Vue Router 文档

Catalog

        - 4.1.1 全局前置守卫
        - 4.1.2 全局解析守卫
        - 4.1.3 全局后置钩子
        - 4.1.4 路由独享守卫
        - 4.1.5 组件内的守卫
        - 4.1.6 完整的导航解析流程
    - 4.2 路由元信息
    - 4.3 过渡动效
        - 4.3.1 单个路由的过渡
        - 4.3.2 基于路由的动态过渡
    - 4.4 数据获取
        - 4.4.1 导航完成后获取数据
        - 4.4.2
    - 4.5 滚动行为
    - 4.6 路由懒加载

New Words

- **route [rut] --n.路线; 航线; 路径; 线路;**
    - an air route. 航空线路
    - The route passes close by the town. 这条路从镇子旁边经过.
    - We stopped for a picnic en route. 我们中途停下来野餐.
    - They marched the route in a day. 他们在一天内走完了全程.
    - The route is shorter than this one. 那条路比这条短.
- **discrete [dɪ'skriːt] --adj. 分离的, 不相关联的**
- **authorization [ɔːθəraɪ'zeɪʃ(ə)n]{UK} --n.授权, 批准**
    - Seeking authorization to begin construction. 请求批准开始建造.
- **guard [gɑrd] --vt.守护,守卫; 看守. --vi.留心. --n.看守; 卫兵; 把守; 警备.**
    - navigator guards 导航守卫
    - guard(vt) one's life. 守护性命
    - A watchdog guarded(vt) the house against thieves. 看门狗保护房子以防小偷.
    - He guarded(vt) us from all harm. 他保护我们免于收到任何伤害.
    - You should guarded(vi) against accidents [catching a cold]. 你应当留心防止意外[感冒].
- **resolve [rɪ'zɒlv]/[rɪ'zɑlv] --v.解决; 解析; 分解. --vi.决心, 决定. --n.坚决; 决心.**
    - I resolved(vt) to lose weight. 我下决心减肥.

    - She resolved(vi) against going. 她决心不去.
    - They resolved(vi) on [against] going back the same way. 它们决定原路回去[不循原路回去].
    - He made a resolve(n) to stop smoking. 他决心戒烟.
    - a man of high resolve(n) 坚毅[果断]的人.
- called before the route that renders this component is confirmed. (在确认呈现此组件的路由之前调用)
- called when the route that renders this component has changed. (当呈现此组件的路由已更改时调用)
- called when the route that renders this component is about to be navigated away from. (当呈现此组件的路由即将离开时调用)

Pre-knowledge Content

- 如果你对下面几个知识点不熟悉, 请一定先看下当前目录的: `./前端路由实现原理/前端路由实现原理.md`
    - (1) history 对象
    - (2) history 对象上的 `length` / `state` 属性, `history.forward()` / `history.back()` / `history.go()` 方法
    - (3) `hashchange` 事件
    - (4) `pushState()` / `replaceState()` 方法
    - (5) `popstate` 事件

Content

1. 安装

1.1 直接下载 / CDN

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

  npkg.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>
```

1.2 NPM

- shell 脚本:

```
npm install vue-router
```

如果在一个模块化工程中使用它, 必修通过 `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).

1.3 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. 介绍

> 版本说明: 对于 TypeScript 用户来说, `vue-router@3.0+` 依赖 `vue@2.5+`, 反之亦然.

- Vue Router 是 **Vue.js** 官方的路由管理器. 它和 Vue.js 的核心深度集成, 让构建单页面引用变得易如反掌. 包含的功能有:
  - (1) 嵌套的路由/视图表
  - (2) 模块化的, 基于组件的路由配置
  - (3) 路由参数, 查询, 通配符
  - (4) 基于 Vue.js 过渡系统的试图过渡效果
  - (5) 细粒度的导航控制
  - (6) 带有自动激活的 CSS class 的链接
  - (7) HTML5 历史模式(history) 模式或 hash 模式, 在 IE9 中自动降级
  - (8) 自定义的滚动条行为 现在开始 **起步** 或尝试一下我们的 **示例** 吧 (常看仓库的 `README.md` 来运行它们).
- 添加上面 `示例` (examples) 下的各个文件的中英对照:
  - (1) `active-links` (激活连接)
  - (2) `auth-flow` (授权流程)
  - (3) `basic` (基础的)
  - (4) `data-fetching` (获取数据)
  - (5) `discrete-components` (分离组件)
  - (6) `hash-mode` (哈希模式)

    - (7) `hash-scroll-behavior`(哈希滚动行为)
    - (8) `keepalive-view`(keepalive 视图)
    - (9) `lazy-loading`(懒加载)
    - (10) `lazy-loading-before-mount`(懒加载编译之前)
    - (11) `multi-app`(多应用)
    - (12) `named-routes`(命名路线)
    - (13) `named-views`(命名视图)
    - (14) `navigation-guards`(导航守卫)
    - (15) `nested-router`(嵌套路由器)
    - (16) `nested-routes`(嵌套路线)
    - (17) `redirect`(重定向)
    - (18) `route-alias`(路线别名)
    - (19) `route-props`(路线 prop)
    - (20) `router-errors`(路由错误)
    - (21) `transitions`(过渡)
- **Tip:** 在 vue-router 中 `router` 是 VueRouter 的实例, `route` 是路由信息对象,

3. 基础

> 注意: 教程中的案例代码使用 **ES2015** 来编写.

> 同时, 所有的例子都将使用完整版的 Vue 以解析模板. 更多细节请 `移步这里`(即: Vue 官网教程 `../../01-Vue.js/01-基础.md -- 1.1 安装`)

3.1 起步

- 用 Vue.js + Vue Router 创建单页应用, 是非常简单的. 使用 Vue.js, 我们已经可以通过组合组件来组成应用程序, 当你要把 Vue Router 添加进来, 我们需要做的是, 将组件 (components) 映射到路由 (routes), 然后告诉 Vue Router 在哪里渲染它们. 下面是个基本例子:
- 因为当前示例是使用完整版(即: 通过 `script` 标签来引入 vue 和 vue-router)的, 所以此处就不再粘贴代码了, 想看 html 页面写法的直接看官网教程即可, 下面把官网的示例改为单文件组件的写法. 详细代码见: `../../../Vue-Examples/vue-router-document-example`
    - (1) 先在 components 文件夹下创建 3.1 文件夹, 在其内部创建 `3.1.vue` 和 2 个子组件(路由组件) `foo.vue` 和 `bar.vue`;
    - (2) 在 `App.vue` 中导入 `3.1.vue` 组件,
    - (3) 我们在当前项目的主入口文件 `main.js` 中引入全局需要的 CSS 文件, 和 `vue-router` 插件, 然后在 new Vue() 实例中, 通过 `router` 配置参数注入路由, 从而让整个应用都有路由功能.

      通过注入路由, 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:

```
<!-- - 例如我们在 3.1.vue 中添加如下代码 -->
<template>
    // ... 其他代码
     <p>{{username}}</p>
</template>

<script>
    export default {
        name: 'Demo31',
        computed: {
            username() {
                // - 因为我们在项目主入口文件 main.js 中引入了 router 路由实例,
                //   所以可以在任何组件内通过 `this.$router`(使用见下面的
methods
                //   内 goBack 方法内) 访问到路由器;
                //   也可以通过 `this.$route` 访问到当前路由.(tip: Vue
                //   内部还是做了封装的, 具体实现以后看了源码便知.)
                console.log(this.$route);
                return null;
            }
        },
        methods: {
            goBack() {
                window.history.length > 1
                    ? this.$router.go(-1)
                    : this.$router.push('/')
            }
        }
    }
</script>
```

- (4) 最后我们在 src/router 下的 index.js 文件夹中, 导入 `App.vue` / `foo.vue` / `bar.vue`, 在 `routes` 数组中定义路由, 每个路由应该映射一个组件. `component` 属性的值便是导入的组件, 最后使用 new VueRouter() 创建 router 路由示例, 把 `routers` 传入.

3.2 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, 全都映射到同一个组件. 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 动态片段) 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, 项 `/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` 模板, 输出当前用的 ID:

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

你可以看看这个 在线例子 .

你可以在一个路由中设置多端 "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/post123	`{username: 'evan', post_id: '123'}`

除了 `$route.params` 外, `$route` 对象还提供了其他有用的信息, 例如, `$route.query`(如果 URL 中有查询参数), `$route.pash` 等等. 你可以查看 API 文档

Hint: 上面示例的单文件写法见: `../../../Vue-Examples/vue-router-document-example/src/components/3.2`

3.2.1 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(检测变化) `$route` 路由信息对象:

```
const User = {
    template; '..',
    watch: {
        // - 下面 4.1.1 全局前置守卫 会详细做讲解.
        $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()
    }
}
```

3.2.2 捕获所有路由或 404 Not found 路由

- 常规阐述只会匹配被 `/` 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
    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. 如果你使用了 History 模式, 请确保 正确配置你的服务器 .

当使用一个通配符时, `$route.params` (tip: 为一个对象, 在 `/3.2/user.vue` 中查看) 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:

```
// - 给出一个路由 { path: '/user-*' }
this.$router.push('/user-admin');
console.log(this.$route.params.pathMatch)   // 'admin'

// - 给出一个路由 { path: '*' }
this.$router.push('/non-existing');     // - 设置一个不存在的 route
console.log(this.$route.params.pathMath);   // '/non-existing'
```

3.2.3 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. 查看它的 文档 学习高阶的路径匹配, 还有 这个例子 展示 `vue-router` 怎么使用这类匹配.

3.2.4 匹配优先级

- 有时候, 同一个路径可以匹配多个路由, 此时, 匹配的优先级就按照路由的定义顺序: 谁先定义的, 谁的优先级就最高.

3.3 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:

```
/user/foo/profile                     /user/foo/posts
+------------------+                  +-----------------+
| User             |                  | User            |
| +--------------+ |                  | +-------------+ |
| | Profile      | |  +------------>  | | Posts       | |
| |              | |                  | |             | |
| +--------------+ |                  | +-------------+ |
+------------------+                  +-----------------+
```

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 不会被渲染在
                    //   User 的 <router-view> 中
                    {
                        path: '',
                        component: UserHome
                    },
                ]
            }
        ]
    })
```

提供以上案例的可运行代码请 移步这里

Hint: 此章节的单组件示例详见: `../../../Vue-Examples/vue-router-document-example/src/components/3.3`.

在 `3.3.vue` 组件中, 因为 `user` 作为路径字符已经在 `3.2.vue` 中使用过了, 所以我更改 `user` 为 `worker`.

3.4 编程式的导航(Programmatic Navigator)

- 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.

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

- **注意:** 在 Vue 实例内部, 你可以通过 `$router` 来访问路由实例. 因此(在组建内)你可以调用 `this.$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(...)`

  Tip: 请注意, 此处 `<router-link :to="{path: '/xxx'}">` 内的 `:to` 属性是动态绑定的, 这大概是为什么称为编程式导航的原因, 上面的示例内 `to` 属性都是静态的.

  Additional Info: 比如在单组件内使用, 举个例子:

```
methods: {
    selectSinger(singer) {
        this.$router.push({
            path: `/singer/${singer.id}`
        });
        // {avatar: "https://xxxx", id: "0025Nh1xxx", name: "xxx"}
        console.log("singer 数据内容为: ", singer);
    },
}
```

  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: 如上示例代码.
router.push({path: 'home'});

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

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

  注意: **如果提供了 `path`, `params` 会被忽略**(请特别注意!), 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.

在 2.2.0+, 可选的在 `router.push` 或 `router.replace` 中提供 `onComplete` 和 `onAbort` 回调作为第 2 个和 第 3 个参数. 这些回调将会在导航成功完成(在所有的异步钩子被解析之后)或终止(导航到相同的路由, 或在当前导航完成之前导航到另一个不同的路由)的时候进行相应的调用. 在 3.1.0+, 可以省略第二个和第三个参数, 此时如果支持 Promise, `router.push` 或 `router.replace` 将返回一个 Promise.

注意: 如果目的地和当前路由相同, 只有(查询)参数(query)发生了改变 (比如从一个用户资料到另一个 `/users/1` -> `/users/2`), 你需要使用 `beforeRouteUpdate` 来响应这个变化(比如抓取用户信息.)

3.4.2 `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(...)`

3.4.3 `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)`. 例子:

```
// 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)
```

3.4.4 操作 History

- 你也许注意到 `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.

因此, 如果你已经熟悉 **Browser History APIs**, 那么在 Vue Router 中操作 history 就是超级简单的.

还有值得提及的, Vue Router 的导航方法(`push` 、 `replace` 、 `go`) 在各类路由模式(`history` 、 `hash` 和 `abstract`) 下表现一致.

- **Additional Info:** 本章节的完整的单组件示例在: `../../../Vue-Examples/vue-router-document-example/src/components/3.4/3.4.vue` 中. 请一定自己写一下. 下面贴出 `3.4.vue` 关于编程式导航的部分.

```
<template>
    <div class="default">
        <h2>3.4 编程式导航</h2>

      <span class="link-span">
            <!-- to="/runner/li" -->
            <!-- 注意: 这里是 a 标签, 不能直接写 router-link -->
            <a @click='runner("li")'>/runner/li</a>
        </span>
        <span class="link-span">
            <!-- to="/runner/wang" -->
            <a @click='runner("wang")'>/runner/wang</a>
        </span>

      <router-view></router-view>
    </div>
</template>

<script>
    export default {
        name: 'Demo34',
        methods: {
            runner(person) {
                this.$router.push({path: `/runner/${person}`});
            }
        }
    }
</script>
```

3.5 命名路由(Named Views)

- 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.

完整的例子请 移步这里 .

下面为上一行示例代码的 copy: (tip: 代码结构比较简单, 不再添加单文件组件写法.)

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

Vue.use(VueRouter)

const Home = { template: '<div>This is Home</div>' }
const Foo = { template: '<div>This is Foo</div>' }
const Bar = { template: '<div>This is Bar {{ $route.params.id }}</div>' }

const router = new VueRouter({
mode: 'history',
base: __dirname,
routes: [
    { path: '/', name: 'home', component: Home },
    { path: '/foo', name: 'foo', component: Foo },
    { path: '/bar/:id', name: 'bar', component: Bar }
]
})

new Vue({
router,
template: `
    <div id="app">
    <h1>Named Routes</h1>
    <p>Current route name: {{ $route.name }}</p>
    <ul>
        <li><router-link :to="{ name: 'home' }">home</router-link></li>
```

```
          <li><router-link :to="{ name: 'foo' }">foo</router-link></li>
          <li><router-link :to="{ name: 'bar', params: { id: 123
}}">bar</router-link></li>
      </ul>
      <router-view class="view"></router-view>
      </div>
  `
  }).$mount('#app')
```

3.6 命名视图(Named Views)

- 有时候你需要同时展示多个视图(multiple views), 而不是嵌套(nesting)它们, 例如. 创建一个布局, 含有 `sidebar` 视图和 `main` 视图. 这个时候命名视图派就派上用场了. 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
              }
          }
      ]
  })
  ```

  以上案例相关的可运行代码请 移步这里.

3.6.1 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. 我们以一个设置面板为例:

```
  /settings/emails                                    /settings/profile
  +-----------------------------------+               +--------------------
---------+
  | UserSettings                      |               | UserSettings
       |
  | +-----+-------------------------+ |               | +-----+------------
-------+ |
  | | Nav | UserEmailsSubscriptions | |  +----------->  | | Nav | UserProfile
     | |
  | |     +-------------------------+ |               | |     +------------
-------+ |
  | |     |                         | |               | |     |
UserProfilePreview | |
  | +-----+-------------------------+ |               | +-----+------------
-------+ |
  +-----------------------------------+               +--------------------
---------+
```

- `Nav` 只是一个常规组件.
- `UserSettings` 是一个视图组件.
- `UserEmailsSubscriptions`、`UserProfile`、`UserProfilePreview` 是嵌套的视图组件.
- **注意**: 我们先忘记 HTML/CSS 具体的布局的样子, 只专注在用到的组件上.

  `UserSettings` 组件的 `<template>` 部分应该是类似下面的这段代码:

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

嵌套的视图组件在此已经被忽略了, 但是你可以在 这里 找到完整的源代码.

然后你可以用这个路由配置完成该布局:

```
{
    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 是有 s 的!
            components: {
                default: UserProfile,
```

```
                    // - 注意上面的命名视图 name="helper", 在这里定义.
                    helper: UserProfilePreview
                }
            }
        ]
    }
```

一个可以工作的示例的 demo 在 这里 .

- **Additional Info:** 当前 `3.6.1` 的示例, 单文件组件的写法在 `../../../Vue-Examples/vue-router-document-example/src/components/3.6-2` 文件夹中, 请一定自己写一遍, 这个示例应该算是比较绕的.

3.7 重定向和别名

3.7.1 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
            }
        }
    ]
})
```

注意 导航守卫 并没有应用在跳转路由上, 而仅仅应用在其目标上. 在下面这个例子中, 为 `/a` 路由添加一个 `beforeEach` 守卫并不会有任何效果.

其它高级用法, 请参考 例子 .

3.7.2 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.

  更多高级用法, 请查看 例子 .

3.8 传递 Props 到路由组件 (Passing Props to Router Components)

- > Notice: 如你所见, 本章的标题 `路由组件传参` 被我改成了 `传递 Props 到路由组件`, vue-router 的英文文档的标题是 `Passing Props to Router Component`, 我觉得直译更简单易懂.

- 在组件中使用 `$route` 会与路由紧密耦合, 从而限制了组件的灵活性, 因为它只能在某些 URL 上使用.

  要将组件与路由解耦, 请使用 `props` 选项:

  代替 `$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 }
        }
    ]
})
```

这样你便可以在任何地方使用该组件, 使得该组件更易于重用和测试.

3.8.1 布尔模式(Boolean mode)

- 如果 `props` 被设置为 `true`, `route.params` 将会被设置为组件 props(属性).

3.8.2 对象模式(Object mode)

- 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 }
        }
    ]
})
```

3.8.3 函数模式(Function mode)

- 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 })
        }
    ]
})
```

URL `/search?q=vue` 会将 `{query: 'vue'}` 作为属性传递给 `SearchUser` 组件.

请尽可能保持 `props` 函数为无状态的, 因为它只会在路由发生变化时起作用. 如果你需要状态来定义 `props`, 请使用包装组件, 这样 Vue 才可以对状态变化做出反应.

更多高级用法, 请查看 例子.

3.9 HTML5 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.

3.9.1 后端配置例子

- **注意**: 下列示例假设你在根目录服务这个应用. 如果想部署到一个子目录, 你需要使用 **Vue CLI 的 `publicPath` 选项** 和相关的 **router `base` property**. 你还需要把下列示例中的根目录调整成为子目录 (例如用 `RewriteBase /name-of-your-subfolder/` 替换掉 `RewriteBase /`).
- Apache

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.html [L]
</IfModule>
```

除了 `mod_rewrite`, 你也可以使用 **`FallbackResource`**.

- nginx

```
location / {
  try_files $uri $uri/ /index.html;
}
```

- 原生 Node.js

```
const http = require('http')
const fs = require('fs')
const httpPort = 80

http.createServer((req, res) => {
  fs.readFile('index.htm', 'utf-8', (err, content) => {
    if (err) {
```

```
        console.log('We cannot open "index.htm" file.')
      }

      res.writeHead(200, {
        'Content-Type': 'text/html; charset=utf-8'
      })

      res.end(content)
    })
  }).listen(httpPort, () => {
    console.log('Server listening on: http://localhost:%s', httpPort)
  })
```

- 基于 Node.js 的 Express

  对于 Node.js/Express, 请考虑使用 **connect-history-api-fallback 中间件**.

- Internet Information Services (IIS)

  1. 安装 **IIS UrlRewrite**
  2. 在你的网站根目录中创建一个 `web.config` 文件, 内容如下:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <configuration>
  <system.webServer>
      <rewrite>
      <rules>
          <rule name="Handle History Mode and custom 404/500"
stopProcessing="true">
          <match url="(.*)" />
          <conditions logicalGrouping="MatchAll">
              <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />
              <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
negate="true" />
          </conditions>
          <action type="Rewrite" url="/" />
          </rule>
      </rules>
      </rewrite>
  </system.webServer>
  </configuration>
```

- Caddy

```
  rewrite {
      regexp .*
      to {path} /
  }
```

- Firebase 主机

  在你的 `firebase.json` 中加入:

```
{
  "hosting": {
    "public": "dist",
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}
```

- 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 }
    ]
  })
  ```

  或者, 如果你使用 Node.js 服务器, 你可以用服务端路由匹配到来的 URL, 并在没有匹配到路由的时候返回 404, 以实现回退. 更多详情请查阅 Vue 服务端渲染文档 .

4. 进阶

4.1 导航守卫/导航保护(Navigation Guards)

- 译者注: "导航(navigator)" 表示路由正在发生改变.
- 顾名思义, vue-router 提供的 "导航守卫" 主要用于通过 **重定向(跳转)** 或 **取消重定向** 来保护导航. 这里有很多方法可以把它挂载到路由导航过程中: 全局(globally), pre-route(按路由/单个路由) 或 组件内(in-component).

  请记住, **参数(params)或查询(query)的更改不会触发 进入/离开 的导航守卫**. 你可以通过 **观察 `$route` 对象**(`3.2.1 响应路由参数的变化`) 来应对这些改变, 或使用 `beforeRouteUpdate` 的组件内保护.

  - 观察 `$route` 对象的单组件示例为: `../../../Vue-Examples/vue-router-document-example/src/components/3.2/user.vue`

4.1.1 全局前置守卫(Global Before Guards)

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

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

// - before each (在每个之前)
router.beforeEach((to, from, next) => {
    // ...
})
```

每当一个导航被触发时, 全局前置守卫将按照创建顺序被调用. 守卫(Guards)是异步解析的, 并且在所有的 hook(钩子函数) 被解析完之前, 路由一直处于 `pending`(等待中).

每个守卫方法接收三个参数:

- (1) **`to: Route`**: 即将要进入的目标 **路由对象**
- (2) **`from: Route`**: 当前导航正要离开的路由
- (3) **`next: Function`**: 一定要调用该方法来 **resolve**(解析) 这个钩子. 执行效果依赖 `next` 方法的调用参数.
    - (a) **`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).
    - (b) **`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.
    - (c) **`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.
    - (d) **`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.

确保 `next` 函数在任何给定的导航守卫中都被严格调用一次. 它可以出现多于一次, 但是只能在所有的逻辑路径都不重叠的情况下, 否则钩子永远都不会被解析或报错. 这里有一个在用户未能验证身份时重定向到 `/login` 的示例:

```
// BAD
router.beforeEach((to, from, next) => {
    if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' })
    // 如果用户未能验证身份, 则 `next` 会被调用两次
    next()
})
```

```
// GOOD
router.beforeEach((to, from, next) => {
    if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' })
    else next()
})
```

4.1.2 全局解析守卫(Global Resolve Guards)

- 可以使用 `router.beforeResolve` 注册一个全局守卫. 这和 `router.beforeEach` 类似, 但不同点是, 解析守卫在导航被确认(confirmed)之前, 同时所有组件内(in-component)守卫和异步路由组件被解析之后, 才会被合理的调用.

```
// - before resolve 解析之前
router.beforeResolve((to, from, next) => {
    // ...
})
```

4.1.3 全局后置钩子(Global After Hooks)

- 你也可以注册全局后置钩子(hooks), 然而和守卫(guards)不同的是, 这些钩子不会接受 `next` 函数也不会改变导航本身.

```
// - after each(每次之后)
router.afterEach((to, from) => {
    // ...
})
```

4.1.4 路由独享守卫(Per-Route Guard)

- 你可以在路由配置上直接定义 `beforeEnter` 守卫:

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

这些守卫和全局前置守卫(`beforeEach`)的方法参数是一样的.

4.1.5 组件内的守卫(In-Component Guards)

- 最后, 你也可以在路由组件内直接定义以下路由导航守卫: (tip: 单文件写法)
  - (1) `beforeRouteEnter` (路由进入之前)
  - (2) `beforeRouteUpdate` (路由更新之前)
  - (3) `beforeRouteLeave` (路由离开之前)

```
<template>
    <div class="foo-router">
        <p class='view-p'>foo 我是 foo 组件</p>
    </div>
</template>
```

```
<script>
    export default {
        name: 'Foo',
        created() { },
        // - before route enter(路由进入之前)
        beforeRouteEnter(to, from, next) {
            // - 在确认呈现此组件的路由之前调用.
            // - 无法访问 "此(`this`)" 组件实例.
            //   因为当当前守卫(guard)被调用时, 当前组件实例还没被创建.
        },
        // - before route update(路由更新之前)
        beforeRouteUpdate(to, from, next) {
            // - 当呈现此组件的路由已更改时调用.
            // - 但此组件在新路由中重复使用. 例如: 对于具有动态参数
            //   `/foo/:id` 的路由, 当我们在 `/foo/1` 和 `/foo/2`
            //   之间导航时, 将重用相同的 `Foo` 组件实例.
            //   当发生这种情况时会调用此钩子(hook).
            // - 当前守卫可以访问组件实例 `this`
        },
        // - before route leave(路由离开之前)
        beforeRouteLeave(to, form, next) {
            // - 当呈现此组件的路由即将离开时调用.
            // - 可以访问组件实例 `this`
        }
    }
</script>
```

`beforeRouteEnter` 守卫不能访问 `this`, 因为守卫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 => {
        // access to component instance via `vm` (通过 vm 访问组件实例)
    })
}
```

注意 `beforeRouteEnter` 是支持给 `next` 传递回调的唯一守卫. 对于 `beforeRouteUpdate` 和 `beforeRouteLeave` 来说, `this` 已经可用了, 所以**不支持**传递回调, 因为没有必要了.

```
beforeRouteUpdate(to, from, next) {
    // - just use `this`
    this.name = to.params.name;
    next();
}
```

这个离开守卫通常用来禁止用户在还未保存修改前突然离开. 改导航可以通过 `next(false)` 来取消:

```
beforeRouteLeave(to, from, next) {
    const answer = window.confirm('Do you really want to leave? you
        have unsaved changes!');
    if (answer) {
        next()
    } else {
        next(false)
    }
}
```

4.1.6 完整的导航解析流程

- (1) 导航(navigation)被触发.
- (2) 在失活的组件里调用 `beforeRouteLeave` 守卫.
- (3) 调用全局的 `beforeEach` 守卫
- (4) 在重用的组件里调用 `beforeRouteUpdate` 守卫.
- (5) 在路由配置里调用 `beforeEnter`.
- (6) 解析异步路由组件.
- (7) 在被激活的组件里调用 `beforeRouteEnter`.
- (8) 调用全局的 `beforeResolve` 守卫.
- (9) 导航被确认(confirmed).
- (10) 调用全局的 `afterEach` 钩子.
- (11) 触发 DOM 更新.
- (12) 用创建好的实例调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数.

4.2 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 `$route` 对象(还有在导航守卫中的路由对象) 的 `$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守卫中检查元字段:

```
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()
    }
})
```

4.3 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.

4.3.1 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>
    `
}
```

Hint: `slide-enter-active` / `slide-leave-active` ...

4.3.2 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>
```

接着在父组件内的 `watch` 属性内监控 `$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-left';
    }
}
```

See full example here.

4.4 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, 在路由进入的守卫中获取数据, 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.

4.4.1 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 v-if="loading" class="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
                }
            })
        }
    }
}
```

4.4.2 在导航完成后获取数据

- 通过这种方式, 我们在导航转入到新的路由前获取数据. 我们可以在接下来的组件的 `beforeRouteEnter` (`4.1.5 组件内的守卫`) 守卫中获取数据, 当数据成功后只用 `next` 方法.

```
export default {
    data() {
        return {
            post: null,
            error: null
        }
    },
    beforeRouteEnter(to, from, next) {
        getPost(to.params.id, (err, post) => {
            next(vm => vm.setData(err, post));
        })
    },
    // - 路由改变前, 组件就已经渲染完了, 逻辑稍稍不同
    beforeRouteUpdate(to, from. next) {
        this.post = null;
        getPost(to.params.id, (err, post) => {
            this.setData(err, post);
            next();
        })
    },
    methods: {
        setData(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.

4.5 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: 这个功能只在支持 `history.pushState` 的浏览器中可用.

  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 `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, offset? : { x: number, y: number }}` (offset 只在 2.6.0+ 支持)

如果返回一个 falsy (译者注: falsy 不是 `false`, 参考这里) 的值, 或者是一个空对象, 那么不会发生滚动.

- falsy 值 (虚值) 是在 Boolean 上下文中认定为 false 的值

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
        }
    }
}
```

我们还可以利用 路由元信息 更细颗粒度地控制滚动.查看完整例子请 移步这里.

4.5.1 异步滚动

- > 2.8.0 新增

- 你也可以返回一个 Promise 来得出预期的位置描述:

```
scrollBehavior (to, from, savedPosition) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({ x: 0, y: 0 })
        }, 500)
    })
}
```

将其挂载到从页面级别的过渡组件的事件上, 令其滚动行为和页面过渡一起良好运行是可能的. 但是考虑到用例的多样性和复杂性, 我们仅提供这个原始的接口, 以支持不同用户场景的具体实现.

4.6 路由懒加载

- 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 **代码分割功能**, 轻松实现路由组件的懒加载.

  首先, 可以将异步组件定义为返回一个 Promise 的工厂函数(该函数返回的 Promise 应该 resolve 组件本身):

  ```
  const Foo = () => Promise.resolve({ /* 组件定义对象 */ })
  ```

  第二, 在 Webpack 2 中, 我们可以使用 **动态 import** 语法来定义代码分块点 (split point):

  ```
  import('./Foo.vue') // 返回 Promise
  ```

  注意

  如果您使用的是 Babel, 你将需要添加 **`syntax-dynamic-import`** 插件, 才能使 Babel 可以正确地解析语法.

  结合这两者, 这就是如何定义一个能够被 Webpack 自动代码分割的异步组件.

  ```
  const Foo = () => import('./Foo.vue')
  ```

  在路由配置中什么都不需要改变, 只需要像往常一样使用 `Foo`:

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

4.6.1 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块(chunk) 中. 只需要使用 **命名 chunk**, 一个特殊的注释语法来提供 chunk name (需要 Webpack > 2.4).

  ```
  const Foo = () => import(/* webpackChunkName: "group-foo" */ './Foo.vue')
  const Bar = () => import(/* webpackChunkName: "group-foo" */ './Bar.vue')
  const Baz = () => import(/* webpackChunkName: "group-foo" */ './Baz.vue')
  ```

Webpack 会将任何一个异步模块与相同的块名称组合到相同的异步块中.