Vue Router 学习笔记

https://router.vuejs.org/

基础

起步

将组件 (components) 映射到路由 (routes)，然后告诉 Vue Router 在哪里渲染它们

HTML:

```
<p>
    <!-- 使用 router-link 组件来导航. -->
    <!-- 通过传入 `to` 属性指定链接. -->
    <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
    <router-link to="/foo">Go to Foo</router-link>
    <router-link to="/bar">Go to Bar</router-link>
  </p>
```

Js

```
// 1. 定义 (路由) 组件。
// 可以从其他文件 import 进来
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. 定义路由
// 每个路由应该映射一个组件。 其中"component" 可以是
// 通过 Vue.extend() 创建的组件构造器，
// 或者，只是一个组件配置对象。
// 我们晚点再讨论嵌套路由。
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. 创建 router 实例，然后传 `routes` 配置
// 你还可以传别的配置参数，不过先这么简单着吧。
const router = new VueRouter({
  routes // (缩写) 相当于 routes: routes
})

// 4. 创建和挂载根实例。
// 记得要通过 router 配置参数注入路由，
// 从而让整个应用都有路由功能
const app = new Vue({
  router
}).$mount('#app')
```

我们可以在任何组件内通过 **this.$router 访问路由器，也可以通过 this.$route 访问当前路由**

动态路由匹配

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件

可以在 vue-router 的路由路径中使用“动态路径参数”(dynamic segment) 来达到这个效果：

```
const router = new VueRouter({
  routes: [
    // 动态路径参数 以冒号开头
    { path: '/user/:id', component: User }
  ]
})
```

一个“路径参数”使用冒号 : 标记。当匹配到一个路由时，参数值会被设置到 this.$route.params，可以在每个组件内使用。

提醒一下，当使用路由参数时，例如从 /user/foo 导航到 /user/bar，原来的组件实例会被复用。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着组件的生命周期钩子不会再被调用。

复用组件时，想对路由参数的变化作出响应的话，你可以简单地 watch (监测变化) $route 对象：

```
const User = {
  template: '...',
  watch: {
    $route(to, from) {
      // 对路由变化作出响应...
    }
  }
}
```

或者使用 2.2 中引入的 beforeRouteUpdate 导航守卫：

```
const User = {
  template: '...',
  beforeRouteUpdate (to, from, next) {
    // react to route changes...
    // don't forget to call next()
  }
}
```

捕获所有路由或 404 Not found 路由

```
常规参数只会匹配被 / 分隔的 URL 片段中的字符。如果想匹配任意路径，我们可以使用通配符 (*):

{
  // 会匹配所有路径，设置为 404 页面，必须放在最后
  path: '*'
}
{
  // 会匹配以 `/user-` 开头的任意路径
  path: '/user-*'
}
```

当使用通配符路由时，请确保路由的顺序是正确的，也就是说**含有通配符的路由应该放在最后**。路由 `{ path: '*' }` 通常用于客户端 404 错误。

当使用一个通配符时，$route.params 内会自动添加一个名为 pathMatch 参数。
它包含了 URL 通过通配符被匹配的部分

高级匹配模式
vue-router 使用 path-to-regexp 作为路径匹配引擎，所以支持很多高级的匹配模式，例如：可选的动态路径参数、匹配零个或多个、一个或多个，甚至是自定义正则匹配。

嵌套路由

```
<div id="app">
  <router-view></router-view>
</div>
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}

const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User }
  ]
})
```

这里的 `<router-view>` 是最顶层的出口，**渲染最高级路由匹配到的组件**。同样地，一个被渲染组件同样可以包含自己的嵌套 `<router-view>`

```
routes: [
    { path: '/user/:id', component: User,
      children: [
        {
          // 当 /user/:id/profile 匹配成功，
          // UserProfile 会被渲染在 User 的 <router-view> 中
          path: 'profile',
          component: UserProfile
        },
        {
          // 当 /user/:id/posts 匹配成功
          // UserPosts 会被渲染在 User 的 <router-view> 中
          path: 'posts',
          component: UserPosts
        }
      ]
    }
```

注意，**以 / 开头的嵌套路径会被当作根路径。** 这让你充分的使用嵌套组件而无须设置嵌套的路径。

编程式的导航

除了使用 `<router-link>` 创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过编写代码来实现。

router.push(location, onComplete?, onAbort?)

注意：在 Vue 实例内部，你可以通过 $router 访问路由实例。因此你可以调用 this.$router.push

想要导航到不同的 URL，则使用 router.push 方法。**这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，则回到之前的 URL。**

当你点击 <router-link> 时，这个方法会在内部调用，所以说，**点击 <router-link :to="..."> 等同于调用 router.push(...)**。

声明式	编程式
<router-link :to="...">	router.push(...)

```
// 字符串
router.push('home')

// 对象
router.push({ path: 'home' })

// 命名的路由
router.push({ name: 'user', params: { userId: '123' }})

// 带查询参数，变成 /register?plan=private
router.push({ path: 'register', query: { plan: 'private' }})
```

注意：**如果提供了 path，params 会被忽略**，上述例子中的 query 并不属于这种情况。取而代之的是下面例子的做法，你需要**提供路由的 name 或手写完整的带有参数的 path**：

```
const userId = '123'
router.push({ name: 'user', params: { userId }}) // -> /user/123
router.push({ path: `/user/${userId}` }) // -> /user/123
// 这里的 params 不生效
router.push({ path: '/user', params: { userId }}) // -> /user
```

同样的规则也适用于 router-link 组件的 to 属性

router.replace(location, onComplete?, onAbort?)

跟 router.push 很像，唯一的不同就是，它**不会向 history 添加新记录，而是替换掉当前的 history 记录**。

声明式	编程式
<router-link :to="..." replace>	router.replace(...)

router.go(n)

这个方法的参数是一个整数，意思是**在 history 记录中向前或者后退多少步**，类似 window.history.go(n)。

```
// 在浏览器记录中前进一步，等同于 history.forward()
router.go(1)

// 后退一步记录，等同于 history.back()
router.go(-1)

// 前进 3 步记录
router.go(3)

// 如果 history 记录不够用，那就默默地失败呗
router.go(-100)
router.go(100)
```

操作 History
router.push 、 router.replace 和 router.go 跟 window.history.pushState 、 window.history.replaceState 和 window.history.go 好像， 实际上它们确实是效仿 window.history API 的

命名路由

通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由

```
{
      path: '/user/:userId',
      name: 'user',
      component: User
    }
```

要链接到一个命名路由，可以给 router-link 的 to 属性传一个对象：

```
<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>
```

命名视图

有时候想同时（同级）展示多个视图，而不是嵌套展示，例如创建一个布局，有 sidebar（侧导航）和 main（主内容）两个视图，这个时候命名视图就派上用场了。你可以在界面中拥有多个单独命名

的视图，而不是只有一个单独的出口。

如果 router-view 没有设置名字，那么默认为 default。

```
<router-view class="view one"></router-view>
<router-view class="view two" name="a"></router-view>
<router-view class="view three" name="b"></router-view>
```

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置 (带上 s):

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      components: {
        default: Foo,
        a: Bar,
        b: Baz
      }
    }
  ]
})
```

嵌套命名视图

我们也有可能使用命名视图创建嵌套视图的复杂布局。这时你也需要命名用到的嵌套 router-view 组件。

```
<!-- UserSettings.vue -->
<div>
  <h1>User Settings</h1>
  <NavBar/>
  <router-view/>
  <router-view name="helper"/>
</div>
```

```
{
  path: '/settings',
  // 你也可以在顶级路由就配置命名视图
  component: UserSettings,
  children: [{
    path: 'emails',
    component: UserEmailsSubscriptions
  }, {
    path: 'profile',
    components: {
      default: UserProfile,
      helper: UserProfilePreview
    }
  }]
}
```

重定向和别名

#重定向

重定向也是通过 routes 配置来完成，下面例子是从 /a 重定向到 /b:

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: '/b' }
  ]
})
```

重定向的目标也可以是一个命名的路由

```
  { path: '/a', redirect: { name: 'foo' }}
```

甚至是一个方法，动态返回重定向目标:

```
{ path: '/a', redirect: to => {
      // 方法接收 目标路由 作为参数
      // return 重定向的 字符串路径/路径对象
    }}
```

别名

a 的别名是 /b，意味着，当用户访问 /b 时，URL 会保持为 /b，但是路由匹配则为 /a，就像用户访问 /a 一样。

```
routes: [
    { path: '/a', component: A, alias: '/b' }
  ]
```

路由组件传参

在组件中使用 $route 会使之与其对应路由形成高度耦合，从而使组件只能在某些特定的 URL 上使用

使用 props 将组件和路由解耦:

```
取代与 $route 的耦合

const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User }
  ]
})
```

```
通过 props 解耦

const User = {
  props: ['id'],
  template: '<div>User {{ id }}</div>'
}
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User, props: true },

    // 对于包含命名视图的路由，你必须分别为每个命名视图添加 `props` 选项:
```

```
    {
      path: '/user/:id',
      components: { default: User, sidebar: Sidebar },
      props: { default: true, sidebar: false }
    }
  ]
})
```

布尔模式

如果 props 被设置为 true，route.params 将会被设置为组件属性

对象模式

如果 props 是一个对象，它会被按原样设置为组件属性。当 props 是静态的时候有用。

```
const router = new VueRouter({
  routes: [
    { path: '/promotion/from-newsletter', component: Promotion, props: { newsletterPopup:
false } }
  ]
})
```

函数模式

你可以创建一个函数返回 props。这样你便可以将参数转换成另一种类型，将静态值与基于路由的值结合等等。

```
const router = new VueRouter({
  routes: [
    { path: '/search', component: SearchUser, props: (route) => ({ query:
route.query.q }) }
  ]
})
```

HTML History 模式

vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。

如果不想要很丑的 hash，我们可以用路由的 history 模式，这种模式充分利用 history.pushState API 来完成 URL 跳转而无须重新加载页面

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id

导航守卫

vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。有多种机会植入路由导航过程中：全局的，单个路由独享的，或者组件级的。

全局前置守卫
你可以使用 router.beforeEach 注册一个全局前置守卫：

```
const router = new VueRouter({ ... })

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

当一个导航触发时，全局前置守卫按照创建顺序调用。守卫是异步解析执行，此时导航在所有守卫 resolve 完之前一直处于 **等待中**。
每个守卫方法接收三个参数：

- **to: Route**: 即将要进入的目标 路由对象
- **from: Route**: 当前导航正要离开的路由
- **next: Function**: 一定要调用该方法来 **resolve** 这个钩子。执行效果依赖 next 方法的调用参数。
    - **next()**: 进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 **confirmed** (确认的)。
    - **next(false)**: 中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 from 路由对应的地址。
    - **next('/') 或者 next({ path: '/' })**: 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。你可以向 next 传递任意位置对象，且允许设置诸如 replace: true、name: 'home' 之类的选项以及任何用在 router-link 的 to prop 或 router.push 中的选项。
    - **next(error)**: (2.4.0+) 如果传入 next 的参数是一个 Error 实例，则导航会被终止且该错误会被传递给 router.onError() 注册过的回调。

路由元信息

过渡动效

数据获取

滚动行为

路由懒加载

导航故障