

第一章 Vue

第一节 Vue学习准备工作

开发工具VSCode:

Vue 的开发工具用的是 VSCode(Visual Studio Code)，这款开发工具是微软官方出品，开源，免费，并且功能相当强大，使用者很多，插件相当丰富，是 Vue 开发的不二之选。以下对 VSCode 的使用做一个简单了解：

下载地址：

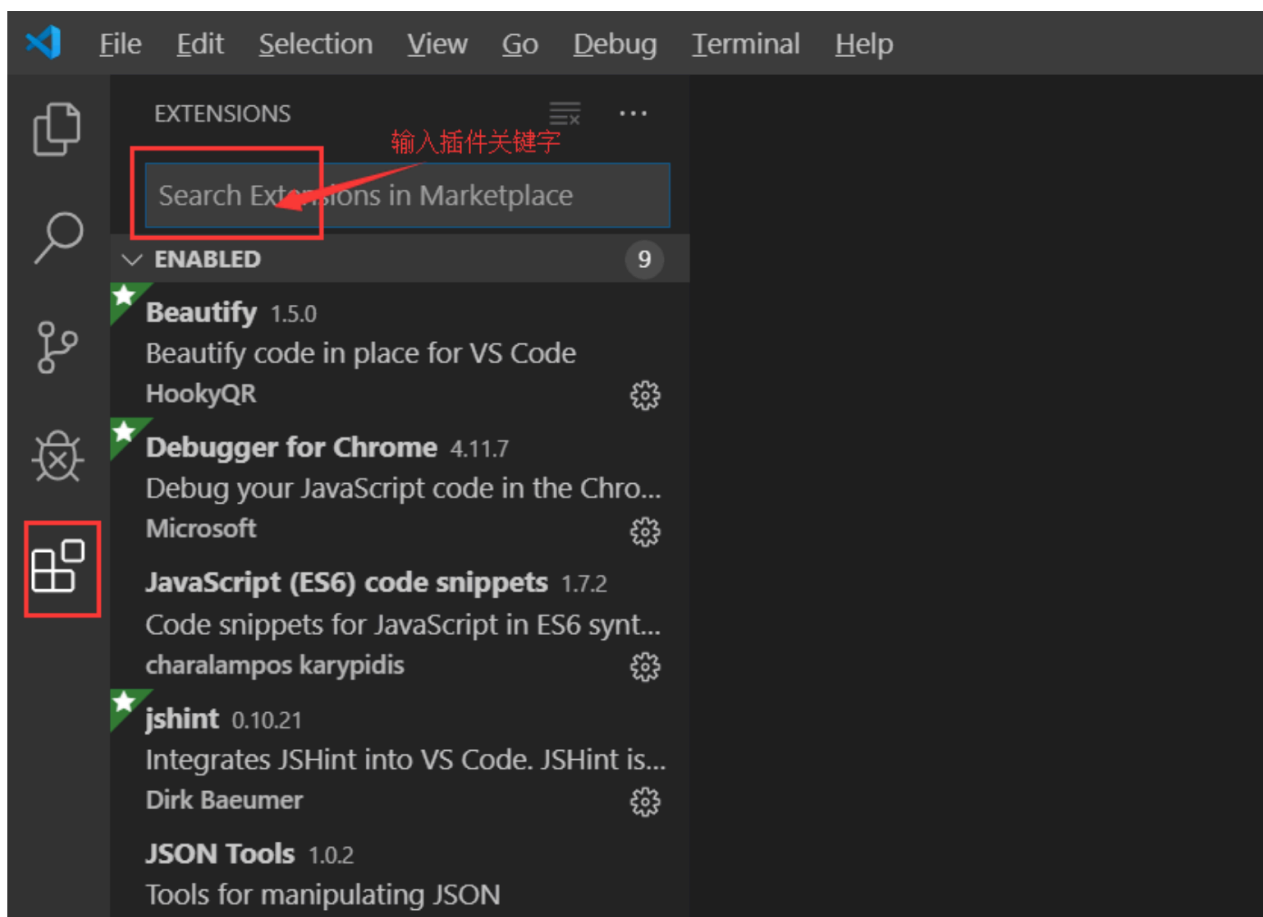
<https://code.visualstudio.com/>。

概念讲解：

VSCode 中分 Workspace 和 Folder，Workspace 相当于是一个项目的集合，可以添加许多的 Folder，在 Workspace 中可以做好一些配置，那里面所有的 Folder 都是按照这个配置来的，就不需要我们每次写个项目都配置一下了。而 Folder 就是我们的每个项目的文件夹。

插件安装：

后面开发 Vue 项目，使用 .vue 的单文件开发，就需要一些插件来帮我们识别 .vue 文件。插件安装在 Extension 中，点开即可看到一个搜索按钮，可以输入关键字搜索自己想要的插件。



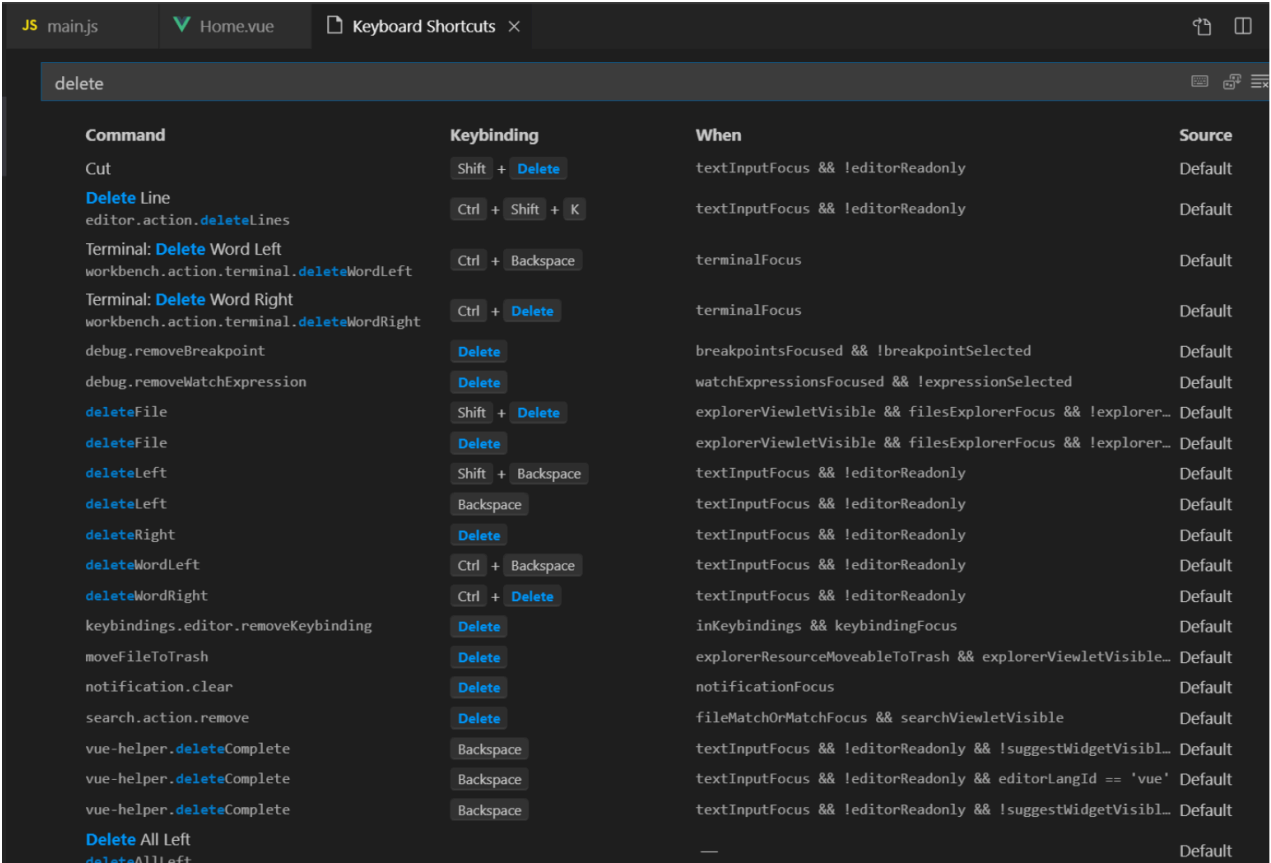
这里我们开发 `vue` 推荐的几个插件：

1. `jshint`： `js` 代码规范检查。
2. `Beautify`： 一键美化代码的插件。
3. `Vetur`： `.vue` 文件识别插件。
4. `Javascript(ES6) code snippets`： `ES6` 语法提示。
5. `Auto Rename Tag`： 自动重命名标签。标签都是成对出现的，开始标签修改了，结束标签也会跟着修改。
6. `Auto Close Tag`： 自动闭合标签。针对一些非标准的标签，这个插件还是很有用的。
7. `vue helper`： 一些 `vue` 代码的快捷代码。
8. `vscode-icons`： 可选。提供了很多类型的文件夹 `icon`，不同类型的文件夹使用不同的 `icon`，会让文件查找更直观。
9. `open in browser`： 可选。右键可以选择在默认浏览器中打开当前网页。

更多插件大家可以自行搜索和安装。

快捷键：

可以按 `ctrl+k` 以及 `ctrl+s` 来打开快捷键窗口，在这个里面可以看到 `vscode` 的所有快捷键。然后在这个里面可以输入关键字来查看对应的快捷键。比如搜索 `delete` 可以看到所有和删除相关的快捷键：



The screenshot shows the VS Code Keyboard Shortcuts editor with the search bar set to 'delete'. The table lists various commands and their associated keybindings and conditions.

Command	Keybinding	When	Source
Cut	Shift + Delete	textInputFocus && !editorReadOnly	Default
Delete Line editor.action.deleteLines	Ctrl + Shift + K	textInputFocus && !editorReadOnly	Default
Terminal: Delete Word Left workbench.action.terminal.deleteWordLeft	Ctrl + Backspace	terminalFocus	Default
Terminal: Delete Word Right workbench.action.terminal.deleteWordRight	Ctrl + Delete	terminalFocus	Default
debug.removeBreakpoint	Delete	breakpointsFocused && !breakpointSelected	Default
debug.removeWatchExpression	Delete	watchExpressionsFocused && !expressionSelected	Default
deleteFile	Shift + Delete	explorerViewletVisible && filesExplorerFocus && !explorer_	Default
deleteFile	Delete	explorerViewletVisible && filesExplorerFocus && !explorer_	Default
deleteLeft	Shift + Backspace	textInputFocus && !editorReadOnly	Default
deleteLeft	Backspace	textInputFocus && !editorReadOnly	Default
deleteRight	Delete	textInputFocus && !editorReadOnly	Default
deleteWordLeft	Ctrl + Backspace	textInputFocus && !editorReadOnly	Default
deleteWordRight	Ctrl + Delete	textInputFocus && !editorReadOnly	Default
keybindings.editor.removeKeybinding	Delete	inKeybindings && keybindingFocus	Default
moveFileToTrash	Delete	explorerResourceMoveableToTrash && explorerViewletVisible_	Default
notification.clear	Delete	notificationFocus	Default
search.action.remove	Delete	fileMatchOrMatchFocus && searchViewletVisible	Default
vue-helper.deleteComplete	Backspace	textInputFocus && !editorReadOnly && !suggestWidgetVisibl_	Default
vue-helper.deleteComplete	Backspace	textInputFocus && !editorReadOnly && editorLangId == 'vue'	Default
vue-helper.deleteComplete	Backspace	textInputFocus && !editorReadOnly && !suggestWidgetVisibl_	Default
Delete All Left deleteAllLeft	—	—	Default

第二节 VSCode快捷键表

快捷键有五种组合方式(科普)

1. `Ctrl + Shift + ?`：这种常规组合按钮
2. `Ctrl + c Ctrl + v`：同时依赖一个按键的组合
3. `Shift + v c`：先组合后单键的输入
4. `Ctrl + Click`： 键盘 + 鼠标点击
5. `Ctrl + DragMouse`： 键盘 + 鼠标拖动 macos下大多键位等同，Ctrl换成Command

通用快捷键

快捷键	作用
Ctrl+Shift+P,F1	展示全局命令面板
Ctrl+P	快速打开最近打开的文件
Ctrl+Shift+N	打开新的编辑器窗口
Ctrl+Shift+W	关闭编辑器

基础编辑

快捷键	作用	
Ctrl + X	剪切	
Ctrl + C	复制	
Alt + up/down	移动行上下	
Shift + Alt up/down	在当前行上下复制当前行	
Ctrl + Shift + K	删除行	
Ctrl + Enter	在当前行下插入新的一行	
Ctrl + Shift + Enter	在当前行上插入新的一行	
Ctrl + Shift + \		匹配花括号的闭合处，跳转
Ctrl +] / [行缩进	
Home	光标跳转到行头	
End	光标跳转到行尾	
Ctrl + Home	跳转到页头	
Ctrl + End	跳转到页尾	
Ctrl + up/down	行视图上下偏移	
Alt + PgUp/PgDown	屏视图上下偏移	
Ctrl + Shift + [折叠区域代码	
Ctrl + Shift +]	展开区域代码	
Ctrl + K Ctrl + [折叠所有子区域代码	
Ctrl + k Ctrl +]	展开所有折叠的子区域代码	
Ctrl + K Ctrl + O	折叠所有区域代码	
Ctrl + K Ctrl + J	展开所有折叠区域代码	
Ctrl + K Ctrl + C	添加行注释	
Ctrl + K Ctrl + U	删除行注释	
Ctrl + /	添加关闭行注释	
Shift + Alt +A	块区域注释	
Alt + Z	添加关闭词汇包含	

导航

快捷键	作用
Ctrl + T	列出所有符号
Ctrl + G	跳转行
Ctrl + P	跳转文件
Ctrl + Shift + O	跳转到符号处
Ctrl + Shift + M	打开问题展示面板
F8	跳转到下一个错误或者警告
Shift + F8	跳转到上一个错误或者警告
Ctrl + Shift + Tab	切换到最近打开的文件
Alt + left / right	向后、向前
Ctrl + M	进入用Tab来移动焦点

查询与替换

快捷键	作用
Ctrl + F	查询
Ctrl + H	替换
F3 / Shift + F3	查询下一个/上一个
Alt + Enter	选中所有出现在查询中的
Ctrl + D	匹配当前选中的词汇或者行，再次选中-可操作
Ctrl + K Ctrl + D	移动当前选择到下个匹配选择的位置(光标选定)

多行光标操作于选择

快捷键	作用
Alt + Click	插入光标-支持多个
Ctrl + Alt + up/down	上下插入光标-支持多个
Ctrl + U	撤销最后一次光标操作
Shift + Alt + I	插入光标到选中范围内所有行结束符
Ctrl + I	选中当前行
Ctrl + Shift + L	选择所有出现在当前选中的行-操作
Ctrl + F2	选择所有出现在当前选中的词汇-操作
Shift + Alt + right	从光标处扩展选中全行
Shift + Alt + left	收缩选择区域
Shift + Alt + (drag mouse)	鼠标拖动区域，同时在多个行结束符插入光标
Ctrl + Shift + Alt + (Arrow Key)	也是插入多行光标的[方向键控制]
Ctrl + Shift + Alt + PgUp/PgDown	也是插入多行光标的[整屏生效]

丰富的语言操作

快捷键	作用
Ctrl + Space	输入建议[智能提示]
Ctrl + Shift + Space	参数提示
Tab	Emmet指令触发/缩进
Shift + Alt + F	格式化代码
Ctrl + K Ctrl + F	格式化选中部分的代码
F12	跳转到定义处
Alt + F12	代码片段显示定义
Ctrl + K F12	在其他窗口打开定义处
Ctrl + .	快速修复部分可以修复的语法错误
Shift + F12	显示所有引用
F2	重命名符号
Ctrl + K Ctrl + X	移除空白字符
Ctrl + K M	更改页面文档格式

编辑器管理

快捷键	作用	
Ctrl + F4, Ctrl + W	关闭编辑器	
Ctrl + k F	关闭当前打开的文件夹	
Ctrl + \		切割编辑窗口
Ctrl + 1/2/3	切换焦点在不同的切割窗口	
Ctrl + K Ctrl <-/->	切换焦点在不同的切割窗口	
Ctrl + Shift + PgUp/PgDown	切换标签页的位置	
Ctrl + K <-/->	切割窗口位置调换	

文件管理

快捷键	作用
Ctrl + N	新建文件
Ctrl + O	打开文件
Ctrl + S	保存文件
Ctrl + Shift + S	另存为
Ctrl + K S	保存所有当前已经打开的文件
Ctrl + F4	关闭当前编辑窗口
Ctrl + K Ctrl + W	关闭所有编辑窗口
Ctrl + Shift + T	撤销最近关闭的一个文件编辑窗口
Ctrl + K Enter	保持开启
Ctrl + Shift + Tab	调出最近打开的文件列表，重复按会切换
Ctrl + Tab	与上面一致，顺序不一致
Ctrl + K P	复制当前打开文件的存放路径
Ctrl + K R	打开当前编辑文件存放位置【文件管理器】
Ctrl + K O	在新的编辑器中打开当前编辑的文件

显示

快捷键	作用
F11	切换全屏模式
Shift + Alt + 1	切换编辑布局 【目前无效】
Ctrl + +/-	放大 / 缩小
Ctrl + B	侧边栏显示隐藏
Ctrl + Shift + E	资源视图和编辑视图的焦点切换
Ctrl + Shift + F	打开全局搜索
Ctrl + Shift + G	打开Git可视管理
Ctrl + Shift + D	打开DeBug面板
Ctrl + Shift + X	打开插件市场面板
Ctrl + Shift + H	在当前文件替换查询替换
Ctrl + Shift + J	开启详细查询
Ctrl + Shift + V	预览Markdown文件 【编译后】
Ctrl + K v	在边栏打开渲染后的视图 【新建】

集成终端

快捷键	作用
Ctrl + `	打开集成终端
Ctrl + Shift + `	创建一个新的终端
Ctrl + Shift + C	复制所选
Ctrl + Shift + V	复制到当前激活的终端
Shift + PgUp / PgDown	页面上下翻页
Ctrl + Home / End	滚动到页面头部或尾部

原文链接: <https://blog.csdn.net/crper/article/details/54099319>

第三节 Vue介绍

Vue介绍

Vue(读音 /vju:/, 类似于 view)是一套用于构建前后端分离的框架。刚开始是由国内优秀选手 尤雨溪 开发出来的, 目前是全球“最”流行的前端框架。使用 vue 开发网页很简单, 并且技术生态环境完善, 社区活跃, 是前后端找工作必备技能!

Vue安装和使用:

vue 的安装大体上分成三种方式, 第一种是通过 script 标签引用的, 第二种是通过 npm(node package manager) 来安装, 第三种是通过 vue-cli 命令行来安装。作为初学者, 建议直接通过第一种方式来安装, 把心思集中在 vue 的学习上, 而不是安装上。

```
# 开发环境
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
# 或者是指定版本号
<script src="https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js"></script>
# 或者是下载代码保存到本地
<script src="lib/vue.js"></script>

# 生产环境, 使用压缩后的文件
<script src="https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.min.js"></script>
```

基本使用:

要使用 vue, 首先需要创建一个 vue 对象, 并且在这个对象中传递 el 参数, el 参数全称是 element, 用来找到一个给 vue 渲染的根元素。并且我们可以传递一个 data 参数, data 中的所有值都可以直接在根元素下使用 {{}} 来使用。示例代码如下:

```
<div id="app">
  <p>{{username}}</p>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      "username": "知了课堂"
    }
  });
</script>
```

其中 data 中的数据, 只能在 vue 的根元素下使用, 在其他地方是不能被 vue 识别和渲染的。比如:

```

<div id="app">
</div>
<!-- 这里渲染不了 -->
<p>{{username}}</p>
<script>
    let vm = new Vue({
        el: "#app",
        data: {
            "username": "知了课堂"
        }
    });
</script>

```

另外也可以在 `vue` 对象中添加 `methods`，这个属性中专门用来存储自己的函数。`methods` 中的函数也可以在模板中使用，并且在 `methods` 中的函数来访问 `data` 中的值，只需要通过 `this.属性名` 就可以访问到了，不需要额外加 `this.data.属性名` 来访问。示例代码如下：

```

<div id="app">
    <p>{{greet()}}</p>
</div>
<script>
    let vm = new Vue({
        el: "#app",
        data: {
            "username": "知了课堂"
        },
        methods: {
            greet: function(){
                return "下午好! " + this.username
            }
        }
    });
</script>

```

第四节 Vue模版语法

文本：

在 `html` 中通过 `{{}}`（双大括号）中可以把 `vue` 对象中的数据插入到网页中。并且只要 `vue` 对象上对应的值发生改变了，那么 `html` 中双大括号中的值也会立马改变。

```

<div id="app">
    <p>{{username}}</p>
    <button v-on:click="change">点击修改</button>
</div>
<script>
    let vm = new Vue({

```

```

    el: "#app",
    data: {
      "username": "知了课堂"
    },
    methods: {
      change: function(){
        this.username = 'China';
      }
    }
  });
</script>

```

当然，如果在 `html` 的 `{{}}` 中，第一次渲染完成后，不想要跟着后期数据的更改而更改，那么可以使用 `v-once` 指令来实现。示例代码如下：

```
<p v-once>{{username}}</p>
```

显示原生的HTML：

有时候我们的 `Vue` 对象中，或者是后台返回给我们一段原生的 `html` 代码，我们需要渲染出来，那么如果直接通过 `{{}}` 渲染，会把这个 `html` 代码当做字符串。这时候我们就可以通过 `v-html` 指令来实现。示例代码如下：

```

<div id="app">
  <div v-html="code"></div>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      "code": "<a href='https://www.baidu.com/'>百度一下，你就知道! </a>"
    }
  });
</script>

```

属性绑定：

如果我们想要在 `html` 元素的属性上绑定我们 `Vue` 对象中的变量，那么需要通过 `v-bind` 来实现。比如以下代码是不行的：

```

<div id="app">
  <img class="{{classname}}">你好，世界</p>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      "classname": "pclass"
    },
  });
</script>

```

需要使用 `v-bind` 才能生效：

```

<div id="app">
  
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      "imgSrc": "https://i.ytimg.com/vi/5HKZ6bU6Zg0/maxresdefault.jpg"
    }
  });
</script>

```

属性绑定Class和Style：

在绑定 `class` 或者 `style` 的时候，可以通过以下方式来实现。

绑定Class：

1. 通过数组的方式来实现：

```

<div id="app">
  <p v-bind:class="[classname1,classname2]">你好，世界</p>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      classname1: "pcolor",
      classname2: "pfont"
    }
  });
</script>

```

2. 通过对象的方式来实现：

```
<div id="app">
  <p v-bind:class="{pcolor:isColor,pfont:isFont}">你好，世界</p>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      isColor: true,
      isFont: true
    }
  });
</script>
```

绑定Style：

1. 用对象的方式。示例代码如下：

```
# 读取对应样式的值
<li :style="{backgroundColor:pBgColor,fontSize:pFontSize}">首页</li>
# 或者是直接读取整个字符串
<li :style="liStyle">首页</li>
```

但是样式如果有横线，则都需要变成驼峰命名的方式。

2. 用数组的方式。示例代码如下：

```
<li :style="[liStyle1,liStyle2]">首页</li>
<script>
  new Vue({
    el: "#app",
    data: {
      liStyle: {
        backgroundColor: "red",
        fontSize: 14
      },
      liStyle2: {
        border: "1px solid blue"
      }
    }
  })
</script>
```

使用 JavaScript 表达式：

在使用了 `v-bind` 的 `html` 属性，或者使用了 `{{}}` 的文本。我们还可以执行一个 `JavaScript` 表达式。示例代码如下：

```
<div id="app">
  <p v-bind:class="color?'pcolor':''">
    {{username.split("").reverse().join("")}}</p>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      username: "zhiliao ketang",
      color: false
    }
  });
</script>
```

注意，只能是 `JavaScript` 表达式，不能是语句，比如 `var a=1;a=2;` 这样的是 `js` 语句，不是表达式了。

条件判断：

在模板中，可以根据条件进行渲染。条件用到的是 `v-if`、`v-else-if` 以及 `v-else` 来组合实现的。示例代码如下：

```
<div id="app">
  <p v-if="weather == 'sun'">今天去公园玩! </p>
  <p v-else-if="weather == 'rain'">今天去看电影! </p>
  <p v-else>今天哪儿也不去! </p>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      weather: 'sun'
    }
  });
</script>
```

有时候我们想要在一个条件中加载多个 `html` 元素，那么我们可以通过 `template` 元素上实现。示例代码如下：

```
<div id="app">
  <template v-if="age<18">
    <p>数学多少分? </p>
```

```

    <p>英语多少分? </p>
  </template>
  <template v-else-if="age>=18 && age<25">
    <p>女朋友找到了吗? </p>
    <p>考研究生了吗? </p>
  </template>
  <template v-else>
    <p>二胎生了吗? </p>
    <p>工资多少? </p>
  </template>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      age: 24
    }
  });
</script>

```

另外，在模板中，Vue 会尽量重用已有的元素，而不是重新渲染，这样可以变得更加高效。如果你允许用户在不同的登录方式之间切换：

```

<div id="app">
  <template v-if="loginType=='username'">
    <label for="username">用户名: </label>
    <input type="text" id="username" name="username" placeholder="用户名">
  </template>
  <template v-else-if="loginType=='email'">
    <label for="email">邮箱: </label>
    <input type="text" id="email" name="email" placeholder="邮箱">
  </template>
</div>
  <button v-on:click="changeLoginType">切换登录类型</button>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      loginType: "username"
    },
    methods: {
      changeLoginType: function(event){
        this.loginType = this.loginType=="username"? "email": "username";
      }
    }
  });

```



```
</script>
```

这个里面会有一个问题，就是如果我在 `username` 的输入框中输入完信息，切换到邮箱中，之前的信息还是保留下来，这样肯定不符合需求的，如果我们想要让 `html` 元素每次切换的时候都重新渲染一遍，可以在需要重新渲染的元素上加上唯一的 `key` 属性，其中 `key` 属性推荐使用整形，字符串类型。示例代码如下：

```
<div id="app">
  <template v-if="loginType=='username'">
    <label for="username">用户名: </label>
    <input type="text" id="username" name="username" placeholder="用户名"
key="username">
  </template>
  <template v-else-if="loginType=='email'">
    <label for="email">邮箱: </label>
    <input type="text" id="email" name="email" placeholder="邮箱"
key="email">
  </template>
  <div>
    <button v-on:click="changeLoginType">切换登录类型</button>
  </div>
</div>
```

注意，`<label>` 元素仍然会被高效地复用，因为它们没有添加 `key` 属性。

v-show和v-if:

`v-if` 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。`v-if` 也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。相比之下，`v-show` 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 `CSS` 进行切换。一般来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 `v-show` 较好；如果在运行时条件很少改变，则使用 `v-if` 较好。

循环:

在模板中可以用 `v-for` 指令来循环数组，对象等。

循环数组:

```
<div id="app">
  <table>
    <thead>
      <tr>
        <th>序号</th>
        <th>标题</th>
        <th>作者</th>
```

```

        </tr>
    </thead>
    <tbody>
        <tr v-for="(book,index) in books">
            <td>{{index}}</td>
            <td>{{book.title}}</td>
            <td>{{book.author}}</td>
        </tr>
    </tbody>
</table>
</div>
<script>
    let vm = new Vue({
        el: "#app",
        data: {
            books: [{
                'title': '三国演义',
                'author': '罗贯中'
            }, {
                'title': '水浒传',
                'author': '施耐庵'
            }, {
                'title': '西游记',
                'author': '吴承恩'
            }, {
                'title': '红楼梦',
                'author': '曹雪芹'
            }
        ]
    });
</script>

```

循环对象：

循环对象跟循环数组是一样的。并且都可以在循环的时候使用接收多个参数。示例代码如下：

```

<div id="app">
    <div v-for="(value,key) in person">
        {{key}}:{{value}}
    </div>
</div>
<script>
    let vm = new Vue({
        el: "#app",
        data: {
            person: {
                "username": "知了",
                "age": 18,
                "homepage": "https://www.baidu.com/"
            }
        }
    });
</script>

```

```
    }  
  }  
});  
</script>
```

保持状态：

循环出来的元素，如果没有使用 `key` 元素来唯一标识，如果后期的数据发生了更改，默认是会重用的，并且元素的顺序不会跟着数据的顺序更改而更改。比如：

```
<div id="app">  
  <div v-for="(book,index) in books">  
    <label for="book">{{book}}</label>  
    <input type="text" v-bind:placeholder="book">  
  </div>  
  <button v-on:click="changeBooks">更换图书</button>  
</div>  
<script>  
  let vm = new Vue({  
    el: "#app",  
    data: {  
      books: [ '三国演义', '水浒传', '红楼梦', '西游记' ]  
    },  
    methods: {  
      changeBooks: function(event){  
        this.books.sort((x,y) => {  
          return 5 - parseInt(Math.random()*10)  
        });  
      }  
    }  
  });  
</script>
```

如果你在某个 `input` 标签中输入了值，然后点击了“更换图书”的按钮，你会发现及时数据更改了，`input` 并不会跟着数据的更改而更改，这时候我们只需要在 `v-for` 的时候加上一个 `key` 属性就可以了。示例代码如下：

```
<div v-for="(book,index) in books" v-bind:key="book">  
  <label for="book">{{book}}</label>  
  <input type="text" v-bind:placeholder="book">  
</div>
```

注意，`key` 只能是整形，或者是字符串类型，不能为数组或者对象。

触发视图更新：

Vue 对一些方法进行了包装和变异，以后数组通过这些方法进行数组更新，会出发视图的更新。这些方法如下：

1. `push()`：添加元素的方法。

```
this.books.push("金瓶梅")
```

```
pop()
```

：删除数组最后一个元素。

```
this.books.pop()
```

```
shift()
```

：删除数组的第一个元素。

```
this.books.shift()
```

```
unshift(item)
```

：在数组的开头位置添加一个元素。

```
this.books.unshift("金瓶梅")
```

```
splice(index,howmany,item1,...,itemX)
```

：向数组中添加或者删除或者替换元素。

```
// 向books第0个位置添加元素
this.books.splice(0,0,"金瓶梅")
// 下标从0开始，删除2个元素
this.books.splice(0,2)
// 下标从0开始，替换2个元素
this.books.splice(0,2,'金瓶梅','骆驼祥子')
```

```
sort(function)
```

：排序。

```
this.books.sort(function(x,y){
  // 取两个随机数排序
  a = Math.random();
  b = Math.random();
  return a-b;
});
```

```
reverse()
```

：将数组元素进行反转。

```
this.books.reverse();
```

还有一些Vue没有包装的方法，比如filter、concat、slice，如果使用这些方法修改了数组，那么只能把修改后的结果重新赋值给原来的数组才能生效。比如：

```
this.books = this.books.filter(function(x){
  return x.length>3?false:true;
})
```

视图更新注意事项：

1. 直接修改数组中的某个值是不会出发视图更新的。比如：

```
this.books[0] = '金瓶梅';
```

这种情况应该改成用splice或者是用Vue.set方法来实现：

```
Vue.set(this.books,0,'金瓶梅');
```

2. 如果动态的给对象添加属性，也不会触发视图更新。只能通过Vue.set来添加。比如：

```
<div id="app">
  <ul>
    <li v-for="(value,name) in person">{{name}}:{{value}}</li>
  </ul>
  <script>
    let vm = new Vue({
      el: "#app",
      data: {
        person: {"username": '知了课堂'}
      },
      methods: {
        changePerson: function(event){
          // 直接修改this.person.age是没有效果的
        }
      }
    })
```

```

        // this.person.age = 18;
        Vue.set(this.person, 'age', 18)
      }
    }
  });
</script>
</div>

```

事件绑定：

事件绑定就是在 HTML 元素中，通过 `v-on` 绑定事件的。事件代码可以直接放到 `v-on` 后面，也可以写成一个函数。示例代码如下：

```

<div id="app">
  <p>{{count}}</p>
  <button v-on:click="count+=1">加</button>
  <button v-on:click="subtract(10)">减10</button>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      count: 0
    },
    methods: {
      subtract: function(value){
        this.count -= value;
      }
    }
  });
</script>

```

传入 `event` 参数：

如果在事件处理函数中，想要获取原生的 DOM 事件，那么在 html 代码中，调用的时候，可以传递一个 `$event` 参数。示例代码如下：

```

<button v-on:click="subtract(10,$event)">减10</button>
...
<script>
...
methods: {
  subtract: function(value,event){
    this.count -= value;
    console.log(event);
  }
}
...
</script>

```

事件修饰符：

有时候事件发生，我们可能需要做一些操作。比如针对这个事件要他的默认行为。那么我们可能通过以下代码来实现：

```

<div id="app">
  <a href="https://www.baidu.com/" v-on:click="gotoWebsite($event)">百度</a>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      count: 0
    },
    methods: {
      gotoWebsite: function(event){
        event.preventDefault();
        window.location = "https://www.360.cn/"
      }
    }
  });
</script>

```

那个阻止默认事件执行的代码，我们可以通过 `click.prevent` 来实现。示例代码如下：

```

<a href="https://www.baidu.com/" v-on:click.prevent="gotoWebsite($event)">百度
</a>

```

另外，常见的修饰符还有以下：

1. `.stop`： `event.stopPropagation`，阻止事件冒泡。
2. `.capture`：事件捕获。
3. `.once`：这个事件只执行一次。
4. `.self`：代表当前这个被点击的元素自身。

5. `.passive`：在页面滚动的时候告诉浏览器不会阻止默认的行为，从而让滚动更加顺畅。

第五节 计算属性和监听器

一般情况下属性都是放到 `data` 中的，但是有些属性可能是需要经过一些逻辑计算后才能得出来，那么我们可以把这类属性变成计算属性。比如以下：

```
<div id="app">
  <label for="length">长: </label>
  <input type="number" name="length" v-model:value="length">
  <label for="width">宽: </label>
  <input type="number" name="width" v-model:value="width">
  <label for="area">面积: </label>
  <input type="number" name="area" v-bind:value="area" readonly>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      length: 0,
      width: 0,
    },
    computed: {
      area: function(){
        return this.length*this.width;
      }
    }
  });
</script>
```

可能有的小伙伴会觉得这个计算属性跟我们之前学过的函数好像有点重复。实际上，计算属性更加智能，他是基于它们的响应式依赖进行缓存的。也就是说只要相关依赖（比如以上例子中的 `area`）没有发生改变，那么这个计算属性的函数不会重新执行，而是直接返回之前的值。这个缓存功能让计算属性访问更加高效。

计算属性的 `set`：

计算属性默认只有 `get`，不过在需要时你也可以提供一个 `set`，但是提供了 `set` 就必须提供 `get` 方法。示例代码如下：

```
<div id="app">
  <div>
    <label>省: </label>
    <input type="text" name="province" v-model:value="province">
  </div>
  <div>
    <label>市: </label>
    <input type="text" name="city" v-model:value="city">
  </div>
</div>
```



```
</div>
<div>
  <label>区: </label>
  <input type="text" name="district" v-model:value="district">
</div>
<div>
  <label>详细地址: </label>
  <input type="text" name="address" v-model:value="address">
</div>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      district: "",
      city: "",
      province: ""
    },
    computed: {
      address: {
        get: function(){
          let result = "";
          if(this.province){
            result = this.province + "省";
          }
          if(this.city){
            result += this.city + "市";
          }
          if(this.district){
            result += this.district + "区";
          }
          return result;
        },
        set: function(newValue){
          let result = newValue.split(/省|市|区/)
          if(result && result.length > 0){
            this.province = result[0];
          }
          if (result && result.length > 1){
            this.city = result[1];
          }
          if(result && result.length > 2){
            this.district = result[2];
          }
        }
      }
    }
  });
</script>
```

监听属性：

监听属性可以针对某个属性进行监听，只要这个属性的值发生了变化，那么就会执行相应的函数。示例代码如下：

```
<div id="app">
  <div>
    <label>搜索： </label>
    <input type="text" name="keyword" v-model:value="keyword">
  </div>
  <div>
    <p>结果： </p>
    <p>{{answer}}</p>
  </div>
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      keyword: "",
      answer: ""
    },
    watch: {
      keyword: function(newKeyword,oldKeyword){
        this.answer = '加载中...';
        let that = this;
        setTimeout(function(){
          that.answer = that.keyword;
        },Math.random()*5*1000);
      }
    }
  });
</script>
```

第六节 表单输入绑定

`v-model` 指定可以实现表单值与属性的双向绑定。即表单元素中更改了值会自动的更新属性中的值，属性中的值更新了会自动更新表单中的值。

绑定的属性和事件：

`v-model` 在内部为不同的输入元素使用不同的属性并抛出不同的事件：

1. `text` 和 `textarea` 元素使用 `value` 属性和 `input` 事件。
2. `checkbox` 和 `radio` 使用 `checked` 属性和 `change` 事件。
3. `select` 字段将 `value` 作为 `prop` 并将 `change` 作为事件。

表单元素绑定：

input绑定：

```
<input v-model="message" placeholder="请输入...">
<p>输入的内容是： {{ message }}</p>
```

textarea绑定：

```
<span>输入的内容是： </span>
<p style="white-space: pre-line;">{{ message }}</p>
<br>
<textarea v-model="message" placeholder="请输入多行内容..."></textarea>
```

checkbox绑定：

checkbox 是复选框。

```
<div id='example-3'>
  <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
  <label for="jack">Jack</label>
  <input type="checkbox" id="john" value="John" v-model="checkedNames">
  <label for="john">John</label>
  <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
  <label for="mike">Mike</label>
  <br>
  <span>Checked names: {{ checkedNames }}</span>
</div>
new Vue({
  el: '#example-3',
  data: {
    checkedNames: []
  }
})
```

radio绑定：

radio 是单选框。

```
<div id="example-4">
  <input type="radio" id="one" value="One" v-model="picked">
  <label for="one">One</label>
  <br>
  <input type="radio" id="two" value="Two" v-model="picked">
  <label for="two">Two</label>
  <br>
  <span>Picked: {{ picked }}</span>
```

```

</div>
new Vue({
  el: '#example-4',
  data: {
    picked: ''
  }
})

```

select绑定：

`select` 是可以下拉的。

```

<div id="example-5">
  <select v-model="selected">
    <option disabled value="">请选择</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>Selected: {{ selected }}</span>
</div>
new Vue({
  el: '...',
  data: {
    selected: ''
  }
})

```

修饰符：

`.lazy`：

在默认情况下，`v-model` 在每次 `input` 事件触发后将输入框的值与数据进行同步 (除了上述输入法组合文字时)。你可以添加 `lazy` 修饰符，从而转变为使用 `change` 事件进行同步：

```

<!-- 在“change”时而非“input”时更新 -->
<input v-model.lazy="msg" >

```

`.number`：

如果想自动将用户的输入值转为数值类型，可以给 `v-model` 添加 `number` 修饰符：

```

<input v-model.number="age" type="number">

```

这通常很有用，因为即使在 `type="number"` 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 `parseFloat()` 解析，则会返回原始的值。

.trim:

如果要自动过滤用户输入的首尾空白字符，可以给 v-model 添加 trim 修饰符：

```
<input v-model.trim="msg">
```

第七节 自定义组件

有时候有一组 `html` 结构的代码，并且这个上面可能还绑定了事件。然后这段代码可能有多个地方都被使用到了，如果都是拷贝来拷贝去，很多代码都是重复的，包括事件部分的代码都是重复的。那么这时候我们就可以把这些代码封装成一个组件，以后在使用的时候就跟使用普通的 `html` 元素一样，拿过来用就可以了。

基本使用：

```
<div id="app">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
<script>
  Vue.component('button-counter', {
    data: function(){
      return {
        count: 0
      }
    },
    template: '<button v-on:click="count++">点击了{{ count }}次</button>'
  });
  let vm = new Vue({
    el: "#app",
    data: {}
  });
</script>
```

以上我们创建了一个叫做 `button-counter` 的组件，这个组件实现了能够记录点击了多少次按钮的功能。后期如果我们想要使用，就直接通过 `button-counter` 使用就可以了。然后因为组件是可复用的 `Vue` 实例，所以它们与 `new Vue` 接收相同的选项，例如 `data`、`computed`、`watch`、`methods` 以及生命周期钩子等。仅有的例外是像 `el` 这样根实例特有的选项。另外需要注意的是：组件中的 `data` 必须为一个函数！

给组件添加属性：

像原始的 `html` 元素都有自己的一些属性，而我们自己创建的组件，也可以通过 `prop` 来添加自己的属性。这样别人在使用你创建的组件的时候就可以传递不同的参数了。示例代码如下：

```
<div id="app">
```

```

    <blog-post v-for="blog in blogs" :title="blog.title"></blog-post>
  </div>
  <script>
    Vue.component('blog-post', {
      props: ['title'],
      template: '<h3>{{ title }}</h3>'
    })
    new Vue({
      el: "#app",
      data: {
        blogs: [
          {"title": "钢铁是怎样练成的?", "id": 1},
          {"title": "AI会毁灭人类吗?", "id": 2},
          {"title": "如何学好vue!", "id": 3},
        ]
      }
    });
  </script>

```

单一根元素：

如果自定义的组件中，会出现很多 `html` 元素，那么根元素必须只能有一个，其余的元素必须包含在这个根元素中。比如以下是一个组件中的代码，会报错：

```

<h3>{{ title }}</h3>
<div v-html="content"></div>

```

我们应该改成：

```

<div class="blog-post">
  <h3>{{ title }}</h3>
  <div v-html="content"></div>
</div>

```

子组件事件和传递事件到父组件：

子组件中添加事件跟之前的方式是一样的，然后如果发生某个事件后想要通知父组件，那么可以使用 `this.$emit` 函数来实现。示例代码如下：

```

<div id="app">
  <blog-post v-for="blog in blogs" :post="blog" :key="blog.id" v-on:like-
  changed="outerLikeChanged"></blog-post>
</div>
<script>
  Vue.component('blog-post', {
    props: ['post'],
    template: `

```

```

        <div>
            <h3>{{ post.title }}</h3>
            <input type="checkbox" v-model="post.like" v-
on:change="innerLikeChanged">
        </div>
    },
    methods: {
        innerLikeChanged: function(){
            this.$emit("like-changed",this.post.id);
        }
    }
})
new Vue({
  el: "#app",
  data: {
    blogs: [
      {"title":"钢铁是怎样练成的? ","id":1,"like":false},
      {"title":"AI会毁灭人类吗? ","id":2,"like":false},
      {"title":"如何学好Vue! ","id":3,"like":false},
    ]
  },
  methods: {
    outerLikeChanged: function(post_id){
      this.blogs.forEach(blog => {
        if(blog['id'] == post_id){
          blog.like = !blog.like
          console.log(blog);
        }
      });
    }
  }
});
</script>

```

需要注意的是，因为 `html` 中大小写是不敏感的，所以在定义子组件传给父组件事件名称的时候，不要使用 `myEvent` 这种驼峰命名法，而是使用 `my-event` 这种规则。

自定义组件 `v-model`：

一个组件上的 `v-model` 默认会利用名为 `value` 的 `prop`(属性) 和名为 `input` 的事件，但是像单选框、复选框等类型的输入控件可能会将 `value` 特性用于不同的目的。这时候我们可以在定义组件的时候，通过设置 `model` 选项可以用来实现不同的处理方式：

```

<div id="app">
  <blog-post v-for="blog in blogs" v-model="blog.like" :blog="blog"
:key="blog.id"></blog-post>
</div>
<script>

```

```

Vue.component('blog-post', {
  model: {
    event: 'myevent',
    prop: 'liked'
  },
  props: {
    blog: Object,
    liked: false
  },
  template: `
    <div>
      <h3>{{blog.title}}</h3>
      <input type="checkbox" v-bind:checked="liked" v-
on:click="likeClick">
    </div>
  `,
  methods: {
    likeClick: function(event){
      this.$emit('myevent',event.target.checked);
    }
  }
})
new Vue({
  el: "#app",
  data: {
    blogs: [
      {"title": "钢铁是怎样练成的?", "id": 1, "like": false},
      {"title": "AI会毁灭人类吗?", "id": 2, "like": false},
      {"title": "如何学好vue!", "id": 3, "like": false},
    ]
  },
  created: function(){
    // setInterval(()=>{
    //   this.blogs.forEach(blog => {
    //     console.log(blog.title, blog.like);
    //   });
    // }, 2000);
  }
});
</script>

```

其中的 `props` 定义的两个属性分别是给外面调用组件的时候使用的。 `model` 总定义的 `prop: 'like'` 是告诉后面使用 `v-model` 的时候，要修改哪个属性； `event: 'myevent'` 是告诉 `v-model`，后面触发哪个事件的时候要修改属性。

插槽：

我们定义完一个组件后，可能在使用的时候还需要往这个组件中插入新的元素或者文本。这时候就可以使用插槽来实现。示例代码如下：

```
<div id="app">
  <navigation-link url="/profile/">
    个人中心
  </navigation-link>
</div>
<script>
  Vue.component('navigation-link', {
    props: ['url'],
    template: `
      <a v-bind:href="url" class="nav-link">
        <slot></slot>
      </a>
    `
  })
  new Vue({
    el: "#app"
  });
</script>
```

当组件渲染的时候，`<slot></slot>` 将会被替换为“个人中心”。插槽内可以包含任何模板代码，包括 HTML：

```
<navigation-link url="/profile">
  <!-- 添加一个 Font Awesome 图标 -->
  <span class="fa fa-user"></span>
  个人中心
</navigation-link>
```

如果 `<navigation-link>` 没有包含一个 `<slot>` 元素，则该组件起始标签和结束标签之间的任何内容都会被抛弃。

作用域：

通过外面传给组件的变量，在以后使用插槽的时候是不能使用的。比如以上 `url` 只能在 `navigation-link` 中使用，但是后面使用插槽的时候不能使用。比如：

```

<navigation-link url="/profile">
  Clicking here will send you to: {{ url }}
<!--
  这里的 `url` 会是 undefined, 因为 "/profile" 是
  _传递给_ <navigation-link> 的而不是
  在 <navigation-link> 组件*内部*定义的。
-->
</navigation-link>

```

插槽默认值：

有时候在使用组件的时候，插槽中绝大部分情况是一种元素。那么我们就可以给插槽提供一个默认值，然后后面如果不像使用这个默认值的时候，就只需要提供自己定义的值就可以了。比如有一个叫做 `submit-button` 的组件，代码如下：

```

<button type="submit">
  <slot>提交</slot>
</button>

```

然后在使用这个组件的时候，可以直接 `<submit-button></submit-button>`，默认在里面就会显示“提交”文字。如果想要在使用的时候显示其他文字，那么也可以通过 `<submit-button>保存</submit-button>` 来实现。

命名插槽：

自定义组件中可以有多多个插槽，这时候就需要通过名字来进行区分了。其实如果没有指定名字，默认是有一个名字叫做 `default` 的。比如我们有一个名叫 `container` 的自定义组件：

```

<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>

```

以后在使用这个组件的时候使用 `v-slot:插槽名` 的方式来加载不同的数据：

```

<container>
  <template v-slot:header>
    这是头部信息
  </template>
  这是主要部分的信息
  <template v-slot:footer>
    这是网页尾部信息
  </template>
</container>

```

插槽作用域：

默认在插槽中的代码只能使用全局 `vue` 中的属性，如果想要使用自定义组件中的属性，那么需要在定义 `slot` 的时候使用 `v-bind` 来进行绑定。示例代码如下：

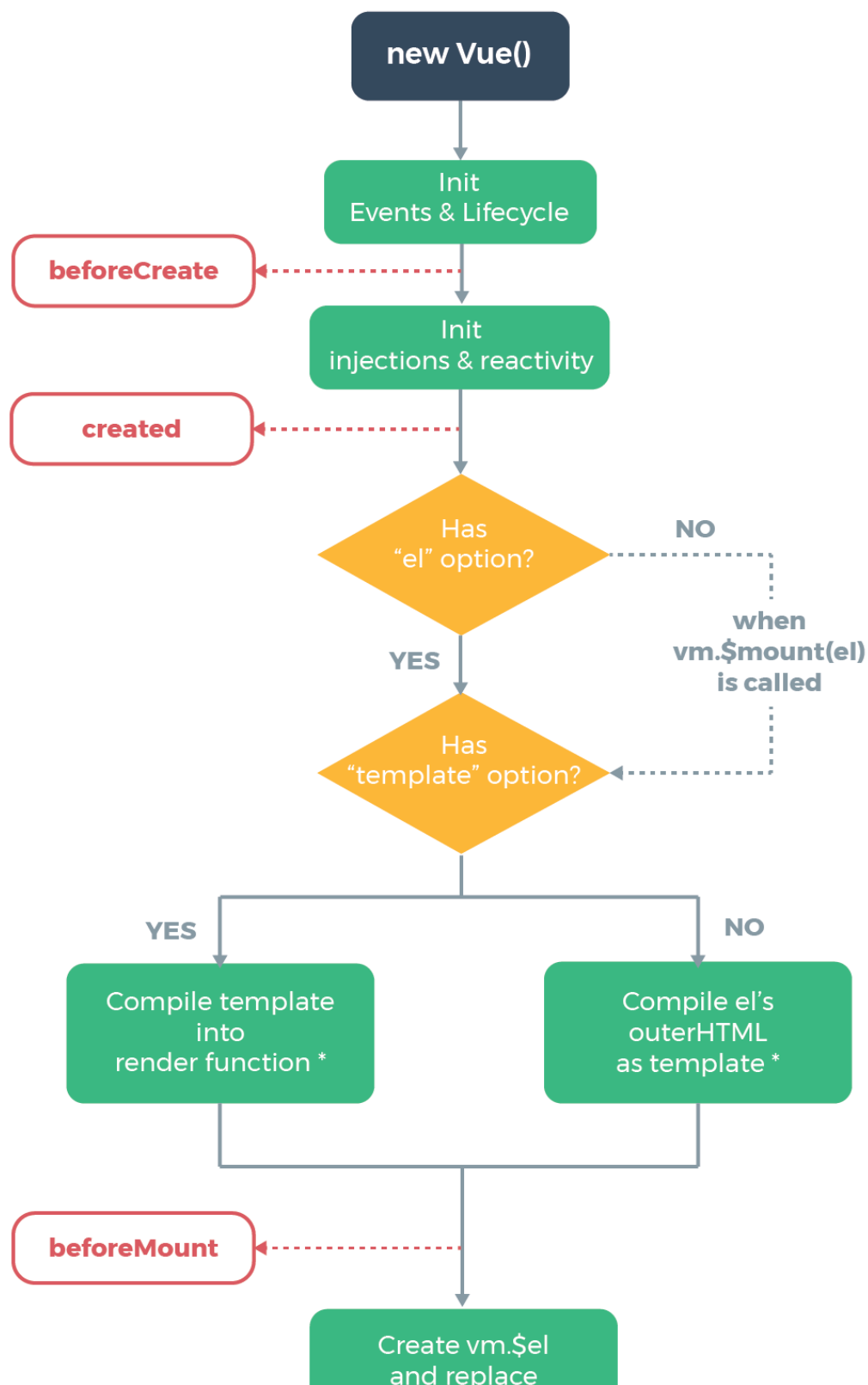
```

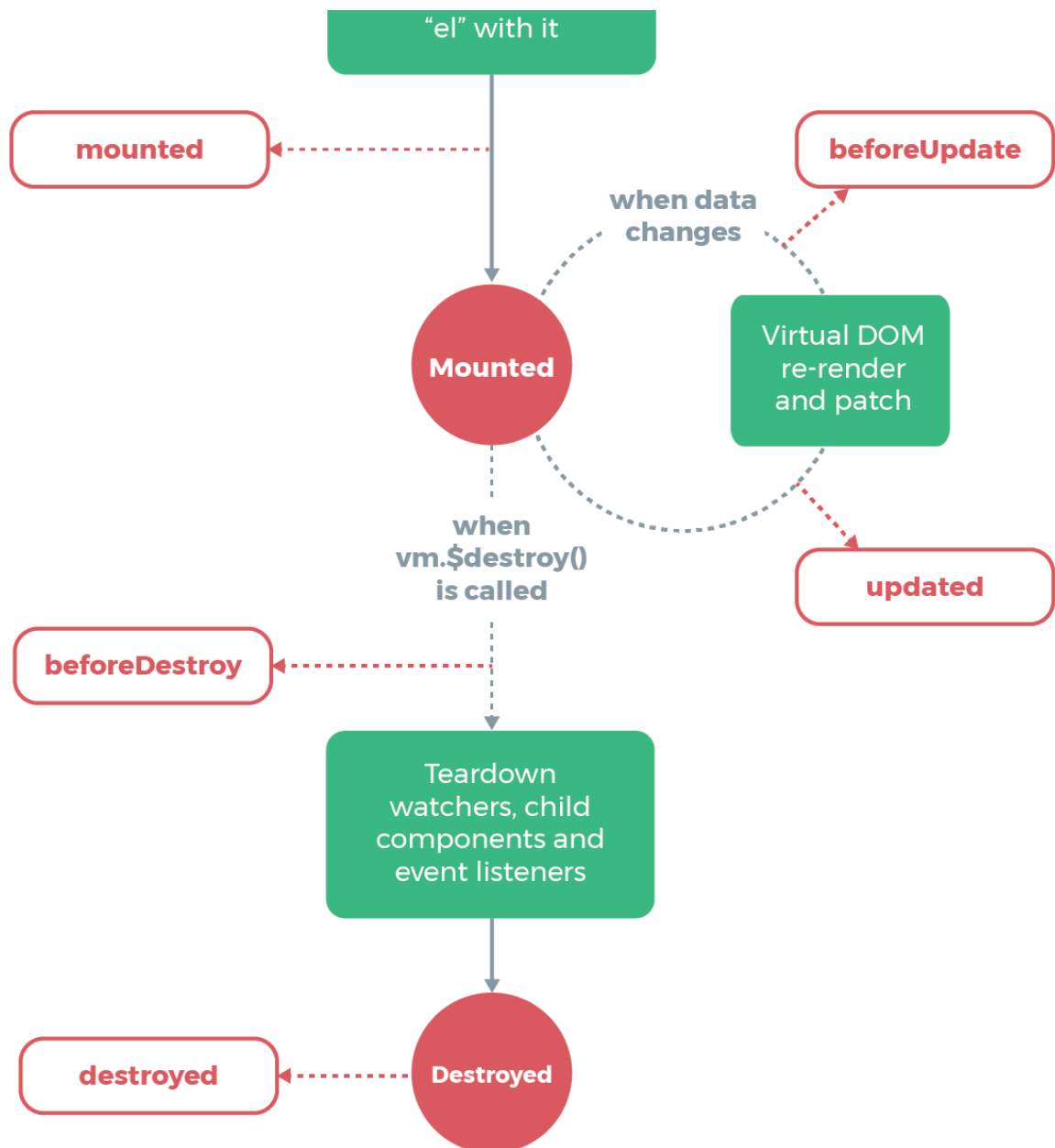
<div id="app">
  <sub-nav v-slot="slotProps">
    当前点击: {{slotProps.index}}
  </sub-nav>
</div>
<script>
  Vue.component('sub-nav', {
    props: ['url'],
    data: function(){
      return {
        navs: ['网络设置', '路由设置', '设备管理'],
        index: 0
      }
    },
    methods: {
      indexBtnClick: function(index){
        this.index = index;
      }
    },
    template: `
      <div class="container">
        <button v-for="(nav,index) in navs" @click="indexBtnClick(index)"
v-bind:key="index">{{nav}}</button>
        <slot v-bind:index="index"></slot>
      </div>
    `
  })
  new Vue({
    el: "#app"
  });
</script>

```

第八节 生命周期函数

生命周期函数代表的是 `Vue` 实例，或者是 `Vue` 组件，在网页中各个生命阶段所执行的函数。生命周期函数可以分为创建阶段和运行期间以及销毁期间。其中创建期间的函数有 `beforeCreate`、`created`、`beforeMount`、`mounted`；运行期间的函数有 `beforeUpdate`、`updated`；销毁期间有 `beforeDestroy`、`destroyed`。以下是官方文档给到的一张图，从这种图中我们可以了解到每个部分执行的函数。





* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

一、创建期间：

1.1 beforeCreate：

`vue` 或者组件刚刚实例化，`data`、`methods` 都还没有被创建。

1.2 created:

此时 `data` 和 `methods` 已经被创建，可以使用了。模板还没有被编译。

1.3 beforeMount:

`created` 的下一阶段。此时模板已经被编译了，但是并没有被挂到网页中。

1.4 mounted:

模板代码已经被加载到网页中了。此时创建期间所有事情都已经准备好了，网页开始运行了。

二、运行期间:

2.1 beforeUpdate:

在网页运行期间，`data` 中的数据可能会进行更新。在这个阶段，数据只是在 `data` 中更新了，但是并没有在模板中进行更新，因此网页中显示的还是之前的。

2.2 updated:

数据在 `data` 中更新了，也在网页中更新了。

三、销毁期间:

3.1 beforeDestroy:

`Vue` 实例或者是组件在被销毁之前执行的函数。在这一个函数中 `Vue` 或者组件中所有的属性都是可以使用的。

3.2 destroyed:

`Vue` 实例或者是组件被销毁后执行的。此时 `Vue` 实例上所有东西都会解绑，所有事件都会被移除，所有子元素都会被销毁。

第九节 过滤器

过滤器就是数据在真正渲染到页面中的时候，可以使用这个过滤器进行一些处理，把最终处理的结果渲染到网页中。

过滤器使用:

过滤器可以用在两个地方：**双花括号插值**和 **`v-bind` 表达式** (后者从**2.1.0+**开始支持)。过滤器应该被添加在 `JavaScript` 表达式的尾部，由“管道”符号指示：

```
<!-- 在双花括号中 -->
{{ message|capitalize }}
<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId|formatId"></div>
```

过滤器定义：

你可以在一个组件的选项中定义本地的过滤器：

```
filters: {  
  capitalize: function (value) {  
    if (!value) return ''  
    value = value.toString()  
    return value.charAt(0).toUpperCase() + value.slice(1)  
  }  
}
```

或者在创建 Vue 实例之前全局定义过滤器：

```
Vue.filter('capitalize', function (value) {  
  if (!value) return ''  
  value = value.toString()  
  return value.charAt(0).toUpperCase() + value.slice(1)  
})  
  
new Vue({  
  // ...  
})
```

过滤器其他：

串联：

过滤器在使用的时候可以使用多个管道符号 `|` 来进行串联，会把上一个过滤器的输出结果放到下一个过滤器中进行使用。示例代码如下：

```
{{ message | filterA | filterB }}
```

传递多个参数：

```
{{ message | filterA('arg1', arg2) }}
```

这里，`filterA` 被定义为接收三个参数的过滤器函数。其中 `message` 的值作为第一个参数，普通字符串 `'arg1'` 作为第二个参数，表达式 `arg2` 的值作为第三个参数。

第二章 Vue-Router

第一节 Vue-Router 简介

`Vue-Router` 是用来将一个 `Vue` 程序的多个页面进行路由的。比如一个 `Vue` 程序（或者说一个网站）有 `登录`、`注册`、`首页` 等模块，那么我们就可以定义 `/login`、`/register`、`/` 来映射每个模块。

安装：

1. 通过 script 加载进来：`<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>`。
2. 通过 npm 安装：`npm install vue-router`。

文档：

1. 官方文档：<https://router.vuejs.org/zh/>。
2. github地址：<https://github.com/vuejs/vue-router>。

版本：

本课程用到的最新的版本是：`3.1.3`。

第二节 Vue-Router 基础

路由基本

在网页中，经常需要发生页面更新或者跳转。这时候我们就可以使用 `Vue-Router` 来帮我们实现。`Vue-Router` 是用来做路由的，也就是定义 `url` 规则 与具体的 `view` 映射的关系。可以在一个单页面中实现数据的更新。

安装：

1. 使用

CDN

:

- 加载最新版的：`<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>`。
 - 加载指定版本的：`<script src="https://unpkg.com/vue-router@3.0.7/dist/vue-router.js"></script>`。
2. 下载到本地：`<script src="../../lib/vue-router.js"></script>`。
 3. 使用 npm 安装：`npm install vue-router`。

基本使用：

```
<div id="app">
  <div class="container-fluid">
    <div class="row">
      <div class="col-md-8 col-md-offset-2">
        <ul class="nav nav-tabs">
          <li role="presentation" class="active"><router-link
to="/find">发现音乐</router-link></li>
```



```

        <li role="presentation"><router-link to="/my">我的音
乐</router-link></li>
        <li role="presentation"><router-link to="/friend">朋友
</router-link></li>
    </ul>
    <!-- 路由出口 -->
    <!-- 路由匹配到的组件将渲染在这里 -->
    <router-view></router-view>
</div>
</div>
</div>
</div>

<script>
    var find = Vue.extend({template: "<h1>发现音乐</h1>"});
    var my = Vue.extend({template: "<h1>我的音乐</h1>"});
    var friend = Vue.extend({template: "<h1>朋友</h1>"});
    var routes = [
        {path: "/find", component: find},
        {path: "/my", component: my},
        {path: "/friend", component: friend},
        {path: "/", component: find}
    ]
    const router = new VueRouter({routes});
    new Vue({router}).$mount("#app");
</script>

```

解释：

1. 在 `vue-router` 中，使用 `<router-link>` 来加载链接，然后使用 `to` 表示跳转的链接。`vue-router` 最终会把 `<router-link>` 渲染成 `<a>` 标签。
2. `<router-view>` 是路由的出口，也就是相应 `url` 下的代码会被渲染到这个地方来。
3. `Vue.extend` 是用来加载模板的。
4. `routes` 是定义一个 `url` 与组件的映射，这个就是路由。
5. `VueRouter` 创建一个路由对象。
6. `$mount` 是挂载到哪个组件上。

动态路由：

在路由中有一些参数是会变化的，比如查看某个用户的个人中心，那肯定需要在 `url` 中加载这个人的 `id`，这时候就需要用到动态路由了。

示例代码如下：

```

<div id="app">
  <router-link to="/user/123">个人中心</router-link>
  <router-view></router-view>
</div>

<script>
  let UserProfile = {template:"<h1>个人中心: {{$route.params.userid}}</h1>"}
  var routes = [
    {path: "/user/:userid",component: UserProfile}
  ]
  const router = new VueRouter({routes});
  new Vue({router}).$mount("#app");
</script>

```

解释：

1. `:userid`：动态的参数。
2. `this.$route.params`：这个里面记录了路由中的参数。

组件复用：

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...
    }
  }
}

```

或者是使用后面跟大家讲到的**导航守卫**：

```

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

```

匹配404错误：

在路由规则中，`*` 代表的是任意字符。所以只要在路由的最后添加一个 `*` 路由，那么以后没有匹配到的 `url` 都会被导入到这个视图中。示例代码如下：

```
let UserProfile = {template: "<h1>个人中心: {{$route.params.userid}}</h1>"};
let NotFound = {template: "<h1>您找的页面已经到火星啦! </h1>"}
var routes = [
  {path: "/user/:userid", component: UserProfile},
  {path: "*", component: NotFound},
]
```

嵌套路由：

有时候在路由中，主要的部分是相同的，但是下面可能是不同的。比如访问用户的个人中心是 `/user/111/profile/`，查看用户发的贴子是 `/user/111/posts/` 等。这时候就需要使用到嵌套路由。示例代码如下：

```
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
        }
      ]
    }
  ]
});
```

编程式导航：

之前我们学习了使用 `<router-link>` 可以在用户点击的情况下进行页面更新。但有时候我们想要在 `js` 中手动的修改页面的跳转，这时候就需要使用编程式导航了。

`$router.push` 跳转：

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
<code><router-link :to="..."></code>	<code>router.push(...)</code>

```
// 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
```

`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
<code><router-link :to="..."></code>	<code>router.push(...)</code>

`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)
```

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 } })
```

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
      }
    }
  ]
})
```

重定向和别名：

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' } }
  ]
})
```

“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' }
  ]
})
```

官方文档：

更多内容请参考 `vue-router` 官方文档：<https://router.vuejs.org/zh/>

第三节 Vue-Router进阶

导航守卫：

导航守卫，就是导航过程中各个阶段的钩子函数。分为全局导航守卫、路由导航守卫、组件导航守卫。以下分别进行讲解。

全局导航守卫：

全局导航守卫，就是在整个网页中，只要发生了路由的变化，都会触发。全局导航守卫主要包括两个函数，分别为：`beforeEach`、`afterEach`。

`beforeEach`：

在路由发生了改变，但是还没有成功跳转的时候会调用。示例代码如下：

```
var router = new VueRouter({
  routes: [
    {path: "/", component: index, name: "index"},
    {path: "/account", component: account, name: "account"},
    {path: "/order", component: order, name: "order"},
    {path: "/login", component: login, name: "login"},
  ]
})

router.beforeEach(function(to, from, next){
  const authRoutes = ['account', 'order']
  if(authRoutes.indexOf(to.name) >= 0){
    if(logined){
      next()
    }else{
      next("/login")
    }
  }
  }else if(to.name == 'login'){
```

```
    if(logined){
      next("/")
    }
  }else{
    next()
  }
})
```

afterEach:

路由已经改变完成后的函数。这个函数没有 `next` 参数。因为页面已经完成了跳转。

```
router.afterEach(function(to,from){
  console.log('to:',to);
  console.log('from:',from);
})
```

路由导航守卫:

路由组件导航守卫，也就是定义在路由上的导航守卫。使用方式就是在定义路由的对象中，传递一个 `beforeEnter` 参数。示例代码如下：

```
var router = new VueRouter({
  routes: [
    {path: "/login",component:
login,name:"login",beforeEnter:function(to,from,next){
      if(logined){
        next("/")
      }else{
        next()
      }
    }},
  ],
})
```

组件导航守卫:

组件导航守卫，就是在组件中写守卫。也就是进入到这个组件之前会调用的方法。组件导航守卫大体上也是三个方法：`beforeRouteEnter`、`beforeRouteUpdate`、`beforeRouteLeave`。以下分别进行讲解。

beforeRouteEnter:

在进入组件之前调用的。示例代码如下：


```
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`
  // 因为当守卫执行前，组件实例还没被创建
  console.log('to:', to);
  console.log('from:', from);
  next(vm => {
    // 通过 `vm` 访问组件实例
    console.log(vm.username);
  })
}
```

beforeRouteUpdate:

```
beforeRouteUpdate(to, from, next) {
  // 在当前路由改变，但是该组件被复用时调用
  // 举例来说，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。而这个钩子就会在这个情况下被调用。
  // 可以访问组件实例 `this`
  console.log('to:', to);
  console.log('from:', from);
  next()
}
```

beforeRouteLeave:

```
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`
  const answer = window.confirm('您确定要离开这个页面吗? ')
  if (answer) {
    next()
  } else {
    next(false)
  }
}
```

导航解析流程:

1. 导航被触发。
2. 在失活的组件里调用离开守卫。
3. 调用全局的 beforeEach 守卫。
4. 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
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 (2.5+)。

9. 导航被确认。
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。

路由元信息：

在定义路由的时候，我们可以通过传递 `meta` 参数可以传递一些额外的信息。比如这个路由是需要登录后才能访问的，那么就可以在 `meta` 中添加一个 `requireAuth:true`。示例代码如下：

```
let router = new VueRouter({
  routes: [
    {path: "/",component: index},
    {path: "/profile/:userid",component: profile,meta:{requireAuth:true}},
    {path: "/login",component: login}
  ]
})

router.beforeEach(function(to,from,next){
  if(to.meta.requireAuth && !logged){
    next('/login')
  }else{
    next()
  }
})
```

第三章 Vue-lic

第一节 node环境配置：

nvm安装：

`nvm` (Node Version Manager) 是一个用来管理 `node` 版本的工具。我们之所以需要使用 `node`，是因为我们需要使用 `node` 中的 `npm`(Node Package Manager)，使用 `npm` 的目的是为了能够方便的管理一些前端开发的包！`nvm` 的安装非常简单，步骤如下：

1. 到这个链接下载 `nvm` 的安装包：<https://github.com/coreybutler/nvm-windows/releases>。
2. 然后点击一顿下一步，安装即可！
3. 安装完成后，还需要配置环境变量。在 我的电脑->属性->高级系统设置->环境变量->系统环境变量->Path 下新建一个，把 `nvm` 所处的路径填入进去即可！
4. 打开 `cmd`，然后输入 `nvm`，如果没有提示没有找到这个命令。说明已经安装成功！
5. `Mac` 或者 `Linux` 安装 `nvm` 请看这里：<https://github.com/creationix/nvm>。也要记得配置环境变量。

`nvm` 常用命令：

1. `nvm install [version]`: 安装指定版本的 `node.js`。
2. `nvm use [version]`: 使用某个版本的 `node`。
3. `nvm list`: 列出当前安装了哪些版本的 `node`。
4. `nvm uninstall [version]`: 卸载指定版本的 `node`。
5. `nvm node_mirror [url]`: 设置 `nvm` 的镜像。
6. `nvm npm_mirror [url]`: 设置 `npm` 的镜像。

node安装:

安装完 `nvm` 后, 我们就可以通过 `nvm` 来安装 `node` 了。这里我们安装 `10.16.0` 版本的 `node.js`。安装命令如下:

```
nvm install 10.16.0
```

如果你的网络够快, 那以上命令在稍等片刻之后会安装成功。如果你的网速很慢, 那以上命令可能会发生超时。因为 `node` 的服务器地址是 `https://nodejs.org/dist/`, 这个域名的服务器是在国外。因此会比较慢。因此我们可以设置一下 `nvm` 的源。

```
nvm node_mirror https://npm.taobao.org/mirrors/node/  
nvm npm_mirror https://npm.taobao.org/mirrors/npm/
```

npm:

`npm(Node Package Manager)` 在安装 `node` 的时候就会自动的安装了。当时前提条件是你需要设置当前的 `node` 的版本: `nvm use 10.16.0`。然后就可以使用 `npm` 了。关于 `npm` 常用命令以及用法, 请看下文。

初始化:

在新的项目中, 需要先执行 `npm init` 初始化, 创建一个 `package.json` 文件用来保存本项目中用到的包。

安装包:

安装包分为全局安装和本地安装。全局安装是安装在当前 `node` 环境中, 可以在 `cmd` 中当作命令使用。而本地安装是安装在当前项目中, 只有当前这个项目能使用, 并且可以通过 `require` 引用。安装的方式只有 `-g` 参数的区别:

```
npm install vue    # 本地安装  
npm install vue --save    # 本地安装, 并且保存到package.json的dependice中  
npm install vue --save-dev # 本地安装, 并且保存到package.json的dependice-dev中  
npm install vue -g    # 全局安装  
npm install -g @vue/cli #全局安装vue-cli
```

本地安装

1. 将安装包放在 `./node_modules` 下（运行 `npm` 命令时所在的目录），如果没有 `node_modules` 目录，会在当前执行 `npm` 命令的目录下生成 `node_modules` 目录。
2. 可以通过 `require()` 来引入本地安装的包。

全局安装

1. 将安装包放在 `/usr/local` 下或者你 `node` 的安装目录。
2. 可以直接在命令行里使用。

卸载包：

```
npm uninstall [package]
```

更新包：

```
npm update [package]
```

搜索包：

```
npm search [package]
```

使用淘宝镜像：

`npm` 的服务器在国外。那么可以安装一下 `cnpm`，并且指定镜像为淘宝的镜像：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

那么以后就可以使用 `cnpm` 来安装包了！

手动安装npm：

有时候使用 `nvm` 安装完 `node` 后，`npm` 没有跟着安装，这时候可以到 <https://github.com/npm/cli/releases> 下载 `6.10.1` 的版本。然后下载完成后，解压开来，放到 `v10.16.0/node_modules` 下，然后修改名字为 `npm`，并且把 `npm/bin` 中的 `npm` 和 `npm.cmd` 两个文件放到 `v10.16.0` 根目录下。

第二节 vue-cli

`vue-cli` 是和 `vue` 进行深度组合的工具，可以快速帮我们创建 `vue` 项目，并且把一些脚手架相关的代码给我们创建好。真正使用 `vue` 开发项目，都是用 `vue-cli` 来创建项目的。

安装：

Vue CLI 需要 Node.js 8.9 或更高版本 (推荐 8.11.0+)。node 环境安装后，直接通过 `npm install -g @vue/cli` 即可安装。安装完成后，输入 `vue --version`，如果出现了版本号，说明已经下载完成。

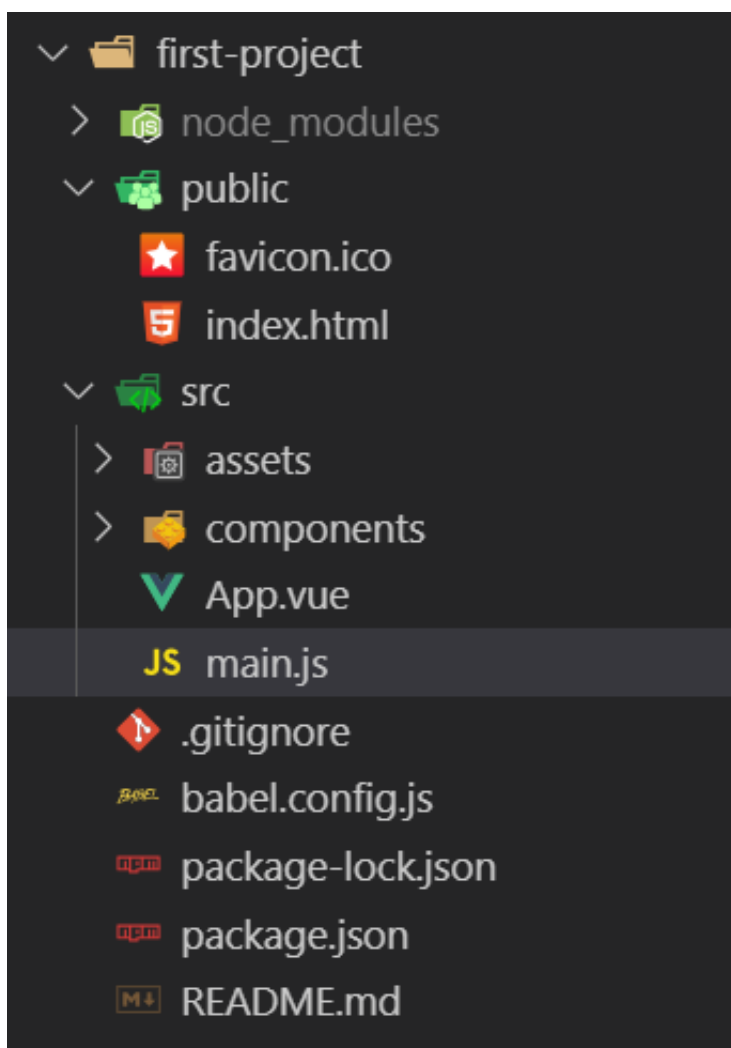
用命令行创建项目：

1. 在指定路径下使用 `vue create [项目名称]` 创建项目。
2. 会让你选择要安装哪些包（默认是 Babel 和 ESLint），也可以手动选择。
3. 如果在安装的时候比较慢，可以在安装的时候使用淘宝的链接：`vue create -r https://registry.npm.taobao.org [项目名称]`。
4. 如果实在不想在创建项目的时候都指定淘宝链接，可以在当前用户目录下，找到 `.npmrc`，然后设置 `registry=https://registry.npm.taobao.org`。

用界面创建项目：

1. 打开 `cmd`，进入到你项目存储的路径下。然后执行 `vue ui`，就会自动打开一个浏览器界面。
2. 按照指引进行选择，然后一顿下一步即可创建。

项目结构介绍：



1. `node_modules`：本地安装的包的文件夹。
2. `public`：项目出口文件。

`src`

：项目源文件：

- `assets`：资源文件，包括字体，图片等。
 - `components`：组件文件。
 - `App.vue`：入口组件。
 - `main.js`：`webpack` 在打包的时候的入口文件。
4. `babel.config.js`：`es*` 转低级 `js` 语言的配置文件。
 5. `package.json`：项目包管理文件。

组件定义和导入：

1. 定义：组件定义跟之前的方式是一模一样的。只不过现在模板代码专门放到 `.vue` 的 `template` 标签中，所以不再需要在定义组件的时候传入 `template` 参数。另外，因为需要让别的组件使用本组件，因此需要用 `export default` 将组件对象进行导出。
2. 导入：因为现在组件是在不同的文件中。所以如果需要引用，那么必须进行导入。用 `ES6` 语法的 `import XXX from XXX`。

局部样式：

默认情况下在 `.vue` 文件中的样式一旦写了，那么会变成全局的。如果只是想要在当前的组件中起作用，那么可以在 `style` 中加上一个 `scoped` 属性即可。示例代码如下：

```
<style scoped>
.info{
  background-color: red;
}
</style>
```

使用 `sass` 语法：

很多小伙伴在写样式代码的时候，不喜欢用原生 `css`，比较喜欢用比如 `sass` 或者 `less`，这里我们以 `sass` 为例，我们可以通过以下两个步骤来实现：

1. 安装

`loader`

:

`webpack`

在解析

```
scss
```

文件的时候，会去加载

```
sass-loader
```

以及

```
node-loader
```

，因此我们首先需要通过

```
npm
```

来安装一下：

```
npm install node-sass@4.12.0 --save-dev  
npm install sass-loader@7.0.3 --save-dev
```

2. 指定 sass 语言：在指定 `style` 的时候，添加 `lang="scss"` 属性，这样就会将 `style` 中的代码识别为 `scss` 语法。

第三节 Sass 介绍：

众所周知，`css` 不是一门编程语言。他没法像 `js` 和 `python` 那样拥有逻辑处理的能力，甚至导入其他的 `css` 文件中的样式都做不到。而 `Sass` 就是为了解决 `css` 的这些问题。他它允许你使用变量、嵌套规则、`mixins`、导入等众多功能，并且完全兼容 `css` 语法。`Sass` 文件不能被网页所识别，写完 `Sass` 后，还需要专门的工具转化为 `css` 才能使用。

Sass 文件的后缀名：

`Sass` 文件有两种后缀名，一个是 `scss`，一个是 `sass`。不同的后缀名，相应的语法也不一样。这里我们使用 `scss` 的后缀名。包括后面讲到的 `Sass` 语法，也都是 `scss` 的后缀名的语法。

Sass 基本语法：

注释：

支持 `/* comment */` 和 `//` 注释两种方式。

嵌套：

Sass 语法允许嵌套。比如 `#main` 下有一个类为 `.header`，那么我们可以写成以下的形式：

```
#main{
  background: #ccc;
  .header{
    width: 20px;
    height: 20px;
  }
}
```

这样写起来更加的直观。一看就知道 `.header` 是在 `#main` 下的。

引用父选择器（&）：

有时候，在嵌套的子选择器中，需要使用父选择器，那么这时候可以通过 `&` 来表示。示例代码如下：

```
a{
  font-weight: bold;
  text-decoration: none;
  &:hover{
    color: #888;
  }
}
```

定义变量：

是的，你没听错。在 Sass 中可以定义变量。对于一些比较常用的值，我们可以通过变量存储起来，以后想要使用的时候就直接用就可以了。定义变量使用 `$` 符号。示例代码如下：

```
$mainWidth: 980px;
#main{
  width: $mainWidth;
}
```

运算：

在 Sass 中支持运算。比如现在有一个容器总宽度是 `900`，要在里面平均放三个盒子，那么我们可以通过变量来设置他们的宽度。示例代码如下：

```
$mainWidth: 900px;
.box{
  width: $mainWidth/3;
}
```


@import语法:

在 `css` 中 `@import` 只能导入 `css` 文件, 而且对网站的性能有很大的影响。而 `Sass` 中的 `@import` 则是完全实现了一套自己的机制。他可以直接将指定文件的代码拷贝到导入的地方。示例代码如下:

```
@import "init.scss";
```

@extend语法:

有时候我们一个选择器中, 可能会需要另外一个选择器的样式, 那么我们就可以通过 `extend` 来直接将指定选择器的样式加入进来。示例代码如下:

```
.error{
  background-color: #fdd;
  border: 1px solid #f00;
}
.serious-error{
  @extend .error;
  border-width: 3px;
}
```

@mixin语法:

有时候一段样式代码。我们可能要用很多地方。那么我们可以把他定义成 `mixin`。需要用的时候就直接引用就可以了。示例代码如下:

```
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}
```

如果其他地方想要使用这个 `mixin` 的时候, 可以通过 `@include` 来包含进来。示例代码如下:

```
.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

`@mixin` 也可以使用参数。示例代码如下:

```
@mixin sexy-border($color, $width) {  
  border: {  
    color: $color;  
    width: $width;  
    style: dashed;  
  }  
}
```

那么以后在 `include` 的时候，就需要传递参数了。示例代码如下：

```
p {  
  @include sexy-border(blue, 1px);  
}
```

更详细的教程：

更详细的教程可以参考：<http://sass.bootcss.com/docs/sass-reference/>。

第四节 rem 移动端适配

在移动端（手机端、Pad端），设备尺寸参差不齐。如果想要写完一套代码，能够在所有移动设备上都正常运行，那么采用原生的 `px` 单位来设置是不行的。举个例子：`iPhone6` 的尺寸是 `375px`，如果我们想要一个盒子是占据整个宽度的一半，那么会将这个盒子的宽度设置为 `187.5px`，但是并不是所有手机的宽度都是 `375px`，因此就会造成问题。这时候我们可以通过 `rem` 来解决这个问题。

什么是 rem：

em

：当前元素字体大小相对于父标签的字体大小。比如：

```
<div style="font-size:16px;">  
  <span style="font-size:2em">你好</span>  
</div>
```

在

div

中字体大小是

16px

，而在

span

标签中因为用的是

2em

，因此是2倍的父标签字体的大小，也就是

32px

。

rem

：跟

em

类似，只不过此时的参照元素不是父元素，而是根元素，也就是

html

元素的大小。比如：

```
<html style="font-size:14px">
  <div style="font-size:16px;">
    <span style="font-size:2rem">你好</span>
  </div>
</html>
```

此时的

span

标签字体大小为

html

标签字体大小的2倍，也就是

28px

。

rem 适配原理：

rem 虽然本身是用来设置字体的大小，但是也可以延伸到设置其他属性的尺寸。利用 rem 我们可以实现等比缩放。比如设计师给的UI设计图是按照 750px 尺寸的，我们可以给 html 的 font-size 为 100px，那么以后我想实现一个 32px 的大小，转化成 rem 就是 0.32rem。这样写是没有问题的，但是如果用户现在的手机不是 750px 的，而是 375px 的，这时候直接写个 0.32rem 不是会有问题吗？目前这样来说是有问题，但是我们只需要设置 html 的 font-size 为 $\text{windowWidth}/750*100$ ，在这个公式中将 windowWidth 换算成 375px，结果为 50px，那么用 0.32rem 后的 px 为 16px，正好是 750px 尺寸的一半，达到了缩小一倍的效果。

在 vue-cli 中实现 rem 布局：

在使用 vue-cli 创建的项目中，我们可以通过一些第三方包来方便的实现 rem 的布局。要安装两个包，分别是：postcss-pxtorem、lib-flexible。安装命令通过 `npm install --save-dev postcss-pxtorem lib-flexible` 安装即可。在安装完包后，还需要配置两个地方：

package.json

:

```
"postcss": {
  "plugins": {
    "autoprefixer": {},
    "postcss-pxtorem": {
      "rootValue": 37.5,
      "propList": [
        "*"
      ],
      "selectorBlackList": [
        "van-*"
      ]
    }
  }
}
```

main.js

:

```
import "lib-flexible"
```

这样以后在写单位的时候，就不需要换算成 rem，直接写 px 就可以，postcss-pxtorem 会自动的将我们写的 px 转化成 rem。而 lib-flexible 会根据当前的尺寸，来调整 html 上的 font-size 进行适配。而其中的 37.5 则根据设计师设计图的尺寸来，比如设计师是用 375px 的尺寸来设计的，那么就是 $375/10$ 。

第五节 vant组件库使用

`vant` 库是有赞公司前端团队开源的一款针对 `vue` 库的组件库。里面集成了很多移动端用到的组件，包括按钮、图片、Icon图标等。而且因为有赞是一个做微商城的公司，所以有很多微商城的组件比如地址列表、商品卡片、优惠券等组件。

安装：

```
npm install vant --save
```

引入组件：

自动按需引入组件 (推荐)

`babel-plugin-import` 是一款 `babel` 插件，它会在编译过程中将 `import` 的写法自动转换为按需引入的方式

```
module.exports = {
  plugins: [
    ['import', {
      libraryName: 'vant',
      libraryDirectory: 'es',
      style: true
    }, 'vant']
  ]
};
// 接着你可以在代码中直接引入 vant 组件
// 插件会自动将代码转化为方式二中的按需引入形式
import { Button } from 'vant';
```

导入所有组件：

`vant` 支持一次性导入所有组件，引入所有组件会增加代码包体积，因此不推荐这种做法

```
import Vue from 'vue';
import Vant from 'vant';
import 'vant/lib/index.css';

Vue.use(Vant);
```

更多：

更多请参考：<https://youzan.github.io/vant/#/zh-CN/quickstart>。

第六节 Element UI组件库

element ui 组件库是由饿了么前端团队专门针对 vue 框架开发的组件库，专门用于电脑端网页的。因为里面集成了很多组件，所以使用他可以非常快速的帮我们实现网站的开发。

安装：

课程中用到的版本是 2.12.0，为了跟课程环境保持一致，安装的时候指定版本：

```
npm install element-ui@2.12.0 --save
```

引入：

引入的时候也是分成两种，一种是全部引入，一种是按需引入。这里我们跟大家介绍按需引入。

安装相关依赖包：

需要借助 babel-plugin-component 这个库，才能实现按需引入。安装的命令为：npm install babel-plugin-component --save-dev。

配置：

在 babel.config.js 中添加以下配置：

```
module.exports = {
  "presets": [
    "@vue/app"
  ],
  "plugins": [
    [
      "component",
      {
        "libraryName": "element-ui",
        "styleLibraryName": "theme-chalk"
      }
    ]
  ]
}
```

使用：

然后在项目中可以进行使用了。先进行导入，然后进行组件注册，最后再使用：

```
<template>
  <div id="app">
    <el-button type="primary">这是一个按钮</el-button>
  </div>
</template>

<script>
```

```
import {Button} from 'element-ui'

export default {
  name: 'app',
  components: {
    [Button.name]: Button
  }
}
</script>
```

第四章 Vuex

Vuex 是在中大型项目中，用来管理一些变量的。因为如果项目比较大，一些数据在各个页面中可能需要交替使用，如果数据量比较大，通过 `params` 或者 `query` 的方式不太方便。这时候就可以使用 Vuex 来管理这些数据。

安装：

1. 通过 `script` 安装：`<script src="https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js"></script>`。
2. 通过 `npm` 安装：`npm install vuex --save`。

如果是通过模块化打包的形式使用 `vuex`，则在使用之前需要先通过 `Vue.use(Vuex)` 来安装：

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)
```

`vuex` 默认依赖 `Promise`，如果你支持的浏览器并没有实现 `Promise` (比如 IE)，那么你可以使用 `es6-promise`。你可以通过 CDN 将其引入：

```
<script src="https://cdn.jsdelivr.net/npm/es6-promise@4/dist/es6-promise.auto.js"></script>
```

或者通过 `npm install es6-promise --save`。如果是用模块化打包的形式使用，那么还需要导入一下：

```
import 'es6-promise/auto'
```

基本使用：

每一个 `vuex` 应用的核心就是 `store` (仓库)。`store` 可以理解为就是一个容器，它包含着你的应用中大部分的状态 (`state`) (就是变量)。`Vuex` 和单纯的全局对象有以下两点不同：

1. `Vuex` 的状态存储是响应式的。当 `Vue` 组件从 `store` 中读取状态的时候，若 `store` 中的状态发生变

化，那么相应的组件也会相应地得到高效更新。

2. 你不能直接改变 `store` 中的状态。改变 `store` 中的状态的唯一途径就是显式地提交 (`commit`) `mutation`。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

示例代码如下：

```
<div id="app">
  <counter></counter>
</div>

<template id="counter">
  <div>
    <div></div>
    <button @click="updateCount">点击修改</button>
  </div>
</template>

<script>
  const store = new Vuex.Store({
    state: {
      count: 0
    },
    mutations: {
      increment(state){
        state.count++
      }
    }
  })

  const Counter = Vue.component("counter", {
    template: "#counter",
    computed: {
      count(){
        return this.$store.state.count
      }
    },
    methods: {
      updateCount: function(){
        this.$store.commit("increment")
      }
    }
  })

  var vm = new Vue({
    el: "#app",
    store,
    components: {
      "counter": Counter
    }
  })
```



```
    }  
  })  
</script>
```

mapState:

当一个组件需要获取多个状态时候，将这些状态都声明为计算属性会有些重复和冗余。为了解决这个问题，我们可以使用 `mapState` 辅助函数帮助我们生成计算属性，让你少按几次键：

```
// 在单独构建的版本中辅助函数为 Vuex.mapState  
import { mapState } from 'vuex'  
  
export default {  
  // ...  
  computed: mapState({  
    // 箭头函数可使代码更简练  
    count: state => state.count,  
  
    // 传字符串参数 'count' 等同于 `state => state.count`  
    countAlias: 'count',  
  
    // 为了能够使用 `this` 获取局部状态，必须使用常规函数  
    countPlusLocalState (state) {  
      return state.count + this.localCount  
    }  
  })  
}
```

对象展开运算符:

有时候我们想要将 `vuex` 中的变量添加到我们当前已有的 `computed` 中。这时候就可以使用对象展开运算符：

```
computed: {  
  localComputed () { /* ... */ },  
  // 使用对象展开运算符将此对象混入到外部对象中  
  ...mapState({  
    // ...  
  })  
}
```

getter:

有时候我们需要从 `store` 中的 `state` 中派生出一些状态，例如对列表进行过滤并计数：

```
computed: {
  doneTodosCount () {
    return this.$store.state.todos.filter(todo => todo.done).length
  }
}
```

如果后期很多地方都要使用到这个变量，那么每次在组件中都写一遍肯定是不合适的，这时候就可以使用 `vuex` 提供的 `getter` 来实现：

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})
```

第五章 Django Rest Framework

第一节 DRF介绍

DRF介绍：

DRF 是 Django Rest Framework 单词的简写，是在 Django 框架中实现 Restful API 的一个插件，使用他可以非常方便的实现接口数据的返回。Django 中也可以使用 `JsonResponse` 直接返回 `json` 格式的数据，但是 DRF 相比直接使用 Django 返回 `json` 数据有以下几个好处：

1. 可以自动生成 API 文档，在前后端分离开发的时候进行沟通比较有用。
2. 授权验证策略比较完整，包含 `OAuth1` 和 `OAuth2` 验证。
3. 支持 ORM 模型和非 ORM 数据的序列化。
4. 高度封装了视图，使得返回 `json` 数据更加的高效。

安装：

`drf` 目前最新的版本是 `3.10`，需要以下依赖：

1. Python (3.5, 3.6, 3.7)
2. Django (1.11, 2.0, 2.1, 2.2)

准备好以上依赖后，可以通过 `pip install djangorestframework` 安装最新的版本。当然为了跟课程中的环境保持一致，可以安装 `3.10` 的版本。

基本使用：

一、注册rest_framework：

安装完后，使用他还需要进行在 `settings.INSTALLED_APPS` 中进行安装。

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

二、创建app和模型：

创建一个名叫 `meituan` 的 `app`，然后在 `meituan.models` 中创建以下模型：

```
from django.db import models
from django.contrib.auth.models import User

class Merchant(models.Model):
    """
    商家
    """
    name = models.CharField(max_length=200, verbose_name='商家名称', null=False)
    address = models.CharField(max_length=200, verbose_name='商家', null=False)
    logo = models.CharField(max_length=200, verbose_name='商家logo', null=False)
    notice = models.CharField(max_length=200, verbose_name='商家的公
告', null=True, blank=True)
    up_send = models.DecimalField(verbose_name='起送
价', default=0, max_digits=6, decimal_places=2)
    lon = models.FloatField(verbose_name='经度')
    lat = models.FloatField(verbose_name='纬度')

    created = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)

class GoodsCategory(models.Model):
    """
    商家商品分类
    """
    name = models.CharField(max_length=20, verbose_name='分类名称')
    merchant =
models.ForeignKey(Merchant, on_delete=models.CASCADE, verbose_name='所属商
家', related_name='categories')

class Goods(models.Model):
    """
    商品
    """
```

```

name = models.CharField(max_length=200,verbose_name='商品名称')
picture = models.CharField(max_length=200,verbose_name='商品图片')
intro = models.CharField(max_length=200)
price = models.DecimalField(verbose_name='商品价格',max_digits=6,decimal_places=2) # 最多6位数, 2位小数。9999.99
category =
models.ForeignKey(GoodsCategory,on_delete=models.CASCADE,related_name='goods_list')

```

三、添加测试数据：

创建完模型后，运行 `makemigrations` 和 `migrate` 后把模型映射到 `mysql` 数据库中。然后在 `navicat` 中，把 `meituan_merchant.sql` 文件运行后，添加测试数据。

四、编写Serializers：

在 `meituan` 这个 `app` 中新创建一个文件 `serializers.py`，然后添加以下代码：

```

from rest_framework import serializers
from .models import Merchant, GoodsCategory, Goods

class MerchantSerializer(serializers.ModelSerializer):
    class Meta:
        model = Merchant
        fields = "__all__"

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = GoodsCategory
        fields = "__all__"

class GoodsSerializer(serializers.ModelSerializer):
    class Meta:
        model = Goods
        fields = "__all__"

```

五、编写视图：

使用 `drf` 我们可以非常方便的创建包含 `get/post` 等 `method` 的视图。在 `meituan.views` 中添加以下代码：

```

class MerchantViewSet(ModelViewSet):
    serializer_class = MerchantSerializer
    queryset = Merchant.objects.all()

```

六、编写路由：

在 `meituan.urls` 中添加以下代码：

```
from rest_framework.routers import DefaultRouter
from . import views

router = DefaultRouter(trailing_slash=False)
router.register('merchant', views.MerchantViewSet, basename='merchant')

urlpatterns = [
] + router.urls
```

然后再在项目的 `urls.py` 中把 `meituan` 的路由添加进去：

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('meituan/', include("meituan.urls"))
]
```

以后我们就可以使用不同的 `method` 向 `/meituan/merchant` 发送请求。比如用 `get`，那么就会返回 `merchant` 的列表，比如用 `post`，那么就会向 `merchant` 表添加数据。

第二节 序列化

`drf` 中的序列化主要是用来将模型序列化成 `JSON` 格式的对象。但是除了序列化，他还具有表单验证功能，数据存储和更新功能。以下将进行讲解。

创建一个Serializer类：

这里我们以上一节的模型 `Merchant`、`GoodsCategory`、`Goods` 为例来讲解。首先我们创建一个 `Merchant` 的 `Serializer` 类。必须继承自 `Serializer` 及其子类。示例代码如下：

```
from rest_framework import serializers
from .models import Merchant, GoodsCategory, Goods

class MerchantSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(required=True, max_length=200)
    logo = serializers.CharField(required=True, max_length=200)
    notice = serializers.CharField(max_length=200, required=False)
    up_send =
serializers.DecimalField(max_digits=6, decimal_places=2, required=False)
    lon = serializers.FloatField(required=True)
```

```

lat = serializers.FloatField(required=True,error_messages={"required": "必须传入lat! "})

def create(self, validated_data):
    # create方法实现
    return Merchant.objects.create(**validated_data)

def update(self,instance, validated_data):
    # update方法实现
    instance.name = validated_data.get('name',instance.name)
    instance.logo = validated_data.get('logo',instance.logo)
    instance.notice = validated_data.get('notice',instance.notice)
    instance.up_send = validated_data.get('up_send',instance.up_send)
    instance.lon = validated_data.get('lon',instance.lon)
    instance.lat = validated_data.get('lat',instance.lat)
    instance.save()
    return instance

```

那么以后在视图函数中，可以使用他来对数据进行序列化，也可以对数据进行校验，然后存储数据。比如以下在视图函数中使用：

```

from .models import Merchant
from .serializers import MerchantSerializer
from django.http.response import JsonResponse
from django.views.decorators.http import require_http_methods

@require_http_methods(['GET', 'POST'])
def merchant(request):
    if request.method == 'GET':
        merchants = Merchant.objects.all()
        serializer = MerchantSerializer(merchants,many=True)
        return JsonResponse(serializer.data,safe=False)
    else:
        serializer = MerchantSerializer(data=request.POST)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data,status=200)
        return JsonResponse(serializer.errors,status=400)

```

ModelSerializer:

之前我们在写序列化类的时候，几乎把模型中所有的字段都写了一遍，我们可以把模型中的字段移植过来即可。这时候就可以使用 `ModelSerializer` 类实现。示例代码如下：

```
class MerchantSerializer(serializers.ModelSerializer):
    class Meta:
        model = Merchant
        fields = "__all__"
```

在视图函数中也是一样的用法。

Serializer的嵌套：

有时候在一个序列化中，我们可能需要其他模型的序列化。这时候就可以使用到序列化的嵌套。比如我们在 `GoodsCategory` 中想要获取 `Merchant` 以及这个分类下的商品 `Goods`（只是为了演示，实际情况不一定要全部返回）。那么示例代码如下：

```
class GoodsSerializer(serializers.ModelSerializer):
    class Meta:
        model = Goods
        fields = "__all__"

class MerchantSerializer(serializers.ModelSerializer):
    class Meta:
        model = Merchant
        fields = "__all__"

class GoodsCategorySerializer(serializers.ModelSerializer):
    merchant = MerchantSerializer(read_only=True, required=False)
    goods_list = GoodsSerializer(many=True, required=False)
    merchant_id = serializers.IntegerField(required=True, write_only=True)
    class Meta:
        model = GoodsCategory
        fields = "__all__"

    def validate_merchant_id(self, value):
        if not Merchant.objects.filter(pk=value).exists():
            raise serializers.ValidationError("商家不存在!")
        return value

    def create(self, validated_data):
        merchant_id = validated_data.get('merchant_id')
        merchant = Merchant.objects.get(pk=merchant_id)
        category =
GoodsCategory.objects.create(name=validated_data.get('name'),
merchant=merchant)
        return category
```

视图函数的写法还是跟之前一样：

```
@require_http_methods(['GET', 'POST'])
def goods_category(request):
    if request.method == 'GET':
        categories = GoodsCategory.objects.all()
        serializer = GoodsCategorySerializer(categories, many=True)
        return JsonResponse(serializer.data, safe=False)
    else:
        serializer = GoodsCategorySerializer(data=request.POST)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        else:
            return JsonResponse(serializer.errors, status=400)
```

关于read_only和write_only:

1. `read_only=True`: 这个字段只能读，只有在返回数据的时候会使用。
2. `write_only=True`: 这个字段只能被写，只有在新增数据或者更新数据的时候会用到。

验证:

验证用户上传上来的字段是否满足要求。可以通过以下三种方式来实现。

1. 验证在 `Field` 中通过参数的形式进行指定。比如 `required` 等。
2. 通过重写 `validate(self, attrs)` 方法进行验证。`attrs` 中包含了所有字段。如果验证不通过，那么调用 `raise serializer.ValidationError('error')` 即可。
3. 重写 `validate_字段名(self, value)` 方法进行验证。这个是针对某个字段进行验证的。如果验证不通过，也可以抛出异常。

更多:

更多请参考:

1. `Serializer`: <https://www.django-rest-framework.org/api-guide/serializers/>。
2. `Serializes Fields` 及其参数: <https://www.django-rest-framework.org/api-guide/fields/>。

第三节 Request和Response

在 drf 中，可以使用 `Request` 和 `Response` 对象来替代 django 内置的 `HttpRequest` 和 `HttpResponse`。替代 django 的对象有很多好处。以下进行简单讲解。

Request对象:

DRF 的 `Request` 对象是从 `HttpRequest` 中拓展出来的，但是增加了一些其他的属性。其中最核心的用得最多的属性便是 `request.data`。`request.data` 比 `request.POST` 更加灵活:

1. `request.POST`: 只能处理表单数据，获取通过 `POST` 方式上传上来的数据。

2. `request.data`：可以处理任意的数据。可以获取通过 `POST`、`PUT`、`PATCH` 等方式上传上来的数据。
3. `request.query_params`：查询参数。比 `request.GET` 更用起来更直白。

Response对象：

`Response` 可以自动的根据返回的数据类型来决定返回什么样的格式。并且会自动的监听如果是浏览器访问，那么会返回这个路由的信息。

状态码：

在 `Restful API` 中，响应的状态码是很重要的一部分。比如请求成功是 `200`，参数错误是 `400` 等。但是具体某个状态码是干什么的，`django` 是没有做过多的解释（这也不是 `django` 所需要解决的问题，因为他只是个 `web` 框架），对于一些初学者而言用起来会有点迷糊。这时候我们可以使用 `DRF` 提供的状态码。比如：

```
from rest_framework.response import Response
from rest_framework import status

@api_view(['GET', 'POST', 'PUT', 'DELETE'])
def merchant(request):
    return Response({"username": "zhiliao"}, status=status.HTTP_200_OK)
```

实现APIView：

以上的 `Response` 和 `Request` 对象都只能在 `DRF` 的 `APIView` 中才能使用。如果是视图函数，那么可以使用装饰器 `rest_framework.decorators.api_view` 进行装饰，这个装饰器中可以传递本视图函数可以使用什么 `method` 进行请求。示例代码如下：

```
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
```

```

        return Response(serializer.data)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

elif request.method == 'DELETE':
    snippet.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)

```

如果是类视图，那么可以让你的类继承自 `rest_framework.views.APIView`。示例代码如下：

```

from rest_framework.views import APIView

class MerchantView(APIView):
    def get(self, request):
        return Response("你好")

```

第四节 类视图

在 DRF 中，推荐使用类视图，因为类视图可以通过继承的方式把一些重复性的工作抽取出来，而使得代码更加简洁。当然如果你不想使用类视图，那么就用 `@api_view` 装饰器包裹一下就可以。

APIView：

`APIView` 是 DRF 中类视图最基本的父类。基本用法跟 Django 中自带的 `View` 类是一样的。也是自己分别实现 `get`、`post` 等方法。示例代码如下：

```

class MerchantView(APIView):
    """
    检索，更新和删除一个merchant实例对象。
    """
    def get_object(self, pk):
        try:
            return Merchant.objects.get(pk=pk)
        except Merchant.DoesNotExist:
            raise Http404

    def get(self, request, pk=None):
        if pk:
            merchant = self.get_object(pk)
            serializer = MerchantSerializer(merchant)
            return Response(serializer.data)
        else:
            queryset = Merchant.objects.all()
            serializer = MerchantSerializer(instance=queryset, many=True)
            return Response(serializer.data)

    def put(self, request, pk):
        merchant = self.get_object(pk)
        serializer = MerchantSerializer(merchant, data=request.data)

```

```

        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        merchant= self.get_object(pk)
        merchant.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)

```

当然 `APIView` 中还继承了一些常用的属性，比如

`authentication_classes`、`permission_classes`、`throttle_classes` 等。

Mixins:

`mixins` 翻译成中文是混入，组件的意思。在 `DRF` 中，针对获取列表，检索，创建等操作，都有相应的 `mixin`。示例代码如下：

```

from .models import Merchant
from .serializers import MerchantSerializer
from rest_framework import mixins
from rest_framework import generics

class MerchantView(
    generics.GenericAPIView,
    mixins.ListModelMixin,
    mixins.RetrieveModelMixin,
    mixins.CreateModelMixin,
    mixins.UpdateModelMixin,
    mixins.DestroyModelMixin
):
    queryset = Merchant.objects.all()
    serializer_class = MerchantSerializer

    def get(self, request, pk=None):
        if pk:
            return self.retrieve(request)
        else:
            return self.list(request)

    def post(self, request):
        return self.create(request)

    def put(self, request, pk=None):
        return self.update(request)

    def delete(self, request, pk=None):
        return self.destroy(request)

```

以上我们通过继承 `generics.GenericAPIView`，可以设置 `queryset` 以及 `serializer_class`，那么视图函数就知道你是要针对哪个模型做处理，你的序列化的类是什么了。接着我们继承 `mixins.ListModelMixin/CreateModelMixin` 类，这样 `MerchantList` 就拥有了获取列表，以及创建数据的功能。下面我们通过写 `get` 和 `post` 方法，调用 `self.list` 和 `self.create` 方法，就可以轻松的实现获取商家列表和创建商家的功能。

Generic类视图：

以上我们通过 `mixin` 可以非常方便的实现一些 `CURD` 操作。实际上针对这些 `mixin`，`DRF` 还进一步的进行了封装，放到 `generics` 下。有以下 `generic` 类视图：

1. `generics.ListAPIView`：实现获取列表的。实现 `get` 方法。
2. `generics.CreateAPIView`：实现创建数据的。实现 `post` 方法。
3. `generics.UpdateAPIView`：实现更新数据的。实现 `put` 方法。
4. `generics.DestroyAPIView`：实现删除数据的。实现 `delete` 方法。
5. `generics.RetrieveAPIView`：实现检索数据的。
6. `generics.ListCreateAPIView`：实现列表和创建数据的。
7. `generics.RetrieveUpdateAPIView`：实现检索和更新数据的。
8. `generics.RetrieveDestroyAPIView`：实现检索和删除数据的。
9. `generics.RetrieveUpdateDestroyAPIView`：实现检索和更新和删除数据的。

用法如下：

```
class MerchantView(  
    generics.CreateAPIView,  
    generics.UpdateAPIView,  
    generics.DestroyAPIView,  
    generics.RetrieveAPIView  
):  
    serializer_class = MerchantSerializer  
    queryset = Merchant.objects.all()
```

那么在定义 `url` 与视图映射的时候，还是按照之前的写法就够了：

```
urlpatterns = [  
    path('merchant/', views.MerchantView.as_view()),  
    path('merchant/<int:pk>', views.MerchantView.as_view())  
    # path('category', views.goods_category, name="category")  
]
```

请求的 `url` 和 `method` 产生的结果如下：

method	url	结果
get	/merchant/31/	获取id=31的merchant数据
post	/merchant/	添加新的merchant数据
put	/merchant/31/	修改id=31的merchant数据
delete	/merchant/31	删除id=31的merchant数据

因为这里 `retrieve` 占用了 `get` 方法，所以如果想要实现获取列表的功能，那么需要再重新定义一个 `url` 和视图：

```
# views.py
class MerchantListView(generics.ListAPIView,):
    serializer_class = MerchantSerializer
    queryset = Merchant.objects.all()

# urls.py
urlpatterns = [
    path('merchant/', views.MerchantView.as_view()),
    path('merchant/<int:pk>/', views.MerchantView.as_view()),
    path('merchants/', views.MerchantListView.as_view())
    ...
]
```

这也是为什么 `List` 和 `Retrieve` 不能同时存在一个视图中的原因。

GenericAPIView介绍：

如果想要深入学会 `generic` 的一些用法。比如如何分页，如何过滤数据等。那么这时候就需要学习 `GenericAPIView` 的使用。

queryset:

`queryset` 是用来控制视图返回给前端的数据。如果没什么逻辑，可以直接写在视图的类属性中，如果逻辑比较复杂，也可以重写 `get_queryset` 方法用来返回一个 `queryset` 对象。如果重写了 `get_queryset`，那么以后获取 `queryset` 的时候就需要通过调用 `get_queryset` 方法。因为 `queryset` 这个属性只会调用一次，以后所有的请求都是使用他的缓存。

serializer_class:

`serializer_class` 用来验证和序列化数据的。也是可以通过直接设置这个属性，也可以通过重写 `get_serializer_class` 来实现。

lookup_field和lookup_url_kwarg:

1. `lookup_field`: 是在检索的时候, 根据什么参数进行检索。默认是 `pk`, 也就是主键。
2. `lookup_url_kwarg`: 在检索的 `url` 中的参数名称。默认没有设置, 跟 `lookup_field` 保持一致。

分页:

分页是通过设置 `pagination_class` 来实现的。默认这个属性的值是 `rest_framework.pagination.PageNumberPagination`, 也就是通过控制页码, 每页的数量来实现的。我们可以通过在 `settings.REST_FRAMEWORK` 中设置 `PAGE_SIZE` 来控制每页的数量, 然后在 `url` 中通过传递 `page` 参数来获取指定页数的数据。

重写方法:

1. `get_queryset(self)`:

用于动态的返回一个 `queryset` 对象。

2. `get_object(self)`:

用于在数据检索的时候, 返回一条数据的。

3. `perform_create(self,serializer)`:

保存对象的时候调用。

4. `perform_update(self,serializer)`:

更新对象的时候调用。

5. `perform_destroy(self,serializer)`:

删除对象的时候调用。

第五节 ViewSet视图集

`ViewSet` 视图集, 相当于是之前我们学习视图的一个集合。在视图集中, 不再有 `get` 和 `post`, 取而代之的是 `list` 和 `create`。以下分别进行讲解。

基本使用:

比如我们想实现一个包含增、删、改、查、列表的视图集。我们可以通过以下代码来实现:

```
from rest_framework import viewsets
from .models import Merchant
from .serializers import MerchantSerializer
from rest_framework.response import Response
from rest_framework import status
from django.shortcuts import get_object_or_404

class MerchantViewSet(viewsets.ViewSet):
```

```

def list(self, request):
    queryset = Merchant.objects.all()
    serializer = MerchantSerializer(queryset, many=True)
    return Response(data=serializer.data)

def create(self, request):
    serializer = MerchantSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response("success")
    else:
        return Response("fail", status=status.HTTP_400_BAD_REQUEST)

def retrieve(self, request, pk=None):
    queryset = Merchant.objects.all()
    merchant = get_object_or_404(queryset, pk=pk)
    serializer = MerchantSerializer(merchant)
    return Response(serializer.data)

def update(self, request, pk=None):
    queryset = Merchant.objects.all()
    merchant = get_object_or_404(queryset, pk=pk)
    serializer = MerchantSerializer()
    serializer.update(merchant, request.data)
    return Response('success')

def destroy(self, request, pk=None):
    queryset = Merchant.objects.all()
    merchant = get_object_or_404(queryset, pk=pk)
    merchant.delete()
    return Response('success')

```

然后在 `urls.py` 中, 通过 `rest_framework.routers.DefaultRouter` 注册路由即可。示例代码如下:

```

router = DefaultRouter()
router.register("merchant", MerchantViewSet, basename="merchant")

urlpatterns = [] + router.urls

```

那么以后通过相应的 `method` 和 `url` 即可进行操作。

ModelViewSet:

因为我们一个视图集基本上都是针对一个模型进行操作的, 那么增删改查操作针对的也就是不同的模型, 所以我们可以使用 `ModelViewSet` 简化以上的代码。比如以上代码, 我们可以写成:

```
class MerchantViewSet(viewsets.ModelViewSet):
    queryset = Merchant.objects.all()
    serializer_class = MerchantSerializer
```

有时候在一个视图中，我们可能还需要增加其他的 `url`，这时候就可以使用 `@action` 来实现：

```
from rest_framework.decorators import action

class MerchantViewSet(viewsets.ModelViewSet):
    queryset = Merchant.objects.all()
    serializer_class = MerchantSerializer

    @action(['GET'], detail=False)
    def cs(self, request, *args, **kwargs):
        queryset = self.get_queryset()
        queryset = queryset.filter(name__contains="长沙")
        serializer = MerchantSerializer(queryset, many=True)
        return Response(serializer.data)
```

`urls.py` 路由部分不需要修改。以后直接可以通过 `/merchant/cs/` 可以访问到 `name` 中包含了 "长沙" 两个字的所有商家。

第六节 认证和权限

认证：

认证可以简单的理解为登录和访问需要登录的接口的认证。只要认证通过了，那么在 `request` 对象（是 `drf` 的 `request` 对象）上便有两个属性，一个是 `request.user`，一个是 `request.auth`，前者就是 `django` 中的 `User` 对象，后者根据不同的认证机制有不同的对象。`DRF` 内置了几个认证的模块。以下进行简单了解。

`rest_framework.authentication.BasicAuthentication`：

基本的授权。每次都要在 `Header` 中把用户名和密码传给服务器，因此不是很安全，不能在生产环境中使用。

`rest_framework.authentication.SessionAuthentication`：

基于 `django` 的 `session` 机制实现的。如果前端部分是网页，那么用他是可以的，如果前端是 `ios` 或者 `Android` 的 `app`，用他就不太方便了（如果要用也是完全可以的）。

`rest_framework.authentication.TokenAuthentication`：

基于 `token` 的认证机制。只要登录完成后便会返回一个 `token`，以后请求一些需要登录的 `api`，就通过传递这个 `token` 就可以了，并且这个 `token` 是存储在服务器的数据库中的。但是这种 `token` 的方式有一个缺点，就是他没有自动过期机制，一旦登录完成后，这个 `token` 是永久有效的，这是不安全的。

JSON Web Token认证机制：

JSON Web Token 简称 JWT。在前后端分离的项目中，或者是 app 项目中。推荐使用 JWT。JWT 是在成功后，把用户的相关信息（比如用户id）以及过期时间进行加密，然后生成一个 token 返回给客户端，客户端拿到后可以存储起来，以后每次请求的时候都携带这个 token，服务器在接收到需要登录的 API 请求候，对这个 token 进行解密，然后获取过期时间和用户信息（比如用户id），如果过期了或者用户信息不对，那么都是认证失败。JWT 的相关代码如下：

```
import jwt
from django.conf import settings
from rest_framework.authentication import
BaseAuthentication, get_authorization_header
from rest_framework import exceptions
from django.contrib.auth import get_user_model
from jwt.exceptions import ExpiredSignatureError
MTUser = get_user_model()
import time

def generate_jwt(user):
    expire_time = int(time.time() + 60*60*24*7)
    return
jwt.encode({"userid":user.pk, "exp":expire_time}, key=settings.SECRET_KEY).decode(
('utf-8'))

class JWTAuthentication(BaseAuthentication):
    keyword = 'JWT'
    def authenticate(self, request):
        auth = get_authorization_header(request).split()

        if not auth or auth[0].lower() != self.keyword.lower().encode():
            return None

        if len(auth) == 1:
            msg = "不可用的JWT请求头！"
            raise exceptions.AuthenticationFailed(msg)
        elif len(auth) > 2:
            msg = '不可用的JWT请求头！JWT Token中间不应该有空格！'
            raise exceptions.AuthenticationFailed(msg)

        try:
            jwt_token = auth[1]
            jwt_info = jwt.decode(jwt_token, settings.SECRET_KEY)
            userid = jwt_info.get('userid')
            try:
                # 绑定当前user到request对象上
                user = MTUser.objects.get(pk=userid)
                return user, jwt_token
```

```

except:
    msg = '用户不存在!'
    raise exceptions.AuthenticationFailed(msg)
except ExpiredSignatureError:
    msg = "JWT Token已过期!"
    raise exceptions.AuthenticationFailed(msg)

```

配置认证:

配置认证有两种方式,一种是全局的,

在 `settings.REST_FRAMEWORK.DEFAULT_AUTHENTICATION_CLASSES` 中配置。第二种就是在需要认证的视图中,通过 `authentication_classes` 进行配置。示例代码如下:

```

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES':
    [ 'apps.mtauth.authentications.JWTAuthentication' ],
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 12,
    'PAGE_QUERY_PARAM': "page"
}

```

权限:

不同的 API 拥有不同的访问权限。比如普通用户有读文章的权限,但是没有删除文章的权限。因此需要用到权限来进行 API 的管理。以下是 DRF 自带的权限。

permissions.AllowAny:

允许所有人访问。

permissions.IsAuthenticated:

是登录的用户即可访问 (判断条件是 `request.user and request.user.is_authenticated`)

permissions.IsAdminUser:

是管理员。(判断条件是 `request.user and request.user.is_staff`)

permissions.IsAuthenticatedOrReadOnly:

是登录的用户,并且这个 API 是只能读的 (也就是 `GET`、`OPTIONS`、`HEAD`)。

自定义权限:

有时候 drf 自带的权限无法满足要求,那么我们可以自定义权限。自定义权限要遵循两个条件:

1. 继承自 `permissions.BasePermission`。
2. 实现 `has_permission(self,request,view)` 或者是 `has_object_permission(self,request, view, obj)` 方法。第一个方法用管理整个视图的访问权限,第二个方法可以用来管理

某个对象的访问权限（比如只能修改自己的用户信息）。

示例代码如下：

```
from rest_framework import permissions
class IsOwnerOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.owner == request.user
```

权限的使用：

权限的使用也是两种方式。第一种是在 `settings.REST_FRAMEWORK.DEFAULT_PERMISSION_CLASSES` 设置。第二种是在具体的视图函数中通过 `permission_classes` 来设置。比如：

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ]
}

# views.py
class ExampleView(APIView):
    permission_classes = [IsAuthenticated]
    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

第七节 限速节流

使用 `drf` 可以给我们的网站 `API` 限速节流。比如一些爬虫，爬你网站的数据，那么我们可以通过设置访问的频率和次数来限制他的行为。

配置：

限速节流配置也是分成两种。第一种是直接 `在 settings.REST_FRAMEWORK 中设置`，第二种是针对每个视图函数进行设置。配置分成两个，一个是 `throttle_classes`，另外一个 `是 throttle_rates`。前者是配置不同的节流方式，后者是配置节流的策略。示例代码如下：

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ]
}
```

```

],
'DEFAULT_THROTTLE_RATES': {
    'anon': '100/day',
    'user': '1000/day'
}
}

# views.py
from rest_framework.response import Response
from rest_framework.throttling import UserRateThrottle
from rest_framework.views import APIView

class ExampleView(APIView):
    throttle_classes = [UserRateThrottle]


    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)

```

节流的类：

在 `drf` 中，节流的类总共有以下三个：

AnonRateThrottle：

针对那些没有登录的用户进行节流。默认会根据 `REMOTE_ADDR`，也就是用户的 IP 地址作为限制的标记。如果用户使用了透明代理（匿名代理没法追踪），那么在 `X-Forwarded-For` 中会保留所有的代理的 IP。比如下图： 这时候就要看 `settings.REST_FRAMEWORK.NUM_PROXIES` 了，如果这个值设置的是 0，那么那么将获取 `REMOTE_ADDR` 也就是真实的 IP 地址，如果设置的是大于 0 的数，那么将获取代理的最后一个 IP。

UserRateThrottle：

根据用户的 `id` 来作为节流的标识。可以通过两种方式设置节流的策略。

1. 在 `settings.REST_FRAMEWORK.DEFAULT_THROTTLE_RATES['user']` 设置。
2. 自定义节流类，继承自 `UserRateThrottle`，然后重写 `rate` 属性。

如果想要针对不同类型的用户实现不同策略的节流，我们可以通过继承 `UserRateThrottle` 类，然后设置 `scope` 属性，然后针对不同的 `scope` 设置不同的节流策略。比如针对管理员 `admin` 和普通用户 `normal`，可以设置不同的策略。代码如下：

```
class AdminRateThrottle(UserRateThrottle):
    scope = 'admin'

class NormalRateThrottle(UserRateThrottle):
    scope = 'normal'
```

然后在 `settings.py` 中可以如下设置：

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'example.throttles.AdminRateThrottle',
        'example.throttles.NormalRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'normal': '60/day',
        'admin': '1000/day'
    }
}
```

ScopedRateThrottle:

这个就不管是登录用户，还是没有登录的用户。都是根据 `scope` 来实现节流策略的。这个就只需要在视图中，重写 `throttle_scope` 属性，指定具体的 `scope`，然后在 `settings.py` 中进行设置。示例代码如下：

```
# views.py
class ContactListView(APIView):
    throttle_scope = 'contacts'
    ...

class ContactDetailView(APIView):
    throttle_scope = 'contacts'
    ...

class UploadView(APIView):
    throttle_scope = 'uploads'
    ...

# settings.py
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.ScopedRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'contacts': '1000/day',
        'uploads': '20/day'
    }
}
```

