



下载APP



## 37 | 前端路由原理：vue-router源码剖析

2022-01-14 大圣

《玩转Vue 3全家桶》

课程介绍 &gt;



讲述：大圣

时长 07:06 大小 6.52M



你好，我是大圣。

上一讲我们学习了下一代 Vuex 框架 Pinia 的原理，今天我来带你分析 Vue 生态中另外一个重要的框架 vue-router 的源码。

课程中我们也实现过一个迷你的 router，我们通过监听路由的变化，把路由数据包裹成响应式对象后，一旦路由发生变化，我们就去定义好的路由数据中查询当前路由对应的组件，在 router-view 中渲染即可。今天我们就进入到 vue-router 源码的内部，看一下实际的 vue-router 和我们实现的迷你版本有什么区别。

### vue-router 入口分析


vue-router 提供了 createRouter 方法来创建路由配置，我们传入每个路由地址对应的组件后，使用 app.use 在 Vue 中加载 vue-router 插件，并且给 Vue 注册了两个内置组件，router-view 负责渲染当前路由匹配的组件，router-link 负责页面的跳转。



我们先来看下 **createRouter 如何实现**，完整的代码你可以在 [@ GitHub](#)上看到。这个函数比较长，还好我们有 TypeScript，我们先看下 createRouter 的参数。

在下面的代码中，参数 RouterOptions 是规范我们配置的路由对象，主要包含 history、routes 等数据。routes 就是我们需要配置的路由对象，类型是 RouteRecordRaw 组成的数组，并且 RouteRecordRaw 的类型是三个类型的合并。然后返回值的类型 Router 就是包含了 addRoute、push、beforeEnter、install 方法的一个对象，**并且维护了 currentRoute 和 options 两个属性。**

并且每个类型方法还有详细的注释，这也极大降低了阅读源码的门槛，可以帮助我们在看到函数的类型时就知道函数大概的功能。我们知道 Vue 中 app.use 实际上执行的就是 router 对象内部的 install 方法，我们先进入到 install 方法看下是如何安装的。

 复制代码

```
1 // createRouter传递参数的类型
2 export interface RouterOptions extends PathParserOptions {
3   history: RouterHistory
4   routes: RouteRecordRaw[]
5   scrollBehavior?: RouterScrollBehavior
6   ...
7 }
8 // 每个路由配置的类型
9 export type RouteRecordRaw =
10   | RouteRecordSingleView
11   | RouteRecordMultipleViews
12   | RouteRecordRedirect
13
14 //... other config
15 // Router接口的全部方法和属性
16 export interface Router {
17   readonly currentRoute: Ref<RouteLocationNormalizedLoaded>
18   readonly options: RouterOptions
19
20   addRoute(parentName: RouteRecordName, route: RouteRecordRaw): () => void
21   addRoute(route: RouteRecordRaw): () => void
22   Route(name: RouteRecordName): void
23   hasRoute(name: RouteRecordName): boolean
24
25   getRoutes(): RouteRecord[]
26   resolve(
27     to: RouteLocationRaw,
28     currentLocation?: RouteLocationNormalizedLoaded
29   ): RouteLocation & { href: string }
30   push(to: RouteLocationRaw): Promise<NavigationFailure | void | undefined>
31   replace(to: RouteLocationRaw): Promise<NavigationFailure | void | undefined>
32   back(): ReturnType<Router['go']>
33   forward(): ReturnType<Router['go']>
34   go(delta: number): void
35   beforeEach(guard: NavigationGuardWithThis<undefined>): () => void
36   beforeResolve(guard: NavigationGuardWithThis<undefined>): () => void
```

```
37   afterEach(guard: NavigationHookAfter): () => void
38   onError(handler: _ErrorHandler): () => void
39   isReady(): Promise<void>
40   install(app: App): void
41 }
42
43
44
45
46
47 export function createRouter(options: RouterOptions): Router {
48
49
50
51 }
```

## 路由安装

从下面的代码中我们可以看到，在 `createRouter` 的最后，创建了包含 `addRoute`、`push` 等方法的对象，并且 `install` 方法内部注册了 `RouterLink` 和 `RouterView` 两个组件。所以我们可以任何组件内部直接使用 `<router-view>` 和 `<router-link>` 组件，然后注册全局变量 `router` 和 `route`，其中

`router` 就是我们通过 `createRouter` 返回的路由对象，包含 `addRoute`、`push` 等方法，`route` 使用 `defineProperty` 的形式返回 `currentRoute` 的值，可以做到和 `currentRoute` 值同步。

然后使用 `computed` 把路由变成响应式对象，存储在 `reactiveRoute` 对象中，再通过 `app.provide` 给全局注册了 `route` 和 `reactive` 包裹后的 `reactiveRoute` 对象。我们之前介绍 `provide` 函数的时候也介绍了，`provide` 提供的数据并没有做响应式的封装，**需要响应式的时候需要自己使用 `ref` 或者 `reactive` 封装为响应式对象**，最后注册 `unmount` 方法实现 `vue-router` 的安装。

```
1 export function createRouter(options: RouterOptions): Router {
2   ....
3   let started: boolean | undefined
4   const installedApps = new Set<App>()
5   // 路由对象
6   const router: Router = {
7     currentRoute,
8
9     addRoute,
10    removeRoute,
11    hasRoute,
12    getRoutes,
13    resolve,
14    options,
```

[复制代码](#)

```
15   push,
16   replace,
17   go,
18   back: () => go(-1),
19   forward: () => go(1),
20
21   beforeEach: beforeGuards.add,
22   beforeResolve: beforeResolveGuards.add,
23   afterEach: afterGuards.add,
24
25   onError: errorHandlers.add,
26   isReady,
27   // 插件按章
28   install(app: App) {
29     const router = this
30     // 注册全局组件 router-link和router-view
31     app.component('RouterLink', RouterLink)
32     app.component('RouterView', RouterView)
33
34     app.config.globalProperties.$router = router
35     Object.defineProperty(app.config.globalProperties, '$route', {
36       enumerable: true,
37       get: () => unref(currentRoute),
38     })
39     if (
40       isBrowser &&
41       !started &&
42       currentRoute.value === START_LOCATION_NORMALIZED
43     ) {
44       // see above
45       started = true
46       push(routerHistory.location).catch(err => {
47         if (__DEV__) warn('Unexpected error when starting the router:', err)
48       })
49     }
50
51     const reactiveRoute = {} as {
52       [k in keyof RouteLocationNormalizedLoaded]: ComputedRef<
53         RouteLocationNormalizedLoaded[k]
54       >
55     }
56     for (const key in START_LOCATION_NORMALIZED) {
57       // @ts-expect-error: the key matches
58       reactiveRoute[key] = computed(() => currentRoute.value[key])
59     }
60     // 提供全局配置
61     app.provide(routerKey, router)
62     app.provide(routeLocationKey, reactive(reactiveRoute))
63     app.provide(routerViewLocationKey, currentRoute)
64
65     const unmountApp = app.unmount
66     installedApps.add(app)
67     app.unmount = function () {
68       installedApps.delete(app)
69       // ...
70       unmountApp()
71     }
```

```
72
73     if ((__DEV__ || __FEATURE_PROD_DEVTOOLS__) && isBrowser) {
74         addDevtools(app, router, matcher)
75     }
76 },
77 }
78
79 return router
80 }
81
```

路由对象创建和安装之后，我们下一步需要了解的就是 **router-link** 和 **router-view** 两个组件的实现方式。

通过下面的代码我们可以看到，RouterView 的 setup 函数返回了一个函数，这个函数就是 RouterView 组件的 render 函数。大部分我们使用的方式就是一个 `<router-view />` 组件，没有 slot 情况下返回的就是 component 变量。component 使用 h 函数返回 ViewComponent 的虚拟 DOM，而 ViewComponent 是根据 `matchedRoute.components[props.name]` 计算而来。

matchedRoute 依赖的 matchedRouteRef 的计算逻辑在如下代码的第 12 ~ 15 行，数据来源 injectedRoute 就是上面我们注入的 currentRoute 对象。

[复制代码](#)

```
1 export const RouterViewImpl = /*#__PURE__*/ defineComponent({
2   name: 'RouterView',
3   props: {
4     name: {
5       type: String as PropType<string>,
6       default: 'default',
7     },
8     route: Object as PropType<RouteLocationNormalizedLoaded>,
9   },
10  // router-view组件源码
11  setup(props, { attrs, slots }) {
12    // 全局的reactiveRoute对象注入
13    const injectedRoute = inject(routerViewLocationKey)!
14
15    const routeToDisplay = computed(() => props.route || injectedRoute.value)
16    const depth = inject(viewDepthKey, 0)
17    const matchedRouteRef = computed<RouteLocationMatched | undefined>(<
18      () => routeToDisplay.value.matched[depth]
19    )
20    // 嵌套层级
21    provide(viewDepthKey, depth + 1)
22    // 匹配的router对象
23    provide(matchedRouteKey, matchedRouteRef)
24    provide(routerViewLocationKey, routeToDisplay)
25
26
```

```

27     const viewRef = ref<ComponentPublicInstance>()
28     // 返回的render函数
29     return () => {
30         const route = routeToDisplay.value
31         const matchedRoute = matchedRouteRef.value
32         const ViewComponent = matchedRoute && matchedRoute.components[props.name]
33         const currentName = props.name
34
35         if (!ViewComponent) {
36             return normalizeSlot(slots.default, { Component: ViewComponent, route })
37         }
38
39         // props from route configuration
40         const routePropsOption = matchedRoute!.props[props.name]
41         const routeProps = routePropsOption
42             ? routePropsOption === true
43               ? route.params
44               : typeof routePropsOption === 'function'
45                 ? routePropsOption(route)
46                 : routePropsOption
47             : null
48
49         const onVnodeUnmounted: VNodeProps['onVnodeUnmounted'] = vnode => {
50             // remove the instance reference to prevent leak
51             if (vnode.component!.isUnmounted) {
52                 matchedRoute!.instances[currentName] = null
53             }
54         }
55         // 创建需要渲染组件的虚拟dom
56         const component = h(
57             ViewComponent,
58             assign({}, routeProps, attrs, {
59                 onVnodeUnmounted,
60                 ref: viewRef,
61             })
62         )
63
64         return (
65             // pass the vnode to the slot as a prop.
66             // h and <component :is="..."> both accept vnodes
67             normalizeSlot(slots.default, { Component: component, route }) ||
68             component
69         )
70     }
71 },

```

## 路由更新

到这我们可以看出，RouterView 渲染的组件是由当前匹配的路由变量 matchedRoute 决定的。接下来我们回到 createRouter 函数中，可以看到 matcher 对象是由 createRouterMatcher 创建，createRouterMatcher 函数传入 routes 配置的路由数组，并

且返回创建的 RouterMatcher 对象，内部遍历 routes 数组，通过 addRoute 挨个处理路由配置。

[复制代码](#)

```
1 export function createRouter(options: RouterOptions): Router {
2   const matcher = createRouterMatcher(options.routes, options)
3   ///....
4 }
5 export function createRouterMatcher(
6   routes: RouteRecordRaw[],
7   globalOptions: PathParserOptions
8 ): RouterMatcher {
9   // matchers数组
10  const matchers: RouteRecordMatcher[] = []
11  // matcher对象
12  const matcherMap = new Map<RouteRecordName, RouteRecordMatcher>()
13  globalOptions = mergeOptions(
14    { strict: false, end: true, sensitive: false } as PathParserOptions,
15    globalOptions
16  )
17  function addRoute(){}
18  function removeRoute(){}
19  function getRoutes(){
20    return matchers
21  }
22  function insertMatcher(){}
23  function resolve(){}
24  // add initial routes
25  routes.forEach(route => addRoute(route))
26
27  return { addRoute, resolve, removeRoute, getRoutes, getRecordMatcher }
28 }
```

在下面的代码中我们可以看到，addRoute 函数内部通过 createRouteRecordMatcher 创建扩展之后的 matcher 对象，包括了 record、parent、children 等树形，可以很好地描述路由之间的嵌套父子关系。这样整个路由对象就已经创建完毕，那我们如何在路由切换的时候寻找到正确的路由对象呢？

[复制代码](#)

```
1 function addRoute(
2   record: RouteRecordRaw,
3   parent?: RouteRecordMatcher,
4   originalRecord?: RouteRecordMatcher
5 ){
6   if ('alias' in record) {
7     // 标准化alias
8   }
9   for (const normalizedRecord of normalizedRecords) {
10     // ...
11     matcher = createRouteRecordMatcher(normalizedRecord, parent, options)
```



```

12     insertMatcher(matcher)
13
14   }
15   return originalMatcher
16   ? () => {
17     // since other matchers are aliases, they should be removed by the origin
18     removeRoute(originalMatcher!)
19   }
20   : noop
21
22 }
23
24 export function createRouteRecordMatcher(
25   record: Readonly<RouteRecord>,
26   parent: RouteRecordMatcher | undefined,
27   options?: PathParserOptions
28 ): RouteRecordMatcher {
29   const parser = tokensToParser(tokenizePath(record.path), options)
30   const matcher: RouteRecordMatcher = assign(parser, {
31     record,
32     parent,
33     // these needs to be populated by the parent
34     children: [],
35     alias: [],
36   })
37
38   if (parent) {
39     if (!matcher.record.aliasOf === !parent.record.aliasOf)
40       parent.children.push(matcher)
41   }
42
43   return matcher
44 }
45

```

在 vue-router 中，路由更新可以通过 router-link 渲染的链接实现，也可以使用 router 对象的 push 等方法实现。下面的代码中，router-link 组件内部也是渲染一个 a 标签，并且注册了 a 标签的 onClick 函数，内部也是通过 router.replace 或者 router.push 来实现。

[复制代码](#)

```

1
2 export const RouterLinkImpl = /*#__PURE__*/ defineComponent({
3   name: 'RouterLink',
4   props: {
5     to: {
6       type: [String, Object] as PropType<RouteLocationRaw>,
7       required: true,
8     },
9     ...
10  },
11  // router-link源码
12  setup(props, { slots }) {
13    const link = reactive(useLink(props))

```



```

14     const { options } = inject(routerKey)!
15
16     const elClass = computed(() => ({
17         ...
18     })))
19
20     return () => {
21         const children = slots.default && slots.default(link)
22         return props.custom
23             ? children
24             : h(
25                 'a',
26                 {
27                     href: link.href,
28                     onClick: link.navigate,
29                     class: elClass.value,
30                 },
31                 children
32             )
33     },
34 },
35 })
36 // 跳转
37 function navigate(
38     e: MouseEvent = {} as MouseEvent
39 ): Promise<void | NavigationFailure> {
40     if (guardEvent(e)) {
41         return router[unref(props.replace) ? 'replace' : 'push'](
42             unref(props.to)
43             // avoid uncaught errors are they are logged anyway
44         ).catch(noop)
45     }
46     return Promise.resolve()
47 }
48

```

现在我们回到 createRouter 函数中，可以看到 push 函数直接调用了 pushWithRedirect 函数来实现，内部通过 resolve(to) 生成 targetLocation 变量。这个变量会赋值给 toLocation，然后执行 navigate(toLocation) 函数。而**这个函数内部会执行一系列的导航守卫函数**，最后会执行 finalizeNavigation 函数完成导航。

[复制代码](#)

```

1 function push(to: RouteLocationRaw | RouteLocation) {
2     return pushWithRedirect(to)
3 }
4
5 function replace(to: RouteLocationRaw | RouteLocationNormalized) {
6     return push(assign(locationAsObject(to), { replace: true }))
7 }
8 // 路由跳转函数
9 function pushWithRedirect(
10     to: RouteLocationRaw | RouteLocation,

```

```
11   redirectedFrom?: RouteLocation
12 ): Promise<NavigationFailure | void | undefined> {
13   const targetLocation: RouteLocation = (pendingLocation = resolve(to))
14   const from = currentRoute.value
15   const data: HistoryState | undefined = (to as RouteLocationOptions).state
16   const force: boolean | undefined = (to as RouteLocationOptions).force
17   // to could be a string where `replace` is a function
18   const replace = (to as RouteLocationOptions).replace === true
19
20
21
22   const toLocation = targetLocation as RouteLocationNormalized
23
24
25   return (failure ? Promise.resolve(failure) : navigate(toLocation, from))
26     .catch((error: NavigationFailure | NavigationRedirectError) =>
27       isNavigationFailure(error)
28         ? error
29         : // reject any unknown error
30           triggerError(error, toLocation, from)
31     )
32     .then((failure: NavigationFailure | NavigationRedirectError | void) => {
33
34       failure = finalizeNavigation(
35         toLocation as RouteLocationNormalizedLoaded,
36         from,
37         true,
38         replace,
39         data
40       )
41
42       triggerAfterEach(
43         toLocation as RouteLocationNormalizedLoaded,
44         from,
45         failure
46       )
47       return failure
48     })
49 }
```

在下面的代码中我们可以看到，`finalizeNavigation` 函数内部通过 `routerHistory.push` 或者 `replace` 实现路由跳转，并且更新 `currentRoute.value`。

`currentRoute` 就是我们在 `install` 方法中注册的全局变量 `route`，每次页面跳转 `currentRoute` 都会更新为 `toLocation`，在任意组件中都可以通 `route` 变量来获取当前路由的数据，最后在 `handleScroll` 设置滚动行为。

`routerHistory` 在 `createRouter` 中通过 `option.history` 获取，就是我们创建 `vue-router` 应用时通过 `createWebHistory` 或者 `createWebHashHistory` 创建的对象。

createWebHistory 返回的是 HTML5 的 history 模式路由对象，createWebHashHistory 是 Hash 模式的路由对象。

[复制代码](#)

```
1  function finalizeNavigation(  
2    toLocation: RouteLocationNormalizedLoaded,  
3    from: RouteLocationNormalizedLoaded,  
4    isPush: boolean,  
5    replace?: boolean,  
6    data?: HistoryState  
7  ): NavigationFailure | void {  
8  
9  
10  
11    const isFirstNavigation = from === START_LOCATION_NORMALIZED  
12    const state = !isBrowser ? {} : history.state  
13  
14    if (isPush) {  
15  
16      if (replace || isFirstNavigation)  
17        routerHistory.replace(  
18          toLocation.fullPath  
19        )  
20      else routerHistory.push(toLocation.fullPath, data)  
21    }  
22  
23    // accept current navigation  
24    currentRoute.value = toLocation  
25    handleScroll(toLocation, from, isPush, isFirstNavigation)  
26  
27    markAsReady()  
28  }  
29  
30  function markAsReady(err?: any): void {  
31    if (ready) return  
32    ready = true  
33    setupListeners()  
34    readyHandlers  
35      .list()  
36      .forEach(([resolve, reject]) => (err ? reject(err) : resolve()))  
37    readyHandlers.reset()  
38  }
```

下面的代码中我们可以看到，createWebHashHistory 和 createWebHistory 的实现，内部都是通过 useHistoryListeners 实现路由的监听，通过 useHistoryStateNavigation 实现路由的切换。useHistoryStateNavigation 会返回 push 或者 replace 方法来更新路由，这两个函数你可以在 [GitHub](#) 上自行学习。

[复制代码](#)

```
1  export function createWebHashHistory(base?: string): RouterHistory {
```

```
2   base = location.host ? base || location.pathname + location.search : ''
3   // allow the user to provide a `#` in the middle: `/base/#/app`
4   if (!base.includes('#')) base += '#'
5   return createWebHistory(base)
6 }
7
8
9
10 export function createWebHistory(base?: string): RouterHistory {
11   base = normalizeBase(base)
12
13   const historyNavigation = useHistoryStateNavigation(base)
14   const historyListeners = useHistoryListeners(
15     base,
16     historyNavigation.state,
17     historyNavigation.location,
18     historyNavigation.replace
19   )
20   function go(delta: number, triggerListeners = true) {
21     if (!triggerListeners) historyListeners.pauseListeners()
22     history.go(delta)
23   }
24
25   const routerHistory: RouterHistory = assign(
26     {
27       // it's overridden right after
28       location: '',
29       base,
30       go,
31       createHref: createHref.bind(null, base),
32     },
33
34     historyNavigation,
35     historyListeners
36   )
37
38   Object.defineProperty(routerHistory, 'location', {
39     enumerable: true,
40     get: () => historyNavigation.location.value,
41   })
42
43   Object.defineProperty(routerHistory, 'state', {
44     enumerable: true,
45     get: () => historyNavigation.state.value,
46   })
47
48   return routerHistory
49 }
50
```

## 总结

以上就是今天的主要内容，我们来总结一下。

这节课我们进入到 vue-router 的源码中分析了 vue-router 内部的执行逻辑，其实我们之前课上已经实现了迷你的 vue-router，在掌握了前端路由实现的原理后，再来看实际的 vue-router 源码难度会下降不少。

首先我们分析了 createRouter 函数入口函数，createRouter 函数返回了 router 对象，router 对象提供了 addRoute、push 等方法，并且在 install 方法中实现了路由，注册了组件 router-link 和 router-view。

然后通过 createRouterMatcher 创建路由匹配对象，并且在路由变化的时候维护 currentRoute，让你可以在每个组件内部 router 和 route 获取路由匹配的数据，并且动态渲染当前路由匹配的组件到 router-view 组件内部，实现了前端的路由系统。

这一讲我们也能感受到，一个玩具的 router 和实际的 vue-router 的距离，也能体会到 TypeScript 在我们阅读代码时的好处。我们阅读源码的目的之一，就是要学习和模仿优秀框架内部的设计思路，然后去优化自己项目中的代码，学会模仿也是一个优秀程序员的优秀品质。

## 思考

最后留给你一个思考题，navigate 函数负责执行路由守卫的功能，你知道它的内部是如何实现的吗？

欢迎在评论区分享你的答案，我们下一讲再见！

分享给需要的人，Ta 订阅超级会员，你将得 50 元

Ta 单独订阅本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 36 | 数据流原理：Vuex & Pinia 源码剖析

## 精选留言 (1)



InfoQ\_e521a4ce8a54

2022-01-14

写留言

navigate 函数主要是执行一个异步队列；核心代码

```
function runGuardQueue(guards: Lazy<any>[]): Promise<void> {  
  return guards.reduce(  
    (promise, guard) => promise.then(() => guard()),  
    Promise.resolve()...
```

展开

