> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

相信对有一定基础的前端开发工程师来说，路由并不陌生，它最初源于服务端，在服务端中路由描述的是 URL 与处理函数之间的映射关系。

而在 Web 前端单页应用 SPA 中，路由描述的是 URL 与视图之间的映射关系，这种映射是单向的，即 URL 变化会引起视图的更新。

相比于后端路由，前端路由的好处是无须刷新页面，减轻了服务器的压力，提升了用户体验。目前主流支持单页应用的前端框架，基本都有配套的或第三方的路由系统。相应的，Vue.js 也提供了官方前端路由实现 Vue Router，那么这节课我们就来学习它的实现原理。

> Vue.js 3.0 配套的 Vue Router 源码在这里，建议你学习前先把源码 clone 下来。如果你还不会使用路由，建议你先看它的官网文档，会使用后再来学习本节课。

路由的基本用法

我们先通过一个简单地示例来看路由的基本用法，希望你也可以使用 Vue cli 脚手架创建一个 Vue.js 3.0 的项目，并安装 4.x 版本的 Vue Router 把项目跑起来。

注意，为了让 Vue.js 可以在线编译模板，你需要在根目录下配置 vue.config.js，并且设置 runtimeCompiler 为 true：

```
module.exports = {

  runtimeCompiler: true

}
```

然后我们修改页面的 HTML 模板，加上如下代码：

```
<div>

<h1>Hello App!</h1>

<p>

<router-link to="/">Go to Home</router-link>

<router-link to="/about">Go to About</router-link>

</p>

<router-view></router-view>
```

```
</div>
```

其中，RouterLink 和 RouterView 是 Vue Router 内置的组件。

RouterLink 表示路由的导航组件，我们可以配置 to 属性来指定它跳转的链接，它最终会在页面上渲染生成 a 标签。

RouterView 表示路由的视图组件，它会渲染路径对应的 Vue 组件，也支持嵌套。

RouterLink 和 RouterView 的具体实现，我们会放到后面去分析。

有了模板之后，我们接下来看如何初始化路由：

```
import { createApp } from 'vue'

import { createRouter, createWebHashHistory } from 'vue-router'

const Home = { template: '<div>Home</div>' }

const About = { template: '<div>About</div>' }

const routes = [

  { path: '/', component: Home },

  { path: '/about', component: About },

]

const router = createRouter({

  history: createWebHistory(),

  routes

})

const app = createApp({

})

app.use(router)
```

```
app.mount('#app')
```

可以看到，路由的初始化过程很简单，首先需要定义一个路由配置，这个配置主要用于描述路径和组件的映射关系，即什么路径下 RouterView 应该渲染什么路由组件。

接着创建路由对象实例，传入路由配置对象，并且也可以指定路由模式，Vue Router 目前支持三种模式，hash 模式，HTML5 模式和 memory 模式，我们常用的是前两种模式。

最后在挂载页面前，我们需要安装路由，这样我们就可以在各个组件中访问路由对象以及使用路由的内置组件 RouterLink 和 RouterView 了。

知道了 Vue Router 的基本用法后，接下来我们就可以探究它的实现原理了。由于 Vue Router 源码加起来有几千行，限于篇幅，我会把重点放在整体的实现流程上，不会讲实现的细节。

路由的实现原理

我们先从用户使用的角度来分析，先从路由对象的创建过程开始。

路由对象的创建

Vue Router 提供了一个 createRouter API，你可以通过它来创建一个路由对象，我们来看它的实现：

```
function createRouter(options) {

const router = {

    currentRoute,

    addRoute,

    removeRoute,

    hasRoute,

    getRoutes,

    resolve,

    options,

    push,

    replace,

    go,
```

```
    back: () => go(-1),

    forward: () => go(1),

    beforeEach: beforeGuards.add,

    beforeResolve: beforeResolveGuards.add,

    afterEach: afterGuards.add,

    onError: errorHandlers.add,

    isReady,

    install(app) {

    }

  }

return router

}
```

我们省略了大部分代码，只保留了路由对象相关的代码，可以看到路由对象 router 就是一个对象，它维护了当前路径 currentRoute，且拥有很多辅助方法。

目前你只需要了解这么多，创建完路由对象后，我们现在来安装它。

路由的安装

Vue Router 作为 Vue 的插件，当我们执行 app.use(router) 的时候，实际上就是在执行 router 的 install 方法来安装路由，并把 app 作为参数传入，来看它的定义：

```
const router = {

  install(app) {

const router = this
```

```
    app.component('RouterLink', RouterLink)

    app.component('RouterView', RouterView)

    app.config.globalProperties.$router = router

    Object.defineProperty(app.config.globalProperties, '$route', {

      get: () => unref(currentRoute),

    })

if (isBrowser &&

      !started &&

      currentRoute.value === START_LOCATION_NORMALIZED) {

      started = true

push(routerHistory.location).catch(err => {

        warn('Unexpected error when starting the router:', err)

      })

    }

const reactiveRoute = {}

for (let key in START_LOCATION_NORMALIZED) {

      reactiveRoute[key] = computed(() => currentRoute.value[key])

    }

    app.provide(routerKey, router)

    app.provide(routeLocationKey, reactive(reactiveRoute))
```

```
    let unmountApp = app.unmount

    installedApps.add(app)

    app.unmount = function () {

      installedApps.delete(app)

if (installedApps.size < 1) {

        removeHistoryListener()

        currentRoute.value = START_LOCATION_NORMALIZED

        started = false

        ready = false

      }

      unmountApp.call(this, arguments)

    }

  }

}
```

路由的安装的过程我们需要记住以下两件事情。

1. 全局注册 RouterView 和 RouterLink 组件——这是你安装了路由后，可以在任何组件中去使用这俩个组件的原因，如果你使用 RouterView 或者 RouterLink 的时候收到提示不能解析 router-link 和 router-view，这说明你压根就没有安装路由。
2. 通过 provide 方式全局注入 router 对象和 reactiveRoute 对象，其中 router 表示用户通过 createRouter 创建的路由对象，我们可以通过它去动态操作路由，reactiveRoute 表示响应式的路径对象，它维护着路径的相关信息。

那么至此我们就已经了解了路由对象的创建，以及路由的安装，但是前端路由的实现，还需要解决几个核心问题：路径是如何管理的，路径和路由组件的渲染是如何映射的。

那么接下来，我们就来更细节地来看，依次来解决这两个问题。

路径的管理

路由的基础结构就是一个路径对应一种视图，当我们切换路径的时候对应的视图也会切换，因此一个很重要的方面就是对路径的管理。

首先，我们需要维护当前的路径 currentRoute，可以给它一个初始值 START_LOCATION_NORMALIZED，如下：

```
const START_LOCATION_NORMALIZED = {

  path: '/',

  name: undefined,

  params: {},

  query: {},

  hash: '',

  fullPath: '/',

  matched: [],

  meta: {},

  redirectedFrom: undefined

}
```

可以看到，路径对象包含了非常丰富的路径信息，具体含义我就不在这多说了，你可以参考官方文档。

路由想要发生变化，就是通过改变路径完成的，路由对象提供了很多改变路径的方法，比如 router.push、router.replace，它们的底层最终都是通过 pushWithRedirect 完成路径的切换，我们来看一下它的实现：

```
function pushWithRedirect(to, redirectedFrom) {

const targetLocation = (pendingLocation = resolve(to))

const from = currentRoute.value

const data = to.state
```

```
const force = to.force

const replace = to.replace === true

const toLocation = targetLocation

  toLocation.redirectedFrom = redirectedFrom

  let failure

if (!force && isSameRouteLocation(stringifyQuery$1, from, targetLocation)) {

    failure = createRouterError(16 , { to: toLocation, from })

    handleScroll(from, from, true, false)

  }

return (failure ? Promise.resolve(failure) : navigate(toLocation, from))

    .catch((error) => {

if (isNavigationFailure(error, 4  |

8  |

2 )) {

return error

      }

return triggerError(error)

    })

    .then((failure) => {
```

```
if (failure) {

      }

else {

        failure = finalizeNavigation(toLocation, from, true, replace, data)

      }

      triggerAfterEach(toLocation, from, failure)

return failure

    })

}
```

我省略了一部分代码的实现，这里主要来看 pushWithRedirect 的核心思路，首先参数 to 可能有多种情况，可以是一个表示路径的字符串，也可以是一个路径对象，所以要先经过一层 resolve 返回一个新的路径对象，它比前面提到的路径对象多了一个 matched 属性，它的作用我们后续会介绍。

得到新的目标路径后，接下来执行 navigate 方法，它实际上是执行路由切换过程中的一系列导航守卫函数，我们后续会介绍。navigate 成功后，会执行 finalizeNavigation 完成导航，在这里完成真正的路径切换，我们来看它的实现：

```
function finalizeNavigation(toLocation, from, isPush, replace, data) {

const error = checkCanceledNavigation(toLocation, from)

if (error)

return error

const isFirstNavigation = from === START_LOCATION_NORMALIZED

const state = !isBrowser ? {} : history.state

if (isPush) {
```

```
if (replace || isFirstNavigation)

      routerHistory.replace(toLocation.fullPath, assign({

        scroll: isFirstNavigation && state && state.scroll,

      }, data))

else

      routerHistory.push(toLocation.fullPath, data)

  }

  currentRoute.value = toLocation

handleScroll(toLocation, from, isPush, isFirstNavigation)

markAsReady()

}
```

这里的 finalizeNavigation 函数，我们重点关注两个逻辑，一个是更新当前的路径 currentRoute 的值，一个是执行 routerHistory.push 或者是 routerHistory.replace 方法更新浏览器的 URL 的记录。

每当我们切换路由的时候，会发现浏览器的 URL 发生了变化，但是页面却没有刷新，它是怎么做的呢？

在我们创建 router 对象的时候，会创建一个 history 对象，前面提到 Vue Router 支持三种模式，这里我们重点分析 HTML5 的 history 的模式：

```
function createWebHistory(base) {

  base = normalizeBase(base)

const historyNavigation = useHistoryStateNavigation(base)

const historyListeners = useHistoryListeners(base, historyNavigation.state,
historyNavigation.location, historyNavigation.replace)

function go(delta, triggerListeners = true) {
```

```
if (!triggerListeners)

      historyListeners.pauseListeners()

    history.go(delta)

  }

const routerHistory = assign({

    location: '',

    base,

    go,

    createHref: createHref.bind(null, base),

  }, historyNavigation, historyListeners)

  Object.defineProperty(routerHistory, 'location', {

    get: () => historyNavigation.location.value,

  })

  Object.defineProperty(routerHistory, 'state', {

    get: () => historyNavigation.state.value,

  })

return routerHistory

}
```

对于 routerHistory 对象而言，它有两个重要的作用，一个是路径的切换，一个是监听路径的变化。

其中，路径切换主要通过 historyNavigation 来完成的，它是 useHistoryStateNavigation 函数的返回值，我们来看它的实现：

```
function useHistoryStateNavigation(base) {

const { history, location } = window

  let currentLocation = {

    value: createCurrentLocation(base, location),

  }

  let historyState = { value: history.state }

if (!historyState.value) {

    changeLocation(currentLocation.value, {

      back: null,

      current: currentLocation.value,

      forward: null,

      position: history.length - 1,

      replaced: true,

      scroll: null,

    }, true)

  }

function changeLocation(to, state, replace) {

const url = createBaseLocation() +
```

```
      (base.indexOf('#') > -1 && location.search

        ? location.pathname + location.search + '#'

        : base) +

      to

try {

      history[replace ? 'replaceState' : 'pushState'](state, '', url)

      historyState.value = state

    }

catch (err) {

      warn('Error with push/replace State', err)

      location[replace ? 'replace' : 'assign'](url)

    }

  }

function replace(to, data) {

const state = assign({}, history.state, buildState(historyState.value.back,

      to, historyState.value.forward, true), data, { position:
historyState.value.position })

    changeLocation(to, state, true)

    currentLocation.value = to

  }
```

```
function push(to, data) {

const currentState = assign({},

      historyState.value, history.state, {

        forward: to,

        scroll: computeScrollPosition(),

      })

if ( !history.state) {

      warn(`history.state seems to have been manually replaced without
preserving the necessary values. Make sure to preserve existing history state if
you are manually calling history.replaceState:\n\n` +

        `history.replaceState(history.state, '', url)\n\n` +

        `You can find more information at https:

    }

    changeLocation(currentState.current, currentState, true)

const state = assign({}, buildState(currentLocation.value, to, null), {
position: currentState.position + 1 }, data)

    changeLocation(to, state, false)

    currentLocation.value = to

  }

return {

    location: currentLocation,

    state: historyState,
```

```
    push,

    replace

  }

}
```

该函数返回的 push 和 replace 函数，会添加给 routerHistory 对象上，因此当我们调用 routerHistory.push 或者是 routerHistory.replace 方法的时候实际上就是在执行这两个函数。

push 和 replace 方法内部都是执行了 changeLocation 方法，该函数内部执行了浏览器底层的 history.pushState 或者 history.replaceState 方法，会向当前浏览器会话的历史堆栈中添加一个状态，这样就在不刷新页面的情况下修改了页面的 URL。

我们使用这种方法修改了路径，这个时候假设我们点击浏览器的回退按钮回到上一个 URL，这需要恢复到上一个路径以及更新路由视图，因此我们还需要监听这种 history 变化的行为，做一些相应的处理。

History 变化的监听主要是通过 historyListeners 来完成的，它是 useHistoryListeners 函数的返回值，我们来看它的实现：

```
function useHistoryListeners(base, historyState, currentLocation, replace) {

  let listeners = []

  let teardowns = []

  let pauseState = null

const popStateHandler = ({ state, }) => {

const to = createCurrentLocation(base, location)

const from = currentLocation.value

const fromState = historyState.value

    let delta = 0

if (state) {
```

```
      currentLocation.value = to

      historyState.value = state

if (pauseState && pauseState === from) {

        pauseState = null

return

      }

      delta = fromState ? state.position - fromState.position : 0

    }

else {

      replace(to)

    }

    listeners.forEach(listener => {

      listener(currentLocation.value, from, {

        delta,

        type: NavigationType.pop,

        direction: delta

          ? delta > 0

            ? NavigationDirection.forward

            : NavigationDirection.back

          : NavigationDirection.unknown,
```

```
      })

    })

  }

function pauseListeners() {

    pauseState = currentLocation.value

  }

function listen(callback) {

    listeners.push(callback)

const teardown = () => {

const index = listeners.indexOf(callback)

if (index > -1)

        listeners.splice(index, 1)

    }

    teardowns.push(teardown)

return teardown

  }

function beforeUnloadListener() {

const { history } = window

if (!history.state)
```

```
      return

    history.replaceState(assign({}, history.state, { scroll:
computeScrollPosition() }), '')

  }

  function destroy() {

for (const teardown of teardowns)

      teardown()

    teardowns = []

    window.removeEventListener('popstate', popStateHandler)

    window.removeEventListener('beforeunload', beforeUnloadListener)

  }

  window.addEventListener('popstate', popStateHandler)

  window.addEventListener('beforeunload', beforeUnloadListener)

return {

    pauseListeners,

    listen,

    destroy

  }

}
```

该函数返回了 listen 方法，允许你添加一些侦听器，侦听 hstory 的变化，同时这个方法也被挂载到了 routerHistory 对象上，这样外部就可以访问到了。

该函数内部还监听了浏览器底层 Window 的 popstate 事件，当我们点击浏览器的回退按钮或者是执行了 history.back 方法的时候，会触发事件的回调函数 popStateHandler，进而遍历侦听器 listeners，执行每一个侦听器函数。

那么，Vue Router 是如何添加这些侦听器的呢？原来在安装路由的时候，会执行一次初始化导航，执行了 push 方法进而执行了 finalizeNavigation 方法。

在 finalizeNavigation 的最后，会执行 markAsReady 方法，我们来看它的实现：

```
function markAsReady(err) {

if (ready)

return

  ready = true

setupListeners()

  readyHandlers

    .list()

    .forEach(([resolve, reject]) => (err ? reject(err) : resolve()))

  readyHandlers.reset()

}
```

markAsReady 内部会执行 setupListeners 函数初始化侦听器，且保证只初始化一次。我们再接着来看 setupListeners 的实现：

```
function setupListeners() {

  removeHistoryListener = routerHistory.listen((to, _from, info) => {

const toLocation = resolve(to)

    pendingLocation = toLocation
```

```
const from = currentRoute.value

if (isBrowser) {

      saveScrollPosition(getScrollKey(from.fullPath, info.delta),
computeScrollPosition())

    }

    navigate(toLocation, from)

      .catch((error) => {

if (isNavigationFailure(error, 4  | 8 )) {

return error

        }

if (isNavigationFailure(error, 2 )) {

if (info.delta)

            routerHistory.go(-info.delta, false)

          pushWithRedirect(error.to, toLocation

          ).catch(noop)

return Promise.reject()

        }

if (info.delta)

          routerHistory.go(-info.delta, false)

return triggerError(error)
```

```
      })

      .then((failure) => {

        failure =

          failure ||

          finalizeNavigation(

            toLocation, from, false)

if (failure && info.delta)

          routerHistory.go(-info.delta, false)

        triggerAfterEach(toLocation, from, failure)

      })

      .catch(noop)

  })

}
```

侦听器函数也是执行 navigate 方法，执行路由切换过程中的一系列导航守卫函数，在 navigate 成功后执行 finalizeNavigation 完成导航，完成真正的路径切换。这样就保证了在用户点击浏览器回退按钮后，可以恢复到上一个路径以及更新路由视图。

至此，我们就完成了路径管理，在内存中通过 currentRoute 维护记录当前的路径，通过浏览器底层 API 实现了路径的切换和 history 变化的监听。