上节课我们学习了 Vue Router 的基本用法，并且开始探究它的实现原理，今天我们继续未完的原理，一起来看路径是如何和路由组件映射的。

路径和路由组件的渲染的映射

通过前面的示例我们了解到，路由组件就是通过 RouterView 组件渲染的，那么 RouterView 是怎么渲染的呢，我们来看它的实现：

```
const RouterView = defineComponent({

  name: 'RouterView',

  props: {

    name: {

      type: String,

default: 'default',

    },

    route: Object,

  },

  setup(props, { attrs, slots }) {

    warnDeprecatedUsage()

const injectedRoute = inject(routeLocationKey)

const depth = inject(viewDepthKey, 0)

const matchedRouteRef = computed(() => (props.route ||
injectedRoute).matched[depth])

    provide(viewDepthKey, depth + 1)
```

```
    provide(matchedRouteKey, matchedRouteRef)

const viewRef = ref()

    watch(() => [viewRef.value, matchedRouteRef.value, props.name], ([instance,
to, name], [oldInstance, from, oldName]) => {

if (to) {

        to.instances[name] = instance

if (from && instance === oldInstance) {

          to.leaveGuards = from.leaveGuards

          to.updateGuards = from.updateGuards

        }

      }

if (instance &&

        to &&

        (!from || !isSameRouteRecord(to, from) || !oldInstance)) {

        (to.enterCallbacks[name] || []).forEach(callback => callback(instance))

      }

    })

return () => {

const route = props.route || injectedRoute

const matchedRoute = matchedRouteRef.value
```

```
const ViewComponent = matchedRoute && matchedRoute.components[props.name]

const currentName = props.name

if (!ViewComponent) {

return slots.default

          ? slots.default({ Component: ViewComponent, route })

          : null

      }

const routePropsOption = matchedRoute.props[props.name]

const routeProps = routePropsOption

        ? routePropsOption === true

          ? route.params

          : typeof routePropsOption === 'function'

            ? routePropsOption(route)

            : routePropsOption

        : null

const onVnodeUnmounted = vnode => {

if (vnode.component.isUnmounted) {

          matchedRoute.instances[currentName] = null

        }
```

```
      }

const component = h(ViewComponent, assign({}, routeProps, attrs, {

        onVnodeUnmounted,

        ref: viewRef,

      }))

return (

        slots.default

          ? slots.default({ Component: component, route })

          : component)

    }

  },

})
```

RouterView 组件也是基于 composition API 实现的，我们重点看它的渲染部分，由于 setup 函数的返回值是一个函数，那这个函数就是它的渲染函数。

我们从后往前看，通常不带插槽的情况下，会返回 component 变量，它是根据 ViewComponent 渲染出来的，而 ViewComponent 是根据 matchedRoute.components[props.name] 求得的，而 matchedRoute 是 matchedRouteRef 对应的 value。

matchedRouteRef 一个计算属性，在不考虑 prop 传入 route 的情况下，它的 getter 是由 injectedRoute.matched[depth] 求得的，而 injectedRoute，就是我们在前面在安装路由时候，注入的响应式 currentRoute 对象，而 depth 就是表示这个 RouterView 的嵌套层级。

所以我们可以看到，RouterView 的渲染的路由组件和当前路径 currentRoute 的 matched 对象相关，也和 RouterView 自身的嵌套层级相关。

那么接下来，我们就来看路径对象中的 matched 的值是怎么在路径切换的情况下更新的。

我们还是通过示例的方式来说明，我们对前面的示例稍做修改，加上嵌套路由的场景：

```
import { createApp } from 'vue'


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

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

const About = {

template: `<div>About

  <router-link to="/about/user">Go User</router-link>

  <router-view></router-view>

  </div>`

}

const User = {

template: '<div>User</div>,'

}

const routes = [

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

  {

path: '/about',

component: About,

children: [

      {

path: 'user',

component: User
```

```
      }

    ]

  }

]

const router = createRouter({

history: createWebHashHistory(),

  routes

})

const app = createApp({})

app.use(router)

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

它和前面示例的区别在于，我们在 About 路由组件中又嵌套了一个 RouterView 组件，然后对 routes 数组中 path 为 /about 的路径配置扩展了 children 属性，对应的就是 About 组件嵌套路由的配置。

当我们执行 createRouter 函数创建路由的时候，内部会执行如下代码来创建一个 matcher 对象：

```
const matcher = createRouterMatcher(options.routes, options)
```

执行了 createRouterMatcher 函数，并传入 routes 路径配置数组，它的目的就是根据路径配置对象创建一个路由的匹配对象，再来看它的实现：

```
function createRouterMatcher(routes, globalOptions) {

const matchers = []

const matcherMap = new Map()

  globalOptions = mergeOptions({ strict: false, end: true, sensitive: false },
globalOptions)
```

```
function addRoute(record, parent, originalRecord) {

    let isRootAdd = !originalRecord

    let mainNormalizedRecord = normalizeRouteRecord(record)

    mainNormalizedRecord.aliasOf = originalRecord && originalRecord.record

const options = mergeOptions(globalOptions, record)

const normalizedRecords = [

      mainNormalizedRecord,

    ]

let matcher

    let originalMatcher

for (const normalizedRecord of normalizedRecords) {

      let { path } = normalizedRecord

if (parent && path[0] !== '/') {

        let parentPath = parent.record.path

        let connectingSlash = parentPath[parentPath.length - 1] === '/' ? '' :
'/'

        normalizedRecord.path =

          parent.record.path + (path && connectingSlash + path)

      }

      matcher = createRouteRecordMatcher(normalizedRecord, parent, options)
```

```
if ( parent && path[0] === '/')

        checkMissingParamsInAbsolutePath(matcher, parent)

if (originalRecord) {

        originalRecord.alias.push(matcher)

        {

          checkSameParams(originalRecord, matcher)

        }

      }

else {

        originalMatcher = originalMatcher || matcher

if (originalMatcher !== matcher)

          originalMatcher.alias.push(matcher)

if (isRootAdd && record.name && !isAliasRecord(matcher))

removeRoute(record.name)

      }

if ('children' in mainNormalizedRecord) {

        let children = mainNormalizedRecord.children

for (let i = 0; i < children.length; i++) {

          addRoute(children[i], matcher, originalRecord &&
originalRecord.children[i])
```

```
        }
      }
      originalRecord = originalRecord || matcher
insertMatcher(matcher)
    }
    return originalMatcher
      ? () => {
        removeRoute(originalMatcher)
      }
      : noop
  }
function insertMatcher(matcher) {
    let i = 0
while (i < matchers.length &&
    comparePathParserScore(matcher, matchers[i]) >= 0)
      i++
    matchers.splice(i, 0, matcher)
if (matcher.record.name && !isAliasRecord(matcher))
      matcherMap.set(matcher.record.name, matcher)
```

```
  }

  routes.forEach(route => addRoute(route))

return { addRoute, resolve, removeRoute, getRoutes, getRecordMatcher }

}
```

createRouterMatcher 函数内部定义了一个 matchers 数组和一些辅助函数，我们先重点关注 addRoute 函数的实现，我们只关注核心流程。

在 createRouterMatcher 函数的最后，会遍历 routes 路径数组调用 addRoute 方法添加初始路径。

在 addRoute 函数内部，首先会把 route 对象标准化成一个 record，其实就是给路径对象添加更丰富的属性。

然后再执行 createRouteRecordMatcher 函数，传入标准化的 record 对象，我们再来看它的实现：

```
function createRouteRecordMatcher(record, parent, options) {

const parser = tokensToParser(tokenizePath(record.path), options)

  {

const existingKeys = new Set()

for (const key of parser.keys) {

if (existingKeys.has(key.name))

        warn(`Found duplicated params with name "${key.name}" for path
"${record.path}". Only the last one will be available on "$route.params".`)

      existingKeys.add(key.name)

    }

  }

const matcher = assign(parser, {

    record,
```

```
    parent,
    children: [],
    alias: []
  })
if (parent) {
if (!matcher.record.aliasOf === !parent.record.aliasOf)
      parent.children.push(matcher)
  }
return matcher
}
```

其实 createRouteRecordMatcher 创建的 matcher 对象不仅仅拥有 record 属性来存储 record，还扩展了一些其他属性，需要注意，如果存在 parent matcher，那么会把当前 matcher 添加到 parent.children 中去，这样就维护了父子关系，构造了树形结构。

那么什么情况下会有 parent matcher 呢？让我们先回到 addRoute 函数，在创建了 matcher 对象后，接着判断 record 中是否有 children 属性，如果有则遍历 children，递归执行 addRoute 方法添加路径，并把创建 matcher 作为第二个参数 parent 传入，这也就是 parent matcher 存在的原因。

所有 children 处理完毕后，再执行 insertMatcher 函数，把创建的 matcher 存入到 matchers 数组中。

至此，我们就根据用户配置的 routes 路径数组，初始化好了 matchers 数组。

那么再回到我们前面的问题，分析路径对象中的 matched 的值是怎么在路径切换的情况下更新的。

之前我们提到过，切换路径会执行 pushWithRedirect 方法，内部会执行一段代码：

```
const targetLocation = (pendingLocation = resolve(to))
```

这里会执行 resolve 函数解析生成 targetLocation，这个 targetLocation 最后也会在 finalizeNavigation 的时候赋值 currentRoute 更新当前路径。我们来看 resolve 函数的实现：

```
function resolve(location, currentLocation) {
```

```
  let matcher

  let params = {}

let path

  let name

if ('name' in location && location.name) {

    matcher = matcherMap.get(location.name)

if (!matcher)

throw createRouterError(1 , {

        location,

      })

    name = matcher.record.name

    params = assign(

      paramsFromLocation(currentLocation.params,

        matcher.keys.filter(k => !k.optional).map(k => k.name)),
location.params)

    path = matcher.stringify(params)

  }

else if ('path' in location) {

    path = location.path

if ( !path.startsWith('/')) {
```

```
      warn(`The Matcher cannot resolve relative paths but received "${path}". Unless you directly called \`matcher.resolve("${path}")\`, this is probably a bug in vue-router. Please open an issue at https:

    }

    matcher = matchers.find(m => m.re.test(path))

if (matcher) {

      params = matcher.parse(path)

      name = matcher.record.name

    }

  }

else {

    matcher = currentLocation.name

      ? matcherMap.get(currentLocation.name)

      : matchers.find(m => m.re.test(currentLocation.path))

if (!matcher)

throw createRouterError(1 , {

        location,

        currentLocation,

      })

    name = matcher.record.name

    params = assign({}, currentLocation.params, location.params)
```

```
    path = matcher.stringify(params)
  }
const matched = []
  let parentMatcher = matcher
while (parentMatcher) {
    matched.unshift(parentMatcher.record)
    parentMatcher = parentMatcher.parent
  }
return {
    name,
    path,
    params,
    matched,
    meta: mergeMetaFields(matched),
  }
}
```

resolve 函数主要做的事情就是根据 location 的 name 或者 path 从我们前面创建的 matchers 数组中找到对应的 matcher，然后再顺着 matcher 的 parent 一直找到链路上所有匹配的 matcher，然后获取其中的 record 属性构造成一个 matched 数组，最终返回包含 matched 属性的新的路径对象。

这么做的目的就是让 matched 数组完整记录 record 路径，它的顺序和嵌套的 RouterView 组件顺序一致，也就是 matched 数组中的第 n 个元素就代表着 RouterView 嵌套的第 n 层。

因此 targetLocation 和 to 相比，其实就是多了一个 matched 对象，这样再回到我们的 RouterView 组件，就可以从 `injectedRoute.matched[depth] [props.name]` 中拿到对应的组件对象定义，去渲染对应的组件了。

至此，我们就搞清楚路径和路由组件的渲染是如何映射的了。

前面的分析过程中，我们提到过在路径切换过程中，会执行 navigate 方法，它包含了一系列的导航守卫钩子函数的执行，接下来我们就来分析这部分的实现原理。

导航守卫的实现

导航守卫主要是让用户在路径切换的生命周期中可以注入钩子函数，执行一些自己的逻辑，也可以取消和重定向导航，举个应用的例子：

```
router.beforeEach((to, from, next) => {

if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' }) else {

    next()

  }

})
```

这里大致含义就是进入路由前检查用户是否登录，如果没有则跳转到登录的视图组件，否则继续。

router.beforeEach 传入的参数是一个函数，我们把这类函数就称为导航守卫。

那么这些导航守卫是怎么执行的呢？这里我并不打算去详细讲 navigate 实现的完整流程，而是讲清楚它的执行原理，关于导航守卫的执行顺序建议你去对照官网文档，然后再来看实现细节。

接下来，我们来看 navigate 函数的实现：

```
function navigate(to, from) {

  let guards

const [leavingRecords, updatingRecords, enteringRecords,] =
extractChangingRecords(to, from)

  guards = extractComponentsGuards(leavingRecords.reverse(), 'beforeRouteLeave',
to, from)

for (const record of leavingRecords) {

for (const guard of record.leaveGuards) {
```

```
      guards.push(guardToPromiseFn(guard, to, from))

    }

  }

const canceledNavigationCheck = checkCanceledNavigationAndReject.bind(null, to,
from)

  guards.push(canceledNavigationCheck)

return (runGuardQueue(guards)

    .then(() => {

      guards = []

for (const guard of beforeGuards.list()) {

        guards.push(guardToPromiseFn(guard, to, from))

      }

      guards.push(canceledNavigationCheck)

return runGuardQueue(guards)

    })

    .then(() => {

      guards = extractComponentsGuards(updatingRecords, 'beforeRouteUpdate', to,
from)

for (const record of updatingRecords) {

for (const guard of record.updateGuards) {

          guards.push(guardToPromiseFn(guard, to, from))
```

```
        }

      }

      guards.push(canceledNavigationCheck)

return runGuardQueue(guards)

    })

    .then(() => {

      guards = []

for (const record of to.matched) {

if (record.beforeEnter && from.matched.indexOf(record) < 0) {

if (Array.isArray(record.beforeEnter)) {

for (const beforeEnter of record.beforeEnter)

              guards.push(guardToPromiseFn(beforeEnter, to, from))

          }

else {

            guards.push(guardToPromiseFn(record.beforeEnter, to, from))

          }

        }

      }

      guards.push(canceledNavigationCheck)
```

```
return runGuardQueue(guards)

    })

    .then(() => {

      to.matched.forEach(record => (record.enterCallbacks = {}))

      guards = extractComponentsGuards(enteringRecords, 'beforeRouteEnter', to,
from)

      guards.push(canceledNavigationCheck)

return runGuardQueue(guards)

    })

    .then(() => {

      guards = []

for (const guard of beforeResolveGuards.list()) {

        guards.push(guardToPromiseFn(guard, to, from))

      }

      guards.push(canceledNavigationCheck)

return runGuardQueue(guards)

    })

    .catch(err => isNavigationFailure(err, 8 )

      ? err

      : Promise.reject(err)))
```

```
}
```

可以看到 navigate 执行导航守卫的方式是先构造 guards 数组，数组中每个元素都是一个返回 Promise 对象的函数。

然后通过 runGuardQueue 去执行这些 guards，来看它的实现：

```
function runGuardQueue(guards) {

return guards.reduce((promise, guard) => promise.then(() => guard()),
Promise.resolve())

}
```

其实就是通过数组的 reduce 方法，链式执行 guard 函数，每个 guard 函数都会返回一个 Promise 对象。

但是从我们的例子看，我们添加的是一个普通函数，并不是一个返回 Promise 对象的函数，那是怎么做的呢?

原来在把 guard 添加到 guards 数组前，都会执行 guardToPromiseFn 函数把普通函数 Promise 化，来看它的实现：

```
import { warn as warn$1 } from "vue/dist/vue"

function guardToPromiseFn(guard, to, from, record, name) {

const enterCallbackArray = record &&

    (record.enterCallbacks[name] = record.enterCallbacks[name] || [])

return () => new Promise((resolve, reject) => {

const next = (valid) => {

if (valid === false)

        reject(createRouterError(4 , {

          from,

          to,
```

```
        }))

else if (valid instanceof Error) {

        reject(valid)

      }

else if (isRouteLocation(valid)) {

        reject(createRouterError(2 , {

          from: to,

          to: valid

        }))

      }

else {

if (enterCallbackArray &&

          record.enterCallbacks[name] === enterCallbackArray &&

          typeof valid === 'function')

          enterCallbackArray.push(valid)

        resolve()

      }

    }

const guardReturn = guard.call(record && record.instances[name], to, from, next
)
```

```
    let guardCall = Promise.resolve(guardReturn)

if (guard.length < 3)

      guardCall = guardCall.then(next)

if (guard.length > 2) {

const message = `The "next" callback was never called inside of ${guard.name ? '"' + guard.name + '"' : ''}:\n${guard.toString()}\n. If you are returning a value instead of calling "next", make sure to remove the "next" parameter from your function.`

if (typeof guardReturn === 'object' && 'then' in guardReturn) {

        guardCall = guardCall.then(resolvedValue => {

if (!next._called) {

            warn$1(message)

return Promise.reject(new Error('Invalid navigation guard'))

          }

return resolvedValue

        })

      }

else if (guardReturn !== undefined) {

if (!next._called) {

          warn$1(message)

          reject(new Error('Invalid navigation guard'))
```

```
return

        }

      }

    }

    guardCall.catch(err => reject(err))

  })

}
```

guardToPromiseFn 函数返回一个新的函数，这个函数内部会执行 guard 函数。

这里我们要注意 next 方法的设计，当我们在导航守卫中执行 next 时，实际上就是执行这里定义的 next 函数。

在执行 next 函数时，如果不传参数，那么则直接 resolve，执行下一个导航守卫；如果参数是 false，则创建一个导航取消的错误 reject 出去；如果参数是一个 Error 实例，则直接执行 reject，并把错误传递出去；如果参数是一个路径对象，则创建一个导航重定向的错误传递出去。

有些时候我们写导航守卫不使用 next 函数，而是直接返回 true 或 false，这种情况则先执行如下代码：

```
guardCall = Promise.resolve(guardReturn)
```

把导航守卫的返回值 Promise 化，然后再执行 guardCall.then(next)，把导航守卫的返回值传给 next 函数。

当然，如果你在导航守卫中定义了第三个参数 next，但是你没有在函数中调用它，这种情况也会报警告。

所以，对于导航守卫而言，经过 Promise 化后添加到 guards 数组中，然后再通过 runGuards 以及 Promise 的方式链式调用，最终依次顺序执行这些导航守卫。

总结

好的，到这里我们这一节的学习也要结束啦，通过这节课的学习，你应该要了解 Vue Router 的基本实现原理，知道路径是如何管理的，路径和路由组件的渲染是如何映射的，导航守卫是如何执行的。

当然，路由实现的细节是非常多的，我希望你学完之后，可以对照着官网的文档的 feature，自行去分析它们的实现原理。

最后，给你留一道思考题目，如果我们想给路由组件传递数据，有几种方式，分别都怎么做呢？欢迎你在留言区与我分享。