Vue.js 高级开发

组件自定义，项目打包，Vue 单文件组件，路由管理与单元测试

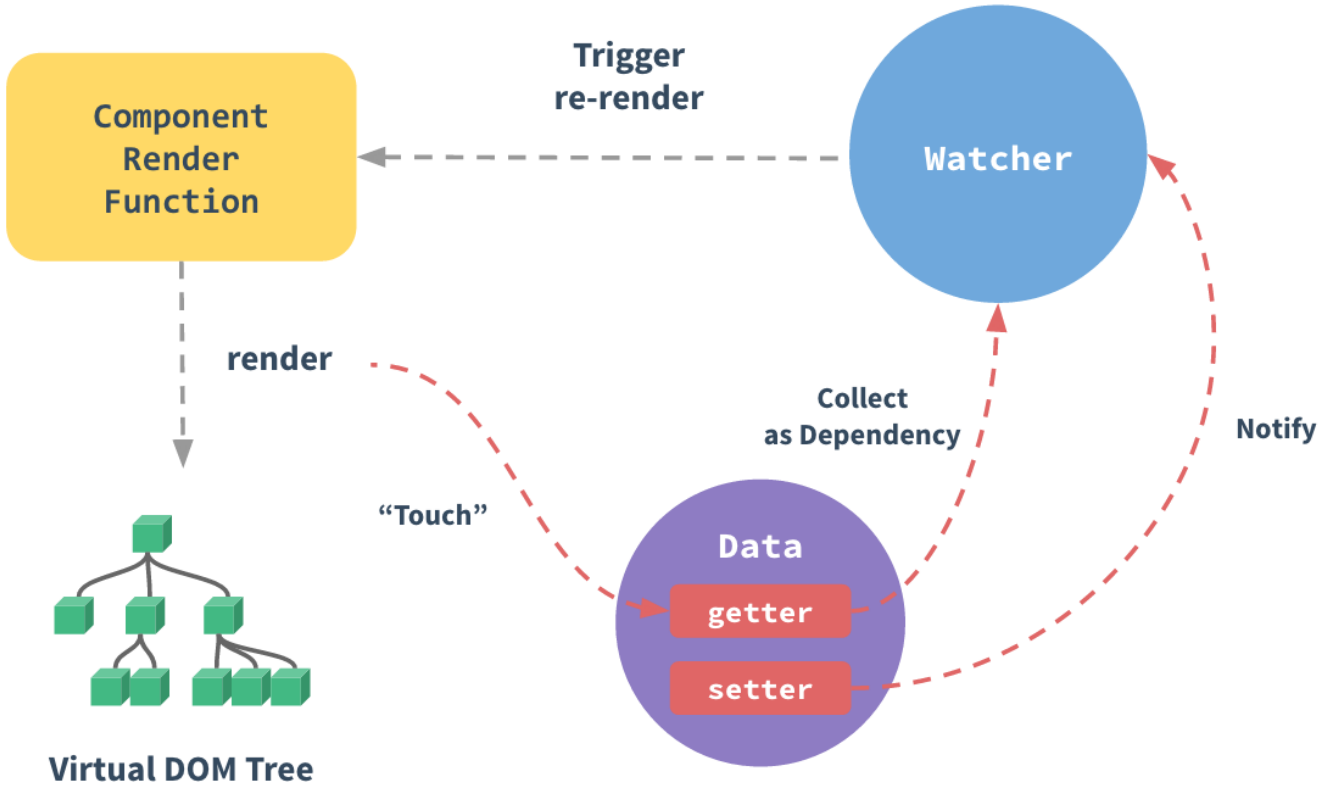
Content

- 异步队列更新
- 自定义指令 Directive
- 混合 Mixin，全局混合
- 插件 Plugins，自定义插件
- 单文件组件和项目
- CSS 抽取
- 路由 Router
- vue-router
- router-link
- router-view
- 路由参数-动态路由匹配

Content

- 响应路由参数的变化
- 嵌套路由（子路由）
- 编程式的导航
- 命名路由，命名视图
- 重定向，别名
- 导航钩子（路由守卫）：全局钩子，单个路由独享的钩子，组件级的钩子
- 过渡效果
- 数据获取：导航完成之后获取，导航完成之前获取
- 滚动行为
- route object 路由信息对象
- Vue 单元测试及覆盖率报告生成

响应式原理


Component
Render
Function
Trigger
re-render
Watcher
render
Collect
as Dependency
Notify
"Touch"
Data
getter
setter
Virtual DOM Tree

异步队列更新

- Vue **异步**执行 DOM 更新，组件会在事件循环队列清空时的下一个“tick”更新。
- 在数据变化之后立即使用 **Vue.nextTick(callback)** 。这样回调函数在 DOM 更新完成后就会调用。

异步队列更新

```
<div id="example">{{message}}</div>

<script type="text/javascript">
    var vm = new Vue({
        el: '#example',
        data: {
            message: '123'
        }
    })
    vm.message = 'new message' // 更改数据
    console.log(vm.$el.textContent === 'new message') // false
    Vue.nextTick(function() {
        console.log(vm.$el.textContent === 'new message') // true
    })
</script>
```

自定义指令 Directive

- 自定义一个 focus 指令

```
// 注册一个全局自定义指令 v-focus
Vue.directive('focus', {
    // 当绑定元素插入到 DOM 中。
    inserted: function(el) {
        // 聚焦元素
        el.focus()
    }
})
```

```
// 也可以在组件中注册局部指令，组件中接受一个 directives 的选项：
directives: {
  focus: {
    // 指令的定义---
  }
}
```

```
<input type="text" name="email" v-focus />
```

混合 Mixin

- 混合是一种灵活的分布式复用 Vue 组件的方式。混合对象可以包含任意组件选项。以组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项。

```
var mixin = {
    data: {
        color: 'red'
    }
}

var vm = new Vue({
    el: "#demo",
    mixins: [mixin]
});
```

全局混合 Mixin

- 影响到 **所有** 之后创建的 Vue 实例

```
Vue.mixin({
     data: function(){
         return {
                 color: 'red'
         };
     }
})

var vm = new Vue({
    el: "#demo"
});
```

插件 Plugins

- 插件可以提供全局功能，除了全局混合还可以使用 Vue 插件。
- 定义插件
  - Vue.js 的插件应当有一个公开方法 **install** 。
- 使用插件
  - 通过全局方法 Vue.use() 使用插件
  - Vue.use 会自动阻止注册相同插件多次，届时只会注册一次该插件。

```
Vue.use(MyPlugin);
```

定义插件

```
var MyPlugin={};
MyPlugin.install = function (Vue, options) {
  // 1. 添加全局方法或属性
  Vue.myGlobalMethod = function () { // 逻辑...  }
  // 2. 添加全局资源
  Vue.directive('my-directive', {
    bind:function (el, binding, vnode, oldVnode) {
      // 逻辑...
    }
  })
  // 3. 注入组件的混合信息
  Vue.mixin({
    created: function () { // 逻辑...  },
    data: function(){
        return { color:'red' }
    }
  })
  // 4. 添加实例方法
  Vue.prototype.$myMethod = function (options) {
    // 逻辑...
  }
}
```

单文件组件

- 在很多Vue项目中，使用 Vue.component 定义全局组件，接着用 new Vue({ el: '#container '}) 在每个页面内指定一个容器元素。
  - **全局定义**(Global definitions) 强制要求每个 component 中的命名不得重复
  - **字符串模板**(String templates) 缺乏语法高亮，在多行HTML 时需要 \
  - **没有 CSS** (No CSS support) CSS 无法组件化明显被遗漏
  - **没有构建步骤(No build step)** 限制只能使用 HTML 和 ES5 JavaScript, 而不能使用预处理器，如 Babel
- 使用.vue 的 **single-file components(单文件组件)** 可以解决以上问题。

Hello.vue

```
<template>
  <p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style scoped>
p {
  font-size: 2em;
  text-align: center;
}
</style>
```

单文件组件优点

- 完整语法高亮
- JS 原生模块
  - export，import
- 组件化的 CSS

```
<style>
/* global styles */
</style>

<style scoped>
/* local styles */
</style>
```

单文件组件项目

- 使用 **webpack-simple** 创建 Vue 单组件开发项目
- 安装，项目初始化

npm i -g npm

npm install -g **vue-cli**

vue init **webpack-simple** my-project

cd my-project

npm install

单文件组件项目

- 热加载运行（端口 8080）

npm run dev

- 产品最小化运行

npm run build

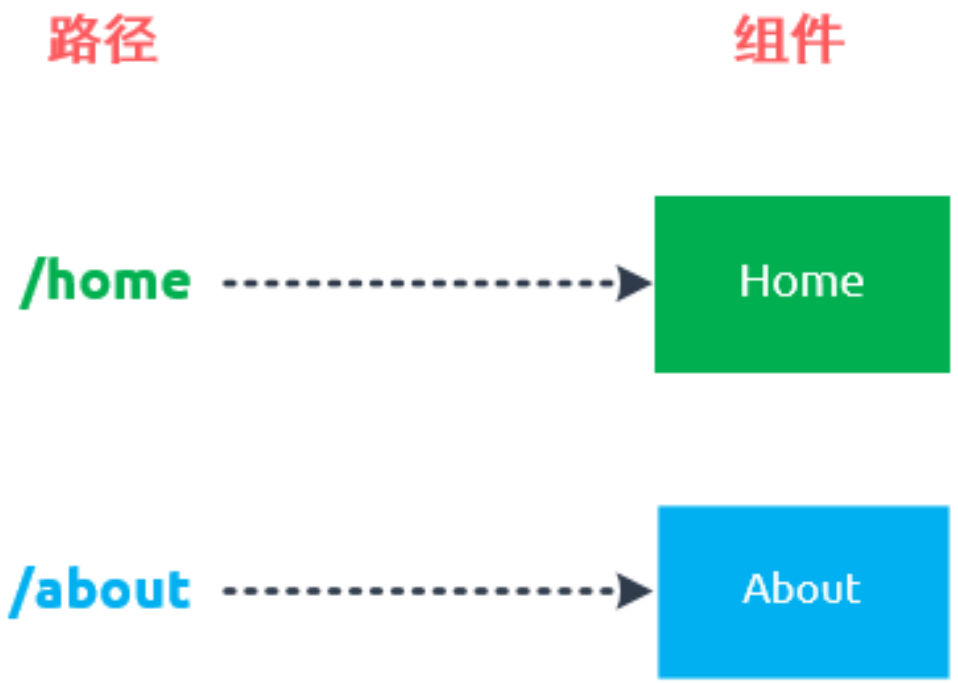
CSS 抽取

- Webpack CSS Code Extracting plugin
- 安装
  - npm install --save-dev **extract-text-webpack-plugin**

webpack.config.js

```
// webpack.config.js
var ExtractTextPlugin = require("extract-text-webpack-plugin")

module.exports = {
  // other options...
  module: {
    rules: [
      {
        test: /\.vue$/,
        loader: 'vue-loader',
        options: {
          loaders: {
            css: ExtractTextPlugin.extract({
              use: 'css-loader',
              fallback: 'vue-style-loader' // <= this is a dep of
            })
          }
        }
      }
    ]
  },
  plugins: [
    new ExtractTextPlugin("style.css")
  ]
}
```

路由 Router



简单路由 Router

- 简单路由

```
const NotFound = { template: '<p>Page not found</p>' }
const Home = { template: '<p>home page</p>' }
const About = { template: '<p>about page</p>' }

const routes = {
  '/': Home,
  '/about': About
}

new Vue({
  el: '#app',
  data: { currentRoute: window.location.pathname },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
})
```

simple-router-slot-webpack

- 基于基础路由结构项目开发

https://github.com/chrisvfritz/vue-2.0-simple-routing-example

- Build Setup
  - **install dependencies**

  npm install
  - **serve with hot reload at localhost:8080**

  npm run dev
  - **build for production with minification**

  npm run build

simple-router-slot-webpack

- 开发指导

1. 在 `pages` 目录下创建 `vue` 单文件组件

   可根据需要创建自己的 layouts

   `MyPage.vue`

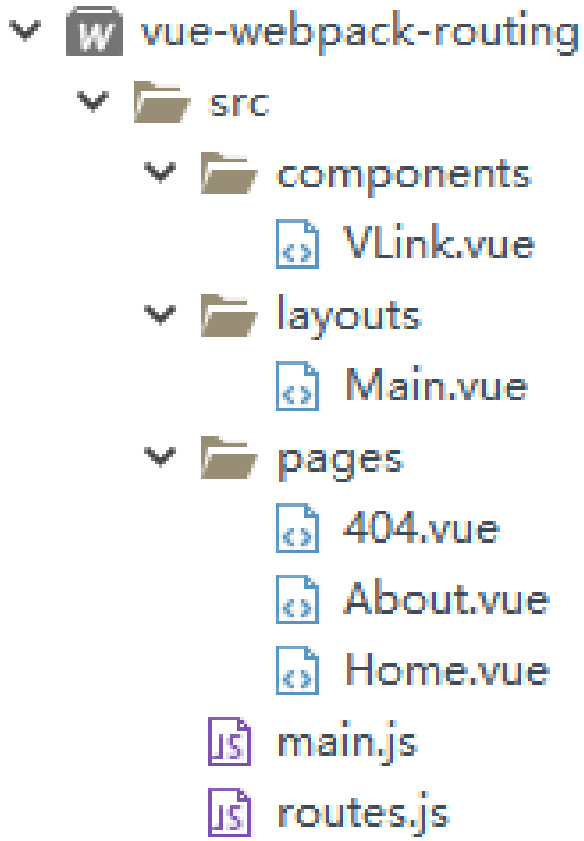
2. 在 `routes.js` 中加入新组件的路由

   `'/page': 'MyPage'`

3. 在 `layouts/Main.vue` 的菜单链接中加入链接

   `<v-link href="/page">MyPage</v-link>`

4. OK



vue-router

- **vue-router** 是官方提供的路由库
- 安装vue-router
- **直接下载**
  - https://unpkg.com/vue-router/dist/vue-router.js
- **NPM**
  - import VueRouter from 'vue-router'
  - Vue.use(VueRouter)

vue-router 使用步骤

1. 将组件(components)映射到路由(routes)
2. 使用 <router-link to="/xxx"> XXX </router-line> 导航
3. 使用 <router-view></router-view> 显示路由渲染内容

HTML

```
<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>
```

JavaScript

```
// 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
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. 创建 router 实例，然后传 `routes` 配置及其他参数
const router = new VueRouter({
  routes: routes
})

// 4. 创建和挂载根实例。让整个应用都有路由功能
const app = new Vue({
  router
}).$mount('#app')
```

router-link

- 在具有路由功能的应用中（点击）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>
```

router-link

- 带历史记录后退的路由跳转

```
<!-- 字符串 -->
<router-link to="home">Home</router-link>

<!-- 绑定属性 -->
<router-link :to="'home'">Home</router-link>

<!-- 同上 -->
<router-link :to="{ path: 'home' }">Home</router-link>

<!-- 带参数的命名的路由 -->
<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>

<!-- 带查询参数，下面的结果为 /register?plan=private -->
 <router-link :to="{ path: 'register', query: { plan: 'private' }}">Register</router-link>
```

router-link

- **replace** 不带历史记录的路由跳转

```
<router-link :to="{ path: '/abc'}" replace></router-link>
```

- **tag** 将路由导航渲染为指定标签

```
<router-link to="/foo" tag="li">foo</router-link>
<!-- 渲染结果 -->
<li>foo</li>
```

router-link

- **active-class** 链接激活时使用的 CSS 类名
  - 当 <router-link> 对应的路由匹配成功，将自动设置 class 属性默认值 "**router-link-active**"
  - 默认值可以通过路由的构造选项 **linkActiveClass** 来全局配置。
- **exact** 路由绝对匹配，不为匹配的父路由设置激活类样式

```
<!-- 这个链接只会在地址为 / 的时候被激活 -->
<router-link to="/" exact>
```

将"激活时的CSS类名"应用在外层元素

- 可以用 <router-link> 渲染外层元素，包裹着内层的原生 <a> 标签：
  - <a> 将作为真实的链接（它会获得正确的 href 的），而 "激活时的CSS类名" 则设置到外层的

```
<router-link tag="li" to="/foo">
  <a>/foo</a>
</router-link>
```

router-view

- 路由出口，路由匹配到的视图组件将渲染在这里

<router-view></router-view>

- 子路由功能：

<router-view> 渲染的组件还可以内嵌自己的 <router-view>，根据嵌套路径，渲染嵌套组件。

vue-router-webpack 开发指导

1. 在 `pages` 目录下创建 `vue` 单文件组件

   `MyPage.vue`

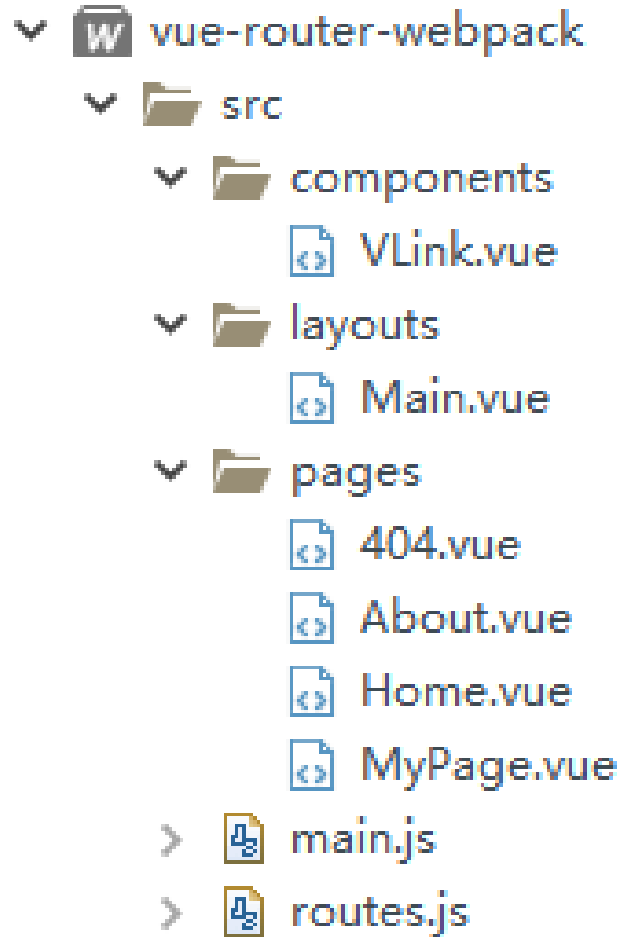
2. 在 `routes.js` 中导入并配置新的路由组件

```
import MyPage from './pages/MyPage.vue'

...

{ path: '/mypage', component: MyPage },
```

3. 在 `VLink.js` 中加入 `<router-link>` 路由连接

```
<router-link to="/mypage" exact>MyPage</router-link>
```

4. OK



路由参数-动态路由匹配

- 使用『动态路径参数』（dynamic segment）来把某种模式匹配到的所有路由，全都映射到同个组件。
- **this.$route.params** 可获得路由参数信息

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
const router = new VueRouter({
  routes: [
    // 动态路径参数 以冒号开头
    { path: '/user/:id', component: User }
  ]
})
```

路由参数-动态路由匹配

- 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

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 }

- 匹配优先级
  - **按定义顺序优先**

响应路由参数的变化

- 使用路由参数时，从 **/user/1001** 导航到 **/user/1002**，因为两个路由都渲染同个组件，原来的组件实例会被复用，不会消耗再创建。
- 因此，组件的生命周期钩子不会再被调用。

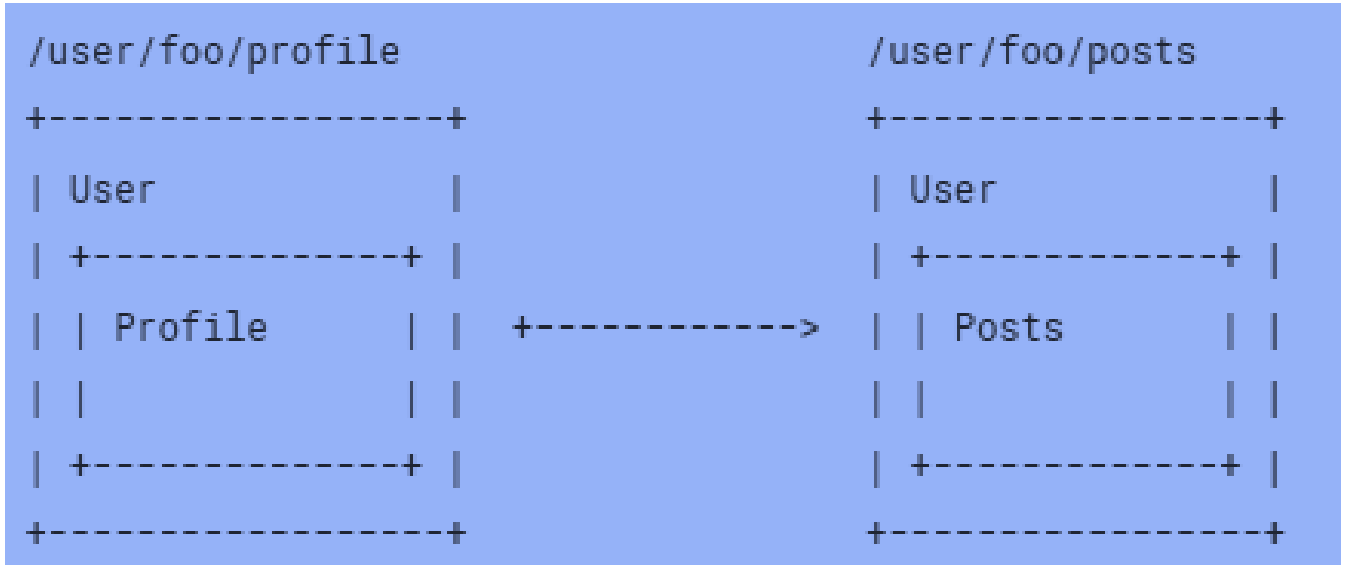
响应路由参数的变化

- 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': function(to, from) {
      // 对路由变化作出响应...
    }
  }
}
```

嵌套路由（子路由）

- 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：



嵌套路由（子路由）

- 组件内部可以包含自己的 router-view
- 子路由功能：

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

嵌套路由（子路由）

- 路由配置

```
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 },
        // 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 }
      ]
    }
  ]
})
```

编程式的导航

- router.push(location)

```
// 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)
  - 替换，不会向 history 添加新记录

编程式的导航

- router.go(n)
  - 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)

命名路由

- **通过一个名称来标识一个路由**，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
    }
  ]
})
```

命名路由

- 跳转到命名路由

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

<!-- JavaScript -->
router.push({ name: 'user', params: { userId: 123 }})
```

命名视图

- 有时候需要同时（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，这个时候可以使用命名视图。
- 可以在界面中拥有**多个单独命名的 router-view**，而不是只有一个单独的出口。如果 router-view 没有设置名字，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>
```

命名视图

- 当有多个单独命名的 router-view 时，**每个视图都要对应一个组件渲染**，因此进入一个路由时，需要使用 **components** 为每个视图配置对应的组件。

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

重定向

- 当用户访问 /a时，URL 将会被替换成 /b，
  - 可以使用字符串，对象，函数返回值作为重定向目标

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

别名

- 可以使用别名访问路由，地址不会变为真实路由

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

导航钩子

- 导航钩子（路由守卫）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

- **全局钩子**
- **单个路由独享的钩子**
- **组件级的钩子**

全局钩子

- 前置钩子：

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

router.beforeEach((to, from, next) => {
  // 前置钩子…
})
```

- 后置钩子：

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

beforeEach

- 钩子是异步解析执行，导航在所有钩子 resolve 完前一直处于**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。**
    - **next():** 进行管道中的下一个钩子。
    - **next(false)**: 中断当前的导航。
    - **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。

单个路由独享的钩子

- 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) => {
        // ...
      }
    }
  ]
})
```

组件级的钩子

- 可以在路由组件内直接定义以下路由导航钩子：
  - beforeRouteEnter
  - beforeRouteUpdate (2.2 新增)
  - beforeRouteLeave

组件级的钩子

```
const Foo = {
  template: `...`,
  beforeRouteEnter: function(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
  },
  beforeRouteUpdate: function(to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用
    // 对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，
    // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。可以访问组件实例 `this`
  },
  beforeRouteLeave: function(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`
  }
}
```

过渡效果

- <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：
- **全局过渡效果**

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

过渡效果

- 单个路由的过渡

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

```
.slide-enter-active, . slide-leave-active {
    transition: width .5s
}
. slide-enter, . slide-leave-active {
  width: 0
}

.fade-enter-active, .fade-leave-active {
    transition: opacity .5s
}
.fade-enter, .fade-leave-active {
  opacity: 0
}
```

动态过渡效果

- 可以动态设置过渡效果

```
<!-- 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':function(to, from) {
    const toDepth = to.path.split('/').length
    const fromDepth = from.path.split('/').length
    this.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-left'
  }
}
```

数据获取

■ 进入某个路由后，需要从服务器获取数据。可以通过两种方式实现：

- **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。

导航完成后获取数据

- 立即导航和渲染组件，然后在组件的 created 钩子中获取数据。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。

导航完成后获取数据

- HTML

```
<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>
```

导航完成后获取数据

- JavaScript

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

在导航完成前获取数据

- 在导航转入新的路由前获取数据。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。
- 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。

route object 路由信息对象

- 一个 **route object（路由信息对象）** 表示当前激活的路由的状态信息，包含了当前 URL 解析得到的信息，还有 URL 匹配到的 **route records（路由记录）**。
- route object 出现在多个地方:
  - 组件内的 **this.$route**
  - 导航钩子的 to 和 from参数

route object 路由信息对象

- **$route.path**：对应当前路由的路径，总是解析为绝对路径，如 "/foo/bar"。
- **$route.params**：一个 key/value 对象，包含了 动态片段 和 全匹配片段，如果没有路由参数，就是一个空对象。
- **$route.query**：一个 key/value 对象，表示 URL 查询参数。例如，对于路径 /foo?user=1，则有 $route.query.user == 1，如果没有查询参数，则是个空对象。
- **$route.hash**：当前路由的 hash 值 (带 #) ，如果没有 hash 值，则为空字符串。
- **$route.fullPath**：完成解析后的 URL，包含查询参数和 hash 的完整路径。
- **$route.name**：当前路由的名称，如果有的话。

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

```
const router = new VueRouter({
  routes: [...],
  scrollBehavior: function(to, from, savedPosition) {
    // return 期望滚动到哪个的位置
     if (savedPosition) {
          return savedPosition; // 仅当浏览器的 前进/后退 按钮触发时才可用
     } else {
          return { x: 0, y: 0 }
     }
  }
})
```

单元测试

- 单元测试属于白盒测试，主要是对功能点的测试
- 测试要素

1. 环境准备
2. 执行测试
3. 断言结果
4. 清理环境

单元测试

- Karma 环境搭建

npm install -g **karma-cli**

npm install **karma** --save-dev

npm install **jasmine-core karma-phantomjs-launcher** --save-dev

单元测试

- 项目初始化

karma init

- testing frameword: **jasmine**
- Require.js: **no**
- browsers automatically: **PhantomJS**
- test files: **test/unit/**/*.spec.js**
- test on change: **no**

单元测试

- 测试代码

```
import Vue from 'vue'
import Home from '../../src/pages/Home.vue'

describe('test app', () => {
   it('测试 msg 是否是 Welcome', () => {
        const vm = new Vue(Home).$mount()
         expect(vm.msg).toEqual('Welcome')
   });

   it('测试 msg 是否是 welcome', () => {
        const vm = new Vue(Home).$mount()
         expect(vm.msg).toEqual('welcome')
   });
});
```

单元测试

- 运行测试

- karma start
- karma run

测试覆盖率报告

- 安装

npm install **karma-coverage isparta isparta-loader** --save-dev

Awesome vue

- Vue 相关组件
  - https://github.com/vuejs/awesome-vue

End

Thanks!

任务

练习

任务一

- 使用基于 webpack 管理的 Vue 单文件组件项目开发 Vue Todo list 应用
- 默认

任务一

- 输入内容，**回车添加新 TODO** 到列表
- todo 选项有三个状态
  - **All** 显示所有
  - **Active** 显示有效的
  - **Completed** 显示已完成的

任务一

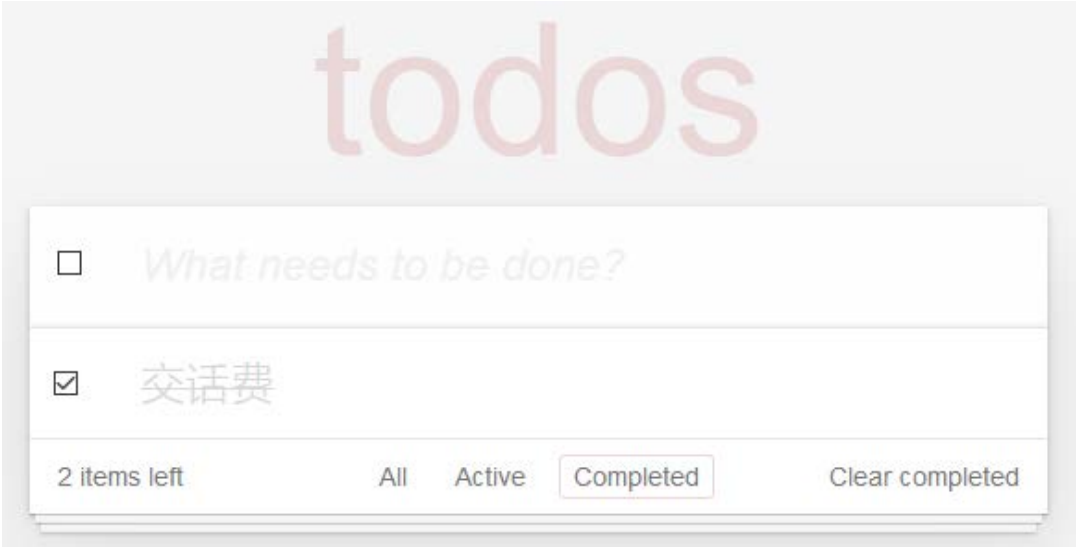
- 点击 todo 前的复选框，标记为**已完成**
  - 已完成的选项外观显示为灰色，带删除线

任务一

- **点击输入框前的复选框**，实现全选，标记为**全部完成**，或**全部取消完成**

任务一

- 查看 Active, Completed 的 todos
  - **在 Completed 中点击复选框，取消完成，重新变为 Active**



任务一

- 鼠标移上，显示删除按钮，点击永久删除 todo 任务

任务一

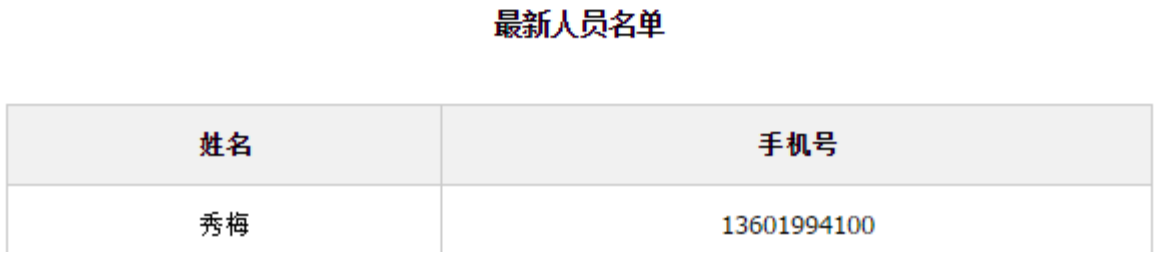
- 在 todo 任务上双击，进入 todo 编辑，光标离开输入完成

任务二

- 制作一个报名登记音乐
- 登录

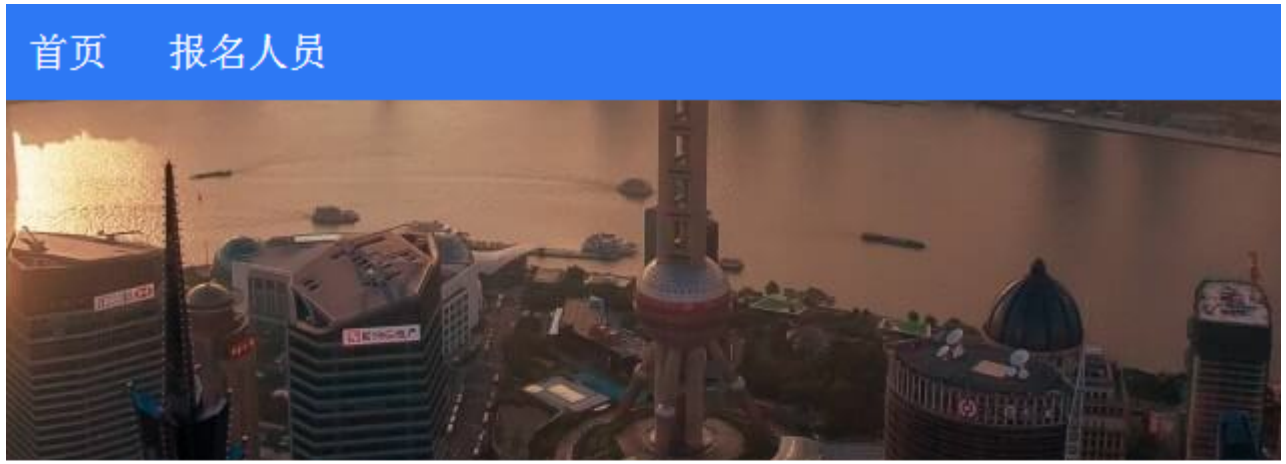
任务二

- 登陆后，进入在线报名



任务二

- 填写完成，显示报名列表
- 实现客户端静态分页



报名人员

姓名	姓别	手机号	操作
kerry	女	18476876654	详情 | 删除
进右五	女	1326776654	详情 | 删除
地方	女	13522367898	详情 | 删除
merry	男	13657654323	详情 | 删除
张三	男	15876890987	详情 | 删除

上一页 1 2 3 4 下一页