vue之路由

- 安装
- 基础
    - 开始
    - 动态路由匹配
    - 嵌套路由
    - 编程式导航
    - 命名路由
    - 命名视图
    - 重定向 和 别名
    - HTML5 History 模式
- 进阶
    - 导航钩子
    - 路由元信息
    - 过渡动效
    - 数据获取
    - 滚动行为
    - 懒加载
- API 文档
    - router-link
    - router-view
    - 路由信息对象
    - Router 构造配置
    - Router 实例
    - 对组件注入

1.安装

直接下载 / CDN

https://unpkg.com/vue-router/dist/vue-router.js

Unpkg.com 提供了基于 NPM 的 CDN 链接。上面的链接会一直指向在 NPM 发布的最新版本。你也可以像 https://unpkg.com/vue-router@2.0.0/dist/vue-router.js 这样指定 版本号 或者 Tag。

在 Vue 后面加载 vue-router，它会自动安装的：

```
<script src="/path/to/vue.js"></script>
<script src="/path/to/vue-router.js"></script>
```

NPM

```
npm install vue-router
```

如果在一个模块化工程中使用它，必须要通过 Vue.use() 明确地安装路由功能：

```
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)
```

如果使用全局的 script 标签，则无须如此（手动安装）。

构建开发版

如果你想使用最新的开发版，就得从 GitHub 上直接 clone，然后自己 build 一个 **vue-router**。

```
git clone https://github.com/vuejs/vue-router.git node_modules/vue-router
 cd node_modules/vue-router
 npm install
 npm run build
```

2.基础

2.1开始

> 教程中的案例代码将使用 **ES2015** 来编写。

用 **Vue.js + vue-router** 创建单页应用，是非常简单的。使用 Vue.js 时，我们就已经把组件组合成一个应用了，当你要把 **vue-router** 加进来，只需要配置组件和路由映射，然后告诉 vue-router 在哪里渲染它们。下面是个基本例子：

HTML

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">
  <h1>Hello App!</h1>
  <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>
  <!-- 路由出口 -->
  <!-- 路由匹配到的组件将渲染在这里 -->
  <router-view></router-view>
</div>
```

JS

```
// 0. 如果使用模块化机制编程，導入Vue和VueRouter，要调用 Vue.use(VueRouter)

// 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')

// 现在，应用已经启动了！
```

参考 **routerStart.html**

> 要注意，当 <router-link>对应的路由匹配成功，将自动设置 **class** 属性值 **.router-link-active**。查看 API 文档 学习更多相关内容。

2.2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。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 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 User = {
  template: '<div>User</div>'
}

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

现在呢，像 **/user/foo** 和 **/user/bar** 都将映射到相同的路由。
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。于是，我们可以更新 User 的模板，输出当前用户的 ID：(参考**params.html**)

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

你可以在一个路由中设置多段『路径参数』，对应的值都会设置到 **$route.params** 中。例如：

模式	**匹配路径**	**$route.params**
user/:username	/user/evan	{ username: ‘evan’ }
/user/:username/post/:post_id	/user/evan/post/123	{ username: ‘evan’, post_id: 123 }

除了 **$ route.params** 外,**$route** 对象还提供了其它有用的信息，例如，**$route.query**（如果 URL 中有查询参数）、**$route.hash** 等等。你可以查看 **API 文档** 的详细说明。

响应路由参数的变化

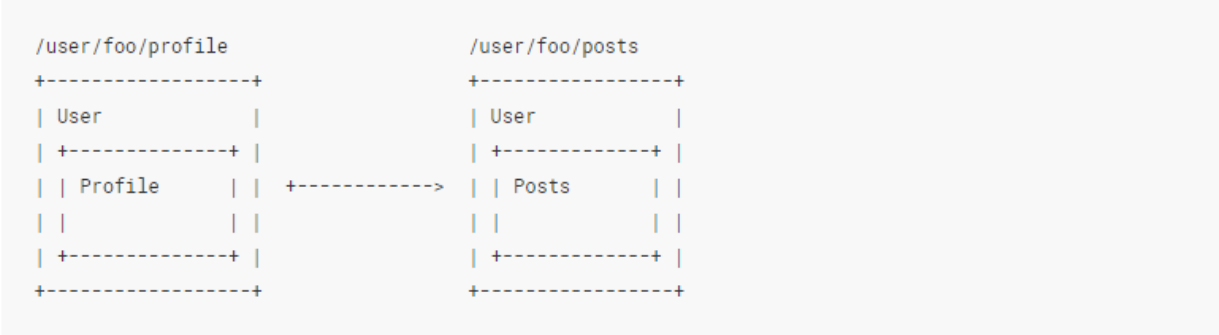
> 提醒一下，当使用路由参数时，例如从 /user/foo 导航到 user/bar，**原来的组件实例会被复用**。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着**组件的生命周期钩子不会再被调用**。

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...
    }
  }
}
```

2.3嵌套路由

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件，例如：



借助 **vue-router**，使用嵌套路由配置，就可以很简单地表达这种关系。
接着上节创建的 app：

```
<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>**。例如，在 User 组件的模板添加一个 **<router-view>**：

```
const User = {
  template: `
    <div class="user">
      <h2>User {{ $route.params.id }}</h2>
      <router-view></router-view>
    </div>
  `
}
```

要在嵌套的出口中渲染组件，需要在 VueRouter 的参数中使用 children 配置：

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

要注意，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。

你会发现，**children** 配置就是像 **routes** 配置一样的路由配置数组，所以呢，你可以嵌套多层路由。

此时，基于上面的配置，当你访问 /user/foo 时，User 的出口是不会渲染任何东西，这是因为没有匹配到合适的子路由。如果你想要渲染点什么，可以提供一个 空的 子路由：

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:id', component: User,
      children: [
        // 当 /user/:id 匹配成功,
        // UserHome 会被渲染在 User 的 <router-view> 中
        { path: '', component: UserHome },

        // ...其他子路由
      ]
    }
  ]
})
```

参考案例(**nest.html**)

2.5编程式导航

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)**

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(...)

该方法的参数可以是一个字符串路径，或者一个描述地址的对象。例如：

```
// 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' }})
```

router.replace(location)

跟 **router.push** 很像，唯一的不同就是，它不会向 **history** 添加新记录，而是跟它的方法名一样 —— 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)
```

2.6命名路由

有时候，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，或者是执行一些跳转的时候。你可以在创建 **Router** 实例的时候，在 **routes** 配置中给某个路由设置名称。

```
const router = new VueRouter({
  routes: [
    {
      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>
```

这跟代码调用 **router.push()** 是一回事：

```
router.push({ name: 'user', params: { userId: 123 }})
```

这两种方式都会把路由导航到 /user/123 路径。(参考**name.html**)

2.7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）：

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

参考 (**nameView.html**)

2.8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：

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

甚至是一个方法，动态返回重定向目标：

```
const router = new VueRouter({
  routes: [
    { 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
    }}
  ]
})
```

参考(**redirectFn.html**)

别名

> 『重定向』的意思是，当用户访问 /a时，URL 将会被替换成 /b，然后匹配路由为 /b，那么『别名』又是什么呢？
> /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。

上面对应的路由配置为：

```
const router = new VueRouter({
  routes: [
    { path: '/a', component: A, alias: '/b' }
  ]
```

『别名』的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。(参考**rename.html**)

2.9HTML5 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，也好看！

> 不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

2.9.1警告

给个警告，因为这么做以后，你的服务器就不再返回 404 错误页面，因为对于所有路径都会返回 index.html 文件。为了避免这种情况，你应该在 Vue 应用里面覆盖所有的路由情况，然后在给出一个 404 页面。

```
const router = new VueRouter({
  mode: 'history',
  routes: [
    { path: '*', component: NotFoundComponent }
  ]
})
```

3.进阶

3.1导航钩子

> 正如其名，vue-router 提供的**导航钩子**主要用来**拦截导航**，让它完成**跳转或取消**。有多种方式可以在路由导航发生时执行钩子：全局的, 单个路由独享的, 或者组件级的。

3.1.1全局钩子

你可以使用 **router.beforeEach** 注册一个全局的 before 钩子：

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

当一个导航触发时，全局的 before 钩子按照创建顺序调用。钩子是异步解析执行，此时导航在所有钩子 resolve 完之前一直处于 **等待中**。
每个钩子方法接收三个参数：

- 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。

确保要调用 next 方法，否则钩子就不会被 resolved。

参考**beforeEach.html**以及**beforeEach1.html**(跳转到不同地址)和**beforeEach2.html**(跳转到不同地址正确)

同样可以注册一个全局的 after 钩子，不过它不像 **before** 钩子那样，**after** 钩子没有 **next** 方法，不能改变导航：

```
router.afterEach(route => {
  // ...
})
```

3.1.2某个路由独享的钩子

你可以在路由配置上直接定义 **beforeEnter** 钩子：

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

这些钩子与全局 before 钩子的方法参数是一样的。参考(**singleHook.html**)

3.1.3组件内的钩子

最后，你可以使用 **beforeRouteEnter** 和 **beforeRouteLeave**，在路由组件内直接定义路由导航钩子，

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不！能！获取组件实例 `this`
    // 因为当钩子执行前，组件实例还没被创建
  },
  beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
  }
}
```

beforeRouteEnter 钩子 不能 访问 **this**，因为钩子在导航确认前被调用,因此即将登场的新组件还没被创建。
不过，你可以通过传一个回调给 **next来访问组件实例**。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数。

```
beforeRouteEnter (to, from, next) {
  next(vm => {
    // 通过 `vm` 访问组件实例
  })
}
```

你可以 在 beforeRouteLeave 中直接访问 this(参考**componentHook.html**)。这个 **leave** 钩子通常用来禁止用户在**还未保存修改前突然离开**(参考**componentHook.html1**)。可以通过 **next(false)** 来取消导航。

3.2路由元信息

定义路由的时候可以配置 meta 字段：

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      children: [
        {
          path: 'bar',
          component: Bar,
          // a meta field
          meta: { requiresAuth: true }
        }
      ]
    }
  ]
})
```

那么如何访问这个 meta 字段呢？

首先，我们称呼 routes 配置中的每个路由对象为 **路由记录**。路由记录可以是嵌套的，因此，当一个路由匹配成功后，他可能匹配多个路由记录
例如，根据上面的路由配置，/foo/bar 这个 URL 将会匹配父路由记录以及子路由记录。
一个路由匹配到的所有路由记录会暴露为 $route 对象（还有在导航钩子中的 route 对象）的 $route.matched 数组。因此，我们需要遍历 $route.matched 来检查路由记录中的 meta 字段。
下面例子展示在全局导航钩子中检查 meta 字段：

```
router.beforeEach((to, from, next) => {
  if (to.matched.some(record => record.meta.requiresAuth)) {
    // this route requires auth, check if logged in
    // if not, redirect to login page.
    if (!auth.loggedIn()) {
      next({
        path: '/login',
        query: { redirect: to.fullPath }
      })
    } else {
      next()
    }
  } else {
    next() // 确保一定要调用 next()
  }
})
```

参考(**roundInf.html**)

3.3过渡动效

<router-view> 是基本的动态组件，所以我们可以用 **<transition>** 组件给它添加一些过渡效果：

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

<transition>的所有功能 在这里同样适用。

3.3.1单个路由的过渡

上面的用法会给所有路由设置一样的过渡效果，如果你想让每个路由组件有各自的过渡效果，可以在各路由组件内使用 **<transition>**并设置不同的 name。

```
const Foo = {
  template: `
    <transition name="slide">
      <div class="foo">...</div>
    </transition>
  `
}

const Bar = {
  template: `
    <transition name="fade">
      <div class="bar">...</div>
    </transition>
  `
}
```

3.3.2基于路由的动态过渡

还可以基于当前路由与目标路由的变化关系，动态设置过渡效果：

```
<!-- 使用动态的 transition name -->
<transition :name="transitionName">
  <router-view></router-view>
</transition>
// 接着在父组件内
// watch $route 决定使用哪种过渡
watch: {
  '$route' (to, from) {
    const toDepth = to.path.split('/').length
    const fromDepth = from.path.split('/').length
    this.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-le
ft'
  }
}
```

参考(**transition.html**)

3.4数据获取

有时候，进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现：

- **导航完成之后获取：**先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示『加载中』之类的指示。
- **导航完成之前获取：**导航完成前，在路由的 enter 钩子中获取数据，在数据获取成功后执行导航。

从技术角度讲，两种方式都不错 —— 就看你想要的用户体验是哪种。

3.4.1导航完成后获取数据

当你使用这种方式时，我们会马上导航和渲染组件，然后在组件的 created 钩子中获取数据。这让我们有机会在数据获取期间展示一个 loading 状态，还可以在不同视图间展示不同的 loading 状态。
假设我们有一个 Post 组件，需要基于 $route.params.id 获取文章数据：

```
<template>
  <div class="post">
    <div class="loading" v-if="loading">
      Loading...
    </div>

    <div v-if="error" class="error">
      {{ error }}
    </div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>
```

```
export default {
  data () {
    return {
      loading: false,
      post: null,
      error: null
    }
  },
  created () {
    // 组件创建完后获取数据，
    // 此时 data 已经被 observed 了
    this.fetchData()
  },
  watch: {
    // 如果路由有变化，会再次执行该方法
    '$route': 'fetchData'
  },
  methods: {
    fetchData () {
      this.error = this.post = null
      this.loading = true
      // replace getPost with your data fetching util / API wrapper
      getPost(this.$route.params.id, (err, post) => {
        this.loading = false
        if (err) {
          this.error = err.toString()
        } else {
          this.post = post
        }
      })
    }
  }
}
```

参考(**data.html**)

3.4.2在导航完成前获取数据

通过这种方式，我们在导航转入新的路由前获取数据。我们可以在接下来的组件的 beforeRouteEnter 钩子中获取数据，当数据获取成功后只调用 next 方法。

```
export default {
  data () {
    return {
      post: null,
      error: null
    }
  },
  beforeRouteEnter (to, from, next) {
    getPost(to.params.id, (err, post) =>
      if (err) {
        // display some global error message
        next(false)
      } else {
        next(vm => {
          vm.post = post
        })
      }
    })
  },
  // 路由改变前，组件就已经渲染完了
  // 逻辑稍稍不同
  watch: {
    $route () {
      this.post = null
      getPost(this.$route.params.id, (err, post) => {
        if (err) {
          this.error = err.toString()
        } else {
          this.post = post
        }
      })
    }
  }
}
```

在为后面的视图获取数据时，用户会停留在当前的界面，因此建议在数据获取期间，显示一些进度条或者别的指示。如果数据获取失败，同样有必要展示一些全局的错误提醒。

3.5滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。 **vue-router** 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。

> 注意: 这个功能只在 **HTML5 history** 模式下可用。

当创建一个 Router 实例，你可以提供一个 **scrollBehavior** 方法：

```
const router = new VueRouter({
  routes: [...],
  scrollBehavior (to, from, savedPosition) {
    // return 期望滚动到哪个的位置
  }
})
```

scrollBehavior 方法接收 **to** 和 **from** 路由对象。第三个参数 **savedPosition** 当且仅当 **popstate** 导航 (通过浏览器的 前进/后退 按钮触发) 时才可用。

这个方法返回滚动位置的对象信息，长这样：

- { x: number, y: number }
- { selector: string }

如果返回一个布尔假的值，或者是一个空对象，那么不会发生滚动。
举例：

```
scrollBehavior (to, from, savedPosition) {
  return { x: 0, y: 0 }
}
```

对于所有路由导航，简单地让页面滚动到顶部。

返回 **savedPosition**，在按下 后退/前进 按钮时，就会像浏览器的原生表现那样：

```
scrollBehavior (to, from, savedPosition) {
  if (savedPosition) {
    return savedPosition
  } else {
    return { x: 0, y: 0 }
  }
}
```

如果你要模拟『滚动到锚点』的行为：

```
scrollBehavior (to, from, savedPosition) {
  if (to.hash) {
    return {
      selector: to.hash
    }
  }
}
```

3.6懒加载

当打包构建应用时，Javascript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

结合 Vue 的 **异步组件** 和 Webpack 的 **code splitting feature**, 轻松实现路由组件的懒加载。
我们要做的就是把路由对应的组件定义成异步组件：

```
const Foo = resolve => {
  // require.ensure 是 Webpack 的特殊语法，用来设置 code-split point
  // （代码分块）
  require.ensure(['./Foo.vue'], () => {
    resolve(require('./Foo.vue'))
  })
}
```

这里还有另一种代码分块的语法，使用 AMD 风格的 require，于是就更简单了：

```
const Foo = resolve => require(['./Foo.vue'], resolve)
```

不需要改变任何路由配置，跟之前一样使用 Foo：

```
const router = new VueRouter({
  routes: [
    { path: '/foo', component: Foo }
  ]
})
```

把组件按组分块

有时候我们想把某个路由下的所有组件都打包在同个异步 chunk 中。只需要 给 chunk 命名，提供 require.ensure 第三个参数作为 **chunk** 的名称:

```
const Foo = r => require.ensure([], () => r(require('./Foo.vue')), 'group
-foo')
const Bar = r => require.ensure([], () => r(require('./Bar.vue')), 'group
-foo')
const Baz = r => require.ensure([], () => r(require('./Baz.vue')), 'group
-foo')
```

Webpack 将相同 chunk 下的所有异步模块打包到一个异步块里面 —— 这也意味着我们无须明确列出 require.ensure 的依赖（传空数组就行）。

4.参考文献

1. 参考demo github地址
2. 参考文档 http://router.vuejs.org/