使用 hooks 来监听 生命周期, 也可用于 Vue 组合 API。
- 但是, 如在组件中编写hooks来监听, 那 create和setup hooks就监听不了, 因为这些hooks已经触发完了监听。

■ 语法: `nuxtApp.hook(app:created, func)`

Hook	Arguments	Environment	Description
<code>app:created</code>	<code>vueApp</code>	Server & Client	Called when initial <code>vueApp</code> instance is created.
<code>app:error</code>	<code>err</code>	Server & Client	Called when a fatal error occurs.
<code>app:error:cleared</code>	<code>{ redirect? }</code>	Server & Client	Called when a fatal error occurs.
<code>app:data:refresh</code>	<code>keys?</code>	Server & Client	(internal)
<code>vue:setup</code>	-	Server & Client	(internal)
<code>vue:error</code>	<code>err, target, info</code>	Server & Client	Called when a vue error propagates to the root component. Learn More .
<code>app:rendered</code>	<code>renderContext</code>	Server	Called when SSR rendering is done.
<code>app:redirected</code>	-	Server	Called before SSR redirection.
<code>app:beforeMount</code>	<code>vueApp</code>	Client	Called before mounting the app, called only on client side.
<code>app:mounted</code>	<code>vueApp</code>	Client	Called when Vue app is initialized and mounted in browser.
<code>app:suspense:resolve</code>	<code>appComponent</code>	Client	On Suspense resolved event.
<code>link:prefetch</code>	<code>to</code>	Client	Called when a <code><NuxtLink></code> is observed to be prefetched.
<code>page:start</code>	<code>pageComponent?</code>	Client	Called on Suspense pending event.
<code>page:finish</code>	<code>pageComponent?</code>	Client	Called on Suspense resolved event.
<code>page:transition:finish</code>	<code>pageComponent?</code>	Client	After page transition onAfterLeave event.

■ 客户端渲染

- `beforeCreate` → use `setup()`
- `created` → use `setup()`
- `beforeMount` → `onBeforeMount`
- `mounted` → `onMounted`
- `beforeUpdate` → `onBeforeUpdate`
- `updated` → `onUpdated`
- `beforeDestroy` → `onBeforeUnmount`
- `destroyed` → `onUnmounted`
- `errorCaptured` → `onErrorCaptured`

■ 服务器端渲染

□ `beforeCreate` -> `setup`

□ `created`

组件生命周期钩子

因为没有任何动态更新，所以像 `mounted` 或者 `updated` 这样的生命周期钩子不会在 SSR 期间被调用，而只会在客户端运行。只有 `beforeCreate` 和 `created` 这两个钩子会在 SSR 期间被调用。

你应该避免在 `beforeCreate` 和 `created` 中使用会产生副作用且需要被清理的代码。这类副作用的常见例子是使用 `setInterval` 设置定时器。我们可能会在客户端特有的代码中设置定时器，然后在 `beforeUnmount` 或 `unmounted` 中清除。然而，由于 `unmount` 钩子不会在 SSR 期间被调用，所以定时器会永远存在。为了避免这种情况，请将含有副作用的代码放到 `mounted` 中。

获取数据

■ 在Nuxt中数据的获取主要是通过下面4个函数来实现（支持Server和Client）：

□ 1.`useAsyncData`(key, func):专门解决异步获取数据的函数，**会阻止**页面导航。

✓ 发起异步请求需用到 `$fetch` 全局函数（类似Fetch API）

➢ `$fetch(url, opts)`是一个类原生fetch的**跨平台请求库**

□ 2.`useFetch`(url, opts)：用于获取任意的URL地址的数据，**会阻止**页面导航

✓ 本质是 `useAsyncData(key, () => $fetch(url, opts))` 的语法糖。

□ 3.`useLazyFetch`(url, opts):用于获取任意URL数据，**不会阻止**页面导航

✓ 本质和 `useFetch` 的 `lazy` 属性设置为 `true` 一样

□ 4.`useLazyAsyncData`(key, func):专门解决异步获取数据的函数。**不会阻止**页面导航

✓ 本质和`useAsyncData`的`lazy`属性设置为`true`一样

■ 注意事项：

□ 这些函数只能用在 `setup` or `Lifecycle Hooks` 中。

```
<script setup>
const { data: count } = await useFetch('/api/count')
</script>

<template>
  Page visits: {{ count }}
</template>
```

```
<script setup>
const { pending, data: posts } = useLazyFetch('/api/posts')
watch(posts, (newPosts) => {
  // Because posts starts out null, you will not have access
  // to its contents immediately, but you can watch it.
})
</script>
```

```
<script setup>
const { data } = await useAsyncData('count', () => $fetch('/api/count'))
</script>

<template>
  Page visits: {{ data }}
</template>
```

useFetch vs axios

■ 获取数据Nuxt推荐使用 useFetch 函数，为什么不是 axios ？

- useFetch 底层调用的是\$fetch函数，该函数是基于[unjs/ohmyfetch](https://github.com/unjs/ohmyfetch)请求库，并与原生的Fetch API有者相同API
- [unjs/ohmyfetch](https://github.com/unjs/ohmyfetch) 是一个跨端请求库： A better fetch API. Works on node, browser and workers
 - ✓ 如果运行在服务器上，它可以智能的处理对 API接口的直接调用。
 - ✓ 如果运行在客户端行，它可以对后台提供的 API接口 正常的调用（类似 axios），当然也支持第三方接口的调用
 - ✓ 会自动解析响应和对数据进行字符串化
- useFetch 支持智能的类型提示和智能的推断 API 响应类型。
- 在setup中用useFetch获取数据，会减去客户端重复发起的请求。

■ useFetch (url, options) 语法

- 参数
 - ✓ url: 请求的路径
 - ✓ options: 请求配置选项
 - method、query (别名 params)、body、headers、baseURL
 - onRequest、 onResponse、 lazy....
- 返回值: data, pending, error, refresh

```
const { data, pending, error, refresh } = await useFetch('/api/auth/login', {
  onRequest({ request, options }) {
    // Set the request headers
    options.headers = options.headers || {}
    options.headers.authorization = '...'
  },
  onRequestError({ request, options, error }) {
    // Handle the request errors
  },
  onResponse({ request, response, options }) {
    // Process the response data
    return response._data
  },
  onResponseError({ request, response, options }) {
    // Handle the response errors
  }
})
```

useFetch的封装

■ 封装useFetch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useFetch发起网络请求
- 4. 添加TypeScript类型声明

```
class HYRequest {  
  request<T = any>(  
    url: string,  
    method: Methods,  
    data?: any,  
    options?: UseFetchOptions<T>  
  ): Promise<AsyncData<T, Error>> {  
    let newOptions = {  
      method: method || "GET",  
      baseUrl: BASE_URL,  
      ...options,  
    };  
    if (method === "GET") {  
      newOptions.params = data || {}; // Alias for query  
    } else {  
      newOptions.body = data || {};  
    }  
    return new Promise((resolve, reject) => {  
      useFetch<T>(url, newOptions as any)  
        .then((res) => {  
          resolve(res as AsyncData<T, Error>);  
        })  
        .catch((error) => {  
          reject(error);  
        });  
    });  
  }  
}
```

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：<https://d.book118.com/285132140300012104>