16 | 单页面应用：如何理解和实现单页面应用开发？

2022-12-30 杨文坚 来自北京

《Vue 3 企业级项目实战课》

课程介绍 >

讲述：杨文坚

时长 17:13　大小 15.72M

你好，我是杨文坚。

今天，我们进入项目功能页面的开发。上一讲说过，运营搭建平台，后台核心功能是搭建功能。不过，搭建功能并不是简单的一个网页就能承载的。

因为搭建功能不是单一的功能点，而是一整个功能链路，包括：搭建的物料注册、物料列表管理、物料详情等物料管理的功能点，搭建页面的布局设计、搭建结果列表、搭建发布流程、搭建版本等搭建管理功能点，还有一些管理权限的控制功能点。最终涉及的功能链路，会是很庞大的功能点集合。

同时，每个功能点基本都是需要一个网页来承载，这就需要多个网页来覆盖整个搭建后台的功能链路。

现在问题来了，一个项目的后台功能链路，页面非常多，而且，操作一个功能链路时，要跳转多个页面，每次跳转都要重新经历 HTML 页面请求响应、JavaScript / CSS / 图片等静态资源加载、服务端接口加载，会经历比较长时间的等待。

那么，有没有方式可以减少页面跳转时页面上 HTTP 的请求，提升不同页面在跳转或者切换过程中的加载和渲染速度呢？答案是有的，我们可以把功能链路的多页面形式变成单页面形式，也就是所谓的单页面应用。

什么是单页面应用？

单页面应用，英文的专业术语是 Single-Page-Application，简称“SPA”，和“多页面应用”相对。所以，单页面应用一般要和多页面对比理解，我们来看一看。

在传统的多页面应用中，每次切换页面的跳转过程，一般分成五个阶段：

- 第一阶段，浏览器 URL 改变发起新页面的 HTTP 请求；
- 第二阶段，服务端接受到请求，通过路由响应请求的 HTML 结果；
- 第三阶段，浏览器接受到 HTML 响应并加载 HTML 结果；
- 第四阶段，浏览器等待加载 HTML 里依赖的资源（JavaScript、CSS、图片等文件）；
- 第五阶段，HTML 页面静态资源加载完后，渲染对应内容。

而单页面应用，每次切换页面的过程，可以缩短成两个执行阶段：

- 第一阶段，浏览器 URL 改变，发起新页面静态资源请求（JavaScript、CSS、图片等文件）；
- 第二阶段，新页面资源加载完后，渲染对应内容。

所以，在单页面应用跳转新页面过程中，其实并没有发起完整的新 HTML 页面请求，只是加载对应的资源，渲染修改指定的页面内容。所以，相比多页面应用，单页面应用，减少了服务端路由操作，也就减少了多个 HTTP 请求和响应的过程，最后就缩短了等待时间。

这就说明，单页面应用是通过浏览器层面控制 URL，代替服务端路由进行页面跳转（或切换）的控制处理，减少 HTTP 请求，提升页面切换速度。这里，浏览器控制 URL 变化的能

力，叫做浏览器路由，也叫前端路由，跟多页面应用的服务器路由是相对的。所以，**单页面应用的核心能力就是浏览器路由能力。**

那么浏览器路由的原理是什么？或者说要实现浏览器路由，有哪些技术方案呢？

实现浏览器路由有哪些技术方案？

浏览器路由的功能，就是当浏览器要控制 URL 变化时，保持页面不会重新请求 HTTP 加载。那么，浏览器有哪些特性可以实现类似的功能呢？

目前，浏览器实现类似功能有两种方式，一种是 **URL 的 hash 值控制**，另外是**浏览器的 history 控制**。

先来看 URL 的 hash 控制方案。这里说的 URL 的 hash，就是 URL 末尾带上 # 字符开头的字符，例如下述链接里，`#title-123` 就是 URL 里的 hash 值。

复制代码

```
https://example.com/page/xxx.html?a=1&b=2#title-123
```

URL 的 hash，传统的作用是来标记页面某个位置的锚点。比如页面某个子标题的位置标记了这个 hash 值，例如“title-123”，URL 带上这个 hash 值后，`#title-123` 会自动滑动到子标题位置。

不过，当页面不存在这个 hash 值标记的锚点位置时，URL 带上的这个 hash 就没有什么作用了。即使强行修改 URL 这个 hash 值，只要页面不存在锚点位置，页面就不会发生任何变化，也不会发生任何页面跳转。

这时候，我们就可以利用这个 URL 的 hash 非锚点特性。**当 hash 变化时候，用 JavaScript 监听这个变化，然后加载指定资源渲染页面，不需要重新请求服务端刷新整个页面。**

怎么基于 hash 实现浏览器路由呢？看代码：

复制代码

```
const viewDom = document.querySelector('#view');
const linksDom = document.querySelector('#links');
```

```
const viewMap = {
  '#/': '<h1>Default View</h1>',
  '#/home': '<h1>Home View</h1>',
  '#/about': '<h1>About View</h1>',
  '#/page/hello': '<h1>Page Hello View</h1>',
  '#/page/001': '<h1>Page 001 View</h1>',
  '#/page/002': '<h1>Page 002 View</h1>',
  '#/404': '<h1>404 View</h1>'
};

const renderLinks = () => {
  const linkList = Object.keys(viewMap);
  const htmlList = [];
  linkList.forEach((link) => {
    htmlList.push(
      `<li><a class="${
        link === location.hash ? 'active' : ''
      }" href="${link}">${link}</a></li>`
    );
  });
  linksDom.innerHTML = htmlList.join('');
};

const renderView = () => {
  // 获取当前URL的hash
  // 渲染对应hash的页面内容
  const hash = window.location.hash;
  const viewPath = hash.split('?')[0];
  const viewHtml = viewMap[viewPath] || viewMap['#/404'];
  viewDom.innerHTML = viewHtml;
};

window.addEventListener('hashchange', () => {
  // 监听URL的 hash 变化时候
  // 重新渲染页面内容
  renderView();
  renderLinks();
});

renderLinks();
renderView();
```

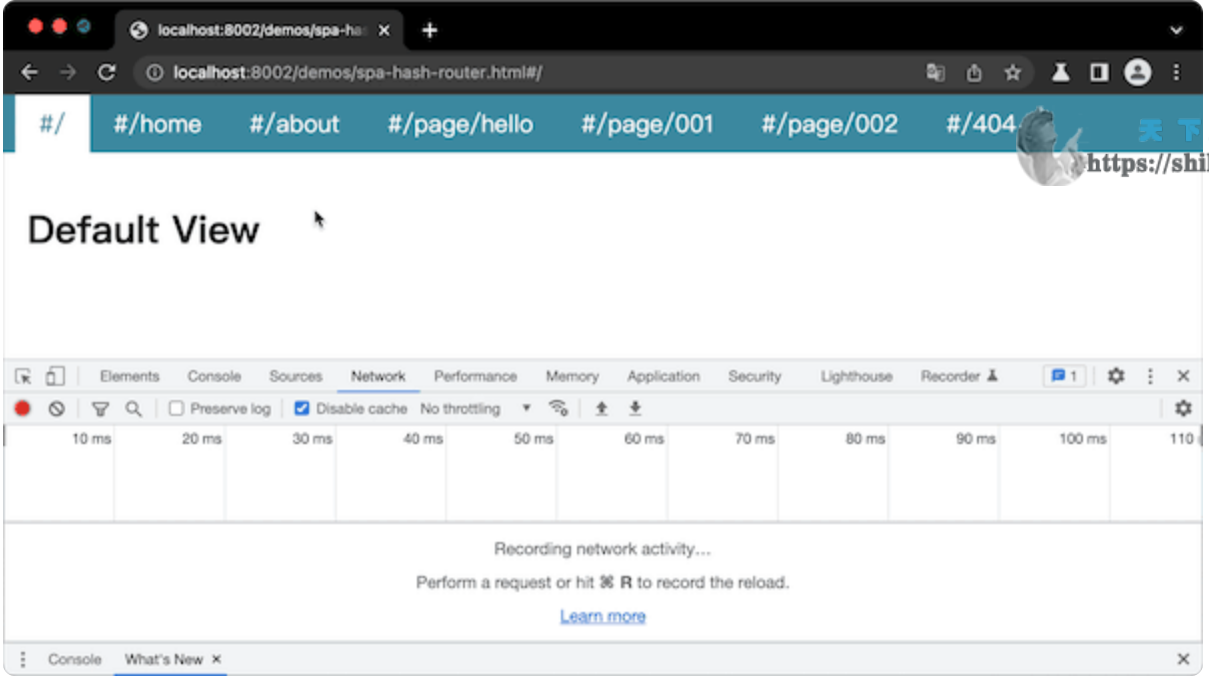

配套 HTML 代码：

复制代码

```
<html>
  <head>
    <meta charset="utf-8" />
    <style>
      html,body { margin: 0; padding: 0 }
```

```
      #links {
        list-style: none;
        margin: 0;
        padding: 0 10px;
        flex-direction: row;
        display: flex;
        background: #3b889f;
      }
      #links li a {
        display: inline-block;
        padding: 10px 20px;
        font-size: 20px;
        color: #ffffff;
        text-decoration: none;
      }
      #links li a.active {
        background: #ffffff;
        color: #3b889f;
      }
      #view {
        padding: 20px;
        margin: 0;
      }
    </style>
  </head>
  <body>
    <ul id="links"></ul>
    <div id="view"></div>
    <script type="module" src="./spa-hash-router.js"></script>
  </body>
</html>
```


代码最终的实现效果如动图所示：



我们在切换页面操作时候，只修改 URL 的 hash 变化，在控制台的请求面板上并不会触发页面刷新的请求。这就是基于 URL 的 hash 值，来实现的浏览器路由能力。

不过，估计你也发现了，URL 加了 hash 后有点长，不怎么优雅。那有更优雅的实现方式吗？

这时，我们就要提到另外一个浏览器路由实现方式了，基于浏览器的 history 特性，它也可以自定义浏览器 URL 变化，同时保持页面不会重新刷新加载。

history，顾名思义就是操作浏览器的访问历史。在浏览器里，访问历史就是记录或者控制 URL 的变化。按这个思路，**history 能通过新增一个“访问历史”来影响当前浏览器的 URL 变化，同时，这个变化的 URL 也是可以自定义的。**

所以，我们可以通过 history 来实现浏览器路由，看代码：

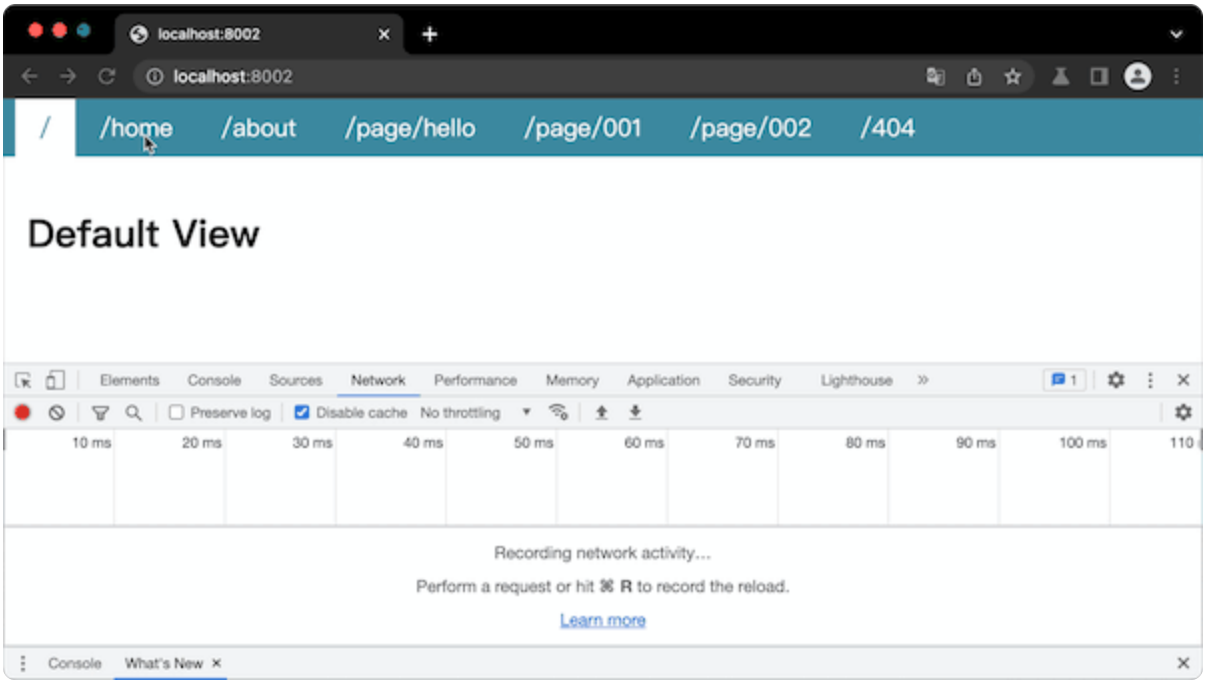
复制代码

```
const viewDom = document.querySelector('#view');
const linksDom = document.querySelector('#links');
const viewMap = {
  '/': '<h1>Default View</h1>',
  '/home': '<h1>Home View</h1>',
  '/about': '<h1>About View</h1>',
  '/page/hello': '<h1>Page Hello View</h1>',
  '/page/001': '<h1>Page 001 View</h1>',
  '/page/002': '<h1>Page 002 View</h1>',
  '/404': '<h1>404 View</h1>'
};
```

```
const renderLinks = () => {
  const linkList = Object.keys(viewMap);
  const htmlList = [];
  linkList.forEach((link) => {
    htmlList.push(
      `<li>
        <a class="${
          link === location.pathname ? 'active' : ''
        }" href="javascript:void(0)" data-href="${link}">${link}</a>
      </li>`
    );
  });
  linksDom.innerHTML = htmlList.join('');
};

const renderView = () => {
  const viewPath = window.location.pathname;
  const viewHtml = viewMap[viewPath] || viewMap['#/404'];
  viewDom.innerHTML = viewHtml;
};

// 人工注册history路由的pushState和replaceState监听
// 因为浏览器原生不支持
function registerHistoryListener(type) {
  const originFunc = window.history[type];
  const e = new Event(type);
  return function () {
    const result = originFunc.apply(this, arguments);
    e.arguments = arguments;
    window.dispatchEvent(e);
    return result;
  };
}

// 人工实现history路由的pushState事件监听
history.pushState = registerHistoryListener('pushState');
// 人工实现history路由的replaceState事件监听
history.replaceState = registerHistoryListener('replaceState');

// 监听history路由pushState
window.addEventListener('pushState', () => {
  renderLinks();
  renderView();
});
// 监听history路由replaceState
window.addEventListener('replaceState', () => {
  renderLinks();
  renderView();
});

linksDom.addEventListener('click', (e) => {
  // 链接点击之间时候
```

```
  // 拦截事件，然后做 history跳转操作
  const dom = e.target;
  const dataHref = dom.getAttribute('data-href');
  if (dataHref) {
    history.pushState({}, '', dataHref);
  }
});

renderLinks();
renderView();

```

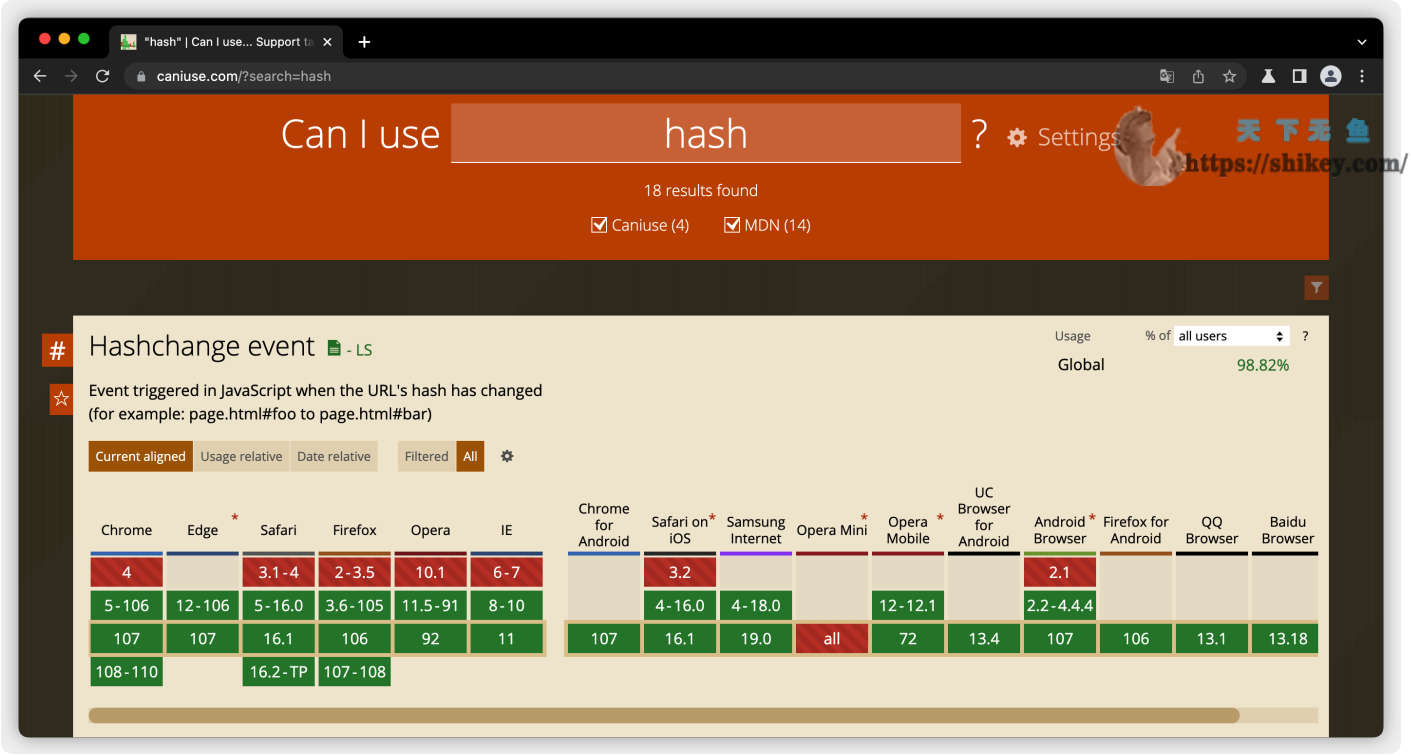

代码的最终实现效果，如动图所示：



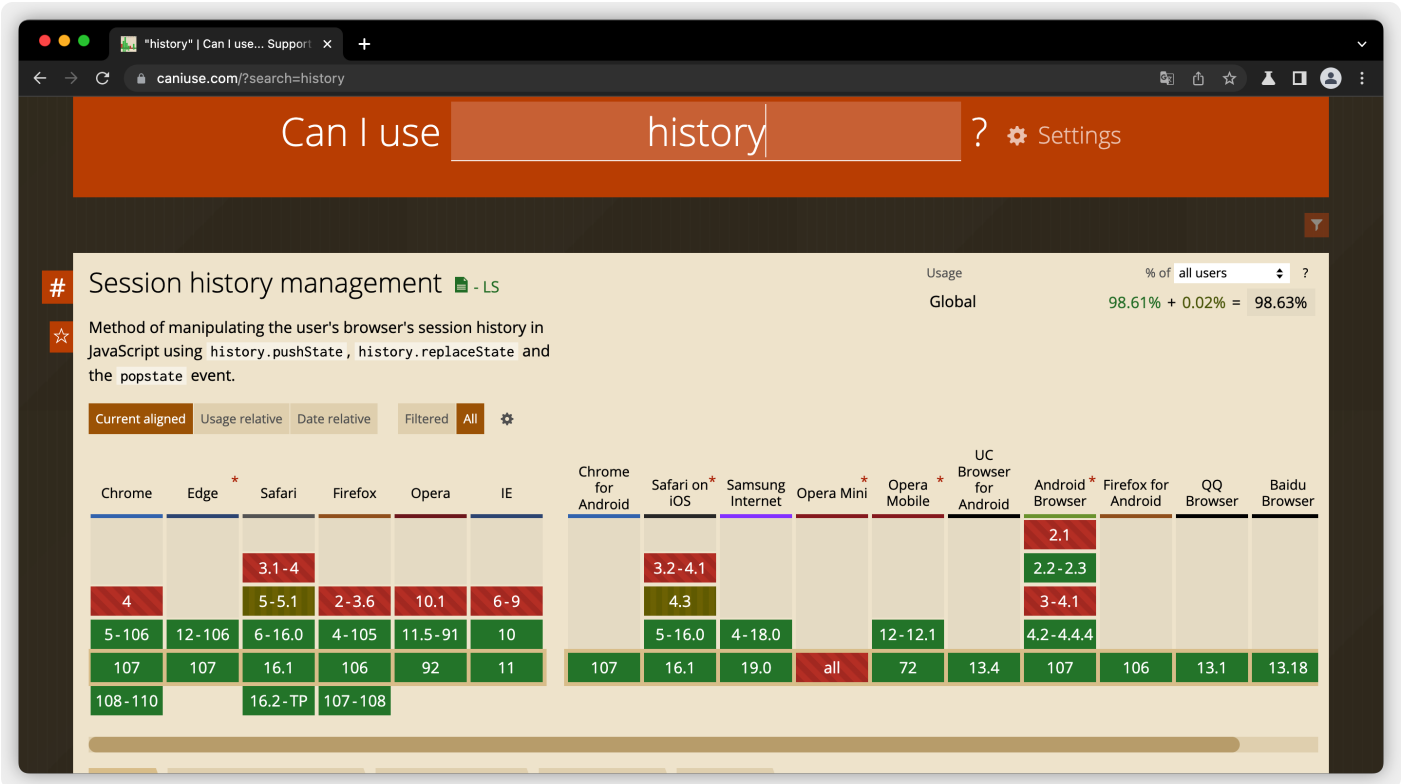
两种浏览器路由的实现原理和效果，我们都掌握了。该如何选择技术方案呢？

同样是浏览器路由的功能，技术原理不一样，就应该存在一定的技术差异和使用场景差异。

在实际开发中，hash 路由和 history 路由最大的差异就是浏览器兼容的差异。hash 路由是利用浏览器传统的 URL hash 锚点能力，这个特性，绝大部分浏览器都兼容，具体你可以通过 https://caniuse.com/ 这个网站来查询：



你看这张图片，连 IE8 这类浏览器都能兼容 hash 特性。但 history 特性的浏览器兼容性就比较弱了：



两种浏览器路由的兼容性差异，也导致了两者适用场景并不一样。如果单页面应用的业务需求要兼容 IE8 等低版本浏览器，那你就只能选择 hash 路由，如果对浏览器兼容要求不高，两者都可以。

除了浏览器这个兼容性差异，它们**对页面所在服务端的要求也不同**。hash 路由只改变 URL 的 hash 值，当前页面强制浏览器刷新时，还能保持在当前页面。但是，history 路由是真的改变了浏览器 URL 的路径 path，当前页面强制浏览器刷新时，要走一遍服务器请求，查询当前 URL 是否存在，所以要保持 history 路由改变的 URL 存在，服务器路由就必须跟前端路由同步。

说了这么多，我们简单总结下，用原生 JavaScript API 实现的浏览器路由，达到实现单页面应用的功能，大概有这几个核心原理：

- 利用 hash 或者 history 进行无刷新修改浏览器 URL；
- 监听 URL 变化，执行页面内容更新或切换；
- 如果用 history 路由，要考虑服务器路由的同步和浏览器的兼容。

看到这里，你是不是觉得单页面应用的实现很简单？但是同样的，实际开发项目中，光靠这三点原理是满足不了实际功能需求的。

实际功能需求比较复杂多样，比如，要求在页面切换时，传递数据到下一个页面，如何设计这个浏览器路由的数据传递呢？再比如，切换的页面不存在，如何做统一的 404 页面呢？这些都需要做很多前端路由的封装工作，不是光靠我们前面的几句代码就能实现的。

那么，在 Vue.js 3.x 项目里，我们如何实现完善的单页面应用呢？

Vue.js 3.x 如何实现完善的单页面应用？

Vue.js 3.x 官方提供了一个前端路由的模块 vue-router，方便开发者使用 Vue.js 开发单页面应用。接下来，我就给你演示一下如何基于 vue-router，实现常见的单页面应用的需求场景。

常见的功能场景主要有这么四种，我们会通过 vue-router 逐一实现：

- hash 路由（面向兼容 IE8 等低版本浏览器）
- history 路由（面向高版本浏览器和优雅 URL 路由）
- 动态路由和嵌套路由（动态 URL 和多级 URL）
- 主动路由跳转（代码主动触发页面跳转）

1. hash 路由

以 hash 路由形式，实现基本的单页面应用功能，具体代码如下所示：

复制代码

```
import { createRouter, createWebHashHistory } from 'vue-router';
import { createApp } from 'vue';
// 主应用
import App from './app.vue';

// 不同路由的页面视图
import HomeView from './views/home.vue';
import ListView from './views/list.vue';
import AboutView from './views/about.vue';

// 定义路由
const router = createRouter({
  linkActiveClass: 'active',
  // hash 路由配置
  history: createWebHashHistory('/'),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView
    },
    {
      path: '/list',
      name: 'list',
      component: ListView
    },
    {
      path: '/about',
      name: 'about',
      component: AboutView
    }
  ]
});

const app = createApp(App);
app.use(router);

app.mount('#app');

```

主应用 app.vue 的配置，代码如下：

```vue
<template>
  <div class="app">
    <header class="header">
      <RouterLink to="/">Home</RouterLink>
      <RouterLink to="/list">List</RouterLink>
      <RouterLink to="/about">About</RouterLink>
    </header>
    <main class="content">
      <RouterView />
    </main>
  </div>
</template>

<script setup lang="ts">
import { RouterLink, RouterView } from 'vue-router';
</script>
```


代码中， RouterLink 是 vue-router 提供的单页面应用链接组件，RouterView 是对应链接的视图，更多使用概念你可以查看官网。

具体的执行过程就是，创建 hash 路由到注册路由页面，再挂载到前端应用上。

2. history 路由

我们再切换成 history 的方式看看。vue-router 统一封装了 hash 路由和 history 路由的使用方式，开发者只要修改路由的选择方式，就可以直接切换两种路由形式：


```vue
// 定义路由
const router = createRouter({
  // 其它代码 ...

  // hash 路由配置
  // history: createWebHashHistory('/'),

  // history 路由配置
  history: createWebHistory('/'),

  // 其它代码 ...
}
```

我们前面说过，history 路由需要浏览器路由做兼容的要求，因为今天的代码例子是基于 Vite 来做的开发服务，所有页面请求都在 Vite 开发服务器中，默认指向项目页面的 index.html 页面，这样等于变相实现了服务端路由对 history 前端路由 URL 的同步。

3. 动态路由和嵌套路由

至于动态路由和嵌套路由，vue-router 提供了很丰富的 API，让开发者可以注册动态路由和路由嵌套，也就是在路由上携带参数和多级 URL 处理。

看路由配置代码，我用注释指明了动态路由和嵌套路由。动态路由是用“:”开头的变量，来标明路由中的动态数据。嵌套路由，是在路由里使用 children 来承载一个新的路由数组，作为当前路径后“接上”的子路由：

复制代码

```
// 定义路由
const router = createRouter({
  // 其它配置 ...
  routes: [
    // 动态路由
    {
      path: '/detail/:id',
      component: DetailView,

      // 嵌套路由
      children: [
        { path: '', component: DetailItemView }
      ]
    }
  ]
```

这里，动态路由的“:id”数据，可以在对应 View 的组件里通过 `$route.params.id` 拿到对应数据，例如 /detail/A001 可以得到 `$route.params.id = "/detail/A001"`。同时，动态路由里的子路由页面，会渲染内部的 component 组件。

4. 主动路由跳转

我们再看最后一个常见功能需求，主动路由跳转。

虽然 vue-router 提供了 RouterLink 组件来承载路由连接，但是这个组件需要人为点击才能触发路由跳转，无法实现代码主动触发跳转。所以， vue-router 又提供另外一个 API，也就是

useRouter 让开发者能主动操作路由实体，例如可以触发路由跳转等。

具体代码实现如下所示：


```
import { useRouter } from 'vue-router';

const router = useRouter();

const onClick = () => {
  router.push('/list');
};
```

上面这四个场景，覆盖了大部分的单页面应用业务需求场景。

不过，我们的单页面应用实现的还不是很完善。你有没有发现，**上面所有的单页面应用里的“子页面”，都是跟“主页面”的代码耦合在一起**，也就是说，前端项目打包时，所有子页面代码都会一起打包进去。如果我们后续扩展了更多子页面，也是直接打包进主页面代码，这会导致单页面应用构建结果体积非常大。

那怎么做才能优雅地扩展单页面应用，不导致主页面构建结果体积过大呢？

如何优雅扩展的单页面应用？

答案就是**按需加载。**我们可以把每个子页面资源独立打包，当主页面切换到子页面时，按需加载对应子页面的 JavaScript 和其他资源。

用到的技术也十分简单，利用 ES Module 的动态 import 模块的能力，通过 import 来动态加载模块。具体实现代码如下所示：

```
const router = createRouter({
  linkActiveClass: 'active',
  // hash 路由配置
  // history: createWebHashHistory('/'),

  // history 路由配置
  history: createWebHistory('/'),
  routes: [
    {
```

```
      path: '/',
      name: 'home',
      component: () => import('./views/home.vue')
    },
    {
      path: '/list',
      name: 'list',
      component: () => import('./views/list.vue')
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('./views/about.vue')
    },
    {
      path: '/detail/:id',
      component: () => import('./views/detail.vue'),
      children: [
        { path: '', component: () => import('./views/detail-item.vue') }
      ]
    }
  ]
});
```


这里，import 动态加载子模块后，Vite 在生产模式构建时，会分割对应代码文件，最终生产环境也能实现单页面应用动态加载模块。如果你使用 Webpack 编译项目，也可以配置类似的构建结果，根据 import 来动态分割代码文件。

扩展单页面应用，除了动态加载子页面，也还有其它优化点，你可以围绕着“**数据隔离**”和“**样式隔离**”两个点来处理。

数据隔离，就是防止全局变量泛滥或者污染。在实际项目中，我们可以规范每个子页面开发过程，禁止操作 window 的全局变量。如果真的需要借助全局变量进行操作，你可以在每个子页面操作全局变量时，加个唯一的前缀，同时，离开页面时清理原有全局变量的内容，防止内存泄露。

样式隔离，就是防止样式污染。如果子页面是通过 Vue.js 模板语法开发的，你可以加上 scoped 属性保证每个子页面的样式都是唯一不冲突，或者，对每个子页面约定一个唯一的 className 前缀。

总结

通过今天的学习，相信你已经知道了如何开发 Vue.js 的单页面应用。

单页面应用，目前有两种主流的技术实现方式，一种是通过 hash 路由来实现，另一种是通过 history 路由来实现：

- hash 路由的浏览器兼容比较好，可以兼容到 IE8 浏览器，但是 URL 的格式就限制必须用 # 号的 hash 值来标识。
- history 路由对浏览器有一定要求，同时如果浏览器强制刷新，就需要服务端做服务端层面的路由支持，但是 history 路由的 URL 格式和正常的服务路由一致。

另外我们还总结了单页面路由的核心原理，你要重点看一下：

- 利用 hash 或者 history 进行无刷新修改浏览器 URL；
- 监听 URL 变化执行页面内容更新或切换；
- 如果用 history 路由，就要考虑服务器路由的同步和浏览器的兼容。

最后，我们通过 Vue.js 官方提供的 vue-router 模块，展示了如何实现单页面应用中常见的四种功能需求场景。通过这节课的学习，希望你充分明白单页面路由的技术原理，理解 API 背后是如何运行的，而不只是停留在 vue-router 的 API 使用。

思考题

如何处理单页面和多页面项目共存？

欢迎留言参与讨论，如果有疑问，也欢迎留言。我们下节课见。

完整的代码在这里

上一篇 15丨定制运营拖拽组件：如何实现运营搭建页面的拖拽功能？

下一篇 17丨Koa.js：如何结合Koa.js开发Node.js Web服务？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。